CS 61C: Great Ideas in Computer Architecture

Pipelining Hazards

Guest Lecturer: Justin Hsia
Great Idea #4: Parallelism

**Software**

- **Parallel Requests**
  Assigned to computer
  e.g. search “Garcia”

- **Parallel Threads**
  Assigned to core
  e.g. lookup, ads

- **Parallel Instructions**
  > 1 instruction @ one time
  e.g. 5 pipelined instructions

- **Parallel Data**
  > 1 data item @ one time
  e.g. add of 4 pairs of words

**Hardware**

Leverage Parallelism & Achieve High Performance

- **Warehouse Scale Computer**

- **Core**
  - Instruction Unit(s)
    - A0+B0, A1+B1, A2+B2, A3+B3
  - Functional Unit(s)
  - Cache Memory
  - Memory
  - Input/Output

**Hardware descriptions**

- All gates functioning in parallel at same time

4/12/2013 Spring 2013 -- Lecture #31
Review of Last Lecture

- Implementing controller for your datapath
  - Take decoded signals from instruction and generate control signals
  - Use “AND” and “OR” Logic scheme

- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for MIPS: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - What can go wrong???
Agenda

• Pipelining Performance
• Structural Hazards
• Administrivia
• Data Hazards
  – Forwarding
  – Load Delay Slot
• Control Hazards
Review: Pipelined Datapath
Pipelined Execution Representation

- Every instruction must take same number of steps, so some stages will idle
  - e.g. MEM stage for any arithmetic instruction
Graphical Pipeline Diagrams

- Use datapath figure below to represent pipeline:

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
Graphical Pipeline Representation

- RegFile: left half is write, right half is read

**Time (clock cycles)**

**Instr Order**
- Load
- Add
- Store
- Sub
- Or
Pipelining Performance (1/3)

• Use $T_c$ (“time between completion of instructions”) to measure speedup
  \[ T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}} \]
  — Equality only achieved if stages are balanced (i.e. take the same amount of time)

• If not balanced, speedup is reduced

• Speedup due to increased throughput
  — *Latency* for each instruction does not decrease
Pipelining Performance (2/3)

• Assume time for stages is
  – 100ps for register read or write
  – 200ps for other stages

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

• What is pipelined clock rate?
  – Compare pipelined datapath with single-cycle datapath
Pipelining Performance (3/3)

Single-cycle
\[ T_c = 800 \text{ ps} \]

Pipelined
\[ T_c = 200 \text{ ps} \]
Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) **Structural hazard**
   - A required resource is busy
     (e.g. needed in multiple stages)

2) **Data hazard**
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write

3) **Control hazard**
   - Flow of execution depends on previous instruction
Agenda

• Pipelining Performance
• Structural Hazards
• Administrivia
• Data Hazards
  – Forwarding
  – Load Delay Slot
• Control Hazards
1. Structural Hazards

• Conflict for use of a resource
• MIPS pipeline with a single memory?
  – Load/Store requires memory access for data
  – Instruction fetch would have to \textit{stall} for that cycle
    • Causes a pipeline “\textit{bubble}”
• Hence, pipelined datapaths require separate instruction/data memories
  – Separate L1 I$ and L1 D$ take care of this
Structural Hazard #1: Single Memory

- **Instr 1**: Load
- **Instr 2**: ALU
- **Instr 3**: ALU
- **Instr 4**: ALU

**Time (clock cycles)**

- **I$:** Source of instructions
- **Reg:** Register
- **D$:** Source of data

**Comment:**

- Trying to read the same memory twice in the same clock cycle.
Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

• Two different solutions have been used:
  1) Split RegFile access in two: Write during 1\textsuperscript{st} half and Read during 2\textsuperscript{nd} half of each clock cycle
     • Possible because RegFile access is \textit{VERY} fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports

• \textbf{Conclusion:} Read and Write to registers during same clock cycle is okay
Agenda

• Pipelining Performance
• Structural Hazards
• Administrivia
• Data Hazards
  – Forwarding
  – Load Delay Slot
• Control Hazards
Administrivia

• Project 2: Performance Optimization
  – Part 1 due Sunday (4/14)
  – Part 2 released by Sunday night, due 4/21
  – Built-in performance competition for Part 2!

• Never too early to start looking at past exams!
  – See Piazza post @1492

4/12/2013
Agenda

• Pipelining Performance

• Structural Hazards

• Administrivia

• Data Hazards
  – Forwarding
  – Load Delay Slot

• Control Hazards
2. 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
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards

![Diagram of data hazards]

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

Time (clock cycles)

4/12/2013 Spring 2013 -- Lecture #31
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it’s not stored in RegFile yet

\begin{align*}
\text{add } & $t0, $t1, $t2 \\
\text{sub } & $t4, $t0, $t3 \\
\text{and } & $t5, $t0, $t6 \\
\text{or } & $t7, $t0, $t8 \\
\text{xor } & $t9, $t0, $t10
\end{align*}
Datapath for Forwarding (1/2)

• What changes need to be made here?
Datapath for Forwarding (2/2)

• Handled by *forwarding unit*
Data Hazard: Loads (1/4)

- **Recall:** 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 “hardware interlock”

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

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

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

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

Schematically, this is what we want, but in reality stalls done “horizontally”

How to stall just part of pipeline?
Data Hazard: Loads (3/4)

- Stall is equivalent to $nop$
  
  \[
  \text{lw } t0, 0(t1) \\
  \text{nop} \\
  \text{sub } t3, t0, t2 \\
  \text{and } t5, t0, t4 \\
  \text{or } t7, t0, t6
  \]
Data Hazard: Loads (4/4)

• Slot after a load is called a load delay slot
  – If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  – Letting the hardware stall the instruction in the delay slot is equivalent to putting a \texttt{nop} in the slot (except the latter uses more code space)

• **Idea:** Let the compiler put an unrelated instruction in that slot \(\rightarrow\) no stall!
Code Scheduling to Avoid Stalls

• Reorder code to avoid use of load result in the next instruction!

• **MIPS code for** \( D = A + B; \quad E = A + C; \)

```
# Method 1:
lw $t1, 0($t0)  \quad lw $t2, 4($t0)
add $t3, $t1, $t2  \quad lw $t4, 8($t0)
sw $t5, 12($t0)  \quad add $t5, $t1, $t4
lw $t5, 16($t0)
```

13 cycles

```
# Method 2:
lw $t1, 0($t0)  \quad lw $t2, 4($t0)
lw $t4, 8($t0)  \quad add $t3, $t1, $t2
add $t5, $t1, $t4  \quad sw $t5, 16($t0)
```

11 cycles

Stall!

Stall!
Agenda

• More Pipelining
• Structural Hazards
• Administrivia
• Data Hazards
  – Forwarding
  – Load Delay Slot
• Control Hazards
3. Control Hazards

• **Branch** \((\text{beq, bne})\) determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on ID stage of branch

• **Simple Solution:** Stall on *every* branch until we have the new PC value
  – How long must we stall?
Branch Stall

- When is comparison result available?

TWO bubbles required per branch!
Summary

• Hazards reduce effectiveness of pipelining
  – Cause stalls/bubbles

• Structural Hazards
  – Conflict in use of datapath component

• Data Hazards
  – Need to wait for result of a previous instruction

• Control Hazards
  – Address of next instruction uncertain/unknown
  – More to come next lecture!
Question: For each code sequences below, choose one of the statements below:

1:
- `lw $t0,0($t0)
- `add $t1,$t0,$t0

2:
- `add $t1,$t0,$t0
- `addi $t2,$t0,5
- `addi $t4,$t1,5

3:
- `addi $t1,$t0,1
- `addi $t2,$t0,2
- `addi $t3,$t0,4
- `addi $t5,$t1,5

A) No stalls as is
B) No stalls with forwarding
C) Must stall
Code Sequence 1

Time (clock cycles)

I$ \quad \text{Reg} \quad \text{ALU} \quad \text{Reg} \quad \text{D$} \quad \text{Reg}

Must stall

Instruction Order:
- lw
- add
- instr
- instr

7/25/2012  Summer 2012 -- Lecture #22
Code Sequence 2

**Instr Order**
- add
- addi
- addi
- instr

**Time (clock cycles)**

- SS $\rightarrow$ Reg
- add $\rightarrow$ Reg
- addi $\rightarrow$ Reg
- instr $\rightarrow$ Reg

**ALU**

- SS $\rightarrow$ ALU
- add $\rightarrow$ ALU
- addi $\rightarrow$ ALU
- instr $\rightarrow$ ALU

- I$\rightarrow$ Reg
- DS $\rightarrow$ Reg
- Reg $\rightarrow$ DS
- Reg $\rightarrow$ ALU

**Forwarding**
- no forwarding
- forward

**Stalls**
- no stalls with forwarding

---

7/25/2012
Summer 2012 -- Lecture #22
Code Sequence 3

Time (clock cycles)

Instr Order

addi
addi
addi
addi

No stalls as is