Last time in Lecture 3

- Microcoding became less attractive as gap between RAM and ROM speeds reduced
- Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
- Iron-law explains architecture design space
  - Trade instructions/program, cycles/instruction, and time/cycle
- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Very similar to vertical microcode
  - Inspired by earlier Cray machines
“Iron Law” of Processor Performance

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

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>

An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines. But can an instruction pipeline satisfy the last condition?
Pipelined MIPS

To pipeline MIPS:

• First build MIPS without pipelining with CPI=1

• Next, add pipeline registers to reduce cycle time while maintaining CPI=1

Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max\{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \quad (= t_{DM\text{ probably}})\]

However, CPI will increase unless instructions are pipelined
Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

Thus, the following timing assumption is reasonable

\[ t_{IM} = t_{RF} = t_{ALU} = t_{DM} = t_{RW} \]

A 5-stage pipelined Harvard architecture will be the focus of our detailed design

5-Stage Pipelined Execution
5-Stage Pipelined Execution
Resource Usage Diagram

Time

I-Fetch (IF)
Decode, Reg. (ID)
Fetch (EX)
Execute (EX)
Memory (MA)
Write-Back (WB)

Resources

IF
ID
EX
MA
WB

Not quite correct!

We need an Instruction Reg (IR) for each stage
How Instructions can Interact with each other in a pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → structural hazard

- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value → data hazard
  - Dependence may be for the next instruction’s address → control hazard (branches, exceptions)
Data Hazards

... r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!

CS152 Administrivia

• PS 1 due Tuesday Feb 10 in class
  • Section covering PS 1 on Wednesday Feb 11
    • Room/time TBD

• First Quiz on Thursday Feb 12
  • In class, closed-book, no computers or calculators
  • Covers lectures 1-5 (this week’s lectures)

• Lecture 7, Tuesday Feb 17 in 320 Soda
• Lecture 8, Thursday Feb 19 back in 306 Soda

• See website for full schedule
Resolving Data Hazards (1)

Strategy 1:

Wait for the result to be available by freezing earlier pipeline stages \(\rightarrow\) interlocks

Feedback to Resolve Hazards

- Later stages provide dependence information to earlier stages which can *stall (or kill) instructions*
- Controlling a pipeline in this manner works provided *the instruction at stage \(i+1\) can complete without any interference from instructions in stages 1 to \(i\)* (otherwise deadlocks may occur)
Interlocks to resolve Data Hazards

Stall Condition

... r1 ← r0 + 10
... r4 ← r1 + 17

Stalled Stages and Pipeline Bubbles

Resource Usage

nop  →  pipeline bubble
Interlock Control Logic

Compare the source registers of the instruction in the decode stage with the destination register of the uncommitted instructions.

Interlock Control Logic

ignoring jumps & branches

Should we always stall if the rs field matches some rd?
not every instruction writes a register ⇒ we
not every instruction reads a register ⇒ re
### Source & Destination Registers

**R-type:**
\[
\begin{array}{cccc|c}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{func} \\
\end{array}
\]

**I-type:**
\[
\begin{array}{cccc|c}
\text{op} & \text{rs} & \text{rt} & \text{immediate16} \\
\end{array}
\]

**J-type:**
\[
\begin{array}{cccc|c}
\text{op} & \text{immediate26} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>rd ← (rs) func (rt)</td>
</tr>
<tr>
<td>ALUi</td>
<td>rt ← (rs) op imm</td>
</tr>
<tr>
<td>LW</td>
<td>rt ← M [(rs) + imm]</td>
</tr>
<tr>
<td>SW</td>
<td>M [(rs) + imm] ← (rt)</td>
</tr>
<tr>
<td>BZ</td>
<td>cond (rs)</td>
</tr>
<tr>
<td></td>
<td>true: PC ← (PC) + imm</td>
</tr>
<tr>
<td></td>
<td>false: PC ← (PC) + 4</td>
</tr>
<tr>
<td>J</td>
<td>PC ← (PC) + imm</td>
</tr>
<tr>
<td>JAL</td>
<td>r31 ← (PC), PC ← (PC) + imm</td>
</tr>
<tr>
<td>JR</td>
<td>PC ← (rs)</td>
</tr>
<tr>
<td>JALR</td>
<td>r31 ← (PC), PC ← (rs)</td>
</tr>
</tbody>
</table>

### Deriving the Stall Signal

\[ C_{\text{dest}} = \text{Case opcode} \]
- ALU    ⇒ rd
- ALUi, LW ⇒ rt
- JAL, JALR ⇒ R31

\[ C_{\text{we}} = \text{Case opcode} \]
- ALU, ALUi, LW ⇒ (ws ≠ 0)
- JAL, JALR ⇒ on
- ... ⇒ off

\[ C_{\text{re1}} = \text{Case opcode} \]
- ALU, ALUi, LW, SW, BZ, JR, JALR ⇒ on
- J, JAL ⇒ off

\[ C_{\text{re2}} = \text{Case opcode} \]
- ALU, SW ⇒ on
- ... ⇒ off


This is not the full story!
Hazards due to Loads & Stores

**Stall Condition**

Is there any possible data hazard in this instruction sequence?

What if \((r1)+7 = (r3)+5\)?

```
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
```

However, the hazard is avoided because our memory system completes writes in one cycle!

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.

Load & Store Hazards

```
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
```

\((r1)+7 = (r3)+5 \Rightarrow data\ hazard\)

However, the hazard is avoided because our memory system completes writes in one cycle!
Resolving Data Hazards (2)

Strategy 2:
Route data as soon as possible after it is calculated to the earlier pipeline stage → bypass

Bypassing

Each stall or kill introduces a bubble in the pipeline ⇒ CPI > 1

A new datapath, i.e., a bypass, can get the data from the output of the ALU to its input

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I₁)</td>
<td>r₁  ← r₀ + 10</td>
<td>IF₁</td>
<td>ID₁</td>
<td>EX₁</td>
<td>MA₁</td>
<td>WB₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I₂)</td>
<td>r₄  ← r₁ + 17</td>
<td>IF₂</td>
<td>ID₂</td>
<td>ID₂</td>
<td>ID₂</td>
<td>EX₂</td>
<td>MA₂</td>
<td>WB₂</td>
<td></td>
</tr>
<tr>
<td>(I₃)</td>
<td></td>
<td>IF₃</td>
<td>ID₃</td>
<td>ID₃</td>
<td>ID₃</td>
<td>ID₃</td>
<td>ID₃</td>
<td>EX₃</td>
<td>MA₃</td>
</tr>
<tr>
<td>(I₄)</td>
<td></td>
<td>IF₄</td>
<td>ID₄</td>
<td>ID₄</td>
<td>ID₄</td>
<td>ID₄</td>
<td>ID₄</td>
<td>EX₄</td>
<td>MA₄</td>
</tr>
<tr>
<td>(I₅)</td>
<td></td>
<td>IF₅</td>
<td>ID₅</td>
<td>ID₅</td>
<td>ID₅</td>
<td>ID₅</td>
<td>ID₅</td>
<td>EX₅</td>
<td>MA₅</td>
</tr>
</tbody>
</table>

stalled stages
Adding a Bypass

When does this bypass help?

\[ \begin{array}{l|l|l|l}
(I_1) & r1 & r1 & JAL 500 \\
(I_2) & r4 & r1+17 & r4 & r31+17 \\
      & yes & no & no \\
\end{array} \]

2/3/2009 CS152-Spring'09

The Bypass Signal

Deriving it from the Stall Signal

\[ \text{stall} = ((\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_1 \]

\[ + ((\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re}_2 \]

\[ \text{ws} = \text{Case opcode} \]

\[
\begin{align*}
\text{ALU} & \Rightarrow \text{rd} \\
\text{ALUi, LW} & \Rightarrow \text{rt} \\
\text{JAL, JALR} & \Rightarrow \text{R31}
\end{align*}
\]

\[ \text{we} = \text{Case opcode} \]

\[
\begin{align*}
\text{ALU, ALUi, LW} & \Rightarrow (\text{ws} \neq 0) \\
\text{JAL, JALR} & \Rightarrow \text{on} \\
\text{...} & \Rightarrow \text{off}
\end{align*}
\]

\[ \text{ASrc} = (\text{rs}_D = \text{ws}_E) \cdot \text{we}_E \cdot \text{re}_1 \]

Is this correct?

No because only ALU and ALUi instructions can benefit from this bypass

Split \text{we}_E into two components: we-bypass, we-stall

2/3/2009 CS152-Spring'09
### Bypass and Stall Signals

Split $w_E$ into two components: we-bypass, we-stall

- **we-bypass**
  - $w_E = \text{Case opcode}_E$
  - ALU, ALUi $\Rightarrow (ws \neq 0)$
  - ... $\Rightarrow$ off

- **we-stall**
  - $w_E = \text{Case opcode}_E$
  - LW $\Rightarrow (ws \neq 0)$
  - JAL, JALR $\Rightarrow$ on
  - ... $\Rightarrow$ off

\[ \text{ASrc} = (rs_D=ws_E).\text{we-bypass}_E . \text{re}_D \]

\[ \text{stall} = (rs_D=ws_E).\text{we-stall}_E + \]
\[ (rs_D=ws_M).\text{we}_M + (rs_D=ws_W).\text{we}_W). \text{re}_D \]
\[ + ((rt_D = ws_E).\text{we}_E + (rt_D = ws_M).\text{we}_M + (rt_D = ws_W).\text{we}_W). \text{re}_2D \]

---

### Fully Bypassed Datapath

Is there still a need for the stall signal?

\[ \text{stall} = (rs_D=ws_E). (\text{opcode}_E=LWE). (ws_E \neq 0 ). \text{re}_1D \]
\[ + (rt_D=ws_E). (\text{opcode}_E=LWE). (ws_E \neq 0 ). \text{re}_2D \]
Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly → do nothing

Guessed incorrectly → kill and restart

Next Time: Control Hazards

- Branches/Jumps
- Exceptions/Interrupts
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252