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 instruction/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

Resource Usage Diagram

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

Resources
- **IF**
- **ID**
- **EX**
- **MA**
- **WB**

<table>
<thead>
<tr>
<th>Resources</th>
<th>time</th>
<th>I0</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>t0</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>t1</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>t2</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA</td>
<td>t3</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>t4</td>
<td>I5</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I5</td>
</tr>
<tr>
<td></td>
<td>t6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I4</td>
</tr>
<tr>
<td></td>
<td>t7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I5</td>
</tr>
</tbody>
</table>

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

\[ r_4 \leftarrow r_1 \ldots \]
\[ r_1 \leftarrow \ldots \]

\[ \ldots \]
\[ r_1 \leftarrow r_0 + 10 \]
\[ r_4 \leftarrow r_1 + 17 \]

\[ r_1 \text{ is stale. Oops!} \]

Resolving Data Hazards (1)

**Strategy 1:**

*Wait for the result to be available by freezing earlier pipeline stages → 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

\[ r1 \leftarrow r0 + 10 \]
\[ r4 \leftarrow r1 + 17 \]
Stalled Stages and Pipeline Bubbles

(time) t0 t1 t2 t3 t4 t5 t6 t7 . . .
(I1) r1 ← (r0) + 10 IF1 IF2 ID1 EX1 MA1 WB1
(I2) r4 ← (r1) + 17 IF3 IF3 IF3 IF3 IF3 IF3 IF3

stalled stages

Resource Usage
nop  !     pipeline bubble

2/5/2008  CS152-Spring’08  17

Interlock Control Logic

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

2/5/2008  CS152-Spring’08  18
**Source & Destination Registers**

**R-type:**

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{func} \\
\end{array}
\]

**I-type:**

\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate16} \\
\end{array}
\]

**J-type:**

\[
\begin{array}{cccc}
\text{op} & \text{immediate26} \\
\end{array}
\]

**source(s)** | **destination**
--- | ---
ALU | rd ← (rs) func (rt)  
ALU i | rt ← (rs) op imm  
LW | rt ← M [(rs) + imm]  
SW | M [(rs) + imm] ← (rt)  
BZ | cond (rs)  
\quad true: \ PC ← (PC) + imm  
\quad false: \ PC ← (PC) + 4  
J | PC ← (PC) + imm  
JAL | r31 ← (PC), PC ← (PC) + imm  
JR | PC ← (rs)  
JAL R | r31 ← (PC), PC ← (rs)
**Deriving the Stall Signal**

\[ C_{dest} \]

ws = Case opcode
- ALU \( \Rightarrow rd \)
- ALU, LW \( \Rightarrow rt \)
- JAL, JALR \( \Rightarrow R31 \)

we = Case opcode
- ALU, ALUi, LW \( \Rightarrow (ws \neq 0) \)
- JAL, JALR \( \Rightarrow on \)
- ... \( \Rightarrow off \)

\[ C_{re} \]

re1 = Case opcode
- ALU, ALUi, LW, SW, BZ, JR, JALR \( \Rightarrow on \)
- J, JAL \( \Rightarrow off \)

re2 = Case opcode
- ALU, SW \( \Rightarrow on \)
- ... \( \Rightarrow off \)

\[ C_{stall} \]

\[ \text{stall} = \left( (r_D = ws_E).we_E + (r_D = ws_M).we_M + (r_D = ws_W).we_W \right) . re1_D + \left( (r_D = ws_E).we_E + (r_D = ws_M).we_M + (r_D = ws_W).we_W \right) . re2_D \]

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

...\( M[(r1)+7] \leftarrow (r2) \)
\( r4 \leftarrow M[(r3)+5] \)

...
Load & Store Hazards

... $M[(r1)+7] \leftarrow (r2)$
$r4 \leftarrow M[(r3)+5]$
...

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

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.

CS152 Administrivia

• Krste, office hours, Monday 1-3pm, 645 Soda
  – email for alternate time
• Henry office hours, 511 Soda
  – 9:30-10:30AM Mondays
  – 2:00-3:00PM Fridays
• First lab and practice problems this evening
• In-class quiz dates:
  – Q1: Tuesday February 19 (ISAs, microcode, simple pipelining)
  – Q2: Tuesday March 4 (memory hierarchies)
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
Adding a Bypass

When does this bypass help?

(I_1) \( r_1 \leftarrow r_0 + 10 \)  yes
(I_2) \( r_4 \leftarrow r_1 + 17 \)  no

The Bypass Signal

Deriving it from the Stall Signal

\[ \text{stall} = ((r_{D} = w_{E}) \cdot w_{E} + (r_{D} = w_{M}) \cdot w_{M} + (r_{D} = w_{W}) \cdot w_{W}) \cdot r_{1_{D}} \]
\[ + ((r_{T} = w_{E}) \cdot w_{E} + (r_{T} = w_{M}) \cdot w_{M} + (r_{T} = w_{W}) \cdot w_{W}) \cdot r_{2_{D}} \]

\( w_{S} = \text{Case opcode} \)
- ALU \( \Rightarrow \) rd
- ALUi, LW \( \Rightarrow \) rt
- JAL, JALR \( \Rightarrow \) R31

\( w_{E} = \text{Case opcode} \)
- ALU, ALUi, LW \( \Rightarrow (w_{S} \neq 0) \)
- JAL, JALR \( \Rightarrow \) on
- ... \( \Rightarrow \) off

\( \text{ASrc} = (r_{S_{D}} = w_{S_{E}}) \cdot w_{E} \cdot r_{1_{D}} \)

Is this correct?

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

Split \( w_{E} \) into two components: we-bypass, we-stall
**Bypass and Stall Signals**

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

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

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

- **$ASrc = (rs_D = ws_E).we-bypass_E . re_1_D$**

- **stall = ((rs_D = ws_E).we-stall$_E$ +
  
  $(rs_D = ws_M).we_M + (rs_D = ws_W).we_W). re_1_D$

  + ($(rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W). re_2_D$**

---

**Fully Bypassed Datapath**

Is there still a need for the stall signal?

- **stall = (rs_D = ws_E). (opcode$_E$ = LW$_E$). (ws$_E \neq 0$). re$_1_D$
  
  + (rt_D = ws_E). (opcode$_E$ = LW$_E$). (ws$_E \neq 0$). re$_2_D$**
Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly → do nothing

Guessed incorrectly → kill and restart

Instruction to Instruction Dependence

• What do we need to calculate next PC:

  – For Jumps
    » Opcode, offset and PC
  – For Jump Register
    » Opcode and Register value
  – For Conditional Branches
    » Opcode, PC, Register (for condition), and offset
  – For all others
    » Opcode and PC
**PC Calculation Bubbles**

**Time**

\[
\begin{align*}
(I_1) & \quad r1 \leftarrow (r0) + 10 \\
(I_2) & \quad r3 \leftarrow (r2) + 17 \\
(I_3) & \quad \text{...} \\
(I_4) & \quad \text{...}
\end{align*}
\]

**Resource Usage**

\[
\text{nop} \Rightarrow \text{pipeline bubble}
\]

---

**Speculate next address is PC+4**

- **PC Src**: \((\text{pc}+4 / \text{jabs} / \text{rind} / \text{br})\)

  - **Add**: \(0x4\)
  - **Instr Memory**: \(104\)
  - **IR**: \(I_1\)
  - **Add**: \(I_2\)
  - **IR**: \(I_3\)
  - **Add**: \(I_4\)

- **IR**
  - **Jump?**
  - **IR**: \(I_1\)

- **IR**: \(I_2\)
  - **Jump?**

- **IR**: \(I_3\)

- **IR**: \(I_4\)

A jump instruction kills (not stalls) the following instruction.
Pipelining Jumps

To kill a fetched instruction -- Insert a mux before IR

Any interaction between stall and jump?

IRSrc = Case opcode
  J, JAL => nop
  ... => IM

Jump Pipeline Diagrams

Resource Usage

nop => pipeline bubble
Pipelining Conditional Branches

Branch condition is not known until the execute stage. What action should be taken in the decode stage?

If the branch is taken:
- Kill the two following instructions
- The instruction at the decode stage is not valid

⇒ Stall signal is not valid
Pipelining Conditional Branches

If the branch is taken

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>096</td>
<td>ADD</td>
</tr>
<tr>
<td>I2</td>
<td>100</td>
<td>BEQZ r1 200</td>
</tr>
<tr>
<td>I3</td>
<td>104</td>
<td>ADD</td>
</tr>
<tr>
<td>I4</td>
<td>304</td>
<td>ADD</td>
</tr>
</tbody>
</table>

- kill the two following instructions
- the instruction at the decode stage is not valid

\[ \Rightarrow \text{stall signal is not valid} \]

New Stall Signal

\[
\text{stall} = \left( (\text{rs}_D = \text{ws}_E).\text{we}_E + (\text{rs}_D = \text{ws}_M).\text{we}_M + (\text{rs}_D = \text{ws}_W).\text{we}_W).\text{re}_1D \\
+ (\text{rt}_D = \text{ws}_E).\text{we}_E + (\text{rt}_D = \text{ws}_M).\text{we}_M + (\text{rt}_D = \text{ws}_W).\text{we}_W).\text{re}_2D \\
\right) \cdot !((\text{opcode}_E = \text{BEQZ}).z + (\text{opcode}_E = \text{BNEZ}).!z)
\]

Don’t stall if the branch is taken. Why?

Instruction at the decode stage is invalid
Control Equations for PC and IR Muxes

\[ \text{PCSrc} = \text{Case opcode}_E \]
\[ \text{BEQZ}.z, \text{BNEZ}.!z \Rightarrow \text{br} \]
\[ \ldots \Rightarrow \]
\[ \text{Case opcode}_D \]
\[ \text{J, JAL} \Rightarrow \text{jabs} \]
\[ \text{JR, JALR} \Rightarrow \text{rind} \]
\[ \ldots \Rightarrow \text{pc+4} \]

\[ \text{IRSrc}_D = \text{Case opcode}_E \]
\[ \text{BEQZ}.z, \text{BNEZ}.!z \Rightarrow \text{nop} \]
\[ \ldots \Rightarrow \]
\[ \text{Case opcode}_D \]
\[ \text{J, JAL, JR, JALR} \Rightarrow \text{nop} \]
\[ \ldots \Rightarrow \text{IM} \]

\[ \text{IRSrc}_E = \text{Case opcode}_E \]
\[ \text{BEQZ}.z, \text{BNEZ}.!z \Rightarrow \text{nop} \]
\[ \ldots \Rightarrow \text{stall.nop + } !\text{stall.IR}_D \]

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction.

Branch Pipeline Diagrams (resolved in execute stage)

\[ \text{time} \]
\[ \begin{array}{cccccccc}
\text{t0} & \text{t1} & \text{t2} & \text{t3} & \text{t4} & \text{t5} & \text{t6} & \text{t7} \\
(I_1) & 096: \text{ADD} & IF_1 & ID_1 & EX_1 & MA_1 & WB_1 \\
(I_2) & 100: \text{BEQZ} 200 & IF_2 & ID_2 & EX_2 & MA_2 & WB_2 \\
(I_3) & 104: \text{ADD} & IF_3 & ID_3 & EX_3 & MA_3 & WB_3 \\
(I_4) & 108: \text{ADD} & IF_4 & ID_4 & EX_4 & MA_4 & WB_4 \\
(I_5) & 304: \text{ADD} & IF_5 & ID_5 & EX_5 & MA_5 & WB_5 \\
\end{array} \]

\[ \text{Resource Usage} \]

\[ \begin{array}{cccccccc}
\text{IF} & \text{I}_1 & \text{I}_2 & \text{I}_3 & \text{I}_4 & \text{I}_5 \\
\text{ID} & \text{I}_1 & \text{I}_2 & \text{I}_3 & \text{nop} & \text{I}_5 \\
\text{EX} & \text{I}_1 & \text{I}_2 & \text{nop} & \text{nop} & \text{I}_5 \\
\text{MA} & \text{I}_1 & \text{I}_2 & \text{nop} & \text{nop} & \text{I}_5 \\
\text{WB} & \text{I}_1 & \text{I}_2 & \text{nop} & \text{nop} & \text{I}_5 \\
\end{array} \]

\[ \text{nop} \Rightarrow \text{pipeline bubble} \]
Reducing Branch Penalty
(resolve in decode stage)

• One pipeline bubble can be removed if an extra comparator is used in the Decode stage

Branch Delay Slots
(expose control hazard to software)

• Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  – gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

| I_1 | 096 | ADD | Delay slot instruction executed regardless of branch outcome |
| I_2 | 100 | BEQZ  r1 200 |
| I_3 | 104 | ADD |
| I_4 | 304 | ADD |

• Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*
Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

*Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.*

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