CS 61C: Great Ideas in Computer Architecture (Machine Structures)  
*Instruction Level Parallelism—Pipelining*

Instructor: Michael Greenbaum

---

You Are Here!

- Parallel Requests
  - Assigned to computer (e.g., Search “Katz”)
- 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 descriptions
  - All gates functioning in parallel at same time

---

Review: Pipelined Datapath

Graphical Pipeline Representation

- **Instr Order**: Load, Add, Store, Sub, Or

---

Agenda

- Pipelining Performance
- Pipelining Hazards
- Administrivia
- Pipelining Hazards (cont’d)
- Break
- Multiple Instruction Issue

---

Pipeline Performance

- Assume time for stages is:
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instruction</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>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>700ps</td>
</tr>
<tr>
<td>r-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100ps</td>
<td>500ps</td>
</tr>
</tbody>
</table>
Pipeline Performance

<table>
<thead>
<tr>
<th>Program execution time (in instructions)</th>
<th>Single-cycle (Tc = 800ps)</th>
<th>Pipelined (Tc = 200ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>Reg</td>
<td>ALU</td>
</tr>
<tr>
<td>1</td>
<td>$1, 100(80)$</td>
<td>800 ps</td>
</tr>
<tr>
<td>2</td>
<td>$2, 200(80)$</td>
<td>200 ps</td>
</tr>
<tr>
<td>3</td>
<td>$3, 300(80)$</td>
<td>300 ps</td>
</tr>
</tbody>
</table>

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions $T_{\text{pipelined}}$ = Time between instructions $T_{\text{nonpipelined}}$ / Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Agenda

- Pipelining Performance
- Pipelining Hazards
- Administrivia
- Pipelining Hazards (cont’d)
- Break
- Multiple Instruction Issue

Hazards

Situations that prevent starting the next instruction in the next clock cycle

1. Structural hazards
   - 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

1. Structural Hazards

- Conflict for use of a resource
- In 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

Time (clock cycles)
Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?

1. 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

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

Data Hazard Solution: Forwarding

- Forward result from one stage to another

Corrected Datapath for Forwarding?
Data Hazard: Loads (1/4)
• Dataflow backwards in time are hazards

![Diagram of pipeline stages with data hazard]

• 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”

![Diagram of pipeline stages with interlock]

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)

Data Hazard: Loads (4/4)
• Stall is equivalent to nop

![Diagram of pipeline stages with nop]

Administrivia
• I’ll be gone for tomorrow, Justin (TA) will be giving a guest lecture.
• Project 2 Part 2 due Sunday.
  — Slides at end of July 12 lecture contain useful info.
• Lab 12 cancelled!
  — Replaced with free study session where you can catch up on labs / work on project 2.
  — The TA’s will still be there.

Agenda
• Pipelining Performance
• Pipelining Hazards
• Administrivia
• Pipelining Hazards (cont’d)
• Break
• Multiple Instruction Issue
Pipelining and ISA Design

• MIPS Instruction Set designed for pipelining
  – MIPS originally stood for Microprocessor without Interlocked Pipeline Stages.
• All instructions are 32-bits
  – Easier to fetch and decode in one cycle
  – x86: 1- to 17-byte instructions
    (x86 HW actually translates to internal RISC instructions!)
• Few and regular instruction formats, 2 source register fields always in same place
  – Can decode and read registers in one step
• Memory operands only in Loads and Stores
  – Can calculate address 3rd stage, access memory 4th stage
• Alignment of memory operands
  – Memory access takes only one cycle

3. Control Hazards

• Branch determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on Decode stage of branch
• BEQ, BNE in MIPS pipeline
  • Simple solution Option 1: **Stall** on every branch until have new PC value
    – Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks

Where do we do the compare for the branch?

3. Control Hazard: Branching

• 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: means that branches are idle in Stages 3, 4 and 5

Corrected Datapath for BEQ/BNE?

One Clock Cycle Stall

Branch comparator moved to Decode stage.
3. Control Hazards

- Option 2: *Predict* outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

3. Control Hazard: Branching

- Option #3: 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 (the *branch-delay slot*)
  - *Delayed Branch* means *we always execute inst after branch*
  - This optimization is used with MIPS

3. Control Hazard: Branching

- Notes on *Branch-Delay Slot*
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of time
    - Jumps also have a delay slot ...

Example: Nondelayed vs. Delayed Branch

- Nondelayed Branch:
  - `lw $t1, 0($t0)`
  - `lw $t2, 4($t0)`
  - `add $t3, $t1, $t2`
  - `sw $t3, 12($t0)`
  - `lw $t4, 8($t0)`
  - `add $t5, $t1, $t4`
  - `sw $t5, 16($t0)`
  - 13 cycles

- Delayed Branch:
  - `lw $t1, 0($t0)`
  - `lw $t2, 4($t0)`
  - `lw $t4, 8($t0)`
  - `add $t3, $t1, $t4` → stall
  - `lw $t5, 16($t0)` → stall
  - 11 cycles

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?
- *JAL executes following instruction (PC+4) so should return to PC+8*

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F:

```c
lw $t1, 0($t0)        lw $t2, 4($t0)        lw $t3, 8($t0)
lw $t3, 12($t0)       add $t1, $t3, $t2
sw $t4, 16($t0)       add $t3, $t1, $t4
lw $t5, 16($t0)       add $t5, $t1, $t5
sw $t5, 16($t0)
```

- 10 cycles
I. Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

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

A) (red) I is True and II is True
B) (blue) I is False and II is True
C) (green) I is True and II is False
D) (yellow) I is False and II is False

Peer Question: Stall, Forward, OK?
For each code sequence, chose whether
I. It must stall
II. It can avoid stalls using only forwarding
III. It can execute without stalling or forwarding

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 $t3,$t0,2 addi $t3,$t0,4 addi $t5,$t1,5

Red: 1 II, 2 II, 3 III
White: 1 I, 2 I, 3 II
Blue: 1 II, 2 III, 3 III
Green: 1 I, 2 I, 3 III
Yellow: 1 II, 2 II, 3 III

Sequence 1

Time (clock cycles)

Instruction Order

Instruct.

Sequence 2

Time (clock cycles)

Instruction Order

Sequence 3

Time (clock cycles)

Instruction Order

Throughput better, not latency!
“...longer pipelines do usually mean faster clock rate, but hazards can cause problems!”
For each code sequence, choose whether
I. It must stall
II. It can avoid stalls using only forwarding
III. It can execute without stalling or forwarding

Peer Question: Stall, Forward, OK?

1: \( \text{lw} \ $t0,0($t0) \)
2: \( \text{add} \ $t1,$t0,$t0 \)
3: add \( $t1,$t0,$t0 \)

White: 1 I, 2 I, 3 II
Green: 1 I, 2 I, 3 II
Purple: All II (must forward)

Red: 1 II, 2 II, 3 III

Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage \( \Rightarrow \) shorter clock cycle
- Multiple issue (superscalar)
  - Replicate pipeline stages \( \Rightarrow \) multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4 GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions
to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Superscalar Laundry: Parallel per stage

- More resources, HW to match mix of parallel tasks?

Pipeline Depth and Issue Width

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Pipeline Depth and Issue Width

Static Multiple Issue

• Compiler groups instructions into issue packets
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
• Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations

Scheduling Static Multiple Issue

• Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
  - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

• Dual-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    • ALU/branch, then load/store
  - Pad an unused instruction with nop

Hazards in the Dual-Issue MIPS

• More instructions executing in parallel
• EXECUTE stage data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
  • add $t0, $s0, $s1
  • load $s2, 0($t0)
  • Split into two packets, effectively a stall
• Load-use hazard
  - Still one cycle use latency, but now two instructions
• More aggressive scheduling required

Scheduling Example

• Schedule this for dual-issue MIPS

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
Loop Unrolling

- Replicate loop body to expose more parallelism
- Use different registers per replication
  - Called register renaming
  - Avoid loop-carried anti-dependencies
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

<table>
<thead>
<tr>
<th>All/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>lw $t1, 12($s1)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>sw $t2, 8($s1)</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

“And in Conclusion, …”

- Instruction Level Parallelism (ILP)
  - Achieved by Pipelining, Multiple Instruction Issue
- Hazards are the enemy of ILP
  - Structural Hazards
  - Data Hazards
  - Control Hazards