CPU Pipelining Issues

What have you been beating your head against?

This pipe stuff makes my head hurt!

Finishing up Chapter 6
Structural Data Hazard

Consider LOADS:
Can we fix all these problems using our previous bypass paths?

For a `lw` instruction fetched during clock 1, data isn’t returned from memory until late into cycle i+3. Bypassing will fix `xor` but not `add`!

Load data hazards are complicated by the fact that their result is resolved later than the ALU pipeline stage.
Load Delays

Bypassing can’t fix the problem with add since the data simply isn’t available! In order to fix it we have to add *pipeline interlock hardware* to stall add’s execution, or else program around it.

```
lw $t4,0($t1)  
add $t5,$t1,$t4  
xor $t6,$t3,$t4  
```

This requires inserting a MUX just before the instruction register of the ALU stage, $\text{IR}^{\text{ALU}}$, to annul the add (by inserting a NOP) as well as, clock enables on the PC and IR pipeline registers of earlier pipeline stages to stall the execution without annuling any instructions. This is how the simulator, SPIM works.
Punting on Load Interlock

Early versions of MIPS did not include a pipeline interlock, thus, requiring the compiler/programmer to work around it.

If compiler knows about load delay, it can often rearrange the code sequence to eliminate the hazard. Many compilers can provide implementation-specific instruction scheduling. This requires no additional H/W, but it further confuses the instruction semantics. We’ll include interlocks for SPIM compatibility.
Load Delays (cont’d)

But, but, what about FASTER processors?

FACT: Processors have become very fast relative to memories!

Can we just stall the pipe longer? Add more NOPs?

ALTERNATIVE: Longer pipelines.

1. Add “MEMORY WAIT” stages between START of read operation & return of data.

2. Build pipelined memories, so that multiple (say, N) memory transactions can be in progress at once.

3. (Optional). Stall pipeline when the N limit is exceeded.

4-Stage pipeline requires READ access in LESS than one clock.

SOLUTION: A 5-Stage pipeline that allows nearly two clocks for data memory accesses...
5-Stage miniMIPS

- Omits some details
- NO bypass or interlock logic

Address is available right after instruction enters Memory stage

almost 2 clock cycles

Data is needed just before rising clock edge at end of Write Back stage
One More Fly in the Ointment

There is one more structural hazard that we have not discussed. That is, the saving, and subsequent accesses, of the return address resulting from the jump-and-link, jal, instruction.

Moreover, given that we have bought into a single delay-slot, which is always executed, we now need to store the address of the instruction following the delay slot instruction.

We need to return here, to PC+8, not PC+4. Once more we need to rewrite the ISA spec!

jal sqr # call procedure
addi $a0,$0,10 # delay slot
addi $t0,$v0,-1 # return address
Return Address Register Writes

```c
# The code: Assume Reg[LP] = 100...
add $ra,$0,$0
jal  f
addi $ra,$ra,4  # In delay slot
...
f: xor  $t0,$ra,$0
or   $r1,$0,$ra
add  $t2,$0,$ra
```

Can we make the regfile accesses of the 3 instructions following the jal work by bypassing?

Where do we get the right return address from?

---

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
<th>i+4</th>
<th>i+5</th>
<th>i+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add</td>
<td>jal</td>
<td>addi</td>
<td>xor</td>
<td>or</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>add</td>
<td>jal</td>
<td>addi</td>
<td>xor</td>
<td>or</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>add</td>
<td>jal</td>
<td>addi</td>
<td>xor</td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>add</td>
<td>jal</td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>add</td>
<td>jal</td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BR Decision Time
ADDI reads
BR writes
addi $31, $31, 4

jal f

add $ra, $0, $0

On JALs, the register file saves the next address from the DELAY SLOT instruction (often PC+8).

Note this bypass is routed from the PC pipeline not from the ALU output. Thus, we need to add bypass paths for PC\textsuperscript{MEM}. 

(fetching xor at f)
JAL PC Bypasses

(fetching or at f+4)

We need another PC\textsuperscript{ALU} bypass.

In this case, the bypass path supplies the $31 operand for the XOR instruction.

xor $8,$31,$0

Add $31,$31,4

jal f

add $ra,$0,$0

Comp 411 – Fall 2007

11/12/06
JAL PC Bypasses

(fetching add at f+8)

And, we need another PC\textsuperscript{MEM} bypass.

In this case, the bypass path supplies the $31$ operand for the OR instruction.

PC\textsuperscript{WB} is already taken care of, for the following ADD, using the WB stage bypass at the output of the WDSEL mux.
5-Stage miniMIPS

We wanted a simple, clean pipeline but...

• broke the sequential semantics of ISA by adding a branch delay-slot and early branch resolution logic

• added A/B bypass muxes to get data before it’s written to regfile

• added CLK EN to freeze IF/RF stages so we can wait for lw to reach WB stage
Bypass MUX Details

The previous diagram was oversimplified. Really need for the bypass muxes to precede the A and B muxes to provide the correct values for the jump target (JT), write data, and early branch decision logic.
Bypass Logic

miniMIPS bypass logic (need two copies for A/B data):

- R S or R T
- R T or R D (WB)*
- R T or R D (MEM)*
- R T or R D (ALU)*
- "O"

5-bit compare

Regfile (no bypass)
WB bypass
MEM bypass
ALU bypass
Select "O"

* If instruction is a sw (doesn’t write into regfile), set rt for ALU/MEM/WB to $0
Final 5-Stage miniMIPS

- Added branch delay slot and early branch resolution logic to fix a CONTROL hazard
- Added lots of bypass paths and detection logic to fix various STRUCTURAL hazards
- Added pipeline interlocks to fix load delay STRUCTURAL hazard
Pipeline Summary (I)

• **Started with unpipelined implementation**
  - direct execute, 1 cycle/instruction
  - it had a long cycle time: mem + regs + alu + mem + wb

• **We ended up with a 5-stage pipelined implementation**
  - increase throughput (3x???)
  - delayed branch decision (1 cycle)
    
    **Choose to execute instruction after branch**
  
  - delayed register writeback (3 cycles)
    
    **Add bypass paths (6 x 2 = 12) to forward correct value**
  
  - memory data available only in WB stage
    
    **Introduce NOPs at IRA\text{ALU}, to stall IF and RF stages until LD result was ready**
Pipeline Summary (II)

Fallacy #1: Pipelining is easy
Smart people get it wrong all of the time! Costs? Re-spins of the design. Force S/W folks to devise program/compiler workarounds.

Fallacy #2: Pipelining is independent of ISA
Many ISA decisions impact how easy/costly it is to implement pipelining (i.e. branch semantics, addressing modes). Bad decisions impact future implementations. (delay slot vs. annul?, load interlocks?) and break otherwise clean semantics. For performance, S/W must be aware!

Fallacy #3: Increasing Pipeline stages improves performance
RISC = Simplicity???

“The P.T. Barnum World’s Tallest Dwarf Competition”
World’s Most Complex RISC?

- RISC was conceived to be SIMPLE
- SIMPLE -> FAST
- MORE SPEED -> Pipelining
- Pipelining -> Complexity
- Complexity increases
delays in worst-case paths