Pipelining the MIPS Datapath
Today’s lecture

- Pipeline implementation
  - Single-cycle Datapath
  - Pipelining performance
  - Pipelined datapath
  - Example
Single-cycle implementation

- So far 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.
Full Machine Datapath – Lab 6
Single-cycle implementation

- For the following lectures, we will use a simpler implementation of the MIPS-based instruction set supporting just the following operations.

  Arithmetic:  
  Data Transfer:  
  Control:

  add  sub  and  or  slt
  lw  sw
  beq
Single-cycle datapath

lw $t0, -4($sp)
Single-cycle datapath

beq $at, $0, offset

A) ALUSrc=0  B) ALUSrc=1
Single-cycle datapath

\[ \text{add } $1, $2, $3 \]

A) ALUSrc=0 RegDst=0
B) ALUSrc=0 RegDst=1
C) ALUSrc=1 RegDst=0
D) ALUSrc=1 RegDst=1
Pipeline Motivation: Single-cycle datapath

- How long does it take to execute each instruction?
How fast can we clock a pipelined datapath?
Break datapath into 5 stages

1. **IF**
   - Read Instruction address [31-0]
   - Instruction memory

2. **ID**
   - Read register 1
   - Read register 2
   - Write register
   - Write data
   - RegWrite

3. **EXE**
   - ALU Zero
   - Result
   - ALUOp
   - ALUSrc

4. **MEM**
   - MemWrite
   - MemRead
   - MemToReg

5. **WB**
   - Write address
   - Data memory
   - Write data

---

**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>lw</td>
<td>$t0, 4($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t1, 8($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t2, 12($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t3, 16($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t4, 20($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelining Performance

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>lw</td>
<td>$t_0$, 4($sp$)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t_1$, 8($sp$)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t_2$, 12($sp$)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t_3$, 16($sp$)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>$t_4$, 20($sp$)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Execution time on ideal pipeline:**
  - time to fill the pipeline + one cycle per instruction
  - How long for N instructions?

- **Compare with other implementations:**
  - Single Cycle: (8ns clock period)

- **How much faster is pipelining for N=1000?**
Pipeline versus Single Cycle implementation

Latency versus Throughput

Single Cycle

Multiple Cycle
Pipelining other instruction types

- **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?

<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>
### Important Observation

- 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>Instruction</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>

Clock cycle
A solution: Insert NOP stages

- Enforce uniformity
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions

<table>
<thead>
<tr>
<th>R-type</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>NOP</th>
<th>WB</th>
</tr>
</thead>
</table>

Clock cycle

<table>
<thead>
<tr>
<th>add</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>NOP</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>or</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>WB</td>
</tr>
<tr>
<td>lw</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...

<table>
<thead>
<tr>
<th>store</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

March 13, 2013  Pipelining
Single-cycle datapath, slightly rearranged
Pipeline registers

- We’ll add intermediate registers to our pipelined datapath.
- 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.
Pipelined datapath

March 13, 2013

Pipelined datapath and control
Propagating values forward

- Any data values required in later stages must be propagated through the pipeline registers.
- The most extreme example is the destination register.
  - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn’t updated until the fifth stage (WB).
  - Thus, the rd field must be passed through all of the pipeline stages, as shown in red on the next slide.
- Notice that we can’t keep a single “instruction register,” because the pipelined machine needs to fetch a new instruction every clock cycle.
The destination register
What about control signals?

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.

- But just like before, some of the control signals will not be needed until some later stage and clock cycle.

- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.

- Control signals can be categorized by the pipeline stage that uses them.
Pipelined datapath and control

March 13, 2013

Pipelined datapath and control
What about control signals?

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.
- Control signals can be categorized 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</td>
</tr>
<tr>
<td>MEM</td>
<td>MemRead    MemWrite    PCSrc</td>
</tr>
<tr>
<td>WB</td>
<td>RegWrite    MemToReg</td>
</tr>
</tbody>
</table>
Pipelined datapath and control
Notes about the diagram

- The control signals are grouped together in the pipeline registers, just to make the diagram a little clearer.
- Not all of the registers have a write enable signal.
  - Because the datapath fetches one instruction per cycle, the PC must also be updated on each clock cycle. Including a write enable for the PC would be redundant.
  - Similarly, the pipeline registers are also written on every cycle, so no explicit write signals are needed.
An example execution sequence

- Here’s a sample sequence of instructions to execute.

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

- We’ll make some assumptions, just so we can show actual data values.
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - Every data memory location contains 99.

- Our pipeline diagrams will follow some conventions.
  - An X indicates values that aren’t important, like the constant field of an R-type instruction.
  - Question marks ??? indicate values we don’t know, usually resulting from instructions coming before and after the ones in our example.
Cycle 4

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

Read Instruction address [31-0]
Instruction memory

Control

Read register 1
Read register 2
Write register
Write data

RegWrite (?)

Shift left 2
Add

ALU Zero
Result

ALUOp (sub)

MemWrite (___)

MemRead (___)

MemToReg (?)

Add

Address

Data memory

Write data
Read data

???

???

???

???

???

???

???

???

???

???

???
Cycle 6 (emptying)

IF: ???
ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

Read address [31-0]
Instruction memory

Instruction

Read Instruction

Add

PCSr

IF/ID

ID/EX

EX/MEM

MEM/WB

Control

Shift left 2

Add

ALU Zero

Result

ALUOp (or)

Register 1

Register 2

Write register

Write data

Registers

Sign extend

RegWrite

RegDst (1)

MemWrite (0)

Address

Data memory

MemRead (0)

MemToReg

MemWrite (0)

RegWrite

RegWrite

Add

MemRead (0)

MemToReg

MemWrite (0)
Cycle 7

IF: ???

ID: ???

EX: add $13, $14, 0

MEM: or $16, $17, $18

WB: and $9, $10, $11

Read Instruction address [31-0]

Instruction memory

Read Instruction

Add

Shift left 2

RegWrite (1)

ALUSrc (0)

ALUOp (add)

ALU Zero

RegDst (1)

MemWrite (0)

MemRead (0)

MemToReg (0)

Write register

Read register 1

Read register 2

Write data 2

Write data 1

Read data 2

Read data

Write register

Registers

Sign extend

RegWrite (1)

MemWrite (0)

MemRead (0)

MemToReg (0)

Write data

Read data
Cycle 9

IF: ???

ID: ???

EX: ???

MEM: ???

WB: add $13, $14, $0

Read Instruction address [31-0]

Instruction memory

Control

RegWrite (1)

Shift left 2

ALU Op (???)

MemWrite (?)

MemToReg (0)

36
That’s a lot of diagrams there

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>$t0, 4(sp)</td>
<td>IF</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>ID</td>
</tr>
<tr>
<td>and $t1, $t2, $t3</td>
<td>EX</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>MEM</td>
</tr>
<tr>
<td>add $t5, $t6, $0</td>
<td>WB</td>
</tr>
</tbody>
</table>

Compare the last nine slides with the pipeline diagram above.

- You can see how instruction executions are overlapped.
- Each functional unit is used by a different instruction in each cycle.
- The pipeline registers save control and data values generated in previous clock cycles for later use.
- When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.

Try to understand this example or the similar one in the book at the end of Section 6.3.
Summary

- The **pipelined datapath** extends the single-cycle processor that we saw earlier to improve instruction throughput.
  - Instruction execution is split into several stages.
  - Multiple instructions flow through the pipeline simultaneously.
- **Pipeline registers** propagate data and control values to later stages.
- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.
- Next lecture, we’ll start talking about **Hazards**.
Note how everything goes left to right, except ...
Cycle 6 (emptying)

IF: ???
ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

March 13, 2013
Pipelined datapath and control