

# Pipelining

CSE 410 - Computer Systems  
October 17, 2001

## Readings and References

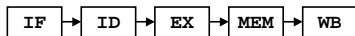
- Reading
  - Sections 6.1 through 6.3, Patterson and Hennessy, Computer Organization & Design
- Other References

17-Oct-2001

CSE 410 - Pipelining

2

## Execution Cycle



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

17-Oct-2001

CSE 410 - Pipelining

3

## IF and ID Stages

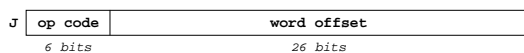
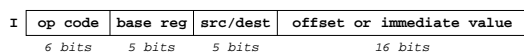
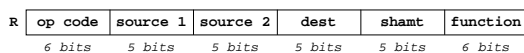
1. Instruction Fetch
  - Get the next instruction from memory
  - Increment Program Counter value by 4
2. Instruction Decode
  - Figure out what the instruction says to do
  - Get values from the named registers
  - Simple instruction format means we know which registers we may need before the instruction is fully decoded

17-Oct-2001

CSE 410 - Pipelining

4

## Simple MIPS Instruction Formats



17-Oct-2001

CSE 410 - Pipelining

5

## 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
  - If branch, replace PC with destination address
  - Otherwise do nothing
5. Write back
  - Place the results in the appropriate register

17-Oct-2001

CSE 410 - Pipelining

6

## Example: add \$s0, \$s1, \$s2

- **IF** get instruction at PC from memory

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 (eg: \$s1=7, \$s2=12)
- **EX** add 7 and 12 = 19
- **MEM** do nothing for this instruction
- **WB** store 19 in register \$s0

17-Oct-2001

CSE 410 - Pipelining

7

## Example: lw \$t2, 16(\$s0)

- **IF** get instruction at PC from memory

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 know that we don't care about \$t2) \$s0=0x200D1C00, \$t2=77763
- **EX** add 16 to 0x200D1C00 = 0x200D1C10
- **MEM** load the word stored at 0x200D1C10
- **WB** store loaded value in \$t2

17-Oct-2001

CSE 410 - Pipelining

8

## Latency & Throughput

1	2	3	4	5	6	7	8	9	10	
IF	ID	EX	MEM	WB						inst 1
					IF	ID	EX	MEM	WB	inst 2

- **Latency**—the time it takes for an individual instruction to execute
  - What's the latency for this implementation?
- **Throughput**—the number of instructions that execute per unit time
  - What's the throughput of this implementation?

17-Oct-2001

CSE 410 - Pipelining

9

## A case for pipelining

- If non-overlapped, the functional units are underutilized because each unit is used only once every five cycles
- If Instruction Set Architecture is carefully designed, organization of the functional units can be arranged so that they execute in parallel
- **Pipelining** overlaps the stages of execution so every stage has something to do each cycle

17-Oct-2001

CSE 410 - Pipelining

10

## 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 throughput of this implementation?
- What's the latency of this implementation?

17-Oct-2001

CSE 410 - Pipelining

11

## Pipelined Analysis

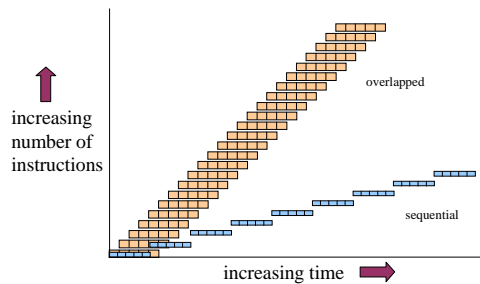
- A pipeline with N stages could improve throughput by N times, but
  - each stage must take the same amount of time
  - each stage must always have work to do
  - there may be some overhead to implement
- Also, latency for each instruction may go up
  - Within some limits, we don't care

17-Oct-2001

CSE 410 - Pipelining

12

## Throughput is good!



17-Oct-2001

CSE 410 - Pipelining

13

## MIPS ISA: Born to Pipeline

- Instructions 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

17-Oct-2001

CSE 410 - Pipelining

14

## Memory accesses

- Efficient pipeline requires each stage to take about the same amount of time
- CPU is much faster than memory hardware
- Cache is provided on chip
  - i-cache holds instructions
  - d-cache holds data
  - critical feature for successful RISC pipeline
  - more about caches next week

17-Oct-2001

CSE 410 - Pipelining

15

## The Hazards of Parallel Activity

- Any time you get several things going at once, you run the risk of interactions and dependencies
  - juggling doesn't take kindly to irregular events
- Unwinding activities after they have started can be very costly in terms of performance
  - drop everything on the floor and start over

17-Oct-2001

CSE 410 - Pipelining

16

## Design for Speed

- Most of what we talk about next relates to the CPU hardware itself
  - problems keeping a pipeline full
  - solutions that are used in the MIPS design
- Some programmer visible effects remain
  - many are hidden by the assembler or compiler
  - the code that you write tells what you want done, but the tools rearrange it for speed

17-Oct-2001

CSE 410 - Pipelining

17

## Pipeline Hazards

- Structural hazards
  - Instructions in different stages need the same resource, eg, memory
- Data hazards
  - data not available to perform next operation
- Control hazards
  - data not available to make branch decision

17-Oct-2001

CSE 410 - Pipelining

18

## Structural Hazards

- Concurrent instructions want same resource
  - `lw` instruction in stage four (memory access)
  - `add` instruction in stage one (instruction fetch)
  - Both of these actions require access to memory; they would collide if not designed for
- Add more hardware to eliminate problem
  - separate instruction and data caches
- Or stall (cheaper & easier), not usually done

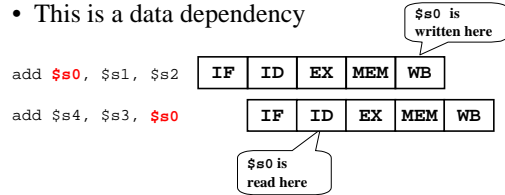
17-Oct-2001

CSE 410 - Pipelining

19

## Data Hazards

- When an instruction depends on the results of a previous instruction still in the pipeline
- This is a data dependency



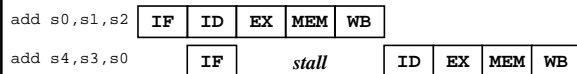
17-Oct-2001

CSE 410 - Pipelining

20

## Stall for register data dependency

- Stall the pipeline until the result is available
  - this would create a 3-cycle *pipeline bubble*



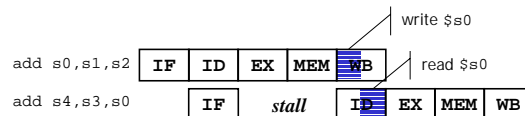
17-Oct-2001

CSE 410 - Pipelining

21

## Read & Write in same Cycle

- Write the register in the first part of the clock cycle
- Read it in the second part of the clock cycle
- A 2-cycle stall is still required



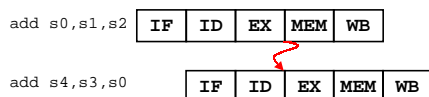
17-Oct-2001

CSE 410 - Pipelining

22

## Solution: Forwarding

- The value of \$s0 is known internally after cycle 3 (after the first instruction's EX stage)
- The value of \$s0 isn't needed until cycle 4 (before the second instruction's EX stage)
- If we **forward** the result there isn't a stall



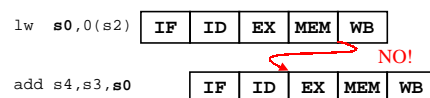
17-Oct-2001

CSE 410 - Pipelining

23

## Another data hazard

- What if the first instruction is `lw`?
- `s0` isn't known until after the MEM stage
  - We can't forward back into the past
- Either **stall** or **reorder** instructions



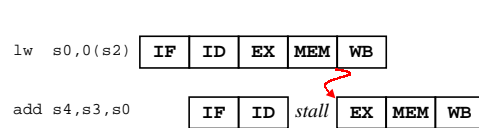
17-Oct-2001

CSE 410 - Pipelining

24

## Stall for **lw** hazard

- We can stall for one cycle, but we hate to stall



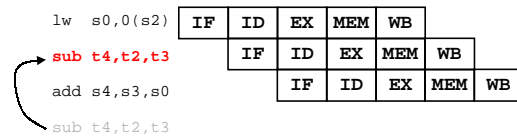
17-Oct-2001

CSE 410 - Pipelining

25

## Instruction Reorder for **lw** hazard

- Try to execute an unrelated instruction between the two instructions



17-Oct-2001

CSE 410 - Pipelining

26

## Reordering Instructions

- Reordering instructions is a common technique for avoiding pipeline stalls
- Static reordering
  - programmer, compiler and assembler do this
- Dynamic reordering
  - modern processors can see several instructions
  - they execute any that have no dependency
  - this is known as *out-of-order execution* and is complicated to implement

17-Oct-2001

CSE 410 - Pipelining

27