Lecture 15: RISC-V Part 2
Implementing Branches

Uses the “B-type” instruction format

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

• RISC-V Assembly Instruction, example:
  
  beq rs1, rs2, label
  
  if rs1==rs2 pc ← pc + offset // offset computed by compiler/assembler and stored in the immediate field(s)
  
  example:
  
  beq x1, x2, L1

• B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate

• But now immediate represents values -4096 to +4094 in 2-byte increments

• The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
Review: Adding \texttt{sw} to datapath
Adding branches to datapath
Adding branches to datapath
Branch Comparator

- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn = 1 selects unsigned comparison for BrLT, 0=signed
- BGE branch: A >= B, if !(A<B)
### RISC-V Immediate Encoding

#### Instruction Encodings, inst[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25 24</th>
<th>21</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit immediates produced, imm[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>20 19</th>
<th>12 11</th>
<th>10</th>
<th>5 4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

- Upper bits sign-extended from inst[31] always
- Only bit 7 of instruction changes role in immediate between S and B
Implementing `jalr` Instruction (I-Format)

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `jalr rd, rs, immediate`
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + offset
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
Review: Adding branches to datapath
Adding jalr to datapath
Adding `jalr` to datapath

Diagram of the datapath with `jalr` instruction added.
Implementing \texttt{jal} Instruction

Uses the “J-type” instruction format

\begin{verbatim}
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}

- JAL saves PC+4 in Reg[rd] (the return address)
- Set \( \text{PC} = \text{PC} + \text{offset} \) (PC-relative jump)
- Target somewhere within \( \pm 2^{19} \) locations, 2 bytes apart
  - \( \pm 2^{18} \) 32-bit instructions
- Immediate encoding optimized similarly to branch instruction
Adding `jal` to datapath
Adding jal to datapath
Single-Cycle RISC-V RV32I Datapath
Controller Implementation:

- Control logic works really well as a case statement...

```verilog
controller
always @* begin
  op = instr[26:31];
  imm = instr[15:0]; ...

  reg_dst = 1'bx;  // Don't care
  reg_write = 1'b0;  // By default don’t write

  ... case (op)
  6'b000000: begin reg_write = 1; ... end
  ...
Processor Pipelining
Review: Processor Performance

Program Execution Time

\[ = (\# \text{ instructions})(\text{cycles/instruction})(\text{seconds/cycle}) \]

\[ = \# \text{ instructions} \times \text{CPI} \times T_C \]
Single-Cycle Performance

- $T_C$ is limited by the critical path (1w)
Single-Cycle Performance

• Single-cycle critical path:
  \[ T_c = t_{q_{PC}} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \]

• In most implementations, limiting paths are:
  – memory, ALU, register file.
  – \[ T_c = t_{q_{PC}} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup} \]
Pipelined Processor

- Use temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages
Single-Cycle vs. Pipelined Performance

Single-Cycle

Pipelined
Single-Cycle and Pipelined Datapath
Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result
Pipelined Control

Same control unit as single-cycle processor

Control delayed to proper pipeline stage
Pipeline Hazards

- Occurs when an instruction depends on results from previous instruction that hasn’t completed.

- Types of hazards:
  - **Data hazard**: register value not written back to register file yet
  - **Control hazard**: next instruction not decided yet (caused by branches)

*We need to design ways to avoid hazards, else we pay the price in CPI (cycles per instruction) and processor performance suffers.*
Processor Pipelining

Deeper pipeline example.

\[
\begin{array}{cccccc}
\text{IF1} & \text{IF2} & \text{ID} & \text{X1} & \text{X2} & \text{M1} & \text{M2} & \text{WB} \\
\text{IF1} & \text{IF2} & \text{ID} & \text{X1} & \text{X2} & \text{M1} & \text{M2} & \text{WB}
\end{array}
\]

Deeper pipelines \(\Rightarrow\) less logic per stage \(\Rightarrow\) high clock rate.

\textit{But}\n
Deeper pipelines* \(\Rightarrow\) more hazards \(\Rightarrow\) more cost and/or higher CPI.

Cycles per instruction might go up because of unresolvable hazards.

\textit{Remember, Performance} = \# instructions \times \text{Frequency}_{\text{clk}} / \text{CPI}

*Many designs included pipelines as long as 7, 10 and even 20 stages (like in the Intel Pentium 4). The later "Prescott" and "Cedar Mill" Pentium 4 cores (and their Pentium D derivatives) had a 31-stage pipeline.

How about shorter pipelines ... Less cost, less performance (but higher cost efficiency)
3-Stage Pipeline
The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

I | X | M

Use PC register as address to IMEM and retrieve next instruction. Instruction gets stored in a pipeline register, also called “instruction register”, in this case.

Use ALU to compute result, memory address, or branch target address.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

Most details you will need to work out for yourself. Some details to follow ...
In particular, let’s look at hazards.
3-stage Pipeline

Data Hazard

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>add x5, x3, x4</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>add x7, x6, x5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

The fix:

Selectively forward ALU result back to input of ALU.

- Need to add mux at input to ALU, add control logic to sense when to activate.
3-stage Pipeline

Load Hazard

<table>
<thead>
<tr>
<th>lw  x5, offset(x4)</th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>add  x7, x6, x5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

The fix: Delay the dependent instruction by one cycle to allow the load to complete, send the result of load directly to the ALU (and to the regfile). No delay if not dependent!

<table>
<thead>
<tr>
<th>lw  x5, offset(x4)</th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>add  x7, x6, x5</td>
<td>I</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>add  x7, x6, x5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>
### Control Hazard

#### 3-stage Pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1 (I)</th>
<th>Stage 2 (X)</th>
<th>Stage 3 (M)</th>
<th>Stage 4 (O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq x1, x2, L1</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x5, x3, x4</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x6, x1, x2</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1: sub x7, x6, x5</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**but needed here!**

**branch address ready here**

#### Several Possibilities:

**The fix:**

1. Always delay fetch of instruction after branch
2. Assume branch “not taken”, continue with instruction at PC+4, and correct later if wrong.
3. Predict branch taken or not based on history (state) and correct later if wrong.

1. Simple, but all branches now take 2 cycles (lowers performance)
2. Simple, only some branches take 2 cycles (better performance)
3. Complex, very few branches take 2 cycles (best performance)

* MIPS defines “branch delay slot”, RISC-V doesn’t


**Control Hazard**

Branch address ready at end of X stage:
- If branch “not taken”, do nothing.
- If branch “taken”, then kill instruction in I stage (about to enter X stage) and fetch at new target address (PC)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneq x1, x1, L1</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>add x5, x3, x4</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>add x6, x1, x2</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>L1: sub x7, x6, x5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

**Not taken**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq x1, x1, L1</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>add x5, x3, x4</td>
<td>I</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>L1: sub x7, x6, x5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

**Taken**
Pipeline rules:
- Writes/reads to/from DMem are clocked on the leading edge of the clock in the “M” stage
- Writes to RegFile at the end of the “M” stage
- Instruction Decode and Register File access is up to you.

Branch: predict “not-taken”

Load: 1 cycle delay/stall on dependent instruction

Bypass ALU for data hazards

More details in upcoming spec
End of Lecture 15