Orange Coast College
Business Division
Computer Science Department

CS 116- Computer Architecture

Pipelining
Topics

• Explaining pipelining through example
• CPU pipelining
• MIPS instructions & pipelining
• Pipeline hazards
• Recent trends in performance
Introduction

- Still in CPU’s Datapath
What is pipelining?

• Implementation technique in which multiple instructions are overlapped in execution

• Real-life pipelining examples?
  – Laundry
  – Factory production lines
  – Traffic??

• Key to making processors fast
Pipelining Example: Laundry

• You have 4 loads of clothes to wash:
• Steps (stages) required: 
  – Wash
  – Dry
  – Fold
  – Store clothes into drawers
    • Maybe let your roommate help here
• Each stage needs 30 minutes
• We can’t start the next step until the previous step is finished
Pipelining Example: Laundry

- There are 2 approaches to do this job:
  - Sequential (non-pipelined):
    - Wait until the first load is put away in order to start the next load
  - Pipelined (ASAP):
    - As soon as the washer is empty, start putting the next load, while the first load is put into dryer
Pipelining Example: Laundry

- Sequential
  - Needs 8 hours for 4 loads
Pipelining Example: Laundry

- Pipelined Laundry:
  - Start work ASAP
  - Needs only 3.5 hours for 4 loads!
Pipelined Laundry Observations:

- At some point, all stages of washing will be operating concurrently.
- Pipelining doesn’t reduce number of stages:
  - Doesn’t help latency of single task.
  - Does help throughput of entire workload.
- As long as we have separate resources, we can pipeline the tasks.
- Multiple tasks operating simultaneously use different resources.
Pipelined Laundry Observations:

- Speedup due to pipelining depends on the number of stages in the pipeline
- Pipeline rate limited by slowest pipeline stage
  - If dryer needs 45 min, time for all stages has to be 45 min to accommodate it
  - Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- If one load depends on another, we will have to wait (Delay/Stall for Dependencies)
Pipelining Example: Problems

- If we have washer-dryer combination instead of separate washer & dryer, we can’t make the two steps in parallel
- If your roommate is busy doing something else, you will have to add a stage of going to the drawer and putting your stuff yourself
  - One resource for folding
  - Another resource for put away

=> Structural Hazard
Pipelining Example: Problems

• If you are cleaning the uniform of a football team, you need to determine whether the detergent & water temperature settings are ok or should be changed, depending on how dirty are the uniforms
  – Stall ? Slow
    • Wait until you get the dry uniforms to see if you need to change setup.
    • Repeat until you have the right formula
  – Continue with the next load & check formula later?

=> Control hazards
Pipelining Example: Problems

• In the folding process, if you discover that the load of socks you have need the mate of each of the socks that is still in the washer, you will have to wait until they dry in order to be able to fold

=> Data hazards
CPU Pipelining

• Can we pipeline instruction execution?
• For the following instructions, which resources do you need for each of these steps?
  – store/ load word
  – add/ subtract/ and/ or/ slt
  – branch if equal
CPU Pipelining

- Review: 5 stages of a MIPS instruction
  - Fetch instruction from instruction memory
  - Read registers while decoding instruction
  - Execute operation or calculate address, depending on the instruction type
  - Access an operand from data memory
  - Write result into a register

- We can reduce the cycles to fit the stages.

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>
CPU Pipelining

• Review: Datapath resources
CPU Pipelining

- Example: Resources for Load Instruction
  - Fetch instruction from instruction memory (Ifetch)
    - Instruction memory (IM)
  - Read registers while decoding instruction (Reg/Dec)
    - Register file & decoder (Reg)
  - Execute operation or calculate address
    - ALU
  - Access an operand from data memory (Mem)
    - Data memory (DM)
  - Write result into a register (Wr)
    - Register file (Reg)
CPU Pipelining

• Note that accessing source & destination registers is performed in two different parts of the cycle

• We need to decide upon which part of the cycle should reading and writing to the register file take place.
CPU Pipelining: Example

• Assumptions:
  – Only consider the following instructions:
    
    \[ lw, sw, add, sub, and, or, slt, beq \]
  – Operation times for instruction classes are:
    • Memory access \( 2 \) ns
    • ALU operation \( 2 \) ns
    • Register file read or write \( 1 \) ns
  – Use a single- cycle model
  – Clock cycle must accommodate the slowest instruction (2 ns)
  – Both pipelined & non-pipelined approaches use the same HW components
CPU Pipelining: Example

- Single-Cycle, non-pipelined execution
- Total time for 3 instructions: 24 ns
CPU Pipelining: Example

- Single-cycle, pipelined execution
  - Improve performance by increasing throughput
  - Total time for 3 instructions = 14 ns
  - Each instruction adds 2 ns to total execution time
  - Stage time limited by slowest resource (2 ns)
CPU Pipelining Example:

- **Instructions total times:**

<table>
<thead>
<tr>
<th>InstrClass</th>
<th>IstrFetch</th>
<th>RegRead</th>
<th>ALUOp</th>
<th>DataAccess</th>
<th>RegWrite</th>
<th>TotTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td>1 ns</td>
<td>8 ns</td>
</tr>
<tr>
<td>sw</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>add, sub, and, or, slt</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td>1 ns</td>
<td>6 ns</td>
</tr>
<tr>
<td>beq</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
</tbody>
</table>

- **Assumption:**
  - No delay in the following processor units:
    - MUX, Control unit, PC access, Sign-extend units

- **Slowest resources:**
  - Instruction memory
  - ALU
  - Data memory
CPU Pipelining Example:

- Theoretically:
  - Speedup should be equal to number of stages

- Practically:
  - Stages are imperfectly balanced
  - Pipelining needs overhead
  - Speedup less than number of stages

- If we have 3 consecutive instructions
  - Non-pipelined needs $8 \times 3 = 24$ ns
  - Pipelined needs $14$ ns

  $\Rightarrow$ Speedup $= 24 / 14 = 1.7$
CPU Pipelining Example:

• If we have 1003 consecutive instructions
  – Add more time for 1000 instruction (i.e. 1003 instruction) on the previous example
  • Non-pipelined total time = 1000 x 8 + 24 = 8024 ns
  • Pipelined total time = 1000 x 2 + 14 = 2014 ns

=> Speedup ~ 3.98 ~(8 ns / 2 ns)
    ~ near perfect speedup

=> Performance increases for larger number of instructions
  – Higher throughput
Pipelining MIPS Instruction Set

• MIPS was designed with pipelining in mind
  => Pipelining is easy in MIPS:
  1) All instruction are the same length
  2) Limited instruction format
  3) Memory operands appear only in lw & sw
  4) Operands must be aligned in memory
Pipelining MIPS Instruction Set

1) All MIPS instruction are the same length
   - Fetch instruction in 1st pipeline stage
   - Decode instructions in 2nd stage
   - If instruction length varies (e.g. 80x86), pipelining will be more challenging
Pipelining MIPS Instruction Set

2) MIPS has limited instruction format

- Source register in the same place for each instruction
  - Symmetry
- 2nd stage can begin reading at the same time as decoding
- If instruction format wasn’t symmetric, stage 2 should be split into 2 distinct stages

=> Total stages = 6 (instead of 5)
Pipelining MIPS Instruction Set

3) Memory operands appear only in lw & sw
   - We can use the execute stage to calculate memory address
   - Access memory in the next stage
   - If we needed to operate on operands in memory (e.g. 80x86), stages 3 & 4 would expand to
     • Address calculation
     • Memory access
     • Execute
Pipelining MIPS Instruction Set

4) Operands must be aligned in memory
   - Transfer of more than one data operand can be done
     in a single stage with no conflicts
   - Need not worry about single data transfer instruction
     requiring 2 data memory accesses
   - Requested data can be transferred between the
     CPU & memory in a single pipeline stage
Pipelined Datapath

• Basic Idea:
  – Resources for Different Stages should be Independent
  – Add registers & other HW to split datapath into stages
Pipelined Datapath

• Example: add Instruction

• Terminology:
  – Shading indicates that the element is used
    • Shading on the right means reading
    • Shading on the left means writing
  – For register file
    • Hashed means not used at the corresponding half of the cycle

\[
\text{add } s_0, t_0, t_1
\]
Pipelined Datapath

- Graphically Representing Pipelines Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?

```
lw $10, 20($1)
sub $11, $2, $3
```

Program execution order (in instructions)

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6
Pipelined Datapath

• Another way to represent pipelining

<table>
<thead>
<tr>
<th>IFetch</th>
<th>Dcd</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>Dcd</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>IFetch</td>
<td>Dcd</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>IFetch</td>
<td>Dcd</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>IFetch</td>
<td>Dcd</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>IFetch</td>
<td>Dcd</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>