Computer Architecture
Faculty of Computer Science & Engineering - HCMUT

Chapter 4: The Processor

Binh Tran-Thanh

thanhbinh@hcmut.edu.vn

This chapter contents

- The basic units in the CPU
- Functions of the major components in the CPU
- Instruction execution at hardware level
- Performance and trace off among CPUs

This chapter outcomes

Students who complete this course will be able to:

- Explain the structure of a computer system and deeply understand how it works at the hardware level.
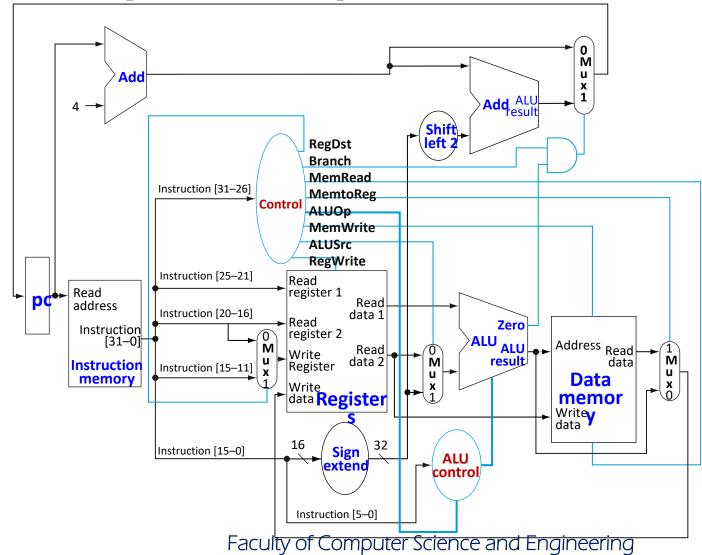
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
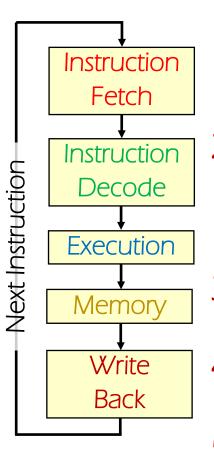 - Control transfer: `beq`, `j`

The *simplified* processor.

MIPS Instruction Execution Cycle

1. **Instruction Fetch**:
   - Get instruction from memory
   - Address is in Program Counter (PC) Register

2. **Instruction Decode**:
   - Find out the operation required and control signals
   - Get operand(s) needed for operation

3. **Execution**:
   - Perform the required operation

4. **Memory**:
   - Access memory (load/store)

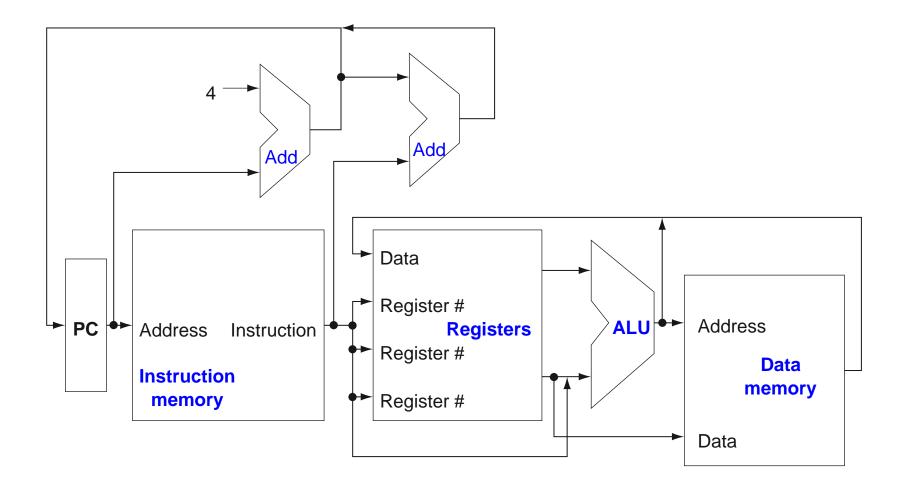5. **Write Back**:
   - Store the result of the operation

```
Next Instruction
  ┌──────────────┐
  │  Instruction │
  │    Fetch     │
  └──────────────┘
  ┌──────────────┐
  │  Instruction │
  │    Decode    │
  └──────────────┘
  ┌──────────────┐
  │  Execution   │
  └──────────────┘
  ┌──────────────┐
  │   Memory     │
  └──────────────┘
  ┌──────────────┐
  │    Write     │
  │    Back      │
  └──────────────┘
```

Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
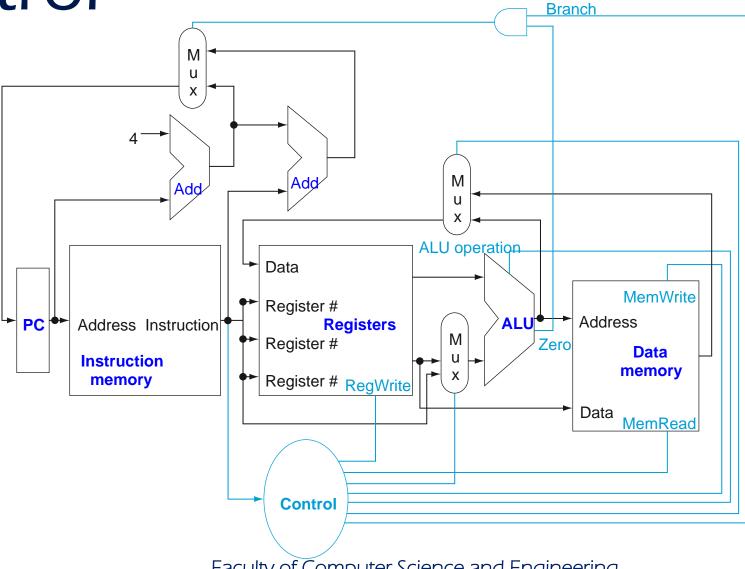 - PC $\leftarrow$ target address or PC + 4

CPU Overview

Multiplexers

- **Can't just join wires together**
  - **Use multiplexers**

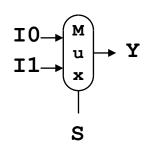# Control

Logic Design Basics

- **Information encoded in binary**
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- **Combinational element**
  - Operate on data
  - Output is a function of input
- **State (sequential) elements**
  - Store information
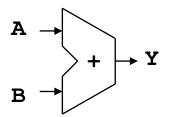
Combinational Elements

- ## AND-gate
  - **Y = A & B**

- ## Adder
  - **Y = A + B**

- ## Multiplexer
  - **Y = S ? I1: I0**

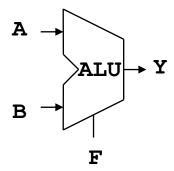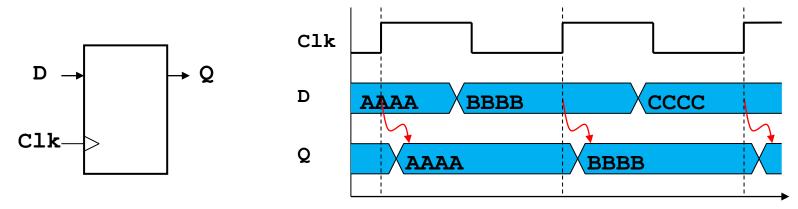- ## Arithmetic/Logic Unit
  - **Y = F(A, B)**
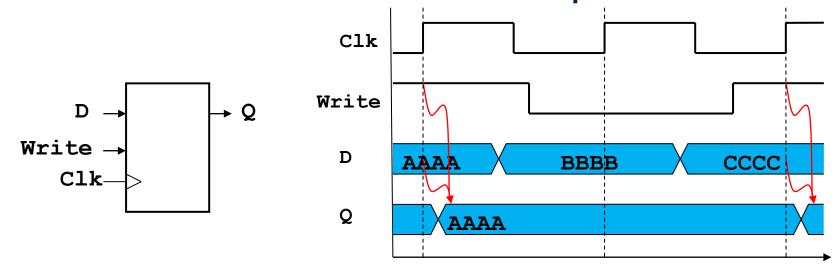
Sequential Elements

- *Register: stores data in a circuit*
  - *Uses a clock signal to determine when to update the stored value*
  - *Edge-triggered: update when Clk changes from 0 to 1*
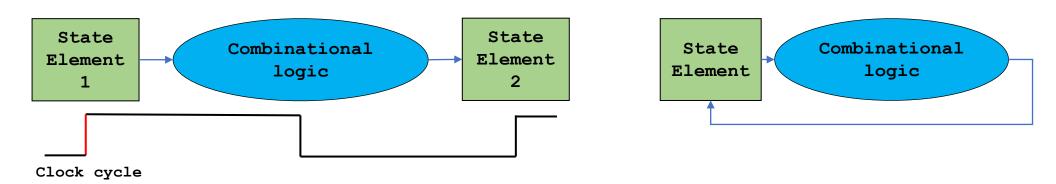
Sequential Elements

- ## Register with write control
  - ### Only updates on clock edge when write control input is 1
  - ### Used when stored value is required later

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
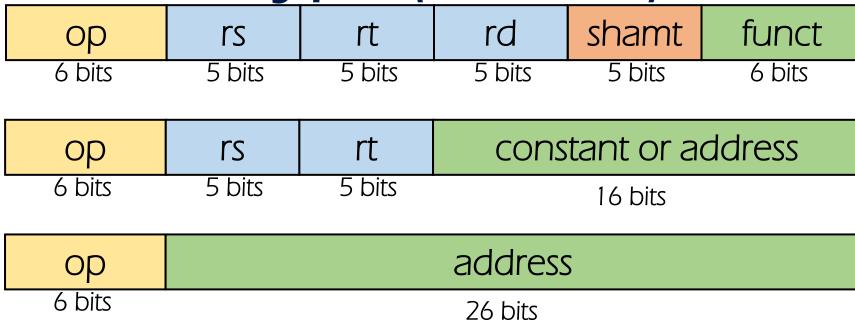 - Longest delay determines clock period

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, ...
- We will build a MIPS data-path incrementally
  - Refining the overview design

Instruction type (review)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

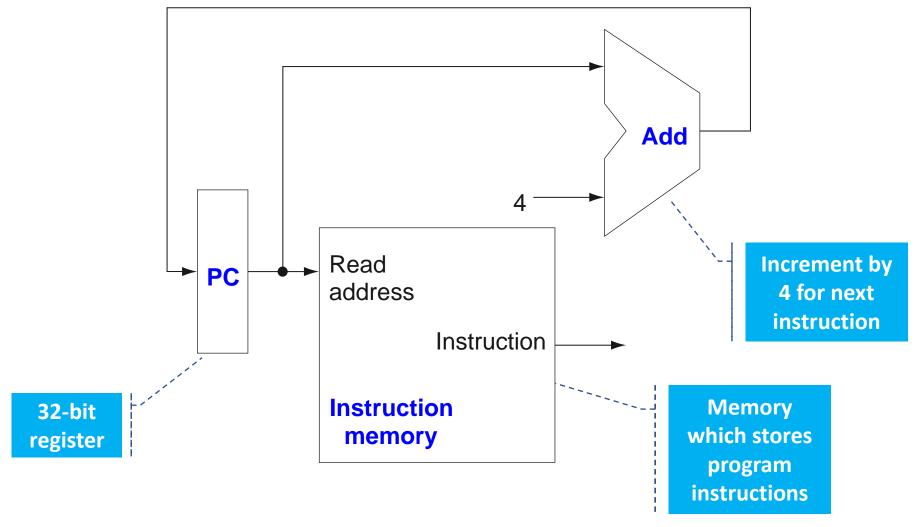| op	address
6 bits	26 bits

- **Note:**
  - All MIPS instructions are 32-bit wise
  - All MIPS instructions contain 6-bit OP (most significant)
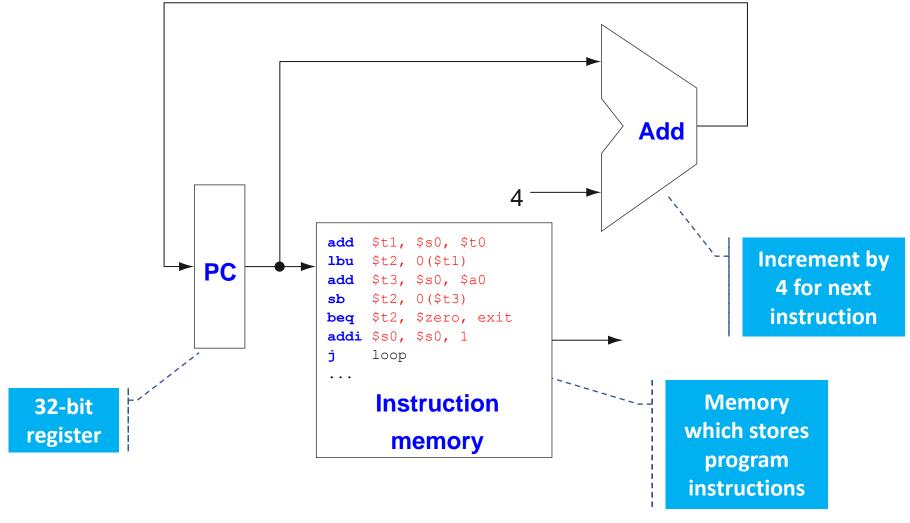
Instruction Fetch

Instruction Fetch (cont.)


```
add  $t1, $s0, $t0
lbu  $t2, 0($t1)
add  $t3, $s0, $a0
sb   $t2, 0($t3)
beq  $t2, $zero, exit
addi $s0, $s0, 1
j    loop
...
```

PC

Add

4

Instruction memory

Increment by 4 for next instruction

Memory which stores program instructions

32-bit register

R-Format Instructions
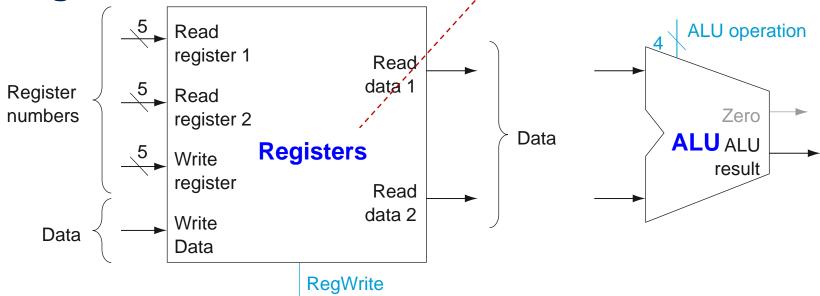
- Read two register operands
- Perform arithmetic/logical operation
- Write register result

Collection of 32 registers, aka register file
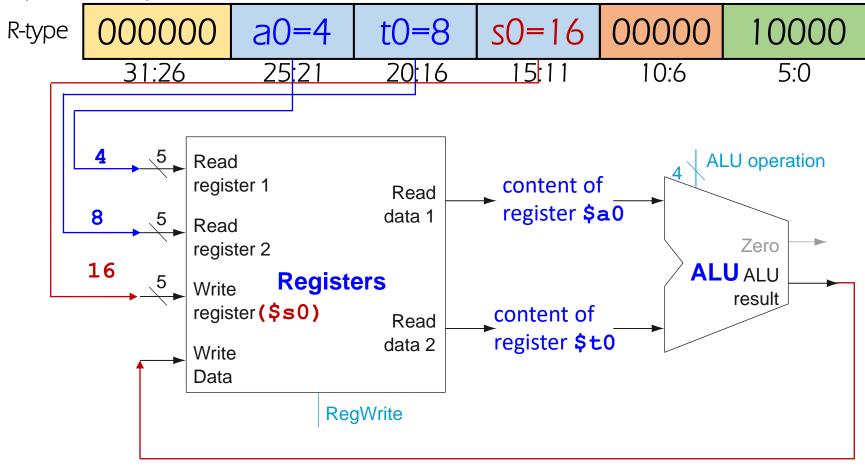

a. Registers

b. ALU

R-Format Instructions (example)

`add $S0, $a0, $t0`

R-type	000000	a0=4	t0=8	s0=16	00000	10000
	31:26	25:21	20:16	15:11	10:6	5:0

Your turn

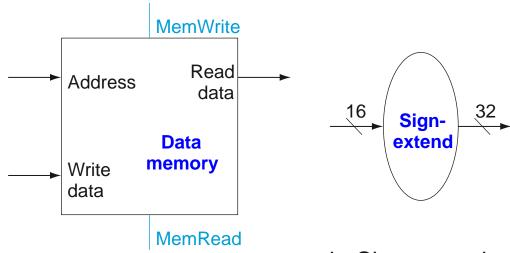- Assume: $4 = 104, $5 = 105, ... , $31 = 131

- What is the value of read data 1 if we assign 6 to read register 1?

- What is the value of read data 2 if read register 2 = 12?

- If write register = 10 and Write data = 12

  - Which register is written and
  - What value is that (in case RegWrite = 0/1)?

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
    - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

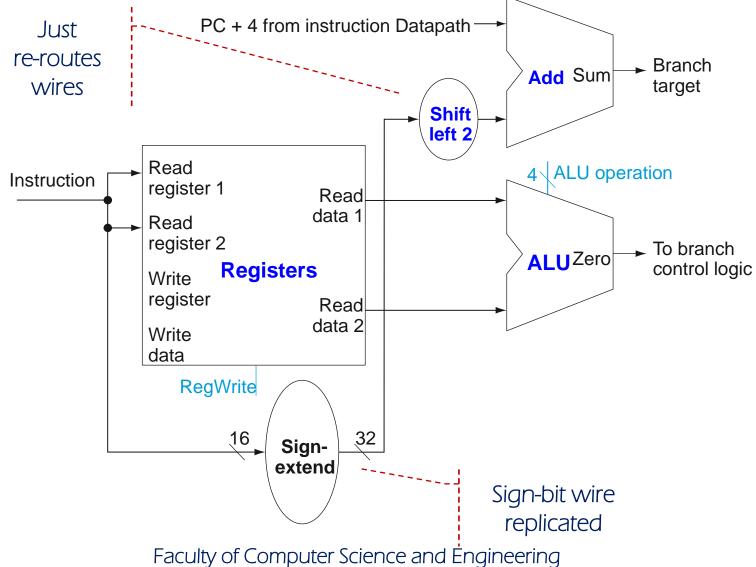a. Data memory unit

b. Sign extension unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions
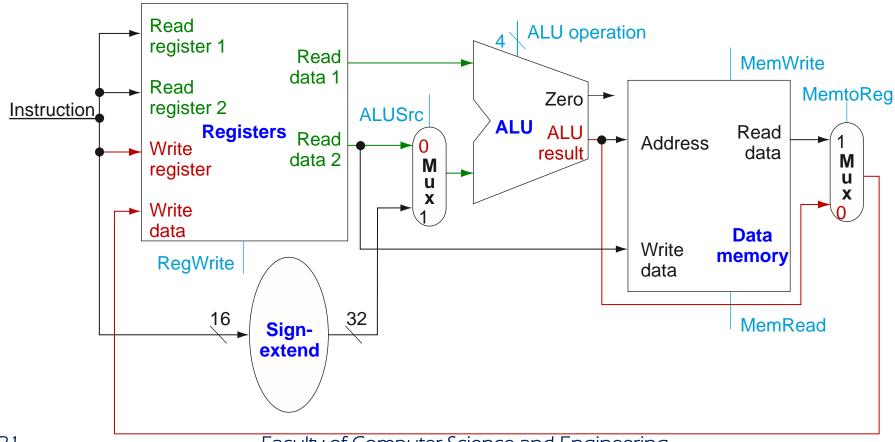
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

- **add $S0, $a0, $t0**
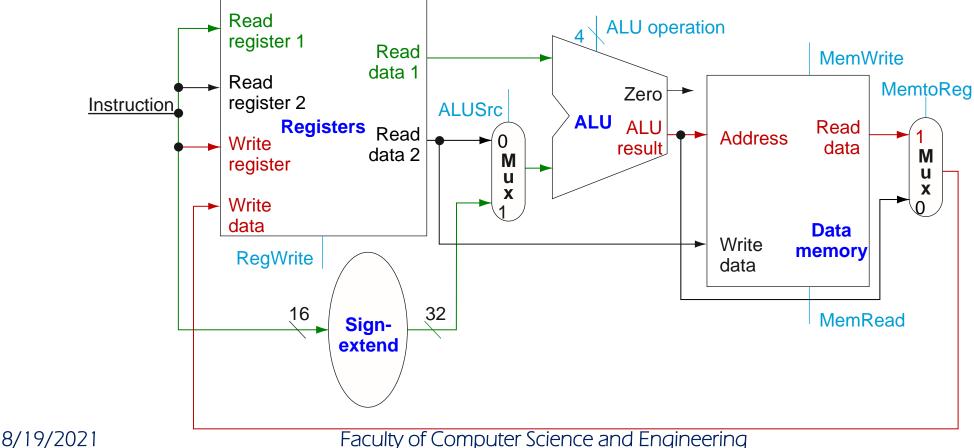
R-Type/Load/Store Datapath

- **lw $S0, 4($a0)**
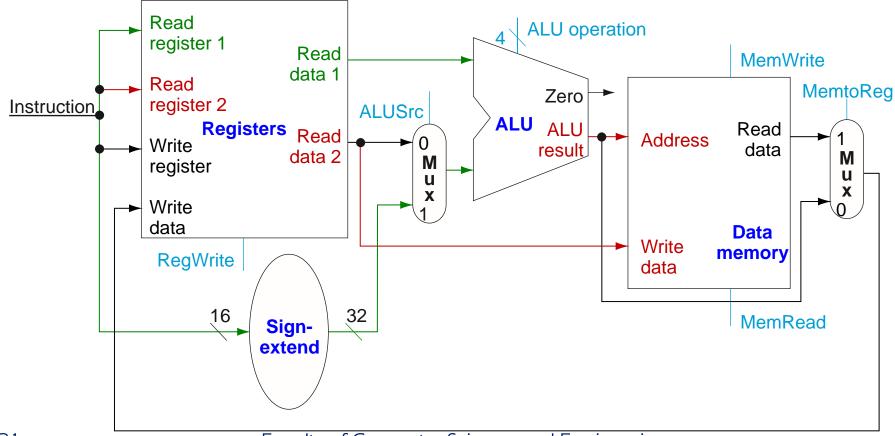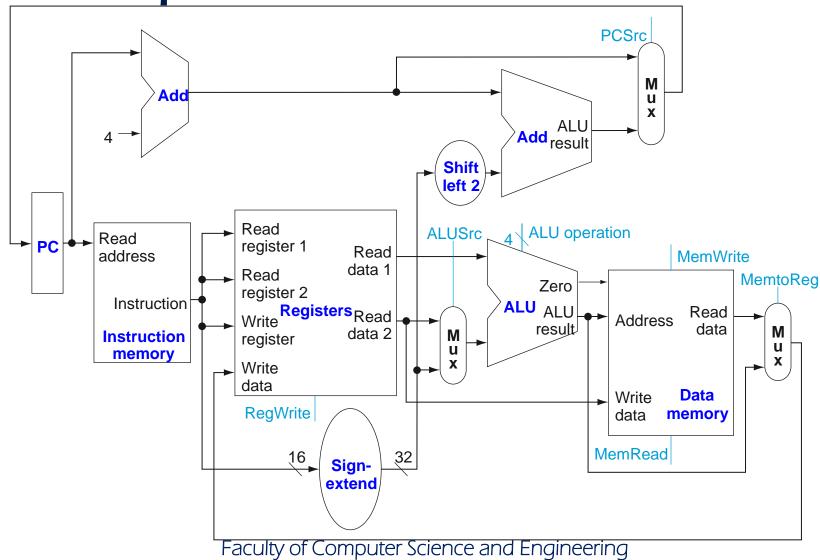
R-Type/Load/Store Datapath

- **sw $S0, 4($a0)**

Full Datapath
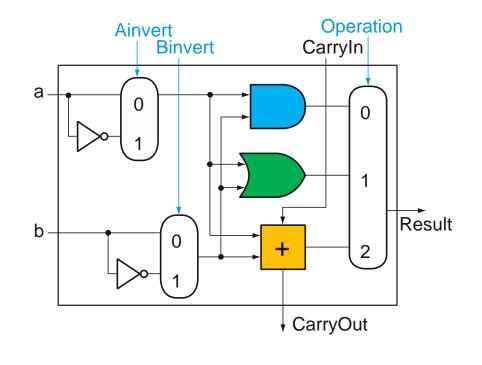
ALU Control

- **ALU used for**
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Closer look at a 1-bit ALU

ALU control			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	ADD
0	1	10	SUB
0	1	11	~~SLT~~
1	1	00	NOR
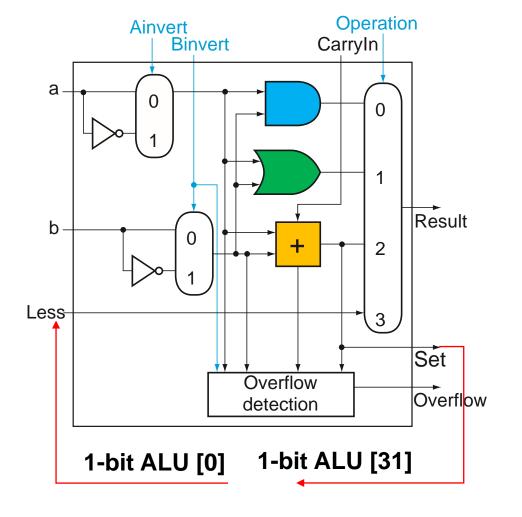
Without SLT implementation

Closer look at a 1-bit ALU

ALU control			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	ADD
0	1	10	SUB
0	1	11	SLT
1	1	00	NOR

With SLT implementation

1-bit ALU [0]   1-bit ALU [31]

ALU Control
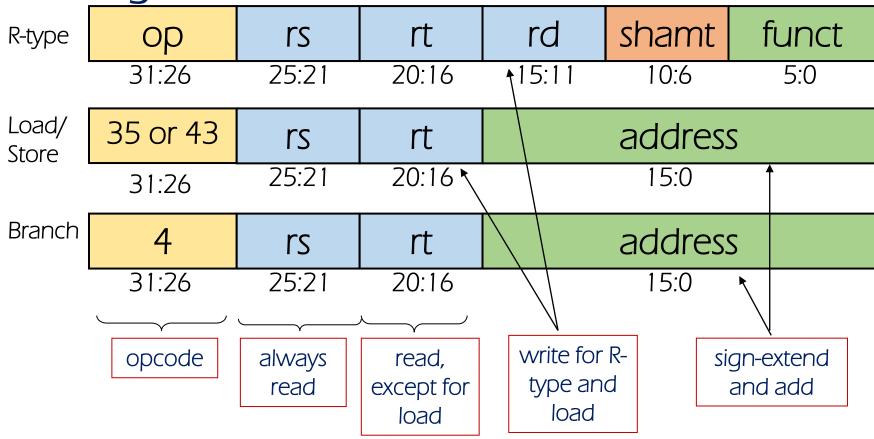
- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111
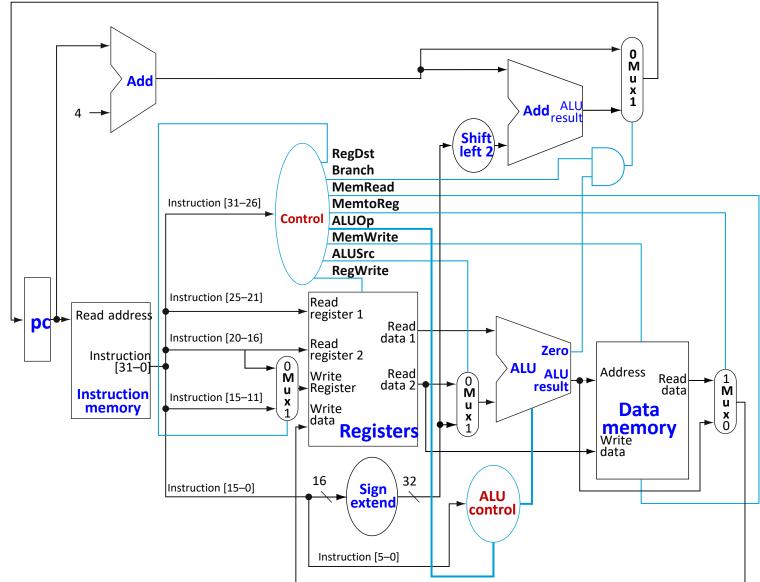
The Main Control Unit

- Control signals derived from instruction

R-type	op	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

Load/Store	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

Branch	4	rs	rt	address
	31:26	25:21	20:16	15:0

opcode

always read

read, except for load

write for R-type and load

sign-extend and add
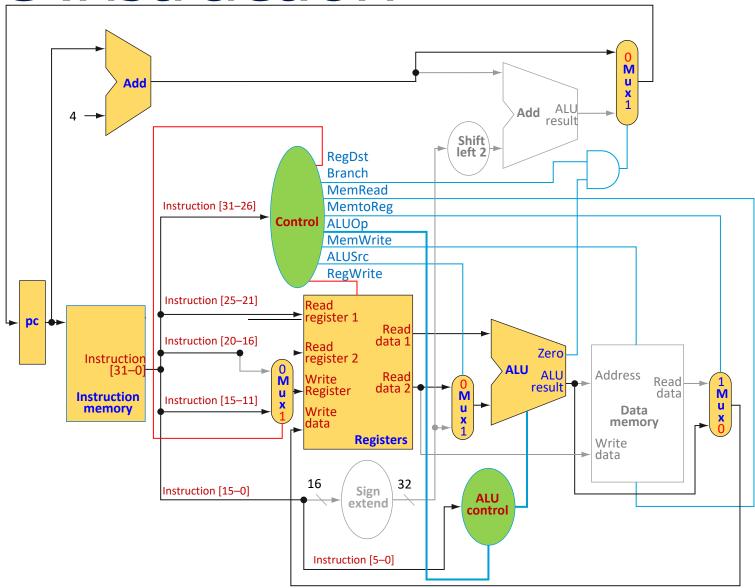
Datapath With Control
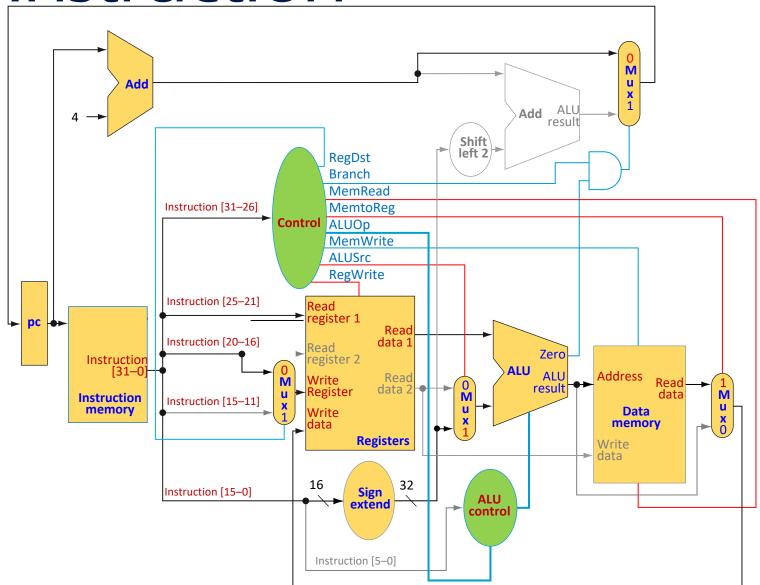
R-Type Instruction
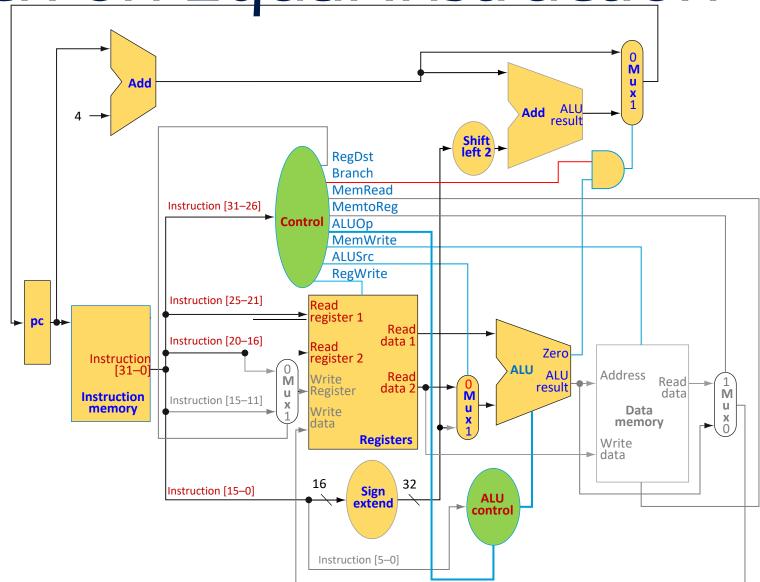
Load Instruction

Branch-on-Equal Instruction

Exercise

- What are the values of control signal of following instructions?

```
bne     $s1, $s2, exit
sw      $s1, 4($a0)
```
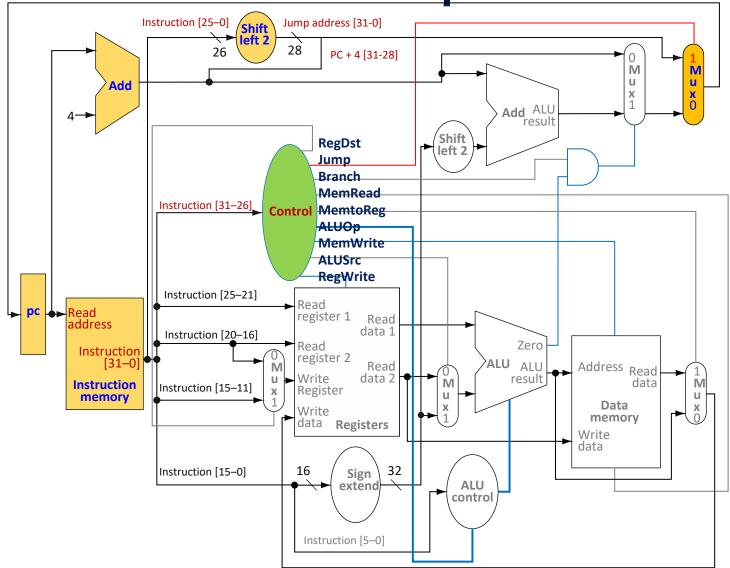
Implementing Jumps

Jump	2	address
	31:26	25:0

- **Jump uses word address**
- **Update PC with concatenation of**
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- **Need an extra control signal decoded from opcode**

Datapath With Jumps Added

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - **Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ MUX.**
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Your turn
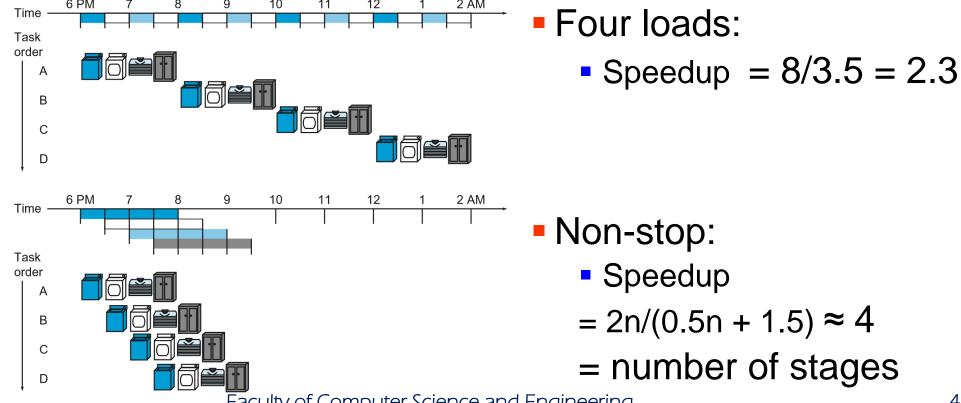
- What is critical path of following instructions:
  ```
  bne, sw
  ```

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

- Four loads:
  - Speedup = 8/3.5 = 2.3

- Non-stop:
  - Speedup
  $$= 2n/(0.5n + 1.5) \approx 4$$
  = number of stages

MIPS Pipeline

- Five stages, one step per stage
  - IF: Instruction fetch from memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register
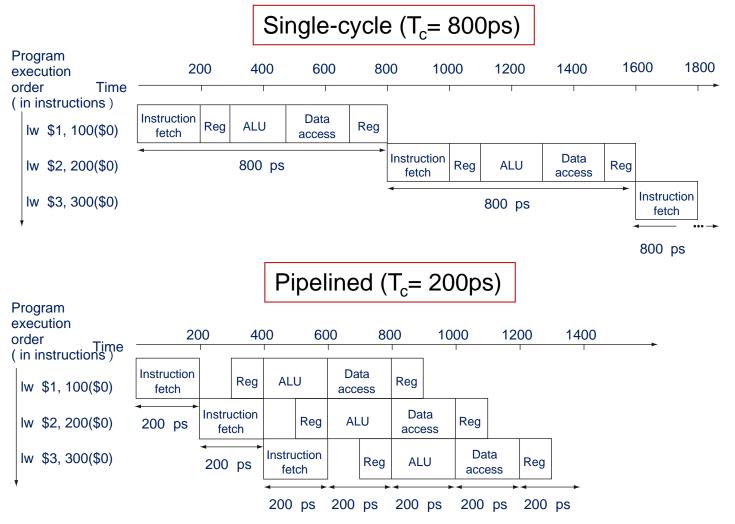
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c = 800ps)

Program execution order (in instructions)

Time

lw $1, 100($0)

Instruction fetch | Reg | ALU | Data access | Reg

800 ps

lw $2, 200($0)

Instruction fetch | Reg | ALU | Data access | Reg

800 ps

lw $3, 300($0)

Instruction fetch

800 ps

Pipelined (T_c = 200ps)

Program execution order (in instructions)

Time

lw $1, 100($0)

Instruction fetch | Reg | ALU | Data access | Reg

200 ps

lw $2, 200($0)

Instruction fetch | Reg | ALU | Data access | Reg

200 ps

lw $3, 300($0)

Instruction fetch | Reg | ALU | Data access | Reg

200 ps 200 ps 200 ps 200 ps 200 ps

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$
    $$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

- What is the value of pipeline CPI?

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
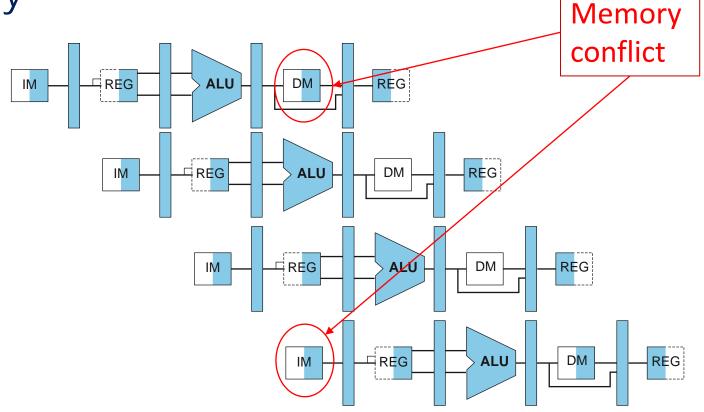   - Memory access takes only one cycle

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

■   *Assume Instruction Memory and Data Memory are in the same single memory*

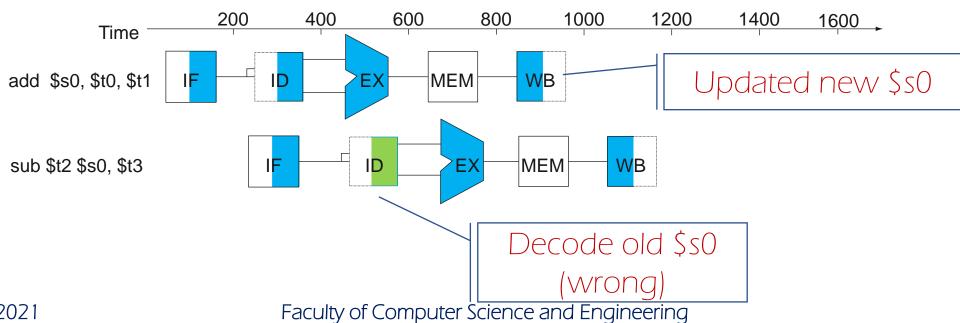

Memory conflict

Structure Hazards

- Conflict for use of a resource

- In MIPS pipeline with a single memory
    - Load/store requires data access
    - Instruction fetch would have to stall for that cycle
        - Would cause a pipeline "bubble"

- Hence, pipelined datapaths require separate instruction/data memories
    - Or separate instruction/data caches

Data Hazards

- *An instruction depends on completion of data access by a previous instruction*
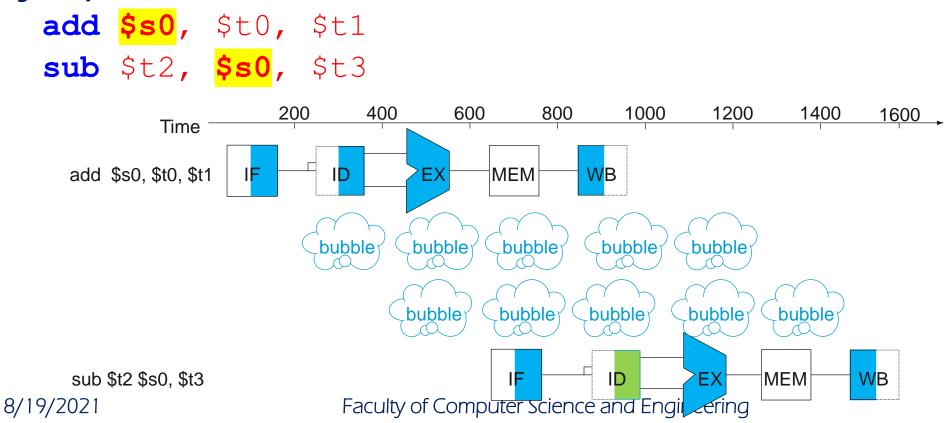
  add <mark>**$s0**</mark>, $t0, $t1

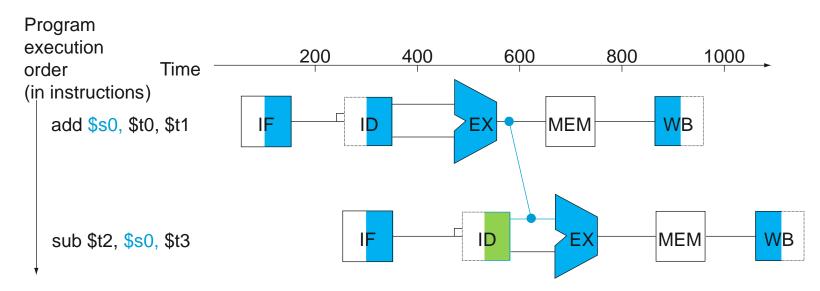 **sub** $t2, <mark>**$s0**</mark>, $t3

Data Hazards (bubble, stall, delay)

- An instruction depends on completion of data access by a previous instruction

  add **$s0**, $t0, $t1

  sub $t2, **$s0**, $t3

Faculty of Computer Science and Engineering
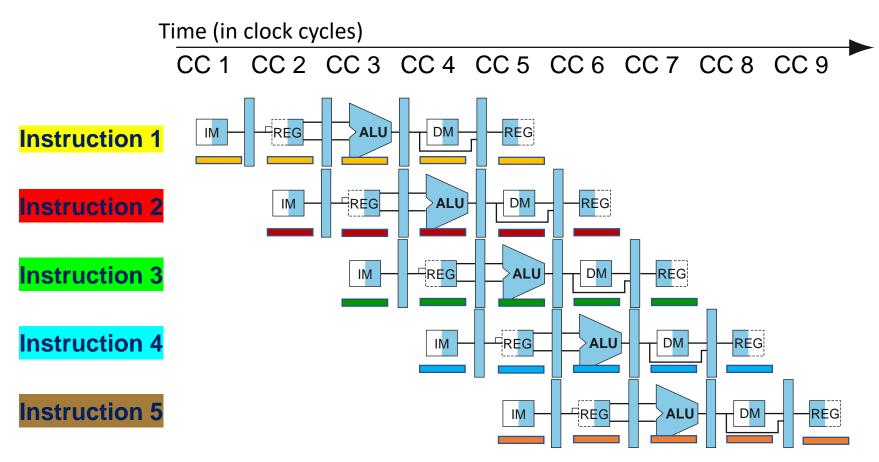
Forwarding (aka Bypassing)

- **Use result when it is computed**
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

Program
execution
order
(in instructions)   Time

200   400   600   800   1000

add $s0, $t0, $t1   IF   ID   EX   MEM   WB

sub $t2, $s0, $t3   IF   ID   EX   MEM   WB

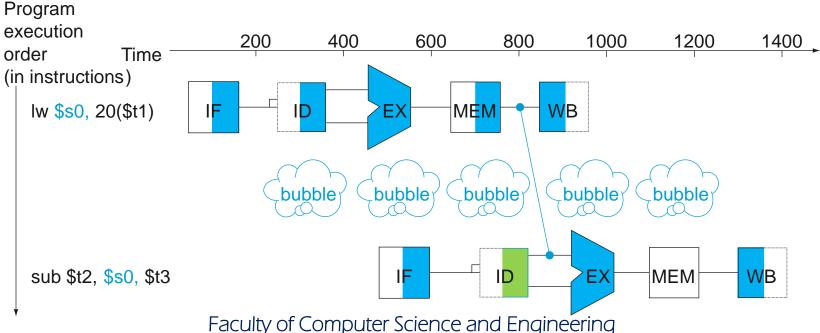# Pipeline visualization

How many stalls?

- Example 1

```
lw   $2,  $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $12, $2
sw   $14, 100($2)
```

- Example 2

```
        addi $2, $0, 10
Loop:   addi $2,  $2,  -1
        bne  $2, $0, Loop
```

Load-Use Data Hazard

- **Can't always avoid stalls by forwarding**
  - If value not computed when needed
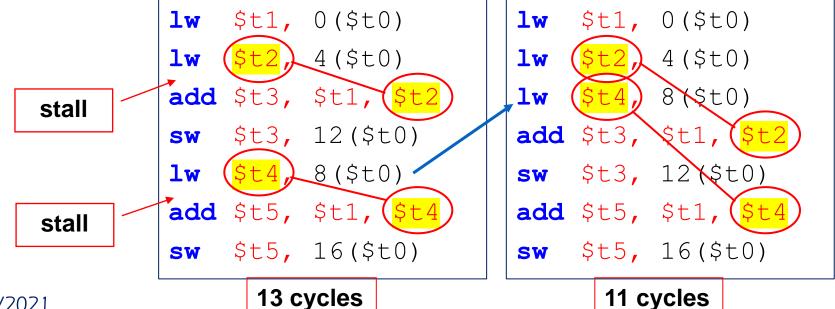 - Can't forward backward in time!

Program execution order (in instructions)

Time → 200 400 600 800 1000 1200 1400

lw $s0, 20($t1)
IF — ID — EX — MEM — WB

bubble bubble bubble bubble bubble

sub $t2, $s0, $t3
IF — ID — EX — MEM — WB

Code Scheduling to Avoid Stalls

- **Reorder code to avoid use of load result in the next instruction**

- **C code for A = B + E; C = B + F;**

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

stall

stall

13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

How many stalls? And …

```
        lw  $10, 20($1)
        bne $10, $9, else
        sub $11, $2, $3
        add $12, $11, $4
        j exit
else:   lw  $13, 24($12)
        add $14, $5, $12
exit:
```
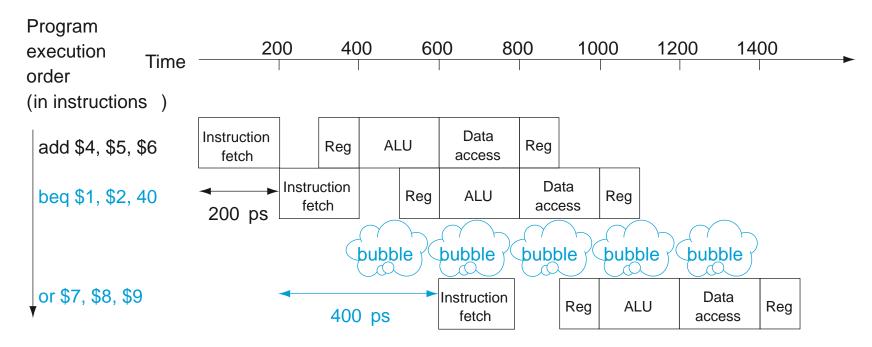
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

Program execution order (in instructions)

Time

| | 200 | 400 | 600 | 800 | 1000 | 1200 | 1400 |

add $4, $5, $6

| Instruction fetch | Reg | ALU | Data access | Reg |

beq $1, $2, 40

| Instruction fetch | Reg | ALU | Data access | Reg |

200 ps

bubble bubble bubble bubble bubble

or $7, $8, $9

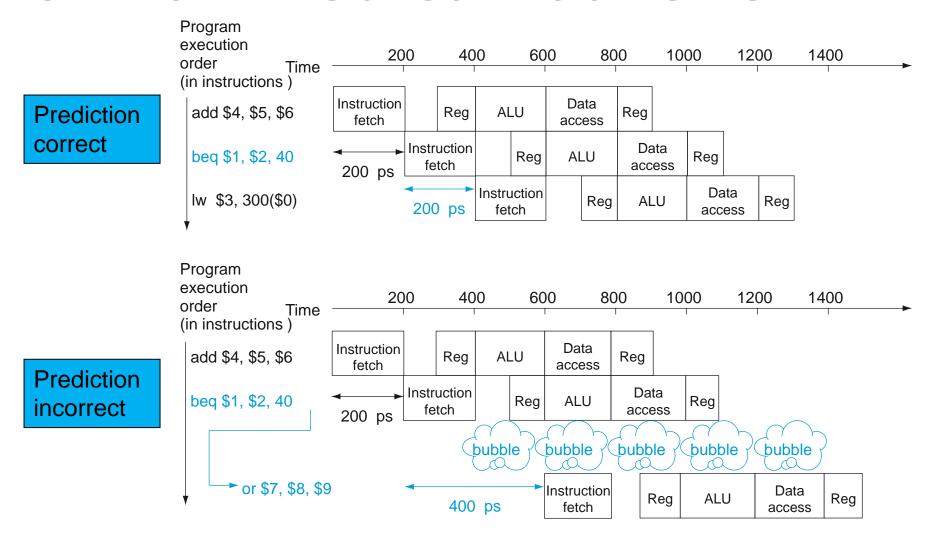| Instruction fetch | Reg | ALU | Data access | Reg |

400 ps

Branch Prediction

- *Longer pipelines can't readily determine branch outcome early*
  - *Stall penalty becomes unacceptable*
- Predict outcome of branch
  - *Only stall if prediction is wrong*
- *In MIPS pipeline*
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
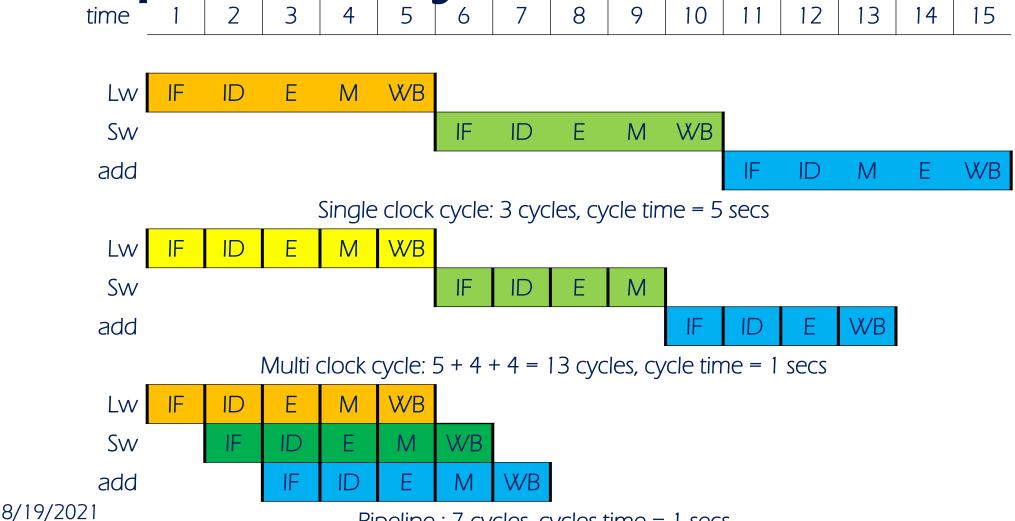   - When wrong, stall while re-fetching, and update history

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

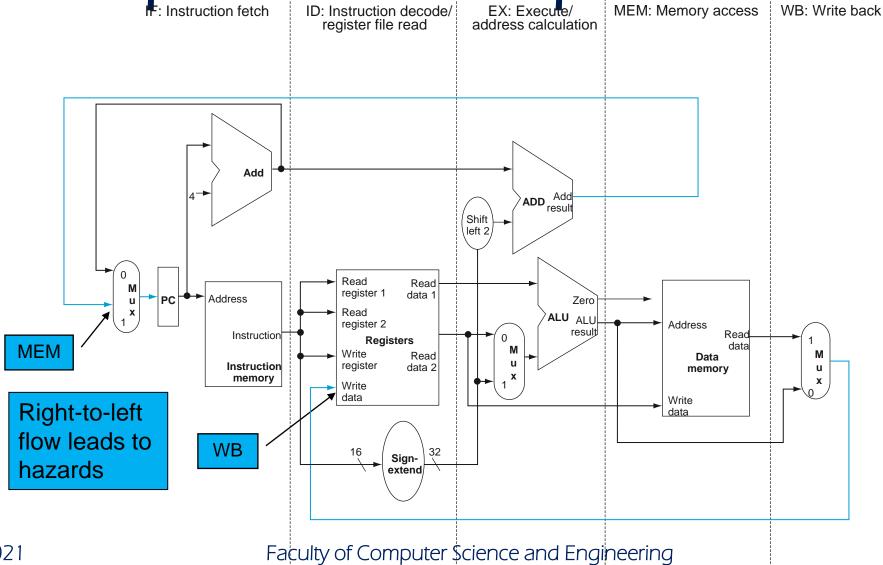# Single clock cycle vs Pipeline vs Multiple clock cycle

Single clock cycle: 3 cycles, cycle time = 5 secs

Multi clock cycle: 5 + 4 + 4 = 13 cycles, cycle time = 1 secs

Pipeline : 7 cycles, cycles time = 1 secs

Multiple clock cycle

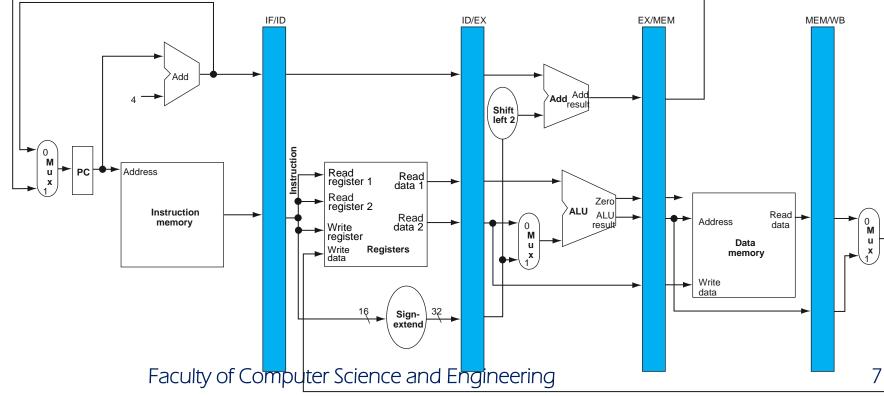| Instruction	#cycles					
Load	5	IF	ID	EXE	MEM	WB
Store	4	IF	ID	EXE	MEM	
Branch	3	IF	ID	EXE		
Arithmetic/logical	4	IF	ID	EXE	WB	
Jump	2	IF	ID			

MIPS Pipelined Datapath

Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
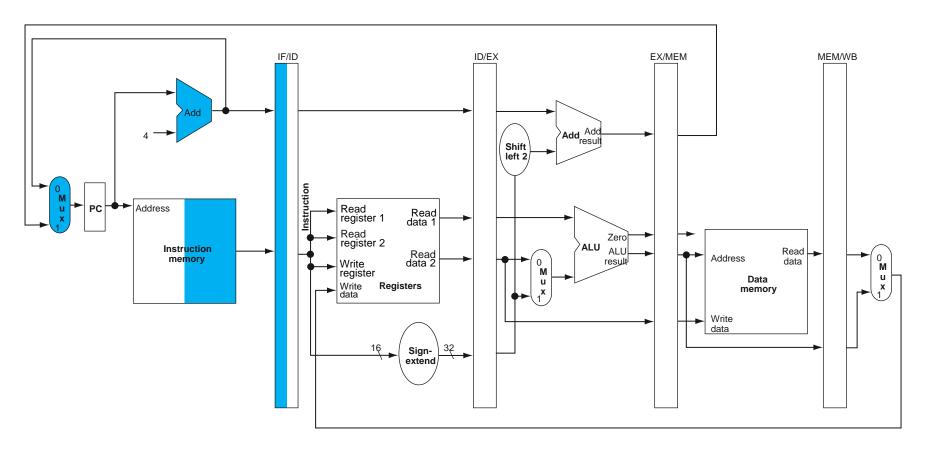
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
    - "Single-clock-cycle" pipeline diagram
        - Shows pipeline usage in a single cycle
        - Highlight resources used
    - c.f. "multi-clock-cycle" diagram
        - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store
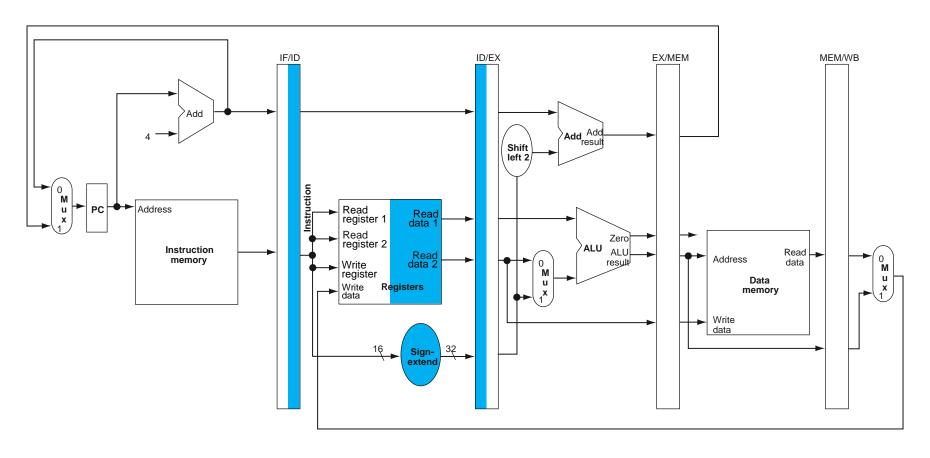
IF for Load, Store, …
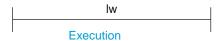
ID for Load, Store, …
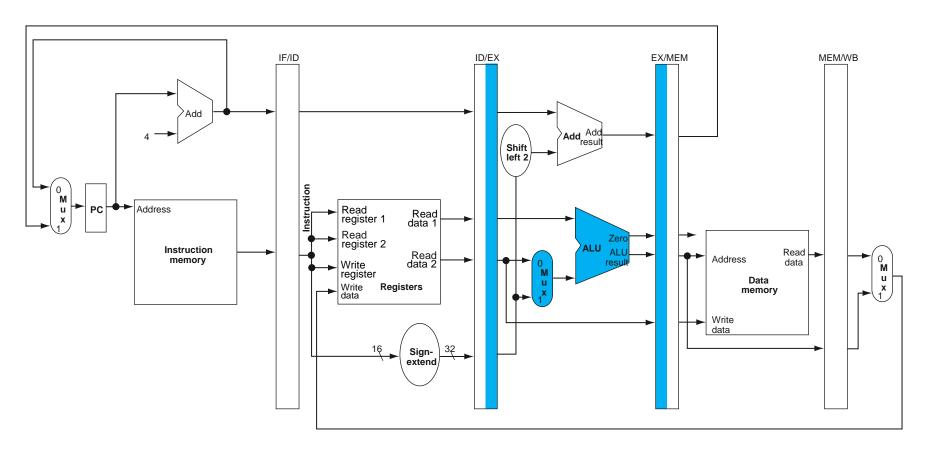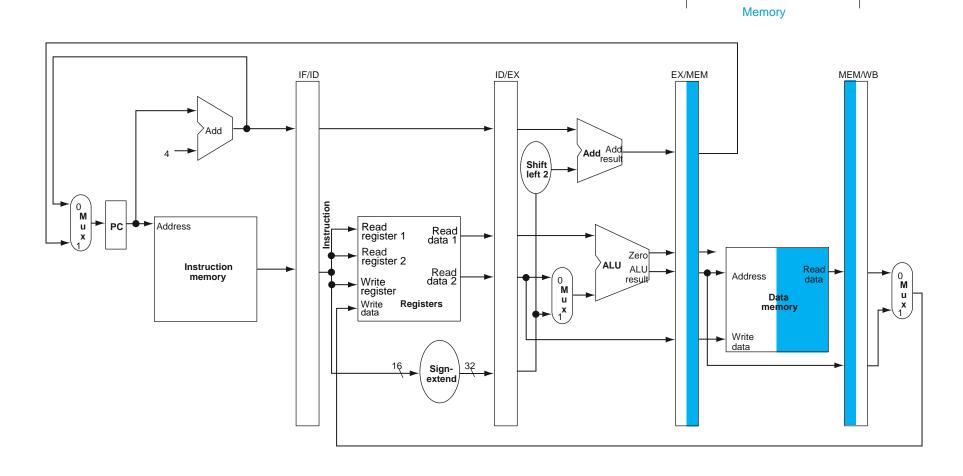
EX for Load
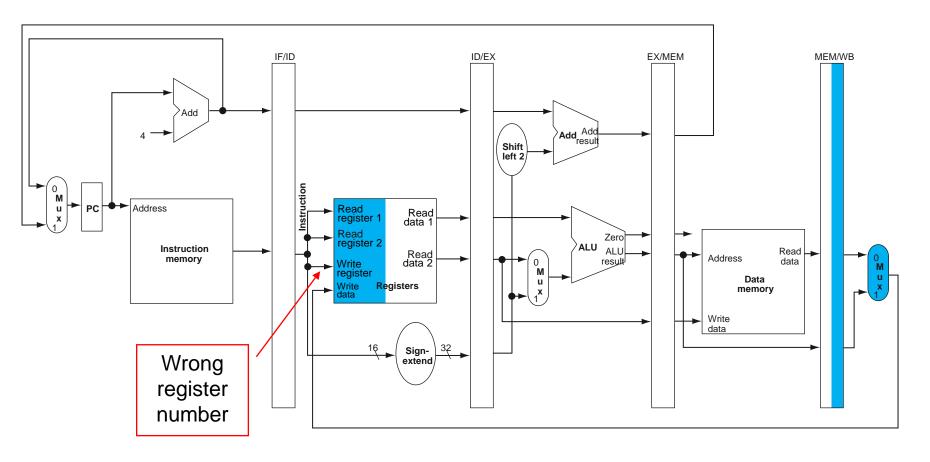
MEM for Load
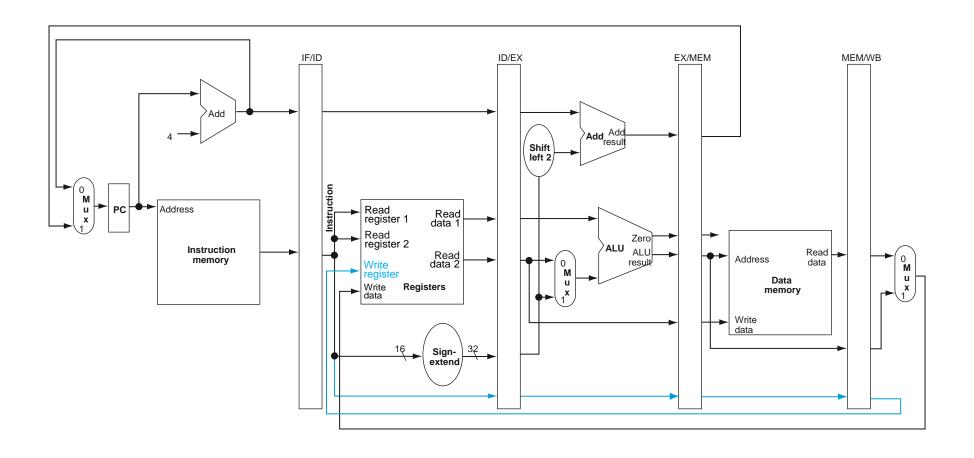
WB for Load
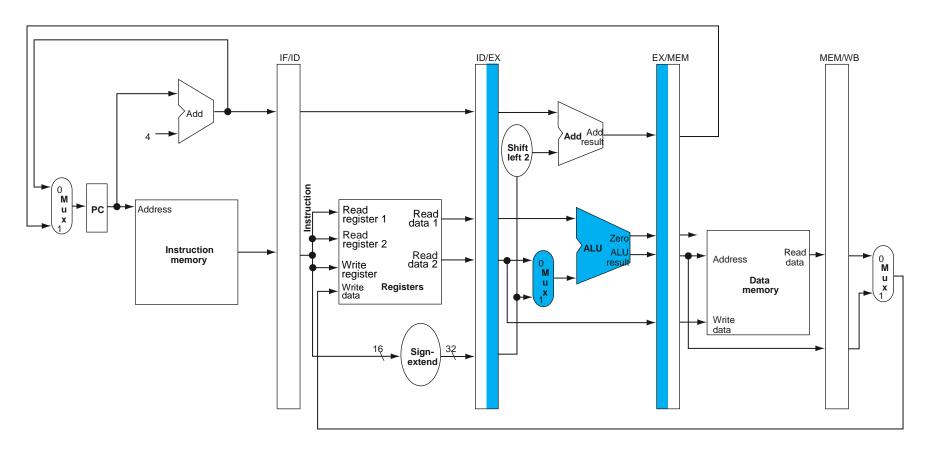
Corrected Datapath for Load
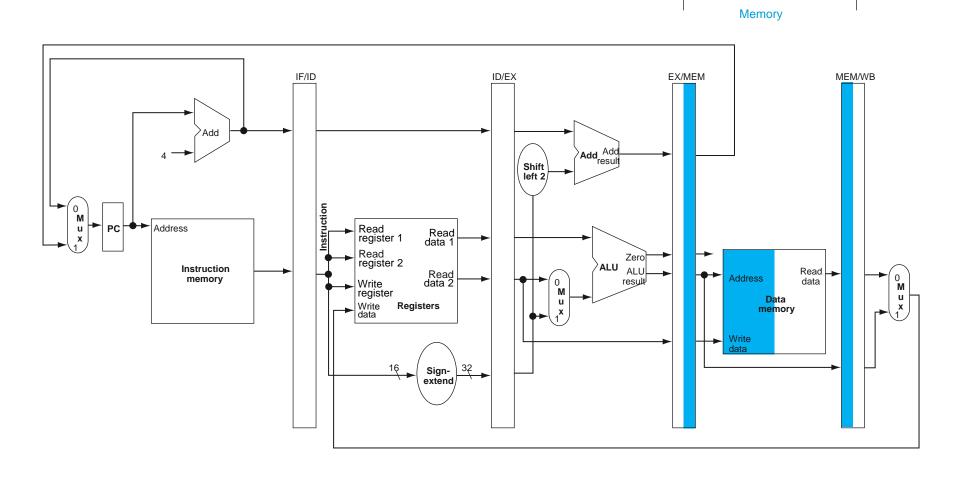
EX for Store
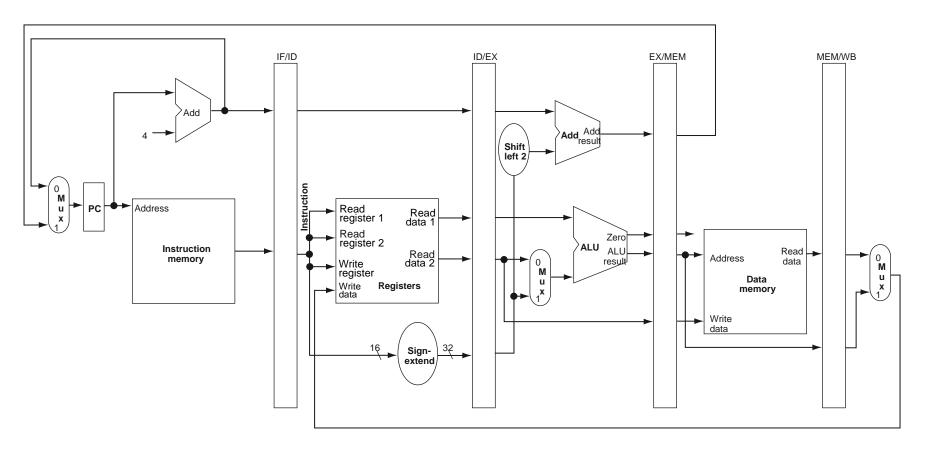
MEM for Store

WB for Store
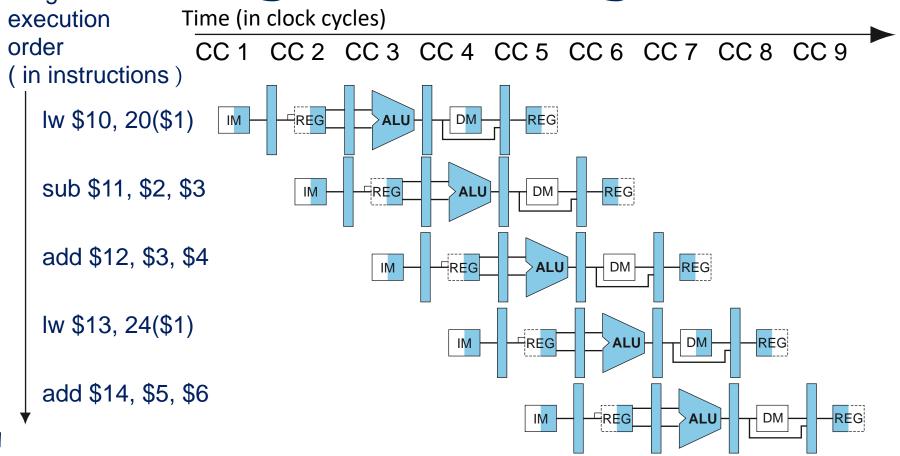
Multi-Cycle Pipeline Diagram

- Form showing resource usage
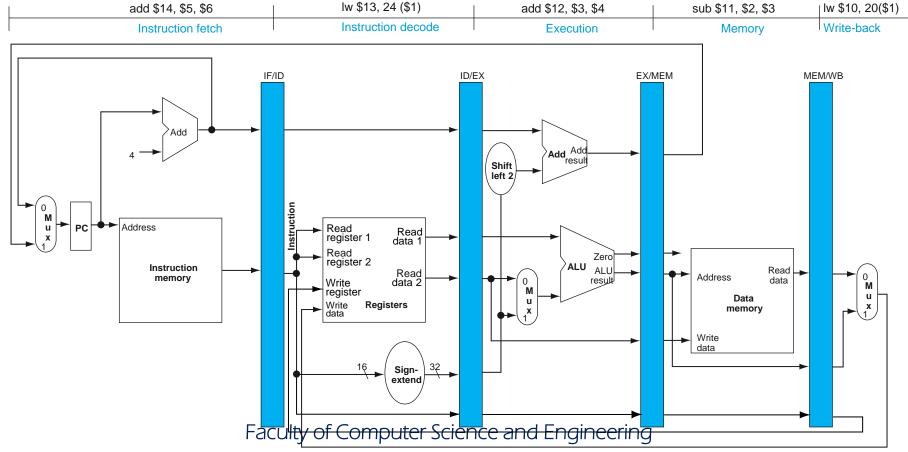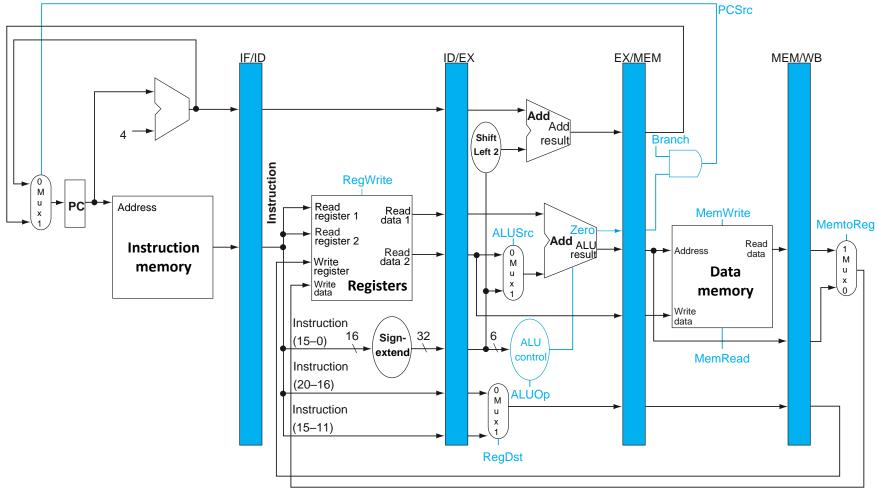
Single-Cycle Pipeline Diagram

- ## *State of pipeline in a given cycle*

Faculty of Computer Science and Engineering

Pipelined Control (Simplified)
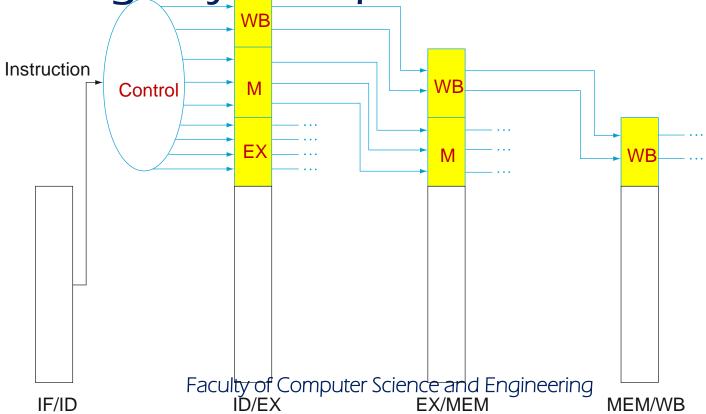
Pipelined Control
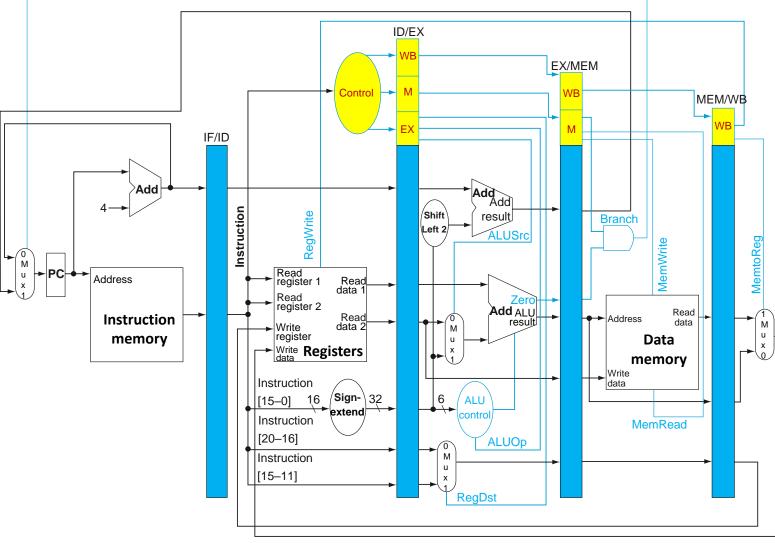
- Control signals derived from instruction
  - As in single-cycle implementation

Pipelined Control

Data Hazards in ALU Instructions
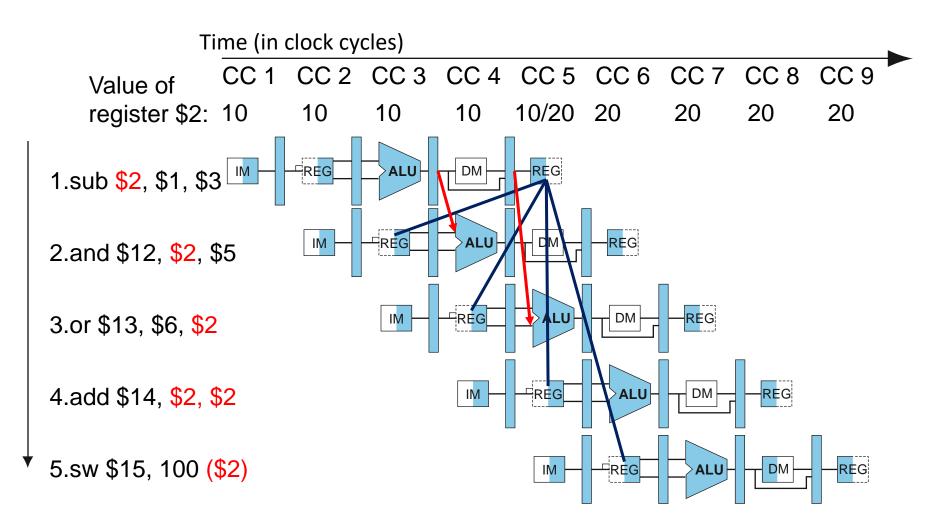
- Consider this sequence:
  ```
  sub $2,  $1, $3
  and $12, $2, $5
  or  $13, $6, $2
  add $14, $2, $2
  sw  $15, 100($2)
  ```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg
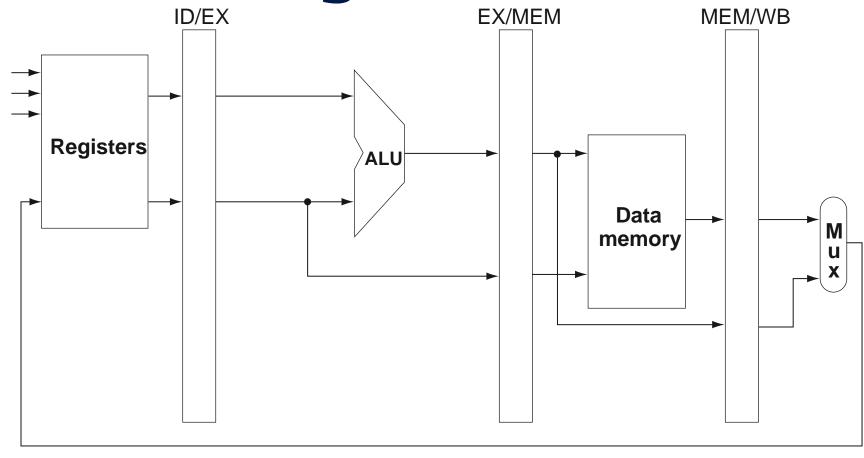
Detecting the Need to Forward

- **But only if forwarding instruction will write to a register!**
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- **And only if Rd for that instruction is not $zero**
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0
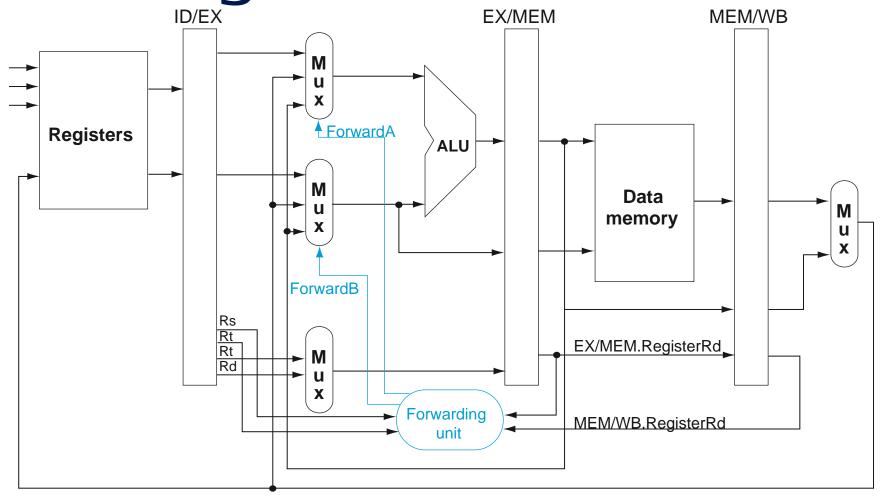
No Forwarding

a. No forwarding

Forwarding Path

b. With forwarding

Forwarding Conditions

- **EX hazard**
  - if (*EX/MEM.RegWrite*
    and (*EX/MEM.RegisterRd ≠ 0*)
    and (*EX/MEM.RegisterRd = ID/EX.RegisterRs*))
    ForwardA = 10
  - if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

Forwarding Conditions

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:
  ```
  add $1,$1,$2
  add $1,$1,$3
  add $1,$1,$4
  ```

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true
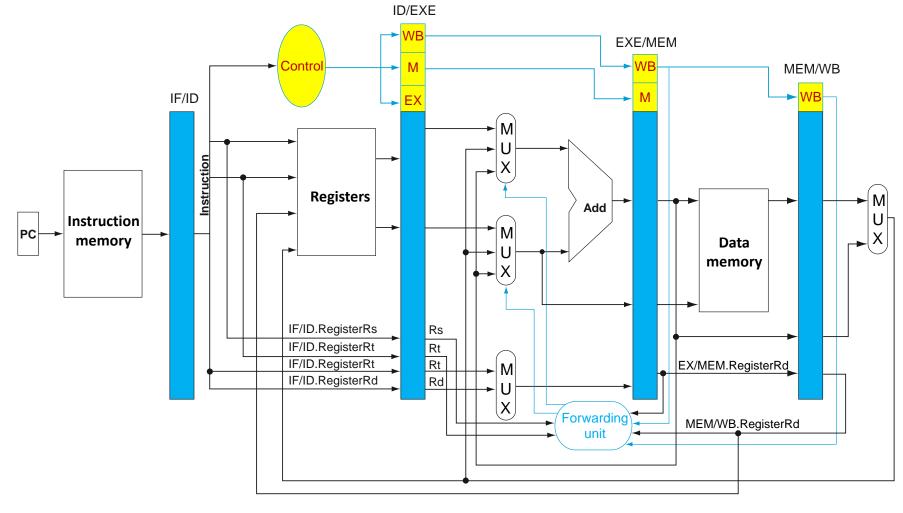
Revised Forwarding Condition

- MEM hazard

if (MEM/WB.RegWrite
        and   (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and   (EX/MEM.RegisterRd = ID/EX.RegisterRs))
        and   (MEM/WB.RegisterRd = ID/EX.RegisterRs))
        ForwardA = 01

if (MEM/WB.RegWrite
        and   (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and   (EX/MEM.RegisterRd = ID/EX.RegisterRt))
        and   (MEM/WB.RegisterRd = ID/EX.RegisterRt))
        ForwardB = 01
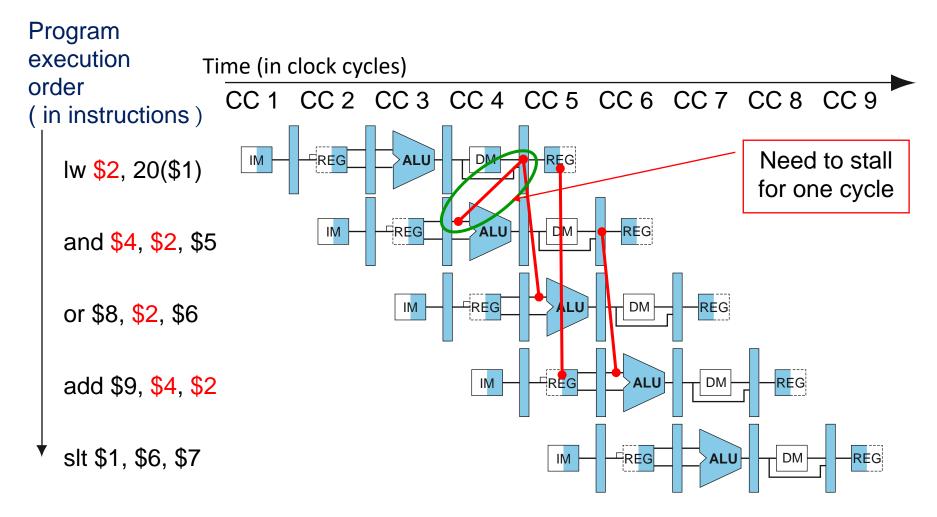
Datapath with Forwarding

Load-Use Data Hazard

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble
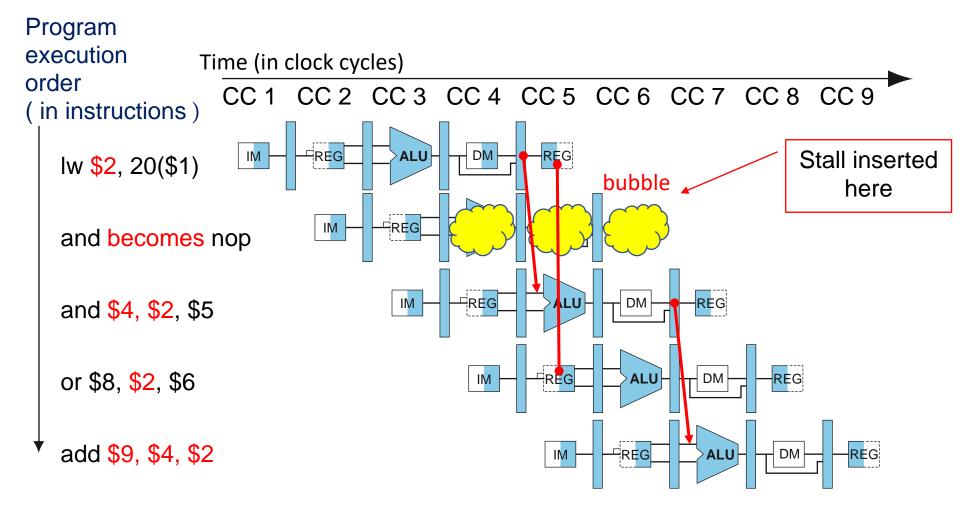
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Load-Use Data Hazard

Program execution order (in instructions)

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

lw $2, 20($1)

and becomes nop

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

bubble

Stall inserted here
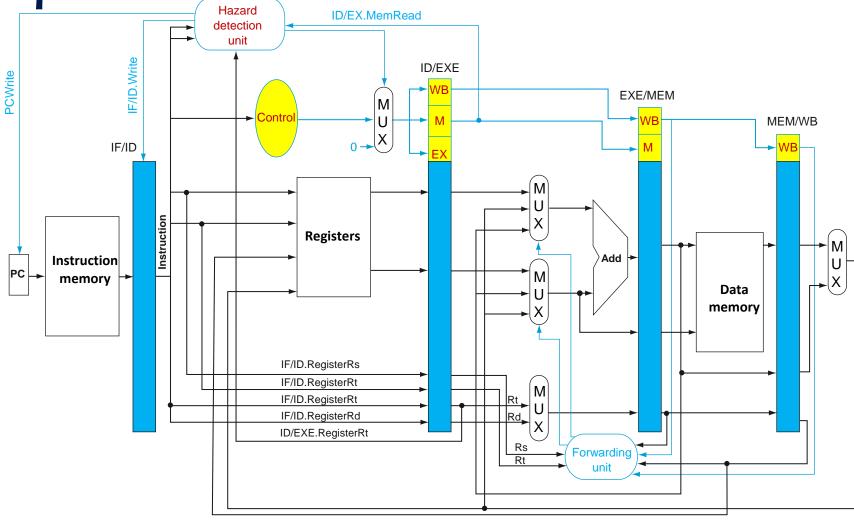
Datapath with Hazard Detection
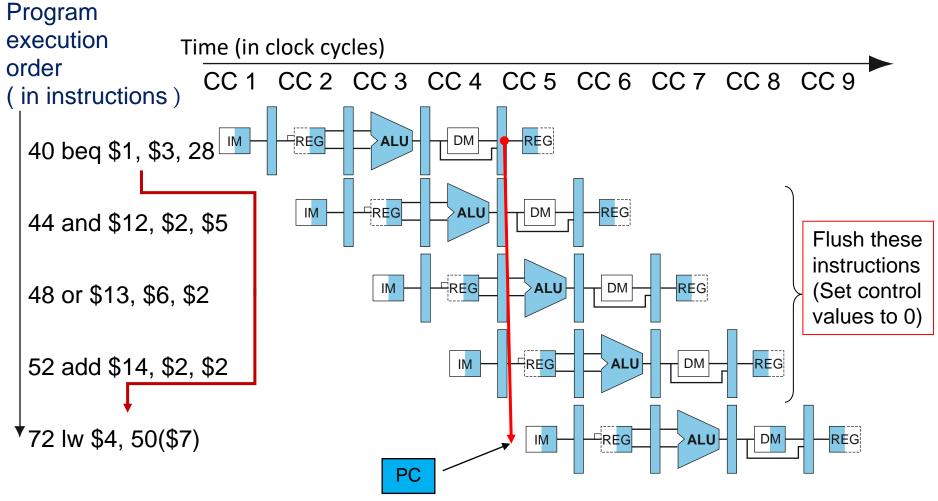
Stalls and Performance

- Stalls reduce performance
    - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
    - Requires knowledge of the pipeline structure

Branch Hazards

Program execution order (in instructions)

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

40 beq $1, $3, 28

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

72 lw $4, 50($7)

Flush these instructions (Set control values to 0)

PC

IM REG ALU DM REG
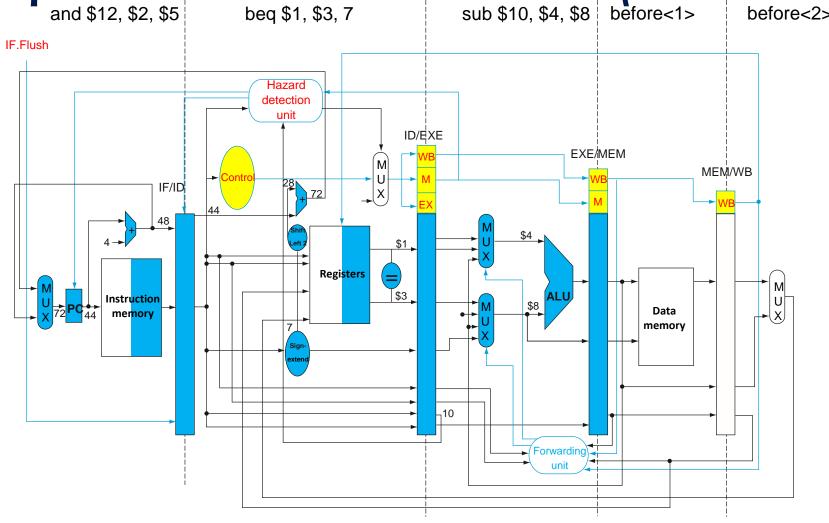
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub $10, $4, $8
40:  beq $1,  $3, 7
44:  and $12, $2, $5
48:  or  $13, $2, $6
52:  add $14, $4, $2
56:  slt $15, $6, $7
     ...
72:  lw $4, 50($7)
```

Example: Branch Taken (clock 3)
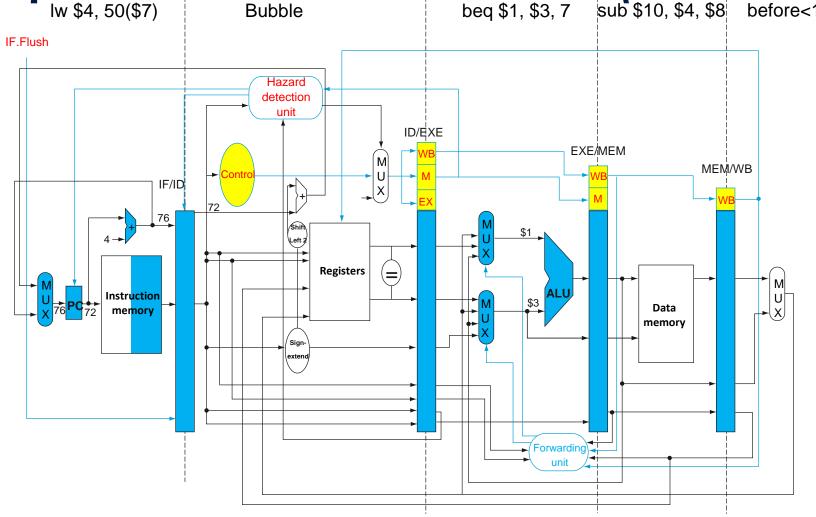
Example: Branch Taken (clock 4)

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

add $1, $2, $3 | IF | ID | EX | MEM | WB |

add $4, $5, $6 | IF | ID | EX | MEM | WB |

… | IF | ID | EX | MEM | WB |

beq $1, $4, target | IF | ID | EX | MEM | WB |

- Can resolve using forwarding

Data Hazards for Branches
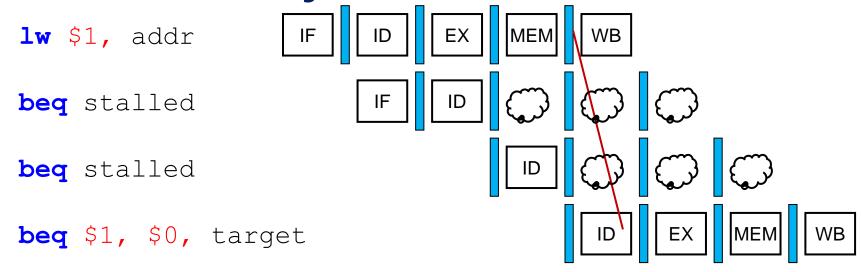
- **If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction**
  - **Need 1 stall cycle**

`lw` `$1,` `addr`   IF | ID | EX | MEM | WB

`add` `$4, $5, $6`   IF | ID | EX | MEM | WB

`beq` `stalled`   IF | ID | ☁ ☁ ☁

`beq` `$1, $4,` `target`   ID | EX | MEM | WB

Data Hazards for Branches

- **If a comparison register is a destination of immediately preceding load instruction**
  - **Need 2 stall cycles**

`lw` `$1,` `addr`   IF ID EX MEM WB

`beq` `stalled`   IF ID ☁ ☁ ☁

`beq` `stalled`   ID ☁ ☁ ☁

`beq` `$1,` `$0,` `target`   ID EX MEM WB

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- *Inner loop branches mispredicted twice!*

```
outer: …
       …
inner: …

       …
       beq …, …, inner
       …
       beq …, …, outer
```
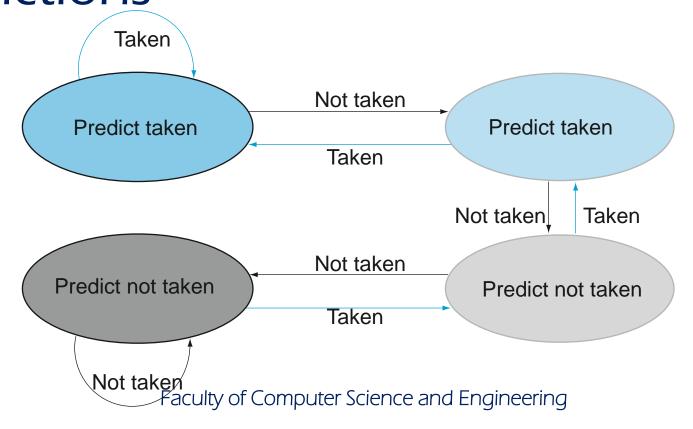
- Mispredict as taken on last iteration of inner loop

- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- *Only change prediction on two successive mispredictions*

Calculating the Branch Target

- *Even with predictor, still need to calculate the target address*
  - *1-cycle penalty for a taken branch*
- *Branch target buffer*
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - *If hit and instruction is branch predicted taken, can fetch target immediately*

Exceptions and Interrupts

- *"Unexpected" events requiring change in flow of control*
  - *Different ISAs use the terms differently*
- *Exception*
  - *Arises within the CPU*
    - *e.g., undefined opcode, overflow, syscall, …*
- *Interrupt*
  - *From an external I/O controller*
- *Dealing with them without sacrificing performance is hard*

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- **Vectored Interrupts**
  - *Handler address determined by the cause*
- **Example:**
  - *Undefined opcode: C000 0000*
  - *Overflow: C000 0020*
  - *....:   C000 0040*
- **Instructions either**
  - *Deal with the interrupt, or*
  - *Jump to real handler*

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …
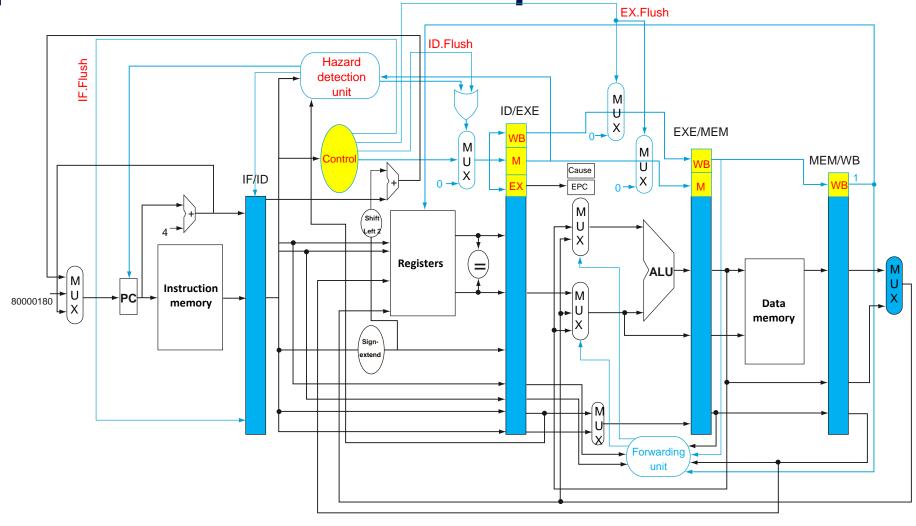
Exceptions in a Pipeline

- **Another form of control hazard**
- **Consider overflow on add in EX stage**
  - add $1, $2, $1
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- **Similar to mispredicted branch**
  - Use much of the same hardware

Pipeline with Exceptions

Exception Properties

- **Restartable exceptions**
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- **PC saved in EPC register**
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

- Exception on add in
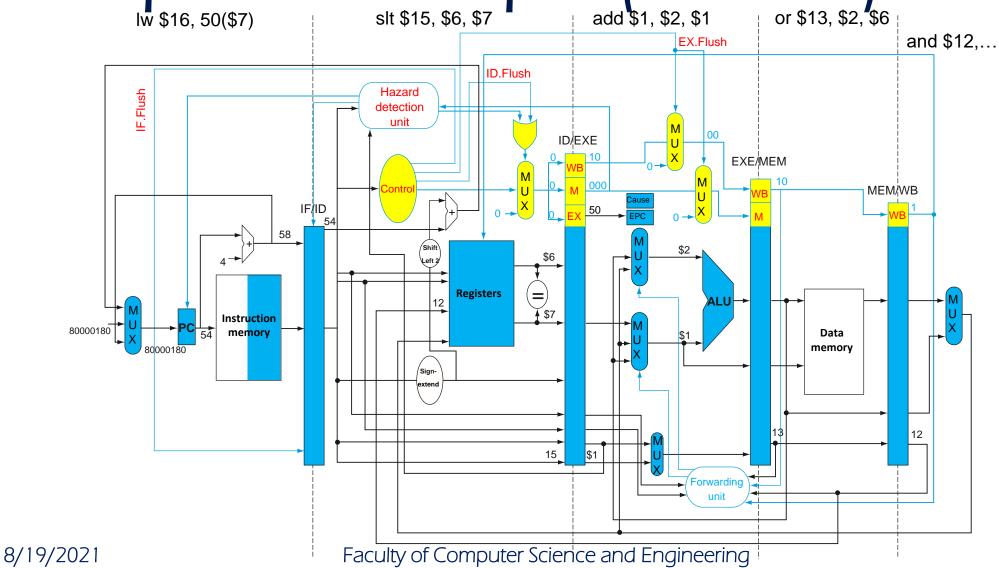
```
40 sub $11, $2, $4
44 and $12, $2, $5
48 or  $13, $2, $6
4C add $1, $2, $1
50 slt $15, $6, $7
54 lw  $16, 50($7)
   …
```

- Handler

```
80000180 sw $25, 1000($0)
80000184 sw $26, 1004($0)
   …
```

Exception Example (clock 6)
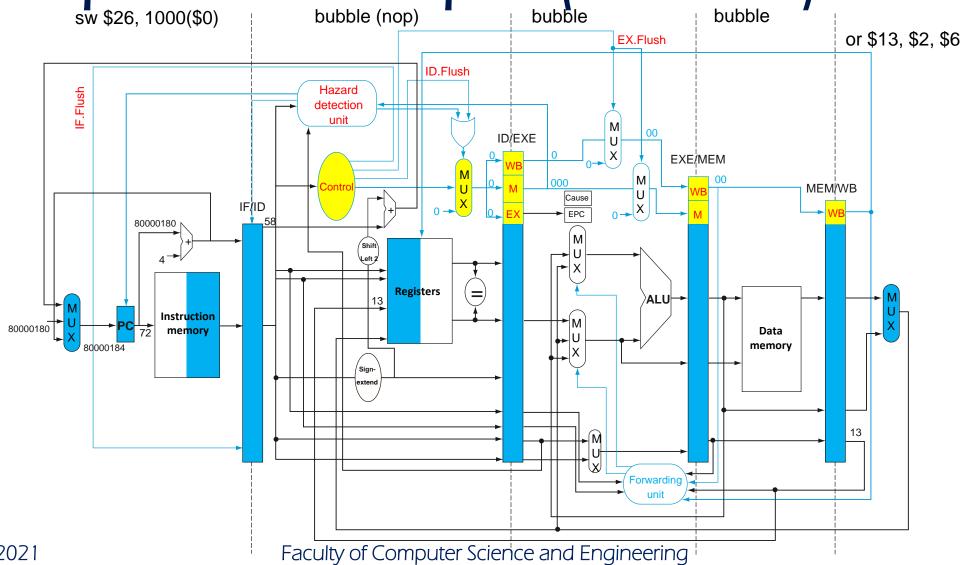
Exception Example (clock 7)

sw $26, 1000($0)   bubble (nop)   bubble   bubble   or $13, $2, $6

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Instruction-Level Parallelism (ILP)

- **Pipelining: executing multiple instructions in parallel**
- **To increase ILP**
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- *"Guess" what to do with an instruction*
  - *Start operation as soon as possible*
  - *Check whether guess was right*
    - *If so, complete the operation*
    - *If not, roll-back and do the right thing*
- *Common to static and dynamic multiple issue*
- *Examples*
  - *Speculate on branch outcome*
    - *Roll back if path taken is different*
  - *Speculate on load*
    - *Roll back if location is updated*

Compiler/Hardware Speculation

- **Compiler can reorder instructions**
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
- **Hardware can look ahead for instructions to execute**
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Static Multiple Issue

- Compiler groups instructions into "issue packets"
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- **Compiler must remove some/all hazards**
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
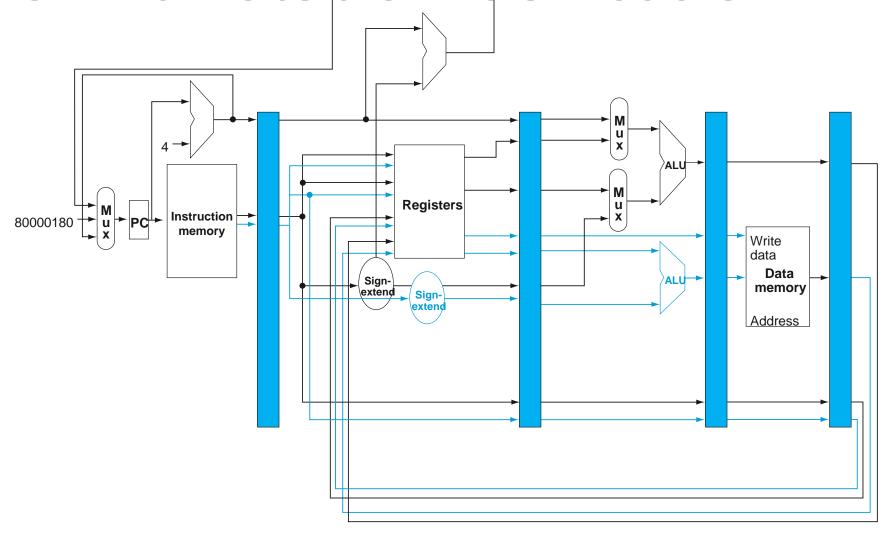
MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages								
n + 00	ALU/branch	IF	ID	EX	MEM	WB				
n + 04	Load/store	IF	ID	EX	MEM	WB				
n + 08	ALU/branch		IF	ID	EX	MEM	WB			
n + 12	Load/store		IF	ID	EX	MEM	WB			
n + 16	ALU/branch			IF	ID	EX	MEM	WB		
n + 20	Load/store			IF	ID	EX	MEM	WB		

MIPS with Static Dual Issue

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    ```
    add $t0, $s0, $s1
    lw  $s2, 0($t0)
    ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- ## Schedule this for dual-issue MIPS

```
Loop:  lw    $t0, 0($s1)      # $t0=array element
       addu  $t0, $t0, $s2     # add scalar in $s2
       sw    $t0, 0($s1)       # store result
       addi  $s1, $s1,-4       # decrement pointer
       bne   $s1, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 0($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- **Replicate loop body to expose more parallelism**
  - Reduces loop-control overhead
- **Use different registers per replication**
  - Called "register renaming"
  - Avoid loop-carried "anti-dependencies"
    - Store followed by a load of the same register
    - Aka "name dependence"
      - Reuse of a register name

Loop Unrolling Example

- ## IPC = 14/8 = 1.75
  - ### Closer to 2, but at cost of registers and code size

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,–16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
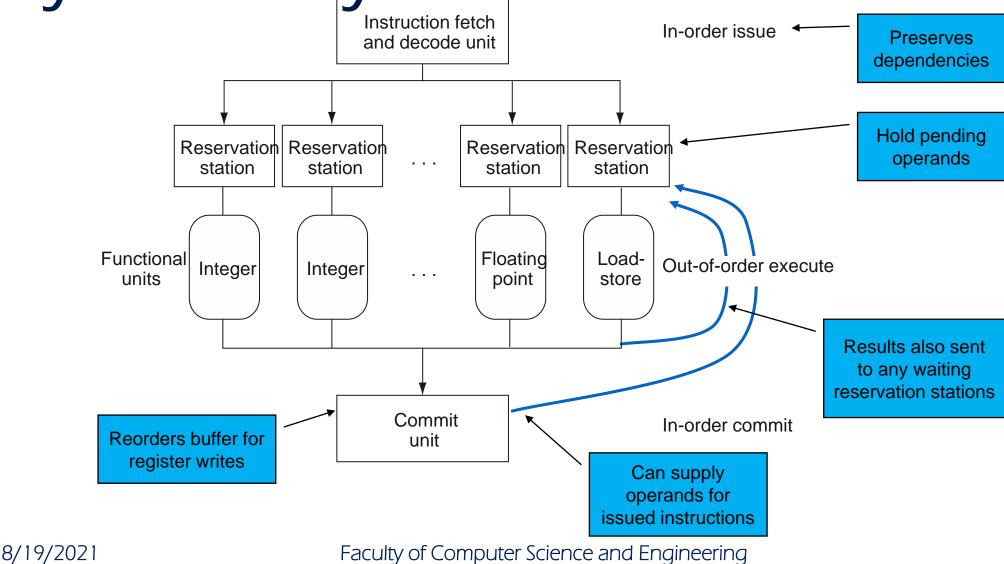
Dynamic Pipeline Scheduling

- *Allow the CPU to execute instructions out of order to avoid stalls*
  - *But commit result to registers in order*
- *Example*

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - *Can start sub while addu is waiting for lw*

Dynamically Scheduled CPU

Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- **Poor ISA design can make pipelining harder**
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall