

# BASIC PROCESSING UNIT

# Fundamental Concepts

- A **processor** is responsible for reading program instructions from the computer's memory and executing them. It fetches one instruction at a time.
- The processor uses the *program counter, PC*, to keep track of the *address of the next instruction* to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence.
- When an instruction is fetched, it is placed in the *instruction register, IR*, from where it is interpreted, or decoded, by the processor's control circuitry. The IR holds the instruction until its execution is completed.
- Fetching an instruction and loading it into the IR is usually referred to as the *instruction **fetch phase***. Performing the operation specified in the instruction constitutes the *instruction **execution phase***.

# RISC-style instruction set

## architecture

To execute an instruction, the processor has to perform the following steps:

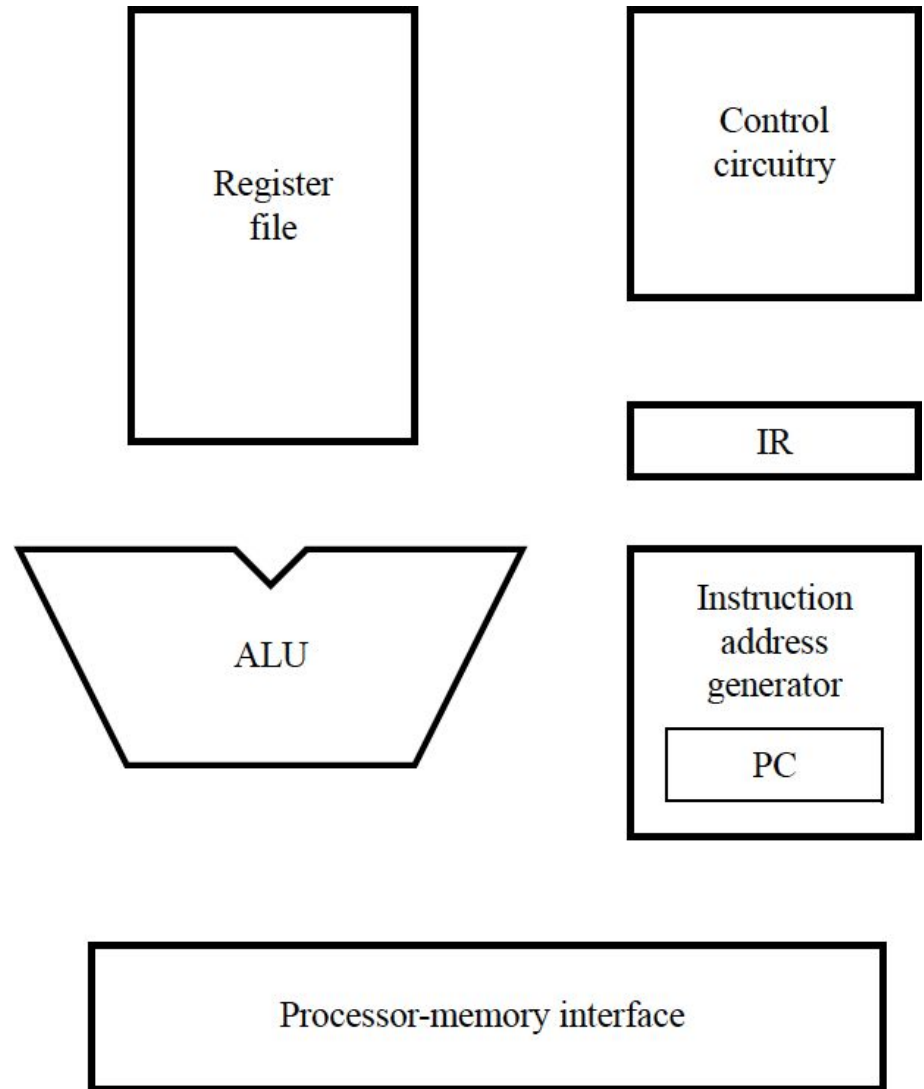
1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR.  
In register transfer notation, the required action is  $IR \leftarrow [[PC]]$
2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is  $PC \leftarrow [PC] + 4$
3. Carry out the operation specified by the instruction in the IR.

The operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

# Processor's building blocks(or) Main hardware components of a processor

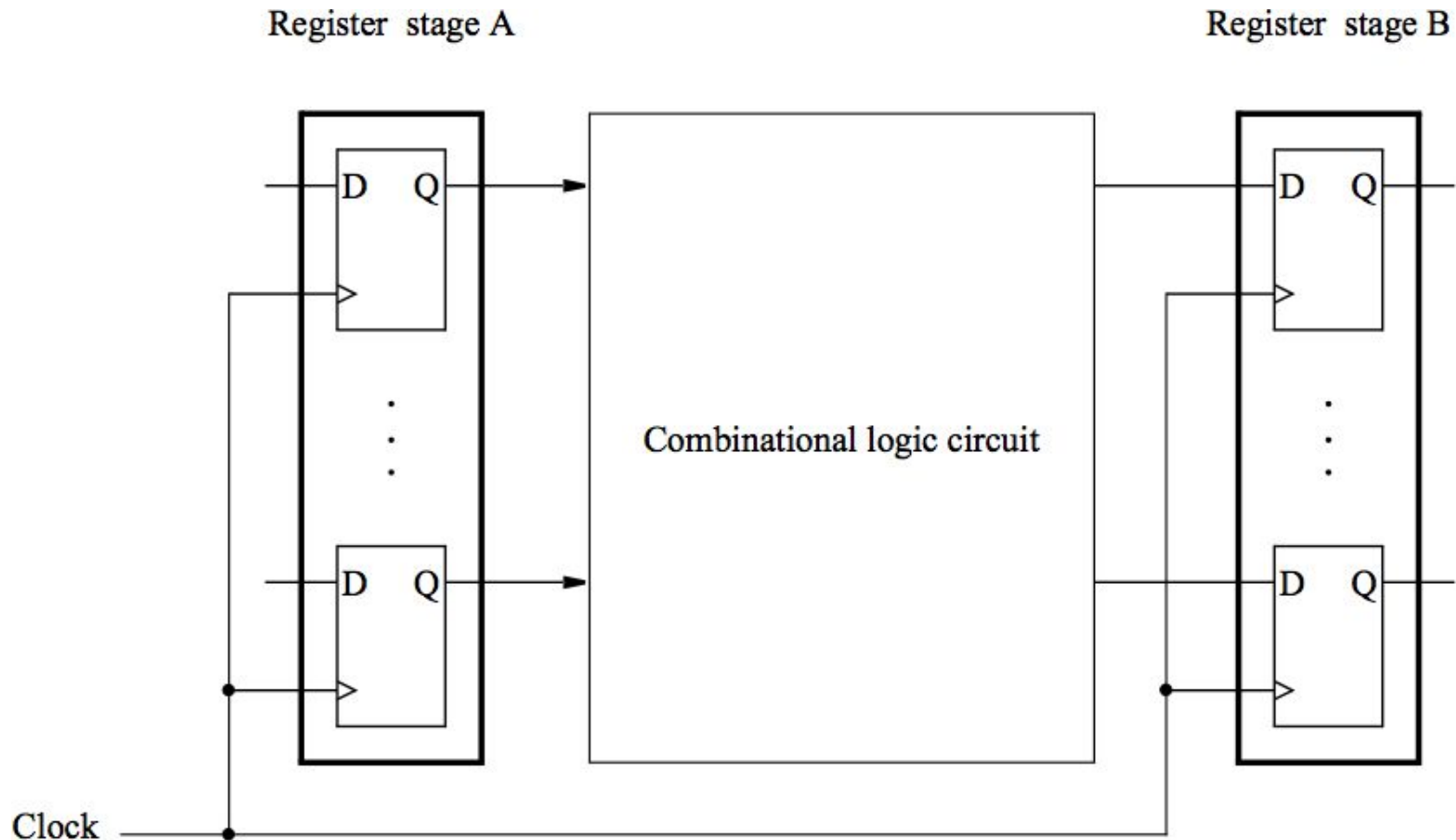
- PC provides instruction address.
- Instruction is fetched into IR
- Instruction address generator updates PC
- Control circuitry interprets instruction and generates control signals to perform the actions needed.



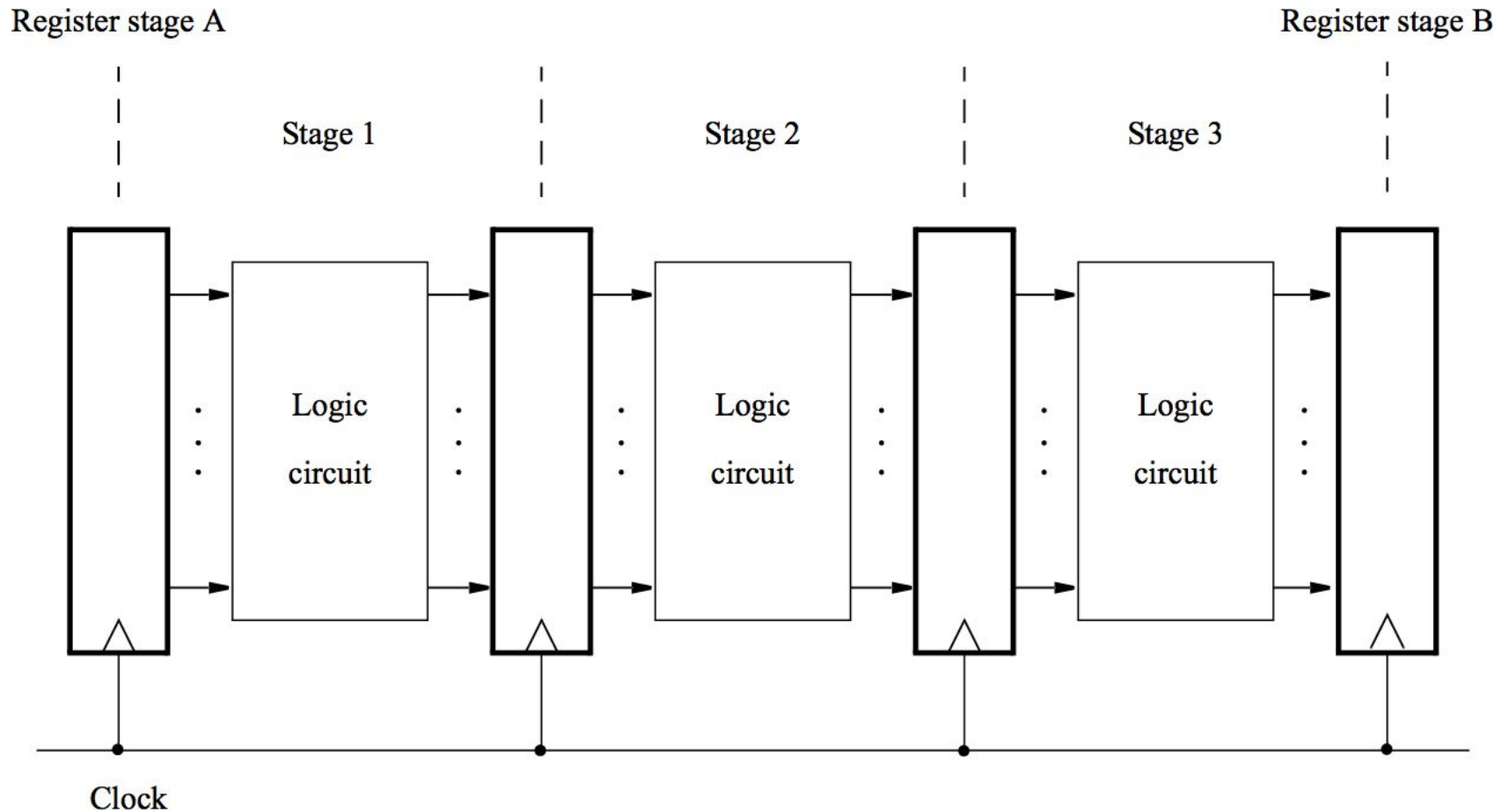
- The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations.
- The instruction address generator updates the contents of the PC after every instruction is fetched.
- The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers.
- During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic unit (ALU), which performs the required computation.
- The results of the computation are stored in a register in the register file.

# A digital processing system

Contents of register A are processed and deposited in register B.



# A multi-stage digital processing system





# Why multi-stage?

- Processing moves from one stage to the next in each clock cycle.
- Such a multi-stage system is known as a **pipeline**.
- High-performance processors have a pipelined organization.
- Pipelining enables the execution of successive instructions to be overlapped.

# Instruction execution

- Pipelined organization is most effective if all instructions can be executed in the same number of steps.
- Each step is carried out in a separate hardware stage.
- Processor design will be illustrated using five hardware stages.
- How can instruction execution be divided into five steps?

## Load Instructions

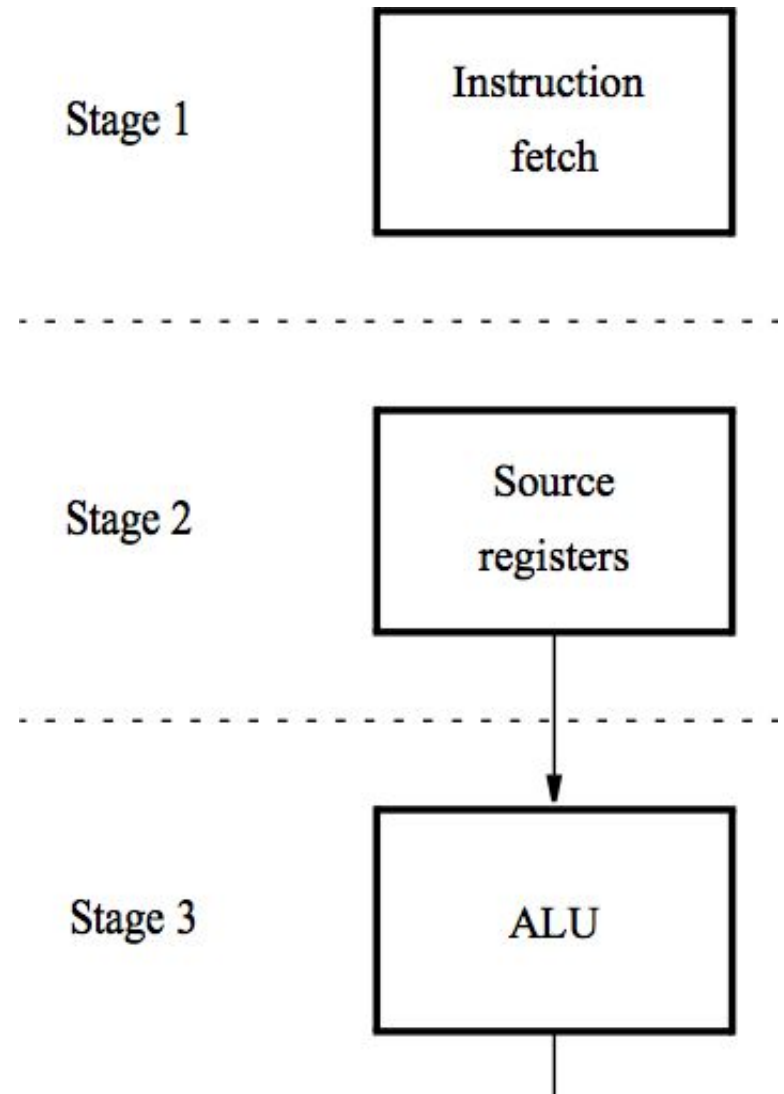
Consider the instruction `Load R5, X(R7)` which uses the Index addressing mode to load a word of data from memory location  $X + [R7]$  into register R5.

Execution of this instruction involves the following actions:

- Fetch the instruction from the memory.
- Increment the program counter.
- Decode the instruction to determine the operation to be performed.
- Read register R7.
- Add the immediate value  $X$  to the contents of R7.
- Use the sum  $X + [R7]$  as the effective address of the source operand, and read the contents of that location in the memory.
- Load the data received from the memory into the destination register, R5.

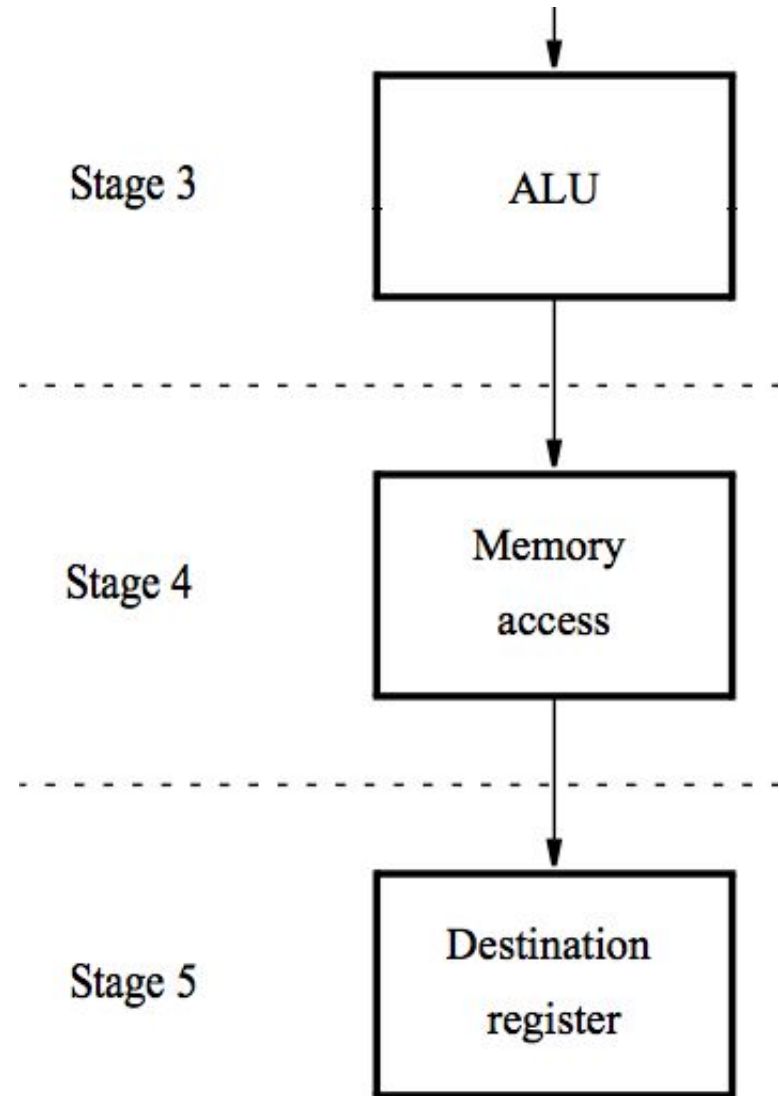
# A 5-stage implementation of a RISC processor

- Instruction processing moves from stage to stage in every clock cycle, starting with fetch.
- The instruction is decoded and the source registers are read in stage 2.
- Computation takes place in the ALU in stage 3.



# A 5-stage implementation of a RISC processor

- If a memory operation is involved, it takes place in stage 4.
- The result of the instruction is stored in the destination register in stage 5.



## A memory access instruction: Load R5, X(R7)

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of register R7 in the register file.
3. Compute the effective address.
4. Read the memory source operand.
5. Load the operand into the destination register, R5.

## A computational instruction: Add R3, R4, R5

1. Fetch the instruction and increment the program counter.
  2. Decode the instruction and read registers R4 and R5.
  3. Compute the sum  $[R4] + [R5]$ .
  4. No action.
  5. Load the result into the destination register, R3.
- *Stage 4 (memory access) is not involved in this instruction.*

## A computational instruction: Add R3, R4, #1000

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R4 .
3. Compute the sum  $[R4] + 1000$ .
4. No action.
5. Load the result into the destination register, R3.



## Instruction storing into memory: Store R6, X(R8)

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R6 and R8.
3. Compute the effective address  $X + [R8]$ .
4. Store the contents of register R6 into memory location  $X + [R8]$ .
5. No action.

## Summary – Actions to implement an instruction

1. Fetch an instruction and increment the program counter.
  2. Decode the instruction and read registers from the register file.
  3. Perform an ALU operation.
  4. Read or write memory data if the instruction involves a memory operand.
  5. Write the result into the destination register.
- This sequence determines the hardware stages needed.

- Instruction processing consists of two phases: the fetch phase and the execution phase.
- It is convenient to divide the processor hardware into two corresponding sections. [Instruction Fetch Section and Execution Section]
- Instruction Fetch Section fetches instructions and the Execution Section executes them.
- Instruction Fetch Section that fetches instructions is also responsible for decoding them and for generating the control signals that cause appropriate actions to take place in the execution section.
- Execution section reads the data operands specified in an instruction, performs the required computations, and stores the results. So, it is called as Data Path
- To organize the hardware into a multi-stage structure, with stages corresponding to the five steps. Among 5 stages, stage1 is in Instruction Fetch Section and from stage2 to stage5 is in Data path.

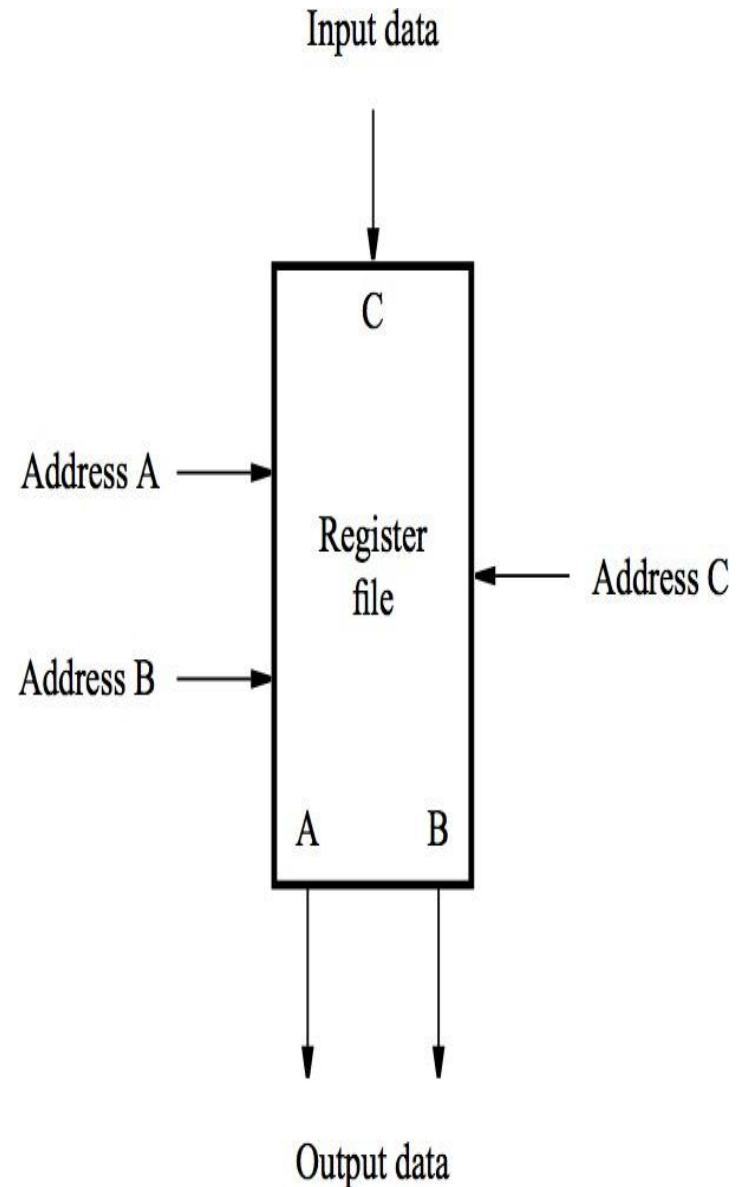
## Datapath (Stages 2 to 5)

- Register file constitutes stage 2. It is necessary to insert registers between stages. Inter-stage registers hold the results produced in one stage so that they can be used as inputs to the next stage during the next clock cycle. Data read from the register file are placed in registers RA and RB.
- Register RA provides the data to input InA of the ALU. Multiplexer MuxB forwards either the contents of RB or the immediate value in the IR to the ALU's second input, InB. The ALU constitutes stage 3, and the result of the computation it performs is placed in register RZ.

# Hardware components: Register file

(stage2)

- General-purpose registers are usually implemented in the form of a register file, which is a small and fast memory block.
- It consists of an array of storage elements, with access circuitry that enables data to be read from or written into any register.
- The access circuitry is designed to enable two registers to be read at the same time, making their contents available at two separate outputs, A and B.



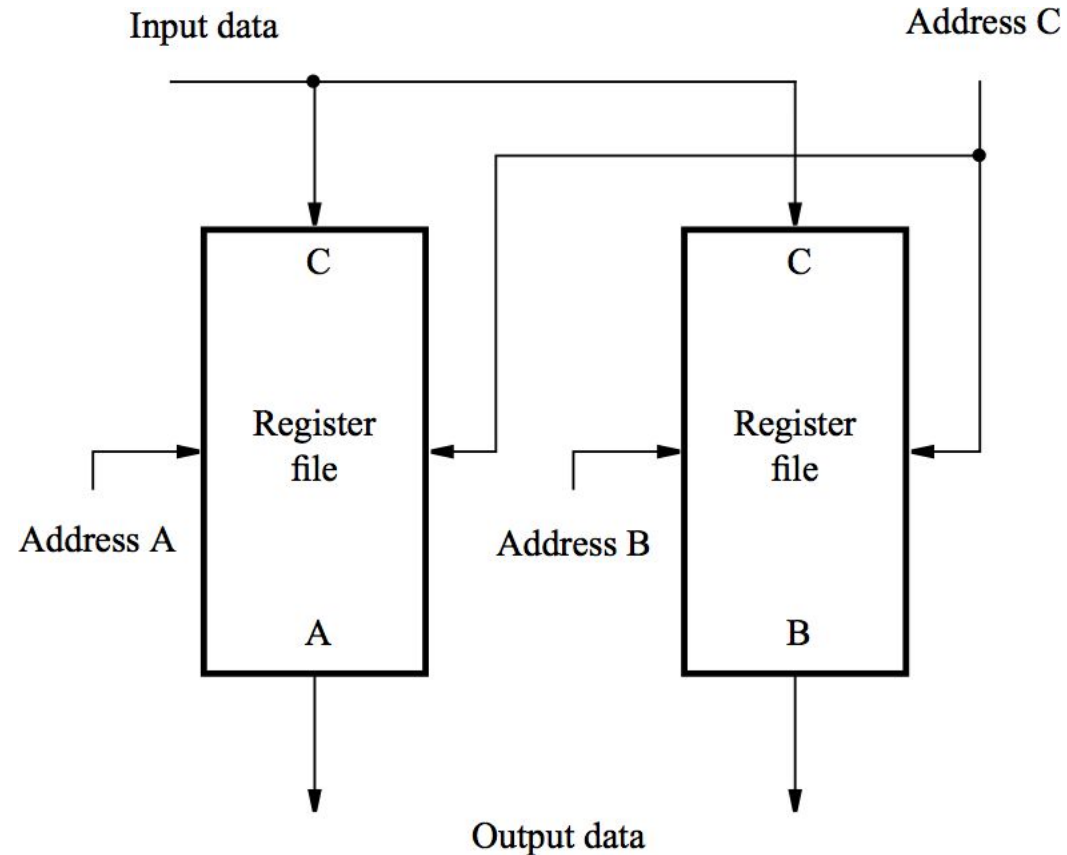
- The register file has two address inputs that select the two registers to be read. These inputs are connected to the fields in the IR that specify the source registers, so that the required registers can be read.
- The register file also has a data input, C, and a corresponding address input to select the register into which data are to be written.
- This address input is connected to the IR field that specifies the destination register of the instruction.
- The inputs and outputs of any memory unit are often called input and output *ports*. A memory unit that has two output ports is said to be *dual-ported*.

To use a single set of registers with two data paths and access circuitry that enable two registers to be read at the same time.

# Alternative implementation of 2-port register file

An alternative is

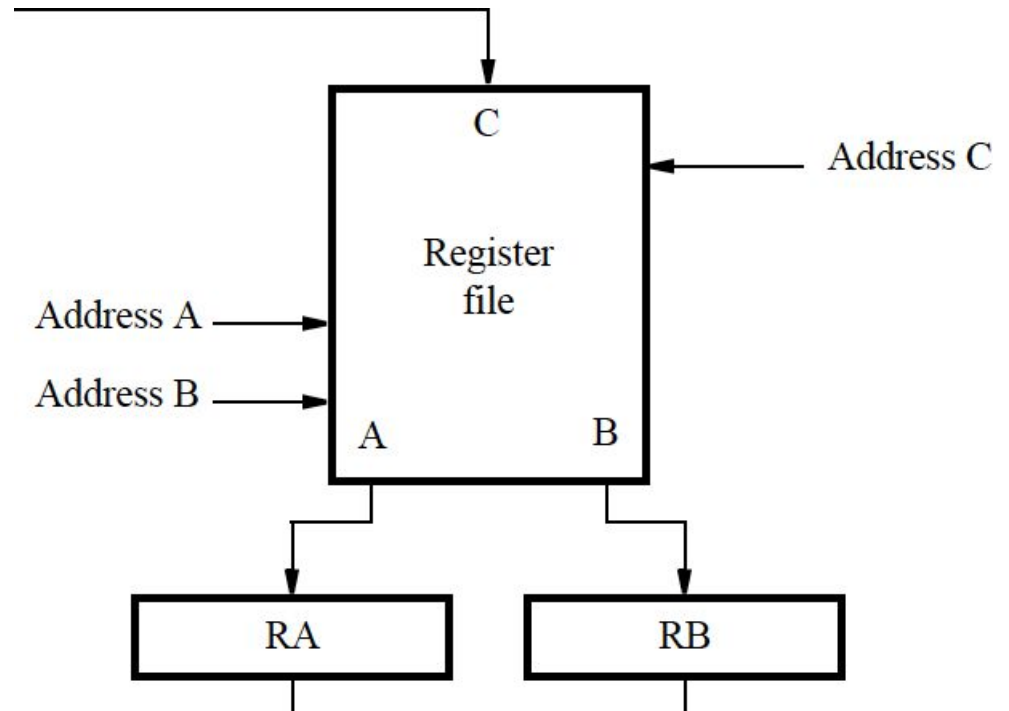
- To use two memory blocks, each containing one copy of the register file. Whenever data are written into a register, they are written into both copies of that register.
- Thus, the two files have identical contents. When an instruction requires data from two registers, one register is accessed in each file.
- In effect, the two register files together function as a single dual-ported register file.



- Using two single-ported memory blocks.

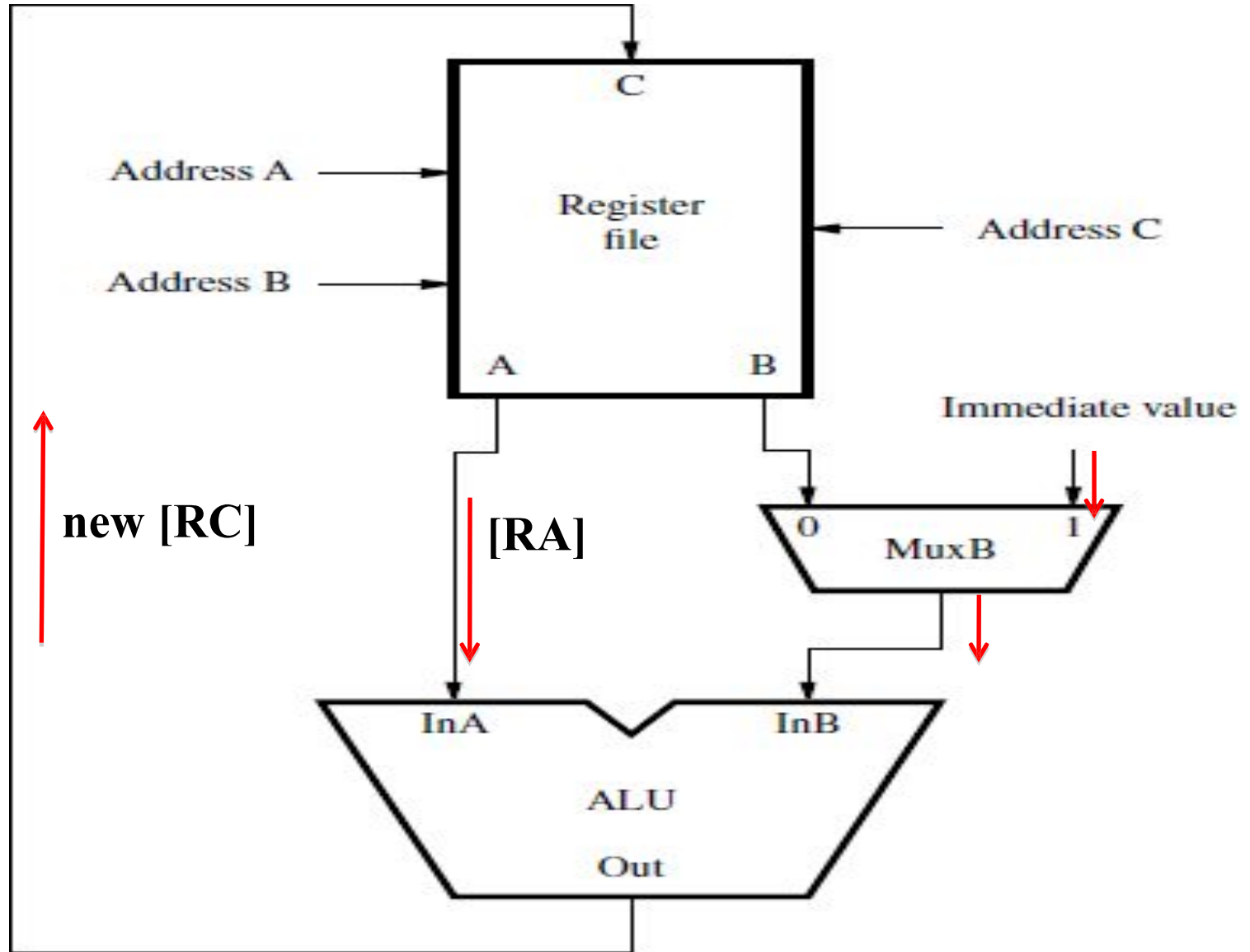
# Register file – Stages 2 & 5 with Inter-stage registers

- Address inputs are connected to the corresponding fields in IR.
- Source registers are read in stage 2; their contents are stored in RA and RB.
- In stage 5, the result of the instruction is stored in the destination register selected by address C.





# ALU(stage3)

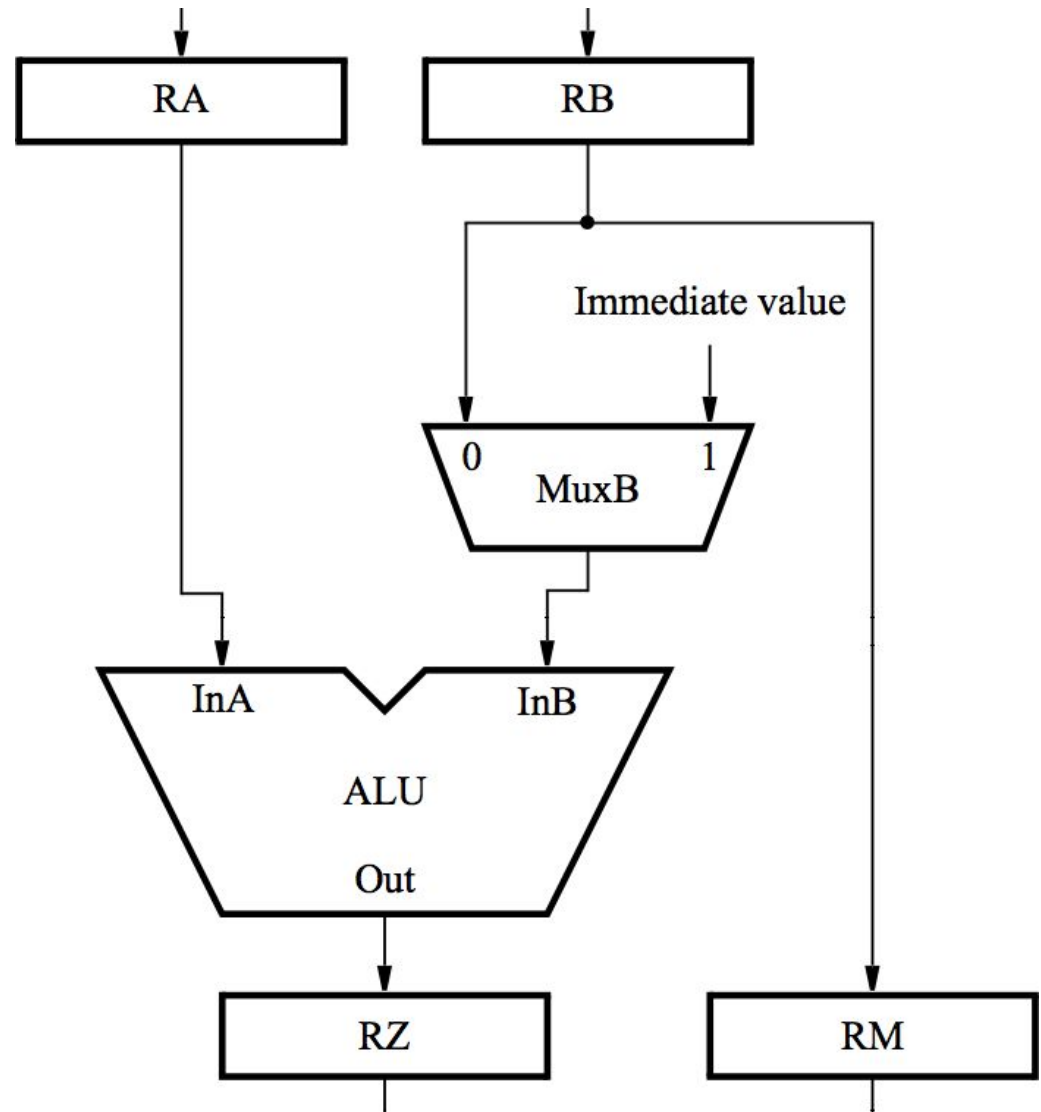


Conceptual view of the hardware needed for computation

- The arithmetic and logic unit is used to manipulate data. It performs arithmetic operations such as addition and subtraction, and logic operations such as AND, OR, and XOR. Conceptually, the register file and the ALU may be connected .
- When an instruction that performs an arithmetic or logic operation is being executed, the contents of the two registers specified in the instruction are read from the register file and become available at outputs A and B. Output A is connected directly to the first input of the ALU, InA, and output B is connected to a multiplexer, MuxB.
- The multiplexer selects either output B of the register file or the immediate value in the IR to be connected to the second ALU input, InB.
- The output of the ALU is connected to the data input, C, of the register file so that the results of a computation can be loaded into the destination register.

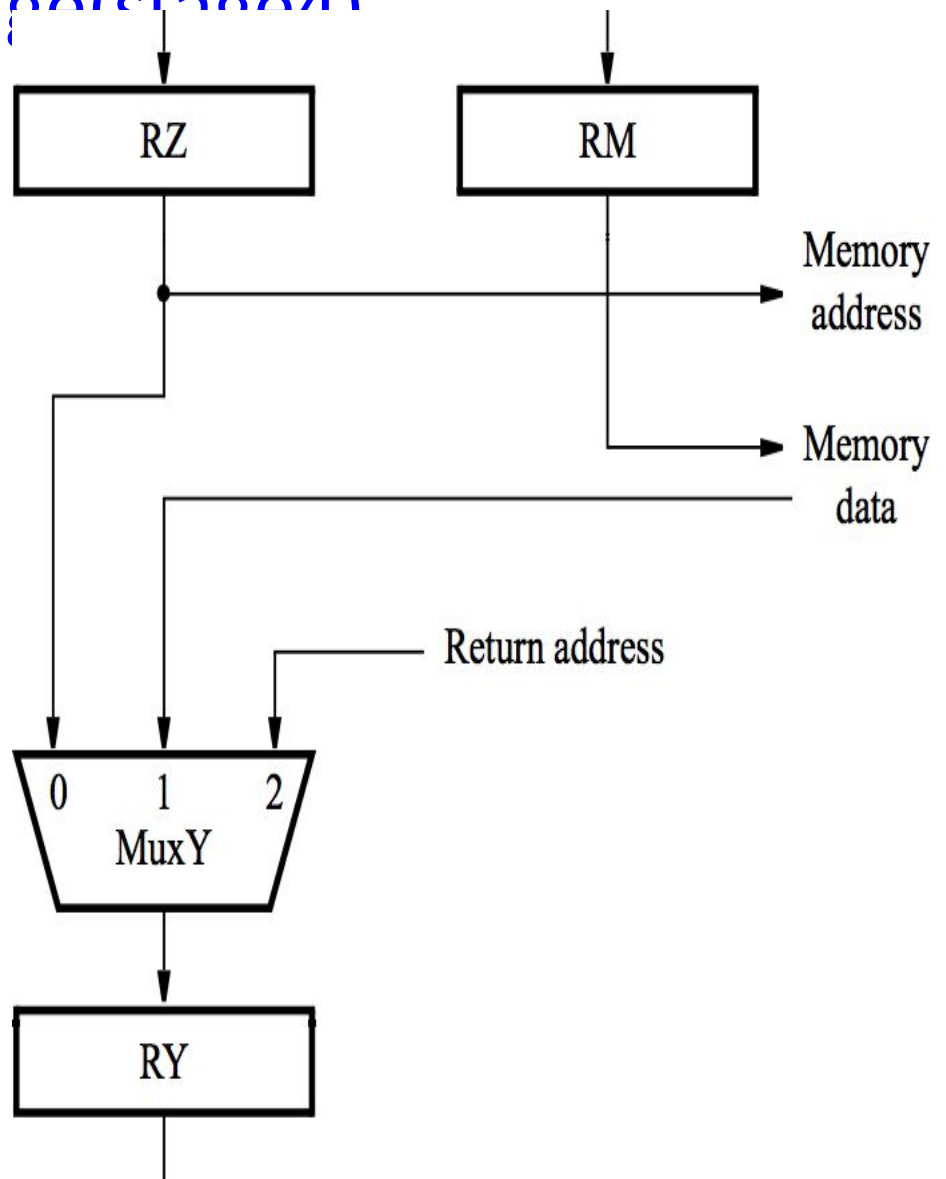
# ALU stage with Inter-stage registers

- ALU performs calculation specified by the instruction.
- Multiplexer MuxB selects either RB or the Immediate field of IR.
- Results stored in RZ.
- Data to be written in the memory are transferred from RB to RM.

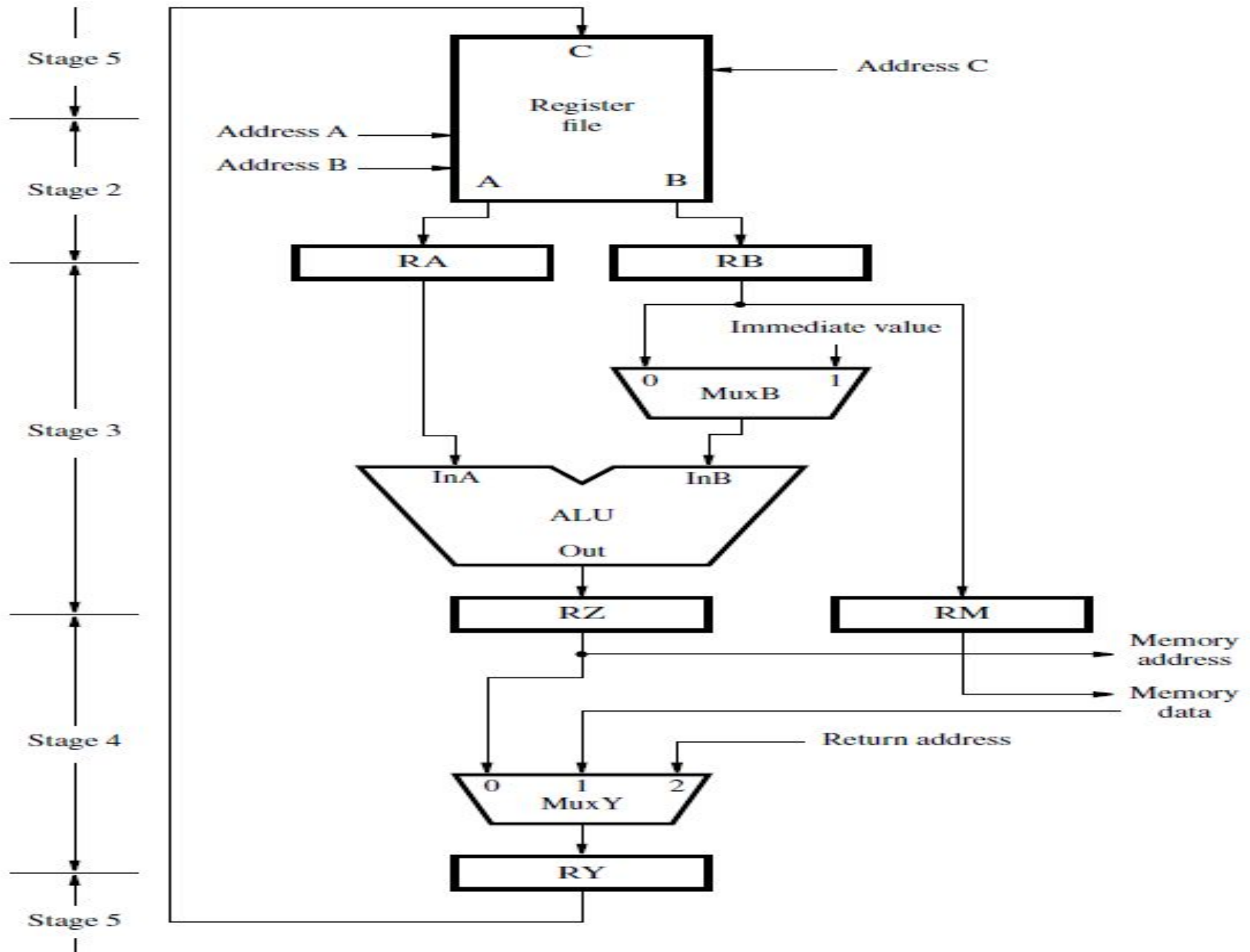


- For a memory instruction, RZ provides memory address, and MuxY selects read data to be placed in RY [ex: Load R5, X(R7)] go into Register file (stage5).
- RM provides data for a memory write operation. [ex: Store R6, X(R8)]
- For a calculation[Add R3, R4, R5] instruction, MuxY selects [RZ] to be placed in RY and go into Register file (stage5).
- Input 2 of MuxY is used in subroutine calls.

## Memory stage(stage4)



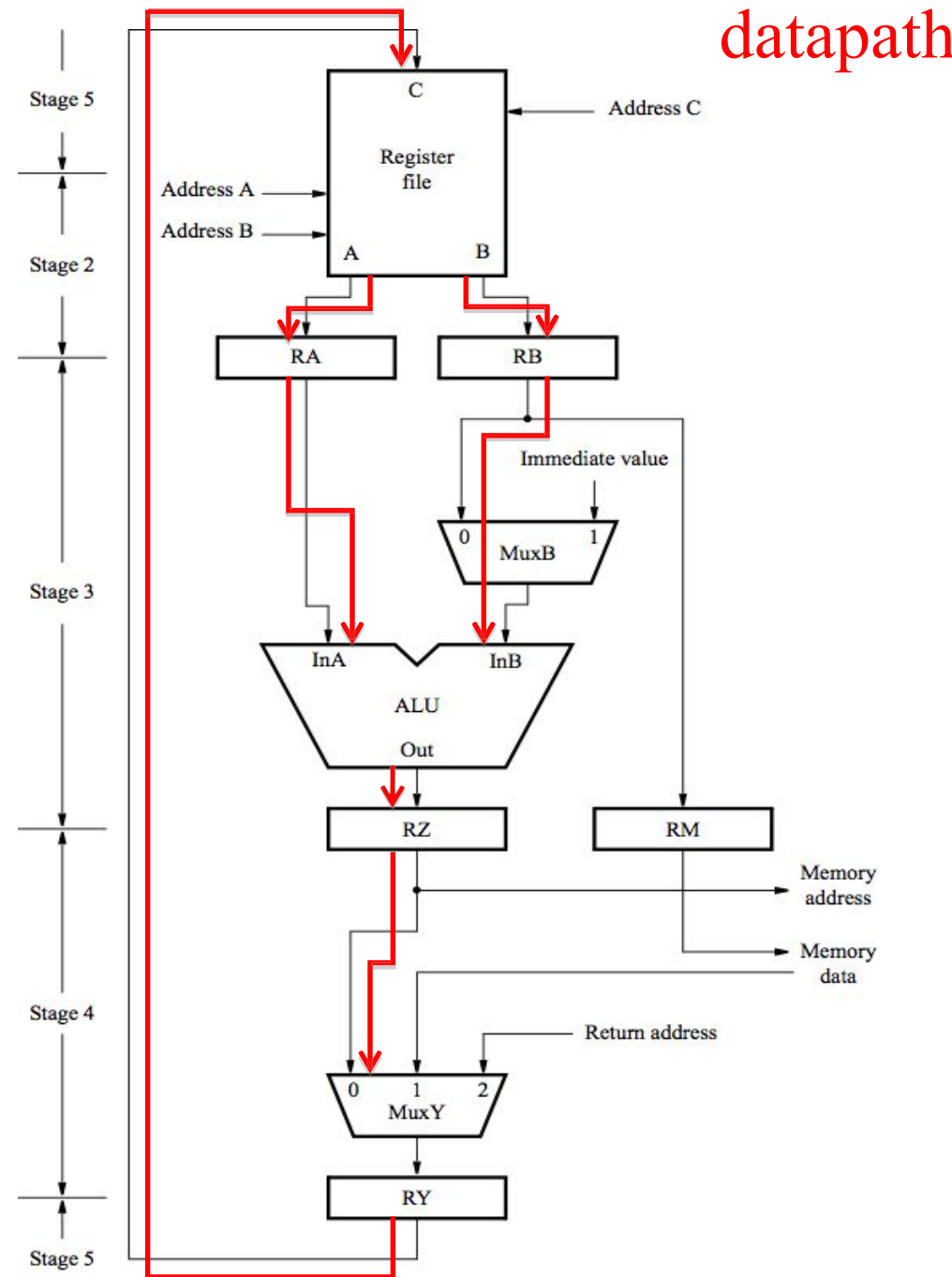
# The datapath in a Processor – Stages 2 to 5



# Instruction Fetch and Execution Steps

Example: Add R3, R4, R5

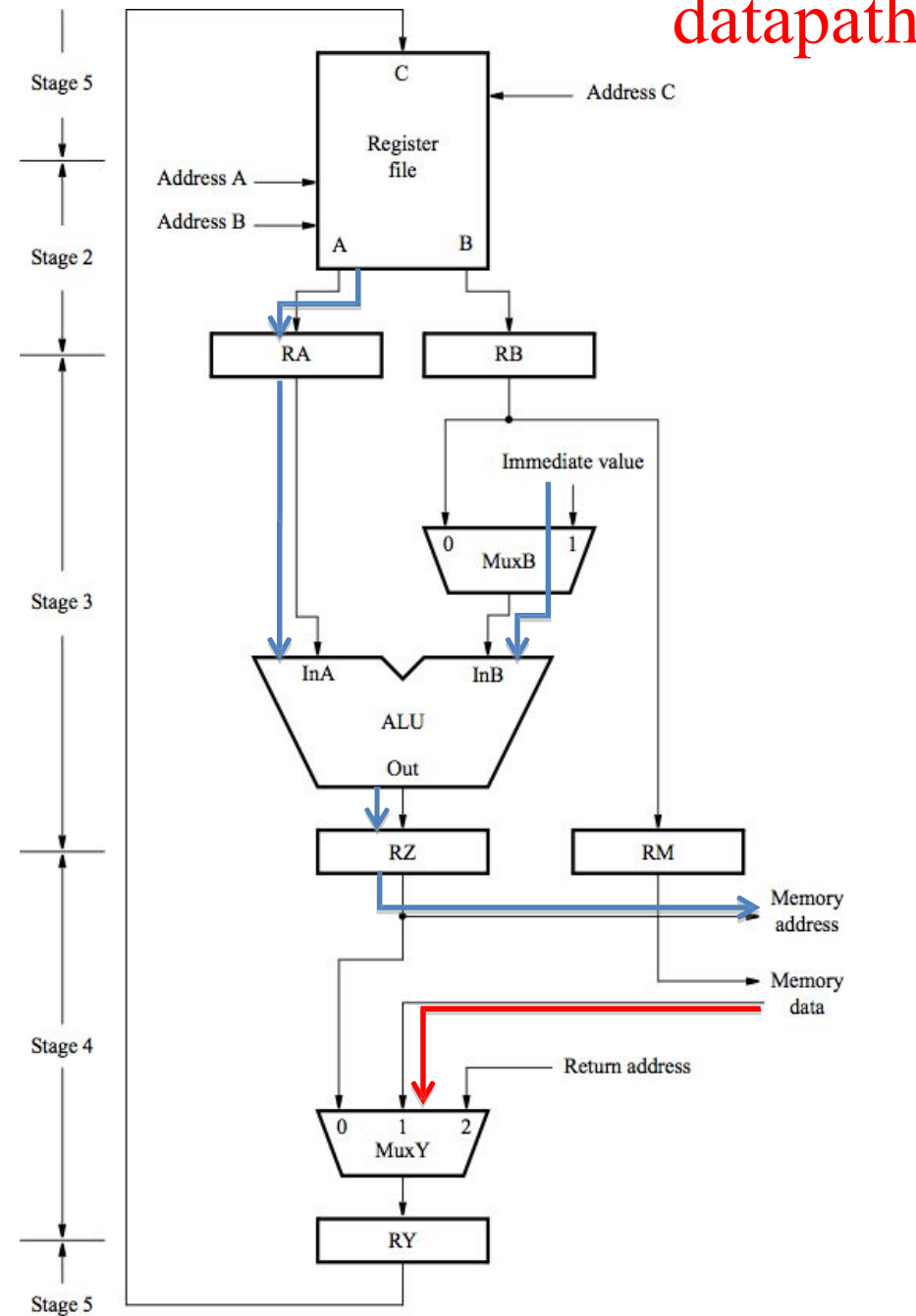
1. Memory address  $\leftarrow [PC]$ ,  
Read memory,  $IR \leftarrow$  Memory  
data,  $PC \leftarrow [PC] + 4$
2. Decode instruction,  
 $RA \leftarrow [R4]$ ,  $RB \leftarrow [R5]$
3.  $RZ \leftarrow [RA] + [RB]$
4.  $RY \leftarrow [RZ]$
5.  $R3 \leftarrow [RY]$



# Instruction Fetch and Execution Steps

## Example: Load R5, X(R7)

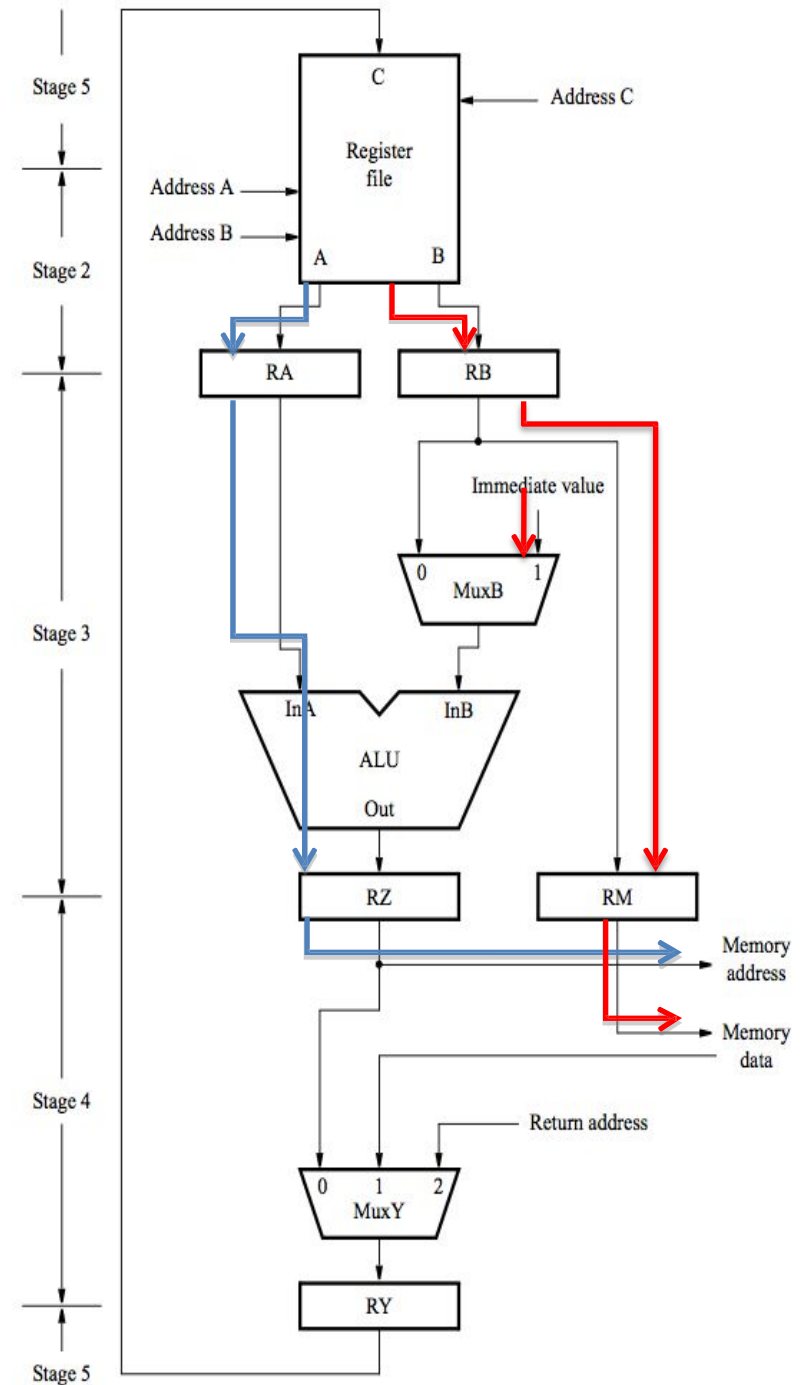
1. Memory address  $\leftarrow [PC]$ ,  
Read memory,  $IR \leftarrow$   
Memory data,  $PC \leftarrow [PC] + 4$
2. Decode instruction,  $RA \leftarrow [R7]$
3.  $RZ \leftarrow [RA] + \text{Immediate value } X$
4. Memory address  $\leftarrow [RZ]$ ,  
Read memory,  $RY \leftarrow$   
Memory data
5.  $R5 \leftarrow [RY]$



# Instruction Fetch and Execution Steps

## Example: Store R6, X(R8)

1. Memory address  $\leftarrow [PC]$ , Read memory,  $IR \leftarrow$  Memory data,  $PC \leftarrow [PC] + 4$
2. Decode instruction,  $RA \leftarrow [R8]$ ,  $RB \leftarrow [R6]$
3.  $RZ \leftarrow [RA] + \text{Immediate value}$ ,  $RM \leftarrow [RB]$
4. Memory address  $\leftarrow [RZ]$ , Memory data  $\leftarrow [RM]$ , Write memory
5. No action





# Instruction Fetch Section

## Memory address generation

□ The addresses used to access the memory come from two sources.

1. From the PC and

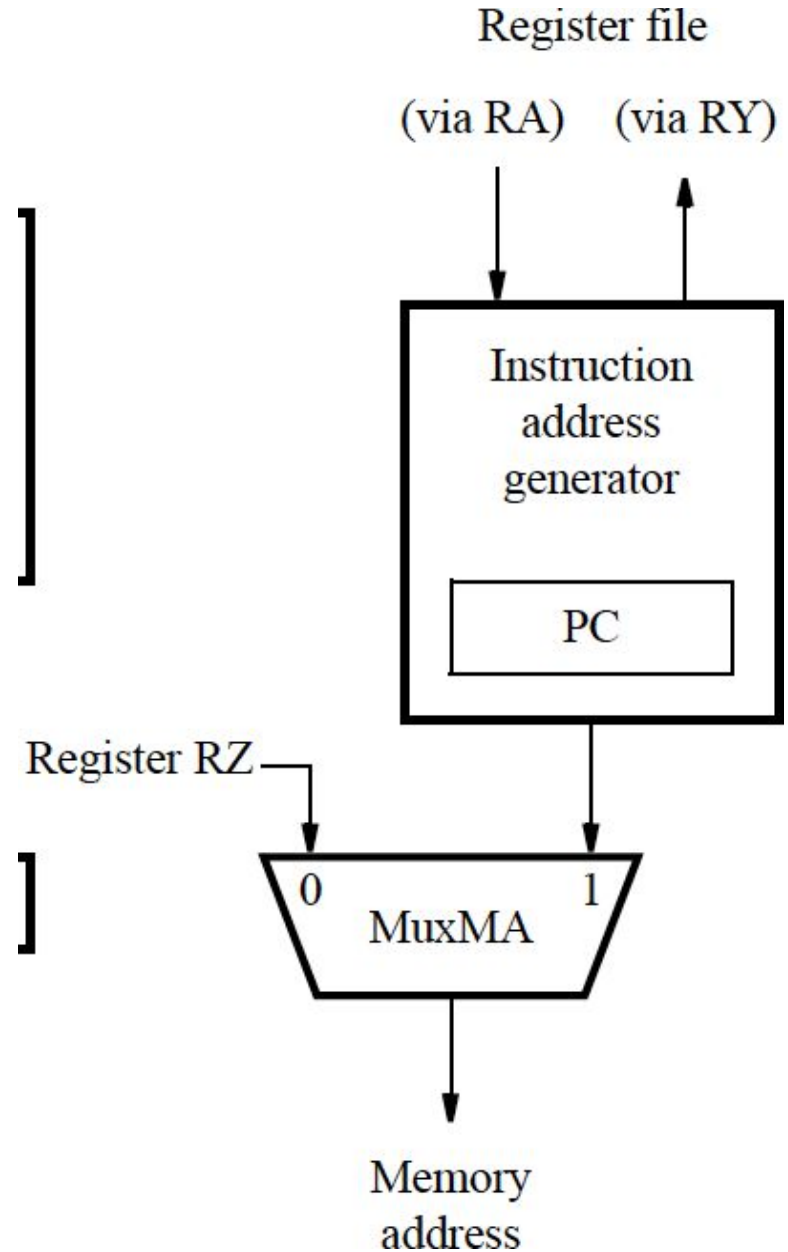
2. From register RZ in the datapath when accessing instruction operands.

□ Multiplexer MuxMA selects one of these two sources to be sent to the processor-memory interface.

□ The Instruction address generator increments the PC after fetching an instruction.

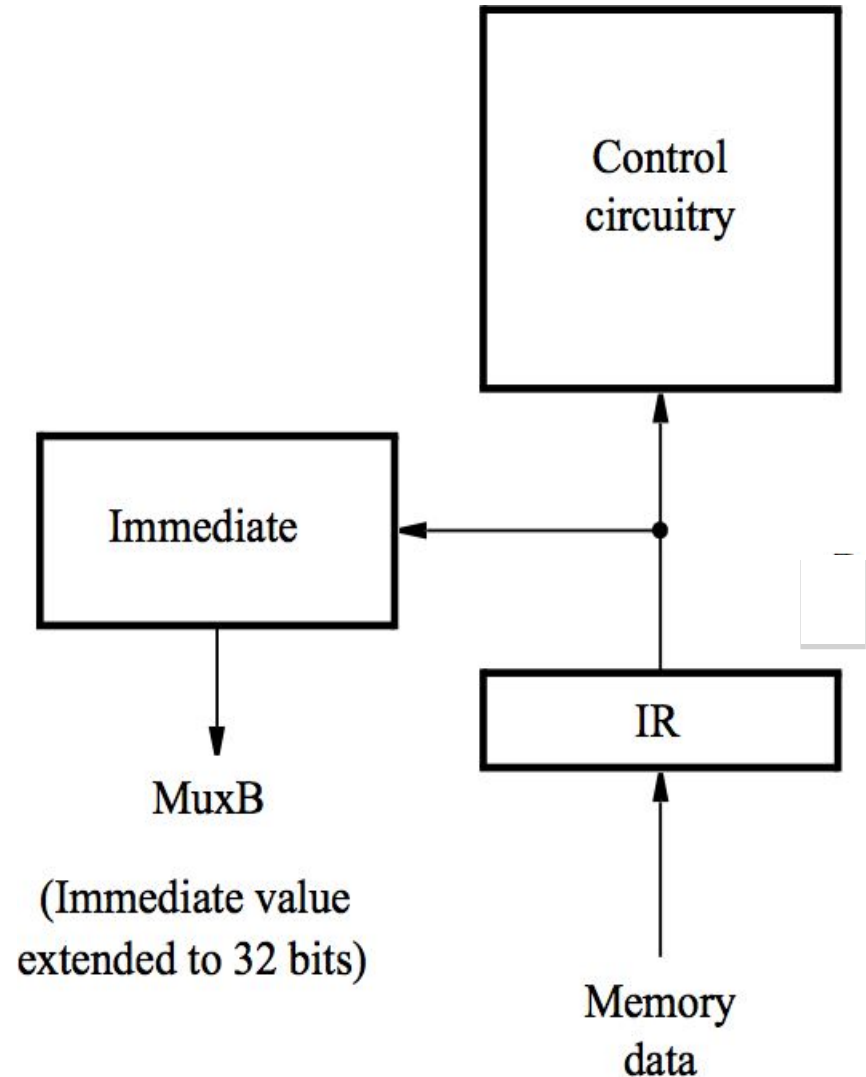
□ It also generates branch and subroutine addresses.

□ MuxMA selects RZ when reading/writing data operands.



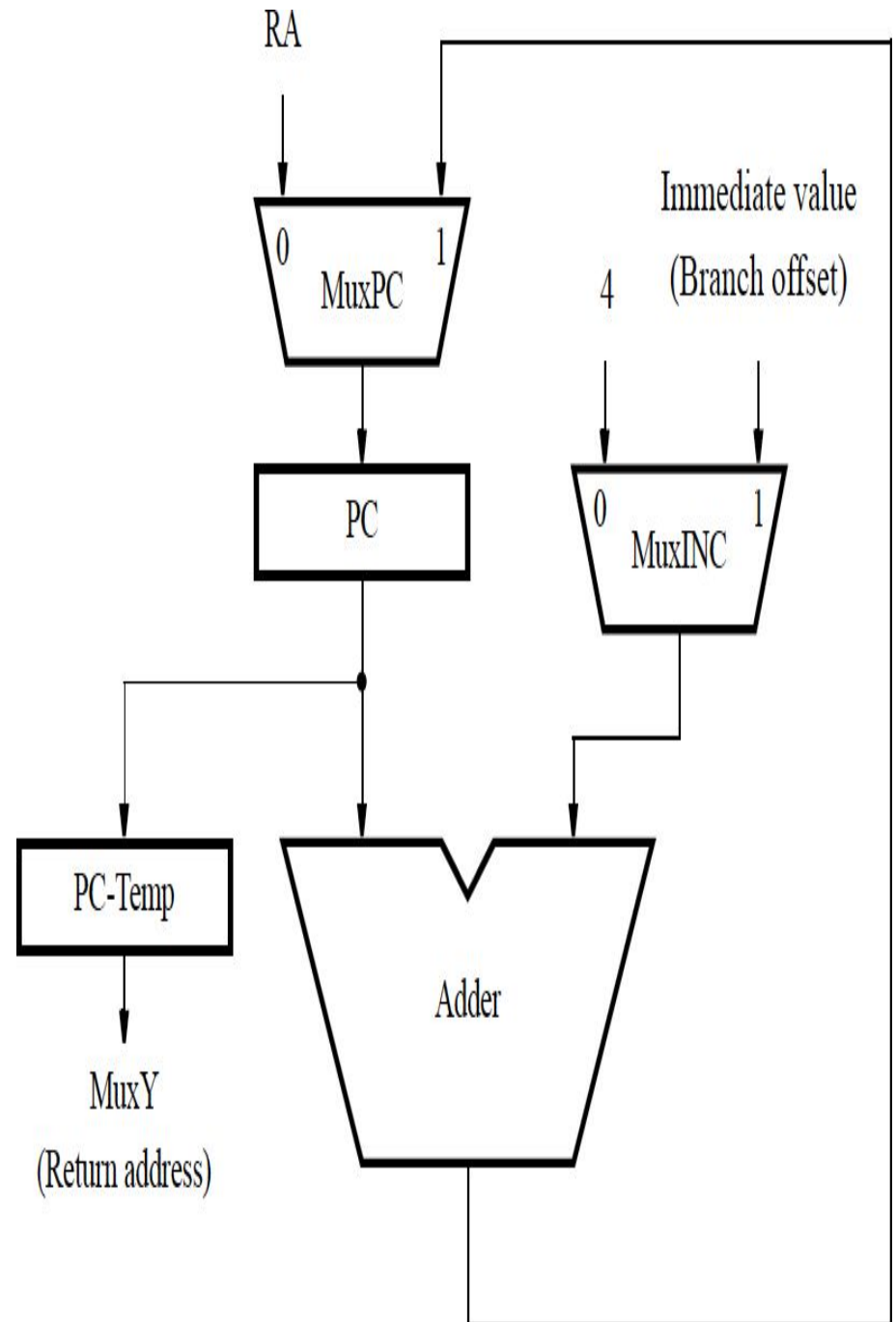
## Processor control section

- When an instruction is read, it is placed in IR.
- The control circuitry decodes the instruction.
- It generates the control signals that drive all units.
- The Immediate block gives the immediate operand value in 32 bits, according to the type of instruction.



# Instruction address generator

- Connections to registers **RA** and **RY** are used to support **subroutine call** and **return** instructions.
- An adder is used to increment the PC by 4 during **straight-line** It is also used to compute a new value to be loaded into the PC when executing **branch** and **subroutine call** instructions.
- One adder input is connected to the PC. The second input is connected to a multiplexer, MuxINC, which selects either the constant 4 or the branch offset to be added to the PC.



□ The branch offset is given in the immediate field of the IR. The output of the adder is routed to the PC via a second multiplexer, MuxPC, which selects between the adder and the output of register RA.

□ The register RA connection is needed when executing subroutine instructions. Register PC-Temp is needed to hold the content of the PC temporarily during the process of the subroutine or to give return address.

# Instruction Fetch and Execution Steps for Unconditional branch [ex: Jump]

1. Memory address  $\leftarrow$  [PC], Read memory, IR  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4
2. Decode instruction
3. PC  $\leftarrow$  [PC] + Branch offset
4. No action
5. No action

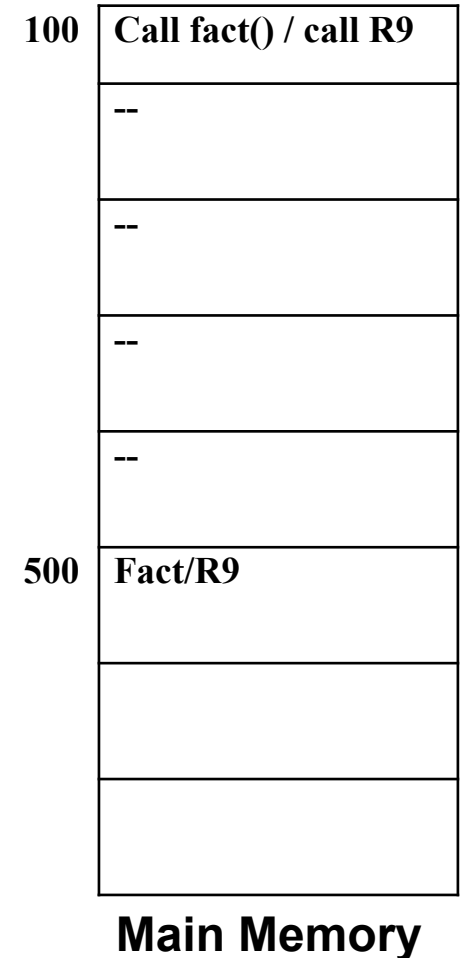
# Instruction Fetch and Execution Steps for Conditional branch: Branch\_if\_[R5]=[R6]

## LOOP

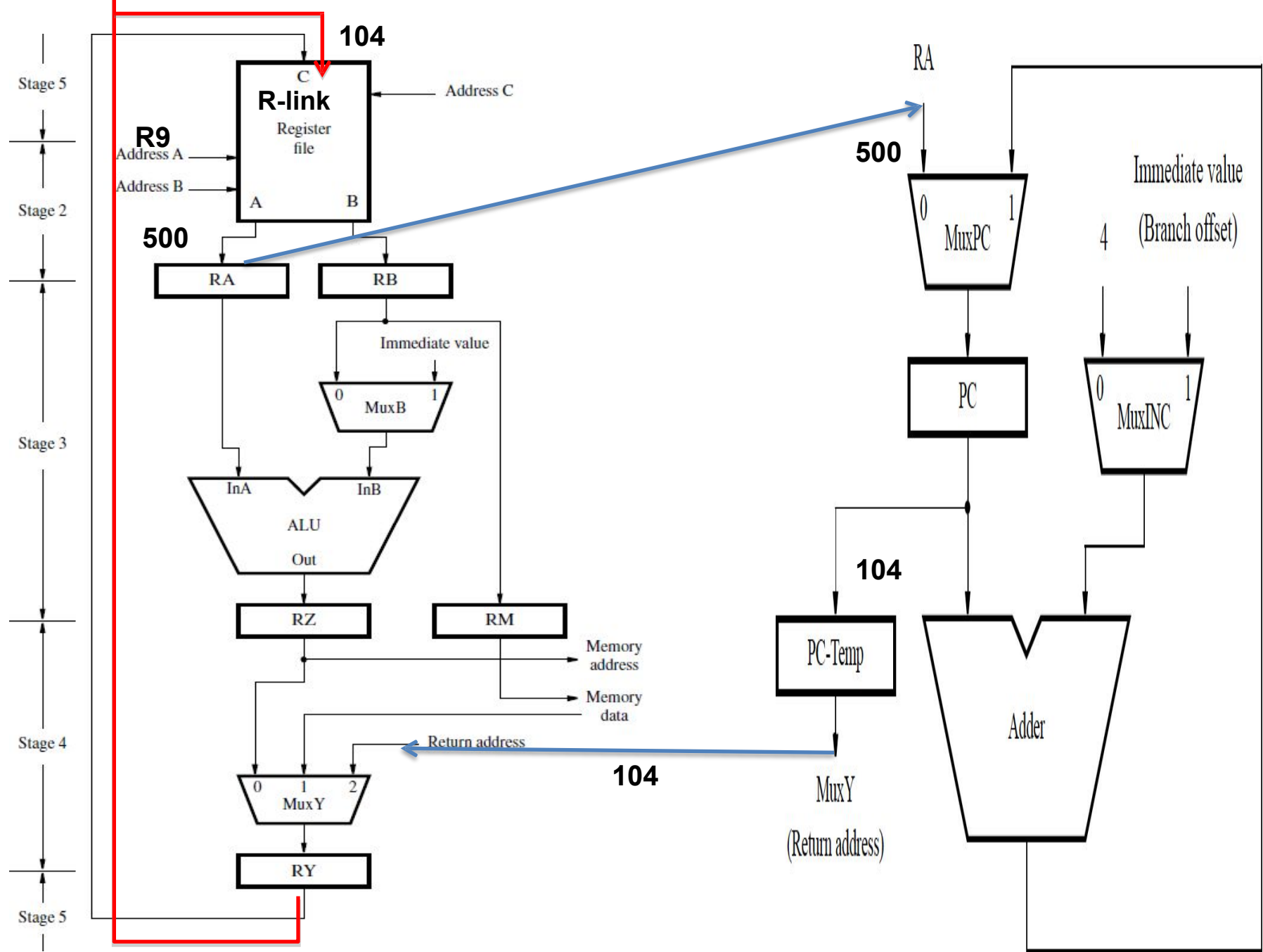
1. Memory address  $\leftarrow$  [PC], Read memory, IR  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4
2. Decode instruction, RA  $\leftarrow$  [R5], RB  $\leftarrow$  [R6]
3. Compare [RA] to [RB],  
If [RA] = [RB], then PC  $\leftarrow$  [PC] + Branch offset
4. No action
5. No action

# Instruction Fetch and Execution Steps for Subroutine call with indirection: Call\_register

1. Memory address  $\leftarrow$  [PC], **R9**  
Read memory, IR  $\leftarrow$  Memory  
data, PC  $\leftarrow$  [PC] + 4
2. Decode instruction, RA  $\leftarrow$  [R9]
3. PC-Temp  $\leftarrow$  [PC], PC  $\leftarrow$  [RA]
4. RY  $\leftarrow$  [PC-Temp]
5. Register LINK  $\leftarrow$  [RY]



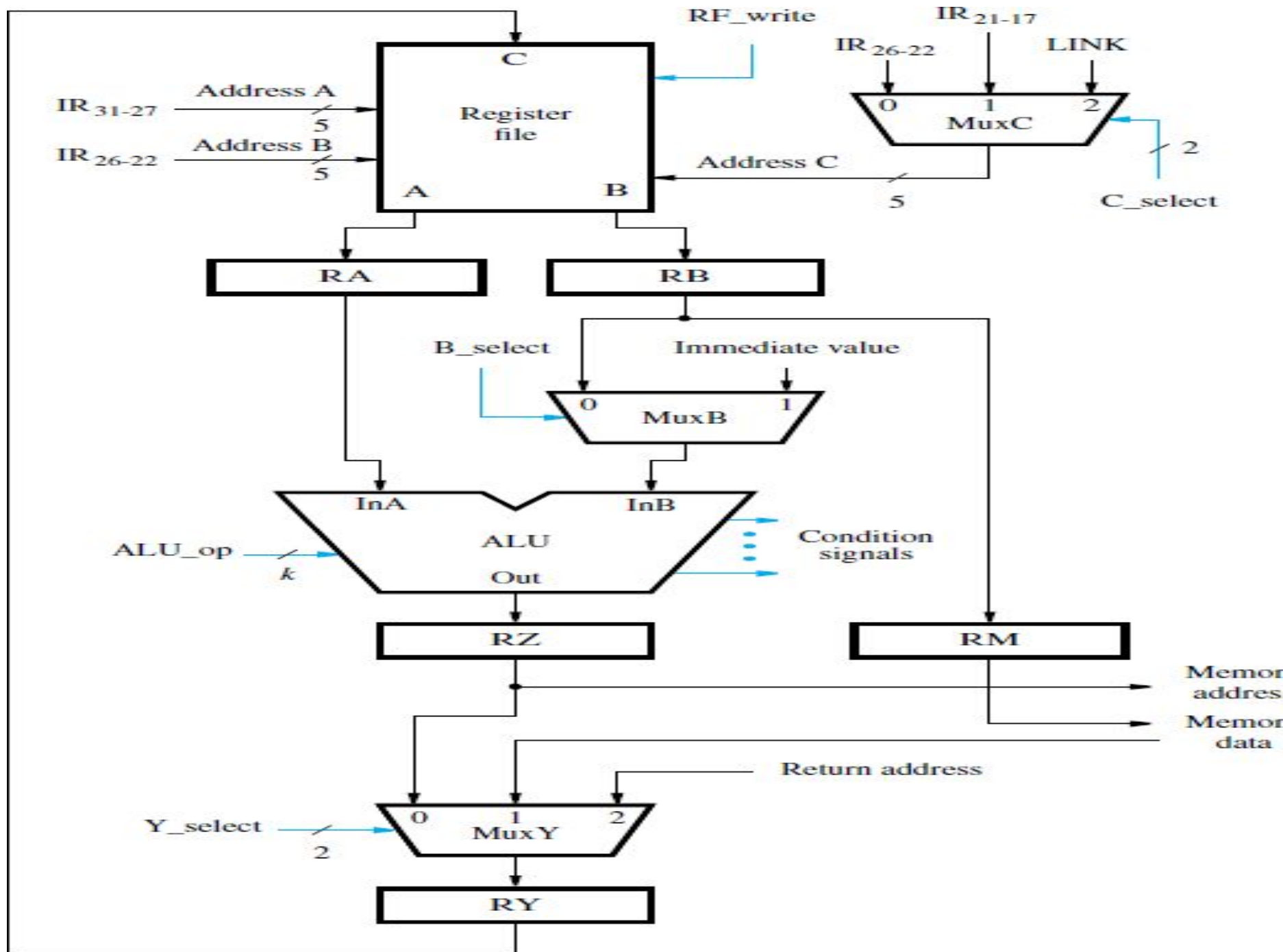
Instruction Fetch and Datapath for Subroutine call with indirection: Call\_register R9



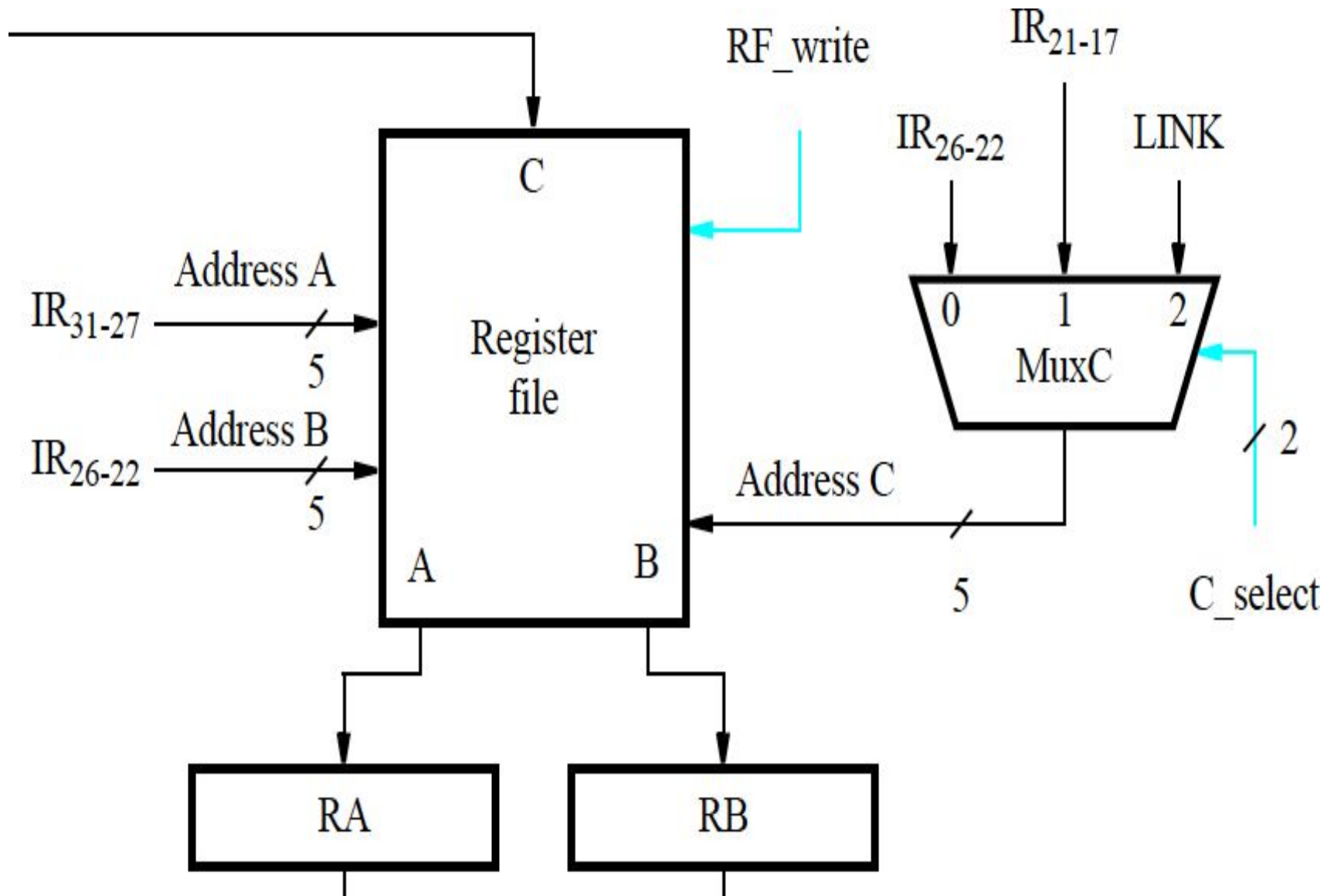


## Control signals

- The operation of the processor's hardware components is governed by *control signals*. These signals used to
- Select multiplexer inputs to guide the flow of data.
- Set the function performed by the ALU.
- Determine when data are written into the PC, the IR, the register file, and the memory.
- In each clock cycle, the results of the actions that take place in one stage are stored in *inter-stage registers*, to be available for use by the next stage in the next clock cycle. Since data are transferred from one stage to the next in every clock cycle, inter-stage registers are always enabled. This is the case for registers RA, RB, RZ, RY, RM, and PC-Temp. The contents of the other registers, namely, the PC, the IR, and the register file, must not be changed in every clock cycle.



# Register file control signals



□ The register file has **three 5-bit address inputs**, allowing access to 32 general-purpose registers. Two of these inputs, Address A and Address B, determine which registers are to be read. They are connected to fields IR31–27 and IR26–22 in the instruction register.

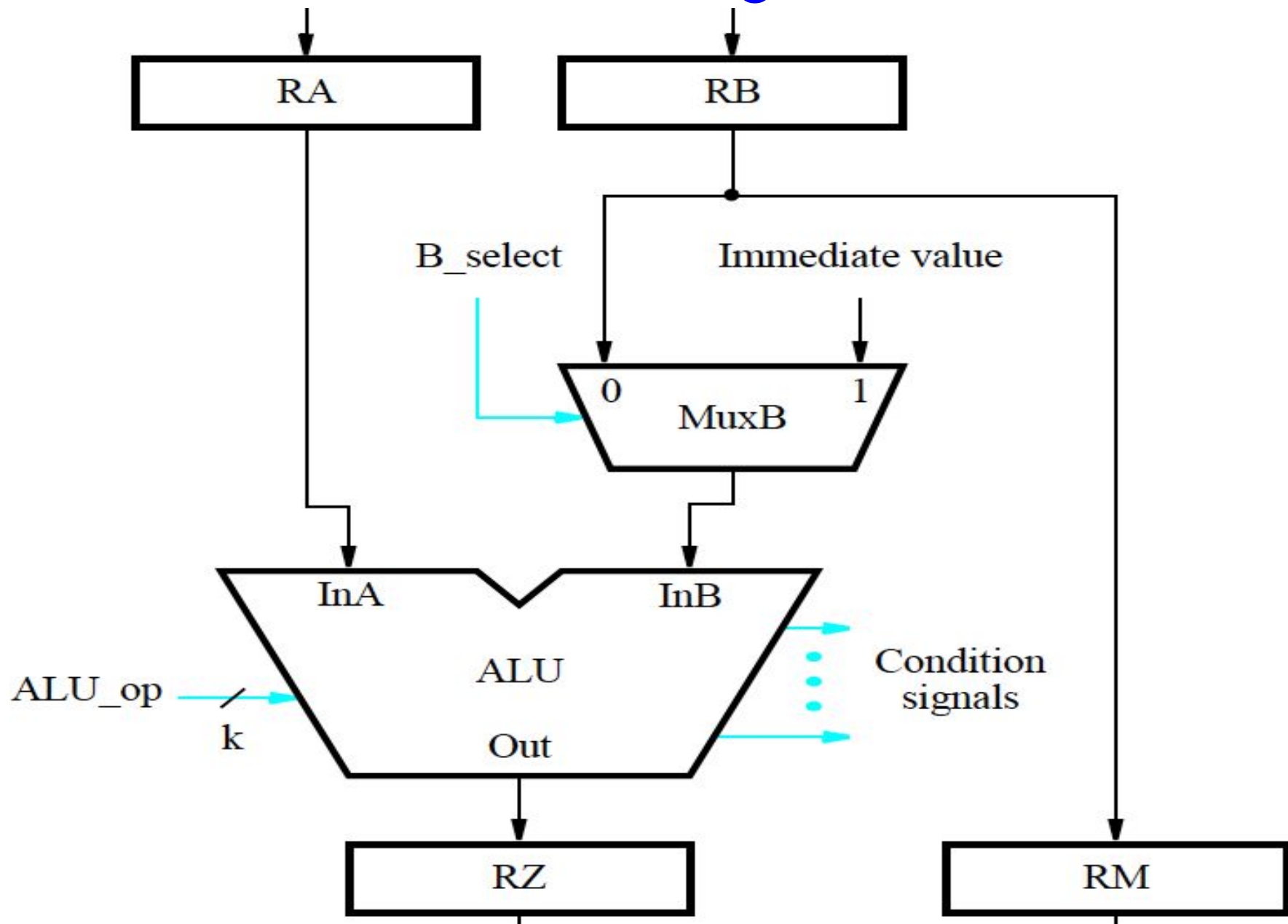
□ The third address input, Address C, selects the destination register, into which the input data at port C are to be written.

□ Multiplexer MuxC selects the source of that address. The **three-register instructions use bits IR21–17** and **other instructions[two address/one address]** use IR26–22 to specify the destination register.

□ The **third input** of the multiplexer is the address of the **link register** used in **subroutine linkage** instructions.

□ New data are loaded into the selected register only when the control signal **RF\_write** is asserted.

# ALU control signals



□ The operation performed by the ALU is determined by a *k-bit control code, ALU\_op*, which can specify up to  $2^k$  *distinct operations, such as Add, Subtract, AND, OR, and XOR*.

□ When an instruction calls for two values to be compared, a comparator performs the comparison specified. The comparator generates condition signals that indicate the result of the comparison. These signals are examined by the control circuitry during the execution of conditional branch instructions to determine whether the branch condition is true or false. **Z (zero)** : Set to 1 if the result is 0.

The other condition signals are

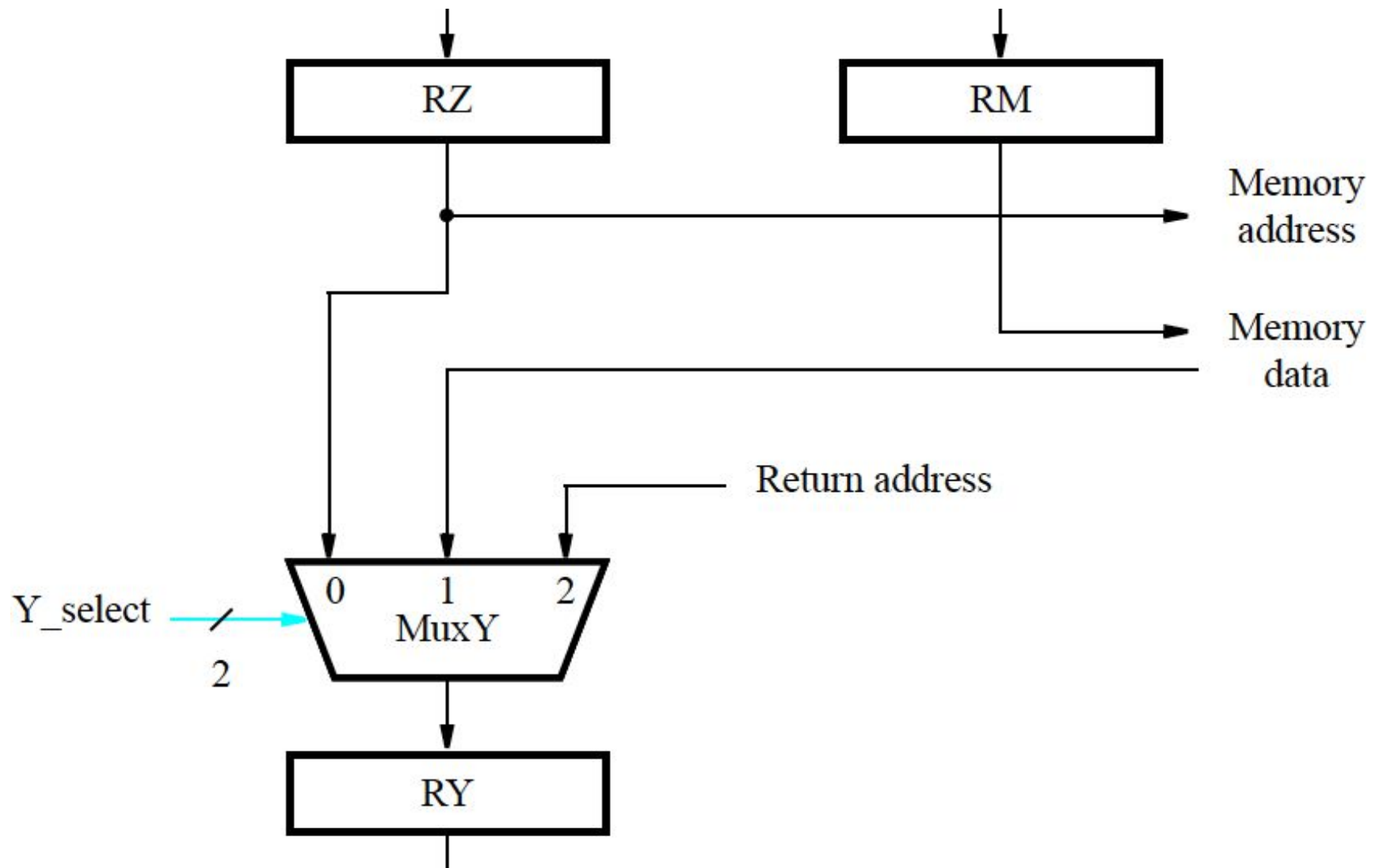
**N (negative)**: Set to 1 if the result is negative .

**V(overflow)**:Set to 1 if arithmetic overflow occurs.

**C(carry)**: Set to 1 if a carry-out results from the operation.

The **N** and **Z** flags record whether the result of an **arithmetic** or **logic** operation is negative or zero.

# Result selection

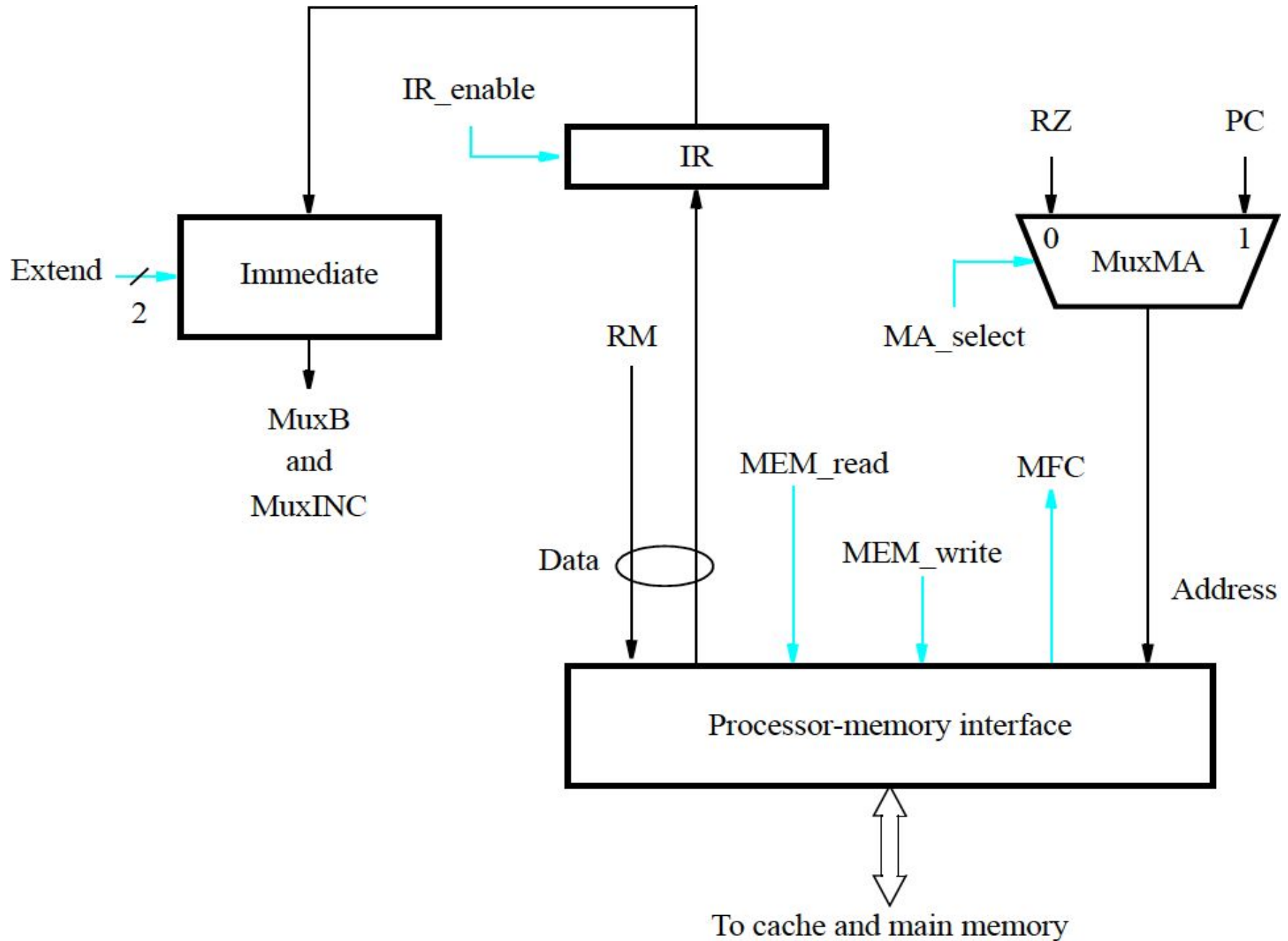


## Memory access

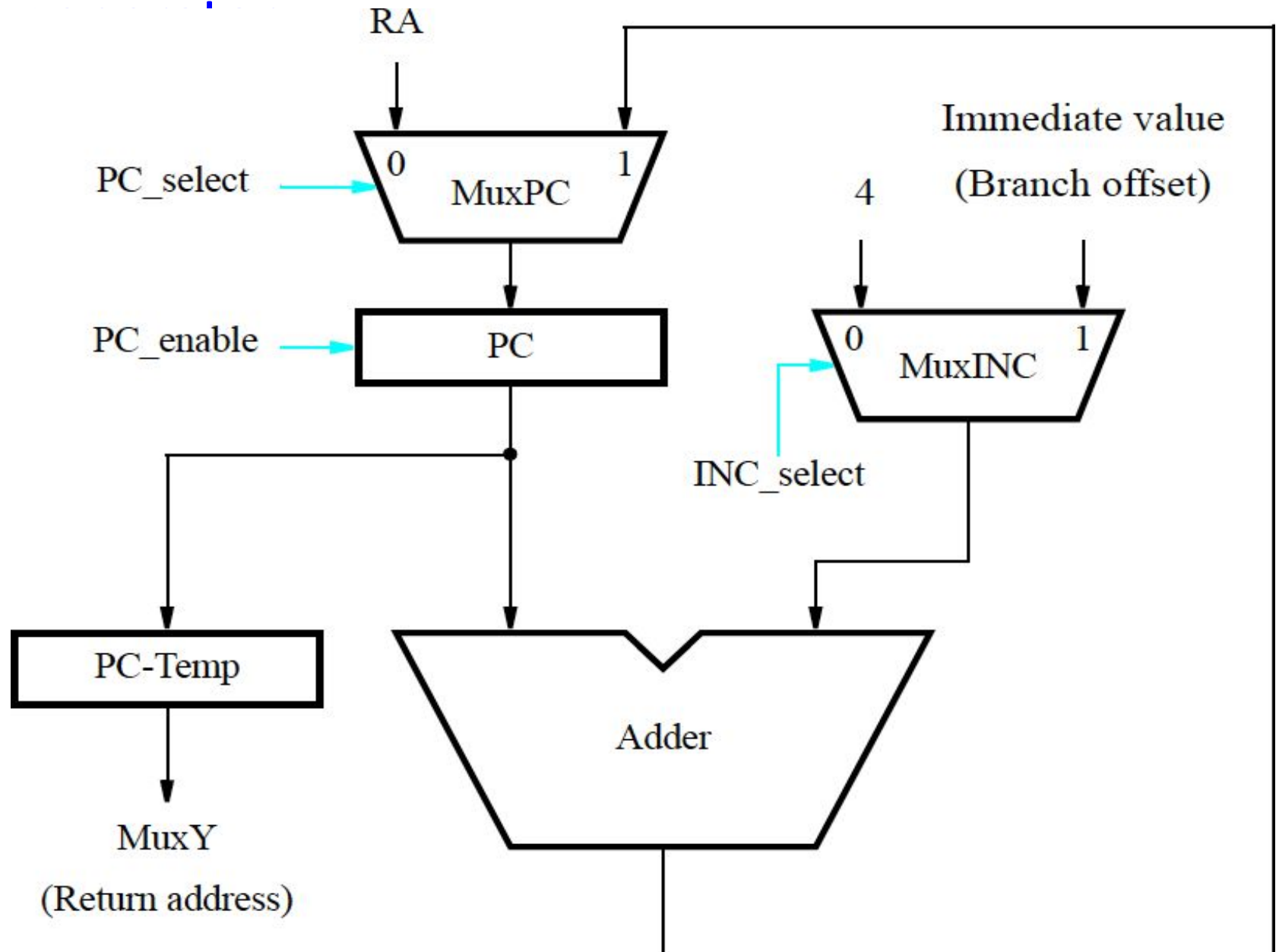
- Cache memory is faster and smaller storage that is an adjunct to the larger and slower main memory. When data are found in the cache, access to memory can be completed in one clock cycle.
- Otherwise, read and write operations may require several clock cycles to load data from main memory into the cache.
- A control signal is needed to indicate that **memory function has been completed (MFC)**. E.g., for step 1:
  1. Memory address  $\leftarrow$  [PC], Read memory, **Wait for MFC**, IR  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4
- Two signals, MEM\_read and MEM\_write are used to initiate a memory Read or a memory Write operation. When the requested operation has been completed, the interface asserts the MFC signal. The instruction register has a control signal, IR\_enable, which enables a new instruction to be loaded into the register.



# Memory and IR control signals



## Control signals of instruction address



- The INC\_select signal selects the value to be added to the PC, either the constant 4 or the branch offset specified in the instruction.
- The PC\_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC\_enable control signal is activated.

## Control signal generation

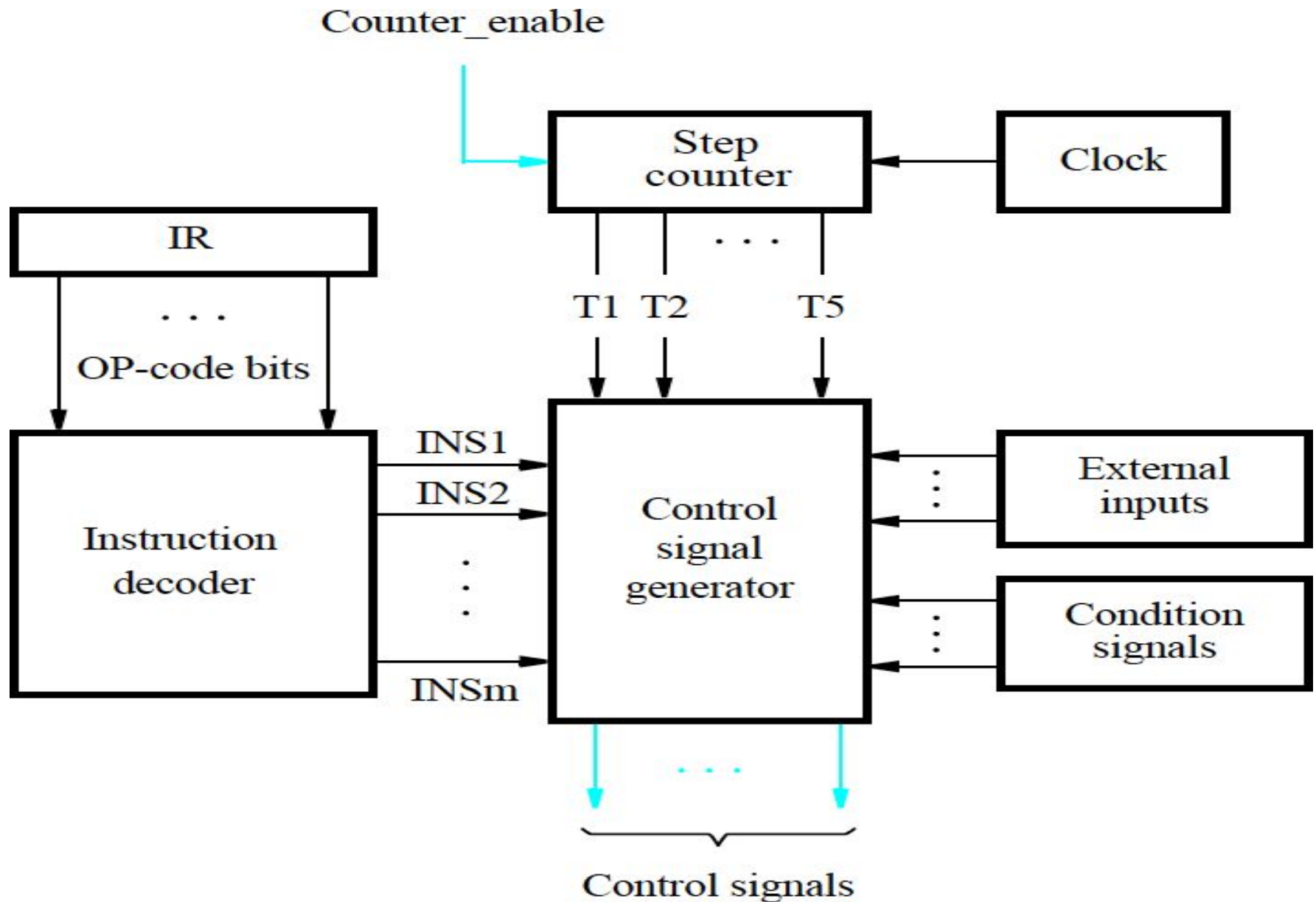
Circuitry must be implemented to generate control signals so actions take place in correct sequence and at correct time.

There are two basic approaches:

1. hardwired control [For RISC Processors]
2. Microprogramming [For CISC Processors]

**Hardwired control** involves implementing circuitry that considers step counter, IR, ALU result, and external inputs.

# Hardwired generation of control signals



**Step counter** keeps track of execution progress, one clock cycle for each of the five steps.(unless a memory access takes longer than one cycle). During each clock cycle, one of the outputs T1 to T5 of the step counter is set to 1 to indicate which of the five steps involved in fetching and executing instructions is being carried out.

The setting of the **control signals depends on:**

- Contents of the step counter
- Contents of the instruction register
- The result of a computation or a comparison operation
- External input signals, such as interrupt requests

The **instruction decoder** interprets the OP-code and addressing mode information in the IR and sets to 1 the corresponding  $INS_i$  output.

The **control signal generator** is a combinational circuit that produces the necessary control signals based on all its inputs.

## Dealing with Memory Delay

The timing signals T1 to T5 are asserted in sequence as the step counter is incremented. However, a step in which a MEM\_read or a MEM\_write command is issued does not end until the MFC signal is asserted, indicating that the requested memory operation has been completed.

To extend the duration of an execution step to more than one clock cycle, to disable the step counter.

Assume that the counter is incremented when enabled by a control signal called Counter\_enable.

Let the need to wait for a memory operation to be completed be indicated by a control signal called WMFC, which is activated during any execution step in which the Wait for MFC command is issued. Counter\_enable should be set to 1 in any step in which WMFC is not asserted. Otherwise, it should be set to 1 when MFC is asserted.

This means that  $\text{Counter\_enable} = \overline{\text{WMFC}} + \text{MFC}$

## Datapath Control Signals

The desired setting of various control signals can be determined by examining the actions taken in each execution step of every instruction. **For example**, the RF\_write signal is set to 1 in step T5 during execution of an instruction that writes data into the register file.

It may be generated by the logic expression

$$\text{RF\_write} = \text{T5} \cdot (\text{ALU} + \text{Load} + \text{Call})$$

where ALU stands for all instructions that perform arithmetic or logic operations, Load stands for all Load instructions, and Call stands for all subroutine-call and software-interrupt instructions. The RF\_write signal is a function of both the instruction and the timing signals.

## GATE Problems Datapath Control Signals

Q1. A hardwired CPU uses 10 control signals S1 to S10, in various time steps T1 to T5, to implement 4 instructions I1 to I4 as shown below: (GATE2005)

	<b>T1</b>	<b>T2</b>	<b>T3</b>	<b>T4</b>	<b>T5</b>
<b>I1</b>	S1, S3, S5	S2, S4, S6	S1, S7	S10	S3, S8
<b>I2</b>	S1, S3, S5	S8, S9, S10	S5, S6, S7	S6	S10
<b>I3</b>	S1, S3, S5	S7, S8, S10	S2, S6, S9	S10	S1, S3
<b>I4</b>	S1, S3, S5	S2, S6, S7	S5, S10	S6, S9	S10

Which of the following pairs of expressions represent the circuit for generating control signals S5 and S10 respectively? ((I<sub>j</sub>+I<sub>k</sub>)T<sub>n</sub> indicates that the control signal should be generated in time step T<sub>n</sub> if the instruction being executed is I<sub>j</sub> or I<sub>k</sub>)

- A.  $S5 = T1 + I2 \cdot T3$  and  $S10 = (I1 + I3) \cdot T4 + (I2 + I4) \cdot T5$
- B.  $S5 = T1 + (I2 + I4) \cdot T3$  and  $S10 = (I1 + I3) \cdot T4 + (I2 + I4) \cdot T5$
- C.  $S5 = T1 + (I2 + I4) \cdot T3$  and  $S10 = (I2 + I3 + I4) \cdot T2 + (I1 + I3) \cdot T4 + (I2 + I4) \cdot T5$
- D.  $S5 = T1 + (I2 + I4) \cdot T3$  and  $S10 = (I2 + I3) \cdot T2 + I4 \cdot T3 + (I1 + I3) \cdot T4 + (I2 + I4) \cdot T5$



Q2.A CPU has only three instructions I1, I2 and I3, which use the following signals in time steps T1-T5:

I1 : T1 : Ain, Bout, Cin T2 : PCout, Bin T3 : Zout, Ain T4 : Bin, Cout T5 : End

I2 : T1 : Cin, Bout, Din T2 : Aout, Bin T3 : Zout, Ain T4 : Bin, Cout T5 : End

I3 : T1 : Din, Aout T2 : Ain, Bout T3 : Zout, Ain T4 : Dout, Ain T5 : End

Which of the following logic functions will generate the hardwired control for the signal Ain ? (GATE2005)

	T1	T2	T3	T4	T5
I1	<b>Ain</b> , Bout, Cin	PCout, Bin	Zout, <b>Ain</b>	Bin, Cout	End
I2	Cin, Bout, Din	Aout, Bin	Zout, <b>Ain</b>	Bin, Cout	End
I3	Din, Aout	<b>Ain</b> , Bout	Zout, <b>Ain</b>	Dout, <b>Ain</b>	End

A.  $T1.I1 + T2.I3 + T4.I3 + T3$

B.  $(T1 + T2 + T3).I3 + T1.I1$

C.  $(T1 + T2 ).I1 + (T2 + T4).I3 + T3$

D.  $(T1 + T2 ).I2 + (T1 + T3).I1 + T3$

Q3. A processor has 40 distinct instructions and 24 general purpose registers. A 32-bit instruction word has an opcode, two register operands and an immediate operand. The number of bits available for the immediate operand field is \_\_\_\_\_.(GATE 2016)

(A) 16      (B) 8      (C) 4      (D) 32

6 bits are needed for 40 distinct instructions ( because,  $32 < 40 < 64$  )  
5 bits are needed for 24 general purpose registers( because,  $16 < 24 < 32$  )  
32-bit instruction word has an opcode(6 bit), two register operands(total 10 bits) and an immediate operand (x bits). The number of bits available for the immediate operand field  $\Rightarrow x = 32 - ( 6 + 10 )$   
 $= 16$  bits

Q4. Consider a processor with 64 registers and an instruction set of size twelve. Each instruction has five distinct fields, namely, opcode, two source register identifiers, one destination register identifier, and a twelve-bit immediate value. Each instruction must be stored in memory in a byte-aligned fashion. If a program has 100 instructions, the amount of memory (in bytes) consumed by the program text is\_\_\_\_\_.(GATE 2016)

(A) 100 (B) 200 (C) 400 (D) 500

One instruction is divided into five parts,

1) The opcode- As we have instruction set of size 16, an instruction opcode can be identified by 4 bits, as  $2^4=16$ .

2) & (3) Two source register identifiers- As there are total 64 registers, they can be identified by 6 bits. As they are two i.e. 6 bit + 6 bit.

4) One destination register identifier- Again it will be 6 bits.

5) A twelve bit immediate value- 12 bit.

Add all we get,  $4 + 6 + 6 + 6 + 12 = 34$  bit =  $34/8$  byte = 4.25 byte.

As there are 100 instructions, We have a size of 425 byte, which can be stored in 500 byte memory from the given options.

Hence (D) 500 is the answer.

Q5. A processor that has get conditional signals from carry, overflow and sign flag bits. ALU performs addition of the following two 2's complement numbers 01001101 and 11101001. After the execution of this addition operation, the status of the carry, overflow and sign flags( or conditional signals ), respectively will be: (Gate 2008)

(A) 1, 1, 0      **(B) 1, 0, 0** (C) 0, 1, 0 (D) 1, 0, 1

$$\begin{array}{r} 01001101 \\ +11101001 \\ \hline \end{array}$$

100110110  
Overflow flag is set only if the X-OR between the carry-into the sign bit and carry -out of the sign bit is 1.∥ that implies —if two binary numbers added with same sign and result has different sign then overflow possible otherwise not possible∥. Also, —if two binary numbers added with different sign then carry possible otherwise not possible∥.

Therefore,

carry flag =1, overflow flag = 0, sign bit = 0

Q6. A processor that has get conditional signals from carry, overflow and sign flag bits. Two eight bit bytes 1100 0011 and 0100 1100 are added. What are the values of the **overflow, carry and zero flags** respectively, if the arithmetic unit of the CPU uses 2's complement form? (ISRO 2013)

A. 0, 1, 1   B. 1, 1, 0   C. 1, 0, 1   **D. 0, 1, 0**

1100 0011

+ 0100 1100

10000 1111   Overflow flag is set only if the X-OR between the carry-into the sign bit and carry -out of the sign bit is 1.∥ that implies —if two binary numbers added with same sign and result has different sign then overflow possible otherwise not possible∥. Also, —if two binary numbers added with different sign then carry possible otherwise not possible∥.

Zero flag is set to be 1 if it is a logic operation.

Therefore, overflow flag = 0, carry flag =1, Zero flag= 0

Q7. In  $X = (M + N \times O) / (P \times Q)$ , how many one-address instructions are required to evaluate it (Unit I) or placed in the Instruction Register (Unit 3)? (ISRO CS 2015)

(A) 4 (B) 6 (C) 8 (D) 10

In One-address instructions, an accumulator register is required to perform all the instructions. Load and store operations are performed to fetch the values of operands from registers or memory to accumulators and to store the value of accumulator to a memory location.

Instructions required to execute the code:

$X = (M + N \times O) / (P \times Q)$

- 1) Load M :  $ACC \leftarrow M[M]$
- 2) Add N :  $ACC \leftarrow ACC +$
- 3) Mul O :  $M[N]$
- 4) Store T :  $M[T] \leftarrow ACC$
- 5) Load P :  $ACC \leftarrow M[P]$
- 6) Mul Q :  $ACC \leftarrow ACC \times M[Q]$
- 7) Div T :  $ACC \leftarrow M[T] / ACC$

Q8.The main difference(s) between a CISC and a RISC processor is/are that a RISC processor typically: (GATE 1999 )

- a) has fewer instructions
- b) has fewer addressing modes
- c) has more registers
- d) is easier to implement using hardwired control logic

(A) a and b                      (B) b and c  
(C) a and d                      **(D) a, b, c and d**

Q9.Consider the following processor design characteristics.(GATE 2018 )

- I. Register-to-register arithmetic operations only
- II. Fixed-length instruction format
- III. Hardwired control unit

Which of the characteristics above are used in the design of a RISC processor?

A I and II only **B II and III only** C I and III only  
D I, II and III

Q10. A CPU has 24-bit instructions. A program starts at address 300 (in decimal) and the program follows sequential execution.. Which one of the following is a legal program counter (all values in decimal)? (GATE 2006 )

(A) 400   (B) 500   **(C) 600**   (D) 700

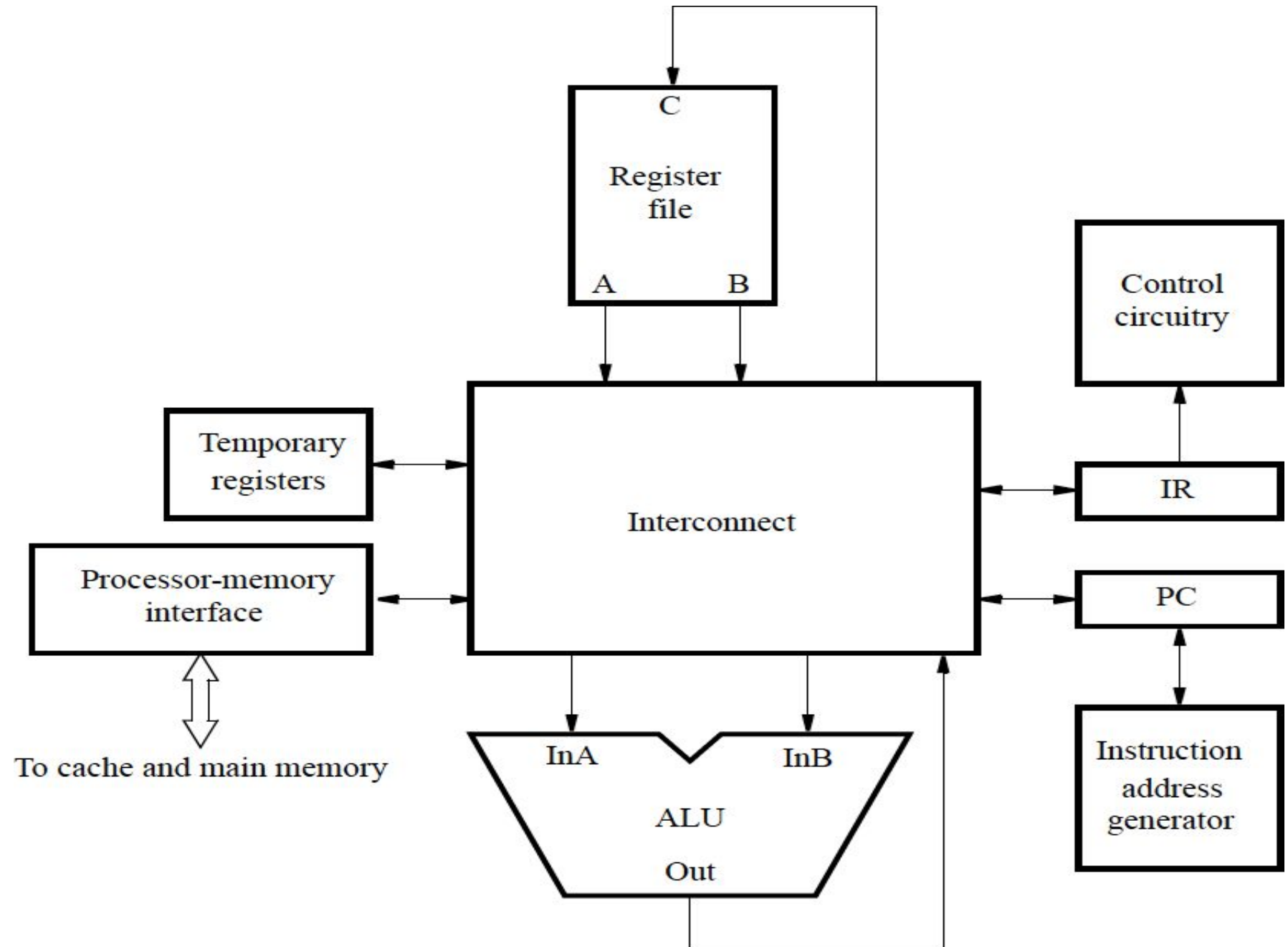
Here, size of instruction =  $24/8 = 3$  bytes. Program Counter can shift 3 bytes at a time to next instruction. So the given options must be divisible by 3. only 600 is satisfied.



## CISC processors

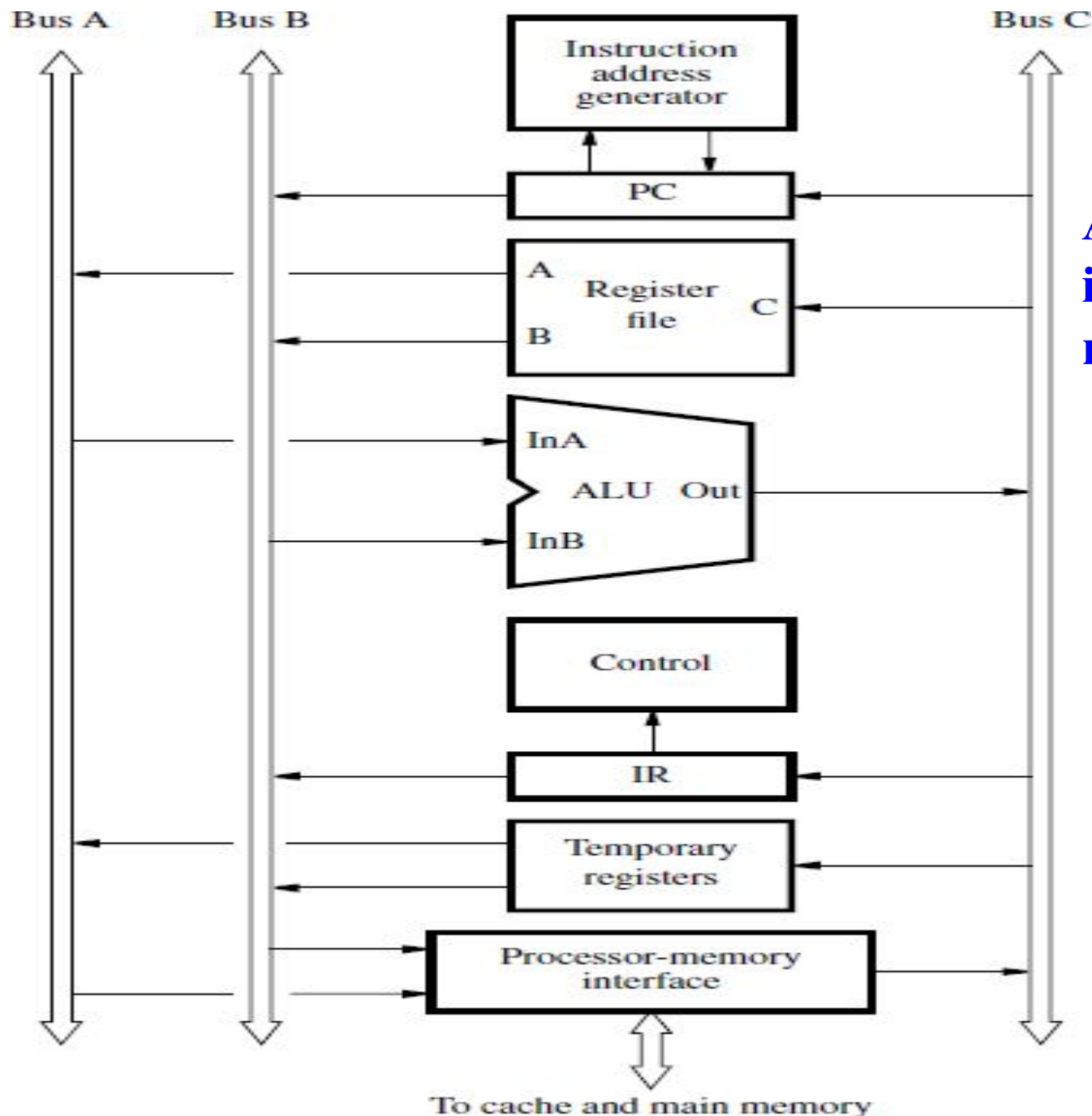
- ❑ CISC-style processors have more complex instructions. The full collection of instructions cannot be implemented in a fixed number of steps. Also, Execution steps for different instructions do not all follow a prescribed sequence of actions.
- ❑ Hardware organization should therefore enable a flexible flow of data and actions to accommodate CISC.
- ❑ **RISC-style instruction sets, where only Load and Store instructions access data in the memory, CISC instructions can operate directly on memory operands.**
- ❑ **Five-stage structure of RISC is the Interconnect block in CISC, which provides interconnections among other blocks, does not prescribe any particular structure or pattern of data flow.**
- ❑ The multi-stage structure uses inter-stage registers, such as **RZ and RY in RISC are not needed in the CISC** organization. some registers are needed to hold intermediate results during instruction execution. The **Two temporary registers Temp1 and Temp2** is provided for this purpose.

# Hardware organization for a CISC computer



# Bus

- A traditional approach to the implementation of the Interconnect is to use buses. A *bus* consists of a set of lines to which several devices may be connected, enabling data to be transferred from any one device to any other. A logic gate that sends a signal over a bus line is called a *bus driver*. Since all devices connected to the bus have the ability to send data, and ensure that only one of them is driving the bus at any given time.
- For this reason, the bus driver is a special type of logic gate called a *tri-state gate*. It has a control input that turns it on or off. When turned on, the gate places a logic signal of 0 or 1 on the bus, according to the value of its input. When turned off, the gate is electrically disconnected from the bus



**A 3-bus  
interconnection  
network**

Consider the two-operand instruction **Add R5, R6** which performs the operation  $R5 \leftarrow [R5] + [R6]$

Step	Action
1	Memory address $\leftarrow [PC]$ , Read memory, Wait for MFC, $IR \leftarrow$ Memory data, $PC \leftarrow [PC] + 4$
2	Decode instruction
3	$R5 \leftarrow [R5] + [R6]$

In step 1, **bus B** is used to send the contents of the **PC** to the **processor-memory interface**, which sends them on the **memory address lines** and initiates a **memory Read operation**. The data received from the memory, which represent an **instruction to be executed**, are sent to the **IR** over **bus C**. The command **Wait for MFC** is included to accommodate the possibility that memory access may take more than one clock cycle.

[]The instruction is decoded in step 2 and the control circuitry begins reading the source registers, R5 and R6. However, **the contents of the registers do not become available at the A and B outputs of the register file** until step 3.

[]**They are sent to the ALU using buses A and B.** The ALU performs the addition operation, and the sum is sent back to the ALU **over bus C, to be written into register R5 of register file** at the end of the clock cycle.

## Example: And X(R7),

1. Memory address  $\leftarrow$  [PC], Read memory, Wait for MFC, IR  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4
2. Decode instruction
3. Memory address  $\leftarrow$  [PC], Read memory, Wait for MFC, Temp1  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4
4. Temp2  $\leftarrow$  [Temp1] + [R7]
5. Memory address  $\leftarrow$  [Temp2], Read memory, Wait for MFC, Temp1  $\leftarrow$  Memory data
6. Temp1  $\leftarrow$  [Temp1] AND [R9]
7. Memory address  $\leftarrow$  [Temp2], Memory data  $\leftarrow$  [Temp1], Write memory, Wait for MFC

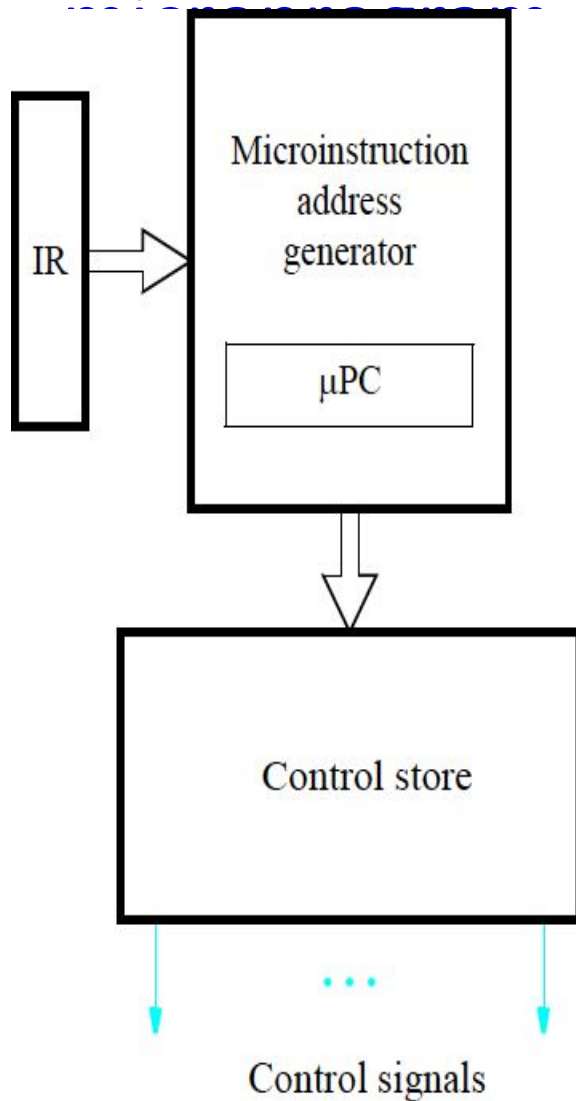
- First, the OP-code word is fetched. Then, when the instruction decoding circuit recognizes the Index addressing mode, the index offset  $X$  is fetched. Next, the memory operand is fetched and the AND operation is performed. Finally, the result is stored back into the memory.
- After decoding the instruction in step 2, the second word of the instruction is read in step 3. The data received, which represent the offset  $X$ , are stored temporarily in register Temp1, to be used in the next step for computing the effective address of the memory operand. In step 4, the contents of registers Temp1 and R7 are sent to the ALU inputs over buses A and B.
- The effective address is computed and placed into register Temp2, then used to read the operand in step 5. Register Temp1 is used again during step 5, this time to hold the data operand received from the memory. The computation is performed in step 6, and the result is placed back in register Temp1.
- In the final step, the result is sent to be stored in the memory at the operand address, which is still available in register Temp2.



# Microprogramming

- **Microprogramming** is a software-based approach for the generation of control signals.
- The values of the control signals for each clock period are stored in a **microinstruction** (control word).
- A processor instruction is implemented by a sequence of microinstructions that are placed in a **control store**.
- From decoding of an instruction in IR, the control circuitry executes the corresponding sequence of microinstructions.
- $\mu$ PC maintains the location of the current microinstruction.
- The sequence of microinstructions corresponding to a given machine instruction constitutes the microroutine that implements that instruction.

# Control signals generated from a



- Figure shows the hardware needed for microprogrammed control. It consists of a microinstruction address generator, which generates the address to be used for reading microinstructions from the control store.
- The address generator uses a *microprogram counter*,  $\mu PC$ , to keep track of control store addresses when reading microinstructions from successive locations. The microinstruction address generator decodes the instruction in the IR to obtain the starting address of the corresponding microroutine and loads that address into the  $\mu PC$ .
- As execution proceeds, the microinstruction address generator increments the  $\mu PC$  to read microinstructions from successive locations in the control store.

## Microprogramming

- Microprogramming provides the flexibility needed to implement more complex instructions in CISC processors.
- However, reading and executing microinstructions incurs undesirably long delays in high-performance processors.
- It is slower than hardwired control. Also, the flexibility it provides is not needed in RISC-style processors.
- The control signals needed to implement RISC-style instructions are quite simple to generate. Since the cost of logic circuitry is no longer a significant factor, **hardwired control has become the preferred choice.**

Ex1. At the time the instruction Load R6, 1000(R9) is fetched, R6 and R9 contain the values 4200 and 85320, respectively. Memory location 86320 contains 75900. Show the contents of the inter stage registers during each of the 5 execution steps of this instruction.

Step	RA	RB	RZ	RM	RY
1	--	--	--	--	--
2	--	--	--	--	--
3	85320	4200	--	--	--
4	85320	4200	86320	4200	--
5	85320	4200	86320	4200	75900

Ex2. At some point in the execution of a program, registers R4, R6, and R7 contain the values 1000, 7500, and 2500, respectively. Show the contents of registers RA, RB, RZ, RY, and R6 during steps 3 to 5 as the instruction **Subtract R6, R4, R7** is fetched and executed, and also during step 1 of the instruction that is fetched next.

Step	RA	RB	RZ	RM	R6
3	1000	2500	--	--	7500
4	1000	2500	-1500	--	7500
5	1000	2500	-1500	-1500	7500
1	1000	2500	-1500	-1500	-1500

## Foreign University Problems( University of North Texas)

1. Consider the following instruction executed on the RISC processor datapath. **Load R4, 40(R5)**

- a) Write down the register transfer notation (RTN) expression for the above instruction.
- b) At the time, the above instruction is fetched, R4 and R5 contain the values 4200 and 85320, respectively. Memory location 85360 contains 75900. Referring to the processor datapath, show the following values during the execution of this instruction: (i) Address input A of register file during stage 2, (ii) Contents of RA and the input selection for multiplexer MuxB during stage 3, (iii) Contents of RZ and the input selection for MuxY during stage 4, (iv) Contents of RY and the address input C of register file during stage 5.

**Answer:**

(a)  $R4 \leftarrow [R5] + 40$

(b) Stage 2: This instruction needs to read R5 for memory address

calculation. Therefore address input A of register file = R5

Stage 3:  $RA = \text{contents of } R5 = 85320$  MuxB selects input 1 (immediate value 40)

Stage 4:  $RZ = 85320 + 40 = 85360$  MuxY selects the return data from memory (input 1)

Stage 5:  $RY = \text{contents of memory location } 85360 = 75900$ . The instruction needs to load data into register R4. Therefore, address input C of register file = R4

2. Consider the following instruction executed on the RISC processor datapath Subtract R5, R4, R7.

) Write down the register transfer notation (RTN) expression for the above instruction.

) At the time, the above instruction is fetched; R4 and R7 contain the values 530 and 360, respectively. Referring to the processor datapath, show the following values during the execution of this instruction:

- (i) Address inputs A and B of register file during stage 2
- ii) Contents of RA, RB and the input selection for multiplexer MuxB during stage 3

(a)  $R5 \leftarrow [R4] - [R7]$

(b) Stage 2: This instruction needs to read R4 and R7 for the subtraction operation. Address input A of register file = R4 Address input B of register file = R7

Stage 3: RA = contents of R4 = 530 RB = contents of R7 = 360 MuxB selects input 0

Stage 4: RZ =  $530 - 360 = 170$  MuxY selects the input 0 (contents of RZ)

Stage 5: RY = 170 The instruction needs to write the result of subtraction operation into register R5. Therefore, address input C of register file = R5.

**Q3. Consider the following instruction executed on the RISC processor datapath      Store R8, 60(R2)**

(a) Write down the register transfer notation (RTN) expression for the above instruction.

(b) At the time, the above instruction is fetched; R2 and R8 contain the values 72400 and 54700, respectively. Referring to the processor datapath, show the following values during the execution of this instruction:

(i) Address inputs A and B of register file during stage 2

(ii) Contents of RA, RB and the input selection for multiplexer MuxB during stage 3

(iii) Contents of RZ and RM during stage 4



(a)  $[R2] + 60 \rightarrow [R8]$

(b) Stage 2: This instruction needs to read R2 for address calculation and R8 for data to be written to memory Address input A of register file = R2 Address input B of register file = R8

Stage 3: RA = contents of R2 = 72400 RB = contents of R7 = 54700

MuxB selects input 1 (immediate value 60)

Stage 4:  $RZ = 72400 + 60 = 72460$  RM = 54700

Q4. Consider the following sequence of instructions being processed on the 5-stage RISC processor .

Load R5, #200(R3)

Add R2, R5, R6

Call\_Register R2

Write down the values of following control signals for each of the three instructions:

- a) PC\_enable during stage 3 of instruction processing
- b) Mem\_read and Y\_select during stage 4 of instruction processing
- c) RF\_write and C\_select during stage 5 of instruction processing

- a) For Load and Add instructions,  $PC\_enable = 0$   
For Call\_Register instruction,  $PC\_enable = 1$
- b) For Load instruction,  $Mem\_read = 1$ ,  $Y\_select = 01$   
(1) For Add instruction,  $Mem\_read = 0$ ,  $Y\_select = 00(0)$   
For Call\_register instruction,  $Mem\_read = 0$ ,  $Y\_select = 10(2)$
- c) For Load instruction,  $RF\_write = 1$ ,  $C\_select = 00$   
(0) For Add instruction,  $RF\_write = 1$ ,  $C\_select = 01$   
(1)  
For Call\_register instruction,  $RF\_write = 1$ ,  $C\_select = 10(2)$

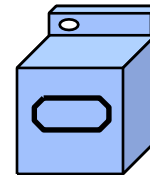
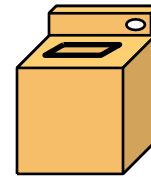
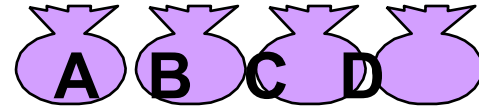
# Pipelining

# Outline

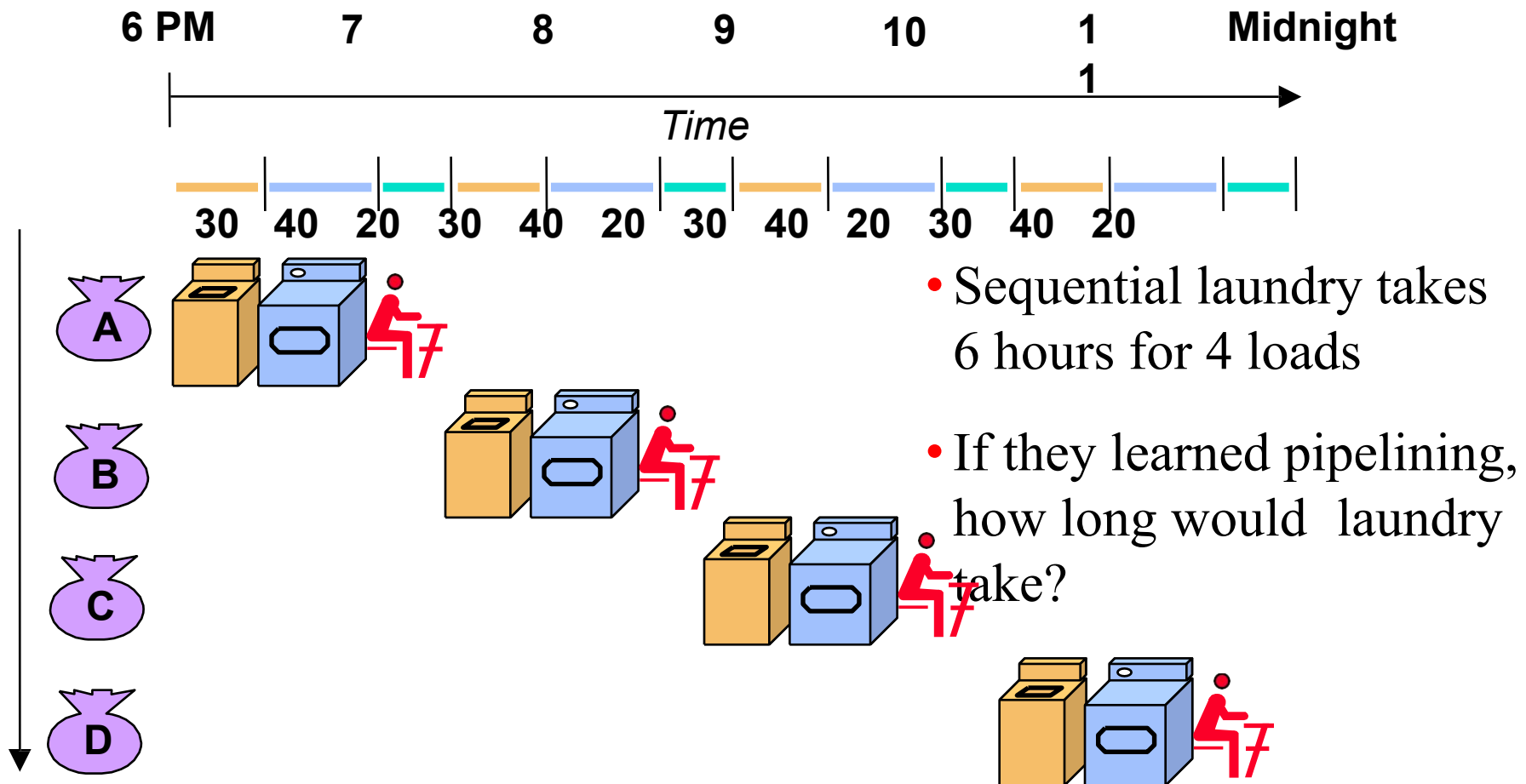
- Pipelining: overlapped instruction execution
- Hazards(Risk or danger) that limit pipelined performance gain
- Hardware/software implications of pipelining
- Influence of pipelining on instruction sets
- Pipelining in superscalar processors

# Difference between Sequential and Pipeline Concept

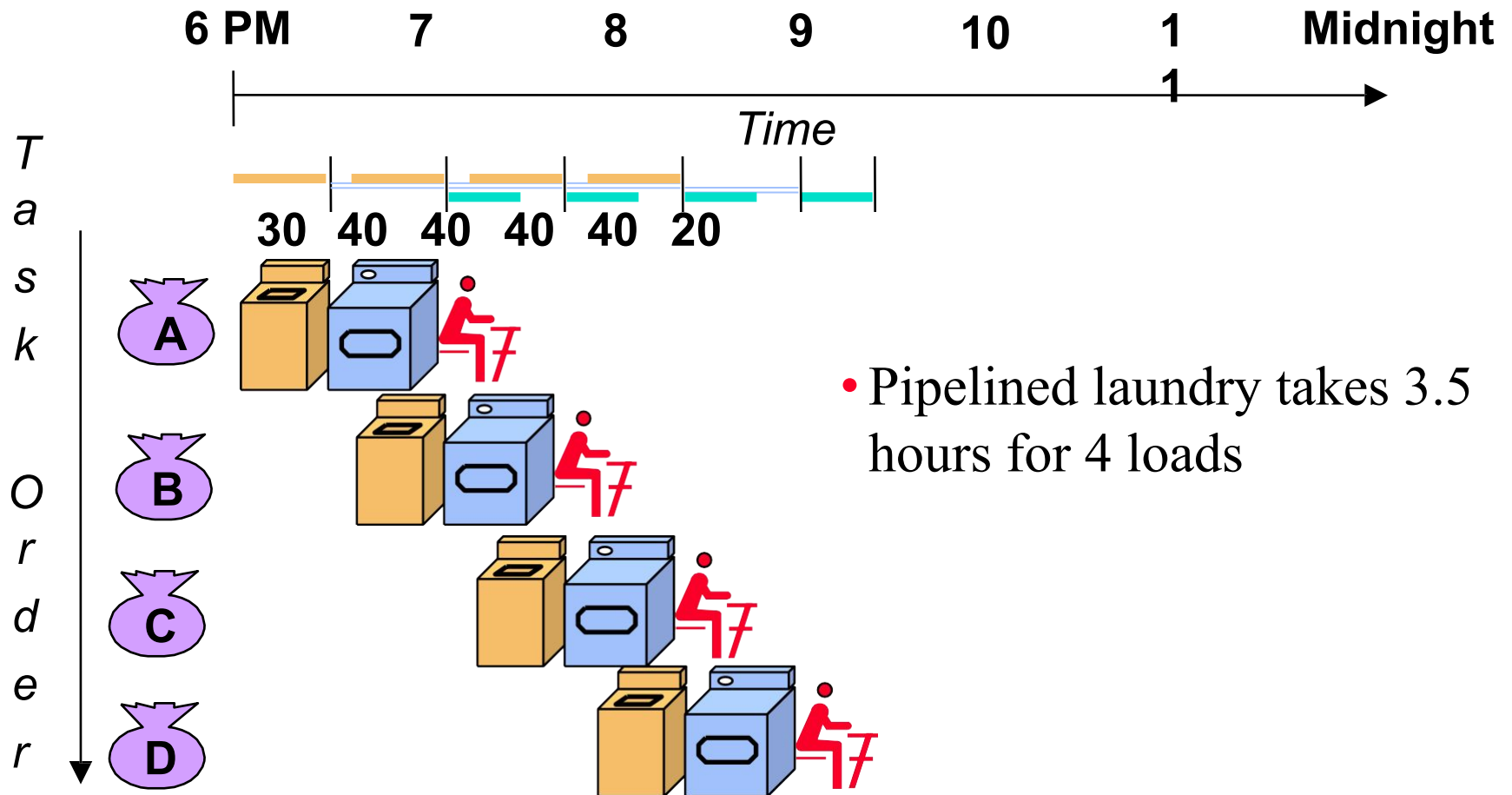
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- —Folder takes 20 minutes



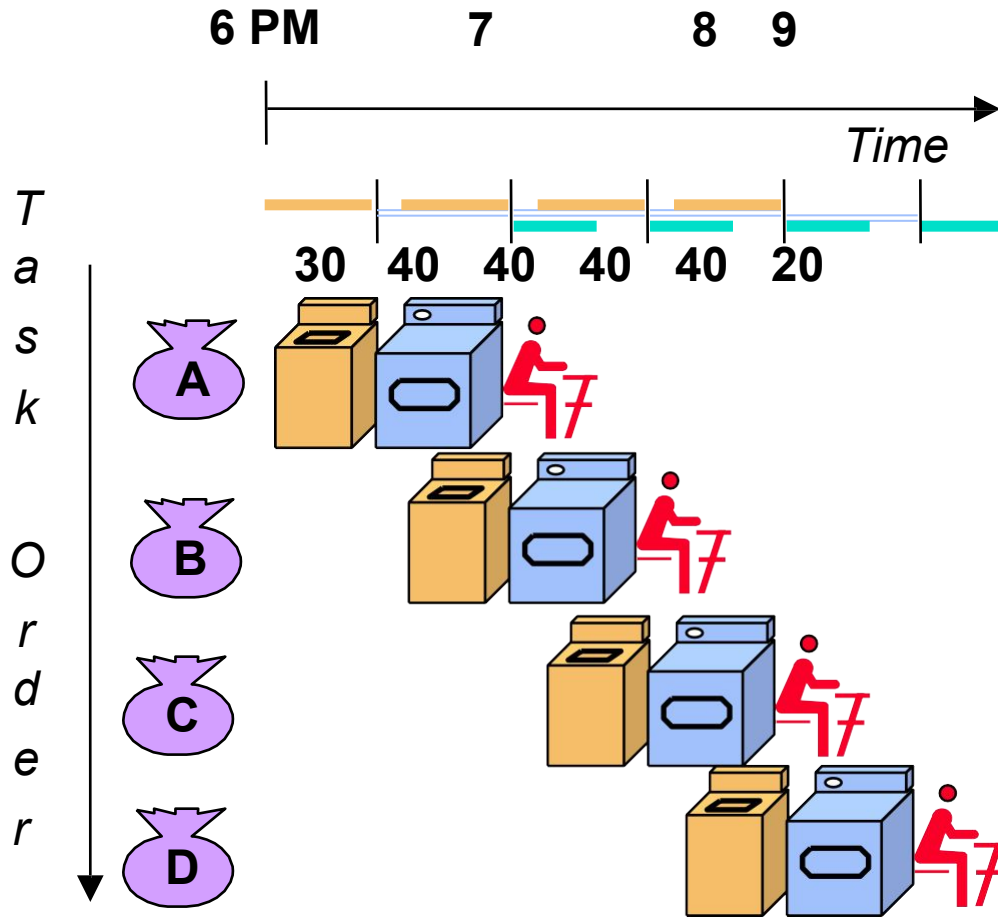
# Difference between Sequential and Pipeline Concept



# Traditional Pipeline Concept



# Traditional Pipeline Concept



- 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 using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to —fill pipeline
- Stall(stop making



## Basic Concept of Pipelining

- Circuit technology and hardware arrangement influence the speed of execution for programs
- All computer units benefit from faster circuits
- Pipelining involves arranging the hardware to *perform multiple operations simultaneously*
- Same total time for each item, but *overlapped*
- Focus on pipelining of *instruction execution* , multistage datapath consists of: Fetch, Decode, Compute, Memory, Write
- Instructions fetched & executed one at a time with only one stage active in any cycle
- *With pipelining*, multiple stages are active simultaneously for different instructions

Clock cycle            1            2            3            4            5            6            7

$I_j$



$I_{j+1}$



$I_{j+2}$



## Pipeline Organization

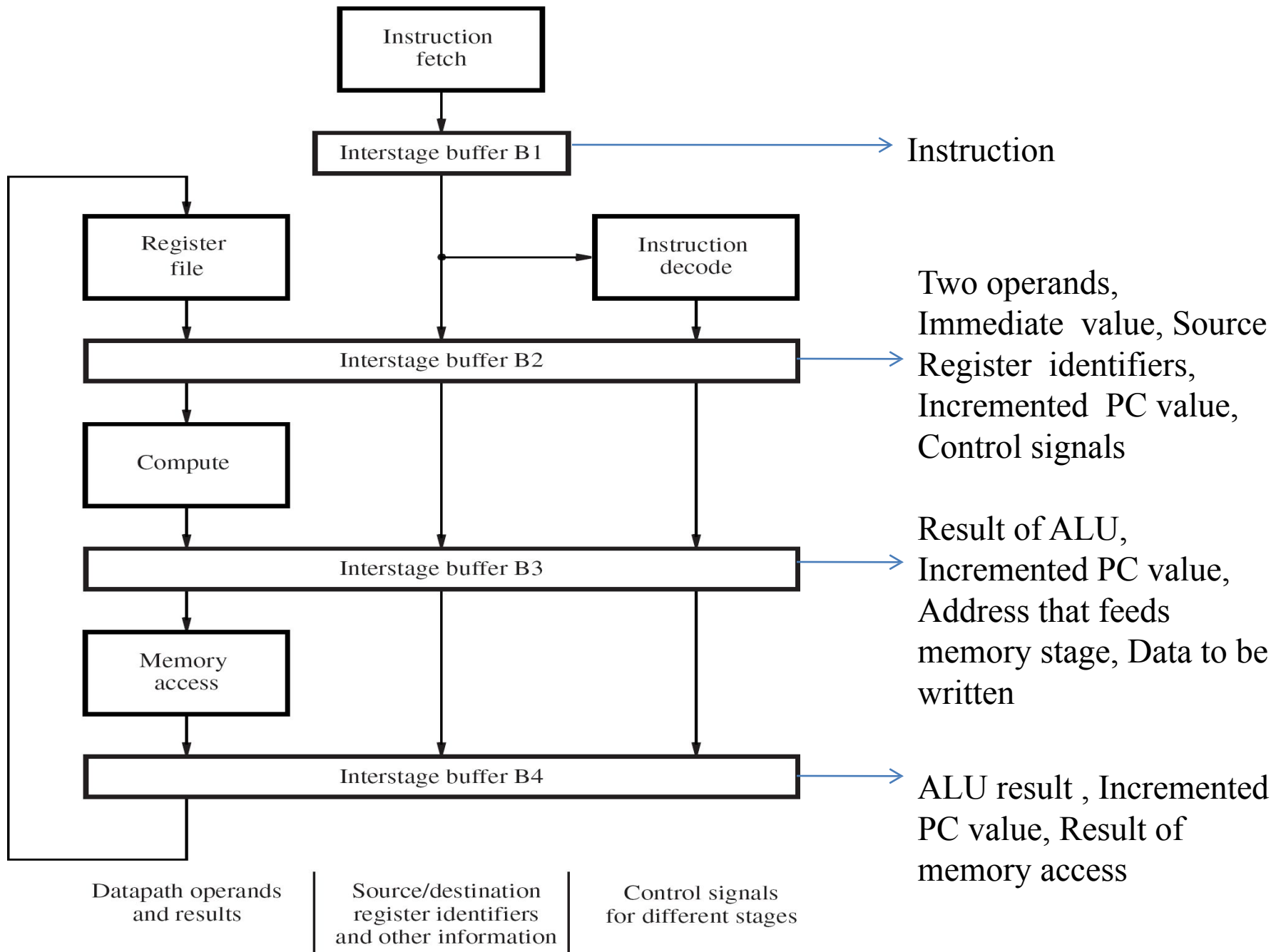
- Use program counter (PC) to fetch instructions and every cycle a new instruction enters in pipeline
- Carry along instruction-specific information as instructions flow through the different stages. Use *interstage buffers* to hold this information. These buffers incorporate RA, RB, RM, RY, RZ, IR, and PC-Temp registers

### The interstage buffers are used as follows

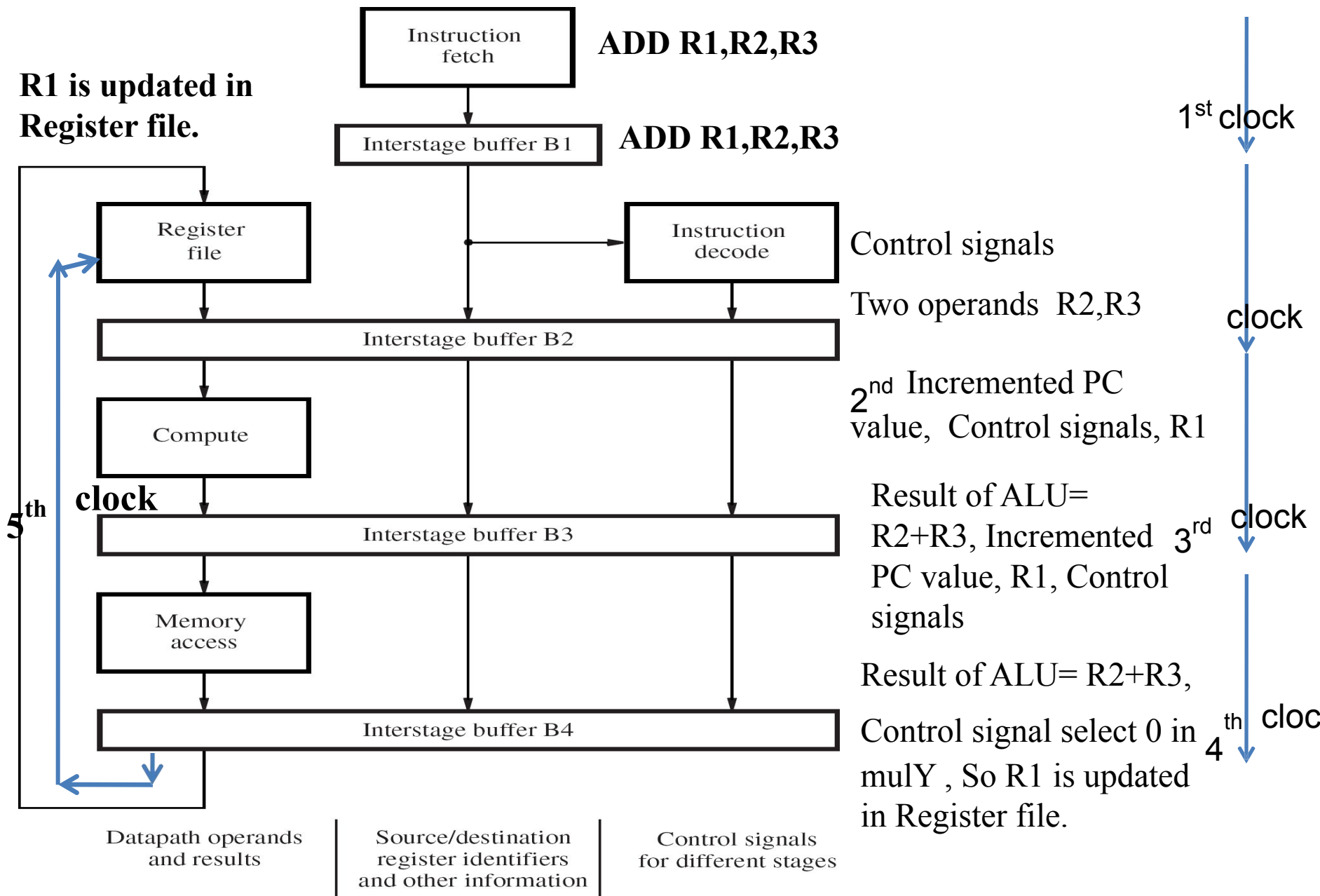
- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
- Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder. The settings for control signals move through the pipeline to determine the ALU operation, the

- **Interstage buffer B3** holds the **result of the ALU** operation, which may be data to be written into the register file or an address that feeds the Memory stage. In the case of a write access to memory, buffer B3 holds the **data to be written**. These data were read from the register file in the Decode stage. The buffer also holds the **incremented PC value** passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction.
- **Interstage buffer B4** feeds the Write stage with a value to be written into the register file. This value may be the **ALU result** from the Compute stage, the **result of the Memory access stage**, or the **incremented PC value** that is used as the return address for a subroutine-call instruction.

# A five-stage pipeline

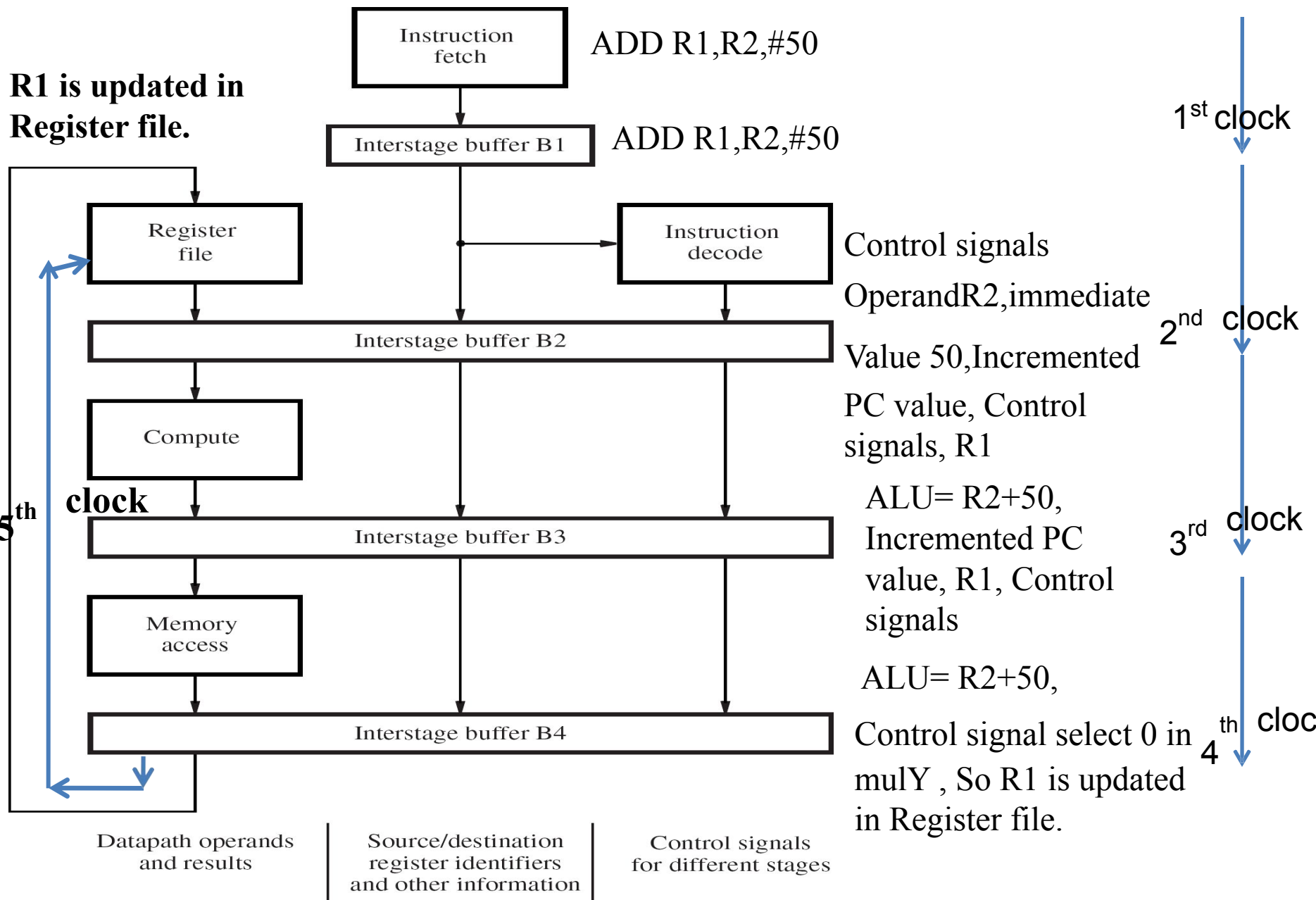


# A five-stage pipeline to execute the instruction ADD R1,R2,R3



# A five-stage pipeline to execute the instruction ADD R1,R2,#50

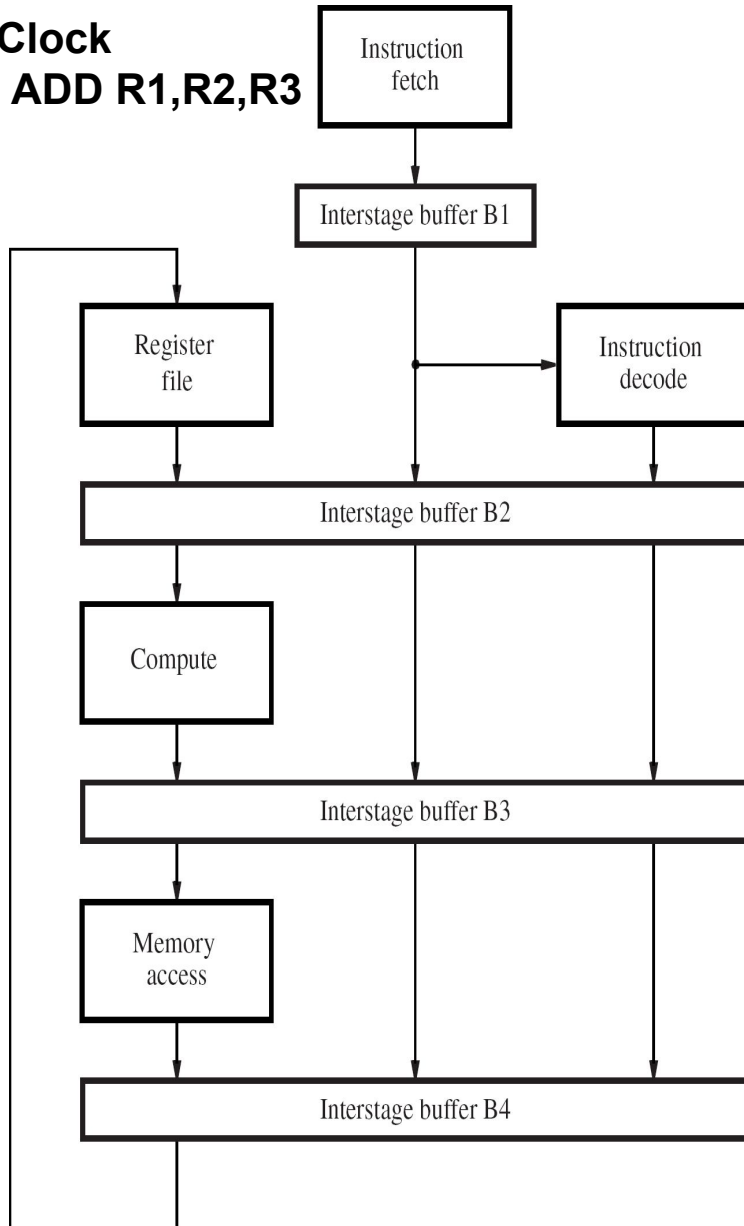
**R1 is updated in Register file.**



# Stages of 5 instructions

**1<sup>st</sup> Clock**

**I1: ADD R1,R2,R3**



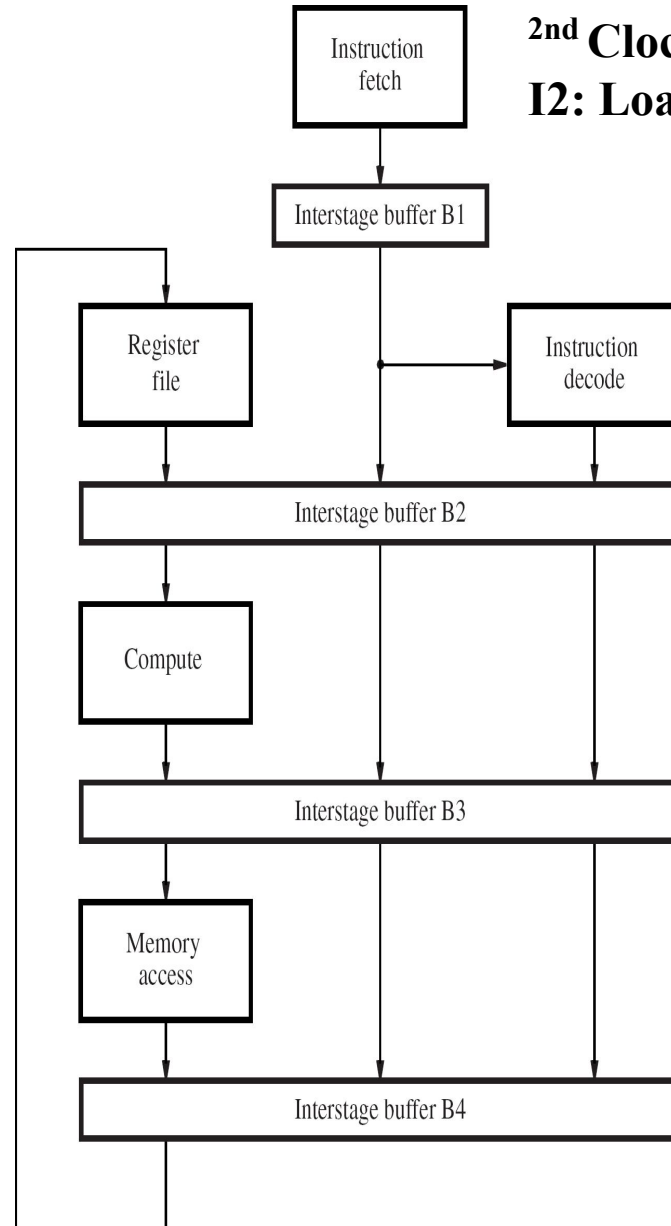
Datapath operands  
and results

Source/destination  
register identifiers  
and other information

Control signals  
for different stages

**2<sup>nd</sup> Clock**

**I2: Load R4,R5,R6**



**2<sup>nd</sup> Clock**

**I1: ADD  
R1,R2,R3**

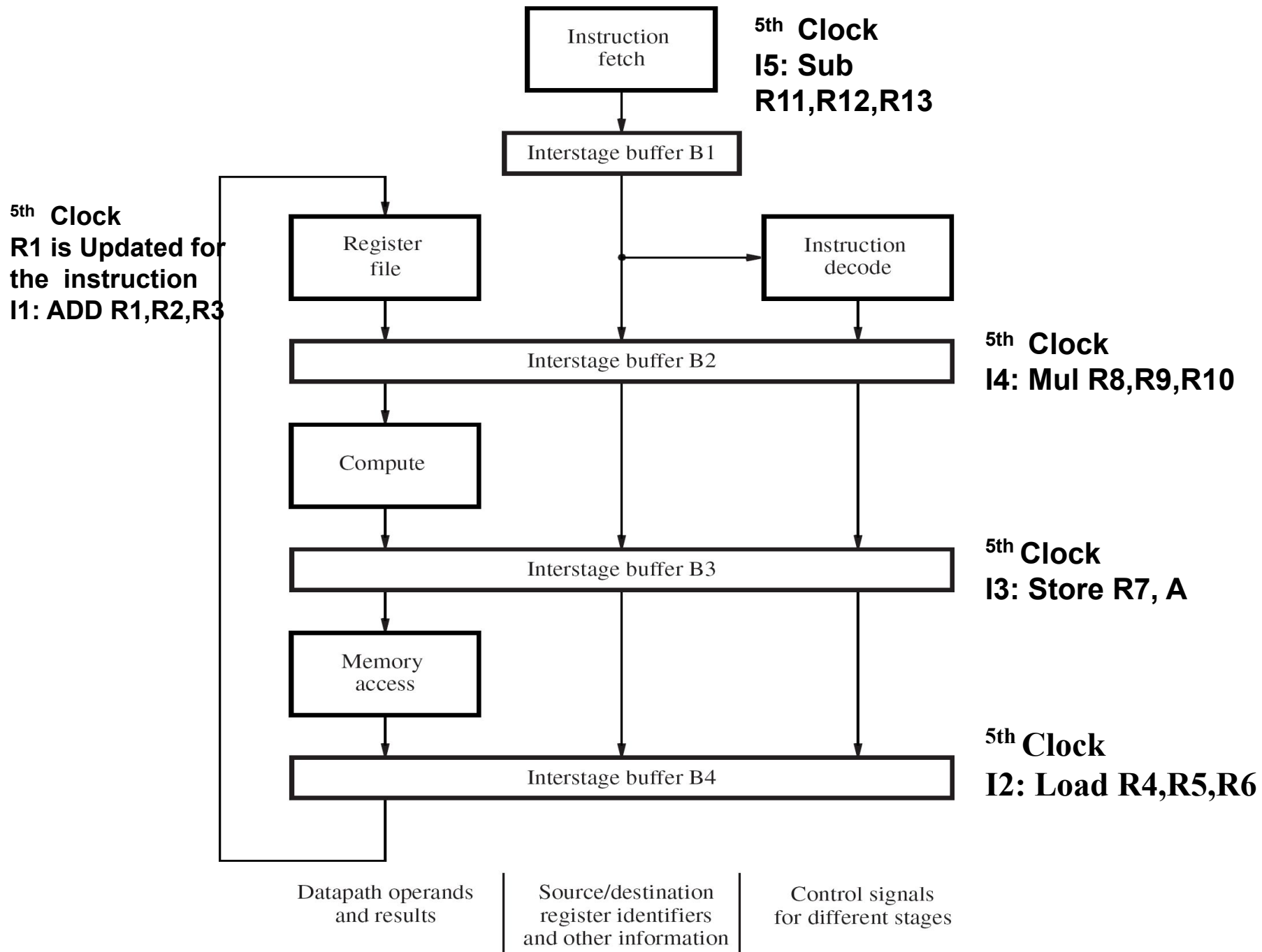
Datapath operands  
and results

Source/destination  
register identifiers  
and other information

Control signals  
for different stages







# Hazard and types of Hazard

Any condition that causes a pipeline to stall is called a hazard.

1. **Data hazard** – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
2. **Instruction (control) hazard** – a delay in the availability of an instruction causes the pipeline to stall.
3. **Structural hazard** – the situation when two instructions require the use of a given hardware resource at the same time.[Ex: Memory,ALU]

# Pipelining Issues

1. Data dependency
2. Memory Delays with Cache Miss and Cache Hit
3. Branch Delay

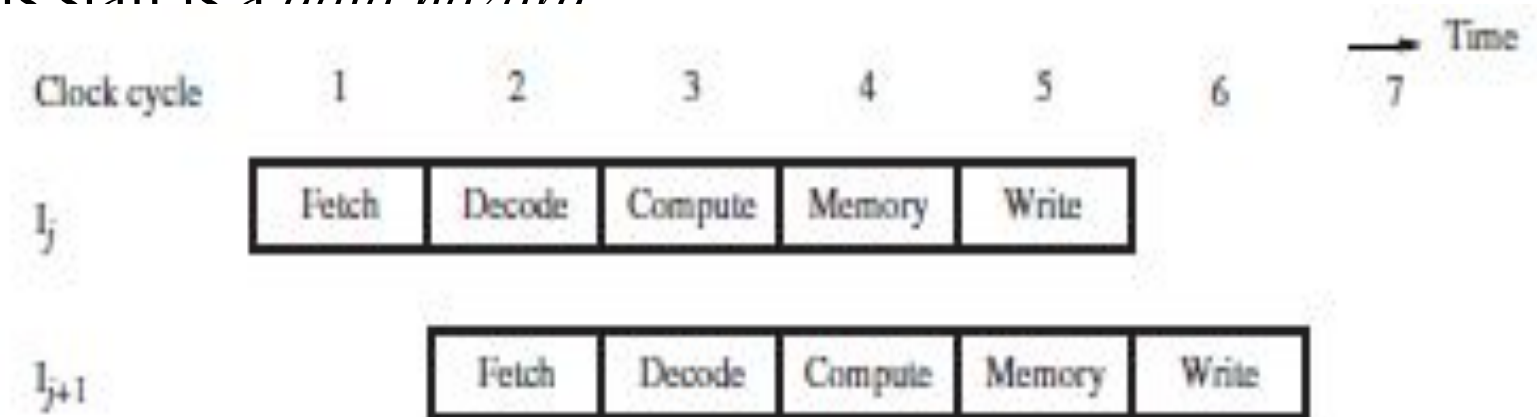
## Data dependency

- Consider two successive instructions  $I_j$  and  $I_{j+1}$ . [ex1]

$A \leftarrow 3 + A$  ;     $I_j$  :    Add R1,#3 ; R1  $\leftarrow$  R1 + 3

$B \leftarrow 4 * A$  ;     $I_{j+1}$  :    Mul R1,#4 ; R1  $\leftarrow$  4 \* R1

- Assume that the destination register of  $I_j$  matches one of the source registers of  $I_{j+1}$
- Result of  $I_j$  is written to destination in cycle 5, But  $I_{j+1}$  reads *old* value of register in cycle 3. Due to pipelining,  $I_{j+1}$  computation is incorrect. So *stall* (delay)  $I_{j+1}$  until  $I_j$  writes the new value. Condition requiring this stall is a *data hazard*

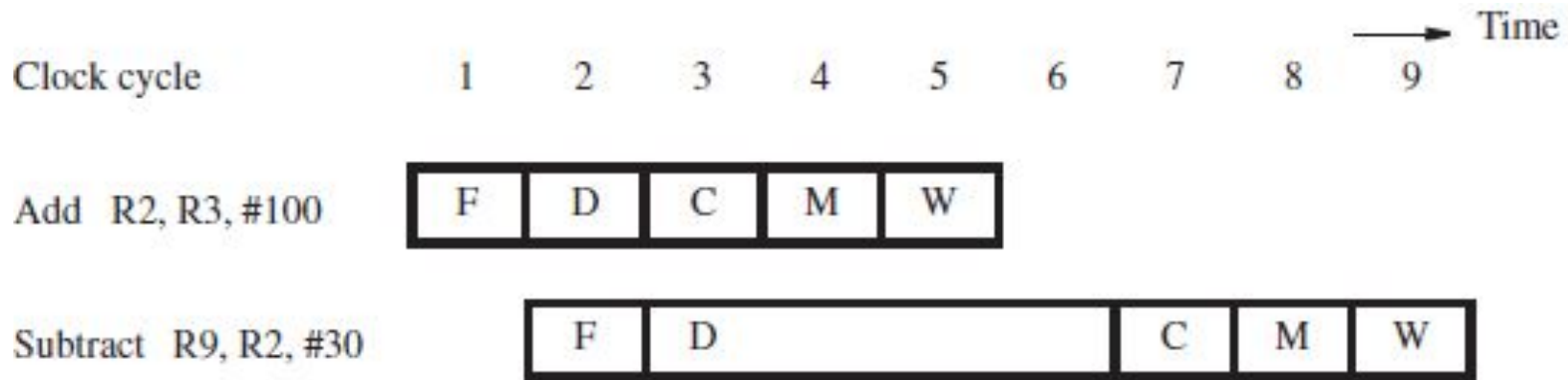


## Data Dependencies

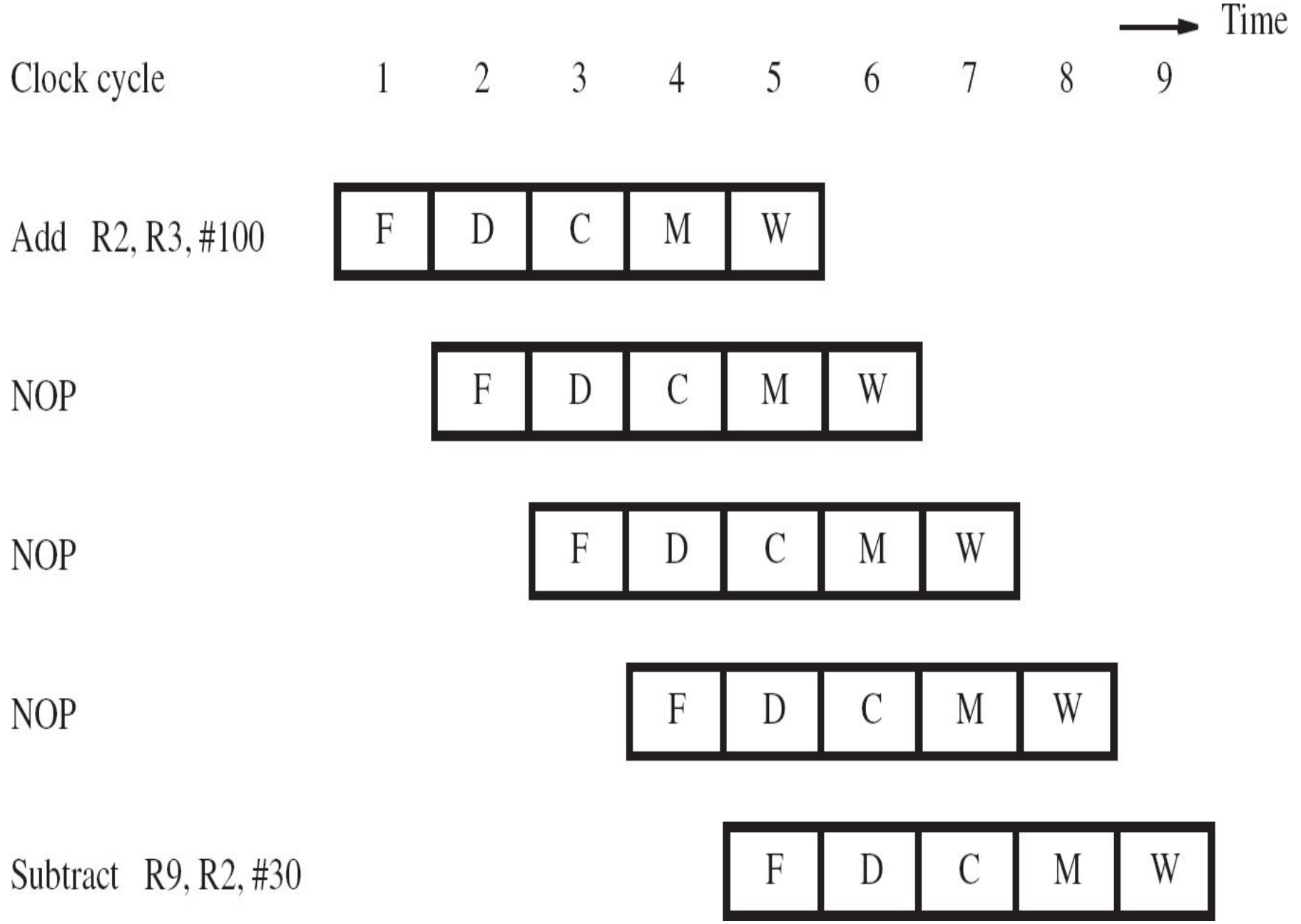
- Data dependencies leads to data hazards
- Now consider the specific instructions  
    Add           R2, R3, #100  
    Subtract    R9, R2, #30
- Destination R2 of Add is a source for Subtract
- There is a *data dependency*[Previous instruction *destination identifier is one of the source identifier in next instruction*] between them because R2 carries data from Add to Subtract
- On *non*-pipelined datapath, result is available in R2 because Add completes before Subtract.
- There are two solution for data dependencies
  1. Stalling
  2. Data Forwarding

## Stalling the Pipeline

- With pipelined execution, old value is still in register R2 when Subtract is in Decode stage
- So **stall** Subtract for 3 cycles in Decode stage.
- New value of R2 is also available in cycle 6
- The idle time from each NOP is called a *bubble* [*Here 3 bubbles*]



Control Unit send  
NOP signal





## Details for Stalling the Pipeline

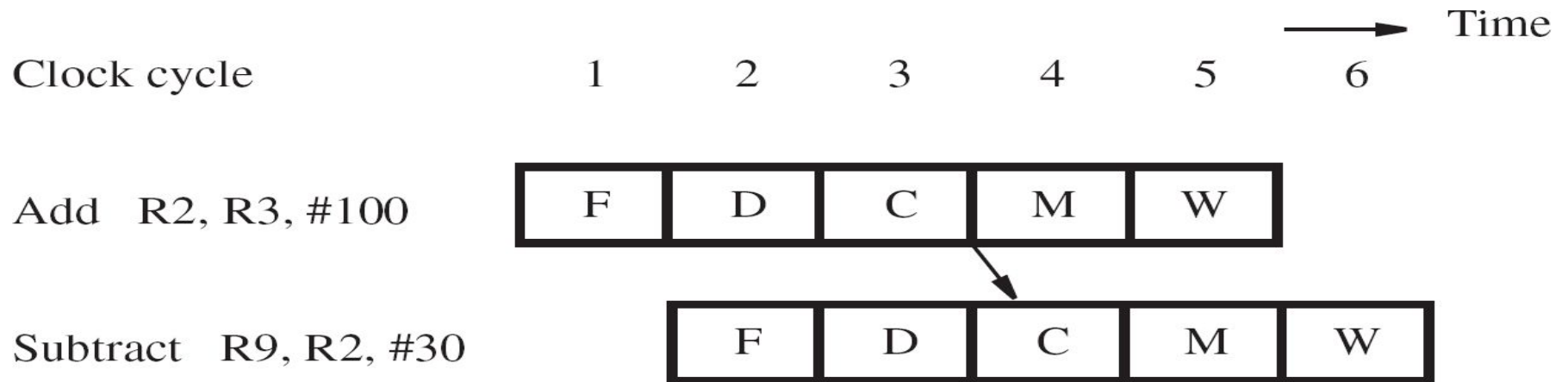
- Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
- R2 matches, so Subtract kept in Decode while Add allowed to continue normally
- Stall the Subtract instruction for 3 cycles by holding interstage buffer B1 contents steady
- But what happens after Add leaves Compute?
- Control signals are set in cycles 3 to 5 to create an *implicit* NOP (No-operation) in Compute

## Software Handling of Dependencies

- Compiler can generate & analyze instructions
- Data dependencies are evident from registers
- Compiler puts *explicit* NOP instructions between instructions having a dependency
- Delay ensures new value available in register but causes total execution time to increase
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit)

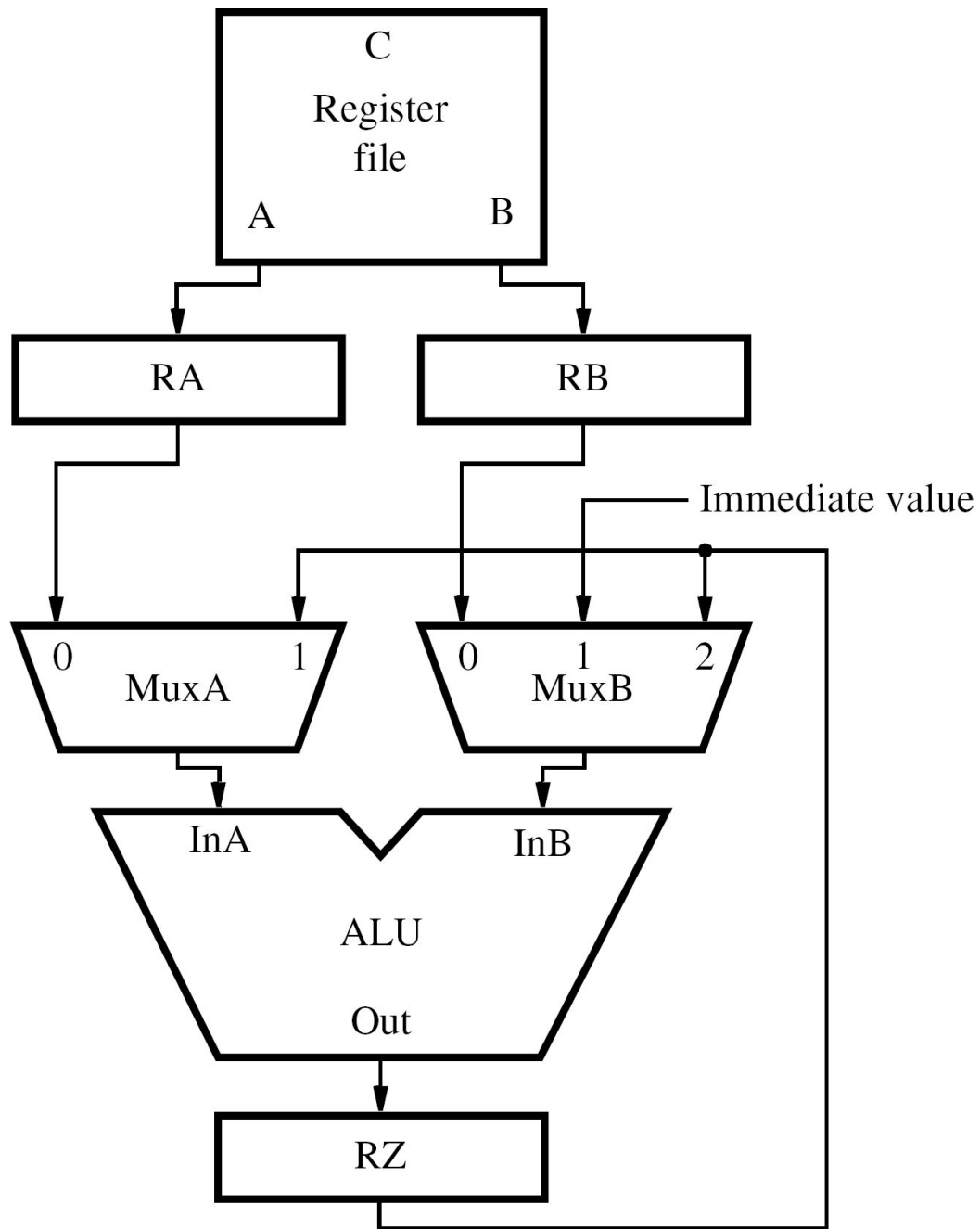
# Operand Forwarding

- **Operand forwarding** handles dependencies without the penalty of stalling the pipeline
- For the preceding sequence of instructions, new value for R2 is available at end of cycle 3



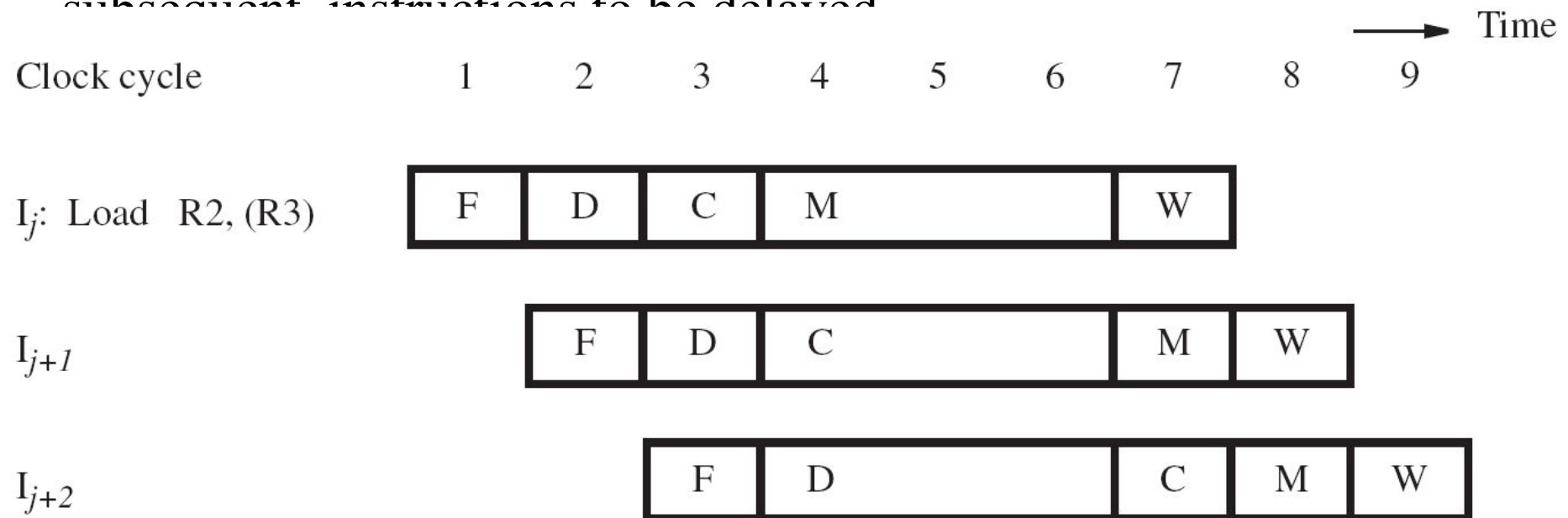
## Details for Operand Forwarding

- Introduce multiplexers before ALU inputs to use contents of register RZ as forwarded value
- Control circuitry now recognizes dependency in cycle 4  
Subtract is in Compute stage when
- Interstage buffers still carry register identifiers
- Compare destination of Add in Memory stage with source(s) of Subtract in Compute stage
- Set multiplexer control based on comparison



## Memory Delays with Cache Miss

- Memory delays can also cause pipeline stalls
- A cache memory holds instructions and data from the main memory, but is faster to access
- With a cache, typical access time is one cycle, but a cache miss requires accessing slower main memory with a much longer delay
- In pipeline, memory delay for one instruction causes subsequent instructions to be delayed



## Memory Delays with Cache hit

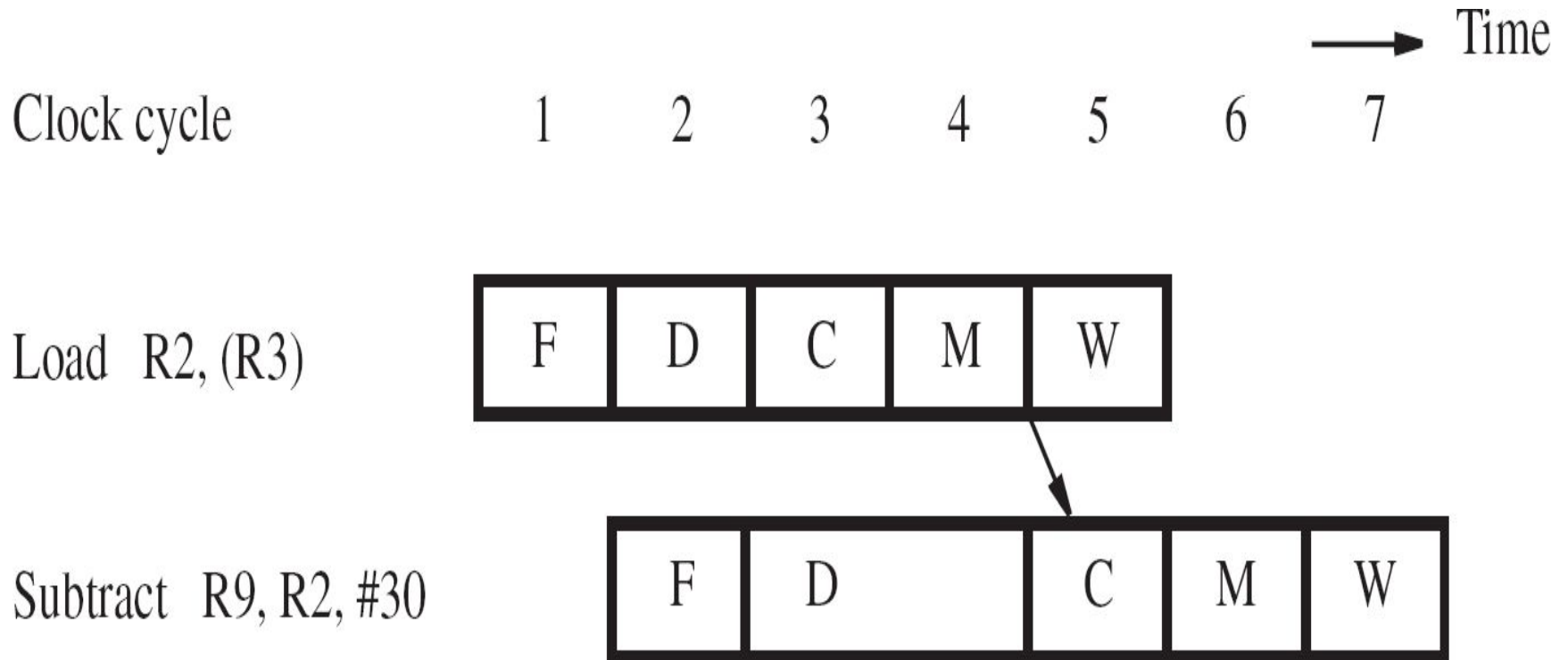
- Consider the instructions:

Load R2, (R3)

Subtract R9, R2, #30

- Even with a cache *hit*, a Load instruction may cause a short delay due to a data dependency. [The destination register R2 for the Load instruction is a source register for the Subtract instruction].
- Operand forwarding cannot be done in the same manner like Add, Sub instructions, because the data read from memory (the cache, in this case) are not available until they are loaded into register RY at the beginning of cycle 5.
- Therefore, the Subtract instruction must be stalled for one cycle, to delay the ALU operation. The memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.
- One-cycle stall required for correct value to be forwarded

## Memory Delays with Cache hit



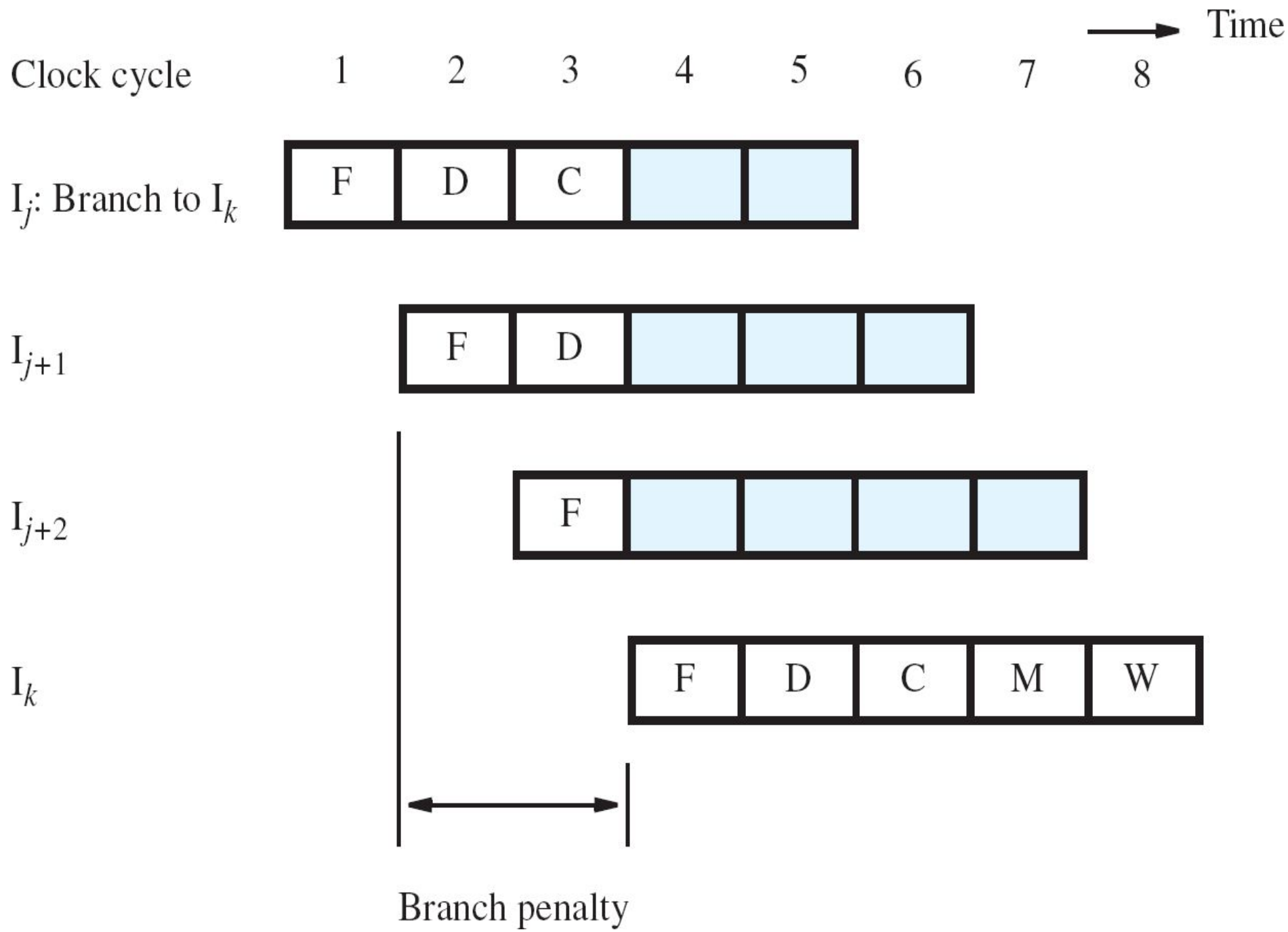


## Branch Delays

- Ideal pipelining: fetch each new instruction while previous instruction is being decoded
- Branch instructions alter execution sequence, but they must be executed first to determine whether and where to branch
- Any delay for determining branch outcome leads to an increase in total execution time

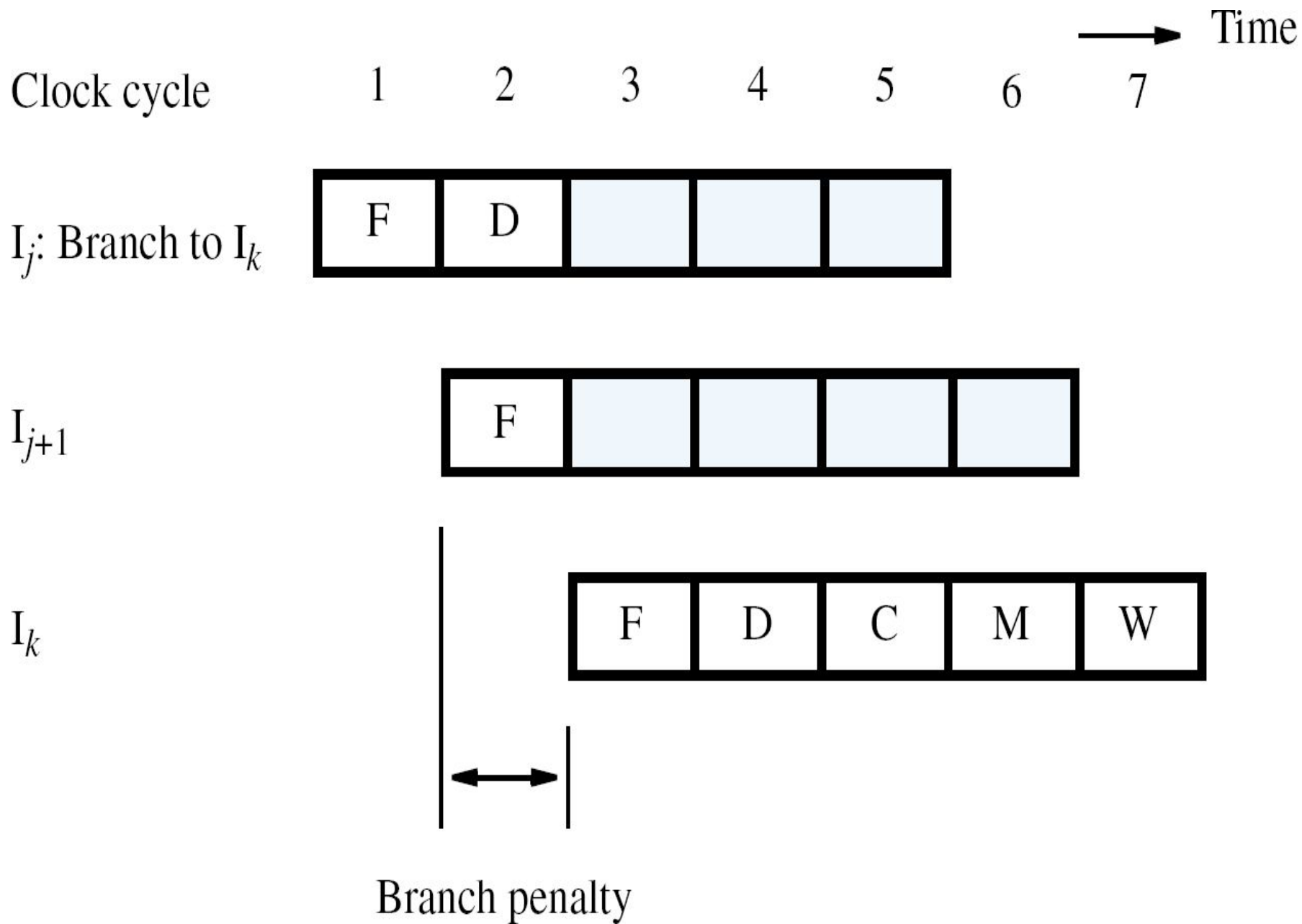
## Unconditional Branches

- Consider instructions  $I_j, I_{j+1}, I_{j+2}$  in sequence and  $I_j$  is an unconditional branch with target  $I_k$ .
- Branch instruction is fetched in cycle 1 and decoded in cycle 2, and the target address is computed in cycle 3. Hence, instruction  $I_k$  is fetched in cycle 4, after the program counter has been updated with the target address.
- In pipelined execution, instructions  $I_{j+1}$  and  $I_{j+2}$  are fetched in cycles 2 and 3, respectively, before the branch instruction is decoded and its target address is known. They must be discarded.
- This results a two-cycle delay called as **branch penalty**.



## Reducing the Branch Penalty

- Reducing the branch penalty requires the branch target address to be computed earlier in the pipeline.
- Rather than wait until the Compute stage, it is possible to determine the target address and update the program counter in the Decode stage.
- Thus, instruction  $I_k$  *can* be fetched one clock cycle earlier, reducing the branch penalty to one cycle.
- This time, only one instruction,  $I_{j+1}$ , *is fetched incorrectly, because the target* address is determined in the Decode stage.



## Conditional Branches

Consider a conditional branch instruction: `Branch_if_[R5]=[R6] LOOP`

- Requires not only target address calculation, but also requires comparison for condition.

1. Memory address  $\leftarrow$  [PC], Read memory, IR  $\leftarrow$  Memory data,

$$PC \leftarrow [PC] + 4$$

2. Decode instruction, RA  $\leftarrow$  [R5], RB  $\leftarrow$  [R6]

3. Compare [RA] to [RB], If [RA] = [RB], then PC  $\leftarrow$  [PC] + Branch offset

4. No action

5. No action

In Pipeline,

- If ALU performed the comparison, then two-cycle delay

# Techniques used to mitigate the effect of branches on execution time

1. Branch Delay Slot
2. Branch Prediction
  1. Static Branch Prediction
  2. Dynamic Branch Prediction

## 1. Branch Delay Slot

- Let both branch decision and target address be determined in Decode stage of pipeline
- Instruction immediately following a branch is always fetched, regardless of branch decision
- That next instruction is discarded with penalty, except when conditional branch is not taken
- The location immediately following the branch is called the *branch delay slot*
- Instead of conditionally discarding instruction in delay slot, *always* let it complete execution
- Let compiler find an instruction *before* branch to move into slot, if data dependencies permit. So it is called as *delayed branching* due to reordering
- If useful instruction put in slot, penalty is *zero*. If not possible, insert



---

Add	R7, R8, R9
Branch_if_[R3]=0	TARGET
$I_{j+1}$	
$\vdots$	
TARGET:	$I_k$

---

(a) Original sequence of instructions containing a conditional branch instruction

---

Branch_if_[R3]=0	TARGET
Add	R7, R8, R9
$I_{j+1}$	
$\vdots$	
TARGET:	$I_k$

---

(b) Placing the Add instruction in the branch delay slot where it is always executed

- The effectiveness of delayed branching depends on how often the compiler can reorder instructions to usefully fill the delay slot.
- Experimental data collected from many programs indicate that the compiler can fill a branch delay slot in 70 percent or more of the cases.

### 3.Branch Prediction

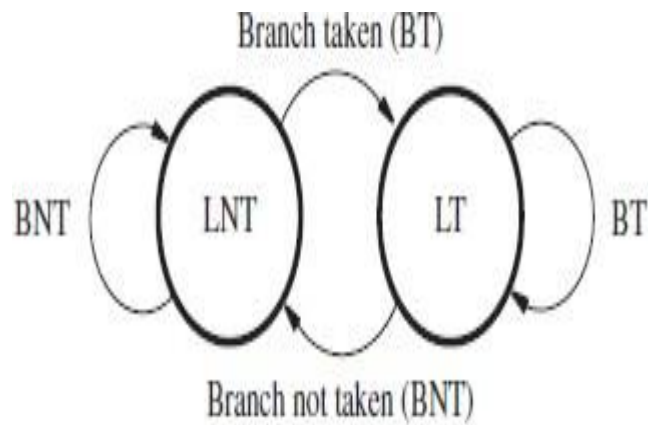
- A branch is decided in Decode stage (cycle 2) while following instruction is *always* fetched
- The decision to fetch this instruction is actually made in cycle 1, when the PC is incremented while the branch instruction itself is being fetched.
- Thus, to reduce the branch penalty further, the processor needs to anticipate that an instruction being fetched is a branch instruction and *predict its outcome to determine which instruction should be fetched in cycle 2.*
- Two aims: (a) *predict* the branch decision  
(b) use prediction *earlier* in cycle 1
- Two types of Prediction: 1.Static branch prediction  
2.Dynamic branch prediction

## Static Branch Prediction

- The simplest form of branch prediction is to assume that the **branch will not be taken** and to fetch the next instruction in sequential address order.
- If the prediction is correct, the fetched instruction is allowed to complete and there is no penalty.
- However, if it is determined that the **branch is to be taken**, the instruction that has been fetched is discarded and the correct branch target instruction is fetched.
- Misprediction incurs the full branch penalty. This simple approach is a form of *static branch prediction*.
- Adv: If branches are random, accuracy is 50%

## Dynamic Branch Prediction

- Track branch decisions during execution for *dynamic* prediction to improve accuracy
- The processor hardware[ Circuit called branch Predictor] assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that a branch instruction is executed.
- A dynamic prediction algorithm can use the result of the most recent execution of a branch instruction. The processor assumes that the next time the instruction is executed, the branch decision is likely to be the same as the last time. Hence, the algorithm may be described by the two-state machine.
- The two states are:
  - LT - Branch is likely to be taken
  - LNT - Branch is likely not to be taken



(a) A 2-state algorithm

- Suppose that the algorithm is started in state LNT. When the branch instruction is executed and the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT.
- The next time the same instruction is encountered, the branch is predicted as taken if the state machine is in state LT. Otherwise it is predicted as not taken.

- This simple scheme, which requires only a single bit to represent the history of execution for a branch instruction, works well inside program loops.
- Once a loop is entered, the decision for the branch instruction that controls looping will always be the same except for the last pass through the loop. Hence, each prediction for the branch instruction will be correct except in the last pass.
- The prediction in the last pass will be incorrect, and the branch history state machine will be changed to the opposite state.
- Unfortunately, the next time this same loop is entered—and assuming that there will be more than one pass through the loop—the state machine will lead to the wrong prediction for the first pass.
- Thus, repeated execution of the same loop results in mispredictions in the first pass and the last pass.

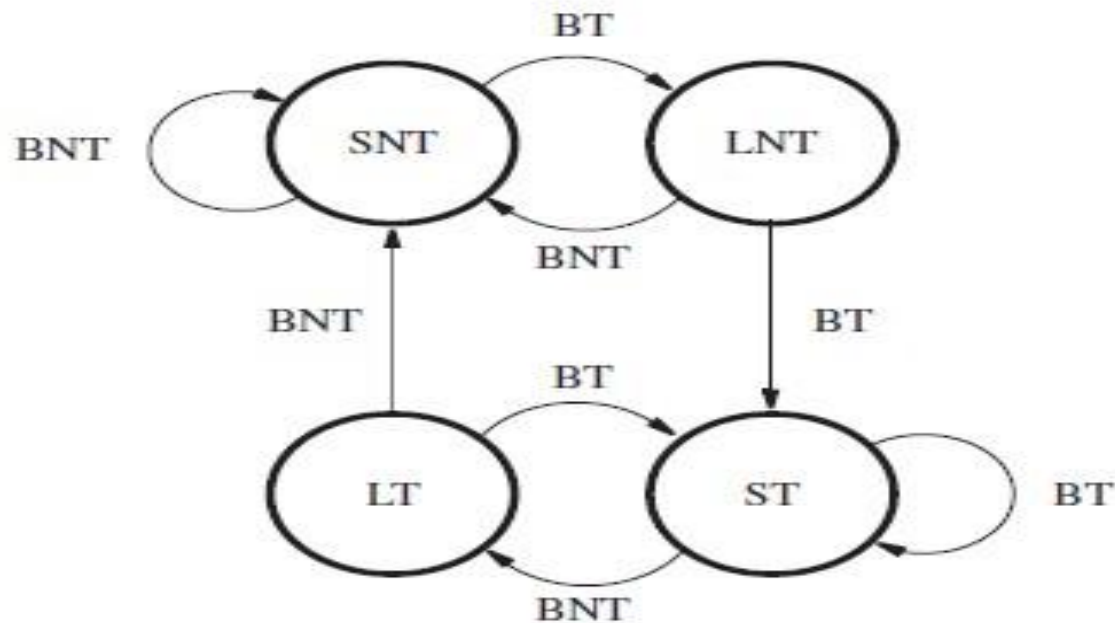
Better prediction accuracy can be achieved by keeping more information about execution history. An algorithm that uses four states is used to provide better accuracy. *The four states are:*

ST - Strongly likely to be

taken LT - Likely to be taken

LNT - Likely not to be taken

SNT - Strongly likely not to be taken



(b) A 4-state algorithm



- Assume that the state of the algorithm is initially set to LNT. After the branch instruction is executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT.
- As program execution progresses and the same branch instruction is encountered multiple times, the state of the prediction algorithm changes. The branch is predicted as taken if the state is either ST or LT. Otherwise, the branch is predicted as not taken.
- When executing a program loop, assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT.
- In the first pass, the prediction (not taken) will be wrong, and hence the state will be changed to ST. In all subsequent passes, the prediction will be correct, except for the last pass. At that time, the state will change to LT.
- When the loop is entered a second time, the prediction in the first pass will be to take the branch, which will be correct if there is more than one iteration. Thus, repeated execution of the same loop now results in only one misprediction in the last pass.

## Branch Target Buffer for Dynamic Prediction

- The key to improving performance is to increase the likelihood that the instruction fetched in cycle 2 is the correct one. This can be achieved only if branch prediction takes place in cycle 1, at the same time that the branch instruction is being fetched.
- To make this possible, the processor needs to keep more information about the history of execution.
- The required information is usually stored in a small, fast memory called the *branch target buffer*.
- The branch target buffer identifies branch instructions by their addresses. As each branch instruction is executed, the processor records the address of the instruction and the outcome of the branch decision in the buffer. The information is organized in the form of a lookup table, in which each entry includes:
  - the address of the branch instruction
  - one or two state bits for the branch prediction algorithm
  - the branch target address

## Performance Evaluation

The performance of the processor is evaluated by two parameters.

1. Execution time(T)
2. Throughput (P)

For a non-pipelined processor, the execution time,  $T$ , of a program is

calculated as  $T = (N \times S) / R$

where  $N$  is Number of instructions

$S$  is the average number of clock cycles it takes to fetch and execute one instruction and

$R$  is the clock rate in cycles per second

*This is often referred to as the **basic performance equation**.*

*Throughput, which is the number of instructions executed persecond.* For non-pipelined execution, the throughput,  $P_{np}$ , is given by

□ Pipelining improves performance by overlapping the execution of successive instructions, which increases instruction throughput even though an individual instruction is still executed in the same number of cycles.

□ For the five-stage pipeline, each instruction is executed in five cycles, but a new instruction can ideally enter the pipeline every cycle. Thus, in the absence of stalls, *S is equal to 1. [except first instruction]*

Throughput with pipelining is  $P_p = R$

□ A five-stage pipeline can potentially increase the throughput by a factor of five.

□ In general, an *n-stage pipeline* has the potential to *increase throughput n times*. Thus, it would appear that the *higher the value of n, the larger the performance gain*.

□ As the number of pipeline stages increases, there are more instructions being executed concurrently. Consequently, there are more potential dependencies between instructions that may lead to pipeline stalls. Also *the performance is affected by branch penalties*. Some recent processor implementations have used twenty or more pipeline stages

G1. We have two designs D1 and D2 for a synchronous pipeline processor. D1 has 5 pipeline stages with execution times of 3 nsec, 2 nsec, 4 nsec, 2nsec and 3 nsec while the design D2 has 8 pipeline stages each with 2nsec execution time How much time can be saved using design D2 over design D1 for executing 100 instructions?

Execution time for the first instruction =  $KT$

Execution time for the remaining instructions =  $(N-1)T$

So, total execution time =  $KT + (N-1)T = (K + N - 1) * T$

Where  $K$  = total number of stages ,  $N$  = total number of instructions  
and  $T$  = maximum clock cycle

For D1 :  $K = 5$  ,  $N = 100$  and  $T = 4\text{ns}$

Total execution time =  $(5 + 100 - 1) * 4 = 416\text{nsec}$

For D2 :  $K = 8$  ,  $N = 100$  and  $T = 2\text{ns}$

Total execution time =  $(8 + 100 - 1) * 2 = 214\text{nsec}$

Thus, time saved using D2 over D1 =  $416 - 214 = 202\text{nsec}$

G2. A non pipelined single cycle processor operating at 100 MHz is converted into a synchronous pipelined processor with five stages requiring 2.5 nsec, 1.5 nsec, 2 nsec, 1.5 nsec and 2.5 nsec, respectively. The delay of the latches is 0.5 nsec. Find the speedup for the pipeline processor.

For non pipelined system time required =  $2.5+1.5+2.0+1.5+2.5$   
= 10nsec For pipelined system =  $\text{Max}(\text{stage delay}) + \text{Max}(\text{Latch delay})$   
 $\Rightarrow 2.5+0.5 = 3.0\text{nsec}$

Speedup =  $\text{time in non-pipelined system} / \text{time in pipelined system}$   
 $= 10/3 = 3.33$

G3. Consider a non-pipelined processor with a clock rate of 2.5 gigahertz and average cycles per instruction of four. The same processor is upgraded to a pipelined processor with five stages; but due to the internal pipeline delay, the clock speed is reduced to 2 gigahertz. Assume that there are no stalls in the pipeline. What speed up is achieved in this pipelined processor?

$$\text{Speedup} = \text{ExecutionTimeOld} / \text{ExecutionTimeNew}$$

$$\text{ExecutionTimeOld} = \text{CPIOld} * \text{CycleTimeOld}$$

[Here CPI is Cycles Per Instruction]

$$= \text{CPIOld} * \text{CycleTimeOld}$$

$$= 4 * 1/2.5 \text{ Nanoseconds}$$

$$= 1.6 \text{ ns}$$

Since there are no stalls, CPUnew can be assumed 1 on average.

$$\text{ExecutionTimeNew} = \text{CPInew} * \text{CycleTimenew}$$

$$= 1 * 1/2$$

$$= 0.5$$

$$\text{Speedup} = 1.6 / 0.5 = 3.2$$

G4. Consider a pipelined processor with the following four stages: IF: Instruction Fetch  
 ID: Instruction Decode and Operand Fetch EX: Execute  
 WB: Write Back

The IF, ID and WB stages take one clock cycle each to complete the operation. The number of clock cycles for the EX stage depends on the instruction. The ADD and SUB instructions need 1 clock cycle and the MUL instruction needs 3 clock cycles in the EX stage.

Operand forwarding is used in the pipelined processor. What is the number of clock cycles taken to complete the following sequence of

~~ADD R2, R1, R0~~       $R2 \leftarrow R0 + R1$   
 ADD R2, R1, R0       $R2 \leftarrow R0 + R1$   
 MUL R4, R3, R2       $R4 \leftarrow R3 * R2$   
 SUB R6, R5, R4       $R6 \leftarrow R5 - R4$



Order of instruction cycle phases —IF‖ —ID‖ —EX‖ —WB‖. We have 3 instructions that represents data dependency.

If operand forwarding is used, then No. of cycles required=8

	1	2	3	4	5	6	7	8
ADD R2,R1,R0	IF	ID	EX	WB				
MUL R4,R3,R2		IF	ID	EX	EX	EX	WB	
SUB R6,R5,R4			IF	ID	----	----	EX	WB

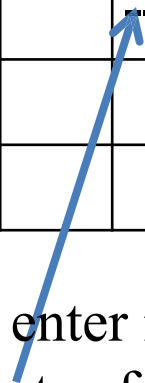
If operand forwarding is not used, then No. of cycles required=10

	1	2	3	4	5	6	7	8	9	10
ADD R2,R1,R0	IF	ID	EX	WB						
MUL R4,R3,R2		IF	ID	---	EX	EX	EX	WB		
SUB R6,R5,R4			IF	ID	---	---	---	---	EX	WB

G5. An instruction pipeline consists of 4 stages: Fetch(F), operand field (D), Decode (E), and Write Back(W). The instructions in a certain instruction sequence are executed in the different number of clock cycles as shown by the table below. No. of cycles needed for each instruction is given below. Find the number of clock cycles needed to perform the 5 instructions.

Instruction	F	D	E	W
1	1	2	1	1
2	1	2	2	1
3	2	1	3	2
4	1	3	2	1
5	1	2	1	2

	Clock														
<b>Instru ction</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>	<b>15</b>
<b>I1</b>	<b>F</b>	<b>D</b>	<b>D</b>	<b>E</b>	<b>W</b>										
<b>I2</b>		<b>F</b>	----	<b>D</b>	<b>D</b>	<b>E</b>	<b>E</b>	<b>W</b>							
<b>I3</b>			----	<b>F</b>	<b>F</b>	<b>D</b>	----	<b>E</b>	<b>E</b>	<b>E</b>	<b>W</b>	<b>W</b>			
<b>I4</b>				----	----	<b>F</b>	----	<b>D</b>	<b>D</b>	<b>D</b>	<b>E</b>	<b>E</b>	<b>W</b>		
<b>I5</b>					----	----	----	<b>F</b>	----	----	<b>D</b>	<b>D</b>	<b>E</b>	<b>W</b>	<b>W</b>



Here I2 is not enter into the decode stage and if I3 is enter into Fetch stage then the contents of I2 in Interstage buffer B1[F/D buffer] is erased

15 Clock cycles are required

G6.A 4-stage pipeline has the stage delays as 150, 120, 160 and 140 nanoseconds respectively. Registers that are used between the stages have a delay of 5 nanoseconds each. Assuming constant clocking rate, what is the total time taken to process 1000 data items on this pipeline?

Delay between each stage is 5 ns.

Total delay in pipeline = 640ns [ Max of 160 is taken for all stages, so  
delay = 160+160+160+160]

Total delay for one data item = 640 + 5\*3 (There are 3 intermediate registers) = 655ns

For 1000 data items, first data will take 655 ns to complete and remaining 999 data will take max of all the stages that is 160 ns + 5 ns register delay

Total Delay = 655 + 999\*165 ns which is equal to 165.5microsecond.

G7. Comparing the time  $T_1$  taken for a single instruction on a pipelined CPU with time  $T_2$  taken on a non-pipelined but identical CPU then which one takes more time? Give your own example.

$T_1 > T_2$

G8. A processor takes 12 cycles to complete an instruction I. The corresponding pipelined processor uses 6 stages with the execution times of 3, 2, 5, 4, 6 and 2 cycles respectively. What is the speedup, assuming that a very large number of instructions are to be executed?

Let the large number of instructions be  $N$ .

Time required for sequential execution = number of cycles  $\times$  number of instructions =  $12N$

Time required in a pipelined processor,

Execution time for the first instruction =  $K \times T$

Execution time for the remaining instructions =  $(N-1)T$

So, total execution time =  $K \times T + (N-1)T = (K + N - 1) \times T = (6 + N - 1) \times 6 = (5 + N) \times 6 = 30 + 6N \approx 6N$  [because  $N$  is large]

Speed up =  $12N / 6N = 2$

G9. A 5 stage pipelined CPU has the following sequence of stages: IF — Instruction fetch from instruction memory, RD — Instruction decode and register read, EX — Execute: ALU operation for data and address computation, MA — Data memory access - for write access, the register read at RD stage is used, WB — Register write back.

Consider the following sequence of instructions:

I1 : Load R0, R0 ← M[loc1]

I2 : Add R0, R0 ← R0 + R0

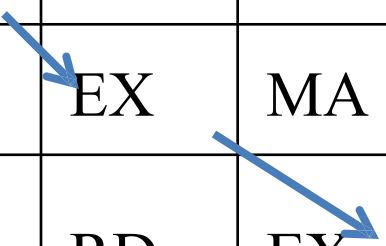
I3 : Subtract R2, R2 ← R2 - R0

At each stage take one clock cycle.

What is the number of clock cycles taken to complete the above sequence of instructions starting from the fetch of I1 if operand forwarding is allowed and not allowed?

If we use operand forwarding from memory stage :

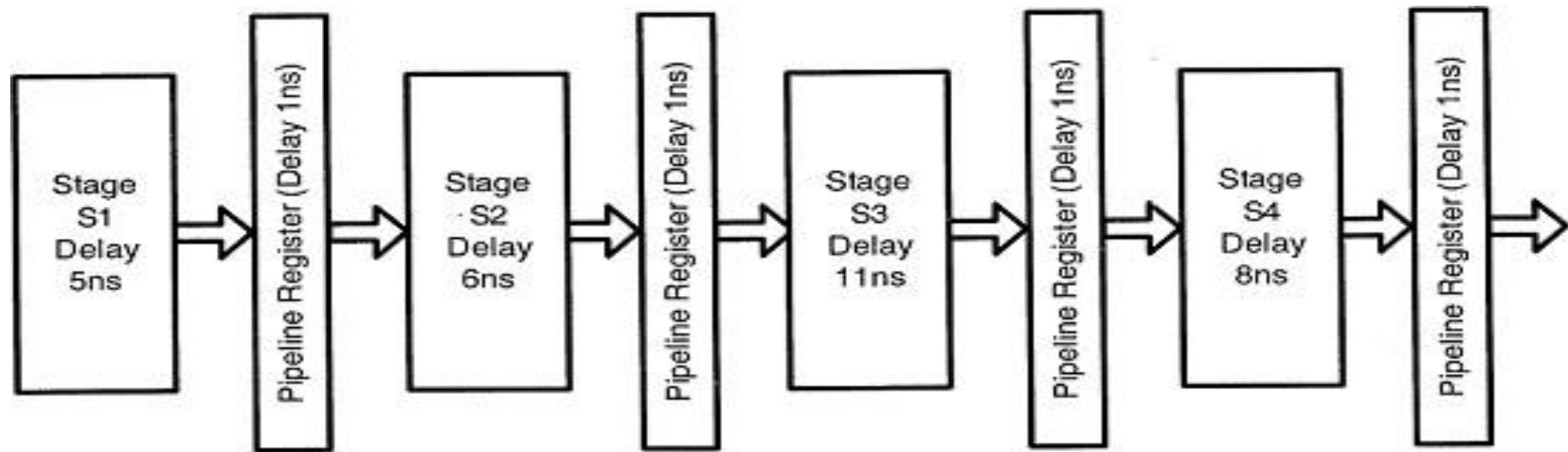
	T1	T2	T3	T4	T5	T6	T7	T8
11	IF	RD	EX	MA	WB			
12		IF	RD	----	EX	MA	WB	
13			IF	----	RD	EX	MA	WB



If we do not use operand forwarding from memory stage :

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
11	IF	RD	EX	MA	WB						
12		IF	----	----	RD	EX	MA	WB			
13					IF	----	----	RD	EX	MA	WB

G10. Consider an instruction pipeline with four stages (S1, S2, S3 and S4) each with combinational circuit only. The pipeline registers are required between each stage and at the end of the last stage. Delays for the stages and for the pipeline registers are as given in the figure, what is the speed up of the pipeline in steady state under ideal conditions when compared to the corresponding non-pipeline implementation ?



Registers overhead is not counted in normal time execution [without pipeline]. So the total time will be  $5+6+11+8=30\text{ns}$

Now, for pipeline, each stage will be of  $11\text{ ns} + 1\text{ ns}$  (for register delay)  $=12\text{ ns}$  [steady state output is produced after every pipeline cycle]

Speedup  $= 30\text{ns} / 12\text{ns} = 2.5\text{ns}$



G11. A 5-stage pipelined processor has Instruction Fetch(IF), Instruction Decode(ID), Operand Fetch(OF), Perform Operation(PO) and Write Operand(WO) stages. The IF, ID, OF and WO stages take 1 clock cycle each for any instruction. The PO stage takes 1 clock cycle for ADD and SUB instructions, 3 clock cycles for MUL instruction and 6 clock cycles for DIV instruction respectively. Operand forwarding is used in the pipeline. What is the number of clock cycles needed to execute the following sequence of instructions?

Instruction      Meaning of instruction

I0 :MUL R2 ,R0 ,R1	$R2 \leftarrow R0 * R1$
I1 :DIV R5 ,R3 ,R4	$R5 \leftarrow R3 / R4$
I2 :ADD R2 ,R5 ,R2	$R2 \leftarrow R5 + R2$
I3 :SUB R5 ,R2 ,R6	$R5 \leftarrow R2 - R6$

Clock Cycles->

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instruction 0	IF	ID	OF	PO	PO	PO	WO								
Instruction 1		IF	ID			OF	PO	PO	PO	PO	PO	PO	WO		
Instruction 2			IF	ID								OF	PO	WO	
Instruction 3				IF	ID								OF	PO	WO

Stall Cycles

Operand Forwarding(  
computed value of  
**R3 / R4** is  
Forwarded)

Operand Forwarding(  
computed  
value of **R5 + R2** is  
Forwarded)

G12. Consider a 4 stage pipeline processor. The number of cycles needed by the four instructions I1, I2, I3, I4 in stages S1, S2, S3, S4 is shown below: What is the number of cycles needed to execute the following loop? For (i=1 to 2) {I1; I2; I3; I4;}




	S1	S2	S3	S4
I1	2	1	1	1
I2	1	3	2	2
I3	1	1	1	3
I4	1	2	2	2

Number of cycles needed is 23



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	(23)
$J_1$	$S_1$	$S_1$	$S_2$	$S_3$	$S_4$																		
$J_2$			$S_1$	$S_2$	$S_2$	$S_2$	$S_3$	$S_3$	$S_4$	$S_4$													
$J_3$				$S_1$	$S_1$	$\times$	$S_2$	$\times$	$S_3$	$\times$	$S_4$	$S_4$	$S_4$										
$J_4$						$S_1$	$\times$	$S_2$	$S_2$	$S_2$	$S_3$	$\times$	$\times$	$S_4$	$S_4$								
$I_1$						$\times$	$S_1$	$S_1$	$\times$	$S_2$	$\times$	$S_3$	$\times$	$\times$	$\times$	$S_4$							
$I_2$									$S_1$	$\times$	$S_2$	$S_2$	$S_2$	$S_3$	$S_3$	$\times$	$S_4$	$S_4$					
$I_3$										$S_1$	$S_1$	$\times$	$\times$	$S_2$	$\times$	$S_3$	$\times$	$\times$	$S_4$	$S_4$	$S_4$		
$I_4$												$S_1$	$\times$	$\times$	$S_2$	$S_2$	$S_3$	$S_3$	$\times$	$\times$	$\times$	$S_4$	$S_4$

G13. Consider an instruction pipeline with five stages without any branch prediction: Fetch Instruction (FI), Decode Instruction (DI), Fetch Operand (FO), Execute Instruction (EI) and Write Operand (WO). The stage delays for FI, DI, FO, EI and WO are 5 ns, 7 ns, 10 ns, 8 ns and 6 ns, respectively. There are intermediate storage buffers after each stage and the delay of each buffer is 1 ns. A program consisting of 12 instructions  $I_1, I_2, I_3, \dots, I_{12}$  is executed in this pipelined processor. Instruction  $I_4$  is the only branch instruction and its branch target is  $I_9$ . If the branch is taken during the execution of this program, calculate the time (in ns) needed to complete the program.

	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>	T <sub>9</sub>	T <sub>10</sub>	T <sub>11</sub>	T <sub>12</sub>	T <sub>13</sub>	T <sub>14</sub>	T <sub>15</sub>
I <sub>1</sub>	FI	DI	FO	EI	WO										
I <sub>2</sub>		FI	DI	FO	EI	WO									
I <sub>3</sub>			FI	DI	FO	EI	WO								
I <sub>4</sub>				FI	DI	FO	EI	WO							
					stall	<b>Branch Penalty</b> 									
						stall	<b>Branch Penalty</b> 								
							stall	<b>Branch Penalty</b> 							
I <sub>9</sub>								FI	DI	FO	EI	WO			
I <sub>10</sub>									FI	DI	FO	EI	WO		
I <sub>11</sub>										FI	DI	FO	EI	WO	
I <sub>12</sub>											FI	DI	FO	EI	WO

Cycle time = max of all stage delays + buffer delay = max (5 ns, 7 ns, 10 ns, 8 ns, 6 ns) + 1 = 10+1 = 11ns

Out of all the instructions  $I_1, I_2, I_3, \dots, I_{12}$  it is given that only  $I_4$  is a branch instruction and when  $I_4$  takes branch the control will jump to instruction  $I_9$  as  $I_9$  is the target instruction. From the timing diagram there is a gap of only 3 stall cycles between  $I_4$  and  $I_9$  because after  $I_4$  enters Decode Instruction (DI) whether there is a branch or not will be known at the end of Execute Instruction (EI) phase. So there are total 3 phases namely DI, FO, EI are in stall. After 3 stall cycles  $I_9$  will start executing as that is the branch target.

Total no. of clock cycles to complete the program = 15

Since 1 clock cycle = 11ns, time to complete the program =  $15 * 11 = 165$ ns

G14. Instruction execution in a processor is divided into 5 stages, Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execute (EX), and Write Back (WB). These stages take 5, 4, 20, 10 and 3 nanoseconds (ns) respectively. A pipelined implementation of the processor requires buffering between each pair of consecutive stages with a delay of 2 ns. Two pipelined implementations of the processor are contemplated:

- (i) a naive pipeline implementation (NP) with 5 stages and
- ii) an efficient pipeline (EP) where the OF stage is divided into stages OF1 and OF2 with execution times of 12 ns and 8 ns respectively.

Find the speedup (correct to two decimal places) achieved by EP over NP in executing 20 independent instructions with no hazards.

$$\begin{aligned} &\text{Speed up of efficient pipeline over native pipeline} \\ &= \text{Naive pipeline execution time} / \text{efficient pipeline execution time} \\ &= 528 / 350 = 1.51 \end{aligned}$$



### Naive Pipeline implementation:

The stage delays are 5, 4, 20, 10 and 3. And buffer delay = 2ns  
So clock cycle time = max of stage delays + buffer delay  
 $= \max(5, 4, 20, 10, 3) + 2 = 20 + 2 = 22\text{ns}$

Execution time for n-instructions in a pipeline with k-stages =  $(k+n-1) * \text{clock cycle time}$

Execution time for 20 instructions in the pipeline with 5-stages  
 $= (5+20-1) * 22\text{ns} = 24 * 22\text{ns} = 528\text{ns}$

### Efficient Pipeline implementation:

OF phase is split into two stages OF1, OF2 with execution times of 12ns, 8ns

New stage delays in this case = 5, 4, 12, 8, 10, 3 and buffer delay is the same 2ns.

So clock cycle time = max of stage delays + buffer delay  
 $= \max(5, 4, 12, 8, 10, 3) + 2 = 12 + 2 = 14\text{ns}$

Execution time =  $(k+n-1) * \text{clock cycle time}$ , Here no. of stages,  $k = 6$

No. of instructions = 20

Execution time =  $(6+20-1) * 14 = 25 * 14 = 350\text{ns}$

### Solved Problem

Consider the pipelined execution of the following sequence of instructions:

Add R4, R3, R2

Or R7, R6, R5

Subtract R8, R7, R4

Initially, registers R2 and R3 contain 4 and 8, respectively. Registers R5 and R6 contain 128 and 2, respectively. Assume that the pipeline provides forwarding paths to the ALU from registers RY and RZ. The first instruction is fetched in cycle 1, and the remaining instructions are fetched in successive cycles.

Draw a diagram to show the pipelined execution of these instructions assuming that the processor uses operand forwarding. Then, describe the contents of registers RY and RZ during cycles 4 to 7.

Clock Cycles	1	2	3	4	5	6	7	8
ADD R4,R3,R2	F	D	C	M	W			
OR R7,R6,R5		F	D	C	M	W		
SUB R8,R7,R4			F	D	C	M	W	

There are data dependencies involving registers R4 and R7. The Subtract instruction needs the new values for these registers before they are written to the register file. Hence, those values need to be forwarded to the ALU inputs when the Subtract instruction is in the Compute stage of the pipeline.

**As for the contents of registers RY and RZ during cycles 4 to 7,**  
 Using the initial values for registers R2 and R3, the Add instruction generates the result of 12 in cycle 3. That result is available in register RZ during cycle 4.

□ In cycle 4, the Or instruction generates the result of 130. That result is placed in register RZ to be available during cycle 5. The result of 12 for the Add instruction is in register RY during cycle 5.

□ In cycle 5, the Subtract instruction is in the Compute stage. To generate a correct result, forwarding is used to provide the value of 130 in register RY and the value of 12 in register RZ. The result from the ALU is  $130 - 12 = 118$ . This result is available in register RZ during cycle 6. The result of the Or instruction, 130, is in register RY during in cycle 6.

□ In cycle 6, the Subtract instruction is in the Memory stage. The unspecified instruction following the Subtract instruction is generating a result in the Compute stage. In cycle 7, the result of the unspecified instruction is in register RZ, and the result of the Subtract instruction is in register RY.

Ex1: Consider the following instructions at the given addresses in the memory:

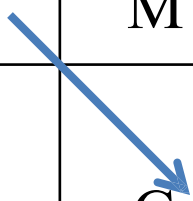
1000	Add	R3, R2, #20
1004	Subtract	R5, R4, #3
1008	And	R6, R4, #0x3A
1012	Add	R7, R2, R4

Initially, registers R2 and R4 contain 2000 and 50, respectively. These instructions are executed in a computer that has a five-stage pipeline. The first instruction is fetched in clock cycle 1, and the remaining instructions are fetched in successive cycles.

a) *Draw a diagram that represents the flow of the instructions through the pipeline. Describe the operation being performed by each pipeline stage during clock cycles 1 through 8.*

b) *Also, describe the contents of registers IR, PC, RA, RB, RY, and RZ in the pipeline during cycles 2 to 8.*

Clock Cycles	1	2	3	4	5	6	7	8
Add R3,R2,#20	F	D	C	M	W			
Sub R5,R4,#3		F	D	C	M	W		
And R6,R4,#0x3A			F	D	C	M	W	
Add R7,R2,R4				F	D	C	M	W



The description of activity in each stage during each

Cycle	Stage	Activity
1	F	fetching Add instruction
2	F	fetching Subtract instruction
	D	decoding Add instruction, reading register R2 (value 2000)
3	F	fetching And instruction
	D	decoding Subtract instruction, reading register R4 (value 50)
	C	performing arithmetic $2000 + 20 = 2020$ for Add instruction

4	F D C M	fetching Add instruction decoding And instruction, reading register R4 (value 50) performing arithmetic $50 - 3 = 47$ for Subtract instruction no operation for Add instruction
5	D  C M W	decoding Add instruction, reading register R2 (value 2000) and register R4 (value 50) performing logic operation $50 \text{ AND } 3A_{16} = 50$ for And instruction no operation for Subtract instruction write result of 2020 for Add instruction to register R3
6	C M W	performing arithmetic $2000 + 50 = 2050$ for Add instruction no operation for And instruction write result of 47 for Subtract instruction to register R5
7	M W	no operation for Add instruction write result of 50 for And instruction to register R6
8	W	write result of 2050 for Add instruction to register R7

The contents of each register during each cycle are

Cycles	1	2	3	4	5	6	7	8
IR (Decode)	--	Add	Sub	And	Add	--	--	--
PC	1000	1004	1008	1012	1016	1020	1024	1028
RA	--	--	2000	50	50	2000	--	--
RB	--	--	--	--	--	50	--	--
RZ	--	--	--	2020	47	50	2050	--
RY	--	--	--	--	2020	47	50	2050



Ex2: Consider the following instructions at the given addresses in the memory:

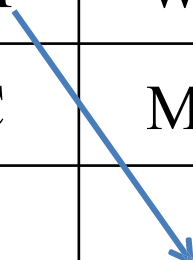
1000	Add	R3, R2, #20
1004	Sub	R5, R4, #3
1008	And	R6, R3, #0x3A
1012	Add	R7, R2, R4

Initially, registers R2 and R4 contain 2000 and 50, respectively. These instructions are executed in a computer that has a five-stage pipeline. The first instruction is fetched in clock cycle 1, and the remaining instructions are fetched in successive cycles.

a) *Draw a diagram that represents the flow of the instructions through the pipeline. Describe the operation being performed by each pipeline stage during clock cycles 1 through 8.*

b) *Also, describe the contents of registers IR, PC, RA, RB, RY, and RZ in the pipeline during cycles 4 to 7.*

Clock Cycles	1	2	3	4	5	6	7	8
Add R3,R2,#20	F	D	C	M	W			
Sub R5,R4,#3		F	D	C	M	W		
And R6,R3,#0x3A			F	D	C	M	W	
Add R7,R2,R4				F	D	C	M	W



The description of activity in each stage during each

Cycle	Stage	Activity
1	F	fetching Add instruction
2	F	fetching Subtract instruction
	D	decoding Add instruction, reading register R2 (value 2000)
3	F	fetching And instruction
	D	decoding Subtract instruction, reading register R4 (value 50)
	C	performing arithmetic $2000 + 20 = 2020$ for Add instruction

4	F D C M	fetching Add instruction decoding And instruction, <b>reading register R3 (value unknown)</b> performing arithmetic $50 - 3 = 47$ for Subtract instruction no operation for Add instruction
5	D  C M W	decoding Add instruction, reading register R2 (value 200) and register R4 (value 50) performing logic operation <b><math>2020 \text{ AND } 3A_{16} = 32</math> for And instruction</b> no operation for Subtract instruction write result of 2020 for Add instruction to register R3
6	C M W	performing arithmetic $2000 + 50 = 2050$ for Add instruction no operation for And instruction write result of 47 for Subtract instruction to register R5
7	M W	no operation for Add instruction W write result of <b>32 for And instruction to register R6</b>

The contents of each register during each cycle are

Cycles	1	2	3	4	5	6	7	8
IR (Decode)	--	Add	Sub	And	Add	--	--	--
PC	1000	1004	1008	1012	1016	1020	1024	1028
RA	--	--	2000	<b>R3 Value</b>	50	2000	--	--
RB	--	--	--	--	--	50	--	--
RZ	--	--	--	2020	47	32	2050	--
RY	--	--	--	--	2020	47	32	2050

The contents of RZ in cycle 6 and RY in cycle 7 are determined as,  
 $(2020 \text{ AND } 3A_{16}) = (7E4_{16} \text{ AND } 3A_{16}) = 20_{16} = 32$