A performance-oriented implementation of the $\beta$ machine
β Machine: instruction formats (review)

All operations of the β Machine (except those accessing the main memory) are performed on registers.

There are two possible instruction formats:

- The format without a literal

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>unused</th>
</tr>
</thead>
</table>
```

- The format with a literal

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>value (2’s complement)</th>
</tr>
</thead>
</table>
```
β Machine: arithmetic instructions (review)

<table>
<thead>
<tr>
<th>Without a literal</th>
<th>With a literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>name</td>
</tr>
<tr>
<td>0x20</td>
<td>ADD</td>
</tr>
<tr>
<td>0x21</td>
<td>SUB</td>
</tr>
<tr>
<td>0x22</td>
<td>MUL</td>
</tr>
<tr>
<td>0x23</td>
<td>DIV</td>
</tr>
</tbody>
</table>

\[
\text{ADD}(Ra,Rb,Rc) : \quad \text{PC} \leftarrow \text{PC} + 4 \\
\quad \text{Reg}[Rc] \leftarrow \text{Reg}[Ra] + \text{Reg}[Rb]
\]

\[
\text{ADDC}(Ra,\text{literal},Rc) : \quad \text{PC} \leftarrow \text{PC} + 4 \\
\quad \text{Reg}[Rc] \leftarrow \text{Reg}[Ra] + \text{SEXT(}\text{literal})
\]

\text{SEXT(}\text{literal}) \text{ represents the value contained in the instruction extended from 16 to 32 bits, the sign being preserved.}

\text{The definitions of SUB, MUL and DIV are similar.}
**β Machine: memory access instructions (review)**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18</td>
<td>LD</td>
</tr>
<tr>
<td>0x19</td>
<td>ST</td>
</tr>
</tbody>
</table>

LD(Ra, literal, Rc) :  
PC ← PC + 4  
EA ← Reg[Ra] + SEXT(literal)  
Reg[Rc] ← Mem[EA]  

ST(Rc, literal, Ra) :  
PC ← PC + 4  
EA ← Reg[Ra] + SEXT(literal)  
Mem[EA] ← Reg[Rc]  

EA is called the “effective address”.
β Machine: memory access instructions II (review)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1F</td>
<td>LDR</td>
</tr>
</tbody>
</table>

LDR(label,Rc) :  
PC ← PC + 4  
EA ← PC + 4 × SEXT(literal)  
Reg[Rc] ← Mem[EA]

In LDR, the instruction given to the assembler contains a label. (label). In the assembled instruction, the literal is computed from the label as follows:

\[
\text{literal} = ((\text{OFFSET(label)} - \text{OFFSET(current inst.)}) \div 4) - 1
\]
\( \beta \) Machine: branch instructions (review)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1B</td>
<td>JMP</td>
</tr>
</tbody>
</table>

\[ \text{JMP(Ra,Rc)} : \quad \text{PC} \leftarrow \text{PC} + 4 \]
\[ \text{EA} \leftarrow \text{Reg[Ra]} \& 0xFFFFFFFC \]
\[ \text{Reg[Rc]} \leftarrow \text{PC} \]
\[ \text{PC} \leftarrow \text{EA} \]

The 2 least significant bits of Ra are set to 0 to force the address to be a word address. The address of the instruction following the JMP is saved in Rc.
**β Machine: branch instructions II (review)**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1D</td>
<td>BEQ/BF</td>
</tr>
<tr>
<td>0x1E</td>
<td>BNE/BT</td>
</tr>
</tbody>
</table>

**BEQ(Ra,label,Rc):**

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{EA} & \leftarrow \text{PC} + 4 \times \text{SEXT(literal)} \\
\text{TMP} & \leftarrow \text{Reg}[Ra] \\
\text{Reg}[Rc] & \leftarrow \text{PC} \\
\text{if } \text{TMP} = 0 \text{ then } \text{PC} & \leftarrow \text{EA}
\end{align*}
\]

In **BEQ**, the instruction given to the assembler contains a label. In the assembled instruction, the literal is computed as follows

\[
\text{literal} = ((\text{OFFSET(label)} - \text{OFFSET(current inst.)}) \div 4) - 1
\]

**BNE** is similar to **BEQ** except that the test is “different from 0”

**TMP** is used because **Ra** and **Rc** could be the same register.
An implementation of the $\beta$ machine that does not use micro-code

The goal is to execute instructions within a single clock cycle. For doing this, one proceeds as follows.

- Several data transfer paths are used in order to make simultaneous transfers possible.
- Most elements then have their input connected to a multiplexer, making it possible to select a source among several.
- *Three port* registers are used. These allow two read and one write operation to be executed simultaneously (at the next clock transition).
- The goal of the control logic is then to select the input of the various elements and to choose the function computed by the ALU.
- The data memory and the instruction memory are viewed as separate (caches).
An implementation of the $\beta$ machine that does not use micro-code (continued)

- The following slide presents the general schema of an implementation of the $\beta$ machine that does not use micro-code.

- *Exceptions* upon which $\text{PC} + 4$ is saved in $\text{XP}$ (register 30) and control jumps to a predefined address are taken into account.

- Supervisor mode is not implemented.

- Virtual memory is not taken into account (one can assume that the caches work with virtual addresses).
The $\beta$ machine without micro-code: the signals

- **Xadr**: address of the interrupt handler.
- **ILL0P**: address of the illegal operation handler.
- **JT**: target of a JMP instruction (jump target).
- **PCSEL**: PC source selection.
- **RA2SEL**: selection of the second register read address.
- **WASEL**: selection of the register write address.
- **XP**: exception register (30).
- **WERF**: Write Enable Register File.
- **Z**: 0 test of the first register output.
- **A(B)SEL**: ALU A (B) input selection.
- **WDSEL**: register write data selection.
- **ALUFN**: ALU function selection.
- **Wr**: "Write" mode for the data memory.
The $\beta$ machine without micro-code: the control logic

This logic can be implemented with a ROM or a PLA.

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>OPC</th>
<th>LD</th>
<th>ST</th>
<th>JMP</th>
<th>BEQ</th>
<th>BNE</th>
<th>LDR</th>
<th>ILLOP</th>
<th>trap</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUFN</td>
<td>$F(op)$</td>
<td>$F(op)$</td>
<td>“+”</td>
<td>“+”</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>“A”</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WERF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BSEL</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WDSEL</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wr</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RA2SEL</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCSEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>$Z==1$</td>
<td>$Z==0$</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ASEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>WASEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The $\beta$ machine without micro-code: performance?

- The implementation we have described executes an instruction per clock cycle, but the cycles have to be quite long.

- Indeed, a cycle has to be long enough for the successive stabilization of the instruction memory, the registers, the ALU, that data memory.

- The solution is to organize the implementation as a pipeline