Review: Pipelining

- Key to pipelining: smooth flow
  - Making all instructions the same length can increase performance!

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction

- Data hazards must be handled carefully:
  - RAW (Read-After-Write) data hazards handled by forwarding
  - WAW (Write-After-Write) and WAR (Write-After-Read) hazards don’t exist in 5-stage pipeline

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
  - Change in programmer semantics to make hardware simpler

Recap: Data Hazards

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>MemOpFetch</th>
<th>OpFetch</th>
<th>Exec</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Structural Hazard

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>OpFetch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Control Hazard

RAW (read after write) Data Hazard

<table>
<thead>
<tr>
<th>IF</th>
<th>DCD</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

WAW Data Hazard (write after write)

WAR Data Hazard (write after read)

Recap: Data Stationary Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later
Hazard Detection

• Suppose instruction \( i \) is about to be issued and a predecessor instruction \( j \) is in the instruction pipeline.

New Inst \[
\begin{array}{c}
\text{Inst } I \\
\text{Inst } J
\end{array}
\]

Instruction Movement:

Window on execution:
Only pending instructions can cause hazards

• A RAW hazard exists on register \( \rho \) if \( \rho \in \text{Rregs}(i) \cap \text{Wregs}(j) \)
  – Keep a record of pending writes (for inst’s in the pipe) and compare with operand regs of current instruction.
  – When instruction issues, reserve its result register.
  – When on operation completes, remove its write reservation.

• A WAW hazard exists on register \( \rho \) if \( \rho \in \text{Wregs}(i) \cap \text{Wregs}(j) \)

• A WAR hazard exists on register \( \rho \) if \( \rho \in \text{Wregs}(i) \cap \text{Rregs}(j) \)

Resolve RAW by “forwarding” (or bypassing)

• Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe

• Increase muxes to add paths from pipeline registers

• Data Forwarding = Data Bypassing

Record of Pending Writes In Pipeline Registers

Current operand registers
Pending writes
• hazard <=

\[
\begin{align*}
&\text{RS} = \text{Wex} \quad \text{or} \quad \text{Wex} \\
&\text{RT} = \text{Wab} \quad \text{or} \quad \text{Wab}
\end{align*}
\]

What about memory operations?

• If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!

• What does delaying WB on arithmetic operations cost?
  – cycles?
  – hardware?

• What about data dependence on loads?
  \( R1 \leftarrow R4 + R5 \)
  \( R2 \leftarrow \text{Mem}[R2 + I] \)
  \( R3 \leftarrow R2 + R1 \)

\Rightarrow “Delayed Loads”

• Can recognize this in decode stage and introduce bubble while stalling fetch stage (hint for lab 5!)

• Tricky situation:
  \( R1 \leftarrow \text{Mem}[R2 + I] \)
  \( \text{Mem}[R3 + 34] \leftarrow R1 \)

Handle with bypass in memory stage!
Compiler Avoiding Load Stalls:

Recall: MIPS I had no pipeline stalls
- “Microprocessor without Interlocking Pipeline Stages
- Consequently, the “Unscheduled” code above would be wrong

% loads stalling pipeline

- gcc scheduled: 54%, unscheduled: 46%
- spice: scheduled: 31%, unscheduled: 69%
- tex: scheduled: 25%, unscheduled: 75%

What about Interrupts, Traps, Faults?

- External Interrupts:
  - Allow pipeline to drain, Fill with NOPs
  - Load PC with interrupt address
- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state
- Recall: Precise Exceptions ⇒ State of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations

Question: Critical Path???

- Bypass path is invariably trouble
- Options?
  - Make logic really fast
  - Move forwarding after muxes
    » Problem: screws up branches that require forwarding!
    » Use same tricks as “carry-skip” adder to fix this?
    » This option may just push delay around….!
  - Insert an extra cycle for branches that need forwarding?
    » Or: hit common case of forwarding from EX stage and stall for forward from memory?

Exception/Interrupts: Implementation questions

- 5 instructions, executing in 5 different pipeline stages!
- Who caused the interrupt?
  
<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation; memory error</td>
</tr>
</tbody>
</table>

- How do we stop the pipeline? How do we restart it?
- Do we interrupt immediately or wait?
- How do we sort all of this out to maintain preciseness?
Exception Handling

- Detect bad instruction address
- Detect bad instruction
- Detect overflow
- Detect bad data address

Allow exception to take effect

Resolution: Freeze above & Bubble Below

- Flush accomplished by setting “invalid” bit in pipeline

Another look at the exception problem

- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don’t act on exception until it reaches WB stage
- Handle interrupts through “faulting noop” in IF stage
- When instruction reaches end of MEM stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

FYI: MIPS R3000 clocking discipline

- 2-phase non-overlapping clocks
- Pipeline stage is two (level sensitive) latches

Edge-triggered
MIPS R3000 Instruction Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Decode Reg. Read</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resource Usage**

<table>
<thead>
<tr>
<th>TLB</th>
<th>I-Cache</th>
<th>RF</th>
<th>ALU</th>
<th>D-Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write in phase 1, read in phase 2 ⇒ eliminates bypass from WB

Recall: Data Hazard on r1

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

With MIPS R3000 pipeline, no need to forward from WB stage

MIPS R3000 Multicycle Operations

- Use control word of local stage to step through multicycle operation
- Stall all stages above multicycle operation in the pipeline
- Drain (bubble) stages below it
- Alternatively, launch multiply/divide to autonomous unit, only stall pipe if attempt to get result before ready
  - This means stall mflo/mfhi in decode stage if multiply/divide still executing
  - Extra credit in Lab 5 does this

Ex: Multiply, Divide, Cache Miss

Is CPI = 1 for our pipeline?

- Remember that CPI is an “Average # cycles/inst

- CPI here is 1, since the average throughput is 1 instruction every cycle.
- What if there are stalls or multi-cycle execution?
- Usually CPI > 1. How close can we get to 1??
Recall: Compute CPI?

- Start with Base CPI
- Add stalls

\[
CPI = CPI_{\text{base}} + CPI_{\text{stall}}
\]

\[
CPI_{\text{stall}} = STALL_{\text{type-1}} \times f_{\text{type-1}} + STALL_{\text{type-2}} \times f_{\text{type-2}}
\]

- Suppose:
  - \(CPI_{\text{base}} = 1\)
  - \(f_{\text{branch}} = 20\%\), \(f_{\text{load}} = 30\%\)
  - Suppose branches always cause 1 cycle stall
  - Loads cause a 100 cycle stall 1% of time
- Then: \(CPI = 1 + (1 \times 0.20) + (100 \times 0.30 \times 0.01) = 1.5\)
- Multicycle? Could treat as:
  \[CPI_{\text{stall}} = (\text{CYCLES} - CPI_{\text{base}}) \times f_{\text{inst}}\]

Administrivia

- Should finish reading Chapter 6
- Dynamic scheduling techniques discussed in the Other Hennessy & Patterson book:
  - “Computer Architecture: A Quantitative Approach”
  - Chapter 4
- Lab 5 coming out Friday
  - Pipelining
  - Real Hardware! (Pushing down to the boards)
  - Sorta have three weeks for it (including spring break)

Case Study: MIPS R4000 (200 MHz)

- 8 Stage Pipeline:
  - IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS—second half of access to instruction cache.
  - RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF—data fetch, first half of access to data cache.
  - DS—second half of access to data cache.
  - TC—tag check, determine whether the data cache access hit.
  - WB—write back for loads and register-register operations.
- 8 Stages:
  What is impact on Load delay? Branch delay? Why?

Administrivia: Be careful about clock edges in lab5!

- Since Register have edge-triggered write:
  - Must have everything set up at end of memory stage
  - This means that “M” register here is not necessary!
- Also, Memories will be synchronous
  - Need to setup addresses and values in advance.
Case Study: MIPS R4000

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FP adder</td>
<td>Mantissa ADD stage</td>
</tr>
<tr>
<td>D</td>
<td>FP divider</td>
<td>Divide pipeline stage</td>
</tr>
<tr>
<td>E</td>
<td>FP multiplier</td>
<td>Exception test stage</td>
</tr>
<tr>
<td>M</td>
<td>FP multiplier</td>
<td>First stage of multiplier</td>
</tr>
<tr>
<td>N</td>
<td>FP multiplier</td>
<td>Second stage of multiplier</td>
</tr>
<tr>
<td>R</td>
<td>FP adder</td>
<td>Rounding stage</td>
</tr>
<tr>
<td>S</td>
<td>FP adder</td>
<td>Operand shift stage</td>
</tr>
<tr>
<td>U</td>
<td>Unpack FP numbers</td>
<td></td>
</tr>
</tbody>
</table>

MIPS FP Pipe Stages

<table>
<thead>
<tr>
<th>FP Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>U</td>
<td>E+M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td>D²</td>
<td>...</td>
<td>D+R</td>
<td>D+R</td>
<td>D+R</td>
<td>D+R</td>
</tr>
<tr>
<td>Square root</td>
<td>U</td>
<td>E</td>
<td>(A+R)¹⁰⁸</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute value</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP compare</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stages:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>First stage of multiplier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Second stage of multiplier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Rounding stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Operand shift stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Unpack FP numbers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R4000 Performance

- Not ideal CPI of 1:
  - FP structural stalls: Not enough FP hardware (parallelism)
  - FP result stalls: RAW data hazard (latency)
  - Branch stalls (2 cycles + unfilled slots)
  - Load stalls (1 or 2 clock cycles)
Summary

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction

- Data hazards must be handled carefully:
  - RAW data hazards handled by forwarding
  - WAW and WAR hazards don’t exist in 5-stage pipeline

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
  - Exceptions in 5-stage pipeline recorded when they occur, but acted on only at WB (end of MEM) stage
  - Must flush all previous instructions

- More performance from deeper pipelines, parallelism