MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures

Spring 1998

6.004 Issued: 3/12/98

β Instruction Set Architecture Reference

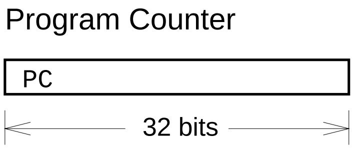
---

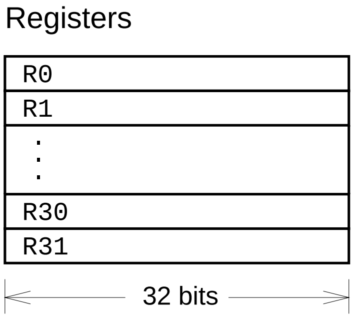
1. Introduction

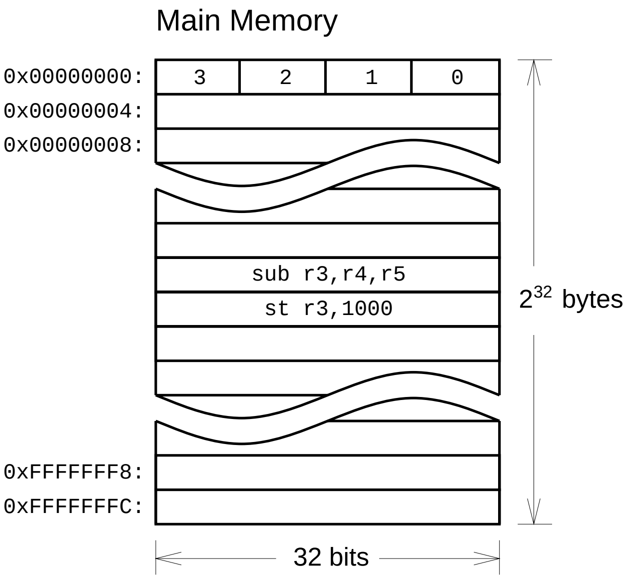
This handout is a reference guide for the β, the RISC processor design for 6.004. This is intended to be a complete and thorough specification of the programmer-visible state and instruction set.

2. Machine Model

The β is a general-purpose 32-bit architecture: all registers are 32 bits wide and when loaded with an address can specify any location in the byte-addressed memory. When read, register 31 is always 0; when written, it serves as a bit bucket.







3. INSTRUCTION ENCODING

Each β instruction is 32 bits long. All integer manipulation is between registers, with up to two source operands (one may be a sign-extended 16-bit literal), and one destination operand. Memory is referenced through load and store instructions which perform no other computation. Conditional branch instructions are separated from comparison instructions: branch instructions test the value of a register which can be the result of a previous compare instruction.

There are only two types of instruction encoding: *Without Literal* and *With Literal*. Instructions without literals include arithmetic and logical operations between two registers whose result is placed in a third register. Instructions with literals include all other operations.

Like all signed quantities on the β, an instruction's literal is represented in twos-complement.

3.1 Without Literal

31 26	25 21	20 16	15 11	10 0
Opcode	Rc	Ra	Rb	unused

3.2 With Literal

31 26	25 21	20 16	15 5 0
Opcode	Rc	Ra	Literal (signed)

4. INSTRUCTION SUMMARY

Below are listed the 32 β instructions and their 6-bit opcodes. For detailed instruction operations, see the following section.

Mnemonic	Opcode
ADD	0x20
ADDC	0x30
AND	0x28
ANDC	0x38
BEQ	0x1D
BNE	0x1E
CMPEQ	0x24
CMPEQC	0x34
CMPLE	0x26
CMPLEC	0x36
CMPLT	0x25
CMPLTC	0x35
DIV	0x23
DIVC	0x33
JMP	0x1B
LD	0x18
LDR	0x1F
MUL	0x22
MULC	0x32
OR	0x29
ORC	0x39
SHL	0x2C
SHLC	0x3C
SHR	0x2D
SHRC	0x3D
SRA	0x2E
SRAC	0x3E
SUB	0x21
SUBC	0x31
ST	0x19
XOR	0x2A
XORC	0x3A

5. INSTRUCTION SPECIFICATIONS

This section contains the specifications for the β instructions, listed alphabetically by mnemonic. No timing-dependent information is given: it is specifically assumed that there are no pathological timing interactions between instructions in this specification. Each instruction is considered atomic and is presumed to complete before the next instruction is executed. No assumptions are made about branch prediction, instruction prefetch, or memory caching.

5.1 *Add*

Usage: `ADD (Ra, Rb, Rc)`

Opcode:

100000	Rc	Ra	Rb	unused

Operation: $\mathtt{PC} \leftarrow \langle \mathtt{PC} \rangle + 4$
$\mathtt{Rc} \leftarrow \langle \mathtt{Ra} \rangle + \langle \mathtt{Rb} \rangle$

The contents of register `Ra` is added to the contents of register `Rb` and the 32-bit sum is written to `Rc`.

This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.2 *Addc*

Usage: `ADDC (Ra,` *literal*`, Rc)`

Opcode:

110000	Rc	Ra	literal

Operation: $\mathtt{PC} \leftarrow \langle \mathtt{PC} \rangle + 4$
$\mathtt{Rc} \leftarrow \langle \mathtt{Ra} \rangle + \textsc{sext}(literal)$

The contents of register `Ra` is added to *literal* and the 32-bit sum is written to `Rc`.

This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.3 *And*

Usage: `AND (Ra, Rb, Rc)`

Opcode:

101000	Rc	Ra	Rb	unused

Operation: $\mathtt{PC} \leftarrow \langle \mathtt{PC} \rangle + 4$
$\mathtt{Rc} \leftarrow \langle \mathtt{Ra} \rangle \wedge \langle \mathtt{Rb} \rangle$

This performs the bitwise boolean AND function between the contents of register `Ra` and the contents of register `Rb`. The result is written to register `Rc`.

5.4 *Andc*

Usage: `ANDC (Ra,` *literal*`, Rc)`

Opcode:

111000	Rc	Ra	literal

Operation: $\mathtt{PC} \leftarrow \langle \mathtt{PC} \rangle + 4$
$\mathtt{Rc} \leftarrow \langle \mathtt{Ra} \rangle \wedge \textsc{sext}(literal)$

This performs the bitwise boolean AND function between the contents of register `Ra` and *literal*. The result is written to register `Rc`.

5.5 *Beq/Bf*

Usage: `BEQ (Ra, label, Rc)`
`BF (Ra, label, Rc)`

Opcode:

`011101`	`Rc`	`Ra`	`literal`

Operation: $\mathit{literal} = ((\mathrm{OFFSET}(\mathit{label}) - \mathrm{OFFSET}(\text{current instruction})) \gg 2) - 1$
$\mathtt{PC} \leftarrow \langle\mathtt{PC}\rangle + 4$
$\mathit{EA} \leftarrow \langle\mathtt{PC}\rangle + 4 \cdot \mathit{SEXT}(\mathit{literal})$
$\mathtt{Rc} \leftarrow \langle\mathtt{PC}\rangle$
If $\langle\mathtt{Ra}\rangle = 0$ then $\mathtt{PC} \leftarrow \mathit{EA}$

The `PC` of the instruction following the BEQ instruction (the updated `PC`) is written to register `Rc`. Then the contents of register `Ra` are tested. If they are zero, the `PC` is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign extended to 32 bits, and added to the updated `PC` to form the target address.

5.6 *Bne/Bt*

Usage: `BT (Ra, label, Rc)`
`BNE (Ra, label, Rc)`

Opcode:

`011110`	`Rc`	`Ra`	`literal`

Operation: $\mathit{literal} = ((\mathrm{OFFSET}(\mathit{label}) - \mathrm{OFFSET}(\text{current instruction})) \gg 2) - 1$
$\mathtt{PC} \leftarrow \langle\mathtt{PC}\rangle + 4$
$\mathit{EA} \leftarrow \langle\mathtt{PC}\rangle + 4 \cdot \mathit{SEXT}(\mathit{literal})$
$\mathtt{Rc} \leftarrow \langle\mathtt{PC}\rangle$
If $\langle\mathtt{Ra}\rangle \neq 0$ then $\mathtt{PC} \leftarrow \mathit{EA}$

The `PC` of the instruction following the BNE instruction (the updated `PC`) is written to register `Rc`. Then the contents of register `Ra` are tested. If they are non-zero, the `PC` is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign extended to 32 bits, and added to the updated `PC` to form the target address.

5.7 *Cmpeq*

Usage: `CMPEQ (Ra, Rb, Rc)`

Opcode:

`100100`	`Rc`	`Ra`	`Rb`	`unused`

Operation: $\mathtt{PC} \leftarrow \langle\mathtt{PC}\rangle + 4$
If $\langle\mathtt{Ra}\rangle = \langle\mathtt{Rb}\rangle$ then $\mathtt{Rc} \leftarrow 1$ else $\mathtt{Rc} \leftarrow 0$

The contents of register `Ra` are compared to the contents of register `Rb`. If the two are equal, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.8 *Cmpeqc*

Usage: `CMPEQC (Ra,` *literal*`, Rc)`

Opcode:

110100	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
If $\langle\texttt{Ra}\rangle = \textit{SEXT}(\textit{literal})$ then $\texttt{Rc} \leftarrow 1$ else $\texttt{Rc} \leftarrow 0$

The contents of register `Ra` are compared to *literal*. If the two are equal, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.9 *Cmple*

Usage: `CMPLE (Ra, Rb, Rc)`

Opcode:

100110	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
If $\langle\texttt{Ra}\rangle \leq \langle\texttt{Rb}\rangle$ then $\texttt{Rc} \leftarrow 1$ else $\texttt{Rc} \leftarrow 0$

The contents of register `Ra` (as a signed quantity) are compared to the contents of register `Rb` (as a signed quantity). If the less-than-or-equal relationship holds, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.10 *Cmplec*

Usage: `CMPLEC (Ra,` *literal*`, Rc)`

Opcode:

110110	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
If $\langle\texttt{Ra}\rangle \leq \textit{SEXT}(\textit{literal})$ then $\texttt{Rc} \leftarrow 1$ else $\texttt{Rc} \leftarrow 0$

The contents of register `Ra` (as a signed quantity) are compared to *literal*. If the less-than-or-equal relationship holds, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.11 *Cmplt*

Usage: `CMPLT (Ra, Rb, Rc)`

Opcode:

100101	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
If $\langle\texttt{Ra}\rangle < \langle\texttt{Rb}\rangle$ then $\texttt{Rc} \leftarrow 1$ else $\texttt{Rc} \leftarrow 0$

The contents of register `Ra` (as a signed quantity) are compared to the contents of register `Rb` (as a signed quantity). If the less-than relationship holds, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.12 Cmpltc

Usage: `CMPLTC (Ra,` *literal*`, Rc)`

Opcode:

110101	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\text{If } \langle\texttt{Ra}\rangle < \mathit{SEXT}(\mathit{literal}) \text{ then } \texttt{Rc} \leftarrow 1 \text{ else } \texttt{Rc} \leftarrow 0$

The contents of register `Ra` (as a signed quantity) are compared to *literal*. If the less-than relationship holds, the value one is written to register `Rc`; otherwise zero is written to `Rc`.

5.13 Div

Usage: `DIV (Ra, Rb, Rc)`

Opcode:

100011	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle / \langle\texttt{Rb}\rangle$

The contents of register `Ra` are divided by the contents of register `Rb` and the 32-bit quotient is written to register `Rc`. The result is truncated.

5.14 Divc

Usage: `DIVC (Ra,` *literal*`, Rc)`

Opcode:

110011	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle / \mathit{SEXT}(\mathit{literal})$

The contents of register `Ra` are divided by *literal* and the 32-bit quotient is written to register `Rc`. The result is truncated.

5.15 Jmp

Usage: `JMP (Ra, Rc)`

Opcode:

011011	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\mathit{EA} \leftarrow \langle\texttt{Ra}\rangle \wedge \texttt{0xFFFFFFFC}$

$\texttt{Rc} \leftarrow \langle\texttt{PC}\rangle$

$\texttt{PC} \leftarrow EA$

The `PC` of the instruction following the JMP instruction (the updated `PC`) is written to register `Rc`, followed by loading the `PC` with the target address. The new contents of `PC` are supplied from register `Ra`. The low two bits of `Ra` are masked. `Ra` and `Rc` may specify the same register; the target calculation using the old value is done before the assignment of the new value. The unused literal field should be filled with zeroes.

5.16 *Ld*

Usage: `LD (Ra,` *literal*`, Rc)`

Opcode:

011000	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
$\mathit{EA} \leftarrow \langle\texttt{Ra}\rangle + \mathit{SEXT}(\mathit{literal})$
$\texttt{Rc} \leftarrow \mathit{Memory}[\mathit{EA}]$

The effective address *EA* is computed by adding the contents of register `Ra` to the sign-extended 16-bit displacement *literal*. The location in memory specified by *EA* is read into register `Rc`.

5.17 *Ldr*

Usage: `LDR (`*label*`, Rc)`

Opcode:

011111	Rc	Ra	literal

Operation: $\mathit{literal} = ((\mathrm{OFFSET}(\mathit{label}) - \mathrm{OFFSET}(\text{current instruction})) \gg 2) - 1$
$\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
$\mathit{EA} \leftarrow \langle\texttt{PC}\rangle + 4 \cdot \mathit{SEXT}(\mathit{literal})$
$\texttt{Rc} \leftarrow \mathit{Memory}[\mathit{EA}]$

The effective address *EA* is computed by shifting the sign-extended *literal* left two bits (to address a longword boundary) and adding it to the updated `PC`. The location in memory specified by *EA* is read into register `Rc`. The `Ra` field is ignored and should be zero.

5.18 *Mul*

Usage: `MUL (Ra, Rb, Rc)`

Opcode:

100010	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \cdot \langle\texttt{Rb}\rangle$

The contents of register `Ra` are multiplied by the contents of register `Rb` and the 32-bit product is written to register `Rc`. On overflow, the least significant 32 bits of the true result are written to the destination register.

5.19 *Mulc*

Usage: `MULC (Ra,` *literal*`, Rc)`

Opcode:

110010	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$
$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \cdot \mathit{SEXT}(\mathit{literal})$

The contents of register `Ra` are multiplied by *literal* and the 32-bit product is written to register `Rc`. On overflow, the least significant 32 bits of the true result are written to the destination register.

5.20 Or

Usage: `OR (Ra, Rb, Rc)`

Opcode:

101001	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \vee \langle\texttt{Rb}\rangle$

This performs the bitwise boolean OR function between the contents of register `Ra` and the contents of register `Rb`. The result is written to register `Rc`.

5.21 Orc

Usage: `ORC (Ra,` *literal*`, Rc)`

Opcode:

111001	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \vee \mathit{SEXT}(\mathit{literal})$

This performs the bitwise boolean OR function between the contents of register `Ra` and *literal*. The result is written to register `Rc`.

5.22 Shl

Usage: `SHL (Ra, Rb, Rc)`

Opcode:

101100	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \ll \langle\texttt{Rb}\rangle_{4:0}$

The contents of register `Ra` are shifted left 0 to 31 bits by the five-bit count in register `Rb`. The result is written to register `Rc`. Zeroes are propagated into the vacated bit positions.

5.23 Shlc

Usage: `SHLC (Ra,` *literal*`, Rc)`

Opcode:

111100	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \ll \mathit{literal}_{4:0}$

The contents of register `Ra` are shifted left 0 to 31 bits by the five-bit count in *literal*. The result is written to register `Rc`. Zeroes are propagated into the vacated bit positions.

5.24 *Shr*

Usage: `SHR (Ra, Rb, Rc)`

Opcode:

101101	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \gg \langle\texttt{Rb}\rangle_{4:0}$

The contents of register `Ra` are shifted logically right 0 to 31 bits by the five-bit count in register `Rb`. The result is written to register `Rc`. Zeroes are propagated into the vacated bit positions.

5.25 *Shrc*

Usage: `SHRC (Ra,` *literal*`, Rc)`

Opcode:

111101	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \gg literal_{4:0}$

The contents of register `Ra` are shifted logically right 0 to 31 bits by the five-bit count in *literal*. The result is written to register `Rc`. Zeroes are propagated into the vacated bit positions.

5.26 *Sra*

Usage: `SRA (Ra, Rb, Rc)`

Opcode:

101110	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \gg \langle\texttt{Rb}\rangle_{4:0}$

The contents of register `Ra` are shifted arithmetically right 0 to 31 bits by the five-bit count in register `Rb`. The result is written to register `Rc`. The sign bit $\langle\texttt{Ra}\rangle_{31}$ is propagated into the vacated bit positions.

5.27 *Srac*

Usage: `SRAC (Ra,` *literal*`, Rc)`

Opcode:

111110	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \gg literal_{4:0}$

The contents of register `Ra` are shifted arithmetically right 0 to 31 bits by the five-bit count in *literal*. The result is written to register `Rc`. The sign bit $\langle\texttt{Ra}\rangle_{31}$ is propagated into the vacated bit positions.

5.28 *St*

Usage: ST (Rc, *literal*, Ra)

Opcode:

011001	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\mathit{EA} \leftarrow \langle\texttt{Ra}\rangle + \mathit{SEXT}(\mathit{literal})$

$\mathit{Memory}[\mathit{EA}] \leftarrow \langle\texttt{Rc}\rangle$

The effective address *EA* is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The contents of register Rc are then written to memory at this address.

5.29 *Sub*

Usage: SUB (Ra, Rb, Rc)

Opcode:

100001	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle - \langle\texttt{Rb}\rangle$

The contents of register Rb is subtracted from the contents of register Ra and the 32-bit difference is written to register Rc.

This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.30 *Subc*

Usage: SUBC (Ra, *literal*, Rc)

Opcode:

110001	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle - \mathit{SEXT}(\mathit{literal})$

The constant *literal* is subtracted from the contents of register Ra and the 32-bit difference is written to register Rc.

This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.31 *Xor*

Usage: XOR (Ra, Rb, Rc)

Opcode:

101010	Rc	Ra	Rb	unused

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \oplus \langle\texttt{Rb}\rangle$

This performs the bitwise boolean XOR function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.32 Xorc

Usage: `XORC (Ra,` *literal*`, Rc)`

Opcode:

111010	Rc	Ra	literal

Operation: $\texttt{PC} \leftarrow \langle\texttt{PC}\rangle + 4$

$\texttt{Rc} \leftarrow \langle\texttt{Ra}\rangle \oplus \mathit{SEXT}(\mathit{literal})$

This performs the bitwise boolean XOR function between the contents of register `Ra` and *literal*. The result is written to register `Rc`.

6. EXTENSIONS FOR EXCEPTION HANDLING

The standard β architecture described above is modified as follows to support exceptions and privileged instructions.

6.1 Exceptions

β exceptions come in three flavors: traps, faults, and interrupts.

Traps and faults are both the direct outcome of an instruction (e.g., an attempt to execute an illegal opcode) and are distinguished by the programmer's intentions. Traps are intentional and are normally used to request service from the operating system. Faults are unintentional and often signify error conditions.

Interrupts are asynchronous with respect to the instruction stream, and are usually caused by external events (e.g., a character appearing on an input device).

6.2 The `XP` *Register*

Register 30 is dedicated as the "Exception Pointer" (`XP`) register. When an exception occurs, the updated `PC` is written to the `XP`. For traps and faults, this will be the `PC` of the instruction following the one which caused the fault; for interrupts, this will be the `PC` of the instruction following the one which was about to be executed when the interrupt occurred.

Since the `XP` can be overwritten at unpredictable times as the result of an interrupt, it should not be used while interrupts are enabled.

6.3 Supervisor Mode

The high bit of the `PC` is dedicated as the "Supervisor" bit. The instruction fetch and LDR instruction ignore this bit, treating it as if it were zero. The JMP instruction is allowed to clear the Supervisor bit but not set it, and no other instructions may have any effect on it. Only exceptions cause the Supervisor bit to become set.

When the Supervisor bit is clear, the processor is said to be in "user mode". Interrupts are enabled while in user mode.

When the Supervisor bit is set, the processor is said to be in "supervisor mode". While in supervisor mode, interrupts are disabled and privileged instructions (see below) may be used. Traps and faults while in supervisor mode have implementation-defined (probably fatal) effects.

Since the JMP instruction can clear the Supervisor bit, it is possible to load the PC with a new value and enter user mode in a single atomic action. This provides a safe mechanism for returning from a call to the Operating System, even if an interrupt is pending at the time.

6.4 Exception Handling

When an exception occurs and the processor is in user mode, the updated `PC` is written to the `XP`, the Supervisor bit is set, the `PC` is loaded with an implementation-defined value, and the processor begins executing instructions from that point. This value is called the "exception vector", and may depend on the kind of exception which occurred.

The only exception which must be supported by all implementations is the "reset" exception (also called the "power up" exception), which occurs immediately before any instructions are executed by the processor. The exception vector for power up is always 0. Thus, at power up time, the Supervisor bit is set, the `XP` is undefined, and execution begins at location 0 of memory.

6.5 Privileged Instructions

Some instructions may be available while in supervisor mode which are not available in user mode (e.g., instructions which interface directly with I/O devices). These are called "privileged instructions". These instructions always have an opcode of 0x00; otherwise, their form and semantics are implementation-defined. Attempts to use privileged instructions while in user mode will result in an illegal instruction exception.