Review Problem 31

- What registers are being read and written in the 5th cycle of a pipelined CPU running this code?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Ifetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD X1, X2, X3</td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td></td>
</tr>
<tr>
<td>ORR X4, X5, X6</td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td></td>
</tr>
<tr>
<td>SUB X7, X8, X9</td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td></td>
</tr>
<tr>
<td>EOR X10, X11, X12</td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td></td>
</tr>
<tr>
<td>AND X13, X14, X15</td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td></td>
</tr>
</tbody>
</table>

- Writing X1
- Reading X11, X12
Solution #3: Branch Delay Slot

Redefine branches: Instruction directly after branch always executed

Instruction after branch is the delay slot

Compiler/assembler fills the delay slot

```
ADD X1, X0, X4
CBZ X2, FOO
ADD X1, X0, X4
CBZ X1, FOO

ADD X1, X0, X4
SUB X2, X0, X3
ADD X1, X0, X4
SUB X2, X0, X3

ADD X1, X0, X4
CBZ X1, FOO
ADD X1, X3, X3
ADD X31, X31, X31

... FOO:
ADD X1, X2, X0

```

No wasted cycles

Assume 50% branches taken/
50% not taken
waste 1/2 cycle

No wasted cycles

Waste 1 cycle
Data Hazards

Consider the following code:

- ADD \texttt{X0}, X1, X2
- SUB X3, \texttt{X0}, X4
- AND X5, \texttt{X0}, X6
- ORR X7, \texttt{X0}, X8
- EOR X9, \texttt{X0}, X10

Clock

- ADD: Ifetch, Reg/Dec, Exec, Mem, Wr
- SUB: Ifetch, Reg/Dec, Exec, Mem, Wr
- AND: Ifetch, Reg/Dec, Exec, Mem, Wr
- ORR: Ifetch, Reg/Dec, Exec, Mem, Wr
- EOR: Ifetch, Reg/Dec, Exec, Mem, Wr
Design Register File Carefully

What if reads see value after write during the same cycle?

ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10

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>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

Lab 4: to simulate this, invert the clock to the next file.

Valid instructions: ORR, EOR
Forwarding

Add logic to pass last two values from ALU output to ALU input(s) as needed

Forward the ALU output to later instructions

ADD X0, X1, X2
SUB X3, X0, X4
AND X5, X0, X6
ORR X7, X0, X8
EOR X9, X0, X10

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>ADD</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>SUB</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>AND</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>ORR</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>EOR</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>
Forwarding (cont.)

Requires values from last two ALU operations.
Remember destination register for operation.
Compare sources of current instruction to destinations of previous 2.
Data Hazards on Loads

LDUR $X0, [X31, 0]
SUB X3, $X0, X4
AND X5, $X0, X6
ORR X7, $X0, X8
EOR X9, $X0, X10

Clock

LDUR Ifetch Reg/Dec Exec Mem Wr
SUB Ifetch Reg/Dec Exec Mem Wr
AND Ifetch Reg/Dec Exec Mem Wr
ORR Ifetch Reg/Dec Exec Mem Wr
EOR Ifetch Reg/Dec Exec Mem Wr

Not solvable
Data Hazards on Loads (cont.)

Solution:

Use same forwarding hardware & register file for hazards 2+ cycles later
Force compiler to not allow register reads within a cycle of load
Fill delay slot, or insert no-op.

Delay slot: troublesome instructions put rules/restictions on the instruction that follows them.

Load delay slot: the next instruction cannot
Read the register I load into.

Branch delay slot: the instruction after the branch always executes.
Review Problem 35

- What should we do to this code to run it on a CPU with delay slots?

```
AND X0, X1, X2
ORRI X0, X0, #7
ADD X3, X4, X5
LDUR X6, [X3, #0]
CBNZ X6, FOO
B BAR

ADD X3, X4, X5
LDUR X6, [X3, #0]
ADD X0, X1, X2
CBNZ X6, FOO
ORRI X0, X0, #7
B BAR
(CADD X3!, X3!, X3!)
```
## Pipelined CPI, cycle time

CPI, assuming compiler can fill 50% of delay slots

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Type Cycles</th>
<th>Type Frequency</th>
<th>Cycles * Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>1</td>
<td>x</td>
<td>= 0.5</td>
</tr>
<tr>
<td>Load</td>
<td>1.5</td>
<td>x</td>
<td>= 0.3</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>x</td>
<td>= 0.1</td>
</tr>
<tr>
<td>Branch</td>
<td>1.5</td>
<td>x</td>
<td>= 0.3</td>
</tr>
</tbody>
</table>

### CPI:

CPI: 1.2

Pipelined: cycle time = 1ns.

\[
\text{Delay for 1M instr:} = \frac{1}{4} \times 1.2 \times 1\text{ns} + 4\text{ns} = 1.2\text{Mns} + 4\text{ns}
\]

Single cycle: CPI = 1.0, cycle time = 4.5ns.

\[
\text{Delay for 1M instr:} = \frac{1}{4} \times 1.0 \times 4.5\text{ns} = 4.5\text{Mns}
\]
Pipelined CPU Summary

Improve cycle time by pipelining

Hazards

Structural

Control

Data
Memory Hierarchy: Caches, Virtual Memory

Readings: 5.1-5.4, 5.8

Big memories are slow

Fast memories are small

Need to get fast, big memories
Random Access Memory

Dynamic Random Access Memory (DRAM)
High density, low power, cheap, but slow
Dynamic since data must be "refreshed" regularly
Random Access since arbitrary memory locations can be read

Static Random Access Memory
Low density, high power, expensive
Static since data held as long as power is on
Fast access time, often 2 to 10 times faster than DRAM

<table>
<thead>
<tr>
<th>Technology</th>
<th>Access Time</th>
<th>Cost/Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>1-7 cycles</td>
<td>10,000x</td>
</tr>
<tr>
<td>DRAM</td>
<td>100 cycles</td>
<td>200x</td>
</tr>
<tr>
<td>Flash</td>
<td>10,000 cycle</td>
<td>15x</td>
</tr>
<tr>
<td>Disk</td>
<td>10,000,000 cycles</td>
<td>1x</td>
</tr>
</tbody>
</table>
The Problem

Cost vs. Performance

Fast memory is expensive
Slow memory can significantly affect performance

Design Philosophy

Use a hybrid approach that uses aspects of both
Keep frequently used things in a small amount of fast/expensive memory
“Cache”
Place everything else in slower/inexpensive memory (even disk)
Make the common case fast
Locality

Programs access a relatively small portion of the address space at a time

```c
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= 'a' && *index <= 'z')
        *index = *index + ('A' - 'a');
    index++;
}
```

Types of Locality

Temporal Locality – If an item has been accessed recently, it will tend to be accessed again soon

Spatial Locality – If an item has been accessed recently, nearby items will tend to be accessed soon

Locality guides caching