Микропроцесорна
техника

Чл. 3. (1) Всяка дисциплина завършва според формите на контрол и формите на оценяване за всеки семестър по учебния план на специалността.

(2) Основни форми за контрол на знанията и уменията на студентите са семестриалното и сесийното оценяване.

(3) За дисциплини с процедура "изпит" окончателната оценка се формира като общ резултат от семестриалното и сесийното оценяване.

(4) За дисциплини с процедура "текуща оценка" окончателната оценка се формира от семестриалното оценяване (по чл.4. ал.3).

(5) Окончателната оценка на курсов проект се формира от семестриалното оценяване (по чл.4. ал.3).

(6) За дисциплини с процедура "зачита се" окончателната оценка в точки е равна на точките от семестриален контрол.

Чл. 4. (1) Окончателните оценки се оформят в цели числа по шестобалната система и се считат за положителни при постигната минимална оценка от 3.00.

(2) За формиране на междинните оценки (семестриални или сесийни) се използват точки от 0 до 100.

(3) При оформяне на окончателната оценка (в т.ч. семестриална - при дисциплини без изпит), точките се приравняват към шестобалната система както следва:

- до 49 точки - Слаб 2.00
- от 50 до 61 точки - Среден 3.00
- от 62 до 74 точки - Добър 4.00
- от 75 до 88 точки - Мн.добър 5.00
- за 89 и повече точки - Отличен 6.00

(4) Междинните оценки могат да бъдат и с по-малък брой точки от 49.

(5) При констатирана измама или опит за измама по време на изпит, студентът получава окончателна оценка по дисциплината Слаб 2.00, а при провеждане на семестриалната форма на контрол – нула точки. (Процедура за организация на семестриален контрол и оценяване на знанията и уменията на студентите)

(2) Академичната измама включва:

1. Всяко неспазване на указанията на изпитващ преподавател при каквато и да е проверка на знанията и уменията на студента или докторанта с цел поставяне на оценка или присъждане на точки по време на изпит, контролна или текуща проверка.

2. Всяко използване или опит за използване на записки, написани на материален носител, върху тялото, кратки бележки по разрешени помощни материали като таблици, речници и всякакви други конвенционални средства за преписване чрез визуализация на текст или изображение по време на изпит и всяка друга проверка на знанията и уменията с цел поставяне на оценка.

3. Използване или опит за използване на непозволени средства за комуникация, включително интернет, мобилни и други електронни и комуникационни средства за подсказване - часовници, очила и др.; чужда помощ от друг чрез гласово диктуване; откриване на непредаден мобилен телефон или друго устройство с интернет по време на изпит или защита на задания за самостоятелна работа;

4. Предаване или опит за предаване на предварително написан изпитен материал или ако такъв бъде намерен в изпитвания без да е предаден.

5. Укриване, подправяне или подмяна на изпитни материали, с цел фалшифициране на изпитните резултати.

6. Променяне, измисляне или фалшифициране на опитни или изследователски данни.

7. Продажба на писмени работи, изпитни материали, проекти, реферати, домашни, курсови работи, дипломни работи, дисертации, всякакъв вид разработки и научни трудове.

8. Ходатайства за вземане на изпити;

9. Подкупване или опит за подкупване на преподавател с материални или нематериални блага, включително всякакъв вид предимства в различни области на живота с цел вземане на изпит, получаване на точки, заверки или всякакво друго преимущество.

10. Поставяне при съпоставими условия на различни оценки.

11. Нечестно присвояване на чужди знания и представянето им като свои.

12. Предоставяне на изпитни материали, станали достъпни за служител или член на академичния състав, преди изпит или друг вид оценяване на студент, докторант или друго заинтересовано лице.

13. Явяване на изпит вместо друг.

14. Всяка намеса от трето лице с цел да повлияе върху обективното оценяване на изпитни материали или материали от конкурс по заемане на длъжност в ТУ-Варна;

15. Всяко разпространение или системно натрапване на информация за нагласи с цел да повлияние върху решение на едноличен орган или върху членове на колективен орган.

16. Всякакви други форми на нечестни прояви в разрез с академичната етика и добрите нрави.

(Чл. 2, ал. 2 на Правилника за академичен интегритет в ТУ–Варна)

VII. НЕПОЧТЕНИ И НЕЧЕСТНИ ПРОЯВИ ПО ВРЕМЕ НА ИЗПИТИ

Чл. 15.

(1) Нарушенията по тази глава се определят като:

1. **Незначително** е нарушението, при което студентът или докторантът не е спазил указания на изпитващия преподавател или квестор при провеждане на изпит, от което не са последвали други нарушения.

2. **Сравнително сериозно** е нарушението, при което студентът или докторантът бъде заловен по време на изпит със записки (пищови) написани на материален носител или върху тялото, както и кратки бележки по разрешени помощни материали като таблици, речници, ако в него се открие непредаден мобилен телефон или друго устройство с интернет - часовници, очила и др., когато този опит за академична измама е първото нарушение на студента, а за докторанта е първи изпит, съобразно индивидуалния му план на обучение.

3. **Сериозно** е нарушението по т. 2, което се извършва повторно от студент или докторант.

4. **Много сериозно е** нарушението при умишлено укриване на записки (пищови), конвенционални средства за преписване, при използване на електронни и комуникационни средства за визуализиране на текст, изображения или гласово диктуване (гласова комуникация), предварително написан изпитен материал (царски пищов).

5. **Изключително сериозно е** нарушението, при което студент или докторант се явява на изпит вместо друг студент.

Вид на нарушението	Предварително нарушение		Първо нарушение		Второ нарушение		Трето нарушение	
	Академично наказание	Административно наказание	Академично наказание	Административно наказание	Академично наказание	Административно наказание	Академично наказание	Административно наказание
Незначително (чл. 15, ал. 1, т. 1; чл. 15, ал. 1, т. 1)	Устна забележка с указание за надлежно поведение	Консултативно писмо от факултетското ръководство	Устна забележка с указание за надлежно поведение	Консултативно писмо от факултетското ръководство	Слаба оценка на изпита	Предупредително писмо от факултетското ръководство	Слаба оценка на изпита	Официално предупредително писмо от факултетското ръководство
Сравнително сериозно (чл. 15, ал. 1, т. 2; ч;. 16, ал. 1, т. 2)	Слаба оценка само за контролното	Предупредително писмо от факултетското ръководство	Слаба оценка на изпита	Предупредително писмо от факултетското ръководство	Слаба оценка на изпита и	Официално предупредително писмо от факултетското ръководство	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината през съответната уч.година	Последно предупреждение за отстраняване
Сериозно (чл. 15, ал. 1, т. 3; чл. 16, ал. 1, т. 3)	Слаба оценка само за контролното	Предупредително писмо от факултетското ръководство	Слаба оценка на изпита и нула точки за следващо явяване от семестъра по дисциплината	Официално предупредително писмо от факултетското ръководство	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината	Последно предупреждение за отстраняване	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината	Отстраняване от университета
Много сериозно (чл. 15, 1.. 1, т. 4; чл. 16, ал. 1, т. 4)	Слаба оценка по дисциплината за семестъра	Официално предупредително писмо от факултетското ръководство	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината през съответната уч.година	Последно предупреждение за отстраняване	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината през съответната уч.година	Отстраняване от университета	Отстраняване от университета	Отстраняване от университета
Изключително сериозно (чл. 15, ал. 1, т. 5; чл. 16, ал. 1, т. 5)	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината	Последно предупреждение за отстраняване	Слаба оценка на изпита без право за явяване на поправителна и годишна поправителна сесия по дисциплината през съответната уч.година	Отстраняване от университета	Отстраняване от университета	Отстраняване от университета	Отстраняване от университета	Отстраняване от университета

Ред за оценяване и провеждане на занятията и изпитите

- Максимално възможен брой контролни точки: 30 от семестриалния контрол (СК) и 70 от изпита.
- От СК – до 10 точки за активно участие и до 10 точки за всяка от двете контролни работи (задачи).
- Водещият занятието е в правото си да изисква **изключване на всички мобилни телефони (МТ)!**
- **Не се допуска видео- или фотозаснимане** нито по време на занятията, нито по време на изпита!
- На изпита могат да се получат до 30 точки за теоретичен въпрос, до 10 за кратък въпрос (отговор с едно изречение **след мислене!**) и до 30 за решена задача (сиреч, програма на **асемблерен** език).
- До изпит се допускат само хора със **заверен** семестър както за лекциите, така и **за упражненията!**
- На контролните работи и на изпита могат да се ползват: до пет книги (които ще бъдат проверени за „*пищови*“!), **отпечатани в печатница** (а не с принтер или ксерокс!), включително по старата учебна програма (в новата има промени!), таблица с команди, **тънка** химикалка и **прозрачно** шише с вода. **Никакви** други материали и предмети не се допускат! **Багажът** (дрехи, шапки, шалове, ръкавици, чанти, торби, раници, храна и др.) се оставя на **перваза** на прозореца и **не се** пипа без позволение!
- Преди СК и изпит всички МТ, таблети, плейъри и др. п. се изключват и с всички часовници, очила, слушалки, ленти, кърпи, **дебели химикалки** и др. **съмнителни** предмети **се събират** заедно. По изключение се допускат **очила с тънки рамки** (за да няма в тях видеокамера и предавател!) и **часовници** с **механични стрелки** и **без** течнокристален **екран**. И двете **уши** трябва да са **открити.**
- След началото на изпита се пази пълна **тишина**! При системен говор или шепот (след 2 забележки) или при евентуално откриване на МТ или друг забранен материал („*пищов*“) или устройство (вкл. **микрофон, камера, слушалка, примка, магнит, „бръмбар“** и др.) следва конфискация и „*двойка*“!
- В тоалетната в даден момент може да има само **един** човек, и то придружен от квестора, и то след като **предаде** изпитната си тетрадка на изпитващия (тя стои там до започване на устния му изпит).
- Краткият **устен изпит**, който се провежда с всеки студент, цели да се установи дали материалът е **разбран** и задачата е решена **самостоятелно** или е налице механично зазубряне или преписване.
- Лекциите и др. материали могат да се копират от компютрите в залата за лабораторни упражнения.
- Моля, научете фамилните имена на преподавателите си и не ги наричайте с малките им имена! ☺

„Нема лабаво!“

Скица..

„Ученикът не е съд, който трябва да бъде запълнен, а факел, който трябва да бъде запален.“ (Плутарх)

„Не можеш да научиш никого на нищо. Можеш само да му помогнеш да го открие за себе си. “ (Галилеи)

„Учителят може да отвори вратата, но ти трябва да влезеш сам.“ (Дзен)

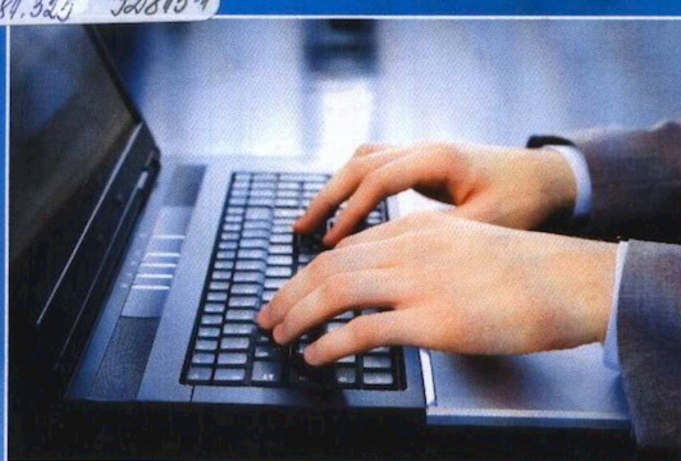
1. Въведение: Развитие на микроелектронните технологии за производство на СГИС. Кратка история на 32-битовите микропроцесори (МП) x86 и „ARM“.
2. Програмен модел: Програмно достъпни регистри в x86 и „ARM“. Флагове на регистъра за кода на условието (РКУ). Програмен модел на някои други МП.
3. Команди за работа с цели числа и системни команди на x86 и „ARM“: Групи команди. Операнди. Адресация. Неявна адресация при x86. Ортогоналност.
4. Команди за обработка на числа с плаваща запетая на x86 и „ARM“: Групи команди. Даннов формат. Стандарт „IEEE 754“. Опростени режими в „ARM“.
5. Команди тип „SIMD“ на x86 и „ARM“: Групи команди. Типове и брой данни.
6. Устройство и конвейери на x86 („P6“ и др.) и „ARM“: Блокови схеми. Работа.
7. Изключения и прекъсвания при x86 и „ARM“: Изключения. Прекъсвания – видове. Таблица на векторите. Начално установяване на МП. Режими на МП.
8. Микропроцесорен набор („chipset“): „Северен“ и „южен“ мостове. Разширен контролер за прекъсвания „APIC“. Симетрична многопроцесорност („SMP“).
9. Развитие на микропроцесорните архитектури: Развитие на МП до 64-битова архитектура. Графични процесори. Многоядреност. Перспективи. Проблеми.
10. Кратки сведения за други МП: Условни преходи и пренос в МП без РКУ („Alpha“, MIPS) и с 2 РКУ (POWER). МП с „регистров прозорец“ (SPARC). Програми „Здравей, свят!“ за различни МП и операционни системи (ОС).

Книги за 16-битови микропроцесори

Димитър Стоянов Тянев
Жейно Иванов Жейнов

МИКРОПРОЦЕСОРНА ТЕХНИКА И ПРОГРАМИРАНЕ НА АСЕМБЛЕР

Ч
42523
681.325 УД813-1



ТЕХНИЧЕСКИ УНИВЕРСИТЕТ
ВАРНА 2010

Л.ДАКОВСКИ М.МАРИНОВ К.ФИЛЬОВ С.ОВЧАРОВ

Свръх-големи интегрални схеми

ТЕХНИКА

auction.BG

ДЖОРДЖ ГОРСЛАЙН

ФАМИЛИЯ ИНТЕЛ 8086/8088

Издателство Техника

ПЕРСОНАЛНИ 16-РАЗРЕДНИ КОМПЮТРИ

ДЕЙВИД БРАДЛИ

ПРОГРАМИРАНЕ НА АСЕМБЛЕР ЗА ПЕРСОНАЛЕН КОМПЮТЪР IBM/PC

Издателство Техника

Brady

THE PETER NORTON PROGRAMMING LIBRARY

Advanced Assembly Language

Steven Holzner and
The Peter Norton Computing Group

- Over 100 ready-to-run programs
- Expert tips on TSRs, programming the mouse, 80x87 device drivers, mixed-language programming, programming for Windows, and more
- Covers Turbo Assembler and Microsoft Macro Assembler

INCLUDES A DISK

Книги за 32-битови микропроцесори

ВОЛФГАНГ ЛИНК

ПРОГРАМИРАНЕ НА АСЕМБЛЕР

8086 8088 80286 80386 80486 PENTIUM

ИЗДАТЕЛСТВО •ТЕХНИКА•

Програмиране на асемблерен език
за 32-битови микропроцесори
Intel
Част 1
А.ЕГОРОВ, Р.СТОЯНОВА
Вътрешна архитектура
Въведение в програмирането на MASM 6.1
Paraflow
Програмиране на асемблерен език
за 32-битови микропроцесори
Intel
Част 2
Р.СТОЯНОВА, А.ЕГОРОВ
Архитектура на аритметичен копроцесор
Системно програмиране на MASM 6.1

Оливер Мюлер

СПРАВОЧНИК АСЕМБЛЕР

- Основи, програмиране под DOS, Win16 и Win 32
- Обектно ориентирано програмиране под TASM
- Програмиране на математическия копроцесор
- Свързване с езиците от високо ниво C/C++ и Pascal/Delphi
- Инструкциите за централния процесор и математическия копроцесор: синтаксис в тактови цикли
- Оператори: MASM и TASM
- Функции на директивите на асемблер: MASM/ TASM и обектно ориентирани разширения на TASM

Техника

Franzis'

Всичко за Pentium

Класически CISC концепции

Обща микропроцесорна архитектура

RISC технологии

Детайлно описание на Pentium

Външна поддръжка и шинни системи

Размита логика и невронни мрежи

metropoint

NISOFT

КП

АРХИТЕКТУРА И СИСТЕМНО ПРОГРАМИРАНЕ ЗА Pentium БАЗИРАНИ КОМПЮТРИ

Р. Иванов
О. Асенов

НАРЪЧНИК

32-РАЗРЕДНИ МИКРОПРОЦЕСОРИ

Общи особености

M68000, M68010,
M68020

I80286, I80386

Z80000, NS32000,
WE32,

micro VAX 78032,
T414, V70

С. В. Рыжиков

МИКРОПРОЦЕССОРЫ 80x86, Pentium

АРХИТЕКТУРА,
ФУНКЦИОНИРОВАНИЕ,
ПРОГРАММИРОВАНИЕ,
ОПТИМИЗАЦИЯ КОДА

01. Иван Найденов, „*Общи конструкции*“, 2014 г. http://asm32.info/?page=content/2_callPower/0articles/beginers/7hll.txt
02. К. Иванов, „*Команди на целочисленото устройство от микропроцесора Intel 80486*“ (кратко описание), 2009 г. http://pmgsz.org/edu/Learn/Informatics/Refl80486.pdf
03. IA32 instruction list (short form), 2014 http://homes.di.unimi.it/~re/Corsi/SOLAB2_1314/IA32_Instruction_Set.pdf
04. Combined volume set of Intel 64 and IA-32 architectures software developer's manuals, 2019 https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
05. Intel architecture software developer's manual, vol. 2: Instruction set reference, Intel Corp, 1997 http://folk.uio.no/inf242/doc/24319101.pdf
06. The P6 architecture: Background information for developers, Intel Corp, 1995 http://people.cs.clemson.edu/~mark/330/colwell/p6arc.pdf
07. A tour of the P6 microarchitecture, Intel Corp, 1995 http://people.cs.clemson.edu/~mark/330/colwell/p6tour.pdf
08. Pentium Pro processor, Intel Corp, 1995 http://datasheets.chipdb.org/Intel/x86/Pentium%20Pro/24276903.PDF
09. Intel 440FX (82441FX PMC + 82442FX DBX), Intel Corp, 1996 http://wiki.qemu.org/images/b/bb/29054901.pdf
10. 82371FB (PIIX) & 82371SB (PIIX3), Intel Corp, 1997 http://pdf.datasheetcatalog.com/datasheet/Intel/mXvqwzr.pdf
11. 82093AA (IOAPIC), Intel Corp, 1996 http://pdos.csail.mit.edu/6.828/2018/readings/ia32/ioapic.pdf
12. NASM – the Netwide assembler, 2017 http://nasm.us/xdoc/2.14.02/nasmdoc.pdf
13. MASM programmer's guide, Microsoft Corp, 1992 http://people.sju.edu/~ggrevera/arch/references/MASM61PROGUIDE.pdf
14. Borland Turbo assembler – User's guide, Borland International, Inc, 1996 http://bitsavers.org/pdf/borland/turbo_assembler/Turbo_Assembler_Version_5_Users_Guide.pdf
15. UASM – a free MASM-compatible assembler based on JWasm, 2019 http://terraspace.co.uk/uasm.html
16. Росен Иванов, Олег Асенов, „*Архитектура и системно програмиране за Pentium-базирани компютри*“, ISBN 9549577139, Габрово, 1998 г.
17. Авторски колектив на «Нисофт», „*Всичко за Pentium*“, ISBN 9548474379, София, 1998 г.
18. Алексей Егоров, Радмила Стоянова, „*Програмиране на асемблерен език за 32-битови микропроцесори Intel*“, части I и II, София, 1997 г.
19. Волфганг Линк, „*Програмиране на асемблер*“, ISBN 9540304741, София, 1996 г.
20. Оливер Мюлер, „*Асемблер – справочник*“, ISBN 9540305942, София, 2000 г.
21. Сергей Зубков, „*Assembler для DOS, Windows и UNIX*“, ISBN 5940742599, Москва, 2004 г.
22. Виктор Юров, „*Assembler. Учебник для вузов*“, ISBN 5947235811, СПб, 2003 г.
23. Виктор Юров, „*Assembler. Практикум*“, ISBN 5947236710, СПб, 2006 г.
24. Peter Norton, „*Advanced assembly language*“, ISBN 0136587747, Brady, 1991.
25. Richard Blum, „*Professional assembly language*“, ISBN 0764579010, Wiley, 2005.
26. Randall Hyde, „*The art of assembly language*“, ISBN 1593272073, No Starch, 2010.
27. Kip Irvine, „*Assembly language for x86 processors*“, ISBN 0133769402, Pearson, 2015.
28. Sivarama Dandamudi, „*Introduction to assembly language programming*“, ISBN 0387206361, Springer, 2005.
29. John Shen, Mikko Lipasti, „*Modern processor design*“, Waveland, ISBN 1478607831, 2005.
30. Robert Colwell, „*The Pentium chronicles*“, Wiley, ISBN 0471736171, 2006.

01. Alex van Someren, Carol Atack, “*The ARM RISC Chip: A Programmer’s Guide*“, ISBN 978-0201624106, Addison Wesley, 1994.
02. The ARM Cookbook (ARM DUYI-0005B), ARM Ltd, 1994.
03. ARM 810 Preliminary Data Sheet (ARM DDI 0081E), ARM Ltd, 1996.
04. ARM Architecture Reference Manual (ARM DDI 0100B), ARM Ltd, 1996.
05. ARM Instruction Set Quick Reference Card’99 http://zap.org.au/elec2041-cdrom/reference/arm-instructions-quickref.pdf
06. VFP11 Vector Floating-Point Coprocessor Technical Reference Manual (ARM DDI 0274H), ARM Ltd, 2007.
07. Атанас Атанасов: „*Ръководство за упражнения по Микропроцесорна техника*“, ISBN 978-9544650544, София, 2012 г.
08. William Hohl, Christopher Hinds, „*ARM Assembly Language: Fundamentals and Techniques*“, ISBN 978-1482229851, CRC Press, 2014.
09. Peter Knaggs, „*ARM Assembly Language Programming*“, Trowbridge, 2016.
10. Larry Pyeatt, „*Modern Assembly Language Programming with the ARM Processor*“, ISBN 978-0128036983, Newnes, 2016.
11. Dezső Sima, „*ARM lines*“ http://users.nik.uni-obuda.hu/sima/letoltes/magyar/SZA2016_osz/nappali/ARM_processors_lecture_2016_12_07.pptx
12. ARM Datasheet, ISBN 1852500263, Acorn Computers Ltd, 1987.
13. ARM Software Development Toolkit Version 2.0: Programming Techniques (ARM DUI 0021A), ARM Ltd, 1995.
14. ARM Instruction Set Quick Reference Card (ARM QRC 0001H), ARM Ltd, 2003.
15. Andrew Sloss, Dominic Symes, Chris Wright, „*ARM System Developer’s Guide: Designing and Optimizing System Software*“, ISBN 978-1558608740, Morgan Kaufmann, 2004.
16. The ARM Instruction Set – ARM University Program – V1.0 (няма №), ARM Ltd, 2009.
17. ARM1176JZF-S Technical Reference Manual (ARM DDI 0301H), ARM Ltd, 2009.
18. ARM Cortex-A Series Programmer’s Guide (ARM DEN0013D), ARM Ltd, 2014.
19. ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition (ARM DDI 0406C.c), ARM Ltd, 2014.
20. Procedure Call Standard for the ARM Architecture (ARM IHI 0042F), ARM Ltd, 2015.
21. Stephen Furber, „*ARM System-on-Chip Architecture*“, ISBN 978-9332555570, Pearson India, 2015.

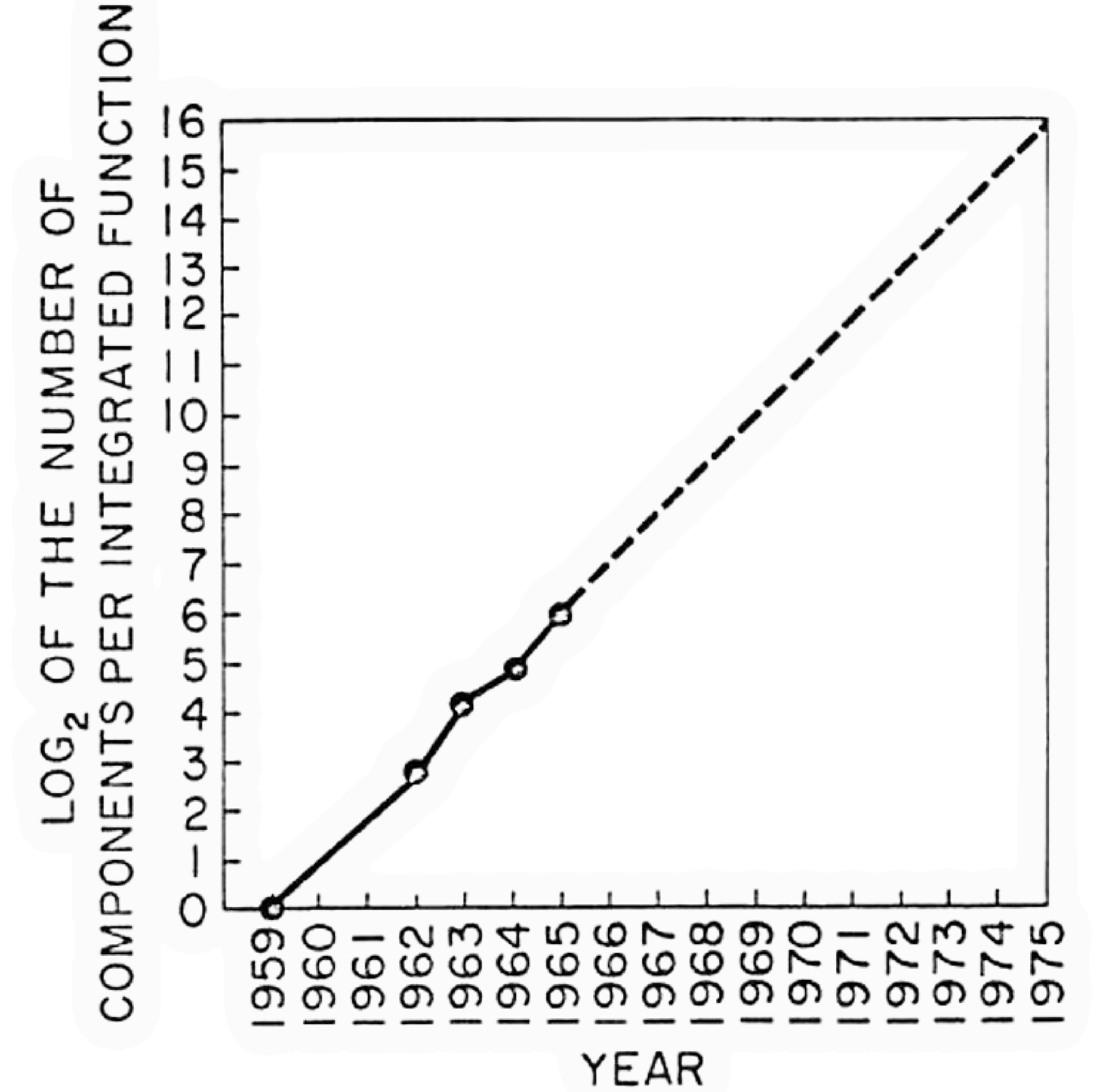
http://umis.tu-varna.bg/prep/upload/190/

Въведение: Развитие на микроелектронните технологии за производство на СГИС. Кратка история на 32-битовите микропроцесори (МП) x86 и „ARM“.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

FIGURE 1.10 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2013 extrapolated by the authors. See Section 1.12.

Gordon Earle Moore (born in 1929)


LOG$_2$ OF THE NUMBER OF
COMPONENTS PER INTEGRATED FUNCTION
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
YEAR

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

Transistor count

50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000

1970 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018

72-core Xeon Phi
Centriq 2400
GC2 IPU
SPARC M7
32-core AMD Epyc
IBM z13 Storage Controller
Apple A12X Bionic
18-core Xeon Haswell-E5
Tegra Xavier SoC
Qualcomm Snapdragon 8cx/SCX8180
Xbox One main SoC
HiSilicon Kirin 980 + Apple A12 Bionic
61-core Xeon Phi
HiSilicon Kirin 710
12-core POWER8
10-core Core i7 Broadwell-E
8-core Xeon Nehalem-EX
Qualcomm Snapdragon 835
Six-core Xeon 7400
Dual-core + GPU Iris Core i7 Broadwell-U
Dual-core Itanium 2
Quad-core + GPU GT2 Core i7 Skylake K
Quad-core + GPU Core i7 Haswell
Pentium D Presler
POWER6
Apple A7 (dual-core ARM64 "mobile SoC")
Itanium 2 with 9 MB cache
Core i7 (Quad)
AMD K10 quad-core 2M L3
Itanium 2 Madison 6M
Core 2 Duo Wolfdale
Pentium D Smithfield
Core 2 Duo Conroe
Itanium 2 McKinley
Cell
Core 2 Duo Wolfdale 3M
Pentium 4 Prescott-2M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
AMD K8
Pentium 4 Prescott
Pentium 4 Northwood
Barton
Atom
Pentium 4 Willamette
Pentium III Tualatin
Pentium II Mobile Dixon
ARM Cortex-A9
AMD K7
Pentium III Coppermine
AMD K6-III
AMD K6
Pentium III Katmai
Pentium II Deschutes
Pentium Pro
Pentium II
Klamath
Pentium
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Intel 80286
ARM 9TDMI
Motorola 68000
Intel 80186
Intel 8086
Intel 8088
ARM 2
ARM 6
ARM 1
WDC 65C816
Novix NC4016
Motorola 6809
TMS 1000
Zilog Z80
WDC 65C02
RCA 1802
Intel 8085
Intel 8080
Intel 8008
MOS Technology 6502
Motorola 6800
Intel 4004

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

COLOSSUS GC2

The world's most complex processor chip with 23.6 billion transistors

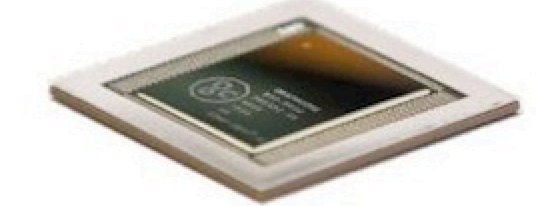
10x IPU-LINKS™
320GB/s chip-to-chip bandwidth

300MB IN-PROCESSOR-MEMORY™
45TB/s memory bandwidth per chip
the whole model held inside the processor

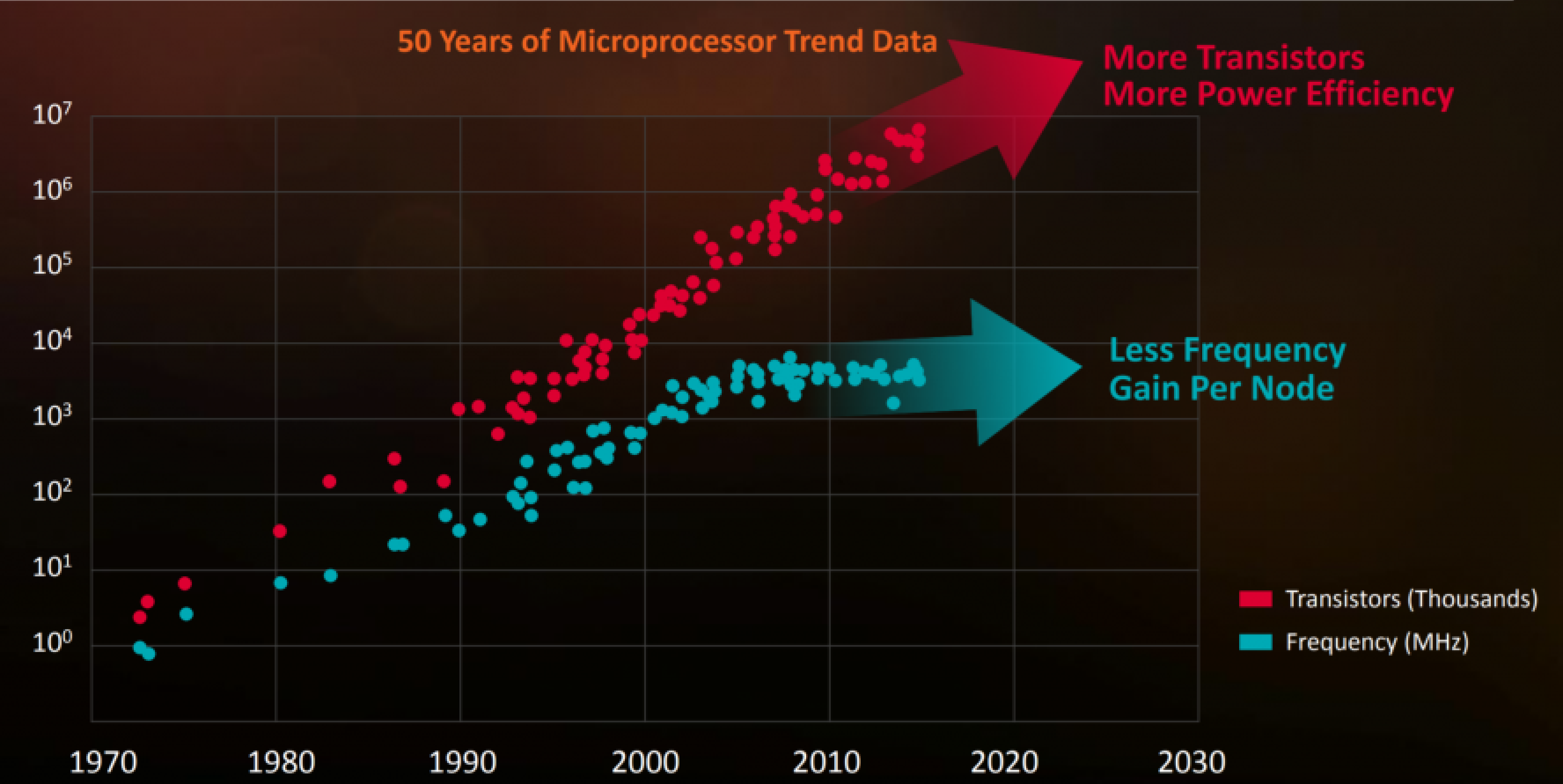
PCIe Gen4 x16
64GB/s bi-directional host
communication bandwidth

1,216 independent IPU-CORES™
each with IN-PROCESSOR-MEMORY™ tile
> 100GFLOPS per IPU-CORE™
> 7,000 programs executing in parallel

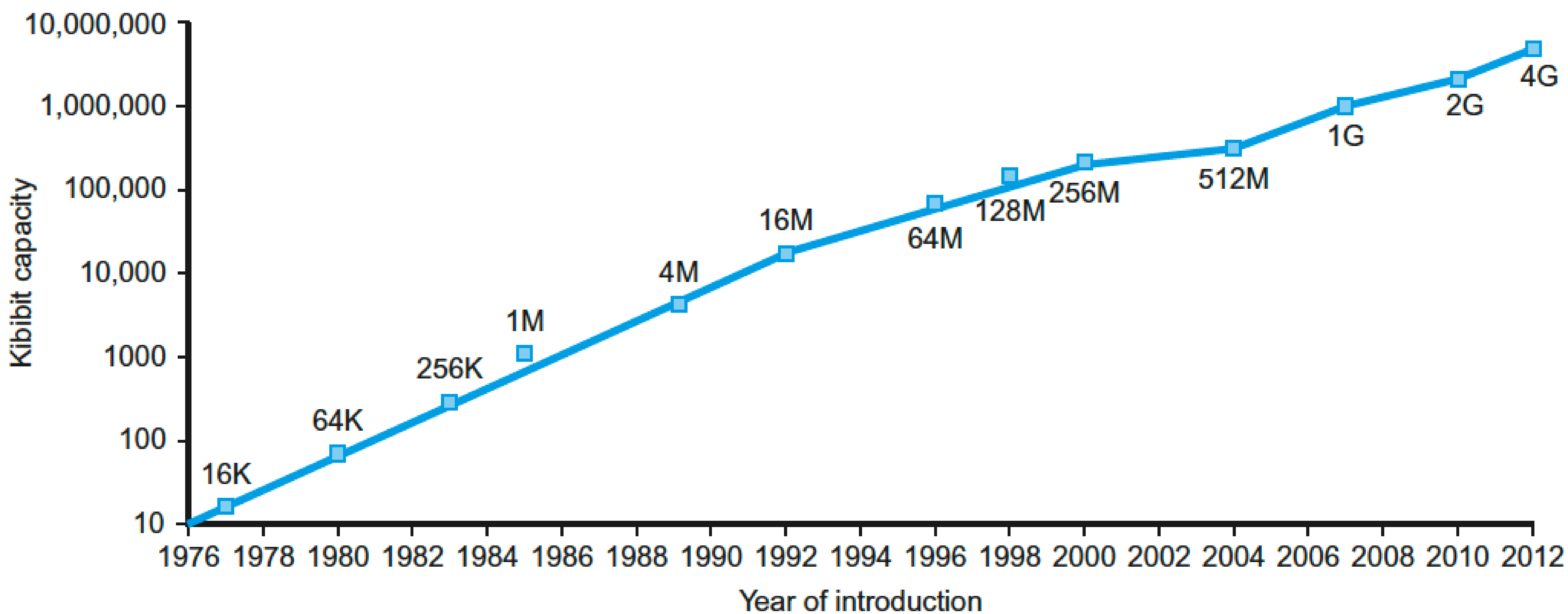
8TB/s all to all IPU-EXCHANGE™
non-blocking, any communication pattern



MOORE'S LAW IS SLOWING



See Endnotes


10,000,000
1,000,000
100,000
10,000
1000
100
10
Kibibit capacity
16K
64K
256K
1M
4M
16M
64M
128M
256M
512M
1G
2G
4G
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
Year of introduction

DIFFERENT FOUNDRIES AND THEIR CAPACITY IN 2015

Capacity in Kilo Wafer / Month 200mm   Capacity in 200mm equivalent   Source: Companies, IC Insight

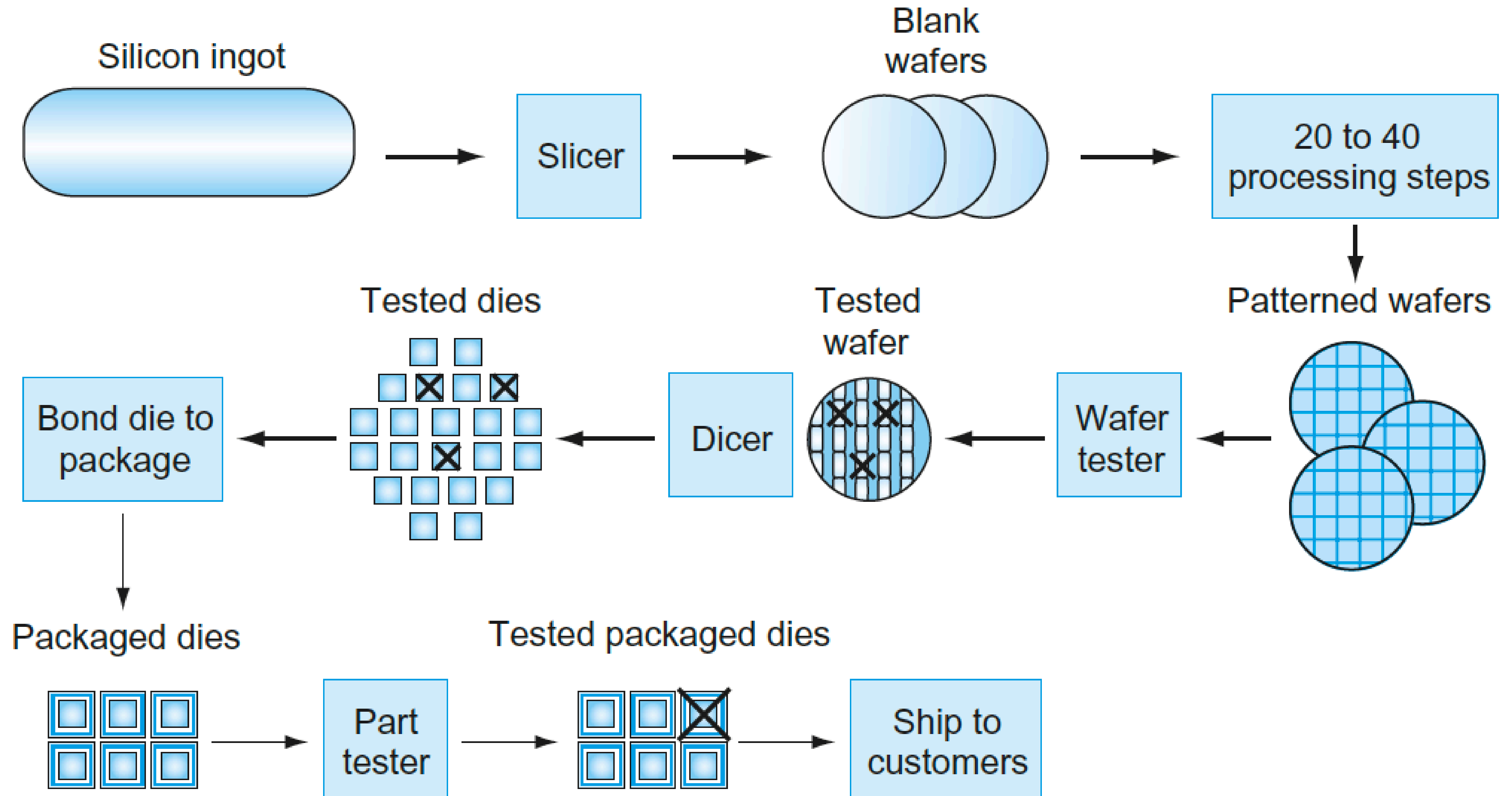
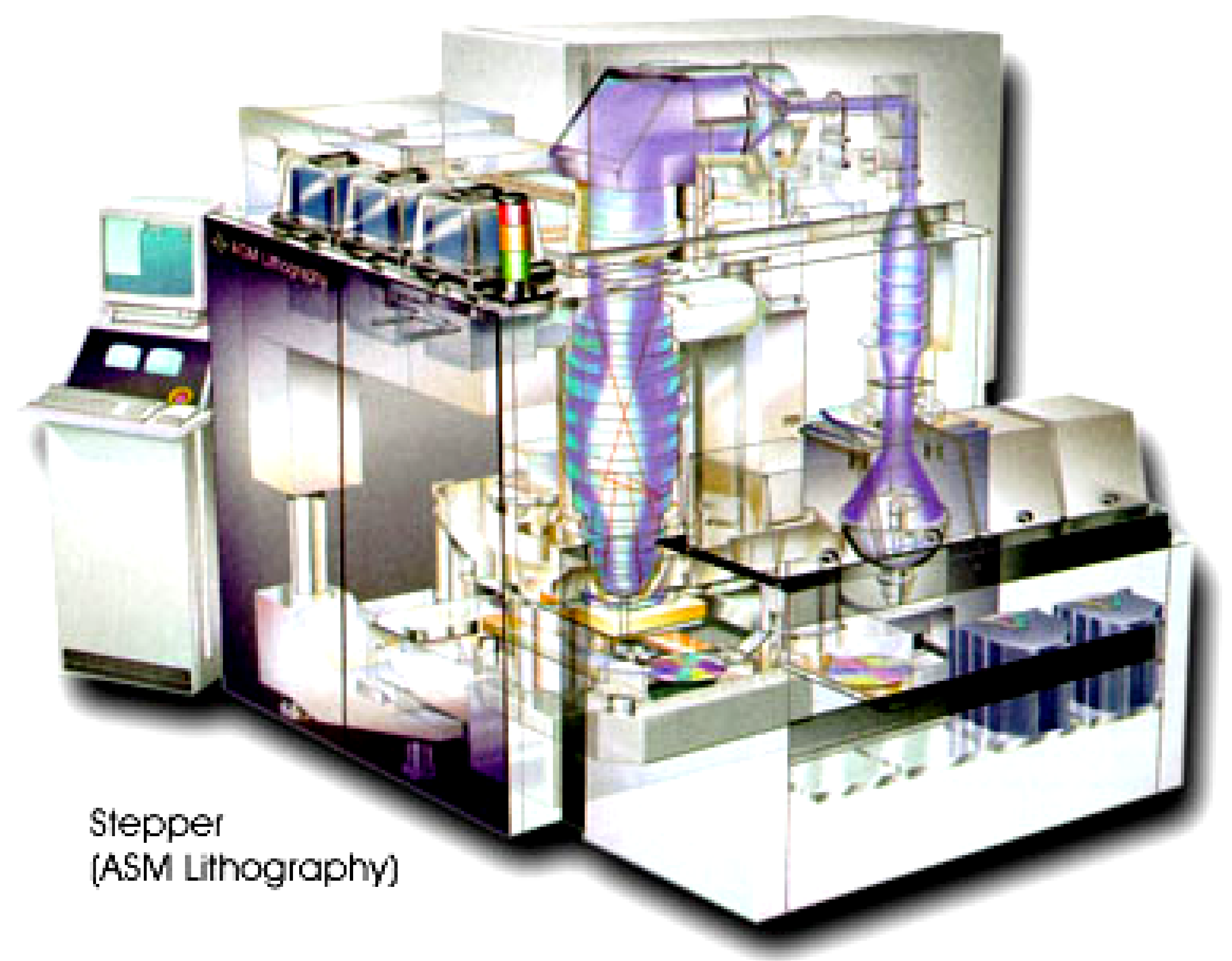


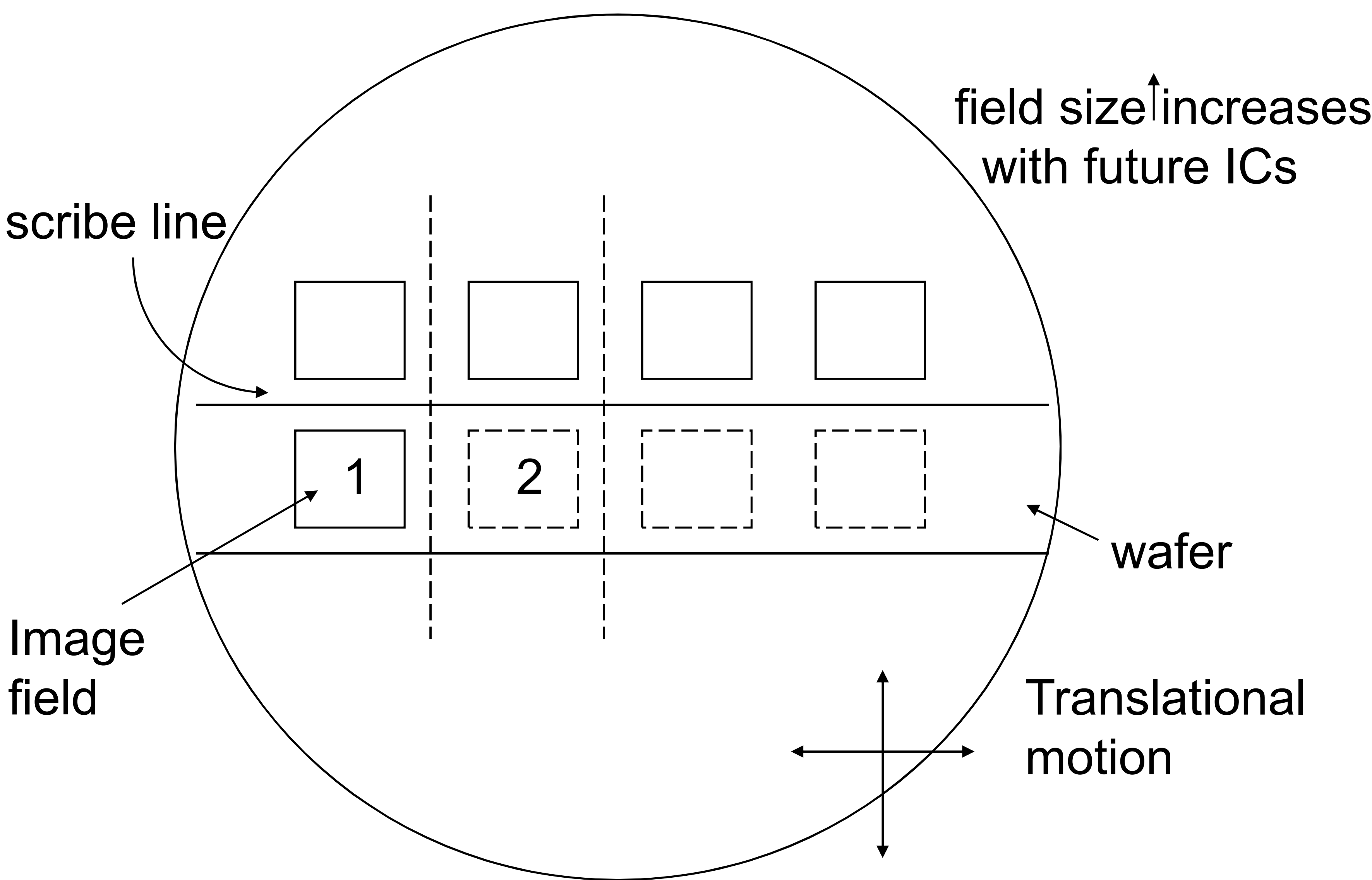
FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Next, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (X means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

Big buildings fo
tiny chips
An Intel 300mm automated fab is over 70 feet tall. Three of the fab's four levels support the cleanroom level, the only place where actual chip production occurs.
Fan deck
The fan deck houses systems that keep the air in the cleanroom particle-free and precisely maintain the right temperature and humidity for production. The fan deck is the tallest level of the fab; people generally don't work here.
Cleanroom level
Several football fields could fit inside a 300mm fab cleanroom, where chip production takes place. An automated transport system carries wafers from station to station. Purified air enters through the ceiling and exits through floor tiles like those you are standing on. Cleanroom workers wear bunny suits to keep lint, hair, and skin flakes off the wafers.
Clean subfab level
The clean subfab contains pumps, transformers, power cabinets, and other systems that support the cleanroom. Large pipes called "laterals" carry gases, liquids, wastes, and exhausts to and from production tools. Workers don't wear bunny suits here, but they do wear hard hats, safety glasses, gloves, and shoe covers because air from the cleanroom flows through the subfab.
Utility level
Electrical panels that support the fab are located here, along with the "mains"— large utility pipes and ductwork that feed up to the lateral pipes in the clean subfab. Also here are systems such as chillers and compressors. Workers who monitor the equipment on this level wear street clothes, hard hats, and safety glasses.

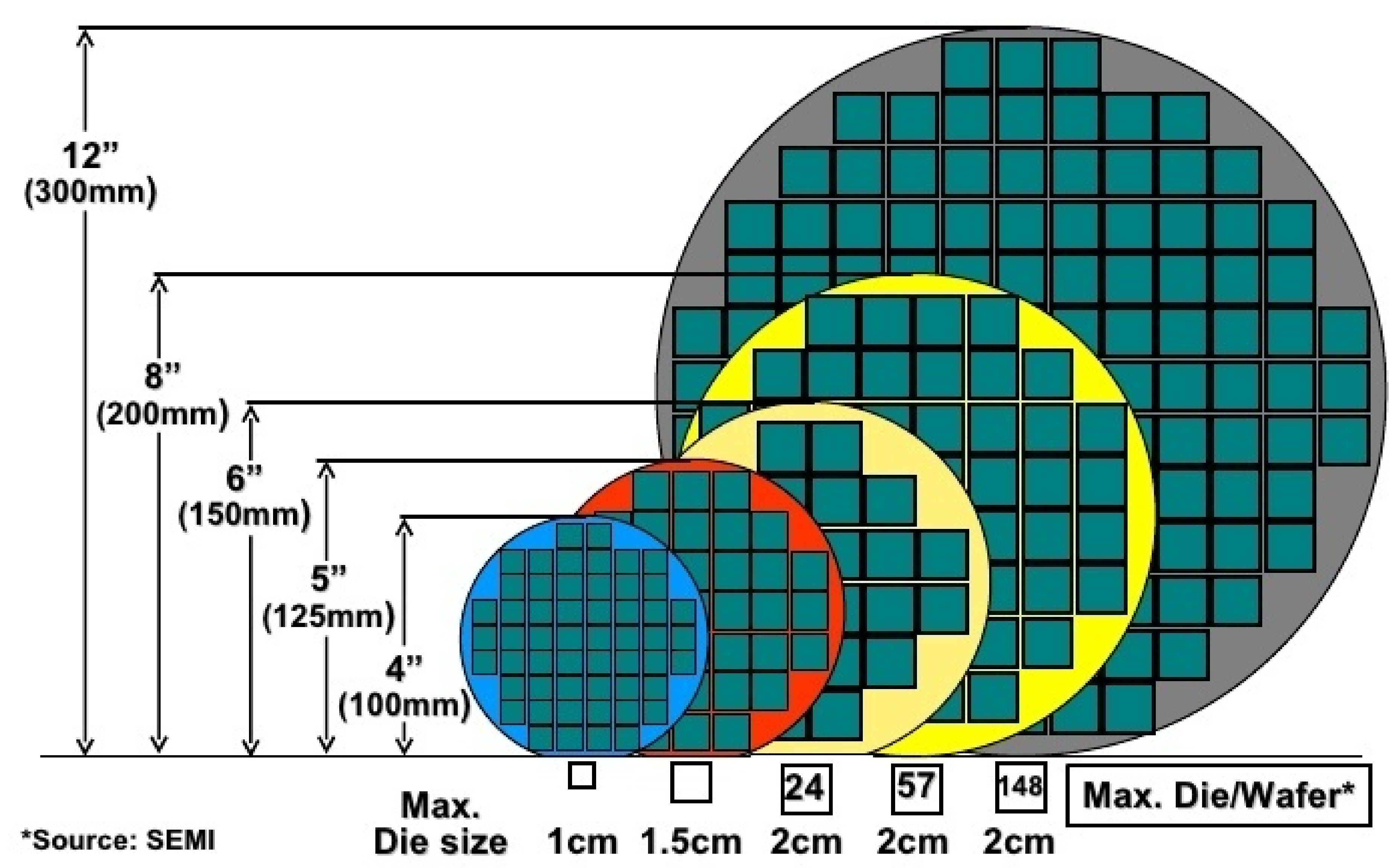
Excimer Laser Stepper

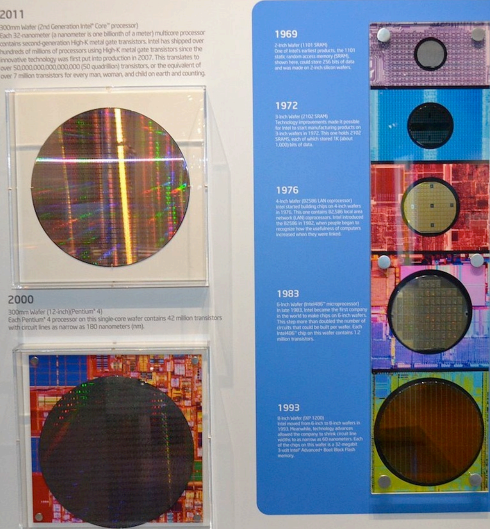


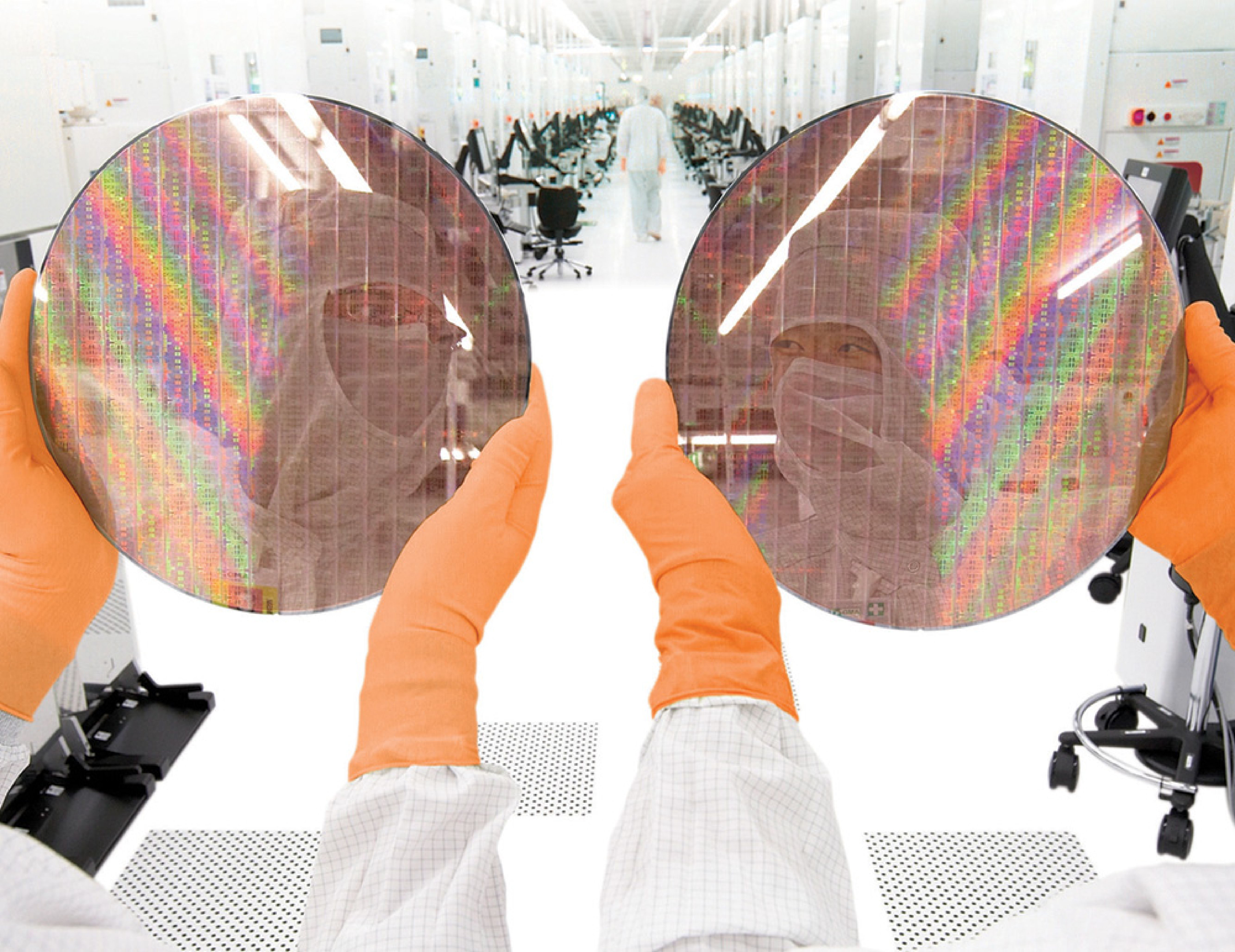
Optical Stepper

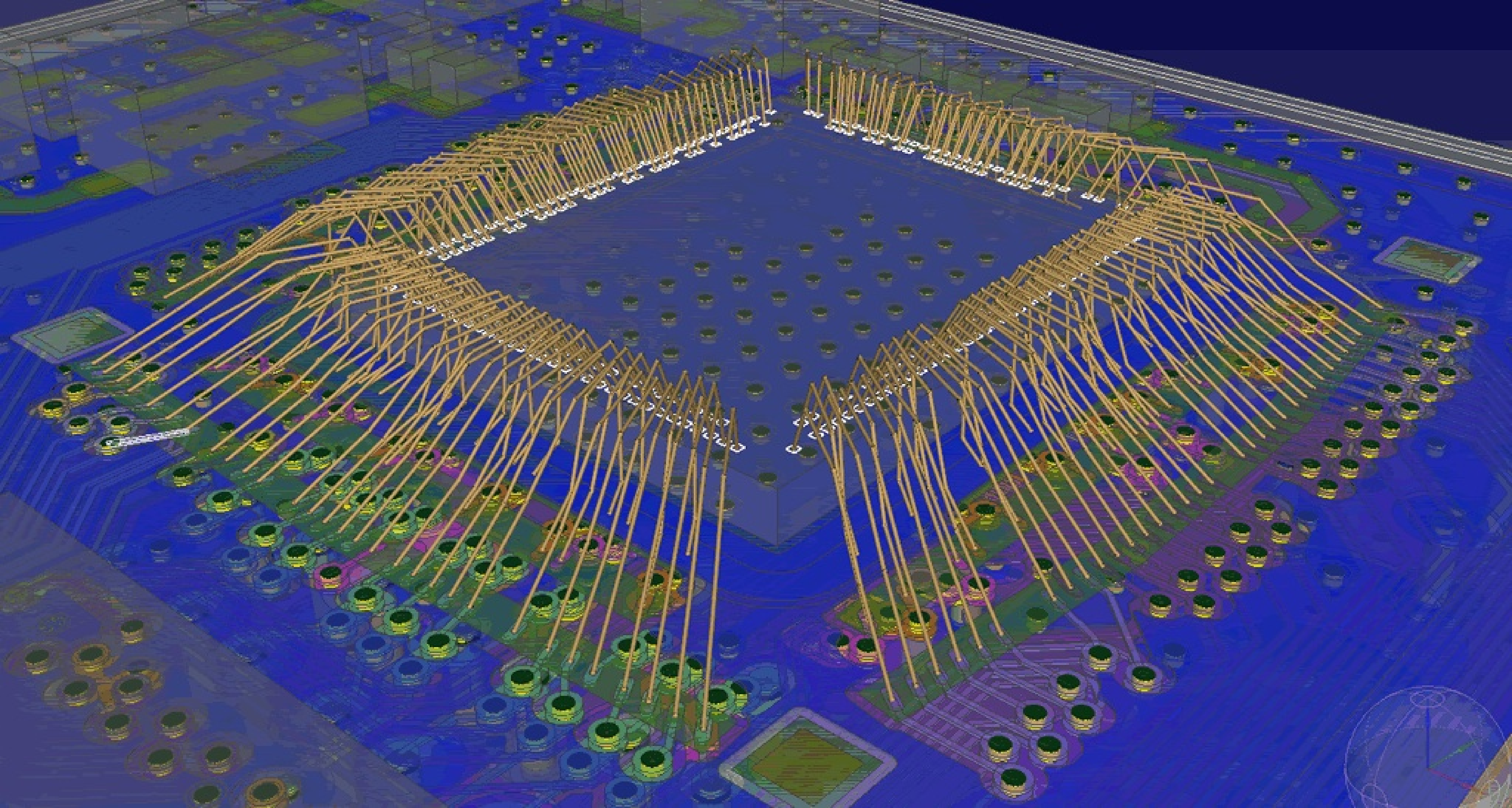


Die per Wafer




2011
300mm Wafer (2nd Generation Intel® Core™ processor)
Each 32-nanometer (a nanometer is one billionth of a meter) multicore processor contains second-generation High-K metal gate transistors. Intel has shipped over hundreds of millions of processors using High-K metal gate transistors since the innovative technology was first put into production in 2007. This translates to over 50,000,000,000,000,000 (50 quadrillion) transistors, or the equivalent of over 7 million transistors for every man, woman, and child on earth and counting.
2000
300mm Wafer (12-inch)(Pentium® 4)
Each Pentium® 4 processor on this single-core wafer contains 42 million transistors with circuit lines as narrow as 180 nanometers (nm).
1969
2-inch Wafer (1101 SRAM)
One of Intel's earliest products, the 1101 static random access memory (SRAM), shown here, could store 256 bits of data and was made on 2-inch silicon wafers.
1972
3-inch Wafer (2102 SRAM)
Technology improvements made it possible for Intel to start manufacturing products on 3-inch wafers in 1972. This one holds 2102 SRAMS, each of which stored 1K (about 1,000) bits of data.
1976
4-inch Wafer (82586 LAN coprocessor)
Intel started building chips on 4-inch wafers in 1976. This one contains 82,586 local area network (LAN) coprocessors. Intel introduced the 82586 in 1982, when people began to recognize how the usefulness of computers increased when they were linked.
1983
6-inch Wafer (Intel486™ microprocessor)
In late 1983, Intel became the first company in the world to make chips on 6-inch wafers. This step more than doubled the number of circuits that could be built per wafer. Each Intel486™ chip on this wafer contains 1.2 million transistors.
1993
8-inch Wafer (IXP 1200)
Intel moved from 6-inch to 8-inch wafers in 1993. Meanwhile, technology advances allowed the company to shrink circuit line widths to as narrow as 60 nanometers. Each of the chips on this wafer is a 32-megabit 3-volt Intel® Advanced+ Boot Block Flash memory.

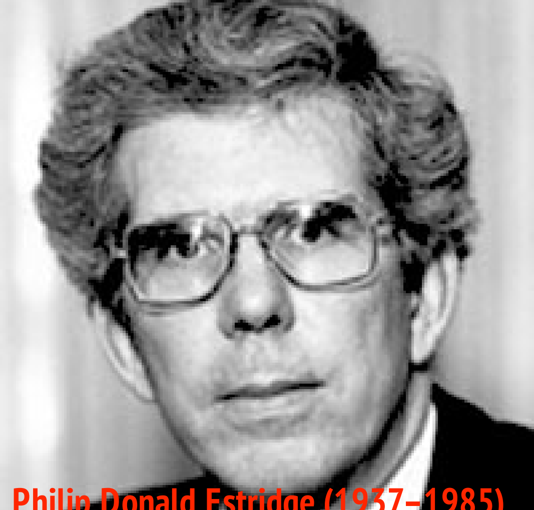




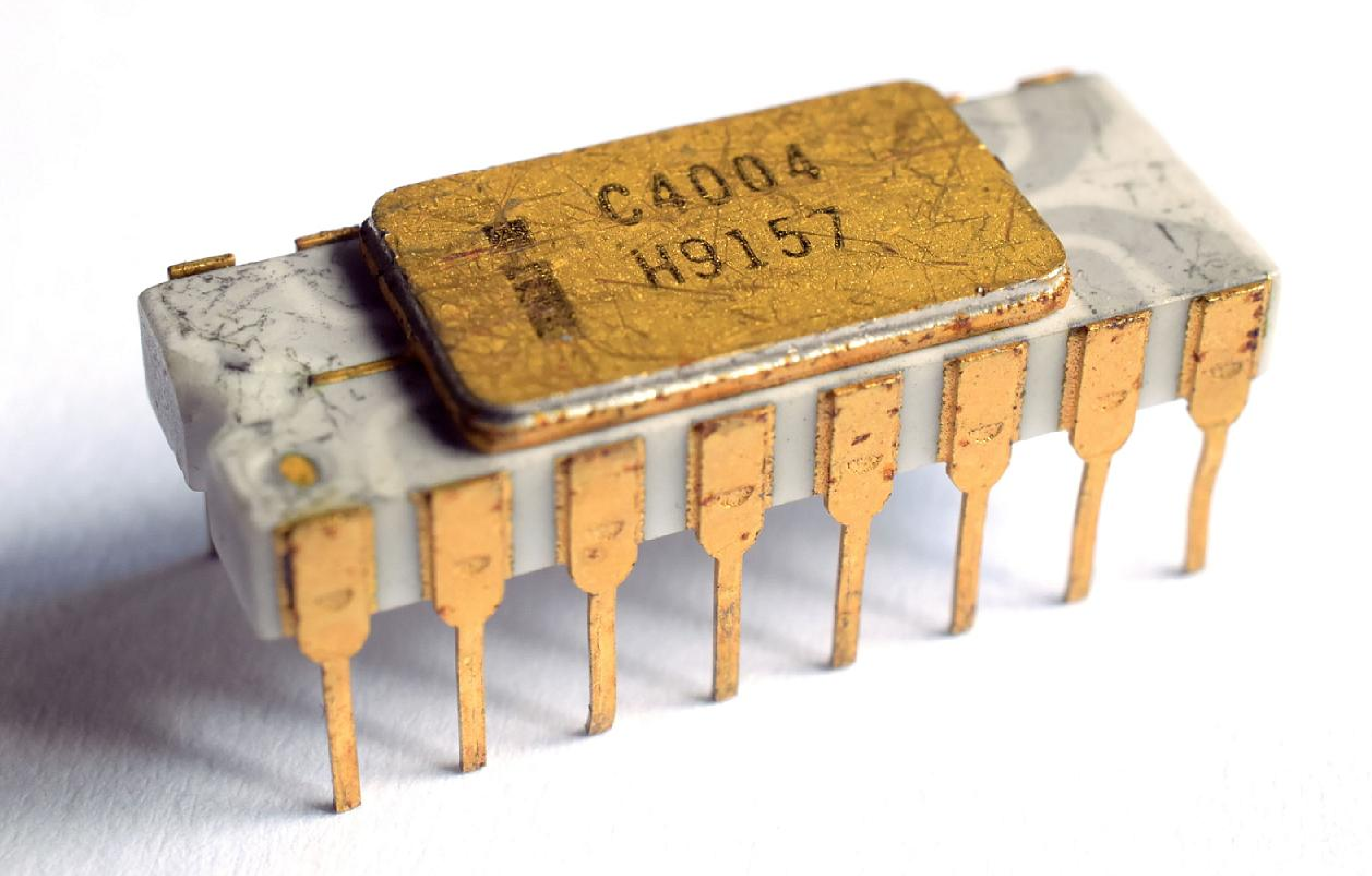
Microprocessor Evolution

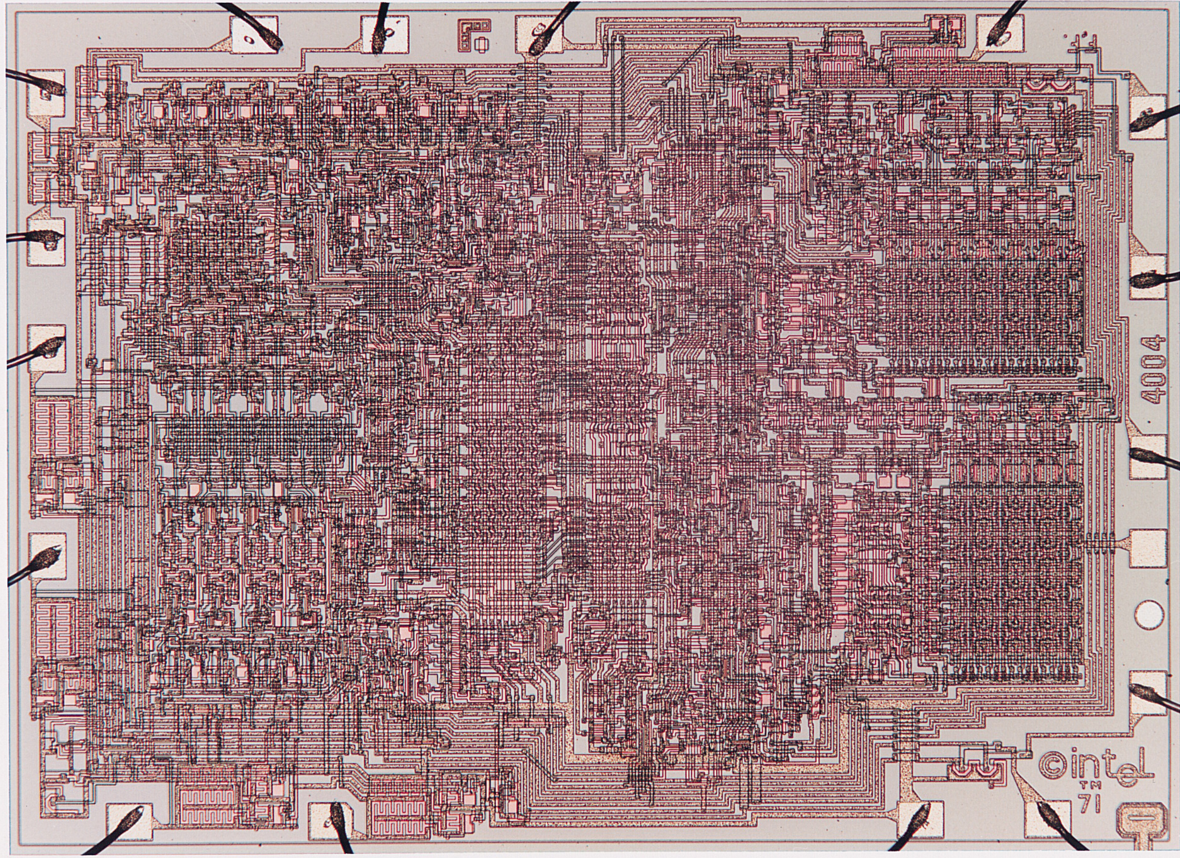
- Rapid progress in 1970s, fueled by advances in MOS technology, imitated minicomputers and mainframe ISAs
- "Microprocessor Wars": compete by adding instructions (easy for microcode), justified given assembly language programming
- Intel iAPX 432: Most ambitious 1970s micro, started in 1975
  - 32-bit capability-based, object-oriented architecture, custom OS written in Ada
  - Severe performance, complexity (multiple chips), and usability problems; announced 1981
- Intel 8086 (1978, 8MHz, 29,000 transistors)
  - "Stopgap" 16-bit processor, 52 weeks to new chip
  - ISA architected in 3 weeks (10 person weeks) assembly-compatible with 8 bit 8080
- IBM PC 1981 picks Intel 8088 for 8-bit bus (and Motorola 68000 was late)
- Estimated PC sales: 250,000
- Actual PC sales: 100,000,000 ⇒ 8086 "overnight" success
- Binary compatibility of PC software ⇒ bright future for 8086

SBIC
SWITCHING POWER SUPPLY
CAUTION!
ATTENTION! ZONE DANGEREUSE
SB-150W

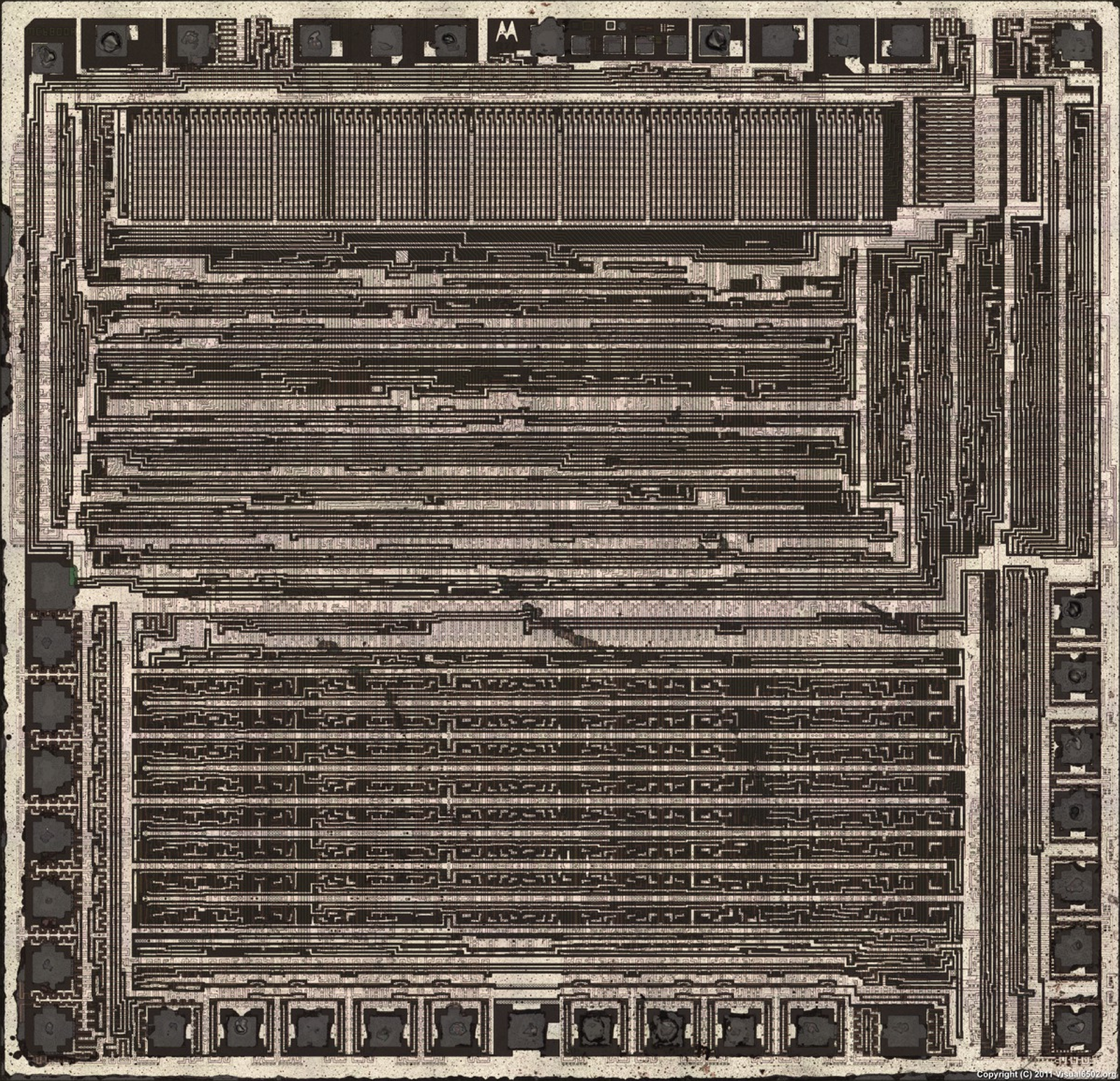


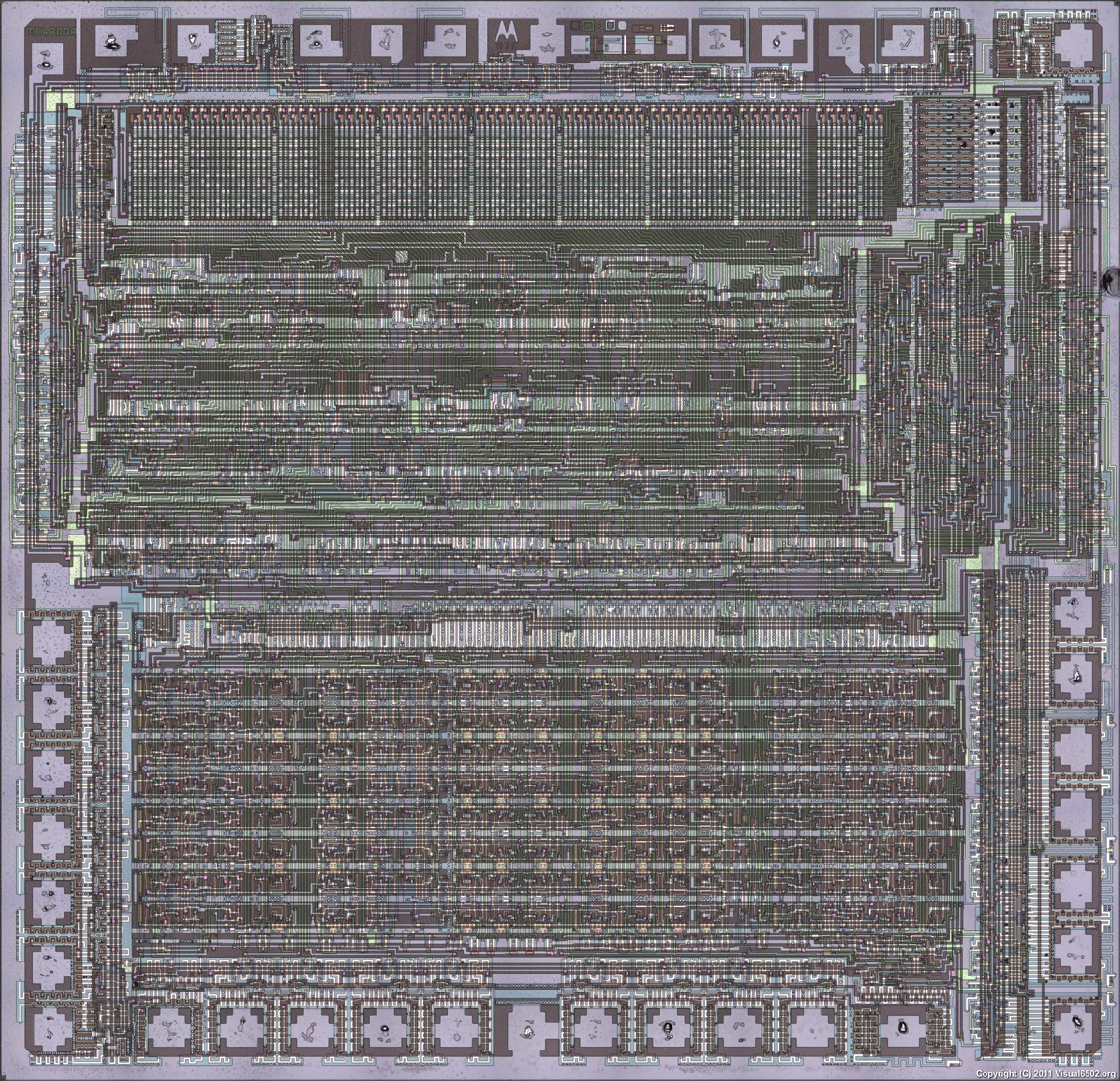
Philip Donald Estridge (1937–1985)

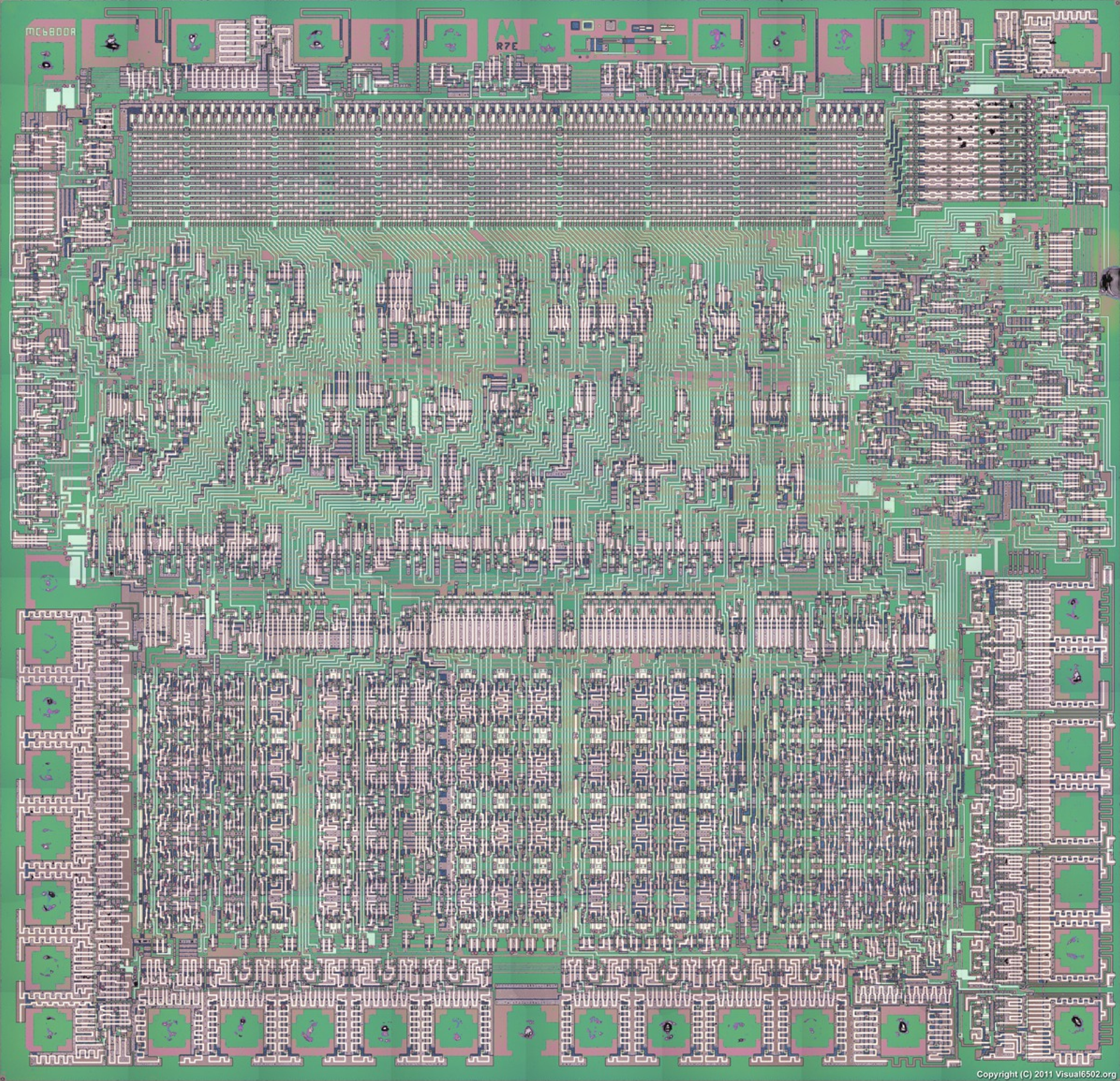

C4004
H9157

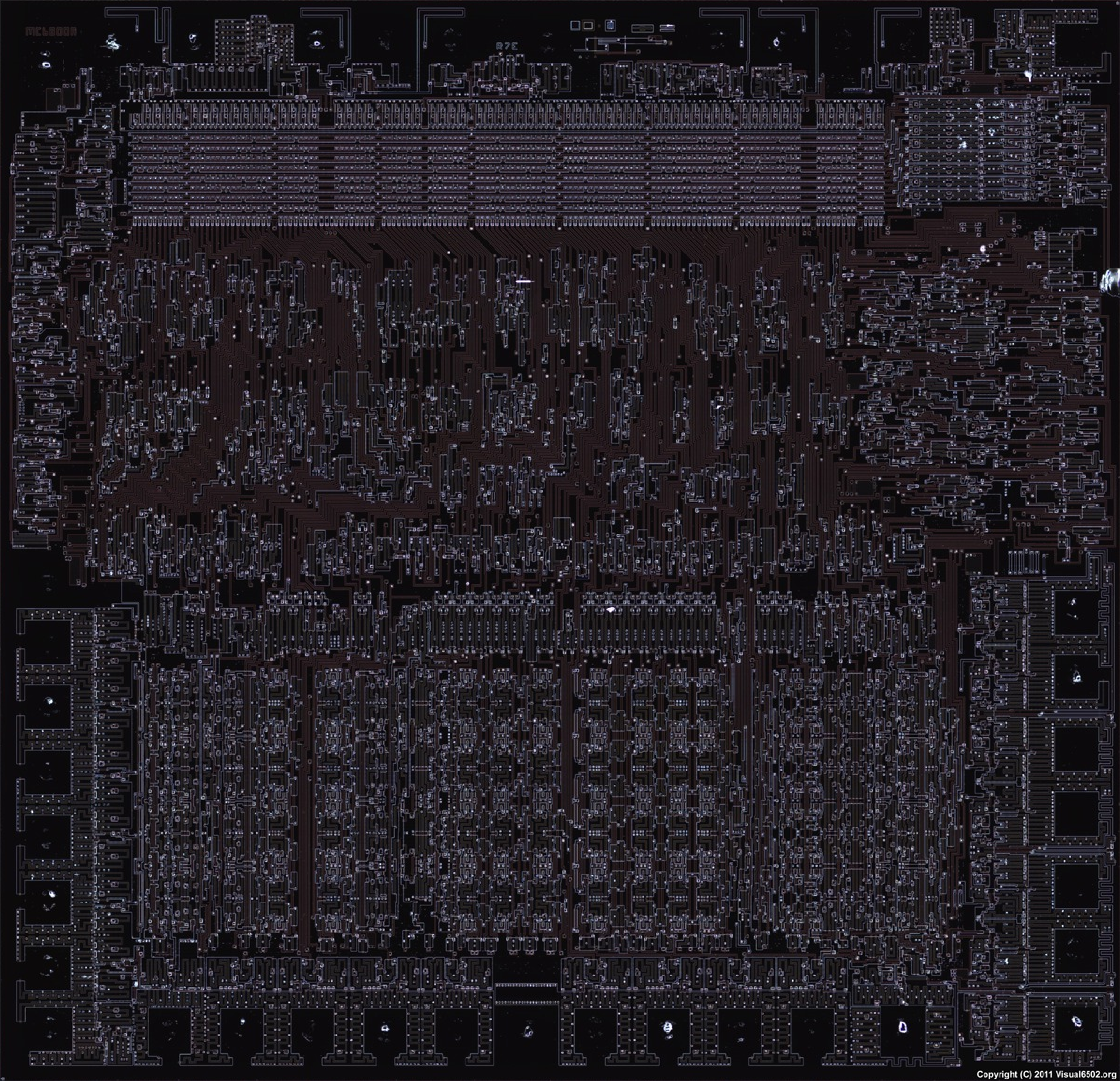

4004
©intel
TM
71

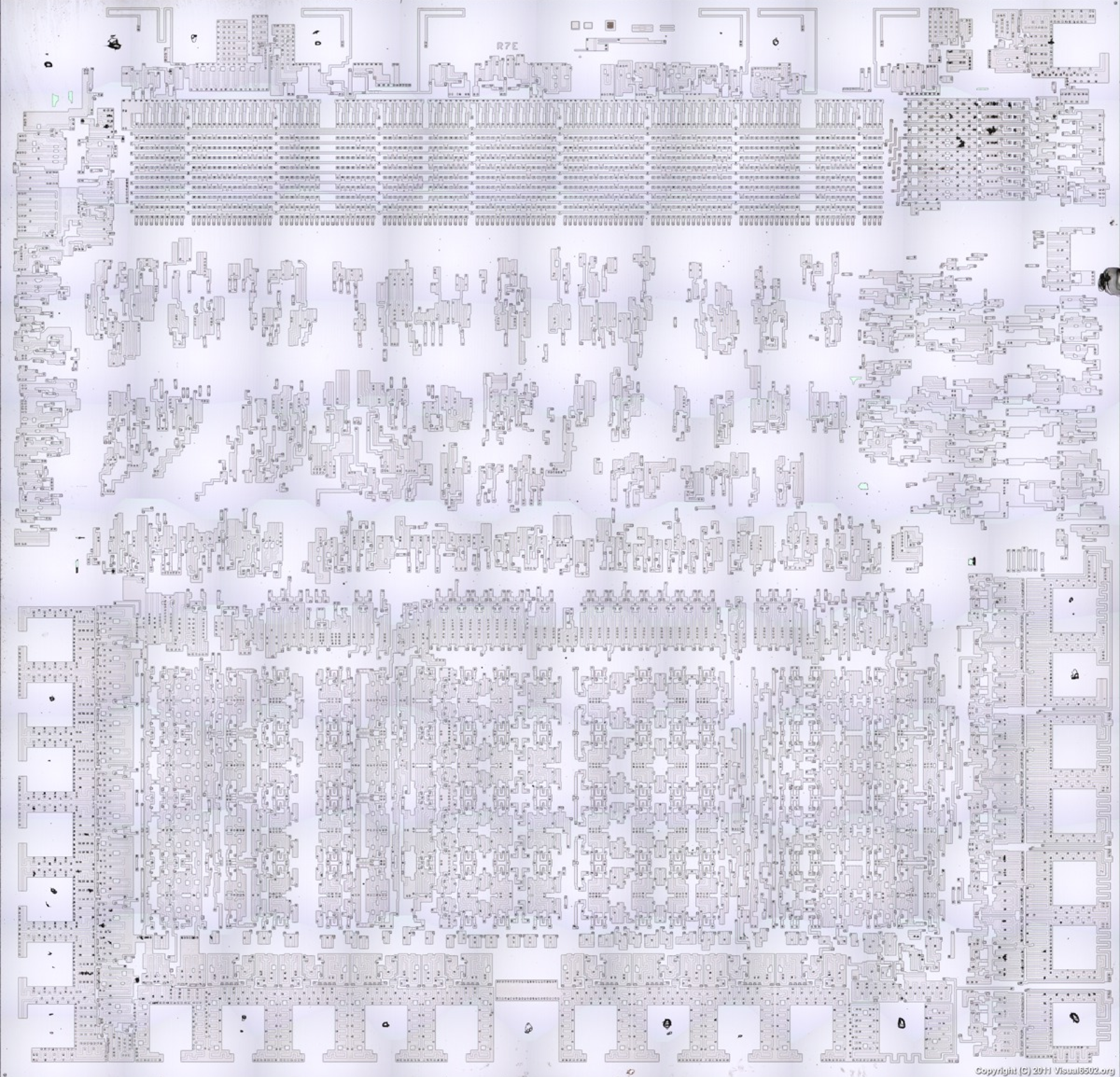
CM 601
1
CM 601
81 26
1
CM601
8608
CM601
8352


Copyright (C) 2011 Visual6502.org

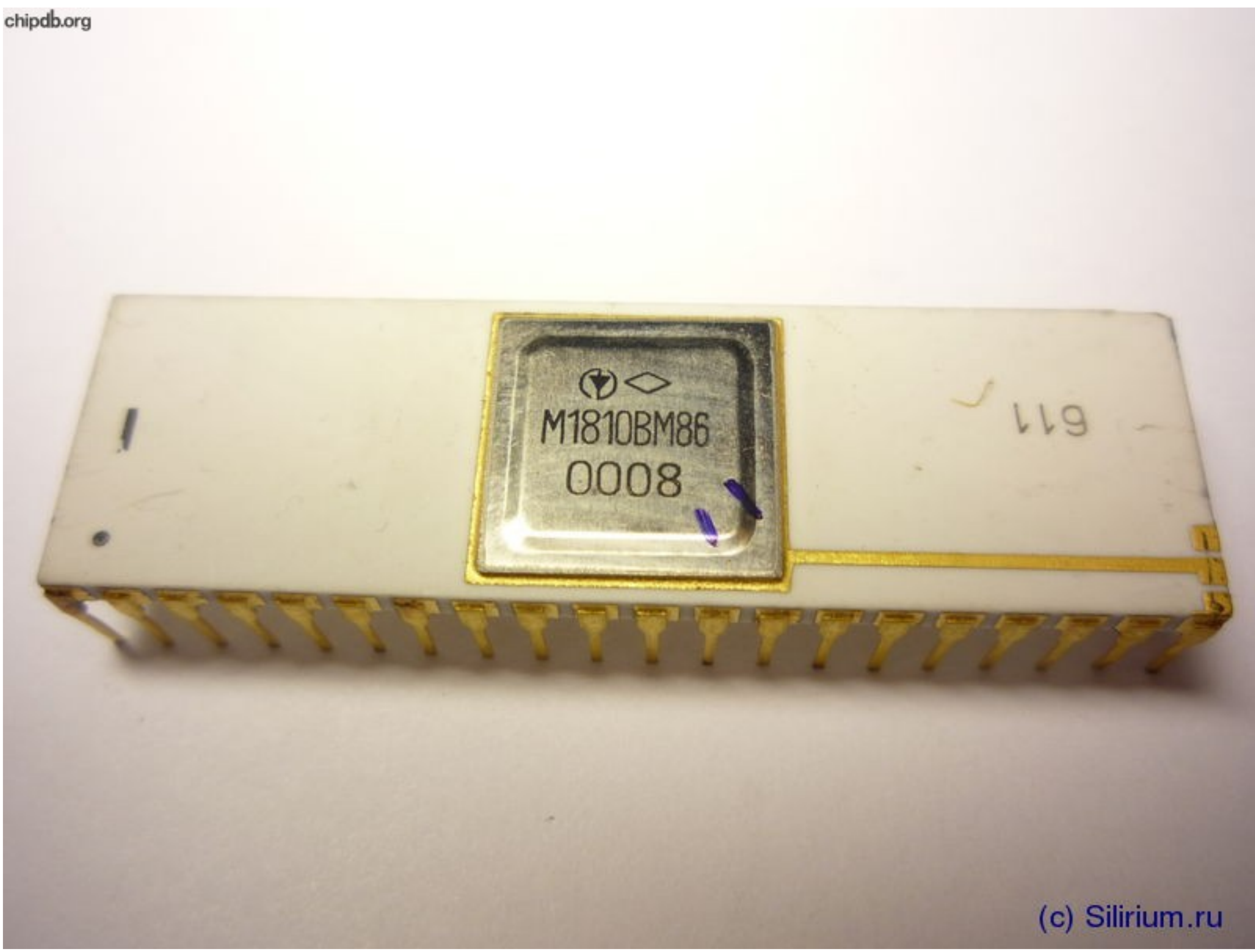

MC6800A
Copyright (C) 2011 Visual6502.org


MC6800A
R7E
Copyright (C) 2011 Visual6502.org


MC6800A
R7E
Copyright (C) 2011 Visual6502.org

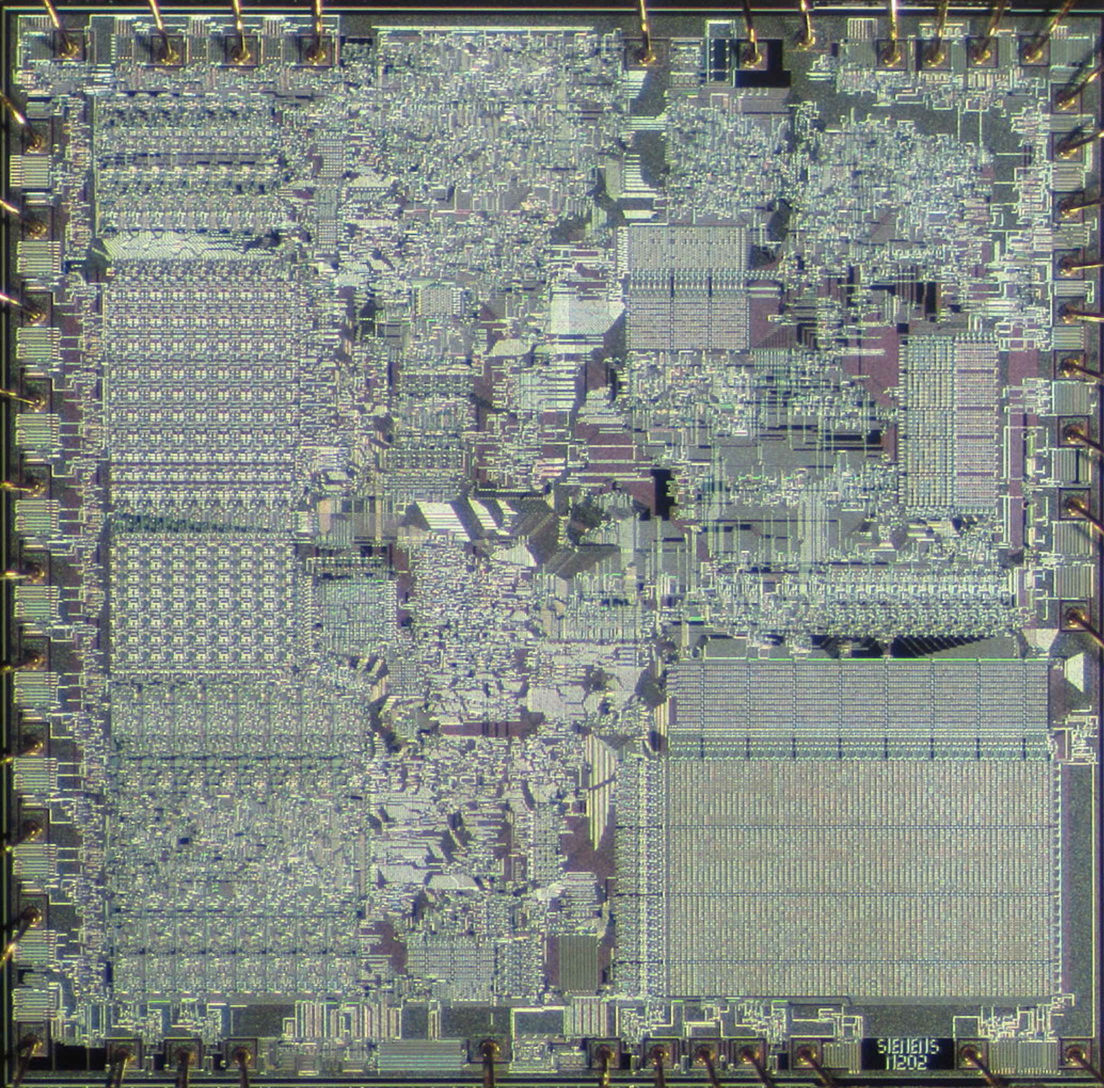

R7E
Copyright (C) 2011 Visual6502.org

CM601P
VIKIWAT
WWW.VIKIWAT.COM

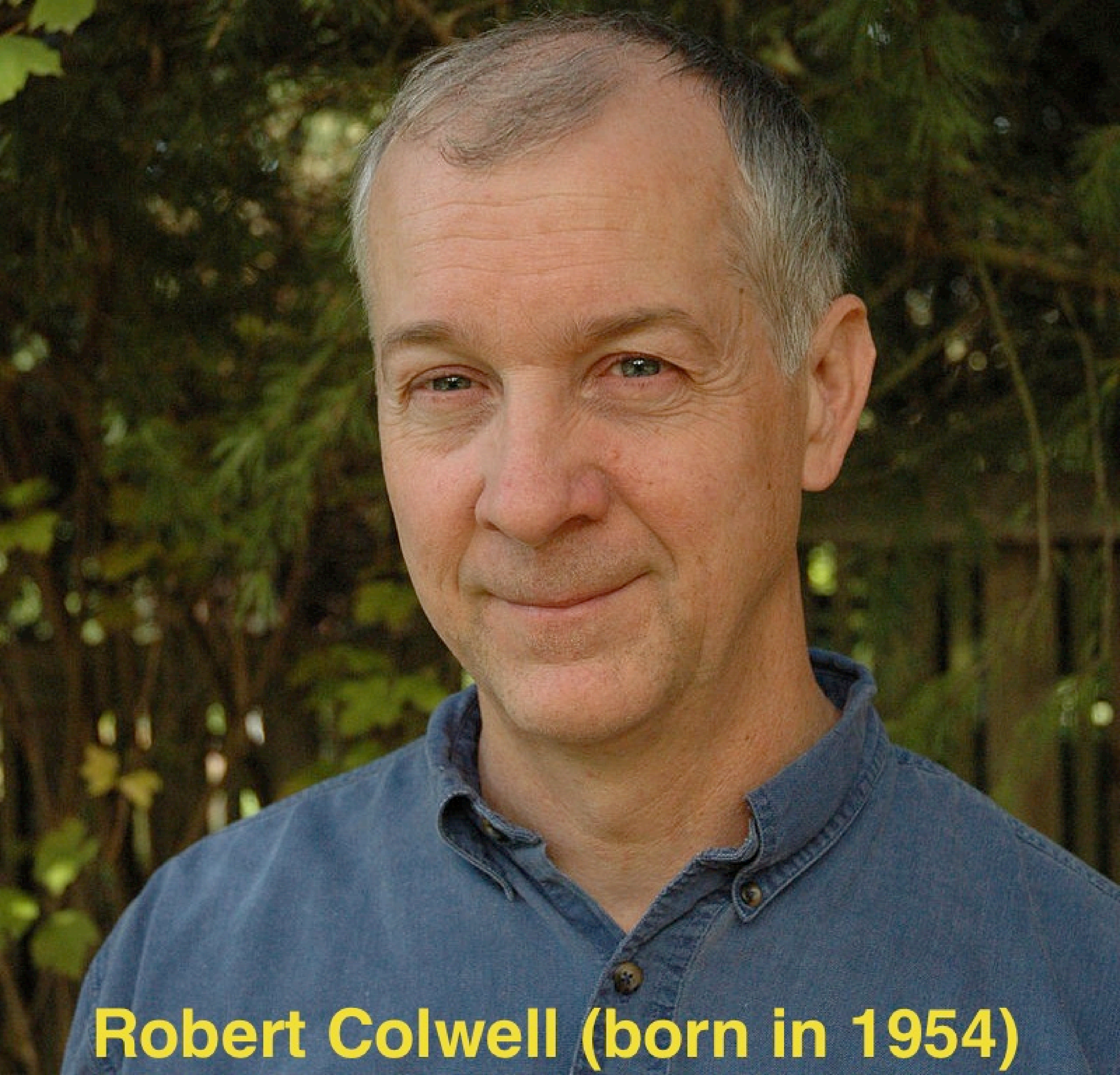

chipdb.org
М1810ВМ86
0008
(c) Silirium.ru

ОСНОВНЫЕ ЭЛЕКТРИЧЕСКИЕ ПАРАМЕТРЫ в диапазоне температур от минус 60 °С до 85 °С

Наименование параметра, единица измерения	Буквенное обозначение	Норма	
		не менее	не более
I. Выходное напряжение высокого уровня, В (I_{OH} = –0,4 мА)	U_{OH}	2,4	–
2. Выходное напряжение низкого уровня, В (I_{OL} = 2,0 мА)	U_{OL}	–	0,45
3. Ток потребления, мА	I_{CC}		360
4. Ток утечки на входах, мкА	I_{LI}		$\vert\pm$I0$\vert$
5. Выходной ток в состоянии "Выключено", мкА	I_{OZ}		$\vert\pm$I0$\vert$
6. Входная емкость, пФ	C_I		I5
7. Емкость входа/выхода, пФ	$C_{I/O}$		


SIEMENS
11202

Stephen Paul Morse (born in 1940)


Robert Colwell (born in 1954)

Cenozoic
First horses appear; Intel adds MMX, SSE, SSE2
Mesozoic
Abundant dinosaurs; Intel extends registers to 32 bits
Paleozoic
First winged insects; Intel adds segments, 16-bit code

Intel Processor	Date Intro-duced	Max. Clock Frequency at Intro-duction	Transis-tors per Die	Register Sizes[1]	Ext. Data Bus Size[2]	Max. Extern. Addr. Space	Caches
8086	1978	8 MHz	29 K	16 GP	16	1 MB	None
Intel 286	1982	12.5 MHz	134 K	16 GP	16	16 MB	Note 3
Intel386 DX Processor	1985	20 MHz	275 K	32 GP	32	4 GB	Note 3
Intel486 DX Processor	1989	25 MHz	1.2 M	32 GP 80 FPU	32	4 GB	L1: 8KB
Pentium Processor	1993	60 MHz	3.1 M	32 GP 80 FPU	64	4 GB	L1:16KB
Pentium Pro Processor	1995	200 MHz	5.5 M	32 GP 80 FPU	64	64 GB	L1: 16KB L2: 256KB or 512KB
Pentium II Processor	1997	266 MHz	7 M	32 GP 80 FPU 64 MMX	64	64 GB	L1: 32KB L2: 256KB or 512KB
Pentium III Processor	1999	500 MHz	8.2 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 32KB L2: 512KB

Figure 1.16 shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 years. Both clock rate and power increased rapidly for decades and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

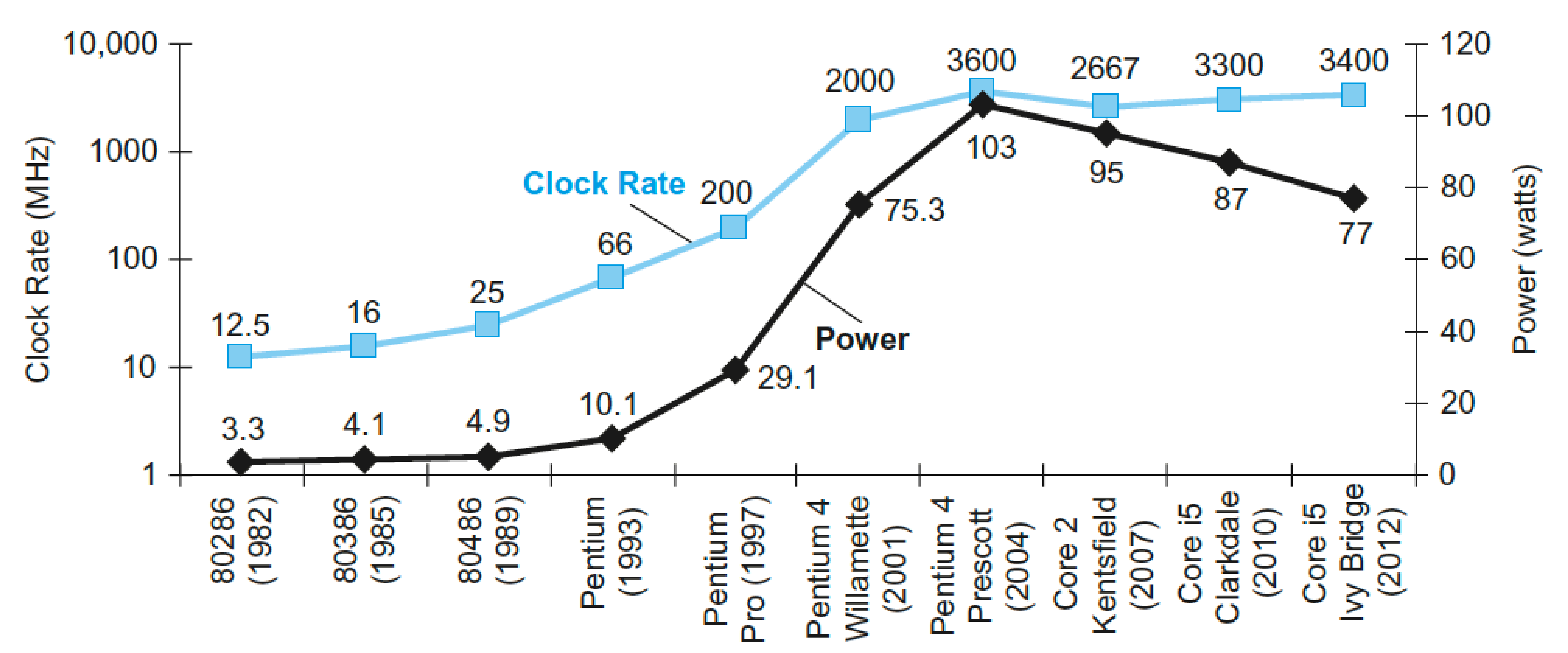


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 30 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

While backwards binary compatibility is sacrosanct, Figure 2.44 shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

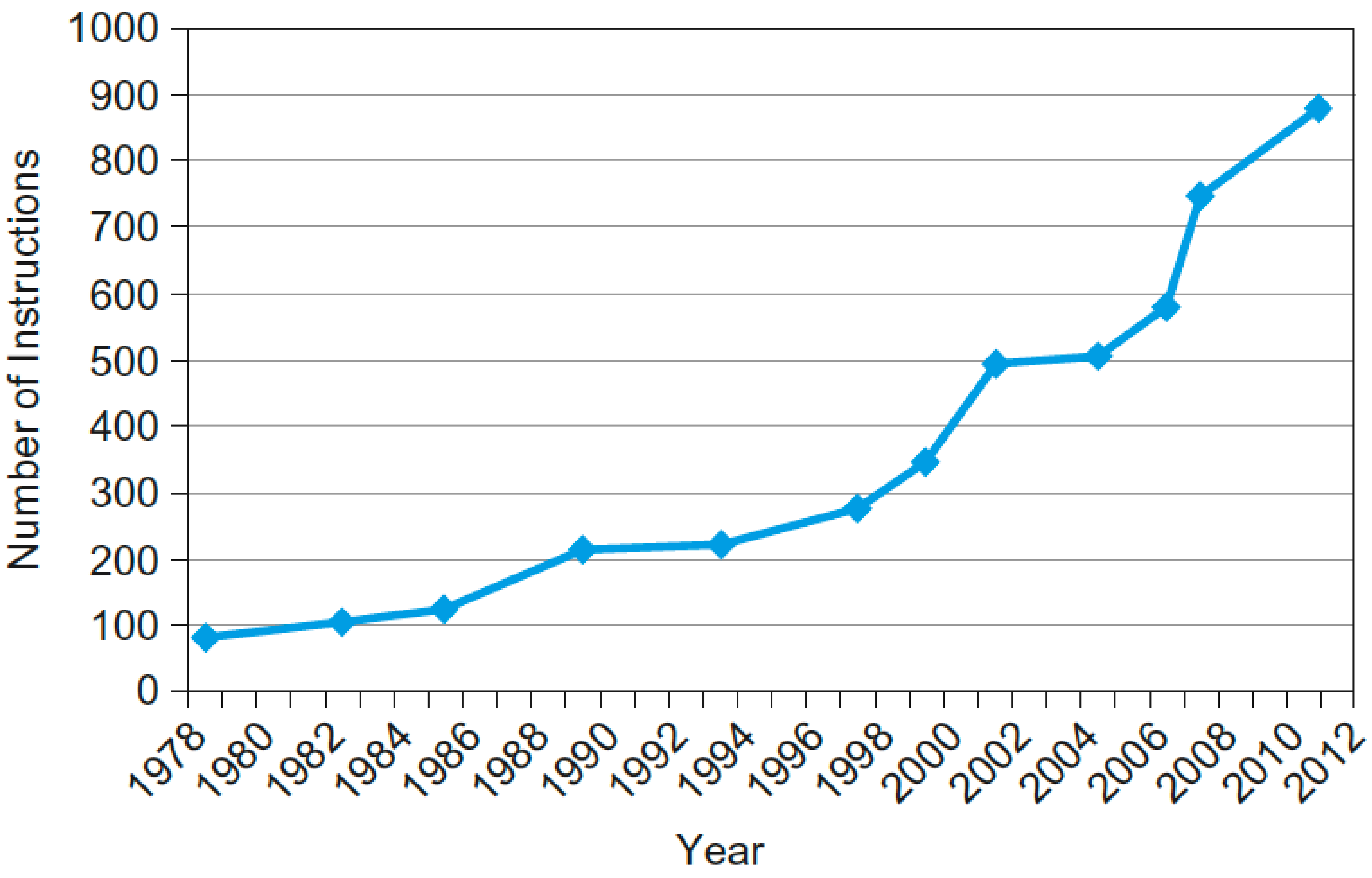


FIGURE 2.44 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

1978	96	8086 (3,2 µm nMOSFET)	
1980	83	8087 (3 µm)	
1982	105	80186	
1982	112	80286	
1982	84	80287	
1985	166	80386 (1,5 µm CHMOS III)	
1987	96	80387	
1989	267	80486DX/P4 (1 µm CHMOS IV)	FPU
1993	273	80586/P5/Pentium(0,8µ BiCMOS)	FPU
1995	304	80686/P6/Pentium Pro (350 nm)	FPU
1997	321	Pentium MMX (280 nm)	FPU, MMX
1997	333	6x86MX (Cyrix)	FPU, MMX, EMMI
1998	353	K6-2 (AMD, 250 nm)	FPU, MMX, 3DNow!
1999	358	K6-2+ (AMD, 180 nm)	FPU, MMX, Enhanced 3DNow!
1999	420	Pentium III (250 nm CMOS)	FPU, MMX, SSE
2000	489	Pentium 4 (180 nm)	FPU, MMX, SSE, SSE2
2003	528	K8 / Athlon 64 (AMD, 130 nm)	FPU,MMX,Enhanced 3DNow!,SSE,SSE2,AMD64
2004	499	Pentium 4 Prescott (90 nm)	FPU, MMX, SSE, SSE2, SSE3

Октомври 2015 г.: Xeon E3 v5 Skylake-DT (14 nm FinFET):

208 + 5 (CLMUL = Carry-less Multiplication) + 24 (BMI = Bit Manipulation Instructions)

+ 96 (FPU) + 20 (FMA = Fused Multiply-Add) =

+ 48 (MMX = Multimedia Extensions | Multiple Math Extensions | Matrix Math Extensions)

+ 68 (SSE = Streaming SIMD (Single Instruction, Multiple Data) Extensions)

+ 69 (SSE2)

+ 10 (SSE3)

+ 16 (SSSE3 = Supplemental SSE) = 515

+ 49 (SSE4.1)

+ 6 (SSE4.2)

+ 14 (x86-64)

+ 2 (ADX = Multi-Precision Add-Carry Instruction Extensions)

+ 12 (AVX = Advanced Vector Extensions)

+ 30 (AVX2)

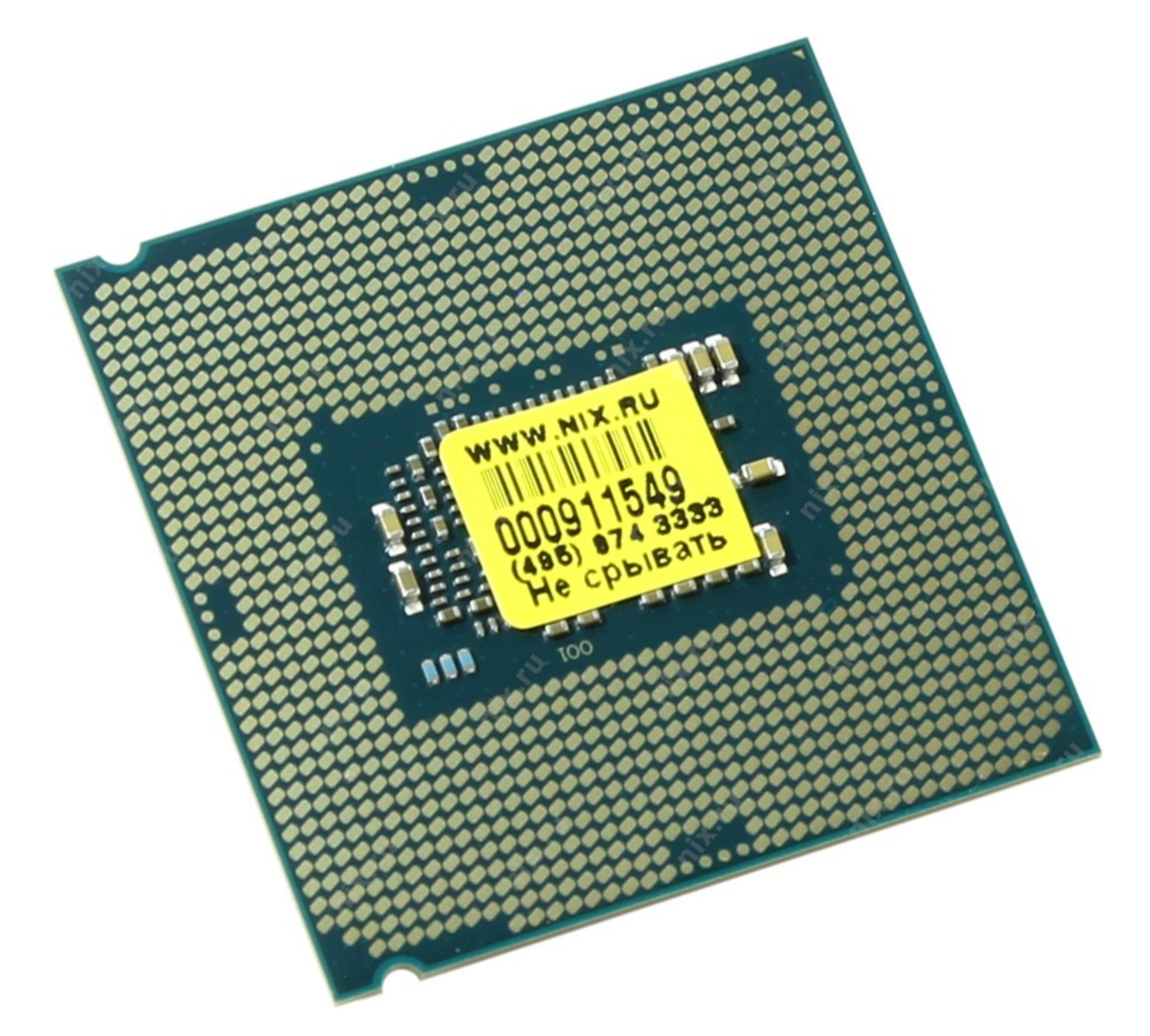
+ 13 + 6 + 8 + 8 + 8 + 18 + 2 + 44 + 12 + 16 + 63 + 6 + 10 + 16 + 12 + 9 + 2 (253 AVX3)

+ 8 (MPX = Memory Protection Extensions)

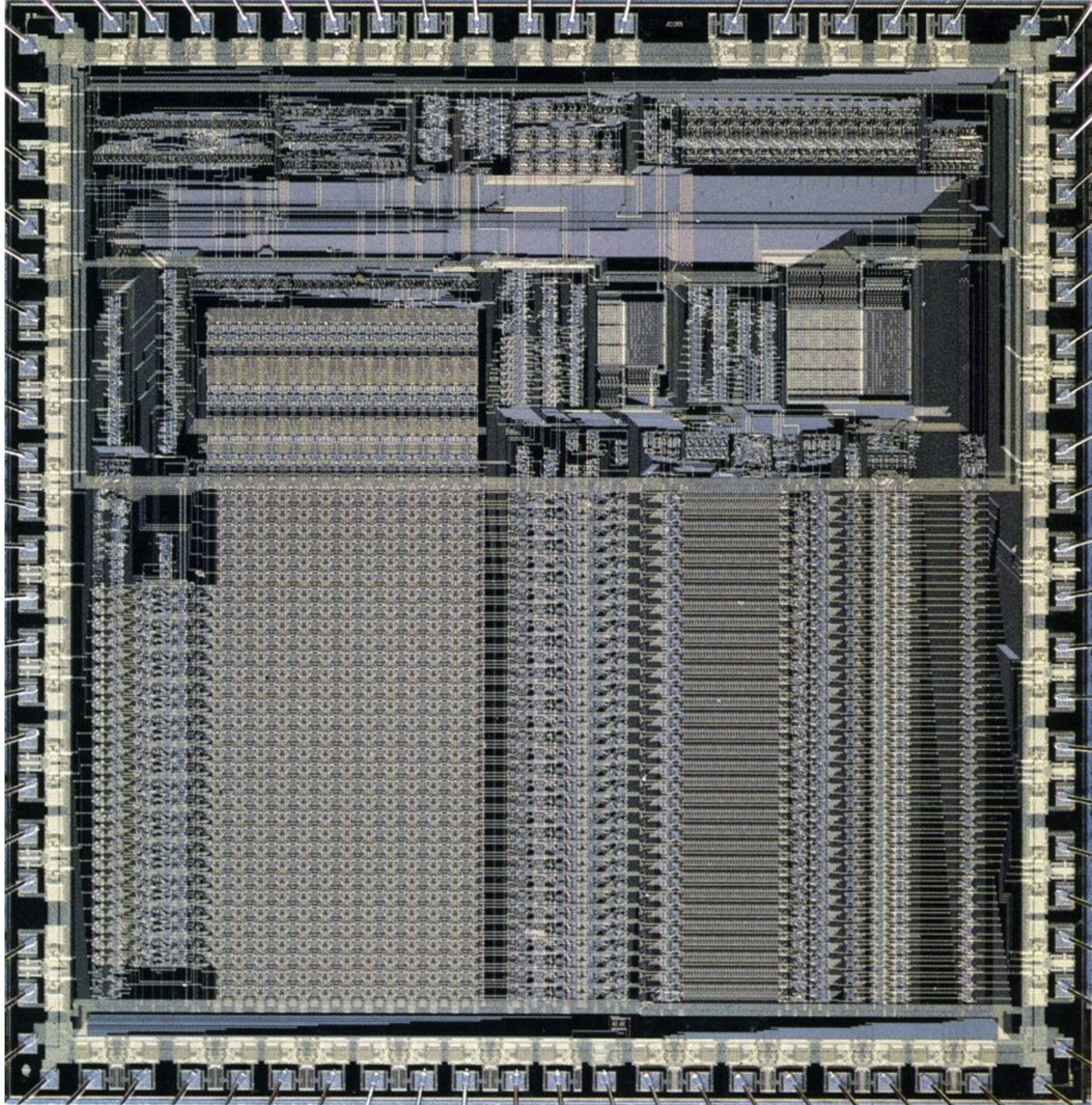
+ 4 (TSX = Transactional Synchronization Extensions) + 2 (SGX = Software Guard Extensions)

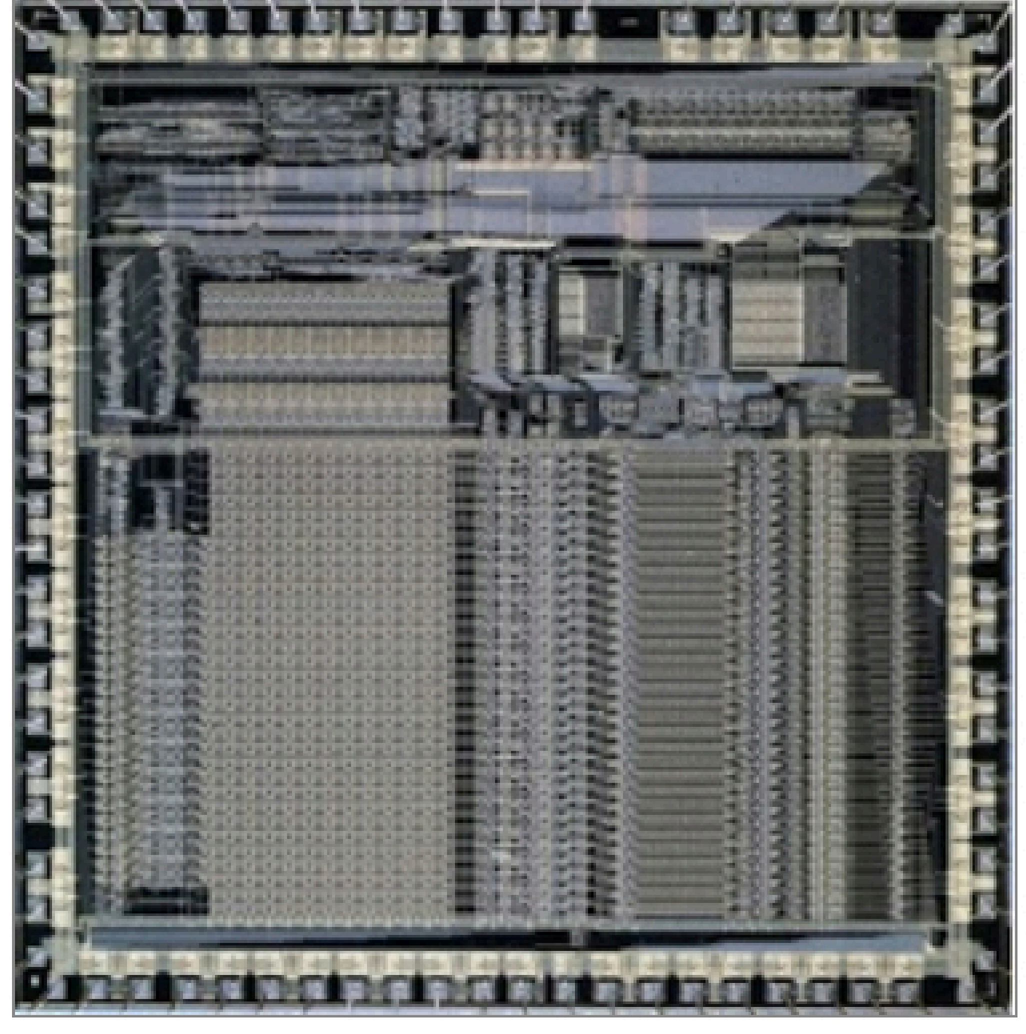
+ 10 (VT-x Virtualization) + 7 (AES-NI = Advanced Encryption Standard New Instructions) = 961

INTEL® XEON®
E3-1230V5
SR2LE 3.40GHZ
X547B152 e4


WWW.NIX.RU
000911549
(495) 974 3333
Не срывать
IOO

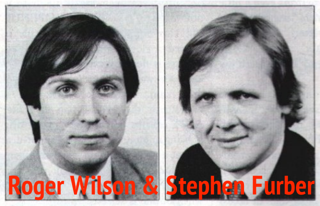
8625V
VC012
VC2588-0001
AUTUMN



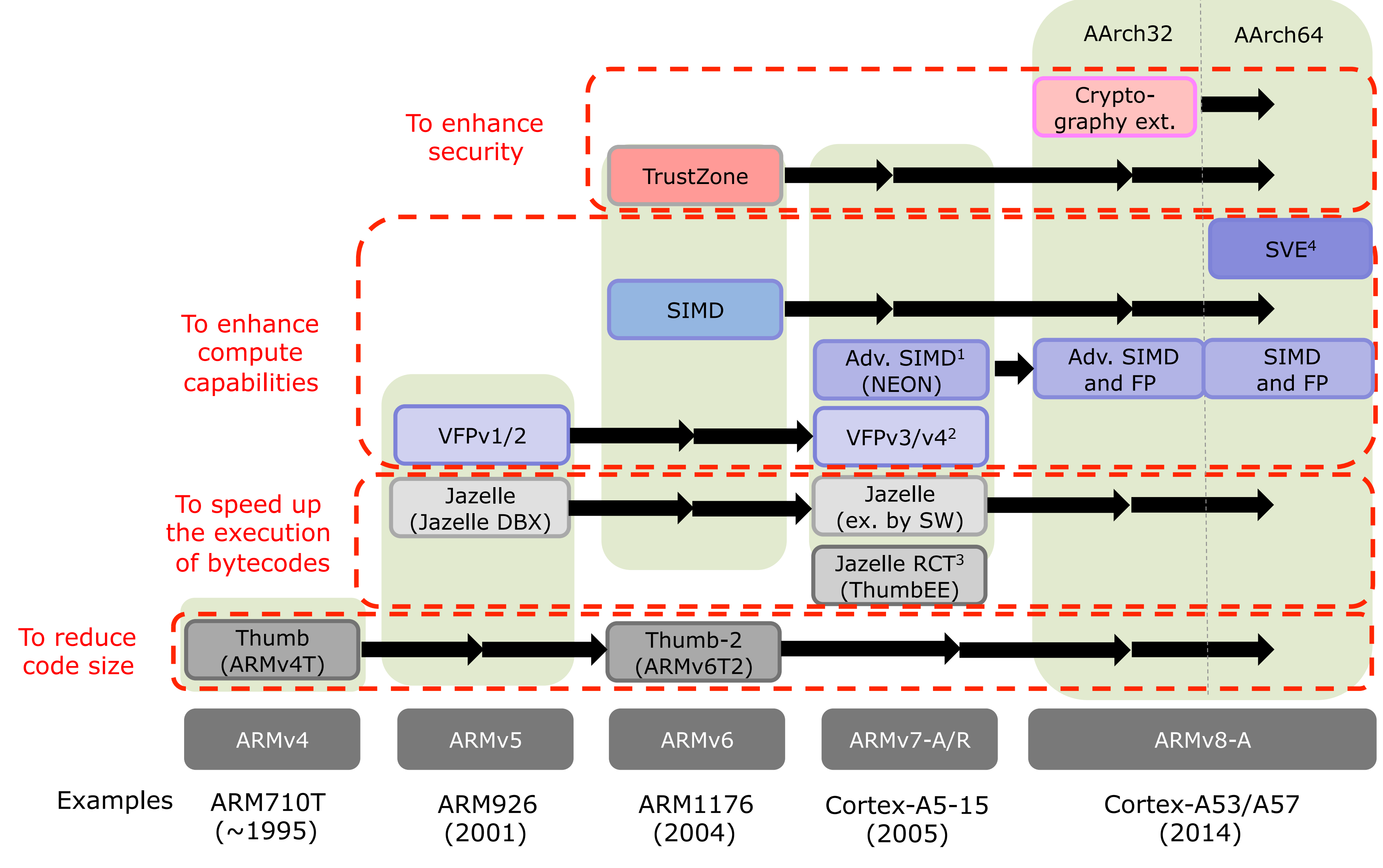


Die photos of the ARM1 processor and the Intel 386 processor to the same scale. The ARM1 is much smaller and contained 25,000 transistors compared to 275,000 in the 386. The 386 was higher density, with a 1.5 micron process compared to 3 micron for the ARM1. ARM1 photo courtesy of Computer History Museum. Intel A80386DX-20 by Pdesousa359, CC BY-SA 3.0.

Because of the ARM1's small transistor count, the chip used very little power: about 1/10 Watt, compared to nearly 2 Watts for the 386. The combination of high performance and low power


Roger Wilson & Stephen Furber

Main extensions introduced in ARM's basic ISA (simplified) -2 (Based on [64])

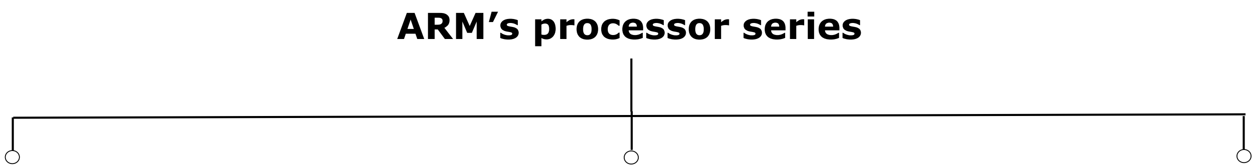


Remarks: See on the next slide.

3.1 Overview of ARM's processor series

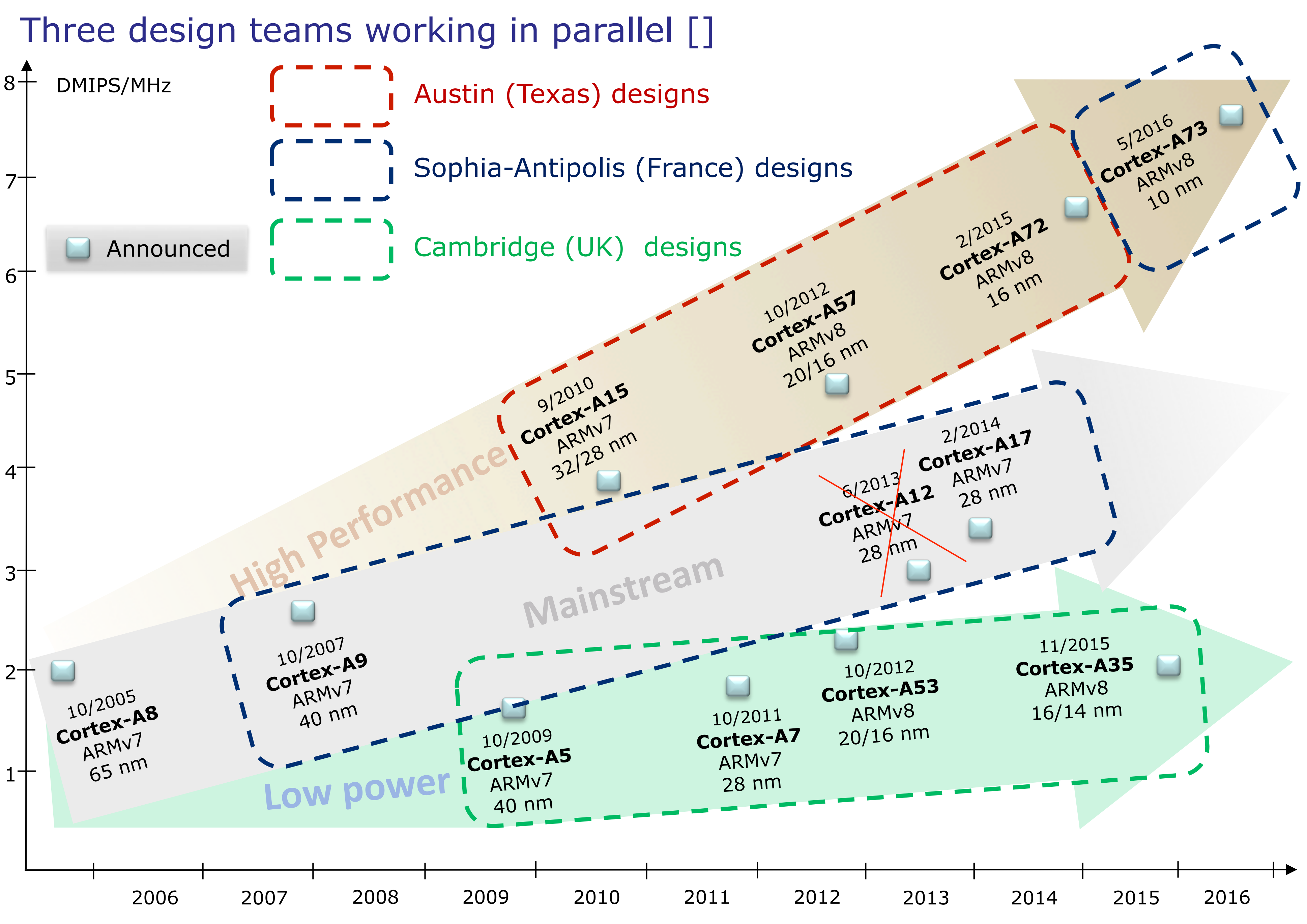
Subsequently, we give an overview of ARM’s processor series subdivided into three sections, according their underlying ISAs, as follows.

ARM’s processor series



Processors implementing the ARMv1 – ARMv2 ISA	**Processors implementing the ARMv3 – v6 ISA**	**Processors implementing the ARMv7 – ARMv8 ISA**
(Earliest ARM processors) (~ 1985-1990)	*(Early ARM processors)* (~ 1991-2004)	*(ARM’s Cortex processors)* (Since 2004/2012)
ARM1-ARM3	*ARM6xx-ARM11xx*	*Cortex lines*
26-bit address bus 32-bit data buses	32-bit address bus 32-bit data buses	ARMv7: 32-bit ARMv8: 64-bit with AArch64 and AArch32 modes

4. Overview of ARM's Cortex-A series (5a)



http://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled

Програмен модел: Програмно достъпни регистри в x86 и „ARM“. Флагове на регистъра за кода на условието (РКУ). Програмен модел на някои други МП.

CISC vs. RISC Today

PC Era

- Hardware translates x86 instructions into internal RISC instructions
  (Compiler vs Interpreter)
- Then use any RISC technique inside MPU
- > 350M / year !
- x86 ISA eventually dominates servers as well as desktops

PostPC Era: Client/Cloud

- IP in SoC vs. MPU
- Value die area, energy as much as performance
- > 20B total / year in 2017
- 99% Processors today are RISC
- *Marketplace settles debate*

*"A Decade of Mobile Computing", Vijay Reddi, 7/21/17, *Computer Architecture Today*

General-Purpose Registers

31 0

EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP

Segment Registers

15 0

CS
DS
SS
ES
FS
GS

Status and Control Registers

31 0

EFLAGS

31 0

EIP

General-Purpose Registers

31 … 16	15 … 8	7 … 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 3-4. Alternate General-Purpose Register Names

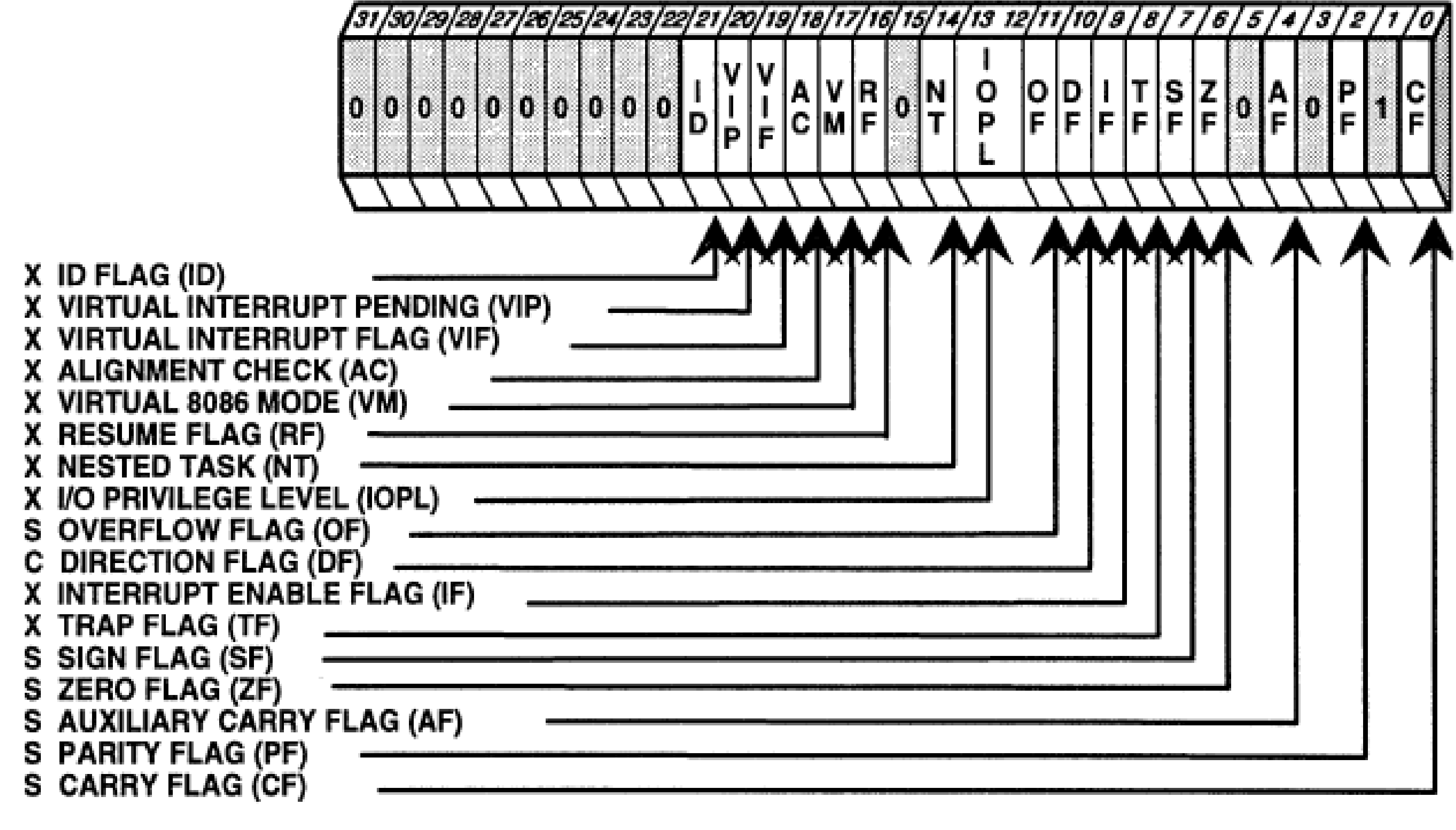


Figure 3-9. EFLAGS Register

Table 3-2. Status Flags

Name	Purpose	Condition Reported
OF	Overflow	Result exceeds positive or negative limit of number range
SF	Sign	Result is negative (less than zero)
ZF	Zero	Result is zero
AF	Auxiliary carry	Carry out of bit position 3 (used for BCD)
PF	Parity	Low byte of result has even parity (even number of set bits)
CF	Carry flag	Carry out of most significant bit of result

6.33.35 x86 Function Attributes

These function attributes are supported by the x86 back end:

`cdecl`
: On the x86-32 targets, the `cdecl` attribute causes the compiler to assume that the calling function pops off the stack space used to pass arguments. This is useful to override the effects of the '`-mrtd`' switch.

`fastcall`
: On x86-32 targets, the `fastcall` attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

`regparm (`*`number`*`)`
: On x86-32 targets, the `regparm` attribute causes the compiler to pass arguments number one to *number* if they are of integral type in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments continue to be passed all of their arguments on the stack.

`stdcall`
: On x86-32 targets, the `stdcall` attribute causes the compiler to assume that the called function pops off the stack space used to pass arguments, unless it takes a variable number of arguments.

„Using the GNU Compiler Collection" (GCC 9.1, 2019)

Microsoft Specific

The **__fastcall** calling convention specifies that arguments to functions are to be passed in registers, when possible. This calling convention only applies to the x86 architecture. The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	The first two DWORD or smaller arguments that are found in the argument list from left to right are passed in ECX and EDX registers; all other arguments are passed on the stack from right to left.
Stack-maintenance responsibility	Called function pops the arguments from the stack.
Name-decoration convention	At sign (@) is prefixed to names; an at sign followed by the number of bytes (in decimal) in the parameter list is suffixed to names.
Case-translation convention	No case translation performed.

Note

Future compiler versions may use different registers to store parameters.

10.4.1 Passing Arguments Using Register-Based Calling Conventions

How arguments are passed to a function with register-based calling conventions is determined by the size (in bytes) of the argument and where in the argument list the argument appears. Depending on the size, arguments are either passed in registers or on the stack. Arguments such as structures are almost always passed on the stack since they are generally too large to fit in registers. Since arguments are processed from left to right, the first few arguments are likely to be passed in registers (if they can fit) and, if the argument list contains many arguments, the last few arguments are likely to be passed on the stack.

The registers used to pass arguments to a function are EAX, EBX, ECX and EDX. The following algorithm describes how arguments are passed to functions.

Initially, we have the following registers available for passing arguments: EAX, EDX, EBX and ECX. Note that registers are selected from this list in the order they appear. That is, the first register selected is EAX and the last is ECX. For each argument *Ai,* starting with the left most argument, perform the following steps.

1. If the size of *Ai* is 1 byte or 2 bytes, convert it to 4 bytes and proceed to the next step. If *Ai* is of type "unsigned char" or "unsigned short int", it is converted to an "unsigned int". If *Ai* is of type "signed char" or "signed short int", it is converted to a "signed int". If *Ai* is a 1-byte or 2-byte structure, the padding is determined by the compiler.

2. If an argument has already been assigned a position on the stack, *Ai* will also be assigned a position on the stack. Otherwise, proceed to the next step.

3. If the size of *Ai* is 4 bytes, select a register from the list of available registers. If a register is available, *Ai* is assigned that register. The register is then removed from the list of available registers. If no registers are available, *Ai* will be assigned a position on the stack.

(OpenWatcom C/C++ User's Guide)

Процесорната фамилия ARM (Advanced RISC Machines) се състои от RISC микропроцесори, които имат 16 регистъра (фиг. 12.1) с общо предназначение с имена от R0 до R15. Регистрите са 32-битови. Те могат да съдържат както адреси, така и данни. Последният регистър R15 се използва за програмен брояч (PC), а регистър R13 служи за организиране на програмен стек (SP). Регистър R14 (LR) се използва като регистър, съдържащ адреса за връщане след подпрограма. Регистрите могат да се използват за съхранение на 8, 16 и 32-битови числа.

бит 31 … бит 0

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (Stack pointer – SP)
R14 (link register – LR)
R15 (PC)

Фигура 12.1. Регистри на процесора ARM

There are a number of different processor modes. These are shown in the following table:

Processor mode			Description
1	User	(usr)	the normal program execution mode
2	FIQ	(fiq)	designed to support a high-speed data transfer or channel process
3	IRQ	(irq)	used for general-purpose interrupt handling
4	Supervisor	(svc)	a protected mode for the operating system
5	Abort	(abt)	used to implement virtual memory and/or memory protection
6	Undefined	(und)	used to support software emulation of hardware coprocessors
7	System	(sys)	used to run privileged operating system tasks (Architecture Version 4 only)

Table 3-1: ARM processor modes

Mode changes may be made under software control or may be caused by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as *privileged* modes, will be entered to service interrupts or exceptions or to access protected resources: see ➲*3.10 Exceptions* on page 3-12.

User/ System	Supervi-sor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

Table 3-2: The ARM register set

Registers 0-12 are always free for general-purpose use. Registers 13 and 14, although available for general use, also have specific roles:

Register 13 (also known as the *Stack Pointer* or SP) is banked across all modes to provide a private Stack Pointer for each mode (except System mode which shares the user mode R13).

Register 14 (also known as the *Link Register* or LR) is used as the subroutine return address link register. R14 is also banked across all modes (except System mode which shares the user mode R14).

When a Subroutine call (Branch and Link instruction) is executed, R14 is set to the subroutine return address. The banked registers R14_SVC, R14_IRQ, R14_FIQ, R14_ABORT and R14_UNDEF are used similarly to hold the return address when exceptions occur (or a subroutine return address if subroutine calls are executed within interrupt or exception routines). R14 may be treated as a general-purpose register at all other times.

Register 15 is used specifically to hold the *Program Counter* (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. When R15 is written bits[1:0] are ignored and bits[31:2] are written to the PC. Depending on how it is used, the value of the PC is either the address of the instruction plus *n* (where *n* is 8 for ARM state and 4 for Thumb state) or is unpredictable.

CPSR is the Current Program Status Register. This is accessible in all processor modes, and contains the condition code flags, interrupt enable flags, and current processor mode. In Architecture 4T, the CPSR also holds the processor state. See ➲*3.9 Program Status Registers* on page 3-10 for more information.

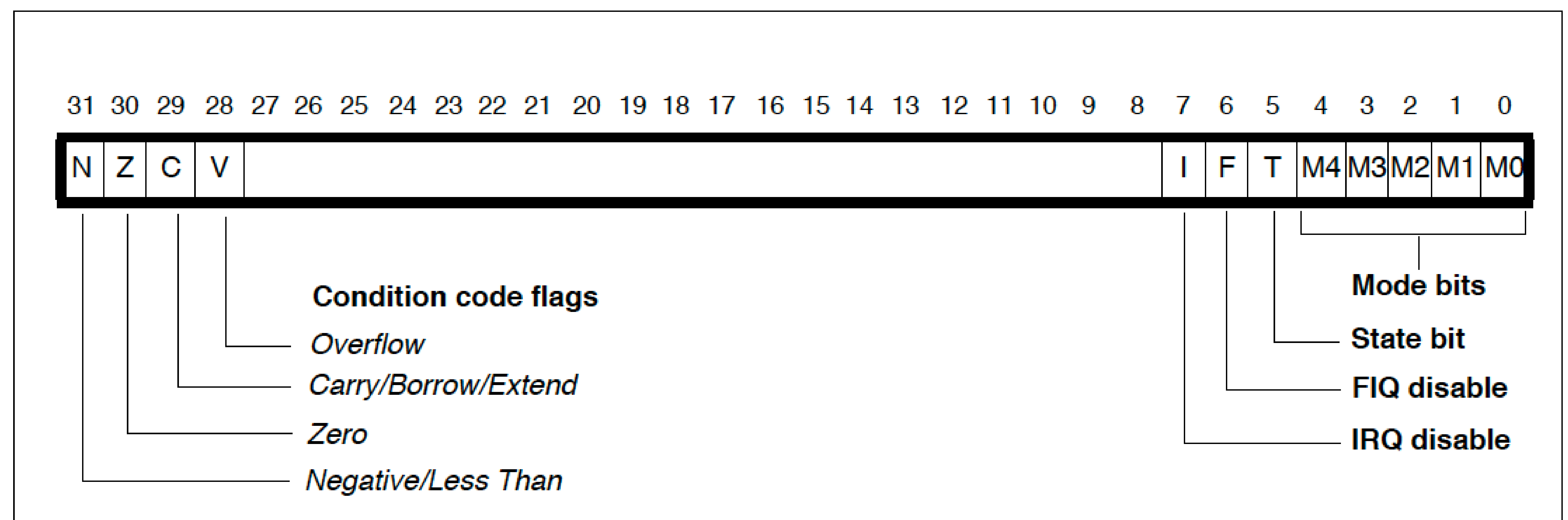


Figure 3-4: Program Status Register format

The condition code flags

The N, Z, C and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*. The condition code flags in the CPSR can be changed as a result of arithmetic and logical operations in the processor, and can be tested by all ARM instructions to determine if the instruction is to be executed. All ARM instructions may be executed conditionally

The bottom 8 bits of a PSR (incorporating I, F, T and M[4:0]) are known collectively as the *control bits*. These change when an exception arises, and can be altered by software only when the processor is in a privileged mode.

Interrupt disable bits — The I and F bits are the *interrupt disable bits*. When set, these disable the IRQ and FIQ interrupts respectively.

The state bit — Bit T is the processor *state bit*. When the state bit is set to 0, this indicates that the processor is in ARM state (ie. executing 32-bit ARM instructions). When it is set to 1, this indicates that the processor is in Thumb state (executing 16-bit Thumb instructions)

The state bit is only implemented on Thumb-aware processors (Architecture 4T). On non Thumb-aware processors the state bit will always be zero.

The mode bits — The M4, M3, M2, M1 and M0 bits (M[4:0]) are the *mode bits*. These determine the mode in which the processor operates, as shown in *Table 3-4: The mode bits*, below. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described can be used.

..

M[4:0]	Mode	Accessible Registers
10000	User	PC, R14 to R0, CPSR
10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq, R13_irq,R12 to R0, CPSR, SPSR_irq
10011	SVC	PC, R14_svc, R13_svc,R12 to R0, CPSR, SPSR_svc
10111	Abort	PC, R14_abt, R13_abt,R12 to R0, CPSR, SPSR_abt
11011	Undef	PC, R14_und, R13_und,R12 to R0, CPSR, SPSR_und
11111	System	PC, R14 to R0, CPSR (Architecture 4 only)

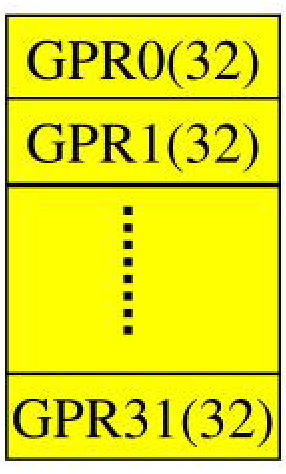
Преносът при изваждане е инверсен!

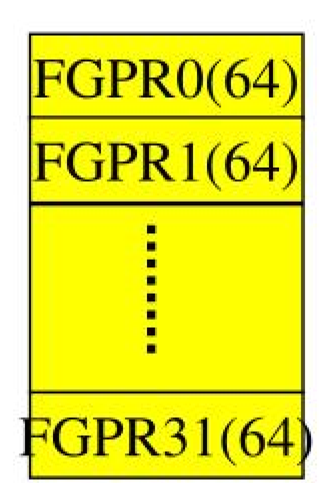
В много микропроцесори се използва този трик – изваждането да се извършва като събиране с инверсната стойност на умалителя плюс лог. 1 на входа за пренос. Така преносът се получава инвертиран на изхода за пренос на суматора. А ако следващата команда е изваждане с пренос (SBC), то тази команда изважда инверсията на преноса. По-подробно развито, ако има пренос, изваждането се свежда отново да събиране с инверсната стойност на умалителя плюс лог. 1 на входа за пренос на суматора, тъй като инвесията на преноса е лог. 0. А ако пренос няма, то от тази лог. 1 се изважда лог. 1 (инверсията на преноса) и така на входа за пренос на суматора ще има лог. 0. На практика това означава, че на втория вход на суматора се подава инверсията на умалителя (получена от инверсните изходи на тригерите от регистъра, където се пази той), а на входа му за пренос – преносът от предходното така извършено изваждане. Така е и при „ARM“.

Register	Synonym	Special	Role in the procedure call standard	Preserve across function calls?
R15		PC	The Program Counter.	Special role register
R14		LR	The Link Register.	Special role register
R13		SP	The Stack Pointer.	Special role register
R12		IP	The Intra-Procedure-call scratch register.	No
R11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.	Yes, if used
R10	v7	SL	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.	Yes, if used
R9	v6	SB	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants.	Yes, if used
R8	v5		ARM-state variable-register 5.	Yes, if used
R7	v4	WR	Variable register (v-register) 4. Thumb-state Work Register.	Yes, if used
R6	v3		Variable register (v-register) 3.	Yes, if used
R5	v2		Variable register (v-register) 2.	Yes, if used
R4	v1		Variable register (v-register) 1.	Yes, if used
R3	a4		Argument/result/scratch register 4.	No
R2	a3		Argument/result/scratch register 3.	No
R1	a2		Argument/result/scratch register 2.	No
R0	a1		Argument/result/scratch register 1.	No

PowerPC programming model - Register Set

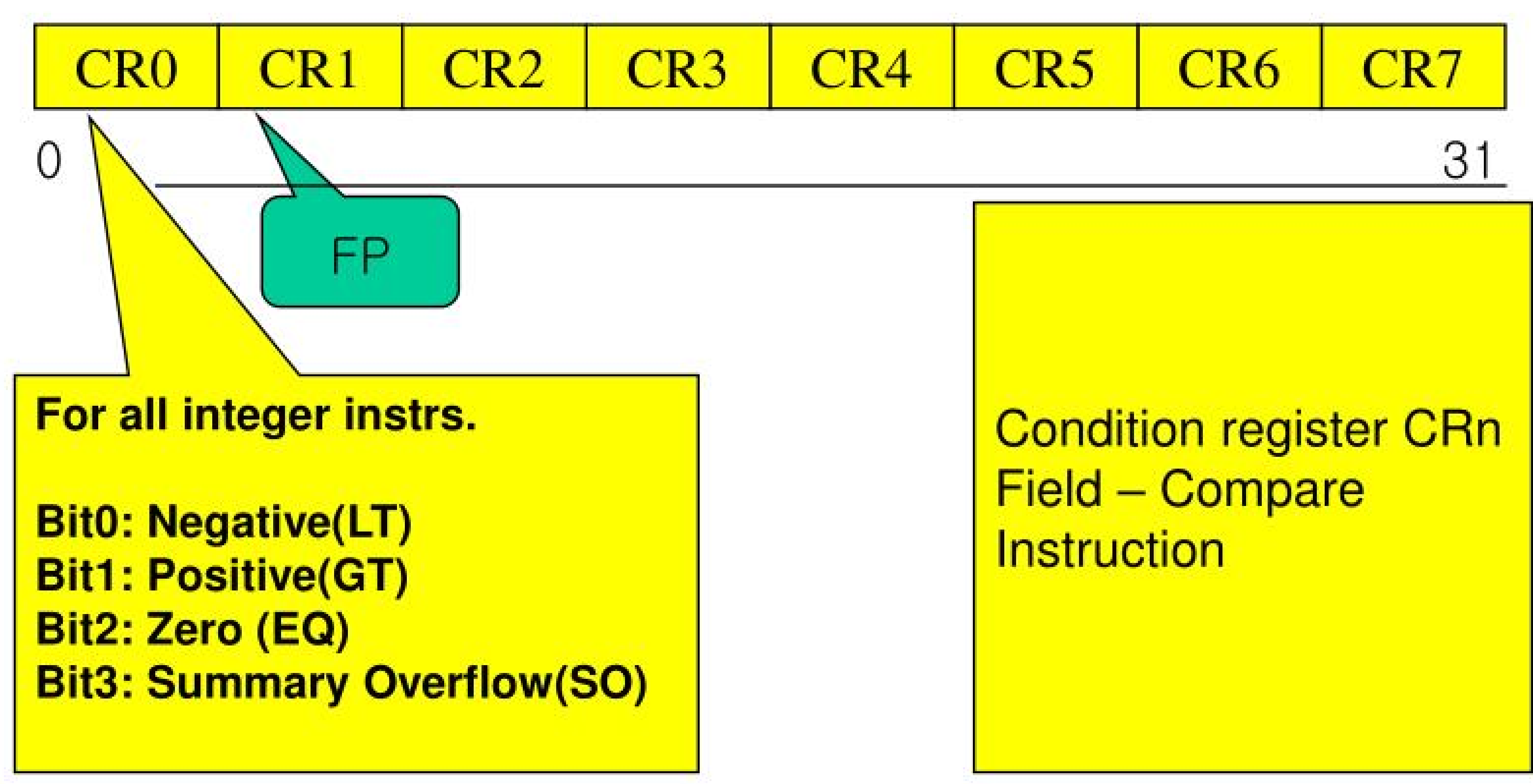
- **User Model – UISA (32-bit architecture)**





Condition Registers (CR)

- **For testing and branching**



back

XER Register (XER)

Figure 2-6. XER Register

back

Table 2-6. XER Bit Definitions

Bits	Name	Description
0	SO	Summary overflow. The summary overflow bit (SO) is set whenever an instruction (except mtspr) sets the overflow bit (OV). Once set, the SO bit remains set until it is cleared by an mtspr instruction (specifying the XER) or an mcrxr instruction. It is not altered by compare instructions, nor by other instructions (except mtspr to the XER, and mcrxr) that cannot overflow. Executing an mtspr instruction to the XER, supplying the values zero for SO and one for OV, causes SO to be cleared and OV to be set.
1	OV	Overflow. The overflow bit (OV) is set to indicate that an overflow has occurred during execution of an instruction. Add, subtract from, and negate instructions having OE = 1 set the OV bit if the carry out of the msb is not equal to the carry out of the msb + 1, and clear it otherwise. Multiply low and divide instructions having OE = 1 set the OV bit if the result cannot be represented in 64 bits (mulld, divd, divdu) or in 32 bits (mullw, divw, divwu), and clear it otherwise. The OV bit is not altered by compare instructions that cannot overflow (except mtspr to the XER, and mcrxr).

XER Register (XER), cont'd

Table 2-6. XER Bit Definitions (continued)

Bits	Name	Description
2	CA	Carry. Set during execution of the following instructions: • Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of the msb, and clear it otherwise. • Shift right algebraic instructions set CA if any 1 bits have been shifted out of a negative operand, and clear it otherwise. The CA bit is not altered by compare instructions, nor by other instructions that cannot carry (except shift right algebraic, mtspr to the XER, and mcrxr).
3–24	—	Reserved
25–31	Byte count	This field specifies the number of bytes to be transferred by a Load String Word Indexed (lswx) or Store String Word Indexed (stswx) instruction.

Table 13-1 GPR Register Usage Conventions

GPR	Type	Must be Preserved?	Usage
`r0`	volatile	no	used in prolog/epilog code
`r1`	dedicated	yes	stack pointer
`r2`			table of contents (TOC) pointer
`r3`	volatile	no	1st fixed-point parameter 1st word of return value
`r4`			2nd fixed-point parameter 2nd word of return value
`r5`	volatile	no	3rd fixed-point parameter
`...`			...
`r10`			8th fixed-point parameter
`r11`	volatile	no	environment pointer (if needed)
`r12`			used by global linkage routines
`r13`	non-volatile	yes	general registers that must be preserved across function calls
`...`			
`r31`			

i0
i1
i2
i3
i4
i5
i6
i7
l0
l1
l2
l3
l4
l5
l6
l7
o0
o1
o2
o3
o4
o5
o6
o7

g0
g1
g2
g3
g4
g5
g6
g7

Y (multiply step)

PSR

NZVC S -cwp-

(cwp = current window pointer)

Trap Base Register (TBR)

Window Invalid Mask (WIM)

PC

nPC

Figure 2.1: SPARC Programming Model

In the HP Calculator, the last number computed could be tested. For example, there was an instruction `ifeq`, which would skip the next instruction in line if the result last computed was zero. A similar technique is used in many computers, in which the state of the execution of each instruction may be tested. In order to do this, only information about the result need be kept, not the result itself. The state of execution is saved in terms of four variables:

Z whether the result was zero

N whether the result was negative

V whether execution resulted in a number too large to store in the register

C whether execution resulted in a number that generated a carry out of the register

This information is kept in four variables, the integer condition codes: Z, N, V, and, C.

Register	Synonyms	Usage
%g0	%r0	Always discards writes and returns zero
%g1	%r1	First of seven registers for data with
%g2	%r2	global context
%g3	%r3	
%g4	%r4	
%g5	%r5	
%g6	%r6	
%g7	%r7	
%o0	%r8	First of six registers for local data
%o1	%r9	and arguments to called subroutines
%o2	%r10	
%o3	%r11	
%o4	%r12	
%o5	%r13	
%sp	%r14 %o6	Stack pointer
%o7	%r15	Called subroutine return address
%l0	%r16	First of eight registers for local
%l1	%r17	variables
%l2	%r18	
%l3	%r19	
%l4	%r20	
%l5	%r21	
%l6	%r22	
%l7	%r23	
%i0	%r24	First of six registers for incoming
%i1	%r25	subroutine arguments
%i2	%r26	
%i3	%r27	
%i4	%r28	
%i5	%r29	
%fp	%r30 %i6	Frame pointer
%i7	%r31	Subroutine return address

Name	Register (bits 31–0)
zero	0
at	1
v0	2
v1	3
a0	4
a1	5
a2	6
a3	7
t0	8
t1	9
t2	10
t3	11
t4	12
t5	13
t6	14
t7	15
s0	16
s1	17
s2	18
s3	19
s4	20
s5	21
s6	22
s7	23
t8	24
t9	25
k0	26
k1	27
gp	28
sp	29
fp	30
ra	31

General–purpose registers

31–0
HI
LO

Multiply and divide registers

31–0
PC

Program counter

Figure 4.1 MIPS registers. All registers are 32-bits wide.

Table 4.1 MIPS registers and their conventional usage

Register name	Number	Intended usage
zero	0	Constant 0
`$at`	1	Reserved for assembler
`$v0`, `$v1`	2, 3	Results of a procedure
`$a0`, `$a1`, `$a2`, `$a3`	4–7	Arguments 1–4 (not preserved across call)
`$t0`– `$t7`	8–15	Temporary (not preserved across call)
`$s0`–`$s7`	16–23	Saved temporary (preserved across call)
`$t8`, `$t9`	24, 25	Temporary (not preserved across call)
`$k0`, `$k1`	26, 27	Reserved for OS kernel
`$gp`	28	Pointer to global area
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer (if needed); otherwise, a saved register `$s8`
`$ra`	31	Return address (used to return from a procedure)

user-level base integer registers

RV128

RV64

RV32

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

RV32E

127 63 31 0

PC

127 63 31 0

Register	ABI Name	Description	Saver
x0	zero	hardwired zero	-
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	Caller

Команди за работа с цели числа и системни команди на x86 и „ARM“: Групи команди. Операнди. Адресация. Неявна адресация при x86. Ортогоналност.

Why Would Anyone Learn This Stuff? Forward

Amazing! You're actually reading this. That puts you into one of three categories: a student who is being forced to read this stuff for a class, someone who picked up this book by accident (probably because you have yet to be indoctrinated by the world at large), or one of the few who actually have an interest in learning assembly language.

Egads. What kind of book begins this way? What kind of author would begin the book with a forward like this one? Well, the truth is, *I* considered putting this stuff into the first chapter since most people never bother reading the forward. A discussion of what's right and what's wrong with assembly language is very important and sticking it into a chapter might encourage someone to read it. However, I quickly found that university students can skip Chapter One as easily as they can skip a forward, so this stuff wound up in a forward after all.

So why would anyone learn this stuff, anyway? Well, there are several reasons which come to mind:

- Your major requires a course in assembly language; i.e., you're here against your will.
- A programmer where you work quit. Most of the source code left behind was written in assembly language and you were elected to maintain it.
- Your boss has the audacity to insist that you write your code in assembly against your strongest wishes.
- Your programs run just a little too slow, or are a little too large and you think assembly language might help you get your project under control.
- You want to understand how computers actually work.
- You're interested in learning how to write efficient code.
- You want to try something new.

Well, whatever the reason you're here, welcome aboard. Let's take a look at the subject you're about to study.

Randall Hyde, "The Art of Assembly", 1st ed.

Защо да се изучава език Асемблер

Първата причина да се работи и изучава Асемблер е, че той дава възможности за разбиране на основните процеси и функционалност на компютърната система. Това помага на програмистите да създават софтуерни продукти с по-логична и смислена структура.

Втората причина е осигуряването на много добър контрол на хардуерните и софтуерните възможности на компютърната система, тъй като в език Асемблер се работи с директни процесорни команди.

Друга причина е, че асемблерските програми са много бързи и имат по-голям обхват от програмите, които могат да се разработят с други програмни езици. Език Асемблер позволява добра оптимизация в програмите, както по отношение на техния размер, така и по отношение на изпълнението им.

Много полезно за прграмистите е да знаят Асемблер, особено в следните случаи:

- Когато трябва да се анализират грешки в програмите;
- Когато дадена програма се изпълнява по различен начин от планирания;
- Когато език от високо ниво не подържа някои хардуерни възможности;
- Когато в линейните процедури се изисква някаква порция асемблерски команди.

Разбирането на процесите на компилация и свързване на програми, написани на език от високо ниво изисква познаването на език Асемблер.

Особено важно е използването на език Асемблер в системите, работещи в реално време. При тях от голяма важност е контролът на отделните инструкции, тъй като се изисква прецизна съгласуваност по време на различни операции. В програмните езици от високо ниво не може да се говори за отделни инструкции и съгласуването на операциите е почти невъзможно.

проф. Станко Щраков

0:47:57 PE: 1990, summer of 90, was to do an annotated bibliography of papers and books that I thought were relevant to what we should design next and a lot of the papers being done back then were out of order. They were predicting that there is a lot of parallelism if you can make a machine that can dig the parallelism out. There were papers from Illinois and other places suggesting some possible ways to do out of order so we looked at them and thought okay, one of those might play out. We were very hopeful that out of order was going to be a viable strategy and we committed to it reasonably early, as of September of 1990, that that's what we are going to end up doing. Another question was the right way to map an out of order schema onto an x86 platform because as I mentioned there's all this research going on but none of it was x86, they were all assuming RISC instruction set architectures, so we had an open question as to whether there anything about the x86 instruction set architecture that would prevent those research results from being relevant. Your research might predict a 3x speed up on important code but it's RISC code and if I did it in x86 would if I find out that I will only get 1.2x because of some horrible thing associated with the architecture of the x86?

None of us were x86 experts back then, not me, not Dave, not Glenn, in fact we had all studiously avoided it because it is ugly, it's not a pretty architecture at all. Suddenly we had to climb up a new learning curve for x86 as well as for out of order. After a discussion one day, Dave and I realized that if we had a tool that could track data dependency chains through assembly language, we would be able to characterize x86 dependency patterns and statistics. Once who depends on whom, you can effectively gauge the efficacy of taking later instructions and pretending that they have been executed at the same time as the previous one. That is of course leaving out all the implementation details of how would you do that, what kind of hardware would it take and so on, but we were just asking a question, if you could build such an x86 engine, conceptually how fast could it possible go.

Robert Colwell, 2009 interview

Second, *look closely at the microcode.* Although microcode seems to have disproportionately more bugs than anything else in the chip, if you understand the x86 and the design process, that outcome is understandable. The x86 is an extremely complex instruction-set architecture, and most of that complexity is embedded in the microcode. You might think that a company that has successfully implemented x86 chips for close to 30 years would have "solved" the microcode problem long ago, but they haven't, because no such solution exists. Every time the microarchitecture must be changed, so must the microcode, and all fundamental changes to either will expose new areas in the microcode, for which the past is a poor guide to correctness.

Robert Colwell, „The Pentium chronicles" (всички необозначени цитати по-нататък са оттам)

Прехвърляне	Аритметични	Побитови	За преходи	За низове	ПСУ
MOV, PUSH, POP, XCHG, XLAT	ADD, ADC, AAA, DAA, INC	AND, OR, XOR, NOT, TEST	CALL, RET, JMP	REP, REPE/ REPZ, REPNE/ REPNZ	INT, INTO, IRET
IN, OUT	SUB, SBB, AAS, DAS, DEC, NEG, CMP	SAL/SHL, SAR, SHR	JA/JNBE, JAE/ JNB/JNC, JB/ JNAE/JC, JBE/ JNA, JCXZ, JE/JZ, JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JNE/JNZ, JNO, JNP/JPO, JNS, JO, JP/JPE, JS	MOVSB, MOVSW	STC, CLC, CMC, STD, CLD, STI, CLI
LEA, LDS, LES	MUL, IMUL, AAM	ROL, ROR, RCL, RCR	LOOP, LOOPE/ LOOPZ, LOOPNE/ LOOPNZ	CMPSB, CMPSW	HLT, WAIT, LOCK (ESC е за копроцесор!)
LAHF, SAHF, PUSHF, POPF	DIV, IDIV, AAD			SCASB, SCASW	(NOP – това е XCHG AX,AX !)
	CBW, CWD			LODSB, LODSW, STOSB, STOSW	+SALC (недо-кументирана) = 96 бр.

Undocumented OpCodes: SALC

SALC - D6 - Set AL on Carry

An undocumented op code that performs an operation common to every Assembly language subroutine to C and many other higher level languages. This instruction is a C programmers 'dream' instruction for interfacing to assembly language.

```
Undocumented:  Available to all Intel x86 processors
               Useful in production source code.
                                                                SALC
Flags:                                              SET Carry flag to AL
+-+-+-+-+-+-+-+-+-+                                         +-----------+
|O|D|I|T|S|Z|A|P|C|                                         | 11010110 |
+-+-+-+-+-+-+-+-+-+                                         +-----------+
| | | | | | | | | |                                         |    D6     |
+-+-+-+-+-+-+-+-+-+                                         +-----------+
```

The name SALC simply stands for SET the Carry flag in AL. This instruction is categorized as an undocumented single-byte proprietary instruction. Intel claims it can be emulated as a NOP. Hardly a NOP, this instruction sets AL=FF if the Carry Flag is set (CF=1), or resets AL=00 if the Carry Flag is clear (CF=0). It can best be emulated as SBB AL,AL. SALC doesn't change any flags, where SBB AL,AL does. This instruction is most useful to high-level language programmers whose programs call assembly language, and expect AL to indicate success or failure. Since it is convenient for assembly language programs to return status in the CF, this instruction will convert that status to a form compatible with high level languages.

Over the years, this instruction has been given many names by various *discoverers*. I originally gave it the name SETCAL, but the most common name I've seen in print is SETALC. The name given above, SALC is an official Intel name. While perusing the P6 opcode map, I always check for *known*, undocumented opcodes. After weeding through the map for many minutes, my patience and perseverance paid off. I found the opcode, and its name. Intel's name for this opcode is SALC. This would indicate that Intel plans to *officially* document this instruction, beginning with the P6.

Преобразувания и прехвърляния

BSWAP (размяна на байтовете);
CBW (байт в дума);
CDQ (двойна в четворна дума);
CWD (дума в двойна дума с участие на DX);
CWDE (дума в двойна дума само в EAX);
LDS (зареждане на пълен логически адрес, включително в DS);
LEA (зареждане на ефективен адрес);
LES (зареждане на пълен логически адрес, включително в ES);
LFS (зареждане на пълен логически адрес, включително в FS);
LGS (зареждане на пълен логически адрес, включително в GS);
LSL (зареждане на граница на сегмент);
LSS (зареждане на пълен логически адрес, включително в SS);
MOV (записване на избрано място);
MOVSX (прехвърляне със знаково разширяване на разрядността);
MOVZX (прехвърляне с беззнаково разширяване на разрядността);
XCHG (размяна на местата на операндите);
XLAT (извлича байт от таблица от до 256 байта) ;
XLATB (като XLAT, но само с регистъра DS).

Работа със стека

ENTER (формиране на стеков кадър);
LAHF (запис на младшия байт от флагове в регистъра AH);
LEAVE (премахване на стеков кадър);
POP (извличане на една данна);
POPA (извличане на 16-разрядните регистри с общо предназначение);
POPAD (извличане на 32-разрядните регистри с общо предназначение);
POPF (извличане на 16-разрядния флагов регистър);
POPFD (извличане на 32-разрядния флагов регистър; при привилегии 1, 2 и 3 не се променят флаговете VM и RF);
PUSH (записване на единична данна);
PUSHA (записване на 16-разрядните регистри с общо предназначение);
PUSHAD (записване на 32-разрядните регистри с общо предназначение);
PUSHF (записване на 16-разрядния флагов регистър);
PUSHFD (записване на 32-разрядния флагов регистър).

Аритметични операции

ADC (събиране заедно с флага CF);
ADD (събиране);
CMP (изваждане с цел сравняване);
CMPXCHG (изваждане с цел сравняване и обмен според равенството);
DEC (намаляване с 1);
DIV (делене на цели без знак);
IDIV (делене на цели със знак);
IMUL (умножение на цели със знак);
INC (увеличаване с 1);
MUL (умножение на цели без знак);
NEG (смяна на знака);
SBB (изваждане със заем от флага CF);
SUB (изваждане);
XADD (размяна и събиране).

За двоично-десетична аритметика

а) в непакетиран формат:
AAA (след събиране);
AAD (преди деление);
AAM (след умножение);
AAS (след изваждане).

б) в пакетиран формат:
DAA (след събиране);
DAS (след изваждане).

Логически операции (поразрядни)

AND (логическо и);
NOT (отрицание);
OR (логическо или);
TEST (сравняване чрез логическо и);
XOR (логическо изключващо или).

Измествания

RCL (ротация наляво през CF);
RCR (ротация надясно през CF);
ROL (ротация наляво);
ROR (ротация надясно);
SAL (аритметично наляво);
SAR (аритметично надясно);
SHL (логическо наляво);
SHLD ("двойно" изместване наляво, т. е. с извличане от друга данна);
SHR (логическо надясно);
SHRD ("двойно" изместване надясно, т. е. с извличане от друга данна).

Побитови обработки

BSF (търсене на единичен бит напред);
BSR (търсене на единичен бит назад);
BT (извличане);
BTC (извличане и инвертиране);
BTR (извличане и нулиране);
BTS (извличане и записване на единица);
SET# (записва в байт стойността на логическо условие като еднобайтова единица или нула; # замества различните мнемонични съкращения от таблицата на стр. 4).

Празна операция

NOP (само изразходва 1 такт).

Работа с флагове

CLC (нулиране на CF);
CLD (нулиране на DF);
CLI (нулиране на IF; привилегирована);
CMC (инвертиране на CF);
LAHF (младшият байт на FLAGS в AH);
SAHF (AH в младшия байт на FLAGS);
STC (записване на 1 във флага CF);
STD (записване на 1 във флага DF) ;
STI (записване на 1 във флага IF; привилегирована).
Също и POPF, POPFD, PUSHF, PUSHFD.

Предаване на управлението (преходи)

CALL (обръщение към подпрограма);
J# (условен преход; # замества различните мнемонични съкращения от таблицата на стр. 4);
JCXZ (преход при CX=0);
JECXZ (преход при ECX=0);
JMP (безусловен преход);
LOOP (за цикъл по ECX);
LOOPE (за цикъл по ECX и ZF);
LOOPNE (за цикъл по ECX и не ZF);
LOOPNZ (=LOOPNE);
LOOPZ (=LOOPE);
RET (връщане от подпрограма).

Работа с низове (вектори от байтове, думи или двойни думи)

CMPS (изваждане на елементи от низове с цел сравняване);
CMPSB (изваждане за сравняване на байтове от низове);
CMPSD (изваждане за сравняване на двойни думи от низове);
CMPSW (изваждане за сравняване на думи от низове);
INS (въвеждане от порт; влияе се от привилегиите);
INSB (въвеждане от порт на байт; влияе се от привилегиите);
INSD (въвеждане от порт на двойна дума; влияе се от привилегиите);
INSW (въвеждане от порт на дума; влияе се от привилегиите);
LODS (за запис на елемент в акумулатора);
LODSB (за запис на байт в акумулатора AL);
LODSD (за запис на двойна дума в акумулатора EAX);
LODSW (за запис на дума в акумулатора AX);
MOVS (прехвърляне от един към друг низ);
MOVSB (прехвърляне на байт от един към друг низ);
MOVSD (прехвърляне на двойна дума от един към друг низ);
MOVSW (прехвърляне на дума от един към друг низ);
OUTS (записване на един елемент в порт; влияе се от привилегиите);
OUTSB (записване на байт в порт; влияе се от привилегиите);
OUTSD (записване на двойна дума в порт; влияе се от привилегиите);
OUTSW (записване на дума в порт; влияе се от привилегиите);
REP (повторение ECX пъти);
REPE (повторение ECX пъти при ZF);
REPNE (повторение ECX пъти при не ZF);
REPNZ (=REPNE);
REPZ (=REPE);
SCAS (сравнява чрез изваждане акумулатора с елемент на низ);
SCASB (сравнява чрез изваждане на акумулатора AL с байт от низ);
SCASD (сравнява чрез изваждане на акумулатора EAX с двойна дума от низ);
SCASW (сравнява чрез изваждане на акумулатора AX с дума от низ);
STOS (записва акумулатора в елемент на низ) ;
STOSB (записва AL в елемент на низ от байтове) ;
STOSD (записва EAX в елемент на низ от двойни думи);
STOSW (записва AX в елемент на низ от думи).

Проверка на индекс или памет

BOUND (дали индекс е в граници);

VERR (дали сегмент е достъпен за четене);

VERW (дали сегмент е достъпен за запис).

За управление на работата с паметта

LOCK (за сигнал LOOK# за монополно владеене на паметта през следващата команда);

WBINVD (обратен запис и недостоверност на кеш-паметта).

Команди за системно програмиране

ARPL (промяна привилегия);
CLTS (нулиране на флага TS в регистъра CR0);
FWAIT (преди да продължи процесорът проверява за някои особени случаи);
HLT (спиране на процесора до възникване на прекъсване);
IN (въвеждане от порт);
INT (предизвикване на избрано прекъсване);
INTO (предизвикване на прекъсване 4);
INVD (недостоверност на кеш-паметта);
INVLPG (недостоверност на елемент от TLB);
IRET (връщане от прекъсване);
IRETD (=IRET);
LAR (зареждане на права за достъп);
LGDT (зареждане на регистъра GDTR);
LIDT (зареждане на регистъра IDTR);
LLDT (зареждане на регистъра LDTR);
LMSW (зареждане на думата на състоянието);
LTR (зареждане регистъра на задачата);
OUT (извеждане в порт);
SGDT (извличане на съдържанието на регистъра GDTR);
SIDT (извличане на съдържанието на регистъра IDTR);
SLDT (извличане на съдържанието на регистъра LDTR);
SMSW (извличане на думата на състоянието, т. е. на младшата половина на регистъра CR0);
STR (съхраняване на регистъра TR на задачата);
WAIT (преди да продължи процесорът проверява за някои особени случаи).

Също така от привилегиите зависят и командите:
CLI; CLTS; INS; INSB; INSD; INSW; IRET; IRETD; LSL; OUTS; OUTSB; OUTSD; OUTSW; POPFD; STI.

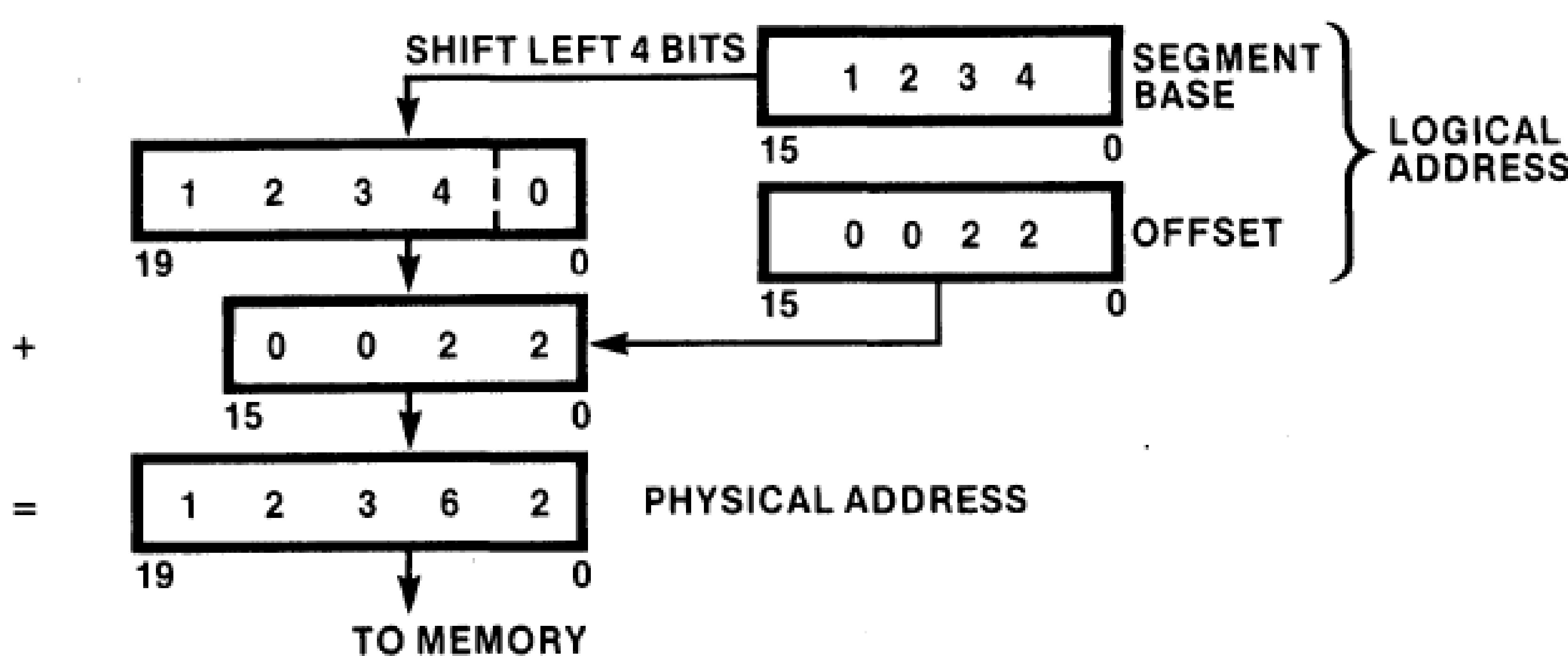


Figure 2-18. Physical Address Generation

Table 2-2. Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

Table 2-1. Default Segment Register Selection Rules

Memory Reference Needed	Segment Register Used	Implicit Segment Selection Rule
Instructions	Code (CS)	Automatic with instruction prefetch
Stack	Stack (SS)	All stack pushes and pops. Any memory reference that uses ESP or EBP as a base register.
Local Data	Data (DS)	All data references except when relative to stack or string destination.
Destination Strings	Extra (ES)	Destination of string instructions.

Special instruction prefix elements may be used to override the default segment selection. Segment-override prefixes allow an explicit segment selection. The 80386 has a segment-override prefix for each of the segment registers. Only in the following special cases is there an implied segment selection that a segment prefix cannot override:

- The use of ES for destination strings in string instructions.
- The use of SS in stack instructions.
- The use of CS for instruction fetches.

*) 8- or 16-bit, signed

Table 2-3. Memory Operand Addressing Modes

Addressing Mode	Offset Calculation
Direct	16-bit Displacement in the instruction
Register Indirect	BX, SI, DI
Based	(BX or BP) + Displacement*
Indexed	(SI or DI) + Displacement*
Based Indexed	(BX or BP) + (SI or DI)
Based Indexed + Displacement	(BX or BP) + (SI or DI) + Displacement*

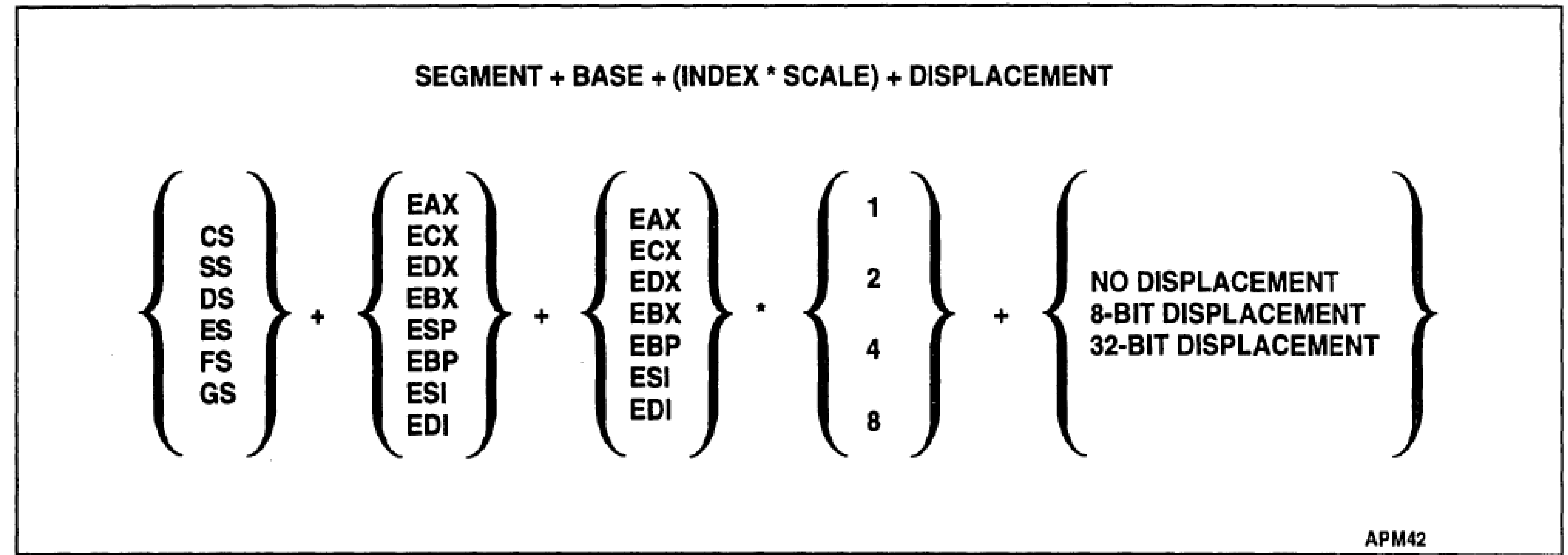


Figure 3-10. Effective Address Computation

The scaling factor permits efficient indexing into an array when the array elements are 2, 4, or 8 bytes. The scaling of the index register is done in hardware at the time the address is evaluated. This eliminates an extra shift or multiply instruction.

The base, index, and displacement components can be used in any combination; any of these components can be null. A scale factor can be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. Suggested uses for some combinations of address components are described below.

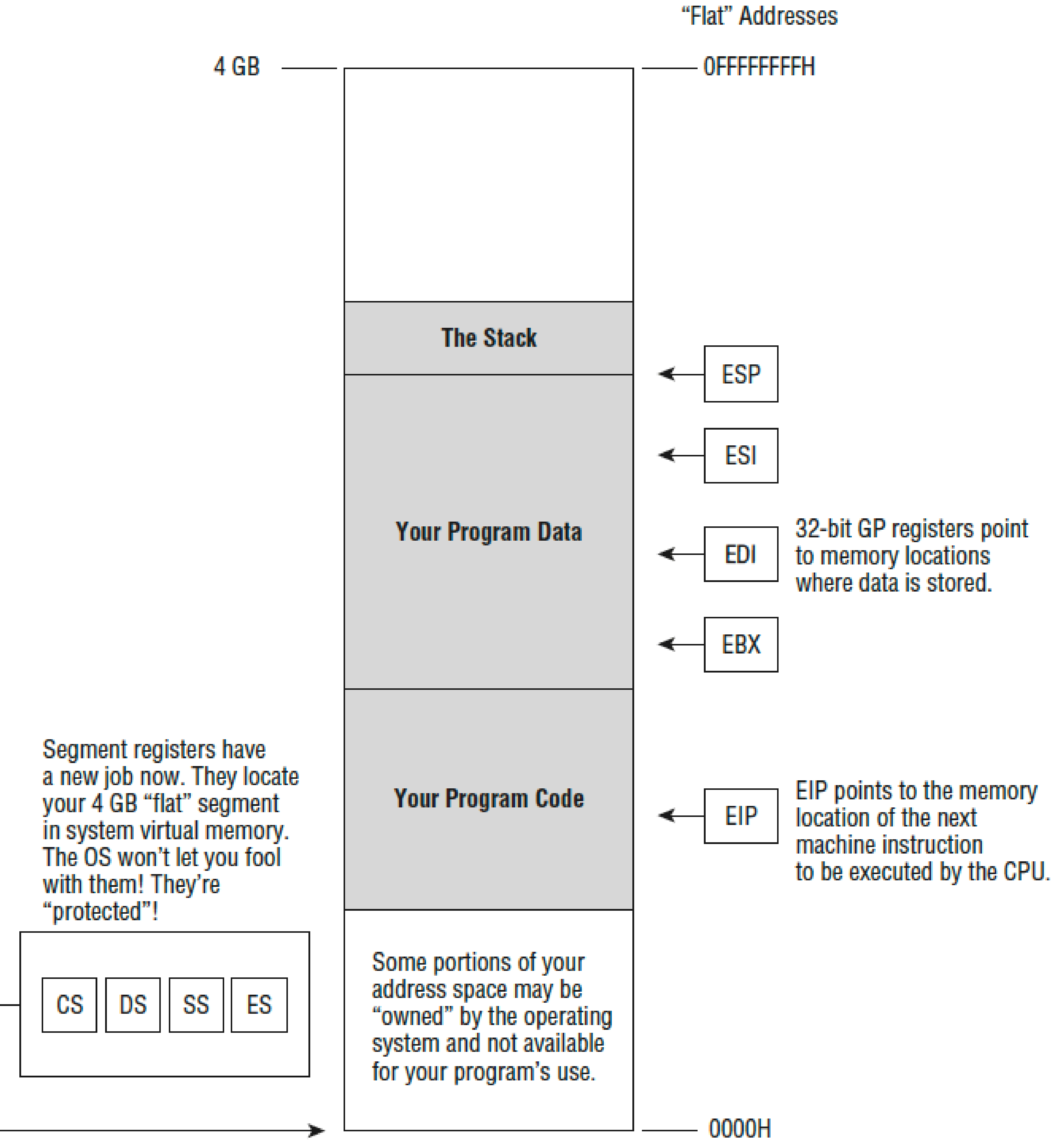


Figure 4-10: Protected mode flat model

13.2. MIXING 16-BIT AND 32-BIT OPERATIONS WITHIN A CODE MODULE

The following two instruction prefixes allow mixing of 32-bit and 16-bit operations within one segment:

- The operand-size prefix (66H)
- The address-size prefix (67H)

These prefixes reverse the default size selected by the D flag in the code-segment descriptor. For example, the processor can interpret the (MOV *mem*, *reg*) instruction in any of four ways:

- In a 32-bit code segment:
  - Moves 32 bits from a 32-bit register to memory using a 32-bit effective address.
  - If preceded by an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by an address-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
- In a 16-bit code segment:
  - Moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
  - If preceded by an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by an address-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 32-bit effective address.

CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Clocks	Description
98	CBW	3	AX ← sign-extend of AL
98	CWDE	3	EAX ← sign-extend of AX

Operation

```
IF OperandSize = 16 (* instruction = CBW *)
THEN AX ← SignExtend(AL);
ELSE (* OperandSize = 32, instruction = CWDE *)
   EAX ← SignExtend(AX);
FI;
```

Description

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH. CWDE converts the signed word in AX to a doubleword in EAX by extending the most significant bit of AX into the two most significant bytes of EAX. Note that CWDE is different from CWD. CWD uses DX:AX rather than EAX as a destination.

Flags Affected

None

AAA – ASCII Adjust after Addition

Корекция след събиране (след ADD) (за 2-10-ична аритметика с непакетиран формат). Няма явен операнд. По подразбиране работи с регистъра ах по следния начин:

```
if ( ( AL and 0fH ) > 9 ) or ( AF = 1 )
then AL <-- ( AL + 6 ) and 0fH
     AH <-- AH + 1;
     AF <-- 1
     CF <-- 1
else CF <-- 0
     AF <-- 0
endif
```

AAS – ASCII Adjust after Subtraction

Корекция след изваждане (след SUB) (за 2-10-ична аритметика с непакетиран формат). Няма явен операнд. По подразбиране работи с регистъра ах по следния начин:

```
if ( ( AL and 0fH ) > 9 ) or ( AF = 1 )
then AL <-- AL - 6
     AL <-- AL and 0fH
     AH <-- AH -1
     AF <-- 1
     CF <-- 1
else CF <-- 0
     AF <-- 0
endif
```

DAA – Decimal Adjust AL after Addition

Десетична корекция на AL след събиране. (За 2-10-ична аритметика с пакетиран формат.) Няма явен операнд. По подразбиране работи с регистъра ах по следния начин:

```
if ( ( AL and 0fH ) > 9 ) or ( AF = 1 )
then AL <-- ( AL + 6 )
     AF <-- 1
else AF <-- 0
endif
if ( AL > 9fH ) or ( CF = 1 )
then AL <-- AL + 60H
     CF <-- 1
else CF <-- 0
endif
```

DAS – Decimal Adjust AL after Substraction

Десетична корекция на AL след изваждане. (За 2-10-ична аритметика с пакетиран формат.) Няма явен операнд. По подразбиране работи с регистъра ах по следния начин:

```
if ( ( AL and 0fH ) > 9 ) or ( AF = 1 )
then AL <-- ( AL - 6 )
     AF <-- 1
else AF <-- 0
endif
if ( AL > 9fH ) or ( CF = 1 )
then AL <-- AL - 60H
     CF <-- 1
else CF <-- 0
endif
```

AAM—ASCII Adjust AX after Multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	18	ASCII adjust AX after multiply

Operation

regAL ← AL;
AH ← regAL / imm8;
AL ← regAL MOD imm8;

NOTE:

imm8 has the value of the instruction's second byte. The second byte under normally assembly of this instruction will be 0A, however, explicit modification of this byte will result in the operation described above and may alter results.

Description

Execute the AAM instruction only after executing a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. The AAM instruction unpacks the AL result by dividing AL by the second byte of the opcode, leaving the quotient (most-significant digit) in the AH register and the remainder (least-significant digit) in the AL register.

Flags Affected

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

AAD—ASCII Adjust AX before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	10	ASCII adjust AX before division

Operation

```
regAL = AL;
regAH = AH;
AL ← (regAH * imm8 + regAL) AND 0FFH;
AH ← 0;
```

NOTE:

imm8 has the value of the instruction's second byte. The second byte under normally assembly of this instruction will be 0A, however, explicit modification of this byte will result in the operation described above and may alter results.

Description

The AAD instruction is used to prepare two unpacked BCD digits (the least-significant digit in the AL register, the most-significant digit in the AH register) for a division operation that will yield an unpacked result. This is accomplished by setting the AL register to AL + (second byte of opcode * AH), and then clearing the AH register. The AX register is then equal to the binary equivalent of the original unpacked two-digit number.

Flags Affected

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

IA32 Instruction List (Short Form)

Description of Operands

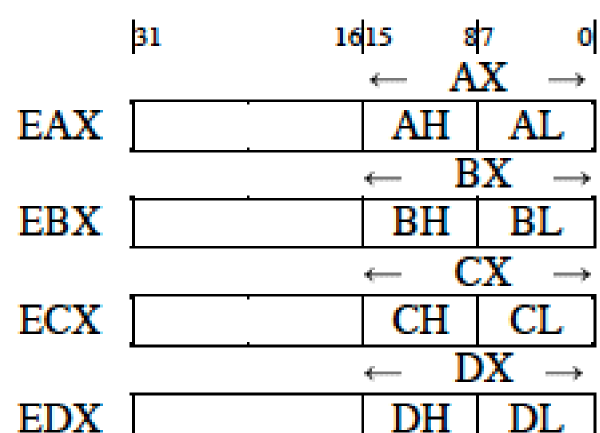
Operand	Description
r8	8-bit general purpose register
r16	16-bit general purpose register
r32	16-bit general purpose register
EDX:EAX	64-bit integer number, EDX – more significant part, EAX – less significant part
imm8	immediate 8-bit value from –128 to 127
imm16	immediate 16-bit value from –32768 to +32767
imm32	immediate 32-bit value from –2147483648 to +2147483647
r/m8	8-bit general purpose register or memory location
r/m16	16-bit general purpose register or memory location
r/m32	32-bit general purpose register or memory location
m	16-bit or 32-bit memory location
m8	8-bit memory location
m16	16-bit memory location
m32	32-bit memory location
m64	64-bit memory location
m128	128-bit memory location
m*N*byte	*N*-byte memory location

Operand	Description
m16:16	a memory location containing a far pointer composed of two 16-bit numbers: segment & offset
m16:32	a memory location containing a far pointer composed of numbers: 16-bit segment & 32-bit offset
m16&16	a memory location containing a data pair: 16&16-bit
m16&32	a memory location containing a data pair: 16&32-bit
m32&32	a memory location containing a data pair: 32&32-bit
moffs8	simple 8-bit memory location, which actual address is given by a simple offset relative to segment base
moffs16	simple 16-bit memory location, which actual address is given by a simple offset relative to segment base
moffs32	simple 32-bit memory location, which actual address is given by a simple offset relative to segment base
Sreg	segment register: CS, DS, SS, ES, FS or GS
rel8	relative address in the range from 128 bytes before to 127 bytes after the end of instruction
rel16	16-bit relative address within the same code segment
rel32	32-bit relative address within the same code segment

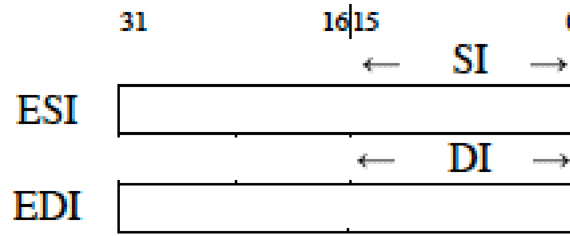
Operand	Description
ptr16:16	16-bir far pointer in a different code segment
ptr32:32	32-bir far pointer in a different code segment
m32fp	a single-precision floating-point memory location
m64fp	a double-precision floating-point memory location
m80fp	an extended-precision floating-point memory location
m16int	a word integer memory location
m32int	a double-word (dword) integer memory location
m64int	a quad-word (qword) integer memory location
ST	the top element of the FPU register stack
ST(0)	the top element of the FPU register stack
ST(i)	the i-th element of the FPU register stack (i←0..7)
mm	64-bit MMX register from MM0 to MM7
mm/m32	low order 32 bits of an MMX register or 32-bit memory location
mm/m64	MMX register or 64-bit memory location
xmm	128-bit XMM register from XMM0 to XMM7
xmm/m32	XMM register or 32-bit memory location
xmm/m64	XMM register or 64-bit memory location
xmm/m128	XMM register or 128-bit memory location

Register Set

General Purpose Registers



Index Registers



Pointer Registers

31 16 15 0

BP

EBP

SP

ESP

Instruction Pointer

31 16 15 0

IP

EIP

Segment Registers

15 0

CS

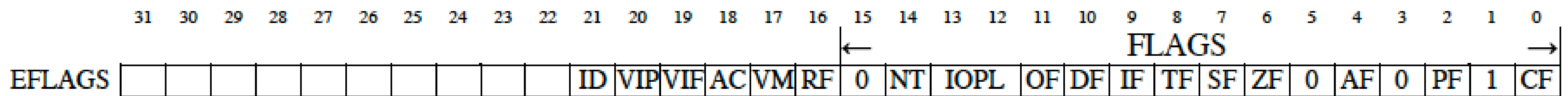
DS

ES

FS

GS

Flags



Bit		Flag	Description
0	CF	Carry Flag	Carry from most significant bit, also borrow for most significant bit; can be considered as overflow in unsigned instructions.
2	PF	Parity Flag	Set to 1 if 8 less significant bits of result have even number of 1's, else set to 0.
4	AF	Auxiliary carry Flag	Used as carry flag in BCD instructions.
6	ZF	Zero Flag	Set to 1 if result is zero, else set to 0.
7	SF	Sign Flag	Set to 1 if result is negative (below zero), else set to 0.
8	TF	Trap Flag	Used by debuggers.
9	IF	Interrupt Flag	If set to 1, then interrupts are enabled, else are disabled.
10	DF	Direction Flag	When set to 0, string instructions increment the index registers, else – decrement the index registers.
11	OF	Overflow Flag	Used in signed instructions.
12,13	IOPL	I/O Privilege Level	Indicates the I/O privilege level of the currently running program or task.
14	NT	Nested Task	Controls the chaining of interrupt and called tasks.
16	RF	Resume Flag	Controls the processor's response to instruction-breakpoint conditions.
17	VM	Virtual 8086 Mode	Set to enable virtual-8086 mode; clear to return to protected mode.
18	AC	Alignment check	Set this flag and the AM flag in control register CR0 to enable alignment checking of memory references.
19	VIF	Virtual Interrupt Flag	Contains a virtual image of the IF flag.
20	VIP	Virtual Interrupt Pending	Set by software to indicate that an interrupt is pending.
21	ID	Identification	The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

CPU Instruction set

Data Transfer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Move	MOV	r/m8, r8	Move *r8* to *r/m8*.	DST ← SRC
		r/m16, r16	Move *r16* to *r/m16*.	
		r/m32, r32	Move *r32* to *r/m32*.	
		r8, r/m8	Move *r/m8* to *r8*.	
		r16, r/m16	Move *r/m16* to *r16*.	
		r32, r/m32	Move *r/m32* to *r32*.	
		r/m16, Sreg	Move segment register to *r/m16*.	
		Sreg, r/m16	Move *r/m16* to segment register.	
		AL, moffs8	Move byte at (segment: offset) to AL.	
		AX, moffs16	Move word at (segment: offset) to AX.	
		EAX, moffs32	Move dword at (segment: offset) to EAX.	
		moffs8, AL	Move AL to byte at (segment: offset).	
		moffs16, AX	Move AX to word at (segment: offset).	
		moffs32, EAX	Move EAX to dword at (segment: offset).	
		r8, imm8	Move *imm8* to *r8*.	
		r16, imm16	Move *imm16* to *r16*.	
		r32, imm32	Move *imm32* to *r32*.	
		r/m8, imm8	Move *imm8* to *r/m8*.	
		r/m16, imm16	Move *imm16* to *r/m16*.	
		r/m32, imm32	Move *imm32* to *r/m32*.	
Conditional Move	CMOVA	r16, r/m16	Move if above (CF=0 and ZF=0)	TMP ← SRC;
	CMOVAE	r32, r/m32	Move if above or equal (CF=0)	IF (condition) THEN
	CMOVB		Move if below (CF=1)	DST ← TMP
	CMOVBE		Move if below or equal (CF=1 or ZF=1)	END
	CMOVC		Move if carry (CF=1)	
	CMOVE		Move if equal (ZF=1)	
	CMOVG		Move if greater (ZF=0 and SF=OF)	
	CMOVGE		Move if greater or equal (SF=OF)	
	CMOVL		Move if less (SF<>OF)	
	CMOVLE		Move if less or equal (ZF=1 or SF<>OF)	
	CMOVNA		Move if not above (CF=1 or ZF=1)	
	CMOVNAE		Move if not above or equal (CF=1)	
	CMOVNB		Move if not below (CF=0)	
	CMOVNBE		Move if not below or equal (CF=0 and ZF=0)	
	CMOVNC		Move if not carry (CF=0)	
	CMOVNE		Move if not equal (ZF=0)	
	CMOVNG		Move if not greater (ZF=1 or SF<>OF)	
	CMOVNGE		Move if not greater or equal (SF<>OF)	
	CMOVNL		Move if not less (SF=OF)	
	CMOVNLE		Move if not less or equal (ZF=0 and SF=OF)	
	CMOVNO		Move if not overflow (OF=0)	

	CMOVNP CMOVNS CMOVNZ CMOVO CMOVP CMOVPE CMOVPO CMOVS CMOVZ		Move if not parity (PF=0) Move if not sign (SF=0) Move if not zero (ZF=0) Move if overflow (OF=1) Move if parity (PF=1) Move if parity even (PF=1) Move if parity odd (PF=0) Move if sign (SF=1) Move if zero (ZF=1)	
Exchange	XCHG	AX, r16 r16, AX EAX, r32 r32, EAX r/m8, r8 r8, r/m8 r/m16, r16 r16, r/m16 r/m32, r32 r32, r/m32	Exchanges the contents of the register with other register or memory location.	TMP ← DST; DST ← SRC; SRC ← TMP
Byte Swap	BSWAP	r32	Reverses the byte order of a 32-bit register.	TMP ← DST; DST[7..0] ← TMP [31..24]; DST[15..8] ← TMP[23..16]; DST[23..16] ← TMP[15..8]; DST[31..24] ← TMP[7..0];
Exchange and ADD	XADD	r/m8, r8 r/m16, r16 r/m32, r32	Exchanges source and destination operands. Load sum into destination operand.	TMP ← SRC + DST SRC ← DST DST ← TMP

Compare and Exchange	CMPXCHG	r/m8, r8
		r/m16, r16
		r/m32, r32
			Compares the accumulator (AL, AX or EAX) with the first operand. If equal ZF is set and the second operand is loaded into the first operand. Else, clears ZF and loads the first operand into the accumulator.	IF ACC = DST THEN
				ZF ← 1; DST ← SRC
				ELSE
				ZF ← 0; ACC ← DST
				END

Compare and Exchange 8 Bytes	CMPXCHG8B	m64	Compares EDX:EAX with the operand. If equal ZF is set and ECX:EBX is loaded into the operand. Else, clears ZF and loads the operand into the EDXLEAX.	IF EDX:EAX = DST THEN
				ZF ← 1; DST ← ECX:EBX
				ELSE
				ZF ← 0; EDX:EAX ← DST
				END

Push onto Stack	PUSH	r/m16 r/m32 imm8 imm16 imm32 DS ES SS FS GS	Decrements stack pointer. Pushes register, memory or immediate value to the top of stack into register or memory, increment stack pointer.	ESP ← ESP – OPERANDSIZE/8; SS:[ESP] ← SRC

Push All general registers	PUSHA		Pushes AX, CX, DX, BX, original SP, BP, SI, and DI.	TMP ← (E)SP; PUSH (E)AX; PUSH (E)CX; PUSH (E)DX; PUSH (E)BX; PUSH TMP; PUSH (E)BP; PUSH (E)SI; PUSH (E)DI
	PUSHAD		Pushes EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.	
Pop from stack	POP	r/m16 r/m32 DS ES SS FS GS	Pops top of stack into register or memory, increments stack pointer.	DST ← SS:[ESP]; ESP ← ESP + OPERANDSIZE/8
Pop All general registers	POPA		Pops AX, CX, DX, BX, original SP, BP, SI, and DI.	POP (E)DI; POP (E)SI; POP (E)BP; ESP ← ESP + OPERANDSIZE/8; POP (E)BX; POP (E)DX; POP (E)CX; POP (E)AX
	POPAD		Pops EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.	
Input from port	IN	AL, imm8 AX, inn8 EAX, imm8 AL, DX AX, DX EAX, DX	Inputs byte from given I/O port address into AL. Inputs word from given I/O port address into AX. Inputs dword from given I/O port address into EAX. Inputs byte from I/O port specified in DX into AL. Inputs word from I/O port specified in DX into AX. Inputs dword from I/O port specified in DX into EAX.	DST ← Port(SRC)
Output from port	OUT	imm8, AL inn8, AX imm8, EAX DX, AL DX, AX DX, EAX	Outputs byte from AL to given I/O port address. Outputs word from AX to given I/O port address. Outputs dword from EAX to given I/O port address. Outputs byte from AL to I/O port specified in DX. Outputs word from AX to I/O port specified in DX. Outputs dword from EAX to I/O port specified in DX.	Port(DST) ← SRC
Convert Word to Dword	CWD		Sign-extends AX to DX:AX	DX:AX ← SignExtend(AX)
Convert Dword to Qword	CDQ		Sign-extends EAX to EDX:EAX	EDX:EAX ← SignExtend(EAX)
Move with Zero-Extend	MOVZX	r16, r/m8 r32, r/m8 r32, r/m16	Move byte to word with zero-extension. Move byte to dword with zero-extension. Move word to dword with zero-extension.	DST ← ZeroExtend(SRC)
Move with Sign-Extend	MOVSX	r16, r/m8 r32, r/m8 r32, r/m16	Move byte to word with sign-extension. Move byte to dword with sign-extension. Move word to dword with sign-extension.	DST ← SignExtend(SRC)

Binary Arithmetic Instructions

<table>
<tr><th>Instruction</th><th>Mnemonic</th><th>Operands</th><th>Description</th><th>Symbolic operations</th></tr>
<tr><td>Increment</td><td>INC</td><td>r/m8
r/m16
r/m32</td><td>Adds 1 to the destination operand, while preserving the state of the CF flag.</td><td>DST ← DST + 1;
SET EFLAGS.OF, .SF, .ZF, .AF, ..PF</td></tr>
<tr><td>Decrement</td><td>DEC</td><td>r/m8
r/m16
r/m32</td><td>Subtracts 1 from the destination operand, while preserving the state of the CF flag.</td><td>DST ← DST – 1;
SET EFLAGS.OF, .SF, .ZF, .AF, ..PF</td></tr>
<tr><td>Arithmetic Negation</td><td>NEG</td><td>r/m8
r/m16
r/m32</td><td>Two's complement negation of the operand.</td><td>IF DST=0 THEN EFLAGS.CF ← 0
ELSE EFLAGS.CF ← 1;
DST ← – DST;
SET EFLAGS.OF, .SF, .ZF, .AF, .PF</td></tr>
<tr><td>Add</td><td>ADD</td><td rowspan="6">AL, imm8
AX, imm16
EAX, imm32
r/m8, imm8
r/m16, imm16
r/m32, imm32
r/m16, imm8
r/m32, imm8
r/m8, r8
r/m16, r16
r/m32, r32
r8, r/m8
r16, r/m16
r32, r/m32</td><td>Adds source (second) operand to the destination (first) operand.</td><td>DST← DST + SRC;
SET EFLAGS.OF, .SF, .ZF, .AF, .CF, ..PF</td></tr>
<tr><td>Add with Carry</td><td>ADC</td><td>Adds source (second) operand with CF to the destination (first) operand.</td><td>DST← DST + SRC +EFLAGS.CF;
SET EFLAGS.OF, .SF, .ZF, .AF, .CF, ..PF</td></tr>
<tr><td>Subtract</td><td>SUB</td><td>Subtracts source (second) operand from the destination (first) operand.</td><td>DST← DST – SRC;
SET EFLAGS.OF, .SF, .ZF, .AF, .CF, ..PF</td></tr>
<tr><td>Subtract with Borrow</td><td>SBB</td><td>Subtracts source (second) operand with CF from the destination (first) operand.</td><td>DST← DST – (SRC +EFLAGS.CF);
SET EFLAGS.OF, .SF, .ZF, .AF, .CF, ..PF</td></tr>
<tr><td>Compare</td><td>CMP</td><td>Compares two operands by subtracting the second operand from the first operand and then setting the status flag in the same manner as the SUB instruction.</td><td>TMP ← SRC1 – SIGNEXTEND(SRC2);
SET EFLAGS.OF, .SF, .ZF, .AF, .CF, ..PF</td></tr>
<tr><td rowspan="3">Unsigned Multiply</td><td rowspan="3">MUL</td><td>r/m8</td><td>Multiplies unsigned AL by r/m8. Stores result in AX.</td><td>AX ← AL * SRC;
IF AH=0 THEN EFLAGS.OF, .CF ← 00B;
ELSE EFLAGS.OF, .CF ← 11B;
// EFLAGS. .ZF, .AF, .PF., SF are undefined</td></tr>
<tr><td>r/m16</td><td>Multiplies unsigned AX by r/m16. Stores result in DX:AX.</td><td>DX:AX ← AX * SRC;
IF DX=0 THEN EFLAGS.OF, .CF ← 00B;
ELSE EFLAGS.OF, .CF ← 11B;
// EFLAGS. .ZF, .AF, .PF., SF are undefined</td></tr>
<tr><td>r/m32</td><td>Multiplies unsigned EAX by r/m32. Stores result in EDX:EAX.</td><td>EDX:EAX ← EAX * SRC;
IF EDX=0 THEN EFLAGS.OF, .CF ← 00B;
ELSE EFLAGS.OF, .CF ← 11B;
// EFLAGS. .ZF, .AF, .PF., SF are undefined</td></tr>
<tr><td rowspan="3">Unsigned Divide</td><td rowspan="3">DIV</td><td>r/m8</td><td>Divides unsigned AX by r/m8. Stores result in AL, remainder in AH.</td><td>AL ← AX / SRC; AH ← AX MOD SRC;
// EFLAGS.CF, .OF, .ZF, .AF, .PF., SF are undefined</td></tr>
<tr><td>r/m16</td><td>Divides unsigned DX:AX by r/m16. Stores result in AX, remainder in DX.</td><td>AX ← DX:AX / SRC; DX ← DX:AX MOD SRC;
// EFLAGS.CF, .OF, .ZF, .AF, .PF., SF are undefined</td></tr>
<tr><td>r/m32</td><td>Divides unsigned EDX:EAX by r/m32. Stores result in EAX, remainder in EDX.</td><td>EAX ← EDX:EAX / SRC; EDX ← EDX:EAX MOD SRC;
// EFLAGS.CF, .OF, .ZF, .AF, .PF., SF are undefined</td></tr>
</table>

Signed Multiply	IMUL	r/m8	Multiplies signed AL by r/m8. Stores result in AX.	AX ← AL * SRC; IF AX=AL THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., .SF are undefined*
		r/m16	Multiplies signed AX by r/m16. Stores result in DX:AX.	DX:AX ← AX * SRC; IF DX:AX=SignExtend(AX) THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., .SF are undefined*
		r/m32	Multiplies signed EAX by r/m32. Stores result in EDX:EAX.	EDX:EAX ← EDX * SRC; IF EDX:EAX=EAX THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., .SF are undefined*
		r16, r/m16	Multiplies signed word register by r/m16 word.	TMP ← DST * SRC; *// TMP is double DST size* DST ← DST * SRC; IF TMP=DST THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., .SF are undefined*
		r32, r/m32	Multiplies signed dword register by r/m32 dword.	
		r16, imm8	Multiplies signed word register by sign-extend imm8 value.	
		r32, imm8	Multiplies signed dword register by sign-extend imm8 value.	
		r16, imm16	Multiplies signed word register by sign-extend imm8 value.	
		r32, imm32	Multiplies signed dword register by sign-extend imm8 value.	
		r16, r/m16, imm8	Multiplies signed r/m16 word by sign-extend imm8 value. Stores result in word register.	TMP ← SRC1 * SRC2; *// TMP is double SRC1 size* DST ← SRC1 * SRC2 IF TMP=DST THEN EFLAGS.CF, .OF ← 00B; ELSE EFLAGS.CF, .OF ← 11B; *// EFLAGS. .ZF, .AF, .PF., .SF are undefined*
		r32, r/m32, imm8	Multiplies signed r/m32 dword by sign-extend imm8 value. Stores result in dword register.	
		r16, r/m16, imm16	Multiplies signed r/m16 word by imm16 value. Stores result in word register.	
		r32, r/m32, imm32	Multiplies signed r/m32 dword by imm16 value. Stores result in dword register.	
Signed Divide	IDIV	r/m8	Divides signed AX by r/m8. Stores result in AL, remainder in AH.	AL ← AX / SRC; AH ← AX MOD SRC; *// EFLAGS.CF, .OF, .ZF, .AF, .PF are undefined*
		r/m16	Divides signed DX:AX by r/m16. Stores result in AX, remainder in DX.	AX ← DX:AX / SRC; DX ← DX:AX MOD SRC; *// EFLAGS.CF, .OF, .ZF, .AF, .PF, .SF are undefined*
		r/m32	Divides signed EDX:EAX by r/m32. Stores result in EAX, remainder in EDX.	EAX ← EDX:EAX / SRC; EDX ← EDX:EAX MOD SRC; *// EFLAGS.CF, .OF, .ZF, .AF, .PF, .SF are undefined*

Decimal Arithmetic Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Decimal Adjust AL after Addition	DAA		Adjust AL after BCD4 addition.	
Decimal Adjust AL after Subtraction	DAS		Adjust AL after BCD4 subtraction.	

ASCII Adjust after Addition	AAA		Adjust AL after decimal addition.	IF ((AL AND 0FH)>9) OR (AF=1) THEN
				AL ← AL + 6; AH ← AH + 1; AF ← 1; CF ← 1;
				ELSE
				CF ← 0; AF ← 0;
				END
				AL ← AL AND 0FH

ASCII Adjust after Subtraction	AAS		Adjust AL after decimal addition.	IF ((AL AND 0FH)>9) OR (AF=1) THEN
				AL ← AL – 6; AH ← AH – 1; AF ← 1; CF ← 1;
				ELSE
				CF ← 0; AF ← 0;
				END
				AL ← AL AND 0FH

ASCII Adjust before Division	AAD		Adjust AZ before decimal division.	TMPAL ← AL; TMPAH ← AH; AH ← (TMPAL + (TMPAH * 10); AH ← 0
ASCII Adjust after Multiplication	AAM		Adjust AZ after decimal multiplication.	TMPAL ← AL; AH ← TMPAL / 10; AL ← TMPAL MOD 10

Logical Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Logical Negation	NOT	r/m8 r/m16 r/m32	Reverses each bit of the operand	DST ← NOT SRC; *// EFLAGS.CF, .OF, .ZF, .AF, .PF are not affected*
Logical AND	AND	AL, imm8 AX, imm16 EAX, imm32	Performs a bitwise AND operation on the destination (first) and source (second) operands and stored the result in the destination operand location.	DST ← DST AND SRC; EFLAGS.OF, .CF ← 00B; SET EFLAGS.SF, .ZF, .PF *// EFLAGS.AF is undefined*
Logical Inclusive OR	OR	r/m8, imm8 r/m16, imm16 r/m32, imm32	Performs a bitwise OR operation on the destination (first) and source (second) operands and stored the result in the destination operand location.	DST ← DST OR SRC; EFLAGS.OF, .CF ← 00B; SET EFLAGS.SF, .ZF, .PF *// EFLAGS.AF is undefined*
Logical Exclusive OR	XOR	r/m16, imm8 r/m32, imm8 r/m8, r8 r/m16, r16 r/m32, r32 r8, r/m8 r16, r/m16 r32, r/m32	Performs a bitwise XOR operation on the destination (first) and source (second) operands and stored the result in the destination operand location.	DST ← DST OR SRC; EFLAGS.OF, .CF ← 00B; SET EFLAGS.SF, .ZF, .PF *// EFLAGS.AF is undefined*

Shift and Rotate Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Shift Left	SHL	r/m8 r/m8, CL r/m8, imm8	Shift register bits left by 1 position. Shift register bits left by 1 position., CL times. Shift register bits left by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← ShiftLeft (DST) NEXT
Shift Right	SHR	r/m16 r/m16, CL r/m16, imm8	Shift register bits right by 1 position. Shift register bits right by 1 position, CL times. Shift register bits right by 1 position, *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← ShiftRight (DST) NEXT
Shift Arithmetic Left	SAL	r/m32 r/m32, CL r/m32, imm8	Multiply signed register by 2. Multiply signed register by 2, CL times. Multiply signed register by 2, *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← DST * 2 NEXT

Shift Arithmetic Right	SAR		Divide signed register by 2. Divide signed register by 2, CL times. Divide signed register by 2, *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← DST / 2 NEXT
Rotate Left	ROL	r/m8 r/m8, CL r/m8, imm8	Rotate register bits left by 1 position. Rotate register bits left by 1 position., CL times. Rotate register bits left by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← RotateLeft (DST) NEXT
Rotate Right	ROR	r/m16 r/m16, CL r/m16, imm8	Rotate register bits right by 1 position. Rotate register bits right by 1 position., CL times. Rotate register bits right by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← RotateRight (DST) NEXT
Rotate thru Carry Left	RCL	r/m32 r/m32, CL r/m32, imm8	Rotate register bits and CF flag left by 1 position. Rotate register bits and CF flag left by 1 position., CL times. Rotate register bits and CF flag left by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH (DST,CF) ← RotateLeft (DST,CF) NEXT
Rotate thru Carry Right	RCR		Rotate register bits and CF flag right by 1 position. Rotate register bits and CF flag right by 1 position., CL times. Rotate register bits and CF flag right by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH (DST,CF) ← RotateRight (DST,CF) NEXT
Shift Left Double	SHLD	r/m16, r16, imm8 r/m16, r16, CL r/m32, r32, imm8 r/m32, r32, CL	Shift first register to left by *imm8*/ CL places while shifting bits from *r16* in from the right.	TMP ← DTS2; FOR i←1 TO COUNT AND 1FH DST1 ← ShiftLeft (DST1, MSB (TMP)); TMP ← ShiftLeft (TMP); NEXT
Shift Right Double	SHRD		Shift first register to right by *imm8*/ CL places while shifting bits from *r16* in from the left.	TMP ← DTS2; FOR i←1 TO COUNT AND 1FH DST1 ← ShiftRight (LSB (TMP), DST1); TMP ← ShiftRight (TMP); NEXT

Bit and Byte Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Bit Test	BT	r/m16,r16 r/m32,r32 r/m16,r16 r/m16,imm8 r/m32,imm8	Selects the bit in a bit string (specified with the first operand called the bit base) at the bit position designated by the bit offset operand (second operand) and stores the value of the bit in CF flag.	CF ← BIT (bit-base, bit-offset)
Bit Test and Complement	BTC	r/m16,r16 r/m32,r32 r/m16,r16 r/m16,imm8 r/m32,imm8	Selects the bit in a bit string (specified with the first operand called the bit base) at the bit position designated by the bit offset operand (second operand) , stores the value of the bit in CF flag, and complements the selected bit in the bit string.	CF ← BIT (bit-base, bit-offset); BIT (bit-base, bit-offset) ← NOT BIT (bit-base, bit-offset)
Bit Test and Reset	BTR	r/m16,r16 r/m32,r32 r/m16,r16 r/m16,imm8 r/m32,imm8	Selects the bit in a bit string (specified with the first operand called the bit base) at the bit position designated by the bit offset operand (second operand) , stores the value of the bit in CF flag, and clears the selected bit to 0.	CF ← BIT (bit-base, bit-offset); BIT (bit-base, bit-offset) ← 0

Bit Test and Set	BTS	r/m16,r16 r/m32,r32 r/m16,r16 r/m16,imm8 r/m32,imm8	Selects the bit in a bit string (specified with the first operand called the bit base) at the bit position designated by the bit offset operand (second operand) , stores the value of the bit in CF flag, and sets the selected bit to 1.	CF ← BIT (bit-base, bit-offset); BIT (bit-base, bit-offset) ← 1

Bit Scan Forward	BSF	r16, r/m16
		r32, r/m32
			Searches the source (second) operand for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination (first) operand.	IF SRC = 0 THEN
				ZF ← 1; *// DST is undefined;*
				ELSE
				ZF ← 0; TMP ← 0;
				WHILE BIT (SRC, TMP) = 0 DO
				TMP ← TMP + 1; DST ← TMP;
				END
				END

Bit Scan Reverse	BSR	r16, r/m16
		r32, r/m32
			Searches the source (second) operand for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination (first) operand.	IF SRC = 0 THEN
				ZF ← 1; *// DST is undefined;*
				ELSE
				ZF ← 0; TMP ← OPERANDSIZE;
				WHILE BIT (SRC, TMP) = 0 DO
				TMP ← TMP – 1; DST ← TMP;
				END
				END

Conditional Set Byte	SETA SETAE SETB SETBE SETC SETE SETG SETGE SETL SETLE SETNA SETNAE SETNB SETNBE SETNC SETNE SETNG SETNGE SETNL SETNLE SETNO SETNP SETNS SETNZ SETO SETP	r/m8	Set byte if above (CF=0 and ZF=0) Set byte if above or equal (CF=0) Set byte if below (CF=1) Set byte if below or equal (CF=1 or ZF=1) Set byte if carry (CF=1) Set byte if equal (ZF=1) Set byte if greater (ZF=0 and SF=OF) Set byte if greater or equal (SF=OF) Set byte if less (SF<>OF) Set byte if less or equal (ZF=1 or SF<>OF) Set byte if not above (CF=1 or ZF=1) Set byte if not above or equal (CF=1) Set byte if not below (CF=0) Set byte if not below or equal (CF=0 and ZF=0) Set byte if not carry (CF=0) Set byte if not equal (ZF=0) Set byte if not greater (ZF=1 or SF<>OF) Set byte if not greater or equal (SF<>OF) Set byte if not less (SF=OF) Set byte if not less or equal (ZF=0 and SF=OF) Set byte if not overflow (OF=0) Set byte if not parity (PF=0) Set byte if not sign (SF=0) Set byte if not zero (ZF=0) Set byte if overflow (OF=1) Set byte if parity (PF=1)	IF (condition) THEN DST ← 1 ELSE DST ← 0

	SETPE SETPO SETS SETZ		Set byte if parity even (PF=1) Set byte if parity odd (PF=0) Set byte if sign (SF=1) Set byte if zero (ZF=1)	
Logical Compare	TEST	AL, imm8 AX, imm16 EAX, imm32 r/m8, imm8 r/m16, imm16 r/m32, imm32 r/m8, r8 r/m16, r16 r/m32, r32	Performs a bitwise AND operation on the destination (first) and source (second) operands. Result is not stored, but flags are affected.	TMP ← DST AND SRC; EFLAGS.OF, .CF ← 00B; SET EFLAGS.SF, .ZF, .PF *// EFLAGS.AF is undefined*

Control Transfer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Jump	JMP	rel8 rel16 rel32	Jumps near, relative, displacement relative to the next instruction	(E)IP ← (E)IP + DST
		r/m16 r/m32	Jumps near, absolute indirect, address given in the register or memory location.	(E)IP ← DST
		ptr16:16 ptr16:32	Jumps far, absolute, address given in operand.	(E)IP ← DST.Offset; CS ← DST.Segment
		m16:16 m16:32	Jumps far, absolute indirect, address given in memory location.	
Jump if condition	JA JAE JB JBE JC JE JG JGE JL JLE JNA JNAE JNB JNBE JNC JNE JNG JNGE JNL JNLE JNO	rel8	Jumps near, relative, if above (CF=0 and ZF=0) Jumps near, relative, if above or equal (CF=0) Jumps near, relative, if below (CF=1) Jumps near, relative, if below or equal (CF=1 or ZF=1) Jumps near, relative, if carry (CF=1) Jumps near, relative, if equal (ZF=1) Jumps near, relative, if greater (ZF=0 and SF=OF) Jumps near, relative, if greater or equal (SF=OF) Jumps near, relative, if less (SF<>OF) Jumps near, relative, if less or equal (ZF=1 or SF<>OF) Jumps near, relative, if not above (CF=1 or ZF=1) Jumps near, relative, if not above or equal (CF=1) Jumps near, relative, if not below (CF=0) Jumps near, relative, if not below or equal (CF=0 and ZF=0) Jumps near, relative, if not carry (CF=0) Jumps near, relative, if not equal (ZF=0) Jumps near, relative, if not greater (ZF=1 or SF<>OF) Jumps near, relative, if not greater or equal (SF<>OF) Jumps near, relative, if not less (SF=OF) Jumps near, relative, if not less or equal (ZF=0 and SF=OF) Jumps near, relative, if not overflow (OF=0)	IF (condition) THEN (E)IP ← (E)IP + DST;

	JNP JNS JNZ JO JP JPE JPO JS JZ		Jumps near, relative, if not parity (PF=0) Jumps near, relative, if not sign (SF=0) Jumps near, relative, if not zero (ZF=0) Jumps near, relative, if overflow (OF=1) Jumps near, relative, if parity (PF=1) Jumps near, relative, if parity even (PF=1) Jumps near, relative, if parity off (PF=0) Jumps near, relative, if sign (SF=1) Jumps near, relative, if zero (ZF=1)	
Jump on (E)CX Zero	JCXZ	rel8	Jumps near, relative, if CX is zero	IF (CX=0) THEN (E)IP ← (E)IP + DST;
	JECXZ		Jumps near, relative, if ECX is zero	IF (ECX=0) THEN (E)IP ← (E)IP + DST;
Loop with counter	LOOP	rel8	Decrements counter, jumps near, relative, if counter<>0.	DEC (E)CX; IF ((E)CX<>0) THEN (E)IP ← (E)IP + DST;
Loop with counter while Zero/Equal	LOOPZ LOOPE	rel8	Decrements counter, jumps near, relative, if counter<>0 and ZF=1.	DEC (E)CX; IF ((E)CX<>0 AND ZF=1) THEN (E)IP ← (E)IP + DST;
Loop with counter while Not Zero/ Not Equal	LOOPNZ LOOPNE	rel8	Decrements counter, jumps near, relative, if counter<>0 and ZF=0.	DEC (E)CX; IF ((E)CX<>0 AND ZF=0) THEN (E)IP ← (E)IP + DST;
Call procedure	CALL	rel16 rel32	Calls near, relative, displacement relative to the next instruction	PUSH (E)IP; (E)IP ← (E)IP + DST
		r/m16 r/m32	Calls near, absolute indirect, address given in the register or memory location.	PUSH (E)IP; (E)IP ← DST
		ptr16:16 ptr16:32	Calls far, absolute, address given in operand.	PUSH CS; PUSH (E)IP; (E)IP ← DST.Offset; CS ← DST.Segment
		m16:16 m16:32	Calls far, absolute indirect, address given in memory location.	
Return from procedure	RET		Returns from near or far procedure (depending on procedure kind).	POP (E)IP
		imm16	Returns from near or far procedure (depending on procedure kind) and pop imm16 bytes from the stack.	POP (E)IP; (E)SP ← (E)SP+SRC

Interrupt call	INT	imm8	Calls to interrupt or exception handler using interrupt vector specified by interrupt number.	IF REALMODE THEN
				PUSH FLAGS
				EFLAGS.IF, .TF, .AC ← 000B;
				PUSH CS;
				PUSH IP;
				CS ← IDT[DST].Segment;
				(E)IP ← IDT[DST].Offset;
				ELSE
				...

Interrupt 3 call	INT	3	Calls to debugger trap.	
Interrupt on Overflow	INTO		Calls to interrupt or exception handler 4 if overflow flag is set to 1.	

Interrupt Return	IRET		Returns from the interrupt or exception handler (16 bits)	IF REALMODE THEN
				POP (E)IP;
				POP CS;
				POP TMP;
				EFLAGS ← (TMP AND 257FD5H)
				OR (EFLAGS AND 1A0000H)
				ELSE
				...

	IRETD		Returns from the interrupt or exception handler (32 bits)	
Enter procedure	ENTER	imm16,0 imm16,1	Creates a stack frame for a procedure. The first operand specifies the size of the stack frame (in bytes), the second operand gives the lexical nesting	NestingLevel ← NestingLevel MOD 32; PUSH (E)BP;

		imm16, imm8	level (0 to 31) of the procedure. It determines the number of the stack frame pointers, that are copied into the "display area" of the new stack frame from the preceding frame.	FrameTMP ← (E)SP
				IF NestingLevel>0 THEN
				FOR I ← 1 TO NestingLevel – 1 DO
				(E)BP ← (E)BP – OPERANDSIZE/8;
				PUSH [EBP]
				NEXT
				PUSH FrameTMP
				END
				(E)BP ← FrameTMP;
				(E)SP ← (E)SP–Size;

Leave procedure	LEAVE		Releases the stack frame set up by an earlier ENTER instruction.	(E)SP ← (E)BP; POP (E)BP;
check Bounds	BOUND	r16, m16[2] r32, m32[2]	Checks if array index in the first operand is within bounds specified by the second (memory) operand. If not, then a Bound Range exception is raised.	IF (REG < MEM[0] OR REG >MEM[1]) THEN RAISE #BR END

String Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations

Move String item	MOVS	m8, m8
		m16, m16
		m32, m32
			Moves byte, word or dword from address DS:(E)SI to the byte, word or double word at address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag). Both operands specify only the type of the compared data, not the location. The locations of the operands are always specified by the DS:(E)SI and ES:(E)DI registers.	ES:[(E)DI] ← DS:[(E)SI];
				IF (DF=0) THEN
				(E)SI ← (E)SI + SIZEOF(SRC);
				(E)DI ← (E)DI + SIZEOF(DST);
				ELSE
				(E)SI ← (E)SI – SIZEOF(SRC);
				(E)DI ← (E)DI – SIZEOF(DST);
				END

Move String Byte	MOVSB		Moves byte from address DS:(E)SI to the byte at address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	
Move String Word	MOVSW		Moves word from address DS:(E)SI to the word at address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	
Move String Dword	MOVSD		Moves dword from address DS:(E)SI to the dword at address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	

Repeat Move String item	REP MOVS	m8, m8
		m16, m16
		m32, m32
			Moves (E)CX bytes, words or dwords from address DS:(E)SI to the byte, word or double word at address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag). Both operands specify only the type of the compared data, not the location. The locations of the operands are always specified by the DS:(E)SI and ES:(E)DI registers.	WHILE (E)CX<>0 DO
				MOVS DST, SRC;
				(E)CX ← (E)CX –1
				END

Repeat Move String Byte	REP MOVSB		Moves (E)CX bytes from address DS:(E)SI to the address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	WHILE (E)CX<>0 DO MOVS(B|W|D) (E)CX ← (E)CX –1 END
Repeat Move String Word	REP MOVSW		Moves (E)CX words from address DS:(E)SI to the address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	
Repeat Move String Dword	REP MOVSD		Moves (E)CX dwords from address DS:(E)SI to the address ES:(E)DI, and increases or decreases (E)SI and (E)DI (depending on DF flag).	

Load String item	LODS	m8
		m16
		m32
			Loads byte from address DS:(E)SI to AL, increases or decreases (E)SI.
			Loads word from address DS:(E)SI to AX, increases or decreases (E)SI.
			Loads dword from address DS:(E)SI to EAX, increases or decreases (E)SI
				ACC ← DS:[(E)SI];
				IF (DF=0) THEN
				(E)SI ← (E)SI + SIZEOF(SRC);
				ELSE
				(E)SI ← (E)SI – SIZEOF(SRC);
				END

Load String Byte	LODSB		Loads byte from address DS:(E)SI to AL, increases or decreases (E)SI.	
Load String Word	LODSW		Loads word from address DS:(E)SI to AX, increases or decreases (E)SI.	
Load String Dword	LODSD		Loads dword from address DS:(E)SI to EAX, increases or decreases (E)SI.	

Repeat Load String item	REP LODS	m8 m16 m32	Loads byte (E)CX times from address DS:(E)SI to AL. Loads word (E)CX times from address DS:(E)SI to AX. Loads dword (E)CX times from address DS:(E)SI to EAX.	WHILE (E)CX<>0 DO LODS DST; (E)CX ← (E)CX –1 END
Repeat Load String Byte	REP LODSB		Loads byte (E)CX times from address DS:(E)SI to AL.	WHILE (E)CX<>0 DO LODS(B|W|D) (E)CX ← (E)CX –1 END
Repeat Load String Word	REP LODSW		Loads word (E)CX times from address DS:(E)SI to AX.	
Repeat Load String Dword	REP LODSD		Loads dword (E)CX times from address DS:(E)SI to EAX.	

Load String item	LODS	m8
		m16
		m32
			Stores byte from AL to address ES:(E)DI, increases or decreases (E)DI.
			Stores word from AX to address ES:(E)DI, increases or decreases (E)DI.
			Stores dword from EAX address ES:(E)DI, increases or decreases (E)DI
				ES:[(E)DI] ← ACC;
				IF (DF=0) THEN
				(E)DI ← (E)DI + SIZEOF(DST);
				ELSE
				(E)DI ← (E)DI – SIZEOF(DST);
				END

Store String Byte	STOSB		Stores byte from AL to address ES:(E)DI, increases or decreases (E)DI.	
Store String Word	STOSW		Stores word from AX to address ES:(E)DI, increases or decreases (E)DI.	
Store String Dword	STOSD		Stores dword from EAX to address ES:(E)DI, increases or decreases (E)DI.	
Repeat Store String item	REP STOS	m8 m16 m32	Stores byte (E)CX times from AL to address ES:(E)DI. Stores word (E)CX times from AX to address ES:(E)DI. Stores dword (E)CX times from EAX to address ES:(E)DI.	WHILE (E)CX<>0 DO STOS DST; (E)CX ← (E)CX –1 END
Repeat Store String Byte	REP STOSB		Stores byte (E)CX times from AL to address ES:(E)DI.	WHILE (E)CX<>0 DO STOS(B|W|D) (E)CX ← (E)CX –1 END
Repeat Store String Word	REP STOSW		Stores word (E)CX times from AX to address ES:(E)DI.	
Repeat Store String Dword	REP STOSD		Stores dword (E)CX times from EAX address ES:(E)DI.	

Compare String item	CMPS	m8, m8
		m16, m16
		m32, m32
			Compares byte, word or dword at address DS:(E)SI with byte, word or dword at address ES:(E)DI and sets the status flags accordingly. Both operands specify only the type of the compared data, not the location. The locations of the operands are always specified by the DS:(E)SI and ES:(E)DI registers.	TMP ← ES:[(E)DI] – DS:[(E)SI];
				SET EFLAGS;
				IF (DF=0) THEN
				(E)SI ← (E)SI + SIZEOF(SRC);
				(E)DI ← (E)DI + SIZEOF(DST);
				ELSE
				(E)SI ← (E)SI – SIZEOF(SRC);
				(E)DI ← (E)DI – SIZEOF(DST);
				END

Compare String Byte	CMPSB		Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly.	
Compare String Word	CMPSW		Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly.	
Compare String Dword	CMPSD		Compares dword at address DS:(E)SI with dword at address ES:(E)DI and sets the status flags accordingly.	

Repeat Compare String item until Equal / Zero	REPE CMPS
	REPZ CMPS
		m8, m8
		m16, m16
		m32, m32
			Repeats (E)CX times comparing byte, word or dword at address DS:(E)SI with byte, word or double word at address ES:(E)DI until ZF flag is set to 0.	WHILE (E)CX<>0 DO
				(E)CX ← (E)CX –1;
				CMPS DST,SRC;
				UNTIL ZF=0

Repeat Compare String Byte until Equal / Zero	REPE CMPSB REPZ CMPSB		Repeats (E)CX times comparing byte at address DS:(E)SI with byte at address ES:(E)DI until ZF flag is set to 0.	WHILE (E)CX<>0 DO (E)CX ← (E)CX –1; CMPS(B|W|D)
Repeat Compare String Word until Equal / Zero	REPE CMPSW REPZ CMPSW		Repeats (E)CX times comparing word at address DS:(E)SI with word at address ES:(E)DI until ZF flag is set to 0.	

Repeat Compare String Dword until Equal / Zero	REPE CMPSD REPZ CMPSD		Repeats (E)CX times comparing dword at address DS:(E)SI with dword at address ES:(E)DI until ZF flag is set to 0.	UNTIL ZF=0

Repeat Compare String item until Not Equal / Not Zero	REPNE CMPS
	REPNZ CMPS
		m8, m8
		m16, m16
		m32, m32
			Repeats (E)CX times comparing byte, word or dword at address DS:(E)SI with byte, word or double word at address ES:(E)DI until ZF flag is set to 1.	WHILE (E)CX<>0 DO
				(E)CX ← (E)CX −1;
				CMPS DST,SRC;
				UNTIL ZF=1

Repeat Compare String Byte until Not Equal /Not Zero	REPNE CMPSB REPNZ CMPSB		Repeats (E)CX times comparing byte at address DS:(E)SI with byte at address ES:(E)DI until ZF flag is set to 1.	WHILE (E)CX<>0 DO (E)CX ← (E)CX −1; CMPS(B|W|D) UNTIL ZF=1
Repeat Compare String Word until Not Equal / Not Zero	REPNE CMPSW REPNZ CMPSW		Repeats (E)CX times comparing word at address DS:(E)SI with word at address ES:(E)DI until ZF flag is set to 1.	
Repeat Compare String Dword until Not Equal	REPNE CMPSD REPNZ CMPSD		Repeats (E)CX times comparing dword at address DS:(E)SI with dword at address ES:(E)DI until ZF flag is set to 1.	

Scan String item	SCAS	m8
		m16
		m32
			Compares AL with byte at ES:(E)DI and sets status flag.
			Compares AX with byte at ES:(E)DI and sets status flag.
			Compares EAX with byte at ES:(E)DI and sets status flag.
				TMP ← ACC − DS:[(E)SI];
				SET EFLAGS;
				IF (DF=0) THEN
				(E)SI ← (E)SI + SIZEOF(SRC);
				(E)DI ← (E)DI + SIZEOF(DST);
				ELSE
				(E)SI ← (E)SI − SIZEOF(SRC);
				(E)DI ← (E)DI − SIZEOF(DST);
				END

Scan String Byte	SCASB		Compares AL with byte at ES:(E)DI and sets status flag.	
Scan String Word	SCASW		Compares AX with byte at ES:(E)DI and sets status flag.	
Scan String Dword	SCASD		Compares EAX with byte at ES:(E)DI and sets status flag.	
Repeat Scan String item until Equal / Zero	REPE SCAS REPZ SCAS	m8 m16 m32	Repeats (E)CX times comparing accumulator with byte, word or dword at address ES:(E)DI until ZF flag is set to 0.	WHILE (E)CX<>0 DO (E)CX ← (E)CX −1; SCAS DST; UNTIL ZF=1
Repeat Scan String Byte until Equal / Zero	REPE SCASB REPZ SCASB		Repeats (E)CX times comparing AL with byte at address ES:(E)DI until ZF flag is set to 0.	WHILE (E)CX<>0 DO (E)CX ← (E)CX −1; SCAS(B|W|D); UNTIL ZF=1
Repeat Scan String Word until Equal / Zero	REPE SCASW REPZ SCASW		Repeats (E)CX times comparing AX with word at address ES:(E)DI until ZF flag is set to 0.	
Repeat Scan String Dword until Equal / Zero	REPE SCASD REPZ SCASD		Repeats (E)CX times comparing EAX with dword at address ES:(E)DI until ZF flag is set to 0.	
Repeat Scan String item until Not Equal / Not Zero	REPNE SCAS REPNZ SCAS	m8 m16 m32	Repeats (E)CX times comparing accumulator with byte, word or dword at address ES:(E)DI until ZF flag is set to 1.	WHILE (E)CX<>0 DO (E)CX ← (E)CX −1; SCAS DST; UNTIL ZF=1
Repeat Scan String Byte until Not Equal / Not Zero	REPNE SCASB REPNZ SCASB		Repeats (E)CX times comparing AL with byte at address ES:(E)DI until ZF flag is set to 1.	WHILE (E)CX<>0 DO (E)CX ← (E)CX −1; SCAS(B|W|D); UNTIL ZF=1
Repeat Scan String Word until Not Equal / Not Zero	REPNE SCASW REPNZ SCASW		Repeats (E)CX times comparing AX with word at address ES:(E)DI until ZF flag is set to 1.	
Repeat Scan String Dword until Not Equal / Not Zero	REPNE SCASD REPNZ SCASD		Repeats (E)CX times comparing EAX with dword at address ES:(E)DI until ZF flag is set to 1.	

Input String item	INS	m8, DX
		m16, DX
		m32, DX
			Inputs byte, word or dword from I/O specified in DX into memory location specified with ES:(E)DI. Increments or decrements (E)DI.	ES:[(E)DI] ← Port(DX)
				IF (DF=0) THEN
				(E)DI ← (E)DI + SIZEOF(DST);
				ELSE
				(E)DI ← (E)DI – SIZEOF(DST);
				END

Input String Byte	INSB		Inputs byte from I/O specified in DX into memory location specified with ES:(E)DI. Increments or decrements (E)DI.	
Input String Word	INSB		Inputs word from I/O specified in DX into memory location specified with ES:(E)DI. Increments or decrements (E)DI.	
Input String Dword	INSB		Inputs dword from I/O specified in DX into memory location specified with ES:(E)DI. Increments or decrements (E)DI.	
Repeat Input String item	REP INS	m8, DX m16, DX m32, DX	Inputs (E)CX bytes, words or dwords from I/O specified in DX into memory at address specified with ES:(E)DI. Increments or decrements (E)DI.	WHILE (E)CX<>0 DO INS SRC,DX; (E)CX ← (E)CX – 1 END
Repeat Input String Byte	REP INSB		Inputs (E)CX bytes from I/O specified in DX into memory at address specified with ES:(E)DI. Increments or decrements (E)DI.	WHILE (E)CX<>0 DO INS(B|W|D); (E)CX ← (E)CX – 1 END
Repeat Input String Word	REP INSW		Inputs (E)CX words from I/O specified in DX into memory at address specified with ES:(E)DI. Increments or decrements (E)DI.	
Repeat Input String Dword	REP INSD		Inputs (E)CX dwords from I/O specified in DX into memory at address specified with ES:(E)DI. Increments or decrements (E)DI.	

Output String item	OUTS	DX, m8
		DX, m16
		DX, m32
			Outputs byte, word or dword from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	Port(DX) ← DS:[(E)SI]
				IF (DF=0) THEN
				(E)SI ← (E)SI + SIZEOF(SRC);
				ELSE
				(E)SI ← (E)SI – SIZEOF(SRC);
				END

Output String Byte	OUTSB		Outputs byte from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	
Output String Word	OUTSB		Outputs word from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	
Output String Dword	OUTSB		Outputs dword from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	
Repeat Output String item	REP OUTS	DX, m8 DX, m16 DX, m32	Outputs (E)CX bytes, words or dwords from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	WHILE (E)CX<>0 DO OUTS SRC,DX; (E)CX ← (E)CX – 1 END
Repeat Output String Byte	REP OUTSB		Outputs (E)CX bytes from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	WHILE (E)CX<>0 DO OUTS(B|W|D); (E)CX ← (E)CX – 1 END
Repeat Output String Word	REP OUTSW		Outputs (E)CX words from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	
Repeat Output String Dword	REP OUTSD		Outputs (E)CX dwords from memory location specified with DS:(E)SI to I/O specified in DX into. Increments or decrements (E)SI.	

Flag Control Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Clear Carry Flag	CLC		Clears CF flag in the EFLAGS register.	CF ← 0
Complement Carry Flag	CMC		Complements CF flag in the EFLAGS register.	CF ← NOT CF
Set Carry Flag	STC		Sets CF flag in the EFLAGS register	CF ← 1
Clear Direction flag	CLD		Clears DF Flag in the EFLAGS register. When the DF flag is set to 0,	DF ← 0

			string instructions increment the index registers.	
Set Direction flag	STD		Sets DF Flag in the EFLAGS register to 1. When the DF flag is set to 1, string instructions decrement the index registers.	DF ← 1
Clear Interrupt Flag	CLI		Clears interrupt flag; interrupts disabled when IF flag is cleared.	IF ← 0
Set Interrupt Flag	STI		Sets interrupt flag; interrupts enabled when IF flag is set to 1.	IF ← 1
Push Flags	PUSHF		Pushes FLAGS (16 bits).	PUSH FLAGS
	PUSHFD		Pushes EFLAGS.	PUSH (EFLAGS AND 00FCFFFFH) *// VM and RF flags are cleared on the stack*
Pop Flags	POPF		Pops FLAGS (16 bits).	POP FLAGS
	POPFD		Pops EFLAGS.	POP EFLAGS *// All non-reserved flags except VIP, VIF and VM can be modified.* *// VIP and VIF are celared; VM is unnafected*
Load Flags into AH	LAHF		Moves the low byte of the EFLAGS register into AH register. Reserved bits (1, 3 , 5) are set accordingly to 1, 0, 0	AH ← EFLAGS[SF:ZF:0:AF:PF:1:CF]
Store AH into Flags	SAHF		Moves AH register into the low byte of the EFLAGS. Reserved bits (1, 3, 5) are ignored.	EFLAGS[SF:ZF:0:AF:PF:1:CF] ← AH

Segment Register Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Load far pointer using DS	LDS	r16, m16:16 r32, m32:32	Loads DS:r16 with far pointer from memory. Loads DS:r32 with far pointer from memory	DS ← SRC.Segment; DST ← SRC.Offset;
Load far pointer using ES	LES	r16, m16:16 r32, m32:32	Loads ES:r16 with far pointer from memory. Loads ES:r32 with far pointer from memory	ES ← SRC.Segment; DST ← SRC.Offset;
Load far pointer using FS	LFS	r16, m16:16 r32, m32:32	Loads FS:r16 with far pointer from memory. Loads FS:r32 with far pointer from memory	FS ← SRC.Segment; DST ← SRC.Offset;
Load far pointer using GS	LGS	r16, m16:16 r32, m32:32	Loads GS:r16 with far pointer from memory. Loads GS:r32 with far pointer from memory	GS ← SRC.Segment; DST ← SRC.Offset;
Load far pointer using SS	LSS	r16, m16:16 r32, m32:32	Loads SS:r16 with far pointer from memory. Loads SS:r32 with far pointer from memory	SS ← SRC.Segment; DST ← SRC.Offset;

Miscellaneous Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Load effective address	LEA	r16, m r32, m	Stores effective address for m in the destination register.	DST ← EffectiveAddress(m)
No Operation	NOP		Do nothing.	
Undefined instruction	UD2		Raises invalid opcode exception.	
Table Look-up Translation	XLAT	m8	Set AL to memory byte DS:[(E)BX + unsigned AL]	AL ← DS:[(E)BX + ZeroExtend(AL)]
	XLATB			
CPU Identification	CPUID		Returns processor identification and feature information to the EAX, EBX, ECX and EDX registers, according to the input value entered initially in the EAX register	

System Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Load Global Description Table Register	LGDT	m16&32	Loads the values from the memory into the global descriptor table register (GDTR).	GDTR ← SRC
Store Global Descriptor Table Register	SGDT	m16&32	Stores the contents of the global description table register (GDTR) to the memory.	DST ← GDTR
Load Interrupt Descriptor Table Register	LIDT	m16&32	Loads the values from the memory into the interrupt descriptor table register (IDTR).	IDTR ← SRC
Store Interrupt Descriptor Table Register	SIDT	m16&32	Stores the contents of the interrupt description table register (IDTR) to the memory.	DST ← IDTR
Load Local Descriptor Table Register	LLDT	r/m16 r32	Loads the segment selector from the local descriptor table register (LDTR) in the register or memory.	LDTR ← SRC
Store Local Descriptor Table Register	SLDT	r/m16 r32	Stores the segment selector from the register or memory to the local descriptor table register (LDTR).	DST ← LDTR
Load Machine Status Word	LMSW	r/m16 r32	Loads the machine status word from the register or memory.	MSW ← SRC
Store Machine Status Word	SMSW	r/m16 r32	Stores the machine status word to the register or memory.	DST ← MSW
Load Task Register	LTR	r/m16	Loads the source operand into the segment selector field of the task register.	TR ← SRC
Store Task Register	STR	r/m16	Stores the segment selector field of the task register into the destination operand	DST ← TR
Move to/from control registers	MOV	CR0, r32 CR2, r32 CR3, r32 CR4, r32 r32, CR0 r32, CR2 r32, CR3 r32, CR4	Moves *r32* to CR0. Moves *r32* to CR2. Moves *r32* to CR3. Moves *r32* to CR4. Moves CR0 to *r32*. Moves CR2 to *r32*. Moves CR3 to *r32*. Moves CR4 to *r32*.	DST ← SRC
Clear Task-Switch	CLTS		Clears the TS flag in the CR0 register. This flag is set every time a task switch occurs.	CR0.TS ←0;

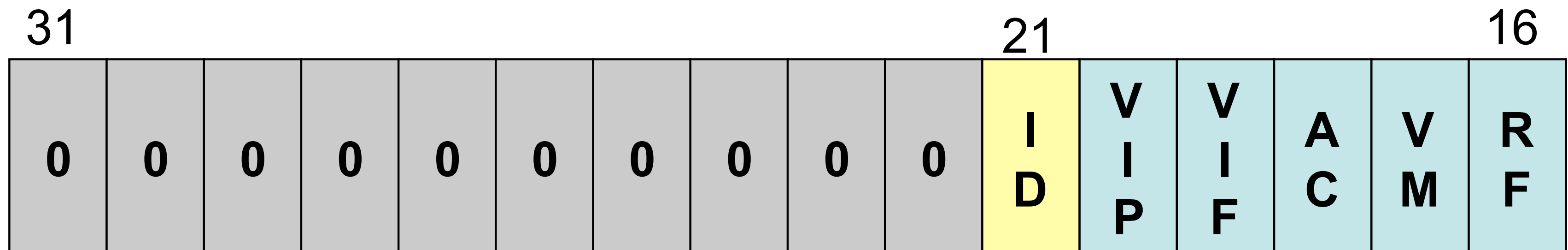
Adjust RPL Field of Segment Selector	ARPL	r/m16, r16	Compares the RPL fields of the two segment selectors. The first (destination) operand contains one segment selector and the second (source) operand contains the other. The RPL field of the first operand is set not less then RPL of the second operand.	IF DST.RPL < SRC.RPL THEN
				ZF ← 1;
				DST.RPL ← SRC.RPL;
				ELSE
				ZF ← 0;
				END;

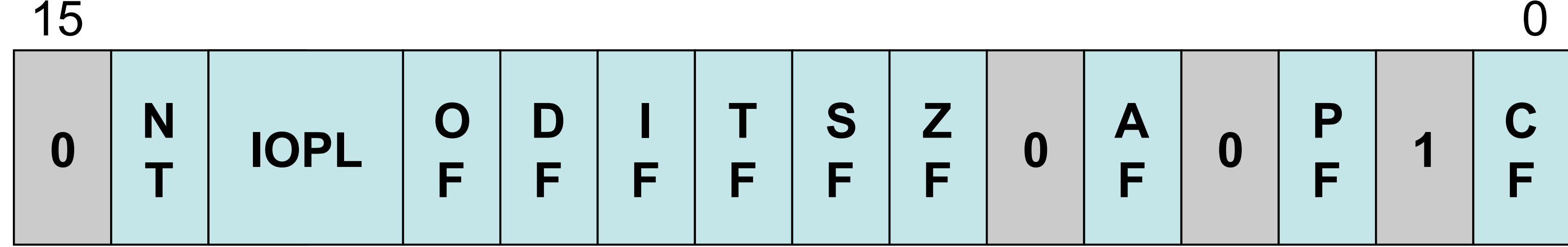
Load Access Rights Byte	LAR	r16, r/m16 r32, r/m32	Loads the access right from the segment descriptor specified by the source operand into the destination operand and sets the ZF flag.	
Load Segment Limit	LSL	r16, r/m16 r32, r/m32	Loads the unscrambled segment limit from the segment descriptor specified by the source operand into the destination operand and sets the ZF flag.	
Verify a Segment for Reading	VERR	r/m16	Sets ZF=1 if segment specified with r/m16 can be read.	
Verify a Segment for Writing	VERW	r/m16	Sets ZF=1 if segment specified with r/m16 can be written.	
Move to/from Debug Registers	MOV	r32, DR0-DR7 DR0-DR7, r32	Moves debug register to *r32*. Moves *r32* to debug register .	DST ← SRC
Invalidate Cache	INVD		Flushes internal cashes; initiates flushing of external caches.	
Write Back and Invalidate Data Cache	WBINVD		Writes back and flushes internal cashes; initiates writing-back and flushing of external caches.	
Invalidate TLB Entry	INVLPG		Invalidates (flushed) the translation look-aside buffer (TLB) entry specified with the source operand.	
Assert LOCK# signal Prefix	LOCK (prefix)		Causes the processor LOCK# signal to be asserted during execution of the accompanying instructions (turns the instruction into an atomic instruction). In a multiprocessor environment, the lock# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.	
Halt	HLT		Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal or the RESET# signal will resume execution.	
Resume from System Management Mode	RSM		Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor receive and SSM interrupt.	
Read from Model-Specific Register	RDMSR		Reads the contents of the 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX.	
Write from Model-Specific Register	WRMSR		Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register.	
Read Performance Monitoring Counters	RDPMC		Reads the contents of the 40-bit performance-monitoring counter (PMC) specified in the ECX register into registers EDX:EAX.	
Read Time-Stamp Counter	RDTSC		Reads the current value of the processors time-stamp counter into the EDX:EAX register.	
Fast System Call	SYSENTER		Fast call to privilege level 0 system procedures or routine.	
Fast Return from Fast System Call	SYSEXIT		Executes a fast return to privilege level 3 user code.	

CPUID

- Recent Intel processors provide a ‘cpuid’ instruction (opcode 0x0F, 0xA2) to assist software in detecting a CPU’s capabilities
- If it’s implemented, this instruction can be executed in any of the processor modes, and at any of its four privilege levels
- But this ‘cpuid’ instruction might not be implemented (e.g., 8086, 80286, 80386)

Intel x86 EFLAGS register





Software can ‘toggle’ the ID-bit (bit #21) in the 32-bit EFLAGS register if the processor is capable of executing the ‘cpuid’ instruction

But what if there’s no EFLAGS?

- The early Intel processors (8086, 80286) did not implement any 32-bit registers
- The FLAGS register was only 16-bits wide
- So there was no ID-bit that software could try to ‘toggle’ (to see if ‘cpuid’ existed)
- How can software be sure that the 32-bit EFLAGS register exists within the CPU?

Detecting 32-bit processors

- There’s a subtle difference in the way the logical shift/rotate instructions work when register CL contains the ‘shift-factor’
- On the 32-bit processors (e.g., 80386+) the value in CL is truncated to 5-bits, but not so on the 16-bit CPUs (8086, 80286)
- Software can exploit this distinction, in order to tell if EFLAGS is implemented

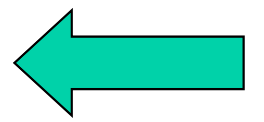
CPUID

Opcode	Instruction	Description
0F A2	CPUID	EAX ← Processor identification information

Input: EAX

Output: EAX, EBX, ECX, and EDX

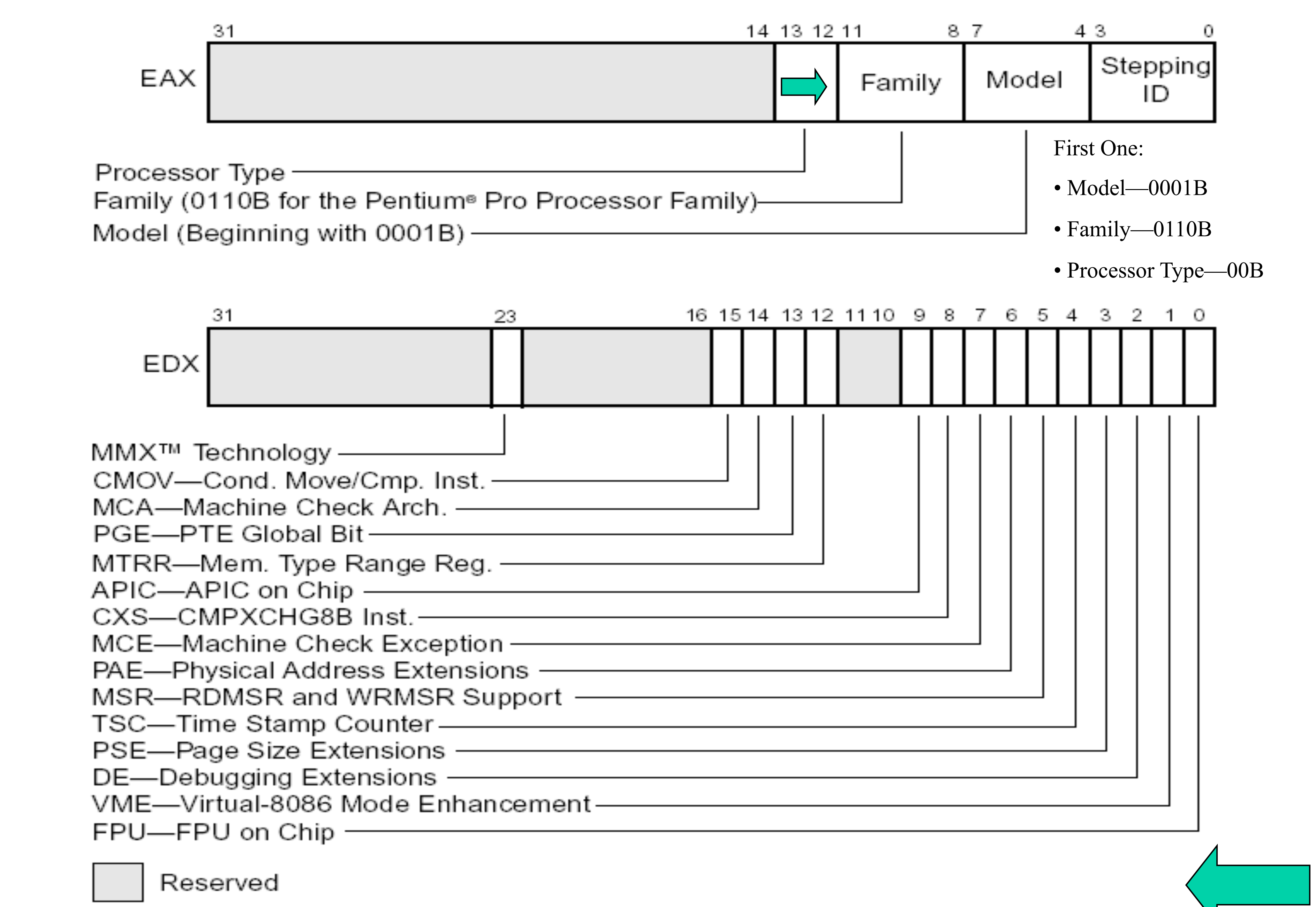
Initial EAX Value		Information Provided about the Processor
0	EAX	Maximum CPUID Input Value (2 for the Pentium® Pro processor and 1 for the Pentium processor and the later versions of Intel486™ processor that support the CPUID instruction).
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
1	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Reserved
	ECX	Reserved
	EDX	Feature Information
2	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information



Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 4 entries
03H	Data TLB: 4K-Byte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4M-Byte Pages, 4-way set associative, 8 entries
06H	Instruction cache: 8K Bytes, 4-way set associative, 32 byte line size
08H	Instruction cache: 16K Bytes, 4-way set associative, 32 byte line size
0AH	Data cache: 8K Bytes, 2-way set associative, 32 byte line size
0CH	Data cache: 16K Bytes, 2-way set associative, 32 byte line size
41H	Unified cache: 128K Bytes, 4-way set associative, 32 byte line size
42H	Unified cache: 256K Bytes, 4-way set associative, 32 byte line size
43H	Unified cache: 512K Bytes, 4-way set associative, 32 byte line size
44H	Unified cache: 1M Byte, 4-way set associative, 32 byte line size

CPUID : EAX =1



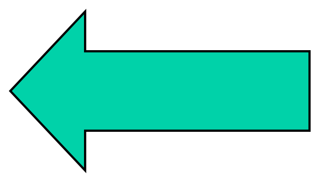
Feature Flags Returned in EDX Register

Bit	Feature	Description
0	FPU—Floating-Point Unit on Chip	Processor contains an FPU and executes the Intel 387 instruction set.
1	VME—Virtual-8086 Mode Enhancements	Processor supports the following virtual-8086 mode enhancements: • CR4.VME bit enables virtual-8086 mode extensions. • CR4.PVI bit enables protected-mode virtual interrupts. • Expansion of the TSS with the software indirection bitmap. • EFLAGS.VIF bit (virtual interrupt flag). • EFLAGS.VIP bit (virtual interrupt pending flag).
2	DE—Debugging Extensions	Processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE—Page Size Extensions	Processor supports 4-Mbyte pages, including the CR4.PSE bit for enabling page size extensions, the modified bit in page directory entries (PDEs), page directory entries, and page table entries (PTEs).
4	TSC—Time Stamp Counter	Processor supports the RDTSC (read time stamp counter) instruction, including the CR4.TSD bit that, along with the CPL, controls whether the time stamp counter can be read.
5	MSR—Model Specific Registers	Processor supports the RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions.

Bit	Feature	Description
6	PAE—Physical Address Extension	Processor supports physical addresses greater than 32 bits, the extended page-table-entry format, an extra level in the page translation tables, and 2-MByte pages. The CR4.PAE bit enables this feature. The number of address bits is implementation specific. The Pentium® Pro processor supports 36 bits of addressing when the PAE bit is set.
7	MCE—Machine Check Exception	Processor supports the CR4.MCE bit, enabling machine check exceptions. However, this feature does not define the model-specific implementations of machine-check error logging, reporting, or processor shutdowns. Machine-check exception handlers might have to check the processor version to do model-specific processing of the exception or check for the presence of the standard machine-check feature.
8	CX8—CMPXCHG8B Instruction	Processor supports the CMPXCHG8B (compare and exchange 8 bytes) instruction.
9	APIC	Processor contains an on-chip Advanced Programmable Interrupt Controller (APIC) and it has been enabled and is available for use.
10,11	Reserved	
12	MTRR—Memory Type Range Registers	Processor supports machine-specific memory-type range registers (MTRRs). The MTRRs contains bit fields that indicate the processor's MTRR capabilities, including which memory types the processor supports, the number of variable MTRRs the processor supports, and whether the processor supports fixed MTRRs.
13	PGE—PTE Global Flag	Processor supports the CR4.PGE flag enabling the global bit in both PTDEs and PTEs. These bits are used to indicate translation lookaside buffer (TLB) entries that are common to different tasks and need not be flushed when control register CR3 is written.
14	MCA—Machine Check Architecture	Processor supports the MCG_CAP (machine check global capability) MSR. The MCG_CAP register indicates how many banks of error reporting MSRs the processor supports.
15	CMOV—Conditional Move and Compare Instructions	Processor supports the CMOV*cc* instruction and, if the FPU feature flag (bit 0) is also set, supports the FCMOV*cc* and FCOMI instructions.
16-22	Reserved	
23	MMX™ Technology	Processor supports the MMX instruction set. These instructions operate in parallel on multiple data elements (8 bytes, 4 words, or 2 doublewords) packed into quadword registers or memory locations.
24-31	Reserved	

Processor Type

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor *	10B
Intel reserved.	11B



arm

Key to Tables

`Rm {, <opsh>}`	See Table **Register, optionally shifted by constant**		
`<Operand2>`	See Table **Flexible Operand 2**. Shift and rotate are only available as part of Operand2.	`<reglist>`	A comma-separated list of registers, enclosed in braces { and }.
`<fields>`	See Table **PSR fields**.	`<reglist-PC>`	As `<reglist>`, must not include the PC.
`<PSR>`	Either CPSR (Current Processor Status Register) or SPSR (Saved Processor Status Register)	`<reglist+PC>`	As `<reglist>`, including the PC.
`C*, V*`	Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later.	`+/-`	+ or –. (+ may be omitted.)
`<Rs|sh>`	Can be Rs or an immediate shift value. The values allowed for each shift type are the same as those shown in Table **Register, optionally shifted by constant**.	§	See Table **ARM architecture versions**.
		`<iflags>`	Interrupt flags. One or more of `a`, `i`, `f` (abort, interrupt, fast interrupt).
`x,y`	B meaning half-register [15:0], or T meaning [31:16].	`<p_mode>`	See Table **Processor Modes**
`<imm8m>`	ARM: a 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.	`SPm`	SP for the processor mode specified by `<p_mode>`
	Thumb: a 32-bit constant, formed by left-shifting an 8-bit value by any number of bits, or a bit pattern of one of the forms 0xXYXYXYXY, 0x00XY00XY or 0xXY00XY00.	`<lsb>`	Least significant bit of bitfield.
		`<width>`	Width of bitfield. `<width>` + `<lsb>` must be <= 32.
`<prefix>`	See Table **Prefixes for Parallel instructions**	`{X}`	RsX is Rs rotated 16 bits if X present. Otherwise, RsX is Rs.
`{IA|IB|DA|DB}`	Increment After, Increment Before, Decrement After, or Decrement Before.	`{!}`	Updates base register after data transfer if ! present (pre-indexed).
	`IB` and `DA` are not available in Thumb state. If omitted, defaults to `IA`.	`{S}`	Updates condition flags if S present.
`<size>`	`B`, `SB`, `H`, or `SH`, meaning Byte, Signed Byte, Halfword, and Signed Halfword respectively.	`{T}`	User mode privilege if T present.
	`SB` and `SH` are not available in `STR` instructions.	`{R}`	Rounds result to nearest if R present, otherwise truncates result.

Operation		§	Assembler	S updates	Action	Notes
Add	Add		`ADD{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2	N
	with carry		`ADC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn + Operand2 + Carry	N
	wide	T2	`ADD Rd, Rn, #<imm12>`		Rd := Rn + imm12, imm12 range 0-4095	T, P
	saturating {doubled}	5E	`Q{D}ADD Rd, Rm, Rn`		Rd := SAT(Rm + Rn) doubled: Rd := SAT(Rm + SAT(Rn * 2))	Q
Address	Form PC-relative address		`ADR Rd, <label>`		Rd := <label>, for <label> range from current instruction see Note L	N, L
Subtract	Subtract		`SUB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2	N
	with carry		`SBC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Rn – Operand2 – NOT(Carry)	N
	wide	T2	`SUB Rd, Rn, #<imm12>`	N Z C V	Rd := Rn – imm12, imm12 range 0-4095	T, P
	reverse subtract		`RSB{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn	N
	reverse subtract with carry		`RSC{S} Rd, Rn, <Operand2>`	N Z C V	Rd := Operand2 – Rn – NOT(Carry)	A
	saturating {doubled}	5E	`Q{D}SUB Rd, Rm, Rn`		Rd := SAT(Rm – Rn) doubled: Rd := SAT(Rm – SAT(Rn * 2))	Q
	Exception return without stack		`SUBS PC, LR, #<imm8>`		PC = LR – imm8, CPSR = SPSR(current mode), imm8 range 0-255.	T
Parallel arithmetic	Halfword-wise addition	6	`<prefix>ADD16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0]	G
	Halfword-wise subtraction	6	`<prefix>SUB16 Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[31:16], Rd[15:0] := Rn[15:0] – Rm[15:0]	G
	Byte-wise addition	6	`<prefix>ADD8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0]	G
	Byte-wise subtraction	6	`<prefix>SUB8 Rd, Rn, Rm`		Rd[31:24] := Rn[31:24] – Rm[31:24], Rd[23:16] := Rn[23:16] – Rm[23:16], Rd[15:8] := Rn[15:8] – Rm[15:8], Rd[7:0] := Rn[7:0] – Rm[7:0]	G
	Halfword-wise exchange, add, subtract	6	`<prefix>ASX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] – Rm[31:16]	G
	Halfword-wise exchange, subtract, add	6	`<prefix>SAX Rd, Rn, Rm`		Rd[31:16] := Rn[31:16] – Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16]	G
	Unsigned sum of absolute differences	6	`USAD8 Rd, Rm, Rs`		Rd := Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
	and accumulate	6	`USADA8 Rd, Rm, Rs, Rn`		Rd := Rn + Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0])	
Saturate	Signed saturate word, right shift	6	`SSAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := SignedSat((Rm ASR sh), sat). `<sat>` range 1-32, `<sh>` range 1-31.	Q, R
	Signed saturate word, left shift	6	`SSAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := SignedSat((Rm LSL sh), sat). `<sat>` range 1-32, `<sh>` range 0-31.	Q
	Signed saturate two halfwords	6	`SSAT16 Rd, #<sat>, Rm`		Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat). `<sat>` range 1-16.	Q
	Unsigned saturate word, right shift	6	`USAT Rd, #<sat>, Rm{, ASR <sh>}`		Rd := UnsignedSat((Rm ASR sh), sat). `<sat>` range 0-31, `<sh>` range 1-31.	Q, R
	Unsigned saturate word, left shift	6	`USAT Rd, #<sat>, Rm{, LSL <sh>}`		Rd := UnsignedSat((Rm LSL sh), sat). `<sat>` range 0-31, `<sh>` range 0-31.	Q
	Unsigned saturate two halfwords	6	`USAT16 Rd, #<sat>, Rm`		Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat). `<sat>` range 0-15.	Q

Operation		§	Assembler	S updates	Action	Notes
Multiply	Multiply		MUL{S} Rd, Rm, Rs	N Z C*	Rd := (Rm * Rs)[31:0] (If Rm is Rd, S can be used in Thumb-2)	N, S
	and accumulate		MLA{S} Rd, Rm, Rs, Rn	N Z C*	Rd := (Rn + (Rm * Rs))[31:0]	S
	and subtract	T2	MLS Rd, Rm, Rs, Rn		Rd := (Rn – (Rm * Rs))[31:0]	
	unsigned long		UMULL{S} RdLo, RdHi, Rm, Rs	N Z C* V*	RdHi,RdLo := unsigned(Rm * Rs)	S
	unsigned accumulate long		UMLAL{S} RdLo, RdHi, Rm, Rs	N Z C* V*	RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)	S
	unsigned double accumulate long	6	UMAAL RdLo, RdHi, Rm, Rs		RdHi,RdLo := unsigned(RdHi + RdLo + Rm * Rs)	
	Signed multiply long		SMULL{S} RdLo, RdHi, Rm, Rs	N Z C* V*	RdHi,RdLo := signed(Rm * Rs)	S
	and accumulate long		SMLAL{S} RdLo, RdHi, Rm, Rs	N Z C* V*	RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)	S
	16 * 16 bit	5E	SMULxy Rd, Rm, Rs		Rd := Rm[x] * Rs[y]	
	32 * 16 bit	5E	SMULWy Rd, Rm, Rs		Rd := (Rm * Rs[y])[47:16]	
	16 * 16 bit and accumulate	5E	SMLAxy Rd, Rm, Rs, Rn		Rd := Rn + Rm[x] * Rs[y]	Q
	32 * 16 bit and accumulate	5E	SMLAWy Rd, Rm, Rs, Rn		Rd := Rn + (Rm * Rs[y])[47:16]	Q
	16 * 16 bit and accumulate long	5E	SMLALxy RdLo, RdHi, Rm, Rs		RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]	
	Dual signed multiply, add	6	SMUAD{X} Rd, Rm, Rs		Rd := Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	Q
	and accumulate	6	SMLAD{X} Rd, Rm, Rs, Rn		Rd := Rn + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	Q
	and accumulate long	6	SMLALD{X} RdLo, RdHi, Rm, Rs		RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16]	
	Dual signed multiply, subtract	6	SMUSD{X} Rd, Rm, Rs		Rd := Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	Q
	and accumulate	6	SMLSD{X} Rd, Rm, Rs, Rn		Rd := Rn + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	Q
	and accumulate long	6	SMLSLD{X} RdLo, RdHi, Rm, Rs		RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16]	
	Signed top word multiply	6	SMMUL{R} Rd, Rm, Rs		Rd := (Rm * Rs)[63:32]	
	and accumulate	6	SMMLA{R} Rd, Rm, Rs, Rn		Rd := Rn + (Rm * Rs)[63:32]	
	and subtract	6	SMMLS{R} Rd, Rm, Rs, Rn		Rd := Rn – (Rm * Rs)[63:32]	
	with internal 40-bit accumulate	XS	MIA Ac, Rm, Rs		Ac := Ac + Rm * Rs	
	packed halfword	XS	MIAPH Ac, Rm, Rs		Ac := Ac + Rm[15:0] * Rs[15:0] + Rm[31:16] * Rs[31:16]	
	halfword	XS	MIAxy Ac, Rm, Rs		Ac := Ac + Rm[x] * Rs[y]	
Divide	Signed or Unsigned	RM	<op> Rd, Rn, Rm		Rd := Rn / Rm <op> is SDIV (signed) or UDIV (unsigned)	
Move data	Move		MOV{S} Rd, <Operand2>	N Z C	Rd := Operand2 See also Shift instructions	N
	NOT		MVN{S} Rd, <Operand2>	N Z C	Rd := 0xFFFFFFFF EOR Operand2	N
	top	T2	MOVT Rd, #<imm16>		Rd[31:16] := imm16, Rd[15:0] unaffected, imm16 range 0-65535	
	wide	T2	MOV Rd, #<imm16>		Rd[15:0] := imm16, Rd[31:16] = 0, imm16 range 0-65535	
	40-bit accumulator to register	XS	MRA RdLo, RdHi, Ac		RdLo := Ac[31:0], RdHi := Ac[39:32]	
	register to 40-bit accumulator	XS	MAR Ac, RdLo, RdHi		Ac[31:0] := RdLo, Ac[39:32] := RdHi	
Shift	Arithmetic shift right		ASR{S} Rd, Rm, <Rs|sh>	N Z C	Rd := ASR(Rm, Rs|sh) Same as MOV{S} Rd, Rm, ASR <Rs|sh>	N
	Logical shift left		LSL{S} Rd, Rm, <Rs|sh>	N Z C	Rd := LSL(Rm, Rs|sh) Same as MOV{S} Rd, Rm, LSL <Rs|sh>	N
	Logical shift right		LSR{S} Rd, Rm, <Rs|sh>	N Z C	Rd := LSR(Rm, Rs|sh) Same as MOV{S} Rd, Rm, LSR <Rs|sh>	N
	Rotate right		ROR{S} Rd, Rm, <Rs|sh>	N Z C	Rd := ROR(Rm, Rs|sh) Same as MOV{S} Rd, Rm, ROR <Rs|sh>	N
	Rotate right with extend		RRX{S} Rd, Rm	N Z C	Rd := RRX(Rm) Same as MOV{S} Rd, Rm, RRX	N
Count leading zeros		5	CLZ Rd, Rm		Rd := number of leading zeros in Rm	
Compare	Compare		CMP Rn, <Operand2>	N Z C V	Update CPSR flags on Rn – Operand2	N
	negative		CMN Rn, <Operand2>	N Z C V	Update CPSR flags on Rn + Operand2	N
Logical	Test		TST Rn, <Operand2>	N Z C	Update CPSR flags on Rn AND Operand2	N
	Test equivalence		TEQ Rn, <Operand2>	N Z C	Update CPSR flags on Rn EOR Operand2	
	AND		AND{S} Rd, Rn, <Operand2>	N Z C	Rd := Rn AND Operand2	N
	EOR		EOR{S} Rd, Rn, <Operand2>	N Z C	Rd := Rn EOR Operand2	N
	ORR		ORR{S} Rd, Rn, <Operand2>	N Z C	Rd := Rn OR Operand2	N
	ORN	T2	ORN{S} Rd, Rn, <Operand2>	N Z C	Rd := Rn OR NOT Operand2	T
	Bit Clear		BIC{S} Rd, Rn, <Operand2>	N Z C	Rd := Rn AND NOT Operand2	N

Operation		§	Assembler	Action	Notes
Bit field	Bit Field Clear	T2	`BFC Rd, #<lsb>, #<width>`	Rd[(width+lsb–1):lsb] := 0, other bits of Rd unaffected	
	Bit Field Insert	T2	`BFI Rd, Rn, #<lsb>, #<width>`	Rd[(width+lsb–1):lsb] := Rn[(width-1):0], other bits of Rd unaffected	
	Signed Bit Field Extract	T2	`SBFX Rd, Rn, #<lsb>, #<width>`	Rd[(width–1):0] = Rn[(width+lsb–1):lsb], Rd[31:width] = Replicate(Rn[width+lsb–1])	
	Unsigned Bit Field Extract	T2	`UBFX Rd, Rn, #<lsb>, #<width>`	Rd[(width–1):0] = Rn[(width+lsb–1):lsb], Rd[31:width] = Replicate(0)	
Pack	Pack halfword bottom + top	6	`PKHBT Rd, Rn, Rm{, LSL #<sh>}`	Rd[15:0] := Rn[15:0], Rd[31:16] := (Rm LSL sh)[31:16]. sh 0-31.	
	Pack halfword top + bottom	6	`PKHTB Rd, Rn, Rm{, ASR #<sh>}`	Rd[31:16] := Rn[31:16], Rd[15:0] := (Rm ASR sh)[15:0]. sh 1-32.	
Signed extend	Halfword to word	6	`SXTH Rd, Rm{, ROR #<sh>}`	Rd[31:0] := SignExtend((Rm ROR (8 * sh))[15:0]). sh 0-3.	N
	Two bytes to halfwords	6	`SXTB16 Rd, Rm{, ROR #<sh>}`	Rd[31:16] := SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
	Byte to word	6	`SXTB Rd, Rm{, ROR #<sh>}`	Rd[31:0] := SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	N
Unsigned extend	Halfword to word	6	`UXTH Rd, Rm{, ROR #<sh>}`	Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[15:0]). sh 0-3.	N
	Two bytes to halfwords	6	`UXTB16 Rd, Rm{, ROR #<sh>}`	Rd[31:16] := ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
	Byte to word	6	`UXTB Rd, Rm{, ROR #<sh>}`	Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	N
Signed extend with add	Halfword to word, add	6	`SXTAH Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[15:0]). sh 0-3.	
	Two bytes to halfwords, add	6	`SXTAB16 Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:16] := Rn[31:16] + SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
	Byte to word, add	6	`SXTAB Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
Unsigned extend with add	Halfword to word, add	6	`UXTAH Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[15:0]). sh 0-3.	
	Two bytes to halfwords, add	6	`UXTAB16 Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:16] := Rn[31:16] + ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
	Byte to word, add	6	`UXTAB Rd, Rn, Rm{, ROR #<sh>}`	Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3.	
Reverse	Bits in word	T2	`RBIT Rd, Rm`	For (i = 0; i < 32; i++) : Rd[i] = Rm[31– i]	
	Bytes in word	6	`REV Rd, Rm`	Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24]	N
	Bytes in both halfwords	6	`REV16 Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24]	N
	Bytes in low halfword, sign extend	6	`REVSH Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF	N
Select	Select bytes	6	`SEL Rd, Rn, Rm`	Rd[7:0] := Rn[7:0] if GE[0] = 1, else Rd[7:0] := Rm[7:0] Bits[15:8], [23:16], [31:24] selected similarly by GE[1], GE[2], GE[3]	
If-Then	If-Then	T2	`IT{pattern} {cond}`	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else). The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table **Condition Field** for available condition codes.	T U
Branch	Branch		`B <label>`	PC := label. label is this instruction ±32MB (T2: ±16MB, T: –252 - +256B)	N, B
	with link		`BL <label>`	LR := address of next instruction, PC := label. label is this instruction ±32MB (T2: ±16MB).	
	and exchange	4T	`BX Rm`	PC := Rm. Target is Thumb if Rm[0] is 1, ARM if Rm[0] is 0.	N
	with link and exchange (1)	5T	`BLX <label>`	LR := address of next instruction, PC := label, Change instruction set. label is this instruction ±32MB (T2: ±16MB).	C
	with link and exchange (2)	5	`BLX Rm`	LR := address of next instruction, PC := Rm[31:1]. Change to Thumb if Rm[0] is 1, to ARM if Rm[0] is 0.	N
	and change to Jazelle state	5J	`BXJ Rm`	Change to Jazelle state if available	
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== or !=} 0 then PC := label. label is (this instruction + 4-130).	N T U
	Table Branch Byte	T2	`TBB [Rn, Rm]`	PC = PC + ZeroExtend(Memory(Rn + Rm, 1) << 1). Branch range 4-512. Rn can be PC.	T U
	Table Branch Halfword	T2	`TBH [Rn, Rm, LSL #1]`	PC = PC + ZeroExtend(Memory(Rn + Rm << 1, 2) << 1). Branch range 4-131072. Rn can be PC.	T U
Move to or from PSR	PSR to register		`MRS Rd, <PSR>`	Rd := PSR	
	register to PSR		`MSR <PSR>_<fields>, Rm`	PSR := Rm (selected bytes only)	
	immediate to PSR		`MSR <PSR>_<fields>, #<imm8m>`	PSR := immed_8r (selected bytes only)	
Processor state change	Change processor state	6	`CPSID <iflags> {, #<p_mode>}`	Disable specified interrupts, optional change mode.	U, N
		6	`CPSIE <iflags> {, #<p_mode>}`	Enable specified interrupts, optional change mode.	U, N
	Change processor mode	6	`CPS #<p_mode>`		U
	Set endianness	6	`SETEND <endianness>`	Sets endianness for loads and saves. `<endianness>` can be `BE` (Big Endian) or `LE` (Little Endian).	U, N

Single data item loads and stores		§	Assembler	Action if <op> is LDR	Action if <op> is STR	Notes
Load or store word, byte or halfword	Immediate offset		<op>{size}{T} Rd, [Rn {, #<offset>}]{!}	Rd := [address, size]	[address, size] := Rd	1, N
	Post-indexed, immediate		<op>{size}{T} Rd, [Rn], #<offset>	Rd := [address, size]	[address, size] := Rd	2
	Register offset		<op>{size} Rd, [Rn, +/-Rm {, <opsh>}]{!}	Rd := [address, size]	[address, size] := Rd	3, N
	Post-indexed, register		<op>{size}{T} Rd, [Rn], +/-Rm {, <opsh>}	Rd := [address, size]	[address, size] := Rd	4
	PC-relative		<op>{size} Rd, <label>	Rd := [label, size]	Not available	5, N
Load or store doubleword	Immediate offset	5E	<op>D Rd1, Rd2, [Rn {, #<offset>}]{!}	Rd1 := [address], Rd2 := [address + 4]	[address] := Rd1, [address + 4] := Rd2	6, 9
	Post-indexed, immediate	5E	<op>D Rd1, Rd2, [Rn], #<offset>	Rd1 := [address], Rd2 := [address + 4]	[address] := Rd1, [address + 4] := Rd2	6, 9
	Register offset	5E	<op>D Rd1, Rd2, [Rn, +/-Rm {, <opsh>}]{!}	Rd1 := [address], Rd2 := [address + 4]	[address] := Rd1, [address + 4] := Rd2	7, 9
	Post-indexed, register	5E	<op>D Rd1, Rd2, [Rn], +/-Rm {, <opsh>}	Rd1 := [address], Rd2 := [address + 4]	[address] := Rd1, [address + 4] := Rd2	7, 9
	PC-relative	5E	<op>D Rd1, Rd2, <label>	Rd1 := [label], Rd2 := [label + 4]	Not available	8, 9

Preload data or instruction		§ (PLD)	§ (PLI)	Assembler	Action if <op> is PLD	Action if <op> is PLI	Notes
	Immediate offset	5E	7	<op> [Rn {, #<offset>}]	Preload [address, 32] (data)	Preload [address, 32] (instruction)	1, C
	Register offset	5E	7	<op> [Rn, +/-Rm {, <opsh>}]	Preload [address, 32] (data)	Preload [address, 32] (instruction)	3, C
	PC-relative	5E	7	<op> <label>	Preload [label, 32] (data)	Preload [label, 32] (instruction)	5, C

Other memory operations		§	Assembler	Action	Notes
Load multiple	Block data load		LDM{IA|IB|DA|DB} Rn{!}, <reglist-PC>	Load list of registers from [Rn]	N, I
	return (and exchange)		LDM{IA|IB|DA|DB} Rn{!}, <reglist+PC>	Load registers, PC := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	I
	and restore CPSR		LDM{IA|IB|DA|DB} Rn{!}, <reglist+PC>^	Load registers, branch (§ 5T: and exchange), CPSR := SPSR. Exception modes only.	I
	User mode registers		LDM{IA|IB|DA|DB} Rn, <reglist-PC>^	Load list of User mode registers from [Rn]. Privileged modes only.	I
Pop			POP <reglist>	Canonical form of LDM SP!, <reglist>	N
Load exclusive	Semaphore operation	6	LDREX Rd, [Rn]	Rd := [Rn], tag address as exclusive access. Outstanding tag set if not shared address. Rd, Rn not PC.	
	Halfword or Byte	6K	LDREX{H|B} Rd, [Rn]	Rd[15:0] := [Rn] or Rd[7:0] := [Rn], tag address as exclusive access. Outstanding tag set if not shared address. Rd, Rn not PC.	
	Doubleword	6K	LDREXD Rd1, Rd2, [Rn]	Rd1 := [Rn], Rd2 := [Rn+4], tag addresses as exclusive access Outstanding tags set if not shared addresses. Rd1, Rd2, Rn not PC.	9
Store multiple	Push, or Block data store		STM{IA|IB|DA|DB} Rn{!}, <reglist>	Store list of registers to [Rn]	N, I
	User mode registers		STM{IA|IB|DA|DB} Rn{!}, <reglist>^	Store list of User mode registers to [Rn]. Privileged modes only.	I
Push			PUSH <reglist>	Canonical form of STMDB SP!, <reglist>	N
Store exclusive	Semaphore operation	6	STREX Rd, Rm, [Rn]	If allowed, [Rn] := Rm, clear exclusive tag, Rd := 0. Else Rd := 1. Rd, Rm, Rn not PC.	
	Halfword or Byte	6K	STREX{H|B} Rd, Rm, [Rn]	If allowed, [Rn] := Rm[15:0] or [Rn] := Rm[7:0], clear exclusive tag, Rd := 0. Else Rd := 1 Rd, Rm, Rn not PC.	
	Doubleword	6K	STREXD Rd, Rm1, Rm2, [Rn]	If allowed, [Rn] := Rm1, [Rn+4] := Rm2, clear exclusive tags, Rd := 0. Else Rd := 1 Rd, Rm1, Rm2, Rn not PC.	9
Clear exclusive		6K	CLREX	Clear local processor exclusive tag	C

Notes: availability and range of options for Load, Store, and Preload operations

Note	ARM Word, B, D	ARM SB, H, SH	ARM T, BT	Thumb-2 Word, B, SB, H, SH, D	Thumb-2 T, BT, SBT, HT, SHT
1	offset: – 4095 to +4095	offset: –255 to +255	Not available	offset: –255 to +255 if writeback, –255 to +4095 otherwise	offset: 0 to +255, writeback not allowed
2	offset: – 4095 to +4095	offset: –255 to +255	offset: – 4095 to +4095	offset: –255 to +255	Not available
3	Full range of {, <opsh>}	{, <opsh>} not allowed	Not available	<opsh> restricted to LSL #<sh>, <sh> range 0 to 3	Not available
4	Full range of {, <opsh>}	{, <opsh>} not allowed	Full range of {, <opsh>}	Not available	Not available
5	label within +/– 4092 of current instruction	Not available	Not available	label within +/– 4092 of current instruction	Not available
6	offset: –255 to +255	-	-	offset: –1020 to +1020, must be multiple of 4.	-
7	{, <opsh>} not allowed	-	-	Not available	-
8	label within +/– 252 of current instruction	-	-	Not available	-
9	Rd1 even, and not r14, Rd2 == Rd1 + 1.	-	-	Rd1 != PC, Rd2 != PC	-

Coprocessor operations	§	Assembler		Action	Notes
Data operations		CDP{2} <copr>, <op1>, CRd, CRn, CRm{, <op2>}		Coprocessor defined	C2
Move to ARM register from coprocessor		MRC{2} <copr>, <op1>, Rd, CRn, CRm{, <op2>}		Coprocessor defined	C2
Two ARM register move	5E	MRRC <copr>, <op1>, Rd, Rn, CRm		Coprocessor defined	
Alternative two ARM register move	6	MRRC2 <copr>, <op1>, Rd, Rn, CRm		Coprocessor defined	C
Move to coproc from ARM reg		MCR{2} <copr>, <op1>, Rd, CRn, CRm{, <op2>}		Coprocessor defined	C2
Two ARM register move	5E	MCRR <copr>, <op1>, Rd, Rn, CRm		Coprocessor defined	
Alternative two ARM register move	6	MCRR2 <copr>, <op1>, Rd, Rn, CRm		Coprocessor defined	C
Loads and stores, pre-indexed		<op>{2} <copr>, CRd, [Rn, #+/-<offset8*4>]{!}	op: LDC or STC. offset: multiple of 4 in range 0 to 1020.	Coprocessor defined	C2
Loads and stores, zero offset		<op>{2} <copr>, CRd, [Rn] {, 8-bit copro. option}	op: LDC or STC.	Coprocessor defined	C2
Loads and stores, post-indexed		<op>{2} <copr>, CRd, [Rn], #+/-<offset8*4>	op: LDC or STC. offset: multiple of 4 in range 0 to 1020.	Coprocessor defined	C2

Miscellaneous operations		§	Assembler	Action	Notes
Swap word			SWP Rd, Rm, [Rn]	temp := [Rn], [Rn] := Rm, Rd := temp.	D
Swap byte			SWPB Rd, Rm, [Rn]	temp := ZeroExtend([Rn][7:0]), [Rn][7:0] := Rm[7:0], Rd := temp	D
Store return state		6	SRS{IA|IB|DA|DB} SP{!}, #<p_mode>	[SPm] := LR, [SPm + 4] := CPSR	C, I
Return from exception		6	RFE{IA|IB|DA|DB} Rn{!}	PC := [Rn], CPSR := [Rn + 4]	C, I
Breakpoint		5	BKPT <imm16>	Prefetch abort *or* enter debug state. 16-bit bitfield encoded in instruction.	C, N
Secure Monitor Call		Z	SMC <imm16>	Secure Monitor Call exception. 16-bit bitfield encoded in instruction. Formerly SMI.	
Supervisor Call			SVC <imm24>	Supervisor Call exception. 24-bit bitfield encoded in instruction. Formerly SWI.	N
No operation		6	NOP	None, might not even consume any time.	N
Hints	Debug Hint	7	DBG	Provide hint to debug and related systems.	
	Data Memory Barrier	7	DMB	Ensure the order of observation of memory accesses.	C
	Data Synchronization Barrier	7	DSB	Ensure the completion of memory accesses,	C
	Instruction Synchronization Barrier	7	ISB	Flush processor pipeline and branch prediction logic.	C
	Set event	T2	SEV	Signal event in multiprocessor system. NOP if not implemented.	N
	Wait for event	T2	WFE	Wait for event, IRQ, FIQ, Imprecise abort, or Debug entry request. NOP if not implemented.	N
	Wait for interrupt	T2	WFI	Wait for IRQ, FIQ, Imprecise abort, or Debug entry request. NOP if not implemented.	N
	Yield	T2	YIELD	Yield control to alternative thread. NOP if not implemented.	N

Notes			
A	Not available in Thumb state.	**N**	Some or all forms of this instruction are 16-bit (Narrow) instructions in Thumb-2 code. For details see the *Thumb 16-bit Instruction Set (UAL) Quick Reference Card*.
B	Can be conditional in Thumb state without having to be in an IT block.		
C	Condition codes are not allowed in ARM state.	**P**	Rn can be the PC in Thumb state in this instruction.
C2	The optional 2 is available from ARMv5. It provides an alternative operation. Condition codes are not allowed for the alternative form in ARM state.	**Q**	Sets the Q flag if saturation (addition or substraction) or overflow (multiplication) occurs. Read and reset the Q flag using MRS and MSR.
D	Deprecated. Use LDREX and STREX instead.	**R**	<sh> range is 1-32 in the ARM instruction.
G	Updates the four GE flags in the CPSR based on the results of the individual operations.	**S**	The S modifier is not available in the Thumb-2 instruction.
I	IA is the default, and is normally omitted.	**T**	Not available in ARM state.
L	ARM: <imm8m>. 16-bit Thumb: multiple of 4 in range 0-1020. 32-bit Thumb: 0-4095.	**U**	Not allowed in an IT block. Condition codes not allowed in either ARM or Thumb state.

ARM architecture versions

n	ARM architecture version *n* and above
*n*T, *n*J	T or J variants of ARM architecture version *n* and above
5E	ARM v5E, and 6 and above
T2	All Thumb-2 versions of ARM v6 and above
6K	ARMv6K and above for ARM instructions, ARMv7 for Thumb
Z	All Security extension versions of ARMv6 and above
RM	ARMv7-R and ARMv7-M only
XS	XScale coprocessor instruction

Flexible Operand 2

Immediate value	#<imm8m>
Register, optionally shifted by constant (see below)	Rm {, <opsh>}
Register, logical shift left by register	Rm, LSL Rs
Register, logical shift right by register	Rm, LSR Rs
Register, arithmetic shift right by register	Rm, ASR Rs
Register, rotate right by register	Rm, ROR Rs

Register, optionally shifted by constant

(No shift)	Rm	Same as Rm, LSL #0
Logical shift left	Rm, LSL #<shift>	Allowed shifts 0-31
Logical shift right	Rm, LSR #<shift>	Allowed shifts 1-32
Arithmetic shift right	Rm, ASR #<shift>	Allowed shifts 1-32
Rotate right	Rm, ROR #<shift>	Allowed shifts 1-31
Rotate right with extend	Rm, RRX	

PSR fields

(use at least one suffix)

Suffix	Meaning	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extension field mask byte	PSR[15:8]

Condition Field

Mnemonic	Description	Description (VFP)
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

All ARM instructions (except those with Note C or Note U) can have any one of these condition codes after the instruction mnemonic (that is, before the first space in the instruction as shown on this card). This condition is encoded in the instruction.

All Thumb-2 instructions (except those with Note U) can have any one of these condition codes after the instruction mnemonic. This condition is encoded in a preceding IT instruction (except in the case of conditional Branch instructions). Condition codes in instructions must match those in the preceding IT instruction.

On processors without Thumb-2, the only Thumb instruction that can have a condition code is B <label>.

Processor Modes

16	User
17	FIQ Fast Interrupt
18	IRQ Interrupt
19	Supervisor
23	Abort
27	Undefined
31	System

Prefixes for Parallel Instructions

S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results

1997

http://web.njit.edu/~baltrush/arm_stuff/ARMInst.ppt

4.5.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction and the shift amount is instruction-specified, the PC value will be the address of the instruction plus 8 bytes.

For any register-controlled shift instructions, neither Rn nor Rm may be R15.

Instruction set orthogonality

Instruction set orthogonality is defined by two characteristics: *independence* and *consistency*. An independent instruction set does not contain any redundant instructions. That is, each instruction performs a unique function, and does not duplicate the function of another instruction. Also, the opcode/operand relationship is independent and consistent in the sense that any operand can be used with any opcode. Ideally, all operands can equally well be utilized with all the opcodes, and all addressing modes can be consistently used will all operands. Basically, the uniformity offered by an orthogonal instruction set makes the task of compiler development easier. The instruction set should be complete while maintaining a high degree of orthogonality.

—Sajjan G. Shiva, **Computer Organization, Design, and Architecture, Fourth Edition**

The orthogonality of an instruction set is the regularity with which any op-code (without data-size encoding within the op-code itself) can be used with any machine-primitive data-type and addressing mode. The orthogonality of the instruction set makes the architecture easy to learn and program. It reduces the time required to write programs but may result in lower code density. Irregularities adversely affect code-generation efficiency.

Mahadevan Ganapathi & James R. Goodman

Orthogonal instructions

An instruction set is said to be **orthogonal** if each choice in the building of an instruction is independent of the other choices. Since add and subtract are similar operations, one would expect to be able to use them in similar contexts. If add uses a 3-address format with register addresses, so should subtract, and in neither case should there be any peculiar restrictions on the registers which may be used.

An orthogonal instruction set is easier for the assembly language programmer to learn and easier for the compiler writer to target. The hardware implementation will usually be more efficient too.

Stephen Byram Furber, “ARM System-on-Chip Architecture”

Повечето съвременни микропроцесори (включително и „ARM“) имат висока степен на ортогоналност на системата машинни команди. Но при x86 не е така. Например при 8086 от общо 96 команди ортогонални са само 36. Това се дължи на произхода на този микропроцесор от семейството 8008/8080/8085 с регистър-акумулатор (A, превърнал се в AL/AX и регистрови двойки BC, DE и HL, превърнали се в BH:BL, CH:CL и DH:DL при 8086).

Предимства на ARM пред x86

Архитектурата ARM е RISC и е създадена по-късно от x86, която е CISC. Въпреки това, следните предимства правят програмите за ARM по-кратки от тези за 80x86:

1. Наличието на *13 регистъра* за обща употреба срещу 7 за 80x86.
2. Наличието на *3 до 4 операнда* на команда при аритметично-логическите операции срещу 2 за 80x86 и дори само 1 за умножението и делението (вярно е, че командата *IMUL* (80186+) има 3-операнден вариант, а някои нови FMA4- и XOP-команди имат до 5 операнда, но те са рядко срещани, специализирани и сложни).
3. Възможността всяка команда да бъде направена *условна*.
4. Възможността за избор дали командата да *променя флаговете* или не.
5. Възможността да се работи с *изместено* копие на десния операнд.
6. *Ортогоналният* набор от команди и адресни режими (на 80x86 е неортогонален).

Недостатъци на ARM спрямо x86

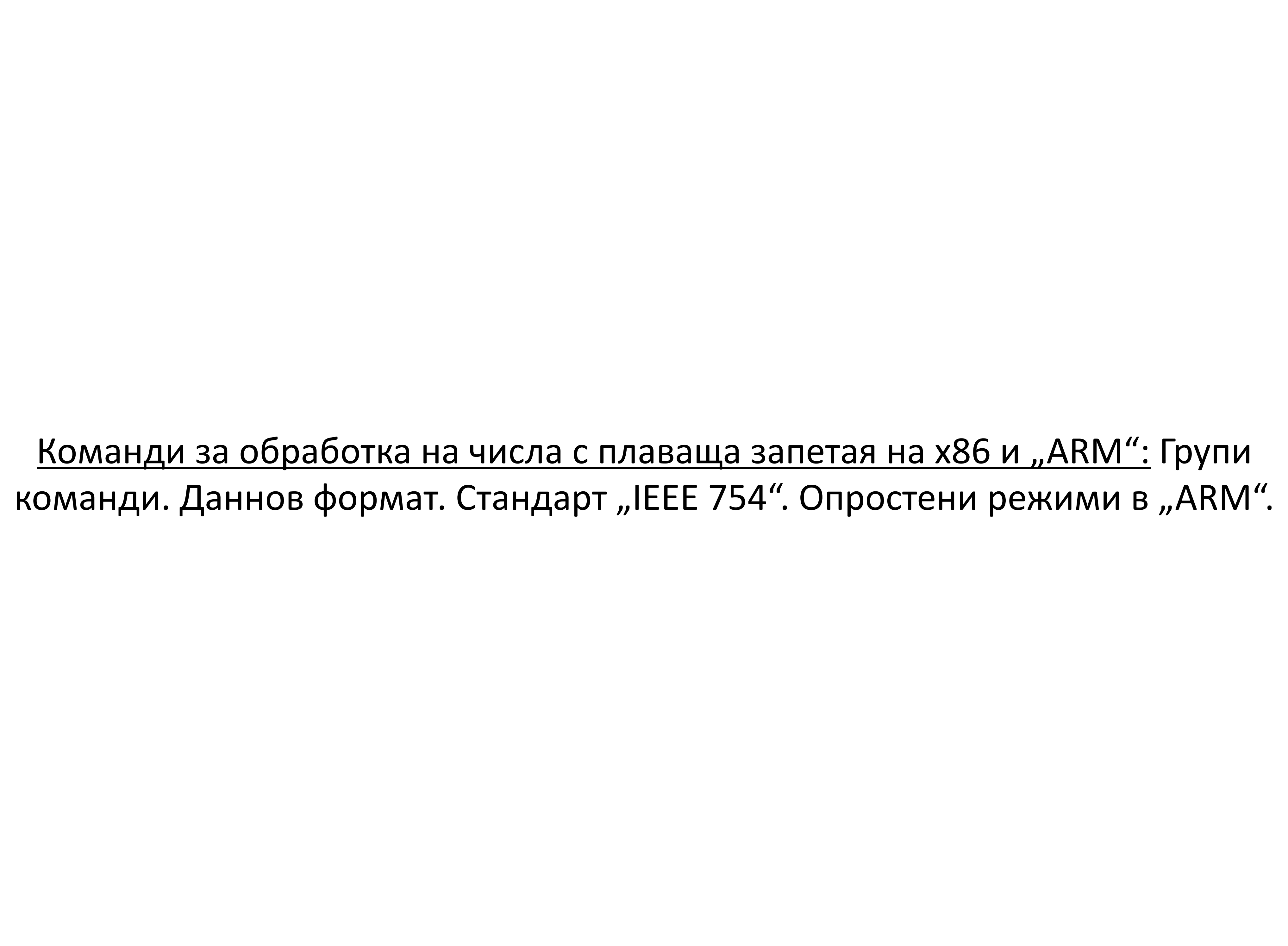
1. Няма трикомпонентен адресен режим (с 2 адресни регистъра плюс отместване-константа), какъвто има при x86.

2. Няма команда, която да променя флаг Z, без да променя флаг C. Това затруднява запазването на преноса между итерациите на цикъла.

3. Няма команда за размяна на съдържанието на 2 регистъра.

4. Няма команда за получаване на остатъка от целочислено деление.

5. Флагът C получава инверсна стойност след изваждане и сравнение, защото в ARM няма субтрактор; има само суматор. (Но това е по-скоро особеност, отколкото недостатък.)


Команди за обработка на числа с плаваща запетая на x86 и „ARM“: Групи команди. Даннов формат. Стандарт „IEEE 754“. Опростени режими в „ARM“.

Rounding mode

The IEEE 754 standard requires all calculations to be performed as if to an infinite precision. For example, a multiply of two single-precision values must accurately calculate the significand to twice the number of bits of the significand. To represent this value in the destination precision, rounding of the significand is often required. The IEEE 754 standard specifies four rounding modes.

In round-to-nearest mode, the result is rounded at the halfway point, with the tie case rounding up if it would clear the least significant bit of the significand, making it even. Round-towards-zero mode chops any bits to the right of the significand, always rounding down, and is used by the C, C++, and Java languages in integer conversions. Round-towards-plus-infinity mode and round-towards-minus-infinity mode are used in interval arithmetic.

NaN Not a number. A symbolic entity encoded in a floating-point format that has the maximum exponent field and a nonzero fraction. An SNaN causes an invalid operand exception if used as an operand and a most significant fraction bit of zero. A QNaN propagates through almost every arithmetic operation without signaling exceptions and has a most significant fraction bit of one.

Signaling NaNs

Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling Nans can be used in debugging, to track down some uses of uninitialized variables.

Quiet NaN

Is a NaN that propagates unchanged through most floating-point operations.

NaN also means " Not any Number " ; NaN does not represent the set of all real numbers, which is an interval for which the appropriate representation is provided by a scheme called " Interval Arithmetic."

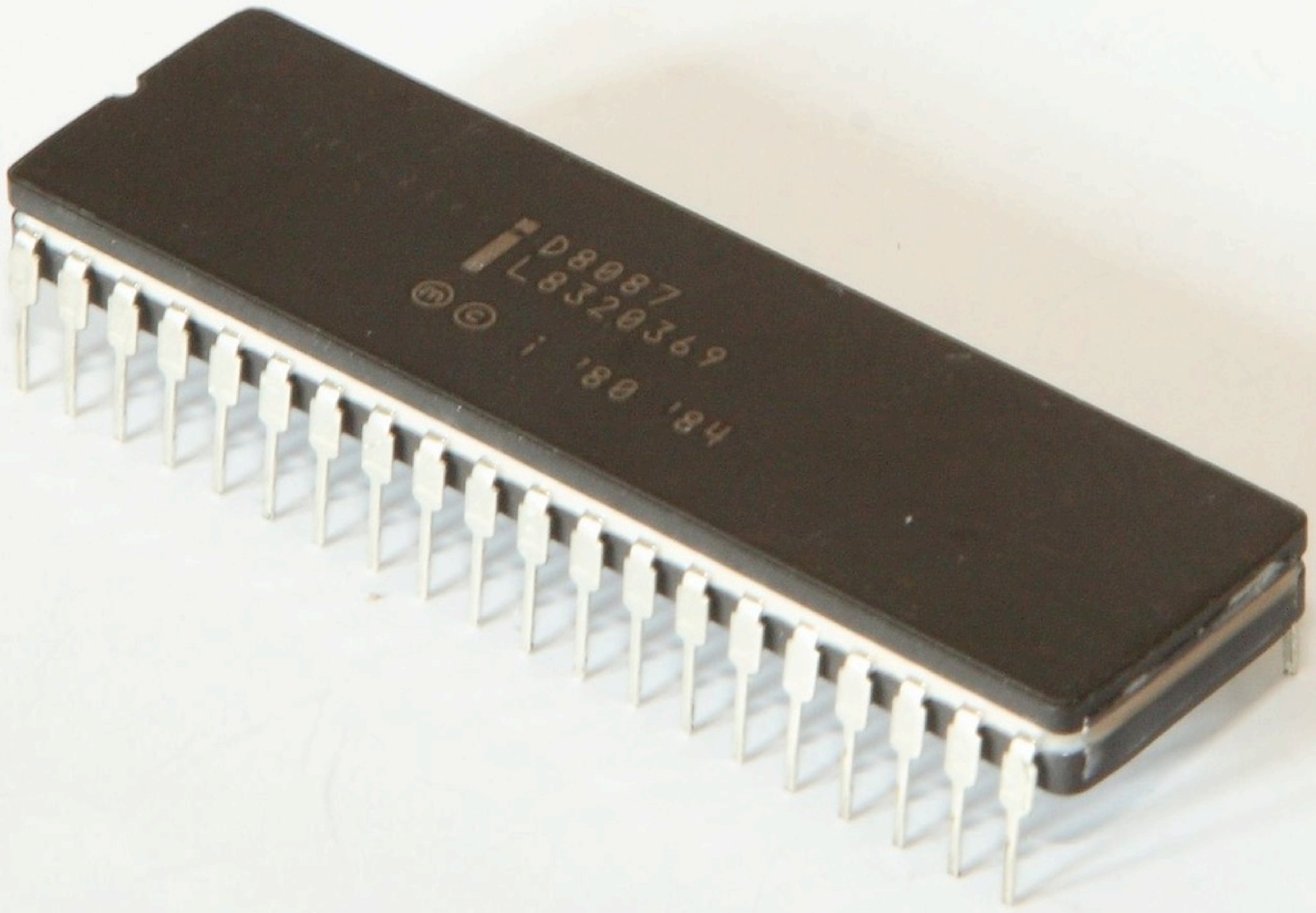
NaN must not be confused with " Undefined." On the contrary, IEEE 754 defines NaN perfectly well even though most language standards ignore and many compilers deviate from that definition. The deviations usually afflict relational expressions, discussed below. Arithmetic operations upon NaNs other than SNaNs (see below) never signal INVALID, and always produce NaN unless replacing every NaN operand by any finite or infinite real values would produce the same finite or infinite floating-point result independent of the replacements.

For example, 0*NaN must be NaN because 0*∞ is an INVALID operation (NaN). On the other hand, for hypot(x, y) := √(x*x + y*y) we find that hypot(∞, y) = +∞ for all real y , finite or not, and deduce that hypot(∞, NaN) = +∞ too; naive implementations of hypot may do differently.

NaNs were not invented out of whole cloth. Konrad Zuse tried similar ideas in the late 1930s; Seymour Cray built " Indefinites " into the CDC 6600 in 1963; then DEC put " Reserved Operands " into their PDP-11 and VAX. But nobody used them because they trap when touched. NaNs do not trap (unless they are " Signaling " SNaNs, which exist mainly for political reasons and are rarely used); NaNs propagate through most computations. Consequently they do get used.

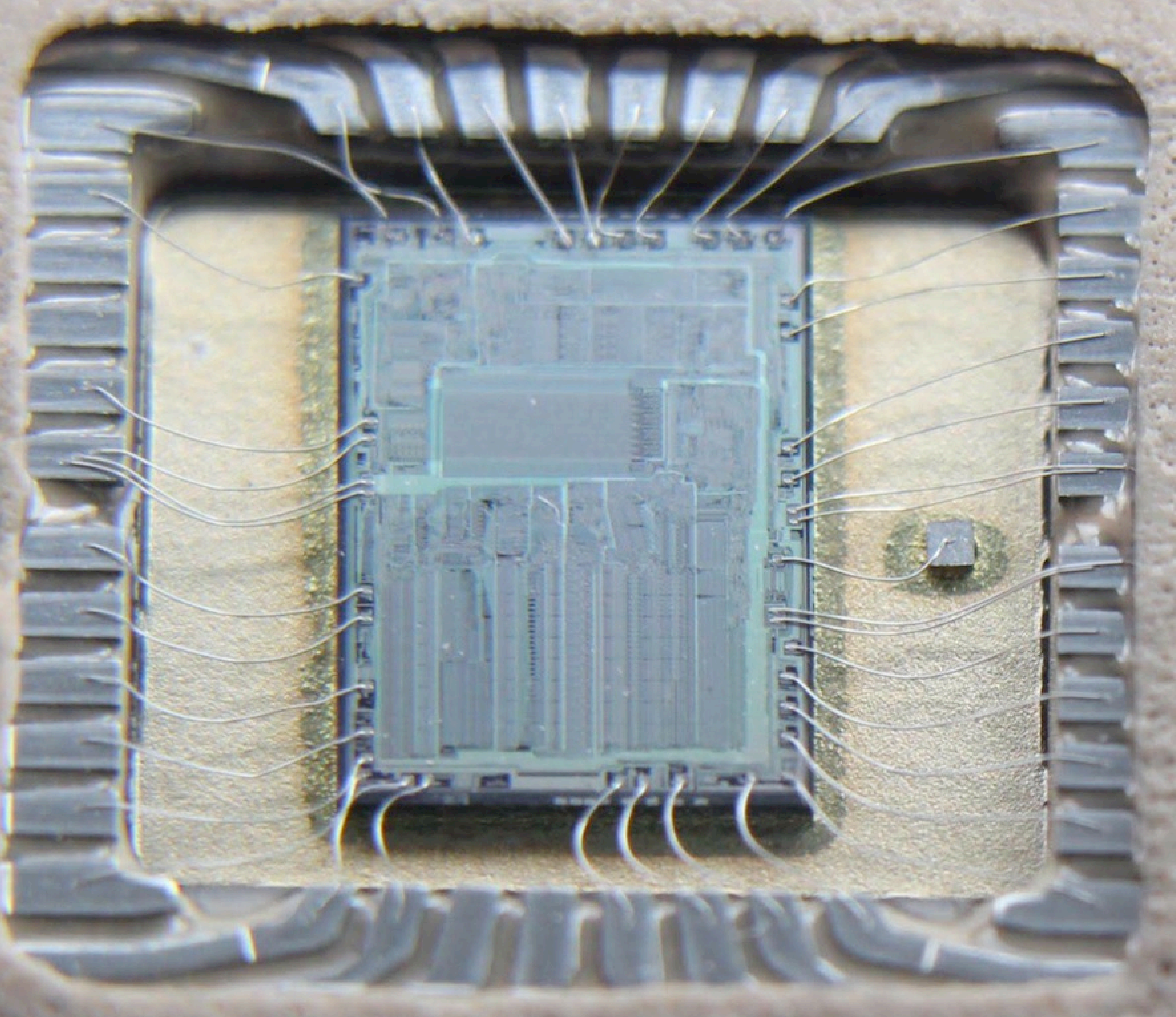
Perhaps NaNs are widely misunderstood because they are not needed for mathematical analysis, whose sequencing is entirely logical; they are needed only for computation, with temporal sequencing that can be hard to revise, harder to reverse. NaNs must conform to mathematically consistent rules that were deduced, not invented arbitrarily, in 1977 during the design of the Intel 8087 that preceded IEEE 754. What had been missing from computation but is now supplied by NaNs is an opportunity (not obligation) for software (especially when searching) to follow an unexceptional path (no need for exotic control structures) to a point where an exceptional event can be appraised after the event, when additional evidence may have accrued. Deferred judgments are *usually* better judgments but *not always*, alas.

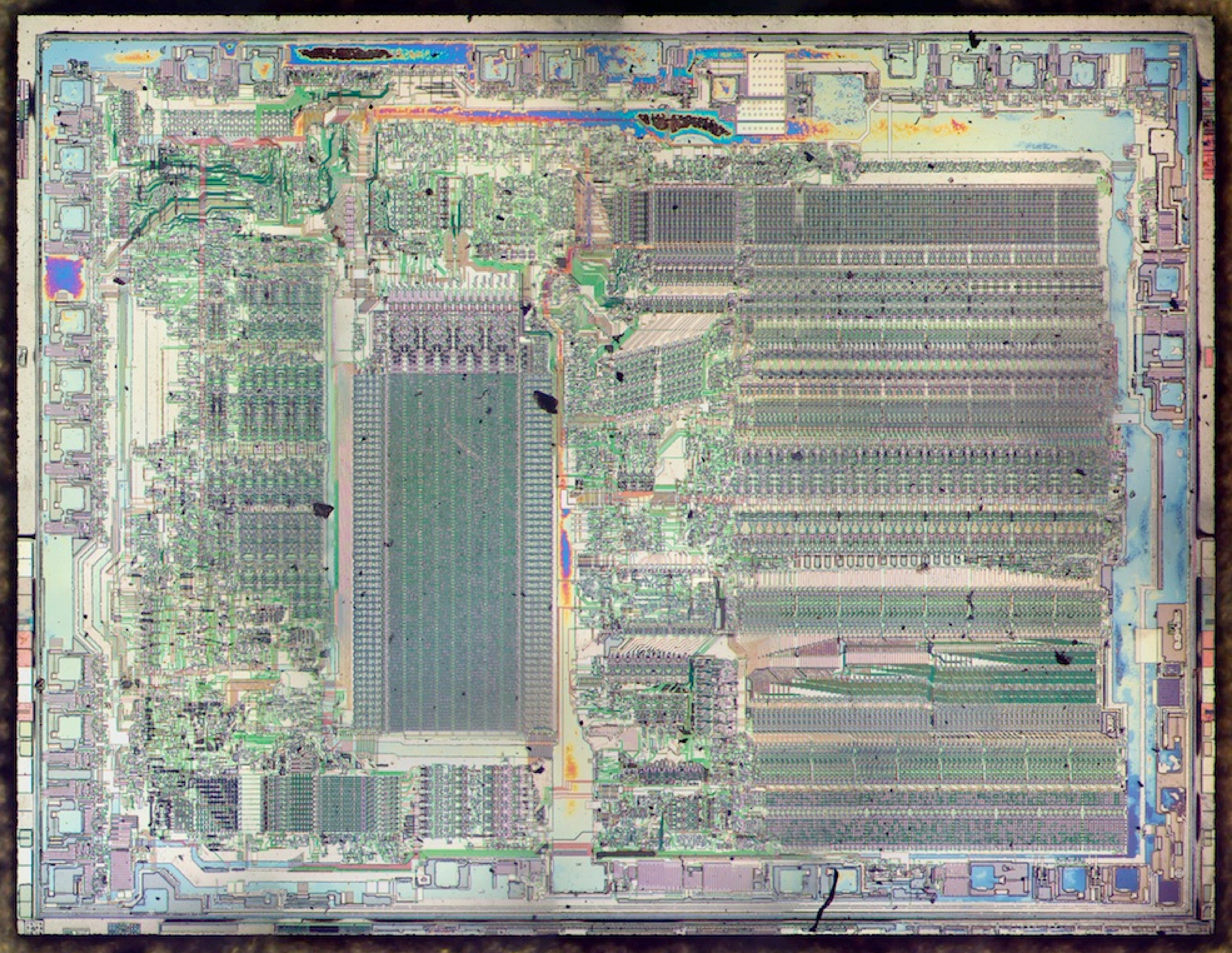
Prof. William Kahan, IEEE 754 architect

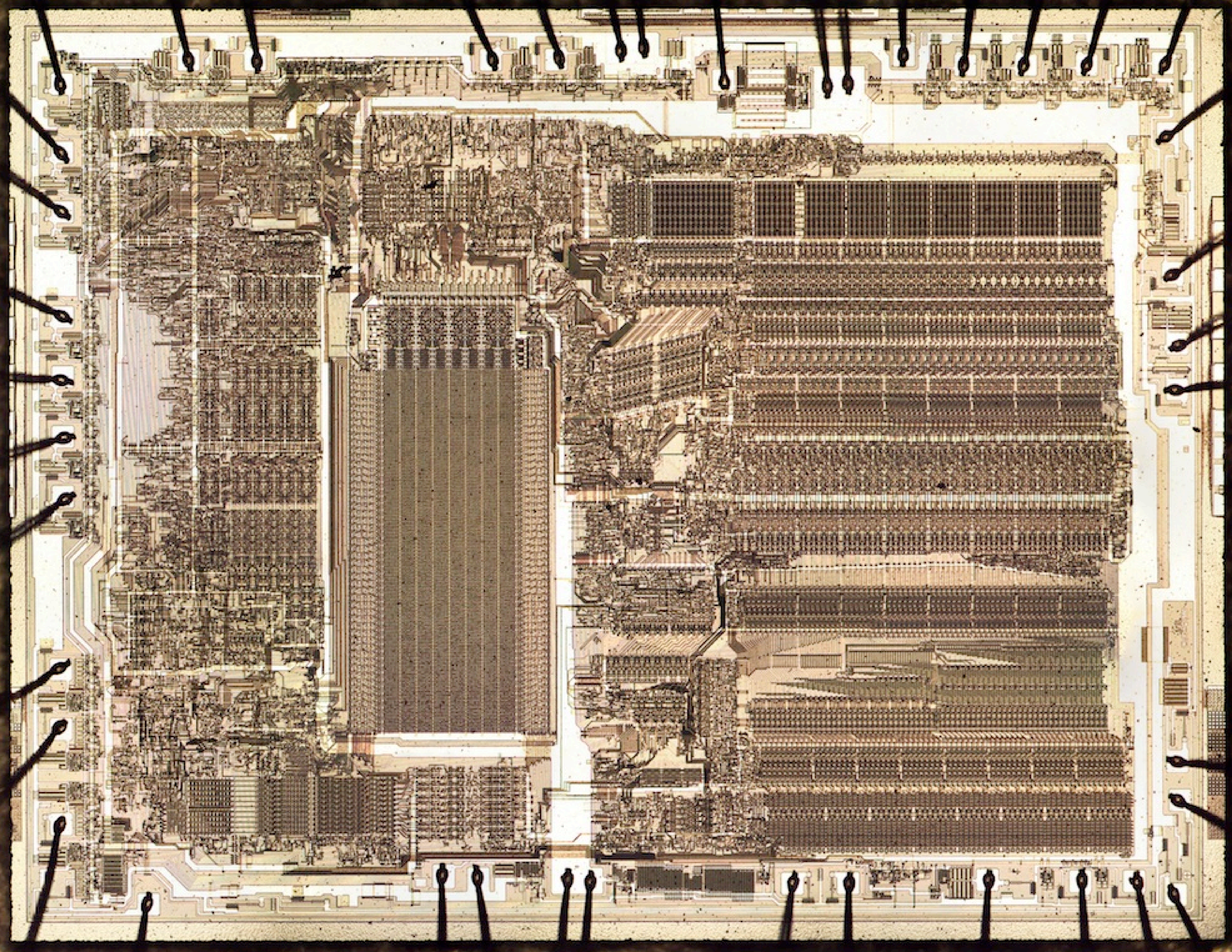

D8087
L8320369
i '80 '84

IBM

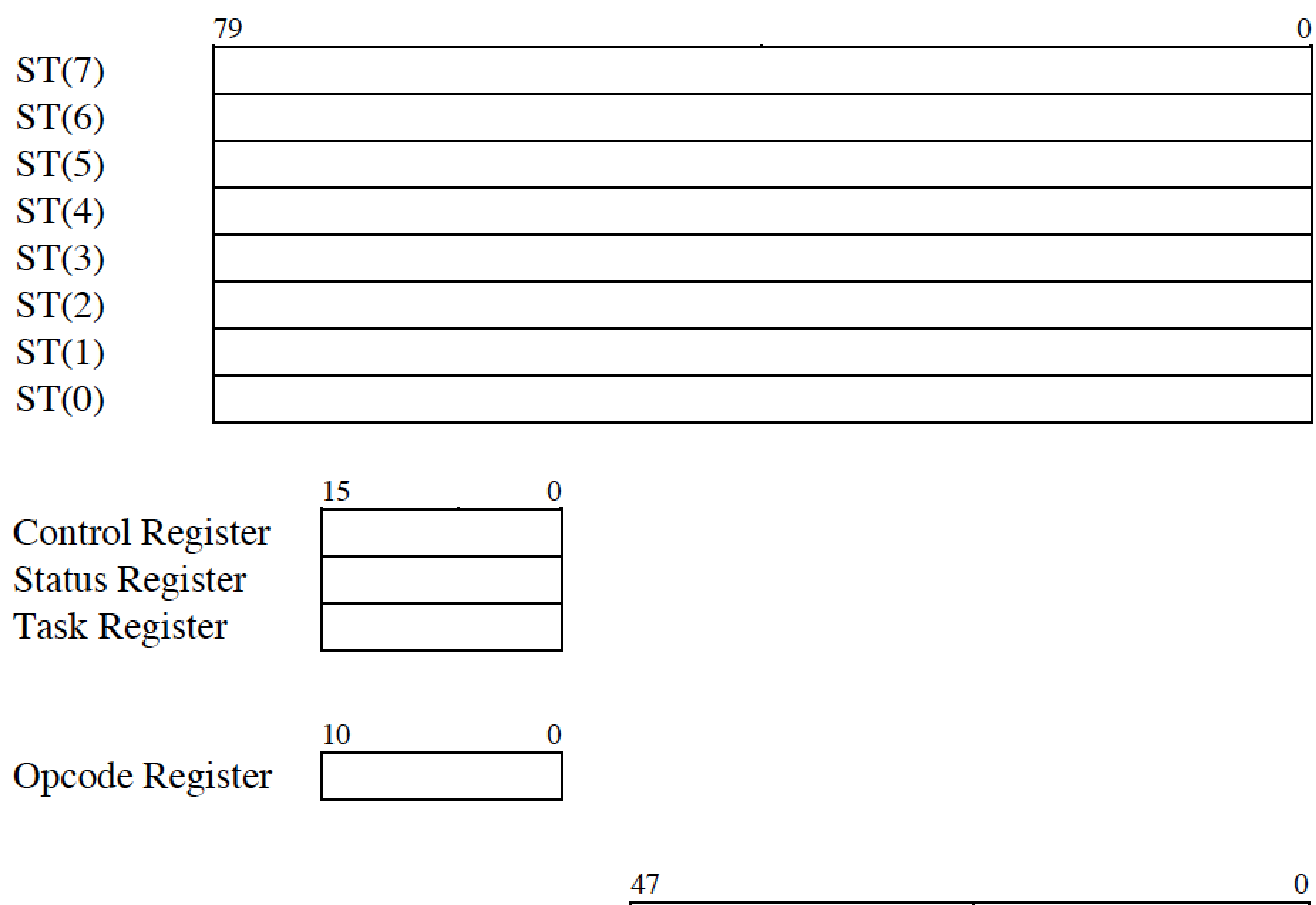
C8087
L5480311
©INTEL '80 '84







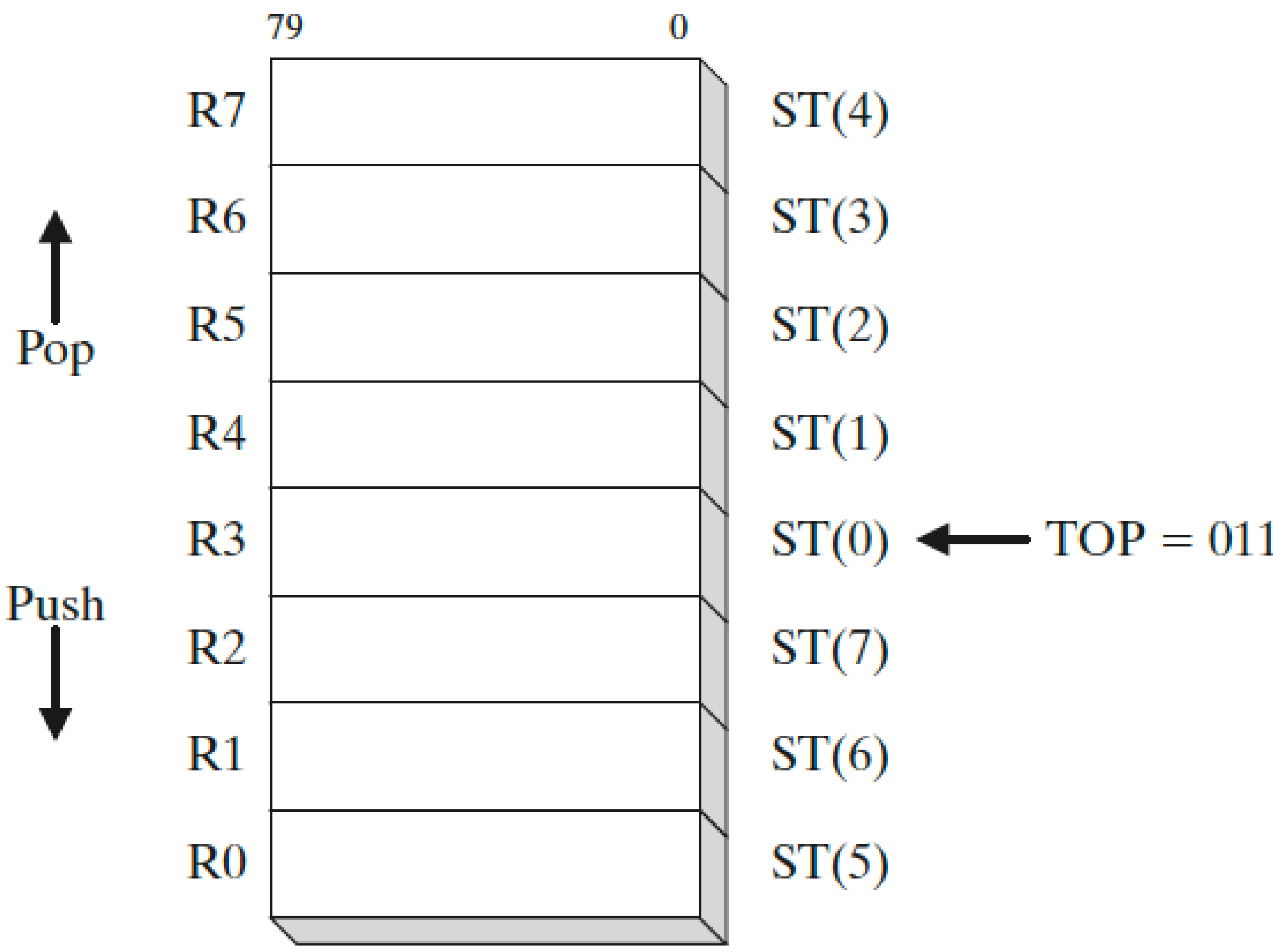
FPU Registers



FPU Data Registers

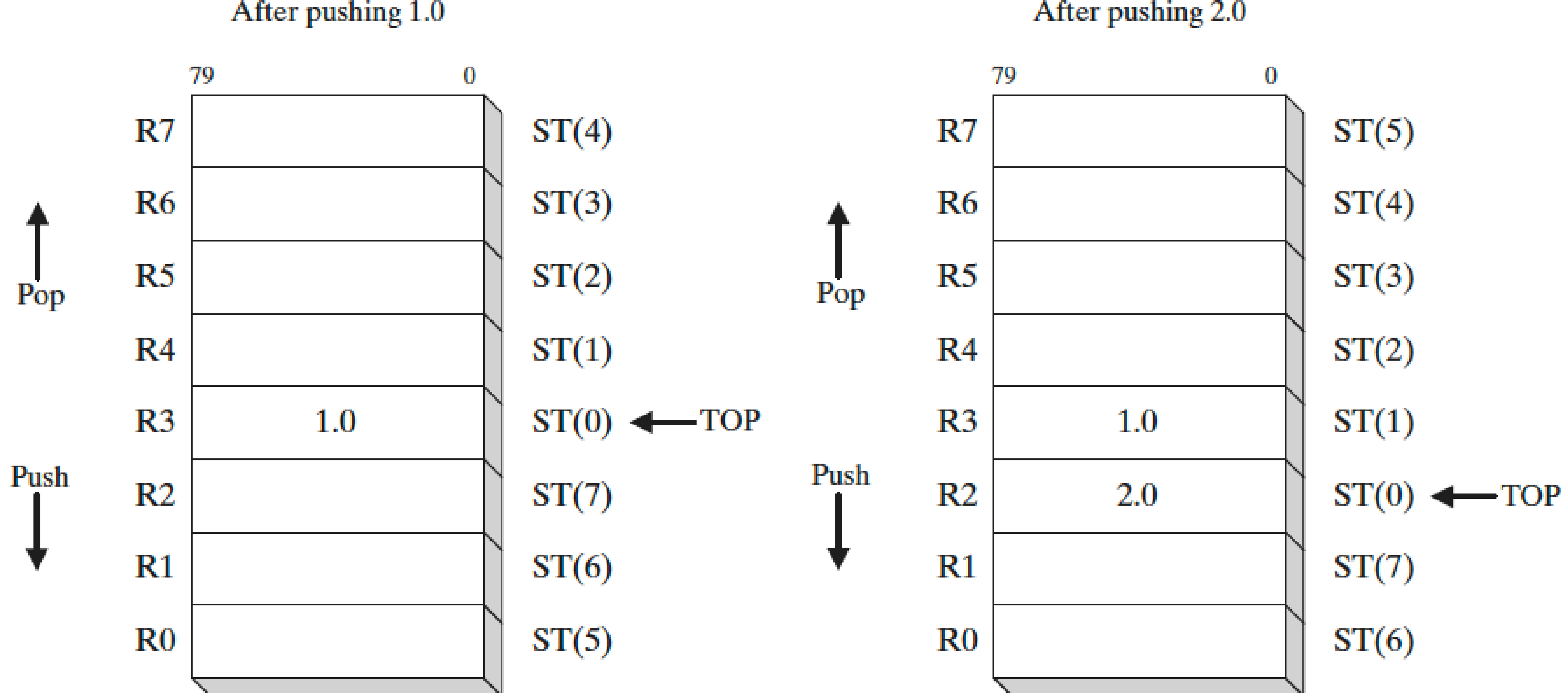
The FPU has eight individually addressable 80-bit data registers named R0 through R7 (see Figure 12–3). Together, they are called a *register stack*. A three-bit field named TOP in the FPU status word identifies the register number that is currently the top of the stack. In Figure 12–3, for example, TOP equals binary 011, identifying R3 as the top of the stack. This stack location is also known as ST(0) (or simply ST) when writing floating-point instructions. The last register is ST(7).

Figure 12–3 Floating-Point Data Register Stack.



As we might expect, a *push* operation (also called *load*) decrements TOP by 1 and copies an operand into the register identified as ST(0). If TOP equals 0 before a push, TOP wraps around to register R7. A *pop* operation (also called *store*) copies the data at ST(0) into an operand, then adds 1 to TOP. If TOP equals 7 before the pop, it wraps around to register R0. If loading a value into the stack would result in overwriting existing data in the register stack, a *floating-point exception* is generated. Figure 12-4 shows the same stack after 1.0 and 2.0 have been pushed (loaded) on the stack.

Figure 12–4 FPU stack after pushing 1.0 and 2.0.



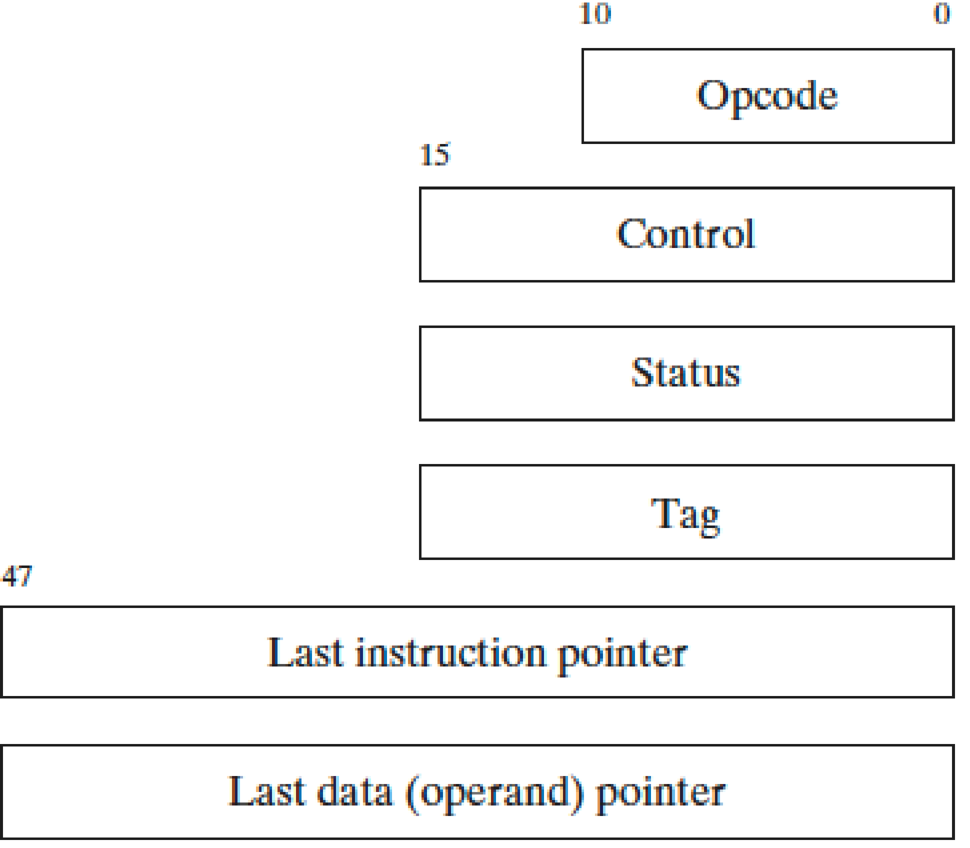
Although it is interesting to understand how the FPU implements the stack using a limited set of registers, we need only focus on the ST(*n*) notation, where ST(0) is always the top of stack.

Special-Purpose Registers

The FPU has six *special-purpose* registers (see Fig. 12-5):

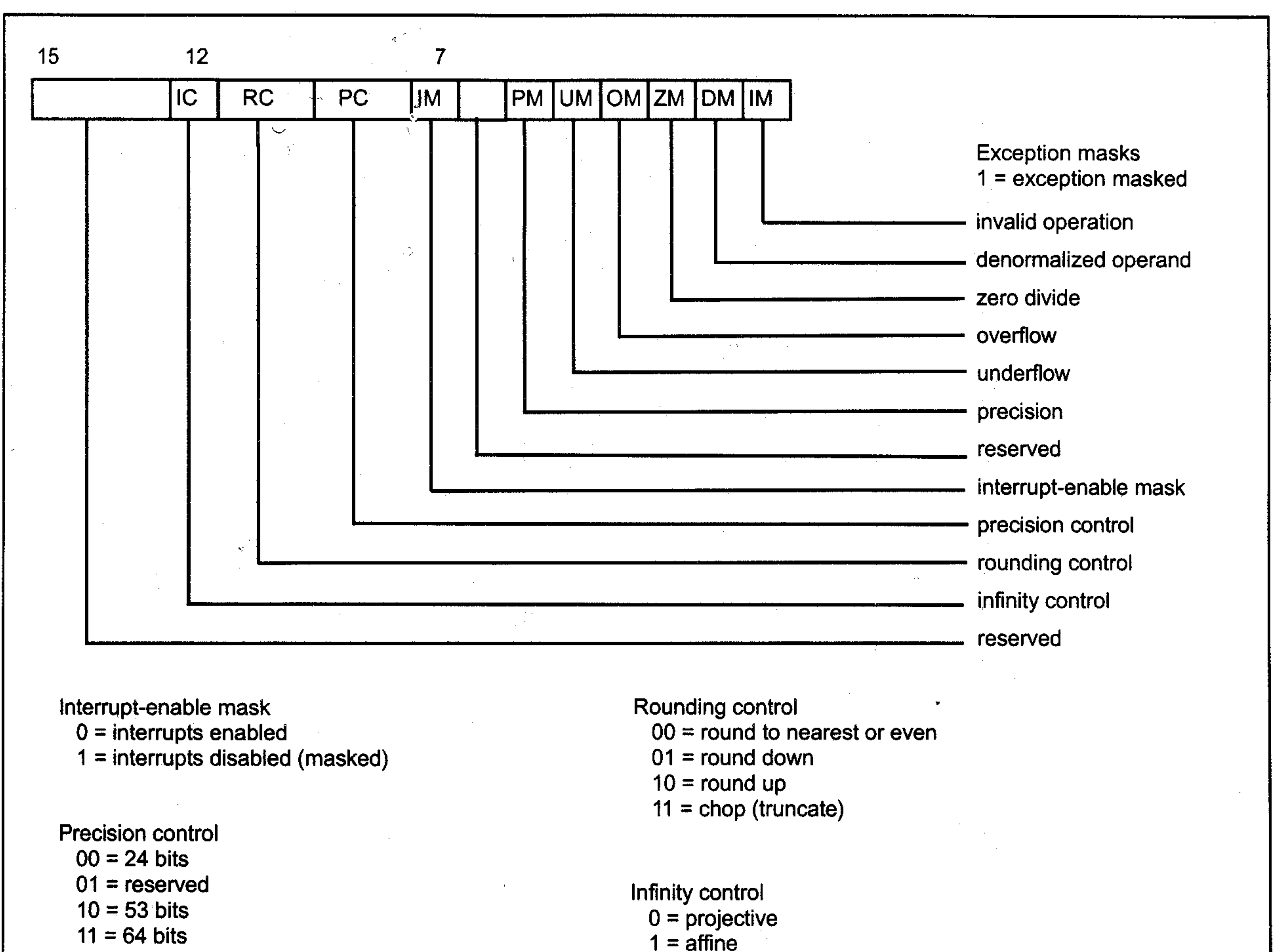
- **Opcode register:** stores the opcode of the last noncontrol instruction executed.
- **Control register:** controls the precision and rounding method used by the FPU when performing calculations. You can also use it to mask out (hide) individual floating-point exceptions.
- **Status register:** contains the top-of-stack pointer, condition codes, and warnings about exceptions.
- **Tag register:** indicates the contents of each register in the FPU data-register stack. It uses two bits per register to indicate whether the register contains a valid number, zero, or a special value (NaN, infinity, denormal, or unsupported format) or is empty.
- **Last instruction pointer register:** stores a pointer to the last noncontrol instruction executed.
- **Last data (operand) pointer register:** stores a pointer to a data operand, if any, used by the last instruction executed.

Figure 12–5 FPU special-purpose registers.



The special-purpose registers are used by operating systems to preserve state information when switching between tasks. We mentioned state preservation in Chapter 2 when explaining how the CPU performs multitasking.

8087 control and status words



8087 control and status words

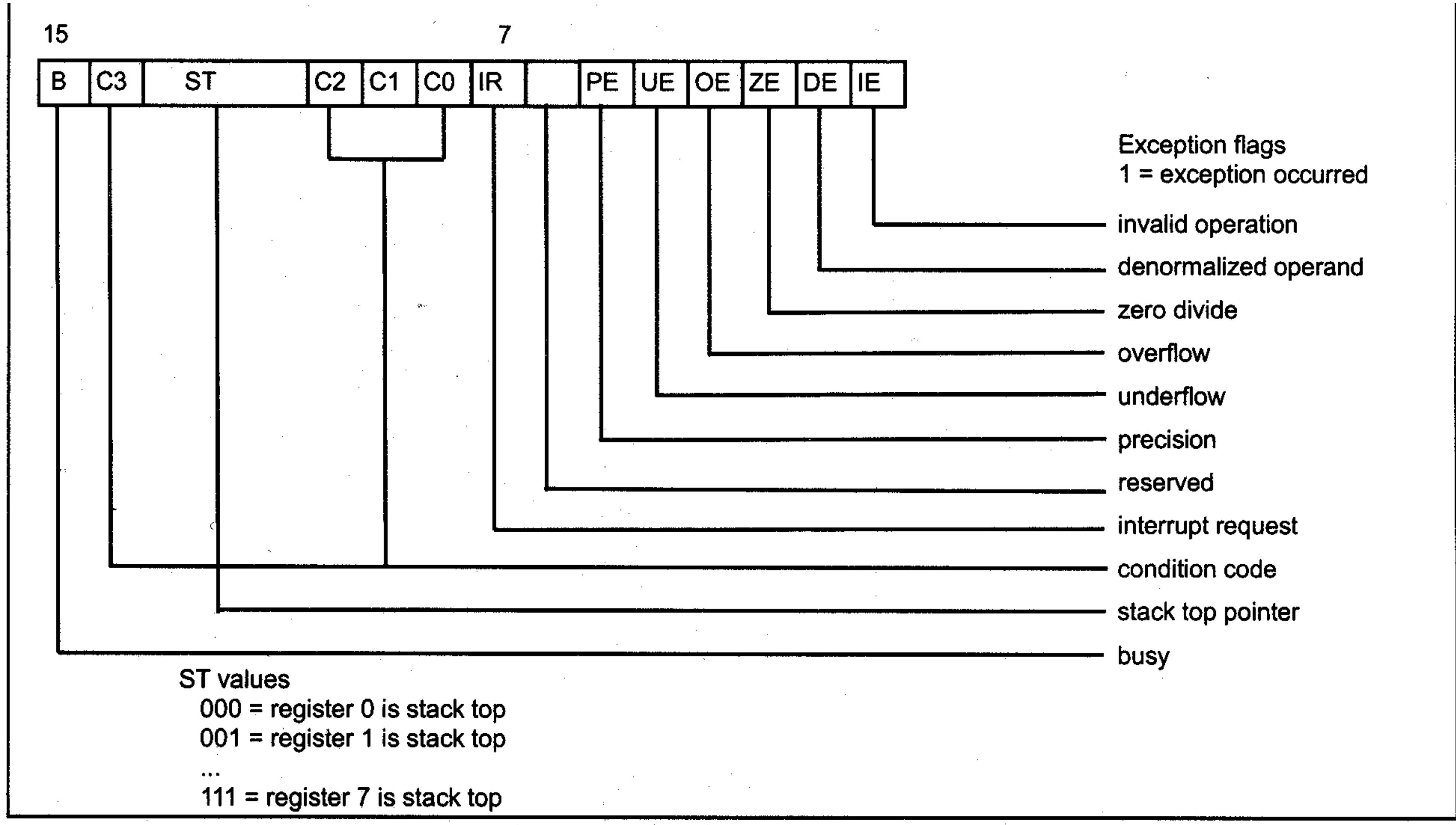


Figure 20-5. 8087 Control and Status Words

Table 4a. Condition Code Interpretation

Instruction Type	C_3	C_2	C_1	C_0	Interpretation
Compare, Test	0	0	X	0	ST > Source or 0 (FTST)
	0	0	X	1	ST < Source or 0 (FTST)
	1	0	X	0	ST = Source or 0 (FTST)
	1	1	X	1	ST is not comparable
Remainder	Q_1	0	Q_0	Q_2	Complete reduction with three low bits of quotient (See Table 4b)
	U	1	U	U	Incomplete Reduction
Examine	0	0	0	0	Valid, positive unnormalized
	0	0	0	1	Invalid, positive, exponent = 0
	0	0	1	0	Valid, negative, unnormalized
	0	0	1	1	Invalid, negative, exponent = 0
	0	1	0	0	Valid, positive, normalized
	0	1	0	1	Infinity, positive
	0	1	1	0	Valid, negative, normalized
	0	1	1	1	Infinity, negative
	1	0	0	0	Zero, positive
	1	0	0	1	Empty
	1	0	1	0	Zero, negative
	1	0	1	1	Empty
	1	1	0	0	Invalid, positive, exponent = 0
	1	1	0	1	Empty
	1	1	1	0	Invalid, negative, exponent = 0
	1	1	1	1	Empty

NOTES:

1. ST = Top of stack
2. X = value is not affected by instruction
3. U = value is undefined following instruction
4. Q_n = Quotient bit n

Table 4b. Condition Code Interpretation after FPREM Instruction As a Function of Divided Value

Dividend Range	Q_2	Q_1	Q_0
Dividend < 2 * Modulus	C_3[1]	C_1[1]	Q_0
Dividend < 4 * Modulus	C_3[1]	Q_1	Q_0
Dividend ≥ 4 * Modulus	Q_2	Q_1	Q_0

NOTE:

1. Previous value of indicated bit, not affected by FPREM instruction execution.

Table 2. 8087 Data Types

Data Formats	Range	Precision	Most Significant Byte (7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0)
Word Integer	10^{4}	16 Bits	I_{15} … I_0 Two's Complement
Short Integer	10^{9}	32 Bits	I_{31} … I_0 Two's Complement
Long Integer	10^{18}	64 Bits	I_{63} … I_0 Two's Complement
Packed BCD	10^{18}	18 Digits	S — $D_{17} D_{16}$ … D_1 D_0
Short Real	$10^{\pm 38}$	24 Bits	S E_7 … E_0 F_1 … F_{23} F_0 Implicit
Long Real	$10^{\pm 308}$	53 Bits	S E_{10} … E_0 F_1 … F_{52} F_0 Implicit
Temporary Real	$10^{\pm 4932}$	64 Bits	S E_{14} … E_0 F_0 … F_{63}

Integer: I
Packed BCD: $(-1)^S(D_{17} \ldots D_0)$
Real: $(-1)^S(2^{E-\text{Bias}})(F_0 \bullet F_1 \ldots)$
bias = 127 for Short Real
1023 for Long Real
16383 for Temp Real

31 23 15 7 0

Sign Bit

Exponent Bits

1

Mantissa Bits

The 24th mantissa bit is implied and is always one.

Figure 14.2 32 Bit Single Precision Floating Point Format

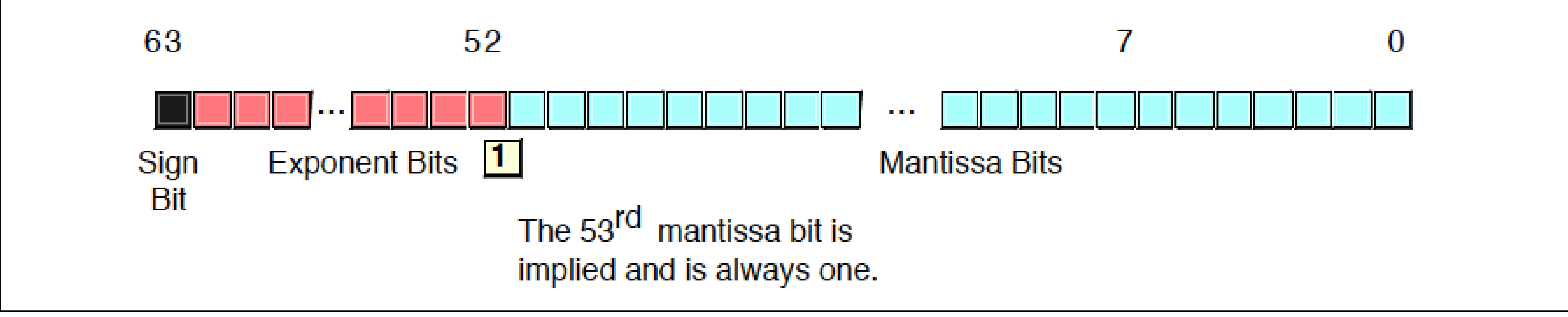


Figure 14.3 64 Bit Double Precision Floating Point Format

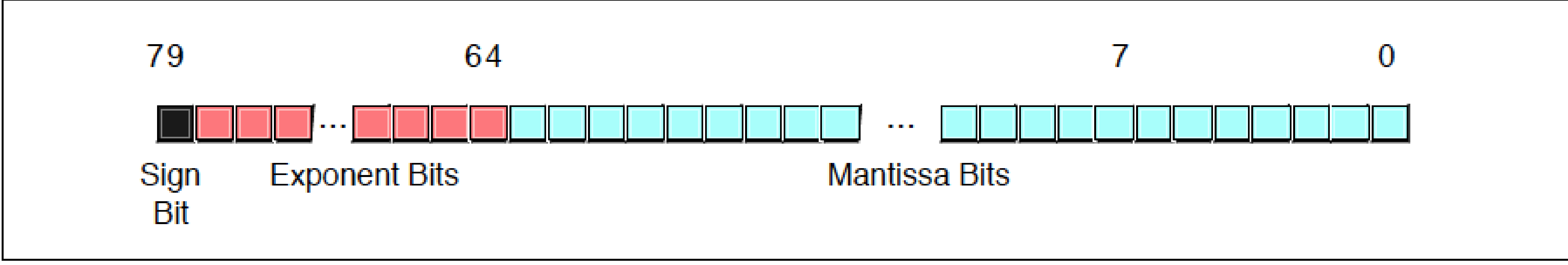


Figure 14.4 80 Bit Extended Precision Floating Point Format

FPU Instruction Set

Data Transfer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Load	FLD	m32fp m64fp m80fp ST(i)	Pushes the source operand onto the FPU stack.	FPU.Top ← (FPU.Top − 1) MOD 8; ST(0) ← SRC
FPU Store	FST	m32fp m64fp ST(i)	Stores ST(0) to destination operand.	DST ← ST(0)
FPU Store and Pop	FSTP	m32fp m64fp m80fp ST(i)	Stores ST(0) to destination operand and pops the register stack	DST ← ST(0); FPU.Pop
FPU Integer Load	FILD	m16int m32int m64int	Converts the signed integer value from memory into extended floating-point and pushes it onto the FPU stack.	FPU.Top ← (FPU.Top − 1) MOD 8; ST(0) ← IntToExtended(SRC)
FPU Integer Store	FIST	m16int m32int	Converts the value in the ST(0) to the signed integer value and stores the result in the destination operand.	DST ← ExtendedToInt(ST(0));
FPU Integer Store and Pop	FISTP	m16int m32int m64int	Converts the value in the ST(0) to the signed integer value, stores the result in the destination operand, and pops register stack.	DST ← ExtendedToInt(ST(0)); FPU.Pop;
FPU BCD Load	FBLD	m80bcd	Converts BCD value from memory to floating-point and pushes it onto the FPU stack.	FPU.Top ← (FPU.Top − 1) MOD 8; ST(0) ← BCDToExtended(SRC)
FPU BCD Store and Pop	FBSTP	m80bcd	Converts the value in the ST(0) to and 18-digit packed BCD, stores the result in memory, and pops the register stack.	DST ← ExtendedToBCD(ST(0)); FPU.Pop;
FPU Exchange	FXCH	ST(i)	Exchanges the contents of the registers ST(0) and ST(i).	TMP ← ST(0); ST(0) ← SRC; SRC ← TMP
			Exchanges the contents of the registers ST(0) and ST(1).	TMP ← ST(0); ST(0) ← ST(1); ST(1) ← TMP
FPU Move if Equal	FCMOVE	ST(0), ST(i)	Moves ST(i) to ST(0) when ZF=1.	IF (condition) THEN ST(0) ← ST(i)
FPU Move if Below	FCMOVB	ST(0), ST(i)	Moves ST(i) to ST(0) when CF=1.	
FPU Move if Below or Equal	FCMOVBE	ST(0), ST(i)	Moves ST(i) to ST(0) when CF=1 or ZF=1.	
FPU Move if Unordered	FCMOVU	ST(0), ST(i)	Moves ST(i) to ST(0) when PF=1.	
FPU Move if Not Equal	FCMOVNE	ST(0), ST(i)	Moves ST(i) to ST(0) when ZF=0.	
FPU Move if Not Below	FCMOVNB	ST(0), ST(i)	Moves ST(i) to ST(0) when CF=0.	
FPU Move if Not Below or Equal	FCMOVNBE	ST(0), ST(i)	Moves ST(i) to ST(0) when CF=0 and ZF=0.	
FPU Move if Not Unordered	FCMOVNU	ST(0), ST(i)	Moves ST(i) to ST(0) when PF=0.	

Floating-Point Basic Arithmetic Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Add	FADD	m32fp	Adds single-precision floating-point value from memory to ST(0)	ST(0) ← ST(0) + SingleToExtended(SRC)
		m64fp	Adds double-precision floating-point value from memory to ST(0)	ST(0) ← ST(0) + DoubleToExtended(SRC)
		ST(0), ST(i)	Adds ST(i) to ST(0) and stores result in ST(0)	ST(0) ← ST(0) + ST(i)

		ST(i), ST(0)	Adds ST(0) to ST(i) and stores result in ST(i)	ST(i) ← ST(i) + ST(0)
FPU Add and Pop	FADDP	ST(i), ST(0)	Adds ST(0) to ST(i), stores result in ST(i), and pops the register stack	ST(i) ← ST(i) + ST(0); FPU.Pop;
	FADDP		Adds ST(0) to ST(1), stores result in ST(1), and pops the register stack	ST(1) ← ST(1) + ST(0); FPU.Pop;
FPU Integer Add	FIADD	m32int	Adds double-word signed integer from memory to ST(0)	ST(0) ← ST(0) + IntToExtended(SRC)
		m64int	Adds quad-word signed integer from memory to ST(0)	ST(0) ← ST(0) + Int64ToExtended(SRC)
FPU Subtract	FSUB	m32fp	Subtracts single-precision floating-point value from memory from ST(0)	ST(0) ← ST(0) – SungleToExtended(SRC)
		m64fp	Subtracts double-precision floating-point value from memory from ST(0)	ST(0) ← ST(0) – DoubleToExtended(SRC)
		ST(0), ST(i)	Subtracts ST(i) from ST(0) and stores result in ST(0)	ST(0) ← ST(0) – ST(i)
		ST(i), ST(0)	Subtracts ST(0) from ST(i) and stores result in ST(i)	ST(i) ← ST(i) – ST(0)
FPU Subtract and Pop	FSUBP	ST(i), ST(0)	Subtracts ST(0) from ST(i), stores result in ST(i), and pops the register stack	ST(i) ← ST(i) – ST(0); FPU.POP;
	FSUBP		Subtracts ST(0) from ST(1), stores result in ST(1), and pops the register stack	ST(1) ← ST(1) – ST(0); FPU.POP;
FPU Integer Subtract	FISUB	m32int	Subtracts double-word signed integer from memory from ST(0)	ST(0) ← ST(0) – IntToExtended(SRC)
	FISUB	m64int	Subtracts quad-word signed integer from memory from ST(0)	ST(0) ← ST(0) – Int64ToExtended(SRC)
FPU Subtract Reverse	FSUBR	m32fp	Subtracts ST(0) from single-precision floating-point value from memory and stores result in ST(0)	ST(0) ← SungleToExtended(SRC) – ST(0)
		m64fp	Subtracts ST(0) from double-precision floating-point value from memory and stores result in ST(0)	ST(0) ← DoubleToExtended(SRC) – ST(0)
		ST(0), ST(i)	Subtracts ST(0) from ST(i) and stores result in ST(0)	ST(0) ← ST(i) – ST(0)
		ST(i), ST(0)	Subtracts ST(i) from ST(0) and stores result in ST(i)	ST(i) ← ST(0) – ST(i)
FPU Subtract Reverse and Pop	FSUBRP	ST(i), ST(0)	Subtracts ST(i) from ST(0), stores result in ST(i), and pops the register stack	ST(i) ← ST(0) – ST(i); FPU.POP;
	FSUBRP		Subtracts ST(1) from ST(0), stores result in ST(1), and pops the register stack	ST(1) ← ST(0) – ST(1); FPU.POP;
FPU Integer Subtract Reverse	FISUBR	m32int	Subtracts ST(0) from double-word signed integer from memory and stores result in ST(0)	ST(0) ← IntToExtended(SRC) – ST(0)
	FISUBR	m64int	Subtracts ST(0) from quad-word signed integer from memory and stores result in ST(0)	ST(0) ← Int64ToExtended(SRC) – ST(0)
FPU Multiply	FMUL	m32fp	Multiplies ST(0) by single-precision floating-point value from memory.	ST(0) ← ST(0) * SingleToExtended(SRC)
		m64fp	Multiplies ST(0) by double-precision floating-point value from memory.	ST(0) ← ST(0) * DoubleToExtended(SRC)
		ST(0), ST(i)	Multiplies ST(0) by ST(i) and stores result in ST(0).	ST(0) ← ST(0) * ST(i)
		ST(i), ST(0)	Multiplies ST(i) by ST(0) and stores result in ST(i).	ST(i) ← ST(i) * ST(0)
FPU Multiply and Pop	FMULP	ST(i), ST(0)	Multiplies ST(i) by ST(0), stores result in ST(i), and pops the register stack.	ST(i) ← ST(i) * ST(0); FPU.Pop;
	FMULP		Multiplies ST(1) by ST(0), stores result in ST(1), and pops the register stack.	ST(1) ← ST(1) * ST(0); FPU.Pop;
FPU Integer Multiply	FIMUL	m32int	Multiplies ST(0) by double-word signed integer from memory.	ST(0) ← ST(0) * IntToExtended(SRC)
		m64int	Multiplies ST(0) by quad-word signed integer from memory.	ST(0) ← ST(0) * Int64ToExtended(SRC)
FPU Divide	FDIV	m32fp	Divides ST(0) by single-precision floating-point value from memory.	ST(0) ← ST(0) / SingleToExtended(SRC)
		m64fp	Divides ST(0) by double-precision floating-point value from memory.	ST(0) ← ST(0) / DoubleToExtended(SRC)
		ST(0), ST(i)	Divides ST(0) by ST(i) and stores result in ST(0).	ST(0) ← ST(0) / ST(i)
		ST(i), ST(0)	Divides ST(i) by ST(0) and stores result in ST(i).	ST(i) ← ST(i) / ST(0)
FPU Divide and Pop	FDIVP	ST(i), ST(0)	Divides ST(i) by ST(0), stores result in ST(i), and pops the register stack.	ST(i) ← ST(i) / ST(0); FPU.Pop;
	FDIVP		Divides ST(1) by ST(0), stores result in ST(1), and pops the register stack.	ST(1) ← ST(1) / ST(0); FPU.Pop;
FPU Integer Divide	FIDIV	m32int	Divides ST(0) by double-word signed integer from memory.	ST(0) ← ST(0) / IntToExtended(SRC)
		m64int	Divides ST(0) by quad-word signed integer from memory.	ST(0) ← ST(0) / Int64ToExtended(SRC)
FPU Divide Reverse	FDIVR	m32fp	Divides single-precision floating-point value from memory by ST(0).	ST(0) ← SingleToExtended(SRC) / ST(0)
		m64fp	Divides double-precision floating-point value from memory by ST(0).	ST(0) ← DoubleToExtended(SRC) / ST(0)
		ST(0), ST(i)	Divides ST(i) by ST(0) and stores result in ST(0).	ST(0) ← ST(i) / ST(0)
		ST(i), ST(0)	Divides ST(0) by ST(i) and stores result in ST(i).	ST(i) ← ST(0) / ST(i)

FPU Divide Reverse and Pop	FDIVRP	ST(i), ST(0)	Divides ST(0) by ST(i), stores result in ST(i), and pops the register stack.	ST(i) ← ST(0) / ST(i); FPU.Pop;
	FDIVRP		Divides ST(0) by ST(1), stores result in ST(1), and pops the register stack.	ST(1) ← ST(0) / ST(1); FPU.Pop;
FPU Integer Divide Reverse	FIDIVR	m32int	Divides double-word signed integer from memory by ST(0).	ST(0) ← IntToExtended(SRC) / ST(0)
		m64int	Divides quad-word signed integer from memory by ST(0).	ST(0) ← Int64ToExtended(SRC) / ST(0)
FPU Partial Remainder	FPREM		Replaces ST(0) with partial remainder obtained from dividing ST(0) by ST(1).	ST(0) ← PartialRemainder (ST(0),ST(1))
FPU Partial Remainder	FPREM1		Replaces ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1). The quotient of ST(0) divided by ST(1) is rounded to an integer.	ST(0) ← IEEERemainder (ST(0),ST(1))
FPU Absolute Value	FABS		Replaces ST(0) with its absolute value.	ST(0) ← Abs(ST(0))
FPU Change Sign	FCHS		Complements the sign bit of ST(0).	ST(0) ← – ST(0)
FPU Round to Integer	FRNDINT		Rounds ST(0) to an integer.	ST(0) ← Round (ST(0))
FPU Scale	FSCALE		Scales ST(0) by ST(1). Truncates the value in the ST(1) to an integral value (toward 0) and adds that value to the exponent of ST(0)	ST(0) ← ST(0) *Base2Power(Truncate(ST(1)))
FPU Square Root	FSQRT		Replaces ST(0) with its square root.	ST(0) ← SquareRoot(ST(0))
FPU Extract Exponent and Significant	FXTRACT		Separates value in ST(0) into exponent and significant, stores exponent in ST(0) and pushes the significant onto the register stack	TMP ← Significant (ST(0)) ST(0) ← Exponent (ST(0)) FPU.Top ← (FPU.Top – 1) MOD 8; ST(0) ← TMP;

Floating-Point Comparison Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Compare	FCOM	m32fp	Compares single-precision floating-point value from memory with ST(0)	CASE OF
		m64fp	Compares double-precision floating-point value from memory with ST(0)	ST(0) > ST(i): FPU.C3, .C2, .C0 ← 000B;
		ST(i)	Compares ST(i) with ST(0)	ST(0) < ST(i): FPU.C3, .C2, .C0 ← 001B;
			Compares ST(1) with ST(0)	ST(0) = ST(i): FPU.C3, .C2, .C0 ← 100B;
FPU Compare and Pop	FCOMP	m32fp	Compares single-precision floating-point value from memory with ST(0) and pops.	Unordeded: FPU.C3, .C2, .C0 ← 111B; *// optionally*
		m64fp	Compares double-precision floating-point value from memory with ST(0) and pops.	END
		ST(i)	Compares ST(i) with ST(0) and pops register stack.	FPU.Pop; *// optionally*
			Compares ST(1) with ST(0) and pops register stack.	FPU.Pop; *// optionally*
FPU Compare and Pop and Pop	FCOMPP		Compares ST(1) with ST(0) and pops register stack twice.	
FPU Unordered Compare	FUCOM	m32fp	Compares single-precision floating-point value from memory with ST(0)	CASE OF
		m64fp	Compares double-precision floating-point value from memory with ST(0)	ST(0) > ST(i): FPU.C3, .C2, .C0 ← 000B;
		ST(i)	Compares ST(i) with ST(0)	ST(0) < ST(i): FPU.C3, .C2, .C0 ← 001B;
			Compares ST(1) with ST(0)	ST(0) = ST(i): FPU.C3, .C2, .C0 ← 100B;
FPU Unordered Compare	FUCOMP	m32fp	Compares single-precision floating-point value from memory with ST(0) and pops.	Unordeded: FPU.C3, .C2, .C0 ← 111B; *// optionally*
		m64fp	Compares double-precision floating-point value from memory with ST(0) and pops.	END
		ST(i)	Compares ST(i) with ST(0) and pops register stack.	FPU.Pop; *// optionally*
			Compares ST(1) with ST(0) and pops register stack.	FPU.Pop; *// optionally*
FPU Unordered Compare, Pop and Pop	FUCOMPP		Compares ST(1) with ST(0) and pops register stack twice.	
FPU Compare and set EFLAGS	FCOMI	ST,ST(i)	Compares ST(0) with ST(i) and sets status flags accordingly.	CASE OF ST(0) > ST(i): ZF, PF, CF ← 000B;
FPU Compare, set EFLAGS and Pop	FCOMIP	ST,ST(i)	Compares ST(0) with ST(i), sets status flags accordingly and pops register stack.	ST(0) < ST(i): ZF, PF, CF ← 001B; ST(0) = ST(i): ZF, PF, CF ← 100B;
FPU Unordered	FUCOMI	ST,ST(i)	Compares ST(0) with ST(i), checks for ordered values, and sets status flags	Unordeded: ZF, PF, CF ← 111B; *// optianally*

Compare and set EFLAGS			accordingly.	END FPU.Pop; // *optionally*
FPU Unordered Compare and set EFLAGS	FUCOMIP	ST,ST(i)	Compares ST(0) with ST(i), checks for ordered values, sets status flags accordingly and pops register stack.	
FPU Integer Compare	FICOM	m16int m32int	Compares ST(0) with integer value in memory and sets condition code flags C0, C2 and C3 in the FPU status word according to the results.	CASE OF ST(0) > ST(i): FPU.C3, .C2, .C0 ← 000B; ST(0) < ST(i): FPU.C3, .C2, .C0 ← 001B; ST(0) = ST(i): FPU.C3, .C2, .C0 ← 100B; Unordeded: FPU.C3, .C2, .C0 ← 111B; END FPU.Pop; // *optionally*
FPU Integer Compare and Pop	FICOMP	m16int m32int	Compares ST(0) with integer value in memory, sets condition code flags C0, C2 and C3 in the FPU status word according to the results and pops register stack.	
FPU Test	FTST		Compares ST(0) with 0.0 and sets condition flags C0, C2 and C3 in the FPU status word according to the results.	CASE OF ST(0) > 0.0: FPU.C3, .C2, .C0 ← 000B; ST(0) < 0.0: FPU.C3, .C2, .C0 ← 001B; ST(0) = 0.0: FPU.C3, .C2, .C0 ← 100B; Unordeded: FPU.C3, .C2, .C0 ← 111B; END
FPU Examine	FXAM		Examines the contents of the ST(0) register and sets the condition flags C0, C2 and C3 in the FPU status word according to the results.	CASE ST(0) OF NaN: FPU.C3, .C2, .C0 ← 001B; normal: FPU.C3, .C2, .C0 ← 010B; infinity: FPU.C3, .C2, .C0 ← 011B; zero: FPU.C3, .C2, .C0 ← 100B; empty: FPU.C3, .C2, .C0 ← 101B; denormal: FPU.C3, .C2, .C0 ← 110B; END FPU.C1 ← Sign(ST(0))

Transcendental Instructions (Trigonometric and Logarithmic Operations)

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Sine	FSIN		Replaces ST(0) with its sine.	ST(0) ← Sine(ST(0))
FPU Cosine	FCOS		Replaces ST(0) with its cosine.	ST(0) ← Cosine(ST(0))
FPU Sine and Cosine	FSINCOS		Replaces ST(0) with its sine and pushes its cosine onto the register stack.	ST(0) ← Sine(ST(0)); TMP ← Cosine(ST(0)); FPU.Top ← (FPU.Top – 1) MOD 8; ST(0) ← TMP
FPU Partial Tangent	FPTAN		Replaces ST(0) with its tangent and pushes 1 onto the register stack.	ST(0) ← Tangent (ST(0)); FPU.Top ← (FPU.Top – 1) MOD 8; ST(0) ← 1;
FPU Partial Arctangent	FPATAN		Replaces ST(1) with arc tangent (ST(1)/ST(0)) and pops the register stack.	ST(1) ← ArcTangent (ST(1)/ST(0)); FPU.Pop
FPU 2^x-1	F2XM1		Replaces ST(0) with ($2^{ST(0)}-1$).	ST(0) ← Base2Power(ST(0))–1)
FPU y*log_2(x)	FYL2X		Replaces ST(1) with ST(1)*log_2ST(0) and pops the register stack.	ST(1) ← ST(1) * Log2(ST(0)); FPU.Pop;
FPU y*log_2(x+1)	FYL2XP1		Replaces ST(1) with ST(1)*log_2(ST(0)+1) and pops the register stack.	ST(1) ← ST(1) * Log2(ST(0)+1); FPU.Pop;

Floating-Point Constant Loading Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Load One	FLD1		Pushes +1.0 onto the FPU stack.	FPU.Top ← FPU.Top – 1 MOD 8; ST(0) ← Constant
FPU Load Zero	FLDZ		Pushes +0.0 onto the FPU stack.	
FPU Load Pi	FLDPI		Pushes π onto the FPU stack.	
FPU Load Base 2 Log of Ten	FLD2T		Pushes $\log_2 10$ onto the FPU stack.	
FPU Load Base 2 Log of E	FLD2E		Pushes $\log_2 e$ onto the FPU stack.	
FPU Load Base 10 Log of 2	FLDLG2		Pushes $\log_{10} 2$ onto the FPU stack.	
FPU Load Base E Log of 2	FLDLN2		Pushes ln2 onto the FPU stack.	

X87 FPU Control Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FPU Initialize	FINIT		Initializes FPU after checking for pending unmasked floating-point exceptions.	FPU.ControlWord ← 037FH; FPU.StatusWord ← 0; FPU.TagWord ← FFFFH; FPU.DataPointer ← 0; FPU.InstructionPointer ← 0; FPU.LastInstructionOpcode ← 0;
	FNINIT		Initializes FPU without checking for pending unmasked floating-point exceptions.	
FPU Clear Exceptions	FCLEX		Clears the floating point exceptions flags after checking for pending unmasked floating-point exceptions.	FPU.StatusWord[0..7] ← 0; FPU.StatusWord[15] ← 0
	FNCLEX		Clears the floating point exceptions flags without checking for pending unmasked floating-point exceptions.	
FPU Decrement Stack-Top Pointer	FDECSTP		Decrements Top field in FPU status word. The effect is to rotate the stack by one position.	FPU.Top ← (FPU.Top – 1) MOD 8;
FPU Increment Stack-Top Pointer	FINCSTP		Increments Top field in FPU status word. The effect is to rotate the stack by one position. Not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.	FPU.Top ← (FPU.Top + 1) MOD 8;
FPU Free floating-point register	FFREE	ST(i)	Sets tag for ST(i) to empty	FPU.Tag(i) ← 11B;
FPU Store Control Word	FSTCW	m2byte	Stores FPU control word to memory after checking for pending unmasked floating point exceptions.	DST ← FPU.ControlWord;
	FNSTCW		Stores FPU control word to memory without checking for pending unmasked floating point exceptions.	
FPU Load Control Word	FLDCW	m2byte	Loads FPU control word from memory.	FPU.ControlWord ← SRC;
FPU Store Status Word	FSTSW	m2byte AX	Stores FPU status word to memory or AX register after checking for pending unmasked floating point exceptions.	DST ← FPU.StatusWord;
	FNSTSW		Stores FPU status word to memory or AX register without checking for pending unmasked floating point exceptions.	
FPU Load Environment	FLDENV	m14/28byte	Loads FPU environment from memory	FPU.ControlWord ← SRC.ControlWord; FPU.StatusWord ← SRC.StatusWord; FPU.TagWord ← SRC.TagWord; FPU.DataPointer ← SRC.DataPointer; FPU.InstructionPointer ← SRC.InstructionPointer; FPU.LastInstructionOpcode ← SRC.LastInstructionOpcode;

FPU Save state	FSAVE	m94/108byte	Stores FPU state to memory after checking for pending unmasked floating-point exceptions. Then reinitializes the FPU.	DST.ControlWord ← FPU.ControlWord; DST.StatusWord ← FPU.StatusWord; DST.TagWord ← FPU.TagWord; DST.DataPointer ← FPU.DataPointer; DST.InstructionPointer ← FPU.InstructionPointer; DST.LastInstructionOpcode ← FPU.LastInstructionOpcode; DST.ST(0) ← FPU.ST(0); DST.ST(1) ← FPU.ST(1); DST.ST(2) ← FPU.ST(2); DST.ST(3) ← FPU.ST(3); DST.ST(4) ← FPU.ST(4); DST.ST(5) ← FPU.ST(5); DST.ST(6) ← FPU.ST(6); DST.ST(7) ← FPU.ST(7); FPU.ControlWord ← 037FH; FPU.StatusWord ← 0; FPU.TagWord ← FFFFH; FPU.DataPointer ← 0; FPU.InstructionPointer ← 0; FPU.LastInstructionOpcode ← 0;
	FNSAVE		Stores FPU state to memory without checking for pending unmasked floating-point exceptions. Then reinitializes the FPU.	
FPU Restore state	FRSTOR	m94/108byte	Loads FPU state from memory	FPU.ControlWord ← SRC.ControlWord; FPU.StatusWord ← SRC.StatusWord; FPU.TagWord ← SRC.TagWord; FPU.DataPointer ← SRC.DataPointer; FPU.InstructionPointer ← SRC.InstructionPointer; FPU.LastInstructionOpcode ← SRC.LastInstructionOpcode; FPU.ST(0) ← SRC.ST(0); FPU.ST(1) ← SRC.ST(1); FPU.ST(2) ← SRC.ST(2); FPU.ST(3) ← SRC.ST(3); FPU.ST(4) ← SRC.ST(4); FPU.ST(5) ← SRC.ST(5); FPU.ST(6) ← SRC.ST(6); FPU.ST(7) ← SRC.ST(7);
Wait for FPU	FWAIT WAIT		Checks pending unmasked floating-point exceptions.	
FPU No Operation	FNOP		No operation is performed	

X87 FPU and SIMD State Management Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
FXSAVE	Save x87 FPU, MMX, SSE and SSE2 State	m512byte	Save x87 FPU and SIMD state in memory.	WORD PTR DST[0] ← FPU.CW; WORD PTR DST[2] ← FPU.SW; BYTE PTR DST[5] ← FPU.TW; WORD PTR DST[6] ← FPU.OP; DWORD PTR DST[8] ← FPU.IP;

				WORD PTR DST[12] ← CS; DWORD PTR DST[16] ← FPU.DP; WORD PTR DST[20] ← DS; DWORD PTR DST[24] ← MXCSR; DWORD PTR DST[28] ← MXCSR_MASK; TWORD PTR DST[32] ← ST0/MM0; TWORD PTR DST[48] ← ST1/MM1; TWORD PTR DST[64] ← ST2/MM2; TWORD PTR DST[80] ← ST3/MM3; TWORD PTR DST[96] ← ST4/MM4; TWORD PTR DST[112] ← ST5/MM5; TWORD PTR DST[128] ← ST6/MM6; TWORD PTR DST[144] ← ST7/MM7; DQWORD PTR DST[160] ← XMM0; DQWORD PTR DST[176] ← XMM1; DQWORD PTR DST[192] ← XMM2; DQWORD PTR DST[208] ← XMM3; DQWORD PTR DST[224] ← XMM4; DQWORD PTR DST[240] ← XMM5; DQWORD PTR DST[256] ← XMM6; DQWORD PTR DST[272] ← XMM7;
FXRSTOR	Restore x87 FPU, MMX, SSE and SSE2 State	m512byte	Restores x87 FPU and SIMD state from memory	FPU.CW ← WORD PTR DST[0]; FPU.SW ← WORD PTR DST[2]; FPU.TW ← BYTE PTR DST[5]; FPU.OP ← WORD PTR DST[6]; FPU.IP ← DWORD PTR DST[8]; CS ← WORD PTR DST[12]; FPU.DP ← DWORD PTR DST[16]; DS ← WORD PTR DST[20]; MXCSR ← DWORD PTR DST[24]; MXCSR_MASK ← DWORD PTR DST[28]; ST0/MM0 ← TWORD PTR DST[32]; ST0/MM0 ← TWORD PTR DST[48]; ST0/MM0 ← TWORD PTR DST[64]; ST0/MM0 ← TWORD PTR DST[80]; ST0/MM0 ← TWORD PTR DST[96]; ST0/MM0 ← TWORD PTR DST[112]; ST0/MM0 ← TWORD PTR DST[128]; ST0/MM0 ← TWORD PTR DST[144]; XMM0 ← DQWORD PTR DST[160]; XMM1 ← DQWORD PTR DST[176]; XMM2 ← DQWORD PTR DST[192]; XMM3 ← DQWORD PTR DST[208]; XMM4 ← DQWORD PTR DST[224]; XMM5 ← DQWORD PTR DST[240]; XMM6 ← DQWORD PTR DST[256]; XMM7 ← DQWORD PTR DST[272];

5.11 Details of comparison predicates

For every supported arithmetic format, it shall be possible to compare one floating-point datum to another in that format (see 5.6.1). Additionally, floating-point data represented in different formats shall be comparable as long as the operands' formats have the same radix.

Four mutually exclusive relations are possible: *less than*, *equal*, *greater than*, and *unordered*. The last case arises when at least one operand is NaN. Every NaN shall compare *unordered* with everything, including itself. Comparisons shall ignore the sign of zero (so $+0 = -0$). Infinite operands of the same sign shall compare *equal*.

Languages define how the result of a comparison shall be delivered, in one of two ways: either as a relation identifying one of the four relations listed above, or as a true-false response to a predicate that names the specific comparison desired.

3.4 Binary interchange format encodings

Each floating-point number has just one encoding in a binary interchange format. To make the encoding unique, in terms of the parameters in 3.3, the value of the significand m is maximized by decreasing e until either $e = emin$ or $m \geq 1$. After this process is done, if $e = emin$ and $0 < m < 1$, the floating-point number is subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value.

Representations of floating-point data in the binary interchange formats are encoded in k bits in the following three fields ordered as shown in Figure 3.1:

a) 1-bit sign S

b) w-bit biased exponent $E = e + bias$

c) $(t = p-1)$-bit trailing significand field digit string $T = d_1 d_2 \ldots d_{p-1}$; the leading bit of the significand, d_0, is implicitly encoded in the biased exponent E.

1 bit	MSB w bits LSB	MSB $t = p-1$ bits LSB
S (sign)	E (biased exponent)	T (trailing significand field)
	$E_0 \ldots E_{w-1}$	$d_1 \ldots d_{p-1}$

Figure 3.1—Binary interchange floating-point format

The values of k, p, t, w, and $bias$ for binary interchange formats are listed in Table 3.5 (see 3.6).

The range of the encoding's biased exponent E shall include:

— every integer between 1 and $2^w - 2$, inclusive, to encode normal numbers

— the reserved value 0 to encode ±0 and subnormal numbers

— the reserved value $2^w - 1$ to encode ±∞ and NaNs.

The representation r of the floating-point datum, and value v of the floating-point datum represented, are inferred from the constituent fields as follows:

a) If $E = 2^w - 1$ and $T \neq 0$, then r is qNaN or sNaN and v is NaN regardless of S (see 6.2.1).

b) If $E = 2^w - 1$ and $T = 0$, then r and $v = (-1)^S \times (+\infty)$.

c) If $1 \leq E \leq 2^w - 2$, then r is $(S, (E - bias), (1 + 2^{1-p} \times T))$;
the value of the corresponding floating-point number is $v = (-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$;
thus normal numbers have an implicit leading significand bit of 1.

d) If $E = 0$ and $T \neq 0$, then r is $(S, emin, (0 + 2^{1-p} \times T))$;
the value of the corresponding floating-point number is $v = (-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$;
thus subnormal numbers have an implicit leading significand bit of 0.

e) If $E = 0$ and $T = 0$, then r is $(S, emin, 0)$ and $v = (-1)^S \times (+0)$ (signed zero, see 6.3).

3.6 Interchange format parameters

Interchange formats support the exchange of floating-point data between implementations. In each radix, the precision and range of an interchange format is defined by its size; interchange of a floating-point datum of a given size is therefore always exact with no possibility of overflow or underflow.

This standard defines binary interchange formats of widths 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits. Decimal interchange formats are defined for any multiple of 32 bits of at least 32 bits.

The parameters p and $emax$ for every interchange format width are shown in Table 3.5 for binary interchange formats and in Table 3.6 for decimal interchange formats. The encodings for the interchange formats are as described in 3.4 and 3.5.2; the encoding parameters for each interchange format width are also shown in Tables 3.5 and 3.6.

Table 3.5—Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \text{round}(4 \times \log2(k)) + 13$
$emax$, maximum exponent e	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
$bias$, $E - e$	15	127	1023	16383	$emax$
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

The function round() in Table 3.5 rounds to the nearest integer.

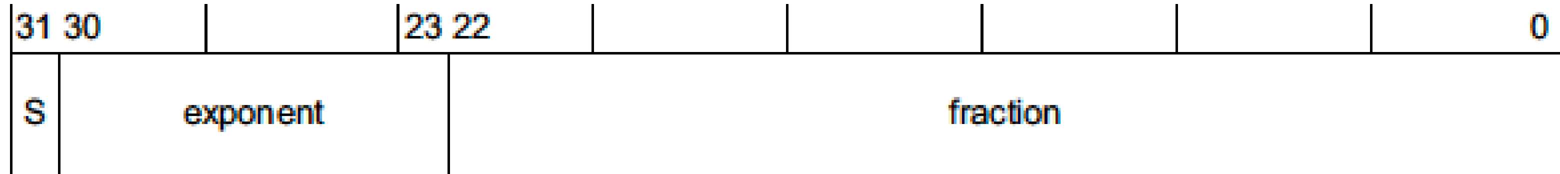
For example, binary256 would have $p = 237$ and $emax = 262143$.

Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes ARM-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:



The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a *normalized number* and is equal to:

$(-1)^S \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a *denormalized number*, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 When S==0.

–0 When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see *Flush-to-zero* on page A1-53.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

In the Floating-point Extension, the two types of NaN are distinguished on the basis of their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN* see *NaN handling and the Default NaN* on page A2-69.

Note

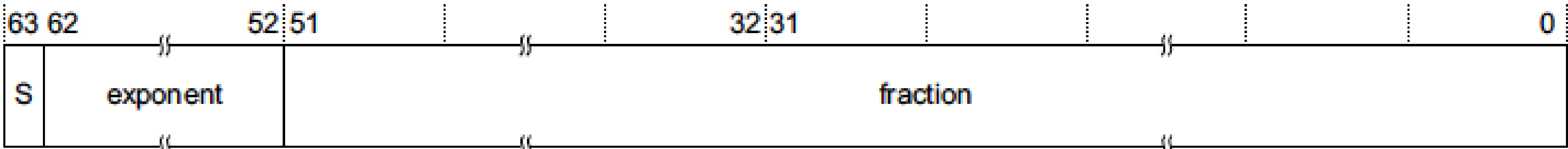
NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

Table 12-6 Specific Single-Precision Encodings.

Value	Sign, Exponent, Significand		
Positive zero	0	00000000	00000000000000000000000
Negative zero	1	00000000	00000000000000000000000
Positive infinity	0	11111111	00000000000000000000000
Negative infinity	1	11111111	00000000000000000000000
QNaN	x	11111111	1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x	11111111	0xxxxxxxxxxxxxxxxxxxxxx[a]

[a] SNaN significand field begins with 0, but at least one of the remaining bits must be 1.

A double-precision value is a 64-bit doubleword, with the format:



Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < 0x7FF

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-1023)} \times (1.fraction)$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0

–0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.fraction)$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see *Flush-to-zero* on page A1-53.

exponent == 0x7FF

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see *NaN handling and the Default NaN* on page A1-54.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

6.1 Infinity arithmetic

The behavior of infinity in floating-point arithmetic is derived from the limiting cases of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is: $-\infty < \{\text{every finite number}\} < +\infty$.

Operations on infinite operands are usually exact and therefore signal no exceptions, including, among others,

— addition(∞, x), addition(x, ∞), subtraction(∞, x), or subtraction(x, ∞), for finite x

— multiplication(∞, x) or multiplication(x, ∞) for finite or infinite $x \neq 0$

— division(∞, x) or division(x, ∞) for finite x

— squareRoot($+\infty$)

— remainder(x, ∞) for finite normal x

— conversion of an infinity into the same infinity in another format.

The exceptions that do pertain to infinities are signaled only when

— ∞ is an invalid operand (see 7.2)

— ∞ is created from finite operands by overflow (see 7.4) or division by zero (see 7.3)

— remainder(subnormal, ∞) signals underflow.

The VFP11 coprocessor provides full IEEE 754 standard compatibility through a combination of hardware and software. There are rare cases that require significant additional compute time to resolve correctly according to the requirements of the IEEE 754 standard. For instance, the VFP11 coprocessor does not process subnormal input values directly. To provide correct handling of subnormal inputs according to the IEEE 754 standard, a trap is made to support code to process the operation. Using the support code for processing this operation can require hundreds of cycles. In some applications this is unavoidable, because compliance with the IEEE 754 standard is essential to proper operation of the program. In many other applications, strict compliance to the IEEE 754 standard is unnecessary, while determinable runtime, low interrupt latency, and low power are of more importance. To accommodate a variety of applications, the VFP11 coprocessor provides four modes of operation:

- *Full-compliance mode*
- *Flush-to-zero mode* on page 1-14
- *Default NaN mode* on page 1-14
- *RunFast mode* on page 1-15.

Flush-to-zero mode

Setting the FZ bit, FPSCR[24], enables flush-to-zero mode and increases performance on very small inputs and results. In flush-to-zero mode, the VFP11 coprocessor treats all subnormal input operands of arithmetic CDP operations as positive zeros in the operation. Exceptions that result from a zero operand are signaled appropriately. FABS, FNEG, FCPY, and FCMP are not considered arithmetic CDP operations and are not affected by flush-to-zero mode. A result that is *tiny*, as described in the IEEE 754 standard, for the destination precision is smaller in magnitude than the minimum normal value *before rounding* and is replaced with a positive zero. The IDC flag, FPSCR[7], indicates when an input flush occurs. The UFC flag, FPSCR[3], indicates when a result flush occurs.

Default NaN mode

Setting the DN bit, FPSCR[25], enables default NaN mode. In default NaN mode, the result of any operation that involves an input NaN or generated a NaN result returns the default NaN. Propagation of the fraction bits is maintained only by FABS, FNEG, and FCPY operations, all other CDP operations ignore any information in the fraction bits of an input NaN. See *NaN handling* on page 3-5 for a description of default NaNs.

RunFast mode

RunFast mode is the combination of the following conditions:

- the VFP11 coprocessor is in flush-to-zero mode
- the VFP11 coprocessor is in default NaN mode
- all exception enable bits are cleared.

In RunFast mode the VFP11 coprocessor:

- processes subnormal input operands as positive zeros
- processes results that are *tiny* before rounding, that is, between the positive and negative minimum normal values for the destination precision, as positive zeros
- processes input NaNs as default NaNs
- returns the default result specified by the IEEE 754 standard for overflow, division by zero, invalid operation, or inexact operation conditions fully in hardware and without additional latency
- processes all operations in hardware without trapping to support code.

RunFast mode enables the programmer to write code for the VFP11 coprocessor that runs in a determinable time without support code assistance, regardless of the characteristics of the input data. In RunFast mode, no user exception traps are available. However, the exception flags in the FPSCR register are compliant with the IEEE 754 standard for Inexact, Overflow, Invalid Operation, and Division by Zero exceptions. The underflow flag is modified for flush-to-zero mode. Each of these flags is set by an exceptional condition and can by cleared only by a write to the FPSCR register.

Short vector instructions

The VFPv2 architecture supports execution of *short vector* instructions of up to eight operations on single-precision data and up to four operations on double-precision data. Short vectors are most useful in graphics and signal-processing applications. They reduce code size, increase speed of execution by supporting parallel operations and multiple transfers, and simplify algorithms with high data throughput.

Short vector operations issue the individual operations specified in the instruction in a serial fashion. To eliminate data hazards, short vector operations begin execution only after all source registers are available, and all destination registers are not targets of other operations.

About the register file

The VFP11 register file contains thirty-two 32-bit registers organized in four banks. Each register can store either a single-precision floating-point number or an integer.

Any consecutive pair of registers, $[R_{even+1}]$:$[R_{even}]$, can store a double-precision floating-point number. Because a load and store operation does not modify the data, the VFP11 registers can also be used as secondary data storage by another application that does not use floating-point values.

The register file can be configured as four circular buffers for use by short vector instructions in applications requiring high data throughput, such as filtering and graphics transforms. For short vector instructions, register addressing is circular within each bank. Load and store operations do not circulate, allowing for multiple banks, up to the entire register file, to be loaded or stored in a single instruction. Short vector operations obey certain rules specifying under what conditions the registers in the argument list specify circular buffers or single-scalar registers. The LEN and STRIDE fields in the FPSCR register specify the number of operations performed by short vector instructions and the increment scheme within the circular register banks. Further information and examples are in Section C5 of the *ARM Architecture Reference Manual*.

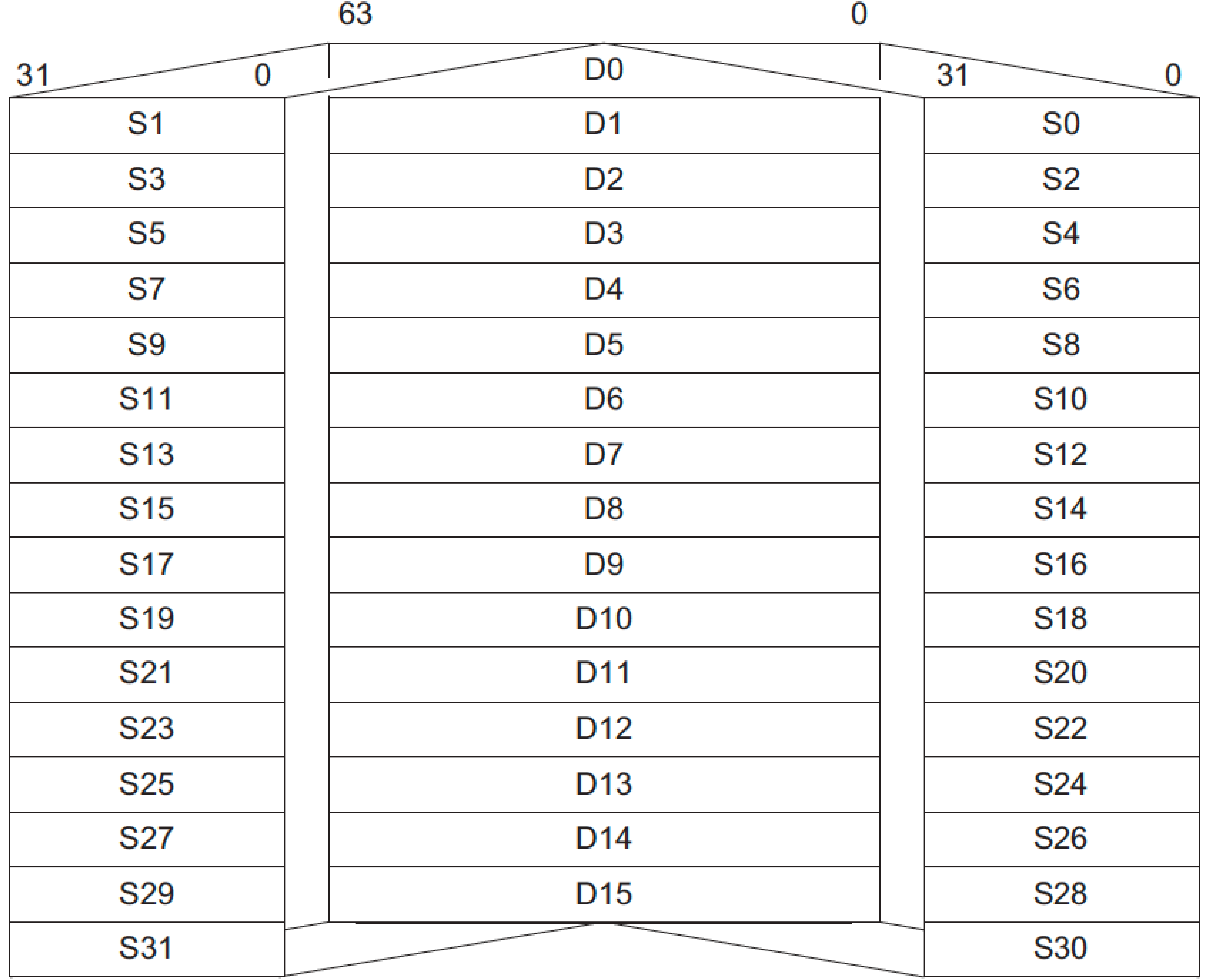


Figure 2-3 Register file access

S1	S0		D0
S3	S2		D1
S5	S4		D2
S7	S6		D3
S9	S8		D4
S11	S10		D5
S13	S12		D6
S15	S14		D7
S17	S16	overlapped with	D8
S19	S18		D9
S21	S20		D10
S23	S22		D11
S25	S24		D12
S27	S26		D13
S29	S28		D14
S31	S30		D15

Figure C2-1 VFP general-purpose registers

About register banks

As Figure 2-4 shows, the register file is divided into four banks with eight registers in each bank for single-precision instructions and four registers per bank for double-precision instructions. CDP instructions access the banks in a circular manner. Load and store multiple instructions do not access the registers in a circular manner but treat the register file as a linearly ordered structure.

See *ARM Architecture Reference Manual, Part C* for more information on VFP addressing modes.

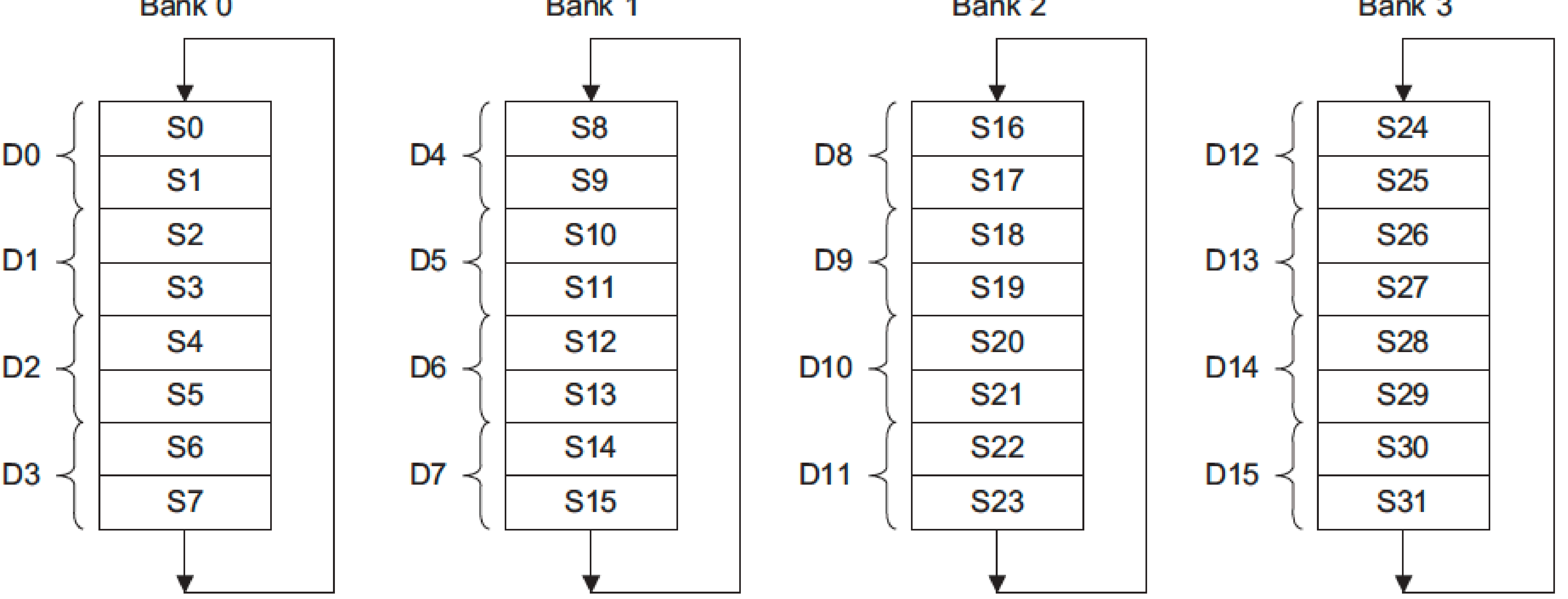


Figure 2-4 Register banks

A short vector CDP operation that has a source or destination vector crossing a bank boundary wraps around and accesses the first register in the bank.

System registers

A VFP implementation contains three or more special-purpose *system registers*:

- The *Floating-point System ID* register (FPSID) is a read-only register whose value indicates which VFP implementation is being used. See *FPSID* on page C2-22 for details.
- The *Floating-point Status and Control* register (FPSCR) is a read/write register which provides all user-level status and control of the floating-point system. See *FPSCR* on page C2-23 for details of the FPSCR.
- The *Floating-point Exception* register (FPEXC) is a read/write register, two bits of which provide system-level status and control. The remaining bits of this register can be used to communicate exception information between the hardware and software components of the implementation, in a SUB-ARCHITECTURE DEFINED manner. See *FPEXC* on page C2-27 for details of the FPEXC.
- Individual VFP implementations can define and use further system registers for the purpose of communicating between the hardware and software components of the implementation, and for other IMPLEMENTATION DEFINED control of the VFP implementation. All such registers are SUB-ARCHITECTURE DEFINED. They must not be used outside the implementation itself, except as described in sub-architecture-specific documentation.

Key to Tables			
{C}	See Table **Condition Field**	<fpconst>	$+/- m * 2^{-n}$ where m and n are integers, $16 <= m <= 31$, $0 <= n <= 7$
<P>	F32 (single precision) or F64 (double precision).	Fd, Fn, Fm	Sd, Sn, Sm (single precision), or Dd, Dn, Dm (double precision).
S, D, H	Single, double, or half-precision (F16).	{E}	E : raise exception on any NaN. Without E : raise exception only on signaling NaNs.
F	Single or double-precision floating point.	{R}	Use FPSCR rounding mode. Otherwise, round towards zero.
SI, UI	Signed or unsigned integer.	<VFPregs>	A comma separated list of *consecutive* VFP registers, enclosed in braces ({ and }).
<VFPsysreg>	FPSCR or FPSID.	<fbits>	Number of fraction bits in fixed-point number, 0-16 or 1-32.
§	2: VFPv2 and above. 3: VFPv3 and above. 3H: VFPv3 and above with half-precision extension.	<type>	S16, S32, U16, or U32, for Signed or Unsigned, 16-bit or 32-bit.

Operation		§	Assembler	Exceptions	Action	Notes
Vector arithmetic	Multiply		VMUL{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fn * Fm	
	and negate		VNMUL{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – (Fn * Fm)	
	and accumulate		VMLA{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd + (Fn * Fm)	
	negate and accumulate		VMLS{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := Fd – (Fn * Fm)	
	and subtract		VNMLS{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – Fd + (Fn * Fm)	
	negate and subtract		VNMLA{C}.<P> Fd, Fn, Fm	IO, OF, UF, IX	Fd := – Fd – (Fn * Fm)	
	Add		VADD{C}.<P> Fd, Fn, Fm	IO, OF, IX	Fd := Fn + Fm	
	Subtract		VSUB{C}.<P> Fd, Fn, Fm	IO, OF, IX	Fd := Fn – Fm	
	Divide		VDIV{C}.<P> Fd, Fn, Fm	IO, DZ, OF, UF, IX	Fd := Fn / Fm	
	Absolute		VABS{C}.<P> Fd, Fm		Fd := abs(Fm)	
	Negative		VNEG{C}.<P> Fd, Fm		Fd := – Fm	
	Square root		VSQRT{C}.<P> Fd, Fm	IO, IX	Fd := sqrt(Fm)	
Scalar compare	Two values		VCMP{E}{C}.<P> Fd, Fm	IO	Set FPSCR flags on Fd – Fm	Use VMRS APSR_nzcv, FPSCR to transfer flags.
	Value with zero		VCMP{E}{C}.<P> Fd, #0.0	IO	Set FPSCR flags on Fd – 0	
Scalar convert	Single to double		VCVT{C}.F64.F32 Dd, Sm	IO	Dd := convertStoD(Sm)	
	Double to single		VCVT{C}.F32.F64 Sd, Dm	IO, OF, UF, IX	Sd := convertDtoS(Dm)	
	Unsigned integer to float		VCVT{C}.<P>.U32 Fd, Sm	IX	Fd := convertUItoF(Sm)	
	Signed integer to float		VCVT{C}.<P>.S32 Fd, Sm	IX	Fd := convertSItoF(Sm)	
	Float to unsigned integer		VCVT{R}{C}.U32.<P> Sd, Fm	IO, IX	Sd := convertFtoUI(Fm)	
	Float to signed integer		VCVT{R}{C}.S32.<P> Sd, Fm	IO, IX	Sd := convertFtoSI(Fm)	
	Fixed-point to float	3	VCVT{C}.<P>.<type> Fd, Fd, #<fbits>	IO, IX	Fd := convert<type>toF(Fd)	Source is in bottom 16 or 32 bits of Fd.
	Float to fixed-point	3	VCVT{C}.<type>.<P> Fd, Fd, #<fbits>	IO, IX	Fd := convertFto<type>(Fd)	Destination is bottom 16 or 32 bits of Fd.
	Single to half-precision	3H	VCVTT{C}.F16.F32 Sd,Sm	ID, IO, OF, UF, IX	Sd:=convertStoH(Sm)	Destination is top 16 bits of Sd
	Single to half-precision	3H	VCVTB{C}.F16.F32 Sd,Sm	ID, IO, OF, UF, IX	Sd:=convertStoH(Sm)	Destination is bottom 16 bits of Sd
	Half to single-precision	3H	VCVTT{C}.F32.F16 Sd,Sm	ID, IO, OF, UF, IX	Sd:=convertHtoS(Sm)	Source is top 16 bits of Sm
	Half to single-precision	3H	VCVTB{C}.F32.F16 Sd,Sm	ID, IO, OF, UF, IX	Sd:=convertHtoS(Sm)	Source is bottom 16 bits of Sm
Insert constant	Insert constant in register	3	VMOV{C}.<P> Fd, #<fpconst>		Fd := <fpconst>	
Transfer registers	Copy VFP register		VMOV{C}.<P> Fd, Fm		Fd := Fm	
	ARM® to single		VMOV{C} Sn, Rd		Sn := Rd	
	Single to ARM		VMOV{C} Rd, Sn		Rd := Sn	
	Two ARM to two singles	2	VMOV{C} Sn, Sm, Rd, Rn		Sn := Rd, Sm := Rn	Sm must be S(n+1)
	Two singles to two ARM	2	VMOV{C} Rd, Rn, Sn, Sm		Rd := Sn, Rn := Sm	Sm must be S(n+1)
	Two ARM to double	2	VMOV{C} Dm, Rd, Rn		Dm[31:0] := Rd, Dm[63:32] := Rn	
	Double to two ARM	2	VMOV{C} Rd, Rn, Dm		Rd := Dm[31:0], Rn := Dm[63:32]	
	ARM to lower half of double		VMOV{C} Dn[0], Rd		Dn[31:0] := Rd	
	Lower half of double to ARM		VMOV{C} Rd, Dn[0]		Rd := Dn[31:0]	

Operation		§	Assembler	Exceptions	Action	Notes
Transfer registers (continued)	ARM to upper half of double		`VMOV{C} Dn[1], Rd`		Dn[63:32] := Rd	
	Upper half of double to ARM		`VMOV{C} Rd, Dn[1]`		Rd := Dn[63:32]	
	ARM to VFP system register		`VMSR{C} <VFPsysreg>, Rd`		VFPsysreg := Rd	
	VFP system register to ARM		`VMRS{C} Rd, <VFPsysreg>`		Rd := VFPsysreg	
	FPSCR flags to APSR		`VMRS{C} APSR_nzcv, FPSCR`		APSR flags := FPSCR flags	

Operation		§	Assembler	Synonyms	Action
Save VFP registers	Single		`VSTR{C} Fd, [Rn{, #<immed>}]`		[address] := Fd. Immediate range 0-1020, multiple of 4.
	Single, PC-relative		`VSTR{C} Fd, <label>`		
	Multiple, unindexed / increment after		`VSTM{C} Rn{!}, <VFPregs>`	`VSTMIA, VSTMEA`	Saves list of VFP registers, starting at address in Rn.
	decrement before		`VSTMDB{C} Rn!, <VFPregs>`	`VSTMFD` (full descending)	
	Push onto stack		`VPUSH{C} <VFPregs>`	`VSTMFD SP!`	
Load VFP registers	Single		`VLDR{C} Fd, [Rn{, #<immed>}]`		Fd := [address]. Immediate range 0-1020, multiple of 4.
	Single, PC-relative		`VLDR{C} Fd, <label>`		
	Multiple, unindexed / increment after		`VLDM{C} Rn{!}, <VFPregs>`	`VLDMIA, VLDMFD`	Loads list of VFP registers, starting at address in Rn.
	decrement before		`VLDMDB{C} Rn!, <VFPregs>`	`VLDMEA` (empty ascending)	
	Pop from stack		`VPOP{C} <VFPregs>`	`VLDM SP!`	

FPSCR format								Rounding		(Stride – 1)*3			Vector length – 1						Exception trap enable bits								Cumulative exception bits				
31	30	29	28	27	26	25	24	23	22	21	20		18	17	16	15			12	11	10	9	8	7			4	3	2	1	0
N	Z	C	V	QC	AHP	DB	FZ	RMODE		STRIDE			LEN			IDE			IXE	UFE	OFE	DZE	IOE	IDC			IXC	UFC	OFC	DZC	IOC

FZ: 1 = flush to zero mode. Rounding: 0 = round to nearest, 1 = towards +∞, 2 = towards –∞, 3 = towards zero. (Vector length * Stride) must not exceed 4 for double precision operands. (Deprecated)

Condition Field

Mnemonic	Description (VFP)	Description (ARM or Thumb®)	Mnemonic	Description (VFP)	Description (ARM or Thumb)
`EQ`	Equal	Equal	`HI`	Greater than, or unordered	Unsigned higher
`NE`	Not equal, or unordered	Not equal	`LS`	Less than or equal	Unsigned lower or same
`CS / HS`	Greater than or equal, or unordered	Carry Set / Unsigned higher or same	`GE`	Greater than or equal	Signed greater than or equal
`CC / LO`	Less than	Carry Clear / Unsigned lower	`LT`	Less than, or unordered	Signed less than
`MI`	Less than	Negative	`GT`	Greater than	Signed greater than
`PL`	Greater than or equal, or unordered	Positive or zero	`LE`	Less than or equal, or unordered	Signed less than or equal
`VS`	Unordered (at least one NaN operand)	Overflow	`AL`	Always (normally omitted)	Always (normally omitted)
`VC`	Not unordered	No overflow			

Exceptions	
`ID`	Input Denormal
`IO`	Invalid operation
`OF`	Overflow
`UF`	Underflow
`IX`	Inexact result
`DZ`	Division by zero

Floating-point exceptions

The IEEE 754 standard specifies five classes of floating-point exception:

Invalid Operation exception

This exception occurs in various cases where neither a numeric value nor an infinity is a sensible result of a floating-point operation, and also when an operand of a floating-point operation is a signaling NaN. For more details of Invalid Operation exceptions, see *NaNs* on page C2-5.

Division by Zero exception

This exception occurs when a normalized or denormalized number is divided by a zero.

Overflow exception

This exception occurs when the result of an arithmetic operation on two floating-point values is too big in magnitude for it to be represented in the destination format without an unusually large rounding error for the rounding mode in use.

More precisely, the *ideal rounded result* of a floating-point operation is defined to be the result that its rounding mode would produce if the destination format had no limits on the unbiased exponent range. If the ideal rounded result has an unbiased exponent too big for the destination format (that is, >127 for single-precision or >1023 for double-precision), it differs from the actual rounded result, and an Overflow exception occurs.

Underflow exception

The conditions for this exception to occur depend on whether *Flush-to-zero* mode is being used and on the value of the *Underflow exception enable* (UFE) bit (bit[11] of the FPSCR).

If *Flush-to-zero* mode is not being used and the UFE bit is 0, underflow occurs if the result before rounding of a floating-point operation satisfies 0 < abs(result) < MinNorm, where MinNorm = 2^{-126} for single precision or 2^{-1022} for double precision, and the final result is inexact (that is, has a different value to the result before rounding).

If *Flush-to-zero* mode is being used or the UFE bit is 1, underflow occurs if the result before rounding of a floating-point operation satisfies 0 < abs(result) < MinNorm, regardless of whether the final result is inexact or not.

An underflow exception that occurs in *Flush-to-zero* mode is always treated as untrapped, regardless of the actual value of the UFE bit. For details of this and other aspects of *Flush-to-zero* mode, see *Flush-to-zero mode* on page C2-14.

Note

The IEEE 754 standard leaves two choices open in its definition of the Underflow exception. In the terminology of the standard, the above description means that the VFP architecture requires these choices to be:

- the *before rounding* form of *tininess*
- the *inexact result* form of *loss of accuracy*.

Tininess is detected before rounding in *Flush-to-zero* mode.

Inexact exception

The result of an arithmetic operation on two floating-point values can have more significant bits than the destination register can contain. When this happens, the result is rounded to a value that the destination register can hold and is said to be *inexact*.

The inexact exception occurs whenever:

- a result is not equal to the computed result before rounding
- an untrapped Overflow exception occurs
- an untrapped Underflow exception occurs, while not in *Flush-to-zero* mode.

Note

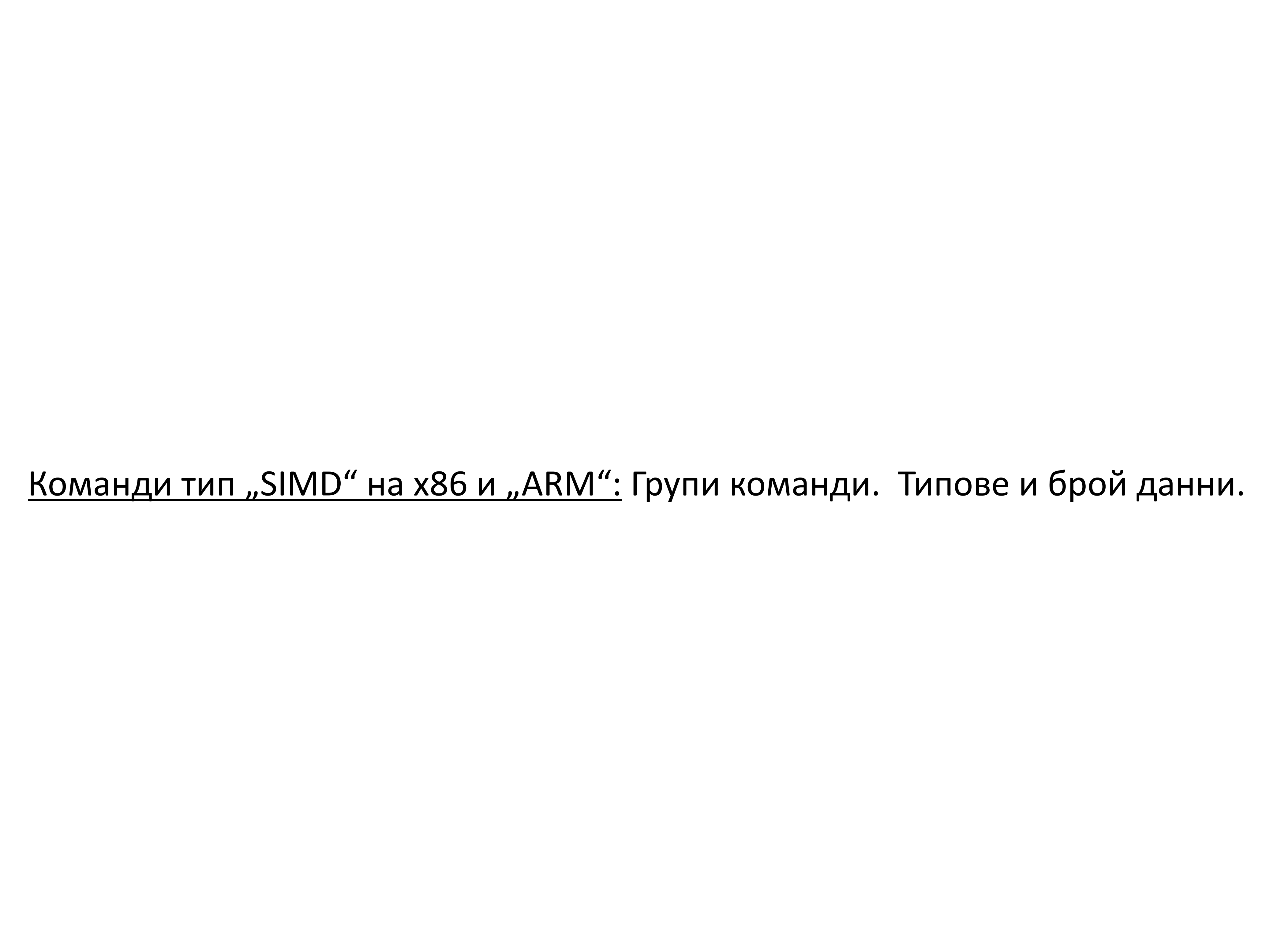
The Inexact exception occurs frequently in normal floating-point calculations and does not indicate a significant numerical error except in some specialized applications. Enabling the Inexact exception can significantly reduce the performance of the coprocessor.

The VFP architecture specifies one additional exception:

Input Denormal exception

This exception occurs only in *Flush-to-zero* mode, when an input to an arithmetic operation is a denormalized number and treated as zero.

This exception does not occur for non-arithmetic operations, `FABS`, `FCPY`, `FNEG`, as described in *Copy, negation and absolute value instructions* on page C3-13.


Команди тип „SIMD“ на x86 и „ARM“: Групи команди. Типове и брой данни.

AbbreviationFinder

MMX
Matrix Math Extensions

SIMD
Single Instruction Multiple Data

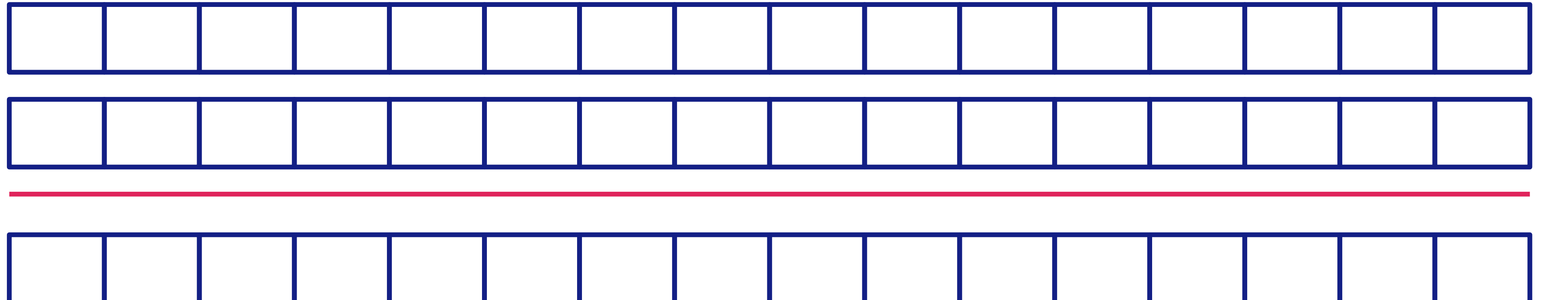
SSE
Streaming SIMD Extensions

www.abbreviationfinder.org

SIMD: using subword parallelism

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE, 1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (AVX, 2010, 2015)
    - Four 64-bit integer/fp ops … -> today upto: 512-bit
  - Operands must be consecutive and aligned memory locations

E.g. 16 bytes in parallel:



x86 architecture SIMD support

- ISA SIMD support
  - MMX, 3DNow!, SSE, SSE2, SSSE3, SSE4, AVX
  - Streaming SIMD Extensions (SSE)
  - SIMD instruction set extension to the x86 architecture

- Micro architecture support
  - Many functional units (for vector operations)
  - Multiple 128-bit vector registers, XMM0, …, XMM15

Performance difference

- C = A * B
  - `for(i=0; i<n; i++) c[i]=a[i]*b[i];`

Scalar loop:

```
L1:
   movss   xmm0, [rdx+r13*4]
   mulss   xmm0, [r8+r13*4]
   movss   [rcx+r13*4], xmm0
   add     r13, 1
   cmp     r13, r9
   jl      L1
```

6 instructions / 1 element

Vector loop:

```
L1:
   movups  xmm1, [rdx+r9*4]
   movups  xmm0, [r8+r9*4]
   mulps   xmm1, xmm0
   movaps  [rcx+r9*4], xmm1
   add     r9, 4
   cmp     r9, rax
   jl      L1
```

7 instructions / 4 elements 1.75 instr./element

MMX Registers ↔ FPU registers

63 0

MMX(7)
MMX(6)
MMX(5)
MMX(4)
MMX(3)
MMX(2)
MMX(1)
MMX(0)

8 * byte
4 * word
2 * dword
1 * qword

SSE/SSE2 Registers

127 0

XMM (7)
XMM(6)
XMM(5)
XMM(4)
XMM(3)
XMM(2)
XMM(1)
XMM(0)

16 *byte
8 * word
4 * dword
4 * single FP
2 * qword
2 * double FP
1 *dbl qword

31 0

MXCSR

MMX/SSE Instruction set

MMX/SSE Data transfer instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Move Dword	MOVD	mm, r/m32	Move double word from r/m32 to mm.	DST[31..0] ← SRC; DST[63..32] ← 0
		r/m32, mm	Move double word from mm to r/m32.	DST ← SRC[31..0]
		xmm, r/m32	Move double word from r/m32 to xmm.	DST[31..0] ← SRC; DST[127..32] ← 0
		r/m32, xmm	Move double word from xmm to r/m32.	DST ← SRC[31..0]
Move Qword	MOVQ	mm, r/m64	Move quad word from r/m64 to mm.	DST ← SRC
		m64, mm	Move quad word from mm to m64.	DST ← SRC
		xmm, m64	Move quad word from m64 to xmm.	DST[63..0] ← SRC; DST[127..64] ← 0
		m64, xmm	Move quad word from xmm to m64.	DST← SRC [63..0]

MMX/SSE Conversion instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Pack Signed Saturated Words to Bytes	PACKSSWB	mm1, mm2/m64	Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed signed byte integers in mm1 using signed saturation.	DST[7..0] ← SaturateSignedWordToSignedByte(DST[15..0]); DST[15..8] ← SaturateSignedWordToSignedByte(DST[31..16]); DST[23..16] ← SaturateSignedWordToSignedByte(DST[47..32]); DST[31..24] ← SaturateSignedWordToSignedByte(DST[63..48]); DST[39..32] ← SaturateSignedWordToSignedByte(SRC[15..0]); DST[47..40] ← SaturateSignedWordToSignedByte(SRC[31..16]); DST[55..48] ← SaturateSignedWordToSignedByte(SRC[47..32]); DST[63..56] ← SaturateSignedWordToSignedByte(SRC[63..48]);
		xmm1, xmm2/m128	Converts 8 packed signed word integers from mm1 and from mm2/m128 into 16 packed signed byte integers in mm1 using signed saturation.	DST[7..0] ← SaturateSignedWordToSignedByte(DST[15..0]); DST[15..8] ← SaturateSignedWordToSignedByte(DST[31..16]); DST[23..16] ← SaturateSignedWordToSignedByte(DST[47..32]); DST[31..24] ← SaturateSignedWordToSignedByte(DST[63..48]); DST[39..32] ← SaturateSignedWordToSignedByte(DST[79..64]); DST[47..40] ← SaturateSignedWordToSignedByte(DST[95..80]); DST[55..48] ← SaturateSignedWordToSignedByte(DST[111..96]); DST[63..56] ← SaturateSignedWordToSignedByte(DST[127..112]); DST[71..64] ← SaturateSignedWordToSignedByte(SRC[15..0]); DST[79..72] ← SaturateSignedWordToSignedByte(SRC[31..16]); DST[87..80] ← SaturateSignedWordToSignedByte(SRC[47..32]); DST[95..88] ← SaturateSignedWordToSignedByte(SRC[63..48]); DST[103..96] ← SaturateSignedWordToSignedByte(SRC[79..64]); DST[111..104] ← SaturateSignedWordToSignedByte(SRC[95..80]); DST[119..112] ← SaturateSignedWordToSignedByte(SRC[111..96]); DST[127..120] ← SaturateSignedWordToSignedByte(SRC[127..112]);

Pack Signed Saturated Dwords to Words	PACKSSDW	mm1, mm2/m64	Converts 2 packed signed dword integers from mm1 and from mm2/m64 into 4 packed signed word integers in mm1 using signed saturation.	DST[15..0] ← SaturateSignedDwordToSignedWord(DST[31..0]); DST[32..16] ← SaturateSignedDwordToSignedWord(DST[63..32]); DST[47..32] ← SaturateSignedDwordToSignedWord(SRC[31..0]); DST[63..48] ← SaturateSignedDwordToSignedWord(SRC[63..32]);
		xmm1, xmm2/m128	Converts 4 packed signed dword integers from mm1 and from mm2/m128 into 8 packed signed word integers in mm1 using signed saturation.	DST[15..0] ← SaturateSignedDwordToSignedWord(DST[31..0]); DST[32..16] ← SaturateSignedDwordToSignedWord(DST[63..32]); DST[47..32] ← SaturateSignedDwordToSignedWord(DST[95..64]); DST[63..48] ← SaturateSignedDwordToSignedWord(DST[127..96]); DST[79..64] ←SaturateSignedDwordToSignedWord(SRC[31..0]); DST[95..80] ← SaturateSignedDwordToSignedWord(SRC[63..32]); DST[111..96] ← SaturateSignedDwordToSignedWord(SRC[95..64]); DST[127..112] ← SaturateSignedDwordToSignedWord(SRC[127..96]);
Pack Unsigned Saturated Words to Bytes	PACKUSWB	mm1, mm2/m64	Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed unsigned byte integers in mm1 using unsigned saturation.	DST[7..0] ← SaturateSignedWordToUnsignedByte(DST[15..0]); DST[15..8] ← SaturateSignedWordToUnsignedByte(DST[31..16]); DST[23..16] ← SaturateSignedWordToUnsignedByte(DST[47..32]); DST[31..24] ← SaturateSignedWordToUnsignedByte(DST[63..48]); DST[39..32] ← SaturateSignedWordToUnsignedByte(SRC[15..0]); DST[47..40] ← SaturateSignedWordToUnsignedByte(SRC[31..16]); DST[55..48] ← SaturateSignedWordToUnsignedByte(SRC[47..32]); DST[63..56] ← SaturateSignedWordToUnsignedByte(SRC[63..48]);
		xmm1, xmm2/m128	Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed unsigned byte integers in xmm1 using unsigned saturation.	DST[7..0] ← SaturateSignedWordToUnsignedByte(DST[15..0]); DST[15..8] ← SaturateSignedWordToUnsignedByte(DST[31..16]); DST[23..16] ← SaturateSignedWordToUnsignedByte(DST[47..32]); DST[31..24] ← SaturateSignedWordToUnsignedByte(DST[63..48]); DST[39..32] ← SaturateSignedWordToUnsignedByte(DST[79..64]); DST[47..40] ← SaturateSignedWordToUnsignedByte(DST[95..80]); DST[55..48] ← SaturateSignedWordToUnsignedByte(DST[111..96]); DST[63..56] ← SaturateSignedWordToUnsignedByte(DST[127..112]); DST[71..64] ← SaturateSignedWordToUnsignedByte(SRC[15..0]); DST[79..72] ← SaturateSignedWordToUnsignedByte(SRC[31..16]); DST[87..80] ← SaturateSignedWordToUnsignedByte(SRC[47..32]); DST[95..88] ← SaturateSignedWordToUnsignedByte(SRC[63..48]); DST[103..96] ← SaturateSignedWordToUnsignedByte(SRC[79..64]); DST[111..104] ← SaturateSignedWordToUnsignedByte(SRC[95..80]); DST[119..112] ← SaturateSignedWordToUnsignedByte(SRC[111..96]); DST[127..120] ← SaturateSignedWordToUnsignedByte(SRC[127..112]);

Unpack interleaving Low-order Bytes to Words	PUNPCKLBW	mm1, mm2/m64	Unpacks and interleaves 4 low-order bytes from mm1 and 4 low-order bytes from mm2/m64 into 4 words in mm1.	DST[7..0] ← DST[7..0]; DST[15..8] ← SRC[7..0]; DST[23..16] ← DST[15..8]; DST[31..24] ← SRC[15..8]; DST[39..32] ← DST[23..16]; DST[47..40] ← SRC[23..16]; DST[55..48] ← DST[31..24]; DST[63..56] ← SRC[31..24];
		xmm1, xmm2/m128	Unpacks and interleaves 8 low-order bytes from xmm1 and 8 low-order bytes from xmm2/m128 into 8 words in xmm1.	DST[7..0] ← DST[7..0]; DST[15..8] ← SRC[7..0]; DST[23..16] ← DST[15..8]; DST[31..24] ← SRC[15..8]; DST[39..32] ← DST[23..16]; DST[47..40] ← SRC[23..16]; DST[55..48] ← DST[31..24]; DST[63..56] ← SRC[31..24]; DST[71..64] ← DST[39..32]; DST[79..72] ← SRC[39..32]; DST[87..80] ← DST[47..40]; DST[95..88] ← SRC[47..40]; DST[103..96] ← DST[55..48]; DST[111..104] ← SRC[55..48]; DST[119..112] ← DST[63..56]; DST[127..120] ← SRC[63..56];
Unpack interleaving Low-order Words to Dwords	PUNPCKLWD	mm1, mm2/m64	Unpacks and interleaves 2 low-order words from mm1 and 2 low-order words from mm2/m64 into 2 dwords in mm1.	DST[15..0] ← DST[15..0]; DST[31..16] ← SRC[15..0]; DST[47..32] ← DST[31..16]; DST[63..48] ← SRC[31..16];
		xmm1, xmm2/m128	Unpacks and interleaves 4 low-order words from xmm1 and 4 low-order words from xmm2/m128 into 4 dwords in mm1.	DST[15..0] ← DST[15..0]; DST[31..16] ← SRC[15..0]; DST[47..32] ← DST[31..16]; DST[63..48] ← SRC[31..16]; DST[79..64] ← DST[47..32]; DST[95..80] ← SRC[47..32]; DST[111..96] ← DST[63..48]; DST[127..112] ← SRC[63..48];

Unpack interleaving Low-order Dwords to Qwords	PUNPCKLDQ	xmm1, xmm2/m128	Unpacks and interleaves 2 low-order dwords from xmm1 and 2 low-order dwords from xmm2/m128 into 2 qwords in mm1.	DST[31..0] ← DST[31..0]; DST[63..32] ← SRC[31..0]; DST[95..64] ← DST[63..32]; DST[127..96] ← SRC[63..32];
Unpack interleaving Low-order Qwords to Qwords	PUNPCKLQDQ	xmm1, xmm2/m128	Unpacks and interleaves low-order qword from xmm1 and low-order qword from xmm2/m128 into mm1.	DST[63..0] ← DST[63..0]; DST[127..64] ← SRC[63..0];
Unpack interleaving High-order Bytes to Words	PUNPCKHBW	mm1, mm2/m64	Unpacks and interleaves 4 high-order bytes from mm1 and 4 high-order bytes from mm2/m64 into 4 words in mm1.	DST[7..0] ← DST[39..32]; DST[15..8] ← SRC[39..32]; DST[23..16] ← DST[47..40]; DST[31..24] ← SRC[47..40]; DST[39..32] ← DST[55..48]; DST[47..40] ← SRC[55..48]; DST[55..48] ← DST[63..56]; DST[63..56] ← SRC[63..56];
		xmm1, xmm2/m128	Unpacks and interleaves 8 high-order bytes from xmm1 and 8 high-order bytes from xmm2/m128 into 8 words in xmm1.	DST[7..0] ← DST[71..64]; DST[15..8] ← SRC[71..64]; DST[23..16] ← DST[79..72]; DST[31..24] ← SRC[79..72]; DST[39..32] ← DST[87..80]; DST[47..40] ← SRC[87..80]; DST[55..48] ← DST[95..88]; DST[63..56] ← SRC[95..88]; DST[71..64] ← DST[103..96]; DST[79..72] ← SRC[103..96]; DST[87..80] ← DST[111..104]; DST[95..88] ← SRC[111..104]; DST[103..96] ← DST[119..113]; DST[111..104] ← SRC[119..113]; DST[119..112] ← DST[127..120]; DST[127..120] ← SRC[127..120];

Unpack interleaving High-order Words to Dwords	PUNPCKHWD	mm1, mm2/m64	Unpacks and interleaves 2 high-order words from mm1 and 2 high-order words from mm2/m64 into 2 dwords in mm1.	DST[15..0] ← DST[47..32]; DST[31..16] ← SRC[47..32]; DST[47..32] ← DST[63..48]; DST[63..48] ← SRC[63..48];
		xmm1, xmm2/m128	Unpacks and interleaves 4 high-order words from xmm1 and 4 high-order words from xmm2/m128 into 4 dwords in mm1.	DST[15..0] ← DST[79..64]; DST[31..16] ← SRC[79..64]; DST[47..32] ← DST[95..80]; DST[63..48] ← SRC[95..80]; DST[79..64] ← DST[111..96]; DST[95..80] ← SRC[111..96]; DST[111..96] ← DST[127..112]; DST[127..112] ← SRC[127..112];
Unpack interleaving High-order Dwords to Qwords	PUNPCKHDQ	xmm1, xmm2/m128	Unpacks and interleaves 2 high-order dwords from xmm1 and 2 high-order dwords from xmm2/m128 into 2 qwords in mm1.	DST[31..0] ← DST[95..64]; DST[63..32] ← SRC[95..64]; DST[95..64] ← DST[127..96]; DST[127..96] ← SRC[127..96];
Unpack interleaving High-order Qwords to Qwords	PUNPCKHQDQ	xmm1, xmm2/m128	Unpacks and interleaves high-order qword from xmm1 and high-order qword from xmm2/m128 into mm1.	DST[63..0] ← DST[127..64]; DST[127..64] ← SRC[127..64];

MMX/SSE Packed Arithmetic instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Packed Add Bytes	PADDB	mm1, mm2/m64	Add 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1.	DST[7..0] ← DST[7..0] + SRC[7..0] DST[15..8] ← DST[15..8] + SRC[15..8] DST[23..16] ← DST[23..16] + SRC[23..16] DST[31..24] ← DST[31..24] + SRC[31..24] DST[39..32] ← DST[39..32] + SRC[39..32] DST[47..40] ← DST[47..40] + SRC[47..40] DST[55..48] ← DST[55..48] + SRC[55..48] DST[63..56] ← DST[63..56] + SRC[63..56]
		xmm1, xmm2/m128	Add 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1.	DST[7..0] ← DST[7..0] + SRC[7..0] DST[15..8] ← DST[15..8] + SRC[15..8] DST[23..16] ← DST[23..16] + SRC[23..16] DST[31..24] ← DST[31..24] + SRC[31..24] DST[39..32] ← DST[39..32] + SRC[39..32] DST[47..40] ← DST[47..40] + SRC[47..40] DST[55..48] ← DST[55..48] + SRC[55..48] DST[63..56] ← DST[63..56] + SRC[63..56] DST[71..64] ← DST[71.64] + SRC[71..64] DST[79..72] ← DST[79..72] + SRC[79..72] DST[87..80] ← DST[87..80] + SRC[87..80] DST[95..88] ← DST[95..88] + SRC[95..88] DST[103..96] ← DST[103..96] + SRC[103..96] DST[111..104] ← DST[111..104] + SRC[111..104] DST[119..112] ← DST[119..112] + SRC[119..112] DST[127..120] ← DST[127..120] + SRC[127..120]
Packed Add Words	PADDW	mm1, mm2/m64	Add 4 packed word integers from mm2/m64 to 4 packed word integers in mm1.	DST[15..0] ← DST[15..0] + SRC[15..0] DST[31..16] ← DST[31..16] + SRC[31..16] DST[47..32] ← DST[47..32] + SRC[47..32] DST[63..48] ← DST[63..48] + SRC[63..48]
		xmm1, xmm2/m128	Add 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1.	DST[15..0] ← DST[15..0] + SRC[15..0] DST[31..16] ← DST[31..16] + SRC[31..16] DST[47..32] ← DST[47..32] + SRC[47..32] DST[63..48] ← DST[63..48] + SRC[63..48] DST[79..64] ← DST[79..64] + SRC[79..64] DST[95..80] ← DST[95..80] + SRC[95..80] DST[111..96] ← DST[111..96] + SRC[111..96] DST[127..112] ← DST[127..112] + SRC[127..112]

Packed Add Dwords	PADDD	mm1, mm2/m64	Add 2 packed double-word integers from mm2/m64 to 2 packed double-word integers in mm1.	DST[31..0] ← DST[31..0] + SRC[31..0] DST[63..32] ← DST[63..32] + SRC[63..32]
		xmm1, xmm2/m128	Add 4 packed double-word integers from xmm2/m128 to 2 packed double-word integers in xmm1.	DST[31..0] ← DST[31..0] + SRC[31..0] DST[63..32] ← DST[63..32] + SRC[63..32] DST[95..64] ← DST[95..64] + SRC[95..64] DST[127..96] ← DST[127..96] + SRC[127..96]
Packed Add Bytes with Saturation	PADDSB	mm1, mm2/m64	Add 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1. Overflow is handled with signed saturation.	DST[7..0] ← SaturateToSignedByte (DST[7..0] + SRC[7..0]) DST[15..8] ← SaturateToSignedByte (DST[15..8] + SRC[15..8]) DST[23..16] ← SaturateToSignedByte (DST[23..16] + SRC[23..16]) DST[31..24] ← SaturateToSignedByte (DST[31..24] + SRC[31..24]) DST[39..32] ← SaturateToSignedByte (DST[39..32] + SRC[39..32]) DST[47..40] ← SaturateToSignedByte (DST[47..40] + SRC[47..40]) DST[55..48] ← SaturateToSignedByte (DST[55..48] + SRC[55..48]) DST[63..56] ← SaturateToSignedByte (DST[63..56] + SRC[63..56])
		xmm1, xmm2/m128	Add 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1. Overflow is handled with signed saturation.	DST[7..0] ← SaturateToSignedByte (DST[7..0] + SRC[7..0]) DST[15..8] ← SaturateToSignedByte (DST[15..8] + SRC[15..8]) DST[23..16] ← SaturateToSignedByte (DST[23..16] + SRC[23..16]) DST[31..24] ← SaturateToSignedByte (DST[31..24] + SRC[31..24]) DST[39..32] ← SaturateToSignedByte (DST[39..32] + SRC[39..32]) DST[47..40] ← SaturateToSignedByte (DST[47..40] + SRC[47..40]) DST[55..48] ← SaturateToSignedByte (DST[55..48] + SRC[55..48]) DST[63..56] ← SaturateToSignedByte (DST[63..56] + SRC[63..56]) DST[71..64] ← SaturateToSignedByte (DST[71.64] + SRC[71..64]) DST[79..72] ← SaturateToSignedByte (DST[79..72] + SRC[79..72]) DST[87..80] ← SaturateToSignedByte (DST[87..80] + SRC[87..80]) DST[95..88] ← SaturateToSignedByte (DST[95..88] + SRC[95..88]) DST[103..96] ← SaturateToSignedByte (DST[103..96] + SRC[103..96]) DST[111..104] ← SaturateToSignedByte (DST[111..104] + SRC[111..104]) DST[119..112] ← SaturateToSignedByte (DST[119..112] + SRC[119..112]) DST[127..120] ← SaturateToSignedByte (DST[127..120] + SRC[127..120])

Packed Add Words with Saturation	PADDSW	mm1, mm2/m64	Add 4 packed word integers from mm2/m64 to 4 packed word integers in mm1. Overflow is handled with signed saturation.	DST[15..0] ← SaturateToSignedWord (DST[15..0] + SRC[15..0]) DST[31..16] ← SaturateToSignedWord (DST[31..16] + SRC[31..16]) DST[47..32] ← SaturateToSignedWord (DST[47..32] + SRC[47..32]) DST[63..48] ← SaturateToSignedWord (DST[63..48] + SRC[63..48])
		xmm1, xmm2/m128	Add 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1. Overflow is handled with signed saturation.	DST[15..0] ← SaturateToSignedWord (DST[15..0] + SRC[15..0]) DST[31..16] ← SaturateToSignedWord (DST[31..16] + SRC[31..16]) DST[47..32] ← SaturateToSignedWord (DST[47..32] + SRC[47..32]) DST[63..48] ← SaturateToSignedWord (DST[63..48] + SRC[63..48]) DST[79..64] ← SaturateToSignedWord (DST[79..64] + SRC[79..64]) DST[95..80] ← SaturateToSignedWord (DST[95..80] + SRC[95..80]) DST[111..96] ← SaturateToSignedWord (DST[111..96] + SRC[111..96]) DST[127..112] ← SaturateToSignedWord (DST[127..112] + SRC[127..112])
Packed Add Bytes with Unsigned Saturation	PADDUSB	mm1, mm2/m64	Add 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1. Overflow is handled with unsigned saturation.	DST[7..0] ← SaturateToUnsignedByte (DST[7..0] + SRC[7..0]) DST[15..8] ← SaturateToUnsignedByte (DST[15..8] + SRC[15..8]) DST[23..16] ← SaturateToUnsignedByte (DST[23..16] + SRC[23..16]) DST[31..24] ← SaturateToUnsignedByte (DST[31..24] + SRC[31..24]) DST[39..32] ← SaturateToUnsignedByte (DST[39..32] + SRC[39..32]) DST[47..40] ← SaturateToUnsignedByte (DST[47..40] + SRC[47..40]) DST[55..48] ← SaturateToUnsignedByte (DST[55..48] + SRC[55..48]) DST[63..56] ← SaturateToUnsignedByte (DST[63..56] + SRC[63..56])
		xmm1, xmm2/m128	Add 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1. Overflow is handled with unsigned saturation.	DST[7..0] ← SaturateToUnsignedByte (DST[7..0] + SRC[7..0]) DST[15..8] ← SaturateToUnsignedByte (DST[15..8] + SRC[15..8]) DST[23..16] ← SaturateToUnsignedByte (DST[23..16] + SRC[23..16]) DST[31..24] ← SaturateToUnsignedByte (DST[31..24] + SRC[31..24]) DST[39..32] ← SaturateToUnsignedByte (DST[39..32] + SRC[39..32]) DST[47..40] ← SaturateToUnsignedByte (DST[47..40] + SRC[47..40]) DST[55..48] ← SaturateToUnsignedByte (DST[55..48] + SRC[55..48]) DST[63..56] ← SaturateToUnsignedByte (DST[63..56] + SRC[63..56]) DST[71..64] ← SaturateToUnsignedByte (DST[71.64] + SRC[71..64]) DST[79..72] ← SaturateToUnsignedByte (DST[79..72] + SRC[79..72]) DST[87..80] ← SaturateToUnsignedByte (DST[87..80] + SRC[87..80]) DST[95..88] ← SaturateToUnsignedByte (DST[95..88] + SRC[95..88]) DST[103..96] ← SaturateToUnsignedByte (DST[103..96] + SRC[103..96]) DST[111..104] ← SaturateToUnsignedByte (DST[111..104] + SRC[111..104]) DST[119..112] ← SaturateToUnsignedByte (DST[119..112] + SRC[119..112]) DST[127..120] ← SaturateToUnsignedByte (DST[127..120] + SRC[127..120])

Packed Add Words with Unsigned Saturation	PADDUSW	mm1, mm2/m64	Add 4 packed word integers from mm2/m64 to 4 packed word integers in mm1. Overflow is handled with unsigned saturation.	DST[15..0] ← SaturateToUnsignedWord (DST[15..0] + SRC[15..0]) DST[31..16] ← SaturateToUnsignedWord (DST[31..16] + SRC[31..16]) DST[47..32] ← SaturateToUnsignedWord (DST[47..32] + SRC[47..32]) DST[63..48] ← SaturateToUnsignedWord (DST[63..48] + SRC[63..48])
		xmm1, xmm2/m128	Add 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1. Overflow is handled with unsigned saturation.	DST[15..0] ← SaturateToUnsignedWord (DST[15..0] + SRC[15..0]) DST[31..16] ← SaturateToUnsignedWord (DST[31..16] + SRC[31..16]) DST[47..32] ← SaturateToUnsignedWord (DST[47..32] + SRC[47..32]) DST[63..48] ← SaturateToUnsignedWord (DST[63..48] + SRC[63..48]) DST[79..64] ← SaturateToUnsignedWord (DST[79..64] + SRC[79..64]) DST[95..80] ← SaturateToUnsignedWord (DST[95..80] + SRC[95..80]) DST[111..96] ← SaturateToUnsignedWord (DST[111..96] + SRC[111..96]) DST[127..112] ← SaturateToUnsignedWord (DST[127..112] + SRC[127..112])
Packed Subtract Bytes	PSUBB	mm1, mm2/m64	Subtract 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1.	DST[7..0] ← DST[7..0] - SRC[7..0] DST[15..8] ← DST[15..8] - SRC[15..8] DST[23..16] ← DST[23..16] - SRC[23..16] DST[31..24] ← DST[31..24] - SRC[31..24] DST[39..32] ← DST[39..32] - SRC[39..32] DST[47..40] ← DST[47..40] - SRC[47..40] DST[55..48] ← DST[55..48] - SRC[55..48] DST[63..56] ← DST[63..56] - SRC[63..56]
		xmm1, xmm2/m128	Subtract 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1.	DST[7..0] ← DST[7..0] - SRC[7..0] DST[15..8] ← DST[15..8] - SRC[15..8] DST[23..16] ← DST[23..16] - SRC[23..16] DST[31..24] ← DST[31..24] - SRC[31..24] DST[39..32] ← DST[39..32] - SRC[39..32] DST[47..40] ← DST[47..40] - SRC[47..40] DST[55..48] ← DST[55..48] - SRC[55..48] DST[63..56] ← DST[63..56] - SRC[63..56] DST[71..64] ← DST[71.64] - SRC[71..64] DST[79..72] ← DST[79..72] - SRC[79..72] DST[87..80] ← DST[87..80] - SRC[87..80] DST[95..88] ← DST[95..88] - SRC[95..88] DST[103..96] ← DST[103..96] - SRC[103..96] DST[111..104] ← DST[111..104] - SRC[111..104] DST[119..112] ← DST[119..112] - SRC[119..112] DST[127..120] ← DST[127..120] - SRC[127..120]

Packed Subtract Words	PSUBW	mm1, mm2/m64	Subtract 4 packed word integers from mm2/m64 to 4 packed word integers in mm1.	DST[15..0] ← DST[15..0] - SRC[15..0] DST[31..16] ← DST[31..16] - SRC[31..16] DST[47..32] ← DST[47..32] - SRC[47..32] DST[63..48] ← DST[63..48] - SRC[63..48]
		xmm1, xmm2/m128	Subtract 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1.	DST[15..0] ← DST[15..0] - SRC[15..0] DST[31..16] ← DST[31..16] - SRC[31..16] DST[47..32] ← DST[47..32] - SRC[47..32] DST[63..48] ← DST[63..48] - SRC[63..48] DST[79..64] ← DST[79..64] - SRC[79..64] DST[95..80] ← DST[95..80] - SRC[95..80] DST[111..96] ← DST[111..96] - SRC[111..96] DST[127..112] ← DST[127..112] - SRC[127..112]
Packed Subtract Dwords	PSUBD	mm1, mm2/m64	Subtract 2 packed double-word integers from mm2/m64 to 2 packed double-word integers in mm1.	DST[31..0] ← DST[31..0] - SRC[31..0] DST[63..32] ← DST[63..32] - SRC[63..32]
		xmm1, xmm2/m128	Subtract 4 packed double-word integers from xmm2/m128 to 2 packed double-word integers in xmm1.	DST[31..0] ← DST[31..0] - SRC[31..0] DST[63..32] ← DST[63..32] - SRC[63..32] DST[95..64] ← DST[95..64] - SRC[95..64] DST[127..96] ← DST[127..96] - SRC[127..96]
Packed Subtract Bytes with Saturation	PSUBSB	mm1, mm2/m64	Subtract 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1. Overflow is handled with signed saturation.	DST[7..0] ← SaturateToSignedByte (DST[7..0] - SRC[7..0]) DST[15..8] ← SaturateToSignedByte (DST[15..8] - SRC[15..8]) DST[23..16] ← SaturateToSignedByte (DST[23..16] - SRC[23..16]) DST[31..24] ← SaturateToSignedByte (DST[31..24] - SRC[31..24]) DST[39..32] ← SaturateToSignedByte (DST[39..32] - SRC[39..32]) DST[47..40] ← SaturateToSignedByte (DST[47..40] - SRC[47..40]) DST[55..48] ← SaturateToSignedByte (DST[55..48] - SRC[55..48]) DST[63..56] ← SaturateToSignedByte (DST[63..56] - SRC[63..56])

		xmm1, xmm2/m128	Subtract 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1. Overflow is handled with signed saturation.	DST[7..0] ← SaturateToSignedByte (DST[7..0] - SRC[7..0]) DST[15..8] ← SaturateToSignedByte (DST[15..8] - SRC[15..8]) DST[23..16] ← SaturateToSignedByte (DST[23..16] - SRC[23..16]) DST[31..24] ← SaturateToSignedByte (DST[31..24] - SRC[31..24]) DST[39..32] ← SaturateToSignedByte (DST[39..32] - SRC[39..32]) DST[47..40] ← SaturateToSignedByte (DST[47..40] - SRC[47..40]) DST[55..48] ← SaturateToSignedByte (DST[55..48] - SRC[55..48]) DST[63..56] ← SaturateToSignedByte (DST[63..56] - SRC[63..56]) DST[71..64] ← SaturateToSignedByte (DST[71.64] - SRC[71..64]) DST[79..72] ← SaturateToSignedByte (DST[79..72] - SRC[79..72]) DST[87..80] ← SaturateToSignedByte (DST[87..80] - SRC[87..80]) DST[95..88] ← SaturateToSignedByte (DST[95..88] - SRC[95..88]) DST[103..96] ← SaturateToSignedByte (DST[103..96] - SRC[103..96]) DST[111..104] ← SaturateToSignedByte (DST[111..104] - SRC[111..104]) DST[119..112] ← SaturateToSignedByte (DST[119..112] - SRC[119..112]) DST[127..120] ← SaturateToSignedByte (DST[127..120] - SRC[127..120])
Packed Subtract Words with Saturation	PSUBSW	mm1, mm2/m64	Subtract 4 packed word integers from mm2/m64 to 4 packed word integers in mm1. Overflow is handled with signed saturation.	DST[15..0] ← SaturateToSignedWord (DST[15..0] - SRC[15..0]) DST[31..16] ← SaturateToSignedWord (DST[31..16] - SRC[31..16]) DST[47..32] ← SaturateToSignedWord (DST[47..32] - SRC[47..32]) DST[63..48] ← SaturateToSignedWord (DST[63..48] - SRC[63..48])
		xmm1, xmm2/m128	Subtract 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1. Overflow is handled with signed saturation.	DST[15..0] ← SaturateToSignedWord (DST[15..0] - SRC[15..0]) DST[31..16] ← SaturateToSignedWord (DST[31..16] - SRC[31..16]) DST[47..32] ← SaturateToSignedWord (DST[47..32] - SRC[47..32]) DST[63..48] ← SaturateToSignedWord (DST[63..48] - SRC[63..48]) DST[79..64] ← SaturateToSignedWord (DST[79..64] - SRC[79..64]) DST[95..80] ← SaturateToSignedWord (DST[95..80] - SRC[95..80]) DST[111..96] ← SaturateToSignedWord (DST[111..96] - SRC[111..96]) DST[127..112] ← SaturateToSignedWord (DST[127..112] - SRC[127..112])
Packed Subtract Bytes with Unsigned Saturation	PSUBUSB	mm1, mm2/m64	Subtract 8 packed byte integers from mm2/m64 to 8 packed byte integers in mm1. Overflow is handled with unsigned saturation.	DST[7..0] ← SaturateToUnsignedByte (DST[7..0] - SRC[7..0]) DST[15..8] ← SaturateToUnsignedByte (DST[15..8] - SRC[15..8]) DST[23..16] ← SaturateToUnsignedByte (DST[23..16] - SRC[23..16]) DST[31..24] ← SaturateToUnsignedByte (DST[31..24] - SRC[31..24]) DST[39..32] ← SaturateToUnsignedByte (DST[39..32] - SRC[39..32]) DST[47..40] ← SaturateToUnsignedByte (DST[47..40] - SRC[47..40]) DST[55..48] ← SaturateToUnsignedByte (DST[55..48] - SRC[55..48]) DST[63..56] ← SaturateToUnsignedByte (DST[63..56] - SRC[63..56])

		xmm1, xmm2/m128	Subtract 16 packed byte integers from xmm2/m128 to 16 packed byte integers in xmm1. Overflow is handled with unsigned saturation.	DST[7..0] ← SaturateToUnsignedByte (DST[7..0] - SRC[7..0]) DST[15..8] ← SaturateToUnsignedByte (DST[15..8] - SRC[15..8]) DST[23..16] ← SaturateToUnsignedByte (DST[23..16] - SRC[23..16]) DST[31..24] ← SaturateToUnsignedByte (DST[31..24] - SRC[31..24]) DST[39..32] ← SaturateToUnsignedByte (DST[39..32] - SRC[39..32]) DST[47..40] ← SaturateToUnsignedByte (DST[47..40] - SRC[47..40]) DST[55..48] ← SaturateToUnsignedByte (DST[55..48] - SRC[55..48]) DST[63..56] ← SaturateToUnsignedByte (DST[63..56] - SRC[63..56]) DST[71..64] ← SaturateToUnsignedByte (DST[71.64] - SRC[71..64]) DST[79..72] ← SaturateToUnsignedByte (DST[79..72] - SRC[79..72]) DST[87..80] ← SaturateToUnsignedByte (DST[87..80] - SRC[87..80]) DST[95..88] ← SaturateToUnsignedByte (DST[95..88] - SRC[95..88]) DST[103..96] ← SaturateToUnsignedByte (DST[103..96] - SRC[103..96]) DST[111..104] ← SaturateToUnsignedByte (DST[111..104] - SRC[111..104]) DST[119..112] ← SaturateToUnsignedByte (DST[119..112] - SRC[119..112]) DST[127..120] ← SaturateToUnsignedByte (DST[127..120] - SRC[127..120])
Packed Subtract Words with Unsigned Saturation	PSUBUSW	mm1, mm2/m64	Subtract 4 packed word integers from mm2/m64 to 4 packed word integers in mm1. Overflow is handled with unsigned saturation.	DST[15..0] ← SaturateToUnsignedWord (DST[15..0] - SRC[15..0]) DST[31..16] ← SaturateToUnsignedWord (DST[31..16] - SRC[31..16]) DST[47..32] ← SaturateToUnsignedWord (DST[47..32] - SRC[47..32]) DST[63..48] ← SaturateToUnsignedWord (DST[63..48] - SRC[63..48])
		xmm1, xmm2/m128	Subtract 8 packed word integers from xmm2/m128 to 8 packed word integers in xmm1. Overflow is handled with unsigned saturation.	DST[15..0] ← SaturateToUnsignedWord (DST[15..0] - SRC[15..0]) DST[31..16] ← SaturateToUnsignedWord (DST[31..16] - SRC[31..16]) DST[47..32] ← SaturateToUnsignedWord (DST[47..32] - SRC[47..32]) DST[63..48] ← SaturateToUnsignedWord (DST[63..48] - SRC[63..48]) DST[79..64] ← SaturateToUnsignedWord (DST[79..64] - SRC[79..64]) DST[95..80] ← SaturateToUnsignedWord (DST[95..80] - SRC[95..80]) DST[111..96] ← SaturateToUnsignedWord (DST[111..96] - SRC[111..96]) DST[127..112] ← SaturateToUnsignedWord (DST[127..112] - SRC[127..112])

Packed Multiply, Low Word	PMULLW	mm1, mm2/m64	Multiply 4 packed word integers from mm2/m64 by 4 packed word integers from mm1. Store low-order result words in mm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] DST[15..0] ← TMP[015..0] DST[31..16] ← TMP1[15..0] DST[47..32] ← TMP2[15..0] DST[63..48] ← TMP3[15..0]
		xmm1, xmm2/m128	Multiply 8 packed word integers from xmm2/m128 by 8 packed word integers from xmm1. Store low-order result words in xmm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] TMP4[31..0] ← DST[79..64] * SRC[79..64] TMP5[31..0] ← DST[95..80] * SRC[95..80] TMP6[31..0] ← DST[111..96] * SRC[111..96] TMP7[31..0] ← DST[127..112] * SRC[127..112] DST[15..0] ← TMP0[15..0] DST[31..16] ← TMP1[15..0] DST[47..32] ← TMP2[15..0] DST[63..48] ← TMP3[15..0] DST[79..64] ← TMP4[15..0] DST[95..80] ← TMP5[15..0] DST[111..96] ← TMP6[15..0] DST[127..112] ← TMP7[15..0]

Packed Multiply, High Word	PMULHW	mm1, mm2/m64	Multiply 4 packed word integers from mm2/m64 by 4 packed word integers from mm1. Store high-order result words in mm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] DST[15..0] ← TMP0[31..16] DST[31..16] ← TMP1[31..16] DST[47..32] ← TMP2[31..16] DST[63..48] ← TMP3[31..16]
		xmm1, xmm2/m128	Multiply 8 packed word integers from xmm2/m128 by 8 packed word integers from xmm1. Store high-order result words in xmm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] TMP4[31..0] ← DST[79..64] * SRC[79..64] TMP5[31..0] ← DST[95..80] * SRC[95..80] TMP6[31..0] ← DST[111..96] * SRC[111..96] TMP7[31..0] ← DST[127..112] * SRC[127..112] DST[15..0] ← TMP0[31..16] DST[31..16] ← TMP1[31..16] DST[47..32] ← TMP2[31..16] DST[63..48] ← TMP3[31..16] DST[79..64] ← TMP4[31..16] DST[95..80] ← TMP5[31..16] DST[111..96] ← TMP6[31..16] DST[127..112] ← TMP7[31..16]

Packed Multiply Unsigned, High Word	PMULUHW	mm1, mm2/m64	Multiply 4 packed word integers from mm2/m64 by 4 packed word integers from mm1. Treat integers as unsigned. Store high-order result words in mm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] // *unsigned multiplication* TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] DST[15..0] ← TMP0[31..16] DST[31..16] ← TMP1[31..16] DST[47..32] ← TMP2[31..16] DST[63..48] ← TMP3[31..16]
		xmm1, xmm2/m128	Multiply 8 packed word integers from xmm2/m128 by 8 packed word integers from xmm1. Treat integers as unsigned. Store high-order result words in xmm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] // *unsigned multiplication* TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] TMP4[31..0] ← DST[79..64] * SRC[79..64] TMP5[31..0] ← DST[95..80] * SRC[95..80] TMP6[31..0] ← DST[111..96] * SRC[111..96] TMP7[31..0] ← DST[127..112] * SRC[127..112] DST[15..0] ← TMP0[31..16] DST[31..16] ← TMP1[31..16] DST[47..32] ← TMP2[31..16] DST[63..48] ← TMP3[31..16] DST[79..64] ← TMP4[31..16] DST[95..80] ← TMP5[31..16] DST[111..96] ← TMP6[31..16] DST[127..112] ← TMP7[31..16]

Packed Multiply Add Word	PMADDWD	mm1, mm2/m64	Multiply 4 packed words in mm1 by 4 packed words in mm2/m64, add adjacent double word results and store in mm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] DST[31..0] ← TMP0[31..0] + TMP1[31..0] DST[63..32] ← TMP2[31..0] + TMP3[31..0]
		xmm1, xmm2/m128	Multiply 8 packed words in xmm1 by 8 packed words in xmm2/m128, add adjacent double word results and store in mm1.	TMP0[31..0] ← DST[15..0] * SRC[15..0] TMP1[31..0] ← DST[31..16] * SRC[31..16] TMP2[31..0] ← DST[47..32] * SRC[47..32] TMP3[31..0] ← DST[63..48] * SRC[63..48] TMP4[31..0] ← DST[79..64] * SRC[79..64] TMP5[31..0] ← DST[95..80] * SRC[95..80] TMP6[31..0] ← DST[111..96] * SRC[111..96] TMP7[31..0] ← DST[127..112] * SRC[127..112] DST[31..0] ← TMP0[31..0] + TMP1[31..0] DST[63..32] ← TMP2[31..0] + TMP3[31..0] DST[95..64] ← TMP4[31..0] + TMP5[31..0] DST[127..96] ← TMP6[31..0] + TMP7[31..0]

MMX/SSE Comparison instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Parallel Compare Bytes for Equality	PCMPEQB	mm1, mm2/m64	Compare 8 packed bytes in mm1 and mm2/m64 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[7..0] = SRC[7..0] THEN DST[7..0] ← 0FFH ELSE DST [7..0] ← 00H IF DST[15..8] = SRC[15..8] THEN DST[15..8] ← 0FFH ELSE DST [15..8] ← 00H IF DST[23..16] = SRC[23..16] THEN DST[23..16] ← 0FFH ELSE DST [23..16] ← 00H IF DST[31..24] = SRC[31..23] THEN DST[31..24] ← 0FFH ELSE DST [31..24] ← 00H IF DST[39..32] = SRC[39..32] THEN DST[39..32] ← 0FFH ELSE DST [39..32] ← 00H IF DST[47..40] = SRC[47..40] THEN DST[47..40] ← 0FFH ELSE DST [47..40] ← 00H IF DST[55..48] = SRC[55..48] THEN DST[55..48] ← 0FFH ELSE DST [55..48] ← 00H IF DST[63..56] = SRC[63..56] THEN DST[63..56] ← 0FFH ELSE DST [63..56] ← 00H
		xmm1, xmm2/m128	Compare 16 packed bytes in mm1 and mm2/m128 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[7..0] = SRC[7..0] THEN DST[7..0] ← 0FFH ELSE DST [7..0] ← 00H IF DST[15..8] = SRC[15..8] THEN DST[15..8] ← 0FFH ELSE DST [15..8] ← 00H IF DST[23..16] = SRC[23..16] THEN DST[23..16] ← 0FFH ELSE DST [23..16] ← 00H IF DST[31..24] = SRC[31..23] THEN DST[31..24] ← 0FFH ELSE DST [31..24] ← 00H IF DST[39..32] = SRC[39..32] THEN DST[39..32] ← 0FFH ELSE DST [39..32] ← 00H IF DST[47..40] = SRC[47..40] THEN DST[47..40] ← 0FFH ELSE DST [47..40] ← 00H IF DST[55..48] = SRC[55..48] THEN DST[55..48] ← 0FFH ELSE DST [55..48] ← 00H IF DST[63..56] = SRC[63..56] THEN DST[63..56] ← 0FFH ELSE DST [63..56] ← 00H IF DST[71..64] = SRC[71..63] THEN DST[71..64] ← 0FFH ELSE DST [71..64] ← 00H IF DST[79..72] = SRC[79..72] THEN DST[79..72] ← 0FFH ELSE DST [79..72] ← 00H IF DST[87..80] = SRC[87..80] THEN DST[87..80] ← 0FFH ELSE DST [87..80] ← 00H IF DST[95..88] = SRC[95..88] THEN DST[95..88] ← 0FFH ELSE DST [95..88] ← 00H IF DST[103..96] = SRC[103..96] THEN DST[103..96] ← 0FFH ELSE DST [103..96] ← 00H IF DST[111..104] = SRC[111..103] THEN DST[111..104] ← 0FFH ELSE DST [111..104] ← 00H

				IF DST[119..112] = SRC[119..112] THEN DST[119..112] ← 0FFH ELSE DST [119..112] ← 00H IF DST[127..120] = SRC[127..120] THEN DST[127..120] ← 0FFH ELSE DST [127..120] ← 00H
Parallel Compare Words for Equality	PCMPEQW	mm1, mm2/m64	Compare 4 packed words in mm1 and mm2/m64 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected. DST 63 SRC 63 DST 63	IF DST[15..0] = SRC[15..0] THEN DST[15..0] ← 0FFH ELSE DST [15..0] ← 00H IF DST[31..16] = SRC[31..16] THEN DST[31..16] ← 0FFH ELSE DST [31..16] ← 00H IF DST[47..32] = SRC[47..32] THEN DST[47..32] ← 0FFH ELSE DST [47..32] ← 00H IF DST[63..48] = SRC[63..48] THEN DST[63..48] ← 0FFH ELSE DST [63..48] ← 00H
		xmm1, xmm2/m128	Compare 8 packed words in mm1 and mm2/m128 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected. DST 127 SRC 127 DST 127	IF DST[15..0] = SRC[15..0] THEN DST[15..0] ← 0FFH ELSE DST [15..0] ← 00H IF DST[31..16] = SRC[31..16] THEN DST[31..16] ← 0FFH ELSE DST [31..16] ← 00H IF DST[47..32] = SRC[47..32] THEN DST[47..32] ← 0FFH ELSE DST [47..32] ← 00H IF DST[63..48] = SRC[63..48] THEN DST[63..48] ← 0FFH ELSE DST [63..48] ← 00H IFDST[79..64] = SRC[79..64] THEN DST[79..64] ← 0FFH ELSE DST [79..64] ← 00H IF DST[95..80] = SRC[95..80] THEN DST[95..80] ← 0FFH ELSE DST [95..80] ← 00H IF DST[111..96] = SRC[111..96] THEN DST[111..96] ← 0FFH ELSE DST [111..96] ← 00H IF DST[127..112] = SRC[127..112] THEN DST[127..112] ← 0FFH ELSE DST [127..112] ← 00H
Parallel Compare Dwords for Equality	PCMPEQD	mm1, mm2/m64	Compare 2 packed double words in mm1 and mm2/m64 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected. DST 63 SRC 63 DST 63	IF DST[31..0] = SRC[31..0] THEN DST[31..0] ← 0FFH ELSE DST [31..0] ← 00H IF DST[63..32] = SRC[63..32] THEN DST[63..32] ← 0FFH ELSE DST [63..32] ← 00H

		xmm1, xmm2/m128	Compare 4 packed double words in mm1 and mm2/m128 for equality. If a pair of data element is equal, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[31..0] = SRC[31..0] THEN DST[31..0] ← 0FFH ELSE DST [31..0] ← 00H IF DST[63..32] = SRC[63..32] THEN DST[63..32] ← 0FFH ELSE DST [63..32] ← 00H IF DST[95..64] = SRC[95..64] THEN DST[95..64] ← 0FFH ELSE DST [95..64] ← 00H IF DST[127..96] = SRC[127..96] THEN DST[127..96] ← 0FFH ELSE DST [127..96] ← 00H
Parallel Compare Bytes for Greater	PCMPGTB	mm1, mm2/m64	Compare 8 packed bytes in mm1 and mm2/m64 for greater. Bytes are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[7..0] > SRC[7..0] THEN DST[7..0] ← 0FFH ELSE DST [7..0] ← 00H IF DST[15..8] > SRC[15..8] THEN DST[15..8] ← 0FFH ELSE DST [15..8] ← 00H IF DST[23..16] > SRC[23..16] THEN DST[23..16] ← 0FFH ELSE DST [23..16] ← 00H IF DST[31..24] > SRC[31..23] THEN DST[31..24] ← 0FFH ELSE DST [31..24] ← 00H IF DST[39..32] > SRC[39..32] THEN DST[39..32] ← 0FFH ELSE DST [39..32] ← 00H IF DST[47..40] > SRC[47..40] THEN DST[47..40] ← 0FFH ELSE DST [47..40] ← 00H IF DST[55..48] > SRC[55..48] THEN DST[55..48] ← 0FFH ELSE DST [55..48] ← 00H IF DST[63..56] > SRC[63..56] THEN DST[63..56] ← 0FFH ELSE DST [63..56] ← 00H
		xmm1, xmm2/m128	Compare 16 packed bytes in mm1 and mm2/m128 for greater. Bytes are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[7..0] > SRC[7..0] THEN DST[7..0] ← 0FFH ELSE DST [7..0] ← 00H IF DST[15..8] > SRC[15..8] THEN DST[15..8] ← 0FFH ELSE DST [15..8] ← 00H IF DST[23..16] > SRC[23..16] THEN DST[23..16] ← 0FFH ELSE DST [23..16] ← 00H IF DST[31..24] > SRC[31..23] THEN DST[31..24] ← 0FFH ELSE DST [31..24] ← 00H IF DST[39..32] > SRC[39..32] THEN DST[39..32] ← 0FFH ELSE DST [39..32] ← 00H IF DST[47..40] > SRC[47..40] THEN DST[47..40] ← 0FFH ELSE DST [47..40] ← 00H IF DST[55..48] > SRC[55..48] THEN DST[55..48] ← 0FFH ELSE DST [55..48] ← 00H IF DST[63..56] > SRC[63..56] THEN DST[63..56] ← 0FFH ELSE DST [63..56] ← 00H IF DST[71..64] > SRC[71..63] THEN DST[71..64] ← 0FFH ELSE DST [71..64] ← 00H IF DST[79..72] > SRC[79..72] THEN DST[79..72] ← 0FFH ELSE DST [79..72] ← 00H

				IF DST[87..80] > SRC[87..80] THEN DST[87..80] ← 0FFH ELSE DST [87..80] ← 00H IF DST[95..88] > SRC[95..88] THEN DST[95..88] ← 0FFH ELSE DST [95..88] ← 00H IF DST[103..96] > SRC[103..96] THEN DST[103..96] ← 0FFH ELSE DST [103..96] ← 00H IF DST[111..104] > SRC[111..103] THEN DST[111..104] ← 0FFH ELSE DST [111..104] ← 00H IF DST[119..112] > SRC[119..112] THEN DST[119..112] ← 0FFH ELSE DST [119..112] ← 00H IF DST[127..120] > SRC[127..120] THEN DST[127..120] ← 0FFH ELSE DST [127..120] ← 00H
Parallel Compare Words for Greater	PCMPGTW	mm1, mm2/m64	Compare 4 packed words in mm1 and mm2/m64 for greater. Words are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[15..0] > SRC[15..0] THEN DST[15..0] ← 0FFFFH ELSE DST [15..0] ← 00H IF DST[31..16] > SRC[31..16] THEN DST[31..16] ← 0FFFFH ELSE DST [31..16] ← 00H IF DST[47..32] > SRC[47..32] THEN DST[47..32] ← 0FFFFH ELSE DST [47..32] ← 00H IF DST[63..48] > SRC[63..48] THEN DST[63..48] ← 0FFFFH ELSE DST [63..48] ← 00H
		xmm1, xmm2/m128	Compare 8 packed words in mm1 and mm2/m128 for greater. Words are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[15..0] > SRC[15..0] THEN DST[15..0] ← 0FFFFH ELSE DST [15..0] ← 00H IF DST[31..16] > SRC[31..16] THEN DST[31..16] ← 0FFFFH ELSE DST [31..16] ← 00H IF DST[47..32] > SRC[47..32] THEN DST[47..32] ← 0FFFFH ELSE DST [47..32] ← 00H IF DST[63..48] > SRC[63..48] THEN DST[63..48] ← 0FFFFH ELSE DST [63..48] ← 00H IFDST[79..64] > SRC[79..64] THEN DST[79..64] ← 0FFFFH ELSE DST [79..64] ← 00H IF DST[95..80] > SRC[95..80] THEN DST[95..80] ← 0FFFFH ELSE DST [95..80] ← 00H IF DST[111..96] > SRC[111..96] THEN DST[111..96] ← 0FFFFH ELSE DST [111..96] ← 00H IF DST[127..112] > SRC[127..112] THEN DST[127..112] ← 0FFFFH ELSE DST [127..112] ← 00H

Parallel Compare Dwords for Greater	PCMPGTD	mm1, mm2/m64	Compare 2 packed double words in mm1 and mm2/m64 for greater. Dwords are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[31..0] > SRC[31..0] THEN DST[31..0] ← 0FFFFFFFFH ELSE DST [31..0] ← 00H IF DST[63..32] > SRC[63..32] THEN DST[63..32] ← 0FFFFFFFFH ELSE DST [63..32] ← 00H
		xmm1, xmm2/m128	Compare 4 packed double words in mm1 and mm2/m128 for greater. Dwords are treated as signed integers. If a pair of data element is greater, then the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. No flags in the EFLAGS registers are affected.	IF DST[31..0] > SRC[31..0] THEN DST[31..0] ← 0FFFFFFFFH ELSE DST [31..0] ← 00H IF DST[63..32] > SRC[63..32] THEN DST[63..32] ← 0FFFFFFFFH ELSE DST [63..32] ← 00H IF DST[95..64] > SRC[95..64] THEN DST[95..64] ← 0FFFFFFFFH ELSE DST [95..64] ← 00H IF DST[127..96] > SRC[127..96] THEN DST[127..96] ← 0FFFFFFFFH ELSE DST [127..96] ← 00H

MMX/SSE Logical Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Parallel AND	PAND	mm1, mm2/m64	Bitwise AND mm2/m64 and mm1. Store result in mm1.	DST ← DST AND SRC
		xmm1, xmm2/m128	Bitwise AND xmm2/m128 and xmm1. Store result in xmm1.	DST ← DST AND SRC
Parallel AND NOT	PANDN	mm1, mm2/m64	Bitwise AND NOT mm2/m64 and mm1. Store result in mm1.	DST ← DST AND NOT SRC
		xmm1, xmm2/m128	Bitwise AND NOT xmm2/m128 and xmm1. Store result in xmm1.	DST ← DST AND NOT SRC
Parallel OR	POR	mm1, mm2/m64	Bitwise OR mm2/m64 and mm1. Store result in mm1.	DST ← DST OR SRC
		xmm1, xmm2/m128	Bitwise OR xmm2/m128 and xmm1. Store result in xmm1.	DST ← DST OR SRC
Parallel XOR	PXOR	mm1, mm2/m64	Bitwise XOR mm2/m64 and mm1. Store result in mm1.	DST ← DST XOR SRC
		xmm1, xmm2/m128	Bitwise XOR xmm2/m128 and xmm1. Store result in xmm1.	DST ← DST XOR SRC

MMX/SSE Shift and Rotate Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Packed Shift Left Logical Words	PSLLW	mm1, mm2/m64 mm1, imm8	Shift words in mm1 left by the specified position count clearing low-order bits.	DST[15..0] ← ZeroExtend(DST[15..0] SHL Count) DST[31..16] ← ZeroExtend(DST[31..15] SHL Count) DST[47..32] ← ZeroExtend(DST[47..32] SHL Count) DST[63..48] ← ZeroExtend(DST[63..48] SHL Count)

		xmm1, xmm2/m128 xmm1, imm8	Shift words in xmm1 left by the specified position count clearing low-order bits.	DST[15..0] ← ZeroExtend(DST[15..0] SHL Count) DST[31..16] ← ZeroExtend(DST[31..15] SHL Count) DST[47..32] ← ZeroExtend(DST[47..32] SHL Count) DST[63..48] ← ZeroExtend(DST[63..48] SHL Count) DST[79..64] ← ZeroExtend(DST[79..64] SHL Count) DST[95..80] ← ZeroExtend(DST[95..80] SHL Count) DST[111..96] ← ZeroExtend(DST[111..96] SHL Count) DST[127..112] ← ZeroExtend(DST[127..112] SHL Count)
Packed Shift Left Logical Dwords	PSLLD	mm1, mm2/m64 mm1, imm8	Shift double words in mm1 left by the specified position count clearing low-order bits.	DST[31..0] ← ZeroExtend(DST[31..0] SHL Count) DST[63..32] ← ZeroExtend(DST[63..32] SHL Count)
		xmm1, xmm2/m128 xmm1, imm8	Shift double words in xmm1 left by the specified position count clearing low-order bits.	DST[31..0] ← ZeroExtend(DST[31..0] SHL Count) DST[63..32] ← ZeroExtend(DST[63..32] SHL Count) DST[95..64] ← ZeroExtend(DST[95..64] SHL Count) DST[127..96] ← ZeroExtend(DST[127..96] SHL Count)
Packed Shift Left Logical Qwords	PSLLQ	mm1, mm2/m64 mm1, imm8	Shift quad word in mm1 left by the specified position count clearing low-order bits.	DST[63..0] ← ZeroExtend(DST[63..0] SHL Count)
		xmm1, xmm2/m128 xmm1, imm8	Shift quad words in xmm1 left by the specified position count clearing low-order bits.	DST[63..0] ← ZeroExtend(DST[63..0] SHL Count) DST[127..64] ← ZeroExtend(DST[127..64] SHL Count)
Packed Shift Right Logical Words	PSRLW	mm1, mm2/m64 mm1, imm8	Shift words in mm1 right by the specified position count clearing high-order bits.	DST[15..0] ← ZeroExtend(DST[15..0] SHR Count) DST[31..16] ← ZeroExtend(DST[31..15] SHR Count) DST[47..32] ← ZeroExtend(DST[47..32] SHR Count) DST[63..48] ← ZeroExtend(DST[63..48] SHR Count)

		xmm1, xmm2/m128 xmm1, imm8	Shift words in xmm1 right by the specified position count clearing high-order bits.	DST[15..0] ← ZeroExtend(DST[15..0] SHR Count) DST[31..16] ← ZeroExtend(DST[31..15] SHR Count) DST[47..32] ← ZeroExtend(DST[47..32] SHR Count) DST[63..48] ← ZeroExtend(DST[63..48] SHR Count) DST[79..64] ← ZeroExtend(DST[79..64] SHR Count) DST[95..80] ← ZeroExtend(DST[95..80] SHR Count) DST[111..96] ← ZeroExtend(DST[111..96] SHR Count) DST[127..112] ← ZeroExtend(DST[127..112] SHR Count)
Packed Shift Right Logical Dwords	PSRLD	mm1, mm2/m64 mm1, imm8	Shift double words in mm1 right by the specified position count clearing high-order bits.	DST[31..0] ← ZeroExtend(DST[31..0] SHR Count) DST[63..32] ← ZeroExtend(DST[63..32] SHR Count)
		xmm1, xmm2/m128 xmm1, imm8	Shift double words in xmm1 right by the specified position count clearing high-order bits.	DST[31..0] ← ZeroExtend(DST[31..0] SHR Count) DST[63..32] ← ZeroExtend(DST[63..32] SHR Count) DST[95..64] ← ZeroExtend(DST[95..64] SHR Count) DST[127..96] ← ZeroExtend(DST[127..96] SHR Count)
Packed Shift Right Logical Qwords	PSRLQ	mm1, mm2/m64 mm1, imm8	Shift quad word in mm1 right by the specified position count clearing high-order bits.	DST[63..0] ← ZeroExtend(DST[63..0] SHR Count)
		xmm1, xmm2/m128 xmm1, imm8	Shift quad words in xmm1 right by the specified position count clearing high-order bits.	DST[63..0] ← ZeroExtend(DST[63..0] SHR Count) DST[127..64] ← ZeroExtend(DST[127..64] SHR Count)
Packed Shift Right Arithmetical Words	PSRAW	mm1, mm2/m64 mm1, imm8	Shift words in mm1 right by the specified position count duplicating sign.	DST[15..0] ← SignExtend(DST[15..0] SHR Count) DST[31..16] ← SignExtend(DST[31..15] SHR Count) DST[47..32] ← SignExtend(DST[47..32] SHR Count) DST[63..48] ← SignExtend(DST[63..48] SHR Count)

		xmm1, xmm2/m128 xmm1, imm8	Shift words in xmm1 right by the specified position count duplicating sign bits.	DST[15..0] ← SignExtend(DST[15..0] SHR Count) DST[31..16] ← SignExtend(DST[31..15] SHR Count) DST[47..32] ← SignExtend(DST[47..32] SHR Count) DST[63..48] ← SignExtend(DST[63..48] SHR Count) DST[79..64] ← SignExtend(DST[79..64] SHR Count) DST[95..80] ← SignExtend(DST[95..80] SHR Count) DST[111..96] ← SignExtend(DST[111..96] SHR Count) DST[127..112] ← SignExtend(DST[127..112] SHR Count)
Packed Shift Right Arithmetical Dwords	PSRAD	mm1, mm2/m64 mm1, imm8	Shift double words in mm1 right by the specified position count duplicating sign bits.	DST[31..0] ← SignExtend(DST[31..0] SHR Count) DST[63..32] ← SignExtend(DST[63..32] SHR Count)
		xmm1, xmm2/m128 xmm1, imm8	Shift double words in xmm1 right by the specified position count duplicating sign bits.	DST[31..0] ← SignExtend(DST[31..0] SHR Count) DST[63..32] ← SignExtend(DST[63..32] SHR Count) DST[95..64] ← SignExtend(DST[95..64] SHR Count) DST[127..96] ← SignExtend(DST[127..96] SHR Count)

MMX State Management Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Empty MMX State	EMMS		Sets the x87 FPU tag word to empty	x87FPUTagWord ← FFFFH

SSE Instruction Set

SSE Data Transfer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Move Aligned Packed Singles	MOVAPS	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves packed single-precision floating-point values from source to destination operand. When the source or destination operand is a memory location, it must be aligned on a 16-byte boundary.	DST ← SRC
Move Unaligned Packed Singles	MOVUPS	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves packed single-precision floating-point values from source to destination operand. When the source or destination operand is a memory location, it may be not aligned on a 16-byte boundary.	DST ← SRC
Move High Packed Singles	MOVHPS	xmm, m64	Move two packed single-precision floating-point values from source to high quad word of destination operand.	DST[127..64] ← SRC *// DST[63..0] remains unchanged*
		m64, xmm		DST ← SRC[127..64]
Move High to Low Packed Singles	MOVHLPS	xmm1, xmm2	Moves two packed single-precision floating-point values from high quad word of xmm2 to low quad word of xmm1.	DST[63..0] ← SRC[127..64] *// DST[127..64] remains unchanged*
Move Low Packed Singles	MOVLPS	xmm, m64	Move two packed single-precision floating-point values from source to low quad word of destination operand.	DST[63...0] ← SRC *// DST[127..64] remains unchanged*
		m64, xmm		DST ← SRC[63..0]

Move Low to High Packed Singles	MOVLHPS	xmm1, xmm2	Moves two packed single-precision floating-point values from low quad word of xmm2 to high quad word of xmm1.	DST[127..64] ← SRC[63..0] *// DST[63..0] remains unchanged*
Extract Packed Singles Sign Mask	MOVMSKPS	r32, xmm	Extracts 4-bit sign mask of from xmm and stores in r32.	DST[0] ← SRC[31] DST[1] ← SRC[63] DST[2] ← SRC[95] DST[3] ← SRC[127] DST[31..4] ← 000000H
Move Scalar Single	MOVSS	xmm, m128	Moves scalar single-precision floating-point value from source to destination operand.	DST[31..0] ← SRC[31..0] DST[127..32] ← 000000000000000000000000H
		m128, xmm		DST[31..0] ← SRC[31..0]
		xmm1, xmm2		DST[31..0] ← SRC[31..0] *//DST[127..32] remains unchanged*

SSE Packed Arithmetic Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Add Packed Singles	ADDPS	xmm1, xmm2/m128	Adds 4 packed single-precision floating-point values from *xmm2/m128* to 4 packed single-precision floating-point values in *xmm1*.	DST[31..0] ← DST[31..0] + SRC[31-0]; DST[63..32] ← DST[63..32] + SRC[63..32]; DST[95..64] ← DST[95..64] + SRC[95..64]; DST[127..96] ← DST[127..96] + SRC[127..96];
Add Scalar Singles	ADDSS	xmm1, xmm2/m32	Adds the low single-precision floating-point value from *xmm2/m32* to the low single-precision floating-point value in *xmm1*.	DST[31..0] ← DST[31..0] + SRC[31..0]; *// DST[127..32] remains unchanged*
Subtract Packed Singles	SUBPS	xmm1, xmm2/m128	Subtracts 4 packed single-precision floating-point values from *xmm2/m128* from 4 packed single-precision floating-point values in *xmm1*.	DST[31..0] ← DST[31..0] – SRC[31-0]; DST[63..32] ← DST[63..32] – SRC[63..32]; DST[95..64] ← DST[95..64] – SRC[95..64]; DST[127..96] ← DST[127..96] – SRC[127..96];

Subtract Scalar Singles	SUBSS	xmm1, xmm2/m32	Subtracts the low single-precision floating-point value from *xmm2/m32* from the low single-precision floating-point value in *xmm1*.	DST[31..0] ← DST[31..0] – SRC[31..0]; *// DST[127..32] remains unchanged*
Multiply Packed Singles	MULPS	xmm1, xmm2/m128	Multiplies 4 packed single-precision floating-point values *xmm1* by 4 packed single-precision floating-point values in *xmm2/m128*.	DST[31..0] ← DST[31..0] * SRC[31-0]; DST[63..32] ← DST[63..32] * SRC[63..32]; DST[95..64] ← DST[95..64] * SRC[95..64]; DST[127..96] ← DST[127..96] * SRC[127..96];
Multiply Scalar Singles	MULSS	xmm1, xmm2/m32	Multiplies the low single-precision floating-point value from xmm1 by the low single-precision floating-point value in *xmm2/m32*.	DST[31..0] ← DST[31..0] * SRC[31..0]; *// DST[127..32] remains unchanged*
Divide Packed Singles	DIVPS	xmm1, xmm2/m128	Divides 4 packed single-precision floating-point values in *xmm1* by 4 packed single-precision floating-point values in *xmm2/m128*.	DST[31..0] ← DST[31..0] / SRC[31..0]; DST[63..32] ← DST[63..32] / SRC[63..32]; DST[95..64] ← DST[95..64] / SRC[95..64]; DST[127..96] ← DST[127..96] / SRC[127..96];
Divide Scalar Singles	DIVSS	xmm1, xmm2/m32	Divides low single-precision floating-point value in *xmm1* by the low single-precision floating-point value in *xmm2/m64*.	DST[31..0] ← DST[31..0] / SRC[31..0]; *// DST[127..32] remains unchanged*

Reciprocals of Packed Singles	RCPPS	xmm1, xmm2/m128	Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.	DST[31..0] ← Approximate (1.0 / SRC[31..0]); DST[63..32] ← Approximate (1.0 / SRC[63..32]); DST[95..64] ← Approximate (1.0 / SRC[95..64]); DST[127..96] ← Approximate (1.0 / SRC[127..96]);
Reciprocals of Scalar Single	RCPSS	xmm1, xmm2/m32	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m128* and stores the result in *xmm1*.	DST[31..0] ← Approximate (1.0 / SRC[31..0]); *// DST[127..32] remains unchanged*
Square Roots of Packed Singles	SQRTPS	xmm1, xmm2/m128	Computes the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.	DST[31..0] ← SquareRoot (SRC[31..0]); DST[63..32] ← SquareRoot (SRC[63..32]); DST[95..64] ← SquareRoot (SRC[95..64]); DST[127..96] ← SquareRoot (SRC[127..96]);
Square Root of Scalar Single	SQRTSS	xmm1, xmm2/m32	Computes the square root of the scalar single-precision floating-point value in *xmm2/m128* and stores the result in *xmm1*.	DST[31..0] ← SquareRoot (SRC[31..0]); *// DST[127..32] remains unchanged*
Reciprocals of Square Roots of Packed Singles	RSQRTPS	xmm1, xmm2/m128	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.	DST[31..0] ← Approximate (1.0 / SquareRoot (SRC[31..0])); DST[63..32] ← Approximate (1.0 / SquareRoot (SRC[63..32])); DST[95..64] ← Approximate (1.0 / SquareRoot (SRC[95..64])); DST[127..96] ← Approximate (1.0 / SquareRoot (SRC[127..96]));
Reciprocals of Square Roots of Scalar Single	RSQRTSS	xmm1, xmm2/m32	Computes the approximate reciprocal of the square root of the scalar single-precision floating-point value in *xmm2/m128* and stores the result in *xmm1*.	DST[31..0] ← Approximate (1.0 / SquareRoot (SRC[31..0])); *// DST[127..32] remains unchanged*

Maximum Packed Single	MAXPS	xmm1, xmm2/m128	Returns the maximum single-precision floating-point values between *xmm2/m128* and *xmm1*.	DST[31..0] ← MaximumOf (DST[31..0], SRC[31..0]); DST[63..32] ← MaximumOf (DST[63..32], SRC[63..32]); DST[95..64] ← Maximum (DST[95..64], SRC[95..64]); DST[127..96] ← MaximumOf (DST[127..96], SRC[127..96]);
Maximum Scalar Single	MAXSS	xmm1, xmm2/m32	Returns the maximum scalar single-precision floating-point value between *xmm2/m128* and *xmm1*.	DST[31..0] ← MaximumOf (DST[31..0], SRC[31..0]); *// DST[127..32] remains unchanged*
Minimum Packed Single	MINPS	xmm1, xmm2/m128	Returns the minimum single-precision floating-point values between *xmm2/m128* and *xmm1*.	DST[31..0] ← MinimumOf (DST[31..0], SRC[31..0]); DST[63..32] ← MinimumOf (DST[63..32], SRC[63..32]); DST[95..64] ← Minimum (DST[95..64], SRC[95..64]); DST[127..96] ← MinimumOf (DST[127..96], SRC[127..96]);
Minimum Scalar Single	MINSS	xmm1, xmm2/m32	Returns the minimum scalar single-precision floating-point value between *xmm2/m128* and *xmm1*.	DST[31..0] ← MinimumOf (DST[31..0], SRC[31..0]); *// DST[127..32] remains unchanged*

SSE Comparison Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Compare Packed Singles	CMPPS	xmm1, xmm2/m128, imm8	Compares 4 packed double-precision floating-point values in *xmm2/m128* and *xmm1* using *imm8* as comparison predicate: 0 – equal, 1 – less than, 2 – less or equal, 3 – unordered, 4 – not equal, 5 – not less, 6 – not less or equal, 7 – ordered. The result of each comparison in a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The unordered relationship is true when at leas one of the two operands is a NAN; the ordered relationship id true when neither operand is a NAN.	CMP0 ← DST[31..0] OP SRC[31..0]; CMP1 ← DST[63..32] OP SRC[63..32]; CMP2 ← DST[95..64] OP SRC[95..64]; CMP3 ← DST[127..96] OP SRC[127..96]; IF CMP0 THEN DST[31..0] ← FFFFFFFFH ELSE DST[31..0] ← 00000000H; IF CMP1 THEN DST[63..32] ← FFFFFFFFH ELSE DST[63..32] ← 00000000H; IF CMP2 THEN DST[95..64] ← FFFFFFFFH ELSE DST[95..64] ← 00000000H; IF CMP3 THEN DST[127..96] ← FFFFFFFFH ELSE DST[127..96] ← 00000000H
Compare Packed Singles	CMPEQPS CMPLTPS CMPLEPS CMPUNORDPS CMPNEQPS CMPNLTPS CMPNLEPS CMPORDPS	xmm1, xmm2	<=> CMPPS xmm1,xmm2, 0 <=> CMPPS xmm1,xmm2, 1 <=> CMPPS xmm1,xmm2, 2 <=> CMPPS xmm1,xmm2, 3 <=> CMPPS xmm1,xmm2, 4 <=> CMPPS xmm1,xmm2, 5 <=> CMPPS xmm1,xmm2, 6 <=> CMPPS xmm1,xmm2, 7	see CMPPS
Compare Scalar Singles	CMPSS	xmm1, xmm2/m32, imm8	Compares the low double-precision floating-point values in *xmm2/m128* and *xmm1* using *imm8* as comparison predicate: 0 – equal, 1 – less than, 2 – less or equal, 3 – unordered, 4 – not equal, 5 – not less, 6 – not less or equal, 7 – ordered. The result of each comparison in a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The unordered relationship is true when at leas one of the two operands is a NAN; the ordered relationship id true when neither operand is a NAN.	CMP0 ← DST[31..0] OP SRC[31..0]; IF CMP0 THEN DST[31..0] ← FFFFFFFFH ELSE DST[31..0] ← 00000000H; *// DST[127..32] remains unchanged*
Compare Scalar Singles	CMPEQSS CMPLTSS CMPLESS CMPUNORDSS CMPNEQSS CMPNLTSS CMPNLESS CMPORDSS	xmm1, xmm2	<=> CMPSS xmm1,xmm2, 0 <=> CMPSS xmm1,xmm2, 1 <=> CMPSS xmm1,xmm2, 2 <=> CMPSS xmm1,xmm2, 3 <=> CMPSS xmm1,xmm2, 4 <=> CMPSS xmm1,xmm2, 5 <=> CMPSS xmm1,xmm2, 6 <=> CMPSS xmm1,xmm2, 7	see CMPSS

Compare Scalar Singles and set EFLAGS	COMISS	xmm1, xmm2/m32	Compares the low single-precision floating-point values in the operands and sets the EFLAGS flags accordingly. Performs ordered compare. This instruction differs from the UCOMISS instruction in that is signals an invalid operation exception when a source operand is a QNan or and SNaN.	Result ← OrderedCompare(DST[31..0], SRC[31..0]) CASE (Result) OF UNORDERED: ZF, PF, CF ← 111; GREATER_THAN: ZF, PF, CF ← 000; LESS_THAN: ZF, PF, CF ← 001; EQUAL: ZF, PF, CF ← 100; END OF, AF, SF ← 0;
Unordered Compare Scalar Singles and set EFLAGS	UCOMISS	xmm1, xmm2/m32	Compares the low single-precision floating-point value in the operands and sets the EFLAGS flags accordingly. Performs unordered compare. This instruction differs from the COMISS instruction in that is signals an invalid operation exception only when a source operand is a SNaN.	Result ← UnorderedCompare(DST[31..0], SRC[31..0]) CASE (Result) OF UNORDERED: ZF, PF, CF ← 111; GREATER_THAN: ZF, PF, CF ← 000; LESS_THAN: ZF, PF, CF ← 001; EQUAL: ZF, PF, CF ← 100; END OF, AF, SF ← 0;

SSE Logical Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
AND of Packed Singles	ANDPS	xmm1, xmm2/m128	Performs a bitwise AND operation of the four packed single-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand.	DST[31..0] ← DST[31..0] AND SRC[31-0]; DST[63..32] ← DST[63..32] AND SRC[63..32]; DST[95..64] ← DST[95..64] AND SRC[95..64]; DST[127..96] ← DST[127..96] AND SRC[127..96];

AND NOT of Packed Singles	ANDNPS	xmm1, xmm2/m128	Inverts the bits of the four packed single-precision floating-point values in the destination (first) operand, performs a bitwise logical AND operation of the four packed single-precision floating-point values from the temporary inverted result and source (second) operand and stored the result in the destination operand. DST 127 / SRC 127 / ANDN ANDN ANDN ANDN / DST 127	DST[31..0] ← (NOT DST[31..0]) AND SRC[31-0]; DST[63..32] ← (NOT DST[63..32]) AND SRC[63..32]; DST[95..64] ← (NOT DST[95..64]) AND SRC[95..64]; DST[127..96] ← (NOT DST[127..96]) AND SRC[127..96];
OR of Packed Singles	ORPS	xmm1, xmm2/m128	Performs a bitwise OR operation of the four packed single-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand. DST 127 / SRC 127 / OR OR OR OR / DST 127	DST[31..0] ← DST[31..0] OR SRC[31-0]; DST[63..32] ← DST[63..32] OR SRC[63..32]; DST[95..64] ← DST[95..64] OR SRC[95..64]; DST[127..96] ← DST[127..96] OR SRC[127..96];
Exclusive OR of Packed Singles	XORPS	xmm1, xmm2/m128	Performs a bitwise XOR operation of the four packed single-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand. DST 127: X3 X2 X1 X0 / SRC 127: Y3 Y2 Y1 Y0 / XOR XOR XOR XOR / DST 127: X3 XOR Y3, X2 XOR Y2, X1 XOR Y1, X0 XOR Y0	DST[31..0] ← DST[31..0] XOR SRC[31-0]; DST[63..32] ← DST[63..32] XOR SRC[63..32]; DST[95..64] ← DST[95..64] XOR SRC[95..64]; DST[127..96] ← DST[127..96] XOR SRC[127..96];

SSE Shuffle and Unpack Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Shuffle Packed Singles	SHUFPS	xmm1, xmm2/m128, imm8	Moves two of the four packed single-precision floating-point values from the destination (first) operand into the low quad word of the destination operand; move two of the four packed single-precision floating-point values from the source (second) operand into the high quad word of the destination operand. The select (third) operand determines which values are moved to the destination operand.	CASE (SEL[1..0]) OF 0: DST[31..0] ← DST[31..0] 1: DST[31..0] ← DST[63..32] 2: DST[31..0] ← DST[95..64] 3: DST[31..0] ← DST[127..96] END CASE (SEL[3..2]) OF 0: DST[63..32] ← DST[31..0] 1: DST[63..32] ← DST[63..32] 2: DST[63..32] ← DST[95..64] 3: DST[63..32] ← DST[127..96] END CASE (SEL[5..4]) OF 0: DST[95..64] ← SRC[31..0] 1: DST[95..64] ← SRC[63..32] 2: DST[95..64] ← SRC[95..64] 3: DST[95..64] ← SRC[127..96] END CASE (SEL[7..6) OF 0: DST[127..96] ← SRC[31..0] 1: DST[127..96] ← SRC[63..32] 2: DST[127..96] ← SRC[95..64] 3: DST[127..96] ← SRC[127..96] END
Unpack Low Packed Singles	UNPCKLPS	xmm1, xmm2/m128	Unpacks and interleaves the low single-precision floating-point values from the low quad words of the source (second) operand and the destination (first) operand.	DST[31..0] ← DST[31..0] DST[63..32] ← SRC[31..0] DST[95..64] ← DST[63..32] DST[127..96] ← SRC[63..32]
Unpack High Packed Singles	UNPCKHPS	xmm1, xmm2/m128	Unpacks and interleaves the low single-precision floating-point values from the high quad words of the source (second) operand and the destination (first) operand.	DST[31..0] ← DST[95..64] DST[63..32] ← SRC[95..64] DST[95..64] ← DST[127..96] DST[127..96] ← SRC[127..96]

SSE Conversion Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Convert Packed Integers to Packed Singles	CVTPI2PS	xmm, mm/m64	Converts two packed signed double word integers from *mm/mem64* to two packed single-precision floating-point values from *xmm*.	DST[31..0] ← IntToSingle (SRC[31..0]); DST[63..32] ← IntToSingle (SRC[63..32]); *// DST[127..64] remains unchanged*
Convert Packed Singles to Packed Integers	CVTPS2PI	mm, xmm/m64	Converts two packed single-precision floating-point values from *xmm/m64* to two packed signed double-word integers in *mm*.	DST[31..0] ← SingleToInt (SRC[31..0]); DST[63..32] ← SingleToInt (SRC[63..32]);
Convert Scalar Integer to Scalar Single	CVTSI2SS	xmm, r/m32	Converts one signed double-word integer from *r/m32* to one scalar single-precision floating-point value in *xmm*..	DST[31..0]← IntToSingle (SRC); *// DST[127..32] remains unchanged*
Convert Scalar Single Scalar Integer	CVTSS2SI	r32, xmm/m32	Converts a single-precision floating-point value from *xmm/m32* to a signed double-word integer in *r32*.	DST← SingleToInt (SRC[31..0]);
Convert with Truncation Packed Singles to Packed Integers	CVTTPS2PI	mm, xmm/m64	Converts two packed single-precision floating-point values from *xmm/m64* to two packed signed double-word integers in *mm* using truncation.	DST[31..0] ← TruncateSingleToInt (SRC[31..0]); DST[63..32] ← TruncateSingleToInt (SRC[63..32]); *// DST[127..64] remains unchanged*
Convert with Truncation Scalar Single to Scalar Integer	CVTTSS2SI	r32, xmm/m32	Converts a single-precision floating-point value from *xmm/m32* to a signed double-word integer in *r32* using truncation.	DST← TruncateSingleToInt (SRC[31..0]);

SSE 64-Bit SIMD Integer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Packed Average Bytes	PAVGB	mm1, mm2/m64	Averages 8 packed unsigned bytes from mm1 and 8 packed unsigned bytes from mm2/m64 with rounding.	DST[7..0] ← (DST[7..0]+SRC[7..0]+1) SHR 1 DST[15..8] ← (DST[15..8]+SRC[15..8]+1) SHR 1 DST[23..16] ← (DST[23..16]+SRC[23..16]+1) SHR 1 DST[31..24] ← (DST[31..24]+SRC[31..24]+1) SHR 1 DST[39..32] ← (DST[39..32]+SRC[39..32]+1) SHR 1 DST[47..40] ← (DST[47..40]+SRC[47..40]+1) SHR 1 DST[55..48] ← (DST[55..48]+SRC[55..48]+1) SHR 1 DST[63..56] ← (DST[63..56]+SRC[63..56]+1) SHR 1
		xmm1,xmm2/m128	Averages 16 packed unsigned bytes from xmm1 and 16 packed unsigned bytes from xmm2/m128 with rounding.	DST[7..0] ← (DST[7..0]+SRC[7..0]+1) SHR 1 DST[15..8] ← (DST[15..8]+SRC[15..8]+1) SHR 1 DST[23..16] ← (DST[23..16]+SRC[23..16]+1) SHR 1 DST[31..24] ← (DST[31..24]+SRC[31..24]+1) SHR 1 DST[39..32] ← (DST[39..32]+SRC[39..32]+1) SHR 1 DST[47..40] ← (DST[47..40]+SRC[47..40]+1) SHR 1 DST[55..48] ← (DST[55..48]+SRC[55..48]+1) SHR 1 DST[63..56] ← (DST[63..56]+SRC[63..56]+1) SHR 1 DST[71..64] ← (DST[71..64]+SRC[71..64]+1) SHR 1 DST[79..72] ← (DST[79..72]+SRC[79..72]+1) SHR 1 DST[87..80] ← (DST[87..80]+SRC[87..80]+1) SHR 1 DST[95..88] ← (DST[95..88]+SRC[95..88]+1) SHR 1 DST[103..96] ← (DST[103..96]+SRC[103..96]+1) SHR 1 DST[111..104] ← (DST[111..104]+SRC[111..104]+1) SHR 1 DST[119..112] ← (DST[119..112]+SRC[119..112]+1) SHR 1 DST[127..120] ← (DST[127..120]+SRC[127..120]+1) SHR 1
Packed Average Words	PAVGW	mm1, mm2/m64	Averages 4 packed unsigned words from mm1 and 4 packed unsigned words from mm2/m64 with rounding.	DST[15..0] ← (DST[15..0]+SRC[15..0]+1) SHR 1 DST[31..16] ← (DST[31..16]+SRC[31..16]+1) SHR 1 DST[47..32] ← (DST[47..32]+SRC[47..32]+1) SHR 1 DST[63..56] ← (DST[63..48]+SRC[63..48]+1) SHR 1
		xmm1,xmm2/m128	Averages 8 packed unsigned words from xmm1 and 8 packed unsigned words from xmm2/m128 with rounding.	DST[15..0] ← (DST[15..0]+SRC[15..0]+1) SHR 1 DST[31..16] ← (DST[31..16]+SRC[31..16]+1) SHR 1 DST[47..32] ← (DST[47..32]+SRC[47..32]+1) SHR 1 DST[63..56] ← (DST[63..48]+SRC[63..48]+1) SHR 1 DST[79..64] ← (DST[79..64]+SRC[79..64]+1) SHR 1 DST[95..80] ← (DST[95..80]+SRC[95..80]+1) SHR 1 DST[111..96] ← (DST[111..96]+SRC[111..96]+1) SHR 1 DST[127..112] ← (DST[127..112]+SRC[127..112]+1) SHR 1

Packed Maximum of Words	PMAXSW	mm1, mm2/m64	Compares 4 signed word integers in mm1 with 4 signed word integers in mm2/m64 and returns maximum values.	DST[15..0] ← MaximumOf (DST[15..0], SRC[15..0]) DST[31..16] ← MaximumOf (DST[31..16], SRC[31..16]) DST[47..32] ← MaximumOf (DST[47..32], SRC[47..32]) DST[63..48] ← MaximumOf (DST[63..48], SRC[63..48])
		xmm1, xmm2/m128	Compares 8 signed word integers in xmm1 with 8 signed word integers in xmm2/m128 and returns maximum values.	DST[15..0] ← MaximumOf (DST[15..0], SRC[15..0]) DST[31..16] ← MaximumOf (DST[31..16], SRC[31..16]) DST[47..32] ← MaximumOf (DST[47..32], SRC[47..32]) DST[63..48] ← MaximumOf (DST[63..48], SRC[63..48]) DST[71..64] ← MaximumOf (DST[71..64], SRC[71..64]) DST[95..80] ← MaximumOf (DST[95..80], SRC[95..80]) DST[111..96] ← MaximumOf (DST[111..96], SRC[111..96]) DST[127..112] ← MaximumOf (DST[127..112], SRC[127..112])
Packed Maximum of Unsigned Bytes	PMAXUB	mm1, mm2/m64	Compares 8 unsigned byte integers in xmm1 with 8 unsigned byte integers in xmm2/m128 and returns maximum values.	DST[7..0] ← MaximumOf (DST[7..0], SRC[7..0]) DST[15..8] ← MaximumOf (DST[15..8], SRC[15..8]) DST[23..16] ← MaximumOf (DST[23..16], SRC[23..16]) DST[31..24] ← MaximumOf (DST[31..24], SRC[31..24]) DST[39..32] ← MaximumOf (DST[39..32], SRC[39..32]) DST[47..40] ← MaximumOf (DST[47..40], SRC[47..40]) DST[55..48] ← MaximumOf (DST[55..48], SRC[55..48]) DST[63..56] ← MaximumOf (DST[63..56], SRC[63..56])
Packed Minimum of Words	PMAXSW	mm1, mm2/m64	Compares 4 signed word integers in mm1 with 4 signed word integers in mm2/m64 and returns minimum values.	DST[15..0] ← MinimumOf (DST[15..0], SRC[15..0]) DST[31..16] ← MinimumOf (DST[31..16], SRC[31..16]) DST[47..32] ← MinimumOf (DST[47..32], SRC[47..32]) DST[63..48] ← MinimumOf (DST[63..48], SRC[63..48])
		xmm1, xmm2/m128	Compares 8 signed word integers in xmm1 with 8 signed word integers in xmm2/m128 and returns minimum values.	DST[15..0] ← MinimumOf (DST[15..0], SRC[15..0]) DST[31..16] ← MinimumOf (DST[31..16], SRC[31..16]) DST[47..32] ← MinimumOf (DST[47..32], SRC[47..32]) DST[63..48] ← MinimumOf (DST[63..48], SRC[63..48]) DST[71..64] ← MinimumOf (DST[71..64], SRC[71..64]) DST[95..80] ← MinimumOf (DST[95..80], SRC[95..80]) DST[111..96] ← MinimumOf (DST[111..96], SRC[111..96]) DST[127..112] ← MinimumOf (DST[127..112], SRC[127..112])

Packed Minimum of Unsigned Bytes	PMAXUB	mm1, mm2/m64	Compares 8 unsigned byte integers in xmm1 with 8 unsigned byte integers in xmm2/m128 and returns minimum values.	DST[7..0] ← MinimumOf (DST[7..0], SRC[7..0]) DST[15..8] ← MinimumOf (DST[15..8], SRC[15..8]) DST[23..16] ← MinimumOf (DST[23..16], SRC[23..16]) DST[31..24] ← MinimumOf (DST[31..24], SRC[31..24]) DST[39..32] ← MinimumOf (DST[39..32], SRC[39..32]) DST[47..40] ← MinimumOf (DST[47..40], SRC[47..40]) DST[55..48] ← MinimumOf (DST[55..48], SRC[55..48]) DST[63..56] ← MinimumOf (DST[63..56], SRC[63..56])
Move Byte Mask To Integer	PMOVMSKB	r32, mm	Creates a mask made up of the most significant bit of each byte of the mmx register and stored the result in the low byte r32 register.	DST[0] ← SRC[7] DST[1] ← SRC[15] DST[2] ← SRC[23] DST[3] ← SRC[31] DST[4] ← SRC[39] DST[5] ← SRC[47] DST[6] ← SRC[55] DST[7] ← SRC[63] DST[31..8] ← 000000H
		r32, xmm	Creates a mask made up of the most significant bit of each byte of the xmm register and stored the result in the low word of xmm register.	DST[0] ← SRC[7] DST[1] ← SRC[15] DST[2] ← SRC[23] DST[3] ← SRC[31] DST[4] ← SRC[39] DST[5] ← SRC[47] DST[6] ← SRC[55] DST[7] ← SRC[63] DST[8] ← SRC[71] DST[9] ← SRC[79] DST[10] ← SRC[87] DST[11] ← SRC[95] DST[12] ← SRC[103] DST[13] ← SRC[111] DST[14] ← SRC[119] DST[15] ← SRC[127] DST[31..16] ← 0000H
Packed Sum of Absolute Differences	PSADBW	mm1, mm2/m64	Computes the absolute differences of the packed unsigned byte integers from mm1 and mm2/m64; differences are then summed to produce an unsigned word integer result.	TMP0 ← ABS(DST[7..0]−SRC[7..0]) TMP1 ← ABS(DST[15..8]−SRC[15..8]) TMP2 ← ABS(DST[23..16]−SRC[23..16]) TMP3 ← ABS(DST[31..24]−SRC[31..24]) TMP4 ← ABS(DST[39..32]−SRC[39..32]) TMP5 ← ABS(DST[47..40]−SRC[47..40]) TMP6 ← ABS(DST[55..48]−SRC[55..48]) TMP7 ← ABS(DST[63..56]−SRC[63..56]) DST[15..0] ← SUM(TMP0..TMP7) DST[63..16] ← 000000000000H

		xmm1, xmm2/m128	Computes the absolute differences of the packed unsigned byte integers from xmm1 and xmm2/m128; the 8 low differences and the high 8 differences are then summed separately to produce an two unsigned word integer results.	TMP0 ← ABS(DST[7..0]−SRC[7..0]) TMP1 ← ABS(DST[15..8]−SRC[15..8]) TMP2 ← ABS(DST[23..16]−SRC[23..16]) TMP3 ← ABS(DST[31..24]−SRC[31..24]) TMP4 ← ABS(DST[39..32]−SRC[39..32]) TMP5 ← ABS(DST[47..40]−SRC[47..40]) TMP6 ← ABS(DST[55..48]−SRC[55..48]) TMP7 ← ABS(DST[63..56]−SRC[63..56]) TMP8 ← ABS(DST[71..64]−SRC[71..64]) TMP9 ← ABS(DST[79..72]−SRC[79..72]) TMP10 ← ABS(DST[87..80]−SRC[87..80]) TMP11 ← ABS(DST[95..88]−SRC[95..88]) TMP12 ← ABS(DST[103..96]−SRC[103..96]) TMP13 ← ABS(DST[111..104]−SRC[111..104]) TMP14 ← ABS(DST[119..112]−SRC[119..112]) TMP15 ← ABS(DST[127..120]−SRC[127..120]) DST[15..0] ← SUM(TMP0..TMP7) DST[63..16] ← 000000000000H DST[79..64] ← SUM(TMP8..TMP15) DST[127..80] ← 000000000000H
Packed Extract Word	PEXTRW	r32, mm, imm8	Extracts the word specified by imm8 from the mmx register and moves it to the r32 register.	SEL ← Count AND 3H TMP ← (SRC SHR (SEL * 16)) AND 0FFFFH DST[15..0] ← TMP[15..0] DST[31..16] ← 0000H
		r32, xmm, imm8	Extracts the word specified by imm8 from the xmm register and moves it to the r32 register.	SEL ← Count AND 7H TMP ← (SRC SHR (SEL * 16)) AND 0FFFFH DST[15..0] ← TMP[15..0] DST[31..16] ← 0000H
Packed Insert Word	PINSRW	mm, r32/m16, imm8	Inserts the low word from the r32 register or memory into the mmx register at the word position specified by imm8.	SEL ← Count AND 3H CASE (SEL) OF 0: MASK ← 000000000000FFFFH 1: MASK ← 00000000FFFF0000H 2: MASK ← 0000FFFF00000000H 3: MASK ← FFFF000000000000H END DST ← (DST AND NOT MASK) OR ((SRC SHL (SEL *16)) AND MASK)

		xmm, r32/m16, imm8	Inserts the low word from the r32 register or memory into the xmm register at the word position specified by imm8.	SEL ← Count AND 7H CASE (SEL) OF 0: MASK ← 0000000000000000000000000000FFFFH 1: MASK ← 000000000000000000000000FFFF0000H 2: MASK ← 00000000000000000000FFFF00000000H 3: MASK ← 0000000000000000FFFF000000000000H 4: MASK ← 000000000000FFFF0000000000000000H 5: MASK ← 00000000FFFF00000000000000000000H 6: MASK ← 0000FFFF000000000000000000000000H 7: MASK ← FFFF0000000000000000000000000000H END DST ← (DST AND NOT MASK) OR ((SRC SHL (SEL *16)) AND MASK)
Packed Shuffle Words	PSHUFW	mm1, mm2/m64, imm8	Copies words from source (second) operand and inserts them into the destination (first) operand at word locations selected with the order (third) operand.	DST[15..0] ← (SRC SHR (ORDER[1..0] * 16))[15..0] DST[31..16] ← (SRC SHR (ORDER[3..2] * 16))[15..0] DST[47..32] ← (SRC SHR (ORDER[5..4] * 16))[15..0] DST[63..48] ← (SRC SHR (ORDER[7..6] * 16))[15..0]

MXCSR State Management Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Store MXCSR Register State	STMXCSR	m32	Store MXCSR register to the memory. The reserved bits are stored as 0s.	DST ← MXCSR
Load MXCSR Register State	LDMXCSR	m32	Load MXCSR register from the memory.	MXCSR ← SRC

SSE Cacheability Control, Prefetch and Instruction Ordering Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations

Store Selected Bytes of Quad-Word using Non-temporal Hint	MASKMOVQ	mm1, mm2	Stores selected bytes from the mmx register (first operand) into a 64-bit memory location. The address of the memory location is specified by DS:[(E)DI] registers. The mask (second) operand selects which bytes from the source operand are written to the memory.	IF (MASK[7]=1) THEN DS:[(E)DI] ← SRC[7..0]
				IF (MASK[15]=1) THEN DS:[(E)DI+1] ← SRC[15..8]
				IF (MASK[23]=1) THEN DS:[(E)DI+2] ← SRC[23..16]
				IF (MASK[31]=1) THEN DS:[(E)DI+3] ← SRC[31..24]
				IF (MASK[39]=1) THEN DS:[(E)DI+4] ← SRC[39..32]
				IF (MASK[47]=1) THEN DS:[(E)DI+5] ← SRC[47..40]
				IF (MASK[55]=1) THEN DS:[(E)DI+6] ← SRC[55..48]
				IF (MASK[63]=1) THEN DS:[(E)DI+7] ← SRC[63..56]

Store Quad-Word Using Non-temporal Hint	MOVNTQ	m128, xmm	Moves the double quad word from xmm to m128 using a non-temporal hint to prevent caching of the data during the write to memory.	DST ← SRC

Store Packed Single-Precision Floating-Point Values Using Non-temporal Hint	MOVNTPS	m128, xmm	Moves the packed single-precision floating-point values from xmm to m128 using a non-temporal hint to minimize cache pollution of the data during the write to memory.	DST ← SRC
Prefetch temporal to All Cache Levels	PREFETCH0	m8	Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy using T0 hint; it means to all levels of the cache hierarchy.	
Prefetch temporal to First Level Cache	PREFETCH1	m8	Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy using T1 hint; it means to the first level cache.	
Prefetch temporal to Second Level Cache	PREFETCH2	m8	Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy using T1 hint; it means to the second level cache.	

SSE2 Instruction set

SSE2 Data Movement Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Move Aligned Packed Doubles	MOVAPD	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves packed double-precision floating-point values from source to destination operand. When the source or destination operand is a memory location, it must be aligned on a 16-byte boundary.	DST ← SRC
Move Unaligned Packed Doubles	MOVUPD	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves packed double-precision floating-point values from source to destination operand. When the source or destination operand is a memory location, it may be unaligned on a 16-byte boundary.	DST ← SRC
Move Double Quad-Word Aligned	MOVDQA	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves a double quad word from the source (second) operand to the destination (first) operand. When the source or destination operand is a memory location, it must be aligned on a 16-byte boundary.	DST ← SRC
Move Double Quad-Word Unaligned	MOVDQU	xmm1, xmm2/m128 xmm1/m128, xmm2	Moves a double quad word from the source (second) operand to the destination (first) operand. When the source or destination operand is a memory location, it may be unaligned on a 16-byte boundary.	DST ← SRC
Move Quad-Word from MMX to XMM Register	MOVQ2DQ	xmm, mm	Moves the quad word from the mmx register to the low quad word of the xmm register.	DST[63..0] ← SRC DST[127..64] ← 0000000000000000H
Move Quad-Word from XMM to MMX Register	MOVDQ2Q	mm, xmm	Moves the low quad word from the xmm register to the mmx register.	DST ← SRC[63..0]
Move Low Packed Double	MOVLPD	xmm, m64	Moves a double-precision floating point from the memory to the low quad word of the xmm register.	DST[63..0] ← SRC *// DST[127..64] remains unchanged*
		m64, xmm	Moves a double-precision floating point from the low quad word of the xmm register to the memory.	DST ← SRC[63..0]
Move High Packed Double	MOVHPD	xmm, m64	Moves a double-precision floating point from the memory to the high quad word of the xmm register.	DST[127..64] ← SRC *// DST[63..0] remains unchanged*
		m64, xmm	Moves a double-precision floating point from the high quad word of the xmm register to the memory.	DST ← SRC[127..63]
Extract Packed Doubles Sign Mask	MOVMSKPD	r32, xmm	Extracts 2-bit sign mask of from xmm and stores in r32.	DST[0] ← SRC[31] DST[1] ← SRC[63] DST[31..2] ← 000000H

Move Scalar Double	MOVSD	xmm, m128	Moves scalar double-precision floating-point value from source to destination operand.	DST[63..0] ← SRC[63..0] DST[127..64] ← 0000000000000000H

SSE2 Packed Arithmetic Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Add Packed Doubles	ADDPD	xmm1, xmm2/m128	Adds 2 packed double-precision floating-point values from *xmm2/m128* to 2 packed double-precision floating-point values in *xmm1*.	DST[63..0] ← DEST[63..0] + SRC[63..0]; DST[127..64] ← DEST[127..64] + SRC[127..64];
Add Scalar Doubles	ADDSD	xmm1, xmm2/m64	Adds the low double-precision floating-point value from *xmm2/m64* to the low double-precision floating-point value in *xmm1*.	DST[63..0] ← DST[63..0] + SRC[63..0]; *// DST[127..64] remains unchanged*
Subtract Packed Doubles	SUBPD	xmm1, xmm2/m128	Subtracts packed double-precision floating-point values from *xmm2/m128* from 2 packed double-precision floating-point values in *xmm1*.	DST[63..0] ← DEST[63..0] – SRC[63..0]; DST[127..64] ← DEST[127..64] – SRC[127..64];

Subtract Scalar Doubles	SUBSD	xmm1, xmm2/m64	Subtracts the low double-precision floating-point value from *xmm2/m64* from the low double-precision floating-point value in *xmm1*.	DST[63..0] ← DST[63..0] – SRC[63..0]; *// DST[127..64] remains unchanged*
Multiply Packed Doubles	MULPD	xmm1, xmm2/m128	Multiplies 2 packed double-precision floating-point values from *mm1* with 2 packed double-precision floating-point values in *xmm2/m128*.	DST[63..0] ← DEST[63..0] * SRC[63..0]; DST[127..64] ← DEST[127..64] + SRC[127..64];
Multiply Scalar Doubles	MULSD	xmm1, xmm2/m64	Multiplies the low double-precision floating-point value from *mm1* with the low double-precision floating-point value in *xmm2/m64*.	DST[63..0] ← DST[63..0] * SRC[63..0]; *// DST[127..64] remains unchanged*
Divide Packed Doubles	DIVPD	xmm1,xmm2/m128	Divides 2 packed double-precision floating-point values from *mm1* by 2 packed double-precision floating-point values in *xmm2/m128*.	DST[63..0] ← DST[63..0] / SRC[63..0]; DST[127..64] ← DST[127..64] / SRC[127..64];
Divide Scalar Doubles	DIVSD	xmm1,xmm2/m64	Divides the low double-precision floating-point value from *mm1* by the low double-precision floating-point value in *xmm2/m64*.	DST[63..0] ← DST[63..0] / SRC[63..0]; *// DST[127..64] remains unchanged*

Square Roots of Packed Doubles	SQRTPD	xmm1, xmm2/m128	Computes the square roots of the packed double-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*. SRC 127 SQRT SQRT DST 127	DST[63..0] ← SquareRoot (SRC[63..0]); DST[127..64] ← SquareRoot (SRC[127..64]);
Square Root of Scalar Double	SQRTSD	xmm1, xmm2/m32	Computes the square root of the scalar double-precision floating-point value in *xmm2/m128* and stores the result in *xmm1*. SRC 127 SQRT DST 127	DST[63..0] ← SquareRoot (SRC[63..0]); *// DST[127..64] remains unchanged*
Maximum Packed Double	MAXPD	xmm1, xmm2/m128	Returns the maximum single-precision floating-point values between *xmm2/m128* and *xmm1*. DST 127 SRC 127 MAX MAX DST 127	DST[63..0] ← MaximumOf (DST[63..0], SRC[63..0]); DST[127..64] ← MaximumOf (DST[127..64], SRC[127..64]);
Maximum Scalar Double	MAXSD	xmm1, xmm2/m32	Returns the maximum scalar single-precision floating-point value between *xmm2/m128* and *xmm1*. DST 127 SRC 127 MAX DST 127	DST[63..0] ← MaximumOf (DST[63..0], SRC[63..0]); *// DST[127..64] remains unchanged*
Minimum Packed Double	MINPD	xmm1, xmm2/m128	Returns the minimum single-precision floating-point values between *xmm2/m128* and *xmm1*. DST 127 SRC 127 MIN MIN DST 127	DST[63..0] ← MinimumOf (DST[63..0], SRC[63..0]); DST[127..64] ← MiniimumOf (DST[127..64], SRC[127..64]);

Minimum Scalar Double	MINSD	xmm1, xmm2/m32	Returns the minimum scalar single-precision floating-point value between *xmm2/m128* and *xmm1*.	DST[63..0] ← MinimumOf (DST[63..0], SRC[63..0]); *// DST[127..64] remains unchanged*

SSE2 Logical Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
AND of Packed Doubles	ANDPD	xmm1, xmm2/m128	Performs a bitwise AND operation of the two packed double-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand.	DST[63..0] ← DEST[63..0] AND SRC[63..0]; DST[127..64] ← DEST[127..64] AND SRC[127..64]
AND NOT of Packed Doubles	ANDNPD	xmm1, xmm2/m128	Inverts the bits of the two packed double-precision floating-point values in the destination (first) operand, performs a bitwise logical AND operation of the two packed double-precision floating-point values from the temporary inverted result and source (second) operand and stored the result in the destination operand.	DST[63..0] ← (NOT DEST[63..0]) AND SRC[63..0]; DST[127..64] ← (NOT DEST[127..64]) AND SRC[127..64]
OR of Packed Doubles	ORPD	xmm1, xmm2/m128	Performs a bitwise OR operation of the two packed double-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand.	DST[63..0] ← DEST[63..0] OR SRC[63..0]; DST[127..64] ← DEST[127..64] OR SRC[127..64]

Exclusive OR of Packed Doubles	XORPD	xmm1, xmm2/m128	Performs a bitwise XOR operation of the two packed double-precision floating-point values from the destination (first) and source (second) operands and stored the result in the destination operand.	DST[63..0] ← DEST[63..0] XOR SRC[63..0]; DST[127..64] ← DEST[127..64] XOR SRC[127..64]

SSE2 Comparison Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Compare Packed Doubles	CMPPD	xmm1, xmm2/m128, imm8	Compares packed double-precision floating-point values in *xmm2/m128* and *xmm1* using *imm8* as comparison predicate: 0 – equal, 1 – less than, 2 – less or equal, 3 – unordered, 4 – not equal, 5 – not less, 6 – not less or equal, 7 – ordered. The result of each comparison in a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The unordered relationship is true when at leas one of the two operands is a NAN; the ordered relationship id true when neither operand is a NAN.	CMP0 ← DST[63..0] OP SRC[63..0]; CMP1 ← DST[127..64] OP SRC[127..64]; IF CMP0 THEN DST[63..0] ← FFFFFFFFFFFFFFFFH ELSE DST[63..0] ← 0000000000000000H; IF CMP1 THEN DST[127..64] ← FFFFFFFFFFFFFFFFH ELSE DST[127..64] ← 0000000000000000H
Compare Packed Doubles	CMPEQPD CMPLTPD CMPLEPD CMPUNORDPD CMPNEQPD CMPNLTPD CMPNLEPD CMPORDPD	xmm1, xmm2	<=> CMPPD xmm1,xmm2, 0 <=> CMPPD xmm1,xmm2, 1 <=> CMPPD xmm1,xmm2, 2 <=> CMPPD xmm1,xmm2, 3 <=> CMPPD xmm1,xmm2, 4 <=> CMPPD xmm1,xmm2, 5 <=> CMPPD xmm1,xmm2, 6 <=> CMPPD xmm1,xmm2, 7	see CMPPD

Compare Scalar Doubles	CMPSD	xmm1, xmm2/m64, imm8	Compares the low double-precision floating-point values in *xmm2/m128* and *xmm1* using *imm8* as comparison predicate: 0 – equal, 1 – less than, 2 – less or equal, 3 – unordered, 4 – not equal, 5 – not less, 6 – not less or equal, 7 – ordered. The result of each comparison in a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The unordered relationship is true when at leas one of the two operands is a NAN; the ordered relationship id true when neither operand is a NAN.	CMP0 ← DST[63..0] OP SRC[63..0]; IF CMP0 THEN DST[63..0] ← FFFFFFFFFFFFFFFFH ELSE DST[63..0] ← 0000000000000000H; *// DST[127..64] remains unchanged*
Compare Scalar Doubles	CMPEQSD CMPLTSD CMPLESD CMPUNORDSD CMPNEQSD CMPNLTSD CMPNLESD CMPORDSD	xmm1, xmm2	<=> CMPSD xmm1,xmm2, 0 <=> CMPSD xmm1,xmm2, 1 <=> CMPSD xmm1,xmm2, 2 <=> CMPSD xmm1,xmm2, 3 <=> CMPSD xmm1,xmm2, 4 <=> CMPSD xmm1,xmm2, 5 <=> CMPSD xmm1,xmm2, 6 <=> CMPSD xmm1,xmm2, 7	see CMPSD
Compare Scalar Doubles and Set EFLAGS	COMISD	xmm1, xmm2/m64	Compares low double-precision floating-point values in the operands and sets the EFLAGS flags accordingly. Performs ordered compare. This instruction differs from the UCOMISS instruction in that is signals an invalid operation exception when a source operand is a QNan or and SNaN.	Result ← OrderedCompare(DST[63..0], SRC[63..0]) CASE (Result) OF UNORDERED: ZF, PF, CF ← 111; GREATER_THAN: ZF, PF, CF ← 000; LESS_THAN: ZF, PF, CF ← 001; EQUAL: ZF, PF, CF ← 100; END OF, AF, SF ← 0;
Unordered Compare Scalar Doubles and set EFLAGS	UCOMISD	xmm1, xmm2/m64	Compares low double-precision floating-point values in the operands and sets the EFLAGS flags accordingly. Performs unordered compare. This instruction differs from the COMISS instruction in that is signals an invalid operation exception only when a source operand is a SNaN.	Result ← UnorderedCompare(DST[63..0], SRC[63..0]) CASE (Result) OF UNORDERED: ZF, PF, CF ← 111; GREATER_THAN: ZF, PF, CF ← 000; LESS_THAN: ZF, PF, CF ← 001; EQUAL: ZF, PF, CF ← 100; END OF, AF, SF ← 0;

SSE2 Shuffle and Unpack Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Shuffle Packed Dwords	PSHUFD	xmm1, xmm2/m128, imm8	Moves double words from source (second) operand and inserts them in the destination (first) operand at locations selected with the order (third) operand.	DST[31..0] ← (SRC SHR (ORDER[1..0]*32))[31..0] DST[63..32] ← (SRC SHR (ORDER[3..2]*32))[31..0] DST[95..64] ← (SRC SHR (ORDER[5..4]*32))[31..0] DST[127..96] ← (SRC SHR (ORDER[7..6]*32))[31..0]
Shuffle Packed Low Words	PSHUFLW	xmm1, xmm2/m128, imm8	Moves the words from the low quad word of the source (second) operand and inserts them to the low quad word of the destination (first) operand at locations selected with the order (third) operand.	DST[15..0] ← (SRC SHR (ORDER[1..0]*16))[15..0] DST[31..16] ← (SRC SHR (ORDER[1..0]*16))[15..0] DST[47..32] ← (SRC SHR (ORDER[1..0]*16))[15..0] DST[63..48] ← (SRC SHR (ORDER[3..2]*16))[15..0] DST[127..64] ← SRC[127..64]
Shuffle Packed High Words	PSHUFHW	xmm1, xmm2/m128, imm8	Moves the words from the high quad word of the source (second) operand and inserts them to the high quad word of the destination (first) operand at locations selected with the order (third) operand.	DST[63..0] ← SRC[63..0] DST[79..64] ← (SRC SHR (ORDER[1..0]*16))[79..64] DST[95..80] ← (SRC SHR (ORDER[1..0]*16))[79..64] DST[111..96] ← (SRC SHR (ORDER[1..0]*16))[79..64] DST[127..112] ← (SRC SHR (ORDER[3..2]*16))[79..64]
Unpack Low Packed Doubles	UNPCKLPD	xmm1, xmm2/m128	Unpacks and interleaves the low double-precision floating-point values from the low quad words of the source (second) operand and the destination (first) operand.	DST[63..0] ← DST[63..0] DST[127..64] ← SRC[63..0]

Unpack High Packed Doubles	UNPCKHPD	xmm1, xmm2/m128	Unpacks and interleaves the low double-precision floating-point values from the high quad words of the source (second) operand and the destination (first) operand.	DST[63..0] ← DST[127..64] DST[127..64] ← DST[127..64]
Unpack Low Data	PUNPCKLQDQ	xmm1, xmm2/m128	Unpacks and interleaves low-order quad words from xmm1 and xmm2/m128 into xmm1 register.	DST[63..0] ← DST[63..0] DST[127..64] ← SRC[63..0]
Unpack Low Data	PUNPCKHQDQ	xmm1, xmm2/m128	Unpacks and interleaves high-order quad words from xmm1 and xmm2/m128 into xmm1 register.	DST[63..0] ← DST[127..6] DST[127..64] ← SRC[127..64]

SSE2 Conversion Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Convert Packed Integers to Packed Doubles	CVTPI2PD	xmm, mm/m64	Converts two packed signed double word integers from *mm/mem64* to two packed double-precision floating-point values from *xmm*.	DST[63..0] ← IntToDouble (SRC[31..0]); DST[127..64] ← IntToDouble (SRC[63..32]);
Convert Packed Doubles to Packed Integers	CVTPD2PI	mm, xmm/m128	Converts two packed double-precision floating-point values from *xmm/m128* to two packed signed double-word integers in *mm*.	DST[31..0] ← DoubleToInt (SRC[63..0]); DST[63..32] ← DoubleToInt (SRC[127..64])

Convert with Truncation Packed Doubles to Packed Integers	CVTTPD2PI	mm, xmm/m128	Converts two packed double-precision floating-point values from *xmm/m128* to two packed signed double-word integers in *mm* using truncation.	DST[31..0] ← TruncateDoubleToInt (SRC[63..0]); DST[63..32] ← TruncateDoubleToInt (SRC[127..64])
Convert Packed Doubles to Packed Dwords	CVTPD2DQ	xmm1, xmm2/m128	Converts two packed double-precision floating-point values from *xmm2/m128* to two packed signed double-word integers in *xmm1*.	DST[31..0] ← DoubleToInt (SRC[63..0]); DST[63..32] ← DoubleToInt (SRC[127..64]); DST[127-64] ← 0000000000000000H
Convert with Truncation Packed Doubles to Packed Dwords	CVTTPD2DQ	xmm1, xmm2/m128	Converts two packed double-precision floating-point values from *xmm2/m128* to two packed signed double-word integers in *xmm1* using truncation.	DST[31..0] ← TruncateDoubleToInt (SRC[63..0]); DST[63..32] ← TruncateDoubleToInt (SRC[127..64]); DST[127-64] ← 0000000000000000H
Convert Packed Dwords to Packed Doubles	CVTDQ2PD	xmm1, xmm2/m64	Converts two packed signed double-word integers from *xmm2/m64* to two packed double-precision floating-point values in *xmm1*.	DST[63..0] ← IntToDouble(SRC[31..0]); DST[127..64] ← IntToDouble(SRC[63..32])
Convert Packed Singles to Packed Doubles	CVTPS2PD	xmm1, xmm2/m64	Converts two packed single-precision floating-point values from *xmm2/m64* to two packed double-precision floating-point values in *xmm1*.	DST[63..0] ← SingleToDouble (SRC[31..0]); DST[127..64] ← SingleToDouble (SRC[63..32])
Convert Packed Doubles to Packed Singles	CVTPD2PS	xmm1, xmm2/m128	Converts two packed double-precision floating-point values from *xmm2/m128* to two packed single-precision floating-point values in *xmm1*.	DST[31..0] ← DoubleToSingle (SRC[63..0]); DST[63..32] ← DoubleToSingle (SRC[127..64]); DST[127-64] ← 0000000000000000H

Convert Scalar Single Scalar Double.	CVTSS2SD	xmm1, xmm2/m32	Converts one scalar single-precision floating-point value from *xmm2/m32* to one double-precision floating-point value in *xmm*.	DST[63..0] ← SingleToDouble (SRC[31..0]); *// DST[127..64] remains unchanged*
Convert Scalar Double to Scalar Single	CVTSD2SS	xmm1, xmm2/m64	Converts one scalar double-precision floating-point value from *xmm/m64* to one single-precision floating-point value in *xmm1*.	DST[31..0] ← DoubleToSingle (SRC[63..0]); *// DST[127..32] remains unchanged*
Convert Scalar Double to Scalar Integer	CVTSD2SI	r32, xmm/m64	Converts one scalar double-precision floating-point value from *xmm/m64* to one signed double-word integer in *r32*.	DST← DoubleToInt (SRC[63..0]);
Convert with Truncation Scalar Double to Scalar Integer	CVTTSD2SI	r32, xmm/m64	Converts one scalar double-precision floating-point value from *xmm/m64* to one signed double-word integer in *r32* using truncation.	DST← TruncateDoubleToInt (SRC[63..0]);
Convert Scalar Integer to Scalar Double	CVTSI2SD	xmm, r/m32	Converts one signed double-word integer from *r/m32* to one scalar double-precision floating-point value in *xmm*.	DST[63..0]← IntToDouble (SRC); *// DST[127..64] remains unchanged*

SSE2 Packed Single-Precision Floating-Point Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Convert Packed Dwords to Packed Singles	CVTDQ2PS	xmm1, xmm2/m128	Converts four packed signed double-word integers from *xmm2/m128* to four packed single-precision floating point values in *xmm1*.	DST[31..0] ← IntToSingle(SRC[31..0]); DST[63..32] ← IntToSingle(SRC[63..32]); DST[95..64] ← IntToSingle(SRC[95..64]); DST[127..96] ← IntToSingle(SRC[127..96])
Convert Packed Singles to Packed Dwords	CVTPS2DQ	xmm1, xmm2/m128	Converts four packed single-precision floating-point values from *xmm2/m128* to four packed signed double-word integers in *xmm1*.	DST[31..0] ← SingleToInt (SRC[31..0]); DST[63..32] ← SingleToInt (SRC[63..32]); DST[95..64] ← SingleToInt (SRC[95..64]); DST[127..96] ← SingleToInt (SRC[127..96])
Convert with	CVTTPS2DQ	xmm1, xmm2/m128	Converts four packed single-precision floating-point values from	DST[31..0] ← TruncateSingleToInt (SRC[31..0]);

Truncation Packed Singles to Packed Dwords			*xmm2/m128* to four packed signed double-word integers in *xmm1* using truncation.	DST[63..32] ← TruncateSingleToInt (SRC[63..32]); DST[95..64] ← TruncateSingleToInt (SRC[95..64]); DST[127..96] ← TruncateSingleToInt (SRC[127..96])

SSE2 128-Bit SIMD Integer Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Add Packed Quad-word Integers	PADDQ	mm1, mm2/m64	Adds the quad word integer from the source (second) operand to the destination (first) operand.	DST[63..0] ← DST[63..0] + SRC[63..0]
		xmm1, xmm2/m128	Adds 2 quad word integers from the source (second) operand to 2 quad word integers in the destination (first) operand.	DST[63..0] ← DST[63..0] + SRC[63..0] DST[127..64] ← DST[127..64] + SRC[127..64]
Subtract Packed Quad-word Integers	PSUBQ	mm1, mm2/m64	Subtracts the quad word integer from the source (second) operand from the destination (first) operand.	DST[63..0] ← DST[63..0] – SRC[63..0]
		xmm1, xmm2/m128	Subtracts 2 quad word integers from the source (second) operand from 2 quad word integers in the destination (first) operand.	DST[63..0] ← DST[63..0] – SRC[63..0] DST[127..64] ← DST[127..64] – SRC[127..64]

Multiply Packed Unsigned Double-word Integers	PMULUDQ	mm1, mm2/m64	Multiplies the unsigned double word integer from the destination (first) operand by the unsigned double word integer from the source (second) operand and stores the quad word result in the destination (first) operand.	DST[63..0] ← DST[31..0] * SRC[31..0]
		xmm1, xmm2/m128	Subtracts 2 quad word integers from the source (second) operand from 2 quad word integers in the destination (first) operand.	DST[63..0] ← DST[31..0] * SRC[31..0] DST[127..64] ← DST[95..64] * SRC[95..64]
Packed Shift Left Logical Quad-word	PSLLDQ	xmm1, imm8	Shifts xmm1 left by imm8 bytes while shifting in 0s.	TMP ← Count; IF (T MP>15) THEN TMP ←16; DST ← DST SHL (TMP * 8)
Packed Shift Right Logical Quad-word	PSRLDQ	xmm1, imm8	Shifts xmm1 right by imm8 bytes while shifting in 0s.	TMP ← Count; IF (T MP>15) THEN TMP ←16; DST ← DST SHR (TMP * 8)

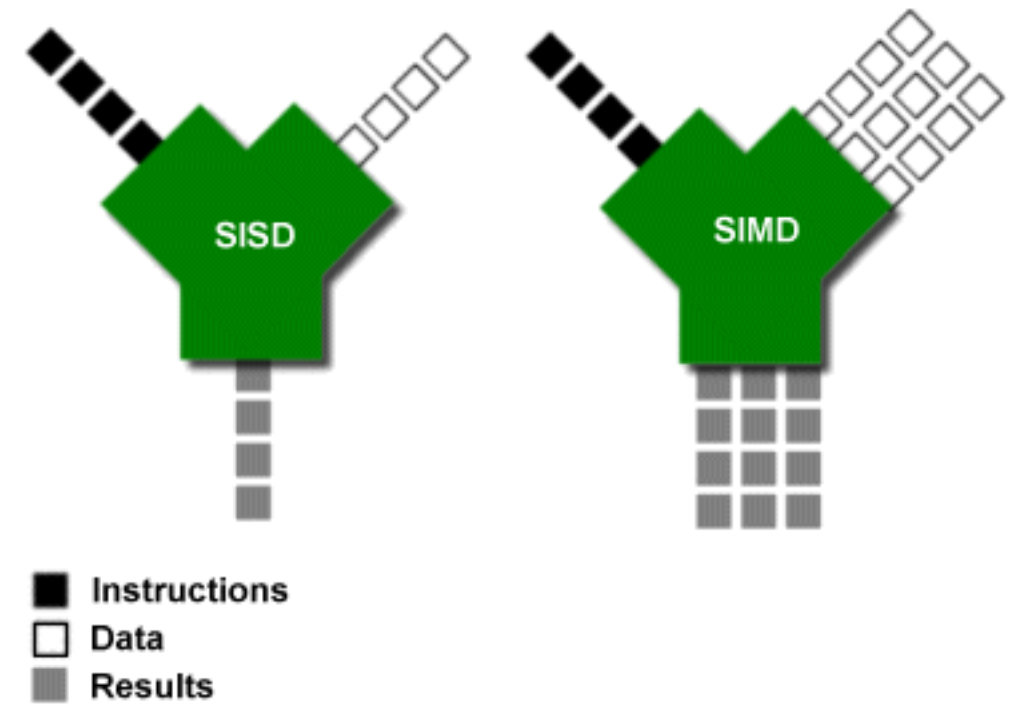
SSE2 Cacheability Control and Instructions Ordering Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations
Flush Cache Line	CLFLUSH	m8	Invalidated the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast through the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with the memory, it is written to memory before invalidation.	
Store Fence	SFENCE		Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction.	
Load Fence	LFENCE		Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction.	
Memory Fence	MFENCE		Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction.	
Spin Loop Hint	PAUSE			

Store Selected Bytes of Double Quad-Word Using Non-temporal Hint	MASKMOVDQU	mm1, mm2	Stores selected bytes from the mmx register (first operand) into a 128-bit memory location. The address of the memory location is specified by DS:[(E)DI] registers. The mask (second) operand selects which bytes from the source operand are written to the memory.	IF (MASK[7]=1) THEN DS:[(E)DI] ← SRC[7..0]
				IF (MASK[15]=1) THEN DS:[(E)DI+1] ← SRC[15..8]
				IF (MASK[23]=1) THEN DS:[(E)DI+2] ← SRC[23..16]
				IF (MASK[31]=1) THEN DS:[(E)DI+3] ← SRC[31..24]
				IF (MASK[39]=1) THEN DS:[(E)DI+4] ← SRC[39..32]
				IF (MASK[47]=1) THEN DS:[(E)DI+5] ← SRC[47..40]
				IF (MASK[55]=1) THEN DS:[(E)DI+6] ← SRC[55..48]
				IF (MASK[63]=1) THEN DS:[(E)DI+7] ← SRC[63..56]
				IF (MASK[71]=1) THEN DS:[(E)DI+8] ← SRC[71..64]
				IF (MASK[79]=1) THEN DS:[(E)DI+9] ← SRC[79..80]
				IF (MASK[87]=1) THEN DS:[(E)DI+10] ← SRC[87..80]
				IF (MASK[95]=1) THEN DS:[(E)DI+11] ← SRC[95..88]
				IF (MASK[103]=1) THEN DS:[(E)DI+12] ← SRC[103..96]
				IF (MASK[111]=1) THEN DS:[(E)DI+13] ← SRC[111..104]
				IF (MASK[119]=1) THEN DS:[(E)DI+14] ← SRC[119..112]
				IF (MASK[127]=1) THEN DS:[(E)DI+15] ← SRC[127..120]

Store Double Quad-Word Using Non-temporal Hint	MOVNTDQ	m128, xmm	Moves the double quad word from xmm to m128 using a non-temporal hint to prevent caching of the data during the write to memory.	DST ← SRC
Store Packed Double-Precision Floating-Point Values Using Non-temporal Hint	MOVNTPD	m128, xmm	Moves the packed double-precision floating-point values from xmm to m128 using a non-temporal hint to minimize cache pollution during the write to memory.	DST ← SRC
Store Double-Word Using Non-temporal Hint	MOVNTI	m32, r32	Moves the double word integer from the r32 register to the m32 memory using a non-temporal hint to minimize cache pollution during the write to memory.	DST ← SRC

NEON == Advanced SIMD

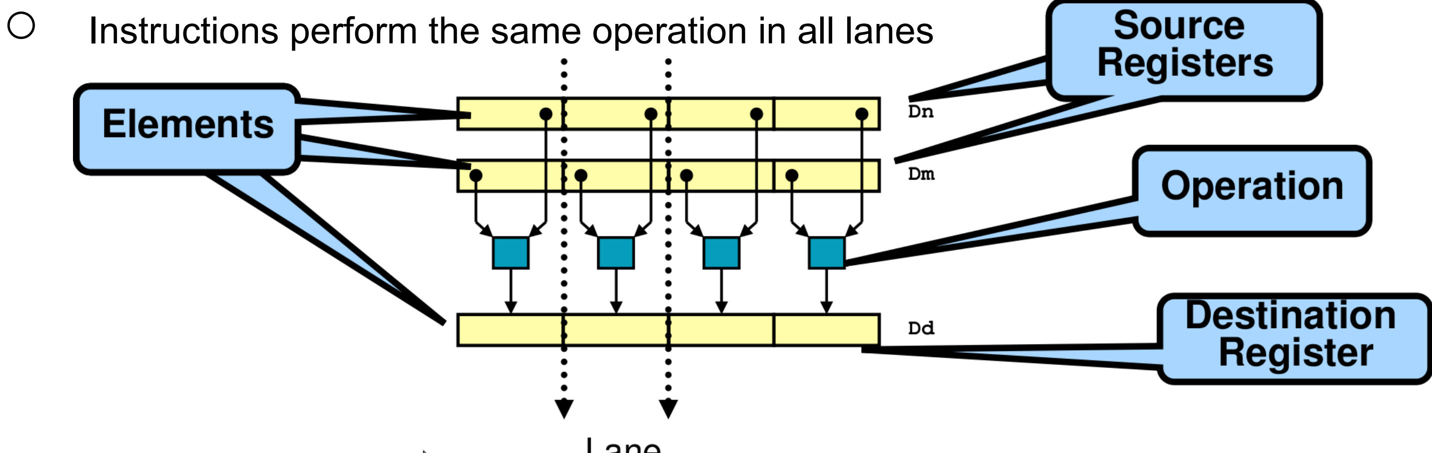


was announced in 2004

NEON often gives 8–20x boost on Cortex-A8 but only 2–5x on Cortex-A9

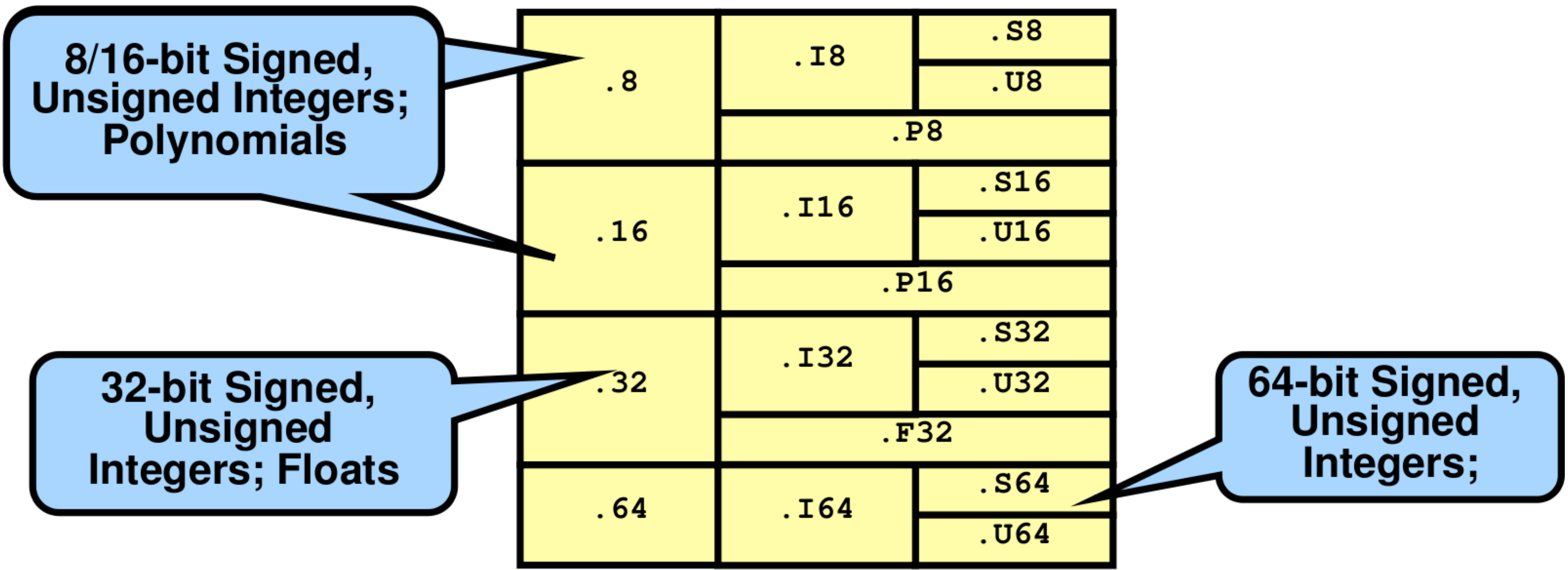
What is NEON?

- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set
  - 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide)
- NEON Instructions perform “Packed SIMD” processing
  - Registers are considered as vectors of elements of the same data type
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions perform the same operation in all lanes



Data Types

- NEON natively supports a set of common data types
  - Integer and Fixed-Point; 8-bit, 16-bit, 32-bit and 64-bit
  - 32-bit **Single-precision** Floating-point



- Data types are represented using a bit-size and format letter

Registers

- NEON provides a 256-byte register file
  - **NEON has its own execution pipelines and a register bank that is distinct from the ARM register bank**
  - shares the same set of registers with VFP that is a floating point hardware accelerator, not parallel like NEON
- NEON Dual view
  - 32 registers, 64-bits wide (Dx)
  - 16 registers, 128-bits wide (Qx)
- VFP Dual view
  - 32 registers, 64-bits wide (Dx)
  - 32 registers, 32-bits wide (Sx)

ARMv7 Advanced SIMD (NEON) and VFPv3 extension registers

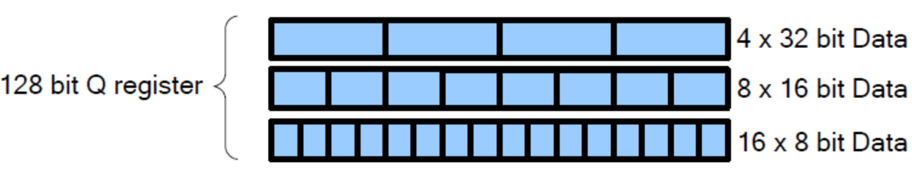
S:32bit D:64bit Q:128bit

VFP	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21	S22	S23	S24	S25	S26	S27	S28	S29	S30	S31
VFP+NEON	D0		D1		D2		D3		D4		D5		D6		D7		D8		D9		D10		D11		D12		D13		D14		D15	
NEON	Q0				Q1				Q2				Q3				Q4				Q5				Q6				Q7			

VFP+NEON	D16	D17	D18	D19	D20	D21	D22	D23	D24	D25	D26	D27	D28	D29	D30	D31
NEON	Q8		Q9		Q10		Q11		Q12		Q13		Q14		Q15	

Register Mapping

- NEON Advanced SIMD and VFP use the same register set



Advanced SIMD and Floating-point register mapping

Figure A2-1 shows the different views of Advanced SIMD and Floating-point register banks, and the relationship between them.

S0-S31 VFP only	D0-D15 VFPv2, VFPv3-D16, or VFPv4-D16	D0-D31 VFPv3-D32, VFPv4-D32, or Advanced SIMD	Q0-Q15 Advanced SIMD only
S0, S1	D0	D0	Q0
S2, S3	D1	D1	Q0
S4, S5	D2	D2	Q1
S6, S7	D3	D3	Q1
...	...	...	...
S28, S29	D14	D14	Q7
S30, S31	D15	D15	Q7
		D16	Q8
		D17	Q8
		...	...
		D30	Q15
		D31	Q15

Figure A2-1 Advanced SIMD and Floating-point Extensions register set

The mapping between the registers is as follows:

- S<2n> maps to the least significant half of D<n>
- S<2n+1> maps to the most significant half of D<n>
- D<2n> maps to the least significant half of Q<n>
- D<2n+1> maps to the most significant half of Q<n>.

For example, software can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.

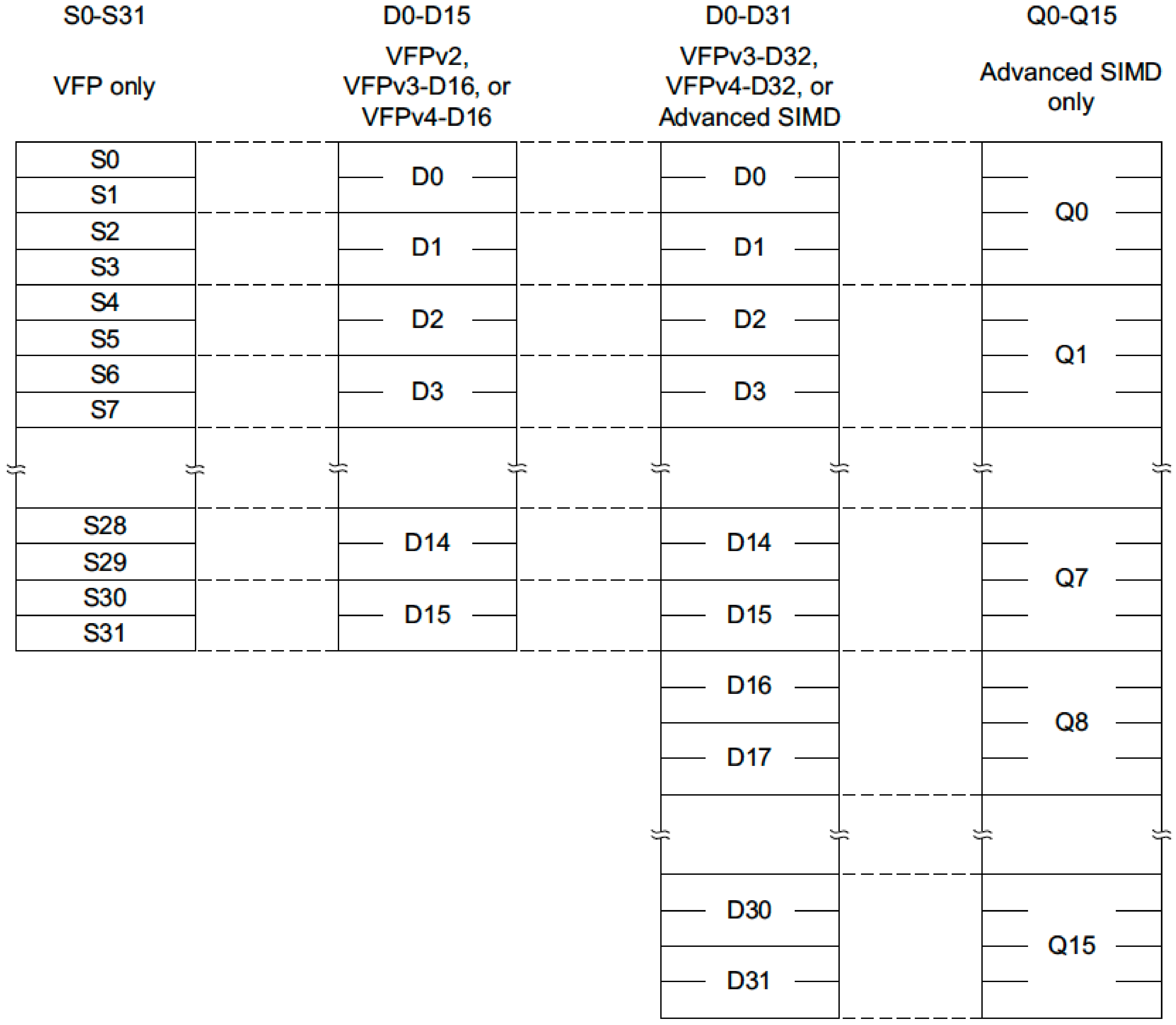
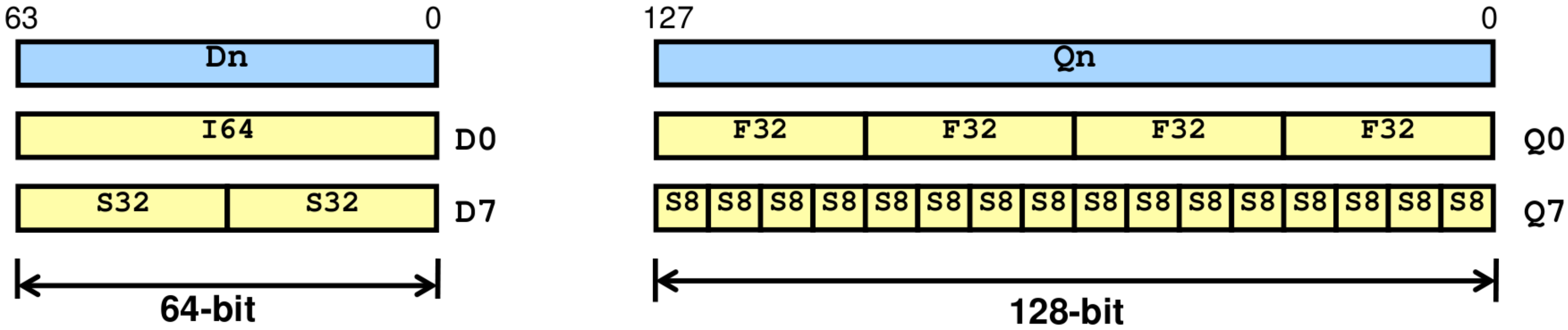


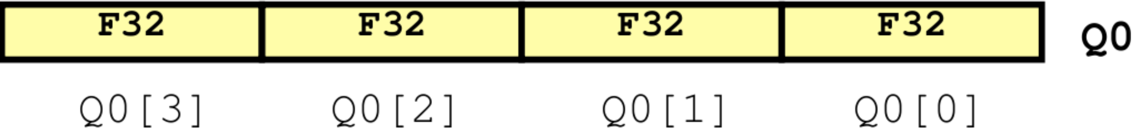
Figure A2-1 Advanced SIMD and Floating-point Extensions register set

Vectors and Scalars

- Registers hold one or more elements of the same data type
  - Vn can be used to reference either a 64-bit Dn or 128-bit Qn register
  - A register, data type combination describes a vector of elements



- Some instructions can reference individual scalar elements
  - Scalar elements are referenced using the array notation Vn[x]



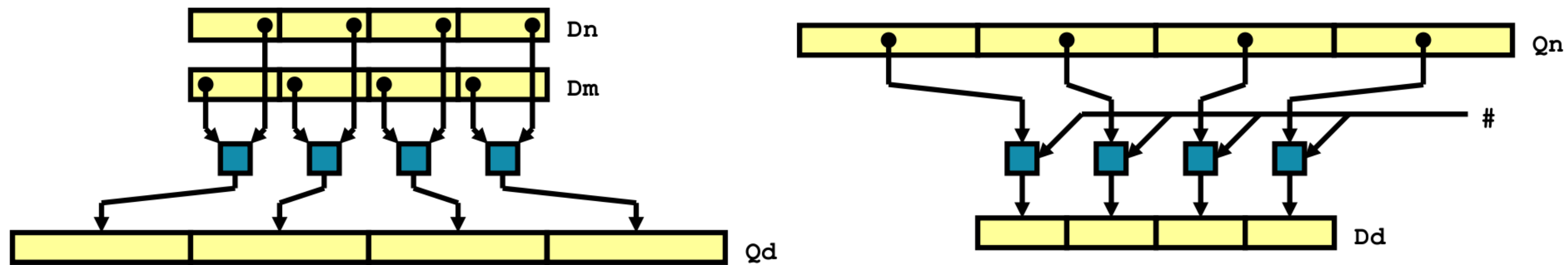
- Array ordering is always from the least significant bit

NEON operations

- Arithmetic
  - VABA, VABD, VABS, VNEG, VADD, VSUB, VADDHN, VSUBHN, VHADD, VHSUB, VPADD, VPADAL, VMAX, VMIN, VPMAX, VPMIN, VCLS, VCLZ, VCNT
- Multiplication
  - VMUL, VMLA, VMLS, VQDMULL, VQDMLAL, VQDMLSL, VQDMULH
- Shifts
  - VSHL, VSHR, VSRA, VSLI, VSRI
- Comparison and Selection
  - VCEQ, VCGE, VCGT, VCLE, VCLT, VTST, VBIF, VBIT, VBSL
- Logical
  - VAND, VBIC, VEOR, VORN, VORR, VMVN
- Reciprocal Estimate/Step, Reciprocal Square Root Estimate/Step
  - VRECPE, VRSQRTE, VRECPS, VRSQRTS
- Miscellaneous
  - VMOV, VDUP, VCVT, VEXT, VREV, VSWP, VTBL, VTBX, VTRN, VUZP, VZIP
- Load/Store
  - VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4

Long, Narrow and Wide Operations

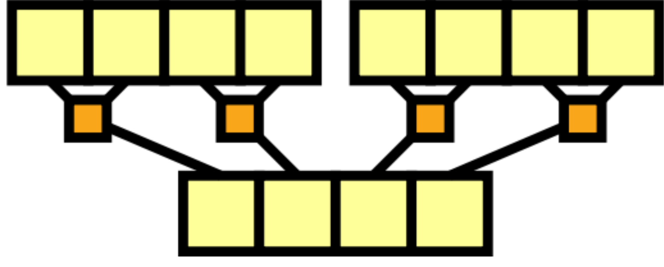
- NEON can utilise both register views in the same instruction
  - Enables instructions to promote or demote elements within operation



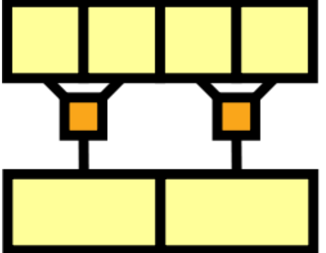
- Long operations promote elements to double the precision
  - Multiply Long (16 x 16 -> 32), Add/Sub Long, Shift Long
- Narrow operations demote data type to half the precision
  - Shift Right and Narrowing Add/Sub, Move
- Wide operations promote the elements of the second operand
  - Add/Sub Wide (16 + 32 -> 32)

Pairwise Operations

- NEON also supports pairwise instructions to add across registers
  - ADD, MIN, MAX
- Normal

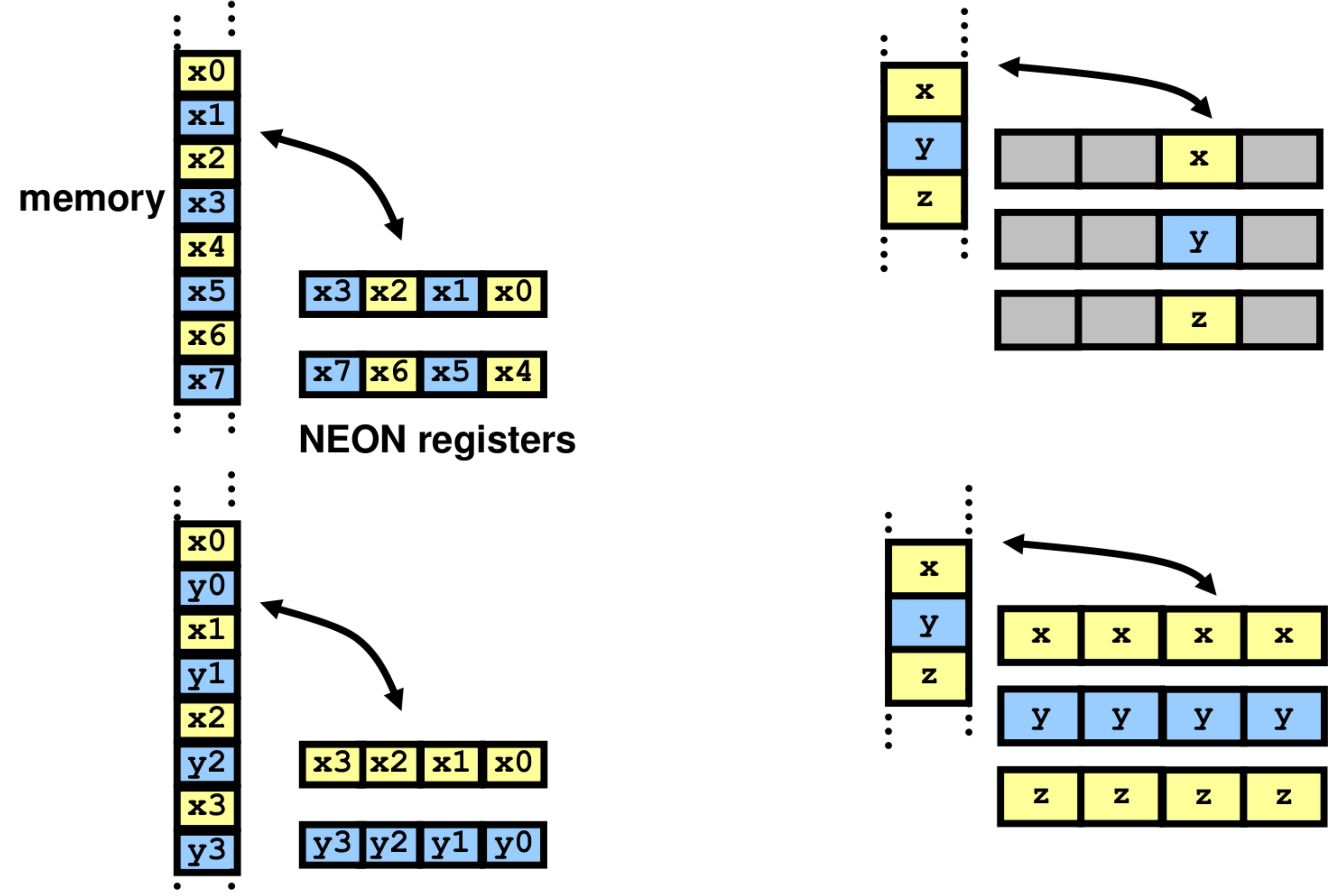


- Long

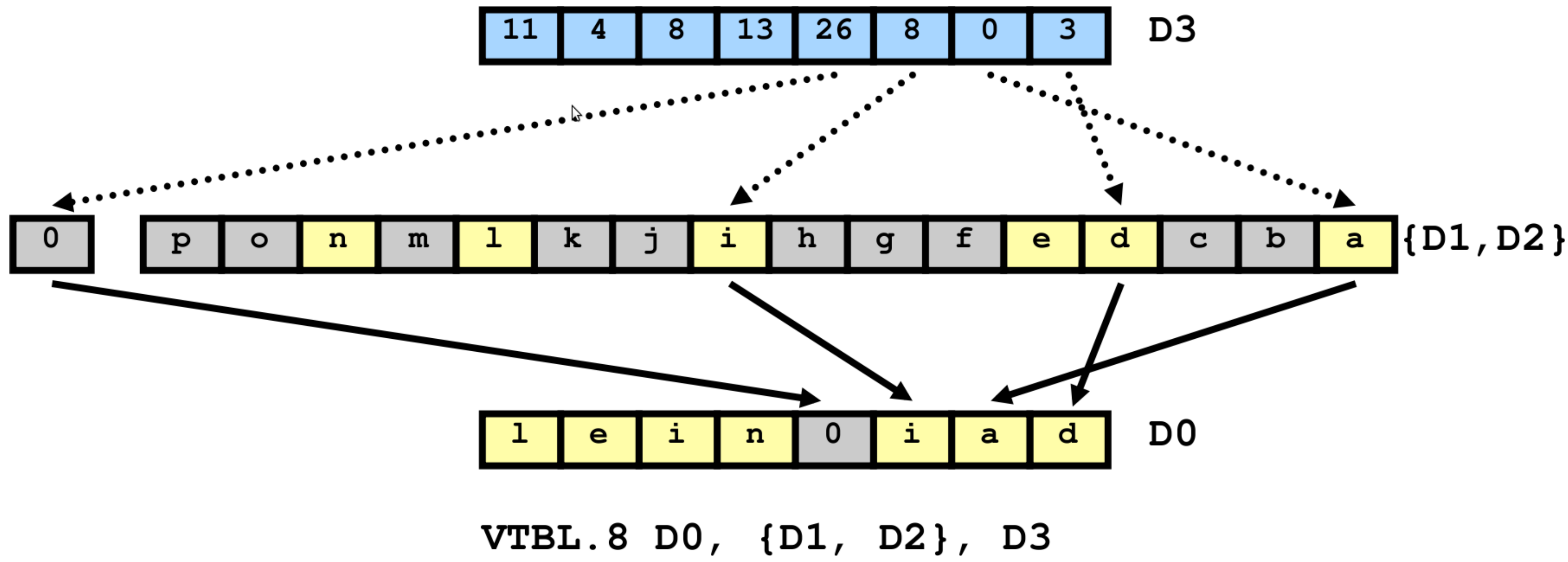


Load/Store Instructions

Several memory access patterns are possible with single instructions



NEON VTBL: Table Lookup



- VTBL : out of range indexes generate 0 result
- VTBX : out of range indexes leave destination unchanged

Example: Binary Threshold

```
void threshold(uchar* src, uchar* dst, int length,
               uchar thresh, uchar maxval)
{
    for(int i = 0; i < length; ++i)
        dst[i] = src[i] <= thresh ? 0 : maxval;
}
```

Binary Threshold: SSE

```
void threshold_SSE(uchar* src, uchar* dst, int length, uchar thresh, uchar maxval)
{
     __m128i _x80 = _mm_set1_epi8('\x80');
     __m128i thresh_s = _mm_set1_epi8(thresh ^ 0x80);
     __m128i maxval_ = _mm_set1_epi8(maxval);
     for(int i = 0; i <= length - 32; i += 32 )
     {
          __m128i v0 = _mm_loadu_si128( (const __m128i*)(src + i) );
          __m128i v1 = _mm_loadu_si128( (const __m128i*)(src + i + 16) );
          v0 = _mm_cmpgt_epi8( _mm_xor_si128(v0, _x80), thresh_s );
          v1 = _mm_cmpgt_epi8( _mm_xor_si128(v1, _x80), thresh_s );
          v0 = _mm_and_si128( v0, maxval_ );
          v1 = _mm_and_si128( v1, maxval_ );
          _mm_storeu_si128( (__m128i*)(dst + i), v0 );
          _mm_storeu_si128( (__m128i*)(dst + i + 16), v1 );
     }
}
```

Binary Threshold: NEON

```
void threshold_NEON(uchar* src, uchar* dst, int length, uchar thresh, uchar maxval)
{
    uint8x16_t vthreshold = vdupq_n_u8(thresh);
    uint8x16_t vvalue = vdupq_n_u8(maxval);
    for(int i = 0; i <= length - 32; i += 32 )
    {
        __builtin_prefetch(src + i + 320);
        uint8x16_t v0 = vld1q_u8(src + i);
        uint8x16_t v1 = vld1q_u8(src + i + 16);
        uint8x16_t r0 = vcgtq_u8(v0, vthreshold);
        uint8x16_t r1 = vcgtq_u8(v1, vthreshold);
        uint8x16_t r0a = vandq_u8(r0, vvalue);
        uint8x16_t r1a = vandq_u8(r1, vvalue);
        vst1q_u8(dst + i, r0a);
        vst1q_u8(dst + i + 16, r1a);
    }
}
```

Example: BGRA unpack

Input:

```
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
```

Output:

```
BBBB BBBB BBBB BBBB
GGGG GGGG GGGG GGGG
RRRR RRRR RRRR RRRR
AAAA AAAA AAAA AAAA
```

BGRA unpack: SSE2 (20 instructions)

```
__m128i v0 = _mm_load_si128((const __m128i*)(bgra));
__m128i v1 = _mm_load_si128((const __m128i*)(bgra + 8));
__m128i v2 = _mm_load_si128((const __m128i*)(bgra + 16));
__m128i v3 = _mm_load_si128((const __m128i*)(bgra + 24));
__m128i t0 = _mm_unpacklo_epi8(v0, v1);
__m128i t1 = _mm_unpackhi_epi8(v0, v1);
__m128i t2 = _mm_unpacklo_epi8(v2, v3);
__m128i t3 = _mm_unpackhi_epi8(v2, v3);
v0 = _mm_unpacklo_epi8(t0, t1);
v1 = _mm_unpackhi_epi8(t0, t1);
v3 = _mm_unpacklo_epi8(t2, t3);
v4 = _mm_unpackhi_epi8(t2, t3);
t0 = _mm_unpacklo_epi32(v0, v1);
t1 = _mm_unpackhi_epi32(v0, v1);
t3 = _mm_unpacklo_epi32(v2, v3);
t4 = _mm_unpackhi_epi32(v2, v3);
__m128i B = _mm_unpacklo_epi64(t0, t2);
__m128i G = _mm_unpackhi_epi64(t0, t2);
__m128i R = _mm_unpacklo_epi64(t1, t3);
__m128i A = _mm_unpackhi_epi64(t1, t3);
```

BGRA unpack: NEON (8 instructions)

```
uint8x16_t v0 = vld1q_u8(bgra);
uint8x16_t v1 = vld1q_u8(bgra+8);
uint8x16_t v2 = vld1q_u8(bgra+16);
uint8x16_t v3 = vld1q_u8(bgra+24);
uint8x16x2_t v01 = vtrnq_u8(v0, v1);
uint8x16x2_t v23 = vtrnq_u8(v2, v3);
uint16x8x2_t BR = vtrnq_u16(vreinterpretq_u16_u8(v01.val[0]),
                            vreinterpretq_u16_u8(v23.val[0]));
uint16x8x2_t GA = vtrnq_u16(vreinterpretq_u16_u8(v01.val[1]),
                            vreinterpretq_u16_u8(v23.val[1]));
uint8x16_t B = vreinterpretq_u8_u16(BR.val[0]);
uint8x16_t G = vreinterpretq_u8_u16(GA.val[0]);
uint8x16_t R = vreinterpretq_u8_u16(BR.val[1]);
uint8x16_t A = vreinterpretq_u8_u16(GA.val[1]);
```

BGRA unpack: NEON 2

```
uint8x16x4_t vbgra = vld4q_u8(bgra);
```

(surprisingly, 2 instructions)

Example: BGR to RGB conversion

```
void convert(uchar* bgr, uchar* rgb, int length)
{
    for(int i = 0; i <= length - 16; i += 16)
    {
        uint8x16x3_t v = vld3q_u8(bgr + i*3);
        uint8x16_t tmp = v.val[0];
        v.val[0] = v.val[2];
        v.val[2] = tmp;
        vst3q_u8(rgb + i*3);
    }
}
```

Example: BGR to RGB conversion 2

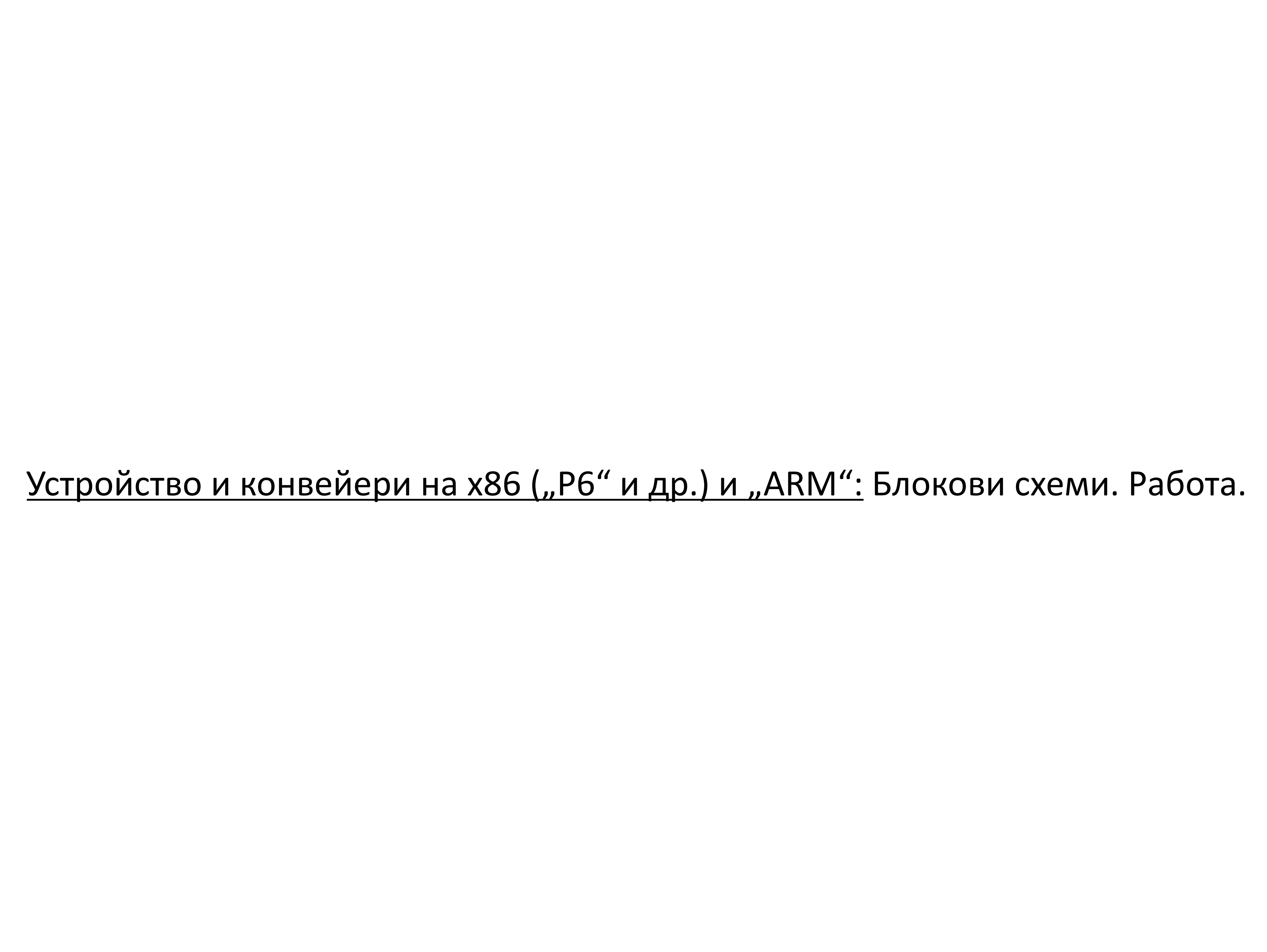
```
void convert(uchar* bgr, uchar* rgb, int length)
{
     for(int i = 0; i <= length - 16; i += 16)
     {
          __asm__ (
               "vld3.8 {d0, d2, d4}, [%[in0]]          \n\t"
               "vld3.8 {d1, d3, d5}, [%[in1]]          \n\t"
               "vswp q0, q2                            \n\t"
               "vst3.8 {d0, d2, d4}, [%[out0]]         \n\t"
               "vst3.8 {d1, d3, d5}, [%[out1]]         \n\t"
               : /*no output*/
               : [out0] "r" (dst + 3 * i),
                 [out1] "r" (dst + 3 * (i + 8)),
                 [in0]  "r" (src + 3 * i),
                 [in1]  "r" (src + 3 * (i + 8))
               : "d0","d1","d2","d3","d4","d5"
          );
     }
} // see http://hardwarebug.org/2010/07/06/arm-inline-asm-secrets/for magic registry codes
```

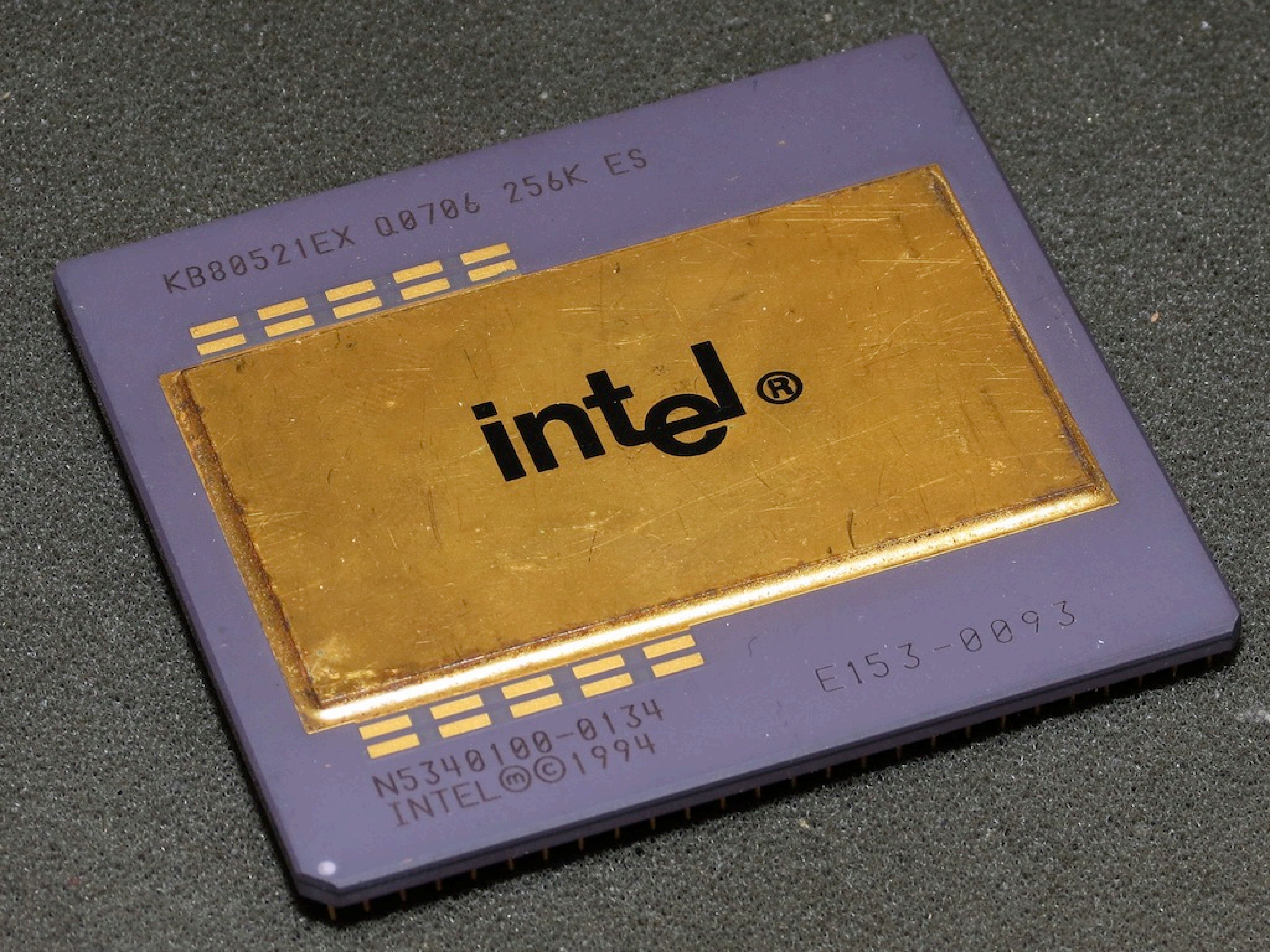
NEON vs SSE

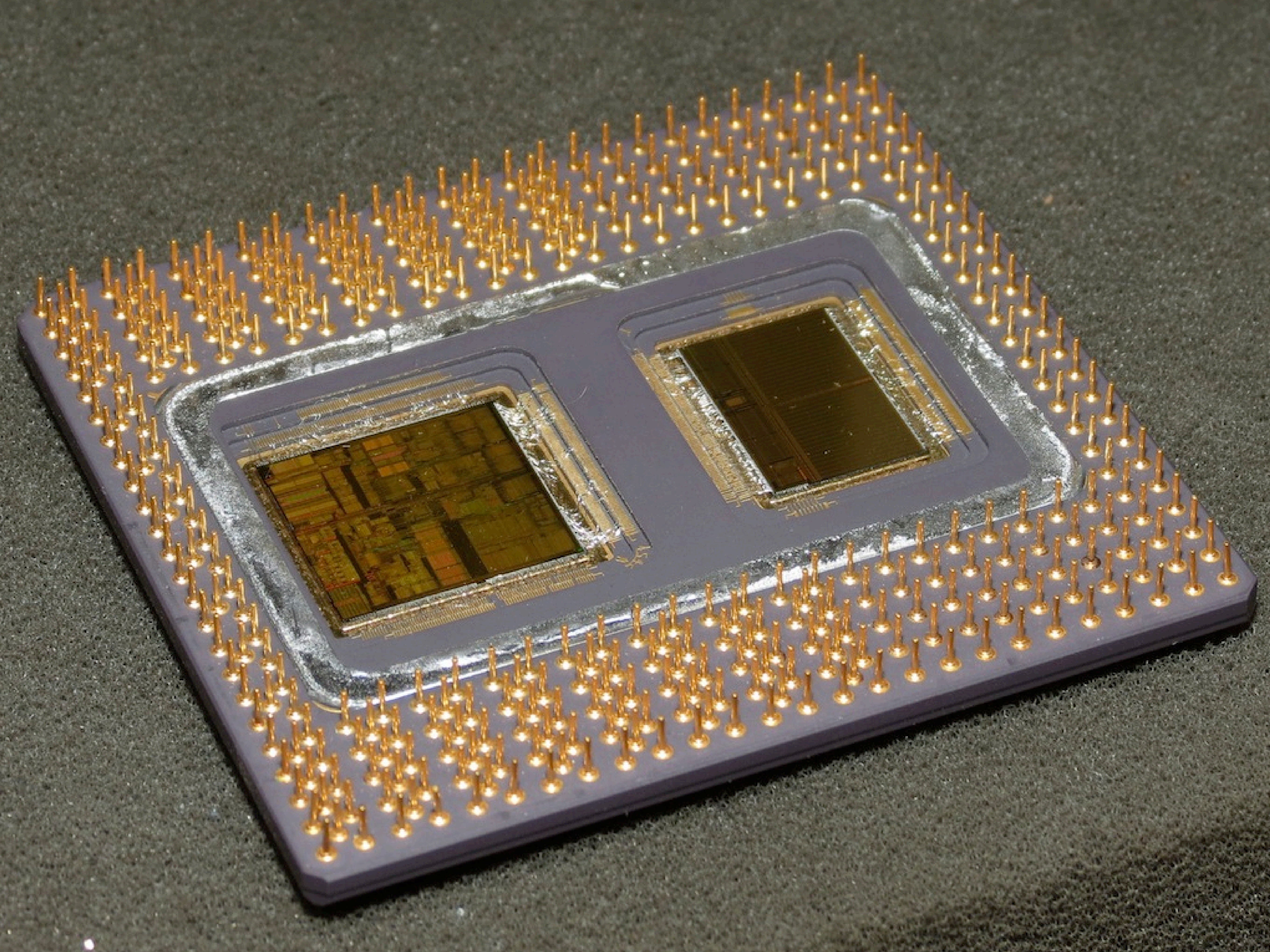
- Separate HW unit
- Strongly typed intrinsics
- 256 byte register bank

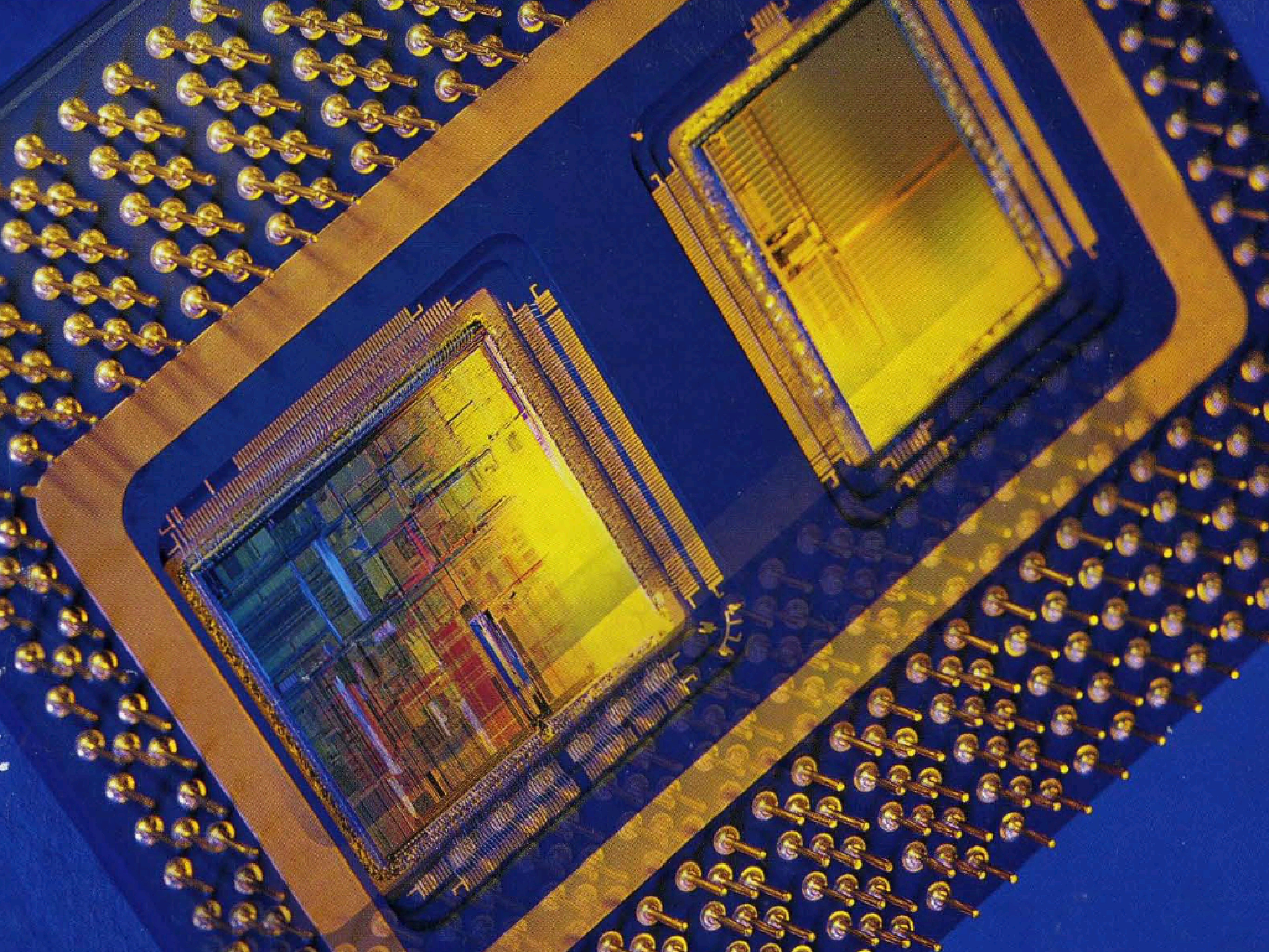
- No double precision
- No way to specify alignment for intrinsics
- Need software prefetch

- Same HW unit
- Type-aware intrinsics
- 128(x86)/256(x64) byte register bank
- Has double precision
- Special instructions for aligned load/store
- Good hardware prefetch


Устройство и конвейери на x86 („P6“ и др.) и „ARM“: Блокови схеми. Работа.


KB80521EX Q0706 256K ES
intel®
E153-0093
N5340100-0134
INTEL ⓜ © 1994





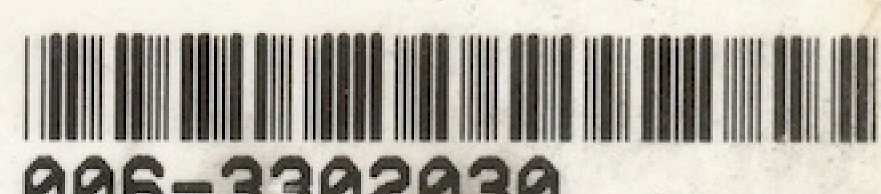
KB80521EX200 SL22Z 512K

intel®

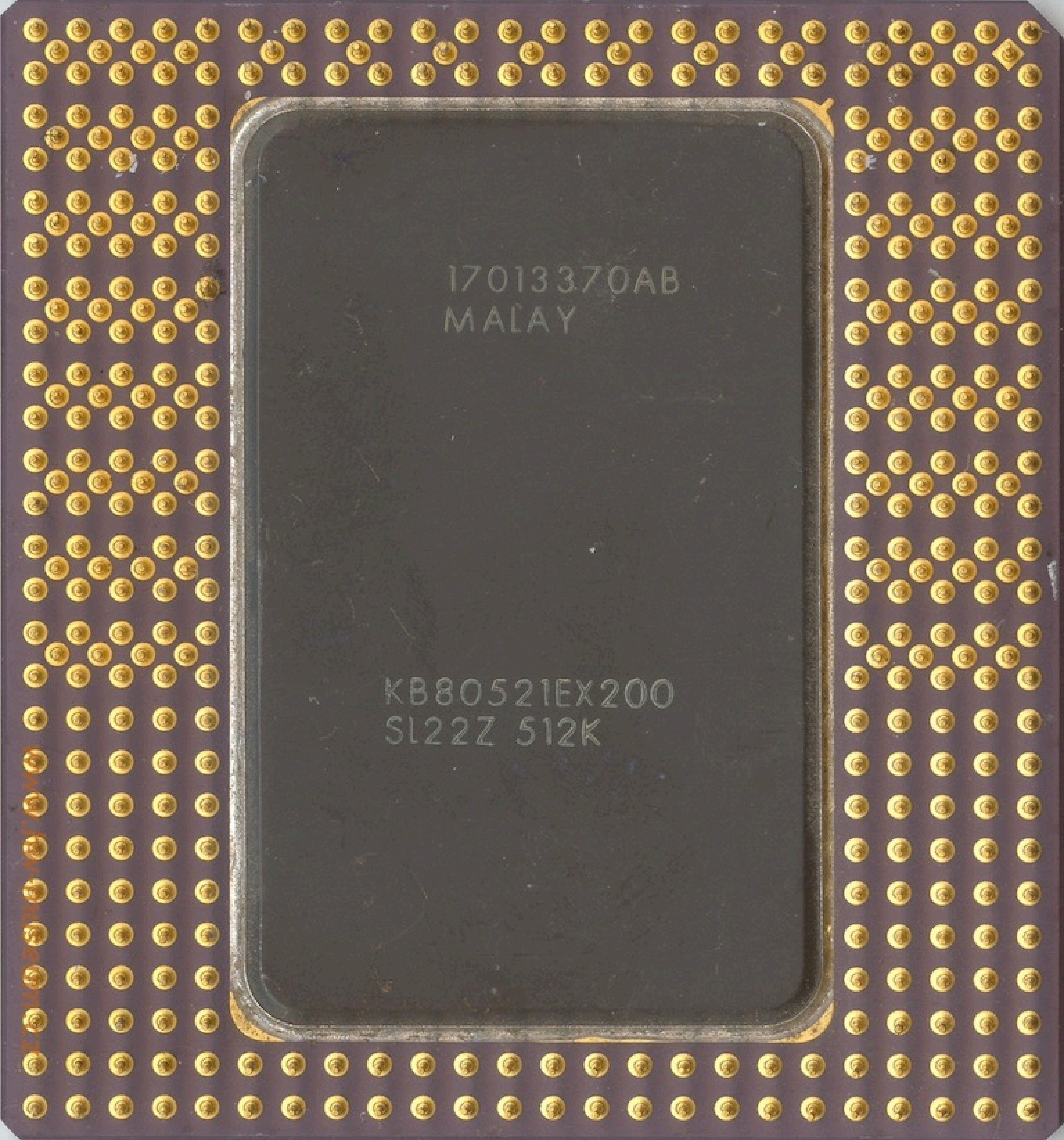
PENTIUM®PRO

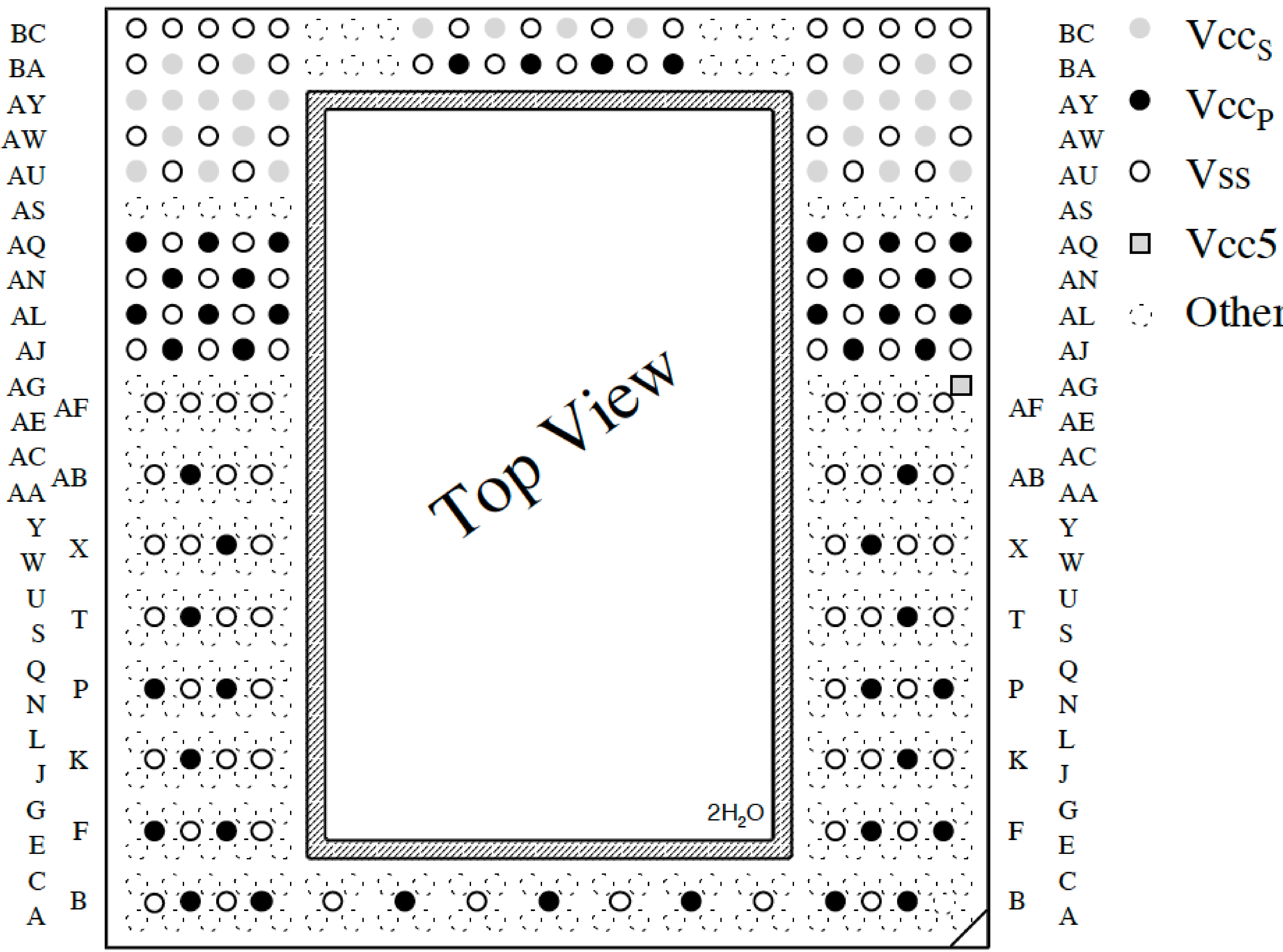
L7123466-1463

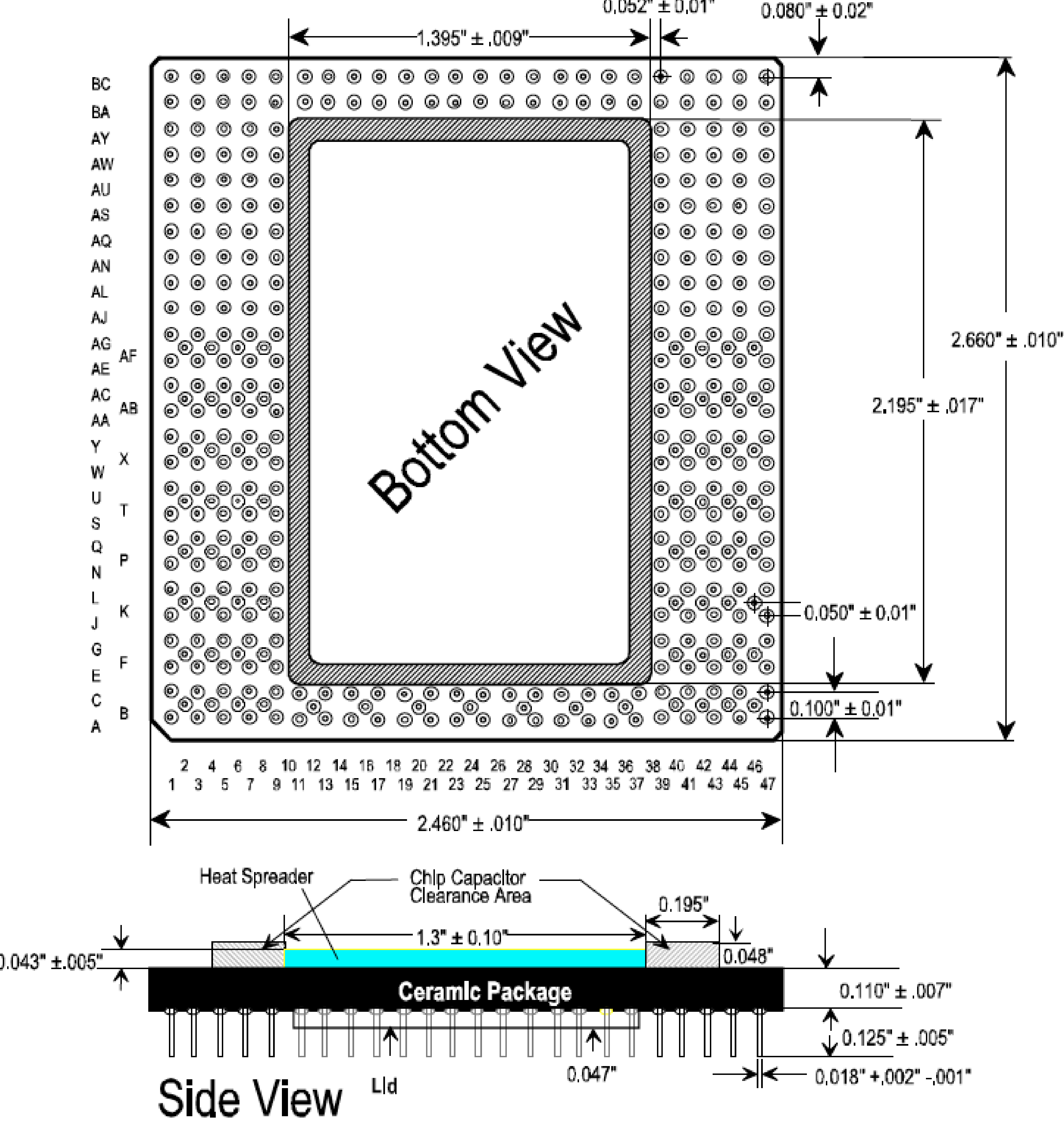
INTEL Ⓜ©'94'96



www.hw-museum.cz


17013370AB
MALAY
KB80521EX200
SL22Z 512K
www.hw-museum.cz


47 45 43 41 39 37 35 33 31 29 27 25 23 21 19 17 15 13 11 9 7 5 3 1
BC
BA
AY
AW
AU
AS
AQ
AN
AL
AJ
AG
AF
AE
AC
AB
AA
Y
X
W
U
T
S
Q
P
N
L
K
J
G
F
E
C
B
A
Top View
2H2O
46 44 42 40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2
Vcc_S
Vcc_P
Vss
Vcc5
Other



0.052" ± 0.01"
0.080" ± 0.02"
1.395" ± .009"
BC
BA
AY
AW
AU
AS
AQ
AN
AL
AJ
AG
AF
AE
AC
AB
AA
Y
X
W
U
T
S
Q
P
N
L
K
J
G
F
E
C
B
A
Bottom View
2.660" ± .010"
2.195" ± .017"
0.050" ± 0.01"
0.100" ± 0.01"
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
2.460" ± .010"
Heat Spreader
Chip Capacitor
Clearance Area
0.195"
1.3" ± 0.10"
0.043" ±.005"
0.048"
Ceramic Package
0.110" ± .007"
0.125" ± .005"
0.047"
0.018" +.002" -.001"
Side View
Lid

Table 27. Pentium® Pro Processor Package

Parameter	Value
Package Type	PGA
Total Pins	387
Pin Array	Modified Staggered
Package Size	2.66” x 2.46” (7.76cm x 6.25cm)
Heat Spreader Size	2.225” x 1.3” x 0.04” (5.65cm x 3.3cm x 0.1cm)
Approximate Weight	90 grams

VERY HIGH YEILD CPU's (CERAMIC)

NEC (server chip - R10000) - 0.27 g per CPU

Toshiba (server chip) - 0.27 g per CPU

AMD K5 – 0.4 g per CPU (0.5 g is pretty much speculated)

Pentium Pro (the holy grail of scammers) – 0.3 up to 0.5 g per CPU (1 g per CPU figure is speculated by "eBay scrap gold advocates"). Yield values differ depending on manufacturing plant & CPU's cache size.

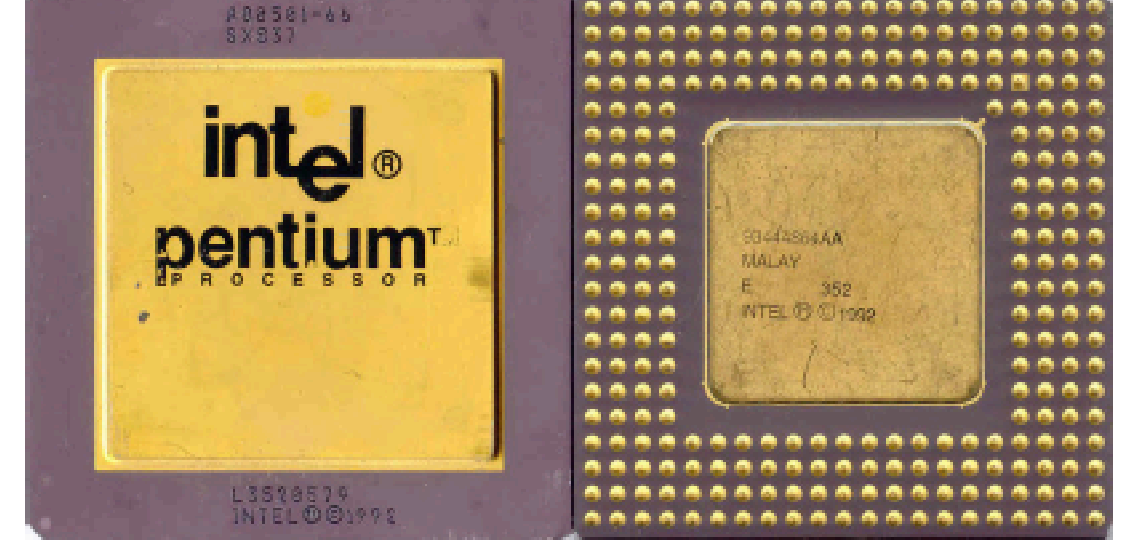
Cyrix 586 – 0.25 g per CPU

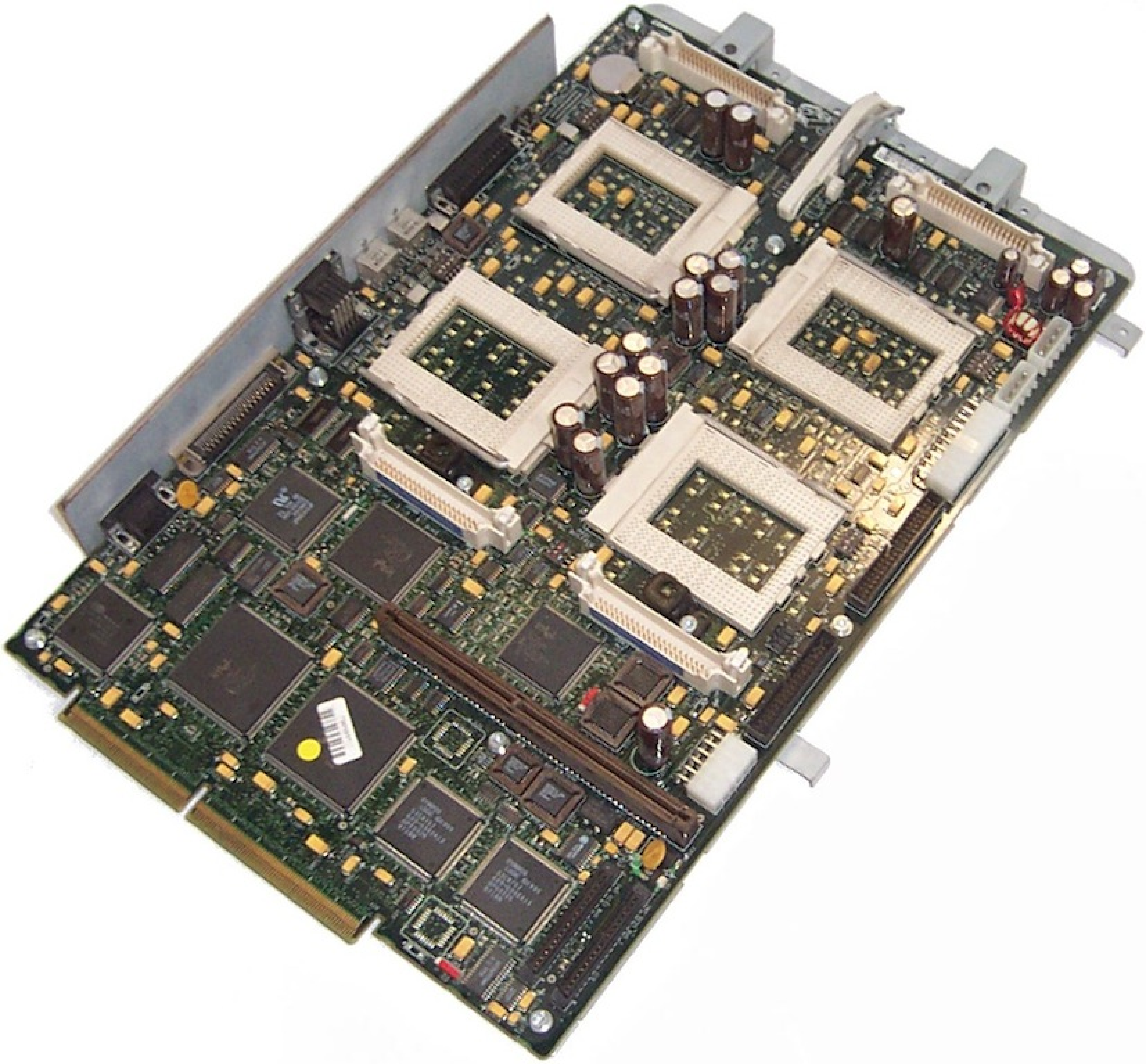
Cyrix 6×86-P166+GP 133MHz - 0.22 g per CPU

IBM 686 PR200 - 0.25 g per CPU

Original **Intel Pentium** 60Mhz - 90Mhz 0.48 g per CPU

Alpha DEC (large square, not smaller round heatsink) - 0.38 g Ag / 0.34 g Au per CPU





Frontside bus The P6 CPU had a bus to its L2 cache, and another bus to the rest of the system. For drawing convenience, the L2 was drawn on the P6's "backside" in block diagrams. The bus connecting P6 to other P6's and the chip set became known as the "frontside" bus to distinguish it from the L2 cache bus.

GTL (Gunning transceiver Logic)

- Invented by William Gunning while working for Xerox at the Palo Alto Research Center
- **GTL** is a type of logic signaling used to drive electronic backplane buses
- It has a voltage swing between 0.4 volts and 1.2 volts—much lower than that used in TTL and CMOS logic—and symmetrical parallel resistive termination
- All Intel front-side buses use GTL.

Table 2. Signal Groups

Group Name	Signals
GTL+ Input	BPRI#, BR[3:1]#[1], DEFER#, RESET#, RS[2:0]#, RSP#, TRDY#
GTL+ Output	PRDY#
GTL+ I/O	A[35:3]#, ADS#, AERR#, AP[1:0]#, BERR#, BINIT#, BNR#, BP[3:2]#, BPM[1:0]#, BR0#, D[63:0]#, DBSY#, DEP[7:0]#, DRDY#, FRCERR, HIT#, HITM#, LOCK#, REQ[4:0]#, RP#
3.3 V Tolerant Input	A20M#, FLUSH#, IGNNE#, INIT#, LINT0/INTR, LINT1/NMI, PREQ#, PWRGOOD[2], SMI#, STPCLK#
3.3 V Tolerant Output	FERR#, IERR#, THERMTRIP#[3]
Clock[4]	BCLK
APIC Clock[4]	PICCLK
APIC I/O[4]	PICD[1:0]
JTAG Input[4]	TCK, TDI, TMS, TRST#
JTAG Output[4]	TDO
Power/Other[5]	CPUPRES#, PLL1, PLL2, TESTHI, TESTLO, UP#, $V_{CC}P$, $V_{CC}S$, $V_{CC}5$, VID[3:0], $V_{REF}[7:0]$, V_{SS}

This appendix provides an alphabetical listing of all Pentium Pro processor signals. **Pins that do not appear here are not considered bus signals and are described in Table 2.**

A.1 A[35:3]# (I/O)

The A[35:3]# signals are the address signals. They are driven during the two-clock Request Phase by the request initiator. The signals in the two clocks are referenced Aa[35:3]# and Ab[35:3]#. During both clocks, A[35:24]# signals are protected with the AP1# parity signal, and A[23:3]# signals are protected with the AP0# parity signal.

The Aa[35:3]# signals are interpreted based on information carried during the first Request Phase clock on the REQa[4:0]# signals.

For memory transactions as defined by REQa[4:0]# = {XX01X,XX10X,XX11X}, the Aa[35:3]# signals define a 2^{36}-byte physical memory address space. The cacheable agents in the system observe the Aa[35:3]# signals and begin an internal snoop. The memory agents in the system observe the Aa[35:3]# signals and begin address decode to determine if they are responsible for the transaction completion. Aa[4:3]# signals define the critical word, the first data chunk to be transferred on the data bus. Cache line transactions use the standard burst order described in *Pentium® Pro Processor Developer's Manual, Volume 1: Specifications (Order* Number 242690) to transfer the remaining three data chunks.

For Pentium Pro processor IO transactions as defined by REQa[4:0]# = 1000X, the signals Aa[16:3]# define a 64K+3 byte physical IO space. The IO agents in the system observe the signals and begin address decode to determine if they are responsible for the transaction completion. Aa[35:17]# are always zero. Aa16# is zero unless the IO space being accessed is the first three bytes of a 64KByte address range.

For deferred reply transactions as defined by REQa[4:0]# = 00000, Aa[23:16]# carry the deferred ID. This signal is the same deferred ID supplied by the request initiator of the original transaction on Ab[23:16]#/DID[7:0]# signals. Pentium Pro processor bus agents that support deferred replies sample the deferred ID and perform an internal match against any outstanding transactions waiting for deferred replies. During a deferred reply, Aa[35:24]# and Aa[15:3]# are reserved.

For the branch-trace message transaction as defined by REQa[4:0]# = 01001 and for special and interrupt acknowledge transactions, as defined by REQa[4:0]# = 01000, the Aa[35:3]# signals are reserved and undefined.

During the second clock of the Request Phase, Ab[35:3]# signals perform identical signal functions for all transactions. For ease of description, these functions are described using new signal names. Ab[31:24]# are renamed the attribute signals ATTR[7:0]#. Ab[23:16]# are renamed the Deferred ID signals DID[7:0]#. Ab[15:8]# are renamed the eight-byte enable signals BE[7:0]#. Ab[7:3]# are renamed the extended function signals EXF[4:0]#.

Table 43. Request Phase Decode

Ab[31:24]#	Ab[23:16]#	Ab[15:8]#	Ab[7:3]#
ATTR[7:0]#	DID[7:0]#	BE[7:0]#	EXF[4:0]#

On the active-to-inactive transition of RESET#, each Pentium Pro processor bus agent samples A[35:3]# signals to determine its power-on configuration.

A.2 A20M# (I)

The A20M# signal is the address-20 mask signal in the PC Compatibility group. If the A20M# input signal is asserted, the Pentium Pro processor masks physical address bit 20 (A20#) before looking up a line in any internal cache and before driving a read/write transaction on the bus. Asserting A20M# emulates the 8086 processor's address wrap around at the one Mbyte boundary. Only assert A20M# when the processor is in real mode. The effect of asserting A20M# in protected mode is undefined and may be implemented differently in future processors.

Snoop requests and cache-line writeback transactions are unaffected by A20M# input. Address 20 is not masked when the processor samples external addresses to perform internal snooping.

A20M# is an asynchronous input. However, to guarantee recognition of this signal following an I/O write instruction, A20M# must be valid with active RS[2:0]# signals of the corresponding I/O Write bus transaction. In FRC mode, A20M# must be synchronous to BCLK.

During active RESET#, the Pentium Pro processor begins sampling the A20M#, IGNNE#, and LINT[1:0] values to determine the ratio of core-clock frequency to bus-clock frequency. After the PLL-lock time, the core clock becomes stable and is locked to the external bus clock. On the active-to-inactive transition of RESET#, the Pentium Pro processor latches A20M#, IGNNE#, and LINT[1:0] and freezes the frequency ratio internally. 29See Table 44.

A.3 ADS# (I/O)

The ADS# signal is the address Strobe signal. It is asserted by the current bus owner for one clock to indicate a new Request Phase. A new Request Phase can only begin if the In-order Queue has less than the maximum number of entries defined by the power-on configuration (1 or 8), the Request Phase is not being stalled by an active BNR# sequence and the ADS# associated with the previous Request Phase is sampled inactive. Along with the ADS#, the request initiator drives A[35:3]#, REQ[4:0]#, AP[1:0]#, and RP# signals for two clocks. During the second Request Phase clock, ADS# must be inactive. RP# provides parity protection for REQ[4:0]# and ADS# signals during both clocks. If the transaction is part of a bus locked operation, LOCK# must be active with ADS#.

If the request initiator continues to own the bus after the first Request Phase, it can issue a new request every three clocks. If the request initiator needs to release the bus ownership after the Request Phase, it can deactivate its BREQn#/ BPRI# arbitration signal as early as with the activation of ADS#.

All bus agents observe the ADS# activation to begin parity checking, protocol checking, address decode, internal snoop, or deferred reply ID match operations associated with the new transaction. On sampling the asserted ADS#, all agents load the new transaction in the In-order Queue and update internal counters. The Error, Snoop, Response, and Data Phase of the transaction are defined with respect to ADS# assertion.

Table 44. Bus Clock Ratios Versus Pin Logic Levels

Ratio of Core Clock to Bus Clock	LINT[1]/NMI	LINT[0]/INTR	IGNNE#	A20M#
2	L	L	L	L
2	H	H	H	H
3	L	L	H	L
4	L	L	L	H
RESERVED	L	L	H	H
5/2	L	H	L	L
7/2	L	H	H	L
RESERVED	L	H	L	H
RESERVED	L	H	H	H
RESERVED	ALL OTHER COMBINATIONS			

A.4 AERR# (I/O)

The AERR# signal is the address parity error signal. Assuming the AERR# driver is enabled during the power-on configuration, a bus agent can drive AERR# active for exactly one clock during the Error Phase of a transaction. AERR# must be inactive for a minimum of two clocks. The Error Phase is always three clocks from the beginning of the Request Phase.

On observing active ADS#, all agents begin parity and protocol checks for the signals valid in the two Request Phase clocks. Parity is checked on AP[1:0]# and RP# signals. AP1# protects A[35:24]#, AP0# protects A[23:3]# and RP# protects REQ[4:0]#. A parity error without a protocol violation is signaled by AERR# assertion.

If AERR# observation is enabled during power-on configuration, AERR# assertion in a valid Error Phase aborts the transaction. All bus agents remove the transaction from the In-order Queue and update internal counters. The Snoop Phase, Response Phase, and Data Phase of the transaction are aborted. All signals in these phases must be deasserted two clocks after AERR# is asserted, even if the signals have been asserted before AERR# has been observed. Specifically if the Snoop Phase associated with the aborted transaction is driven in the next clock, the snoop results, including a STALL condition (HIT# and HITM# asserted for one clock), are ignored. All bus agents must also begin an arbitration reset sequence and deassert BREQn#/BPRI# arbitration signals on sampling AERR# active. A current bus owner in the middle of a bus lock operation must keep LOCK# asserted and assert its arbitration request BPRI#/BREQn# after keeping it inactive for two clocks to retain its bus ownership and guarantee lock atomicity. All other agents, including the current bus owner not in the middle of a bus lock operation, must wait at least 4 clocks before asserting BPRI#/BREQn# and beginning a new arbitration.

If AERR# observation is enabled, the request initiator can retry the transaction up to n times until it reaches the retry limit defined by its implementation. (The Pentium Pro processor retries once.) After n retries, the request initiator treats the error as a hard error. The request initiator asserts BERR# or enters the Machine Check Exception handler, as defined by the system configuration.

If AERR# observation is disabled during power-on configuration, AERR# assertion is ignored by all bus agents except a central agent. Based on the Machine Check Architecture of the system, the central agent can ignore AERR#, assert NMI to execute NMI handler, or assert BINIT# to reset the bus units of all agents and execute an MCE handler.

A.5 AP[1:0]# (I/O)

The AP[1:0]# signals are the address parity signals. They are driven by the request initiator during the two Request Phase clocks along with ADS#, A[35:3]#, REQ[4:0]#, and RP#. AP1# covers A[35:24]#. AP0# covers A[23:3]#. A correct parity signal is high if an even number of covered signals are low and low if an odd number of covered signals are low. This rule allows parity to be high when all the covered signals are high.

Provided "AERR# drive" is enabled during the power-on configuration, all bus agents begin parity checking on observing active ADS# and determine if there is a parity error. On observing a parity error on any one of the two Request Phase clocks, the bus agent asserts AERR# during the Error Phase of the transaction.

A.6 ASZ[1:0]# (I/O)

The ASZ[1:0]# signals are the memory address-space size signals. They are driven by the request initiator during the first Request Phase clock on the REQa[4:3]# pins. The ASZ[1:0]# signals are valid only when REQa[1:0]# signals equal 01B, 10B, or 11B, indicating a memory access transaction. The ASZ[1:0]# decode is defined in Table 45.

Table 45. ASZ[1:0]# Signal Decode

ASZ[1:0]#		Description
0	0	0 <= A[35:3]# < 4 GB
0	1	4 GB <= A[35:3]# < 64 GB
1	X	Reserved

If the memory access is within the 0-to-(4GByte -1) address space, ASZ[1:0]# must be 00B. If the memory access is within the 4Gbyte-to-(64GByte -1) address space, ASZ[1:0]# must be 01B. All observing bus agents that support the 4Gbyte (32 bit) address space must respond to the transaction only

when ASZ[1:0]# equals 00. All observing bus agents that support the 64GByte (36- bit) address space must respond to the transaction when ASZ[1:0]# equals 00B or 01B.

A.7 ATTR[7:0]# (I/O)

The ATTR[7:0]# signals are the attribute signals. They are driven by the request initiator during the second Request Phase clock on the Ab[31:24]# pins. The ATTR[7:0]# signals are valid for all transactions. The ATTR[7:3]# are reserved and undefined. The ATTR[2:0]# are driven based on the Memory Range Register attributes and the Page Table attributes. Table 47. defines ATTR[3:0]# signals.

A.8 BCLK (I)

The BCLK (clock) signal is the Execution Control group input signal. It determines the bus frequency. All agents drive their outputs and latch their inputs on the BCLK rising edge.

The BCLK signal indirectly determines the Pentium Pro processor's internal clock frequency. Each Pentium Pro processor derives its internal clock from BCLK by multiplying the BCLK frequency by a ratio as defined and allowed by the power-on configuration. See Table 42.

All external timing parameters are specified with respect to the BCLK signal.

A.9 BE[7:0]# (I/O)

The BE[7:0]# signals are the byte-enable signals. They are driven by the request initiator during the second Request Phase clock on the Ab[15:8]# pins. These signals carry various information depending on the REQ[4:0]# value.

For memory or I/O transactions (REQa[4:0]# = {10000B, 10001B, XX01XB, XX10XB, XX11XB}) the byte-enable signals indicate that valid data is requested or being transferred on the corresponding byte on the 64 bit data bus. BE0# indicates D[7:0]# is valid, BE1# indicates D[15:8]# is valid,..., BE7# indicates D[63:56]# is valid.

For Special transactions ((REQa[4:0]# = 01000B) and (REQb[1:0]# = 01B)), the BE[7:0]# signals carry special cycle encodings as defined in Table 46. All other encodings are reserved.

Table 46. Special Transaction Encoding on BE[7:0]#

BE[7:0]#	Special Cycle
0000 0000	Reserved
0000 0001	Shutdown
0000 0010	Flush
0000 0011	Halt
0000 0100	Sync
0000 0101	Flush Acknowledge
00000 0110	Stop Clock Acknowledge
00000 0111	SMI Acknowledge
Other	Reserved

For Deferred Reply, Interrupt Acknowledge, and Branch Trace Message transactions, the BE[7:0]# signals are undefined.

Table 47. ATTR[7:0]# Field Descriptions

ATTR[7:3]#	ATTR[2]#	ATTR[1:0]#			
XXXXX	X	11	10	01	00
Reserved	Potentially Speculatable	Write-Back	Write-Protect	Write-Through	UnCacheable

A.10 BERR# (I/O)

The BERR# signal is the Error group Bus Error signal. It is asserted to indicate an unrecoverable error without a bus protocol violation.

The BERR# protocol is as follows: If an agent detects an unrecoverable error for which BERR# is a valid error response and BERR# is sampled inactive, it asserts BERR# for three clocks. An agent can assert BERR# only after observing that the signal is inactive. An agent asserting BERR# must deassert the signal in two clocks if it observes that another agent began asserting BERR# in the previous clock.

BERR# assertion conditions are defined by the system configuration. Configuration options enable the BERR# driver as follows:

- Enabled or disabled
- Asserted optionally for internal errors along with IERR#
- Optionally asserted by the request initiator of a bus transaction after it observes an error
- Asserted by any bus agent when it observes an error in a bus transaction

BERR# sampling conditions are also defined by the system configuration. Configuration options enable the BERR# receiver to be enabled or disabled. When the bus agent samples an active BERR# signal and if MCE is enabled, the Pentium Pro processor enters the Machine Check Handler. If MCE is disabled, typically the central agent forwards BERR# as an NMI to one of the processors. The Pentium Pro processor does not support BERR# sampling (always disabled).

A.11 BINIT# (I/O)

The BINIT# signal is the bus initialization signal. If the BINIT# driver is enabled during the power on configuration, BINIT# is asserted to signal any bus condition that prevents reliable future information.

The BINIT# protocol is as follows: If an agent detects an error for which BINIT# is a valid error response, and BINIT# is sampled inactive, it asserts BINIT# for three clocks. An agent can assert BINIT# only after observing that the signal is inactive. An agent asserting BINIT# must deassert the signal in two clocks if it observes that another agent began asserting BINIT# in the previous clock.

If BINIT# observation is enabled during power-on configuration, and BINIT# is sampled asserted, all bus state machines are reset. All agents reset their rotating ID for bus arbitration to the state after reset, and internal count information is lost. The L1 and L2 caches are not affected.

If BINIT# observation is disabled during power-on configuration, BINIT# is ignored by all bus agents except a central agent that must handle the error in a manner appropriate to the system architecture.

A.12 BNR# (I/O)

The BNR# signal is the Block Next Request signal in the Arbitration group. The BNR# signal is used to assert a bus stall by any bus agent who is unable to accept new bus transactions to avoid an internal transaction queue overflow. During a bus stall, the current bus owner cannot issue any new transactions.

Since multiple agents might need to request a bus stall at the same time, BNR# is a wire-OR signal. In order to avoid wire-OR glitches associated with simultaneous edge transitions driven by multiple drivers, BNR# is activated on specific clock edges and sampled on specific clock edges. A valid bus stall involves assertion of BNR# for one clock on a well-defined clock edge (T1), followed by de-assertion of BNR# for one clock on the next clock edge (T1+1). BNR# can first be sampled on the second clock edge (T1+1) and must always be ignored on the third clock edge (T1+2). An extension of a bus stall requires one clock active (T1+2), one clock inactive (T1+3) BNR# sequence with BNR# sampling points every two clocks (T1+1, T1+3,...).

After the RESET# active-to-inactive transition, bus agents might need to perform hardware initialization of their bus unit logic. Bus agents intending to create a request stall must assert BNR# in the clock after RESET# is sampled inactive.

After BINIT# assertion, all bus agents go through a similar hardware initialization and can create a request stall by asserting BNR# four clocks after BINIT# assertion is sampled.

On the first BNR# sampling clock that BNR# is sampled inactive, the current bus owner is allowed to issue one new request. Any bus agent can immediately reassert BNR# (four clocks from the previous assertion or two clocks from the previous de-assertion) to create a new bus stall. This throttling

mechanism enables independent control on every new request generation.

If BNR# is deasserted on two consecutive sampling points, new requests can be freely generated on the bus. After receiving a new transaction, a bus agent can require an address stall due to an anticipated transaction-queue overflow condition. In response, the bus agent can assert BNR#, three clocks from active ADS# assertion and create a bus stall. Once a bus stall is created, the bus remains stalled until BNR# is sampled asserted on subsequent sampling points.

A.13 BP[3:2]# (I/O)

The BP[3:2]# signals are the System Support group Breakpoint signals. They are outputs from the Pentium Pro processor that indicate the status of breakpoints.

A.14 BPM[1:0]# (I/O)

The BPM[1:0]# signals are more System Support group breakpoint and performance monitor signals. They are outputs from the Pentium Pro processor that indicate the status of breakpoints and programmable counters used for monitoring Pentium Pro processor performance.

A.15 BPRI# (I)

The BPRI# signal is the Priority-agent Bus Request signal. The priority agent arbitrates for the bus by asserting BPRI#. The priority agent is always be the next bus owner. Observing BPRI# active causes the current symmetric owner to stop issuing new requests, unless such requests are part of an ongoing locked operation.

If LOCK# is sampled inactive two clocks from BPRI# driven asserted, the priority agent can issue a new request within four clocks of asserting BPRI#. The priority agent can further reduce its arbitration latency to two clocks if it samples active ADS# and inactive LOCK# on the clock in which BPRI# was driven active and to three clocks if it samples active ADS# and inactive LOCK# on the clock in which BPRI# was sampled active. If LOCK# is sampled active, the priority agent must wait for LOCK# deasserted and gains bus ownership in two clocks after LOCK# is sampled deasserted. The priority agent can keep BPRI# asserted until all of its requests are completed and can release the bus by de-asserting BPRI# as early as the same clock edge on which it issues the last request.

On observation of active AERR#, RESET#, or BINIT#, BPRI# must be deasserted in the next clock. BPRI# can be reasserted in the clock after sampling the RESET# active-to-inactive transition or three clocks after sampling BINIT# active and RESET# inactive. On AERR# assertion, if the priority agent is in the middle of a bus-locked operation, BPRI# must be re-asserted after two clocks, otherwise BPRI# must stay inactive for at least 4 clocks.

After the RESET# inactive transition, Pentium Pro processor bus agents begin BPRI# and BNR# sampling on BNR# sample points. When both BNR# and BPRI# are observed inactive on a BNR# sampling point, the APIC units in Pentium Pro processors on a common APIC bus are synchronized.

A.16 BR0#(I/O), BR[3:1]# (I)

The BR[3:0]# pins are the physical bus request pins that drive the BREQ[3:0]# signals in the system. The BREQ[3:0]# signals are interconnected in a rotating manner to individual processor pins. Table 48 gives the rotating interconnect between the processor and bus signals.

Table 48. BR[3:0]# Signals Rotating Interconnect

Bus Signal	Agent 0 Pins	Agent 1 Pins	Agent 2Pins	Agent 3 Pins
BREQ0#	BR0#	BR3#	BR2#	BR1#
BREQ1#	BR1#	BR0#	BR3#	BR2#
BREQ2#	BR2#	BR1#	BR0#	BR3#
BREQ3#	BR3#	BR2#	BR1#	BR0#

During power-up configuration, the central agent must assert the BR0# bus signal. All symmetric agents sample their BR[3:0]# pins on active-to-inactive transition of RESET#. The pin on which the agent samples an active level determines its agent ID. All agents then configure their pins to match the appropriate bus signal protocol, as shown in Table 49.

Table 49. BR[3:0]# Signal Agent IDs

Pin Sampled Active on RESET#	Agent ID
BR0#	0
BR3#	1
BR2#	2
BR1#	3

A.17 BREQ[3:0]# (I/O)

The BREQ[3:0]# signals are the Symmetric-agent Arbitration Bus signals (called bus request). A symmetric agent n arbitrates for the bus by asserting its BREQn# signal. Agent n drives BREQn# as an output and receives the remaining BREQ[3:0]# signals as inputs.

The symmetric agents support distributed arbitration based on a round-robin mechanism. The rotating ID is an internal state used by all symmetric agents to track the agent with the lowest priority at the next arbitration event. At power-on, the rotating ID is initialized to three, allowing agent 0 to be the highest priority symmetric agent. After a new arbitration event, the rotating ID of all symmetric agents is updated to the agent ID of the symmetric owner. This update gives the new symmetric owner lowest priority in the next arbitration event.

A new arbitration event occurs either when a symmetric agent asserts its BREQn# on an Idle bus (all BREQ[3:0]# previously inactive), or the current symmetric owner de-asserts BREQm# to release the bus ownership to a new bus owner n. On a new arbitration event, based on BREQ[3:0]#, and the rotating ID, all symmetric agents simultaneously determine the new symmetric owner. The symmetric owner can park on the bus (hold the bus) provided that no other symmetric agent is requesting its use. The symmetric owner parks by keeping its BREQn# signal active. On sampling active BREQm# asserted by another symmetric agent, the symmetric owner de-asserts BREQn# as soon as possible to release the bus. A symmetric owner stops issuing new requests that are not part of an existing locked operation upon observing BPRI# active.

A symmetric agent can not deassert BREQn# until it becomes a symmetric owner. A symmetric agent can reassert BREQn# after keeping it inactive for one clock.

On observation of active AERR#, RESET#, or BINIT#, the BREQ[3:0]# signals must be deasserted in the next clock. BREQ[3:0]# can be reasserted in the clock after sampling the RESET# active-to-inactive transition or three clocks after sampling BINIT# active and RESET# inactive. On AERR# assertion, if bus agent n is in the middle of a bus-locked operation, BREQn# must be re-asserted after two clocks, otherwise BREQ[3:0]# must stay inactive for at least 4 clocks.

A.18 D[63:0]# (I/O)

The D[63:0]# signals are the data signals. They are driven during the Data Phase by the agent responsible for driving the data. These signals provide a 64-bit data path between various Pentium Pro processor bus agents. 32-byte line transfers

require four data transfer clocks with valid data on all eight bytes. Partial transfers require one data transfer clock with valid data on the byte(s) indicated by active byte enables BE[7:0]#. Data signals not valid for a particular transfer must still have correct ECC (if data bus ECC is selected). If BE0# is asserted, D[7:0]# transfers the least significant byte. If BE7# is asserted, D[63:56]# transfers the most significant byte.

The data driver asserts DRDY# to indicate a valid data transfer. If the Data Phase involves more than one clock the data driver also asserts DBSY# at the beginning of the Data Phase and de-asserts DBSY# no earlier than on the same clock that it performs the last data transfer.

A.19 DBSY# (I/O)

The DBSY# signal is the Data-bus Busy signal. It indicates that the data bus is busy. It is asserted by the agent responsible for driving the data during the Data Phase, provided the Data Phase involves more than one clock. DBSY# is asserted at the beginning of the Data Phase and may be deasserted on or after the clock on which the last data is driven. The data bus is released one clock after DBSY# is deasserted.

When normal read data is being returned, the Data Phase begins with the Response Phase. Thus the agent returning read data can assert DBSY# when the transaction reaches the top of the In-order Queue and it is ready to return response on RS[2:0]# signals. In response to a write request, the agent driving the write data must drive DBSY# active after the write transaction reaches the top of the In-order Queue and it sees active TRDY# with inactive DBSY# indicating that the target is ready to receive data. For an implicit writeback response, the snoop agent must assert DBSY# active after the target memory agent of the implicit writeback asserts TRDY#. Implicit writeback TRDY# assertion begins after the transaction reaches the top of the In-order Queue, and TRDY# de-assertion associated with the write portion of the transaction, if any is completed. In this case, the memory agent guarantees assertion of implicit writeback response in the same clock in which the snooping agent asserts DBSY#.

A.20 DEFER# (I)

The DEFER# signal is the defer signal. It is asserted by an agent during the Snoop Phase to indicate that the transaction cannot be guaranteed in-order completion. Assertion of DEFER# is normally the responsibility of the addressed memory agent or I/O agent. For systems that involve resources on a system bus other than the Pentium Pro processor bus, a bridge agent can accept the DEFER# assertion responsibility on behalf of the addressed agent.

When HITM# and DEFER# are both active during the Snoop Phase, HITM# is given priority and the transaction must be completed with implicit writeback response. If HITM# is inactive, and DEFER# active, the agent asserting DEFER# must complete the transaction with a Deferred or Retry response.

If DEFER# is inactive, or HITM# is active, then the transaction is committed for in-order completion and snoop ownership is transferred normally between the requesting agent, the snooping agents, and the response agent.

If DEFER# is active with HITM# inactive, the transaction commitment is deferred. If the defer agent completes the transaction with a retry response, the requesting agent must retry the transaction. If the defer agent returns a deferred response, the requesting agent must freeze snoop state transitions associated with the deferred transaction and issues of new order-dependent transactions until the corresponding deferred reply transaction. In the meantime, the ownership of the deferred address is transferred to the defer agent and it must guarantee management of conflicting transactions issued to the same address.

If DEFER# is active in response to a newly issued bus-lock transaction, the entire bus-locked operation is re-initiated regardless of HITM#. This feature is useful for a bridge agent in response to a split bus-locked operation. It is recommended that the bridge agent extend the Snoop Phase of the first transaction in a split locked operation until it can either guarantee ownership of all system resources to enable successful completion of the split sequence or assert DEFER# followed by a Retry Response to abort the split sequence.

A.21 DEN# (I/0)

The DEN# signal is the defer-enable signal. It is driven to the bus on the second clock of the Request Phase on the EXF1#/Ab4# pin. DEN# is asserted to indicate that the transaction can be deferred by the responding agent.

A.22 DEP[7:0]# (I/O)

The DEP[7:0]# signals are the data bus ECC protection signals. They are driven during the Data Phase by the agent responsible for driving D[63:0]#. The DEP[7:0]# signals provide optional ECC protection for the data bus. During power-on configuration, DEP[7:0]# signals can be enabled for either ECC checking or no checking.

The ECC error correcting code can detect and correct single-bit errors and detect double-bit or nibble errors. The *Pentium® Pro Processor Developer's Manual, Volume 1: Specifications* (Order Number 242690) provides more information about ECC.

DEP[7:0]# provide valid ECC for the entire data bus on each data clock, regardless of which bytes are valid. If checking is enabled, receiving agents check the ECC signals for all 64 data signals.

A.23 DID[7:0]# (I/O)

The DID[7:0]# signals are Deferred Identifier signals. They are transferred using A[23:16]# signals by the request initiator. They are transferred on Ab[23:16]# during the second clock of the Request Phase on all transactions, but only defined for deferrable transactions (DEN# asserted). DID[7:0]# is also transferred on Aa[23:16]# during the first clock of the Request Phase for Deferred Reply transactions.

The deferred identifier defines the token supplied by the request initiator. DID[7:4]# carry the request initiators' agent identifier and DID[3:0]# carry a transaction identifier associated with the request. This configuration limits the bus specification to 16 bus masters with each one of the bus masters capable of making up to sixteen requests.

Every deferrable transaction issued on the Pentium Pro processor bus which has not been guaranteed completion (has not successfully passed its Snoop Result Phase) will have a unique Deferred ID. This includes all outstanding transactions which have not had their snoop result reported, or have had their snoop results deferred. After a deferrable transaction passes its Snoop Result Phase without DEFER# asserted, its Deferred ID may be reused. Similarly, the deferred ID of a transaction which was deferred may be reused after the completion of the snoop window of the deferred reply.

DID[7]# indicates the agent type. Symmetric agents use 0. Priority agents use 1. DID[6:4]# indicates the agent ID. Symmetric agents use their arbitration ID. The Pentium Pro processor has four symmetric agents, so does not assert DID[6]#. DID[3:0]# indicates the transaction ID for an agent. The transaction ID must be unique for all transactions issued by an agent which have not reported their snoop results.

Table 50. DID[7:0]# Encoding

DID[7]	DID[6:4]	DID[3:0]
Agent Type	Agent ID	Transaction ID

The Deferred Reply agent transmits the DID[7:0]# (Ab[23:16]#) signals received during the original transaction on the Aa[23:16]# signals during the Deferred Reply transaction. This process enables the original request initiator to make an identifier match and wake up the original request waiting for completion.

A.24 DRDY# (I/O)

The DRDY# signal is the Data Phase data-ready signal. The data driver asserts DRDY# on each data transfer, indicating valid data on the data bus. In a multi-cycle data transfer, DRDY# can be deasserted to insert idle clocks in the Data Phase. During a line transfer, DRDY# is active for four clocks. During a partial 1-to-8 byte transfer, DRDY# is active for one clock. If a data transfer is exactly one clock, then the entire Data Phase may consist of only one clock active DRDY# and inactive DBSY#. If DBSY# is asserted for a 1-to-8 byte transfer, then the data bus is not released until one clock after DBSY# is deasserted.

A.25 DSZ[1:0]# (I/O)

The DSZ[1:0]# signals are the data-size signals. They are transferred on REQb[4:3]# signals in the second clock of Request Phase by the requesting agent. The DSZ[1:0]# signals define the data transfer

capability of the requesting agent. For the Pentium Pro processor, DSZ#= 00, always.

A.26 EXF[4:0]# (I/O)

The EXF[4:0]# signals are the Extended Function signals and are transferred on the Ab[7:3]# signals by the request initiator during the second clock of the Request Phase. The signals specify any special functional requirement associated with the transaction based on the requester mode or capability. The signals are defined in Table 51.

Table 51. EXF[4:0]# Signal Definitions

EXF	NAME	External Functionality	When Activated
EXF4#	SMMEM#	SMM Mode	After entering SMM mode
EXF3#	SPLCK#	Split Lock	The first transaction of a split bus lock operation
EXF2#	Reserved	Reserved	
EXF1#	DEN#	Defer Enable	The transactions for which Defer or Retry Response is acceptable.
EXF0#	Reserved	Reserved	

A.27 FERR# (O)

The FERR# signal is the PC Compatibility group Floating-point Error signal. The Pentium Pro processor asserts FERR# when it detects an unmasked floating-point error. FERR# is similar to the ERROR# signal on the Intel387™ coprocessor. FERR# is included for compatibility with systems using DOS-type floating-point error reporting.

A.28 FLUSH# (I)

When the FLUSH# input signal is asserted, the Pentium Pro processor bus agent writes back all internal cache lines in the Modified state and invalidates all internal cache lines. At the completion of a flush operation, the Pentium Pro processor issues a Flush Acknowledge transaction to indicate that the cache flush operation is complete. The Pentium Pro processor stops caching any new data while the FLUSH# signal remains asserted.

FLUSH# is an asynchronous input. However, to guarantee recognition of this signal following an I/O write instruction, FLUSH# must be valid along with RS[2:0]# in the Response Phase of the corresponding I/O Write bus transaction. In FRC mode, FLUSH# must be synchronous to BCLK.

On the active-to-inactive transition of RESET#, each Pentium Pro processor bus agent samples FLUSH# to determine its power-on configuration. See Table 44.

A.29 FRCERR (I/O)

The FRCERR signal is the Error group Functional-redundancy-check Error signal. If two Pentium Pro processors are configured in an FRC pair, as a single "logical" processor, then the checker processor asserts FRCERR if it detects a mismatch between its internally sampled outputs and the master processor's outputs. The checker's FRCERR output pin is connected to the master's FRCERR input pin.

For point-to-point connections, the checker always compares against the master's outputs. For bussed single-driver signals, the checker compares against the signal when the master is the only allowed driver. For bussed multiple-driver Wire-OR signals, the checker compares against the signal only if the master is expected to drive the signal low.

FRCERR is also toggled during the Pentium Pro processor's reset action. A Pentium Pro processor asserts FRCERR for approximately 1 second after RESET's active-to-inactive transition if it executes its built-in self-test (BIST). When BIST execution

completes, the Pentium Pro processor de-asserts FRCERR if BIST completed successfully and continues to assert FRCERR if BIST fails. If the Pentium Pro processor does not execute the BIST action, then it keeps FRCERR asserted for approximately 20 clocks and then de-asserts it.

The Pentium® Pro Processor Developer's Manual, Volume 1: Specifications (Order Number 242690) describes how a Pentium Pro processor can be configured as a master or a checker.

A.30 HIT# (I/O), HITM# (I/O)

The HIT# and HITM# signals are Snoop-hit and Hit-modified signals. They are snoop results asserted by any Pentium Pro processor bus agent in the Snoop Phase.

Any bus agent can assert both HIT# and HITM# together for one clock in the Snoop Phase to indicate that it requires a snoop stall. When a stall condition is sampled, all bus agents extend the Snoop Phase by two clocks. The stall can be continued by reasserting HIT# and HITM# together every other clock for one clock.

A caching agent must assert HITM# for one clock in the Snoop Phase if the transaction hits a Modified line, and the snooping agent must perform an implicit writeback to update main memory. The snooping agent with the Modified line makes a transition to Shared state if the original transaction is Read Line or Read Partial, otherwise it transitions to Invalid state. A Deferred Reply transaction may have HITM# asserted to indicate the return of unexpected data.

A snooping agent must assert HIT# for one clock during the Snoop Phase if the line does not hit a Modified line in its writeback cache and if at the end of the transaction it plans to keep the line in Shared state. Multiple caching agents can assert HIT# in the same Snoop Phase. If the requesting agent observes HIT# active during the Snoop Phase it can not cache the line in Exclusive or Modified state.

On observing a snoop stall, the agents asserting HIT# and HITM# independently reassert the signal after one inactive clock so that the correct snoop result is available, in case the Snoop Phase terminates after the two clock extension.

A.31 IERR# (O)

The IERR# signal is the Error group Internal Error signal. A Pentium Pro processor asserts IERR# when it observes an internal error. It keeps IERR# asserted until it is turned off as part of the Machine Check Error or the NMI handler in software, or with RESET#, BINIT#, and INIT# assertion.

An internal error can be handled in several ways inside the processor based on its power-on configuration. If Machine Check Exception (MCE) is enabled, IERR# causes an MCE entry. IERR# can also be directed on the BERR# pin to indicate an error. Usually BERR# is sampled back by all processors to enter MCE or it can be redirected as an NMI by the central agent.

A.32 IGNNE# (I)

The IGNNE# signal is the Intel Architecture Compatability group Ignore Numeric Error signal. If IGNNE# is asserted, the Pentium Pro processor ignores a numeric error and continues to execute non-control floating-point instructions. If IGNNE# is deasserted, the Pentium Pro processor freezes on a non-control floating-point instruction if a previous instruction caused an error.

IGNNE# has no effect when the NE bit in control register 0 is set.

IGNNE# is an asynchronous input. However, to guarantee recognition of this signal following an I/O write instruction, IGNNE# must be valid along with RS[2:0]# in the Response Phase of the corresponding I/O Write bus transaction. In FRC mode, IGNNE# must be synchronous to BCLK.

During active RESET#, the Pentium Pro processor begins sampling the A20M#, IGNNE# and LINT[1:0] values to determine the ratio of core-clock frequency to bus-clock frequency. See Table 44. After the PLL-lock time, the core clock becomes stable and is locked to the external bus clock. On the active-to-inactive transition of RESET#, the Pentium Pro processor latches A20M# and IGNNE# and freezes the frequency ratio internally. Normal operation on the two signals continues two clocks after RESET# inactive is sampled.

A.33 INIT# (I)

The INIT# signal is the Execution Control group initialization signal. Active INIT# input resets integer registers inside all Pentium Pro processors without affecting their internal (L1 or L2) caches or their floating-point registers. Each Pentium Pro processor begins execution at the power-on reset vector configured during power-on configuration regardless of whether INIT# has gone inactive. The processor continues to handle snoop requests during INIT# assertion.

INIT# can be used to help performance of DOS extenders written for the Intel 80286 processor. INIT# provides a method to switch from protected mode to real mode while maintaining the contents of the internal caches and floating-point state. INIT# can not be used in lieu of RESET# after power-up.

On active-to-inactive transition of RESET#, each Pentium Pro processor bus agent samples INIT# signals to determine its power-on configuration. Two clocks after RESET# is sampled deasserted, these signals begin normal operation.

INIT# is an asynchronous input. In FRC mode, INIT# must be synchronous to BCLK.

A.34 INTR (I)

The INTR signal is the Interrupt Request signal. The INTR input indicates that an external interrupt has been generated. The interrupt is maskable using the IF bit in the EFLAGS register. If the IF bit is set, the Pentium Pro processor vectors to the interrupt handler after the current instruction execution is completed. Upon recognizing the interrupt request, the Pentium Pro processor issues a single Interrupt Acknowledge (INTA) bus transaction. INTR must remain active until the INTA bus transaction to guarantee its recognition.

INTR is sampled on every rising BCLK edge. INTR is an asynchronous input but recognition of INTR is guaranteed in a specific clock if it is asserted synchronously and meets the setup and hold times. INTR must also be deasserted for a minimum of two clocks to guarantee its inactive recognition. In FRC mode, INTR must be synchronous to BCLK. On power-up the LINT[1:0] signals are used for power-on-configuration of clock ratios. Both these signals must be software configured by programming the APIC register space to be used either as NMI/INTR or LINT[1:0] in the BIOS. Because APIC is enabled after reset, LINT[1:0] is the default configuration.

A.35 LEN[1:0]# (I/O)

The LEN[1:0]# signals are data-length signals. They are transmitted using REQb[1:0]# signals by the request initiator in the second clock of Request Phase. LEN[1:0]# define the length of the data transfer requested by the request initiator as defined in Table 52. The LEN[1:0]#, HITM#, and RS[2:0]# signals together define the length of the actual data transfer.

Table 52. LEN[1:0]# Data Transfer Lengths

LEN[1:0]#	Request Initiator's Data Transfer Length
00	0-8 Bytes
01	16 Bytes
10	32 Bytes
11	Reserved

A.36 LINT[1:0] (I)

The LINT[1:0] signals are the Execution Control group Local Interrupt signals. When APIC is disabled, the LINT0 signal becomes INTR, a maskable interrupt request signal, and LINT1 becomes NMI, a non-maskable interrupt. INTR and NMI are backward compatible with the same signals for the Pentium processor. Both signals are asynchronous inputs. In FRC mode, LINT[1:0] must be synchronous to BCLK.

During active RESET#, the Pentium Pro processor continuously samples the A20M#, IGNNE# and LINT[1:0] values to determine the ratio of core-clock frequency to bus-clock frequency. See Table 44. After the PLL-lock time, the core clock becomes stable and is locked to the external bus clock. On the active-to-inactive transition of RESET#, the Pentium Pro processor latches the ratio internally.

Both these signals must be software configured by programming the APIC register space to be used either as NMI/INTR or LINT[1:0] in the BIOS. Because APIC is enabled after reset, LINT[1:0] is the default configuration.

A.37 LOCK# (I/O)

The LOCK# signal is the Arbitration group bus lock signal. For a locked sequence of transactions, LOCK# is asserted from the first transaction's Request Phase through the last transaction's Response Phase. A locked operation can be prematurely aborted (and LOCK# deasserted) if AERR# or DEFER# is asserted during the first bus transaction of the sequence. The sequence can also be prematurely aborted if a hard error (such as a hard failure response or AERR# assertion beyond the retry limit) occurs on any one of the transactions during the locked operation.

When the priority agent asserts BPRI# to arbitrate for bus ownership, it waits until it observes LOCK# deasserted. This enables symmetric agents to retain bus ownership throughout the bus locked operation and guarantee the atomicity of lock. If AERR# is asserted up to the retry limit during an ongoing locked operation, the arbitration protocol ensures that the lock owner receives the bus ownership after arbitration logic is reset. This result is accomplished by requiring the lock owner to reactivate its arbitration request one clock ahead of other agents' arbitration request. LOCK# is kept asserted throughout the arbitration reset sequence.

A.38 NMI (I)

The NMI signal is the Non-maskable Interrupt signal. It is the state of the LINT1 signal when APIC is disabled. Asserting NMI causes an interrupt with an internally supplied vector value of 2. An external interrupt-acknowledge transaction is not generated. If NMI is asserted during the execution of an NMI service routine, it remains pending and is recognized after the IRET is executed by the NMI service routine. At most, one assertion of NMI is held pending.

NMI is rising-edge sensitive. Recognition of NMI is guaranteed in a specific clock if it is asserted synchronously and meets the setup and hold times. If asserted asynchronously, active and inactive pulse widths must be a minimum of two clocks. In FRC mode, NMI must be synchronous to BCLK.

A.39 PICCLK (I)

The PICCLK signal is the Execution Control group APIC Clock signal. It is an input clock to the Pentium Pro processor for synchronous operation of the APIC bus. PICCLK must be synchronous to BCLK in FRC mode.

A.40 PICD[1:0] (I/O)

The PICD[1:0] signals are the Execution Control group APIC Data signals. They are used for bi-directional serial message passing on the APIC bus.

A.41 PWRGOOD (I)

PWRGOOD is driven to the Pentium Pro processor by the system to indicate that the clocks and power supplies are within their specification. See Section 3.9 for additional details. This signal will not affect FRC operation.

A.42 REQ[4:0]# (I/O)

The REQ[4:0]# signals are the Request Command signals. They are asserted by the current bus owner in both clocks of the Request Phase. In the first clock, the REQa[4:0]# signals define the transaction type to a level of detail that is sufficient to begin a snoop request. In the second clock, REQb[4:0]# signals carry additional information to define the complete transaction type. REQb[4:2]# is reserved. REQb[1:0]# signals transmit LEN[1:0]# (the data transfer length information). In both clocks, REQ[4:0]# and ADS# are protected by parity RP#.

All receiving agents observe the REQ[4:0]# signals to determine the transaction type and participate in the transaction as necessary, as shown in Table 53.

Table 53. Transaction Types Defined by REQa#/REQb# Signals

	REQa[4:0]#					REQb[4:0]#				
Transaction	**4**	**3**	**2**	**1**	**0**	**4**	**3**	**2**	**1**	**0**
Deferred Reply	0	0	0	0	0	X	X	X	X	X
Rsvd (Ignore)	0	0	0	0	1	X	X	X	X	X
Interrupt Acknowledge	0	1	0	0	0	DSZ#		X	0	0
Special Transactions	0	1	0	0	0	DSZ#		X	0	1
Rsvd (Central agent response)	0	1	0	0	0	DSZ#		X	1	X
Branch Trace Message	0	1	0	0	1	DSZ#		X	0	0
Rsvd (Central agent response)	0	1	0	0	1	DSZ#		X	0	1
Rsvd (Central agent response)	0	1	0	0	1	DSZ#		X	1	X
I/O Read	1	0	0	0	0	DSZ#		X	LEN#	
I/O Write	1	0	0	0	1	DSZ#		X	LEN#	
Rsvd (Ignore)	1	1	0	0	X	DSZ#		X	X	X
Memory Read & Invalidate	ASZ#		0	1	0	DSZ#		X	LEN#	
Rsvd (Memory Write)	ASZ#		0	1	1	DSZ#		X	LEN#	
Memory Code Read	ASZ#		1	D/C#=0	0	DSZ#		X	LEN#	
Memory Data Read	ASZ#		1	D/C#=1	0	DSZ#		X	LEN#	
Memory Write (may not be retried)	ASZ#		1	W/WB#=0	1	DSZ#		X	LEN#	
Memory Write (may not be retried)	ASZ#		1	W/WB#=1	1	DSZ#		X	LEN#	

A.43 RESET# (I)

The RESET# signal is the Execution Control group reset signal. Asserting RESET# resets all Pentium Pro processors to known states and invalidates their L1 and L2 caches without writing back Modified (M state) lines. For a power-on type reset, RESET# must stay active for at least one millisecond after $V_{CC}P$ and CLK have reached their proper DC and AC specifications. On observing active RESET#, all bus agents must deassert their outputs within two clocks.

A number of bus signals are sampled at the active-to-inactive transition of RESET# for the power-on configuration. The configuration options are described in the *Pentium® Pro Processor Developer's Manual, Volume 1: Specifications (Order Number 242690)* and in the pertinent signal descriptions in this appendix.

Unless its outputs are tristated during power-on configuration, after active-to-inactive transition of RESET#, the Pentium Pro processor optionally executes its built-in self-test (BIST) and begins program execution at reset-vector 0_000F_FFF0H or 0_FFFF_FFF0H.

A.44 RP# (I/O)

The RP# signal is the Request Parity signal. It is driven by the request initiator in both clocks of the Request Phase. RP# provides parity protection on ADS# and REQ[4:0]#. When a Pentium Pro processor bus agent observes an RP# parity error on any one of the two Request Phase clocks, it must assert AERR# in the Error Phase, provided "AERR# drive" is enabled during the power-on configuration.

A correct parity signal is high if an even number of covered signals are low and low if an odd number of covered signals are low. This definition allows parity to be high when all covered signals are high.

A.45 RS[2:0]# (I)

The RS[2:0]# signals are the Response Status signals. They are driven by the response agent (the agent responsible for completion of the transaction at the top of the In-order Queue). Assertion of RS[2:0]# to a non-zero value for one clock completes the Response Phase for a transaction. The response encodings are shown in Table 55. Only certain response combinations are valid, based on the snoop result signaled during the transaction's Snoop Phase.

The RS[2:0]# assertion for a transaction is initiated when all of the following conditions are met:

- All bus agents have observed the Snoop Phase completion of the transaction.
- The transaction is at the top of the In-order Queue.
- RS[2:0]# are sampled in the Idle state

The response driven depends on the transaction as described below:

- The response agent returns a hard-failure response for any transaction in which the response agent observes a hard error.
- The response agent returns a Normal with data response for a read transaction with HITM# and DEFER# deasserted in the Snoop Phase, when the addressed agent is ready to return data and samples inactive DBSY#.
- The response agent returns a Normal without data response for a write transaction with HITM# and DEFER# deasserted in the Snoop Phase, when the addressed agent samples TRDY# active and DBSY# inactive, and it is ready to complete the transaction.

Table 54. Transaction Response Encodings

RS[2:0]	Description	HITM#	DEFER#
000	Idle State.	N/A	N/A
001	Retry Response. The transaction is canceled and must be retried by the initiator.	0	1
010	Defer Response. The transaction is suspended. The defer agent will complete it with a defer reply	0	1
011	Reserved.	0	1
100	Hard Failure. The transaction received a hard error. Exception handling is required.	X	X
101	Normal without data	0	0
110	Implicit WriteBack Response. Snooping agent will transfer the modified cache line on the data bus.	1	X
111	Normal with data.	0	0

- The response agent must return an Implicit writeback response in the next clock for a read transaction with HITM# asserted in the Snoop Phase, when the addressed agent samples TRDY# active and DBSY# inactive.
- The addressed agent must return an Implicit writeback response in the clock after the following sequence is sampled for a write transaction with HITM# asserted:
  1. TRDY# active and DBSY# inactive
  2. Followed by TRDY# inactive
  3. Followed by TRDY# active and DBSY# inactive
- The defer agent can return a Deferred, Retry, or Split response anytime for a read transaction with HITM# deasserted and DEFER# asserted.
- The defer agent can return Deferred, Retry, or Split response when it samples TRDY# active and DBSY# inactive for a write transaction with HITM# deasserted and DEFER# asserted.

A.46 RSP# (I)

The RSP# signal is the Response Parity signal. It is driven by the response agent during assertion of RS[2:0]#. RSP# provides parity protection for RS[2:0]#.

A correct parity signal is high if an even number of covered signals are low and low if an odd number of covered signals are low. During Idle state of RS[2:0]# (RS[2:0]#=000), RSP# is also high since it is not driven by any agent guaranteeing correct parity.

Pentium Pro processor bus agents can check RSP# at all times and if a parity error is observed, treat it as a protocol violation error. If the BINIT# driver is enabled during configuration, the agent observing RSP# parity error can assert BINIT#.

A.47 SMI# (I)

System Management Interrupt is asserted asynchronously by system logic. On accepting a System Management Interrupt, the Pentium Pro processor saves the current state and enters SMM mode. It issues an SMI Acknowledge Bus transaction and then begins program execution from the SMM handler.

A.48 SMMEM# (I/O)

The SMMEM# signal is the System Management Mode Memory signal. It is driven on the second clock

of the Request Phase on the EXF4#/Ab7# signal. It is asserted by the Pentium Pro processor to indicate that the processor is in System Management Mode and is executing out of SMRAM space.

A.49 SPLCK# (I/O)

The SPLCK# signal is the Split Lock signal. It is driven in the second clock of the Request Phase on the EXF3#/Ab6# signal of the first transaction of a locked operation. It is driven to indicate that the locked operation will consist of four locked transactions. Note that SPLCK# is asserted only for locked operations and only in the first transaction of the locked operation.

A.50 STPCLK# (I)

The STPCLK# signal is the Stop Clock signal. When asserted, the Pentium Pro processor enters a low-power state, the stop-clock state. The processor issues a Stop Clock Acknowledge special transaction, and stops providing internal clock signals to all units except the bus unit and the APIC unit. The processor continues to snoop bus transactions and service interrupts while in stop clock state. When STPCLK# is deasserted, the processor restarts its internal clock to all units and resumes execution. The assertion of STPCLK# has no effect on the bus clock.

STPCLK# is an asynchronous input. In FRC mode, STPCLK# must be synchronous to BCLK.

A.51 TCK (I)

The TCK signal is the System Support group Test Clock signal. TCK provides the clock input for the test bus (also known as the test access port). Make certain that TCK is active before initializing the TAP.

A.52 TDI(I)

The TDI signal is the System Support group test-data-in signal. TDI transfers serial test data into the Pentium Pro processor. TDI provides the serial input needed for JTAG support.

A.53 TDO (O)

The TDO signal is the System Support group test-data-out signal. TDO transfers serial test data out from the Pentium Pro processor. TDO provides the serial output needed for JTAG support.

A.54 TMS (I)

The TMS signal is an additional System Support group JTAG-support signal.

A.55 TRDY (I)

The TRDY# signal is the target Ready signal. It is asserted by the target in the Response Phase to indicate that the target is ready to receive write or implicit writeback data transfer. This enables the request initiator or the snooping agent to begin the appropriate data transfer. There will be no data transfer after a TRDY# assertion if a write has zero length indicated in the Request Phase. The data transfer is optional if an implicit writeback occurs for a transaction which writes a full cache line (the Pentium Pro processor will perform the implicit writeback).

TRDY# for a write transaction is driven by the addressed agent when:

- When the transaction has a write or writeback data transfer
- It has a free buffer available to receive the write data
- A minimum of 3 clocks after ADS# for the transaction
- The transaction reaches the top-of-the-In-order Queue
- A minimum of 1 clock after RS[2:0]# active assertion for transaction "n-1". (After the transaction reaches the top of the In-order Queue).

TRDY# for an implicit writeback is driven by the addressed agent when:

- The transaction has an implicit writeback data transfer indicated in the Snoop Result Phase.
- It has a free cache line buffer to receive the cache line writeback

- If the transaction also has a request initiated transfer, that the request initiated TRDY# was asserted and then deasserted (TRDY# must be deasserted for at least one clock between the TRDY# for the write and the TRDY# for the implicit writeback),
- A minimum of 1 clock after RS[2:0]# active assertion for transaction “n-1”. After the transaction reaches the top of the In-order Queue).

TRDY# for a write or an implicit writeback may be deasserted when:

- Inactive DBSY# and active TRDY# are observed.
- DBSY# is observed inactive on the clock TRDY# is asserted.
- A minimum of three clocks can be guaranteed between two active-to-inactive transitions of TRDY#
- The response is driven on RS[2:0]#.
- Inactive DBSY# and active TRDY# are observed for a write, and TRDY# is required for an implicit writeback.

A.56 TRST (I)

The TRST# signal resets the JTAG logic.

In summer 1992, two years after joining the project, I was promoted to architecture manager and served as Intel's lead IA-32 architect from 1992 through 2000.

Somewhat to my surprise, the P6 design project turned out to be a watershed event in the history of the computer industry and the Internet; it could keep up with the industry's fastest chips, especially those from reduced-instruction-set computer (RISC) manufacturers, and it had enough flexibility and headroom to serve as the basis for many future proliferation designs.

It also gave Intel a foothold in the maturing workstation market, and it immediately established them in the server space just as the Internet was driving up demand for inexpensive Web servers.

> ***The P6 has become the most successful general-purpose processor ever created.***

The P6 project would eventually grow to over 400 design and validation engineers and take 4.5 years to production. But that huge investment paid off—P6 became the Pentium Pro microprocessor, was adapted to become the Pentium II and then the Pentium III, and, most recently, has evolved into the Centrino mobile line. From the basic design have come numerous Xeon and Celeron variants.

In short, the P6 has become the most successful general-purpose processor ever created, with hundreds of millions of chips being shipped. This book is my personal account of that project, with occasional excursions into Pentium 4.

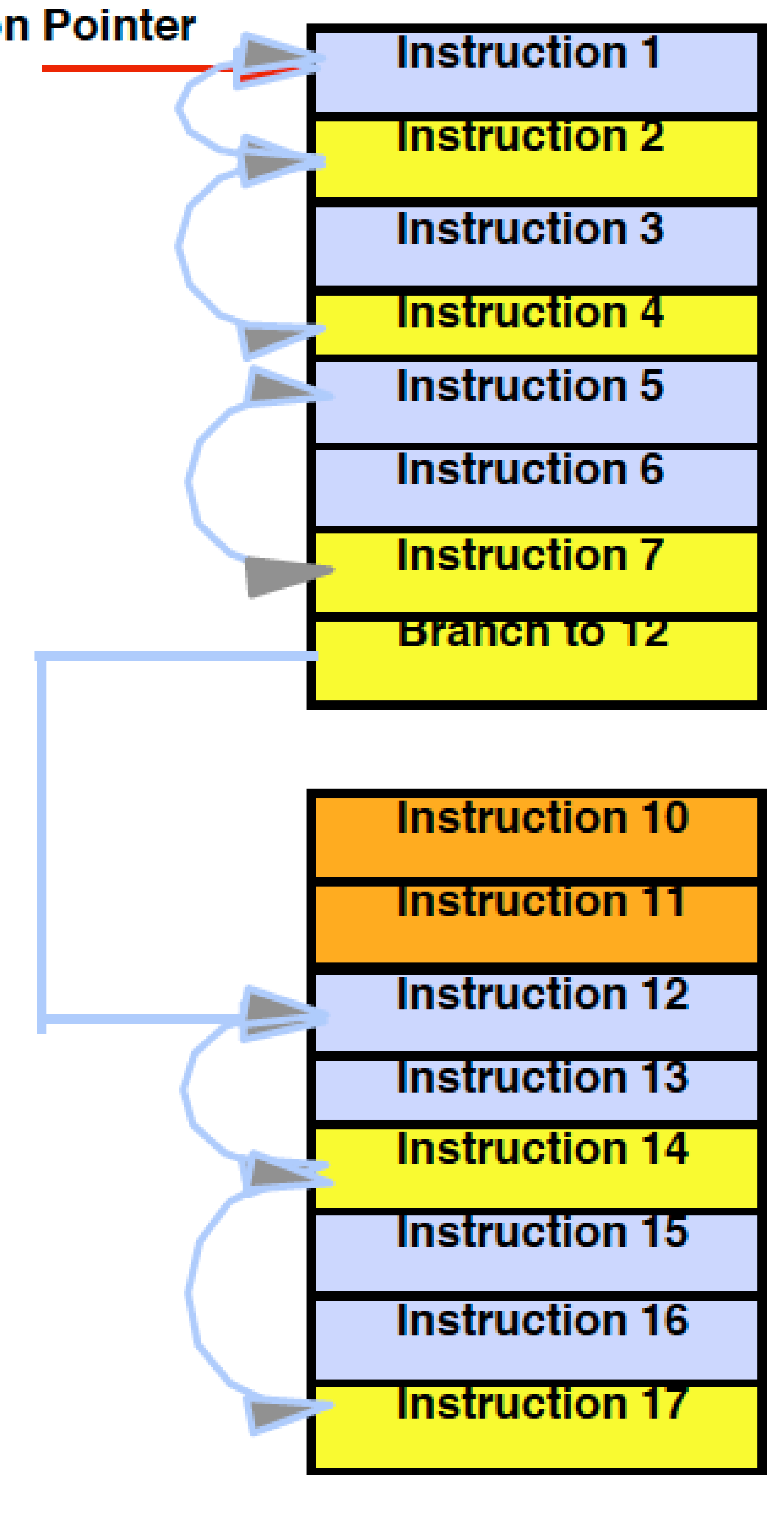
P6 Dynamic Execution Architecture

Extends the Intel Architecture Beyond Superscalar

- Non-blocking architecture
  - –Prevents processor stalls during cache, memory, I/O accesses
- Out-of-order execution
  - –Executes all available instructions as data dependencies are resolved
- Speculative execution with multiple branch prediction
  - –Eliminates CPU stalls when branch occurs
  - –Further improves effectiveness of O-O-O
- Register Renaming
  - –Eliminates false dependencies caused by limited register set size
  - –Also improves effectiveness of O-O-O

Data Flow Engine

- P6 Implements a dataflow engine
- Instructions usually have numerous dependencies (registers, flags, memory, etc.)
- Instruction dependencies are rigorously observed
- When all sources are available instruction is ready to execute
- When execution unit is available, instruction executes



OUT-OF-ORDER?

- What is Out-of Order?
  - Instructions with dependencies resolved, or no dependencies, may execute ahead of their von Neumann Schedule
  - This is a data flow sequence
- Example on right: a:, c: and d: could execute at the same time
  - This would enable us to execute the sequence in 2 cycles rather than 4
- In previous Superscalar processors, compilers re-scheduled code to create the same effect. Thus P6 has less dependence on compilers for performance.

a: r1 = Mem(x)

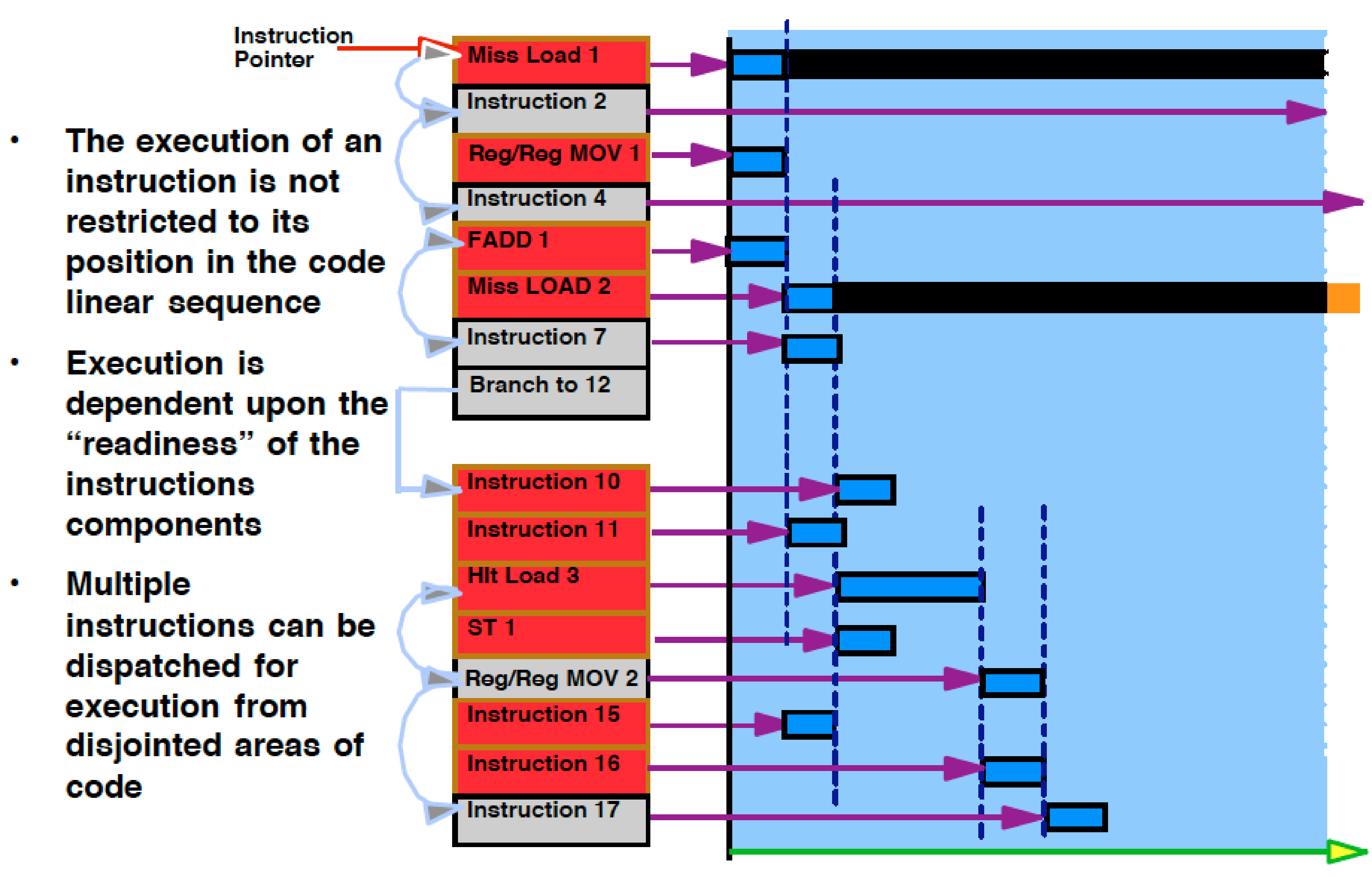
b: r3 = r1+ r2

c: r5 = 9

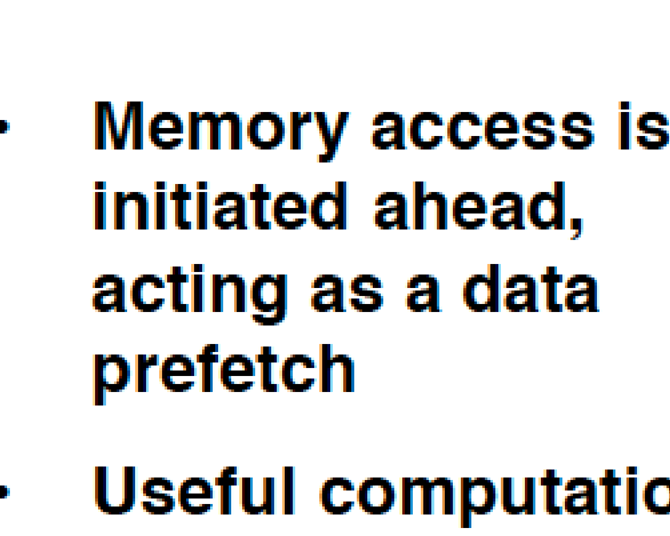
d: r4 = r4 + 1

OOO allows the processor to do useful work even though instructions may be blocked!

Speculative Execution



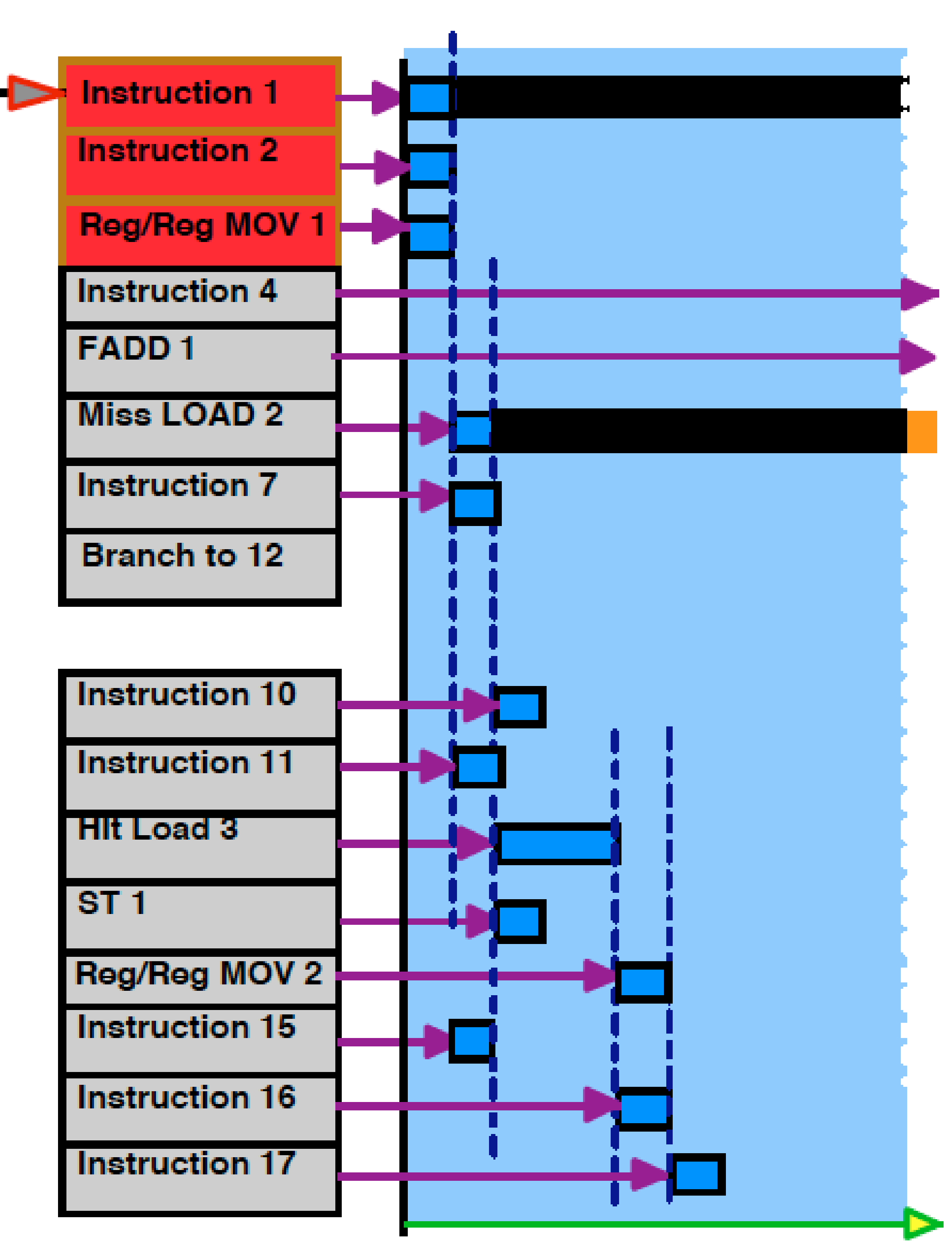
Non Blocking Architecture



- Memory access is initiated ahead, acting as a data prefetch
- Useful computation is done while waiting for data to return
- Multiple cache misses do not cumulatively add their latencies

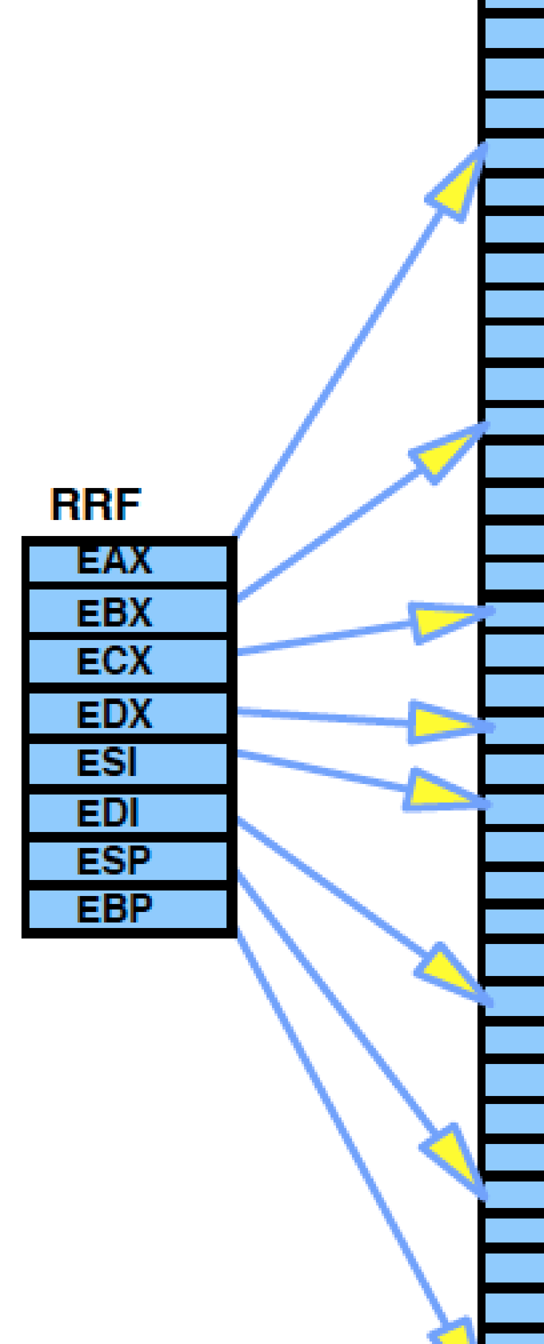
Instruction Retirement

- Instructions are fetched in-order, executed Out-Of-Order
- The retirement of instructions is in-order
- Retirement pointer identifies the block of instructions that are being committed to permanent processor state
- Multiple instructions can be retired. They always live in a sequential block of 3 instructions



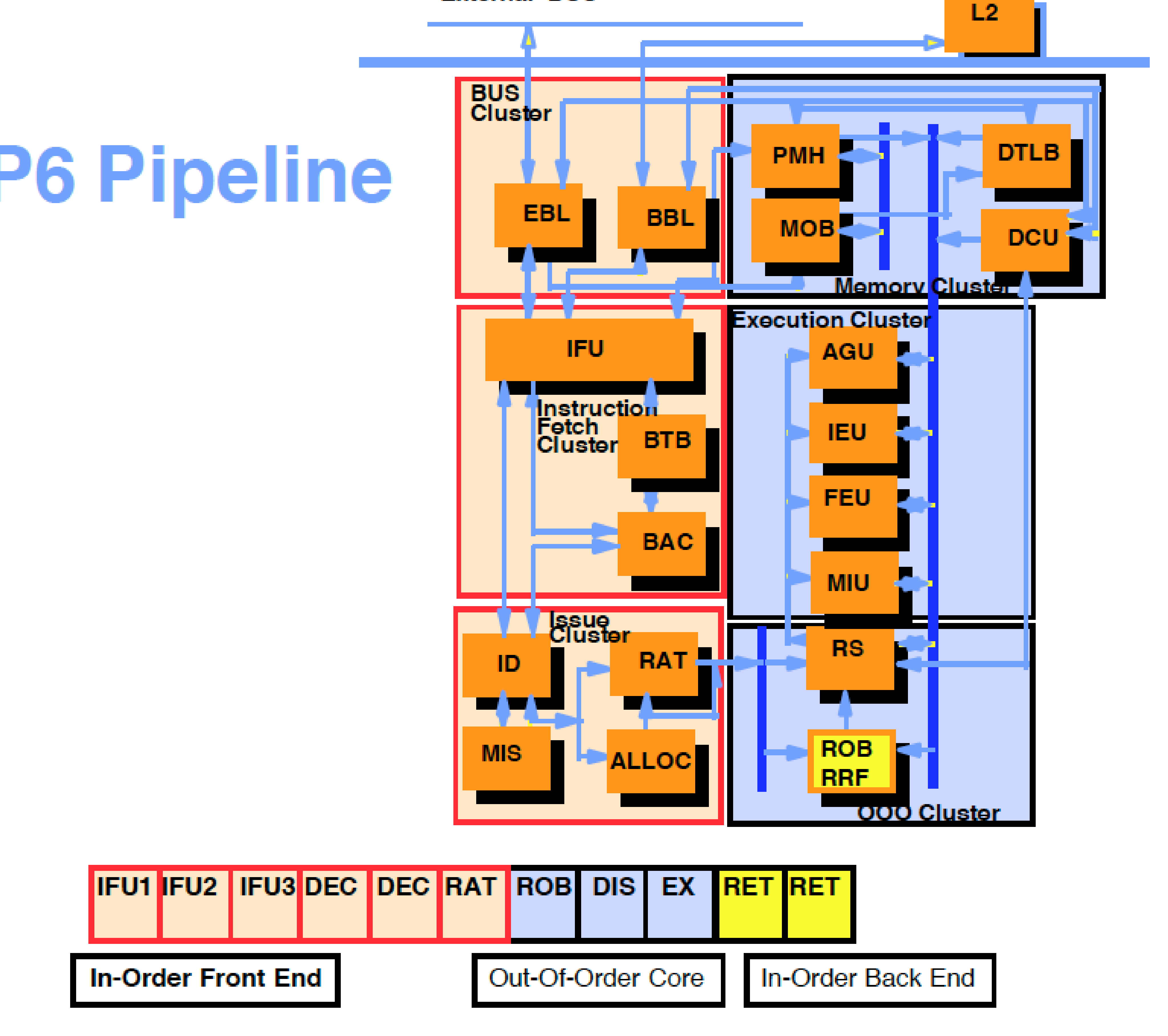
Register Renaming

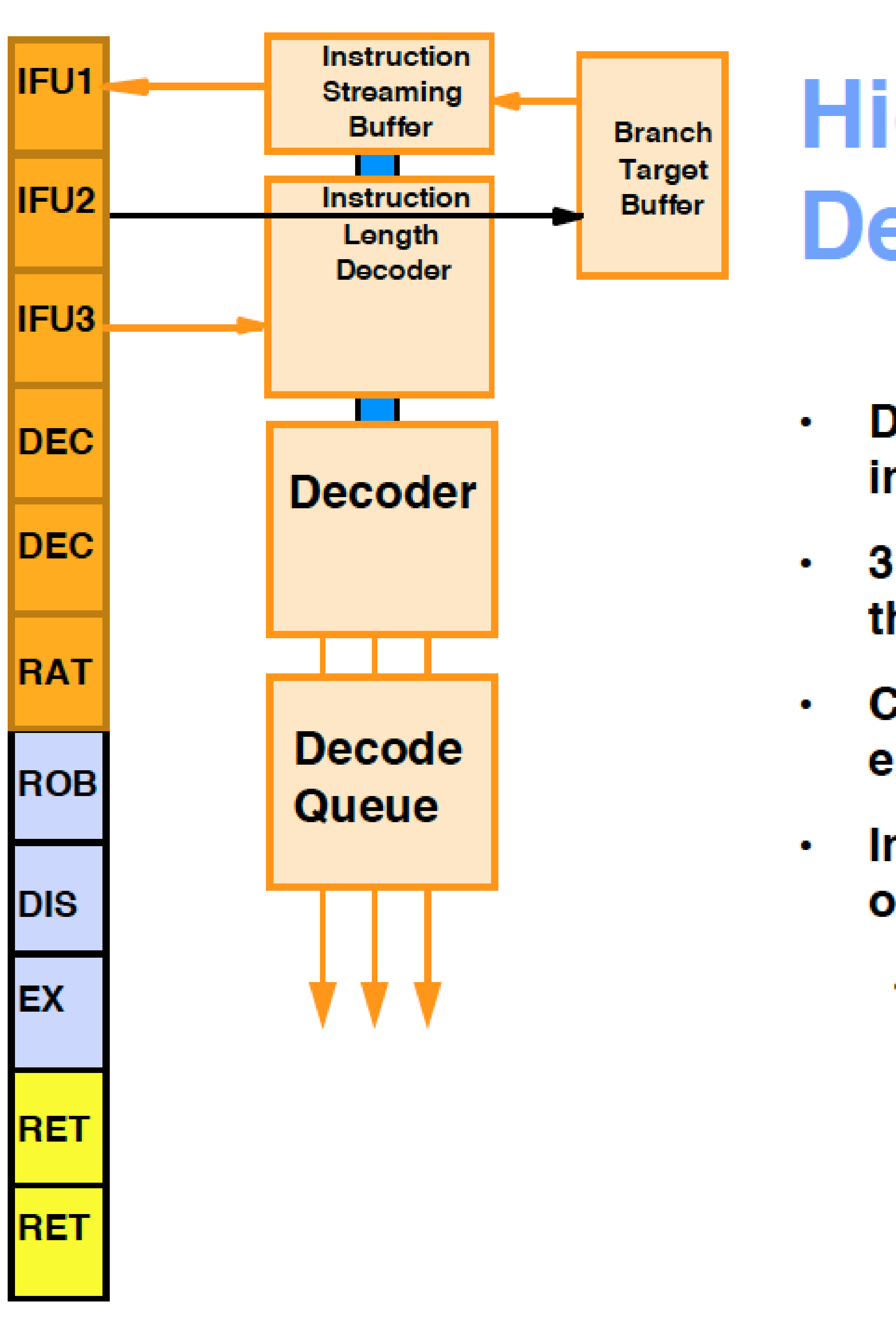
- Intel Architecture processors have a relatively small number of general purpose registers
- False dependencies can be caused by the need to reuse registers for unconnected reasons, i.e.
  - MOV EAX, 17
  - ADD Mem, EAX
  - MOV EAX, 3
  - ADD EAX, EBX



Increased number of registers allows non-blocking architecture to continue execution

The P6 Pipeline



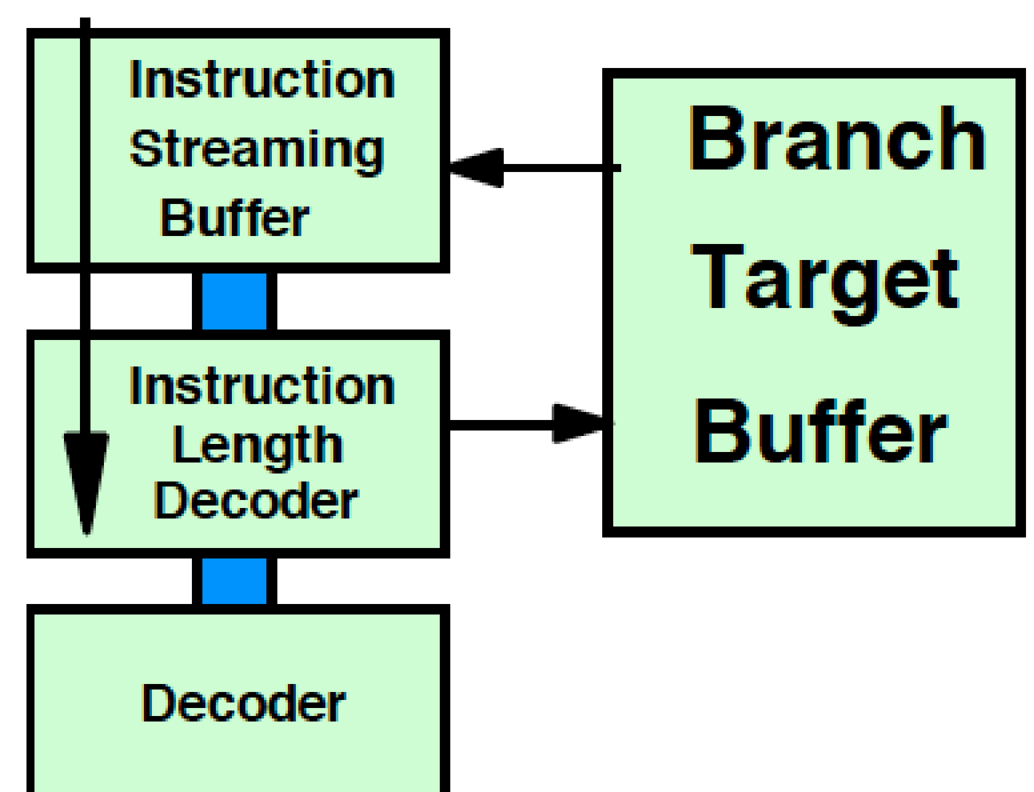


High Bandwidth Decoder

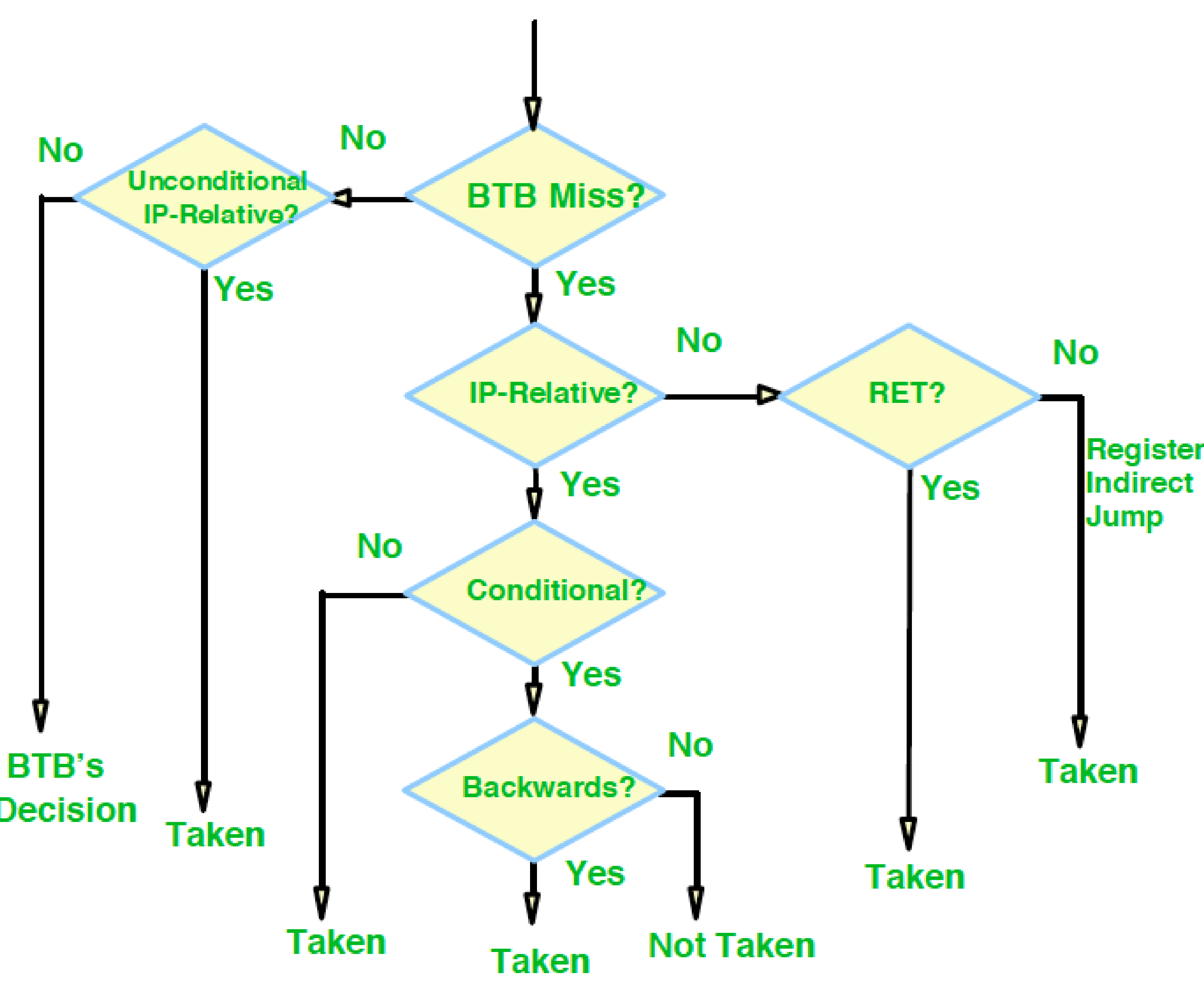
- Decoder throughput is independent of cache hit/miss
- 3 instructions/clock maximum throughput
- Code only cached in L1 when executed
- Instruction streaming buffer overruns on a branch
  - This can increase performance!

Large Branch Target Buffer

- 512 Branch to/from entries cached
  - Branch targets only cached when seen taken
- 2 level adaptive algorithm
- Static branch predictor
- Return Stack Buffer
  - Reduces misprediction for RET instructions



Static Prediction



http://csit-sun.pub.ro/~cpop/Tools_SMP/DYNEXEC.EXE

Part of the motivation for designing an out-of-order, superscalar engine is to exploit these opportunities for concurrent instruction execution. Equally important is the out-of-order engine's tolerance of cache latency. Caches are fast, physically local hardware structures that maintain copies of what is in main memory. When the processor executes a memory load to an address for which the cache happens to have a copy, that load is said to hit the cache and might take only one or two clock cycles. If the cache does not have a copy, the load misses the cache, and the load instruction incurring the miss must now wait until the hardware contacts main memory and transfers a copy of the missing data into the cache.

In P6's era, cache misses required a few dozen clock cycles to service. With the Pentium 4 generation, a cache miss could take several hundred clock cycles, and the general trend is worsening with each new process generation. As in Program 2, most code must perform a series of loads to get data, then operate on that data, and, finally, send out the data just created. Until the loads have completed, nothing else can happen; the program stalls for the duration of a cache miss.

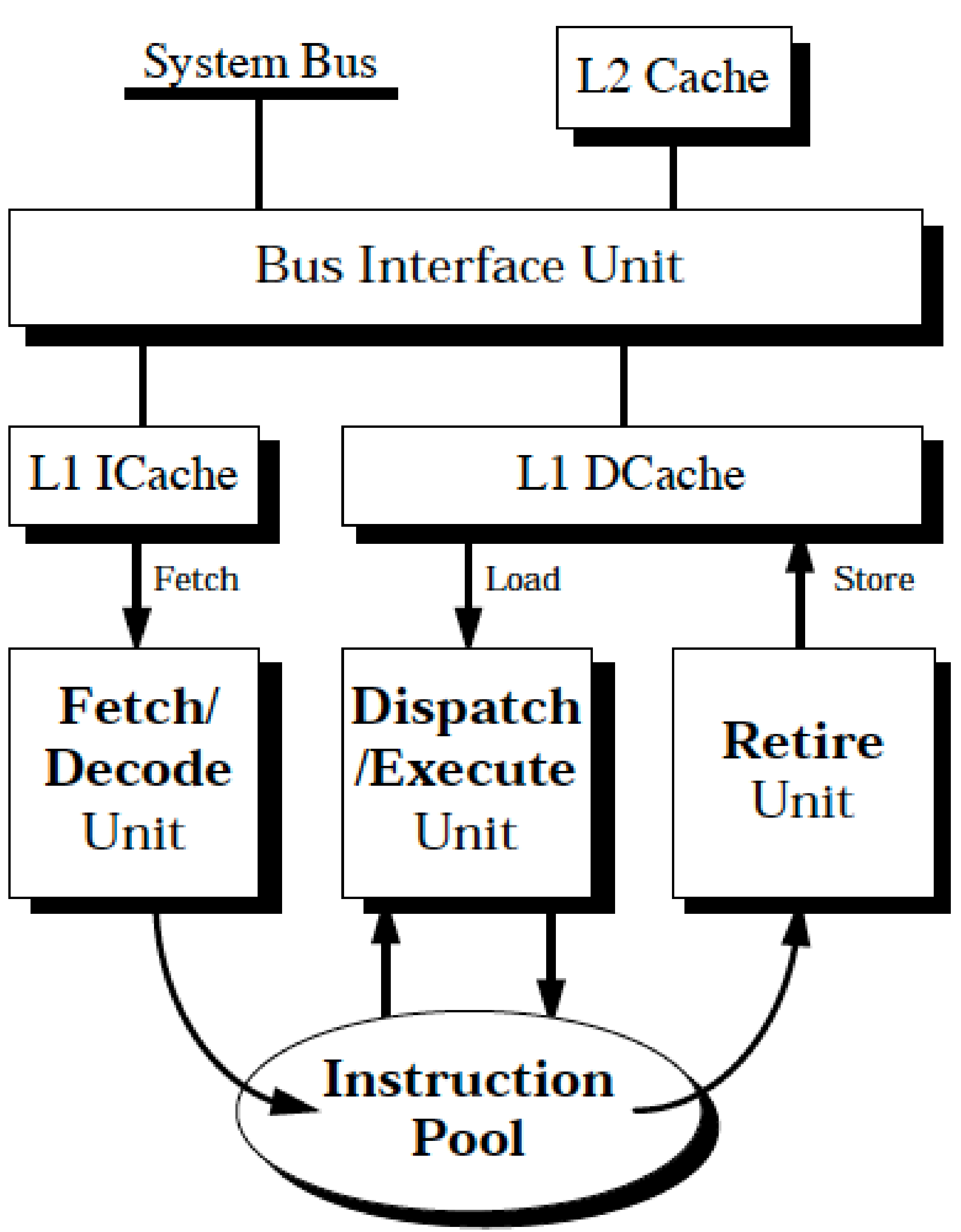
An out-of-order engine partially circumvents this limitation: If one load misses the cache, all instructions that have a true data dependency on that load must themselves wait. But any instructions that are not data-dependent on that load, including other loads, can "go around" the stalled load and try their luck.

The FETCH/DECODE unit: An in-order unit that takes as input the user program instruction stream from the instruction cache, and decodes them into a series of micro-operations (uops) that represent the dataflow of that instruction stream. The program pre-fetch is itself speculative.

The DISPATCH/EXECUTE unit: An out-of-order unit that accepts the dataflow stream, schedules execution of the uops subject to data dependencies and resource availability and temporarily stores the results of these speculative executions.

The RETIRE unit: An in-order unit that knows how and when to commit ("retire") the temporary, speculative results to permanent architectural state.

The BUS INTERFACE unit: A partially ordered unit responsible for connecting the three internal units to the real world. The bus interface unit communicates directly with the L2 cache supporting up to four concurrent cache accesses. The bus interface unit also controls a transaction bus, with MESI snooping protocol, to system memory.



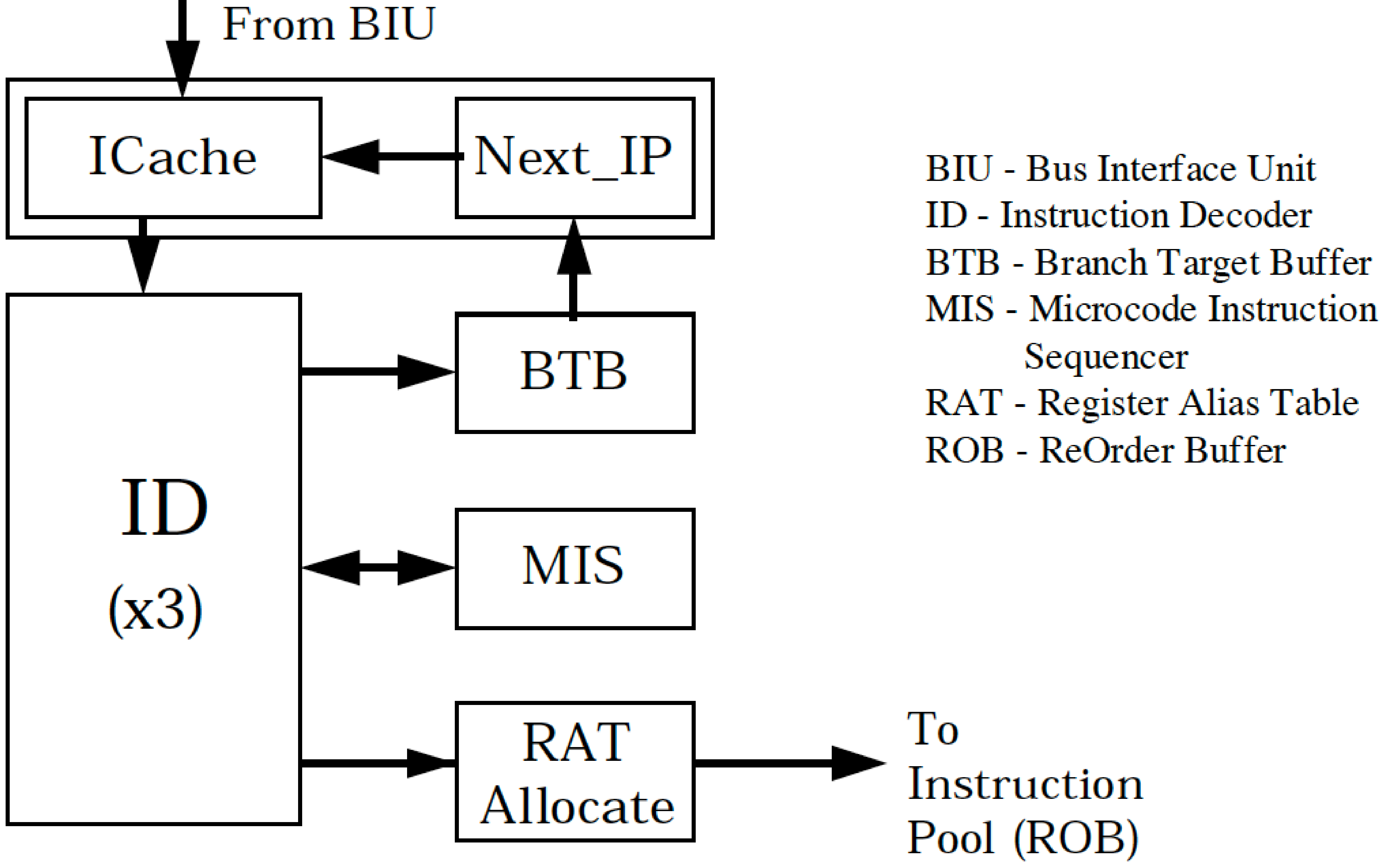


Figure 4: Looking inside the Fetch/Decode Unit

The dispatch unit selects uops from the instruction pool depending upon their status. If the status indicates that a uop has all of its operands then the dispatch unit checks to see if the execution resource needed by that uop is also available. If both are true, it removes that uop and sends it to the resource where it is executed. The results of the uop are later returned to the pool. There are five ports on the Reservation Station and the multiple resources are accessed as shown in Figure 5:

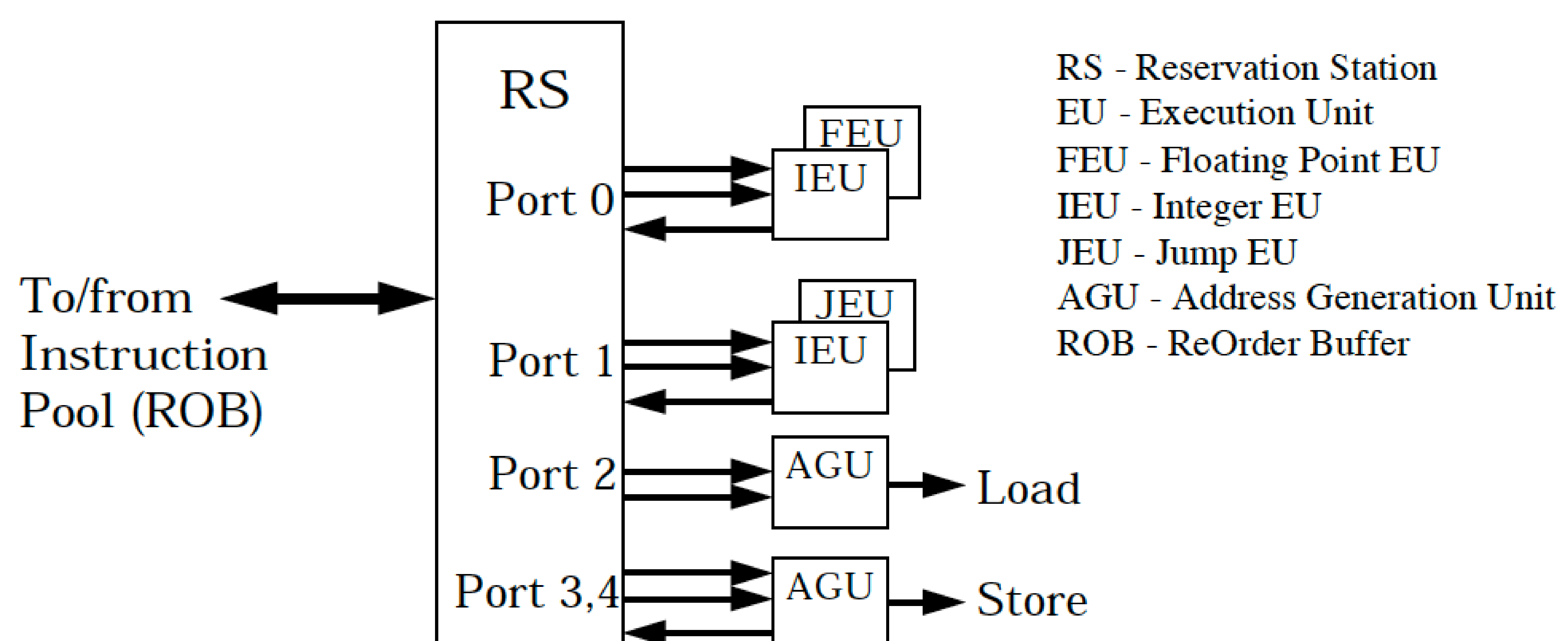


Figure 5: Looking inside the Dispatch/Execute Unit

The P6 can schedule at a peak rate of 5 uops per clock, one to each resource port, but a sustained rate of 3 uops per clock is typical. The activity of this scheduling process is the quintessential out-of-order process; uops are dispatched to the execution resources strictly according to dataflow constraints and resource availability, without regard to the original ordering of the program.

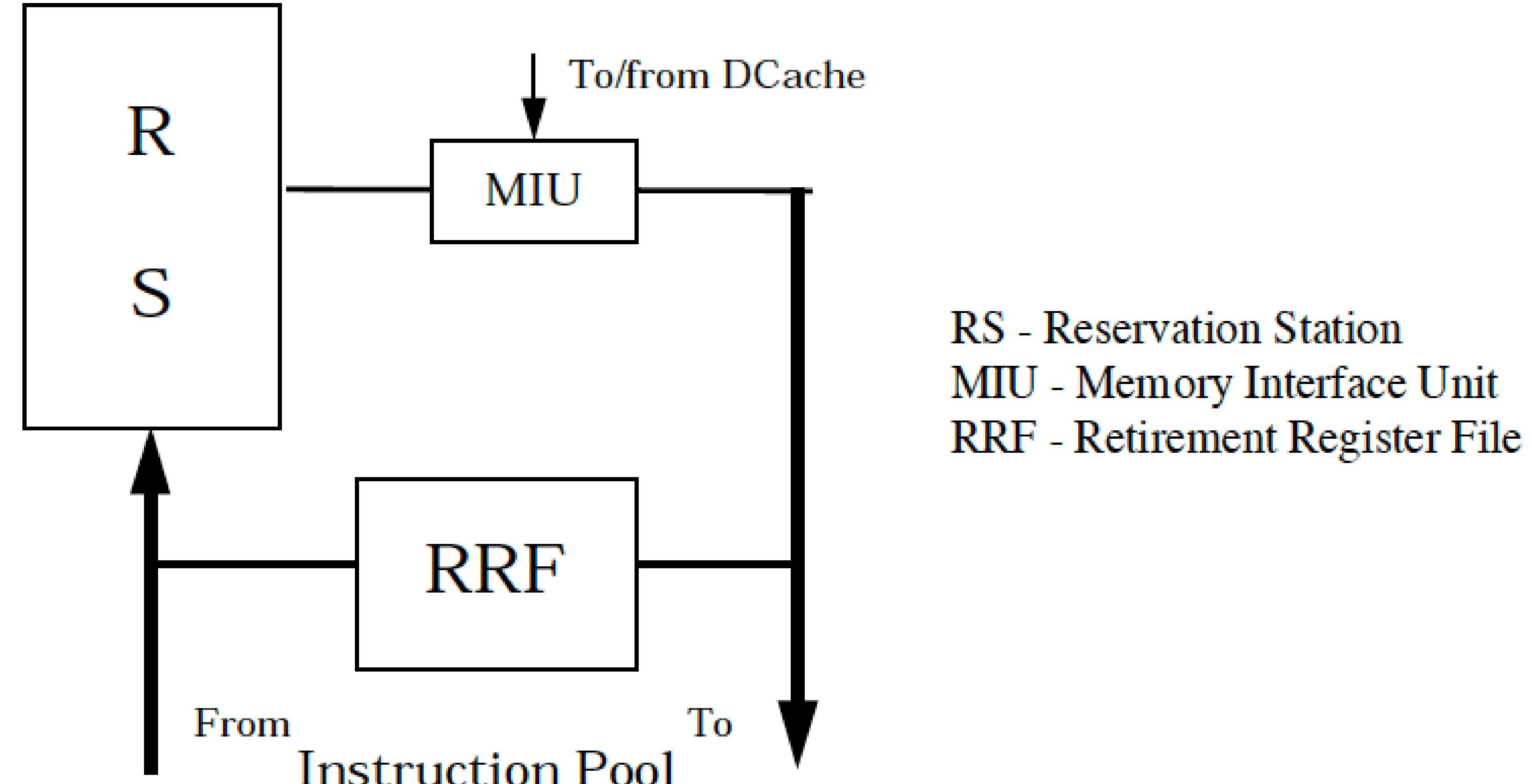


Figure 6: Looking inside the Retire Unit

The retire unit is also checking the status of uops in the instruction pool - it is looking for uops that have executed and can be removed from the pool. Once removed, the uops' original architectural target is written as per the original IA instruction. The retirement unit must not only notice which uops are complete, it must also re-impose the original program order on them. It must also do this in the face of interrupts, traps, faults, breakpoints and mis-predictions.

There are two clock cycles devoted to the retirement process. The retirement unit must first read the instruction pool to find the potential candidates for retirement and determine which of these candidates are next in the original program order. Then it writes the results of this cycle's retirements to both the Instruction Pool and the RRF. The retirement unit is capable of retiring 3 uops per clock.

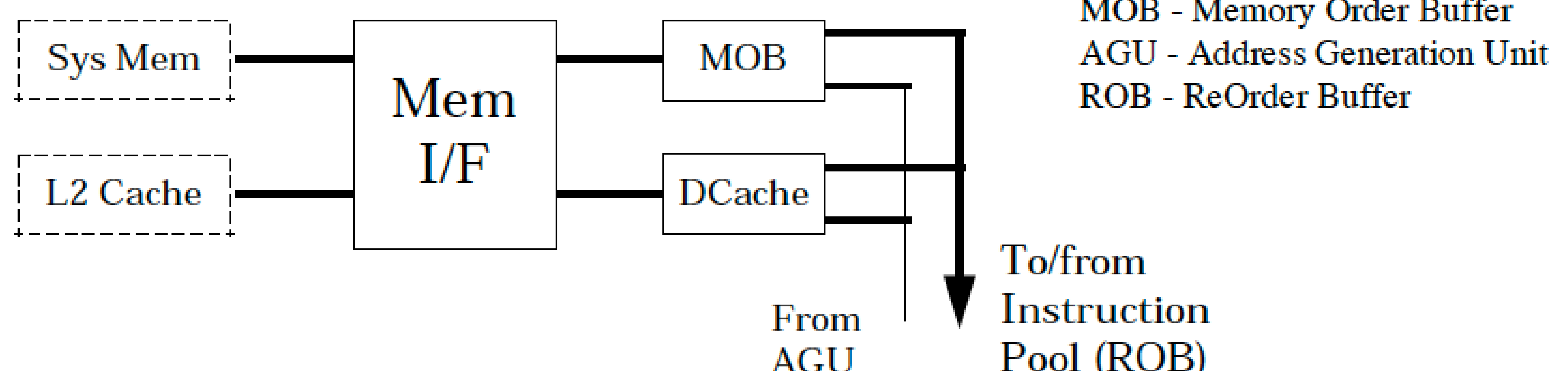


Figure 7: Looking inside the Bus Interface Unit

There are two types of memory access: loads and stores. Loads only need to specify the memory address to be accessed, the width of the data being retrieved, and the destination register. Loads are encoded into a single uop.

Stores need to provide a memory address, a data width, and the data to be written. Stores therefore require two uops, one to generate the address, one to generate the data. These uops are scheduled independently to maximize their concurrency, but must re-combine in the store buffer for the store to complete.

Stores are never performed speculatively, there being no transparent way to undo them. Stores are also never re-ordered among themselves. The Store Buffer dispatches a store only when the store has both its address and its data, and there are no older stores awaiting dispatch.

What impact will a speculative core have on the real world? Early in the P6 project, we studied the importance of memory access reordering. The basic conclusions were as follows:

- Stores must be constrained from passing other stores, for only a small impact on performance.
- Stores can be constrained from passing loads, for an inconsequential performance loss.
- Constraining loads from passing other loads or from passing stores creates a significant impact on performance.

So what we need is a memory subsystem architecture that allows loads to pass stores. And we need to make it possible for loads to pass loads. The Memory Order Buffer (MOB) accomplishes this task by acting like a reservation station and re-order buffer, in that it holds suspended loads and stores, redispatching them when the blocking condition (dependency or resource) disappears.

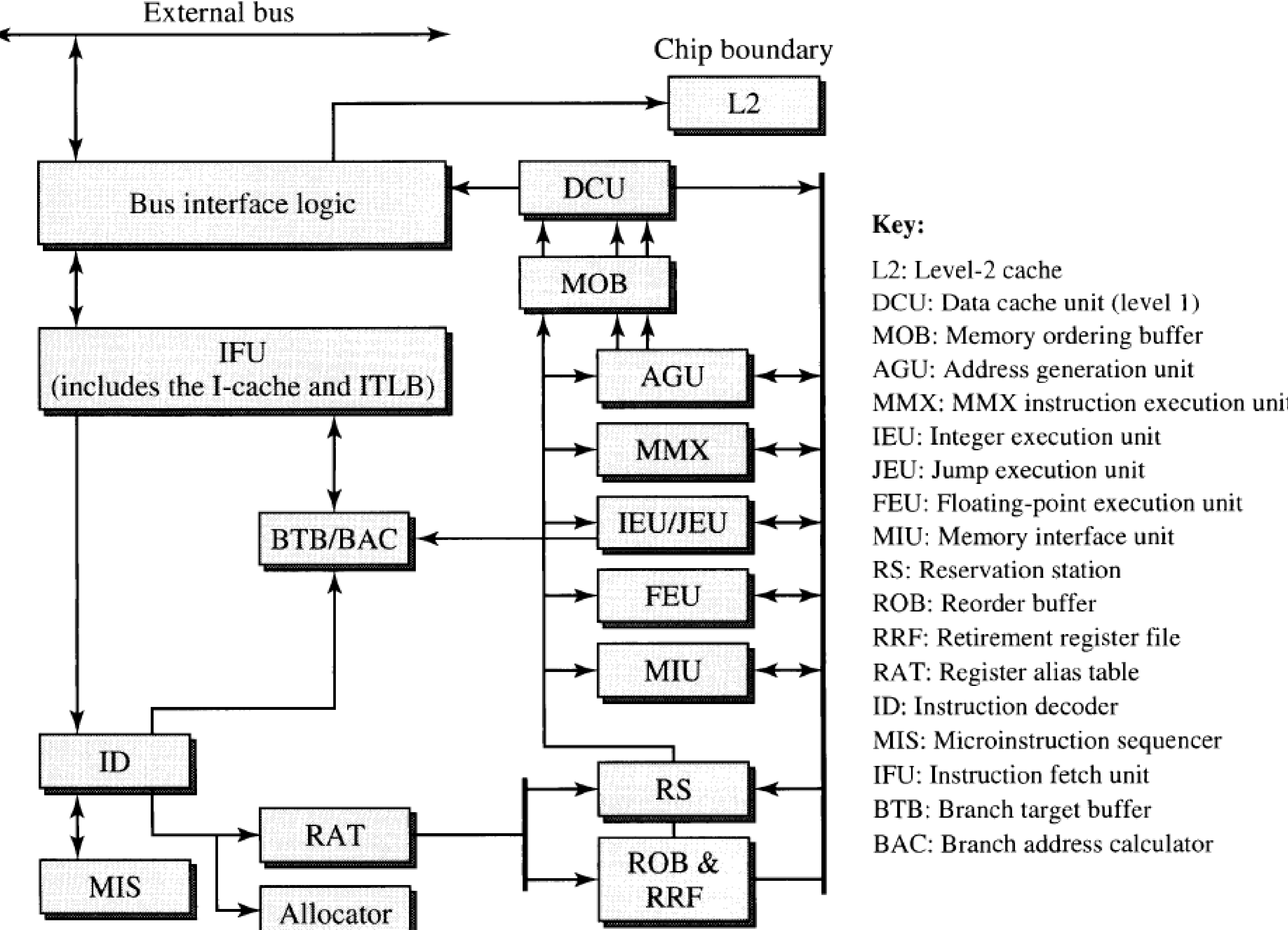


Figure 7.1

P6 Microarchitecture Block Diagram.

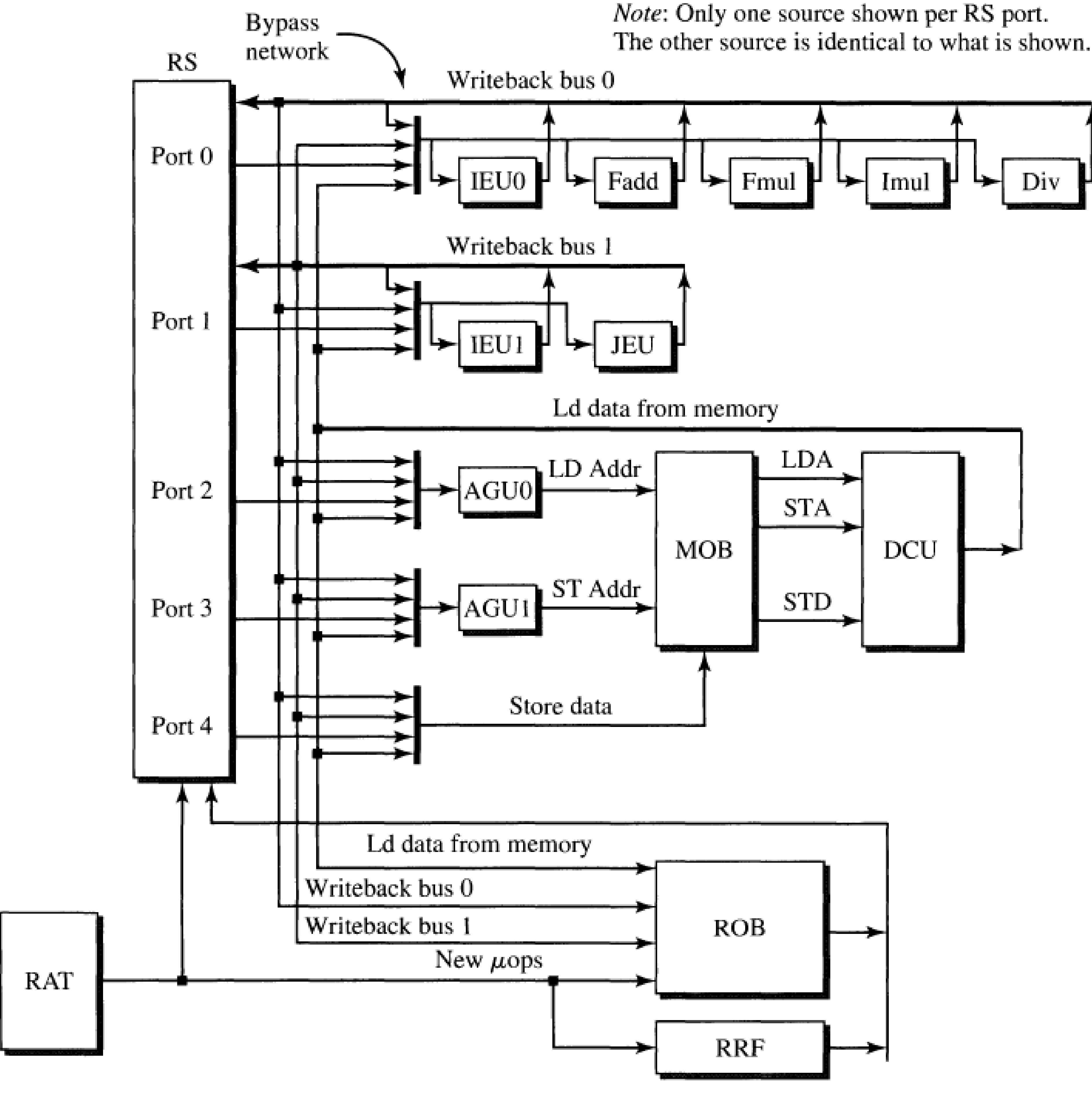


Figure 7.14
Execution Unit Data Paths.

Table 7.1
Registers in the RRF

Qty.	Register Name(s)	Size (bits)	Description
8	i486 general registers	32	EAX, ECX, EDX, EBX, EBP, ESP, ESI, EDI
8	i486 FP stack registers	86	FST(0-7)
12	General microcode temp. registers	86	For storing both integer and FP values
4	Integer microcode temp. registers	32	For storing integer values
1	EFLAGS	32	The i486 system flags register
1	ArithFlags	8	The i486 flags which are renamed
2	FCC	4	The FP condition codes
1	EIP	32	The architectural instruction pointer
1	FIP	32	The architectural FP instruction pointer
1	EventUIP	12	The micro-instruction reporting an event
2	FSW	16	The FP status word

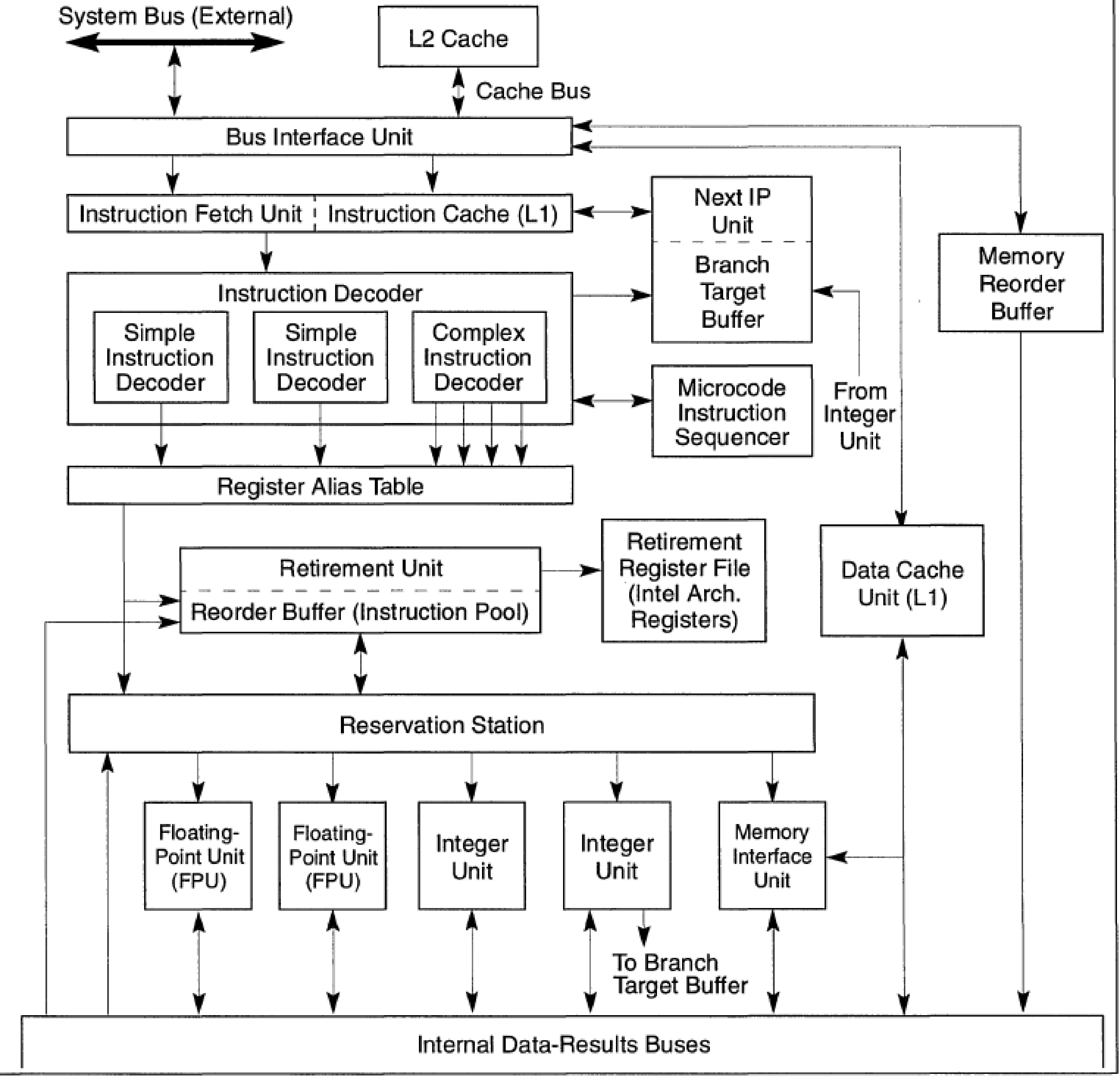


Figure 2-2. Functional Block Diagram of the Pentium® Pro Processor Microarchitecture

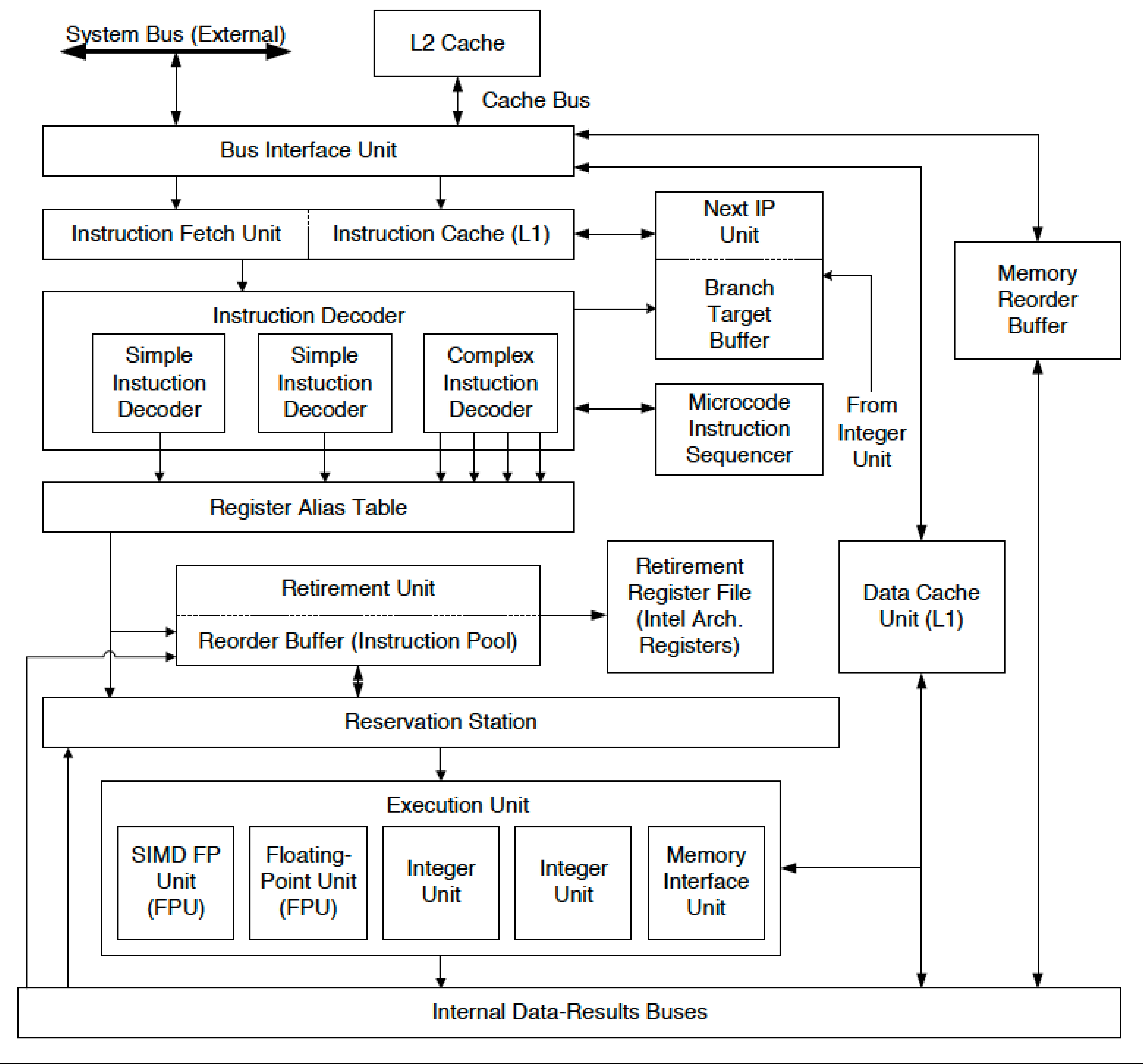
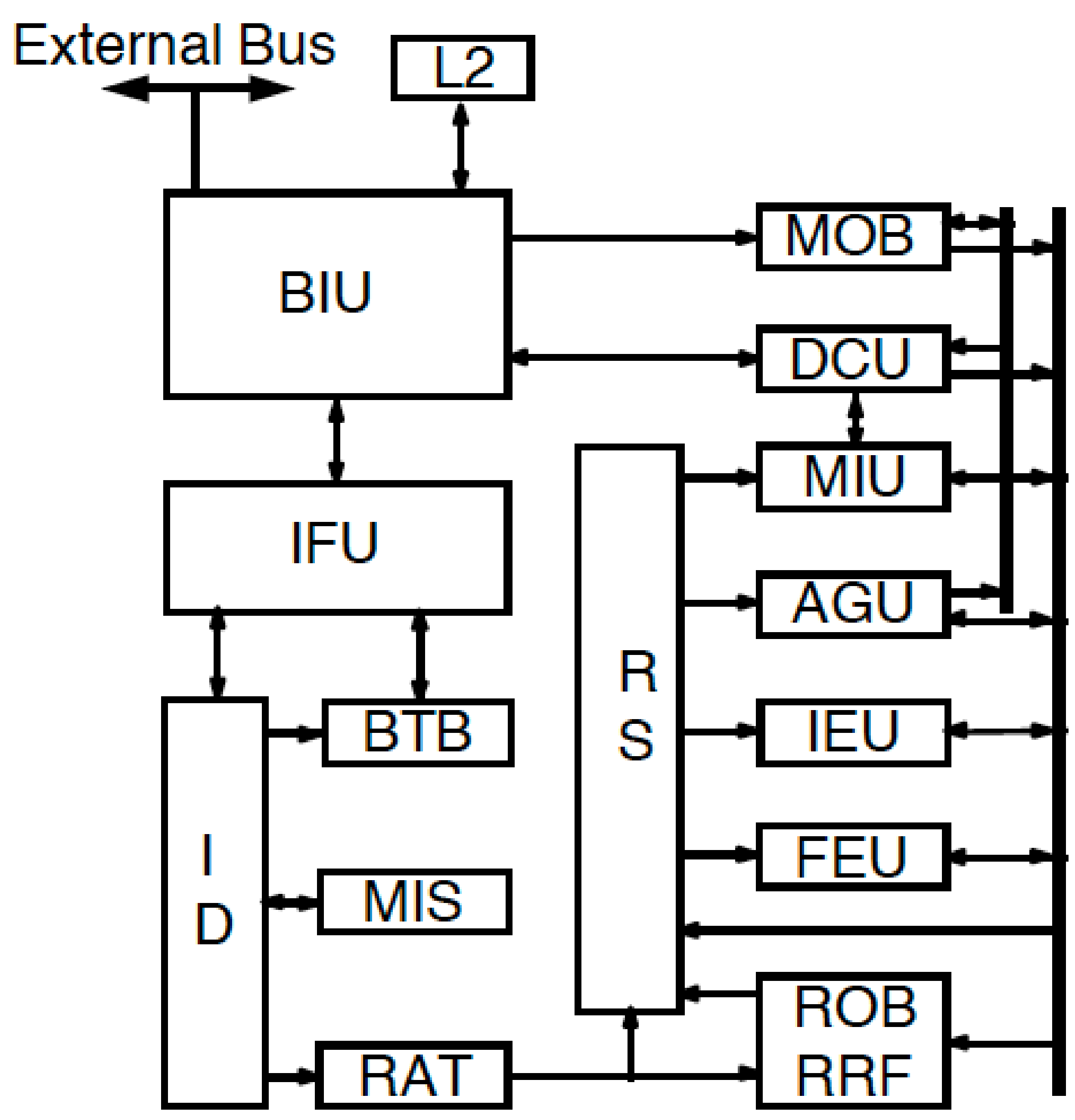
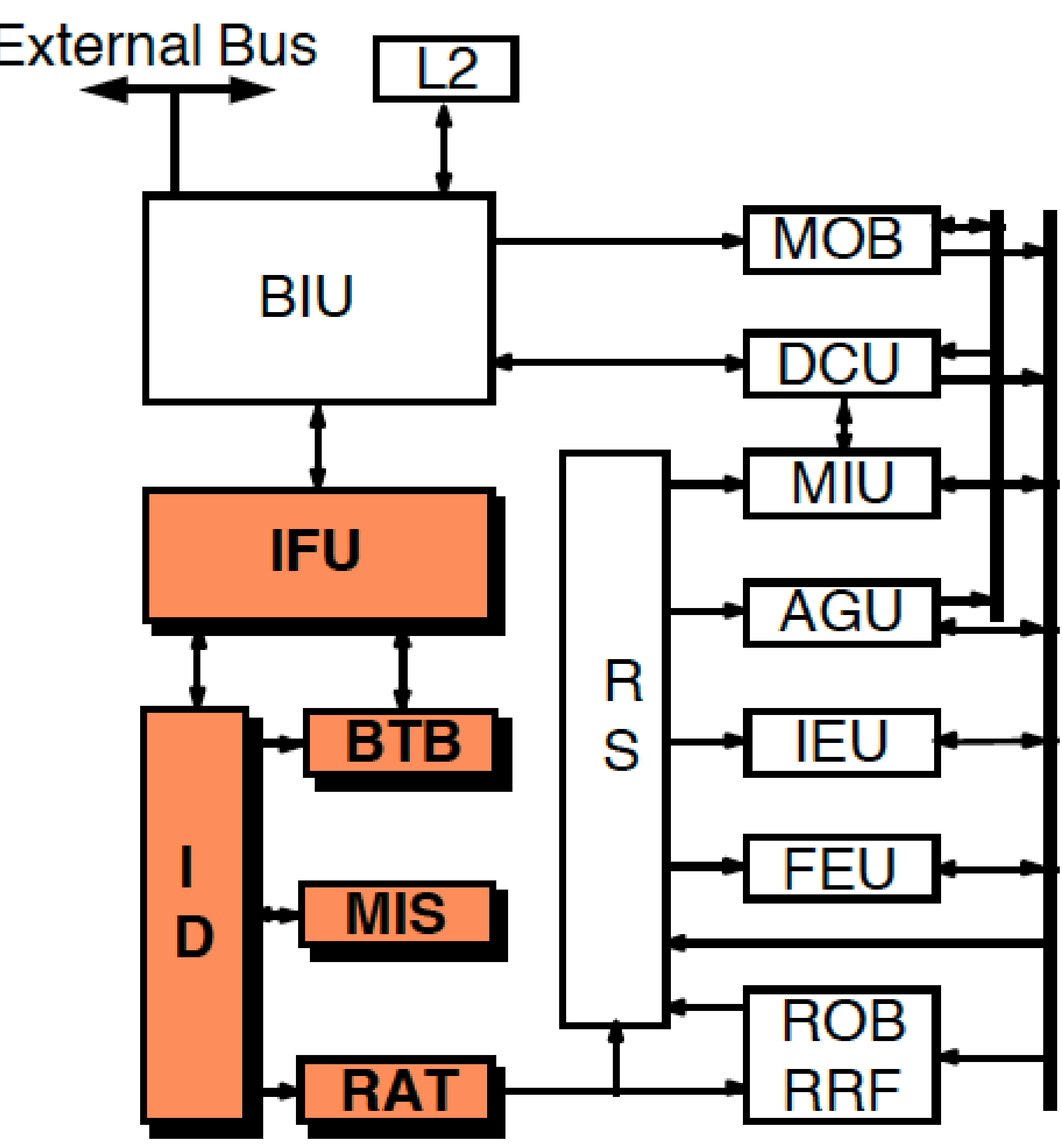


Figure 2-2. Functional Block Diagram of the P6 Family Processor Microarchitecture

Implementation: Microarchitecture

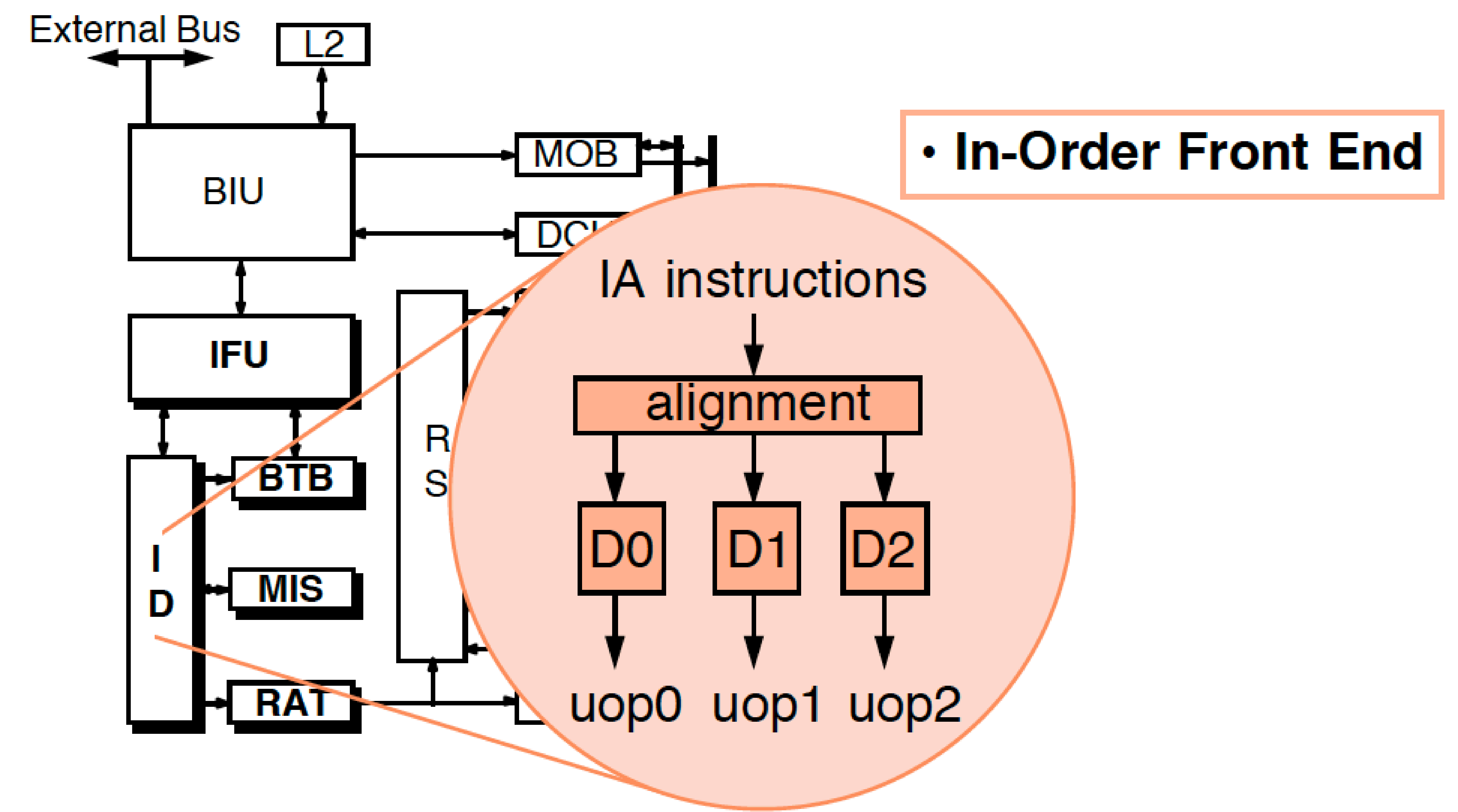


Implementation: Microarchitecture

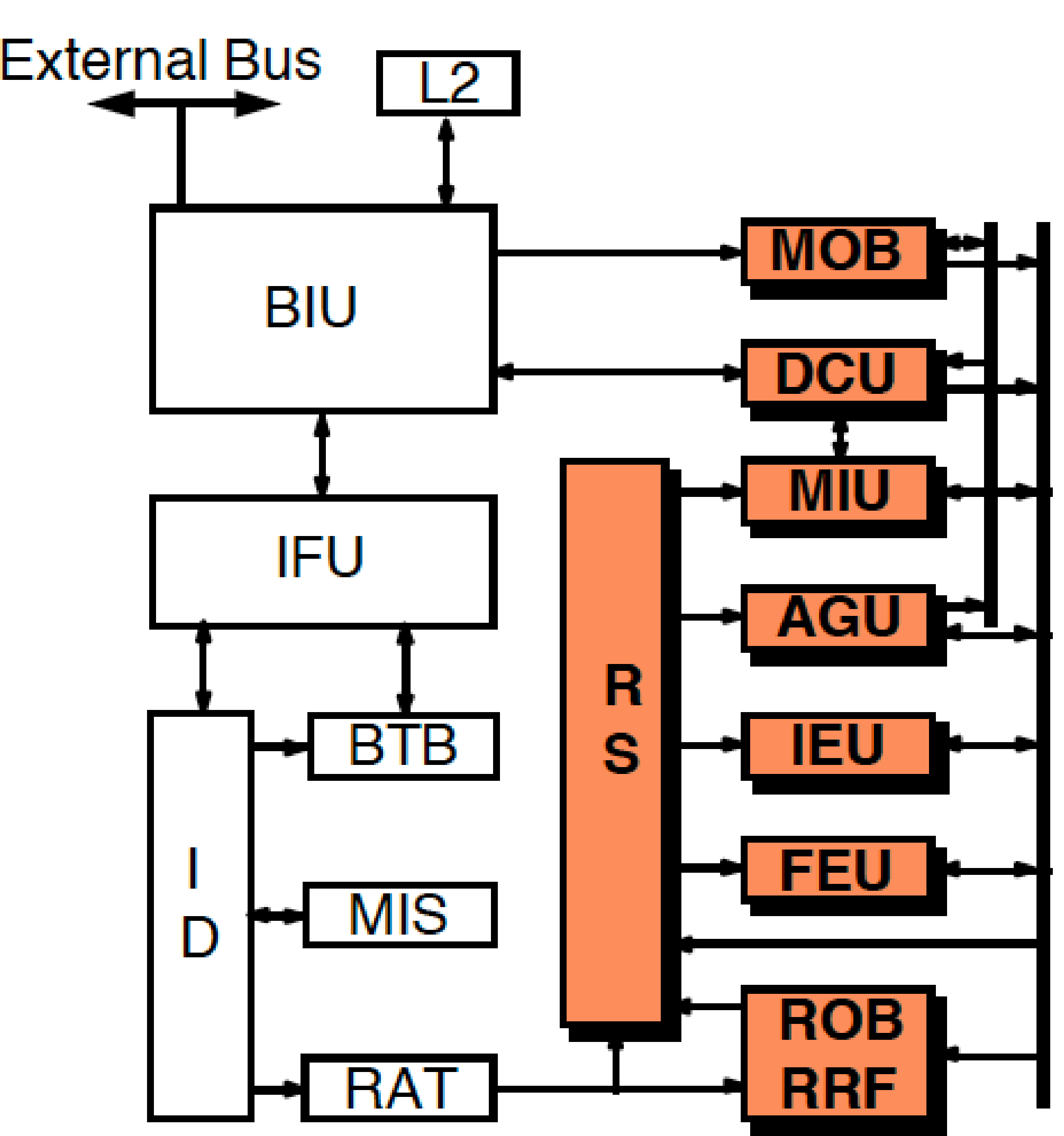


- In-Order Front End

Implementation: Microarchitecture

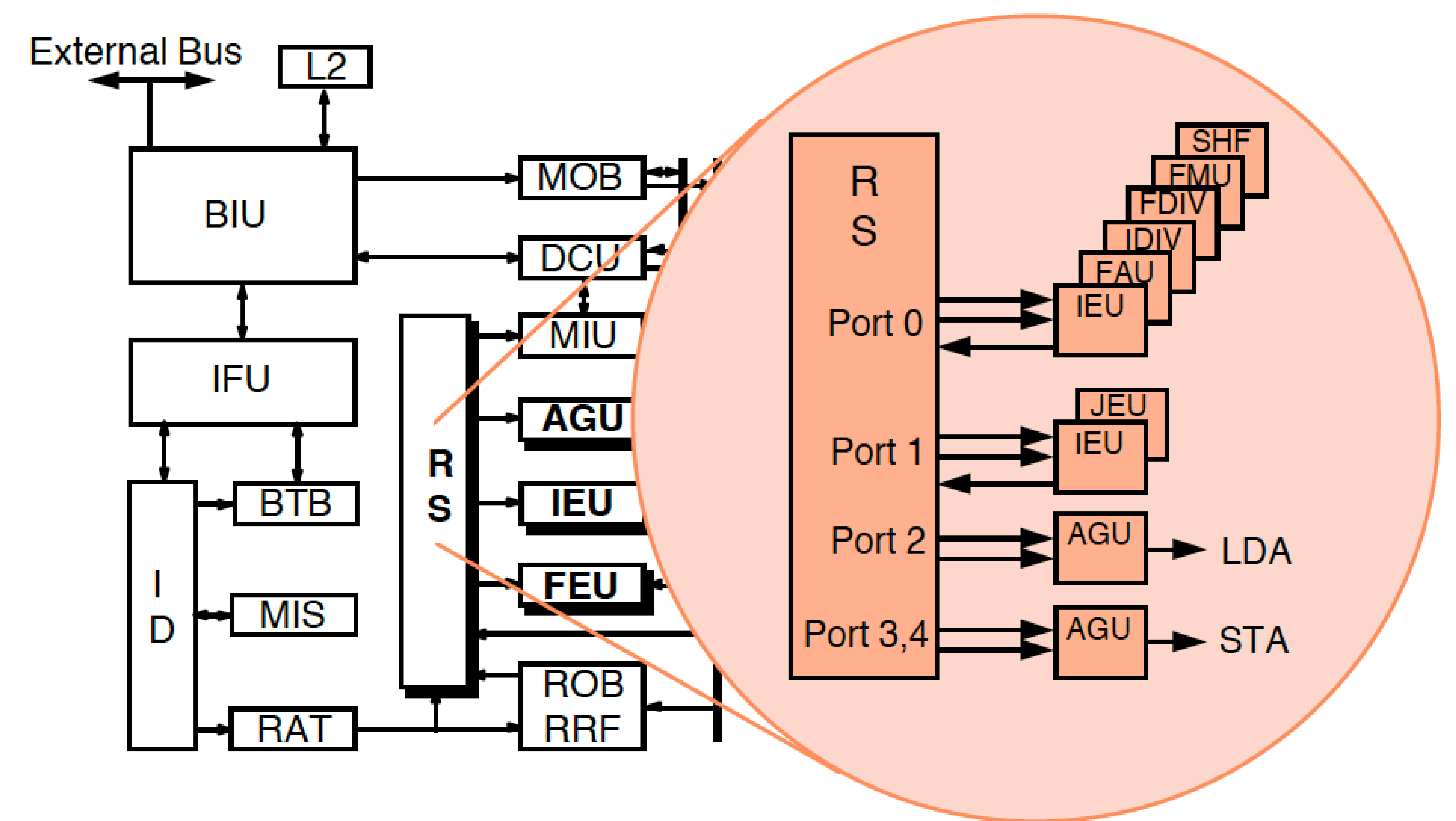


Implementation: Microarchitecture

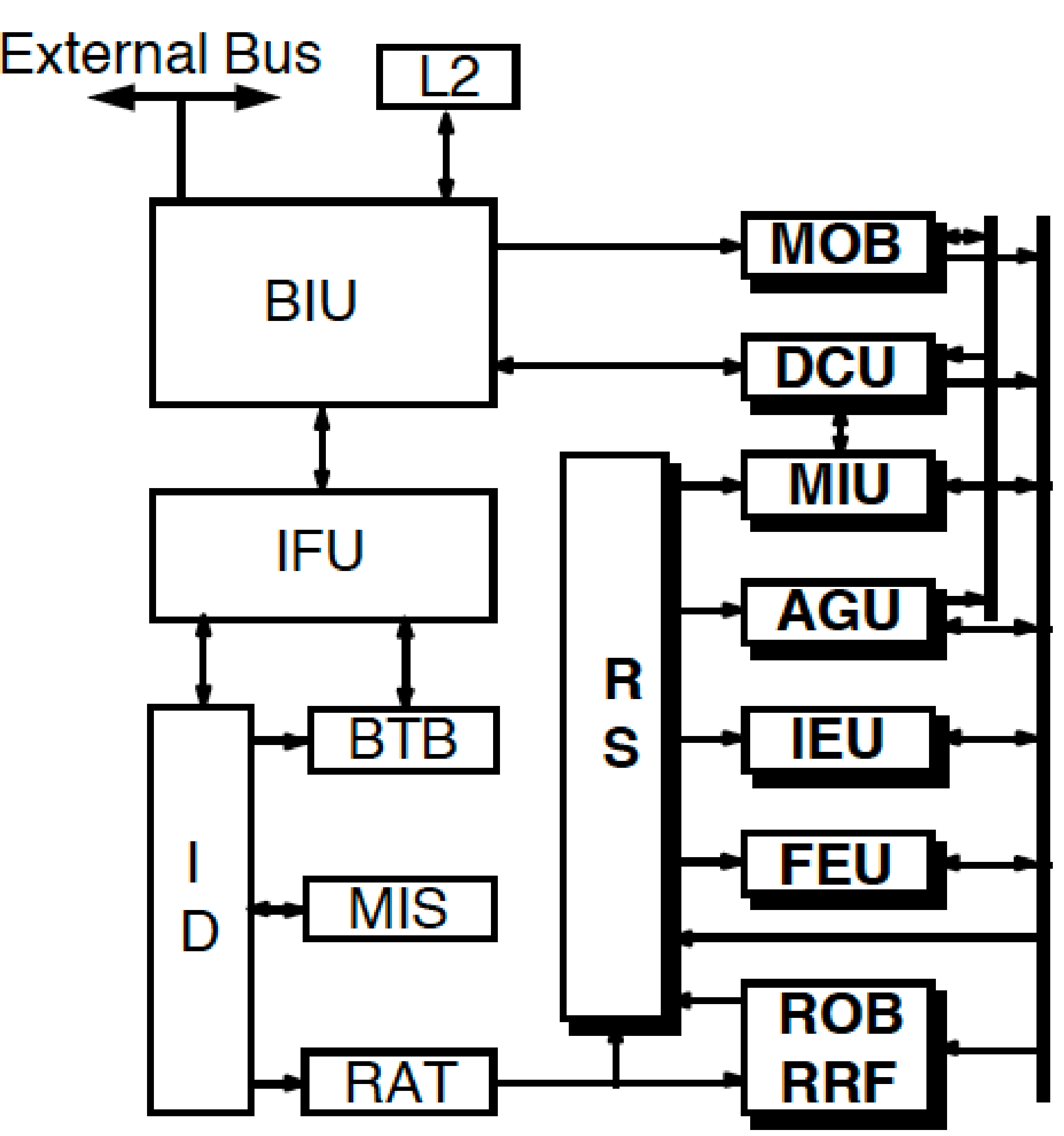


- In-Order Front End
- **Out-of-order Core**

Implementation: Microarchitecture

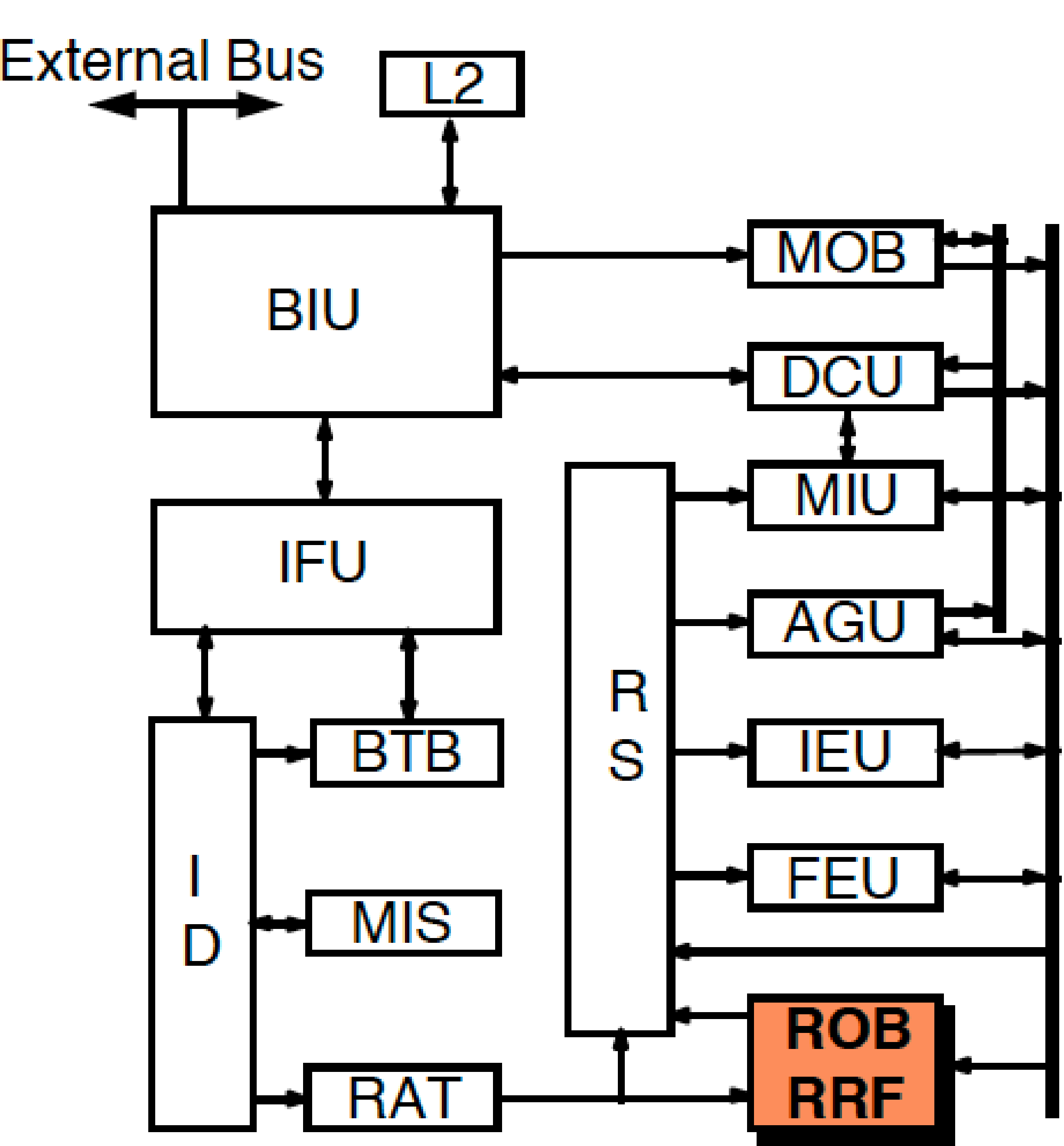


Implementation: Microarchitecture



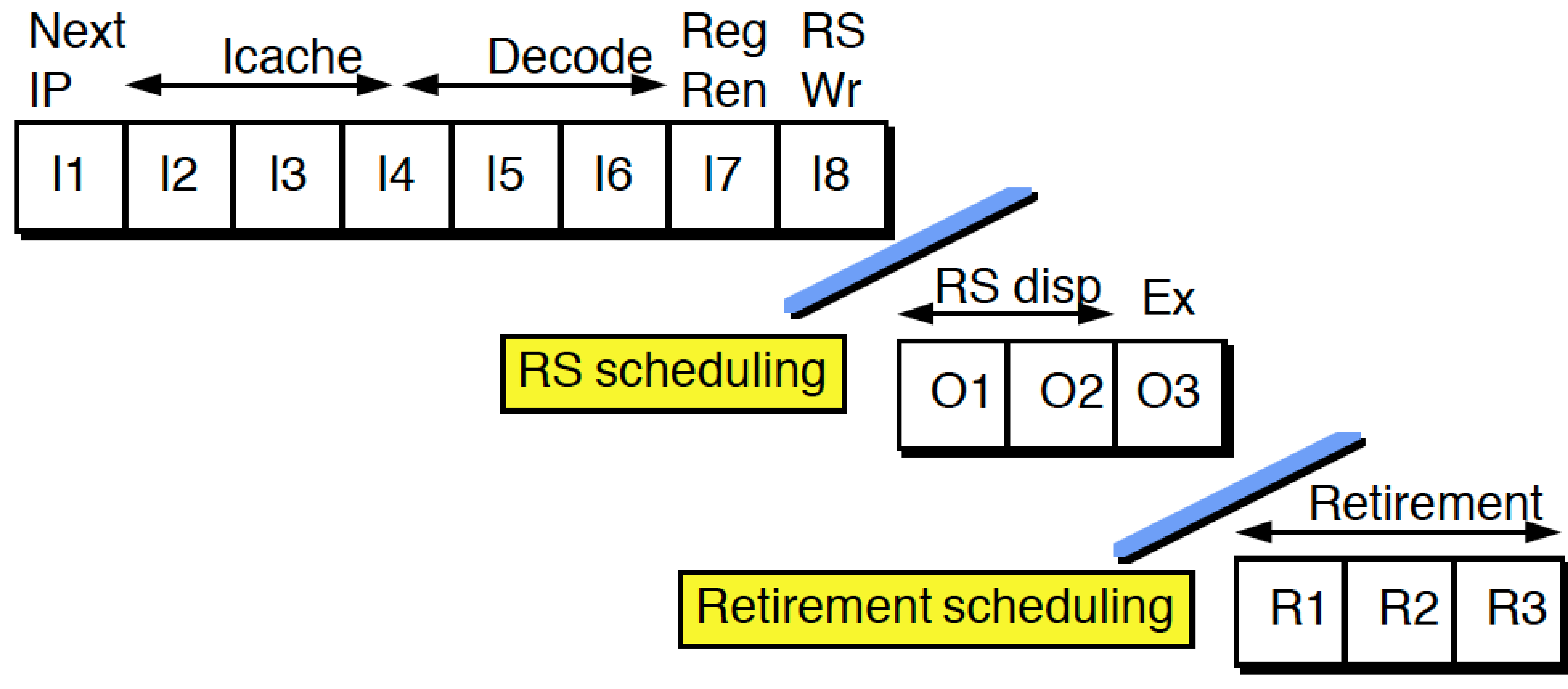
- In-Order Front End
- **Out-of-order Core**

Implementation: Microarchitecture

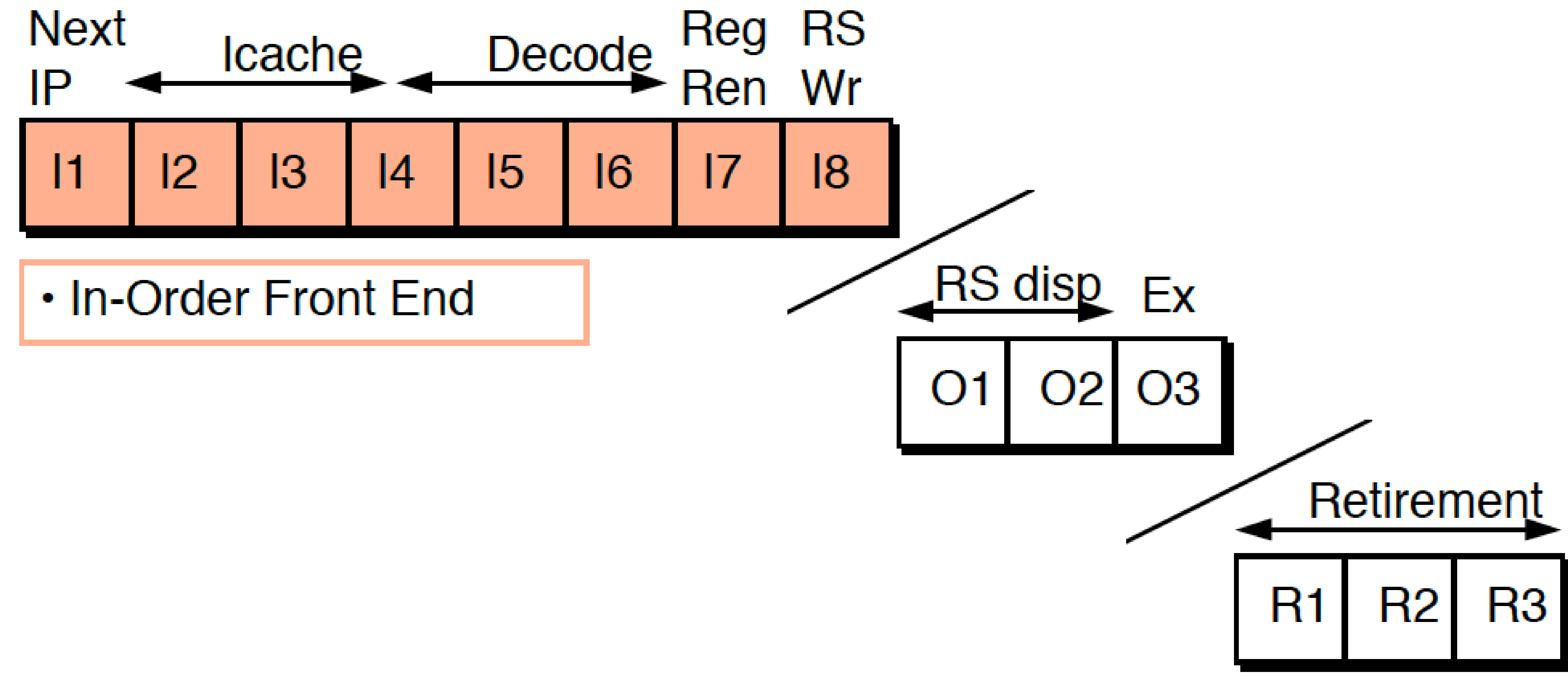


- In-Order Front End
- Out-of-order Core
- **In-Order Retirement**

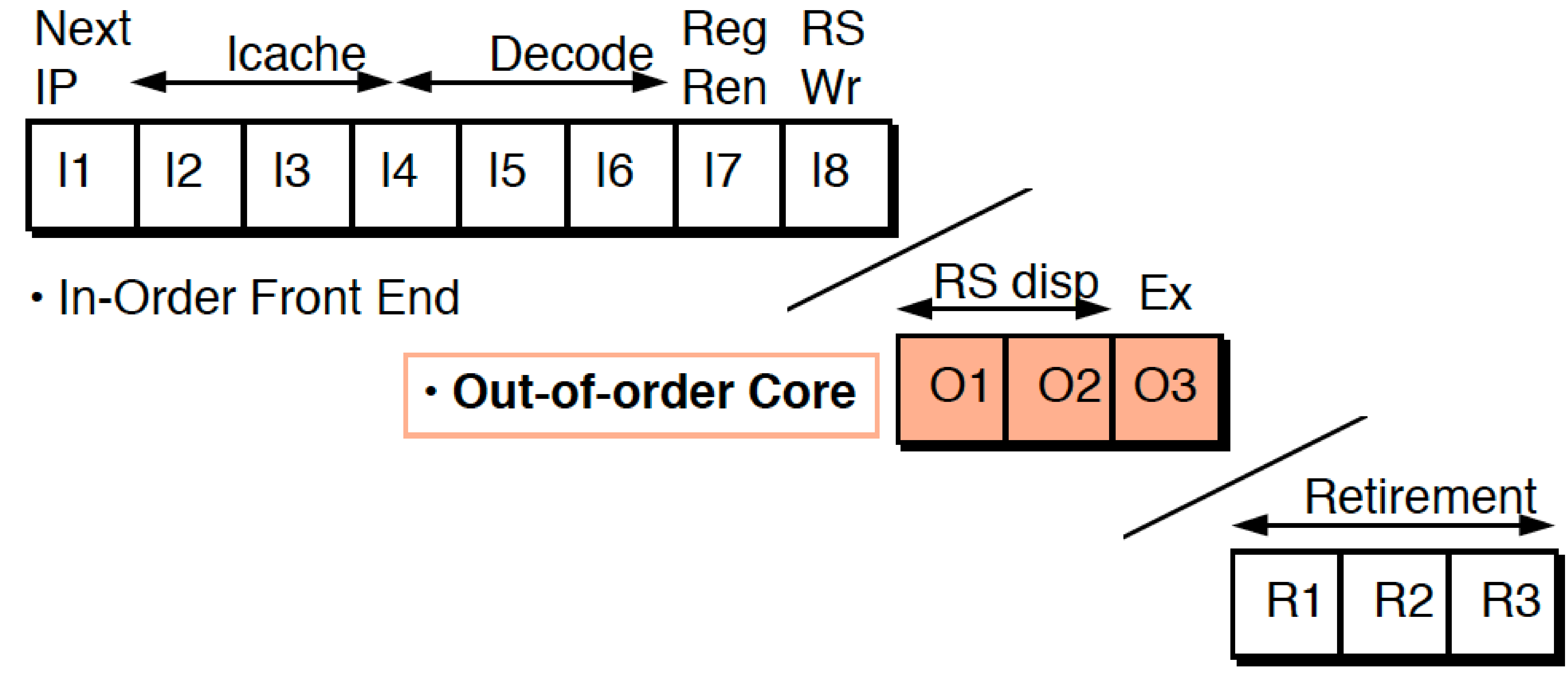
Implementation: Pipeline



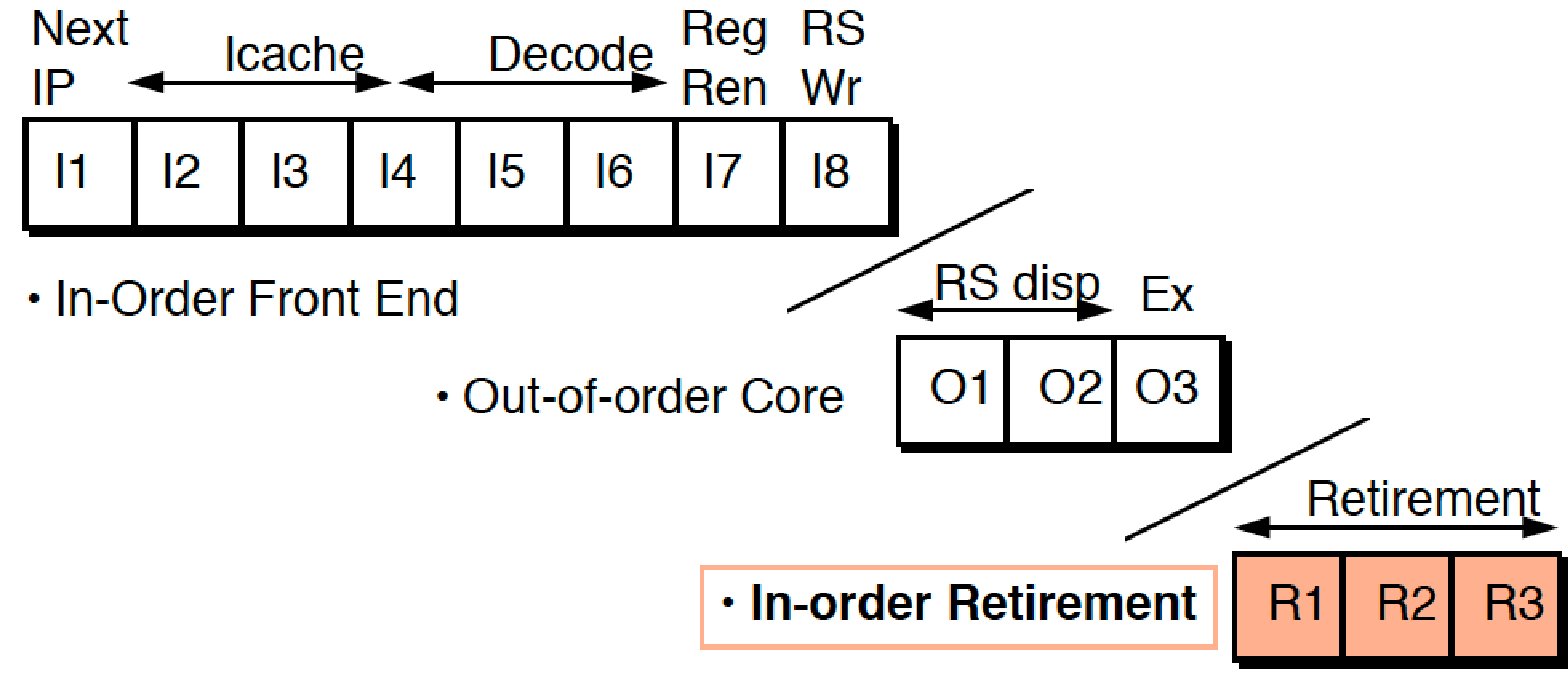
Implementation: Pipeline



Implementation: Pipeline



Implementation: Pipeline



Retrospective

- Most commercially successful microarchitecture in history
- Evolution
  - Pentium II/III, Xeon, etc.
    - Derivatives with on-chip L2, ISA extensions, etc.
  - Replaced by Pentium 4 as flagship in 2001
    - High frequency, deep pipeline, extreme speculation
  - Resurfaced as Pentium M in 2003
    - Initially a response to Transmeta in laptop market
    - Pentium 4 derivative (90nm Prescott) delayed, slow, hot
  - Core Duo, Core 2 Duo, Core i7 replaced Pentium 4

Microarchitectural Updates

- Pentium M (Banias), Core Duo (Yonah)
  - Micro-op fusion (also in AMD K7/K8)
    - Multiple uops in one: (add eax,[mem] => ld/alu), sta/std
    - These uops decode/dispatch/commit once, issue twice
  - Better branch prediction
    - Loop count predictor
    - Indirect branch predictor
  - Slightly deeper pipeline (12 stages)
    - Extra decode stage for micro-op fusion
    - Extra stage between issue and execute (for RS/PLRAM read)
  - Data-capture reservation station (payload RAM)
    - Clock gated for 32 (int) , 64 (fp), and 128 (SSE) operands

© Shen, Lipasti

Microarchitectural Updates

- Core 2 Duo (Merom)
  - 64-bit ISA from AMD K8
  - Macro-op fusion
    - Merge uops from two x86 ops
    - E.g. cmp, jne => cmpjne
  - 4-wide decoder (Complex + 3x Simple)
    - Peak x86 decode throughput is 5 due to macro-op fusion
  - Loop buffer
    - Loops that fit in 18-entry instruction queue avoid fetch/decode overhead
  - Even deeper pipeline (14 stages)
  - Larger reservation station (32), instruction window (96)
  - Memory dependence prediction

© Shen, Lipasti

Microarchitectural Updates

- Nehalem (Core i7/i5/i3)
  - RS size 36, ROB 128
  - Loop cache up to 28 uops
  - L2 branch predictor
  - L2 TLB
  - I$ and D$ now 32K, L2 back to 256K, inclusive L3 up to 8M
  - Simultaneous multithreading
  - RAS now renamed (repaired)
  - 6 issue, 48 load buffers, 32 store buffers
  - New system interface (QPI) – finally dropped front-side bus
  - Integrated memory controller (up to 3 channels)
  - New STTNI instructions for string/text handling

Microarchitectural Updates

- Sandybridge/Ivy Bridge (2nd-3rd generation Core i7)
  - On-chip integrated graphics (GPU)
  - Decoded uop cache up to 1.5K uops, handles loops, but more general
  - 54-entry RS, 168-entry ROB
  - Physical register file: 144 FP, 160 integer
  - 256-bit AVX units: 8 DPFLOP/cycle, 16 SPFLOP/cycle
  - 2 general AGUs enable 2 ld/cycle, 2 st/cycle or any combination, 2x128-bit load path from L1 D$

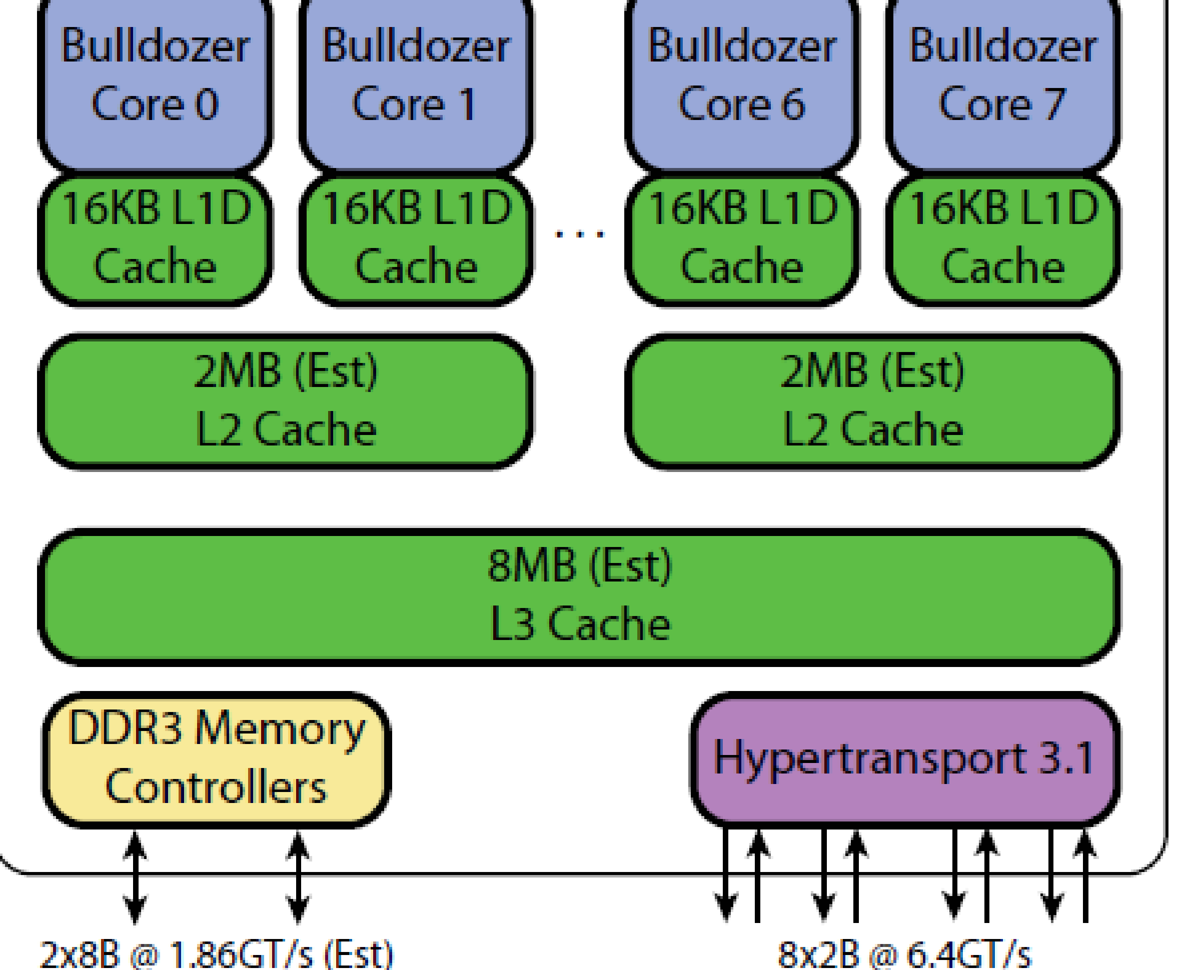
© Shen, Lipasti

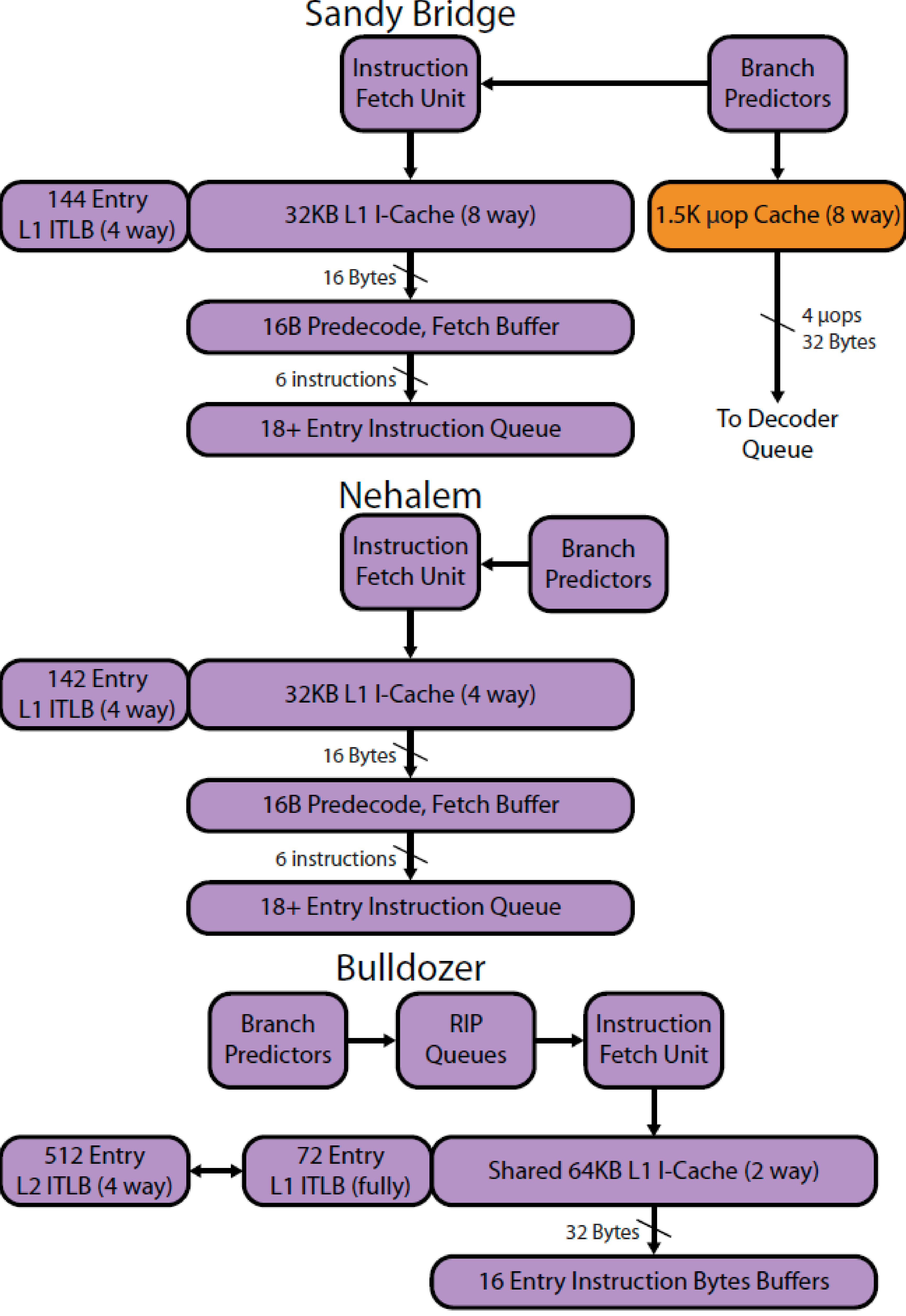
Microarchitectural Updates

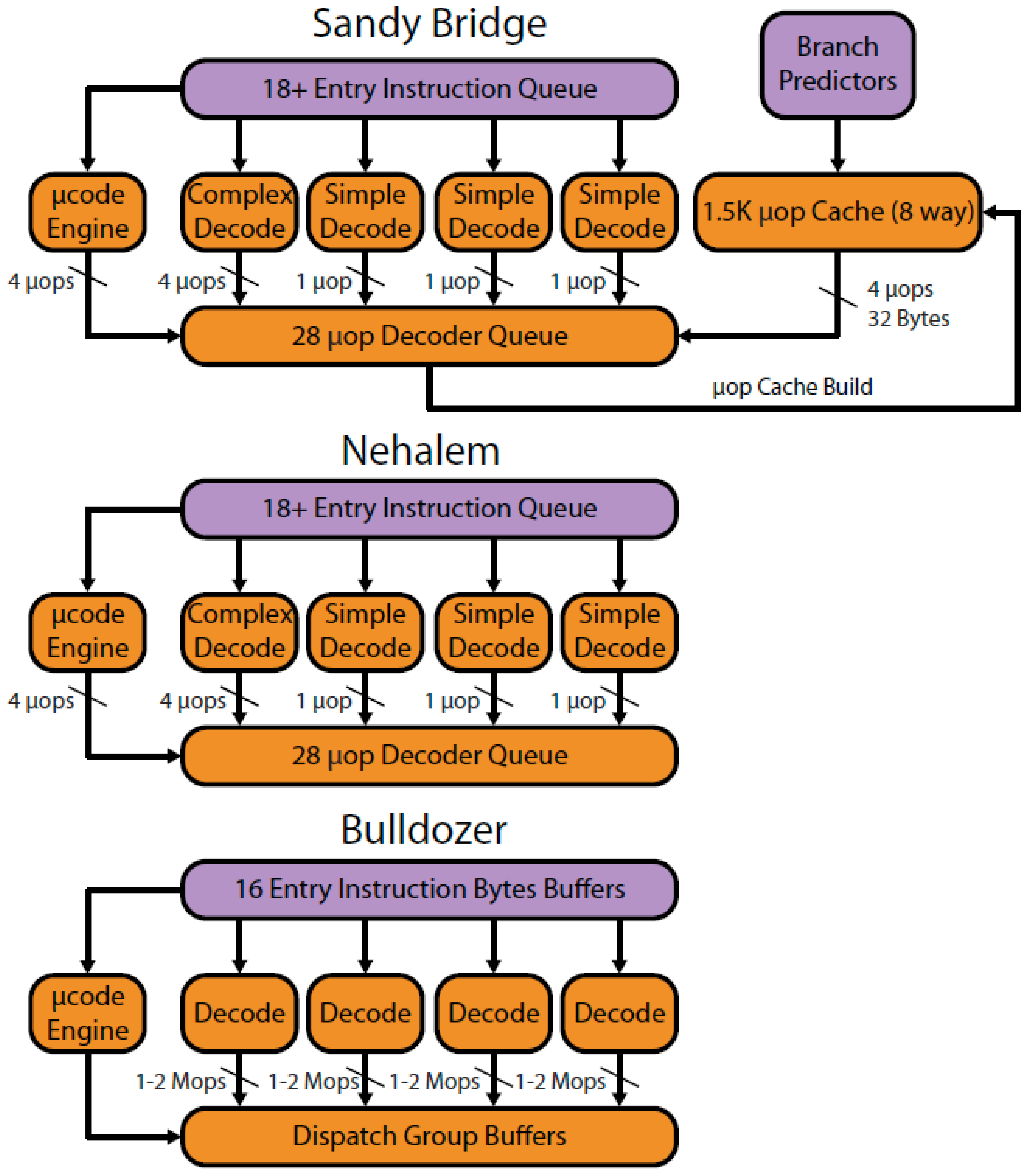
- Haswell/Broadwell/Skylake: wider & deeper
  - 8-wide issue (up from 6 wide)
  - 4th integer ALU, third AGU, second branch unit
  - 60-entry RS, 192-entry ROB
  - 72-entry load queue/42-entry store queue
  - Physical register file: 168 FP, 168 integer
  - Doubled FP throughput (32 SP/16 DP)
  - Load/store bandwidth to L1 doubled (64B/32B)
  - TSX (transactional memory)
  - Integrated voltage regulator

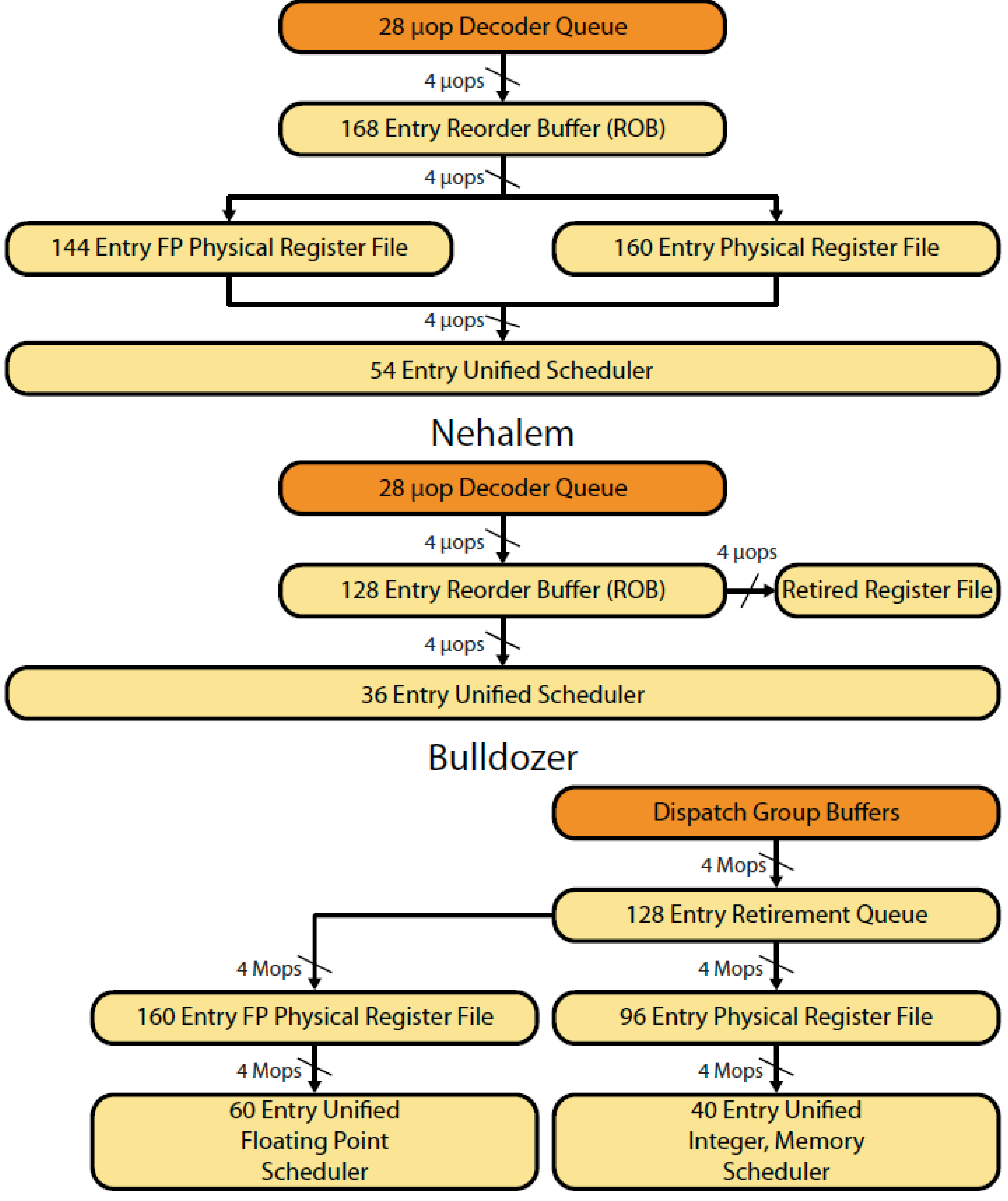
Sandy Bridge Client
3.4 GHz Sandy Br Core 0 (Est)
32KB L1D Cache
256KB L2 Cache
3.4 GHz Sandy Br Core 3 (Est)
32KB L1D Cache
256KB L2 Cache
0.85 GHz Gen 6 GPU (Est)
8MB L3 Cache
DDR3 Memory Controllers
Direct Media Interface
PCI-E 2.0 Controller
2x8B @ 2.13GT/s (Est)
2x4b @ 2.5GT/s
2x16b @ 5GT/s

Westmere
3.33 GHz Westmere Core 0
32KB L1D Cache
256KB L2 Cache
3.33 GHz Westmere Core 5
32KB L1D Cache
256KB L2 Cache
12MB L3 Cache
DDR3 Memory Controllers
Quick Path Interconnect
3x8B @ 1.33GT/s
4x20b @ 6.4GT/s

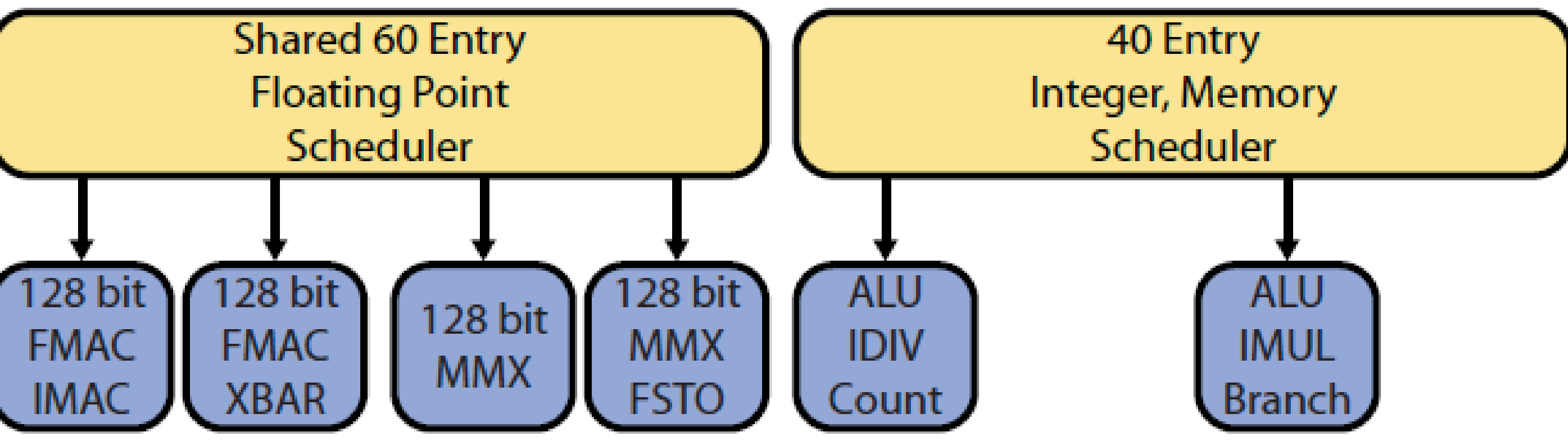

Interlagos Node
Bulldozer Core 0
16KB L1D Cache
Bulldozer Core 1
16KB L1D Cache
Bulldozer Core 6
16KB L1D Cache
Bulldozer Core 7
16KB L1D Cache
2MB (Est) L2 Cache
2MB (Est) L2 Cache
8MB (Est) L3 Cache
DDR3 Memory Controllers
Hypertransport 3.1
2x8B @ 1.86GT/s (Est)
8x2B @ 6.4GT/s


Sandy Bridge
Instruction Fetch Unit
Branch Predictors
144 Entry L1 ITLB (4 way)
32KB L1 I-Cache (8 way)
1.5K µop Cache (8 way)
16 Bytes
16B Predecode, Fetch Buffer
4 µops
32 Bytes
6 instructions
18+ Entry Instruction Queue
To Decoder Queue
Nehalem
Instruction Fetch Unit
Branch Predictors
142 Entry L1 ITLB (4 way)
32KB L1 I-Cache (4 way)
16 Bytes
16B Predecode, Fetch Buffer
6 instructions
18+ Entry Instruction Queue
Bulldozer
Branch Predictors
RIP Queues
Instruction Fetch Unit
512 Entry L2 ITLB (4 way)
72 Entry L1 ITLB (fully)
Shared 64KB L1 I-Cache (2 way)
32 Bytes
16 Entry Instruction Bytes Buffers

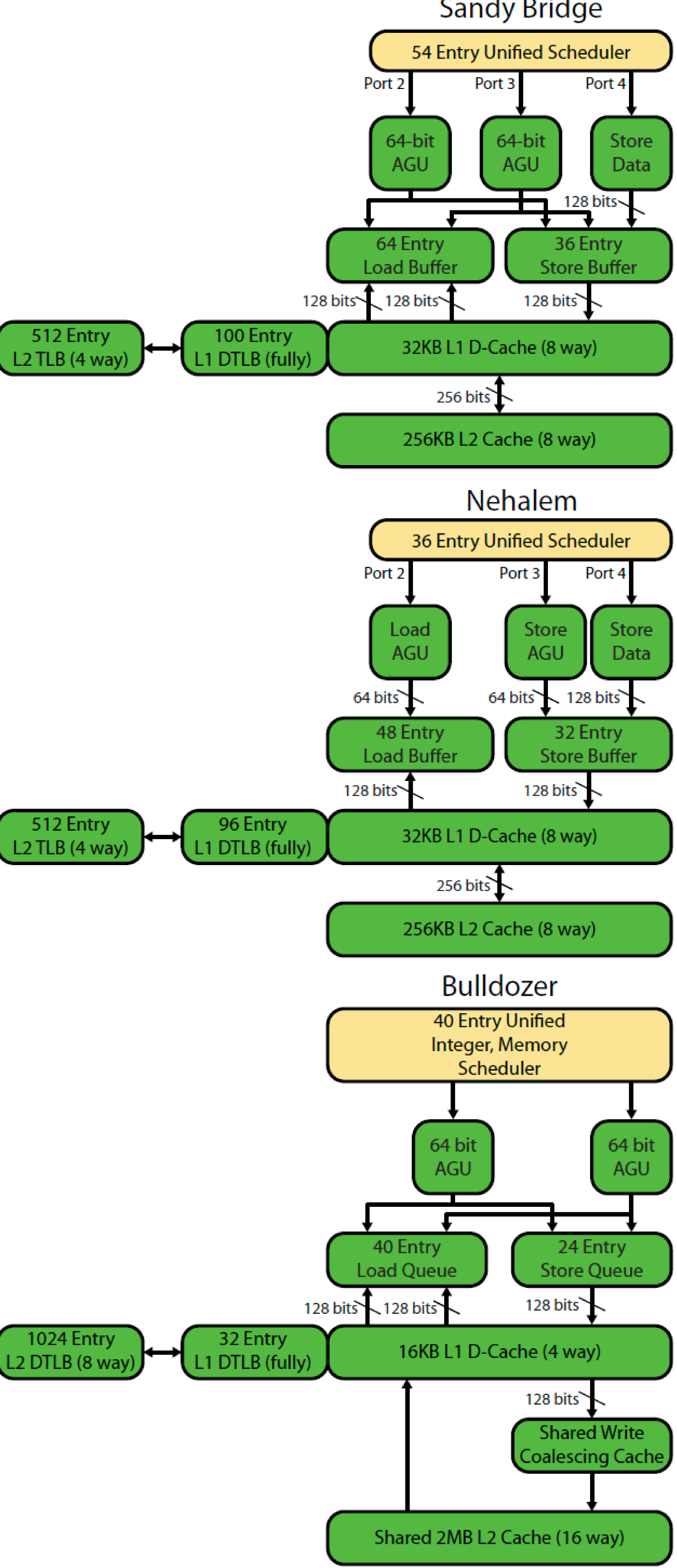

Sandy Bridge
18+ Entry Instruction Queue
Branch Predictors
µcode Engine
Complex Decode
Simple Decode
Simple Decode
Simple Decode
1.5K µop Cache (8 way)
4 µops
4 µops
1 µop
1 µop
1 µop
4 µops
32 Bytes
28 µop Decoder Queue
µop Cache Build
Nehalem
18+ Entry Instruction Queue
µcode Engine
Complex Decode
Simple Decode
Simple Decode
Simple Decode
4 µops
4 µops
1 µop
1 µop
1 µop
28 µop Decoder Queue
Bulldozer
16 Entry Instruction Bytes Buffers
µcode Engine
Decode
Decode
Decode
Decode
1-2 Mops
1-2 Mops
1-2 Mops
1-2 Mops
Dispatch Group Buffers



Sandy Bridge
28 µop Decoder Queue
4 µops
168 Entry Reorder Buffer (ROB)
4 µops
144 Entry FP Physical Register File
160 Entry Physical Register File
4 µops
54 Entry Unified Scheduler
Nehalem
28 µop Decoder Queue
4 µops
128 Entry Reorder Buffer (ROB)
4 µops
Retired Register File
4 µops
36 Entry Unified Scheduler
Bulldozer
Dispatch Group Buffers
4 Mops
128 Entry Retirement Queue
4 Mops
160 Entry FP Physical Register File
4 Mops
96 Entry Physical Register File
4 Mops
60 Entry Unified Floating Point Scheduler
4 Mops
40 Entry Unified Integer, Memory Scheduler



Sandy Bridge
54 Entry Unified Scheduler
Port 0
Port 1
Port 5
ALU LEA Shift
SIMD MUL Shift
ALU LEA MUL
ALU Shift Branch
SIMD ALU Shuffle
256-bit FMUL Blend
256-bit FADD
SIMD ALU Shuffle
256-bit Shuffle Blend
Nehalem
36 Entry Unified Scheduler
Port 0
Port 1
Port 5
ALU Shift
SIMD ALU Shuffle
ALU LEA MUL
ALU Shift Branch
SIMD ALU Shuffle
128-bit FMUL Shuffle
128-bit FADD
SIMD MUL Shift
128-bit FMove
Bulldozer
Shared 60 Entry Floating Point Scheduler
40 Entry Integer, Memory Scheduler
128 bit FMAC IMAC
128 bit FMAC XBAR
128 bit MMX
128 bit MMX FSTO
ALU IDIV Count
ALU IMUL Branch


Sandy Bridge
54 Entry Unified Scheduler
Port 2
Port 3
Port 4
64-bit AGU
64-bit AGU
Store Data
128 bits
64 Entry Load Buffer
36 Entry Store Buffer
128 bits
128 bits
128 bits
512 Entry L2 TLB (4 way)
100 Entry L1 DTLB (fully)
32KB L1 D-Cache (8 way)
256 bits
256KB L2 Cache (8 way)
Nehalem
36 Entry Unified Scheduler
Port 2
Port 3
Port 4
Load AGU
Store AGU
Store Data
64 bits
64 bits
128 bits
48 Entry Load Buffer
32 Entry Store Buffer
128 bits
128 bits
512 Entry L2 TLB (4 way)
96 Entry L1 DTLB (fully)
32KB L1 D-Cache (8 way)
256 bits
256KB L2 Cache (8 way)
Bulldozer
40 Entry Unified Integer, Memory Scheduler
64 bit AGU
64 bit AGU
40 Entry Load Queue
24 Entry Store Queue
128 bits
128 bits
128 bits
1024 Entry L2 DTLB (8 way)
32 Entry L1 DTLB (fully)
16KB L1 D-Cache (4 way)
128 bits
Shared Write Coalescing Cache
Shared 2MB L2 Cache (16 way)

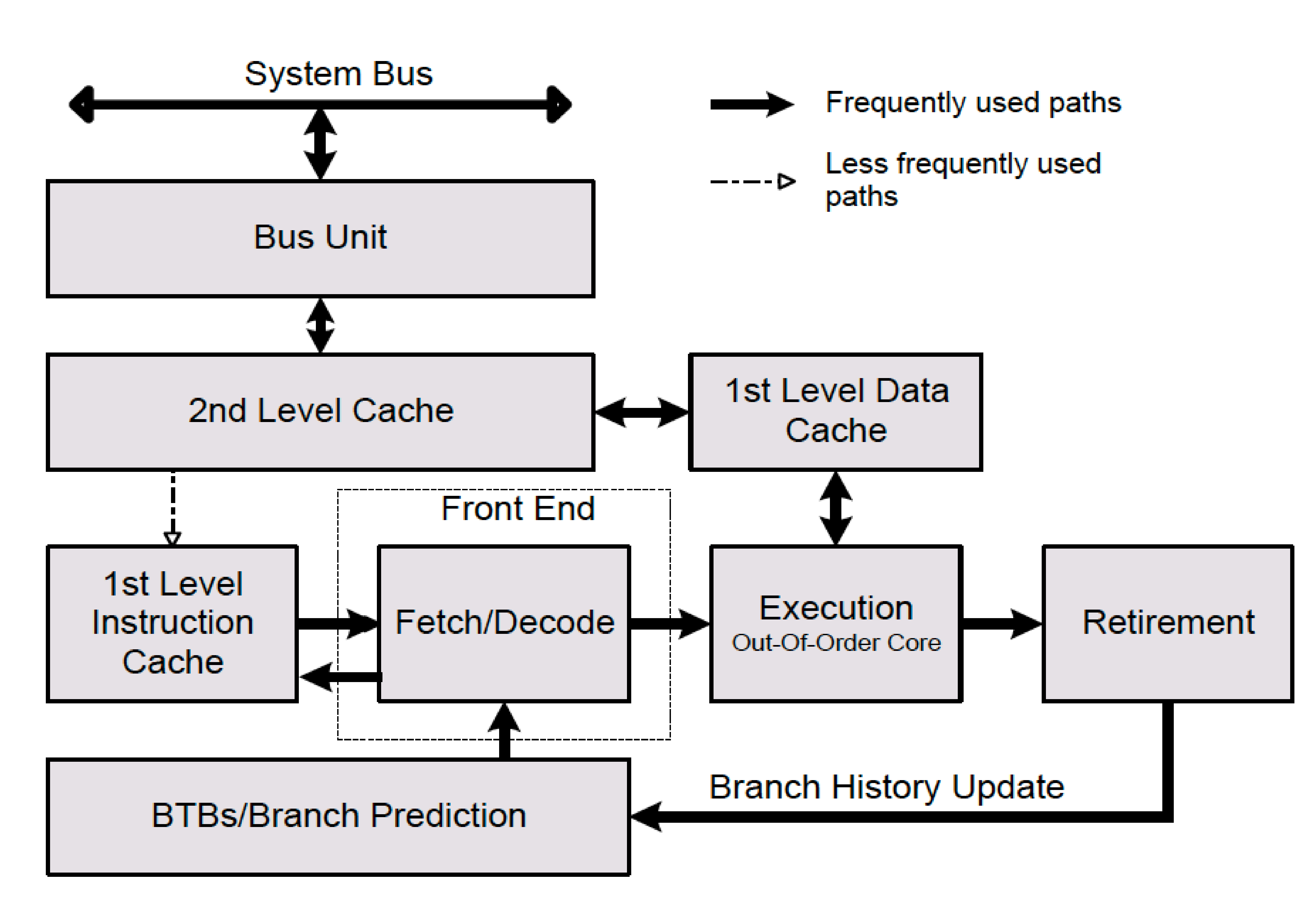


Figure 2-1. The P6 Processor Micro-Architecture with Advanced Transfer Cache Enhancement

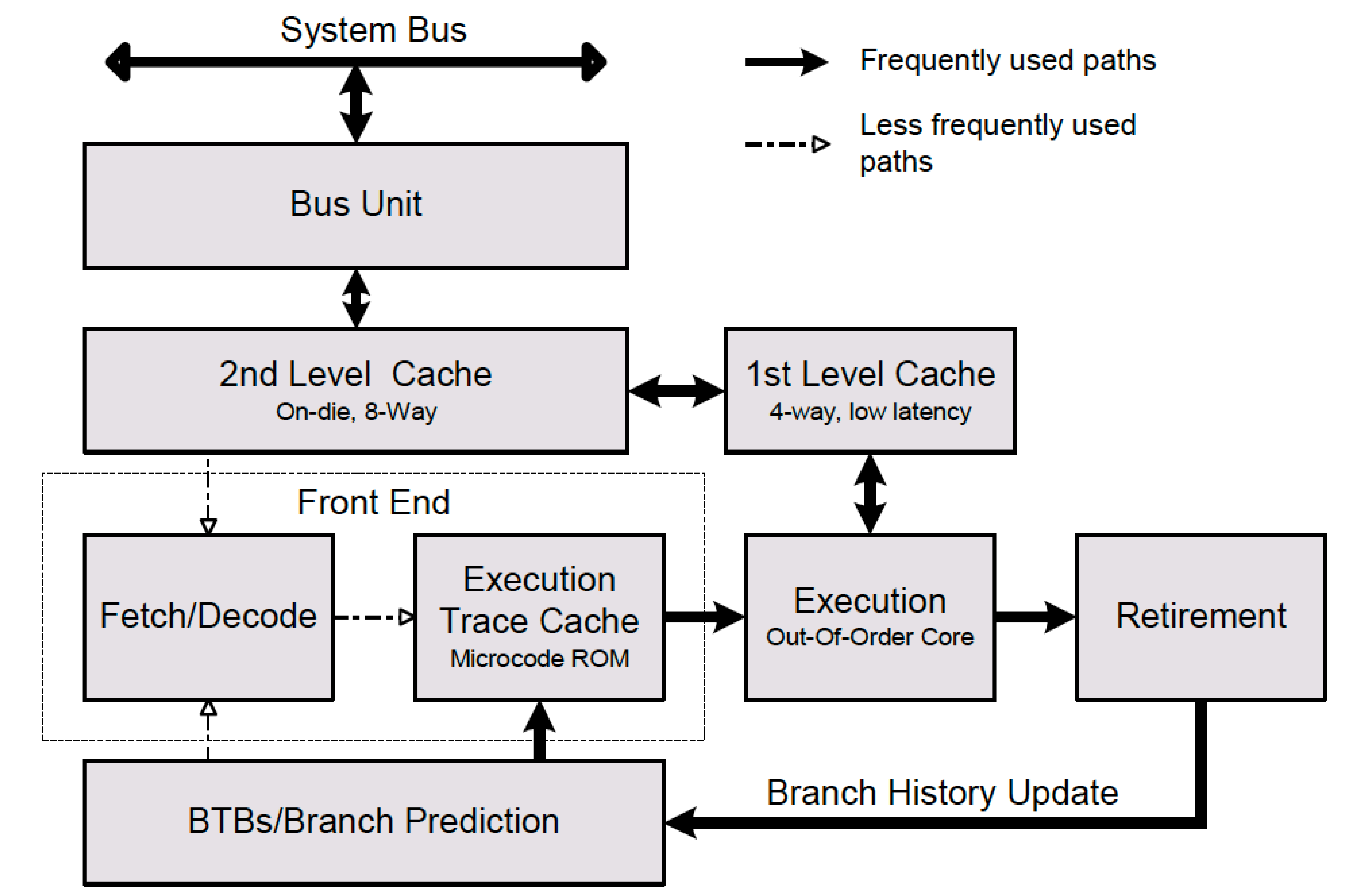


Figure 2-2. The Intel NetBurst Micro-Architecture

I felt that we had hit a home run with the P6 project, and that it was (and is) successful far beyond what anyone had a right to expect. That project was just plain fun, working so closely with so many incredible engineers.

But the Willamette project was not fun. Part of the problem for me was the design manager, Randy Steck's successor, with whom I had serious differences of opinion about the chip development process. But an even bigger problem was the corporate interference I have detailed elsewhere in this book. And while I was off trying to keep the company from destructively interfering with Pentium 4 (through quantum entanglement with the Itanium Processor Family) the chip itself was sailing into complexity waters that seem, in retrospect, far too deep for what they were worth. Beyond all of that, however, was a looming thermal power wall that was no longer off in the distance, as in P6, but instead was casting its long, ugly shadow directly over everything we did. That experience was primarily why I was so sure I did not want to work on any high-clock-rate chips beyond Willamette. I just did not think there would be enough end-user performance payoff to justify the nightmarish complexity incurred in a power-dominated, high-performance design.

I think the future is mobile. In particular, I think the future is battery- or fuel-cell-operated and that what will be valued in the mainstream a decade or two from now will be *sufficient* performance to accomplish some desired end goal (like HDTV playback or GPS reception) at very long battery life. This is a daunting goal, and so far removed from historical CPU design goals (especially the highest possible performance at all costs) that it will take a minor miracle to get existing teams to achieve it. Without inspired direction from corporate leadership, it is not going to happen. Business as usual is no longer going to work. I left Intel because I did not want to be the one who proved that.

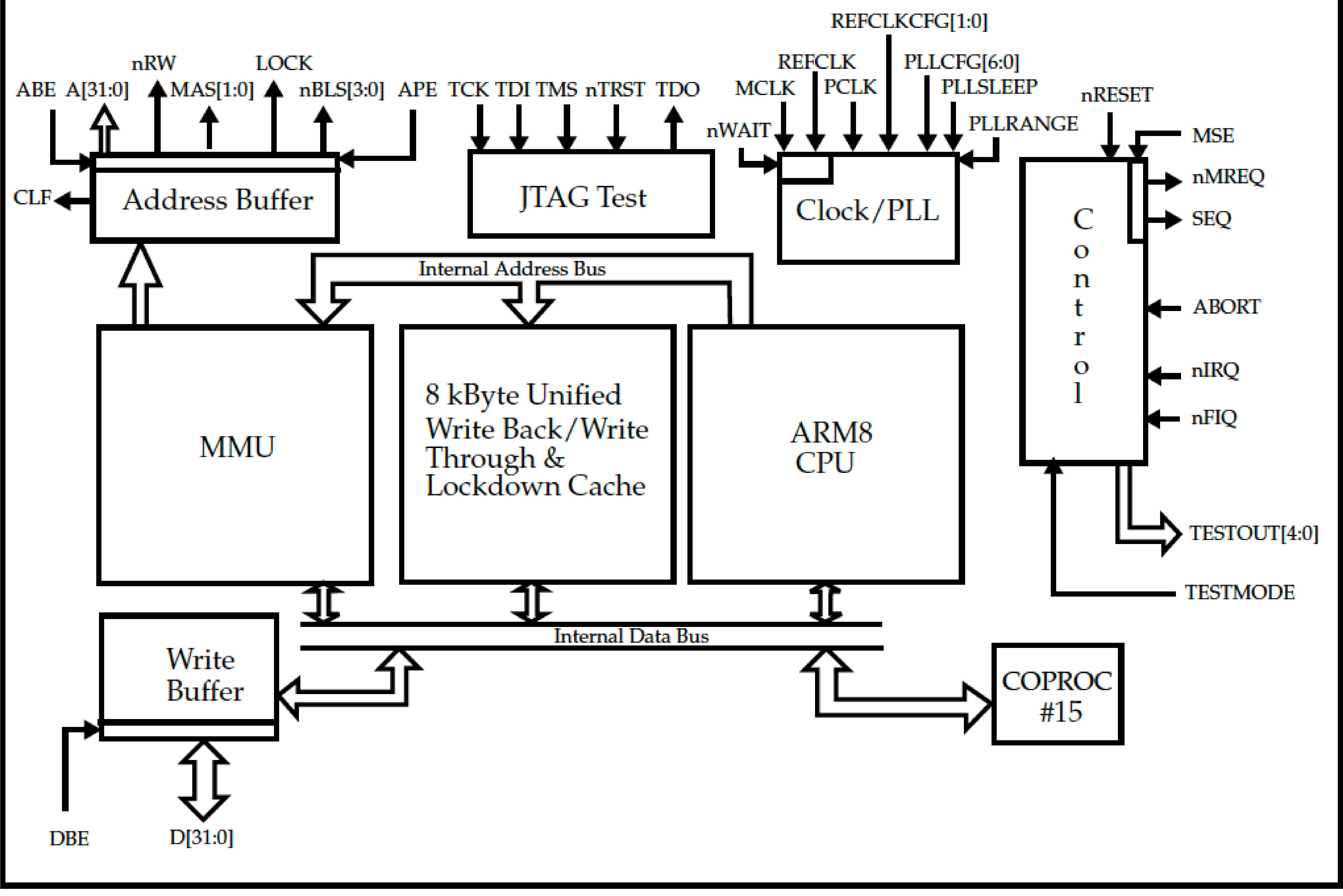


Figure 1-1: ARM810 block diagram

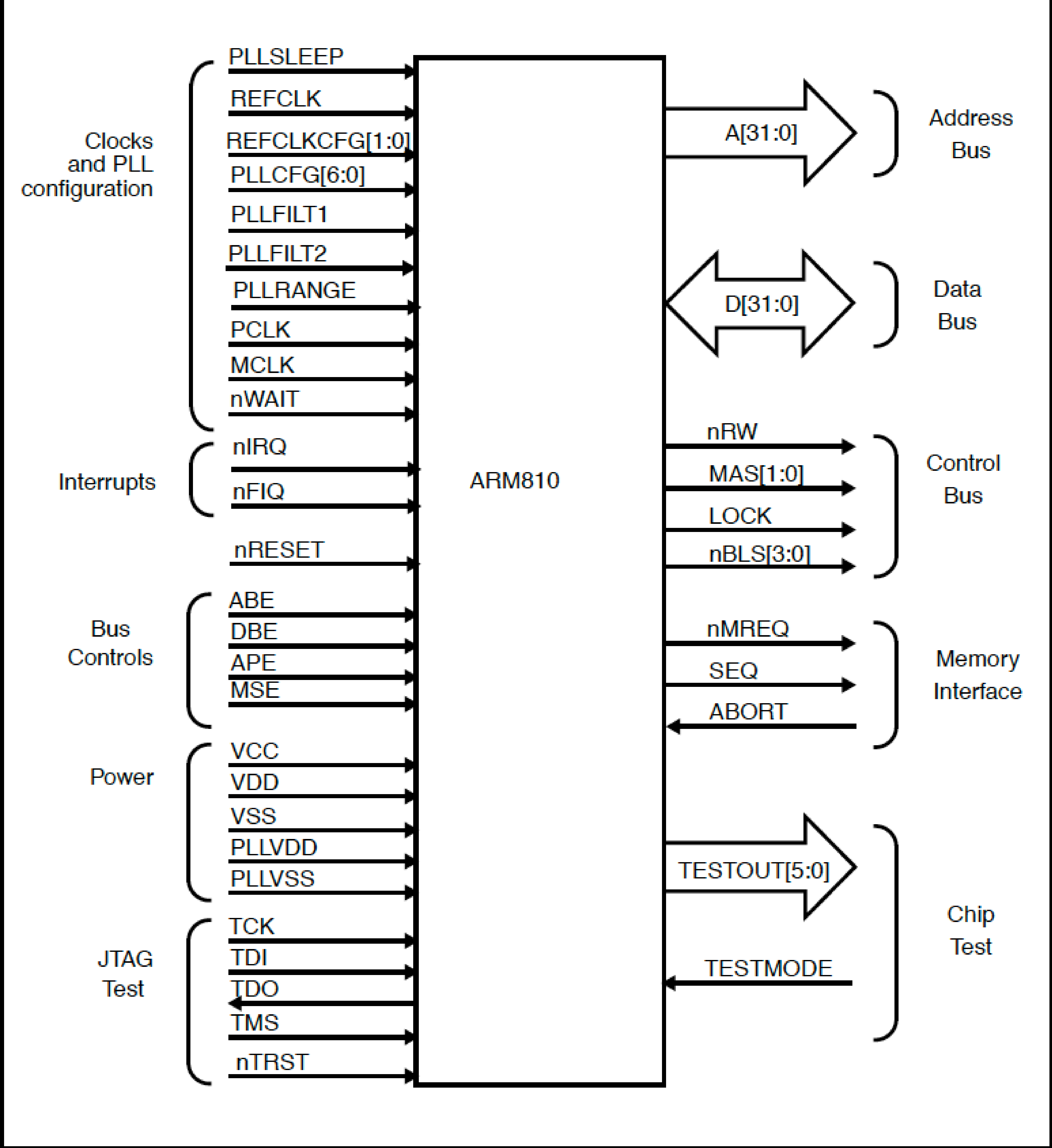


Figure 2-1: ARM810 functional diagram

Key to signal types:

I	Input
OCZ	Output, CMOS levels, tristateable
IOCZ	Input/output tristateable, CMOS levels
ICK	Clock input

A[31:0]	OCZ	Address Bus. This bus signals the address requested for memory accesses. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
ABE	I	Address bus enable. When this input is LOW, the address bus **A[31:0]**, **MAS[1:0]**, **CLF**, **nBLS[3:0]**, **nRW** and **LOCK** are put into a high impedance state (Note 1).
ABORT	I	External abort. Allows the memory system to tell the processor that a requested access has failed. Only monitored when ARM810 is accessing external memory.
APE	I	Address pipeline enable control input. When **APE** is HIGH, address and address-timed outputs are generated with normal pipeliined timing, where a new address is generated in the second phase of the bus clock (**MCLK** HIGH or **PCLK** LOW). Taking **APE** LOW delays these signals by one clock phase so they change in the first phase of the following bus cycle (**MCLK** LOW or **PCLK** HIGH). See the descriptions for **MCLK**/**PCLK** and ***Chapter 11, ARM810 Clocking*** for bus clock information. The address-timed signals are **A[31:0]**, **MAS[1:0]**, **nBLS[3:0**], **CLF**, **LOCK** and **nRW**.
CLF	O	Cache line fill. **CLF** HIGH indicates that the current read cycle is cacheable. **CLF** is always HIGH for writes. This signal may be used to indicate to a second level cache controller that a read is cacheable in the second level cache (if present).

D[31:0]	IOCZ	Data bus. These are bi-directional signal paths used for data transfers between the processor and external memory. For read operations (when **nRW** is LOW), the input data must be valid before the falling edge of **MCLK**. For write operations (when **nRW** is HIGH), the output data will become valid while **MCLK** is LOW. At high clock frequencies the data may not become valid until just after the **MCLK** rising edge.
DBE	I	Data bus enable. When this input is LOW, the data bus, **D[31:0]** is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and **DBE** must be driven HIGH in systems which do not require the data bus for DMA or similar activities.
LOCK	OCZ	Locked operation. **LOCK** is driven HIGH, to signal a “locked” memory access sequence, and the memory manager should wait until **LOCK** goes LOW before allowing another device to access the memory. **LOCK** remains HIGH during the locked memory sequence. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
MCLK	I	This is a bus clock input. Bus cycles start and end with falling edges of **MCLK**. Hold **PCLK** HIGH to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further details. This signal is provided for backwards compatibility with previous processors, see **PCLK** for the preferred bus clock input.

MSE	I	Memory request/sequential enable. When this input is LOW, the **nMREQ** and **SEQ** outputs are put into a high impedance state (Note 1).
MAS[1:0]	OCZ	Memory Access Size. An output bus used by the processor to indicate the size of the next data transfer to the external memory system as being a byte, half word or full 32 bit word in length. MAS[1:0] is valid for both read and write operations. Normally it changes during phase 2 of the bus clock.The timing can be changed using **APE**.
nBLS[3:0]	OCZ	Not Byte Lane Selects. These signify which bytes of the memory are being accessed. For a word access all will be LOW. Normally they change during phase 2 of the bus clock. The timing can be changed using **APE**.
nFIQ	I	Not fast interrupt request. If FIQs are enabled, the processor will respond to a LOW level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input to guarantee that the interrupt has been taken.,
nIRQ	I	Not interrupt request. As **nFIQ**, but with lower priority. If IRQs are enabled, the processor will respond to a low level on this signal by taking the IRQ interrupt exception.

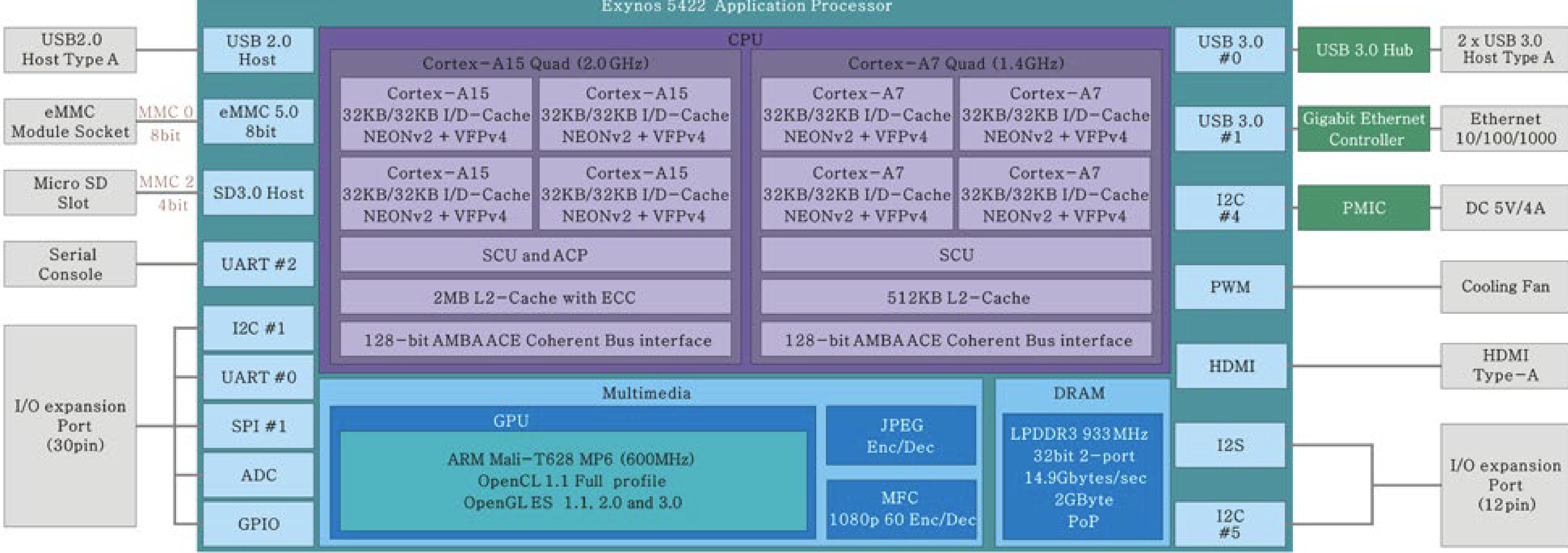
nMREQ	OCZ	Not memory request. A pipelined signal that changes while **MCLK** is LOW to indicate whether or not in the following cycle, the processor will be accessing external memory. When **nMREQ** is LOW, the processor will be accessing external memory in the next bus cycle.
nRESET	I	Not reset. This is a level sensitive input which is used to start the processor from a known address. A LOW level will cause the current instruction to terminate abnormally, and the on-chip cache, MMU, and write buffer to be disabled. When **nRESET** is driven HIGH, the processor will re-start from address 0. **nRESET** must remain LOW for at least 5 full fast clock cycles or 5 full bus clock cycles whichever is greater. While **nRESET** is LOW the processor will perform idle cycles and **nWAIT** must be HIGH.
nRW	OCZ	Not read/write. When HIGH this signal indicates a processor write operation; when LOW, a read. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
nTRST	I	Test interface reset. Note this signal does NOT have an internal pullup resistor. This signal must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (**nRESET**).
nWAIT	I	Not wait. When LOW this allows extra **MCLK** cycles to be inserted in memory accesses. It must change during the LOW phase of the **MCLK** cycle to be extended.

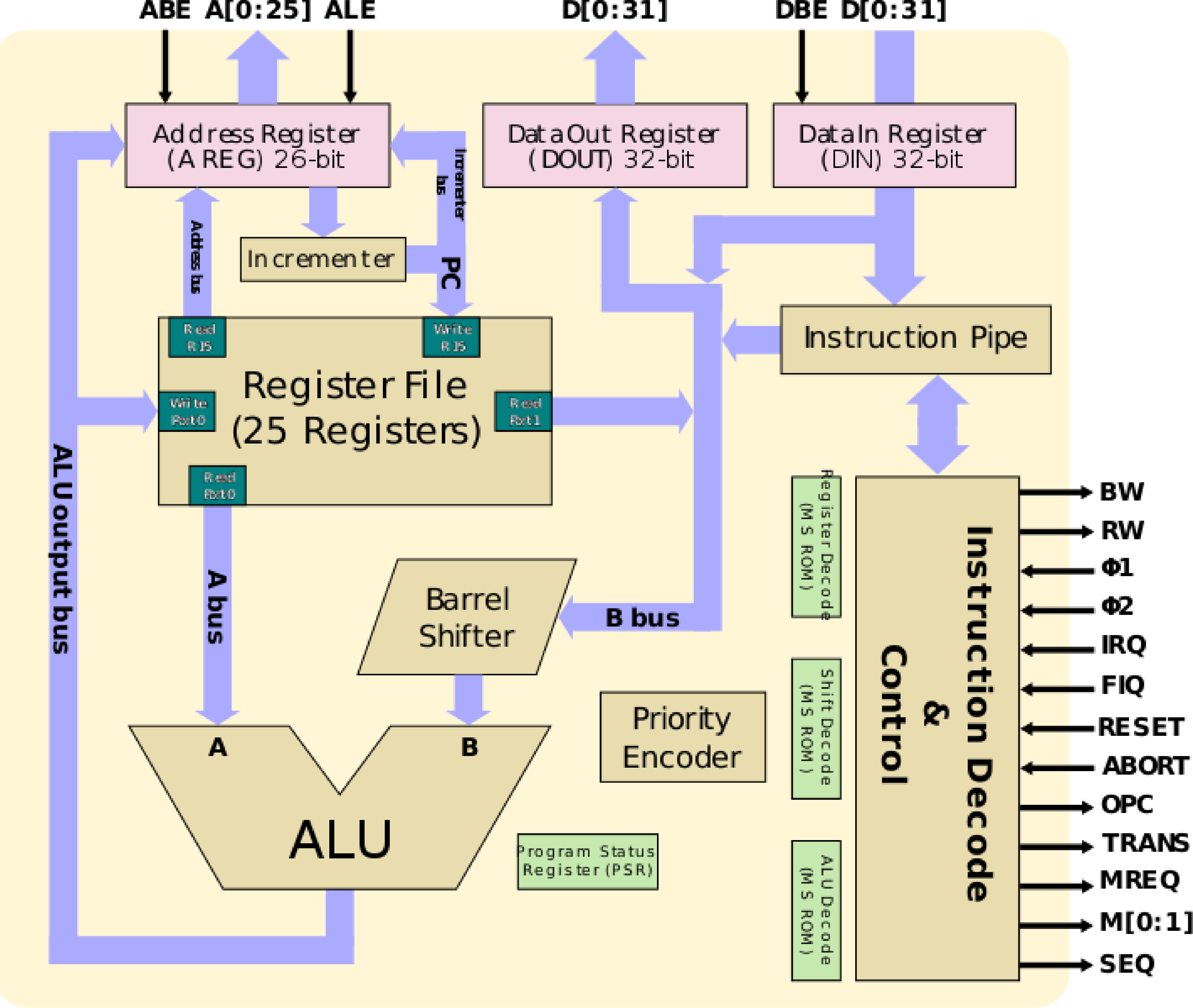
PCLK	I	This is an inverted bus clock input. Bus cycles start and end with rising edges of **PCLK**. Hold **MCLK** LOW to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further information. We recommend using this bus clock input for compatibility with the new generations of synchronous memory systems (SSRAM, SDRAM) and future ARM microprocessors. The **MCLK** input is provided for compatibility with earlier ARM processors.
PLLCFG[6:0]	I	Phase locked loop configuration input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details.
PLLFILT1		Analog filter pin for PLL.
PLLFILT2		Analog filter fast start pin for PLL.
PLLRANGE	IOCZ	In normal operation, an input which selects the **PLL** output frequency range. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. This pin is also used as an output when the device is in some test modes. The output driver is guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
PLLSLEEP	I	When HIGH, this puts the **PLL** into low power sleep mode. Please refer to ***11.5 Low Power Idle and Sleep*** on page 11-10 for further details.
PLLVDD		VDD supply for analog components in PLL. 1 pin. Should be appropriately isolated from digital noise on supply.

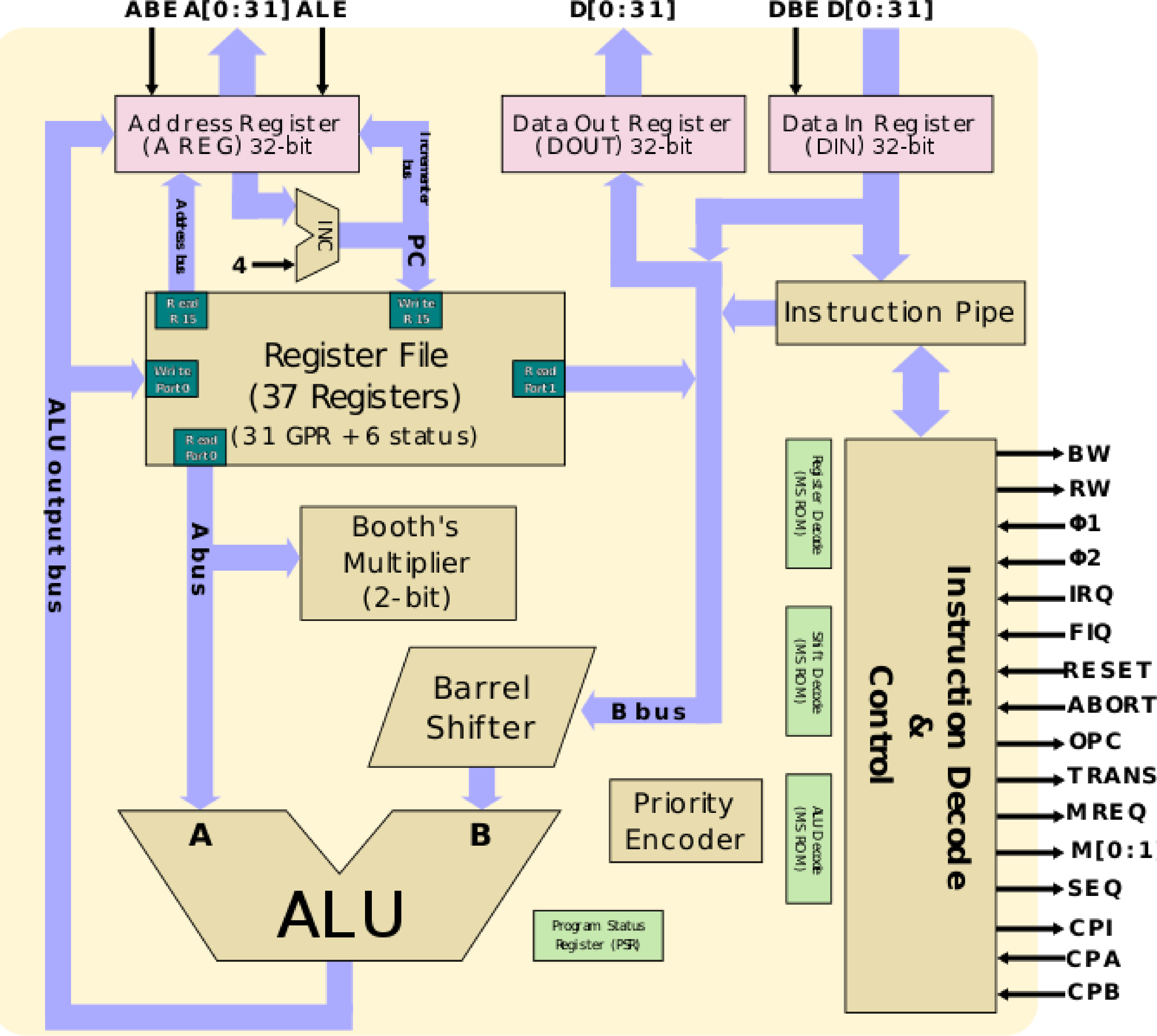
PLLVSS		Ground supply for analog components in PLL. 1 pin.
REFCLK	I	Clock input which is divided by the prescaler to provide the **PLL** reference clock. **REFCLK** can also be configured to a direct source of the internal fast clock, bypassing the **PLL**. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 and ***11.3.3 Fast clock direct (bypassing the PLL)*** on page 11-8 for further details.
REFCLKCFG[1:0]	IOCZ	In normal operation, an input which selects the divide ratio for the **PLL** reference clock prescaler on the **REFCLK** input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. These pins are also used as an output when the device is in some test modes. The output drivers are guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
SEQ	OCZ	Sequential address. This signal is the inverse of **nMREQ**, and is provided for compatibility with existing ARM memory systems.
TESTMODE	I	This signal must be tied LOW.
TESTOUT[4:0]	O	This bus should be left unconnected. These outputs will be driven LOW except when device test features are enabled. They will not be tri-stated, except via the JTAG test port.
TCK	I	Test interface reference Clock. This times all the transfers on the JTAG test interface.

TDI	I	Test interface data input. Note this signal does *not* have an internal pullup resistor.
TDO	OCZ	Test interface data output. Note this signal does *not* have an internal pullup resistor.
TMS	I	Test interface mode select. Note this signal does *not* have an internal pullup resistor.
VCC		Pad voltage reference. 1 pin is allocated to VCC. This should be tied to the system power supply, ie. 5V in a TLL system or 3.3V in a 3.3V system. See ***Appendix A, Use of the ARM810 in a 5V TTL System***.
VDD		Positive supply. 15 pins are allocated to **VDD** in the 144 TQFP package.
VSS		Ground supply. 15 pins are allocated to **VSS** in the 144 TQFP package.

ODROID-XU4 BLOCK DIAGRAM



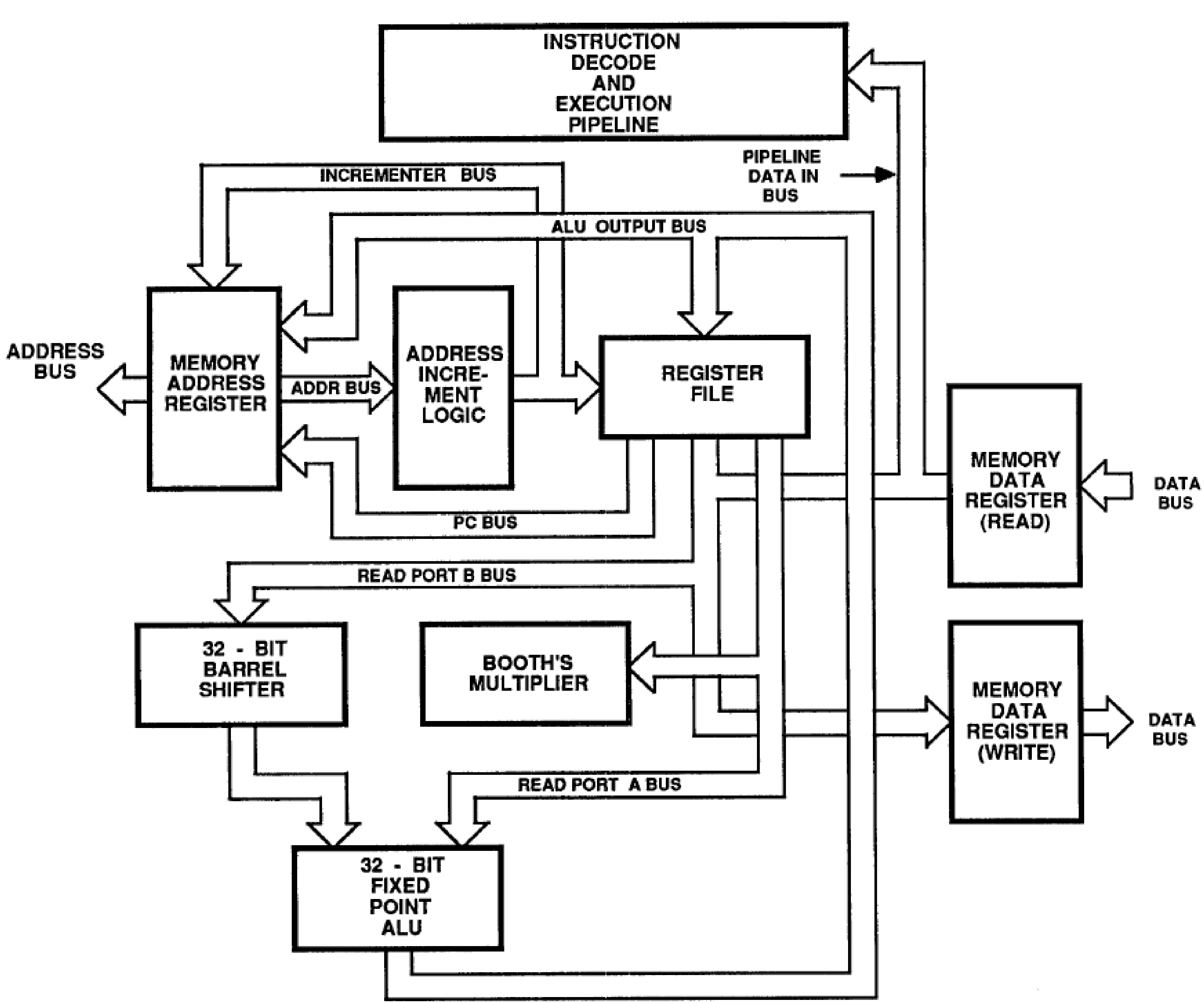

ABE
A[0:25]
ALE
D[0:31]
DBE
D[0:31]
Address Register
(A REG) 26-bit
Data Out Register
(DOUT) 32-bit
Data In Register
(DIN) 32-bit
Incrementer
Incrementer bus
PC
Address bus
Read R15
Write R15
Register File
(25 Registers)
Write Port0
Read Port1
Read Port0
Instruction Pipe
ALU output bus
A bus
Barrel Shifter
B bus
Register Decode (MS ROM)
Shift Decode (MS ROM)
ALU Decode (MS ROM)
Instruction Decode & Control
Priority Encoder
A
B
ALU
Program Status Register (PSR)
BW
RW
Φ1
Φ2
IRQ
FIQ
RESET
ABORT
OPC
TRANS
MREQ
M[0:1]
SEQ

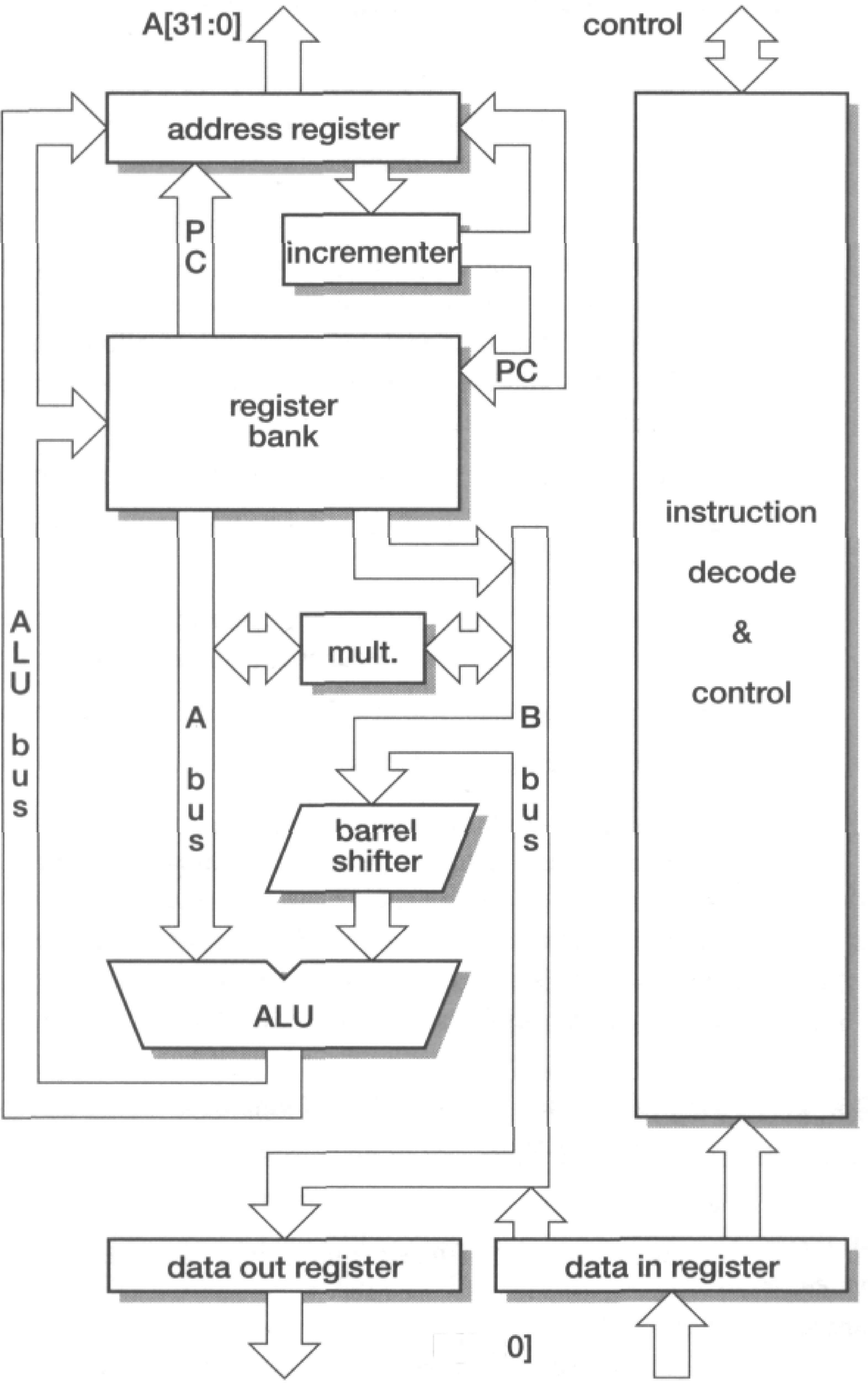


ABE A[0:31] ALE
D[0:31]
DBE D[0:31]
Address Register
(A REG) 32-bit
Data Out Register
(DOUT) 32-bit
Data In Register
(DIN) 32-bit
Incrementer bus
Address bus
INC
4
PC
Instruction Pipe
Read R15
Write R15
Register File
(37 Registers)
(31 GPR + 6 status)
Write Port 0
Read Port 1
Read Port 0
ALU output bus
A bus
Booth's
Multiplier
(2-bit)
Barrel
Shifter
B bus
Priority
Encoder
A
B
ALU
Program Status Register (PSR)
Register Decode (MS ROM)
Shift Decode (MS ROM)
ALU Decode (MS ROM)
Instruction Decode
&
Control
BW
RW
Φ1
Φ2
IRQ
FIQ
RESET
ABORT
OPC
TRANS
MREQ
M[0:1]
SEQ
CPI
CPA
CPB

Instruction Fetch	Instruction Decode	Register Select	Register Read	Shift	ALU	Register Write
Fetch	**Decode**		**Execute**			

BLOCK DIAGRAM





A[31:0]
control
address register
P
C
incrementer
PC
register
bank
instruction
decode
&
control
A
L
U
b
u
s
mult.
A
b
u
s
B
b
u
s
barrel
shifter
ALU
data out register
data in register
0]

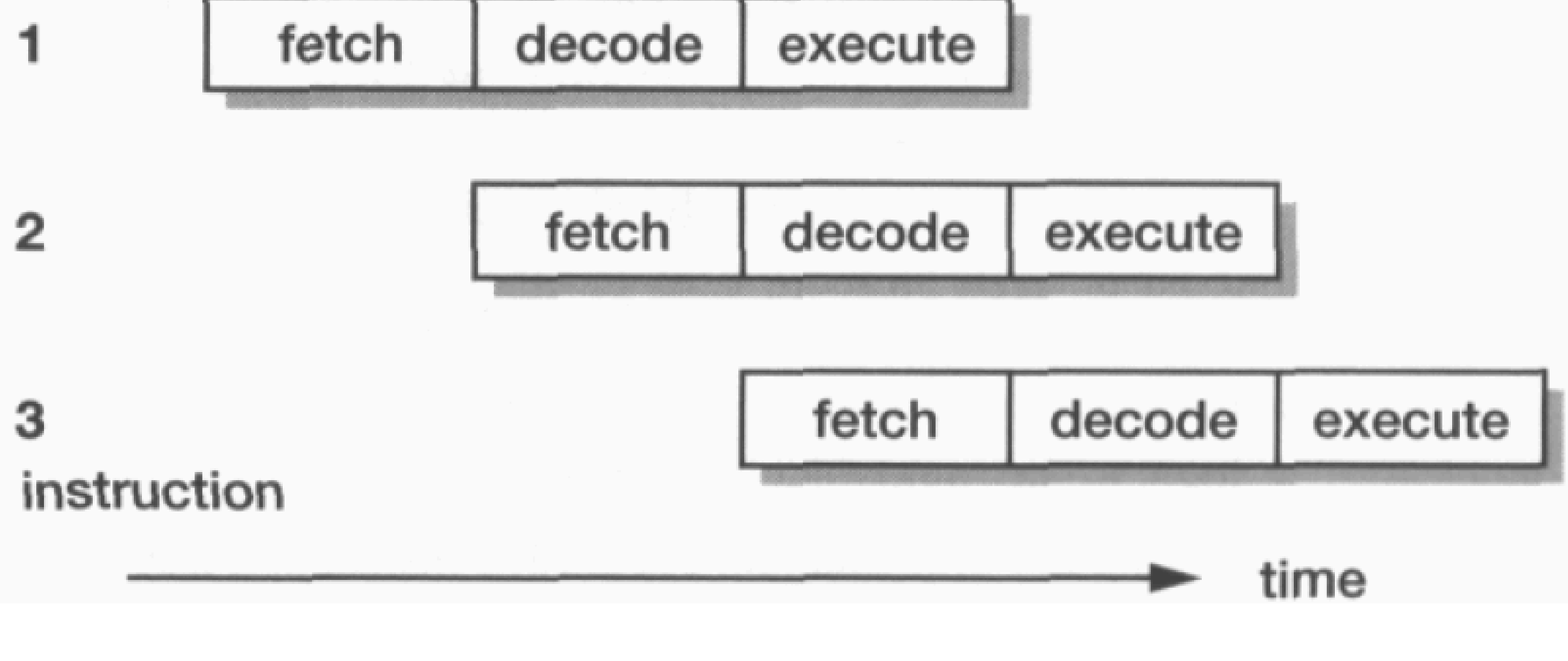


Figure 4.2 ARM single-cycle instruction 3-stage pipeline operation.

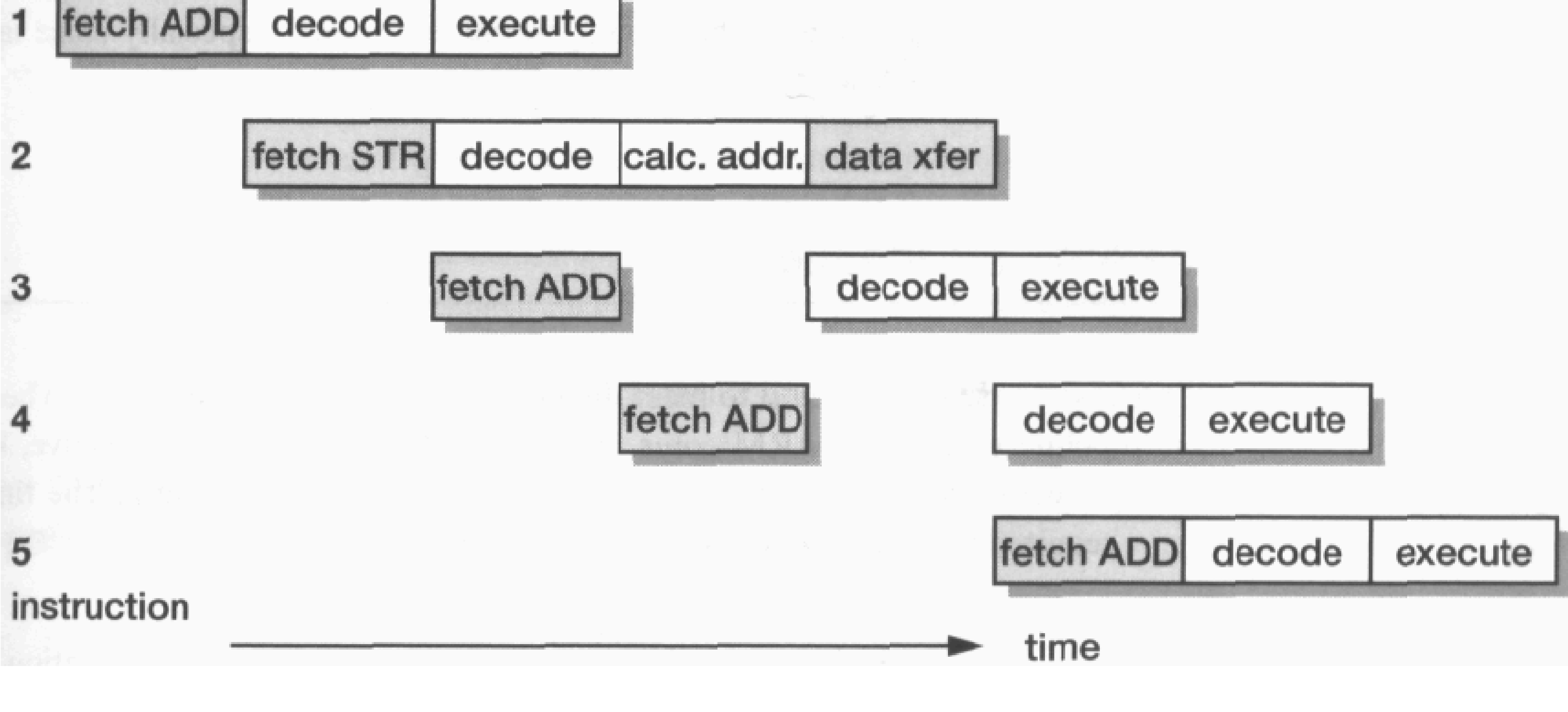


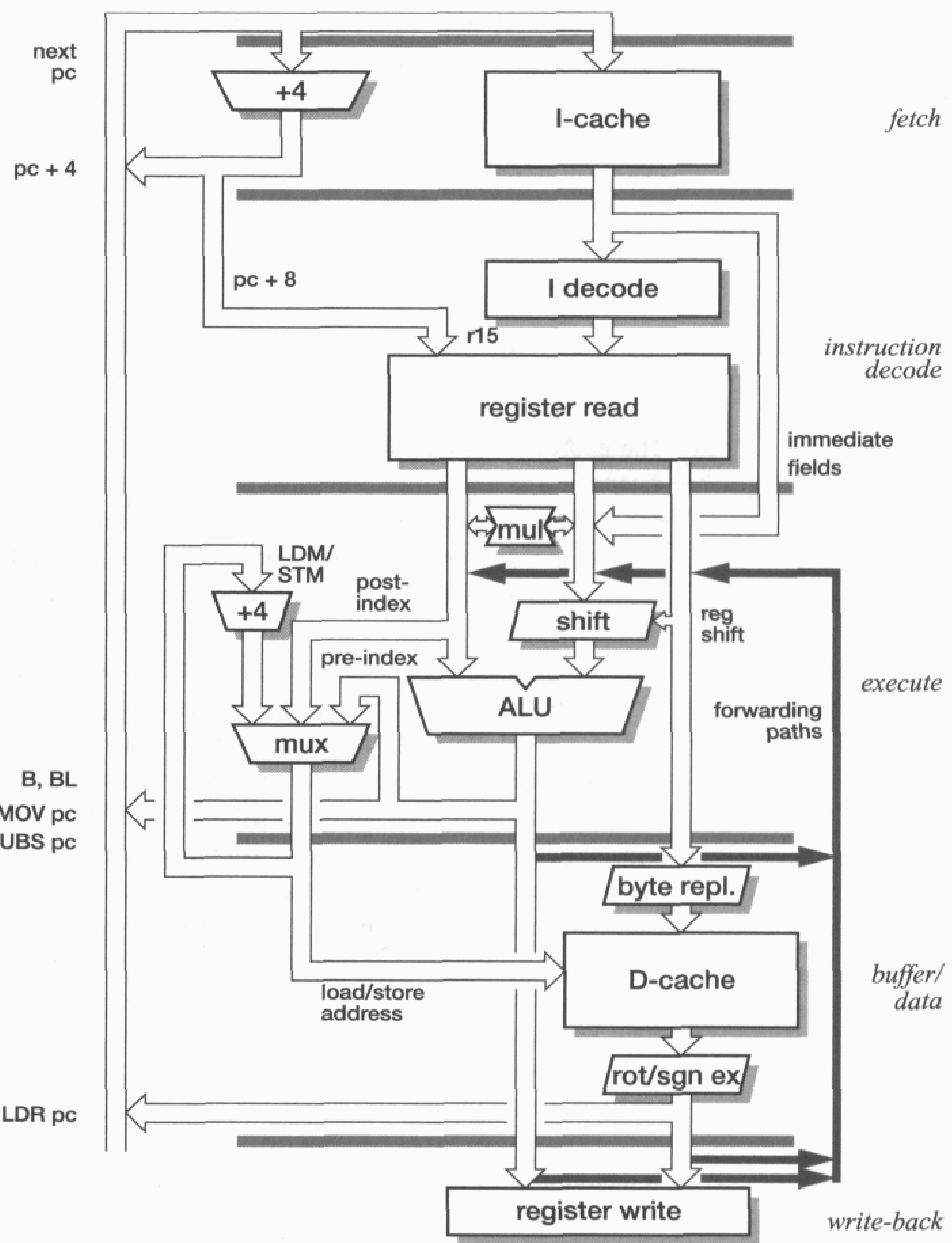
Figure 4.3 ARM multi-cycle instruction 3-stage pipeline operation.

PC behaviour

One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r!5, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

This is, indeed, what happens, and the programmer who attempts to access the PC directly through r!5 must take account of the exposure of the pipeline here. However, for most normal purposes the assembler or compiler handles all the details.

Even more complex behaviour is exposed if r!5 is used later than the first cycle of an instruction, since the instruction will itself have incremented the PC during its first cycle. Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided, especially since later ARMs do not have the same behaviour in these cases.



next pc
+4
I-cache
fetch
pc + 4
pc + 8
I decode
r15
instruction decode
register read
immediate fields
mul
LDM/ STM
post-index
+4
shift
reg shift
pre-index
ALU
execute
forwarding paths
mux
B, BL
MOV pc
SUBS pc
byte repl.
D-cache
buffer/ data
load/store address
rot/sgn ex
LDR pc
register write
write-back

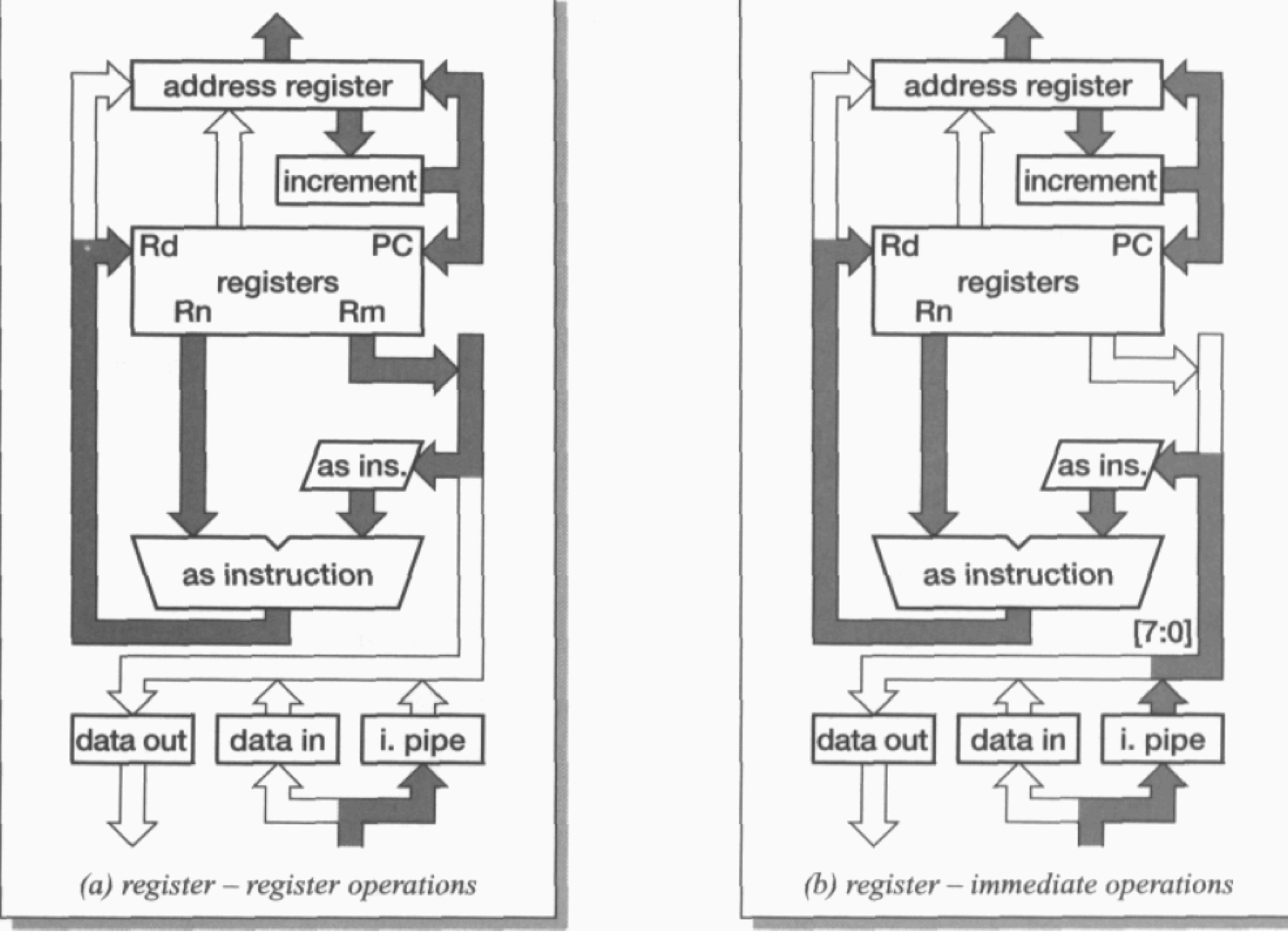


Figure 4.5 Data processing instruction datapath activity.

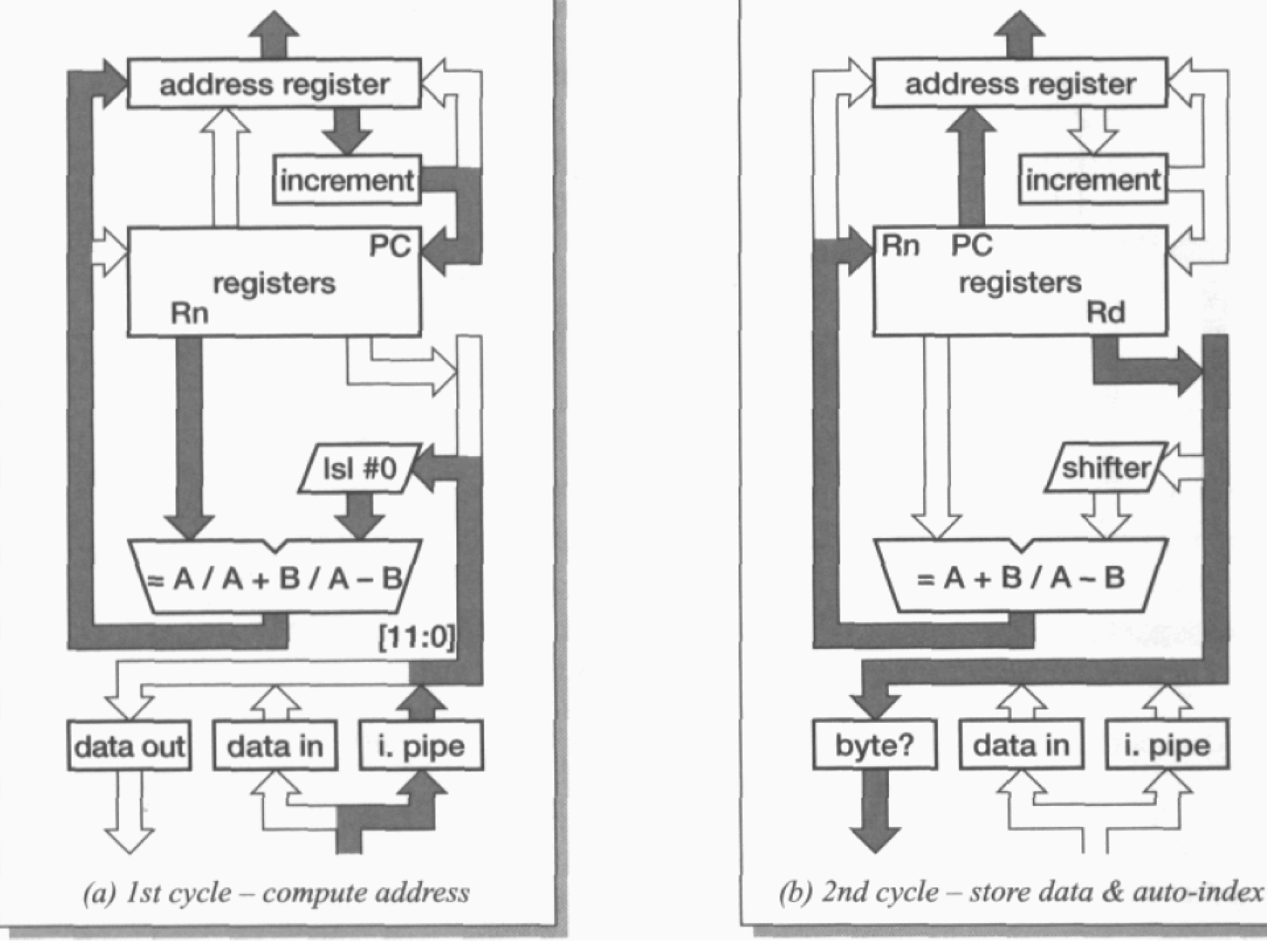


Figure 4.6 SIR (store register) datapath activity.

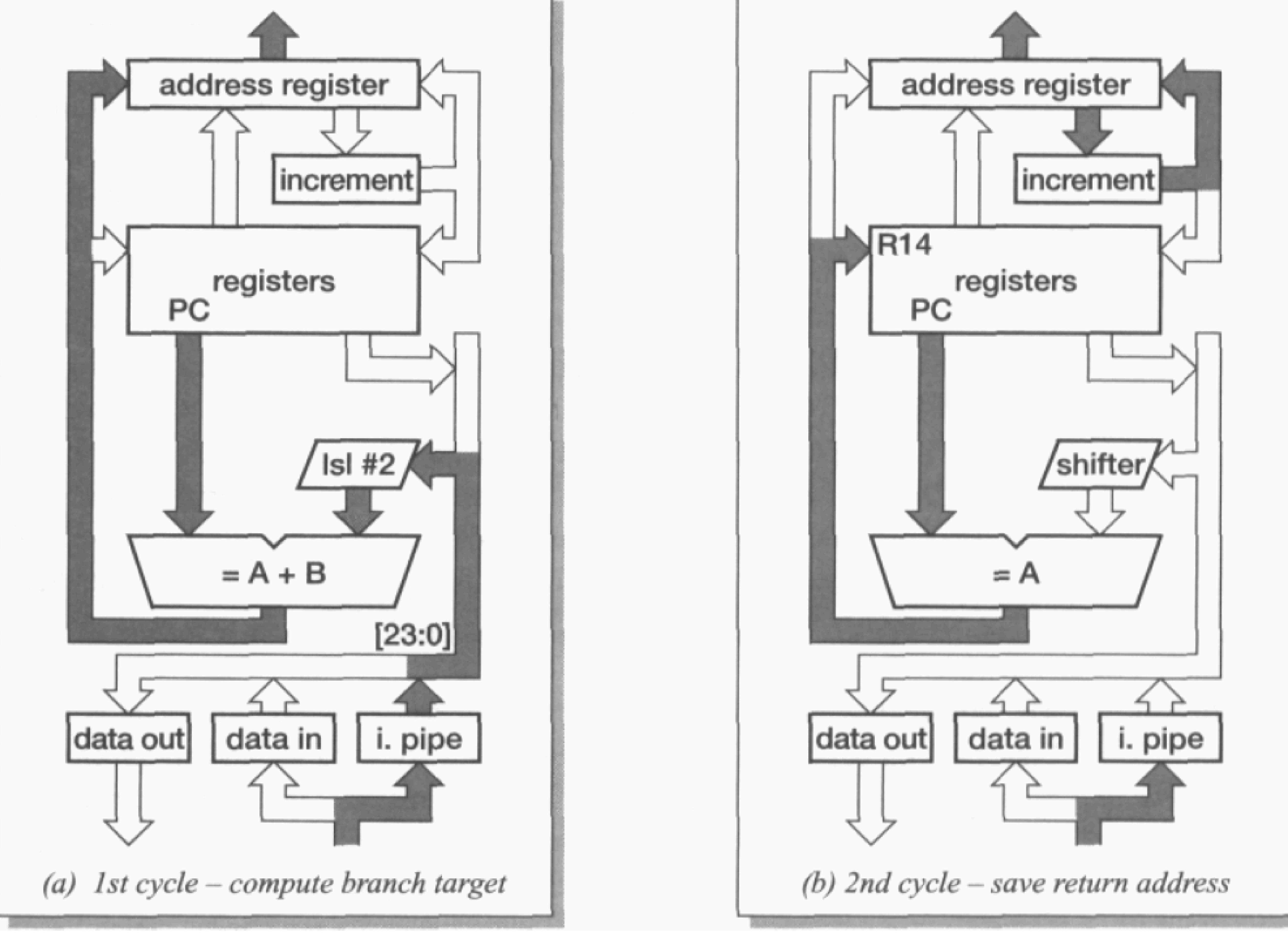


Figure 4.7 The first two (of three) cycles of a branch instruction.

Figure 1-2 shows:

- the two Fetch stages
- a Decode stage
- an Issue stage
- the four stages of the ARM1176JZF-S integer execution pipeline.

These eight stages make up the processor pipeline.

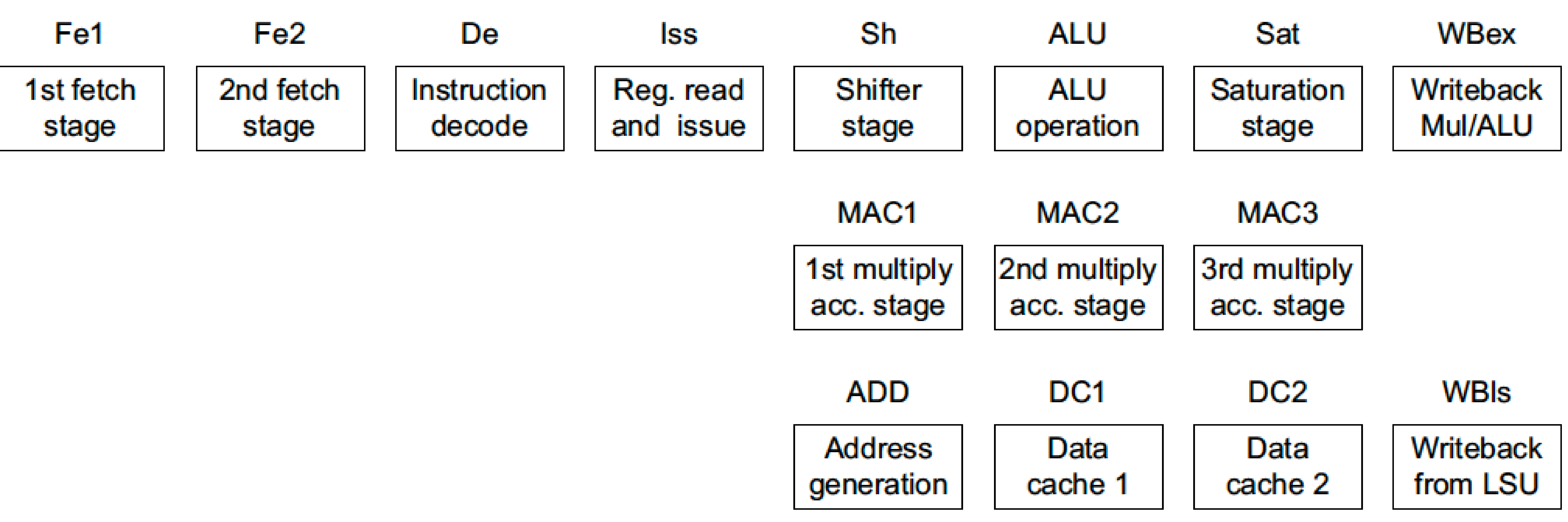


Figure 1-2 ARM1176JZF-S pipeline stages

Fe1 First stage of instruction fetch where address is issued to memory and data returns from memory

Fe2 Second stage of instruction fetch and branch prediction.

De Instruction decode.

Iss Register read and instruction issue.

Sh Shifter stage.

ALU Main integer operation calculation.

Sat Pipeline stage to enable saturation of integer results.

WBex Write back of data from the multiply or main execution pipelines.

MAC1 First stage of the multiply-accumulate pipeline.

MAC2 Second stage of the multiply-accumulate pipeline.

MAC3 Third stage of the multiply-accumulate pipeline.

ADD Address generation stage.

DC1 First stage of data cache access.

DC2 Second stage of data cache access.

WBls Write back of data from the Load Store Unit.

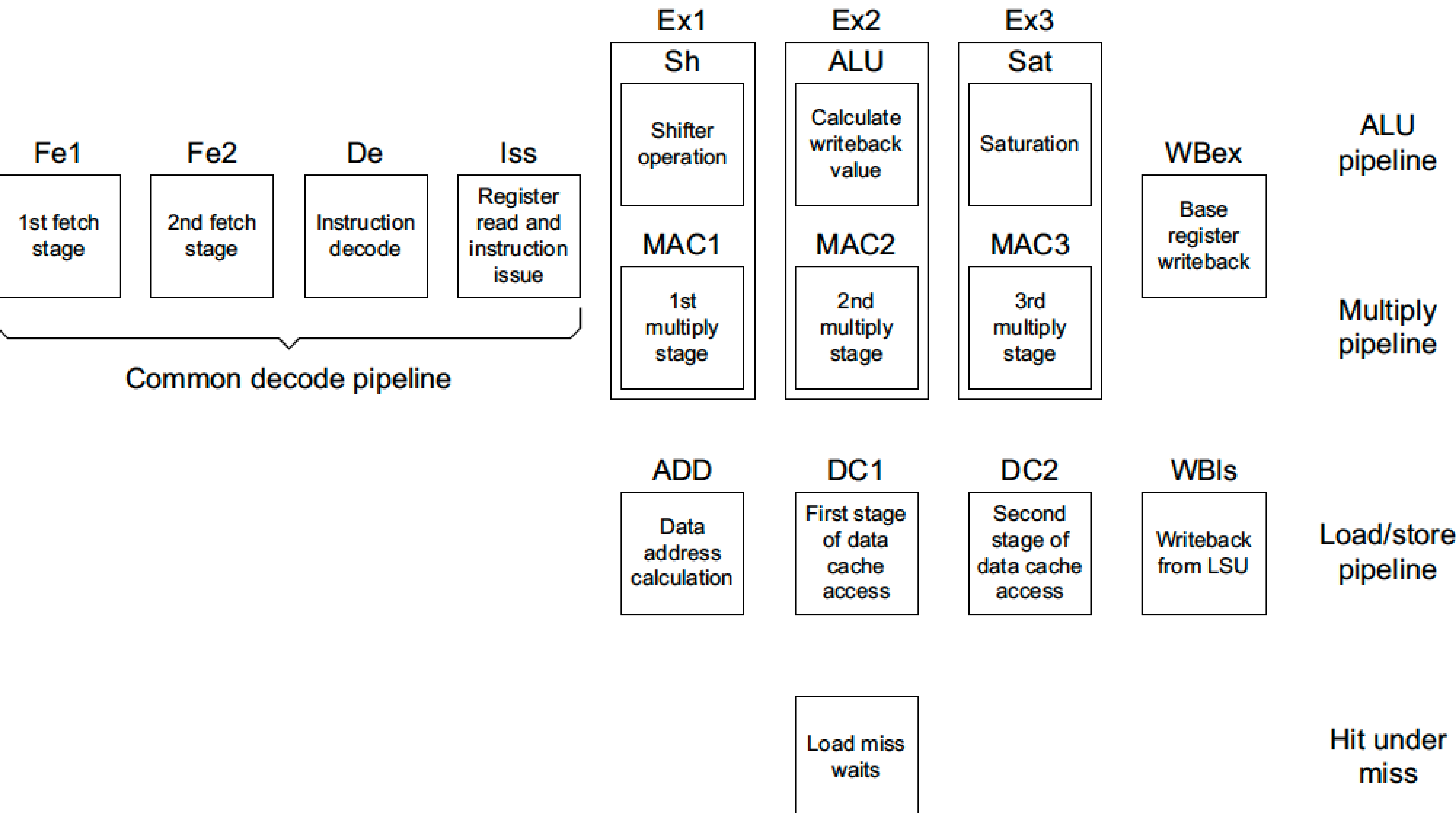


Figure 1-3 Typical operations in pipeline stages

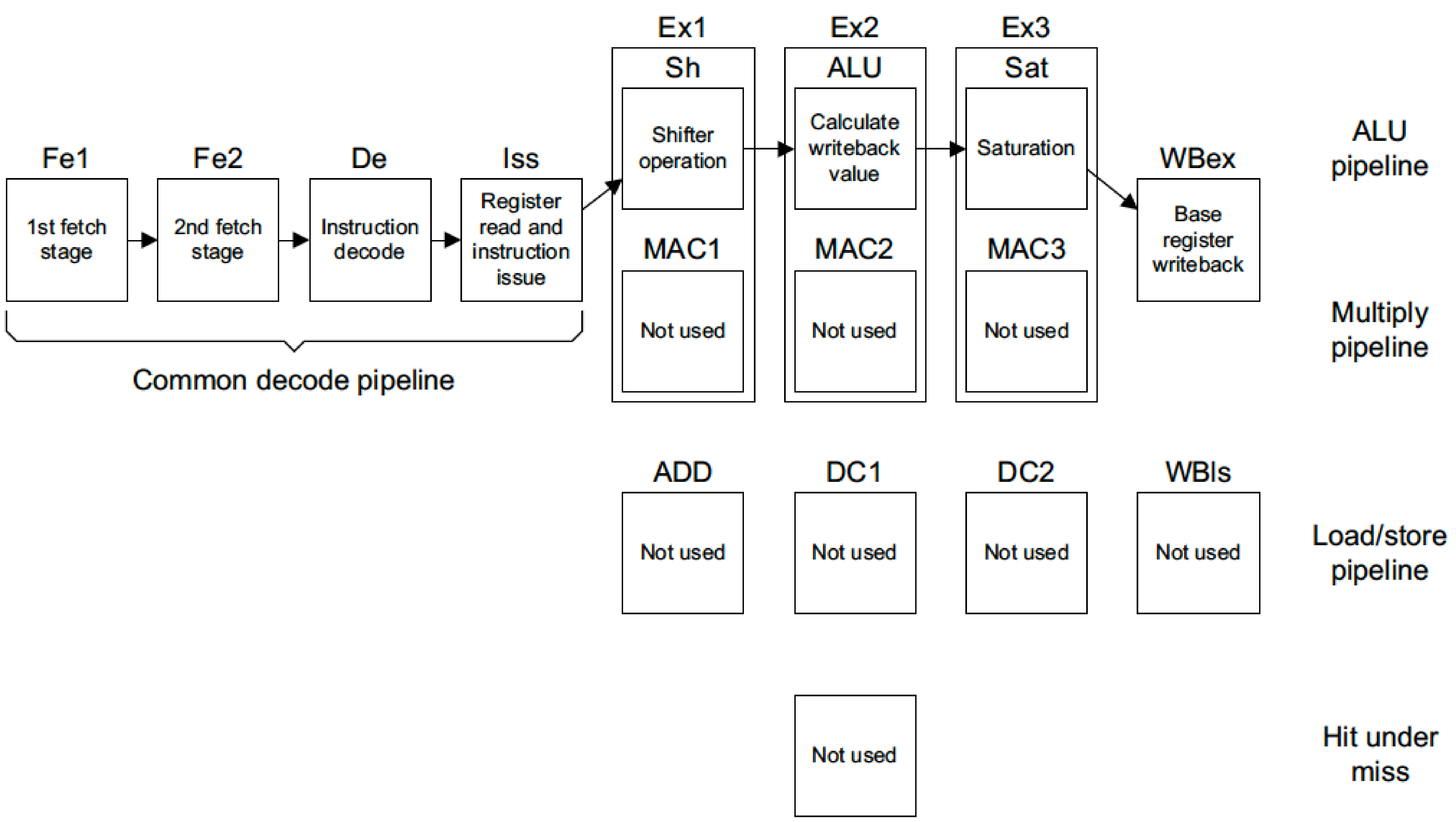


Figure 1-4 Typical ALU operation

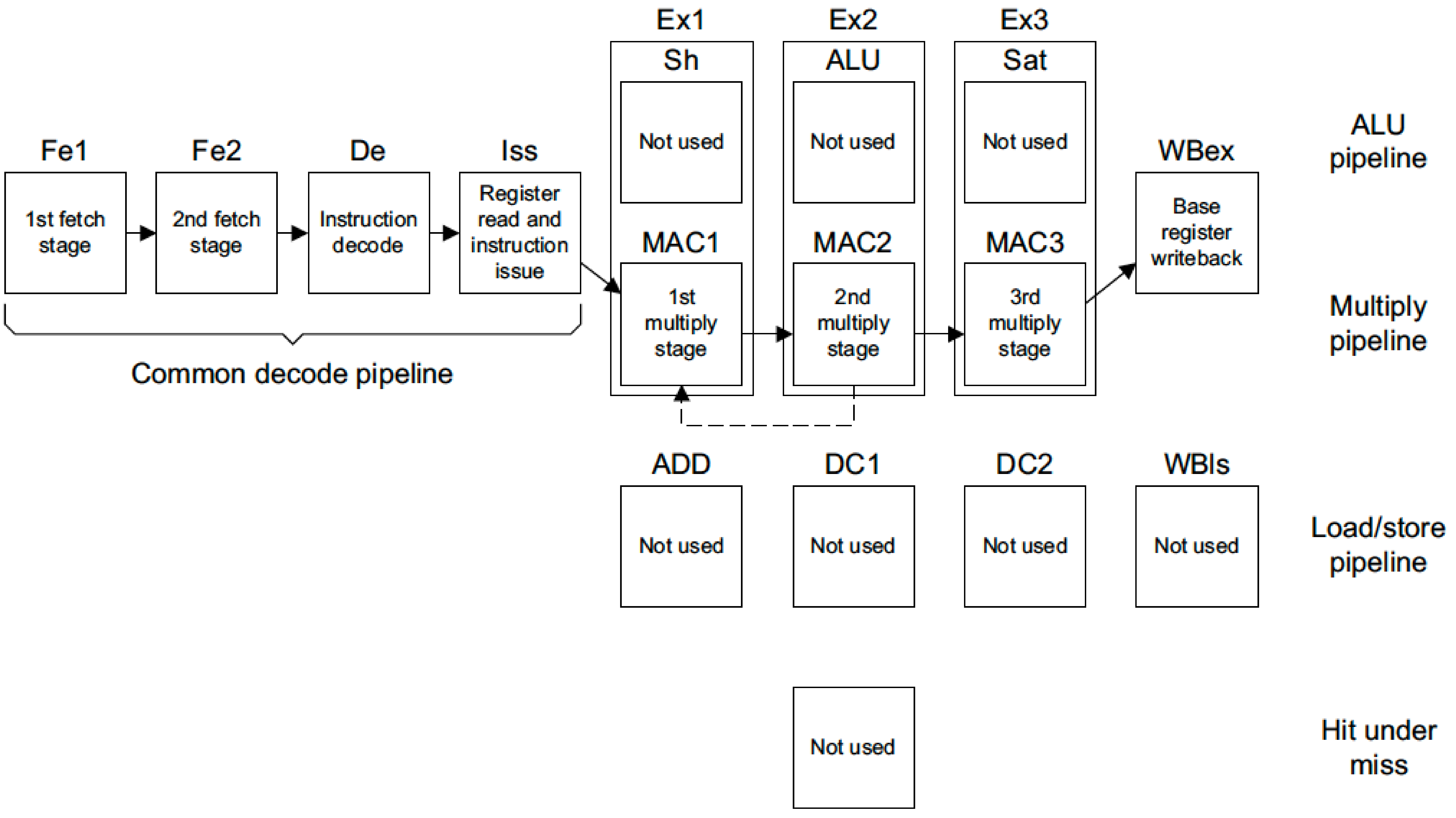


Figure 1-5 Typical multiply operation

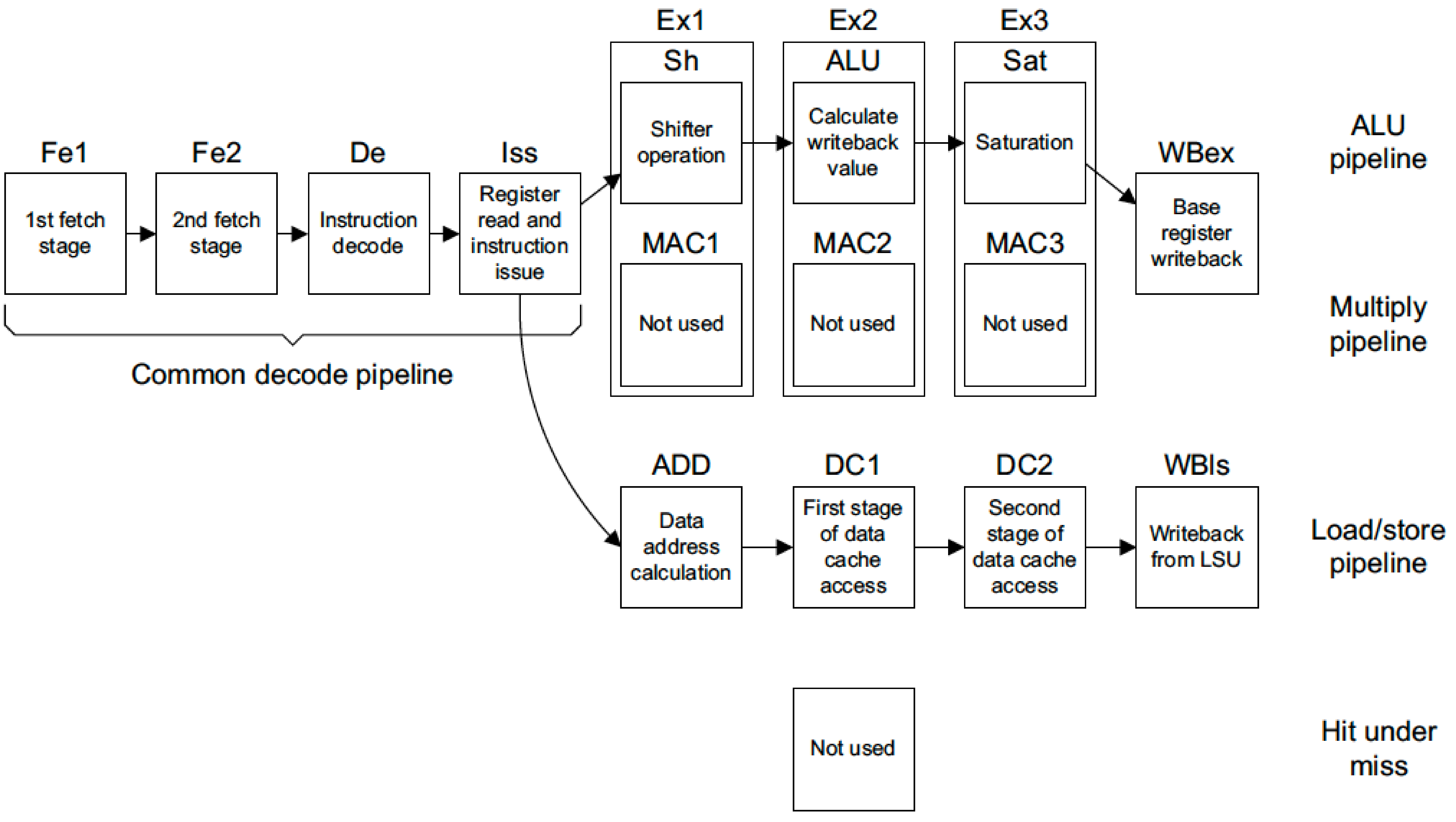


Figure 1-6 Progression of an LDR/STR operation

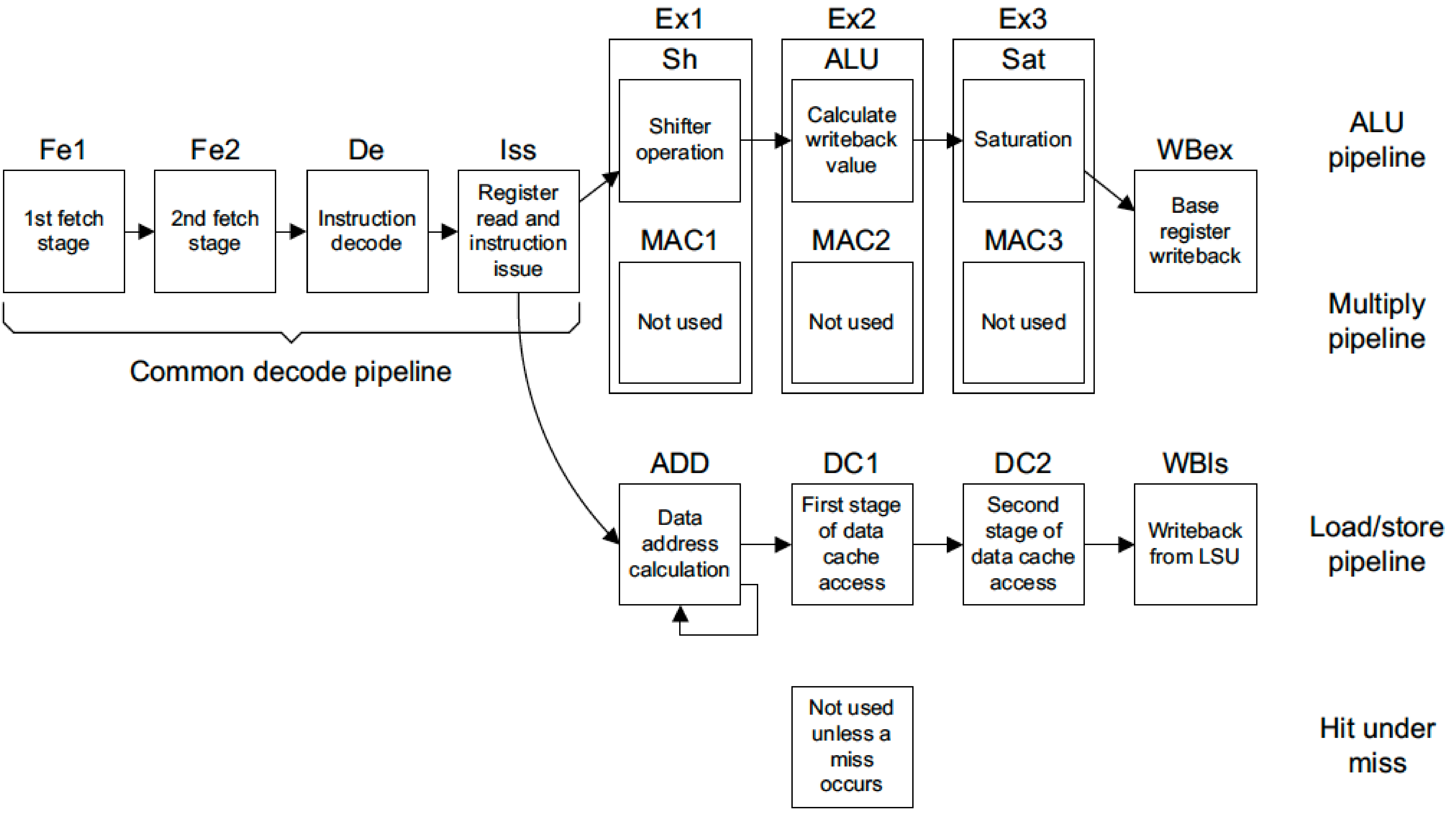


Figure 1-7 Progression of an LDM/STM operation

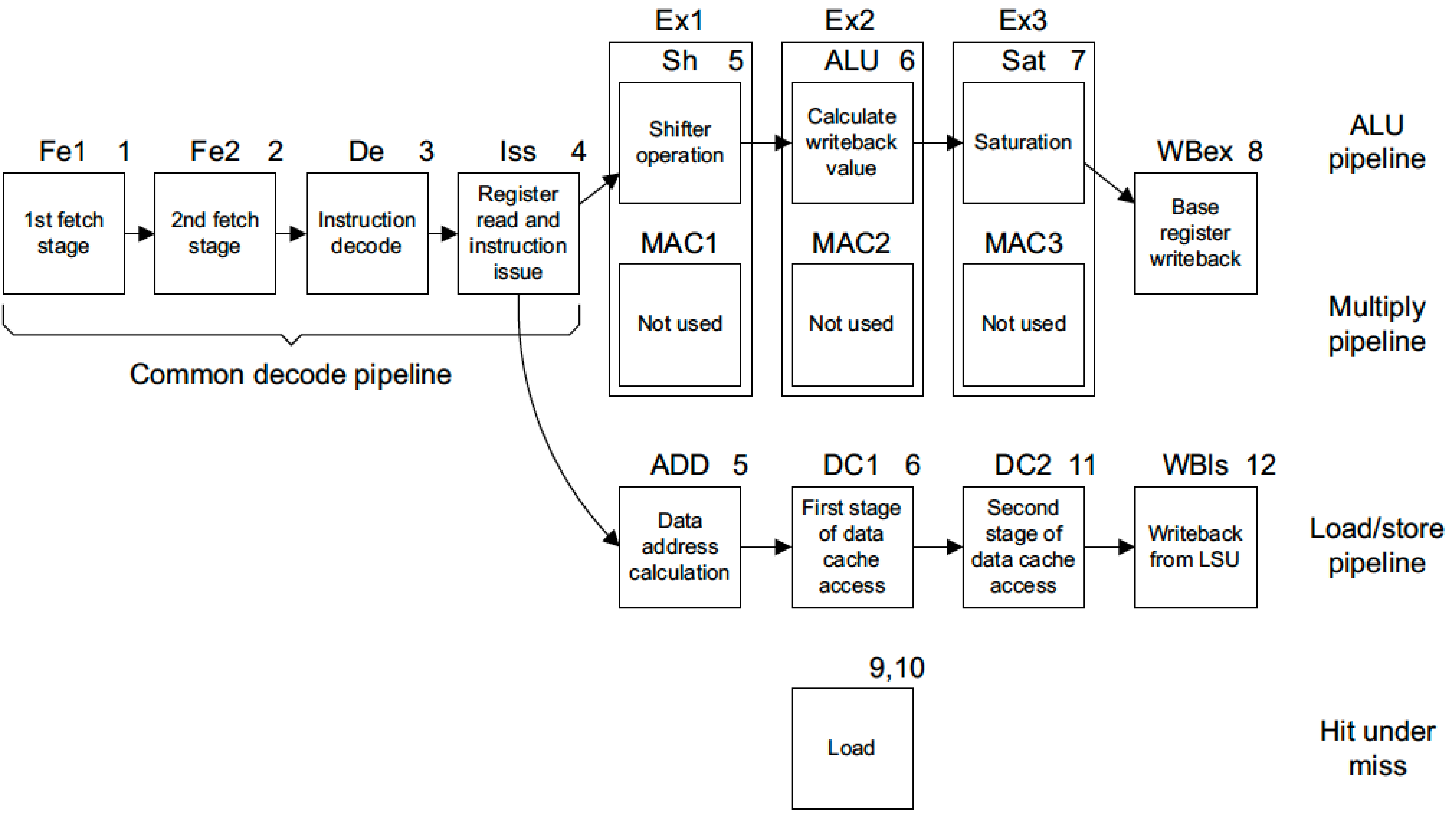


Figure 1-8 Progression of an LDR that misses

The VFP11 coprocessor has three separate instruction pipelines:

- the *Multiply and Accumulate* (FMAC) pipeline
- the *Divide and Square root* (DS) pipeline
- the *Load/Store* (LS) pipeline.

Each pipeline can operate independently of the other pipelines and in parallel with them. Each of the three pipelines shares the first two pipeline stages, Decode and Issue. These two stages and the first cycle of the Execute stage of each pipeline remain in lockstep with the ARM11 pipeline stage but effectively one cycle behind the ARM11 pipeline. When the ARM11 processor is in the Issue stage for a particular VFP instruction, the VFP11 coprocessor is in the Decode stage for the same instruction. This lockstep mechanism maintains in-order issue of instructions between the ARM11 processor and the VFP11 coprocessor.

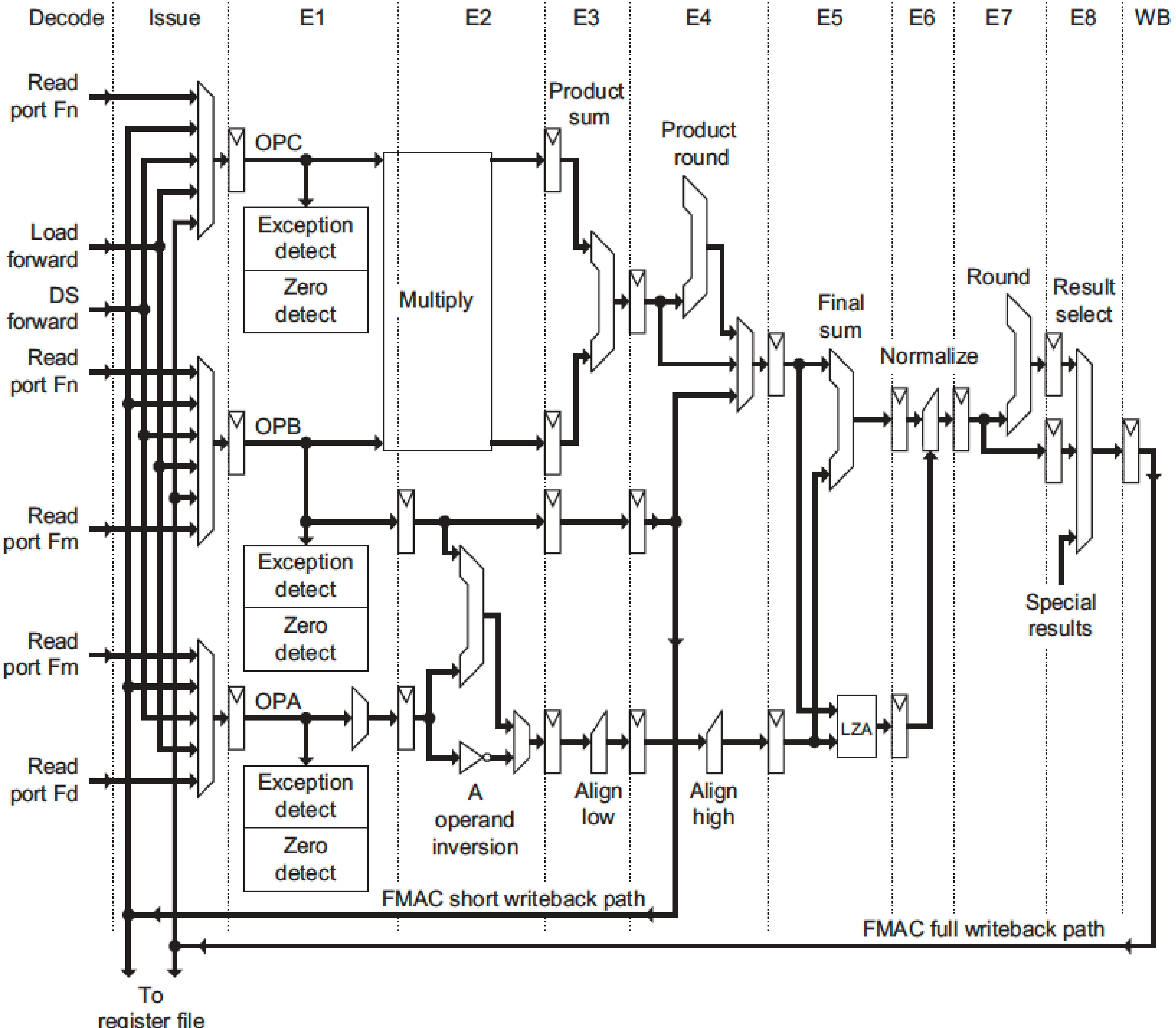


Figure 1-1 FMAC pipeline

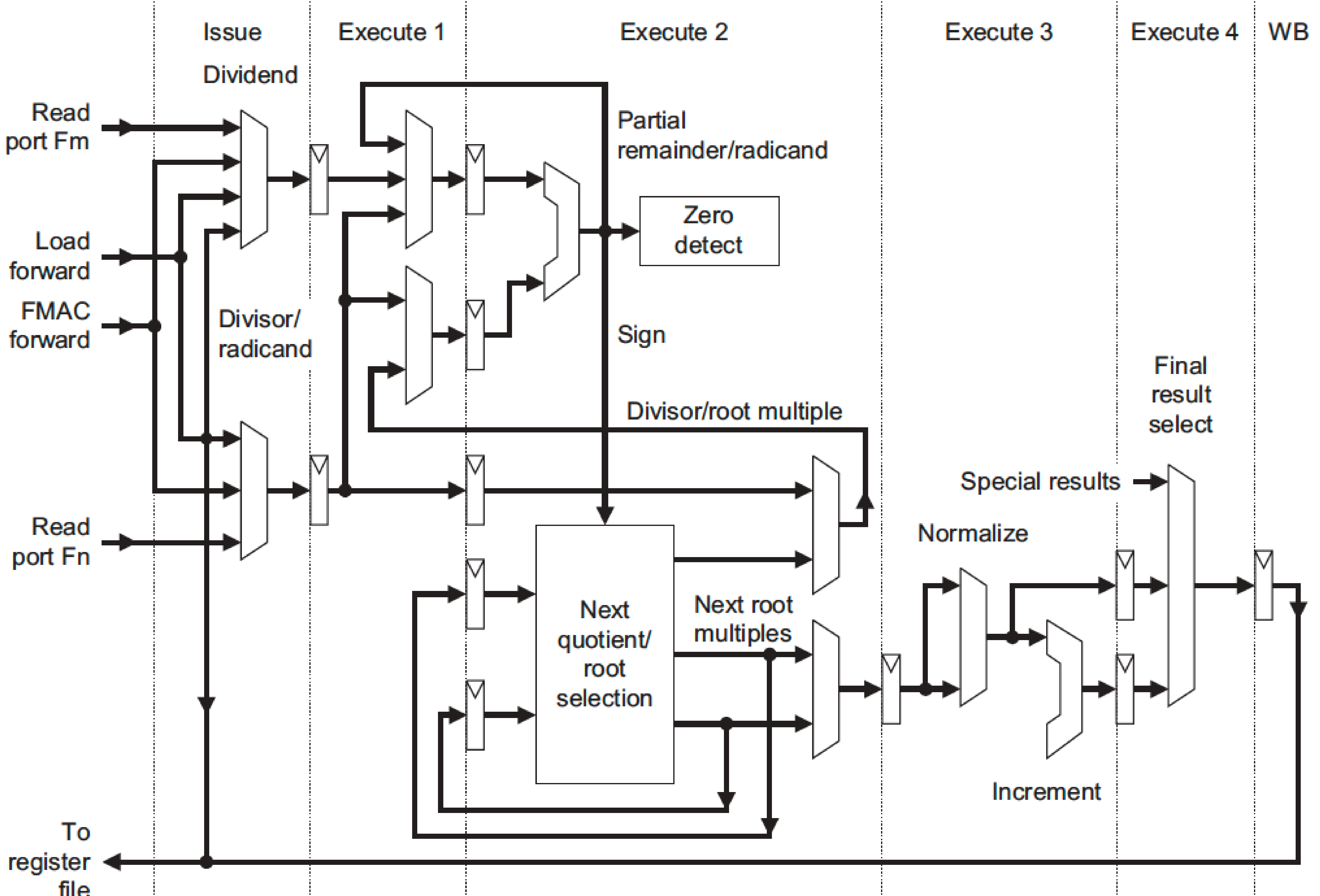


Figure 1-2 DS pipeline

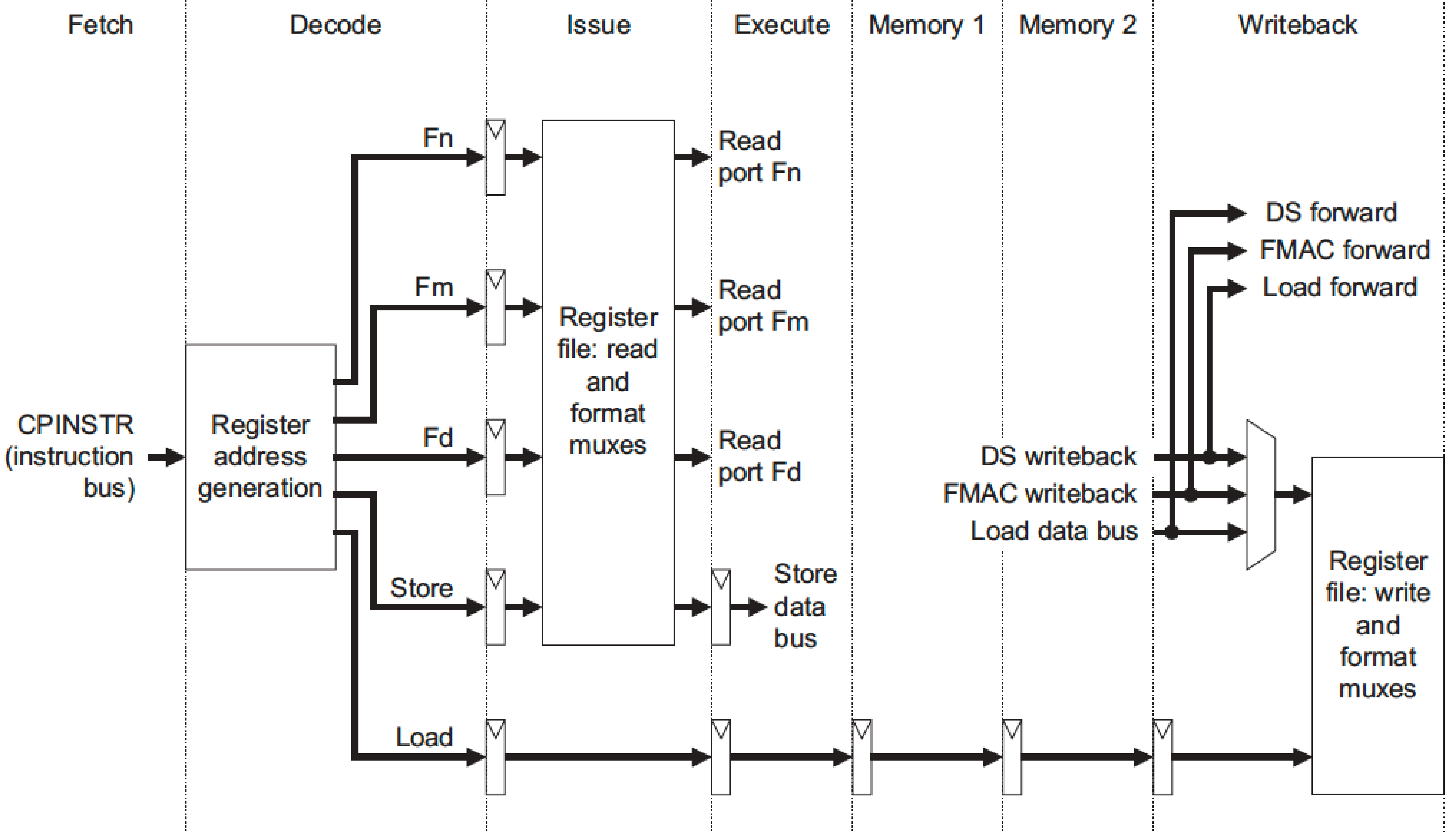


Figure 1-3 LS pipeline

Knights Landing Processor Architecture

MCDRAM
x36
PCIe3
MCDRAM

DDR4
DDR4

MCDRAM
MCDRAM

TILE

2VPU
HUB
2VPU
1MB
L2
Core
Core

Up to 72 Intel Architecture cores based on Silvermont (Intel® Atom processor)

- Four threads/core
- Two 512b vector units/core
- Up to 3x single thread performance improvement over KNC generation

Full Intel® Xeon processor ISA compatibility through AVX-512 (except TSX)

6 channels of DDR4 2400 MHz -up to 384GB

36 lanes PCI Express* Gen 3

8/16GB of high-bandwidth on-package MCDRAM memory >500GB/sec

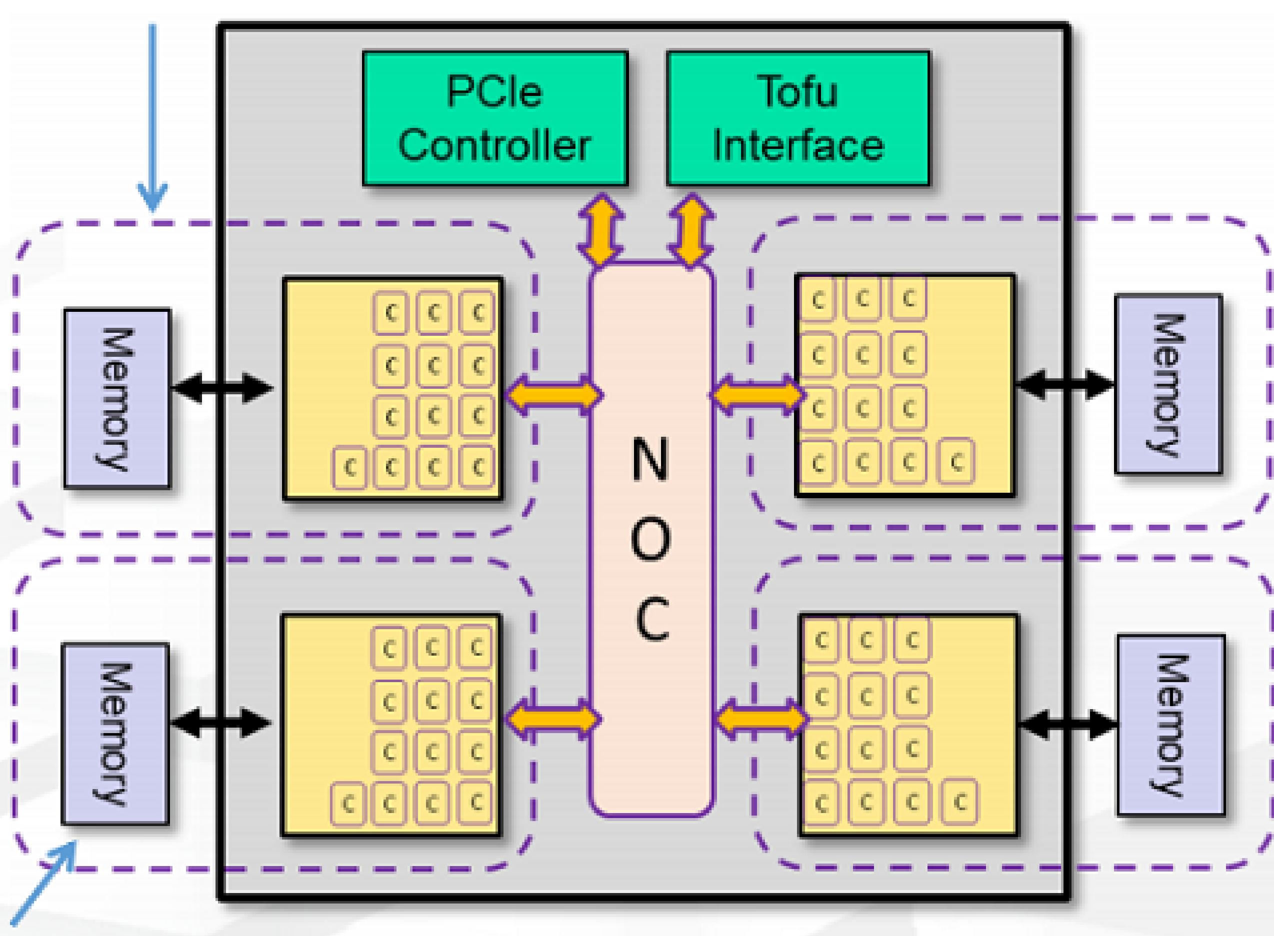
200W TDP

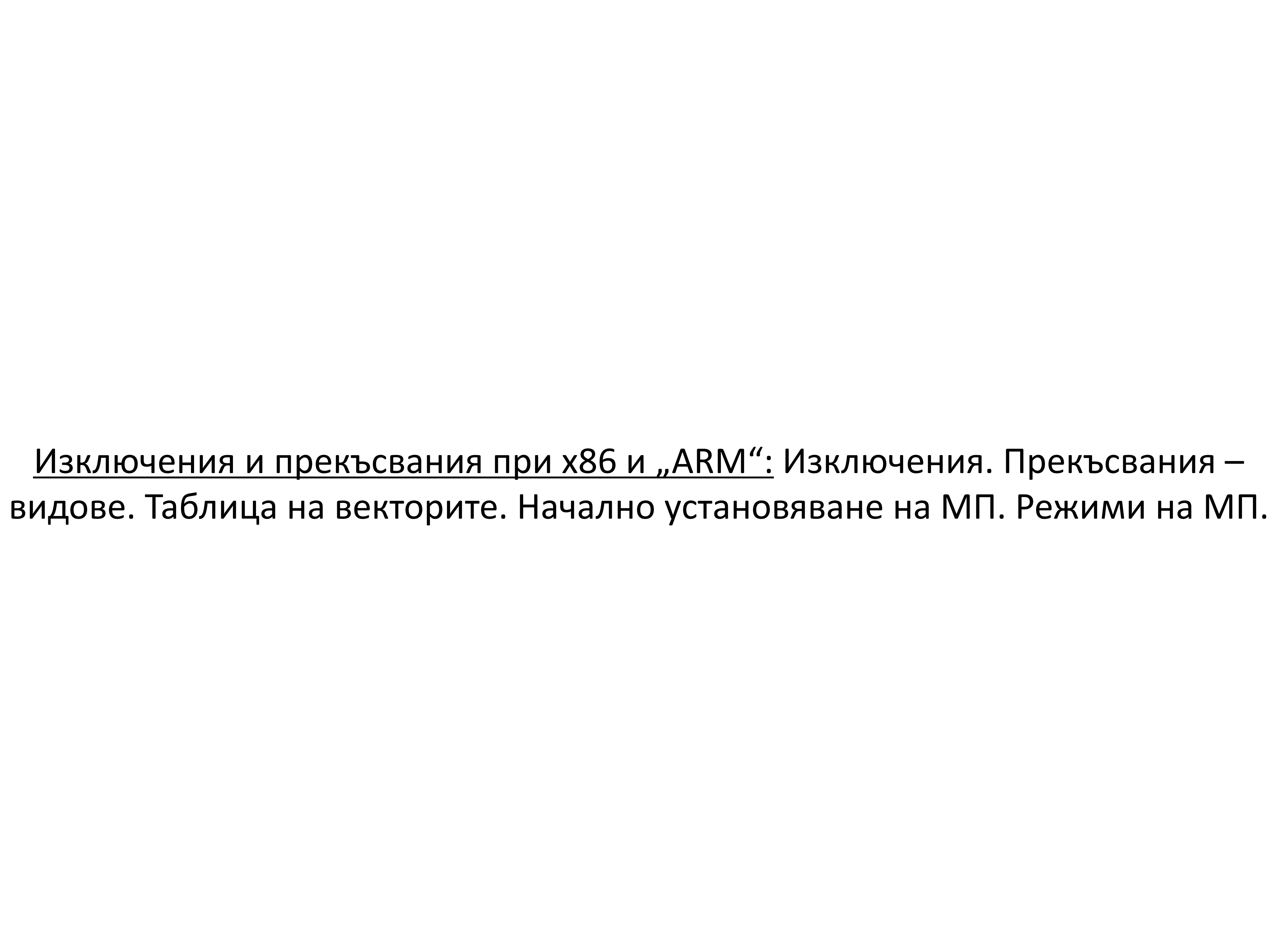
CPU Architecture: A64FX

- **Armv8.2-A (AArch64 only) + SVE (Scalable Vector Extension)**
  - FP64/FP32/FP16 (https://developer.arm.com/products/architecture/a-profile/docs)
- **SVE 512-bit wide SIMD**
- **# of Cores: 48 + (2/4 for OS)**
- Co-design with application developers and high memory bandwidth utilizing on-package stacked memory: **HBM2(32GiB)**
- Leading-edge Si-technology (7nm FinFET), low power logic design **(approx. 15 GF/W (dgemm))**, and power-controlling knobs
- PCIe Gen3 16 lanes
- Peak performance
  - > 2.7 TFLOPS (>90% @ dgemm)
  - Memory B/W 1024GB/s (>80% stream)
  - Byte per Flops: approx. 0.4

◆ "Common" programing model will be to run each MPI process on a NUMA node (CMG) with OpenMP-MPI hybrid programming.

◆ 48 threads OpenMP is also supported.




Изключения и прекъсвания при x86 и „ARM“: Изключения. Прекъсвания – видове. Таблица на векторите. Начално установяване на МП. Режими на МП.

5.1. INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are forced transfers of execution to a procedure or task. The procedure or task is called a *handler*. Interrupts typically occur at random times during the execution of a program, in response to signals from hardware. They are used to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT *n* instruction. Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults.

The processor's interrupt and exception handling mechanism allows interrupts and exceptions to be handled transparently to application programs and the operating system or executive. When an interrupt is received or an exception is detected, the currently running procedure or task is automatically suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

The processor receives interrupts from three sources and exceptions from two sources:

- Interrupts
  - **Non-maskable interrupts (NMIs).** These interrupts are received on the processor's NMI# input pin. The processor does not provide a mechanism to prevent non-maskable interrupts.
  - **Maskable interrupts.** These interrupts are received either at the processor's INTR# (interrupt) pin from an external, system-based interrupt controller (8259A) or as a serial message on the LINT[1:0] pins from a system-based I/O APIC. The processor does not act on maskable interrupts unless the IF (interrupt-enable) flag in the EFLAGS register is set.
  - **Software-generated interrupts.** These are generated by INT *n* instruction. The processor does not provide a mechanism for masking interrupts generated in this manner.
- Exceptions
  - **Processor-detected exceptions.** These are generated when the processor detects program and machine errors. They are further classified as *faults*, *traps*, and *aborts*.
  - **Software-generated exceptions.** The INTO, INT3, BOUND, and INT*n* instructions generate exceptions. (The INT*n* instruction generates an exception when an exception vector number as an operand.) These instructions allow checks for specific exception conditions to be performed a specific points in the instruction stream. For example, the INT3 instruction causes a breakpoint exception to be generated.

5.3. EXCEPTION CLASSIFICATIONS

Exceptions are classified as *faults*, *traps*, or *aborts* depending on the way they are reported and whether the instruction that caused the exception can be restarted with no loss of program or task continuity.

Faults — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than the instruction following the faulting instruction.

Traps — A trap is an exception that is reported immediately following the execution of the trapping instruction. Some traps allow execution of a program or task to be continued without loss of program continuity; others do not. The return address for the trap handler points to the instruction to be executed after the trapping instruction.

Aborts — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Table 5-1. Protected Mode Exceptions and Interrupts

Vector No.	Description	Interrupt or Exception Type	Error Code	Source
0	Divide Error (#DE)	Fault	No	DIV and IDIV instructions.
1	Debug (#DB)	Fault/ Trap	No	Any code or data reference.
2	NMI Interrupt	Non-Maskable	No	External interrupt.
3	Breakpoint (#BP)	Trap	No	INT3 instruction.
4	Overflow (#OF)	Trap	No	INTO instruction.
5	BOUND Range Exceeded (#BR)	Fault	No	BOUND instruction.
6	Invalid Opcode (#UD)	Fault	No	UD2 instruction or reserved opcode.
7	Device Not Available (#NM)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	Double Fault (#DF)	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9	CoProcessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. Pentium® Pro processor does not generate this exception.

10	Invalid TSS (#TS)	Fault	Yes	Task switch or TSS access.
11	Segment Not Present (#NP)	Fault	Yes	Loading segment registers or accessing system segments.
12	Stack Fault (#SS)	Fault	Yes	Stack operations and SS register loads.
13	General Protection (#GP)	Fault/Trap	Yes	Any memory reference and other protection checks.
14	Page Fault (#PF)	Fault	Yes	Any memory reference.
15	(Intel reserved. Do not use.)		No	
16	Floating-Point Error (#MF)	Fault	No	Floating-point or WAIT/FWAIT instruction.
17	Alignment Check (#AC)	Fault	Yes (Zero)	Any data reference in memory.
18	Machine Check (#MC)	Abort	Model Dependent	Model dependent.
19-31	(Intel reserved. Do not use.)			
32-255	Maskable Interrupts	Maskable		External interrupt or INT *n* instruction.

Table 5-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Descriptions
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	External Interrupts - NMI Interrupts - Maskable Interrupts
6	Faults from Fetching Next Instruction - Code Breakpoint Fault - Code Segment Limit Violation - Code Page Fault
7	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Illegal Opcode - Coprocessor Not Available
8 (Lowest)	Faults on Executing an Instruction - Floating-point exception - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check

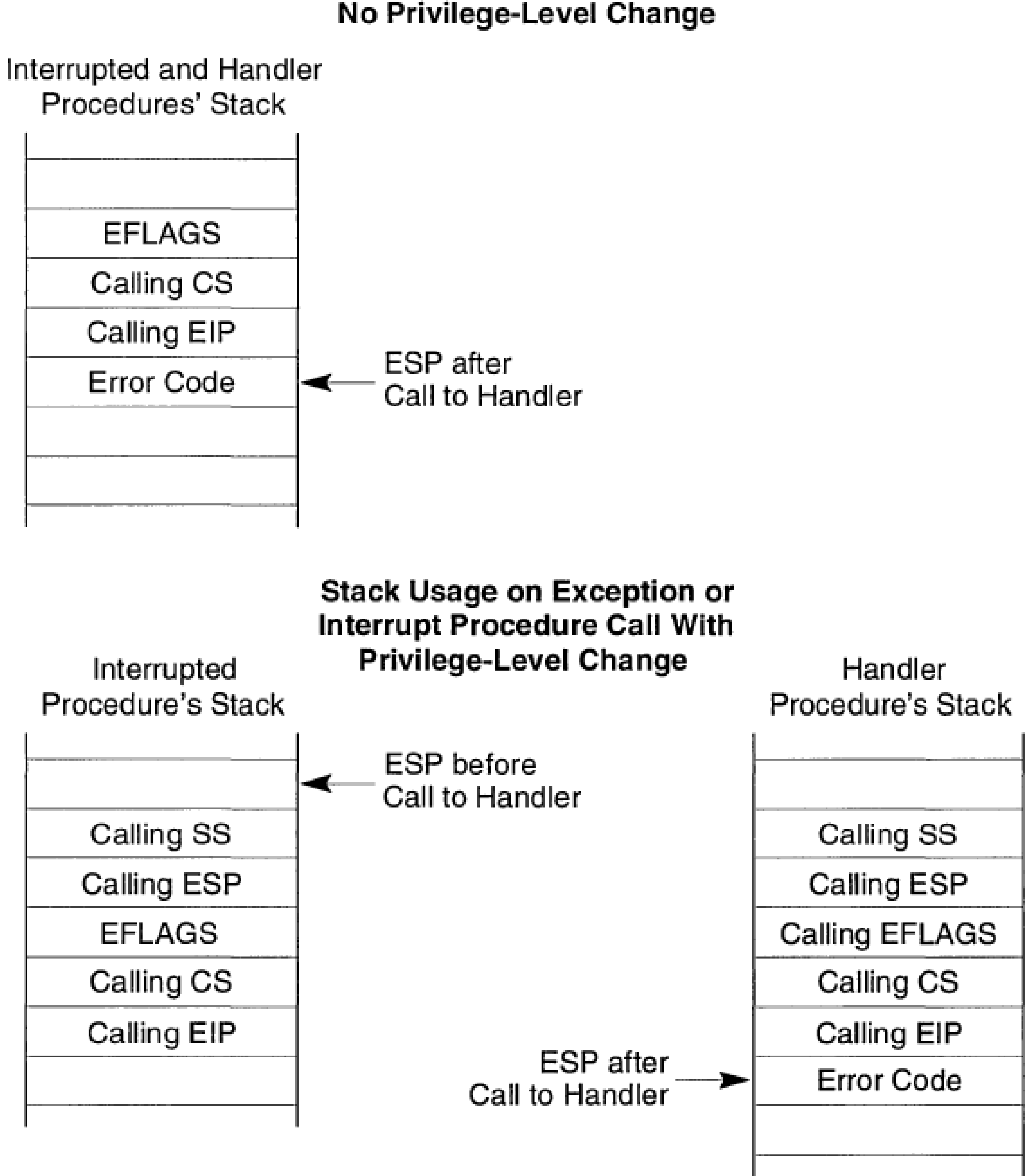


Figure 5-4. Stack Usage on Calls to Interrupt and Exception Handling Routines

2.2. MODES OF OPERATION

The Pentium Pro processor supports three operating modes and one quasi-operating mode:

- **Protected mode.** This is the native operating mode of the processor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode for all new applications and operating systems.

- **Real-address mode.** This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).

- **System management mode (SMM).** The system management mode (SMM) is a standard architectural feature in all Intel Architecture processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

- **Virtual-8086 mode.** In protected mode, the processor supports a quasi-operating mode known as *virtual-8086 mode*. This mode allows the processor execute 8086 software in a protected, multi-tasking environment.

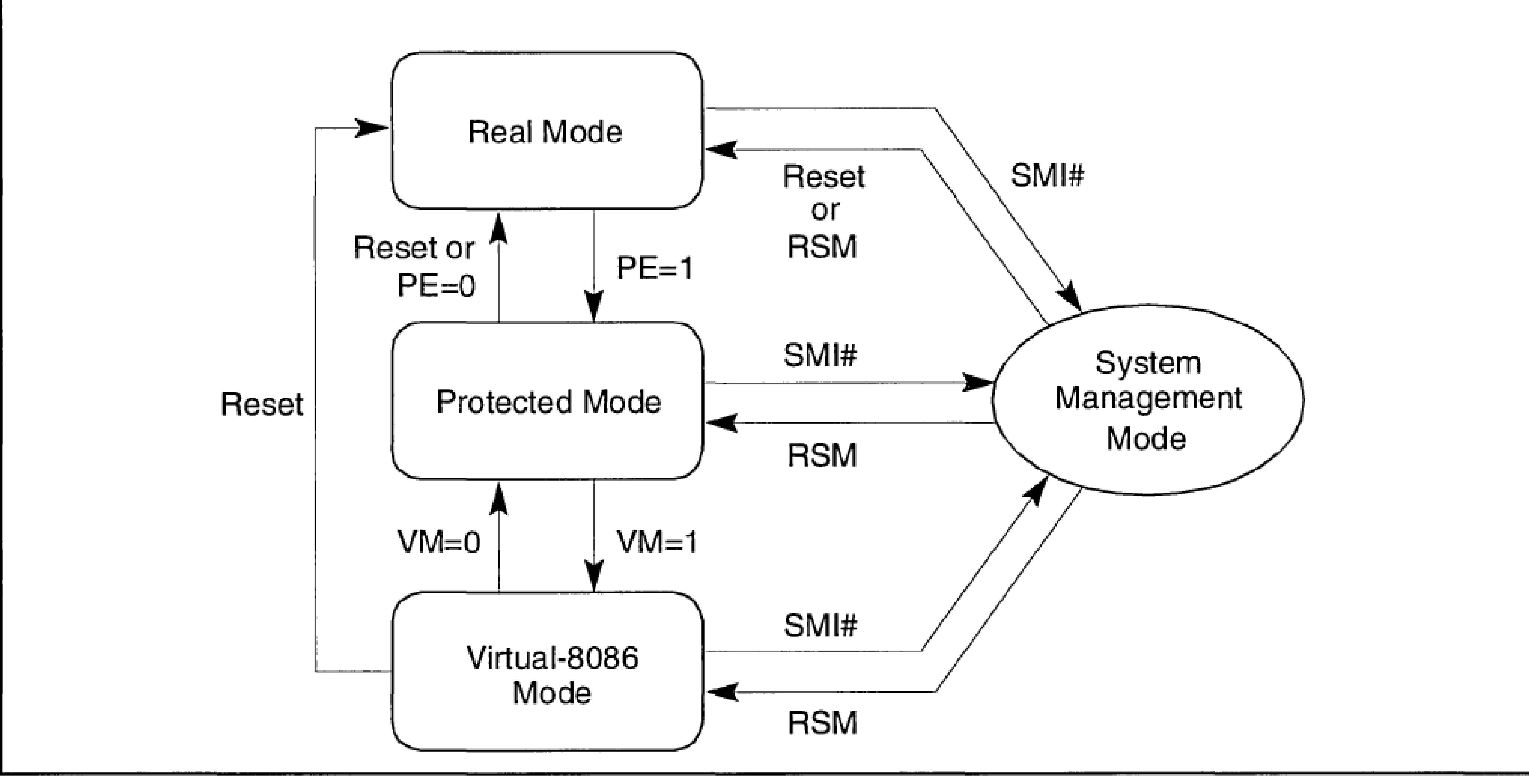


Figure 2-2. Transitions Among the Processor's Operating Modes

8.1.2. Processor Built-In Self Test (BIST)

Hardware may request that the BIST be performed at power-up. If the BIST is performed, it takes about 5.5 million clock periods to complete on the Pentium Pro processor. (This clock count is model-specific and Intel reserves the right to change the exact number of periods without notification.)

The EAX register is clear (0H) if the processor passed the BIST. A non-zero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

Table 8-1. Pentium® Pro Processor State Following Reset

Register	RESET	INIT
EFLAGS[1]	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H
CR0	60000010H	Note 2
CR2/CR3/CR4	00000000H	00000000H
CS	selector = F000H base = FFFF0000H limit = FFFFH AR = Present, R/W, Accessed	selector = 0F000H base = FFFF000H limit = FFFFH AR = Present, R/W, Accessed

SS, DS, ES, FS, GS	selector = 0000 base = 0000H limit = FFFFH AR = Present, R/W, Accessed	selector = 0000 base =0000H limit = 0FFFFH AR = Present, R/W, Accessed
EDX	000006xxH	000006xxH
EAX	0[3]	0
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H
LDTR, Task Register	selector = 0000H base = 00000000H limit = FFFFH AR = Present, R/W	selector = 0000H base = 00000000H limit = FFFFH AR = Present, R/W
GDTR,IDTR	base = 00000000H limit = FFFFH AR = Present, R/W	base = 00000000H limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H
Time Stamp Counter	0	Unchanged
Perf. Counters and Event Select	0	Unchanged

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. If Built-In Self Test (BIST) is invoked, EAX is 0 only if all tests passed.

8.1.3. Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 8-2). The device ID field is set to the value 6H, 5H, 4H, or 3H to indicate a Pentium Pro, Pentium, Intel486, or Intel386 processor, respectively. Different values may be returned for the various members of these Intel Architecture families. For example the Intel386 SX processor returns 23H in the device ID field. Binary object code can be made compatible with other Intel processors by using this number to select the correct initialization software.

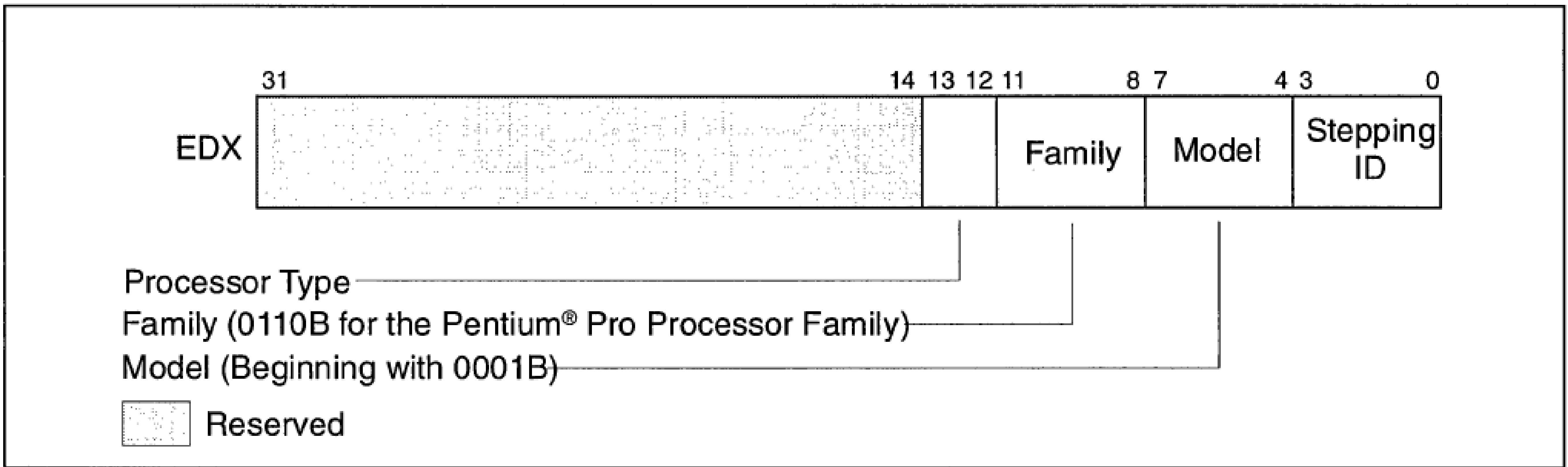


Figure 8-2. Processor Type and Signature in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The upper word of EDX is reserved following reset.

8.1.4. First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the uppermost physical address of the Pentium Pro processor. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call (which would cause the CS selector value to be changed).

Four Gigabytes in Real Mode

by Thomas Roden

Until now, all *X*86 family processors have been limited to 1M (or 1M + 64K - 16 in the case of the 80286) of immediately available linear memory when running under real-address mode (real mode). With the advent of the 80386 and 80486 processors, this is no longer the case. With a little manipulation at initialization time, it is possible to address all 4 gigabytes of physical address space in real mode without the overhead of virtual 86 mode. In addition, it is not necessary to violate DOS to do so.

While real mode of the 80386 family of processors is generally viewed as a mode that is equivalent to the normal operation of the 8086 and 8088 processors, this is not entirely true. The real modes of these processors are intended to be only compatible with—not identical to—lesser processors of the *X*86 family. This is borne out by the fact that extended registers and the additional segment registers FS and GS are available in real mode.

It is not illegal to use the extended registers for addressing with the additional segment registers, as in the following statement:

```
MOV    DWORD PTR GS:[EAX], EBX
```

The only proviso is that EAX must be less than 64K; otherwise, a general protection fault results. It seems curious that such an error can occur in real mode—a mode where protection is nominally unused.

Another curious feature is that when returning to real mode from

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

protected mode by clearing the Protection Enabled (PE) bit in Control Register 0 (CR0), the system is required to first load all segment registers with selectors that reference segments of size 64K (see the Intel *80386 Programmers Reference Manual*, Ahearn-Wallstrom, 1986). When this is not done, general protection faults can occur upon access of those segments.

This behavior suggests that the limit-checking aspects of protection are active in real mode, but the default state (set at power on or reset) is such that the limits of the segments are all set to 64K - 1, normally obscuring the fact that limit checking is truly in force.

A way to test this is to have a program enter protected mode, set a segment register to refer to a selector of size greater than 64K, return to real mode, and attempt to access data farther than 64K from the beginning to the segment in question.

Tests on the 80386DX, 80386SX, and 80486 have shown that it is indeed possible to reference data more distant than 1M + 128K (120000h) from the beginning of a segment, and there is no reason to believe that this behavior is not consistent for the entire 4 gigabytes that can be referenced by an extended register.

In general, the 80386 and 80486 can be used as more than a fast 8088 without having to support an operating system other than DOS.

This technique has many potential uses. Applications that are 386 aware can run in the faster real mode and still access all linear memory available in the system without disturbing DOS. System software can access normally hidden memory-mapped I/O locations without the overhead it takes to change processor modes. In general, the 80386 and 80486 can be used as more than a fast 8088 without having to support an operating system other than DOS.

A danger of using this technique is that software that performs a hardware reset (normally via the keyboard controller to return from protected mode to real mode) will cause the segment limits to return to 64K - 1. This can be handled by an Interrupt Service Routine (ISR) that detects when general protection faults are caused by an attempt to address in this fashion and reinstates the extended limits.

It's important to remember that this is a system-oriented technique and that it risks collision with other techniques that attempt to allow greater memory access. Most notably, if this is attempted from a virtual 86 task, it will definitely cause a general protection fault. More generally, any extended memory manager will be in danger of colliding with data accessed via the extended registers.

Only data-segment limits have been explored with this technique; however, it may be possible to perform the same sort of manipulations on the Default Bit (D-bit) of code segments to have full use of the 32-bit Extended Instruction Pointer (EIP) as well. Similarly, it may be possible to affect the Expand-Down (E-bit) to instill better stack-underflow recovery in real mode as well.

It is important to keep in mind that the 80386 family of processors is extremely powerful. Even in real mode, these processors allow new features that have yet to be fully investigated and exploited. PJ

Thomas Roden is a Software Engineer for AST Research, Inc., of Irvine, California, where he is working on software validation of system integrity and general pontification. Readers may write to Thomas in care of AST Research, Inc., 2121 Alton Ave., Irvine, CA 92714.

Code follows.

C Programmers
Explore a world of DOS memory
Beyond 640K

Use the PowerSTOR C library to exploit:
- LIM 4.0 or LIM 3.2Expanded Memory
- AT Style Extended Memory
- Hard Disk Space
- Any Combination

- PowerSTOR is the easy way to use system memory resources. It self-configures at run time, under control of your software.
- PowerSTOR makes system memory resources available to your software as STORheaps. STORheaps can be allocated and de-allocated dynamically. Up to a gigabyte of PowerSTOR memory can be used as one huge STORheap or many smaller STORheaps.
- PowerSTOR uses none of the normal heap and it minimizes its usage of stack space. These valuable resources are preserved for your use.

Library with K & R standard source code — only $92.50
(includes shipping)

Order today and explore beyond the limits.

Acme Software

P. O. Box 40734, Portland, Oregon 97240 — Phone (503) 235-2816

Number 305 on the Reader Service Card

PRODUCED 2004 BOUNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

All code in this issue is available on 360K, MS-DOS disks.
ORDER BY PHONE — VISA/MC orders call toll free 800-234-0386.
Mail-order information on pages 8 and 96.

Code from "Four Gigabytes . . . ," by Thomas Roden

1

LISTING 1

```
;***********************************************************************
;*
;*      TOMCOM.ASM -    .COM program to relax segment limit on GS in
;*                      real-address mode.
;*
;*      Assembled using Microsoft Macro Assembler 5.1
;*
;************************************************************************

;*      Need to use 80386 protected mode instructions
        .386P

;
;       JUMPFAR does a far jump in USE16 segments
;
JUMPFAR MACRO   PARM1,PARM2
        DB      0EAH                    ; jump far direct
        DW      (OFFSET PARM1)          ; to this offset
        DW      PARM2                   ; in this segment
        ENDM

CMOS_ADDR               EQU     0070H
CMOS_DATA               EQU     0071H

SYS_PROT_CS             EQU     0008H
SYS_REAL_SEG            EQU     0010H
SYS_MONDO_SEG           EQU     0018H

cr              equ     0dh
lf              equ     0ah
dos_call        equ     21h
print_func      equ     09h

cseg    segment use16 public    'code'
        assume  cs:cseg,ds:cseg,es:cseg,ss:cseg

        org     100h
begin:
;
;       Setup Environment
;
                mov     sp,offset stk_top               ; setup stack
                mov     ax,cs                           ; store program segment

;       WARNING:  Self-modifying code.  Not needed if done in ROM.
                mov     word ptr cs:self_mod_cs,ax

                mov     ds,ax                           ; set ds := cs
                jmp     init_com                        ; go do the work

;-------------------------------------------------------------------
;
;       Variables
;
;-------------------------------------------------------------------
com_gdt_ptr     dq      ?                               ; for LGDT
;                       Limit   Base    Rights  LimitHi
com_gdt         dw      00000h, 00000h, 00000h, 00000h  ; unusable
                dw      0FFFFh, 00000h, 09A00h, 00000h  ; code seg
                dw      0FFFFh, 00000h, 09200h, 00000h  ; data seg
                dw      0FFFFh, 00000h, 09200h, 0008Fh  ; mondo seg (4Gb)
com_gdt_end     label   word
;-------------------------------------------------------------------

load_msg        db      'Segment Helper Loaded',cr,lf
                db      'Use extended register and GS: to activate',cr,lf,'$'

fixup_Msg       db      'System Altered',cr,lf,'$'

                db      255 dup(0ffh)                   ;local stack
```

2

```
stk_top         db              (0ffh)


;*****************************************************************
;*      Go to protected mode to relax limit on GS               *
;*      Entry   -       none
;*      Exit    -       gs: invisible selector information fixed up
;*                      ds & ss: set to cs (ok for COM file)
;*      WARNING:        MUST EXECUTE BELOW 1M, GATE A20 NOT DISABLED
;*****************************************************************
kill_seg_limit  proc    near

                mov     ax, cs                          ; get linear address
                movzx   eax, ax
                shl     eax,4

                mov     ebx, eax                        ; store copy of CS linear
                mov     word ptr cs:com_gdt+10, ax      ; store in code segment desc
                mov     word ptr cs:com_gdt+18, ax      ; store in data segment desc
                ror     eax, 16                         ; swap words
                mov     byte ptr cs:com_gdt+12, al      ; bits 16-23
                mov     byte ptr cs:com_gdt+20, al      ; bits 16-23
;
;       Setup Limit and Base for GDTR
;
                add     ebx, offset com_gdt
                mov     word ptr cs:com_gdt_ptr, (offset com_gdt_end - com_gdt - 1)
                mov     dword ptr cs:com_gdt_ptr+2, ebx

;       store flags for restoring after cli
                pushf
;       do NOT allow interrupts as IDT is not valid in protected mode
                cli
;       disable NMI's here
                in      al,CMOS_ADDR
                mov     ah,al
                or      al, 080H                ; don't disturb rest of 70
                out     CMOS_ADDR, al
                and     ah, 080H
                mov     ch, ah                  ; store old state of NMI mask

                lgdt    fword ptr cs:com_gdt_ptr        ; assume DOS isn't using this

                mov     bx, cs                  ; save com segment

                mov     eax,cr0
                or      al,01                   ; set PE bit
                mov     cr0,eax                 ; protection enabled
                jumpfar ksl_pmode,SYS_PROT_CS   ; purge queue and fix CS
ksl_pmode:
                mov     ax, SYS_REAL_SEG        ; prepare limits on segments
                mov     ss, ax
                mov     ds, ax
                mov     es, ax
                mov     fs, ax
;
;       Here are the instructions that makes it all possible
;
                mov     ax, SYS_MONDO_SEG               ; gs will now be 4G
                mov     gs, ax

                mov     eax,cr0
                and     al,NOT 01               ; clear PE bit
                mov     cr0,eax                 ; protection disabled
;
;       the following relies on self-modification performed at beginning
;       of program.  The following macro would be useful here for ROM code only,
;       so it is manually expanded to facilitate the (yech) self-modification.
;
;               jumpfar ksl_rmode,SYS_REAL_CSEG ; purge queue and fix CS
;
                DB      0EAH                    ; jump far direct
                DW      (OFFSET ksl_rmode)      ; to this offset
;       (please don't send me to programmer's hell for this one)
self_mod_cs     DW      ?                       ; in this segment
ksl_rmode:
```

Code continues . . .

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Roden code, continued

3

```
                mov     ss, bx                          ; get seg regs back
                mov     ds, bx
                xor     ax, ax                          ; clear unused to be cleanly
                mov     es, ax
                mov     fs, ax
                mov     gs, ax
;
;       back in Real Mode, IDT is ok again
;
                in      al,CMOS_ADDR
                and     al, 07FH
                or      al, ch                          ; restore old state of NMI mask
                out     CMOS_ADDR, al
;
;       note that 386 does not have POPF problem
;
                popf                                    ; restore IRQ mask

                ret

kill_seg_limit  endp


;*****************************************************************
;*      Main Control Routine Starts Here                         *
;*****************************************************************
initialize      proc    near
        assume  ds:cseg
;
init_com:

;display message and then relax limit checking on GS
;
        mov     dx,offset load_msg
        mov     ah,print_func
        int     dos_call
;
; fix gs
;
        call    kill_seg_limit

        mov     dx,offset fixup_Msg
        mov     ah,print_func
        int     dos_call
;
;check amount of memory to retain
;
        mov     ax,4C00h        ;Terminate with return code function
        int     dos_call
;
initialize      endp
;-----------------------------------------------------------


cseg    ends
        end     begin
```

LISTING 2

```
/*
 * tompeek.c
 *
 * High memory peek utility
 *
 * This program returns a screen dump of ram (by byte) at the provided
 * linear (which is also physical) address for the provided number of bytes.
 *
 *              Compiled using Microsoft C 5.1--Small Model
 *
 */
#include <stdio.h>
#include <dos.h>

/* Main function.
```

4

```
 *
 * Check command line information, and perform the appropriate hex-dump
 *
 */
main(argc, argv)
int     argc;
char    **argv;
{
        short   size;
        long    source;
        void    tomPeek(long,short);

        if(argc==3)
                {
                sscanf(argv[1],"%lx",&source);
                sscanf(argv[2],"%x",&size);
                tomPeek(source,size);
                }
        else
                {
                printf("usage: TomPeek <address> <size>\n");
                }

}

/*
 *      tomPeek -       Dump the memory at linear address "source" for "size" bytes
 *                              of sixteen-byte lines
 */
void    tomPeek(source,size)
long    source;
short   size;
{
        char    peekBuf[16];
        short   i,j,k;
        extern short near       get_high_mem(char *,short,long);

        for(i=0;i<size;i+=16)
                {
                j = size-i;
                if(j>16)
                        j = 16;
                get_high_mem(peekBuf,j,source);
                source += j;
                for(k=0;k<j;k++)
                        {
                        printf("%02X ",((short)(peekBuf[k]))&0xFF);
                        }
                printf("\n");
                }
}
```

LISTING 3

```
;*****************************************************************
;*
;*      PEEPER.ASM -    Assembly subroutines to move memory from a 4G linear
;*                      address into a buffer in the current data segment
;*
;*      Assembled using Microsoft Macro Assembler 5.1
;*
;*****************************************************************

;*      Need to use 80386 protected mode instructions
        .386P


KBC_IBF_BIT             EQU     02H
KBC_WOP_CMD             EQU     0D1H
KBC_ROP_CMD             EQU     0D0H
A20_OFF_DATA            EQU     0DDH
A20_ON_DATA             EQU     0DFH
KBC_NOP_CMD             EQU     0FFH

KBC_DATA_PORT           EQU     060H
KBC_CMD_PORT            EQU     064H
```

Code continues . . .

ELECTRONIC REPRODUCTION PROHIBITED
PRODUCED 2004 BY UNZ.ORG

5

```
GA20_FAIL_ERR_CODE          EQU      0FFFFH

_TEXT    segment use16 public    'CODE'
         assume   cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT


;*********************************************************************
;*      get_high_mem(dest,len,source) - copy memory from source to dest (len bytes)
;*      Entry    -         dest -           buffer in current DS
;*                         len -            size of buffer in bytes
;*                         source -         32-bit linear address of source memory
;*      Exit     -         gs: set to zero
;*      WARNING:           Forces GateA20 low when done
;*********************************************************************
                 public   _get_high_mem
ghm_dest         equ      word ptr [bp+4]
ghm_len          equ      word ptr [bp+6]
ghm_source       equ      dword ptr [bp+8]
_get_high_mem    proc     near
                 push     bp
                 mov      bp, sp
                 push     di

                 mov      al, 080h                      ; allow normal A20
                 call     gate_A20

                 xor      ax, ax
                 mov      gs, ax                        ; zero out gs
                 mov      ax, ds
                 mov      es, ax                        ; es := ds
                 mov      di, ghm_dest
                 mov      cx, ghm_len
                 mov      ebx, ghm_source
                 jcxz     ghm_bottom
ghm_top:
                 mov      al, byte ptr gs:[ebx]
                 inc      ebx
                 stosb
                 loop     ghm_top
ghm_bottom:
                 xor      al, al              ; cripple A20 for DOS
                 call     gate_A20

                 pop      di
                 pop      bp
                 ret

_get_high_mem    endp
;*********************************************************************
;*      gate_A20(how)    -        set gateA20 to the desired state
;*      Entry        - AL==80h => A20 on
;*                     AL==00h => A20 off
;*      Exit     -         GateA20 in desired state
;*      WARNING:           Assumes preferred states of output port bits
;*********************************************************************

;
;       gate_A20 -         set gateA20 to the desired state
;                Entry  -  AL==80h => A20 on  (no 1M wrap)
;                          AL==00h => A20 off (1M wrap)
;
gate_A20         proc     near
                 test     al,080H             ; al&80 = 0 ?
                 jnz      short ga20_on
;        al is zero, turn a20 off
                 mov      ah,A20_OFF_DATA     ; kbc off gate a20
                 call     empty_KBC           ; clear to use
                 jnz      short ga20_err      ; didn't clear
                 jmp      short ga20_KBC_ctl  ; cleared, go fix KBC A20
ga20_on:
                 mov      ah,A20_ON_DATA      ; kbc on gate a20 if needed
ga20_KBC_ctl:
                 mov      al,KBC_WOP_CMD      ; Write Output Port
                 out      KBC_CMD_PORT,al     ; command it
                 call     empty_KBC
                 jnz      short ga20_err
                 mov      al,ah
```

6

```
                 out      KBC_DATA_PORT,al
                 call     empty_KBC                ; wait till it clears
                 jnz      short ga20_err           ; didn't clear
                 mov      al,KBC_NOP_CMD           ; flush it through
                 out      KBC_CMD_PORT,al          ; command it
                 call     empty_KBC
                 jnz      short ga20_err
;        we're here, it worked!!
ga20_ok:
                 xor      ax,ax
                 jmp      short ga20_out
ga20_err:
                 mov      ax,GA20_FAIL_ERR_CODE
ga20_out:
                 ret
gate_A20         endp

;
;       empty_KBC -      Empty keyboard controller input buffer
;                Entry - none
;                Exit  - bashes AL
;
empty_KBC        proc     near
                 push     cx                       ; save reg
                 xor      cx,cx                    ; set for max timeout
empty_KBC_loop:
                 in       al,KBC_CMD_PORT          ; get KBC status
                 test     al,KBC_IBF_BIT           ; check IBF bit
                 loopnz   empty_KBC_loop           ; try till timeout
                 pop      cx                       ; restore reg
                 ret                               ; that's all folks
empty_KBC        endp



_TEXT    ends
         end;
```

LISTING 4

```
masm /Zi tomcom;
link tomcom /co;
exe2bin tomcom.exe tomcom.com
del tomcom.exe
```

LISTING 5

```
TOMPEEK.OBJ : TOMPEEK.C
    CL /c /Zi TOMPEEK.C

PEEPER.OBJ : PEEPER.ASM
    MASM PEEPER;

TOMPEEK.EXE : TOMPEEK.OBJ PEEPER.OBJ
    LINK /CO TOMPEEK+PEEPER;
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Big Real Mode

Post-286 processors can address up to 4GB of memory space while in real mode, so long as they have at least once been switched to protected mode and back to real mode since the last reset. This is sometimes referred to as big real mode.

As an example, prior to switching back to real mode, the protected mode software sets up a segment descriptor table entry (see the chapter entitled “Intro to Segmentation” on page 77) that describes a segment as starting somewhere above 1MB and having a length of 64KB. The segment register is then loaded with a value that selects this descriptor, loading the invisible part of the segment register with the new start address and length. When the switch is made back to real mode, as long as the programmer doesn’t load a new value into the segment register, the previous segment definition holds true. This permits the programmer to access any location ~~within 64K of the segment’s base address.~~

~~Note that the offset is still restricted to a maximum of FFFFh. A GP exception results when a larger offset is specified.~~

The 286 lacks this capability (because it cannot be switched from protected to real mode without resetting the processor, thus setting the contents of the invisible part of the segment register to values restricting the processor to accesses within the first meg of memory space).

Tom Shanley, ISBN 020155447X, 1996

ATBASE.ASM:

```
                ALIGN   4
                Public  GDTR1
GDTR1:                                          ; global descriptor table register
ifndef  Flash_16K_8K_8K_Unit
                dw      8*5                     ; LIMIT
else    ;Flash_16K_8K_8K_Unit
                dw      8*6                     ; LIMIT
endif   ;Flash_16K_8K_8K_Unit
                dw      offset GDT1
                dw      0fh                     ; in 0F000h segment

                Public  GDT1
GDT1:                                           ; null descriptor
                dw      0                       ; limit
                dw      0                       ; base
                db      0                       ; hibase
                db      0                       ; access
                db      0                       ; hilimit
                db      0                       ; msbase

                Public  CODE1_DT
CODE1_DT:                                       ; cs - prom code segment
CODE1_INDEX     =       ((offset CODE1_DT - offset GDT1)/8) SHL 3

                dw      0ffffh                  ; limit
                dw      0                       ; base  a15-a0
                db      0fh                     ; hibase   a23-a16, assume we have 64k prom
                db      9fh                     ; access
                db      0                       ; hilimit
                db      0                       ; msbase   a31-a24
```

ATORGS.ASM:

```
;[]======================================================================[]
;EnterBigRealMode
;[]======================================================================[]
EnterBigRealMode:
                mov     ax,cs
                mov     ds,ax
                assume  ds:dgroup
                mov     si, offset GDTR1
                lgdt    fword ptr ds:[si]       ;load descriptor table
                mov     eax,cr0                 ;Enter protected mode
                or      al,1
                mov     cr0,eax
                jmp     short Enter_Protect_Mode1
Enter_Protect_Mode1:
                mov     ax,offset DATA1_INDEX
                mov     ds,ax                   ; ds = 00000000h
                mov     es,ax                   ;
                mov     ds,ax
;Switch back to real mode, DS and ES limits are set
;
                mov  eax,cr0
                and  al,NOT 1
                mov  cr0,eax
                jmp  @f                         ; clear prefetch queue.
@@:
                xor     ax, ax
                mov     ds, ax                  ;DS and ES limit are set
                mov     es, ax
                ret
```

```
GDT     dw      15,0,0,0,-1,0,9200h,8Fh; point to 0-4GB (CF or 8F?)

.code
.586p

; Set CPU to "voodoo" (unreal, real flat) mode
; so we can access up to 4 GB RAM in real mode
; Return 0 in AL if OK

_SetToFlat      proc    far
        pushad
        mov     eax,cr0          ; Test if CPU in protected mode
        test    al,1
        jz      ldlimit
        popad
        mov     al,1
        ret
ldlimit:xor     eax,eax
        xor     ebx,ebx
        push    ds               ; Load GDT with a 4 GB limit
        pop     ax
        mov     bx,offset GDT
        shl     eax,4
        add     eax,ebx
        mov     dword ptr GDT+2,eax
        cli
;ifndef __WASM__
        db      66h
;endif
        lgdt    fword ptr GDT
        mov     bx,8
        push    ds
        push    es
        push    fs
        push    gs
        mov     eax,cr0
        or      ax,1
        mov     cr0,eax
        jmp     $+2              ; Enter protected mode
        mov     gs,bx
        mov     fs,bx
        mov     es,bx
        mov     ds,bx
        and     al,not 1
        mov     cr0,eax
        jmp     $+2              ; Return to real mode
        pop     gs
        pop     fs
        pop     es
        pop     ds
        sti
        in      al,92h           ; Enable A20 line
        or      al,2
        out     92h,al
        popad
        xor     al,al
        ret
_SetToFlat      endp
```

Exceptions are generated by internal and external sources to cause the processor to handle an event; for example, an externally generated interrupt, or an attempt to execute an undefined instruction. The processor state just before handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. More than one exception may arise at the same time.

ARM supports 7 types of exception and has a privileged processor mode for each type of exception. *Table 2-3: Exception processing modes* lists the types of exception and the processor mode that is used to process that exception. When an exception occurs execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the ***Hard Vectors.***

The reserved entry at address 0x14 is for an Address Exception vector used when the processor is configured for a 26-bit address space. See *Chapter 5, The 26-bit Architectures* for more information.

Exception type	Mode	Vector address
Reset	SVC	0x00000000
Undefined instructions	UNDEF	0x00000004
Software Interrupt (SWI)	SVC	0x00000008
Prefetch Abort (Instruction fetch memory abort)	ABORT	0x0000000c
Data Abort (Data Access memory abort)	ABORT	0x00000010
IRQ (Interrupt)	IRQ	0x00000018
FIQ (Fast Interrupt)	FIQ	0x0000001c

Table 2-3: Exception processing modes

When taking an exception, the banked registers are used to save state. When an exception occurs, these actions are performed:

```
R14_<exception_mode> = PC
SPSR_<exception_mode> = CPSR
CPSR[5:0] = Exception mode number
CPSR[6] = if <exception_mode> == Reset or FIQ then = 1 else unchanged
CPSR[7] = 1; Interrupt disabled
PC = Exception vector address
```

To return after handling the exception, the SPSR is moved into the CPSR and R14 is moved to the PC. This can be done atomically in two ways:

1. Using a data-processing instruction with the S bit set, and the PC as the destination.
2. Using the Load Multiple and Restore PSR instruction.

When the processor's Reset input is asserted, ARM immediately stops execution of the current instruction. When the Reset is de-asserted, the following actions are performed:

```
R14_svc = unpredictable value
SPSR_svc = CPSR
CPSR[5:0] = 0b010011   ; Supervisor mode
CPSR[6] = 1            ; Fast Interrupts disabled
CPSR[7] = 1            ; Interrupts disabled
PC = 0x0
```

Therefore, after reset, ARM begins execution at address 0x0 in supervisor mode with interrupts disabled. See *7.6 Memory Management Unit (MMU) Architecture* on page 7-14 for more information on the effects of Reset.

If ARM executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an undefined instruction exception occurs. If an attempt is made to execute an instruction that is undefined, an undefined instruction exception occurs (see *3.14.5 Undefined instruction Space* on page 3-27).

The undefined instruction exception may be used for software emulation of a coprocessor in a system that does not have the physical coprocessor (hardware), or for general-purpose instruction set extension by software emulation.

When an undefined instruction exception occurs, the following actions are performed:

```
R14_und = address of undefined instruction + 4
SPSR_und = CPSR
CPSR[5:0] = 0b011011     ; Undefined mode
CPSR[6] = unchanged      ; Fast Interrupt status is unchanged
CPSR[7] = 1              ; (Normal) Interrupts disabled
PC = 0x4
```

To return after emulating the undefined instruction, use:

```
      MOVS PC,R14
```

This restores the PC (from R14_und) and CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

The software interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor (Operating System) function. When a SWI is executed, the following are performed:

```
R14_svc = address of SWI instruction + 4
SPSR_svc = CPSR
CPSR[5:0] = 0b010011   ; Supervisor mode
CPSR[6] = unchanged    ; Fast Interrupt status is unchanged
CPSR[7] = 1            ; (Normal) Interrupts disabled
PC = 0x8
```

To return after performing the SWI operation, use:

```
       MOVS PC,R14
```

This restores the PC (from R14_svc) and CPSR (from SPSR_svc) and returns to the instruction following the SWI.

A memory abort is signalled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. An abort will take place if the processor attempts to execute the invalid instruction. If the instruction is not executed (for example as a result of a branch being taken while it is in the pipeline), no prefetch abort will occur.

When an attempt is made to execute an aborted instruction, the following actions are performed:

```
R14_abt = address of the aborted instruction + 4
SPSR_abt = CPSR
CPSR[5:0] = 0b010111   ; Abort mode
CPSR[6] = unchanged    ; Fast Interrupt status is unchanged
CPSR[7] = 1            ; (Normal) Interrupts disabled
PC = 0xc
```

To return after fixing the reason for the abort, use:

```
        SUBS PC,R14,#4
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt) and returns to the aborted instruction.

A memory abort is signalled by the memory system. Activating an abort in response to a data access (Load or Store) marks the data as invalid. A data abort exception will occur before any following instructions or exceptions have altered the state of the CPU, and the following actions are performed:

```
R14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[5:0] = 0b010111   ; Abort mode
CPSR[6] = unchanged    ; Fast Interrupt status is unchanged
CPSR[7] = 1            ; (Normal) Interrupts disabled
PC = 0x10
```

To return after fixing the reason for the abort, use:

```
      SUBS PC,R14,#8
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt) and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed use:

```
      SUBS PC,R14,#4
```

The final value left in the base register used in memory access instructions which specify writeback and generate a data abort (LDR, LDRH, LDRSH, LDRB, LDRSB, STR, STRH, STRB, LDM, STM, LDC, STC) is IMPLEMENTATION DEFINED.

An implementation can choose to leave either the original value or the updated value in the base register, but the same behaviour must be implemented for all memory access instructions.

The IRQ (Interrupt ReQuest) exception is externally generated by asserting the processor's IRQ input. It has a lower priority than FIQ (see below), and is masked out when a FIQ sequence is entered. Interrupts are disabled when the I bit in the CPSR is set (but note that the I bit can only be altered from a privileged mode). If the I flag is clear, ARM checks for a IRQ at instruction boundaries.

When an IRQ is detected, the following actions are performed:

```
R14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[5:0] = 0b010010    ; Interrupt mode
CPSR[6] = unchanged     ; Fast Interrupt status is unchanged
CPSR[7] = 1             ; (Normal) Interrupts disabled
PC = 0x18
```

To return after servicing the interrupt, use:

```
      SUBS PC,R14,#4
```

This restores both the PC (from R14_irq) and CPSR (from SPSR_irq) and resumes execution of the interrupted code.

The FIQ (Fast Interrupt reQuest) exception is externally generated by asserting the processor's FIQ input. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching).

Fast interrupts are disabled when the F bit in the CPSR is set (but note that the F bit can only be altered from a privileged mode). If the F flag is clear, ARM checks for a FIQ at instruction boundaries.

When a FIQ is detected, the following actions are performed:

```
R14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[5:0] = 0b010001   ; FIQ mode
CPSR[6] = unchanged    ; Fast Interrupt disabled
CPSR[7] = 1            ; Interrupts disabled
PC = 0x1c
```

To return after servicing the interrupt, use:

```
      SUBS PC, R14,#4
```

This restores both the PC (from R14_fiq) and CPSR (from SPSR_fiq) and resumes execution of the interrupted code.

The FIQ vector is deliberately the last vector to allow the FIQ exception-handler software to be placed directly at address 0x1c, and not require a branch instruction from the vector.

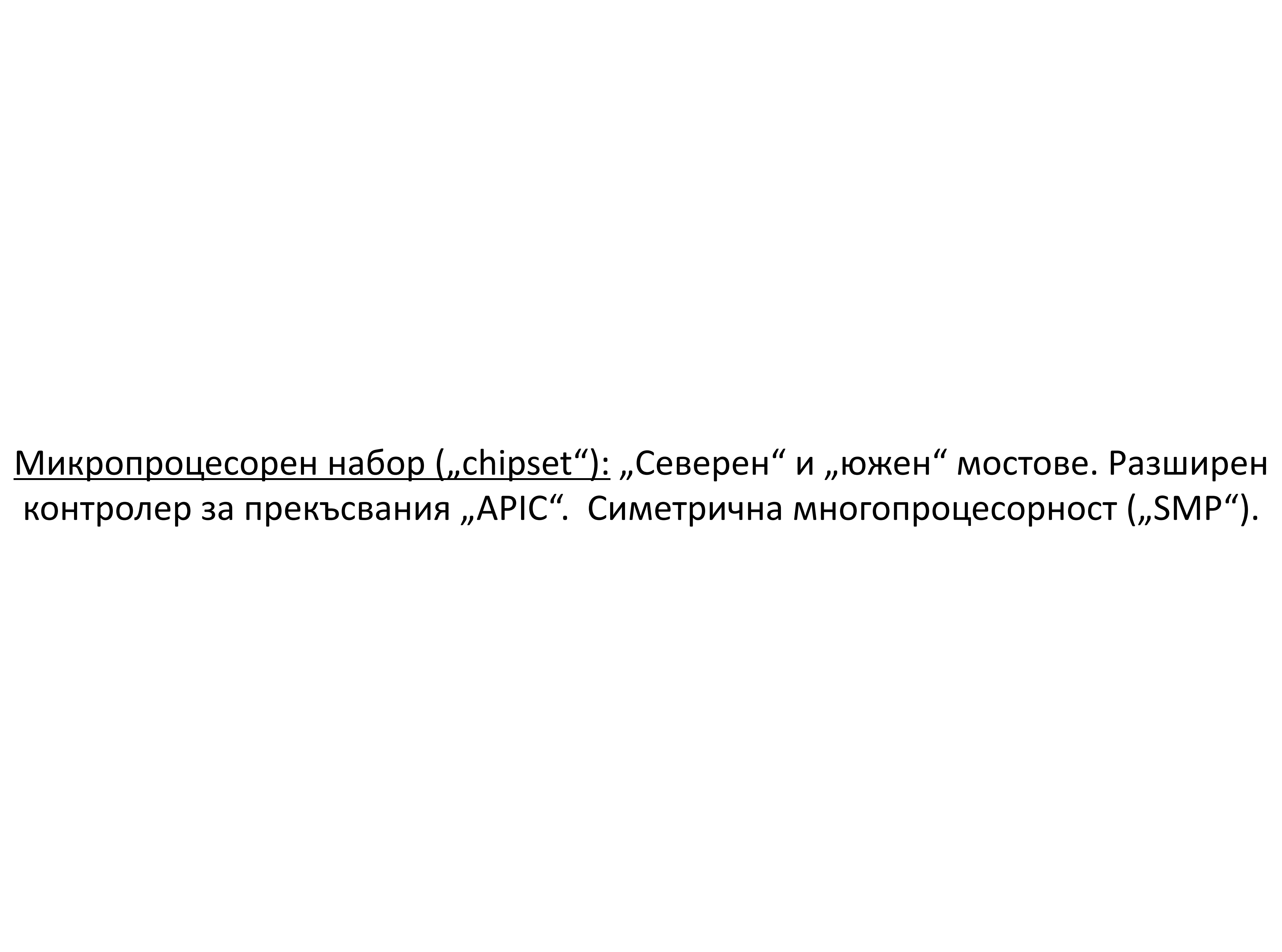
The Reset exception has the highest priority. FIQ has higher priority than IRQ. IRQ has higher priority than prefetch abort.

Undefined instruction and software interrupt cannot occur at the same time, as they each correspond to particular (non-overlapping) decodings of the current instruction, and both must be lower priority than prefetch abort, as a prefetch abort indicates that no valid instruction was fetched.

The priority of data abort is higher than FIQ and lower priority than Reset, which ensures that the data-abort handler is entered before the FIQ handler is entered (so that the data abort will be resolved after the FIQ handler has completed).

Exception	Priority
Reset	1 (Highest)
Data Abort	2
FIQ	3
IRQ	4
Prefetch Abort	5
Undefined Instruction, SWI	6 (Lowest)

Table 2-4: Exception priorities


Микропроцесорен набор („chipset“): „Северен“ и „южен“ мостове. Разширен контролер за прекъсвания „APIC“. Симетрична многопроцесорност („SMP“).

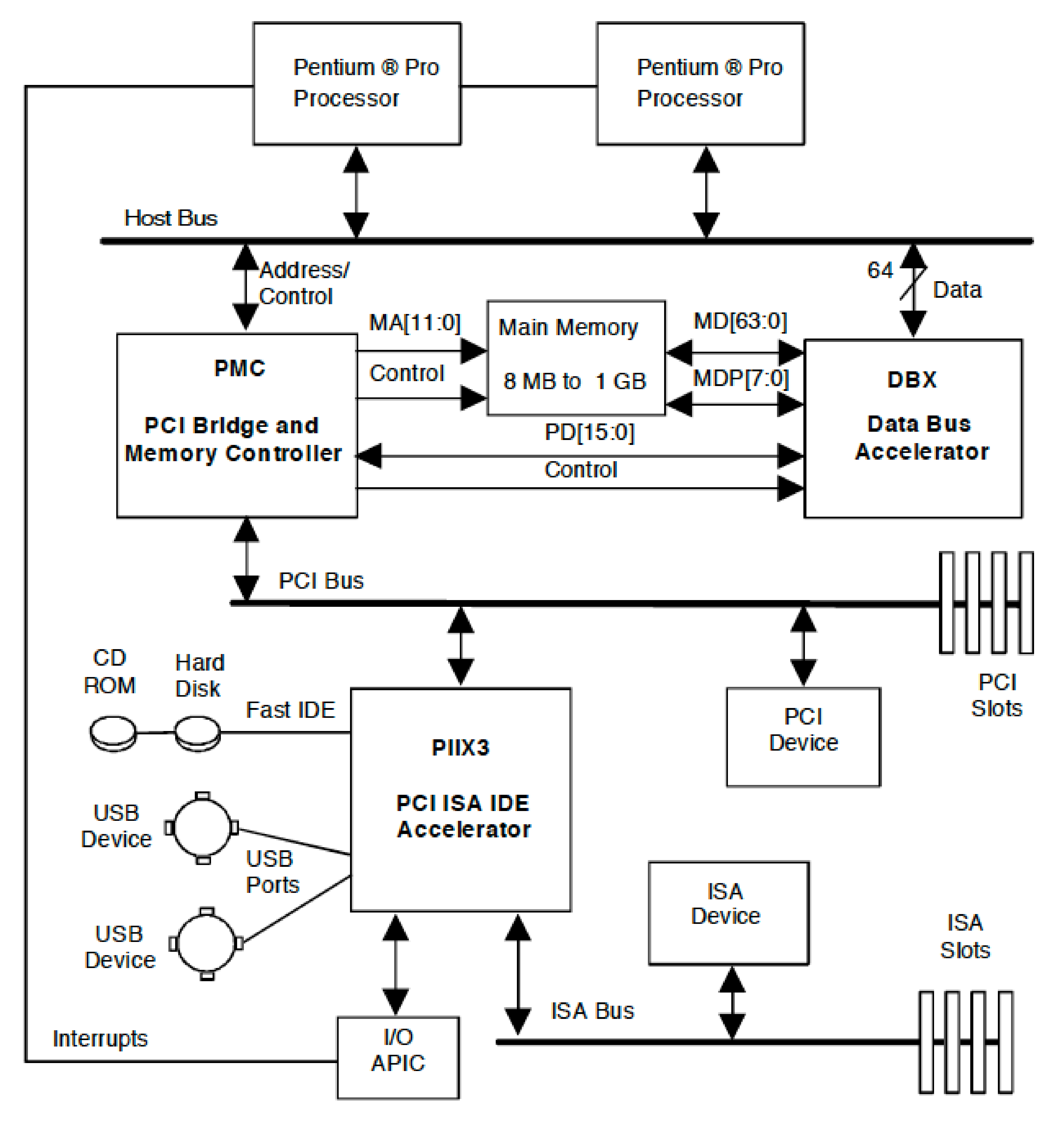
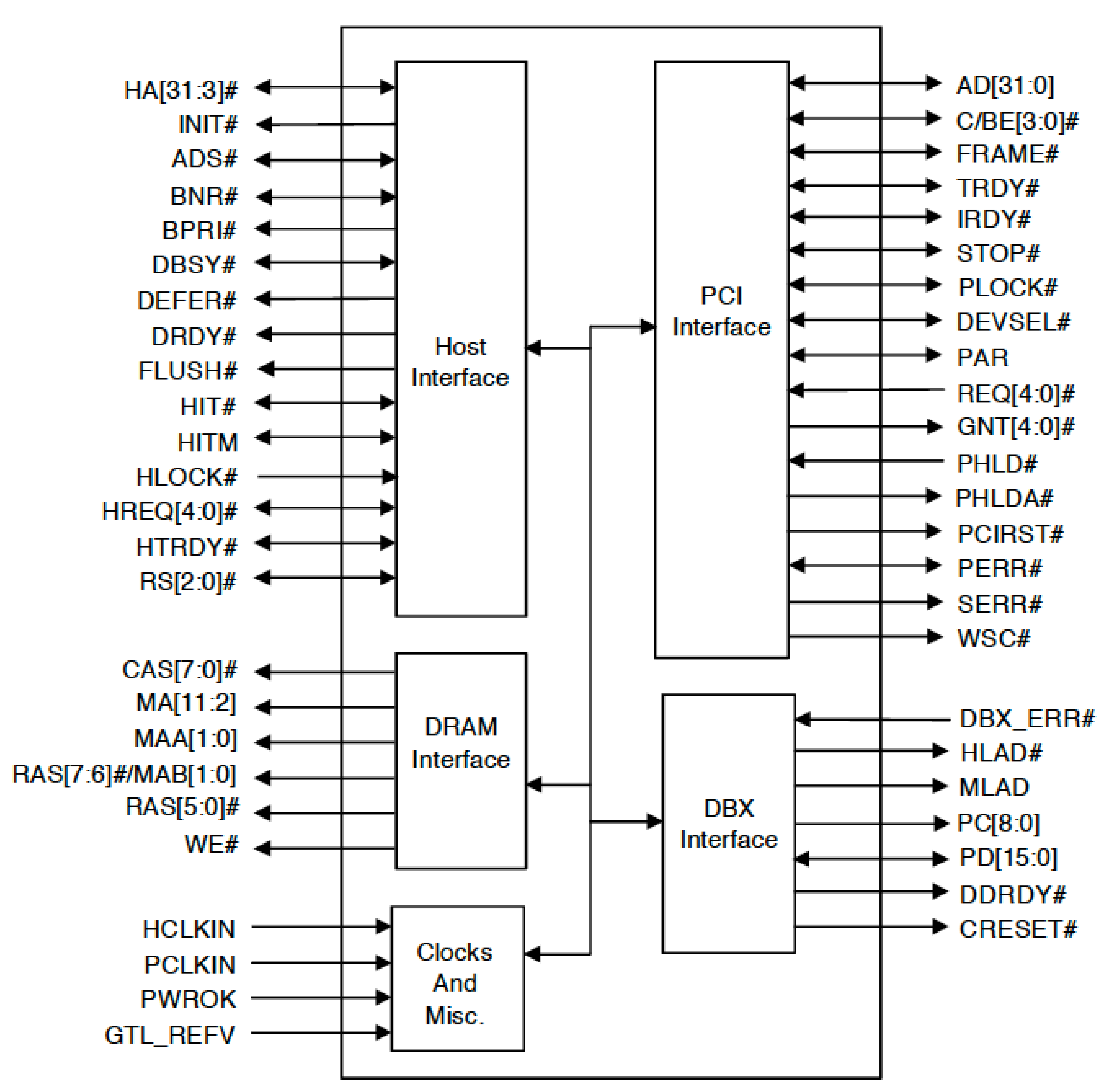


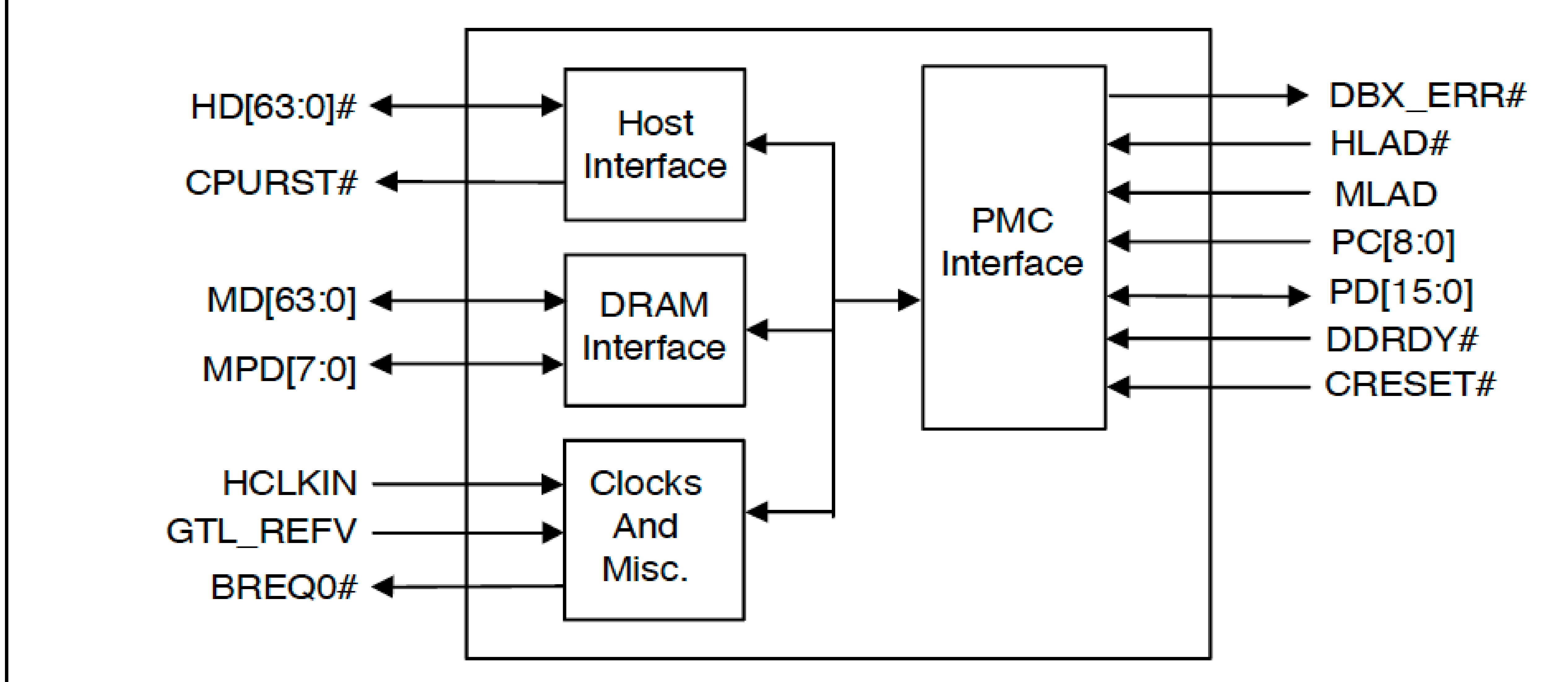
Figure 1. 440FX PCIset System Block Diagram

INTEL 440FX PCISET 82441FX PCI AND MEMORY CONTROLLER (PMC) AND 82442FX DATA BUS ACCELERATOR (DBX)

- Supports the Pentium® Pro Processors at Bus Frequencies Up To 66 Mhz
  - Supports 32-Bit Addressing
  - Optimized in-Order and Request Queue
  - Full Symmetric Multi-Processor (SMP) Protocol for up to Two Processors
  - Dynamic Deferred Transaction Support
  - GTL+ Compliant Host Bus
  - Supports USWC Cycles
- Integrated DRAM Controller
  - 8 MB to 1 GB Main Memory
  - 64/72-Bit Non-Interleaved Path to Memory
  - FPM (Fast Page Mode), EDO (Extended Data Out -Page Mode), BEDO (Extended Data Out -Burst Mode) DRAMs Providing x-222 to x-4-4-4 Burst Capability
  - Support for Auto Detection of Memory Type: BEDO, EDO or FPM
  - 8 RAS Lines Available
  - Support for 4-, 16- and 64-Mb DRAM Devices
  - Support for Symmetrical and Asymmetrical DRAM Addressing
  - Configurable Support for ECC or Parity
  - ECC with Single Bit Error Correction and Multiple Bit Error Detection
  - Read-Around-Write Support for Host and PCI DRAM Read Accesses
  - Supports 3.3V or 5V DRAMs
- PCI Bus Interface
  - PCI Rev. 2.1, 5V Interface Compliant
  - Greater than 100 MBps Data Streaming for PCI to DRAM Accesses Enables Native Signal Processing (NSP) on Systems Designed With the Pentium Pro Processor
  - Integrated Arbiter With Multi-Transaction PCI Arbitration Accelerator Hooks
  - 5 PCI Bus Masters are Supported in Addition to the Host and PCI-to-ISA I/O Bridge
  - Delayed Transaction Support
  - PCI Parity Checking and Generation Support
  - Supports Concurrent Pentium Pro and PCI Transactions to Main Memory
- Data Buffering For Increased Performance
  - Extensive CPU-to-DRAM and PCI-to-DRAM Write Data Buffering
  - Write Combining Support for CPU-to-PCI Burst Writes
- System Management Mode (SMM) Compliant
- 208-Pin PQFP PCI Bridge/ Memory Controller (PMC), 208-Pin PQFP for the 440FX PCIset Data Bus Accelerator (DBX)



PMC Simplified Block Diagram



DBX Simplified Block Diagram

2.0. SIGNAL DESCRIPTION

This section provides a detailed description of each signal. The signals are arranged in functional groups according to their associated interface.

The "#" symbol at the end of a signal name indicates that the active, or asserted state occurs when the signal is at a low voltage level. When "#" is not present after the signal name the signal is asserted when at the high voltage level.

The terms assertion and negation are used extensively. This is done to avoid confusion when working with a mixture of "active-low" and "active-high" signals. The term **assert**, or **assertion** indicates that a signal is active, independent of whether that level is represented by a high or low voltage. The term **negate**, or **negation** indicates that a signal is inactive.

The following notations are used to describe the signal and type:

I	Input pin
O	Output pin
OD	Open Drain Output pin. This requires a pull-up to the VCC of the processor core
I/O	Bi-directional Input/Output pin

The signal description also includes the type of buffer used for the particular signal:

GTL+	Open Drain GTL+ interface signal. Refer to the GTL+ I/O Specification for complete details
PCI	PCI bus interface signals. These signals are compliant with the PCI 5.0V Signaling Environment DC and AC Specifications
LVTTL	Low Voltage TTL compatible signals. These are also 3.3V outputs with 5V tolerant inputs.

2.1. PMC Signals

2.1.1. HOST INTERFACE (PMC)

Name	Type	Description
INIT#	O LVTTL	**INITIALIZATION:** INIT# is asserted (soft reset) by the PMC during a CPU shutdown bus cycle, or after the writing to the reset control register to initiate a soft reset.
HA[31:3]#	I/O GTL+	**ADDRESS BUS:** HA[31:3]# connects to the CPU address bus. The PMC drives HA[31:3]# during snoop cycles on behalf of PCI initiators. Note that the CPU address bus is an inverted bus.
ADS#	I/O GTL+	**ADDRESS STROBE:** The CPU bus owner asserts ADS# to indicate the first of two cycles of a request phase.
BNR#	O GTL+	**BLOCK NEXT REQUEST:** Used to block the current request bus owner from issuing new requests. This signal is used to dynamically control the CPU bus pipeline depth.
BPRI#	O GTL+	**PRIORITY AGENT BUS REQUEST:** The owner of this signal will always be the next bus owner. This signal has priority over symmetric bus requests and causes the current symmetric owner to stop issuing new transactions unless the HLOCK# signal is asserted. The PMC drives this signal to gain control of the CPU bus.
DBSY#	I/O GTL+	**DATA BUS BUSY:** Used by the data bus owner to hold the data bus for transfers requiring more than one cycle.
DEFER#	O GTL+	**DEFER:** The PMC uses a dynamic deferring policy to optimize for system performance. The PMC also uses the DEFER# signal to indicate a CPU retry response.
DRDY#	I/O GTL+	**DATA READY:** Asserted for each cycle that data is transferred.
FLUSH#	OD LVTTL	**FLUSH:** Issued to CPU(s) for L1/L2 cache for a write back of all cache lines in modified state and then invalidate all cache lines. This signal is asserted by the PMC to throttle the CPU bus in the deturbo mode of operation.
HIT#	I/O GTL+	**HIT:** Indicates that a caching agent holds an unmodified version of the requested line. Also, driven in conjunction with HITM#, by the target, to extend the snoop window.
HITM#	I/O GTL+	**HIT MODIFIED:** Indicates that a caching agent holds a modified version of the requested line and that this agent assumes responsibility for providing the line. Also, driven in conjunction with HIT# to extend the snoop window.
HLOCK#	I GTL+	**HOST LOCK**: All CPU bus cycles sampled with the assertion of HLOCK# and ADS#, until the negation of HLOCK# must be atomic (i.e., no PCI activity to DRAM is allowed and the locked cycle is translated to PCI, if targeted for the PCI bus.)
HREQ[4:0]#	I/O GTL+	**REQUEST COMMAND:** Asserted during both clocks of the request phase. In the first clock, the signals define the transaction type to a level of detail that is sufficient to begin a snoop request. In the second clock, the signals carry additional information to define the complete transaction type.
HTRDY#	I/O GTL+	**HOST TARGET READY:** Indicates that the target of the CPU transaction is able to enter the data transfer phase.

Name	Type	Description
RS[2:0]#	I/O GTL+	**RESPONSE SIGNALS:** Indicates the type of response: **RS[2:0] Response type** 000 Idle state 001 Retry response 010 Defer response 011 Reserved 100 Hard Failure 101 Normal without data 110 Implicit Writeback 111 Normal with data

Note: All of the signals in the host interface are described in the Pentium Pro datasheet. The preceding table highlights 440FX PCIset specific uses of these signals.

2.1.2. DRAM INTERFACE (PMC)

Name	Type	Description
CAS[7:0]#	O LVTTL	**COLUMN ADDRESS STROBE:** The CAS[7:0]# signals are used to latch the column address on the MA[11:0] lines into the DRAMs. These signals drive the DRAM array directly without external buffering.
MA[11:2]	O LVTTL	**MEMORY ADDRESS:** MA[11:2] provide multiplexed row and column address to DRAM. MA[11:2] are externally buffered to drive the address lines of the DRAM.
MAA[1:0]	O LVTTL	**LOWER MEMORY ADDRESS SET A:** MAA[1:0] are the lower two bits of the memory address used to complete the row and column address to the DRAM. These two pins are toggled during the burst phase.
RAS[7:6]#/ MAB[1:0]	O LVTTL	**ROW ADDRESS STROBES RAS7# AND RAS6# OR LOWER MEMORY ADDRESS SET B:** MAB[1:0] are the lower two bits of the memory address used to complete the row and column address to the DRAM. These signals are toggled during the burst phase. RAS[7:6]# are used to latch the row address on the MA[11:0] lines into the DRAMs. These signals should be used to select the upper two rows in the memory array. These signals drive the DRAM array directly without external buffers. The strapping on **PC8** selects the function of these pins.
RAS[5:0]#	O LVTTL	**ROW ADDRESS STROBE:** The RAS[5:0]# signals are used to latch the row address on the MA[11:0] lines into the DRAMs. Each signal is used to select one DRAM row. These signals drive the DRAM array directly without any external buffers.
WE#	O LVTTL	**WRITE ENABLE SIGNAL:** WE# is asserted during writes to main memory. During burst writes to main memory, WE# is externally buffered to drive the WE# inputs of the DRAM.

2.1.3. PCI INTERFACE (PMC)

Name	Type	Description
AD[31:0]	I/O PCI	**PCI ADDRESS/DATA:** These signals are connected to the PCI address/data bus. Address is driven by the PMC with FRAME# assertion, data is driven or received in following clocks.
DEVSEL#	I/O PCI	**DEVICE SELECT:** Device select, when asserted, indicates that a PCI target device has decoded its address as the target of the current access. The PMC asserts DEVSEL# based on the DRAM address range being accessed by a PCI initiator or if it decodes the current configuration cycle is targeted to the PMC.
FRAME#	I/O PCI	**FRAME:** FRAME# is an output when the PMC acts as an initiator on the PCI Bus. FRAME# is asserted by the PMC to indicate the beginning and duration of an access. The PMC asserts FRAME# to indicate a bus transaction is beginning.
IRDY#	I/O PCI	**INITIATOR READY:** IRDY# is an output when PMC acts as a PCI initiator and an input when the PMC acts as a PCI target. The assertion of IRDY# indicates the current PCI Bus initiator's ability to complete the current data phase of the transaction.
PLOCK#	I/O PCI	**PLOCK:** PLOCK# indicates an exclusive bus operation and may require multiple transactions to complete. When PLOCK# is asserted, non-exclusive transactions may proceed. A grant to start a transaction on the PCI Bus does not guarantee control of the PLOCK# signal. Control of the PLOCK# signal is obtained under its own protocol in conjunction with the GNT# signal. The PMC supports bus lock mode of operation.
TRDY#	I/O PCI	**TARGET READY:** TRDY# is an input when the PMC acts as a PCI initiator and an output when the PMC acts as a PCI target. The assertion of TRDY# indicates the target agent's ability to complete the current data phase of the transaction.
C/BE[3:0]#	I/O PCI	**COMMAND/BYTE ENABLE:** PCI Bus Command and Byte Enable signals are multiplexed on the same pins. During the address phase of a transaction, C/BE[3:0]# define the bus command. During the data phase C/BE[3:0]# are used as byte enables. The byte enables determine which byte lanes carry meaningful data. PCI Bus command encoding and types are listed below. **C/BE[3:0]#** — **Command Type** 0 0 0 0 — Interrupt Acknowledge 0 0 0 1 — Special Cycle 0 0 1 0 — I/O Read 0 0 1 1 — I/O Write 0 1 0 0 — Reserved 0 1 0 1 — Reserved 0 1 1 0 — Memory Read 0 1 1 1 — Memory Write 1 0 0 0 — Reserved 1 0 0 1 — Reserved 1 0 1 0 — Configuration Read 1 0 1 1 — Configuration Write 1 1 0 0 — Memory Read Multiple 1 1 0 1 — Reserved (Dual Address Cycle) 1 1 1 0 — Memory Read Line 1 1 1 1 — Memory Write and Invalidate

Name	Type	Description
PAR	I/O PCI	**PARITY:** PAR is driven by the PMC when it acts as a PCI initiator during address and data phases for a write cycle, and during the address phase for a read cycle. PAR is driven by the PMC when it acts as a PCI target during each data phase of a PCI memory read cycle. Even parity is generated across AD[31:0] and C/BE[3:0]#.
PERR#	I/O PCI	**PCI PARITY ERROR:** Pulsed by an agent receiving data with bad parity one clock after PAR is asserted. The PMC generates PERR# active if it detects a parity error on the PCI bus and the PERR# Enable bit is set.
SERR#	O PCI	**SYSTEM ERROR:** The PMC can be programmed to assert SERR# for 2 types of memory error conditions: 1. Main memory single bit ECC error 2. Main memory (DRAM) parity or multiple bit ECC error The PMC can be programmed to assert SERR# when it detects a target abort on a PMC initiated PCI cycle and when PERR# is sampled active.
PCIRST#	O PCI	**PCI RESET:** PCI bus reset forces the PCI interfaces of each device to a known state. The PMC generates a minimum 1 ms pulse for PCIRST#.
STOP#	I/O PCI	**STOP:** STOP# is an input when the PMC acts as a PCI initiator and an output when the PMC acts as a PCI target. STOP# indicates that the bus initiator must immediately terminate its current PCI Bus cycle at the next clock edge and release control of the PCI Bus. STOP# is used for disconnect, retry, and abort sequences on the PCI Bus.

2.1.4. PCI SIDEBAND INTERFACE (PMC)

Name	Type	Description
PHOLD#	I PCI	**PCI HOLD:** The PIIX3 asserts this signal to request the PCI bus.
PHLDA#	O PCI	**PCI HOLD ACKNOWLEDGE:** The PMC asserts this signal to grant PCI bus ownership to the PIIX3.
WSC#	O PCI	**WRITE SNOOP COMPLETE:** Asserted to indicate that all that the snoop activity on the CPU bus on behalf of the last PCI-to-DRAM write transaction is complete.
REQ[4:0]#	I PCI	**PCI BUS REQUEST:** REQ[4:0]# are the PCI bus request signals used by the PMC for PCI initiator arbitration.
GNT[4:0]#	O PCI	**PCI GRANT:** GNT[4:0]# are the PCI bus grant signals used by the PMC for PCI initiator arbitration.

2.1.5. DBX INTERFACE (PMC)

Name	Type	Description
DBX_ERR#	I LVTTL	**DBX ERROR:** Asserted by the DBX if an ECC or parity error occurred during a memory cycle. DBX_ERR# is asserted for 5 host clocks to indicate a Single-bit ECC error and 6 host clocks to indicate a parity or Multi-bit ECC error.
HLAD#	O LVTTL	**HOST LATCH AND ADVANCE:** During CPU reads (both from DRAM and PCI), this signal controls the latching of the read data into the DBX CPU interface output latch.
MLAD	O LVTTL	**MEMORY LATCH AND ADVANCE:** During DRAM reads, asserting this signal latches memory read data into the DBX. During DRAM writes, asserting this signal latches write data out of the DBX.
PC[8:0]	I/O LVTTL	**PMC CONTROL SIGNALS:** PC[8:0] are control signals between the PMC and DBX.
PD[15:0]	I/O LVTTL	**PRIVATE DATA BUS:** This is a 16 bit private data path between the PMC and DBX. This bus runs at the host clock rate and is used to transfer data during CPU-to-PCI cycles and PCI to DRAM cycles
DDRDY#	O LVTTL	**DELAYED DATA READY:** This delayed version of the DRDY# signal is asserted by the PMC to the DBX.

2.1.6. CLOCKS (PMC)

Name	Type	Description
HCLKIN	I 2.5V LVTTL	**HOST CLOCK IN:** This pin receives a host clock input from an external clock source. The input is configurable via the PD1 strap. If the PD1 is sampled low at reset(default), 3.3V buffer mode is enabled. This is normal operation enabled by internal pulldowns. If PD1 is sampled high, 2.5V buffer mode is enabled.
PCLKIN	I LVTTL	**PCI CLOCK IN:** This pin receives a PCI clock reference that is synchronous with respect to the host clock. This is the PCI clock reference that can be synchronously derived by an external clock synthesizer component from the host clock (divide-by-2). This signal clocks the PMC logic that is in the PCI clock domain.

2.1.7. MISCELLANEOUS (PMC)

Name	Type	Description
CRESET#	O LVTTL	**CHIP RESET:** This is a reset output signal driven by the PMC to the DBX. CRESET# is driven active for 2 msec. The DBX drives CPURST# to the CPUs, which is a 2 host clocks delayed version of the CRESET#. The PMC can also activate CRESET# under software control by writing to the internal reset configuration regsiter to initiate a hard reset or CPU BIST.
GTL_REFV	I	**GTL+ REFERENCE VOLTAGE:** This is the reference voltage derived from the termination voltage to the pullup resistors and determines the noise margin for the signals.

Name	Type	Description
PWROK	I LVTTL	**POWER OK:** This input goes active after all the power supplies in the system have reached their specified values. PWROK forces all of the PMC internal state machines to their default values. PWROK inactive generates CPURST# and PCIRST# active. The rising edge of PWROK is asynchronous, but must meet set-up and hold specifications for recognition on any specific clock. The PMC holds CPURST# for 2 msec and PCIRST# active for 1 msec after the rising edge of PWROK.

2.1.8. POWER UP STRAP OPTIONS (PMC)

Below is a list of all power on options that are loaded into the PMC based on the voltage level present on the respective strappings at the rising edge of PWROK. The PMC floats all signals connected to straps during CRESET# and keeps them floated for a minimum of 4 host clocks after the negation of CRESET#. To enable the different modes, external pullups should be approximately 10 KΩ to 3.3V (does not apply to A7#). Note that all signals that are used to select powerup strap options are connected to weak internal pulldowns.

Signal	Register Name/bit	Description
PC8	PMCCFG[14]	**Rows 7 And 8 Enable:** PC8 selects if RAS[7:6]#/ MAB[1:0] pins are used as row selects or extra copies of the lower two memory addresses. These are selected as follows: **PC8** — **RAS[7:6]/MAB[1:0]** 0 — MAB[1:0] 1 — RAS[7:6]#
PC[3:2]	PMCCFG[9:8]	**Host Frequency Select:** PC[3:2] selects the CPU bus frequency. **PC[3:2]** — **CPU Bus Frequency** 0 0 — Reserved 0 1 — 60 MHz 1 0 — 66 MHz 1 1 — Reserved
PD[15:12]		**Test Mode:** See Testability Section
PD1		**HCLKIN Input Buffer Select:** PD1 selects whether the 2.5V or 3.3V mode is enabled. **PC1** — **HCLKIN Input Buffer Select** 0 — 3.3V Input (Default) 1 — 2.5V Input
A7#	PMCCFG2	**In-order Queue Depth Select/Enable:** The value on A7# sampled on the rising edge of CRESET# reflects if the IOQD is set to 1 or maximum of four. Note that A7# is pulled up as a GTL+ signal and can be driven by to zero by external logic.

2.2. DBX Signals

2.2.1. DRAM INTERFACE SIGNALS (DBX)

Name	Type	Description
MD[63:0]	I/O LVTTL	**MEMORY DATA:** These signals are connected to the DRAM data bus and have weak internal pulldowns.
MPD[7:0]	I/O LVTTL	**MEMORY PARITY DATA:** These signals are connected to the parity or ECC bits of the DRAM data bus and have weak internal pulldowns.

2.2.2. PMC INTERFACE SIGNALS (DBX)

Name	Type	Description
DBX_ERR#	O LVTTL	**DBX ERROR:** DBX_ERR# is generated for ECC or parity errors during a memory read cycle. DBX_ERR# is asserted for 5 host clocks to indicate a Single-bit ECC error and 6 host clocks to indicate a parity or Multi-bit ECC error.
HLAD#	I LVTTL	**HOST LATCH AND ADVANCE SIGNAL:** During CPU reads, HLAD# controls the latching of read data into the DBX CPU interface output latch.
MLAD	I LVTTL	**MEMORY LATCH AND ADVANCE SIGNAL:** During DRAM reads, the PMC asserts this signal to latch memory read data into the DBX. During DRAM writes, the PMC asserts this signal to latch write data from the DBX.
PC[8:0]	I LVTTL	**PMC DBX CONTROL SIGNALS:** PC[8:0] are control signals between the PMC and DBX.
DDRDY#	I LVTTL	**DELAYED DATA READY:** The PMC asserts this delayed version of DRDY# to the DBX.
PD[15:0]	I/O LVTTL	**PRIVATE DATA BUS:** These signals are connected to the PD data bus on the PMC. This is the data path for the PCI-to-DRAM and CPU-to-PCI cycles. During PCI-to-DRAM reads and CPU-to-PCI writes, the DBX drives data on this bus. During CPU-to-PCI reads and PCI-to-DRAM writes, the DBX receives data on this bus.

2.2.3. HOST INTERFACE SIGNALS (DBX)

Name	Type	Description
HD[63:0]#	I/O GTL+	**HOST DATA:** These signals are connected to the CPU data bus. Note that the data signals are inverted on the CPU bus.
CPURST#	O GTL+	**CPU RESET:** The CPURST# pin is an output from the DBX that is driven directly from the CRESET#. It allows the CPUs to begin execution at a known state.

2.2.4. MISCELLANEOUS (DBX)

Name	Type	Description
HCLKIN	I 2.5V LVTTL	**HOST CLOCK IN:** This pin receives a host clock input from an external source. The input is configurable via the PD1 strap. If the PD1 is sampled low at reset (default), 3.3V buffer mode is enabled. This is normal operation enabled by internal pulldowns. If PD1 is sampled high, 2.5V buffer mode is enabled.
CRESET#	I LVTTL	**CHIP RESET:** This is a reset input signal driven by the PMC to the DBX. It forces the DBX to begin execution in a known state. This signal is also used to drive the CPURST# to the CPUs.
GTL_REFV	I	**GTL REFERENCE VOLTAGE:** This is the reference voltage derived from the termination voltage to the pullup resistors and determines the noise margin for the signals. This signal goes the reference input of the GTL+ sense amp on each GTL+ input or I/O pin.
BREQ0#	O GTL+	**SYMMETRIC AGENT BUS REQUEST:** Driven by the DBX during CPURST# to configure the symmetric bus agents.

2.2.5. POWER UP STRAP OPTIONS (DBX)

Below is a list of all power on options that are loaded into the DBX, based on the voltage level present on the respective strappings at the rising edge of CRESET#. To enable the different modes, external pullups should be approximately 10 KΩ to 3.3V. Note that all signals that are used to select powerup strap options are connected to weak internal pulldowns.

Signal	Register Name/bit	Description
PD[5:2]		**Test Mode:** See Testability Section
PD1		**HCLKIN Input Buffer Select:** PD1 selects whether the 2.5V or 3.3V mode is enabled. **PC1** **HCLKIN Input Buffer Select** 0 3.3V Input (Default) 1 2.5V Input

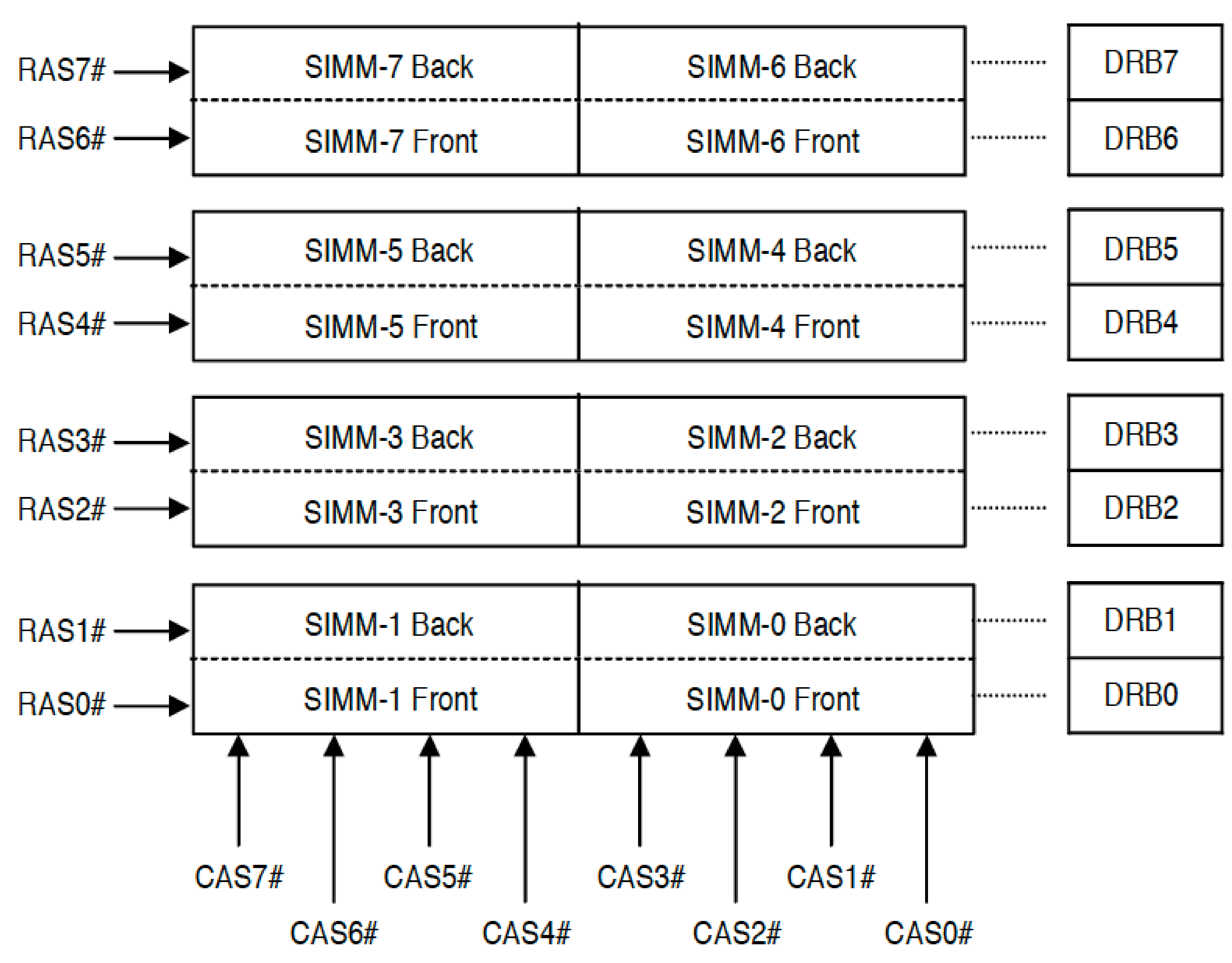


Figure 2. SIMMs and Corresponding DRB Registers

4.1.1. MEMORY ADDRESS RANGES

Figure 3 represents system memory address map. It shows the main memory regions defined and supported by the 440FX PCIset. At the highest level, the address space is divided into four conceptual regions (Figure 3). These are the 0–1-Mbyte DOS Compatibility Area, the 1-Mbyte to 16-Mbyte Extended Memory region used by ISA, the 16-Mbyte to 4-Gbyte Extended Memory region, and the 4-Gbyte to 64-Gbyte Extended Memory introduced by 36 bit addressing.

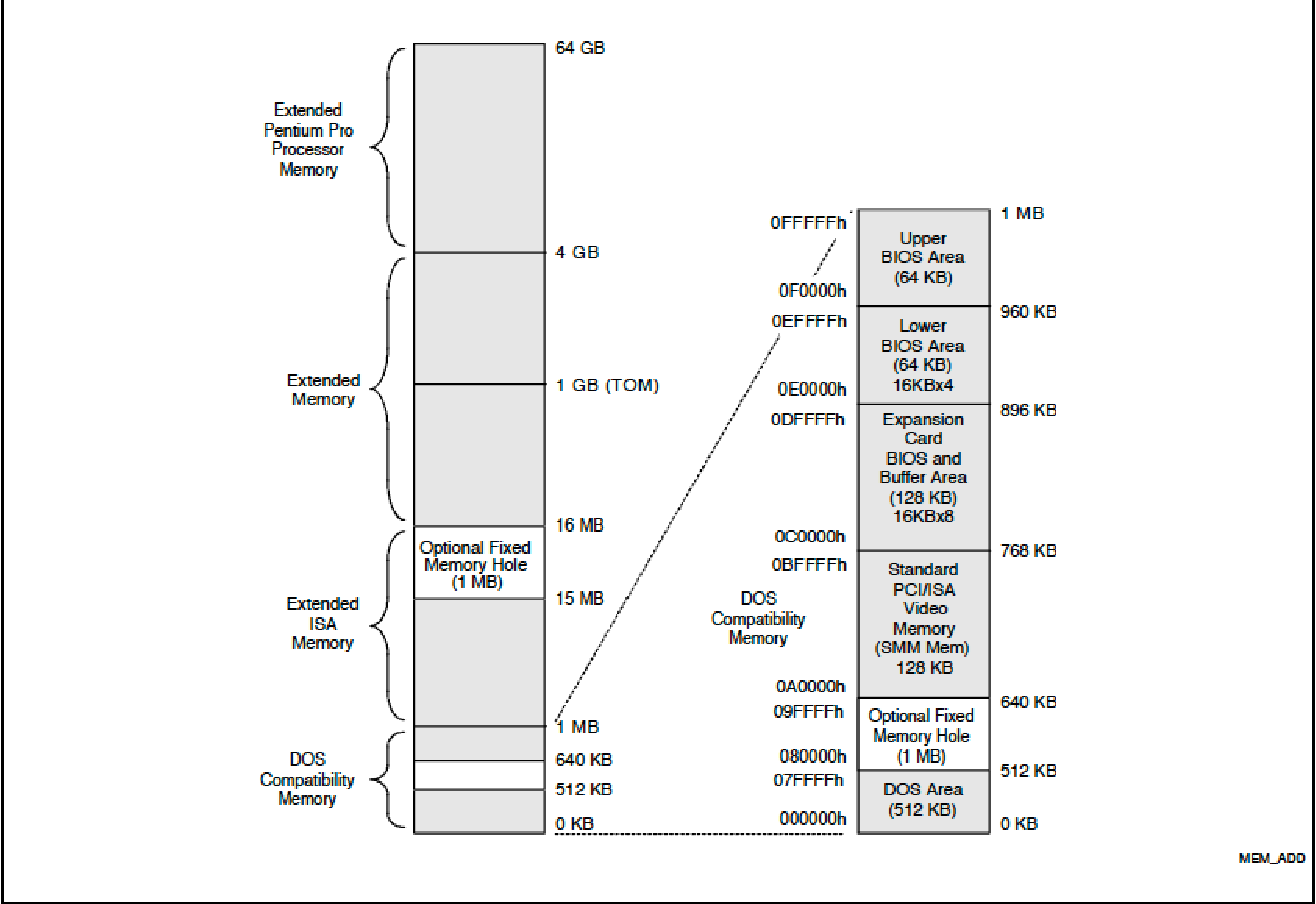


Figure 3. Memory Address Map

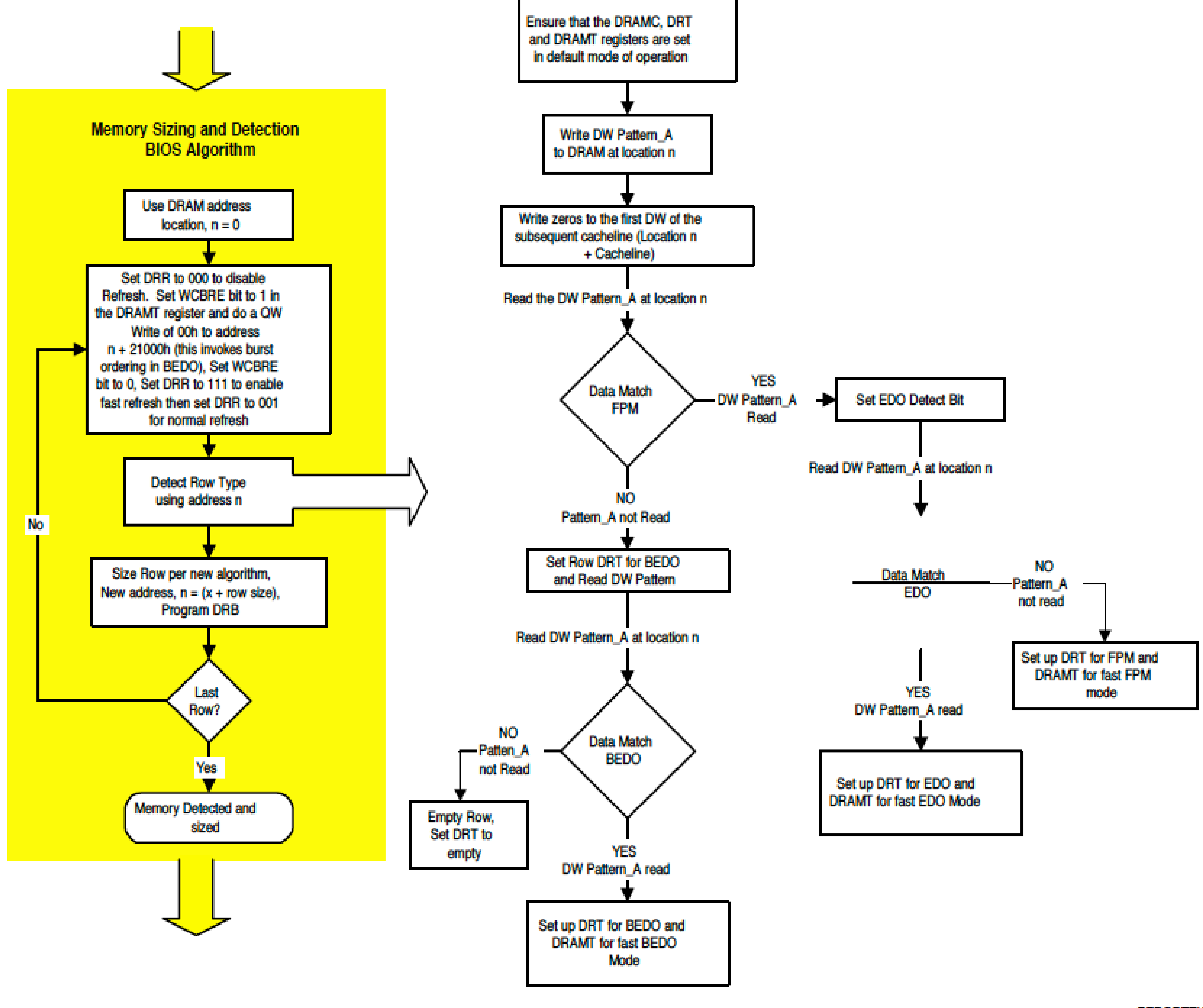


Figure 4. DRAM Auto-Detection Algorithm

Table 7. Memory Mapping Options

Memory Org.	Addressing	Address Size
4 Mb		
1M x 4	Symmetric	10 x 10
16 Mb		
1M x 16	Symmetric	10 x 10
2M x 8	Asymmetric	11 x 10
4M x 4	Symmetric	11 x 11
	Asymmetric	12 x 10
64 Mb		
4M x 16	Symmetric	11 x 11
	Asymmetric	12 x 10
8M x 8	Asymmetric	12 x 11
16M x 4	Symmetric	12 x 12

4.3.4. PSEUDO-ALGORITHM FOR DYNAMIC MEMORY SIZING

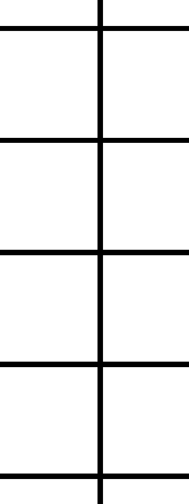
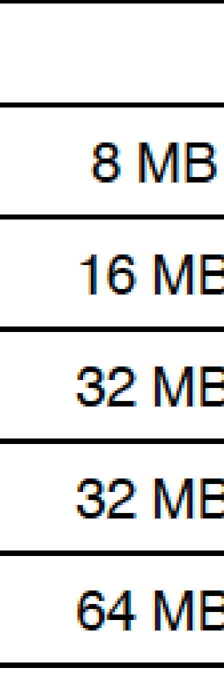
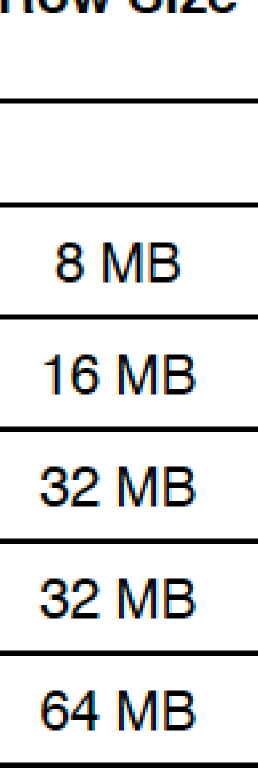
PMC implements asymmetrical addressing as described in Section 5.3 including support for 12 x 10 DRAM addressing. This section describes a pseudo-algorithm for calculating the memory sizing dynamically, including identification of memory addressing type. This pseudo-algorithm should be appropriately added to the algorithm used currently in the BIOS or the OS. A generic algorithm is described as follows:

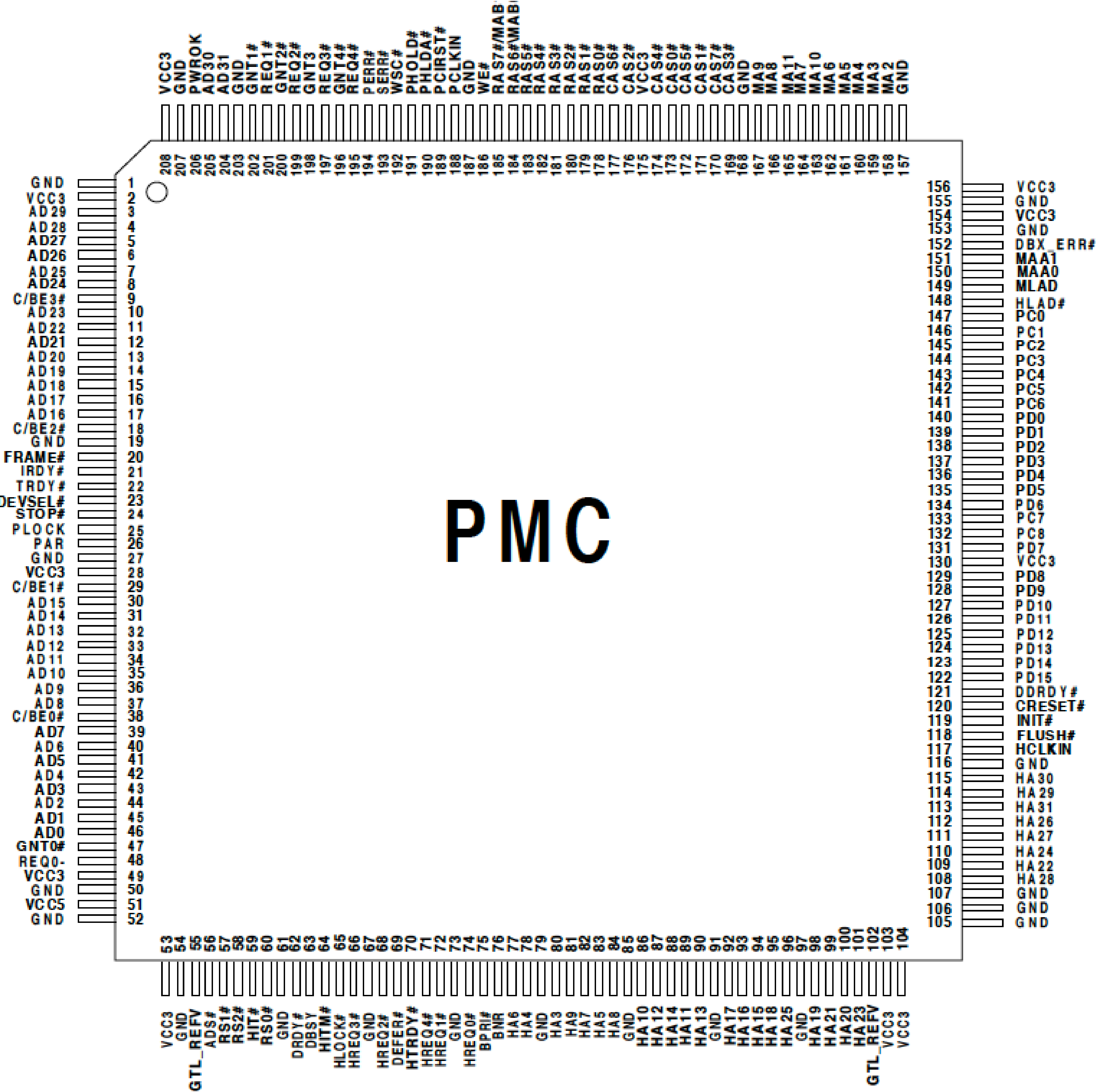
1. Configure row size for 128 MBytes (12 x 12 addressing) with base address = Baddr
2. Write a pattern 0Ch to location Baddr + 0000_0000h (encoding for 8 MB)
3. Write a pattern 04h to location Baddr + 0400_0000h (encoding for 16 MB)
4. Write a pattern 03h to location Baddr + 0200_0000h (encoding for 32 MB)
5. Write a pattern 01h to location Baddr + 0100_0000h (encoding for 64 MB)
6. Write a pattern 00h to location Baddr + 0080_0000h (encoding for 128 MB)
7. Read from locations Baddr + 0200_0000h into Register X
8. OR the value in register X with data from location Baddr + 0000_0000h into register X
9. Increment register X

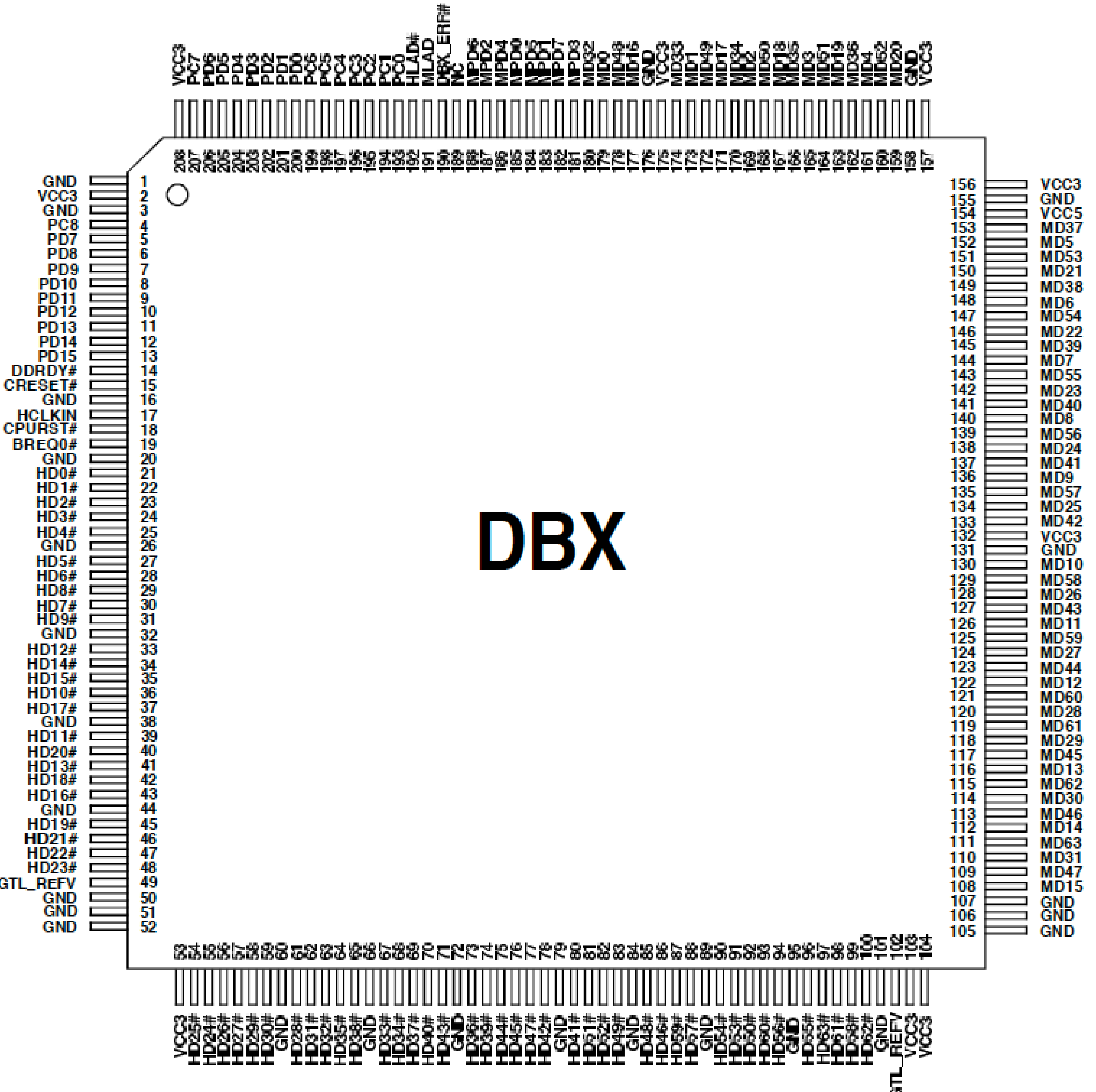
The result of this register X contains the correct value to add to the previous DRB register to get the correct value for the current DRB register. It is important to note that all the DRBs must be programmed to 128 MB until all the rows have been sized. The correct value of the row sizes should be programmed in all the DRBs after all the rows have been sized.

Table 8. Algorithm Results

Baddr + 0000_0000h	Baddr + 0200_0000h	Split	Register "X" at each Step			Row Size
			Step 7	Step 8	Step 9	
00h	00h	10 x 10	00h	00h	01h	8 MB
01h	01h	11 x 10	01h	01h	02h	16 MB
01h	01h	11 x 11	03h	03h	04h	32 MB
03h	03h	12 x 10	03h	03h	04h	32 MB
04h	03h	12 x 11	03h	07h	08h	64 MB
0Ch	0Ch	12 x 12	03h	0Fh	10h	128 MB


PMC
1 GND
2 VCC3
3 AD29
4 AD28
5 AD27
6 AD26
7 AD25
8 AD24
9 C/BE3#
10 AD23
11 AD22
12 AD21
13 AD20
14 AD19
15 AD18
16 AD17
17 AD16
18 C/BE2#
19 GND
20 FRAME#
21 IRDY#
22 TRDY#
23 DEVSEL#
24 STOP#
25 PLOCK
26 PAR
27 GND
28 VCC3
29 C/BE1#
30 AD15
31 AD14
32 AD13
33 AD12
34 AD11
35 AD10
36 AD9
37 AD8
38 C/BE0#
39 AD7
40 AD6
41 AD5
42 AD4
43 AD3
44 AD2
45 AD1
46 AD0
47 GNT0#
48 REQ0-
49 VCC3
50 GND
51 VCC5
52 GND
53 VCC3
54 GND
55 GTL_REFV
56 ADS#
57 RS1#
58 RS2#
59 HIT#
60 RS0#
61 GND
62 DRDY#
63 DBSY
64 HITM#
65 HLOCK#
66 HREQ3#
67 GND
68 HREQ2#
69 DEFER#
70 HTRDY#
71 HREQ4#
72 HREQ1#
73 GND
74 HREQ0#
75 BPRI#
76 BNR
77 HA6
78 HA4
79 GND
80 HA3
81 HA9
82 HA7
83 HA5
84 HA8
85 GND
86 HA10
87 HA12
88 HA14
89 HA11
90 HA13
91 GND
92 HA17
93 HA16
94 HA15
95 HA18
96 HA25
97 GND
98 HA19
99 HA21
100 HA20
101 HA23
102 GTL_REFV
103 VCC3
104 VCC3
105 GND
106 GND
107 GND
108 HA28
109 HA22
110 HA24
111 HA27
112 HA26
113 HA31
114 HA29
115 HA30
116 GND
117 HCLKIN
118 FLUSH#
119 INIT#
120 CRESET#
121 DDRDY#
122 PD15
123 PD14
124 PD13
125 PD12
126 PD11
127 PD10
128 PD9
129 PD8
130 VCC3
131 PD7
132 PC8
133 PC7
134 PD6
135 PD5
136 PD4
137 PD3
138 PD2
139 PD1
140 PD0
141 PC6
142 PC5
143 PC4
144 PC3
145 PC2
146 PC1
147 PC0
148 HLAD#
149 MLAD
150 MAA0
151 MAA1
152 DBX_ERR#
153 GND
154 VCC3
155 GND
156 VCC3
157 GND
158 MA2
159 MA3
160 MA4
161 MA5
162 MA6
163 MA10
164 MA7
165 MA11
166 MA8
167 MA9
168 GND
169 CAS3#
170 CAS7#
171 CAS1#
172 CAS5#
173 CAS0#
174 CAS4#
175 VCC3
176 CAS2#
177 CAS6#
178 RAS0#
179 RAS1#
180 RAS2#
181 RAS3#
182 RAS4#
183 RAS5#
184 RAS6#/MAB
185 RAS7#/MAB
186 WE#
187 GND
188 PCLKIN
189 PCIRST#
190 PHLDA#
191 PHOLD#
192 WSC#
193 SERR#
194 PERR#
195 REQ4#
196 GNT4#
197 REQ3#
198 GNT3
199 REQ2#
200 GNT2#
201 REQ1#
202 GNT1#
203 GND
204 AD31
205 AD30
206 PWROK
207 GND
208 VCC3



DBX
GND 1
VCC3 2
GND 3
PC8 4
PD7 5
PD8 6
PD9 7
PD10 8
PD11 9
PD12 10
PD13 11
PD14 12
PD15 13
DDRDY# 14
CRESET# 15
GND 16
HCLKIN 17
CPURST# 18
BREQ0# 19
GND 20
HD0# 21
HD1# 22
HD2# 23
HD3# 24
HD4# 25
GND 26
HD5# 27
HD6# 28
HD8# 29
HD7# 30
HD9# 31
GND 32
HD12# 33
HD14# 34
HD15# 35
HD10# 36
HD17# 37
GND 38
HD11# 39
HD20# 40
HD13# 41
HD18# 42
HD16# 43
GND 44
HD19# 45
HD21# 46
HD22# 47
HD23# 48
GTL_REFV 49
GND 50
GND 51
GND 52
53 VCC3
54 HD25#
55 HD24#
56 HD26#
57 HD27#
58 HD29#
59 HD30#
60 GND
61 HD28#
62 HD31#
63 HD32#
64 HD35#
65 HD38#
66 GND
67 HD33#
68 HD34#
69 HD37#
70 HD40#
71 HD43#
72 GND
73 HD36#
74 HD39#
75 HD44#
76 HD45#
77 HD47#
78 HD42#
79 GND
80 HD41#
81 HD51#
82 HD52#
83 HD49#
84 GND
85 HD48#
86 HD46#
87 HD59#
88 HD57#
89 GND
90 HD54#
91 HD53#
92 HD50#
93 HD60#
94 HD56#
95 GND
96 HD55#
97 HD63#
98 HD61#
99 HD58#
100 HD62#
101 GND
102 GTL_REFV
103 VCC3
104 VCC3
105 GND
106 GND
107 GND
108 MD15
109 MD47
110 MD31
111 MD63
112 MD14
113 MD46
114 MD30
115 MD62
116 MD13
117 MD45
118 MD29
119 MD61
120 MD28
121 MD60
122 MD12
123 MD44
124 MD27
125 MD59
126 MD11
127 MD43
128 MD26
129 MD58
130 MD10
131 GND
132 VCC3
133 MD42
134 MD25
135 MD57
136 MD9
137 MD41
138 MD24
139 MD56
140 MD8
141 MD40
142 MD23
143 MD55
144 MD7
145 MD39
146 MD22
147 MD54
148 MD6
149 MD38
150 MD21
151 MD53
152 MD5
153 MD37
154 VCC5
155 GND
156 VCC3
157 VCC3
158 GND
159 MD20
160 MD52
161 MD4
162 MD36
163 MD19
164 MD51
165 MD3
166 MD35
167 MD18
168 MD50
169 MD2
170 MD34
171 MD17
172 MD49
173 MD1
174 MD33
175 VCC3
176 GND
177 MD16
178 MD48
179 MD0
180 MD32
181 MPD3
182 MPD7
183 MPD1
184 MPD5
185 MPD0
186 MPD4
187 MPD2
188 MPD6
189 NC
190 DBX_ERR#
191 MLAD
192 HLAD#
193 PC0
194 PC1
195 PC2
196 PC3
197 PC4
198 PC5
199 PC6
200 PD0
201 PD1
202 PD2
203 PD3
204 PD4
205 PD5
206 PD6
207 PC7
208 VCC3

82371FB (PIIX) AND 82371SB (PIIX3) PCI ISA IDE XCELERATOR

- Bridge Between the PCI Bus and ISA Bus
- PCI and ISA Master/Slave Interface
  - PCI from 25–33 MHz
  - ISA from 7.5–8.33 MHz
  - 5 ISA Slots
- Fast IDE Interface
  - Supports PIO and Bus Master IDE
  - Supports up to Mode 4 Timings
  - Transfer Rates to 22 MB/Sec
  - 8 x 32-Bit Buffer for Bus Master IDE PCI Burst Transfers
  - Separate Master/Slave IDE Mode Support (PIIX3)
- Plug-n-Play Port for Motherboard Devices
  - 2 Steerable DMA Channels (PIIX Only)
  - Fast DMA with 4-Byte Buffer (PIIX Only)
  - 2 Steerable Interrupts Lines on the PIIX and 1 Steerable Interrupt Line on the PIIX3
  - 1 Programmable Chip Select
- Steerable PCI Interrupts for PCI Device Plug-n-Play
- PCI Specification Revision 2.1 Compliant (PIIX3)
- Functionality of One 82C54 Timer
  - System Timer; Refresh Request; Speaker Tone Output
- Two 82C59 Interrupt Controller Functions
  - 14 Interrupts Supported
  - Independently Programmable for Edge/Level Sensitivity
- Enhanced DMA Functions
  - Two 8237 DMA Controllers
  - Fast Type F DMA
  - Compatible DMA Transfers
  - 7 Independently Programmable Channels
- X-Bus Peripheral Support
  - Chip Select Decode
  - Controls Lower X-Bus Data Byte Transceiver
- I/O Advanced Programmable Interrupt Controller (IOAPIC) Support (PIIX3)
- Universal Serial Bus (USB) Host Controller (PIIX3)
  - Compatible with Universal Host Controller Interface (UHCI)
  - Contains Root Hub with 2 USB Ports
- System Power Management (Intel SMM Support)
  - Programmable System Management Interrupt (SMI)—Hardware Events, Software Events, EXTSMI#
  - Programmable CPU Clock Control (STPCLK#)
  - Fast On/Off Mode
- Non-Maskable Interrupts (NMI)
  - PCI System Error Reporting
- NAND Tree for Board-Level ATE Testing
- 208-Pin QFP

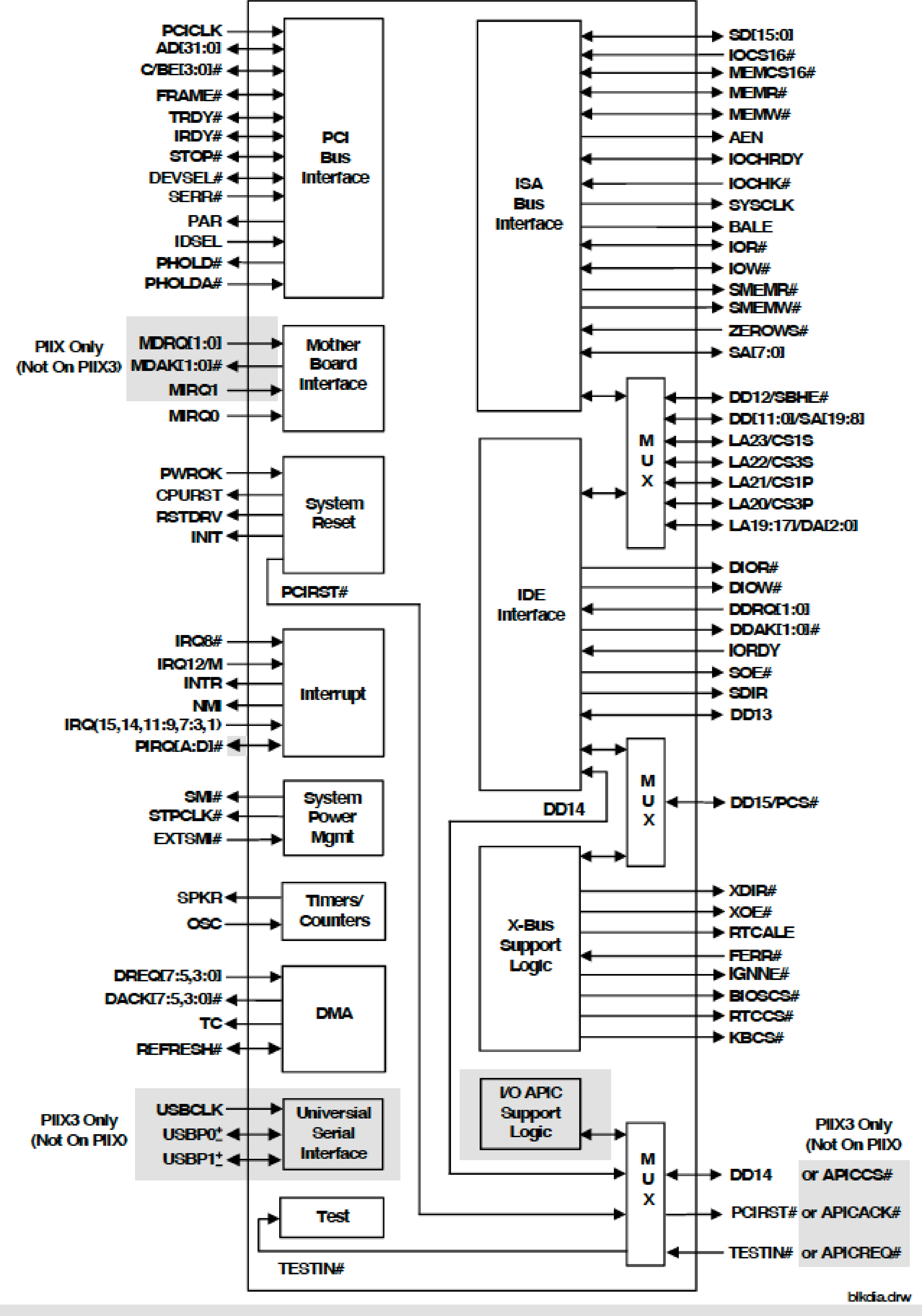
The 82371FB (PIIX) and 82371SB (PIIX3) PCI ISA IDE Xcelerators are multi-function PCI devices implementing a PCI-to-ISA bridge function and an PCI IDE function. In addition, the PIIX3 implements a Universal Serial Bus host/hub function. As a PCI-to-ISA bridge, the PIIX/PIIX3 integrates many common I/O functions found in ISA-based PC systems—a seven-channel DMA controller, two 82C59 interrupt controllers, an 8254 timer/counter, and power management support. In addition to compatible transfers, each DMA channel supports type F transfers. Chip select decoding is provided for BIOS, real time clock, and keyboard controller. Edge/Level interrupts and interrupt steering are supported for PCI plug and play compatibility. The PIIX/PIIX3 supports two IDE connectors for up to four IDE devices providing an interface for IDE hard disks and CD ROMs. The PIIX/PIIX3 provides motherboard plug and play compatibility. PIIX implements two steerable DMA channels (including type F transfers) and up to two steerable interrupt lines. PIIX3 implements one steerable interrupt line. The interrupt lines can be routed to any of the available ISA interrupts. Both PIIX/PIIX3 implement a programmable chip select.

PIIX3 contains a Universal Serial Bus (USB) Host Controller that is UHCI compatible. The Host Controller's root hub has two programmable USB ports. PIIX3 also provides support for an external IOAPIC.

This document describes the PIIX3 Component. Unshaded areas describe the 82371FB PIIX. Shaded areas, like this one, describe the PIIX3 operations that differ from the 82371FB PIIX.

© INTEL CORPORATION 1996, 1997
April 1997

Order Number: 290550-002



Note:

1. IOAPIC signals are multiplexed with signals from the System Reset, Test, and IDE Interface blocks
2. PIRQD# is an input on the PIIX and bi-directional on PIIX3.

PIIX/PIIX3 Simplified Block Diagram

1.0. SIGNAL DESCRIPTION

This section contains a detailed description of each signal. The signals are arranged in functional groups according to their interface.

Note that the '#' symbol at the end of a signal name indicates that the active, or asserted state occurs when the signal is at a low voltage level. When '#' is not present after the signal name, the signal is asserted when at the high voltage level.

The terms assertion and negation are used extensively. This is done to avoid confusion when working with a mixture of 'active-low' and 'active-high' signals. The term **assert**, or **assertion** indicates that a signal is active, independent of whether that level is represented by a high or low voltage. The term **negate**, or **negation** indicates that a signal is inactive.

Note that certain signal pins provide two separate functions. At the system level, these pins drive other signals with different functions through external buffers or transceivers. These pins have two different signal names depending on the function. These signal names have been noted in the signal description tables, with the signal whose function is being described in **bold** font. (For example, LA23/**CS1S** is in the section describing CS1S and **LA23**/CS1S is in the section describing LA23).

The following notations are used to describe the signal type.

I *Input* is a standard input-only signal.
O *Totem Pole Output* is a standard active driver.
I/O *Input/Output* is a bi-directional, tri-state signal.
od *Open Drain* allows multiple devices to share as a wire-OR.
st Schmitt Trigger input.
t/s *Tri-State* is a bi-directional, tri-state input/output pin.
s/t/s *Sustained Tri-state* is an active low tri-state signal owned and driven by one and only one agent at a time. The agent that drives a s/t/s pin low must drive it high for at least one clock before letting it float. A new agent can not start driving a s/t/s signal any sooner than one clock after the previous owner tri-states it. An external pull-up is required to sustain the inactive state until another agent drives it and must be provided by the central resource.

1.1. PCI Interface Signals

Signal Name	Type	Description
PCICLK	I	**PCI CLOCK:** PCICLK provides timing for all transactions on the PCI Bus. All other PCI signals are sampled on the rising edge of PCICLK, and all timing parameters are defined with respect to this edge. PCI frequencies of 25–33 MHz are supported.
AD[31:0]	I/O	**PCI ADDRESS/DATA:** The standard PCI address and data lines. The address is driven with FRAME# assertion and data is driven or received in following clocks
C/BE[3:0]#	I/O	**BUS COMMAND AND BYTE ENABLES:** The command is driven with FRAME# assertion. Byte enables corresponding to supplied or requested data is driven on following clocks.
FRAME#	I/O (s/t/s)	**FRAME:** Assertion indicates the address phase of a PCI transfer. Negation indicates that one more data transfer is desired by the cycle initiator.

Signal Name	Type	Description
TRDY#	I/O (s/t/s)	**TARGET READY:** Asserted when the target is ready for a data transfer.
IRDY#	I/O (s/t/s)	**INITIATOR READY:** Asserted when the initiator is ready for a data transfer.
STOP#	I/O (s/t/s)	**STOP:** Asserted by the target to request the master to stop the current transaction.
IDSEL	I	**INITIALIZATION DEVICE SELECT:** IDSEL is used as a chip select during configuration read and write transactions.
DEVSEL#	I/O (s/t/s)	**DEVICE SELECT:** The PIIX/PIIX3 asserts DEVSEL# to claim a PCI transaction through positive or subtractive decoding.
PAR	O	**CALCULATED PARITY SIGNAL:** PAR is "even" parity and is calculated on 36 bits—AD[31:0] plus C/BE[3:0]#.
SERR#	I	**SYSTEM ERROR:** SERR# can be pulsed active by any PCI device that detects a system error condition. Upon sampling SERR# active, the PIIX/PIIX3 can be programmed to generate a non-maskable interrupt (NMI) to the CPU.
PHOLD#	O	**PCI HOLD:** The PIIX/PIIX3 asserts this signal to request the PCI Bus. The PIIX3 implements the passive release mechanism by toggling PHOLD# inactive for one PCICLK.
PHLDA#	I	**PCI HOLD ACKNOWLEDGE:** This signal is asserted to grant the PCI bus to the PIIX/PIIX3.

1.2. Motherboard I/O Device Interface Signals

Signal Name	Type	Description
MDRQ[1:0] (PIIX Only)	I	**MOTHERBOARD DEVICE DMA REQUEST:** These signals can be connected internally to any of DREQ[3:0,7:5]. Each pair of request/ acknowledge signals is controlled by a separate register. Each signal can be configured as steerable interrupts for motherboard devices.
MDAK[1:0]# (PIIX Only)	O	**MOTHERBOARD DEVICE DMA ACKNOWLEDGE:** These signals can be connected internally to any of DACK[3:0,7:5]. Each pair of request/ acknowledge signals is controlled by a separate register. Each signal can be configured as steerable interrupts for motherboard devices.

Signal Name	Type	Description
MIRQ0/IRQ0 (PIIX3 Only) MIRQ[1:0] (PIIX Only)	I/O I	**MOTHERBOARD DEVICE INTERRUPT REQUEST**: The MIRQx signals can be internally connected to interrupts IRQ[15,14,12:9,7:3]. Each MIRQx line has a separate Route Control Register. If MIRQx and PIRQx# are steered to the same ISA interrupt, the device connected to the MIRQx should produce active high, level interrupts. The MIRQ0/IRQ0 signal has two functions (for PIIX3 only), depending on the programming of the IRQ0 Enable bit (MIRQ0 Register). In the systems that include the PIIX3 and IOAPIC, the MIRQ0/IRQ0 pin will function as the IRQ0 output and should be connected to the INTIN2 input of the IOAPIC. The interrupt from the Secondary IDE Channel should be connected to the IRQ15 input on PIIX3 and to the INTIN15 input on the IOAPIC. In the systems that include the PIIX3 only, the interrupt from the Secondary IDE Channel should be connected to the MIRQ0/IRQ0 input. If an MIRQ line is steered to a given IRQ input to the internal 8259, the corresponding ISA IRQ is masked, unless the Route Control register is programmed to allow the interrupts to be shared. This should only be done if the device connected to the MIRQ line and the device connected to the ISA IRQ line both produce active high, level interrupts. MIRQ0 can be configured as an output to connect the internal IRQ0 signal to an external IO-APIC.

1.3. IDE Interface Signals

Signal Name	Type	Description
DD[15:0]/ PCS#, SBHE#, SA[19:8] APICCS# (PIIX3)	I/O O I/O I/O O	**DISK DATA**: These signals directly drive the corresponding signals on up to two IDE connectors (primary and secondary). In addition, these signals are buffered (using 2xALS245's on the motherboard) to produce the SA[19:8], PCS#, and SBHE# signals (see separate descriptions). For the PIIX3, DD14 is buffered to produce APICCS#
DIOR#	O	**DISK I/O READ**: This signal directly drives the corresponding signal on up to two IDE connectors (primary and secondary).
DIOW#	O	**DISK I/O WRITE**: This signal directly drives the corresponding signal on up to two IDE connectors (primary and secondary).
DDRQ[1:0]	I	**DISK DMA REQUEST**: These input signals are directly driven from the DRQ signals on the primary (DDRQ0) and secondary (DDRQ1) IDE connectors. They are used in conjunction with the PCI Bus master IDE function and are not associated with any ISA-Compatible DMA channel.
DDAK[1:0]#	O	**DISK DMA ACKNOWLEDGE**: These signals directly drive the DAK# signals on the primary (DDAK0#) and secondary (DDAK1#) IDE connectors. These signals are used in conjunction with the PCI Bus master IDE function and are not associated with any ISA-Compatible DMA channel.

Signal Name	Type	Description
IORDY	I	**IO CHANNEL READY**: This input signal is directly driven by the corresponding signal on up to two IDE connectors (primary and secondary).
SOE#	O	**SYSTEM ADDRESS TRANSCEIVER OUTPUT ENABLE**: This signal controls the output enables of the '245 transceivers that interface the DD[15:0] signals to the SA[19:8], SBHE#, PCS# and APICCS# (PIIX3 only) signals.
SDIR	O	**SYSTEM ADDRESS TRANSCEIVER DIRECTION**: This signal controls the direction of the '245 transceivers that interface the DD[15:0] signals to the SA[19:8], SBHE#, PCIS, and APICCS# (PIIX3 only), signals. Default condition is high (transmit). When an ISA Bus master is granted use of the bus, the transceivers are turned around to drive the ISA address [19:8] on DD[15:3]. The address can then be latched by the PIIX/PIIX3. In this case, the SDIR signal is low (receive). The SOE# and SDIR signals taken together as a group can assume one of three states: **SOE# SDIR State** 0 1 PCI to ISA transaction 1 1 PCI to IDE 0 0 ISA Bus master

Signals Buffered from LA[23:17]

These signals are buffered from the LA[23:17] lines by an ALS244 tri-state buffer. The output enable of this buffer is tied asserted. These signals are set up with respect to the IDE command strobes (DIOR# and IOW#) and are valid throughout I/O transactions targeting the ATA register block(s).

Signal Name	Type	Description
LA23/ **CS1S**	I/O	**CHIP SELECT:** CS1S is for the ATA command register block and corresponds to the inverted CS1FX# on the secondary IDE connector. CS1S is inverted externally (see PCI Local Bus IDE section).
LA22/ **CS3S**	I/O	**CHIP SELECT:** CS3S is for the ATA control register block and corresponds to the inverted CS3FX# on the secondary IDE connector. CS3S is inverted externally (see PCI Local Bus IDE section).
LA21/ **CS1P**	I/O	**CHIP SELECT:** CS1P is for the ATA command register block and corresponds to the inverted CS1FX# on the primary IDE connector. CS1P is inverted externally (see PCI Local Bus IDE section).
LA20/ **CS3P**	I/O	**CHIP SELECT:** CS3P is for the ATA control register block and corresponds to the inverted CS3FX# on the primary IDE connector. CS3P is inverted externally (see PCI Local Bus IDE section).
LA[19:17] **DA[2:0]**	I/O	**DISK ADDRESS:** DA[2:0] are used to indicate which byte in either the ATA command block or control block is being addressed.

1.4. ISA Interface Signals

Signal Name	Type	Description
BALE	O	**BUS ADDRESS LATCH ENABLE:** BALE is an active high signal asserted by the PIIX/PIIX3 to indicate that the address (SA[19:0], LA[23:17]) and SBHE# signal lines are valid.
AEN	O	**ADDRESS ENABLE:** AEN is asserted during DMA cycles to prevent I/O slaves from misinterpreting DMA cycles as valid I/O cycles. This signal is also driven high during PIIX/PIIX3 initiated refresh cycles. For the PIIX, when TC is sampled low on the assertion of PWORK (External DMA mode), the PIIX tri-states this signal.
SYSCLK	O	**ISA SYSTEM CLOCK:** SYSCLK is the reference clock for the ISA Bus and drives the bus directly. SYSCLK is generated by dividing PCICLK by 3 or 4. The SYSCLK frequencies supported are 7.5 MHz and 8.33 MHz. SYSCLK is a divided down version of PCICLK. **Hardware Strapping Option** SYSCLK is tri-stated when PWROK is negated. The value of SYSCLK is sampled on the assertion of PWROK: If sampled high, the ISA clock divisor is 3 (for 25 MHz PCI). If sampled low, the divisor is 4 (for 30 and 33 MHz PCI).
IOCHRDY	I/O	**I/O CHANNEL READY:** Resources on the ISA Bus negate IOCHRDY to indicate that additional time (wait states) is required to complete the cycle. This signal is normally high on the ISA Bus. IOCHRDY is an input when the PIIX/PIIX3 owns the ISA Bus and the CPU or a PCI agent is accessing an ISA slave or during DMA transfers. IOCHRDY is output when an external ISA Bus Master owns the ISA Bus and is accessing DRAM or a PIIX/PIIX3 register.
IOCS16#	I	**16-BIT I/O CHIP SELECT:** This signal is driven by I/O devices on the ISA Bus to indicate that they support 16-bit I/O bus cycles.
IOCHK#	I	**I/O CHANNEL CHECK:** IOCHK# can be driven by any resource on the ISA Bus. When asserted, it indicates that a parity or an uncorrectable error has occurred for a device or memory on the ISA Bus. If enabled, a NMI is generated to the CPU.
IOR#	I/O	**I/O READ:** IOR# is the command to an ISA I/O slave device that the slave may drive data on to the ISA data bus (SD[15:0]).
IOW#	I/O	**I/O WRITE:** IOW# is the command to an ISA I/O slave device that the slave may latch data from the ISA data bus (SD[15:0]).
LA[23:17]/ CS1S CS3S CS1P CS3P DA[2:0]	**I/O/** O O O O O	**UNLATCHED ADDRESS:** The LA[23:17] address lines are bi-directional. These address lines allow accesses to physical memory on the ISA Bus up to 16 Mbytes. The LA[23:17] are also used to drive the IDE interface chip selects and address lines via an external ALS244 buffer. See the IDE interface signal descriptions.

Signal Name	Type	Description
SA[7:0], **SA[19:8]/** DD[11:0]	I/O **I/O** I/O	**SYSTEM ADDRESS BUS:** These bi-directional address lines define the selection with the granularity of one byte within the one-Mbyte section of memory defined by the LA[23:17] address lines. The address lines SA[19:17] that are coincident with LA[19:17] are defined to have the same values as LA[19:17] for all memory cycles. For I/O accesses, only SA[15:0] are used.
SBHE#/ DD12	**I/O** I/O	**SYSTEM BYTE HIGH ENABLE:** SBHE# indicates, when asserted, that a byte is being transferred on the upper byte (SD[15:8]) of the data bus. SBHE# is negated during refresh cycles.
MEMCS16#	od	**MEMORY CHIP SELECT 16:** MEMCS16# is a decode of LA[23:17] without any qualification of the command signal lines. ISA slaves that are 16-bit memory devices drive this signal low. The PIIX/PIIX3 drives this signal low during ISA master to DRAM Cycles.
MEMR#	I/O	**MEMORY READ:** MEMR# is the command to a memory slave that it may drive data onto the ISA data bus. This signal is also driven by the PIIX/PIIX3 during refresh cycles.
MEMW#	I/O	**MEMORY WRITE:** MEMW# is the command to a memory slave that it may latch data from the ISA data bus.
SMEMR#	O	**STANDARD MEMORY READ:** The PIIX/PIIX3 asserts SMEMR# to request an ISA memory slave to drive data onto the data lines. If the access is below 1 Mbyte (00000000–000FFFFFh) during DMA compatible, PIIX/PIIX3 master, or ISA master cycles, the PIIX/PIIX3 asserts SMEMR#. SMEMR# is a delayed version of MEMR#.
SMEMW#	O	**STANDARD MEMORY WRITE:** The PIIX/PIIX3 asserts SMEMW# to request an ISA memory slave to accept data from the data lines. If the access is below 1 Mbyte (00000000–000FFFFFh) during DMA compatible, PIIX/PIIX3 master, or ISA master cycles, the PIIX/PIIX3 asserts SMEMW#. SMEMW# is a delayed version of MEMW#.
ZEROWS#	I	**ZERO WAIT-STATES:** An ISA slave asserts ZEROWS# after its address and command signals have been decoded to indicate that the current cycle can be shortened. A 16-bit ISA memory cycle can be reduced to two SYSCLKs. An 8-bit memory or I/O cycle can be reduced to three SYSCLKs. ZEROWS# has no effect during 16-bit I/O cycles.
SD[15:0]	I/O	**SYSTEM DATA:** SD[15:0] provide the 16-bit data path for devices residing on the ISA Bus. SD[15:8] correspond to the high order byte and SD[7:0] correspond to the low order byte. SD[15:0] are undefined during refresh.

1.5. DMA Signals

Signal Name	Type	Description
DREQ [7:5,3:0]	I	**DMA REQUEST:** The DREQ lines are used to request DMA service from the PIIX/PIIX3 's DMA controller or for a 16-bit master to gain control of the ISA expansion bus. The active level (high or low) is programmed via the DMA Command Register. The request must remain active until the appropriate DACKx# signal is asserted.
DACK [7:5,3:0]#	O	**DMA ACKNOWLEDGE:** The DACK output lines indicate that a request for DMA service has been granted by the PIIX/PIIX3 or that a 16-bit master has been granted the bus. The active level (high or low) is programmed via the DMA Command Register. These lines should be used to decode the DMA slave device with the IOR# or IOW# line to indicate selection. If used to signal acceptance of a bus master request, this signal indicates when it is legal to assert MASTER#. For the PIIX, when TC is sampled low on the assertion of PWORK (External DMA mode), the PIIX tri-states these signals. This mode is not available on PIIX3.
TC	O	**TERMINAL COUNT:** The PIIX/PIIX3 asserts TC to DMA slaves as a terminal count indicator. When all the DMA channels are not in use, TC is negated (low). **Hardware Strapping Option (PIIX Only)** This strapping option selects between the internal ISA DMA mode and External DMA mode. When TC is sampled high on the assertion of PWROK (ISA DMA mode), the PIIX drives the AEN, TC, and DACK#[7:5, 3:0] normally. When TC is sampled low on the assertion of PWROK (External DMA mode), the PIIX tri-states the AEN, TC, and DACK[7:5, 3:0]# signals, and also forwards PCI masters I/O accesses to location 0000h to ISA. TC has an internal pull-up resistor. To tie TC low, an external 1 KΩ pull-down resistor should be used. For the PIIX3, this signal should not be pulled down.
REFRESH#	I/O	**REFRESH:** As an output, REFRESH# indicates when a refresh cycle is in progress. It should be used to enable the SA[7:0] address to the row address inputs of all banks of dynamic memory on the ISA Bus. Thus, when MEMR# is asserted, the entire expansion bus dynamic memory is refreshed. Memory slaves must not drive any data onto the bus during refresh. As an output, this signal is driven directly onto the ISA Bus. This signal is an output only when the PIIX/PIIX3 DMA refresh controller is a master on the bus responding to an internally generated request for refresh. As an input, REFRESH# is driven by 16-bit ISA Bus masters to initiate refresh cycles.

1.6. Timer/Counter Signals

Signal Name	Type	Description
SPKR	O	**SPEAKER DRIVE:** The SPKR signal is the output of counter 2.
OSC	I	**OSCILLATOR:** OSC is the 14.31818 MHz ISA clock signal. It is used by the internal 8254 Timer.

1.7. Interrupt Controller Signals

Signal Name	Type	Description
IRQ[15,14, 11:9, 7:3,1]	I	**INTERRUPT REQUEST:** The IRQ signals provide both system board components and ISA Bus I/O devices with a mechanism for asynchronously interrupting the CPU. The assertion mode of these inputs depends on the programming of the two ELCR registers. The IRQ14 signal must be used by the Bus Master IDE interface function to signal interrupts on the primary IDE channel.
IRQ8#	I	**INTERRUPT REQUEST EIGHT SIGNAL:** IRQ8# is always an active low edge triggered interrupt input (i.e., this interrupt can not be modified by software). Upon PCIRST#, IRQ8# is placed in active low edge sensitive mode.
IRQ12/M	I	**INTERRUPT REQUEST/MOUSE INTERRUPT:** In addition to providing the standard interrupt function (see IRQ[15,14,11:9,7:3,1] signal description), this pin can be programmed (via X-Bus Chip Select Register) to provide a mouse interrupt function.
PIRQ[D:A]#	I I/O For PIRQD# (PIIX3 only)	**PROGRAMMABLE INTERRUPT REQUEST:** The PIRQx# signals can be shared with interrupts IRQ[15,14,12:9,7:3] as described in the Interrupt Steering section. Each PIRQx# line has a separate Route Control Register. These signals require external pull-up resisters. For the PIIX3, the USB interrupt is output on PIRQD#.
INTR	od	**CPU INTERRUPT:** INTR is driven by the PIIX/PIIX3 to signal the CPU that an interrupt request is pending and needs to be serviced. The interrupt controller must be programmed following PCIRST# to ensure that INTR is at a known state.
NMI	od	**NON-MASKABLE INTERRUPT:** NMI is used to force a non-maskable interrupt to the CPU. The PIIX/PIIX3 generates an NMI when either SERR# or IOCHK# is asserted, depending on how the NMI Status and Control Register is programmed.

1.8. System Power Management (SMM) Signals

Signal Name	Type	Description
SMI#	od	**SYSTEM MANAGEMENT INTERRUPT:** SMI# is an active low synchronous output that is asserted by the PIIX/PIIX3 in response to one of many enabled hardware or software events.
STPCLK#	od	**STOP CLOCK:** STPCLK# is an active low synchronous output that is asserted by the PIIX/PIIX3 in response to one of many hardware or software events. STPCLK# connects directly to the CPU and is synchronous to PCICLK.
EXTSMI#	I	**EXTERNAL SYSTEM MANAGEMENT INTERRUPT:** EXTSMI# is a falling edge triggered input to the PIIX/PIIX3 indicating that an external device is requesting the system to enter SMM mode. This signal contains a weak internal pullup.

1.9. X-Bus Signals

Signal Name	Type	Description
XDIR#	O	**X-BUS DIRECTION:** XDIR# is tied directly to the direction control of a 74F245 that buffers the X-Bus data (XD[7:0]). XDIR# is asserted for all I/O read cycles, regardless if the accesses are to a PIIX/PIIX3 supported device. XDIR# is only asserted for memory cycles if BIOS sapce (PIIX and PIIX3) or APIC space (PIIX3 only) has been decoded. For PCI master inititated read cycles, XDIR# is asserted from the falling edge of either IOR# or MEMR# (from MEMR# only if BIOS space (PIIX and PIIX3) or APIC (PIIX3 only) space has been decoded), depending on the cycle type. For ISA master-initiated read cycles, XDIR# is asserted from the falling edge of either IOR# or MEMR# (from MEMR# only if BIOS space has been decoded), depending on the cycle type. When the rising edge of IOR# or MEMR# occurs, the PIIX/PIIX3 negates XDIR#. For DMA read cycles from the X-Bus, XDIR# is asserted from DACKx# falling and negated from DACKx# rising. At all other times, XDIR# is negated.
XOE#	O	**X-BUS OUTPUT ENABLE:** XOE# is tied directly to the output enable of a 74F245 that buffers the X-Bus data (XD[7:0]) from the system data bus (SD[7:0]). XOE# is asserted when a PIIX/PIIX3 supported X-Bus device is decoded, and the devices decode is enabled in the X-Bus Chip Select Enable Register (XBCS Register). XOE# is asserted from the falling edge of the ISA commands (IOR#, IOW#, MEMR#, or MEMW#) for PCI Master and ISA master-initiated cycles. XOE# is negated from the rising edge of the ISA command signals for CPU and PCI Master-initiated cycles and the SA[16:0] and LA[23:17] address for ISA master-initiated cycles. XOE# is not generated during any access to an X-Bus peripheral in which its decode space has been disabled.
DD15/ **PCS#**	O	**PROGRAMMABLE CHIP SELECT:** PCS# is asserted for ISA I/O cycles that are generated by PCI masters and subtractively decoded to ISA, if the access hits the address range programmed into the PCSC Register. The X-Bus buffer signals are enabled when the chip select is asserted (i.e., it is assumed that the peripheral that is selected via this pin resides on the X-Bus).
BIOSCS#	O	**BIOS CHIP SELECT:** BIOSCS# is asserted during read or write accesses to BIOS. BIOSCS# is driven combinatorially from the ISA addresses SA[16:0] and LA [23:17], except during DMA. During DMA cycles, BIOSCS# is not generated.
KBCS#	O	**KEYBOARD CONTROLLER CHIP SELECT:** KBCS# is asserted during I/O read or write accesses to KBC locations 60h and 64h. This signal is driven combinatorially from the ISA addresses SA[16:0] and LA [23:17]. For DMA cycles, KBCS# is never asserted.
RTCCS#	O	**REAL TIME CLOCK CHIP SELECT:** RTCCS# is asserted during read or write accesses to RTC location 71h. RTCCS# can be tied to a pair of external OR gates to generate the real time clock read and write command signals.
RTCALE	O	**REAL TIME CLOCK ADDRESS LATCH:** RTCALE is used to latch the appropriate memory address into the RTC. A write to port 70h with the appropriate RTC memory address that will be written to or read from, causes RTCALE to be asserted. RTCALE is asserted based on IOW# falling and remains asserted for two SYSCLKs.

Signal Name	Type	Description
FERR#	I	**NUMERIC COPROCESSOR ERROR:** This signal is tied to the coprocessor error signal on the CPU. IGNNE# is only used if the PIIX/PIIX3 coprocessor error reporting function is enabled in the XBCSA Register. If FERR# is asserted, the PIIX/PIIX3 generates an internal IRQ13 to its interrupt controller unit. The PIIX/PIIX3 then asserts the INTR output to the CPU. FERR# is also used to gate the IGNNE# signal to ensure that IGNNE# is not asserted to the CPU unless FERR# is active. FERR# has a weak internal pull-up used to ensure a high level when the coprocessor error function is disabled.
IGNNE#	od	**IGNORE ERROR:** This signal is connected to the ignore error pin on the CPU. IGNNE# is only used if the PIIX/PIIX3 coprocessor error reporting function is enabled in the XBCSA Register. If FERR# is asserted, indicating a coprocessor error, a write to the Coprocessor Error Register (F0h) causes the IGNNE# to be asserted. IGNNE# remains asserted until FERR# is negated. If FERR# is not asserted when the Coprocessor Error Register is written, the IGNNE# signal is not asserted.

1.10. APIC Bus Signals (PIIX3 Only)

Signal Name	Type	Description
DD14/ **APICCS#**	I/O O	**APIC CHIP SELECT (PIIX3 only).** This active low output signal is asserted when the APIC Chip Select is enabled and a PCI originated cycle is positively decoded within the programmed IOAPIC address space. The default addresses of the IOAPIC are Memory FEC0_0000h and FEC0_0010h. System Design Note: The DD[14]/APICCS# signal is demuxed externally with a 245 transceiver. The output of the transceiver drives the IOAPIC's CS# signal. At certain times the transceiver floats its outputs, therefore a pullup resistor on the output of the tranceiver is required to keep this signal negated.
TESTIN#/ **APICREQ#**	I	**APIC REQUEST (PIIX3 only).** This signal has two functions, depending on the programming of the APIC Chip Select bit (XBCS Register). See the Test Signal Description for the TESTIN# function. APICREQ# is asserted by an external APIC device prior to sending an interrupt over the APIC serial bus. When the PIIX3 samples this pin active it flushes its F-type DMA buffers pointing towards PCI. Once the buffers are flushed, the PIIX3 asserts APICACK# to inform the external APIC that it can proceed to send the APIC interrupt. APICREQ# must be synchronous to PCICLK.
PCIRST#/ **APICACK#**	O O	**APIC ACKNOWLEDGE (PIIX3 only).** This signal has two functions, depending on the programming of the APIC Chip Select bit (XBCS Register). See the System Reset Signal Description for the PCIRST# function. The PIIX3 asserts APICACK# after its internal buffers are flushed in response to the APICREQ# signal. When the IOAPIC samples this signal asserted it knows that the PIIX3's buffers are flushed and that it can proceed to send the APIC interrupt. The signal is driven from the rising edge of PCICLK and is negated while PCIRST# is asserted.

1.11. Universal Serial Bus Signals (PIIX3 Only)

Signal Name	Type	Description
USBCLK	I	**UNIVERSAL SERIAL BUS CLOCK.** This signal clocks the universial serial bus clock.
USBP0+ USBP0-	I/O	**UNIVERSAL SERIAL BUS PORT 0.** These signals are the differential data pair for Serial Port 0.
USBP1+ USBP1-	I/O	**UNIVERSIAL SERIAL BUS PORT 1.** These signals are the differential data pair for Serial Port 1.

1.12. System Reset Signals

Signal Name	Type	Description
PWROK	I	**POWER OK:** When asserted, PWROK is an indication to the PIIX/PIIX3 that power and PCICLK have been stable for at least 1 ms. PWROK can be driven asynchronously. When PWROK is negated, the PIIX/PIIX3 asserts CPURST, PCIRST# and RSTDRV. When PWROK is asserted, the PIIX/PIIX3 negates CPURST, PCIRST#, and RSTDRV.
CPURST	od	**CPU RESET:** The PIIX/PIIX3 asserts CPURST to reset the CPU. The PIIX/PIIX3 asserts CPURST during power-up and when a hard reset sequence is initiated through the RC register. CPURST is driven synchronously to the rising edge of PCICLK. If a hard reset is initiated through the RC register, the PIIX/PIIX3 resets it's internal registers to the default state.
PCIRST#/ APICACK# (PIIX3 Only)	O O	**PCI RESET:** This signal has two functions, depending on the programming of the APIC Chip Select bit (XBCS Register). See the APIC SIgnal Description for the APICACK# function. The PIIX/PIIX3 asserts PCIRST# to reset devices that reside on the PCI Bus. The PIIX/PIIX3 asserts PCIRST# during power-up and when a hard reset sequence is initiated through the RC register. PCIRST# is driven inactive a minimum of 1 ms after PWROK is driven active. PCIRST# is driven active for a minimum of 1ms when initiated through the RC register. PCIRST# is driven asynchronously relative to PCICLK.
INIT	OD	**INITIALIZATION:** The PIIX/PIIX3 asserts INIT if it detects a shut down special cycle on the PCI Bus or if a soft reset is initiated via the RC Register.
RSTDRV	O	**RESET DRIVE:** The PIIX/PIIX3 asserts this signal during a hard reset and during power-up to reset ISA Bus devices. RSTDRV is also asserted for a minimum of 1 ms if a hard reset has been programmed in the RC Register.

1.13. Test Signals

Signal Name	Type	Description
TESTIN#/ APICREQ# (PIIX3 Only)	I I	**TEST INPUT:** This signal has two functions, depending on the programming of the APIC Chip Select bit (XBCS Register). See the APIC SIgnal Description for the APICREQ# function. The TESTIN# signal is used in conjunction with the IRQ signals to select the various test modes of the PIIX/PIIX3. This input contains an internal pull up resistor. After a hard reset, this pin functions as a TESTIN# signal. An external weak pull-up resistor (4.7k to 20k ohms) is required to 5V.

1.14. Power and Ground Signals

Signal Name	Type	Description
VCC		**Power (5 Volts):** This pin is connected to the 5 volt power supply.
VCC3 (PIIX3)		**Power (3.3 Volts):** This pin is connected to the 3.3 volt power supply. Note that, if the the Universal Serial Bus function is not used, this pin can be connected to the 5 volt power supply.
GND		**Ground:** This pin is connected to the ground plane.

3.3. ISA Interface

The PIIX/PIIX3 incorporates a fully ISA Bus compatible master and slave interface. The PIIX/PIIX3 directly drives five ISA slots without external data buffers. External transceivers are used on the SA[19:8] and SBHE# signals to permit these signals to be used with the IDE interface (Figure 1). The ISA interface also provides byte swap logic, I/O recovery support, wait state generation, and SYSCLK generation.

The ISA interface supports the following types of cycles:

- PCI master-initiated I/O and memory cycles to the ISA Bus
- DMA compatible cycles between main memory and ISA I/O and between ISA I/O and ISA memory
- Enhanced DMA cycles between PCI memory and ISA I/O (for motherboard devices only)
- ISA refresh cycles initiated by either the PIIX/PIIX3 or an external ISA master
- ISA master-initiated memory cycles to PCI and ISA master-initiated I/O cycles to the internal PIIX/PIIX3 registers, as shown in ISA-Compatible Register table in the Register Description section.

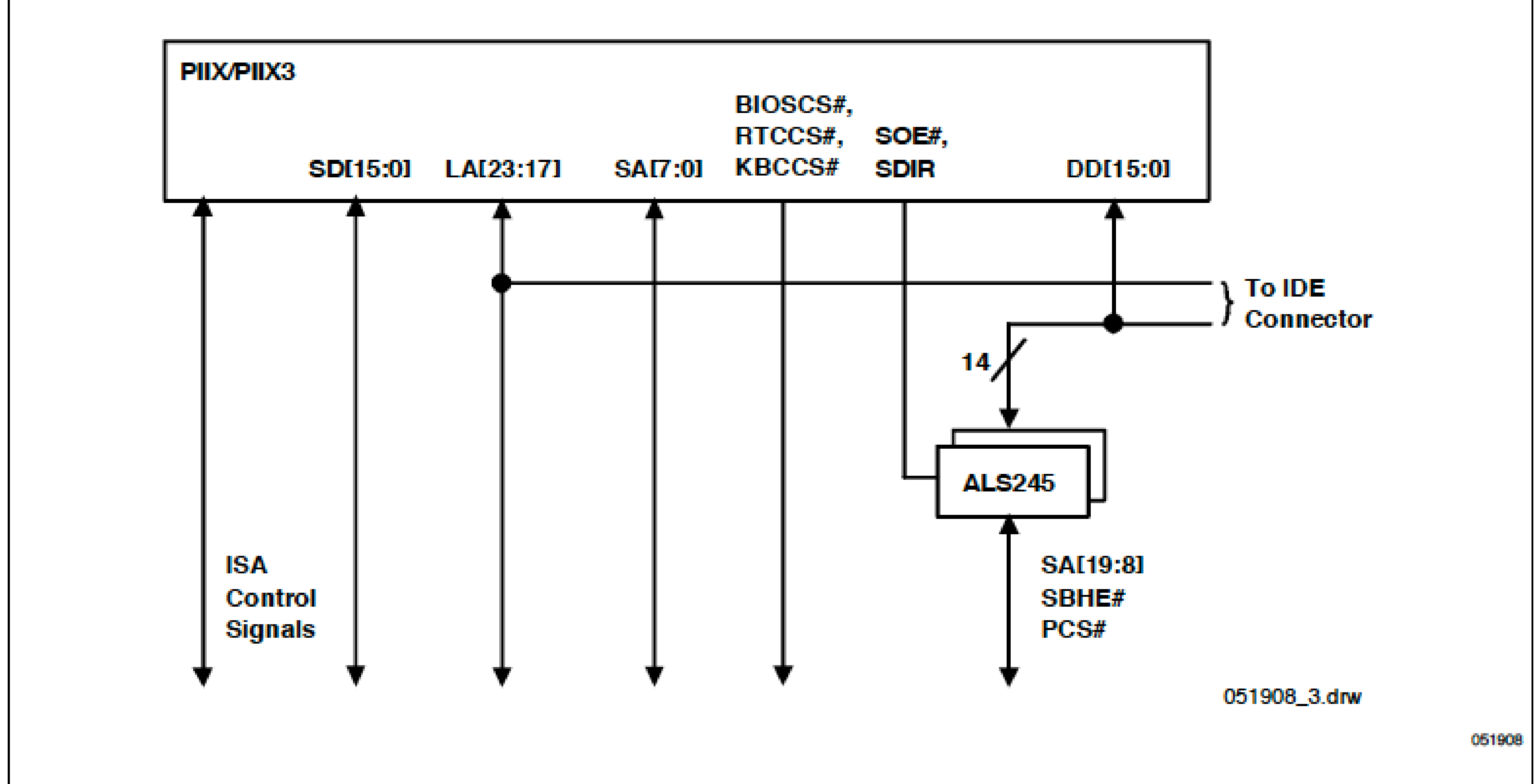


Figure 1. ISA Interface

3.4. DMA Controller

The DMA circuitry incorporates the functionality of two 82C37 DMA controllers with seven independently programmable channels (Channels 0-3 and Channels 5-7). DMA Channel 4 is used to cascade the two controllers and defaults to cascade mode in the DMA Channel Mode (DCM) Register. In addition to accepting requests from DMA slaves, the DMA controller also responds to requests that are initiated by software. Software may initiate a DMA service request by setting any bit in the DMA Channel Request Register to a 1. The DMA controller for Channels 0-3 is referred to as "DMA-1" and the controller for Channels 4-7 is referred to as "DMA-2".

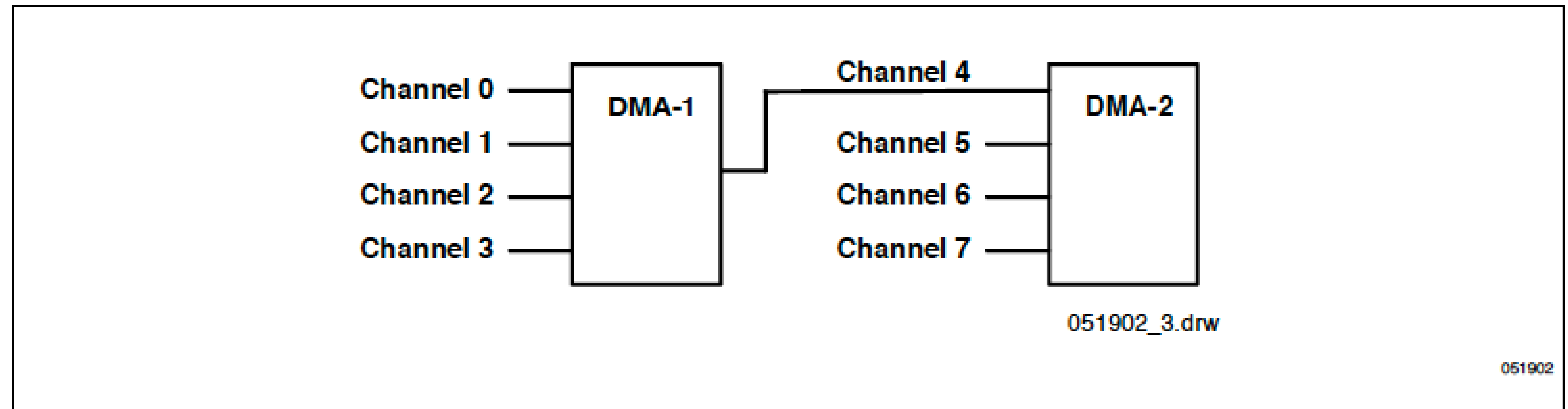


Figure 2. Internal DMA Controller

Each DMA channel is hardwired to the compatible settings for DMA device size; channels [3:0] are hardwired to 8-bit count-by-bytes transfers and channels [7:5] are hardwired to 16-bit count-by-words (address shifted) transfers. The PIIX/PIIX3 provides the timing control and data size translation necessary for the DMA transfer between the memory (ISA or main memory) and the ISA Bus device. ISA Compatible and F type DMA timing are supported. Type F DMA is selected via the MBDMA[1:0] Registers and permits up to two channels to be programmed for type F transfers at the same time.

The PIIX/PIIX3 provides 24-bit addressing in compliance with the ISA-Compatible specification. Each channel includes a 16-bit ISA-Compatible Current Register that contains the 16 least-significant bits of the 24-bit address, an ISA Compatible Page Register that contains the eight next most significant bits of address. The DMA controller also features refresh address generation, and auto-initialization following a DMA termination.

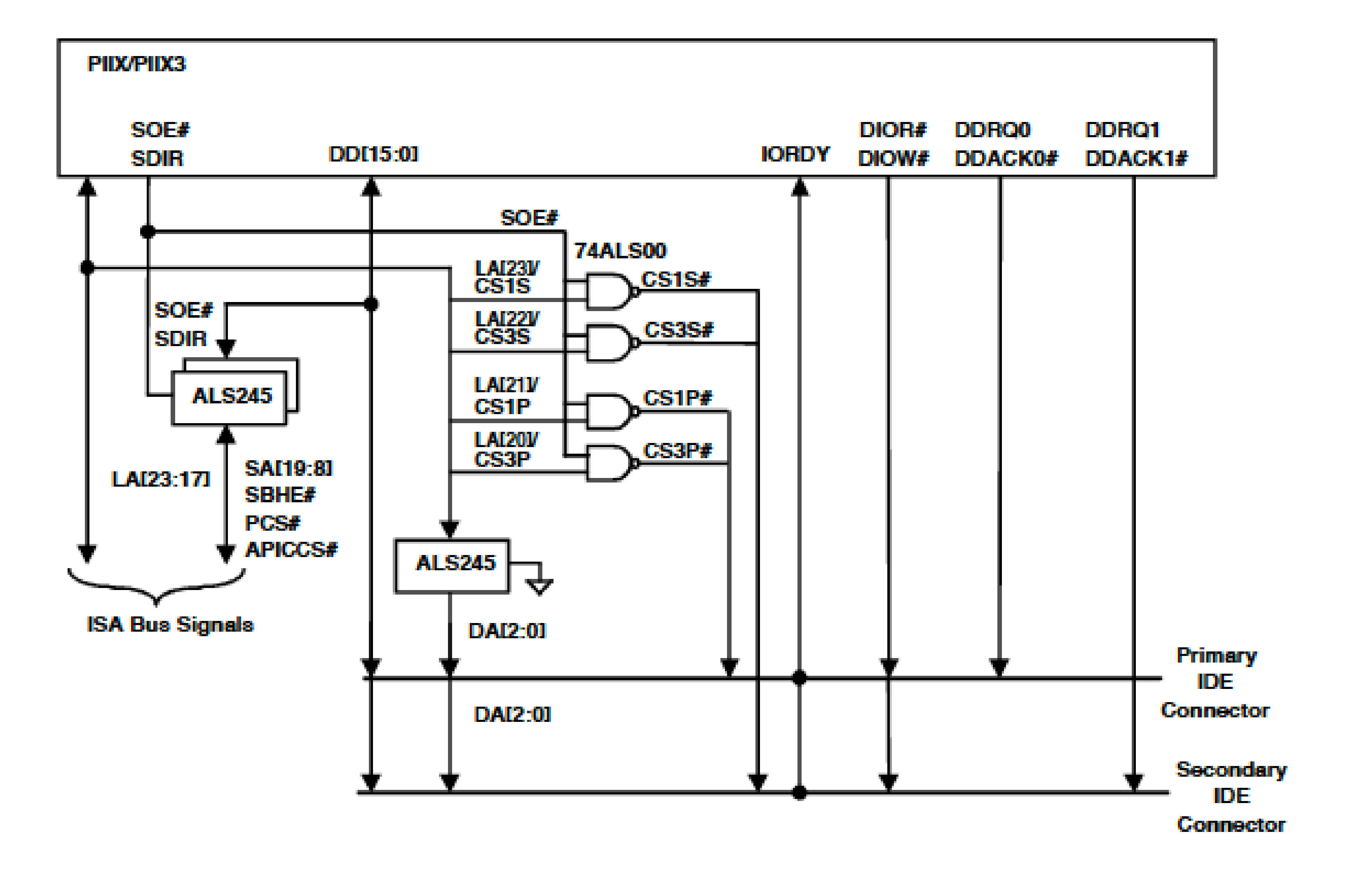
The DMA controller is either in master or slave mode. In master mode, the DMA controller is either servicing a DMA slave's request for DMA cycles or allowing a 16-bit ISA master to use the bus (via a cascaded DREQ signal). In slave mode, the PIIX/PIIX3 monitors both the ISA Bus and PCI, decoding and responding to I/O read and write commands that address its registers.

3.5. PCI Local Bus IDE

The PIIX/PIIX3 integrates a high performance interface from PCI to IDE. This interface is capable of accelerated PIO data transfers as well as acting as a PCI Bus master on behalf of an IDE DMA slave device. The PIIX/PIIX3 provides an interface for both primary and secondary IDE connectors (Figure 3).

The IDE data transfer command strobes, DMA request and grant signals, and IORDY signal interface directly to the PIIX. The IDE data lines interface directly to the PIIX, and are buffered to provide part of the ISA address bus as well as the X-Bus chip select signals. The IDE address and chip select signals are multiplexed onto the LA[23:17] lines. The IDE connector signals are driven from the LA[23:17] lines by an ALS244 buffer.

Only PCI masters have access to the IDE port. ISA Bus masters cannot access the IDE I/O port addresses. Memory targeted by the IDE interface acting as a PCI Bus master on behalf of IDE DMA slaves must reside on PCI, usually main memory implemented by the host-to-PCI bridge.



051903

NOTES:

Support for Older Drives: There are cases where the PIIX/PIIX3 asserts both IDE chip selects (CS1x and CS3x). Some older drives may not operate properly when both chip selects are asserted. Because the IDE chip selects are muxed with the ISA LA lines, the 74ALS00 in the figure is used to ensure proper operation of older drives by gating the LA signals with SOE#.

Figure 3. PIIX/PIIX3 IDE Interface

Two connectors (primary and secondary) and two drives per connector (master and slave) are supported as shown in Figure 4.

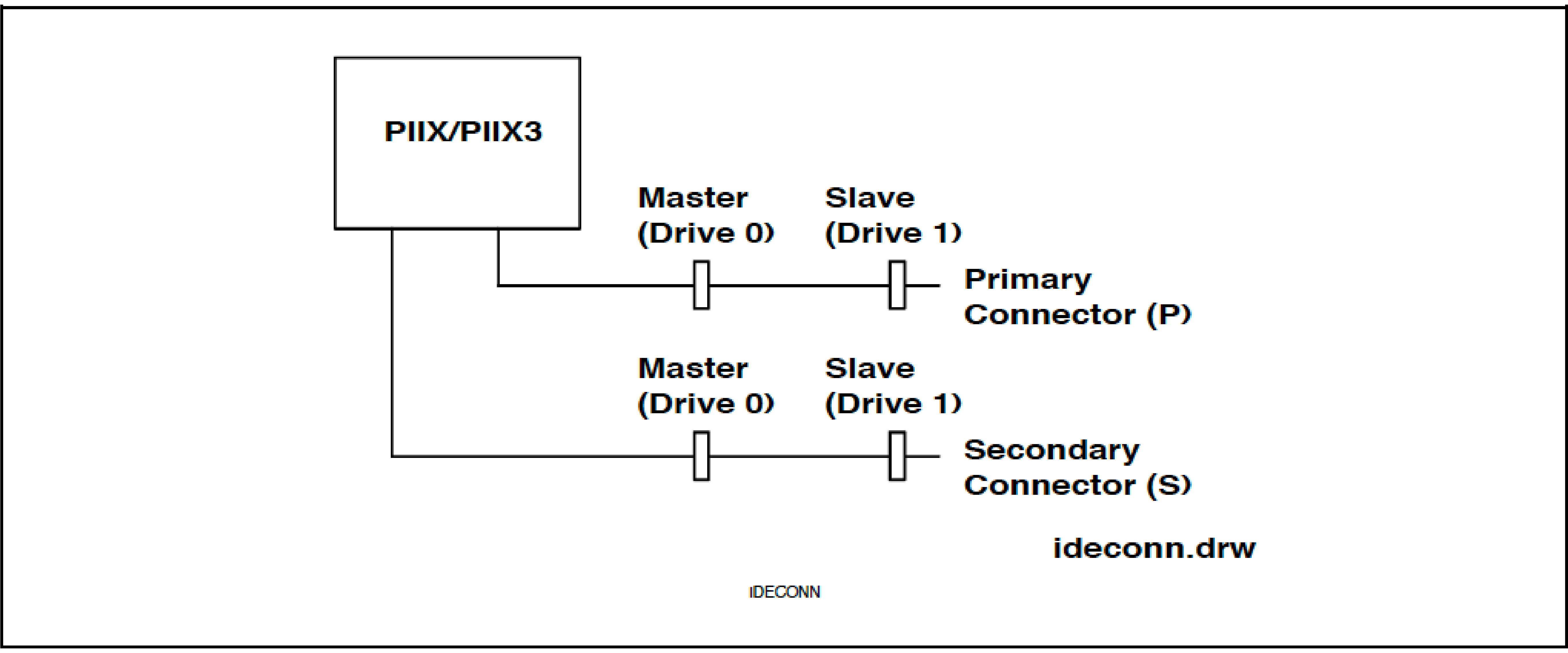


Figure 4. IDE Connector and Drive Nomenclature

3.7. Interval Timer

The PIIX/PIIX3 contains three counters that are equivalent to those found in the 82C54 programmable interval timer. The three counters are contained in one PIIX/PIIX3 timer unit, referred to as Timer-1. Each counter output provides a key system function. Counter 0 is connected to interrupt controller IRQ0 and provides a system timer interrupt for a time-of-day, diskette time-out, or other system timing functions. Counter 1 generates a refresh request signal and Counter 2 generates the tone for the speaker. The 14.31818 MHz counters normally use OSC as a clock source.

Counter 0, System Timer

This counter functions as the system timer by controlling the state of IRQ0 and is typically programmed for mode 3 operation. The counter produces a square wave with a period equal to the product of the counter period (838 ns) and the initial count value. The counter loads the initial count value one counter period after software writes the count value to the counter I/O address. The counter initially asserts IRQ0 and decrements the count value by two each counter period. The counter negates IRQ0 when the count value reaches 0. It then reloads the initial count value and again decrements the initial count value by two each counter period. The counter then asserts IRQ0 when the count value reaches 0, reloads the initial count value, and repeats the cycle, alternately asserting and negating IRQ0.

Counter 1, Refresh Request Signal

This counter provides the refresh request signal and is typically programmed for mode 2 operation. The counter negates refresh request for one counter period (838 ns) during each count cycle. The initial count value is loaded one counter period after being written to the counter I/O address. The counter initially asserts refresh request, and negates it for 1 counter period when the count value reaches 1. The counter then asserts refresh request and continues counting from the initial count value.

Counter 2, Speaker Tone

This counter provides the speaker tone and is typically programmed for mode 3 operation. The counter provides a speaker frequency equal to the counter clock frequency (1.193 MHz) divided by the initial count value. The speaker must be enabled by a write to port 061h.

3.8. Interrupt Controller

The PIIX/PIIX3 provides an ISA compatible interrupt controller that incorporates the functionality of two 82C59 interrupt controllers. The two controllers are cascaded so that 13 external and three internal interrupts are possible. The master interrupt controller provides IRQ [7:0] and the slave interrupt controller provides IRQ [15:8] (Figure 7). The three internal interrupts are used for internal functions only and are not available to the user. IRQ2 is used to cascade the two controllers together. IRQ0 is used as a system timer interrupt and is tied to Interval Timer 1, Counter 0. The MIRQ0/IRQ0 pin will function as the IRQ0 output and should be connected to the INTIN2 input of the IOAPIC when IRQ0 enable bit is set in the MIRQ0 register. IRQ13 is connected internally to FERR#. The remaining 13 interrupt lines (IRQ[15,14,12:3,1]) are available for external system interrupts. Edge or level sense selection is programmable on an individual channel by channel basis.

The Interrupt unit also supports interrupt steering. The PIIX/PIIX3 can be programmed to allow the four PCI active low interrupts (PIRQ[D:A]#) to be internally routed to one of 11 interrupts (IRQ[15,14,12:9,7:3]). In addition, the motherboard interrupts (MIRQ[1:0] for PIIX and MIRQ0 for PIIX3) may be routed to any of the 11 interrupts.

The Interrupt Controller consists of two separate 82C59 cores. Interrupt Controller 1 (CNTRL-1) and Interrupt Controller 2 (CNTRL-2) are initialized separately and can be programmed to operate in different modes. The default settings are: 80x86 Mode, Edge Sensitive (IRQ[15:0]) Detection, Normal EOI, Non-Buffered Mode, Special Fully Nested Mode disabled, and Cascade Mode. CNTRL-1 is connected as the Master Interrupt Controller and CNTRL-2 is connected as the Slave Interrupt Controller.

Note that IRQ13 is generated internally (as part of the coprocessor error support) by the PIIX. IRQ12/M is generated internally (as part of the mouse support) when bit-4 in the XBCS Register is set to a 1. When this bit is set to a 0, the standard IRQ12 function is provided and IRQ12 appears externally.

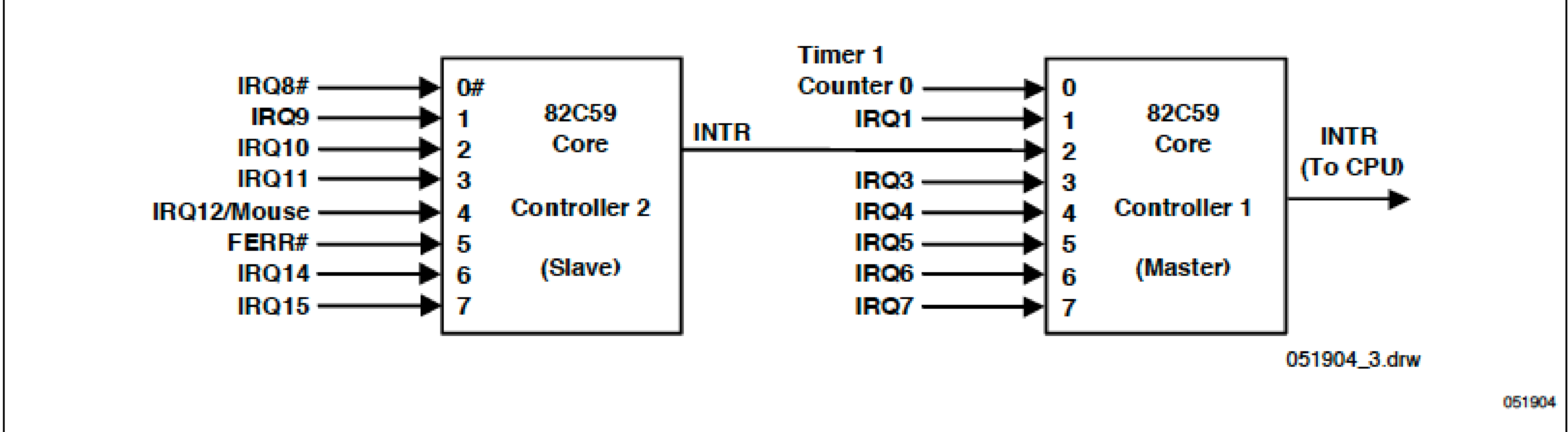


Figure 7. Block Diagram of the Interrupt Controller

3.9. Stand-Alone IOAPIC Support (PIIX3)

The PIIX3 supports a stand-alone IOAPIC device on the ISA X-Bus. The PIIX3 provides a chip select signal (APICCS#) for the IOAPIC. It also provides handshake signals to maintain buffer coherency in the IOAPIC environment.

APICCS# is generated when the PCI memory cycle address matches the APIC's programmed address and the APICCS# function is enabled in the XBCS Register. The APIC address can be relocated by programming the APIC Base Address Register (APICBASE).

APICCS# is only generated for PCI originated cycles and is not generated for ISA originated cycles. This PCI cycle is forwarded to the ISA bus. To avoid address aliasing conflicts with other ISA devices, PIIX3 drives SA[19:16] and LA[23:17] to 0 and drives SA[15:0] corresponding to PCI AD[15:2] and C/BE[3:0]#.

When the APICCS# function is enabled, the XOE#/XDIR# signals controlling the X-bus tranceiver and the SOE#/SDIR signals controlling the IDE DD isolation transceiver are also enabled during accesses to the IOAPIC.

The IOAPIC signals (APICCS#, APICREQ#, and APICACK#) are multiplexed with DD14, TESTIN#, and PCIRST#, respectively. Figure 9 shows how these signals are connected in systems with and without the IOAPIC device.

The internal IRQ0 signal can be routed to the external pin MIRQ0 using bit 5 in the MBIRQ Route Control Register 0. This changes MIRQ0 to an output signal and allows the IRQ0 signal to be connected to the external IO-APIC. The secondary IDE device interrupt should then be routed to IRQ15.

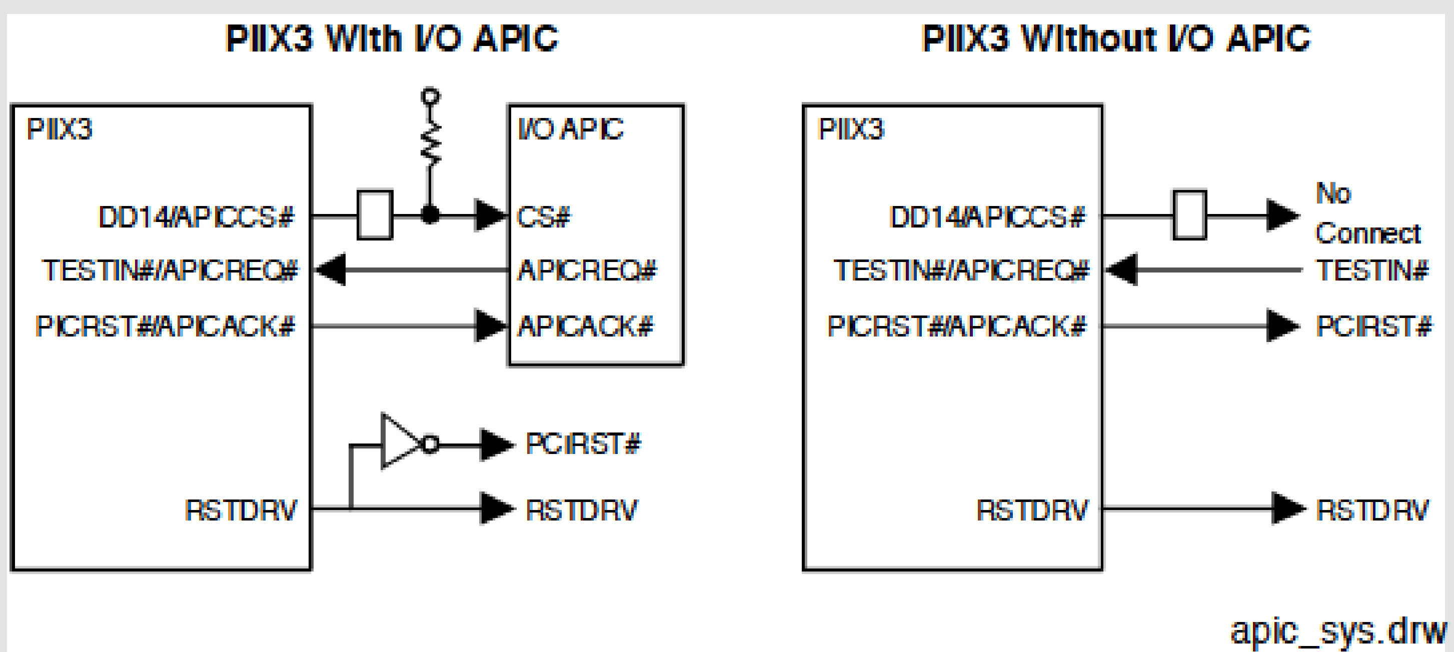
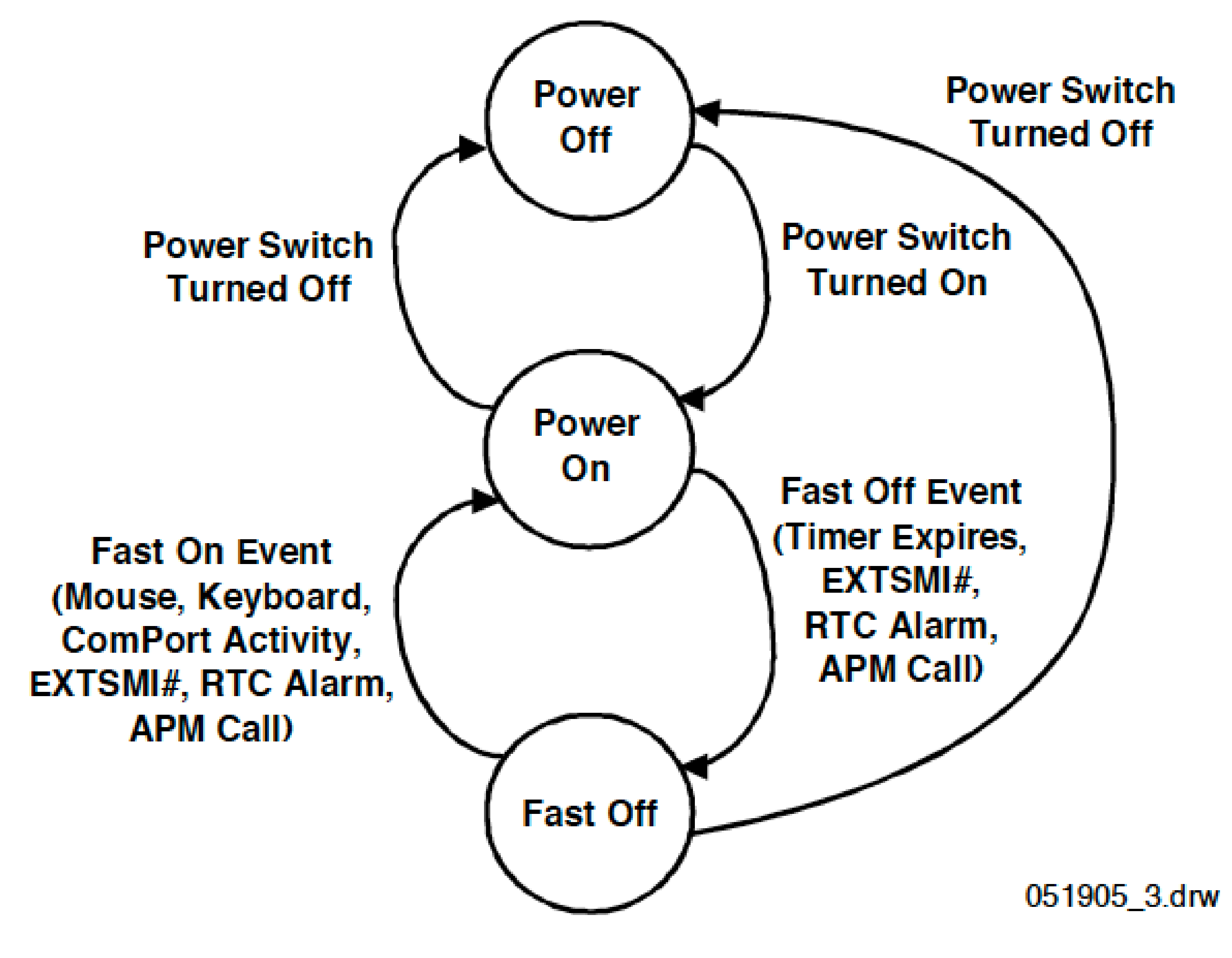
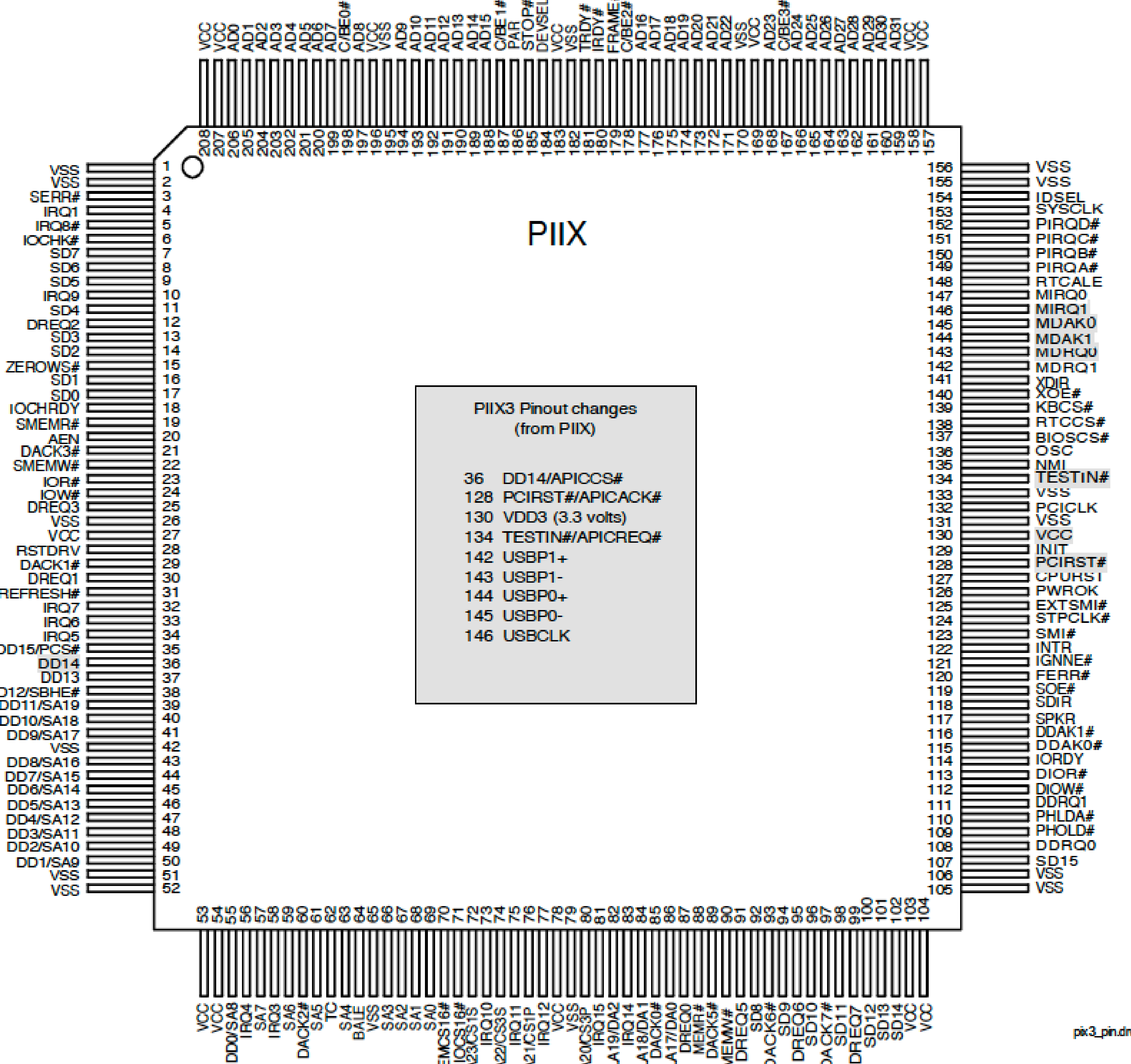


Figure 9. APIC Signal Connections



051905

Figure 10. Fast On/Off Flow


PIIX
1 VSS
2 VSS
3 SERR#
4 IRQ1
5 IRQ8#
6 IOCHK#
7 SD7
8 SD6
9 SD5
10 IRQ9
11 SD4
12 DREQ2
13 SD3
14 SD2
15 ZEROWS#
16 SD1
17 SD0
18 IOCHRDY
19 SMEMR#
20 AEN
21 DACK3#
22 SMEMW#
23 IOR#
24 IOW#
25 DREQ3
26 VSS
27 VCC
28 RSTDRV
29 DACK1#
30 DREQ1
31 REFRESH#
32 IRQ7
33 IRQ6
34 IRQ5
35 DD15/PCS#
36 DD14
37 DD13
38 DD12/SBHE#
39 DD11/SA19
40 DD10/SA18
41 DD9/SA17
42 VSS
43 DD8/SA16
44 DD7/SA15
45 DD6/SA14
46 DD5/SA13
47 DD4/SA12
48 DD3/SA11
49 DD2/SA10
50 DD1/SA9
51 VSS
52 VSS
53 VCC
54 VCC
55 DD0/SA8
56 IRQ4
57 SA7
58 IRQ3
59 SA6
60 DACK2#
61 SA5
62 TC
63 SA4
64 BALE
65 VSS
66 SA3
67 SA2
68 SA1
69 SA0
70 MEMCS16#
71 IOCS16#
72 LA23/CS1S
73 IRQ10
74 LA22/CS3S
75 IRQ11
76 LA21/CS1P
77 IRQ12
78 VCC
79 VSS
80 LA20/CS3P
81 IRQ15
82 LA19/DA2
83 IRQ14
84 LA18/DA1
85 DACK0#
86 LA17/DA0
87 DREQ0
88 MEMR#
89 DACK5#
90 MEMW#
91 DREQ5
92 SD8
93 DACK6#
94 SD9
95 DREQ6
96 SD10
97 DACK7#
98 SD11
99 DREQ7
100 SD12
101 SD13
102 SD14
103 VCC
104 VCC
105 VSS
106 VSS
107 SD15
108 DDRQ0
109 PHOLD#
110 PHLDA#
111 DDRQ1
112 DIOW#
113 DIOR#
114 IORDY
115 DDAK0#
116 DDAK1#
117 SPKR
118 SDIR
119 SOE#
120 FERR#
121 IGNNE#
122 INTR
123 SMI#
124 STPCLK#
125 EXTSMI#
126 PWROK
127 CPURST
128 PCIRST#
129 INIT
130 VCC
131 VSS
132 PCICLK
133 VSS
134 TESTIN#
135 NMI
136 OSC
137 BIOSCS#
138 RTCCS#
139 KBCS#
140 XOE#
141 XDIR
142 MDRQ1
143 MDRQ0
144 MDAK1
145 MDAK0
146 MIRQ1
147 MIRQ0
148 RTCALE
149 PIRQA#
150 PIRQB#
151 PIRQC#
152 PIRQD#
153 SYSCLK
154 IDSEL
155 VSS
156 VSS
157 VCC
158 VCC
159 AD31
160 AD30
161 AD29
162 AD28
163 AD27
164 AD26
165 AD25
166 AD24
167 C/BE3#
168 AD23
169 VCC
170 VSS
171 AD22
172 AD21
173 AD20
174 AD19
175 AD18
176 AD17
177 AD16
178 C/BE2#
179 FRAME#
180 IRDY#
181 TRDY#
182 VSS
183 VCC
184 DEVSEL#
185 STOP#
186 PAR
187 C/BE1#
188 AD15
189 AD14
190 AD13
191 AD12
192 AD11
193 AD10
194 AD9
195 VSS
196 VCC
197 AD8
198 C/BE0#
199 AD7
200 AD6
201 AD5
202 AD4
203 AD3
204 AD2
205 AD1
206 AD0
207 VCC
208 VCC
PIIX3 Pinout changes
(from PIIX)
36 DD14/APICCS#
128 PCIRST#/APICACK#
130 VDD3 (3.3 volts)
134 TESTIN#/APICREQ#
142 USBP1+
143 USBP1-
144 USBP0+
145 USBP0-
146 USBCLK
pix3_pin.drw

82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC)

- **Provides Multiprocessor Interrupt Management**
  - Dynamic Interrupt Distribution-Routing Interrupt to the Lowest Priority Processor
  - Software Programmable Control of Interrupt Inputs
  - Off Loads Interrupt Related Traffic From the Memory Bus
- **24 Programmable Interrupts**
  - 13 ISA Interrupts Supported
  - 4 PCI Interrupts
  - 1 Interrupt/SMI# Rerouting
  - 2 Motherboard Interrupts
  - 1 Interrupt Used for INTR Input
  - 3 General Purpose Interrupts
  - Independently Programmable for Edge/Level Sensitivity Interrupts
  - Each Interrupt Can Be Programmed to Respond to Active High or Low Inputs
- **X-Bus Interface**
  - CS For Flexible Decode of the IOAPIC Device.
  - Index Register Interface for Optimum Memory Usage
  - Registers are 32-Bit Wide to Match the PCI to Host Bridge Architecture
- **Package 64-Pin PQFP**

The 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC) provides multi-processor interrupt management and incorporates both static and dynamic symmetric interrupt distribution across all processors. In systems with multiple I/O subsystems, each subsystem can have its own set of interrupts. Each interrupt pin is individually programmable as either edge or level triggered. The interrupt vector and interrupt steering information can be specified per interrupt. An indirect register accessing scheme optimizes the memory space needed to access the IOAPIC's internal registers. To increase system flexibility when assigning memory space usage, the The IOAPIC's 2-register memory space is re-locatable.

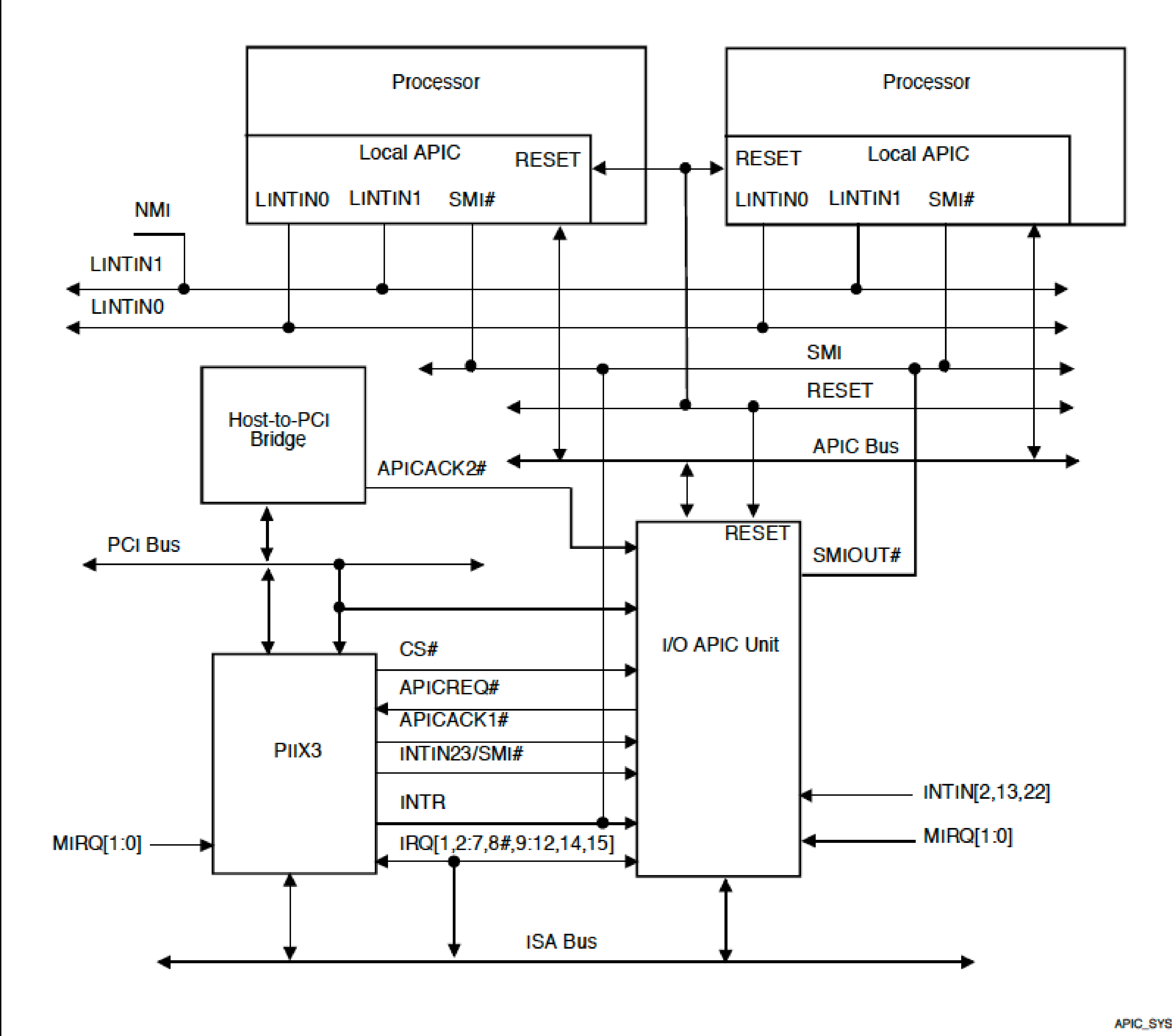


Figure 2. I/O And Local APIC Units

2.0. SIGNAL DESCRIPTION

This section contains a detailed description of each signal. The signals are arranged in function groups according to their interface.

Note that the “#” symbol at the end of a signal name indicates that the active, or asserted state occurs when the signal is at a low voltage level. When “#” is not present after the signal name, the signal is asserted when at the high voltage level.

The terms' assertion and negation are used extensively. This is done to avoid confusion when working with a mixture of ‘active-low’ and ‘active-high’ signals. The term assert, or assertion indicates that a signal is active, independent of whether that level is represented by a high or low voltage. The term negate, or negation indicates that a signal is inactive.

The following notations are used to describe the signal and type:

I	Input pin
O	Output pin
ST	Schmitt Trigger Input pin
OD	Open Drain Output pin. This requires a pull-up to the VCC of the processor core
I/OD	Bi-directional Input withOpen Drain Output pin.
I/O	Bi-directional Input/Output pin

2.1. System Bus Signals

Signal Name	Type	Description
D[7:0]	I/O	**DATA:** D[7:0] contain the data when writing to or reading from internal IOAPIC registers. These signals are outputs when reading data from the IOAPIC and they are inputs when writing data to the IOAPIC. These signals are tri-stated during reset.
D/I#	I	**DATA/INDEX#:** This input selects whether the I/O Register Select (IOREGSEL) Register or I/O Window (IOWIN) Register is accessed. All internal IOAPIC registers are accessed with an indexing scheme. When the D/I# pin is low, the IOREGSEL Register is accessed. When the D/I# pin is high, the data becomes available from the register pointed to by the index register. Typically, this signal is connected to SA4 on the ISA bus (i.e., IOREGSEL Register is at 00h and IOWIN Register is at 10h).
A[1:0]	I	**ADDRESS:** The IOAPIC is a 32 bit device with an 8 bit ISA interface. A[1:0] steer the data byte to the correct 8 bit location within the 32 bit register. Typically, these input signals are connected to SA[1:0] of the ISA bus.
RD#	I	**READ STROBE:** RD# causes the IOAPIC to respond by driving internal register data onto the D[7:0] pins. Typically this pin is connected to the MEMRD# signal on the ISA bus.
WR#	I	**WRITE STROBE:** When this signal transitions from low to high, the data present on the IOAPIC's D[7:0] signals are written to an internal register. Typically, this signal is connected to the MEMWR# signal on the ISA bus.
CS#	I	**CHIP SELECT:** This active low input selects the IOAPIC as the target of the current read or write transaction.

Signal Name	Type	Description
APICREQ#	O	**APIC REQUEST:** APICREQ# is asserted prior to the APIC sending an interrupt message over the APIC data bus. This is the request part of a handshake that insures system level buffer coherency prior to sending an interrupt over the APIC bus. This signal is tri-stated during reset. This signal has an internal pull-up resistor.
APICACK1#	I	**APIC ACKNOWLEDGE 1:** This signal is the acknowledge part of the handshake indicating that the APIC can send the interrupt message over the APIC bus. This signal is typically connected to the PIIX3.
APICACK2#	I	**APIC ACKNOWLEDGE 2:** This signal is the second half of the acknowledge handshake indicating that the APIC can send the interrupt message over the APIC bus. This signal is typically connected to the host-to-PCI bridge and along with APICREQ# and APICACK1# makes up the complete buffer coherency protocol cycles. If the system does not have a host-to-PCI bridge, this signal can be tied low.

2.2. Clock and Reset Signals

Signal Name	Type	Description
PCICLK	I	**PCI CLOCK:** This signal is used to synchronize and strobe the data buffer status signals (APICREQ#, APICACK1#, and APICACK2#). This signal is typically connect to the PCI clock.
RESET	I	**RESET:** RESET initializes the IOAPIC's internal logic and sets the register bits to their default value.

2.3. APIC Bus Interface

Signal Name	Type	Description
APICD[1:0]	I/OD	**APIC DATA:** These signals are used to send and receive data over the APIC bus. These signals are tri-stated during reset and must be pulled up to the appropriate VCC levels of the CPU.
APICCLK	I	**APIC CLOCK:** The input signal is used to determine when valid data is being sent over the APIC bus.

2.4. Interrupt Signals

Signal Name	Type	Description
INTIN0	ST	**Interrupt Input 0:** This signal is connected to the redirection table entry 0. Typically, this signal may be connected to the INTR on the PIIX3 to communicate the status of IRQ0 and IRQ13 interrupts. Note that the IRQ0 and IRQ13 interrupts are embedded in the PIIX3 and are not available to the rest of the system.
INTIN1	ST	**Interrupt Input 1:** INTIN1 is connected to the redirection table entry 1. Typically, this signal will be connected to the keyboard interrupt (IRQ1).

Signal Name	Type	Description
INTIN2	ST	**Interrupt Input 2:** This signal is connected to the redirection table entry 2. If IRQ0 interrupt is available in hardware, it is connected to this pin.
INTIN[3:11, 14,15]	ST	**Interrupt Inputs 3 through 11, 14 and 15:** These signals are connected to the redirection table entries 3:11, 14 & 15. Typically, these signals are connected to the ISA interrupts IRQ[3:7,8#,9:11,14:15] respectively.
INTIN12	ST	**Interrupt Input 12:** This signal is connected to the redirection table entry 12. Typically, this signal will be connected to the mouse interrupt (IRQ12/M).
INTIN13	ST	**Interrupt input 13:** This signal is connected to the redirection table entry 13. If IRQ13 interrupt is available in hardware, it is connected to this signal. If IRQ13 is not available, it is routed through the INTR interrupt and this signal becomes INTIN13 (redirection table entry 13).
INTIN[16:19]	ST	**Interrupt inputs 16 through 19:** These signals are connected to the redirection table entries [16:19]. Typically, these signals are connected to the PCI interrupts (PIRQ[0:3]). The steering of the PCI IRQs to the ISA IRQs is accomplished in the IOAPIC by setting the PCI redirection table entry to the correct ISA interrupt vector.
INTIN[20:21]	ST	**Interrupt inputs 20 and 21:** These signals are connected to the redirection table entries 20 and 21. Typically, these signals are connected to the motherboard interrupts (MIRQ[0:1]). These pins could be used for the NMI and INIT signals or just general purpose interrupts.
INTIN22	ST	**Interrupt input 22:** This signal is connect to the redirection table entry 22. This signal is a general purpose interrupt.
INTIN23/ SMI#	ST	**Interrupt input 23:** This signal is connected to the redirection table entry 23. This input has a special feature for the SMI# interrupt routing. If the Mask bit is not set, the signal is a normal interrupt input that is sent over the APIC bus just like all the other interrupts. When the Mask bit is set, the INTIN23/SMI# input is routed through the IOAPIC to the SMIOUT# output signal.
SMIOUT#	OD	**SMI OUTPUT:** This signal is an output in response to the SMI# input when the MASK bit for the redirection table entry number 23 is set. If the MASK bit is not set, the redirection table can be setup to deliver an SMI# over the APIC bus.

2.5. Test and Power Signals

Pin Name	Type	Description
TESTIN#	I	**TEST INPUT:** This active-low input is used to invoke test modes. TESTIN# should be pulled high during normal operation.
VCC		**VCC POWER PIN:** 5V ± 10%.
GND		**GROUND POWER PIN:**

3.0. REGISTER DESCRIPTION

The IOAPIC is addressed with a CS# and the D/I# pin. The PIIX3 decodes the IOAPIC in memory space and sends a CS# to the IOAPIC device, when it is selected. The D/I# pin selects between the IOREGSEL Register (D/I#=0) and the IOWIN Register (D/I#=1). Typically, D/I# is connected to SA4 on the ISA bus (i.e., IOREGSEL Register is at 00h and IOWIN Register is at 10h).

The IOAPIC registers are accessed by an indirect addressing scheme using two registers (IOREGSEL and IOWIN) that are located in the CPU's memory space (memory address specified by the APICBASE Register located in the PIIX3). These two registers are re-locateable (via the APICBASE Register) as shown in Table 3.1. In the IOAPIC only the IOREGSEL and IOWIN Registers are directly accesable in the memory address space.

To reference an IOAPIC register, a byte memory write that the PIIX3 decodes for the IOAPIC loads the IOREGSEL Register with an 8-bit value that specifies the IOAPIC register (address offset in Table 3.2) to be accessed. The IOWIN Register is then used to read/write the desired data from/to the IOAPIC register specified by bits [7:0] of the IOREGSEL Register. The IOWIN Register must be accessed as a Dword quantity.

The IOREGSEL and IOWIN Registers (Table 3.1) can be relocated via the APIC Base Address Relocation Register in the PIIX3 and are aligned on 128 bit boundaries. All APIC registers are accessed using 32 bit loads and stores. This implies that to modify a field (e.g., bit, byte) in any register, the whole 32 bit register must be read, the field modified, and the 32 bits written back. In addition, registers that are described as 64 bits wide are accessed as multiple independent 32 bit registers.

Table 1. Memory Mapped Registers For Accessing IOAPIC Registers

Memory Address	Mnemonic	Register Name	Access	D/I# Signal
FEC0 xy00h	IOREGSEL	I/O Register Select (index)	R/W	0
FEC0 xy10h	IOWIN	I/O Window (data)	R/W	1

NOTES:

xy are determined by the x and y fields in the APIC Base Address Relocation Register located in the PIIX3. Range for x = 0-Fh and the range for y = 0,4,8,Ch.

Table 2. IOAPIC Registers

Address Offset	Mnemonic	Register Name	Access
00h	IOAPICID	IOAPIC ID	R/W
01h	IOAPICVER	IOAPIC Version	RO
02h	IOAPICARB	IOAPIC Arbitration ID	RO
10–3Fh	IOREDTBL[0:23]	Redirection Table (Entries 0-23) (64 bits each)	R/W

NOTES:

Address Offset is determined by I/O Register Select Bits [7:0].

3.1. Memory Mapped Registers for Accessing IOAPIC Registers

3.1.1. IOREGSEL—I/O REGISTER SELECT REGISTER

Memory Address: FEC0 xy00h (xy=See APICBASE Register in the PIIX3)
Default Value: 00h
Attribute: Read/Write

This register selects the IOAPIC Register to be read/written. The data is then read from or written to the selected register through the IOWIN Register.

Bit	Description
31:8	**Reserved.**
7:0	**APIC Register Address—R/W.** Bits [7:0] specify the IOAPIC register to be read/written via the IOWIN Register.

3.1.2. IOWIN—I/O WINDOW REGISTER

Memory Address: FEC0 xy10h (xy=See APICBASE Register in PIIX3)
Default Value: 00h
Attribute: Read/Write

This register is used to write to and read from the register selected by the IOREGSEL Register. Readability/writability is determined by the IOAPIC register that is currently selected.

Bit	Description
31:0	**APIC Register Data—R/W.** Memory references to this register are mapped to the APIC register specified by the contents of the IOREGSEL Register.

3.2. IOAPIC Registers

3.2.1. IOAPICID—IOAPIC IDENTIFICATION REGISTER

Address Offset: 00h
Default Value: 00h
Attribute: Read/Write

This register contains the 4-bit APIC ID. The ID serves as a physical name of the IOAPIC. All APIC devices using the APIC bus should have a unique APIC ID. The APIC bus arbitration ID for the I/O unit is also writtten during a write to the APICID Register (same data is loaded into both). This register must be programmed with the correct ID value before using the IOAPIC for message transmission.

Bit	Description
31:28	**Reserved.**
27:24	**IOAPIC Identification—R/W.** This 4 bit field contains the IOAPIC identification.
23:0	**Reserved.**

3.2.2. IOAPICVER—IOAPIC VERSION REGISTER

Address Offset: 01h
Default Value: 00170011h
Attribute: Read Only

The IOAPIC Version Register identifies the APIC hardware version. Software can use this to provide compatibility between different APIC implementations and their versions. In addition, this register provides the maximum number of entries in the I/O Redirection Table.

Bit	Descriptions
31:24	**Reserved.**
23:16	**Maximum Redirection Entry—RO.** This field contains the entry number (0 being the lowest entry) of the highest entry in the I/O Redirection Table. The value is equal to the number of interrupt input pins for the IOAPIC minus one. The range of values is 0 through 239. For this IOAPIC, the value is 17h.
15:8	**Reserved.**
7:0	**APIC VERSION—RO.** This 8 bit field identifies the implementation version. The version number assigned to the IOAPIC is 11h.

3.2.3. IOAPICARB—IOAPIC ARBITRATION REGISTER

Address Offset: 02h
Default Value: 0000_0000h
Attribute: Read Only

The APICARB Register contains the bus arbitration priority for the IOAPIC. This register is loaded when the IOAPIC ID Register is written.

The APIC uses a one wire arbitration to win bus ownership. A rotating priority scheme is used for arbitration. The winner of the arbitration becomes the lowest priority agent and assumes an arbitration ID of 0.

All other agents, except the agent whose arbitration ID is 15, increment their arbitration IDs by one. The agent whose ID was 15 takes the winner's arbitration ID and increments it by one. Arbitration IDs are changed (incremented or assumed) only for messages that are transmitted successfully (except, in the case of low priority messages where Arbitration ID is changed even if message was not successfully transmitted). A message is transmitted successfully if no checksum error or acceptance error is reported for that message. The APICARB Register is always loaded with IOAPIC ID during a "level triggered INIT with de-assert" message.

Bit	Description
31:28	**Reserved.**
27:24	**IOAPIC Identification—R/W.** This 4 bit field contains the IOAPIC Arbitration ID.
23:0	**Reserved.**

3.2.4. IOREDTBL[23:0]—I/O REDIRECTION TABLE REGISTERS

Address Offset:	10–11h (IOREDTBL0)	28–29h (IOREDTBL12)
	12–13h (IOREDTBL1)	2A–2Bh (IOREDTBL13)
	14–15h (IOREDTBL2)	2C–2Dh (IOREDTBL14)
	16–17h (IOREDTBL3)	2E–2Fh (IOREDTBL15)
	18–19h (IOREDTBL4)	30–31h (IOREDTBL16)
	1A–1Bh (IOREDTBL5)	32–33Fh (IOREDTBL17)
	1C–1Dh (IOREDTBL6)	34–35h (IOREDTBL18)
	1E–1Fh (IOREDTBL7)	36–37h (IOREDTBL19)
	20–21h (IOREDTBL8)	38–39h (IOREDTBL20)
	22–23h (IOREDTBL9)	3A–3Bh (IOREDTBL21)
	24–25h (IOREDTBL10)	3C–3Dh (IOREDTBL22)
	26–27h (IOREDTBL11)	3E–3Fh (IOREDTBL23)
Default Value:	xxx1 xxxx xxxx xxxxh	
Attribute:	Read/Write	

There are 24 I/O Redirection Table entry registers. Each register is a dedicated entry for each interrupt input signal. Unlike IRQ pins of the 8259A, the notion of interrupt priority is completely unrelated to the position of the physical interrupt input signal on the APIC. Instead, software determines the vector (and therefore the priority) for each corresponding interrupt input signal. For each interrupt signal, the operating system can also specify the signal polarity (low active or high active), whether the interrupt is signaled as edges or levels, as well as the destination and delivery mode of the interrupt. The information in the redirection table is used to translate the corresponding interrupt pin information into an inter-APIC message.

The IOAPIC responds to an edge triggered interrupt as long as the interrupt is wider than one CLK cycle. The interrupt input is asynchronous; thus, setup and hold times need to be guaranteed for at lease one rising edge of the CLK input. Once the interrupt is detected, a delivery status bit internal to the IOAPIC is set. A new edge on that Interrupt input pin will not be recongnized until the IOAPIC Unit broadcasts the corresponding message over the APIC bus and the message has been accepted by the destination(s) specified in the destination field. That new edge only results in a new invocation of the handler if its acceptance by the destination APIC causes the Interrupt Request Register bit to go from 0 to 1. (In other words, if the interrupt wasn't already pending at the destination.)

Bit	Description
63:56	**Destination Field—R/W.** If the Destination Mode of this entry is Physical Mode (bit 11=0), bits [59:56] contain an APIC ID. If Logical Mode is selected (bit 11=1), the Destination Field potentially defines a set of processors. Bits [63:56] of the Destination Field specify the logical destination address. **Destination Mode IOREDTBLx[11]** — **Logical Destination Address** 0, Physical Mode — IOREDTBLx[59:56] = APIC ID 1, Logical Mode — IOREDTBLx[63:56] = Set of processors
55:17	**Reserved.**

Bit	Description
16	**Interrupt Mask—R/W.** When this bit is 1, the interrupt signal is masked. Edge-sensitive interrupts signaled on a masked interrupt pin are ignored (i.e., not delivered or held pending). Level-asserts or negates occurring on a masked level-sensitive pin are also ignored and have no side effects. Changing the mask bit from unmasked to masked after the interrupt is accepted by a local APIC has no effect on that interrupt. This behavior is identical to the case where the device withdraws the interrupt before that interrupt is posted to the processor. It is software's responsibility to handle the case where the mask bit is set after the interrupt message has been accepted by a local APIC unit but before the interrupt is dispensed to the processor. When this bit is 0, the interrupt is not masked. An edge or level on an interrupt pin that is not masked results in the delivery of the interrupt to the destination.
15	**Trigger Mode—R/W.** The trigger mode field indicates the type of signal on the interrupt pin that triggers an interrupt. 1=Level sensitive, 0=Edge sensitive.
14	**Remote IRR—RO.** This bit is used for level triggered interrupts. Its meaning is undefined for edge triggered interrupts. For level triggered interrupts, this bit is set to 1 when local APIC(s) accept the level interrupt sent by the IOAPIC. The Remote IRR bit is set to 0 when an EOI message with a matching interrupt vector is received from a local APIC.
13	**Interrupt Input Pin Polarity (INTPOL)—R/W.** This bit specifies the polarity of the interrupt signal. 0=High active, 1=Low active.
12	**Delivery Status (DELIVS)—RO.** The Delivery Status bit contains the current status of the delivery of this interrupt. Delivery Status is read-only and writes to this bit (as part of a 32 bit word) do not effect this bit. 0=IDLE (there is currently no activity for this interrupt). 1=Send Pending (the interrupt has been injected but its delivery is temporarily held up due to the APIC bus being busy or the inability of the receiving APIC unit to accept that interrupt at that time).
11	**Destination Mode (DESTMOD)—R/W.** This field determines the interpretation of the Destination field. When DESTMOD=0 (physical mode), a destination APIC is identified by its ID. Bits 56 through 59 of the Destination field specify the 4 bit APIC ID. When DESTMOD=1 (logical mode), destinations are identified by matching on the logical destination under the control of the Destination Format Register and Logical Destination Register in each Local APIC. **Destination Mode IOREDTBLx[11]** — **Logical Destination Address** 0, Physical Mode — IOREDTBLx[59:56] = APIC ID 1, Logical Mode — IOREDTBLx[63:56] = Set of processors

Bit	Description
10:8	**Delivery Mode (DELMOD)—R/W.** The Delivery Mode is a 3 bit field that specifies how the APICs listed in the destination field should act upon reception of this signal. Note that certain Delivery Modes only operate as intended when used in conjunction with a specific trigger Mode. These restrictions are indicated in the following table for each Delivery Mode. **Bits [10:8] Mode Description** 000 Fixed: Deliver the signal on the INTR signal of all processor cores listed in the destination. Trigger Mode for "fixed" Delivery Mode can be edge or level. 001 Lowest Priority: Deliver the signal on the INTR signal of the processor core that is executing at the lowest priority among all the processors listed in the specified destination. Trigger Mode for "lowest priority". Delivery Mode can be edge or level. 010 SMI: System Management Interrupt. A delivery mode equal to SMI requires an edge trigger mode. The vector information is ignored but must be programmed to all zeroes for future compatibility. 011 Reserved 100 NMI: Deliver the signal on the NMI signal of all processor cores listed in the destination. Vector information is ignored. NMI is treated as an edge triggered interrupt, even if it is programmed as a level triggered interrupt. For proper operation, this redirection table entry must be programmed to “edge” triggered interrupt. 101 INIT: Deliver the signal to all processor cores listed in the destination by asserting the INIT signal. All addressed local APICs will assume their INIT state. INIT is always treated as an edge triggered interrupt, even if programmed otherwise. For proper operation, this redirection table entry must be programmed to “edge” triggered interrupt. 110 Reserved 111 ExtINT: Deliver the signal to the INTR signal of all processor cores listed in the destination as an interrupt that originated in an externally connected (8259A-compatible) interrupt controller. The INTA cycle that corresponds to this ExtINT delivery is routed to the external controller that is expected to supply the vector. A Delivery Mode of "ExtINT" requires an edge trigger mode.
7:0	**Interrupt Vector (INTVEC)—R/W:** The vector field is an 8 bit field containing the interrupt vector for this interrupt. Vector values range from 10h to FEh.

4.1. INTIN23/SMI# and SMIOUT# Functionality

The SMI# interrupt pin can be routed through the IOAPIC device. The redirection table entry can be programmed to send an SMI# interrupt over the APIC bus. The SMI# input is sent over the APIC bus when the delivery mode register is set to 010 for INTIN23/SMI# input and the Mask bit is set to 0. During this time, the SMIOUT# pin is inactive. If the Mask bit is set to 1, the INTIN23/SMI# input is routed to the SMIOUT# pin on the IOAPIC. Refer to Figure 3.

If the SMI# routing feature is not desired, the SMIOUT# pin is not connected to SMI# of the CPU's. The SMIOUT# pin is left open in the system. The INTIN23/SMI# redirection table entry has the same functionality as all other redirection table entries.

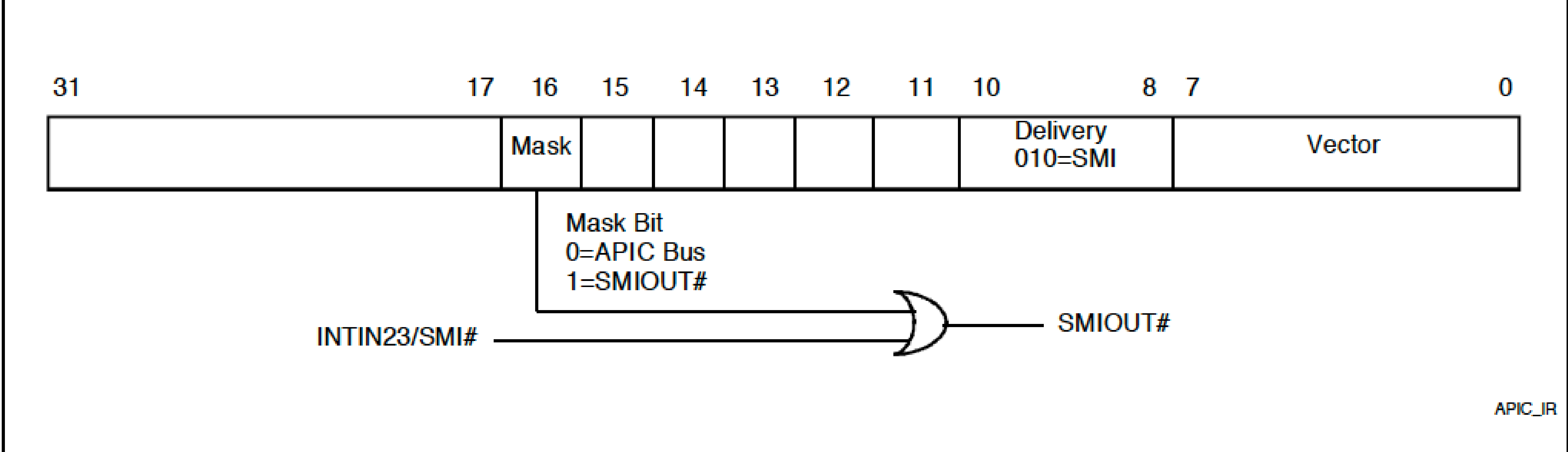
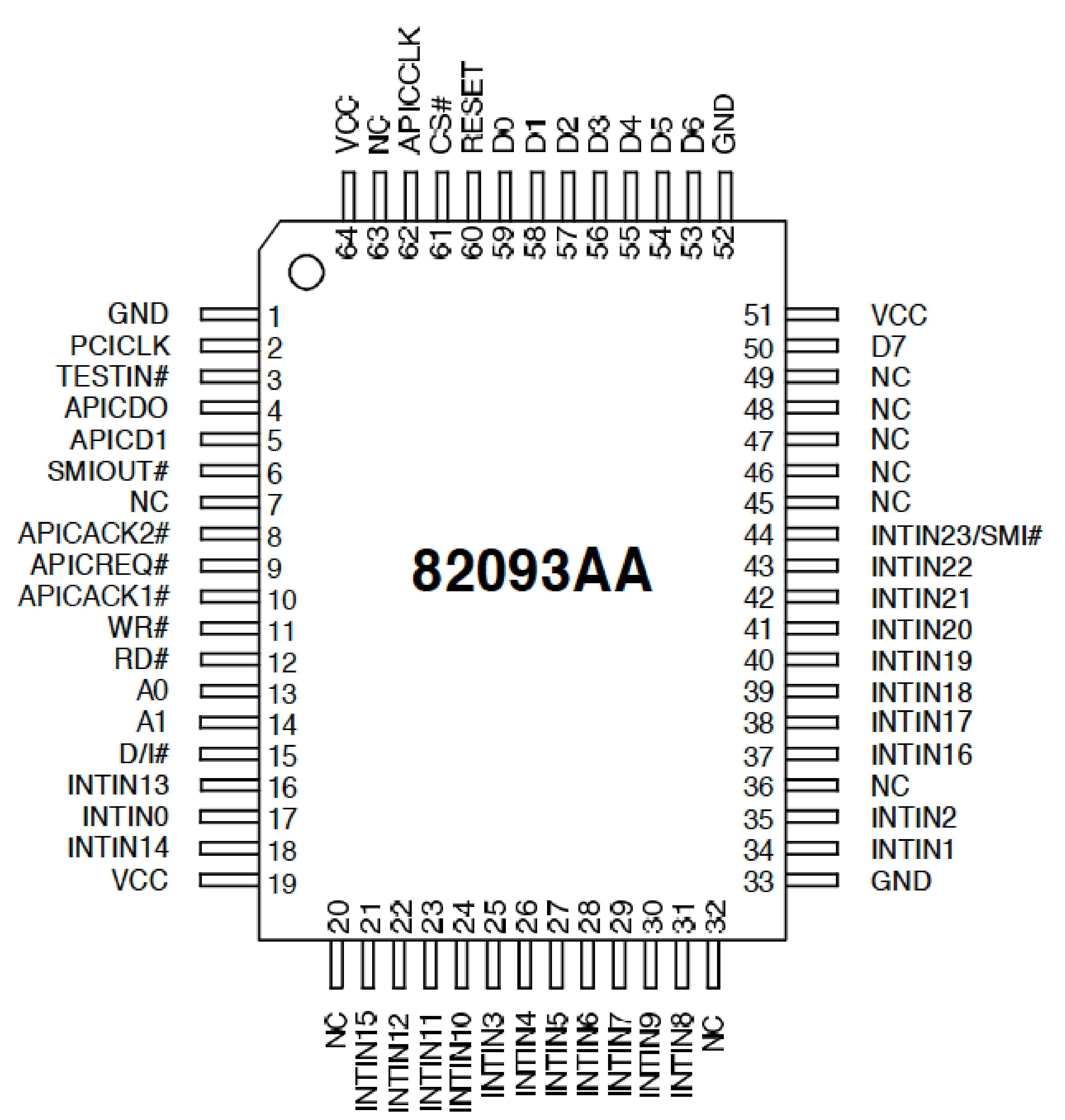
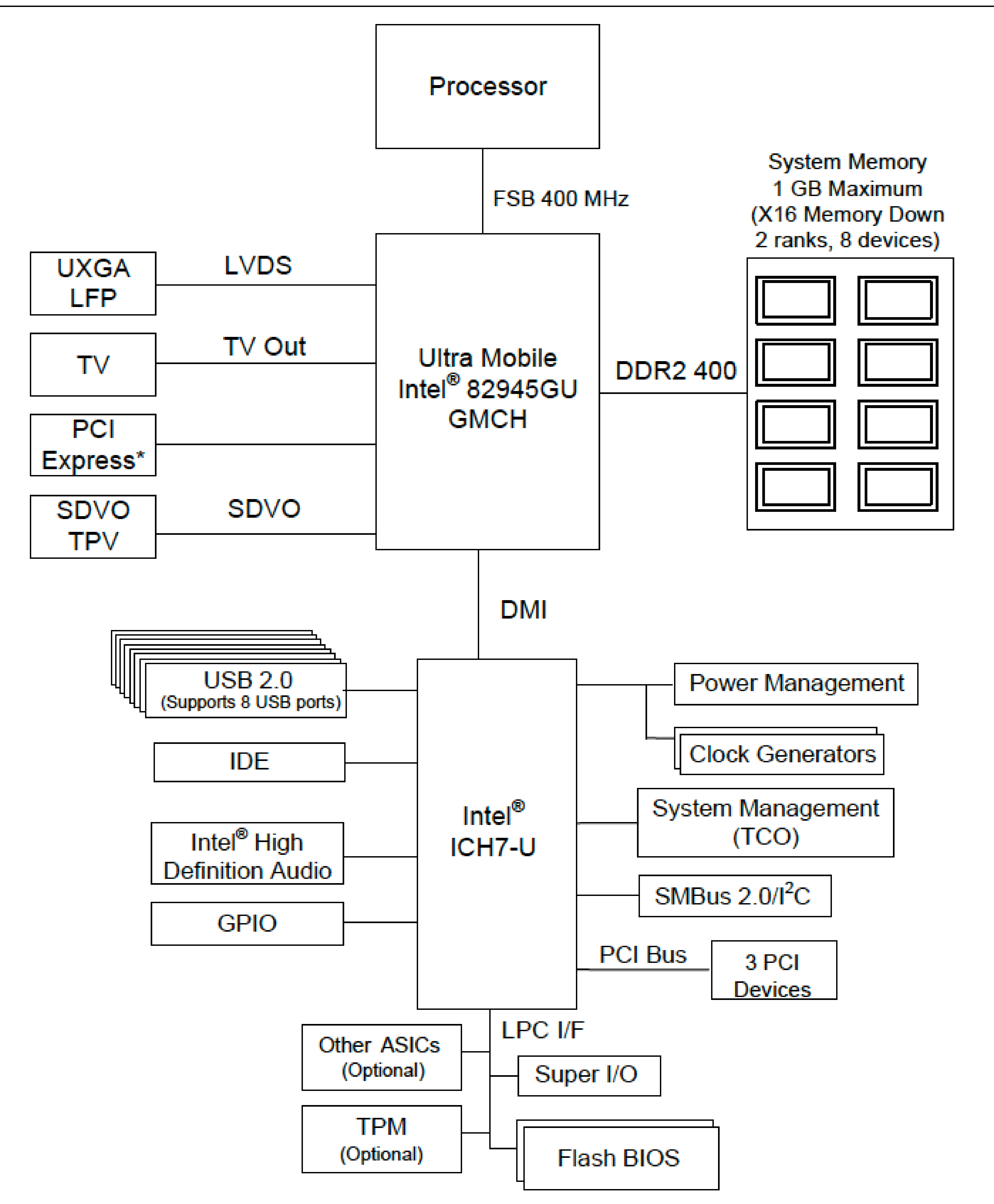


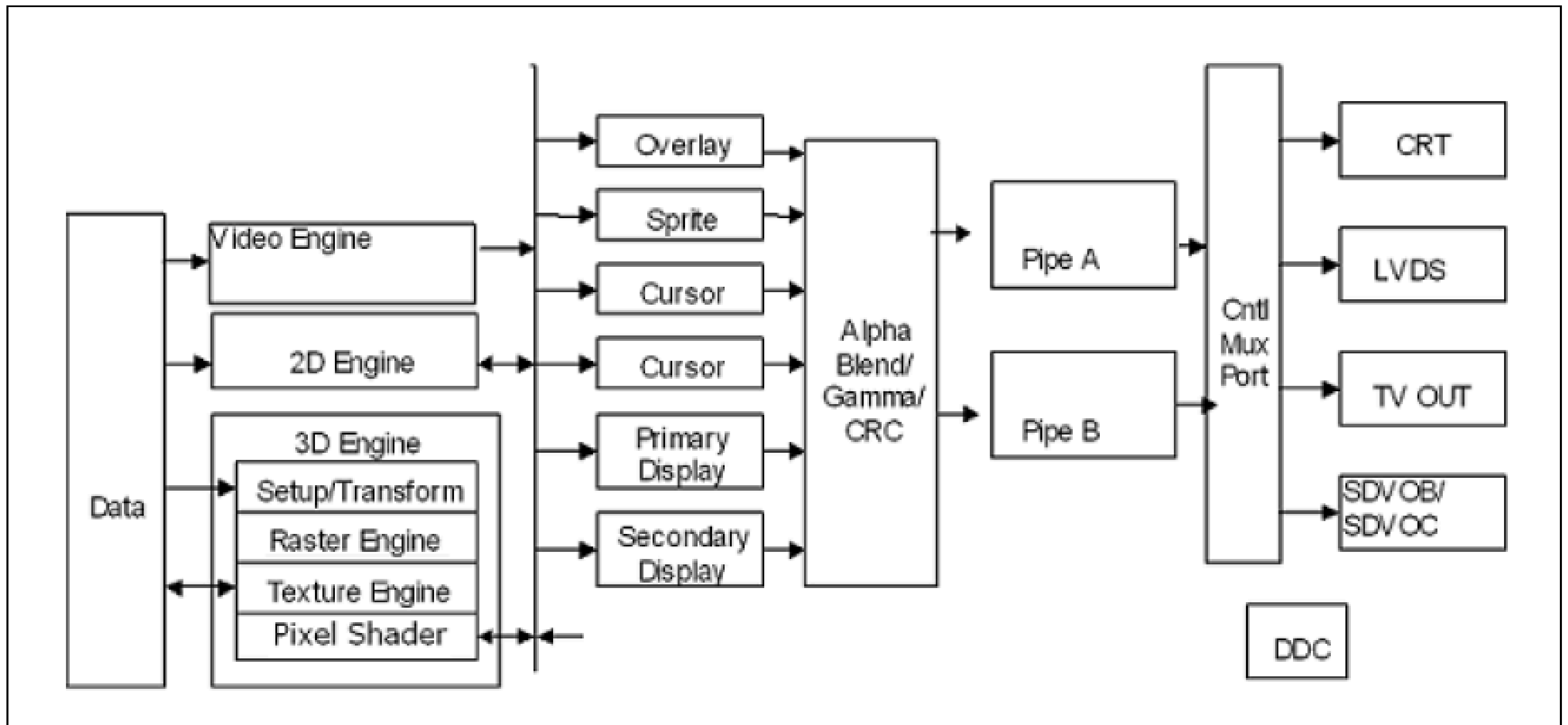
Figure 3. INTIN23/SMI# Routing And Control From Redirection Table Mask Bit


82093AA
VCC 64
NC 63
APICCLK 62
CS# 61
RESET 60
D0 59
D1 58
D2 57
D3 56
D4 55
D5 54
D6 53
GND 52
GND 1
PCICLK 2
TESTIN# 3
APICDO 4
APICD1 5
SMIOUT# 6
NC 7
APICACK2# 8
APICREQ# 9
APICACK1# 10
WR# 11
RD# 12
A0 13
A1 14
D/I# 15
INTIN13 16
INTIN0 17
INTIN14 18
VCC 19
51 VCC
50 D7
49 NC
48 NC
47 NC
46 NC
45 NC
44 INTIN23/SMI#
43 INTIN22
42 INTIN21
41 INTIN20
40 INTIN19
39 INTIN18
38 INTIN17
37 INTIN16
36 NC
35 INTIN2
34 INTIN1
33 GND
20 NC
21 INTIN15
22 INTIN12
23 INTIN11
24 INTIN10
25 INTIN3
26 INTIN4
27 INTIN5
28 INTIN6
29 INTIN7
30 INTIN9
31 INTIN8
32 NC
IOA_PIN

Ultra Mobile Intel® 945GU Express Chipset Example System Diagram



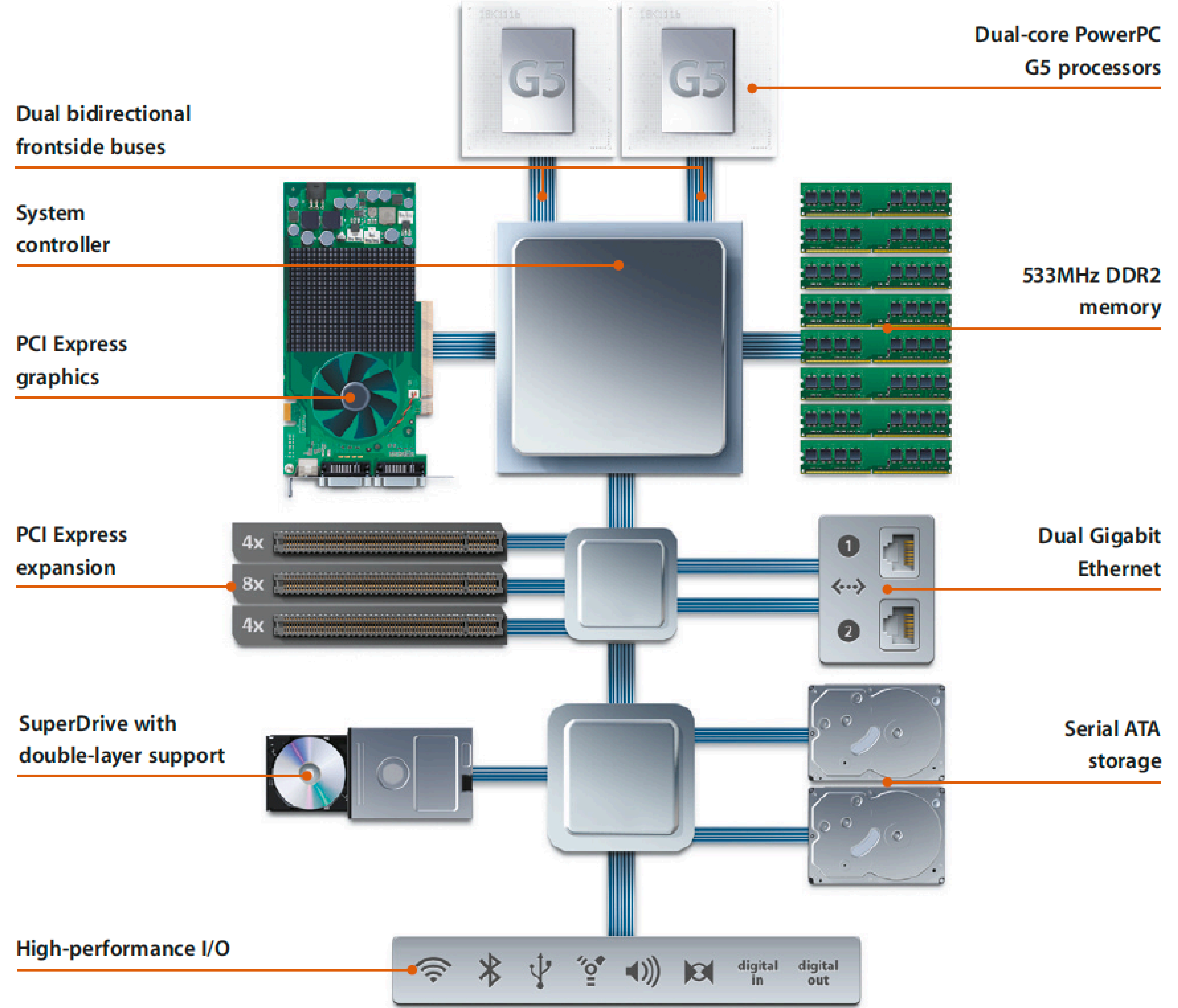
(G)MCH Graphics Controller Block Diagram



The (G)MCH contains a variety of planes, such as display, overlay, cursor and VGA. A plane consists of rectangular shaped image that has characteristics such as source, size, position, method, and format. These planes get attached to source surfaces, which are rectangular memory surfaces with a similar set of characteristics. They are also associated with a particular destination pipe.

A pipe consists of a set of combined planes and a timing generator. The (G)MCH has two independent display pipes, allowing for support of two independent display streams. A port is the destination for the result of the pipe.

The entire IGD is fed with data from its memory controller. The (G)MCH's graphics performance is directly related to the amount of bandwidth available. If the engines are not receiving data fast enough from the memory controller (e.g., single-channel DDR2 533), the rest of the IGD will also be affected.



Dual-core PowerPC G5 processors
G5
G5
Dual bidirectional frontside buses
System controller
533MHz DDR2 memory
PCI Express graphics
PCI Express expansion
4x
8x
4x
1
2
Dual Gigabit Ethernet
SuperDrive with double-layer support
Serial ATA storage
High-performance I/O
digital in
digital out

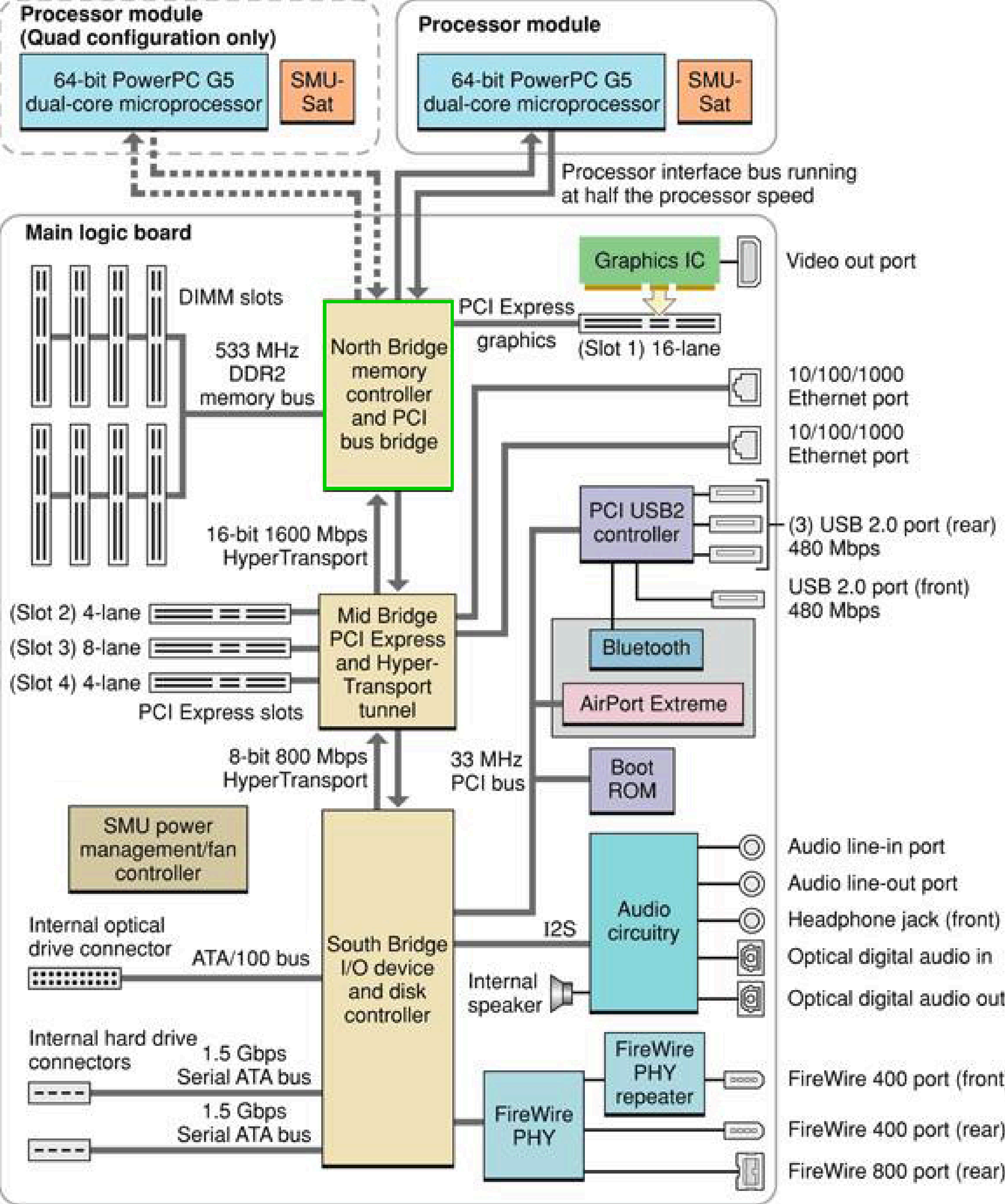

Processor module
(Quad configuration only)
64-bit PowerPC G5
dual-core microprocessor
SMU-
Sat
Processor module
64-bit PowerPC G5
dual-core microprocessor
SMU-
Sat
Processor interface bus running
at half the processor speed
Main logic board
Graphics IC
Video out port
DIMM slots
PCI Express
graphics
(Slot 1) 16-lane
533 MHz
DDR2
memory bus
North Bridge
memory
controller
and PCI
bus bridge
10/100/1000
Ethernet port
10/100/1000
Ethernet port
PCI USB2
controller
(3) USB 2.0 port (rear)
480 Mbps
16-bit 1600 Mbps
HyperTransport
USB 2.0 port (front)
480 Mbps
(Slot 2) 4-lane
(Slot 3) 8-lane
(Slot 4) 4-lane
PCI Express slots
Mid Bridge
PCI Express
and Hyper-
Transport
tunnel
Bluetooth
AirPort Extreme
8-bit 800 Mbps
HyperTransport
33 MHz
PCI bus
Boot
ROM
SMU power
management/fan
controller
Audio line-in port
Audio line-out port
Headphone jack (front)
Optical digital audio in
Optical digital audio out
Internal optical
drive connector
ATA/100 bus
I2S
South Bridge
I/O device
and disk
controller
Audio
circuitry
Internal
speaker
Internal hard drive
connectors
1.5 Gbps
Serial ATA bus
1.5 Gbps
Serial ATA bus
FireWire
PHY
repeater
FireWire
PHY
FireWire 400 port (front)
FireWire 400 port (rear)
FireWire 800 port (rear)

Figure 1-1. CPC945 Bridge and Memory Controller Block Diagram

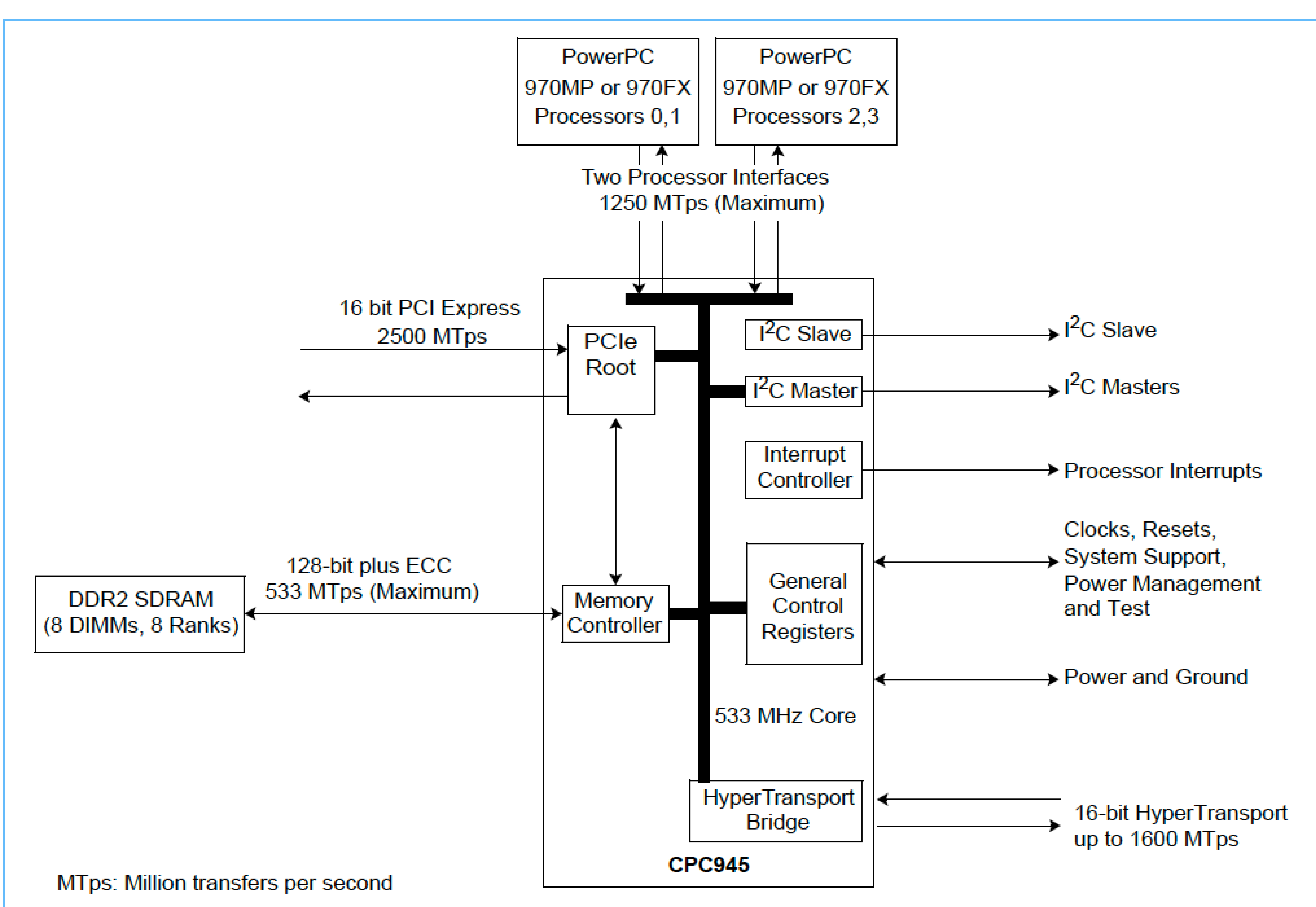


Figure 7-6. External Multiplexers

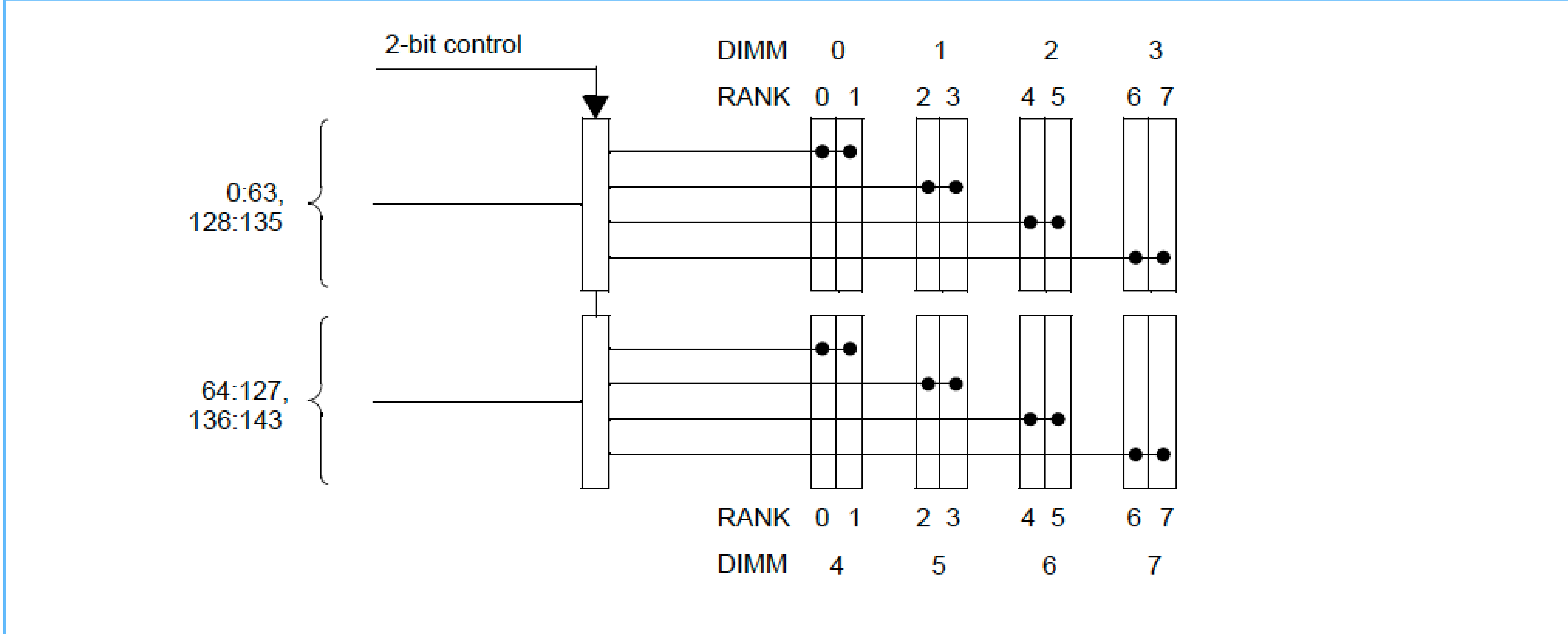
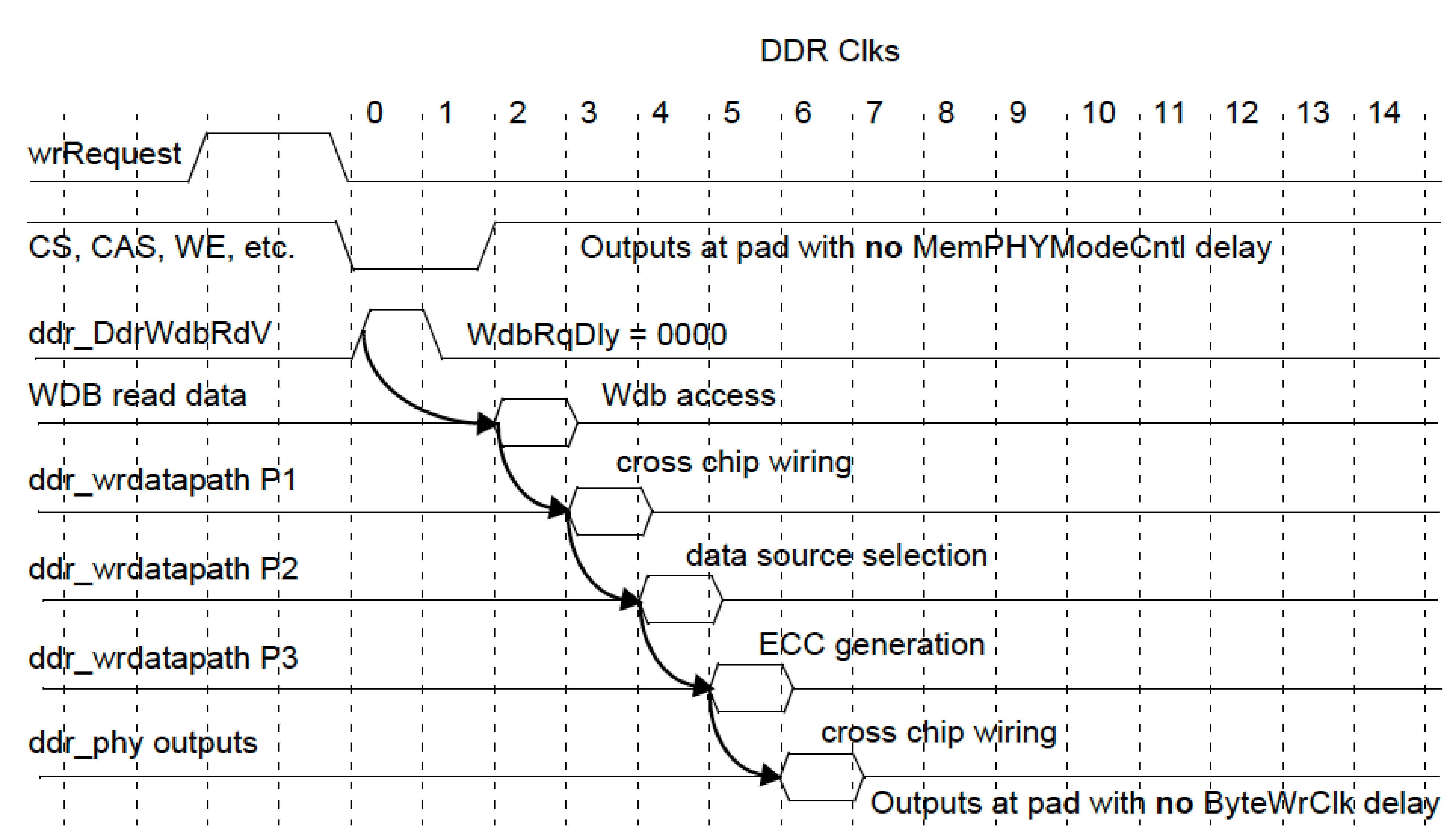
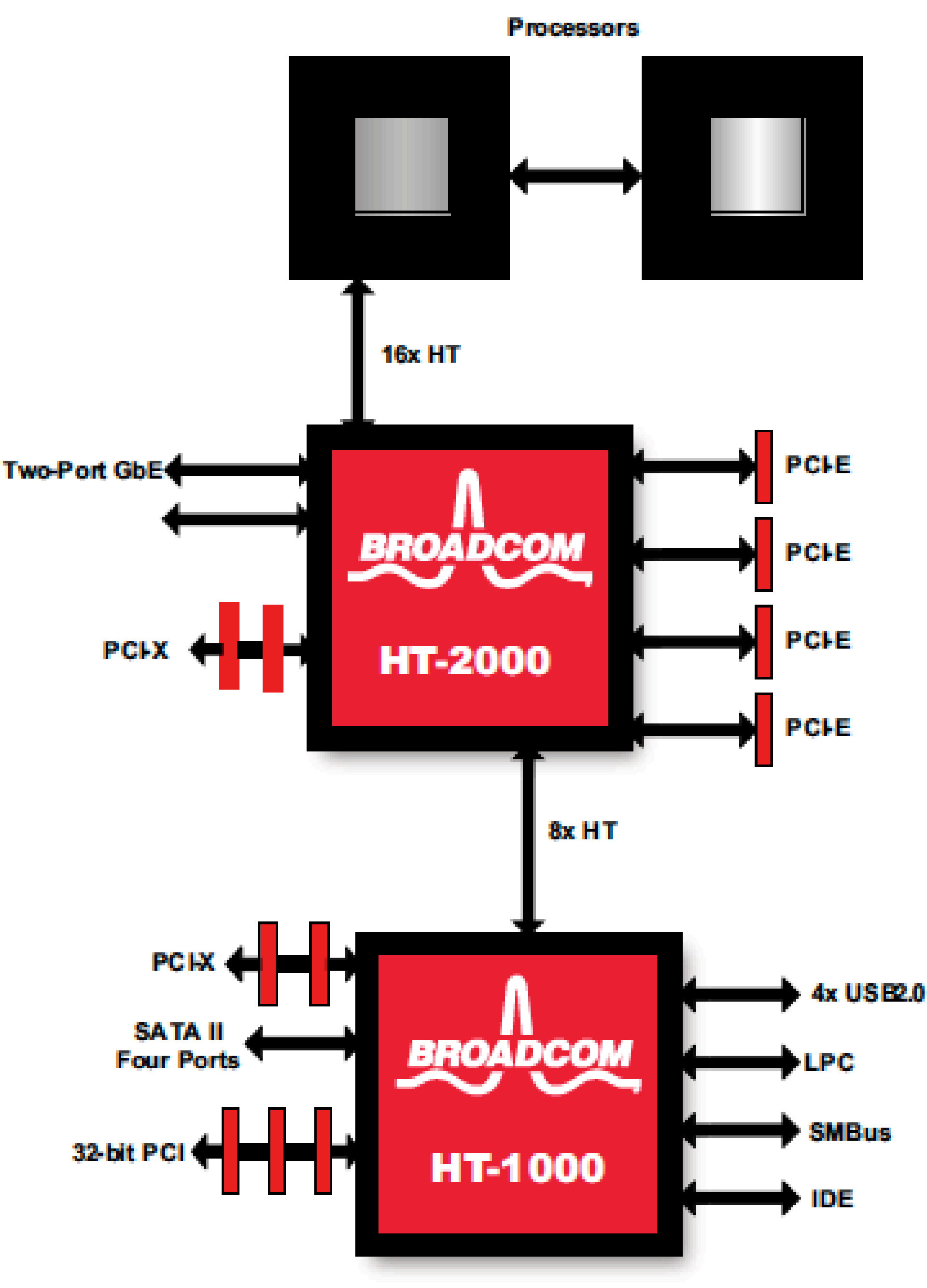
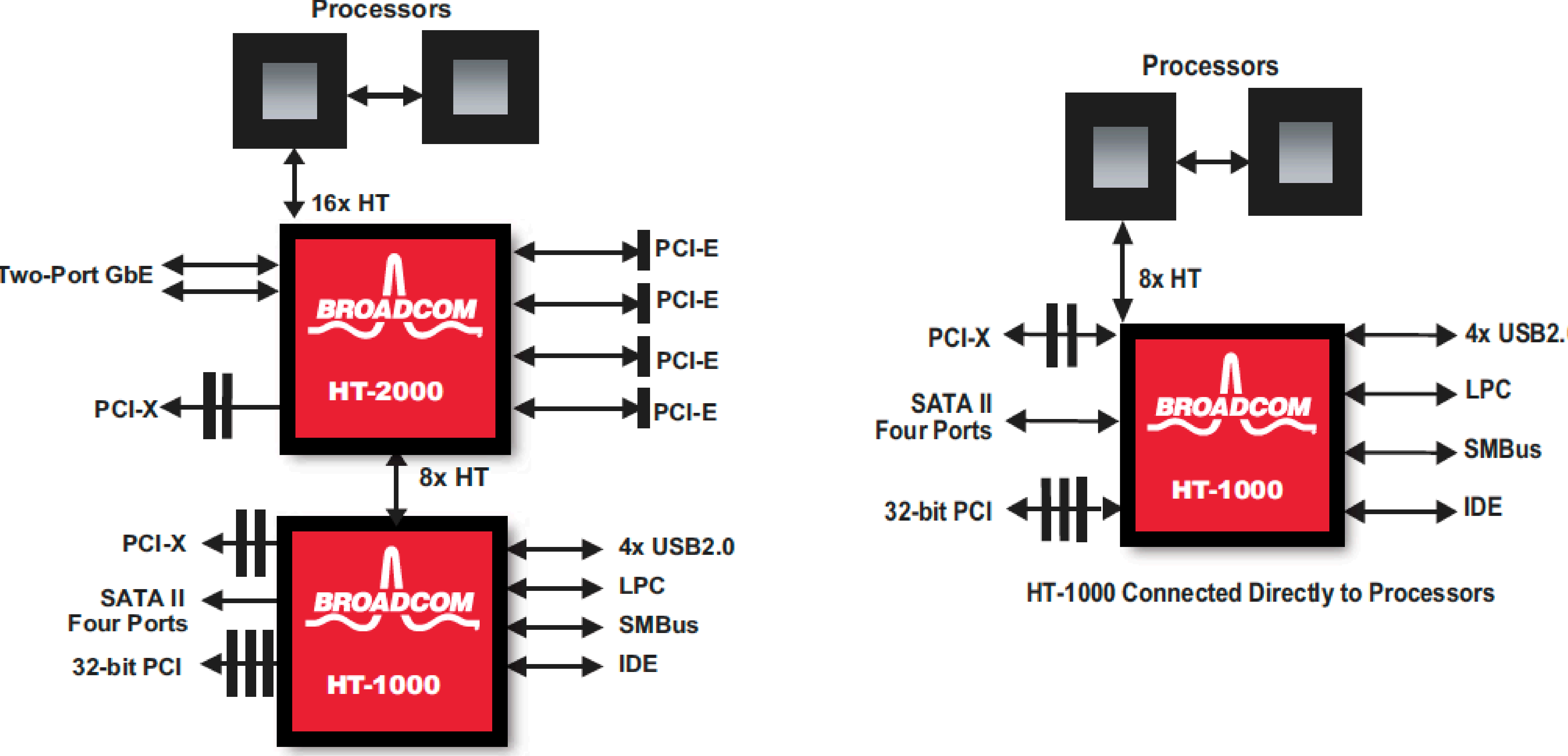


Figure 7-12. Write Data Timing Example

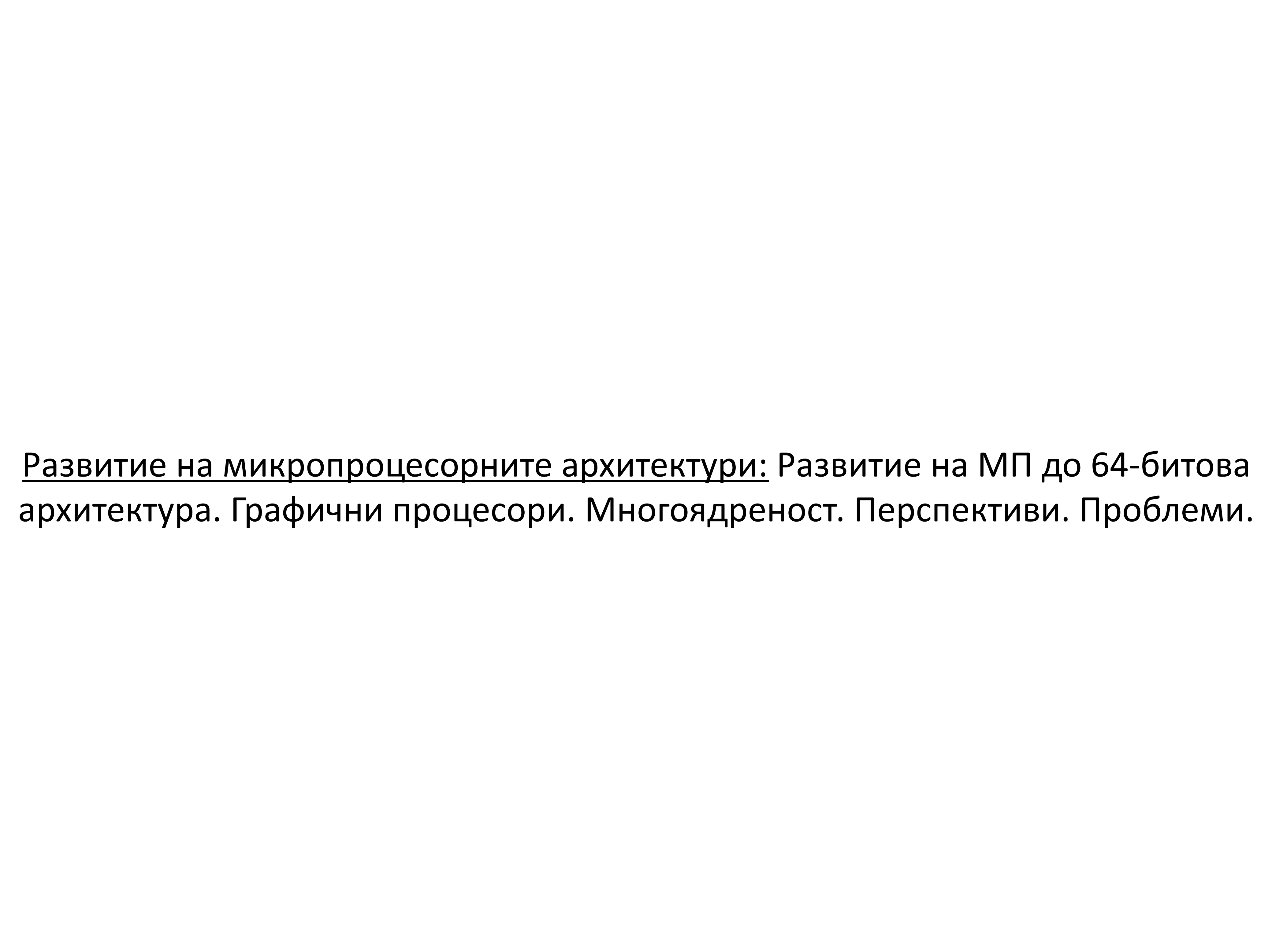



Processors
16x HT
Two-Port GbE
PCI-X
BROADCOM
HT-2000
PCI-E
PCI-E
PCI-E
PCI-E
8x HT
PCI-X
SATA II
Four Ports
32-bit PCI
BROADCOM
HT-1000
4x USB2.0
LPC
SMBus
IDE

Processors
16x HT
16x HT
17 x PCI-E
PCI-X
Dual GbE
BROADCOM
HT-2000
BROADCOM
HT-2000
17x PCI-E
PCI-X
Dual GbE
8x HT
PCI-X
SATA II
Four Ports
32-bit PCI
BROADCOM
HT-1000
4x USB2.0
LPC
SMBus
IDE



Two-Way Multiprocessing Server Configuration


Развитие на микропроцесорните архитектури: Развитие на МП до 64-битова архитектура. Графични процесори. Многоядреност. Перспективи. Проблеми.

Result Details

Your fastest score for Firefox 8.0a1

7009 Points

32-bit

Suite	Result
Rendering	4591
Social networking	4426
Complex graphics	21600
Data	12461
DOM operations	5598
Text parsing	11935

Result Details

Your fastest score for Firefox 8.0a1

7847 Points

64-bit

Suite	Result
Rendering	4654
Social networking	5933
Complex graphics	22718
Data	11905
DOM operations	7295
Text parsing	12409

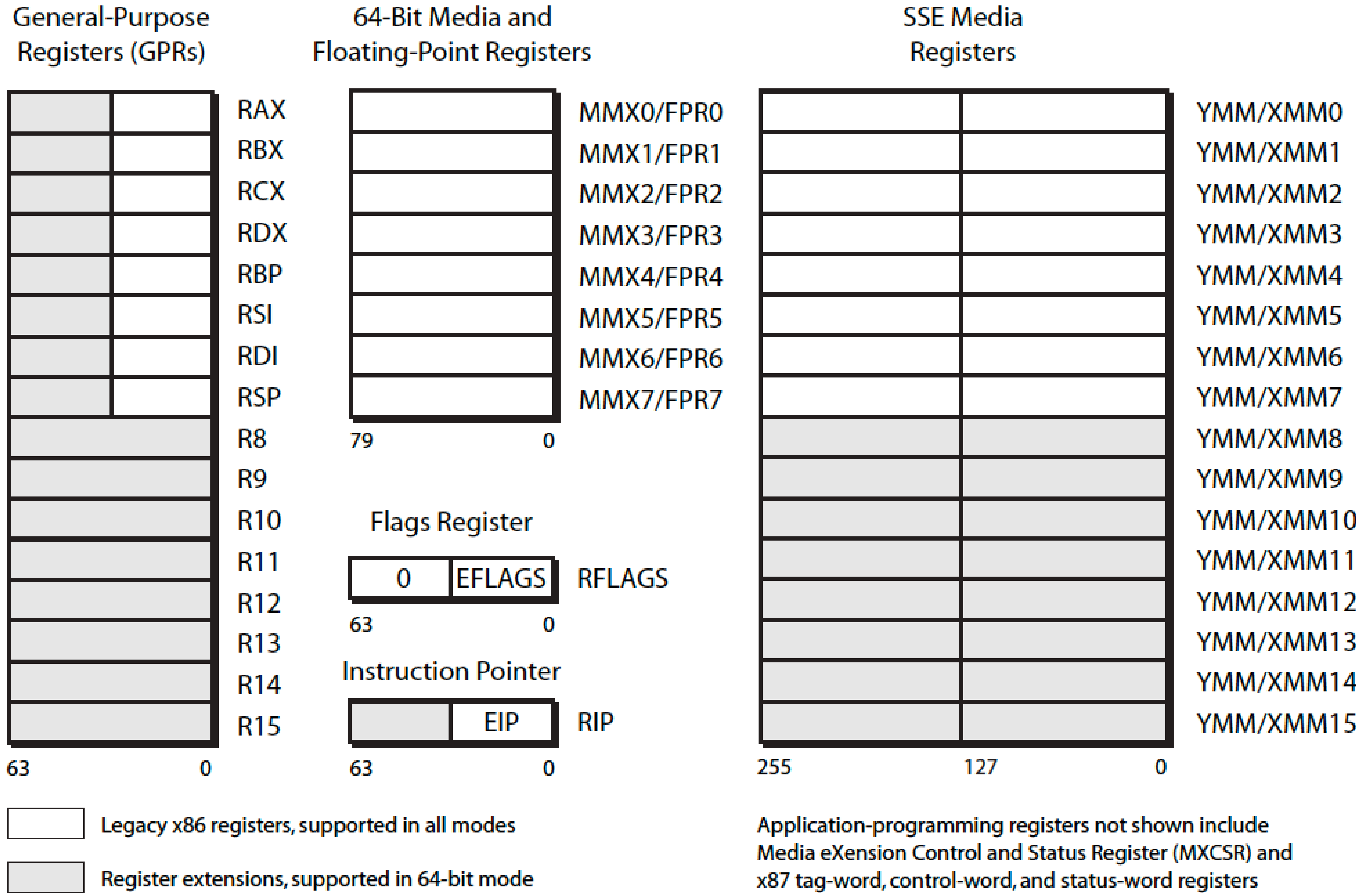


Figure 1-1. Application-Programming Register Set

Table 1-1. Operating Modes

Operating Mode		Operating System Required	Application Recompile Required	Defaults		Register Extensions	Typical
				Address Size (bits)	Operand Size (bits)		GPR Width (bits)
Long Mode	64-Bit Mode	64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		16
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		16
	Real Mode	Legacy 16-bit OS					

Table 1-2. Application Registers and Stack, by Operating Mode

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode[1]		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)[2]	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
256-bit YMM Registers	YMM0–YMM7[3]	8	256	YMM0–YMM15[3]	16	256
128-Bit XMM Registers	XMM0–XMM7[3]	8	128	XMM0–XMM15[3]	16	128
64-Bit MMX Registers	MMX0–MMX7[4]	8	64	MMX0–MMX7[4]	8	64
x87 Registers	FPR0–FPR7[4]	8	80	FPR0–FPR7[4]	8	80
Instruction Pointer[2]	EIP	1	32	RIP	1	64
Flags[2]	EFLAGS	1	32	RFLAGS	1	64
Stack	—		16 or 32	—		64

Note:

1. *Gray-shaded entries indicate differences between the modes. These differences (except stack-width difference) are the AMD64 architecture's register extensions.*
2. *GPRs are listed using their full-width names. In legacy and compatibility modes,16-bit and 8-bit mappings of the registers are also accessible. In 64-bit mode, 32-bit, 16-bit, and 8-bit mappings of the registers are accessible. See Section 3.1. "Registers" on page 23.*
3. *The XMM registers overlay the lower octword of the YMM registers. See Section 4.2. "Registers" on page 111.*
4. *The MMX0–MMX7 registers are mapped onto the FPR0–FPR7 physical registers, as shown in Figure 1-1. The x87 stack registers, ST(0)–ST(7), are the logical mappings of the FPR0–FPR7 physical registers.*

6.1 Overview of ARM's 64-bit Cortex-A series (6)

Main features of ARM's 64-bit Cortex-A series [51]

CPU Core	Architecture	Efficiency	big.LITTLE	Announced	Available in devices	Target
Cortex-A73	ARMv8 (64-bit)	7.4-8.5 DMIPS/MHz	Yes (with A53/A35)	2016	2017	High-end
Cortex-A72	ARMv8 (64-bit)	6.3-7.3 DMIPS/MHz	Yes (with A53/A35)	2015	2016	High-end
Cortex-A57	ARMv8 (64-bit)	4,8 DMIPS/MHz	Yes (with A53)	2012	2015	High-end
Cortex-A53	ARMv8 (64-bit)	2,3 DMIPS/MHz	Yes (with A57)	2012	2H 2014	Low power
Cortex-A35	ARMv8 (64-bit)	2,1 DMIPS/MHz	Yes (with A57/ A72)	2015	2H 2016	Low power
Cortex-A17	ARMv7 (32-bit)	4,0 DMIPS/MHz	Yes (with A7)	2014	2015	Mainstream
Cortex-A15	ARMv7 (32-bit)	4,0 DMIPS/MHz	Yes (with A7)	2010	Now	High-end
(Cortex-A12	ARMv7 (32-bit)	3,0 DMIPS/MHz	-	2013	2H 2015	Mainstream)
Cortex-A9	ARMv7 (32-bit)	2,5 DMIPS/MHz	-	2007	Now (EOL)	High-end
Cortex-A8	ARMv7 (32-bit)	2,0 DMIPS/MHz	-	2005	Now (EOL)	High-end
Cortex-A7	ARMv7 (32-bit)	1,9 DMIPS/MHz	Yes (A15/A17)	2011	Now	Low power
Cortex-A5	ARMv7 (32-bit)	1,6 DMIPS/MHz	-	2009	Now	Low power

Performance comparison: ARM’s Cortex-A72 vs. Intel’s Core-M [72]

Single-thread | Multi-thread* | Memory

Core-M (14nm)
Cortex-A72 2.5 GHz est (16nm)

4W
<IW

1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0

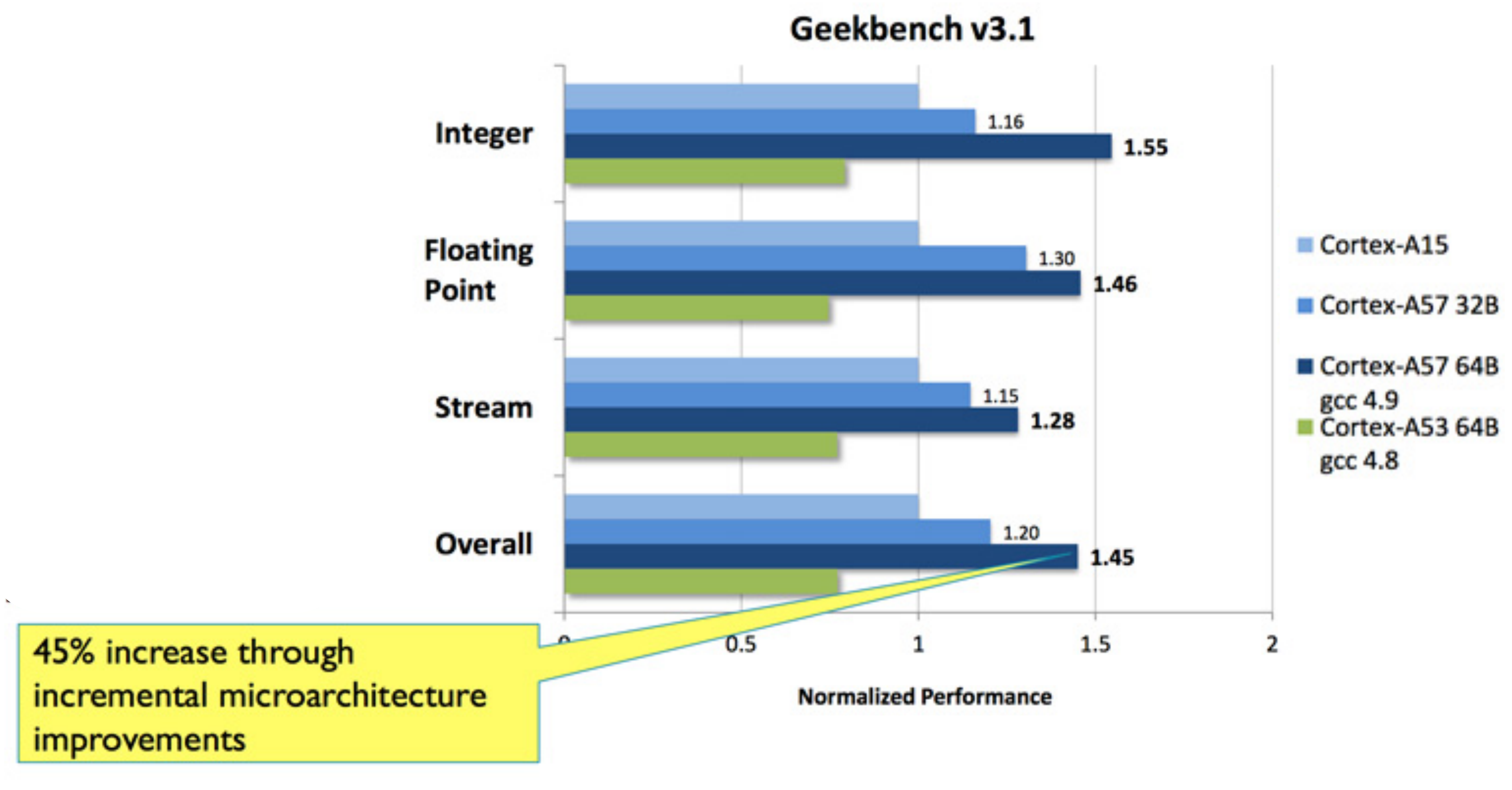
Geekbench ST | SPECint | SPECfp | Geekbench MT (4T) | SPECintRate (4T) | STREAM Add | STREAM Copy | STREAM Scale | STREAM Triad

- Intel workloads measured on Dell Venue Pro II. SPEC benchmarks measured using gcc compiler v4.9 with –o3 flag.
- Cortex-A72 measured on RTL with realistic memory system with gcc compiler v4.9 - o3 settings.
- Multi-threaded workloads use 2C4T Core-M CPU and estimated on 4C Cortex-A72 configuration w/2MB L2 cache.
- Core-M 5Y10C has maximum rated frequency rating of 2GHz. (Source: ark.intel.com)

* For mult-threaded workloads, the Core-M will be thermally limited and not able to reach maximum target frequency.

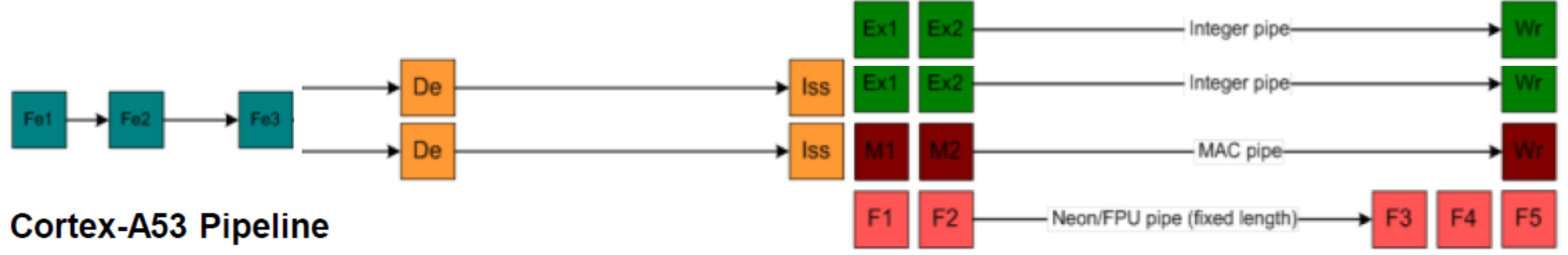
59

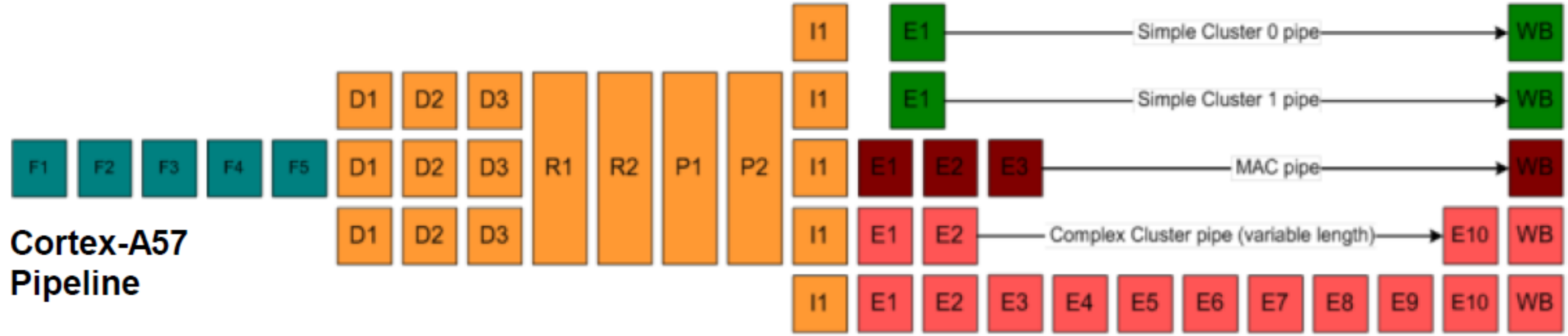
Cortex-A57/A53 performance - compared to the Cortex-A15 [55]



Contrasting the Cortex-A53 and Cortex-A57 arithmetic pipelines

[Based on 54]

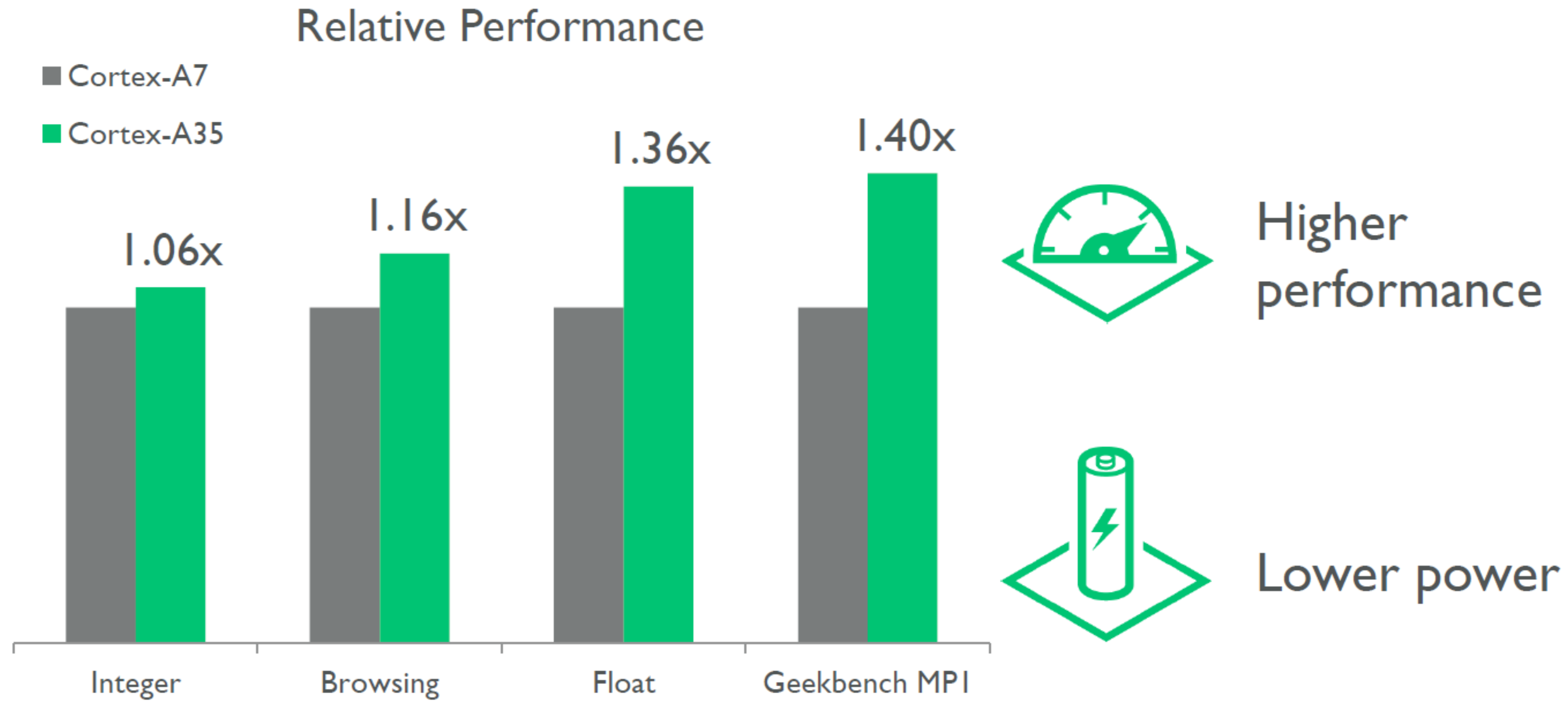




D: Decode
R: Rename
P: Dispatch
I: Issue
E: Execute
WB: Write Back

Note: Branch and Load/Store pipelines not shown
(1x Load/Store pipeline for the Cortex A-53 and
2x Load/Store and 1x Branch pipeline for the Cortex-A-57)

Relative performance of the Cortex-A35 vs. the Cortex-A7 assuming the same process technology (28 nm) [90]



The change from 32-bit to 64-bit

There are several performance gains derived from moving to a 64-bit processor.

- The A64 instruction set provides some significant performance benefits, including a larger register pool. The additional registers and the *ARM Architecture Procedure Call Standard* (AAPCS) provide a performance boost when you must pass more than four registers in a function call. On ARMv7, this would require using the stack, whereas in AArch64 up to eight parameters can be passed in registers.
- Wider integer registers enable code that operates on 64-bit data to work more efficiently. A 32-bit processor might require several operations to perform an arithmetic operation on 64-bit data. A 64-bit processor might be able to perform the same task in a single operation, typically at the same speed required by the same processor to perform a 32-bit operation. Therefore, code that performs many 64-bit sized operations is significantly faster.
- 64-bit operation enables applications to use a larger virtual address space. While the *Large Physical Address Extension* (LPAE) extends the physical address space of a 32-bit processor to 40-bit, it does not extend the virtual address space. This means that even with LPAE, a single application is limited to a 32-bit (4GB) address space. This is because some of this address space is reserved for the operating system.
- Software running on a 32-bit architecture might need to map some data in or out of memory while executing. Having a larger address space, with 64-bit pointers, avoids this problem. However, using 64-bit pointers does incur some cost. The same piece of code typically uses more memory when running with 64-pointers than with 32-bit pointers. Each pointer is stored in memory and requires eight bytes instead of four. This might sound trivial, but can add up to a significant penalty. Furthermore, the increased usage of memory space associated with a move to 64-bits can cause a drop in the number of accesses that hit in the cache. This in turn can reduce performance.

  The larger virtual address space also enables memory-mapping larger files. This is the mapping of the file contents into the memory map of a thread. This can occur even though the physical RAM might not be large enough to contain the whole file.

Registers in AArch64 state

In the AArch64 application level view, an ARM processing element has:

R0-R30 31 general-purpose registers, R0 to R30. Each register can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

See the register name mapping in Figure B1-1.

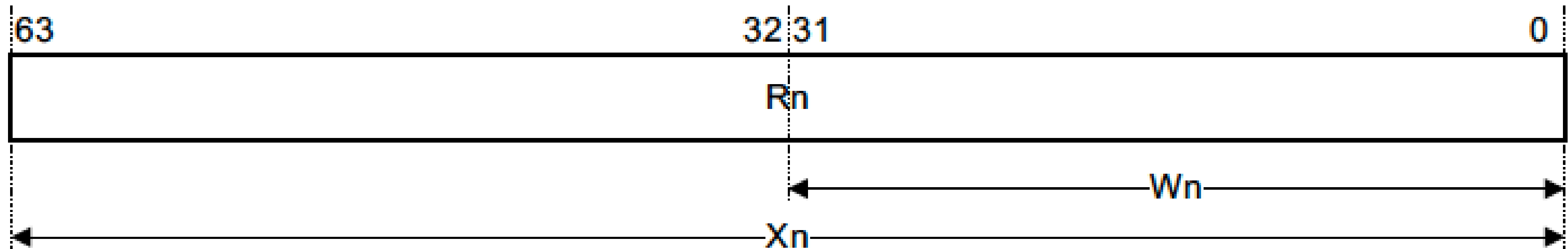


Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

Note

In instruction encodings, the value `0b11111` (31) is used to indicate the ZR (zero register). This indicates that the argument takes the value zero, but does not indicate that the ZR is implemented as a physical register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack-pointer can be accessed via the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information see the *Procedure Call Standard for the ARM 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can only be updated on a branch, exception entry or exception return.

Type	Mnemonic	Instruction
Arithmetic Regster	**ADD**	Add
	ADDS	Add and set flags
	SUB	Subtract
	SUBS	Subtract and set flags
	CMP	Compare
	CMN	Compare negative
	NEG	Negate
	NEGS	Negate and set flags
Arithmetic Immediate	**ADDI**	Add Immediate
	ADDIS	Add and set flags Immediate
	SUBI	Subtract Immediate
	SUBIS	Subtract and set flags Immediate
	CMPI	Compare Immediate
	CMNI	Compare negative Immediate
Arithmetic Extended	ADD	Add Extended Register
	ADDS	Add and set flags Extended
	SUB	Subtract Extended Register
	SUBS	Subtract and set flags Extended
	CMP	Compare Extended Register
	CMN	Compare negative Extended
Arithmetic with Carry	ADC	Add with carry
	ADCS	Add with carry and set flags
	SBC	Subtract with carry
	SBCS	Subtract with carry and set flags
	NGC	Negate with carry
	NGCS	Negate with carry and set flags
Logical Regster	**AND**	Bitwise AND
	ANDS	Bitwise AND and set flags
	ORR	Bitwise inclusive OR
	EOR	Bitwise exclusive OR
	BIC	Bitwise bit clear
	BICS	Bitwise bit clear and set flags
	ORN	Bitwise inclusive OR NOT
	EON	Bitwise exclusive OR NOT
	MVN	Bitwise NOT
	TST	Test bits

Type	Mnemonic	Instruction
Logical Immediate	**ANDI**	Bitwise AND Immediate
	ANDIS	Bitwise AND and set flags Immediate
	ORRI	Bitwise inclusive OR Immediate
	EORI	Bitwise exclusive OR Immediate
	TSTI	Test bits Immediate
Shift Register Shift Immed	**LSL**	Logical shift left Immediate
	LSR	Logical shift right Immediate
	ASR	Arithmetic shift right Immediate
	ROR	Rotate right Immediate
	LSRV	Logical shift right register
	LSLV	Logical shift left register
	ASRV	Arithmetic shift right register
	RORV	Rotate right register
Move Wide Immed iate	**MOVZ**	Move wide with zero
	MOVK	Move wide with keep
	MOVN	Move wide with NOT
	MOV	Move register
Bit Field Insert & Extract	BFM	Bitfield move
	SBFM	Signed bitfield move
	UBFM	Unsigned bitfield move (32-bit)
	BFI	Bitfield insert
	BFXIL	Bitfield extract and insert low
	SBFIZ	Signed bitfield insert in zero
	SBFX	Signed bitfield extract
	UBFIZ	Unsigned bitfield insert in zero
	UBFX	Unsigned bitfield extract
	EXTR	Extract register from pair
Sign Extend	*SXTB*	Sign-extend byte
	SXTH	Sign-extend halfword
	SXTW	Sign-extend word
	UXTB	Unsigned extend byte
	UXTH	Unsigned extend halfword
Bit Operation	CLS	Count leading sign bits
	CLZ	Count leading zero bits
	RBIT	Reverse bit order
	REV	Reverse bytes in register
	REV16	Reverse bytes in halfwords
	REV32	Reverses bytes in words

Type	Mnemonic	Instruction
Unscaled	**LDUR**	Load register (unscaled offset)
	LDURB	Load byte (unscaled offset)
	LDURSB	Load signed byte (unscaled offset)
	LDURH	Load halfword (unscaled offset)
	LDURSH	Load signed halfword (unscaled offset)
	LDURSW	Load signed word (unscaled offset)
	STUR	Store register (unscaled offset)
	STURB	Store byte (unscaled offset)
	STURH	Store halfword (unscaled offset)
	STURW	Store word (unscaled offset)
	LDA	Load address
Scaled, Extended, Pre-& Post-Indexed	LDR	Load register
	LDRB	Load byte
	LDRSB	Load signed byte
	LDRH	Load halfword
	LDRSH	Load signed halfword
	LDRSW	Load signed word
	STR	Store register
	STRB	Store byte
	STRH	Store halfword

Type	Mnemonic	Instruction
Exclusive	**LDXR**	Load Exclusive register
	LDXRB	Load Exclusive byte
	LDXRH	Load Exclusive halfword
	LDXP	Load Exclusive Pair
	STXR	Store Exclusive register
	STXRB	Store Exclusive byte
	STXRH	Store Exclusive halfword
	STXP	Store Exclusive Pair
Exclusive Aquire/Release	LDAXR	Load-aquire Exclusive register
	LDAXRB	Load-aquire Exclusive byte
	LDAXRH	Load-aquire Exclusive halfword
	LDAXP	Load-aquire Exclusive Pair
	STLXR	Store-release Exclusive register
	STLXRB	Store-release Exclusive byte
	STLXRH	Store-release Exclusive halfword
	STLXP	Store-release Exclusive Pair
Pair	LDP	Load Pair
	LDPSW	Load Pair signed words
	STP	Store Pair
PC	ADRP	Compute address of 4KB page at a PC-relative offset
	ADR	Compute address of label at a PC-relative offset

FIGURE 2.42 The list of assembly language instructions for the integer data transfer operations in the full ARMv8 instruction set. Bold means the instruction is also in LEGv8, italic means it is a pseudoinstruction, and bold italic means it is a pseudoinstruction that is also in LEGv8.

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Conditional Branch	**B.cond**	Branch conditionally	Conditional Select	CSEL	Conditional select
	CBNZ	Compare and branch if nonzero		CSINC	Conditional select increment
	CBZ	Compare and branch if zero		CSINV	Conditional select inversion
	TBNZ	Test bit and branch if nonzero		CSNEG	Conditional select negation
	TBZ	Test bit and branch if zero		*CSET*	Conditional set
Unconditional Branch	**B**	Branch unconditionally		*CSETM*	Conditional set mask
	BL	Branch with link		*CINC*	Conditional increment
	BLR	Branch with link to register		*CINV*	Conditional invert
	BR	Branch to register		*CNEG*	Conditional negate
	RET	Return from subroutine	Conditional Compare	CCMP	Conditional compare register
				CCMPI	Conditional compare immediate
				CCMN	Conditional compare negative register
				CCMNI	Conditional compare negative immediate

FIGURE 2.43 The list of assembly language instructions for the branches of the ARMv8 instruction set. Bold means the instruction is also in LEGv8, italic means it is a pseudoinstruction, and bold italic means it is a pseudoinstruction that is also in LEGv8.

Figure 4.1 CPU Registers for MIPS64

General Purpose Registers

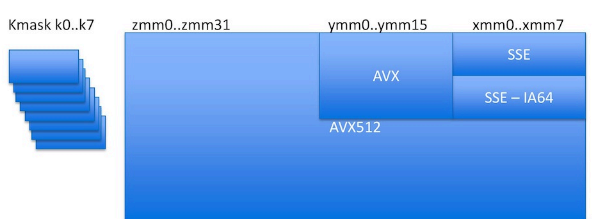
63	32 31	0
	r0 (hardwired to zero)	
	r1	
	r2	
	r3	
	r4	
	r5	
	r6	
	r7	
	r8	
	r9	
	r10	
	r11	
	r12	
	r13	
	r14	
	r15	
	r16	
	r17	
	r18	
	r19	
	r20	
	r21	
	r22	
	r23	
	r24	
	r25	
	r26	
	r27	
	r28	
	r29	
	r30	
	r31	

Special Purpose Registers

63	32 31	0
	HI	
	LO	

63	32 31	0
	PC	

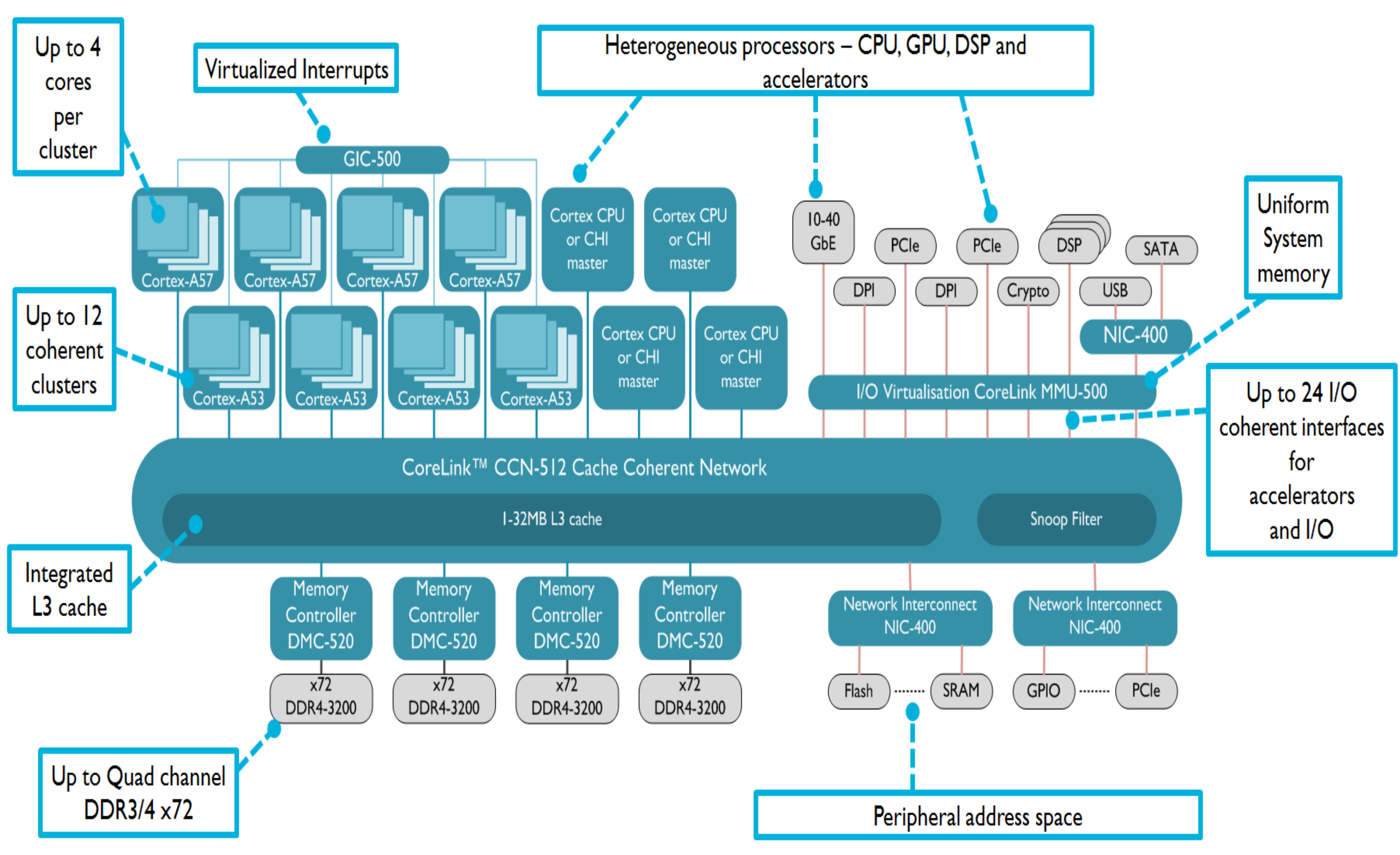
AVX512 state



High amounts of compute need large amounts of state to compensate for memory BW
AVX512 has 8x state compared to SSE (commensurate with its 8x flops level)

Intel confidential — presented under NDA only — under embargo until 6:01 a.m. PDT, June 19, 2017

Up to 48 core server SoC based on the CoreLink CCN512 interconnect [72]



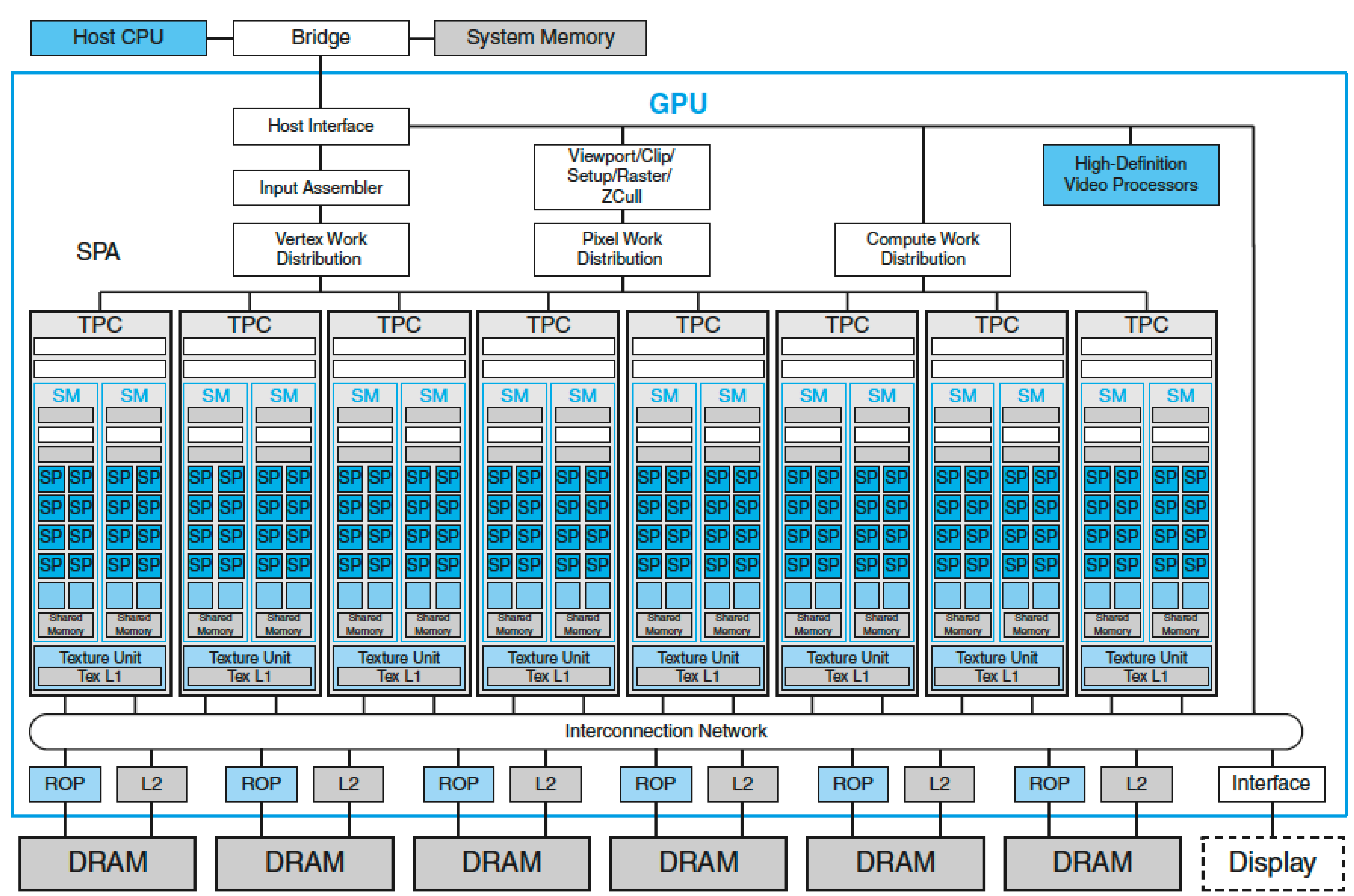
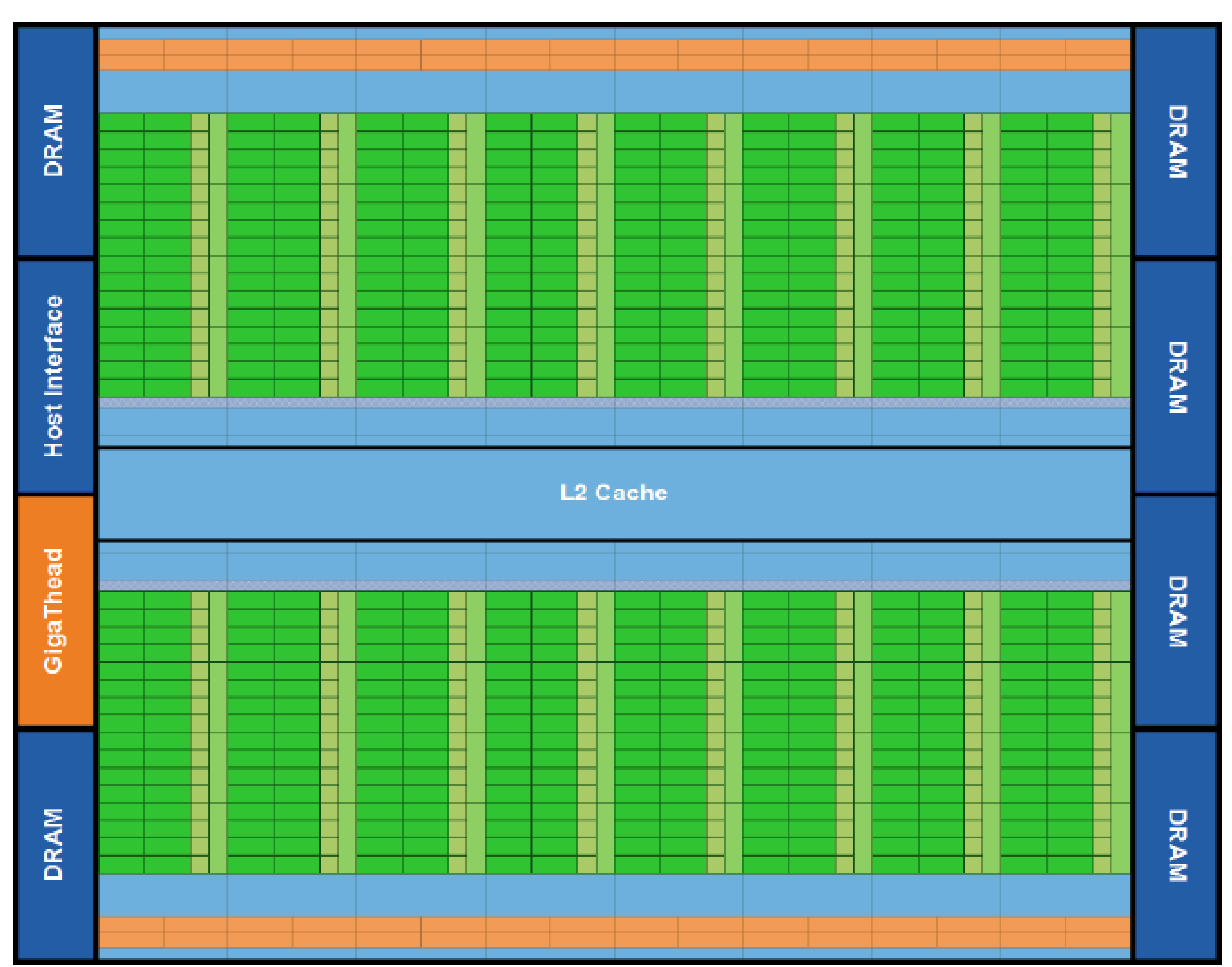


FIGURE B.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.



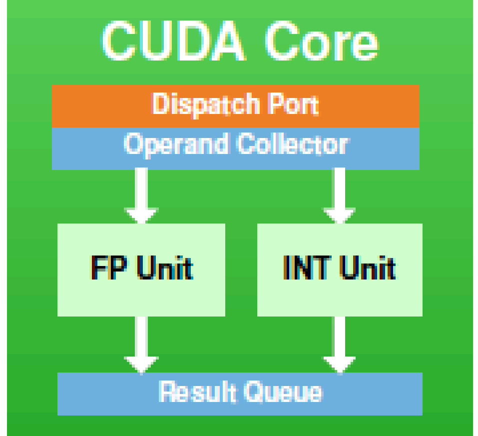
Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Third Generation Streaming Multiprocessor

The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

512 High Performance CUDA cores

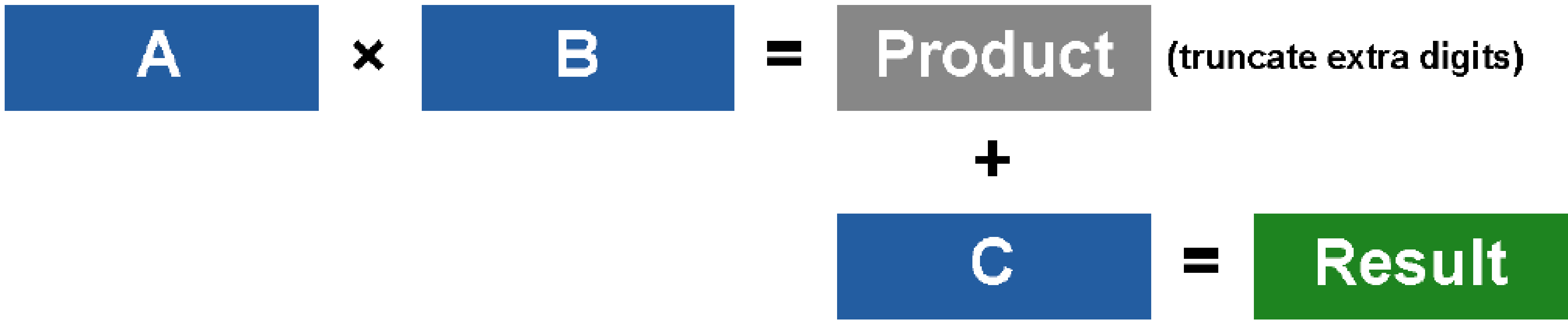
Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately. GT200 implemented double precision FMA.



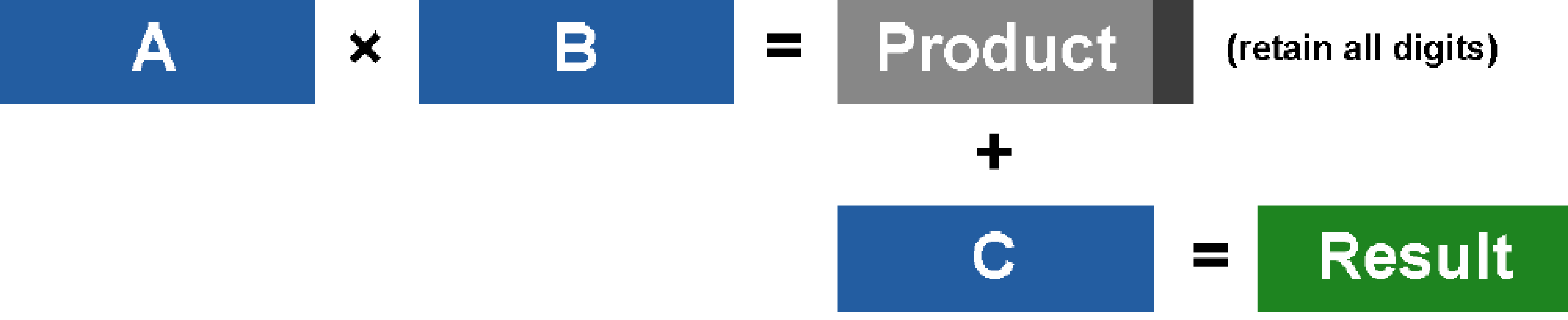
Instruction Cache
Warp Scheduler | Warp Scheduler
Dispatch Unit | Dispatch Unit
Register File (32,768 x 32-bit)
Core (x32)
LD/ST (x16)
SFU (x4)
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

Fermi Streaming Multiprocessor (SM)

Multiply-Add (MAD):

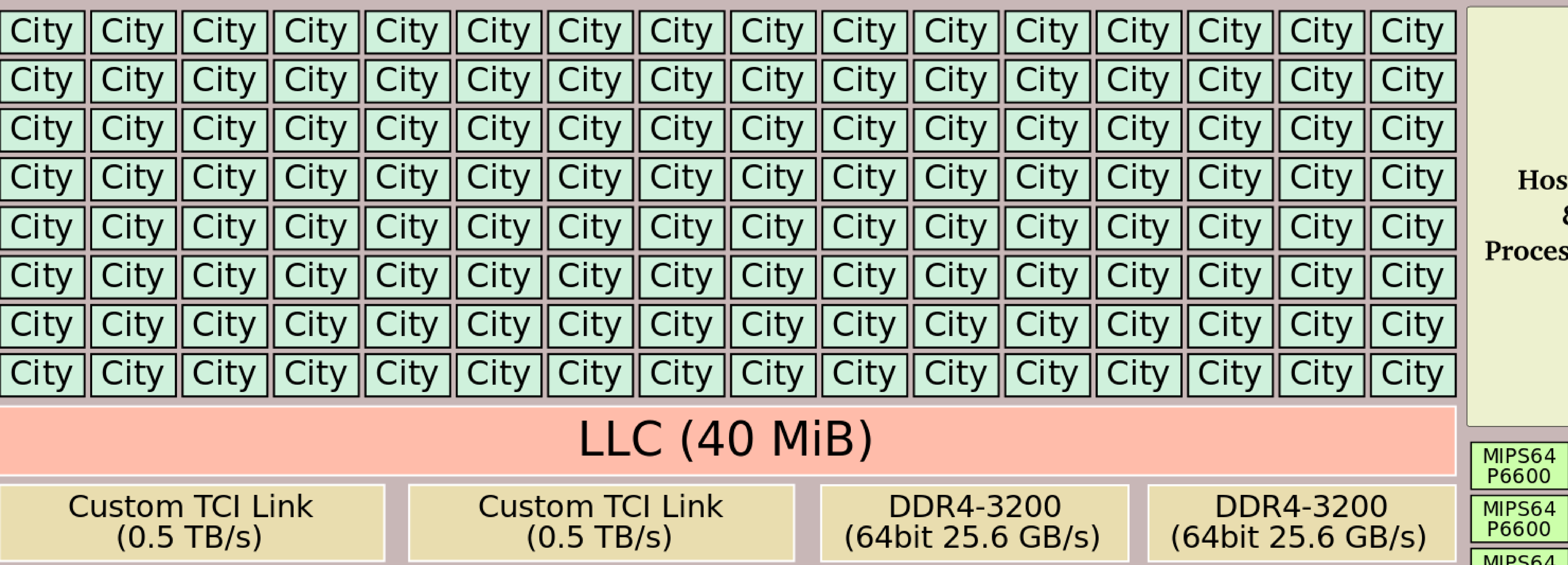


Fused Multiply-Add (FMA)



PEZY-SCx Processor Roadmap

	PEZY-SC	**PEZY-SC2**	PEZY-SC3	PEZY-SC4
Process	28nm	16nm	7nm	5nm
Die Size	412mm2	620mm2	700mm2	740mm2
Number of Cores	**1,024**	**2,048**	**8,096**	**16,192**
Core Voltage	**0.9V**	**0.8V**	**0.65V**	**0.55V**
Core Clock	733MHz	1GHz	1.33GHz	1.6GHz
DRAM-IO	DDR4	DDR4	DDR4/5	DDR5
DDR Clock	2,133MHz	2,666MHz	3.6GHz	4GHz
Port数	8	4	4	4
Wide-IO Clock	-	2GHz DDR	2GHz DDR	3GHz DDR
Wide-IO Width		1,024bit	3,072bit	4,096bit
Wide-IO Ports		4	8	8
Memory Bandwidth	**153.6GB/s**	**2.1TB/s**	**12.2TB/s**	**24.4TB/s**
Peripheral IO	PCI3e Gen3	PCIe Gen4	**Custom Optical**	**Custom Optical**
Peripheral IO lane	24	32	128	512
Peripheral IO Bandwidth	**32GB/s**	**64GB/s**	**256GB/s**	**1TB/s**
DP Performance	**1.5TFLOPS**	**4.1TFLOPS**	**21.8TFLOPS**	**52.5TFLOPS**
SP Performance	3.0TFLOPS	8.2TFLOPS	43.6TFLOPS	105TFLOPS
HP Performance	-	16.4TFLOPS	87.2TFLOPS	210TFLOPS
Power Consumption	100W	200W	400W	640W
Power Efficiency	15GFLOPS/w	20.5GFLOPS/w	54.5GFLOPS/w	82.0GFLOPS/w
System Efficiency	**6.7GFLOPS/w**	**15GFLOPS/w**	**40GFLOPS/w**	**60GFLOPS/w**


City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
City City City City City City City City City City City City City City City City
Host I/F
&
Processor I/F
LLC (40 MiB)
MIPS64 P6600
MIPS64 P6600
Custom TCI Link (0.5 TB/s)
Custom TCI Link (0.5 TB/s)
DDR4-3200 (64bit 25.6 GB/s)
DDR4-3200 (64bit 25.6 GB/s)
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
Custom TCI Link (0.5 TB/s)
Custom TCI Link (0.5 TB/s)
DDR4-3200 (64bit 25.6 GB/s)
DDR4-3200 (64bit 25.6 GB/s)
MIPS64 P6600
MIPS64 P6600

City

Special Function Unit

Village

Village

Village

Village

L2D$ (64 KiB)

Village

Processing Element

Processing Element

Processing Element

Processing Element

L1D$ (2 KiB)

L1D$ (2 KiB)

Processing Element

8x Program Counter

L1I$ (256W x 64-bit)
(2 KiB)

ALU
4 FLOP/cycle

Register File
(256W x 32-bit)
(1 KiB)

Local Storage
(4096W x 32-bit)
(16 KiB)

Кратки сведения за други МП: Условни преходи и пренос в МП без РКУ („Alpha“, MIPS) и с 2 РКУ (POWER). МП с „регистров прозорец“ (SPARC). Програми „Здравей, свят!“ за различни МП и операционни системи (ОС).

	Alpha	MIPS I	PA-RISC 1.1	PowerPC	SPARCv8
Date announced	1992	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	1	1	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	8 KB	4 KB	4 KB	4 KB	8 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 32 bits	31 GPR × 32 bits	32 GPR × 32 bits	31 GPR × 32 bits
Separate floating-point registers	31 × 32 or 31 × 64 bits	16 × 32 or 16 × 64 bits	56 × 32 or 28 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

FIGURE E.1.1 Summary of the first version of five architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure E.17.1. Later versions of these architectures all support a flat, 64-bit address space.

Addressing mode	Alpha	MIPS-64	PA-RISC 2.0	PowerPC	SPARCv9
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)		X (FP)	X (Loads)	X	X
Register + scaled register (scaled)			X		
Register + offset and update register			X	X	
Register + register and update register			X	X	

FIGURE E.2.1 Summary of data addressing modes supported by the desktop architectures. PA RISC also has short address versions of the offset addressing modes. MIPS-64 has indexed addressing for floating-point loads and stores. (These addressing modes are described in Figure 2.18.)

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (BEQ) or inequality (BNE), and then the branch is taken if the condition holds. The set on less than instructions (SLT, SLTI, SLTU, SLTIU) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare and branch instructions for all such comparisons: greater than or equal to zero (BGEZ), greater than zero (BGTZ), less than or equal to zero (BLEZ), and less than zero (BLTZ). Of course, equal and not equal to zero can be synthesized using r0 with BEQ and BNE. Like SPARC, MIPS I uses a condition code for floating point with separate floating-point compare and branch instructions; MIPS IV expanded this to eight floating-point condition codes, with the floating point comparisons and branch instructions specifying the condition to set or test.

Alpha compares (CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE) test two registers and set a third to 1 if the condition is true and to 0 otherwise. Floating-point compares (CMTEQ, CMTLT, CMTLE, CMTUN) set the result to 2.0 if the condition holds and to 0 otherwise. The branch instructions compare one register to 0 (BEQ, BGE, BGT, BLE, BLT, BNE) or its least significant bit to 0 (BLBC, BLBS) and then branch if the condition holds.

In the future, I'm going to write x to mean "L or Q", and Rb/#b to mean "a register (*Rb*) or a small constant in the range 0 to 255."

The Alpha AXP has no corresponding trap variant for arithmetic carry. So how would you detect carry?[1]

Answer: The same way you detect carry in C, or pretty much any other programming language that doesn't support carry.

To detect carry during addition, you check whether the sum is less than either addend. If the sum is less than one addend, then it will also be less than the other addend, so use whichever addend is most convenient.

```
; Rc = Ra + Rb, with Rd receiving carry
; Assumes Rc is not the same as Ra
ADDx    Ra, Rb, Rc      ; Rc = Ra + Rb
CMPULT  Ra, Rc, Rd      ; Rd = carry

; Rc = Ra + Rb, with Rd receiving carry
; Assumes Rc is not the same as Rb
ADDx    Ra, Rb, Rc      ; Rc = Ra + Rb
CMPULT  Rb, Rc, Rd      ; Rd = carry

; Rc = Rc + Rc, with Rd receiving carry
; Assumes Rd is distinct from Rc
BIS     Rd, Rc, Rc      ; Rd = Rc
ADDx    Rc, Rc, Rc      ; Rc = Rc + Rc
CMPULT  Rd, Rc, Rd      ; Rd = carry
```

The last case is where the output overwrites both inputs, so we have to stash one of the inputs in *Rd* so we can compare it to the result afterwards.

To detect borrow during subtraction, you check whether the subtrahend is greater than the minuend.

```
; Rc = Ra − Rb, with Rd receiving borrow
; Assumes Rd is distinct from both inputs
CMPULT  Ra, Rb, Rd      ; Rd = borrow
SUBx    Ra, Rb, Rc      ; Rc = Ra − Rb
```

Raymond Chen - MSFT August 22, 2017

3.3.3 64-Bit Addition and Subtraction

In some cases, the numbers being added or subtracted can be more than 32-bits long. Since general-purpose registers are only 32-bits wide, it is the job of the programmer (or the compiler) to write the code to break down large numbers into smaller chunks to be processed by the CPU. Figure 3–3 illustrates this. In Figure 3–3, r3 contains the upper 32 bits of a 64-bit constant, and r2 contains the lower 32 bits of that 64-bit constant. Likewise, r5 and r4 together contain a 64-bit constant.

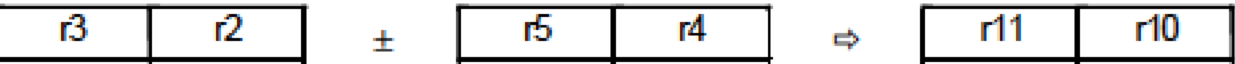


Figure 3-3 64-Bit Addition and Subtraction

Add with Carry

Below is an example of code to add two 64-bit constants together:

```
ADDU     r10,r2,r4   # r10 ← r2 + r4
SLTU     r11,r10,r2  # r11=1 if r10 (sum) is less than r2
ADD(U)   r11,r11,r3  # r11 ← r11 (carry) + r3
ADD(U)   r11,r11,r5  # r11 ← r11 + r5
```

The first ADDU instruction adds the lower 32 bits of two constants together and puts the result in r10. The TX19 architecture does not provide a flag bit to indicate whether an arithmetic operation results in a carry-out. Therefore, it is necessary to somehow record an occurrence of a carry-out resulting from an addition. For the sake of discussion, let's assume that the two operands are positive values. Then, based on the fact that if the sum is less than one of the operands added, a carry-out occurred, the next SLTU (Set on Less Than Unsigned) instruction sets r11 to 1 if r10 is less than r2. The following two ADD(U) instructions add the carry-out bit (1 or 0) and the upper 32 bits of the two 64-bit constants.

The last two instructions can be either ADD or ADDU. The only difference between these two instructions is that ADDU (Add Unsigned) never causes an integer overflow exception. When you use the ADDU instruction, you need to write the code to explicitly test for an occurrence of the overflow condition. This is discussed in the next section.

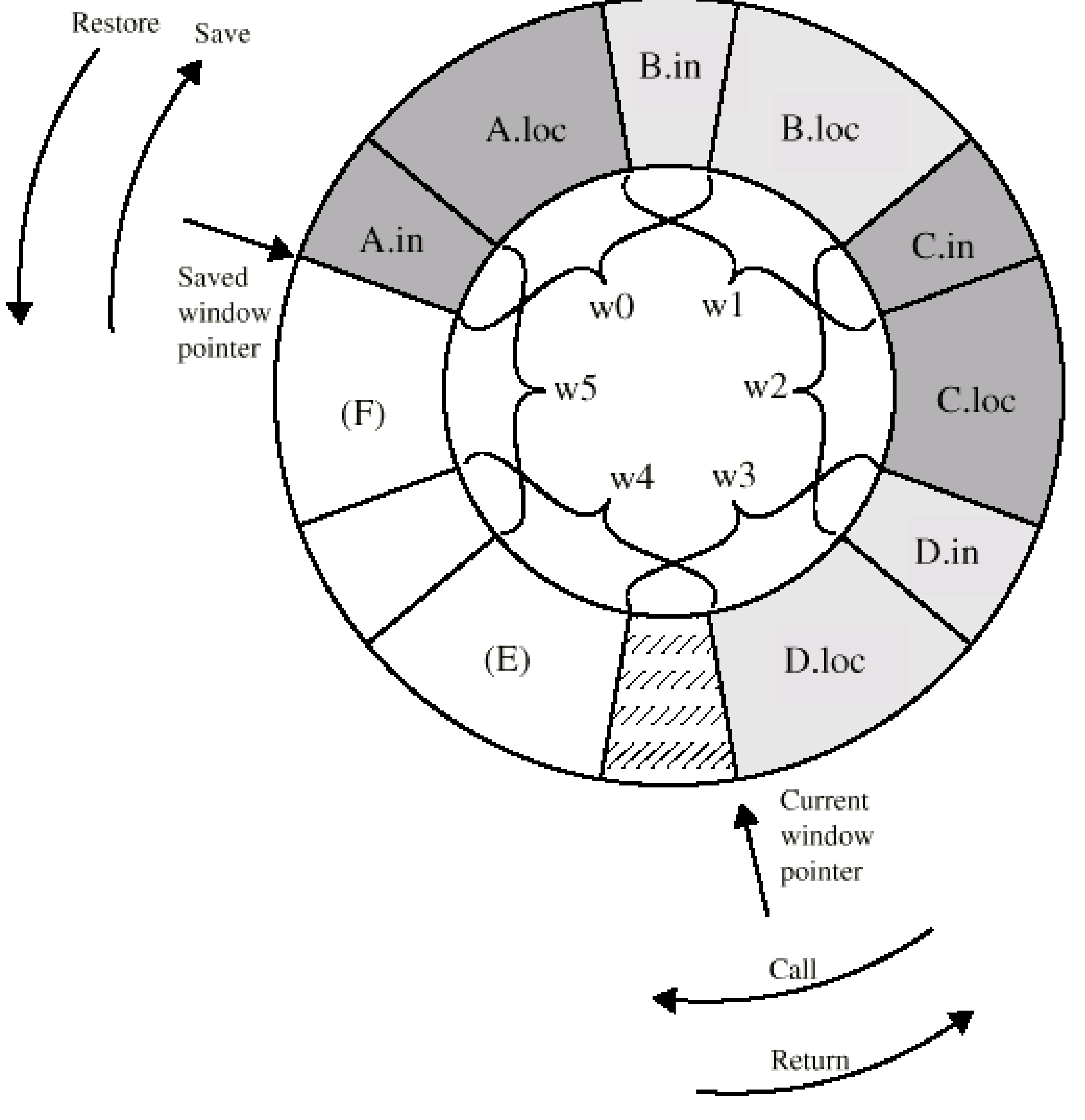
Subtract with Borrow

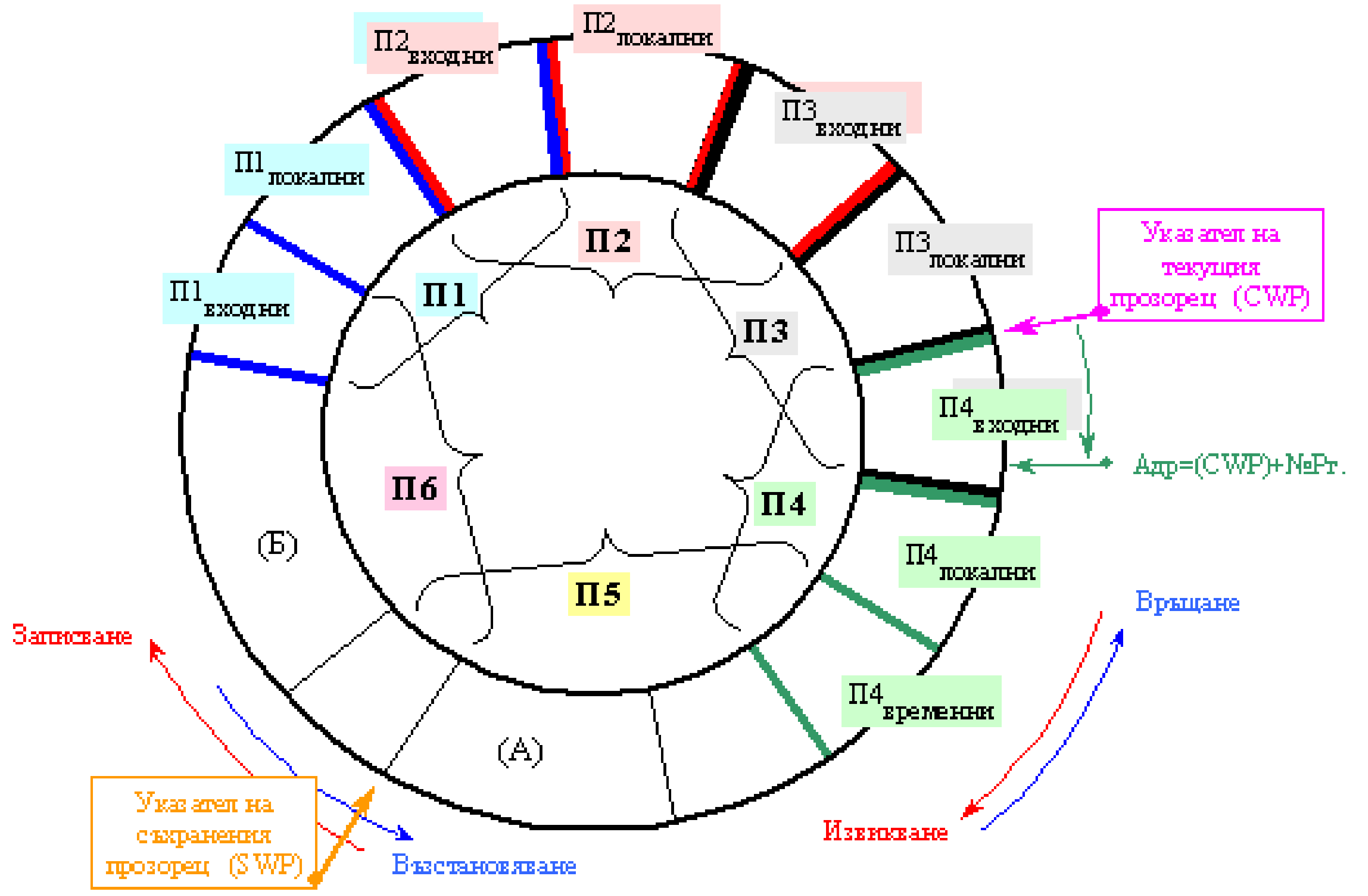
In 64-bit subtraction, the code must take care of the borrow of the lower operand. The technique for performing subtract-with-borrow is quite similar to add-with-carry. Below is an example of code to subtract a 64-bit constant from a 64-bit constant.

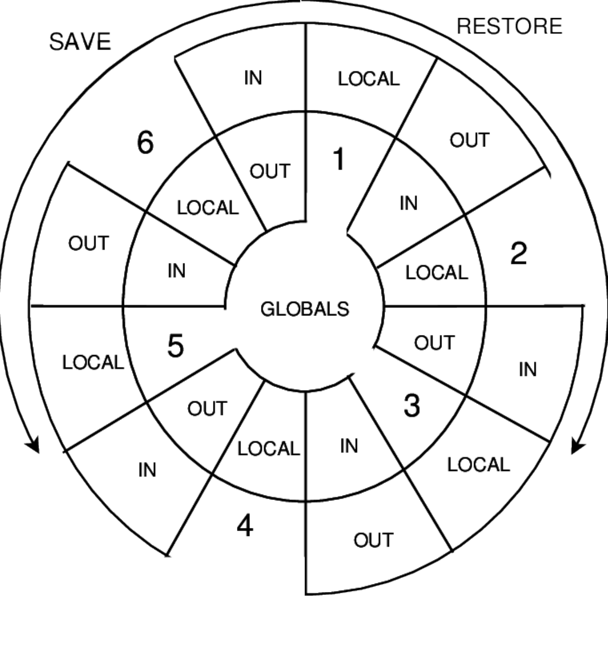
```
SLTU     r8,r2,r4    # r8=1 if r2 is less than r4
SUBU     r10,r2,r4   # r10 ← r2 - r4
SUB(U)   r11,r3,r5   # r11 ← r3 - r5
SUB(U)   r11,r11,r8  # r11 ← r11 - r8 (borrow)
```

First of all, the SLTU instruction checks if r2 (minuend) is smaller than r4 (subtrahend). If it is, r8 is set to 1. That is, if there is a borrow resulting from the subtraction of the lower 32 bits, its occurrence is recorded in r8. The content of r8 is subtracted in the last SUB(U) instruction.

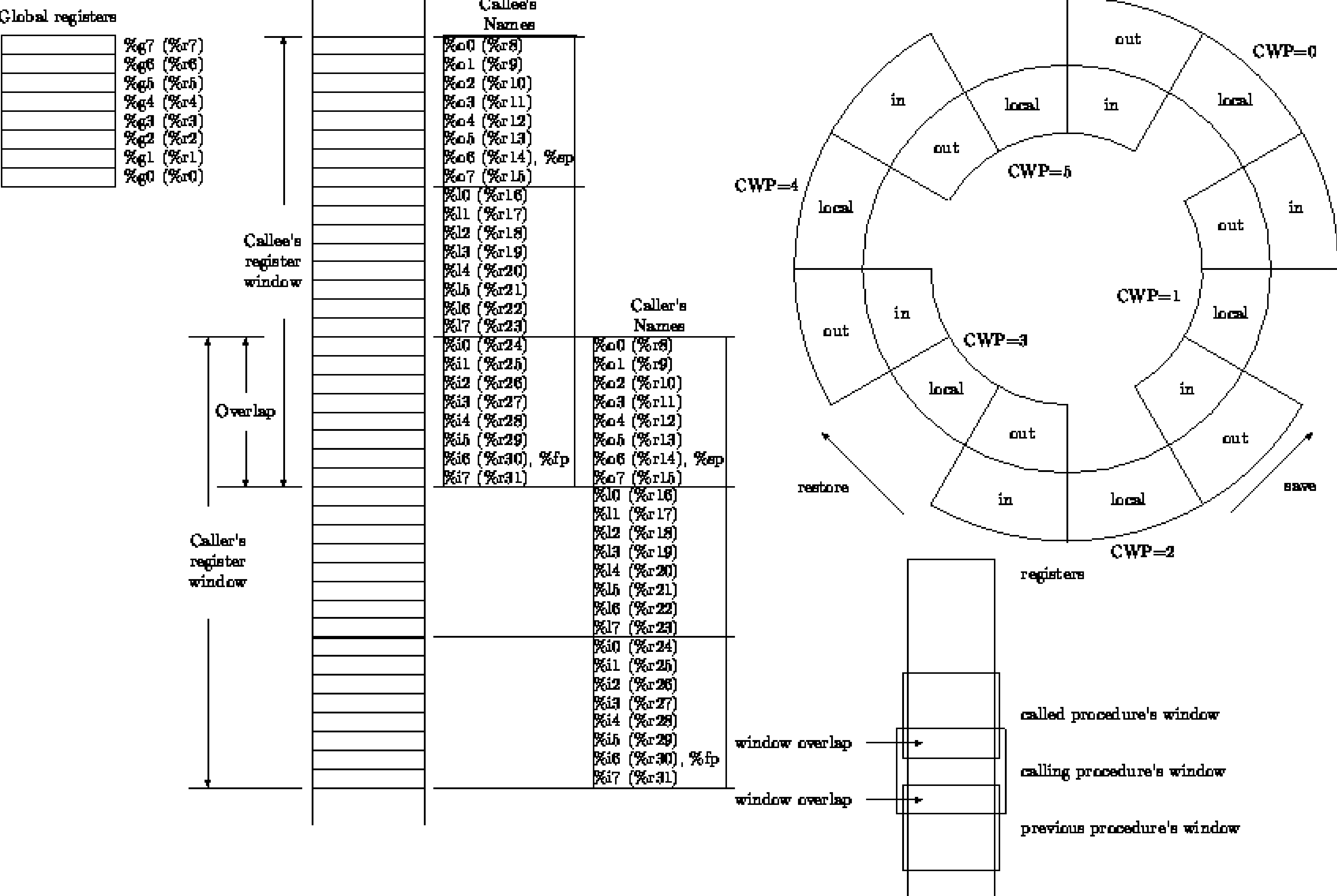
Again, the only difference between the SUB and SUBU instructions is that SUBU (Subtract Unsigned) never causes an integer overflow exception.

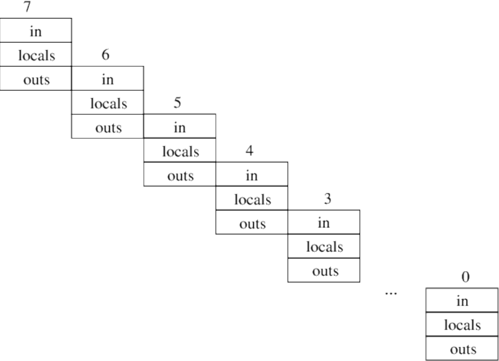

Restore
Save
B.in
A.loc
B.loc
A.in
C.in
Saved window pointer
w0
w1
w5
w2
(F)
C.loc
w4
w3
D.in
(E)
D.loc
Current window pointer
Call
Return


П2 входни
П2 локални
П3 входни
П1 локални
П2
П3 локални
Указател на текущия прозорец (CWP)
П1 входни
П1
П3
П4 входни
Адр=(CWP)+№Рг.
П6
П4
(Б)
П4 локални
П5
Връщане
Записване
П4 временни
(А)
Указател на съхранения прозорец (SWP)
Извикване
Възстановяване



SAVE
RESTORE
IN
LOCAL
OUT
1
OUT
LOCAL
6
IN
OUT
LOCAL
IN
2
GLOBALS
OUT
IN
LOCAL
5
OUT
3
LOCAL
IN
IN
LOCAL
4
OUT


Global registers
%g7 (%r7)
%g6 (%r6)
%g5 (%r5)
%g4 (%r4)
%g3 (%r3)
%g2 (%r2)
%g1 (%r1)
%g0 (%r0)
Callee's Names
%o0 (%r8)
%o1 (%r9)
%o2 (%r10)
%o3 (%r11)
%o4 (%r12)
%o5 (%r13)
%o6 (%r14), %sp
%o7 (%r15)
%l0 (%r16)
%l1 (%r17)
%l2 (%r18)
%l3 (%r19)
%l4 (%r20)
%l5 (%r21)
%l6 (%r22)
%l7 (%r23)
%i0 (%r24)
%i1 (%r25)
%i2 (%r26)
%i3 (%r27)
%i4 (%r28)
%i5 (%r29)
%i6 (%r30), %fp
%i7 (%r31)
Callee's register window
Overlap
Caller's register window
Caller's Names
%o0 (%r8)
%o1 (%r9)
%o2 (%r10)
%o3 (%r11)
%o4 (%r12)
%o5 (%r13)
%o6 (%r14), %sp
%o7 (%r15)
%l0 (%r16)
%l1 (%r17)
%l2 (%r18)
%l3 (%r19)
%l4 (%r20)
%l5 (%r21)
%l6 (%r22)
%l7 (%r23)
%i0 (%r24)
%i1 (%r25)
%i2 (%r26)
%i3 (%r27)
%i4 (%r28)
%i5 (%r29)
%i6 (%r30), %fp
%i7 (%r31)
CWP=0
CWP=1
CWP=2
CWP=3
CWP=4
CWP=5
in
local
out
restore
save
registers
called procedure's window
window overlap
calling procedure's window
window overlap
previous procedure's window


7
in
locals
outs
6
in
locals
outs
5
in
locals
outs
4
in
locals
outs
3
in
locals
outs
...
0
in
locals
outs

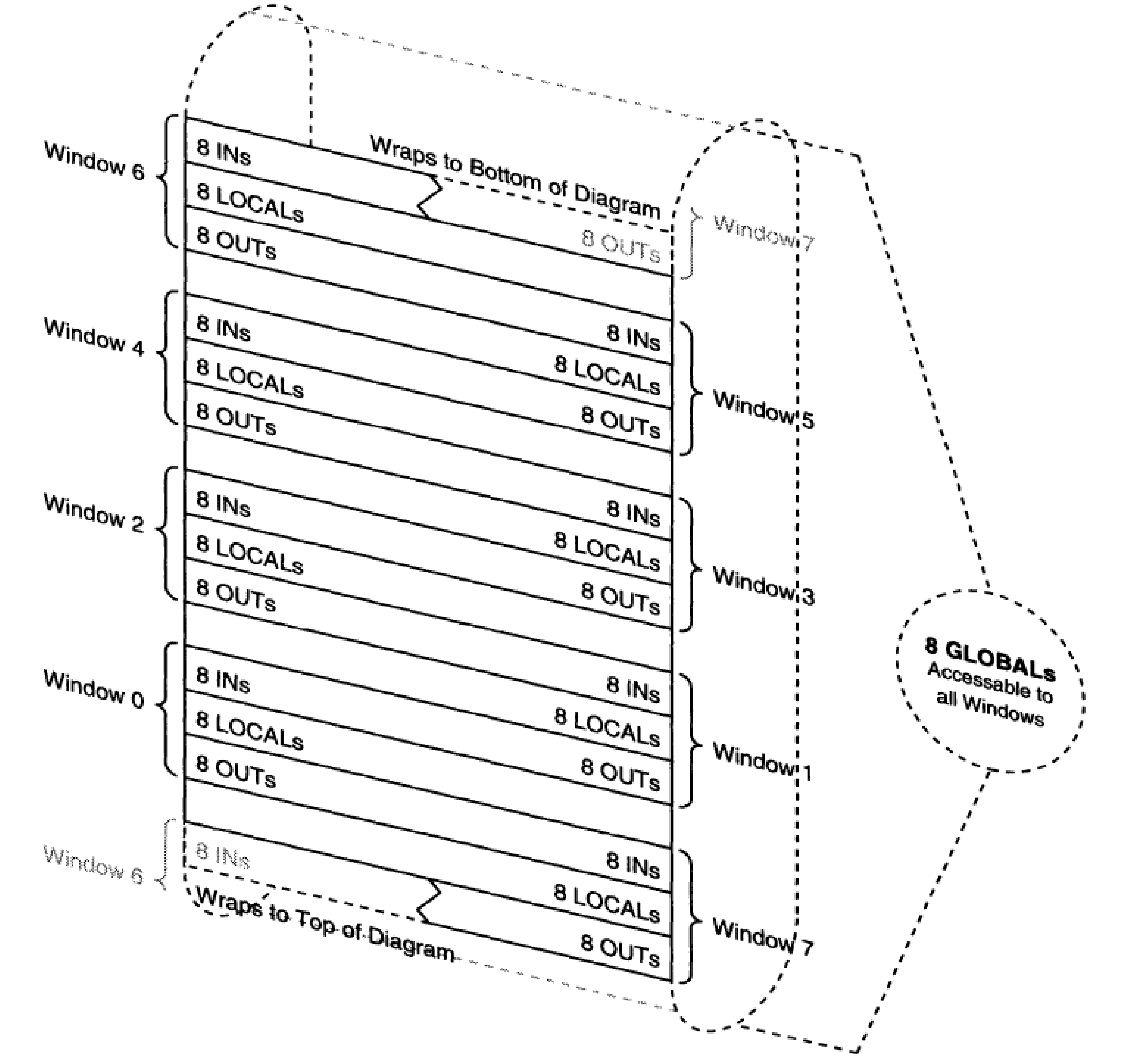
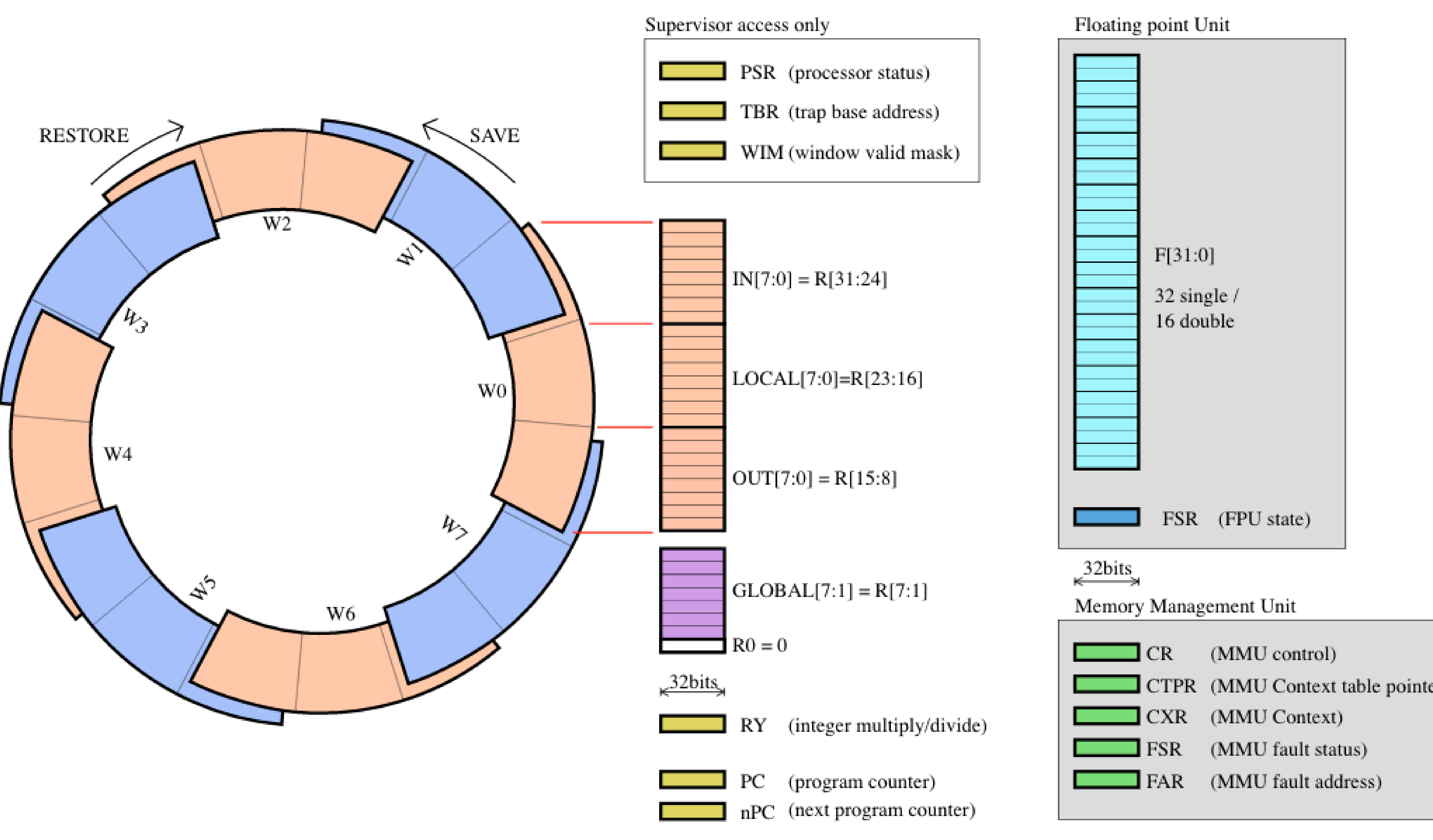


Figure 2-2. Register Windows


Supervisor access only
PSR (processor status)
TBR (trap base address)
WIM (window valid mask)
RESTORE
SAVE
W0
W1
W2
W3
W4
W5
W6
W7
IN[7:0] = R[31:24]
LOCAL[7:0]=R[23:16]
OUT[7:0] = R[15:8]
GLOBAL[7:1] = R[7:1]
R0 = 0
32bits
RY (integer multiply/divide)
PC (program counter)
nPC (next program counter)
Floating point Unit
F[31:0]
32 single /
16 double
FSR (FPU state)
32bits
Memory Management Unit
CR (MMU control)
CTPR (MMU Context table pointe
CXR (MMU Context)
FSR (MMU fault status)
FAR (MMU fault address)

chipdb.org

Програми „Здравей, свят!“

Следват 27 програми „Здравей, свят“ за 16 различни микропроцесорни архитектури и 19 различни операционни системи, повечето написани от автора (други частично взаимствани от други автори) и изпробвани лично от него на реални компютри (не емулатори). Едноименните архитектури с различна разрядност (16, 32 и 64 бита) се броят за различни поради голямата разлика в зареждането на адреса на низа, начина на извикване на ядрото или системата от команди. В програмите не са използвани никакви външни обектни файлове или библиотеки.

hellodos.s   Програма „Здравей, свят!“ за 8086+ на MS-DOS (NASM)

```
org     0x100
        MOV     AH,9
        MOV     DX,MSG
        INT     0x21
        RET
MSG:    DB      "Hello, world!",13,10,'$'
```

```
; nasm -f obj helloos2.s
; link386 /pm:vio helloos2,,nul,os2386;

segment class=code use32 flat
extern  Dos32Write,Dos32Exit
..start:
        PUSH    WRITTEN
        PUSH    LEN
        PUSH    MSG
        PUSH    1
        CALL    Dos32Write
        PUSH    0
        PUSH    0
        CALL    Dos32Exit

segment class=data use32 flat
MSG:    db      "Hello, world!",13,10
LEN     equ     $ - MSG
WRITTEN:resd    1

segment class=stack stack use32 flat
        resd    1024

segment bss class=bss use32
        resd    1

group   dgroup  bss
```

hellowin.s   Програма „Здравей, свят!“ за i386+ на Windows (gas)

```
# as -o hellowin.obj hellowin.s; golink -console hellowin.obj kernel32.dll

.global Start
Start:                                  # Входна точка
        PUSHL   $-11                    # STD_OUTPUT_HANDLE (стандартен изход)
        CALL    GetStdHandle            # Върни № на файловия дескриптор
        PUSHL   $0                      # Запасен аргумент, трябва да бъде NULL
        PUSHL   $WRITTEN                # Брой действително записани байтове
        PUSHL   $LEN                    # Дължина на низа (UTF-8)
        PUSHL   $MSG                    # Адрес на низа
        PUSHL   %EAX                    # № на дескриптора на стандартния изход
        CALL    WriteConsoleA           # Функцията премахва от стека 20 байта
        CALL    ExitProcess             # Завършване на процеса
.data
MSG:    .ascii  "Здравей, свят!\n\n"    # CHCP 65001, за да се види този текст
        LEN = . - MSG
WRITTEN:.int    0
```

hellobsd.s   Програма „Здравей, свят!“ за i386+ на BSD/OS (gas)

```
.globl  _start,main
_start:                 # Входна точка
main:                   # Точка на прекъсване на gdb
        PUSH    $LEN     # Дължина на низа (UTF-8)
        PUSH    $MSG     # Адрес на низа
        PUSH    $1       # Файлов дескриптор 1: stdout (стандартен изход)
        SUB     $4,%ESP  # BSD изисква още 1 дума в стека
        MOV     $4,%EAX  # SYS_write (запис: /usr/include/sys/syscall.h)
        LCALL   $7,$0    # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX  # SYS_exit (завършване на процеса)
        LCALL   $7,$0
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellounx.s   Програма „Здравей, свят!“ за i386+ (as на UnixWare)

```
.globl  _start,main
_start:                 # Входна точка
main:                   # Точка на прекъсване на debug
        push    $LEN    # Дължина на низа (UTF-8)
        push    $MSG    # Адрес на низа
        push    $1      # Файлов дескриптор 1: stdout (стандартен изход)
        sub     $4,%esp # Unixware като BSD изисква още 1 дума в стека
        mov     $4,%eax # SYS_write (запис: /usr/include/sys/syscall.h)
        lcall   $7,$0   # Извикай съответната функция на ядрото на ОС
        mov     $1,%eax # SYS_exit (завършване на процеса)
        lcall   $7,$0
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellomac.s   Програма „Здравей, свят!“ за i386+ на Mac OS X (gas)

```
.globl  start,main
start:                  # Входна точка
main:                   # Точка на прекъсване на gdb
        PUSH    $LEN    # Дължина на низа (UTF-8)
        PUSH    $MSG    # Адрес на низа
        PUSH    $1      # Файлов дескриптор 1: stdout (стандартен изход)
        SUB     $4,%ESP # Mac OS X (и BSD въобще) изискват още 1 дума в стека
        MOV     $4,%EAX # SYS_write (запис: /usr/include/sys/syscall.h)
        INT     $0x80   # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX # SYS_exit (завършване на процеса)
        INT     $0x80
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellofbs.s   Програма „Здравей, свят!“ за i386+ на FreeBSD (gas)

```
.globl  _start,main
_start:                 # Входна точка
main:                   # Точка на прекъсване на gdb
        PUSH    %EBP     # „Зацепка“ за gdb, за да влезе в стъпков режим
        MOV     %ESP,%EBP
        PUSH    $LEN     # Дължина на низа (UTF-8)
        PUSH    $MSG     # Адрес на низа
        PUSH    $1       # Файлов дескриптор 1: stdout (стандартен изход)
        SUB     $4,%ESP  # BSD изисква още 1 дума в стека
        MOV     $4,%EAX  # SYS_write (запис: /usr/include/sys/syscall.h)
        INT     $0x80    # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX  # SYS_exit (завършване на процеса)
        INT     $0x80
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

helloopn.s Програма „Здравей, свят!“ за i386+ на OpenBSD (gas)

```
# as -o helloopn.o helloopn.s
# ld --dynamic-linker /usr/libexec/ld.so -o helloopn helloopn.o

.globl  _start,main
_start:                 # Входна точка
main:                   # Точка на прекъсване на gdb
        PUSH    %EBP    # „Зацепка“ за gdb, за да влезе в стъпков режим
        MOV     %ESP,%EBP
        PUSH    $LEN    # Дължина на низа (UTF-8)
        PUSH    $MSG    # Адрес на низа
        PUSH    $1      # Файлов дескриптор 1: stdout (стандартен изход)
        SUB     $4,%ESP # BSD изисква още 1 дума в стека
        MOV     $4,%EAX # SYS_write (запис: /usr/include/sys/syscall.h)
        INT     $0x80   # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX # SYS_exit (завършване на процеса)
        INT     $0x80
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG

.section ".note.netbsd.ident", "a", %note
        .p2align 2
        .int    8,4,1
        .asciz  "OpenBSD"
        .int    0
```

hello386.s   Програма „Здравей, свят!“ за i386+ на Linux (gas)

```
.global _start,main
_start:                 # Входна точка
main:                   # Точка на прекъсване на gdb
        MOV     $4,%EAX  # SYS_write (запис: /usr/include/{архит.}/asm/unistd.h
        MOV     $1,%EBX  # Файлов дескриптор 1: stdout (стандартен изход)
        MOV     $MSG,%ECX# Адрес на низа
        MOV     $LEN,%EDX# Дължина на низа (UTF-8)
        INT     $0x80    # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX  # SYS_exit (завършване на процеса)
        INT     $0x80
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellomnx.s   Програма „Здравей, свят!“ за i386+ на MINIX 3 (gas)

```
.globl  _start          # Входна точка
_start:
        MOV     $1,%EAX # Получател на съобщението
        MOV     $MSG_write,%EBX # Указател към структурата на съобщението
        MOV     $3,%ECX # SENDREC (прм–прд, вж. /usr/include/minix/ipcconst.h)
        INT     $0x21   # Предай съобщение на микроядрото чрез SYS386_VECTOR
        MOV     $MSG_exit,%EBX
        MOV     $3,%ECX
        INT     $0x21
.data
STR:    .ascii  "Здравей, свят!\n\n"
MSG_write:      # 4: WRITE (/usr/include/minix/callnr.h), 1: stdout (станд.изх.
.int    0,4,1, MSG_write – STR, 0,STR   # Адрес на записвания низ
.space  8       # Цялото съобщение е 32 байта; дотук са 24, значи остават още 8
MSG_exit:
.int    0,1     # 1: EXIT (завърши процеса, вж. /usr/include/minix/callnr.h)
.space  24      # Допълни до 32 байта (32 – 8 = 24)
```

helloind.s   Програма „Здравей, свят!“ за AMD64 на OpenIndiana (gas)

```
# as --64 -o helloind.o helloind.s && ld -m elf_x86_64 -o helloind helloind.o

.global _start,main
_start:                         # Входна точка
main:                           # Точка на прекъсване на gdb
        MOV     $4,%RAX         # SYS_write (запис: вж. /usr/include/sys/syscall.h)
        MOV     $1,%RDI         # Файлов дескриптор 1: stdout (стандартен изход)
        LEA     MSG,%RSI        # Адрес на низа
        MOV     $LEN,%RDX       # Дължина на низа (UTF-8)
        SYSCALL                 # Извикай съответната функция на ядрото на ОС
        MOV     $1,%EAX         # SYS_exit (завършване на процеса)
        SYSCALL
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellom64.s   Програма „Здравей, свят!“ за AMD64 на Mac OS X (gas)

```
# as -o hellom64.o hellom64.s
# ld -macosx_version_min 10.7 -o hellom64 hellom64.o
#
# 2 и 24 по-долу са SYSCALL_CLASS_UNIX и SYSCALL_CLASS_SHIFT, дефинирани в
# http://opensource.apple.com/source/xnu/xnu-1228/osfmk/mach/i386/syscall_sw.h

.globl  start,main
start:                              # Входна точка
main:                               # Точка на прекъсване на gdb
        MOV     $(2 << 24 | 4),%RAX # SYS_write (/usr/include/sys/syscall.h)
        MOV     $1,%RDI             # Файлов дескриптор 1: стандартен изход
        LEA     MSG(%RIP),%RSI      # Адрес на низа
        MOV     LEN(%RIP),%RDX      # Дължина на низа (UTF-8)
        SYSCALL                     # Извикай съответната функция на ядрото
        MOV     $(2 << 24 | 1),%RAX # SYS_exit (завършване на процеса)
        SYSCALL
.data
MSG:    .ascii  "Здравей, свят!\n\n"
LEN:    .long   . - MSG
```

helloarm.s   Програма „Здравей, свят!“ за ARM на Linux (gas)

```
.global _start,main
_start:                 // Входна точка
main:                   // Точка на прекъсване на gdb
        MOV     R7,#4   // SYS_write (запис: /usr/include/{архит.}/asm/unistd.h
        MOV     R0,#1   // Файлов дескриптор 1: stdout (стандартен изход)
        LDR     R1,=MSG // Адрес на низа
        MOV     R2,#LEN // Дължина на низа (UTF-8)
        SWI     0       // Извикай съответната функция на ядрото на ОС
        MOV     R7,#1   // SYS_exit (завършване на процеса)
        SWI     0
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellonet.s   Програма „Здравей, свят!“ за ARM на NetBSD (gas)

```
.global _start,main
_start:                 // Входна точка
main:                   // Точка на прекъсване на gdb
        MOV     R0,#1   // Файлов дескриптор 1: stdout (стандартен изход)
        LDR     R1,=MSG // Адрес на низа
        MOV     R2,#LEN // Дължина на низа (UTF-8)
        SWI     0xA00004// 4: SYS_write (запис, вж. /usr/include/sys/syscall.h)
        SWI     0xA00001// 1: SYS_exit (завършване на процеса)
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG

.section ".note.netbsd.ident", "a", %note
        .int    7,4,1
        .ascii  "NetBSD"
        .p2align 2
        .int    102000000//Версия 1.2 е първата, пренесена на ARM
```

helloa64.s   Програма „Здравей, свят!“ за ARM64 на FreeBSD (clang)

```
# FreeBSD: clang -c -o helloa64.o helloa64.s; ld -o helloa64 helloa64.o

.global _start,main
_start:                 // Входна точка
main:                   // Точка на прекъсване на gdb
        MOV     X8,#4   // SYS_write (запис: /usr/include/sys/syscall.h)
        MOV     X0,#1   // Файлов дескриптор 1: stdout (стандартен изход)
        LDR     X1,=MSG // Адрес на низа
        MOV     X2,#27  // Дължина на низа (UTF-8)
        SVC     0       // Извикай съответната функция на ядрото на ОС
        MOV     X8,#1   // SYS_exit (завършване на процеса)
        SVC     0
.data
MSG:    .ascii "Здравей, свят!\n\n"
```

helloaa6.s   Програма „Здравей, свят!“ за ARM64 на Linux (gas)

```
.global _start,main
_start:                 // Входна точка
main:                   // Точка на прекъсване на gdb
        MOV     X8,#64  // SYS_write (запис: /usr/include/asm-generic/unistd.h)
        MOV     X0,#1   // Файлов дескриптор 1: stdout (стандартен изход)
        LDR     X1,=MSG // Адрес на низа
        MOV     X2,#LEN // Дължина на низа (UTF-8)
        SVC     0       // Извикай съответната функция на ядрото на ОС
        MOV     X8,#93  // SYS_exit (завършване на процеса)
        SVC     0
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

helloppc.s   Програма „Здравей, свят!“ за PowerPC на Mac OS X (gas)

```
.globl  start,_main
start:                          ; Входна точка
_main:                          ; Точка на прекъсване на gdb
        li      r0,4            ; SYS_write (запис: /usr/include/sys/syscall.h)
        li      r3,1            ; Файлов дескриптор 1: stdout (стандартен изход
        lis     r4,hi16(MSG)    ; Зареди старшата част на адреса на низа, << 16
        addi    r4,r4,lo16(MSG) ; Добави младшата му част
        li      r5,27           ; Дължина на низа (UTF-8)
        sc                      ; Извикай съответната функция на ядрото на ОС
        nop                     ; Ще бъде прескочена при успешен SC (SysCall)
        li      r0,1            ; SYS_exit (завършване на процеса)
        sc
.data
MSG:    .ascii "Здравей, свят!\n\n"
```

```
# as -a64 -o helloaix.o helloaix.s && ld -b64 -o helloaix helloaix.o
# Дългият пролог е необходим, за да работи командата "start" на gdb.
# ВНИМАНИЕ: Номерата на системните извиквания важат само за AIX версия
# 7100-00-03-1115 (oslevel -s), и то само за 64-битови програми!

.csect  main[DS]
.globl  __start
__start:                # Входна точка
.llong  .main
.csect  .text[PR]
.globl  .main
.main:                  # Точка на прекъсване на gdb
        la      4,T.MSG(2) # Адрес на низа
        li      2,312   # write (запис – вж. забележката за номерата по-горе!)
        li      3,1     # Файлов дескриптор 1: stdout (стандартен изход)
        li      5,LEN   # Дължина на низа (UTF-8)
        bl      l1      # Върни адреса на mflr в lr
l1:     mflr    6       # Има още 4 команди до командата след svca;
        addi    6,6,4*4 # затова коригирай адреса в lr с 4 × 4,
        mtlr    6       # та да указва към нея.
        svca    0       # Извикай съответната функция на ядрото на ОС
        li      2,52    # exit (завършване на процеса – вж. забележката за №№)
        svca    0
.csect  .data[RW]
MSG:    .byte   "Здравей, свят!"
        .byte   10,10
.set    LEN,$ - MSG
.align  3
.toc
T.MSG:  .tc     MSG[TC],MSG
```

helloirx.s   Програма „Здравей, свят!“ за MIPS32 (as на Irix)

```
 # as -nocpp -non_shared helloirx.s; ld -non_shared -o helloirx helloirx.o

        li      $4,1    # Файлов дескриптор 1: stdout (стандартен изход)
        la      $5,MSG  # Адрес на низа
        li      $6,27   # Дължина на низа (UTF-8)
        li      $2,1004 # SYS_write (запис: /usr/include/sys.s)
        syscall         # Извикай съответната функция на ядрото на ОС
        li      $2,1001 # SYS_exit (завършване на процеса)
        syscall
.data
MSG:    .ascii  "Здравей, свят!\n\n"
```

helloi64.s   Програма „Здравей, свят!“ за MIPS64 (as на Irix)

```
 # as -64 -nocpp -non_shared helloirx.s; ld -non_shared -o helloirx helloirx.o

        li      $4,1    # Файлов дескриптор 1: stdout (стандартен изход)
        dla     $5,MSG  # Адрес на низа
        li      $6,27   # Дължина на низа (UTF-8)
        li      $2,1004 # SYS_write (запис: /usr/include/sys.s)
        syscall         # Извикай съответната функция на ядрото на ОС
        li      $2,1001 # SYS_exit (завършване на процеса)
        syscall
.data
MSG:    .ascii  "Здравей, свят!\n\n"
```

hellosun.s   Програма „Здравей, свят!“ за SPARC на Solaris (gas)

```
.globl  _start,main
_start:                 ! Входна точка
main:                   ! Точка на прекъсване на gdb
        MOV     1,%o0   ! Файлов дескриптор 1: stdout (стандартен изход)
        SET     MSG,%o1 ! Адрес на низа
        MOV     LEN,%o2 ! Дължина на низа (UTF-8)
        MOV     4,%g1   ! SYS_write (запис: /usr/include/sys/syscall.h)
        TA      8       ! Извикай съответната функция на ядрото на ОС
        MOV     1,%g1   ! SYS_exit (завършване на процеса)
        TA      8
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellos64.s   Програма „Здравей, свят!“ за SPARC64 на Solaris (gas)

```
! as -Av9 -64 -o hellos64.o hellos64.s
! ld -Av9 -m elf64_sparc -o hellos64 hellos64.o

.globl  _start,main
_start:                 ! Входна точка
main:                   ! Точка на прекъсване на gdb
        MOV     1,%o0   ! Файлов дескриптор 1: stdout (стандартен изход)
        SETX    MSG,%o2,%o1 ! Адрес на низа
        MOV     LEN,%o2 ! Дължина на низа (UTF-8)
        MOV     4,%g1   ! SYS_write (запис: /usr/include/sys/syscall.h)
        TA      0x40    ! Извикай съответната функция на ядрото на ОС
        MOV     1,%g1   ! SYS_exit (завършване на процеса)
        TA      0x40
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN = . - MSG
```

hellopar.s   Програма „Здравей, свят!“ за PA-RISC (as на HP-UX)

```
.code
.export $START$
$START$                         ; Входна точка
        LDIL    0x180000,%r18   ; 0x180000 << 11 = 0xC0000000, база на спод.об.
        LDI     4,%r22          ; SYS_write (/usr/include/sys/scall_define.h)
        LDI     1,%r26          ; Файлов дескриптор 1: stdout (стандартен изход
        LDIL    L%MSG,%r25      ; Зареди старшата част на адреса на низа, << 11
        LDO     R%MSG(%r25),%r25; Добави младшите му 11 бита като отместване
        BE,L    4(%sr7,%r18)    ; Извикай функцията на ядрото на ОС (отложено)
        LDI     27,%r24         ; Дължина на низа (изпълнява се преди BE,L!)
        BE      4(%sr7,%r18)
        LDI     1,%r22          ; SYS_exit (завършване; изпълнява се преди BE)
.data
MSG     .string "Здравей, свят!\n\n"

        .subspa $UNWIND_END$,access=0x1F
        .export $UNWIND_END
$UNWIND_END
```

hellop64.s Програма „Здравей, свят!“ за PA-RISC64 (as на HP-UX)

```
; as -o hellop64.o hellop64.s; ld -noshared -o hellop64 hellop64.o

.level  2.0w
.code
.export $START$
$START$                         ; Входна точка
        ADDI    MSG-$START$-3,%r31,%r25; Адрес на низа
        LDI     1,%r26          ; Файлов дескриптор 1: stdout (стандартен изход
        LDD     T%MSG(%r30),%r1 ; Предотврати "cannot execute binary file"
        LDI     4,%r22          ; SYS_write (/usr/include/sys/scall_define.h)
        LDIL    L%0x60000800,%r1
        ADD     %r1,%r1,%r18    ; 2 * 0x60000800 = 0xC0001000
        BE,L    0(%sr4,%r18)    ; Извикай функцията на ядрото на ОС (отложено)
        LDI     27,%r24         ; Дължина на низа (изпълнява се преди BE,L!)
        BE      0(%sr4,%r18)
        LDI     1,%r22          ; SYS_exit (завършване; изпълнява се преди BE)

MSG     .string "Здравей, свят!\n\n"
```

hellovax.s   Програма „Здравей, свят!“ за VAX на Ultrix (gas)

```
.globl  _start
_start:
        .word   0       # Входна маска (gas няма директива .entry)
        PUSHL   $LEN    # Дължина на низа (UTF-8)
        PUSHAL  MSG     # Адрес на низа
        PUSHL   $1      # Файлов дескриптор 1: stdout (стандартен изход)
        PUSHL   $3      # Брой аргументи
        MOVL    SP,AP   # Направи SP указател към аргументите
        CHMK    $4      # SYS_write (запис: /usr/sys/h/syscall.h)
        PUSHL   $0
        MOVL    SP,AP
        CHMK    $1      # SYS_exit (завършване на процеса)
.data
MSG:    .ascii "Здравей, свят!\n\n"
        LEN     = . - MSG
```

hellotru.s   Програма „Здравей, свят!“ за Alpha (α; as на Tru64)

```
.globl  main
.ent    main
main:                   # Входна точка и точка на прекъсване на gdb
        ldah    $29,0($27)!gpdisp!1 # pv ($27) не е валиден => и gp ($29) не е,
        lda     $29,0($29)!gpdisp!1 # но без този пролог b main (gdb) не работи
        br      $27,l1  # Върни програмния брояч pc в pv (procedure value)
l1:     ldgp    $29,0($27)  # as разширява този макрос като ldah/lda по-горе
        ldah    $17,MSG($29)!gprelhigh!2 # Ст.16 б. на 32-б. знаково отм. от gp
        lda     $17,MSG($17)!gprellow!2  # Младши 16 бита на горното отместване
        ldil    $16,1   # Файлов дескриптор 1: stdout (стандартен изход)
        ldil    $18,27  # Дължина на низа (UTF-8)
        ldil    $0,4    # SYS_write (запис: /usr/include/sys/syscall.h)
        call_pal 0x83   # Извикай съответната функция на ядрото на ОС
        ldil    $0,1    # SYS_exit (завършване на процеса)
        call_pal 0x83
.end    main
.data
MSG:    .ascii  "Здравей, свят!\n\n"
```

hellovms.mar   Програма „Здравей, свят!“ за IA–64/α на OpenVMS (MACRO)

```
.psect  data    wrt,noexe
MSG:    .ascid  "Hello, Itanium!"<13><10>

.psect  code    nowrt,exe
.entry  start,0
        PUSHAQ  MSG
        CALLS   #1,G^LIB$PUT_OUTPUT
        RET
.end    start
```

КРАЙ