6.004 Computation Structures
Spring 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.
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 point to any location in the byte-addressed memory. When read, register 31 is always 0; when written, the new value is discarded.

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 register. Memory is referenced through load and store instructions that perform no other computation. Conditional branch instructions are separated from comparison instructions: branch instructions test the value of a register that 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 \( \beta \), an instruction's literal is represented in two’s complement.

### 3.1 Without Literal

- Opcode: 31, 26, 25, 21, 20, 16, 15, 11, 10, 0

### 3.2 With Literal

- Opcode: 31, 26, 25, 21, 20, 16, 15, 0
  - Ra: literal (two’s complement)

### 4. Instruction Summary

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

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

### 5. Instruction Specifications

This section contains the specifications for the \( \beta \) 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: PC ← PC + 4
 Reg[Rc] ← Reg[Ra] + Reg[Rb]

The contents of register Ra are 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: PC ← PC + 4
 Reg[Rc] ← Reg[Ra] + SEXT(literal)

The contents of register Ra are 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: PC ← PC + 4
 Reg[Rc] ← Reg[Ra] & Reg[Rb]

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: PC ← PC + 4
 Reg[Rc] ← Reg[Ra] & 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:  
literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1  
PC ← PC + 4  
EA ← PC + 4*SEXT(literal)  
TEMP ← Reg[Ra]  
Reg[Rc] ← PC  
if TEMP = 0 then PC ← EA

The PC of the instruction following the BEQ instruction (the updated PC) is written to register Rc. If the contents of register Ra 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 word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

5.6 BNE/BT

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

 Opcode:  
| 011110 | Rc | Ra | literal |

 Operation:  
literal = ((OFFSET(label) – OFFSET(current instruction)) ÷ 4)–1  
PC ← PC + 4  
EA ← PC + 4*SEXT(literal)  
TEMP ← Reg[Ra]  
Reg[Rc] ← PC  
if TEMP ≠ 0 then PC ← EA

The PC of the instruction following the BNE instruction (the updated PC) is written to register Rc. If the contents of register Ra 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 word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.
5.7 CMPEQ

Usage: CMPEQ(Ra,Rb,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>100100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 
PC ← PC + 4  
if Reg[Ra] = Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

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

5.8 CMPEQC

Usage: CMPEQC(Ra,literal,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 
PC ← PC + 4  
if Reg[Ra] = SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

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

5.9 CMPLE

Usage: CMPLE(Ra,Rb,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>100110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 
PC ← PC + 4  
if Reg[Ra] ≤ Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or equal to the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.10 CMPLEC

Usage: CMPLEC(Ra,literal,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 
PC ← PC + 4  
if Reg[Ra] ≤ SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or equal to literal, the value one is written to register Rc; otherwise zero is written to Rc.
**5.11 CMPLT**

Usage: CMPLT(Ra,Rb,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>100101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:
\[
PC \leftarrow PC + 4
\]
\[
\text{if } \text{Reg}[Ra] < \text{Reg}[Rb] \text{ then } \text{Reg}[Rc] \leftarrow 1 \text{ else } \text{Reg}[Rc] \leftarrow 0
\]

If the contents of register Ra are less than the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

**5.12 CMPLTC**

Usage: CMPLTC(Ra,literal,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:
\[
PC \leftarrow PC + 4
\]
\[
\text{if } \text{Reg}[Ra] < \text{SEXT(literal)} \text{ then } \text{Reg}[Rc] \leftarrow 1 \text{ else } \text{Reg}[Rc] \leftarrow 0
\]

If the contents of register Ra are less than literal, the value one is written to register Rc; otherwise zero is written to Rc.

**5.13 DIV**

Usage: DIV(Ra,Rb,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:
\[
PC \leftarrow PC + 4
\]
\[
\text{Reg}[Rc] \leftarrow \text{Reg}[Ra] / \text{Reg}[Rb]
\]

The contents of register Ra are divided by the contents of register Rb and the low-order 32 bits of the quotient are written to Rc.

**5.14 DIVC**

Usage: DIVC(Ra,literal,Rc)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:
\[
PC \leftarrow PC + 4
\]
\[
\text{Reg}[Rc] \leftarrow \text{Reg}[Ra] / \text{SEXT(literal)}
\]

The contents of register Ra are divided by literal and the low-order 32 bits of the quotient is written to Rc.
5.15 JMP

Usage: \texttt{JMP(Ra,Rc)}

Opcode: \begin{tabular}{c|c|c|c}
\texttt{011011} & \texttt{Rc} & \texttt{Ra} & \texttt{0000000000000000} \\
\end{tabular}

Operation:
\begin{align*}
\text{PC} & \leftarrow \text{PC}+4 \\
\text{EA} & \leftarrow \text{Reg}[\text{Ra}] \& 0xFFFFFFFF \\
\text{Reg}[\text{Rc}] & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{EA}
\end{align*}

The PC of the instruction following the JMP instruction (the updated PC) is written to register Rc, then the PC is loaded with the contents of register Ra. The low two bits of Ra are masked to ensure that the target address is aligned on a 4-byte boundary. 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. Note that JMP can clear the supervisor bit (bit 31 of the PC) but not set it – see section 6.3 for details.

5.16 LD

Usage: \texttt{LD(Ra,literal,Rc)}

Opcode: \begin{tabular}{c|c|c|c}
\texttt{011000} & \texttt{Rc} & \texttt{Ra} & \texttt{literal} \\
\end{tabular}

Operation:
\begin{align*}
\text{PC} & \leftarrow \text{PC}+4 \\
\text{EA} & \leftarrow \text{Reg}[\text{Ra}] + \text{SEXT(literal)} \\
\text{Reg}[\text{Rc}] & \leftarrow \text{Mem[EA]}
\end{align*}

The effective address \( \text{EA} \) is computed by adding the contents of register Ra to the sign-extended 16-bit displacement \textit{l literal}. The location in memory specified by \( \text{EA} \) is read into register Rc.

5.17 LDR

Usage: \texttt{LDR(label,Rc)}

Opcode: \begin{tabular}{c|c|c|c|c}
\texttt{011111} & \texttt{Rc} & \texttt{11111} & \texttt{literal} \\
\end{tabular}

Operation:
\begin{align*}
\text{literal} & \leftarrow ((\text{OFFSET(label)} - \text{OFFSET(current instruction)}) / 4) - 1 \\
\text{PC} & \leftarrow \text{PC}+4 \\
\text{EA} & \leftarrow \text{PC} + 4 \times \text{SEXT(literal)} \\
\text{Reg}[\text{Rc}] & \leftarrow \text{Mem[EA]}
\end{align*}

The effective address \( \text{EA} \) is computed by multiplying the sign-extended \textit{literal} by 4 (to convert it to a byte offset) and adding it to the updated PC. The location in memory specified by \( \text{EA} \) is read into register \( \text{Rc} \). The \( \text{Ra} \) field is ignored and should be \texttt{11111} (R31). The supervisor bit (bit 31 of the PC) is ignored (i.e., treated as zero) when computing \( \text{EA} \).
### 5.18 MUL

**Usage:** \( \text{MUL(Ra,Rb,Re)} \)

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>100010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Re]} & \leftarrow \text{Reg[Ra]} \times \text{Reg[Rb]}
\end{align*}
\]

The contents of register Ra are multiplied by the contents of register Rb and the low-order 32 bits of the product are written to Rc.

### 5.19 MULC

**Usage:** \( \text{MULC(Ra,literal,Re)} \)

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110010</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Re]} & \leftarrow \text{Reg[Ra]} \times \text{SEXT(literal)}
\end{align*}
\]

The contents of register Ra are multiplied by \( \text{literal} \) and the low-order 32 bits of the product are written to Rc.

### 5.20 OR

**Usage:** \( \text{OR(Ra,Rb,Re)} \)

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>101001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Re]} & \leftarrow \text{Reg[Ra]} \mid \text{Reg[Rb]}
\end{align*}
\]

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:** \( \text{ORC(Ra,literal,Re)} \)

**Opcode:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>111001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Re]} & \leftarrow \text{Reg[Ra]} \mid \text{SEXT(literal)}
\end{align*}
\]

This performs the bitwise boolean OR function between the contents of register Ra and \( \text{literal} \). The result is written to register Rc.
5.22 SHL

Usage: \( \text{SHL(Ra,Rb,Rc)} \)
Opcode: 
\[
\begin{array}{c|c|c|c}
\text{Opcode} & \text{Rc} & \text{Ra} & \text{Rb} \\
101100 & & & \\
\end{array}
\]
Operation: 
\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Rc]} & \leftarrow \text{Reg[Ra]} \ll \text{Reg[Rb]}_{4:0}
\end{align*}
\]

The contents of register Ra are shifted left 0 to 31 bits as specified 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: \( \text{SHLC(Ra,\text{literal},Rc)} \)
Opcode: 
\[
\begin{array}{c|c|c|c}
\text{Opcode} & \text{Rc} & \text{Ra} & \text{literal} \\
111100 & & & \\
\end{array}
\]
Operation: 
\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Rc]} & \leftarrow \text{Reg[Ra]} \ll \text{\text{literal}}_{4:0}
\end{align*}
\]

The contents of register Ra are shifted left 0 to 31 bits as specified 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: \( \text{SHR(Ra,Rb,Rc)} \)
Opcode: 
\[
\begin{array}{c|c|c|c}
\text{Opcode} & \text{Rc} & \text{Ra} & \text{Rb} \\
101101 & & & \\
\end{array}
\]
Operation: 
\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Rc]} & \leftarrow \text{Reg[Ra]} \gg \text{Reg[Rb]}_{4:0}
\end{align*}
\]

The contents of register Ra are shifted right 0 to 31 bits as specified 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: \( \text{SHRC(Ra,\text{literal},Rc)} \)
Opcode: 
\[
\begin{array}{c|c|c|c}
\text{Opcode} & \text{Rc} & \text{Ra} & \text{literal} \\
111101 & & & \\
\end{array}
\]
Operation: 
\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{Reg[Rc]} & \leftarrow \text{Reg[Ra]} \gg \text{\text{literal}}_{4:0}
\end{align*}
\]

The contents of register Ra are shifted right 0 to 31 bits as specified by the five-bit count in literal. The result 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: PC ← PC + 4
Reg[Rc] ← Reg[Ra] >> Reg[Rb]_{4:0}

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.25 SRAC
Usage: SRAC(Ra,literal,Rc)
Opcode: 111110 | Rc | Ra | literal
Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] >> literal_{4:0}

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in literal. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.28 ST
Usage: ST(Rc,literal,Ra)
Opcode: 011001 | Rc | Ra | literal
Operation: PC ← PC + 4
EA ← Reg[Ra] + SEXT(literal)
Mem[EA] ← Reg[Rc]

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 the location in memory specified by EA.

5.29 SUB
Usage: SUB(Ra,Rb,Rc)
Opcode: 100001 | Rc | Ra | Rb | unused
Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] – Reg[Rb]

The contents of register Rb are subtracted from the contents of register Ra and the 32-bit difference is written to Rc. This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.
5.30 SUBC

Usage: \( \text{SUBC}(R_a, \text{literal}, R_c) \)

 Opcode: 

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>110001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 

- \( \text{PC} \leftarrow \text{PC} + 4 \)
- \( \text{Reg}[R_c] \leftarrow \text{Reg}[R_a] - \text{SEXT}(\text{literal}) \)

The constant \( \text{literal} \) is subtracted from the contents of register \( R_a \) and the 32-bit difference is written to \( R_c \). This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.31 XOR

Usage: \( \text{XOR}(R_a, R_b, R_c) \)

 Opcode: 

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 

- \( \text{PC} \leftarrow \text{PC} + 4 \)
- \( \text{Reg}[R_c] \leftarrow \text{Reg}[R_a] \oplus \text{Reg}[R_b] \)

This performs the bitwise boolean XOR function between the contents of register \( R_a \) and the contents of register \( R_b \). The result is written to register \( R_c \).

5.32 XORC

Usage: \( \text{XORC}(R_a, \text{literal}, R_c) \)

 Opcode: 

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
<th>Ra</th>
<th>literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>111010</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: 

- \( \text{PC} \leftarrow \text{PC} + 4 \)
- \( \text{Reg}[R_c] \leftarrow \text{Reg}[R_a] \oplus \text{SEXT}(\text{literal}) \)

This performs the bitwise boolean XOR function between the contents of register \( R_a \) and \( \text{literal} \). The result is written to register \( R_c \).

6. Extensions for Exception Handling

The standard \( \beta \) architecture described above is modified as follows to support exceptions and privileged instructions.

6.1 Exceptions

\( \beta \) 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. The instruction pointed to by XP–4 has not been executed.

Since the XP can be overwritten at unpredictable times as the result of an interrupt, it should not be used by user-mode programs 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 trap 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.

7. Software Conventions

This section describes our software programming conventions that supplement the basic architecture.

7.1 Reserved Registers

It is convenient to reserve a number of registers for pervasive standard uses. The hardware itself reserves R31 and R30; in addition, our software conventions reserve R29, R28, and R27.

These are summarized in the following table and are described more fully below.

<table>
<thead>
<tr>
<th>Register</th>
<th>Symbol</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R31</td>
<td>R31</td>
<td>Always zero</td>
</tr>
<tr>
<td>R30</td>
<td>XP</td>
<td>Exception pointer</td>
</tr>
<tr>
<td>R29</td>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>R28</td>
<td>LP</td>
<td>Linkage pointer</td>
</tr>
<tr>
<td>R27</td>
<td>BP</td>
<td>Base of frame pointer</td>
</tr>
</tbody>
</table>

7.2 Convenience Macros

We augment the basic β instruction set with the following macros, making it easier to express certain common operations:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ(Ra, label)</td>
<td>BEQ(Ra, label, R31)</td>
</tr>
<tr>
<td>BF(Ra, label)</td>
<td>BF(Ra, label, R31)</td>
</tr>
<tr>
<td>BNE(Ra, label)</td>
<td>BNE(Ra, label, R31)</td>
</tr>
<tr>
<td>BT(Ra, label)</td>
<td>BT(Ra, label, R31)</td>
</tr>
<tr>
<td>BR(label, Rc)</td>
<td>BEQ(R31, label, Rc)</td>
</tr>
<tr>
<td>BR(label)</td>
<td>BR(label, R31)</td>
</tr>
<tr>
<td>JMP(Ra)</td>
<td>JMP(Ra, R31)</td>
</tr>
<tr>
<td>LD(label, Rc)</td>
<td>LD(R31, label, Rc)</td>
</tr>
<tr>
<td>ST(Rc, label)</td>
<td>ST(Rc, label, R31)</td>
</tr>
<tr>
<td>MOVE(Ra, Rc)</td>
<td>ADD(Ra, R31, Rc)</td>
</tr>
<tr>
<td>CMOVE(c, Rc)</td>
<td>ADDC(R31, c, Rc)</td>
</tr>
<tr>
<td>PUSH(Ra)</td>
<td>ADDC(SP, 4, SP)</td>
</tr>
<tr>
<td></td>
<td>ST(Ra, -4, SP)</td>
</tr>
<tr>
<td>POP(Rc)</td>
<td>LD(SP, -4, Rc)</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td>SUBC(SP, 4, SP)</td>
</tr>
<tr>
<td>ALLOCATE(k)</td>
<td>ADDC(SP, 4*k, SP)</td>
</tr>
<tr>
<td>DEALLOCATE(k)</td>
<td>SUBC(SP, 4*k, SP)</td>
</tr>
</tbody>
</table>

### 7.3 Stack Implementation

SP is a reserved register that points to the top of the stack. The stack is an arbitrary contiguous region of memory. The contents of SP are always a multiple of 4 and each stack slot is 4 bytes. SP points to the location just beyond the topmost element on the stack. The stack grows upward in memory (i.e., towards higher addresses). Four macros are defined for manipulating the stack:

- **PUSH(Ra)** - Push the contents of register Ra onto the stack
- **POP(Rc)** - Pop the top element of the stack into Rc
- **ALLOCATE(k)** - Push k words of uninitialized data onto the stack
- **DEALLOCATE(k)** - Pop k words off of the stack and throw them away

### 7.4 Procedure Linkage

A procedure’s arguments are passed on the stack. Specifically, when a procedure is entered, the topmost element on the stack is the first argument to the procedure; the next element on the stack is the second argument to the procedure, and so on. A procedure’s return address is passed in LP, which is a register reserved for this purpose. A procedure returns its value (if any) in R0 and must leave all other registers, including the reserved registers, unaltered.

Thus, a typical call to a procedure named F looks like:

```plaintext
(push arg_{n-1})
...
(push arg_1)
(push arg_0)
BR (F, LP)
DEALLOCATE (n)
(use R0, which is now F(arg_0, arg_1, ..., arg_{n-1}))
```
7.5 Stack Frames

The preceding section describes the rules which procedures must follow to interoperate properly. This section describes our conventional means of writing a procedure which follows those rules.

A procedure invocation requires storage for its arguments, its local variables, and any registers it needs to save and restore. All of this storage is allocated in a contiguous region of the stack called a “stack frame”. Procedures “activate” stack frames on entry and “deactivate” them on exit. BP is a reserved register which points to a fixed location within the currently active stack frame. Procedures use a standard prologue and epilogue to activate and deactivate the stack frame.

The standard prologue is:

```
PUSH (LP)
PUSH (BP)
MOVE (SP, BP)
ALLOCATE (k) | allocate space for locals
```

(push registers which are used by procedure)

Note that either of the last two steps may be omitted if there are no local variables or if there are no registers to save.

The standard epilogue is:

```
(pop registers which are used by procedure)
MOVE (BP, SP) | deallocate space for locals
POP (BP)
POP (LP)
JMP (LP)
```

Note that the epilogue assumes that the body of the procedure has no net effect on SP. Also note that either or both of the first two steps may be omitted if there are no registers to restore or if there are no local variables.

The standard prologue and epilogue together with the argument passing conventions imply the following layout for a stack frame:

```
| arg_{n-1} |
| ... |
| arg_{1} |
| arg_{0} |
| saved LP |
| saved BP |
| local_{0} |
| local_{1} |
| ... |
| local_{K-1} |
```

```
BP =>
```

Higher Addresses
Note that BP always points to the first stack slot above the saved BP, which is the same as the first local variable (if any). It also points to the third stack slot above the first argument (if any). So within the procedure’s body, its arguments and locals may be accessed via constant offsets from BP.
Summary of β Instruction Formats

Operate Class:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

OP(Ra,Rb,Rc): \(\text{Reg[Rc]} \leftarrow \text{Reg[Ra]} \text{ op Reg[Rb]}\)

Opcodes: ADD (plus), SUB (minus), MUL (multiply), DIV (divided by)
AND (bitwise and), OR (bitwise or), XOR (bitwise exclusive or)
CMPEQ (equal), CMPLT (less than), CMPLE (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), SHR (right shift w/o sign extension), SRA (right shift w/ sign extension)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>literal (two’s complement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OPC(Ra,literal,Rc): \(\text{Reg[Rc]} \leftarrow \text{Reg[Ra]} \text{ op SEXT(literal)}\)

Opcodes: ADDC (plus), SUBC (minus), MULC (multiply), DIVC (divided by)
ANDC (bitwise and), ORC (bitwise or), XORC (bitwise exclusive or)
CMPEQC (equal), CMPLTC (less than), CMPLEC (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), SHRC (right shift w/o sign extension), SRAC (right shift w/ sign extension)

Other:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>literal (two’s complement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD(Ra,literal,Rc): \(\text{Reg[Rc]} \leftarrow \text{Mem[Reg[Ra] + SEXT(literal)]}\)
ST(Rc,literal,Ra): \(\text{Mem[Reg[Ra] + SEXT(literal)]} \leftarrow \text{Reg[Rc]}\)
JMP(Ra,Rc): \(\text{Reg[Rc]} \leftarrow \text{PC + 4}; \text{PC} \leftarrow \text{Reg[Ra]}\)
BEQ/BF(Ra,label,Rc): \(\text{Reg[Rc]} \leftarrow \text{PC + 4}; \text{if Reg[Ra] = 0 then PC} \leftarrow \text{PC + 4 + 4*SEXT(literal)}\)
BNE/BT(Ra,label,Rc): \(\text{Reg[Rc]} \leftarrow \text{PC + 4}; \text{if Reg[Ra] \neq 0 then PC} \leftarrow \text{PC + 4 + 4*SEXT(literal)}\)
LDR(Ra,label,Rc): \(\text{Reg[Rc]} \leftarrow \text{Mem[PC + 4 + 4*SEXT(literal)]}\)

Opcode Table: (*optional opcodes)

<table>
<thead>
<tr>
<th>5:3</th>
<th>2:0</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>LD</td>
<td>ST</td>
<td>JMP</td>
<td>BEQ</td>
<td>BNE</td>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>ADD</td>
<td>SUB</td>
<td>MUL*</td>
<td>DIV*</td>
<td>CMPEQ</td>
<td>CMPLT</td>
<td>CMPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>AND</td>
<td>OR</td>
<td>XOR</td>
<td>SHL</td>
<td>SHR</td>
<td>SRA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>ADDC</td>
<td>SUBC</td>
<td>MULC*</td>
<td>DIVC*</td>
<td>CMPEQC</td>
<td>CMPLTC</td>
<td>CMPLEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>ANDC</td>
<td>ORC</td>
<td>XORC</td>
<td>SHLC</td>
<td>SHRC</td>
<td>SRAC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.004 Computation Structures - 17 - β Documentation