COMPUTER ARCHITECTURE

2nd Year Computer science

Chapiter 3:

PROCESSOR
(The MIPS R3000 microprocessor)

Abdelhafid Boussouf University
2025-2026

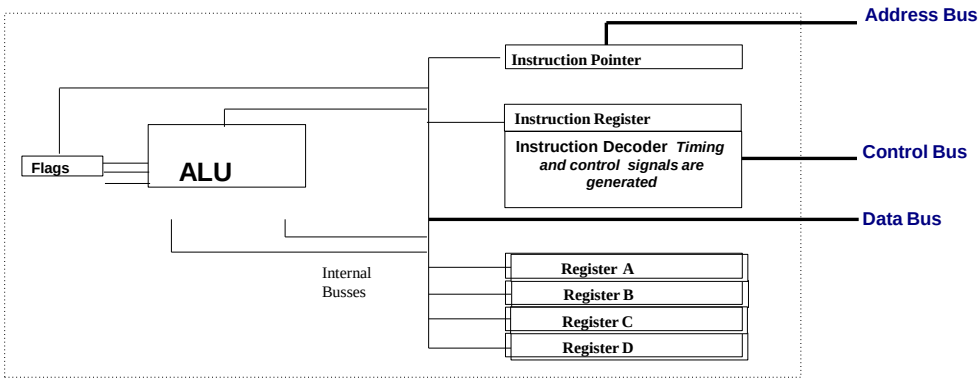
MICROPROCESSORS

A program stored in the memory provides instructions to the CPU to perform a specific action. This action can be a simple addition. It is function of the CPU to ***fetch*** the program instructions from the memory and ***execute*** them.

- The CPU contains a number of ***registers*** to store information inside the CPU temporarily. Registers inside the CPU can be 8-bit, 16-bit, 32-bit or even 64-bit depending on the CPU.
- The CPU also contains ***Arithmetic and Logic Unit (ALU).*** The ALU performs arithmetic (add, subtract, multiply, divide) and logic (AND, OR, NOT) functions.
- The CPU contains a program counter also known as the ***Instruction Pointer*** to point the address of the next instruction to be executed.
- ***Instruction Decoder*** is a kind of dictionary which is used to interpret the meaning of the instruction fetched into the CPU. Appropriate control signals are generated according to the meaning of the instruction.

MICROPROCESSORS

Inside the CPU:



Internal block diagram of a CPU

3

Instruction Set Architecture (ISA)

Instruction set architecture (ISA) is the interface between the hardware and the lowest-level software. This is one of the most important abstractions.

4

Instruction Set Architecture (ISA)

- An ISA includes the following …
  - Instructions and Instruction Formats
  - Data Types, Encodings, and Representations
  - Programmable Storage: Registers and Memory
  - Addressing Modes: to address Instructions and Data
  - Handling Exceptional Conditions (like overflow)

5

Instruction Set Architecture (ISA)

ISA Classification

- Complex instruction set computer (CISC)
  - x86/x64 (Intel and AMD)
- Reduced instruction set computer (RISC)
  - ARM, PowerPC, MIPS, RISC-V
- Very long instruction word (VLIW)
  - Itanium, Elbrus

6

Reduced Instruction Set Computing (RISC)

Reduced Instruction Set Computing (RISC) concept was proposed by teams of researchers at **Stanford University** (John Hennessy) and **University of California Berkeley** (David Paterson) in **early 1980s** as an alternative of Complex Instruction Set Computing (CISC) dominating at that time.

- RISC ISAs dominate – most mobile devices use ARM (RISC)
- Modern CISC ISAs (x86/x64) are RISC-like underneath
- 2017 Turing Award to Patterson and Hennessy

7

RISC vs. CISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

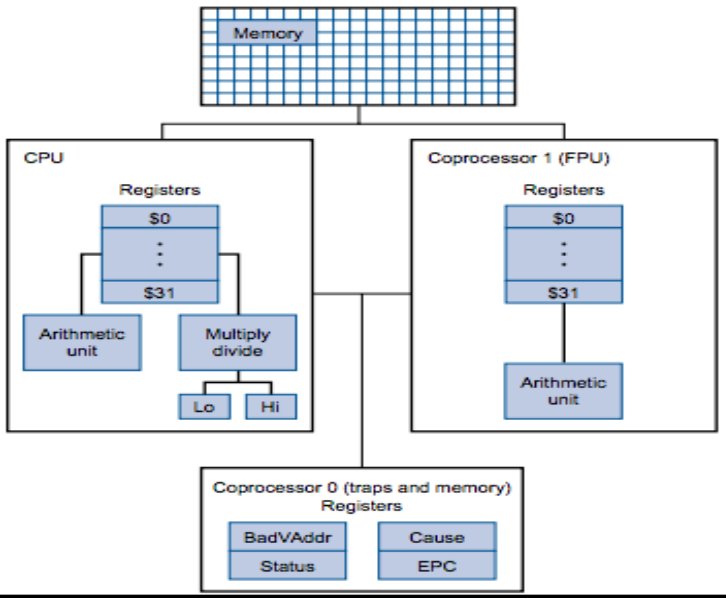
8

MIPS Microprocessor

MIPS (Microprocessor without Interlocked Pipelined Stages) is a family of reduced instruction set computer (RISC) instruction set architectures (ISA); developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

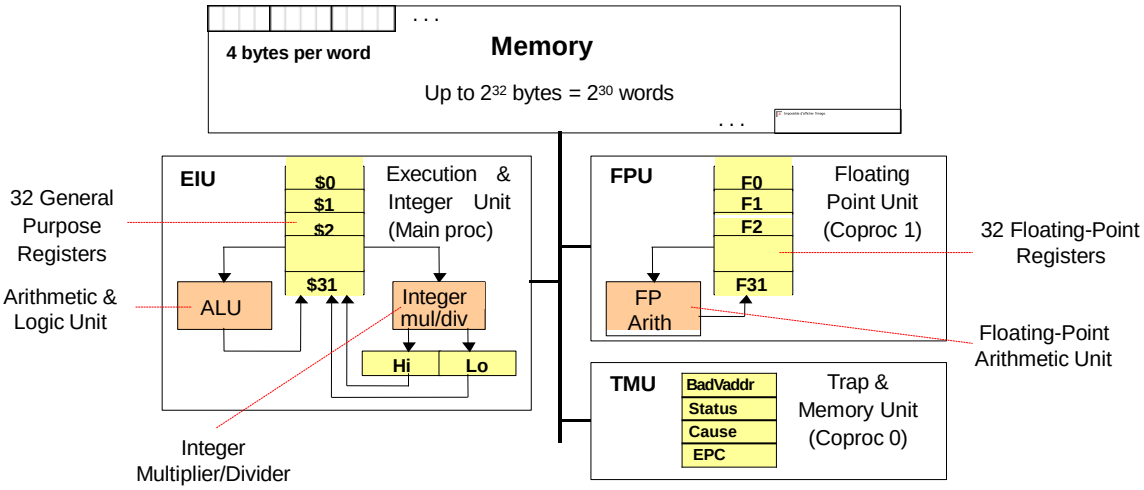
9

Overview of the MIPS Architecture



10

Overview of the MIPS Architecture



11

MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - All registers are 32-bit wide in the MIPS 32-bit architecture
  - Software defines names for registers to standardize their use
  - Assembler can refer to registers by name or by number ($ notation)

Name	Register	Usage	
$zero	$0	Always 0	(forced by hardware)
$at	$1	Reserved for assembler use	
$v0 - $v1	$2 - $3	Result values of a function	
$a0 - $a3	$4 - $7	Arguments of a function	
$t0 - $t7	$8 - $15	Temporary Values	
$s0 - $s7	$16 - $23	Saved registers	(preserved across call)
$t8 - $t9	$24 - $25	More temporaries	
$k0 - $k1	$26 - $27	Reserved for OS kernel	
$gp	$28	Global pointer	(points to global data)
$sp	$29	Stack pointer	(points to top of stack)
$fp	$30	Frame pointer	(points to stack frame)
$ra	$31	Return address	(used by jal for function call)

12

Special-Purpose Registers

-**PC** (Program Counter), points to the next instruction to be executed

-**Hi** :High result of multiplication and division operations

-**Lo** :Low result of multiplication and division operations

-**SR** (status): Status Register, Contains the interrupt mask and enable bits

-**CAUSE** : specifies what kind of interrupt or exception just happened.

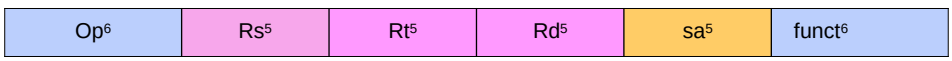
-**EPC** : Exception PC, Contains the address of the instruction when the exception occurred.

-**Vaddr**: Bad Address Register, Contains the invalid memory address caused by load, store, or fetch.

13

Instruction Formats

- All instructions are 32-bit wide, Three instruction formats:
- Register (R-Type)
  - Register-to-register instructions
  - Op: operation code specifies the format of the instruction

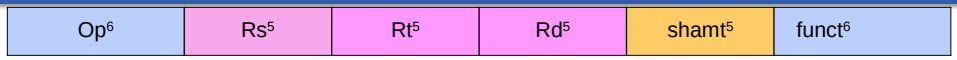


- Immediate (I-Type)
  - 16-bit immediate constant is part in the instruction

- Jump (J-Type)
  - Used by jump instructions

14

R-Type Instruction Format



- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define many R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs**, **Rt**: first and second source operands
  - **Rd**: destination operand
  - **shamt**: the shift amount used by shift instructions

15

Encoding MIPS Instructions

Field Opcode

	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

16

Encoding MIPS Instructions

Field func when OPCOD = SPECIAL

	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

17

Encoding MIPS Instructions

Examples:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,4097	4: lw $t1, x
	0x00400004	0x8c290000	lw $9,0($1)	
	0x00400008	0x20010001	addi $1,$0,1	5: subi $t2,$t1,1
	0x0040000c	0x01215022	sub $10,$9,$1	
	0x00400010	0x3c011001	lui $1,4097	6: sw $t2, x
	0x00400014	0xac2a0000	sw $10,0($1)	

18

Encoding MIPS Instructions

Examples:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c010001	lui $1,1	4: lui $1,1
	0x00400004	0x24090005	addiu $9,$0,5	5: li $t1,5

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090005	addiu $9,$0,5	4: li $t1,5
	0x00400004	0x240a0006	addiu $10,$0,6	5: li $t2,6
	0x00400008	0x012a5820	add $11,$9,$10	6: add $t3,$t1,$t2

19

From Assembly to Machine Code

Let's see an example of a R-format instruction, first as a combination of decimal numbers and then of binary numbers. Consider the instruction:

add $t0, $s1, $s2

The op and funct fields in combination (0 and 32 in this case) tell that this instruction performs addition (add).

The rs and rt fields, registers $s1 (17) and $s2 (18), are the source operands, and the rd field, register $t0 (8), is the destination operand.

The shamt field is unused in this instruction, so it is set to 0.

20

From Assembly to Machine Code

Thus, the decimal representation of instruction add $t0, $s1, $s2 is:

- op = 000000 (special)
- rs = 17 ($s1)
- rt = 18 ($s2)
- rd = 8 ($t0)
- shamt = 0 (not used)
- funct = 100000(add)

The binary representation is:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

21

Pseudo-Instructions

Most assembler instructions represent machine instructions one-to-one. The assembler can also treat common variations of machine instructions as if they were instructions in their own right. Such instructions are called pseudo-instructions.

The hardware need not implement the pseudo-instructions, but their appearance in assembly language simplifies programming. Register $at (assembler temporary) is reserved for this purpose.

blt $s1, $s2, L	□	slt $at, $s1, $s2 bne $at, $zero, L
li $s1, 20	□	addiu $s1, $zero, 20
move $t0, $t1	□	addu $t0, $zero, $t1

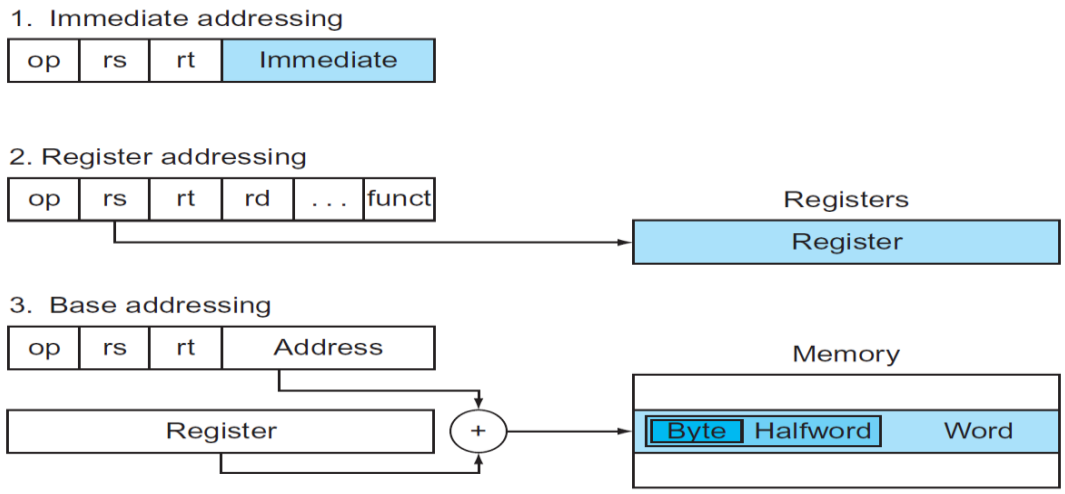
22

Addressing Modes

- MIPS addressing modes are:
  1. Immediate addressing where the operand is a constant in the instruction itself
  2. Register addressing where the operand is a register
  3. Base or displacement addressing where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
  4. PC-relative addressing where the branch address is the sum of the PC with a constant in the instruction
  5. Pseudo-direct addressing where the jump address is a constant in the instruction concatenated with the upper bits of the PC

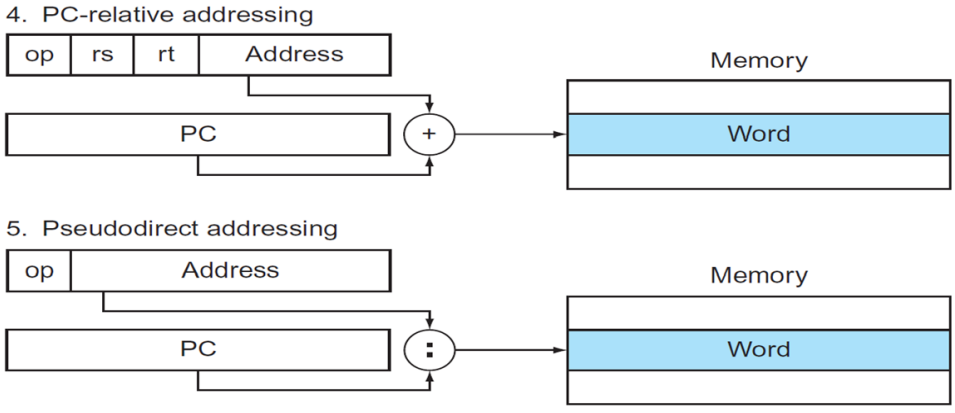
23

Addressing Modes



24

Addressing Modes



25

Addressing Modes

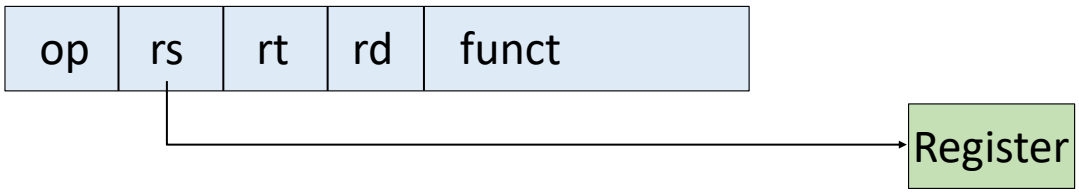
Immediate

addi $t0, $t1, 5

26

Addressing Modes

Register

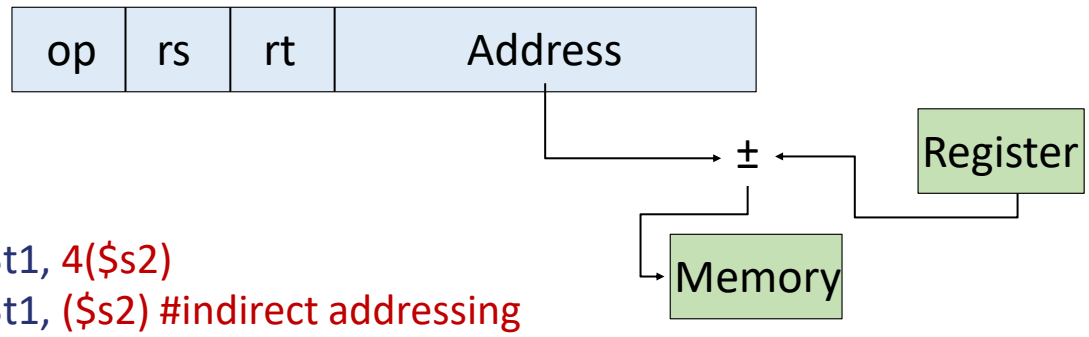


add $t0, $t1, $t2

27

Addressing Modes

Base (Arrays, structures, pointers)

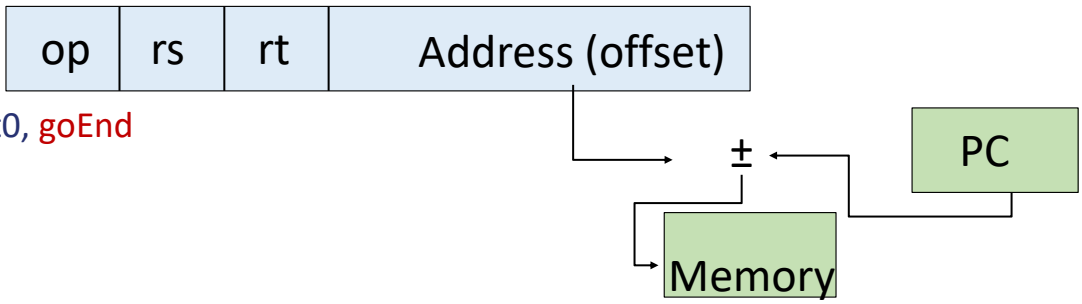


lw $t1, 4($s2)
lw $t1, ($s2) #indirect addressing

28

Addressing Modes

PC-relative (e.g., conditional branches, need an offset)



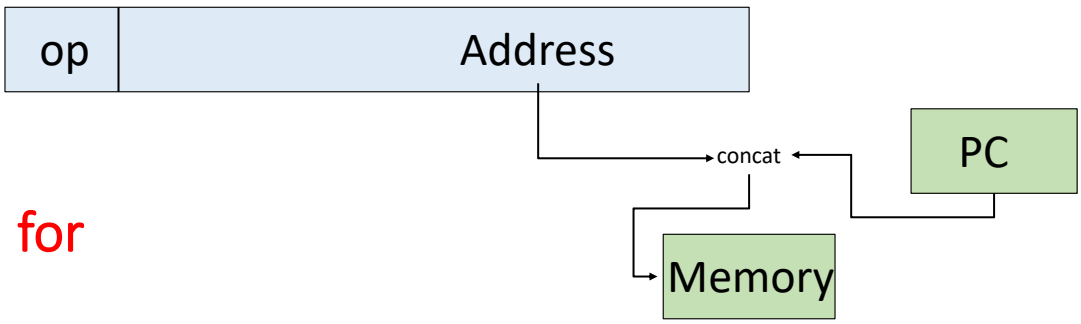
beqz $t0, goEnd

- Effective Address: adding a 16-bit address shifted left 2 bits to thePC

29

Addressing Modes

Pseudodirect



j for

- Effective Address: Concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

30

Addressing Modes

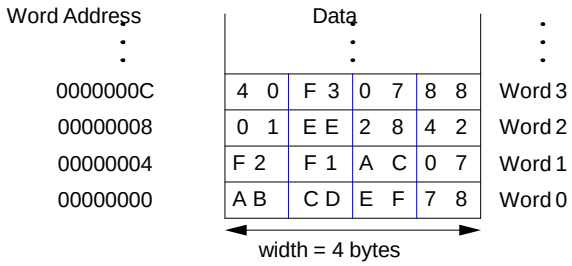
Examples:

Address	Code	Basic	Source
0x00400000	0x3c011001	lui $1,4097	5: la $t0, vars
0x00400004	0x34280000	ori $8,$1,0	
0x00400008	0x8d090000	lw $9,0($8)	6: lw $t1, 0($t0)
0x0040000c	0x8d0a0004	lw $10,4($8)	7: lw $t2, 4($t0)
0x00400010	0x012a082a	slt $1,$9,$10	8: saut: bge $t1, $t2, exit
0x00400014	0x10200005	beq $1,$0,5	
0x00400018	0x00092021	addu $4,$0,$9	9: move $a0, $t1
0x0040001c	0x24020001	addiu $2,$0,1	10: li $v0, 1
0x00400020	0x0000000c	syscall	11: syscall
0x00400024	0x21290001	addi $9,$9,1	12: addi $t1, $t1, 1
0x00400028	0x08100004	j 0x00400010	13: j saut
0x0040002c	0x2402000a	addiu $2,$0,10	14: exit: li $v0, 10
0x00400030	0x0000000c	syscall	15: syscall

31

Byte--Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- Each 32--bit words has 4 bytes, so the word address increments by 4. **MIPS uses byte addressable memory**



32

Address Space

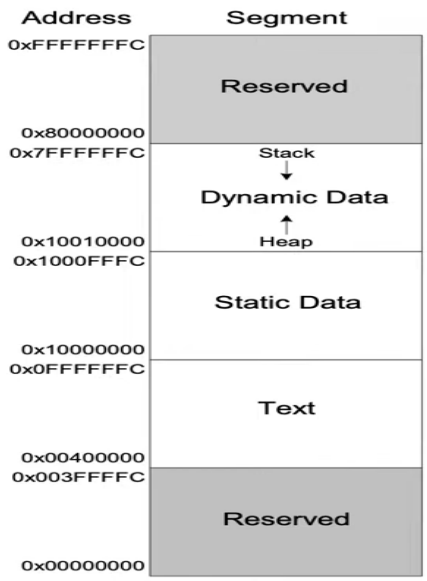
The MIPS address space is divided in four segments:

- Text, which contains the program code
- Data, which contains constants and global variables
- Heap, which contains memory dynamically allocated during runtime
- Stack, which contains temporary data for handling procedure calls

The heap and stack segments grow toward each other, thereby allowing the efficient use of memory as the two segments expand and shrink.

33

Address Space

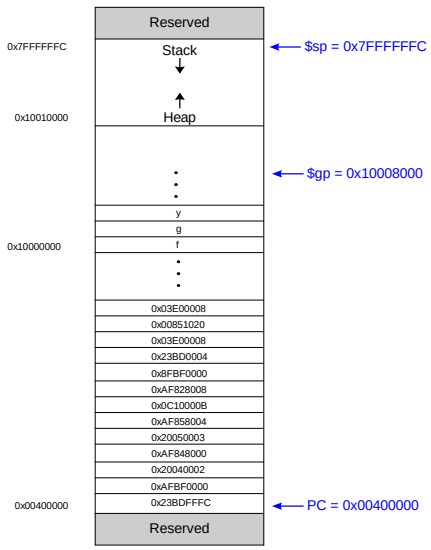


34

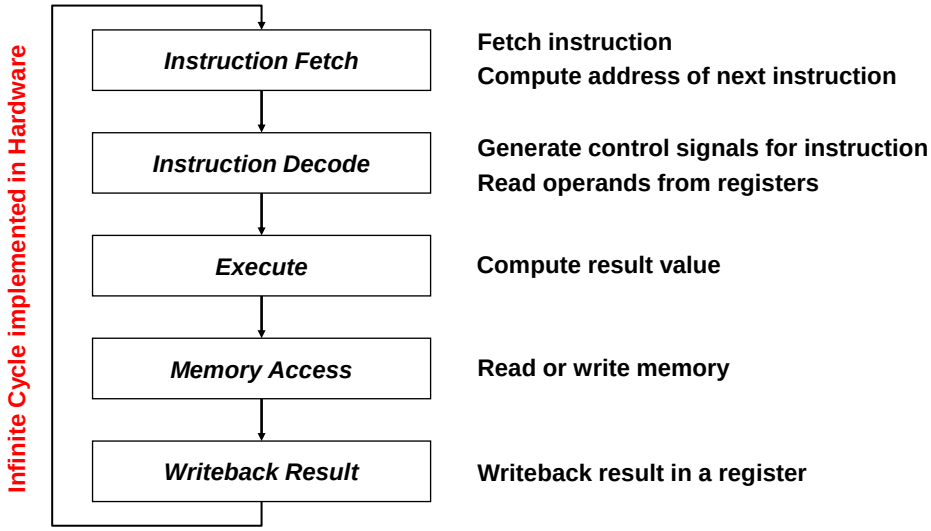
Example Program: Executable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E0008
Data segment	**Address**	**Data**
	0x10000000	f g y
	0x10000004	
	0x10000008	

Example Program: In Memory



Fetch - Execute Cycle



37

MIPS Subset of Instructions

- ALU instructions (R-type): **add, sub, and, or, xor, slt**
- Immediate instructions (I-type): **addi, slti, andi, ori, xori**
- Load and Store (I-type): **lw, sw**
- Branch (I-type): **beq, bne**
- Jump (J-type): **j**

- This subset does not include all the integer instructions
- But sufficient to illustrate design of datapath and control
- Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers

38

Details of the MIPS Subset

Instruction		Meaning	Format					
add	rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub	rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and	rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or	rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor	rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt	rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi	rt, rs, imm^{16}	add immediate	0x08	rs^5	rt^5	imm^{16}		
slti	rt, rs, imm^{16}	slt immediate	0x0a	rs^5	rt^5	imm^{16}		
andi	rt, rs, imm^{16}	and immediate	0x0c	rs^5	rt^5	imm^{16}		
ori	rt, rs, imm^{16}	or immediate	0x0d	rs^5	rt^5	imm^{16}		
xori	rt, imm^{16}	xor immediate	0x0e	rs^5	rt^5	imm^{16}		
lw	rt, imm^{16}(rs)	load word	0x23	rs^5	rt^5	imm^{16}		
sw	rt, imm^{16}(rs)	store word	0x2b	rs^5	rt^5	imm^{16}		
beq	rs, rt, $offset^{16}$	branch if equal	0x04	rs^5	rt^5	$offset^{16}$		
bne	rs, rt, $offset^{16}$	branch not equal	0x05	rs^5	rt^5	$offset^{16}$		
j	$address^{26}$	jump	0x02	$address^{26}$				

Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

Instruction	RTL Description	
ADD	Reg(rd) ← Reg(rs) + Reg(rt);	PC ← PC + 4
SUB	Reg(rd) ← Reg(rs) – Reg(rt);	PC ← PC + 4
ORI	Reg(rt) ← Reg(rs) | zero_ext(imm^{16});	PC ← PC + 4
LW	Reg(rt) ← MEM[Reg(rs) + sign_ext(imm^{16})];	PC ← PC + 4
SW	MEM[Reg(rs) + sign_ext(imm^{16})] ← Reg(rt);	PC ← PC + 4
BEQ	if (Reg(rs) == Reg(rt)) PC ← PC + 4 + 4 × sign_ext($offset^{16}$) else PC ← PC + 4	

40

Instruction Fetch/Execute

▪ R-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(rs), data2 ← Reg(rt)
	Execute operation:	ALU_result ← func(data1, data2)
	Write ALU result:	Reg(rd) ← ALU_result
	Next PC address:	PC ← PC + 4
▪ I-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(rs), data2 ← Extend(imm^{16})
	Execute operation:	ALU_result ← op(data1, data2)
	Write ALU result:	Reg(rt) ← ALU_result
	Next PC address:	PC ← PC + 4
▪ BEQ	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(rs), data2 ← Reg(rt)
	Equality:	zero ← subtract(data1, data2)
	Branch:	if (zero) PC ← PC + 4 + 4×sign_ext($offset^{16}$)
		else PC ← PC + 4

41

Instruction Fetch/Execute

▪ LW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch base register:	base ← Reg(rs)
	Calculate address:	address ← base + sign_extend(imm^{16})
	Read memory:	data ← MEM[address]
	Write register Rt:	Reg(rt) ← data
	Next PC address:	PC ← PC + 4
▪ SW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch registers:	base ← Reg(rs), data ← Reg(rt)
	Calculate address:	address ← base + sign_extend(imm^{16})
	Write memory:	MEM[address] ← data
	Next PC address:	PC ← PC + 4
▪ Jump	Fetch instruction:	Instruction ← MEM[PC]
	Target PC address:	target ← PC[31:28] || $address^{26}$ || '00'
	Jump:	PC ← target

42

Requirements of the Instruction Set

- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 31 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and Adder to increment PC
- Sign and Zero extender for immediate constant
- ALU for executing instructions

Pipelining

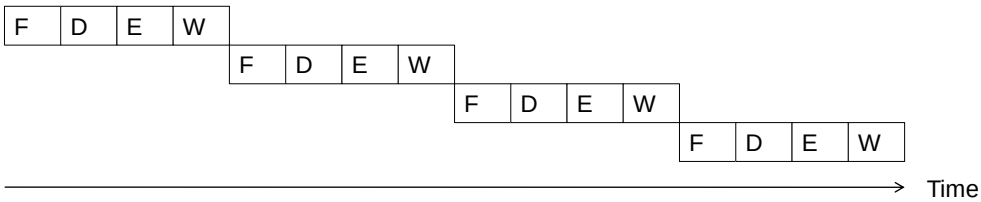
Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)

45

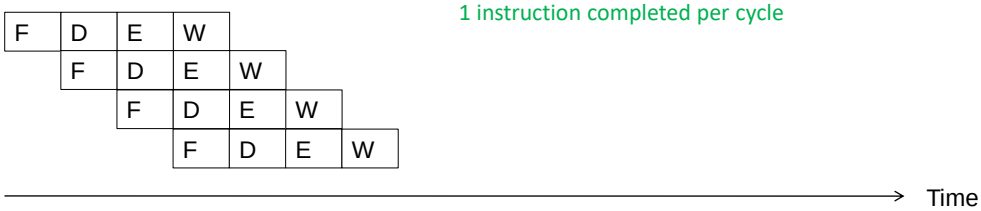
Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction



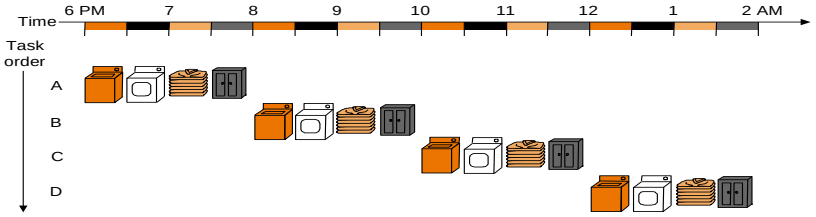
- Pipelined: 4 cycles per 4 instructions

1 instruction completed per cycle



46

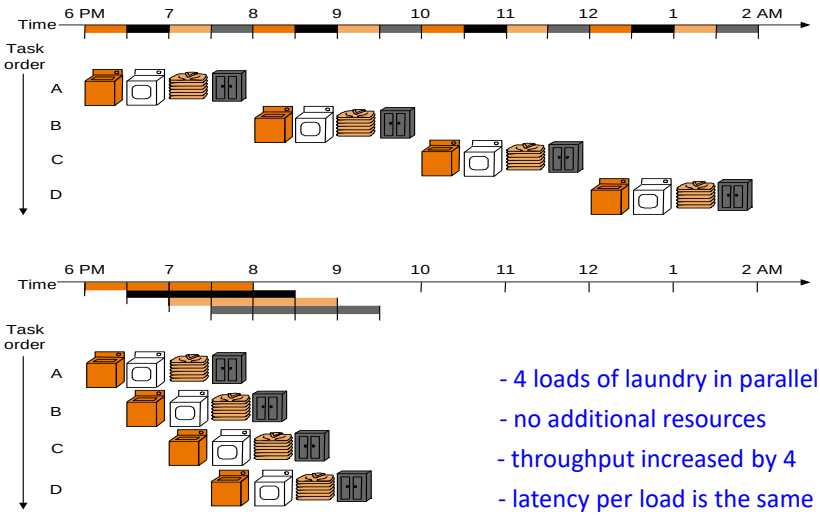
The Laundry Analogy



- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

47

Pipelining Multiple Loads of Laundry



- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

48

Remember: The Instruction Processing Cycle

- 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.

49

Pipelining and ISA Design

MIPS ISA designed for pipelining

- All instructions are 32-bits
  - Easier to fetch in one cycle
- 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

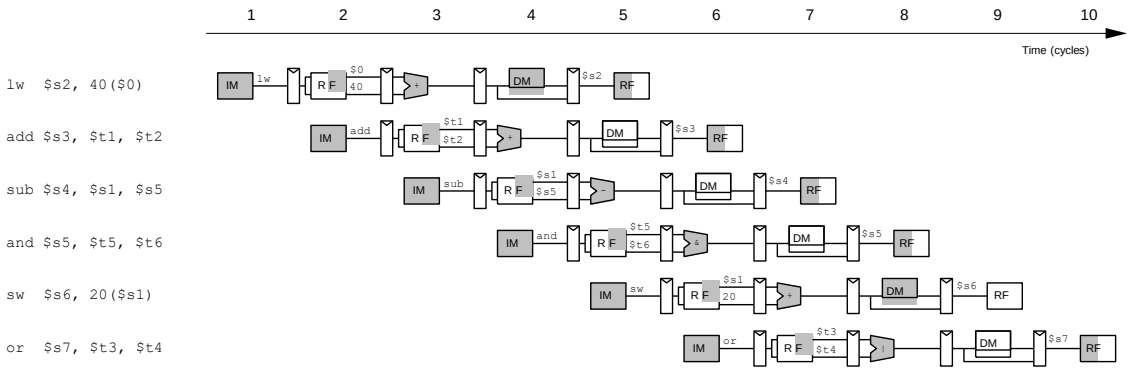
50

Remember: The Instruction Processing Cycle

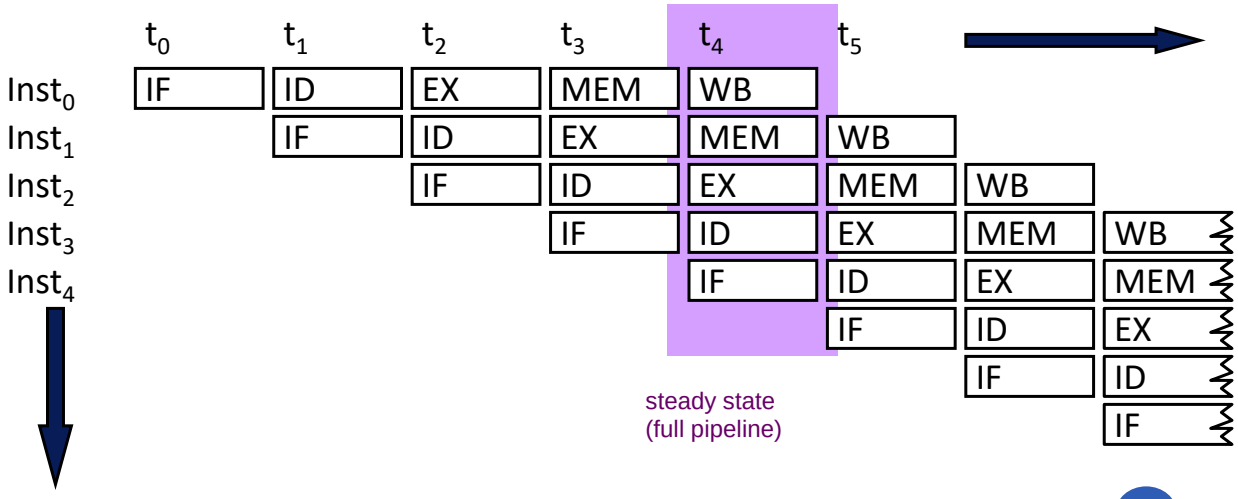
However, as we saw, not all instructions need all five steps.

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

Pipelining Abstraction



Illustrating Pipeline Operation: Operation View



53

Illustrating Pipeline Operation: Resource View

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

54

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!
  - ⇒ different instructions → not all need the same stages
    - Forcing different instructions to go through the same pipe stages
    - → external fragmentation (some pipe stages idle for some instructions)
- Uniform suboperations ... NOT!
  - ⇒ different pipeline stages → not the same latency
    - Need to force each stage to be controlled by the same clock
    - → internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)
- Independent operations ... NOT!
  - ⇒ instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependences to ensure the pipeline provides correct results
    - → pipeline stalls (pipeline is not always moving)

55

Pipelining Hazards

- There are situations in pipelining when the next instruction can not execute in the following clock cycle. These events are called hazards
- In other word, any condition that causes a pipeline to stall is called a hazard.
- There are three types of hazards:
  - Structural hazards:
    - A required resource is busy
  - Data hazards:
    - Need to wait for previous instruction to complete its data read/write
  - Control hazards:
    - Deciding on control action depends on previous instruction

56

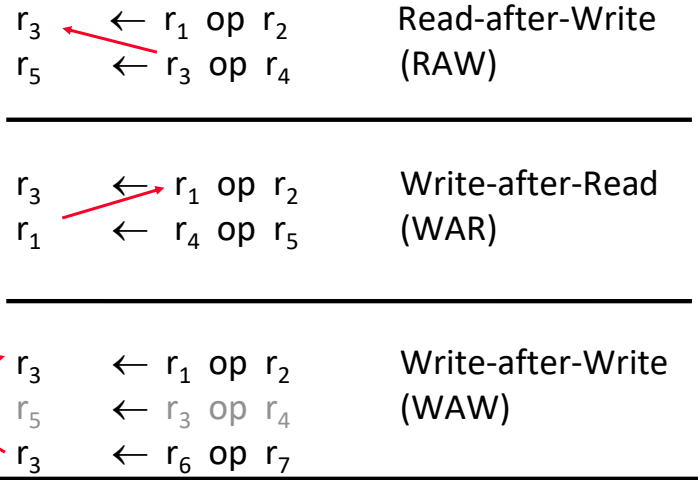
Structural hazard

- Caused by limitations in hardware that don't allow concurrent execution of different instructions
- Examples
  - Bus
  - Single ALU
  - Single Memory for instructions and data
  - Single IR
- Remedy is to add additional elements to datapath to eliminate hazard

57

Data Hazards Types

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



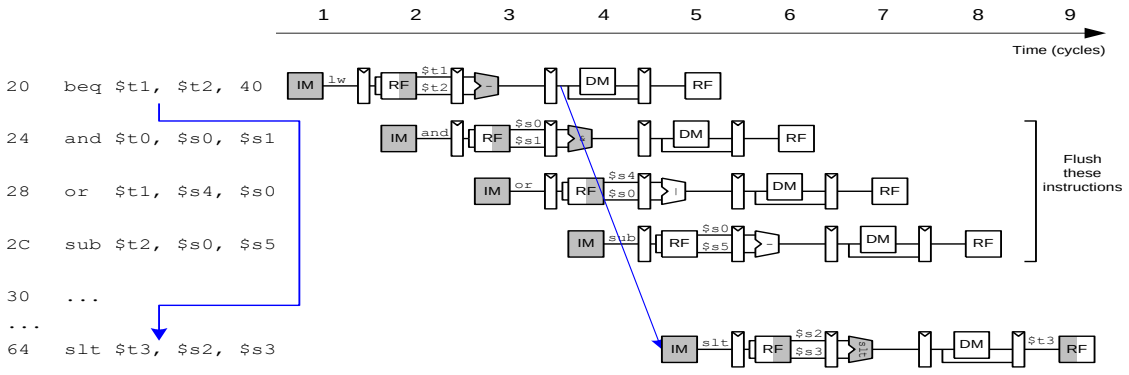
58

Control Hazard

- Special case of data dependence: dependence on PC
- beq:
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch is resolved
    - Always predict that the next sequential instruction is fetched
    - Called “Always not taken” prediction
  - These instructions must be flushed if the branch is taken
- Branch misprediction penalty
  - **number of instructions flushed when branch is taken**
  - May be reduced by determining branch earlier

59

Control Hazard



60

Causes of Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependencies (between instructions)
  - Data hazard
  - Control hazard
- Long-latency (multi-cycle) operations

61

How Can You Handle Data Hazards?

- **Insert "NOP"s (No OPeration) in code at compile time**
- **Rearrange code at compile time**
- **Forward data at run time**
- **Stall the processor at run time**

62

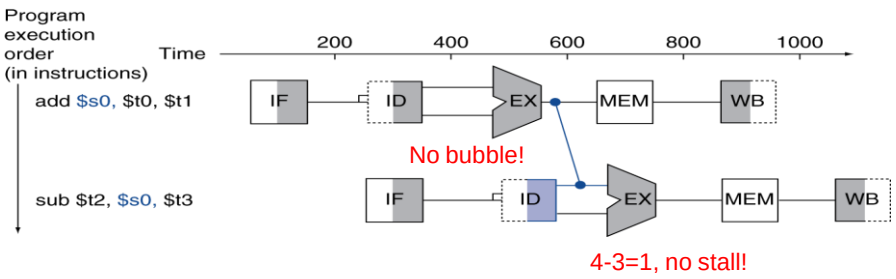
Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

63

Data Forwarding/Bypassing

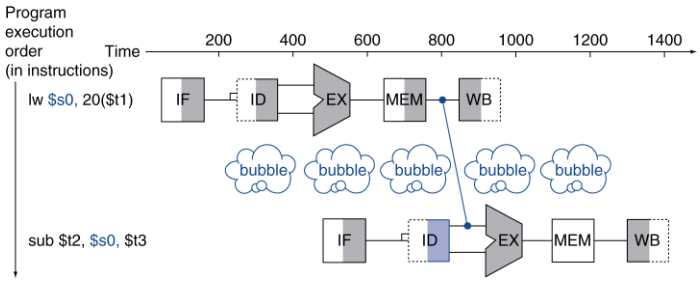
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



64

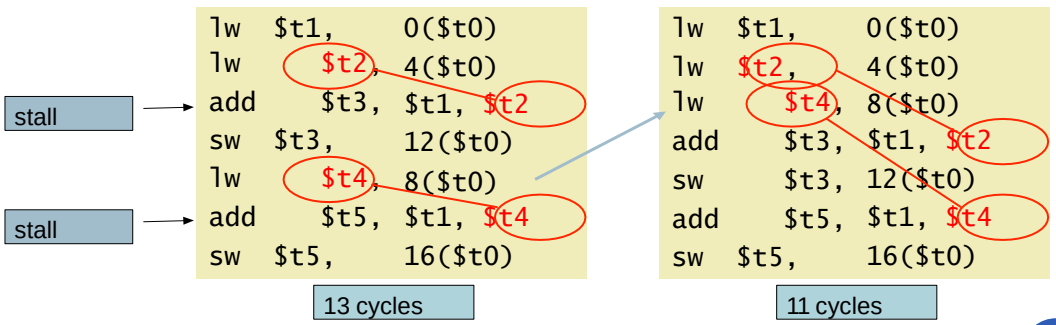
Data Forwarding/Bypassing

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



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;`



Stall on Branch

- In MIPS pipeline
  - Need to compare registers and compute target earlier in the pipeline
  - Add extra hardware to do it in ID stage (earliest ?)
- Wait until branch outcome determined before fetching next instruction
- 1 bubble when determine in ID
- Is no stall possible? IF, prediction

67