Reading

• Computer Organization and Design
  – Section 4.1, Processor Introduction (skim the hardware details – just get an idea of what the pieces are)
  – Section 4.5, Overview of Pipelining
Our Simplified Execution Model

• We have been working with a simple model of how instructions are executed:

\[
\text{do } \{
\quad \text{fetch instruction at Mem[PC];}
\quad \text{PC} = \text{PC} + 4; \quad \text{// advance to next instruction}
\quad \text{execute fetched instruction;}
\} \text{ while (processor not halted);}
\]

• In reality, instruction execution is broken into several steps, each of which is performed by separate hardware components
Actual Execution Cycle

- The classic MIPS implementation executes each instruction in 5 stages (typical of many processors)

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory
5. Write Back

- Each stage takes one clock cycle
IF and ID Stages

1. Instruction Fetch
   - Get the next instruction from memory
   - Increment Program Counter by 4

2. Instruction Decode
   - Figure out what the instruction says to do
   - Get values from the named registers
   - Simple instruction format means we know which registers we may need before the instruction is fully decoded
EX, MEM, and WB stages

3. Execute
   – On a memory reference, add up base and offset
   – On an arithmetic instruction, do the math

4. Memory Access
   – If load or store, access memory
   – Otherwise do nothing

5. Write back
   – Place the results in the appropriate register
Example: add $s0, $s1, $s2

- IF get instruction at PC from memory; increment PC

<table>
<thead>
<tr>
<th>op code</th>
<th>source 1</th>
<th>source 2</th>
<th>dest</th>
<th>shamt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

- ID determine what instruction is and read registers
  - 000000 with 100000 is the add instruction
  - get contents of $s1 and $s2 (example, suppose: $s1=7, $s2=12)
- EX add 7 and 12 = 19
- MEM do nothing for this instruction
- WB store 19 in register $s0
Example: lw $t2, 16($s0)

- IF get instruction at PC from memory; increment PC

<table>
<thead>
<tr>
<th>op code</th>
<th>base reg</th>
<th>src/dest</th>
<th>offset or immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>010111</td>
<td>10000</td>
<td>01000</td>
<td>00000000000010000</td>
</tr>
</tbody>
</table>

- ID determine what 010111 is
  - 010111 is lw
  - get contents of $s0 and $t2 (we don’t yet know that we don’t care about $t2). Let’s assume $s0=0x200D1C00 and $t2=77763

- EX add $16_{10}$ to 0x200D1C00 = 0x200D1C10
- MEM load the word stored at 0x200D1C10
- WB store loaded value in $t2
Instruction execution review

• Executing a MIPS instruction can take up to five steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IF</td>
<td>Read an instruction from memory.</td>
</tr>
<tr>
<td>Instruction Decode</td>
<td>ID</td>
<td>Read source registers and generate control signals.</td>
</tr>
<tr>
<td>Execute</td>
<td>EX</td>
<td>Compute an R-type result or a branch outcome.</td>
</tr>
<tr>
<td>Memory</td>
<td>MEM</td>
<td>Read or write the data memory.</td>
</tr>
<tr>
<td>Writeback</td>
<td>WB</td>
<td>Store a result in the destination register.</td>
</tr>
</tbody>
</table>

• However, as we saw, not all instructions need all five steps.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Steps required</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>IF ID EX</td>
</tr>
<tr>
<td>R-type</td>
<td>IF ID EX WB</td>
</tr>
<tr>
<td>sw</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>lw</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
A bunch of lazy functional units

- Notice that each execution step uses a different part of the hardware.
- In other words, the units are idle most of the time!
  - The instruction memory is read only when an instruction is fetched
  - Registers are read only during ID, and written during WB.
  - The ALU is used only in the middle of execution.
  - The data memory is idle most of the time.
- That’s a lot of hardware sitting around doing nothing.
Latency & Throughput

Latency – time it takes for an individual instruction to execute

What’s the latency for this implementation?
One instruction takes 5 clock cycles
Cycles per Instruction (CPI) = 5

Throughput – number of instructions executed per unit time

What’s the throughput of this implementation?
One instruction is completed every 5 clock cycles
Average CPI = 5
A relevant question

• Assuming you’ve got:
  – One washer (takes 30 minutes)
  – One drier (takes 40 minutes)
  – One “folder” (takes 20 minutes)

• It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  – How long does 4 loads take?
The slow way

- If each load is done sequentially it takes 6 hours
Laundry Pipelining

- Start each load as soon as possible
  - Overlap loads

- Pipelined laundry takes 3.5 hours
Pipelining Lessons

- Pipelining doesn’t help latency of single load, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
Pipelining

• Pipelining is a general-purpose efficiency technique
  – It is not specific to processors

• Pipelining is used in:
  – Assembly lines
  – Bucket brigades
  – Fast food restaurants

• Pipelining is used in other CS disciplines:
  – Networking
  – Server software architecture

• Useful to increase throughput in the presence of long latency
  – More on that later…
Pipelined Latency & Throughput

What’s the latency of this implementation?
What’s the throughput of this implementation?
Pipelined Analysis

• A pipeline with N stages could improve throughput by N times, but
  – each stage must take the same amount of time
  – each stage must always have work to do
  – there may be some overhead to implement
• Also, latency for each instruction may go up
  – Within some limits, we don’t care
MIPS ISA: Designed for Pipelining

• Instructions are all one length
  – simplifies Instruction Fetch stage
• Regular format
  – simplifies Instruction Decode
• Few memory operands, only registers
  – only lw and sw instructions access memory
• Aligned memory operands
  – only one memory access per operand
Making Pipelining Work

- We’ll make our pipeline 5 stages long
  - Stages are: IF, ID, EX, MEM, and WB
- We want to support executing 5 instructions simultaneously: one in each stage.
Pipelining Loads

| lw  | $t0, 4($sp) | IF | ID | EX | MEM | WB |
| lw  | $t1, 8($sp) | IF | ID | EX | MEM | WB |
| lw  | $t2, 12($sp)| IF | ID | EX | MEM | WB |
| lw  | $t3, 16($sp)| IF | ID | EX | MEM | WB |
| lw  | $t4, 20($sp)| IF | ID | EX | MEM | WB |

Clock cycle

Time

6 PM 7 8 9

30 40 40 40 20
A pipeline diagram shows the execution of a series of instructions.  
- The instruction sequence is shown vertically, from top to bottom.  
- Clock cycles are shown horizontally, from left to right.  
- Each instruction is divided into its component stages.  

This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.  
- The “lw” instruction is in its Execute stage.  
- Simultaneously, the “sub” is in its Instruction Decode stage.  
- Also, the “and” instruction is just being fetched.
**Pipeline terminology**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<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>sub $v0, $a0, $a1</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>and $t1, $t2, $t3</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>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $sp, $sp, -4</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>

- **The pipeline depth** is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is **filling**, since there are unused functional units.
- In cycle 5, the pipeline is **full**. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is **emptying**.
### 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><strong>lw</strong></td>
<td>$t0$, 4($sp)$</td>
<td><strong>IF</strong></td>
<td><strong>ID</strong></td>
<td><strong>EX</strong></td>
<td><strong>MEM</strong></td>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>$t1$, 8($sp)$</td>
<td><strong>IF</strong></td>
<td><strong>ID</strong></td>
<td><strong>EX</strong></td>
<td><strong>MEM</strong></td>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>$t2$, 12($sp)$</td>
<td><strong>IF</strong></td>
<td><strong>ID</strong></td>
<td><strong>EX</strong></td>
<td><strong>MEM</strong></td>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>$t3$, 16($sp)$</td>
<td><strong>IF</strong></td>
<td><strong>ID</strong></td>
<td><strong>EX</strong></td>
<td><strong>MEM</strong></td>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>$t4$, 20($sp)$</td>
<td><strong>IF</strong></td>
<td><strong>ID</strong></td>
<td><strong>EX</strong></td>
<td><strong>MEM</strong></td>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **filling**

• **Execution time on ideal pipeline:**
  - *time to fill the pipeline* + *one cycle per instruction*
  - N instructions -> 4 cycles + N cycles or (2N + 8) ns for 2ns clock period

• **How much faster is pipelining for N=1000 ?**
Pipeline Datapath: Resource Requirements

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t2, 12($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t3, 16($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t4, 20($sp)</td>
<td>IF</td>
</tr>
</tbody>
</table>

- We need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.
- Thus a pipelined processor duplicates hardware elements that are needed several times in the same clock 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>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>IF</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>IF</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. See the problem if:
  - Load writes to the register file during its 5th stage.
  - R-type writes to the register file during its 4th stage.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>add $sp, $sp, -4</td>
<td>IF</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>IF</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>IF</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>IF</td>
</tr>
</tbody>
</table>
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>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</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>
Summary

• Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions.
• Pipelining offers amazing speedup.
  – In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
• This is great if all of the instructions can execute independently of each other, but…
  – What happens when an instruction needs data produced by the previous one?
  – What happens if an instruction stage takes a long time?
    (Memory is a lot slower than the registers!)
  – Stay tuned…