The Midterm is Coming

• Midterm on May 6th.
• Midterm review on May 1th.
  • Come to class with questions.
• Midterm will cover everything before it
• It will includes
  • Questions similar to the homeworks
  • Questions about asking you to discuss high-level aspects of the papers we have read
  • Material from the book
• It will be challenging.
• It will be curved.
Implementing a MIPS Processor

Readings: 4.1-4.9
Pipelining Review
Our Hardware is Mostly Idle

Cycle time = 18 ns
Slowest module (alu) is ~6ns
Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
- Each stage can act on different instruction/data
- States/Control Signals of instructions are held in pipeline registers (latches)
Pipelining

cycle #1

cycle #2

cycle #3

cycle #4

cycle #5
Recap: Clock

- A hardware signal defines when data is valid and stable
  - Think about the clock in real life!
- We use edge-triggered clocking
  - Values stored in the sequential logic is updated only on a clock edge
The 5-Stage MIPS Pipeline

- Instruction Fetch
  - Read the instruction
- Decode
  - Figure out the incoming instruction?
  - Fetch the operands from the register file
- Execution: ALU
  - Perform ALU functions
- Memory access
  - Read/write data memory
- Write back results to registers
  - Write to register file
Pipelined datapath

**Instruction Fetch**
- PCSrc = Branch & Zero

**Instruction Decode**
- Read Address
- inst[31:0]

**Execution**
- Add
- 4

**Memory Access**
- MemWrite
- MemRead

**Write Back**
- MemtoReg

Will this work?
Pipelined datapath

- **Instruction Memory**
  - Read Address
  - Read inst[31:0]

- **Register File**
  - Read Reg 1
  - Read Reg 2
  - Read Data 1
  - Read Data 2
  - Write Reg

- **ALU**
  - Add
  - ALUop

- **Add**

- **Shift left 2**

- **Zero**

- **MemWrite**
  - Address
  - Read Data

- **MemtoReg**
  - Write Data

- **Data Memory**

- **PCSrc**

- **MemRead**

- **IF/ID**
  - add $1, $2, $3
  - lw $4, 0($5)
  - sub $6, $7, $8
  - sub $9, $10, $11
  - sw $1, 0($12)

- **ID/EX**
  - RegWrite
  - 16
  - sign-ext
d  - 32

- **EX/MEM**
  - RegDst
  - ALUSrc
  - Zero

- **MEM/WB**
  - MemWrite
Pipelined datapath

IF/ID

Add

ID/EX

Shift left 2

ALU

EX/MEM

Zero

ALUop

MEM/WB

MemWrite

MemRead

Data Memory

Read Data

Write Data
Pipelined datapath

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined datapath

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined datapath

Is this right?

add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
Pipelined datapath

- Instruction Memory
- Read Address inst[31:0]
- PC
- Add
- 4
- PCSrc
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB
- ALUop
- ALUSrc
- Zero
- Shift left 2
- Add
- RegWrite
- Inst[25:21]
- Inst[20:16]
- Inst[15:11]
Pipelined datapath + control
In Search of Instruction-level Parallelism

- Instruction level parallelism (ILP) lets multiple instructions execute *at the same time*.
- There’s a moderate amount of ILP in practice, but it is very valuable.
Approach 1: Widen the pipeline

- Process two instructions at once instead of 1
- 2-wide, in-order, superscalar processor
Dual issue: Structural Hazards

- Structural hazards
  - We might not replicate everything
  - Perhaps only one multiplier, one shifter, and one load/store unit
  - What if the instruction is in the wrong place?

If an “upper” instruction needs the “lower” pipeline, squash the “lower” instruction
Dual issue: Data Hazards

• The “lower” instruction may need a value produced by the “upper” instruction
• Forwarding cannot help us -- we must stall.
Approach 2: Out of Order

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t5,$t1,$t2
4: add $t3,$t1,$t2

We can parallelize instructions that do not have a “read-after-write” dependence (RAW)

There is parallelism!!
We can execute 1 & 2 at once and 3 & 4 at once
Data dependences

• In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.
• But beware:
  • Is there a dependence here?
    1: add $t1,$s2,$s3
    2: sub $t1,$s3,$s4
  • Can we reorder the instructions?
    2: sub $t1,$s3,$s4
    1: add $t1,$s2,$s3
  • Is the result the same?

No! The final value of $t1$ is different
False Dependence #1

• Also called “Write-after-Write” dependences (WAW) occur when two instructions write to the same value
• The dependence is “false” because no data flows between the instructions -- They just produce an output with the same name.
Beware again!

• Is there a dependence here?
  1: add $t1, $s2, $s3
  2: sub $s2, $s3, $s4

• Can we reorder the instructions?
  2: sub $s2, $s3, $s4
  1: add $t1, $s2, $s3

• Is the result the same?
  No! The value in $s2 that 1 needs will be destroyed
False Dependence #2

• This is a Write-after-Read (WAR) dependence
• Again, it is “false” because no data flows between the instructions
Out-of-Order Execution

• Any sequence of instructions has set of RAW, WAW, and WAR hazards that constrain its execution.
• Can we design a processor that extracts as much parallelism as possible, while still respecting these dependences?
The Central OOO Idea

1. Fetch a bunch of instructions
2. Build the dependence graph
3. Find all instructions with no unmet dependences
4. Execute them.
5. Repeat
Example

1: add $t1, $s2, $s3
2: sub $t2, $s3, $s4
3: or $t3, $t1, $t2
4: add $t5, $t1, $t2
5: or $t4, $s1, $s3
6: mul $t2, $t3, $s5
7: sl $t3, $t4, $t2
8: add $t3, $t5, $t1

8 Instructions in 5 cycles
Simplified OOO Pipeline

- A new “schedule” stage manages the “Instruction Window”
- The window holds the set of instruction the processor examines
  - The fetch and decode fill the window
  - Execute stage drains it
- Typically, OOO pipelines are also “wide” (i.e., they can execute multiple instructions at once) but it is not necessary.
The Instruction Window

• The “Instruction Window” is the set of instruction the processor examines
  • The fetch and decode fill the window
  • Execute stage drains it
• The larger the window, the more parallelism the processor can find, but...
• Keeping the window filled is a challenge
The Issue Window
The Issue Window
Keeping the Window Filled

• Keeping the instruction window filled is key!
• Instruction windows are about 32 instructions
  • (size is limited by their complexity, which is considerable)
• Branches are every 4-5 instructions.
• This means that the processor predict 6-8 consecutive branches correctly to keep the window full.
• On a mispredict, you flush the pipeline, which includes the emptying the window.
How Much Parallelism is There?

• Not much, in the presence of WAW and WAR dependences.
• These arise because we must reuse registers, and there are a limited number we can freely reuse.
• How can we get rid of them?
Removing False Dependences

- If WAW and WAR dependences arise because we have too few registers
  - Let’s add more!
- But! We can’t! The Architecture only gives us 32 (why or why did we only use 5 bits?)
- Solution:
  - Define a set of internal “physical” register that is as large as the number of instructions that can be “in flight” -- 128 in the latest intel chip.
  - Every instruction in the pipeline gets a registers
  - Maintaining a register mapping table that determines which physical register currently holds the value for the required “architectural” registers.
- This is called “Register Renaming”
1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

Register map table

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RAW → WAW → WAR
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

p1 currently holds the value of architectural registers r1

Register map table

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

1: Add  \( r_3, r_2, r_3 \)  
2: Sub  \( r_2, r_1, r_3 \)  
3: Mult  \( r_1, r_3, r_1 \)  
4: Add  \( r_2, r_3, r_1 \)  
5: Add  \( r_2, r_1, r_3 \)  

\[ \text{p4, p2, p3} \]  

\[
\begin{array}{ccc}
\text{r1} & \text{r2} & \text{r3} \\
p1 & p2 & p3 \\
1: & p1 & p2 & p4 \\
2: & & & \\
3: & & & \\
4: & & & \\
5: & & & \\
\end{array}
\]  

RAW  \quad \rightarrow \quad WAW  \quad \rightarrow \quad WAR
### Alpha 21264: Renaming

1. Add \( r3, r2, r3 \)
2. Sub \( r2, r1, r3 \)
3. Mult \( r1, r3, r1 \)
4. Add \( r2, r3, r1 \)
5. Add \( r2, r1, r3 \)

\[
\begin{array}{c}
p4, p2, p3 \\
p5, p1, p4
\end{array}
\]

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### RAW → WAW → WAR
Alpha 21264: Renaming

1: Add  r3, r2, r3  
2: Sub  r2, r1, r3  
3: Mult r1, r3, r1  
4: Add  r2, r3, r1  
5: Add  r2, r1, r3  

\[
\begin{array}{ccc}
1: & p4, & p2, & p3 \\
2: & p5, & p1, & p4 \\
3: & p6, & p4, & p1 \\
\end{array}
\]

\[
\begin{array}{ccc}
0: & p1 & p2 & p3 \\
1: & p1 & p2 & p4 \\
2: & p1 & p5 & p4 \\
3: & p6 & p5 & p4 \\
4: & & & \\
5: & & & \\
\end{array}
\]
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p7</td>
<td>p4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RAW → WAW → WAR
Alpha 21264: Renaming

1: Add  r3, r2, r3  \[ p4, p2, p3 \]
2: Sub  r2, r1, r3  \[ p5, p1, p4 \]
3: Mult r1, r3, r1  \[ p6, p4, p1 \]
4: Add  r2, r3, r1  \[ p7, p4, p6 \]
5: Add  r2, r1, r3  \[ p8, p6, p4 \]

\[
\begin{array}{|c|c|c|c|}
\hline
   & r1 & r2 & r3 \\
\hline
0: & p1 & p2 & p3 \\
1: & p1 & p2 & p4 \\
2: & p1 & p5 & p4 \\
3: & p6 & p5 & p4 \\
4: & p6 & p7 & p4 \\
5: & p6 & p8 & p4 \\
\hline
\end{array}
\]
1: Add r3, r2, r3  
2: Sub r2, r1, r3  
3: Mult r1, r3, r1  
4: Add r2, r3, r1  
5: Add r2, r1, r3

\[
\begin{array}{ccc}
\text{r1} & \text{r2} & \text{r3} \\
0: & p1 & p2 & p3 \\
1: & p1 & p2 & p4 \\
2: & p1 & p5 & p4 \\
3: & p6 & p5 & p4 \\
4: & p6 & p7 & p4 \\
5: & p6 & p8 & p4 \\
\end{array}
\]
1. Use symbols to represent the physical resources with the abbreviations for pipeline stages.
   1. IF, ID, EXE, MEM, WB

2. Horizontal axis represent the timeline, vertical axis for the instruction stream

3. Example:

   ```
   add $1, $2, $3
   lw $4, 0($5)
   sub $6, $7, $8
   sub $9, $10, $11
   sw $1, 0($12)
   ```
Figure 1 Data registers and transfer paths without CDB.