ОККА, част II: Компютърни архитектури

Този кратък цикъл упражнения по компютърни архитектури си поставя следните цели:

- Разширяване на кръгозора ви на бъдещи компютърни инженери чрез запознаване с общите черти и различията между най-разпространените компютърни архитектури.
- Осъзнаване на важността на производителността и на единството между програмно и апаратно осигуряване за оптимално използване на особеностите на последното.
- Развиване умения за работа с команден ред, анализ на код и търсене на информация.

师傅领进门，修行在个人。 = „Учителят може да отвори вратата, но просветлението зависи само от ученика.“

Всяко занятие се прави с по 1 компютър и има 2 части: измерване на производителност и запознаване с архитектурата на системата му от команди от гледна точка на програмиста.

0. Включете компютъра от бутона „⏻“ на лицевия му панел; изчакайте да се зареди 🪟.
1. Влезте като „*SSH*“ с парола „`linux`“ и изчакайте около минута да се зареди „*KiTTY*“.
2. Появи ли се съобщение за грешка с код 5, натиснете бутона „*OK*“ или клавиша „↵“.
3. Ако работното място е с „*Windows 11*“, вместо т. 1 и 2 влезте като „*student*“ и пуснете „*CMD*“. Трябва да има файл „`.ssh\config`“ със съдържание „`User student№`“, където № е № на работното място (1–9, A–C, P–W); ако го няма, го създайте: „`mkdir .ssh`“, „`cd .ssh`“, „`echo User student№ > config`“. Задайте команда „`ssh` IP-адрес“.
4. Задайте IP-адреса или името в DNS на сървъра, указан от ръководителя на занятието. (Заявките към IP-адреса му се пренасочват към вътрешен такъв, който всяка седмица се настройва да бъде различен – на различен компютър, с различна архитектура.)
5. Натиснете бутона „*Open*“ или клавиша „↵“.
6. На поканата за парола въведете отново „`linux`“ (този път на екрана не излиза *****!)
7. Изпълнете командата „`bash coremark.sh [N]`“, където „*N*“ е броят на логическите ядра на процесора на съответния сървър от таблицата на следващата стр. Ако има само 1 ядро, изпуснете аргумента „*N*“. Квадратните скоби означават незадължителен аргумент и не се пишат. (**Заб.:** Командата ще бъде изпълнена само от първия пуснал я студент и ще предотврати други пускания до завършване на текущото изпълнение.)
8. Студентът, изпълнил програмата, да обяви производителността (в брой итерации в секунда) от редовете, започващи с „*CoreMark 1.0*“ и пропускателната способност на паметта от редовете, започващи с „*Copy*“, закръглени до най-близкото цяло число.
9. Изчислете броя итерации за 1 MHz, разделяйки първото от горните числа (броя итерации в секунда за 1 ядро на процесора) на тактовата му честота в MHz от таблицата.
10. Изчислете броя итерации за 1 *J* от максималната консумирана мощност и ценовата ефективност, разделяйки производителността съответно на мощността и на цената.
11. Обобщете с помощта на ръководителя на занятието получените дотук резултати.
12. Изпълнете командата „`make -B POWER=4`“ за компилиране на тестовите програми.
13. Разгледайте програмата за проверка на подреждането на байтовете в паметта и на дължината на регистрите на микропроцесора с командата „`cat testend.c`“. За да видите какво ще ви покаже тя, изпълнете командата „`env PATH= testend`“.
14. Изпълнете командата „`gdb meantst`“, за да заредите програмата за изчисляване на средно аритметично на елементите на масив в „*gdb*“ (средство за анализ на код).
15. В „*gdb*“ изпълнете командата „`start`“ и натиснете „*Ctrl*–*X*“ и след това „*s*“, за да преминете в режим на управление с единично натискане на клавиши (без „↵“).
16. Изпълнете постъпково програмата с натискане на клавиша „*n*“. Когато стигнете до функцията „*meanval*“, вместо „*n*“ натиснете „*s*“ и проследете действието на всяка команда, намирайки я в приложените справочни таблици. Обърнете внимание на програмния модел, адресацията на паметта, условните преходи и обработката на преноса и на битовите полета. Обсъдете всичко това с ръководителя на занятието.
17. Завършете програмата с натискане на „*c*“. Излезте от „*gdb*“ и сървъра с двукратно натискане на „*Ctrl*–*D*“. Изключете компютъра от бутона „⏻“ на лицевия му панел.

Технически данни на използваните сървъри

Модел	**ThinkCentre M700 Tiny**	**Raspberry Pi 5**	**Mac mini G4**	**Sun Fire T1000**	**EdgeRouter 4**	**BeagleV Ahead**
Архитектура	IA-32/AMD64	ARM	POWER[PC]	SPARC	MIPS	RISC-V
Тип архитектура	CISC	RISC	RISC	RISC	RISC	RISC
Централен процесор	i3-6100T	BCM2712	MPC7447A	UltraSPARC T1	CN7130	TH1520
Тип на централния процесор	МП	ЕЧС	МП	МП	ЕЧС	ЕЧС
Брой физически ядра	2	4	1	6	4	4
Брой логически ядра	4	4	1	24	4	4
Вътрешна тактова честота на ядрата [MHz]	3200	2400	1500	1000	1000	1850
Разделителна способност на процеса [nm]	14	16	130	90	28	12
Дължина на регистрите за обща употреба [bit]	64	64	32	64	64	64
Подредба на байтовете	Little-Endian	Little-Endian	Big-Endian	Big-Endian	Big-Endian	Little-Endian
Конфигурация на паметта	1G × 128	2G × 32	128M × 64	128M × 256	256M × 32	512M × 64
Тип на паметта	DDR4	DDR4	DDR	DDR2	DDR3	DDR4
Коефициент на умножение	8	8	2	4	8	8
Тактова честота на паметта [MHz]	267	533	167	100	133	267
Теоретична пропускателна способност [GB/s]	34,1	17,1	2,7	12,8	4,3	17,1
Пропускателна способност (*stream v5.1*) [GB/s]						
Брой итерации в секунда (*CoreMark 1.0*)						
Брой итерации за 1 MHz						
Енергийна ефективност [брой итерации за 1 J]						
Ценова ефективност [*CoreMark* / $]						
Номинално захранващо напрежение [V]	20 ⎓	5 ⎓	18,5 ⎓	230 ∿	230 ∿	5 ⎓
Максимална консумирана мощност [W]	65	25	85	220	13	10
Габаритни размери [mm]	179×183×35	99×70×33	165×165×51	467×425×43	229×136×31	97×61×20
Максимално тегло без захранващ адаптер [kg]	1,3	0,162	1,3	10,9	0,8	0,13
Цена на дребно в годината на появяване [$]	540	80	600	3000	200	150
Година на пускане в производство, страна	2016 Унгария	2023 Англия	2005 Китай	2006 Тайланд	2017 Виетнам	2023 Китай

Заб.: Непопълнените графи на горната таблица трябва да се определят по време на занятието (но *в никакъв случай* да не се попълват тук, за да имат и тези след вас какво да правят!).

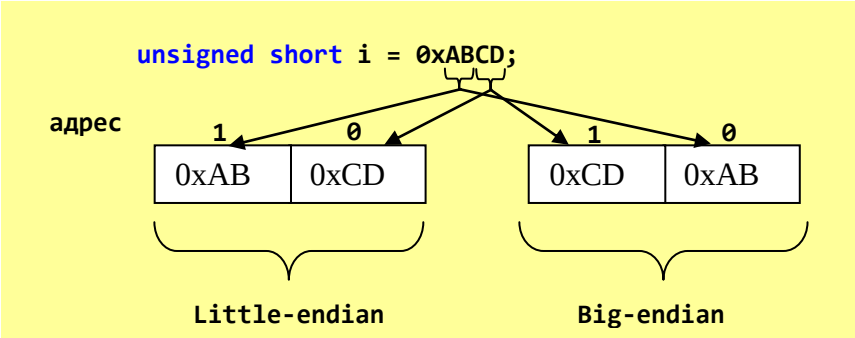
9.8 Big-endian и Little-endian

В зависимост от това как стойността на многобайтовите променливи се разполага в паметта, заделена за тях, различаваме Big-endian и Little-endian компютри.

Little-endian компютрите разполагат стойността на променливата, започвайки от най-младшия байт на паметта, заделена за нея.

Big-endian компютрите разполагат стойността на променливата, започвайки от най-старшия байт на паметта, заделена за нея.

Фиг.38 показва нагледно това.



Фиг. 38 Big-endina и Little-endian

Някои компютри позволяват това да се конфигурира.

testend.c:

```
#include <stdint.h>
#include <stdio.h>

int main()
{
        static union {
                int16_t n;
                char c[3];      /* "AB" */
        } u1 = {0x4142};
        static union {
                int32_t n;
                char c[5];      /* "ABCD" */
        } u2 = {0x41424344};
        static union {
                int64_t n;
                char c[9];      /* "ABCDEFGH" */
        } u3 = {0x4142434445464748LL};
#if __WORDSIZE == 64
        static union {
                __int128_t n;
                char c[17];     /* "ABCDEFGHIJKLMNOP" */
        } u4;
        u4.n = (__int128_t)0x4142434445464748LL << 64 | 0x494A4B4C4D4E4F50LL;
#endif

        puts(u1.c);
        puts(u2.c);
        puts(u3.c);
#if __WORDSIZE == 64
        puts(u4.c);
#endif
        return 0;
}
```

---

meanval.S:

```
.globl  meanval,_meanval
meanval:
_meanval:

#ifdef __i386
        pushl   %ebx
        xorl    %ebx,%ebx
        xorl    %edx,%edx
        movl    $1 << POWER,%ecx
L1:     addl    -4(%eax,%ecx,4),%ebx
        adcl    $0,%edx
        loop    L1
        shrdl   $POWER,%edx,%ebx
        xchgl   %ebx,%eax
        popl    %ebx
        retl
#elif defined __arm__
        mov     r1,#0
        mov     r2,#0
        mov     r3,#1 << POWER
L1:     ldr     ip,[r0],#4
        adds    r1,ip
        adc     r2,#0
        subs    r3,#1
        bne     L1
        mov     r0,r1,lsr # POWER
        orr     r0,r2,lsl #32 - POWER
        bx      lr
#elif defined __powerpc__
        li      %r4,0
        li      %r5,0
        li      %r0,1 << POWER
        mtctr   %r0
        addi    %r3,%r3,-4
L1:     lwzu    %r0,4(%r3)
        addc    %r4,%r4,%r0
        addze   %r5,%r5
        bdnz    L1
        rlwinm  %r3,%r4,32 - POWER,POWER,31
        rlwimi  %r3,%r5,32 - POWER,0,POWER - 1
        blr
#elif defined __sparc
        clr     %o1
        clr     %o2
        mov     1 << POWER,%o3
L1:     ld      [%o0],%o4
        addcc   %o1,%o4,%o1
        addx    %o2,%g0,%o2
        deccc   %o3
        bne     L1
        add     %o0,4,%o0
        srl     %o1,POWER,%o1
        sll     %o2,32 - POWER,%o2
        or      %o1,%o2,%o0
        retl
        nop
#elif defined __mips__
#  include <regdef.h>
.ent    meanval
        li      a1,0
        li      a2,0
        addiu   a3,a0,1 << (POWER + 2)
L1:     lw      t0,(a0)
        addiu   a0,a0,4
        addu    a1,a1,t0
        sltu    t1,a1,t0
        addu    a2,t1,a2
        bne     a0,a3,L1
        ext     v0,a1,POWER,32 - POWER
        ins     v0,a2,32 - POWER,POWER
        jr      ra
.end    meanval
#elif defined __riscv
        li      a1,0
        li      a2,0
        addi    a3,a0,1 << (POWER + 2)
L1:     lw      t0,(a0)
        addi    a0,a0,4
        add     a1,a1,t0
        sltu    t1,a1,t0
        add     a2,t1,a2
        bne     a0,a3,L1
        srli    a1,a1,POWER
        slli    a2,a2,32 - POWER
        or      a0,a1,a2
        ret
#else
#       error   <Αρχιτεκτονική?!>
#endif
```

Програмни особености на изследваните архитектури

Архитектура	**IA-32**	**ARM** (32-битова)	**POWER[PC]**	**SPARC**	**MIPS**	**RISC-V**
Тип	CISC	RISC	RISC	RISC	RISC	RISC
Целочислени регистри	8	16	32	32	32	32
Регистрови прозорци	няма	няма	няма	8	няма	няма
Регистър с нулево съдържание	няма	няма	няма	%g0	$0	x0
Методи за адресация на паметта	6	7	2	2	1	1
Мащабиране на индекса	от 1 до 2^3	от 1 до 2^{31}	няма	няма	няма	няма
Подразбираща се адресация	да	не	не	не	не	не
Пре- и постиндексна адресация	само с подразбираща се адресация	да	само преиндексна	не	не	не
Трикомпонентна адресация	да	не	не	не	не	не
Команди, адресиращи паметта	почти всички целочислени и мн. др.	само тия за зареждане и съхранение	само тия за зареждане и съхранение	само тия за зареждане и съхранение	само тия за зареждане и съхранение	само тия за зареждане и съхранение
Целочислени аритметично-логически команди	едно- и двуадресни с изключение на IMUL	три- и четириадресни	триадресни	триадресни	триадресни	триадресни
Команди за целочислено сравнение	общи за числа с и без знак	общи за числа с и без знак	отделни за числа с и без знак	общи за числа с и без знак	отделни за числа с и без знак; записват резултата в регистър	отделни за числа с и без знак
Условни преходи	с флагове O, S, Z, C	с флагове N, Z, C, V	с флагове <, >, =, SO	с флагове N, Z, V, C	при < 0, > 0, ≤ 0, ≥ 0, Rs = Rt, Rs ≠ Rt	при Rs < Rt, Rs ≥ Rt, Rs = Rt, Rs ≠ Rt
Особености в преходите; СК=следваща команда, ако няма преход	няма	няма	„намеци“ за предиктора на преходи с +/-	задържани преходи и анулиране на СК с ,a	задържани преходи, „намеци“ и анулиране на СК (суфикс -L)	общи команди за сравнение и преход
Регистри за код на условие (РКУ)	1	1	8	1	няма	няма
Промяна на флаговете на РКУ	твърдо определена	по желание	по желание – на РКУ, CA и/или SO	по желание	няма РКУ	няма РКУ
Условни команди	само преходите	почти всички	само преходите	само преходите	само преходите	само преходите
Специализиран регистър-брояч	за някои команди – ECX	няма	CTR	няма	няма	няма
Обработка на преноса	с флаг C, вкл. при ротация	с флаг C, вкл. при ротация	с флаг CA, без изместване и ротация	с флаг C, няма команди за ротация	като пренос се използва регистър	като пренос се използва регистър
Команди за битови полета	BEXTR, PDEP, PEXT (2013+)	BFI, SBFX/UBFX (2003+)	rlwimi, rlwinm	няма	EXT, INS (2003+)	много, в незадължително разширение (β)

Компютър „M700 Tiny“ с архитектура „IA-32/AMD64“ (IA = Intel Architecture)

IA32 Instruction List (Short Form)

Description of Operands

r8	8-bit general purpose register
r16	16-bit general purpose register
r32	16-bit general purpose register
EDX:EAX	64-bit integer number, EDX – more significant part, EAX – less significant part

imm8	immediate 8-bit value from –128 to 127
imm16	immediate 16-bit value from –32768 to +32767
imm32	immediate 32-bit value from –2147483648 to +2147483647

r/m8	8-bit general purpose register or memory location
r/m16	16-bit general purpose register or memory location
r/m32	32-bit general purpose register or memory location

m	16-bit or 32-bit memory location
m8	8-bit memory location
m16	16-bit memory location
m32	32-bit memory location
m64	64-bit memory location
m128	128-bit memory location
m*N*byte	*N*-byte memory location

m16:16	a memory location containing a far pointer composed of two 16-bit numbers: segment & offset
m16:32	a memory location containing a far pointer composed of numbers: 16-bit segment & 32-bit offset
m16&16	a memory location containing a data pair: 16&16-bit
m16&32	a memory location containing a data pair: 16&32-bit
m32&32	a memory location containing a data pair: 32&32-bit

moffs8	simple 8-bit memory location, which actual address is given by a simple offset relative to segment base
moffs16	simple 16-bit memory location, which actual address is given by a simple offset relative to segment base
moffs32	simple 32-bit memory location, which actual address is given by a simple offset relative to segment base

Sreg	segment register: CS, DS, SS, ES, FS or GS

rel8	relative address in the range from 128 bytes before to 127 bytes after the end of instruction
rel16	16-bit relative address within the same code segment
rel32	32-bit relative address within the same code segment

ptr16:16	16-bir far pointer in a different code segment
ptr32:32	32-bir far pointer in a different code segment
m32fp	a single-precision floating-point memory location
m64fp	a double-precision floating-point memory location
m80fp	an extended-precision floating-point memory location

m16int	a word integer memory location
m32int	a double-word (dword) integer memory location
m64int	a quad-word (qword) integer memory location

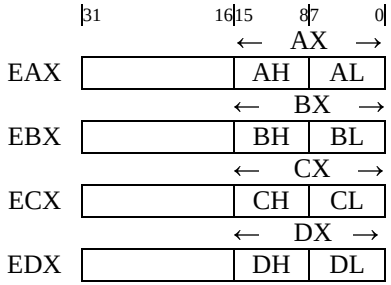
ST	the top element of the FPU register stack
ST(0)	the top element of the FPU register stack
ST(i)	the i-th element of the FPU register stack (i←–0..7)

mm	64-bit MMX register from MM0 to MM7
mm/m32	low order 32 bits of an MMX register or 32-bit memory location
mm/m64	MMX register or 64-bit memory location

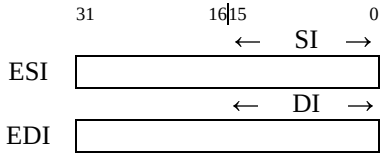
xmm	128-bit XMM register from XMM0 to XMM7
xmm/m32	XMM register or 32-bit memory location
xmm/m64	XMM register or 64-bit memory location
xmm/m128	XMM register or 128-bit memory location

Register Set

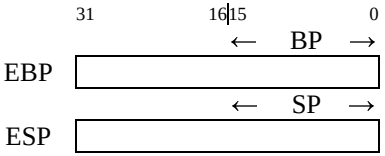
General Purpose Registers



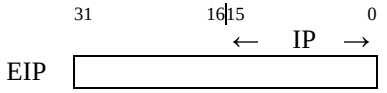
Index Registers



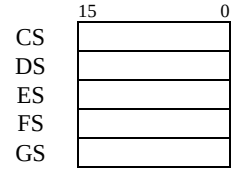
Pointer Registers



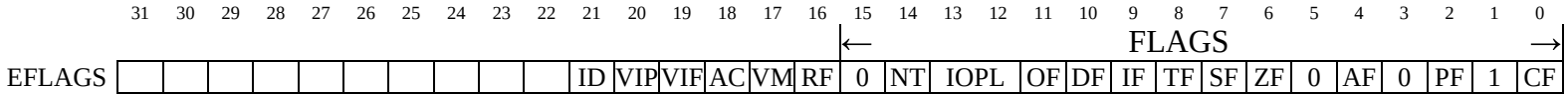
Instruction Pointer



Segment Registers



Flags



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

The "*regparm*" calling convention and IA-32 register preservation requirements

"Using the GNU Compiler Collection", for GCC version 9.2.0 (http://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc.pdf), p. 526:

`regparm (`*`number`*`)`

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

"System V Application Binary Interface", Fourth Edition (http://sco.com/developers/devspecs/abi386-4.pdf), p. 3-11:

All registers on the Intel386 are global and thus visible to both a calling and a called function. Registers `%ebp`, `%ebx`, `%edi`, `%esi`, and `%esp` "belong" to the calling function. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Ibid., p. 3-12:

A function that returns an integral or pointer value places its result in register `%eax`.

x86 Addressing Modes

The addressing mode determines, for an instruction that accesses a memory location, how the address for the memory location is specified.

To summarize, the following diagram (from http://en.wikipedia.org/wiki/X86#Addressing_modes) shows the possible ways an address could be specified:

$$\left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{Bmatrix}\right] + \left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{Bmatrix} * \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix}\right] + [\text{displacement}]$$

Each square bracket in the above diagram indicates an optional part of the address specification. These parts (from left to right) are: A register used as a base address, a register used as an index, a width (or scale) value to multiply the register by, and an displacement (aka offset) which is an integer. The address is computed as the sum of: the base register, the index times the width, and the displacement.

The Intel & AT&T syntax for various addressing modes, depending on which parts of the above diagram are used, is shown in the table below from http://simon.baymoo.org/universe/tools/symset/symset.txt (slightly modified):

Mode	Intel	AT&T
Absolute	MOV EAX, [0100]	movl 0x0100, %eax
Register	MOV EAX, [ESI]	movl (%esi), %eax
Reg + Off	MOV EAX, [EBP-8]	movl -8(%ebp), %eax
R*W + Off	MOV EAX, [EBX*4 + 0100]	movl 0x100(,%ebx,4), %eax
B + R*W + O	MOV EAX, [EDX + EBX*4 + 8]	movl 0x8(%edx,%ebx,4), %eax

Note that, given the definition of a label `x` in the `.data` section of an assembly program, using x to indicate the memory location, as in

```
mov  eax,x  #Intel
```

or

```
mov  x,%eax  #AT&T
```

is just absolute addressing (i.e. using just a displacement), where the assembler essentially replaces the name `x` with the address corresponding to `x`.

CPU Instruction set

Data Transfer Instructions

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

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

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

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

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

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

Binary Arithmetic Instructions

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

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

Decimal Arithmetic Instructions

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

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

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

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

Logical Instructions

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

Shift and Rotate Instructions

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

Shift Arithmetic Right	SAR		Divide signed register by 2. Divide signed register by 2, CL times. Divide signed register by 2, *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← DST / 2 NEXT
Rotate Left	ROL	r/m8 r/m8, CL r/m8, imm8 r/m16 r/m16, CL r/m16, imm8 r/m32 r/m32, CL r/m32, imm8	Rotate register bits left by 1 position. Rotate register bits left by 1 position., CL times. Rotate register bits left by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← RotateLeft (DST) NEXT
Rotate Right	ROR		Rotate register bits right by 1 position. Rotate register bits right by 1 position., CL times. Rotate register bits right by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH DST ← RotateRight (DST) NEXT
Rotate thru Carry Left	RCL		Rotate register bits and CF flag left by 1 position. Rotate register bits and CF flag left by 1 position., CL times. Rotate register bits and CF flag left by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH (DST,CF) ← RotateLeft (DST,CF) NEXT
Rotate thru Carry Right	RCR		Rotate register bits and CF flag right by 1 position. Rotate register bits and CF flag right by 1 position., CL times. Rotate register bits and CF flag right by 1 position., *imm8* times.	FOR i←1 TO COUNT AND 1FH (DST,CF) ← RotateRight (DST,CF) NEXT

Shift Left Double	SHLD	r/m16, r16, imm8
		r/m16, r16, CL
		r/m32, r32, imm8
		r/m32, r32, CL
			Shift first register to left by *imm8/* CL places while shifting bits from *r16* in from the right.	TMP ← DTS2;
				FOR i←1 TO COUNT AND 1FH
				  DST1 ← ShiftLeft (DST1, MSB (TMP));
				  TMP ← ShiftLeft (TMP);
				NEXT

Shift Right Double	SHRD		Shift first register to right by *imm8/* CL places while shifting bits from *r16* in from the left.	TMP ← DTS2;
				FOR i←1 TO COUNT AND 1FH
				  DST1 ← ShiftRight (LSB (TMP), DST1);
				  TMP ← ShiftRight (TMP);
				NEXT

Bit and Byte Instructions

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

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

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

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

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

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

Control Transfer Instructions

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

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

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

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

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

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

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

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

String Instructions

Instruction	Mnemonic	Operands	Description	Symbolic operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Flag Control Instructions

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

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

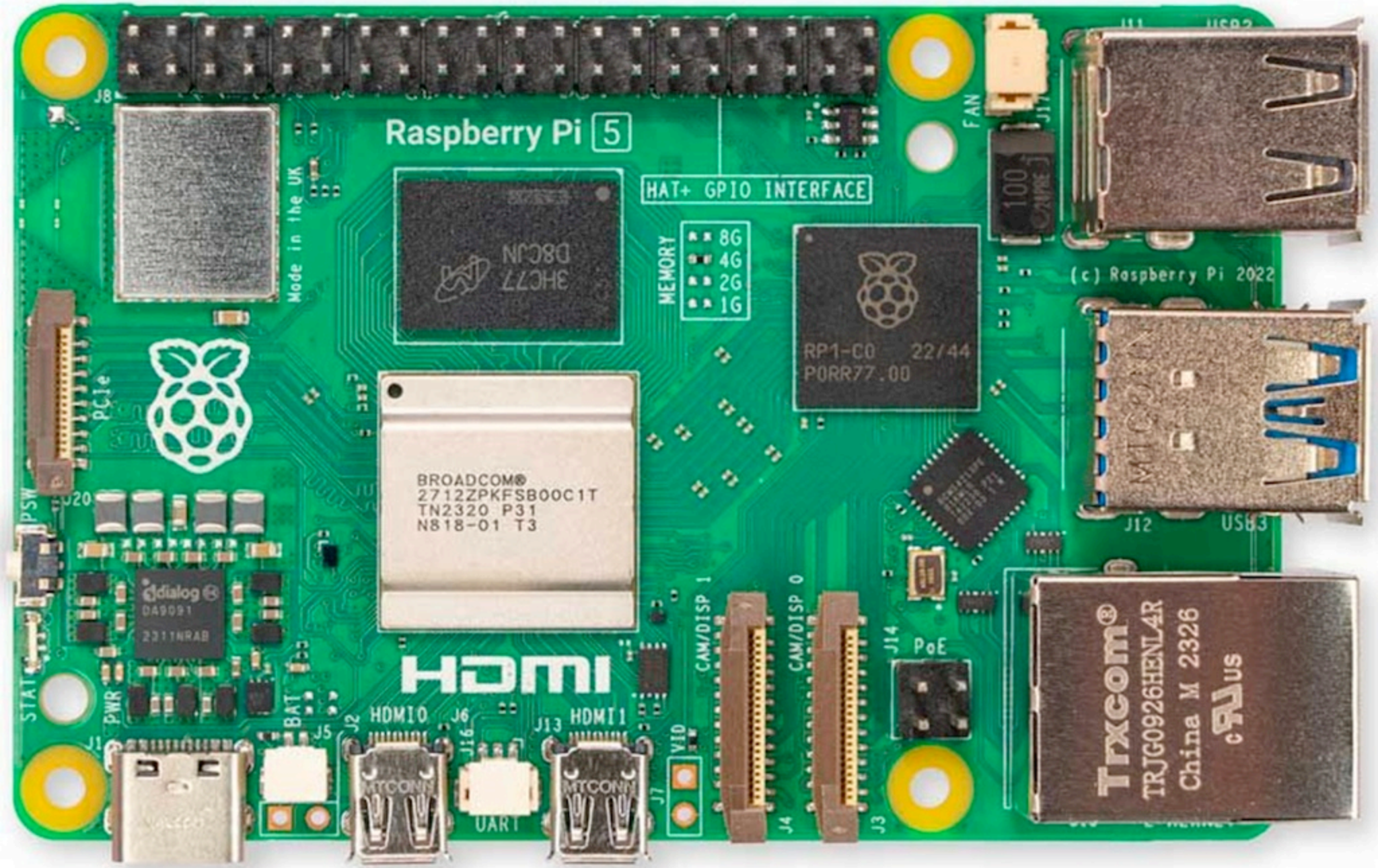
Segment Register Instructions

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

Miscellaneous Instructions

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

Компютър „Raspberry Pi 5“ с RISC-архитектура „ARM“ = „Advanced RISC Machine“



ARM Instruction Set
Quick Reference Card

Key to Tables	
{cond}	Refer to Table **Condition Field {cond}**
<Oprnd2>	Refer to Table **Operand 2**
<fields>	Refer to Table **PSR fields**
{S}	Updates condition flags if S present
C*, V*	Flag is unpredictable after these instructions in Architecture v4 and earlier
Q	Sticky flag. Always updates on overflow (no S option). Read and reset using MRS and MSR
x,y	B meaning half-register [15:0], or T meaning [31:16]
<immed_8r>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<immed_8*4>	A 10-bit constant, formed by left-shifting an 8-bit value by two bits

<a_mode2>	Refer to Table **Addressing Mode 2**
<a_mode2P>	Refer to Table **Addressing Mode 2 (Post-indexed only)**
<a_mode3>	Refer to Table **Addressing Mode 3**
<a_mode4L>	Refer to Table **Addressing Mode 4 (Block load or Stack pop)**
<a_mode4S>	Refer to Table **Addressing Mode 4 (Block store or Stack push)**
<a_mode5>	Refer to Table **Addressing Mode 5**
<reglist>	A comma-separated list of registers, enclosed in braces ({ and })
{!}	Updates base register after data transfer if ! present
§	Refer to Table **ARM architecture versions**

Operation		§	Assembler	S updates	Q	Action	Notes
Move	Move		MOV{cond}{S} Rd, <Oprnd2>	N Z C		Rd := Oprnd2	
	NOT		MVN{cond}{S} Rd, <Oprnd2>	N Z C		Rd := 0xFFFFFFFF EOR Oprnd2	
	SPSR to register	3	MRS{cond} Rd, SPSR			Rd := SPSR	
	CPSR to register	3	MRS{cond} Rd, CPSR			Rd := CPSR	
	register to SPSR	3	MSR{cond} SPSR_<fields>, Rm			SPSR := Rm (selected bytes only)	
	register to CPSR	3	MSR{cond} CPSR_<fields>, Rm			CPSR := Rm (selected bytes only)	
	immediate to SPSR	3	MSR{cond} SPSR_<fields>, #<immed_8r>			SPSR := immed_8r (selected bytes only)	
	immediate to CPSR	3	MSR{cond} CPSR_<fields>, #<immed_8r>			CPSR := immed_8r (selected bytes only)	
Arithmetic	Add		ADD{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn + Oprnd2	
	with carry		ADC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn + Oprnd2 + Carry	
	saturating	5E	QADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + Rn)	No shift/rotate.
	double saturating	5E	QDADD{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm + SAT(Rn * 2))	No shift/rotate.
	Subtract		SUB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn - Oprnd2	
	with carry		SBC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Rn - Oprnd2 - NOT(Carry)	
	reverse subtract		RSB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Oprnd2 - Rn	
	reverse subtract with carry		RSC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V		Rd := Oprnd2 - Rn - NOT(Carry)	
	saturating	5E	QSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm - Rn)	No shift/rotate.
	double saturating	5E	QDSUB{cond} Rd, Rm, Rn		Q	Rd := SAT(Rm - SAT(Rn * 2))	No shift/rotate.
	Multiply	2	MUL{cond}{S} Rd, Rm, Rs	N Z C*		Rd := (Rm * Rs)[31:0]	
	accumulate	2	MLA{cond}{S} Rd, Rm, Rs, Rn	N Z C*		Rd := ((Rm * Rs) + Rn)[31:0]	
	unsigned long	M	UMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(Rm * Rs)	
	unsigned accumulate long	M	UMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)	
	signed long	M	SMULL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(Rm * Rs)	
	signed accumulate long	M	SMLAL{cond}{S} RdLo, RdHi, Rm, Rs	N Z C* V*		RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs)	
	signed 16 * 16 bit	5E	SMULxy{cond} Rd, Rm, Rs			Rd := Rm[x] * Rs[y]	No shift/rotate.
	signed 32 * 16 bit	5E	SMULWy{cond} Rd, Rm, Rs			Rd := (Rm * Rs[y])[47:16]	No shift/rotate.
	signed accumulate 16 * 16	5E	SMLAxy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + Rm[x] * Rs[y]	No shift/rotate.
	signed accumulate 32 * 16	5E	SMLAWy{cond} Rd, Rm, Rs, Rn		Q	Rd := Rn + (Rm * Rs[y])[47:16]	No shift/rotate.
	signed accumulate long 16 * 16	5E	SMLALxy{cond} RdLo, RdHi, Rm, Rs			RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y]	No shift/rotate.
	Count leading zeroes	5	CLZ{cond} Rd, Rm			Rd := number of leading zeroes in Rm	
Logical	Test		TST{cond} Rn, <Oprnd2>	N Z C		Update CPSR flags on Rn AND Oprnd2	
	Test equivalence		TEQ{cond} Rn, <Oprnd2>	N Z C		Update CPSR flags on Rn EOR Oprnd2	
	AND		AND{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn AND Oprnd2	
	EOR		EOR{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn EOR Oprnd2	
	ORR		ORR{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn OR Oprnd2	
	Bit Clear		BIC{cond}{S} Rd, Rn, <Oprnd2>	N Z C		Rd := Rn AND NOT Oprnd2	
	No operation		NOP			R0 := R0	Flags not affected.
	Shift/Rotate						See Table **Operand 2.**
Compare	Compare		CMP{cond} Rn, <Oprnd2>	N Z C V		Update CPSR flags on Rn - Oprnd2	
	negative		CMN{cond} Rn, <Oprnd2>	N Z C V		Update CPSR flags on Rn + Oprnd2	

ARM Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Branch	Branch		`B{cond} label`	R15 := label	label must be within ±32Mb of current instruction.
	with link		`BL{cond} label`	R14 := R15-4, R15 := label	label must be within ±32Mb of current instruction.
	and exchange	4T	`BX{cond} Rm`	R15 := Rm, Change to Thumb if Rm[0] is 1	
	with link and exchange (1)	5T	`BLX label`	R14 := R15 - 4, R15 := label, Change to Thumb	Cannot be conditional.
	with link and exchange (2)	5T	`BLX{cond} Rm`	R14 := R15 - 4, R15 := Rm[31:1] Change to Thumb if Rm[0] is 1	label must be within ±32Mb of current instruction.
Load	Word		`LDR{cond} Rd, <a_mode2>`	Rd := [address]	
	User mode privilege		`LDR{cond}T Rd, <a_mode2P>`		
	branch (and exchange)		`LDR{cond} R15, <a_mode2>`	R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	Byte		`LDR{cond}B Rd, <a_mode2>`	Rd := ZeroExtend[byte from address]	
	User mode privilege		`LDR{cond}BT Rd, <a_mode2P>`		
	signed	4	`LDR{cond}SB Rd, <a_mode3>`	Rd := SignExtend[byte from address]	
	Halfword	4	`LDR{cond}H Rd, <a_mode3>`	Rd := ZeroExtent[halfword from address]	
	signed	4	`LDR{cond}SH Rd, <a_mode3>`	Rd := SignExtend[halfword from address]	
Load multiple	Pop, or Block data load		`LDM{cond}<a_mode4L> Rd{!}, <reglist-pc>`	Load list of registers from [Rd]	
	return (and exchange)		`LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>`	Load registers, R15 := [address][31:1] (§ 5T: Change to Thumb if [address][0] is 1)	
	and restore CPSR		`LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^`	Load registers, branch (§ 5T: and exchange), CPSR := SPSR	Use from exception modes only.
	User mode registers		`LDM{cond}<a_mode4L> Rd, <reglist-pc>^`	Load list of User mode registers from [Rd]	Use from privileged modes only.
Store	Word		`STR{cond} Rd, <a_mode2>`	[address] := Rd	
	User mode privilege		`STR{cond}T Rd, <a_mode2P>`	[address] := Rd	
	Byte		`STR{cond}B Rd, <a_mode2>`	[address][7:0] := Rd[7:0]	
	User mode privilege		`STR{cond}BT Rd, <a_mode2P>`	[address][7:0] := Rd[7:0]	
	Halfword	4	`STR{cond}H Rd, <a_mode3>`	[address][15:0] := Rd[15:0]	
Store multiple	Push, or Block data store		`STM{cond}<a_mode4S> Rd{!}, <reglist>`	Store list of registers to [Rd]	
	User mode registers		`STM{cond}<a_mode4S> Rd{!}, <reglist>^`	Store list of User mode registers to [Rd]	Use from privileged modes only.
Swap	Word	3	`SWP{cond} Rd, Rm, [Rn]`	temp := [Rn], [Rn] := Rm, Rd := temp	
	Byte	3	`SWP{cond}B Rd, Rm, [Rn]`	temp := ZeroExtend([Rn][7:0]), [Rn][7:0] := Rm[7:0], Rd := temp	
Coprocessors	Data operations	2	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`	Coprocessor defined	
		5	`CDP2 p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`		Cannot be conditional.
	Move to ARM reg from coproc	2	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
		5	`MRC2 p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		Cannot be conditional.
	Move to coproc from ARM reg	2	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
		5	`MCR2 p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		Cannot be conditional.
	Load	2	`LDC{cond} p<cpnum>, CRd, <a_mode5>`		
		5	`LDC2 p<cpnum>, CRd, <a_mode5>`		Cannot be conditional.
	Store	2	`STC{cond} p<cpnum>, CRd, <a_mode5>`		
		5	`STC2 p<cpnum>, CRd, <a_mode5>`		Cannot be conditional.
Software interrupt			`SWI{cond} <immed_24>`	Software interrupt processor exception	24-bit value encoded in instruction.
Breakpoint		5	`BKPT <immed_16>`	Prefetch abort *or* enter debug state	Cannot be conditional.

ARM Addressing Modes
Quick Reference Card

Addressing Mode 2 - Word and Unsigned Byte Data Transfer			
Pre-indexed	Immediate offset	`[Rn, #+/-<immed_12>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
	Register offset	`[Rn, +/-Rm]{!}`	
	Scaled register offset	`[Rn, +/-Rm, LSL #<immed_5>]{!}`	Allowed shifts 0-31
		`[Rn, +/-Rm, LSR #<immed_5>]{!}`	Allowed shifts 1-32
		`[Rn, +/-Rm, ASR #<immed_5>]{!}`	Allowed shifts 1-32
		`[Rn, +/-Rm, ROR #<immed_5>]{!}`	Allowed shifts 1-31
		`[Rn, +/-Rm, RRX]{!}`	
Post-indexed	Immediate offset	`[Rn], #+/-<immed_12>`	
	Register offset	`[Rn], +/-Rm`	
	Scaled register offset	`[Rn], +/-Rm, LSL #<immed_5>`	Allowed shifts 0-31
		`[Rn], +/-Rm, LSR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ASR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ROR #<immed_5>`	Allowed shifts 1-31
		`[Rn], +/-Rm, RRX`	

Addressing Mode 2 (Post-indexed only)			
Post-indexed	Immediate offset	`[Rn], #+/-<immed_12>`	
	Zero offset	`[Rn]`	Equivalent to [Rn],#0
	Register offset	`[Rn], +/-Rm`	
	Scaled register offset	`[Rn], +/-Rm, LSL #<immed_5>`	Allowed shifts 0-31
		`[Rn], +/-Rm, LSR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ASR #<immed_5>`	Allowed shifts 1-32
		`[Rn], +/-Rm, ROR #<immed_5>`	Allowed shifts 1-31
		`[Rn], +/-Rm, RRX`	

Addressing Mode 3 - Halfword and Signed Byte Data Transfer			
Pre-indexed	Immediate offset	`[Rn, #+/-<immed_8>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
	Register	`[Rn, +/-Rm]{!}`	
Post-indexed	Immediate offset	`[Rn], #+/-<immed_8>`	
	Register	`[Rn], +/-Rm`	

Addressing Mode 4 - Multiple Data Transfer			
Block load		**Stack pop**	
`IA`	Increment After	`FD`	Full Descending
`IB`	Increment Before	`ED`	Empty Descending
`DA`	Decrement After	`FA`	Full Ascending
`DB`	Decrement Before	`EA`	Empty Ascending
Block store		**Stack push**	
`IA`	Increment After	`EA`	Empty Ascending
`IB`	Increment Before	`FA`	Full Ascending
`DA`	Decrement After	`ED`	Empty Descending
`DB`	Decrement Before	`FD`	Full Descending

Addressing Mode 5 - Coprocessor Data Transfer			
Pre-indexed	Immediate offset	`[Rn, #+/-<immed_8*4>]{!}`	
	Zero offset	`[Rn]`	Equivalent to [Rn,#0]
Post-indexed	Immediate offset	`[Rn], #+/-<immed_8*4>`	
Unindexed	No offset	`[Rn], {8-bit copro. option}`	

ARM architecture versions	
n	ARM architecture version *n* and above.
*n*T	T variants of ARM architecture version *n* and above.
M	ARM architecture version 3M, and 4 and above excluding xM variants
*n*E	E variants of ARM architecture version *n* and above.

Operand 2		
Immediate value	`#<immed_8r>`	
Logical shift left immediate	`Rm, LSL #<immed_5>`	Allowed shifts 0-31
Logical shift right immediate	`Rm, LSR #<immed_5>`	Allowed shifts 1-32
Arithmetic shift right immediate	`Rm, ASR #<immed_5>`	Allowed shifts 1-32
Rotate right immediate	`Rm, ROR #<immed_5>`	Allowed shifts 1-31
Register	`Rm`	
Rotate right extended	`Rm, RRX`	
Logical shift left register	`Rm, LSL Rs`	
Logical shift right register	`Rm, LSR Rs`	
Arithmetic shift right register	`Rm, ASR Rs`	
Rotate right register	`Rm, ROR Rs`	

PSR fields (use at least one suffix)		
Suffix	**Meaning**	
`c`	Control field mask byte	PSR[7:0]
`f`	Flags field mask byte	PSR[31:24]
`s`	Status field mask byte	PSR[23:16]
`x`	Extension field mask byte	PSR[15:8]

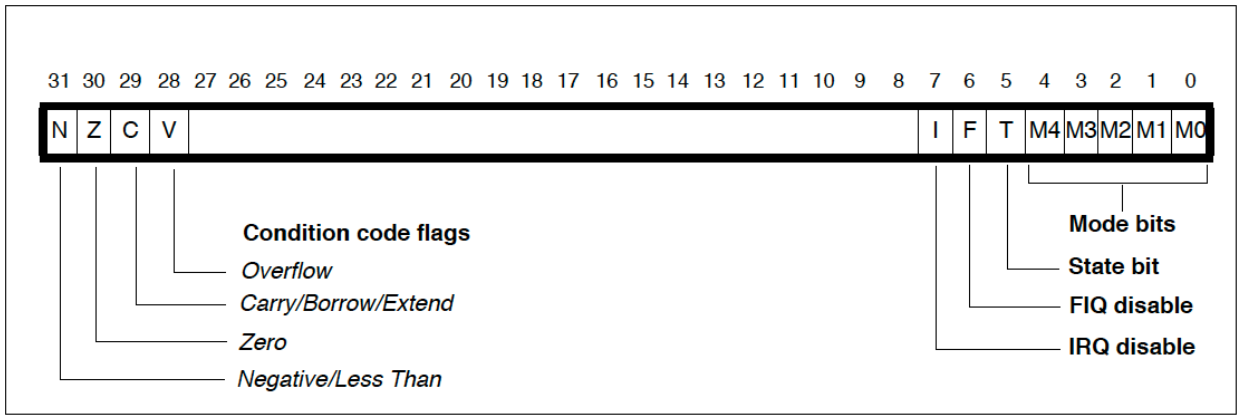
Condition Field {cond}		
Mnemonic	**Description**	**Description (VFP)**
`EQ`	Equal	Equal
`NE`	Not equal	Not equal, or unordered
`CS / HS`	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
`CC / LO`	Carry Clear / Unsigned lower	Less than
`MI`	Negative	Less than
`PL`	Positive or zero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

Key to tables	
`{!}`	Updates base register after data transfer if ! present. (Post-indexed always updates.)
`<immed_8r>`	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.
`+/-`	+ or -. (+ may be omitted.)

Operation		§	Assembler	S updates	Action	Notes
Divide	Signed or Unsigned	RM	`<op> Rd, Rn, Rm`		Rd := Rn / Rm `<op>` is `SDIV` (signed) or `UDIV` (unsigned)	

Operation		§	Assembler	Action	Notes
Reverse	Bits in word	T2	`RBIT Rd, Rm`	For (i = 0; i < 32; i++) : Rd[i] = Rm[31– i]	

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



Note: C is inverted after subtraction!

Компютър „Mac mini G4“ с RISC-архитектура „POWER[PC]“ = „Performance Optimization With Enhanced RISC“

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, and parameter passing. The system libraries described in Chapter 6 require this calling sequence.

Note – The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions as long as they conform to the tag requirements for stack traceback; see **Stack Traceback Using Tags** in Chapter 4. Nonetheless, it is recommended that all functions use the standard calling sequences when possible.

Note – C programs follow the conventions given here. For specific information on the implementation of C, see **Coding Examples** in this chapter.

Registers

The PowerPC Architecture provides 32 general purpose registers, each 32 bits wide. In addition, the architecture provides 32 floating-point registers, each 64 bits wide, and several special purpose registers. All of the integer, special purpose, and floating-point registers are global to all functions in a running program. Brief register descriptions appear in Table 3-3, followed by more detailed information about the registers.

Table 3-3 Processor Registers

Register Name	Usage
`r0`	Volatile register which may be modified during function linkage
`r1`	Stack frame pointer, always valid
`r2`	System-reserved register
`r3-r4`	Volatile registers used for parameter passing and return values
`r5-r10`	Volatile registers used for parameter passing
`r11-r12`	Volatile registers which may be modified during function linkage
`r13`	Small data area pointer register
`r14-r30`	Registers used for local variables
`r31`	Used for local variables or "environment pointers"
`f0`	Volatile register
`f1`	Volatile register used for parameter passing and return values
`f2-f8`	Volatile registers used for parameter passing

Table 3-3 Processor Registers *(Continued)*

Register Name	Usage
f9-f13	Volatile registers
f14-f31	Registers used for local variables
CR0-CR7	Condition Register Fields, each 4 bits wide
LR	Link Register
CTR	Count Register
XER	Fixed-Point Exception Register
FPSCR	Floating-Point Status and Control Register

Registers r1, r14 through r31, and f14 through f31 are nonvolatile; that is, they "belong" to the calling function. A called function shall save these registers' values before it changes them, restoring their values before it returns. Registers r0, r3 through r12, f0 through f13, and the special purpose registers CTR and XER are volatile; that is, they are not preserved across function calls. Furthermore, the values in registers r0, r11, and r12 may be altered by cross-module calls, so a function cannot depend on the values in these registers having the same values that were placed in them by the caller.

Register r2 is reserved for system use and should not be changed by application code.

Register r13 is the small data area pointer. Process startup code for executables that reference data in the small data area with 16-bit offset addressing relative to r13 must load the base of the small data area (the value of the loader-defined symbol _SDA_BASE_) into r13. Shared objects shall not alter the value in r13. See **Small Data Area** in Chapter 4 for more details.

Languages that require "environment pointers" shall use r31 for that purpose.

Fields CR2, CR3, and CR4 of the condition register are nonvolatile (value on entry must be preserved on exit); the rest are volatile (value in the field need not be preserved). The VE, OE, UE, ZE, XE, NI, and RN (rounding mode) bits of the FPSCR may be changed only by a called function (for example, fpsetround()) that has the documented effect of changing them. The rest of the FPSCR is volatile.

The following registers have assigned roles in the standard calling sequence:

`r1`	The stack pointer (stored in `r1`) shall maintain 16-byte alignment. It shall always point to the lowest allocated, valid stack frame, and grow toward low addresses. The contents of the word at that address always point to the previously allocated stack frame. If required, it can be decremented by the called function; see **Dynamic Stack Space Allocation** later in this chapter.
`r3` through `r10` and `f1` through `f8`	These sets of volatile registers may be modified across function invocations and shall therefore be presumed by the calling function to be destroyed. They are used for passing parameters to the called function; see **Parameter Passing** in this chapter. In addition, registers `r3`, `r4`, and `f1` are used to return values from the called function, as described in **Return Values**.
`CR` bit 6 (`CR1`, "floating-point invalid exception")	This bit shall be set by the caller of a "variable argument list" function, as described in **Variable Argument Lists** later in this chapter.
`LR` (Link Register)	This register shall contain the address to which a called function normally returns. `LR` is volatile across function calls.

Signals can interrupt processes (see `signal (BA_OS)` in the *System V Interface Definition*). Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with all registers restored to their original values. Thus, programs and compilers may freely use all registers above except those reserved for system use without the danger of signal handlers inadvertently changing their values.

PowerPC User-Level Instruction Set Quick Reference Card

Based on a mnemonic presentation idea from Bill Karsh in his PowerPC tutorial series in MacTech magazine (http://macte.ch/luHry)

Notation

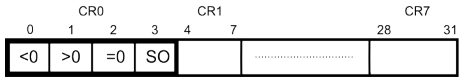
||	Concatenation of bit blocks
|	Alternation
UIMMnn	Unsigned immediate of *nn* bits (ie: UIMM16 = 16 bits)
SIMMnn	Signed immediate of nn bits (ie: SIMM26 = 26 bits)
EXT	Sign-extend to word
(rA|0)	In some instances, value "0" for register rA (meaning r0) is a special case that actually means "use the value 0".
< >	List of functional suffixes (append 0 or 1 from the list)
[ ]	List of optional suffixes (0 or more, in the order specified)

Example of multiple suffixes for an instruction:

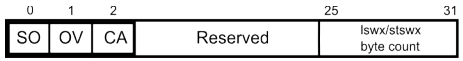
add<c | e | me | ze>[o, .]: **add**, **addc**, **addme, addze, addo, addco**, **addeo**, **addmeo**, **addzeo**, **add.**, **addc.**, **adde.**, **addme.**, **addze.**, **addo.**, **addeo.**, **addmeo.**, **addzeo**. are all valid.

Registers

- r0-r31: General-purpose integer registers
- LR: Link register, saves return address of branches that link
- CTR: Counter for auto-decrementing loops
- CR: Condition register
  - composed of 8 condition records (CR0-CR7)
  - saves results of comparisons and ALU operations



- XER: Exception register, saves overflow and carry states



Label suffixes

Assemblers general recognize several suffixes for labels and values. These suffixes provide ways to extract only parts of an operand for use in immediate values.

- VALUE**@h**: Only the high 16-bit part (bits 0-15).
- VALUE**@ha**: Like @h, but adjusted to compensate for sign extension applied by an "addi VALUE@l" on the same register.
- VALUE**@l**: Only the low 16-bit part (bits 16-31).

NOTE: Compare the following two ways to load an immediate value into a register (equivalent in result but different in spirit):

1. addis rD, 0, VALUE@ha
   addi rD, rD, VALUE@l
2. addis rD, 0, VALUE@h
   ori rD, rD, VALUE@l

With the first method, the **addi** instruction does sign extension on its 16-bit signed immediate operand. If we want to load, for instance 0x12348765, the value 0x8765 from "0x12348765**@l**" will be sign-extended to 0xFFFF8765. This will cause an off-by-one error if we add it with 0x12340000 from **addis** rD,0, 0x12348765**@h**. The **@ha** suffix verifies this condition ("negative" low 16 bits) and adjusts the high-part so that when it is added with the sign-extended low part, the result correct: 0x12348765 in our case. With the second method, using **ori** which does not sign-extend its operand, the high part requires no adjustment.

Load and store instructions

Addressing modes

The PowerPC has only two addressing modes, but combining them with the load/store instructions options yields many possibilities. The addressing modes (using **lwz** as an example) are:

- **lwz** rD, offset(rA|0) → Register-indirect with immediate offset
  → EA = (rA + offset) or (0 + offset)
  → Offset is a 16 bit signed immediate value
- **lwzx** rD, (rA|0), rB → Register-indirect with indexing
  → EA = (rA + rB) or (0 + rB)

Single loads and stores

Instruction	Operation
`lbz[u,x] rD,d(rA)`	rD ← *byte from* MEM[EA]
`lhz[u,x] rD,d(rA)`	rD ← *half-word from* MEM[EA]
`lha[u,x] rD,d(rA)`	rD ← *sign-extended half word from* MEM[EA]
`lwz[u,x] rD,d(rA)`	rD ← word from MEM[EA]
`stb[u,x] rS,d(rA)`	rS[24:31] → MEM[EA] (*store byte*)
`sth[u,x] rS,d(rA)`	rS[16:31] → MEM[EA] (*store half-word*)
`stw[u,x] rS,d(rA)`	rS → MEM[EA] (*store word*)

- **"z"** load suffix: treat as unsigned, zero-extend, right-justify.
- **"a"** load suffix: "algebraic": sign-extend to word.
- `[u]`: "update": if (rA != 0) then rA←EA after load or store. In the case of loads, condition (rD != rA) also applies (logically so).
- `[x]`: "with indexing" (see addressing modes above), use operands as in "lwzx rD, (rA|0), rB" instead of "lwz rD, d(RA)".

Multiple loads and stores

Instruction	Operation
`lmw rD,d(rA)`	n = (32 – rD); n consecutive words starting at EA are loaded into GPRs rD through r31. For example : lmw r29,0(r8) loads r29, r30 and r31 from consecutive, increasing addresses starting at EA.
`stmw rS,d(rA)`	n = (32 – rS); n consecutive words starting at EA are stored from the GPRs rS through r31. For example, if rS = 29, r29, 30 and r31 are stored at consecutive, increasing addresses starting at EA.
String loads and stores (lswi, lswx, stswi, stswx) are omitted for brevity and because they are not available on all PPCs.	

Arithmetic and logic instructions

Addition, subtraction, negation

Instruction	Operands	Operation
`add<c,e>[o,.]`	`rD,rA,rB`	rD ← rA + rB
`addi<s,c,c.>`	`rD,(rA|0),SIMM`	rD ← (rA|0) + EXT(SIMM16)
`addme[o,.]`	`rD,rA`	rD ← rA + XER[CA] - 1
`addze[o,.]`	`rD,rA`	rD ← rA + 0 + XER[CA]
`neg[o,.]`	`rD,rA`	rD ← (¬rA + 1) *(2's complement negation)*
`subf<c,e>[o,.]`	`rD,rA,rB`	rD ← rB – rA
`subfic`	`rD,rA,SIMM`	rD ← EXT(SIMM16) – rA
`subfme[o,.]`	`rD,rA`	rD ← -1 – rA + XER[CA]
`subfze[o,.]`	`rD,rA`	rD ← 0 – rA + XER[CA]

- **"i"** suffix: "immediate": second operand is 16-bit sign-extended immediate value.
- **"s"** suffix: "shifted": immediate value is logical shifted left 16 bits prior to being used.
- **"z"** suffix: replaces rB with immediate value 0 (0x00000000).
- **"m"** suffix: replaces rB with immediate value -1 (0xFFFFFFFF).
- **"e"** suffix: extended add or subtract. The value of `XER[CA]` is added to the result, enabling multi-word carry arithmetic. The value of `XER[CA]` is updated by these operations also.
- **"c"** suffix: carry updated. `XER[CA]` is updated with the operation's carry state (by default, the carry is unaffected).
- `[o]`: overflow updated. `XER[OV]` and `XER[SO]` are updated according to whether the operation overflows or not.
- `[.]`: Record result of operation in `CR0` (<0, >0, =0, SO)

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.
Revision 1, October 12, 2010.

Bitwise logical operations and shifts

Instruction	Operands	Operation
`and[c,.]`	`rD,rA,rB`	rD ← rA ∧ rB
`andi.`	`rD,rA,UIMM`	rD ← rA ∧ UIMM16
`andis.`	`rD,rA,UIMM`	rD ← rA ∧ (UIMM16 << 16)
`cntlzw[.]`	`rD,rA`	rD ← *number of leading zeros in* rA
`eqv[.]`	`rD,rA,rB`	rD ← ¬(rA ⊕ rB) *(would be "xnor")*
`extsb[.]`	`rD,rA`	rD ← EXT(rA[24:31]) (*sign-extend low byte of* rA)
`extsh[.]`	`rD,rA`	rD ← EXT(rA[16:31]) (*sign-extend low half-word of* rA)
`nand[.]`	`rD,rA,rB`	rD ← ¬(rA ∧ rB)
`nor[.]`	`rD,rA,rB`	rD ← ¬(rA ∨ rB)
`or[c,.]`	`rD,rA,rB`	rD ← rA ∨ rB
`ori`	`rD,rA,UIMM`	rD ← rA ∨ (UIMM16)
`oris`	`rD,rA,UIMM`	rD ← rA ∨ (UIMM16 << 16)
`slw[.]`	`rD,rA,rB`	rD ← rA << rB[26:31] (logical)
`srw[.]`	`rD,rA,rB`	rD ← rA >> rB[26:31] (logical)
`srawi[.]`	`rD,rA,UIMM`	rD ← rA >> UIMM5 (arithmetic)
`sraw[.]`	`rD,rA,rB`	rD ← rA >> rB[26:31] (arithmetic)
`xor[c,.]`	`rD,rA,rB`	rD ← rA ⊕ rB
`xori`	`rD,rA,UIMM`	rD ← rA ⊕ (UIMM16)
`xoris`	`rD,rA,UIMM`	rD ← rA ⊕ (UIMM16 << 16)

- `[c]`: Complement (invert) the value from rB prior to using it. The actual value residing in rB is unaffected.
- `[.]`: Record result of operation in CR0 (<0, >0, =0, SO)
- NOTE: on shifts, values 0-32 are valid. For arithmetic shift rights, a value of 32 fills the word with the sign bit. For logical shift lefts, a value of 32 sets the word to 0.

Multiplication

Inst.	Operands	Operation
`mulhw`	`rD,rA,rB`	rD ← rA × rB *(32 upper bits of 64-bit result)*
`mulli`	`rD,rA,SIMM`	rD ← (rA × SIMM16) *(32 lower bits of 48-bit result)*
`mullw`	`rD,rA,rB`	rD ← rA × rB *(32 lower bits of 64-bit result)*

Division

Inst.	Operands	Operation
`divw<u>[o,.]`	`rD,rA,rB`	rD ← rA ÷ rB

- "`u`" suffix: treat operands as unsigned numbers
- `[o]`: Record overflow of result
- `[.]`: Record result of operation in CR0 (<0, >0, =0, SO)

Rotate and mask

Inst.	Operands	Operation
`rlwimi[.]`	`rD,rA,UIMM,MB,ME`	rD ← rotate rA left by UIMM bits, mask and insert result in rD
`rlwinm[.]`	`rD,rA,UIMM,MB,ME`	rD ← *rotate rA left by UIMM bits and mask*
`rlwnm[.]`	`rD,rA,rB,MB,ME`	rD ← *rotate rA left by rB bits and mask*

- For all these instructions, a mask M is built by starting with a zero-word (0x00000000) and setting bits to "1" starting at bit number MB and ending at bit number ME, both inclusive. It is possible to wrap-around while generating the mask (ie: MB > ME).

Examples:

MASK(MB,ME) with MB=29 and ME=3:

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

MASK(MB,ME) with MB=8 and ME=14:

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

`rlwimi r3,r4,6,20,25 (r4 = 0x0FF0_0017, r3 = 0x12AB_CDEF)`

1.	Generate mask :	MASK(20,25)	= `0x0000_0FC0`
2.	Rotate source:	tmp1 = r4 ROL 6	= `0xFC00_05C3`
3.	Extract field:	tmp2 = tmp1 ∧ MASK	= `0x0000_05C0`
4.	Mask destination:	tmp3 = r3 ∧ ¬MASK	= `0x12AB_C02F`
5.	Insert field in destination:	r3 ← tmp2 ∨ tmp3	= `0x12AB_C5EF`

The previous 5 steps as binary:

```
1. 0b0000_0000_0000_0000_0000_1111_1100_0000
2. 0b1111_1100_0000_0000_0000_0101_1100_0011
3. 0b0000_0000_0000_0000_0000_0101_1100_0000
4. 0b0001_0010_1010_1011_1100_0000_0010_1111
5. 0b0001_0010_1010_1011_1100_0101_1110_1111
```

`rlwinm r3,r4,12,20,31 (r4 = 0x5A70_00BB)`

1.	Generate mask :	MASK(20,31)	= `0x0000_0FFF`
2.	Rotate source:	tmp1 = r4 ROL 12	= `0x000B_B5A7`
3.	Extract field in destination:	r3 ← tmp1 ∨ MASK	= `0x0000_05A7`

Comparison instructions

Inst.	Operands	Operation
`cmp`	`crD,L,rA,rB`	Compare signed rA to rB
`cmpi`	`crD,L,rA,SIMM`	Compare signed rA to EXT(SIMM16)
`cmpl`	`crD,L,rA,rB`	Compare unsigned rA to rB
`cmpli`	`crD,L,rA,UIMM`	Compare unsigned rA to (0x0000 || UIMM16)

- `crD` can be omitted. In that case, the assembler assumes `cr0.`
- The L field means "Long" (64-bit compare) if set to "1", or 32-bit compare if set to "0". On 32-bit PowerPC, L should always be set to "0". Because of this, a simplified mnemonic exists for all "cmp"-series instructions: "`cmpw crD, rA, rB`" is equivalent to "`cmp crD,0,rA,rB`", etc.
- For all these instructions, the result of a comparison from rA to (rB|SIMM|UIMM) is stored in the specified condition register crD. For example, cmp 3,0,rA,rB would yield `cr3` = "`100` || `XER[SO]`" if rA < rB, `cr3` = "`010` || `XER[SO]`" if rA > rB and `cr3` = "`001` || `XER[SO]`" if rA = rB.

Condition register manipulation instructions

Inst.	Operands	Operation
`crand`	`crbD,crbA,crbB`	crbD ← crbA ∧ crbB
`crandc`	`crbD,crbA,crbB`	crbD ← crbA ∧ ¬crbB
`creqv`	`crbD,crbA,crbB`	crbD ← ¬(crbA ⊕ crbB)
`crnand`	`crbD,crbA,crbB`	crbD ← ¬(crbA ∧ crbB)
`crnor`	`crbD,crbA,crbB`	crbD ← ¬(crbA ∨ crbB)
`cror`	`crbD,crbA,crbB`	crbD ← crbA ∨ crbB
`crorc`	`crbD,crbA,crbB`	crbD ← crbA ∨ ¬crbB
`crxor`	`crbD,crbA,crbB`	crbD ← crbA ⊕ crbB
`mcrf`	`crD,crA`	crD ← crA (*move field A to field D*)
`crclr`	`crbD`	*Simplified for* `crxor crbD,crbD,crbD`
`crmove`	`crbD,crbA`	*Simplified for* `cror crbD,crbA,crbA`
`crnot`	`crbD,crbA`	*Simplified for* `crnor crbD,crbA,crbA`
`crset`	`crbD`	*Simplified for* `creqv crbD,crbD,crbD`

- For the `cr<OP>` instructions, operands `crb[A,B,D]` mean "condition register bit", with value 0-31. All of these instructions carry-out logical operations between single bits of the CR, no matter what the conventional "meanings" of the bits are (<0, >0, =0, SO).

Example:

- `cror 0,5,6` : `CR[0]` ← `CR[5]` ∨ `CR[6]` , thus `cr0[=0]` ← 1, if `cr1` had ">=" comparison result, otherwise `cr0[=0]` ← 0.

Branch instructions

The PowerPC architecture uses a very flexible branching unit to decode the several fields contained in branch instructions. We will cover the basic branch instructions and their fields, and then present tables and examples of simplified branch mnemonics.

Field names

- BI (Branch Input): which bit of the CR is used as a branch condition
- BO (Branch Options): how to treat CTR and BI to determine if branching occurs
- Target: where to branch

Branch instructions

Inst.	Operands	Operation
`b[l,a]`	`target`	Branch unconditionally
`bc[l,a]`	`BO,BI,target`	Branch conditionally
`bclr[l]`	`BO,BI`	Branch to LR conditionally
`bcctr[l]`	`BO,BI`	Branch to CTR conditionally

- `[l]`: linking: store current PC + 4 in LR, so that a "`blr`" instruction can be used to return from a function call.
- `[a]`: absolute: target is an absolute address instead of a PC-relative displacement.
    - For the `b[l,a]` instruction, target is a 26-bit signed immediate with 2 LSbs always "0" (4-bytes aligned). Maximum branch distance is [-33,554,432...33,554,428].
    - For the `bc[l,a]` instruction, target is a 16-bit signed immediate with 2 LSbs always "0" (4-bytes aligned). Maximum branch distance is [-32,768...32764].
    - In the case of non-absolute (no `[a]` option) branches, the target displacement is added to PC. A displacement of 0 is an infinite loop at the current PC. For the unconditional branch (`[a]` option), the target is still signed, but the displacement is based around 0x0000_0000.
- To obtain the value of PC, one can branch linking to the next instruction (`bl +4`). The LR will contain the PC value at that next instruction. This trick is used by compilers to access local constant pools inserted after function return instructions.
- PowerPC assemblers and linkers will always adjust relocations so that displacements and labels can be specified directly, without having to adjust the value formats to the field formats. For example, "`b +8`" will get encoded as a target of 0x000002 (stripped of the 2 LSbs) automatically in the instruction.
- BI values can be simplified with constants named `cr0` through `cr7` with values 0-7 respectively and constants named lt,gt,eq,so with values 0-3 respectively . Then, `cr4[<0]`, which is BI=16 can be written as `(cr4*4)+lt`.

BO values

BO	Branch if	Symbol
`0000y`	Decremented CTR ≠ 0 and the condition is **false**.	`dnzf`
`0001y`	Decremented CTR = 0 and the condition is **false**.	`dzf`
`001zy`	Branch if the condition is **false**.	`f`
`0100y`	Decremented CTR ≠ 0 and the condition is **true**.	`dnzt`
`0101y`	Decremented CTR = 0 and the condition is **true**.	`dzt`
`011zy`	Branch if the condition is **true**.	`t`
`1z00y`	Decremented CTR ≠ 0 (only CTR checked).	`dnz`
`1z01y`	Decremented CTR = 0 (only CTR checked).	`dz`
`1z1zz`	Branch always.	`–`

- Symbols (3rd column of table above): the "`c`" of "`bc`" and the BO field value can be omitted and replaced with one of these symbols as a suffix. For example, and assuming y=z=0, the instruction **"`bc 8,5,label`"** can be replaced with **"`bdnzt 5,label`"**.
- "y" bits are "branch likely to be taken" hints if set to "1". This is ignored by many implementations. A suffix of "-" added to the instruction clears this bit (branch not likely taken). A suffix of "+" added to the instruction sets this bit (branch likely taken). Example: **"`bdnzt+ 5,label`"** is equivalent to **"`bc 9,5,label`"**. Many processors of the PowerPC family ignore this hint.
- "z" bits should be zeroed as they are for future extensions.

Examples:

- `bc 8,5,label` : Branch if decremented CTR ≠ 0 and CR[5] = "1".
- `bdnzt 5,label` : same as above.
- `bdnzt (cr1*4)+gt,label` : same as above.
- `bl label`: Branch and link to label (call function, return with `blr`).
- `blr`: Branch unconditional to LR (return from function).
- `bdza label`: Branch absolute to label if decremented CTR = 0.
- `btctr lt`: Branch to CTR if `cr0[<0]` (`CR[0]`) = "1".
- `bf eq,label`: Branch to label if `cr0[=0]` (`CR[2]`) = "0".

Simplified branches (or "classic" branches)

There are simplified "branch conditional" mnemonics that emulate the classic branches of other instruction sets. These mnemonics are for instructions that do not test the CTR.

Instruction	Operands	Operation
`b<test>[l,a]`	`[crN,]target`	Branch conditionally
`b<test>lr[l]`	`[crN]`	Branch to LR conditionally
`b<test>ctr[l]`	`[crN]`	Branch to CTR conditionally

- `[crN]` is an optional CR subfield number (ie: `cr0-cr7`), on which the test will take place. If omitted, the default is `cr0`.

Simplified branches tests (using b<test> as example)

Symbol	Branch if
`beq`	Equal, or zero (`cr[=0]` = "1")
`bge`	Greater than or equal (`cr[>0]` = "1" ∨ `cr[=0]` = "1")
`bgt`	Greater than (`cr[>0]` = "1")
`ble`	Less than or equal (`cr[<0]` = "1" ∨ `cr[=0]` = "1")
`blt`	Less than (`cr[<0]` = "1")
`bne`	No equal, or not zero (`cr[=0]` = "0")
`bng`	Not greater than (equivalent to b**le**)
`bnl`	Not less than (equivalent to b**ge**)
`bns`	Not summary overflow (`cr[SO]` = "0")
`bso`	Summary overflow (`cr[SO]` = "1")

Examples:

- `bne label` : Branch to label if `cr0[=0]` = "0".
- `bsola cr2,label` : Branch absolute linking to label if `cr2[SO]` = "1".
- `bltl label` : Branch linking to label if `cr0[<0]` = "1".
- `beqctr` cr4 : Branch to CTR if `cr4[=0]` = "1".
- `bgtlrl` : Branch linking to LR if `cr0[>0]` = "1".
- `bl label`: Branch and link to label (call function, return with `blr`).
- `blr`: Branch unconditional to LR (return from function).

Special Purpose Register (SPR) Operations

Inst.	Operands	Operation
`mcrxr`	`crD`	crD ←XER[0:3] then zero XER[0:3]
`mfcr`	`rD`	rD ← CR[0:31]
`mfspr`	`rD,SPR`	rD ← SPR
`mtcrf`	`crM,rS`	CR updated with rS[crM] (*see notes below*)
`mtspr`	`SPR,rS`	SPR ← rS
`mtcr`	`rS`	*Simplified for* `mtcrf 0xFF, rS`

- `crM` is an 8 bit immediate mask (value 0x00-0xFF). The MSb means `cr0`, the LSb means `cr7`, and bits in between mean `cr1-cr6`. For example, `crM = 0xA2 = 0b1010_0010` would mean to load `cr0`, `cr2` and `cr6` from rS into the CR, and leave the other fields (`cr1`,`cr3`,`cr4`,`cr5` and `cr7`) intact.
- There are simplified `mtspr` mnemonics for several SPRs which allow the omission of the SPR number: `mtctr`, `mtlr`, `mtxer`.
- There are simplified `mfspr` mnemonics for several SPRs which allow the omission of the SPR number: `mfctr`, `mflr`, `mfxer`.

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.

Trap and System Call Instructions

Inst.	Operands	Operation
sc	—	System call
tw	TO,rA,rB	Trap if rA <TO> rB is true
twi	TO,rA,SIMM	Trap if rA <TO> EXT(SIMM16) is true

- TO is a 5-bit field of conditions to test. If any of the conditions are met, the trap is taken.
  - TO[0] (ie: mask = 0b10000) means (a < b)
  - TO[1] (ie: mask = 0b01000) means (a > b)
  - TO[2] (ie: mask = 0b00100) means (a = b)
  - TO[3] means (a < b) with unsigned compare
  - TO[4] means (a > b) with unsigned compare

Condensed alphabetical instructions list

Instruction		Operation
add[.]	rD,rA,rB	Add
addc[o,.]	rD,rA,rB	Add, saving carry
adde[o,.]	rD,rA,rB	Add extended (adding carry)
addi	rD,(rA|0),SIMM	Add immediate
addis	rD,(rA|0),SIMM	Add immediate shifted
addic[.]	rD,(rA|0),SIMM	Add immediate shifted saving carry
addme[o,.]	rD,rA	Add to minus one, extended
addze[o,.]	rD,rA	Add to zero, extended
and[.]		AND
andc[.]		AND with complement
andi.	rD,rA,UIMM	AND with immediate
andis.	rD,rA,UIMM	AND with shifted immediate
b[l,a]	target	Branch always
bc[l,a]	BO,BI,target	Branch conditionally
bcctr[l]	BO,BI	Branch conditionally to CTR
bclr[l]	BO,BI	Branch conditionally to LR
beq[l,a]	[crN,]target	Branch on equal (or zero)
bge[l,a]	[crN,]target	Branch on greater than or equal
bgt[l,a]	[crN,]target	Branch on greater than
ble[l,a]	[crN,]target	Branch on lower than or equal
blt[l,a]	[crN,]target	Branch on lower than
bne[l,a]	[crN,]target	Branch on not equal (or non-zero)
bng[l,a]	[crN,]target	Branch on not greater than
bnl[l,a]	[crN,]target	Branch on not lower than
bns[l,a]	[crN,]target	Branch on not summary overflow
bso[l,a]	[crN,]target	Branch on summary overflow
cmp	[crD,]L,rA,rB	Compare signed
cmpi	[crD,]L,rA,SIMM	Compare signed with immediate
cmpl	[crD,]L,rA,rB	Compare unsigned
cmpli	[crD,]L,rA,UIMM	Compare unsigned with immed.
cntlzw[.]	rD,rA	Count leading zeros in word
crand	crbD,crbA,crbB	AND on CR bits
crandc	crbD,crbA,crbB	AND complemented on CR bits
crclr	crbD	Clear CR bit
creqv	crbD,crbA,crbB	EQV on CR bits
crmove	crbD,crbA	Move CR bit
crnand	crbD,crbA,crbB	NAND on CR bits
crnor	crbD,crbA,crbB	NOR on CR bits
crnot	crbD,crbA	NOT on CR bit
cror	crbD,crbA,crbB	OR on CR bits
crorc	crbD,crbA,crbB	OR complemented on CR bits
crset	crbD	Set CR bit
crxor	crbD,crbA,crbB	XOR on CR bits
divw[o,.]	rD,rA,rB	Divide word
divwu[o,.]	rD,rA,rB	Divide word unsigned
eqv[.]	rD,rA,rB	EQV (NOT (rA XOR rB)
extsb[.]	rD,rA	Sign-extend byte
extsh[.]	rD,rA	Sign-extend half-word
lbz[u,x]	rD,d(rA)	Load byte unsigned
lha[u,x]	rD,d(rA)	Load half-word and sign-extend
lhz[u,x]	rD,d(rA)	Load half-word unsigned
lmw	rD,d(rA)	Load multiple words
lwz[u,x]	rD,d(rA)	Load word
mcrf	crD,crA	Move condition register field
mcrxr	crD	Move XER[0:3] to CR field
mfcr	rD	Move from CR
mfspr	rD,SPR	Move from SPR
mtcr	rS	Move to CR
mtcrf	crM,rS	Update CR fields
mtspr	SPR,rS	Move to SPR
mulhw	rD,rA,rB	Multiply high word
mulli	rD,rA,SIMM	Multiply low immediate
mullw	rD,rA,rB	Multiply low word
nand[.]	rD,rA,rB	NAND
neg[o,.]	rD,rA	Negate (2's complement)
nor[.]	rD,rA,rB	NOR
or[.]	rD,rA,rB	OR
orc[.]	rD,rA,rB	OR with complement
ori	rD,rA,UIMM	OR with immediate
oris	rD,rA,UIMM	OR with shifted immediate
rlwimi[.]	rD,rA,UIMM,MB,ME	Rotate left word immediate and mask insert
rlwinm[.]	rD,rA,UIMM,MB,ME	Rotate left word immediate and mask
rlwnm[.]	rD,rA,rB,MB,ME	Rotate left word and mask
sc		System call
slw[.]	rD,rA,rB	Shift left word (logical)
sraw[.]	rD,rA,rB	Shift right word (arithmetic)
srawi[.]	rD,rA,UIMM	Shift right immediate (arithmetic)
srw[.]	rD,rA,rB	Shift right word (logical)
stb[u,x]	rS,d(rA)	Store byte
sth[u,x]	rS,d(rA)	Store half-word
stmw	rS,d(rA)	Store multiple words
stw[u,x]	rS,d(rA)	Store word
subf[o,.]	rD,rA,rB	Subtract from
subfc[o,.]	rD,rA,rB	Subtract from, update carry
subfe[o,.]	rD,rA,rB	Subtract from, extended
subfic	rD,rA,SIMM	Subtract from immediate, update carry
subfme[o,.]	rD,rA	Subtract from -1, extended
subfze[o,.]	rD,rA	Subtract from 0, extended
tw	TO,rA,rB	Trap word
twi	TO,rA,SIMM	Trap word immediate
xor[.]	rD,rA,rB	XOR
xorc[.]	rD,rA,rB	XOR with complement
xori	rD,rA,UIMM	XOR with immediate
xoris	rD,rA,UIMM	XOR with shifted immediate

NOTE: Many features have been omitted for brevity. This includes Floating-Point operations and registers, supervisor-level operations, string load and stores and ALTIVEC instructions.

Table F-19. Simplified Mnemonics for SPRs (Continued)

Special-Purpose Register	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
Time base lower	**mttbl r**S	**mtspr 284,r**S	**mftb r**D	**mftb r**D**,268**
Time base upper	**mttbu r**S	**mtspr 285,r**S	**mftbu r**D	**mftb r**D**,269**
Processor version register	—	—	**mfpvr r**D	**mfspr r**D**,287**
IBAT register, upper	**mtibatu** *n,* **r**S	**mtspr** 528 + (2 * *n*)**,r**S	**mfibatu r**D*, n*	**mfspr r**D,528 + (2 * *n*)
IBAT register, lower	**mtibatl** *n,* **r**S	**mtspr** 529 + (2 * *n*)**,r**S	**mfibatl r**D*, n*	**mfspr r**D,529 + (2 * *n*)
DBAT register, upper	**mtdbatu** *n,* **r**S	**mtspr** 536 + (2 **n*)**,r**S	**mfdbatu r**D*, n*	**mfspr r**D,536 + (2 **n*)
DBAT register, lower	**mtdbatl** *n,* **r**S	**mtspr** 537 + (2 * *n*)**,r**S	**mfdbatl r**D*, n*	**mfspr r**D,537 + (2 * *n*)

Following are examples using the SPR simplified mnemonics found in Table F-19:

1. Copy the contents of **r**S to the XER.
   mtxer rS (equivalent to **mtspr 1,r**S)
2. Copy the contents of the LR to **r**S.
   mflr rS (equivalent to **mfspr r**S**,8**)
3. Copy the contents of **r**S to the CTR.
   mtctr rS (equivalent to **mtspr 9,r**S)

F.9 Recommended Simplified Mnemonics

This section describes some of the most commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

F.9.1 No-Op (nop)

Many PowerPC instructions can be coded in a way that, effectively, no operation is performed. An additional mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that triggers the following:

nop (equivalent to **ori 0,0,0**)

F.9.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD**,**value (equivalent to **addi r**D**,0,**value)

2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD**,**value (equivalent to **addis r**D**,0,**value)

F.9.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction which normally requires a separate register and immediate operands.

la rD**,**d(**r**A) (equivalent to **addi r**D**,r**A**,**d)

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in register **r***v*, and the assembler has been told to use register **r***v* as a base for references to the data structure containing *v*, the following line causes the address of *v* to be loaded into register **r**D:

la rD**,***v* (equivalent to **addi r**D**,r***v***,**d*v*

F.9.4 Move Register (mr)

Several PowerPC instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA**,r**S (equivalent to **or r**A**,r**S**,r**S)

F.9.5 Complement Register (not)

Several PowerPC instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not rA**,r**S (equivalent to **nor r**A**,r**S**,r**S)

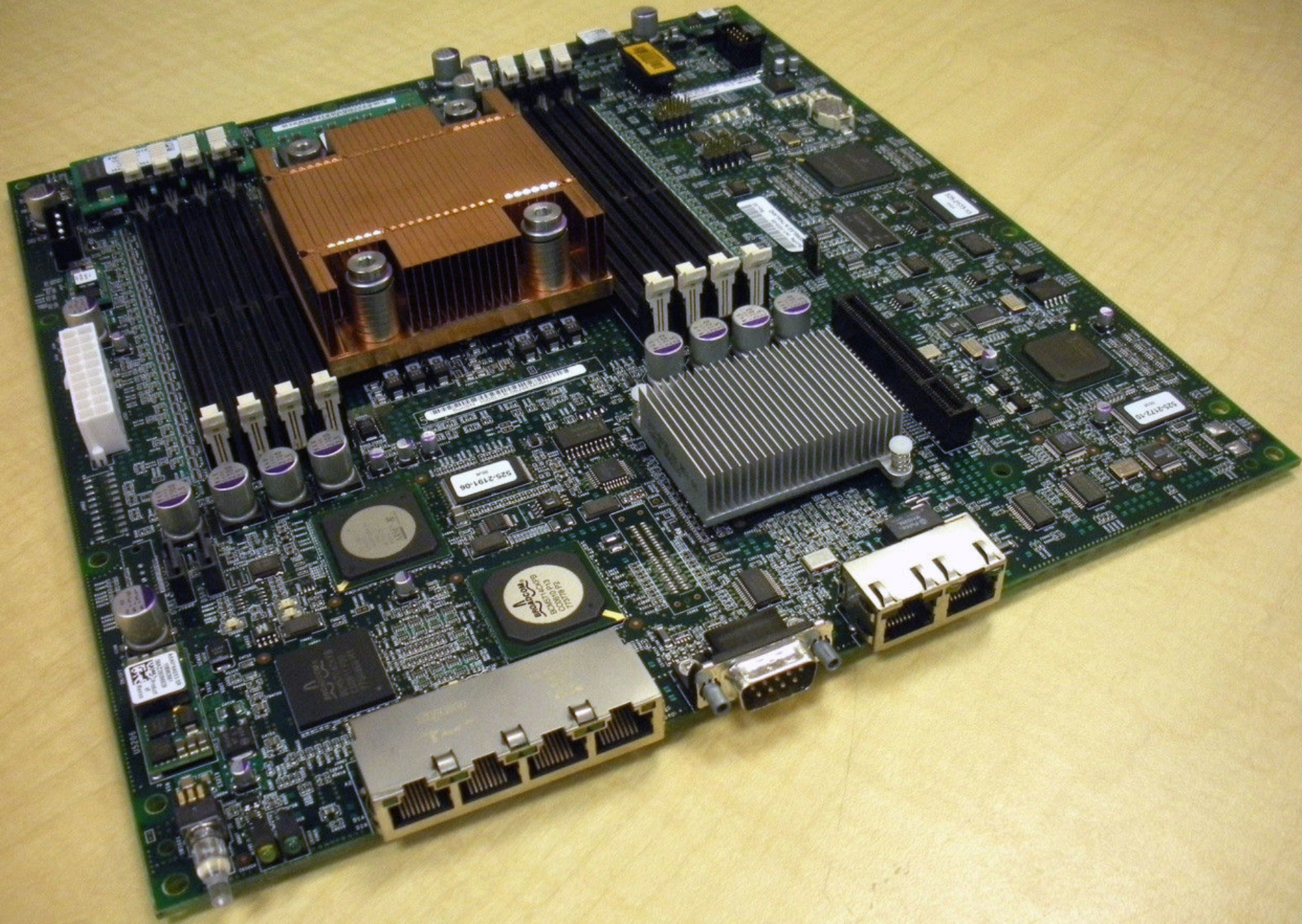
F

F.9.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the condition register, using the same syntax as the **mfcr** instruction.

mtcr rS (equivalent to **mtcrf** 0xFF**,r**S)

Платка на компютър „Sun Fire T1000“ с RISC-архитектура „SPARC“ = „Scalable Processor ARChitecture“



Handout 6: SPARC Assembly Quick Reference

References:

- Richard Paul. 1999. *SPARC Architecture, Assembly Language Programming, and C, 2nd ed*. Prentice-Hall.
- *SPARC assembly language reference* (on-line) accessible from the course page (Links/Compilation area).

1. Registers

Category	Register	Synonyms	Note
Global	**%g0**	%r0	Zero always
	%g1 ~ **7**	%r1 ~ 7	General-purpose
Out	**%o0** ~ **5**	%r8 ~13	Subroutine arguments (to be set before calling) Also as local variable
	%sp (%o6)	%r14	Stack pointer
	%o7	%r15	Subroutine return address
Local	**%l0** ~ **7**	%r16 ~ 23	General-purpose
In	**%i0** ~ **5**	%r24 ~ 29	Subroutine arguments (available after being called)
	%fp (%i6)	%r30	Frame pointer
	%i7	%r31	Subroutine return address

2. Instructions

Instruction [*Operands*] Notes

Arithmetic

add Reg_1, Reg_2/*Val*, Reg_{dest}
add... other addition
sub Reg_1, Reg_2/*Val*, Reg_{dest}
subcc Reg_1, Reg_2/*Val*, Reg_{dest} set cond code
sub... other subtraction

Logical

and Reg_1, Reg_2/*Val*, Reg_{dest}
and... other **and**
or Reg_1, Reg_2/*Val*, Reg_{dest}
or... other **or**

Shift

sll Reg_1, Reg_2/*Val*, Reg_{dest} shift left
srl Reg_1, Reg_2/*Val*, Reg_{dest} shift right

Load/Store

ld *Addr*, *Reg* from memory
ld... other load
st *Reg*, *Addr* to memory
st... other store

Branch

ba *Label* always
b *Label* always[1]
be *Label* on equal
bne *Label* on not equal
bl *Label* on <
ble *Label* on ≤
bg *Label* on >
bge *Label* on ≥
b... other branch

Others

nop no operation
ta *Addr* trap always
t... other trap
jmp *Addr* jump
call *Label* function call
call .[**u**]**mul** multiplication[2]
ret return
clr *Reg* clear
set *Val*, *Reg* set value
sethi *Val*, *Reg* set high 22 bits
mov Reg_1/*Val*, Reg_2 move
cmp Reg_1, Reg_2/*Val* compare
save Reg_1, Reg_2/*Val*, Reg_{dest} save window[3]
restore Reg_1, Reg_2/*Val*, Reg_{dest} restore window

[1] **b** is generated by **gcc**. Synonymous to **ba**.
[2] Operation (signed/unsigned): **%o0** * **%o1** = **%o0**
[3] SPARC register window

<End>

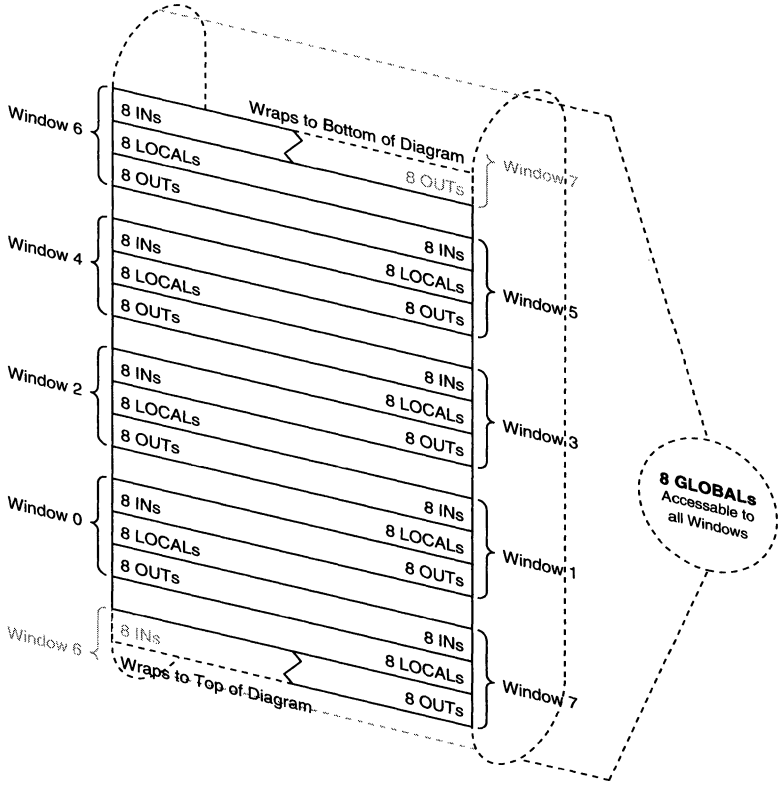


Figure 2-2. Register Windows

Register Addressing

There are up to three address fields associated with a SPARC instruction. In the case of a three-address instruction, these are the *rs1* field, the *rs2* field, and the *rd* field. Rs1 and rs2 are the *logical register addresses* of the two source operands of the instruction while rd is the logical register address of the destination operand.

SPARC Assembler Language Programming in a Nutshell

Registers

General purpose

- 32 registers %r0..%r31 - 32 bits each
- Used as four groups of 8 registers each:

Reg#	Group	Assem. Syntax
0..7	Globals	%g0..%g7
8..15	Outs	%o0..%o7 (letter "oh")
16..23	Locals	%l0..%l7 (letter "ell")
24..31	Ins	%i0..%i7

- Register %g0 is a constant 0 value. It cannot be altered.
- There is only one set of global registers, shared by all.
- There are 2..32 other *sets* of 16 registers (32..512 regs).
- At any time, you have a "*window*" into these sets.
- %l0..%l7 and %i0..%i7 (%r16..%r31) are the current set.
- %o0..%o7 (%r8..%r15) are the In regs from the next set.
- SAVE/RESTORE instructions switch to the next/previous set.
- %i7 and %o7 are used for return addresses. DO NOT USE!
- %i6 and %o6 are used for stack/frame pointers.
- Aliases: %sp = %o6 %fp = %i6.

Special purpose

PC - Program Counter - address of executing instruction
nPC - Next Program Counter - address of instruction being fetched
%y - Holds high-order half of product/dividend for multiply/divide
PSR - Processor State Register - includes many fields, including
iCC - integer condition code (N Z V C bits)
CWP - current window pointer - selects the register set

Assembly Language Syntax

- Comments: ! to end of line, or between /* and */
- Labels are case-sensitive, always terminated by a colon (:)
- Labels may contain _ $ and . characters.
- Labels that begin with _ $ and . are special. DO NOT USE!
- Use 0x to prefix hexadecimal values (e.g. 0xF123ABC).
- Pseudo-operations (assembler directives) always begin with ".".
- Either 'chars' or "chars" may be used for ascii characters.
- Special symbols (such as register names) begin with "%".

Addressing Modes

There are no orthogonal "modes". There are only a few combinations of operands allowed in the basic instructions, which fall (with very few exceptions) into the following categories:

1) Load/Store (memory) instructions:
opcode [%reg+const],%reg
opcode [%reg+%reg],%reg
(reverse the operand order for Store instructions)
2) Arithmetic/Logical/Shift instructions:
opcode %reg,%reg,%reg
opcode %reg,const,%reg
3) Branch instructions:
opcode address

- In the above, "const" is a 13-bit signed integer (-4096..4095),
- "address" is a 22-bit signed integer (but branch instructions add two low-order 0's since all instructions are word-aligned, giving, in effect, 24 bits (+/- 8 Meg)).
- The assembler accepts [%reg] which it turns into [%reg+%g0].
- Examples:

```
LD      [%i3+%L1],%L3       ! Loads reg L3
ST      %L2,[%sp - 48]      ! Stores L2 in stack
LD      [%L1],%L2           ! L1 contains an address
ADD     %L1,%L2,%L3         ! L1+L2 placed in L3
ADD     %g1,48,%g1          ! add 48 to reg G1
```

Basic Instruction Set

Load/Store (memory) instructions

- *Only* these instructions reference data stored in memory.

LDUB	Load Unsigned Byte (padded with 0's)
LDSB	Load Signed Byte (sign-extended)
LDUH	Load Unsigned Halfword (padded with 0's)
LDSH	Load Signed Halfword (sign-extended)
LD	Load (a word - 32 bits - 4 bytes)
LDD	Load Doubleword (register # must be even)
STB	Store Byte
STH	Store Halfword
ST	Store (a word)
STD	Store Doubleword (register # must be even)

- After a load, the data always occupies the entire register(s).
- All addresses must be aligned (by 2/4/8 for half/word/double).

Arithmetic/Logical

ADD	addition (Op1+Op2->Op3)
SUB	subtraction (Op1-Op2->Op3)
AND	bitwise AND (Op1^Op2->Op3)
ANDN	bitwise AND, Op2 inverted (Op1^Op2'->Op3)
OR	bitwise OR (Op1vOp2->Op3)
ORN	bitwise OR, Op2 inverted (Op1vOp2'->Op3)
XOR	bitwise exclusive OR (Op1 exor Op2->Op3)
XNOR	bitwise exclusive NOR (Op1 exor Op2'->Op3)

- To set the condition code (iCC), add "cc" to any of the above.
- Example: SUBcc %L1,%L2,%G0 !compare L1 to L2
- All operations always use the full 32 bits of the register(s).

SETHI/NOP

- Has a special instruction format and syntax:
SETHI const,%reg
- Places the 22-bit constant in the *high-order* 22 bits of %reg.
- The low-order 10 bits of the register are cleared to 0's.
- The special assembler functions %hi(x) and %lo(x) will give the high-order 22 bits and low-order 10 bits of any constant x.
- To place a 32-bit constant X (such as a fixed address) in a reg:

```
SETHI   %hi(X),%reg     ! Set the high 22 bits
OR      %reg,%lo(X),%reg! Insert the low 10 bits
```

- The assembler will accept SET X,%reg instead of the above.
- NOP (no operation) (takes no parameters) is really SETHI 0,%G0

SAVE/RESTORE

- Identical to ADD except that SAVE/RESTORE switch to the next/previous set of registers.
- The addition is typically used to modify the stack pointer.
- Example:

```
SAVE    %sp,-96,%sp
! switch register windows and allocate 96 bytes
```

- RESTORE with no operands is really RESTORE %g0,%g0,%g0

Branch

BA	Branch Always
BE	Branch Equal
BNE	Branch Not Equal
BL	Branch Less
BLE	Branch Less or Equal
BG	Branch Greater
BGE	Branch Greater or Equal

- The above (except BA) should be used following operations on *signed* arithmetic only. There are others for *unsigned* arithmetic.

SPARC Assembler Language Programming in a Nutshell

Delayed Branching

• All branches (including the one caused by CALL, below) take place *after* execution of the following instruction.
• The position immediately after a branch is the "delay slot" and the instruction found there is the "delay instruction".
• If possible, place a useful instruction in the delay slot (one which can safely be done whether or not a conditional branch is taken).
• If not, place a NOP in the delay slot.
• *Never* place any other branch instruction in a delay slot.
• Do not use SET in a delay slot (only half of it is really there).

Calling Library Routines

• Use

```
CALL    address
```

• Place the parameters (up to 6) in %o0..%o5 first.
• A scalar result, if any, will be returned in %o0.
• Any standard C library routine may be used.
• Example of using "printf" for output:

```
SET     outstr,%o0
CALL    printf
ADD     %L1,%G0,%o1 !done before CALL
....
outstr: .asciz  "Register L1 contains %d\n"
```

• Example of using "scanf" for input:

```
SET     instr,%o0
SET     data,%o1        !ADDRESS of data
CALL    scanf
NOP                     ! Don't put SET here!
...
instr:  .asciz  "%d"
data:   .word
```

Pseudo-operations (Assembler directives)

• Forcing proper boundary alignment:

```
.align  2/4/8
```

• Selecting the code segment or data segment:

```
.section ".text"   ! for the code
.section ".data"   ! for the data
```

• Allocating blocks of memory

```
label:  .skip   num-of-bytes
```

• Allocating initialized memory

```
.byte   value[,value,value...]
.half   value[,value,value...]
.word   value[,value,value...]
```

(Don't use .double - it gives *real* values!)

```
.ascii  "characters"
.asciz  "characters"    !adds a NULL (\0) at the end
```

• Declaring symbols to be used externally

```
.global symbol
```

Minimal Linkage

• The environment expects at least 92 bytes (23 words) of temporary storage on the stack. (Register window overflow/underflow, other interrupts, and library routines may need and use this memory.) The stack pointer must be doubleword aligned (divisible by 8), so 96 is the minimum allocation.
• The entry point into the program is the global symbol "main".
• The minimal linkage for a program is therefore

```
        .global main
main:   save    %sp,-96,%sp
```

• To return to the operating system, use

```
ret     ! standard subprogram return instruction
restore ! restore the old set of registers (delay slot)
```

Synthetic Instructions

As a RISC architecture, SPARC lacks many common instructions found on other processors. For convenience, the assembler provides many *synthetic instructions* which it translates for you.

Synthetic Instruction		Assembled As	
clr	%reg	or	%g0,%g0,%reg
clr	[address]	st	%g0,[address]
clrh	[address]	sth	%g0,[address]
clrb	[address]	stb	%g0,[address]
cmp	%reg,%reg	subcc	%reg,%reg,%g0
cmp	%reg,const	subcc	%reg,const,%g0
dec	%reg	sub	%reg,1,%reg
deccc	%reg	subcc	%reg,1,%reg
inc	%reg	add	%reg,1,%reg
inccc	%reg	addcc	%reg,1,%reg
mov	%reg,%reg	or	%g0,%reg,%reg
mov	const,%reg	or	%g0,const,%reg
not	%reg	xnor	%reg,%g0,%reg
neg	%reg	sub	%g0,%reg,%reg
restore		restore	%g0,%g0,%g0
ret		jmpl	%i7+8,%g0
set	const22,%reg	sethi	%hi(const22),%reg
		or	%reg,%lo(const22),%reg
tst	%reg	orcc	%reg,%g0,%g0

Sample Program

```
!==== Minimal prologue ====
        .section        ".text"
        .global         main
        .align          4
main:   save            %sp,-96,%sp !minimum!
!====================

! Call a routine (printf) to print a message.
        set     string,%o0
        call    printf
        nop     !Always fill the "delay slot"!

!====Standard epilogue ====
        ret             !Return to the OS.
        restore         !Delay slot - done first
!====================

! The data section
        .section        ".data"
string: .asciz          "Hello, world!\n"
```

Princeton University
COS 217: Introduction to Programming Systems
SPARC Assembly Language Summary

Abbreviations Used in Instruction Descriptions	
rs	Source register
rd	Destination register
ris	Source register, or Source immediate constant represented in machine language using 13 bits
is	Source immediate constant represented in machine language using 13 bits
is22	Source immediate constant represented in machine language using 22 bits
is32	Source immediate constant represented in machine language using 32 bits
addr	Memory address expressed in one of these formats: rs rs + ris rs – is is + rs is
label	Label represented in machine language as a 22-bit displacement relative to %pc
label30	Label represented in machine language as a 30-bit displacement relative to %pc
Z	Zero condition code
N	Negative condition code
V	oVerflow condition code
C	Carry condition code
r[31]	Bit 31 of r

Load and StoreInstructions (Format 3)	
ldub [addr], rd	Load an unsigned byte from addr into rd
ldsb [addr], rd	Load a signed byte from addr into rd
lduh [addr], rd	Load an unsigned halfword from addr into rd
ldsh [addr], rd	Load a signed halfword from addr into rd
ld [addr], rd	Load a word from addr into rd
ldd [addr], rd	Load a doubleword from addr and addr+1 into rd and rd+1
swap [addr], rd	Swap the contents of addr and rd
stb rs, [addr]	Store a byte from rs into addr
sth rs, [addr]	Store a halfword from rs into addr
st rs, [addr]	Store a word from rs into addr
std rs, [addr]	Store a doubleword from rs and rs+1 into addr and addr+1
clrb [addr]	Synthetic instruction for: stb %g0, addr
clrh [addr]	Synthetic instruction for: sth %g0, addr
clr [addr]	Synthetic instruction for: st %g0, addr

Shift Instructions (Format 3)	
sll rs, ris, rd	rd = rs << ris
srl rs, ris, rd	rd = rs >> ris (fill with zeros)
sra rs, ris, rd	rd = rs >> ris (fill by extending sign)

Arithmetic Instructions (Format 3)	
add rs, ris, rd	rd = rs+ ris
addcc rs, ris, rd	rd = rs + ris N = rd[31] == 1 Z = rd == 0 V = (rs[31] & ris[31] & ~rd[31]) | (~rs[31] & ~ris[31] & rd[31]) C = (rs[31] & ris[31]) | (~rd[31] & (rs[31] | ris[31]))
addx rs, ris, rd	rd = rs – ris + C
addxcc rs, ris, rd	rd = rs + ris + C N = rd[31] == 1 Z = rd == 0 V = (rs[31] & ris[31] & ~rd[31]) | (~rs[31] & ~ris[31] & rd[31]) C = (rs[31] & ris[31]) | (~rd[31] & (rs[31] | ris[31]))
sub rs, ris, rd	rd = rs – ris
subcc rs, ris, rd	rd = rs – ris N = rd[31] == 1 Z = rd == 0 V = (rs[31] & ~ris[31] & ~rd[31]) | (~rs[31] & ris[31] & rd[31]) C = (~rs[31] & ris[31]) | (rd[31] & (~rs[31] | ris[31]))
subx rs, ris, rd	rd = rs – ris – C
subxcc rs, ris, rd	rd = rs – ris - C N = rd[31] == 1 Z = rd == 0 V = (rs[31] & ~ris[31] & ~rd[31]) | (~rs[31] & ris[31] & rd[31]) C = (~rs[31] & ris[31) | (rd[31] & (~rs[31] | ris[31]))
neg rs, rd	Synthetic instruction for: sub %g0, rs, rd
neg rd	Synthetic instruction for: sub %g0, rd, rd
inc rd	Synthetic instruction for: add rd, 1, rd
inc is, rd	Synthetic instruction for: add rd, is, rd
inccc rd	Synthetic instruction for: addcc rd, 1, rd
inccc is, rd	Synthetic instruction for: addcc rd, is, rd
dec rd	Synthetic instruction for: sub rd, 1, rd
dec is, rd	Synthetic instruction for: sub rd, is, rd
deccc rd	Synthetic instruction for: subcc rd, 1, rd
deccc is, rd	Synthetic instruction for: subcc rd, is, rd
cmp rs, ris	Synthetic instruction for: subcc rs, ris, %g0

Logical Instructions (Format 3)	
and rs, ris, rd	rd = rs & ris
andcc rs, ris, rd	rd = rs & ris; N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
andn rs, ris, rd	rd = rs & ~ris
andncc rs, ris, rd	rd = rs & ~ris; N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
or rs, ris, rd	rd = rs | ris
orcc rs, ris, rd	rd = rs | ris; N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
orn rs, ris, rd	rd = rs | ~ris
orncc rs, ris, rd	rd = rs | ~ris; N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
xor rs, ris, rd	rd = rs ^ ris
xorcc rs, ris, rd	rd = rs ^ ris; N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
xnor rs, ris, rd	rd = ~(rs ^ ris)
xnorcc rs, ris, rd	rd = ~(rs ^ ris); N = rd[31] == 1; Z = rd == 0; V = 0; C = 0
clr rd	Synthetic instruction for: or %g0, %g0, rd
mov ris, rd	Synthetic instruction for: or %g0, ris, rd
tst rs	Synthetic instruction for: orcc %g0, rs, %g0
btst ris, rs	Synthetic instruction for: andcc rs, ris, %g0
bset ris, rd	Synthetic instruction for: or rd, ris, rd
bclr ris, rd	Synthetic instruction for: andn rd, ris, rd
btog ris, rd	Synthetic instruction for: xor rd, ris, rd
not rs, rd	Synthetic instruction for: xnor rs, %g0, rd
not rd	Synthetic instruction for: xnor rd, %g0, rd

Integer Branch Instructions (Format 2)

	Unconditional branching:
ba{,a} label	Branch to label always
bn{,a} label	Branch to label never
	Signed number branching:
bl{,a} label	Branch to label if N ^ V
ble{,a} label	Branch to label if Z | (N ^ V)
bge{,a} label	Branch to label if ~(N ^ V)
bg{,a} label	Branch to label if ~(Z | (N ^ V))
	Unsigned number branching:
blu{,a} label	Synonym for: bcs{,a} label
bleu{,a} label	Branch to label if C | Z
bgeu{,a} label	Synonym for: bcc{,a} label
bgu{,a} label	Branch to label if ~(C | Z)
	Individual condition code branching:
be{,a} label	Branch to label if Z
bne{,a} label	Branch to label if ~Z
bpos{,a} label	Branch to label if ~N
bneg{,a} label	Branch to label if N
bcs{,a} label	Branch to label if C
bcc{,a} label	Branch to label if ~C
bvs{,a} label	Branch to label if V
bvc{,a} label	Branch to label if ~V
bz{,a} label	Synonym for: be{,a} label
bnz{,a} label	Synonym for: bne{,a} label

Control Instructions (Format 3)

jmpl addr, rd	Store %pc in rd, and jump to addr
jmp addr	Synthetic instruction for: jmpl addr, %g0
call ris	Synthetic instruction for: jmpl ris, %o7
ret	Synthetic instruction for: jmpl %i7 + 8, %g0
retl	Synthetic instruction for: jmpl %o7 + 8, %g0
save rs, ris, rd	Save register window. rd = rs + ris
restore rs, ris, rd	Restore register window. rd = rs + ris
restore	Synthetic instruction for: restore %g0, %g0, %g0

Control Instructions (Format 2)

nop	No operation
sethi is22, rd	Set the high-order 22 bits of rd to is22, and set the low-order 10 bits of rd to 0
set is32, rd	Synthetic instruction for: sethi %hi(is32), rd or rd, %lo(is32), rd

Control Instructions (Format 1)

call label30	Store %pc in %o7, and jump to label30

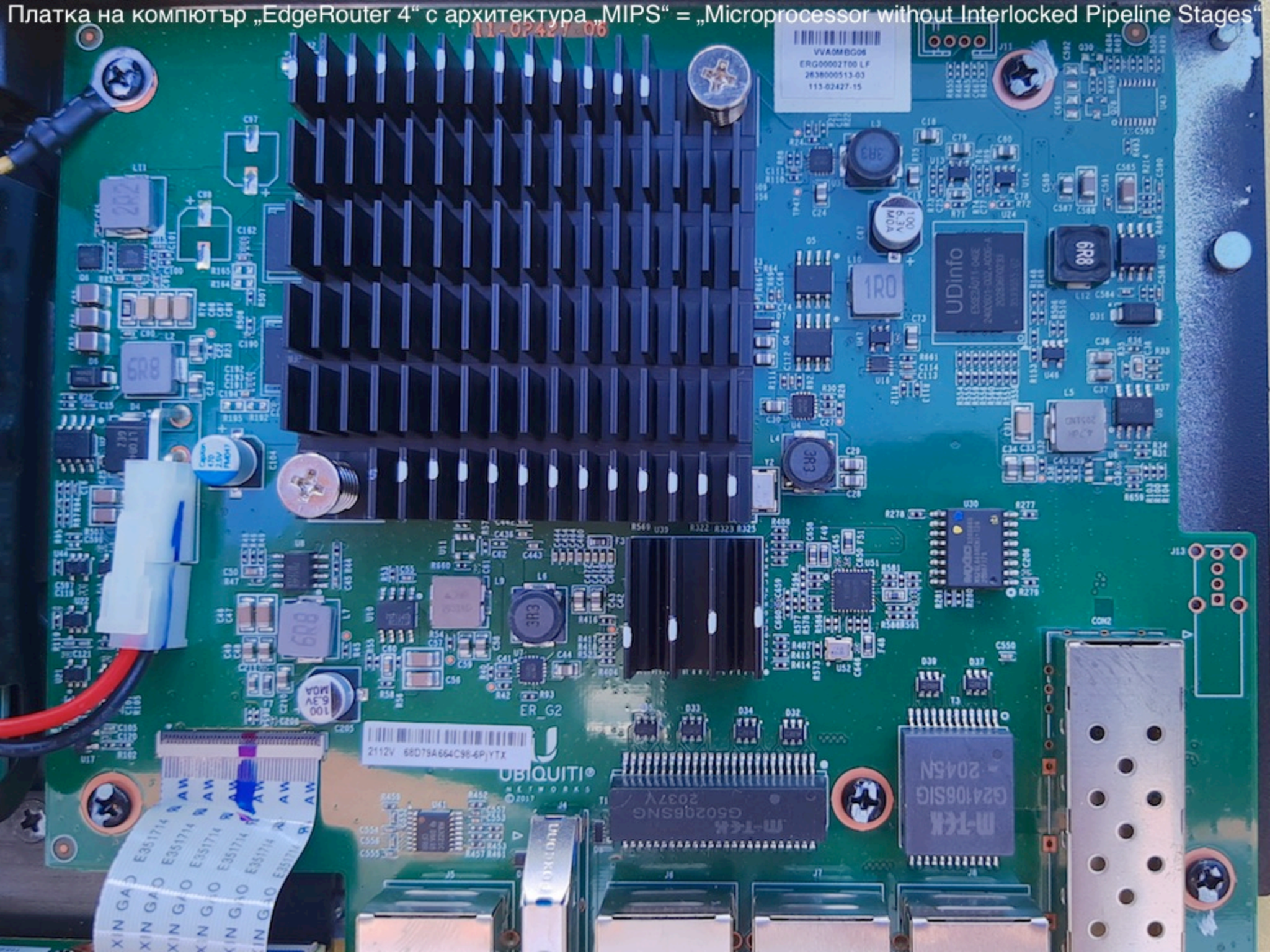
Trap Instructions (Format 3)

ta addr	Trap always to addr (typically 0)
...	

Floating-Point Instructions (Format 3)

...	

Pseudo-Ops	
symbol:	Define a label named symbol whose value is the current location counter.
symbol = expr	Define an assembler constant. The assembler replaces symbol with the value of expr.
.section ".text"	Add the following code to the text section. The text section contains executable code.
.section ".data"	Add the following code to the data section. The data section contains program-initialized read-write data.
.section ".bss"	Add the following code to the bss section. The bss section contains read-write data that is initialized to 0.
.section ".rodata"	Add the following code to the rodata section. The rodata contains read-only data.
.skip n	Skip n bytes of memory.
.align n	Increase the location counter so its value is evenly divisible by n.
.byte bytevalue1, bytevalue2, ...	Allocate memory containing bytevalue1, bytevalue2, ...
.half halfvalue1, halfvalue2, ...	Allocate memory containing halfvalue1, halfvalue2, ...
.word wordvalue1, wordvalue2, ...	Allocate memory containing wordvalue1, wordvalue2, ...
.ascii "string1", "string2", ...	Allocate memory containing the characters from string1, string2, ...
.asciz "string1", "string2", ...	Allocate memory containing string1, string2, ... where each string is NULL terminated.
.common symbol, size	Declare the name and size of a common area of memory to be shared by multiple object files.
.global symbol1, symbol2, ...	Mark symbol1, symbol2, ... so they are available to the linker.
.empty	Suppress assembler warnings about the next instruction's presence in a delay slot.



Платка на компютър „EdgeRouter 4“ с архитектура „MIPS“ = „Microprocessor without Interlocked Pipeline Stages“

MIPS 32 Instruction Reference (10/20/2017)

Arithmetic

Inst	Format	Type	Function
ADD	RD, RS, RT	R	RD = RS + RT (Overflow trap)
ADDI	RT, RS, Imm16	I	RT = RS + se(Imm16) (Overflow trap)
ADDIU	RT, RS, Imm16	I	RT = RS + se(Imm16)
ADDU	RD, RS, RT	R	RD = RS + RT
CLO	RD, RS	R	RD = countleadingones(RS)
CLZ	RD, RS	R	RD = countleadingzeros(RS)
LA	RD, label	P	RD = address(label)
LI	RD, Imm32	P	RD = Imm32
LUI	RT, Imm16	I	RT = Imm16 << 16
SEB	RD, RT	R	RD = se(RT7:0)
SEH	RD, RT	R	RD = se(RT15:0)
SUB	RD, RS, RT	R	RD = RS - RT (Overflow trap)
SUBU	RD, RS, RT	R	RD = RS - RT

Shift and Rotate

ROTR	RD, RT, sa	R	RD = rotr(RT, sa)
ROTRV	RD, RT, RS	R	RD = rotr(RT, RS mod 32)
SLL	RD, RT, sa	R	RD = RT << sa
SLLV	RD, RT, RS	R	RD = RT << (RS mod 32)
SRA	RD, RT, sa	R	RD = RT >>arith sa
SRAV	RD, RT, RS	R	RD = RT >>arith (RS mod 32)
SRL	RD, RT, sa	R	RD = RT >> sa
SRLV	RD, RT, RS	R	RD = RT >> (RS mod 32)

Logical

AND	RD, RS, RT	R	RD = RS & RT
ANDI	RT, RS, Imm16	I	RT = RS & ze(Imm16)
NOR	RD, RS, RT	R	RD = ~(RS | RT)
OR	RD, RS, RT	R	RD = RS | RT
ORI	RT, RS, Imm16	I	RT = RS | ze(Imm16)
XOR	RD, RS, RT	R	RD = RS exor RT
XORI	RT, RS, Imm16	I	RT = RS exor ze(Imm16)

Bit Field

EXT	RT, RS, sa, rd	R	extract rd bits from RS at bit sa
INS	RT, RS, sa, rd	R	insert rd lsbs of RS into RT at bit sa
WSBH	RD, RT	R	RD = RT23:16 :: RT31:24 :: RT7:0 :: RT15:8

Condition Testing

SLT	RD, RS, RT	R	RD = (RS < RT) ? 1 : 0 (signed)
SLTI	RT, RS, Imm16	I	RT = (RS < se(Imm16)) ? 1 : 0 (signed)
SLTIU	RT, RS, Imm16	I	RT = (RS < se(Imm16)) ? 1 : 0 (uns)
SLTU	RD, RS, RT	R	RD = (RS < RT) ? 1 : 0 (uns)
MOVN	RD, RS, RT	R	If RT != 0, RD = RS
MOVZ	RD, RS, RT	R	If RT == 0, RD = RS

Multiply and Divide

DIV	$0, RS, RT	R	LO = RS / RT; HI = rem(RS, RT) (signed)
DIVU	$0, RS, RT	R	LO = RS / RT; HI = rem(RS, RT) (uns)
MADD	RS, RT	R	HI,LO += RS x RT (signed)
MADDU	RS, RT	R	HI,LO += RS x RT (uns)
MSUB	RS, RT	R	HI,LO -= RS x RT (signed)
MSUBU	RS, RT	R	HI,LO -= RS x RT (uns)
MUL	RD, RS, RT	R	RD = RS x RT (signed)
MULT	RS, RT	R	HI,LO = RS x RT (signed)
MULTU	RS, RT	R	HI,LO = RS x RT (uns)
MFHI	RD	R	RD = HI
MFLO	RD	R	RD = LO
MTHI	RS	R	HI = RS
MTLO	RS	R	LO = RS

Jumps and Branches

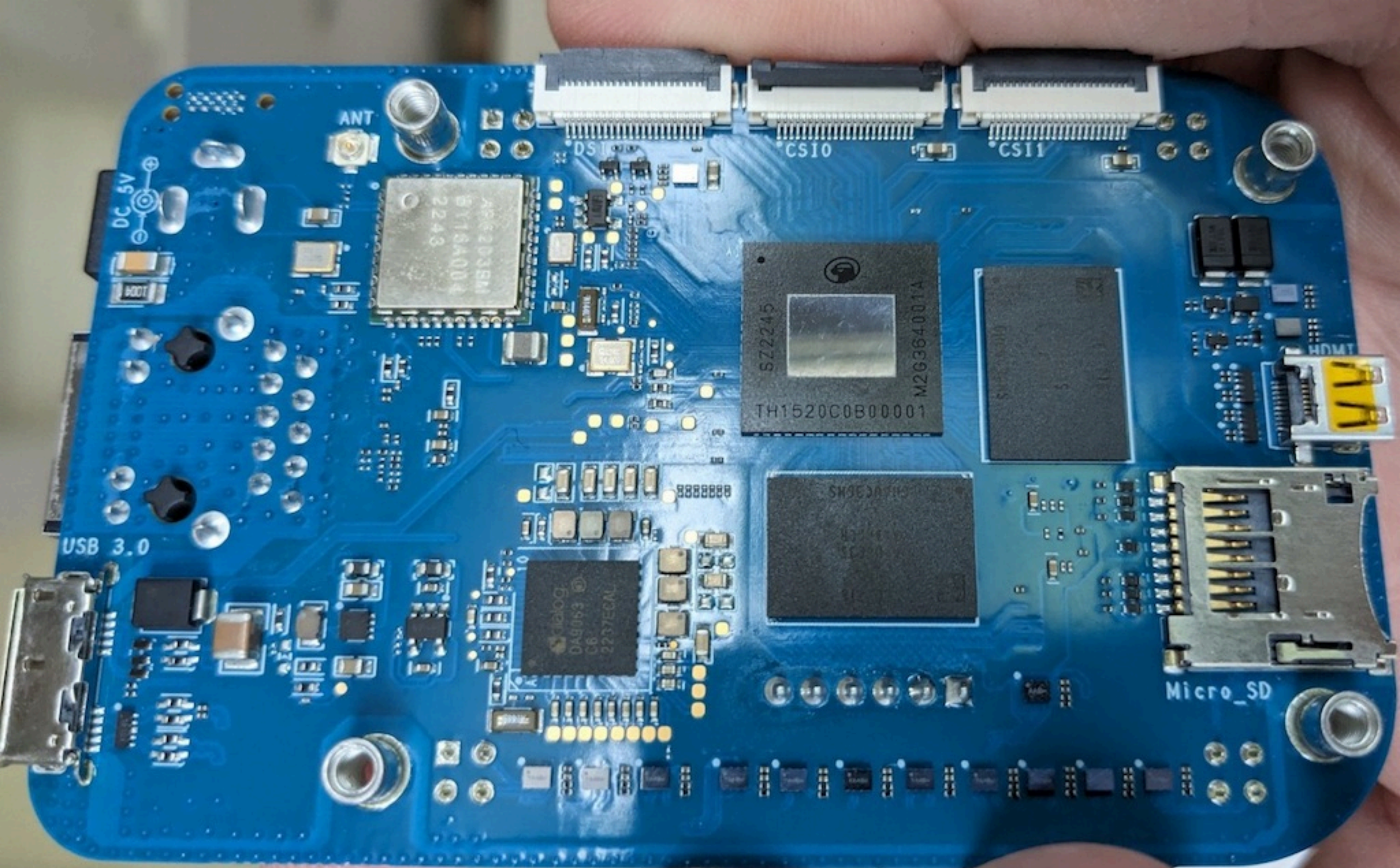
BEQ	RS, RT, label	I	If RS == RT, PC += 4 + se(Off16 << 2)
BGEZ	RS, label	I	If RS >= 0, PC += 4 + se(Off16 << 2)
BGEZAL	RS, label	I	RA = PC + 8; IF RS >= 0, PC += 4 + se(Off16 << 2)
BGTZ	RS, label	I	If RS > 0, PC += 4 + se(Off16 << 2)
BLEZ	RS, label	I	If RS <= 0, PC += 4 + se(Off16 << 2)
BLTZ	RS, label	I	If RS < 0, PC += 4 + se(Off16 << 2)
BLTZAL	RS, label	I	RA = PC + 8; IF RS < 0, PC += 4 + se(Off16 << 2)
BNE	RS, RT, label	I	If RS != RT, PC += 4 + se(Off16 << 2)
B	label	P	PC += 4 + se(Off16 << 2)
BLT	RS, RT, label	P	If RS < RT (sgn), PC += 4 + se(Off16 << 2)
BLE	RS, RT, label	P	If RS <= RT (sgn), PC += 4 + se(Off16 << 2)
BGT	RS, RT, label	P	If RS > RT (sgn), PC += 4 + se(Off16 << 2)
BGE	RS, RT, label	P	If RS >= RT (sgn), PC += 4 + se(Off16 << 2)
BLTU	RS, RT, label	P	If RS < RT (uns), PC += 4 + se(Off16 << 2)
BLEU	RS, RT, label	P	If RS <= RT (uns), PC += 4 + se(Off16 << 2)
BGTU	RS, RT, label	P	If RS > RT (uns), PC += 4 + se(Off16 << 2)
BGEU	RS, RT, label	P	If RS >= RT (uns), PC += 4 + se(Off16 << 2)
J	label	J	PC = (PC + 4)31:28 :: Addr26 << 2
JAL	label	J	RA = PC + 8; PC = (PC + 4)31:28 :: Addr26 << 2
JALR	RD, RS	R	RD = PC + 8; PC = RS
JR	RS	R	PC = RS

Loads and Stores

LB	RT, Off16(RS)	I	RT = se(Mem8(RS + se(Off16)))
LBU	RT, Off16(RS)	I	RT = ze(Mem8(RS + se(Off16)))
LH	RT, Off16(RS)	I	RT = se(Mem16(RS + se(Off16)))
LHU	RT, Off16(RS)	I	RT = ze(Mem16(RS + se(Off16)))
LW	RT, Off16(RS)	I	RT = Mem32(RS + se(Off16))
SB	RT, Off16(RS)	I	Mem8(RS + se(Off16)) = RT7:0
SH	RT, Off16(RS)	I	Mem16(RS + se(Off16)) = RT15:0
SW	RT, Off16(RS)	I	Mem32(RS + se(Off16)) = RT
LW	RT, label	P	RT = Mem32(label)
SW	RT, label	P	Mem32(label) = RT

Registers			Preserved
0	zero	Always equal to zero	na
1	at	Assembler temporary; used by the assembler	N
2-3	v0-v1	Return value from a function call	N
4-7	a0-a3	First four parameters for a function call	N
8-15	t0-t7	Temporary variables	N
16-23	s0-s7	Function variables	Y
24-25	t8-t9	Two more temporary variables	N
26-27	k0-k1	Kernel use registers; may change unexpectedly	N
28	gp	Global pointer	Y
29	sp	Stack pointer	Y
30	fp/s8	Stack frame pointer or subroutine variable	Y
31	ra	Return address of the last subroutine call	Y

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


ANT
DC 5V
DS
CS10
CS11
SZ2245
TH1520C0B00001
M2G364001A
HDMI
USB 3.0
Micro_SD

RISC-V Reference

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

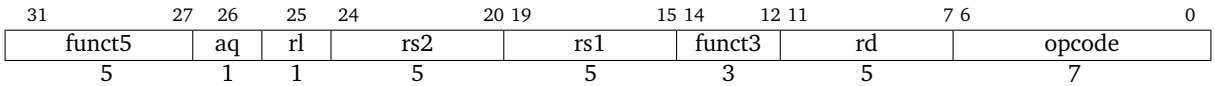
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

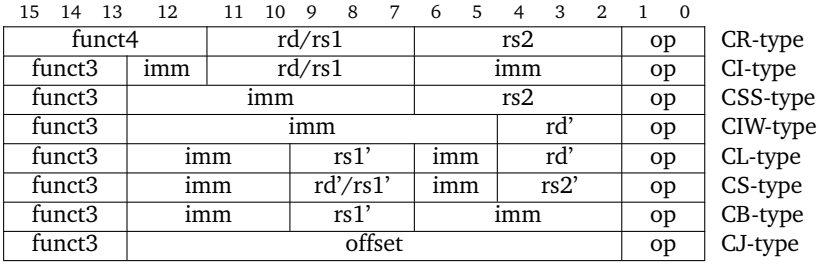


Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

RV32C Compressed Extension



Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd', (4*imm)(rs1')
c.sw	Store Word	CS	00	110	sw rs1', (4*imm)(rs2')
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs', x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs', x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd', sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd', rd', imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd', rd', imm
c.andi	AND Imm	CB	01	100x10	andi rd', rd', imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd', rd', rs2'
c.or	OR	CS	01	10001110	or rd', rd', rs2'
c.xor	XOR	CS	01	10001101	xor rd', rd', rs2'
c.sub	SUB	CS	01	10001100	sub rd', rd', rs2'
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller