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**

- **Warehouse Scale Computer**
- **Leverage Parallelism & Achieve High Performance**

**Computer**

- Core
- ... Core
- Memory
- Input/Output

**Instruction Unit(s)**

- Logic Gates

**Functional Unit(s)**

- $A_0 + B_0$
- $A_1 + B_1$
- $A_2 + B_2$
- $A_3 + B_3$

**Cache Memory**

**Logic Gates**

All gates functioning in parallel at the same time
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

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

- Use datapath figure below to represent pipeline:
Graphical Pipeline Representation

- RegFile: left half is write, right half is read
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
  – \textit{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

\[ T_c = 800 \text{ ps} \]

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$

Pipelined

\[ T_c = 200 \text{ ps} \]

- lw $1, 100(0)$
- lw $2, 200(0)$
- lw $3, 300(0)$
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 stall for that cycle
    • Causes a pipeline “bubble”
• Hence, pipelined datapaths require separate instruction/data memories
  – Separate L1 I$ and L1 D$ take care of this
Structural Hazard #1: Single Memory

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

**Time (clock cycles)**

- Trying to read the same memory twice in the same clock cycle.
Structural Hazard #2: Registers (1/2)

Time (clock cycles)

Load
Instr 1
Instr 2
Instr 3
Instr 4

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

• Check-in
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

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

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

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

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

• Can’t solve all cases with forwarding
  – Must *stall* instruction dependent on load, then forward (more hardware)
• *Hardware* stalls pipeline
  – Called “hardware interlock”

```
lw $t0, 0($t1)
sub $t3,$t0,$t2
and $t5,$t0,$t4
or  $t7,$t0,$t6
```

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

How to stall just *part* of pipeline?
Stall is equivalent to `nop`

`lw $t0, 0($t1)`

`nop`

`sub $t3,$t0,$t2`

and `$t5,$t0,$t4`

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 `nop` in the slot (except the latter uses more code space)

- **Idea**: Let the compiler put an unrelated instruction in that slot → no stall!
Code Scheduling to Avoid Stalls

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

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

<table>
<thead>
<tr>
<th># Method 1:</th>
<th># Method 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t1, 0($t0)</td>
<td>lw $t1, 0($t0)</td>
</tr>
<tr>
<td>lw $t2, 4($t0)</td>
<td>lw $t2, 4($t0)</td>
</tr>
<tr>
<td>add $t3, $t1, $t2</td>
<td>add $t3, $t1, $t2</td>
</tr>
<tr>
<td>sw $t3, 12($t0)</td>
<td>sw $t3, 12($t0)</td>
</tr>
<tr>
<td>lw $t4, 8($t0)</td>
<td>lw $t4, 8($t0)</td>
</tr>
<tr>
<td>add $t5, $t1, $t4</td>
<td>add $t5, $t1, $t4</td>
</tr>
<tr>
<td>sw $t5, 16($t0)</td>
<td>sw $t5, 16($t0)</td>
</tr>
</tbody>
</table>

13 cycles

Stall!

11 cycles

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?

Time (clock cycles)

Instr Order
beq Instr 1 Instr 2 Instr 3 Instr 4

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!
Code Sequence 1

I$  Reg  ALU  D$  Reg  ALU  I$  Reg  D$  Reg  ALU  I$  Reg  D$  Reg  ALU  I$  Reg  D$  Reg  ALU  I$  Reg  D$  Reg  ALU  I$  Reg  D$  Reg  ALU  I$  Reg  D$  Reg

Instr Order
lw  add  instr  instr  instr

Time (clock cycles)

Must stall
Code Sequence 2

Time (clock cycles)

Instr Order

add
addi
addi
instr
instr

forwarding

No stalls with forwarding

no forwarding

35
Code Sequence 3

Time (clock cycles)

Instr Order

addi
addi
addi
addi
addi

No stalls as is