Building the Beta

I wonder where this goes?

Notes:
Quiz 2 – Friday

Enjoy Spring Break!
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:

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \quad \text{Perf} = \frac{1}{\text{Time}}
\]

• 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

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*
- DIV*
- CMPEQ
- CMPLE
- CMPLT
- AND
- OR
- XOR
- SHL
- SHR
- SRA
- XNOR

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: \( \text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{PC} + 4 + 4\times\text{SXT}(C)] \)
BEQ: \( \text{Reg}[\text{Rc}] \leftarrow \text{PC}+4; \text{ if Reg}[\text{Ra}]=0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4\times\text{SXT}(C) \)
BNE: \( \text{Reg}[\text{Rc}] \leftarrow \text{PC}+4; \text{ if Reg}[\text{Ra}]\neq0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4\times\text{SXT}(C) \)
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

Diagram showing:
- ALU
- Register File (3-port)
- Instruction Memory
- Data Memory
Multi-Ported Register File

2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[31] reads as 0
Register File Timing

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:

```
10000000010000010000110000000000000
```

```
 Opcode   Rc   Ra   Rb   (unused)
```

\[\text{Means, to Beta, } \text{Reg}[R4] \leftarrow \text{Reg}[R2] + \text{Reg}[R3]\]

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

Sign-extension requires no logic...

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

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

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

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

0 1 1 0 0 0 | Rc | Ra | Literal C (signed)
LD: Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]
Store Instruction

ID[31:0]:
- PC: 00
- Instruction Memory
  - ID[31:0]
- +4
- Instruction
- Memory
- A
- D
- RA: ID[20:16]
- RD: ID[25:21]
- C: SXT(ID[15:0])

Register File
- RA1
- RA2
- RD1
- RD2
- WERF
- BSEL
- WDSEL
- ALUFN
- MWR, MOE
- WERF
- RA2SEL

Control Logic
- BSEL
- WDSEL
- ALUFN
- MWR, MOE
- WERF

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

Data Memory
- Addr
- RD
- R/W
- OE
- MWR

ST:
- Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]

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

No WERF!
Store Instruction

011001  Rc  Ra

Literal C (signed)

ST: Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]

Diagram showing the process of storing a value from a register into memory, with details on the instruction format and data flow through the registers and memory.
**JMP Instruction**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C: SXT(ID[15:0])</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Logic</th>
<th>PCSEL</th>
<th>RA2SEL</th>
<th>BSEL</th>
<th>WDSEL</th>
<th>ALUFN</th>
<th>MWR, MOE</th>
<th>WERF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCSEL</th>
<th>RA2SEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Memory</th>
<th>ID[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC</td>
</tr>
<tr>
<td>+4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra: ID[20:16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rb: ID[15:11]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rc: ID[25:21]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUFN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MWR, MOE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WERF</td>
<td></td>
</tr>
</tbody>
</table>

**JMP:**

\[
\text{Reg}[\text{Rc}] \leftarrow \text{PC} + 4; \quad \text{PC} \leftarrow \text{Reg}[\text{Ra}]
\]
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>
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:

- \( \text{LD}(X, r0) \)
- \( \text{LDR}(c1, r1) \)
- \( \text{MUL}(r0, r1, r0) \)
- \( \text{ST}(r0, X) \)
- ...

\( c1: \text{LONG}(123456) \)
LDR Instruction

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

Why not put the ASEL mux here?
LDR Instruction

LDR: Reg[Rc] ← Mem[PC + 4 + 4*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>--</td>
<td>1</td>
<td>1</td>
<td>0</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 ? 1 : 0</td>
<td>Z ? 0 : 1</td>
<td>3</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>--</td>
<td>0</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.