CSE 410
Computer Systems

Hal Perkins

Spring 2010

Lecture 9 – Pipelining I

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:

```
do {
    fetch instruction at Mem[PC];
    PC = PC + 4;  // advance to next instruction
    execute fetched instruction;
} 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)

$$\begin{tikzcd}\boxed{\texttt{IF}} \arrow[r] & \boxed{\texttt{ID}} \arrow[r] & \boxed{\texttt{EX}} \arrow[r] & \boxed{\texttt{MEM}} \arrow[r] & \boxed{\texttt{WB}}\end{tikzcd}$$

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

op code	source 1	source 2	dest	shamt	function
000000	10001	10010	10000	00000	100000

- 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

op code	base reg	src/dest	offset or immediate value
010111	10000	01000	0000000000010000

- 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.

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

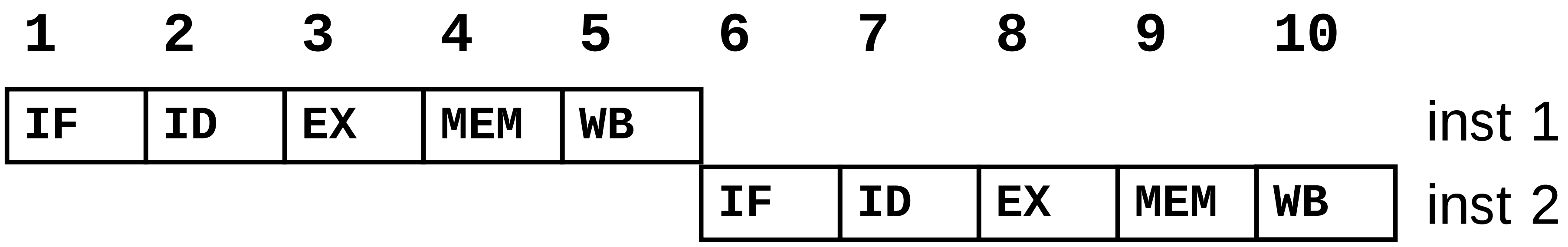
- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

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

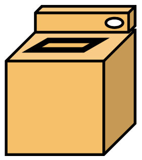
What's the throughput of this implementation?

One instruction is completed every 5 clock cycles

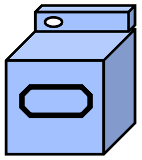
Average CPI = 5

A relevant question

- Assuming you’ve got:
  - One washer (takes 30 minutes)



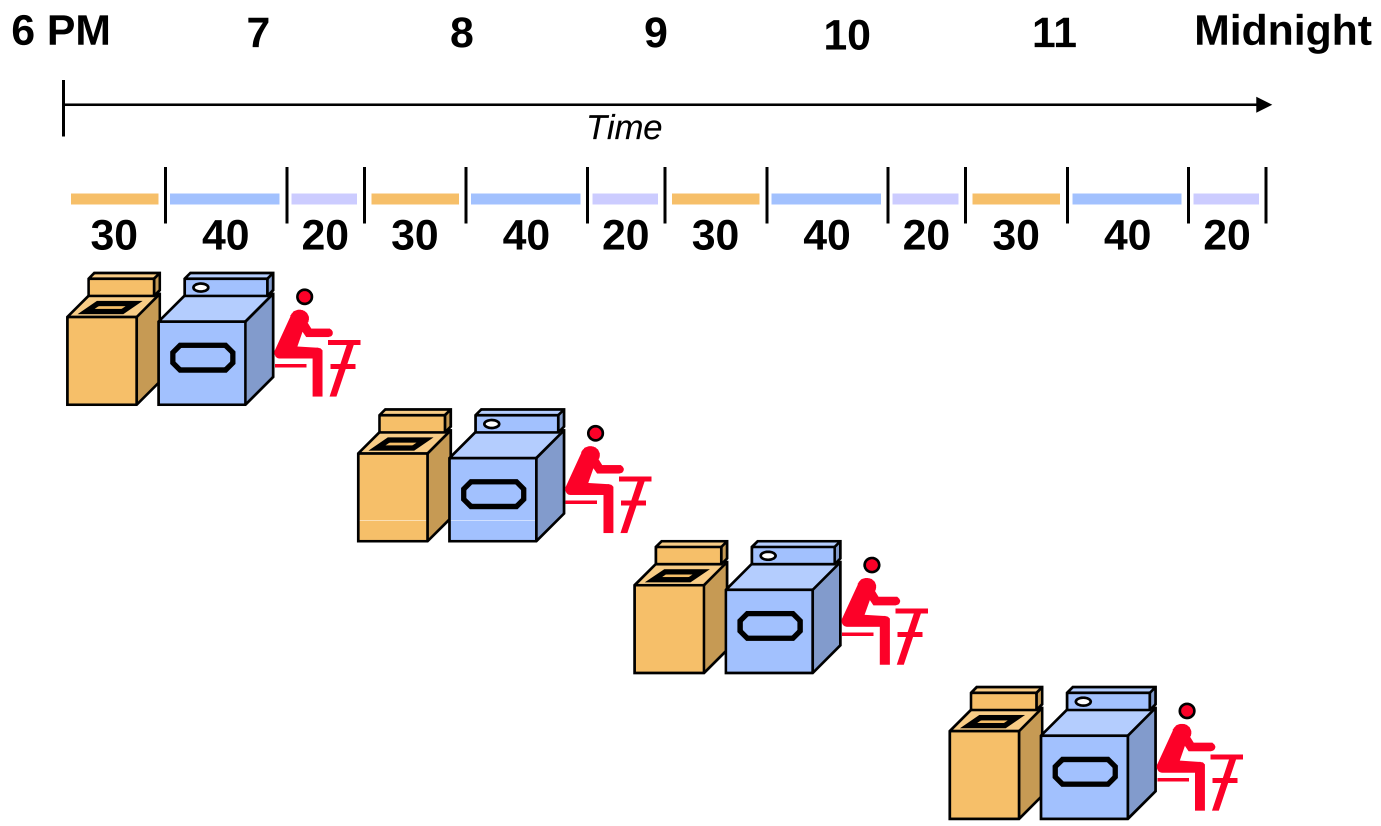
  - One drier (takes 40 minutes)



  - One “folder” (takes 20 minutes)

- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  - How long does 4 loads take?

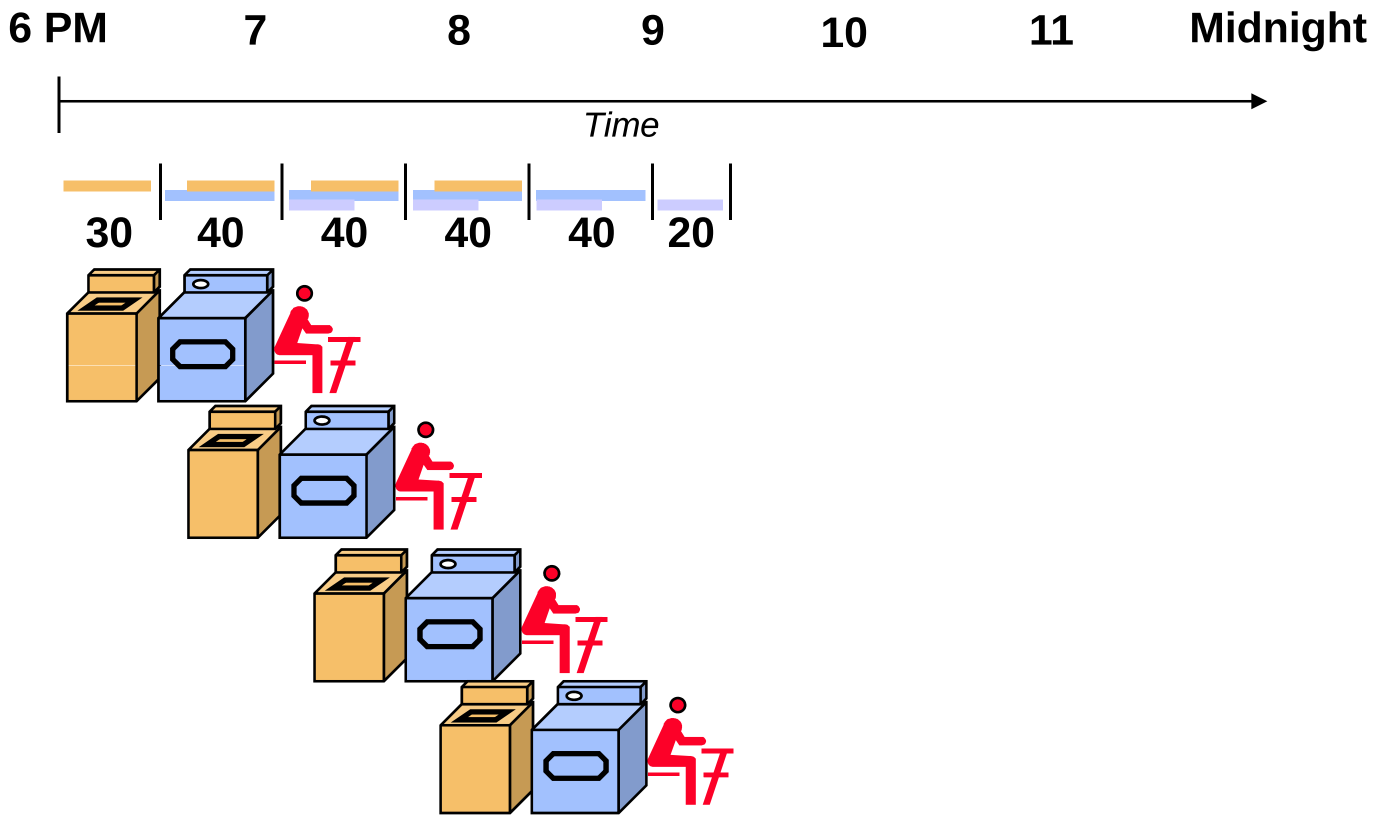
The slow way



- If each load is done sequentially it takes 6 hours

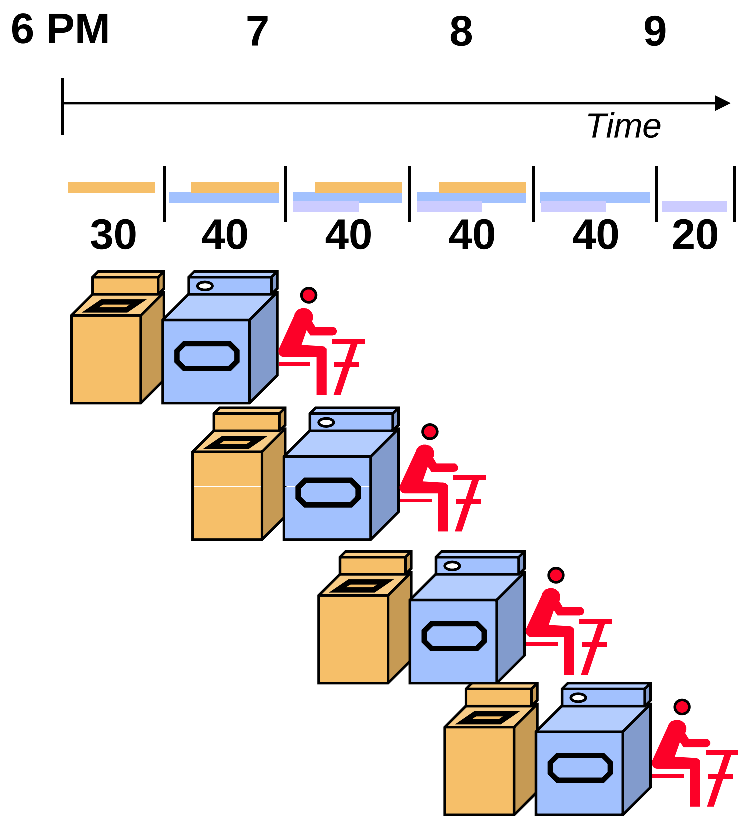
Laundry Pipelining

- 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

1	2	3	4	5	6	7	8	9	
IF	ID	EX	MEM	WB					inst 1
	IF	ID	EX	MEM	WB				inst 2
		IF	ID	EX	MEM	WB			inst 3
			IF	ID	EX	MEM	WB		inst 4
				IF	ID	EX	MEM	WB	inst 5

- 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

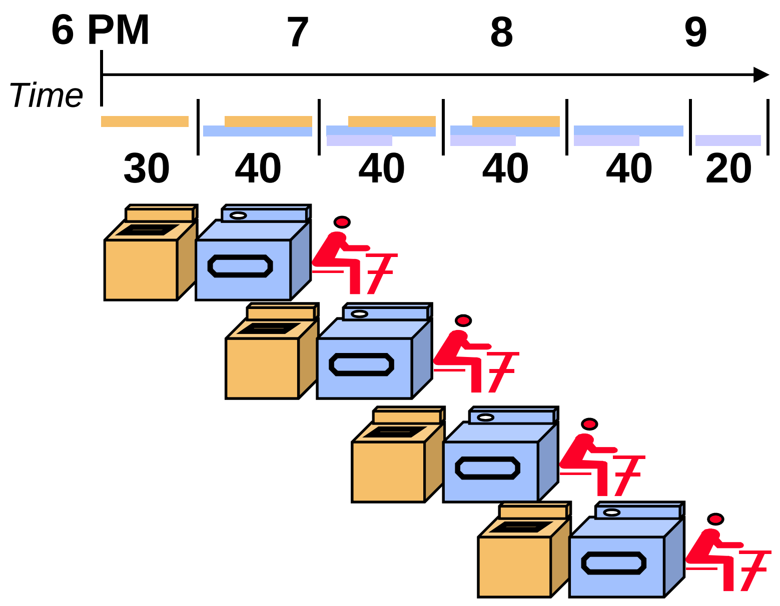
Making Pipelining Work

- 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

Clock cycle

		1	2	3	4	5	6	7	8	9
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



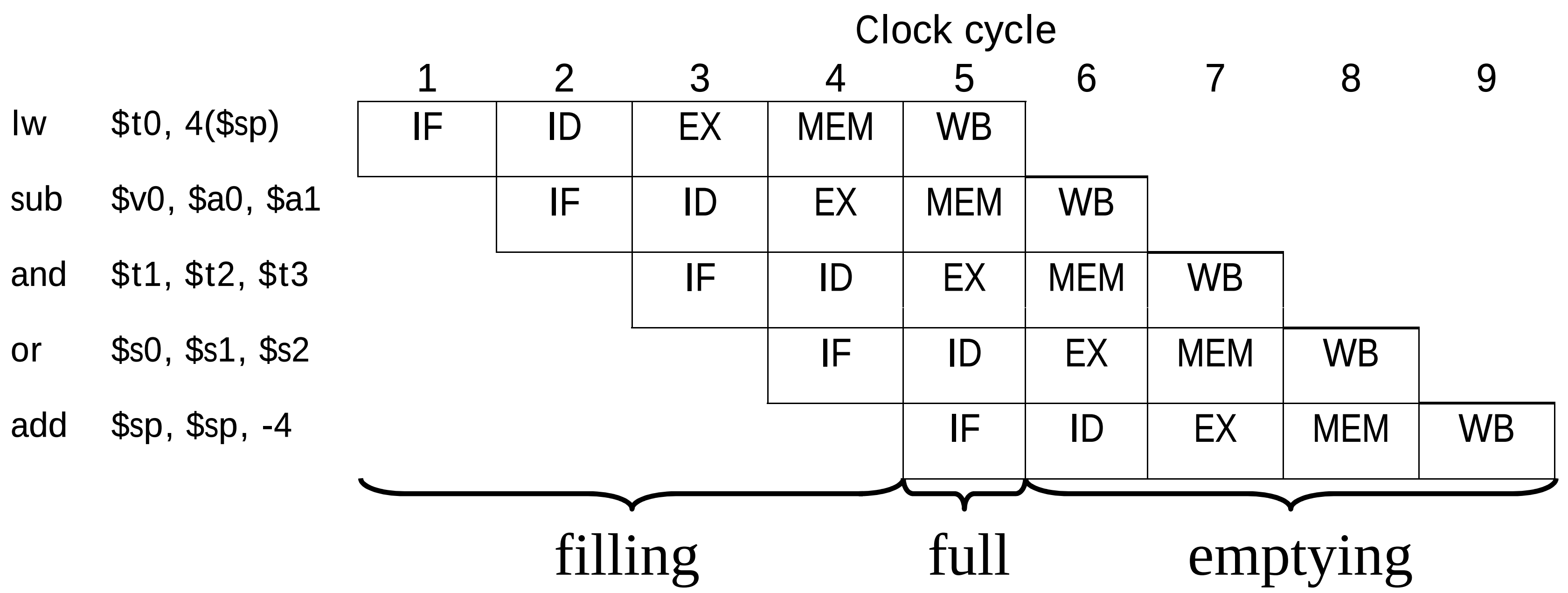
A pipeline diagram

Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

- 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



		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

- 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

	1	2	3	4	5	6	7	8	9
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

$\underbrace{\hspace{6em}}_{\text{filling}}$

- 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

Clock cycle

		1	2	3	4	5	6	7	8	9
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

- 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?

Clock cycle

		1	2	3	4	5	6	7	8	9
add	$sp, $sp, -4	IF	ID	EX	WB					
sub	$v0, $a0, $a1		IF	ID	EX	WB				
lw	$t0, 4($sp)			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	WB		
lw	$t1, 8($sp)					IF	ID	EX	MEM	WB

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

						Clock cycle				
		1	2	3	4	5	6	7	8	9
add	\$sp, \$sp, -4	IF	ID	EX	WB					
sub	\$v0, \$a0, \$a1		IF	ID	EX	WB				
lw	\$t0, 4(\$sp)			IF	ID	EX	MEM	WB		
or	\$s0, \$s1, \$s2				IF	ID	EX	WB		
lw	\$t1, 8(\$sp)					IF	ID	EX	MEM	WB

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

R-type	IF	ID	EX	NOP	WB

Clock cycle

		1	2	3	4	5	6	7	8	9
add	$sp, $sp, -4	IF	ID	EX	NOP	WB				
sub	$v0, $a0, $a1		IF	ID	EX	NOP	WB			
lw	$t0, 4($sp)			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	NOP	WB	
lw	$t1, 8($sp)					IF	ID	EX	MEM	WB

- Stores and Branches have NOP stages, too…

store	IF	ID	EX	MEM	NOP
branch	IF	ID	EX	NOP	NOP

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…