Pipelining the MIPS Datapath

Today & Next week

CBTF
SPim Bot

Done through Exam 4+5
Today’s lecture

- Pipeline implementation
  - Single-cycle Datapath
  - Pipelining performance
  - Pipelined datapath
  - Example
We have built a **single-cycle implementation** of a subset of the MIPS-based instruction set

- We have assumed that instructions execute in the same amount of time; this determines the clock cycle time.
- We have implemented the datapath and the control unit.
Refresher: The Full Single-cycle Datapath
We will use a simplified implementation of MIPS to create a pipelined version

Arithmetic: add sub and or slt

Data Transfer: lw sw

Control: beq
lw $t0, -4($sp)
beq $at, $0, offset

A) ALUSrc=0   B) ALUSrc=1
add $1, $2, $3

A) ALUSrc=0
   RegDst=0

B) ALUSrc=0
   RegDst=1

C) ALUSrc=1
   RegDst=0

D) ALUSrc=1
   RegDst=1
Worst-case delay from register-read to register-write determines clock speed

What is the worst-case delay for an instruction?

a) 6 ns  
b) 7 ns  
c) 8 ns  
d) 10 ns  
e) 11 ns
How fast can we clock a pipelined datapath?
Break datapath into 5 stages

IF
- lw $t0, 4($sp)
- lw $t1, 8($sp)
- lw $t2, 12($sp)
- lw $t3, 16($sp)
- lw $t4, 20($sp)

ID
- Read Instruction address [31-0]
- Instruction memory
- I [25 - 21]
- I [20 - 16]
- I [15 - 11]
- RegDst
- I [15 - 0]
- Read register 1
- Read register 2
- Write register
- Write data
- Read data 1
- Read data 2
- Write data
- Registers
- ALUOp
- ALUSrc
- Sign
- extend

EXE
- 0 Mux
- 1 Mux
- 0 Mux
- 1 Mux
- ALUZero
- Result
- MemWrite
- Write data
- Data memory
- MemRead
- MemToReg
- MemWrite
- Write data

MEM

WB

Clock cycle

1  2  3  4  5  6  7  8  9

lw $t0, 4($sp)
lw $t1, 8($sp)
lw $t2, 12($sp)
lw $t3, 16($sp)
lw $t4, 20($sp)
Single-cycle datapath rearranged to align with pipeline stages
Add pipeline registers in between stages

- There’s a lot of information to save, however. We’ll simplify our diagrams by drawing just one big pipeline register between each stage.
- The registers are named for the stages they connect.

  IF/ID       ID/EX       EX/MEM      MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done.
Paths from register-read to register-write are now shorter
Ideal pipeline performance is **time to fill the pipeline + one cycle per instruction**

- How long for \( N \) instructions on pipelined architecture?
  \[
  (4 + N) \cdot 2\text{ns} = 8 + 2N \text{ns}
  \]

- How long for \( N \) instructions on single-cycle (8ns clock period)?
  \[
  N \cdot 8\text{ns} = 8N \text{ns}
  \]

- How much faster is pipelining for \( N=1000 \) ?
  \[
  \frac{8 + 2 \cdot 1000}{8 \cdot 1000} \approx \frac{2010}{8000} = \frac{1}{4} \text{ns}
  \]
Pipelining improves throughput at the cost of increased latency.
Data values required in later stages must be **propagated forward** through the pipeline registers.

- **Example**
  - Destination register \((rd)\) is determined during the *first* stage (IF)
  - We store into the destination register during the *fifth* stage (WB)
  - \(rd\) must be passed through all of the pipeline stages
Note – We cannot keep values like destination register in the “instruction register”
Control signals are generated in the decode stage and are propagated across stages.

Group control signals based on the stages they are used in.
Categorize control signals by the pipeline stage that uses them

<table>
<thead>
<tr>
<th>Stage</th>
<th>Control signals needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>ALUSrc  ALUOp  RegDst  PCSrc</td>
</tr>
<tr>
<td>MEM</td>
<td>MemRead MemWrite</td>
</tr>
<tr>
<td>WB</td>
<td>RegWrite MemToReg</td>
</tr>
</tbody>
</table>
The pipeline registers and program counter update every clock cycle, so they do not have write enable controls.
An example execution sequence

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Base register</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>lw</td>
<td>$8, 4($29)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>sub</td>
<td>$2, $4, $5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>and</td>
<td>$9, $10, $11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>or</td>
<td>$16, $17, $18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1016</td>
<td>add</td>
<td>$13, $14, $0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASSUMPTIONS**

**CONVENTIONS**
- X indicates values that are not important, Example: Imm16 for R-type.
- Question marks ??? indicate values we do not know, usually resulting from instructions coming before and after the ones in our example.
Cycle 1 (filling)

IF: lw $8, 4($29):

ID: ???

EX: ???

MEM: ???

WB: ???

Read address
Instruction memory

Control

ALU Zero

Add

Shift left 2

ALUSrc (???)

RegWrite (?)

Read register 1
Read register 2
Write register
Write data

ALUOp (???)

RegDst (?)

ALUSrc (?)

ZeroALU

MemWrite (?)

MemRead (?)

MemToReg (?)

Data memory

MemWrite (?)

MemRead (?)

Add

0

1

Write data

Read data

PC

IF/ID

Add

Instruction address [31-0]

Instruction memory

MemWrite (?)

MemToReg (?)

1

0

P

C

Sign extend

0

1
Cycle 2

IF: sub $2, $4, $5
ID: lw $8, 4($29)

EX: ???
MEM: ???
WB: ???
Cycle 3

IF: and $9, $10, $11
ID: sub $2, $4, $5
EX: lw $8, 4($29)
MEM: ???
WB: ???

IF/ID
ID/EX
EX/MEM
MEM/WB

Control

PCSrc
Add
1012

1008

Read Instruction address [31-0]
Instruction memory

Add

Shift left 2

RegWrite (?)

Read register 1
Read data 1
Write register
Write data

alu
ALU Zero
Result

ALUSrc (add)

RegDest ()

Sign extend

PC

Add

Add

Add

ALU Zero
Result

MemToReg (?)

MemWrite (?)

MemRead (?)

MemToReg (?)

MEM: ???
WB: ???
Cycle 4

IF: or $16, $17, $18
ID: and $9, $10, $11
EX: sub $2, $4, $5
MEM: lw $8, 4($29)
WB: ???
Cycle 5 (full)

IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)
Cycle 7

IF: ???

ID: ???

EX: add $13, $14, $0

MEM: or $16, $17, $18

WB: and $9, $10, $11
Cycle 8

IF: ???
ID: ???
EX: ???
MEM: add $13, $14, $0
WB: or $16, $17, $18
Cycle 9

IF: ???

ID: ???

EX: ???

MEM: ???

WB: add $13, $14, $0
Some instructions do not require all five stages, can we skip stages?

R-type

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
</table>

Can we skip a stage?

a) yes
b) no
Some instructions do not require all five stages, can we skip stages?

- Example: R-type instructions only require 4 stages: IF, ID, EX, and WB
  - We don’t need the MEM stage
- What happens if we try to pipeline loads with R-type instructions?

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4(sp)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8(sp)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trying to use the single stage for multiple instructions creates a structural hazard

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File’s Write Port during its 5th stage
  - R-type uses Register File’s Write Port during its 4th stage

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Insert NOP stages to avoid structural hazards

- All instructions take 5 cycles with the same stages in the same order
  - Some stages will do nothing for some instructions

R-type

<table>
<thead>
<tr>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>NOP</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Stores and Branches have NOP stages, too...

Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>NOP</th>
</tr>
</thead>
</table>

Branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>NOP</th>
<th>NOP</th>
</tr>
</thead>
</table>
Things to notice from the last nine slides

- Instruction executions overlap
- Each functional unit is used by a different instruction in each cycle.
- In clock cycle 5, all of the hardware units are used (the pipeline is full). This is the ideal situation, and what makes pipelined processors so fast
- Similar example in the book available at the end of Section 6.3.
MIPs ISA makes pipelining “easy”

- Instruction formats are the same length and uniform
- Addressing modes are simple
- Each instruction takes only one cycle
Note how everything goes left to right, except …

Next time: We will discuss Data Hazards