Chapter 12 – Part 1

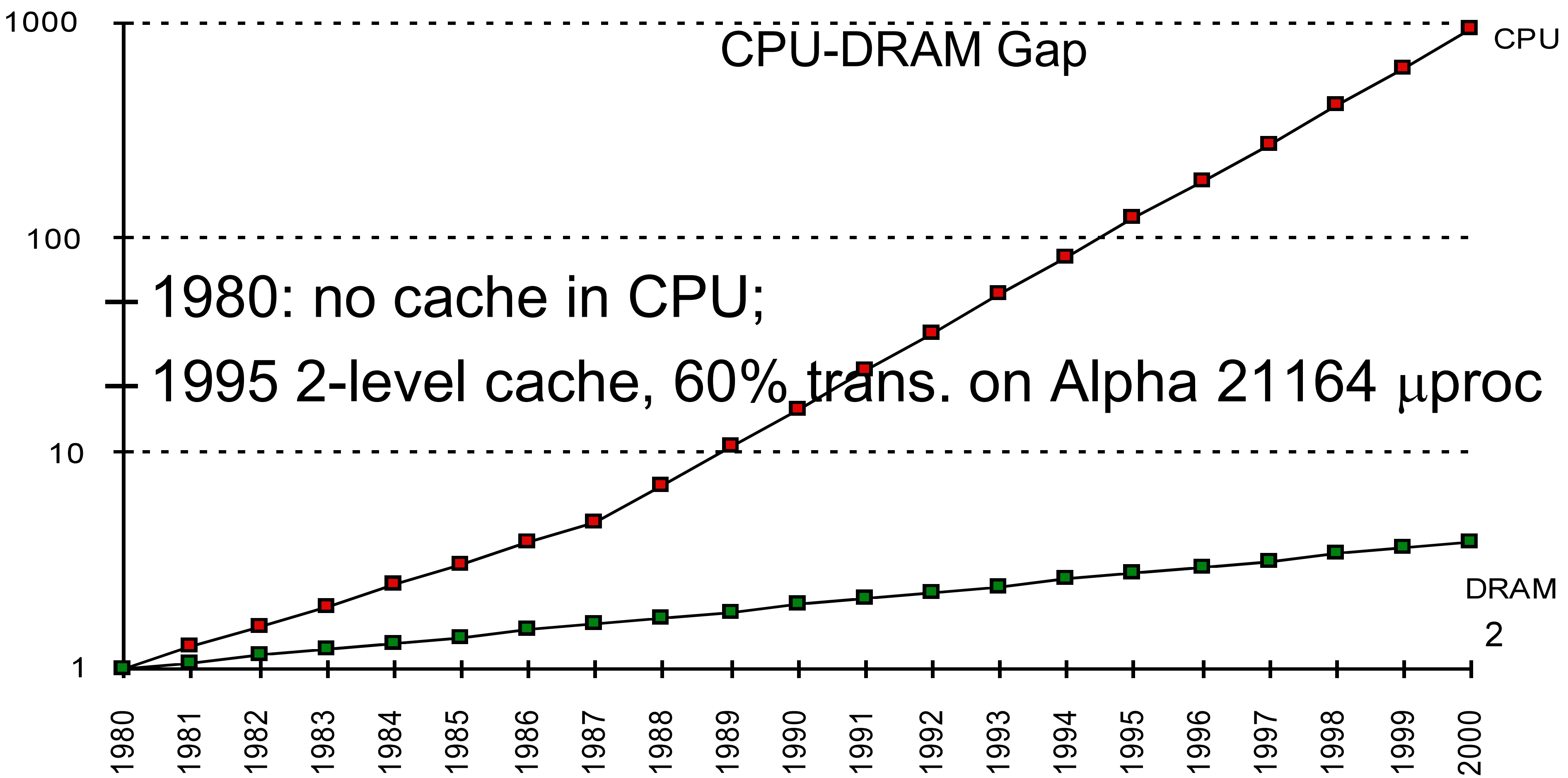
Memory Organization

Based on slides by:
Prof. Myung-Eui Lee
Korea University of Technology & Education
Department of Information & Communication

Alon Schclar, Tel-Aviv College, 2009

Why is memory important?

- **CPU performance** has **increased** at a much **faster** rate **than memory performance**, making main memory the bottleneck.



CPU-DRAM Gap

1980: no cache in CPU;

1995 2-level cache, 60% trans. on Alpha 21164 µproc

Memory Hierarchy

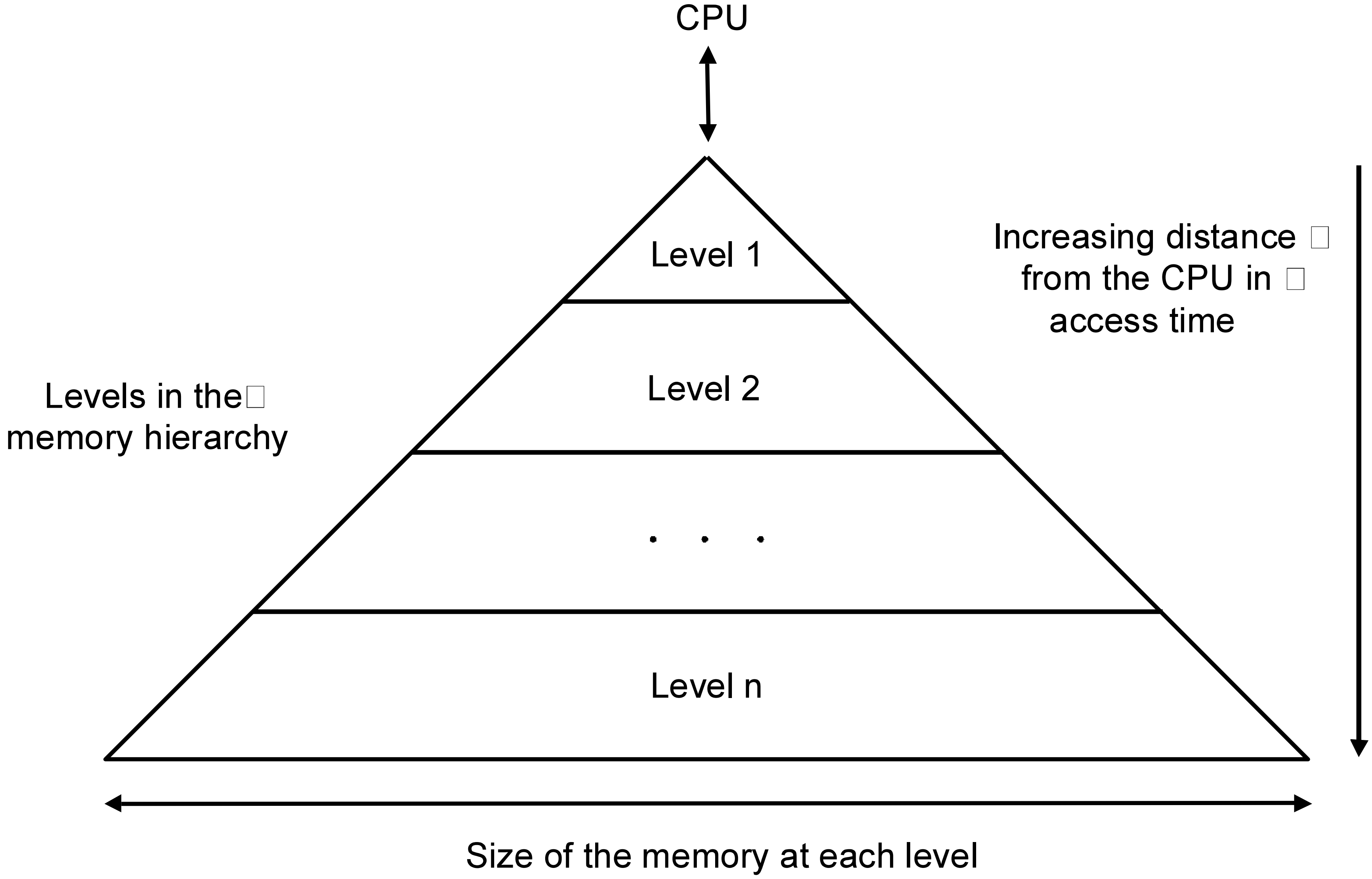
- A memory hierarchy in which the
  - **faster** but **smaller** part is “**close**” to the **CPU** and **used most** of the **time**
  - **Slower** but **larger** part is ‘’**far**” from the **CPU**, will give us the **illusion** of having a **fast large inexpensive** memory
- Data flow
  - **slower** → **faster** when is needed
  - **faster** → **slower** when is not needed

Memory Hierarchy

	Type	Access time	Capacity	Price
Cache	SRAM	2 – 25 ns	1-4 MB	
Memory	DRAM	40 – 80 ns	1-4 GB	
Auxiliary	Disk	10-20 **m**s	0.5-1 TB	

- Overall goal
  - **highest-possible average access speed** while
  - **minimizing** the total **cost** of the **entire memory**

Memory Hierarchy



CPU

Level 1

Level 2

. . .

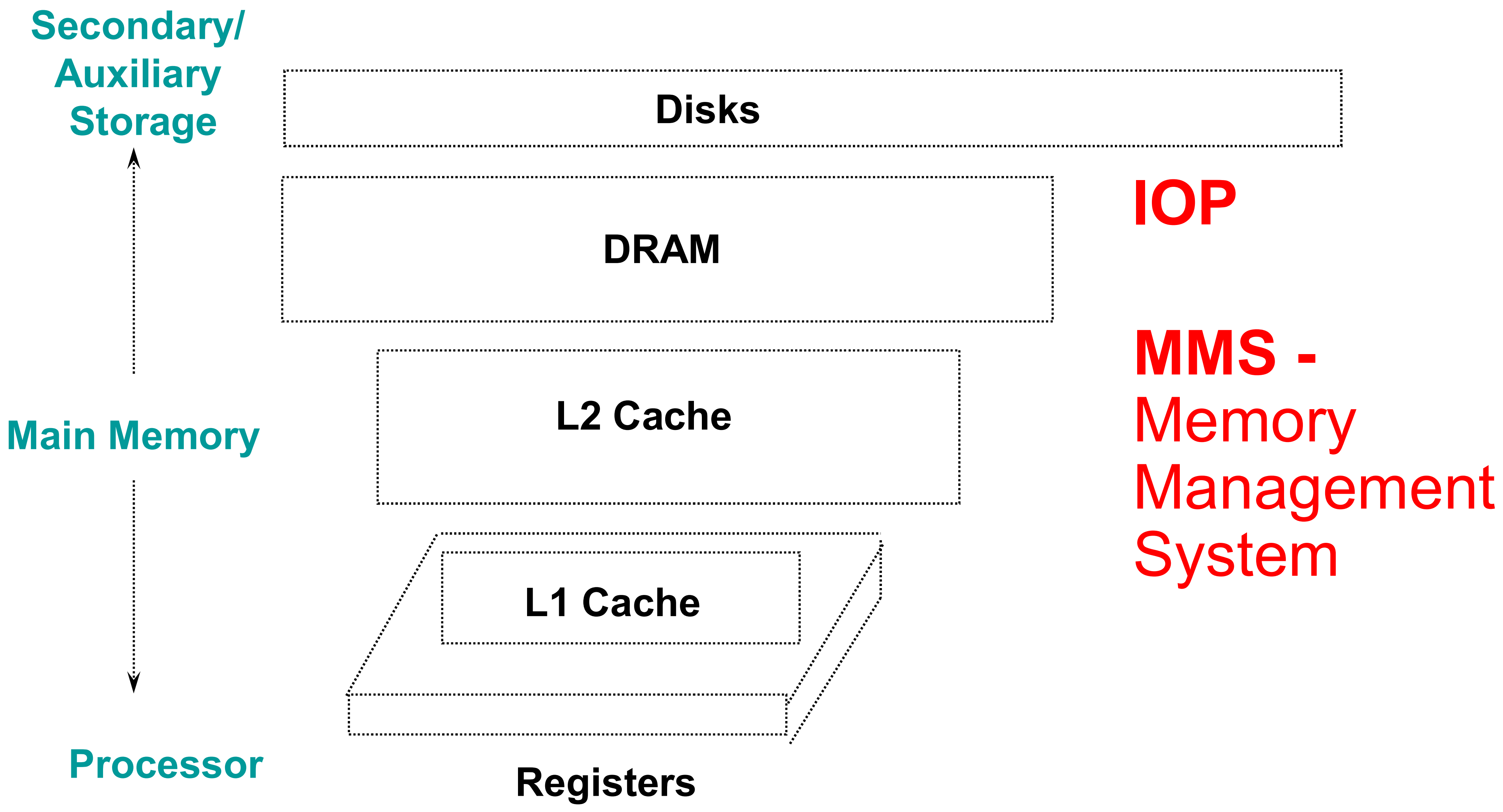
Level n

Levels in the□ memory hierarchy

Increasing distance □ from the CPU in □ access time

Size of the memory at each level

Memory Hierarchy



Secondary/
Auxiliary
Storage

Main Memory

Processor

Disks

DRAM

L2 Cache

L1 Cache

Registers

IOP

MMS -
Memory
Management
System

Cache

- **CPU logic** is usually **faster than** main **memory**
  - main memory is the bottleneck
- A **very-high-speed** memory **between** the **CPU** and **main** memory
- **Access time** is **close** to **processor** logic clock cycle time.
- **Holds current programs** and **data**
- **Available** to the CPU at a **rapid rate**
- **Resides in CPU**

Multiprogramming

- **OS feature**
- Enable the **CPU** to **process** a **number** of independent **programs *concurrently***
- **Two or more programs** exist in **different parts** of the memory **hierarchy** at the **same time**.
  - At least **one in main** memory
- Keep all parts of the computer **busy** by working with several programs in sequence.
  - For example, when an **I/O transfer** is required
  - The **CPU initiates** the IOP to start executing the transfer
  - **Meanwhile** the **CPU** is **free** to execute another program

Virtual memory

- Programs are sometimes **too long to fit** in main memory
  - In multiprogramming – **all programs do not fit** in main memory at the **same time**
- A **need** for **running partial** programs,
  - For **varying** the **amount** of main **memory** in **use** by a given **program**
  - and for **moving programs** around the memory **hierarchy**.

Virtual memory

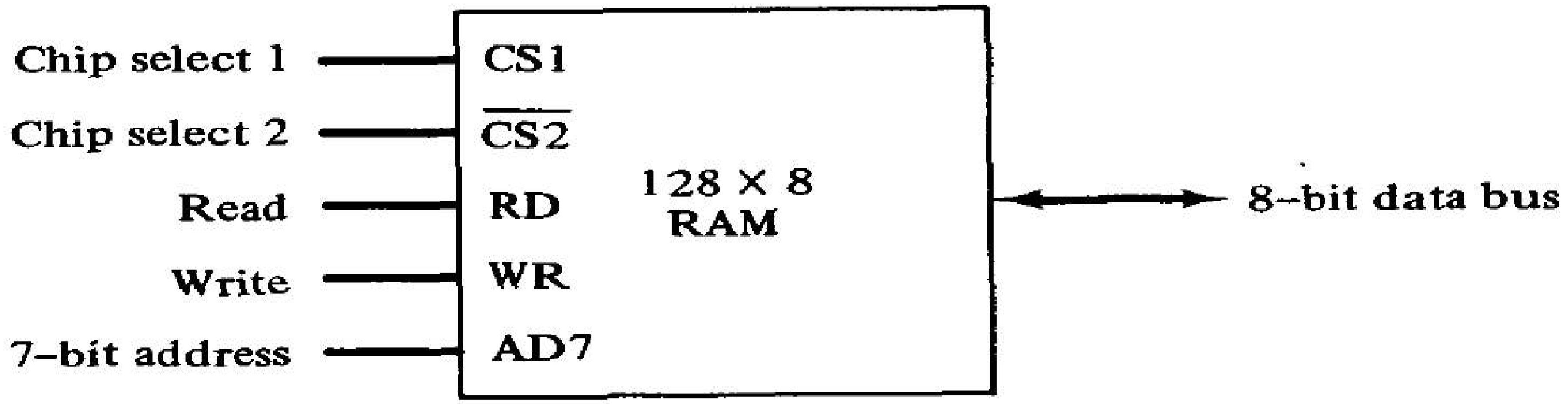
- A **program** and **its data** normally **resides** in **auxiliary** memory.
  - **When** the **program** or a **segment** of the program is to be **executed**, it is **transferred** to **main** memory
  - Thus **auxiliary** memory is an integral **extension** of **main** memory.
  - OS maintains in main memory currently active portions
  - ***Memory management system (MMS)***

RAM – Random access Memory

- ***SRAM*** - Static Random Access Memory
  - **FFs** that store the binary information.
  - Remains **valid** as long as **power is ON**
  - Fast (shorter R/W cycles) and **easier to use**
  - offers **reduced power consumption**
- ***DRAM*** - Dynamic RAM
  - Info stored as **electric charges** applied to **capacitors**.
  - The stored **charge tends to discharge** with time
  - Capacitors must be **periodically** recharged (**refreshing**)
    - Cycling through words every **few *ms*** to restore decaying charge.
  - Offers **larger storage** capacity on a **single** memory **chip**.

Typical RAM chip

Figure 12-2 Typical RAM chip.



(a) Block diagram

CS1	$\overline{\text{CS2}}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

RAM – cont.

- **More than one control input**
  - **For chip selection** when **multiple** chips are used (via **address decoding**)
- **Read and write** sometimes **combined** into **one** line labeled **R/W.**

ROM – Read Only Memory

- Has **random** access **w/o write** capabilities
- **ROM** content **remains** unchanged **regardless** of **power**
  - **RAM** is **volatile** - its **contents** is **lost** when **power** is turned **off**.
- Stores **information** that **does not change**
- Used for storing **programs** that are **permanently** resident in the computer
  - ***bootstrap loader.***

ROM – cont.

- **Organized** externally in a **similar** manner to **RAM**
- The **data bus** only **in output mode** (**no write**)
- More bits per ***mm*^{2}** than RAM
  - Different implementation

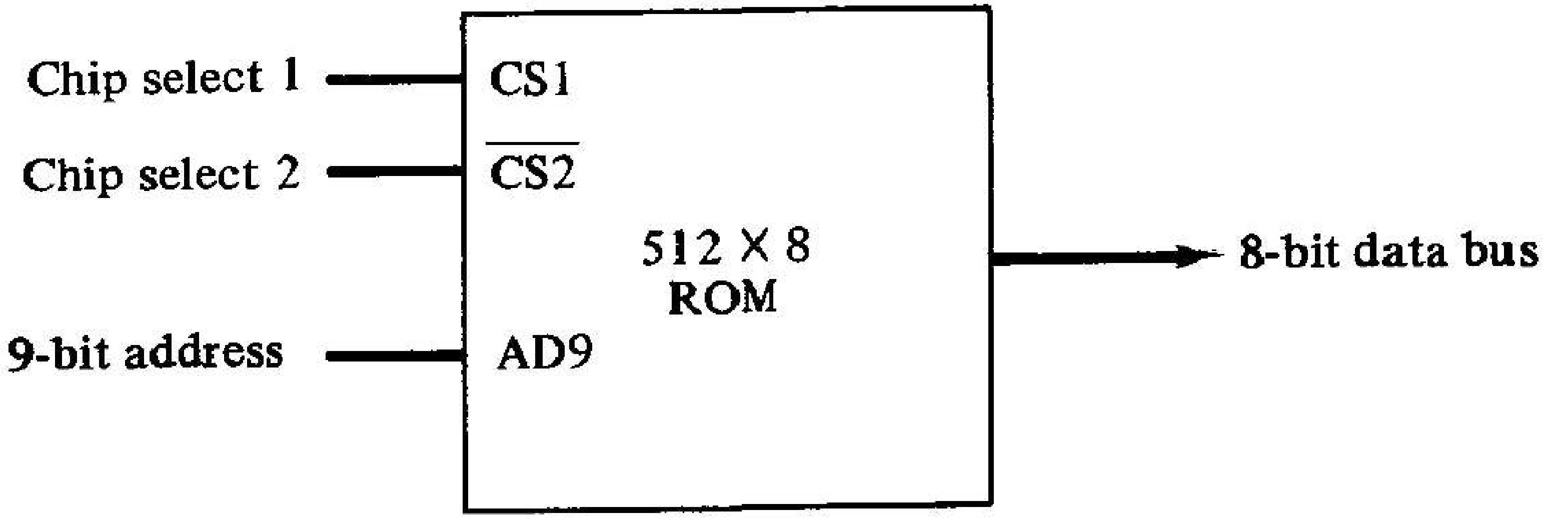


Figure 12-3 Typical ROM chip.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Bootstrap Loader

- **Program** that **starts** the **computer** operating when **power** is turned **on**.
  - **PC** is set to the **starting address** of **BL**
- **Loads** a **portion** of the **operating system** from **disk** to **main** memory
- **Control** is **transferred** to **OS**
  - **prepares** the **computer** for general **use**

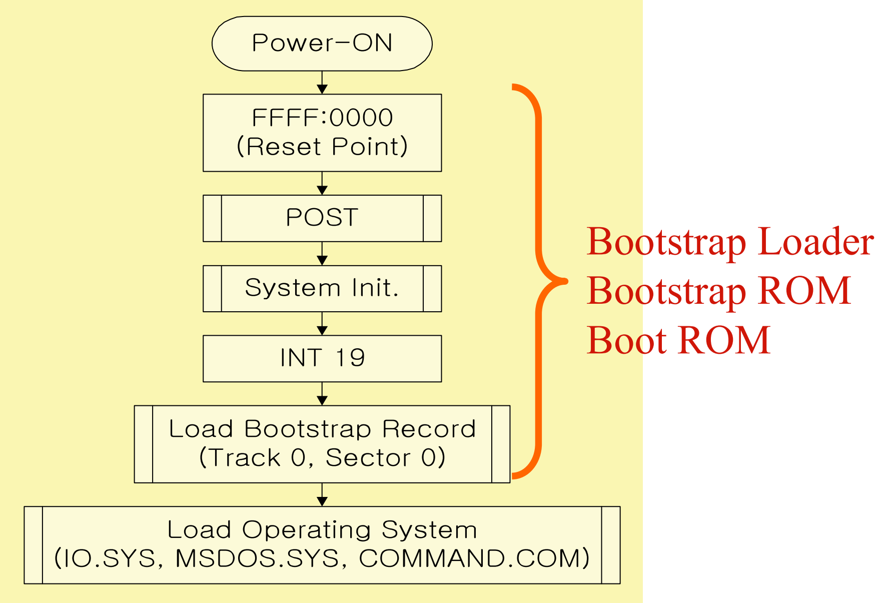
12-2 Main Memory

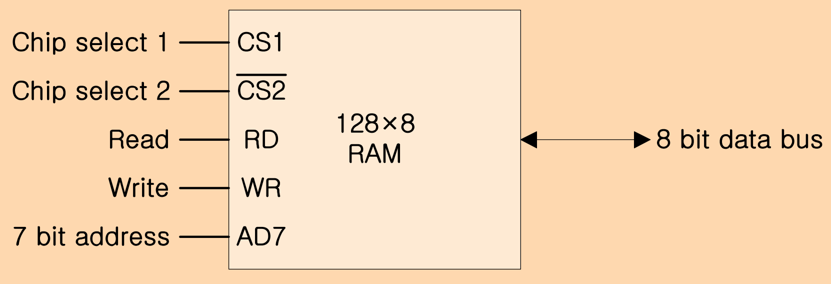
Bootstrap Loader

- A program whose function is to start the computer software operating when power is turned on

RAM and ROM Chips

- Typical RAM chip : ***Fig. 12-2***
  - » 128 X 8 RAM : $2^7 = 128$ (7 bit address lines)
- Typical ROM chip : ***Fig. 12-3***
  - » 512 X 8 ROM : $2^9 = 512$ (9 bit address lines)

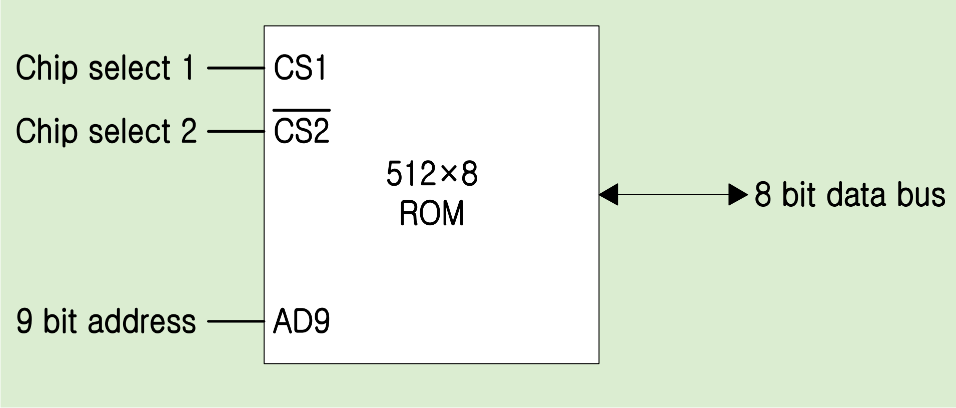

Power−ON
FFFF:0000 (Reset Point)
POST
System Init.
INT 19
Load Bootstrap Record (Track 0, Sector 0)
Load Operating System (IO.SYS, MSDOS.SYS, COMMAND.COM)
Bootstrap Loader
Bootstrap ROM
Boot ROM



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High−impedance
0	1	×	×	Inhibit	High−impedance
1	0	0	0	Inhibit	High−impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High−impedance

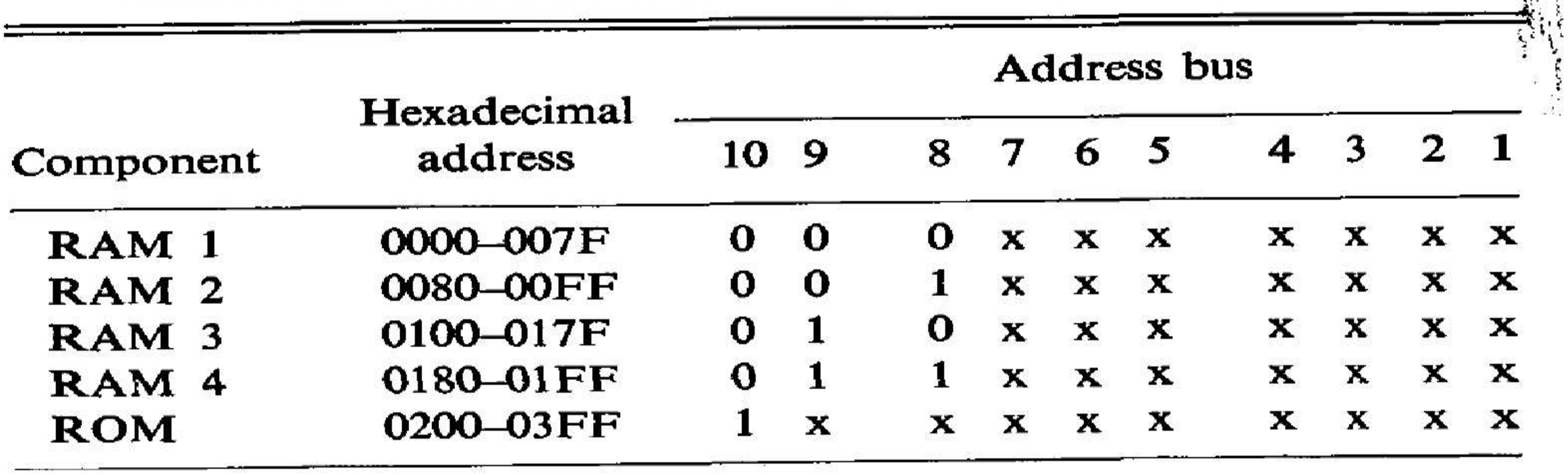
(b) Function table



Memory Map

- The computer **architect** calculates **required memory - RAM & ROM**.
- **Memory** and **processor** are **interconnected** according to
  - **memory size** and
  - RAM and ROM **chip types**
- ***Memory address map***
  - Table with address space of every memory chip.

TABLE 12-1 Memory Address Map for Microprocomputer



Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Memory Map – cont.

- **16 lines** in **address bus** – only **10** are **used** (**rest** is **zero**)
- **RAM chips** have **128 bytes - 7 address lines**
- The **ROM** chip has **512 bytes** - **9 address lines**
- **Distinguish 4 RAM** chips by assigning different addresses
  - **Lines 8 and 9** represent **four** distinct **combinations**
  - A **combination** per chip
  - **Any other pair** of unused bus lines **can** be **chosen**
- The **nine low-order bus lines** constitute **RAM space**: ***2^9 = 512 bytes*.**
- **RAM/ROM address distinction –** bus line#10.
  - **0/1** - CPU selects a **RAM/ROM**

Memory Connection to CPU

- **RAM & ROM connected** to **CPU** through **data** and **address buses**
  - The **low-order lines** in address bus select the **byte within the chips**
  - **other** address bus **lines select** a **chip** through its chip-select **(CS) inputs**.
- ***Example: memory capacity of 4X128=512 bytes of RAM and 512 bytes of ROM.***
  - Each **RAM chip receives**
    - the **7 low-order bits** of the **address bus** - select 1 of 128 possible bytes.
  - A **particular RAM chip** selected by **lines 8 & 9** in the address bus
    - a **2 x 4 decoder** whose outputs **go to the CS1** inputs in each RAM chip.
    - **When 00**, the **first RAM chip** is **selected**.
    - **When 01**, the **second RAM** chip is selected, and so on.
  - **RD & WR CPU-outputs** applied to the **inputs** of **every RAM chip**
  - **RAM/ROM select** through **bus line#10: 0/1 – RAM/ROM**

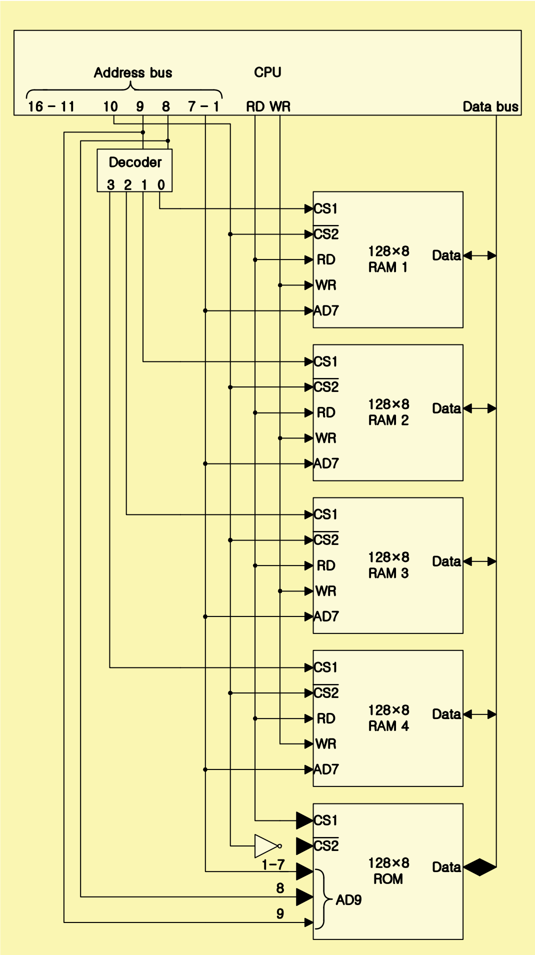
Memory Connection to CPU

- **CPU RD** goes to ***chip select*** input in ROM
  - **ROM** is **enabled only during** a **read** operation.
- Address **bus lines#1-9** go to ROM **w/o decoder**
  - Addresses **0 - 511** to **RAM**
  - Addresses **512 - 1023** to **ROM**
  - **Data bus**
    - **ROM** – **only output** capability,
    - **RAM** – **bidirectional** info transfer
- If **more ROM/RAM** chips

  → **more** external **decoders** / **bigger decoder**
- Architect must determine a **memory map**
  - addresses to chips

Memory Address Map

- Memory Configuration : **512** bytes **RAM** + **512** bytes **ROM**
  - » **1** x **512** byte **ROM** + **4** x **128** bytes RAM
- Memory Address Map : ***Tab. 12-1***
  - » Address line 9 8
    - RAM 1 **0 0** : 0000 - 007F
    - RAM 1 **0 1** : 0080 - 00FF
    - RAM 1 **1 0** : 0100 - 017F
    - RAM 1 **1 1** : 0180 - 01FF
  - » Address line 10
    - ROM **1** : 0200 - 03FF
- Memory Connection to CPU : ***Fig. 12-4***
  - » 2 x 4 Decoder : RAM select (**CS1**)
  - » Address line 10
    - RAM select : $\overline{\textbf{CS2}}$
    - ROM select : **CS2** 의 **Invert**



Memory connection to the CPU

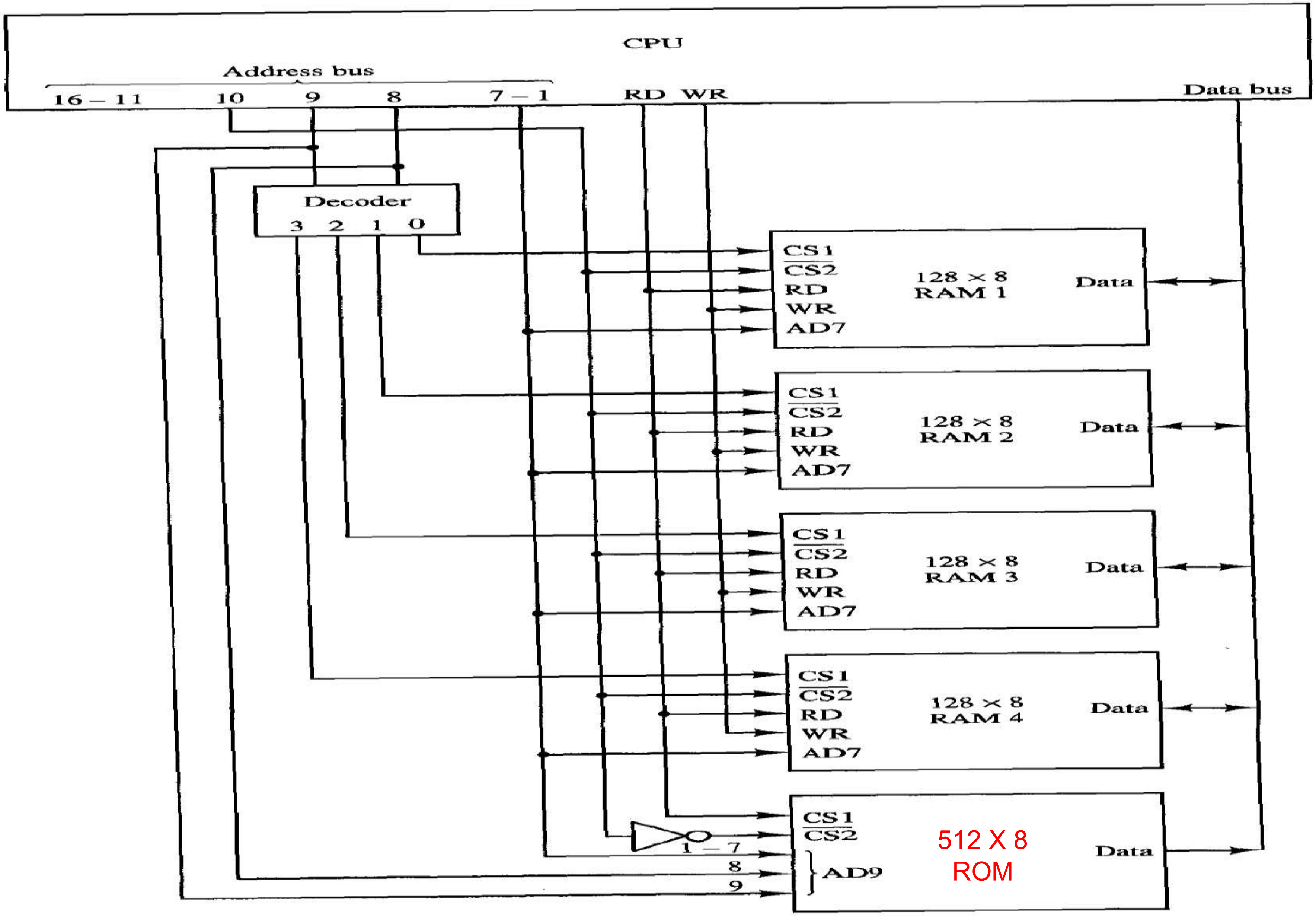


Figure 12-4 Memory connection to the CPU.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Auxiliary Memory

- **HDD, SSD**
- Important characteristics:
  1. **Access mode**
  2. **Access time** – **seek** time + **transfer** time
     - **Seek** time – **find** the memory **location** (**milliseconds**) - ***mechanical***
     - **Transfer** time – time to transfer the content - much **faster** than seek-time - ***electronic***
  3. **Transfer** rate – chars(words) per sec after location found
  4. **Capacity**
  5. **Cost**
- **Organized** in **blocks** or **records** (a specified number of characters/words)
  – **Read/write** - always done on **entire records**

Hard disks

- **Circular metal plate(s) coated with** magnetized material.
- **Read/write heads** on each surface.
- All disks **rotate together non-stop** at high uniform speed
- Bits are stored in **spots** along **concentric circles** called ***tracks***
  - Read/write head
  - Read - a **change** in **magnetic field** produced by a recorded spot passing through a read head
- Tracks are commonly divided into sectors
  - the **minimum quantity** which can be **transferred** is a **sector**

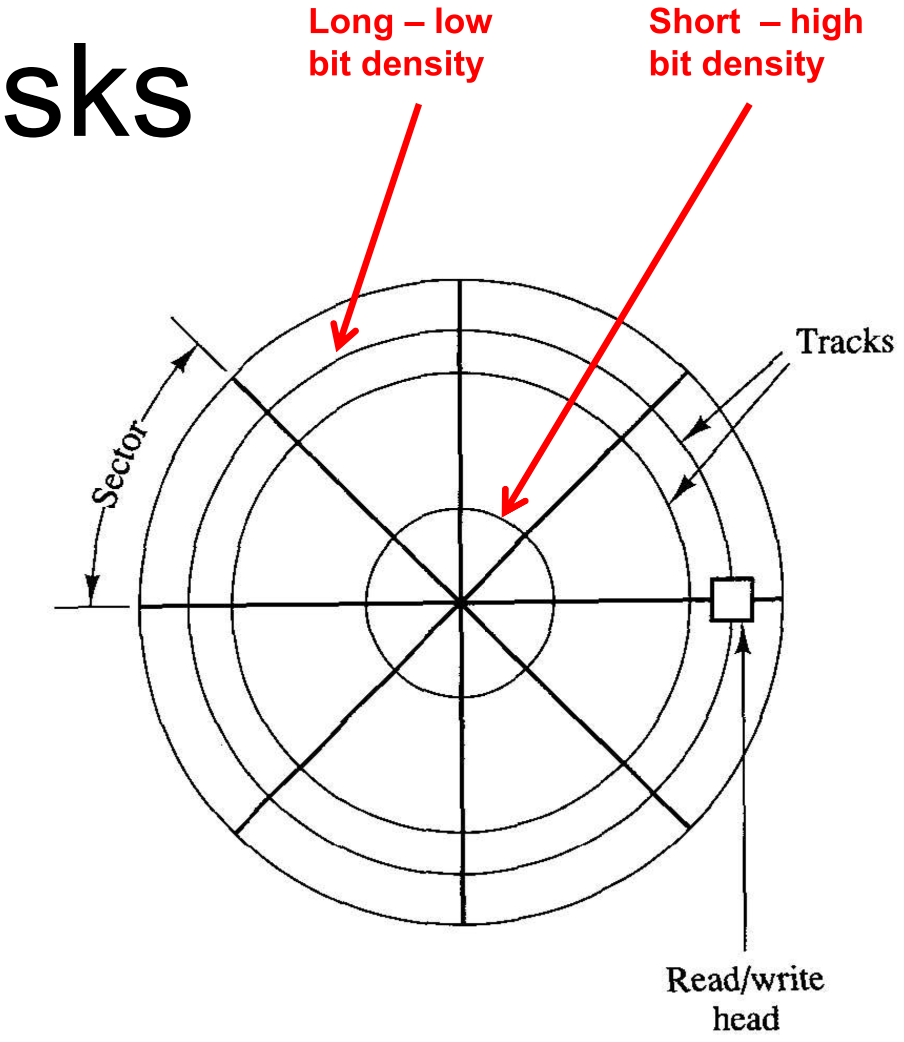


Figure 12-5 Magnetic disk.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Hard disks

- **Timing tracks** used to
  - **synchronize bits** and **recognize sectors**
- Every **sector** has an **address**
  1. disk number
  2. disk surface
  3. sector number
  4. track within the sector
- Seek consists of:
  1. **Position** the read/write **head** in the **specified track**,
  2. **Wait** until the rotating disk **reaches** the specified **sector**
- Information **transfer** is **very fast**
  - once sector beginning is reached
- Disks may have **multiple heads**
  - **simultaneous** transfer of bits from **several tracks**

Associative memory

- **Search** of **items** in a **table** in **memory**
  - **<u>Example:</u>** An assembler - **symbol address table -** extract a symbol's binary equivalent
- Common way to search a table
  - **choose** a **sequence** of **addresses**
  - **read** the **content** of memory at each address,
  - **compare** the **content** with **searched-item** until **match**
- **Number** of **accesses** depends
  - on **item location** and
  - **search efficiency - minimize** the number of **accesses** in random/sequential access memory

Associative memory – cont.

- **Considerably reduces** the **search** time
- **Identifies** data for access **by content** not address
  - **No address** is **given**
- Searches **simultaneously** and in **parallel** by **data association**
- **Write**:
  - The memory **finds** an **empty** unused **location** to **store** the word
- **Read**:
  - the **content** (or **part**) of the word is **specified**
  - Memory **locates** all **words matching** specified **content**
  - **marks** them **for reading**

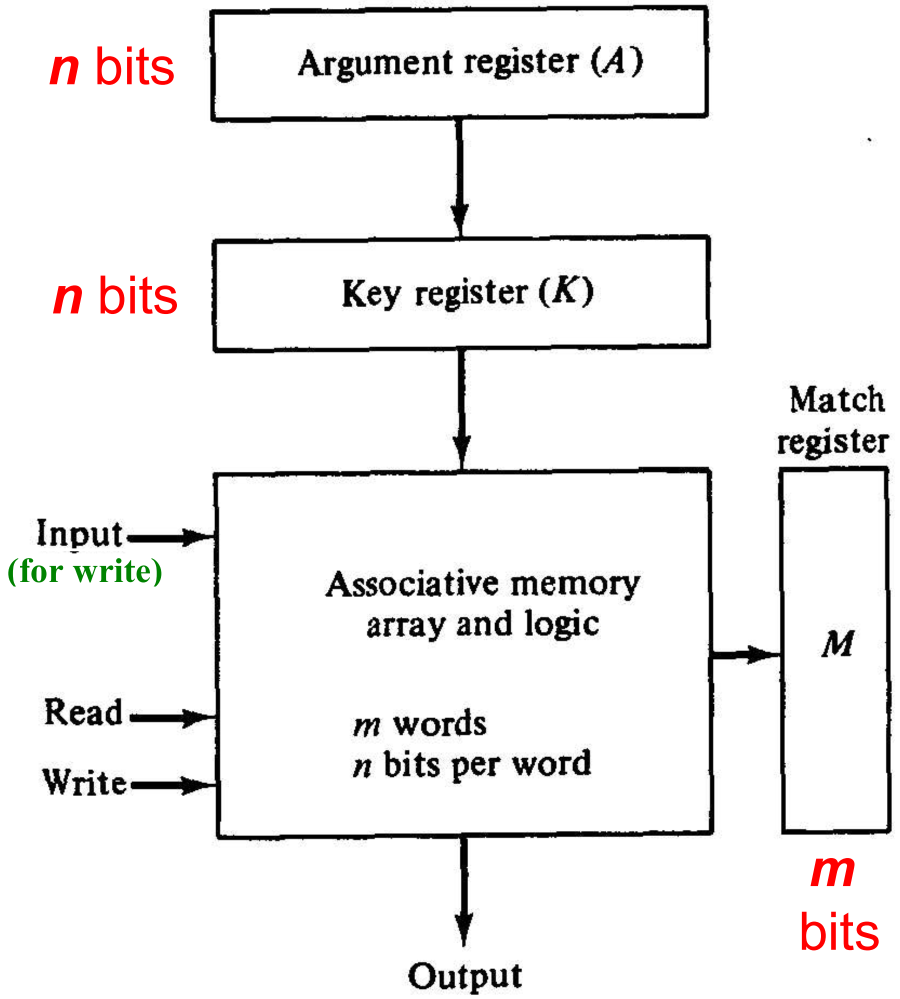
Associative memory – Hardware Organization

- Each **cell** has both
  - **storage** capability
  - **logic circuits** for **matching** its content with an external argument

  ⟶ more **expensive** than RAM

  ⟶ **used** in **applications** where search **time** is very **critical** and must be very short

Figure 12-6 Block diagram of associative memory.

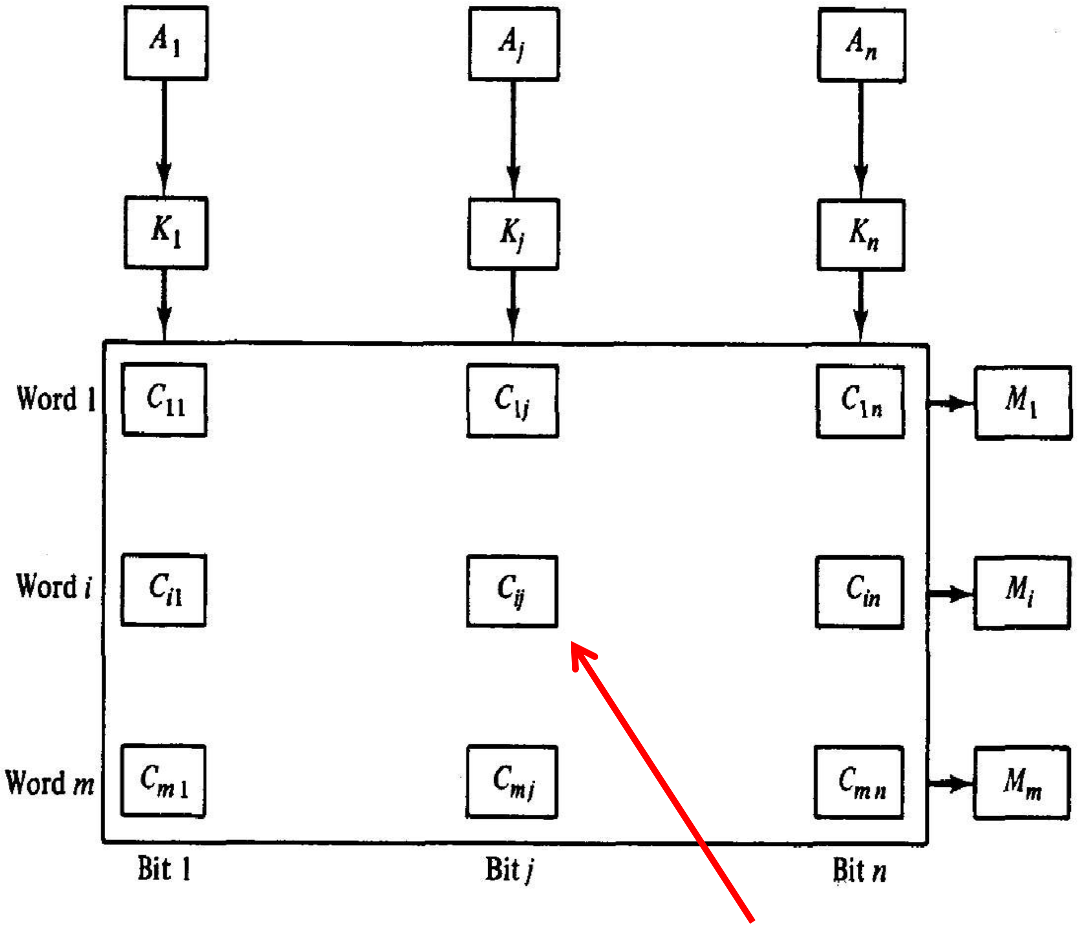


Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Hardware Organization – cont.

- Each memory **word compared** in **parallel** with ***A***
- **Words** that **match *A* set** a corresponding bit in ***M***
- **Reading: sequential access** to memory **words** whose corresponding **bits** in ***M*** have been **set**
- ***K*** provides a **mask** for **choosing fields** or **keys** in ***A***
- **Only bits** in ***A*** that have 1's in ***K*** (same pos) are **compared**
- Example:
  - **A 101 111100**
  - **K 111 000000**
  - **Word 1 100 111100 no match**
  - **Word 2 101 000001 match**
- **Only** the **three leftmost** bits of ***A* are compared** with memory words

Figure 12-7 Associative memory of m word, n cells per word.



j-th bit of the *i*-th word

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Figure 12-7 Associative memory of m word, n cells per word.

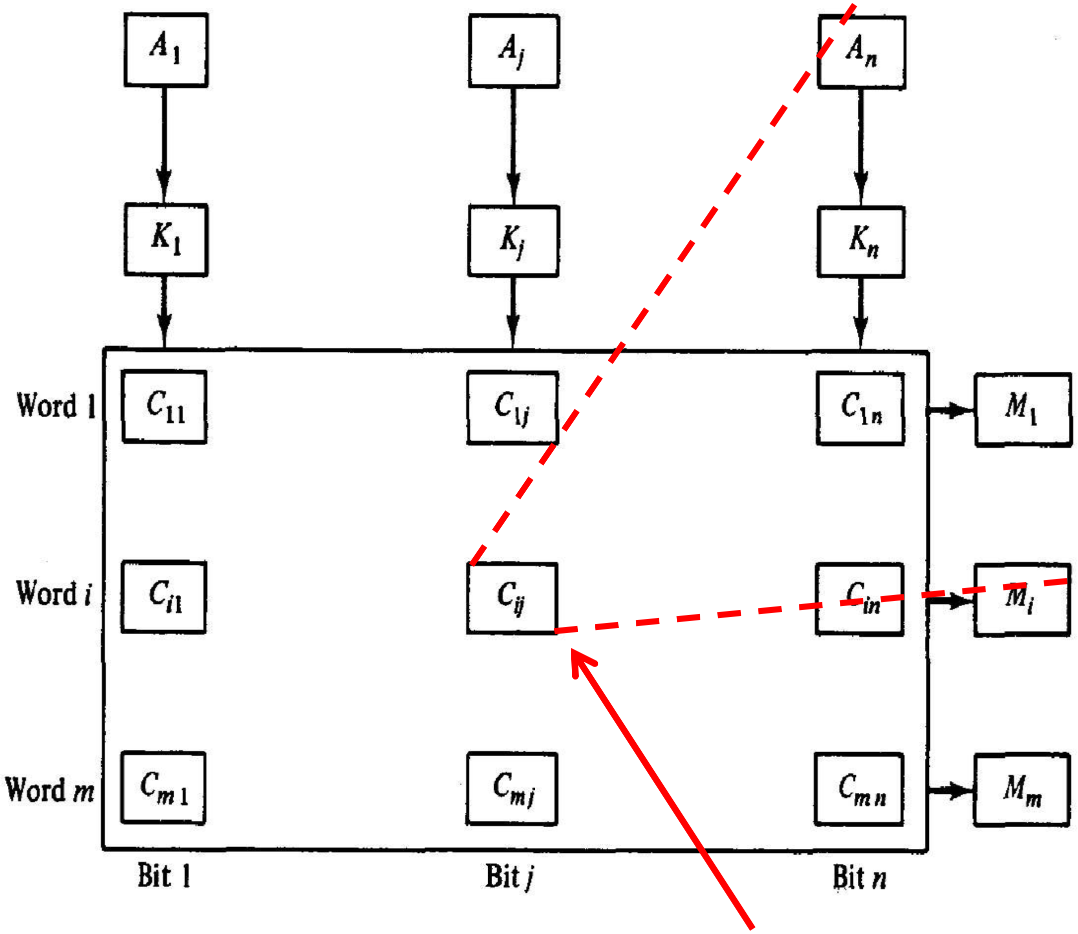
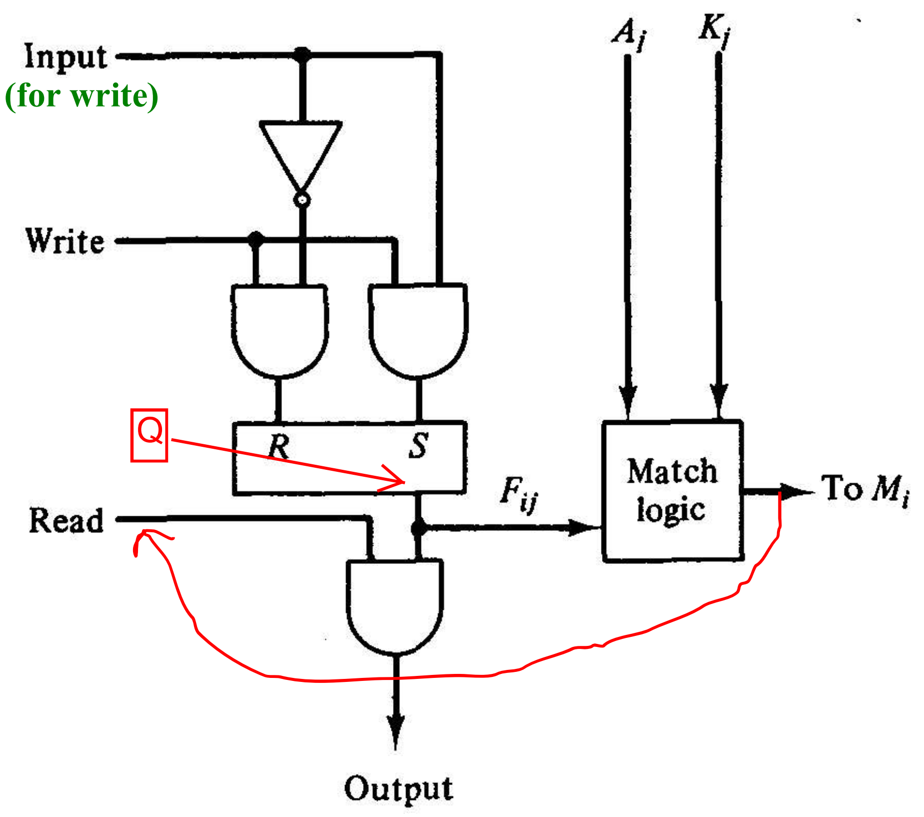


Figure 12-8 One cell of associative memory.

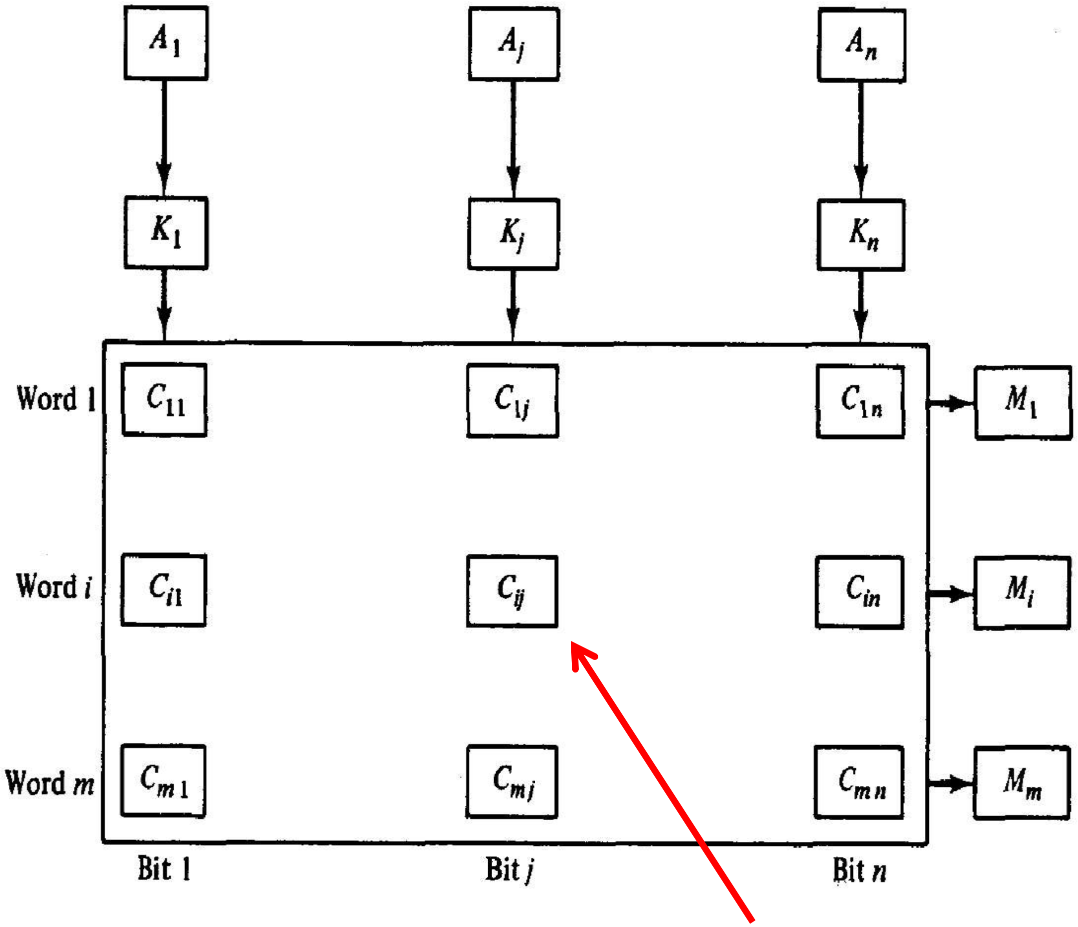


***j*-th bit of the *i*-th word**

Alon Schclar, Tel-Aviv College, 2009

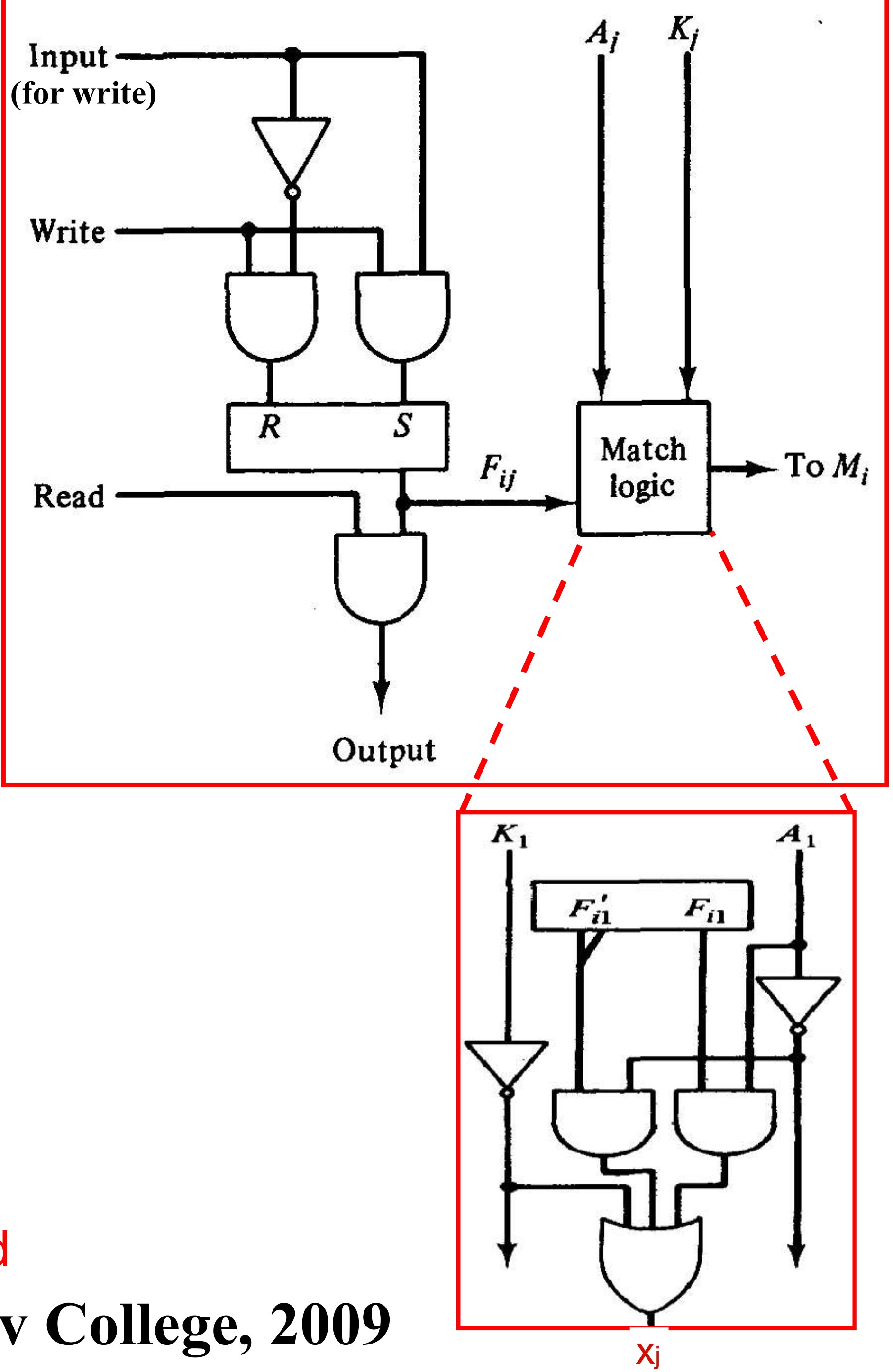
Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Figure 12-7 Associative memory of m word, n cells per word.



j-th bit of the *i*-th word

Figure 12-8 One cell of associative memory.



Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Match Logic

$x_j = A_j F_{ij}(1 \text{ AND } 1) + A_j'F_{ij}'(0 \text{ AND } 0)$

$x_j = A_j \text{ nxor } F_{ij} = A_j F_{ij} + A_j'F_{ij}'$

- Comparison of two bits

$$x_j + K_j' = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

- Combining all results $i = 1, 2, 3, \ldots, m$

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

- When $\boldsymbol{K_j = 1} \rightarrow \boldsymbol{K_j' = 0} \rightarrow \boldsymbol{x_j + 0 = x_j}$
- When $\boldsymbol{K_j = 0} \rightarrow \boldsymbol{K_j' = 1} \rightarrow \boldsymbol{x_j + 1 = 1}$
- $\boldsymbol{(x_j + K'_j) = 1}$ if its pair of bits is not compared → no effect on $\boldsymbol{M_i}$

Read

- If **more** than **one** memory word **matches** ***K***-masked ***A***
  - **Matched words** have **1's** in **corresponding** bit position of ***M***
  - Necessary to **scan** the **bits** of ***M*** one at a time.
  - **Matched** words are **read** in **sequence** by **applying** a **read** signal to word ***i*** if $\boldsymbol{M_i}$ = 1.
- **Usually - associative** memory contains **unique entries**
  - **only one word** may **match** the unmasked argument field
- $\boldsymbol{M_i}$ is connected **directly** to the **read line** in the **same** word **position** (instead of the ***M*** register),
  - **content** of the **matched word** → **output lines**
  - **no** special **read command** signal is **needed**
- Furthermore, **assuming non-zero** word content
  - an **all-zero** output **indicate no match**

Cache memory

- **Locality of reference**
  - References to memory at any **given (short) interval** of **time** tend to be **confined** within a few **localized areas** in memory
    - while the remainder of memory is accessed relatively infrequently.
  - **Code**: **loops** and subroutine calls – **instructions** stored **continuously** in memory → localized references
  - **Data**: Memory references to **arrays** e.g. table-lookup
- **Cache - fast small memory**
  - Stores **active** portions of the **program** and **data**
  - ***Placed between the CPU and main memory***
  - **access time 5-10 times faster than** memory access time
  - **Reduces average memory access** time
    - **many** memory **requests** will be **found** in the **cache** memory
→ **average** memory **access** time will **approach cache access** time
→ **reduces** the **total execution** time
  - **fastest component in memory** hierarchy - approaches the speed of CPU components

Cache operation

- CPU needs to access memory for a word **X**
  - the **cache** is **examined**.
  - If X is **found** in **cache**, it is **read** from it
  - **Otherwise**, the **main memory** is **accessed**
    - **A continuous block of words** containing **X** is fetched from main memory to cache memory.
      - **Block size** 1-16 words (1 = just **X**)
      - **future near references** can **find** the required **words** in the **cache**
  - Every cache word has a duplicate in main mem

Hit ratio

- Measures performance of cache memory
- **Hit** – When CPU finds X in cache
- **Miss** – otherwise
- **Hit ratio = hits / (hits+misses)**
  - hits+misses = total CPU references to memory
- Measured experimentally by **benchmark programs**
  - **executed** during a given **interval** of time.
- Hit ratios ≥ **0.9** reported
  - **Verifies locality of reference** validity

Example: hit ratio 0.9 (9 from 10 reads in found in cache); Mem read time 100ns; Cache read time 10ns

Average read time: (9*10 + 1*(10+100))/10 = 20ns ~ more than 5 times faster than w/o using cache mem

Mapping

- **Transformation** of data **from main** memory **to cache**
- Three types of mapping
  1. **Associative**
  2. **Direct**
  3. **Set-associative**
- **<u>Example</u>**:
  - Main memory stores **2^{15}=32K words of 12 bits each**
  - **Cache stores 512 words**
- **CPU communicates** with **both** memories
  - **First** sends a 15-bit address to **cache**
    - **Hit**: the CPU accepts the 12-bit data from cache.
    - **Miss**:
      - CPU reads the word from main memory
      - the word is then transferred to cache.

Example

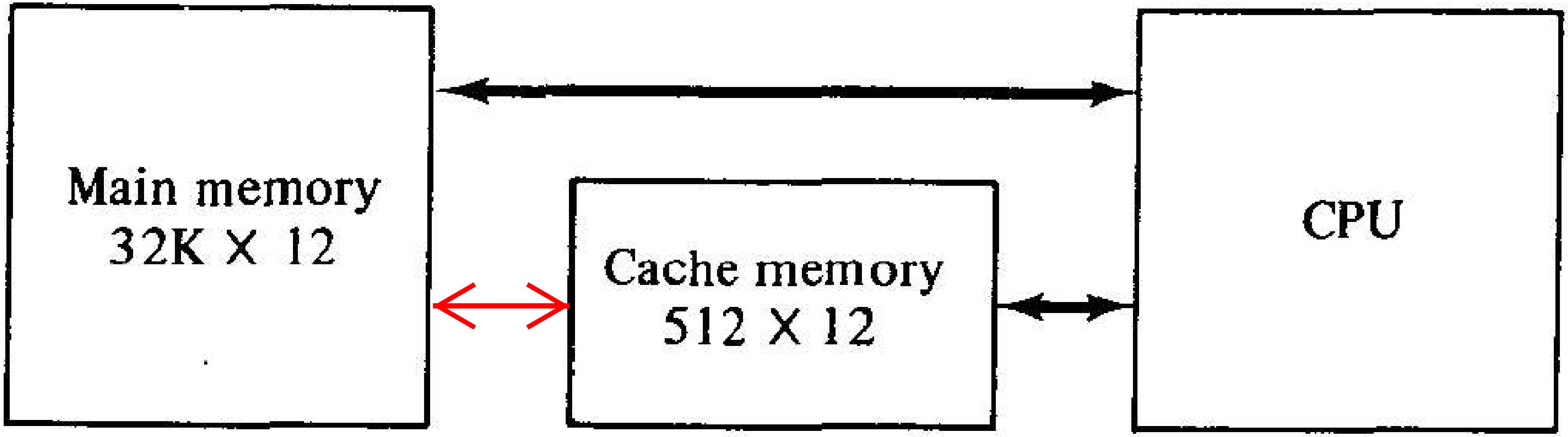


Figure 12-10 Example of cache memory.

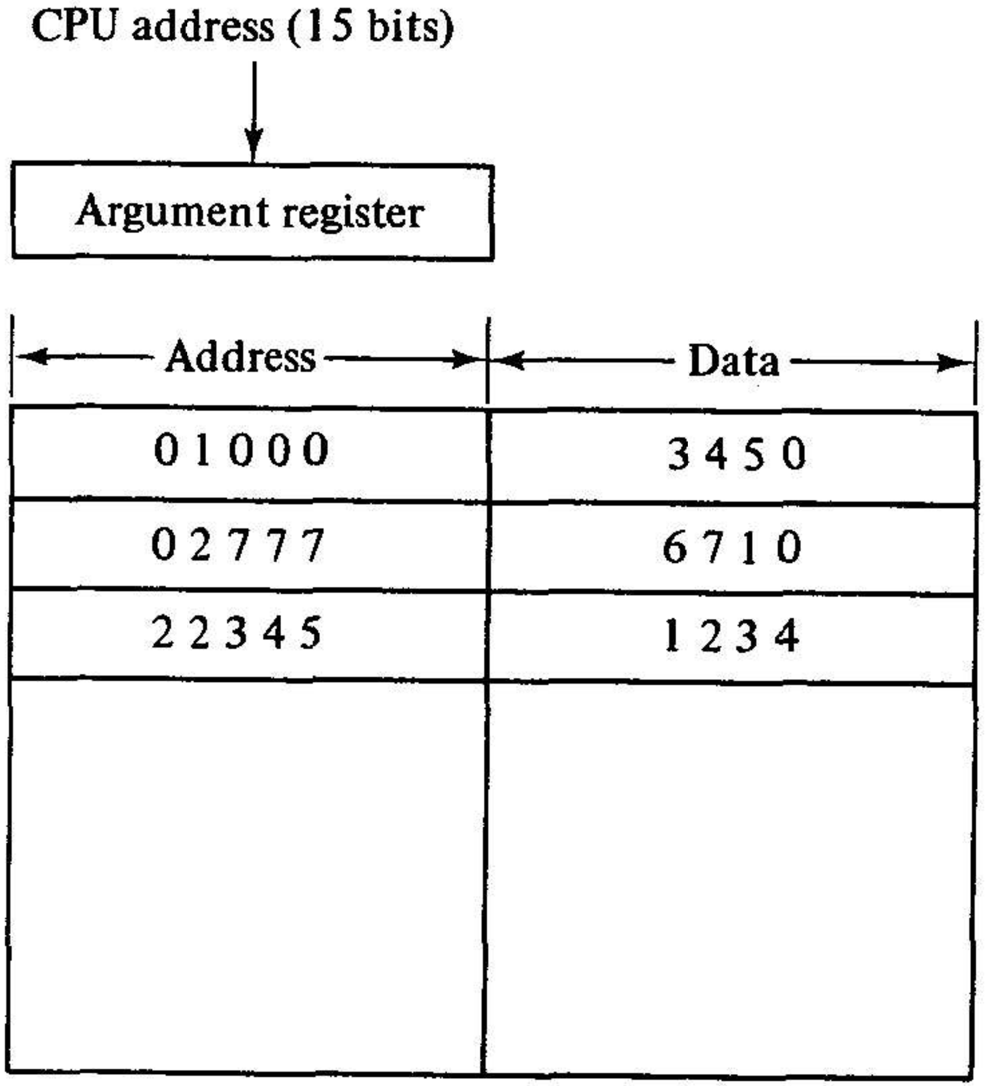
Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Associative mapping

- The **fastest** and **most flexible** cache organization
- Uses an **associative memory** (AM)
- **Stores both address** and **content** of memory word
  - Any location in cache can store any word from main mem
- A CPU **address placed** in **argument register (A)**
- **AM** is **searched** for a **matching address**
  - **Found**: corresponding 12-bit data is fetched to CPU
  - **No match:**
    - the **main** memory **accessed**
    - The **address-data pair transferred** to **AM** cache
      - **If** cache is **full**, an address-data **pair** is **displaced** – replacement algorithm – **FIFO**, **LRU**, etc

Associative mapping example

Figure 12-11 Associative mapping cache (all numbers in octal).



Address	Data
01000	3450
02777	6710
22345	1234

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

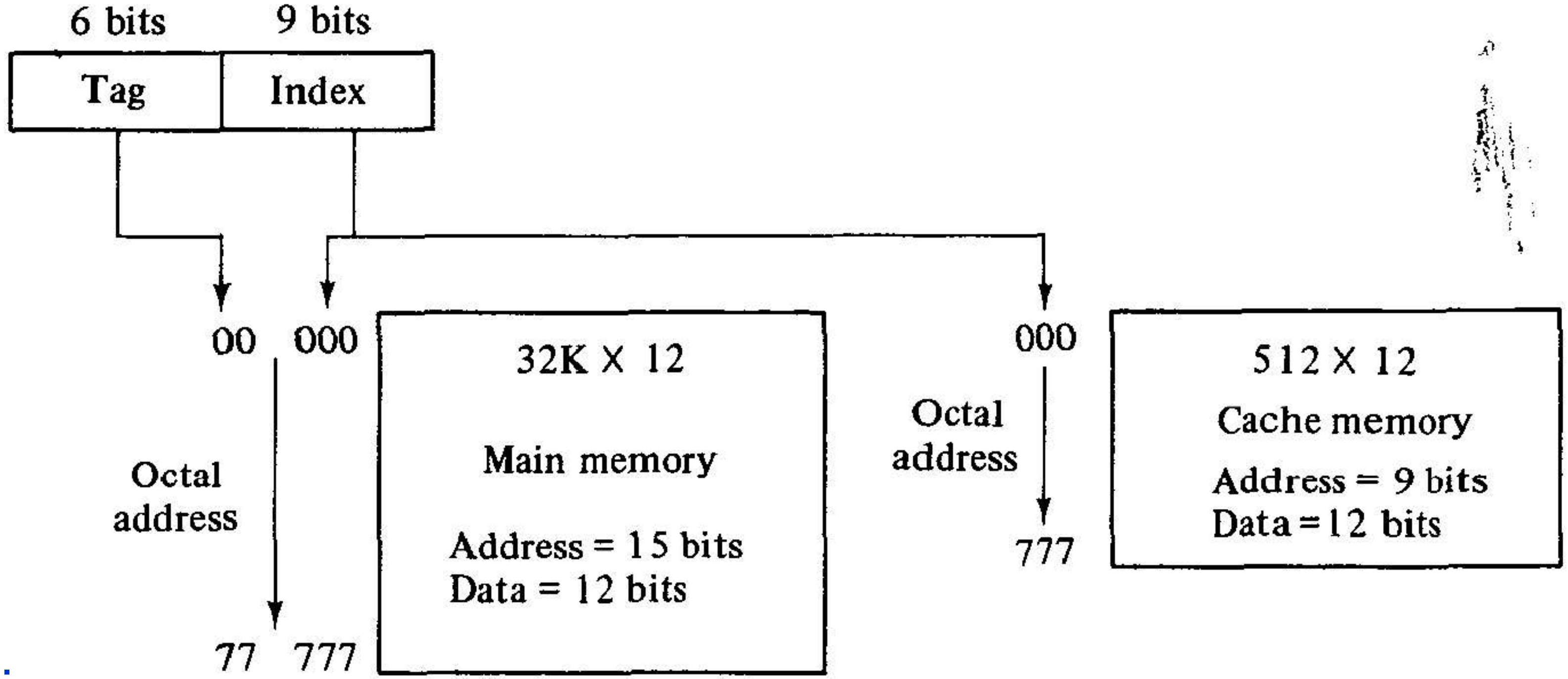
Direct mapping - (DM)

- **AMs** are **expensive** compared to RAM (**added logic**)
- DM uses RAM
- Usually: **cache** contains **2^k words** and **main** mem **2^n** (**k<<n**)
- The n-bit memory address is divided into two fields:
  - **k bits** for the ***index field*** (like last) and LSB
  - **(n – k) bits** for the ***tag field*** (first name) MSB
- DM uses
  - the **n-bit** address to **access** the **main** memory and
  - the k-bit **index** to **access** the **cache**.
- Each **word** in **cache** consists of the **data** word and its associated **tag**
- **CPU** memory request
  - the **index** field is **used** for the address to **access** the **cache**.
  - **tag** field of the **CPU address** is **compared** with the **tag** in **cache** word
    - **If tags match → hit**
    - **Otherwise a miss → required word** is **read** from **main** memory
      - **store** word in the **cache** (**new tag and content**)
- **Disadvantage**: Frequent **access** to words **same index** but **different tags**
  - possibility is **minimized** since words are **relatively far** apart (multiples of **2^k**)

Direct Mapping - example

- CPU address of 15 bits
  - **Index field**:
    - Use as the address in the cache memory
    - The 9 least significant bits
  - **Tag field**: remaining 6 bits

Figure 12-12 Addressing relationships between main and cache memories.

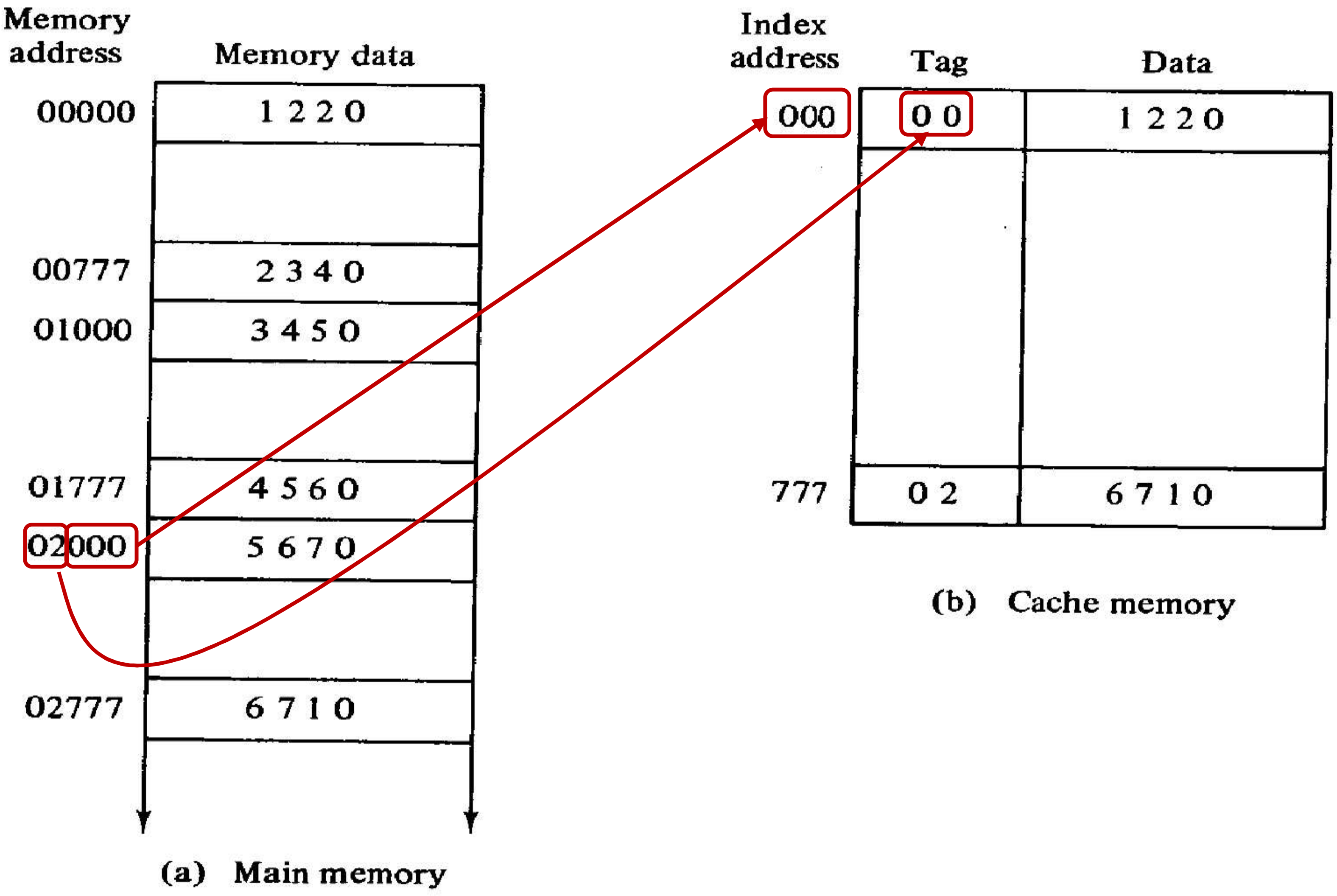


Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Direct Mapping - example

- **MEM[0]=1220** is currently stored in cache
  - index = 000, tag = **00**, data = 1220
- **CPU** now wants to access the word at address 02000
  - index = 000, tag = **02**
- The two tags are then compared.
  - **00 ≠ 02 → no match**
    - main memory is accessed
    - data word 5670 is transferred to the CPU.
    - Cache[000].**tag** ← **02**
    - Cache[000].**data** ← **5670**

Direct Mapping - example



Memory address	Memory data
00000	1 2 2 0
00777	2 3 4 0
01000	3 4 5 0
01777	4 5 6 0
02000	5 6 7 0
02777	6 7 1 0

(a) Main memory

Index address	Tag	Data
000	0 0	1 2 2 0
777	0 2	6 7 1 0

(b) Cache memory

Figure 12-13 Direct mapping cache organization.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

DM with block-size > 1

- So far block size = 1 word.
- **Block** can contain ***2^h* words**
- The index field divided into two parts:
  - the ***block number*** field – ***k-h*** bits
  - ***Word number*** within block field – ***h*** bits
- **Example**:
  - 512-word cache, 8 words per block, 64 blocks (64 X 8 = 512)
  - **Block number** is specified by **6-bit** and
  - the **word number** within the block is specified by a **3-bit** field.
- **Tag field common to all eight words of same block**
- **Miss $x_4x_3x_2x_1x_0$ - *octal*:** x_4x_3-tag ; x_2x_1-block# ; x_0-word#
  - entire block (addresses $x_4x_3x_2x_1$*0-7*) fetched from main to cache
  - **Although** takes **extra time**, the hit ratio will most **likely improve** with a **larger** block size because of the **sequential nature** of computer programs

Direct mapping with 8-word block

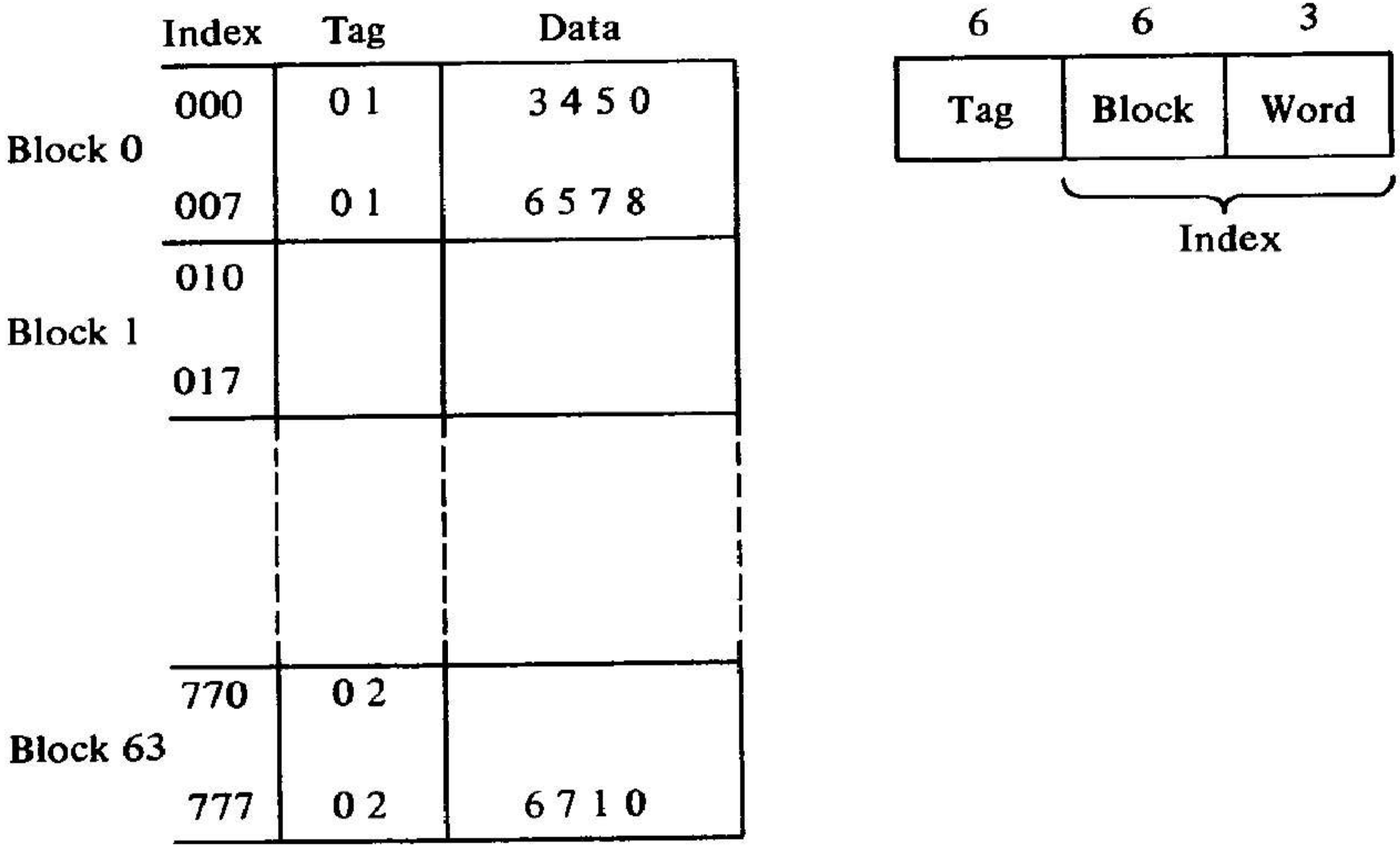


Figure 12-14 Direct mapping cache with block size of 8 words.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Set-Associative Mapping (N-way)

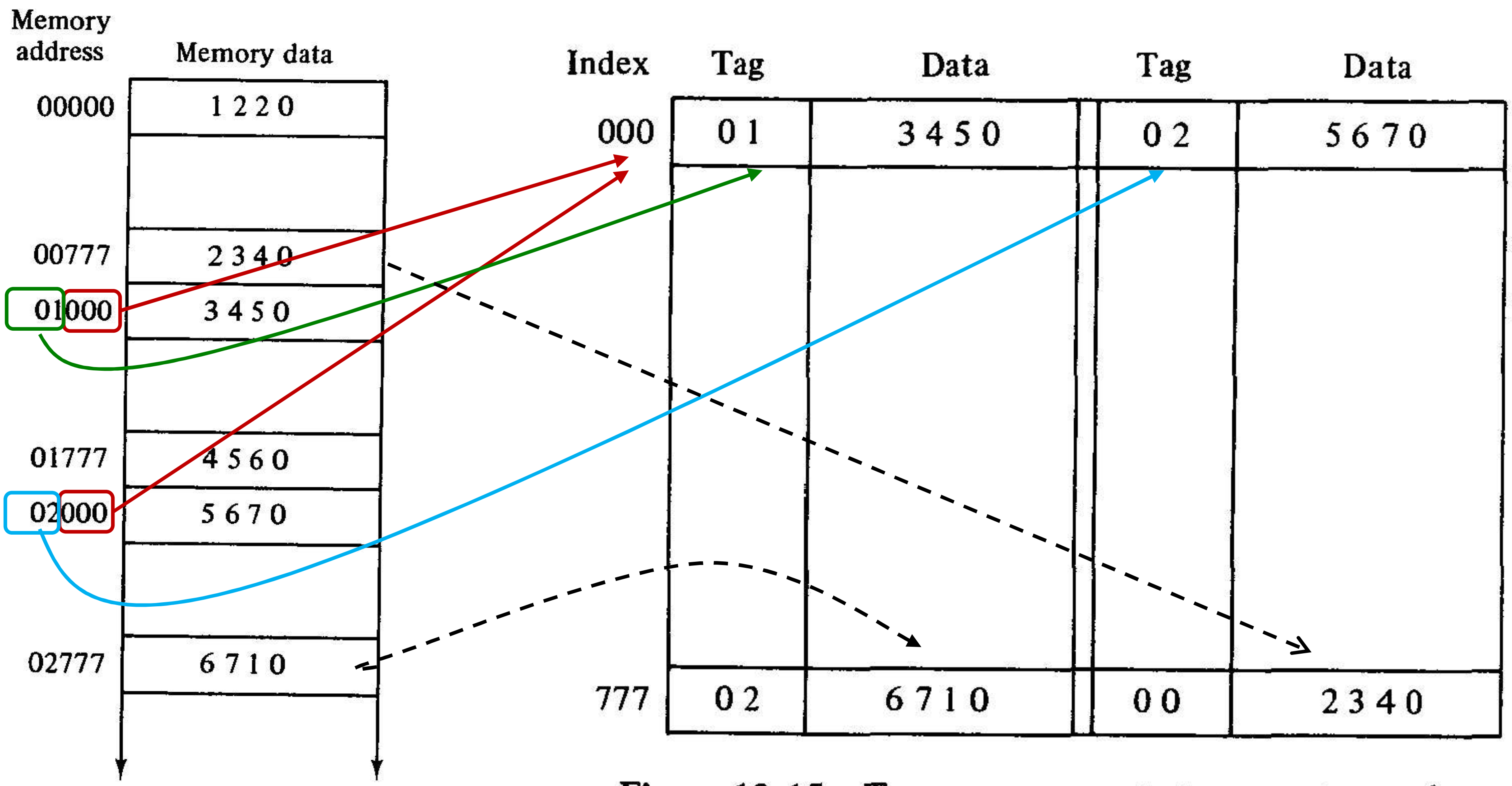
- **Extension** of **DM**
- **cache word** (***entry***) ≥ **main** memory **word**
  - **Each cache index** address refers to ***N*** **memory words** and their associated tags.
  - Direct map - N=1
  - All **entry words** share the **same *index*** **but** have **different *tags***
- **Example: N=2 & Index address of 9 bits**

  512 entries(words)
  - Cache accommodates **1024 memory words**:
    - **cache word = 2 main memory words**.
  - Each **tag** requires **6 bits** and each **data** word has **12 bits**
  - **Index entry** length is **2(6 + 12) = 36 bits**
  - **Total size** of cache memory: 512 X 36 bits

2-way associative mapping example

- MEM[01000] and MEM[02000] stored in cache[000]
- MEM[02777] and MEM[00777] stored in cache[777]
- CPU memory request
  - **index** value of address **used** to **access** the **cache**
  - **tag** field of **address compared** with **both** cache **entry** tags
    - comparison logic via **associative search** of tags (similar to associative memory → name "set-associative”)
- The hit ratio improves as the set size increases
  - more words with same index, different tags
  - However,
    - an **increase** in **number** of **bits** in words of cache and
    - more **complex** comparison **logic**

2-way set-associative cache



(a) Main memory

Figure 12-15 Two-way set-associative mapping cache.

Taken from: **M. Mano/Computer Design and Architecture 3rd Ed.**

Replacement algorithms (RA)

- When **miss** occurs **and** the set is **full**
  - it is **necessary** to **replace one** of the **tag-data** items with a new value
- The most common RA’s :
  - **Random** – randomly choose one entry
  - **FIFO** - item that has **been** in the set the **longest**
  - → **LRU** - item **least recently used** by the CPU
- **FIFO** and **LRU** can be implemented by adding a **few extra bits** in each entry

Writing to Cache

- ***Write-through method***
  - **Update** main memory with **every memory write** operation
  - **Update cache** memory in **parallel** if it contains the updated word
  - ***Advantage***: **main** memory always in **sync** with **cache**
    - **Important** in systems with **DMA** transfers.
    - Data residing in **main memory** are **valid** at **all times**
    - **DMA write delivers valid** and **most recent** data to I/O device
- ***Write-back method***
  - **Only** the **cache** location is **updated** during write
  - The **word** is marked by a **flag** – "***need to write*** to main mem"
  - Word is **copied** into main memory **only when displaced** from cache
  - Why ?
    - **Faster** if program **updates** word **several** times **when** it is **in cache**
      - results show # of **memory writes is *10%-30%*** of the total memory **refs**
    - **Copy** in **main** memory **can be out of date**
      - **requests** for word are **filled from** the **cache**

Cache Initialization

- Cache initialized **when**
  - **power** is turned **on** or
  - main memory is **loaded** with a **complete** set of **programs** from **aux.** mem.
- **After initialization - cache** is considered **empty**
  - in **effect** it contains some **non valid** (**garbage**) data
- Include a **valid bit** in **each** cache **word**
  - **indicate** whether or not the **word** contains **valid data**
  - **Cache initialization: clear all valid bits to *0***
  - set to ***1*** after the **first time** word is **loaded** from **main** mem
  - and **stays** set **until** next **initialization**
- A **cache word** is **replaced** by another only if
  - the **valid bit** is set to ***1*** and
  - a **mismatch of tags** occurs.
- IF ***valid* bit = 0, cache** word is **replaced regardless** of **tags**
  - **forces misses until** cache **fills** with **valid** data