Football Student Season Tickets are now on-sale to incoming Freshmen, Transfers, and Graduate students (that’s me!). There is no longer an excuse for anyone not to have their season tickets (unless you aren’t a Cal student, in which case, single game tickets are on-sale now).

http://www.calbears.com
Review

• Pipelining is a BIG idea

• 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.
Problems for Pipelining CPUs

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support some combination of instructions (single person to fold and put clothes away)
  - **Control hazards**: Pipelining of branches causes later instruction fetches to wait for the result of the branch
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline

- These might result in pipeline stalls or “bubbles” in the pipeline.
Steps in Executing MIPS

1) **IFetch**: Instruction Fetch, Increment PC

2) **Dec**: Instruction Decode, Read Registers

3) **Exec**:  
   - Mem-ref: Calculate Address  
   - Arith-log: Perform Operation

4) **Mem**:  
   - Load: Read Data from Memory  
   - Store: Write Data to Memory

5) **WB**: Write Data Back to Register
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 memory
  • (We’ll learn about this more this week)
  • so simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
  • 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)

Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is VERY fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports

- Result: can perform Read and Write during same clock cycle
Control Hazard: Branching (1/9)

Where do we do the compare for the branch?
Control Hazard: Branching (2/9)

• We had put branch decision-making hardware in ALU stage
  • therefore 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 (3/9)

- **Initial Solution**: Stall until decision is made
  - insert “no-op” instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles).
  - **Drawback**: branches take 3 clock cycles each (assuming comparator is put in ALU stage)
Control Hazard: Branching (4/9)

• Optimization #1:
  • insert special branch comparator in Stage 2
  • as soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  • 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 (5/9)

Branch comparator moved to Decode stage.

Instr 1
Instr 2
Instr 3
Instr 4

In order

Time (clock cycles)
Control Hazard: Branching (6/9)

• User inserting no-op instruction

Impact: 2 clock cycles per branch instruction ⇒ slow
Control Hazard: Branching (7/9)

- Controller inserting a single bubble

Impact: 2 clock cycles per branch instruction ⇒ slow

...story about engineer, physicist, mathematician asked to build a fence around a flock of sheep using minimal fence...
Control Hazard: Branching (8/9)

- 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)

- The term “Delayed Branch” means we always execute inst after branch

- This optimization is used with MIPS
Control Hazard: Branching (9/9)

• 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
    ▪ 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
```
Administrivia

• HW8 due date being moved to Wednesday at midnight.

• Project 2 still due next Monday. You are encouraged to work with a partner.

• Other administrivia?
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)

- Data-flow backward 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

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 (1/4)

- Dataflow backwards in time are hazards

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

- Can’t solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/4)

- **Hardware** stalls pipeline
  - Called “interlock”

```
lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or  $t7,$t0,$t6
```
Data Hazard: Loads (3/4)

• 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 is equivalent to putting a nop in the slot (except the latter uses more code space)
• Stall is equivalent to nop

```
lw $t0, 0($t1)
nop
sub $t3,$t0,$t2
and $t5,$t0,$t4
or $t7,$t0,$t6
```
1) Thanks to pipelining, I have *reduced the time* it took me to wash my one shirt.

2) Longer pipelines are *always a win* (since less work per stage & a faster clock).

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b)</td>
<td>FT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c)</td>
<td>TF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d)</td>
<td>TT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Peer Instruction (1/2)

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after $10^3$ loops, so pipeline full)

Loop:

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

- How many pipeline stages (clock cycles) per loop iteration to execute this code?

  - a) 5
  - b) 6
  - c) 7
  - d) 8
  - e) 9
Peer Instruction (2/2)

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after $10^3$ loops, so pipeline full). Rewrite this code to reduce pipeline stages (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 pipeline stages (clock cycles) per loop iteration to execute this code?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>8</td>
</tr>
</tbody>
</table>
“And in Conclusion..”

- Pipeline challenge is hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary

- More aggressive performance:
  - Superscalar
  - Out-of-order execution
  - See bonus slides
**Bonus slides**

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.

- The slides will appear in the order they would have in the normal presentation
Historical Trivia

- First MIPS design did not interlock and stall on load-use data hazard

- Real reason for name behind MIPS: Microprocessor without Interlocked Pipeline Stages

  - Word Play on acronym for Millions of Instructions Per Second, also called MIPS
Pipeline Hazard: Matching socks in later load

- A depends on D; stall since folder tied up; Note this is much different from processor cases so far. We have not had an earlier instruction depend on a later one.
Out-of-Order Laundry: Don’t Wait

- A depends on D; rest continue; need more resources to allow out-of-order
Superscalar Laundry: Parallel per stage

- More resources, HW to match mix of parallel tasks?
Superscalar Laundry: Mismatch Mix

- Task mix underutilizes extra resources

![Graph showing task mix and time]

6 PM 7 8 9 10 11 12 1 2 AM

Time

30 30 30 30 30 30 30

(light clothing)

(light clothing)

(dark clothing)

(light clothing)