Pipelining

CSE 410 - Computer Systems
October 17, 2001

Readings and References

- Reading
  - Sections 6.1 through 6.3, Patterson and Hennesey, Computer Organization & Design

- Other References

Execution Cycle

IF → ID → EX → MEM → WB

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory
5. Write Back

IF and ID Stages

1. Instruction Fetch
   - Get the next instruction from memory
   - Increment Program Counter value by 4
2. Instruction Decode
   - Figure out what the instruction says to do
   - Get values from the named registers
   - Simple instruction format means we know which registers we may need before the instruction is fully decoded

Simple MIPS Instruction Formats

- op code source 1 source 2 dest shamt function
  - 6 bits 5 bits 5 bits 5 bits 4 bits

- op code base reg src/dest offset or immediate value
  - 6 bits 5 bits 5 bits 16 bits

- op code word offset
  - 6 bits 24 bits

EX, MEM, and WB stages

3. Execute
   - On a memory reference, add up base and offset
   - On an arithmetic instruction, do the math
4. Memory Access
   - If load or store, access memory
   - If branch, replace PC with destination address
   - Otherwise do nothing
5. Write back
   - Place the results in the appropriate register
Example: add $s0, $s1, $s2

- **IF** get instruction at PC from memory
  - | op code | source 1 | source 2 | dest | shamt | function |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>1000</td>
<td>00000</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- **ID** determine what instruction is and read registers
  - 000000 with 100000 is the add instruction
  - get contents of $s1 and $s2 (eg: $s1=7, $s2=12)
- **EX** add 7 and 12 = 19
- **MEM** do nothing for this instruction
- **WB** store 19 in register $s0

Example: lw $t2, 16($s0)

- **IF** get instruction at PC from memory
  - | op code | base reg | src/dest | offset or immediate value |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>010111</td>
<td>10000</td>
<td>01000</td>
<td>0000000000010000</td>
</tr>
</tbody>
</table>

- **ID** determine what 010111 is
  - 010111 is lw
  - get contents of $s0 and $t2 (we don’t know that we don’t care about $t2) $s0=0x200D1C00, $t2=77763
- **EX** add 16 to 0x200D1C00 = 0x200D1C10
- **MEM** load the word stored at 0x200D1C10
- **WB** store loaded value in $t2

Latency & Throughput

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 1</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 2</th>
</tr>
</thead>
</table>

- **Latency**—the time it takes for an individual instruction to execute
  - What’s the latency for this implementation?
- **Throughput**—the number of instructions that execute per unit time
  - What’s the throughput of this implementation?

A case for pipelining

- If non-overlapped, the functional units are underutilized because each unit is used only once every five cycles
- If Instruction Set Architecture is carefully designed, organization of the functional units can be arranged so that they execute in parallel
- **Pipelining** overlaps the stages of execution so every stage has something to do each cycle

Pipelined Latency & Throughput

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 1</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 2</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 3</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 4</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>inst 5</th>
</tr>
</thead>
</table>

- What’s the throughput of this implementation?
- What’s the latency of this implementation?

Pipelined Analysis

- A pipeline with N stages could improve throughput by N times, but
  - each stage must take the same amount of time
  - each stage must always have work to do
  - there may be some overhead to implement
- Also, latency for each instruction may go up
  - Within some limits, we don’t care
Throughput is good!

- Increasing number of instructions
- Overlapped vs sequential

MIPS ISA: Born to Pipeline

- Instructions all one length
  - Simplifies Instruction Fetch stage
- Regular format
  - Simplifies Instruction Decode
- Few memory operands, only registers
  - Only lw and sw instructions access memory
- Aligned memory operands
  - Only one memory access per operand

Memory accesses

- Efficient pipeline requires each stage to take about the same amount of time
- CPU is much faster than memory hardware
- Cache is provided on chip
  - I-cache holds instructions
  - D-cache holds data
  - Critical feature for successful RISC pipeline
  - More about caches next week

The Hazards of Parallel Activity

- Any time you get several things going at once, you run the risk of interactions and dependencies
  - Juggling doesn’t take kindly to irregular events
- Unwinding activities after they have started can be very costly in terms of performance
  - Drop everything on the floor and start over

Design for Speed

- Most of what we talk about next relates to the CPU hardware itself
  - Problems keeping a pipeline full
  - Solutions that are used in the MIPS design
- Some programmer visible effects remain
  - Many are hidden by the assembler or compiler
  - The code that you write tells what you want done, but the tools rearrange it for speed

Pipeline Hazards

- Structural hazards
  - Instructions in different stages need the same resource, e.g., memory
- Data hazards
  - Data not available to perform next operation
- Control hazards
  - Data not available to make branch decision
Structural Hazards

• Concurrent instructions want same resource
  – \texttt{lw} instruction in stage four (memory access)
  – \texttt{add} instruction in stage one (instruction fetch)
  – Both of these actions require access to memory; they would collide if not designed for
• Add more hardware to eliminate problem
  – separate instruction and data caches
• Or stall (cheaper & easier), not usually done

Data Hazards

• When an instruction depends on the results of a previous instruction still in the pipeline
• This is a data dependency

\[
\begin{align*}
\text{add} & \ $s0,\ $s1,\ $s2 \\
\text{add} & \ $s4,\ $s3,\ $s0
\end{align*}
\]

Stall for register data dependency

• Stall the pipeline until the result is available
  – this would create a 3-cycle pipeline bubble

\[
\begin{align*}
\text{add} & \ s0,s1,s2 \\
\text{add} & \ s4,s3,s0
\end{align*}
\]

Solution: Forwarding

• The value of $s0$ is known internally after cycle 3
  (after the first instruction’s EX stage)
• The value of $s0$ isn’t needed until cycle 4 (before
  the second instruction’s EX stage)
• If we \texttt{forward} the result there isn’t a stall

\[
\begin{align*}
\text{add} & \ s0,s1,s2 \\
\text{add} & \ s4,s3,s0
\end{align*}
\]

Read & Write in same Cycle

• Write the register in the first part of the clock cycle
• Read it in the second part of the clock cycle
• A 2-cycle stall is still required

\[
\begin{align*}
\text{add} & \ s0,s1,s2 \\
\text{add} & \ s4,s3,s0
\end{align*}
\]

Another data hazard

• What if the first instruction is \texttt{lw}?
• $s0$ isn’t known until after the MEM stage
  – We can’t forward back into the past
• Either \texttt{stall} or \texttt{reorder} instructions

\[
\begin{align*}
\text{lw} & \ a0,0(s2) \\
\text{add} & \ s4,s3,s0
\end{align*}
\]

\[
\text{NO!}\]
Stall for \texttt{lw} hazard

- We can stall for one cycle, but we hate to stall

\begin{center}
\begin{tabular}{c|c|c|c|c|c}
\hline
\texttt{lw} & \texttt{add} & & & & \\
\hline
\texttt{IF} & \texttt{ID} & \texttt{EX} & \texttt{MEM} & \texttt{WB} & \\
\hline
\end{tabular}
\end{center}

Instruction Reorder for \texttt{lw} hazard

- Try to execute an unrelated instruction between the two instructions

\begin{center}
\begin{tabular}{c|c|c|c|c|c}
\hline
\texttt{lw} & \texttt{sub} & \texttt{add} & & & \\
\hline
\texttt{IF} & \texttt{ID} & \texttt{EX} & \texttt{MEM} & \texttt{WB} & \\
\hline
\texttt{t4,t2,t3} & \texttt{t4,t2,t3} & \texttt{s4,s3,s0} & & & \\
\hline
\end{tabular}
\end{center}

Reordering Instructions

- Reordering instructions is a common technique for avoiding pipeline stalls
- Static reordering
  - programmer, compiler and assembler do this
- Dynamic reordering
  - modern processors can see several instructions
  - they execute any that have no dependency
  - this is known as \textit{out-of-order execution} and is complicated to implement