Lecture 04
Instruction Pipelining (Appendix A and C)

Outline
- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion

5 Steps of a MIPS Data Path (with pipeline latches)
Figure C.22

Control Hazard on Branch - Three Cycle Stall
(Caused if Decide Branches in 4th Stage)

Time (clock cycles)

- Old simple design put branch completion in stage 4 (Mem)
- What can be done with the 3 instructions between beq & xor?
- Code between beq & xor must not start until know beq not branch => 3 stalls
- Adding 3 cycle stall after every branch (1/7 of instructions) => CPI += 3/7. BAD!
Branch Stall Impact if Commit in Stage 4

- If CPI = 1 and 15% of instructions are branches,
  - Stall 3 cycles => new CPI = 1.45 \((1 + 3 \times 0.15)\) Too much!

- Question: When do (or can) we find out that the PC needs to be modified?
  - Answer: In pipeline stage ID of a branch instruction
  - So, if a branch is not-taken (i.e., if the PC is not modified), need a one-cycle delay

- Question: When is a taken branch’s address known?
  - ALU used to compute, so EX stage
  - Need two (or three) cycle delay

Control hazards (also see Fig. C.24, 25, 28, 29)

- Two-part solution:
  - Determine sooner whether branch taken or not, AND
  - Compute taken branch address earlier
  - MIPS branch tests if register = 0 or \(\neq 0\)

- Original 1986 MIPS Solution:
  - Move zero_test to ID/RF (Inst. decode & Reg. fetch) stage from MEM
  - Add extra adder to calculate new PC (Program Counter) in ID/RF stage
  - Result is 1 clock cycle penalty for branch versus 3 when decided in MEM

- If branch in 15% of instructions, then what is the average CPI?
  - Per cycle, have 85% of instructions executing in 1 cycle and 15% of instructions executing in 2 cycles
  - An average of \(0.85 \times 1 + 0.15 \times 2 = 1.15\). Worse by 15%

New Pipelined MIPS Data Path: Faster Branch

- Example of interplay of instruction set design and cycle time.
- Example of interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

- #1: Stall until branch direction is clearly known
- #2: Predict Branch Not Taken – Easy Solution
  - Execute the next instructions in sequence
  - PC+4 already calculated, so use it to get next instruction
  - Nullify bad instructions in pipeline if branch is actually taken
  - Nullify easier since pipeline state updates are late (MEM, WB)
  - 47% MIPS branches not taken on average

- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But have not calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Some other CPUs: branch target known before outcome

- #4: But Have No Branch Taken
Last of Four Branch Hazard Alternatives

- **#4: Delayed Branch**
  - Used Only in 1st MIPS "Killer Micro"
  - Define branch to take place AFTER a following instruction

  \[
  \text{branch instruction} \rightarrow \text{branch target if taken} \\
  \text{sequential successor}_1 \quad \text{sequential successor}_2 \quad \text{sequential successor}_n \quad \text{Branch delay of length } n
  \]

  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
  - Later versions of MIPS did not use this; deeper pipeline

Scheduling branch delay slot

- If taken from before branch,
  - branch must not depend on rescheduled instruction
  - always improves performance
- If taken from branch target,
  - must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate instructions
  - improve performance when branch taken
- If taken from fall through,
  - must be OK to execute instructions if branch taken
  - improve performance when branch not taken

Delayed Branch Not Used in Modern CPUs

- 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; cannot fill 2 or more

  **Delayed Branch downside?**
  - As processor designs use deeper pipelines and multiple issue, the branch delay grows and needs many more delay slots
  - Delayed branching soon lost effectiveness and popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors soon permitted dynamic approaches that keep records of branch locations, taken/not-taken decisions, and target addresses
Evaluating Branch Alternatives

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

**Assume:**
- 4% unconditional branch,
- 6% conditional branch-untaken,
- 10% conditional branch-taken

*Also see another example in Fig-C.15*

**Scheduling scheme**

<table>
<thead>
<tr>
<th>Branch penalty</th>
<th>Stall pipeline</th>
<th>Speedup v. Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.20</td>
<td>1.33</td>
</tr>
<tr>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>0.5</td>
<td>1.10</td>
<td>1.45</td>
</tr>
</tbody>
</table>

**Sample calculation**

- \( 1.60 = 1 + 3 \times (4+10+6)\% \) (4.5=5/1.10) (1.45=1.6/1.1)
-\( 1.20 = 1 + 1 \times (4+10)\% \) (even for taken case, we need time to calculate taken target)
- \( 1.14 = 1 + 1 \times (4+10)\% \) (refetch for jump, taken-branch)

Another Problem with Pipelining

- **Question:** What makes pipelining hard to implement?
- **Answer:** Surprises

**Technical names for surprises:**
- exceptions
- faults
- interrupts

Exception (and Interrupt)

- **Exception:** An unusual event happens to an instruction during its execution (caused by instructions executing)
  - Request for I/O
  - Arithmetic troubles: overflow or underflow
  - Page fault: data not in (physical) memory
  - Illegal address, giving a memory protection violation
  - Hardware failure

- **Interrupt:** A hardware signal to switch the processor to a new instruction stream (not directly caused by code)
  - A sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

Classifying exceptions

- **Synchronous:** Repeatable every time
  - Example: DIV R2, R2, R0
- **Asynchronous:** Caused by external events like hardware failure and devices external to processor and memory
  - User requested: user task asks for it (e.g., breakpoint)
  - Coerced: cannot be predicted by user

- **User maskable:** Can be disabled by user task
  - Example: arithmetic exception
- **Non-maskable:** Cannot be turned off
  - Example: hardware failure
Classifying exceptions (cont.)

- Within instruction: prevents instruction from completing
- Between instructions: no instruction prevented
- Terminating: stops the task
- Resuming: task can continue

Machines that handle exceptions, save the state, and then restart correctly are said to be restartable.

Categorizing exceptions (cont.)

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. asynch.</th>
<th>User request vs. coerce</th>
<th>User maskable vs. not</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned memory access</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Mem. prot. violation</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware malfunction</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>

Precise Exception Handling

- Must seem as if the exception or interrupt appeared between 2 instructions \(I_i\) and \(I_{i+1}\) although several instructions were executing at the time
- All instructions up to and including \(I_i\) are totally completed
- No effect of any instruction after \(I_i\) is allowed to be saved
- After a precise interrupt handling, the interrupt (exception) handler either aborts the program or restarts at instruction \(I_{i+1}\)
- This is the right way to do it, but sometimes architects/manufacturers take shortcuts
The most difficult exceptions...

- ... are those that occur within EX or MEM stages and also must be handled in a restartable way

- Steps to save the pipeline state safely:
  - Force the next IF to get a "trap instruction"
  - until the trap is taken, turn off all "writes" for the faulting instruction and for all instructions that follow it
  - what does the trap do?
    - The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault
  - the task is then resumed, using the saved PC and the MIPS instruction RFE or something like it

- Note: May need to save several PCs if delayed branches are involved

When do MIPS exceptions occur?

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault, misaligned memory access,</td>
</tr>
<tr>
<td></td>
<td>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, misaligned memory access,</td>
</tr>
<tr>
<td></td>
<td>memory-protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None!!!</td>
</tr>
</tbody>
</table>

Examples of exception handling

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Handle the MEM fault first, then restart

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

How is this done?

- Answer: Don't handle exceptions until the WB stage
  - each instruction has an associated status vector that keeps track of faults
  - any bit set in the status vector turns off register writes and memory writes
  - in WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled before faults for later instructions
  - Unfortunately, will need to violate this later (for instructions that don’t reach WB in proper order)
Precise Exceptions in Static Pipelines

Key observation: "Architected" states change only in memory (M) and register write (W) stages.

Commitment

- When an instruction is guaranteed to complete, it is committed.
- Life is easier if no instruction changes the permanent machine state before it is committed.
- In MIPS, commitment occurs at the end of the MEM stage - that’s why register update occurs in the stage after that.
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started.

And In Conclusion: Control and Pipelining

- 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 by stalling:
  - Structural: need more HW resources
  - Data (RAW, WAR, WAW): need forwarding, compiler scheduling
  - Control: delayed branch or branch (taken/not-taken) prediction

- Exceptions and interrupts add complexity.