CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instruction Level Parallelism—Pipelining

Instructor: Michael Greenbaum
You Are Here!

**Software**

- **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**

- **Harness Parallelism & Achieve High Performance**
  [Warehouse Scale Computer]

- **Computer**
  [Diagram of computer architecture]

- **Core**

- **Memory**
  [Cache]

- **Input/Output**

- **Instruction Unit(s)**

- **Functional Unit(s)**
  \( A_0 + B_0 \)
  \( A_1 + B_1 \)
  \( A_2 + B_2 \)
  \( A_3 + B_3 \)

- **Cache**

- **Logic Gates**

---

7/27/2011

Summer 2011 -- Lecture #22
Review: Pipelined Datapath
Graphical Pipeline Representation
(In Reg, right half highlight read, left half write)

Time (clock cycles)

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>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>
Pipeline Performance

Single-cycle ($T_c = 800\text{ps}$)

Program execution order (in instructions)

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

Time

200 400 600 800 1000 1200 1400 1600 1800

Pipelined ($T_c = 200\text{ps}$)

Program execution order (in instructions)

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

Time

200 400 600 800 1000 1200 1400
Pipeline Speedup

• If all stages are balanced
  – i.e., all take the same time
  – Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{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)

Load
Instr 1
Instr 2
Instr 3
Instr 4

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

7/26/2011
Summer 2011 -- Lecture #22
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

**Time (clock cycles)**

**Instruction 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)
Corrected Datapath for Forwarding?
Data Hazard: Loads (1/4)

- Dataflow backwards in time are hazards
- Can’t solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

\[
\text{lw } \$t0,0(\$t1) \\
\text{sub } \$t3,\$t0,\$t2
\]
Data Hazard: Loads (2/4)

- Hardware stalls pipeline
  - Called “interlock”

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

- Stall is equivalent to `nop`.

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

- `nop`

- `sub $t3,$t0,$t2` and `$t5,$t0,$t4`

- Or `$t7,$t0,$t6`
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.

Time (clock cycles)
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

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

Exit:

Delayed Branch

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

Exit:
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 \( A = B + E; \ C = B + F; \)

\[
\begin{align*}
\text{lw} & \quad \text{\$t1, 0($t0)} \\
\text{lw} & \quad \text{\$t2, 4($t0)} \\
\text{add} & \quad \text{\$t3, \$t1, \$t2} \\
\text{sw} & \quad \text{\$t3, 12($t0)} \\
\text{lw} & \quad \text{\$t4, 8($t0)} \\
\text{add} & \quad \text{\$t5, \$t1, \$t4} \\
\text{sw} & \quad \text{\$t5, 16($t0)}
\end{align*}
\]

13 cycles

\[
\begin{align*}
\text{lw} & \quad \text{\$t1, 0($t0)} \\
\text{lw} & \quad \text{\$t2, 4($t0)} \\
\text{lw} & \quad \text{\$t4, 8($t0)} \\
\text{add} & \quad \text{\$t3, \$t1, \$t2} \\
\text{sw} & \quad \text{\$t3, 12($t0)} \\
\text{add} & \quad \text{\$t5, \$t1, \$t4} \\
\text{sw} & \quad \text{\$t5, 16($t0)}
\end{align*}
\]

11 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
1) **Throughput** better, not latency!

2) “…longer pipelines do usually mean faster clock rate, but hazards can cause problems!”

1) Thanks to pipelining, I have [redacted] the time it took me to wash my one shirt.

2) Longer pipelines are [redacted] (since less work per stage & a faster clock).
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

1:
1w $t0,0($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
add $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$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
Purple: All II (must forward)
Yellow: 1 I, 2 II, 3 III
Sequence 1

Time (clock cycles)

Instr Order

lw add instr instr instr

IS Reg ALU D$ Reg
IS Reg ALU
IS Reg ALU
IS Reg ALU
IS Reg ALU D$
IS Reg ALU D$
IS Reg ALU D$
IS Reg ALU D$

7/27/2011

Spring 2011 -- Lecture #21
Sequence 2

Time (clock cycles)

Instr. Order

add
addi
addi
instr
instr
Sequence 3

Time (clock cycles)

Instr. Order

addi

addi

addi

addi

addi
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)
ad $t1,$t0,$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 $t4,$t1,#5
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
Purple: All II (must forward)

Yellow: 1 I, 2 II, 3 III
Agenda

• Pipelining Performance
• Pipelining Hazards
• Administrivia
• Pipelining Hazards (cont’d)
• Break
• Multiple Instruction Issue
Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage ⇒ shorter clock cycle
- Multiple issue *(superscalar)*
  - Replicate pipeline stages ⇒ 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

- Time

6 PM 7 8 9 10 11 12 1 2 AM
30 30 30 30 30

- Tasks

A (light clothing)
B (dark clothing)
C (very dirty clothing)
D (light clothing)
E (dark clothing)
F (very dirty clothing)

- Order

T a s k
O r d e r

- More resources, HW to match mix of parallel tasks?
Pipeline Depth and Issue Width

• Intel Processors over Time

<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

- Clock
- Power
- Pipeline Stages
- Issue width
- Cores


Values for Clock and Power are on a logarithmic scale.
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>
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
    - \texttt{add \$t0, \$s0, \$s1}
    - \texttt{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

```
Loop: lw $t0, 0($s1)  # $t0=array element
      addu $t0, $t0, $s2  # add scalar in $s2
      sw $t0, 0($s1)      # store result
      addi $s1, $s1, -4   # decrement pointer
      bne $s1, $zero, Loop # branch $s1!=0
```

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

- 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>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,-16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</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>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</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