Steps in Executing MIPS

1) **IFetch**: Fetch Instruction, Increment PC
2) **Decode** Instruction, Read Registers
3) **Execute**:
   - Mem-reference: Calculate Address
   - Arith-logical: Perform Operation
4) **Memory**:
   - Load: Read Data from Memory
   - Store: Write Data to Memory
5) **Write Back**: Write Data to Register
Pipelined Execution Representation

Every instruction must take the same number of steps, also called pipeline “stages”, so some will go idle sometimes.

Review: Datapath for MIPS

Use datapath figure to represent pipeline.
Problems for Computers

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  1. **Structural hazards**: HW cannot support some combination of instructions (e.g., read instruction and data from a single memory block)
  2. **Control hazards**: Pipelining of branches. Target address not known in time for next instruction fetch.
  3. **Data hazards**: Instruction depends on result of prior instruction still in the pipeline.

---

### Structural Hazard #1: Single Memory (1/2)

<table>
<thead>
<tr>
<th>Instr. Order</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read same memory twice in same clock cycle
Structural Hazard #1: Single Memory (2/2)

Solution:
- Infeasible and inefficient to create second main memory
- So simulate this by having two Level 1 Caches
- Have both an L1 Instruction Cache and an L1 Data Cache
- Need complex hardware to control when both caches miss

Structural Hazard #2: Registers (1/2)

Read and write registers simultaneously?
Structural Hazard #2: Registers (2/2)

Solution:
• Build registers with multiple ports, so can both read and write at the same time

What if read and write same register?
• Design to that it writes in first half of clock cycle, read in second half of clock cycle
• Thus will read what is written, reading the new contents

Data Hazards (1/2)

Consider the following sequence of instructions

add $t0, $t1, $t2
sub $t4, $t0, $t3
and $t5, $t0, $t6
or $t7, $t0, $t8
xor $t9, $t0, $t10
Data Hazards (2/2)

Dependencies backwards in time are hazards

Data Hazard Solution: Forwarding

• **Forward** result from one stage to another

```
add $t0,$t1,$t2
sub $t4,$t0,$t3
and $t5,$t0,$t6
or $t7,$t0,$t8
xor $t9,$t0,$t10
```

“or” hazard solved by register hardware
Data Hazard: Loads (1/5)

- Dependencies backwards in time are hazards

```
lw $t0, 0($t1)  
sub $t3, $t0, $t2
```

- Can’t solve with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/5)

- Hardware must stall pipeline
- Called “interlock”

```
lw $t0, 0($t1)  
sub $t3, $t0, $t2  
and $t5, $t0, $t4  
or $t7, $t0, $t6
```
Data Hazard: Loads (3/5)

° Instruction slot after a load is called “load delay slot”

° If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.

° If the compiler puts an unrelated instruction in that slot, then no stall

° Letting the hardware stall the instruction in the delay slot has the same net effect of putting a nop in the delay slot (except for the later uses more code space)

---

Data Hazard: Loads (4/5)

° Stall has same net effect as nop

lw $t0, 0($t1)  
sub $t3,$t0,$t2  
and $t5,$t0,$t4  
or $t7,$t0,$t6
Data Hazard: Loads (5/5)

° Stall has same net effect as nop

\[ \text{lw } \$t0, 0(\$t1) \]

\[ \text{nop} \]

\[ \text{sub } \$t3, \$t0, \$t2 \]

\[ \text{and } \$t5, \$t0, \$t4 \]

\[ \text{or } \$t7, \$t0, \$t6 \]

Control Hazard: Branching (1/6)

° Suppose we put branch decision-making hardware in ALU stage

• then two more instructions after the branch will always be fetched, whether or not the branch is taken

° Desired functionality of a branch

• if we do not take the branch, don’t waste any time and continue executing normally

• if we take the branch, don’t execute any instructions after the branch, just go to the desired label
Control Hazard: Branching (2/6)

° Initial Solution: Stall until decision is made
  • insert “no-op” instructions: those that accomplish nothing, just take time
  • Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard: Branching (3/6)

° Optimization #1:
  • move comparator up to Stage 2
  • as soon as instruction is decoded (Opcode identifies as a branch), immediately make a decision and set the value of the PC (if necessary)
  • Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  • Side Note: This means that branches are idle in Stages 3, 4 and 5.
Control Hazard: Branching (4/6)

° Insert a single no-op (bubble)

**Time (clock cycles)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IS</th>
<th>DS</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bubble</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

° Impact: 2 clock cycles per branch instruction ⇒ slow

---

Forwarding and Moving Branch Decision

° Forwarding/bypassing currently affects Execution stage:
  - Instead of using value from register read in Decode Stage, use value from ALU output or Memory output

° Moving branch decision from Execution Stage to Decode Stage means forwarding/bypassing must be replicated in Decode Stage for branches. I.e., Code below must still work:

```
addiu $s1, $s1, -4
beq  $s1, $s2, Exit
```
Control Hazard: Branching (5/6)

° Optimization #2: Redefine branches

• Old definition: if we take the branch, none of the instructions after the branch get executed by accident
• New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the **branch-delay slot**)

Control Hazard: Branching (6/6)

° Notes on Branch-Delay Slot

• Worst-Case Scenario: can always put a no-op in the branch-delay slot
• Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
  - Compiler can usually can find such an instruction at least 50% of the time
• Jumps also have a delay slot
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11
```

Delayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11
```

Exit:

How long to execute?

° Assume delayed branch, 5 stage pipeline, forwarding/bypassing, interlock on unresolved load hazards

```
Loop: lw $t0, 0($s1)
     addiu $t0, $t0, $s2
     sw $t0, 0($s1)
     addiu $s1, $s1, -4
     bne $s1, $zero, Loop
     nop
```

° How many clock cycles on average to execute this code per loop iteration?  
  a) <= 5  b) 6  c) 7  d) 8  e) >= 9

° (after 1000 iterations, so pipeline is full)
How long to execute?

° Assume delayed branch, 5 stage pipeline, forwarding/bypassing, interlock on unresolved hazards

° Look at this code:

Loop: 1. lw $t0, 0($s1)
2. (data hazard so stall)
3. addu $t0, $t0, $s2
4. sw $t0, 0($s1)
5. addiu $s1, $s1, -4
6. bne $s1, $zero, Loop
7. nop (delayed branch so exe nop)

° How many clock cycles to execute this code per loop iteration?
  a) <= 5  b) 6  c) 7  d) 8  e) >=9

Rewrite the loop to improve performance

° Rewrite this code to reduce clock cycles per loop to as few as possible:

Loop:  lw $t0, 0($s1)
     addu $t0, $t0, $s2
     sw $t0, 0($s1)
     addiu $s1, $s1, -4
     bne $s1, $zero, Loop
     nop

° How many clock cycles to execute your revised code per loop iteration?
  a) 4  b) 5  c) 6  d) 7
Rewrite the loop to improve performance

° Rewrite this code to reduce clock cycles per loop to as few as possible:

\[
\text{Loop: } \begin{align*}
1. \ & \text{lw} \quad & \text{t0} & \quad 0($s1) \\
2. \ & \text{addiu} \quad & s1, \ & s1, \ -4 \\
3. \ & \text{addu} \quad & \text{t0}, \ & \text{t0}, \ s2 \\
4. \ & \text{bne} \quad & s1, \ & \text{$zero}, \ \text{Loop} \\
5. \ & \text{sw} \quad & \text{t0}, \ & \text{+4($s1)} \\
\end{align*}
\]

(no hazard since extra cycle)

°How many clock cycles to execute your revised code per loop iteration?
   a) 4   b) 5   c) 6   d) 7

Things to Remember (1/2)

° Optimal Pipeline

• Each stage is executing part of an instruction each clock cycle.
• One instruction finishes during each clock cycle.
• On average, execute far quicker because of higher clock frequency.

° What makes this work?

• Similarities between instructions allow us to use same stages for all instructions (generally).
• Each stage takes about the same amount of time as all others: little wasted time.
Things to Remember (2/2)

° Pipelining a Big Idea: widely used concept

° What makes it less than perfect?
  • Structural hazards: suppose we had only one cache?
    ⇒ Need more HW resources
  • Control hazards: need to worry about branch instructions?
    ⇒ Delayed branch or branch prediction
  • Data hazards: an instruction depends on a previous instruction?