

# COMPUTER ARCHITECTURE

---

Intensive Computation

**Annalisa Massini**

2022-2023

*Lecture 2*

- Overview of the conventional computer architecture and organization: **Von Neumann architecture**
- <http://WilliamStallings.com/COA/COA7e.html>

# INSTRUCTION CHARACTERISTICS

---

# Instruction Set

- The **instruction set** is the complete collection of instructions that the processor can execute
- Each **instruction must contain the information** required by the processor for execution
- In **machine code** each instruction has a unique bit pattern
- For human understanding a ***symbolic representation*** is used instead of the sequence of bits both for *opcodes* and *operands*
  - **Opcodes** - ADD, SUB, LOAD
  - **Operands** - ADD A,B

# Elements of an Instruction

Elements of a machine instruction are:

- **Operation code** (Op code)
  - Do this
- **Source Operand reference**
  - To this
- **Result Operand reference**
  - Put the answer here
- **Next Instruction Reference**
  - When you have done that, do this...

# Instruction Types

We can categorize instruction types as follows:

- **Data processing** - Arithmetic and logic instructions
- **Data storage** (main memory) - Movement of data into or out of register and or memory locations
- **Data movement** - I/O instructions
- **Control** - Test and branch instructions

# Number of Addresses

- An instruction could be required to contain up to **four** address references:
  - **two** source operands
  - **one** destination operand
  - the address of the **next** instruction
- In most architectures, most instructions have:
  - one, two, or three **operand addresses**
  - address of the **next instruction implicit** (obtained from the program counter)

# Number of Addresses

- **3 addresses**
- Operand 1, Operand 2, Result ->  $a = b + c$ ;
  - (next instruction implicit)
  - Needs very long words to hold everything
- Three-address instruction formats are **not common** because they require a relatively long instruction format to hold the three address references



# Number of Addresses

- **2 addresses**

- One address doubles as operand and result  $b = a + b$
- Reduces length of instruction
- Can require some extra work (to avoid altering the value of an operand, a MOVE instruction is used to save the value to a temporary location before the operation)

- **1 address** (Common on early machines)

- Implicit second address - Usually a register (accumulator)

- **0 addresses**

- All addresses implicit - applicable to a memory organization as a stack

# How Many Addresses

- **More addresses**
  - More complex (powerful?) instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- **Fewer addresses**
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions

# Instruction Set Design

- The **design** of an **instruction set** is **very complex** because it affects many aspects of the computer system
- The instruction set:
  - **defines** many of the **functions** performed by the processor
  - **has** a significant effect on the **implementation** of the processor
  - is the programmer's means of **controlling the processor**

# Instruction Set Design

The most important of fundamental design issues include:

- **Operation repertoire**
  - How many and which operations to provide
  - How complex operations should be
- **Data types**
  - Various types of data upon which operations are performed
- **Instruction formats**
  - Instruction length (in bits)
  - Number of addresses
  - Size of various fields

# Instruction Set Design

The most important of fundamental design issues include (cont):

- **Registers**
  - Number of CPU registers available
  - Which operations can be performed on which registers
- **Addressing modes**
  - The modes by which the address of an operand is specified
- **RISC v CISC**

# Types of Operand

- Machine instructions operate on data
- The most important general categories of data are:
  - **Addresses**
  - **Numbers**
    - Integer/floating point
  - **Characters**
    - ASCII etc.
  - **Logical Data**
    - Bits or flags

# ADDRESSING MODES AND FORMATS

---

# Addressing Modes

- The **address field** or fields in a typical instruction format are relatively **small**
- But we would like to be able to reference a **large range of locations** in main memory
- A variety of addressing techniques have been employed
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register Indirect
  - Displacement (Indexed)
  - Stack



# Immediate Addressing

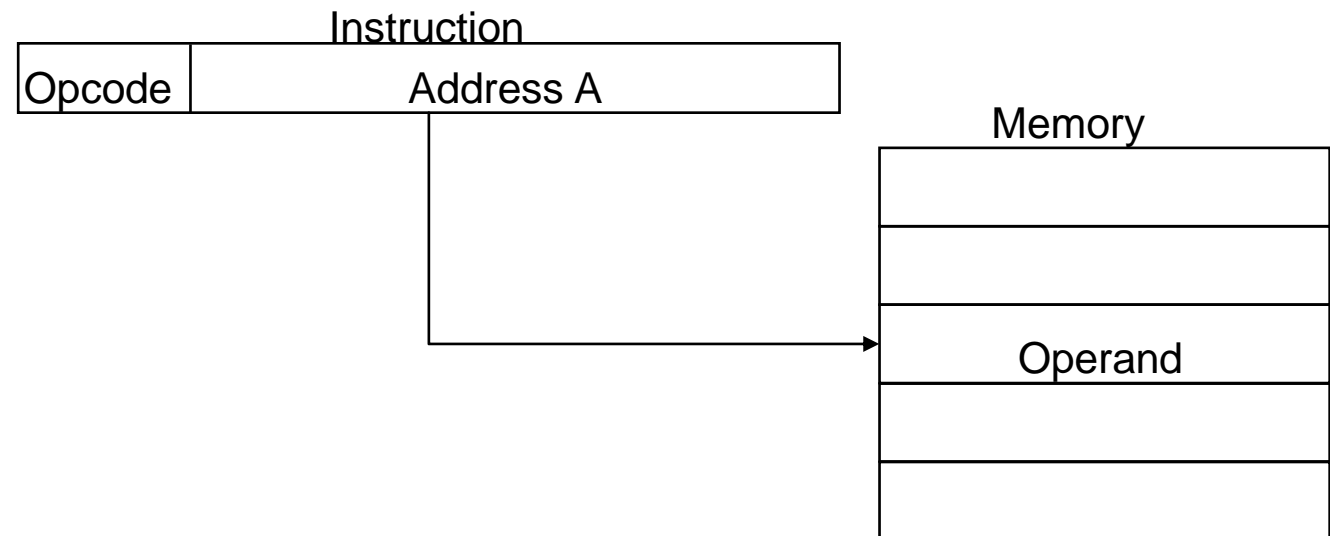
- **Operand is part of instruction**
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Instruction



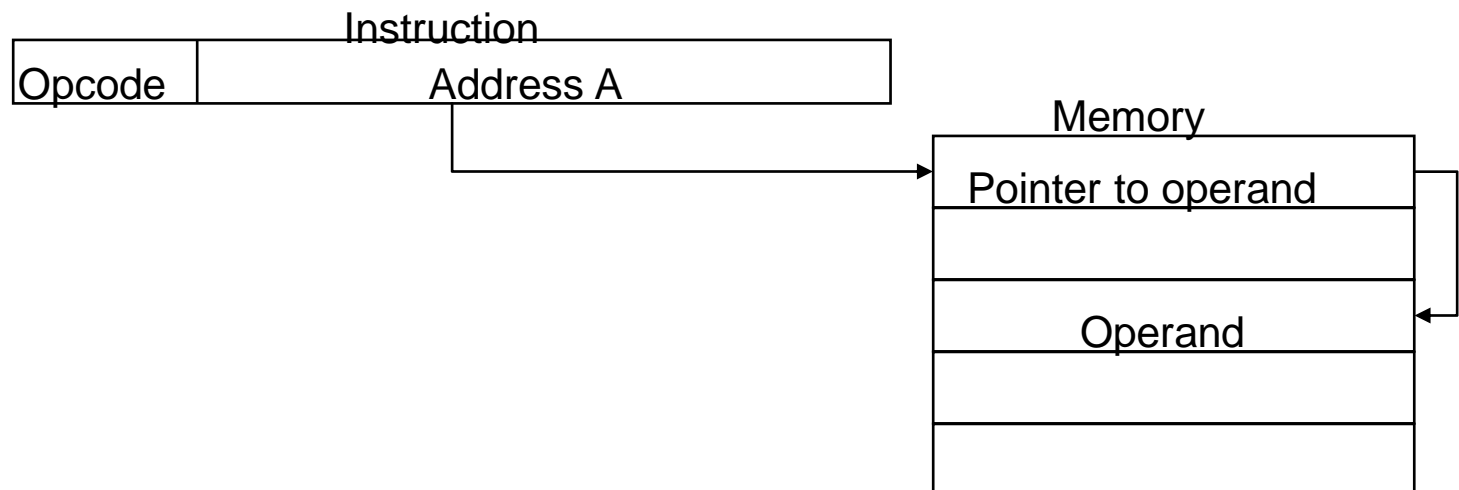
# Direct Addressing

- Address field contains **address of operand**
- Effective address (EA) = address field (A)
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space



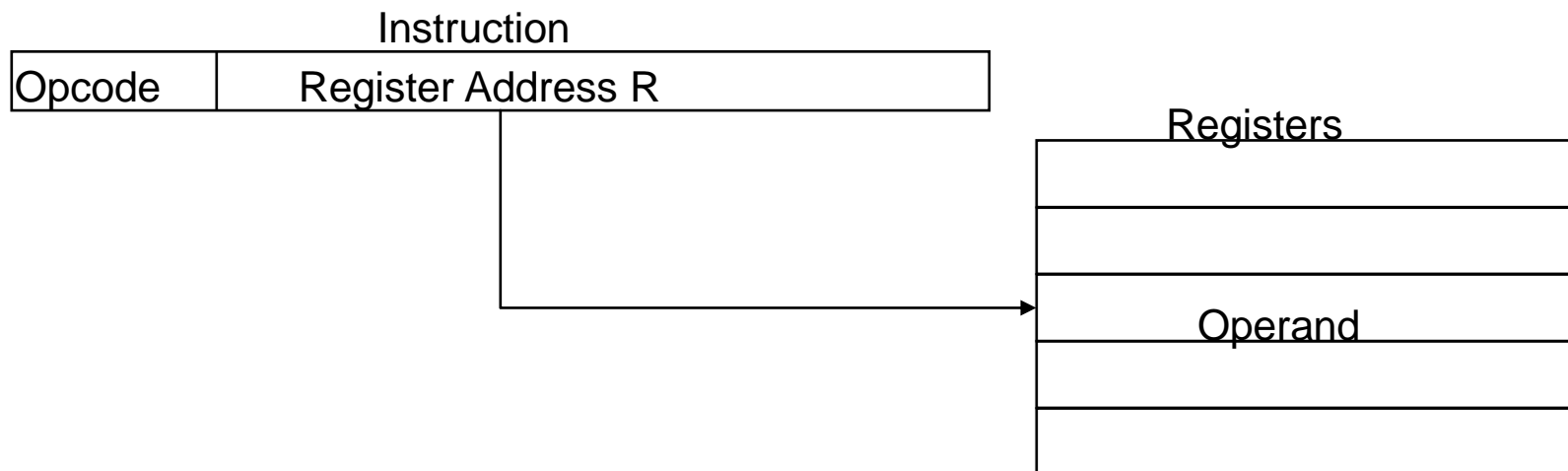
# Indirect Addressing

- **Memory cell pointed to by address field** contains the address of (pointer to) the operand
- $EA = (A)$ 
  - Look in A, find address (A) and look there for operand
- Large address space -  $2^n$  where  $n$  = word length
- Multiple memory accesses to find operand - hence slower



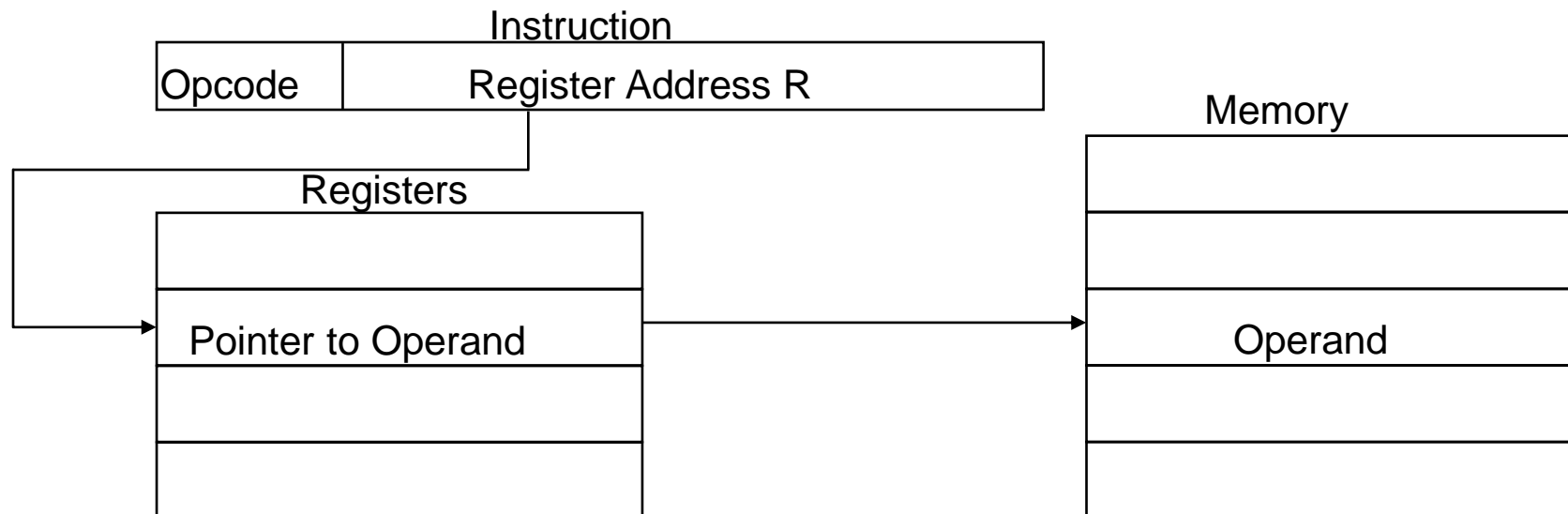
# Register Addressing

- **Operand is held in register** named in address filed
- Limited number of registers
- Very small address field needed
  - Shorter instructions - Faster instruction fetch
- No memory access - Very fast execution
- Very limited address space
- Multiple registers helps performance



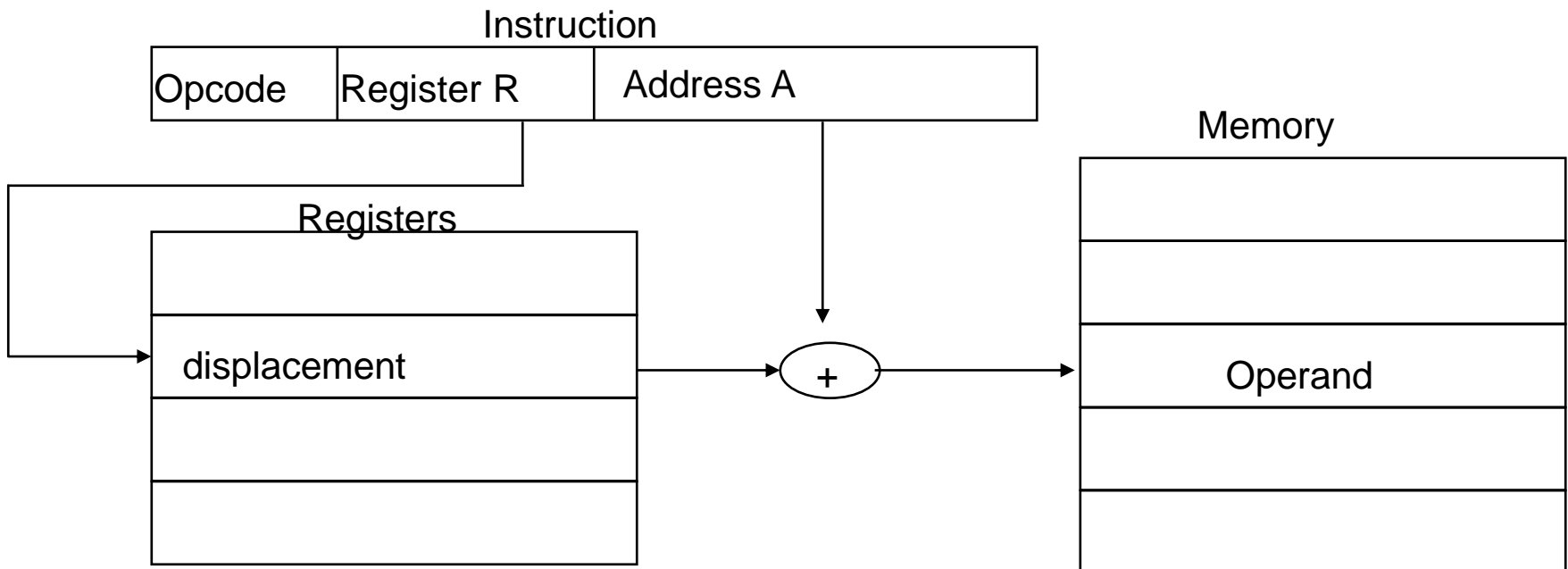
# Register Indirect Addressing

- It is **indirect addressing**
- $EA = (R)$
- Operand is in memory cell pointed to by contents of register R
- Large address space ( $2^n$ )
- One fewer memory access than indirect addressing



# Displacement Addressing

- $EA = A + (R)$
- Address field hold **two values**
  - $A =$  **base value**
  - $R =$  register that holds **displacement**
  - or vice versa

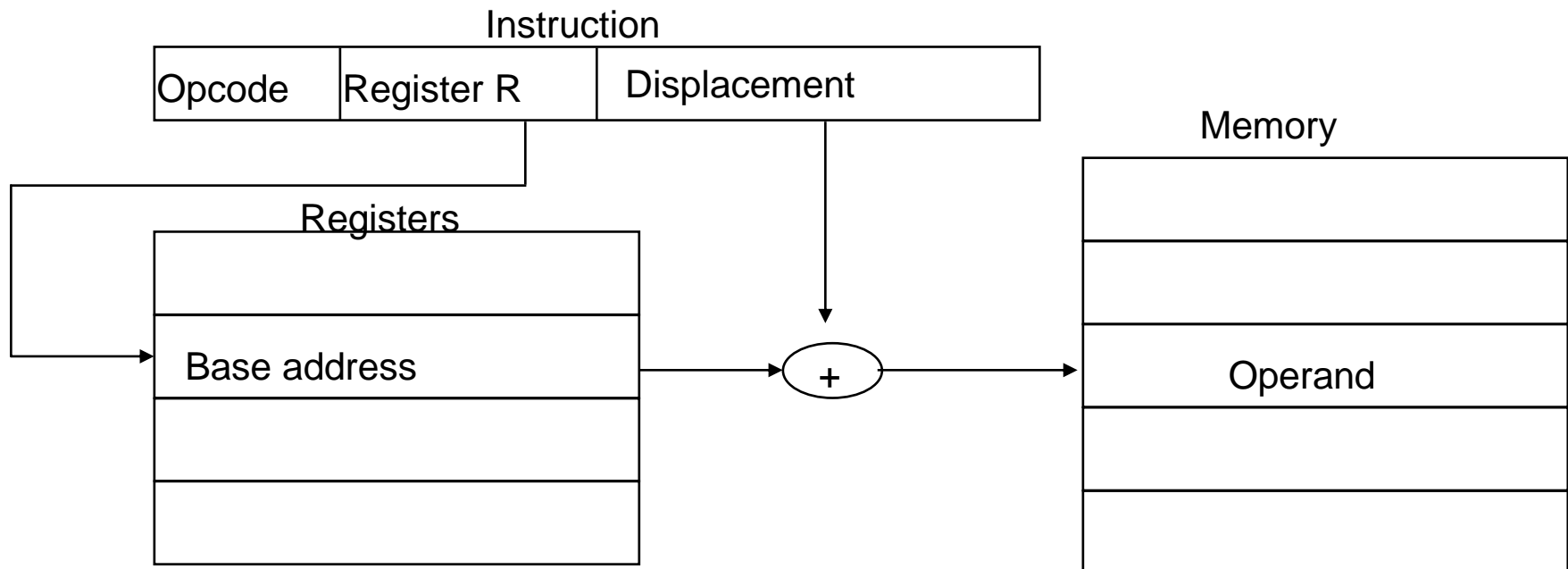


# Relative Addressing

- A version of **displacement addressing**
- $R = \text{Program counter}, PC$
- $EA = A + (PC)$
- i.e. operand from A cells from current location pointed to by PC
- locality of reference & cache usage

# Base-Register Addressing

- A version of **displacement addressing**
- A holds **displacement**
- R holds **base address (through a pointer)**
- R may be explicit or implicit
- e.g. segment registers in 80x86





# Indexed Addressing

- A version of **displacement addressing with indexing**
- $A$  = base address
- $R$  = **displacement**
- $EA = A + R$
- Good for accessing arrays (by increment)
  - $EA = A + R$
  - $R++$

# Stack Addressing

- A **stack** is a linear array of **reserved memory locations**
- Can be sometimes referred to as a ***pushdown list*** or ***last-in-first-out queue***
- Items are appended to the top of the stack
- At any given time, the location block is partially filled
- Associated with the stack is a pointer - **stack pointer** - whose value is the address of the **top of the stack**

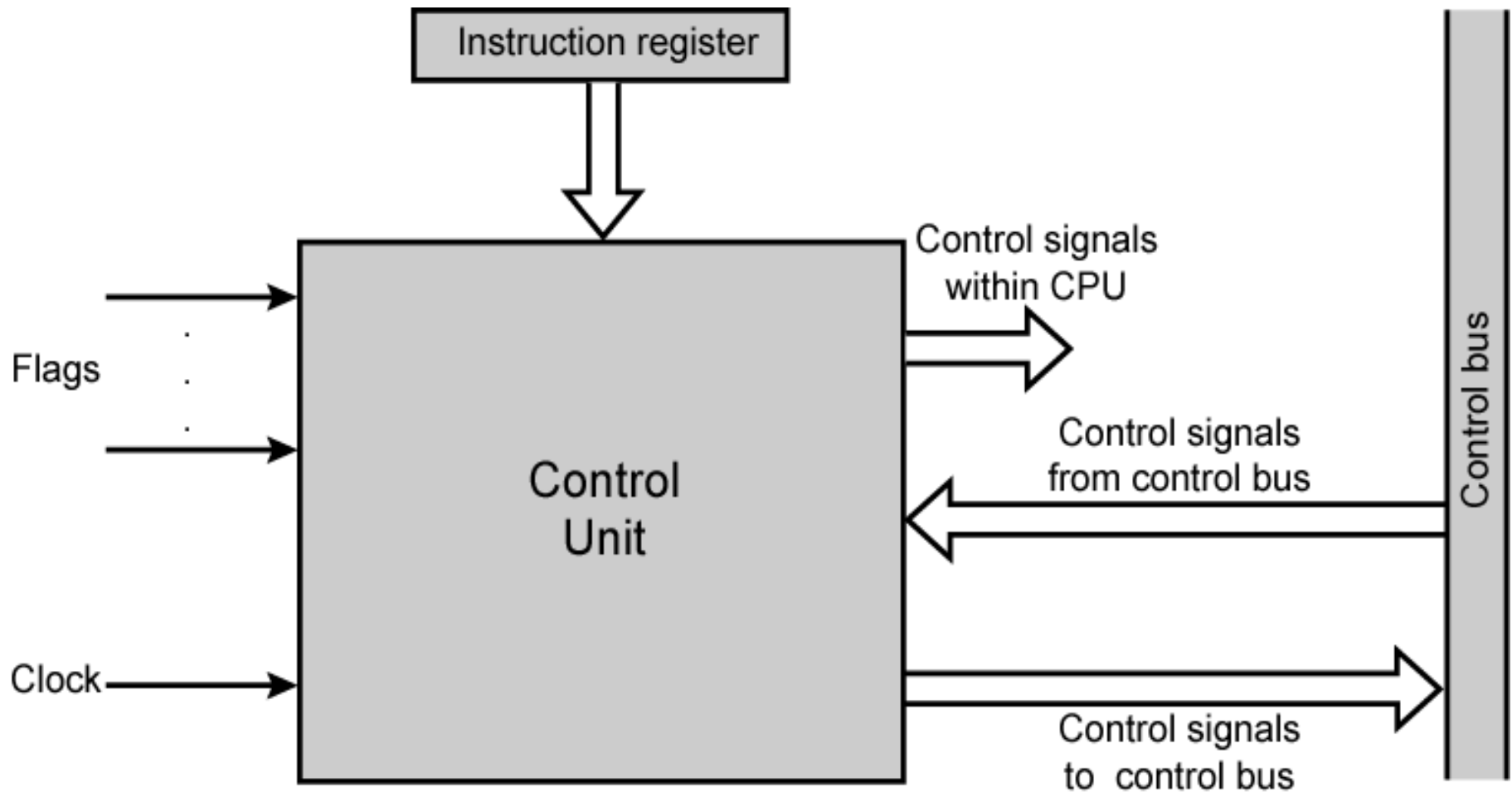
# CONTROL UNIT

---

# Control unit

- The **control unit** is that portion of the processor that actually causes things to happen
- The control unit issues:
  - Control signals **external** to the processor to cause **data exchange** with **memory and I/O modules**
  - Control signals **internal** to the processor to move data between registers, to cause the **ALU** to perform a specified function, and to regulate other **internal operations**
- Input to the control unit consists of:
  - the instruction register
  - flags
  - control signals from external sources (e.g., interrupt signals)

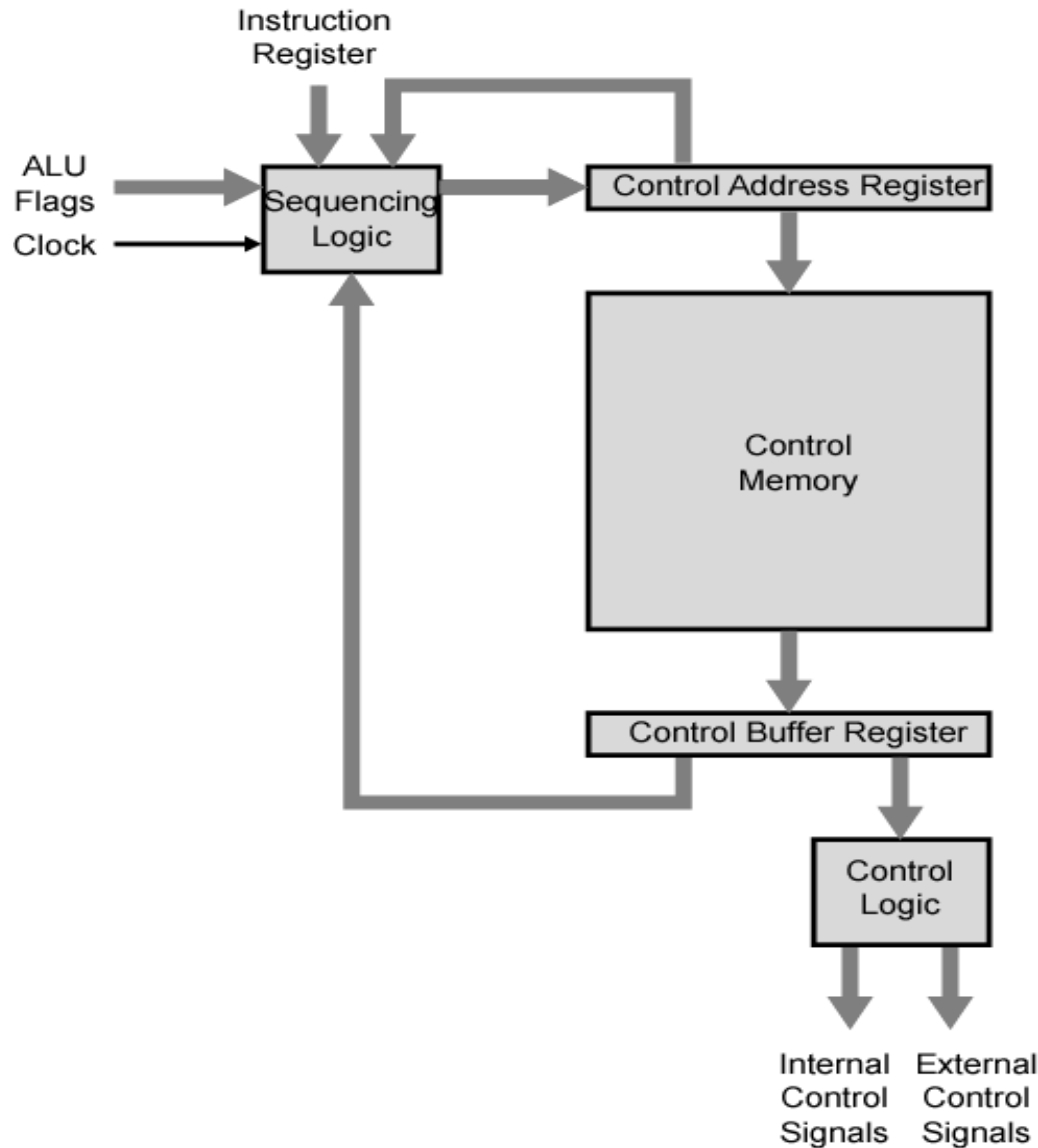
# Model of Control Unit



# Functions of Control Unit

- **Sequencing**
  - Causing the CPU to step through a series of micro-operations
- **Execution**
  - Causing the performance of each micro-op
- This is done using **Control Signals**

# Control Unit Organization



# Types of Micro-operation

- The execution of a program consists of *operations* involving different processor elements
- Operations consist of a sequence of **micro-operations**
- All **micro-operations** fall into one of the following **categories**:
  - Transfer data between registers
  - Transfer data from register to external
  - Transfer data from external to register
  - Perform an arithmetic or logical operation, using registers for input and output



# Control Signals

- **Clock**
  - One micro-instruction (or set of parallel micro-instructions) per clock cycle
- **Instruction register**
  - Op-code for current instruction
  - Determines which micro-instructions are performed
- **Flags**
  - State of CPU
  - Results of previous operations
- **From control bus**
  - Interrupts
  - Acknowledgements

# Control Signals - output

- **Within CPU**
  - Cause data movement
  - Activate specific functions
- **Via control bus**
  - To memory
  - To I/O modules

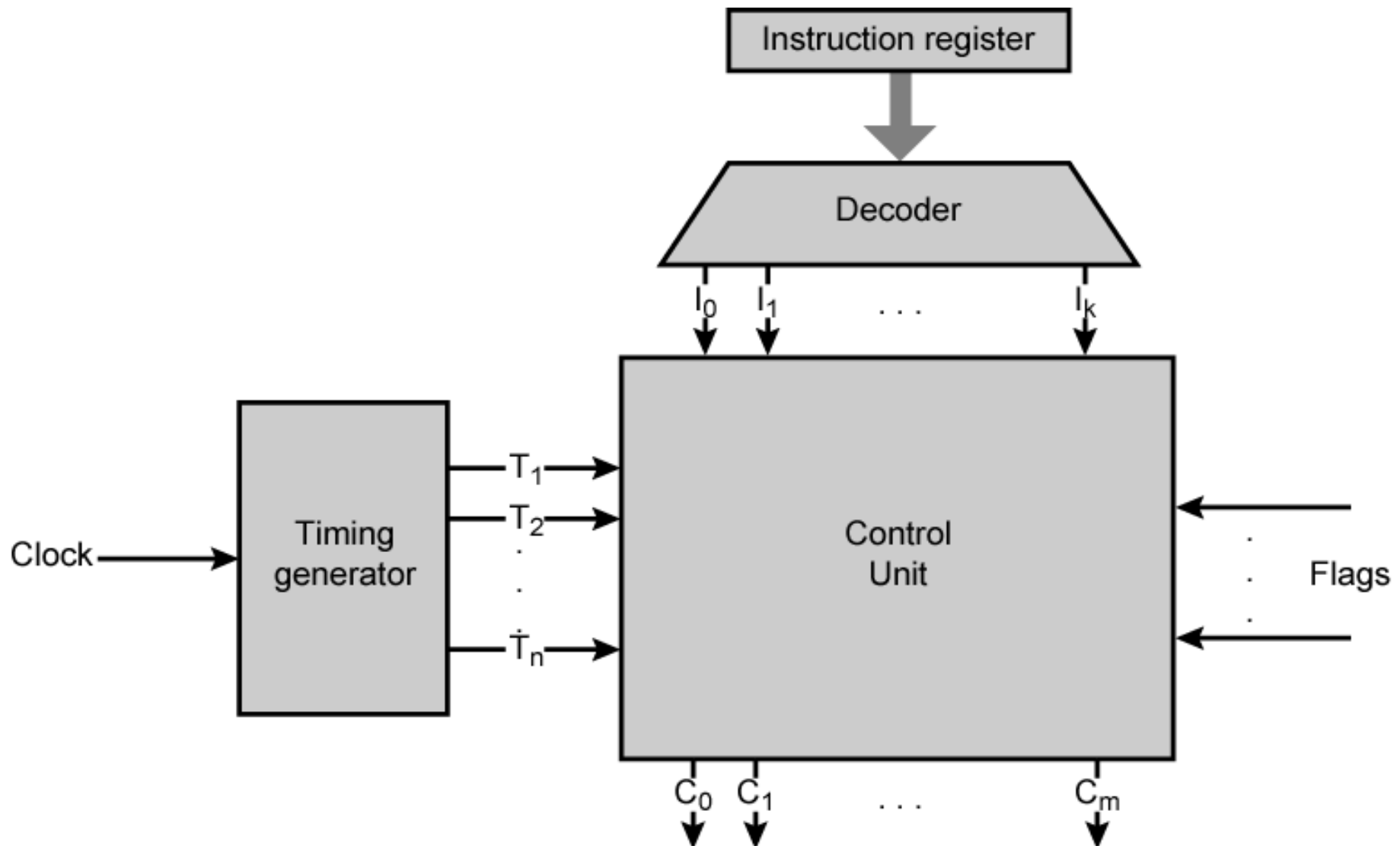
# Implementations

- A wide variety of techniques for the control unit implementation have been used
- Most of these fall into one of two categories:
  - **Hardwired** implementation
  - **Microprogrammed** implementation
- In a **hardwired** implementation, the control unit is essentially a **state machine circuit**:
  - Its input logic signals are transformed into a set of output logic signals, which are the control signals

# Hardwired Implementation

- **Control unit inputs**
- Flags and control bus (each bit means something)
- **Instruction register**
  - Different control signals for each different instruction
  - Unique logic for each op-code
  - Decoder takes encoded input and produces single output
  - $n$  binary inputs and  $2^n$  outputs
- **Clock**
  - Repetitive sequence of pulses
  - Must be long enough to allow signal propagation
  - Different control signals at different times within instruction cycle

# Control Unit with Decoded Inputs



# Problems With Hardwired Designs

- Complex sequencing & micro-operation logic
- **Difficult** to design and test
- **Inflexible** design
- **Difficult** to add new instructions

# Microprogrammed Control

- An alternative to a hardwired control unit is a **microprogrammed control unit**
- The logic of the control unit is specified by a **microprogram**:
  - sequence of instructions in a **microprogramming** language
- The (very simple) instructions specifies **micro-operations**
- A microprogrammed control unit is a relatively simple logic circuit that is capable of
  - **sequencing** through microinstructions
  - **generating control signals** to execute each microinstruction

# Microprogrammed Control

- Each **microinstruction** line describes a **set of micro-operations** occurring at one time
- For each **micro-operation**, all that the control unit is allowed to do is **generate a set of control signals**
- Thus, for any micro-operation, each control line emanating from the control unit is either on or off
- Each micro-operation is represented by a different pattern of 1s and 0s: the **control word**
- In a *control word* each bit represents one control line
- A **sequence of control words** represents the sequence of micro-operations performed by the control unit

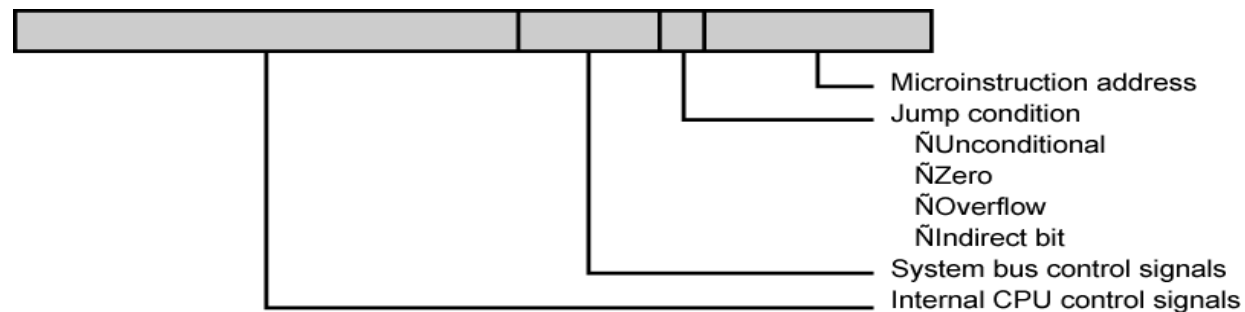


# Implementation

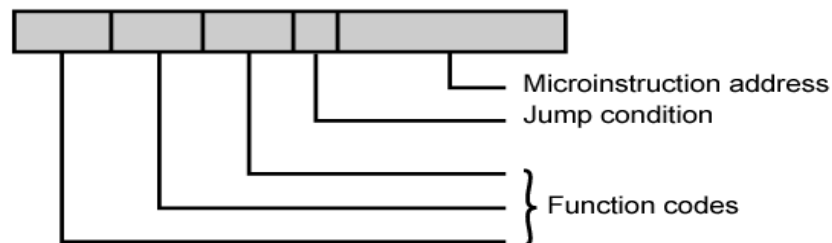
- **Control words** are put in a memory (ROM) and each word has a unique address
- An **address field** is added to each control word indicating the location of the next control word to be executed if a certain condition is true
- There is a sequence of control words **for each machine code instruction**
- Today's large microprocessor
  - Many instructions and associated register-level hardware
  - Many control points to be manipulated

# Micro-instruction Types

- **Vertical micro-programming:** each micro-instruction specifies (single or few) micro-operations to be performed
- **Horizontal micro-programming:** each micro-instruction specifies many different micro-operations to be performed in parallel



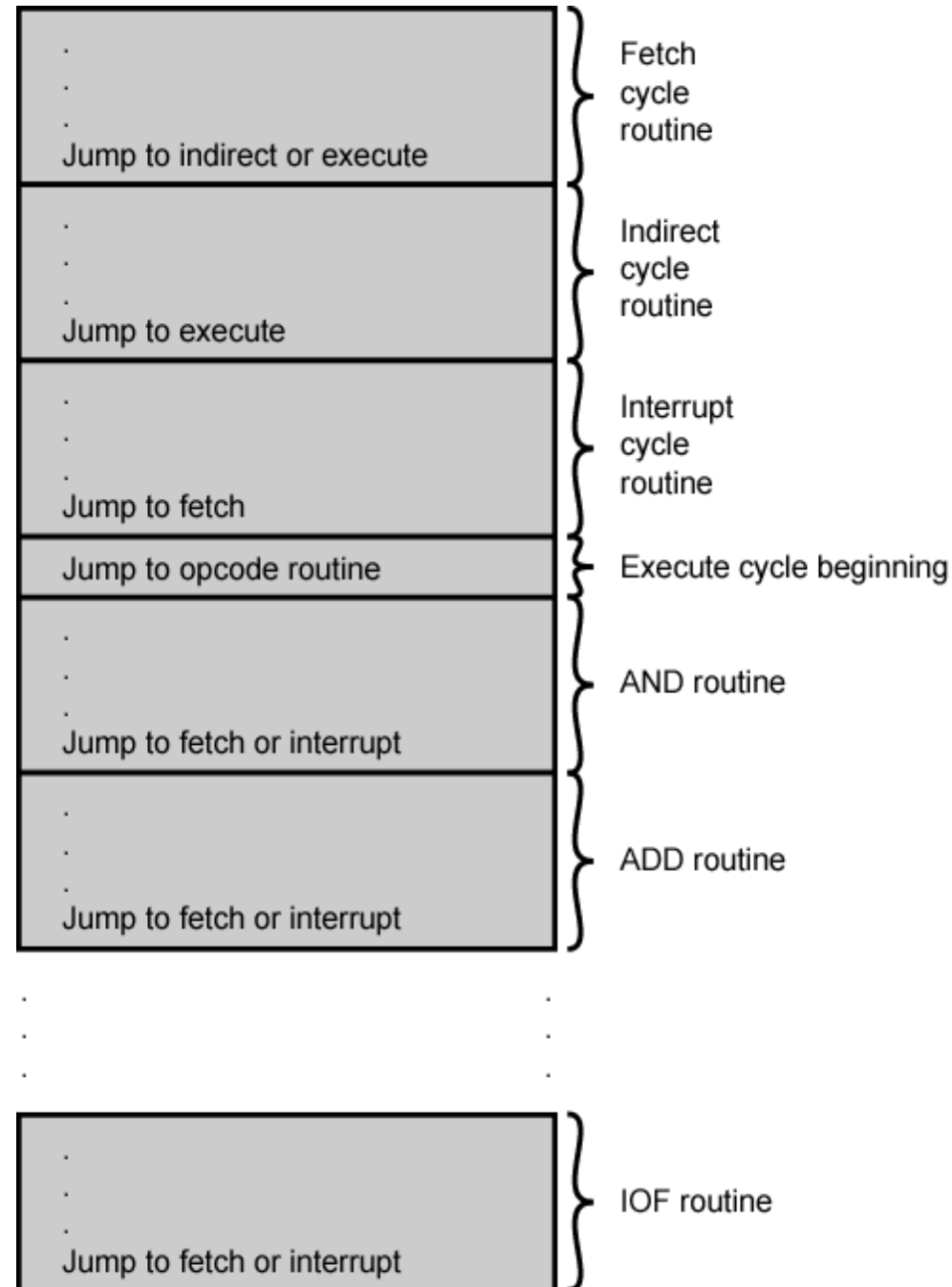
(a) Horizontal microinstruction



(b) Vertical microinstruction

# Organization of Control Memory

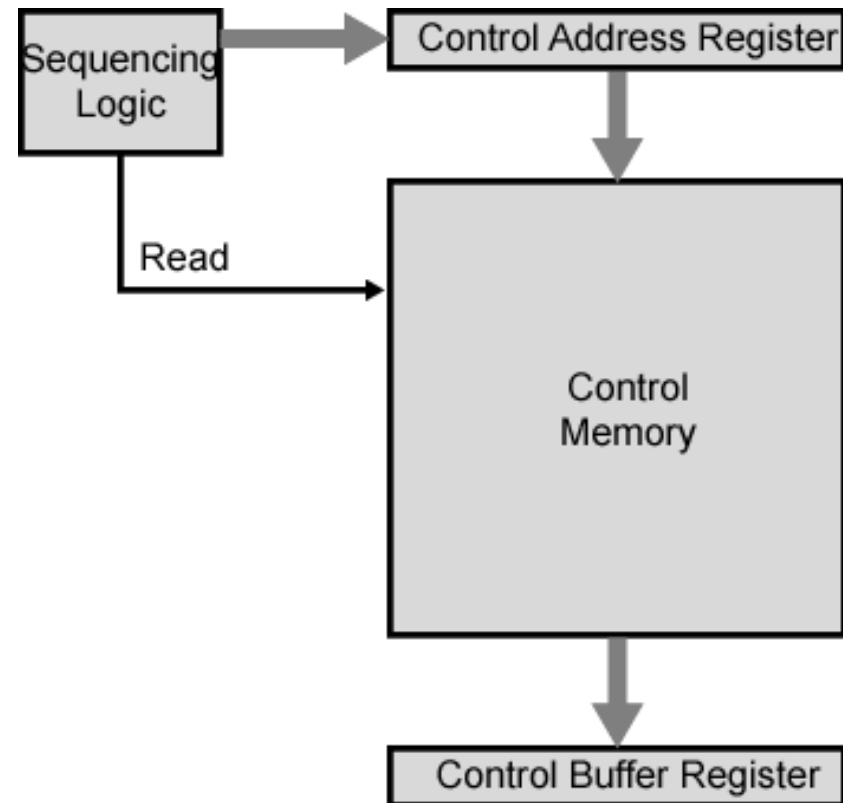
- ▶ Microinstructions (or control words) are arranged in a **control memory** (ROM)
- ▶ Microinstructions in a routine are executed sequentially
- ▶ Each **routine** ends with a **branch** or **jump** instruction indicating where to go next
- ▶ A special execute cycle routine specifies the instruction routine (AND, ADD, and so on) to be executed next (according to the current opcode)



# Control Unit

Key elements of microprogrammed implementation:

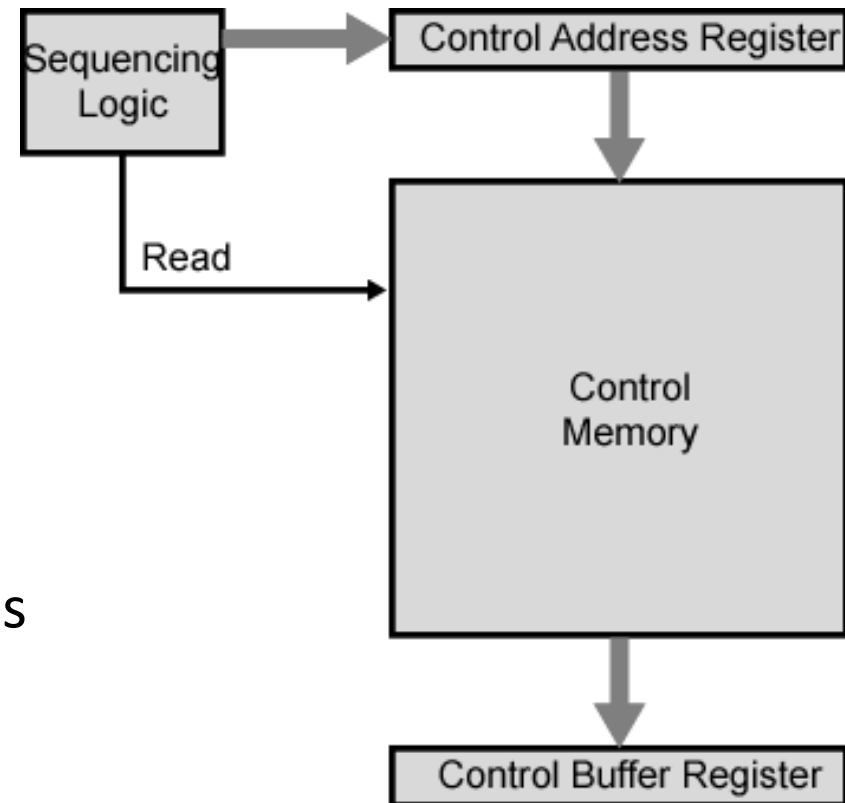
- ▶ The set of microinstructions is stored in the *control memory*
- ▶ The *control address register* contains the address of the next microinstruction to be read



# Control Unit

Key elements of microprogrammed implementation:

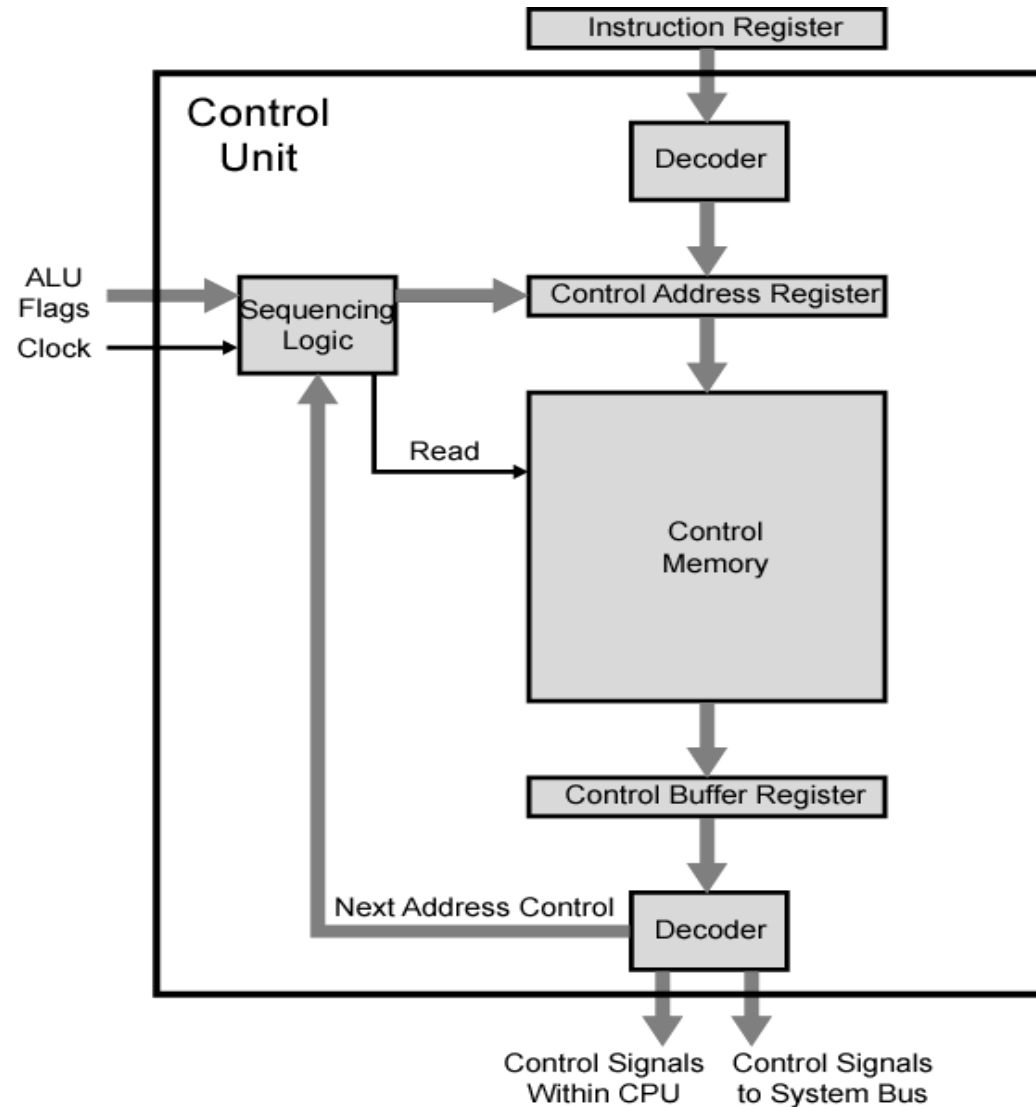
- ▶ When a microinstruction is read from the control memory, it is transferred to a *control buffer register*
- ▶ *Reading* a microinstruction from the control memory is the same as *executing* that microinstruction
- ▶ The *sequencing unit* loads the control address register and issues a read command



# Functioning of Microprogrammed Control Unit

The control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a **READ command** to the control memory
2. The word whose address is specified in the control address register is read into the **control buffer register**

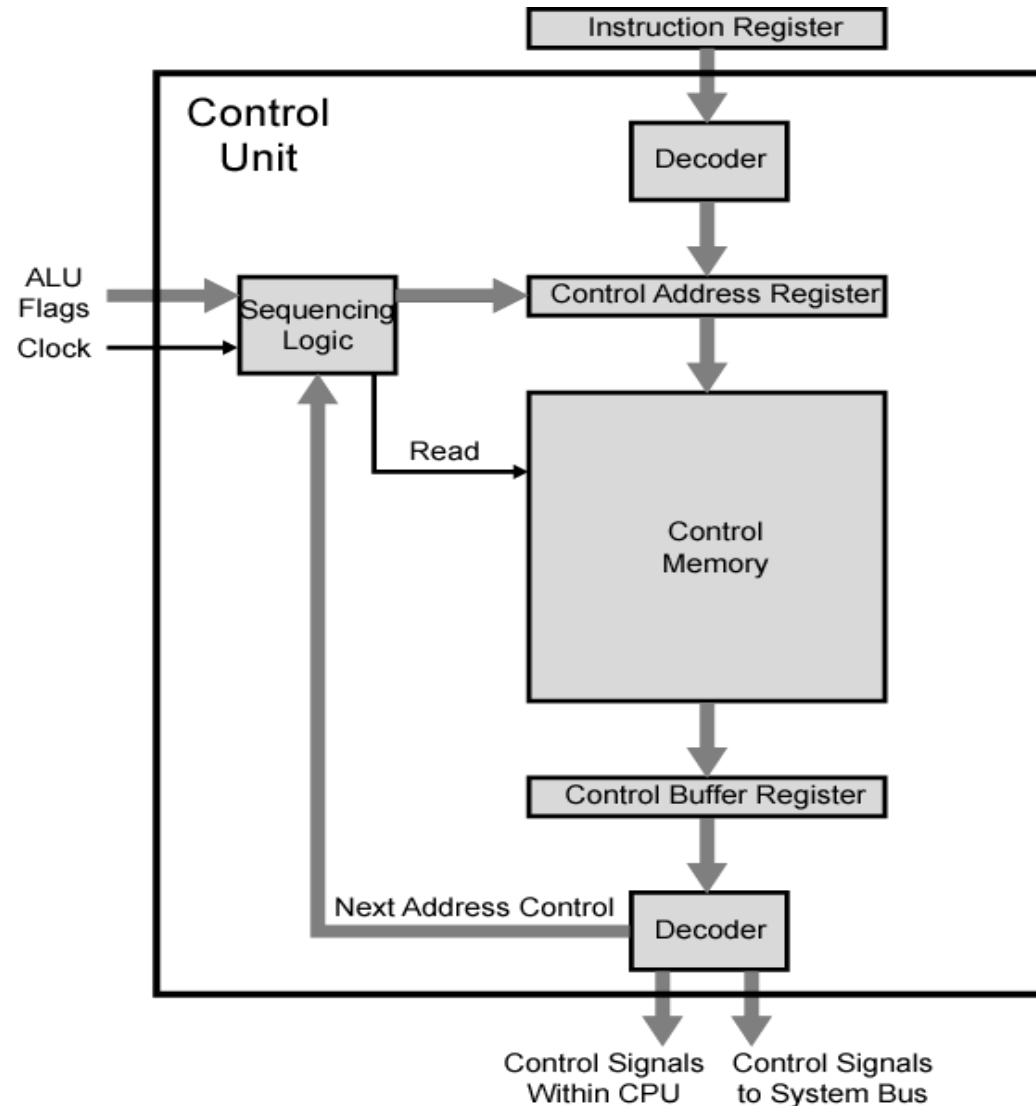


# Functioning of Microprogrammed Control Unit

3. The content of the control buffer register generates control signals and **next address information** for the sequencing logic unit

4. The sequencing logic unit **loads a new address** into the control address register (based on the next-address information from the control buffer register and the ALU flags)

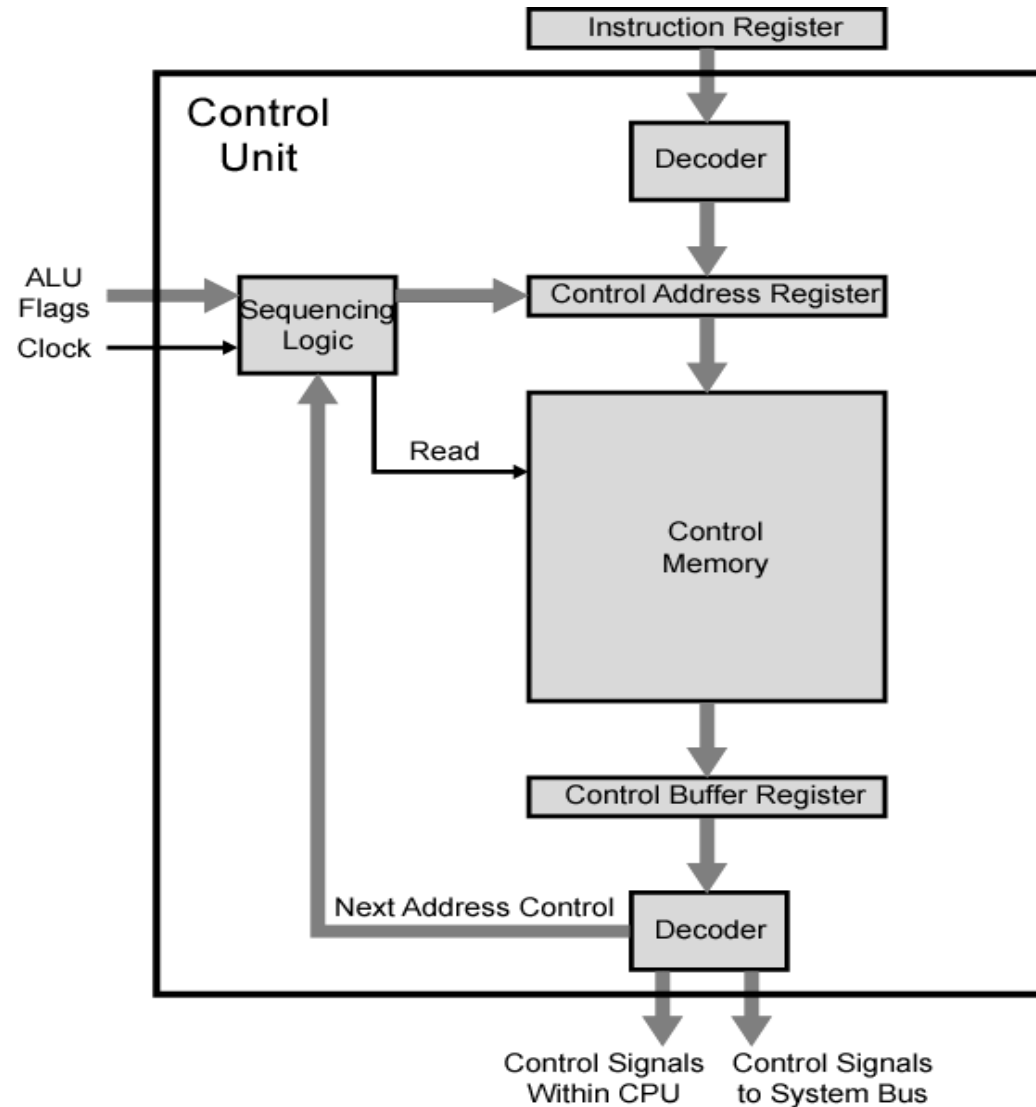
*All this happens during one clock pulse*



# Functioning of Microprogrammed Control Unit

Figure shows two *decoder*:

- ▶ The *upper* decoder translates the opcode of the IR into a control memory address
- ▶ The *lower* decoder is used for *vertical microinstructions*





# MEMORY HIERARCHY

---

# Memory Hierarchy

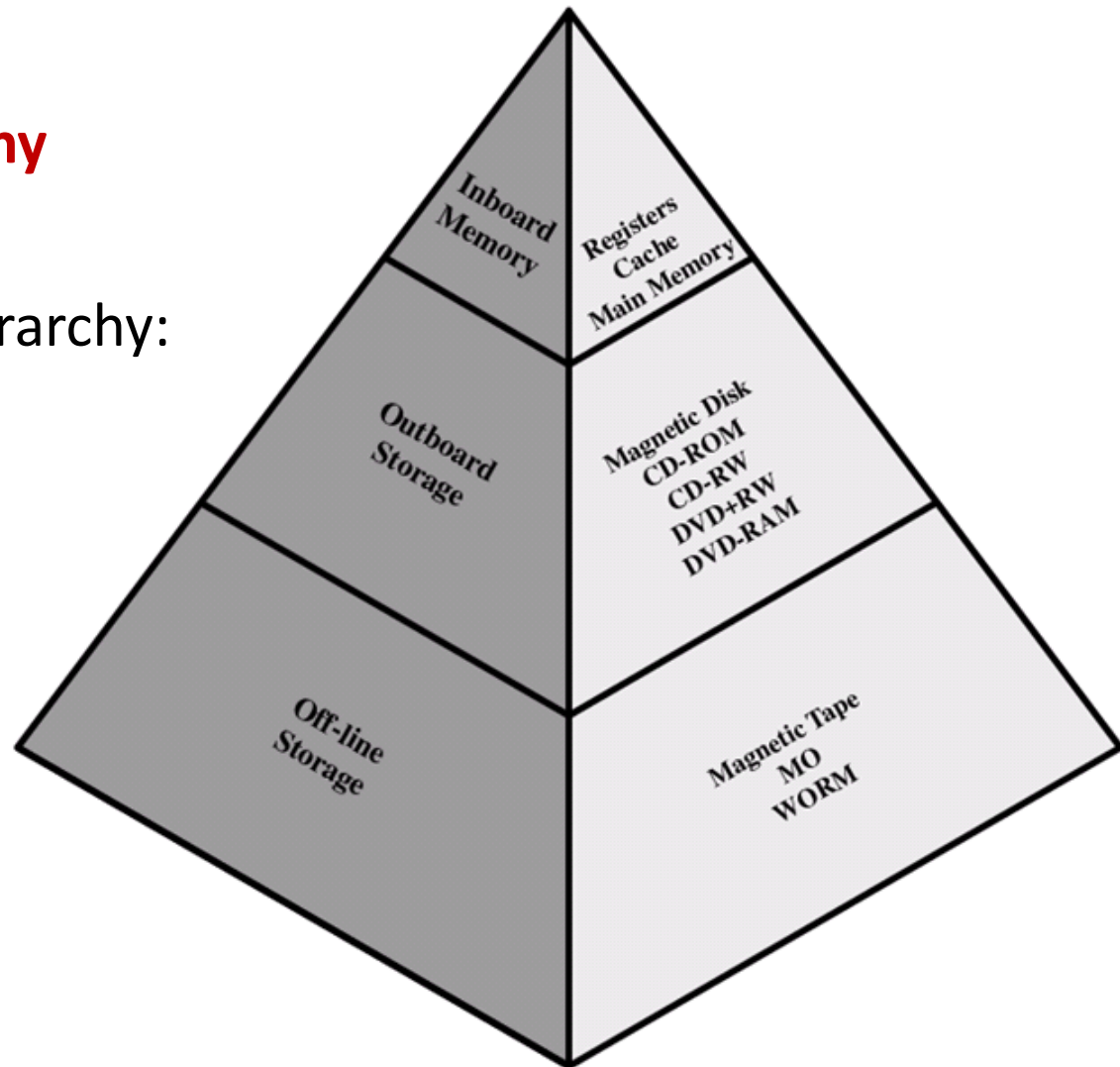
- The three key characteristics of memory:
  - **Capacity**
  - **Access time**
  - **Cost**
- A variety of technologies are used to implement memory systems
- The following relationships hold:
  - Faster access time, greater cost per bit
  - Greater capacity, smaller cost per bit
  - Greater capacity, slower access time

# Memory Hierarchy

The solution is to employ  
a **memory hierarchy**

As one goes down the hierarchy:

- Decreasing **cost** per bit
- Increasing **capacity**
- Increasing **access time**
- Decreasing **frequency of access** of the memory by the processor

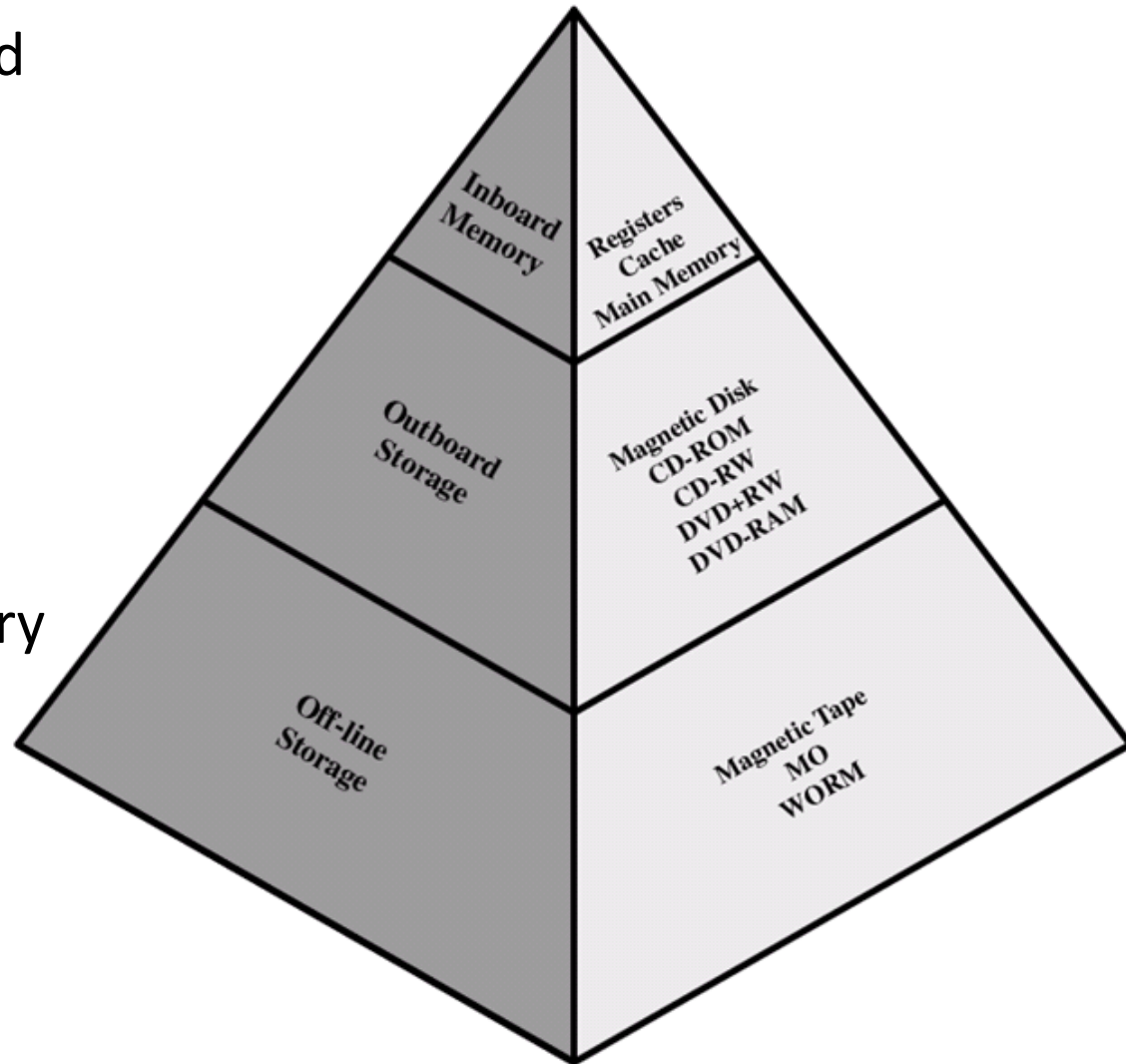


# Memory Hierarchy

- During the execution of a program, memory references for instructions and data tend to cluster: **locality of reference**
- Programs typically contain a number of iterative **loops** and **subroutines** (repeated references to a small set of **instructions**)
- Similarly, operations on **tables** and **arrays** involve access to a clustered set of **data** words
- It is possible to **organize data across the hierarchy** such that the percentage of accesses to each successively lower level is substantially less than that of the level above

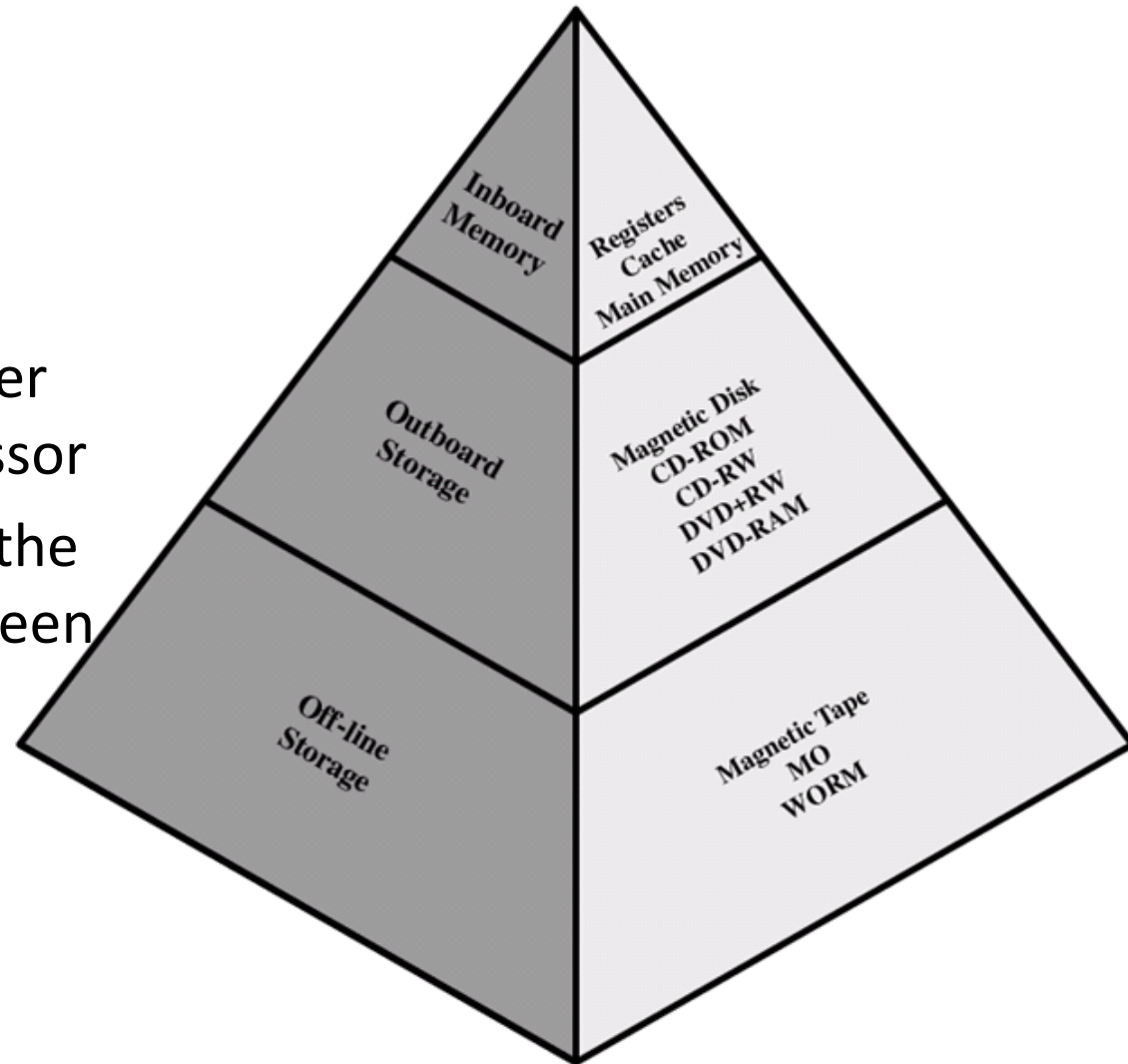
# Memory Hierarchy

- The fastest, smallest, and most expensive type of memory consists of the **registers** internal to the processor
- **Main memory** is the principal internal memory system of the computer



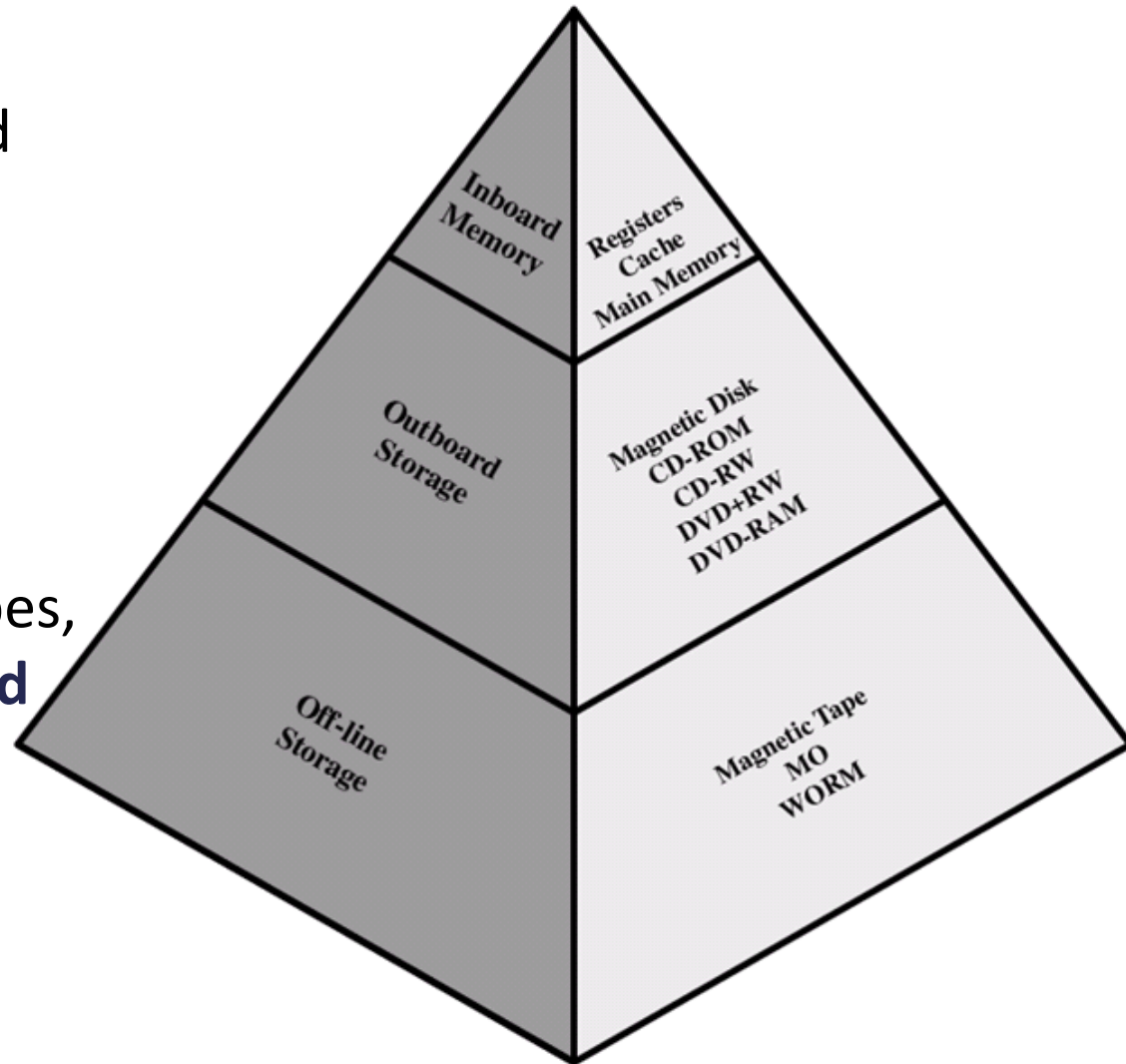
# Memory Hierarchy

- Main memory is usually extended with a higher-speed, smaller **cache**
- The cache is not usually visible to the programmer or, indeed, to the processor
- It is a device for staging the movement of data between main memory and processor registers to improve performance



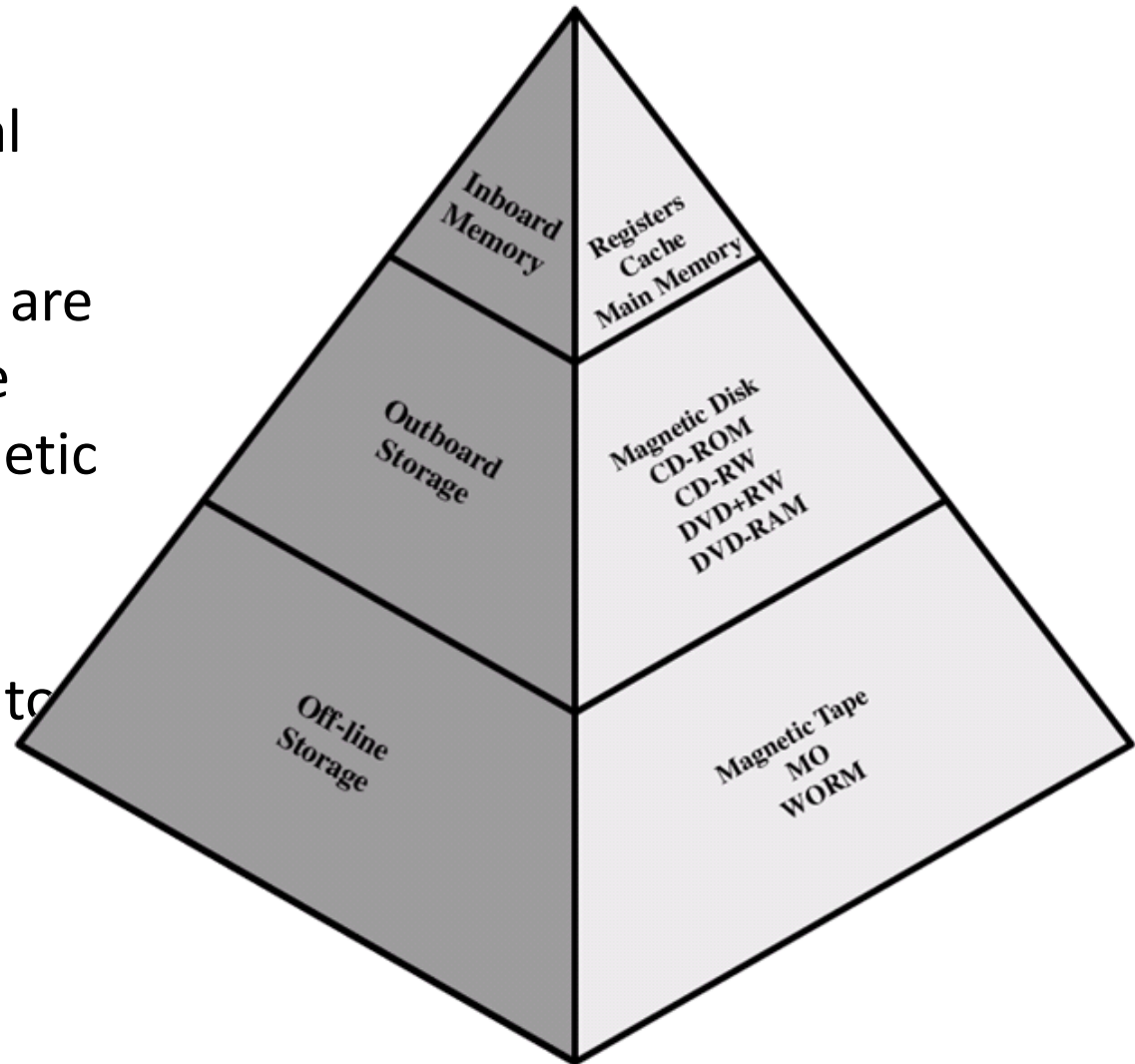
# Memory Hierarchy

- These three forms of memory are **volatile** and employ **semiconductor technology**
- The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in **speed and cost**



# Memory Hierarchy

- Data are stored more **permanently** on external mass storage devices
- The most common ones are hard disk and removable media (removable magnetic and optical storage)
- External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**





# Characteristics of memory systems

- **Location**
  - Internal (e.g. processor registers, main memory, cache)
  - External (e.g. optical disks, magnetic disks, tapes)
- **Capacity**
  - Word size
  - Number of words, Number of bytes
- **Unit of transfer**
  - Word
  - Block

# Characteristics

- **Access method**
  - Sequential (e.g. tape)
    - Start at the beginning and read through in order
    - Access time depends on location of data and previous location
  - Direct (e.g. disk)
    - Individual blocks have unique address
    - Access time depends on location and previous location
  - Random (e.g. RAM)
    - Individual addresses identify locations exactly
    - Access time is independent of location or previous access
  - Associative (e.g. Cache)
    - Data is located by a comparison with contents of a portion of the store
    - Access time is independent of location or previous access

# Characteristics

- **Performance**
  - Access time
    - Time between presenting the address and getting the valid data
  - Memory Cycle time
    - Time may be required for the memory to “recover” before next access
    - Cycle time is access + recovery
  - Transfer Rate
    - Rate at which data can be moved

# Characteristics

- **Physical type**
  - Semiconductor (RAM)
  - Magnetic (Disk & Tape)
  - Optical (CD & DVD)
- **Physical characteristics**
  - Volatile/nonvolatile
  - Erasable/nonerasable
  - Power consumption
- **Organization**
  - Physical arrangement of bits into words
  - Memory modules

# Hierarchy List

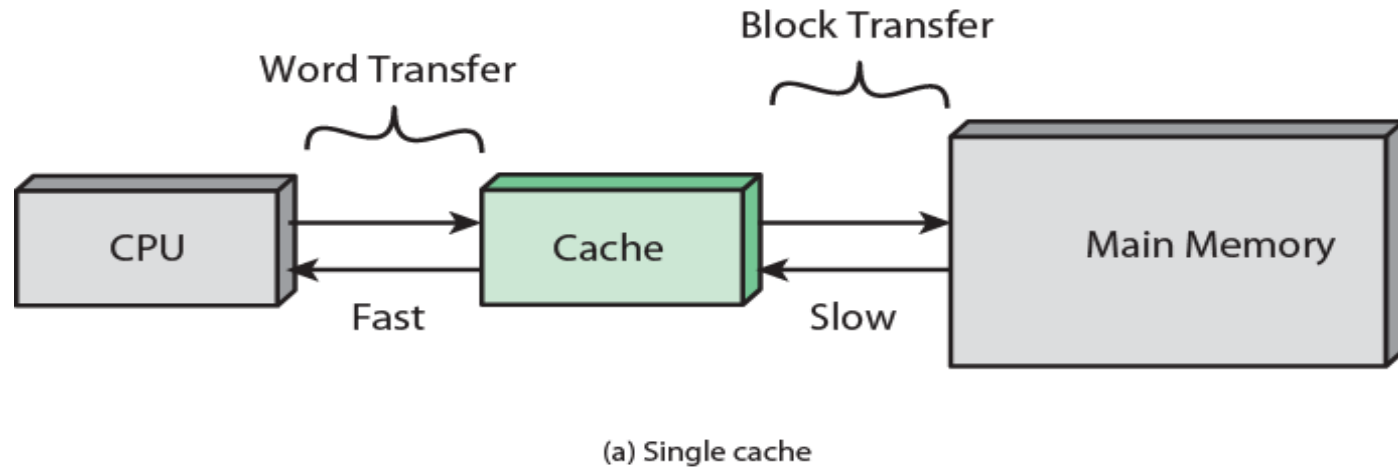
- Registers
- L1 Cache
- L2 Cache
- L3 Cache
- Main memory
- Disk cache
- Disk
- Optical
- Tape

# CACHE MEMORY

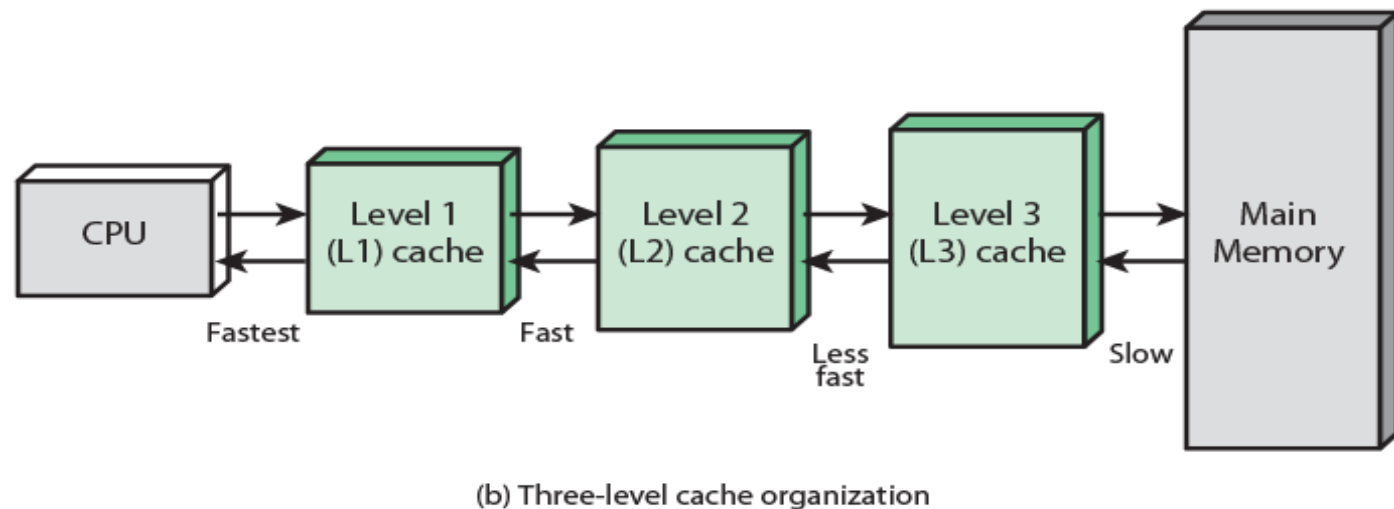
---

# Cache and Main Memory

- ▶ A relatively *large and slow* main memory together with a smaller, faster cache memory

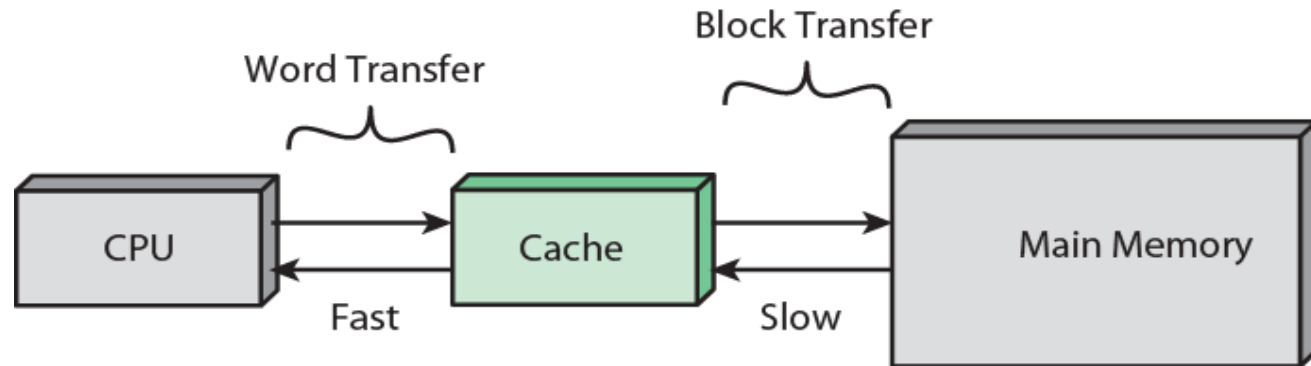


- ▶ The cache contains a copy of portions of main memory

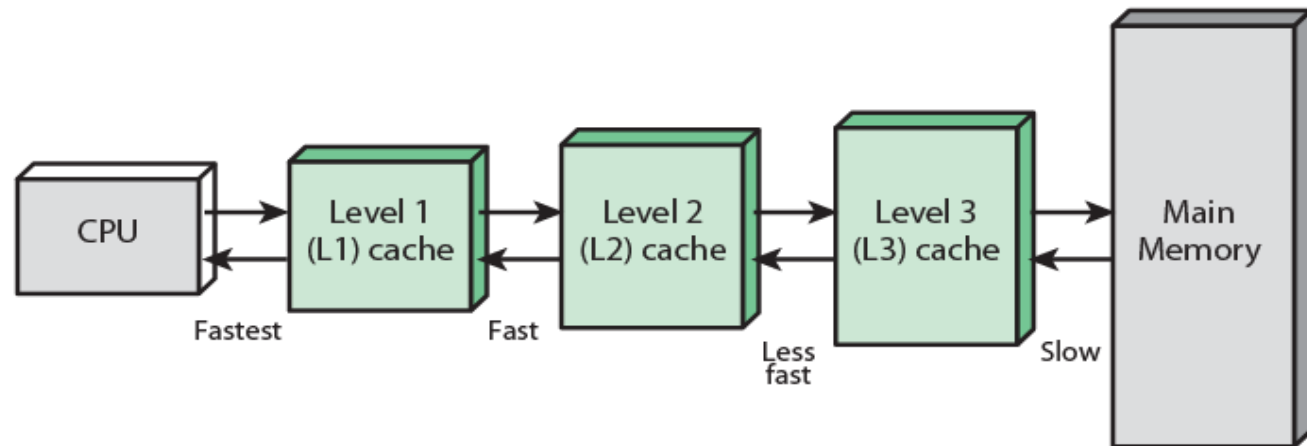


# Cache and Main Memory

- ▶ *Multiple levels of cache*
- ▶ The L2 cache is slower and typically larger than the L1 cache
- ▶ The L3 cache is slower and typically larger than the L2 cache



(a) Single cache

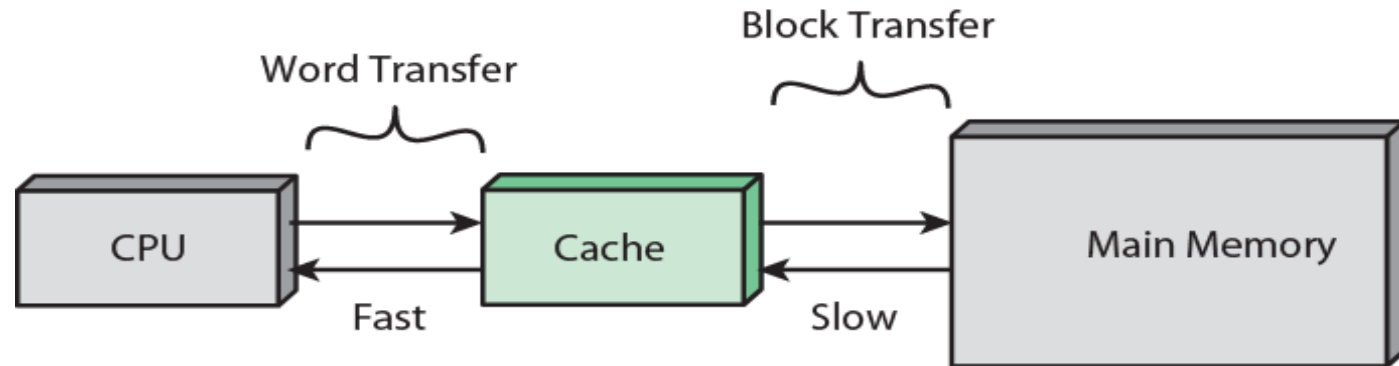


(b) Three-level cache organization

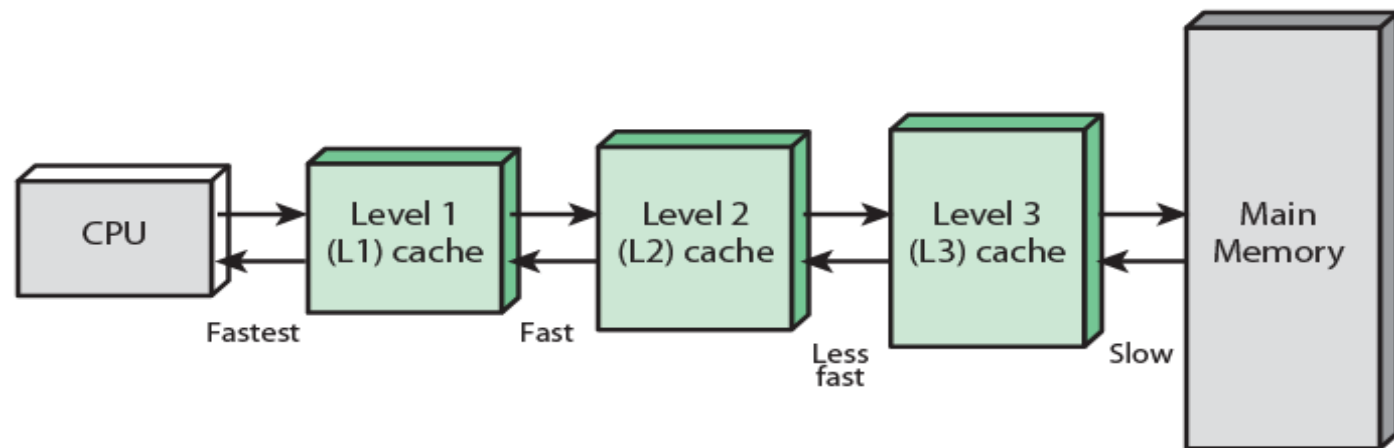


# Cache and Main Memory

- ▶ When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache



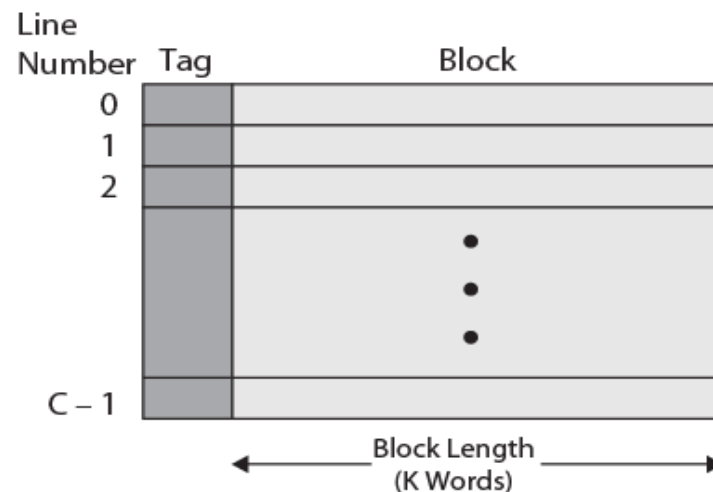
(a) Single cache



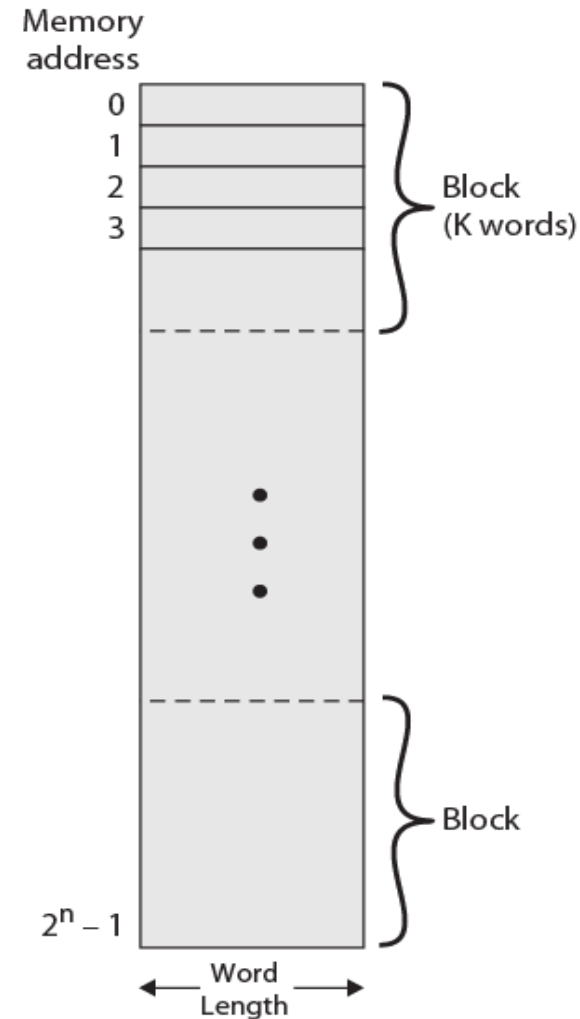
(b) Three-level cache organization

# Cache/Main Memory Structure

- ▶ Main memory - up to  $2^n$  addressable words
- ▶ Each word - *unique*  $n$ -bit address
- ▶ Main memory is considered to consist of
- ▶  $M$  blocks of  $K$  words each  $\rightarrow M=2^n/K$  blocks



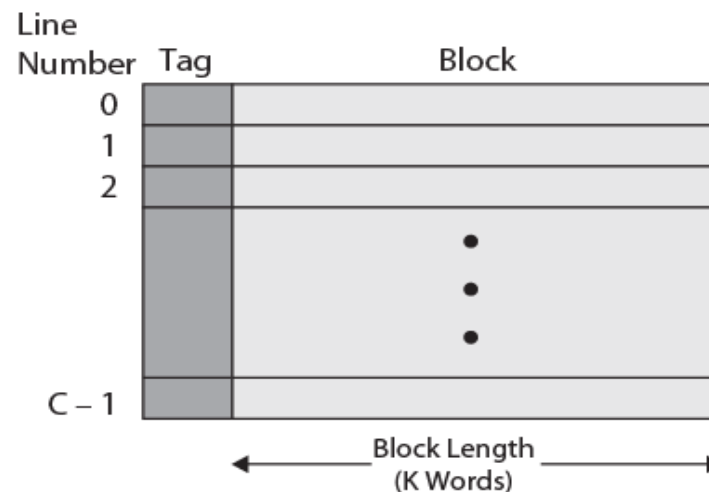
(a) Cache



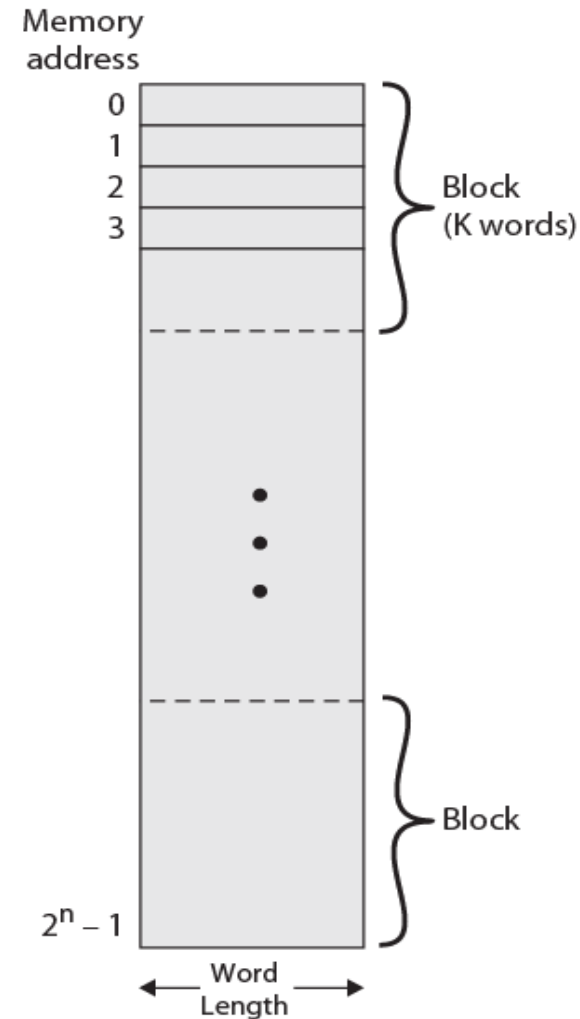
(b) Main memory

# Cache/Main Memory Structure

- ▶ The cache consists of  $C$  lines
- ▶ Each line contains  $K$  words, plus a tag
- ▶ Each line of cache corresponds to a block in main memory



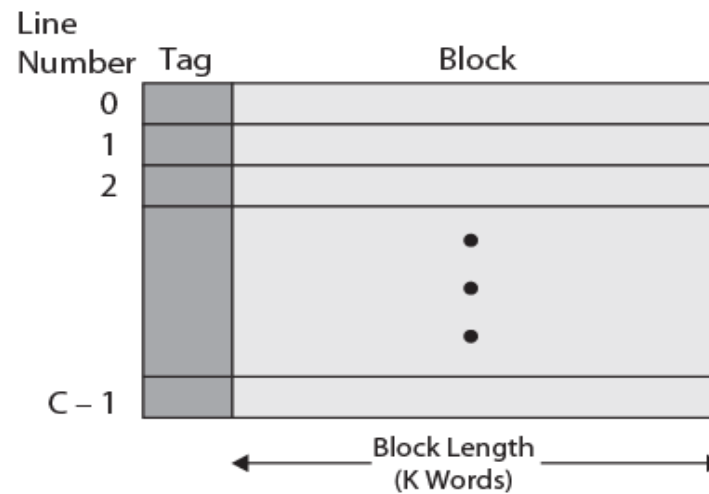
(a) Cache



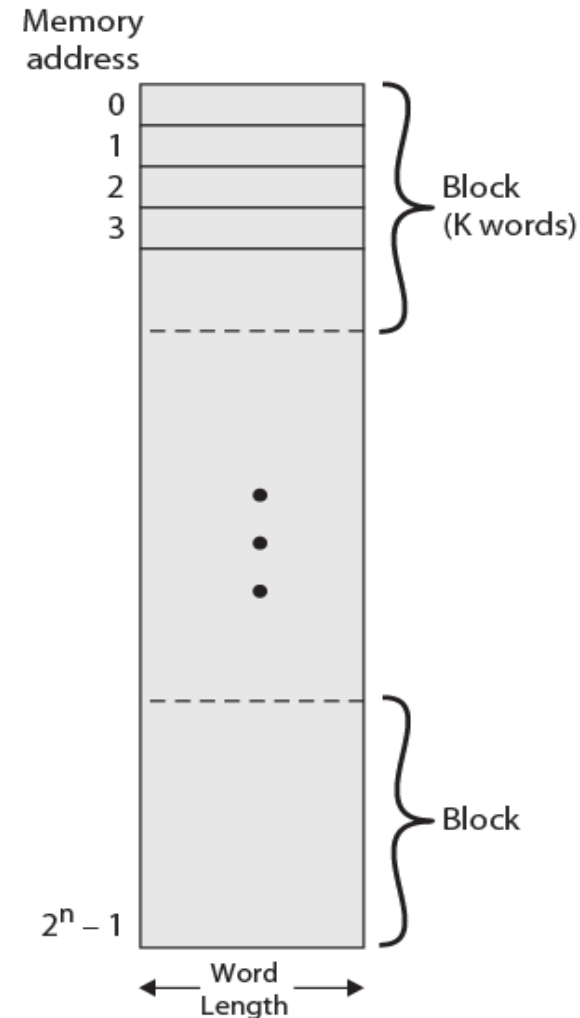
(b) Main memory

# Cache/Main Memory Structure

- ▶ The *number of lines* is *considerably less* than the *number of main memory blocks*
- ▶ At any time, some subset of the blocks of memory resides in lines in the cache



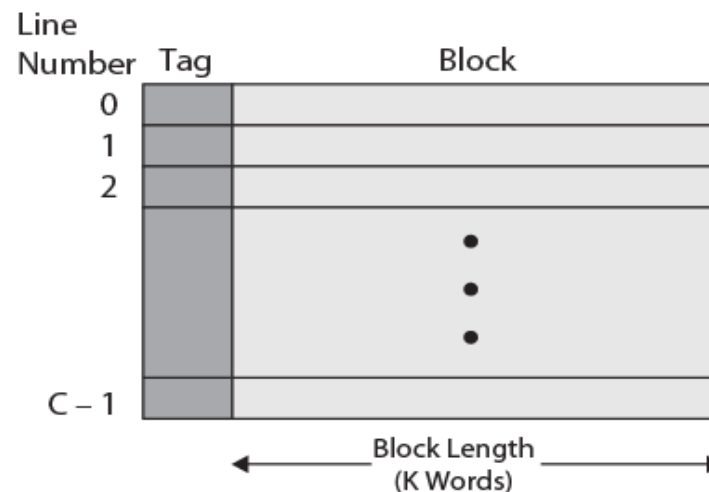
(a) Cache



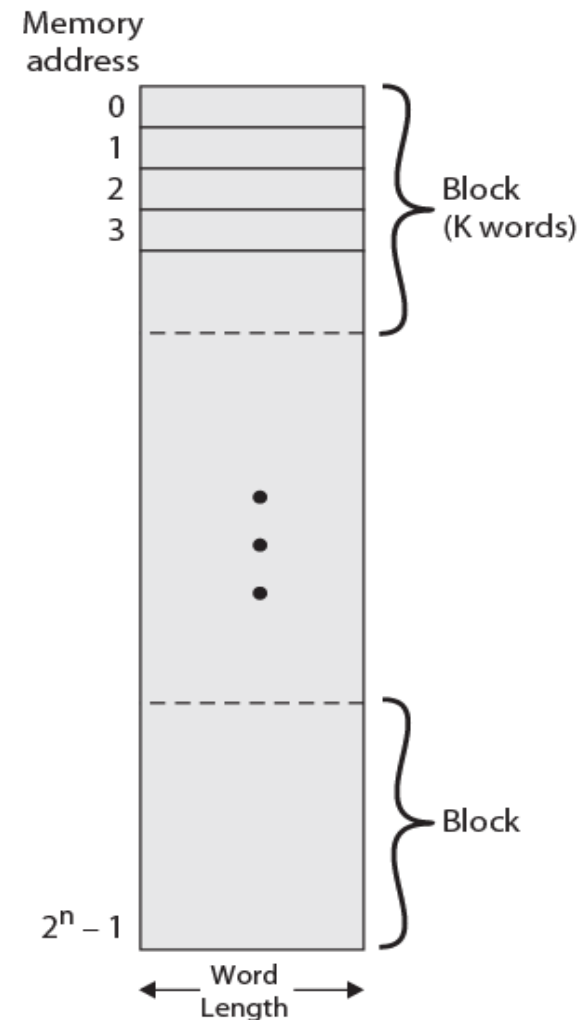
(b) Main memory

# Cache/Main Memory Structure

- ▶ If a word in a block of memory is read, that block is transferred to one of cache lines
- ▶ An individual line cannot be uniquely and permanently dedicated to a particular block  
→ tag identifying the block is being stored



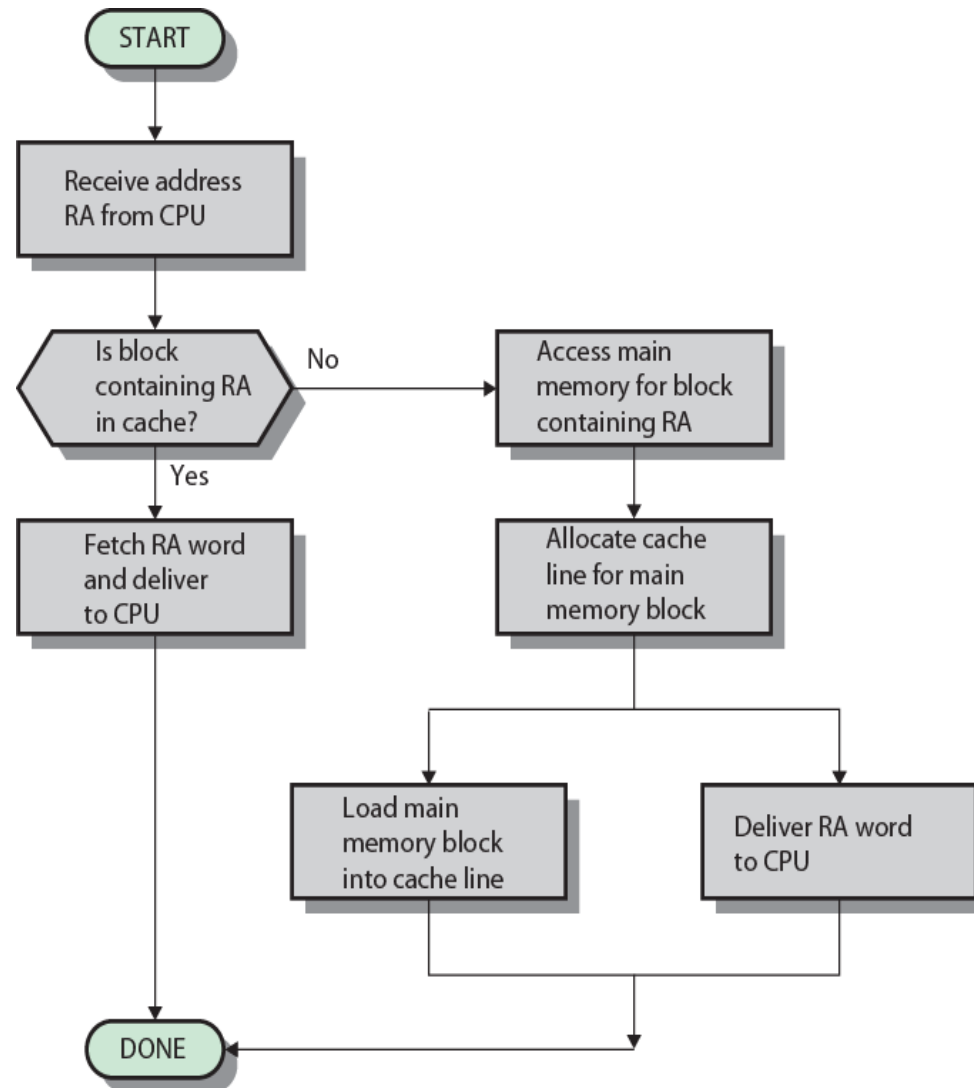
(a) Cache



(b) Main memory

# Cache – Read operation

- CPU requests contents of memory location
- Check cache for this data
- If present:
  - get from cache
  - else read required block from main memory to cache
- Then deliver to CPU

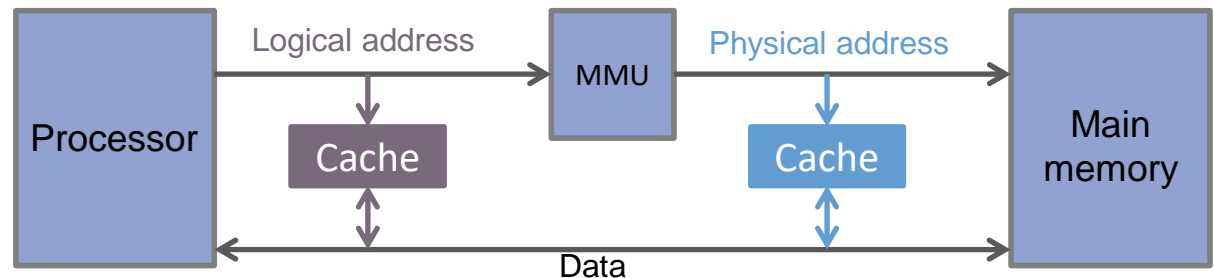


# Cache Design

- Addressing
- Size
- Mapping Function
  - Direct
  - Associative
  - Set Associative
- Replacement Algorithm
  - Least recently used (LRU)
  - First in first out (FIFO)
  - Least frequently used (LFU)
  - Random
- Write Policy
  - Write through
  - Write back
  - Write once
- Line(Block) Size
- Number of Caches
  - Levels
  - Unified or split

# Cache Addresses

- Cache can be located
  - **Between processor and virtual MMU**
  - **Between MMU and main memory**
- **Logical cache** (virtual cache) stores data using virtual addresses
  - Processor accesses cache directly, not thorough physical cache
  - Cache access faster, before MMU address translation
  - Virtual addresses use same address space for different applications
- **Physical cache** stores data using main memory physical addresses



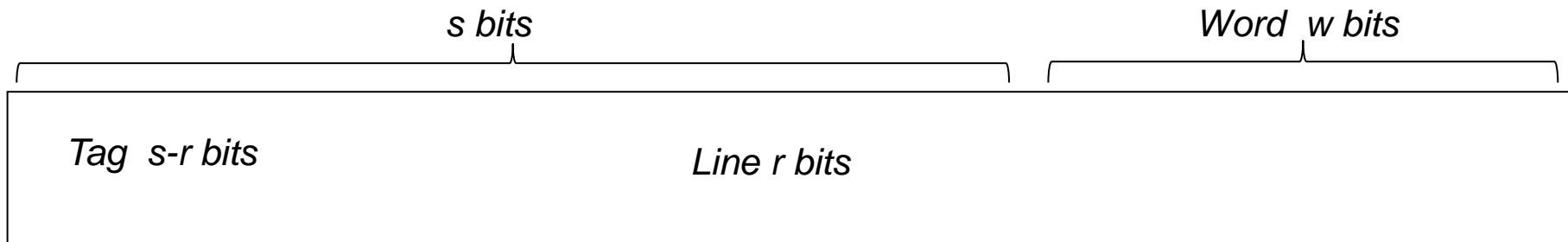


# Mapping function

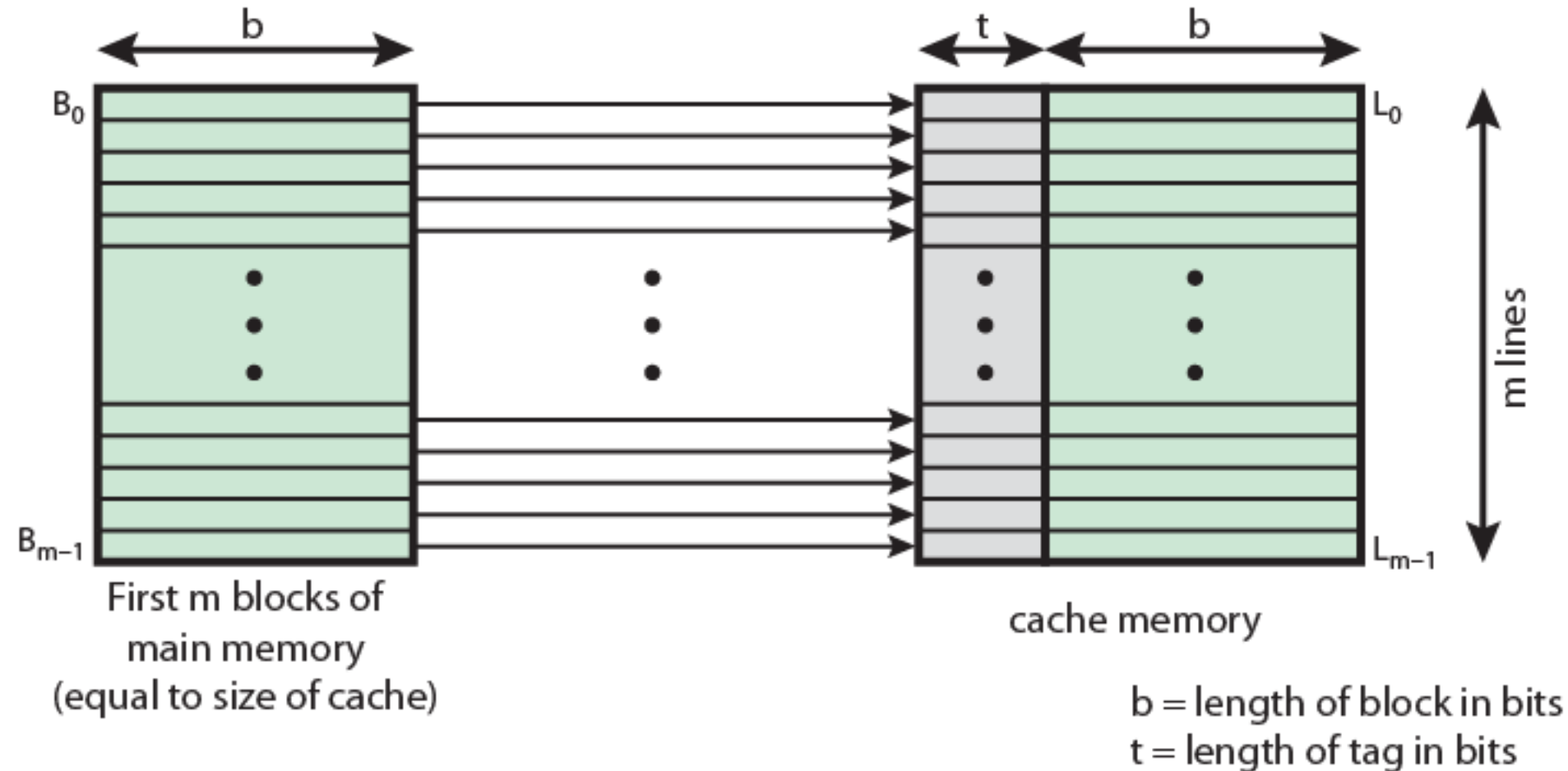
- Fewer cache lines than main memory blocks:
  - ***algorithm for mapping main memory blocks into cache lines***
  - ***means for determining which main memory block currently occupies a cache line***
- Mapping function → cache organization
- Three techniques can be used:
  - Direct
  - Associative
  - Set associative

# Direct Mapping

- Each block of main memory maps to only one cache line
  - **i.e. if a block is in cache, it must be in one specific place**
- Main memory address can be divided in two parts fields:
  - Least significant **w bits** identify unique word
  - Most significant **s bits** specify one of the  $2^s$  memory blocks:
    - cache line - r bits
    - tag - (s-r) bits

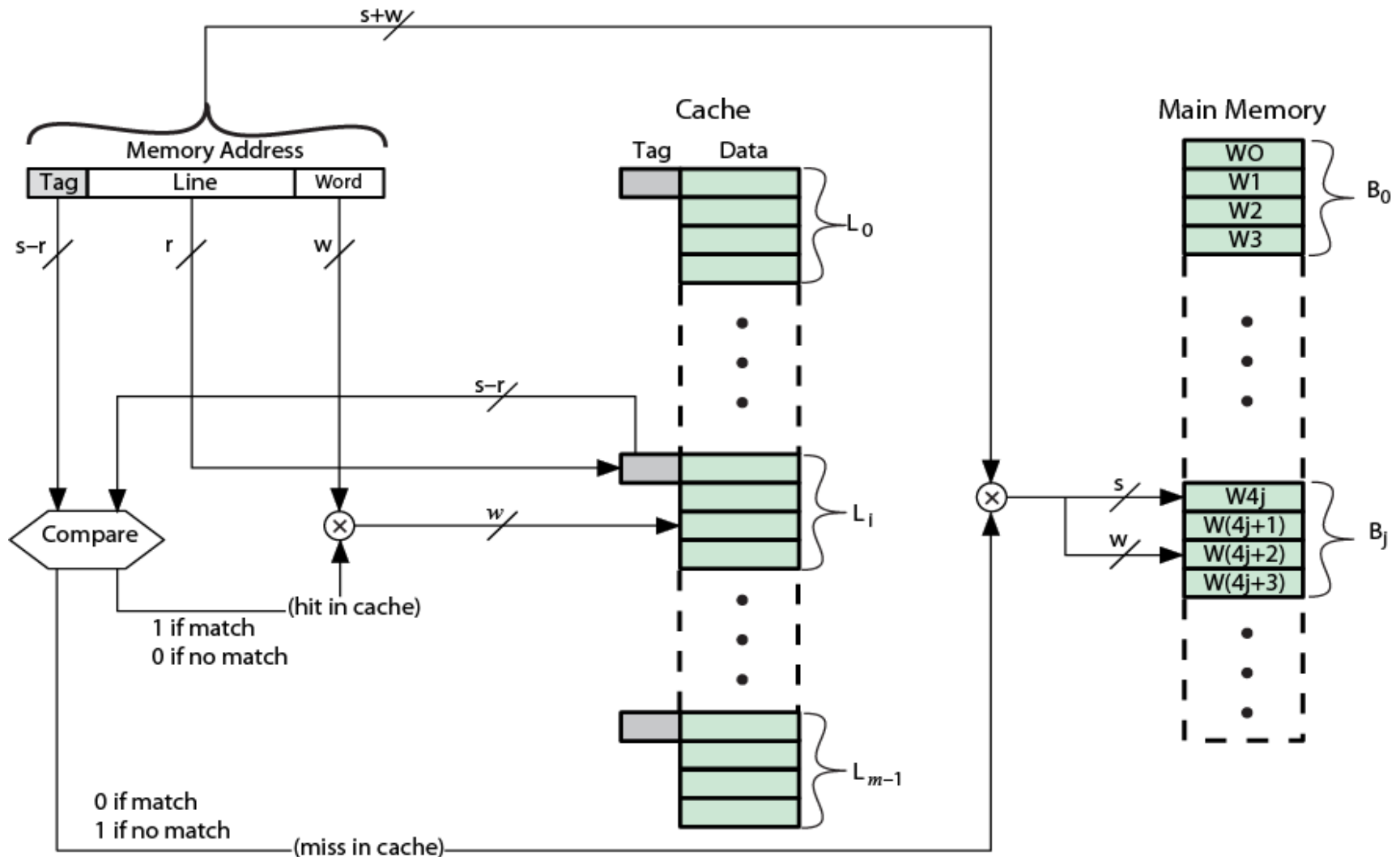


# Direct Mapping from Cache to Main Memory



(a) Direct mapping

# Direct Mapping Cache Organization



# Direct Mapping pros & cons

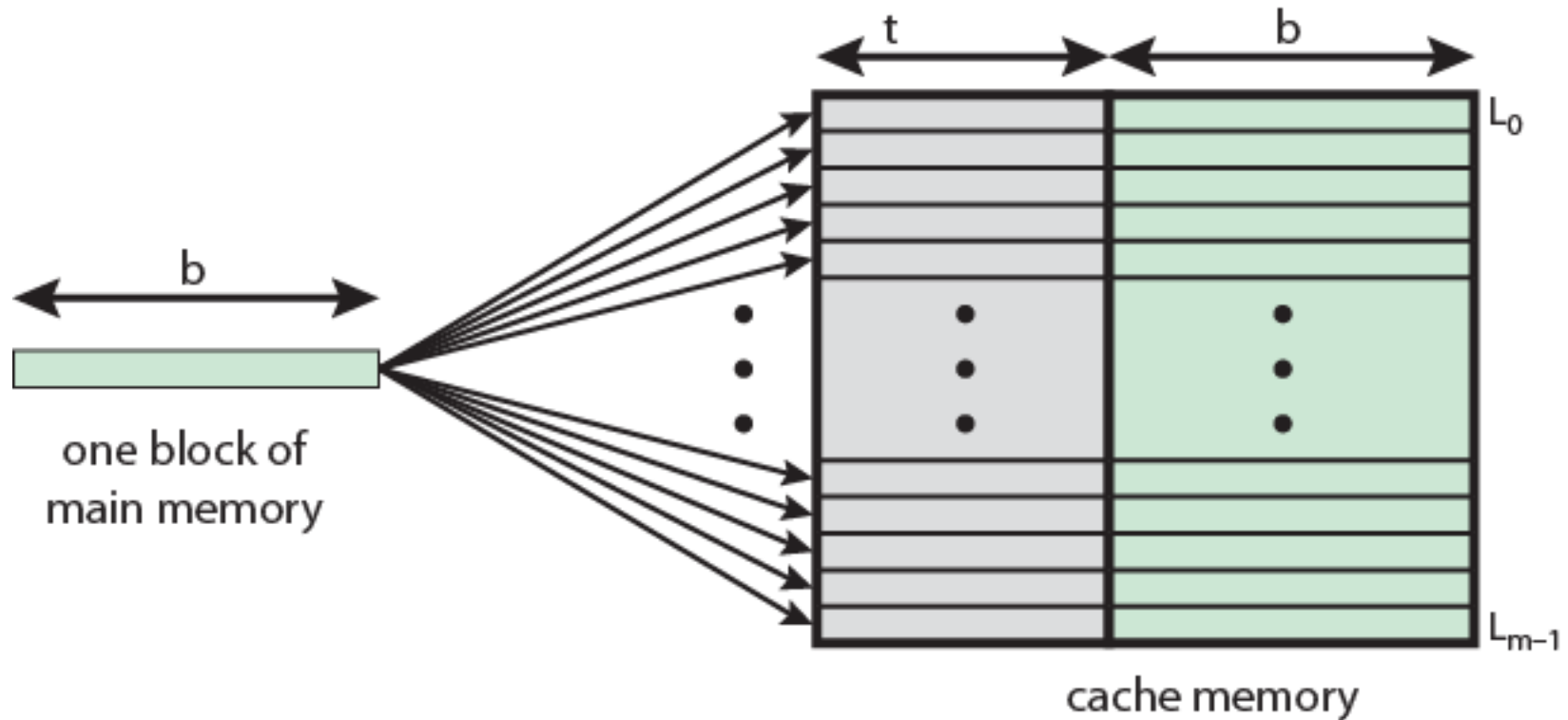
- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

# Associative Mapping

- A main **memory block** can be loaded into **any line of cache**
- Memory address is interpreted as
  - **Tag** field
  - **Word** field
- Tag uniquely identifies block of memory
- No field in the address corresponds to the line number

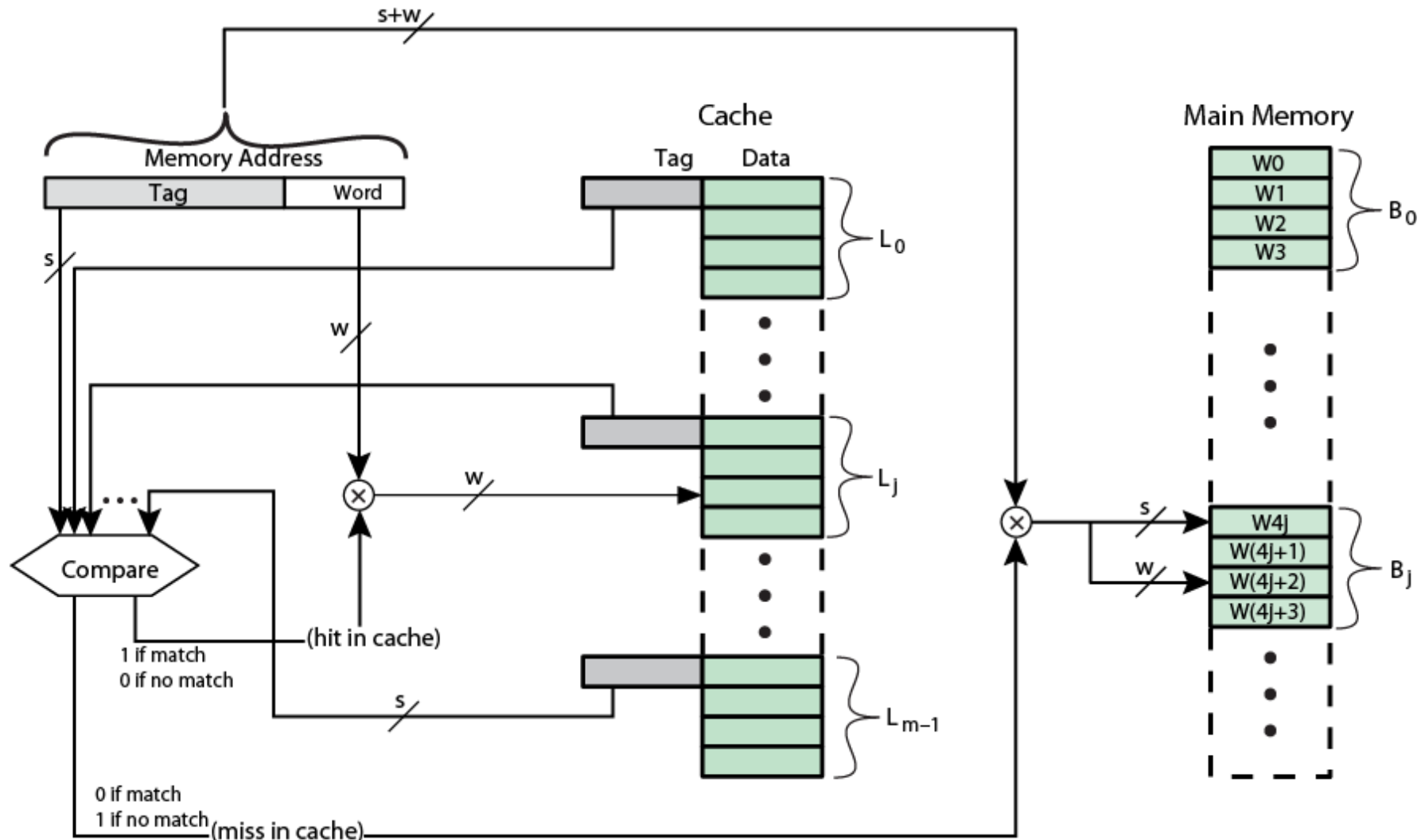
Tag	Word
-----	------

# Associative Mapping from Cache to Main Mem



- ▶ To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match

# Fully Associative Cache Organization





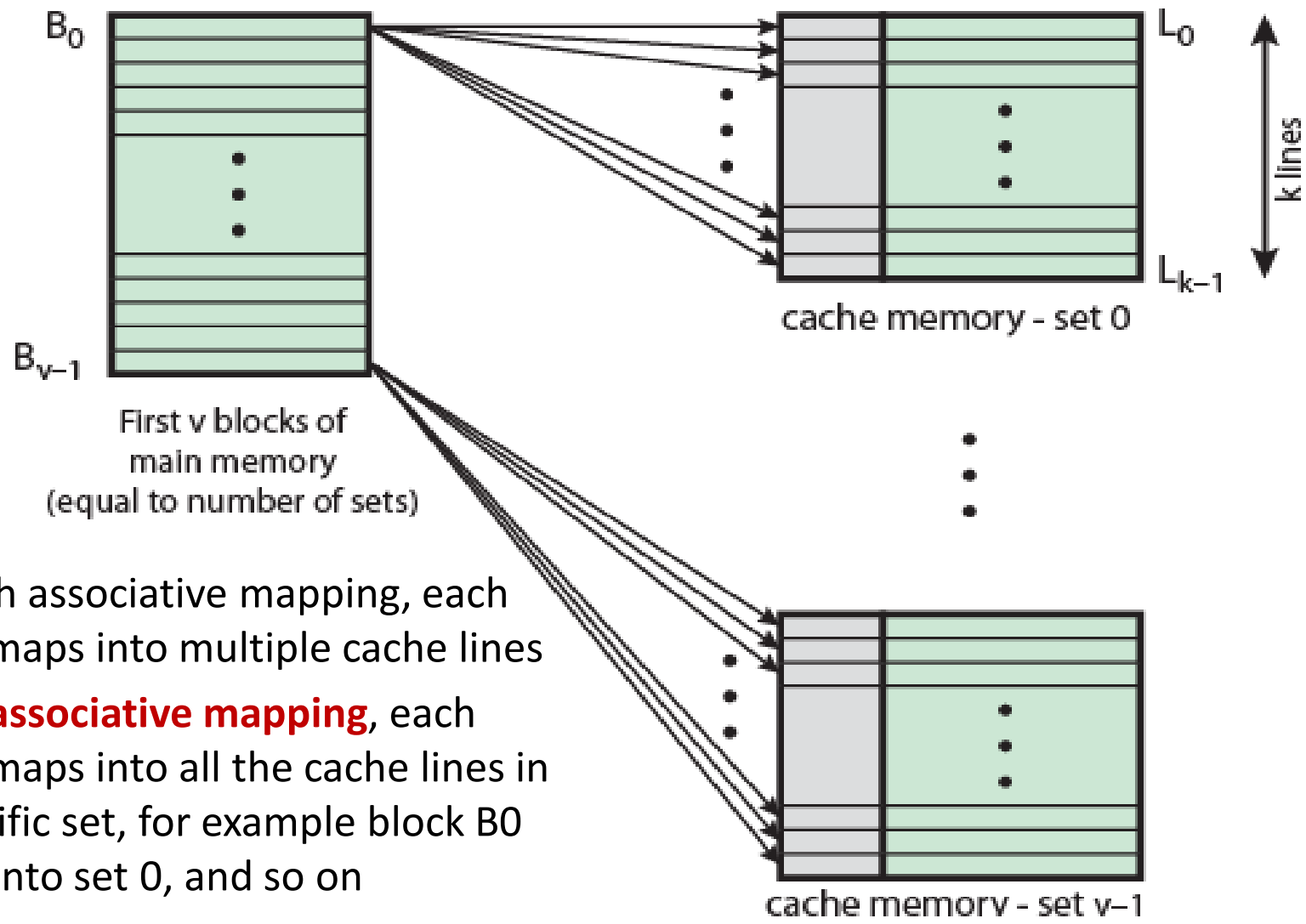
# Associative Mapping Summary

- Address length =  $(s + w)$  bits
- Number of addressable units =  $2^{s+w}$  words or bytes
- Block size = line size =  $2^w$  words or bytes
- Number of blocks in main memory =  $2^{s+w}/2^w = 2^s$
- Number of lines in cache = undetermined
- Size of tag =  $s$  bits

# Set Associative Mapping

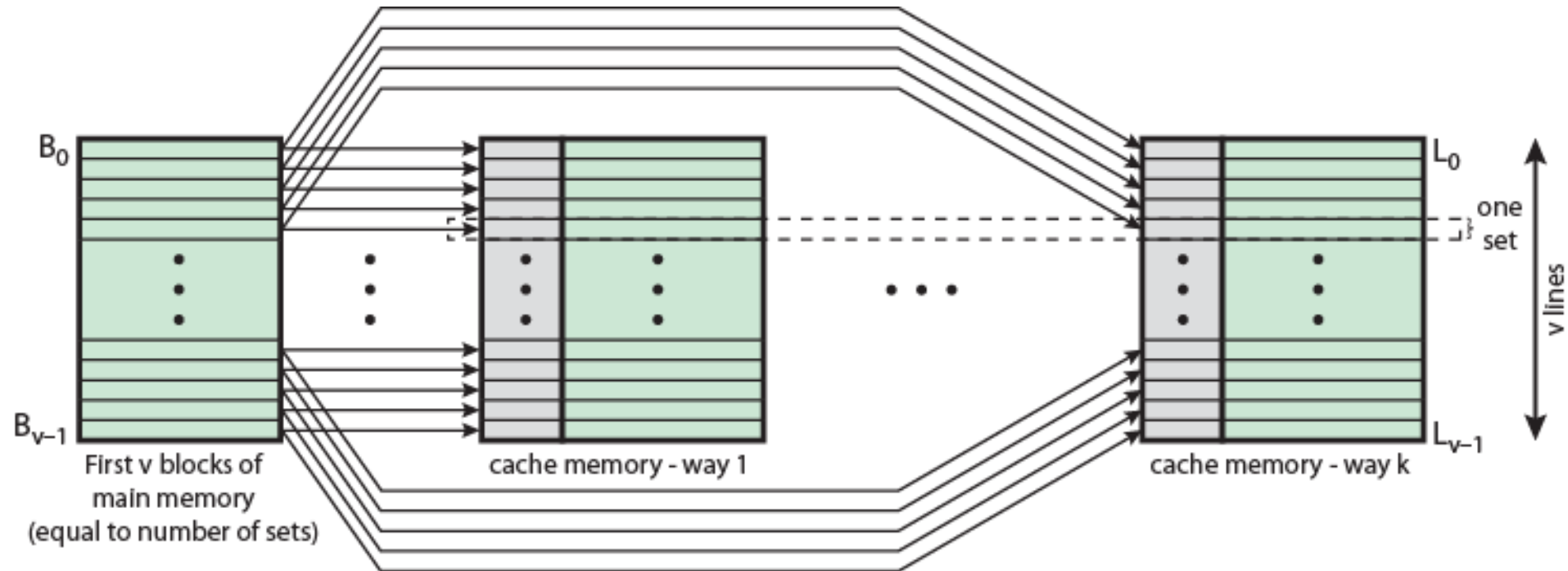
- Set-associative mapping exhibits the **strengths of both** the **direct** and **associative** approaches while reducing their disadvantages
- Cache is divided into a number of sets
- Each set contains a number of lines
- The set-associative cache can be physically implemented as:
  - **$v$  Associative-mapped caches**
    - in this case we have  $v$  associative caches
  - **$k$  Direct-mapped caches**

# $v$ Associative-mapped cache



- As with associative mapping, each word maps into multiple cache lines
- For  **$v$ -associative mapping**, each word maps into all the cache lines in a specific set, for example block  $B_0$  maps into set 0, and so on

# $k$ Associative-mapped ache



- Each direct-mapped cache is referred to as a *way*, consisting of lines
- The first lines of main memory are direct mapped into the lines of each way; the next group of lines are similarly mapped, and so on

# Set Associative Mapping

- The direct-mapped implementation is typically used for small degrees of associativity (small values of  $k$ )
- The associative-mapped implementation is typically used for higher degrees of associativity

# Replacement Algorithms

- Direct mapping
  - No choice
  - Each block only maps to one line
  - Replace that line
- Associative & Set Associative
  - Hardware implemented algorithm (speed)
  - **Least Recently used** (LRU)
    - but in 2 way set associative “Which of the 2 block is LRU?”
  - **First in first out** (FIFO)
  - **Least frequently used**
    - replace block which has had fewest hits
  - **Random**

# Write Policy

- **Write through**

- All writes go to main memory as well as cache
- Lots of traffic
- Slows down writes

- **Write back**

- Updates initially made in cache only and *update bit* is set
- If block is to be replaced, write to main memory if update bit
- Other caches get out of sync
- I/O must access main memory through cache

# Write Policy

- In a bus organization in which:
  - **more than one device** (typically a processor) has a **cache**
  - **main memory is shared**a new problem is introduced
- If data in one cache are altered, this invalidates:
  - **not only** the corresponding word in **main memory**
  - **but also** that same word in **other caches** (if any other cache happens to have that word)
- There are different possible approaches to maintain cache coherency



# Line Size

- When a **block of data** is retrieved and placed in the cache
  - not only the desired word is retrieved
  - but also some number of adjacent words
- Increased block size will increase hit ratio
  - principle of locality
- Hit ratio will decrease as block becomes even bigger
  - Probability of using newly fetched information becomes less than probability of reusing replaced

# Line Size

- Larger blocks
  - Reduce number of blocks that fit in cache
  - Data overwritten shortly after being fetched
  - Each additional word is less local so less likely to be needed
- No definitive optimum value has been found
  - 8 to 64 bytes seems reasonable close to optimum
  - For HPC systems, 64 and 128 byte most common

# Multilevel Caches

- The use of **multiple caches** has become the **norm**
- High logic density enables **caches on chip**
  - Faster than bus access
  - When the requested instruction or data is found in the on-chip cache, the bus access is eliminated
  - Bus is free for other transfers
- Common to use both **on and off chip cache**
  - L1 on chip, L2 off chip in static RAM
  - L2 access much faster than DRAM or ROM
  - L2 often uses separate data path
  - L2 may be on chip
  - Resulting in L3 cache

# Unified versus Split Caches

- It is quite common to **split the cache into two**:
  - one dedicated to instructions
  - one dedicated to data
- These two caches both exist at the same level, typically as two L1 caches
  - When the processor attempts to fetch an **instruction** from main memory, it first consults the **instruction L1 cache**
  - when the processor attempts to fetch **data** from main memory, it first consults the **data L1 cache**

# Unified versus Split Caches

- Advantages of **unified cache**
  - Higher hit rate
    - Balances load of instruction and data fetch
      - if many more instruction fetches are involved in the execution, then the cache will tend to fill up with instructions
      - if an execution pattern involves relatively more data fetches, the opposite will occur
  - Only one cache to design & implement
- Advantages of **split cache**
  - Eliminates cache contention between instruction fetch/decode unit and execution unit
    - Important in pipelining