Lecture Topics

- Pipelining
  - Hazards and Stalls
    - Effect of Stalls on Pipeline Performance
    - Structural hazards
    - Data Hazards

Reference:
- Appendix C: Sections C.1, C.2

Pipelining in Processors

- Exploit parallelism in sequential instruction stream
  - Resources (e.g. ALU, memory, register file) can be used concurrently by different instructions
  - Multiple instructions processed in parallel → More instructions completed per unit time → Higher throughput (performance)

Pipelining and ISA

- What is desirable in instruction sets for pipelining
  - Variable length instructions vs. fixed length instructions?
  - Memory operands supported for any instruction or memory operands only in loads/stores?
  - Register operands in different places in each instruction format vs. in the same place in different instruction formats?
MIPS Choices for Pipelining

- 32-bit fixed instruction format
- Memory access only via load/store instruction
- Opcode field for all instruction formats in the same place
- Source registers for R-type and I-type instruction formats in the same place
- Single addressing mode for load/store instructions
- Simple branch conditions

5-Stage MIPS Pipeline

At any given time, each pipeline stage processes a different instruction.
Register file appears in two stages: read in stage-2 and written in stage-5

How do we ensure that each stage has the correct inputs for the instruction that is being processed by that stage?

Information needed by an instruction is carried through the pipeline (via pipeline registers) as the instruction proceeds from one stage to the next.
Representing Pipelined Execution

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Performance

- Pipelining
  - At best no impact on latency
    - Still need to wait “n” stages (cycles) for completion of instruction
  - Improves “throughput”
    - No single instruction executes faster but overall throughput is higher
    - Average instruction execution time decreases
    - Successive instructions complete in each successive cycle (no 5 cycle wait between instructions)
- Reality
  - Clock determined by slowest stage
  - Pipeline overhead
    - Clock skew
    - Register delay
    - Pipeline fill and drain

Pipeline Performance: Example

- Ideal case
  - Balanced pipeline (each stage has the same delay)
  - Zero overhead due to clock skew and pipeline registers
  - Ignore pipeline fill and drain overheads

\[
\text{Speedup from pipelining} = \frac{(\text{Average time per instruction})_{\text{nonpipelined}}}{(\text{Average time per instruction})_{\text{pipelined}}}
\]

\[
\text{Speedup from pipelining} = \text{Number of pipeline stages} \quad \text{(in an ideal case)}
\]

- Example: A program consisting of 500 instructions is executed on a 5-stage processor. How many cycles would be required to complete the program, (i) without pipelining, (ii) with pipelining? Assume ideal overlap in case of pipelining.

- Solution:
  - Without pipelining: Each instruction will require 5 cycles. There will be no overlap amongst successive instructions.
    \[
    \text{Number of cycles} = 500 \times 5 = 2500
    \]
  - With pipelining: Each pipeline stage will process a different instruction every cycle. First instruction will complete in 5 cycles, then one instruction will complete in every cycle, due to ideal overlap.
    \[
    \text{Number of cycles} = 5 + ((500-1)\times 1) = 504
    \]
- Speedup for ideal pipelining = 2500/504 = 4.96 (or approx. 5)
Pipeline Performance: Example

**Problem:** Consider a non-pipelined processor using the 5-stage datapath with 1 ns clock cycle. Assume that due to clock skew and pipeline registers, pipelining the processor adds 0.2 ns of overhead to the clock speed. How much speedup can we expect to gain from pipelining? Assume a balanced pipeline and ignore the pipeline fill and drain overheads.

**Solution:**

**Without pipelining:** Clock period = 1 ns, CPI = 5

**With pipelining:** Clock period = 1 + 0.2 = 1.2 ns, CPI = 1

Speedup from pipelining = \( \frac{\text{Average time per instruction}_{\text{non-pipelined}}}{\text{Average time per instruction}_{\text{pipelined}}} \)

\[ \frac{1 \text{ ns} \times 5}{1.2 \text{ ns} \times 1} = \frac{5}{1.2} = 4.17 \]

Pipeline Performance (cont.)

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages
- However, this increase would be achieved only if
  - all pipeline stages require the same time to complete, and
  - there is no interruption throughout program execution
- Unfortunately, this is not true
  - there are times when an instruction cannot proceed from one stage to the next in every clock cycle

**Pipeline Performance (cont.)**

- **Clock Cycle**
  - Time
  - IF ID EX MEM WB

- **I_j**
  - Cycle 1: IF ID EX MEM WB
  - Cycle 2: IF ID EX MEM WB
  - Cycle 3: IF ID EX MEM WB
  - Cycle 4: IF ID EX MEM WB
  - Cycle 5: IF ID EX MEM WB
  - Cycle 6: IF ID EX MEM WB
  - Cycle 7: IF ID EX MEM WB
  - Cycle 8: IF ID EX MEM WB
  - Cycle 9: IF ID EX MEM WB

- **I_{j+1}**
  - Cycle 1: IF ID EX MEM WB
  - Cycle 2: IF ID EX MEM WB
  - Cycle 3: IF ID EX MEM WB
  - Cycle 4: IF ID EX MEM WB
  - Cycle 5: IF ID EX MEM WB
  - Cycle 6: IF ID EX MEM WB
  - Cycle 7: IF ID EX MEM WB
  - Cycle 8: IF ID EX MEM WB
  - Cycle 9: IF ID EX MEM WB

- **I_{j+2}**
  - Cycle 1: IF ID EX MEM WB
  - Cycle 2: IF ID EX MEM WB
  - Cycle 3: IF ID EX MEM WB
  - Cycle 4: IF ID EX MEM WB
  - Cycle 5: IF ID EX MEM WB
  - Cycle 6: IF ID EX MEM WB
  - Cycle 7: IF ID EX MEM WB
  - Cycle 8: IF ID EX MEM WB
  - Cycle 9: IF ID EX MEM WB

- Assume that instruction I_{j+1} is *stalled* in the decode stage for two extra cycles
- This will cause I_{j+2} to be stalled in the fetch stage, until I_{j+3} proceeds
- New instructions cannot enter the pipeline until I_{j+2} proceeds past the fetch stage after cycle 5 => execution time increases by two cycles

**Major Hurdle of Pipelining: Hazards**

- Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle
- Three types of pipeline hazards
  - **Structural hazard** – a situation where two (or more) instructions require the use of a given hardware resource at the same time
  - **Data hazard** – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
    - Instruction processing will be delayed until operands become available
  - **Control hazard** – a delay in the availability of an instruction or the memory address needed to fetch the instruction
Major Hurdle of Pipelining: Hazards

- Hazards may require “stalling” the pipeline allowing some instructions to proceed while others are delayed (and no additional instructions fetched) until the conditions that caused the hazard do not exist anymore.
- This is called “clearing the hazard”
- Stalling increases the CPI \( \rightarrow \) reduces the speedup from pipelining

\[
\text{Speedup from Pipelining} = \frac{\text{Number of pipeline stages}}{1 + \text{Pipeline stall cycles per instruction}}
\]

Major Hurdle of Pipelining: Hazards

- Three types of pipeline hazards
  - **Structural hazard** – a situation where two (or more) instructions require the use of a given hardware resource at the same time
  - **Data hazard** – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
  - **Control hazard** – a delay in the availability of an instruction or the memory address needed to fetch the instruction

Structural Hazards

- A situation where two (or more) instructions require the use of a given hardware resource at the same time
- **Example:** In the MIPS 5-stage pipeline, both the “IF” and “MEM” stages require memory access
- In the same cycle
  - A new instruction is fetched from memory in the IF stage
  - A “Load” instruction reads data from memory in the MEM stage
- What happens if the memory has only one read port?

Structural Hazards: An Example
Stalls

- Instructions ahead of the stall proceed to completion
- Stall delays instruction and those following it

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>1</td>
</tr>
<tr>
<td>Instruction + 1</td>
<td>2</td>
</tr>
<tr>
<td>Instruction + 2</td>
<td>3</td>
</tr>
<tr>
<td>Instruction + 3</td>
<td>4</td>
</tr>
<tr>
<td>Instruction + 4</td>
<td>5</td>
</tr>
<tr>
<td>Instruction + 5</td>
<td>6</td>
</tr>
<tr>
<td>Instruction + 6</td>
<td>7</td>
</tr>
</tbody>
</table>

- Cost of Stall

Problem: Consider two pipelined processors. Suppose data references represent 40% of the instructions executed and that the ideal CPI of the pipelined processor, ignoring the structural hazard is 1. Assume that the processor with the hazard has a clock rate that is 1.05 times higher than the hazard free processor and incurs an one cycle stall on structural hazards as in our example. Is the pipeline with or without the structural hazard faster, and by how much?

Solution:
- Without structural hazard: Clock period = C, CPI = 1
- With structural hazard: Clock period = C/1.05, CPI = 1 + (0.4 * 1) = 1.4

\[
\frac{(\text{Execution Time})_{\text{with structural hazard}}}{(\text{Execution Time})_{\text{without structural hazard}}} = \frac{1.05^{1.4}}{1} = \frac{1.4}{1.05} = 1.33
\]

Processor without the structural hazard is 1.33 times faster

How to Mitigate Structural Hazards?

- **Key Idea**: Add more hardware resources

- How to avoid structural hazard on memory access during IF and MEM stages?
  - Separate instruction and data caches

- How to avoid structural hazard on register access during ID and WB stages?
  - Dual ported register file
  - Write early in WB stage
  - Read late in ID stage

- Why no structural hazard on ALU during IF (PC ← PC + 4) and MEM stages?
  - PC + 4 in IF stage uses a dedicated adder, not the ALU

Major Hurdle of Pipelining: Hazards

- Three types of pipeline hazards
  - **Structural hazard** – a situation where two (or more) instructions require the use of a given hardware resource at the same time
  - **Data hazard** – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
  - **Control hazard** – a delay in the availability of an instruction or the memory address needed to fetch the instruction
Data Hazards

• Consider the following two instructions executed in a program sequence:
  Add R2, R3, #100
  Subtract R9, R2, #30
• The destination register (R2) for the first instruction is a source register for the second instruction
• Register R2 carries data from the first instruction to the second instruction => There is a data dependency between these two instructions
• First instruction writes to register R2 in the WB stage
• Second instruction reads register R2 in the ID stage
• If second instruction reads R2 before the first instruction writes R2, the result of second instruction would be incorrect, as it would be based on R2’s old value (read-after-write dependence)
• To obtain the correct result, second instruction needs to wait until the first instruction has written to R2

Types of Data Hazards

• Three general types of data hazards
  – Read After Write (RAW) True dependence
    • Caused by “dependence”. Subsequent instruction has actual need for data produced by earlier instruction
  – Write after Read (WAR) False dependence
    • Called “anti-dependence”. Can’t occur in in-order pipelines (e.g., our simple MIPS pipeline). We’ll see it later in advanced pipelines
  – Write after Write (WAW) False dependence
    • Called “output dependence”. Can’t occur in in-order pipelines (e.g., our simple MIPS pipeline). We’ll see it later in advanced pipelines

Read-after-Write (RAW) Hazard

• Instr_j tries to read operand before Instr_i write it (where j > i)
  Instr_i: add r1, r2, r3
  Instr_j: sub r4, r1, r3
• This hazard results from a true dependence: an actual need for communication from the first instruction to the second
**Write-after-Read (WAR) Hazard**

- Instr\(_j\) tries to write operand before Instr\(_i\) reads it (where \(j > i\))
  
  Instr\(_i\): \texttt{sub r4, r1, r3}  
  Instr\(_j\): \texttt{add r1, r2, r3}  

- This is an “anti-dependence” (not a true dependence)
  
  - Arises from the reuse of register “r1”
  - If Instr\(_j\) writes to r1 before Instr\(_i\) reads r1, then Instr\(_i\) will use the value written by Instr\(_j\) => Incorrect behavior

- WAR hazards cannot happen in MIPS 5-stage pipeline because:
  
  - Instructions are executed in order
  - Register reads happen earlier (stage-2) than register writes (stage-5)

**Write-after-Write (WAW) Hazard**

- Instr\(_j\) writes to operand before Instr\(_i\) writes to it (where \(j > i\))
  
  Instr\(_i\): \texttt{sub r1, r4, r3}  
  Instr\(_j\): \texttt{add r1, r2, r3}  
  Instr\(_k\): \texttt{mul r6, r1, r7}  

- This is an “output dependence” (not a true dependence)
  
  - Arises from the reuse of register r1
  - If Instr\(_j\) writes to r1 before Instr\(_i\) then Instr\(_k\) will use the value written by Instr\(_i\) => violation of program order

- WAW hazards cannot happen in MIPS 5-stage pipeline because:
  
  - Instructions are executed in order
  - Register writes are always in stage-5