Topic 9: Pipelining

- The Concept
- The 5 Pipeline Phases (of MIPS)
- Pipeline Registers
- Modeling pipelines
- Control Bits
Questions:

Laundry: 30 minutes per step

- How long per load (total)?
- How many loads per hour?

- What is the washing machine doing while the dryer runs?
Questions:

- How long per load?
- How many loads per hour?
- How much work is done on the time step circled?
Pipelining is like an assembly line.

Lots of specialized “workers.” Lots of partially-completed objects.
WARNING WARNING WARNING

I'm about to talk about a pipelined Ripple-Carry adder. In the Real World, nobody builds adders like this; they are too critical for performance.

But it's a nice illustration of the concept.
Ripple Carry

One Clock Tick

Add one bit
Add one bit
Add one bit
Ripple Carry

Question:

Suppose that a 1-bit adder takes $A$ picoseconds to run.

To allow for a 32-bit Ripple Carry adder, how long must the clock cycles be?
Ripple Carry

Question:

Suppose that a 1-bit adder takes $A$ picoseconds to run.

To allow for a 32-bit Ripple Carry adder, how long must the clock cycles be?

- Each clock cycle must be at least $32 \times A$ picoseconds long.
All of the hardware is running all the time.

But not all is doing useful work!

Adders early in the chain have *already finished*, and their outputs will never change.

Adders late in the chain *do not have their inputs yet*, and so are not doing useful work yet.
Multi-Cycle Ripple Carry

IDEA:

Store adder results in temporary registers, in-between the 1-bit adders. Only advance the add by 1 bit per clock cycle.
Ripple Carry

Clock 1

Pipeline Register

Clock 2

Pipeline Register

Clock 3

Add one bit

Add one bit

Add one bit
Multi-Cycle Ripple Carry

Group Exercise:

Suppose that a 1-bit adder takes A picoseconds to run.

If we construct a 32-bit adder to only calculate one bit per clock cycle:

- How fast can our clock cycles be?
- How many clock cycles will an add take to complete?
- What is the total time to complete a single add?
Multi-Cycle Ripple Carry

Group Exercise:
Suppose that a 1-bit adder takes time $A$ to run.

- The clock can tick once per $A$ picoseconds
- 32-bit add takes 32 cycles
- Total time: $32 \times A$
  - Same as the single-cycle version!
Ripple Carry

This new circuit is not any faster than the old one, we still ripple.

But the clock is now very very fast – it only has to wait for a single adder!

Also, we have clear boundaries – most hardware is idle.
Pipelined Ripple Carry

**IDEA:**

Use idle adders to work on other addition operations at the same time.
Pipelined Ripple Carry

**Group Exercise:**

To make better use of hardware, we've pipelined the add operations. Each operation uses one of the 1-bit adders at a time.

- **Latency:** How long does it take to complete a single add?
- **Throughput:** How many adds finish per cycle?

- If a 1-bit adder takes $A$ picoseconds to complete, how long between adds?
- How long was it in the single-cycle Ripple Carry?
Pipelined Ripple Carry

Group Exercise:

Multiple add operations at once.

- **Latency**: Each add takes 32 cycles to complete
- **Throughput**: One add completes per cycle

- An add finishes every $A$ picoseconds.
- In a single-cycle Ripple Carry, an add finished every $32 \times A$ picoseconds.
Which of these operations started first?

Which is oldest? Newest?

Which comes first in the program? Last?
Group Exercise:

In our Pipelined Ripple Carry:

- What does each pipeline register need to store?
Multi-Cycle Ripple Carry

Group Exercise:

Pipelined Ripple Carry: what's in the registers?

- Store inputs
- Store output
- Store temporary values (carry bits)
Ripple Carry, Pipelined

- Pipeline register mix changes over time
  - Start with all inputs
  - End with all outputs
  - Store temporaries as needed
Pipelining Overview

KEY CONCEPT:

- We'd like to keep all of the hardware busy, at all times
- When a hardware component is done with an instruction, it should start the next immediately
Pipelining Overview

KEY CONCEPT:

- To keep all hardware busy, we need **multiple operations** going at the same time.
- Break them into “**phases**” to keep them separated.
Pipelining Overview

KEY CONCEPT:

• The clock can be as fast as the slowest pipeline phase.
  – **Much** faster than a single-cycle design!

• We finish one operation per clock cycle.
Pipelining Overview

KEY CONCEPT:

• Pipelining does not change the latency – but it can vastly improve the throughput.

• The max speedup is equal to the # of phases.
  - Actual speedup will be lower, because not all phases take the same time.
Pipelining Overview

KEY CONCEPT:

- In a pipeline, the instruction at the beginning is the **last instruction**.
- The oldest (that is, first) instruction is the one that has gone the farthest in the pipeline.
Questions:
Consider the circled time (basically, a clock cycle!)

- Which load started first?
- Which will end first?
- Which load is in the dryer?
MIPS 5-Stage Pipeline

IF → ID → EX → MEM → WB

Instruction Fetch  Instruction Decode  Execute  Memory  Writeback
MIPS 5-Stage Pipeline

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

Only blue lines go backwards...
MIPS 5-Stage Pipeline

![MIPS 5-Stage Pipeline Diagram](image-url)
MIPS 5-Stage Pipeline

- Each operation takes 5 cycles
- Do all operations do useful work in all stages?
- Can we “skip over” stages we don't need?
## Different Instructions, Different Tasks

### Group Exercise:
What stages are needed by which instructions?

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Inst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Regs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do Calc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Load word (**lw**)
- Store word (**sw**)
- R-format (**add, sub, and, or, slt**)
- Branch (**beq**)

---
Different Instructions, Different Tasks

**Group Exercise:**
What stages are needed by which instructions?

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>IF Get Inst</th>
<th>ID Read Regs</th>
<th>EX Do Calc</th>
<th>MEM Load/Store</th>
<th>WB Store Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load word (lw)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Store word (sw)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R-format (add, sub, and, or, slt)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question:
Is it possible for an instruction to “skip over” a stage to save time?
To answer this, let's animate some instructions.
To answer this, let's animate some instructions.
To answer this, let's animate some instructions.
To answer this, let's animate some instructions.
Instructions **cannot** skip over phases, because other instruction are already using the other pieces of hardware.
**Group Exercise:**

- How long would each instruction take, without pipelining?
- What would the clock speed be, without pipelining?
- How long does each phase take?
- What is the clock speed of the pipelined design?
- How does this affect the latency of each instruction?

<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></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></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>200 ps</td>
<td>100 ps</td>
<td></td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td>100 ps</td>
<td>100 ps</td>
</tr>
</tbody>
</table>
Group Exercise:
- How long would each instruction take, without pipelining?
- What would the clock speed be, without pipelining?
- How long does each phase take?
- What is the clock speed of the pipelined design?
- How does this affect the latency of each instruction?

<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>
Without pipelining, loads take 800 ps
Clock speed = 800 ps
3 loads take 2.4 ns

Group Exercise:
How does pipelining affect this?
- Clock speed
- # of Cycles
- Total time for 3 loads
- 200 ns clock speed  
  was: 800 ns
- 5 clocks per load
- Finish in 7 clocks (1.4 ns)  
  was: 2.4 ns
Pipeline Registers

Group Exercise:
In the next slide, we'll look at the 5-stage MIPS pipeline again.

For each pipeline register, how many bits are required?

Remember, you have to store anything that will be used later:
• Inputs that haven't been used yet
• Outputs that can't be delivered yet
• Temporaries
MIPS 5-Stage Pipeline

IF: Instruction fetch
    - Read Address
    - Instruction [31-0]
    - Instruction memory
    - Mux

ID: Instruction decode/register file read
    - Read register 1
    - Read data 1
    - Read register 2
    - Write register
    - Write data
    - Mux

EX: Execute or address calculation
    - Shift left 2
    - Sum Add
    - Mux

MEM: Memory access
    - Address
    - Read data
    - Write data

WB: Write back
    - Mux

Registers

PC

Data memory

Sign extend 32

IF/ID:
PC+4  (32)
Instruction  (32)
ID/EX:
PC+4 (32)
Read Data 1 (32)
Read Data 2 (32)
Imm32 (32)

(Instruction is NOT saved here. Why?)
EX/MEM:
- Branch Dest (32)
- ALU Result (32)
- Zero (1)
- Read Data 2 (32)
Important Details We Missed

• Things we forgot to account for:
  – Destination register field
  – Control Bits
    (and more, later in the slides)
The Destination Register

- Chosen in ID
- Not used until WB
- Write Data, Write Reg, RegWrite basically “travel back in time”
Solution:
• Save in pipeline registers until needed
• True for all control bits! (ID → whenever)
Modeling Pipelining

- It's often useful to model instructions as they pass through the pipeline.
- Horizontal axis is time.
- Shading is used to indicate what sort of operation is ongoing.
  - Right/left is read/write
Group Exercise:
Analyze this picture. What can we deduce about this instruction? Is it clearly one instruction? Are there any that it cannot be?
Modeling Pipelining

This instruction:
- Uses the ALU
- Doesn't read or write memory
- Writes back to a register

Could be any R-format, and most I-format. Cannot be lw, sw, beq/bne, or j.
Group Exercise:
Identify the physical components used in each pipeline phase.

Is there any circumstance where two instructions might use the same component in the clock?
Modeling Pipelining

IF: PC, Instruction Memory
ID: Registers
EX: ALU
MEM: Main Memory
WB: Registers
MIPS Solution:
This is OK – in fact, it's normal.
The WB happens **before** the ID, in the same clock cycle.
**Data Hazards:**

Instruction 1 writes before Instruction 4 reads.

But Instructions 2 & 3 read too early.

We'll fix this in the next slide deck.
Control Bits in a Pipeline

- Where to calculate control?
- Could copy instruction down the line
  - Takes up too much space
- Better to copy the control bits
What is the appropriate phase for Control?

- Needs to read instruction, so can't be IF
- Want to be early as possible
What is the appropriate phase for Control?

- Control happens in ID ("instruction decode")
- In parallel with register reads
**EXAMPLE:** RegDst

- Control bit calculated in ID
- Used in EX
- Saved until WB
EXAMPLE: RegDst

- How many new bits did we add to ID/EX?
- EX/MEM? MEM/WB?
EXAMPLE: RegDst

- 2x5+1 new bits in ID/EX
- 5 new bits in EX/MEM and MEM/WB
Group Exercise:
What other control bits are used in the EX phase?

How much space do they consume?
- ALUsrc
- ALUop
Question:
What bits are used in MEM? EX?
Pipelining Overview - Summary

- Single-cycle hardware is \textit{wasteful} because components sit idle
- Pipelined hardware has many \textit{phases} working in parallel

- Pipelining doesn't improve \textit{latency}
  - Often makes it worse!
- Pipelining improves \textit{throughput}
  - But it takes a few cycles to fill the pipeline
Pipelining Overview - Summary

- Clock speed is speed of **slowest phase**
  - Other phases will wait around

- Every clock tick, every phase performs 1 action
- Many instructions make progress every clock

- Every tick, one instruction completes
  - After the pipeline is full
Pipelining Overview - Summary

- Control bits are calculated in ID
- Control bits are saved in pipeline registers until they are needed

- Pipeline registers store:
  - Inputs not used yet
  - Outputs not delivered yet
  - Temporaries (instruction, control bits, etc.)
Pipelining Overview - Summary

- Pictures allow us to model active components, track dependencies
- WB happens before ID