CS3350B Computer Architecture
Instructional Level Parallelism Fundamentals

Ning Xie

http://www.csd.uwo.ca/courses/CS3350b
Department of Computer Science
University of Western Ontario, Canada

Winter, 2016
Analogy: Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- **Sequential** laundry takes 8 hours for 4 loads
Pipelined Laundry

- **Pipelined** laundry takes 3.5 hours for 4 loads!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to “fill” pipeline and time to “drain” it reduce speedup: 2.3x vs. 4x in this example
Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
Recap: MIPS Three Instruction Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Field</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>D</th>
<th>SHAMT</th>
<th>FUNCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>op</td>
<td>6 bits</td>
<td>rs</td>
<td>5 bits</td>
<td>rt</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>5 bits</td>
<td>rt</td>
<td>5 bits</td>
<td>rd</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>5 bits</td>
<td>rd</td>
<td>5 bits</td>
<td>shamt</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td>funct</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td>I-format</td>
<td>op</td>
<td>6 bits</td>
<td>rs</td>
<td>5 bits</td>
<td>rt</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>5 bits</td>
<td>rt</td>
<td>5 bits</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>5 bits</td>
<td>immediate</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-format</td>
<td>op</td>
<td>6 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>target address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>target address</td>
<td>6 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bits</td>
</tr>
</tbody>
</table>

Examples:
- R-format: add, sub, jr
- I-format: ori, lw, sw, beq, bne
- J-format: j, jal
Recap: Five Stages in Executing MIPS

1. Instruction Fetch (IFetch)
   - Fetch an instruction; increment PC

2. Instruction Decode (Dec)
   - Decode instruction; read registers; sign extend offset

3. ALU (Arithmetic-Logic Unit) (Exec)
   - Execute R-format operations; calculate memory address; branch comparison; branch and jump completion

4. Memory Access (Mem)
   - Read data from memory for load or write data to memory for store

5. Register Write (WB)
   - Write data back to register
Graphical Representation

Graphical representation:

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write

Short name:
- IFetch
- Dec
- Exec
- Mem
- WB
» All stages of an instruction completed within one long clock cycle
  - Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write
  - 200ps for other events

- Clock rate of the single cycle datapath is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td>100ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
  Want increased clock rate to mean faster programs
Multiple-cycle CPU Clocking

- Only one stage of instruction per clock cycle
  - Clock is made as long as the slowest stage

- **Advantages** over single cycle execution:
  Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped)
How Can We Make It Faster?

- Split the multiple instruction cycle into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining - (all?) modern processors are pipelined for performance
  - Remember the performance equation:
    \[
    \text{CPU time} = \text{CPI} \times \text{CC} \times \text{IC}
    \]
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing - stay tuned
Start the next instruction before the current one has completed
- improves throughput - total amount of work done in a given time
- instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is not reduced

- **clock cycle** (pipeline stage time) is limited by the **slowest** stage
- for some instructions, some stages are **wasted** cycles
Why Pipeline? For Performance!

- Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages.
  - A five-stage pipeline is nearly five times faster.

![Diagram of pipeline stages](image)

- Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

- Time to fill the pipeline
Pipeline Performance

- Assume time for actions are
  - 100ps for register read or write
  - 200ps for other events

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td>100ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Pipeline Performance

Single-cycle (Tc= 800ps)

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 200($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

200 ps

Pipelined (Tc= 200ps)

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 200($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

200 ps
Pipeline Speedup

- If all stages are **balanced** (i.e. all take the same time) and there are no dependencies between the instructions,
  - CPI = 1 (each instruction takes 5 cycles, but 1 completes each cycle)
  - Ideal speedup is:

\[
\text{Number of stages} = \frac{\text{Time between instructions nonpipelined}}{\text{Time between instructions pipelined}}
\]

- **Speedup** due to increased **throughput**;
  **Latency** (time for each instruction) does not decrease

- If **not balanced**, speedup is less
  - Pipelining the three `lw`, speedup: \(2400ps/1400ps = 1.7\)
  - Add 1000,000 more instructions, the speedup:
    \[\frac{(10^6 \times 200 + 1400)}{(10^6 \times 800 + 2400)} \sim 800/200 = 4\]

Ning Xie (http://www.csd.uwo.ca/courses/CS3350B Computer Architecture Instructional Level F Winter, 2016)
MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our MIPS datapath?
  - Add State registers between each pipeline stage to isolate them
Pipelining the MIPS ISA

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
    (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address at 3rd stage, access memory at 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle
Other Sample Pipeline Alternatives

- ARM7

```
IM -- Reg -- EX
PC update  decode  ALU op
IM access  reg access DM access
           shift/rotate
           commit result
           (write back)
```

- Intel XScale

```
IM1 -- IM2 -- Reg -- SHFT -- DM1 -- Reg -- DM2
PC update  decode  ALU op  DM write
BTB access  reg 1 access reg write
start IM access  ALU op
IM access  shift/rotate
           reg 2 access
           start DM access
           exception
```
Yes: **Pipeline Hazards**

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
  - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
  - branch instructions

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards
Takeaway

- All modern day processors use pipelining
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and a faster CC
  - Recall CPU time = CPI * CC * IC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages make for inefficiencies
  - The time to “fill” pipeline and time to “drain” it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI more than the ideal of 1)
  - Compiler can arrange code to avoid hazards and stalls: Requires knowledge of the pipeline structure