الجامعة اللبنانية الأميركية
Lebanese American University

Processor Structure & Function

Chapter 16

Based on:

William Stallings
Computer Organization and Architecture, 11th Global Edition

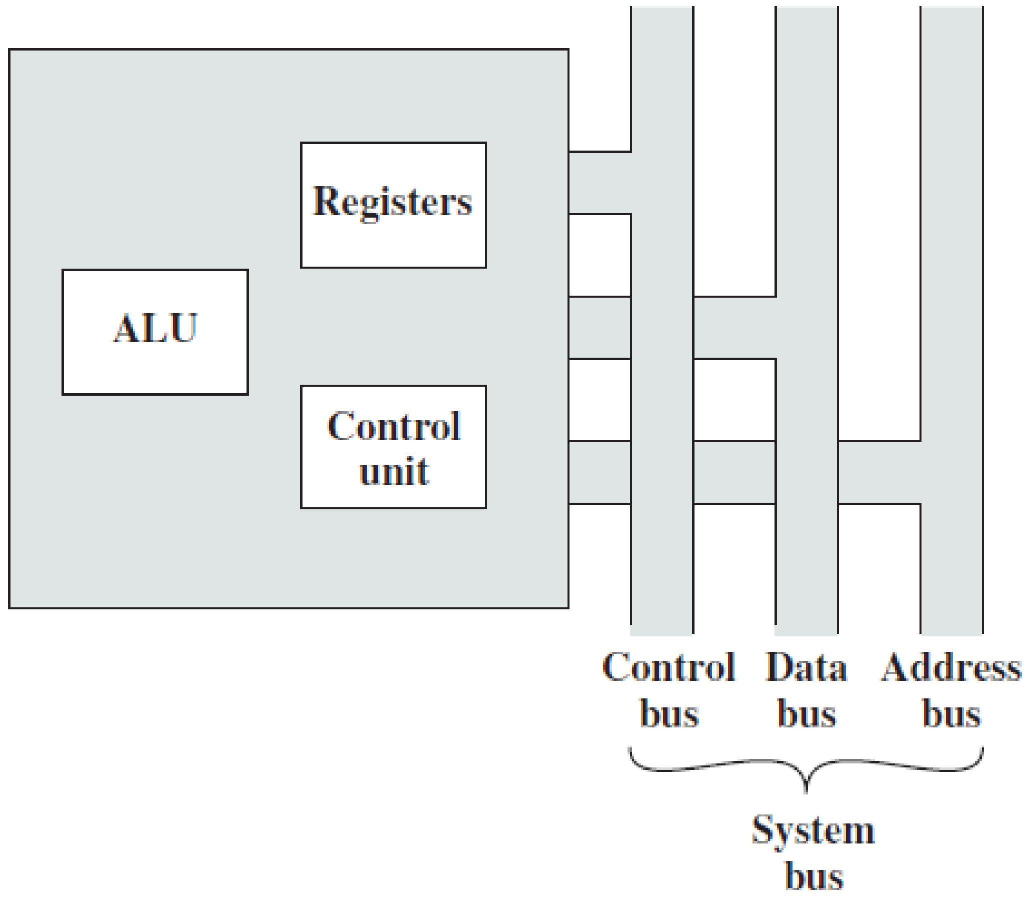
Patterson and Hennessy
Computer Organization and Design, 5th Edition

Processor Organization

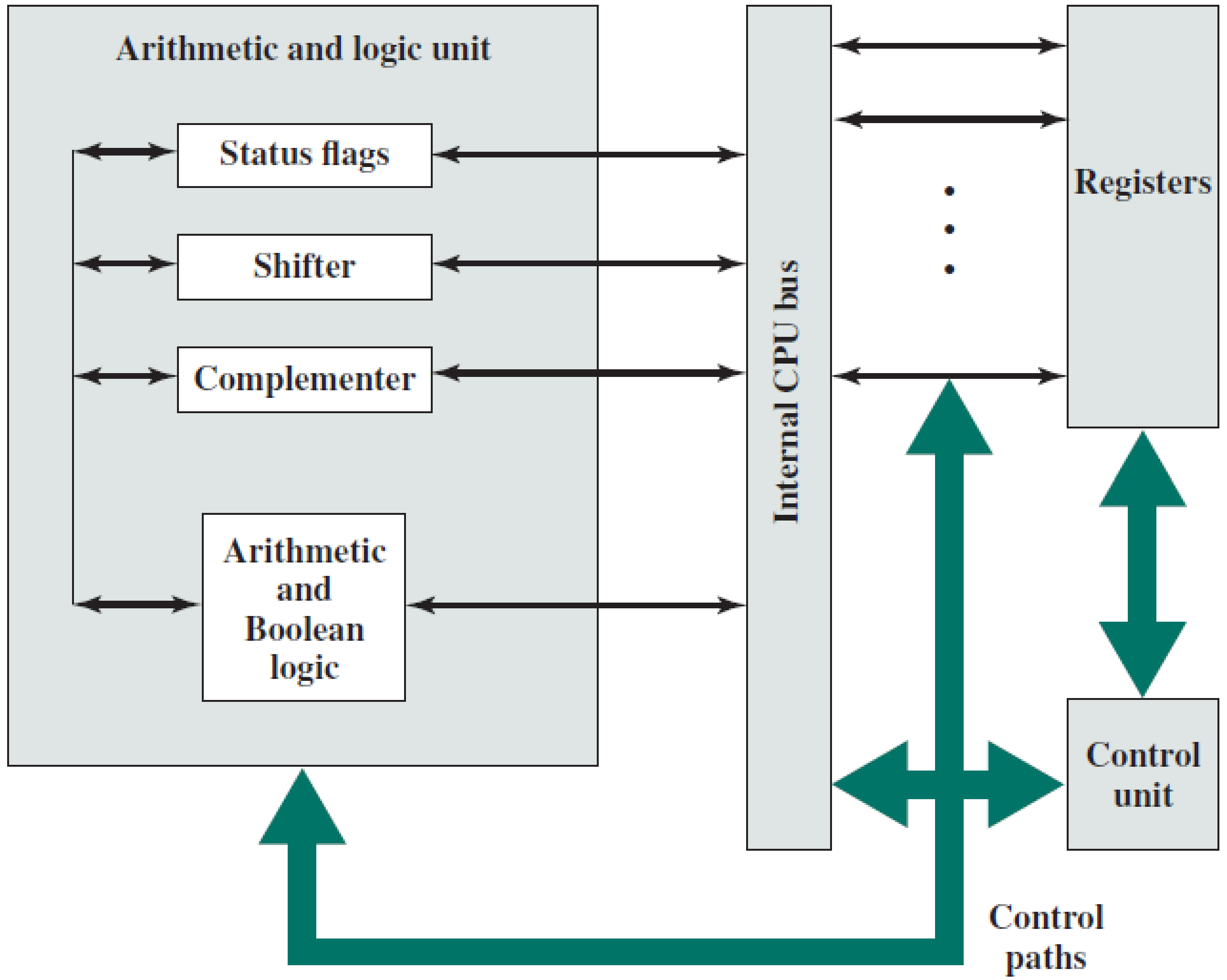
Processor Requirements:

- Fetch instruction
  - Processor reads an instruction from memory (register, cache, main memory)
- Interpret instruction
  - Instruction is decoded to determine what action is required
- Fetch data
  - Execution of an instruction may require reading data from memory or an I/O module
- Process data
  - Execution of an instruction may require performing some arithmetic or logical operation on data
- Write data
  - Results of an execution may require writing data to memory or an I/O module
- Processor needs to store some data temporarily and therefore needs a small internal memory

The CPU with the System Bus



Internal Structure of the CPU



Register Organization

- Within the processor there is a set of registers that function as a level of memory above main memory and cache in the hierarchy

- The registers in the processor perform two roles:
  - User-Visible Registers
    - Enable the machine or assembly language programmer to minimize main memory references by optimizing use of registers
  - Control and Status Registers
    - Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs

User-Visible Registers

- Referenced by means of the machine language that the processor executes
- Categories:
  - General purpose
    - Can be assigned to a variety of functions by the programmer
  - Data
    - May be used only to hold data and cannot be employed in the calculation of an operand address
  - Address
    - May be somewhat general purpose or may be devoted to a particular addressing mode
    - Examples: segment pointers, index registers, stack pointer
  - Condition codes
    - Also referred to as *flags*
    - Bits set by the processor hardware as the result of operations

Control and Status Registers

Four registers are essential to instruction execution:

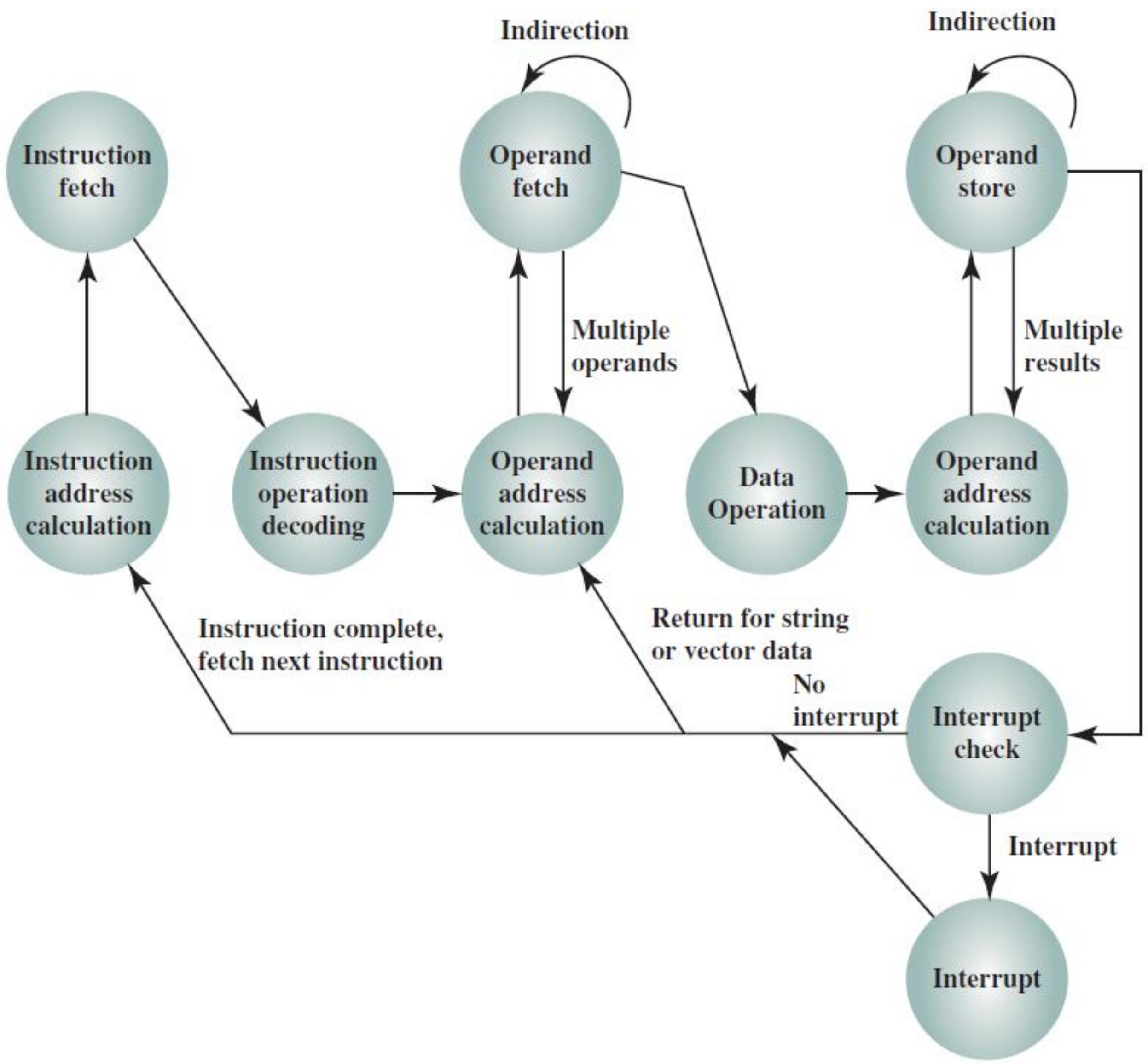
- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Memory address register (MAR)
  - Contains the address of a location in memory
- Memory buffer register (MBR)
  - Contains a word of data to be written to memory or the word most recently read

Program Status Word (PSW)

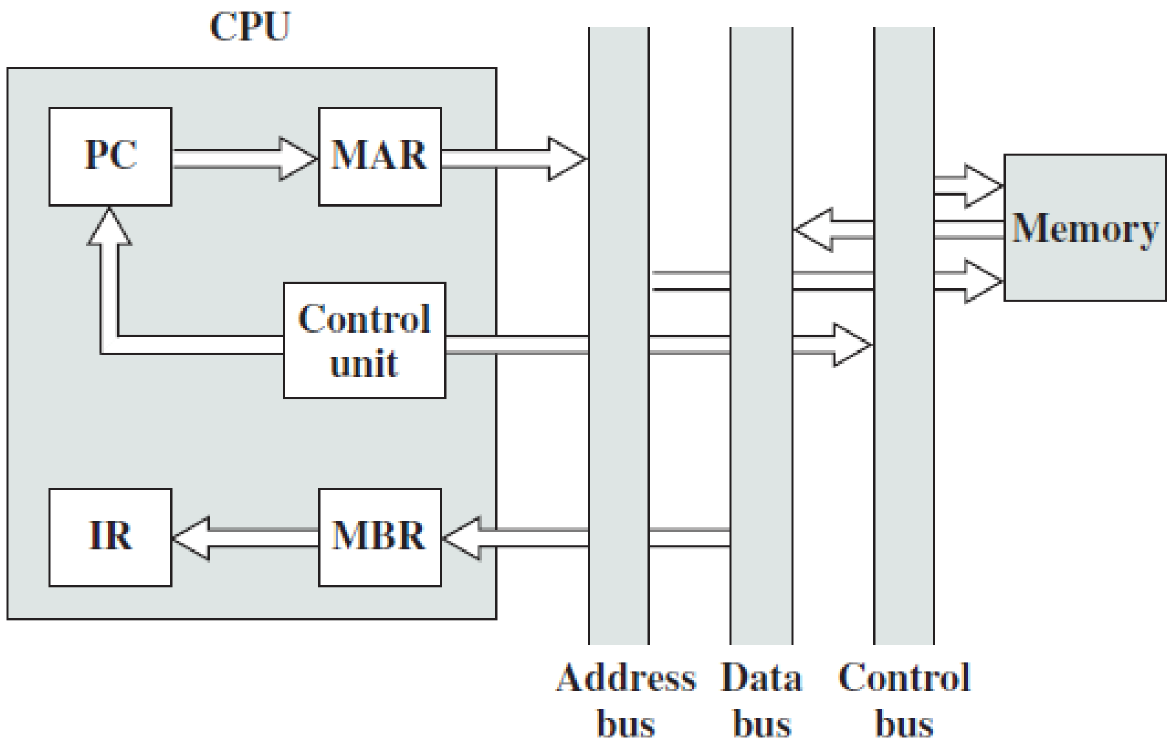
- Register or set of registers that contain condition codes plus other status information

- Common fields or flags include:
  - Sign
  - Zero
  - Carry
  - Equal
  - Overflow
  - Interrupt Enable/Disable
  - Supervisor

Instruction Cycle State Diagram

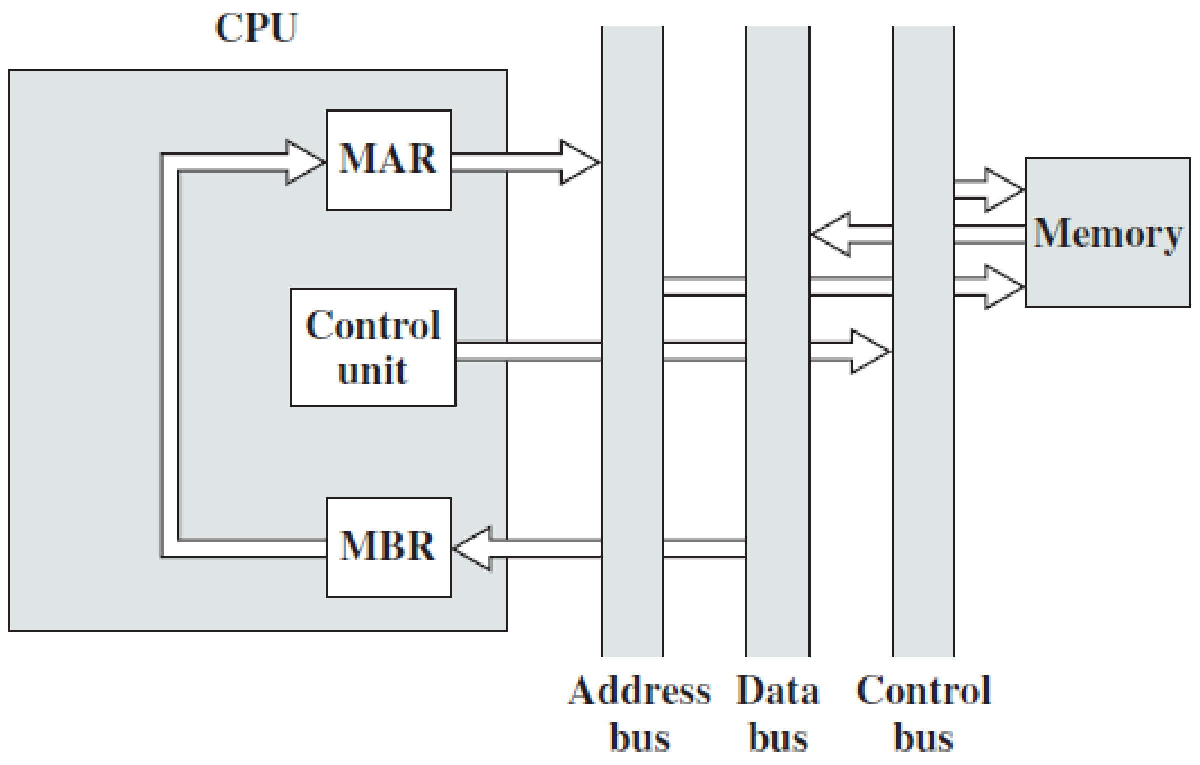

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt
Interrupt check
Interrupt
Interrupt

Data Flow, Fetch Cycle

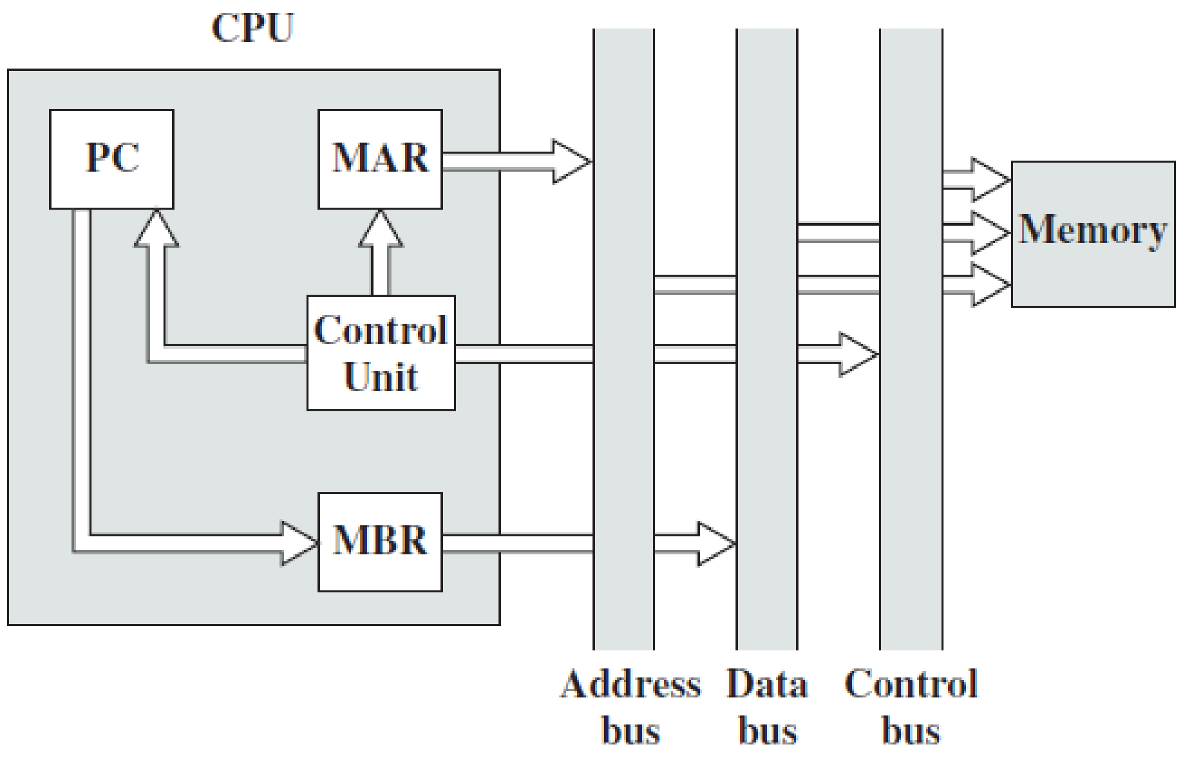


MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Data Flow, Indirect Cycle



Data Flow, Interrupt Cycle

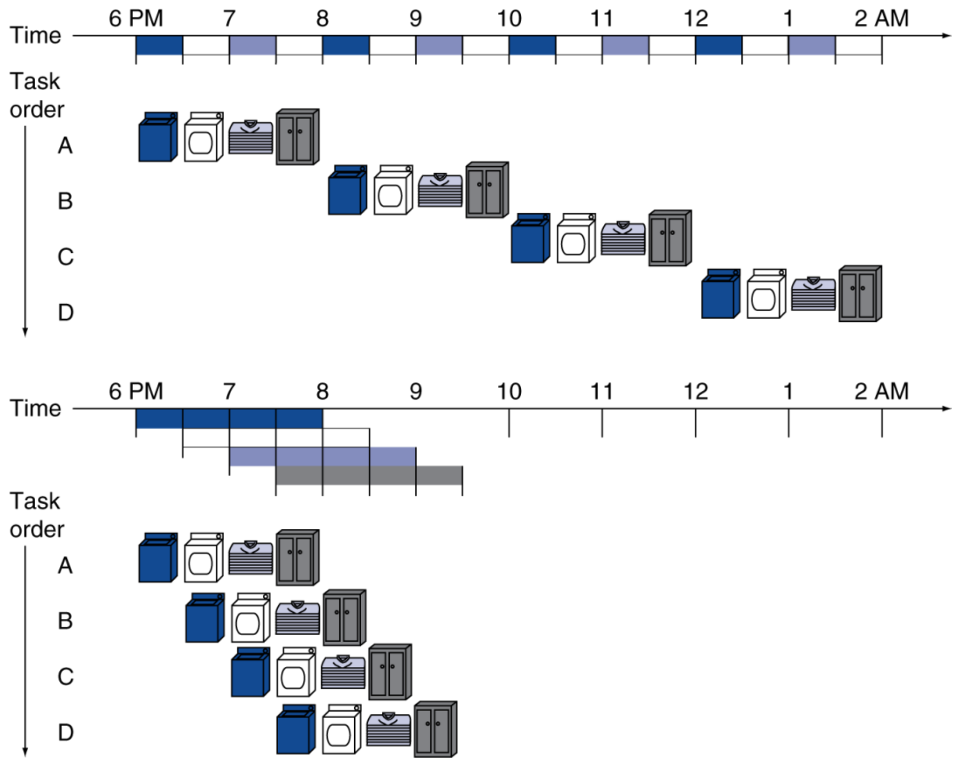


Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

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

- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline (4 in this case)



Instruction Stages

- Fetch instruction (FI)
  - Read the next expected instruction into a buffer
- Decode instruction (DI)
  - Determine the opcode and the operand specifiers
- Calculate operands (CO)
  - Calculate the effective address of each source operand
  - This may involve displacement, register indirect, indirect, or other forms of address calculation
- Fetch operands (FO)
  - Fetch each operand from memory
  - Operands in registers need not be fetched
- Execute instruction (EI)
  - Perform the indicated operation and store the result, if any, in the specified destination operand location
- Write operand (WO)
  - Store the result in memory

Case Study: MIPS Pipeline

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

- Each of these five steps will take one clock cycle to complete

Pipeline Performance

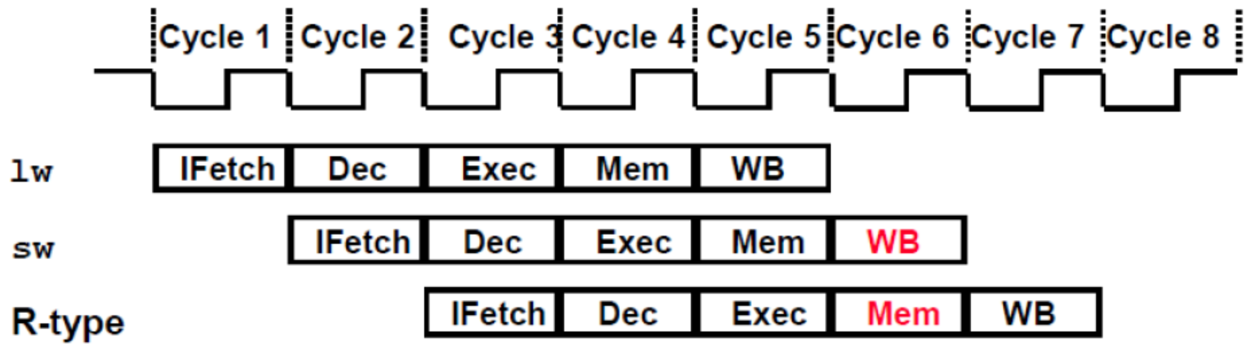
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

- 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

Pipelined MIPS Processor

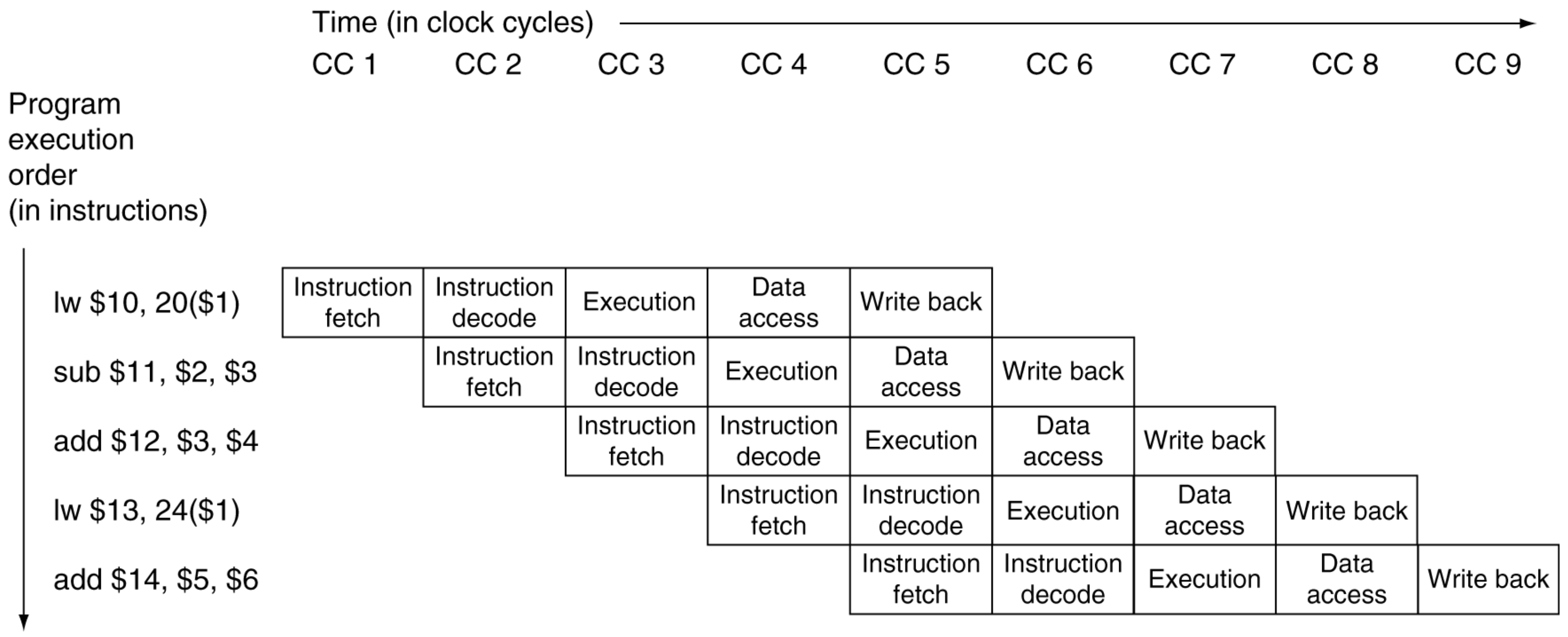
- The next instruction starts before the current one completes
  - Improves throughput: total amount of work done in a given time
  - Instruction latency (time from the start of an instruction to its completion) is not reduced



- Clk cycle (pipeline stage time) is limited by the slowest stage
  - Some stages do not need the whole clk cycle
  - For some instructions, some stages are wasted (nothing is done during that cycle for that instruction)

Multi-Cycle Pipeline Diagram

Traditional form

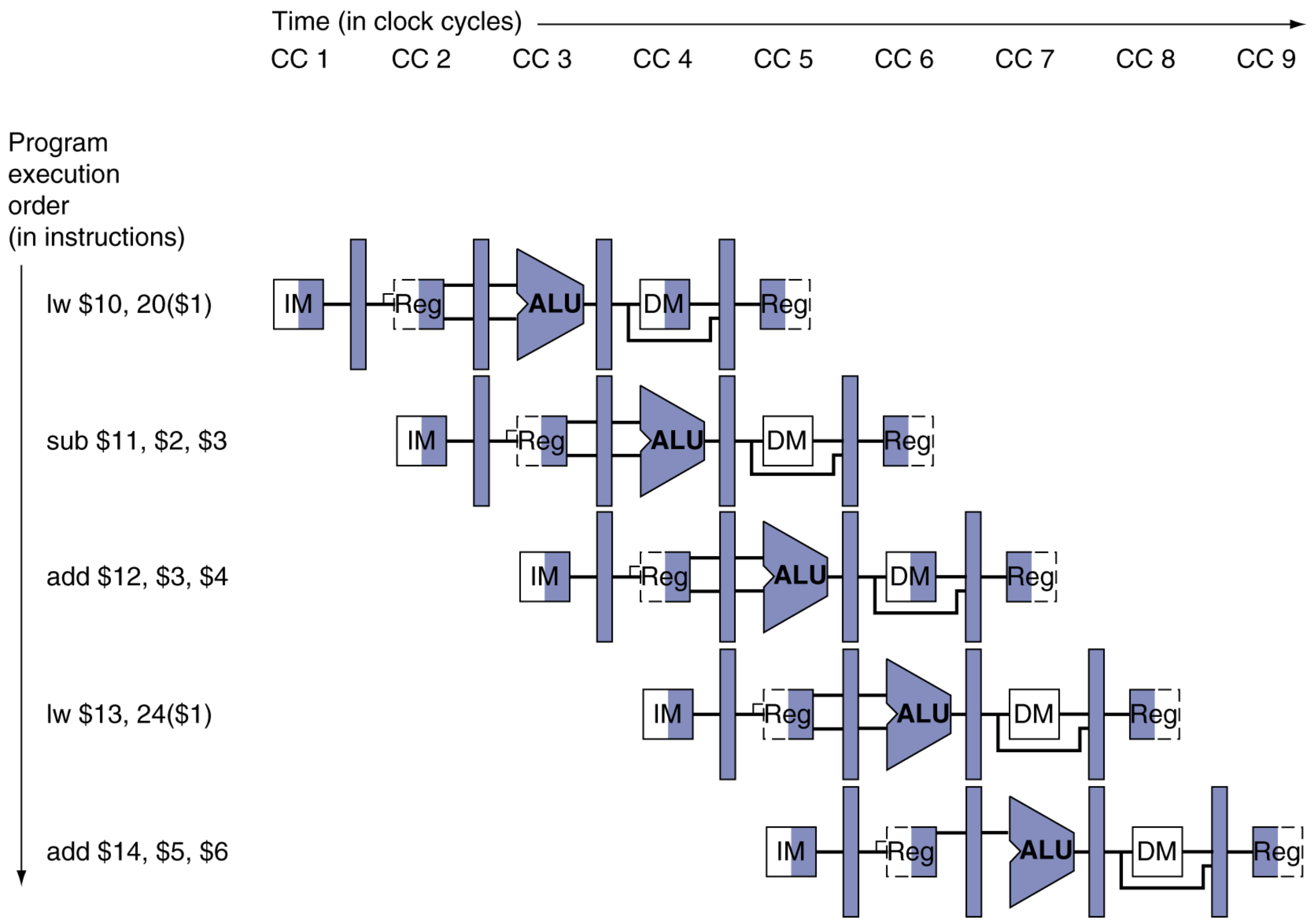


Time (in clock cycles) →

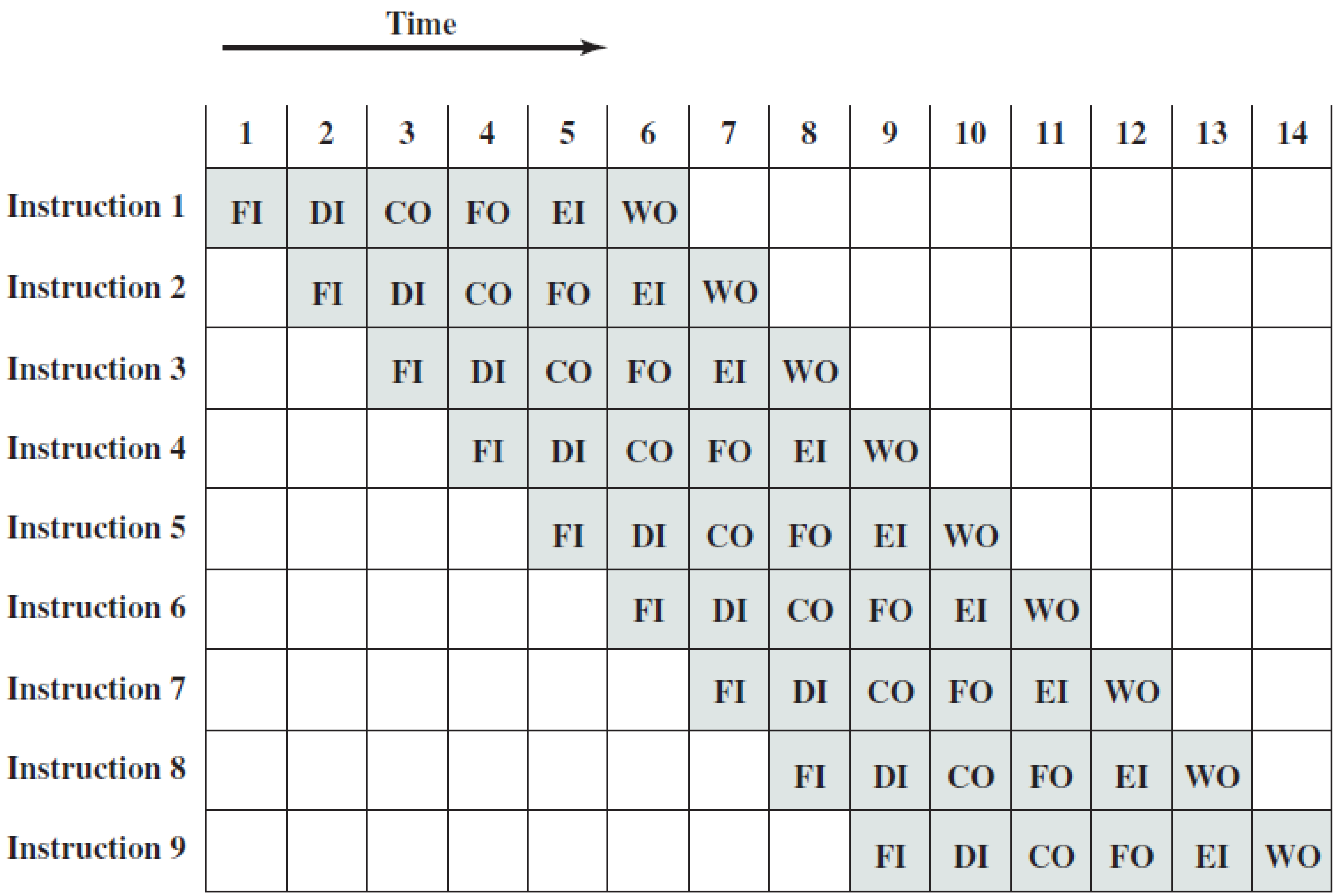
Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Multi-Cycle Pipeline Diagram

Form showing resource usage

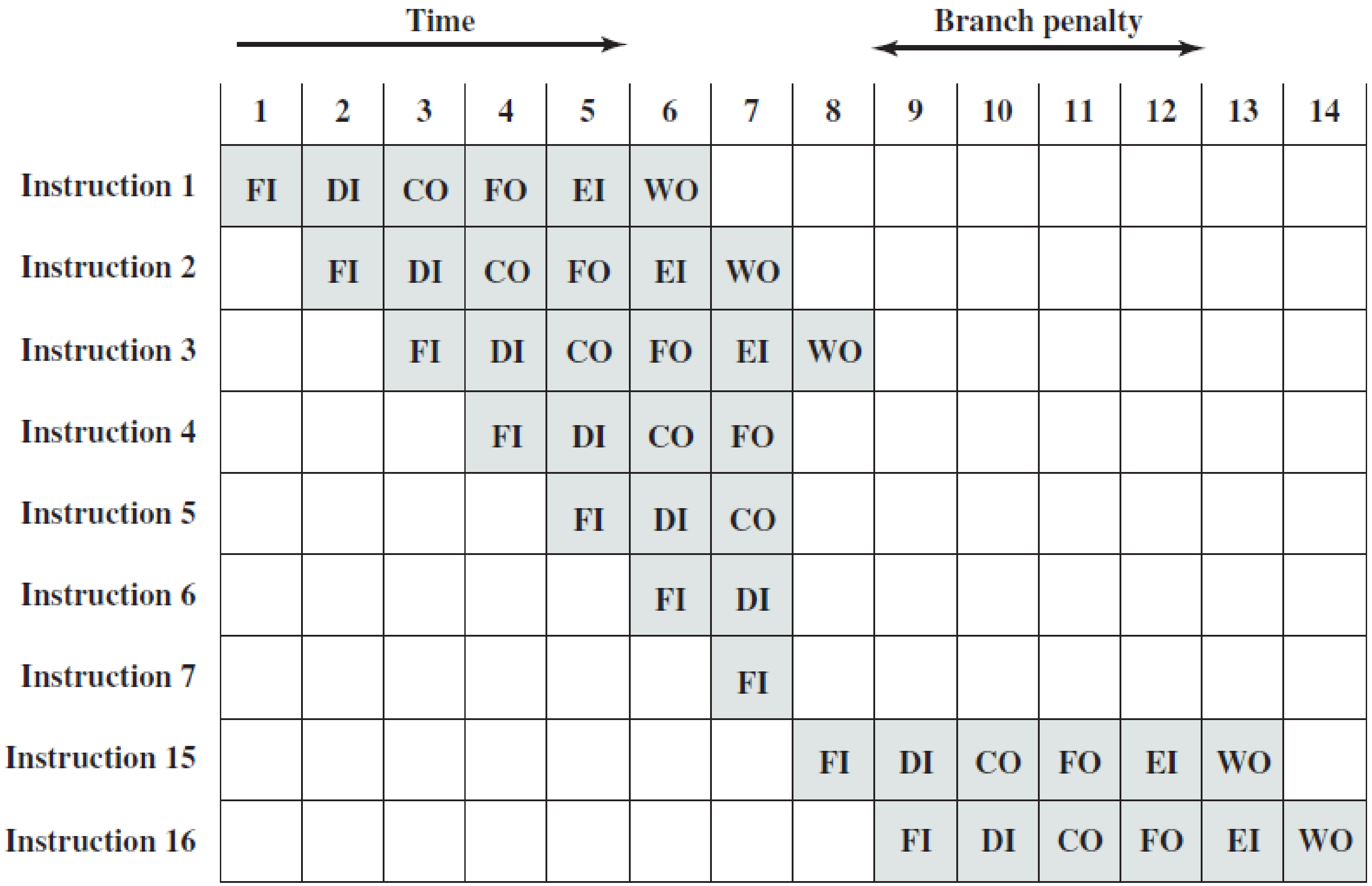


Timing Diagram for Instruction Pipeline Operation



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Effect of a Conditional Branch on Instruction Pipeline Operation


Time
Branch penalty

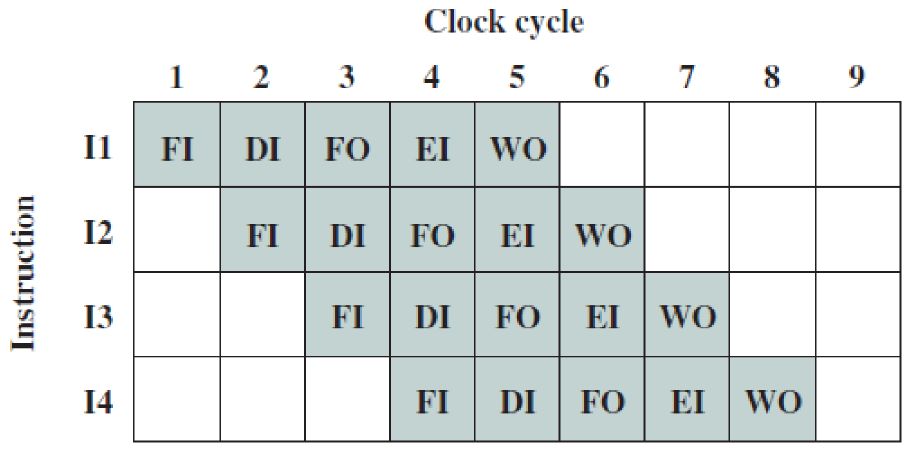
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Hazards

- Situations that prevent starting the next instruction in the next cycle
- **Structure hazard**
  - A required resource is busy
- **Data hazard**
  - Attempt to use data before it’s ready
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction
  - Attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated

Structure Hazards

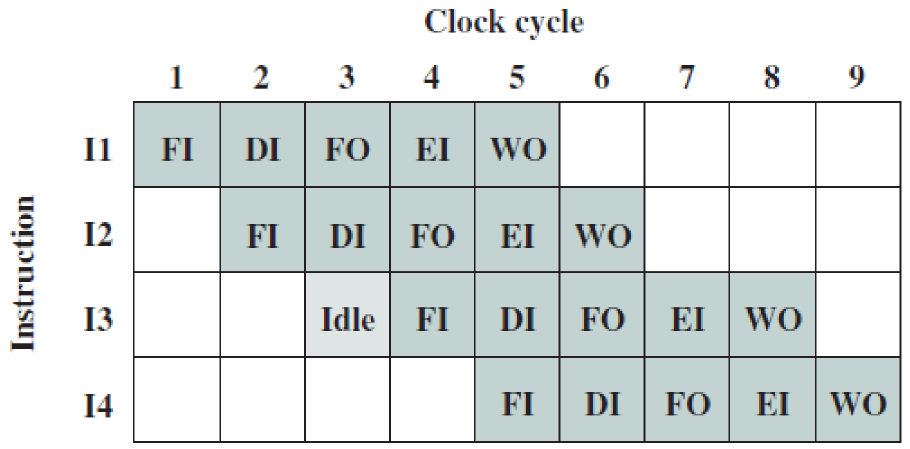
- Conflict for use of a resource
- With a single memory
  - Load/store requires data access
  - Instruction fetch would have to **stall** for that cycle
    - Would cause a pipeline “bubble” (nop)



Clock cycle

Instruction	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			FI	DI	FO	EI	WO		
I4				FI	DI	FO	EI	WO	

(a) Five-stage pipeline, ideal case



Clock cycle

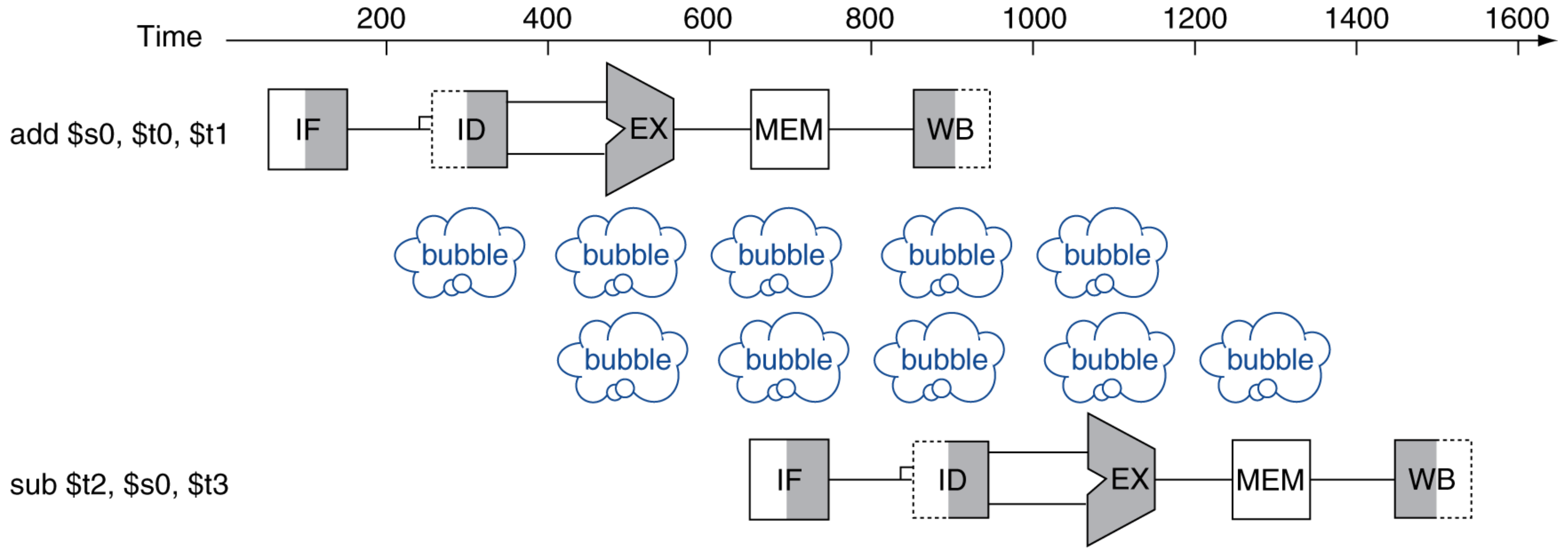
Instruction	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			Idle	FI	DI	FO	EI	WO	
I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

Data Hazards

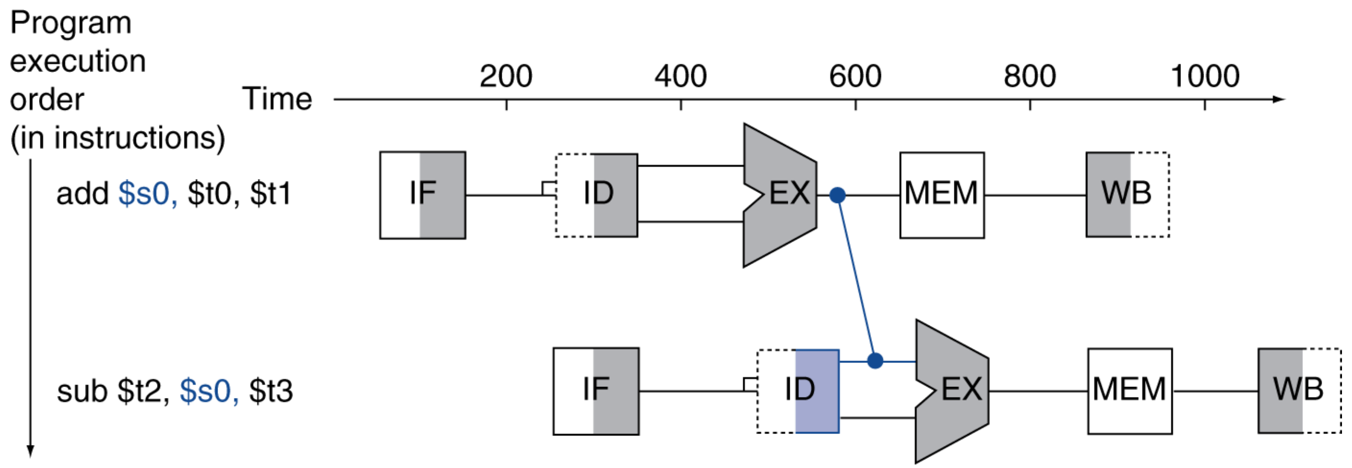
- An instruction depends on completion of data access by a previous instruction

  -
    ```
    add    $s0, $t0, $t1
    sub    $t2, $s0, $t3
    ```



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



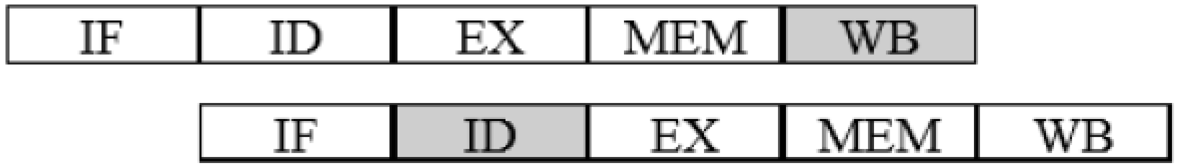
Types of Data Hazards

- **RAW**: Read After Write (**true dependency**)
- **WAR**: Write After Read (**anti-dependency**)
- **WAW**: Write After Write (**output dependency**)
- **RAR**: Read After Read (not a hazard)

RAW - Example

- Extremely common
- Instruction 2 tries to read an operand before instruction 1 writes to it

- Example of the dependency:
  - I1: add R1, R2, R3 # I1 writes to R1
  - I2: add R5, R1, R4 # I2 reads from R1



WAR - Example

- Instruction 2 tries to write to a destination before it is read by instruction 1.

- Example of the dependency:
  - I1: add R4, R1, R5   # I1 reads from R5
  - I2: add R5, R1, R2   # I2 writes to R5

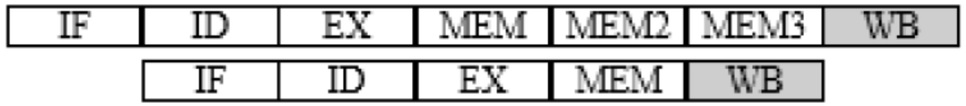
- In the situation where I2 might finish before I1 (e.g. concurrent execution), it must be ensured that the result of R5 is not stored before I1 has had the chance to fetch it.

- WAR hazard is uncommon/impossible in a reasonable (in-order) pipeline

WAW - Example

- Instruction 2 tries to write to an operand before instruction 1 writes to it

- Example of the dependency:
  - I1: add R1, R2, R3   # I1 writes to R1
  - I2: add R1, R2, R4   # I2 writes to R1



IF	ID	EX	MEM	MEM2	MEM3	WB
	IF	ID	EX	MEM	WB	

- The WB of I2 must be delayed until I1 finishes executing

- WAW hazard possible in a reasonable pipeline, but not in the very simple pipeline we're assuming

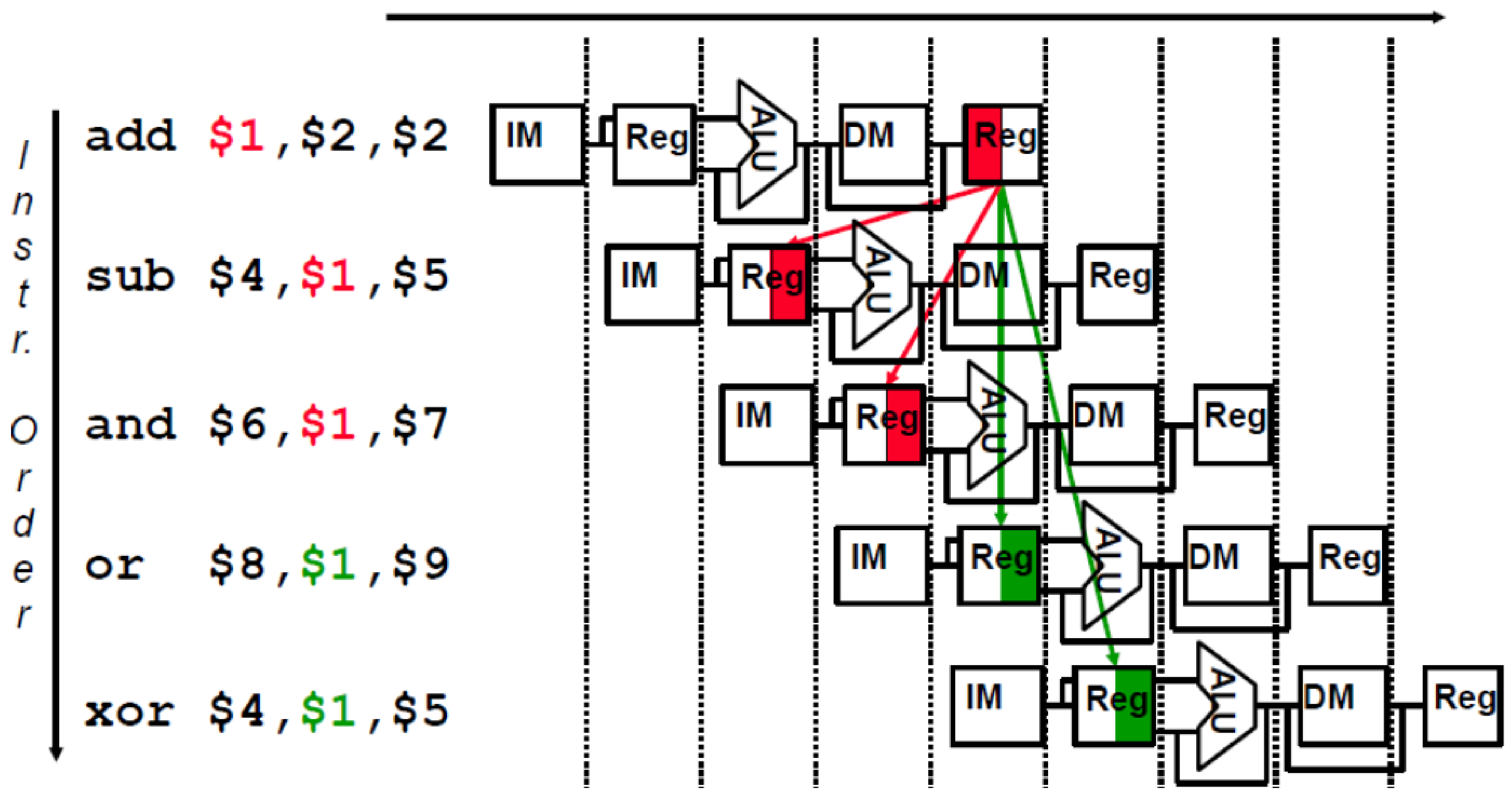
RAR - Example

- Not a hazard

- Example of the dependency:
  - I1: add R1, R2, R3 # I1 reads from R2
  - I2: add R5, R4, R2 # I2 reads from R2

- Register value does not change
- The order of the two reads is not important

Example


Instr. Order
add $1,$2,$2
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5
IM
Reg
ALU
DM
Reg

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
- Occurs when the pipeline makes the wrong decision on a branch prediction
- Brings instructions into the pipeline that must subsequently be discarded
- Dealing with Branches:
  - Stall on branch
  - Multiple streams
  - Prefetch branch target
  - Loop buffer
  - Branch prediction