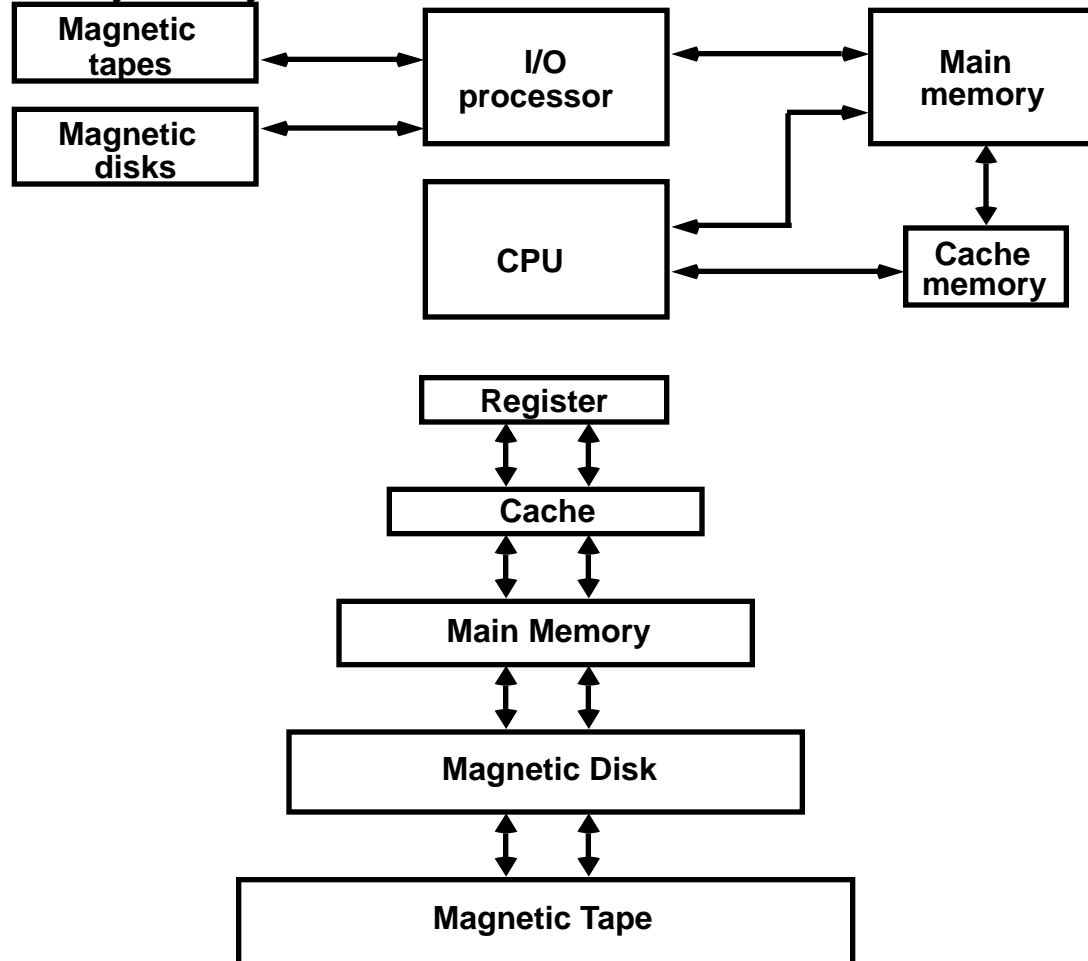# MEMORY ORGANIZATION

- **Memory Hierarchy**

- **Main Memory**

- **Auxiliary Memory**

- **Associative Memory**

- **Cache Memory**

- **Virtual Memory**

- **Memory Management Hardware**

MEMORY HIERARCHY

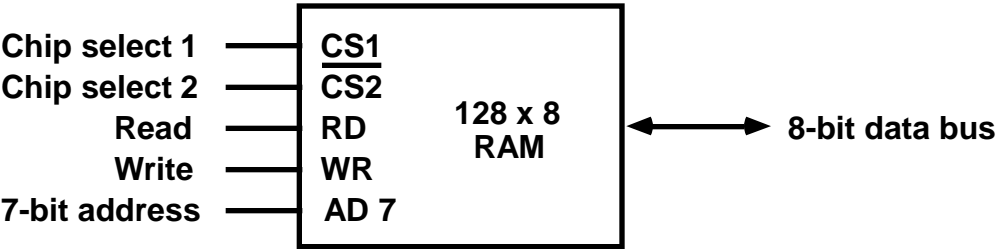**Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system**

Auxiliary memory

MAIN MEMORY

RAM and ROM Chips

Typical RAM chip

Chip select 1 ————	CS1	
Chip select 2 ————	CS2	
Read ————	RD	128 x 8
Write ————	WR	RAM
7-bit address ————	AD 7	

8-bit data bus

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

Typical ROM chip

Chip select 1 ————	CS1	
Chip select 2 ————	CS2	
		512 x 8
		ROM
9-bit address ————	AD 9	

8-bit data bus
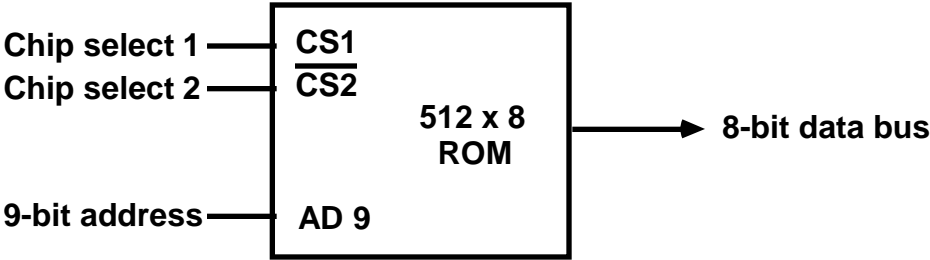
MEMORY ADDRESS MAP

Address space assignment to each memory chip

Example: 512 bytes RAM and 512 bytes ROM

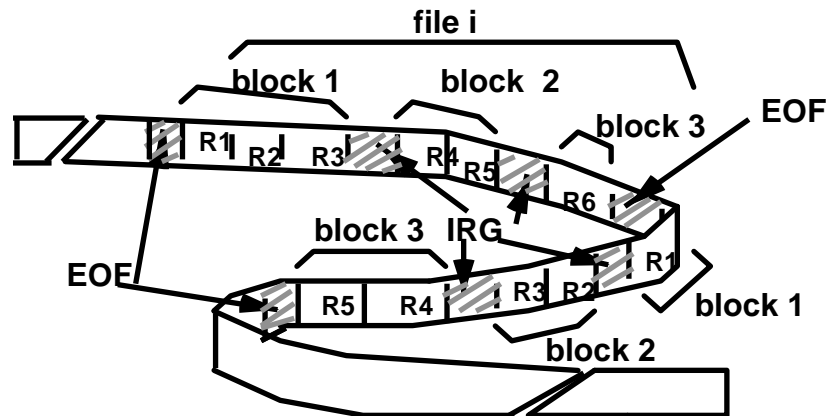| Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

Memory Connection to CPU

> **- RAM and ROM chips are connected to a CPU through the data and address buses**

> **- The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs**

CONNECTION OF MEMORY TO CPU
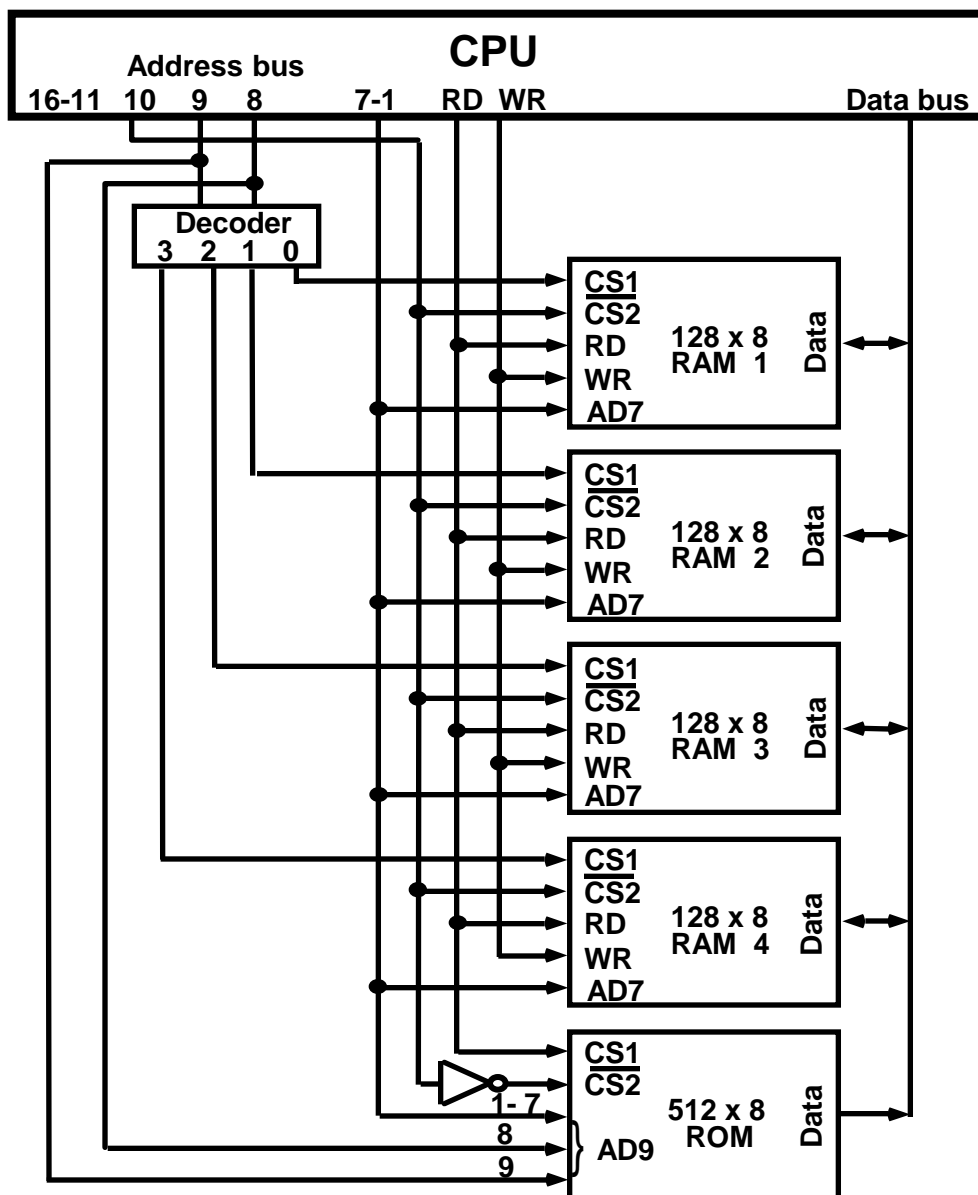
AUXILIARY MEMORY

Information Organization on Magnetic Tapes

file i

block 1   block 2

block 3   EOF

R1 R2 R3 R4 R5 R6

EOF   block 3 IRG   block 1

R5 R4 R3 R2 R1   block 2

Organization of Disk Hardware
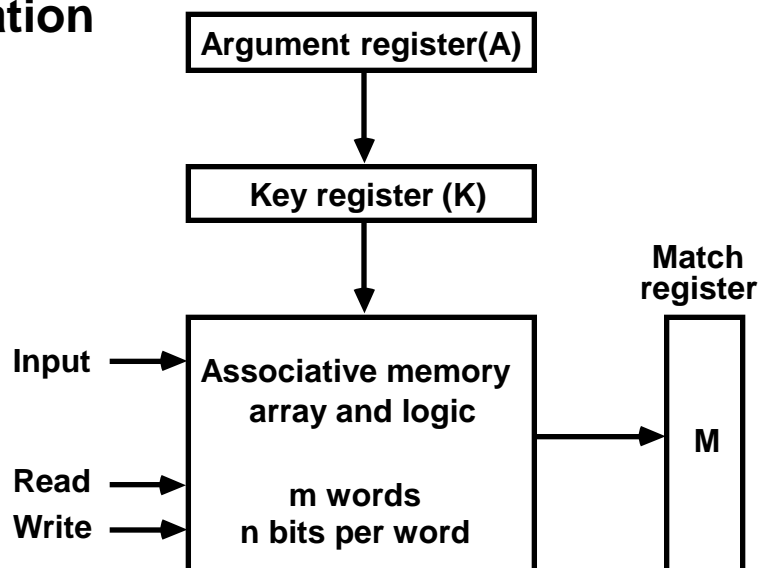### Moving Head Disk   Fixed Head Disk

Track
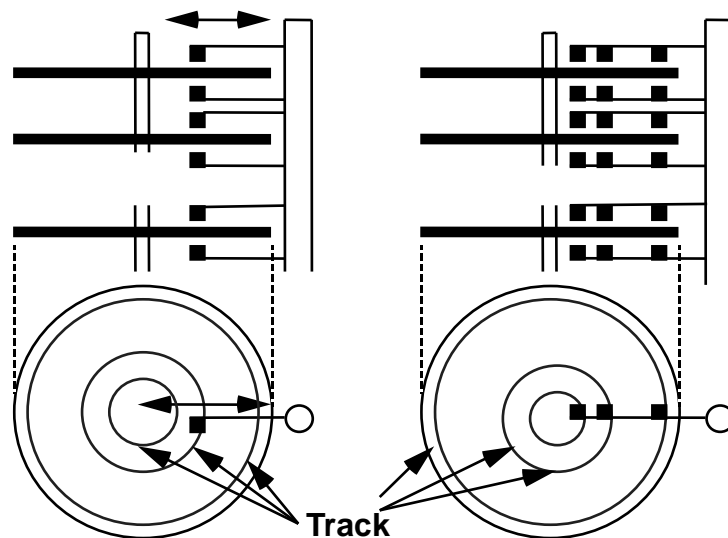
ASSOCIATIVE MEMORY

- Accessed by the content of the data rather than by an address
**- Also called Content Addressable Memory (CAM)**

Hardware Organization

```
                    ┌─────────────────────┐
                    │ Argument register(A) │
                    └─────────────────────┘
                              │
                              ▼
                    ┌─────────────────────┐
                    │   Key register (K)   │
                    └─────────────────────┘
                              │              Match
                              │              register
                              ▼          ┌──────────┐
 Input ──────▶ ┌─────────────────────┐   │          │
               │ Associative memory  │   │          │
               │   array and logic   │──▶│    M     │
 Read  ──────▶ │                     │   │          │
 Write ──────▶ │      m words        │   │          │
               │    n bits per word  │   └──────────┘
               └─────────────────────┘
```

- Compare each word in CAM in parallel with the
     content of A(Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for CAM Word(i) for M(i) = 1
- K(Key Register) provides a mask for choosing a
     particular field or key in the argument in A
     (only those bits in the argument that have 1's in
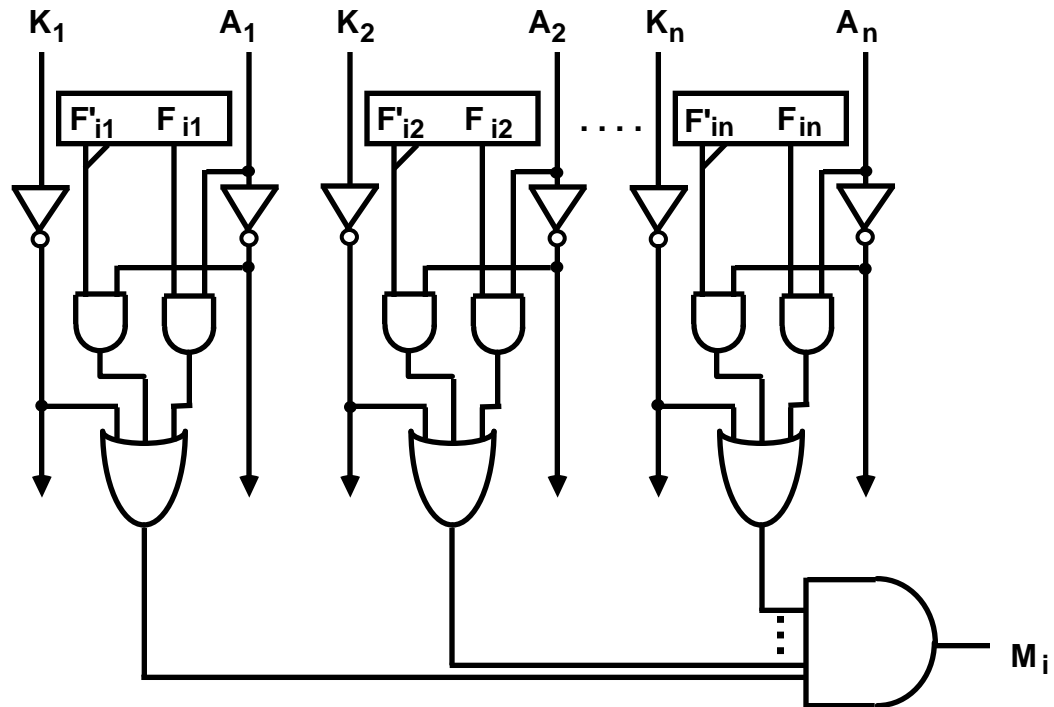   **their corresponding position of K are compared)**

ORGANIZATION OF CAM
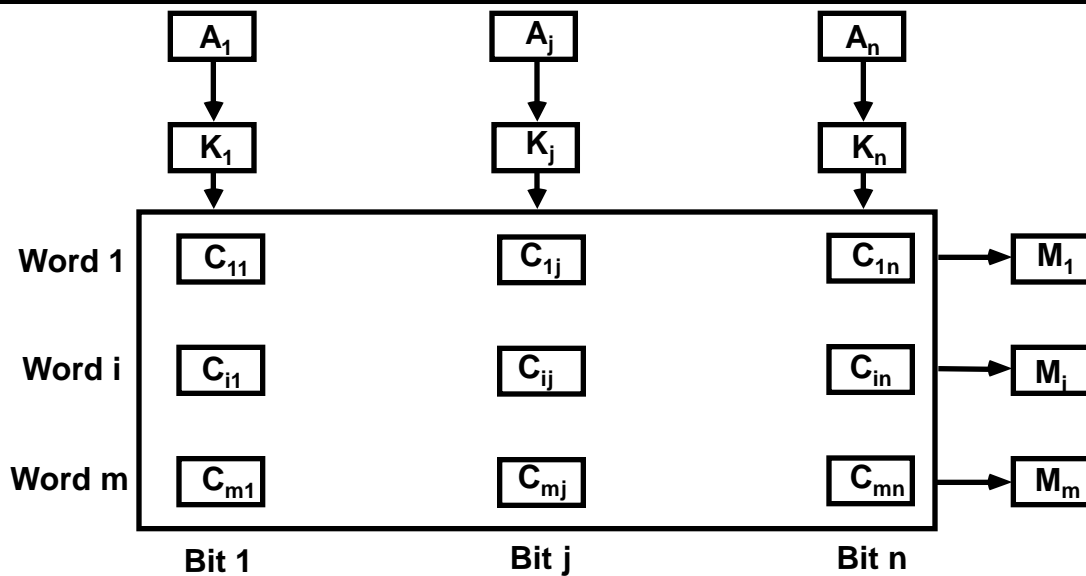
Internal organization of a typical cell C_{ij}
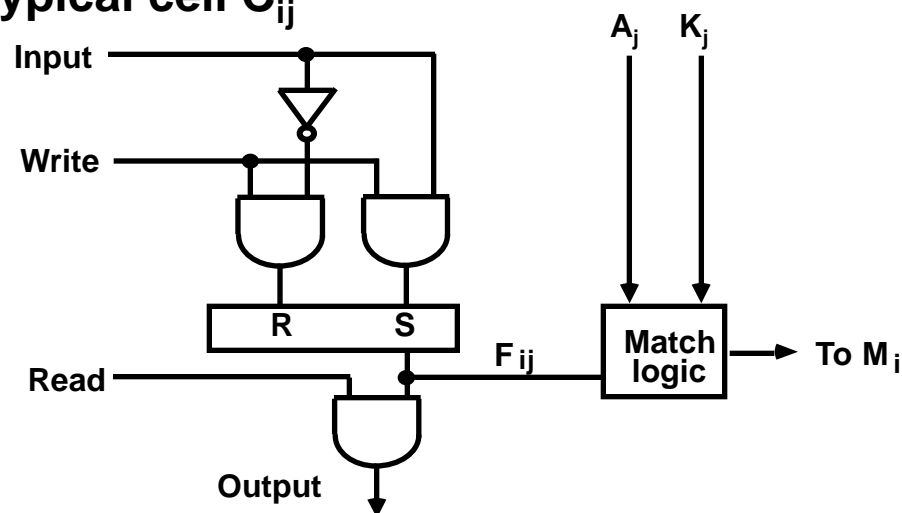
MATCH LOGIC

CACHE MEMORY

Locality of Reference

- **The references to memory at any given time interval tend to be confined within a localized areas**
- **This area contains a set of information and the membership changes gradually as time goes by**
- ***Temporal Locality***
  The information which will be used in near future is likely to be in use already (e.g. Reuse of information in loops)
- ***Spatial Locality***
  If a word is accessed, adjacent(near) words are likely accessed soon (e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

Cache

- **The property of Locality of Reference makes the Cache memory systems work**
- **Cache is a fast small capacity memory that should hold those information which are most likely to be accessed**
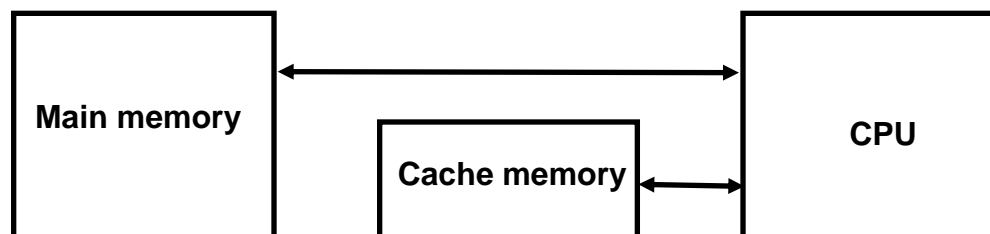
PERFORMANCE OF CACHE

Memory Access

All the memory accesses are directed first to Cache
If the word is in Cache; Access cache to provide it to CPU
If the word is not in Cache; Bring a block (or a line) including
that word to replace a block now in Cache

- How can we know if the word that is required
  is there ?
- If a new block is to replace one of the old blocks,
  which one should we choose ?

Performance of Cache Memory System

Hit Ratio - % of memory accesses satisfied by Cache memory system
T_e: Effective memory access time in Cache memory system
T_c: Cache access time
T_m: Main memory access time

$$T_e = T_c + (1 - h) T_m$$

Example: T_c = 0.4 μs, T_m = 1.2μs, h = 85%
T_e = 0.4 + (1 - 0.85) * 1.2 = 0.58μs
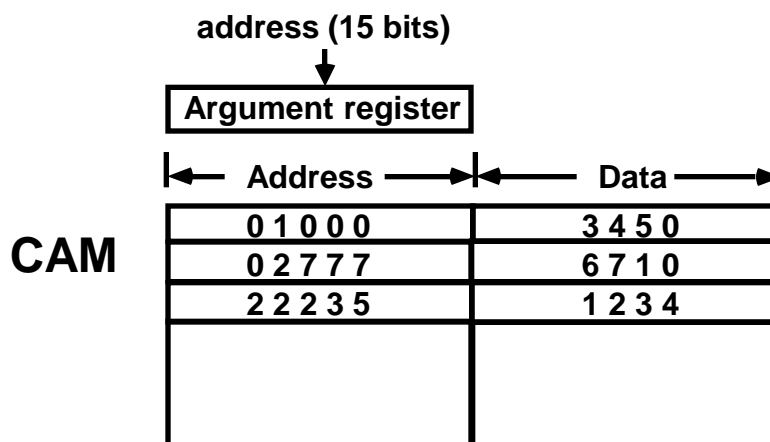
MEMORY AND CACHE MAPPING - ASSOCIATIVE MAPPLING -

Mapping Function

Specification of correspondence between main memory blocks and cache blocks

> **Associative mapping**
> **Direct mapping**
> **Set-associative mapping**

Associative Mapping

- Any block location in Cache can store any block in memory
 ⇒ **Most flexible**
- Mapping Table is implemented in an associative memory
 ⇒ **Very Expensive**
- Mapping Table
 Stores both address and the content of the memory word

address (15 bits)

Argument register

←——— Address ———→	←——— Data ———→
0 1 0 0 0	3 4 5 0
0 2 7 7 7	6 7 1 0
2 2 2 3 5	1 2 3 4

CAM

MEMORY AND CACHE MAPPING - DIRECT MAPPING

- **Each memory block has only one place to load in Cache**
- **Mapping Table is made of RAM instead of CAM**
- **n-bit memory address consists of 2 parts; k bits of Index field and**
   n-k bits of Tag field
- **n-bit addresses are used to access main memory**
   and k-bit Index is used to access the Cache

Addressing Relationships

Tag(6)	Index(9)

00 000

32K x 12

Main memory
Address = 15 bits
Data = 12 bits

77 777

000

512 x 12
Cache memory
Address = 9 bits
Data = 12 bits

777
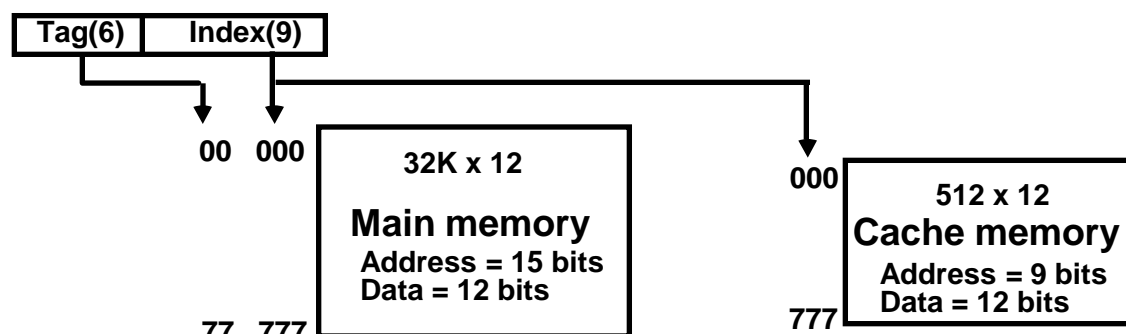
Direct Mapping Cache Organization

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

Cache memory

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

DIRECT MAPPING

Operation

  - CPU generates a memory request with (TAG;INDEX)
  - Access Cache using INDEX ; (tag; data)
      Compare TAG and tag
  - If matches $\Rightarrow$ Hit
      Provide Cache[INDEX](data) to CPU
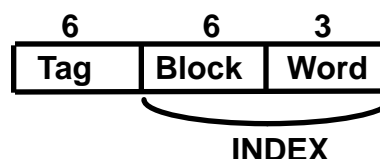 **- If not match $\Rightarrow$ Miss**
      M[tag;INDEX] $\leftarrow$ Cache[INDEX](data)
      Cache[INDEX] $\leftarrow$ (TAG;M[TAG; INDEX])
      CPU $\leftarrow$ Cache[INDEX](data)

Direct Mapping with block size of 8 words

Index	tag	data
000	**0 1**	**3 4 5 0**
007	**0 1**	**6 5 7 8**
010		
017		
770	**0 2**	
777	**0 2**	**6 7 1 0**

Block 0 → rows 000, 007
Block 1 → rows 010, 017
Block 63 → rows 770, 777

6	6	3
Tag	Block	Word

INDEX (Block + Word)

SET ASSOCIATIVE MAPPING

- Each memory block has a set of locations in the Cache to load

Set Associative Mapping Cache with set size of two

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

Operation
- **CPU generates a memory address(TAG; INDEX)**
- **Access Cache with INDEX,   (Cache word = (tag 0, data 0); (tag 1, data 1))**
- **Compare TAG and tag 0 and then tag 1**
- **If tag i = TAG $\Rightarrow$ Hit, CPU $\leftarrow$ data i**
- **If tag i $\neq$ TAG $\Rightarrow$ Miss,**
  Replace either (tag 0, data 0) or (tag 1, data 1),
  Assume (tag 0, data 0) is selected for replacement,
  (Why (tag 0, data 0) instead of (tag 1, data 1) ?)
  M[tag 0, INDEX] $\leftarrow$ Cache[INDEX](data 0)
  Cache[INDEX](tag 0, data 0) $\leftarrow$ (TAG, M[TAG,INDEX]),
  CPU $\leftarrow$ Cache[INDEX](data 0)

BLOCK REPLACEMENT POLICY

Many different block replacement policies are available

LRU(Least Recently Used) is most easy to implement

Cache word = (tag 0, data 0, *U0*);(tag 1, data 1, *U1*), Ui = 0 or 1(binary)

Implementation of LRU in the Set Associative Mapping with set size = 2

Initially all U0 = U1 = 1
When Hit to (tag 0, data 0, U0), U1 ← 1(least recently used)
(When Hit to (tag 1, data 1, U1), U0 ← 1(least recently used))
When Miss, find the least recently used one(Ui=1)
  If U0 = 1, and U1 = 0, then replace (tag 0, data 0)
    M[tag 0, INDEX] ← Cache[INDEX](data 0)
    Cache[INDEX](tag 0, data 0, U0) ← (TAG,M[TAG,INDEX], 0); U1 ← 1
  If U0 = 0, and U1 = 1, then replace (tag 1, data 1)
    Similar to above; U0 ← 1
  If U0 = U1 = 0, this condition does not exist
  If U0 = U1 = 1, Both of them are candidates,
    Take arbitrary selection

CACHE WRITE

Write Through

When writing into memory

  If Hit, both Cache and memory is written in parallel
  If Miss, Memory is written
    For a read miss, missing block may be
    overloaded onto a cache block

Memory is always updated
-> Important when CPU and DMA I/O are both executing

Slow, due to the memory access time

Write-Back (Copy-Back)

When writing into memory

  If Hit, only Cache is written
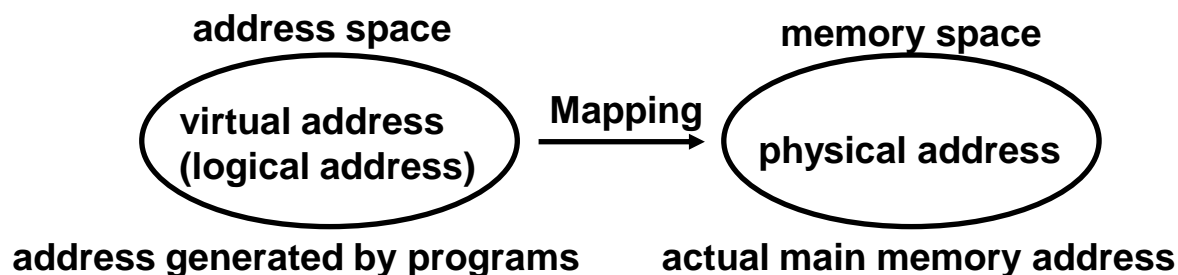 **If Miss, missing block is brought to Cache and write into Cache**
    For a read miss, candidate block must be
    written back to the memory

Memory is not up-to-date, i.e., the same item in
  Cache and memory may have different value

VIRTUAL MEMORY

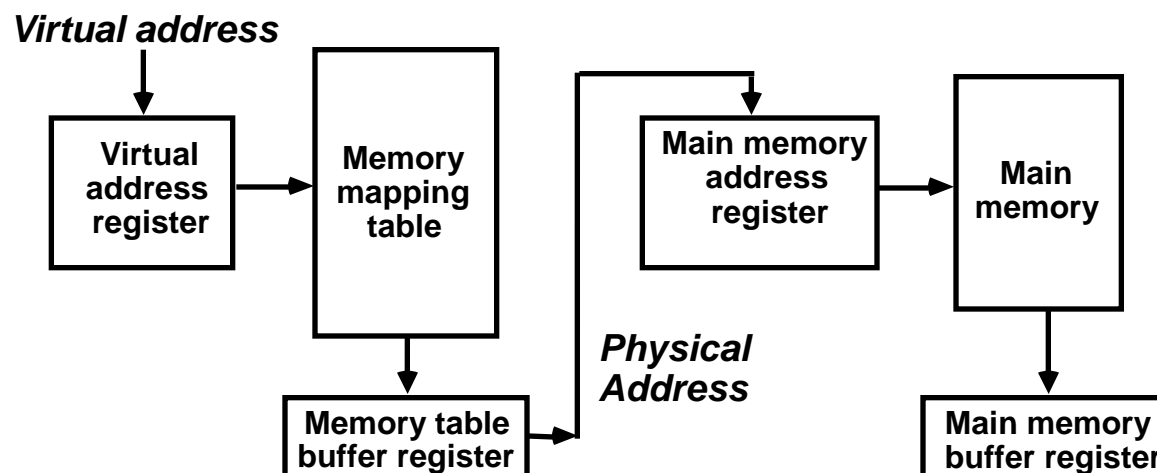**Give the programmer the illusion that the system has a very large memory, even though the computer actually has a relatively small main memory**

Address Space(Logical) and Memory Space(Physical)

address space   **memory space**

virtual address (logical address)   →Mapping→   **physical address**

address generated by programs   **actual main memory address**

Address Mapping
Memory *Mapping Table* for *Virtual Address -> Physical Address*

Virtual address

| Virtual address register | → | Memory mapping table | → | Main memory address register | → | Main memory |

Physical Address

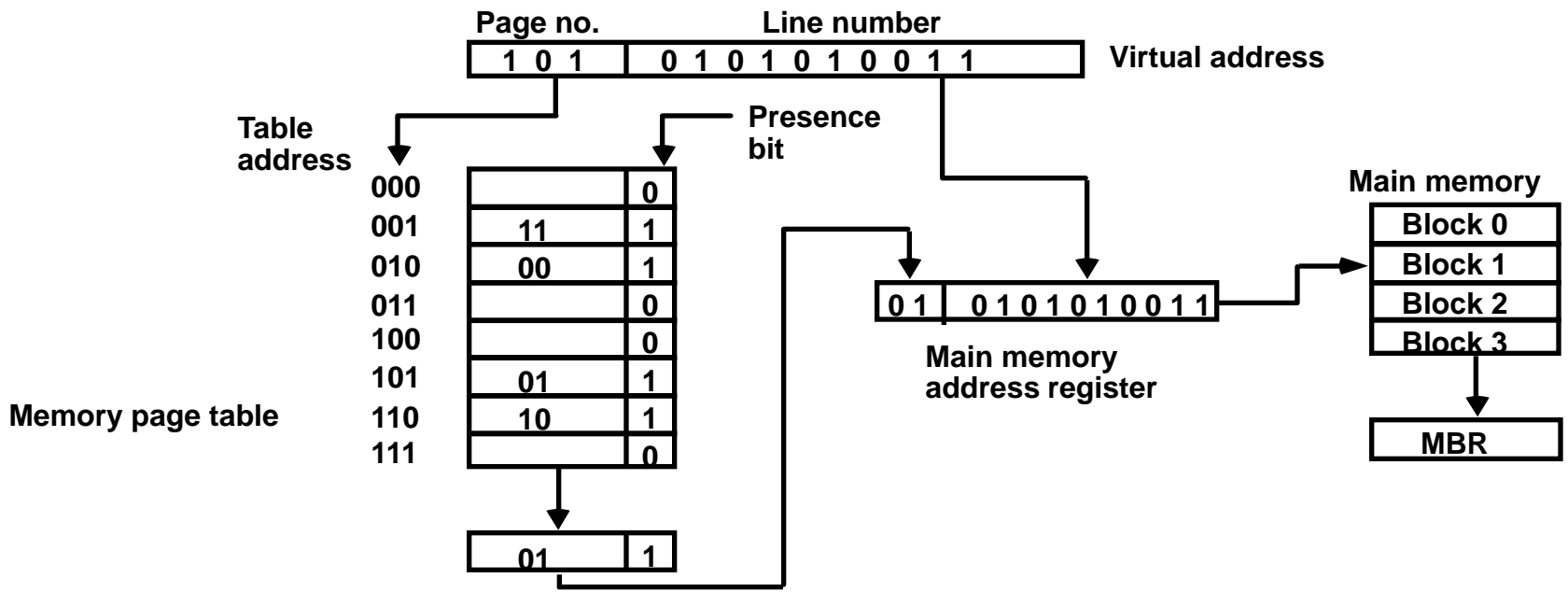| Memory table buffer register | | Main memory buffer register |

ADDRESS MAPPING

Address Space and Memory Space are each divided into fixed size group of words called *blocks* or *pages*

1K words group

Address space
$N = 8K = 2^{13}$

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

Memory space
$M = 4K = 2^{12}$

Block 0
Block 1
Block 2
Block 3
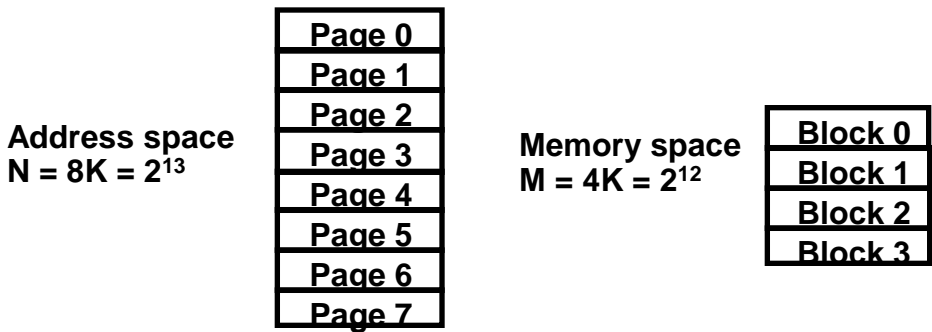
Organization of memory Mapping Table in a paged system

Page no.	Line number		
1 0 1	0 1 0 1 0 1 0 0 1 1		Virtual address

Table address

Presence bit

Memory page table

000		0
001	11	1
010	00	1
011		0
100		0
101	01	1
110	10	1
111		0

01	1

0 1	0 1 0 1 0 1 0 0 1 1

Main memory address register

Main memory

Block 0
Block 1
Block 2
Block 3

MBR

ASSOCIATIVE MEMORY PAGE TABLE

Assume that
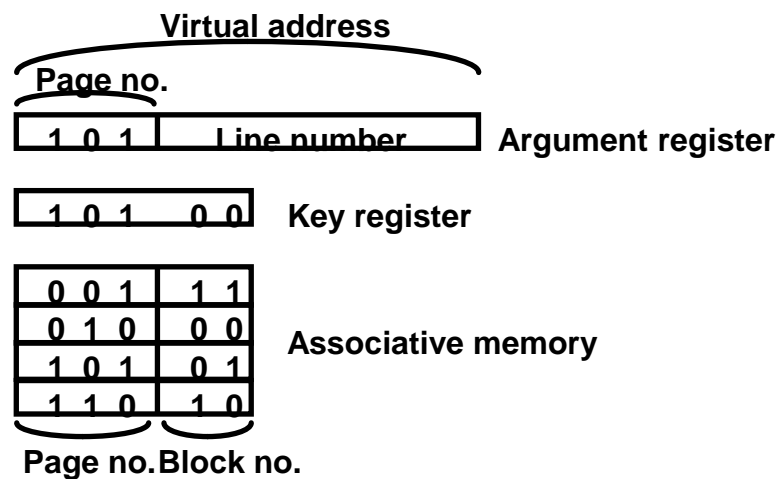   **Number of Blocks in memory = m**
   Number of Pages in Virtual Address Space = n

Page Table
   - Straight forward design -> n entry table in memory
    Inefficient storage space utilization
    <- n-m entries of the table is empty

   - More efficient method is m-entry Page Table
        Page Table made of an Associative Memory
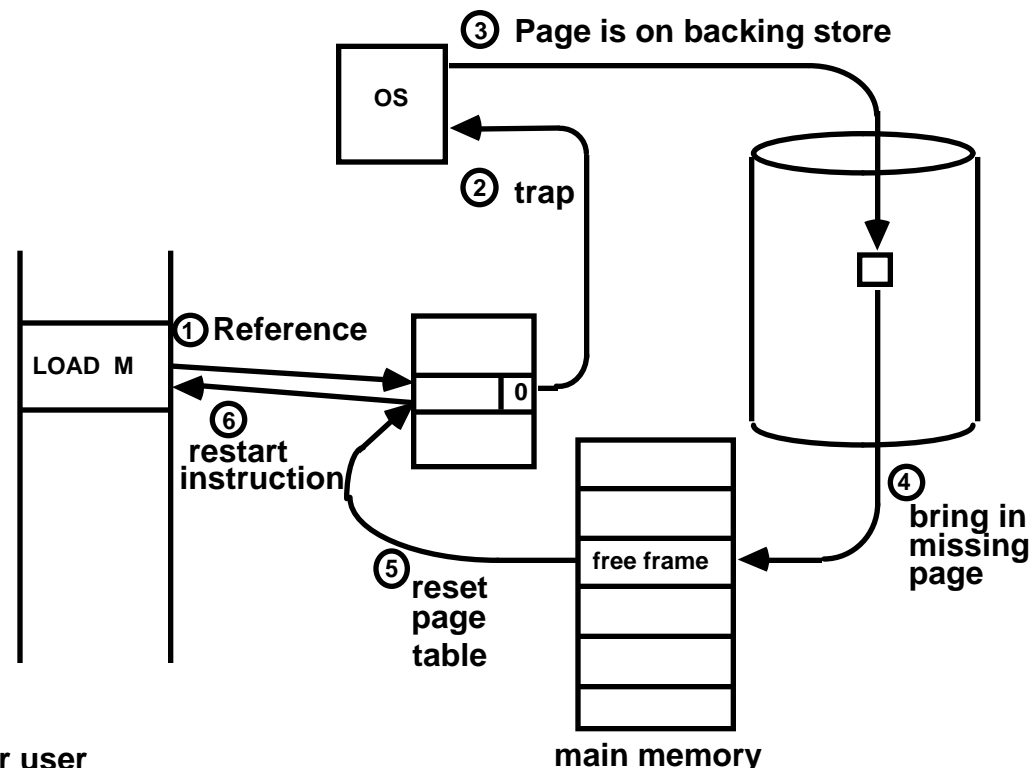   **m words; (Page Number:Block Number)**

Virtual address

Page no.

| 1 0 1 | Line number | **Argument register** |

| 1 0 1 | 0 0 | **Key register** |

0 0 1	1 1
0 1 0	0 0
1 0 1	0 1
1 1 0	1 0

Associative memory

Page no. Block no.

Page Fault
   Page number cannot be found in the Page Table

PAGE FAULT

1. **Trap to the OS**

2. **Save the user registers and program state**

3. **Determine that the interrupt was a page fault**

4. **Check that the page reference was legal and determine the location of the page on the backing store(disk)**

5. **Issue a read from the backing store to a free frame**

   a. **Wait in a queue for this device until serviced**

   b. **Wait for the device seek and/or latency time**

   c. **Begin the transfer of the page to a free frame**

6. **While waiting, the CPU may be allocated to some other process**

7. **Interrupt from the backing store (I/O completed)**

8. **Save the registers and program state for the other user**

9. **Determine that the interrupt was from the backing store**

10. **Correct the page tables (the desired page is now in memory)**

11. **Wait for the CPU to be allocated to this process again**

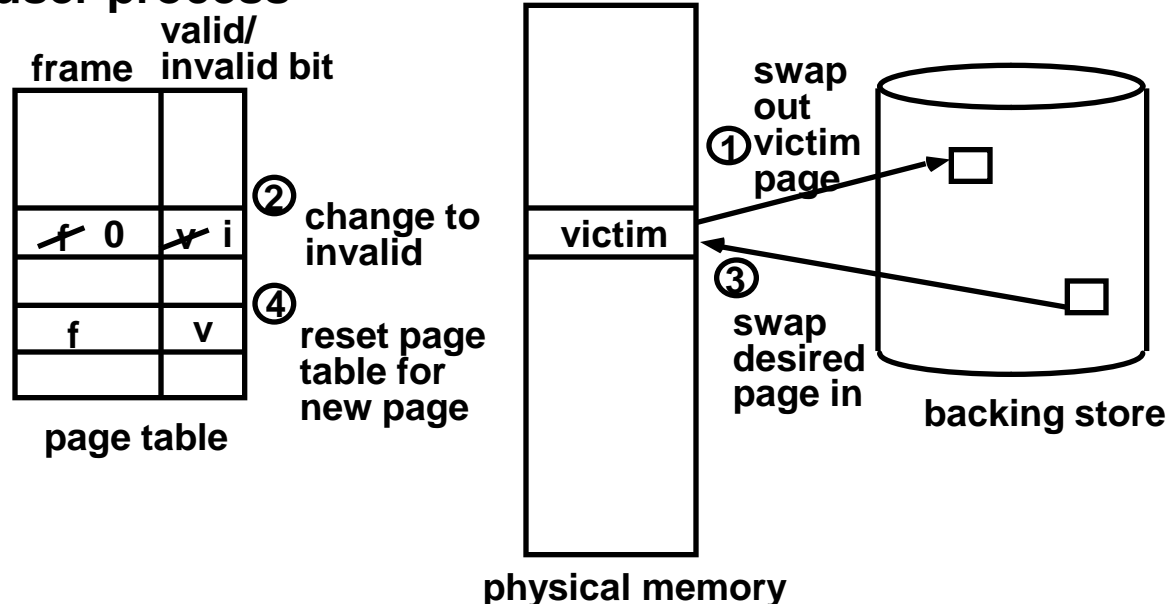12. **Restore the user registers, program state, and new page table, then resume the interrupted instruction.**

③ **Page is on backing store**

OS

② **trap**

① **Reference**

LOAD M

⑥ **restart instruction**

④ **bring in missing page**

free frame

⑤ **reset page table**

main memory

Processor architecture should provide the ability to restart any instruction after a page fault.

PAGE REPLACEMENT

**Decision on which page to displace to make room for
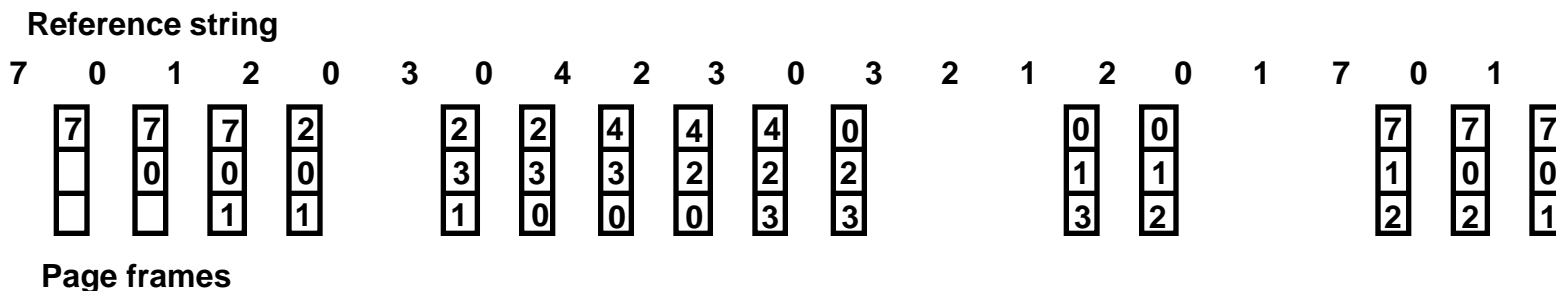an incoming page when no free frame is available**

Modified page fault service routine

1. Find the location of the desired page on the backing store

2. Find a free frame

> **- If there is a free frame, use it**

> **- Otherwise, use a page-replacement algorithm to select a *victim* frame**

> **- Write the victim page to the backing store**

3. Read the desired page into the (newly) free frame

4. Restart the user process

**valid/
invalid bit**

frame

② **change to
invalid**

**swap
out
① victim
page**

0   i

victim

④

③

f   v

**reset page
table for
new page**

**swap
desired
page in**

page table

backing store

physical memory

PAGE REPLACEMENT ALGORITHMS

FIFO

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1

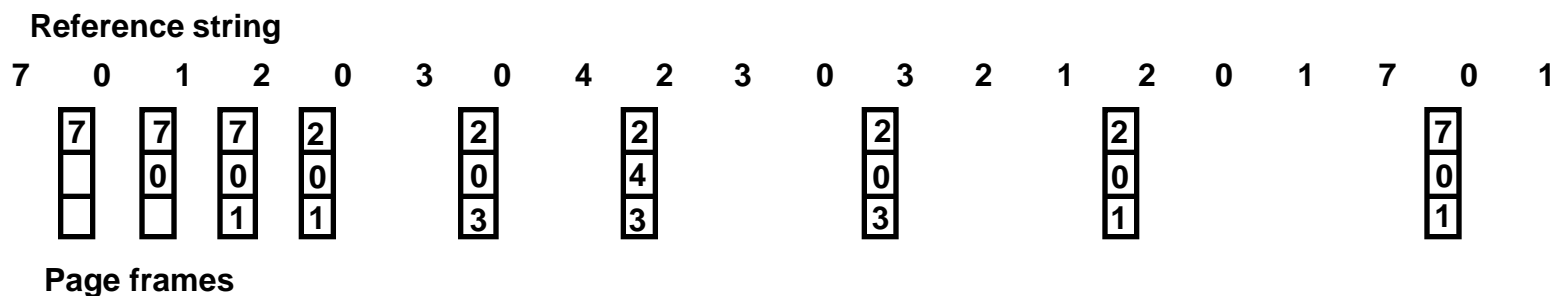| 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| | | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |

Page frames

FIFO algorithm selects the page that has been in memory the longest time
Using a queue - every time a page is loaded, its
identification is inserted in the queue
- Easy to implement
May result in a frequent page fault

Optimal Replacement (OPT) - Lowest page fault rate of all algorithms

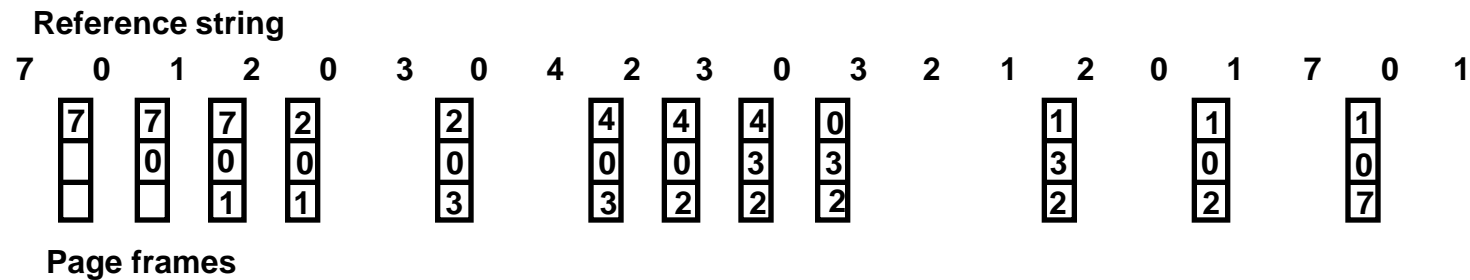**Replace that page which will not be used for the longest period of time**

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1

7	7	7	2		2		2			2			2			7			
	0	0	0		0		4			0			0			0			
		1	1		3		3			3			1			1			

Page frames

PAGE REPLACEMENT ALGORITHMS

LRU

- **OPT is difficult to implement since it requires future knowledge**
- **LRU uses the recent past as an approximation of near future.**

> **Replace that page which has not been used for the longest period of time**

Reference string

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

7	7	7	2		2		4	4	4	0				1		1		1	
	0	0	0		0		0	0	3	3				3		0		0	
		1	1		3		3	2	2	2				2		2		7	

Page frames

- **LRU may require substantial hardware assistance**
- **The problem is to determine an order for the frames defined by the time of last use**

PAGE REPLACEMENT ALGORITHMS

LRU Implementation Methods

• **Counters**

    - For each page table entry - time-of-use register

    - Incremented for every memory reference

    - Page with the smallest value in time-of-use register is replaced

• **Stack**

    - Stack of page numbers

    **- Whenever a page is referenced its page number is
removed from the stack and pushed on top**

    - Least recently used page number is at the bottom

Reference string

4   7   0   7   1   0   1   2   1   2   7   1   2

2
1
0
7
4

7
2
1
0
4
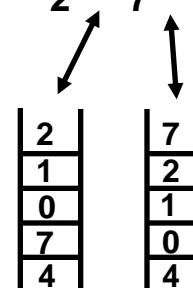
LRU Approximation

    - Reference (or use) bit is used to approximate the LRU

    **- Turned on when the corresponding page is
referenced after its initial loading**

    - Additional reference bits may be used
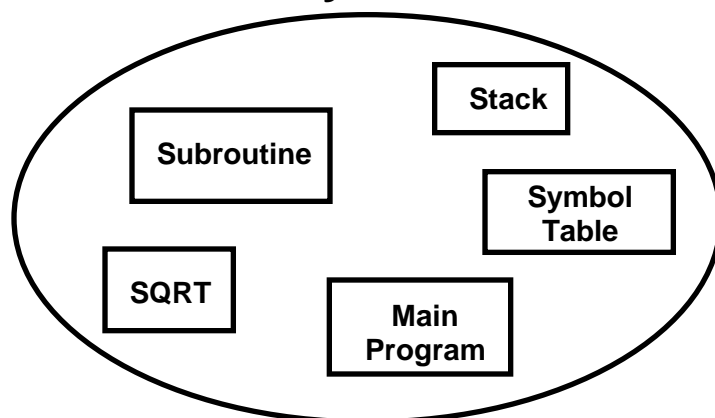
MEMORY MANAGEMENT HARDWARE

Basic Functions of MM

- *Dynamic Storage Relocation* - **mapping logical**
  memory references to physical memory references
- **Provision for** *Sharing* **common information stored**
  in memory by different users
- *Protection* **of information against unauthorized access**

Segmentation

- **A segment is a set of logically related instructions**
  or data elements associated with a given name
- **Variable size**

User's view of memory

Subroutine

Stack

Symbol Table

SQRT

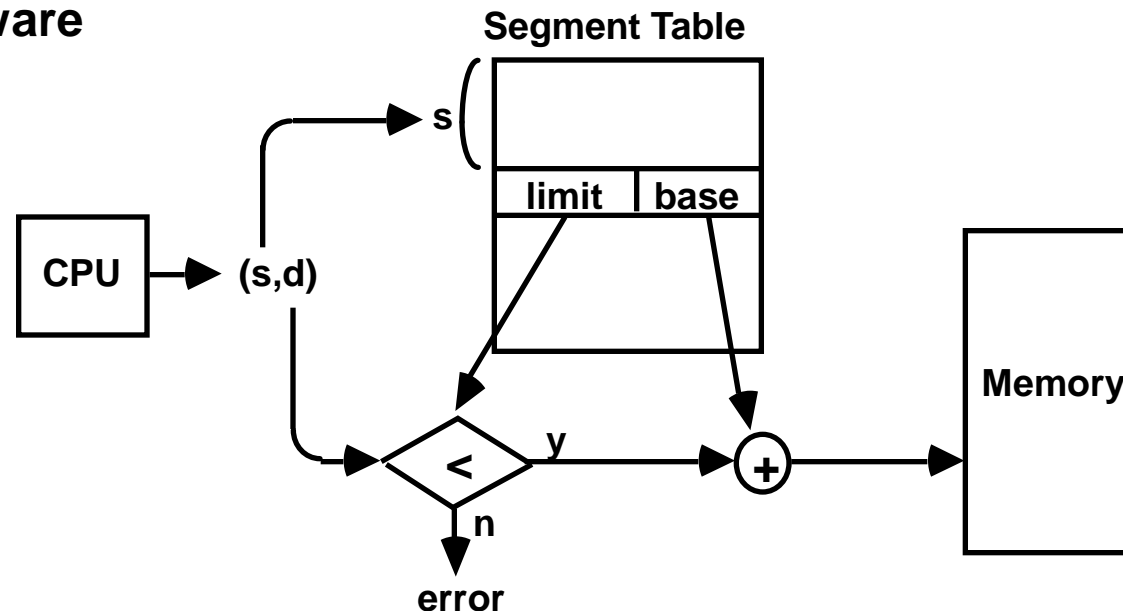**Main Program**

The user does not think of memory as a linear array of words. Rather the user prefers to view memory as a collection of variable sized segments, with no necessary ordering among segments.
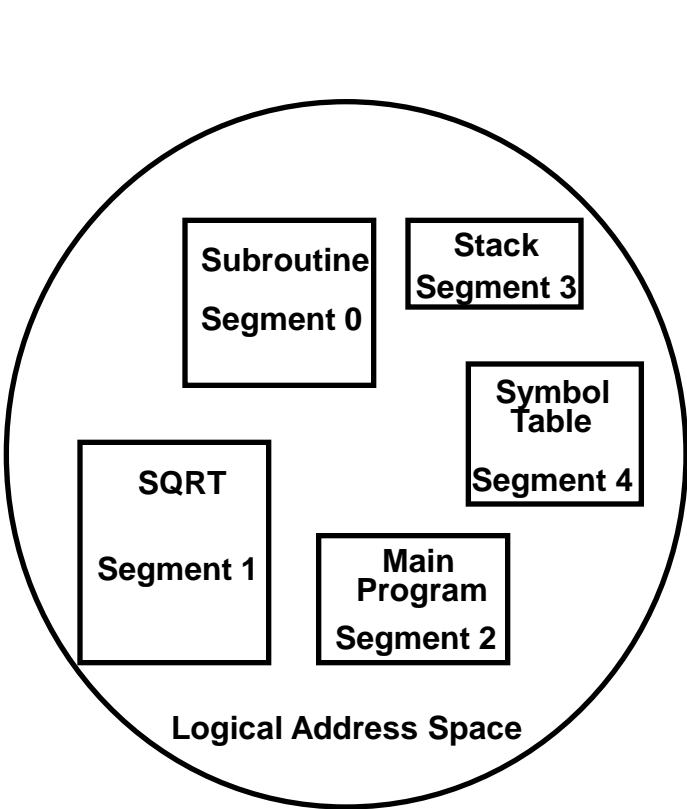
User's view of a program

SEGMENTATION

- **A memory management scheme which supports**
  user's view of memory
- **A logical address space is a collection of segments**
- **Each segment has a name and a length**
- **Address specify both the segment name and the**
  offset within the segment.
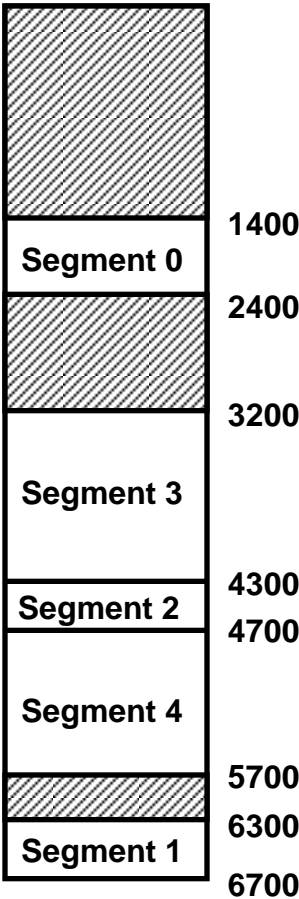- **For simplicity of implementations, segments are numbered.**
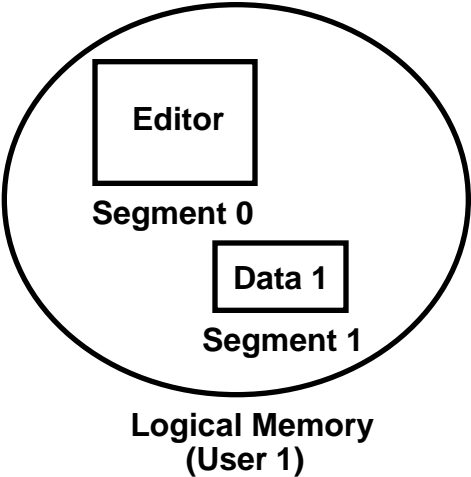
Segmentation Hardware

SEGMENTATION EXAMPLE

Subroutine

Segment 0

Stack
Segment 3

Symbol
Table

Segment 4

SQRT

Segment 1

Main
Program

Segment 2

Logical Address Space

1400

Segment 0

2400

3200

Segment 3

4300

Segment 2

4700

Segment 4

5700

6300

Segment 1

6700

Segment Table

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

SHARING OF SEGMENTS

Editor

Segment 0

Data 1

Segment 1

**Logical Memory
(User 1)**

	limit	base
0	25286	43062
1	4425	68348

**Segment Table
(User 1)**

Editor

Segment 0

Data 2

Segment 1

**Logical Memory
(User 2)**
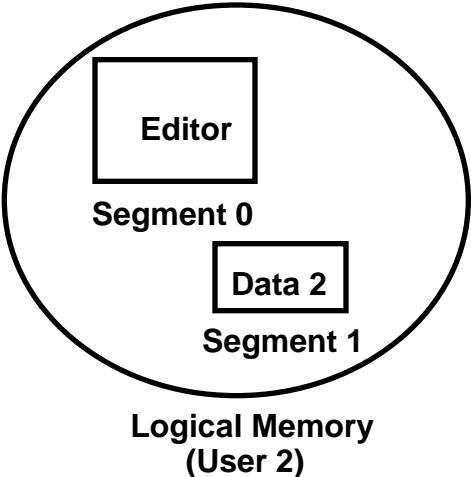
	limit	base
0	25286	43062
1	8550	90003

**Segment Table
(User 2)**

43062

Editor

68348

Data 1

72773

90003

Data 2

98556

Physical Memory

SEGMENTED PAGE SYSTEM

Logical address

Segment	Page	Word

Segment table

Page table

+

Block	Word

Physical address
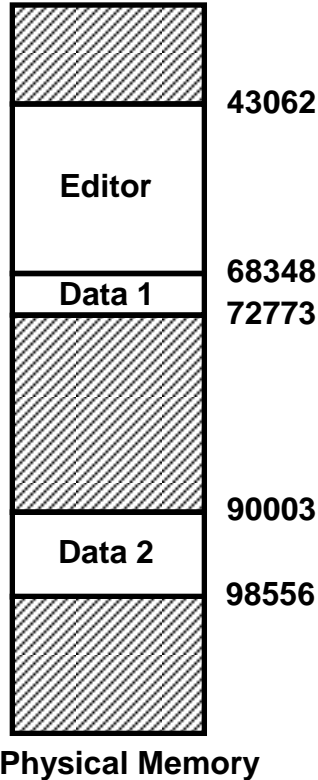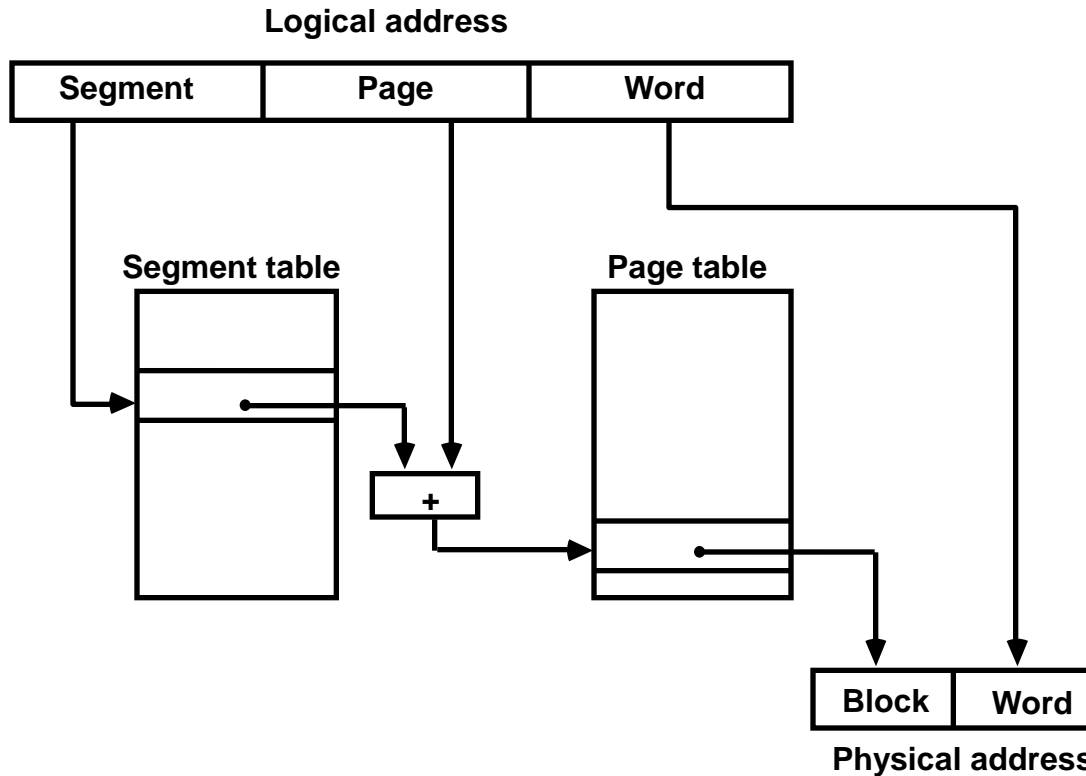
IMPLEMENTATION OF PAGE AND SEGMENT TABLES

Implementation of the Page Table

- **- Hardware registers (if the page table is reasonably small)**
- **- Main memory**

    - **Page Table Base Register(PTBR) points to PT**
    - **Two memory accesses are needed to access a word; one for the page table, one for the word**

- **- Cache memory (TLB: Translation Lookaside Buffer)**

    - **To speedup the effective memory access time, a special small memory called associative memory, or cache is used**
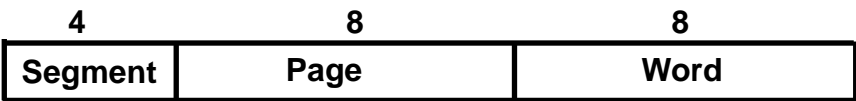
Implementation of the Segment Table

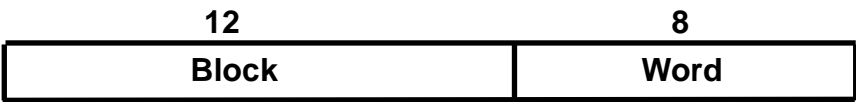   **Similar to the case of the page table**

EXAMPLE

Logical and Physical Addresses

Logical address format: 16 segments of 256 pages each, each page has 256words

4	8	8
Segment	Page	Word

2^{20} x 32 Physical memory

Physical address format: 4096 blocks of 256 words each, each word has 32bits

12	8
Block	Word

Logical and Physical Memory Address Assignment

Hexa address   **Page number**

Hexa address	Page number
60000	Page 0
60100	Page 1
60200	Page 2
60300	Page 3
60400	Page 4
604FF	

Segment	Page	Block
6	00	012
6	01	000
6	02	019
6	03	053
6	04	A61

(a) Logical address assignment

(b) Segment-page versus memory block assignment

LOGICAL TO PHYSICAL MEMORY MAPPING

Segment and page table mapping

Logical address (in hexadecimal)

6	02	7E

Segment table		Page table		Physical memory	
0		00		00000	Block 0
				000FF	
6	35	35	012		
		36	000	01200	
		37	019		Block 12
		38	053	012FF	
F	A3	39	A61		
				01900	32-bit word
				0197E	
		A3	012	019FF	
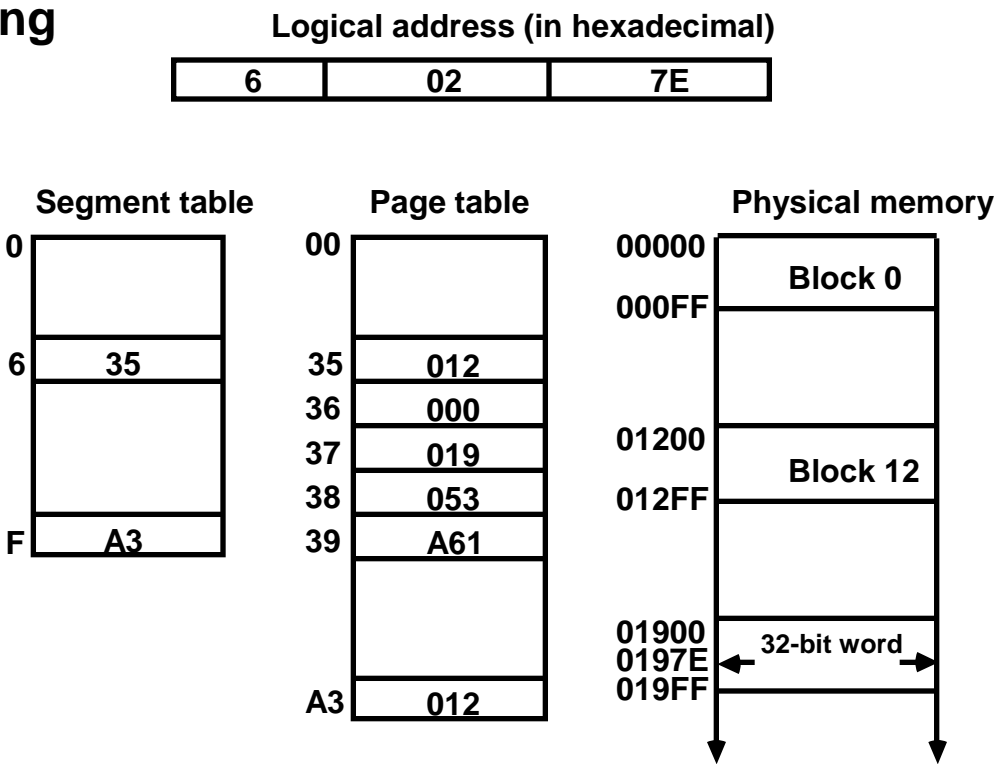
Associative memory mapping

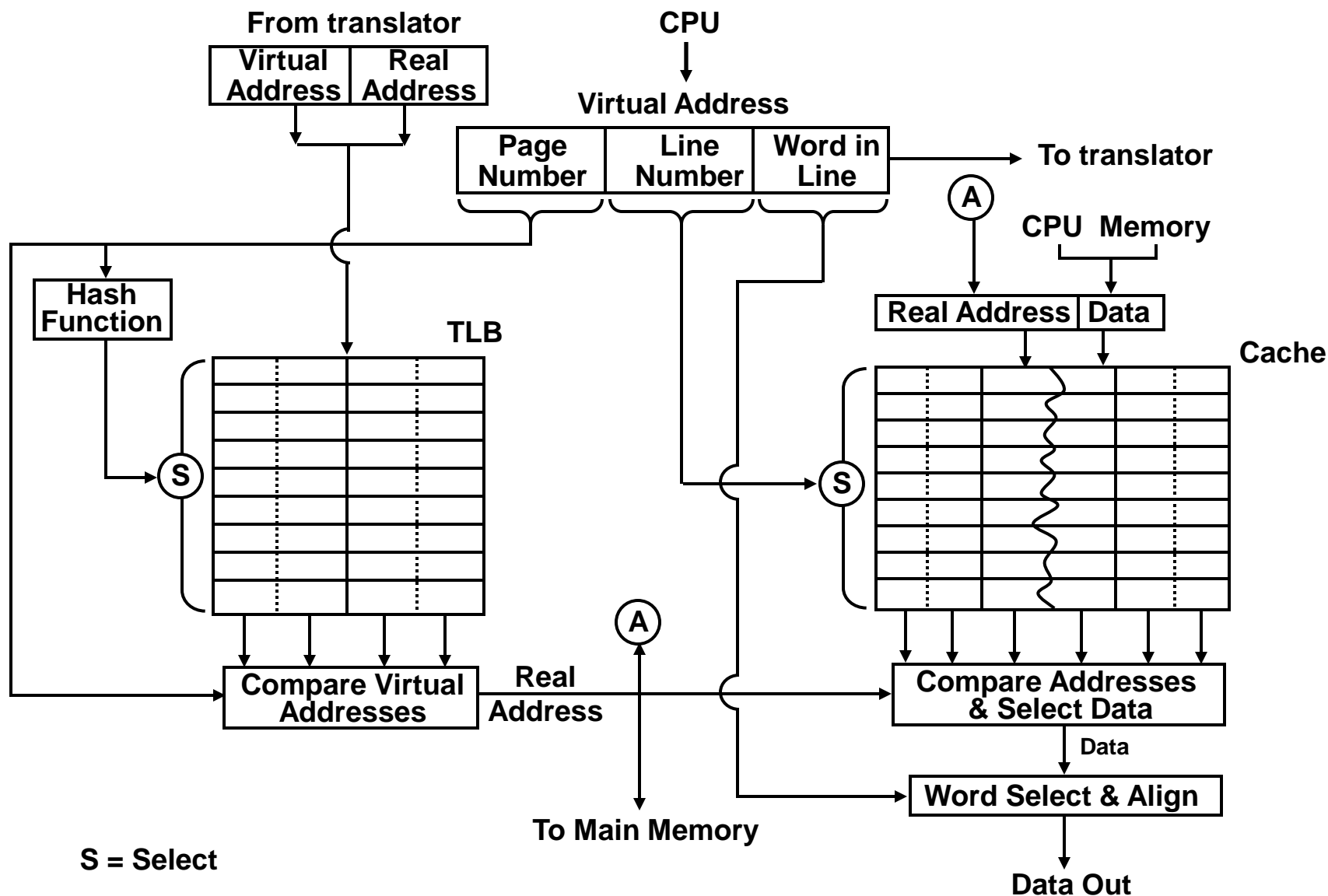| Segment	Page	Block
6	02	019
6	04	A61

MEMORY PROTECTION

Protection information can be included in the segment table or segment register of the memory management hardware

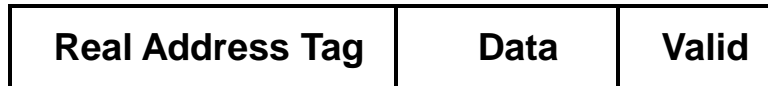**- Format of a typical segment descriptor**

Base address	Length	Protection

- The protection field in a segment descriptor specifies the *Access Rights* to the particular segment

- In a segmented-page organization, each entry in the page table may have its own protection field to describe the Access Rights of each page

- Access Rights: **Full read and write privileges.**
Read only (write protection)
Execute only (program protection)
System only (O.S. Protection)

A TYPICAL CACHE AND TLB DESIGN

From translator

Virtual Address	Real Address

CPU

Virtual Address

Page Number	Line Number	Word in Line

→ **To translator**

A

CPU   Memory

Hash Function

TLB

Real Address	Data

Cache

S

S

Compare Virtual Addresses	Real Address

A

Compare Addresses & Select Data

Data

To Main Memory

Word Select & Align

S = Select

Data Out

Structure of Cache Entry and Cache Set

Real Address Tag	Data	Valid

Cache Entry

Entry 1	Entry 2	• • •	Entry E	Replacement status

Cache Set

Cache Operation Flow Chart
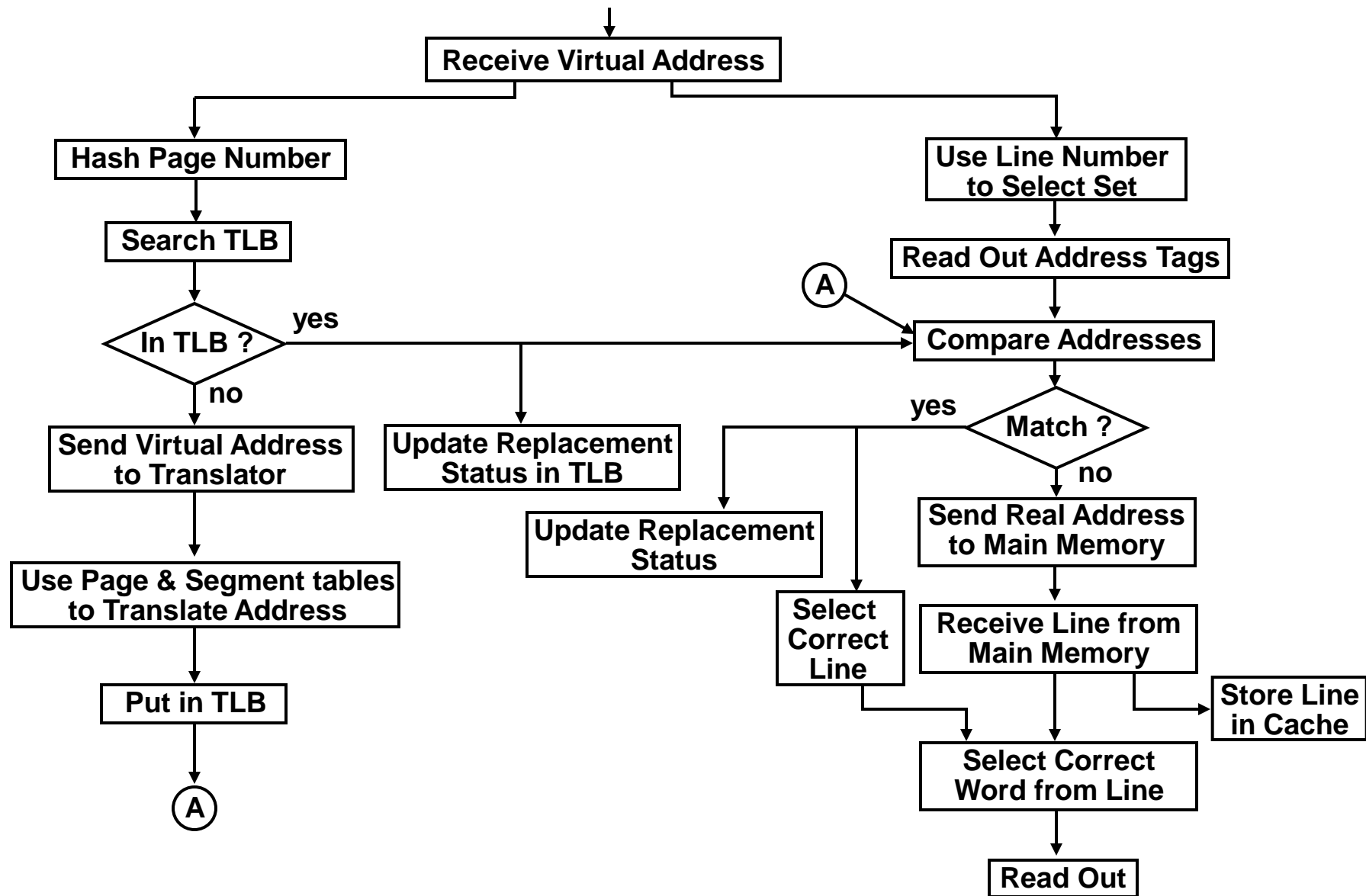
Virtual Address Format - Example

Page number

Byte within page

Byte within line

| 31 | 21 | 20 | 17 | 12 | 11 | 10 | 4 | 3 | 2 | 1 | 0 |

Select set
in TLB

Select set
in cache

Byte within
word

Map through
page directory

Map through
page table

Line number

Word within
line
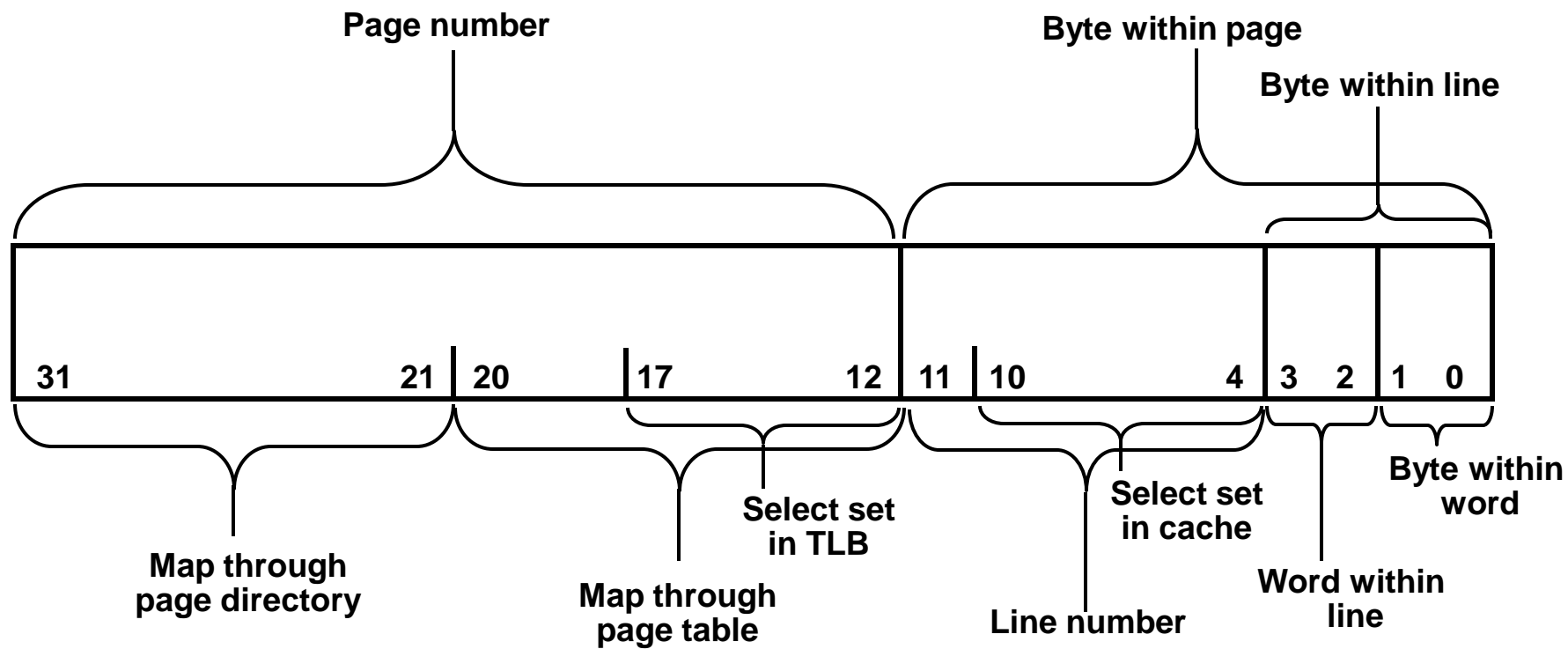
Virtual Address of Fairchild Clipper