Pipelining

Read: Chapter 4, Sections 4.5 to 4.8 (4th edition)

• Laundry example: washing (30 minutes), drying (30 minutes), folding (30 minutes), “stashing” (30 minutes).
• If only one person’s wash, it takes 2 hours to complete.
• If several folks need to do laundry, can do in 2 hours each — sequential solution:

• But, the washer, dryer, “folder”, and “stasher” are independent units.
**Pipeline basics:**

- Pipelined laundry takes 3.5 hours for four loads:

- Pipelining:
  - Does not help the *latency* of a single task — still takes 2 hours to do one person’s laundry.
  - Does help the *throughput* of the entire work load — 3.5 hours vs. 8 hours.
  - *Multiple* tasks operating simultaneously, each using different resources.
  - Potential *speedup* = number of pipe stages.
  - Rate limited by slowest pipeline stage.
  - Unbalanced lengths of pipe stages reduces speedup.
  - Time to “fill” pipeline and time to “drain” it reduces speedup.
Pipeline basics (continued):

- Consider the load word instruction:
  
  \[ \text{lw} \quad \$s0, \ 0(\$t0) \]

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFnetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Write</td>
</tr>
</tbody>
</table>

- **IFnetch**: Instruction Fetch: get the instruction from memory.
- **Reg/Dec**: Fetch values from Registers and Decode the instruction.
- **Exec**: Execute; calculate the memory address from which to load the word.
- **Mem**: Read the word from Memory.
- **Write**: Write the word to the Register.
Pipeline basics (continued):

- A more realistic picture: Not all cycles take the same amount of time:
  - Memory access is slower.
  - ALU computation is slower.
  - Register access is faster.
- Figure 4.26, page 333 (4th edition):

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruction fetch</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Data access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load word (lw)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
</tr>
<tr>
<td>Store word (sw)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>700 ps</td>
</tr>
<tr>
<td>R-format (add, sub, and, or, slt)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td>100 ps</td>
<td>600 ps</td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td></td>
<td>500 ps</td>
</tr>
</tbody>
</table>
Pipeline basics (continued):

- Can improve performance by increasing the instruction throughput:

- 3 load word ops, 2.4 nanoseconds:

- Becomes 3 load word ops, 1.4 nanoseconds:

  - Clock cycle time dependent on the slowest phases: 200 picoseconds in this case.
Single-cycle Implementation (continued):
Building a Pipelined datapath:

- Need to re-arrange some items from single-clock cycle implementation to split the data path into stages:

IF: Instruction fetch
ID: Instruction decode/register file read
EX: Execute or address calculation
MEM: Memory access
WB: Write back

CSc 252 — Computer Organization
Building a Pipelined datapath (continued):

- Add pipeline registers in-between each pipeline stage.

**IF:**
Instruction fetch

**ID:**
Instruction decode/register file read

**EX:**
Execute or address calculation

**MEM:**
Memory access

**WB:**
Write back
Building a Pipelined datapath (continued):

- How big is each pipeline register? How many bits are in each? (We’ll need more bits before we are done…)

![Diagram of a pipelined datapath with stages: IF (Instruction fetch), ID (Instruction decode/register file read), EX (Execute or address calculation), MEM (Memory access), WB (Write back). Each stage includes operations such as read data, address memory, and ALU operations.]
Building a Pipelined datapath (continued):

- **Problem:**

  - We know the number of the register to write on the 2nd clock cycle.
  - But, we do not have the data to write until the 5th clock cycle.
  - We need to “remember” the number of the register until the 5th clock cycle…

```assembly
and $t1, $t2, $t3  IF  ID  EX  MEM  WB
or $t4, $t7, $t2   IF  ID  EX  MEM  WB
slt $t5, $t5, $t7   IF  ID  EX  MEM  WB
add $t6, $t3, $t2   IF  ID  EX  MEM  WB
sub $t7, $t1, $t4   IF  ID  EX  MEM  WB
```
and or slt add sub

Building a Pipelined datapath (continued):

- The write register value is stored in the ID/EX register on cycle 2, then in EX/MEM on cycle 3, then in MEM/WB on cycle 4. The value is finally used on cycle 5.

```
and $t1, $t2, $t3 IF ID EX MEM WB
or $t4, $t7, $t2 IF ID EX MEM WB
slt $t5, $t5, $t7 IF ID EX MEM WB
add $t6, $t3, $t2 IF ID EX MEM WB
sub $t7, $t1, $t4 IF ID EX MEM WB
```
Building a Pipelined datapath (continued):

• What makes pipelining easy?
  • All instructions are the same length.
  • Only a few instruction formats (MIPS uses three: R-type, I-type, J-type)
  • Memory operands appear only in loads and stores.

• What makes it hard?
  • Structural hazards: suppose we have only one memory.
  • Data hazards: an instruction depends on a previous instruction.
  • Control hazards: need to worry about branch instructions.

• We’ll build a simple pipeline and look at (some of) these issues.
• (Time permitting) We’ll talk about modern processors and what really makes it hard:
  • Exception handling.
  • Trying to improve performance with out-of-order execution, etc.
**Representing Pipelines:**

- Simplified drawing to enable us to talk about multiple instructions executing in sequence.

- Example: the add command:

```
IF   ID    EX       MEM    WB
```

- Full shading indicates combinational unit that is active on that stage:

- The shading on the right side indicates the unit is being READ on that clock cycle: ID
  - Reads occur at the **end** of the clock cycle.

- The shading on the left side indicates the unit is being WRITTEN on that clock cycle: WB
  - Writes occur at the **beginning** of the clock cycle.

- No shading indicates the unit is not being used on that clock cycle: MEM

```
IF  ID  EX  MEM  WB
```

```
IF  ID  EX  MEM  WB
```
Representing Pipelines (continued):

- Can help with answering questions such as:
  
  - How many clock cycles does it take to execute this code?
  - What is the ALU doing during clock cycle 4? What else is happening during clock cycle 4?
  - Can use this representation to help understand datapaths through the CPU.

```
  lw $10, 20($1)           IF   ID    EX    MEM   WB
  sub $11,$2,$3            IF   ID    EX    MEM   WB
  sw $12,28($4)            IF   ID    EX    MEM   WB
```
**Pipeline Control:**

- Control wires are (more or less) the same ones we used before (as per the single-clock cycle implementation).

The `lw` instruction uses bits 20-16.

The arithmetic (\texttt{add}, \texttt{sub}, \texttt{and}, \texttt{or}, \texttt{slt}) instructions use bits 15-11.
Pipeline Control (continued):

- A specific example: the number of the register to which the result is written.
  - The `lw` instruction uses bits 20-16.
  - The arithmetic (`add`, `sub`, `and`, `or`, `slt`) instructions use bits 15-11.
- We do not know which set of bits to use until the end of the second clock cycle.
  - Therefore, the control wire cannot be turned on or off until the third clock cycle.
  - The multiplexor has to be in the third clock cycle portion of the CPU.
Pipeline Control (continued):

- The second clock cycle has the Control unit. It gets the opcode from wires 31-26.
- Control turns the RegDst control wire on or off.
- Store the RegDst control wire in the ID/EX pipeline register for use during the 3rd clock cycle.
Pipe[10]line Control (continued):

- Other control wires for 3rd clock cycle:
  - ALUSrc (1 wire)
  - ALUOp (2 wires)
  - Both from single clock cycle implementation.

How many bits are now in the ID/EX pipeline register?
Pipeline Control (continued):

- 4th clock cycle control wires:
  - Branch, MemRead, MemWrite.
Pipeline Control (continued):

How many bits in the EX/MEM pipeline register?
Pipeline Control (continued):

- 5th clock cycle control wires:
  - MemtoReg, RegWrite.
Pipeline Control (continued):

- 5th clock cycle control wires:
  - MemtoReg, RegWrite.
  
- $\text{and } t_1, t_2, t_3$
- $\text{or } t_4, t_7, t_2$
- $\text{slt } t_5, t_5, t_7$
- $\text{add } t_6, t_3, t_2$
- $\text{sub } t_7, t_1, t_4$

CSc 252 — Computer Organization
Pipeline Control (continued):

- How many bits?
Pipeline Control (continued):

- What are the settings needed for each control line on each stage of the pipeline?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation 3rd Stage</th>
<th>Memory Access 4th Stage</th>
<th>Write-back 5th Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RegDst</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Only need to look at stages 3, 4, and 5.
- Why not 1 and 2?
- X = “does not care”
- Taken from the single-clock cycle, but re-arranged to fit the stages for pipelining.
Data Hazards and Forwarding:

- Problems can arise when the next instruction is started before the current one is finished.
- Dependencies that “go backward in time” are data hazards.

Time flows down: program execution order

- `sub $2, $1, $3`  
- `and $12,$2,$5`  
- `or $13,$5,$2`  
- `add $14,$2,$2`  
- `sw $15,100($2)`
Data Hazards and Forwarding (continued):

- Solution: All assembly languages have a “no operation” instruction.

- `nop` in MIPS.

- The CPU does nothing on a `nop` operation.

- Have the compiler insert `nop`’s as needed so the CPU “waits” for the value to be written into $2.

- Where do we insert the `nop`’s?

  - `sub    $2, $1, $3    ` IF ID EX MEM WB
  - `nop                      ` IF ID EX MEM WB
  - `nop                          ` IF ID EX MEM WB
  - `and   $12, $2, $5              ` IF ID EX MEM WB
  - `or    $13, $6, $2                    ` IF ID EX MEM WB
  - `add   $14, $2, $2                        ` IF ID EX MEM WB
  - `sw    $15, 100($2)                   ` IF ID EX MEM WB

- What is wrong with this approach?
Data Hazards and Forwarding (continued):

- The value we need is present in a pipeline register — Use it instead of waiting.
- Register file forwarding to handle read/write to same register:

<table>
<thead>
<tr>
<th></th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg $2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

There are three cases to consider:
- Value needed is in the register
- Value needed is in MEM/WB
- Value needed is in EX/MEM
Data Hazards and Forwarding (continued):

- Write data
- Read register 1
- Write register
- Read data 1
- Read register 2
- Read data 2
- Read Address
- Instruction memory
- Instruction [31-0]
- PC

Forwarding unit:
- Tells the register computed on the previous clock cycle
- Tells the register computed two clock cycles ago

Tells the 2 registers needed for ALU on this clock cycle
Data Hazards and Forwarding (continued):
Data Hazards and Forwarding (continued):

3 inputs:
- Value from Registers
- Value from EX/MEM
- Value from MEM/WB

2 pairs of control wires carry results from Forwarding unit

30
**Data Hazards — Stalls:**

- Load word can still cause a hazard — try to read register following a load instruction that writes that register.

```
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```
Data Hazards — Stalls (continued):

- Stall the pipeline by keeping an instruction in the same stage.

lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
Data Hazards — Stalls (continued):

- Detecting the data hazard.

lw $2, 20($1)
and $4, $2, $5
or $5, $4, $1

Registers (two) needed by current instruction

MemRead, to determine if memory will be read

MemWrite
MemtoReg
Branch
ALUOp
ALUSrc
RegDst
RegWrite

Forwarding unit

CSc 252 — Computer Organization
Data Hazards — Stalls (continued):

Add Control wire: Determine if PC does or does not change

Add control wire: Determine if new instruction is written to IF/ID

Add Multiplexor: What control wires are sent to later cycles

CSc 252 — Computer Organization
Data Hazards — Stalls (continued):

Choose: PCWrite On
Change PC for next instruction

Choose: IF/ID.Write On
Write instruction in IF/ID

Choose: Control wire values to be passed to later cycles

No stall necessary!
Data Hazards — Stalls (continued):

Choose: PCWrite Off
PC stays the same to re-read instruction

Choose: IF/ID.Write Off
Leave previous instruction in IF/ID

Choose: All control wires turned off for stall

Stall is needed!
Branch Hazards:

• Pages 339-343 of Section 4.5 and Section 4.8 (4th edition)

• For beq, the current design:
  • 2nd clock cycle: Fetches the registers and decodes the beq.
  • 3rd clock cycle:
    • **ALU** subtracts to determine if two registers are equal
    • Special-purpose **add** computes branch address
  • 4th clock cycle: If equal, branch occurs.

• Consider:
Branch Hazards (continued):

- The **and**, **or**, **add** instructions are already started by the time the **beq** changes the PC.

40: `beq $1, $3, 7`

44: `and $12, $2, $5`

48: `or $13, $6, $2`

52: `add $14, $2, $2`

72: `lw $4, 50($7)`
Branch Hazards (continued):

[Diagram of computer pipeline stages: IF, ID, EX, MEM, WB]

- **PC**: Program Counter
- **Instruction memory**: Source of instructions
- **Instruction [31-0]**: The instruction's address
- **Mux 0**: Multiplexor for select
- **Mux 1**: Multiplexor for another select
- **aluOp**: ALU operation (e.g., add, subtract)
- **aluSrc**: ALU source (e.g., register, immediate)
- **RegWrite**: Register write enable
- **MemRead**: Memory read enable
- **MemWrite**: Memory write enable

Flow of operations:
- **Read Address**: PC
- **Instruction**: From memory
- **Read register 1/2**: From registers
- **Write register**: Into registers
- **Read data 1/2**: From memory
- **Shift left 2**: ALU operation
- **Addition/Subtraction**: ALU operation
- **Sign extend**: Data preprocessing
- **Zero**: Result check
- **Branch**: Control flow
- **data**: Result of ALU operation
- **Write data**: Result written to memory
- **MemRead/MemWrite**: Data transfer
- **RegWrite/MemtoReg**: Register modification
- **MemtoReg**: Data transfer to memory

This diagram illustrates the pipeline stages of a computer processor, detailing how instructions are fetched, decoded, executed, and stored.
Branch Hazards (continued):

- Need to “predict” whether the `beq` is true or false.
  - Is this just a 50/50 guess?
  - Sometimes not:
    - ??

- Still, the prediction will not be correct all the time.
  - Need to reduce the loss when `beq` is true.
Branch Hazards (continued):

- Can we make the `beq` “faster” — use fewer clock cycles?

Move address computation to 2nd cycle.
Branch Hazards (continued):

- Compute the address while getting values from the registers.
Branch Hazards (continued):

- How to compare the two register contents?
  - Not possible to put another ALU into the second clock cycle. Why?
  - Need a faster way to compare:
Branch Hazards (continued):

- Branch moved to 2nd clock cycle:
Branch Hazards (continued):

40: `beq $1, $3, 7`

44: `and $12, $2, $5`

72: `lw $4, 50($7)`

```
add    $9, $7, $8
sub    $4, $7, $1
beq    $1, $3, ...
slt    $5, $7, $9
changes to  
add    $9, $7, $8
beq    $1, $3, ...
```

```
add    $9, $7, $8
beq    $1, $3, ...
sub    $4, $7, $1
slt    $5, $7, $9
```
Branch Hazards (continued):

- There are still problems:
Branch Hazards (continued):

• Rather than always assume “branch not taken”:
  • Remember what happened the last time the branch was executed.
  • Simple technique:
    • Store the PC for the branch and a single bit that indicates what happened the last time.
    • But, this can lead to problems:
      • Consider a loop executed 10 times.
      • Only correct 80% of the time. Why?
Branch Hazards (continued):

- Better approach: “Remember” two bits of information: (Figure 4.63, page 381)
  
  - Deterministic Finite State (DFS) Diagram:
Summary:

- Improving Performance:
  - Try and avoid stalls — re-order instructions when possible.
  - Can be done in hardware, or by the compiler.
    - Often by both!
  - Branch prediction to reduce problems with branch hazards.
  - Superscalar: start more than one instruction in the same cycle.

- Dynamic Scheduling:
  - Hardware tries to find instructions to execute.
  - Out-of-order execution is possible.
  - Speculative execution and dynamic branch prediction.

- You have the background you need to learn more.
  - And, there is more to learn about this!
Controlling Hardware:

- Reading: Appendix D, Sections D.1 and D.2.
- The MIPS ALU has 4 control wires and 6 operations that it performs.
  - MIPS defines NOR, which was not used by the subset we covered.
  - NOR requires the extra bit; the left-most Operation bit shown below
- Figure 4.12, page 317.

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Funct field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>0</td>
<td>load word</td>
<td>X X X X X X</td>
<td>00 10</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>store word</td>
<td>X X X X X X</td>
<td>00 10</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>branch equal</td>
<td>X X X X X X</td>
<td>01 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>add</td>
<td>1 0 0 0 0 0</td>
<td>00 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>subtract</td>
<td>1 0 0 0 1 0</td>
<td>01 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>AND</td>
<td>1 0 0 1 0 0</td>
<td>00 00</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>OR</td>
<td>1 0 0 1 0 1</td>
<td>00 01</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>set less than</td>
<td>1 0 1 0 1 0</td>
<td>01 11</td>
</tr>
</tbody>
</table>
Controlling Hardware (continued):

- When is **operation** a 1?

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Function code fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>load word</td>
<td>0</td>
<td>0</td>
<td>00 10</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>store word</td>
<td>0</td>
<td>0</td>
<td>00 10</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>branch equal</td>
<td>1</td>
<td>0</td>
<td>01 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
<td>1</td>
<td>0</td>
<td>00 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>subtract</td>
<td>1</td>
<td>0</td>
<td>01 10</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>00 00</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>OR</td>
<td>1</td>
<td>0</td>
<td>00 01</td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>set less than</td>
<td>1</td>
<td>0</td>
<td>01 11</td>
</tr>
</tbody>
</table>