Steps in Executing MIPS

1) **IFetch**: Fetch Instruction, Increment PC
2) **Decode** Instruction, Read Registers
3) **Execute**: Mem-ref: Calculate Address
   Arith-log: 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 same number of steps, also called pipeline “stages”, so some will go idle sometimes
Review: Datapath for MIPS

Stage 1: Instruction Fetch
Stage 2: Decode/Register Read
Stage 3: Execute
Stage 4: Memory
Stage 5: Write Back

Use datapath figure to represent pipeline
Problems for Computers

° Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle

  - **Structural hazards**: HW cannot support this combination of instructions (e.g., read instruction and data from memory)
  
  - **Control hazards**: Pipelining of branches & other instructions *stall* the pipeline until the hazard “*bubbles*” in the pipeline
  
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (read and write same data)
Structural Hazard #1: Single Memory (1/2)

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 more complex hardware to control when both caches miss
Structural Hazard #2: Registers (1/2)

Read and write registers simultaneously?
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

\[
\begin{align*}
\text{add} & \quad \textit{\$t0}, \quad \textit{\$t1}, \quad \textit{\$t2} \\
\text{sub} & \quad \textit{\$t4}, \quad \textit{\$t0}, \quad \textit{\$t3} \\
\text{and} & \quad \textit{\$t5}, \quad \textit{\$t0}, \quad \textit{\$t6} \\
\text{or} & \quad \textit{\$t7}, \quad \textit{\$t0}, \quad \textit{\$t8} \\
\text{xor} & \quad \textit{\$t9}, \quad \textit{\$t0}, \quad \textit{\$t10}
\end{align*}
\]
Data Hazards (2/2)

Dependencies backwards in time are hazards

Time (clock cycles)

Instr Order

add $t0,$t1,$t2
sub $t4,$t0,$t3
and $t5,$t0,$t6
or $t7,$t0,$t8
xor $t9,$t0,$t10
Data Hazard Solution: Forwarding

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

\[
\begin{align*}
\text{add } & \texttt{\$t0,\$t1,\$t2} \\
\text{sub } & \texttt{\$t4,\$t0,\$t3} \\
\text{and } & \texttt{\$t5,\$t0,\$t6} \\
\text{or } & \texttt{\$t7,\$t0,\$t8} \\
\text{xor } & \texttt{\$t9,\$t0,\$t10}
\end{align*}
\]

“\text{or}” hazard solved by register hardware
Data Hazard: Loads (1/2)

- Dependencies backwards in time are hazards

\[
\begin{align*}
\text{lw } & \texttt{ $t0,0($t1) } \\
\text{sub } & \texttt{ $t3,$t0,$t2 }
\end{align*}
\]

- Can’t solve with forwarding
- Must stall instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/2)

- Hardware must insert no-op in pipeline

\[
\begin{align*}
lw & \quad \text{\$t0, 0(\$t1)} \\
\text{sub} & \quad \text{\$t3, \$t0, \$t2} \\
\text{and} & \quad \text{\$t5, \$t0, \$t4} \\
\text{or} & \quad \text{\$t7, \$t0, \$t6}
\end{align*}
\]
Administrivia: Rest of 61C

• Rest of 61C slower pace

F 12/1 Review: Caches/TLB/VM; Section 7.5

M 12/4 Deadline to correct your grade record

W 12/6 Review: Interrupts (A.7); Feedback lab

F 12/8 61C Summary / Your Cal heritage / HKN Course Evaluation

Sun 12/10 Final Review, 2PM (155 Dwinelle)

Tues 12/12 Final (5PM 1 Pimintel)

Final: Just bring pencils: leave home back packs, cell phones, calculators

Will check that notes are handwritten

Got a final conflict? Email now for Beta
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 is 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.
Insert a single no-op (bubble)

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
  ```
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
  - re-ordering instructions is a common method of speeding up programs
  - compiler must be very smart in order to find instructions to do this
  - usually can find such an instruction at least 50% of the time
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

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

Delayed Branch

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

Exit:
Try “Peer-to-Peer” Instruction

° Given question, everyone has one minute to pick an answer

° First raise hands to pick

° Then break into groups of 5, talk about the solution for a few minutes

° Then vote again (each group all votes together for the groups choice)

° Discussion should lead to convergence

° Give the answer, and see if there are questions

° Will try this twice today
How long to execute?

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

Loop:  
\[
\begin{align*}
&\text{lw} \quad $t0, 0($s1) \\
&\text{addiu} \quad $t0, $t0, $s2 \\
&\text{sw} \quad $t0, 0($s1) \\
&\text{addiu} \quad $s1, $s1, -4 \\
&\text{bne} \quad $s1, $zero, \text{Loop} \\
&\text{nop}
\end{align*}
\]

° How many clock cycles on average to execute this code per loop iteration?  
  a) \( \leq 5 \)  
  b) 6  
  c) 7  
  d) 8  
  e) \( \geq 9 \)

° (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. addiu $t0, $t0, $s2
4. sw $t0, 0($s1)
5. addiu $s1, $s1, -4
6. bne $s1, $zero, Loop
7. nop (delayed branch so execute 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:

Loop:
1. lw \$t0, 0($s1)
2. addiu \$s1, \$s1, -4
3. addu \$t0, \$t0, \$s2
4. bne \$s1, \$zero, Loop
5. sw \$t0, +4($s1)

(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

(modified sw to put past addiu)
State of the Art: Pentium 4

° 1 8KB Instruction cache, 1 8 KB Data cache, 256 KB L2 cache on chip

° Clock cycle = 0.67 nanoseconds, or 1500 MHz clock rate (667 picoseconds, 1.5 GHz)

° HW translates from 80x86 to MIPS-like micro-ops

° 20 stage pipeline

° Superscalar: fetch, retire up to 3 instructions /clock cycle; Execution out-of-order

° Faster memory bus: 400 MHz
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 more quickly.

° 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?