Previous Lecture

- Measurements and metrics: Performance, Cost, Dependability, Power
- Guidelines and principles in the design of computers

- Monday 8/10 → Friday 12/10
- Monday 15/10 → Friday 19/10
- Wednesday 17/10 → TBD
Outline

- Processor review
- Hazards
  - Structural
  - Data
  - Control
- Performance
- Exceptions
Clock Cycle

- **Old days:** 10 levels of gates
- **Today:** determined by numerous time-of-flight issues + gate delays
  - clock propagation, wire lengths, drivers
Datapath vs Control

- **Datapath**: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals

- **Controller**: State machine to orchestrate operation on the data path
  - Based on desired function and signals
“Typical” RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: 32bit MIPS

**Register-Register**

- **Op**
- **Rs1**
- **Rs2**
- **Rd**
- **Op**

**Register-Immediate**

- **Op**
- **Rs1**
- **Rd**
- Immediate

**Branch**

- **Op**
- **Rs1**
- **Rs2/Opx**
- Immediate

**Jump / Call**

- **Op**
- Target

**Example:**
- `lw $2, 100($5)`
- `add $4, $5, $6`
- `beq $3, $4, label`
Example Execution Steps

**Obtain instruction from program storage**

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction

5-stage execution is a bit different (see next slides)…
Pipelining: Latency vs Throughput

Pipelining doesn’t help **latency** of single task, it helps **throughput** of entire workload.
5-stage Instruction Execution - Datapath

IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

rslt <= A \text{ op}_{IRop} B
WB <= rslt
Reg[IR_{rd}] <= WB

- Data stationary control
  - local decode for each instruction phase / pipeline stage
Visualizing Pipelining

Time (clock cycles)

Cycle 1       Cycle 2       Cycle 3       Cycle 4       Cycle 5       Cycle 6       Cycle 7

Ifetch → Reg → ALU → DMem → Reg

Ifetch → Reg → ALU → DMem → Reg

Ifetch → Reg → ALU → DMem → Reg

Ifetch → Reg → ALU → DMem → Reg

Ifetch → Reg → ALU → DMem → Reg

Ifetch → Reg → ALU → DMem → Reg
5-stage Instruction Execution - Control

Pipeline Registers: IR, A, B, r, WB
Limits in Pipelining

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
Example of Structural Hazard

**Time (clock cycles)**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td>Instr 1</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td>Instr 2</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td>Instr 3</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
</tr>
<tr>
<td>Instr 4</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
</tr>
</tbody>
</table>
Example of Structural Hazard

**Time (clock cycles)**

```
<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load</strong></td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td><strong>Instr 1</strong></td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td><strong>Instr 2</strong></td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td><strong>Stall</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instr 3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**How do you “bubble” this pipe (if instr1 = load)?**
Speed Up Equation of Pipelining

\[
\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

For simple RISC pipeline, Ideal CPI = 1:

\[
\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}
\]
Machine A: Dual read ported memory (“Harvard Architecture”)

Machine B: Single read ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Ideal CPI = 1 for both

Suppose that Loads are 40% of instructions executed

\[
\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}
\]

\[
= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05}
\]

\[
= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
\]

Machine A is 1.33 times faster
Data Hazard

\[ \text{Instr.} \]
\[ \text{Order} \]
\[ \text{Time (clock cycles)} \]

**Instr.**
- `add r1, r2, r3`
- `sub r4, r1, r3`
- `and r6, r1, r7`
- `or r8, r1, r9`
- `xor r10, r1, r11`

**Order**

**Time (clock cycles)**

**IF**
- `Ifetch`
- `Reg`
- `ALU`
- `DMem`
- `Reg`

**ID/RF**
- `Ifetch`
- `Reg`
- `ALU`
- `DMem`
- `Reg`

**EX**
- `Ifetch`
- `Reg`
- `ALU`
- `DMem`
- `Reg`

**MEM**
- `Ifetch`
- `Reg`
- `ALU`
- `DMem`
- `Reg`

**WB**
- `Ifetch`
- `Reg`
- `ALU`
- `DMem`
- `Reg`
Read After Write

- Read After Write (RAW)
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

\begin{align*}
I: & \text{add } r1, r2, r3 \\
J: & \text{sub } r4, r1, r3
\end{align*}

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.
Write After Read

- **Write After Read (WAR)**
  - Instr$_J$ writes operand *before* Instr$_I$ reads it

  \[ I: \text{sub} \ r4,r1,r3 \]
  \[ J: \text{add} \ r1,r2,r3 \]
  \[ K: \text{mul} \ r6,r1,r7 \]

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Write After Write

- **Write After Write (WAW)**
  Instr\(_j\) writes operand *before* Instr\(_i\) writes it.
  
  \[
  \begin{align*}
  I: & \quad \text{sub} \ r1, r4, r3 \\
  J: & \quad \text{add} \ r1, r2, r3 \\
  K: & \quad \text{mul} \ r6, r1, r7
  \end{align*}
  \]

- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in more complicated pipes
Forwarding to avoid data hazards

\begin{itemize}
\item add r1, r2, r3
\item sub r4, r1, r3
\item and r6, r1, r7
\item or r8, r1, r9
\item xor r10, r1, r11
\end{itemize}
HW Change for Forwarding

What circuit detects and resolves this hazard?
Why we need forwarding lines for both inputs of the ALU?
Forwarding to Avoid LW-SW Data Hazard

Time (clock cycles)

Instr. | Order
--- | ---
add r1, r2, r3
lw r4, 0(r1)
sw r4, 12(r1)
or r8, r6, r9
xor r10, r9, r11
Data Hazard Even with Forwarding

\[ \text{lw } r1, 0(r2) \]
\[ \text{sub } r4, r1, r6 \]
\[ \text{and } r6, r1, r7 \]
\[ \text{or } r8, r1, r9 \]
Data Hazard Even with Forwarding

**Time (clock cycles)**

**Instruction Order**

- `lw r1, 0(r2)`
- `sub r4, r1, r6`
- `and r6, r1, r7`
- `or r8, r1, r9`
Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[ a = b + c; \]
\[ d = e - f; \]

assuming \( a, b, c, d, e, \) and \( f \) in memory.

<table>
<thead>
<tr>
<th>Slow code</th>
<th>Fast code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td>LW Rb,b</td>
</tr>
<tr>
<td>LW Rc,c</td>
<td>LW Rc,c</td>
</tr>
<tr>
<td>ADD Ra,Rb,Rc</td>
<td>LW Re,e</td>
</tr>
<tr>
<td>SW a,Ra</td>
<td>ADD Ra,Rb,Rc</td>
</tr>
<tr>
<td>LW Re,e</td>
<td>LW Rf,f</td>
</tr>
<tr>
<td>LW Rf,f</td>
<td>SW a,Ra</td>
</tr>
<tr>
<td>SUB Rd,Re,Rf</td>
<td>SUB Rd,Re,Rf</td>
</tr>
<tr>
<td>SW d,Rd</td>
<td>SW d,Rd</td>
</tr>
</tbody>
</table>
Control Hazard on Branches
Three Stage Stall

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
36: xor r10, r1, r11

What do you do with the 3 instructions in between?
How do you do it?
Where is the “commit”? 
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!

- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier

- MIPS branch tests if register = 0 or ≠ 0

- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3
Pipelined MIPS Datapath

- Interplay of instruction set design and cycle time.
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But haven’t calculated branch target address in MIPS
    - MIPS still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome
Four Branch Hazard Alternatives

#4: Delayed Branch

- Define branch to take place \textbf{AFTER} a following instruction

\begin{align*}
\text{branch instruction} \\
\text{sequential successor}_1 \\
\text{sequential successor}_2 \\
\vdots \\
\text{sequential successor}_n \\
\text{branch target if taken}
\end{align*}

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline

- MIPS uses this
Scheduling Branch Delay Slots

A. From before branch

\[
\text{add } \$1,\$2,\$3 \\
\text{if } \$2=0 \text{ then} \\
\text{delay slot}
\]

becomes

\[
\text{if } \$2=0 \text{ then} \\
\text{add } \$1,\$2,\$3
\]

B. From branch target

\[
\text{sub } \$4,\$5,\$6 \\
\text{if } \$1=0 \text{ then} \\
\text{delay slot}
\]

becomes

\[
\text{add } \$1,\$2,\$3 \\
\text{if } \$1=0 \text{ then} \\
\text{sub } \$4,\$5,\$6
\]

C. From fall through

\[
\text{add } \$1,\$2,\$3 \\
\text{if } \$1=0 \text{ then} \\
\text{delay slot}
\]

becomes

\[
\text{add } \$1,\$2,\$3 \\
\text{if } \$1=0 \text{ then} \\
\text{sub } \$4,\$5,\$6
\]

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the \text{sub} instruction may need to be copied, increasing IC
- In B and C, must be okay to execute \text{sub} when branch fails
Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper
Evaluating Branch Alternatives

Pipeline speedup = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}

<table>
<thead>
<tr>
<th>Branch type</th>
<th>Unconditional branch</th>
<th>Conditional branch, untaken</th>
<th>Conditional branch, taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional branch</td>
<td>4%</td>
<td>6%</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Penalty unconditional</th>
<th>Penalty untaken</th>
<th>Penalty taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Unconditional branches</th>
<th>Untaken conditional branches</th>
<th>Taken conditional branches</th>
<th>All branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency of event</td>
<td>4%</td>
<td>6%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>Stall pipeline</td>
<td>0.08</td>
<td>0.18</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>0.08</td>
<td>0.18</td>
<td>0.20</td>
<td>0.46</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>0.08</td>
<td>0.00</td>
<td>0.30</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Deep pipeline in this example
Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode

- **Interrupt**: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

- **Problem**: It must appear that the exception or interrupt must appear between 2 instructions ($I_i$ and $I_{i+1}$)
  - The effect of all instructions up to and including $I_i$ is totally complete
  - No effect of any instruction after $I_i$ can take place

- The interrupt (exception) handler either aborts program or restarts at instruction $I_{i+1}$
Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.
Summary: Pipelining

- Next time: Read Appendix A
- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity