

# Instruction Set Architecture and Memory Unit

---

Module 2

---

# Addressing Modes

---

- Every instruction has an **operation field** (what to do) and **operands** (data to use).
- Operands can come from **registers** or **memory**.
- Addressing modes describe how the location of an operand is specified in an instruction. They define how the processor accesses data stored in registers or memory during execution.
- **Why Addressing Modes are Needed**
- **Programming flexibility**
  - Use pointers, counters, loops, indexing, program relocation.
- **Efficiency**
  - Reduce instruction size (fewer address bits).
  - Write faster, shorter programs.

# Addressing Modes

- **Role in Instruction Cycle**
- **Fetch** → PC gives the address of the instruction.
- **Decode** → Control unit identifies:
  - The **operation** to perform
  - The **addressing mode** used
  - The **location of operands**
- **Execute** → Instruction is carried out, then PC moves to the next.

# Types of Addressing Modes

- 1) Immediate Mode
- 2) Register Mode
- 3) Absolute Mode (Direct Mode)
- 4) Indirect Mode
- 5) Index Mode
- 6) Base with Index Mode
- 7) Base with Index & Offset Mode
- 8) Relative Mode
- 9) Auto-increment Mode
- 10) Auto-decrement Mode

# Immediate Mode

Operand is written directly within the instruction itself. This means the processor does not need to look anywhere else to find the data—it's already provided.

- This mode is fast because the data is immediately available and doesn't require memory access or additional calculations.
- **Key Features**
  - Operand value is part of the instruction.
  - No memory or register lookup is required.
  - Used when constants or fixed values are needed.
- **Immediate Operand Notation**
  - The sharp sign # is used before a value in an instruction to indicate that this value is an **immediate operand**.
  - It tells the processor that the value is to be used as it is, not as an address or register reference.

# Immediate Mode

- Example
- `MOV R1, #10`

This means:

- Move the constant value 10 directly into register R1
- The processor reads 10 from the instruction and stores it in R1 immediately.

# Register Mode

is the data stored inside a processor register. The instruction specifies the name or address of the register where the data is located.

- Since the data is already inside the register, the processor can quickly access it without needing to fetch it from memory.

- **Key Features**

- Operand is inside a register.
- Instruction refers to the register's name.
- Provides faster access compared to memory-based modes.

# Register Mode

- Example
- `MOV R1, R2`

This means:

- Copy the contents of register R2 into register R1
- No memory access is needed, making it efficient.
- **When is it used?**
- Moving data between registers.
- Performing arithmetic and logic operations using register values.
- Efficient access during execution of instructions.



# Absolute Mode (Direct Mode)

**Direct Mode**, the memory address of the operand is directly specified in the instruction. The processor uses this address to access the data from memory.

- This mode allows instructions to access data stored at a particular location in memory without extra calculations.
- **Key Features**
- The instruction contains the actual memory address.
- The processor directly retrieves data from that address.
- It's simple and straightforward but requires knowing the memory location beforehand.

# Absolute Mode

- Example
- `MOV R1, [5000]`
- This means:
- Access the memory location 5000 and load its content into register R1
- The processor reads from address 5000 in memory.

# Indirect Addressing Mode

in many programs, we need to access different memory locations repeatedly, such as when processing a list or array. The **indirect addressing mode** provides this flexibility by using a processor register to hold the address of the operand instead of hardcoding it in the instruction.

- **How It Works**

- **Register as a Pointer**

A register, say R5 is used to store the memory address of the operand. The instruction refers to this register, and the processor uses its contents to access the operand.

# Indirect Addressing Mode

- **Dynamic Access**

The contents of the register can be changed during program execution. This means the same instruction can access different data at different times, simply by updating the register's value.

- **Example from Programming**

In C language, the statement `A = *B;` means that the contents at the memory location pointed to by **B** are loaded into A

- Example: Load R5, (R4)
- Here, R4 contains the address of the memory location where the operand is stored.
- The processor reads the contents of the memory location pointed to by R4 and loads it into register R5

# Indirect Addressing Mode

- Adding a List of Numbers
- A list of numbers is stored in memory locations like NUM1, NUM2, etc
- Register R4 is used as a pointer, initially holding the address of NUM.
- The program loads each number using indirect addressing: Load R5, (R4) and then updates the pointer to the next number by incrementing R4
- Example – Register-Based Indirect Access
- Move R4, #NUM1
- Load R5, (R4)
- Add R4, R4, #4

# Indirect Addressing Mode

- Move R4, #NUM1
- This instruction loads the address of the first element in the list NUM1 into register R4
- #NUM1 is an **immediate value**, meaning that the actual address of the first number is directly provided in the instruction.
- If the memory layout is something like:

Address	Value
1000	5
1004	10
1008	15

# Indirect Addressing Mode

- Then #NUM1 might represent 1000 so after this instruction:
- $R4 = 1000$

Load R5, (R4)

It reads the value stored at the memory address pointed to by R4

That value is then loaded into R5

At this point,  $R4 = 1000$  so the instruction accesses memory at 1000

It reads the value stored there 5 and loads it into R5

$R5 = 5$

This is how indirect addressing works—the memory location is not directly written in the instruction but is referenced by the register R4

# Indirect Addressing Mode

- Add R4, R4, #4
- Adds 4 to the current value of R4 and stores the result back into R4
- This effectively moves the pointer from the current memory location to the next one.
- Why add 4?
- In a typical system, integers are stored in **4 bytes**.
- So moving from one integer to the next requires adding 4 to the memory address.
- Before execution:  $R4 = 1000$
- After execution:  $R4 = 1000 + 4 = 1004$
- Now R4 points to NUM2 the next integer in the list



# Indirect Addressing Mode

- **Why This Is Important**
- **Reusability** – The same code can work on different lists just by changing the pointer's starting address.
- **Efficiency** – The processor doesn't need to rewrite the instruction for each element.
- **Flexibility** – Allows programs to handle arrays, linked lists, or other data structures dynamically.

# Indexing and Arrays

- The **Index Addressing Mode** is a powerful way to access elements in lists or arrays, providing flexibility in dealing with structured data.
- **How Index Mode Works**
- The **effective address (EA)** of the operand is calculated by adding:
- **$EA = X + [Ri]$**
- where:
- X is a constant offset provided in the instruction.
- Ri is a general-purpose register known as the **index register**.
- This mode is especially useful for accessing elements of arrays or records where items are stored consecutively in memory.

# Indexing and Arrays

- **Notation**
- The instruction is written as:
- **X(Ri)**
- Example:  
**4(R2)** → Accesses the memory location at [R2] + 4
- **Example - Student Records**
- Suppose a list of student records is stored in memory beginning at LIST
- Each record contains:
- Student ID
- Scores for three tests
- Each record is 4 words long → 16 bytes if the word length is 4 bytes.
-

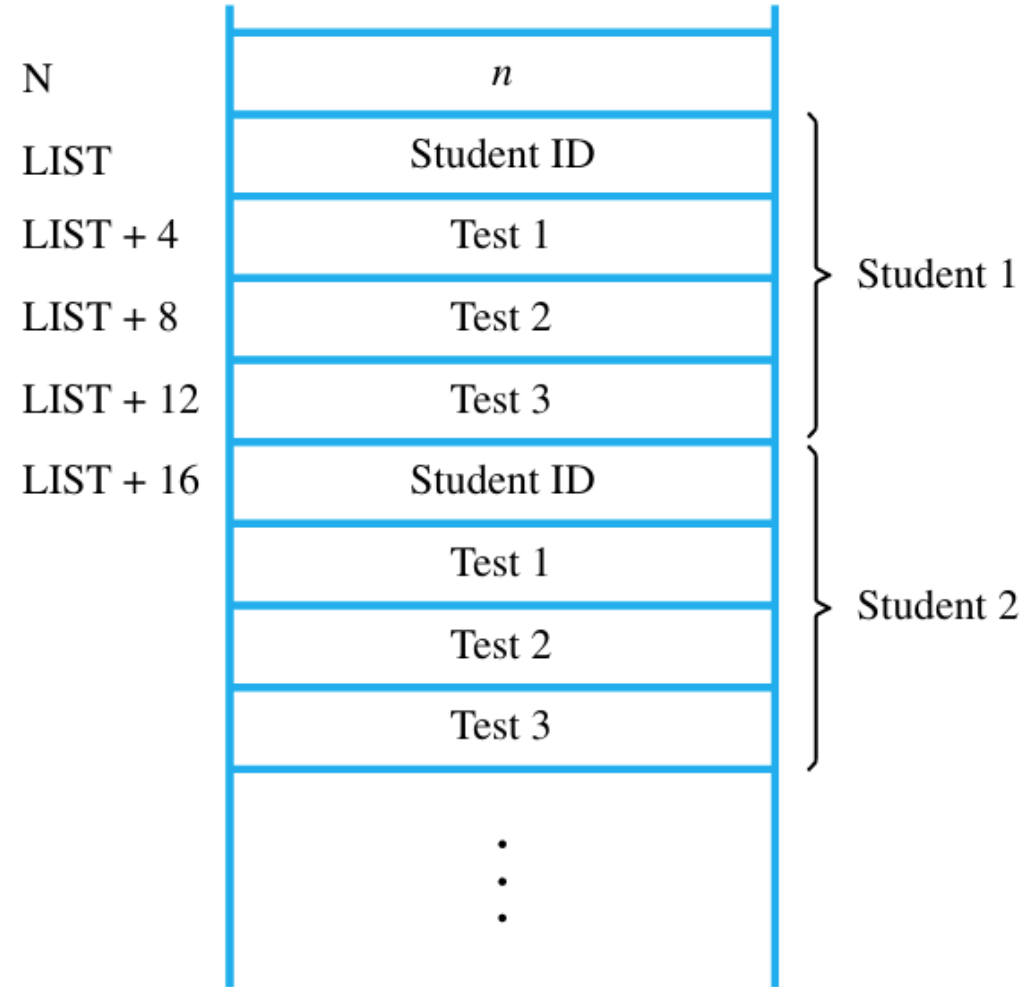
# Indexing and Arrays

- Accessing the test scores:
- 4(R2) → First test score
- 8(R2) → Second test score
- 12(R2) → Third test score

Program Example - Sum of Test Scores

- Load R5, 4(R2) ; Load first test score
- Load R4, 8(R2) ; Load second test score
- Load R3, 12(R2) ; Load third test score
- Add R2, R2, #16 ; Move to next record

The loop processes each record until R6 Reaches 0



**Figure 2.10** A list of students' marks.

# Indexing and Arrays

- student record.
- R3, R4, R5 hold accumulated sums of each test score.
- R6 holds the number of students  $n$  and is decremented with each iteration.
- **Why Index Mode is Useful**
- **Efficient Access** – The offset allows easy navigation through structured data.
- **Flexibility** – The same instruction can process multiple records by adjusting the index register.
- **Reusability** – Code remains generic and adaptable to different data sets.
- **Structured Data Handling** – Ideal for arrays, tables, or multi-field records

# Base with Index Mode

address of the operand is calculated by adding the contents of two registers together – one acting as the **base** and the other as the **index**.

- $EA = [Base\ Register] + [Index\ Register]$   
or
- $(R_i, R_j)$  The effective address is the sum of two registers  $\rightarrow EA = [R_i] + [R_j]$
- `MOV R1, ([R2], [R3])`
- R2 holds the base address (for example, the start of a record).
- R3 holds the index or offset to reach a specific field inside the record.
- The effective address is computed by adding both registers  $\rightarrow EA = R2 + R3$

# Base with Index & Offset Mode

- In **Base with Index & Offset mode**, the effective address is calculated by adding a constant value (offset) to the sum of two registers – one base and one index.
- $EA = X + [Base\ Register] + [Index\ Register]$  or
- $(X(R_i, R_j))$  Combines two registers and a constant  $\rightarrow EA = X + [R_i] + [R_j]$
- `MOV R1, 8([R2], [R3])`
- R2 holds the base address (for example, the start of a data structure).
- R3 holds the index or offset from that base.
- 8 is a constant offset added to reach a specific part within the structure.
- The effective address is  $EA = R2 + R3 + 8$



# Auto-Increment Mode

operand is accessed first from the memory location pointed to by a register, and then the register is automatically incremented by a predefined amount (usually the size of the data type, such as 4 bytes for a 32-bit word).

- **How it works**
- The processor reads the value from the memory location stored in the register.
- After accessing the operand, the register is incremented so that it points to the next memory location.

# Auto-Increment Mode

- `MOV R1, (R2) +`
- Here R2 holds the address of the current operand.
- The processor accesses the value at the address stored in R2 loads into R1 and then increments R2 by 4 (if the data is 32 bits).

Before execution

R2 = 1000

Memory[1000] → 5

After execution

R2 = 1004

Value 5 is loaded into R1

# Auto-Decrement Mode

- **Definition:**  
In **Auto-Decrement mode**, the register is decremented by a predefined amount before accessing the operand. The processor adjusts the register first, then accesses the data at the updated address.
- **How it works**
  - The register is decremented first to point to the previous memory location.
  - Then the operand is read from the new memory address.

# Auto-Decrement Mode

- `MOV R1, -(R2)`
- Here, R2 is decremented by 4 before being used to access the memory.
- The operand is then loaded into R1

Before execution

R2 = 1004

Memory[1000] → 5

After execution

R2 = 1000

Value 5 is loaded into R1

Feature	Auto-Increment	Auto-Decrement
Register update	After operand access	Before operand access
Usage	Forward iteration	Reverse iteration or stack operations
Example syntax	$(R2) +$	$-(R2)$
Efficiency	Traversing lists, buffers	Traversing backward, popping stack

# Stack

---

## What a stack is

---

A **stack** is just a way to organize data where you can only insert or remove items from one end — the **top**.

---

Think of a **pile of trays** in a cafeteria. You always put new trays on top and take trays off from the top.

---

This gives the property **LIFO** (last-in, first-out). The last item you pushed is the first item you'll pop out.

# Stack

- **Push** → Add a new item onto the stack.
- **Pop** → Remove the top item from the stack.
- **Top** → Where pushes and pops happen.
- **Bottom** → The end of the stack that you never directly access.



# Stack

- The **stack lives inside main memory**. It's not some special hardware box; it's just a region of RAM set aside for stack operations.
- The **Stack Pointer (SP)** is a register in the processor. Its job is to always keep track of *where the top of the stack is* right now.
- Each item in the stack sits in a memory location. The elements are laid out one after another in memory.
- By convention (and in most real machines), the **stack grows downward** in memory. That means:
  - The **bottom** of the stack is at a higher memory address.
  - Every time you **push**, SP moves to a slightly *lower* address, and the data is written there.
  - Every time you **pop**, SP moves *upward* again to a higher address.



# Stack

- Example
- Assumptions:
- Memory is byte-addressable.
- A word = 4 bytes.
- Stack grows downward (to smaller addresses)
- SP (Stack Pointer) points to the current top element.
- Address      Value
- 110            43            ← bottom
- 106            739
- 102            17
- 98             -28            ← current top, so  
  SP = 98

# Stack

- A word stored at address 98 occupies bytes 98,99,100,101. A word at address 94 would occupy bytes 94-97, and so on.
- **PUSH 55**
- Subtract SP, SP, #4 ;  $SP := SP - 4$
- Store Rj, (SP) ;  $mem[SP] := Rj$
- **1) Before: SP = 98 (points to -28)**
- **2) Subtract SP, SP, #4**  
Compute  $SP_{new} = SP_{old} - 4$   
 $98 - 4 = 94$   
Now  $SP = 94$
- **3) Store Rj, (SP)**
- Write the value in Rj (which is 55) into memory at address SP (94)
- Memory at 94-97 now holds the word 55.
- After the store, the top of the stack is the new word at address 94, and SP correctly points to it.

# Stack

- **Stack after push:**

- Address      Value

- 110            43

- 106            739

- 102            17

- 98             -28

- 94             55

SP = 94

← new top,



# Stack

- POP (take the top off and put it into Rj)
- Load Rj, (SP) ; Rj := mem[SP]
- Add SP, SP, #4 ; SP := SP + 4
- 1) **Before pop: SP = 94 and mem[94] = 55**
- 2) **Load Rj, (SP)**
- Read the word at address SP (94) into Rj
- So Rj now contains 55
- We have retrieved the top item but we haven't changed SP yet.
- 3) **Add SP, SP, #4**
- Compute  $SP_{new} = SP_{old} + 4$
- $94 + 4 = 98$
- Now SP = 98 that makes the top element again the word at address 98 (-28).

# Stack


- Stack after pop
- Address      Value
- 110            43
- 106            739
- 102            17
- 98             -28             $\leftarrow$  SP = 98  
    (current top)
- 94             55             $\leftarrow$  still in  
    memory, but considered  
    free/unused
- Important: the bytes at 94 still  
    physically contain 55 until  
    something overwrites them. But  
    by convention the stack now  
    treats that slot as free because  
    SP no longer points to it.

# Stack

- A **normal push** just subtracts 4 from SP and stores the new item.
- A **normal pop** just loads the item at SP and then adds 4.
- But what if:
- You **push when the stack is already full** (no more reserved memory)? → You overwrite something else in memory. That's called **stack overflow**.
- You **pop when the stack is already empty** (SP has reached the bottom)? → You read garbage or cause a crash. That's called **stack underflow**.




# Safe PUSH

- 
- Before pushing:
  - Check if there is space left in the stack area.
  - If there is space → do the normal push.
  - If not → raise an error or stop the operation.
  - So a **safe push = check for overflow + then push.**



# Safe POP

- 
- Before popping:
  - Check if the stack has at least one element (i.e., SP is not already at the bottom).
  - If the stack is empty → raise an error or stop.
  - If not empty → do the normal pop.
  - So a **safe pop = check for underflow + then pop.**



# Subroutine

- **What is a subroutine?**
- A subroutine is simply a small block of code that performs a specific task.
- Instead of writing the same set of instructions again and again in a program, we write it once as a subroutine and "call" it whenever needed.
- When a program calls a subroutine, the CPU needs to **remember where to come back** after the subroutine finishes. This is where the **stack** comes in.



# Subroutine

- Suppose your program needs to calculate the square of numbers in multiple places. Instead of repeating the square calculation code everywhere, you put it in a subroutine called `Square()`
- Then, whenever you need the square, you just call `Square()`
- **Why do we use subroutines?**
- Saves memory (one copy of code is stored instead of repeating it everywhere).
- Makes programs easier to read and maintain.
- Encourages code reusability.

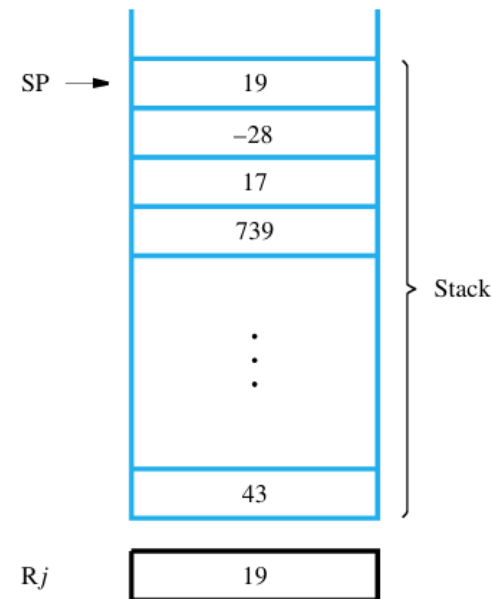


# Subroutine

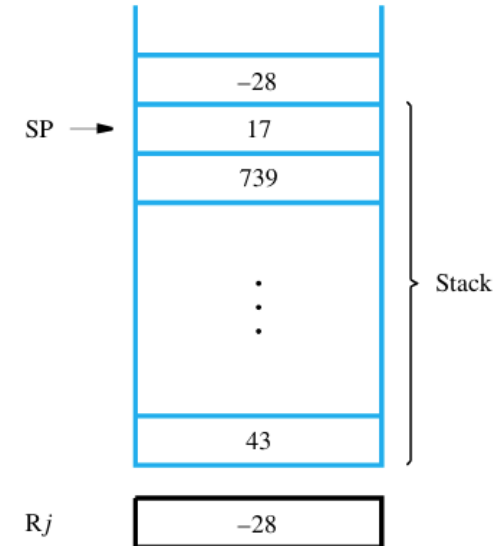
- **How do subroutines work?**
- The main program **calls** the subroutine using a **Call instruction**.
- After finishing, the subroutine must **return** to the exact line where it was called.
- This is where the **Program Counter (PC)** and **Link Register (LR)** come in:
- **PC (Program Counter)**: Always points to the next instruction to execute.
- **Link Register**: Temporarily stores the return address (the place in the main program where execution should continue after the subroutine finishes).

# Subroutine

- **(a) After PUSH from Rj**
- A value (here 19) from register Rj is placed on top of the stack.
- The **stack pointer (SP)** moves to point to this new top value.
- Now the top of the stack is 19
- **(b) After POP into Rj**
- The top value (-28) is removed from the stack.
- The **SP** moves down to the next value, which is 17
- That popped value (-28) is copied into register Rj



(a) After push from Rj



(b) After pop into Rj



# Subroutine

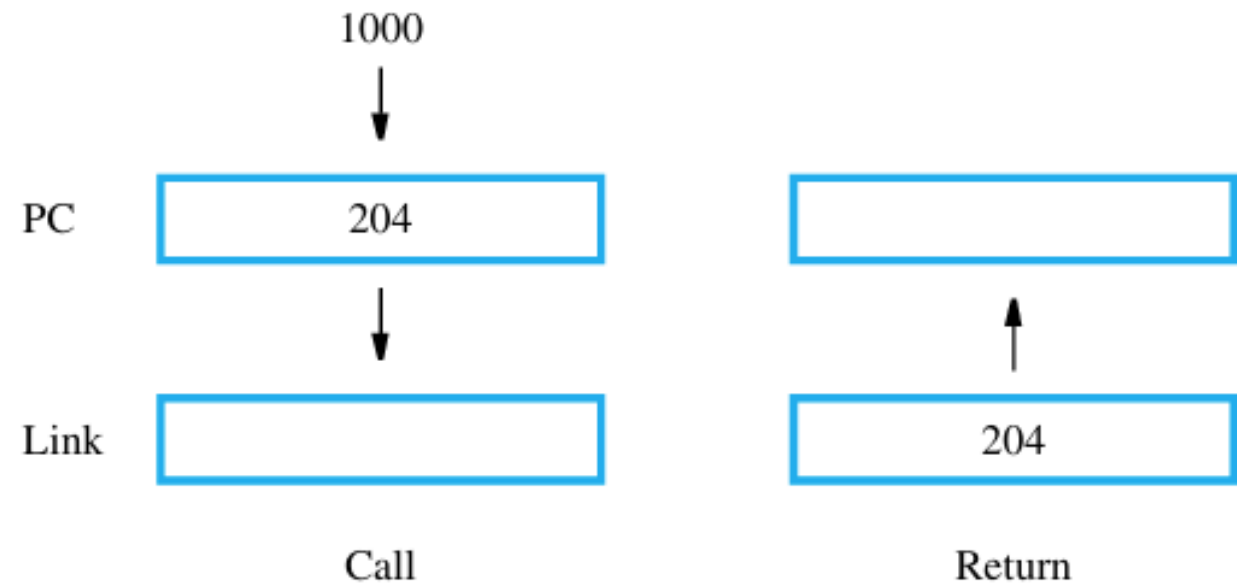
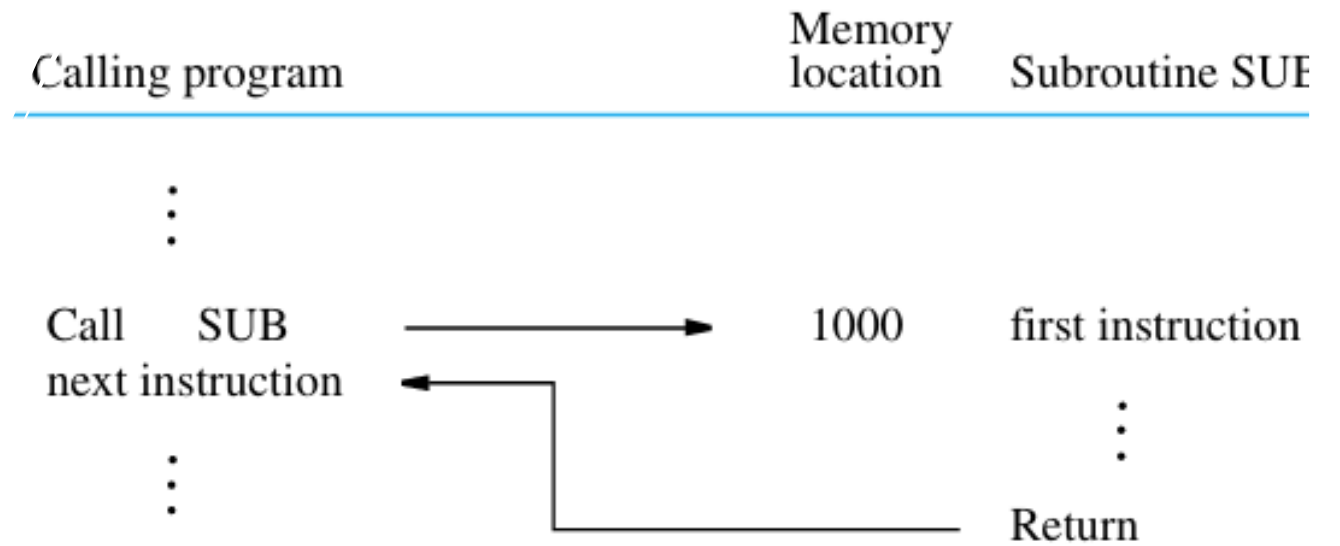
- **Call Instruction**
- When the program calls a subroutine:
- The **current PC value** (address of the next instruction in the main program) is saved into the **Link Register**.
- The PC is updated to the starting address of the subroutine.



# Subroutine

- **Return Instruction**
- When the subroutine finishes:
- The CPU **loads the address from the Link Register back into the PC.**
- This makes the program continue from where it left off.

# Subroutine





# Subroutine Nesting

- **What is Subroutine Nesting?**
- Normally, a subroutine is called from the main program, does its work, and then returns.
- But sometimes, **one subroutine may need to call another subroutine** before it finishes.
- This situation is called **subroutine nesting**.
- Example:
- Main Program → calls **SUB1**
- Inside **SUB1**, another call is made to **SUB2**





# Subroutine Nesting

- **The Problem with Only a Link Register**
- When a subroutine is called, the CPU saves the **return address** (the place to continue after the subroutine finishes) in the **Link Register (LR)**.
- If **SUB1** calls **SUB2**:
  - First, the return address of **SUB1** (to go back to main) is stored in the LR.
  - Then, when **SUB1** calls **SUB2**, the LR gets overwritten with a new return address (to go back to SUB1).
- The old return address (back to main) is lost.
- Result: When SUB1 tries to return, it no longer knows where to go.



# Subroutine Nesting

- **Why the Processor Stack is Needed**
- To solve this, the contents of the Link Register must be **saved** somewhere safe before calling another subroutine.
- The ideal place is the **processor stack**, because:
  - The stack follows **Last-In, First-Out (LIFO)** order.
  - This matches the way subroutine calls work: the most recent call must finish and return first.




# Subroutine Nesting

- **How It Works Step by Step**
- **Main calls SUB1**
  - Return address to main is placed in the Link Register.
  - SUB1 starts executing.
- **SUB1 calls SUB2**
  - Before making this call, SUB1 saves its return address (from LR) **onto the stack**.
  - Now LR can safely be overwritten with SUB2's return address.
- **SUB2 finishes**
  - Executes Return → uses LR to go back to SUB1.
- **SUB1 finishes**
  - Before returning, SUB1 **pops** its old return address from the stack back into LR.
  - Now SUB1 can return correctly to the main program.




# Subroutine Nesting

- **Why This is Important**
- Subroutine nesting can go to any depth: SUB1 → SUB2 → SUB3 ... and so on.
- Each time, the current return address is **pushed** onto the stack before the next call.
- When returning, addresses are **popped** in reverse order, ensuring correct sequencing.




# Parameter Passing

- **What is Parameter Passing?**
- When a program calls a subroutine, it often needs to give that subroutine **inputs (parameters)**.
- Inputs can be numbers, addresses, or anything the subroutine needs to compute.
- After the subroutine finishes, it may return results (also parameters).
- This whole exchange is called **parameter passing**.



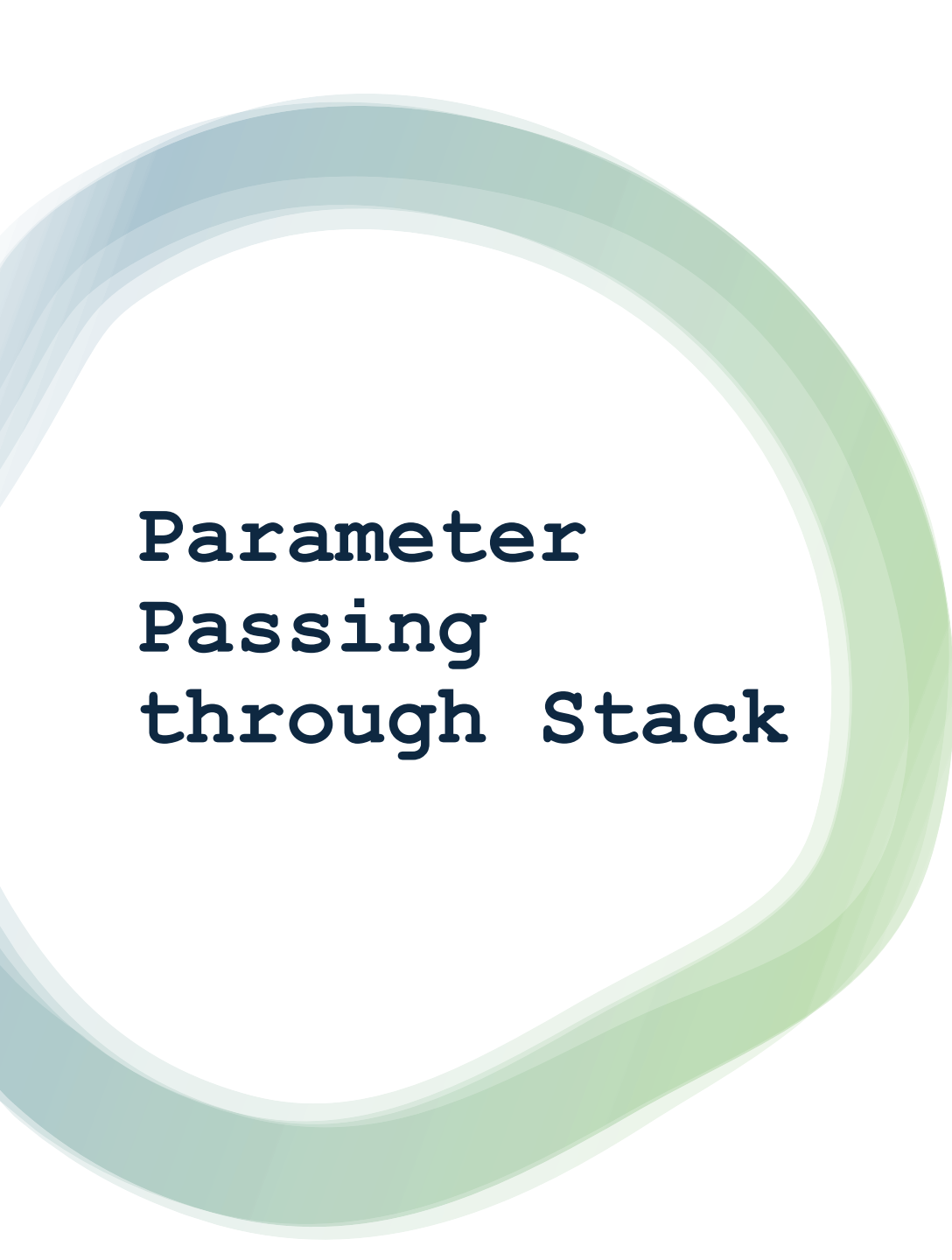
# Parameter Passing

- **Ways to Pass Parameters**
- There are **three common methods**:
- **Registers**
  - Parameters are put directly into CPU registers.
  - Simple, very fast.
  - But limited: only works if you have a few parameters, since registers are scarce.
- **Memory variables**
  - Parameters are stored in fixed memory locations.
  - Subroutine knows where to look.
  - Flexible, but slower than registers.
- **Stack**
  - Parameters are pushed onto the stack before the call.
  - Subroutine pops them off when needed.
  - Allows *any number* of parameters, very flexible.
  - Also fits naturally with nested subroutine calls.



# Parameter Passing through Registers

- Add up a list of numbers.
- **Calling program does this:**
- Load `n` (list size) into register `R2`.
- Load `NUM1`(address of first number in the list) into register `R4`.
- Call the subroutine `LISTADD`
- **Subroutine `LISTADD` does this:**
- Clears `R3 = 0` (this will hold the sum).
- Loops `n` times:
- Loads the next number using `R4` (the pointer).
- Adds it to `R3`.
- Increments pointer `R4`, decrements counter `R2`.
- Returns with result in `R3`.
- **After return:**
- The calling program stores the value of `R3` into memory (`SUM`).
- Here parameters are:
- Passed *into* subroutine `n` in `R2` `NUM1` in `R4`
- Passed *back* from subroutine: sum in `R3`



# Parameter Passing through Stack

- **Example with Stack**
- Same task, but using the **stack**.
- **Before calling:**
- Push NUM1 (address of first number).
- Push n (list size).
- Call LISTADD
- **Inside LISTADD:**
- First thing: save registers it will use (R2-R5) by pushing them onto the stack.
- Get n and NUM1 from stack into R2 and R4.
- Clear R3 = 0.
- Loop over the list like before.
- When done, put the result (sum in R3) back into the stack, overwriting the NUM1 slot.
- Restore saved registers from the stack.
- Return to caller.
- **After return:**
- The calling program pops the result (sum) from the stack and stores it into SUM.
- Cleans up the stack pointer to its original position.
- Here parameters are:
- Passed *into* subroutine: n and NUM1 via stack
- Passed *back* from subroutine: sum, also via stack





# Memory Unit

# Memory Unit Basic Concepts

- **Address size decides max memory**

- 16-bit address  $\Rightarrow 2^{16}$  locations = **64 KB**

- 32-bit  $\Rightarrow 2^{32}$  locations = **4 GB**

- 64-bit  $\Rightarrow 2^{64}$  locations

- **Byte-addressable example**

A 32-bit address points to individual bytes. The upper 30 bits choose the 4-byte word, and the lower 2 bits select the exact byte inside that word.

- **Processor–memory link**

- **Address lines** specify which location to access.

- **Data lines** carry the data being read or written.

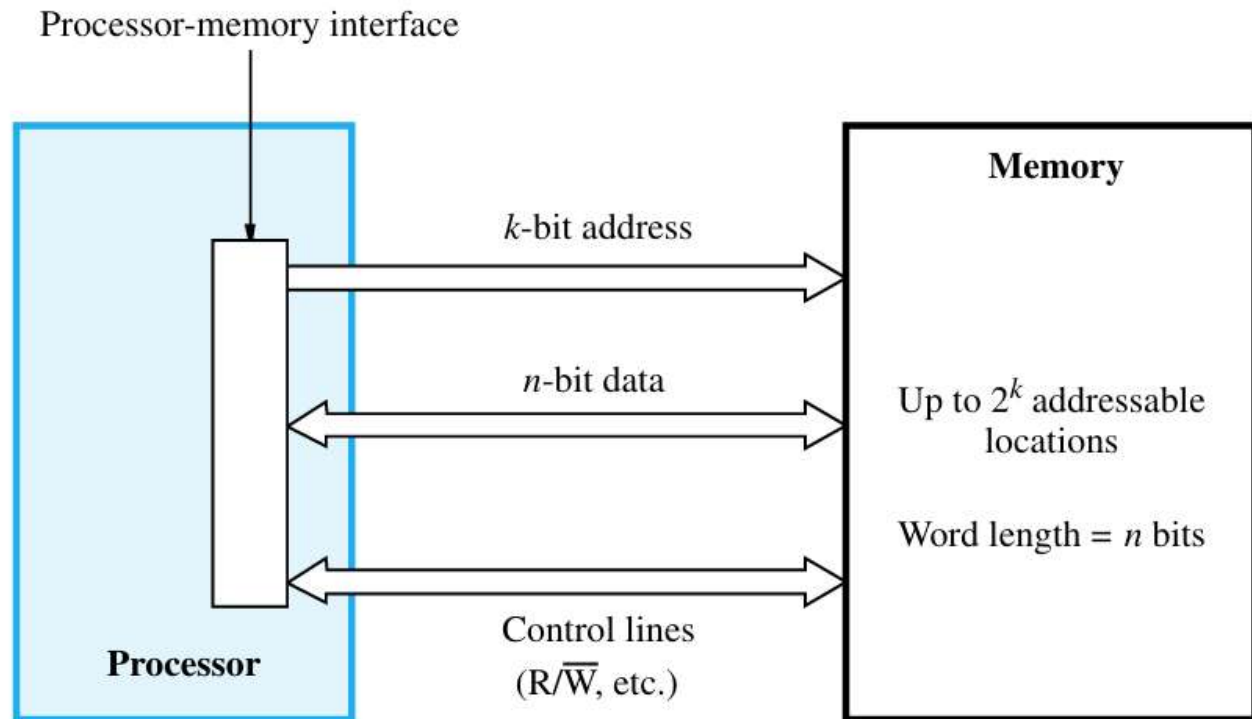
- **Control lines** indicate read/write and handle timing.

- **MFC signal**

After memory finishes the operation it asserts the “Memory Function Complete” (MFC) signal, telling the processor it can continue.

# Basic Concepts

- In order to transfer data between memory & processor, we need some interface circuitry & is as shown in the fig.
- Data transfer between memory & processor is done through MAR & MDR
- MAR and MDR are two key registers that sit between the CPU and main memory:
- **MAR – Memory Address Register**  
Holds the *address* of the memory location the CPU wants to access.  
Think of it as the pointer saying “go to this spot.”
- **MDR – Memory Data Register**  
Holds the *data* moving to or from that address.
  - During a **read**, the memory puts the requested data into the MDR so the CPU can use it.
  - During a **write**, the CPU places the data in the MDR, and the memory system takes it from there.





# Basic Concepts

- Measures for the speed of a memory:
- **Memory access time** - How long it takes from starting a read or write until the data transfer finishes.
- **Memory cycle time** - Minimum gap between two back-to-back operations, like two reads. It's usually a bit longer than access time.
- An important design issue is to provide a computer system with as large and fast memory as possible, within a given cost target.
- Several techniques to increase the effective size and speed of the memory:
- **Cache memory** - A small, very fast layer of memory that sits between the CPU and the slower main RAM. It keeps the instructions and data the processor is using right now, so the CPU doesn't have to wait on slower main memory.
- **Virtual memory** - Lets a computer pretend it has more RAM than it really does. Only the active parts of a program stay in physical RAM. The rest stays on the hard drive or SSD and is swapped in and out



# The Memory System

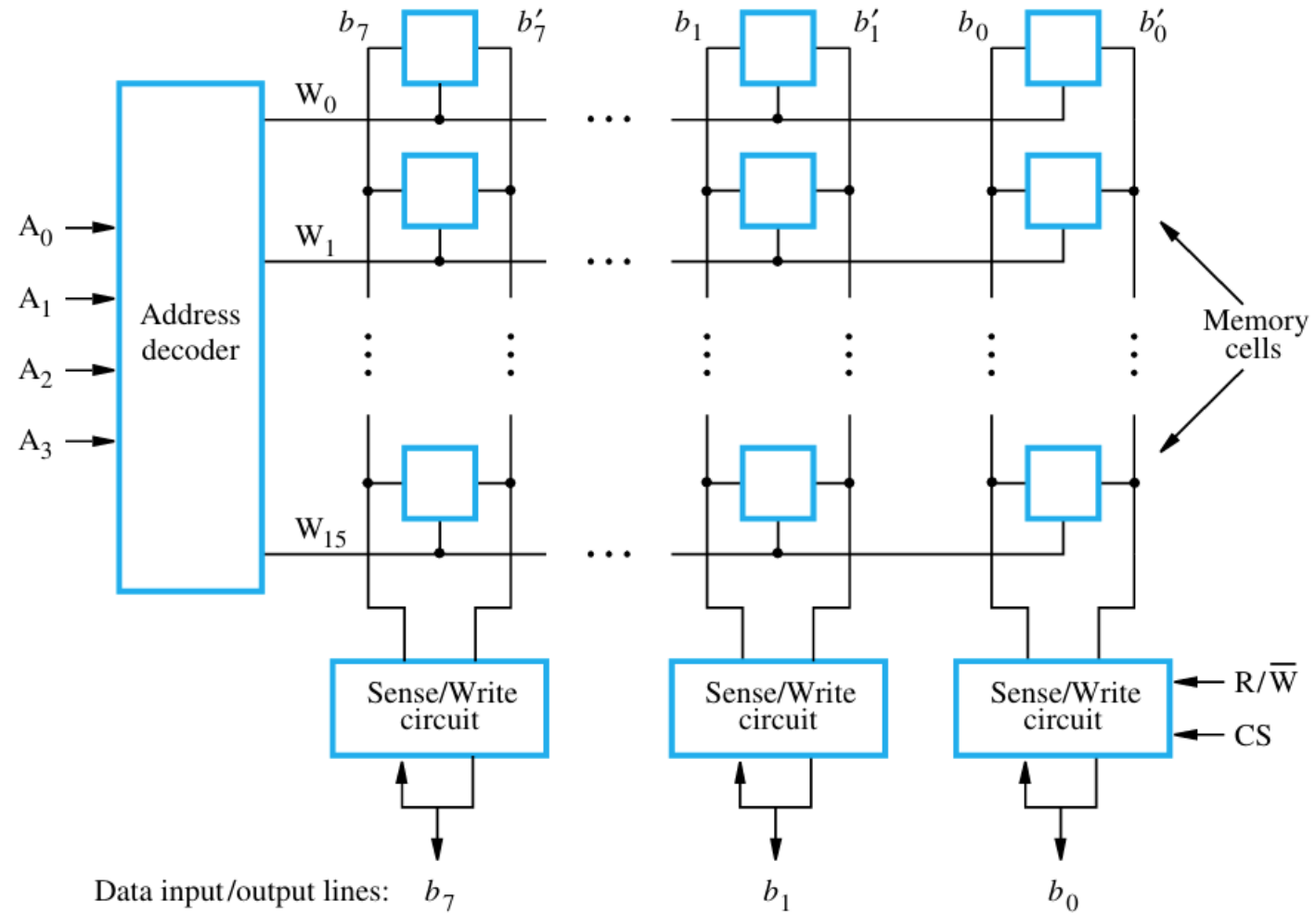
## Semiconductor RAM

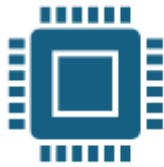
### memories

# Internal Organization of Memory Chips

- Each memory cell can hold one bit of information.
- Memory cells are organized in the form of an array.
- One row is one memory word.
- All cells of a row are connected to a common line, known as the "word line".
- Word line is connected to the address decoder.
- Sense/write circuits are connected to the data input/output lines of the memory chip.

Internal  
organiza  
tion of  
memory  
chips 16  
x 8  
memory  
chip





# Internal organization of memory chips 16 x 8 memory chip

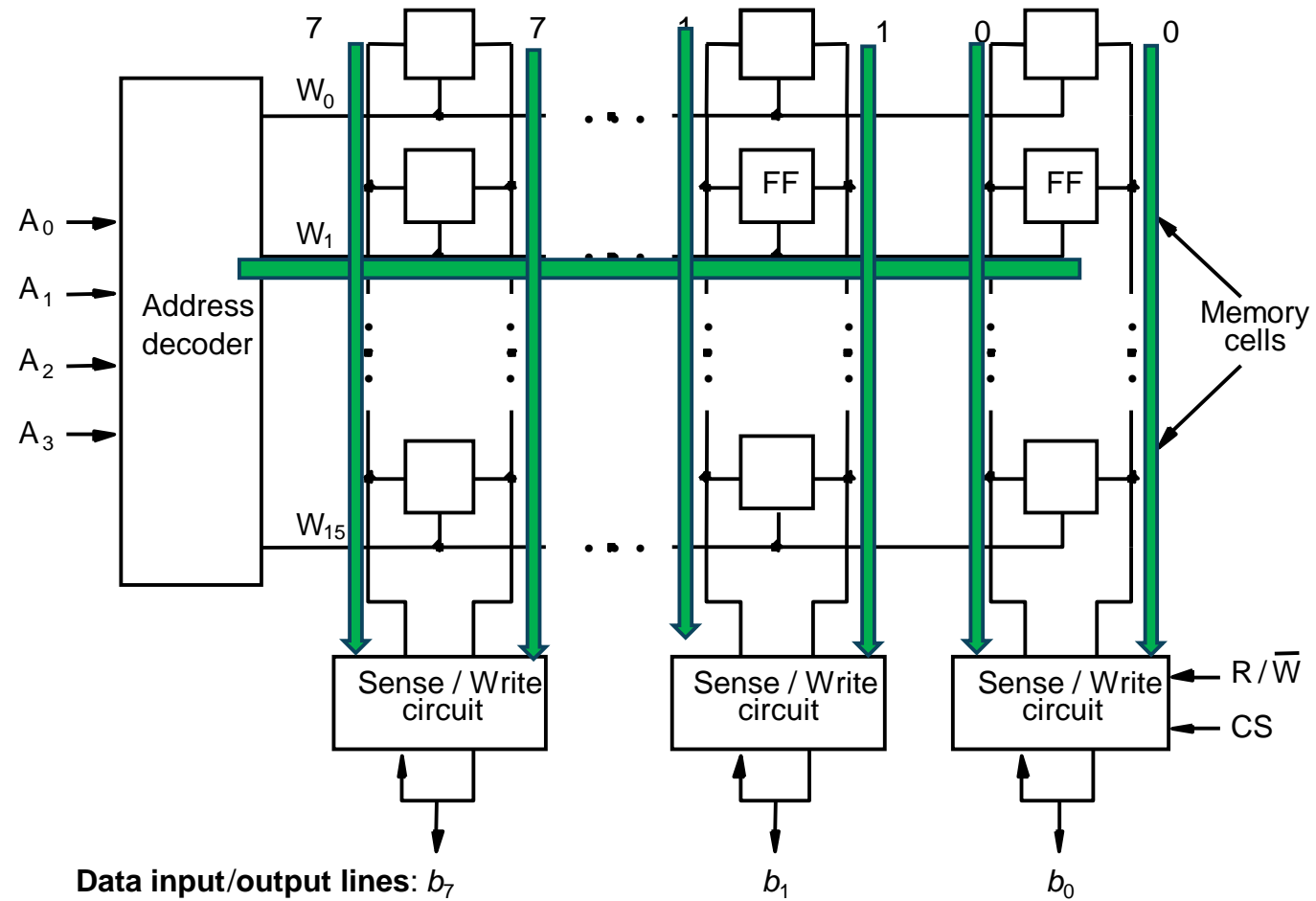
- This diagram shows the internal layout of a small static RAM chip—specifically a **16 × 8 memory** (16 words, each 8 bits). Here's what each part does:
- **1. Address inputs and decoder**
  - The four address lines **A0–A3** carry the binary address of the word you want ( $2^4 = 16$  possible rows).
  - The **address decoder** turns that 4-bit address into one active **word line** (W0...W15). Only the selected row is enabled.
- **2. Memory cell array**
  - Each square is a **1-bit memory cell**.
  - A horizontal **word line** activates an entire row.
  - Vertical **bit lines** ( $b_0, b_1 \dots b_7$ ) carry the data for each column. Each column also has a complementary line ( $b'_0, b'_1 \dots$ ) used by the sense amplifiers for reliable reads.
- **3. Sense/Write circuits**
  - At the bottom of each column, a **Sense/Write circuit** connects the bit line pair to the chip's external data pin (one pin per column, so 8 total).
  - **Read:** When a word line is active, the tiny voltage difference from the selected cells is amplified and put on the output pin.
  - **Write:** Data from the external pin drives the bit lines to set the selected cells.



# Internal organization of memory chips 16 x 8 memory chip

- **4. Control signals**
- **R/  $\overline{W}$**  decides if the chip is reading or writing.
- **CS (Chip Select)** enables the whole chip when it's part of a larger memory system.
- **How a read works step by step**
- The CPU places a 4-bit address on A0-A3 and asserts CS with R/  $\overline{W}$  high.
- The decoder activates the matching word line, say W5.
- All 8 cells in row 5 connect to their bit lines.
- Each Sense circuit detects the stored 0 or 1 and drives the external data pins b<sub>0</sub>...b<sub>7</sub>.
- **Write is similar**, but with R/  $\overline{W}$  low: the Sense/Write circuits force the bit lines to the input values, overwriting that row.

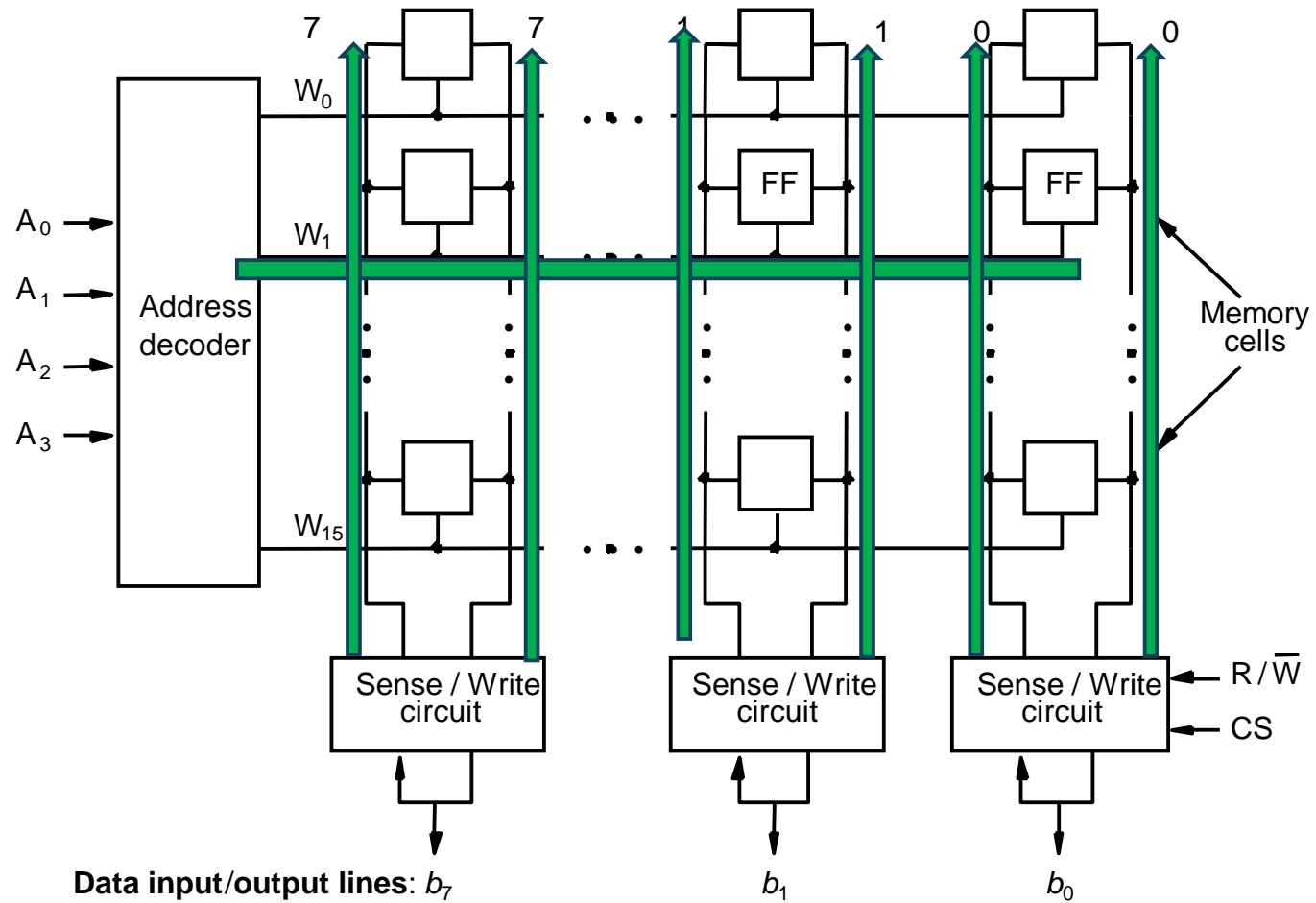
# Read Operation



# Read Operation

- Highlighted paths show the “active highway” for data to travel out:
- **Address decode**  
The 4 address bits select one word line, say W1. That horizontal green line lights up in the diagram.
- **Cells drive the bit lines**  
Every cell in row W1 connects to its column's vertical bit line.  
The stored 0/1 in each cell places a small voltage on that bit line.
- **Sense circuits**  
At the bottom, the sense amplifiers detect those voltages and output b7...b0 to the data bus.
- So the green highlight shows the *selected row* and the *bit lines being sensed*.

# Write Operation



# Write Operation

- Here the green highlight shows the same selected row, but now the data flow is upward:
- **Address decode**  
Same as before: the chosen word line (say W1) is activated.
- **Write enable (R/ W = Write)**  
The control signal switches the sense/write circuits into *write mode*.
- **Driving the bit lines**  
The external data bits b7...b0 are forced onto the vertical bit lines.
- **Cells updated**  
Because W1 is active, only those cells in row W1 are connected, so each cell captures the voltage on its bit line and stores the new 0/1.
- The highlight shows the active word line and the bit lines carrying the *incoming* data.

- The highlight visually tracks the **path of action**:
- During **read**, charge flows *from* the selected cells *down* the bit lines to the sense circuits.
- During **write**, signals travel *up* the bit lines into the selected cells.

# Internal organization of memory chips

**words .**

Total bits = (number of words) × (bits per word) .

External connector count (ignoring Vcc/GND) =

**address lines + data lines + control lines .**

- **Address lines** pick one word out of many. If there are  $W$  words, you need  $\log_2(W)$  address lines (because each address bit doubles the choices) .
- **Data lines** = bits per word (how many bits you read/write at once) .
- **Control lines** are signals like Chip-Select (CS) and Read/Write (R/ W) .

# Internal organization of memory chips

- The memory circuit stores 128 bits and requires 14 external connections for address, data, and control lines
- Choose words = 16 and bits/word = 8 (because  $16 \times 8 = 128$ ).
- Words = 16  $\rightarrow$  address lines =  $\log_2(16) = \mathbf{4}$
- Data lines = **8**
- Control lines = **2**
- Total =  $4 + 8 + 2 = \mathbf{14}$   
(matches)

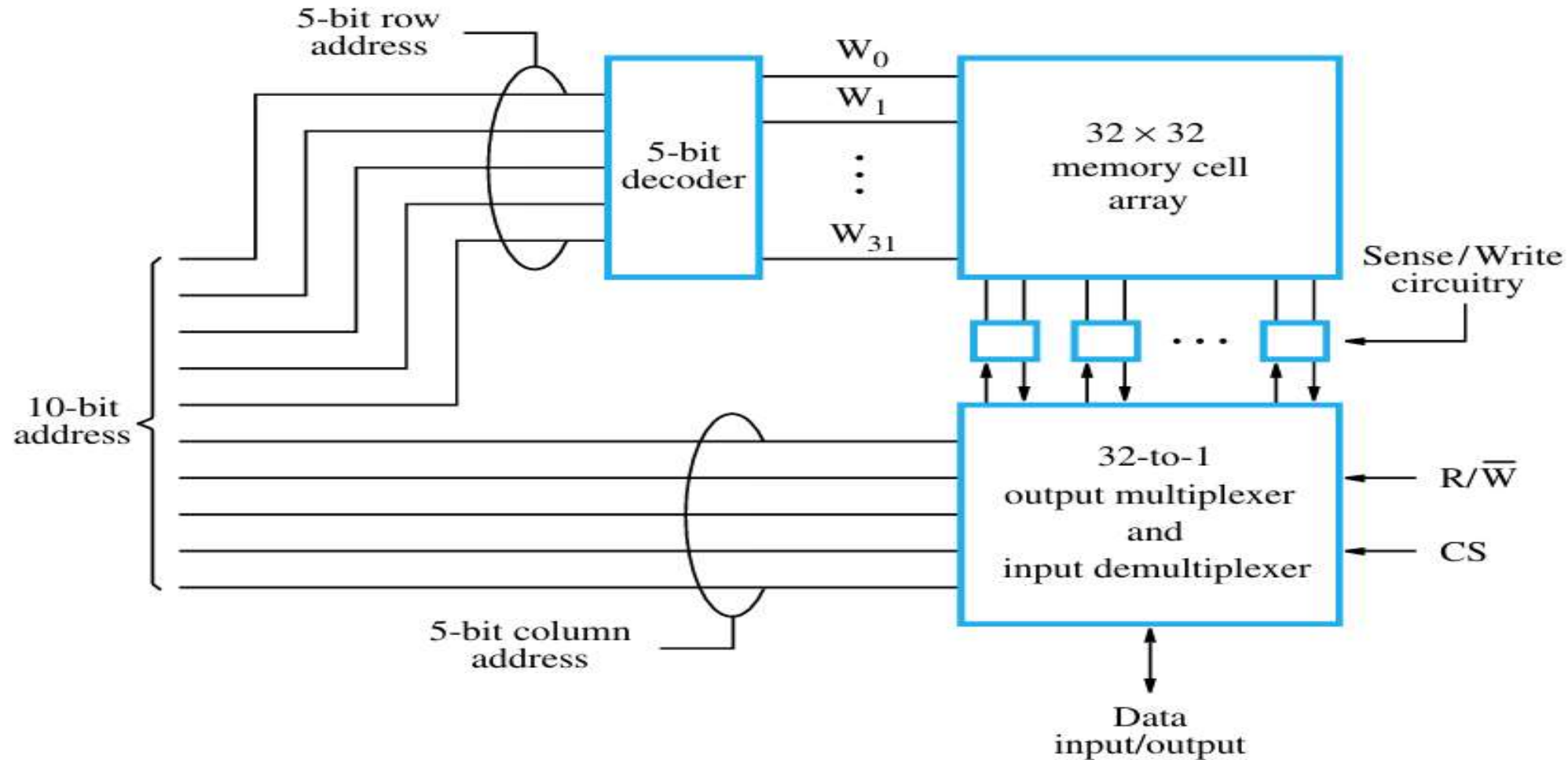


Bigger memory:  $1\text{K cells} = 1024$   
bits

- **128 × 8 organization**

- Words = 128 (because  $128 \times 8 = 1024$ )
- Address lines =  $\log_2(128) = 7$
- Data lines = **8**
- Control lines = **2**
- Total external connections =  $7 + 8 + 2 = 17$
- With Vcc & GND → **19 pins**
- So a 128×8 layout needs 7 address pins and 8 data pins.

# Organization of a $1K \times 1$ memory chip



1K × 1  
organization

- Words = 1024 (1 bit per word)
- Address lines =  $\log_2(1024) = \mathbf{10}$
- Data lines = **1**
- Control lines = **2**
- Total external connections =  $10 + 1 + 2 = \mathbf{13}$
- With Vcc & GND → **15 pins**



# The Memory System

Read-Only Memories (ROMs)

# Read-Only Memories (ROMs)

- **Volatile vs non-volatile**
- **SRAM** (static RAM) and **SDRAM** (synchronous dynamic RAM) are volatile. They forget everything the moment power is lost. That's why the contents of your laptop's main memory disappear when you shut it down.
- If you want data to survive a power cut, you need **non-volatile** memory.
- **ROM as non-volatile storage**
- **ROM** (Read-Only Memory) is the classic example. Once written, its data stays even with no power. Flash memory, EEPROM, and the storage inside SSDs are all modern non-volatile technologies.

# Read-Only Memories (ROMs)

- **Why we still need disks or flash**
- Many programs and operating systems are too large to fit permanently in a small ROM chip. So computers keep most of their software on big non-volatile storage like hard drives or SSDs.
- **Bootstrapping the system**
- When you power on a computer, the operating system isn't yet in RAM.
- A tiny piece of code—the **boot program** or **firmware**—lives in non-volatile memory on the motherboard
- This boot code runs first, knows how to talk to the disk or SSD, and copies the operating system into main memory.
- Only after that does the OS take over and start everything else.

# Read-Only Memories (ROMs)

- Power on.
- CPU executes the boot code stored in ROM/flash.
- Boot code loads the operating system from long-term storage (hard disk, SSD) into volatile RAM.
- OS runs, and now the system is ready for you.
- Generally, non-volatile memory is normally used for only reading operation, hence they are called Read Only Memory (ROM) .
- To store information in ROM, special writing process is needed.

# Read-Only Memories

- **Plain ROM**
- The data is fixed at the factory.
- You can't change it later.
- Good for mass-produced devices where every chip holds the same program.
- **Why that can be a problem**
- If each customer needs different data—say, a custom lookup table—you'd have to order a custom batch of chips. That's expensive



# PROM (Programmable Read-Only Memory)

- Starts life with every bit set to **0**.
- You can “program” it **once** yourself after purchase.
- Programming is done by sending a high current through tiny fuses in the chip. Blowing a fuse flips the stored bit from 0 to 1.
- Once a fuse is blown, it’s permanent—there’s no way to reset it. That’s why it’s called *one-time programmable*.

# Erasable Programmable Read- Only Memory (EPROM)

- **How it stores data**
- Each bit is held by a tiny transistor with a floating gate that traps electrons.
- Programming injects electrons onto that gate, which stays charged for years with no power, so the data is non-volatile.
- **Erasing the chip**
- To clear it, you shine **ultraviolet (UV) light** through a little quartz window on top of the package.
- The UV light knocks the electrons off the floating gates, returning all bits to their "blank" state.
- This takes minutes, and you must physically remove the chip from the circuit to expose it.
- **Reprogramming**
- After erasure, you can program it again with a special programmer that applies high voltage to set new bits.
- **Why it mattered**
- During development you could update firmware repeatedly without ordering new ROMs.
- It kept its contents for many years without power, so it worked like a normal ROM once programmed.

# Electrically Erasable Programmable Read- Only Memory.

- **What it is**
- EEPROM = **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
- Like EPROM, it keeps data with the power off, but you can erase and rewrite it *electrically*.
- **Why it's better than EPROM**
- No UV lamp or chip removal.
- The system can update its own firmware or settings while the chip stays soldered to the board.
- **How it operates**
- Special high-voltage pulses tunnel electrons on or off the floating gates inside each memory cell.
- Reading uses normal logic voltage (for example 3.3 V or 5 V).
- Writing and erasing often need a slightly higher programming voltage supplied by the chip or an external source.
- **Drawbacks of classic EEPROM**
- Disadvantage is that, we need different voltages to erase, write & read the stored data.
- Another disadvantage is that, each byte has to be erased separately.
- Because of those limits, **Flash memory** was developed:

# Flash memory

- It is possible to Read the contents of a single cell, but not possible to write single cell.
- Here, we need to write entire block of cells at a time.
- It requires single supply voltage for read, write & erase.
- Flash devices have greater density. Higher capacity and low storage cost per bit.
- Power consumption of flash memory is very low, making it attractive for use in equipment that is battery-driven.
- Single flash chips are not sufficiently large, so larger memory modules are implemented using flash cards and flash drives.
- For larger storage you see flash combined into bigger modules:
- **Memory cards** (SD, microSD)
- **USB flash drives**
- **SSD drives** (essentially many flash chips managed by a controller)
-

# Speed, Size, and Cost

computer system is to provide a sufficiently large memory, with a reasonable speed at an affordable cost.

- **Static RAM:**

- Very fast, but expensive, because a basic SRAM cell has a complex circuit making it impossible to pack a large number of cells onto a single chip. Used in Cache

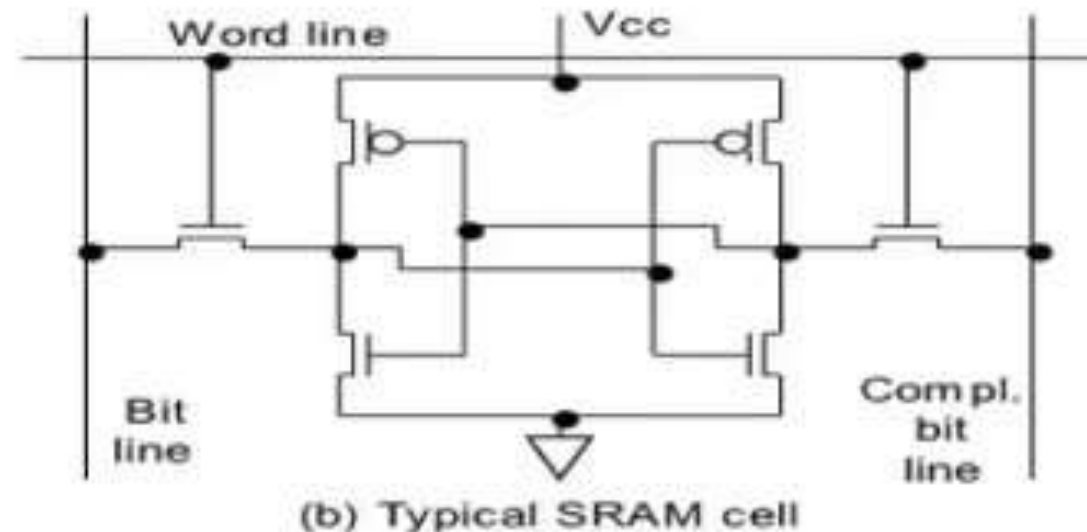
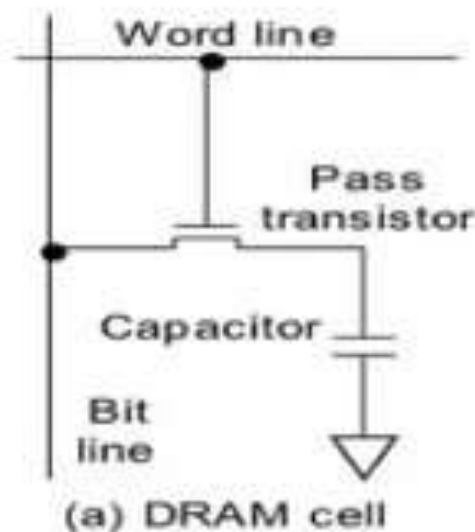
- **Dynamic RAM:**

- Simpler basic cell circuit, hence are much less expensive, but significantly slower than SRAMs. Used in Main Memory
- Magnetic disks:
- Storage provided by DRAMs is higher than SRAMs, but is still less than what is necessary.
- Secondary storage such as magnetic disks provide a large amount
- of storage, but is much slower than DRAMs.

# SRAM Vs DRAM

Static RAM	Dynamic RAM
<ul style="list-style-type: none"><li>➤ SRAM uses transistor to store a single bit of data</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM uses a separate capacitor to store each bit of data</li></ul>
<ul style="list-style-type: none"><li>➤ SRAM does not need periodic refreshment to maintain data</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM needs periodic refreshment to maintain the charge in the capacitors for data</li></ul>
<ul style="list-style-type: none"><li>➤ SRAM's structure is complex than DRAM</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM's structure is simplex than SRAM</li></ul>
<ul style="list-style-type: none"><li>➤ SRAM are expensive as compared to DRAM</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM's are less expensive as compared to SRAM</li></ul>
<ul style="list-style-type: none"><li>➤ SRAM are faster than DRAM</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM's are slower than SRAM</li></ul>
<ul style="list-style-type: none"><li>➤ SRAM are used in Cache memory</li></ul>	<ul style="list-style-type: none"><li>➤ DRAM are used in Main memory</li></ul>

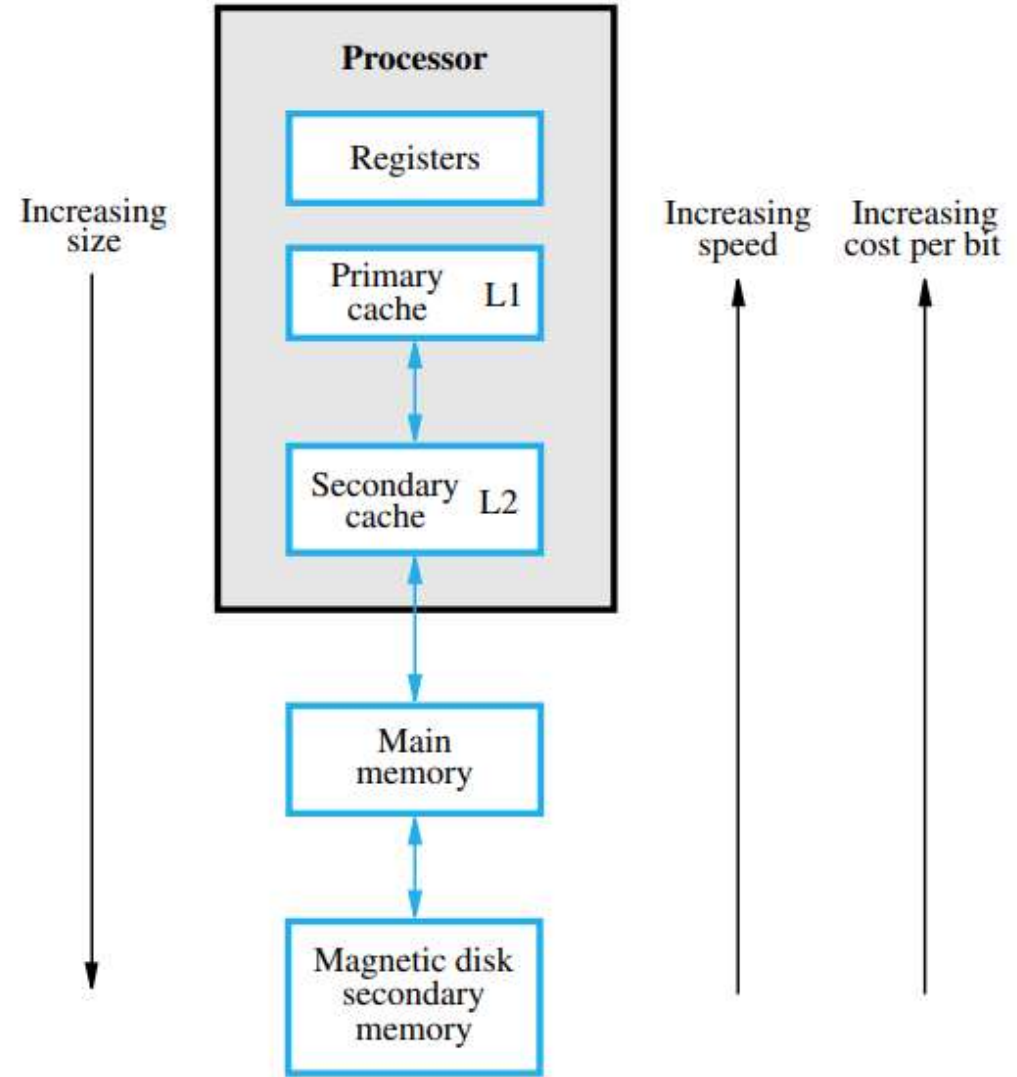
Below is the structure for SRAM and DRAM for 1 bit storage.



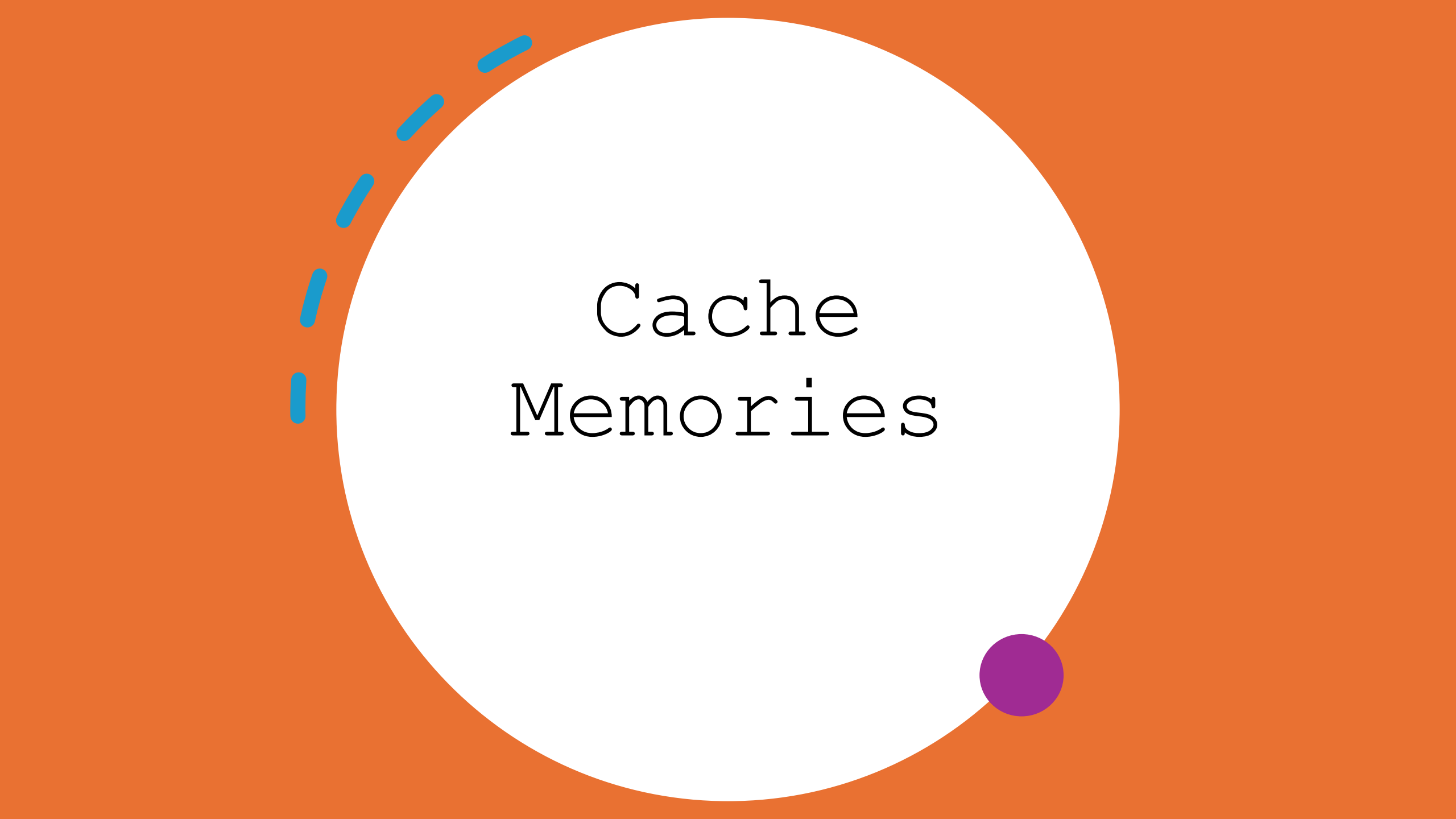
# Memory Hierarchy

- *Fastest access is to the data held in processor registers. Registers are at the top of the memory hierarchy.*
- *Relatively small amount of memory that can be implemented on the processor chip. This is processor cache.*
- *Two levels of cache. Level 1 (L1) cache is on the processor chip. Level 2 (L2) cache is in between main memory and processor.*
- *Next level is main memory, implemented as SIMMs a Single Inline Memory Module.. Much larger, but much slower than cache memory.*
- *Next level is magnetic disks. Huge amount of inexpensive storage.*
- *Speed of memory access is critical, the idea is to bring instructions and data that will be used in the near future as close to the processor as possible.*

# Memory Hierarchy







# Cache Memories

# Cache Memories

- The core problem here is a **speed mismatch** between the processor and main memory. Modern CPUs are extremely fast, but main memory (RAM) is comparatively slow. Without any optimization, the CPU spends a lot of time **waiting** for data or instructions to arrive from RAM. This waiting is called a **memory bottleneck** and is a major limitation on performance.
- **Main memory can't just be made infinitely fast**—technological and cost limits prevent that.
- **Cache memory** is a small, very fast memory placed between the CPU and main memory. Its job is to **store copies of data and instructions that the CPU is likely to use soon**, so the CPU can access them without waiting for the slower main memory.
- Cache takes advantage of **locality of reference**:

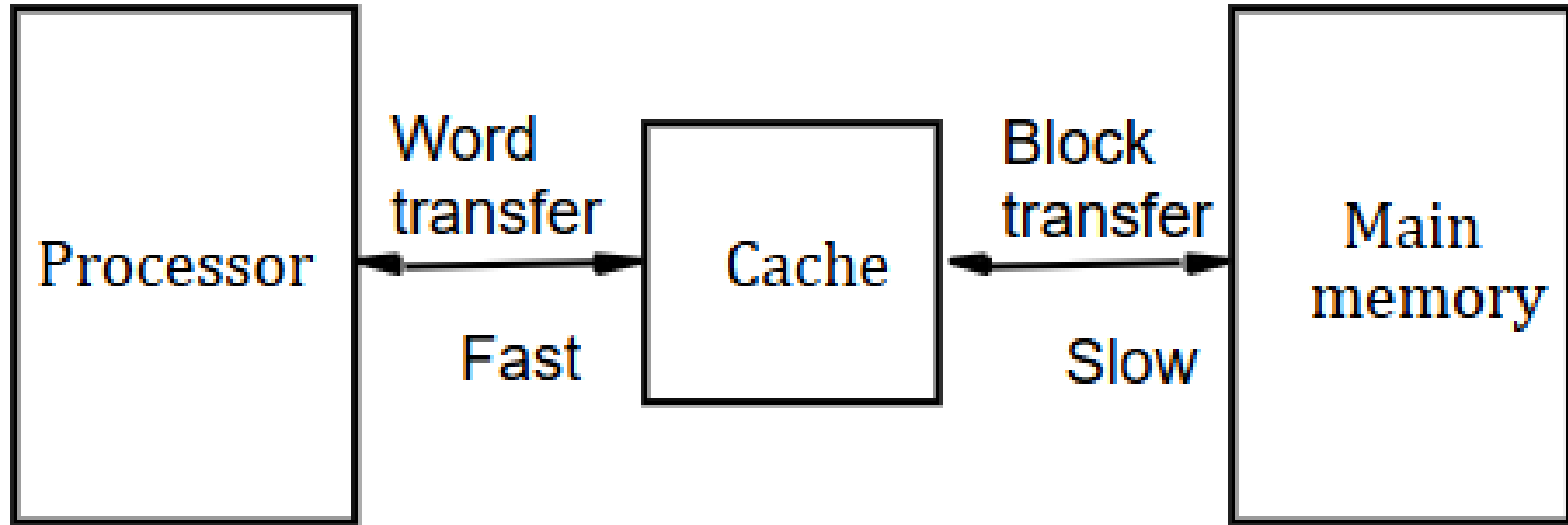
# locality of reference :

- instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently.
- These instructions may be the ones in a loop, nested loop or few procedures calling each other repeatedly.
  - This is called "locality of reference".
  - **Temporal locality:** If the CPU accesses a memory location, it's likely to access the same location again soon.
  - **Spatial locality:** If the CPU accesses a memory location, it's likely to access nearby memory locations soon.
  - By storing recently accessed data and nearby data in cache, the system effectively **makes the main memory appear faster** to the CPU, improving overall performance without actually speeding up RAM.

# Cache memories

- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- The cache is small compared to main memory, so at any moment it contains **only a subset of memory blocks**.
- Which main memory blocks are stored in the cache is determined by a **mapping function**. Common types of mapping include **direct-mapped, fully associative, and set-associative**.
- When the cache is full and a new block needs to be loaded, the system must **evict an existing block** to make space.
- Which block gets replaced is decided by a **replacement algorithm**

# Cache Memories



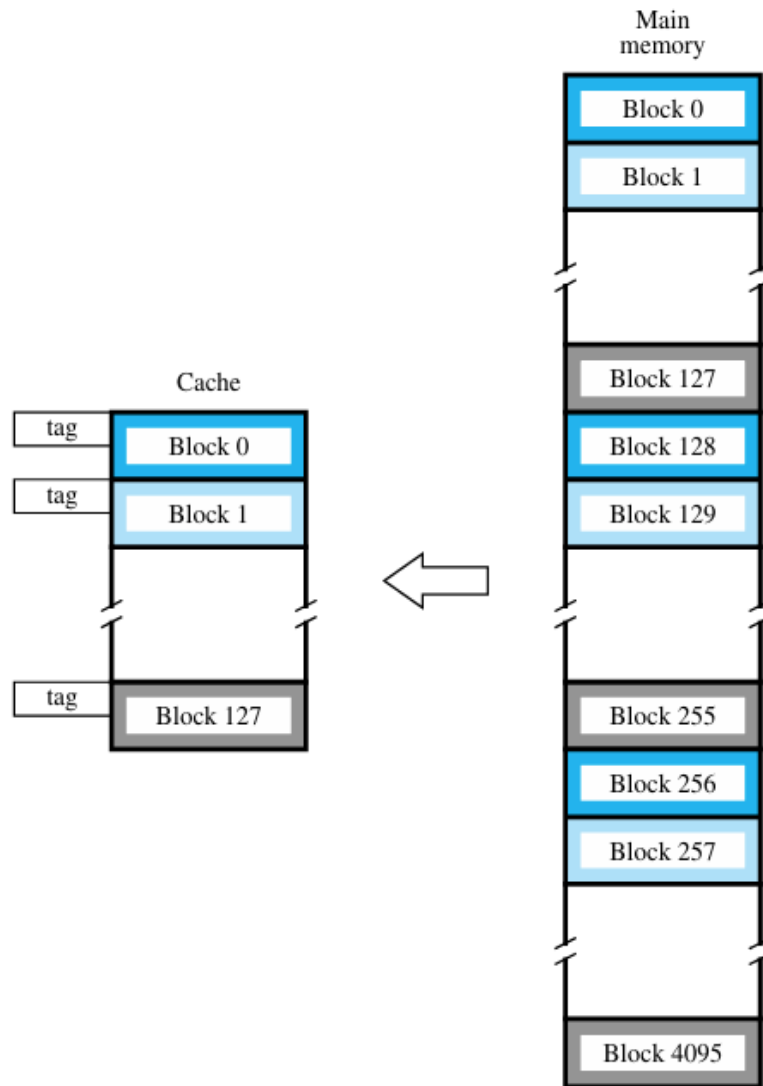
# Cache hit

- *Existence of a cache is transparent to the processor. The processor issues Read and Write requests in the same manner.*
- *If the data is in the cache it is called a Read hit or Write **HIT**.*
- *If the data is not present in the cache, then a Read miss or Write **MISS** occurs.*

# Mapping functions

- Mapping functions determine how & where memory blocks are placed in the cache.
- A simple processor example:
- **Cache:** 128 blocks, each block holds 16 words  
→  $128 \times 16 = \mathbf{2048 \text{ words}}$  total (that's 2 K words of cache).
- **Main memory:** Addressable with a **16-bit address**  
→  $2^{16} = \mathbf{65\ 536 \text{ words}}$ , which is 64 K words.
- **Block size:** 16 words  
→  $64 \text{ K} \div 16 = \mathbf{4096 \text{ blocks}}$  in main memory.
- Three mapping functions:
- Direct mapping
- Associative mapping
- Set-associative mapping.

# Direct mapping



Tag	Block	Word
5	7	4

Main memory address

- **Direct Mapping Rule:**

Each block in main memory maps to exactly one block in the cache. The formula is:

- $\text{Cache Block} = (\text{Memory Block}) \bmod (\text{Number of Cache Blocks})$



# Direct mapping

- **Conflict or contention:**

Because multiple memory blocks can map to the same cache block, there can be conflicts even if the cache isn't full. For example, if a program's instructions span memory blocks 1 and 129, both need cache block 1. Direct mapping solves this by simple replacement: the new block overwrites the old one.

- **How access works:**

When the processor requests an address:

- The cache block field tells which cache block to check.
- The tag bits are compared with the stored tag for that block.
- If the tag matches → cache hit, the word is in cache.
- If the tag doesn't match → cache miss, the block is loaded from main memory, replacing the old block.

# Direct mapping

- If  $2^k$  is the no. of words in a block, then  $k$  bits to represent a word.
- If  $2^m$  is the no. of blocks in cache memory, then  $m$  bits to represent a block.
- No. of tags = (no. of main memory blocks) / (no. of cache blocks) =  $2^t$ , then  $t$  bits to represent a tag.
- OR

- No. of tag bits = total address bits - (no. of
- |        |          |      |
|--------|----------|------|
| Tag no | Block no | word |
|--------|----------|------|
- $t \qquad m \qquad k$
- + no. of bits for block)

# Direct mapping

- Consider a cache consisting of 256 blocks of 16 words each, for a total of 4096 words and assume main memory is addressable by 16 bit address and it consists of 4K blocks. How many bits are there in each of Tag, block/set and word fields for different mapping techniques?

B0 – 16 words	0
B1 – 16 words	1
:	:
:	:
:	:
B255 – 16 words	255

- Given: main memory – 16 bits address  $\Rightarrow$  memory of  $2^{16}$  words.

- Direct mapping –

- No. of words =  $16 = 2^4 \Rightarrow 4$  bits

- No. of blocks =  $256 = 2^8 \Rightarrow 8$  bits

- No. of tag bits = total address bits – (no. of bits of block no. + no. of bits of word)

- No. of tag bits =  $16 - 4 - 4 = 8$

-

# Direct mapping

- Consider a cache consisting of 64 blocks and main memory consisting of 4096 blocks of 128 words each. i) How many bits for memory address? ii) How many bits for representing tag, block and word fields?

Given: main memory - 4096 blocks of 128 words =>  
 $2^{12}$  blocks X  $2^7$  words. =>  $2^{19}$  words

- Hence 19 bits to represent memory address

- Direct mapping -

- No. of words = 128 =  $2^7$  =>

- No. of blocks = 64 =  $2^6$  =>

- No. of tag bits = 19 -
- |        |          |      |
|--------|----------|------|
| Tag no | Block no | word |
| t      | m        | k    |

- 19 - (7 + 6) = 6
- |        |          |      |
|--------|----------|------|
| Tag no | Block no | word |
| 6      | 6        | 7    |

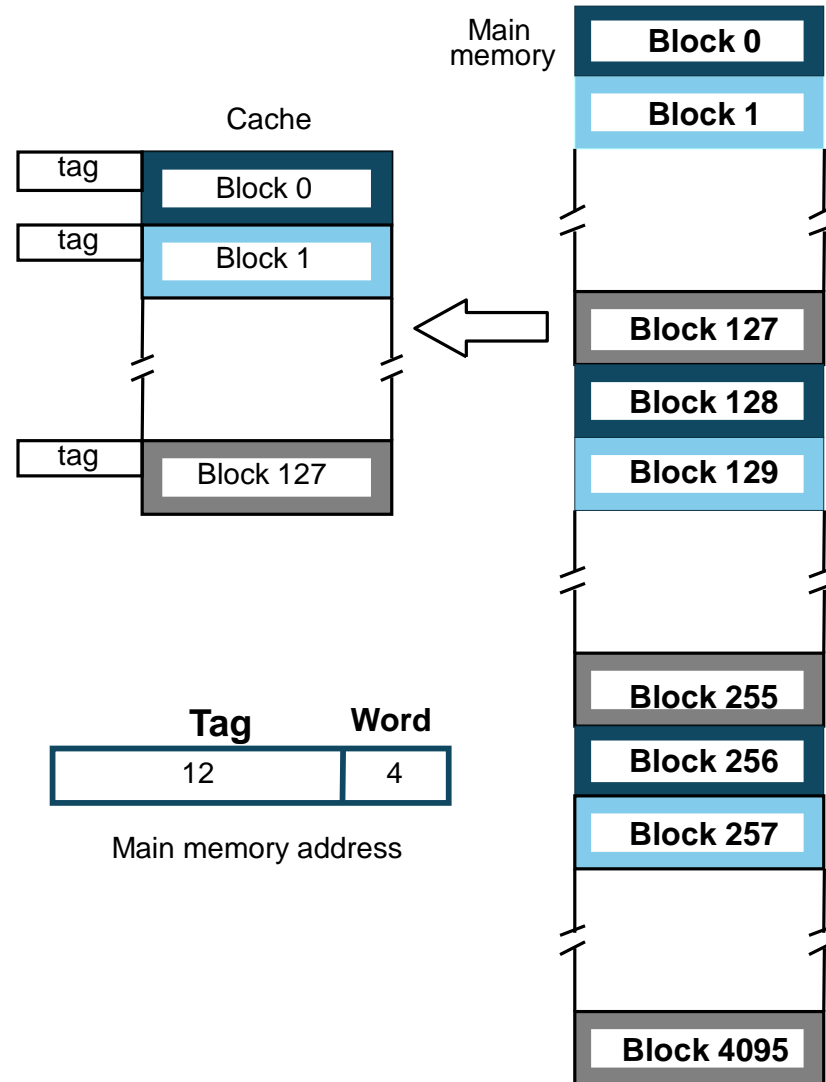
B0 – 128 words	0
B1 – 128 words	1
:	:
:	:
:	:
B63 – 128 words	63

1) Consider a cache consisting of 128 blocks of 4 words each and main memory consisting of 4096 blocks. i) How many bits for memory address? ii) How many bits for representing tag, block and word fields?

2) A computer system has a main memory of 16K blocks of 4 words each and has a 512 words cache. How many bits are required for tag, block / set and word fields in each mapping techniques ( In set- associative, 4 blocks per set).

3) Consider a cache memory of 16 words. Each block consists of 4 words. Size of the main memory is 256 bytes. How many bits for representing tag, block and word fields?

# Associative mapping



# Associative mapping

- **Associative Mapping**
- In **associative mapping**, any block of main memory can be placed into any cache line.
- This makes it more **flexible** and ensures **better utilization of cache space** compared to direct mapping.
- **Address Structure**
- The **memory address** is divided into two fields:
  - **Word field (low-order 4 bits)**: Identifies the word within a block.
  - **Tag field (high-order 12 bits)**: Uniquely identifies a memory block when it is stored in the cache.
- Formula:
  - $\text{Tag bits} = \text{Total address bits} - \text{Word bits}$
- **Operation**
- When the CPU generates a memory address:
  - The **tag field** of the address is compared with the **tags of all cache entries** simultaneously.
  - If a match is found, it's a **cache hit**.
  - If no match is found, it's a **cache miss**, and the block must be fetched from main memory.
- 
-

# Associative mapping

- **Advantages**

- Fully flexible – any block can go into any cache line.
- Maximizes cache space utilization.
- Allows use of **replacement algorithms** (e.g., FIFO, LRU, Random) to decide which block to evict when the cache is full.

- **Disadvantages**

- Requires **searching all cache entries** in parallel to find a match → increases **hardware complexity**.
- More **expensive** than direct-mapped caches because it needs comparators for all lines (e.g., 128 comparisons for 128 cache lines).



# Associative mapping

- Consider a cache consisting of 256 blocks of 16 words each, for a total of 4096 words and assume main memory is addressable by 16 bit address and it consists of 4K blocks. How many bits are there in each of Tag, block/set and word fields for different mapping techniques?

Given: main memory - 16 bits address  $\Rightarrow$  memory  $2^{16}$  words.

<b>B0 – 16 words</b>	<b>0</b>
<b>B1 – 16 words</b>	<b>1</b>
:	:
:	:
:	:
<b>B255 – 16 words</b>	<b>255</b>

- Associative mapping -

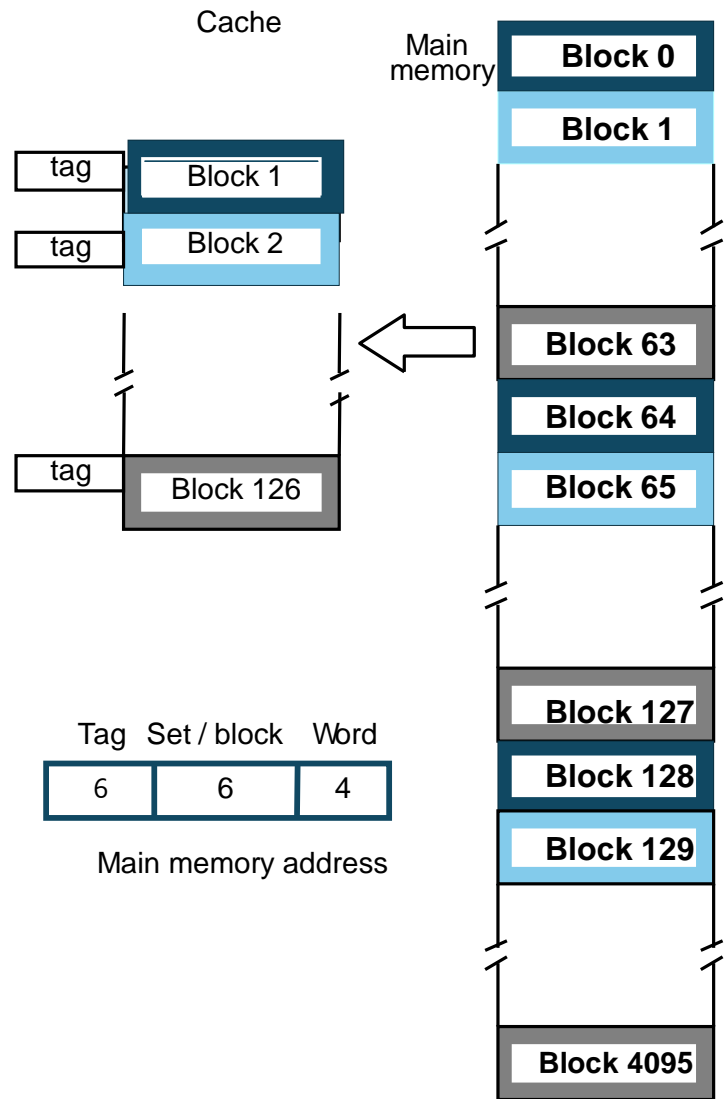
- No. of words = 16  $\Rightarrow$  24

Tag no	word	Tag no	word
--------	------	--------	------

- t k y bits = 12 4 address bits - word bits  $\Rightarrow$  16-4  $\Rightarrow$  12

# Set- Associative mapping

- **Definition:**
- Cache blocks are grouped into **sets**.
- A main memory block can be placed **only within a specific set**, but it may go into **any block inside that set**.
- This scheme is a **hybrid of direct mapping and associative mapping**.
- **Cache Organization Example**
- Suppose: Cache is divided into **64 sets**.
- Each set contains **2 blocks** → called **2-way set-associative mapping**.
- Example mapping:
  - Memory blocks 0, 64, 128 ... map to **set 0**, and can occupy **either of the 2 blocks** in that set.



# Set- Associative Mapping

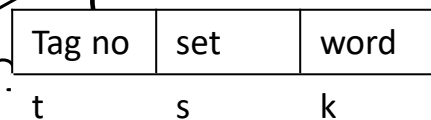
- **Address Structure**
- The memory address is divided into **three fields**:
- **Word field** - Identifies the word within a block.
- **Set field (6 bits)** - Determines the **set number**.
- $\text{No. of set bits} = \log(\text{No. of})$

# Set- Associative Mapping

- Number of sets in cache:  
No. of sets =  $\frac{\text{Total no. of blocks in cache}}{\text{Blocks per set}}$
- **Number of tags:**
- No. of tags  
$$= \frac{\text{No. of main memory blocks}}{\text{No. of sets}}$$
- If number of tags =  $2^t$ , then **t bits are required for the tag field.**

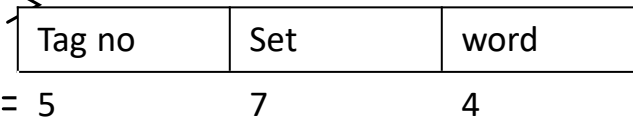
- Consider a cache consisting of 256 blocks of 16 words each, for a total of 4096 words and assume main memory is addressable by 16 bit address and it consists of 4K blocks. How many bits are there in each of Tag, block/set and word fields for different mapping techniques?  
Given: main memory - 16 bits address  $\Rightarrow$  memory is of  $2^{16}$  words

- Set - Associative mapping  $\Rightarrow$  (Assume that, there are 2 cache sets.)



<b>B0 – 16 words</b>	<b>0</b>
<b>B1 – 16 words</b>	<b>1</b>
⋮	⋮
⋮	⋮
⋮	⋮
<b>B255 – 16 words</b>	<b>255</b>

- No. of words = 16 =  $2^4 \Rightarrow$
- No. of sets =  $256/2 = 128 = 2^7$
- $4096 = 2^{12}$
- No. of tags =  $(2^{12}) / (2^7) \Rightarrow 2^5 \Rightarrow$



1) Direct mapping –

Tag no	Block no	word
t	m	k

2) Associative mapping –

Tag no	word
t	k

3) Set – Associative mapping

Tag no	set	word
t	s	k

- Suppose main memory consists of 32 blocks of 4 words each and cache consists of 4 blocks. How many bits required for main memory address? How many bits are there in each of Tag, block/set and word fields for different mapping techniques?  
Given: main memory – 32 blocks of 4 words each=>  $2^5$  blocks X  $2^2$  words.  $2^7$  words memory address is 7 bits

- Direct mapping –

Tag no	Block no	word
3	2	2

- No. of words = 4 =  $2^2$
- No. of blocks = 4 =  $2^2$  =>
- No. of tags = 7 – (2+2) = 3



- 2) Associative mapping -

- No. of words = 4 =  $2^2$   $\Rightarrow$

Tag no	word
5	2

- No. of tag bits = total bits - word bits  $\Rightarrow 7 - 2 = 5$

- 3) Set - Associative mapping  $\rightarrow$  (  
Assume that, there are 2 cache blocks in each  
set.)

- 

- No. of words = 4 =  $2^2$

Tag no	Set	word
4	1	2

- No. of sets =  $4/2 = 2^1$   $\Rightarrow$

- No. of tags =  $7 - (2 + 1) = 4$

- END of Module 2