Lecture 10: Building the Beta

I wonder where this goes?
CPU Design Tradeoffs

Maximum Performance: measured by the numbers of instructions executed per second

Minimum Cost: measured by the size of the circuit.

Best Performance/Price: measured by the ratio of MIPS to size. In power-sensitive applications MIPS/Watt is important too.
Processor Performance

• “Iron Law” of performance:

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

\[
\text{Perf} = \frac{1}{\text{Time}}
\]

• Options to reduce execution time:
  – Executed instructions ↓ (work/instruction ↑)
  – Cycles per instruction (CPI) ↓
  – Cycle time ↓ (frequency ↑)

• Today: Simple, CPI=1 but low-frequency Beta
  – Later: Pipelining to increase frequency
Reminder: Beta ISA

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 XX XX</td>
<td>Rc</td>
<td>Ra</td>
<td>Rb</td>
<td>(UNUSED)</td>
</tr>
</tbody>
</table>

**Operate class:** \( \text{Reg}[\text{Rc}] \leftarrow \text{Reg}[\text{Ra}] \text{ op } \text{Reg}[\text{Rb}] \)

**Operate class:** \( \text{Reg}[\text{Rc}] \leftarrow \text{Reg}[\text{Ra}] \text{ op } \text{SXT}(C) \)

Opcodes, both formats:

- ADD
- SUB
- MUL* (if applicable)
- DIV* (if applicable)
- CMPEQ
- CMPLT
- AND
- OR
- XOR
- XNOR
- SHL
- SHR
- SRA

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Literal C (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 XX XX</td>
<td>Rc</td>
<td>Ra</td>
<td>Literal C (signed)</td>
</tr>
</tbody>
</table>

- LD: \( \text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{Reg}[\text{Ra}]+\text{SXT}(C)] \)
- ST: \( \text{Mem}[\text{Reg}[\text{Ra}]+\text{SXT}(C)] \leftarrow \text{Reg}[\text{Rc}] \)
- LDR: \( \text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{PC}+4+4*\text{SXT}(C)] \)
- BEQ: \( \text{Reg}[\text{Rc}] \leftarrow \text{PC}+4; \text{ if } \text{Reg}[\text{Ra}]=0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4*\text{SXT}(C) \)
- BNE: \( \text{Reg}[\text{Rc}] \leftarrow \text{PC}+4; \text{ if } \text{Reg}[\text{Ra}]\neq0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4*\text{SXT}(C) \)
- JMP: \( \text{Reg}[\text{Rc}] \leftarrow \text{PC}+4; \text{ PC } \leftarrow \text{Reg}[\text{Ra}] \)
Approach: Incremental Featurism

We’ll implement datapaths for each instruction class individually, and merge them (using MUXes, etc)

Steps:
1. ALU instructions
2. Load & store instructions
3. Jump & branch instructions
4. Exceptions

Component Repertoire:
- Registers
- Muxes
- “Black box” ALU
- Memories
Multi-Ported Register File

2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[31] reads as 0
2 combinational READ ports, 1 clocked WRITE port

What if (say) WA=RA1???
RD1 reads “old” value of Reg[RA1] until next clock edge!
### ALU Instructions

32-bit (4-byte) ADD instruction:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>(unused)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000001000001000011000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Means, to Beta,  Reg[R4] ← Reg[R2] + Reg[R3]*

Need hardware to:

- **FETCH** (read) 32-bit instruction for the current cycle
- **DECODE** instruction: ADD, SUB, XOR, etc
- **READ** operands (Ra, Rb) from Register File
- **EXECUTE** operation
- **WRITE-BACK** result into Register File (Rc)
Instruction Fetch/Decode

Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)

- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
  - Use some instruction fields directly (register numbers, 16-bit constant)
  - Use bits [31:26] to generate control signals
ALU Op Datapath

Operate class: Reg[Rc] ← Reg[Ra] op Reg[Rb]
ALU Op Datapath

Operate class: Reg[Rc] ← Reg[Ra] op Reg[Rb]
ALU Operations (with constant)

Operate class: Reg[Rc] ← Reg[Ra] op SXT(C)

Sign-extension requires no logic...

Just replicate ID[15] sixteen times to create high-order bits!
ALU Operations (with constant)

```
1 1 XXXX  Rc  Ra  Literal C (signed)
```

Operate class: Reg[Rc] ← Reg[Ra] op SXT(C)
Load Instruction

LD: \( \text{Reg}[Rc] \leftarrow \text{Mem}[\text{Reg}[Ra]+\text{SXT}(C)] \)

This is just like \( \text{ADDC}(Ra, C, \ldots) \)
Load Instruction

\[ \text{LD: } \text{Reg}[^{\text{Rc}}] \leftarrow \text{Mem}[\text{Reg}[^{\text{Ra}}]+\text{SXT}(C)] \]
Store Instruction

ST: $\text{Mem}[	ext{Reg}[Ra]+\text{SXT}(C)] \leftarrow \text{Reg}[Rc]$
Store Instruction

0 1 1 0 0 1  Rc  Ra  Literal C (signed)

ST:  Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]
JMP Instruction

\[
\text{JMP: Reg[Rc] } \leftarrow \text{PC+4; PC } \leftarrow \text{Reg[Ra]}
\]
JMP Instruction

```
<table>
<thead>
<tr>
<th>0 1 1 0 1 1</th>
<th>Rc</th>
<th>Ra</th>
<th>Literal C (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP: Reg[Rc] ← PC+4; PC ← Reg[Ra]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:

- Instruction Memory
  - ID[31:0]
  - ID[31:26] → C: SXT(ID[15:0])
  - PCSEL
  - RA2SEL
  - BSEL
  - WDSEL
  - ALUFN
  - MWR, MOE
  - WERF

- Register File
  - RA1
  - RA2
  - RA2SEL
  - RD1
  - RD2
  - WE
  - WD

- ALU
  - A
  - B
  - ALUFN
  - WDSEL
  - MWR, MOE
  - WERF

- Data Memory
  - Addr
  - RD
  - MWR

- Control Logic
  - PCSEL
  - RA2SEL
  - BSEL
  - ALUFN
  - WDSEL
  - MWR, MOE
  - WERF
BEQ/BNE Instructions

BEQ: Reg[Rc] ← PC+4; if Reg[Ra]=0 then PC ← (PC+4)+4*SXT(C)

BNE: Reg[Rc] ← PC+4; if Reg[Ra]≠0 then PC ← (PC+4)+4*SXT(C)

"4*" only requires adding 0b00 to low-order bits - no HW needed!
BEQ/BNE Instructions

**BEQ**:
- `Reg[Rc] ← PC+4; if Reg[Ra]=0 then PC ← (PC+4)+4*SXT(C)`

**BNE**:
- `Reg[Rc] ← PC+4; if Reg[Ra]≠0 then PC ← (PC+4)+4*SXT(C)`
Load Relative Instruction

What’s Load Relative good for anyway?? I thought

• Code is “PURE”, i.e. READ-ONLY; and stored in a “PROGRAM” region of memory;
• Data is READ-WRITE, and stored either
  • On the STACK (local); or
  • In some GLOBAL VARIABLE region; or
  • In a global storage HEAP.

So why have an instruction designed to load data that’s “near” the instruction??

Addresses & other large constants

C: \[ X = X \times 123456; \]

BETA:

\[
\begin{align*}
LDR & (X, r0) \\
LDR & (c1, r1) \\
MUL & (r0, r1, r0) \\
ST & (r0, X) \\
& \ldots \\
c1: & \text{LONG}(123456)
\end{align*}
\]
LDR Instruction

![Diagram of LDR Instruction]

LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]

Why not put the ASEL mux here?
LDR Instruction

\[ \text{LDR: } \text{Reg}[Rc] \leftarrow \text{Mem}[PC + 4 + 4 \times \text{SXT}(C)] \]
Exceptions

• What if something bad happens?
  – Execution of an illegal opcode
  – Reference to non-existent memory
  – Divide by zero

• Or maybe just something unanticipated
  – User hits a key
  – A packet comes in via the network

• Exceptions let us handle these cases in software:
  – Treat each case as an (implicit) procedure call
  – Procedure handles problem, returns to interrupted program
  – Transparent to interrupted program!
  – Important added capability: handlers for certain errors (illegal opcodes), can extend ISA using software
Exception Processing

• Plan:
  – Interrupt running program
  – Invoke exception handler (like a procedure call)
  – Return to continue execution

• Exception and interrupt terms often used interchangeably, with minor distinctions:
  – **Exceptions** usually refer to **synchronous events**, generated by program (e.g., illegal instruction, divide-by-0, illegal address)
  – **Interrupts** usually refer to **asynchronous events**, generated by I/O devices (e.g., keystroke, packet received, disk transfer complete)
Exception Implementation

• Instead of executing instruction, fake a procedure call
  – Save current PC+4 (as branches do)
  – Load PC with exception vector: 0x4 for synchronous events, 0x8 for asynchronous events

• We save PC+4 in register R30 (which we call XP)
  – ... and prohibit programs from using XP (why?)

• Example: DIV unimplemented

```
LD(R31,A,R0)
LD(R31,B,R1)
DIV(R0,R1,R2)
ST(R2,C,R31)
```

IllOp:
-
PUSH(XP)
-
Fetch inst. at Mem[Reg[XP]−4]
-
check for DIV opcode, get reg numbers
-
perform operation in SW, fill result reg
-
POP(XP)
-
JMP(XP)

Forced by hardware
Exceptions

Bad Opcode: Reg[XP] ← PC+4;  PC ← “IllOp”
Other:   Reg[XP] ← PC+4;  PC ← “Xadr”
Exceptions

Bad Opcode: Reg[XP] ← PC+4; PC ← “IllOp”
Other: Reg[XP] ← PC+4; PC ← “Xadr”
Beta: Our “Final Answer”
# Control Logic

<table>
<thead>
<tr>
<th></th>
<th>RESET</th>
<th>IRQ</th>
<th>OP</th>
<th>OPC</th>
<th>LD</th>
<th>LDR</th>
<th>ST</th>
<th>JMP</th>
<th>BEQ</th>
<th>BNE</th>
<th>ILLOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUFN</td>
<td>--</td>
<td>--</td>
<td>F(op)</td>
<td>F(op)</td>
<td>&quot;+&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;+&quot;</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>ASEL</td>
<td>--</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>BSEL</td>
<td>--</td>
<td>--</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MOE</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MWR</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>PCSEL</td>
<td>--</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Z</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RA2SEL</td>
<td>--</td>
<td>--</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>WASEL</td>
<td>--</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>WDSEL</td>
<td>--</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WERF</td>
<td>--</td>
<td>1</td>
<td>1</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>
</tr>
</tbody>
</table>

**Implementation choices:**
- 64-location ROM indexed by opcode with external logic to handle changes due to Z and IRQ inputs
- Entirely combinational logic (faster, but much more work!)
Is that all there is to building a processor???

No. You've gotta print up all those little "Beta Inside" stickers.