# Basic Processing Unit

FOR CSE & CST, IV SEM

# Introduction

- Instruction Set Processor (ISP) or processor
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executed by carrying out a sequence of more rudimentary operations such as fetch, decode execute and store.

# Fundamental Concepts

- Processor *fetches one instruction at a time* and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

# Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

  $IR \leftarrow [[PC]]$

- Assuming that the memory is byte addressable, increment the contents of the PC by 1 to get the next instruction(fetch phase).

  $PC \leftarrow [PC] + 1$

- Carry out the actions specified by the instruction in the IR (execution phase).

# Processor Organization

Datapath

# Components of the processor

- ***ALU***
- Registers for temporary storage
- Various digital circuits for executing different operations.(gates, MUX, decoders, counters).
- Internal path for movement of data between ALU and registers.
- Driver circuits for transmitting signals to external units.
- Receiver circuits for incoming signals from external units.
- ***Program Counter (PC):***
- Keeps track of execution of a program
- Contains the memory address of the next instruction to be fetched and executed.
- ***Memory Address Register(MAR):***
- Holds the address of the location to be accessed.
- I/P of MAR is connected to Internal bus and an O/P to external bus.

- ***Memory Data Register(MDR):***

- Contains data to be written into or read out of the addressed location.
- Data can be loaded into MDR either from memory bus or from internal processor bus.
- The data and address lines are connected to the internal bus via MDR and MAR.

***Registers:***

- The processor registers R0 to Rn-1 vary considerably from one processor to another.
- Registers are provided for general purpose used by programmer.
- Registers Y,Z &TEMP are temporary registers used by processor during the execution of instruction.

***Multiplexer:***

- Select either the output of the register Y or a constant value 4 to be provided as input A of the ALU.
- Constant 4 is used by the processor to increment the contents of PC.

# ALU:

- Used to perform arithmetic and logical operation.

Data Path:

The *registers, ALU and interconnecting bus* are collectively referred to as the data path.

# Register Transfers

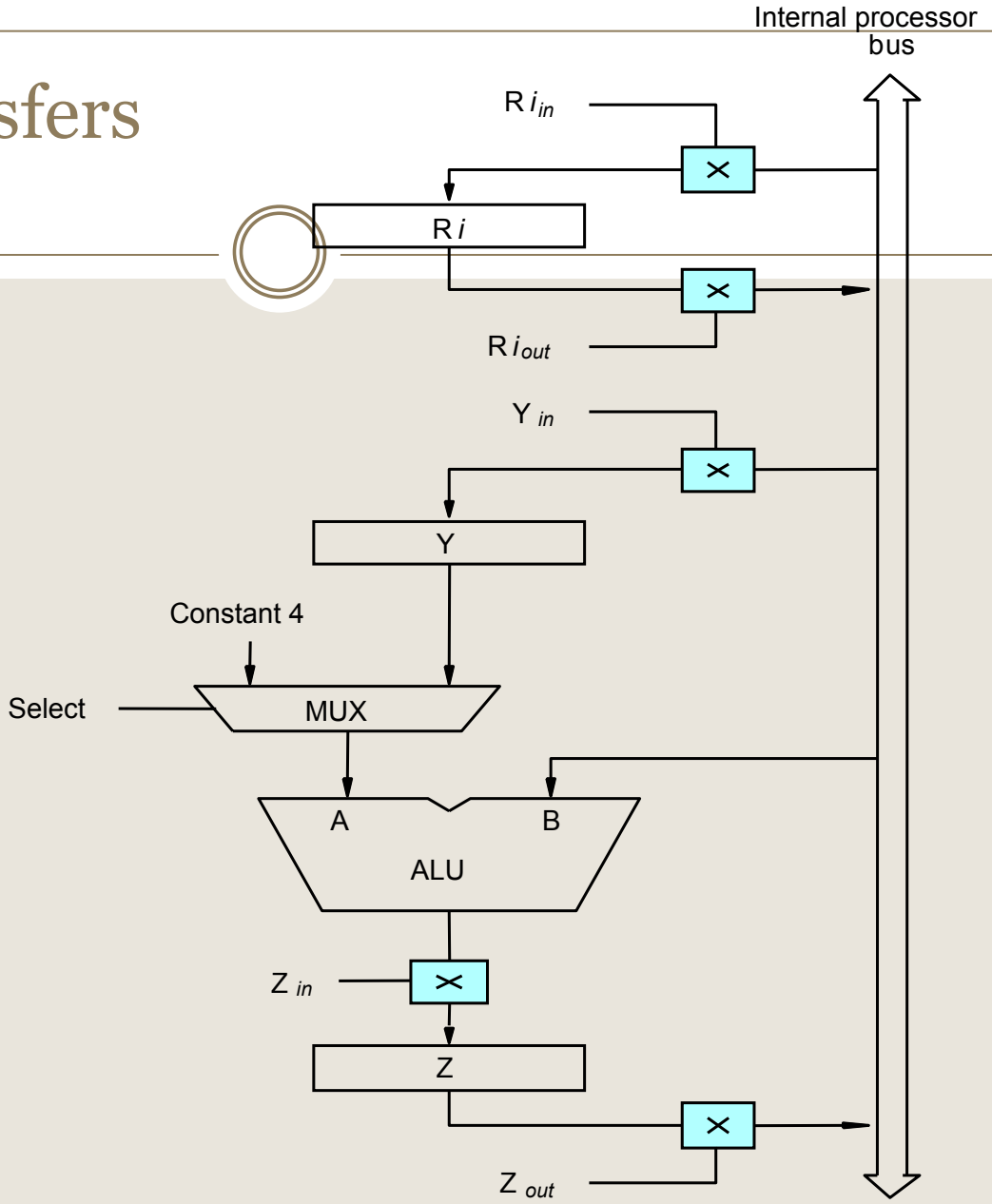



Figure 7.2. Input and output gating for the registers in Figure 7.1.

- The input and output gates for register Ri are controlled by signals i.e. $Ri_{in}$ and $Ri_{out}$ .
- When $Ri_{in}$ is set to1 – data available on internal processor bus are loaded into Ri.
- When $Ri_{iout}$ is **set to1** – the contents of register are placed internal processor bus .
- When $Ri_{iout}$ is set to 0 – the bus can be used for transferring data from other registers .

# Data transfer between two registers:

EX: Move R1,R4

Transfer the contents of R1 to R4.

1. Enable output of register R1 by setting R1out=1. This places the contents of R1 on internal processor bus .
2. Enable input of register R4 by setting R4in=1. This loads the data from internal processor bus into register R4.

# Register Transfers contd.

- All operations and data transfers are controlled by the processor clock.

Figure 7.3. Input and output gating for one register bit.

# Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?

ADD R1,R2,R3

Datapath

Step 1: Output of the register R1 and input of the register Y are enabled, causing the contents of R1 to be transferred to Y.

Step 2: The multiplexer's select signal is set to select Y causing the multiplexer to gate the contents of register Y to input A of the ALU.

Step 3: The contents of Z are transferred to the destination register R3.

In terms of signals

- **R1out, Yin**
- **Select Y ,R2out, Add, Zin**
- **Zout, R3in**

# Fetching a Word from Memory

- Move (R1), R2

*Address hold by R1 into MAR; issue Read operation; data into MDR; data from MDR to R2.*

*The actions needed to execute this instruction are:*

- MAR ← [R1]
- Start a Read operation on the memory bus
- Wait for the MFC(Memory-Function-Completed) response from the memory
- Data into MDR or Load MDR from the memory bus
- R2 ← [MDR]

## In terms of signals

- $R1_{out}$, $MAR_{in,Read}$
- $MDR_{inE}$, WMFC, $MDR_{In}$
- $MDR_{out}$, $R2_{In}$

# Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.

Figure 7.4. Connection and control signals for register MDR.

# Storing a word in memory

Example: *Move R2,(R1): Move the content of R2 to the memory location which is hold by reg. R1.*

*The actions needed to execute this instruction are:*

- Address is loaded into MAR
- Data which is to be written, will be loaded into MDR.
- Write command is issued.

In terms of signals

- $R1_{out}$,$MAR_{in}$
  $R2_{out}$,$MDR_{in}$,Write
  $MDR_{outE}$, WMFC

# Execution of a Complete Instruction

- Consider the instruction; Add (R3), R1

To execute the instruction following actions are required:

- Fetch the instruction or instruction code and store in IR.
- Fetch the first operand (the contents of the memory location is hold by reg. R3)
- Perform the addition
- Load the result into R1

# Execution of a Complete Instruction

Add (R3), R1

# Sequence of actions needed to fetch and execute the instruction:
# Add R3, R4, R5

| Step | Action |
|---|---|
| 1 | Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4 |
| 2 | Decode instruction, RA ← [R4], RB ← [R5] |
| 3 | RZ ← [RA] + [RB] |
| 4 | RY ← [RZ] |
| 5 | R3 ← [RY] |

# Load R5, X(R7)

| Step | Action |
|---|---|
| 1 | Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4 |
| 2 | Decode instruction, RA ← [R7] |
| 3 | RZ ← [RA] + Immediate value X |
| 4 | Memory address ← [RZ], Read memory, RY ← Memory data |
| 5 | R5 ← [RY] |

# Store R6, X(R8)

| Step | Action |
|---|---|
| 1 | Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4 |
| 2 | Decode instruction, RA ← [R8], RB ← [R6] |
| 3 | RZ ← [RA] + Immediate value X, RM ← [RB] |
| 4 | Memory address ← [RZ], Memory data ← [RM], Write memory |
| 5 | No action |

# Execution of Branch Instructions

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

Types of branch instructions;

- *Unconditional branch*
- *Conditional branch*

# Execution of Branch Instructions

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

Figure 7.7. Control sequence for an unconditional branch instruction.

# Execution of Conditional Branch Instruction

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

**Check the status of the condition code before loading a new value into PC.**

If N= 0 then

**Processor returns to step No. 1 immediately after step 4.**

**If N= 1**

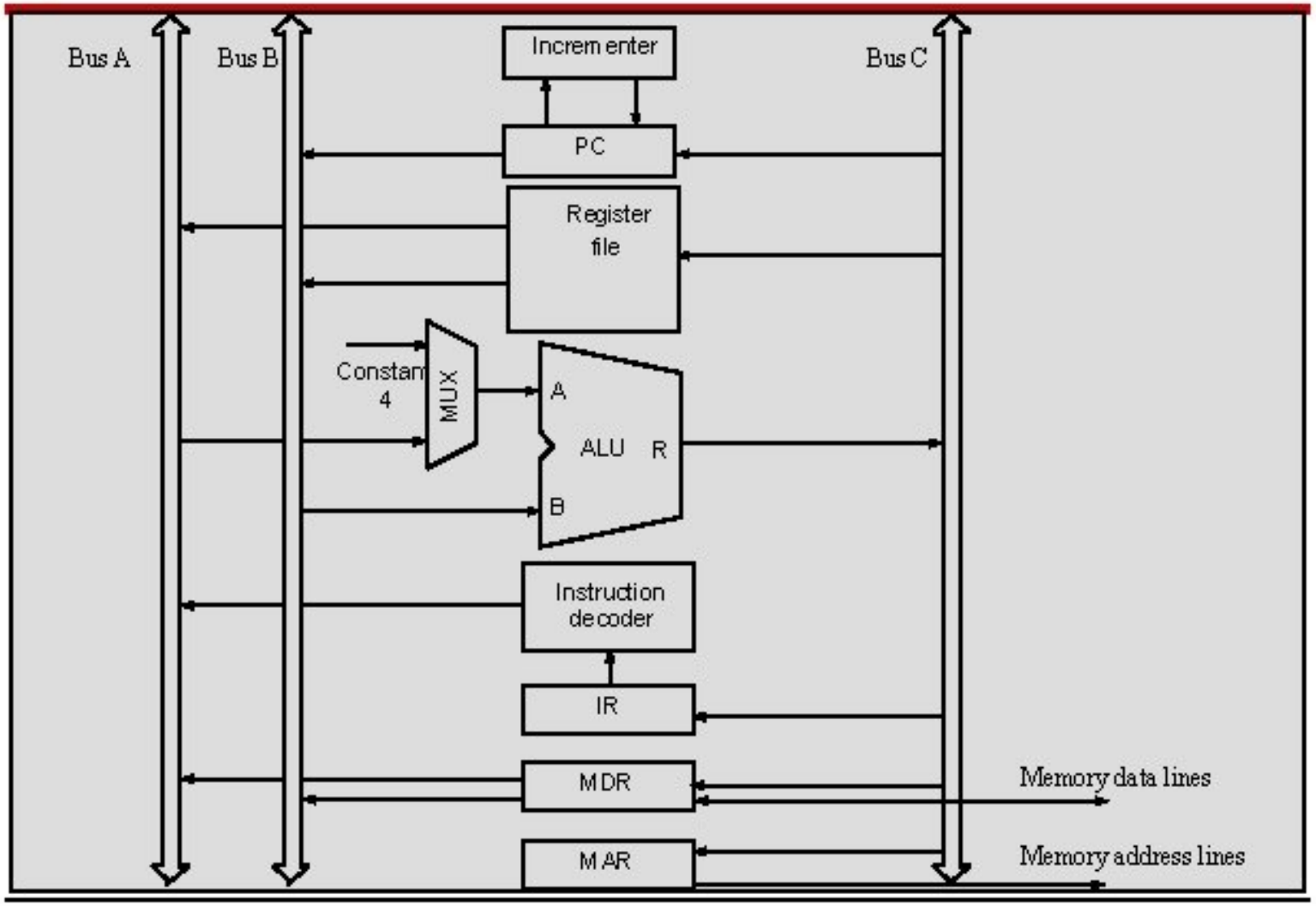
**Then step 5 is performed to load a new value into PC to start branch operation**

Control sequence for an conditional branch instruction.

# Multiple-Bus Organization

- Allow the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B.
- Allow the data on bus C to be loaded into a third register during the same clock cycle.
- It has an Incrementer unit to increment the content of PC by 4 to get the address of the next instruction.
- ALU simply passes one of its two input operands unmodified to bus C by enabling control signal: R=A or R=B.

# Multiple bus organization (contd..)



- General purpose registers are combined into a single block called register file. It has 3ports.
- 2 are output ports which are used to access two different registers and have their contents on buses A and B
- Third port allows data on bus c during same clock cycle.
- Bus A & B are used to transfer the source operands to A & B inputs of the ALU.
- ALU operation is performed.
- The result is transferred to the destination over the bus C.

- ALU may  simply pass one of its 2 input operands unmodified to bus C.
- The ALU control signals for such an operation R=A or R=B.
- Incrementer unit is used to increment the PC by 4.
- Using the incrementer eliminates the need to add the constant value 4 to the PC using the main ALU.
- The source for the constant 4 at the ALU input multiplexer can be used to increment other address such as load-multiple & store-multiple instructions.

# Multiple-Bus Organization

- Add R4, R5, R6

| Step | Action |
|---|---|
| 1 | $PC_{out}$, R=B, $MAR_{in}$, Read, IncPC |
| 2 | WMFC |
| 3 | $MDR_{outB}$, R=B, $IR_{in}$ |
| 4 | $R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End |

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6,
for the three-bus organization in Figure 7.8.

- Step 1:The contents of PC are passed through the ALU using R=B control signal & loaded into MAR to start a memory read operation.

  At the same time PC is incrementer by 4
- Step 2:The processor waits for MFC
- Step 3: Loads the data ,received into MDR ,then transfers them to IR.
- Step 4: The execution phase of the instruction requires only one control step to complete.

# TYPES OF CONTROL SIGNAL

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- **Two categories:**
  1. **Hardwired control**
  2. **Micro-programmed control**
     - Hardwired system can operate at high speed; but with little flexibility.

# Hardwired Control

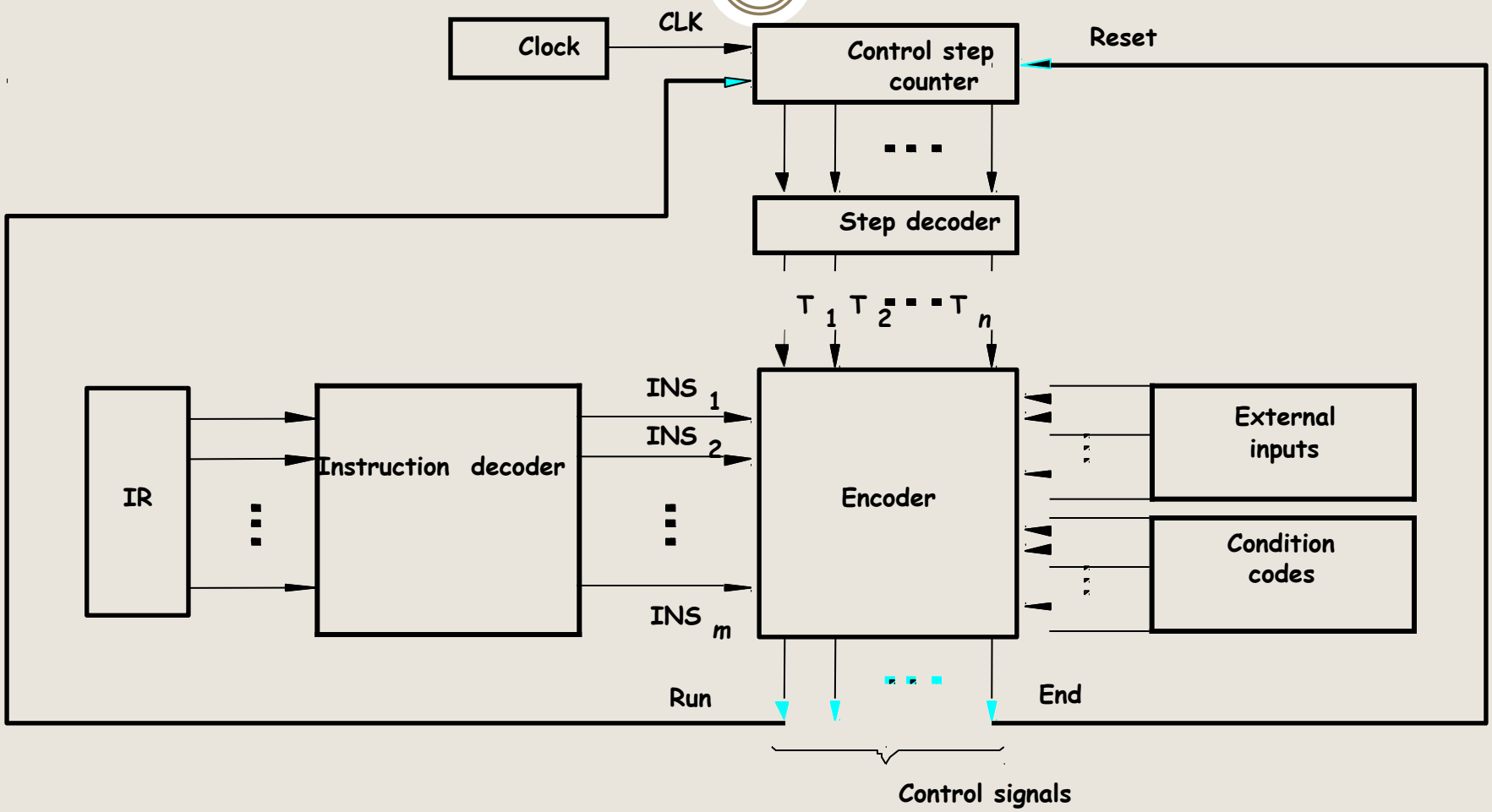
All time cycles are equal of equal duration & must be at least of such a duration which large enough to complete the specified action.

CU use a counter which is driven by a clock signal for each step being required for complete execution of an instruction.

Required control signal are uniquely determined by the following facts:-

1. Contains of the control step counter.
2. Contains of the instruction register.
3. Contains of the condition codes & status flags.
4. External input signal such as MFC & interrupt request.

# DETAILED BLOCK DESCRIPTION OF
## HARDWIRED CONTROL




Separation of the decoding and encoding functions.

The decoder / encoder block is a combinational circuit which generates required control signal depending on the state of all inputs.

Step decoder is placed at the control step counter to provide separate signal line for each step or clock cycle, in the control sequence.

Instruction decoder decodes each instruction to specify type of operation to be performed & maintains separate lines for each machine instruction.

# Microprogrammed Control

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

# MICROPROGRAMMED CONTROL

An alternative approach to hardwired control is micro-programmed control in which control signals are generated by a program.

Control Word (CW) :-

1. A word whose individual bits represents various control signals.
2. Each of control stepin control sequence of an instruction can be defined as control word.
3. A control word always unique combination of 0 & 1.
4. A sequence of control word corresponding to the sequence of control steps of an instruction is known as micro-routine for that instruction.
5. Individual control word in micro-routine is called as micro instruction.
6. Micro-routines corresponding to an instruction set, of a computer program are stored in a special memory called as micro-program memory/control store.

# MICROPROGRAMMED CONTROL

7. Control unit can generate control signals for any instruction by sequentially reading control word of the corresponding micro-routine from control store.
8. To read control word sequentially from control store, a micro-program counter [µPC] is used.
9. Each time a new instruction is loaded into IR, the output of the block labeled “Starting Address Generator” is loaded into the µPC.
10. The µPC is then automatically incremented by the clock, causing successive micro-instructions to b read from the control store.
11. Therefore the control signals are delivered to the various parts of the processor in the correct sequence.

# Basic organization of a micro-programmed control unit

- Control st

One function cannot be carried out by this simple organization.

# Microprogrammed Control

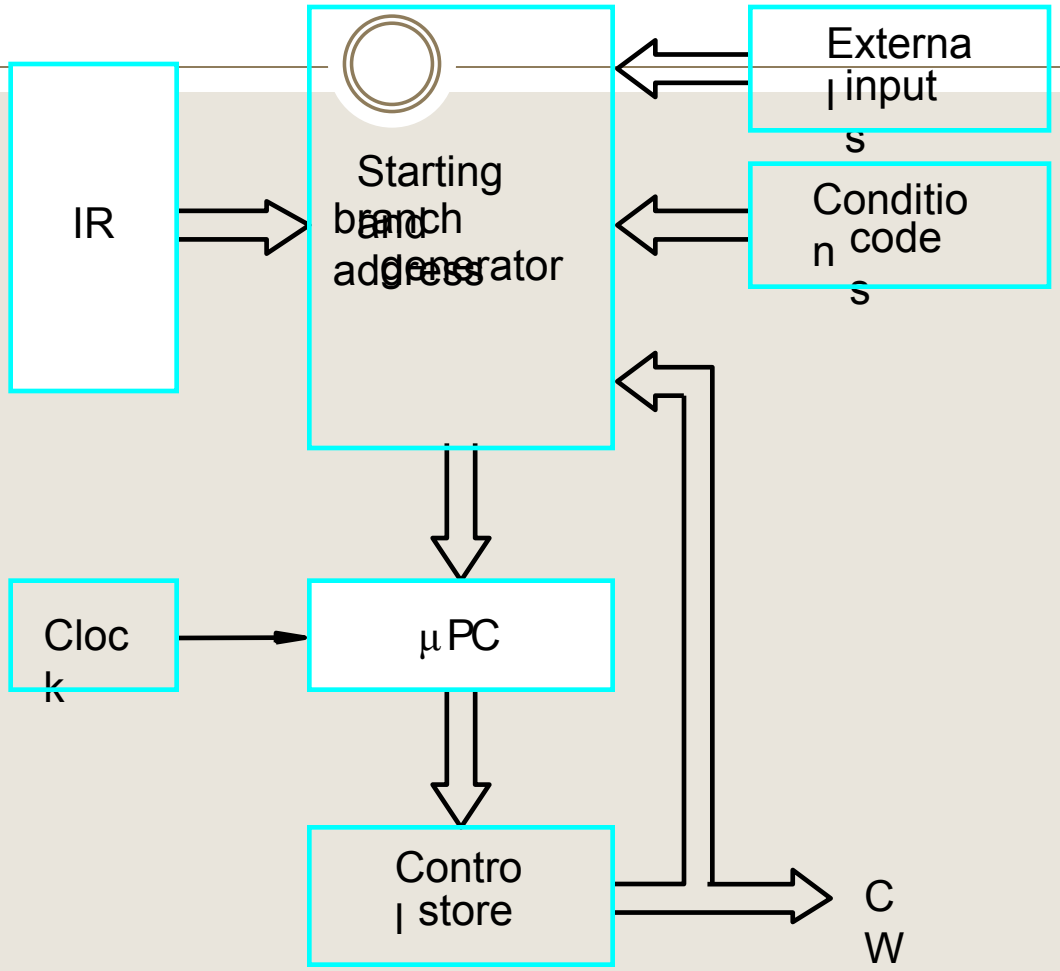
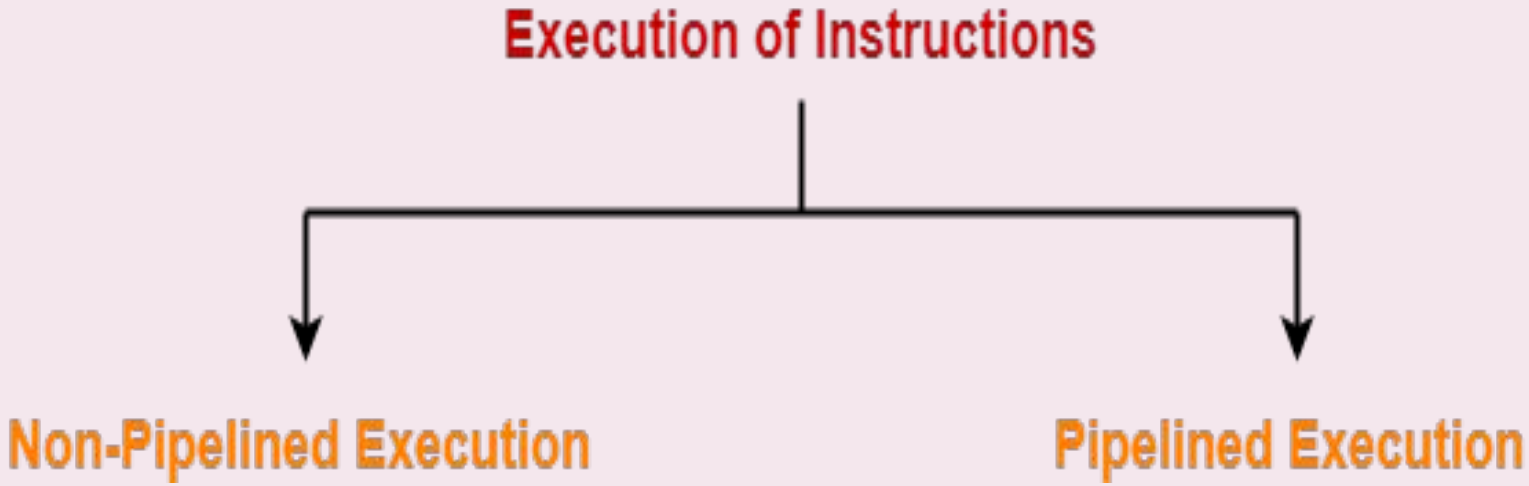


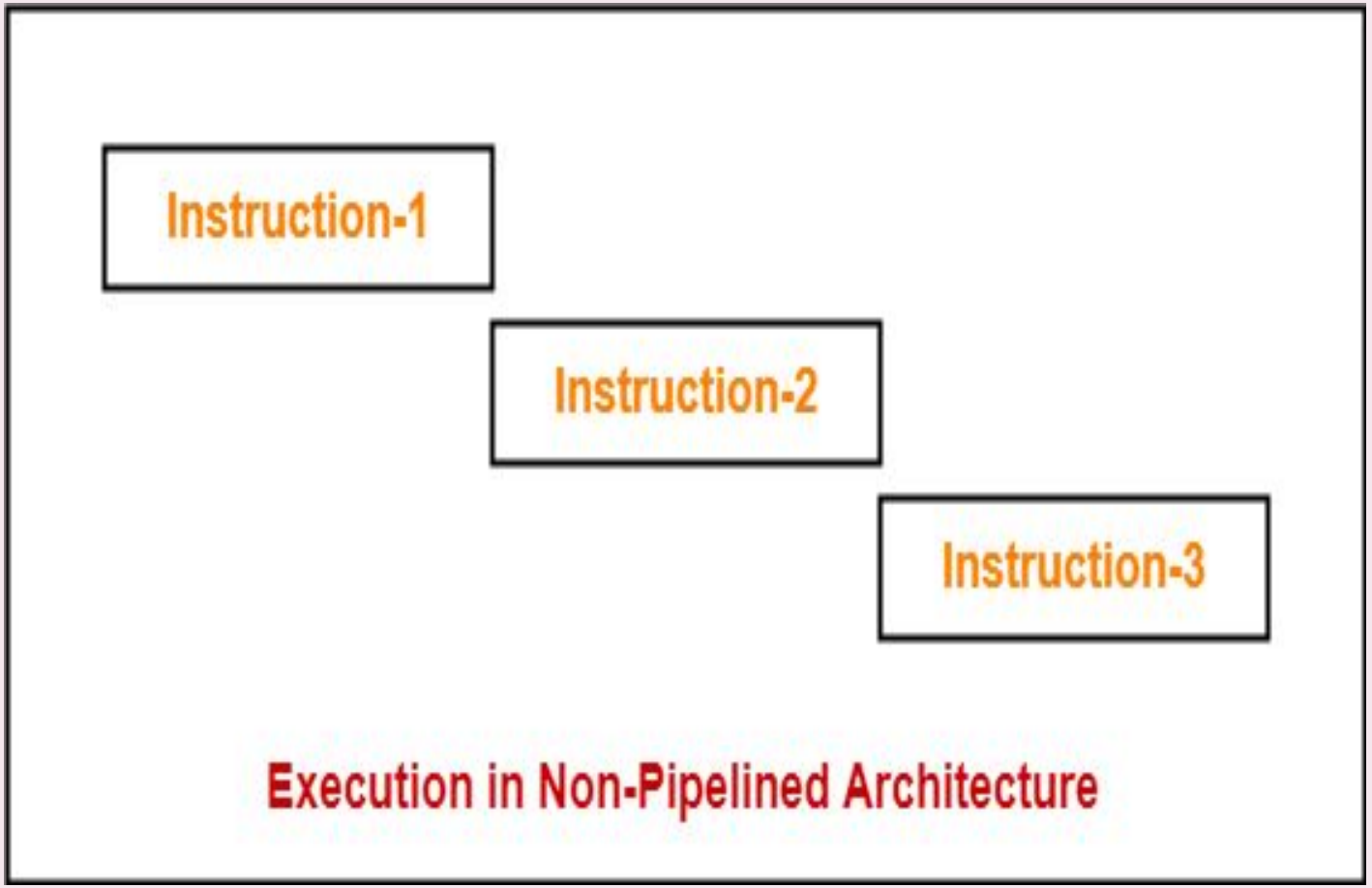

Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

# Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

# Pipelining


Execution of Instructions
Non-Pipelined Execution
Pipelined Execution


Instruction-1
Instruction-2
Instruction-3
Execution in Non-Pipelined Architecture

# To improve the performance of a CPU we have two options:

1) **Improve the hardware by introducing faster circuits.**

2) **Arrange the hardware such that more *than one operation can be performed at the same time*.**

- **Since, there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2nd option.**

# Pipelining

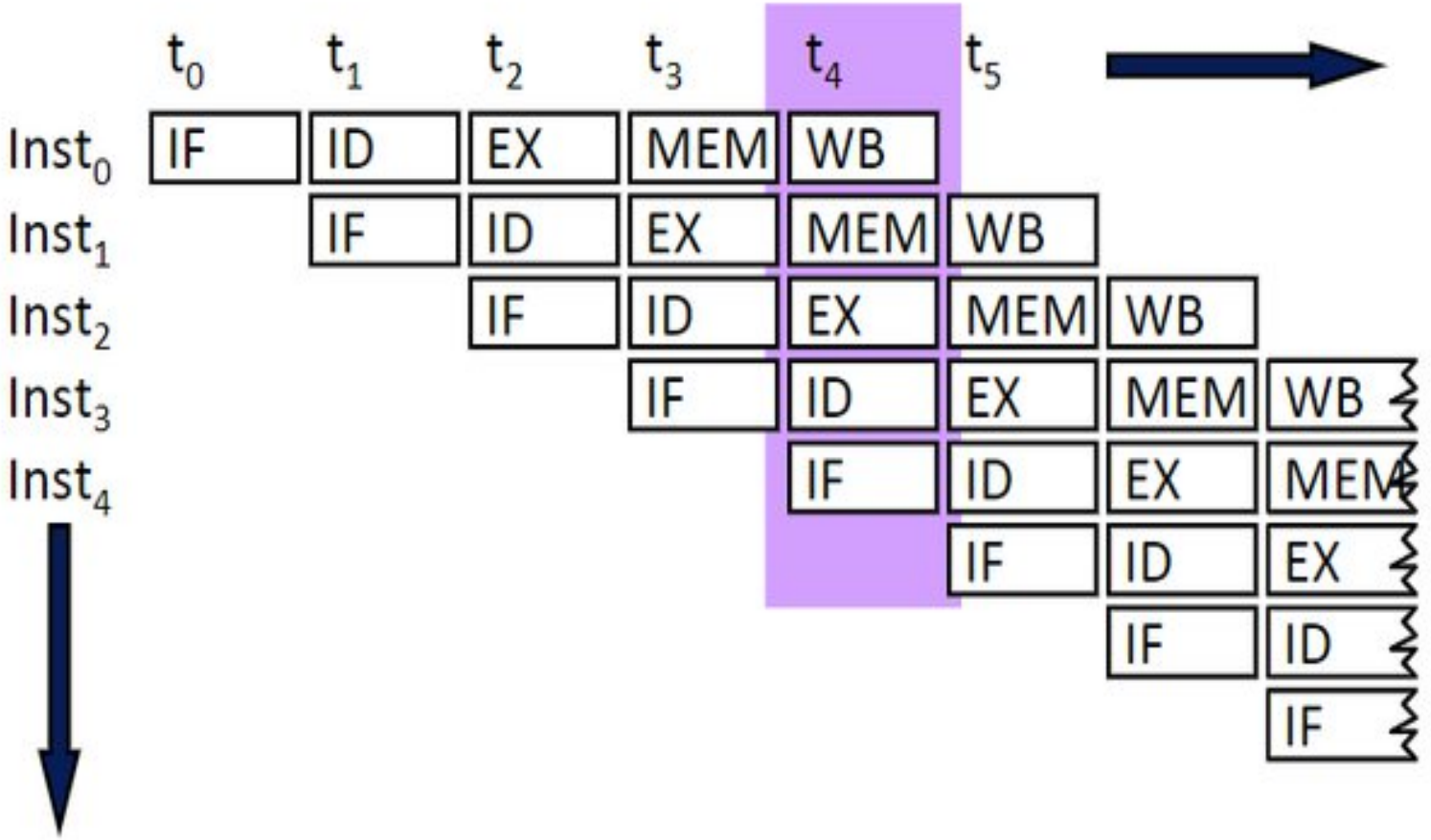
: Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.

- Pipelining is widely used in modern processors.
- Pipelining improves system performance.
- Pipelined organization requires sophisticated compilation techniques.

# Pipeline Stages

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.
- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

# Illustrating Pipeline Operation: Operation View



# Role of Cache Memory in pipelining

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

# Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

# Pipeline Performance

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall or Idle causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

# Execution in a pipelined processor

- Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:
- **Non overlapped execution:** **Overlapped execution**

| Stage / Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| S1 | $I_1$ | | | | $I_2$ | | | |
| S2 | | $I_1$ | | | | $I_2$ | | |
| S3 | | | $I_1$ | | | | $I_2$ | |
| S4 | | | | $I_1$ | | | | $I_2$ |

| Stage / Cycle | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| S1 | $I_1$ | $I_2$ | | | |
| S2 | | $I_1$ | $I_2$ | | |
| S3 | | | $I_1$ | $I_2$ | |
| S4 | | | | $I_1$ | $I_2$ |

# Performance of a pipelined processor

- Consider a ‘k’ segment pipeline with clock cycle time as ‘Tp’.
- Let there be ‘n’ tasks to be completed in the pipelined processor.
- Now, the first instruction is going to take ‘k’ cycles to come out of the pipeline but the other ‘n – 1’ instructions will take only ‘1’ cycle each,
- i.e, a total of ‘n – 1’ cycles. So, time taken to execute ‘n’ instructions in a pipelined processor:

- $ET_{pipeline} = k + n - 1$ cycles
- $= (k + n - 1)\ Tp$

- In the same case, for a non-pipelined processor, execution time of ‘n’ instructions will be:
- $ET_{non\text{-}pipeline} = n * k * Tp$

- So, speedup (S) of the pipelined processor over non-pipelined processor, when ‘n’ tasks are executed on the same processor is:
- S = Performance of pipelined processor / Performance of Non-pipelined processor
- $S = ET_{non\text{-}pipeline} / ET_{pipeline}$
- $\Rightarrow S = [n * k * Tp] / [(k + n - 1) * Tp]$
- $S = [n * k] / [k + n - 1]$

- When the number of tasks ‘n’ are significantly larger than k, that is, n >> k
- S = n * k / n
- S = k
- where ‘k’ are the number of stages in the pipeline.
- Also, **Efficiency** = Given speed up / Max speed up = S / $S_{max}$
- **Throughput** = Number of instructions / Total time to complete the instructions
- So, **Throughput** = n / (k + n – 1) * Tp

# Pipeline Hazards

- Hazards: situations that makes the pipeline to stall or idle.
- Where one instruction can not immediately follow another

## Types

1. Structural hazards
2. Control hazards
3. Data hazards

# Pipeline hazards

- Hazards reduce the performance from the ideal speedup gained by pipelines:
- **Structural hazard**: Resource conflict. Hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- **Data hazard**: When an instruction depends on the results of the previous instruction.
- **Control hazard**: Due to branches and other instructions that affect the PC.

# Structural dependency (Structural hazards)

- This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.
- In the below scenario, in cycle 4, instructions $I_1$ and $I_4$ are trying to access same resource (Memory) which introduces a resource conflict.

| Instruction / Cycle | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $I_1$ | IF(Mem) | ID | EX | Mem | |
| $I_2$ | | IF(Mem) | ID | EX | |
| $I_3$ | | | IF(Mem) | ID | EX |
| $I_4$ | | | | IF(Mem) | ID |

- **Solution for structural dependency:**
- **Renaming :** According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

| Instruction/ Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $I_1$ | IF(CM) | ID | EX | DM | WB | | |
| $I_2$ | | IF(CM) | ID | EX | DM | WB | |
| $I_3$ | | | IF(CM) | ID | EX | DM | WB |
| $I_4$ | | | | IF(CM) | ID | EX | DM |
| $I_5$ | | | | | IF(CM) | ID | EX |

# Control Hazards

- Can cause a greater performance loss than the data hazards
- When a branch is executed it may or it may not change the PC (to other value than its value + 4)
  - If a branch is changing the PC to its target address, than it is a ***taken*** branch
  - If a branch doesn't change the PC to its target address, than it is a ***not taken*** branch
- If instruction i is a taken branch, than the value of PC will not change until the end MEM stage of the instruction execution in the pipeline
  - A simple method to deal with branches is to stall the pipe as soon as we detect a branch until we know the result of the branch

# Control Dependency (Control Hazards)

- This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.
- Consider the following sequence of instructions in the program:
  100: $I_1$
  101: $I_2$ (JMP 250)
  102: $I_3$
  .
  .
  250: $BI_1$
- Expected output: $I_1$ -> $I_2$ -> $BI_1$

- Generally, the target address of the JMP instruction is known after ID stage only.
- Output Sequence: $I_1$ -> $I_2$ -> $I_3$ -> $BI_1$
- To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

| Instruction/ Cycle | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB | |
| $I_2$ | | IF | ID (PC:250) | EX | Mem | WB |
| $I_3$ | | | IF | ID | EX | Mem |
| $BI_1$ | | | | IF | ID | EX |

| Instruction/ Cycle | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB | |
| $I_2$ | | IF | ID (PC:250) | EX | Mem | WB |
| Delay | - | - | - | - | - | - |
| $BI_1$ | | | | IF | ID | EX |

Output Sequence: $I_1$ -> $I_2$ -> Delay (Stall) -> $BI_1$

- **Solution for Control dependency** Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this at 1st stage prediction is done about which branch will be taken. For branch prediction Branch penalty is zero.

# Data Dependency (Data Hazard)

- Example: Let there be two instructions I1 and I2 such that:
  I1 : ADD R1, R2, R3
  I2 : SUB R4, R1, R2
- When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that $I_2$ tries to read the data before $I_1$ writes it, therefore, $I_2$ incorrectly gets the old value from $I_1$.

| Instruction / Cycle | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $I_1$ | IF | ID | EX | DM |
| $I_2$ | | IF | ID (Old value) | EX |

- To minimize data dependency stalls in the pipeline, **operand forwarding** is used.
- **Operand Forwarding :** In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.

## There are mainly three types of data hazards:

- 1) **RAW** (Read after Write) [Flow/True data dependency]
  2) **WAR** (Write after Read) [Anti-Data dependency]
  3) **WAW** (Write after Write) [Output data dependency]
- Let there be two instructions I and J, such that J follow I. Then,
- **RAW** hazard occurs when instruction J tries to read data before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R4 <- R2 + R3
- **WAR** hazard occurs when instruction J tries to write data before instruction I reads it.
  Eg:
  I: R2 <- R1 + R3
  J: R3 <- R4 + R5
- **WAW** hazard occurs when instruction J tries to write output before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R2 <- R4 + R5

# Multiprocessor Architecture

- Most computer systems are single processor systems i.e. they only have one processor.
- However, multiprocessor or parallel systems are increasing in importance now a days.
- These systems have multiple processors working in parallel that share the computer clock, memory, bus, peripheral devices etc.

- An image demonstrating the multiprocessor architecture is:

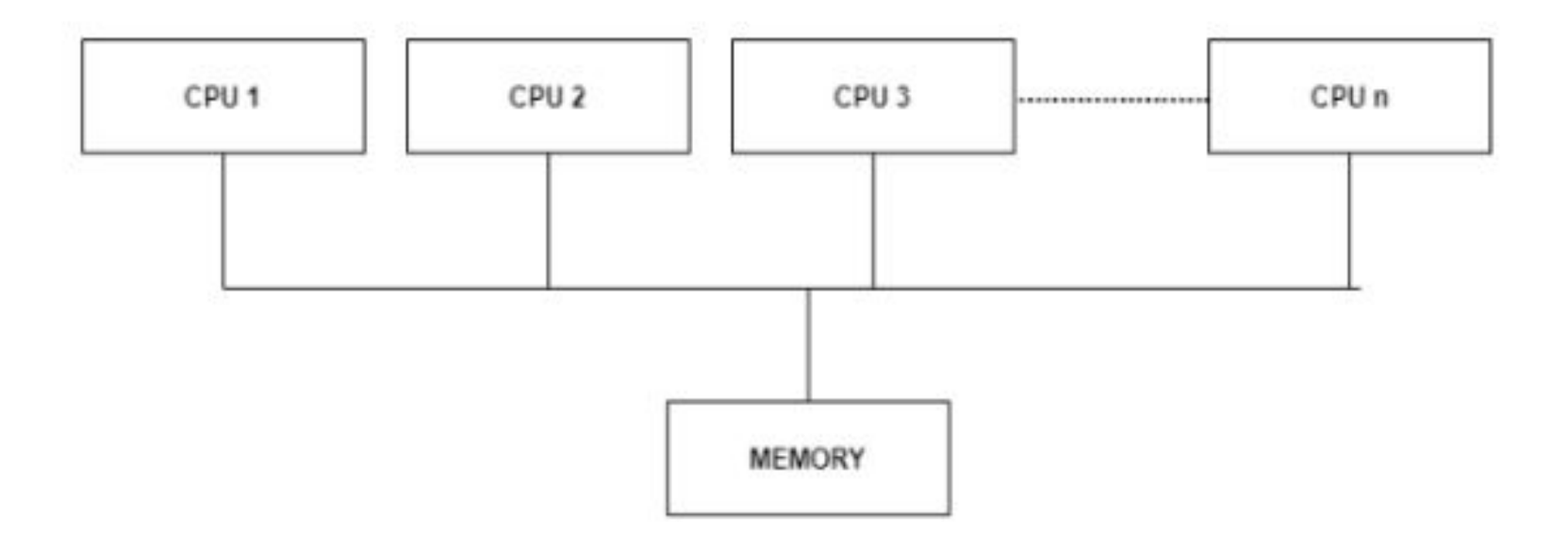


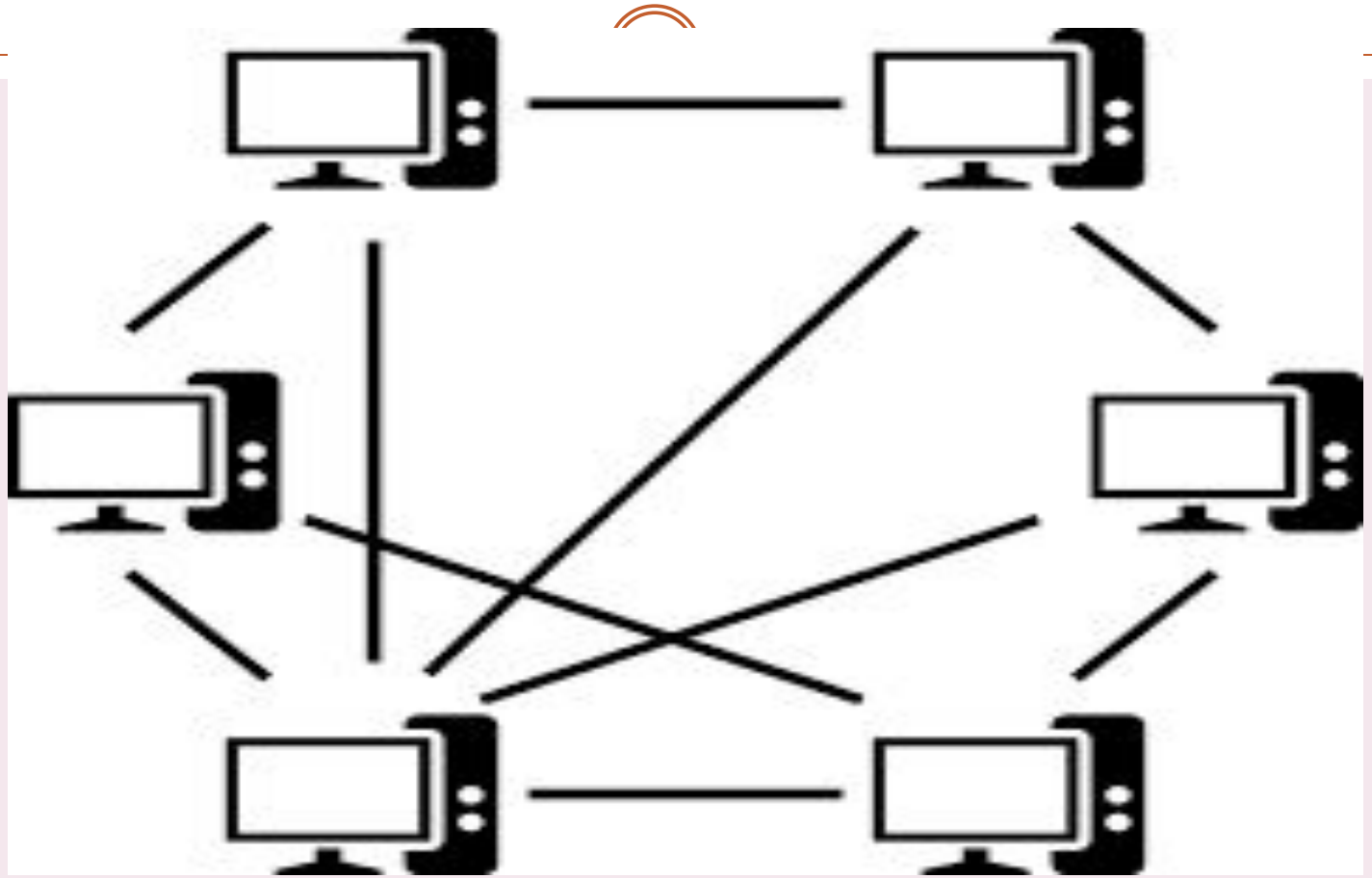

**Fig. Multiprocessor Architecture**

# Types of Multiprocessors

There are mainly two types of multiprocessors i.e. symmetric and asymmetric multiprocessors. Details about them are as follows −

## Symmetric Multiprocessors

In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer to peer relationship i.e. no master - slave relationship exists between them.

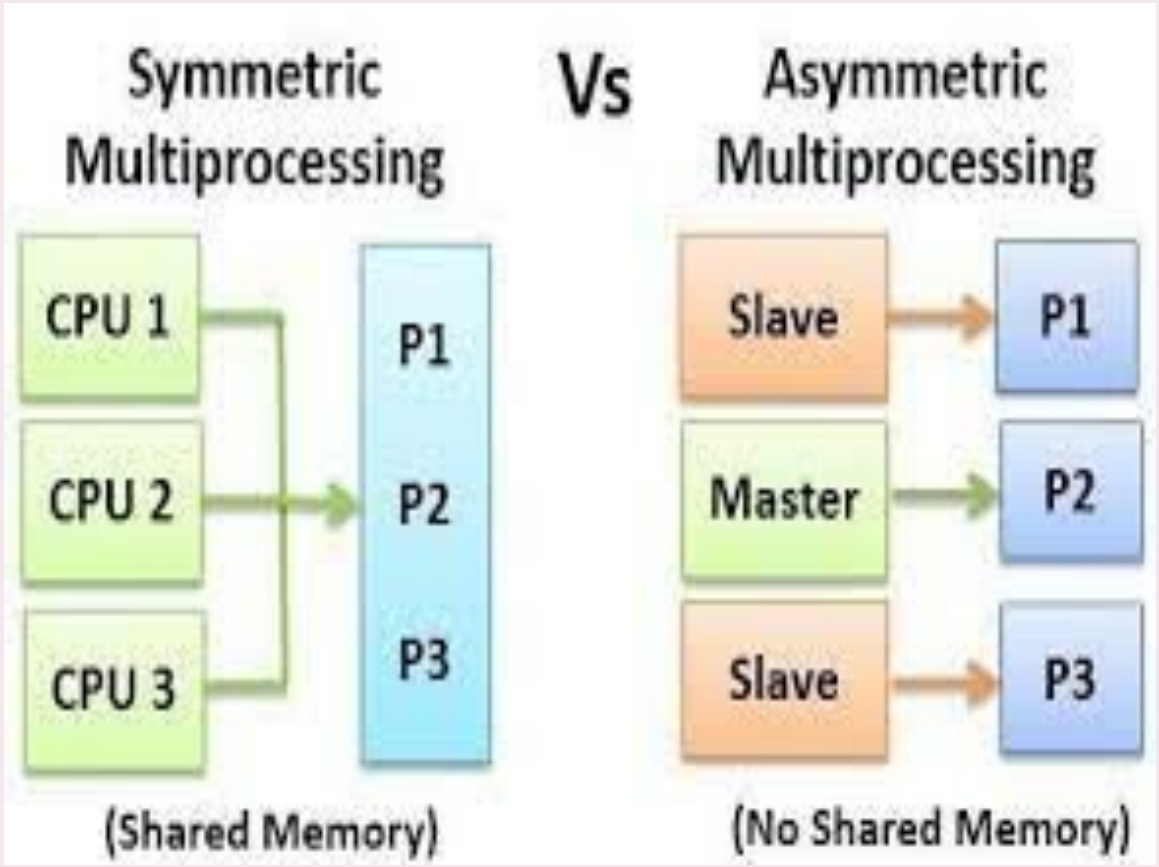


Peer to peer connection

# Asymmetric Multiprocessors

In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors.

Asymmetric multiprocessor system contains a master slave relationship.


Symmetric Multiprocessing
Vs
Asymmetric Multiprocessing
CPU 1
CPU 2
CPU 3
P1
P2
P3
Slave
Master
Slave
P1
P2
P3
(Shared Memory)
(No Shared Memory)

# Advantages of Multiprocessor Systems

There are multiple advantages to multiprocessor systems. Some of these are −

**More reliable Systems**

In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So **the system only becomes slower** and does not ground to a halt.

**Enhanced Throughput:** If multiple processors are working in tandem, then the throughput of the **system increases** i.e. number of processes getting executed per unit of time increase. If there are N processors then the throughput increases by an amount just under N.

# Disadvantages of Multiprocessor Systems

There are some disadvantages as well to multiprocessor systems. Some of these are:

**Increased Expense**

It is quite expensive. It is much cheaper to buy a simple single processor system than a multiprocessor system.

**Complicated Operating System Required:** There are multiple processors in a multiprocessor system that share peripherals, memory etc. So, it is much more complicated to schedule processes and impart resources to processes than in single processor systems. Hence, a more complex and complicated operating system is required in multiprocessor systems.

**Large Main Memory Required:** All the processors in the multiprocessor system share the memory. So a much larger pool of memory is required as compared to single processor systems.

In Multi-processor architecture, the shared memory is used where the entire memory can be shared by multiple processor.

In the shared-memory architecture, the entire memory, i.e., main memory and disks, is shared by all processors. A special, fast interconnection network (e.g., a high-speed bus or a cross-bar switch) allows any processor to access any part of the memory in parallel.

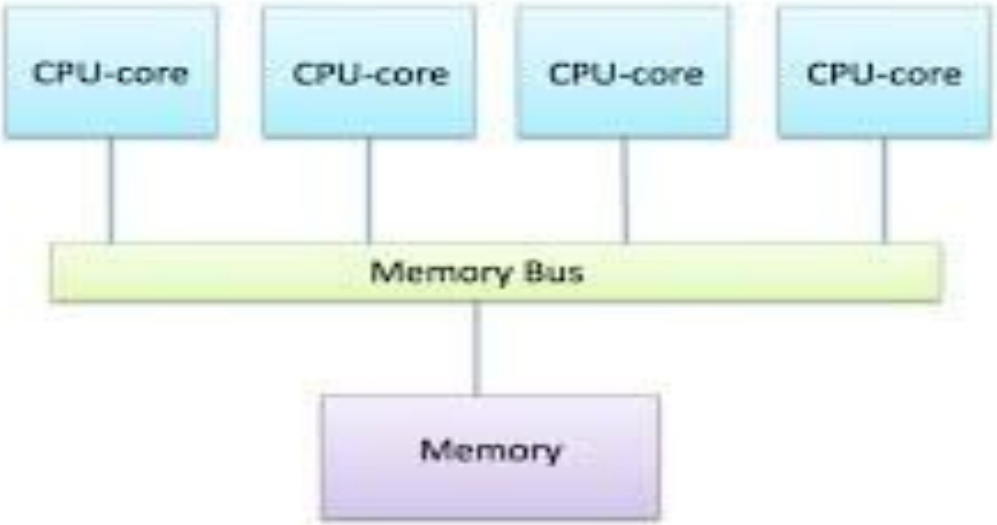



**Fig. Shared Memory**

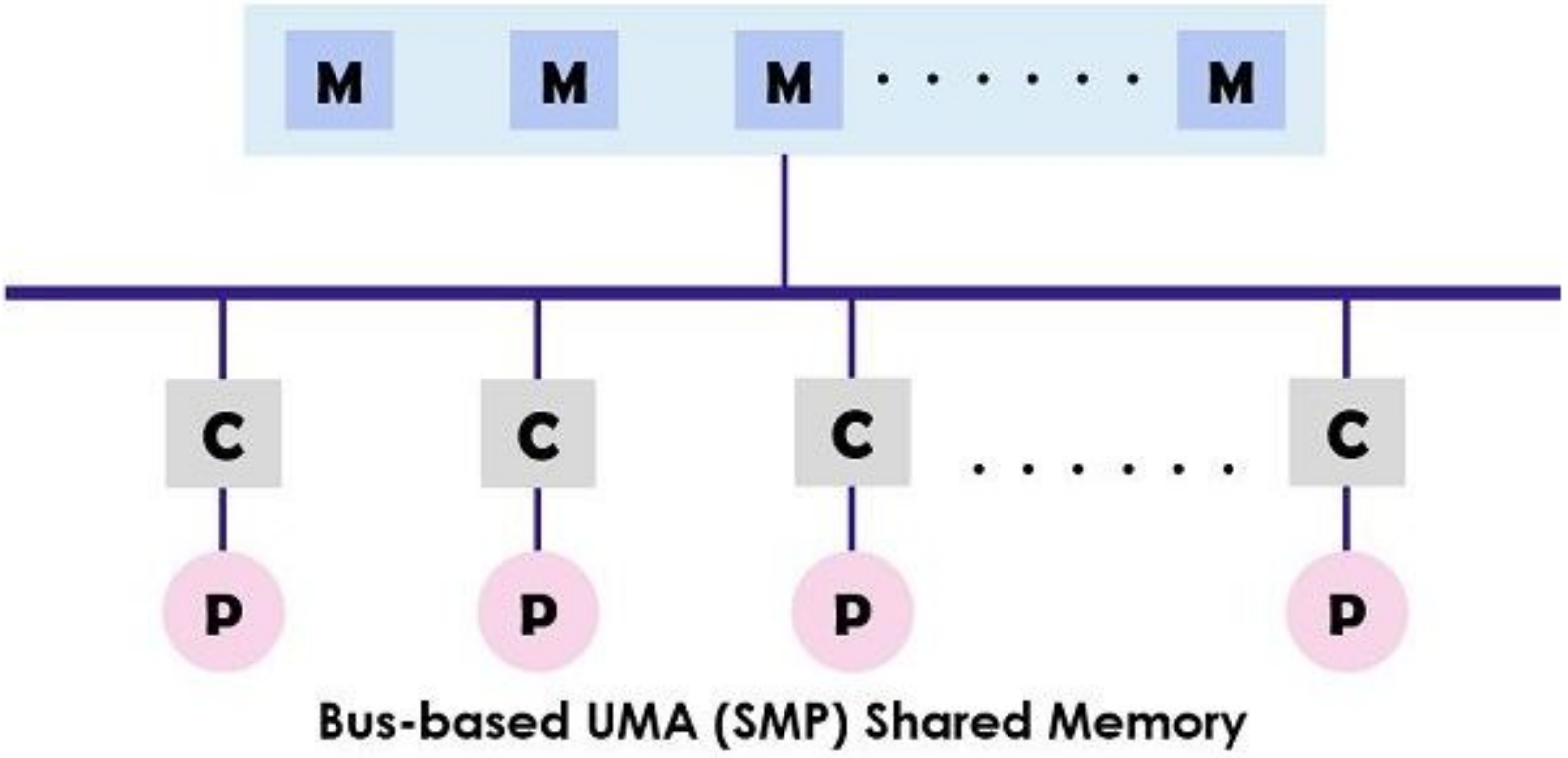
The two basic types of shared memory architectures are:

- **Uniform Memory Access (UMA)**
  **and**
- **Non-Uniform Memory Access (NUMA)**

# UMA (Uniform Memory Access)

system is a shared memory architecture for the multiprocessors. In this model, a single memory is used and accessed by all the processors present in the multiprocessor system with the help of the interconnection network. Each processor has equal memory accessing time and access speed.

- It can employ either of the single bus, multiple bus or crossbar switch. As it provides balanced shared memory access, it is also known as **SMP (Symmetric multiprocessor)** systems.




The typical design of the SMP is shown above where each processor is first connected to the cache then the cache is linked to the bus.

- At last the bus is connected to the memory.
- This UMA architecture reduces the contention for the bus through fetching the instructions directly from the individual isolated cache.
- It also provides an equal probability for reading and writing to each processor.
- The typical examples of the UMA model are Sun Starfire servers, Compaq alpha server.

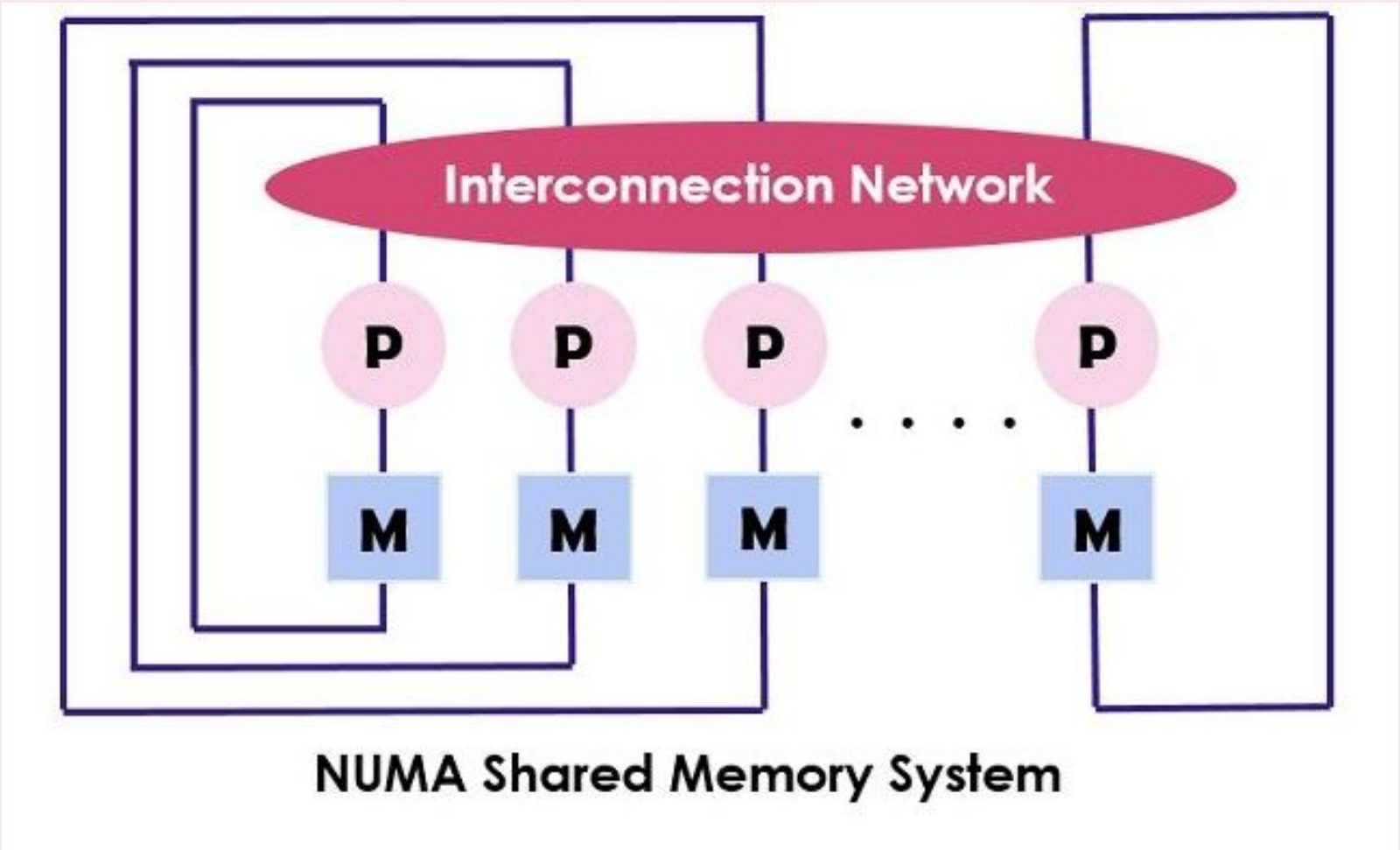
# NUMA (Non-uniform Memory Access)

**NUMA** is also a multiprocessor model in which each processor connected with the dedicated memory.

- However, these small parts of the memory combine to make a single address space.

- The main point here is that unlike UMA, the access time of the memory relies on the distance where the processor is placed which means varying memory access time.

- It allows access to any of the memory location by using the physical address.




NUMA Shared Memory System

- In non-uniform memory access, memory access time is not balanced or equal.
- There are two types of buses used in Non-uniform Memory Access, they include: Tree and Hierarchical.
- Non-uniform Memory Access is used in real-time applications and time-critical applications.
- Non-uniform Memory Access has more bandwidth when compared to Uniform Memory Access.
- Non-uniform Memory Access has multiple memory controllers.
- Non-uniform Memory Access is faster than uniform memory access.

# Difference between UMA and NUMA

| BASIS OF COMPARISON | UMA | NUMA |
|---|---|---|
| **Memory Access Time** | Memory access time is balanced or equal. | Memory access time is not balanced or equal. |
| **Types Of Buses** | There are three types of buses used in Uniform Memory Access, they include: single, multiple and Crossbar. | There are two types of buses used in Non-uniform Memory Access, they include: Tree and Hierarchical. |
| **Application** | Used in time-sharing applications and general purpose applications. | Used in real-time applications and time-critical applications. |
| **Bandwidth** | Has less bandwidth when compared to Non-uniform Memory Access. | Has more bandwidth when compared to Uniform Memory Access (UMA). |
| **Memory Controllers** | Has a single memory controller. | Has multiple memory controllers. |
| **Speed** | It is slower than Non-uniform Memory Access (NUMA). | It is faster than uniform memory access (UMA). |

# Flynn's classification

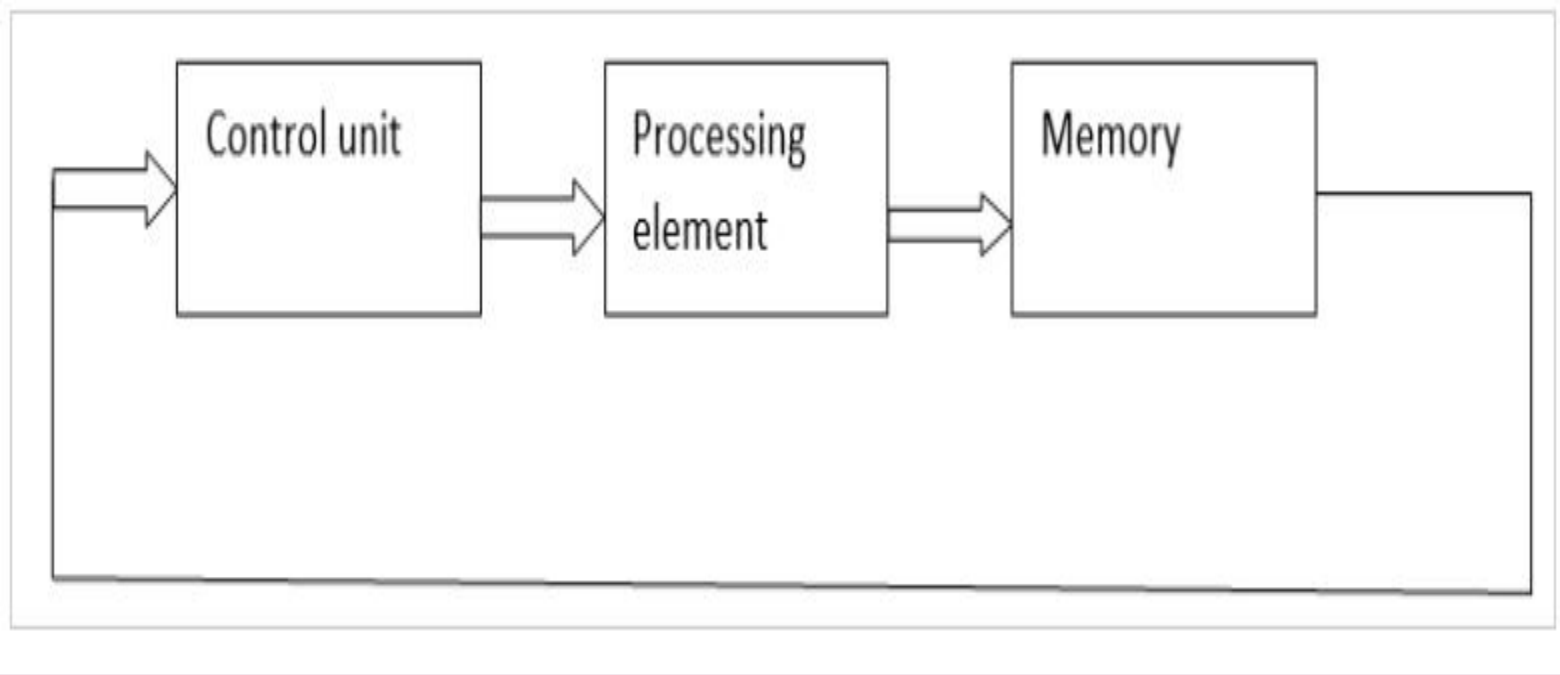
Flynn's classification divides computers into four major groups that are:

- **Single instruction stream, single data stream (SISD)**
- **Single instruction stream, multiple data stream (SIMD)**
- **Multiple instruction stream, single data stream (MISD)**
- **Multiple-instruction, multiple-data (MIMD)**

# 1) SISD (Single Instruction Single Data Stream)

**Single instruction:** Only one instruction stream is being acted or executed by CPU during one clock cycle.

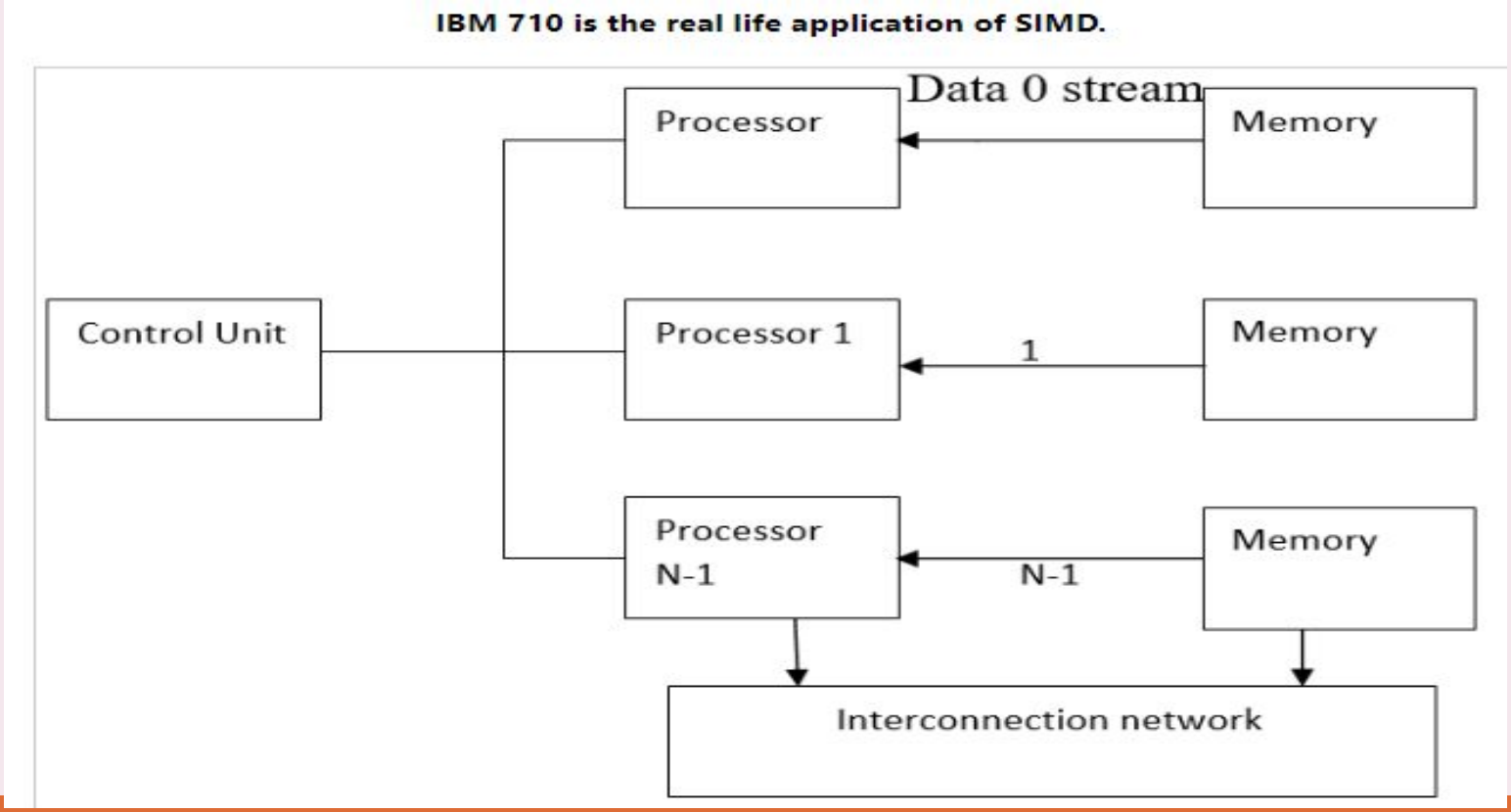
**Single data stream:** Only one data stream is used as input during one clock cycle.



- A SISD computing system is a uniprocessor machine that is capable of executing a single instruction operating on a single data stream.
- Most conventional computers have SISD architecture where all the instruction and data to be processed have to be stored in primary memory.

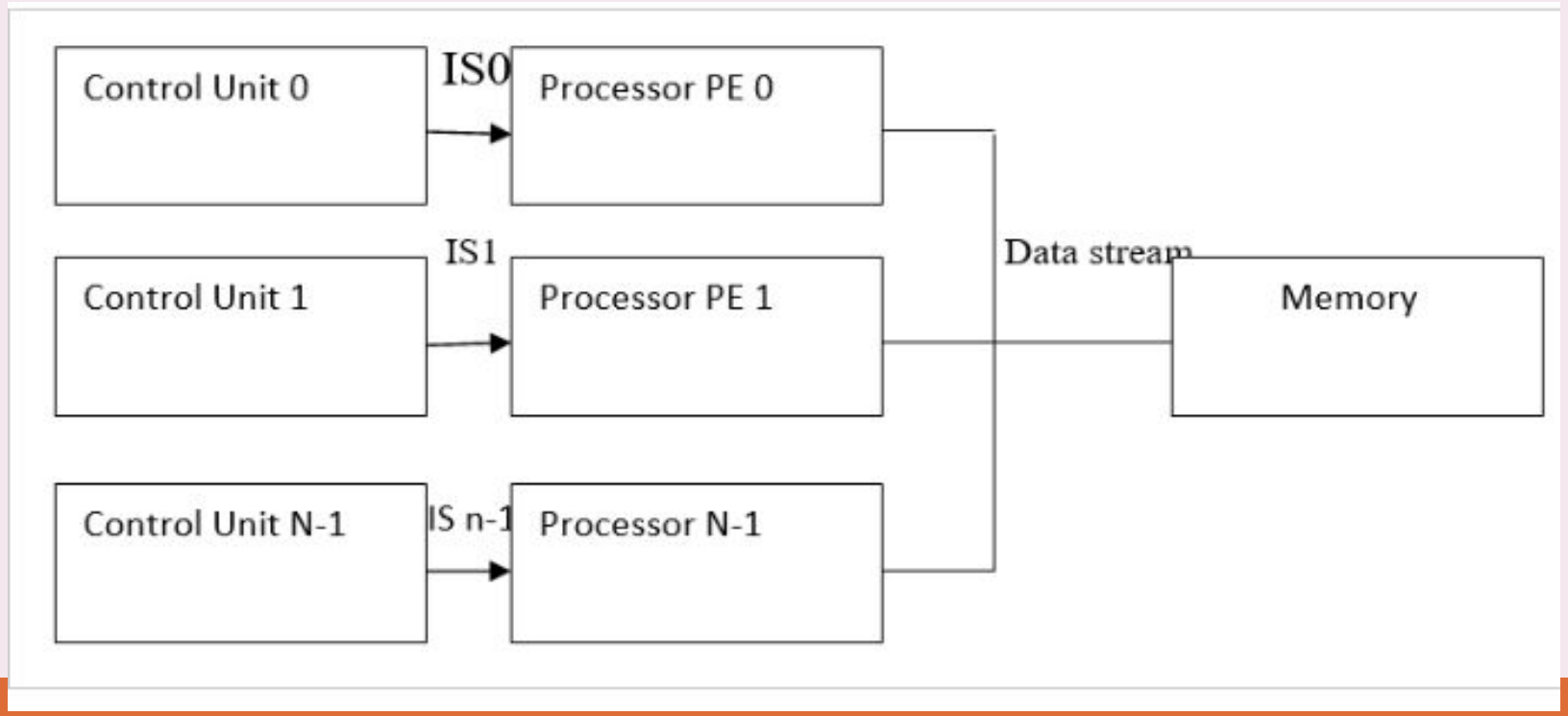
# 2) SIMD (Single Instruction Multiple Data Stream)

A SIMD system is a multiprocessor machine, capable of executing the same instruction on all the CPUs but operating on the different data stream.



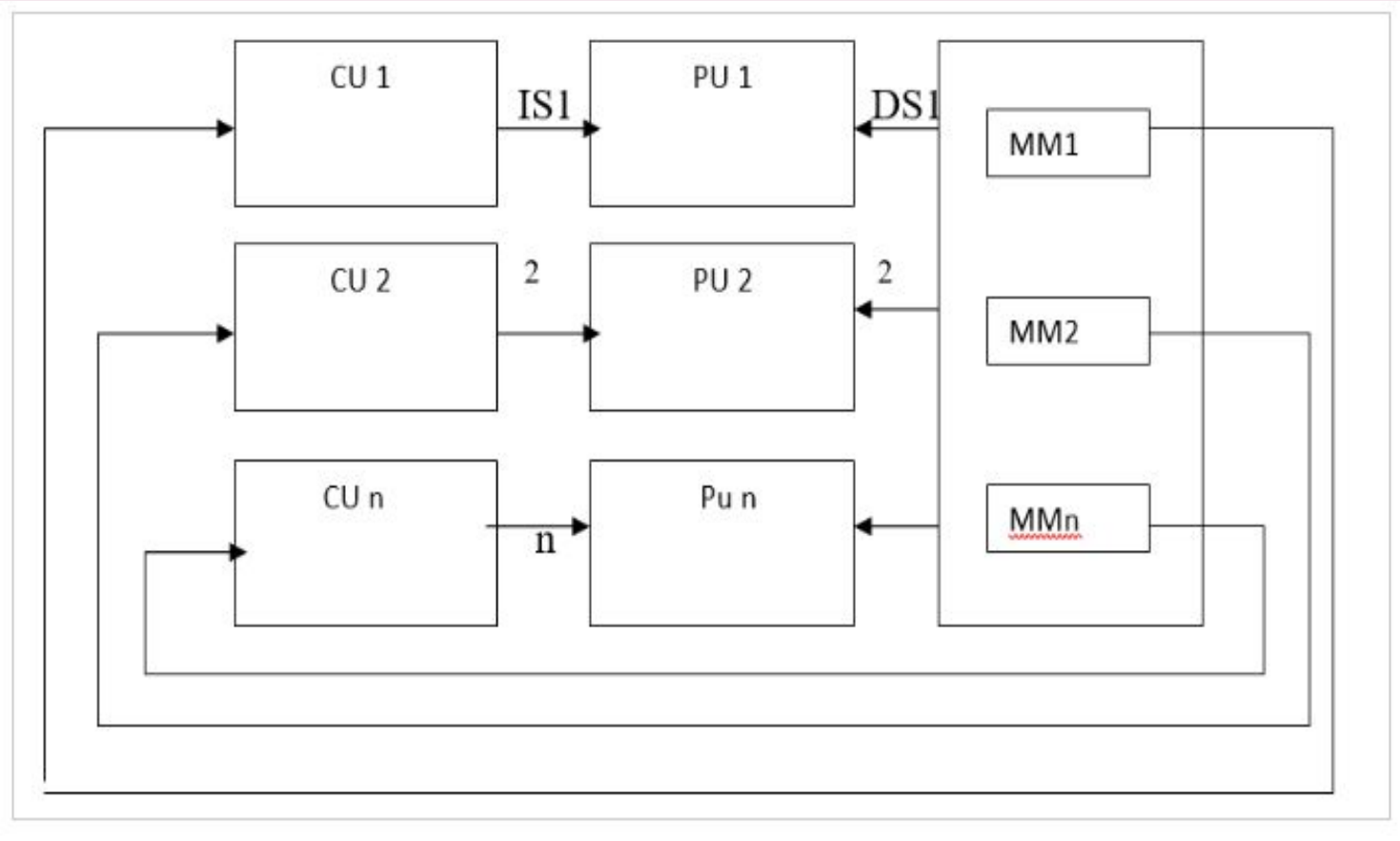
# 3) MISD (Multiple Instruction Single Data stream)

An MISD computing is a multiprocessor machine capable of executing different instructions on processing elements but all of them operating on the same data set.



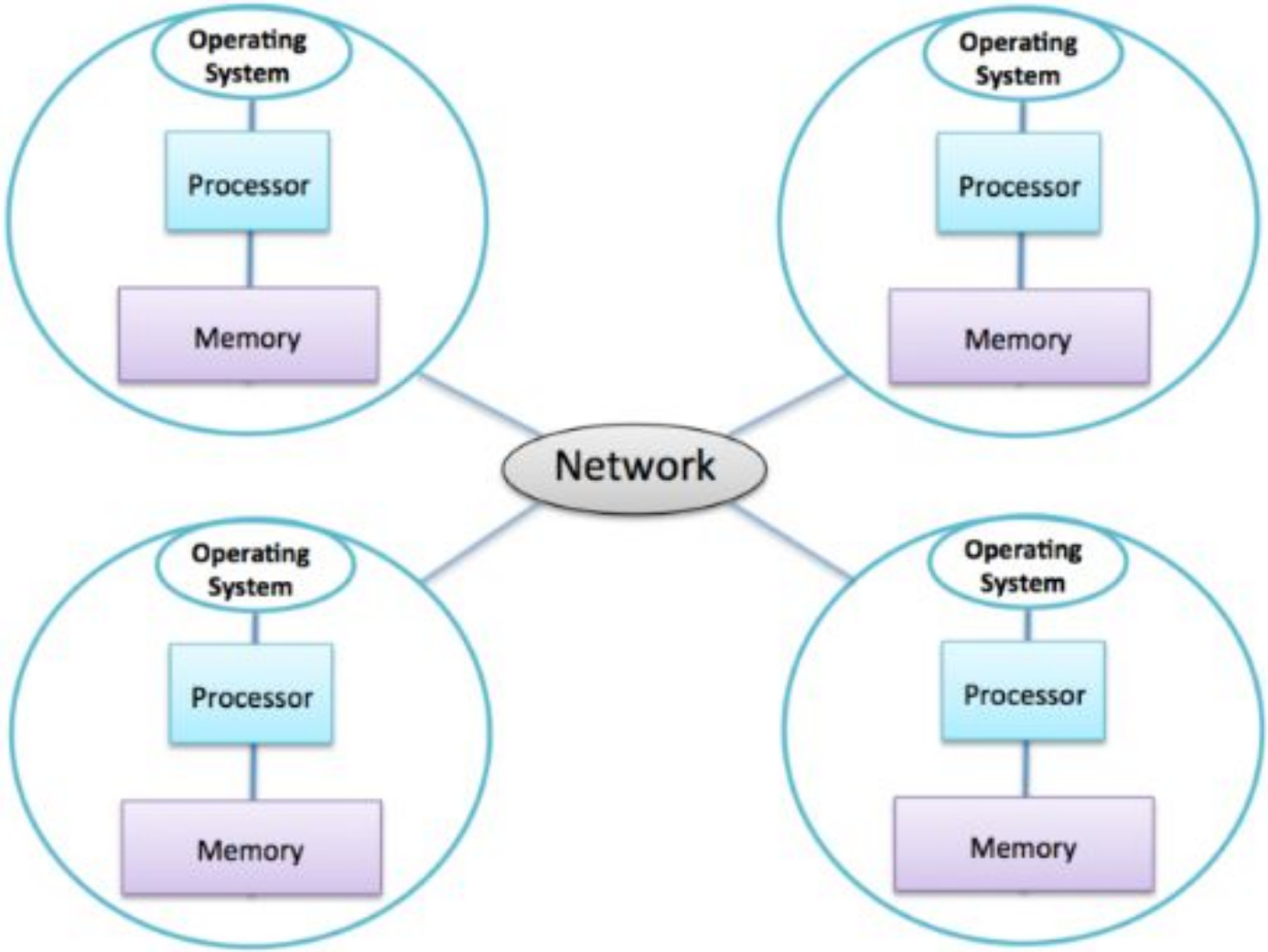
# 4) MIMD (Multiple Instruction Multiple Data Stream)

A MIMD system is a multiprocessor machine that is capable of executing multiple instructions over multiple data streams. Each processing element has a separate instruction stream and data stream.



# Distributed Memory Architecture

Distributed memory architecture is used by all large supercomputers. Because of the difficulty of having very large numbers of CPU-cores in a single shared-memory computer, all of today's supercomputers use the same basic approach to build a very large system: take lots of separate computers and connect them together with a fast network.


Operating System
Processor
Memory
Operating System
Processor
Memory
Network
Operating System
Processor
Memory
Operating System
Processor
Memory

## The most important points are:

- every separate computer is usually called *a node*
- each node has its own memory, totally separate from all the other nodes
- each node runs a separate copy of the operating system
- the only way that two nodes can interact with each other is by communication over the network.

In distributed memory architecture, each processor has its memory location.

Each processor has no explicit knowledge about other processor's memory.

For data to be transmitted, it should be shared from one processor to another as a message.

It is not economically possible to connect multiple processors directly to each other.

The office analogy is very useful here: a distributed-memory parallel computer is like workers all in separate offices, each with their own personal whiteboard, who can only communicate by phoning each other.

| Advantages | Disadvantages |
| --- | --- |
| the number of whiteboards (i.e. the total memory) grows as we add more offices | if we have large amounts of data, we have to decide how to split it up across all the different offices |
| there is no overcrowding so every worker has easy access to a whiteboard | we need to have lots of separate copies of the operating system |
| we can, in principle, add as many workers as we want provided the telephone network can cope. | it is more difficult to communicate with each other as you cannot see each others whiteboards so you have to make a phone call |

- The second disadvantage can be a pain when we do software updates – we have to upgrade thousands of copies of the OS!
- However, it doesn't have any direct cost implications as almost all supercomputers use some version of the Linux OS which is free.
- It turns out that it is much easier to build networks that can connect large numbers of computers together than it is to have large numbers of CPU-cores in a single shared-memory computer.
- This means it is relatively straightforward to build very large supercomputers – it is an engineering challenge, but a challenge that the computer engineers seem to be very good at tackling!

# Vector Processors

A **vector processor** is designed for vector computations. A vector is an array of operands of the same type. Consider the following vectors:

Vector A $(a_1, a_2, a_3, \ldots\ldots, a_n)$

Vector B $(b_1, b_2, b_3, \ldots\ldots, b_n)$

Vector C = Vector A + Vector B

$= C(c_1, c_2, c_3, \ldots\ldots, c_n)$, where $c_1 = a_1 + b_1$, $c_2 = a_2 + b_2$, $\ldots\ldots, C_n = a_n + b_n$

A vector processor adds all the elements of vector A and Vector B using a single vector instruction with **hardware approach**.

**Examples of vector processors are:**

- DEC's VAX 9000,
- IBM 390/VF,
- CRAY Research Y-MP family,
- Hitachi's S-810/20, etc.

## Vector Processing Applications

- Problems that can be efficiently formulated in terms of vectors
  - Long-range weather forecasting
  - Petroleum explorations
  - Seismic data analysis
  - Medical diagnosis
  - Aerodynamics and space flight simulations
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing

## Vector Processor (computer)

Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers

Vector Processors may also be pipelined

# Array Processor

- An array processor is a processor that performs computations on large arrays of data.
- The term is used to refer to two different types of processors.
  - An attached array processor
  - An SIMD array processor

# Attached Array Processors

- An attached array processor is a processor which is attached to a general purpose computer and its purpose is to enhance and improve the performance of that computer in numerical computational tasks.
- It achieves high performance by means of parallel processing with multiple functional units.
- To improve the performance of the host computer in numerical computational tasks auxiliary processor is attached to it.

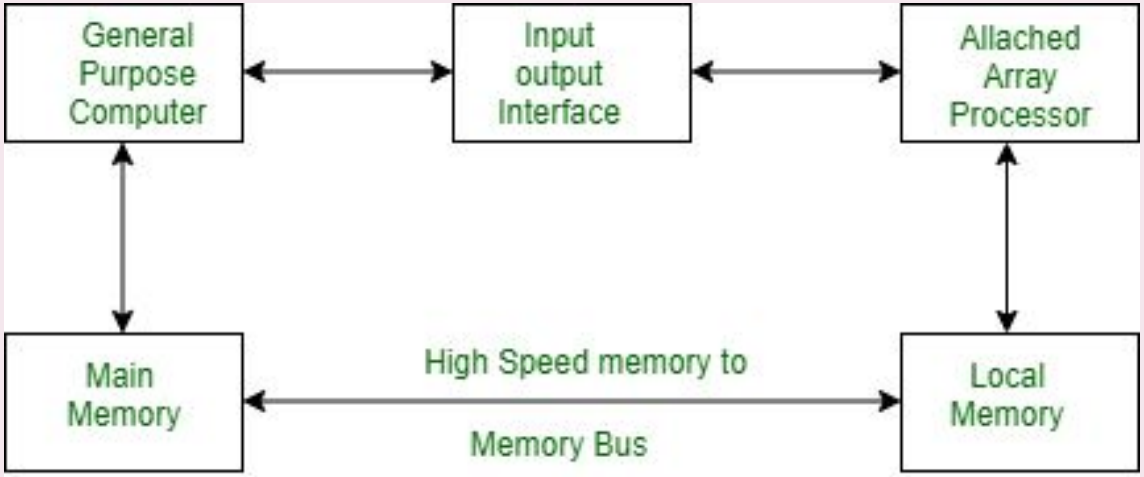
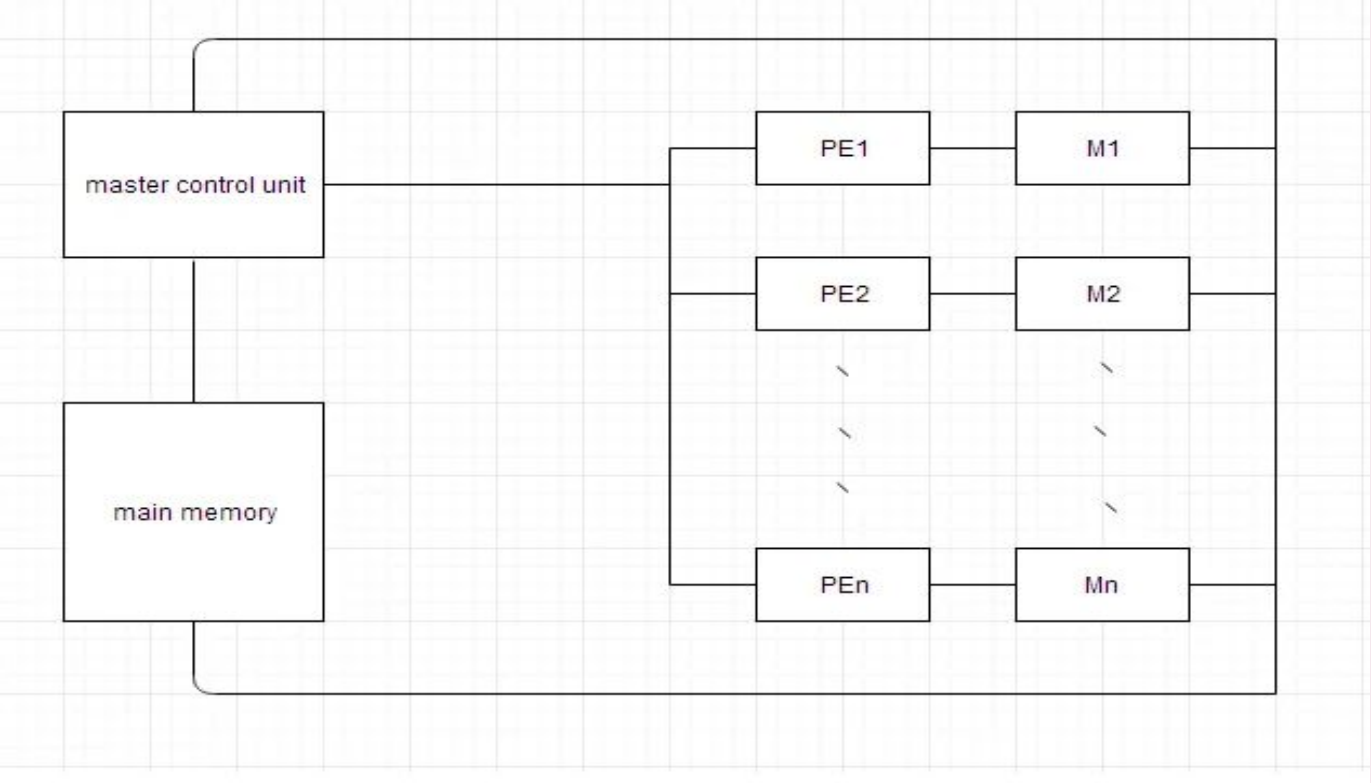



Figure - Interconnection of an attached array Processor to a host computer

# SIMD Array Processors

- SIMD is the organization of a single computer containing multiple processors operating in parallel.
- The processing units are made to operate under the control of a common control unit, thus providing a single instruction stream and multiple data streams.
- A general block diagram of an array processor is shown below. It contains a set of identical processing elements (PE's), each of which is having a local memory M.
- Each processor element includes an ALU and registers.
- The master control unit controls all the operations of the processor elements. It also decodes the instructions and determines how the instruction is to be executed.

- The main memory is used for storing the program. The control unit is responsible for fetching the instructions.
- Vector instructions are send to all PE's simultaneously and results are returned to the memory.



## Why use the Array Processor?

- Array processors increases the overall instruction processing speed.
- As most of the Array processors operates asynchronously from the host CPU, hence it improves the overall capacity of the system.
- Array Processors has its own local memory, hence providing extra memory for systems with low memory.

## Advantages of Vector:

- Size of the vector can be changed or Vector size can increase
- Multiple objects can be stored
- Elements can be deleted from a vector

## Disadvantages of Vector:

- A vector is an object, memory consumption is more.

### Advantages of Arrays:

- It is easy to sort an array.
- They are more appropriate for storing fixed number of elements

### Disadvantages of Arrays:

Can not modify or increase the size of the array

- Elements can not be deleted
- Dynamic creation of arrays is not possible
- Multiple data types can not be stored

# Short Type

- Define pipelining? Give an example of pipeline execution,.
- Draw the block diagram of hardwired control unit.
- Give the various types of multiprocessor environment based on Flynn's classification.

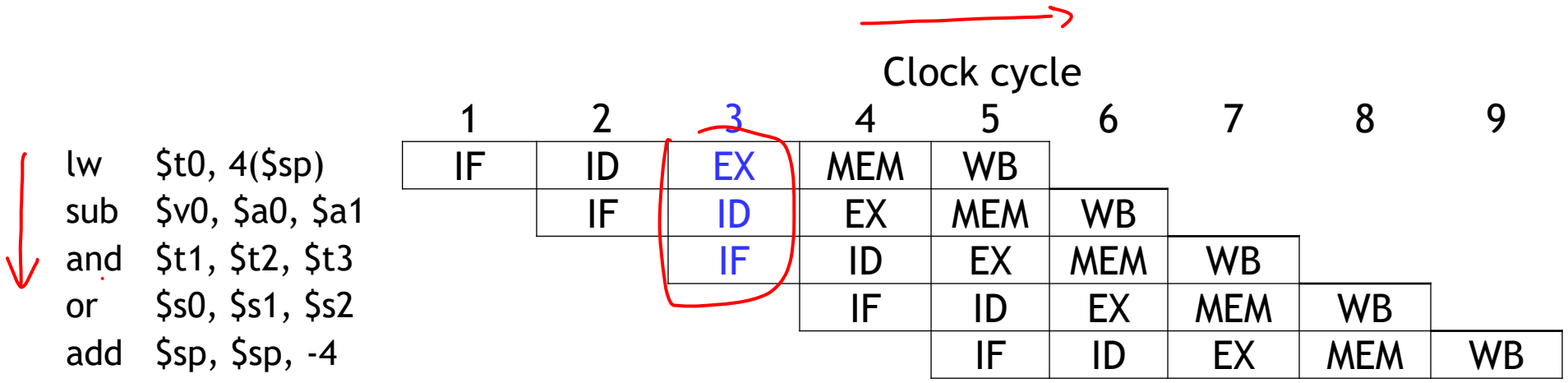
- Illustrate the block diagram of a PE in Illiac machine.
- Write any to implementation of array processor.
- Explain single bus data path. x
- Write the difference between horizontal and vertical microprogrammed control.
- Write the techniques to avoid instruction hazard in conditional branch instruction.
- Write the techniques to avoid instruction hazard in unconditional branch instruction.
- Write the techniques to avoid data hazard.

# Long Type

- Write the difference between hardwired and micro-programmed control unit.
- Define pipeline architecture of a system.
- What are different hazards occurring in pipeline architecture.
- Define data hazard occur in a system.
- What are Flynn's classification of a computer system?
- Differentiate UMA and NUMA architecture.
- How does multiprocessor system improve the performance of a system?
- What is SIMD processor? write one example of it.
- Write down the advantages of array processor.
- How can data hazard beavoided in programming?
- Explain with neat diagram single bus data-path in a processor.
- Write the control sequence step for the following instructions,
  - ADD (R3), R1
  - MUL (R1), R2, (R3)
-

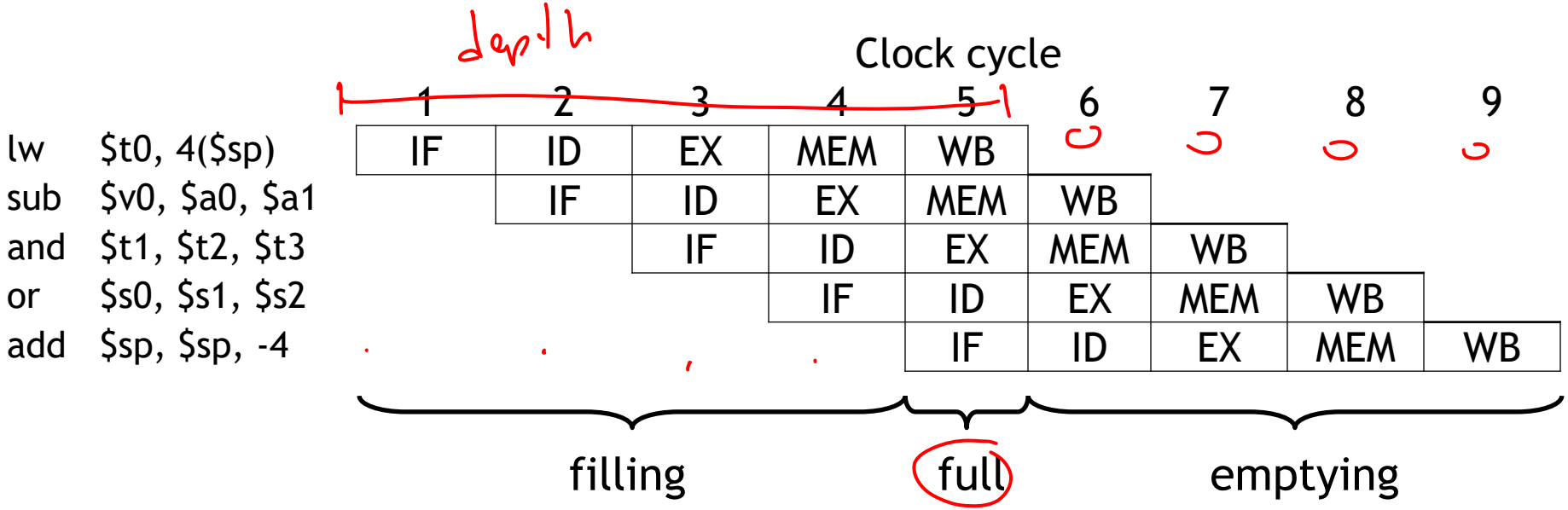
- Explain with neat diagram multi-bus datapath in a processor.
- Explain horizontal and vertical microprogrammed control with suitable example.
- Explain data forwarding with suitable example for avoiding the data hazard in pipeline execution.
- Explain the data path in pipeline execution.
- Explain the execution cycle for instruction execution with example.
- Write the various techniques to avoid the instruction hazard in pipeline execution.
- Define multiprocessor computing? Explain with suitable example for various types of multiprocessor machine according to Flynn's classification.

# A pipeline diagram




- A pipeline diagram shows the execution of a series of instructions.
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)
- This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.
  - The "lw" instruction is in its Execute stage.
  - Simultaneously, the "sub" is in its Instruction Decode stage.
  - Also, the "and" instruction is just being fetched.

# Pipeline terminology



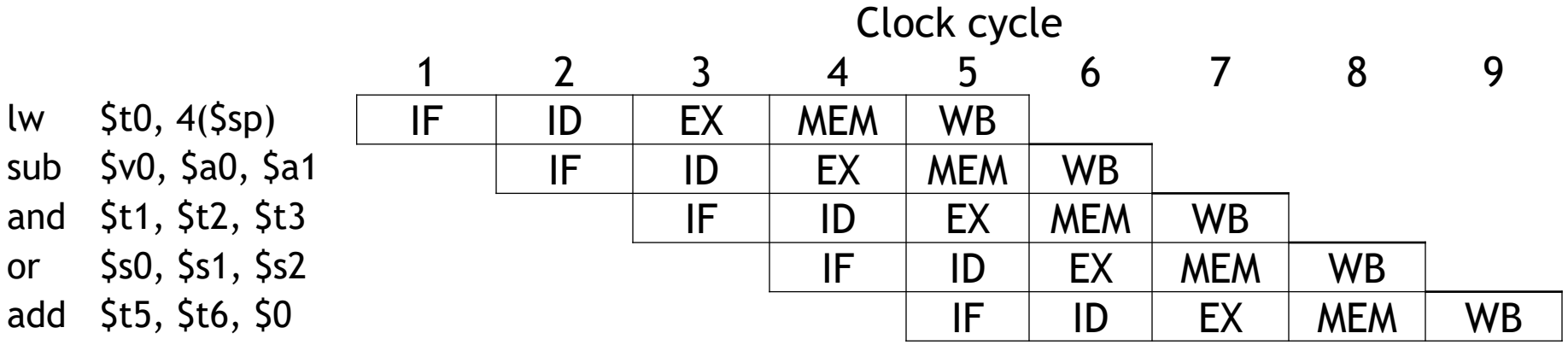

- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.

# Pipelined datapath and control

- Now we'll see a basic implementation of a pipelined processor.
  - The datapath and control unit share similarities with both the single-cycle and multicycle implementations that we already saw.
  - An example execution highlights important pipelining concepts.
- In future lectures, we'll discuss several complications of pipelining that we're hiding from you for now.

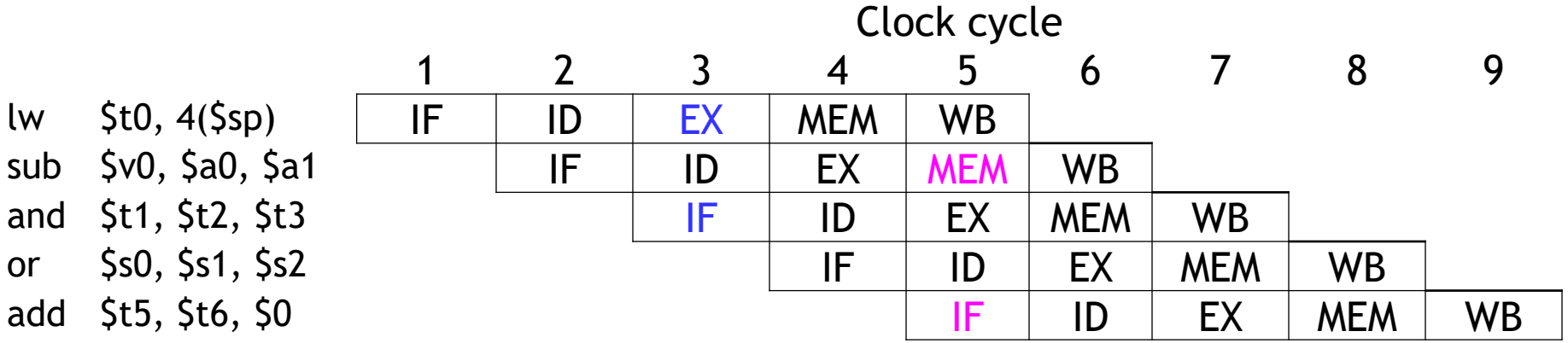
# Pipelining concepts

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.
- This increases throughput, so programs can run faster.
  - One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.



# Pipelined Datapath

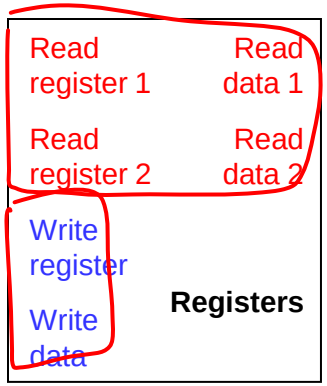
- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.



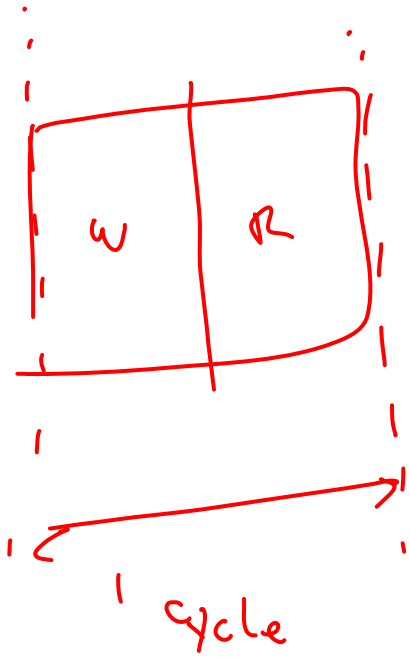

- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

# One register file is enough

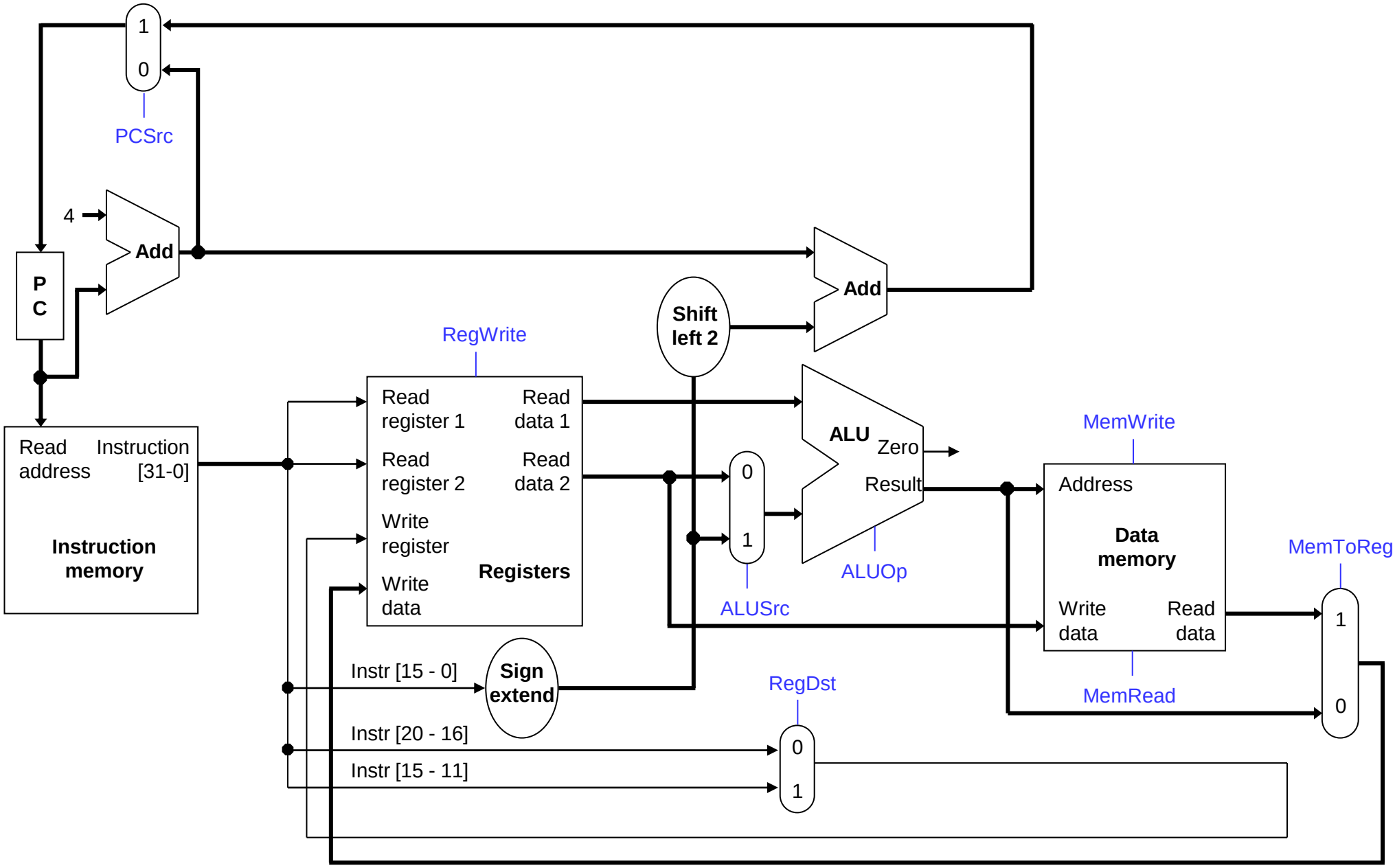
- We need only one register file to support both the ID and WB stages.



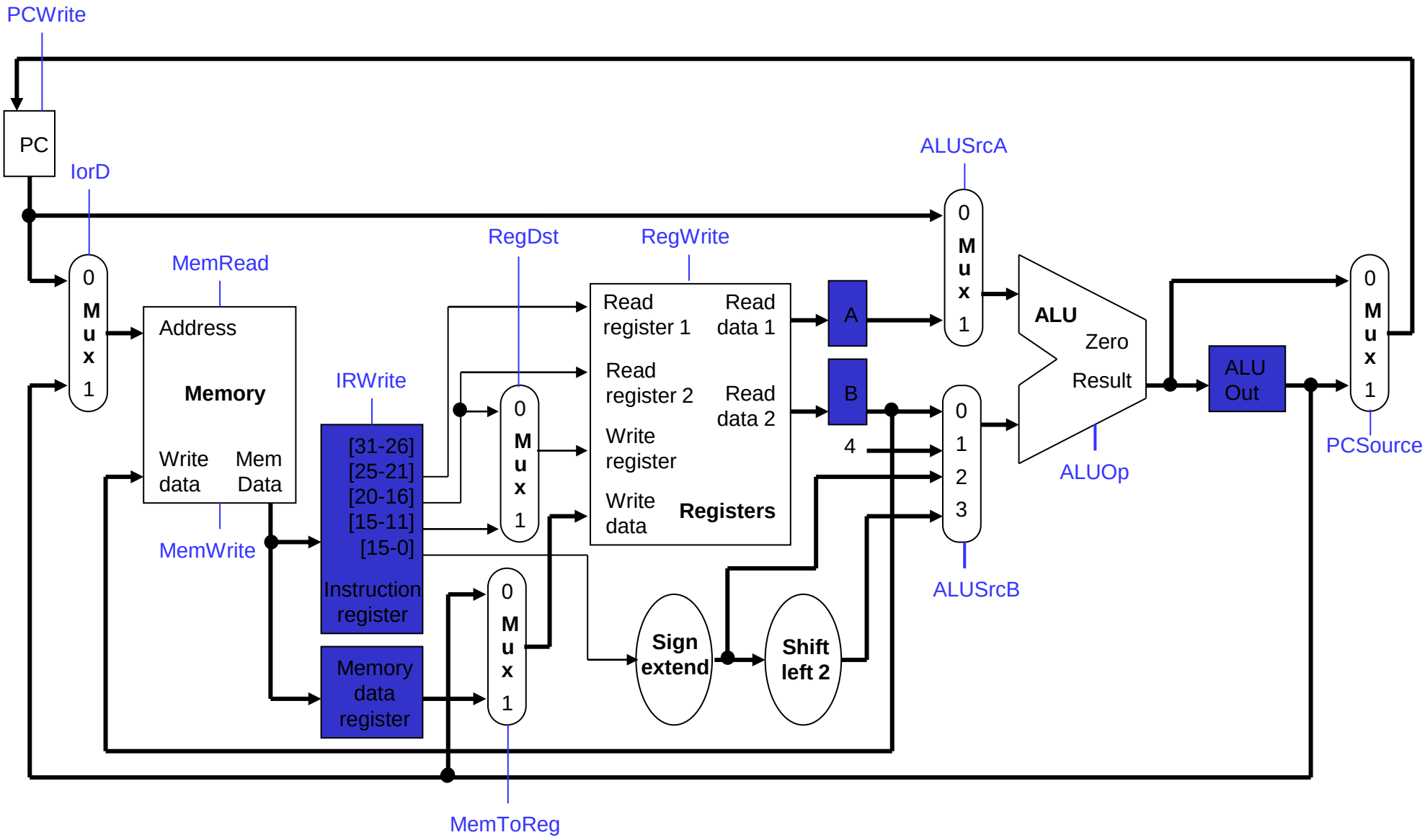

- Reads and writes go to separate ports on the register file.
- Writes occur in the first half of the cycle, reads occur in the second half.



# Single-cycle datapath, slightly rearranged



# Registers added to the multi-cycle

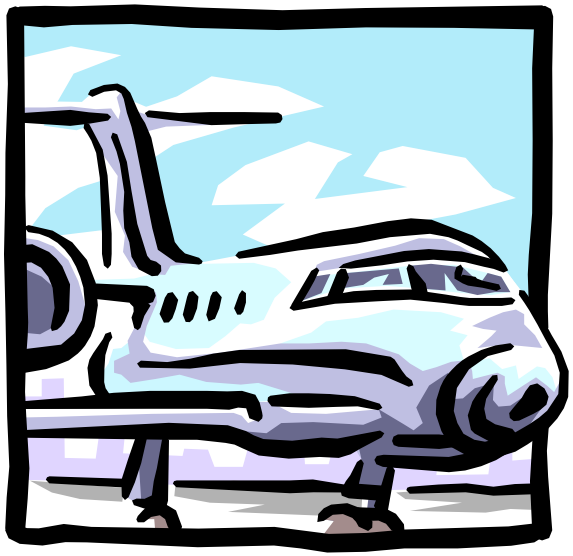


# Pipeline registers

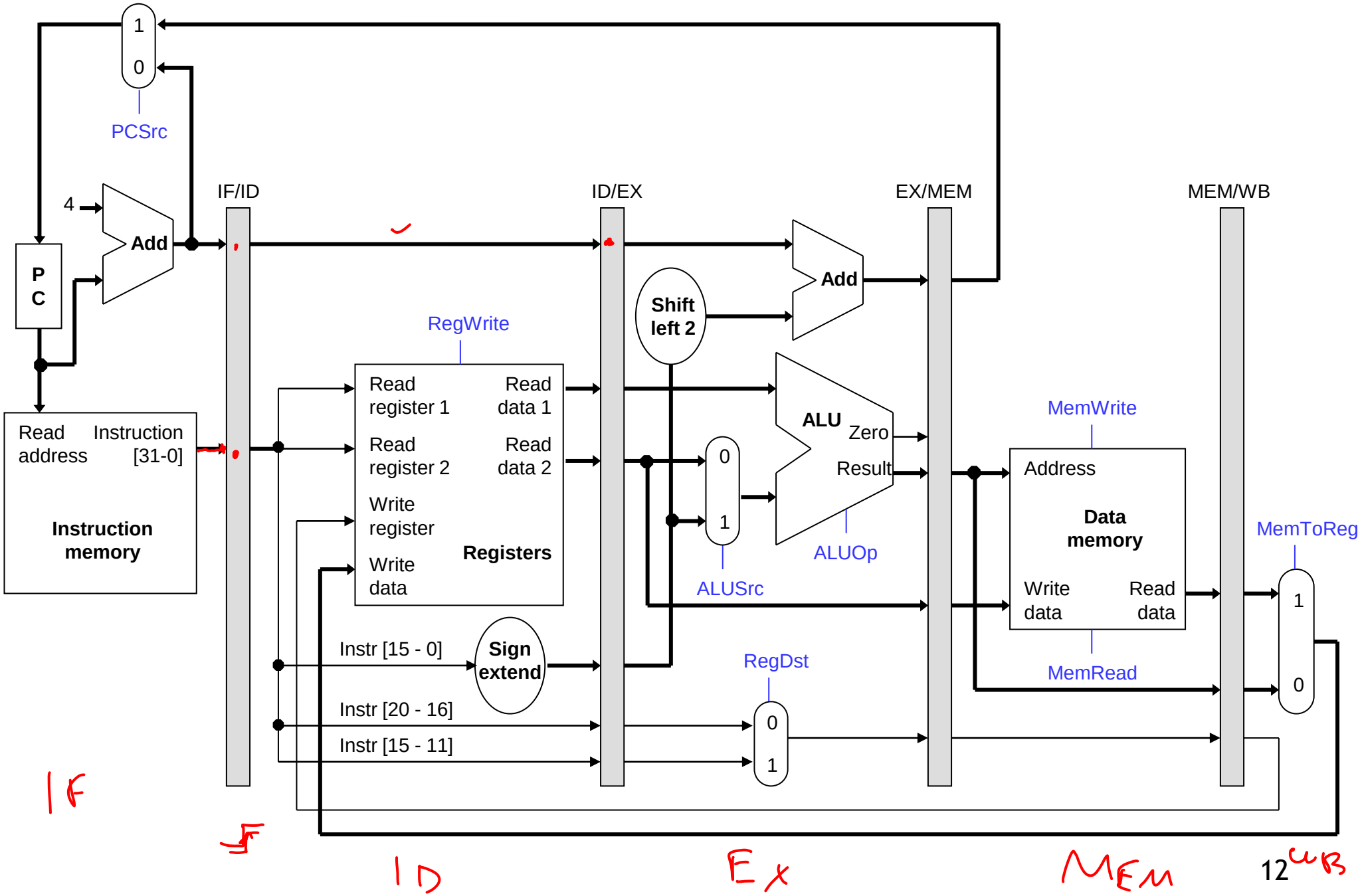
- We'll add intermediate registers to our pipelined datapath too.
- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
- The registers are named for the stages they connect.

IF/ID ID/EX EX/MEM MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done.



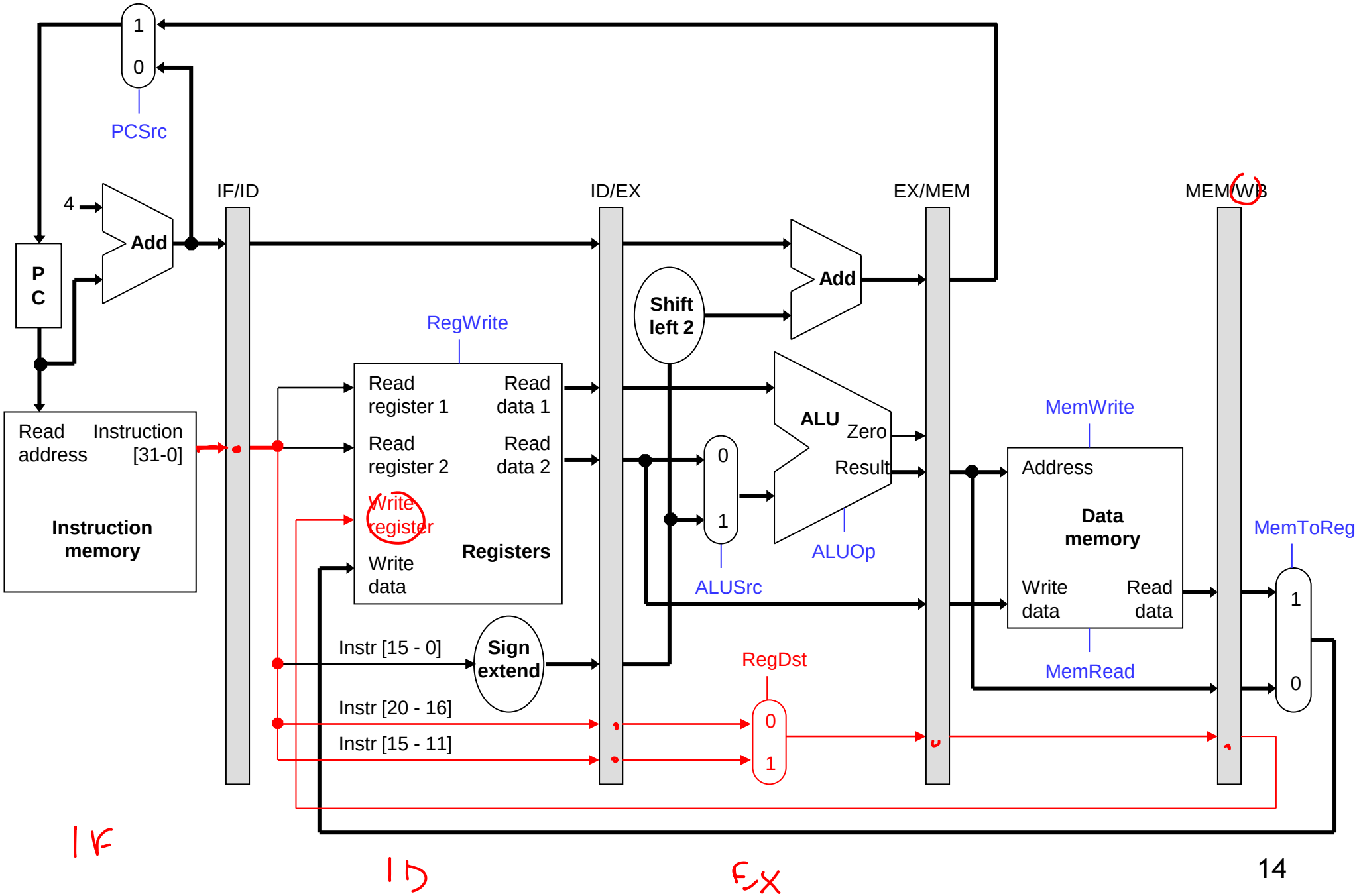
# Pipelined datapath



# Propagating values forward

- Any data values required in later stages must be propagated through the pipeline registers.
- The most extreme example is the destination register.
  - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn't updated until the *fifth* stage (WB).
  - Thus, the rd field must be passed through all of the pipeline stages, as shown in red on the next slide.
- Why can't we keep a single instruction register like we did in the multi-cycle data-path?

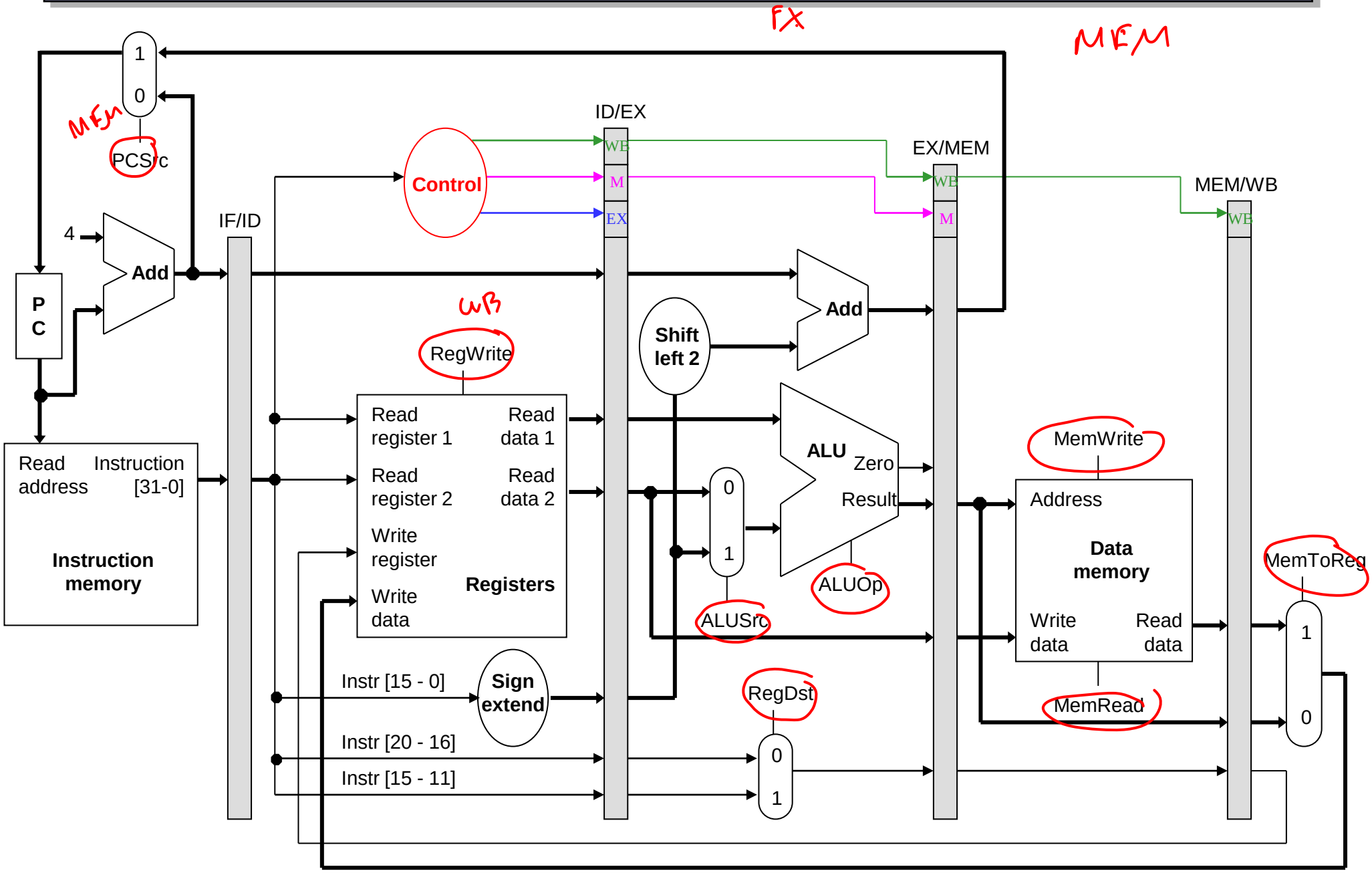
# The destination register



# What about control signals?

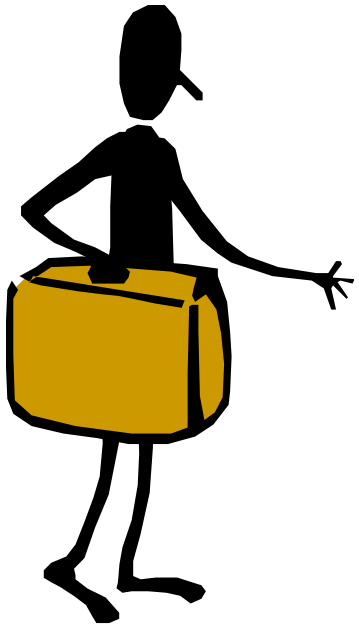
- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.
- Control signals can be categorized by the pipeline stage that uses them.

# Pipelined datapath and control



# Notes about the diagram

- The control signals are grouped together in the pipeline registers, just to make the diagram a little clearer.
- Not all of the registers have a write enable signal.
  - Because the datapath fetches one instruction per cycle, the PC must also be updated on each clock cycle. Including a write enable for the PC would be redundant.
  - Similarly, the pipeline registers are also written on every cycle, so no explicit write signals are needed.



# An example execution sequence

- Here's a sample sequence of instructions to execute.

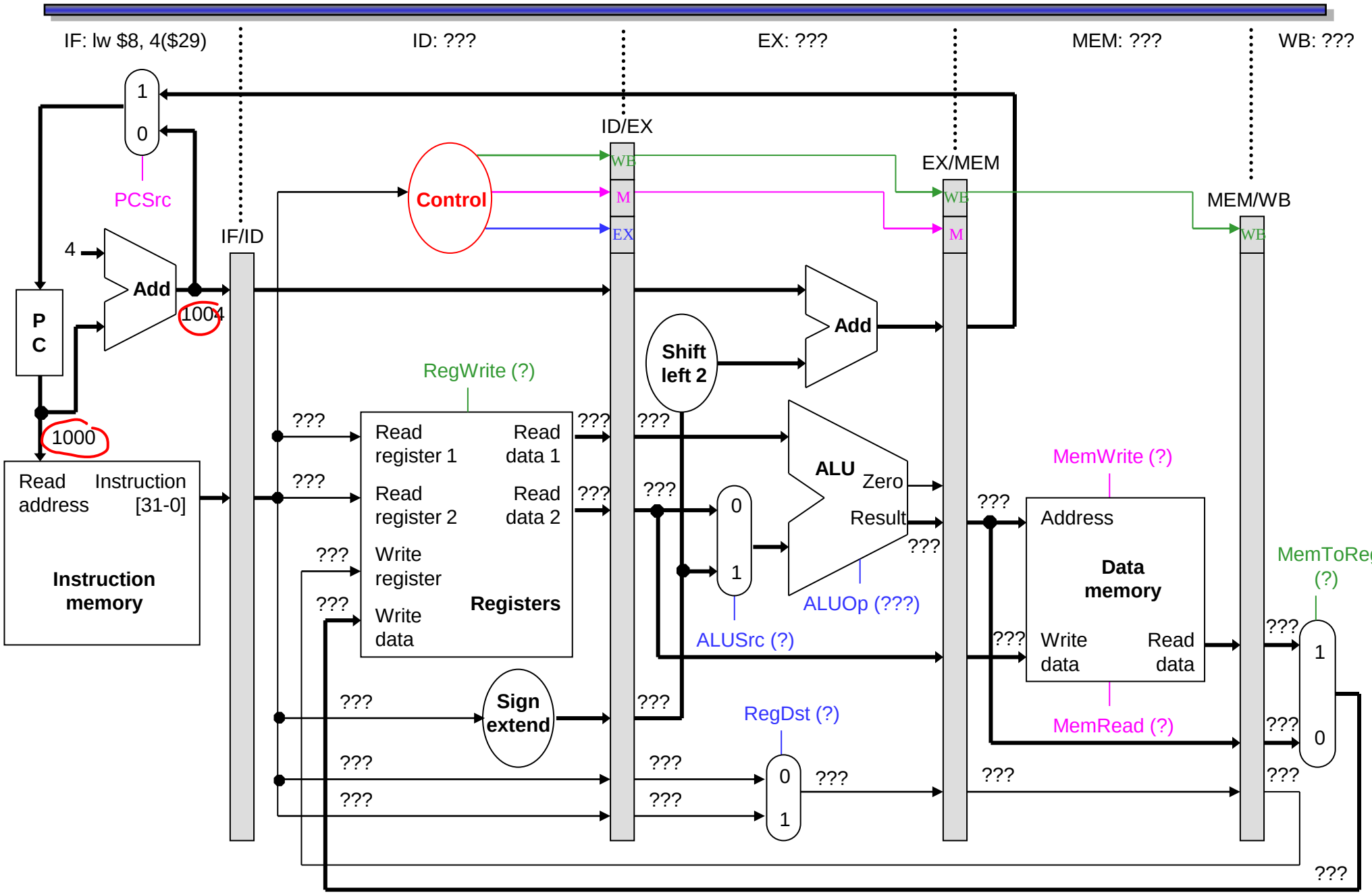
addresses in decimal

```
1000:  lw   $8, 4($29)
1004:  sub  $2, $4, $5
1008:  and  $9, $10, $11
1012:  or   $16, $17, $18
1016:  add  $13, $14, $0
```

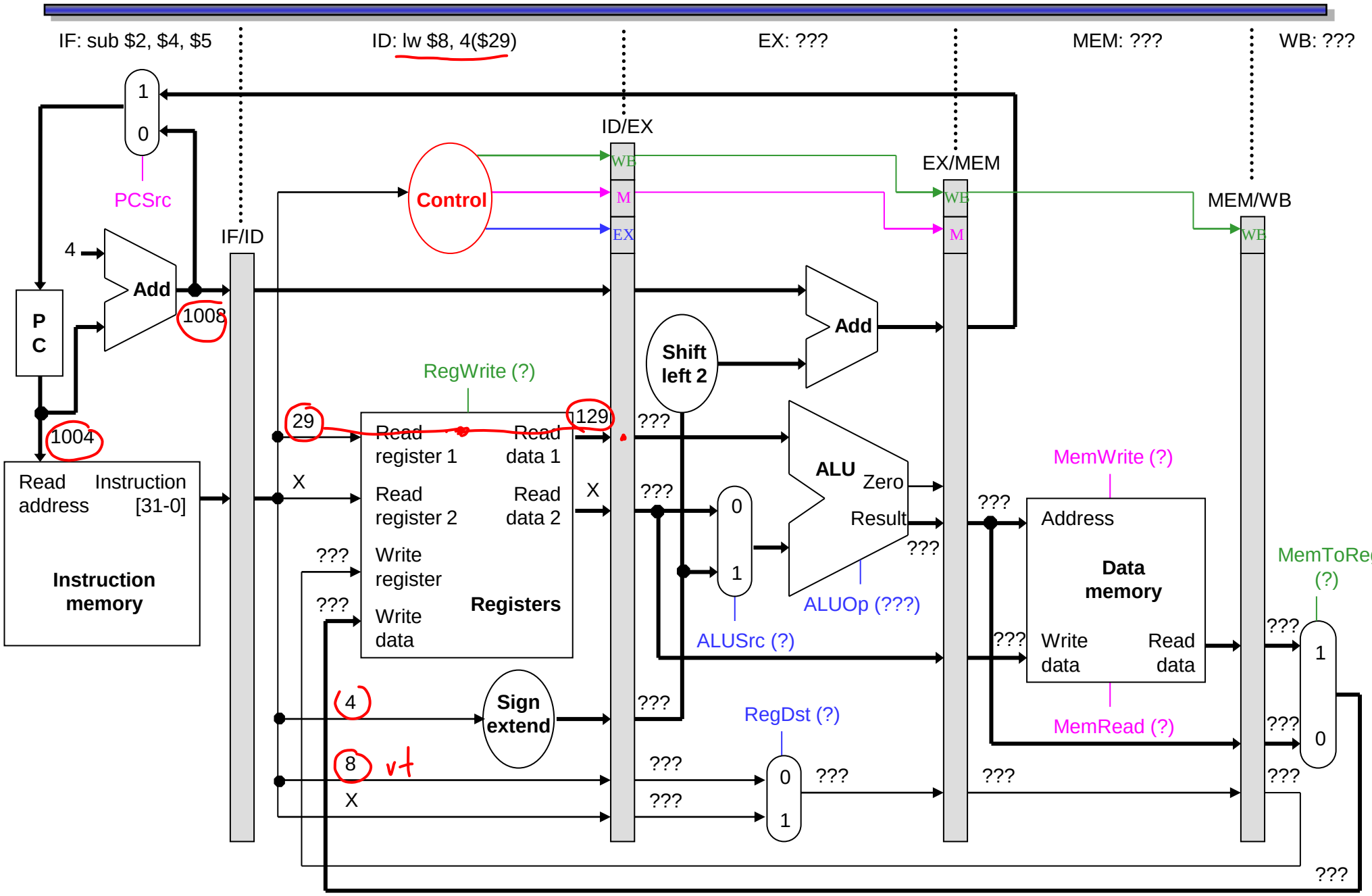
9 cycles
4 to fill
5 to complete

- We'll make some assumptions, just so we can show actual data values.
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - Every data memory location contains 99.
- Our pipeline diagrams will follow some conventions.
  - An X indicates values that aren't important, like the constant field of an R-type instruction.
  - Question marks ??? indicate values we don't know, usually resulting from instructions coming before and after the ones in our example.

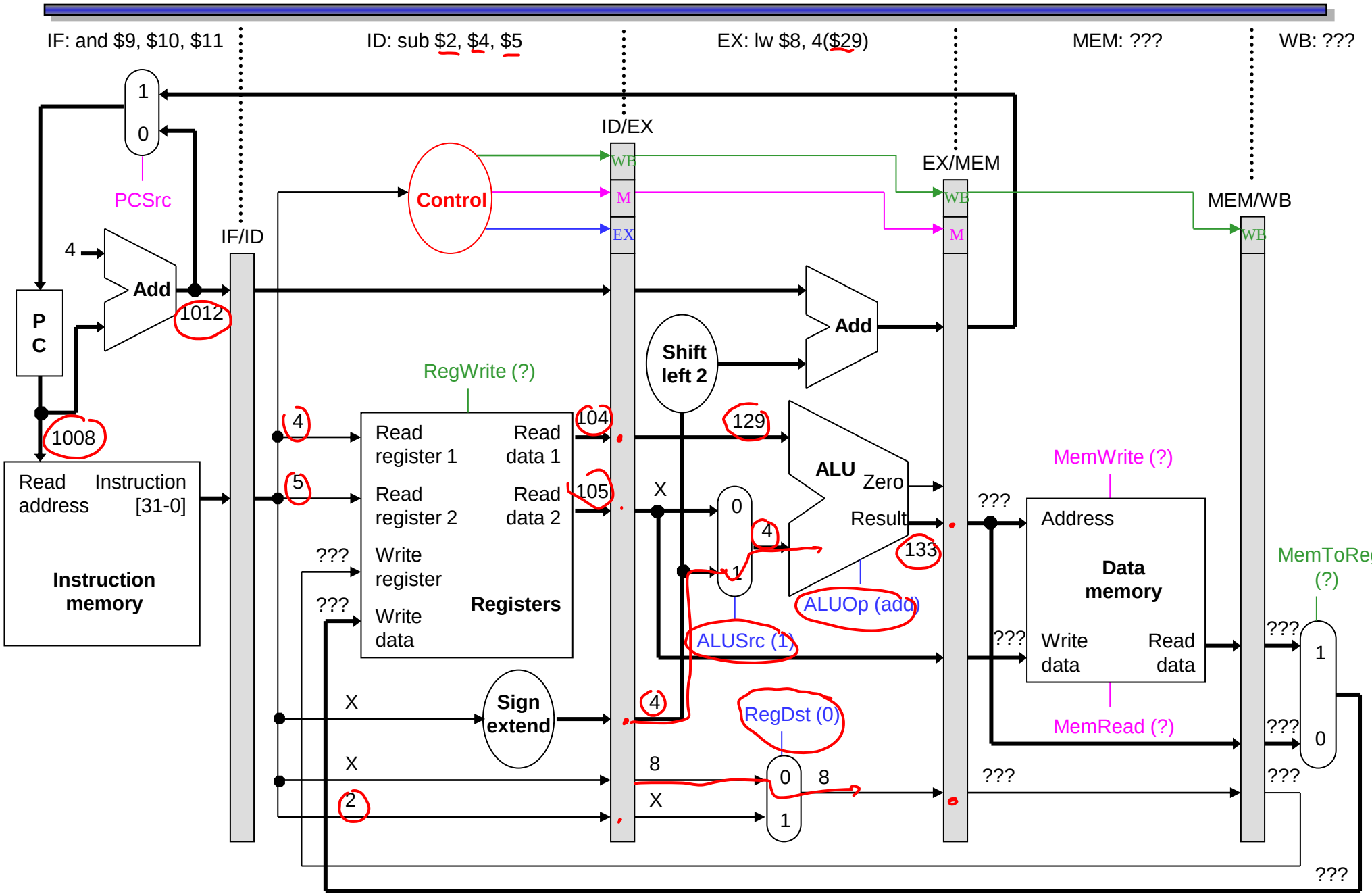
# Cycle 1 (filling)



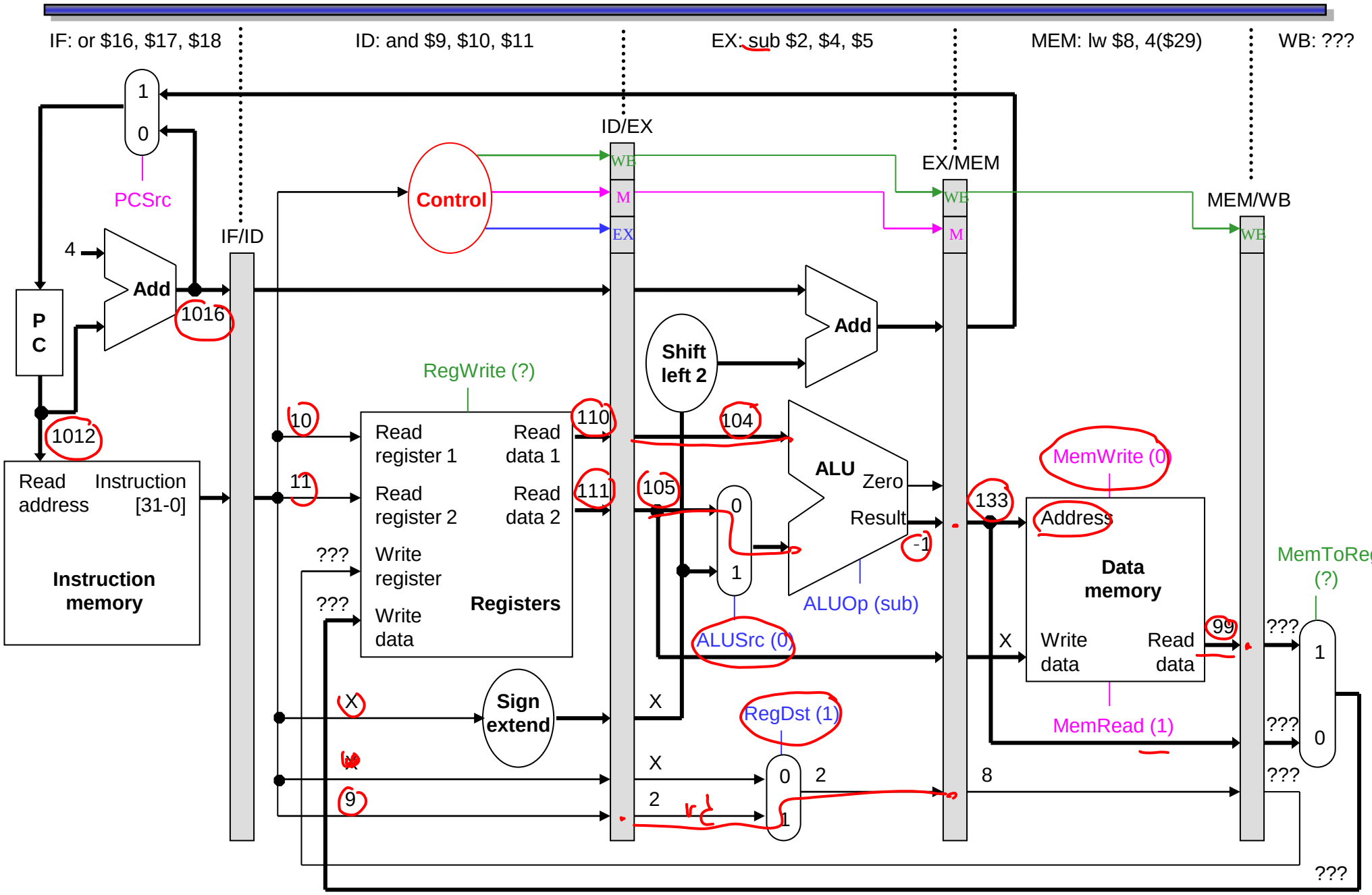
# Cycle 2


IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
4
Add
P
C
1008
IF/ID
Add
Shift
left 2
RegWrite (?)
29
1004
Read
register 1
Read
data 1
129
???
Read
address
Instruction
[31-0]
Instruction
memory
X
Read
register 2
Read
data 2
X
???
ALU
Zero
Result
???
MemWrite (?)
???
Write
register
???
Write
data
Registers
0
1
ALUOp (???)
ALUSrc (?)
Address
Data
memory
Write
data
Read
data
MemToReg
(?)
???
???
4
Sign
extend
???
RegDst (?)
MemRead (?)
8
???
???
0
1
???
???
???
???
???
???
???

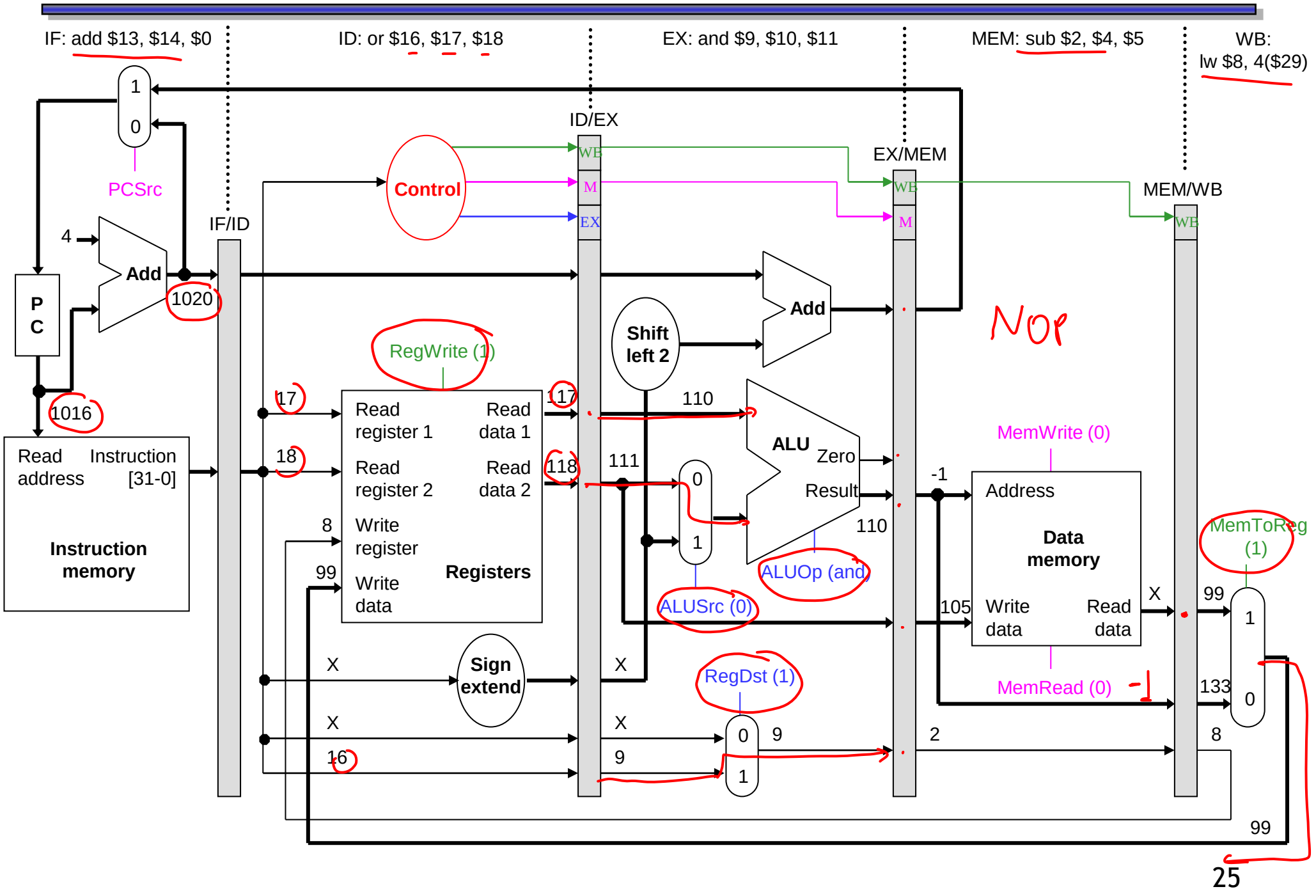
# Cycle 3


IF: and $9, $10, $11
ID: sub $2, $4, $5
EX: lw $8, 4($29)
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
1012
P C
1008
Read address
Instruction [31-0]
Instruction memory
Shift left 2
RegWrite (?)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
104
105
129
ALU
Zero
Result
133
ALUOp (add)
ALUSrc (1)
RegDst (0)
Sign extend
MemWrite (?)
MemRead (?)
MemToReg (?)
Address
Data memory
Write data
Read data
???
X
8
2
5

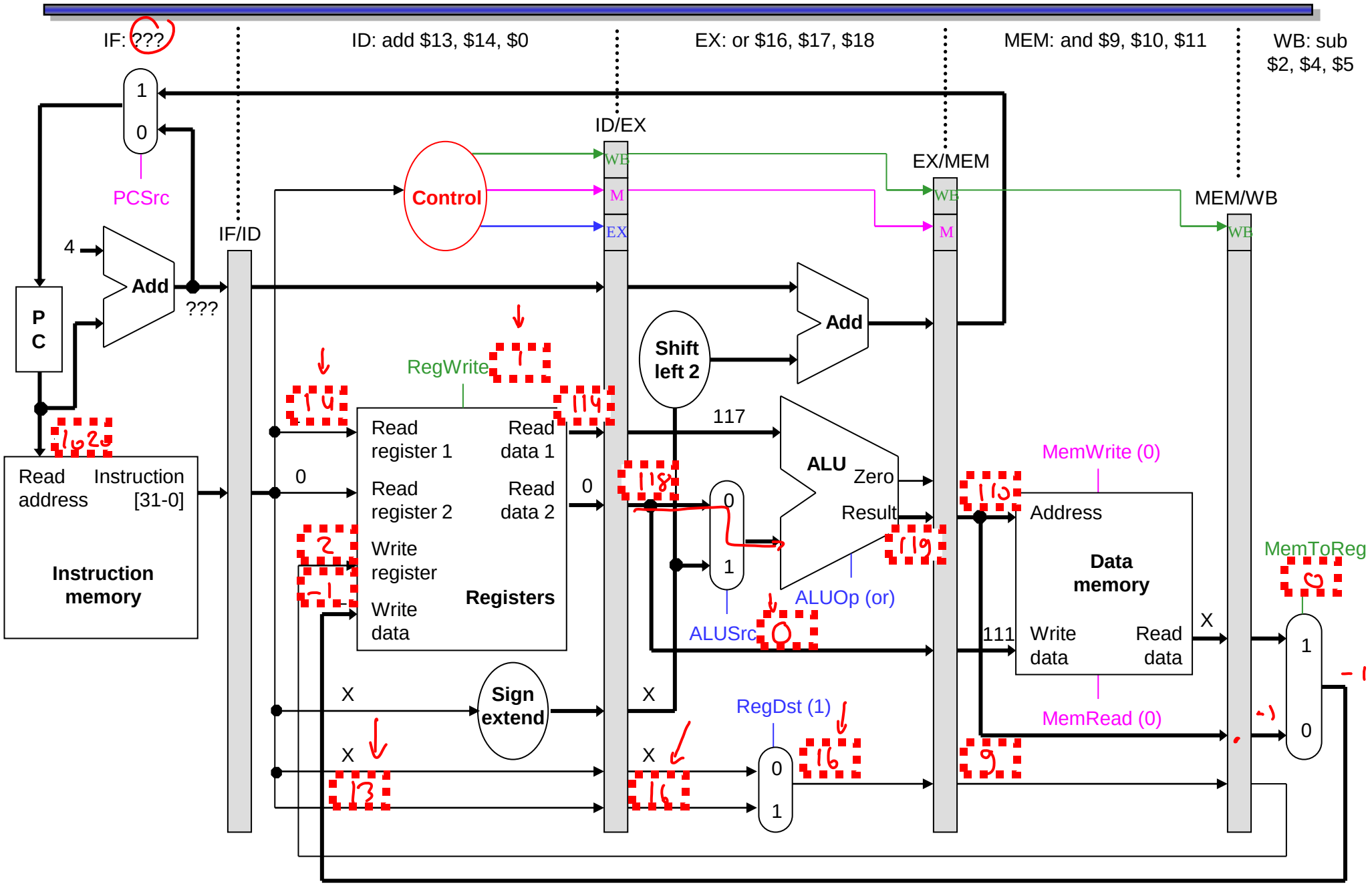
# Cycle 4


IF: or $16, $17, $18
ID: and $9, $10, $11
EX: sub $2, $4, $5
MEM: lw $8, 4($29)
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
1016
P C
1012
Read address
Instruction [31-0]
Instruction memory
RegWrite (?)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
10
11
???
???
110
111
Shift left 2
104
105
ALU
Zero
Result
-1
ALUOp (sub)
ALUSrc (0)
Sign extend
X
X
X
RegDst (1)
2
9
133
MemWrite (0)
Address
Data memory
Write data
Read data
99
X
MemRead (1)
MemToReg (?)
???
???
???
???
8

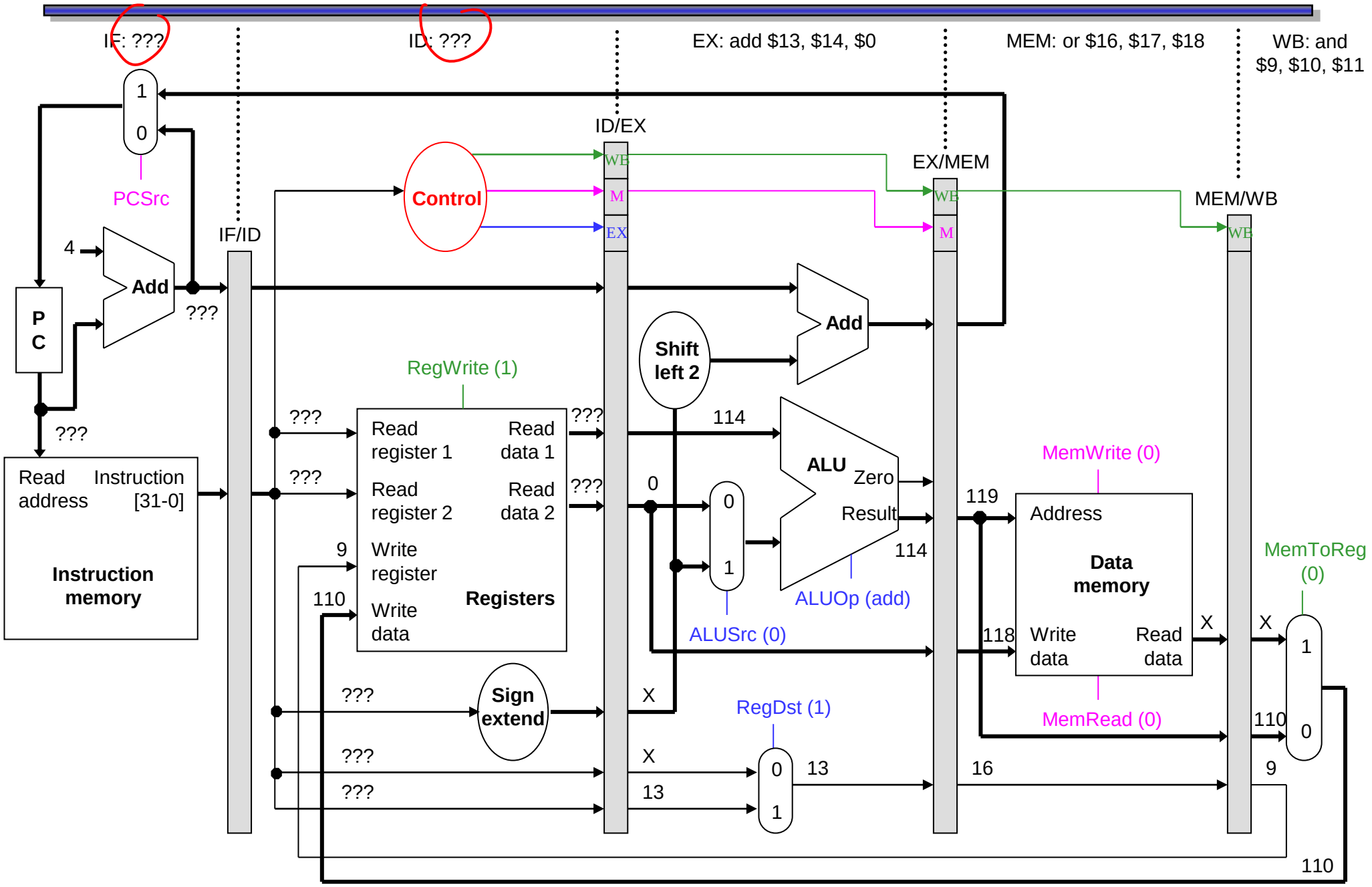
# Cycle 5 (full)


IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
1020
P C
1016
Read address
Instruction [31-0]
Instruction memory
RegWrite (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
17
18
8
99
117
118
Shift left 2
110
111
ALU
Zero
Result
ALUOp (and)
ALUSrc (0)
MemWrite (0)
MemRead (0)
Address
Data memory
Write data
Read data
-1
105
X
MemToReg (1)
133
Sign extend
RegDst (1)
16
9
2
Nop
Yay!
25

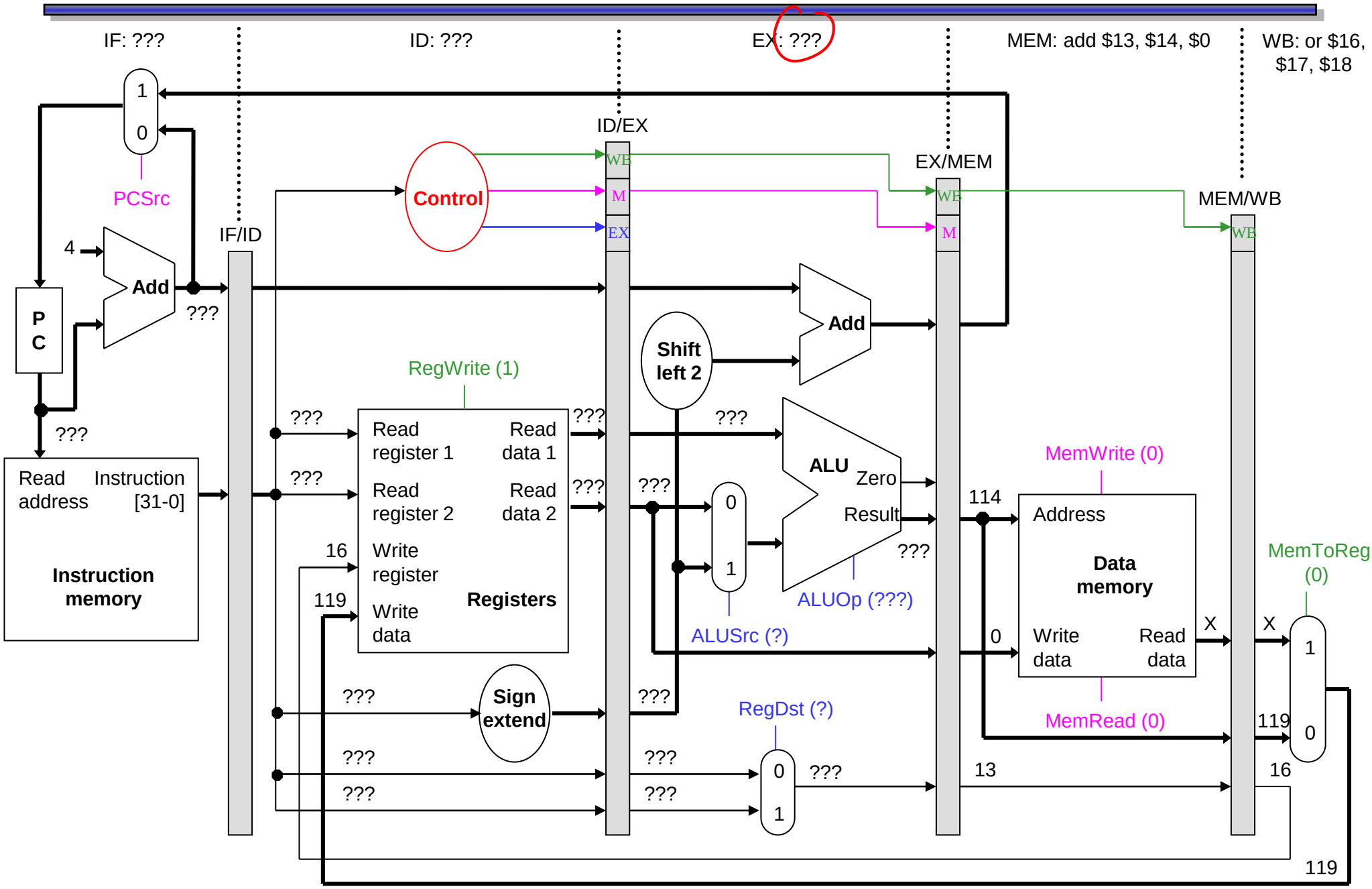
# Cycle 6 (emptying)



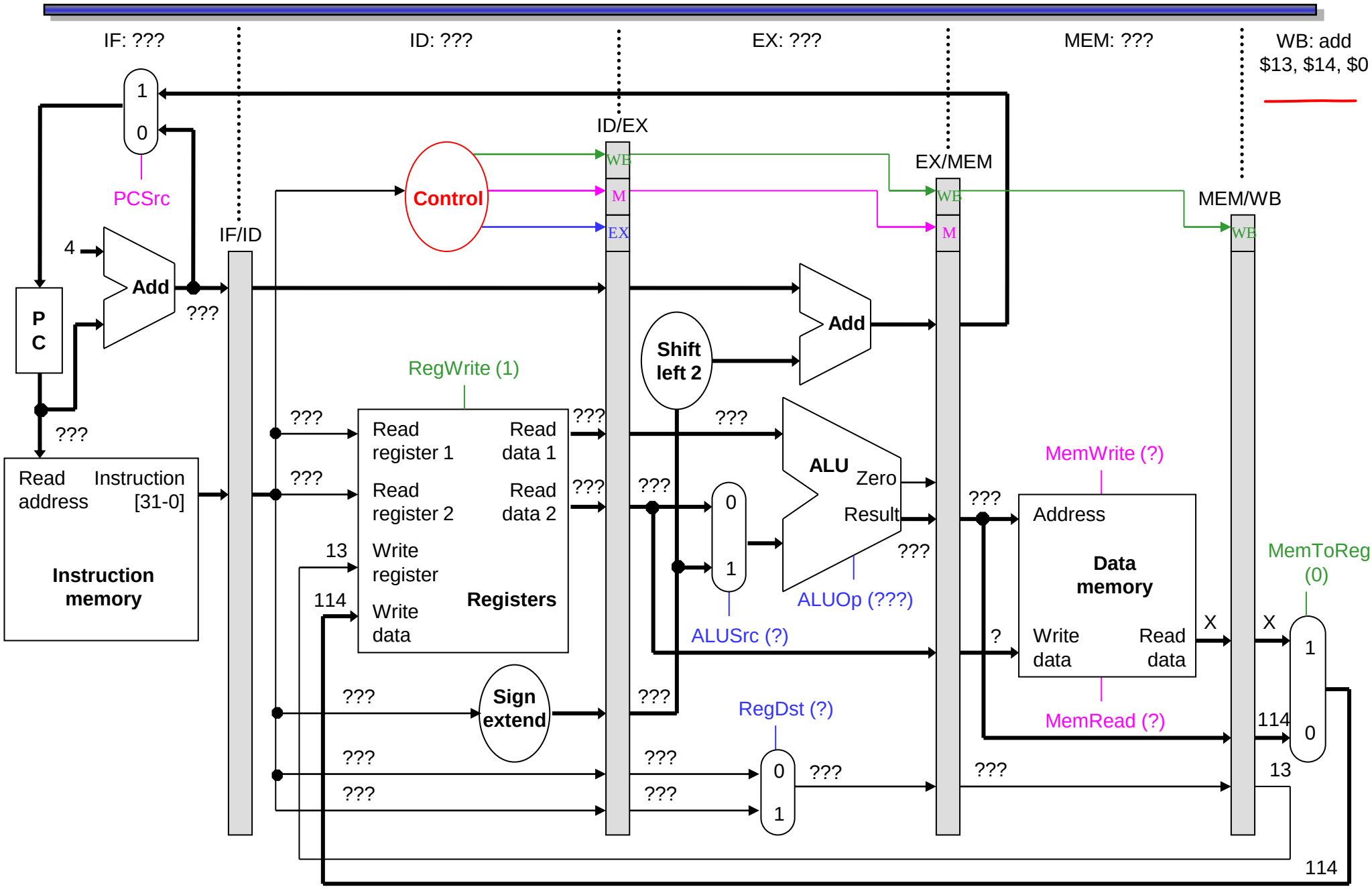
# Cycle 7


IF: ???
ID: ???
EX: add $13, $14, $0
MEM: or $16, $17, $18
WB: and $9, $10, $11
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
4
Add
???
P C
Read address
Instruction [31-0]
Instruction memory
RegWrite (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
9
110
Shift left 2
Add
114
ALU
Zero
Result
ALUOp (add)
ALUSrc (0)
0
1
MemWrite (0)
MemRead (0)
MemToReg (0)
Address
Data memory
Write data
Read data
119
118
X
Sign extend
RegDst (1)
13
16
110
9

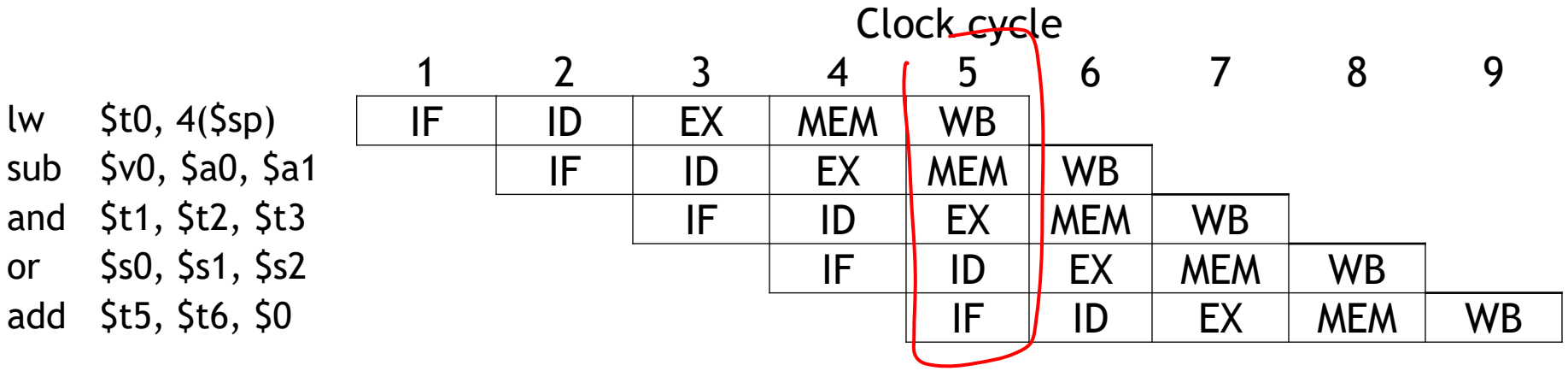
# Cycle 8


IF: ???
ID: ???
EX: ???
MEM: add $13, $14, $0
WB: or $16, $17, $18
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
P C
???
Shift left 2
RegWrite (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Read address
Instruction [31-0]
Instruction memory
16
119
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (0)
MemRead (0)
MemToReg (0)
Address
Data memory
Write data
Read data
114
0
1
X
Sign extend
RegDst (?)
13
16
119

# Cycle 9


IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
4
Add
IF/ID
P C
???
Shift left 2
Add
RegWrite (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
13
114
Instruction memory
Read address
Instruction [31-0]
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (0)
Sign extend
RegDst (?)
X
?
114
13

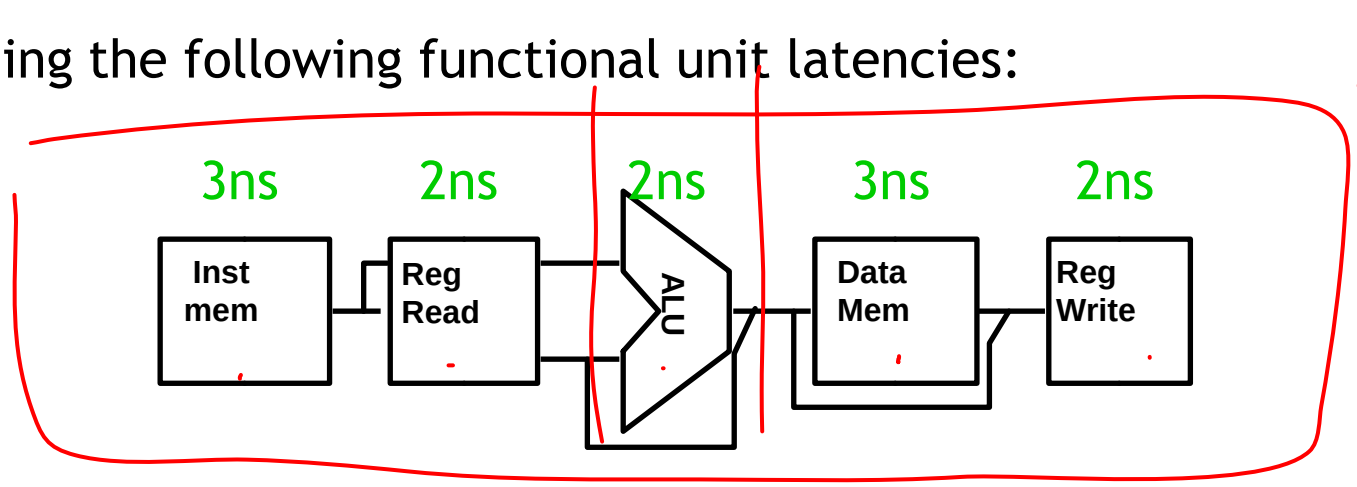
# That's a lot of diagrams there




- Compare the last nine slides with the pipeline diagram above.
  - You can see how instruction executions are overlapped.
  - Each functional unit is used by a *different* instruction in each cycle.
  - The pipeline registers save control and data values generated in previous clock cycles for later use.
  - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

# Performance Revisited

- Assuming the following functional unit latencies:




- What is the cycle time of a single-cycle implementation?
  - What is its throughput?

12ns

$\frac{1}{12ns}$

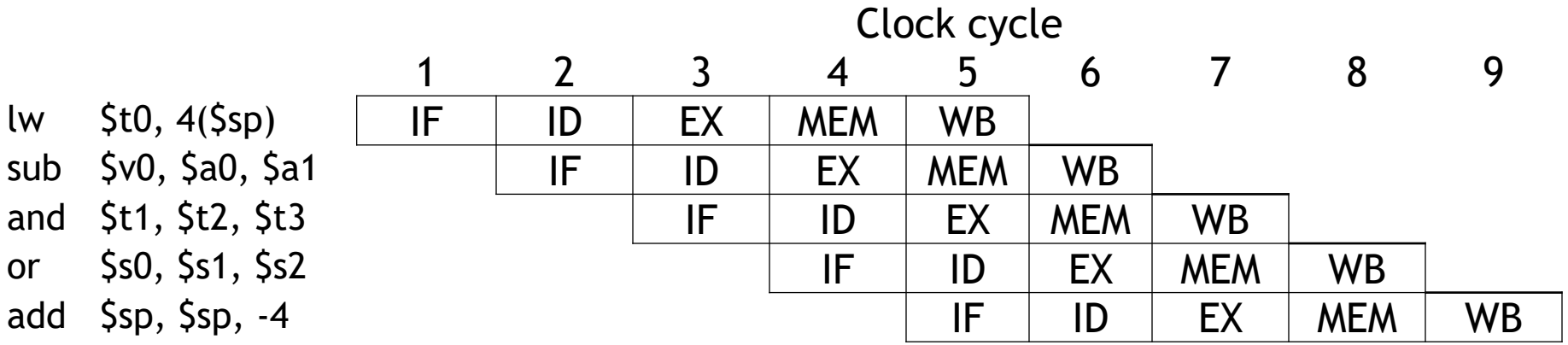
- What is the cycle time of a ideal pipelined implementation?
  - What is its steady-state throughput?

3ns

- How much faster is pipelining?

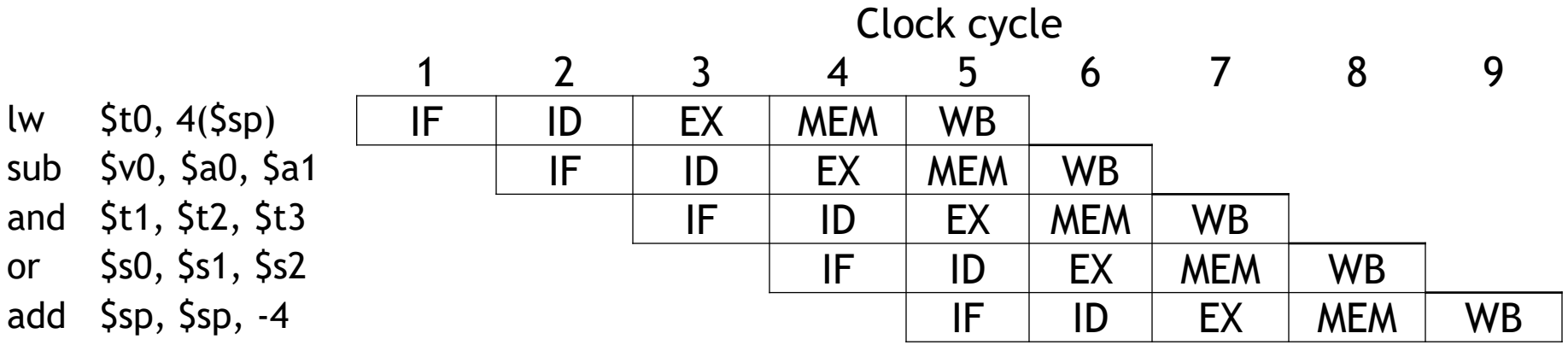
$\frac{12ns}{3ns} = 4\times$ Speedup

# Ideal speedup




- In our pipeline, we can execute up to five instructions simultaneously.
  - This implies that the maximum speedup is 5 times.
  - In general, the ideal speedup equals the pipeline depth.
- Why was our speedup on the previous slide "only" 4 times?
  - The pipeline stages are imbalanced: a register file and ALU operations can be done in 2ns, but we must stretch that out to 3ns to keep the ID, EX, and WB stages synchronized with IF and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

# The pipelining paradox




- Pipelining does *not* improve the execution time of any single instruction. Each instruction here actually takes *longer* to execute than in a single-cycle datapath (15ns vs. 12ns)!
- Instead, pipelining increases the throughput, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.
- The result is improved execution time for a *sequence* of instructions, such as an entire program.

# Instruction set architectures and pipelining

- The MIPS instruction set was designed especially for easy pipelining.
  - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle.
  - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage.
  - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler.
- Pipelining is harder for older, more complex instruction sets.
  - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields.
  - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage.

# Summary

- The pipelined datapath combines ideas from the single and multicycle processors that we saw earlier.
  - It uses multiple memories and ALUs.
  - Instruction execution is split into several stages.
- Pipeline registers propagate data and control values to later stages.
- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.

- Next lecture, we'll start talking about Hazards.

## Unit-II

## Basic Processing Unit

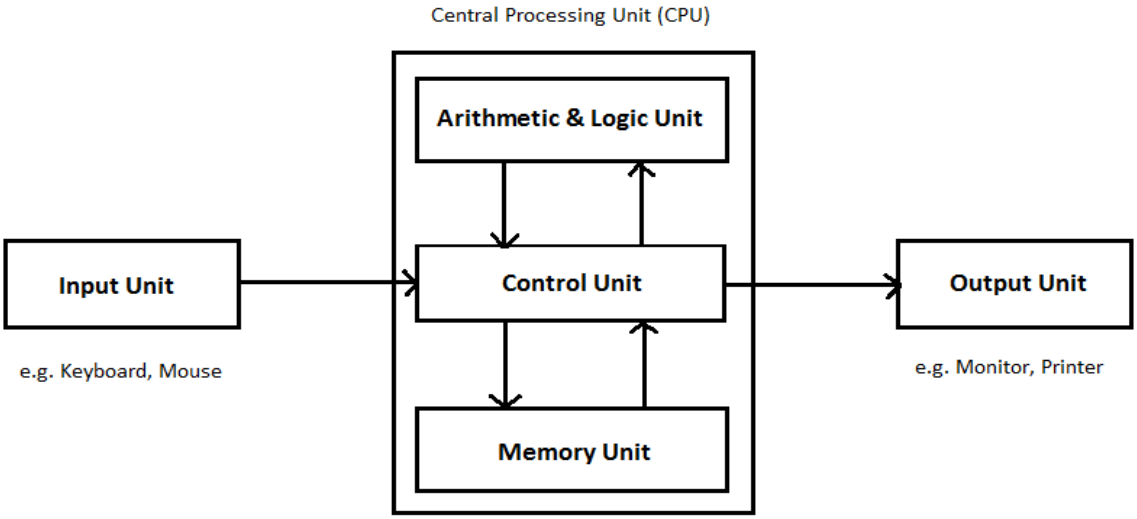
Components of the Processor:

A processor contains the following components,

- **Control Unit** - fetches, decodes, executes instructions.
- **Arithmetic & Logic Unit** - performs arithmetic and logical operations on data.
- **Registers** - fast, on-chip memory inside the CPU, dedicated or general purpose.
- **Internal Clock** - derived directly or indirectly from the system clock
- **Internal Buses** - to connect the components
- **Logic Gates** - to control the flow of information

- **What Is A Register?**
- Registers are storage locations within the circuitry of the CPU. They are very fast on-chip memory storing binary values using 32 or 64 bits. Information is held there while it is being interpreted or manipulated. Registers are **dedicated** or **general purpose**.
- **General Purpose Registers**
- Can be used by programmers to store data temporarily. Some computers may have up to 16 general purpose registers (R1...R16). Processor designers have assigned no specific role to these registers.
- **Stack Pointer (SP)**
- Points to a stack data structure holding return addresses, procedure or function parameters and local variables. Used when a procedure or function is called. Also used when an interrupt is serviced. (Interrupt = a signal from a source (hardware or software) requesting the attention of the processor)
- **Program Counter (PC)**
- Holds the address in main memory of the next instruction to be executed.
- AKA Sequence Control Register (SCR) or Sequence Register.
- **Status Register (SR)**
- Contains bits that are set and cleared based on the results of an instruction. Allows the CPU to store information such as the occurrence of an overflow. This information can be used to decide whether or not to branch out of a given sequence of instructions.
- **The Accumulator**
- Holds the result of the current set of calculations. The instruction ADD #25 means add the value 25 to the contents of the accumulator and store the result in the accumulator.
- **Current Instruction Register (CIR)**
- Stores operator and operand for the current instruction.
- Eg LDA 1000 (Load location 1000 into Accumulator)
- LDA **operator** what you do 1000 **operand** what you do it to
- **Memory Address Register (MAR)**
- Holds the address of the memory location from or to which data is to be read or written. This could be the address of an instruction to be fetched or the address of data to be used in the instruction. When fetching instructions, copies the contents of the Program Counter.
- **Memory Buffer Register (MBR)**
- AKA Memory Data Register (MDR) Temporarily stores data read from or written to memory. CPU and Memory operate at different speeds (hence buffer)
- Eg LDA 1000 placed here en route to the CIR for decoding.

- Operand (1000) placed in MAR. Contents of 1000 copied to the MBR.




**Data Path and Control Path:** A data path (also written as datapath) is a set of functional units that carry out data processing operations. Datapaths, along with a control unit, make up the CPU (central processing unit) of a computer system. A larger data path can also be created by joining more than one together using multiplexers**.**

**Execution of Complete Instruction :**

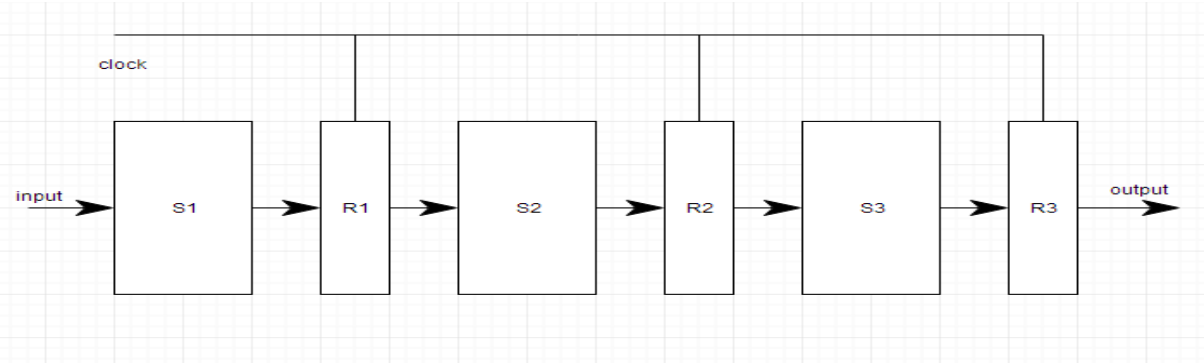
Step 1: Fetch instruction. Execution cycle starts with fetching instruction from main memory. ...
Step 2: Decode instruction. ...
Step 3: Perform ALU operation. ...
Step 4: Access memory. ...
Step 5: Update Register File. ...
Step 6: Update the PC (Program Counter)

Note: And study the note book whatever given on that.

**Pipelining:**

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like struct




ure. Instructions enter from one end and exit from another end.

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

$X = A*2^a$

$Y = B*2^b$

Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Instruction Pipeline

In this a stream of instructions can be executed by
overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

## Pipeline Conflicts

There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below:

### 1. Timing Variations

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

### 2. Data Hazards

When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

### 3. Branching

In order to fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

### 4. Interrupts

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

### 5. Data Dependency

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

## Advantages of Pipelining

1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

## Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

## Types of Hazards:

Data hazards:

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (also termed race hazards). There are three situations in which a data hazard can occur:

1. read after write (RAW), a *true dependency*
2. write after read (WAR), an *anti-dependency*
3. write after write (WAW), an *output dependency*

Structural hazards

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory.[3] They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline. It can be detected by using a hazard control unit.

Control hazards (branch hazards)

Branching hazards (also termed control hazards) occur with branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).

Exception handling:

*Exceptions* and *interrupts* are unexpected events that disrupt the normal flow of instruction execution. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor. You are to implement exception and interrupt handling in your multicycle CPU design.

When an exception or interrupt occurs, the hardware begins executing code that performs an action in response to the exception. This action may involve killing a process, outputting a error message, communicating with an external device, or horribly crashing the entire computer system by initiating a "Blue Screen of Death" and halting the CPU. The instructions responsible for this action reside in the *operating system kernel*, and the code that performs this action is called the *interrupt handler code*. You can think of handler code as an operating system subroutine. After the handler code is executed, it may be possible to continue execution after the instruction where the execution or interrupt occurred.

Exceptions: Types

For your project, there are three events that will trigger an exception: *arithmetic overflow*, *undefined instruction*, and *system call*.

*Arithmetic overflow* occurs during the execution of an add, addi, or sub instruction. If the result of the computation is too large or too small to hold in the result register, the *Overflow* output of the ALU will become high during the execute state. This event triggers an exception.

*Undefined instruction* occurs when an unknown instruction is fetched. This exception is caused by an instruction in the IR that has an unknown opcode or an R-type instruction that has an unknown function code.

*System call* occurs when the processor executes a syscall instruction. Syscall instructions are used to implement operating system services (functions).

Interrupts:

In **computer architecture**, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

Types of Interrupts:

Synchronous Interrupt: The source of interrupt is in phase to the system clock is called synchronous interrupt. In other words, interrupts which are dependent on the system clock. ...

Asynchronous Interrupts: If the interrupts are independent or not in phase to the system clock is called asynchronous interrupt.

NEED FOR INTERRUPTS

The operating system is a reactive program

When you give some input it will perform computations and produces output but meanwhile you can interact with the system by interrupting the running process or you can stop and start another process

This reactiveness is due to the interrupts

Modern operating systems are interrupt driven

Parallelism and multiprocessor architecture:

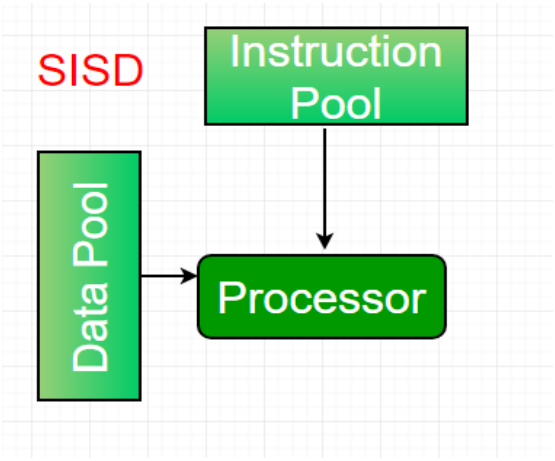
Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task.

A multiprocessor is a computer system with two or more central processing units (CPUs), with each one sharing the common main memory as well as the peripherals. This helps in simultaneous processing of programs. ... A good illustration of a multiprocessor is a single central tower attached to two computer systems.

Flynn's Classification:

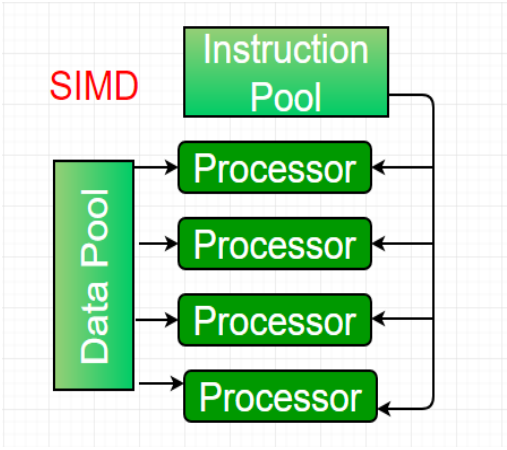
Single-instruction, single-data (SISD) systems –
An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential

computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.



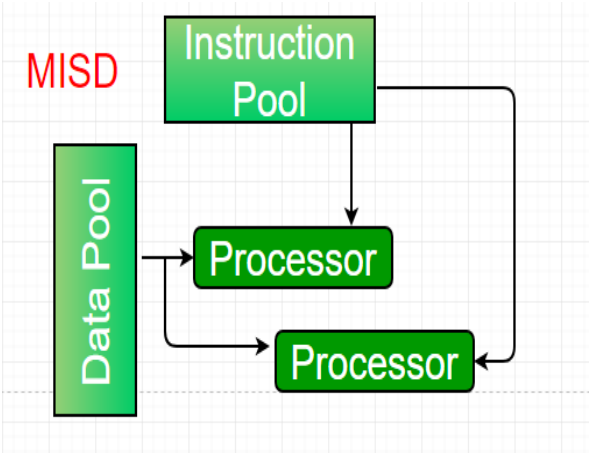

The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, workstations.

Single-instruction, multiple-data (SIMD) systems –
An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.




Dominant representative SIMD systems is Cray's vector processing machine.

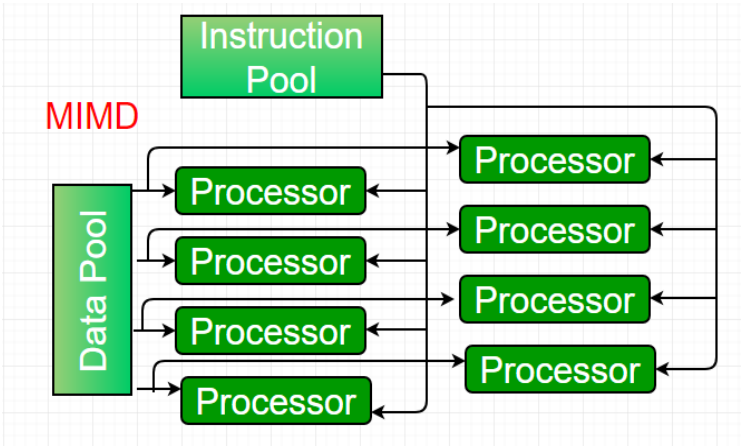
Multiple-instruction, single-data (MISD) systems –
An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .




Example Z = sin(x)+cos(x)+tan(x)
The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.

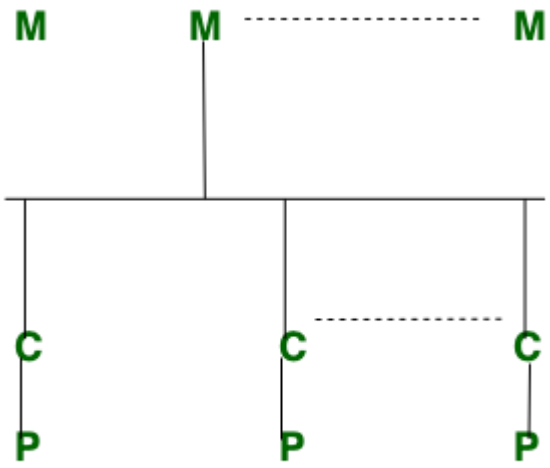
Multiple-instruction, multiple-data (MIMD) systems –

An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.



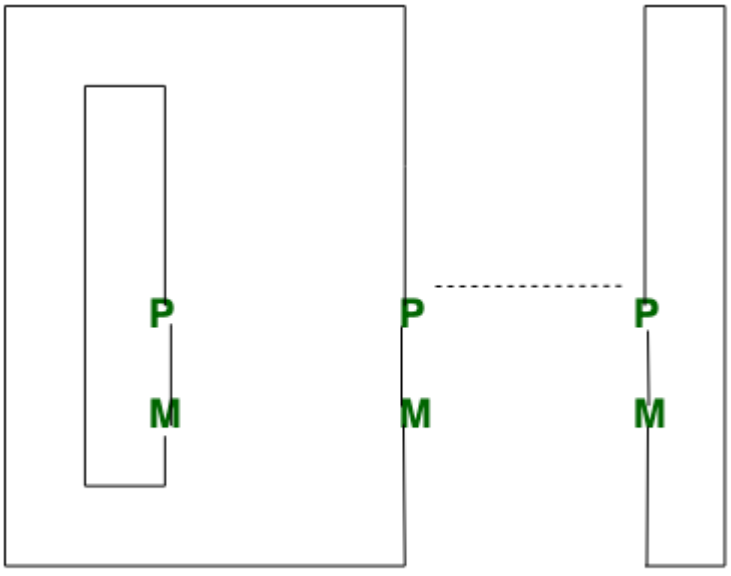

Difference between Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA):

Uniform Memory Access (UMA): In UMA, where Single memory controller is used. Uniform Memory Access is slower than non-uniform Memory Access. In Uniform Memory Access, bandwidth is restricted or limited rather than non-uniform memory access. There are 3 types of buses used in uniform Memory Access which are: Single, Multiple and Crossbar. It is applicable for general purpose applications and time-sharing applications.




UMA shared memory

Non-uniform Memory Access (NUMA): In NUMA, where different memory controller is used. Non-uniform Memory Access is faster than uniform Memory Access. Non-uniform Memory Access is applicable for real-time applications and time-critical applications.




NUMA shared memory

| S.NO | UMA | NUMA |
| --- | --- | --- |
| 1. | UMA stands for Uniform Memory Access. | NUMA stands for Non-uniform Memory Access. |
| 2. | In Uniform Memory Access, Single memory controller is used. | In Non-uniform Memory Access, Different memory controller is used. |
| 3. | Uniform Memory Access is slower than non-uniform Memory Access. | Non-uniform Memory Access is faster than uniform Memory Access. |
| 4. | Uniform Memory Access has limited bandwidth. | Non-uniform Memory Access has more bandwidth than uniform Memory Access. |
| 5. | Uniform Memory Access is applicable for general purpose applications and time-sharing applications. | Non-uniform Memory Access is applicable for real-time applications and time-critical applications. |
| 6. | In uniform Memory Access, memory access time is balanced or equal. | In non-uniform Memory Access, memory access time is not equal. |
| 7. | There are 3 types of buses used in uniform Memory Access which | While in non-uniform Memory Access, There are 2 types of |

| | |
|---|---|
| are: Single, Multiple and Crossbar. | buses used which are: Tree and hierarchical. |

Array and Vector Processor:

In computing, a vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors. ... Vector and array processing technology are most often seen in high-traffic servers.

An array is made up of indexed collections of information called indices. Though an array can, in rare cases, have only one index collection, a vector is technically indicative of an array with at least two indices. Vectors are sometimes referred to as "blocks" of computer data.

Vector and array processing technology are most often seen in high-traffic servers.