Review Problem 22

- Develop a single-cycle CPU that can do LDUR and STUR (only). Make it as simple as possible.
Advanced: Exceptions

Exception = unusual event in processor
  Arithmetic overflow, divide by zero, ...
  Call an undefined instruction
  Hardware failure
  I/O device request (called an "interrupt")

Approaches
  Make software test for exceptional events when they may occur ("polling")
  Have hardware detect these events & react:
    Save state (Exception Program Counter, protect the GPRs, note cause)
    Call Operating System
      If (undef_instr) PC = C0000000
      If (overflow) PC = C0000020
      If (I/O) PC = C0000040
      ...

System
Exception
Handler

return from exception
Performance of Single-Cycle Machine

CPI? \( \frac{1}{n} \) by definition

### ADD, SUB

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr. Memory</th>
<th>mux</th>
<th>Reg Read</th>
<th>mux</th>
<th>ALU</th>
<th>mux</th>
<th>Reg Setup</th>
</tr>
</thead>
</table>

### LDUR

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr. Memory</th>
<th>Reg Read</th>
<th>ALU</th>
<th>Data Memory</th>
<th>mux</th>
<th>Reg Setup</th>
</tr>
</thead>
</table>

### STUR

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr. Memory</th>
<th>Reg Read</th>
<th>ALU</th>
<th>Data Memory</th>
</tr>
</thead>
</table>

### CBZ

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr. Memory</th>
<th>mux</th>
<th>Reg Read</th>
<th>mux</th>
<th>ALU</th>
<th>mux</th>
<th>PC setup</th>
</tr>
</thead>
</table>

### B

<table>
<thead>
<tr>
<th>PC</th>
<th>Instr. Memory</th>
<th>Adder</th>
<th>mux</th>
<th>PC setup</th>
</tr>
</thead>
</table>

Clock Period
Reducing Cycle Time

Cut combinational dependency graph and insert register / latch
Do same work in two fast cycles, rather than one slow one
Pipelined Processor Overview

Divide datapath into multiple stages

IF
Instruction Fetch

RF
Register Fetch

EX
Execute

MEM
Data Memory

WB
Writeback

Instr. Memory

Register File

Data Memory

Register File

PC

Slow: Memory, RF, ALU, Adder, Control Logic
Pipelining

Readings: 4.5-4.8

Example: Doing the laundry

Ann, Brian, Cathy, & Dave

each have one load of clothes to wash, dry, and fold

Washer takes 30 minutes

Dryer takes 40 minutes

"Folder" takes 20 minutes
Sequential Laundry

Sequential laundry takes 6 hours for 4 loads
If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

6 PM  7  8  9  10  11  Midnight

Time

30  40  40  40  40  20

Task Order

A
B
C
D
evry 40

Pipelined laundry takes 3.5 hours for 4 loads
Pipelining Lessons

Pipelining doesn’t help latency of single task, it helps throughput of entire workload

Pipeline rate limited by slowest pipeline stage

Multiple tasks operating simultaneously using different resources

Potential speedup = Number pipe stages

Unbalanced lengths of pipe stages reduces speedup

Time to “fill” pipeline and time to “drain” it reduces speedup

Stall for Dependences
Pipelined Execution

Program Flow

Now we just have to make it work

How does an instr get done?

What is happening in this machine at right now?

(What does a HW unit do?)
Review Problem 28

- Given what we know about pipelining, assume in a widget factory it takes 40 minutes to make 1 widget. If we pipeline the process into S stages, how long will it take to make N widgets?

\[
\text{Cycle Time} = \frac{40}{S}
\]

\[
\text{Exec} = \left( \frac{N + (S-1)}{S} \right) \times \frac{40}{S}
\]

\[
\text{Start} \quad \text{Drain} \quad \text{Cycle}
\]

Note: assume it is an ideal pipeline
- Bank stages equally.
- Adding stages adds delay.
Single Cycle vs. Pipeline

Single Cycle Implementation:

Load

Store

Waste

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 | Cycle 10

Pipeline Implementation:

Load  Ifetch  Reg  Exec  Mem  Wr

Store  Ifetch  Reg  Exec  Mem  Wr

R-type  Ifetch  Reg  Exec  Mem  Wr
Why Pipeline?

Suppose we execute 100 instructions

Single Cycle Machine

\[45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4,500 \text{ ns}\]

Ideal pipelined machine

\[10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1,040 \text{ ns}\]
CPI for Pipelined Processors

Ideal pipelined machine
10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = ____ ns

CPI in pipelined processor is “issue rate”. Ignore fill/drain, ignore latency.

Example: A processor wastes 2 cycles after every branch, and 1 after every load, during which it cannot issue a new instruction. If a program has 10% branches and 30% loads, what is the CPI on this program?

\[
\begin{align*}
\text{Branch} & \quad 3 \times 10\% = 0.3 \\
\text{Load} & \quad 2 \times 30\% = 0.6 \\
\text{Other} & \quad 1 \times 60\% = 0.6 \\
\text{CPI} & = 1.5
\end{align*}
\]
Pipelined Datapath

Divide datapath into multiple pipeline stages
Piped Control

The Main Control generates the control signals during Reg/Dec
Control signals for Exec (ALUOp, ALUSrc, ...) are used 1 cycle later
Control signals for Mem (MemWE, Mem2Reg, ...) are used 2 cycles later
Control signals for Wr (RegWE, ...) are used 3 cycles later
Can pipelining get us into trouble?

Yes: Pipeline Hazards

_structural hazards_: attempt to use the same resource two different ways at the same time

E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)

_data hazards_: attempt to use item before it is ready

E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer

_instruction depends on result of prior instruction still in the pipeline_

_control hazards_: attempt to make decision before condition evaluated

E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in branch instructions

Can always resolve hazards by waiting

pipeline control must detect the hazard take action (or delay action) to resolve hazards
Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are:
- Instruction Memory for the Ifetch stage
- Register File’s Read ports (bus A and bus B) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File’s Write port (bus W) for the Wr stage
The Four Stages of R-type

Ifetch: Fetch the instruction from the Instruction Memory
Reg/Dec: Register Fetch and Instruction Decode
Exec: ALU operates on the two register operands
Wr: Write the ALU output back to the register file
Structural Hazard

Interaction between R-type and loads causes structural hazard on writeback
Important Observation

Each functional unit can only be used once per instruction.
Each functional unit must be used at the same stage for all instructions:
Load uses Register File's Write Port during its 5th stage

<table>
<thead>
<tr>
<th>Load</th>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

R-type uses Register File's Write Port during its 4th stage

<table>
<thead>
<tr>
<th>R-type</th>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Wr</th>
</tr>
</thead>
</table>

Solution: Delay R-type's register write by one cycle:
Now R-type instructions also use Reg File's write port at Stage 5.
Mem stage is a NOOP stage: nothing is being done.

<table>
<thead>
<tr>
<th>R-type</th>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>
No structural hazards.

Pipelining the R-type Instruction

Clock

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Four Stages of Store

Ifetch: Fetch the instruction from the Instruction Memory
Reg/Dec: Register Fetch and Instruction Decode
Exec: Calculate the memory address
Mem: Write the data into the Data Memory
Wr: NOOP

Compatible with Load & R-type instructions
The Stages of Conditional Branch

Ifetch: Fetch the instruction from the Instruction Memory
Reg/Dec: Register Fetch and Instruction Decode, compute branch target
Exec: Test condition & update the PC
Mem: NOOP
Wr: NOOP
Control Hazard

Branch updates the PC at the end of the Exec stage.

<table>
<thead>
<tr>
<th>Clock</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBZ</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC+4/PC+5E</td>
<td>load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type | Ifetch | Reg/Dec | Exec | Mem | Wr
To do a branch

1. Know it is a branch

2. Compute the condition

3. Compute the targets
   PC + 4
   PC + SE (If arm)
Accelerate Branches

When can we compute branch target address? \( \text{PC} + 4 \quad \text{PC} + 5E(I_{\text{imm}}) \leq 2 \)

When can we compute the CBZ condition? \( \text{zero flag} \)
Control Hazard 2

Branch updates the PC at the end of the Reg/Dec stage.

Clock

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
</table>

R-type

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

R-type

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

R-type

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

CBZ

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

load

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

R-type

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

R-type

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 |

Beq

<table>
<thead>
<tr>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

Finish the branch