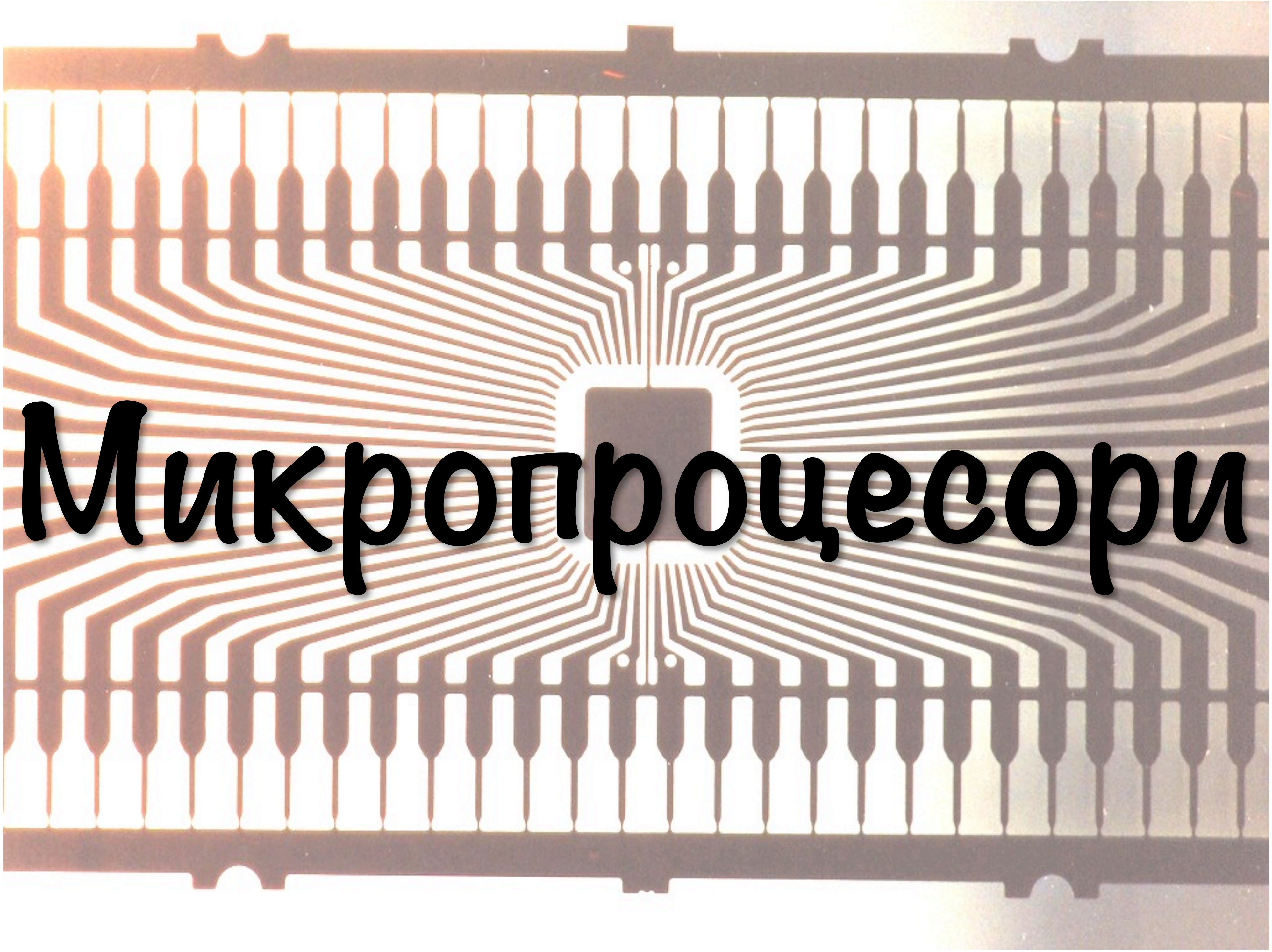
Микропроцесори



Тема 0.	Въведение: Развитие на микроелектронните технологии за производство на СГИС. Кратка история на 32-битовите микропроцесори (МП) x86 и „ARM“.	1 час
Тема 1.	Програмен модел на МП: Понятие за програмен модел. Режими. Регистри за обща употреба. Специализирани регистри. Флагове на регистъра за кода на условието (РКУ). Особености. Обзор на програмния модел на други МП.	1 час
Тема 2.	Система от машинни команди: Групи команди. Формат на командите. „Операнд 2“. Методи за адресация. Ортогоналност на системата команди.	2 часа
Тема 3.	Структура на МП: Основни функционални блокове в МП. Вътрешни шини. Работа на конвейера.	2 часа
Тема 4.	Системна магистрала: Сигнали на шините за адреси и данни. Управляващи сигнали. Организация на обмена на данни. Видове цикли. Времедиаграми.	2 часа
Тема 5.	Устройство за плаваща запетая: Конвейери за умножение и натрупване, делене и коренуване и зареждане и съхранение. Режими. Обработка на къси вектори. Регистров файл. Програмен модел. Команди. Изключения.	2 часа
Тема 6.	Изключения и прекъсвания: Изключения. Прекъсвания – видове и връзка с режимите на МП. Таблица на векторите на изключенията и прекъсванията. Начално установяване на МП.	1 час
Тема 7.	Устройство за управление на паметта: Функции. Регистри. Транслация на адресите. Дескриптори. Кеширане и буфериране. Грешки. Буфер за запис.	1 час
Тема 8.	Развитие на микропроцесорната архитектура: Развитие на МП до 64-битова архитектура. Графични процесори. Многоядреност.	2 часа
Тема 9.	Кратки сведения за други МП: Условни преходи и пренос в МП без РКУ („Alpha“, MIPS). МП с „регистров прозорец“ (SPARC). Програми „Здравей, свят!“ за различни МП и операционни системи (ОС).	1 час

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“.

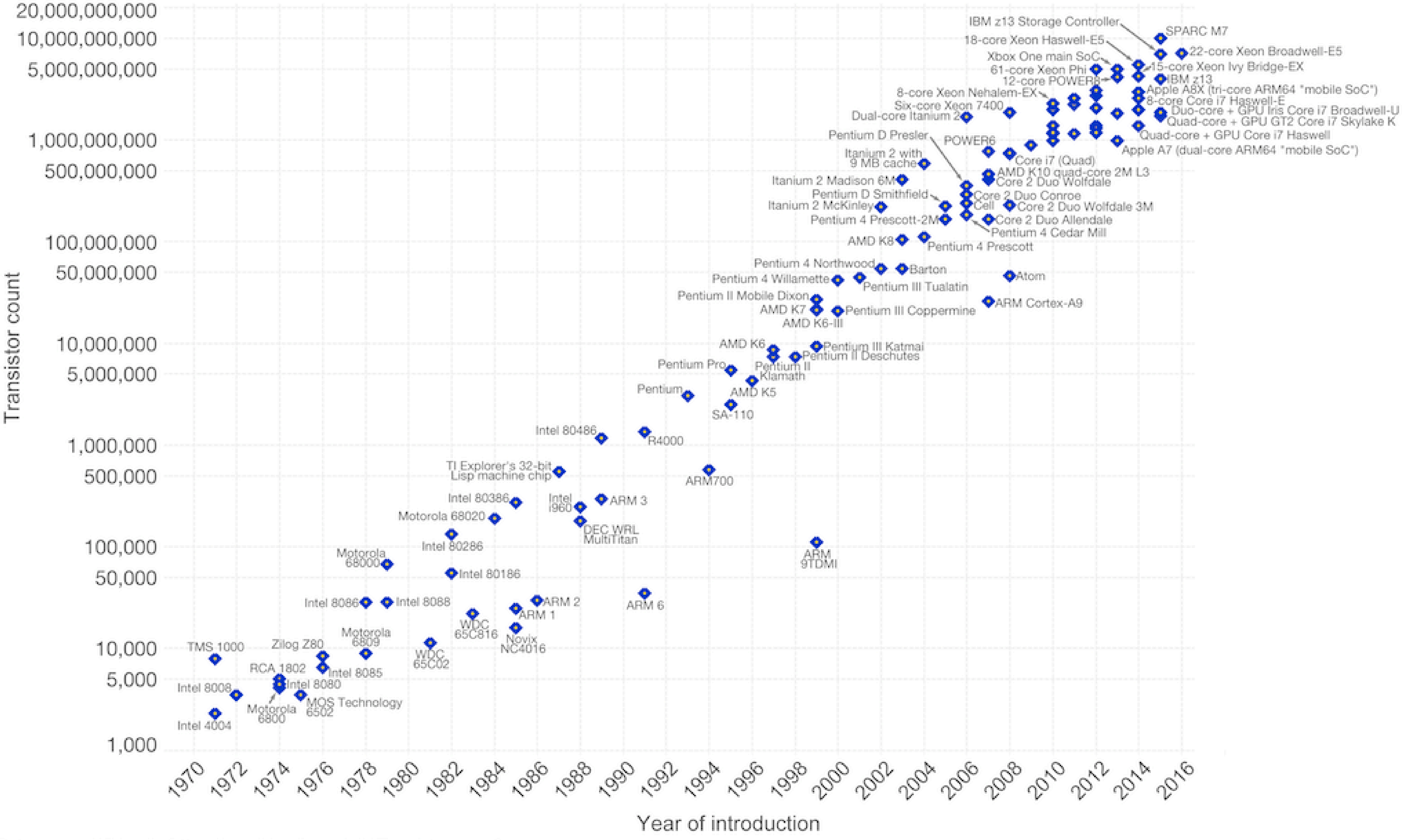
Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

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.

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

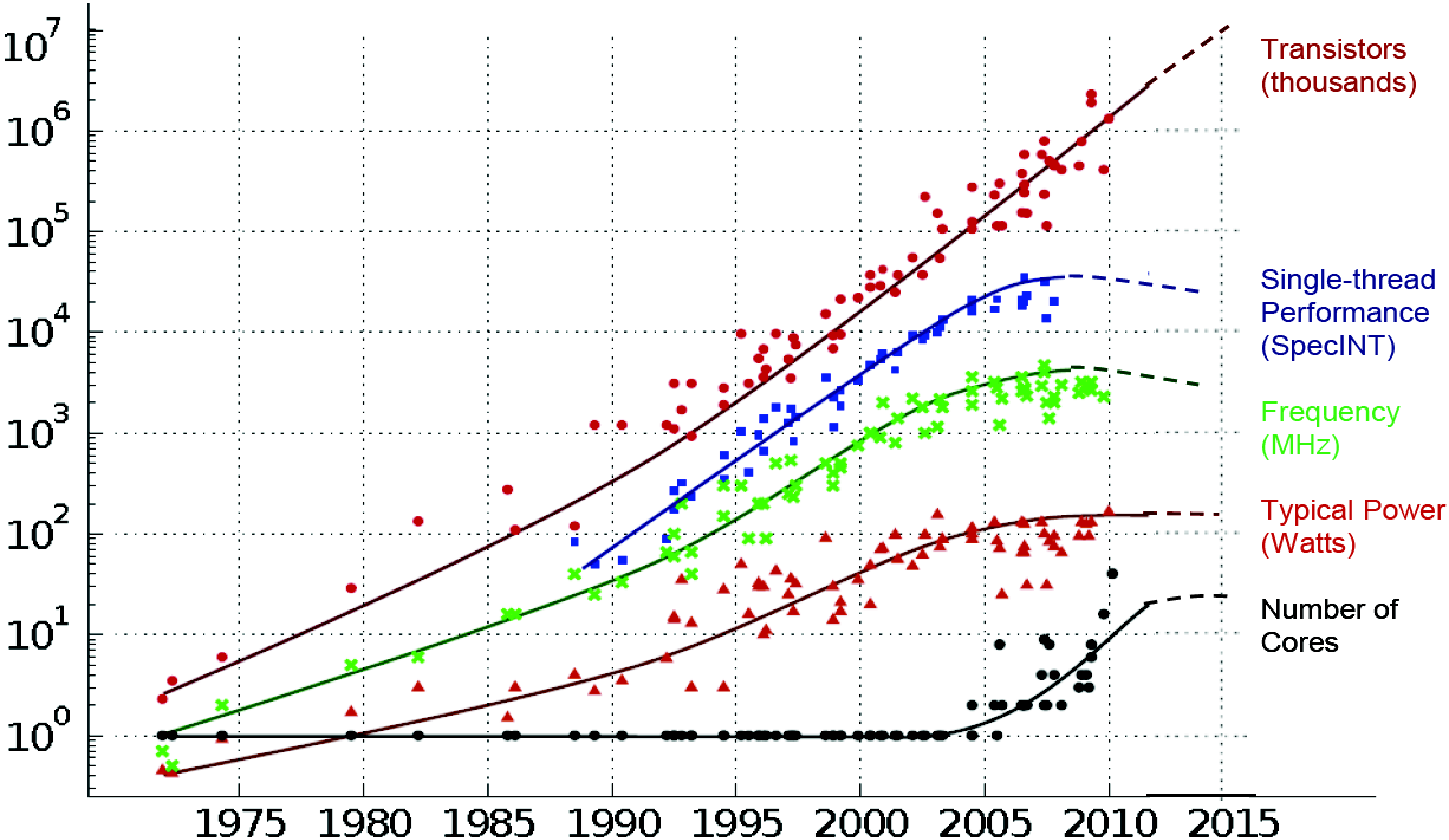
Our World in Data

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 strongly linked to Moore's law.



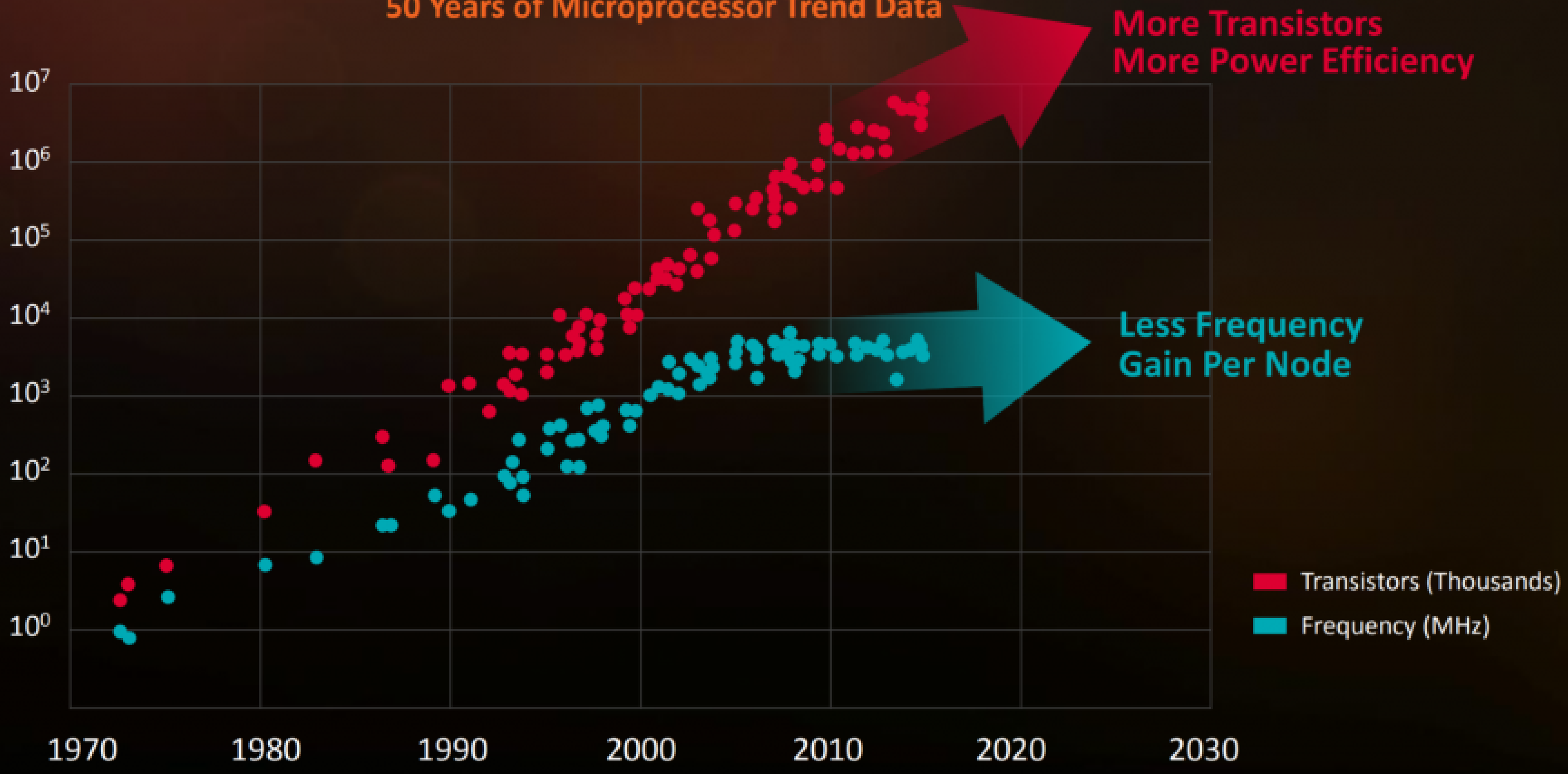
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

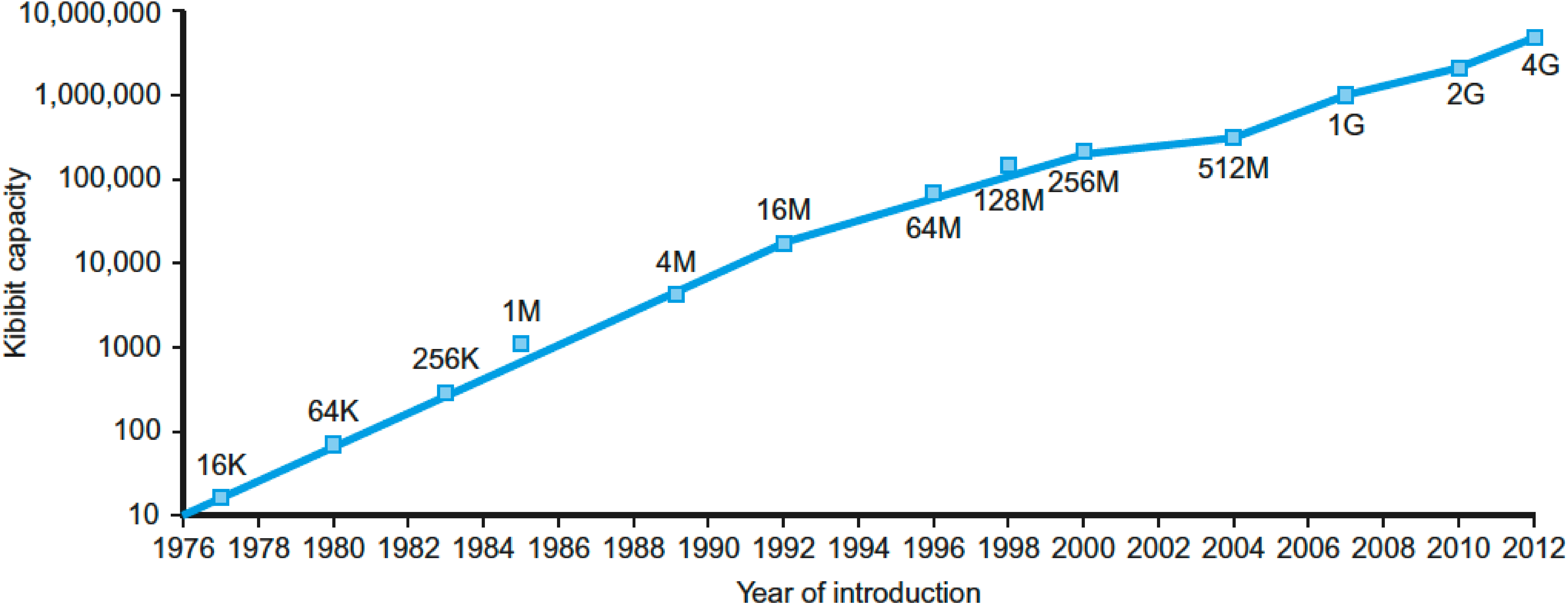
MOORE'S LAW IS SLOWING



50 Years of Microprocessor Trend Data

More Transistors
More Power Efficiency

Less Frequency
Gain Per Node


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

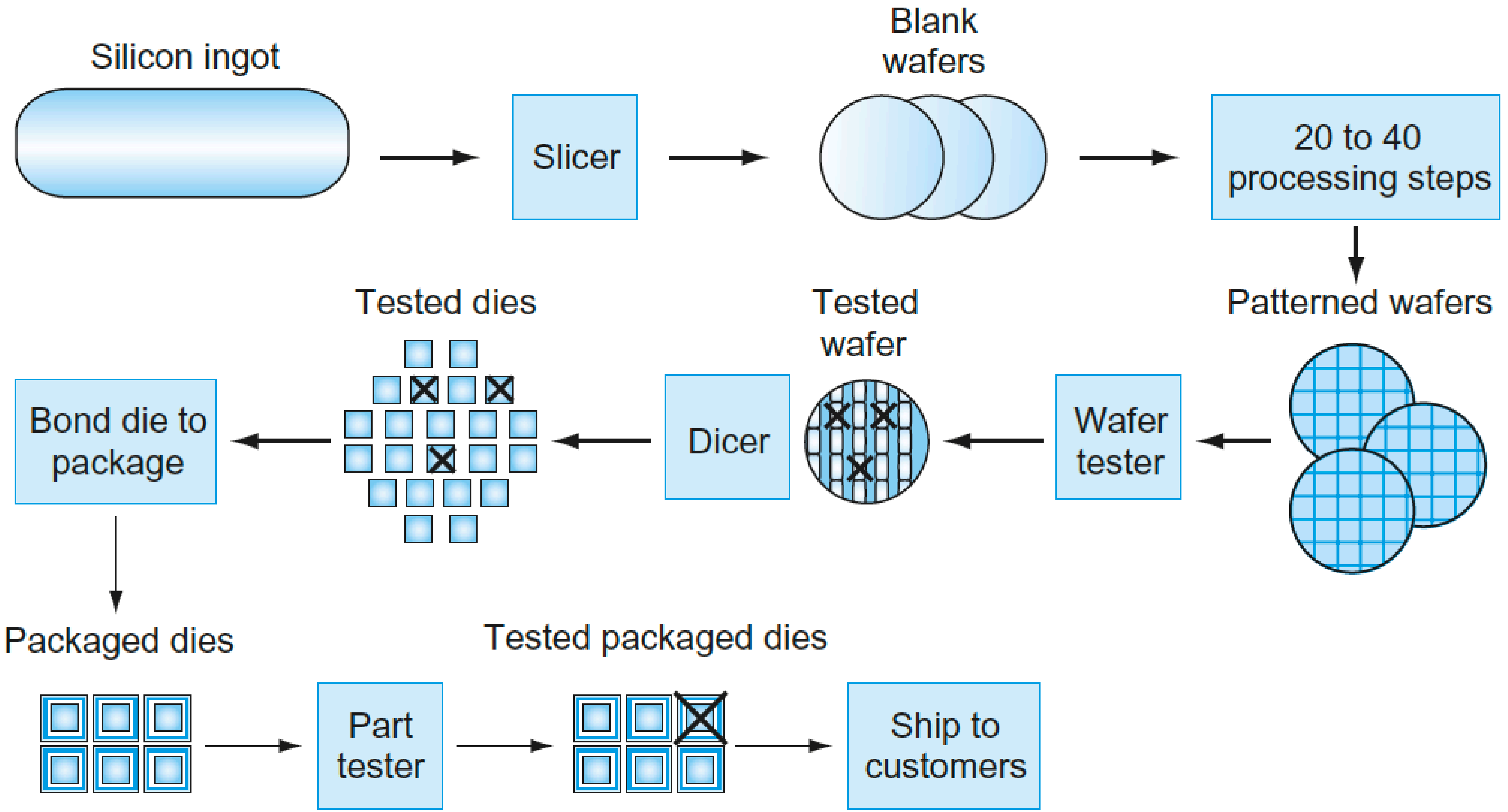
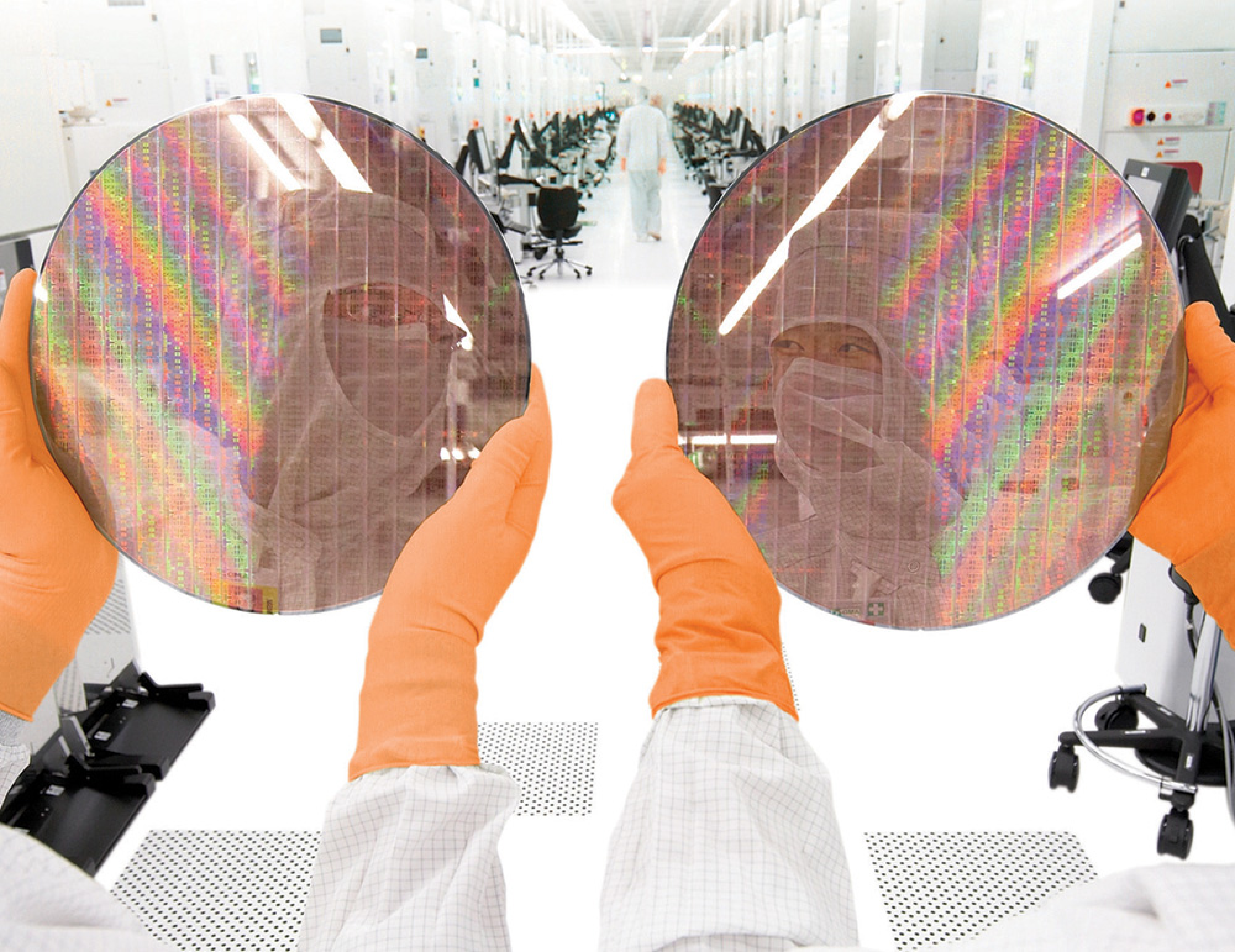
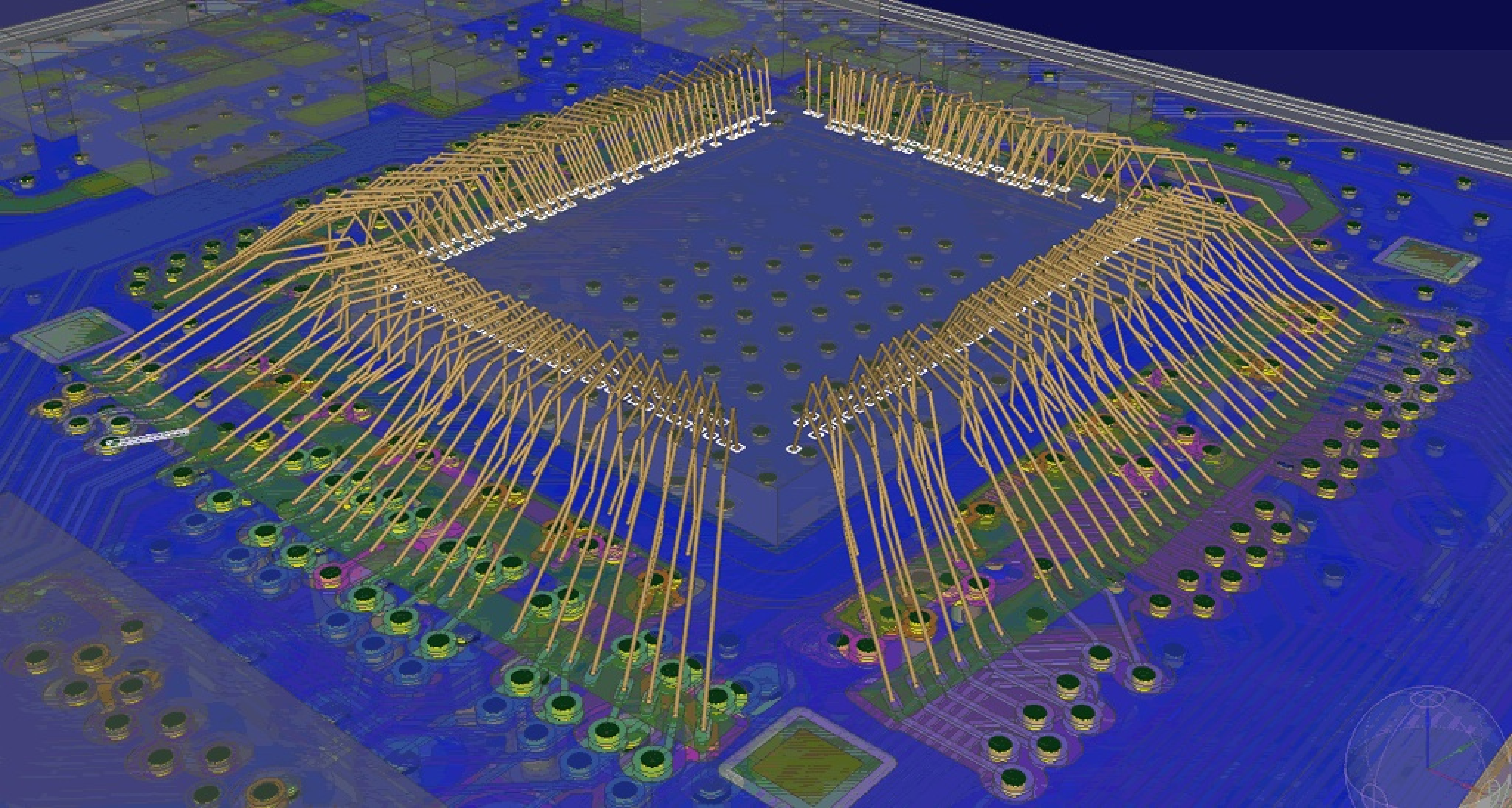
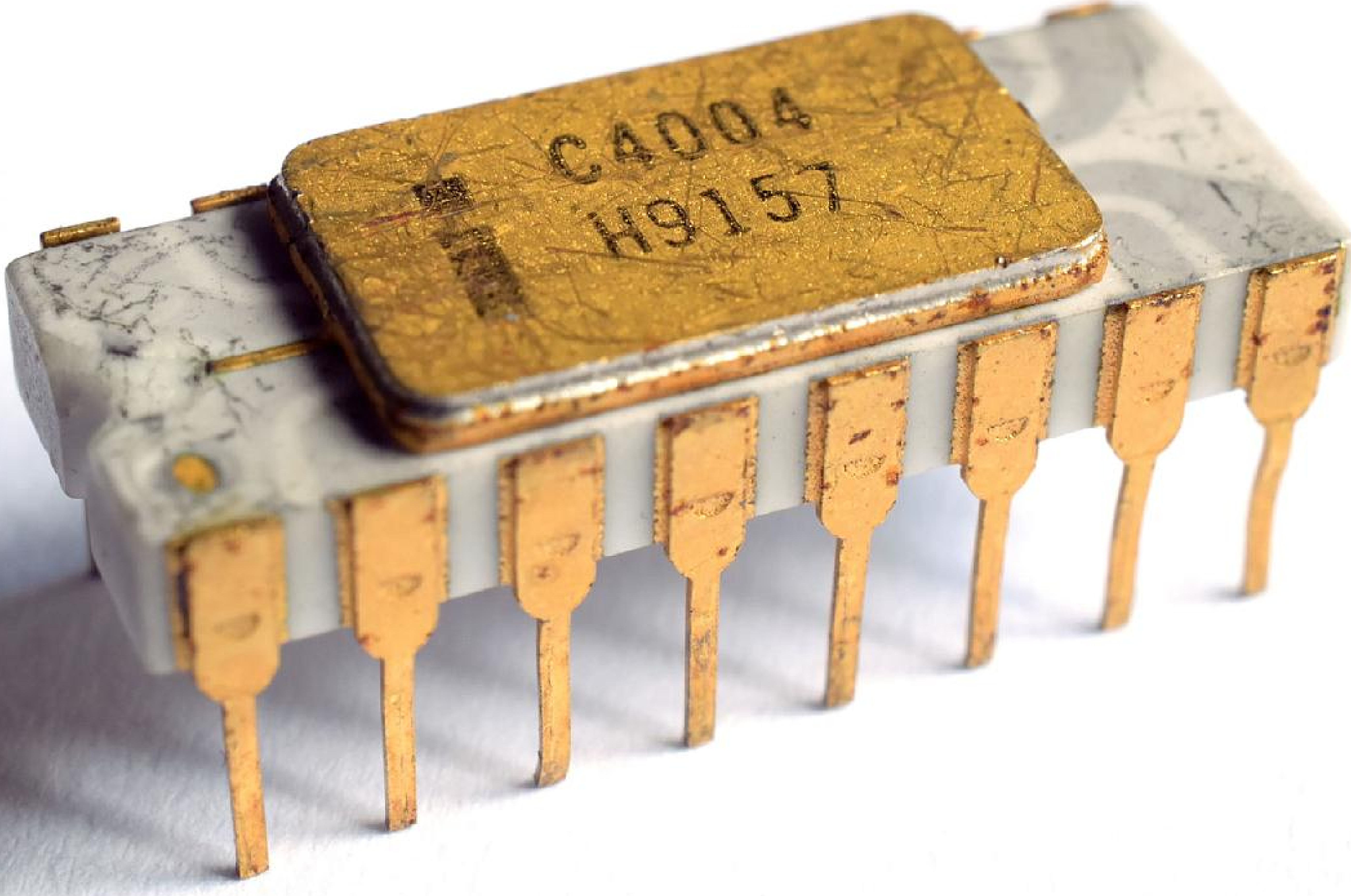
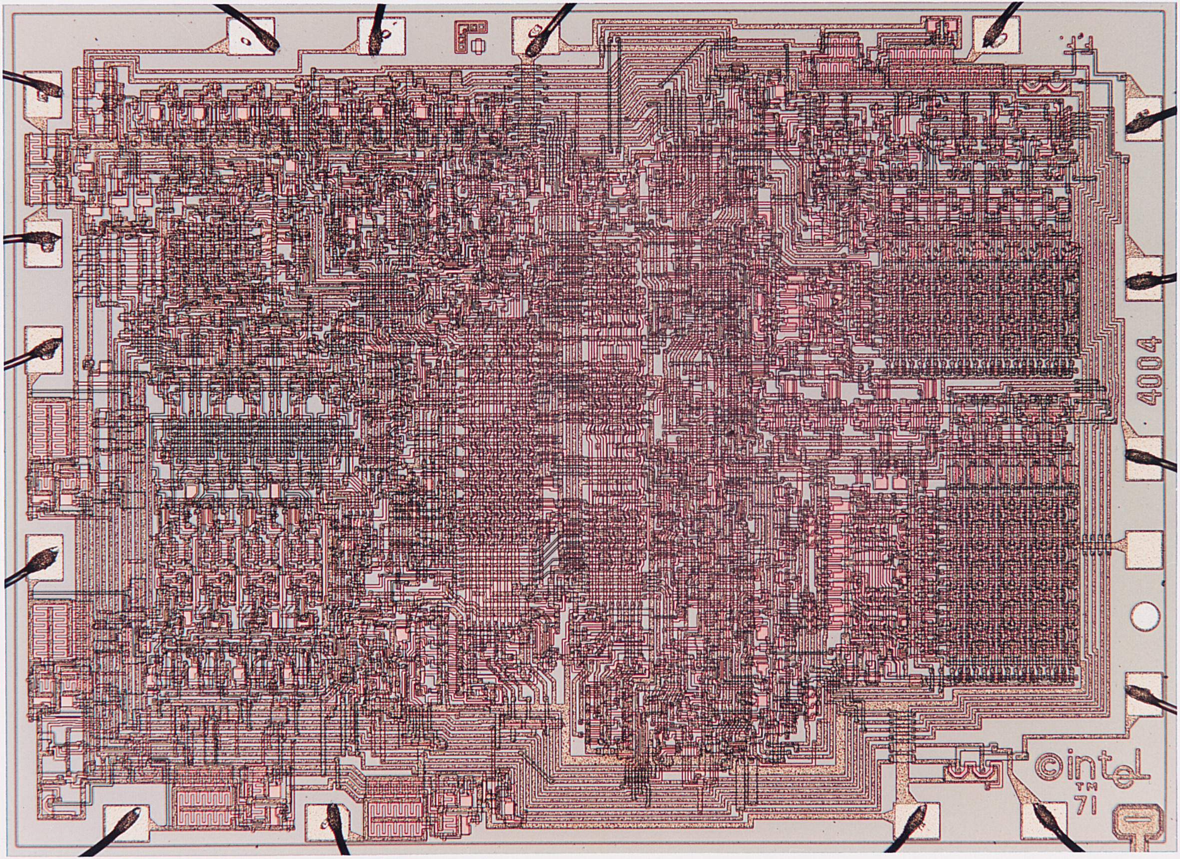


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Next, the wafers are diced 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.

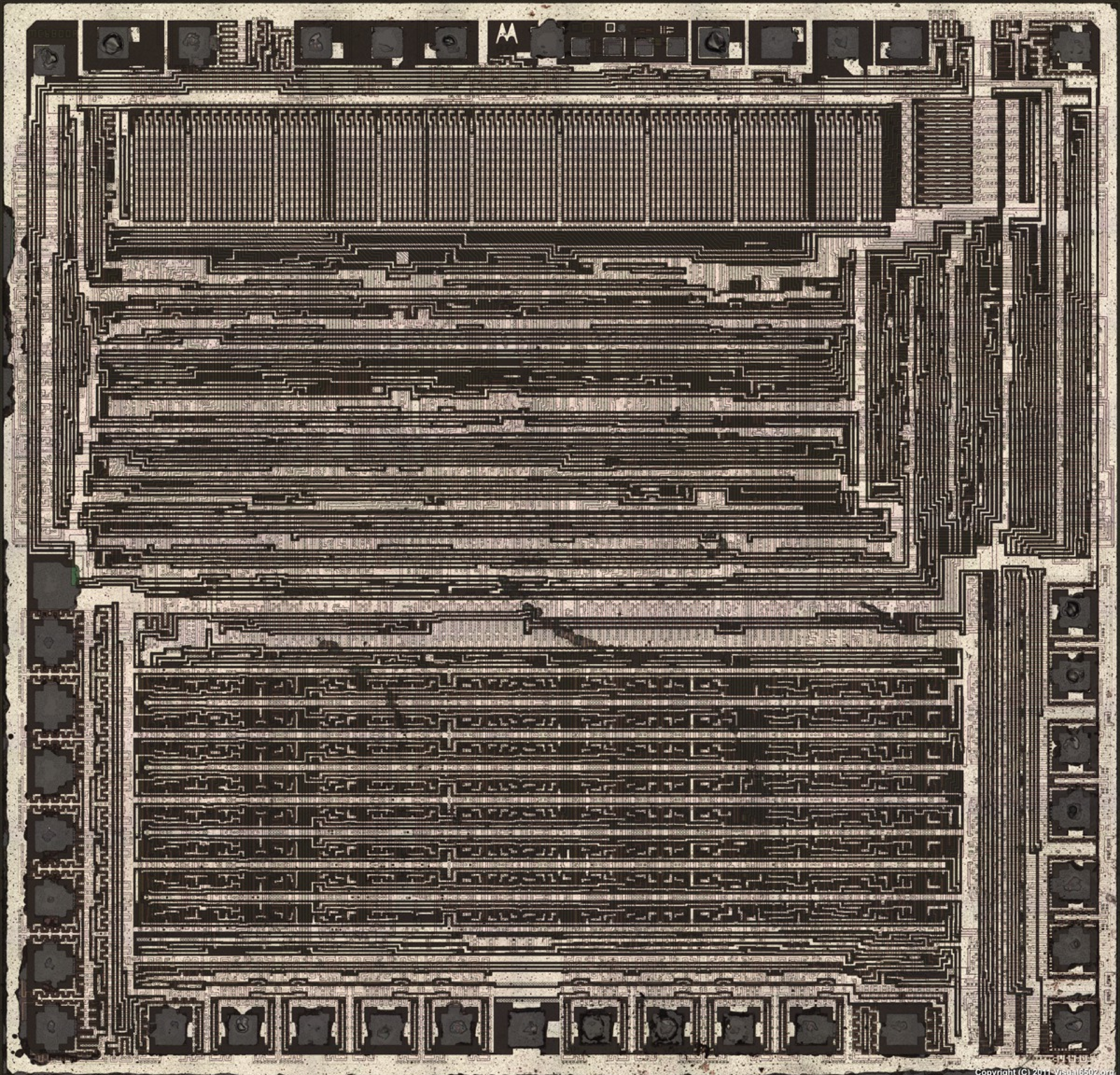


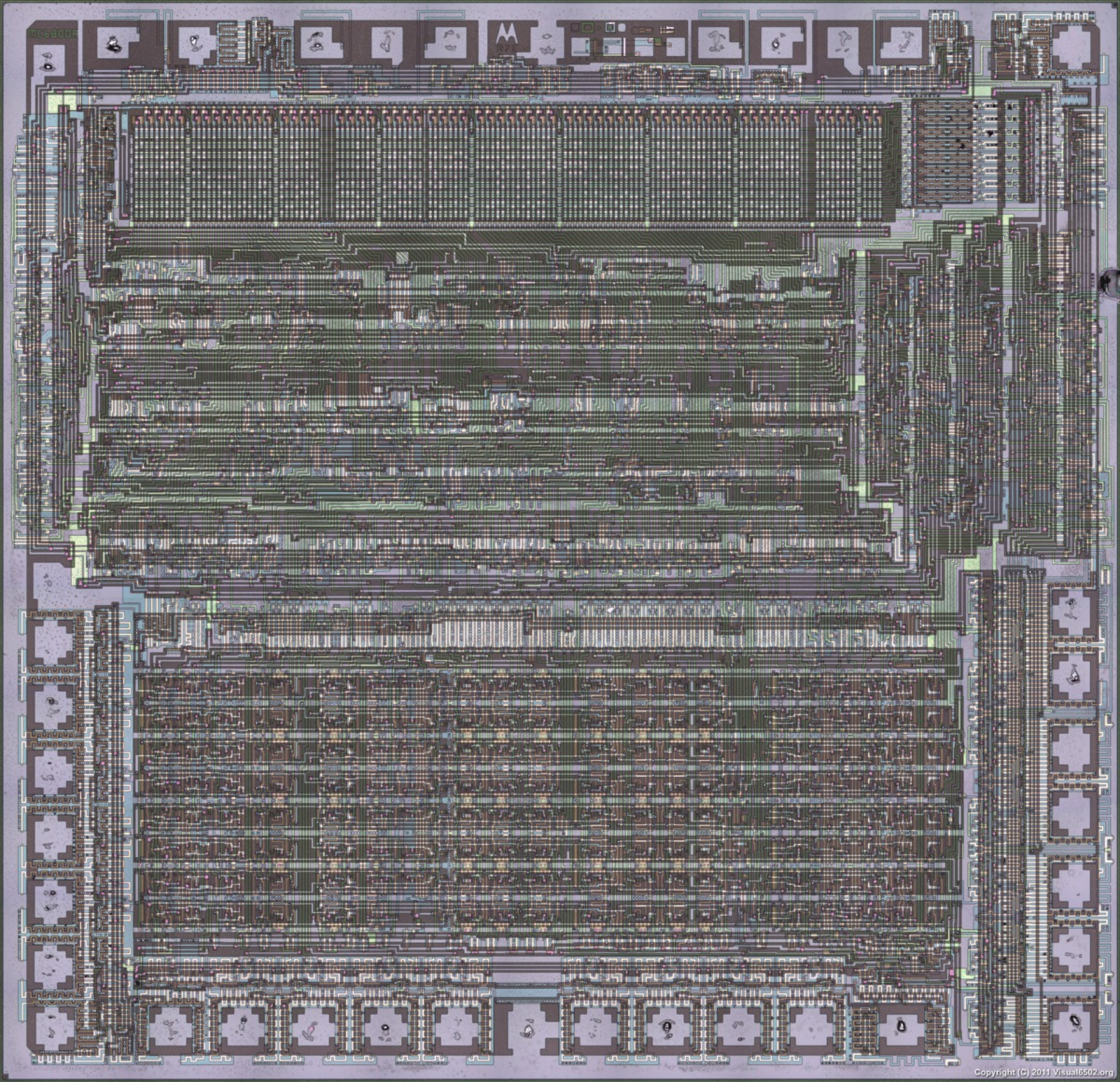



C4004
H9157

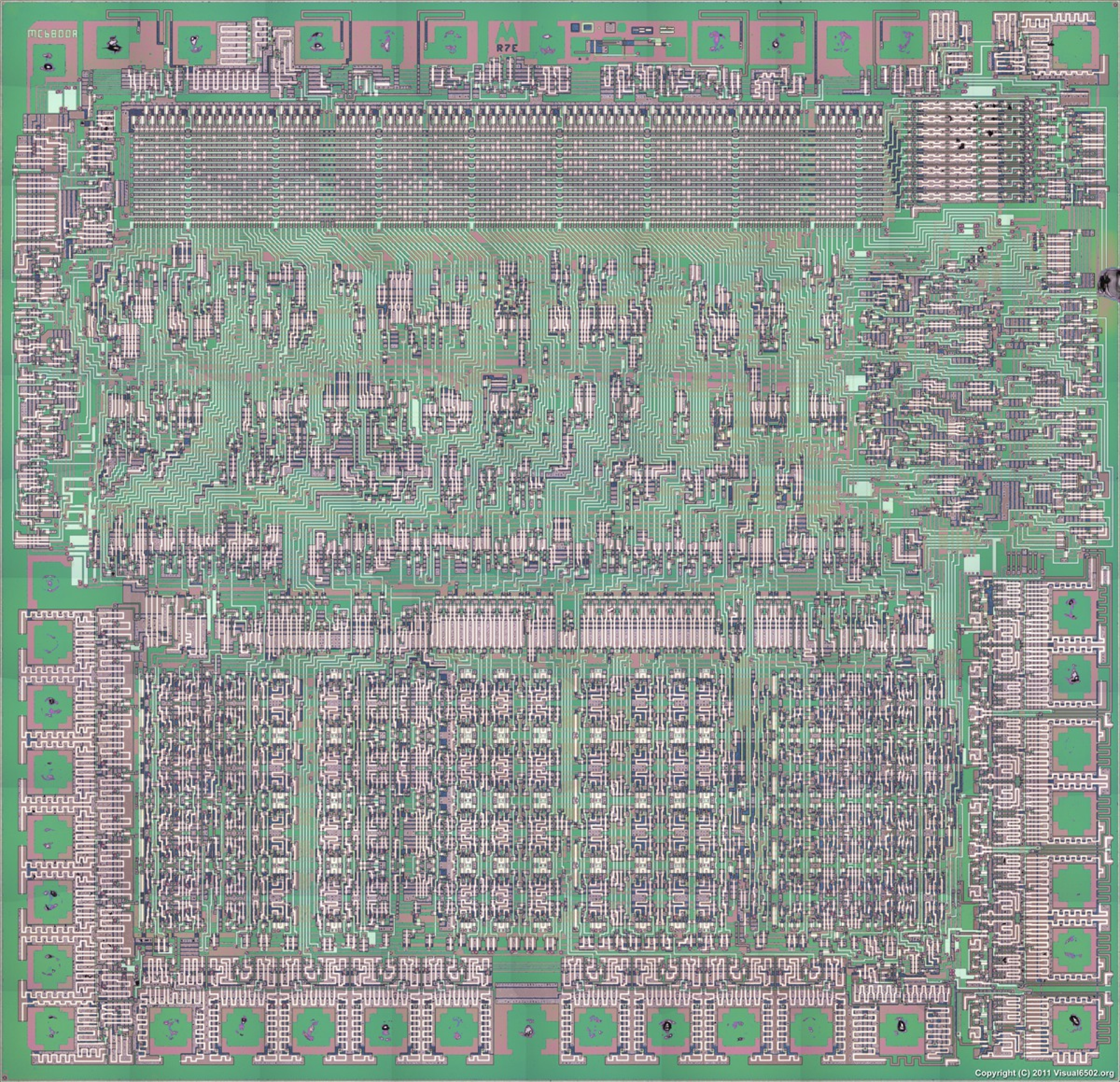

4004
©intel
TM
71

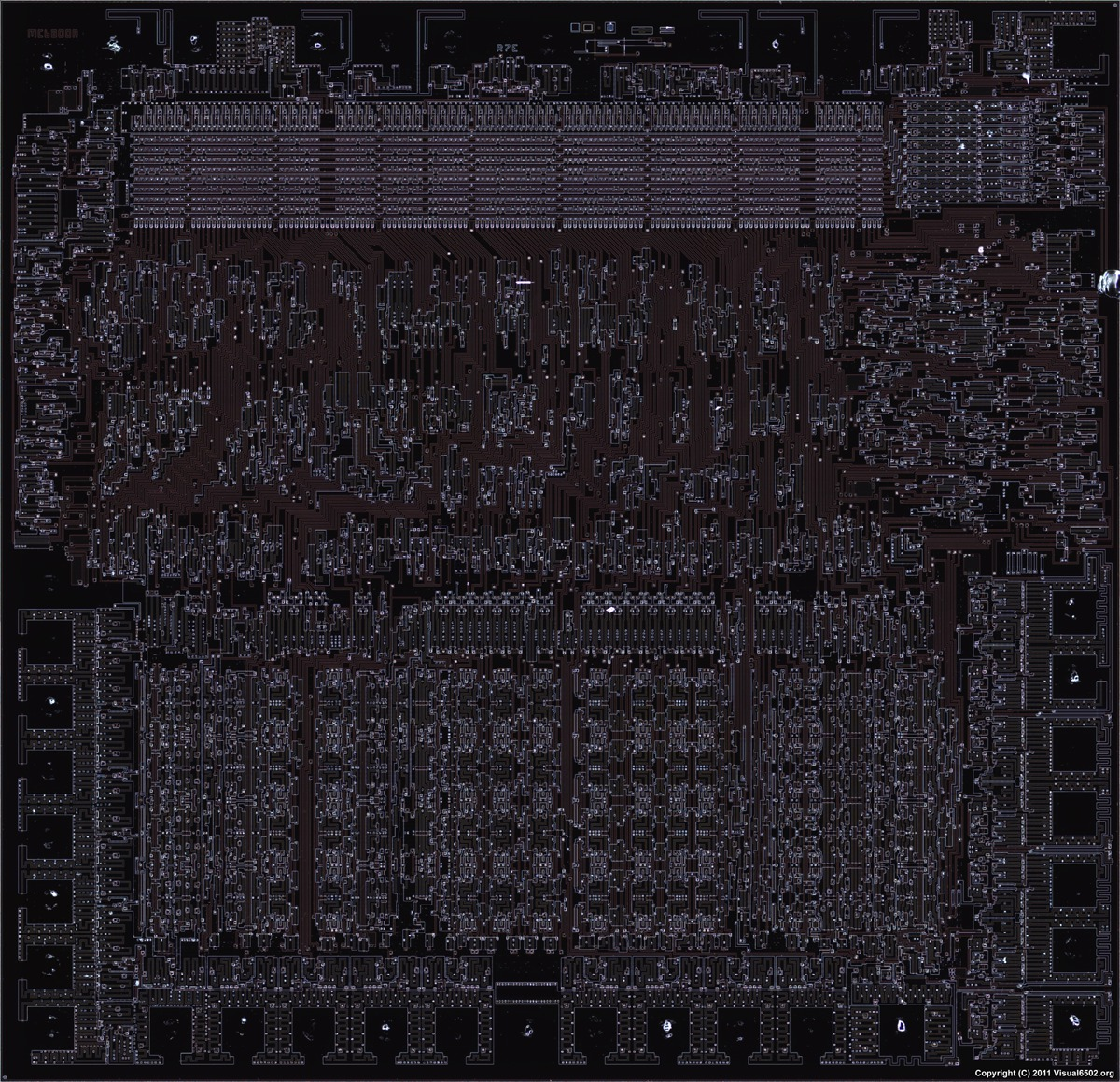

CM 601
IV-80
1
CM 601
81 26
1
CM601
8608
CM601
8352



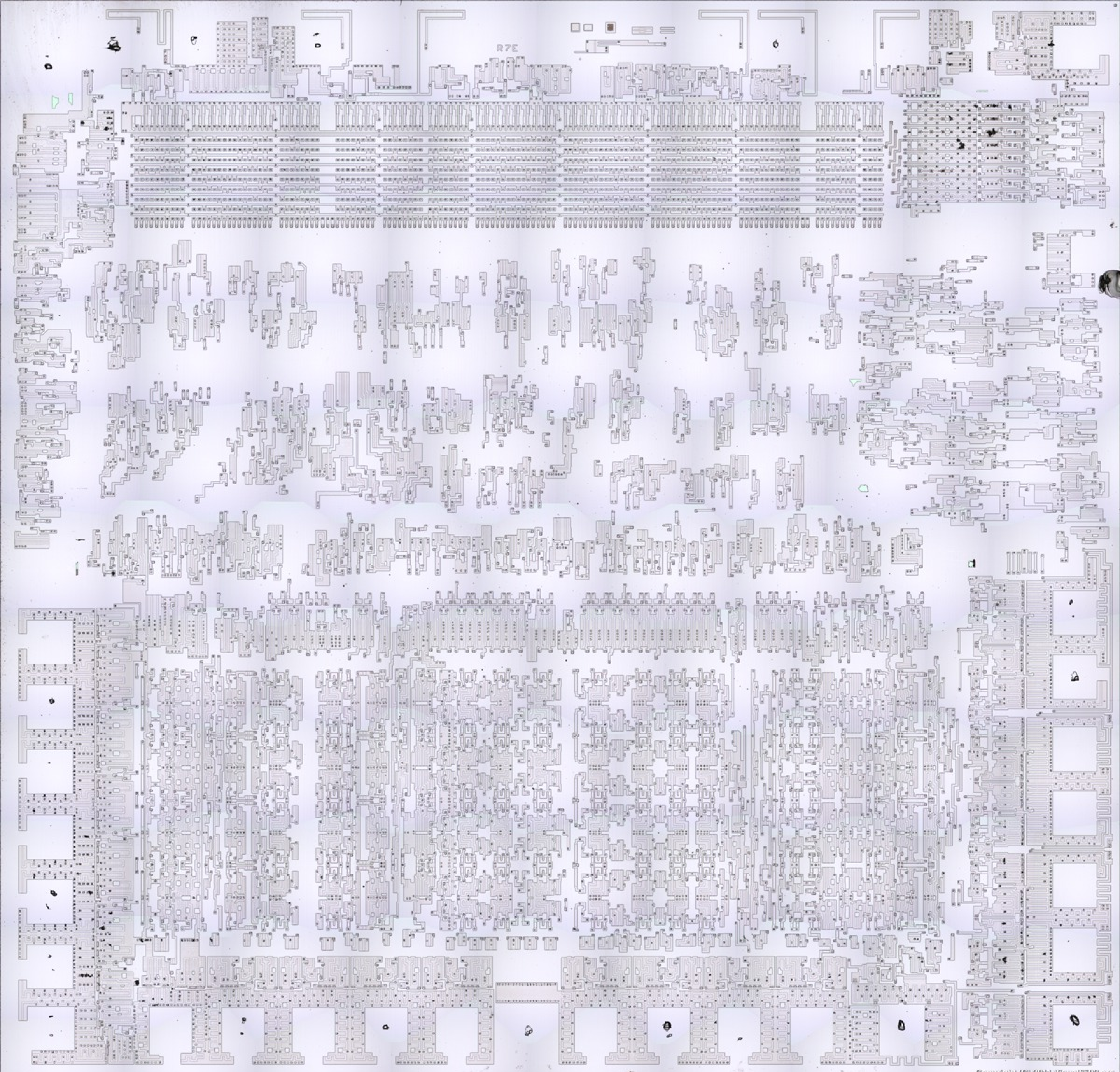


MC6800A

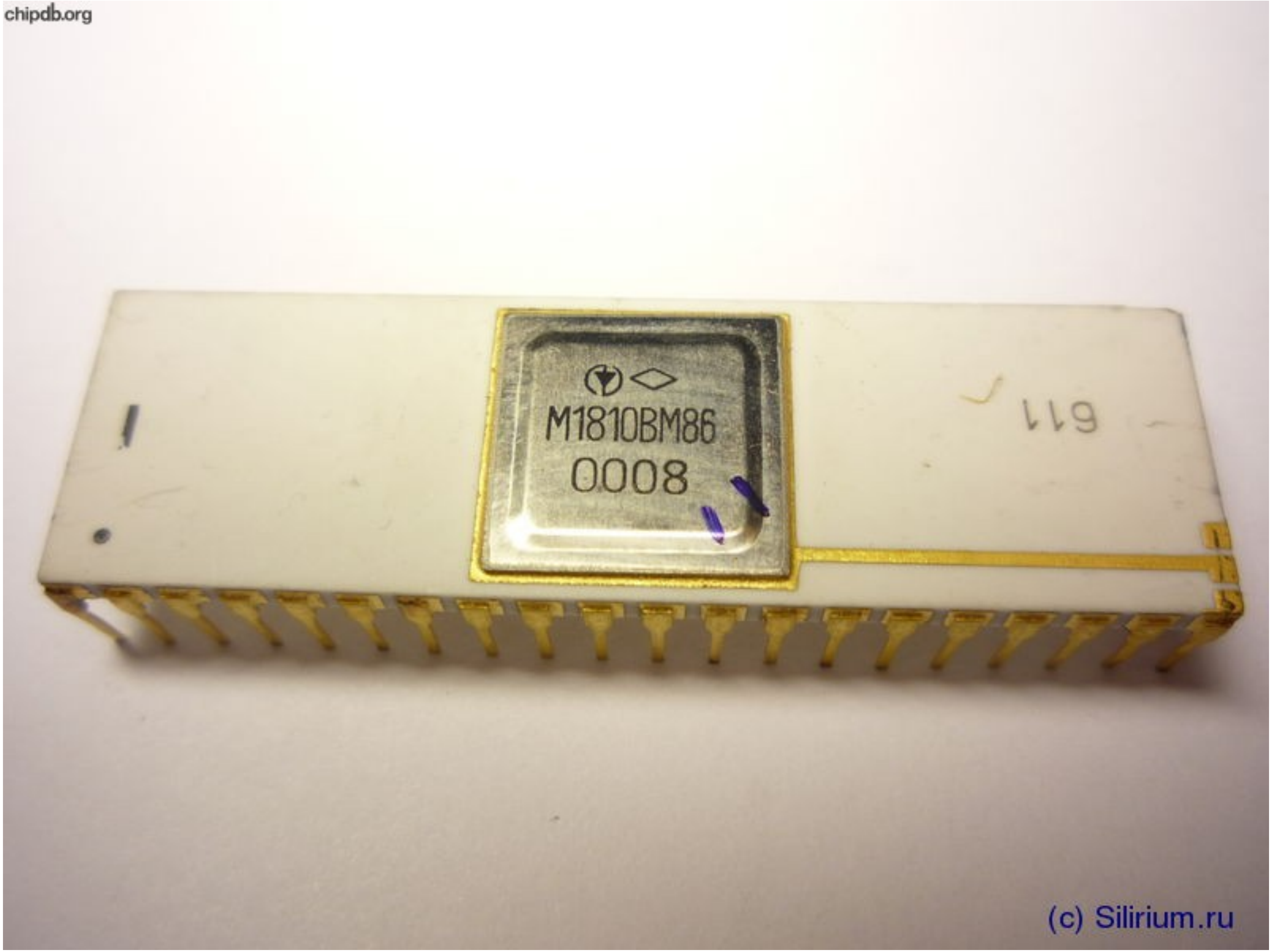

MC6800A
R7E
Copyright (C) 2011 Visual6502.org



Copyright (C) 2011 Visual6502.org


R7E


CM601P
K9318


М1810ВМ86
0008

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

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}		$\|\pm I0\|$
5. Выходной ток в состоянии "Выключено", мкА	I_{OZ}		$\|\pm I0\|$
6. Входная емкость, пФ	C_I		I5
7. Емкость входа/выхода, пФ	$C_{I/O}$		

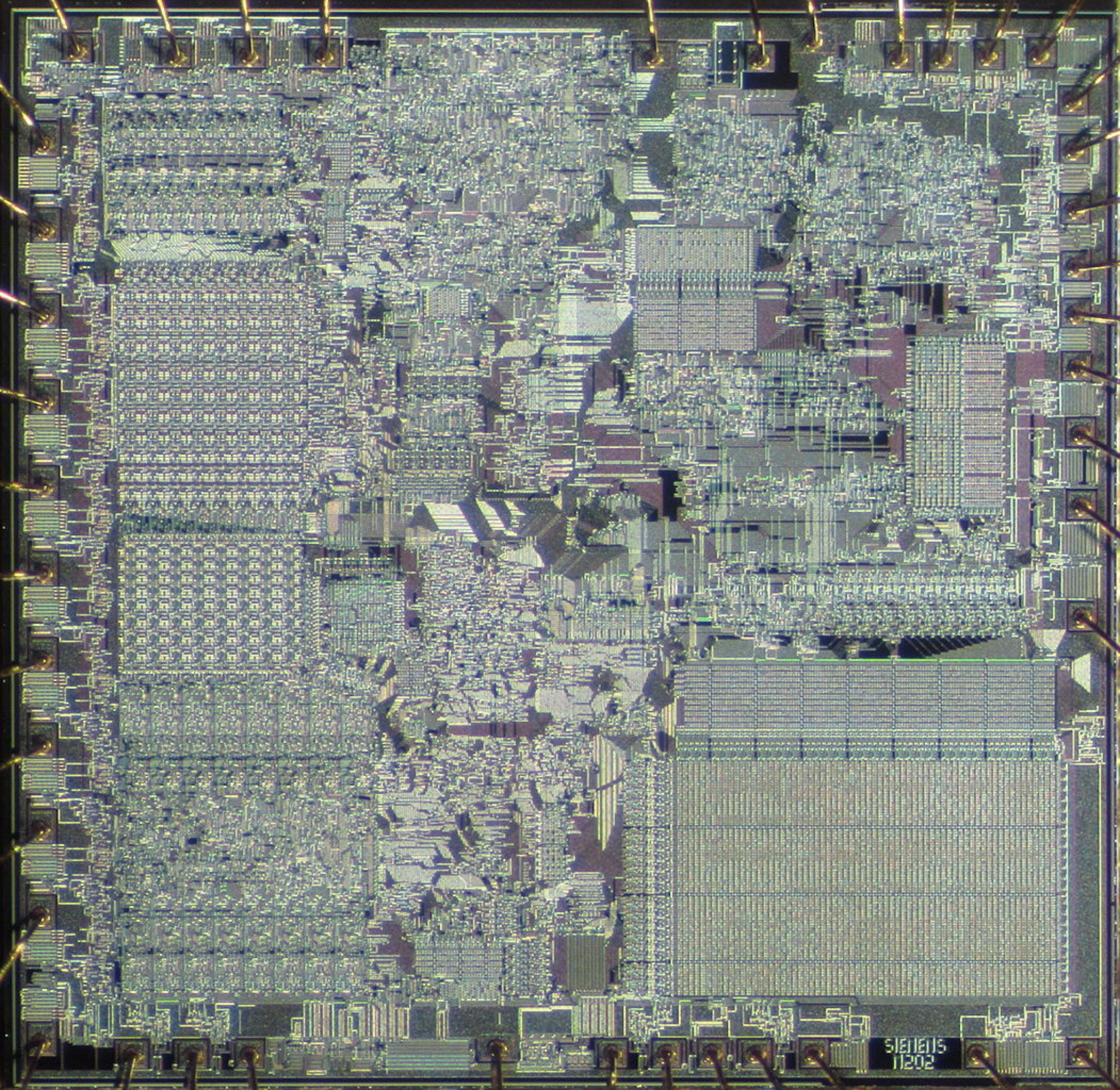

SIEMENS
11202

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.

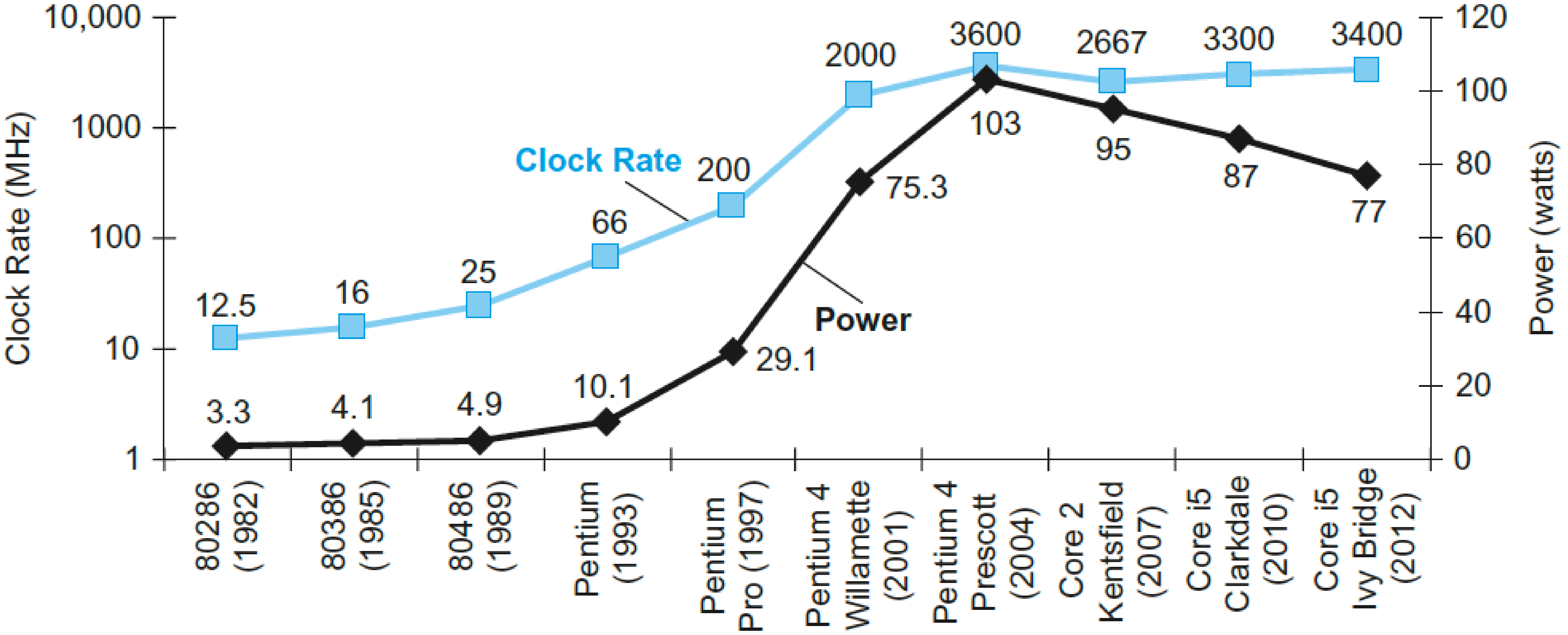


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 30 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline 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!

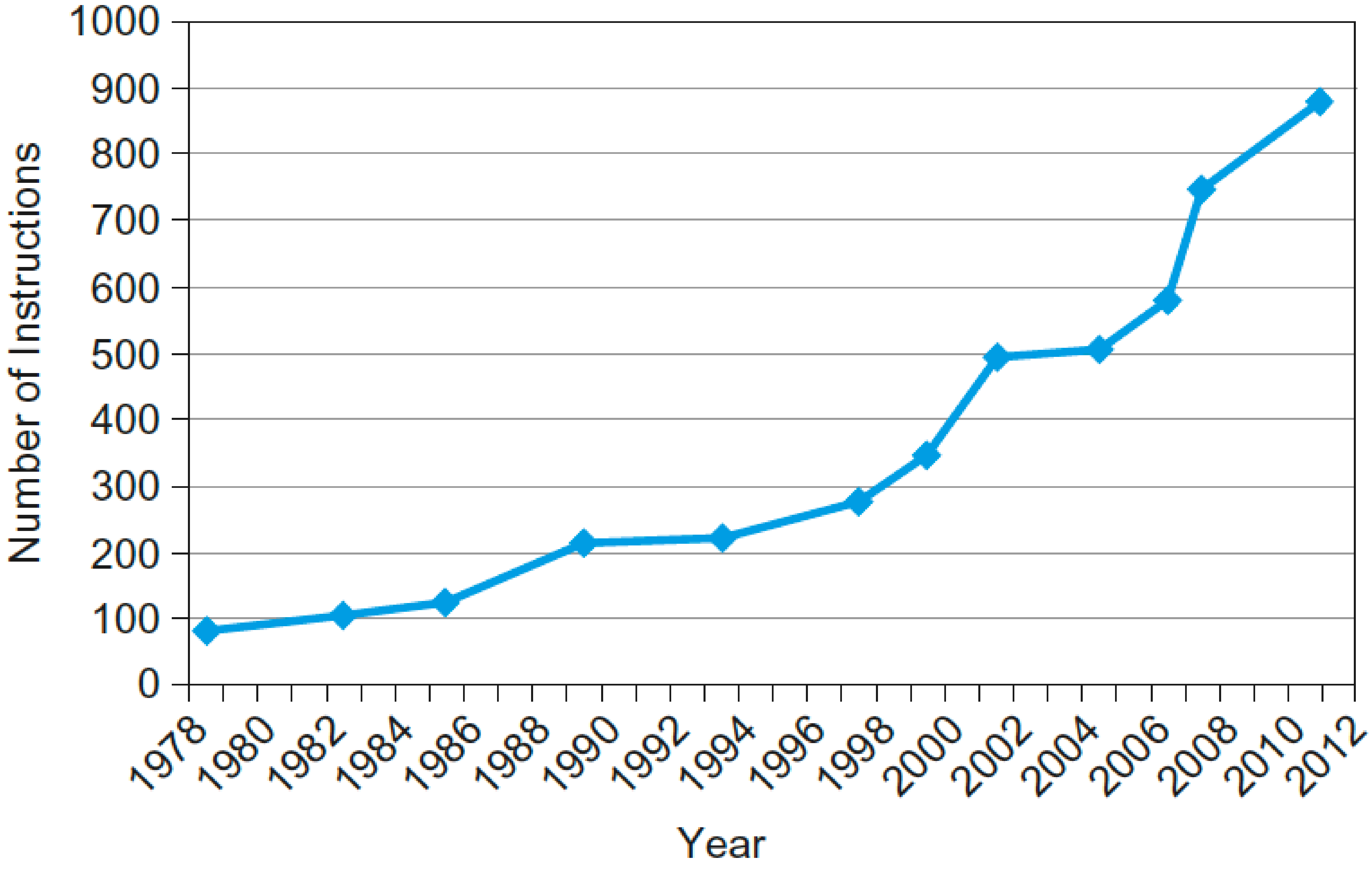


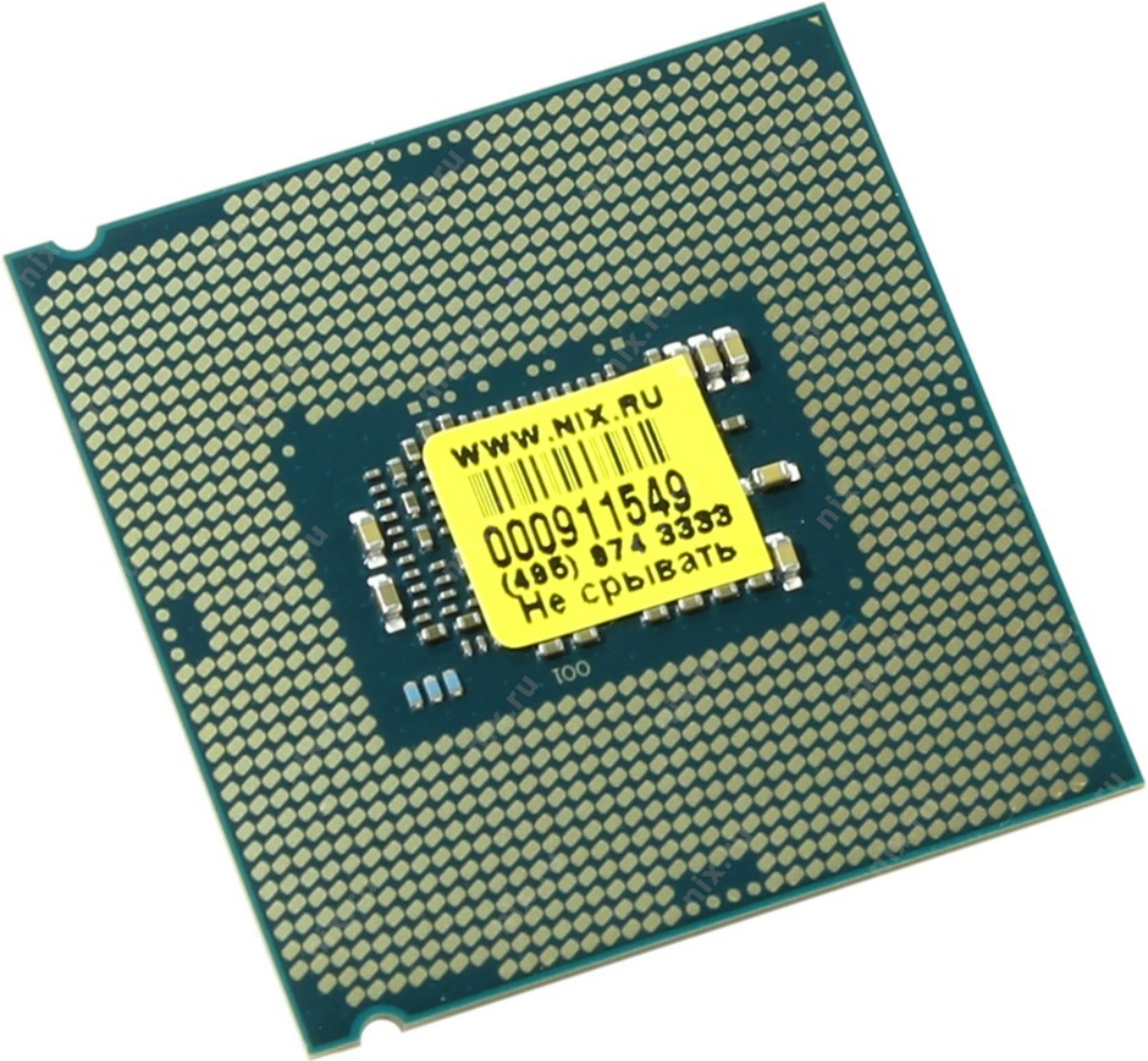
FIGURE 2.44 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

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 (недокументирана) = 96 бр.

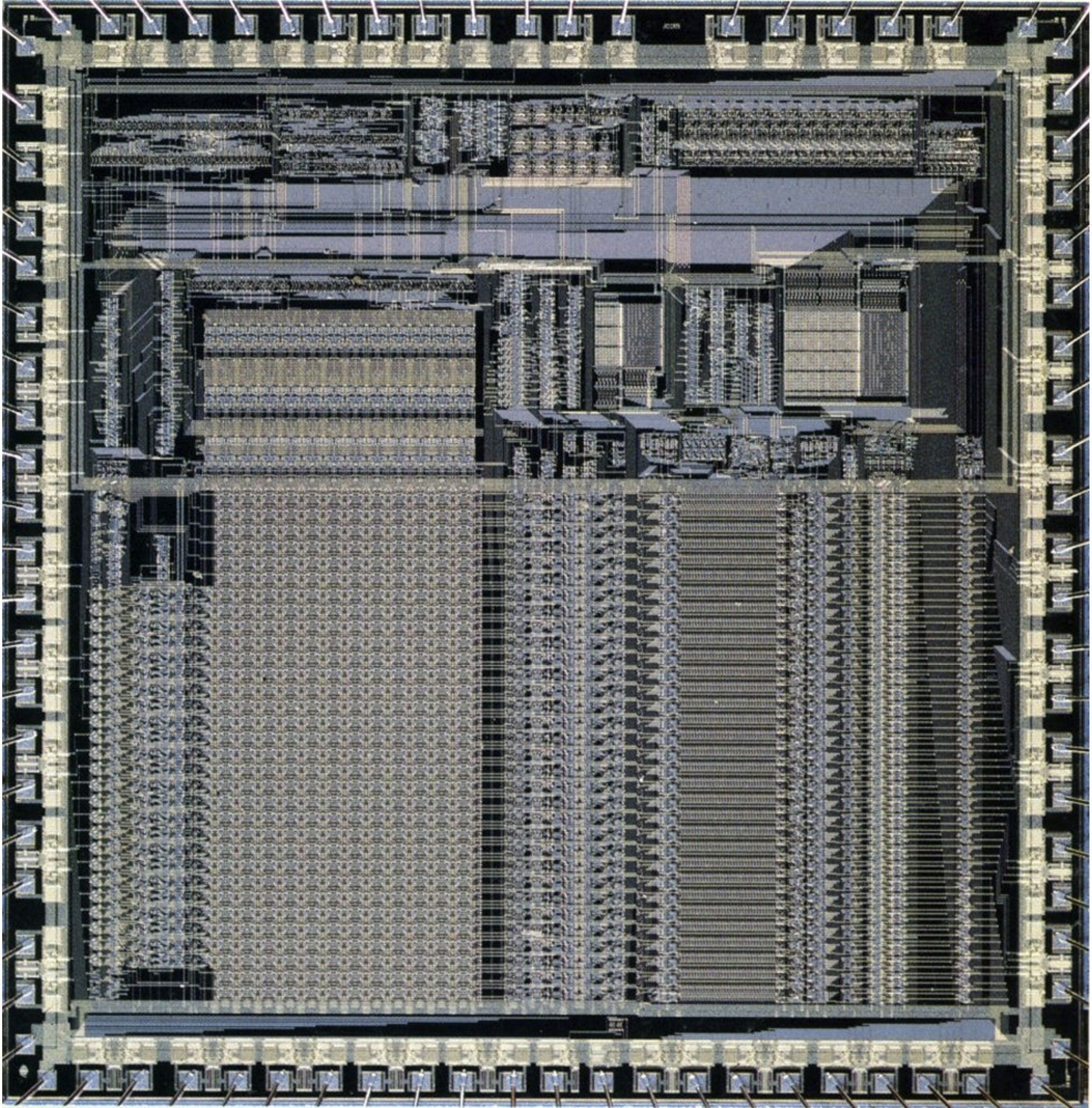
1978	96	8086 (3,2 µm nMOSFET)	
1980	83	8087 (3 µm)	
1982	105	80186	
1982	112	80286	
1982	84	80287	
1985	166	80386 (1,5 µm CHMOS III)	
1987	96	80387	
1989	267	80486DX/P4 (1 µm CHMOS IV)	FPU
1993	273	80586/P5/Pentium(0,8µ BiCMOS)	FPU
1995	304	80686/P6/Pentium Pro (350 nm)	FPU
1997	321	Pentium MMX (280 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

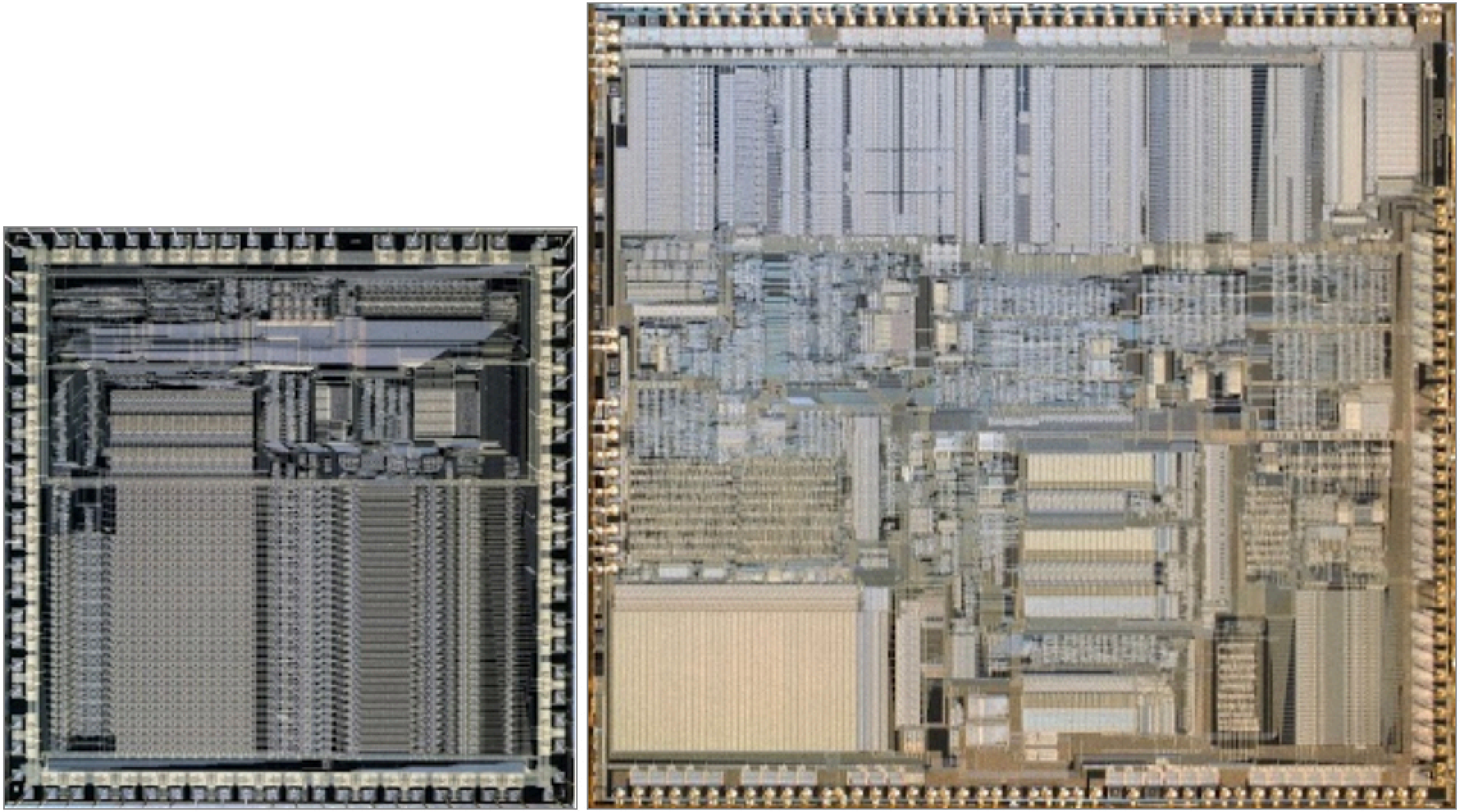
Октомври 2015 г.: Xeon E3 v5 Skylake-DT (14 nm FinFET):
208 + 5 (CLMUL = Carry-less Multiplication) + 24 (BMI = Bit Manipulation Instructions)
+ 96 (FPU) + 20 (FMA = Fused Multiply-Add) =
+ 48 (MMX = Multimedia Extensions | Multiple Math Extensions | Matrix Math Extensions)
+ 68 (SSE = Streaming SIMD (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
01310


WWW.NIX.RU
000911549
(495) 974 3333
Не срывать
I00

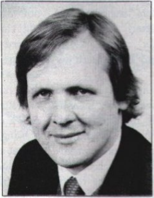

8625V
VC012
VC2588-0001
AUTUMN
8430





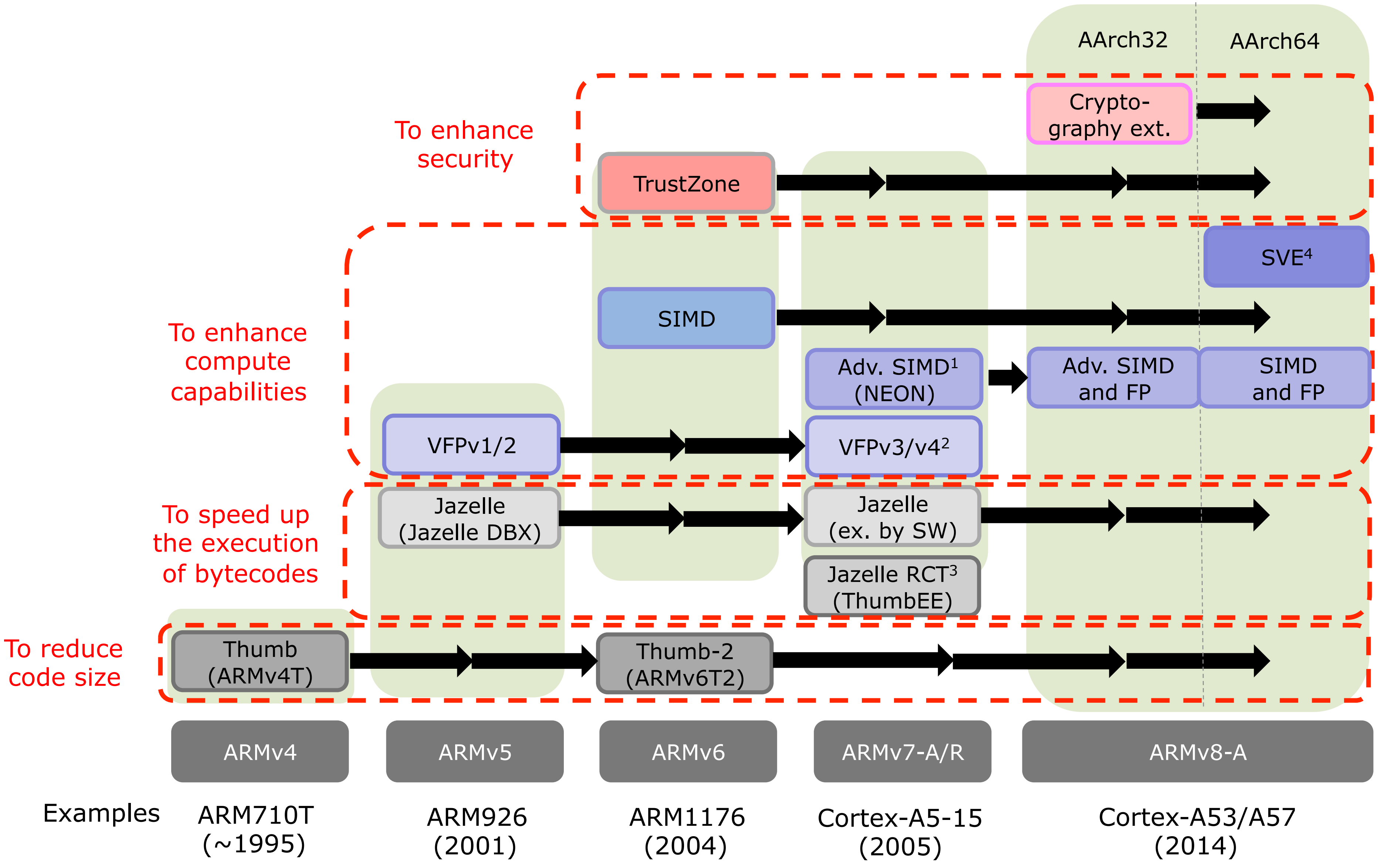
Die photos of the ARM1 processor and the Intel 386 processor to the same scale. The ARM1 is much smaller and contained 25,000 transistors compared to 275,000 in the 386. The 386 was higher density, with a 1.5 micron process compared to 3 micron for the ARM1. ARM1 photo courtesy of Computer History Museum.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



2.1 Overview (4)

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



AArch32

AArch64

To enhance security

Crypto-graphy ext.

TrustZone

To enhance compute capabilities

SVE[4]

SIMD

Adv. SIMD[1] (NEON)

Adv. SIMD and FP

SIMD and FP

VFPv1/2

VFPv3/v4[2]

To speed up the execution of bytecodes

Jazelle (Jazelle DBX)

Jazelle (ex. by SW)

Jazelle RCT[3] (ThumbEE)

To reduce code size

Thumb (ARMv4T)

Thumb-2 (ARMv6T2)

ARMv4

ARMv5

ARMv6

ARMv7-A/R

ARMv8-A

Examples

ARM710T (~1995)

ARM926 (2001)

ARM1176 (2004)

Cortex-A5-15 (2005)

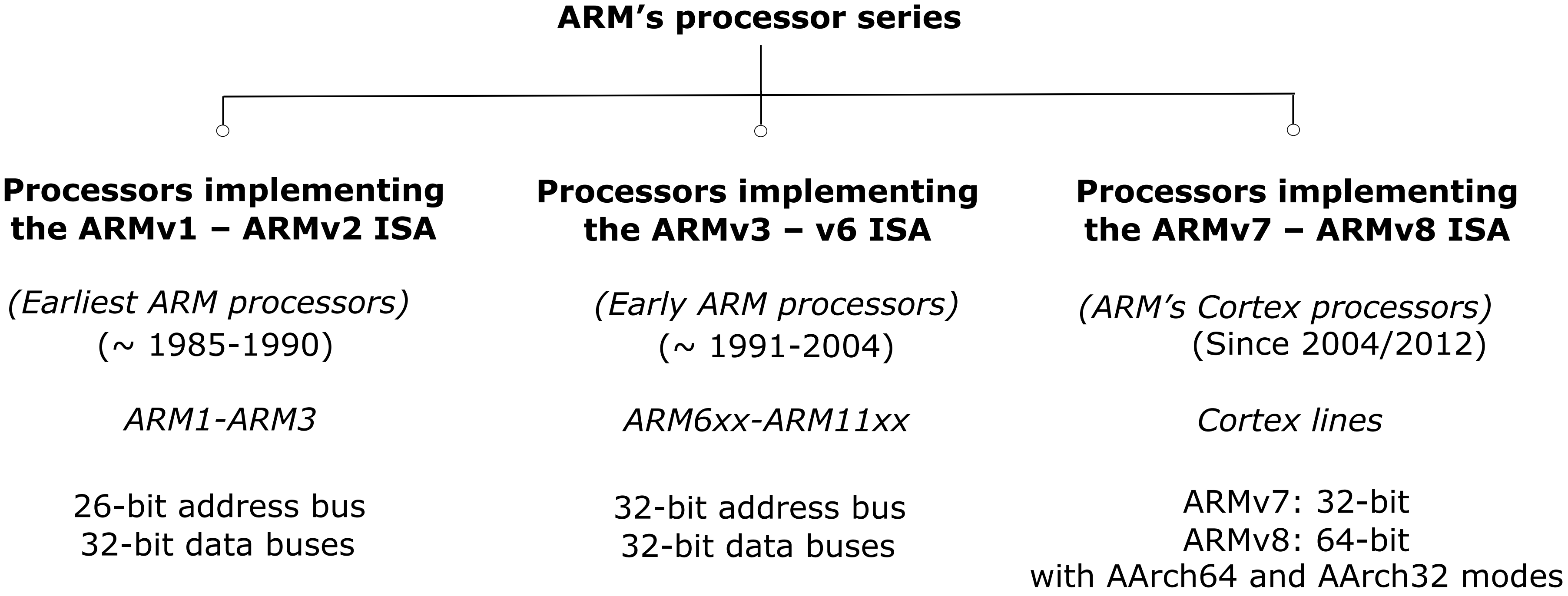
Cortex-A53/A57 (2014)

Remarks: See on the next slide.

3.1 Overview of ARM’s processor lines (1)

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

(Earliest ARM processors)
(~ 1985-1990)

ARM1-ARM3

26-bit address bus
32-bit data buses

Processors implementing the ARMv3 – v6 ISA

(Early ARM processors)
(~ 1991-2004)

ARM6xx-ARM11xx

32-bit address bus
32-bit data buses

Processors implementing the ARMv7 – ARMv8 ISA

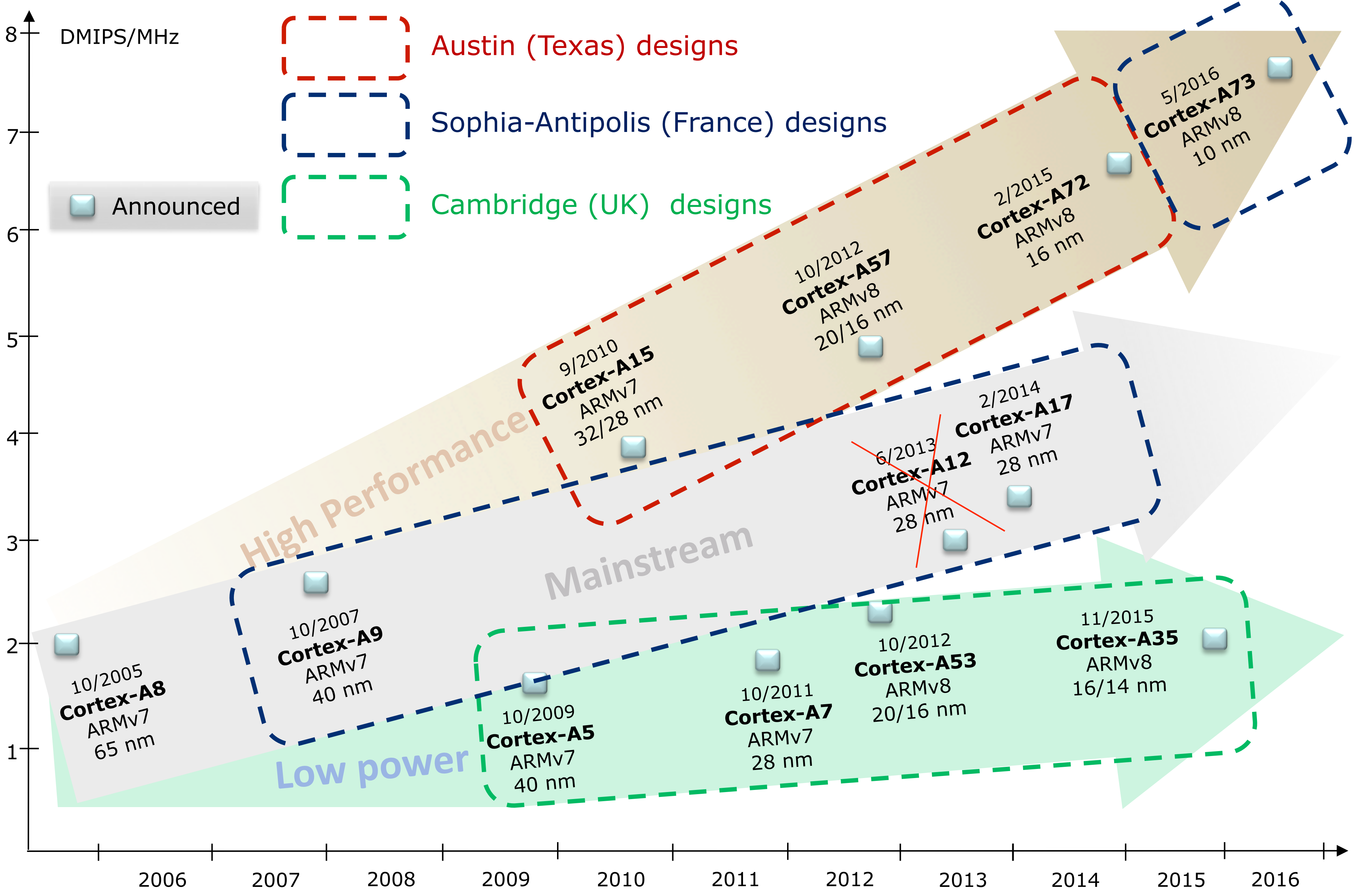
(ARM’s Cortex processors)
(Since 2004/2012)

Cortex lines

ARMv7: 32-bit
ARMv8: 64-bit
with AArch64 and AArch32 modes

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

Three design teams working in parallel []



Announced

Austin (Texas) designs

Sophia-Antipolis (France) designs

Cambridge (UK) designs

10/2005
Cortex-A8
ARMv7
65 nm

10/2007
Cortex-A9
ARMv7
40 nm

10/2009
Cortex-A5
ARMv7
40 nm

9/2010
Cortex-A15
ARMv7
32/28 nm

10/2011
Cortex-A7
ARMv7
28 nm

10/2012
Cortex-A57
ARMv8
20/16 nm

10/2012
Cortex-A53
ARMv8
20/16 nm

~~6/2013~~
~~**Cortex-A12**~~
~~ARMv7~~
~~28 nm~~

2/2014
Cortex-A17
ARMv7
28 nm

2/2015
Cortex-A72
ARMv8
16 nm

11/2015
Cortex-A35
ARMv8
16/14 nm

5/2016
Cortex-A73
ARMv8
10 nm

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

Програмен модел на МП: Понятие за програмен модел. Режими. Регистри за обща употреба. Специализирани регистри. Флагове на регистъра за кода на условието (РКУ). Особености. Обзор на програмния модел на други МП.

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.

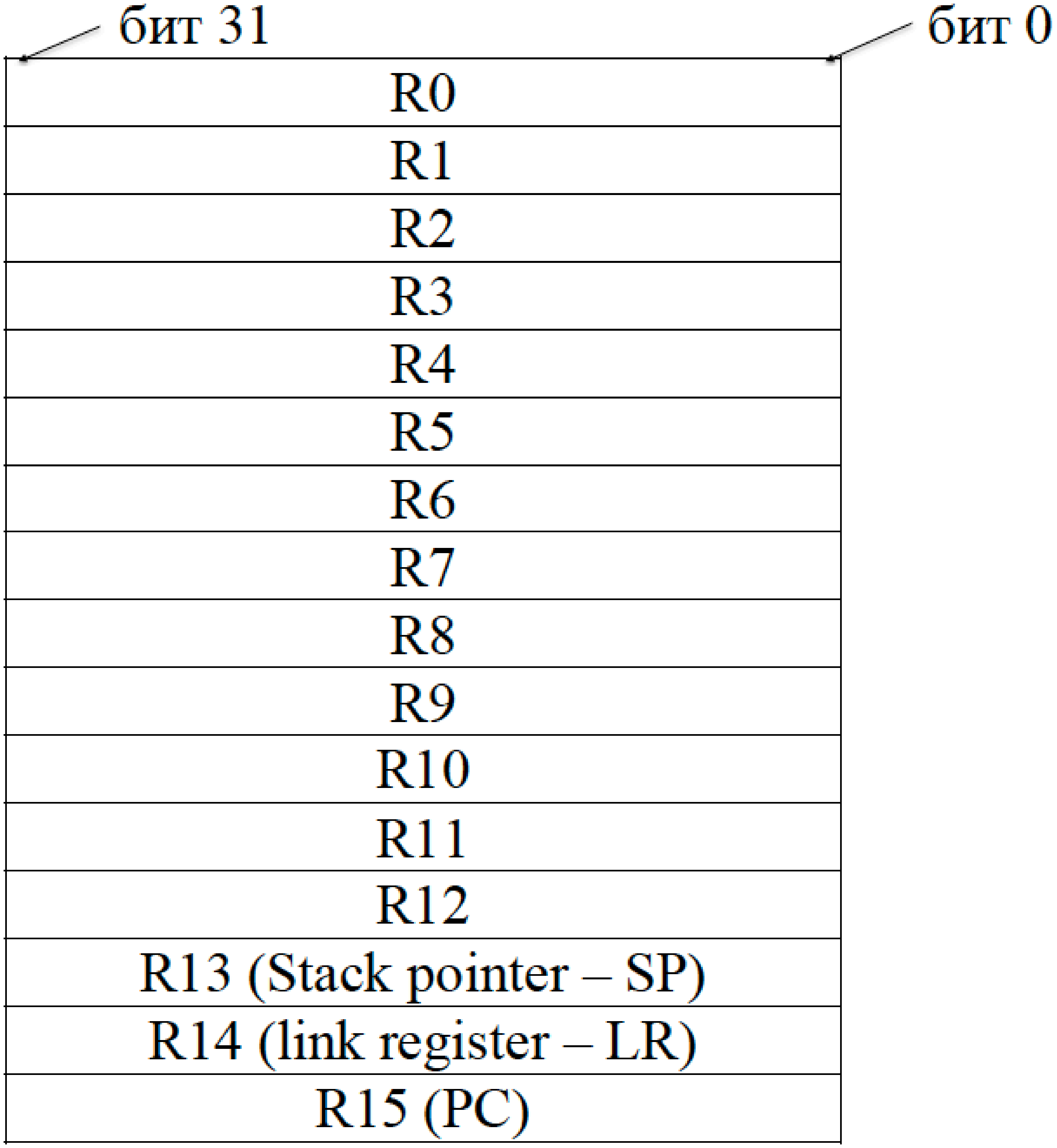
User/ System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	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.

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.



Фигура 12.1. Регистри на процесора ARM

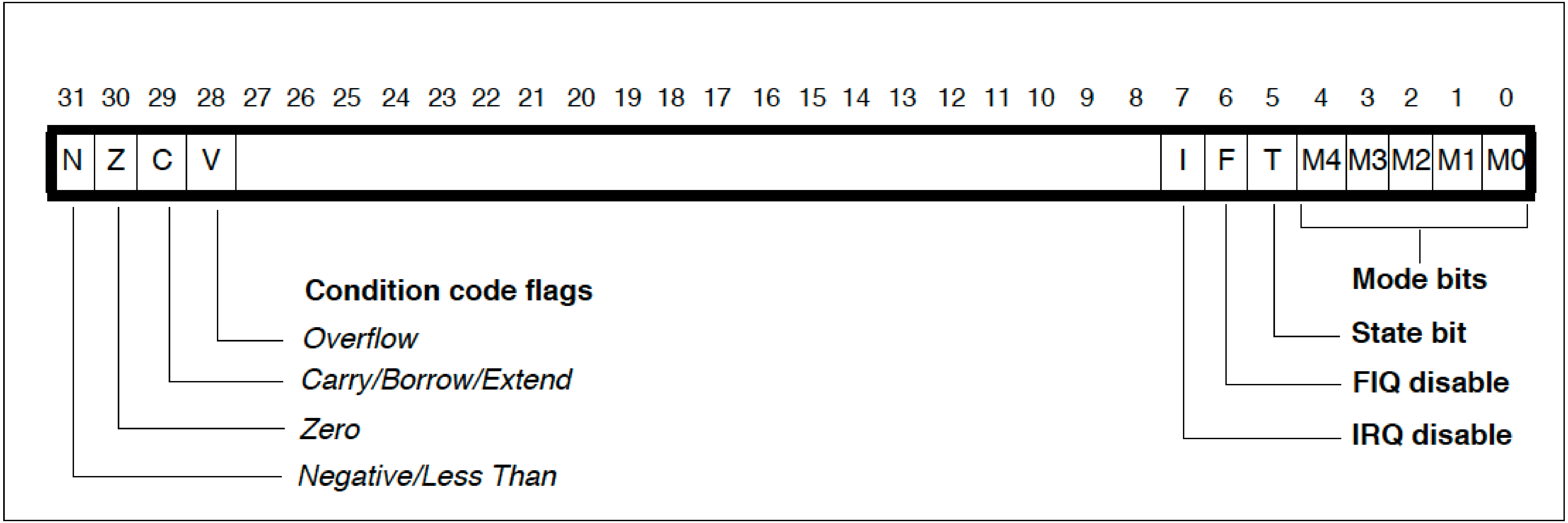


Figure 3-4: Program Status Register format

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.

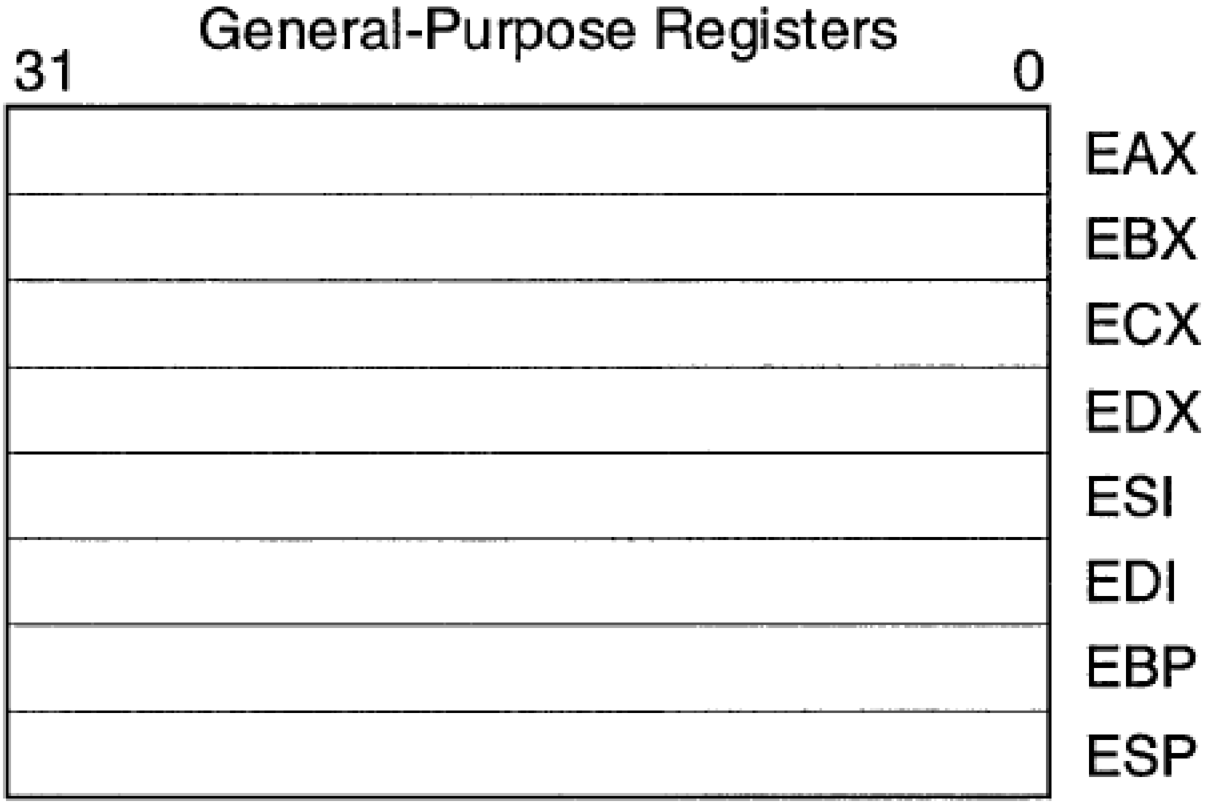
Interrupt disable bits	The I and F bits are the *interrupt disable bits*. When set, these disable the IRQ and FIQ interrupts respectively.
The state bit	Bit T is the processor *state bit*. When the state bit is set to 0, this indicates that the processor is in ARM state (ie. executing 32-bit ARM instructions). 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.

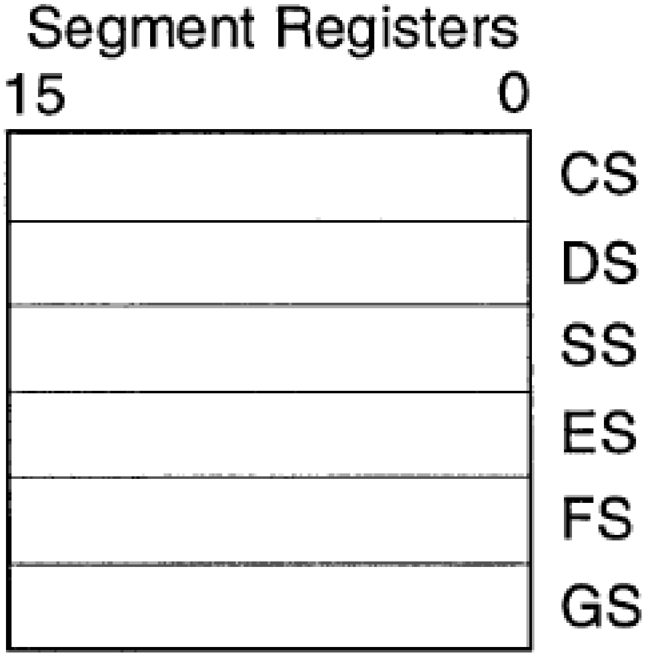
..

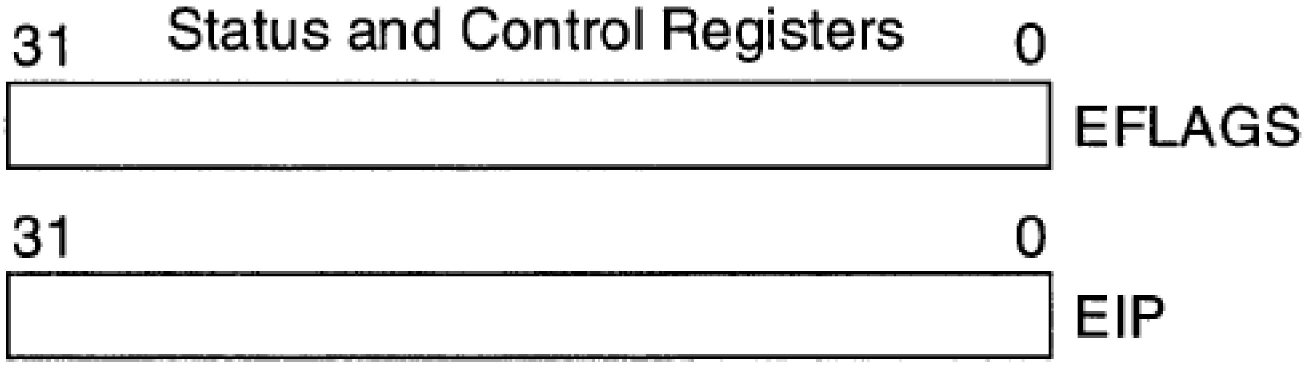
M[4:0]	Mode	Accessible Registers
10000	User	PC, R14 to R0, CPSR
10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq, R13_irq,R12 to R0, CPSR, SPSR_irq
10011	SVC	PC, R14_svc, R13_svc,R12 to R0, CPSR, SPSR_svc
10111	Abort	PC, R14_abt, R13_abt,R12 to R0, CPSR, SPSR_abt
11011	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“.


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

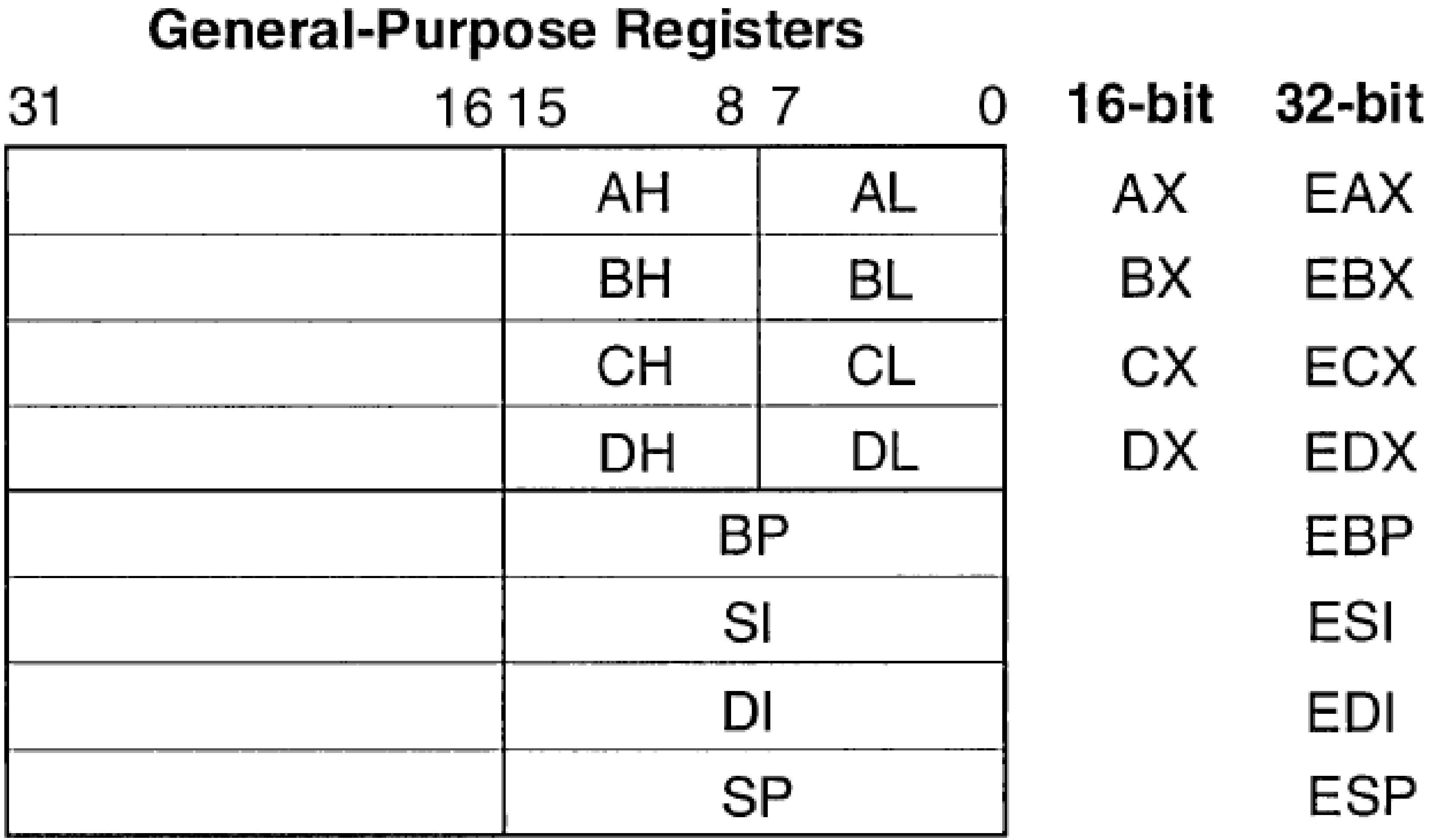


Figure 3-4. Alternate General-Purpose Register Names

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.

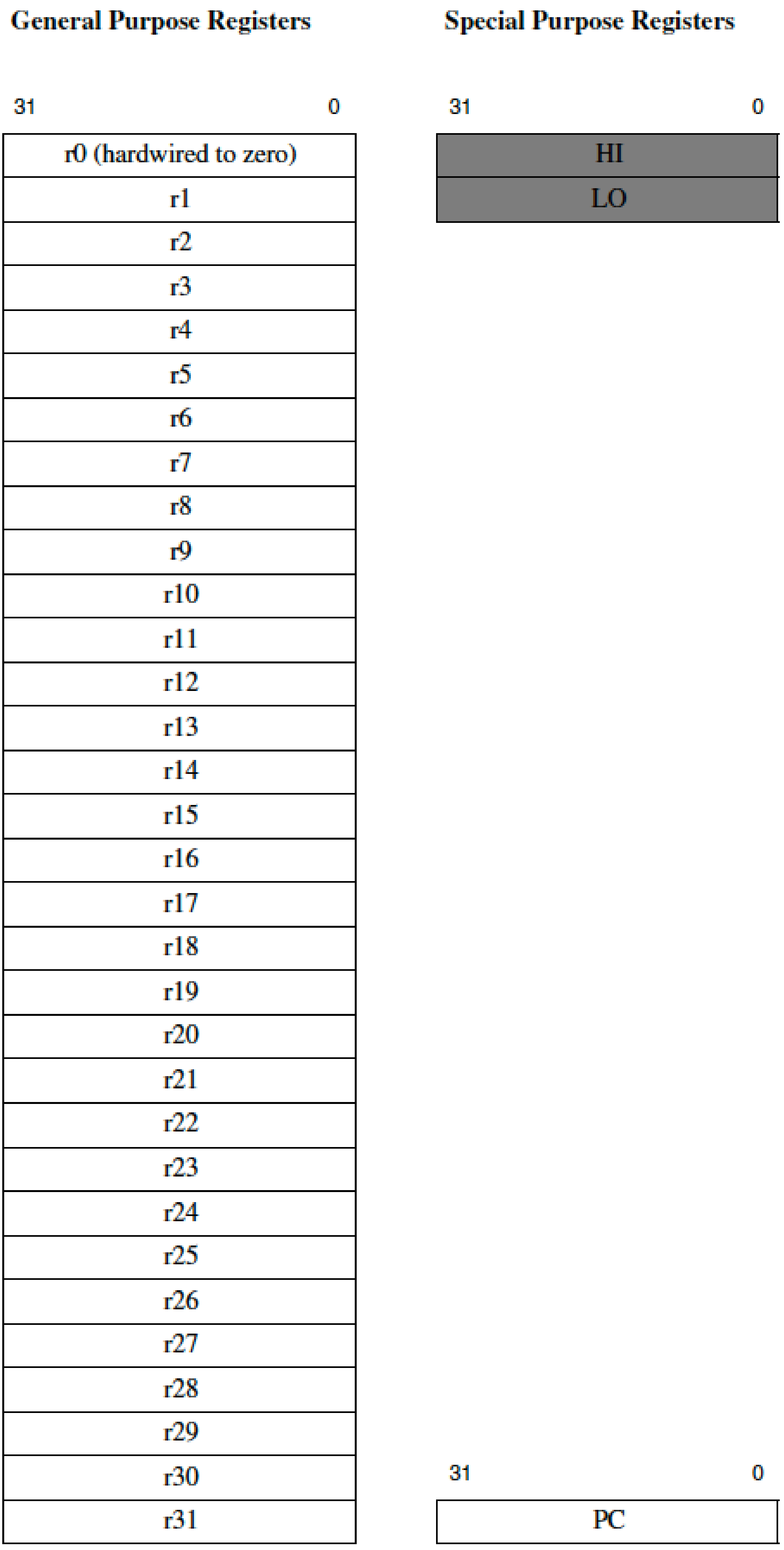
2. Няма команда, която да променя флаг Z, без да променя флаг C. Това затруднява запазването на преноса между итерациите на цикъла.

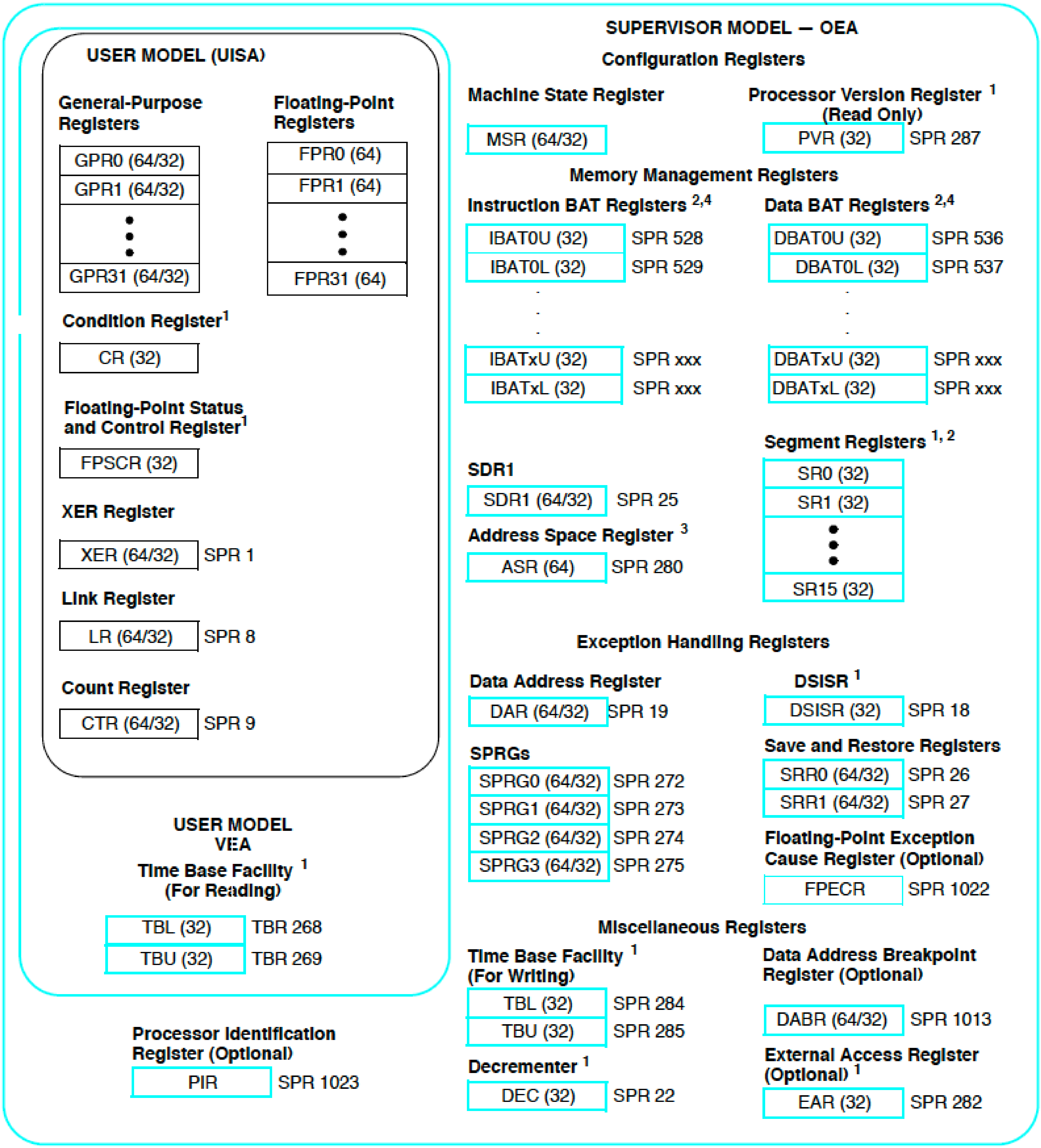
3. Няма команда за размяна на съдържанието на 2 регистъра.

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.)

Figure 4.1 CPU Registers for MIPS32





1. These registers are 32-bit registers only.
2. These registers are on 32-bit implementations only.
3. These registers are on 64-bit implementations only.
4. These registers are implementation dependent.
5. 64-bit registers operating in 32-bit mode clear the high order 32-bits.

Система от машинни команди: Групи команди. Формат на командите. „Операнд 2“. Методи за адресация. Ортогоналност на системата команди.

The ARM instruction set can be divided into six broad classes of instruction:

- *Branch instructions*
- *Data-processing instructions* on page A1-7
- *Status register transfer instructions* on page A1-8
- *Load and store instructions* on page A1-8
- *Coprocessor instructions* on page A1-10
- *Exception-generating instructions* on page A1-10.

Most data-processing instructions and one type of coprocessor instruction can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

Almost all ARM instructions contain a 4-bit *condition* field. One value of this field specifies that the instruction is executed unconditionally.

Fourteen other values specify *conditional execution* of the instruction. If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing. The 14 available conditions allow:

- tests for equality and non-equality
- tests for <, <=, >, and >= inequalities, in both signed and unsigned arithmetic
- each condition code flag to be tested individually.

The sixteenth value of the condition field encodes alternative instructions. These do not allow conditional execution. Before ARMv5 these instructions were UNPREDICTABLE.

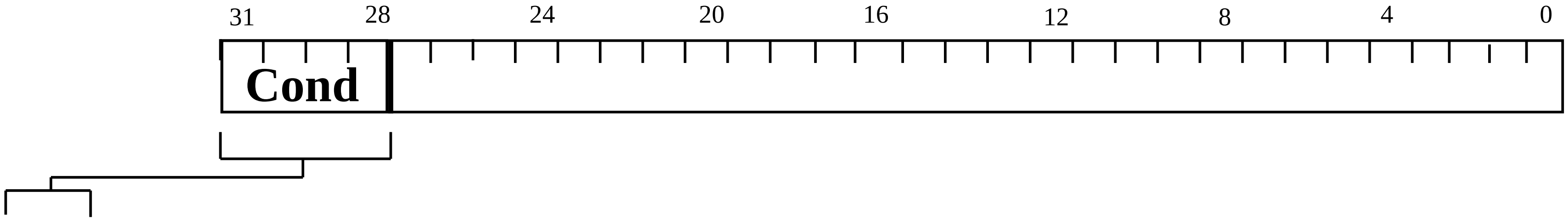
ARM Instruction Set Format

31 28	27	16	15	8	7	0	**Instruction type**
Cond	0 0 I Opcode S	Rn	Rd	Operand2			Data processing / PSR Transfer
Cond	0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Cond	0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm	Long Multiply **(v3M / v4 only)**
Cond	0 0 0 1 0 B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	Swap
Cond	0 1 I P U B W L	Rn	Rd	Offset			Load/Store Byte/Word
Cond	1 0 0 P U S W L	Rn	Register List				Load/Store Multiple
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2	Halfword transfer : Immediate offset **(v4 only)**
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm	Halfword transfer: Register offset **(v4 only)**
Cond	1 0 1 L	Offset					Branch
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn	Branch Exchange **(v4T only)**
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset		Coprocessor data transfer
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm	Coprocessor data operation
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm	Coprocessor register transfer
Cond	1 1 1 1	SWI Number					Software interrupt

Conditional Execution

- **Most instruction sets only allow branches to be executed conditionally.**
- **However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.**
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
    - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- **This removes the need for many branches, which stall the pipeline (3 cycles to refill).**
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Using and updating the Condition Field

- **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:

    ```
    – ADD r0,r1,r2        ; r0 = r1 + r2 (ADDAL)
    ```

  - To execute this only if the zero flag is set:

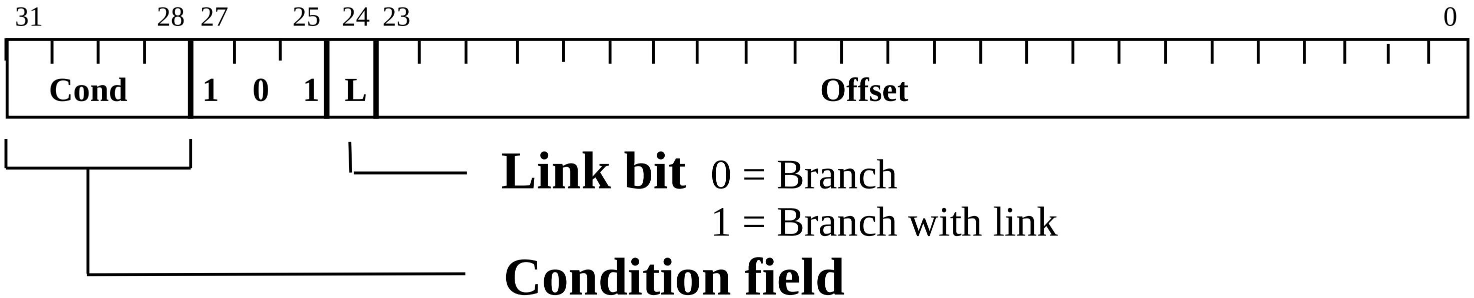
    ```
    – ADDEQ r0,r1,r2      ; If zero flag set then…
                          ; ... r0 = r1 + r2
    ```

- **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.**
  - For example to add two numbers and set the condition flags:

    ```
    – ADDS r0,r1,r2       ; r0 = r1 + r2
                          ; ... and set flags
    ```

Branch instructions (1)

- * **Branch :** `B{<cond>} label`
- * **Branch with Link :** `BL{<cond>} sub_routine_label`



Link bit 0 = Branch
1 = Branch with link

Condition field

- * **The offset for branch instructions is calculated by the assembler:**
  - By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
  - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word – aligned) and stored into the instruction encoding.
  - This gives a range of ± 32 Mbytes.

Branch instructions (2)

- **When executing the instruction, the processor:**
  - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- **Execution then continues from the new PC, once the pipeline has been refilled.**
- **The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.**
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- **To return from subroutine, simply need to restore the PC from the LR:**
  - `MOV pc, lr`
  - Again, pipeline has to refill before execution continues.
- **The "Branch" instruction does not affect LR.**
- **Note: Architecture 4T offers a further ARM branch instruction, BX**
  - See Thumb Instruction Set Module for details.

Data processing Instructions

- **Largest family of ARM instructions, all sharing the same instruction format.**
- **Contains:**
    - Arithmetic operations
    - Comparisons (no results - just set condition codes)
    - Logical operations
    - Data movement between registers
- **Remember, this is a load / store architecture**
    - These instruction only work on registers, ***NOT*** memory.
- **They each perform a specific operation on one or two operands.**
    - First operand always a register - Rn
    - Second operand sent to the ALU via barrel shifter.
- **We will examine the barrel shifter shortly.**

Arithmetic Operations

- **Operations are:**

ADD	operand1 + operand2
ADC	operand1 + operand2 + carry
SUB	operand1 - operand2
SBC	operand1 - operand2 + carry -1
RSB	operand2 - operand1
RSC	operand2 - operand1 + carry - 1

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples**
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparisons

- **The only effect of the comparisons is to**
  - ***UPDATE THE CONDITION FLAGS*.** Thus no need to set S bit.
- **Operations are:**
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 EOR operand2, but result not written
- **Syntax:**
  - <Operation>{<cond>} Rn, Operand2
- **Examples:**
  - CMP r0, r1
  - TSTEQ r2, #5

Logical Operations

- **Operations are:**

AND	operand1 AND operand2
EOR	operand1 EOR operand2
ORR	operand1 OR operand2
BIC	operand1 AND NOT operand2 [ie bit clear]

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples:**

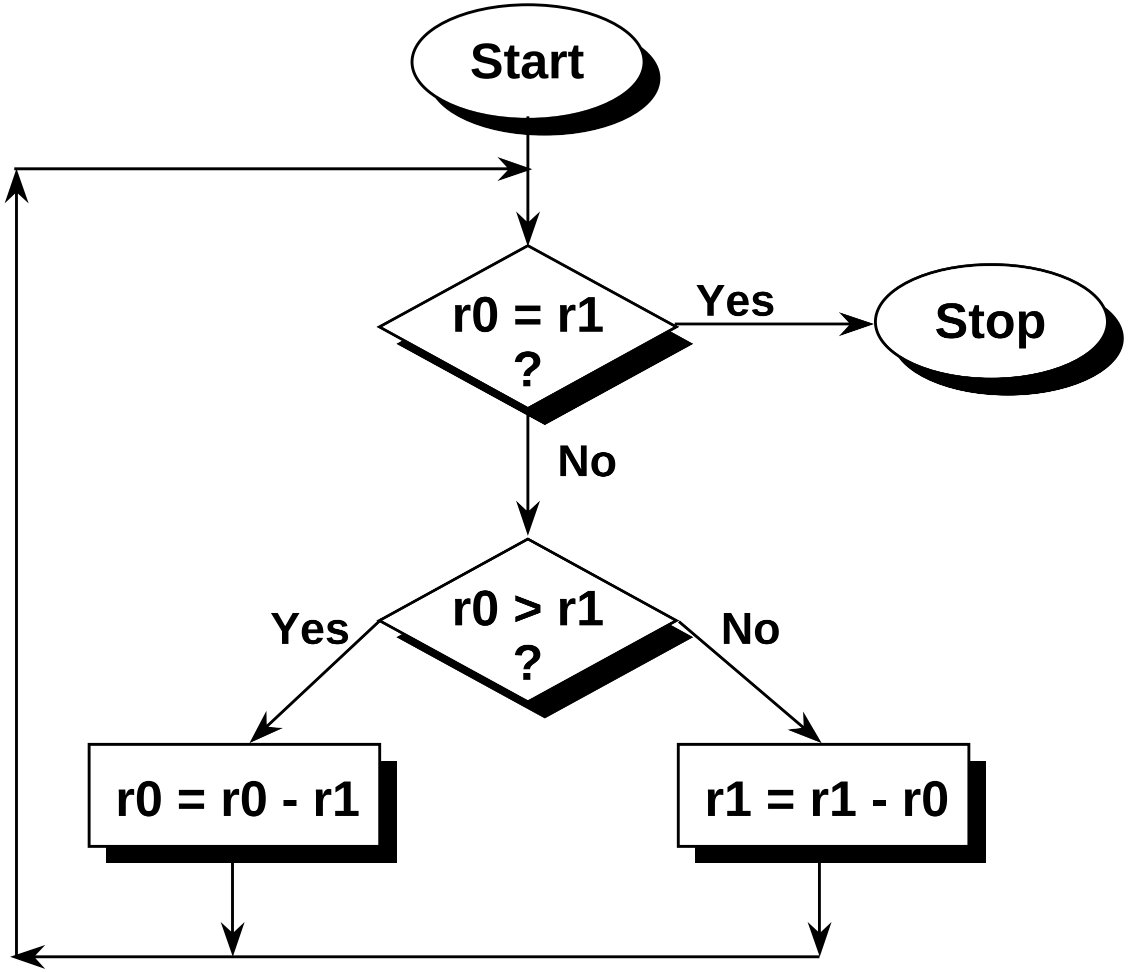
AND	r0, r1, r2
BICEQ	r2, r3, #7
EORS	r1,r3,r0

Data Movement

- **Operations are:**
  - MOV operand2
  - MVN NOT operand2

  Note that these make no use of operand1.
- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Operand2
- **Examples:**
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

Quiz #2



- **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

- **The only instructions you need are CMP, B and SUB.**

Quiz #2 - Sample Solutions

“Normal” Assembler

```
gcd    cmp r0, r1      ;reached the end?
       beq stop
       blt less        ;if r0 > r1
       sub r0, r0, r1  ;subtract r1 from r0
       bal gcd
less   sub r1, r1, r0  ;subtract r0 from r1
       bal gcd
stop
```

ARM Conditional Assembler

```
gcd    cmp   r0, r1      ;if r0 > r1
       subgt r0, r0, r1  ;subtract r1 from r0
       sublt r1, r1, r0  ;else subtract r0 from r1
       bne   gcd         ;reached the end?
```

The Barrel Shifter

- The ARM doesn’t have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?

Barrel Shifter - Left Shift

- **Shifts left by the specified amount (multiplies by powers of two) e.g.**

  LSL #5 = multiply by 32

Logical Shift Left (LSL)

Barrel Shifter - Right Shifts

Logical Shift Right

- Shifts right by the specified amount (divides by powers of two) e.g.

  LSR #5 = divide by 32

Logical Shift Right

...0

Destination

CF

Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

  ASR #5 = divide by 32

Arithmetic Shift Right

Destination

CF

Sign bit shifted in

Barrel Shifter - Rotations

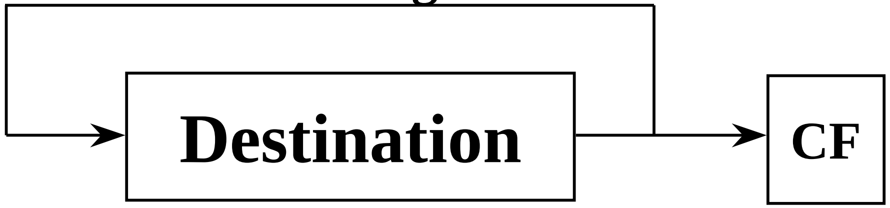
Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

  e.g. ROR #5

- Note the last bit rotated is also used as the Carry Out.

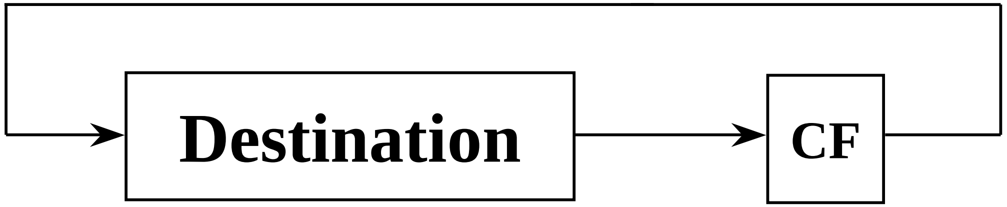
Rotate Right



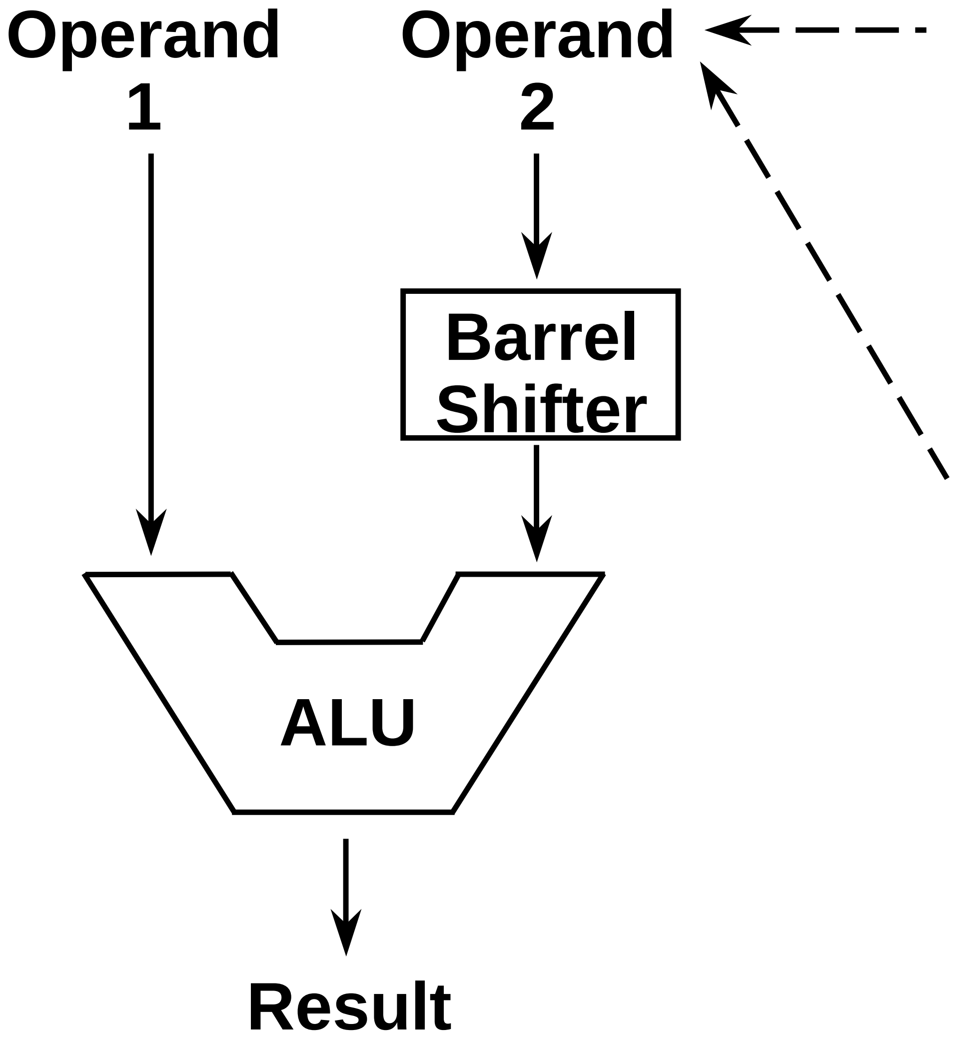
Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.

Rotate Right through Carry



Using the Barrel Shifter: The Second Operand



Operand 1

Operand 2

Barrel Shifter

ALU

Result

- **Register, optionally with shift operation applied.**
- **Shift value can be either be:**
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

- **Immediate value**
  - 8 bit number
  - Can be rotated right through an even number of positions.
  - Assembler will calculate rotate for you from constant.

Second Operand : Shifted Register

- **The amount by which the register is to be shifted is contained in either:**
  - the immediate 5-bit field in the instruction
    - *NO OVERHEAD*
    - Shift is done for free - executes in single cycle.
  - the bottom byte of a register (not PC)
    - Then takes extra cycle to execute
    - ARM doesn’t have enough read ports to read 3 registers at once.
    - Then same as on other processors where shift is separate instruction.
- **If no shift is specified then a default shift is applied: LSL #0**
  - i.e. barrel shifter has no effect on value in register.

Second Operand : Using a Shifted Register

- **Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.**
- **A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.**
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- **Example: r0 = r1 * 5**
  Example: r0 = r1 + (r1 * 4)

  ï ADD r0, r1, r1, LSL #2
- **Example: r2 = r3 * 105**
  Example: r2 = r3 * 15 * 7
  Example: r2 = r3 * (16 - 1) * (8 - 1)
  ï RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
  ï RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

Second Operand : Immediate Value (1)

- **There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.**
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data.
- **The data processing instruction format has 12 bits available for operand2**
  - If used directly this would only give a range of 4096.
- **Instead it is used to store 8 bit constants, giving a range of 0 - 255.**
- **These 8 bits can then be rotated right through an even number of positions (ie RORs by 0, 2, 4,..30).**
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

Second Operand : Immediate Value (2)

- **This gives us:**
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
- **These can be loaded using, for example:**
  - MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)
- **To make this easier, the assembler will convert to this form for us if simply given the required constant:**
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
- **The bitwise complements can also be formed using MVN:**
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
- **If the required constant cannot be generated, an error will be reported.**

Loading full 32 bit constants

- **Although the MOV/MVN mechansim will load a large range of constants into a register, sometimes this mechansim will not generate the required constant.**
- **Therefore, the assembler also provides a method which will load *ANY* 32 bit constant:**
  - `LDR rd,=numeric constant`
- **If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated.**
- **Otherwise, the assembler will produce an LDR instruction with a PC-relative address to read the constant from a literal pool.**
  - `LDR r0,=0x42       ; generates MOV r0,#0x42`
  - `LDR r0,=0x55555555 ; generate LDR r0,[pc, offset to lit pool]`
- **As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.**

Multiplication Instructions

- **The Basic ARM provides two multiplication instructions.**
- **Multiply**
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
- **Multiply Accumulate - does addition for free**
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- **Restrictions on use:**
  - Rd and Rm cannot be the same register
    - Can be avoid by swapping Rm and Rs around. This works because multiplication is commutative.
  - Cannot use PC.

  These will be picked up by the assembler if overlooked.
- **Operands can be considered signed or unsigned**
  - Up to user to interpret correctly.

Multiplication Implementation

- **The ARM makes use of Booth’s Algorithm to perform integer multiplication.**
- **On non-M ARMs this operates on 2 bits of Rs at a time.**
  - For each pair of bits this takes 1 cycle (plus 1 cycle to start with).
  - However when there are no more 1’s left in Rs, the multiplication will early-terminate.
- **Example: Multiply 18 and -1 : Rd = Rm * Rs**

Rm	**18**	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 1 0	**18**	**Rs**
Rs	**-1**	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	**-1**	**Rm**
17 cycles											**4 cycles**

- **Note: Compiler does not use early termination criteria to decide on which order to place operands.**

Extended Multiply Instructions

- **M variants of ARM cores contain extended multiplication hardware. This provides three enhancements:**
  - An *8 bit Booth's Algorithm* is used
    - Multiplication is carried out faster (maximum for standard instructions is now 5 cycles).
  - *Early termination method improved* so that now completes multiplication when all remaining bit sets contain
    - all zeroes (as with non-M ARMs), or
    - all ones.

    Thus the previous example would early terminate in 2 cycles in both cases.
  - *64 bit results* can now be produced from two 32bit operands
    - Higher accuracy.
    - Pair of registers used to store result.

Multiply-Long and Multiply-Accumulate Long

- **Instructions are**
  - MULL which gives RdHi,RdLo:=Rm*Rs
  - MLAL which gives RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo
- **However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)**
  - Need to specify whether operands are signed or unsigned
- **Therefore syntax of new instructions are:**
  - UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
  - SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
- **Not generated by the compiler.**

Warning : Unpredictable on non-M ARMs.

Quiz #3

1. Specify instructions which will implement the following:

a) r0 = 16

b) r1 = r0 * 4

c) r0 = r1 / 16 (r1 signed 2's comp.)

d) r1 = r2 * 7

2. What will the following instructions do?

a) ADDS r0, r1, r1, LSL #2

b) RSB r2, r1, #0

3. What does the following instruction sequence do?

```
ADD r0, r1, r1, LSL #1
SUB r0, r0, r1, LSL #4
ADD r0, r0, r1, LSL #7
```

Load / Store Instructions

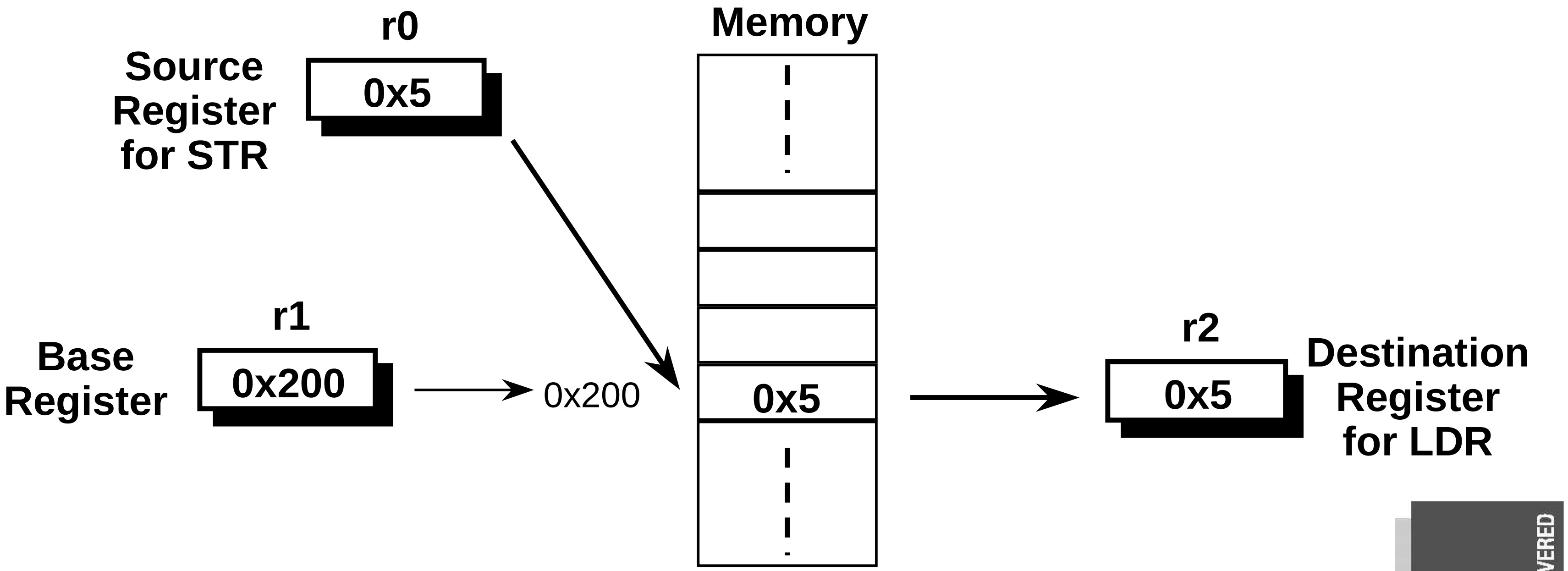
- **The ARM is a Load / Store Architecture:**
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- **This might sound inefficient, but in practice isn't:**
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- **The ARM has three sets of instructions which interact with main memory. These are:**
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

Single register data transfer

- **The basic load and store instructions are:**
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- **ARM Architecture Version 4 also adds support for halfwords and signed data.**
  - Load and Store Halfword
    - LDRH / STRH
  - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- **All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.**
  - e.g. LDREQB
- **Syntax:**
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load and Store Word or Byte: Base Register

* **The memory location to be accessed is held in a base register**
  - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.

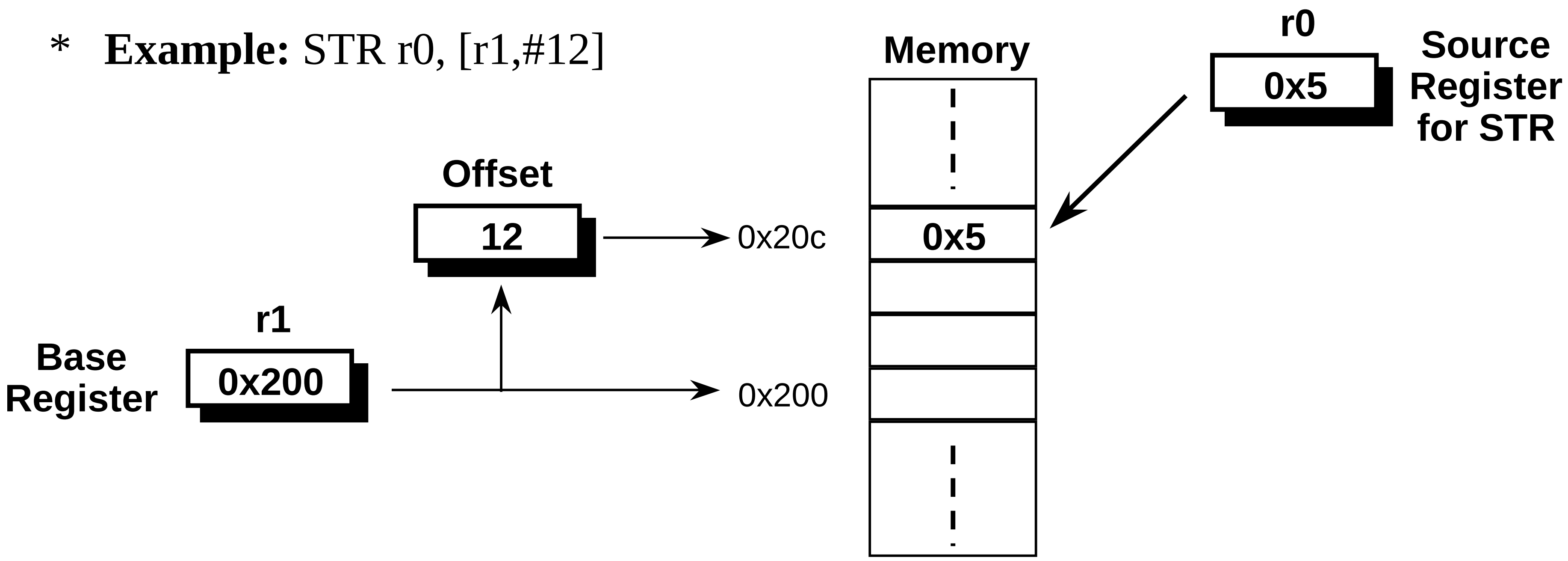


Load and Store Word or Byte: Offsets from the Base Register

- **As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.**
- **This offset can be**
  - An unsigned 12bit immediate value (ie 0 - 4095 bytes).
  - A register, optionally shifted by an immediate value
- **This can be either added or subtracted from the base register:**
  - Prefix the offset value or register with ‘**+**’ (default) or ‘**-**’.
- **This offset can be applied:**
  - before the transfer is made: ***Pre-indexed addressing***
    - optionally *auto-incrementing* the base register, by postfixing the instruction with an ‘**!**’.
  - after the transfer is made: ***Post-indexed addressing***
    - causing the base register to be *auto-incremented*.

Load and Store Word or Byte: Pre-indexed Addressing

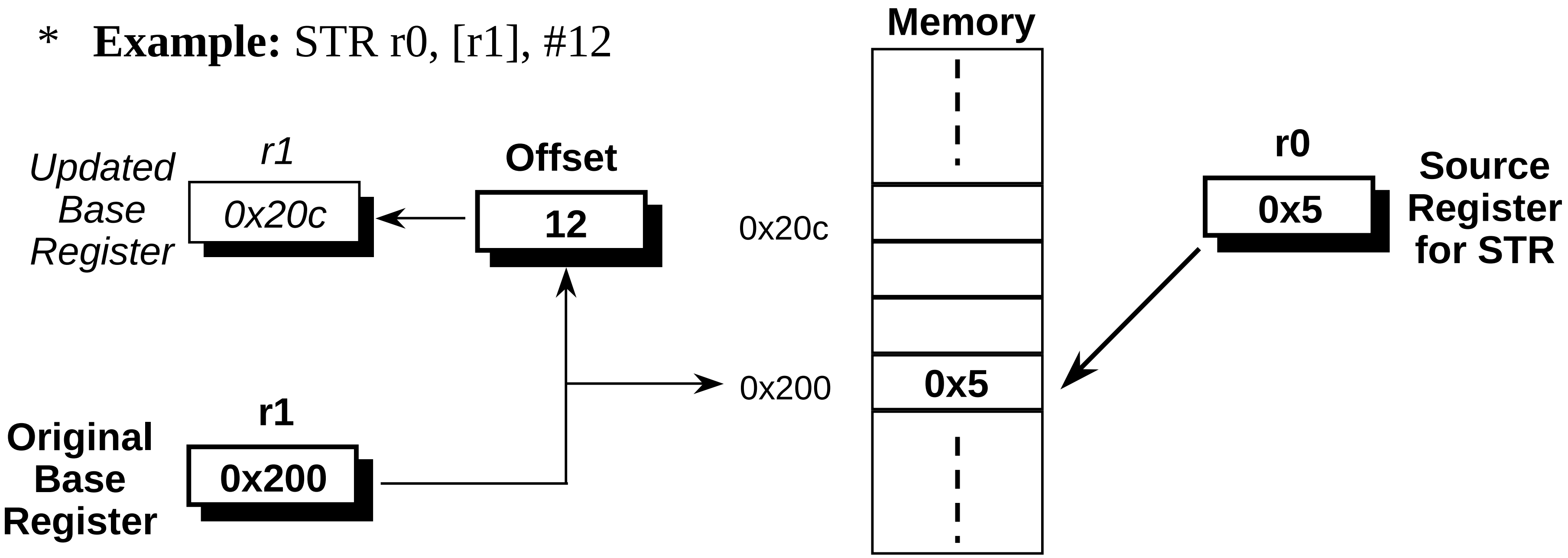
- **Example:** STR r0, [r1,#12]



- **To store to location 0x1f4 instead use:** STR r0, [r1,#-12]
- **To auto-increment base pointer to 0x20c use:** STR r0, [r1, #12]!
- **If r2 contains 3, access 0x20c by multiplying this by 4:**
  - STR r0, [r1, r2, LSL #2]

Load and Store Word or Byte: Post-indexed Addressing

- **Example:** STR r0, [r1], #12



Memory

Updated Base Register

r1

0x20c

Offset

12

0x20c

r0

0x5

Source Register for STR

0x200

0x5

r1

Original Base Register

0x200

- **To auto-increment the base register to location 0x1f4 instead use:**
  - STR r0, [r1], #-12
- **If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:**
  - STR r0, [r1], r2, LSL #2

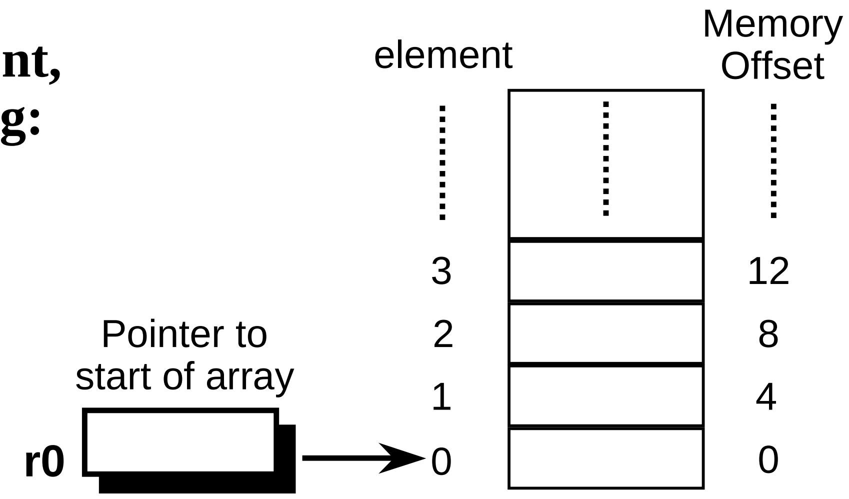
Load and Stores with User Mode Privilege

- **When using post-indexed addressing, there is a further form of Load/Store Word/Byte:**
  - <LDR|STR>{<cond>}{B}**T** Rd, <post_indexed_address>

- **When used in a privileged mode, this does the load/store with user mode privilege.**
  - Normally used by an exception handler that is emulating a memory access instruction that would normally execute in user mode.

Example Usage of Addressing Modes

- **Imagine an array, the first element of which is pointed to by the contents of r0.**
- **If we want to access a particular element, then we can use pre-indexed addressing:**
  - r1 is element we want.
  - LDR r2, [r0, r1, LSL #2]



- **If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:**
  - r1 is address of current element (initially equal to r0).
  - LDR r2, [r1], #4

Use a further register to store the address of final element, so that the loop can be correctly terminated.

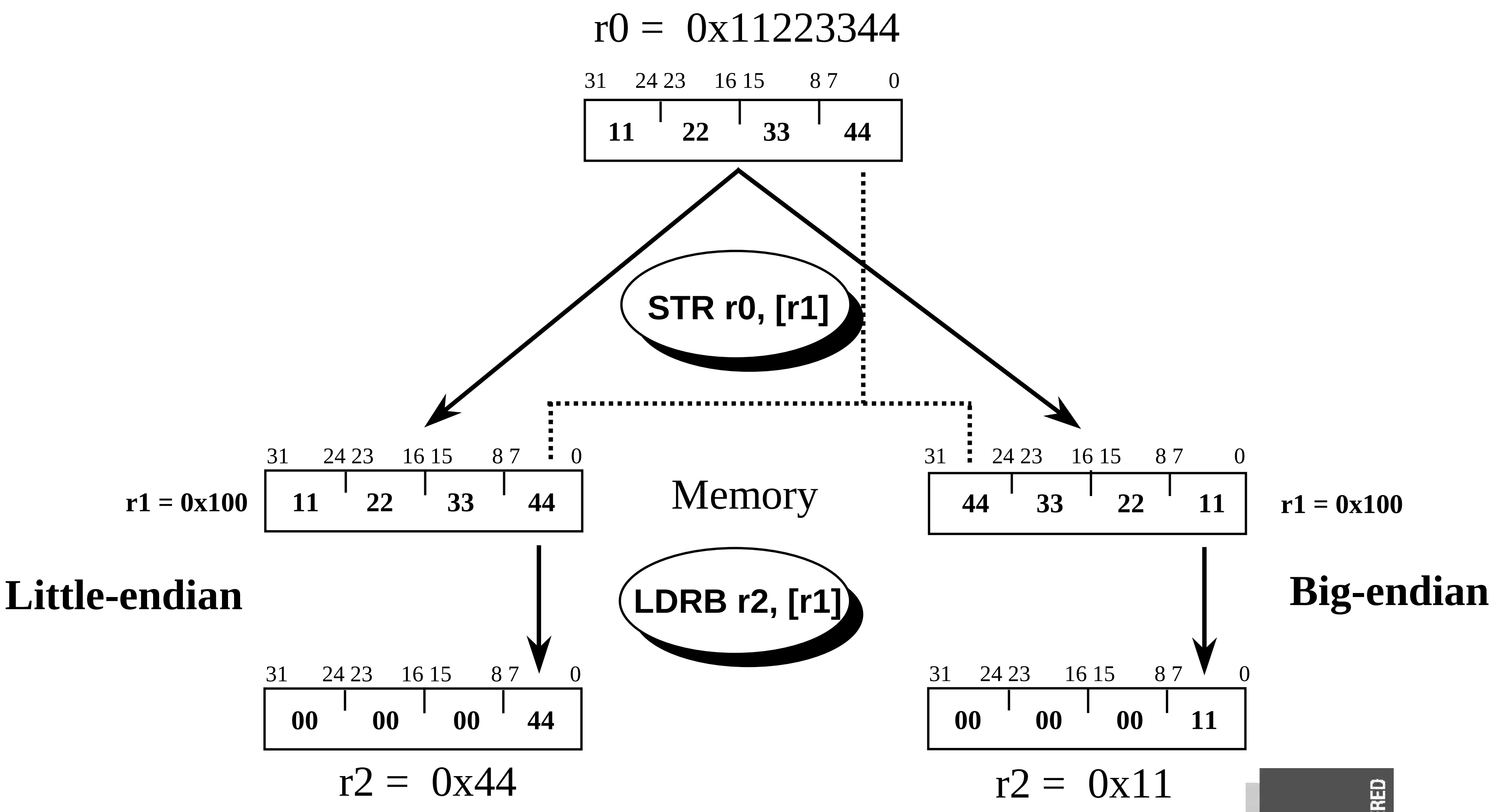
Offsets for Halfword and Signed Halfword / Byte Access

- **The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.**
- **However the actual offset formats are more constrained:**
  - The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.
  - The register form cannot have a shift applied to it.

Effect of endianess

- **The ARM can be set up to access its data in either little or big endian format.**
- **Little endian:**
  - Least significant byte of a word is stored in ***bits 0-7*** of an addressed word.
- **Big endian:**
  - Least significant byte of a word is stored in ***bits 24-31*** of an addressed word.
- **This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).**
  - Which byte / halfword is accessed will depend on the endianess of the system involved.

Endianess Example



r0 = 0x11223344

31 24 23 16 15 8 7 0

11 **22** **33** **44**

STR r0, [r1]

Little-endian

31 24 23 16 15 8 7 0

r1 = 0x100 **11** **22** **33** **44**

Memory

Big-endian

31 24 23 16 15 8 7 0

44 **33** **22** **11** **r1 = 0x100**

LDRB r2, [r1]

31 24 23 16 15 8 7 0

00 **00** **00** **44**

r2 = 0x44

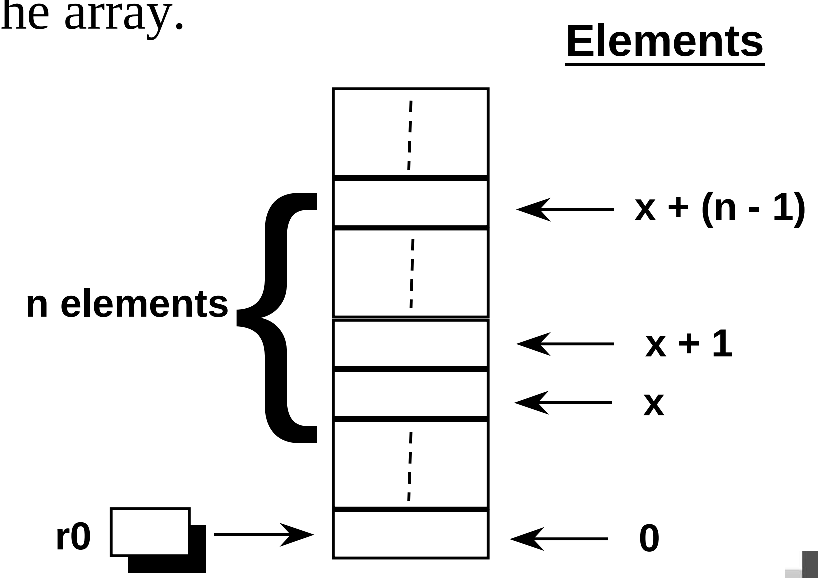
31 24 23 16 15 8 7 0

00 **00** **00** **11**

r2 = 0x11

Quiz #4

- **Write a segment of code that add together elements x to x+(n-1) of an array, where the element x=0 is the first element of the array.**
- **Each element of the array is word sized (ie. 32 bits).**
- **The segment should use post-indexed addressing.**
- **At the start of your segments, you should assume that:**
  - r0 points to the start of the array.
  - r1 = x
  - r2 = n

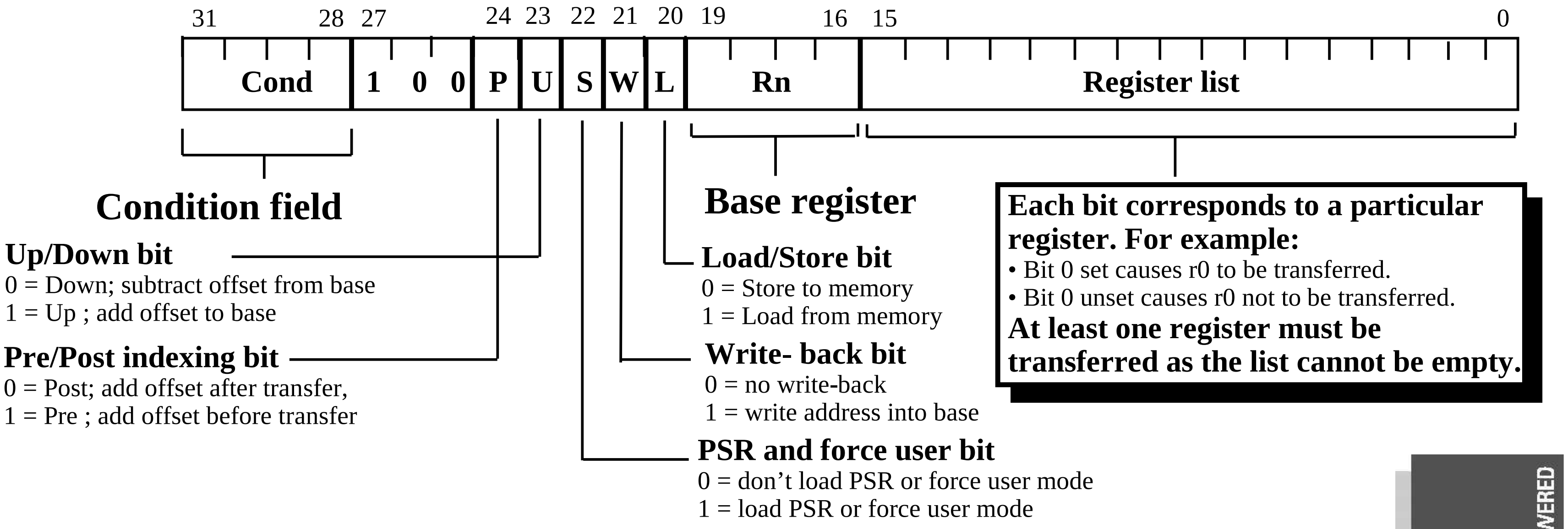


Quiz #4 - Sample Solution

```
__ADD r0, r0, r1, LSL#2        ; Set r0 to address of element x
  ADD r2, r0, r2, LSL#2        ; Set r2 to address of element n+1
  MOV r1, #0                   ; Initialise counter
loop
  LDR r3, [r0], #4             ; Access element and move to next
  ADD r1, r1, r3               ; Add contents to counter
  CMP r0, r2                   ; Have we reached element x+n?
  BLT loop                     ; If not - repeat for
                               ;       next element
  ; on exit sum contained in r1
```

Block Data Transfer (1)

- **The Load and Store Multiple instructions (LDM / STM) allow betweeen 1 and 16 registers to be transferred to or from memory.**
- **The transferred registers can be either:**
  - Any subset of the current bank of registers (default).
  - Any subset of the user mode bank of registers when in a priviledged mode (postfix instruction with a ‘^’).



31–28	27–25	24	23	22	21	20	19–16	15–0
Cond	**1 0 0**	**P**	**U**	**S**	**W**	**L**	**Rn**	**Register list**

Condition field

Up/Down bit
0 = Down; subtract offset from base
1 = Up ; add offset to base

Pre/Post indexing bit
0 = Post; add offset after transfer,
1 = Pre ; add offset before transfer

Base register

Load/Store bit
0 = Store to memory
1 = Load from memory

Write- back bit
0 = no write-back
1 = write address into base

PSR and force user bit
0 = don’t load PSR or force user mode
1 = load PSR or force user mode

Each bit corresponds to a particular register. For example:
- Bit 0 set causes r0 to be transferred.
- Bit 0 unset causes r0 not to be transferred.

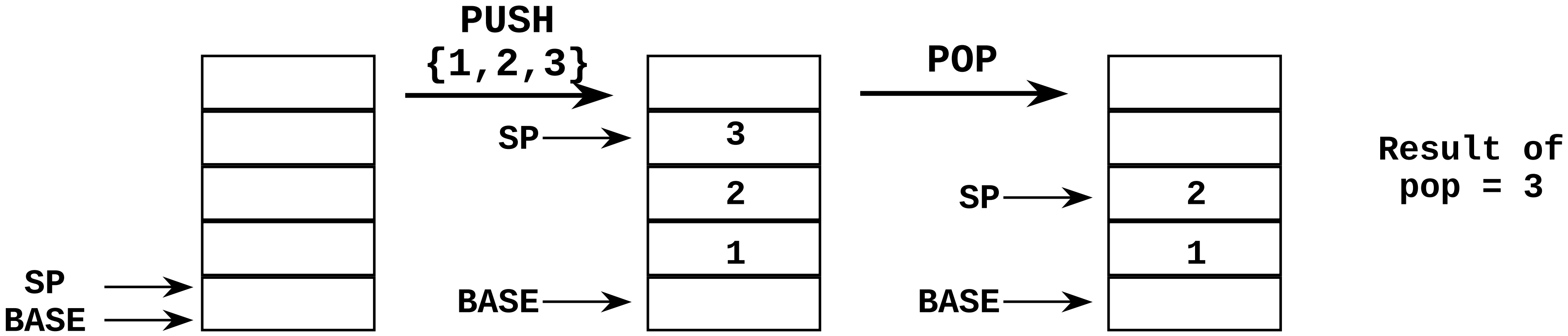
At least one register must be transferred as the list cannot be empty.

Block Data Transfer (2)

- **Base register used to determine where memory access should occur.**
  - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer (by appending it with an ‘**!**’.
  - Lowest register number is always transferred to/from lowest memory location accessed.
- **These instructions are very efficient for**
  - Saving and restoring context
    - For this useful to view memory as a stack.
  - Moving large blocks of data around memory
    - For this useful to directly represent functionality of the instructions.

Stacks

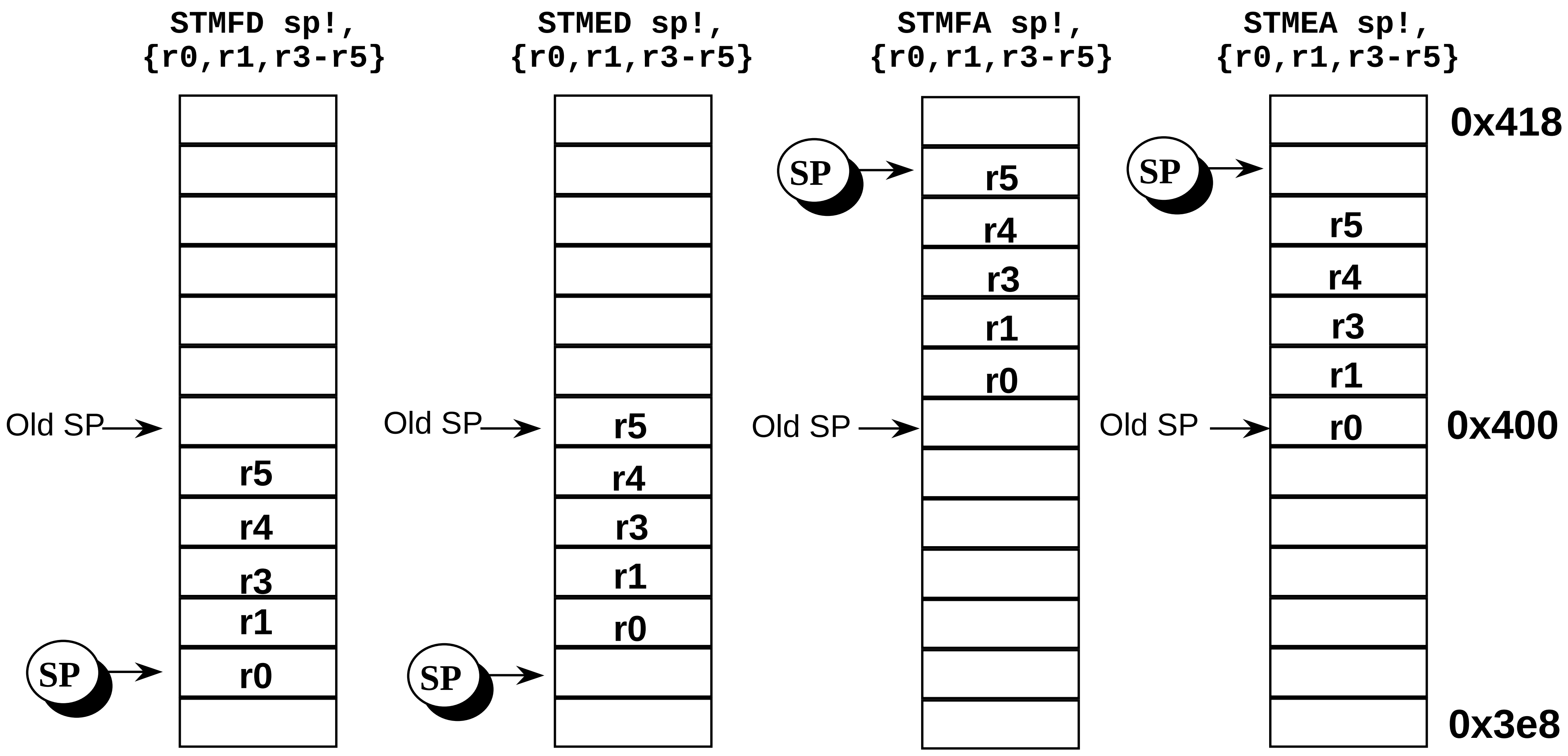
- **A stack is an area of memory which grows as new data is “pushed” onto the “top” of it, and shrinks as data is “popped” off the top.**
- **Two pointers define the current limits of the stack.**
  - A base pointer
    - used to point to the “bottom” of the stack (the first location).
  - A stack pointer
    - used to point the current “top” of the stack.



Stack Operation

- **Traditionally, a stack grows down in memory, with the last “pushed” value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory.**
- **The value of the stack pointer can either:**
  - Point to the last occupied address (Full stack)
    - and so needs pre-decrementing (ie before the push)
  - Point to the next occupied address (Empty stack)
    - and so needs post-decrementing (ie after the push)
- **The stack type to be used is given by the postfix to the instruction:**
  - STMFD / LDMFD : Full Descending stack
  - STMFA / LDMFA : Full Ascending stack.
  - STMED / LDMED : Empty Descending stack
  - STMEA / LDMEA : Empty Ascending stack
- **Note: ARM Compiler will always use a Full descending stack.**

Stack Examples



Stacks and Subroutines

- One use of stacks is to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller :

```
STMFD sp!,{r0-r12, lr}          ; stack all registers
........                        ; and the return address
........
LDMFD sp!,{r0-r12, pc}          ; load all the registers
                                ; and return automatically
```

- See the chapter on the ARM Procedure Call Standard in the SDT Reference Manual for further details of register usage within subroutines.
- If the pop instruction also had the ‘S’ bit set (using ‘^’) then the transfer of the PC when in a priviledged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

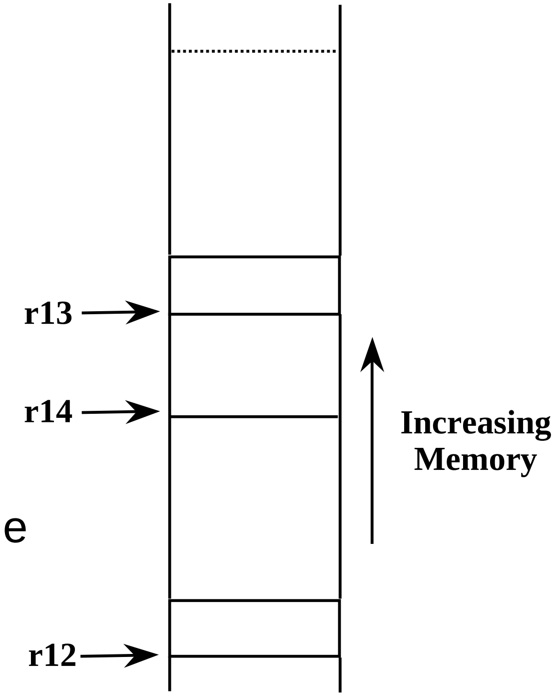
Direct functionality of Block Data Transfer

- **When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:**
  - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- **In order to do this, LDM / STM support a further syntax in addition to the stack one:**
  - STMIA / LDMIA : Increment After
  - STMIB / LDMIB : Increment Before
  - STMDA / LDMDA : Decrement After
  - STMDB / LDMDB : Decrement Before

Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11}  ; load 48 bytes
        STMIA   r13!, {r0-r11}  ; and store them
        CMP     r12, r14        ; check for the end
        BNE     loop            ; and loop until done
```

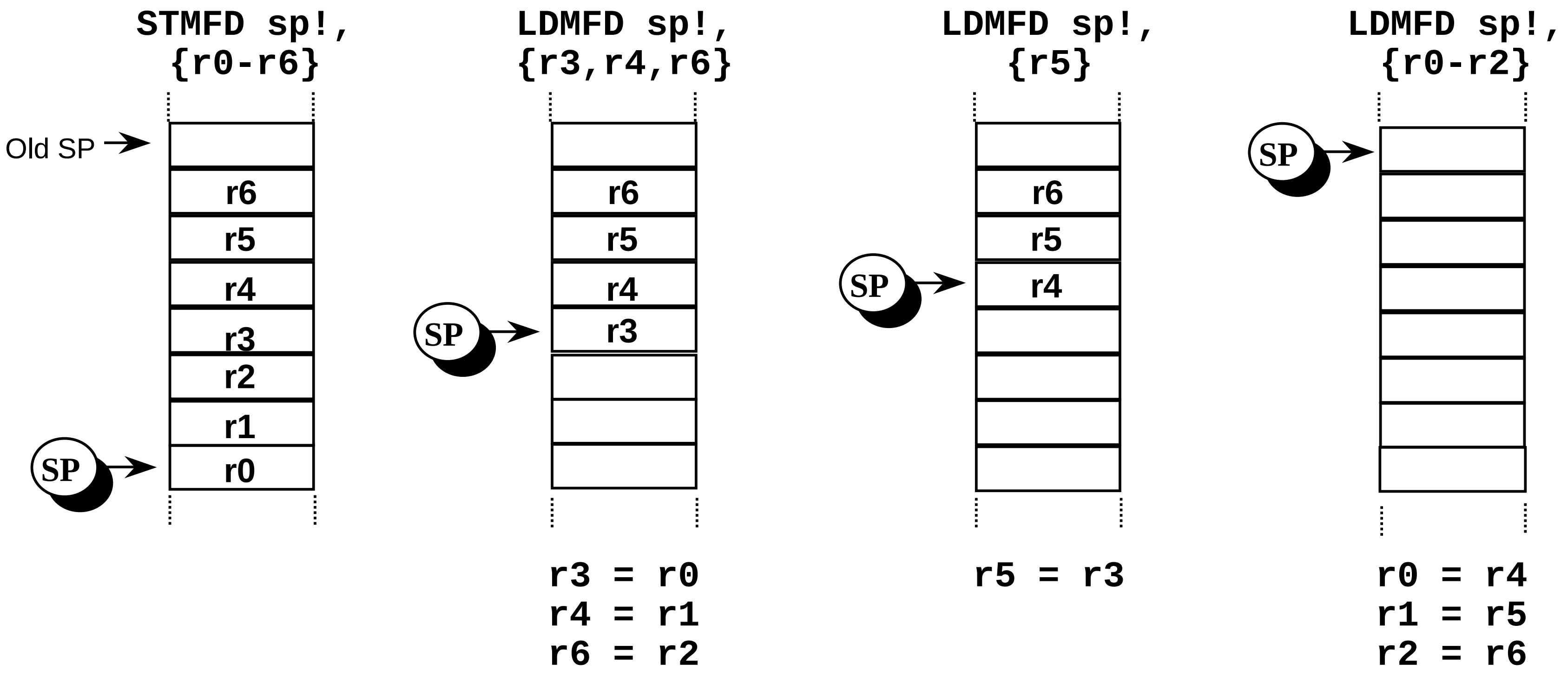


- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

Quiz #5

- **The contents of registers r0 to r6 need to be swapped around thus:**
  - r0 moved into r3
  - r1 moved into r4
  - r2 moved into r6
  - r3 moved into r5
  - r4 moved into r0
  - r5 moved into r1
  - r6 moved into r2
- **Write a segment of code that uses full descending stack operations to carry this out, and hence requires no use of any other registers for temporary storage.**

Quiz #5 - Sample Solution



`STMFD sp!, {r0-r6}`

Old SP

r6
r5
r4
r3
r2
r1
r0

SP

`LDMFD sp!, {r3,r4,r6}`

r6
r5
r4
r3

SP

```
r3 = r0
r4 = r1
r6 = r2
```

`LDMFD sp!, {r5}`

r6
r5
r4

SP

```
r5 = r3
```

`LDMFD sp!, {r0-r2}`

SP

```
r0 = r4
r1 = r5
r2 = r6
```

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.

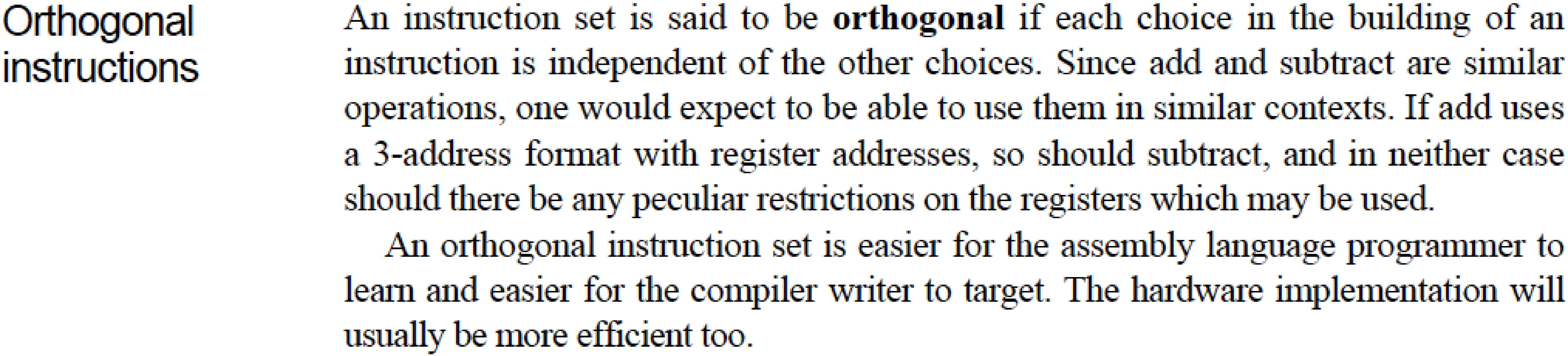
Instruction set orthogonality

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.

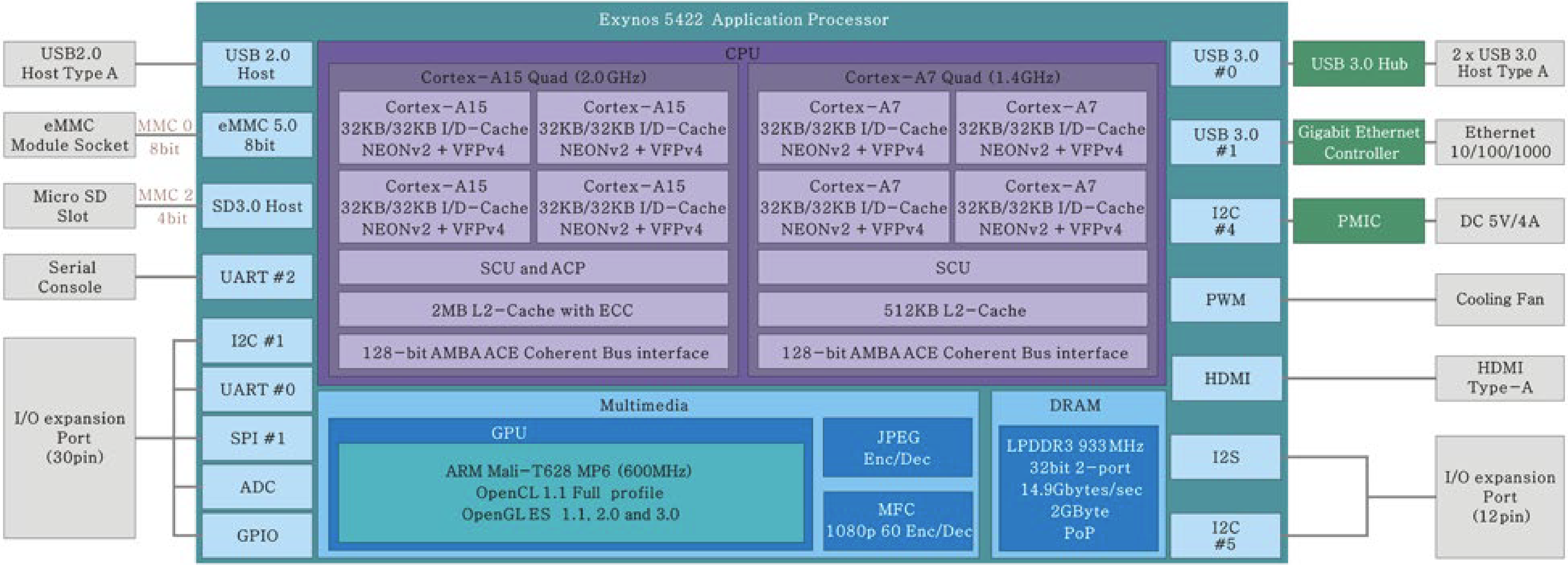
Stephen Byram Furber, “ARM System-on-Chip Architecture”

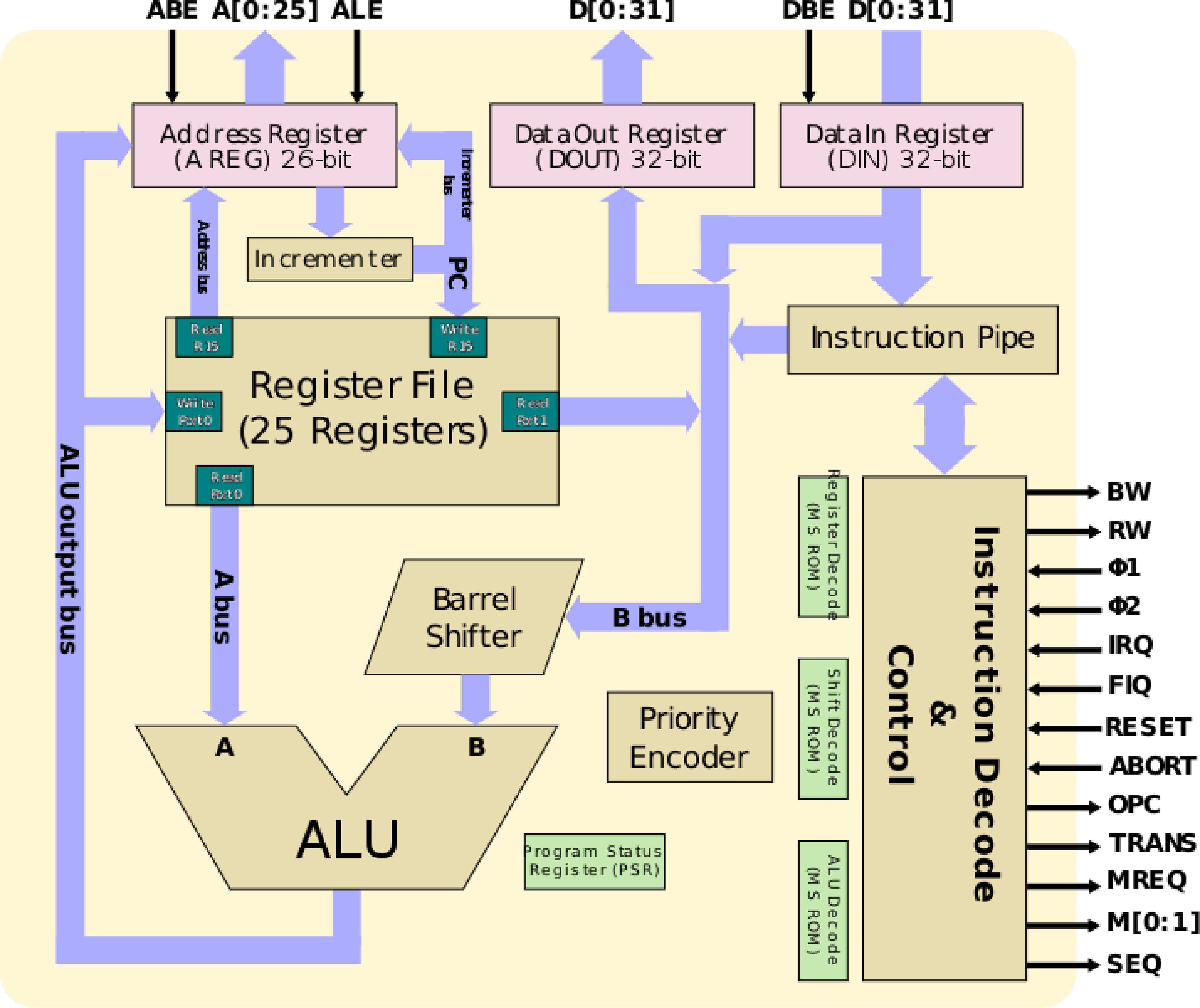
Повечето съвременни микропроцесори (включително и „ARM“) имат висока степен на ортогоналност на системата машинни команди. Но при x86 не е така. Например при 8086 от общо 96 команди ортогонални са само 36. Това се дължи на произхода на този микропроцесор от семейството 8008/8080/8085 с регистър-акумулатор (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).

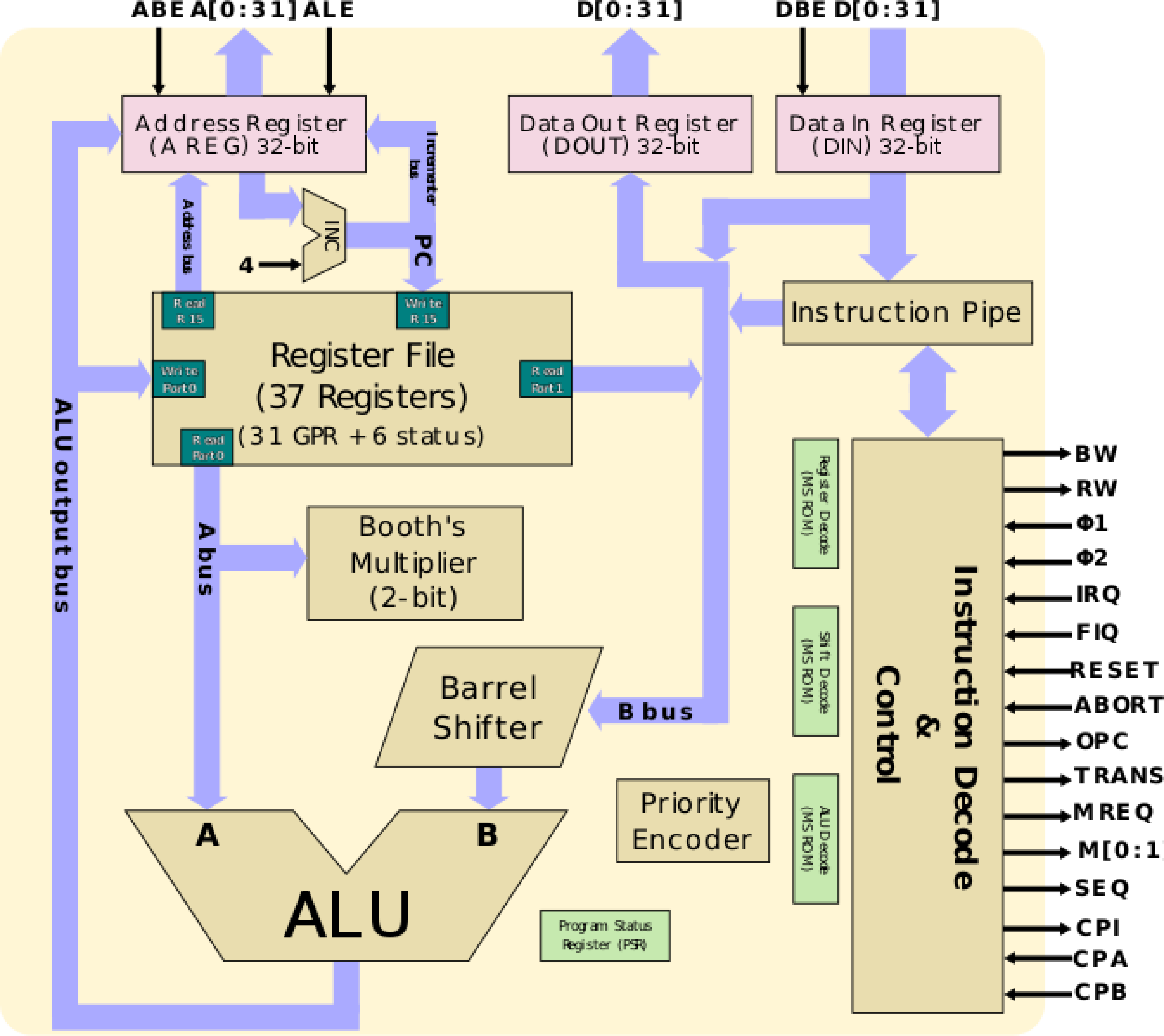

— SBIC —
SWITCHING POWER SUPPLY
CAUTION! HAZARDOUS AREA
ATTENTION! ZONE DANGEREUSE
SB-150W

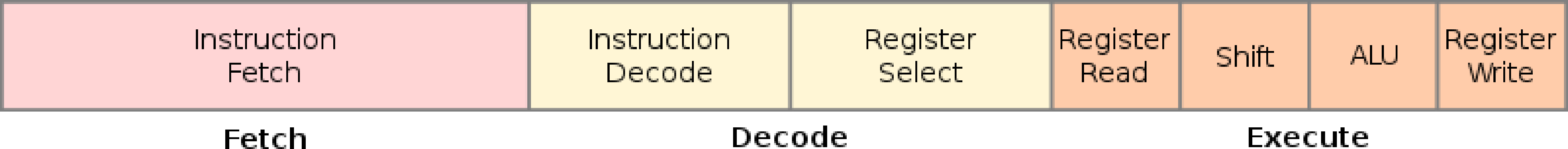
Структура на МП: Основни функционални блокове в МП. Вътрешни шини. Работа на конвейера.

ODROID-XU4 BLOCK DIAGRAM

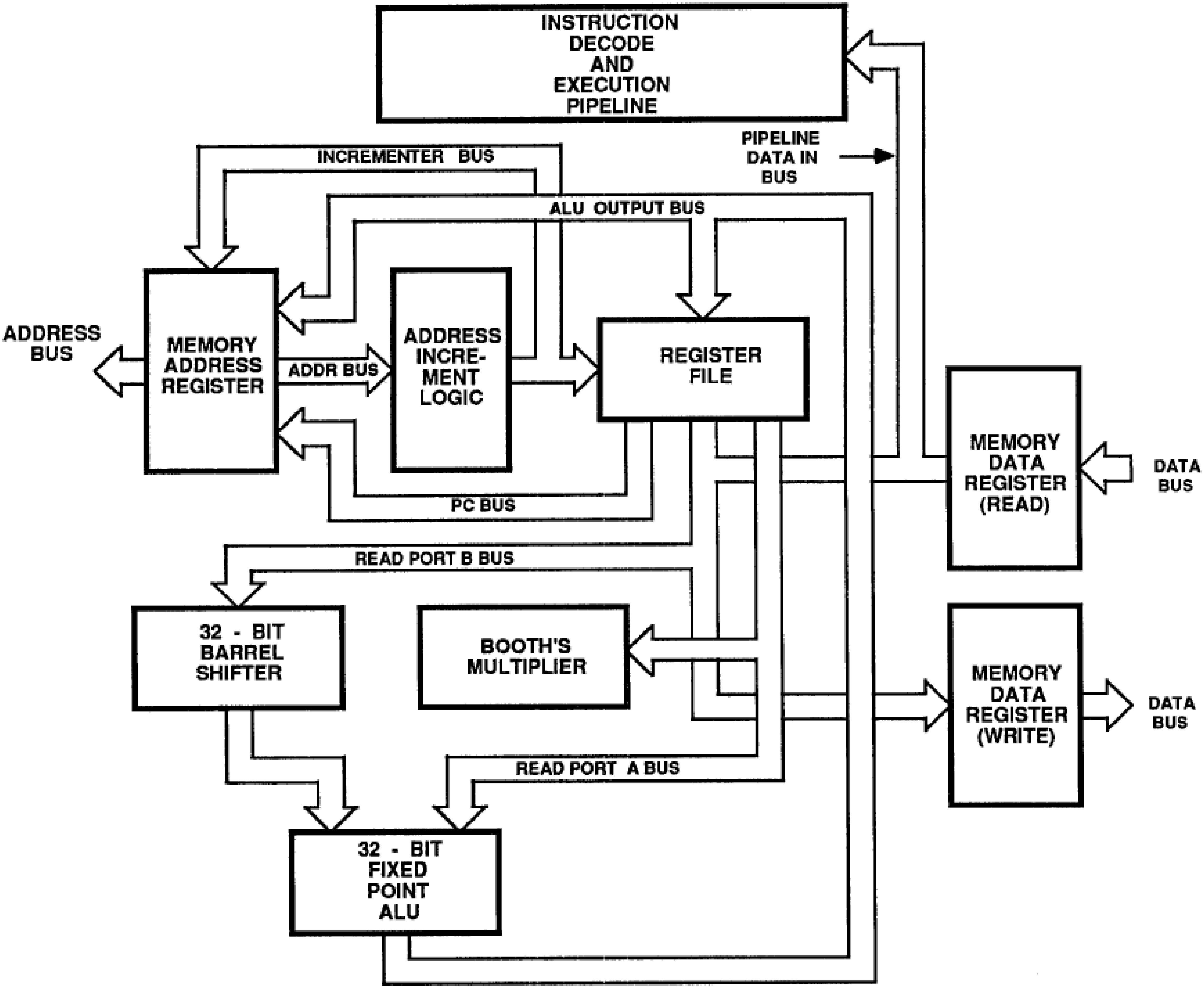


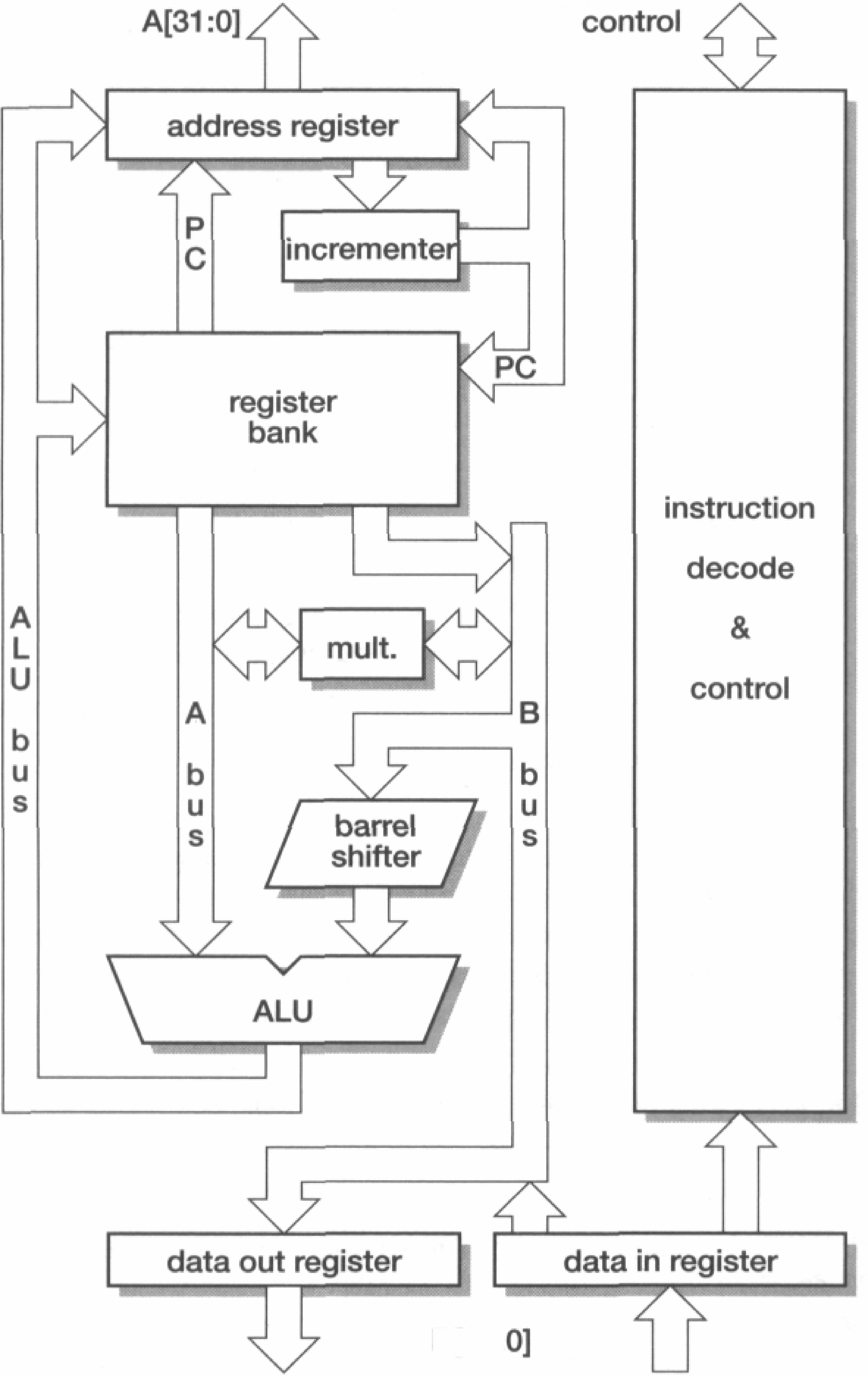

ABE
A[0:25]
ALE
D[0:31]
DBE
D[0:31]
Address Register (A REG) 26-bit
DataOut Register (DOUT) 32-bit
DataIn Register (DIN) 32-bit
Incrementer bus
Address bus
Incrementer
PC
Instruction Pipe
Read R15
Write R15
Register File (25 Registers)
Write Port0
Read Port1
Read Port0
ALU output bus
A bus
Barrel Shifter
B bus
Register Decode (M S ROM)
Shift Decode (M S ROM)
ALU Decode (M S 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
Incrementer bus
Data Out Register
(DOUT) 32-bit
Data In Register
(DIN) 32-bit
Address bus
INC
4
PC
Read R15
Write R15
Instruction Pipe
Write Port 0
Register File
(37 Registers)
(31 GPR + 6 status)
Read Port 1
Read Port 0
ALU output bus
A bus
Booth's Multiplier
(2-bit)
Register Decode (MS ROM)
Shift Decode (MS ROM)
ALU Decode (MS ROM)
Instruction Decode & Control
Barrel Shifter
B bus
Priority Encoder
A
B
ALU
Program Status Register (PSR)
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 bus
A bus
mult.
B bus
barrel shifter
ALU
data out register
data in register
0]

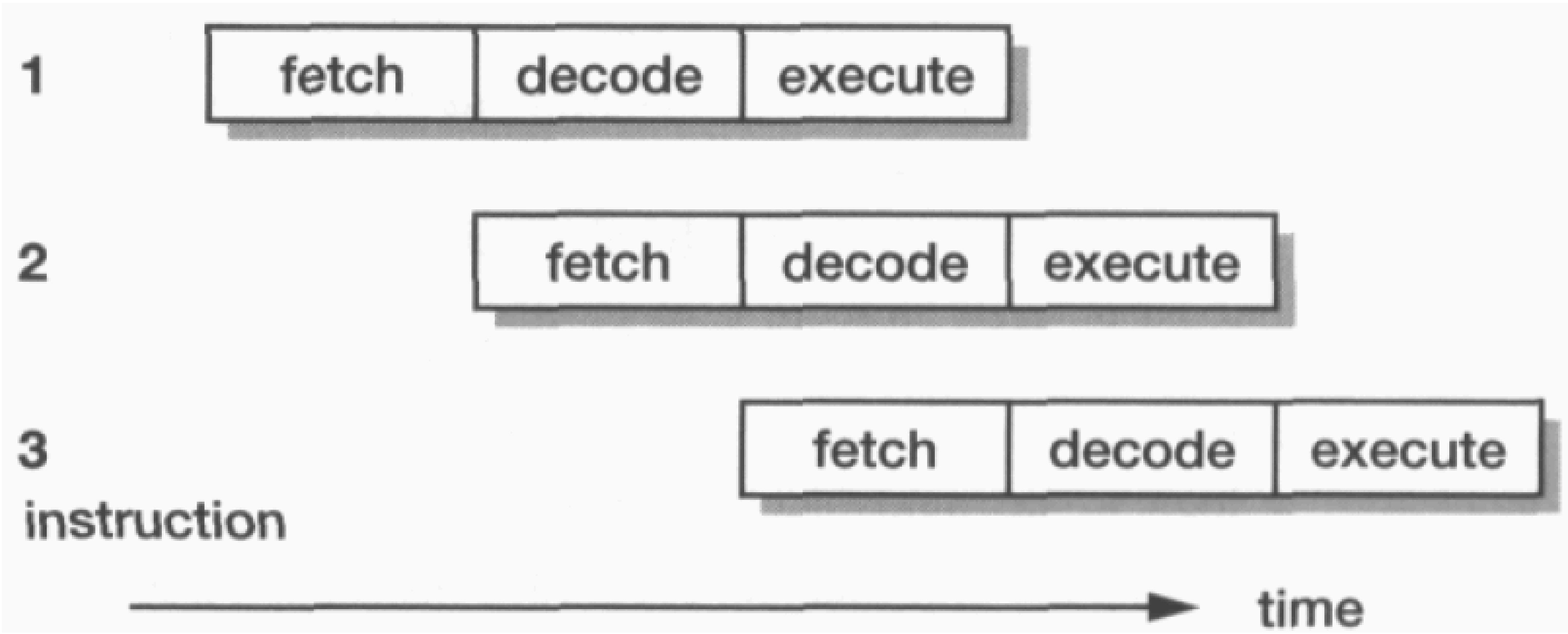


Figure 4.2 ARM single-cycle instruction 3-stage pipeline operation.

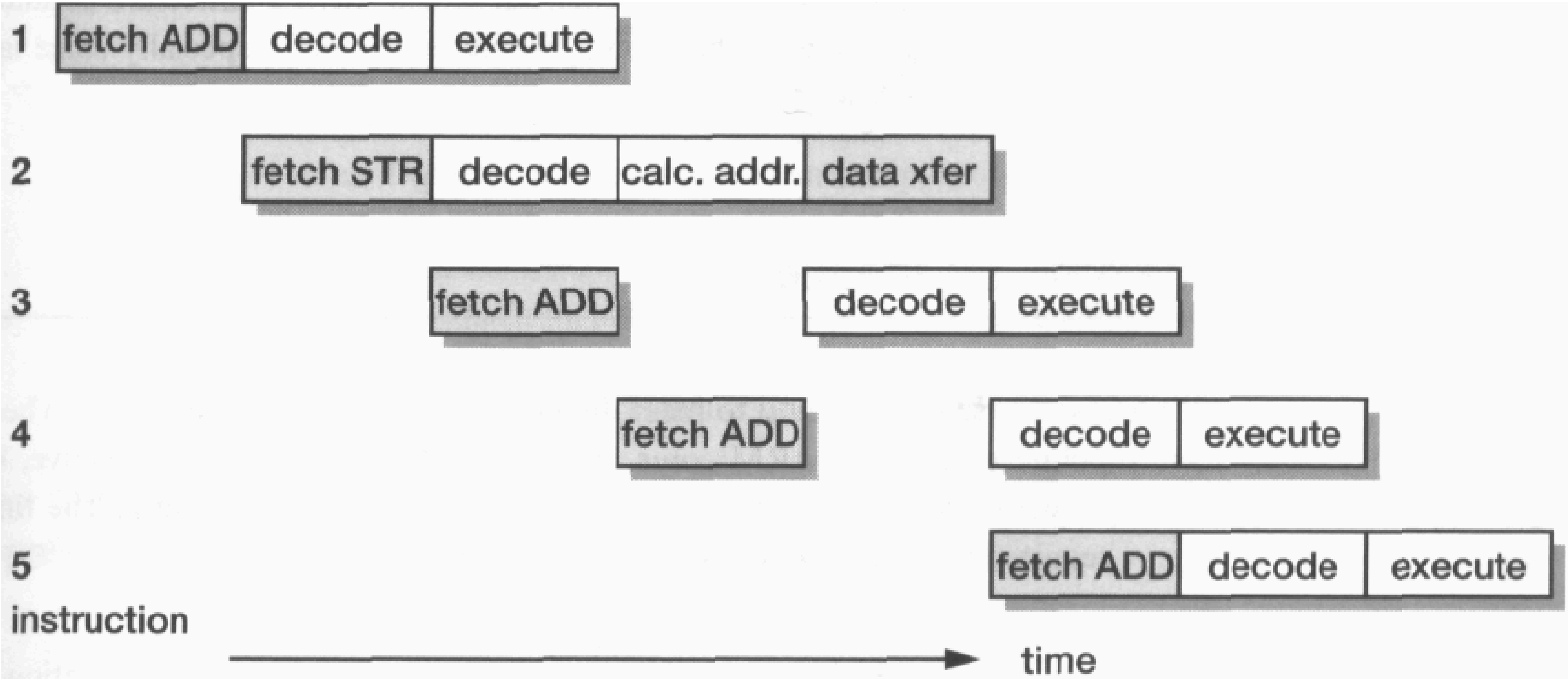


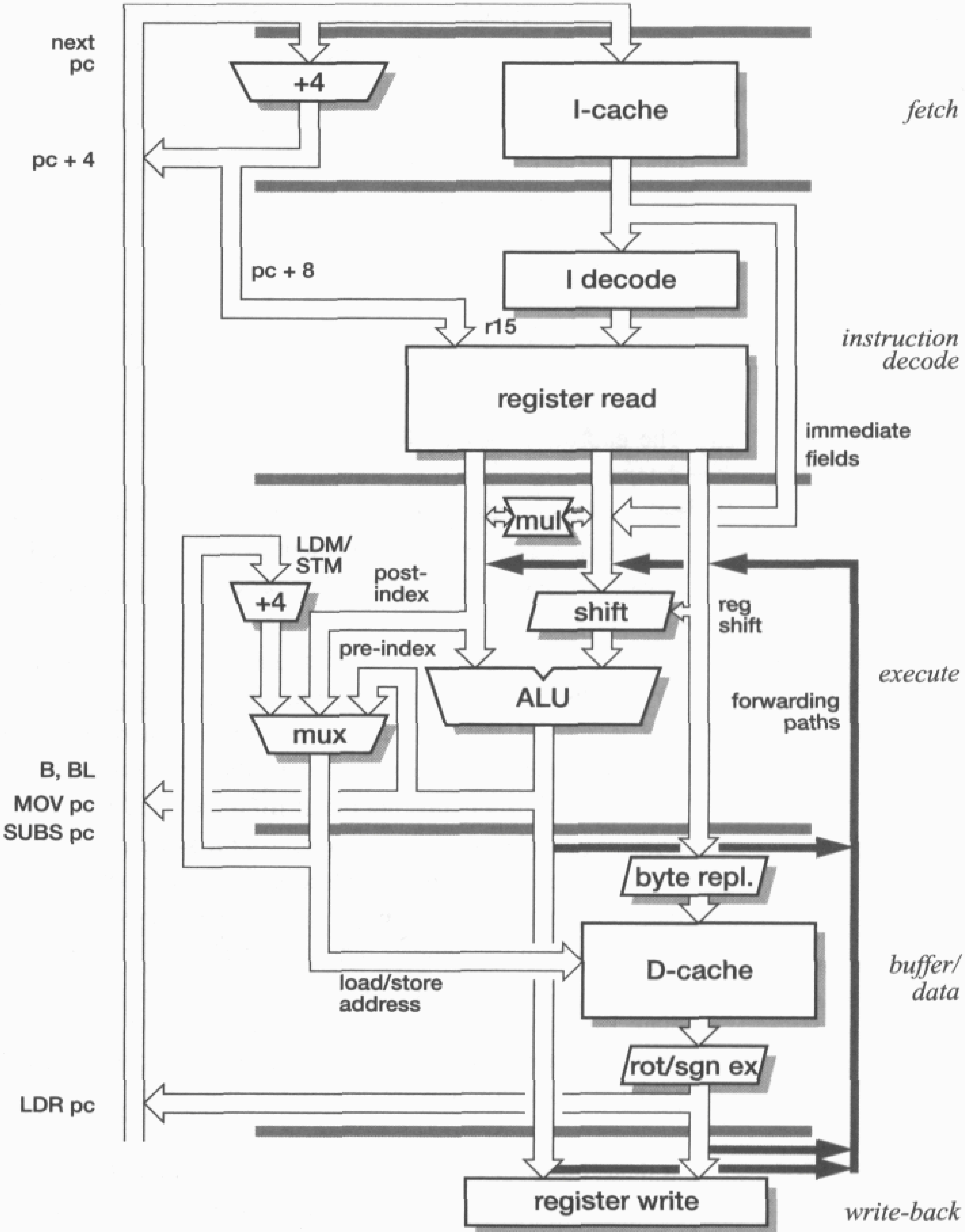
Figure 4.3 ARM multi-cycle instruction 3-stage pipeline operation.

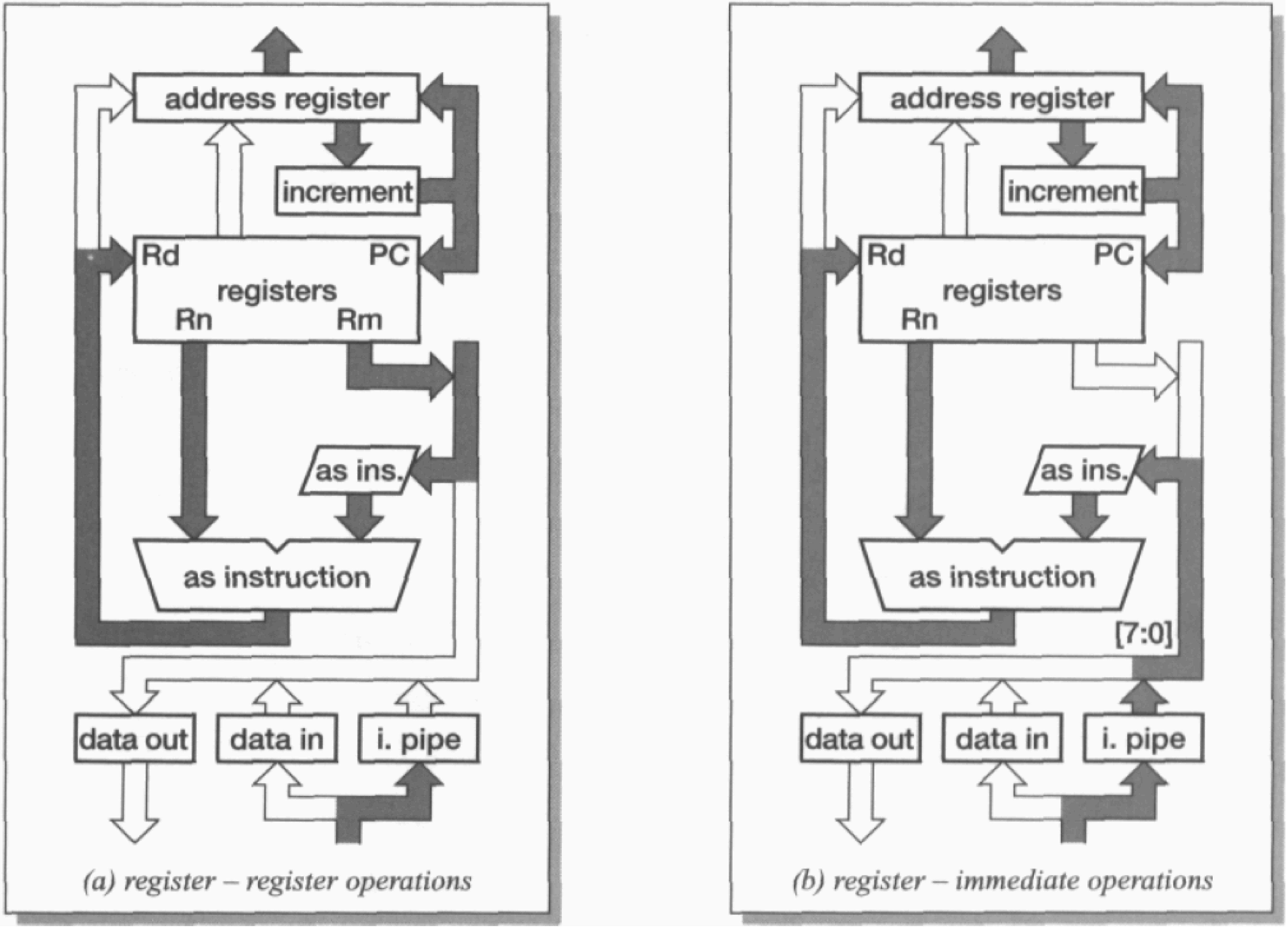
PC behaviour

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.

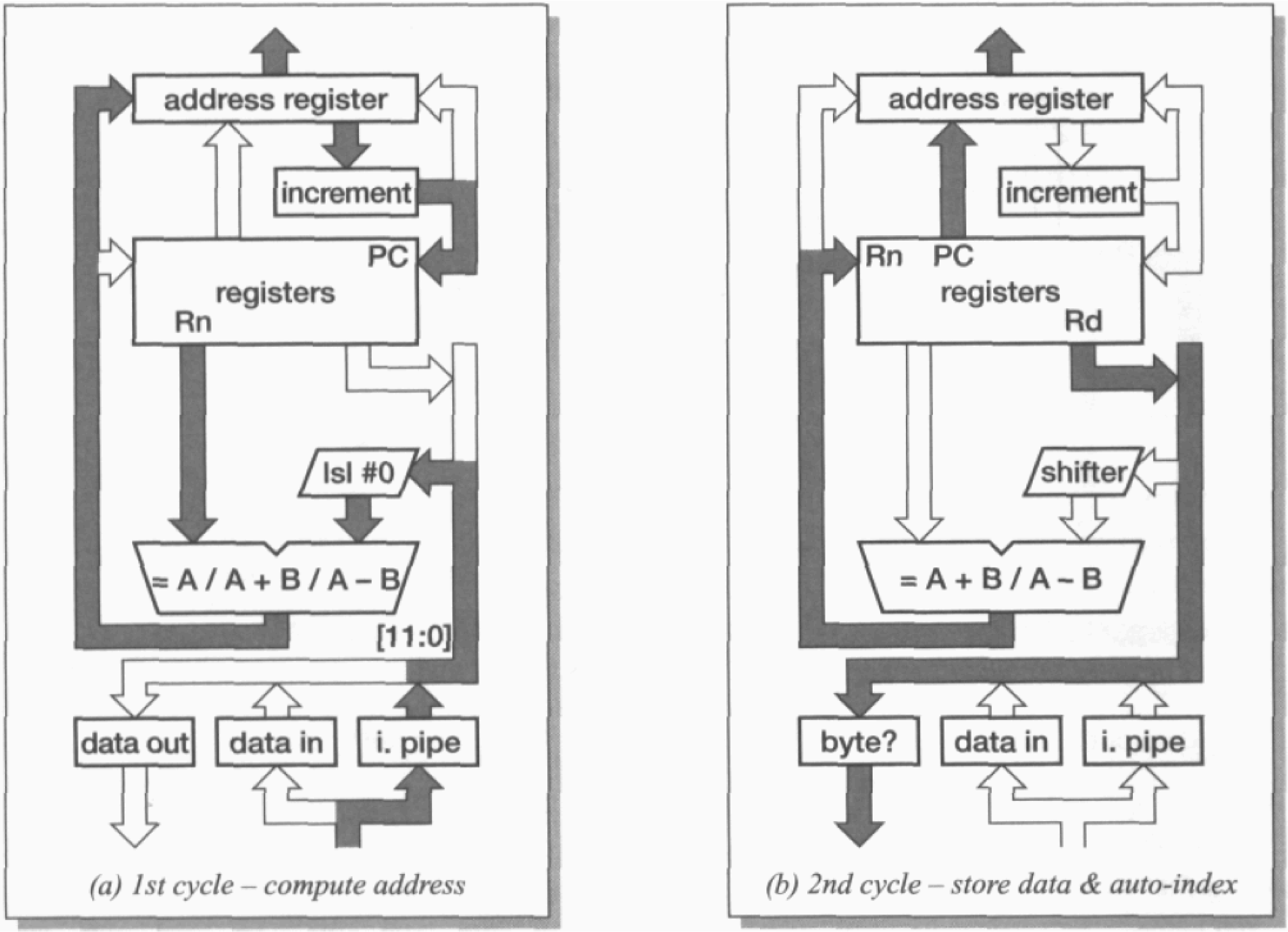

next pc
+4
I-cache
fetch
pc + 4
pc + 8
I decode
r15
instruction decode
register read
immediate fields
mul
LDM/STM
post-index
+4
shift
reg shift
pre-index
ALU
execute
forwarding paths
mux
B, BL
MOV pc
SUBS pc
byte repl.
D-cache
buffer/data
load/store address
rot/sgn ex
LDR pc
register write
write-back



(a) register – register operations

(b) register – immediate operations

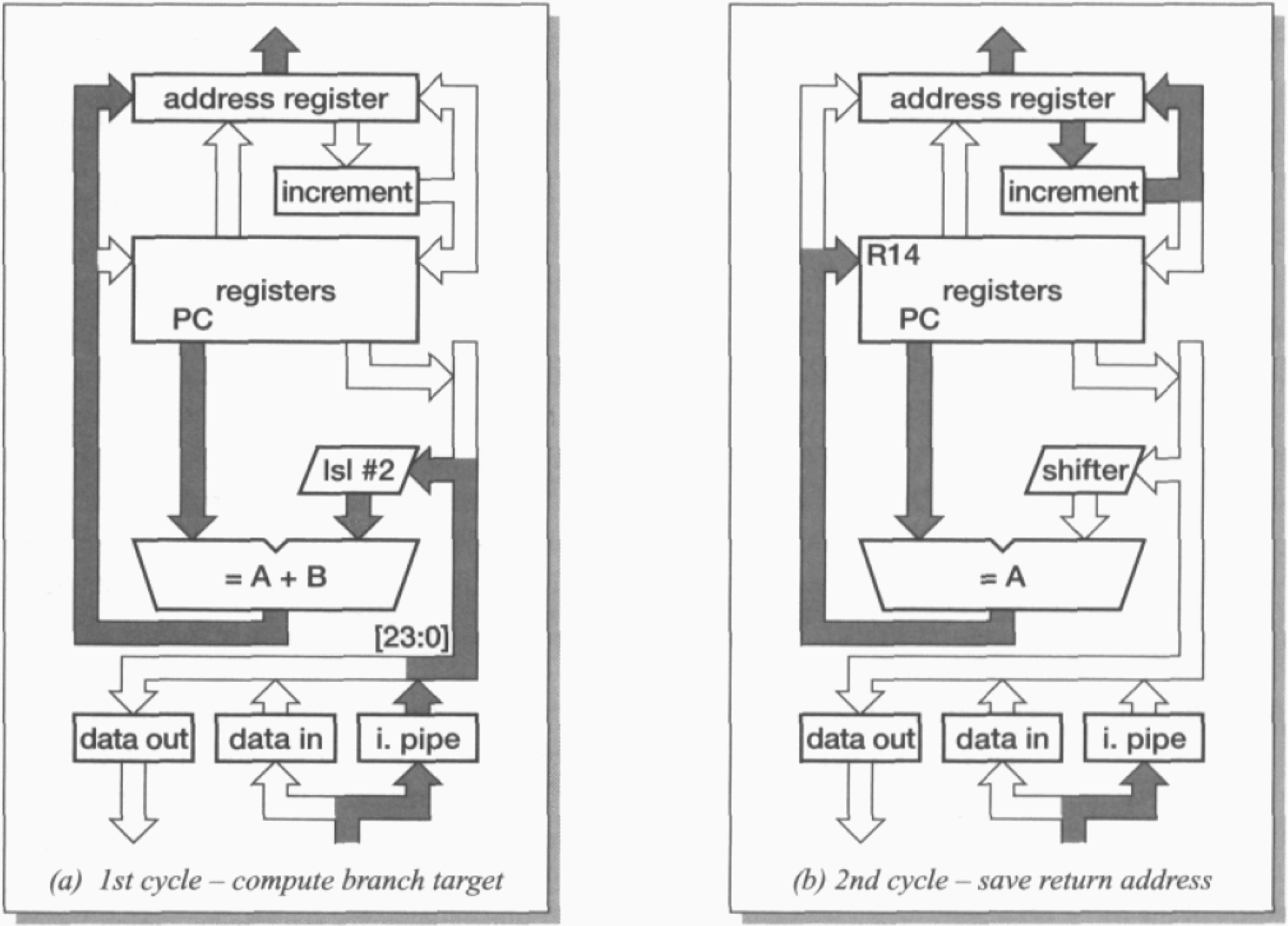
Figure 4.5 Data processing instruction datapath activity.



(a) 1st cycle – compute address

(b) 2nd cycle – store data & auto-index

Figure 4.6 SIR (store register) datapath activity.



(a) 1st cycle – compute branch target

(b) 2nd cycle – save return address

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.

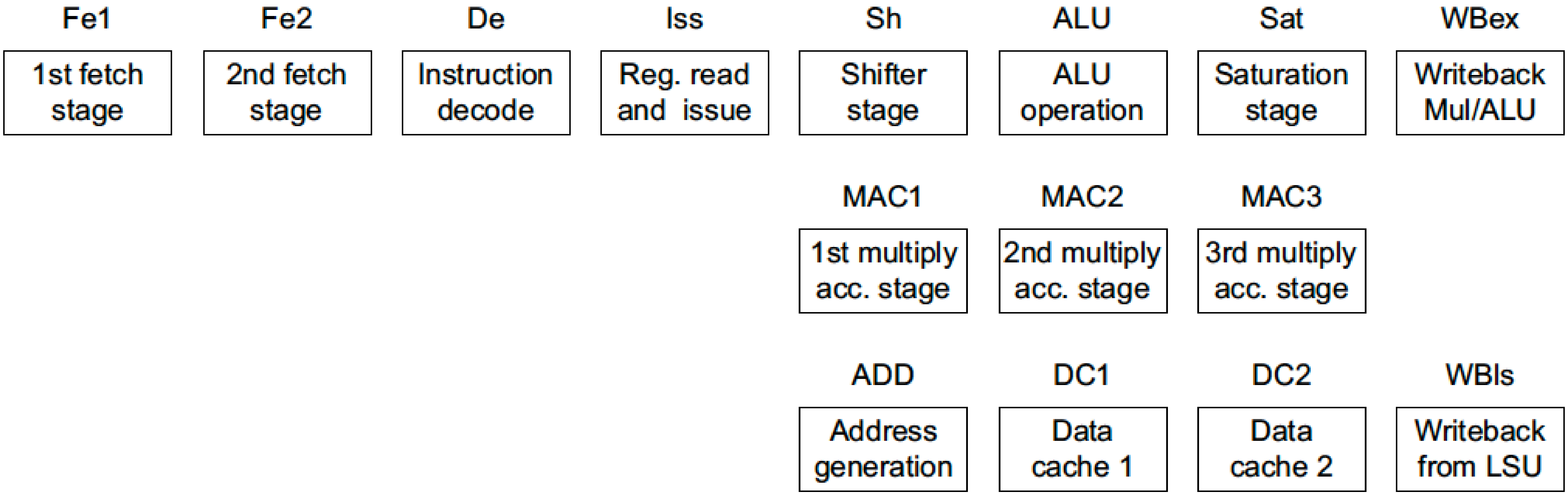


Figure 1-2 ARM1176JZF-S pipeline stages

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.

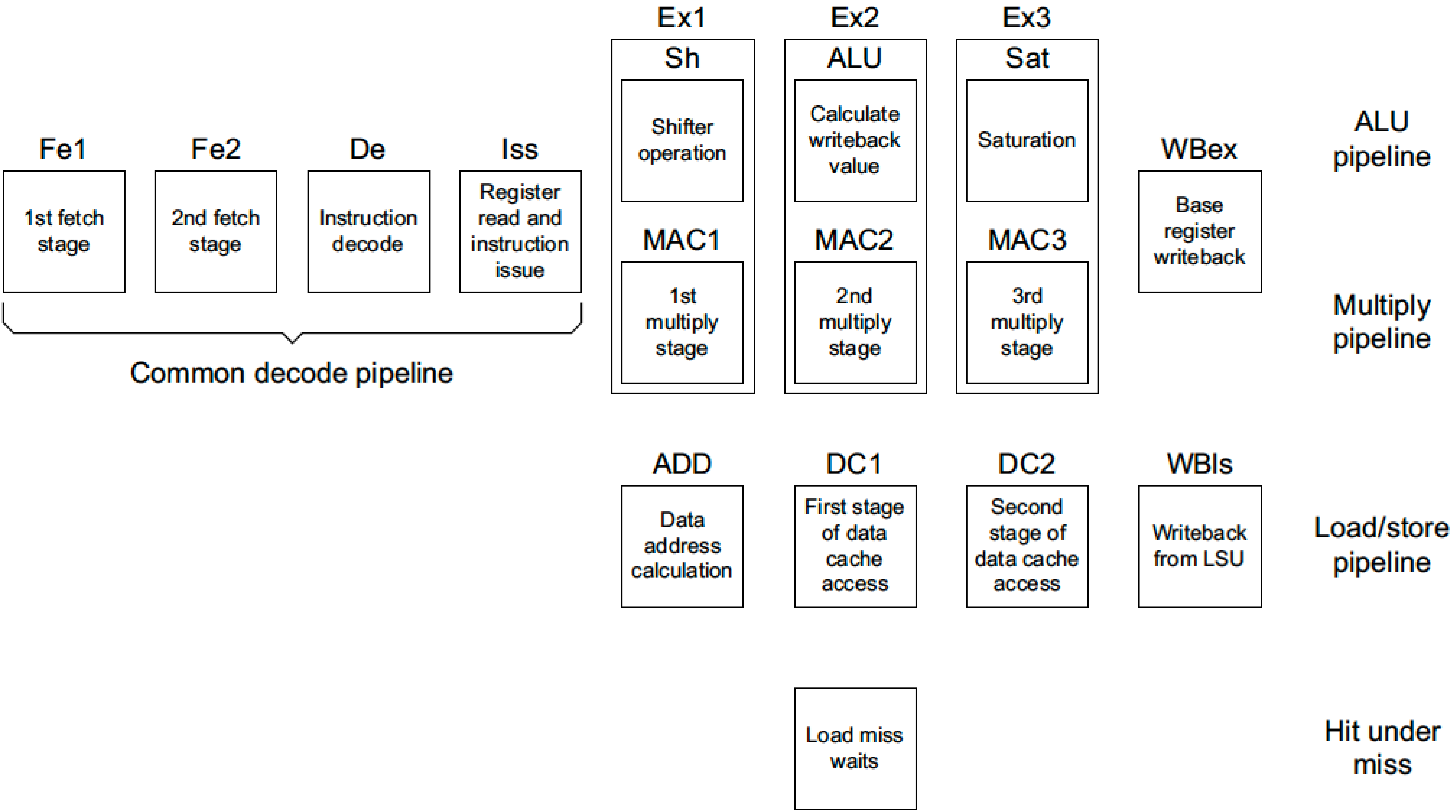


Figure 1-3 Typical operations in pipeline stages

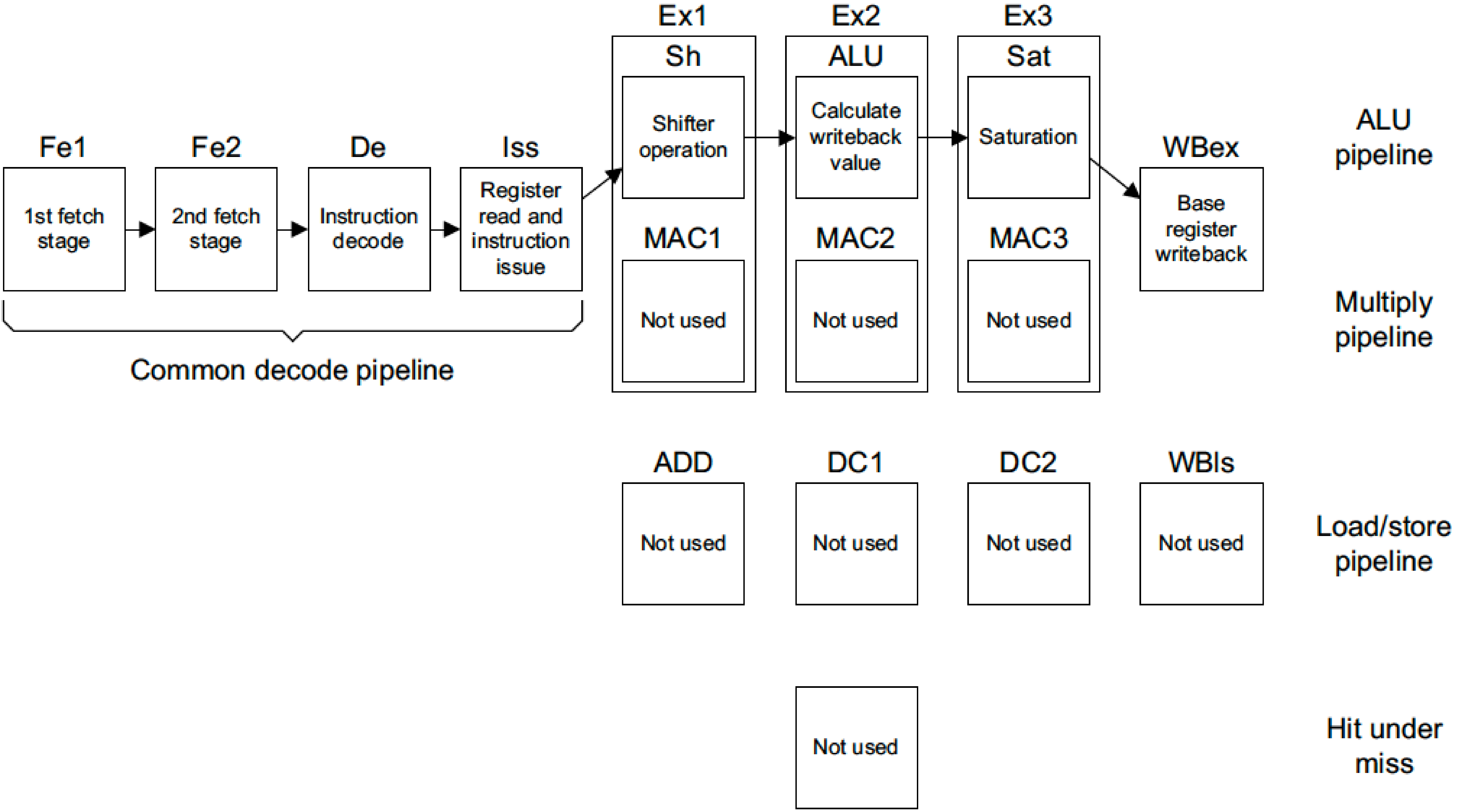


Figure 1-4 Typical ALU operation

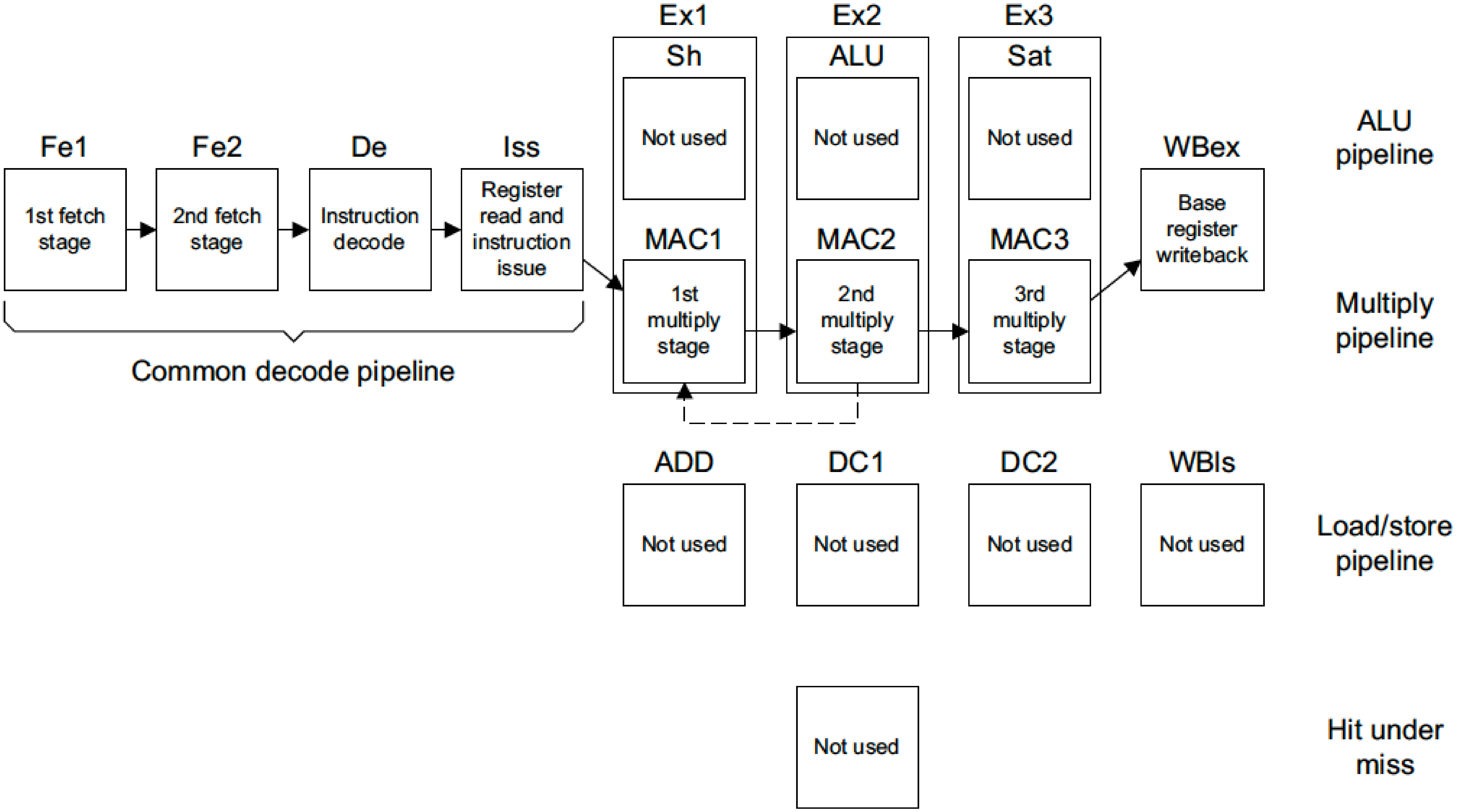


Figure 1-5 Typical multiply operation

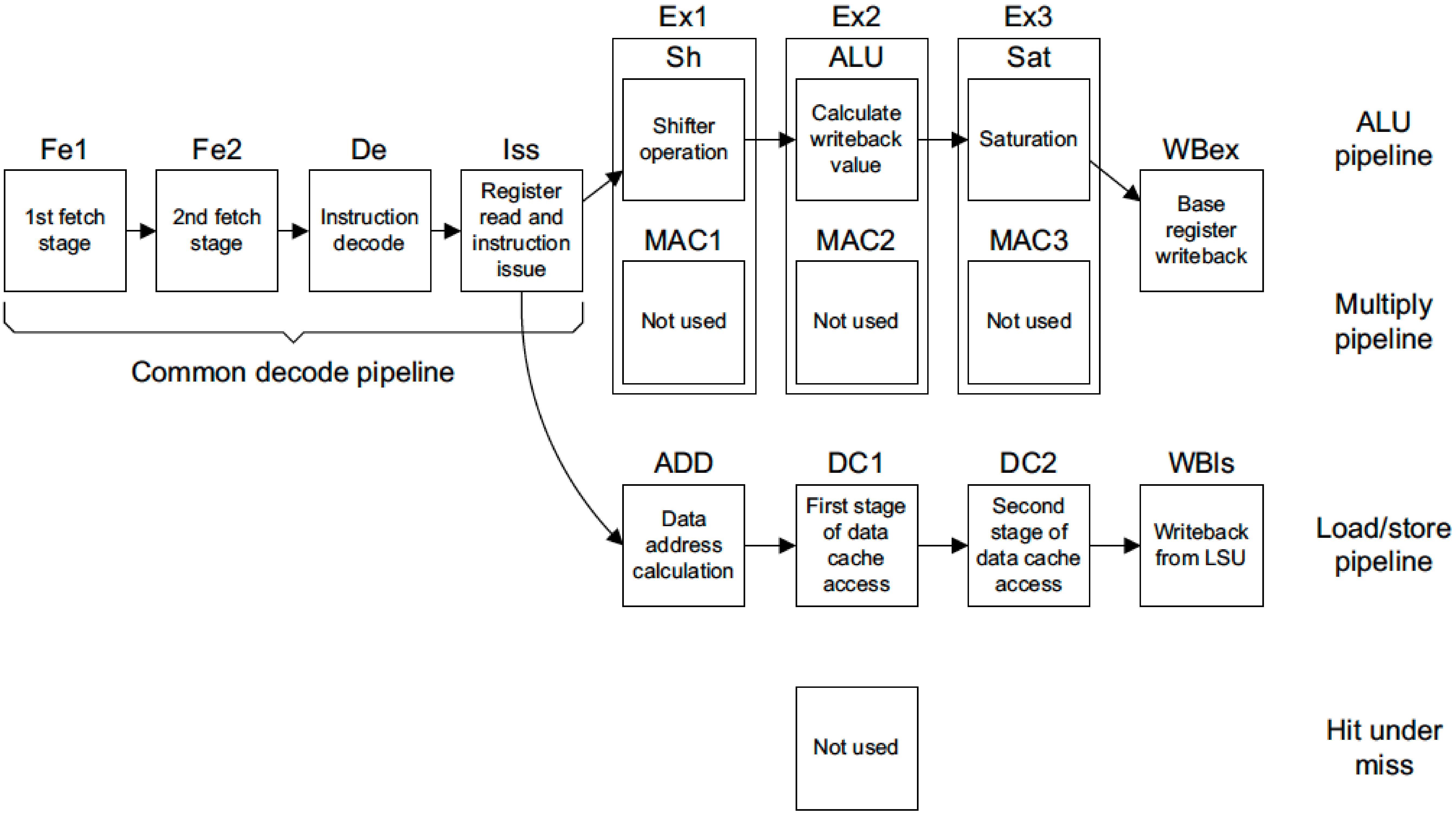


Figure 1-6 Progression of an LDR/STR operation

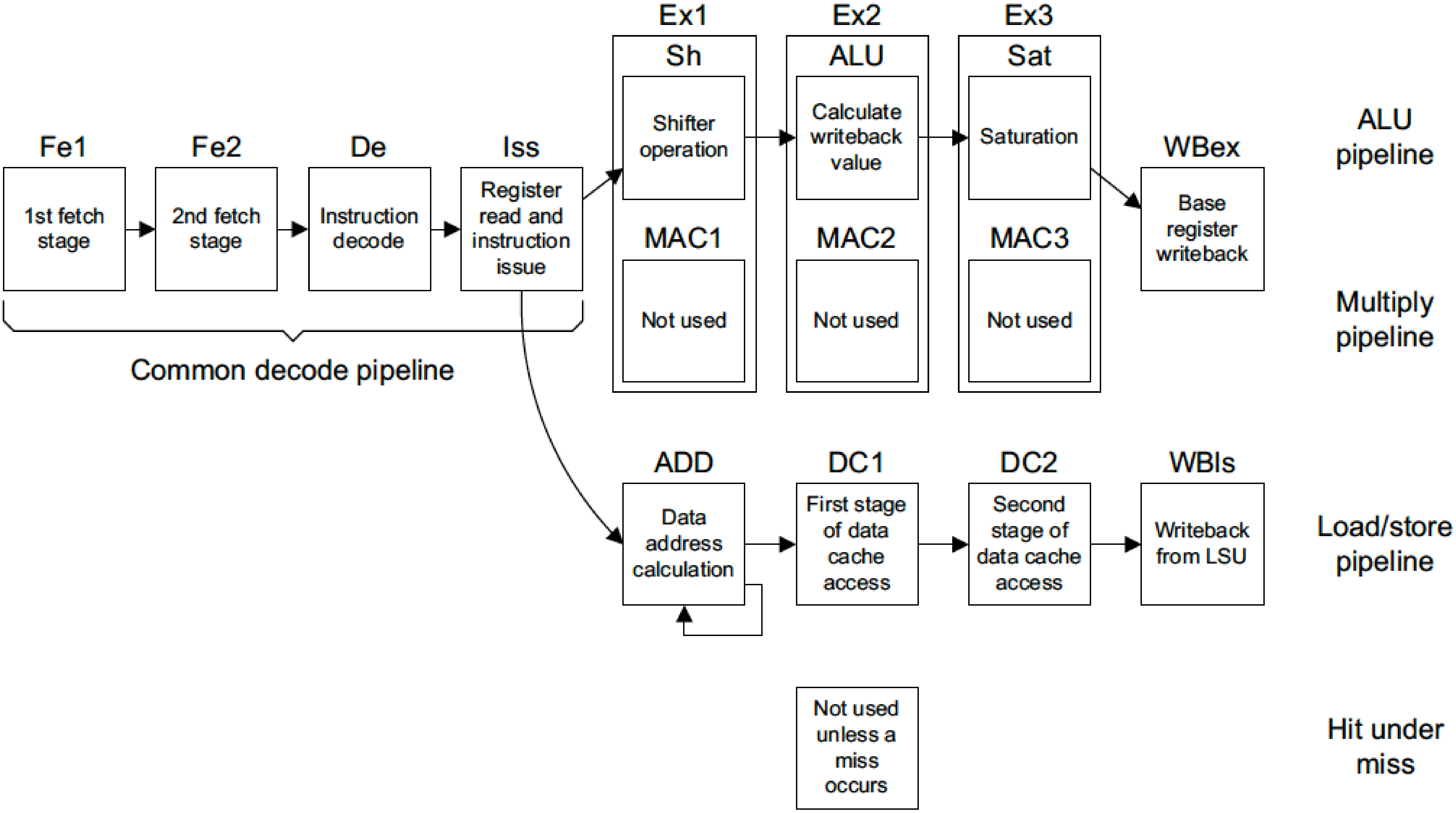


Figure 1-7 Progression of an LDM/STM operation

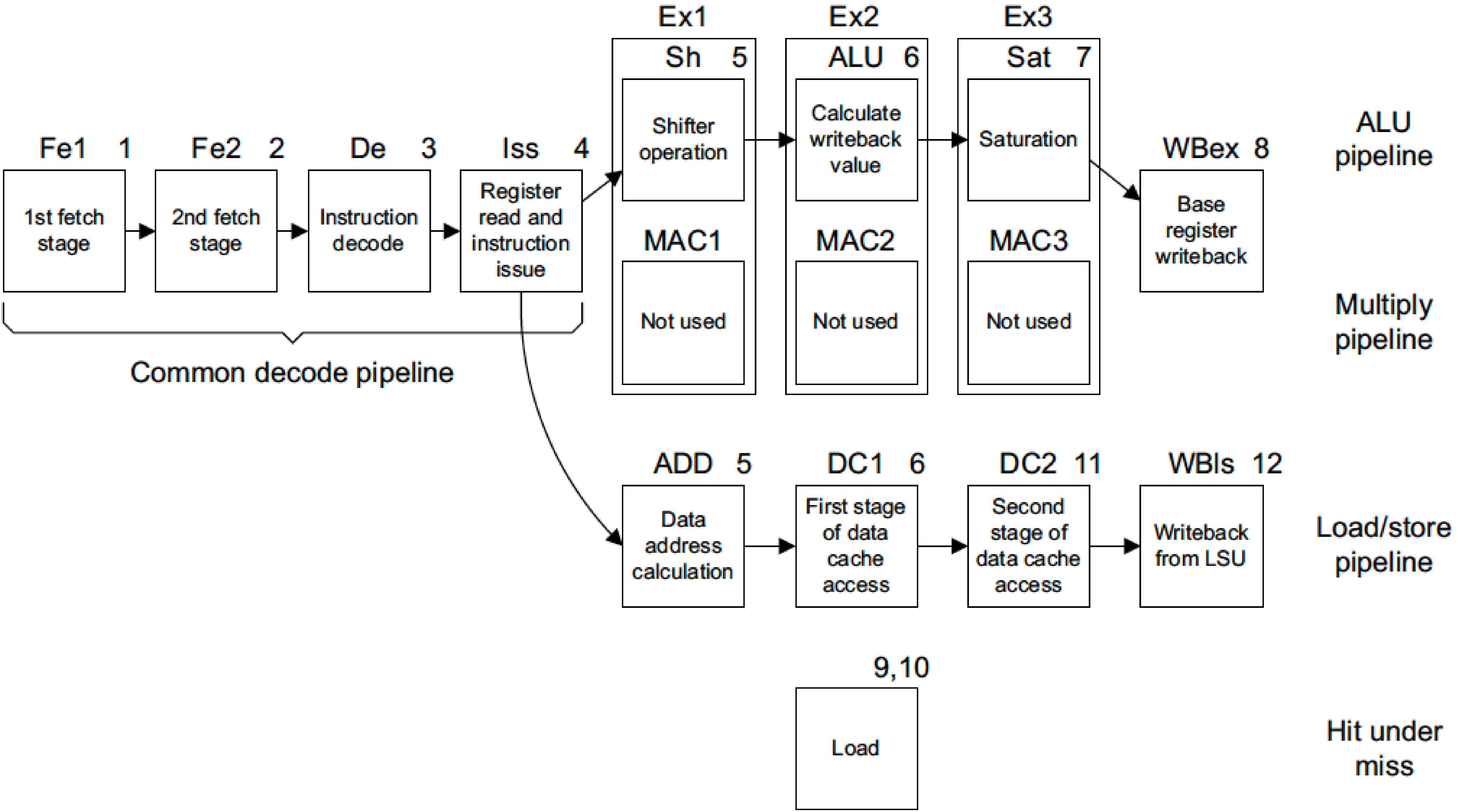


Figure 1-8 Progression of an LDR that misses

<u>Системна магистрала:</u> Сигнали на шините за адреси и данни. Управляващи сигнали. Организация на обмена на данни. Видове цикли. Времедиаграми.

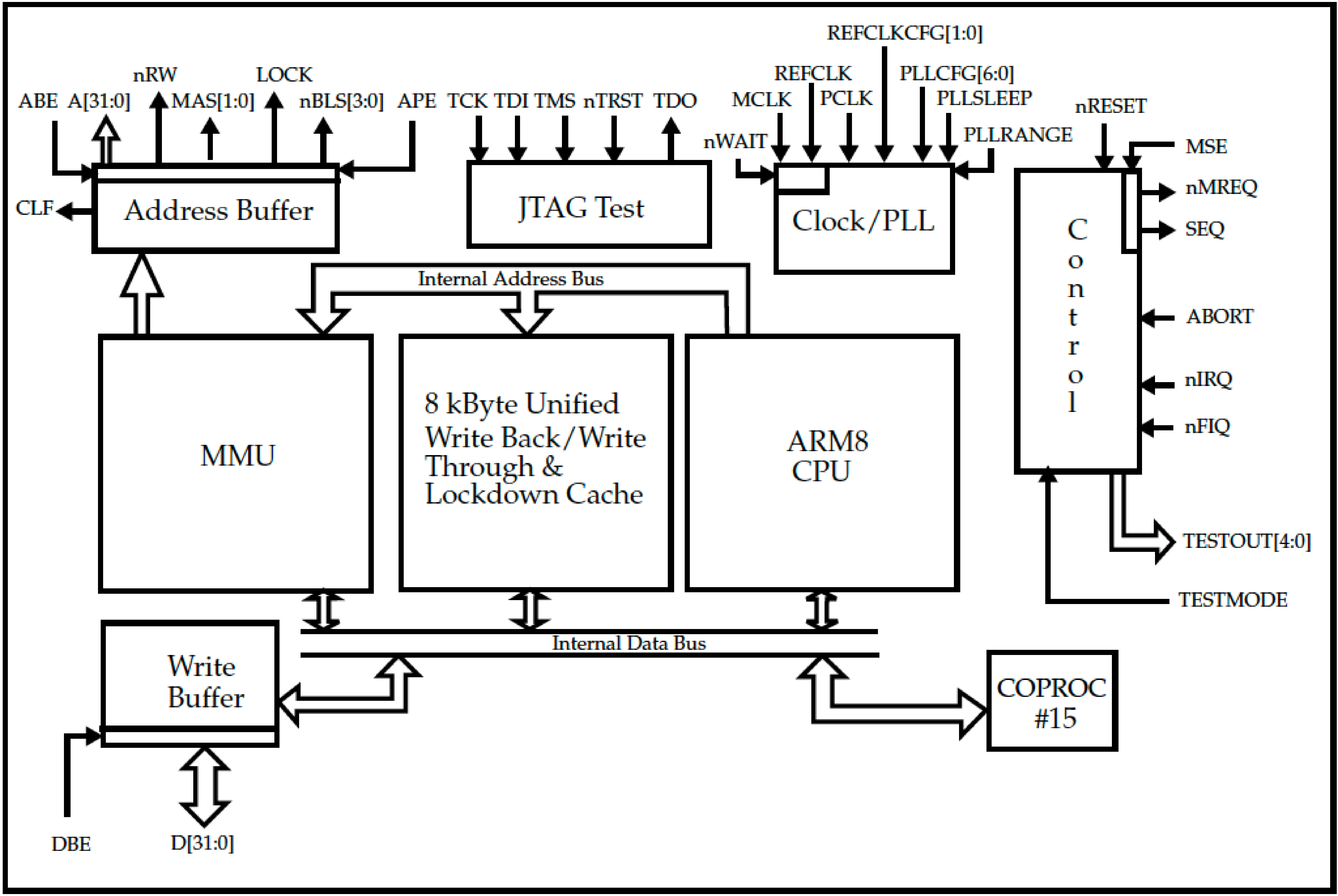


Figure 1-1: ARM810 block diagram

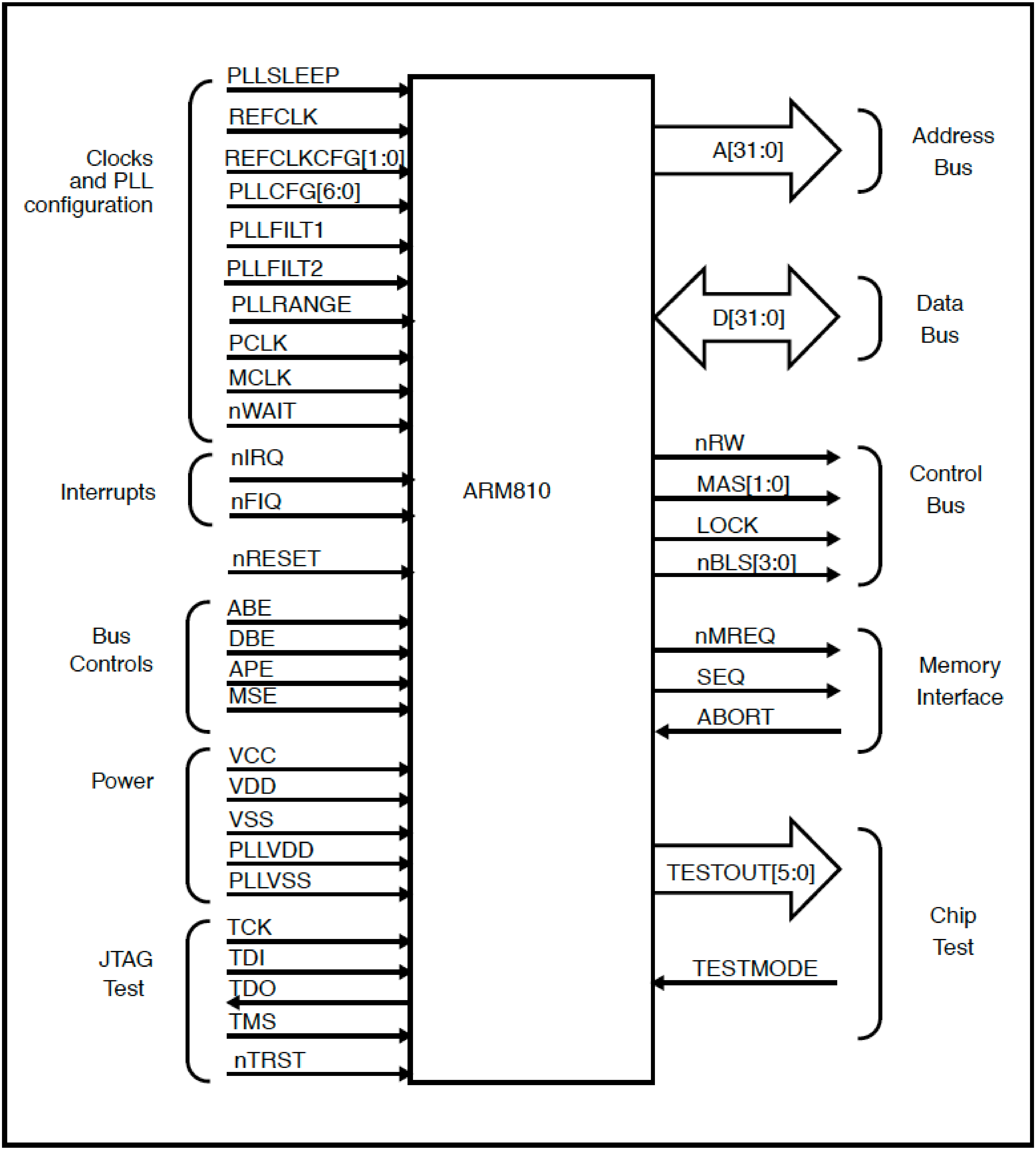


Figure 2-1: ARM810 functional diagram

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.

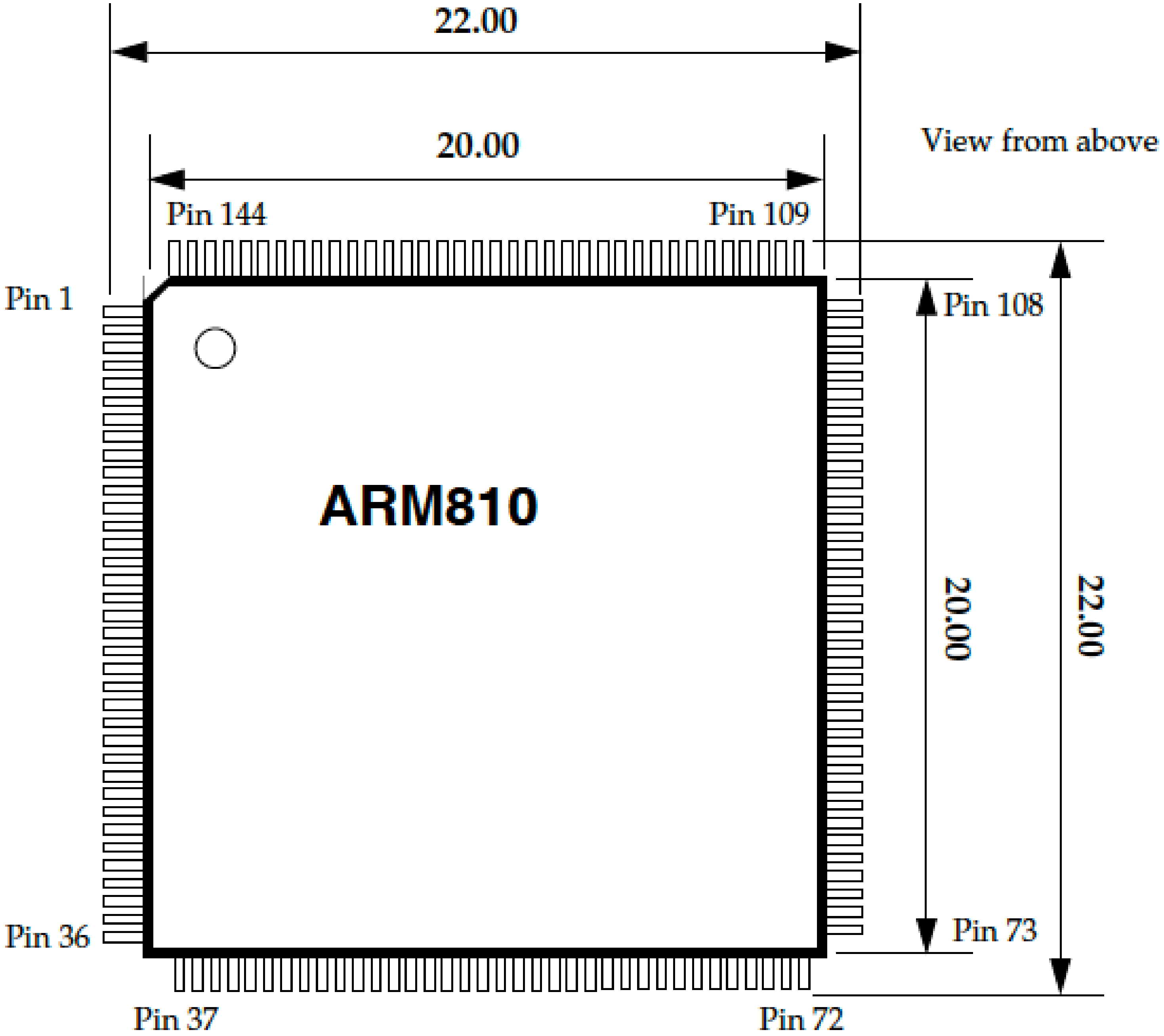
MSE	I	Memory request/sequential enable. When this input is LOW, the **nMREQ** and **SEQ** outputs are put into a high impedance state (Note 1).
MAS[1:0]	OCZ	Memory Access Size. An output bus used by the processor to indicate the size of the next data transfer to the external memory system as being a 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.

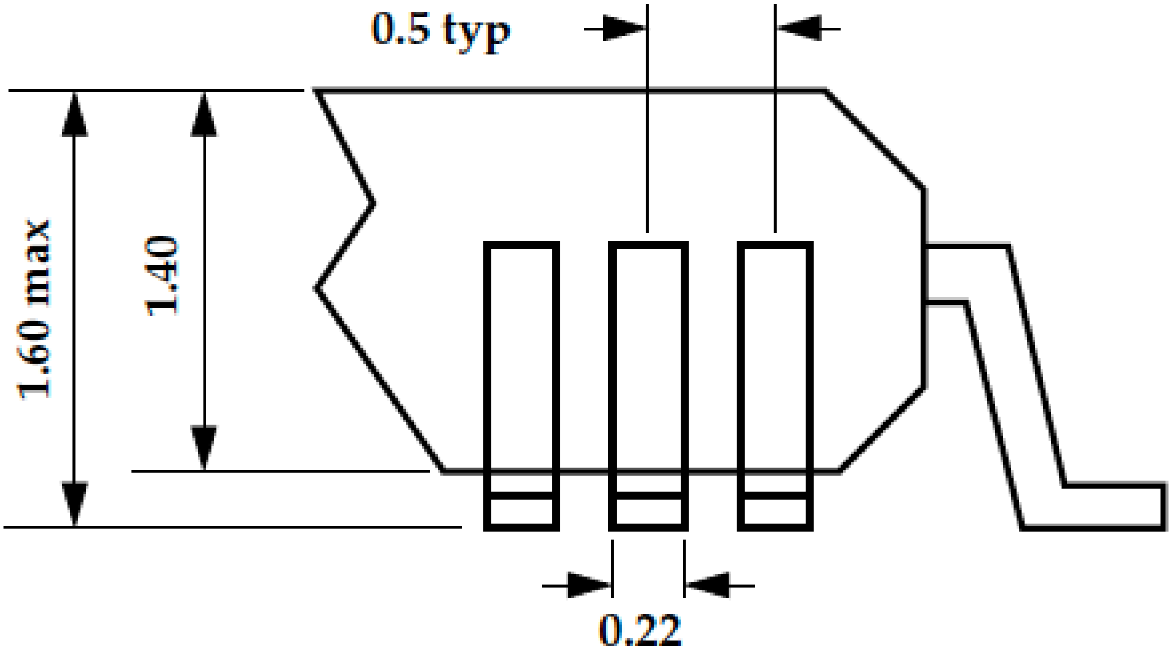
nMREQ	OCZ	Not memory request. A pipelined signal that changes while **MCLK** is LOW to indicate whether or not in the following cycle, the processor will be accessing external memory. When **nMREQ** is LOW, the processor will be accessing external memory in the next bus cycle.
nRESET	I	Not reset. This 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.


22.00
20.00
View from above
Pin 144
Pin 109
Pin 1
Pin 108
ARM810
20.00
22.00
Pin 36
Pin 73
Pin 37
Pin 72


0.5 typ
1.60 max
1.40
0.22

Pin	Signal
1	MSE
2	SEQ
3	NMREQ
4	REFCLKCFG[0]
5	REFCLKCFG[1]
6	Vdd_core
7	PLLSLEEP
8	Vss_core
9	PLLRANGE
10	PLLVDD
11	PLLFILT2
12	PLLFILT1
13	PLLGND
14	NWAIT
15	REFCLK
16	Vdd_pad
17	PCLK
18	MCLK
19	Vss_pad
20	DBE
21	D[0]
22	D[1]
23	D[2]
24	D[3]
25	D[4]
26	D[5]
27	D[6]
28	Vdd_pad
29	D[7]
30	Vss_pad
31	D[8]
32	D[9]
33	D[10]
34	D[11]
35	D[12]
36	D[13]
37	D[14]
38	D[15]
39	D[16]
40	Vdd_pad
41	D[17]
42	Vss_pad
43	D[18]
44	D[19]
45	Vdd_core
46	D[20]
47	Vss_core
48	D[21]
49	D[22]
50	D[23]
51	D[24]
52	Vdd_pad
53	D[25]
54	Vss_pad
55	D[26]
56	D[27]
57	D[28]
58	D[29]
59	D[30]
60	D[31]
61	TDO
62	TCK
63	TMS
64	nTRST
65	TDI
66	Vdd_pad
67	NBLS[0]
68	Vss_pad
69	NBLS[1]
70	NBLS[2]
71	NBLS[3]
72	NRW
73	MAS[0]
74	MAS[1]
75	CLF
76	LOCK
77	A[0]
78	A[1]
79	Vdd_pad
80	A[2]
81	Vss_pad
82	A[3]
83	A[4]
84	Vdd_core
85	A[5]
86	Vss_core
87	A[6]

Pin	Signal
88	A[7]
89	A[8]
90	Vdd_pad
91	A[9]
92	Vss_pad
93	A[10]
94	A[11]
95	A[12]
96	Vdd_core
97	A[13]
98	Vss_core
99	A[14]
100	A[15]
101	A[16]
102	Vdd_pad
103	A[17]
104	Vss_pad
105	A[18]
106	A[19]
107	A[20]
108	A[21]
109	A[22]
110	A[23]
111	A[24]
112	A[25]
113	A[26]
114	Vdd_pad
115	A[27]
116	Vss_pad
117	A[28]
118	A[29]
119	A[30]
120	A[31]
121	ABE
122	APE
123	Vcc
124	TESTOUT[0]
125	TESTOUT[1]
126	Vdd_core
127	TESTOUT[2]
128	Vss_core
129	TESTOUT[3]
130	TESTOUT[4]
131	TESTMODE
132	NIRQ
133	Vdd_pad
134	NRESET
135	Vss_pad
136	NFIQ
137	ABORT
138	PLLCFG[0]
139	PLLCFG[1]
140	PLLCFG[2]
141	PLLCFG[3]
142	PLLCFG[4]
143	PLLCFG[5]
144	PLLCFG[6]

10.20 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 10-22: ARM instruction speed summary* on page 10-20. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

n	is the number of words transferred
m is	1 if bits [32:8] of the multiplier operand are all zero or one. 2 if bits[32:16] of the multiplier operand are all zero or one. 3if bits[31:24] of the multiplier operand are all zero or all one. 4 otherwise.
b	is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all the instructions take one S-cycle. The cycle types N, S, I, and C are defined in *Chapter 6, Memory Interface.*

Instruction	Cycle count	Additional
Data Processing	1S	+ 1I for SHIFT(Rs) + 1S + 1N if R15 written
MSR, MRS	1S	
LDR	1S+1N+1I	+ 1S + 1N if R15 loaded
STR	2N	
LDM	nS+1N+1I	+ 1S + 1N if R15 loaded
STM	(n-1)S+2N	
SWP	1S+2N+1I	
B,BL	2S+1N	
SWI, trap	2S+1N	
MUL	1S+mI	
MLA	1S+(m+1)I	
MULL	1S+(m+1)I	
MLAL	1S+(m+2)I	
CDP	1S+bI	
LDC,STC	(n-1)S+2N+bI	
MCR	1N+bI+1C	
MRC	1S+(b+1)I+1C	

Table 10-22: ARM instruction speed summary

6.2 Cycle Types

All memory transfer cycles can be placed in one of four categories:

1. Non-sequential cycle. ARM7TDMI requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.
2. Sequential cycle. ARM7TDMI requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word or halfword after the preceding address.
3. Internal cycle. ARM7TDMI does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.
4. Coprocessor register transfer. ARM7TDMI wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

These four classes are distinguishable to the memory system by inspection of the **nMREQ** and **SEQ** control lines (see ➲*Table 6-1: Memory cycle types*). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

nMREQ	SEQ	Cycle type
0	0	Non-sequential (N-cycle)
0	1	Sequential (S-cycle)
1	0	Internal (I-cycle)
1	1	Coprocessor register transfer (C-cycle)

Table 6-1: Memory cycle types

Bus Interface Signals

The signals in the Bus interface can be grouped into 3 categories:

Addressing signals:

- **A[31:0]**
- **nRW**
- **MAS[1:0]**
- **LOCK**
- **nBLS[3:0]]**
- **CLF**

Memory Request signals:

- **nMREQ**
- **SEQ**

Data sampled signals:

- **D[31:0]**

Abort signal:

- **ABORT**

Each of these groups shares a common timing relationship to the bus interface cycles. The ARM bus interface addressing signals and memory request signals are pipelined ahead of the data. **nMREQ** and **SEQ** are pipelined by a whole bus cycle, and the address timed signals by 1/2 a cycle. The timing of the address timed signal can be altered by the **APE** pin.

Note *Unless otherwise specified, all diagrams in this chapter show the ARM810 operating with the* **APE** *pin held HIGH.*

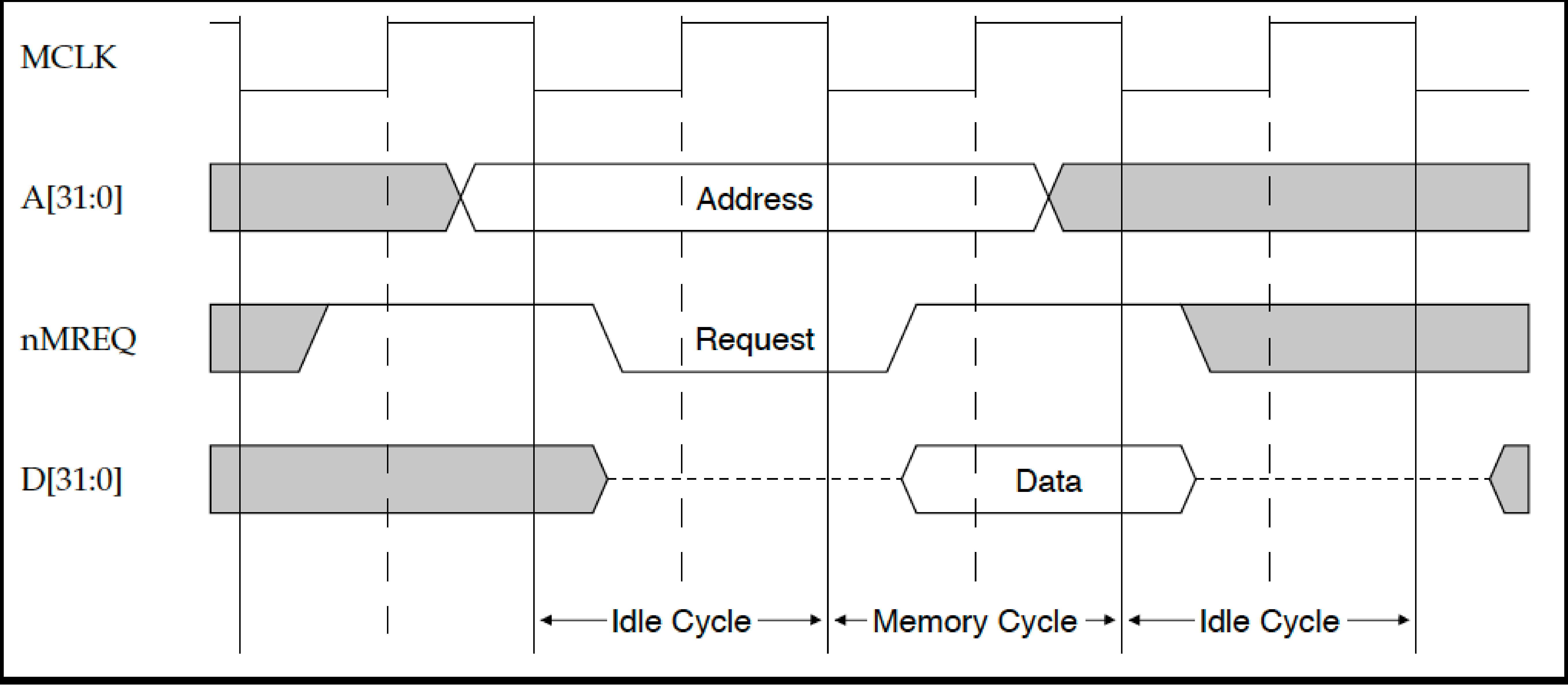


Figure 12-1: Simplified single cycle access

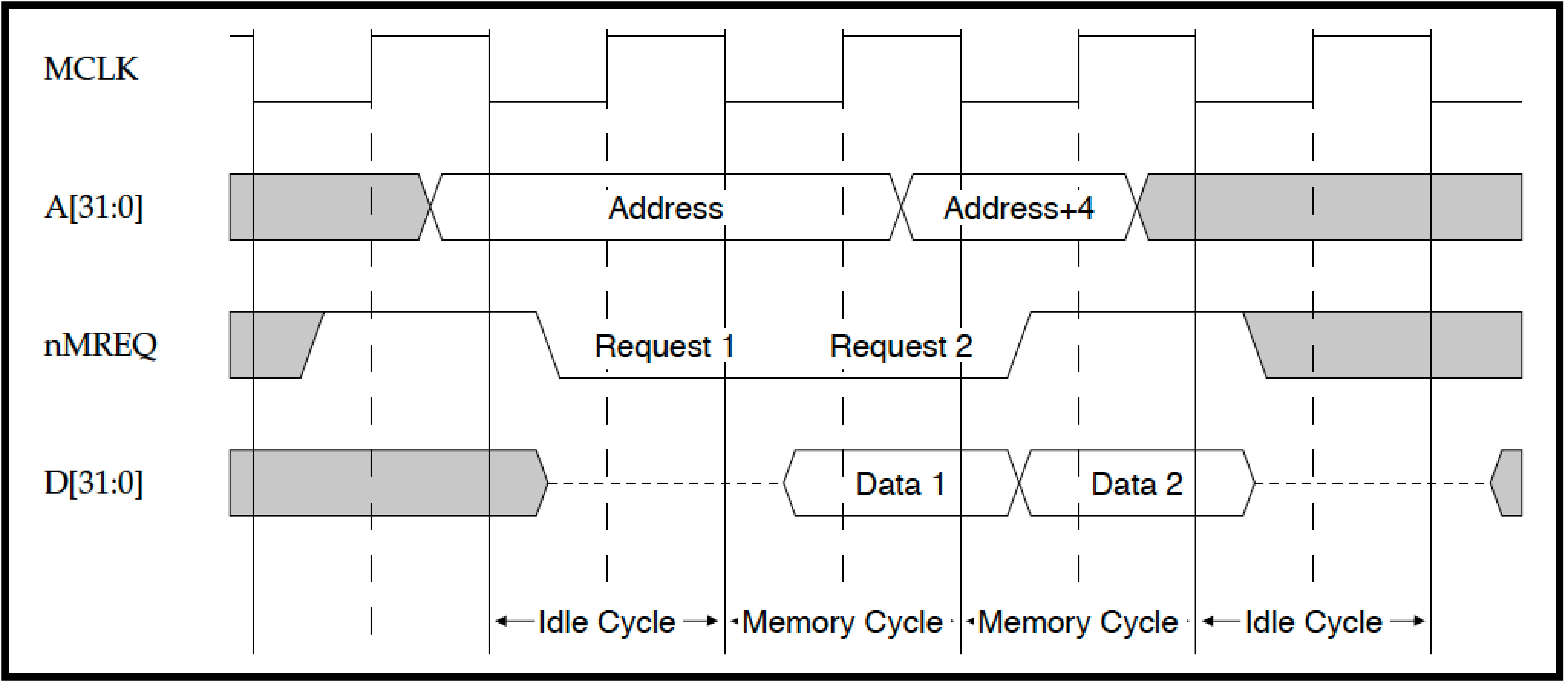


Figure 12-2: Simplified sequential access

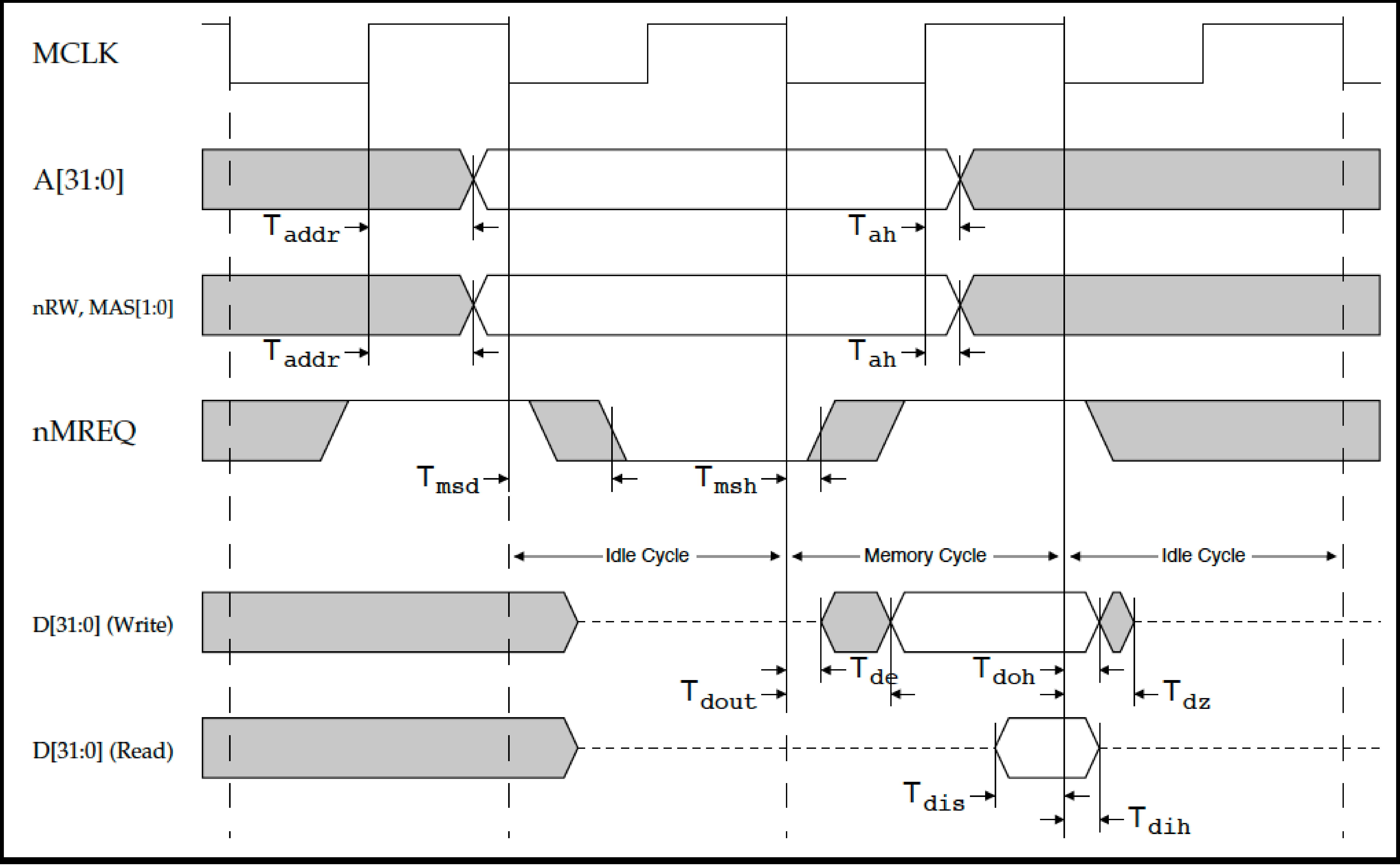


Figure 12-3: Single word read or write

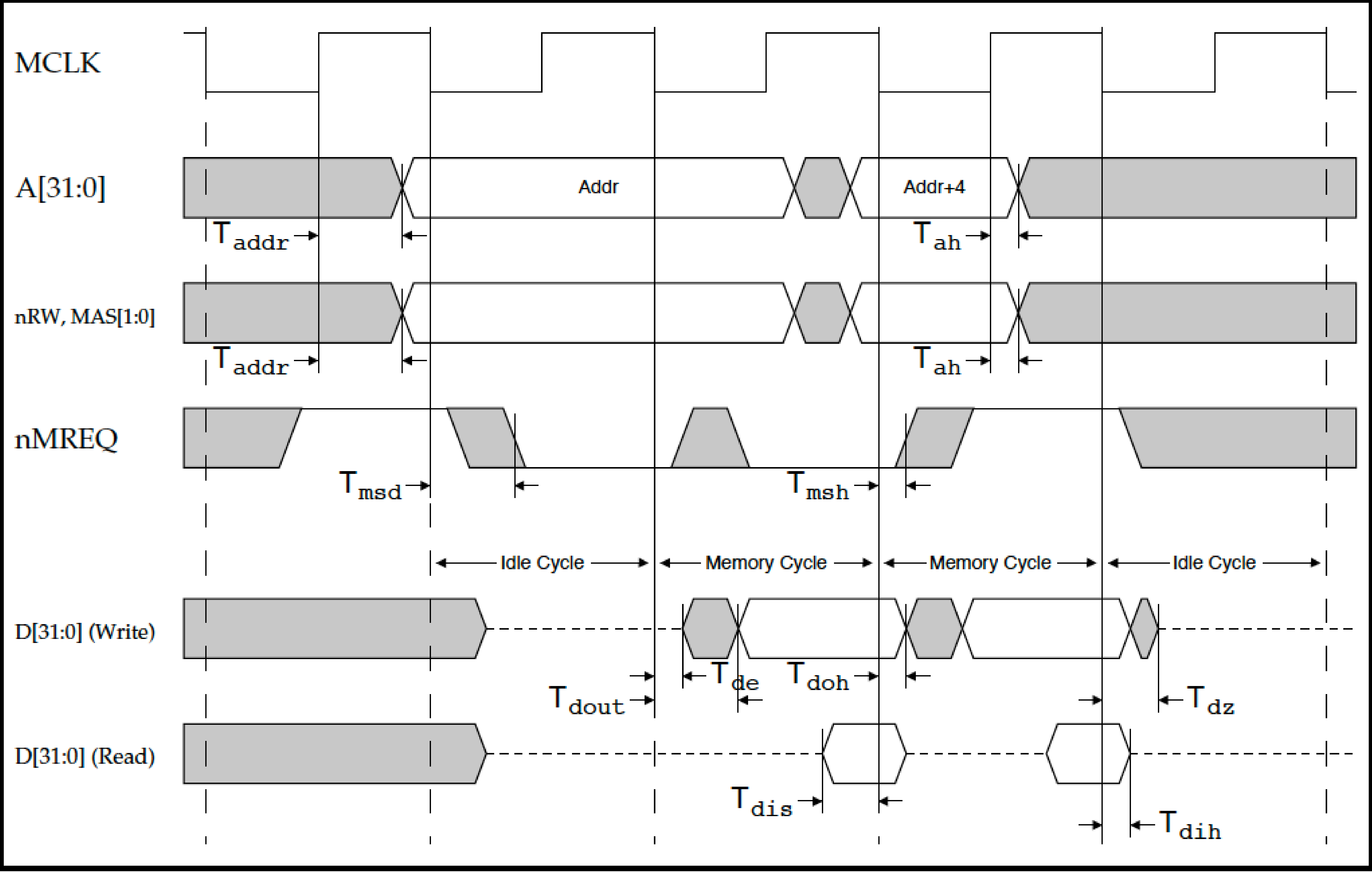


Figure 12-4: Two word sequential read or write

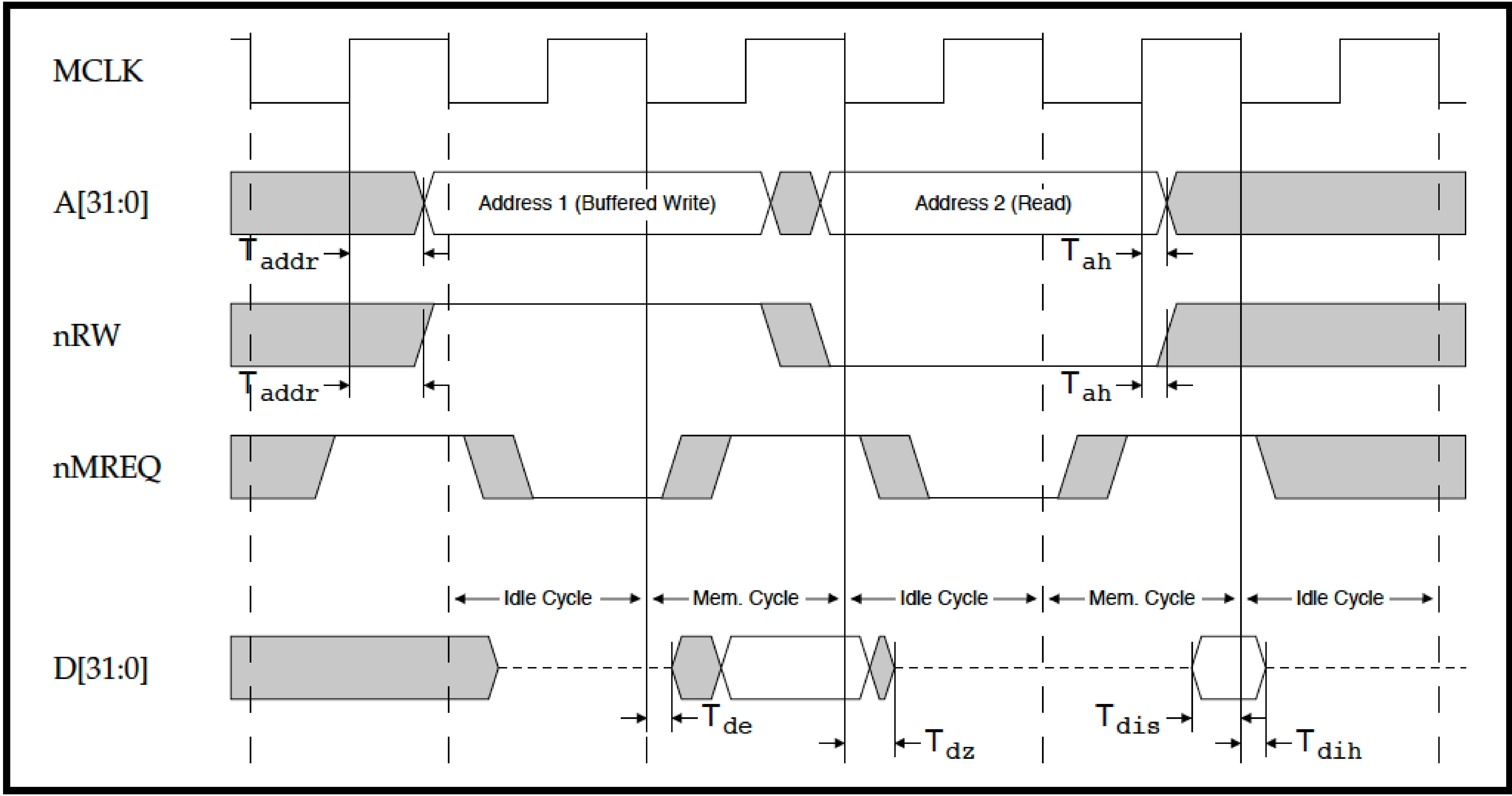


Figure 12-5: Minimum interval between bus accesses

MAS[1:0] is encoded as follows:

MAS bit 1	bit 0	Access size
0	0	byte
0	1	halfword
1	0	word
1	1	reserved, not used

Table 12-1: MAS encoding

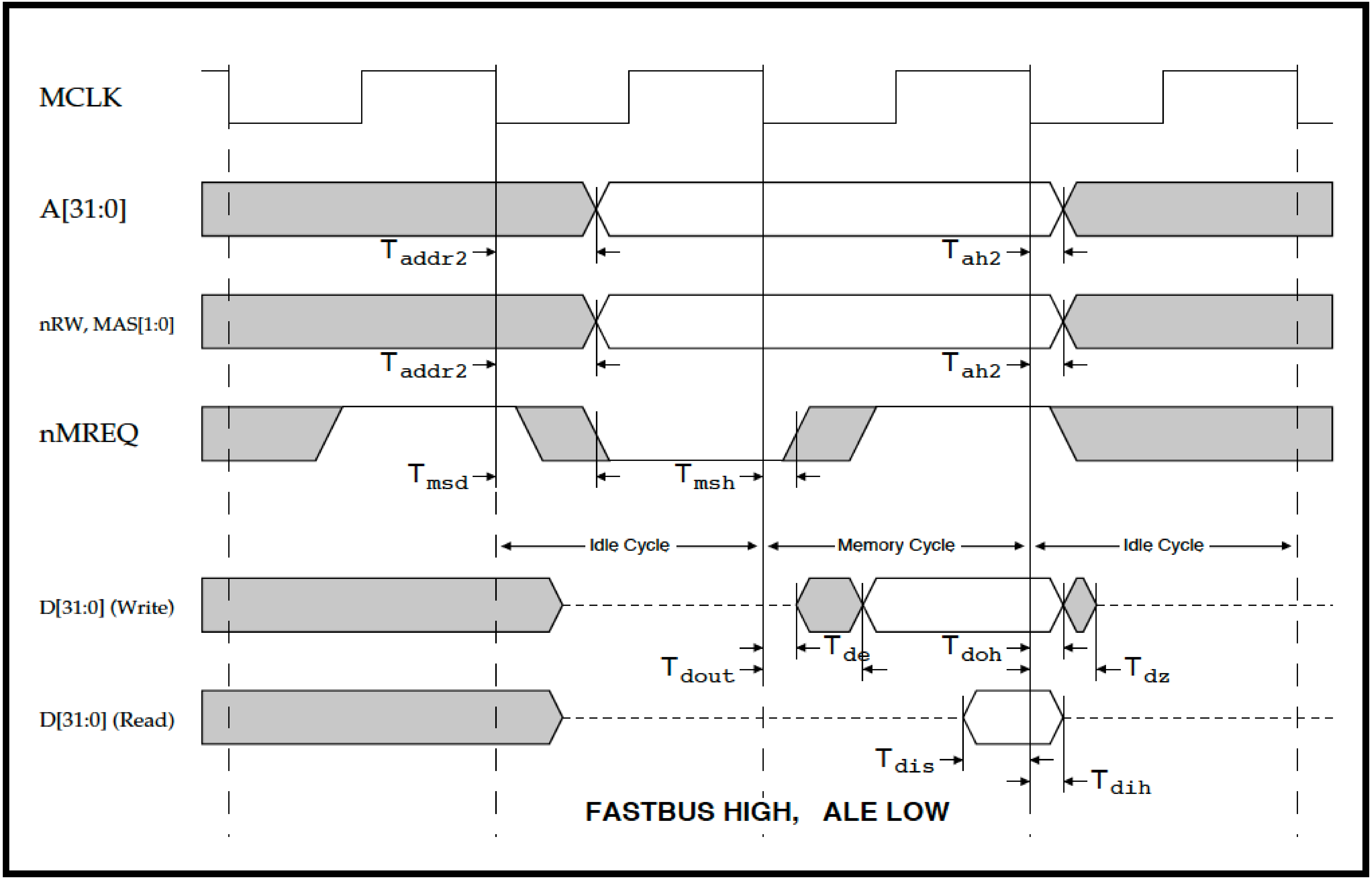
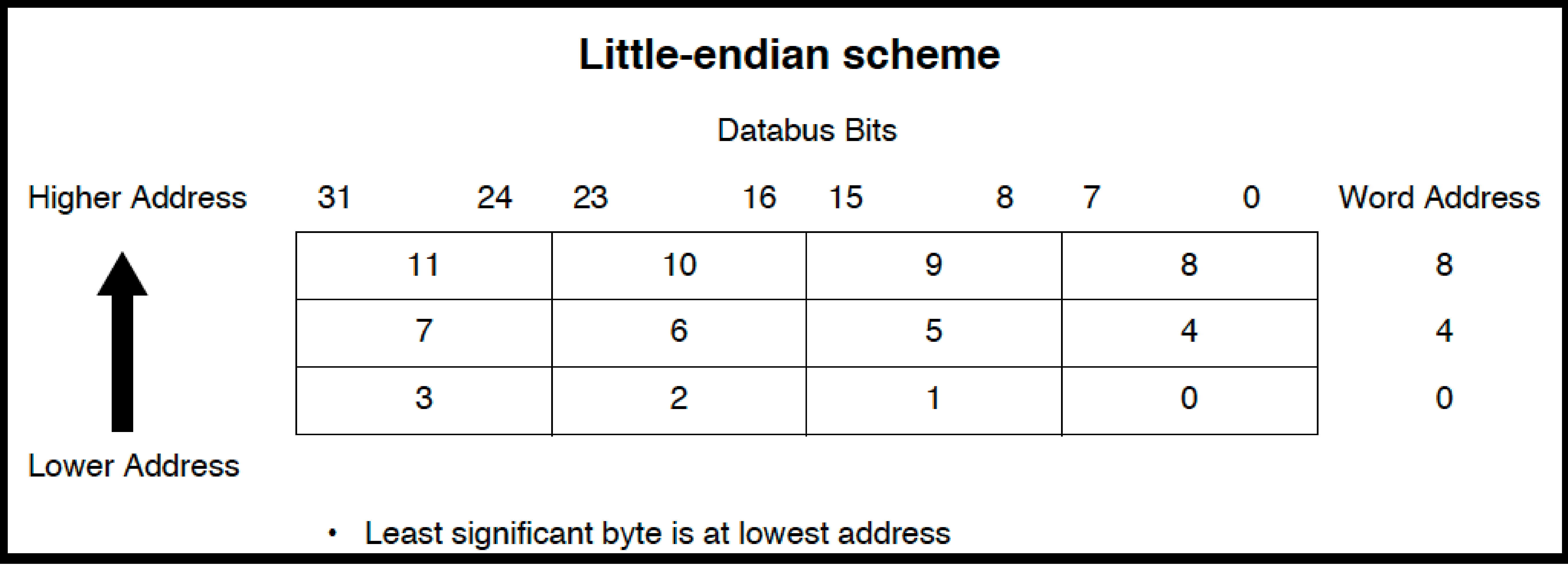


Figure 12-6: Single word read or write with delayed addressing



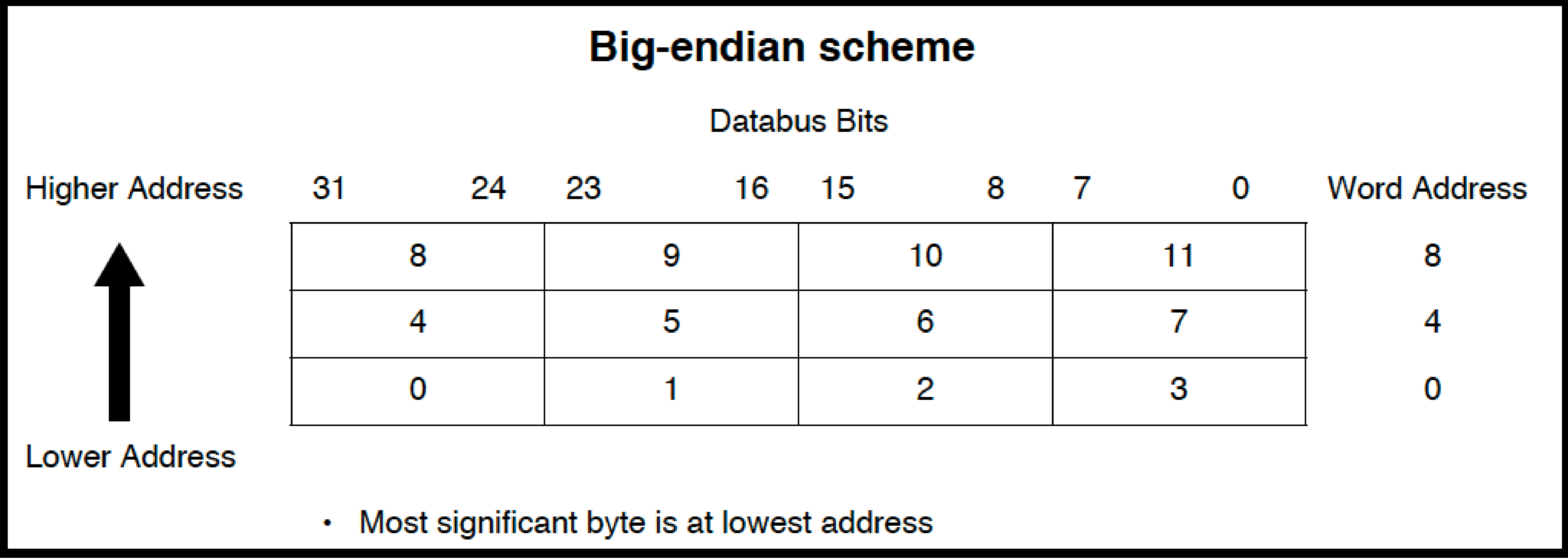
Little-endian scheme

Databus Bits

31 … 24	23 … 16	15 … 8	7 … 0	Word Address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

- Least significant byte is at lowest address

Figure 12-12: Little-endian addresses of bytes within word



31 24	23 16	15 8	7 0	Word Address
8	9	10	11	8
4	5	6	7	4
0	1	2	3	0

- Most significant byte is at lowest address

Figure 12-14: Big-endian addresses of bytes within words

MAS[1:0] Indicates	MAS[1]	MAS[0]	A[1]	A[0]	Memory Read/Write the Byte on			
					D[31:24]	D[23:16]	D[15:8]	D[7:0]
Word	1	0	X	X	Yes	Yes	Yes	Yes
Halfword	0	1	0	X	No	No	Yes	Yes
			1	X	Yes	Yes	No	No
Byte	0	0	0	0	No	No	No	Yes
			0	1	No	No	Yes	No
			1	0	No	Yes	No	No
			1	1	Yes	No	No	No
Reserved	1	1	X	X	Yes	Yes	Yes	Yes

Table 12-4: Decoding Byte Activity for little-endian system.

Notes *X means "don't care".*

MAS[1:0] *= 11 is Reserved for future use, it is never used by ARM810.*

The Byte Activity Decode indicated is recommended for compatibility with future ARM Microprocessors.

MAS[1:0] Indicates	MAS[1]	MAS[0]	A[1]	A[0]	Memory Read/Write the Byte on			
					D[31:24]	D[23:16]	D[15:8]	D[7:0]
Word	1	0	X	X	Yes	Yes	Yes	Yes
Halfword	0	1	0	X	Yes	Yes	No	No
			1	X	No	No	Yes	Yes
Byte	0	0	0	0	Yes	No	No	No
			0	1	No	Yes	No	No
			1	0	No	No	Yes	No
			1	1	No	No	No	Yes
Reserved	1	1	X	X	Yes	Yes	Yes	Yes

Table 12-5: Decoding Byte Activity for big-endian system.

Notes *X means "don't care".*

MAS[1:0] *= 11 is Reserved for future use, it is never used by ARM810.*

The Byte Activity Decode indicated is recommended for compatibility with future ARM Microprocessors.

CP15 Control Register B Bit	MAS[1:0]	A[1:0]	nBLS
0 (Little-endian)	1 0 (Word)	X X	0000
0	0 1 (Halfword)	0 X	1100
0	0 1	1 X	0011
0	0 0 (Byte)	0 0	1110
0	0 0	0 1	1101
0	0 0	1 0	1011
0	0 0	1 1	0111
1 (Big-endian)	1 0 (Word)	X X	0000
1	0 1 (Halfword)	0 X	0011
1	0 1	1 X	1100
1	0 0 (Byte)	0 0	0111
1	0 0	0 1	1011
1	0 0	1 0	1101
1	0 0	1 1	1110

Table 12-6: nBLS[3:0] *as a function of B,* MAS[1:0] *and* A[1:0]

Signal	When Low, enable read or write of byte connected to data bus bits
nBLS[0]	D[7:0]
nBLS[1]	D[15:8]
nBLS[2]	D[23:16]
nBLS[3]	D[31:24]

Table 12-7: nBLS[3:0] ***and Bytes of memory system***

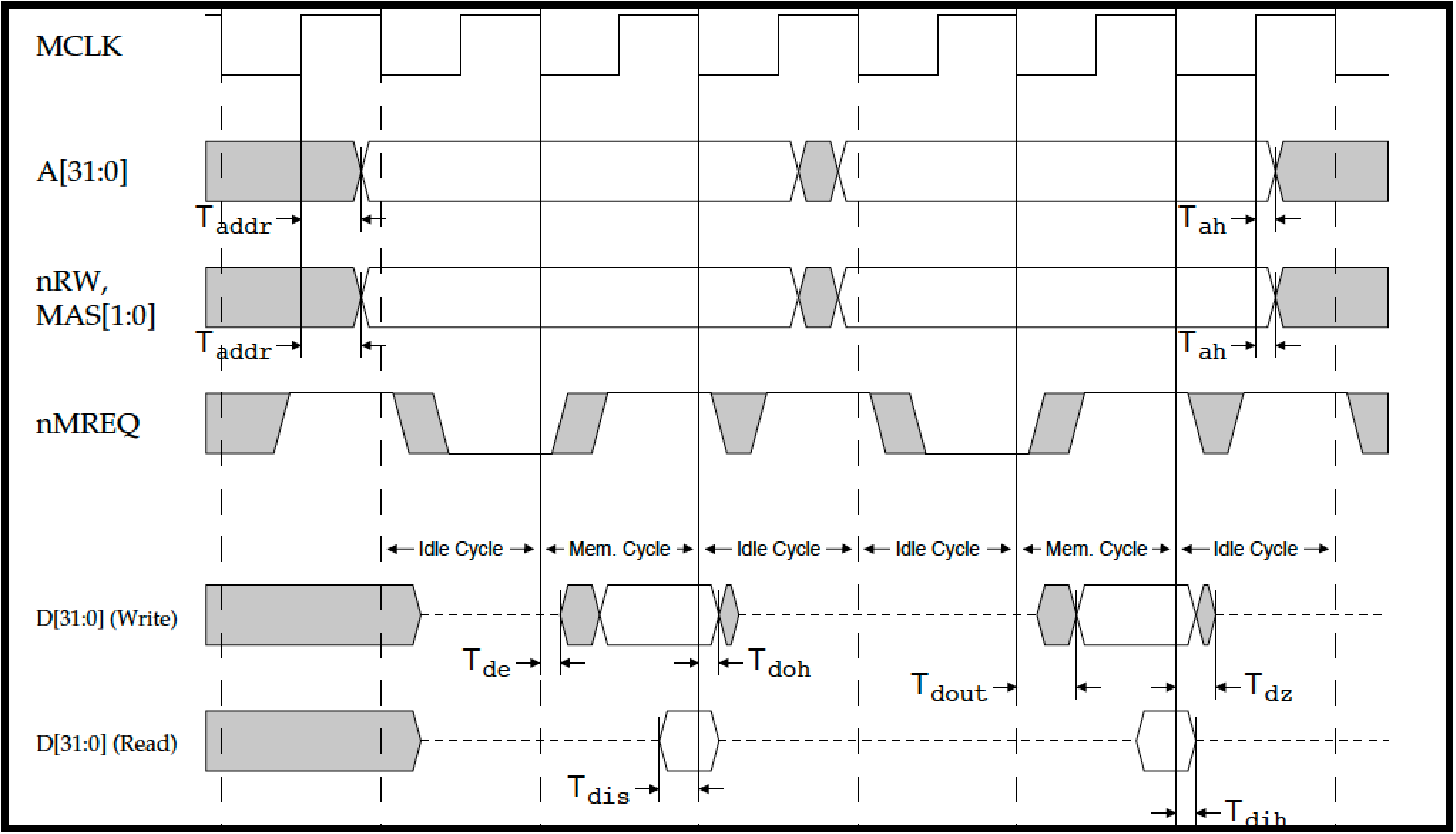


Figure 12-15: Two single word non-sequential unbuffered accesses

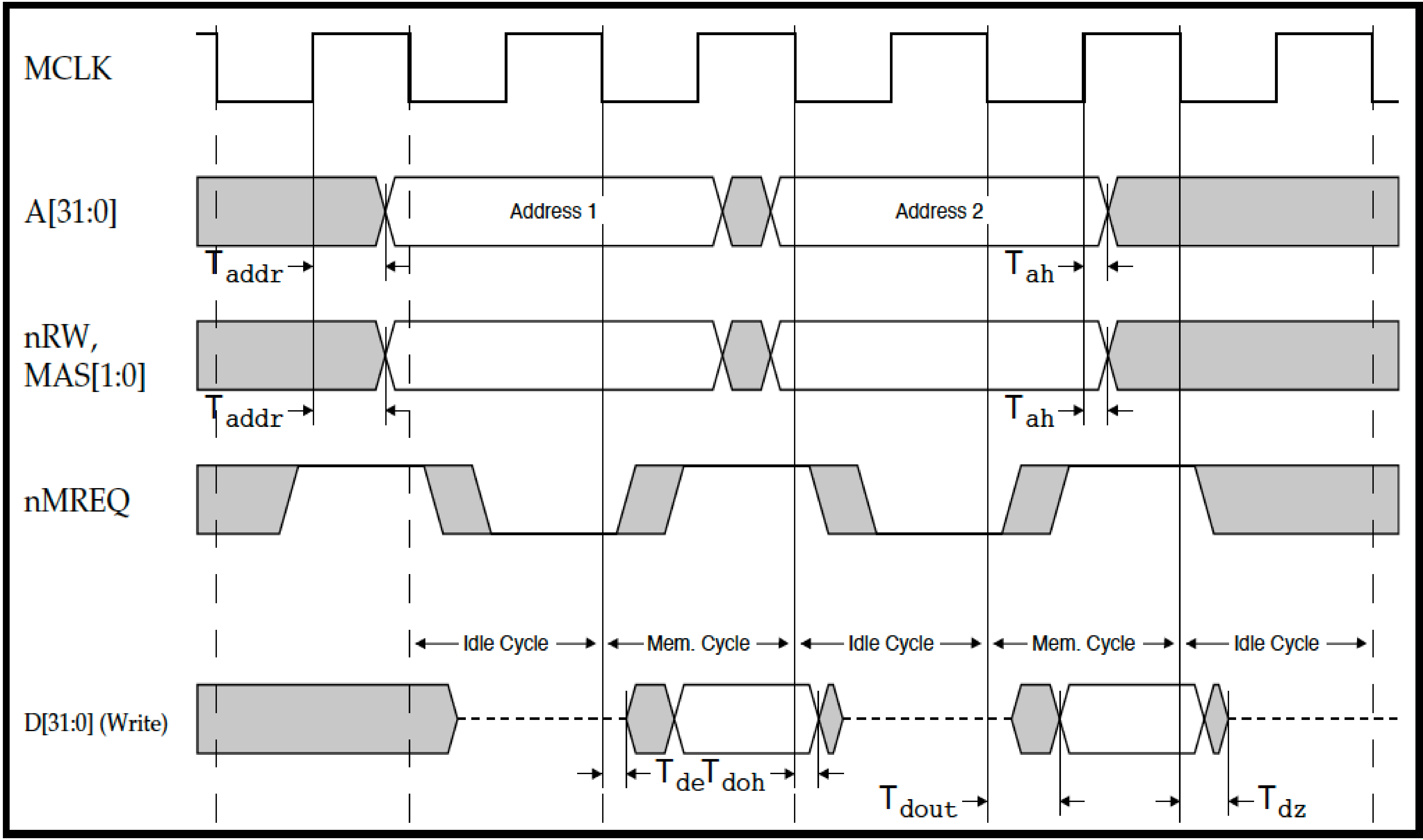


Figure 12-16: Two single word non-sequential buffered writes

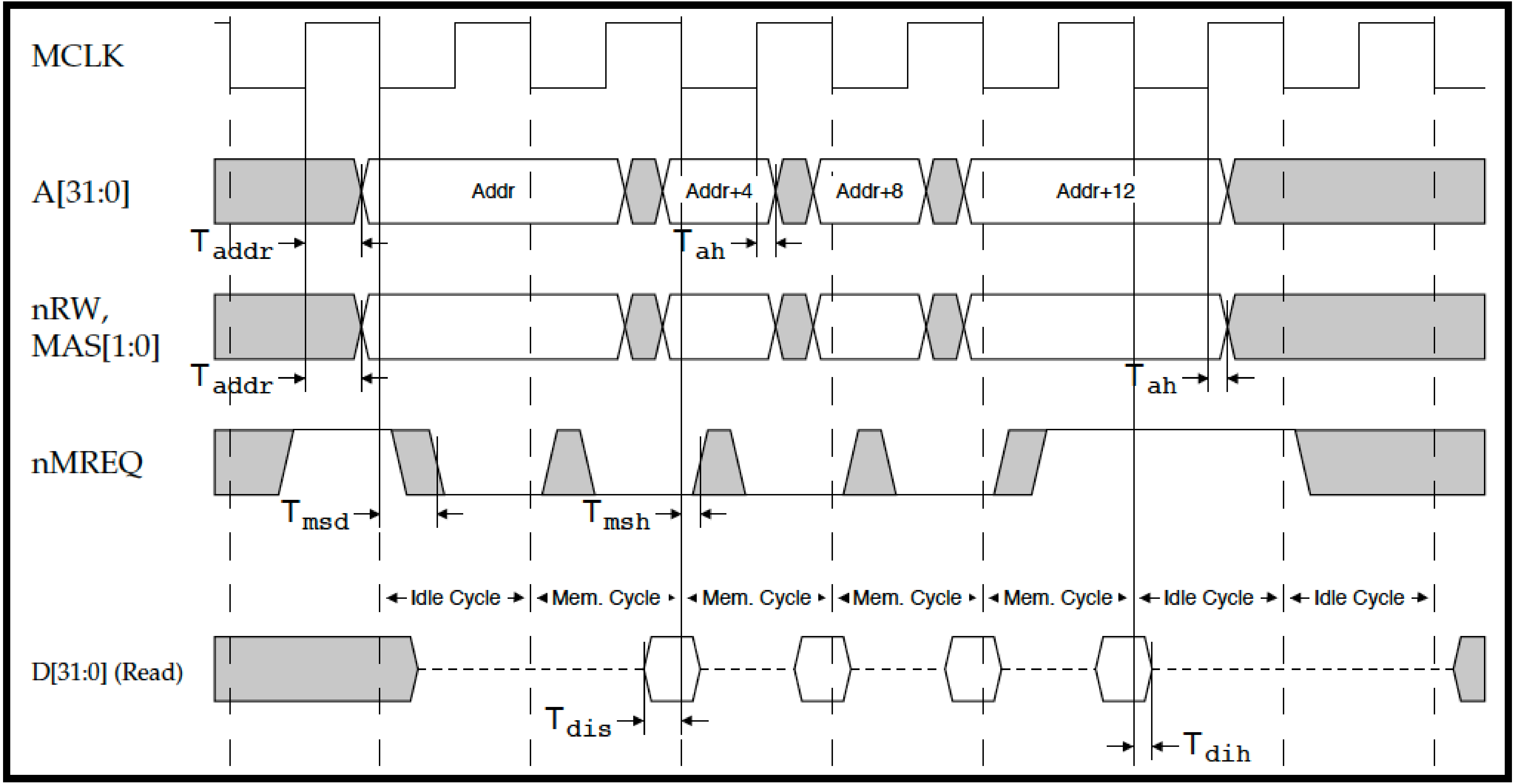


Figure 12-17: Linefetch

Устройство за плаваща запетая: Конвейери за умножение и натрупване, делене и коренуване и зареждане и съхранение. Режими. Обработка на къси вектори. Регистров файл. Програмен модел. Команди. Изключения.

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.

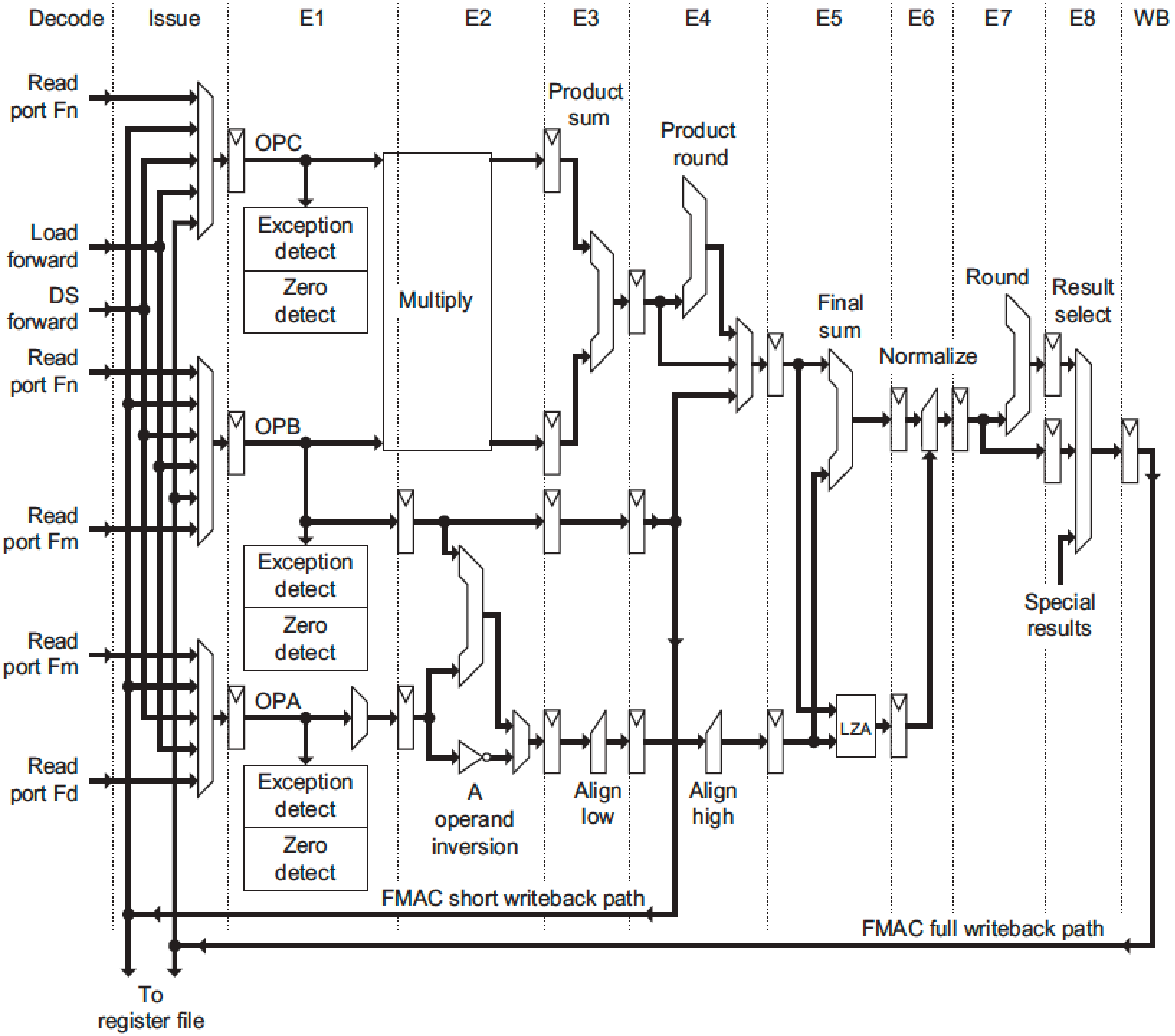


Figure 1-1 FMAC pipeline

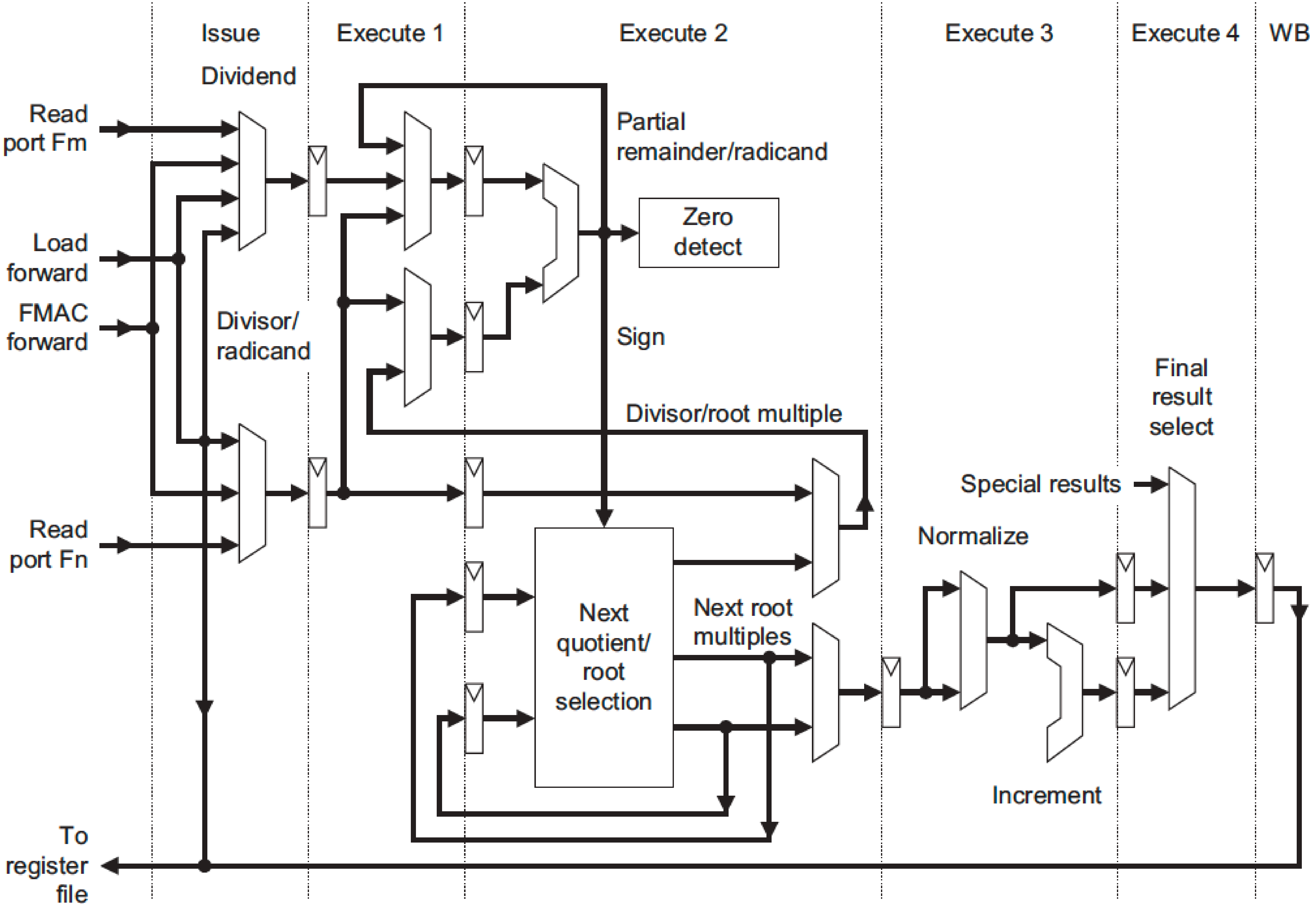


Figure 1-2 DS pipeline

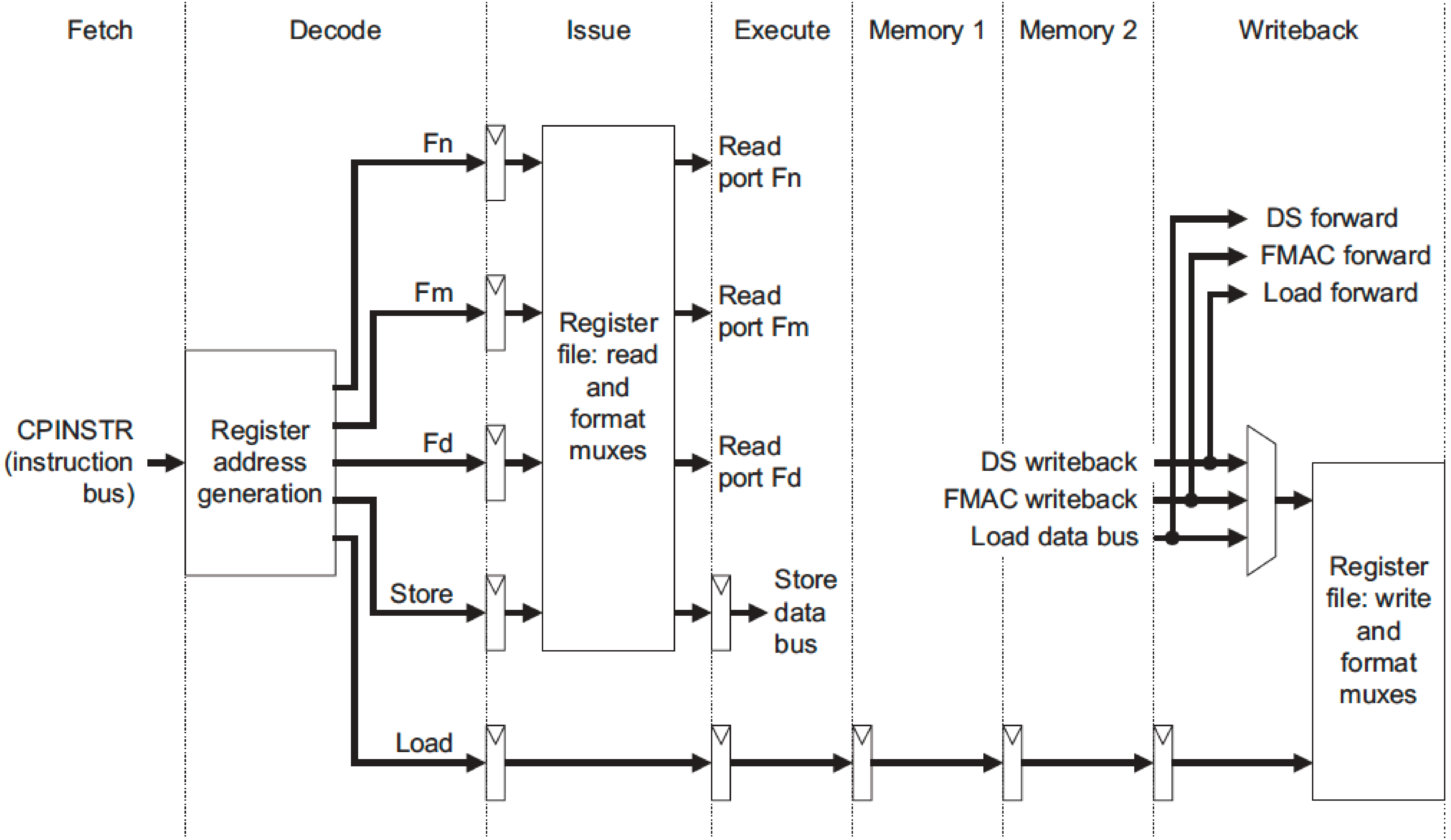


Figure 1-3 LS pipeline

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 2-1 shows the single-precision bit fields.

Figure 2-1 Single-precision data format

The single-precision data format contains:

- the sign bit, bit [31]
- the exponent, bits [30:23]
- the fraction, bits [22:0].

Single-precision format

A single-precision value is a 32-bit word, and must be word-aligned when held in memory. It has the following format:

The value represented depends primarily on the exponent field:

- If $0 <$ exponent $<$ `0xFF`, the value is a *normalized number* and is equal to:

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

  The *mantissa* of the value is the number 1.fraction, consisting of:

  - 1
  - a binary point
  - the 23 fraction bits.

  The mantissa therefore lies in the range $1 \leq \text{mantissa} < 2$ and is a multiple of 2^{-23}.

  The *unbiased exponent* of the value is the power to which 2 is raised in this formula. In this case, it is (exponent–127).

  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}.

Double-precision format has a *Most Significant Word* (MSW) and a *Least Significant Word* (LSW). Figure 2-2 shows the double-precision format.

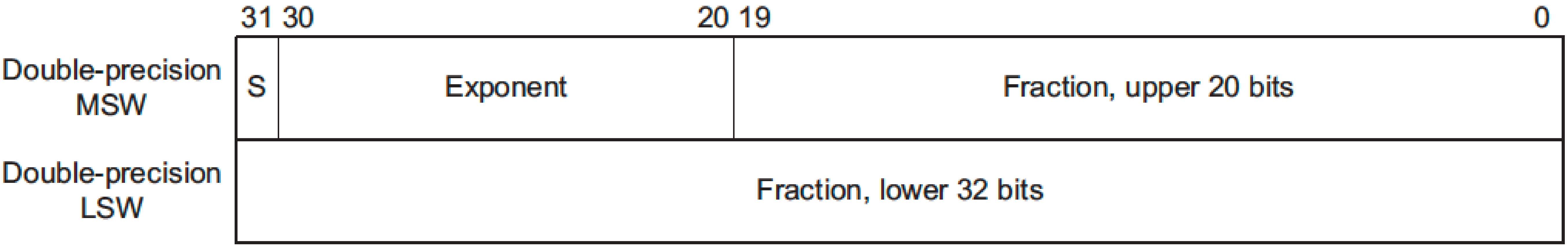


Figure 2-2 Double-precision data format

The MSW contains:

- the sign bit, bit [31]
- the exponent, bits [30:20]
- the upper 20 bits of the fraction, bits [19:0].

The LSW contains the lower 32 bits of the fraction.

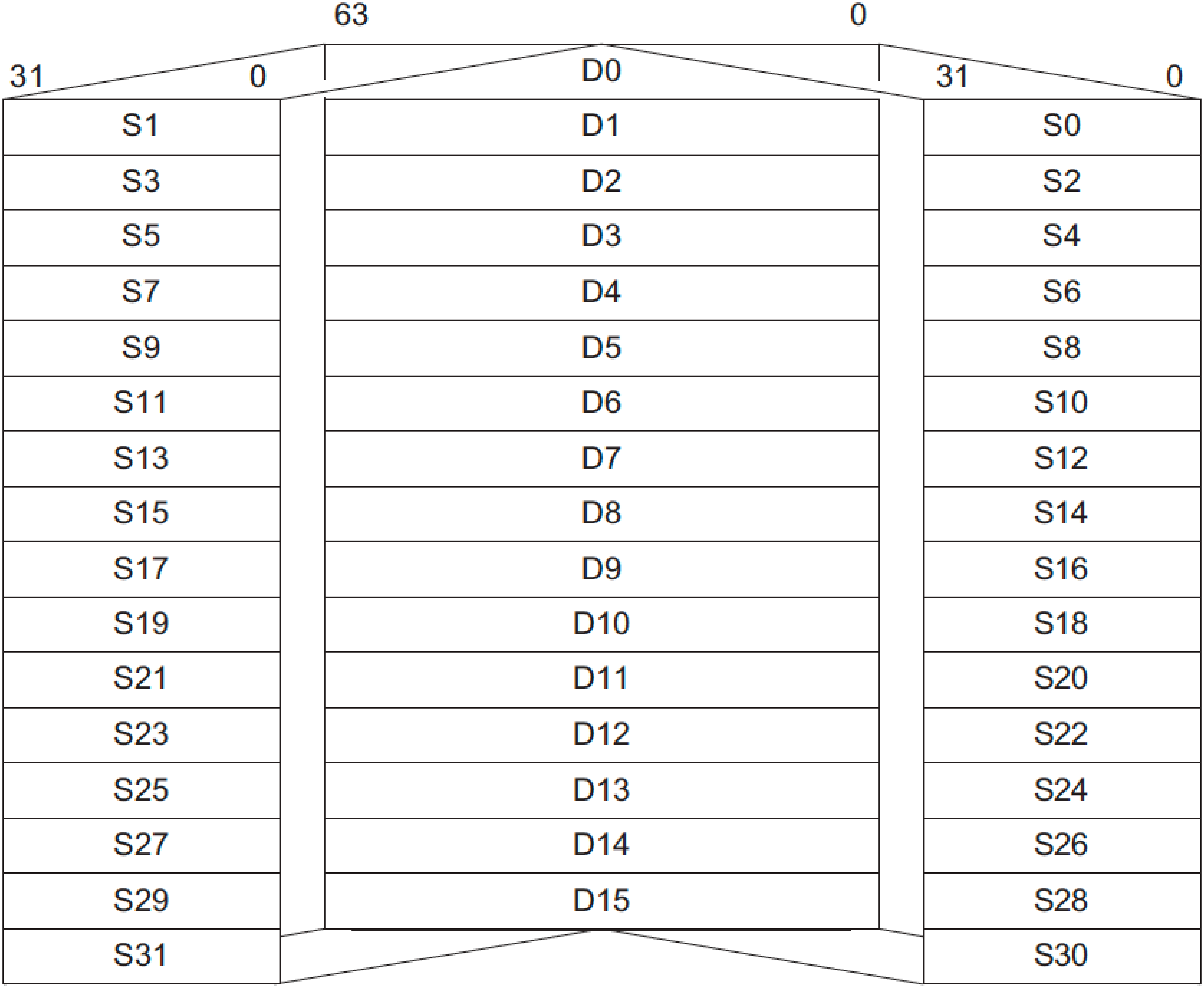


Figure 2-3 Register file access

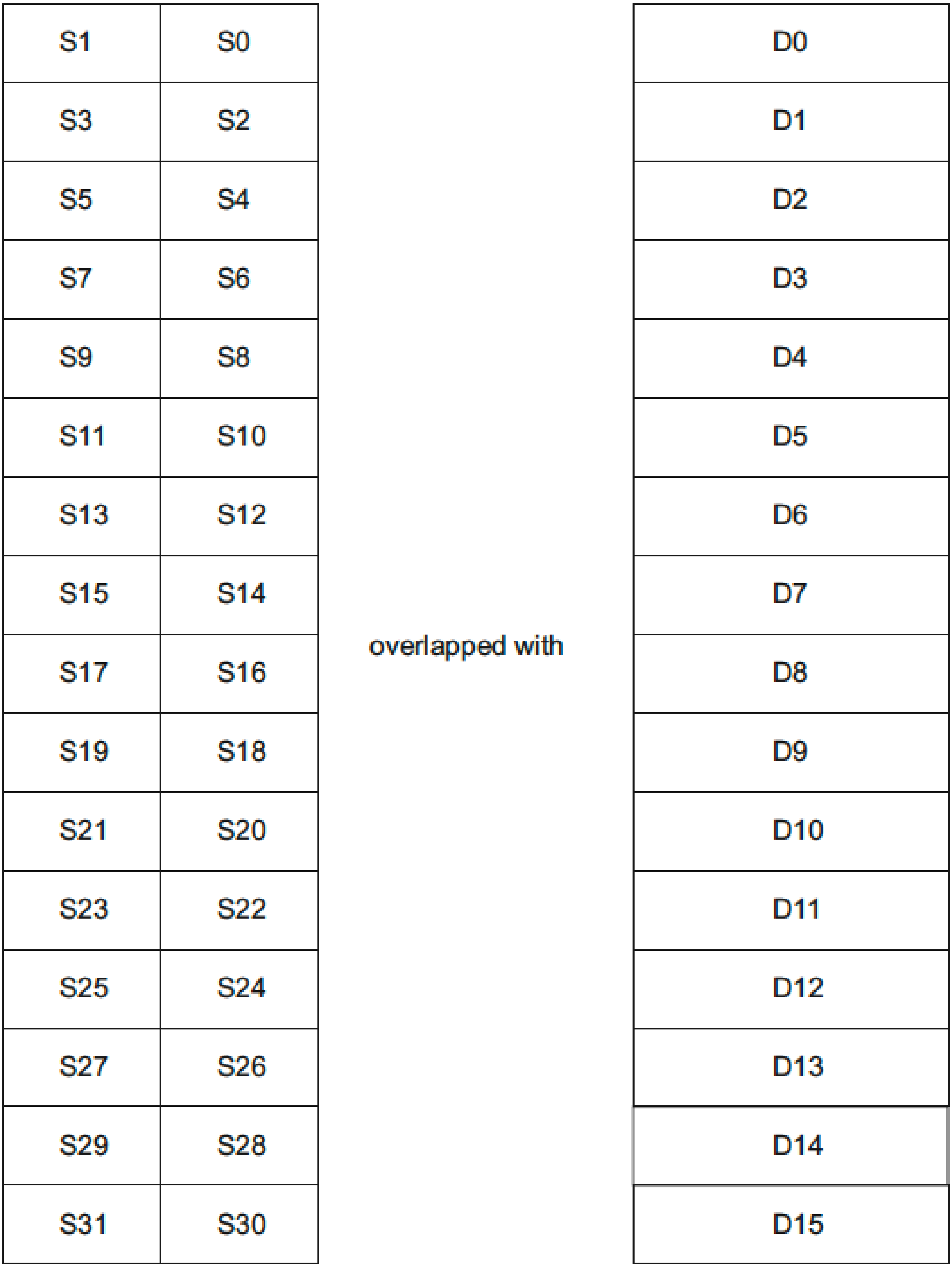


Figure C2-1 VFP general-purpose registers

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.

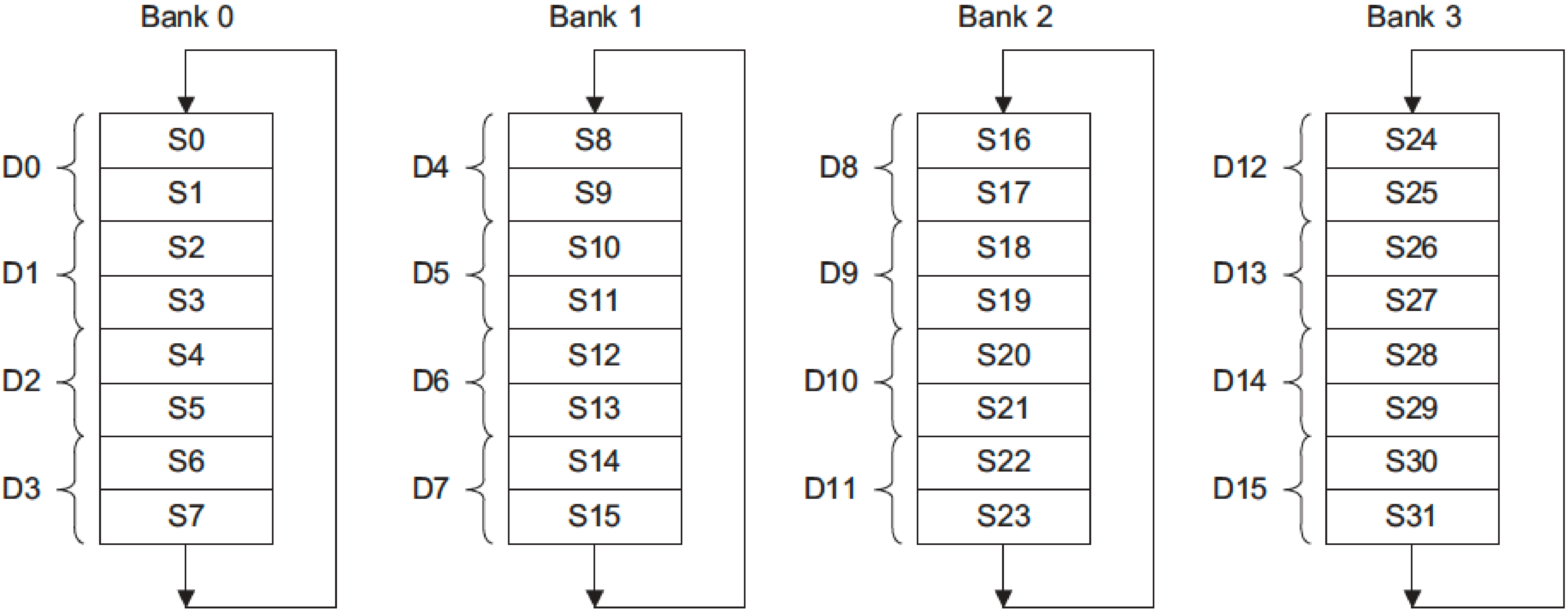


Figure 2-4 Register banks

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.

Table C3-1 VFP data-processing primary opcodes

p	q	r	s	Instruction name cp_num=10	Instruction name cp_num=11	Instruction functionality
0	0	0	0	`FMACS`	`FMACD`	Fd = Fd + (Fn * Fm)
0	0	0	1	`FNMACS`	`FNMACD`	Fd = Fd - (Fn * Fm)
0	0	1	0	`FMSCS`	`FMSCD`	Fd = -Fd + (Fn * Fm)
0	0	1	1	`FNMSCS`	`FNMSCD`	Fd = -Fd - (Fn * Fm)
0	1	0	0	`FMULS`	`FMULD`	Fd = Fn * Fm
0	1	0	1	`FNMULS`	`FNMULD`	Fd = -(Fn * Fm)
0	1	1	0	`FADDS`	`FADDD`	Fd = Fn + Fm
0	1	1	1	`FSUBS`	`FSUBD`	Fd = Fn - Fm
1	0	0	0	`FDIVS`	`FDIVD`	Fd = Fn / Fm
1	0	0	1	-	-	UNDEFINED
1	0	1	0	-	-	UNDEFINED
1	0	1	1	-	-	UNDEFINED
1	1	0	0	-	-	UNDEFINED
1	1	0	1	-	-	UNDEFINED
1	1	1	0	-	-	UNDEFINED
1	1	1	1	See Table C3-2 on page C3-4	See Table C3-2 on page C3-4	Extension instructions

Table C3-2 VFP data-processing extension opcodes

Extension opcode		Instruction name		
Fn	N	cp_num=10	cp_num=11	Instruction functionality
0000	0	FCPYS	FCPYD	Fd = Fm
0000	1	FABSS	FABSD	Fd = abs(Fm)
0001	0	FNEGS	FNEGD	Fd = -Fm
0001	1	FSQRTS	FSQRTD	Fd = sqrt(Fm)
001x	x	-	-	UNDEFINED
0100	0	FCMPS	FCMPD	Compare Fd with Fm, no exceptions on quiet NaNs
0100	1	FCMPES	FCMPED	Compare Fd with Fm, with exceptions on quiet NaNs
0101	0	FCMPZS	FCMPZD	Compare Fd with 0, no exceptions on quiet NaNs
0101	1	FCMPEZS	FCMPEZD	Compare Fd with 0, with exceptions on quiet NaNs
0110	x	-	-	UNDEFINED
0111	0	-	-	UNDEFINED
0111	1	FCVTDS	FCVTSD	Single ↔ double-precision conversions
1000	0	FUITOS	FUITOD	Unsigned integer → floating-point conversions
1000	1	FSITOS	FSITOD	Signed integer → floating-point conversions
1001	x	-	-	UNDEFINED
101x	x	-	-	UNDEFINED
1100	0	FTOUIS	FTOUID	Floating-point → unsigned integer conversions
1100	1	FTOUIZS	FTOUIZD	Floating-point → unsigned integer conversions, RZ mode
1101	0	FTOSIS	FTOSID	Floating-point → signed integer conversions
1101	1	FTOSIZS	FTOSIZD	Floating-point → signed integer conversions, RZ mode
111x	x	-	-	UNDEFINED

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.

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.

Изключения и прекъсвания: Изключения. Прекъсвания – видове и връзка с режимите на МП. Таблица на векторите на изключенията и прекъсванията. Начално установяване на МП.

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

Устройство за управление на паметта: Функции. Регистри. Транслация на адресите. Дескриптори. Кеширане и буфериране. Грешки. Буфер за запис.

The Memory Management MMU performs two primary functions: it translates virtual addresses into physical addresses, and it controls memory access permissions. The MMU hardware required to perform these functions consists of a Translation Look-aside Buffer (TLB), access control logic, and translation table walking logic.

The MMU supports memory accesses based on Sections or Pages. Sections are comprised of 1MB blocks of memory. Two different page sizes are supported: Small Pages consist of 4KB blocks of memory and Large Pages consist of 64KB blocks of memory. (Large Pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB). Additional access control mechanisms are extended within Small Pages to 1KB Sub-Pages and within Large Pages to 16KB Sub-Pages.

The MMU also supports the concept of domains - areas of memory that can be defined to possess individual access rights. The Domain Access Control Register is used to specify access rights for up to 16 separate domains.

The TLB caches 64 translated entries. During most memory accesses, the TLB provides the translation information to the access control logic.

If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted and an off-chip access is required, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing value. The entry to be overwritten is chosen by cycling sequentially through the TLB locations.

When the MMU is turned off (as happens on reset), the virtual address is output directly onto the physical address bus.

<table>
<tr><th>Register</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1 write</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="16">Control</td><td>0</td><td>R</td><td>S</td><td>B</td><td>1</td><td>D</td><td>P</td><td>W</td><td>C</td><td>A</td><td>M</td></tr>
<tr><td>2 write</td><td colspan="18">Translation Table Base</td><td colspan="14"></td></tr>
<tr><td rowspan="2">3 write</td><td colspan="32">Domain Access Control</td></tr>
<tr><td colspan="2">15</td><td colspan="2">14</td><td colspan="2">13</td><td colspan="2">12</td><td colspan="2">11</td><td colspan="2">10</td><td colspan="2">9</td><td colspan="2">8</td><td colspan="2">7</td><td colspan="2">6</td><td colspan="2">5</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">2</td><td colspan="2">1</td><td colspan="2">0</td></tr>
<tr><td>5 read</td><td colspan="20">Fault Status</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="4">Domain</td><td colspan="4">Status</td></tr>
<tr><td>5 write</td><td colspan="32">Flush TLB</td></tr>
<tr><td>6 read</td><td colspan="32">Fault Address</td></tr>
<tr><td>6 write</td><td colspan="20">Purge Address</td><td colspan="12"></td></tr>
</table>

Figure 9-1: MMU register summary

Note *The registers not shown are reserved and should not be used.*

The ARM710a Processor provides several 32-bit registers which determine the operation of the MMU. The format for these registers is shown in ➲*Figure 9-1: MMU register summary* on page 9-3. A brief description of the registers is provided below. Each register will be discussed in more detail within the section that describes its use.

Data is written to and read from the MMU's registers using the ARM CPU's MRC and MCR coprocessor instructions.

The **Translation Table Base Register** holds the physical address of the base of the translation table maintained in main memory. Note that this base must reside on a 16kB boundary.

The **Domain Access Control Register** consists of sixteen 2-bit fields, each of which defines the access permissions for one of the sixteen Domains (D15-D0).

The **Fault Status Register** indicates the domain and type of access being attempted when an abort occurred. Bits 7:4 specify which of the sixteen domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is different for internal and external faults (as indicated by bit 0 in the register) and is shown in ➲*Table 9-4: Priority encoding of fault status* on page 9-12. A write to this register flushes the TLB.

The **Fault Address Register** holds the virtual address of the access which was attempted when a fault occurred. A write to this register causes the data written to be treated as an address and, if it is found in the TLB, the entry is marked as invalid. (This operation is known as a TLB purge). The Fault Status Register and Fault Address Register are only updated for data faults, not for prefetch faults.

Bits 31:14 of the Translation Table Base register are concatenated with bits 31:20 of the virtual address to produce a 30-bit address as illustrated in *Figure 9-3: Accessing the translation table first level descriptors*. This address selects a four-byte translation table entry which is a First Level Descriptor for either a Section or a Page (bit1 of the descriptor returned specifies whether it is for a Section or Page)

.

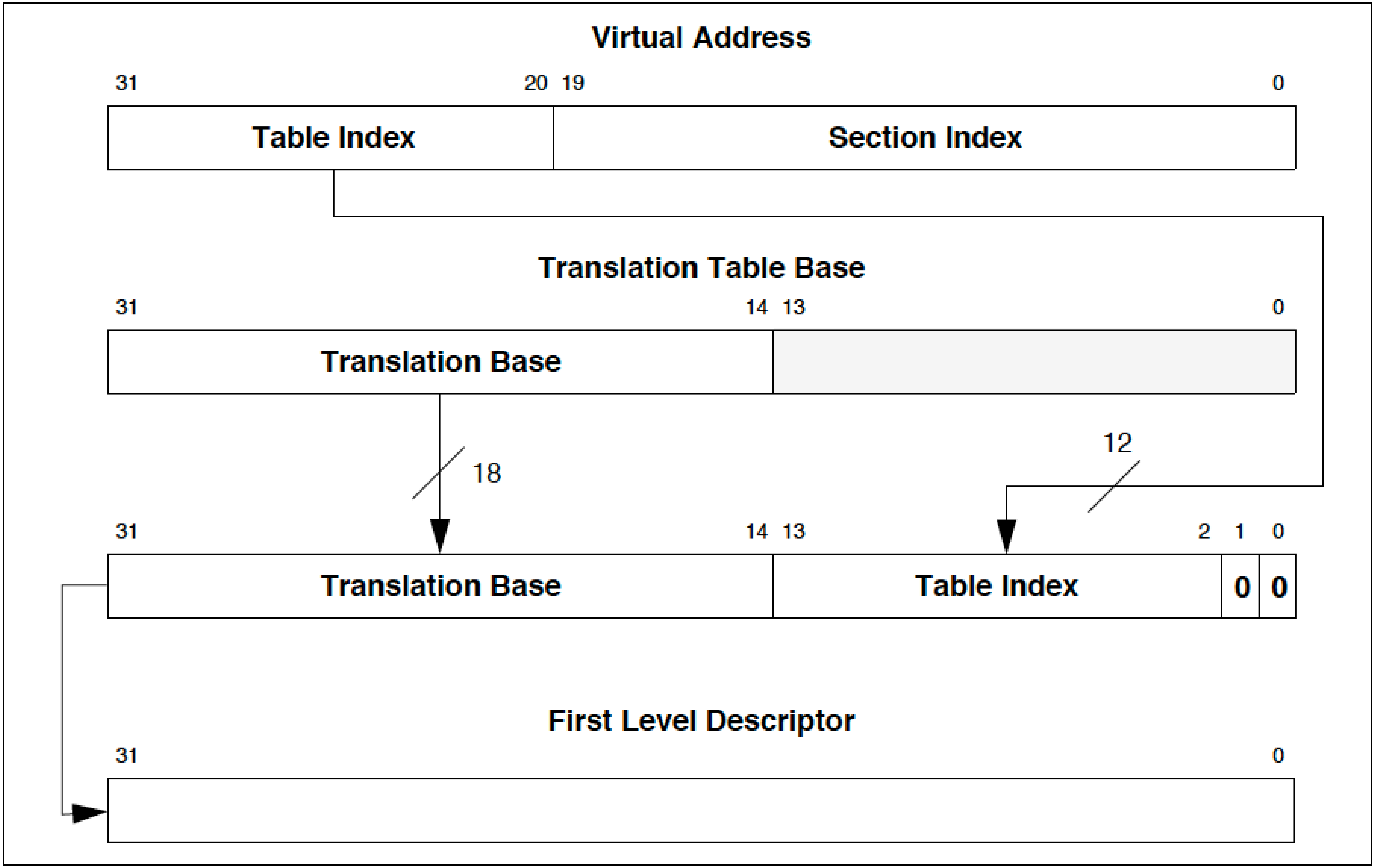


Figure 9-3: Accessing the translation table first level descriptors

9.4 Level One Descriptor

The Level One Descriptor returned is either a Page Table Descriptor or a Section Descriptor, and its format varies accordingly. The following figure illustrates the format of Level One Descriptors.

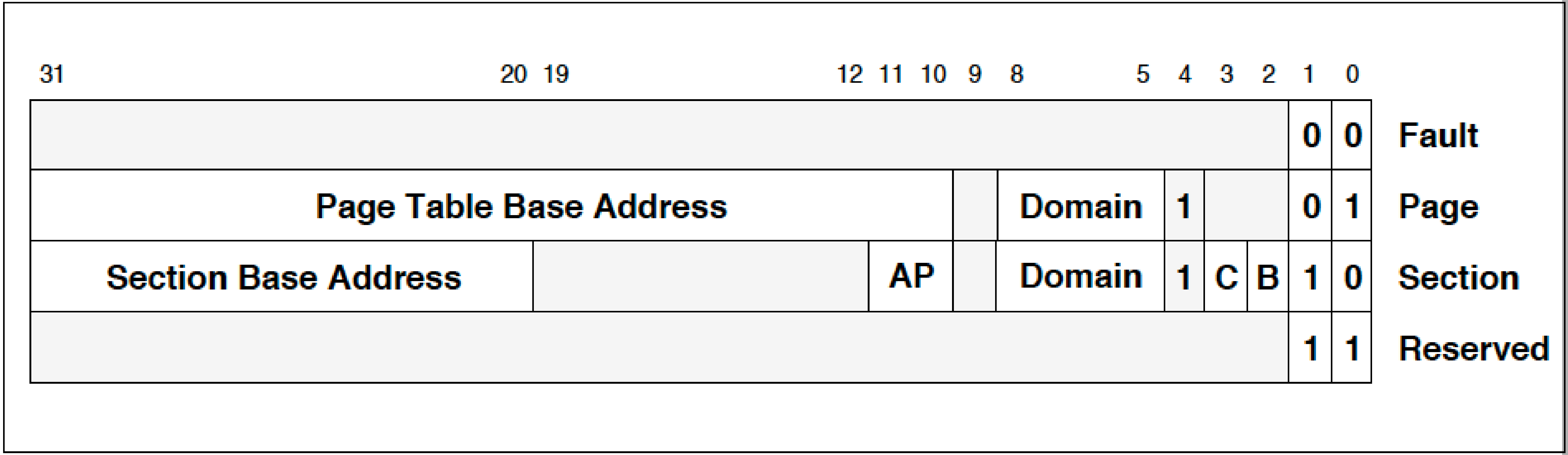


Figure 9-4: Level one descriptors

The two least significant bits indicate the descriptor type and validity, and are interpreted as shown below..

Value	Meaning	Notes
0 0	Invalid	Generates a Section Translation Fault
0 1	Page	Indicates that this is a Page Descriptor
1 0	Section	Indicates that this is a Section Descriptor
1 1	Reserved	Reserved for future use

Table 9-1: Interpreting level one descriptor Bits [1:0]

9.5 Page Table Descriptor

Bits 3:2 are always written as 0.

Bit 4 should be written to 1 for backward compatibility.

Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.

Bits 31:10 form the base for referencing the Page Table Entry. (The page table index for the entry is derived from the virtual address as illustrated in ◆*Figure 9-7: Small page translation* on page 9-9).

If a Page Table Descriptor is returned from the Level One fetch, a Level Two fetch is initiated as described below.

9.6 Section Descriptor

Bits 3:2 (C, & B) control the cache- and write-buffer-related functions as follows:

C - Cacheable: indicates that data at this address will be placed in the cache (if the cache is enabled).

B - Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

Bit 4 should be written to 1 for backward compatibility.

Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.

Bits 11:10 (AP) specify the access permissions for this section and are interpreted as shown in ➲*Table 9-2: Interpreting access permission (AP) bits* on page 9-6. Their interpretation is dependent upon the setting of the S and R bits (control register bits 8 and 9). Note that the Domain Access Control specifies the primary access control; the AP bits only have an effect in client mode. Refer to section on access permissions

AP	S	R	Permissions Supervisor	User	Notes
00	0	0	No Access	No Access	Any access generates a permission fault
00	1	0	Read Only	No Access	Supervisor read only permitted
00	0	1	Read Only	Read Only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/Write	No Access	Access allowed only in Supervisor mode
10	x	x	Read/Write	Read Only	Writes in User mode cause permission fault
11	x	x	Read/Write	Read/Write	All access types permitted in both modes.
xx	1	1	Reserved		

Table 9-2: Interpreting access permission (AP) bits

Bits 19:12 are always written as 0.

Bits 31:20 form the corresponding bits of the physical address for the 1MByte section.

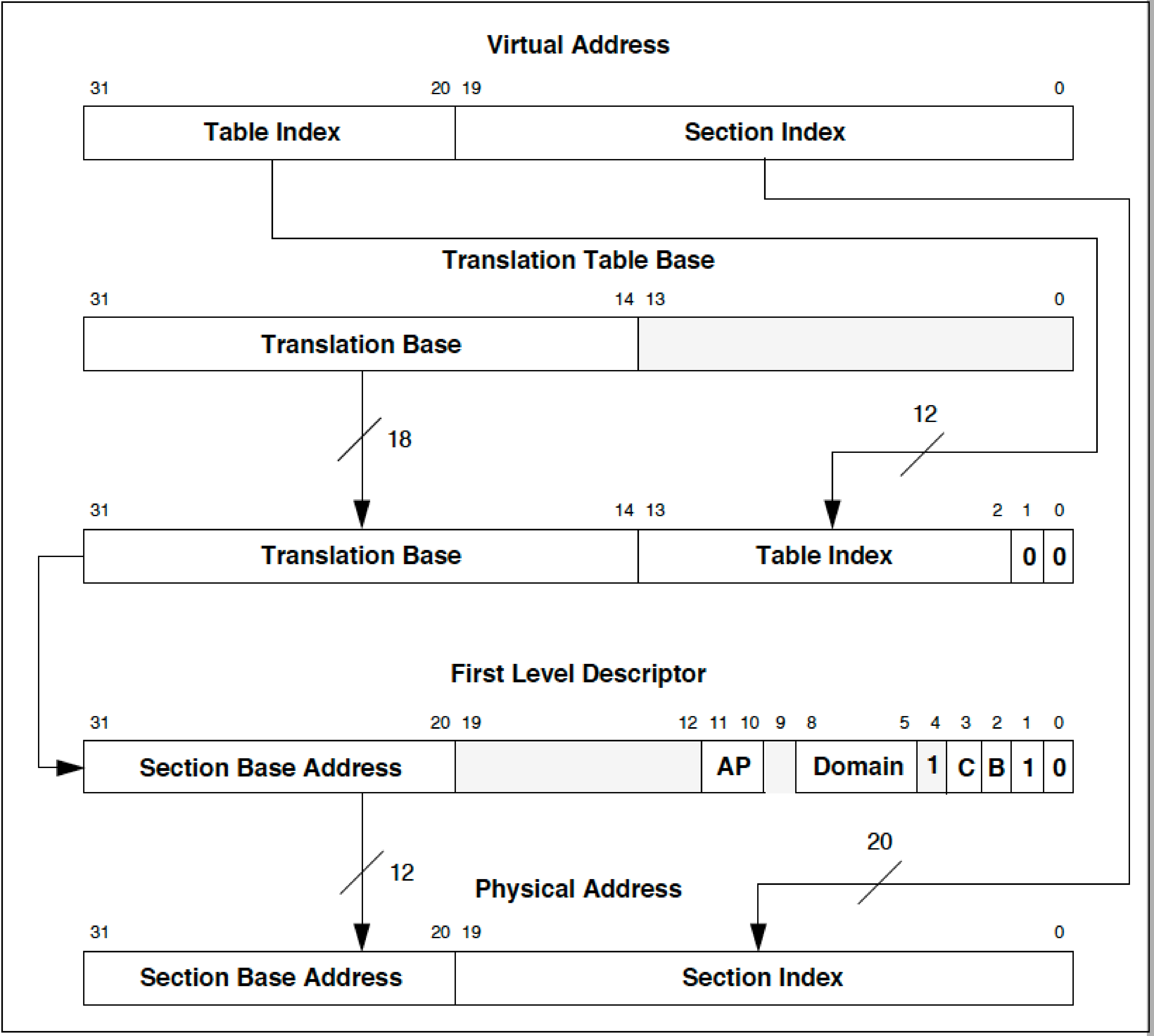


Figure 9-5: Section translation

9.8 Level Two Descriptor

If the Level One fetch returns a Page Table Descriptor, this provides the base address of the page table to be used. The page table is then accessed as described in *Figure 9-7: Small page translation* on page 9-9, and a Page Table Entry, or Level Two Descriptor, is returned. This in turn may define either a Small Page or a Large Page access. The figure below shows the format of Level Two Descriptors.

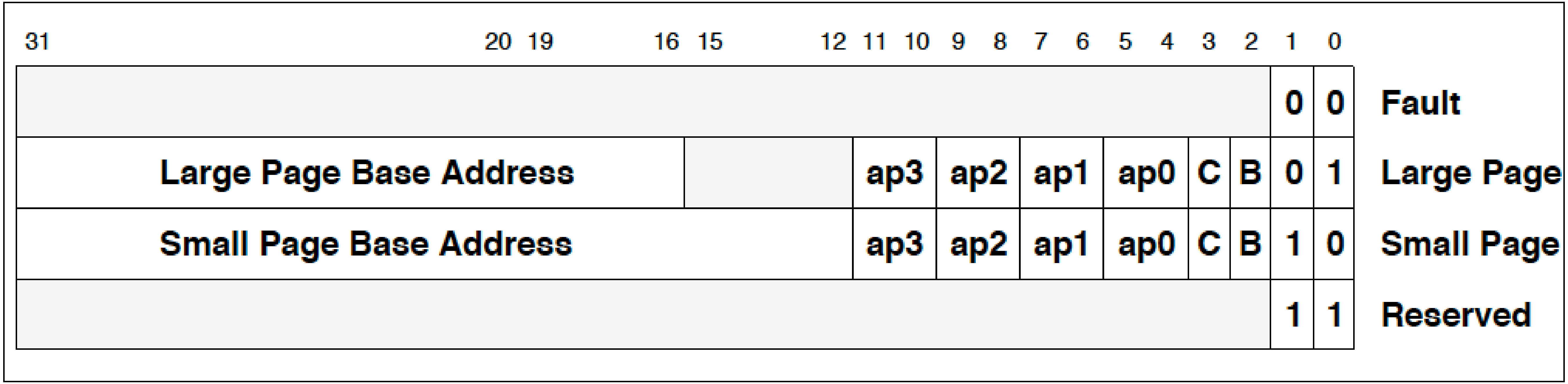


Figure 9-6: Page table entry (level two descriptor)

The two least significant bits indicate the page size and validity, and are interpreted as follows.

Value	Meaning	Notes
0 0	Invalid	Generates a Page Translation Fault
0 1	Large Page	Indicates that this is a 64 kB Page
1 0	Small Page	Indicates that this is a 4 kB Page
1 1	Reserved	Reserved for future use

Table 9-3: Interpreting page table entry bits 1:0

Bit 2 B - Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

Bit 3 C - Cacheable: indicates that data at this address will be placed in the IDC (if the cache is enabled).

Bits 11:4 specify the access permissions (ap3 - ap0) for the four sub-pages and interpretation of these bits is described earlier in ➲*Table 9-1: Interpreting level one descriptor Bits [1:0]* on page 9-5.

For large pages, **bits 15:12** are programmed as 0.

Bits 31:12 (small pages) or bits **31:16** (large pages) are used to form the corresponding bits of the physical address - the physical page number. (The page index is derived from the virtual address as illustrated in ➲*Figure 9-7: Small page translation* on page 9-9 and ➲*Figure 9-8: Large page translation* on page 9-10).

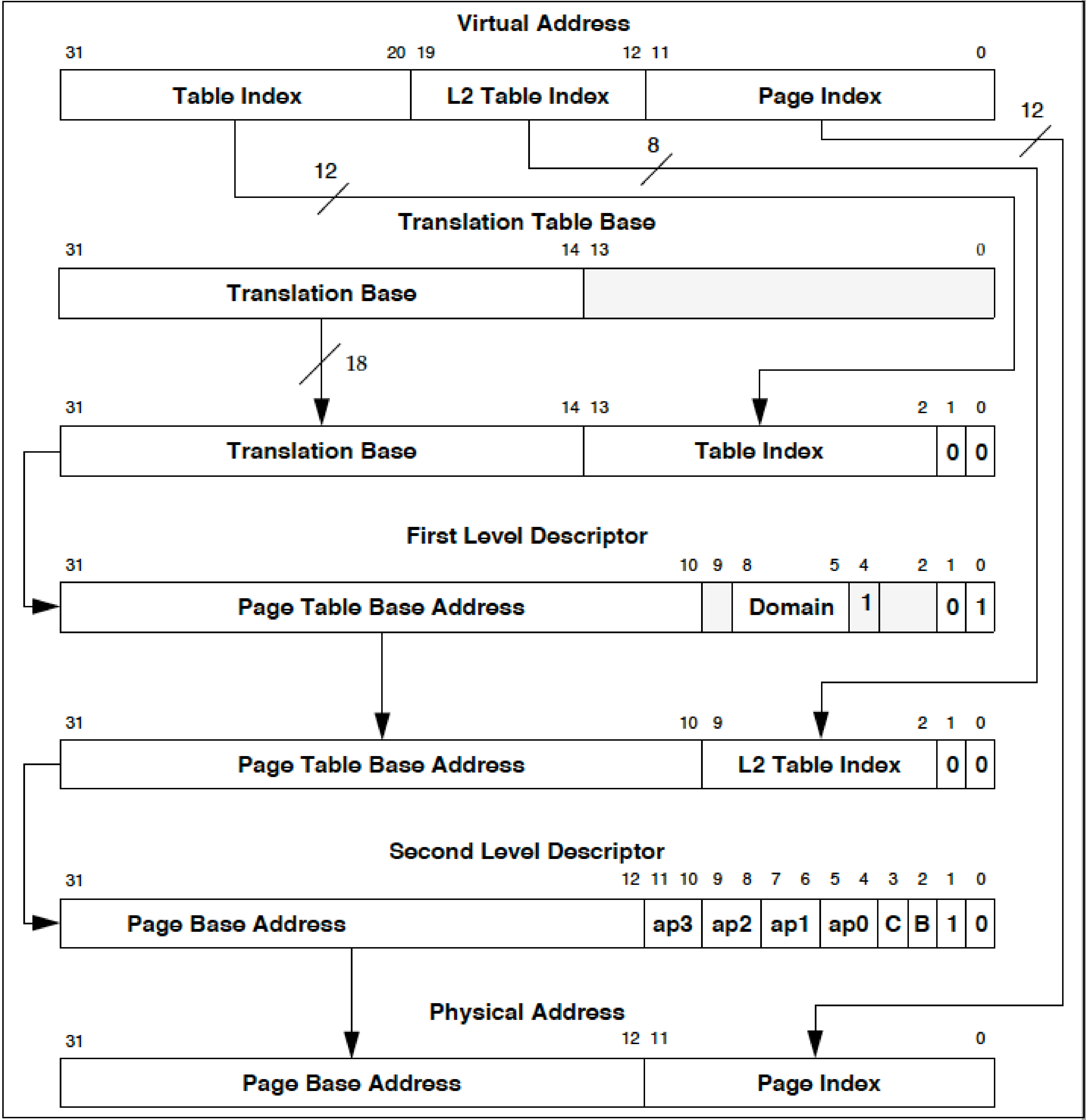


Figure 9-7: Small page translation

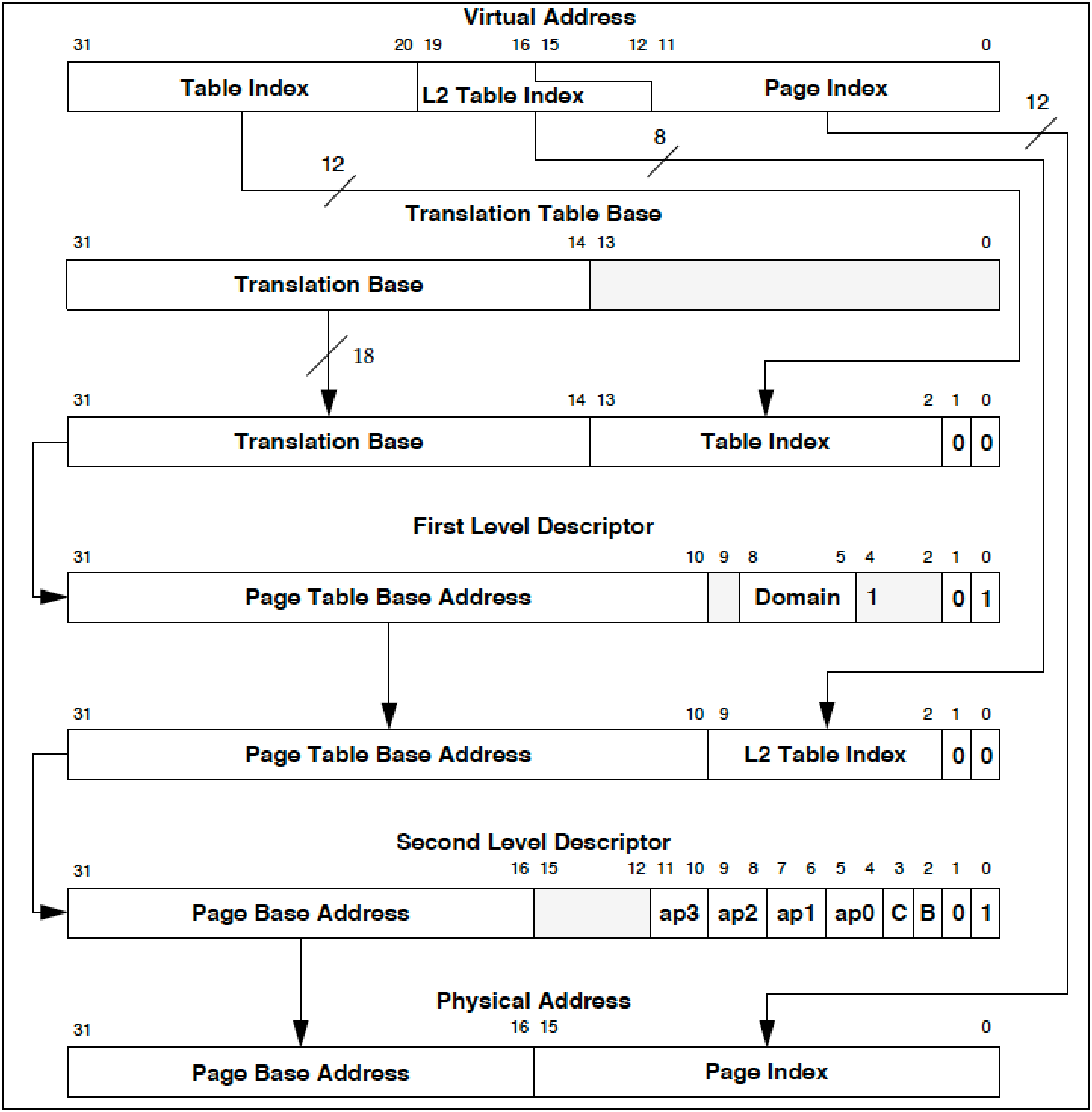


Figure 9-8: Large page translation

8.11 Cacheable and Bufferable Status of Memory Regions

For first level translation table descriptor for each Section, and the second level translation table descriptor for each Large Page, and each Small Page contain two bits—the C-bit and the B-bit—which specify whether the memory in that Section or Page will be cached or buffered, and whether it will be cached with Write-Through or Write-Back behaviour.†

In addition the cache and write buffer behaviour is controlled by the cache enable bit (C-bit) and write buffer enable bit (W-bit) in the CP15 Control Register.

To differentiate the two C bits, we shall add the subscript "tt" to the translation table bits giving us Ctt and Btt, and the subscript "cr" to the control register bits giving us Ccr and Wcr.

The Cache and Write Buffer Configuration is determined by the values of Ctt, Btt, Ccr, Wcr as shown in ***Table 8-5: Cache and write buffer configuration***.

Note † *Write-Back caches are also known as Copy-Back caches.*
"AND" means bitwise AND function.

Ctt AND Ccr	Btt AND Wcr	Cache, Writebuffer & External Abort Operation
0	0	Non-Cached, Non-Buffered (NCNB) • Reads and Writes are not cached. • Writes are not buffered. • Reads and writes may be externally aborted.*
0	1	Non-Cached Buffered (NCB) • Reads and Writes are not cached. • Writes are buffered. • Reads may be externally aborted. • Writes cannot be externally aborted.
1	0	Cached, Write-Through Mode. (WT) • Reads which hit in the cache read the data from the cache and do not perform an external access. • Reads which miss in the cache cause line fills which may be externally aborted. • All writes go off chip and are buffered. • Writes which hit in the cache update the cache. • Writes cannot be externally aborted.
1	1	Cached, Write-Back Mode. (WB) • Reads which hit in the cache read the data from the cache and do not perform an external access. • Reads which miss in the cache cause line fills which may be externally aborted. • Writes which miss in the cache go off-chip and are buffered. • Writes which hit in the cache update the cache and mark the entry as dirty, and do not cause an external access. • Cache write-backs are buffered. • Writes (Cache Write-Misses & Cache Write-Backs) cannot be externally aborted.

Table 8-5: Cache and write buffer configuration (Continued)

8.12 MMU Faults and CPU Aborts

The MMU generates six types of faults:

Alignment Fault
Translation Fault
Domain Fault
Permission Fault
Terminal Fault
Vector Fault

In addition, an external abort may be raised on external data access.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU will abort the access and signal the fault condition to the CPU. The MMU is also capable of retaining status and address information about the abort. The CPU recognises two types of abort: data aborts and prefetch aborts, and these are treated differently by the MMU. See ***8.13 Fault Address and Fault Status Registers (FAR and FSR)***.

If the MMU detects an access violation, it will do so before the external memory access takes place, and it will therefore inhibit the access. External aborts will not necessarily inhibit the external access, as described in the section on external aborts.

8.13 Fault Address and Fault Status Registers (FAR and FSR)

Aborts resulting from data accesses (data aborts) are acted upon by the CPU immediately, and the MMU places an encoded 4 bit value FS[3:0], along with the 4 bit encoded Domain number, in the Fault Status Register (FSR). In addition, the virtual processor address associated with the data abort is latched into the Fault Address Register (FAR). If an access violation simultaneously generates more than one source of abort, they are encoded in the priority given in ***Table 8-6: Priority Encoding of Fault Status*** on page 8-17.

CPU instructions on the other hand are prefetched, so a prefetch abort simply flags the instruction as it enters the instruction pipeline. Only when (and if) the instruction is executed does it cause an abort; an abort is not acted upon if the instruction is not used (i.e. it is branched around). Because instruction prefetch aborts may or may not be acted upon, the MMU status information is not preserved for the resulting CPU abort; for a prefetch abort, the MMU does not update the FSR or FAR.

The sections that follow describe the various access permissions and controls supported by the MMU and detail how these are interpreted to generate faults.

Source		Priority	Domain[3:0]	FAR
highest priority				
Terminal Exception		0b0010	invalid	VA of start of cache line being written-back
Vector Exception		0b0000	invalid	VA of access causing abort
Alignment		0b00x1	invalid	VA of access causing abort
External Abort on Translation	First level	0b1100	invalid	VA of access causing abort
	Second level	0b1110	valid	
Translation	Section	0b0101	invalid	VA of access causing abort
	Page	0b0111	valid	
Domain	Section	0b1001	valid	VA of access causing abort
	Page	0b1011	valid	
Permission	Section	0b1101	valid	VA of access causing abort
	Page	0b1111	valid	
External Abort on linefetch	Section	0b0100	valid	VA of start of cache line being loaded
	Page	0b0110	valid	
External Abort on non-linefetch	Section	0b1000	valid	VA of access causing abort
	Page	0b1010	valid	
lowest priority				

Table 8-6: Priority Encoding of Fault Status

8.14 Domain Access Control

MMU accesses are primarily controlled via domains. There are 16 domains, and each has a 2-bit field to define it. Two basic kinds of users are supported: Clients and Managers. Clients use a domain; Managers control the behaviour of the domain. The domains are defined in the Domain Access Control Register. ***Figure 8-8: Domain Access Control Register format*** on page 8-19 illustrates how the 32 bits of the register are allocated to define the sixteen 2-bit domains.

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Figure 8-8: Domain Access Control Register format

Table 8-7: Interpreting access bits in Domain Access Control Register defines how the bits within each domain are interpreted to specify the access permissions.

Value	**Meaning**	**Notes**
00	No Access	Any access will generate a Domain Fault.
01	Client	Accesses are checked against the access permission bits in the Section or Page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are NOT checked against the access Permission bits so a Permission fault cannot be generated.

Table 8-7: Interpreting access bits in Domain Access Control Register

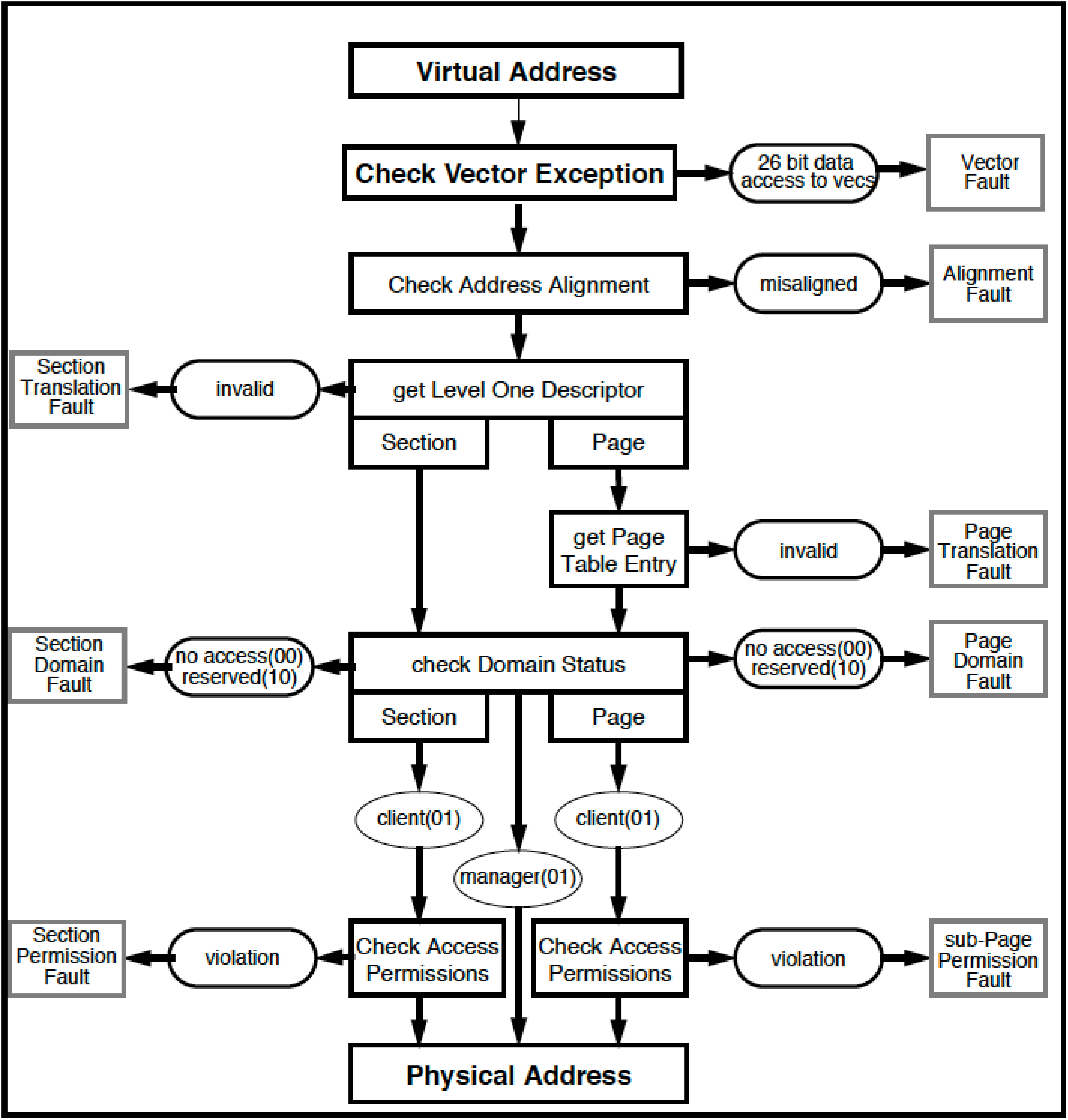


Figure 8-9: Sequence for checking faults

8.15.1 Terminal fault

A terminal fault indicates a system software error in the maintenance of the translation tables in main memory when using the Instruction-Data-Cache in Write-Back mode. It is indicated in theFault Address Register and Fault Status Register to aid debugging system software.

A terminal fault is indicated when a cache-write-back fails to translate the virtual address of the cache line to be written-back into a physical address because the associated translation table walk was aborted by the memory system or returned an invalid Level One or Level Two descriptor [A descriptor is invalid if bits[1:0] have the value “00” or “11”].

System Software must ensure that the cache contains no dirty-data for a page or section before changing the virtual-to-physical mapping of that page or section or disabling the virtual-to-physical mapping of that page or section. A Terminal Fault indicates that system software has failed to do this. When a terminal fault occurs, the data to be written-back from the cache to main memory is irrecoverably lost. A terminal fault is therefore not a reversible fault.

8.15.2 Vector fault

A Vector fault is generated by the MMU if the processor attempts a load or store data access to an address in the range &00000000 and &0000001F inclusive when operating in a 26-bit Mode. Vector faults are never generated for instruction fetches. Vector faults are generated regardless of the setting of the MMU enable bit (M-bit) in the System Control Coprocessor Control Register.

8.15.3 Alignment fault

If Alignment Fault is enabled (bit 1 in Control Register set), the MMU will generate an alignment fault on any data word access the address of which is not word-aligned irrespective of whether the MMU is enabled or not; in other words, if either of virtual address bits [1:0] are not 0. Alignment fault will not be generated on any instruction fetch, nor on any byte access. Note that if the access generates an alignment fault, the access sequence will abort without reference to further permission checks.

8.15.4 Translation fault

There are two types of translation fault: section and page.

1. A Section Translation Fault is generated if the Level One descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0 or both 1.
2. A Page Translation Fault is generated if the Page Table Entry is marked as invalid. This happens if bits[1:0] of the entry are both 0 or both 1.

8.15.5 Domain fault

There are two types of domain fault: section and page. In both cases the Level One descriptor holds the 4-bit Domain field which selects one of the sixteen 2-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions as detailed in ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9. In the case of a section, the domain is checked once the Level One descriptor is returned, and in the case of a page, the domain is checked once the Page Table Entry is returned.

If the specified access is either No Access (00) or Reserved (10) then either a Section Domain Fault or Page Domain Fault occurs.

8.15.6 Permission fault

There are two types of permission fault: section and sub-page. Permission fault is checked at the same time as Domain fault. If the 2-bit domain field returns client (01), then the permission access check is invoked as follows:

section:

If the Level One descriptor defines a section-mapped access, then the AP bits of the descriptor define whether or not the access is allowed according to ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9. Their interpretation is dependent upon the setting of the S bit (Control Register bit 8). If the access is not allowed, then a Section Permission fault is generated.

sub-page:

If the Level One descriptor defines a page-mapped access, then the Level Two descriptor specifies four access permission fields (ap3..ap0) each corresponding to one quarter of the page. Hence for small pages, ap3 is selected by the top 1KB of the page, and ap0 is selected by the bottom 1KB of the page; for large pages, ap3 is selected by the top 16KB of the page, and ap0 is selected by the bottom 16KB of the page. The selected AP bits are then interpreted in exactly the same way as for a section (see ***Table 8-3: Interpreting access permission (AP) Bits*** on page 8-9), the only difference being that the fault generated is a sub-page permission fault.

8.16 External Aborts

In addition to the MMU-generated aborts, ARM810 has an external abort pin which may be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so this pin must be used with great care. The following section describes the restrictions.

The following accesses may be aborted and restarted safely. In the case of a read-lock-write sequence in which the read aborts, the write will not happen.

Reads

Unbuffered writes

Level One descriptor fetch

Level Two descriptor fetch

read-lock-write sequence

Cacheable reads (linefetches)

A linefetch may be safely aborted on any word in the transfer. If an abort occurs during the linefetch then the cache line will be invalidated. If the abort happens on a word that has been requested by the ARM8, the instruction will be aborted, otherwise the cache line will be invalidated but program flow will *not* be interrupted. The line is therefore invalidated under all circumstances.

Buffered writes.

Buffered writes cannot be externally aborted. Therefore, the system should be configured such that it does not do buffered writes to areas of memory which are capable of flagging an external abort.

Writes to Cacheable Regions

Writes to cacheable regions and cache write-backs are performed as buffered writes and cannot be externally aborted. The system design should ensure that writes to cacheable regions are not externally aborted.

8.17 Interaction of the MMU, IDC and Write Buffer

The MMU, IDC, WB and Branch prediction may be enabled/disabled independently. However, in order for the write buffer or the cache to be enabled the MMU must also be enabled. Also, Branch prediction must never be enabled when the cache is disabled. There are no hardware interlocks on these restrictions, so invalid combinations will cause undefined results.

MMU	IDC	WB
off	off	off
on	off	off
on	on	off
on	off	on
on	on	on

Table 8-8: Valid MMU, IDC and Write Buffer combinations

The following procedures must be observed.

To enable the MMU:

1. Program the Translation Table Base and Domain Access Control Registers
2. Program Level 1 and Level 2 page tables as required
3. Enable the MMU by setting bit 0 in the Control Register.

The ARM810 write buffer is provided to improve system performance. It can buffer up to 8 words of data, and 4 independent addresses. It may be enabled or disabled via the W bit (bit 3) in the ARM810 Control Register and the buffer is disabled and flushed on reset. The operation of the write buffer is further controlled by the C and B bits which are stored in the Memory Management Page Tables. For this reason, in order to use the write buffer, the MMU must be enabled. The two functions may however be enabled simultaneously, with a single write to the Control Register. For a write to use the write buffer, both the W bit in the Control Register and either the C or B bit in the corresponding page table must be set.

It is not possible to abort buffered writes externally; the abort pin will be ignored. Areas of memory which may generate aborts should be marked as unbufferable in the MMU page tables.

9.2 Write Buffer Operation

9.2.1 Bufferable write

If the write buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the write buffer at **FCLK** (**MCLK** if running with fastbus extension) speeds and the CPU continues execution. The write buffer then performs the external write in parallel. If however the write buffer is full (either because there are already 8 words of data in the buffer, or because there is no slot for the new address) then the processor is stalled until there is sufficient space in the buffer.

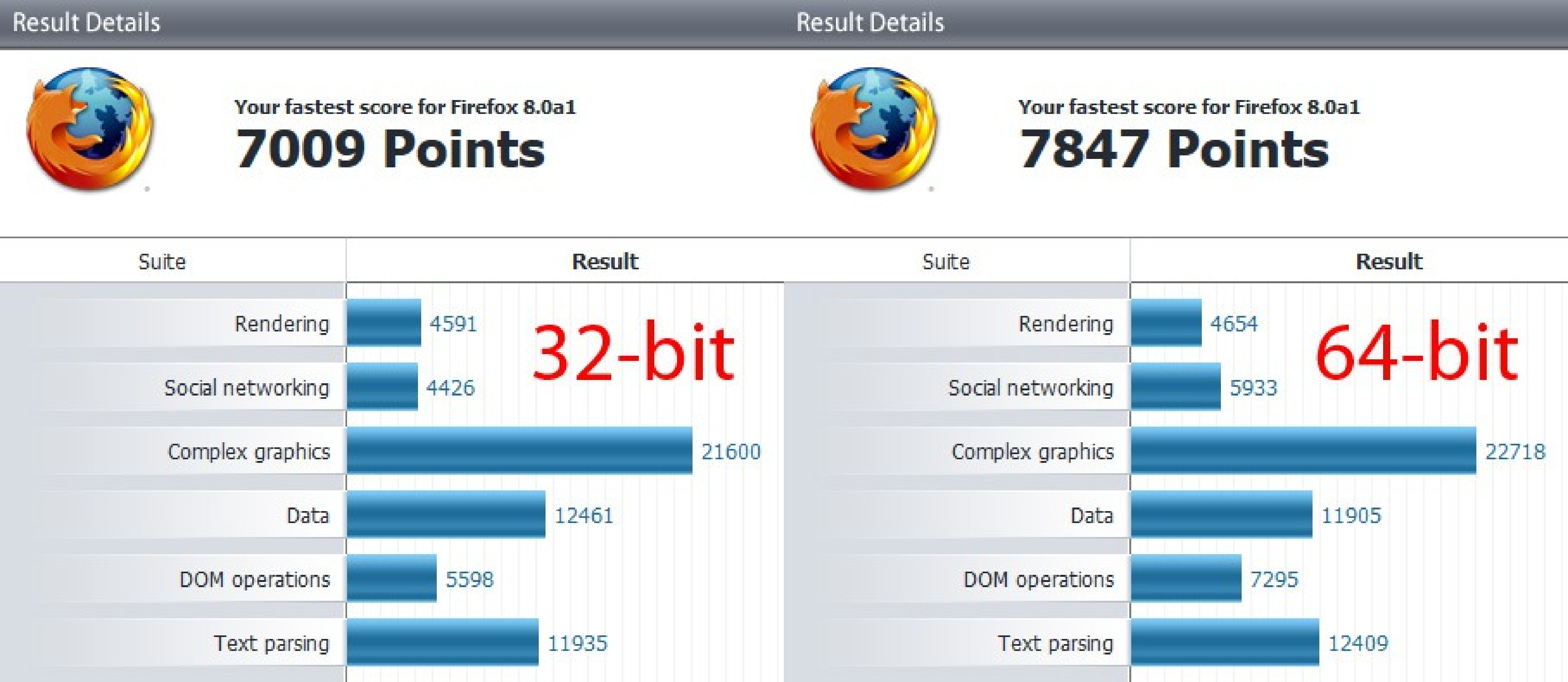
9.2.2 Unbufferable writes

If the write buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write buffer empties and the unbufferable write completes externally, which may require synchronisation and several external clock cycles.

9.2.3 Read-lock-write

The write phase of a read-lock-write sequence is treated as an Unbuffered write, even if it is marked as buffered.

Развитие на микропроцесорната архитектура: Развитие на МП до 64-битова архитектура. Графични процесори. Многоядреност.



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

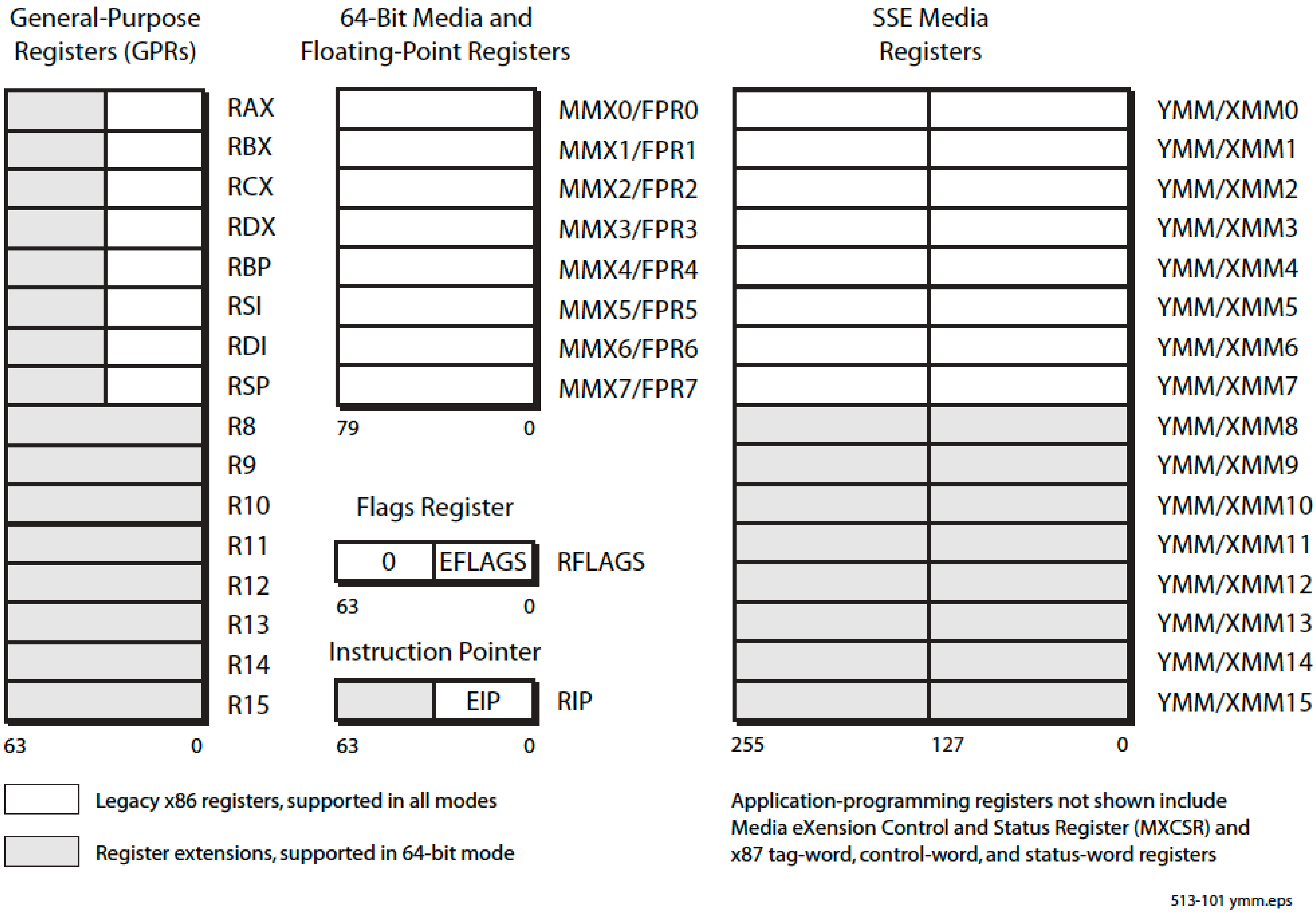
Result Details

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



☐ Legacy x86 registers, supported in all modes

▨ Register extensions, supported in 64-bit mode

Application-programming registers not shown include Media eXension Control and Status Register (MXCSR) and x87 tag-word, control-word, and status-word registers

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.*

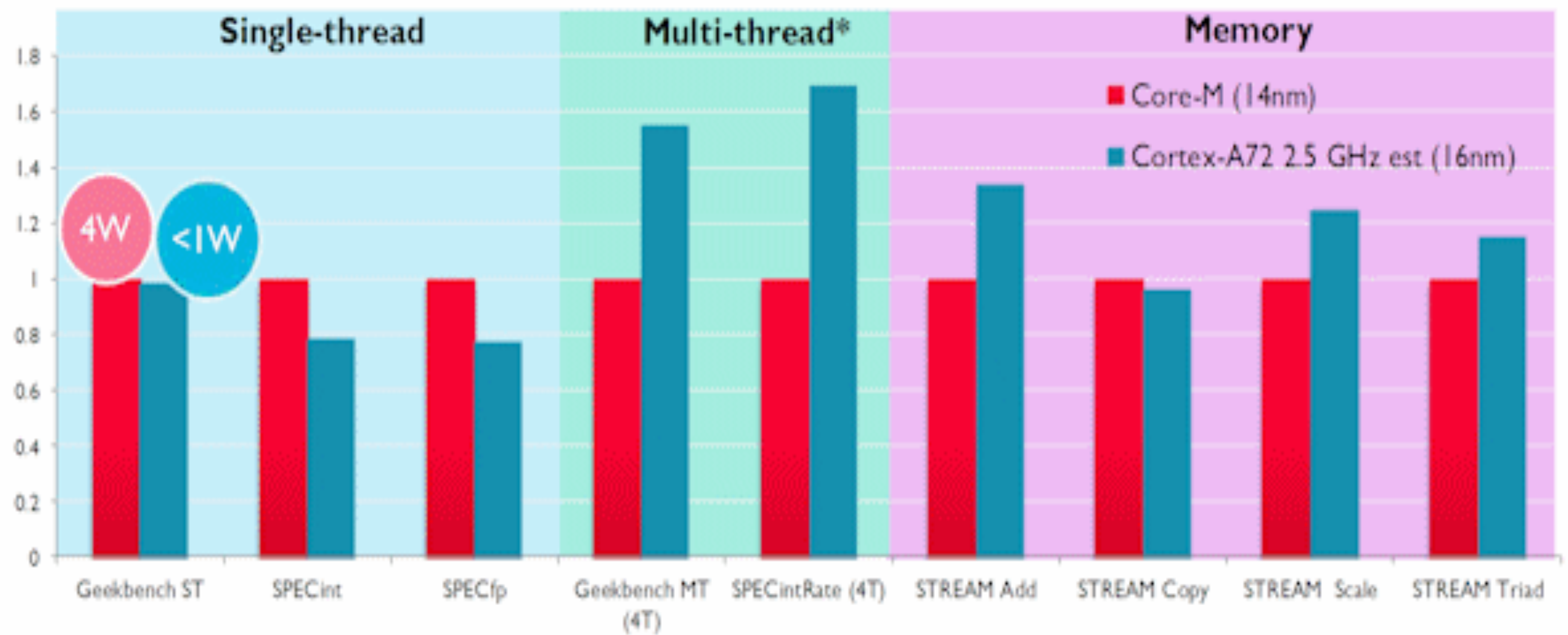
6.1 Overview of ARM’s 64-bit Cortex-A series (6)

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

6.1 Overview of ARM’s 64-bit Cortex-A series (8)

Performance comparison: ARM’s Cortex-A72 vs. Intel’s Core-M [72]



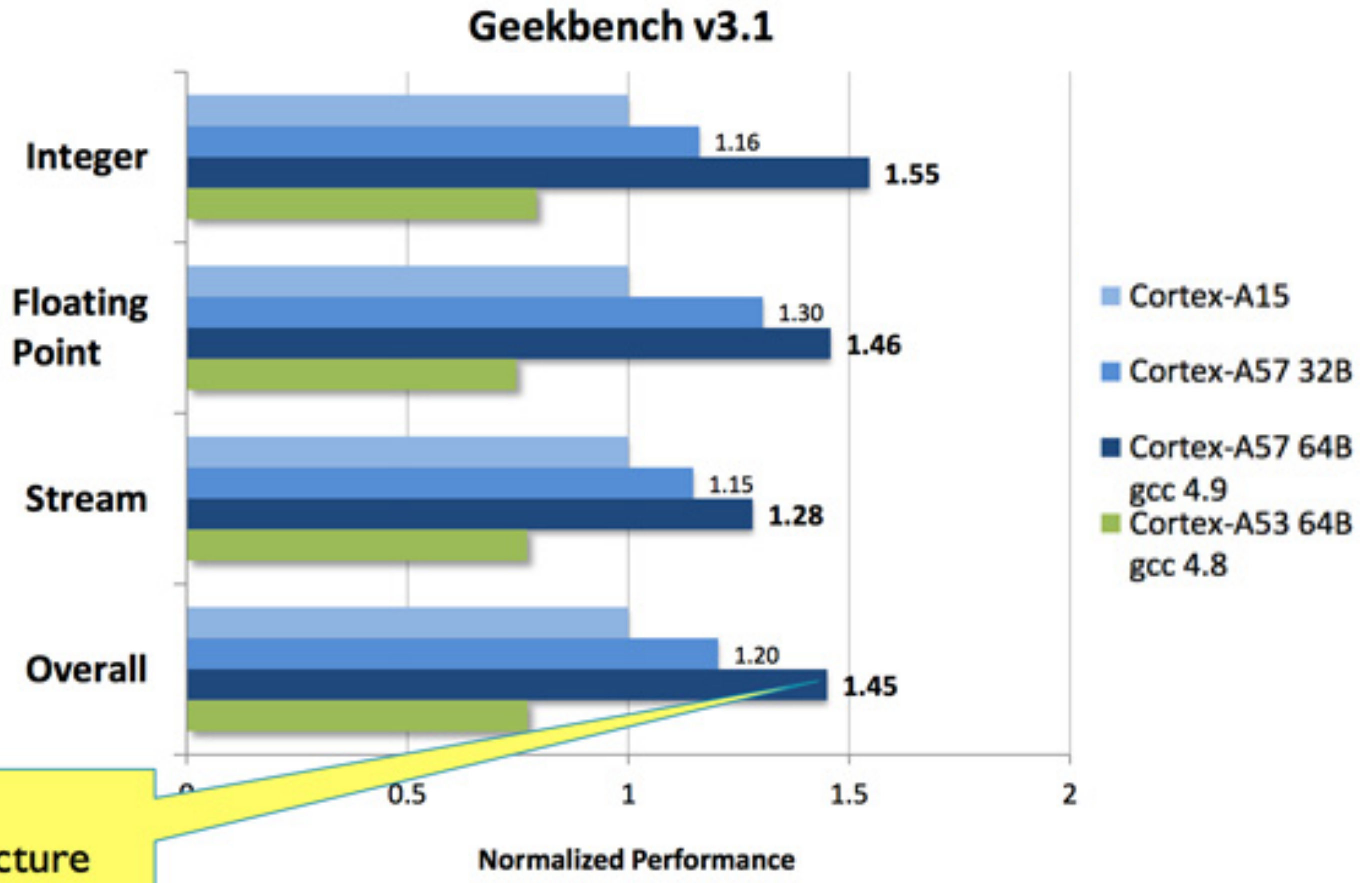
- Intel workloads measured on Dell Venue Pro II. SPEC benchmarks measured using gcc compiler v4.9 with –o3 flag.
- Cortex-A72 measured on RTL with realistic memory system with gcc compiler v4.9 - o3 settings.
- Multi-threaded workloads use 2C4T Core-M CPU and estimated on 4C Cortex-A72 configuration 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

6.2.1 The high performance Cortex-A57 (12)

Cortex-A57/A53 performance - compared to the Cortex-A15 [55]

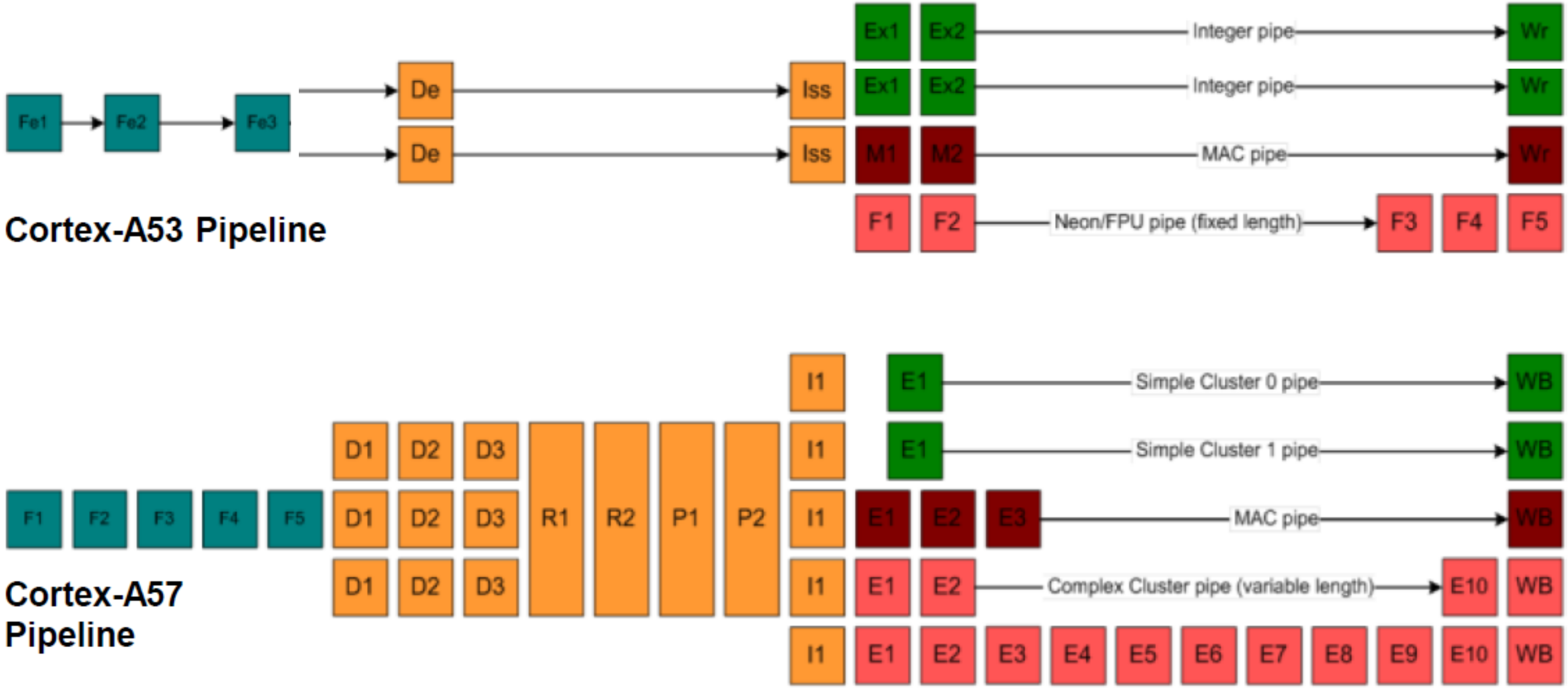


45% increase through incremental microarchitecture improvements

6.2.1 The high performance Cortex-A57 (10)

Contrasting the Cortex-A53 and Cortex-A57 arithmetic pipelines

[Based on 54]

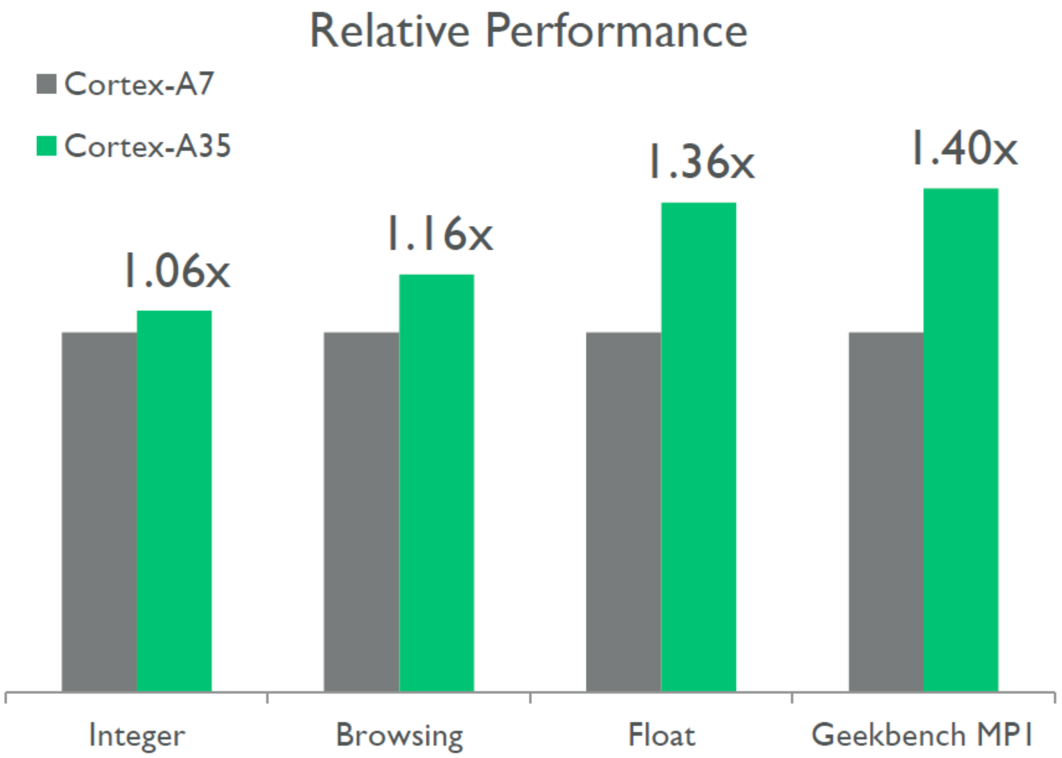


D: Decode
R: Rename
P: Dispatch
I: Issue
E: Execute
WB: Write Back

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)

6.2.5 The low power Cortex-A35 (6)

Relative performance of the Cortex-A35 vs. the Cortex-A7 assuming the same process technology (28 nm) [90]



Higher performance

Lower power

Comparisons assume same process technology and implementation for both processors

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.

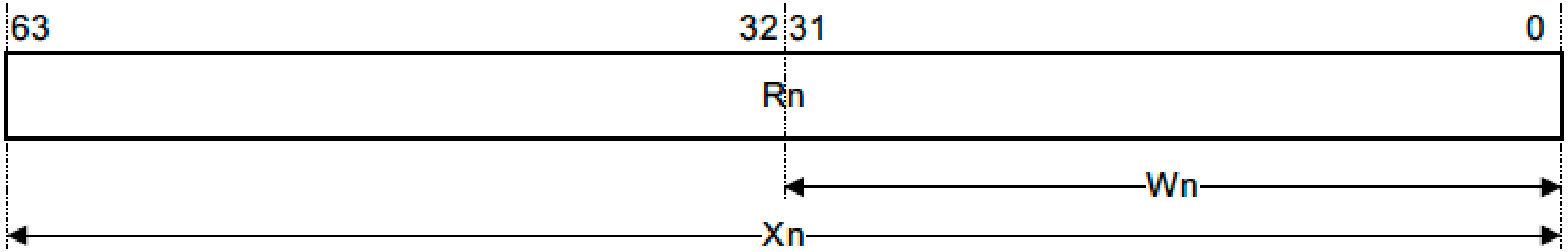


Figure B1-1 General-purpose register naming

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 Register	**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
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 Immediate	**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

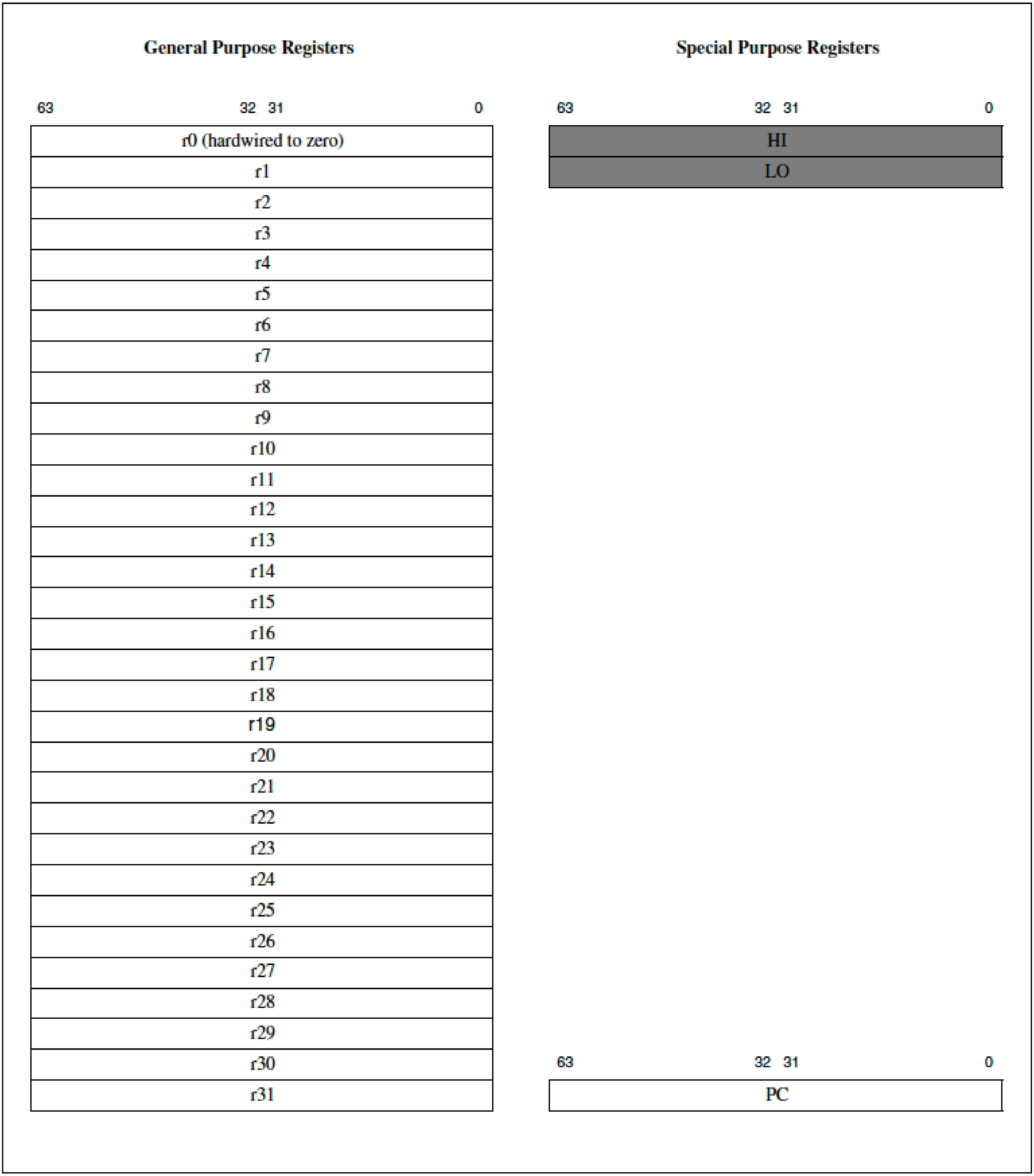
Type	Mnemonic	Instruction
Exclusive	**LDXR**	Load Exclusive register
	LDXRB	Load Exclusive byte
	LDXRH	Load Exclusive halfword
	LDXP	Load Exclusive Pair
	STXR	Store Exclusive register
	STXRB	Store Exclusive byte
	STXRH	Store Exclusive halfword
	STXP	Store Exclusive Pair
Exclusive Aquire/Release	LDAXR	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.

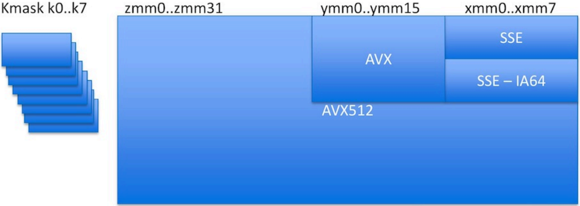
Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Conditional Branch	**B.cond**	Branch conditionally	Conditional Select	CSEL	Conditional select
	CBNZ	Compare and branch if nonzero		CSINC	Conditional select increment
	CBZ	Compare and branch if zero		CSINV	Conditional select inversion
	TBNZ	Test bit 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



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)

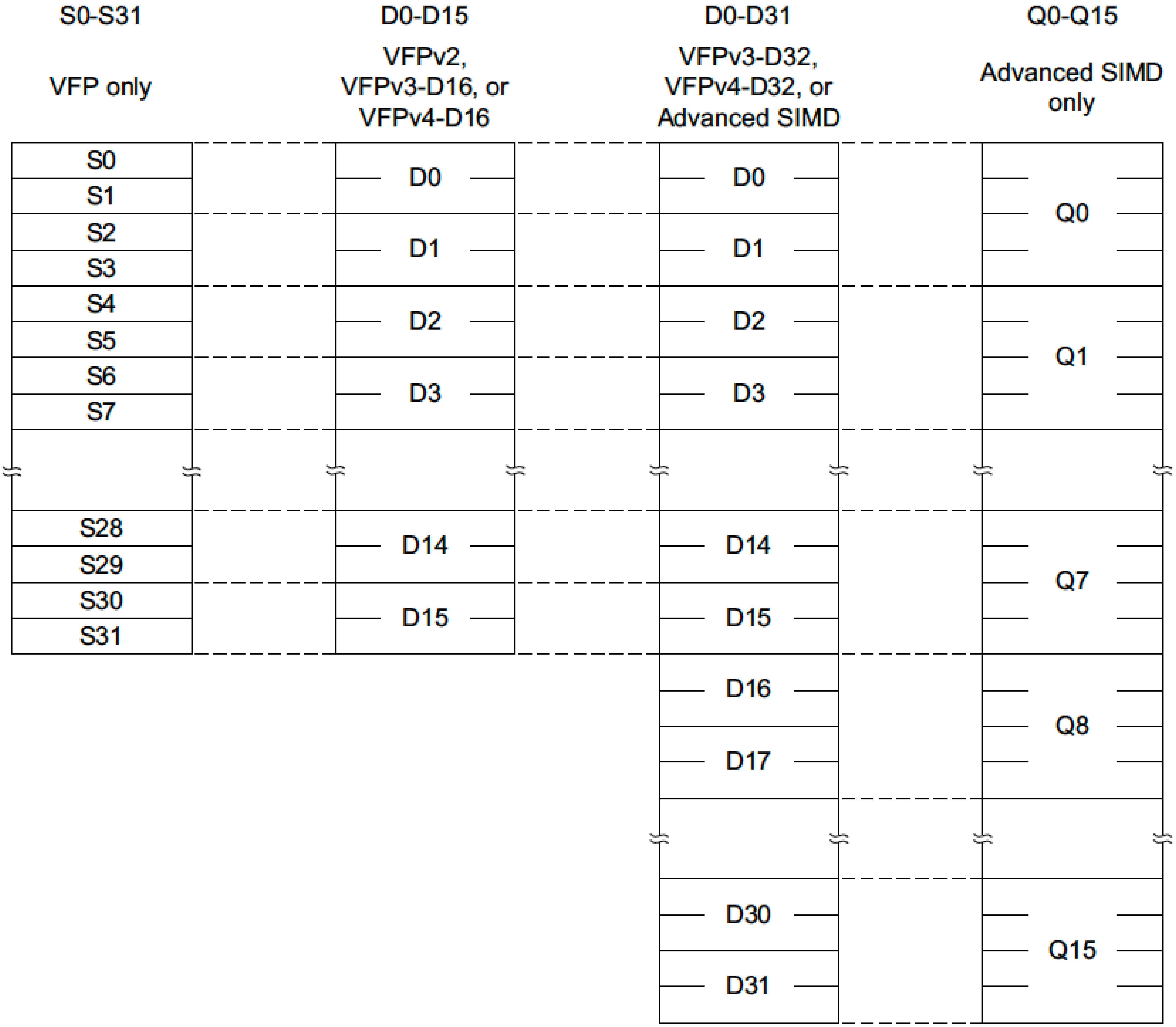
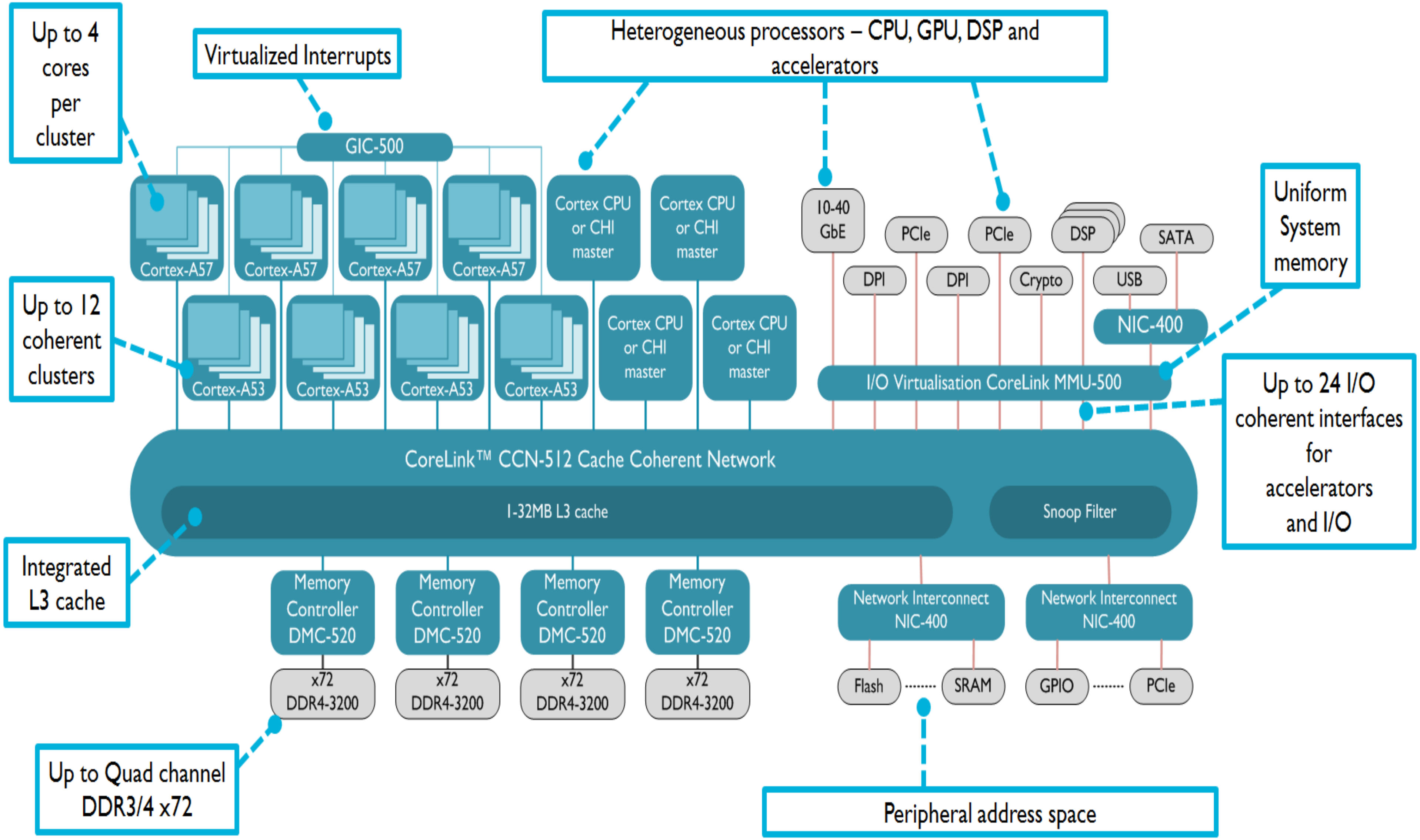


Figure A2-1 Advanced SIMD and Floating-point Extensions register set

6.1 Overview of ARM’s 64-bit Cortex-A series (10)

Up to 48 core server SoC based on the CoreLink CCN512 interconnect [72]



Up to 4 cores per cluster

Virtualized Interrupts

Heterogeneous processors – CPU, GPU, DSP and accelerators

Uniform System memory

Up to 12 coherent clusters

Up to 24 I/O coherent interfaces for accelerators and I/O

Integrated L3 cache

Up to Quad channel DDR3/4 x72

Peripheral address space

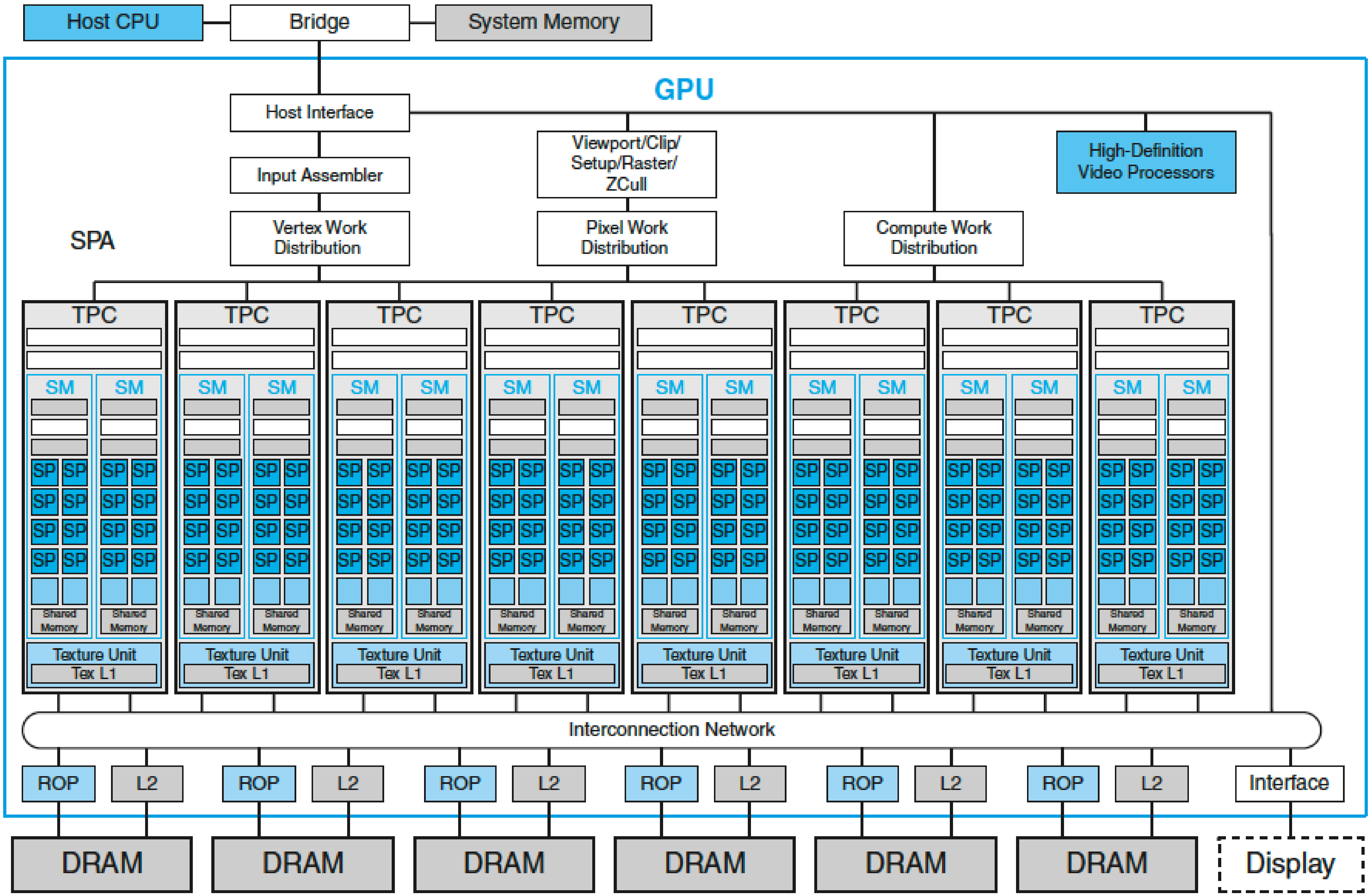
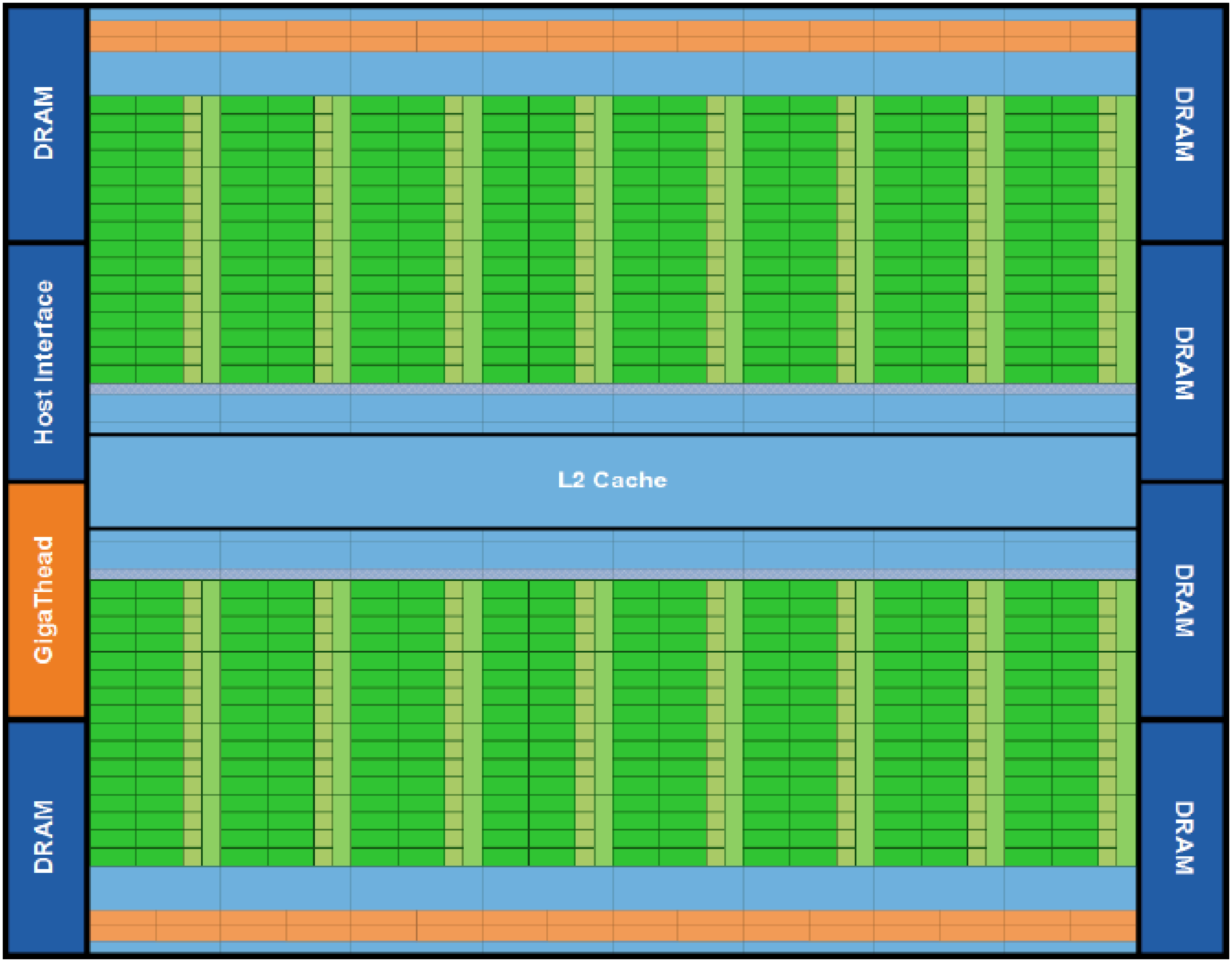


FIGURE B.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection 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.

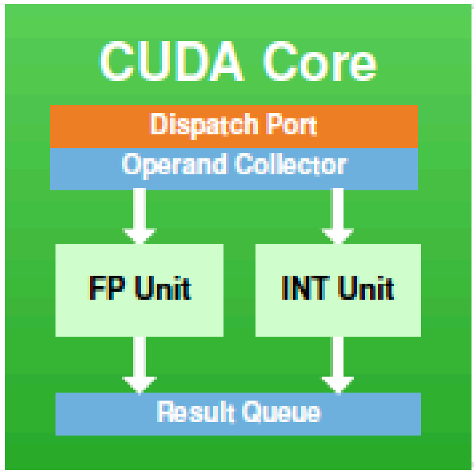


Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

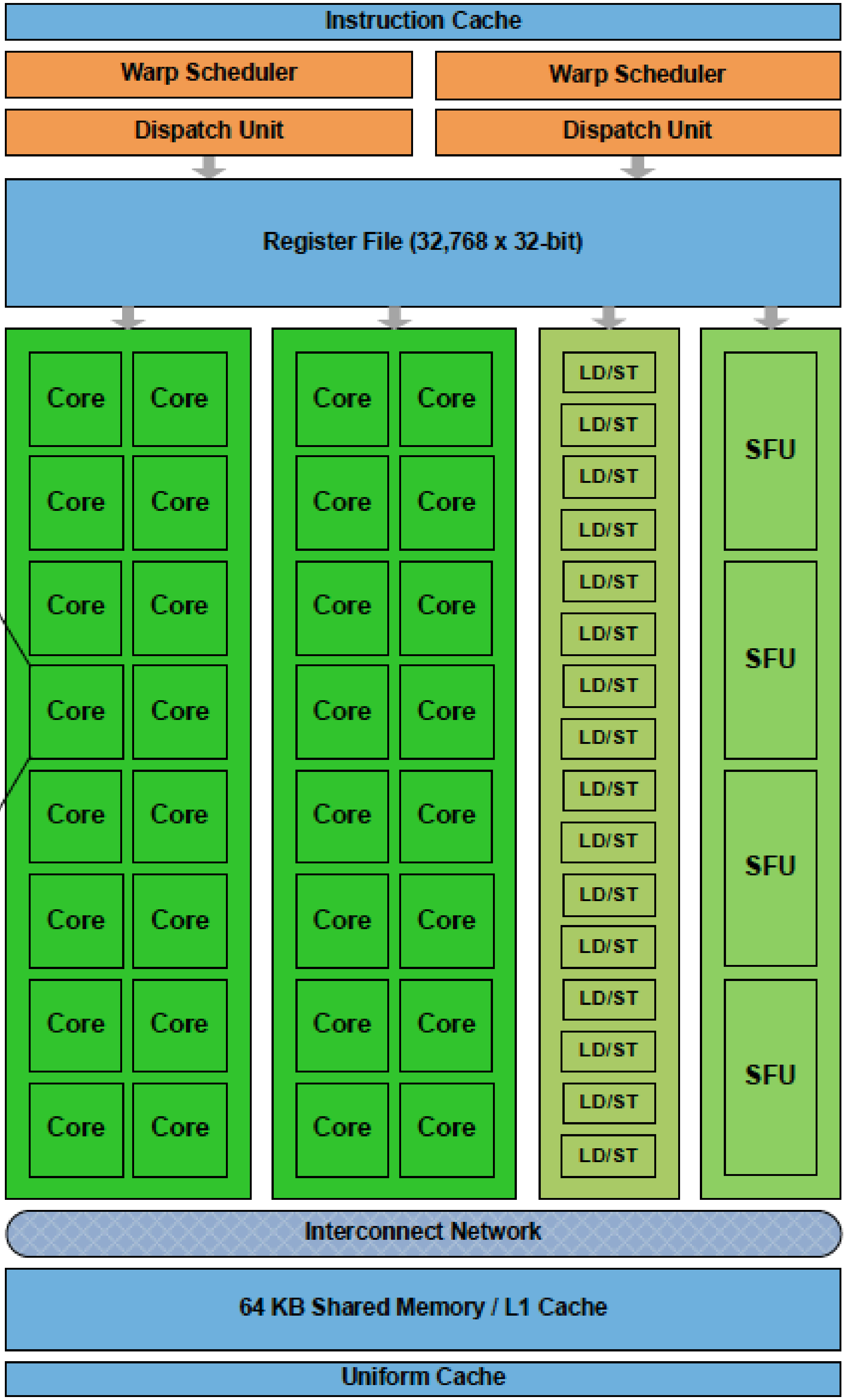
Third Generation Streaming Multiprocessor

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

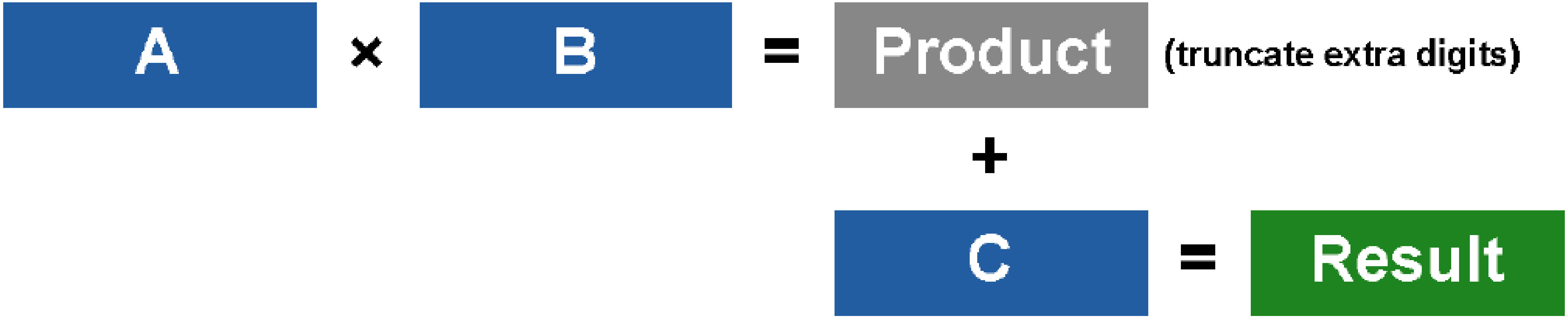


Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 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.

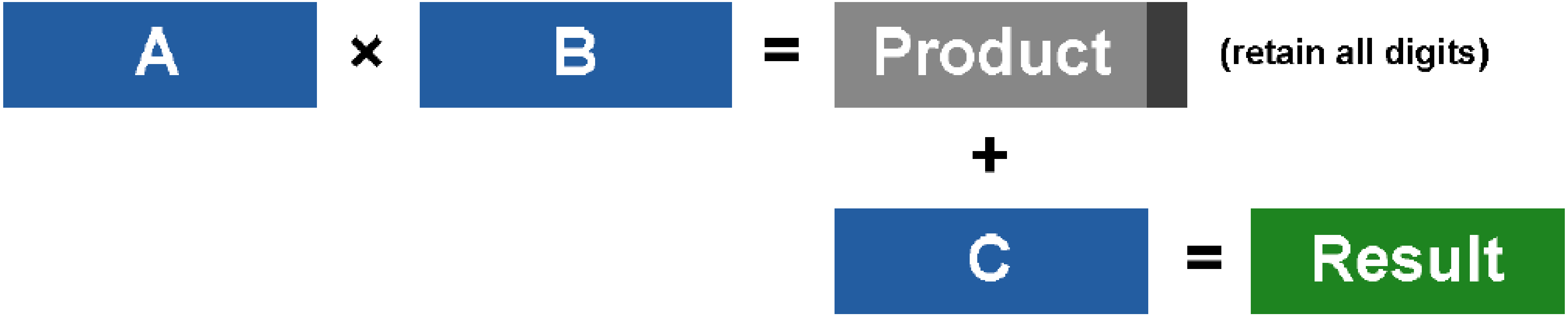


Fermi Streaming Multiprocessor (SM)

Multiply-Add (MAD):

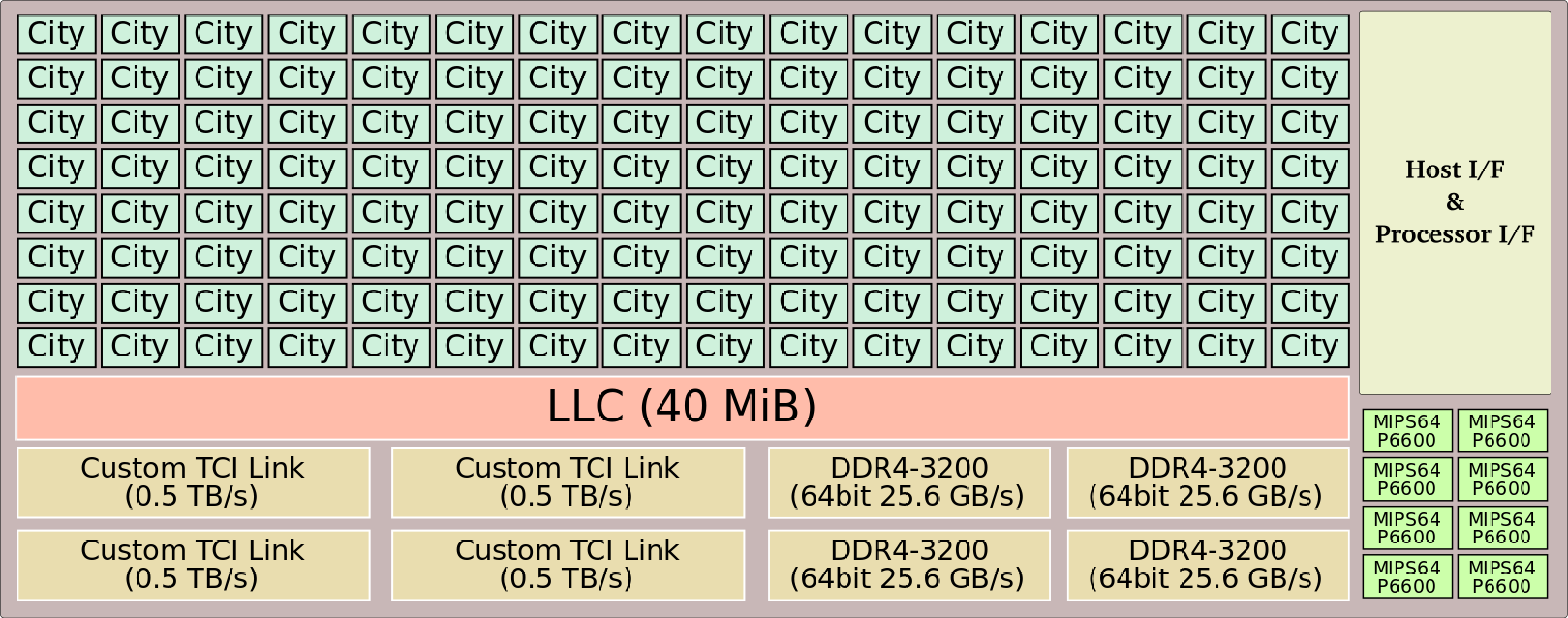


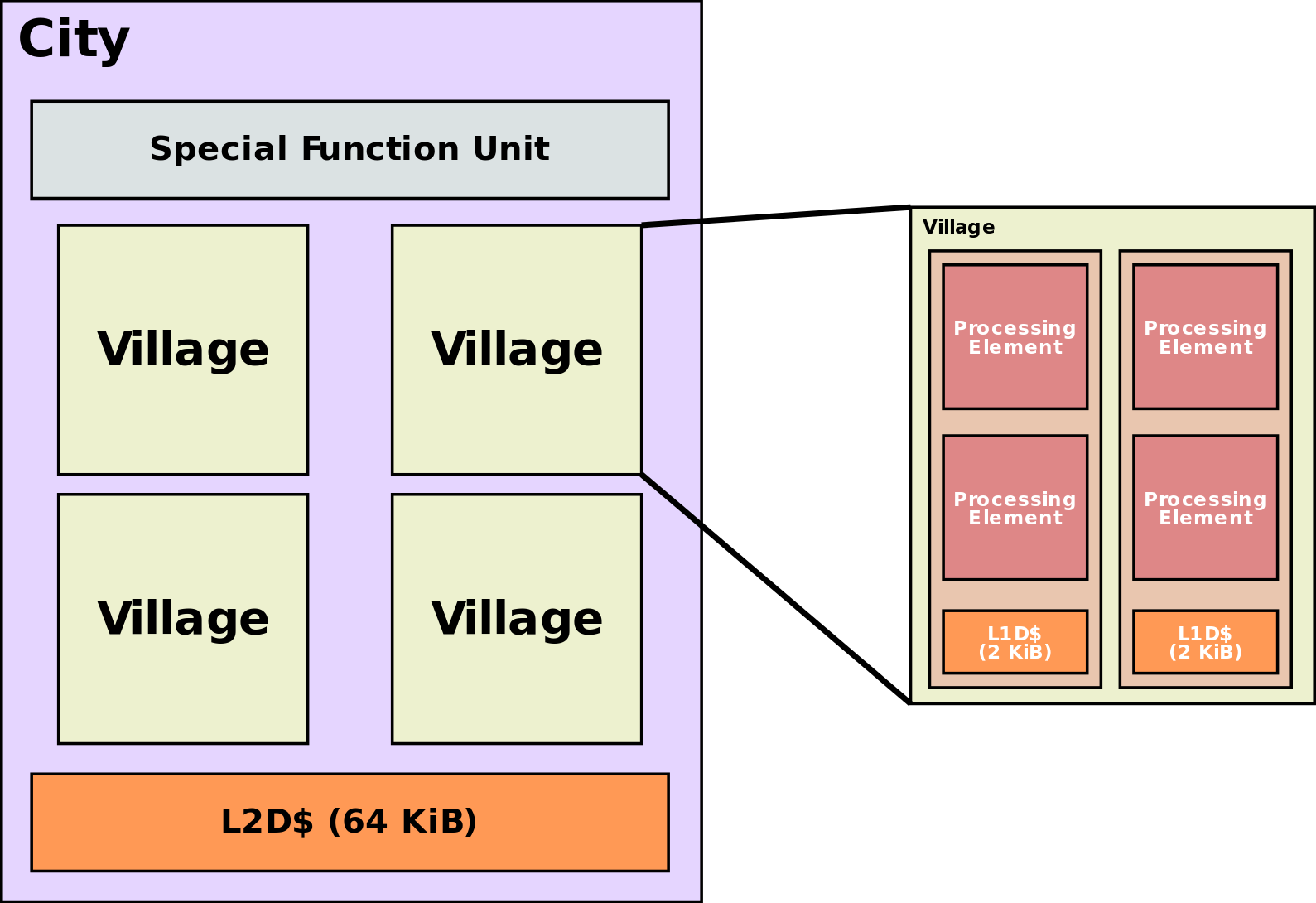
Fused Multiply-Add (FMA)

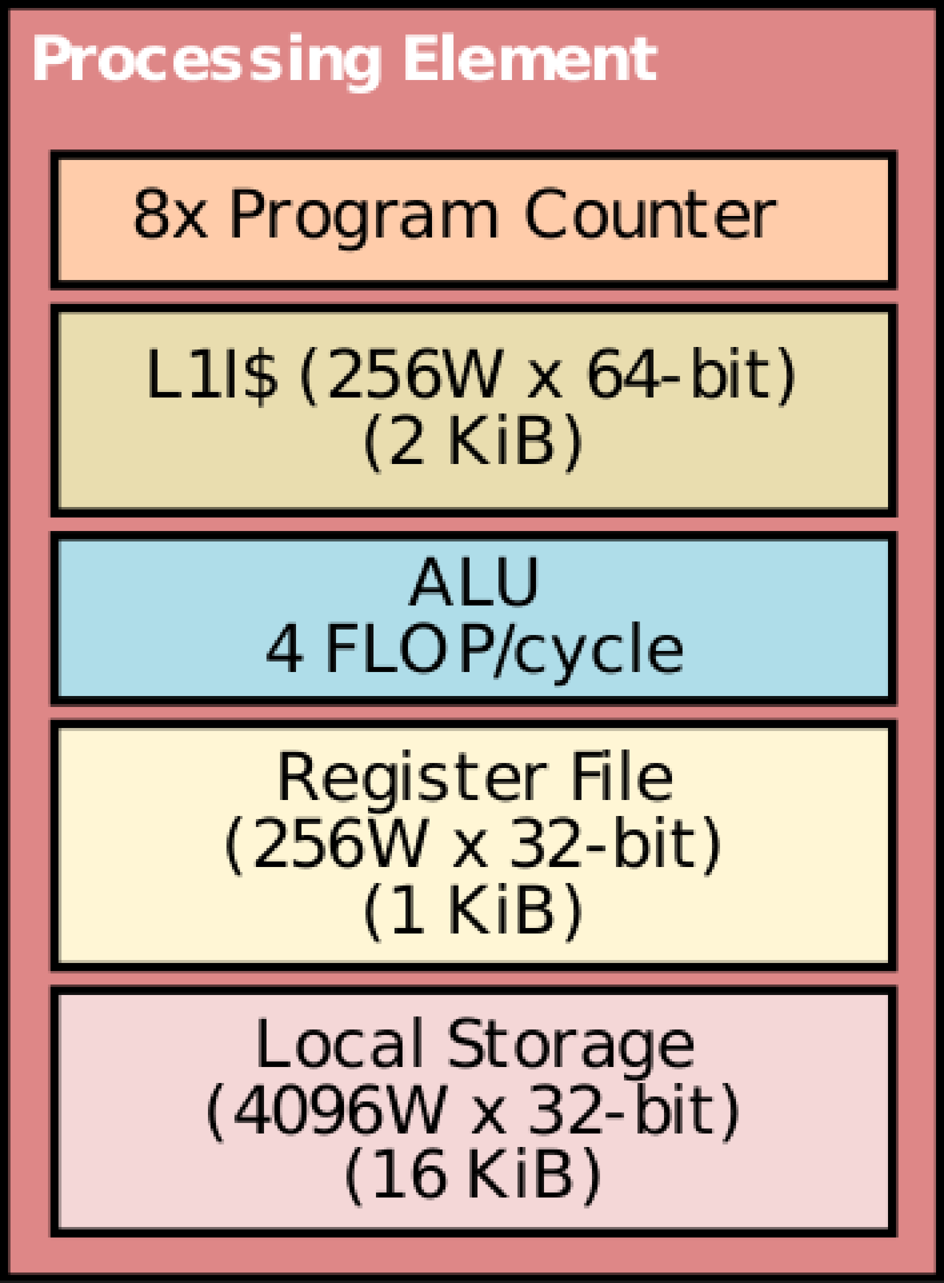


PEZY-SCx Processor Roadmap

	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
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
MIPS64 P6600
Custom TCI Link (0.5 TB/s)
Custom TCI Link (0.5 TB/s)
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)
DDR4-3200 (64bit 25.6 GB/s)
DDR4-3200 (64bit 25.6 GB/s)


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). 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.

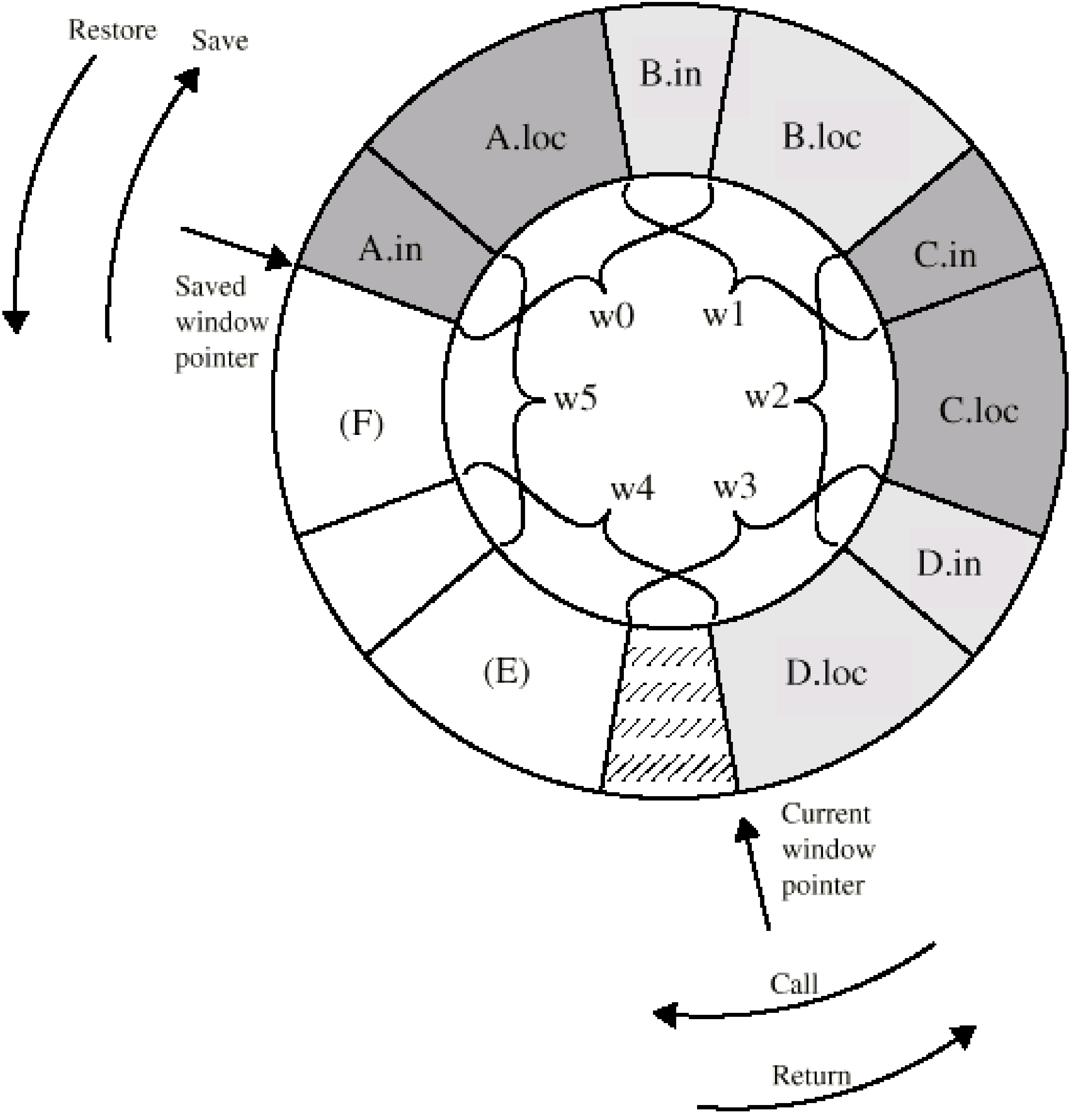
```
; Rc = Ra + Rb, with Rd receiving carry
; Assumes Rc is not the same as Ra
ADDx    Ra, Rb, Rc      ; Rc = Ra + Rb
CMPULT  Ra, Rc, Rd      ; Rd = carry

; 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.

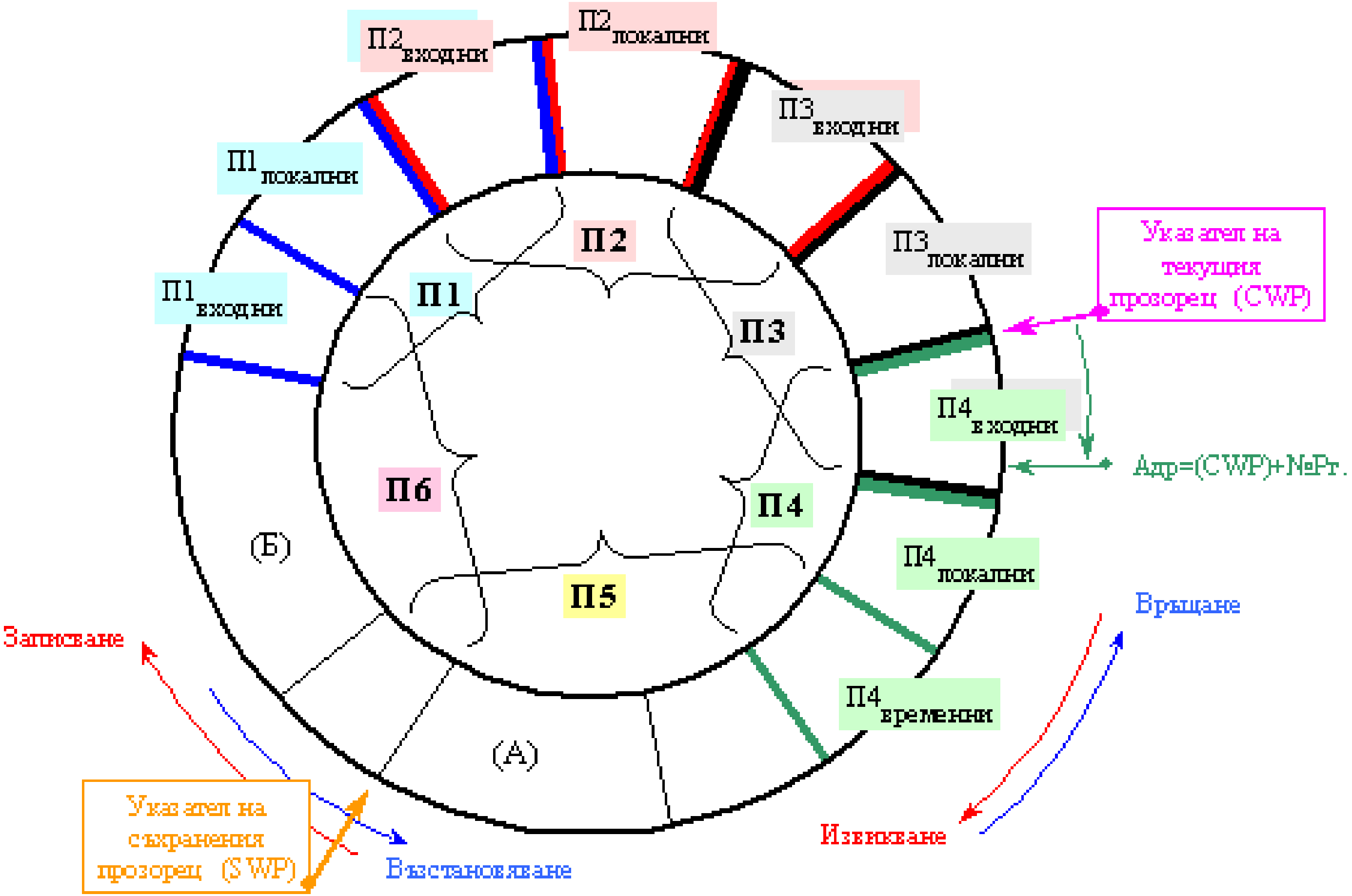
To detect borrow during subtraction, you check whether the subtrahend is greater than the minuend.

```
; 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



Restore
Save
Saved window pointer
A.loc
B.in
B.loc
A.in
C.in
w0
w1
w5
w2
C.loc
(F)
w4
w3
D.in
(E)
D.loc
Current window pointer
Call
Return


П2 входни
П2 локални
П3 входни
П1 локални
П3 локални
Указател на текущия прозорец (CWP)
П2
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)
Възстановяване
Извикване


Sun
©Ⓜ1996,SUN
UltraSPARC II
C6296447 ?.05
PG4.1.1 GS1 USA
STP 1031 LGA
980
-300
A80

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,'$'
```

helloos2.s   Програма „Здравей, свят!“ за i386+ на OS/2 / eCS (NASM)

```
; 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"
```

helloaix.s Програма „Здравей, свят!“ за POWER (as на AIX 7.1-1115)

```
# 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