Pipelining
-and-
Review for Final Exam

March 15, 2013
Outline

Pipelined execution
   What is pipelining?
   MIPS pipeline
   Hazards

Review for final
   Overview of final
   More practice assembling
Pipelining

**Goal:** execute programs faster!

**Basic idea**
- separate processor into stages
- overlap the execution of consecutive instructions

**Laundry room analogy**
- it takes 60 minutes for one person to do their laundry
  - 20 minutes each for washer, dryer, and folding
- at 1pm, 5 people all want to wash their laundry ASAP
Laundry room analogy

One person in the laundry room at a time (not pipelined)

<table>
<thead>
<tr>
<th>1pm</th>
<th>2pm</th>
<th>3pm</th>
<th>4pm</th>
<th>5pm</th>
</tr>
</thead>
<tbody>
<tr>
<td>W D F</td>
<td>W D F</td>
<td>W D F</td>
<td>W D F</td>
<td>W D F</td>
</tr>
</tbody>
</table>

- 5 people = 5h
- 1 person / hour

Separate laundry room into three stations (pipelined)

<table>
<thead>
<tr>
<th>1pm</th>
<th>2pm</th>
<th>3pm</th>
</tr>
</thead>
<tbody>
<tr>
<td>W D F</td>
<td>W D F</td>
<td>W D F</td>
</tr>
</tbody>
</table>

- 5 people = 2h20m
- \( \lim_{n \to \infty} 1 \text{ person} / 20m \)
MIPS is designed for pipelining

Originally: **Microprocessor without Interlocked Pipeline Stages**

**MIPS 5-stage pipeline**

[Diagram of the MIPS 5-stage pipeline]

- **Instruction Fetch (IF)**: Fetch the next instruction from memory.
- **Instruction Register Decode (ID)**: Decode the instruction and fetch required registers.
- **Execute Address Calculation (EX)**: Perform address calculation for memory access.
- **Memory Access (MEM)**: Access memory based on the calculated address.
- **Write Back (WB)**: Write back the results to registers.

**Key Components**:
- **PC**: Program Counter
- **IR**: Instruction Register
- **Register File**: Stores registers for calculations
- **Sign Extend**: Handles sign extension of immediate values
- **ALU**: Arithmetic Logic Unit for calculations
- **Memory**: Stores data and instructions
- **Branch**: Conditional branch based on calculated conditions
- **Zero?**: Branch taken if zero
- **Next PC**: Next Program Counter

**Flow**:
- Instruction fetch leads to decoder and register fetch.
- Calculated address leads to memory access.
- Results are written back to registers.
**Instruction pipelining**

**Pipeline stages**
1. IF – Instruction Fetch
2. ID – Instruction Decode
3. EX – EXecute
4. ME – MEmory access
5. WB – Write Back

**Example**

| lw $s0, 20($sp) | IF ID EX ME WB |
| lw $s1, 24($sp) | IF ID EX ME WB |
| lw $s2, 28($sp) | IF ID EX ME WB |
| lw $s3, 32($sp) | IF ID EX ME WB |
Hazards

**Hazard**: a dependency that breaks pipelining

**Two kinds of hazards**

- **Data hazard** – need a value that hasn’t been computed yet
- **Control hazard** – don’t know which instruction comes next

**Example: Data hazard**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1, $t2, $t3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td>$t1 set here</td>
</tr>
<tr>
<td>addi $t4, $t1, 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

$t1$ read here
Resolving hazards

### Example: Data hazard

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>add $t1, $t2, $t3</th>
<th>addi $t4, $t1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>IF ID EX ME WB</td>
<td>IF ID EX ME WB</td>
</tr>
</tbody>
</table>

$t1$ set here

$t1$ read here

### Solution: Processor inserts delays

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>add $t1, $t2, $t3</th>
<th>addi $t4, $t1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9</td>
<td>IF ID EX ME WB</td>
<td>IF XX XX XX ID EX ME WB</td>
</tr>
</tbody>
</table>
Avoiding hazards

Delays negate the benefit of pipelining!

- would rather avoid data hazards then resolve them

Can reorder instructions to avoid data hazards

| lw  | $ra, 16($sp)                           | add   | $t1, $t2, $t3  |
| lw  | $t2, 20($sp)                           | lw    | $ra, 16($sp)   |
| lw  | $t3, 24($sp)                           | lw    | $t2, 20($sp)   |
| add | $t1, $t2, $t3                           | lw    | $t3, 24($sp)   |
| addi| $t4, $t1, 1                             | addi  | $t4, $t1, 1    |
Outline

Pipelined execution
  What is pipelining?
  MIPS pipeline
  Hazards

Review for final
  Overview of final
  More practice assembling
I will provide . . .

- ASCII character encoding table
- interface of any relevant system calls
- a diagram of the stack frame layout
  - same as in slides, but black & white
- instruction format diagrams
- table of op/funct codes and syntax of any instructions you will need to assemble
You should bring . . .

- a pencil and eraser
- one page of notes (optional)
- a calculator (optional)
Focusing your study

Every problem on the final will be either . . .

• similar to a problem on Midterm 1
• similar to a problem on Midterm 2
• related to material covered since Midterm 2
From Midterm 1

- computer architecture vocabulary
- data representation
  - converting between bases
  - representing integers (two’s complement)
  - null-terminated ASCII strings
- binary arithmetic (addition, multiplication)
- implementing math expressions in assembly
From Midterm 2

- procedure call vocabulary
- array addressing
  - integer arrays
  - strings (character arrays)
- implementing control structures in assembly
  - if-then, if-then-else, do-while, while, for
- procedure calling conventions
  - calling and returning from procedures
  - argument passing and return values
  - saving registers and managing the stack
Since Midterm 2

- subroutine linkage (implements the calling conventions)
- function pointers
- assembling to machine code
  - instruction formats
  - assembling R-type instructions
  - assembling I-type instructions (both math and branches)
  - (I won’t have you assemble a J-type instruction)
- (nothing about pipelines either)
Exercise (basically how it will look on the final)

### Relevant instructions

<table>
<thead>
<tr>
<th>Instruction syntax</th>
<th># op/fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $rd, $rs, $rt</td>
<td>32</td>
</tr>
<tr>
<td>addi $rt, $rs, imm</td>
<td>8</td>
</tr>
<tr>
<td>beq $rs, $rt, offset</td>
<td>4</td>
</tr>
<tr>
<td>bne $rs, $rt, offset</td>
<td>5</td>
</tr>
</tbody>
</table>

### Assemble the following program:

```
beq $t0, $t1, label
addi $t0, $t0, 10
label: add $t2, $t0, $t1
beq $t2, $t3, label
```

### Name and Number

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4–7</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8–15</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>16–23</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24–25</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
</tr>
</tbody>
</table>