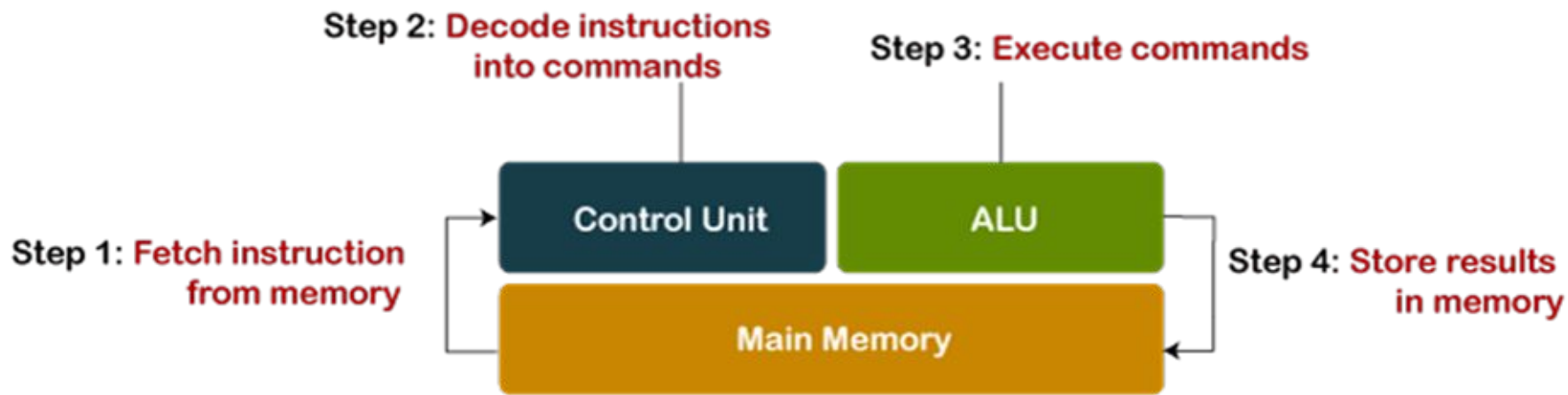


# Unit IV

# Syllabus

*Basic processing unit, ALU operations, Instruction execution, Branch instruction, Multiple bus organization, Hardwired control, Generation of control signals, Micro-programmed control; Pipelining: Basic concepts of pipelining, Performance, Hazards-Data, Instruction and Control, Influence on instruction sets.*

## Machine Cycle



# Basic Processing unit

- Processing unit - executes machine instructions and coordinates the other functions of other units.
- This is often named as instruction processor/processor.
- Its internal structure is evaluated here and how it fetches, decodes and executes instructions.
- The processing unit was called as the central processing unit.
- Since modern computers have many processors, it is not called as central.
- The advancement in technology, makes processors act in parallel as much as possible.

# Basic Processing unit

- High performance processors have pipelined organization.
- pipelined organization - execution of instruction starts before preceding instruction.
- Superscalar operation - several instructions are fetched and executed at the same time;
- A typical computing task - series of task specified by a sequence of instructions called program.

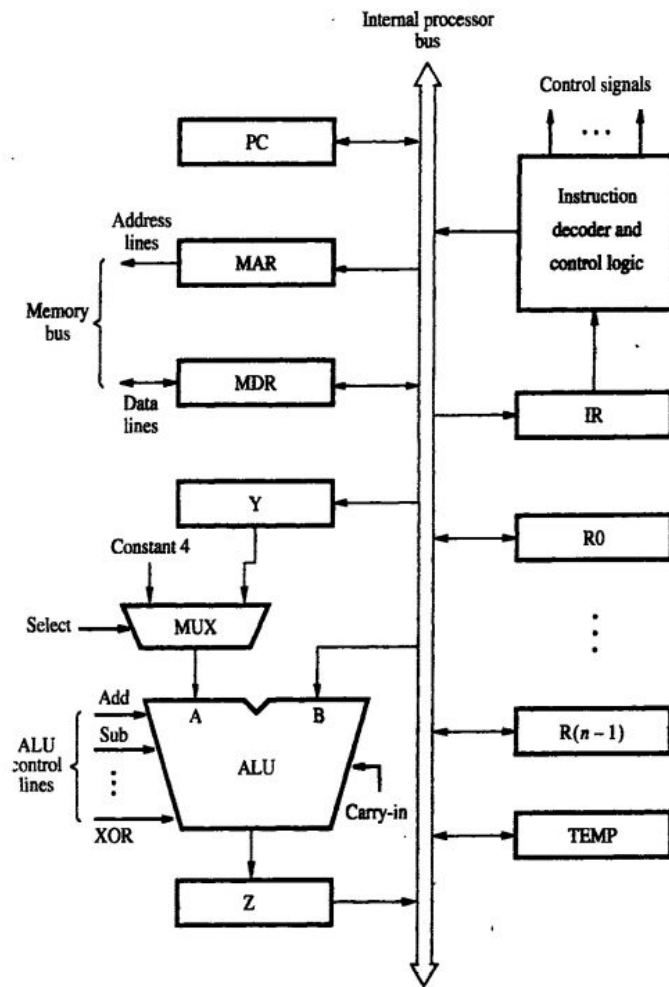


Figure 7.1 Single-bus organization of the datapath inside a processor.

# Basic Processing unit

- ALU is interconnected to other registers through this internal bus.
- Data and address lines of the external memory are connected through MDR and MAR respectively.
- MDR - 2 inputs and 2 outputs.
- Data is loaded into MDR either through memory bus/internal processor bus.
- Input to MAR is connected to internal bus and output is connected to external bus.
- Control lines of memory are connected to instruction decoder and control logic block.
  - Issues the control signal that controls the operation of all the units inside the processor and for interacting with the processor.

- The number and use of registers  $R0$  to  $R(n-1)$  varies considerably from one processor to another.
- Registers - general purpose by the programmer.
- Some registers are dedicated as special purpose registers - index registers/stack registers.
- $Y, Z, TEMP$  - Not transparent to the programmer.
  - Used by processor for temporary storage during execution of instructions.
  - Never used for storing data that are later use by another instruction.



- Multiplexer MUX, selects output of register Y/constant value 4 - which is input A to ALU.
- 4 - increments the content of PC
- MUX control input SELECT
  - Select4 and SelectY

- MUX control input SELECT
  - Select4 and SelectY
- As instruction execution progresses, data are transferred from one register to another passing through ALU.
- Instruction decoder and control unit - implements the actions specified by instruction in the IR register.
- Decoder generates the signal needed to select the registers involved and directs the transfer of data.
- Registers + ALU + interconnecting bus = datapath

- With few exceptions, the sequence of operations are:
  - Transfer a word of data from one processor to another/ALU
  - Perform ALU operation and store result in processor register
  - Load contents from memory location to processor register.
  - Store a word of data from processor register into memory location.

# Register transfers

- During instruction execution, a sequence of steps are executed in which data are transferred from one register to another.
- Each register - 2 control signals.
  1. Place the contents of register on the bus
  2. Load data on the bus to register.
- Input switch -  $Ri_{in}$  and
- Output switch -  $Ri_{out}$
- when  $Ri_{in}$  is 1, data on bus are loaded into  $Ri_{in}$
- when  $Ri_{out}$  is 1, content of Register  $Ri$  are placed on bus.
- While  $Ri_{out} = 0$ , the bus can be used for transferring data from other registers.

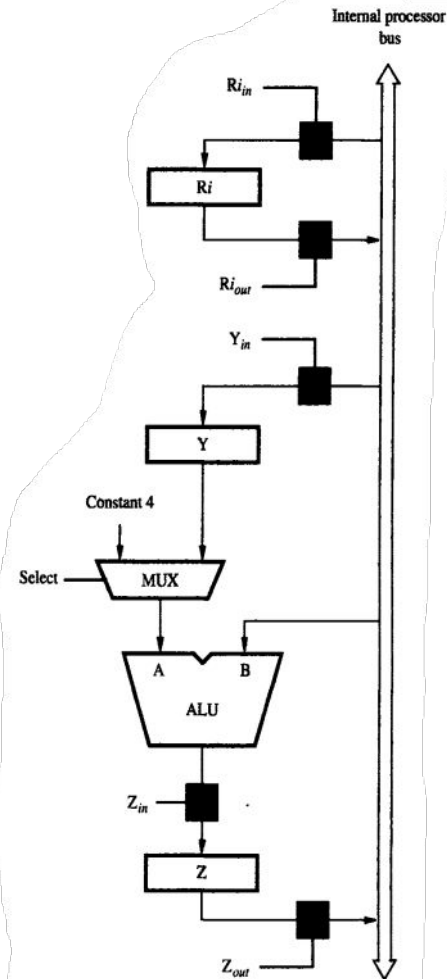


Figure 7.2 Input and output gating for the registers in Figure 7.1.

# Register transfers

- Transferring data from R1 to R4
  - Enable o/p of R1,  $R1_{out}$  to 1 → places the contents of R1 on the processor bus.
  - Enable i/p of R4,  $R4_{in}$  to 1 → loads the data from the processor bus into R4.

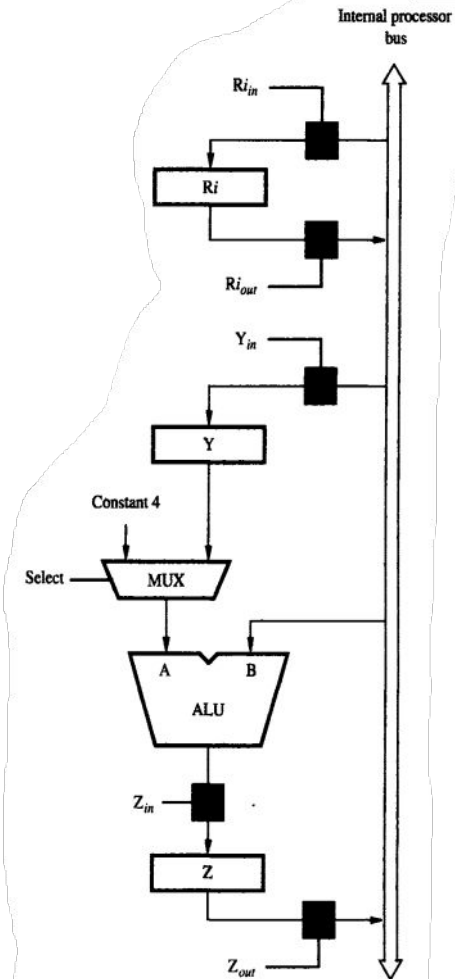


Figure 7.2 Input and output gating for the registers in Figure 7.1.

# Flip flop

- 1 - bit memory cell that stores data.
- Stores 0 or 1.
- 2 types:
  - Edge triggered
  - Level triggered.
- Level triggered:
  - Activated when the clock signal is either 0.
- Edge triggered:
  - Activated when clock changes its state. Either from 1 to 0 or from 0 to 1.

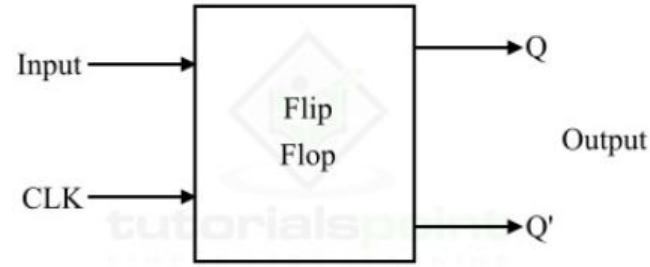


Figure 1 - Flip Flop



Figure 2 - Clock Pulse Changes State from 0 to 1 or 1 to 0

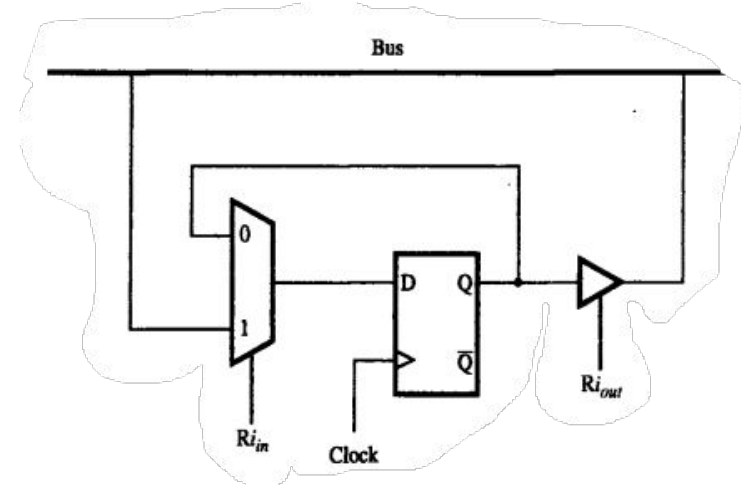
- All operations and data transfers within processor takes place within time periods defined by the *processor clock*.
- Control signals - govern a particular transfer at the start of the clock cycle.
- Ex:  $R1_{out}$  and  $R4_{in} = 1$
- Registers consist of edge triggered flip-flop, hence at next active edge of the clock -  $R4_{in}$  will load input that is waiting.
- Then,  $R1_{out}$  and  $R4_{in} = 0$

- When edge triggered flip-flops are not used, 2 or more clocks may be needed to guarantee the proper transfer of data - multiphase clocking



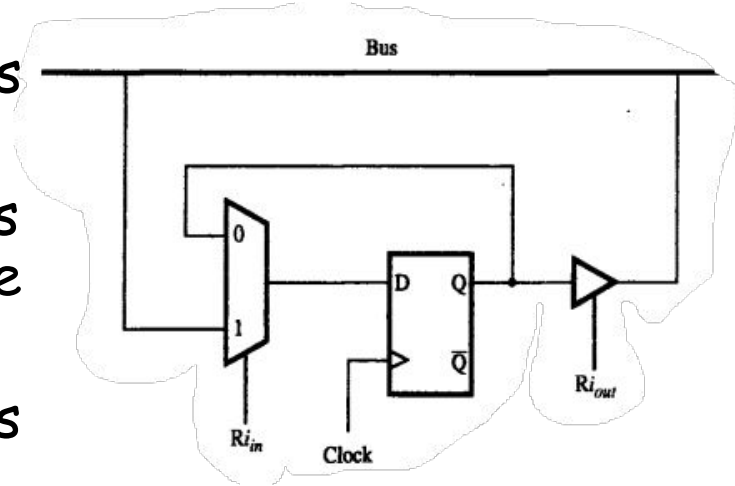
# A single bit implementation

- A 2 i/p multiplexer is shown,
- Selects the data applied to the input of an edge triggered D flip-flop.
- When  $Ri_{in} = 1$ , the multiplexer selects the data on the bus
- This data is loaded into flip-flop at the rising edge of the clock.
- When  $Ri_{in} = 0$ , the multiplexer feeds back the value currently stored in the flip-flop.



# A single bit implementation

- The Q output is connected to the bus via a tri-state gate.
- When  $Ri_{out} = 0$ , it means the output is in high impedance state - this is the open circuit state of the switch.
- When  $Ri_{out} = 1$ , the gate drives the bus to 0 or 1 - depending on value of Q.



# Performing an arithmetic and logic unit operation

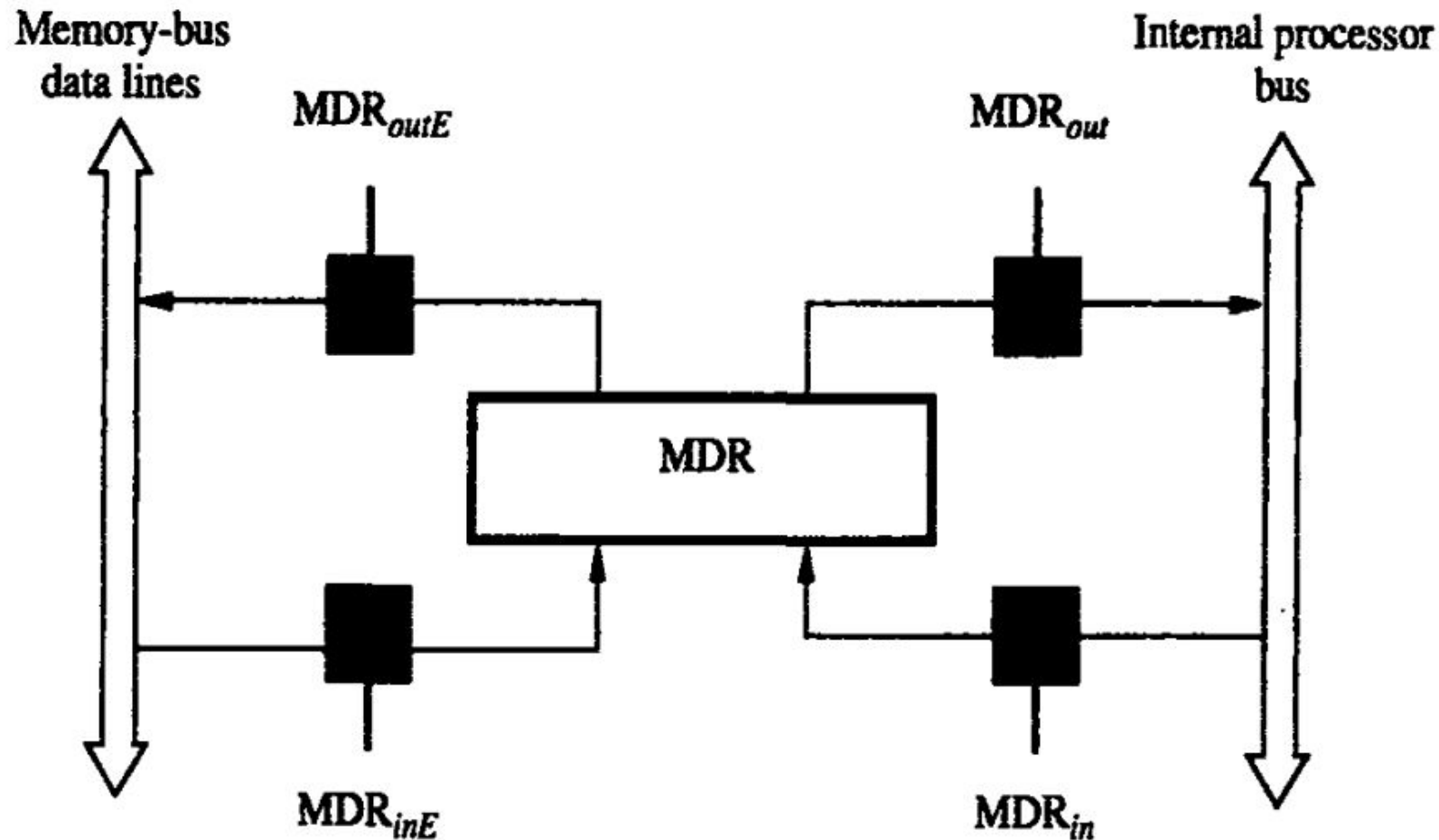
- ALU - combinational circuit with no internal storage.
- Performs arithmetic operations on A and B
- One operand is from MUX and the other is got directly from bus.
- The result from ALU is temporarily stored in register Z.
- Sequence of operation to add contents of register R1 to R2 and store the result in R3 is,
  - $R1_{out}, Y_{in}$
  - $R2_{out}, \text{Select } Y, \text{Add}, Z_{in}$
  - $Z_{out}, R3_{in}$

- Signals whose name are given in the step is activated during the clock cycle.
- Other signals - deactivated.
  - $R1_{out}, Y_{in}$
  - $R2_{out}, \text{Select } Y, \text{Add}, Z_{in}$
  - $Z_{out}, R3_{in}$
- Step1: o/p of R1 and i/p of Y - enabled  $\Rightarrow$  contents of R1 are transferred to Y.
- Step2: the MUX selects Y, which is A
  - $\Rightarrow$  the contents of R2 are gated onto bus and is B.
  - $\Rightarrow$  ALU performs the operation based on the signals applied to control line.
  - $\Rightarrow$  add line = 1, hence addition is performed.
  - $\Rightarrow$  the output of ALU is the sum of A and B.
  - $\Rightarrow$  the sum is loaded into Z - its i/p control signal is activated.
- Step3: contents of Z are loaded into R3 - destination register.
  - $\Rightarrow$  this cannot be done in step2 - only one o/p can be connected to the bus.

- We assume that there are dedicated signals for each operations - addition, subtraction, division and multiplication.
- But in reality ALU performs 8 diff operations but 3 signals are sufficient.

# Fetching a word from memory

- To fetch a word, the address is specified by the processor, then requests a read operation - applies whether an instruction / operand is to be fetched.
- The processor transfers the required address to MAR.
- The output is connected to the address lines of the memory bus.
- The control lines of the memory bus is used to specify a read operation.
- When the data is received, it is placed in MDR and then required transfer is made.



**Figure 7.4** Connection and control signals for register MDR.

- 4 control signals.
- $MDR_{in}$  and  $MDR_{out}$  □ Controls the connection to the internal bus.
- $MDR_{inE}$  and  $MDR_{outE}$  □ controls the connection to the external bus.
- A 3 i/p multiplexer can be used - memory data line connected to the third input - selected when  $MDR_{inE} = 1$
- A second tri state gate controlled by  $MDR_{outE}$  can be used to connect the output of flip-flop to the memory bus.
- During the R/W operations, the timing of the internal processor operations should be coordinated with the response of the addressed device on the memory bus.
- Internal processor completes data transfer in 1 clock cycle.
- Speed of addressed device varies with the device.
- Some I/O operations and cache miss, introduces a delay in clock cycle.
- To balance this, a signal called Memory Function Completed (MFC) is used.



• Ex: Mov (R1), R2

1.  $MAR \leftarrow [R1]$
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5.  $R2 \leftarrow [MDR]$

## Step 1 & 2

- Actions can be carried out as separate steps - some can be combined into a single step.
- Each action can be completed in 1 clock cycle.
- Except action 3, which requires 1 or more clock cycles - this depends on the speed of the addressed device.
- Simply, assume the o/p of MAR is enabled all the time.
- Contents of the MAR are always available in memory bus - processor is the bus master here.
- When a new address is loaded into the MAR - it will appear on the memory bus at the beginning of the next clock cycle.
- A read signal is activated, at the same time MAR is loaded.
- This signal will cause a bus interface circuit to send a read command, MR on the bus.

### Step 3 & 4

- Action 3 & 4, can be combined by activating control signal  $MDR_{inE}$  while waiting for a response from the memory.
- Data received from memory are loaded into MDR at the end of the clock cycle  $\square$  MFC is received.
- $MDR_{Out}$  is activated to transfer data to R2.

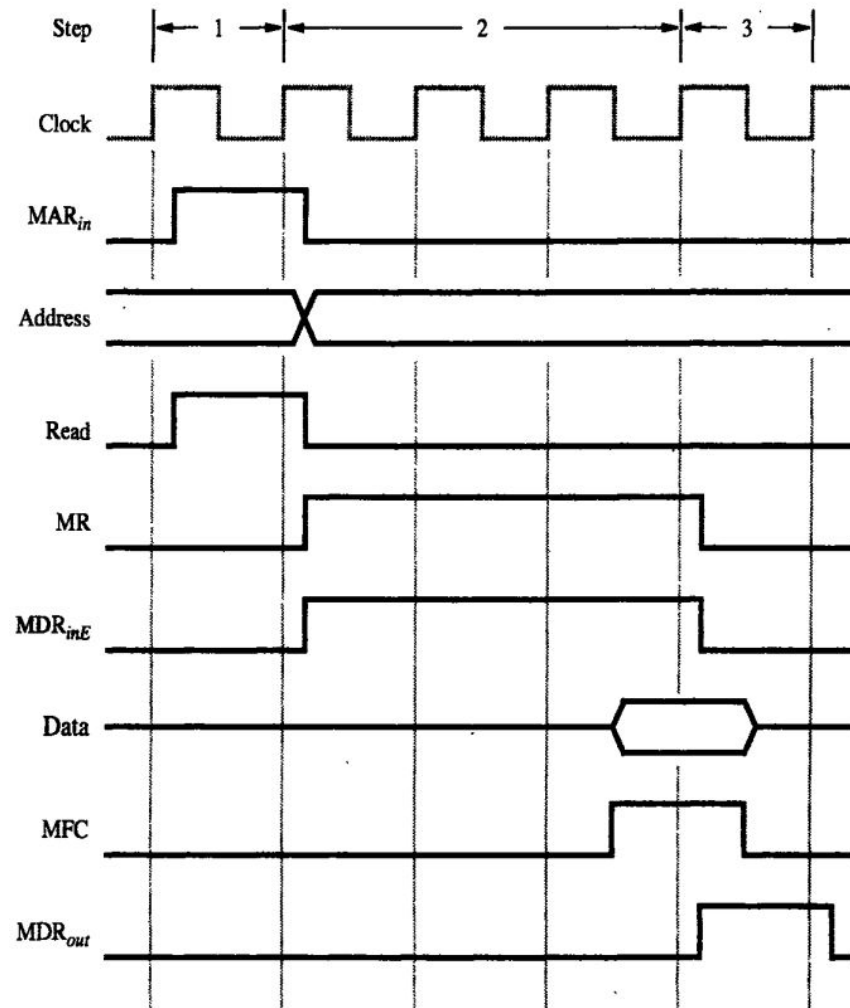
- The read operation can be simplified as,

1.  $R1_{out}, MAR_{in}, \text{Read}$

2.  $MDR_{inE}, WMFC$

3.  $MDR_{out}, R2_{in}$

- $WMFC$  □ makes the processor to wait for  $MFC$  signal.



**Figure 7.5** Timing of a memory Read operation.

# Storing a word in memory

- The desired address is loaded into MAR.
- Data to be written are loaded into MDR, then a write command is issued.
- Move R2,(R1)
  1.  $R1_{out}, MAR_{in}$
  2.  $R2_{out}, MDR_{in}, Write$
  3.  $MDR_{outE}, WMFC$
- The processor remains in step3, until MFC is received.

# Execution of a Complete Instruction

## ➤ Add (R3), R1

➤ Fetch the instruction

➤ Fetch the first operand (the contents of the memory location pointed to by R3)

➤ Perform the addition

➤ Load the result into R1

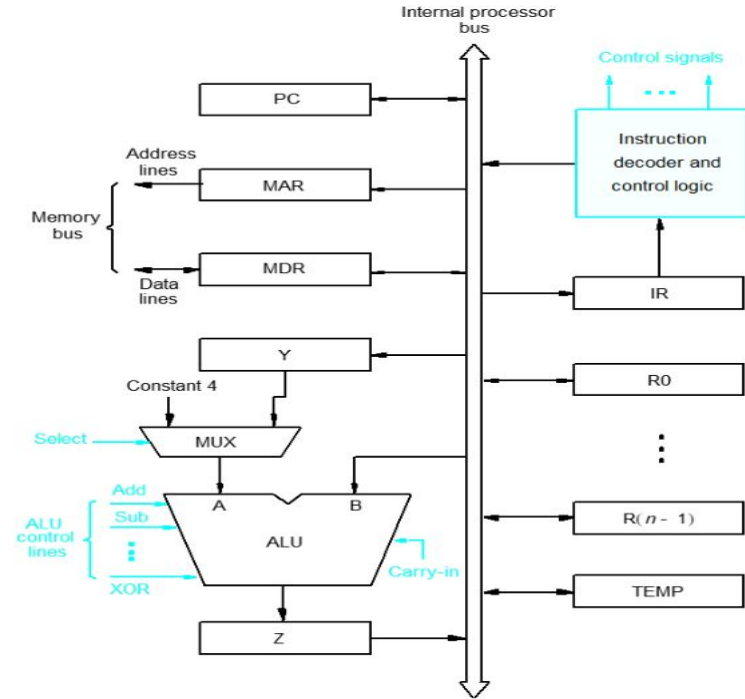
# Execution of a Complete Instruction

- Add (R3), R1

Step	Action
1	PC <sub>out</sub> , MAR <sub>in</sub> , Read, Select4, Add, Z <sub>in</sub>
2	Z <sub>out</sub> , PC <sub>in</sub> , Y <sub>in</sub> , WMFC
3	MDR <sub>out</sub> , IR <sub>in</sub>
4	R3 <sub>out</sub> , MAR <sub>in</sub> , Read
5	R1 <sub>out</sub> , Y <sub>in</sub> , WMFC
6	MDR <sub>out</sub> , SelectY, Add, Z <sub>in</sub>
7	Z <sub>out</sub> , R1 <sub>in</sub> , End

Figure

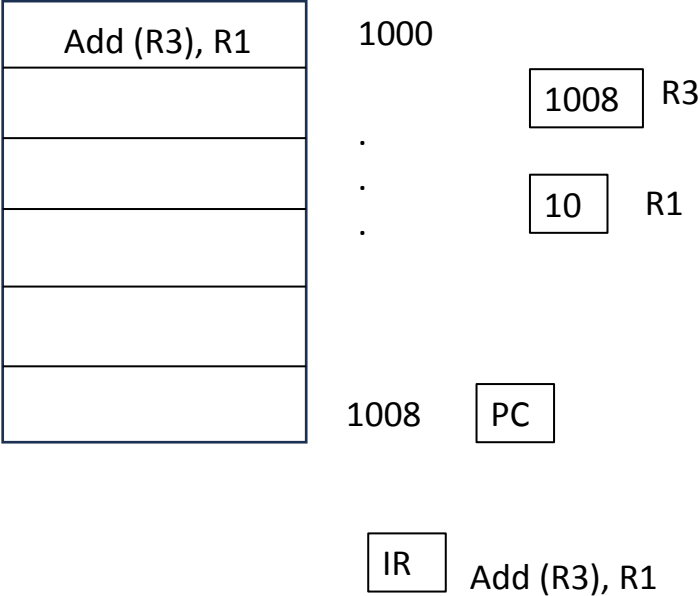
Figure.





Step 1-3 => Instruction Fetch Phase

Step 4 -7 => Execution Phase



Step	Action
1	$PC_{out}$ , $MAR_{in}$ , Read, Select4,Add, $Z_{in}$
2	$Z_{out}$ , $PC_{in}$ , $Y_{in}$ , WMF C
3	$MDR_{out}$ , $IR_{in}$
4	$R3_{out}$ , $MAR_{in}$ , Read
5	$R1_{out}$ , $Y_{in}$ , WMF C
6	$MDR_{out}$ , SelectY, Add, $Z_{in}$
7	$Z_{out}$ , $R1_{in}$ , End

# Execution of Branch Instruction

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset  $X$  given in the branch instruction.
- The offset  $X$  is usually the difference between the branch target address and the address immediately following the branch instruction.
- UnConditional branch

# Execution of Branch Instruction

---

Step	Action
------	--------

---

- |   |  |
|---|--|
| 1 | $PC_{out}$ , $MAR_{in}$ , Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$ , $PC_{in}$ , $Y_{in}$ , WMF C               |
| 3 | $MDR_{out}$ , $IR_{in}$                                |
| 4 | Offset-field-of- $IR_{out}$ , Add, $Z_{in}$            |
| 5 | $Z_{out}$ , $PC_{in}$ , End                            |
- 

**Figure.** Control Sequence for Unconditional Branch Instructions



## Step Action

- 1       $PC_{out}$  ,  $MAR_{in}$  , Read, Select4, Add,  $Z_{in}$
- 2       $Z_{out}$  ,  $PC_{in}$  ,  $Y_{in}$  , WMF C
- 3       $MDR_{out}$  ,  $IR_{in}$
- 4      Offset-field-of-IR<sub>out</sub> , Add,  $Z_{in}$
- 5       $Z_{out}$  ,  $PC_{in}$  , End

In case of conditional branch ,  
For example , for (branch<0) instruction step 4 is replaced with  
**Offset-field-of IR<sub>out</sub> , Add ,  $Z_{in}$  , if  $N=0$  then End**

If  $N=0$  the processor returns to step 1 immediately after step 4.  
If  $N=1$ , step 5 is performed to load a new value into PC and performing branch operation

## Bus

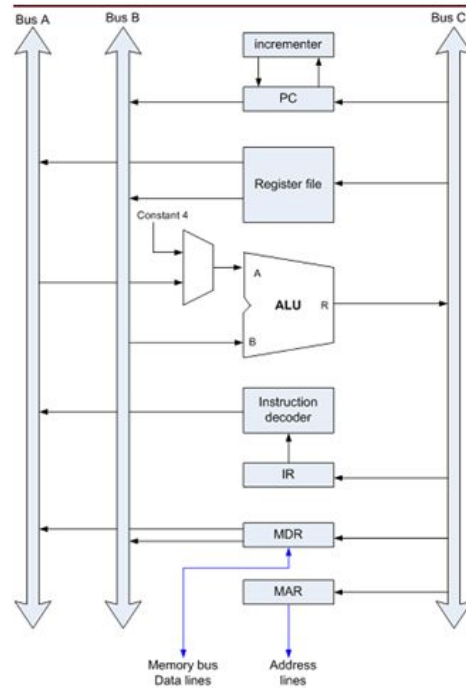
### Simple single-bus structure

- Has one common bus for data transfer.
- Interconnected circuits/devices which has varying speed of execution like CPU and memory.
- Results in long control sequences, because only one data item can be transferred over the bus in a clock cycle.

### Multi-bus structure

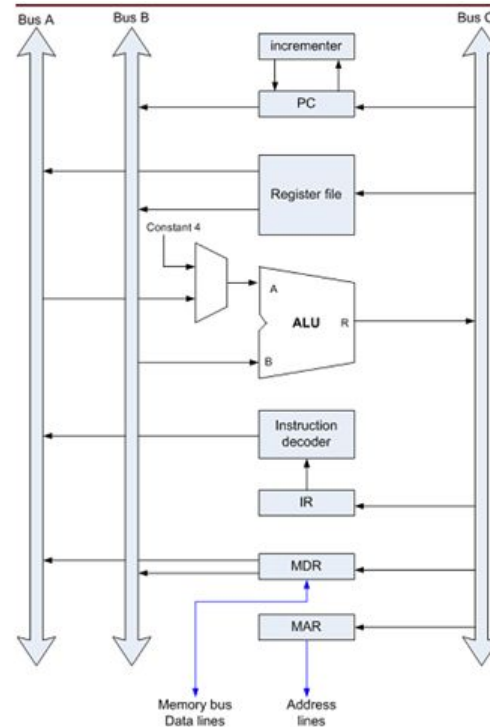
- Most commercial processors provide multiple internal paths to enable several transfers to take place in parallel.
- Data transfer requires less control sequences.
- Multiple data transfer can be done in a single clock cycle

# Multibus Architecture

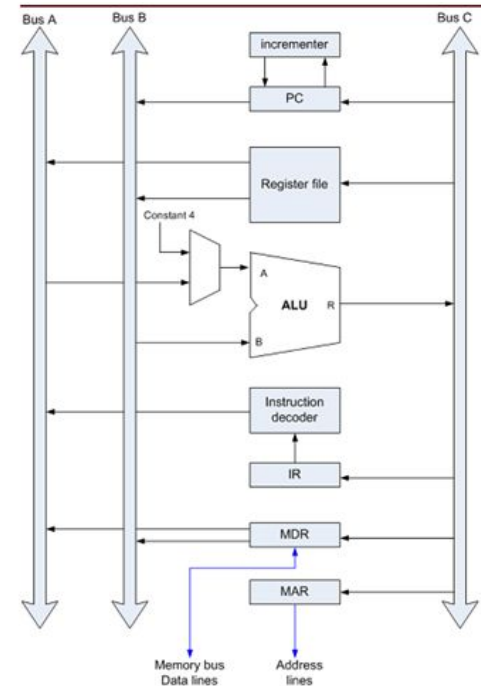


# Multibus Architecture

- Three-bus organization to connect the registers and the ALU of a processor.
- All general-purpose registers are combined into a single block called register file.
- Register file has three ports.
- Two outputs ports connected to buses A and B, allowing the contents of two different registers to be accessed simultaneously, and placed on buses A and B.
- Third input port allows the data on bus C to be loaded into a third register during the same clock cycle.
- Inputs to the ALU and outputs from the ALU:
- Buses A and B are used to transfer the source operands to the A and B inputs of the ALU.
- Result is transferred to the destination over bus C.



- ALU can also pass one of its two input operands unmodified if needed:
- **Control signals** for such an operation are  $R=A$  or  $R=B$ .  
(Passing the value of A to Bus C through ALU or Passing the value of B to Bus C through ALU)
- Three bus arrangement obviates the need for Registers Y and Z in the single bus organization.
- Incrementer unit: Used to increment the PC by 4.
- Source for the constant 4 at the ALU multiplexer can be used to increment other addresses such as the memory addresses in multiple load/store instructions.





## Input Output Specifications

Component	Input	Output
Program Counter	PC <sub>IN</sub>	PC <sub>OUT</sub>
Register File	R1 <sub>IN</sub>	R1 <sub>OUT</sub>
ALU	A ,B 4	R=B for selecting B Select A for selecting A Select 4 – for fetching consecutive memory locations
Instruction Register	IR <sub>IN</sub>	IR <sub>OUT</sub>
Memory Data Register	MDR <sub>IN</sub>	MDR <sub>OUTA/OUTB</sub>
Memory Address Register	MAR <sub>IN</sub>	MAR <sub>OUT</sub>

# Execution of the Statement “Add R4 R5 R6” in multibus environment

Example :Add R4, R5, R6

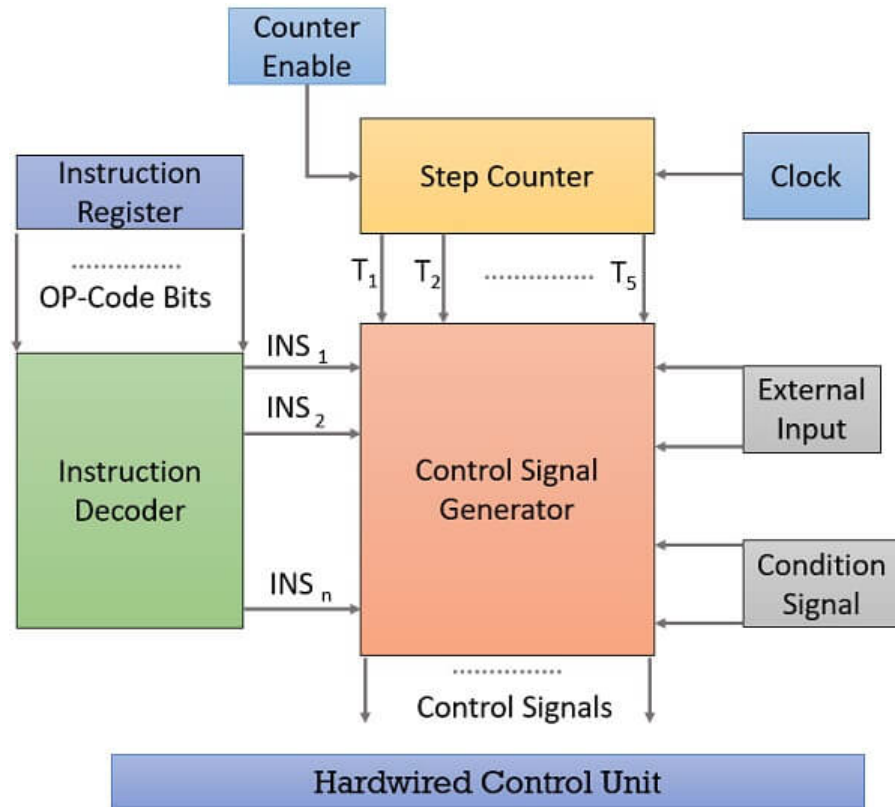
↳

```
1    PCout, R=B, MARin, Read, IncPC
2    WMFC
3    MDRoutB, R=B, IRin
4    R4outA, R5outB, SelectA, Add, R6in, End
```

---

1. PC value is passed to Bus B, From bus B (Through ALU) to Bus C.
2. Bus C to MAR
3. PC value will be incremented
4. Wait for Memory function complete operation
5. MDR(out) to Bus B (Instruction to B)
6. B to IR
7. R4 given to Bus A
8. R5 Given to Bus B
9. Select A => means A will be selected in ALU. Bus B passes R5 value directly.
10. The OP of ALU will be passed to Bus C and to Register R6

- The control signals are generated by the control unit to carry out the execution of instructions.
  - Once the instructions are fetched from the memory and placed in the IR, it is sent to the decoder unit to carry out the operation mentioned in the instructions.
  - Once the instruction is decoded, the appropriate control signals will be generated by the control unit.
  - The control signals are generated using two techniques, such as Hardwired and Microprogrammed control unit.
- 
- **Hardwired Control Unit:** it uses set of logic gates and circuits to execute instructions.
  - **Mircoprogrammed Control Unit:** it uses microcode to execute instructions. Microcode is a set of instructions that can be modified or updated.



- **Instruction Register:** it is a processor register used to store the instructions that is currently being executed.
- **Instruction Decoder:** it takes the Op code from the IR and it will send the OP code bits as the input to the Control Signal Generator.
- **Step Counter:** it will mention the current step or state of the instruction such as fetch, decode, operands read, ALU and store. Based on the step the signals of the step counter will be set to 1 and given as the input to the Control Signal Generator.
- **Clock:** one clock pulse is required to complete one state (step) of the instruction. Example if the step counter is in T3, after completing one clock signal the step counter will move to T4.
- **Counter Enable:** it controls the step counter, even if one clock signal completed, the counter enable will allow the step counter to move to the next state (i.e. from fetch to decode).
- **Condition signal:** start, stop or reset.
- **External input:** it represents some kind of interrupt or any error message.

# Microprogrammed Control Unit

- The control signals required inside the processor can be generated using the control step counter and a decoder unit.
- An alternative scheme, called microprogrammed control, in which control signals are generated by a program similar to a machine language program.

## Common Terms:

- **Control Word:** it is a word whose individual bits represents various control signals.
- **Microroutine:** a sequence of CWs corresponding to the control sequence of a machine instruction constitutes a microroutine for that instruction.
- **Microinstruction:** the individual control word in a microroutine are referred to as microinstruction.

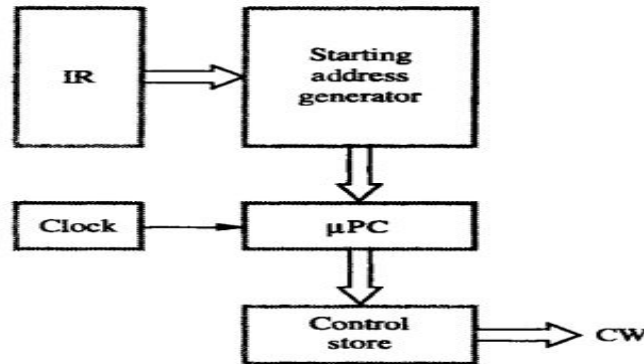
# Control Sequence and Control Word for the instruction Add (R3),R1

## Step Action

- 1  $PC_{out}$ ,  $MAR_{in}$ , Read, Select4, Add,  $Z_{in}$
- 2  $Z_{out}$ ,  $PC_{in}$ ,  $Y_{in}$ , WMFC
- 3  $MDR_{out}$ ,  $IR_{in}$
- 4  $R3_{out}$ ,  $MAR_{in}$ , Read
- 5  $R1_{out}$ ,  $Y_{in}$ , WMFC
- 6  $MDR_{out}$ , SelectY, Add,  $Z_{in}$
- 7  $Z_{out}$ ,  $R1_{in}$ , End

Micro - instruction	..	$PC_{in}$	$PC_{out}$	$MAR_{in}$	Read	$MDR_{out}$	$IR_{in}$	$Y_{in}$	Select	Add	$Z_{in}$	$Z_{out}$	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	:
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

- The microroutines for all the instructions in the instruction set of a computer are stored in a special memory called **control store**.
- The control unit can generate the control signals for any instruction by reading the control word from the control store.
- A micro program counter ( $\mu$ PC) is used to read the control words sequentially from

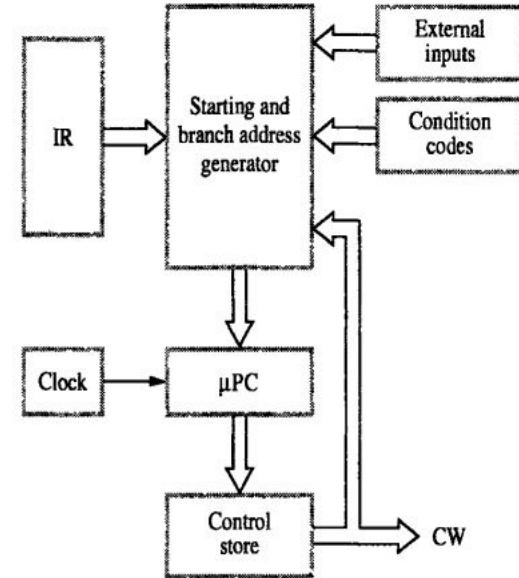




- Whenever a new instruction is loaded into the IR, the output of the block “starting address generator” is loaded into the  $\mu$ PC.
- The  $\mu$ PC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence the control signals are delivered to various parts of the processor in the correct sequence.
- Whenever a branch instruction is encountered, a branch microinstruction transfers the control to the corresponding microroutine, which is assumed to start at a location (example: 25) in the control store.
- That address is the output of the starting address generator block.

Address	Microinstruction
0	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
1	$Z_{out}, PC_{in}, Y_{in}, WMFC$
2	$MDR_{out}, IR_{in}$
3	Branch to starting address of appropriate microroutine
.....	
25	If $N=0$ , then branch to microinstruction 0
26	Offset-field-of- $IR_{out}, SelectY, Add, Z_{in}$
27	$Z_{out}, PC_{in}, End$

- The microinstruction at location 25 test the N bit of the condition codes. If this bit is equals to 0, a branch takes place to location 0 to fetch a new machine instruction.
- Otherwise, the machine instruction at location 26 is executed to put the branch target address in register Z.
- To support microprogram branching, the organization of the control unit should be modified as shown in the figure.



# What is Pipelining?

A way of speeding up execution of instructions

*Key idea:*

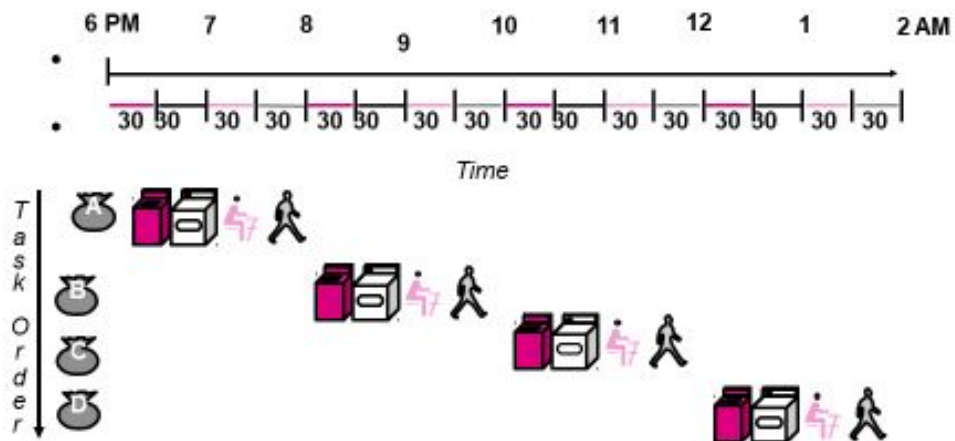
overlap execution of multiple instructions

# The Laundry Analogy

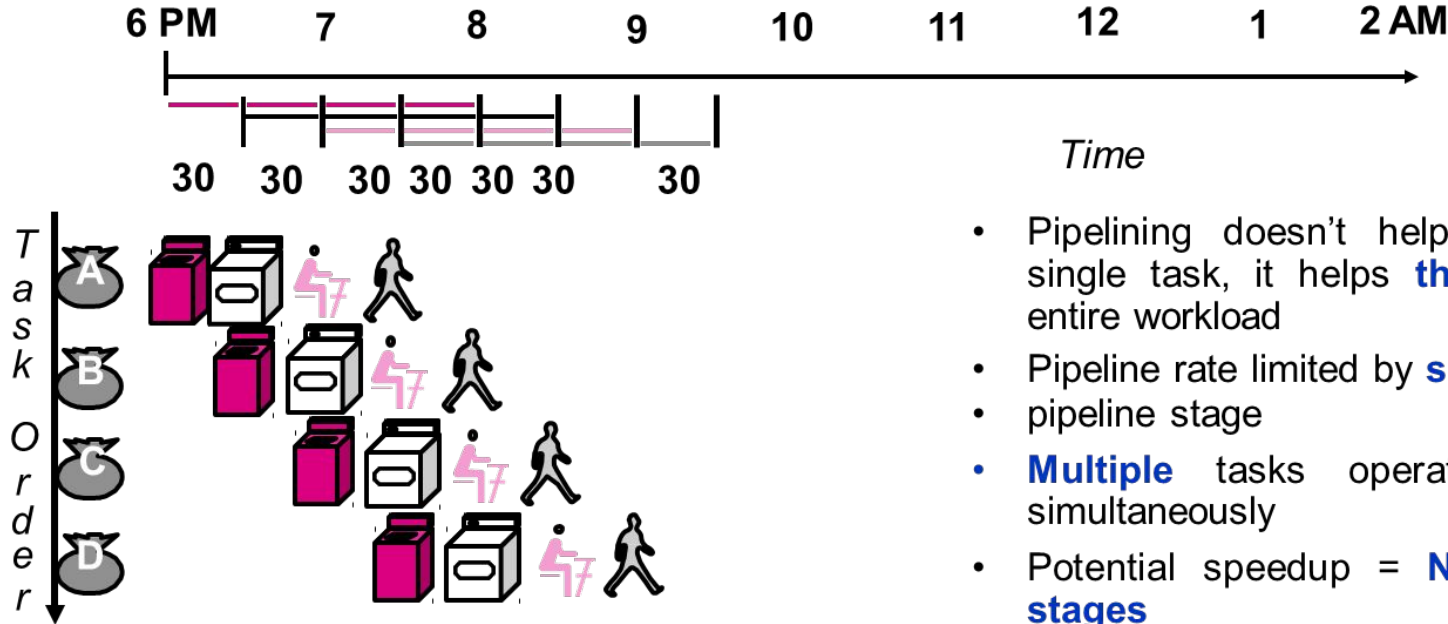
- A, B, C, D  
each have one load of  
clothes to wash, dry, and fold
- Washer takes 30  
minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes  
to put clothes into drawers



If we do laundry sequentially...



# To Pipeline, We Overlap Tasks



- Latency is the **time from start to finish** for a given task.

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

- Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased.
- Simultaneous execution of more than one instruction takes place in a pipelined processor.
- Let us see a real-life example that works on the concept of pipelined operation. Consider a water bottle packaging plant. Let there be 3 stages that a bottle should pass through, Inserting the bottle(I), Filling water in the bottle(F), and Sealing the bottle(S). Let us consider these stages as stage 1, stage 2, and stage 3 respectively. Let each stage take 1 minute to complete its operation.
- Now, in a non-pipelined operation, a bottle is first inserted in the plant, after 1 minute it is moved to stage 2 where water is filled. Now, in stage 1 nothing is happening. Similarly, when the bottle moves to stage 3, both stage 1 and stage 2 are idle. But in pipelined operation, when the bottle is in stage 2, another bottle can be loaded at stage 1. Similarly, when the bottle is in stage 3, there can be one bottle each in stage 1 and stage 2. So, after each minute, we get a new bottle at the end of stage 3. Hence, the average time taken to manufacture 1 bottle is:



The same principles apply to processors where we pipeline instruction-execution.

MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register



# Pipelining a Digital System

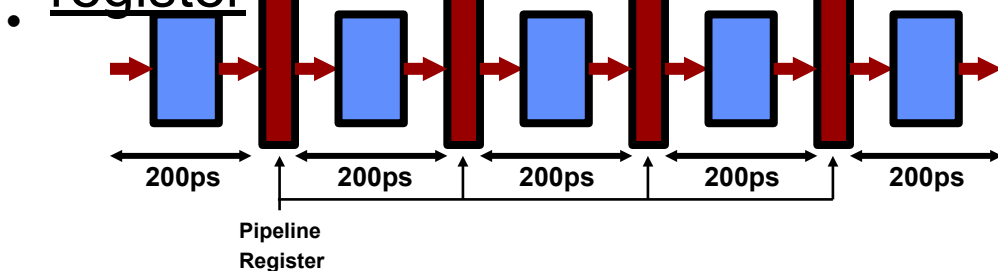
- Key idea: break big computation up into pieces



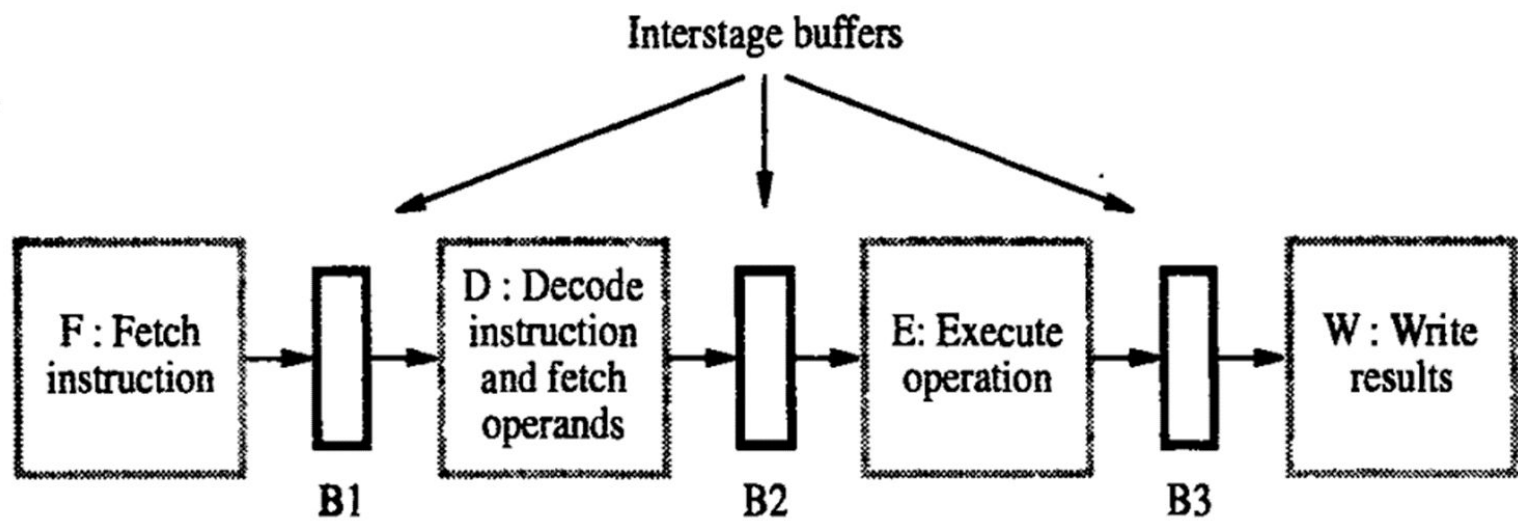
1ns

Separate each piece with a pipeline

register



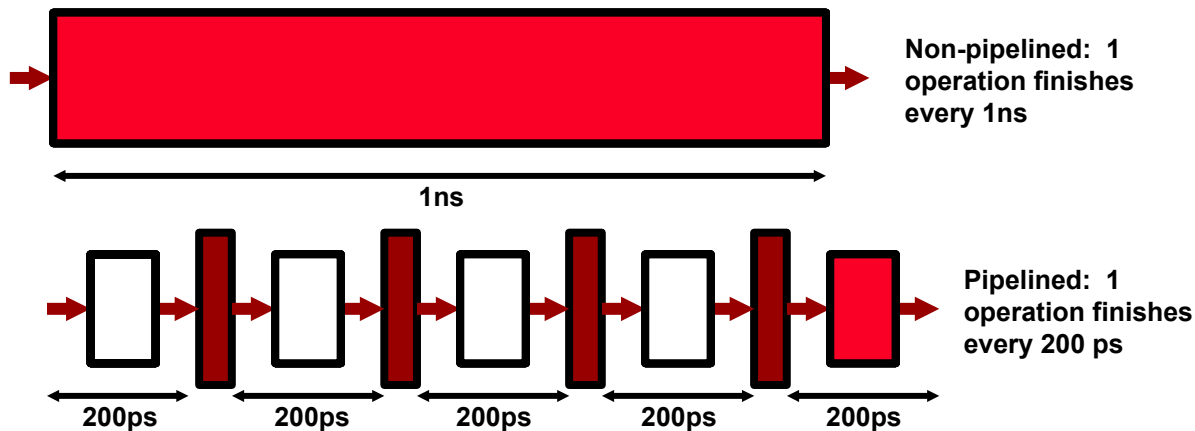
- F**      **Fetch:** read the instruction from the memory.
- D**      **Decode:** decode the instruction and fetch the source operand(s).
- E**      **Execute:** perform the operation specified by the instruction.
- W**      **Write:** store the result in the destination location.



(b) Hardware organization

# Pipelining a Digital System

Why do this? Because it's faster for repeated computations



- Suppose we need to perform multiply and add operation with a stream of numbers
- $A_i * B_i + C_i$  for  $i = 1, 2, 3, \dots, 7$
- Each subinstruction is implemented in a segment within the pipeline. Each segment has one or two registers and a combinational circuit

The sub operations performed in each segment are

- as follows

$$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$$

Input  $A_i$  and  $B_i$

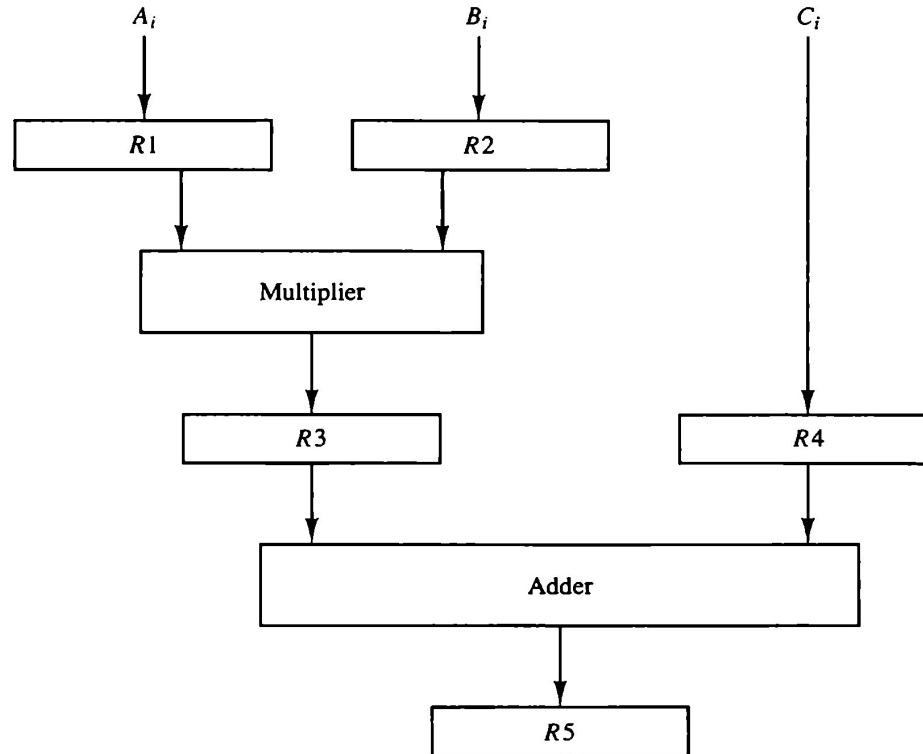
$$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$$

Multiply and input  $C_i$

$$R5 \leftarrow R3 + R4$$

Add  $C_i$  to product

# Example of Pipeline Processing



# Content of Registers in Pipeline

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>R4</i>	<i>R5</i>
1	$A_1$	$B_1$	—	—	—
2	$A_2$	$B_2$	$A_1 * B_1$	$C_1$	—
3	$A_3$	$B_3$	$A_2 * B_2$	$C_2$	$A_1 * B_1 + C_1$
4	$A_4$	$B_4$	$A_3 * B_3$	$C_3$	$A_2 * B_2 + C_2$
5	$A_5$	$B_5$	$A_4 * B_4$	$C_4$	$A_3 * B_3 + C_3$
6	$A_6$	$B_6$	$A_5 * B_5$	$C_5$	$A_4 * B_4 + C_4$
7	$A_7$	$B_7$	$A_6 * B_6$	$C_6$	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	$C_7$	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

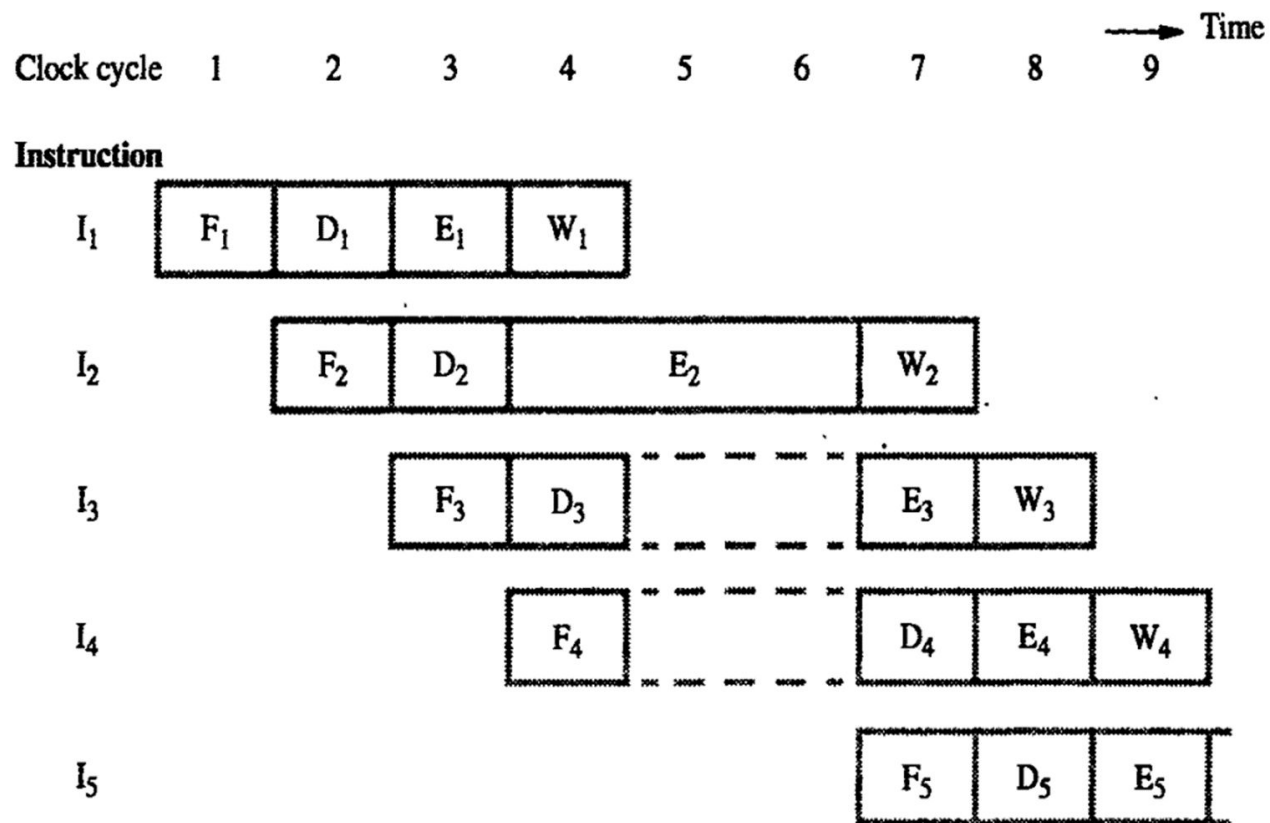
# Space Time Diagram of Pipeline

		1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment:	1	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$				
	2		$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$			
	3			$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$		
	4				$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	



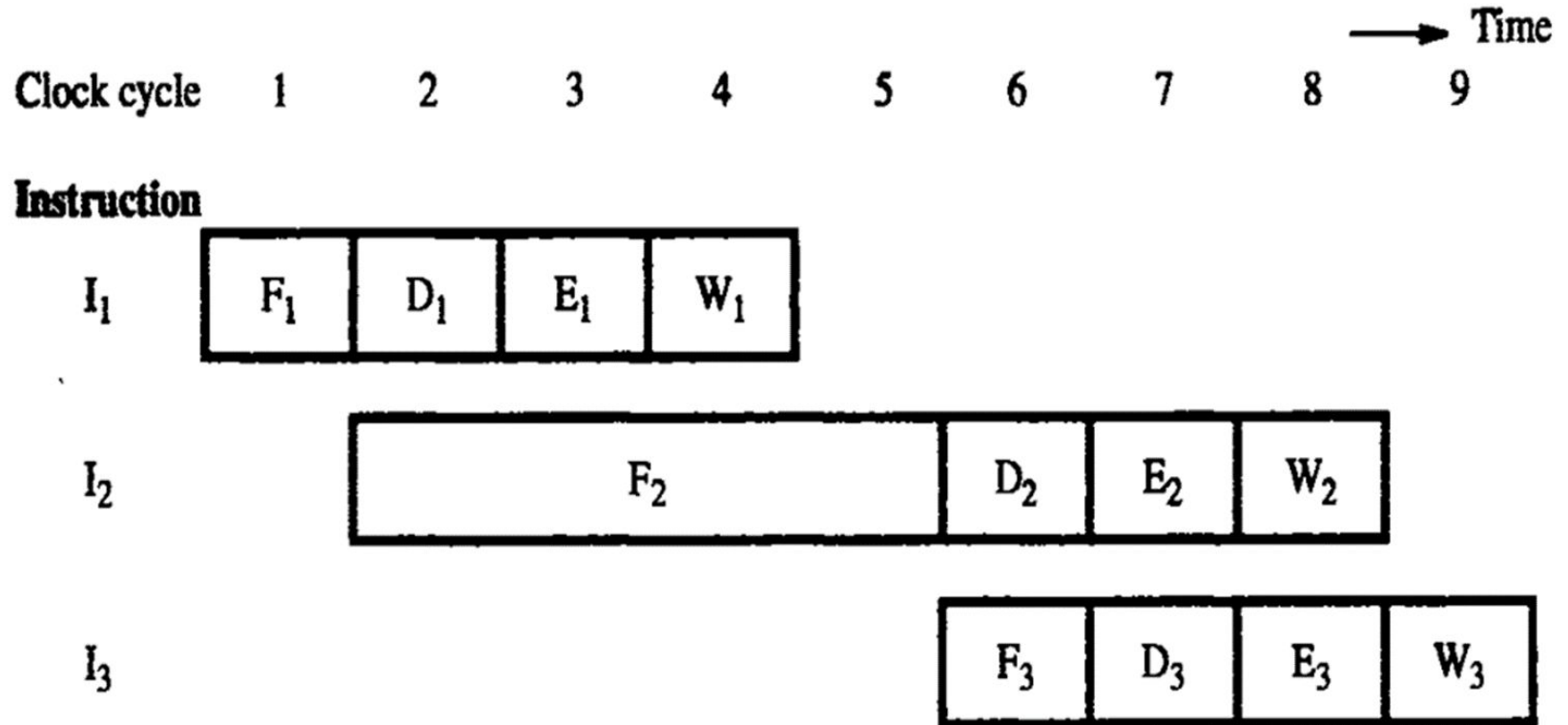
# Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.
- Where one instruction cannot immediately follow another.
- Types of hazards
  - Structural hazards - attempt to use the same resource by two or more instructions
  - Control hazards - attempt to make branching decisions before branch condition is evaluated
  - Data hazards - attempt to use data before it is ready
  - Instructional Hazard
- Can always resolve hazards by waiting



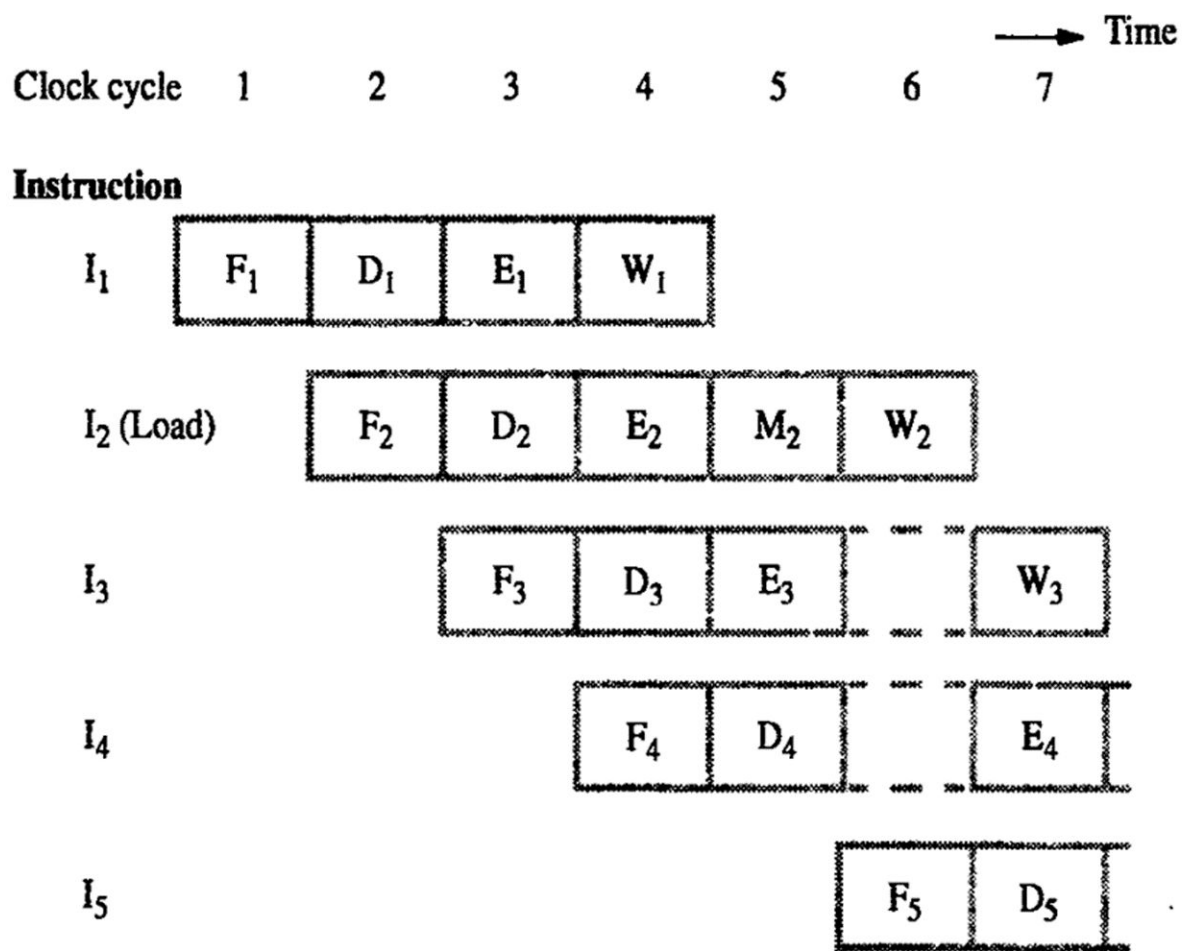
**Figure 8.3** Effect of an execution operation taking more than one clock cycle.

# Instruction Hazard



# Structural Hazards

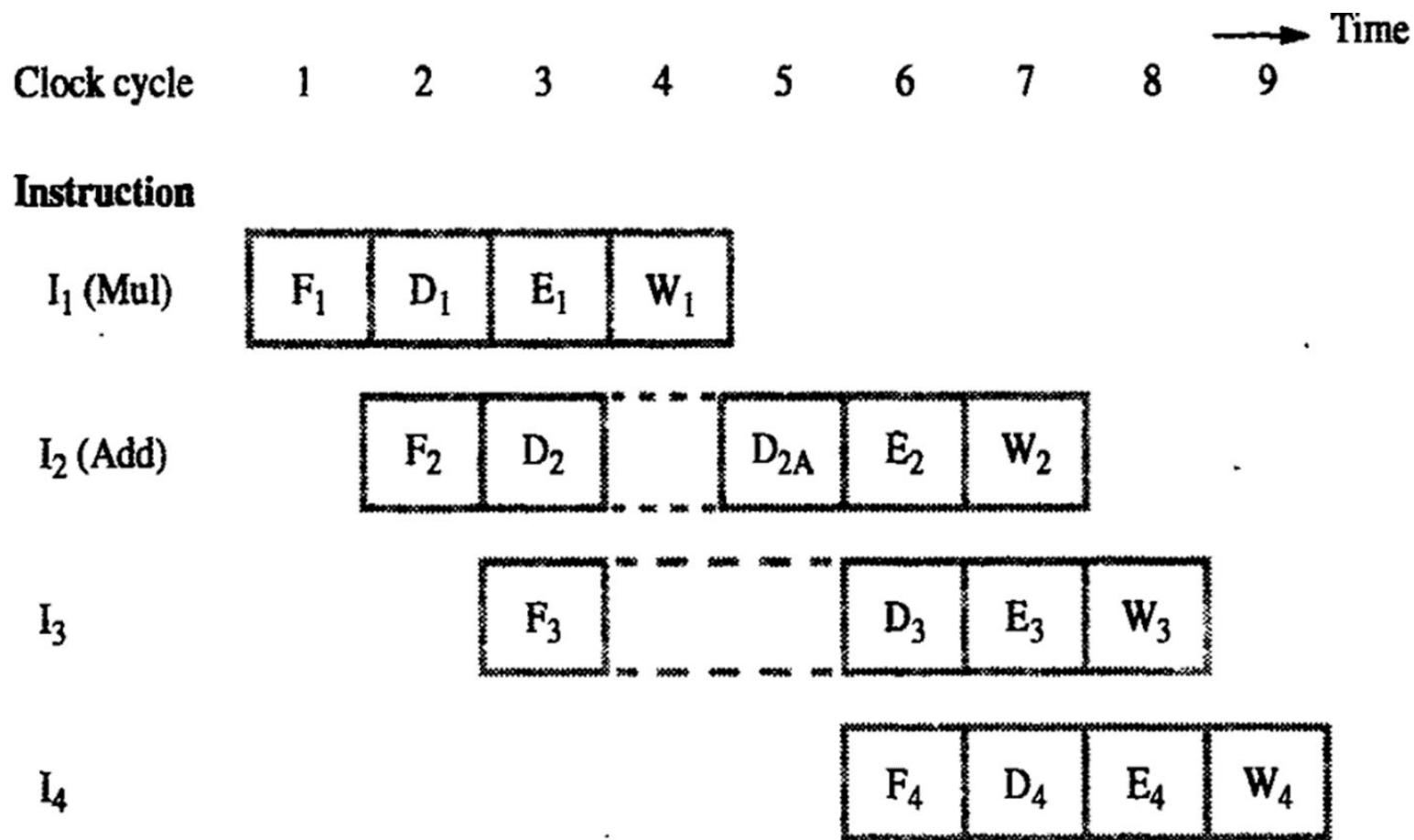
- Attempt to use the same resource by two or more instructions at the same time
- Occurs when hardware is required by 2 instructions
- Example: Single Memory for instructions and data
  - Accessed by IF stage
  - Accessed at same time by MEM stage
- Solutions
  - Delay the second access by one clock cycle, OR
  - Provide separate memories for instructions & data



**Figure 8.5** Effect of a Load instruction on pipeline timing.

# Data Hazard

- Data that is required is delayed for some reason.



**Figure 8.6** Pipeline stalled by data dependency between  $D_2$  and  $W_1$ .

# Pipelined Example - Executing Multiple Instructions

- **Consider the following instruction sequence:**

```
lw $r0, 10($r1)  sw
```

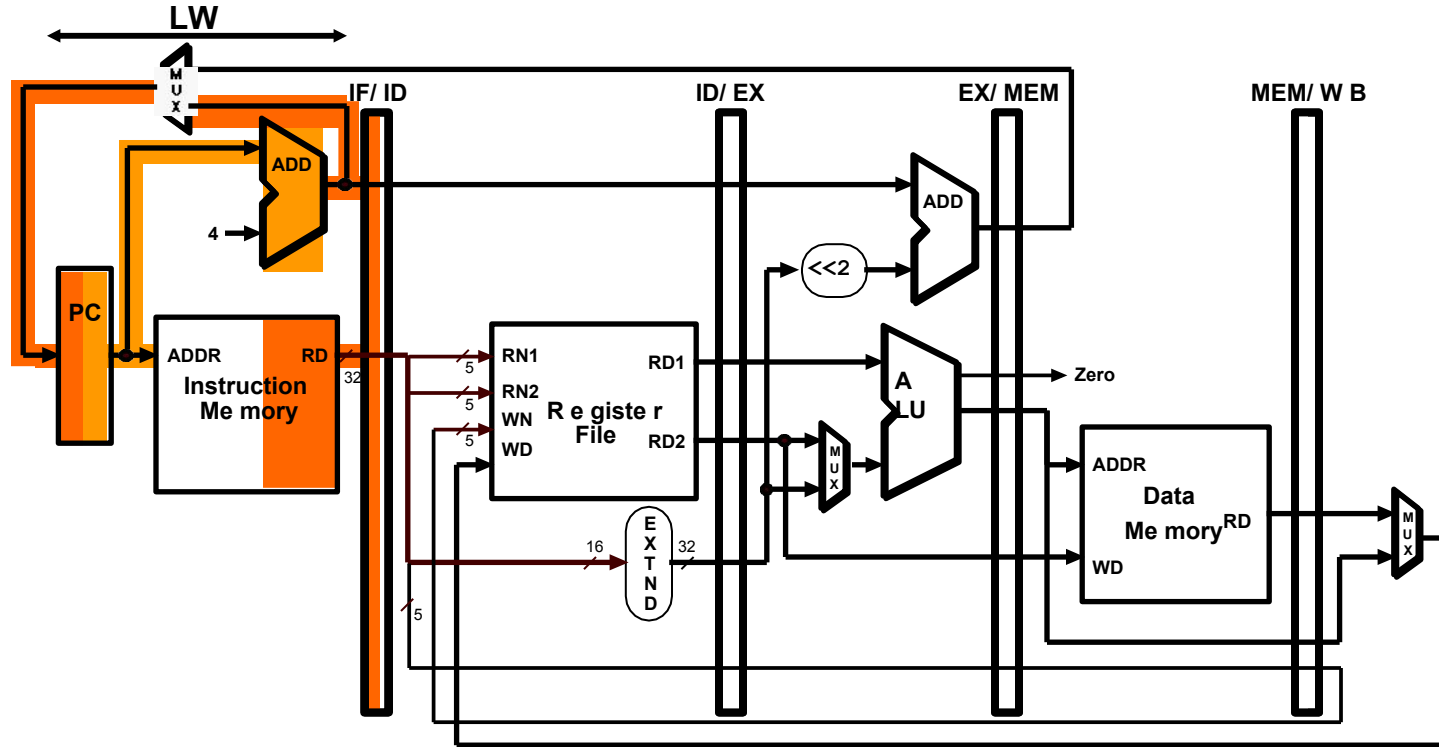
```
$sr3, 20($r4)
```

```
add $r5, $r6, $r7
```

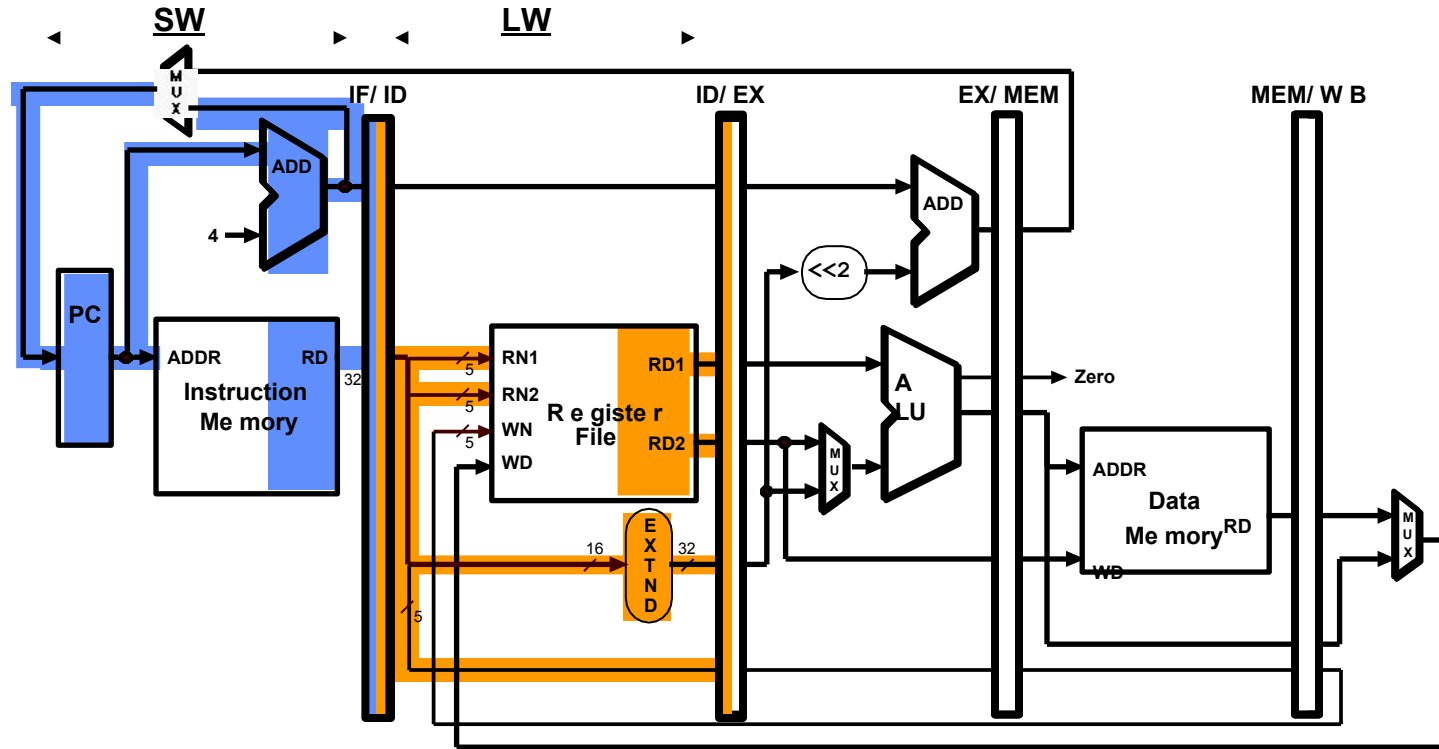
```
sub $r8, $r9, $r10
```



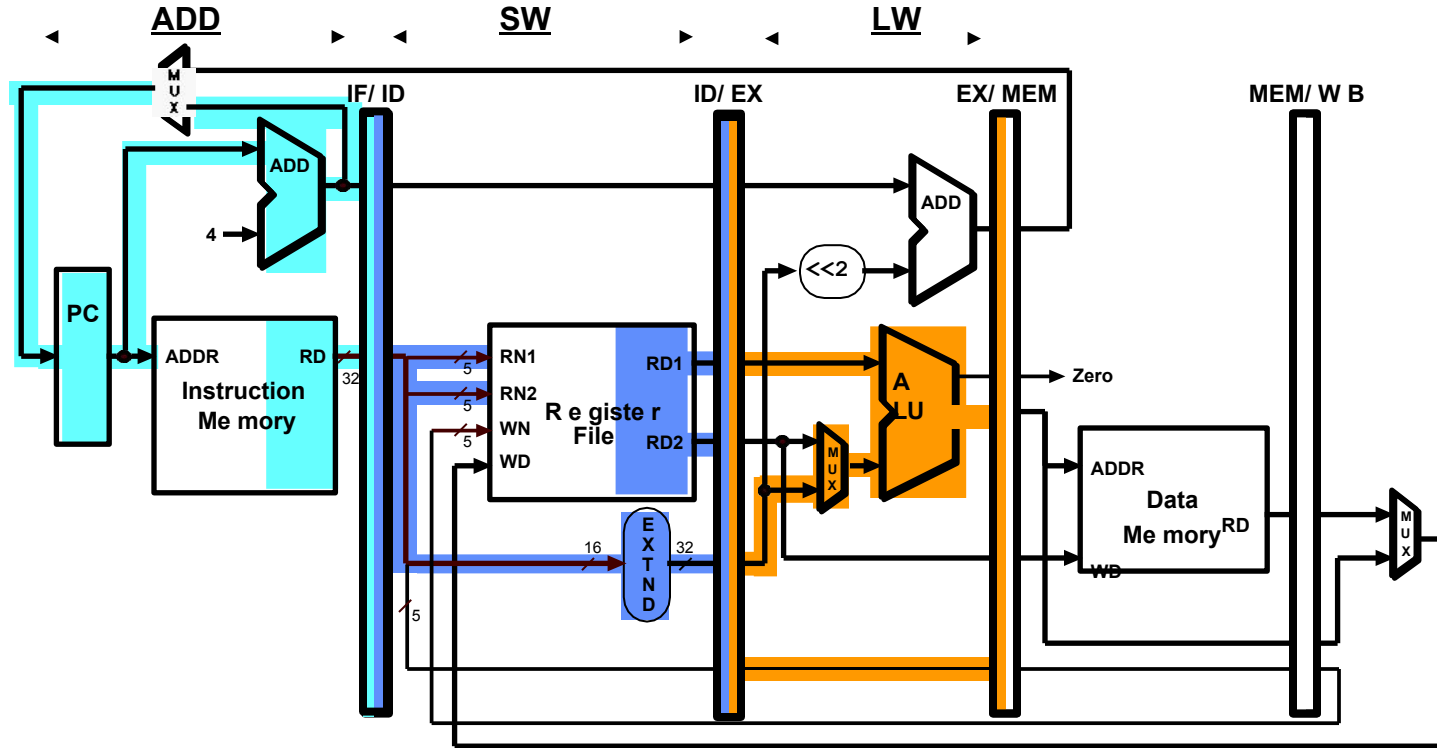
# Executing Multiple Instructions Clock Cycle 1



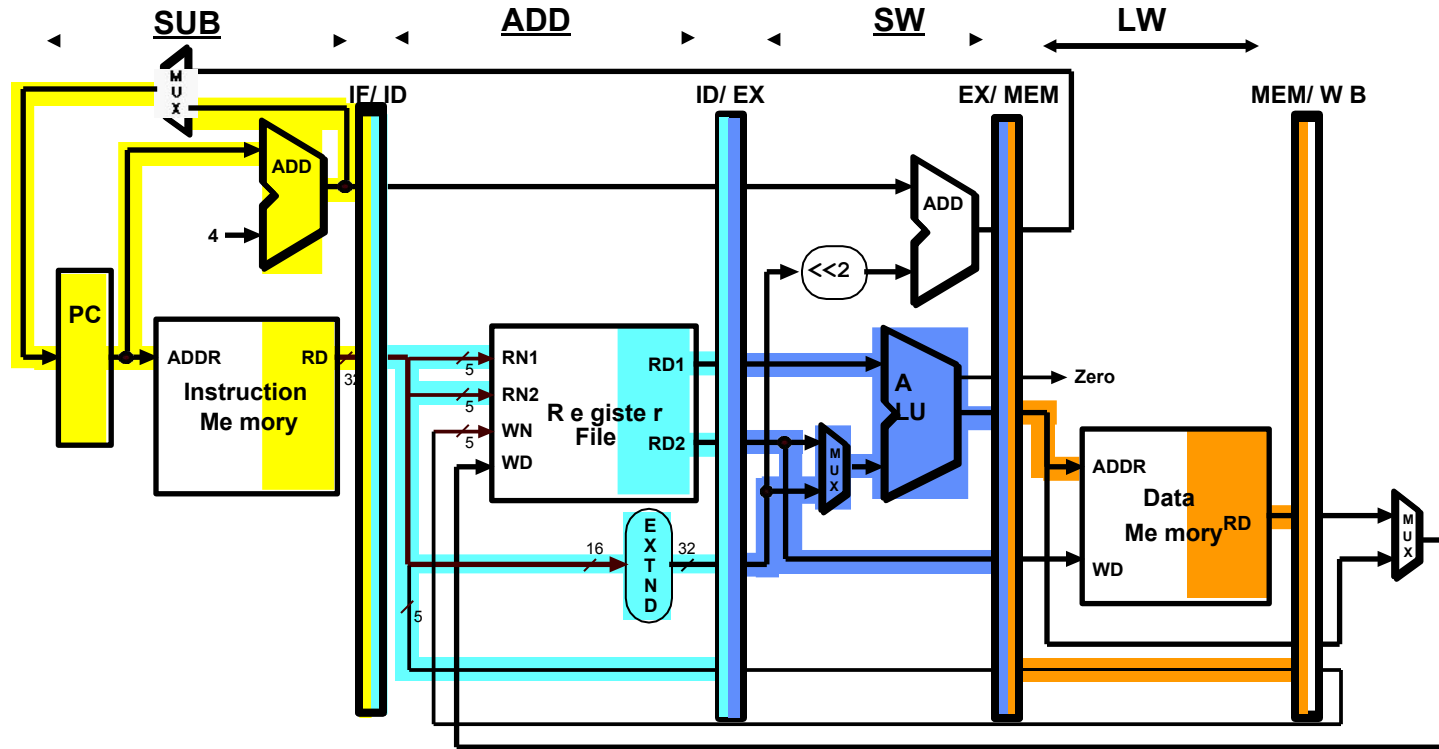
# Executing Multiple Instructions Clock Cycle



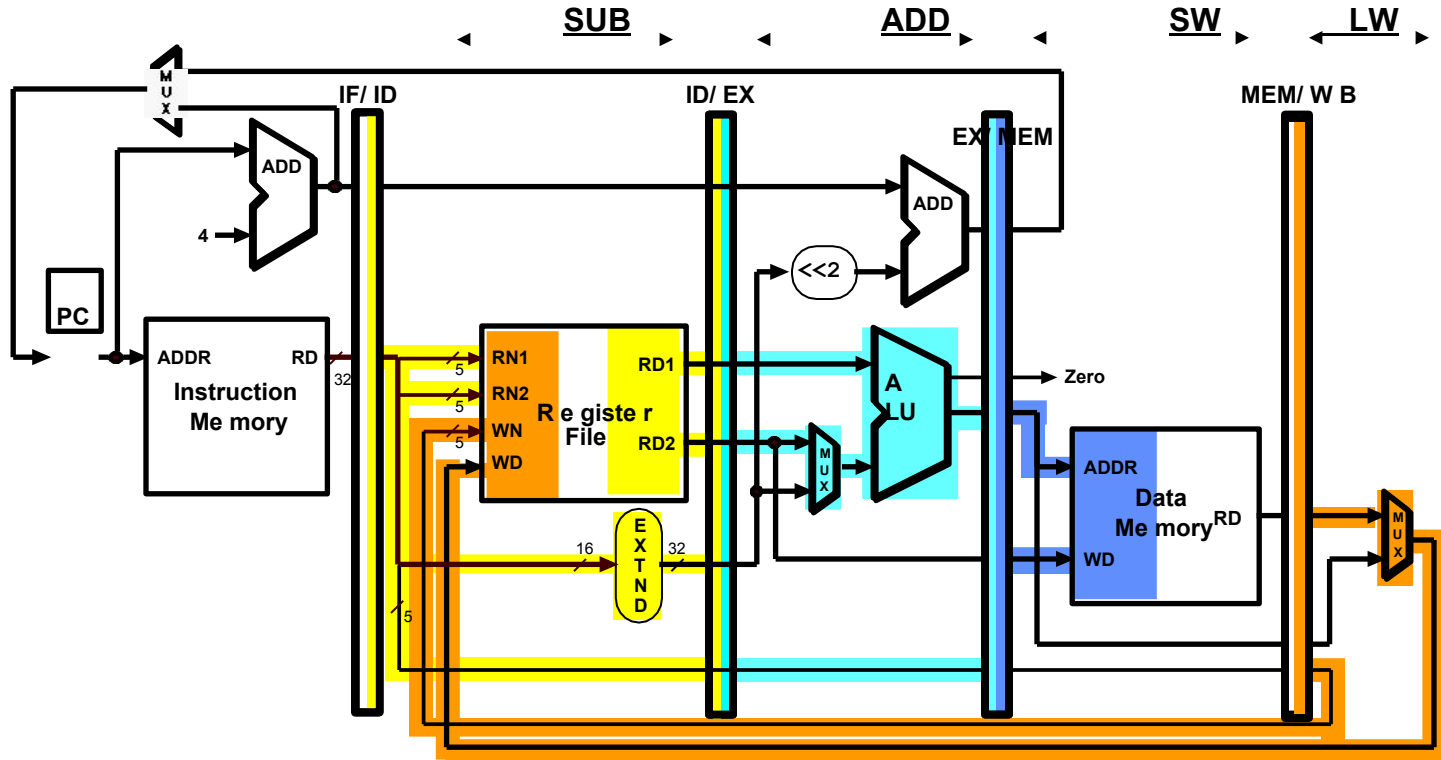
# Executing Multiple Instructions Clock Cycle



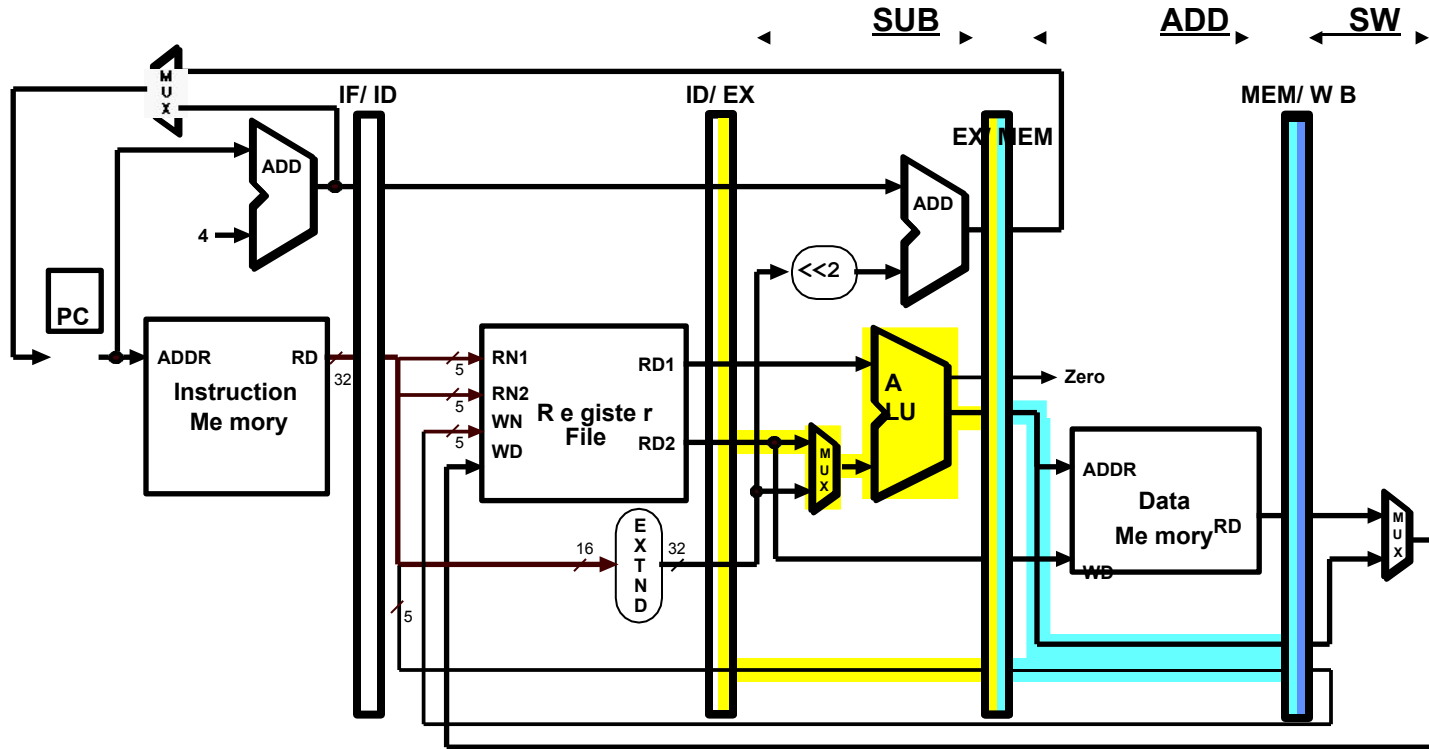
# Executing Multiple Instructions Clock Cycle



# Executing Multiple Instructions Clock Cycle

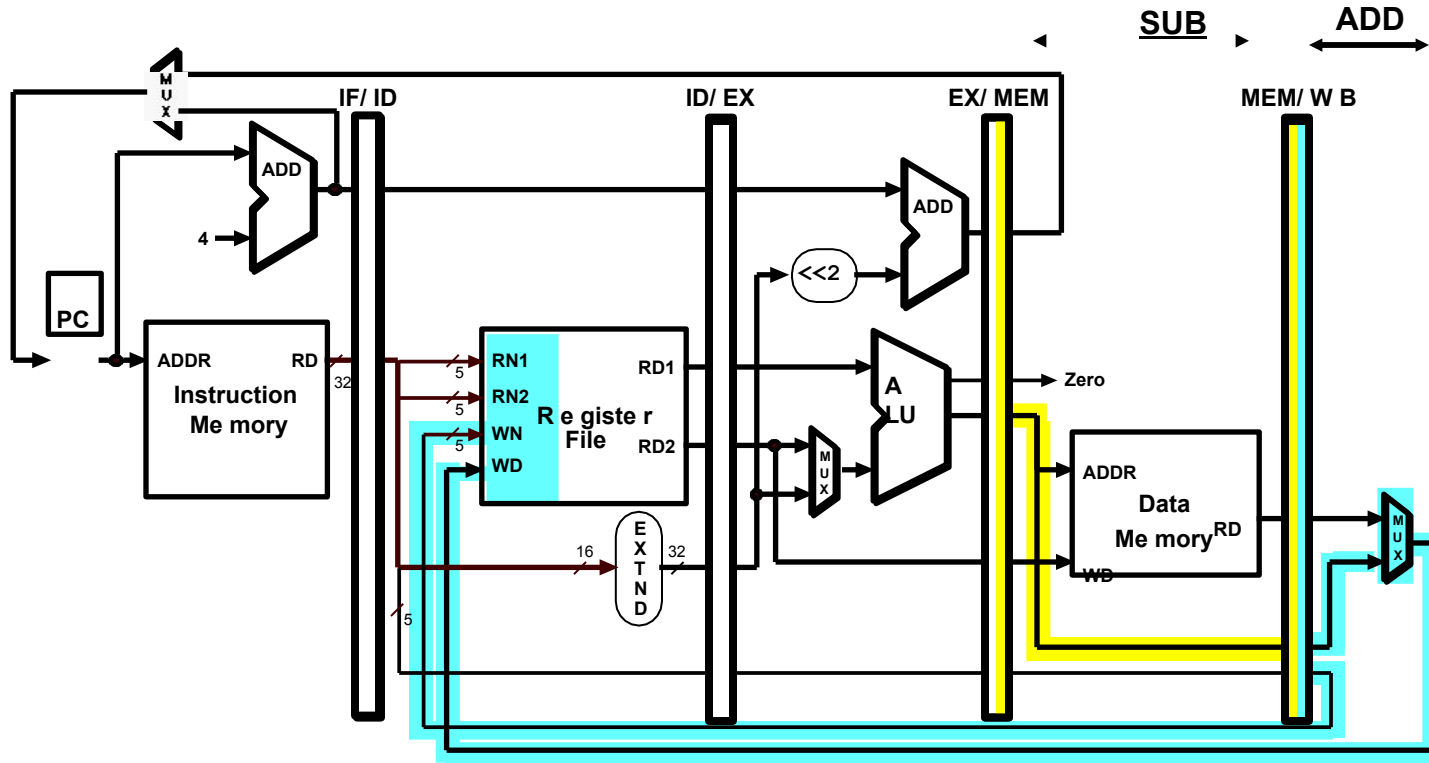


# Executing Multiple Instructions Clock Cycle



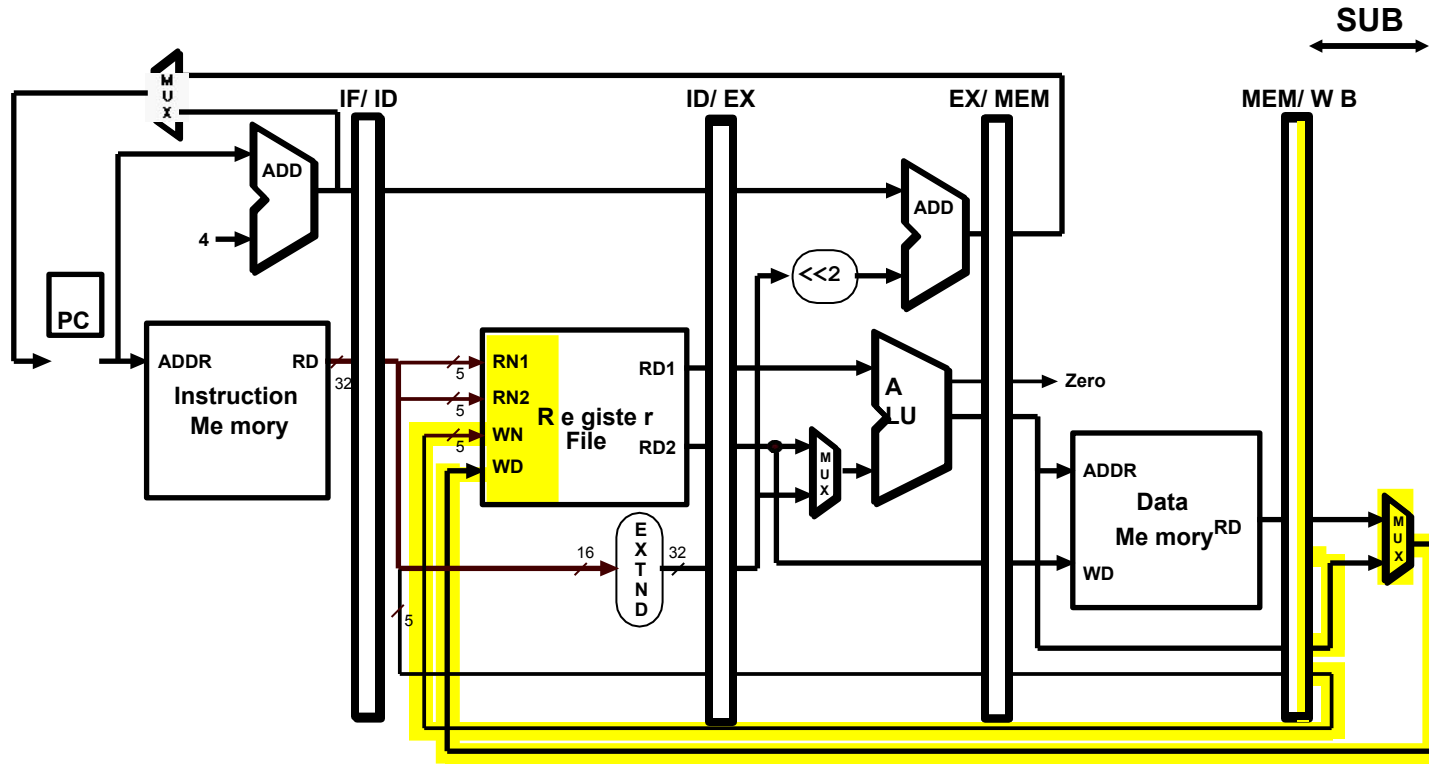
# Executing Multiple Instructions Clock Cycle

7



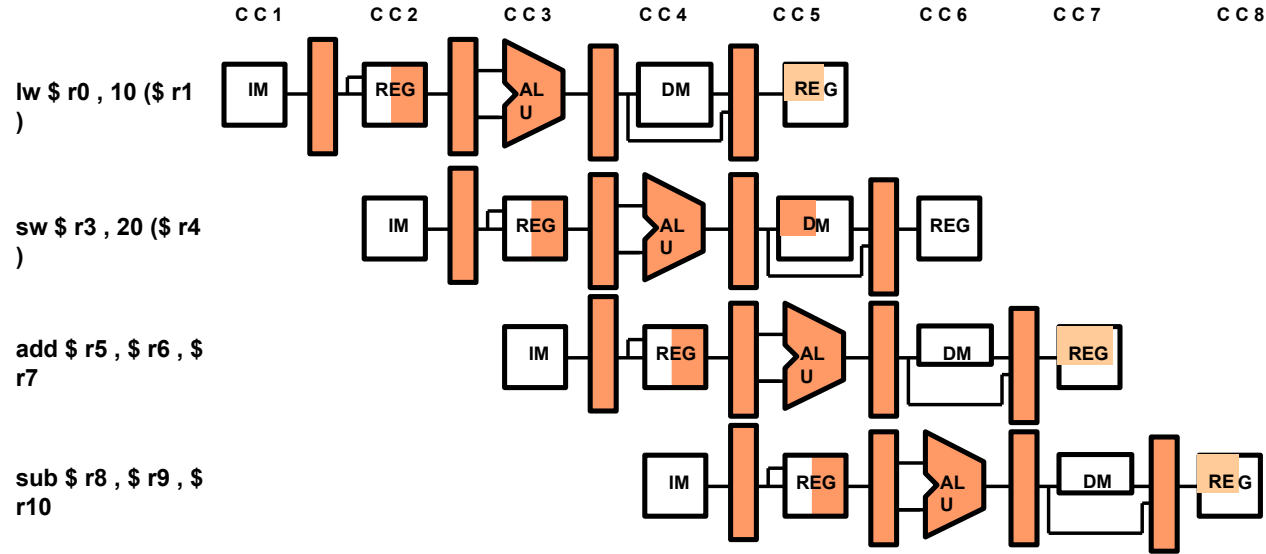
# Executing Multiple Instructions Clock Cycle

8

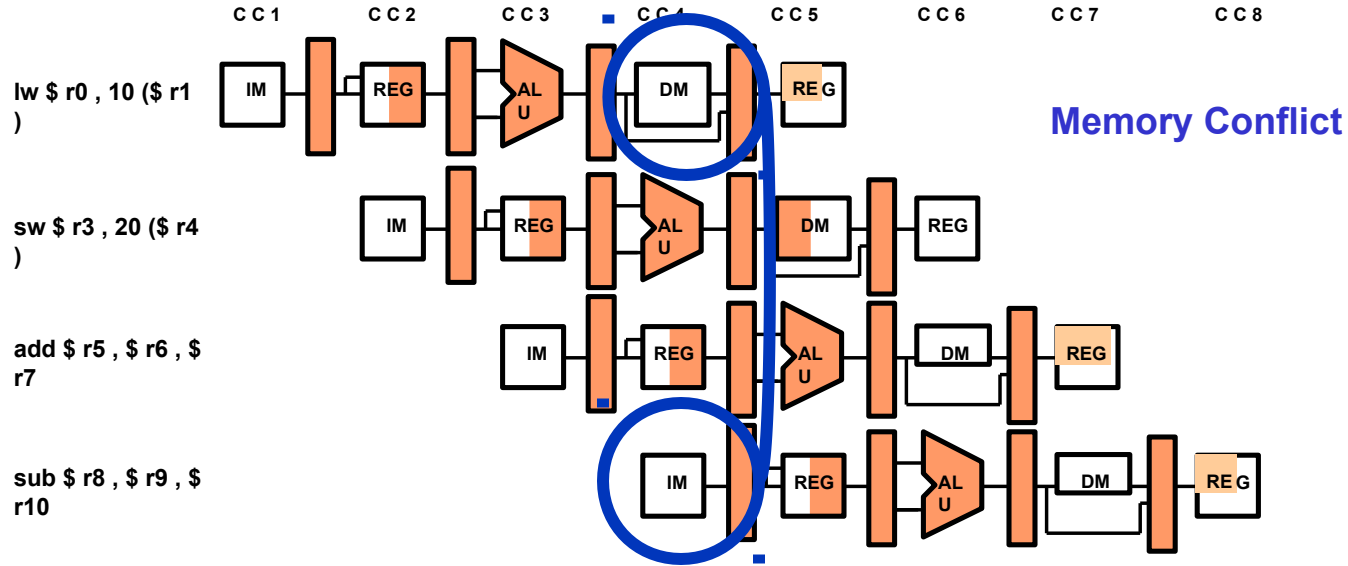




# Alternative View - Multicycle Diagram



# Alternative View - Multicycle Diagram



# Structural Hazards

## Some common Structural Hazards:

- **Memory:**
- **Floating point:**
- - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.
- **Starting up more of one type of instruction than there are resources.**
  - For instance, the PA-8600 can support two ALU + two load/store instructions per cycle - that's how much hardware it has available.

# Structural Hazards

## Dealing with Structural Hazards

**Stall** :Stalling: the dependent instruction is “pushed back” for one or more clock cycles.

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

### Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

### Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

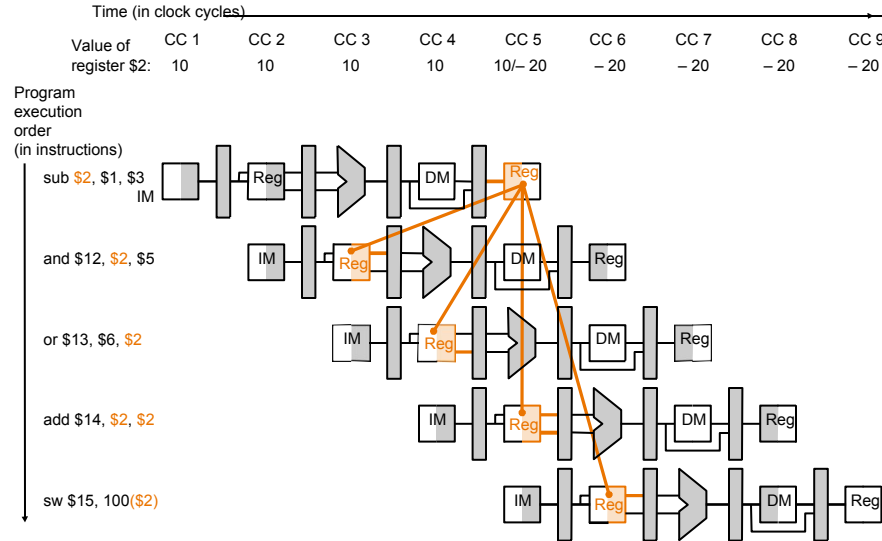
- Speed Up <= Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \frac{\text{Pipeline stall}}{\text{CPI}}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW)
  - Control

# Data Hazards

- Data hazards occur when data is used before it is ready



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register \$2 is not written until after those instructions read it.

# Data Hazards

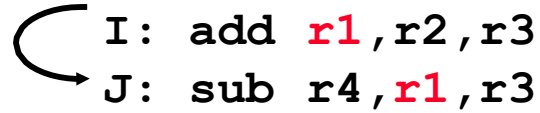
Execution Order is:

Instr<sub>i</sub>

Instr<sub>j</sub>

**Read After Write  
(RAW)**

Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it



I: add r1, r2, r3  
J: sub r4, r1, r3

- Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.

# Data Hazards

Execution Order is:


Instr<sub>i</sub>

Instr<sub>j</sub>

## Write After Read (WAR)

Instr<sub>j</sub> tries to write operand before Instr<sub>i</sub> reads it

– Gets wrong operand



```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

– Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages,
  - and Reads are always in stage 2, and Writes are always in stage 5



# Data Hazards

Execution Order is:

Instr<sub>i</sub>

Instr<sub>j</sub>

## Write After Write (WAW)

Instr<sub>j</sub> tries to write operand before Instr<sub>i</sub> writes it

– Leaves wrong result ( Instr<sub>i</sub> not

Instr<sub>j</sub>)

I: sub **r1**, r4, r3

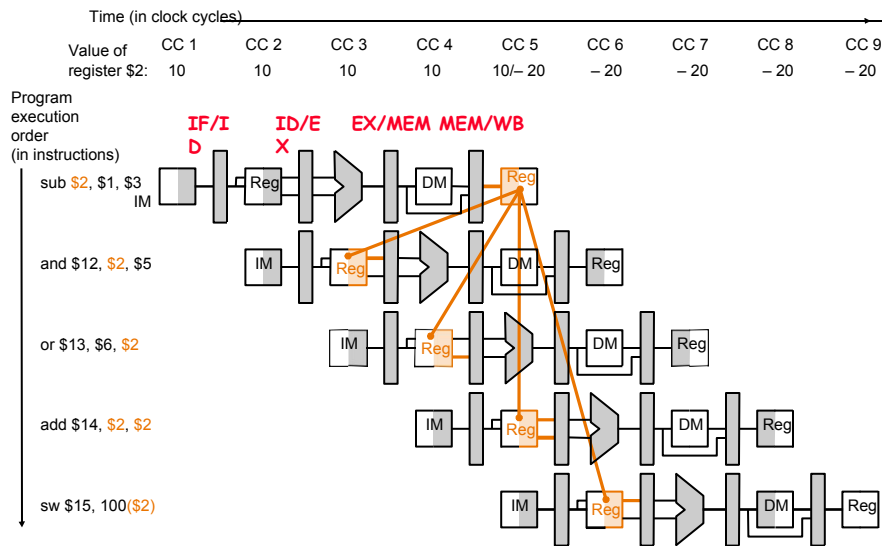
J: add **r1**, r2, r3

K: mul r6, r1, r7

- Called an “**output dependence**” by compiler writers This also results from the reuse of name “**r1**”.
- Can't happen in MIPS 5 stage pipeline because instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW later in more complicated pipes

# Data Hazard Detection in MIPS (1)

## Read after Write



1a:  $EX/MEM.RegisterRd =$

$ID/EX.RegisterRs$  1b:  $EX/MEM.RegisterRd = ID/EX.RegisterRt$

2a:  $MEM/WB.RegisterRd =$

$ID/EX.RegisterRs$  2b:

$MEM/WB.RegisterRd = ID/EX.RegisterRt$

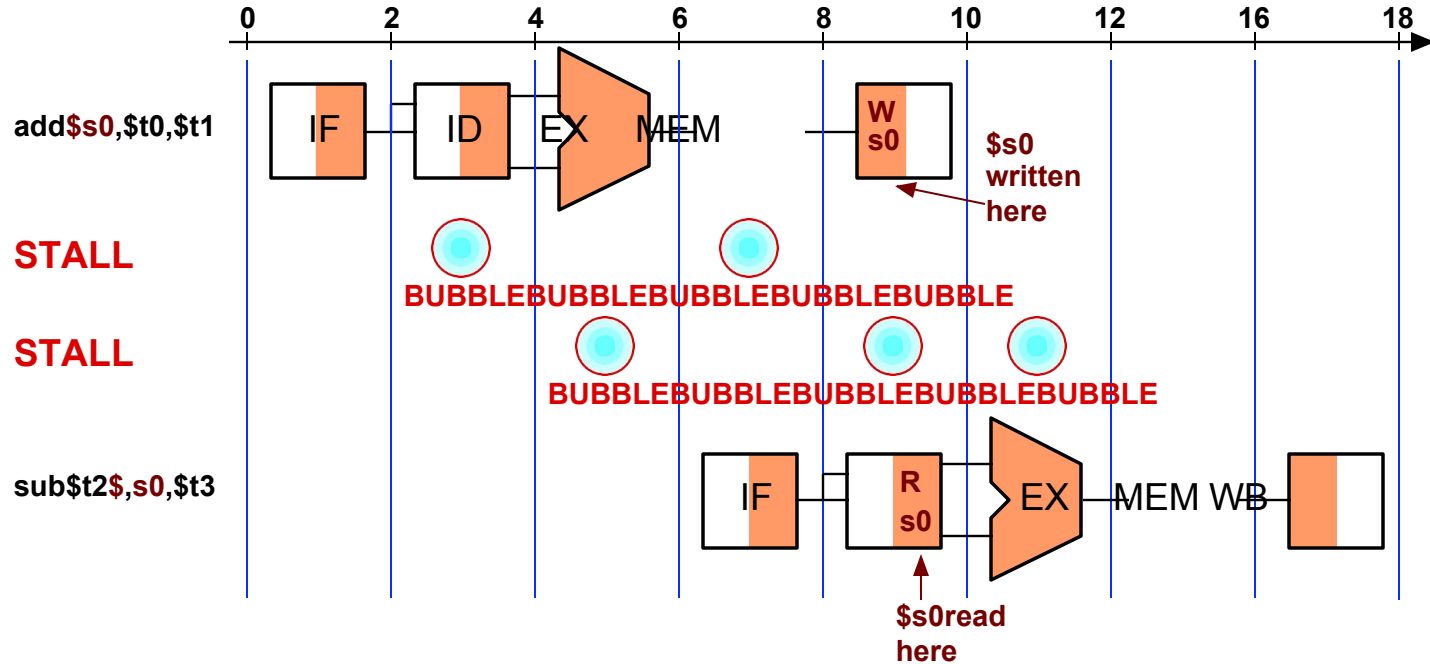
} EX hazard

} MEM hazard

# Data Hazards

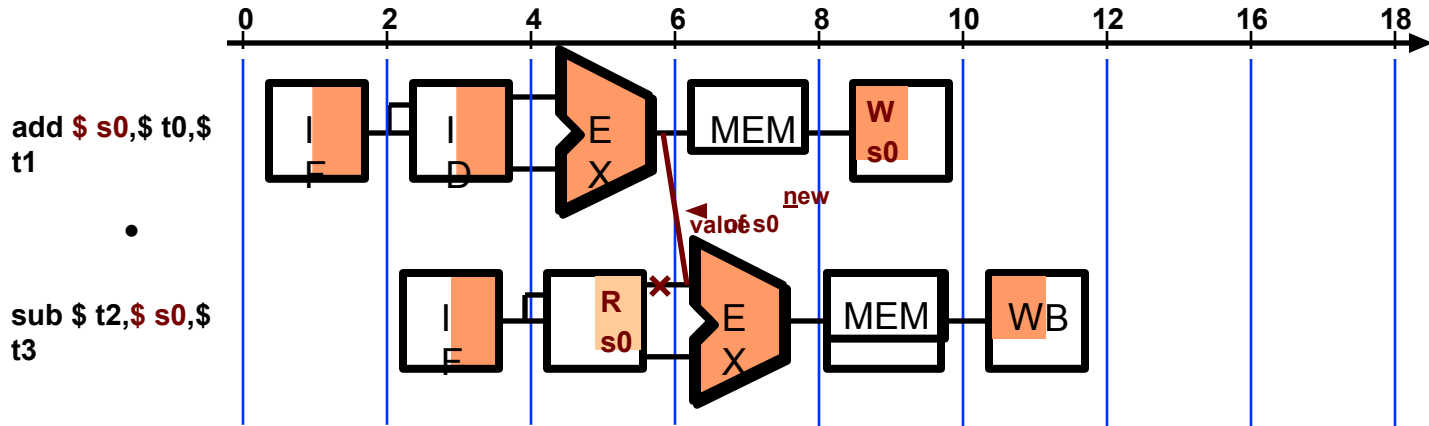
- **Solutions for Data Hazards**
  - **Stalling**
  - **Forwarding:**
    - » connect new value directly to next stage
  - **Reordering**

# Data Hazard - Stalling



# Data Hazards - Forwarding

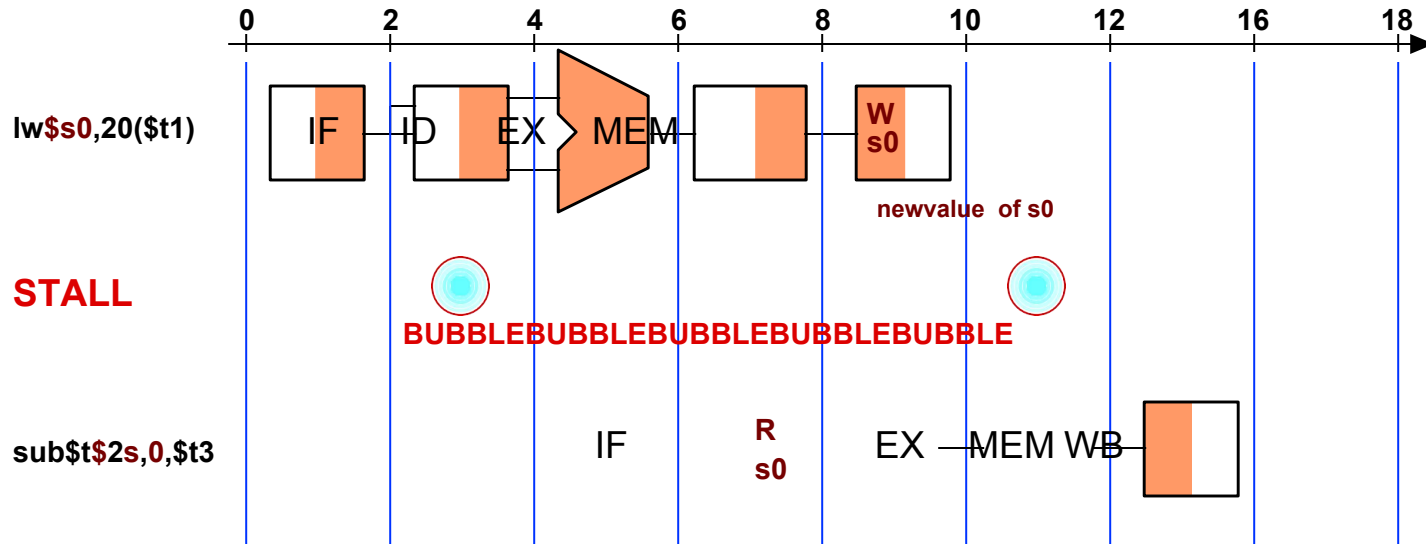
- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



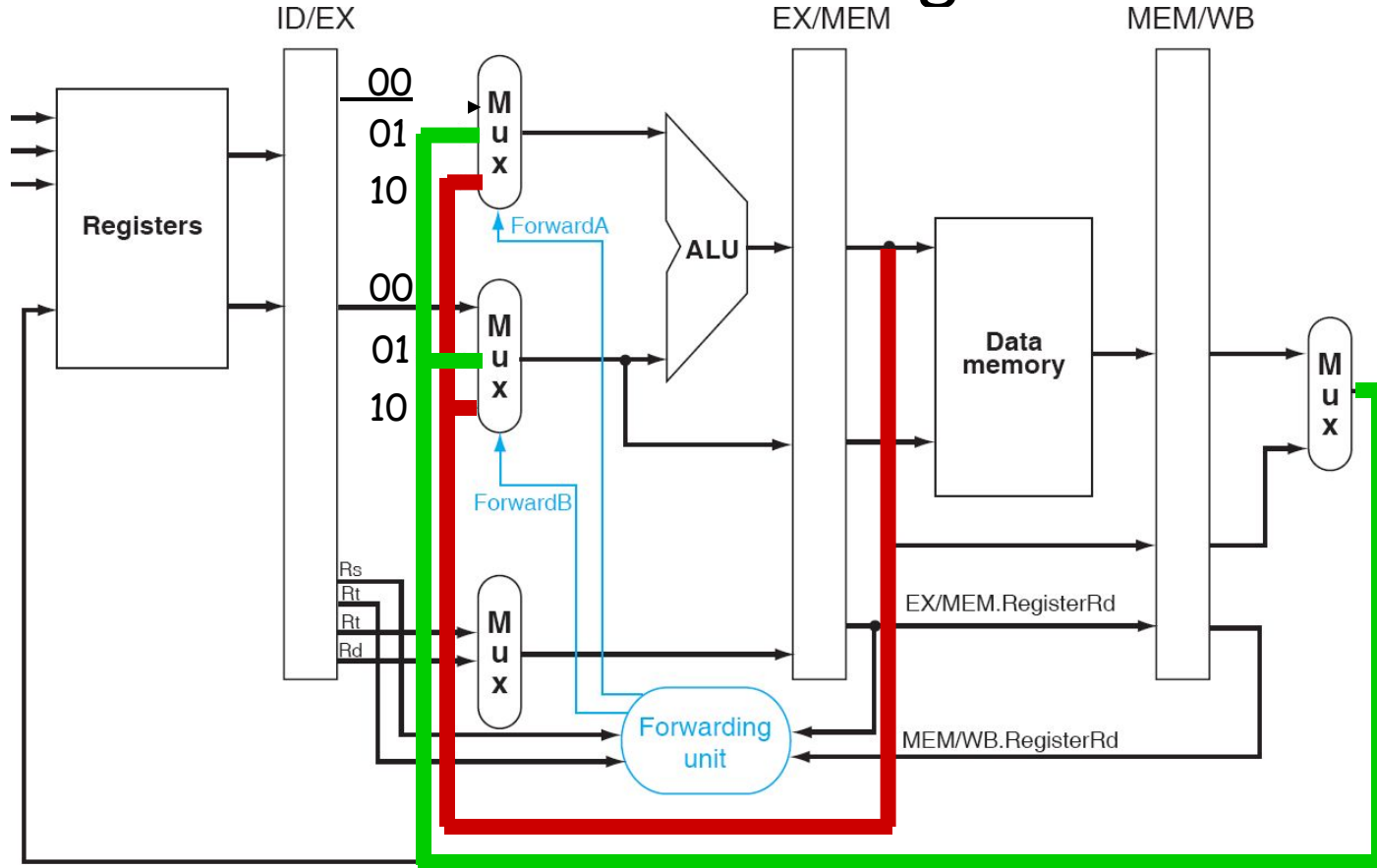
- Problem: what about load instructions?

# Data Hazards - Forwarding

- STALL still required for load - data avail. after MEM
- MIPS architecture calls this delayed load, initial implementations required compiler to deal with this



# Forwarding



Add hardware to feed back ALU and MEM results to both ALU inputs

# Data Hazards: A Classic Example

- Identify the data dependencies in the following code. Which of them can be resolved through forwarding?

**SUB \$2, \$1, \$3**

**OR \$12, \$2, \$5**

**SW \$13, 100(\$2)**

**ADD \$14, \$2, \$2**

**LW \$15, 100(\$2)**

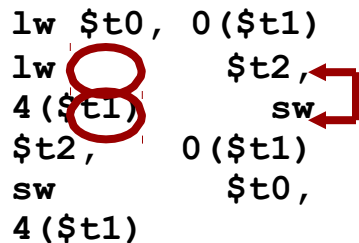
**ADD \$4, \$7, \$15**



# Data Hazards - Reordering Instructions

- Assuming we have data forwarding, what are the hazards in this code?

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```



- Reorder instructions to remove hazard:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

# Control Hazards

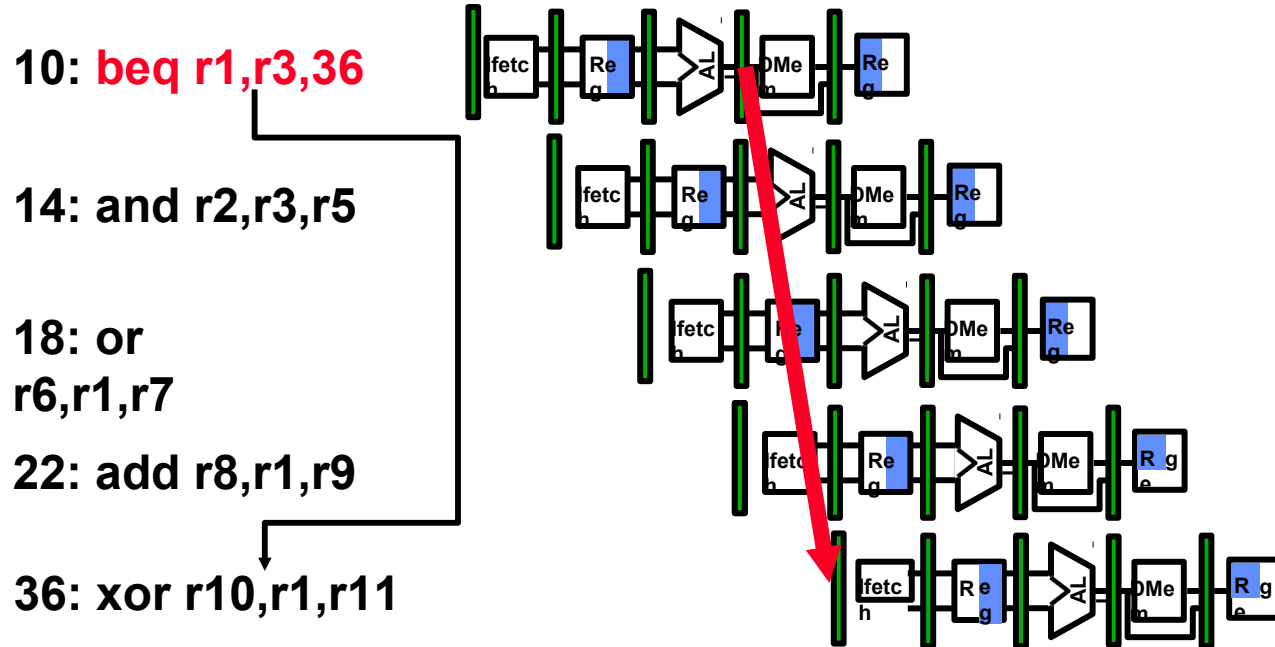
A **control hazard** is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

- **Taken**:  $PC \leq PC + 4 +$
- **Immediate Not Taken**:  $PC \leq$   
 $PC + 4$

# Control Hazards

Control Hazard on Branches  
Three Stage Stall



The penalty when branch take is 3  
cycles!

# Branch Hazards

- Just stalling for each branch is not practical
- Common assumption: branch not taken
- When assumption fails: flush three instructions

