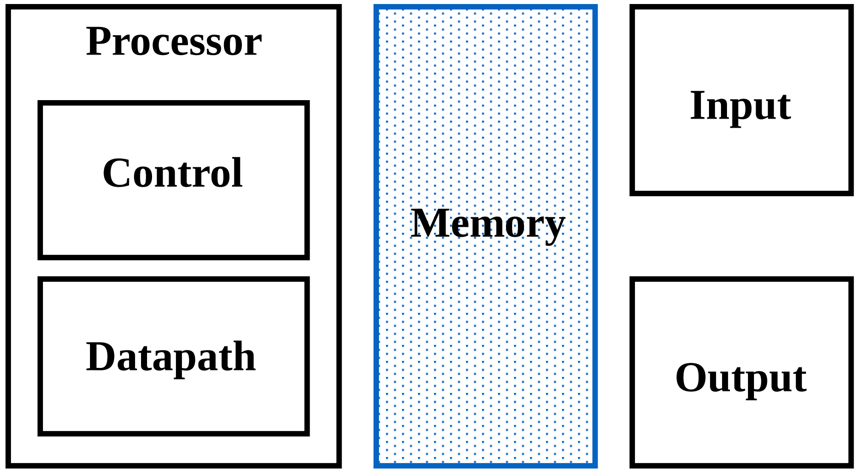
Memory Organization

for better performance and cost effective systems

Computer Organization and Architecture, William Stallings

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
- Memory is usually implemented as:
  - Dynamic Random Access Memory (DRAM) - for main memory
  - Static Random Access Memory (SRAM) - for cache



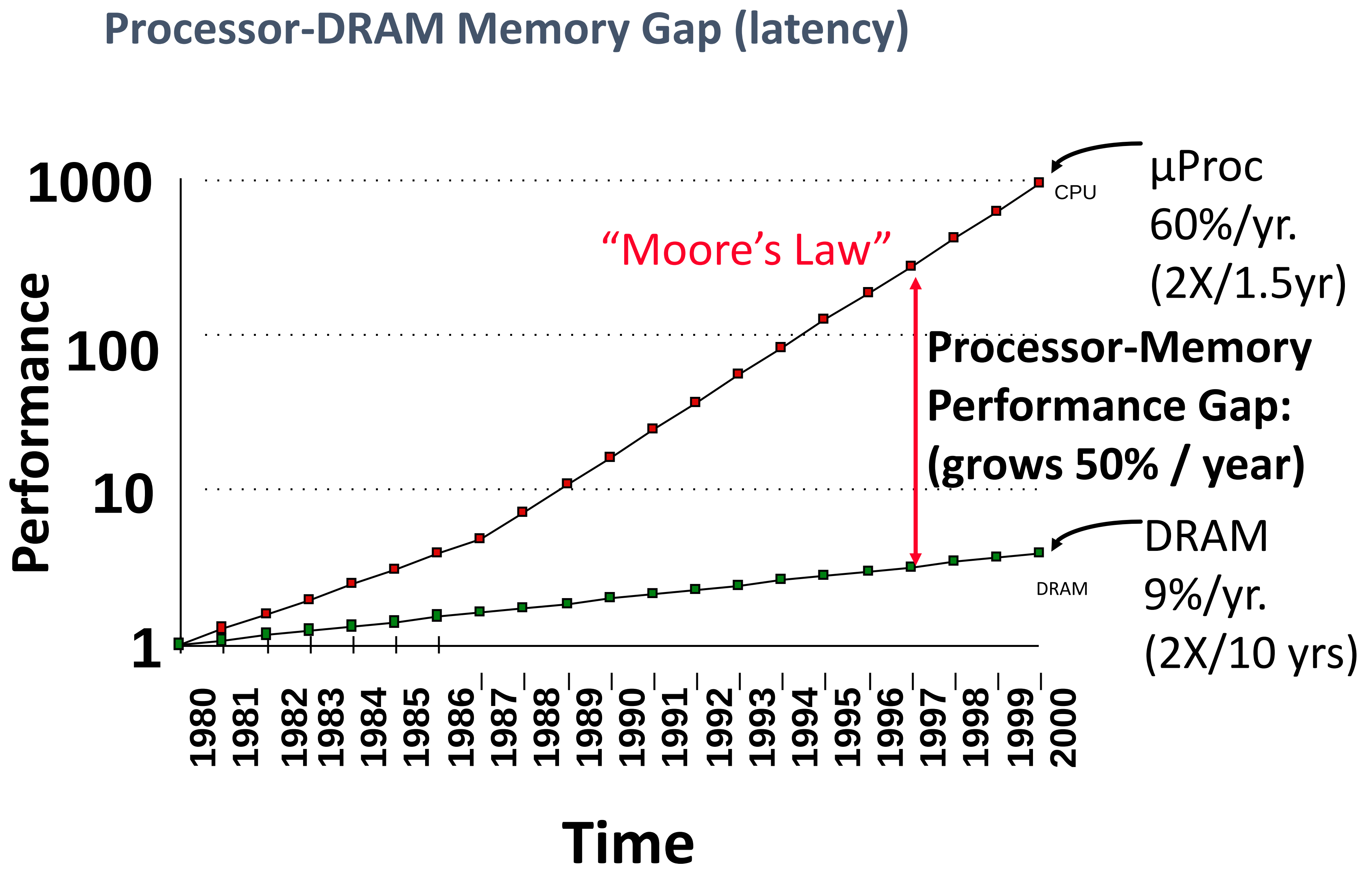
Characteristics of memory systems

- Location
  - CPU ; registers
  - Internal (on board) ; cache and RAM
  - External ; HDD
- Capacity
  - Word size and number of words (i.e. number of memory locations)
- Access method
  - Sequential ; Tape
  - Random ; RAM or ROM
  - Direct or semi-random ; HDD

Characteristics of memory systems

- Physical type
  - Semiconductor ; RAM, ROM
  - magnetic ; HDD, FDD
  - Optical ; CDs, DVDs
- Physical characteristics
  - Volatile and non-volatile
  - Erasable and non- erasable

Why should we bother about Memory speed?



Characteristics of memory systems

- Performance
  - Access time
    - For RAM access time is the time between presenting an address to memory and getting the data on the bus
  - Cycle time
    - Primarily applied to RAM; access time + additional time before a second access can start

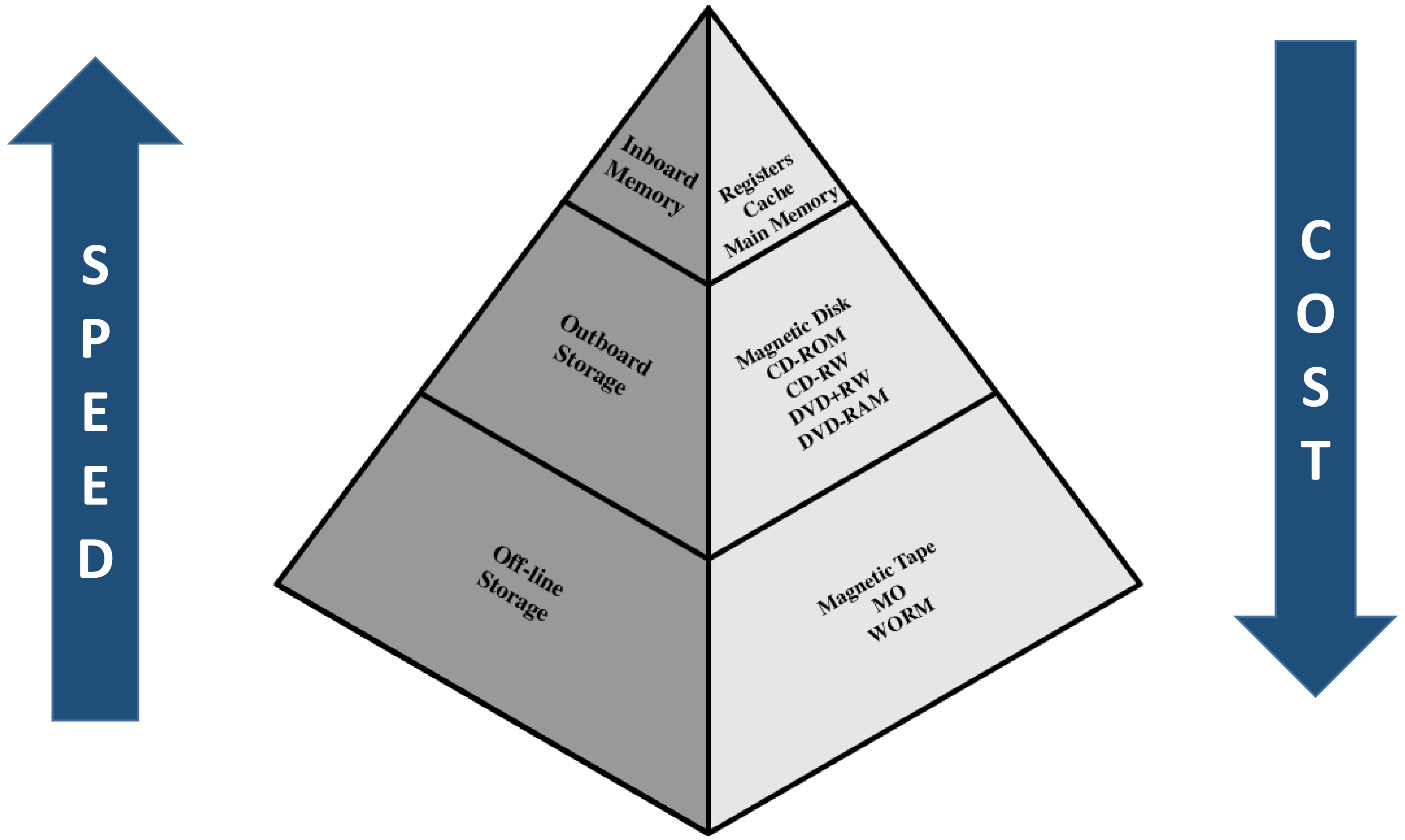
  — memory system needs to recover for the next access

  — Function of memory components and system bus, not the processor

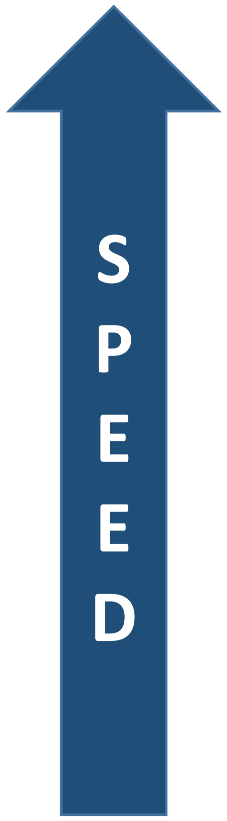
Semiconductor RAM

- Static RAM
  - Flip-flop based – data is in flip-flops
  - Fast
  - Bulky
  - expensive
- Dynamic RAM
  - Capacitor based - data is in form of charge on the capacitor
  - Needs refreshing of data
  - Compact
  - Cheap
  - Slower

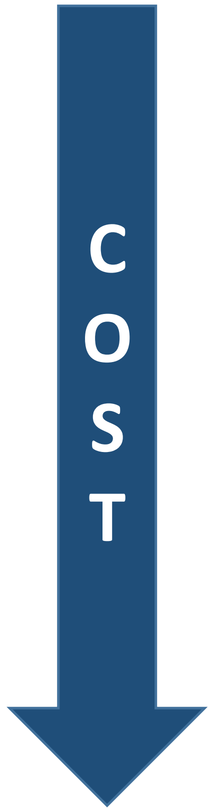
Memory Hierarchy - Diagram



Hierarchy List



- Registers – semiconductor memory on the CPU; fastest
- Cache – Static RAM placed close to CPU
- Main memory – Dynamic RAM placed on board
- Disk – for bulk data
- Optical – for bulk data
- Tape – for bulk data



The Bottom Line

- Performance of CPU depends on access time of memory.
- How to build a system which is having best performance and at the same time low cost?
- How much?
  - Capacity KB , MB, GB, TB ?
- How fast?
  - Access / Transfer Rate
- How expensive?
  - $$$$$

fast and cost effective system

- Is it possible to build a computer which uses only static RAM (large capacity of fast memory)?
- This would be a very fast computer
- also very costly

- To reduce cost, can we use the below mentioned configuration?
  - Kilo bytes of cache (SRAM)
  - Mega bytes of DRAM
  - Giga bytes of secondary memory
  - This will definitely cost less
- will this give best performance?

Locality of reference

- We know that code should be loaded in main memory for it to execute
- One good observation is that during the course of the execution of a program, memory references tend to cluster
- e.g. programs - loops, nesting, …

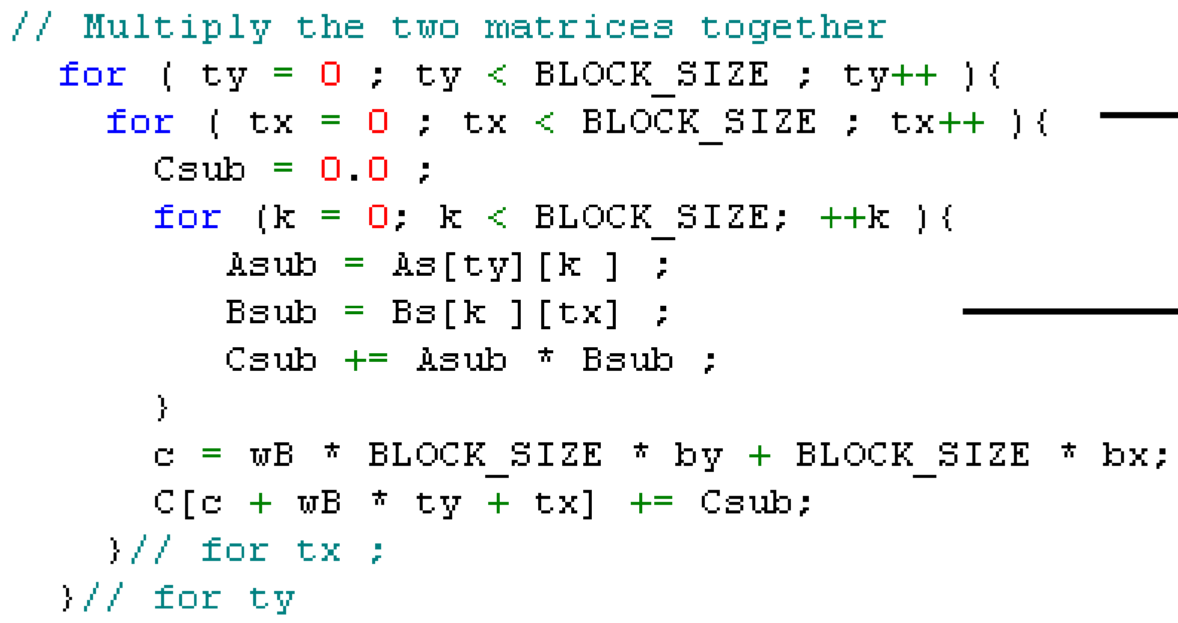
  data – strings, lists, arrays, …

- Therefore, answer to the previous question on performance is YES!!!
- WE GET THE BEST PERFORMANCE AT LESS PRICE
- Its all possible only due to ‘ locality of reference’

Principle of locality

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon.

- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.



```
// Multiply the two matrices together
  for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
    for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
      Csub = 0.0 ;
      for (k = 0; k < BLOCK_SIZE; ++k ){
         Asub = As[ty][k ] ;
         Bsub = Bs[k ][tx] ;
         Csub += Asub * Bsub ;
      }
      c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
      C[c + wB * ty + tx] += Csub;
    }// for tx ;
  }// for ty
```

for-loop is temporal locality

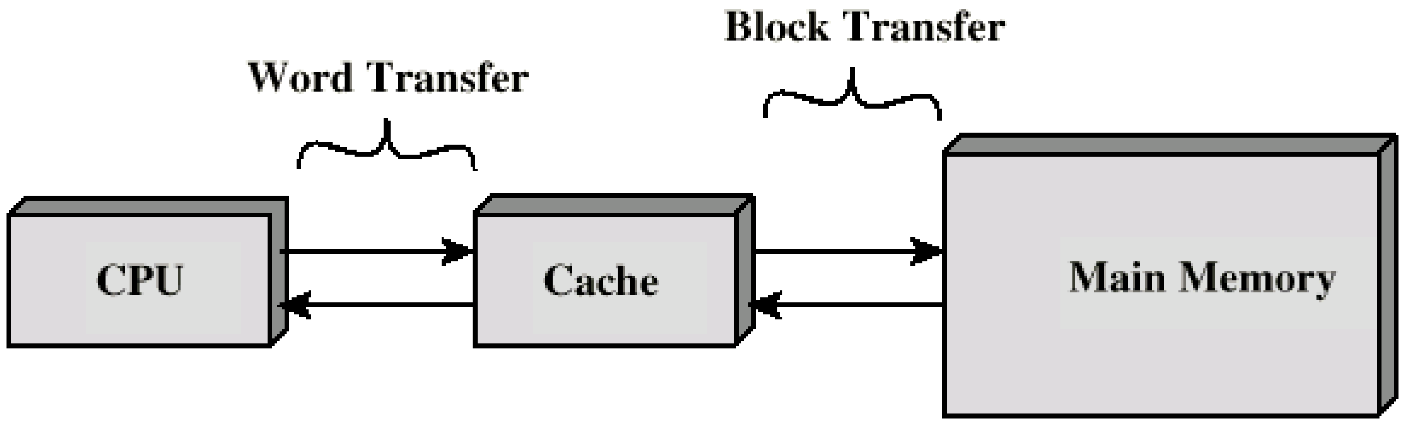
array is spatial locality

The Principal of Locality of reference

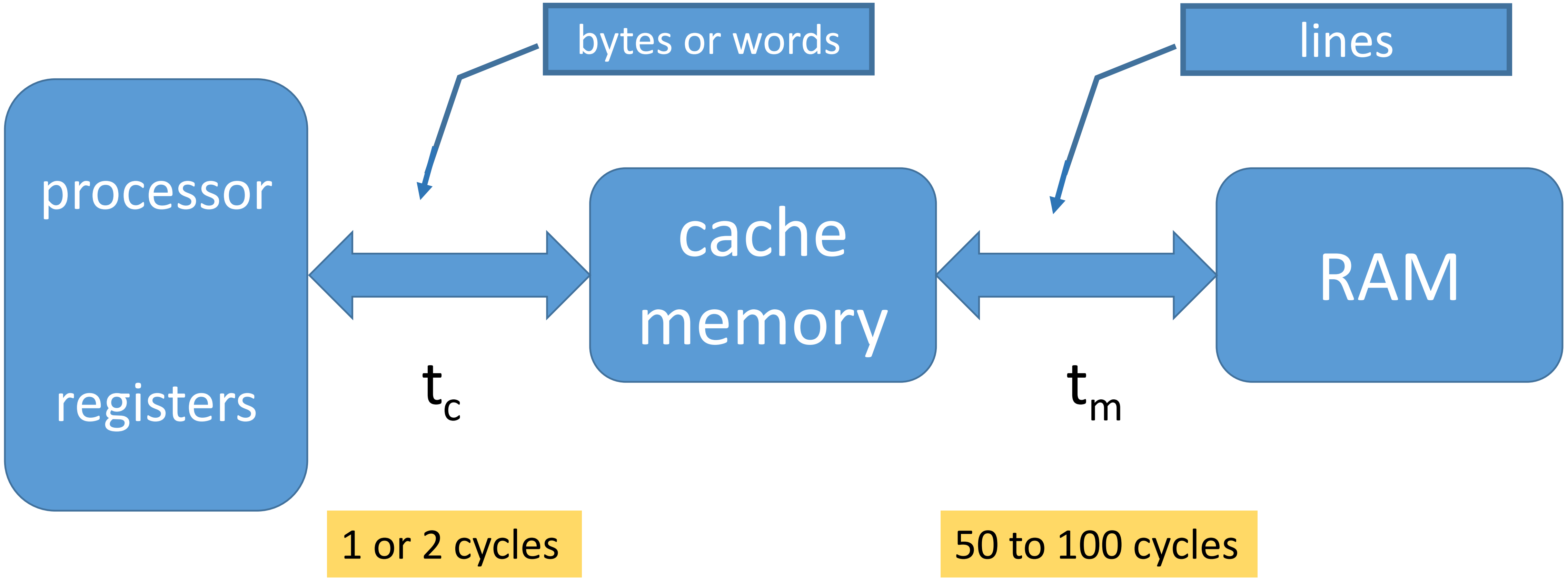
- How can we use very fast memory effectively?
- Temporal Locality (Close in Time)
  - Memory that has been accessed recently is most likely to be accessed again sooner
- Spatial Locality (Close in location)
  - Memory that is close to memory that has been accessed recently is most likely to be accessed
- Organize memory in blocks
  - Keep blocks likely to be used soon in very fast memory
  - Keep the next most likely blocks in medium fast memory
  - Keep those not likely to be used soon in slower memory

Cache memory

- Small amount of fast memory
- Sits between main memory (RAM) and CPU
- May be located on CPU chip or in system
- Objective is to make slower memory system look like fast memory.

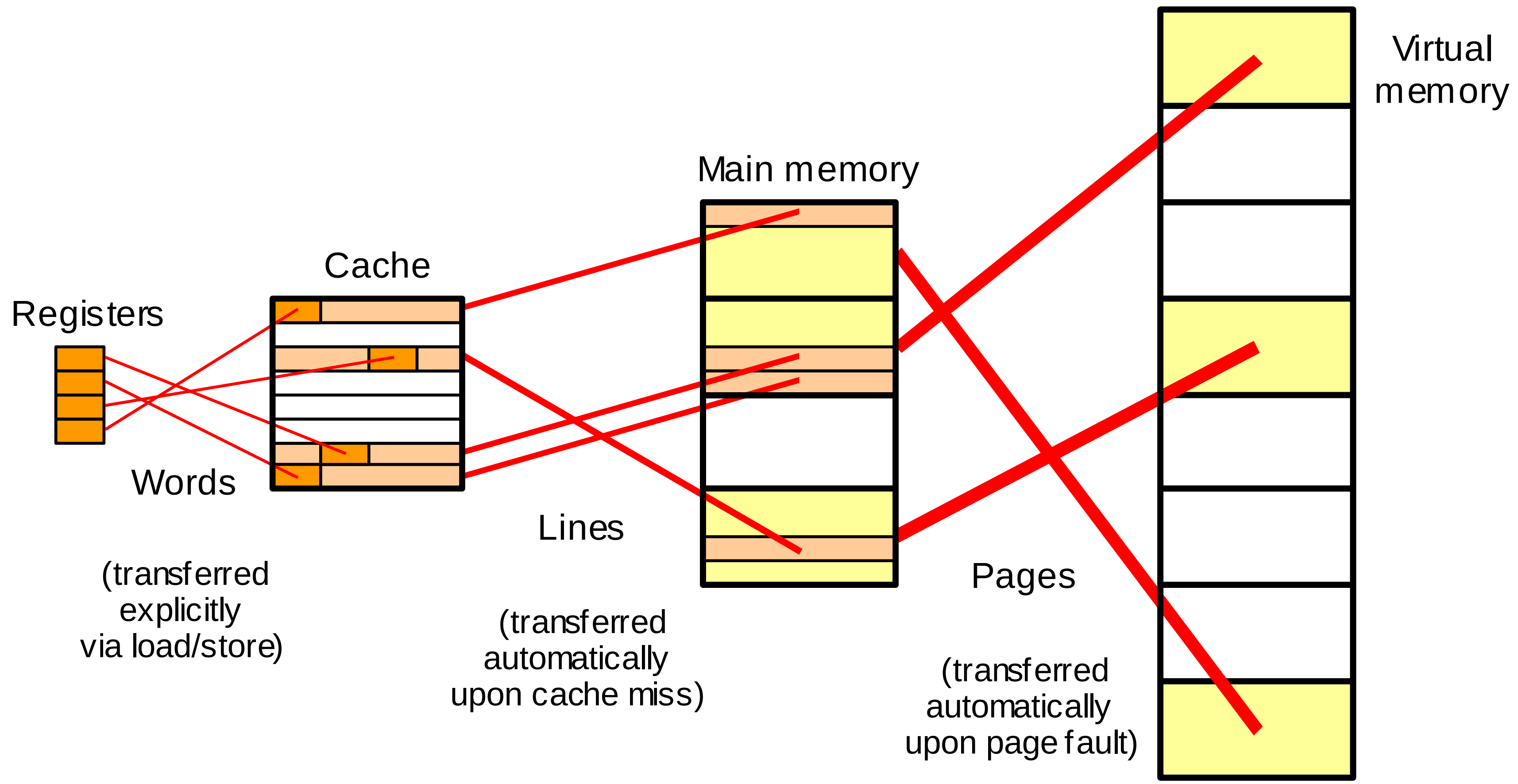


Cache memory



How much would be typical these access time in terms of cycles?

Memory Hierarchy: The Big Picture

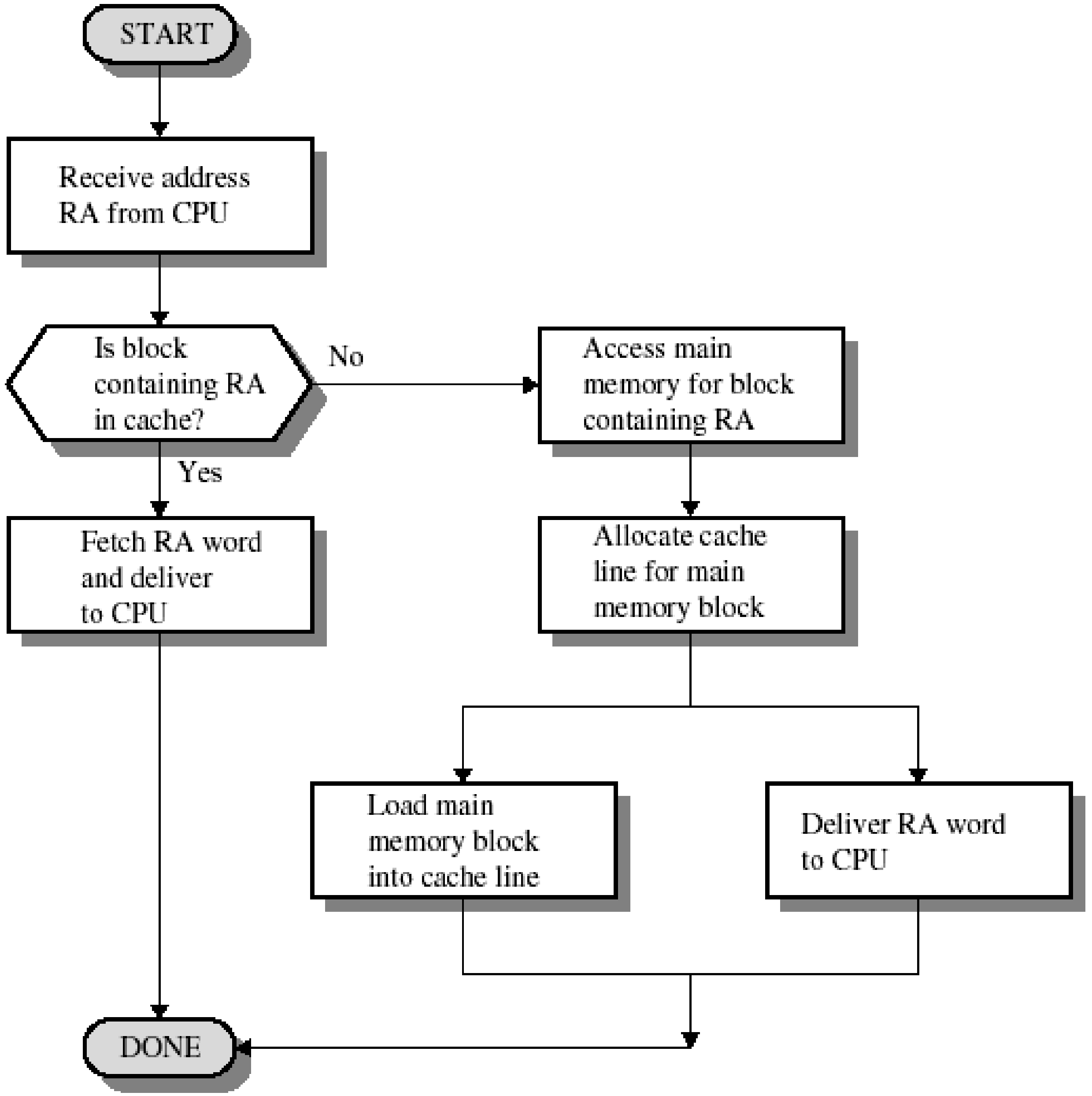

Registers
Cache
Main memory
Virtual memory
Words
(transferred explicitly via load/store)
Lines
(transferred automatically upon cache miss)
Pages
(transferred automatically upon page fault)

Data movement in a memory hierarchy.

Cache operation – overview

- CPU requests contents of memory location
- Check if this requested data is in cache
- If found in cache, it’s a hit;
- Else, it’s a miss ;
- Cache "misses" are unavoidable, i.e., every piece of data and code must be loaded at least once
- What does a processor do during a miss?
  - It waits for the data to be loaded from RAM to cache.
  - Loading is done in two possible ways:
    1. read required block from main memory to cache then deliver from cache to CPU (cache physically between CPU and bus)
    2. read required block from main memory to cache and simultaneously deliver to CPU (CPU and cache both receive data from the same data bus buffer)

Cache Operation



START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

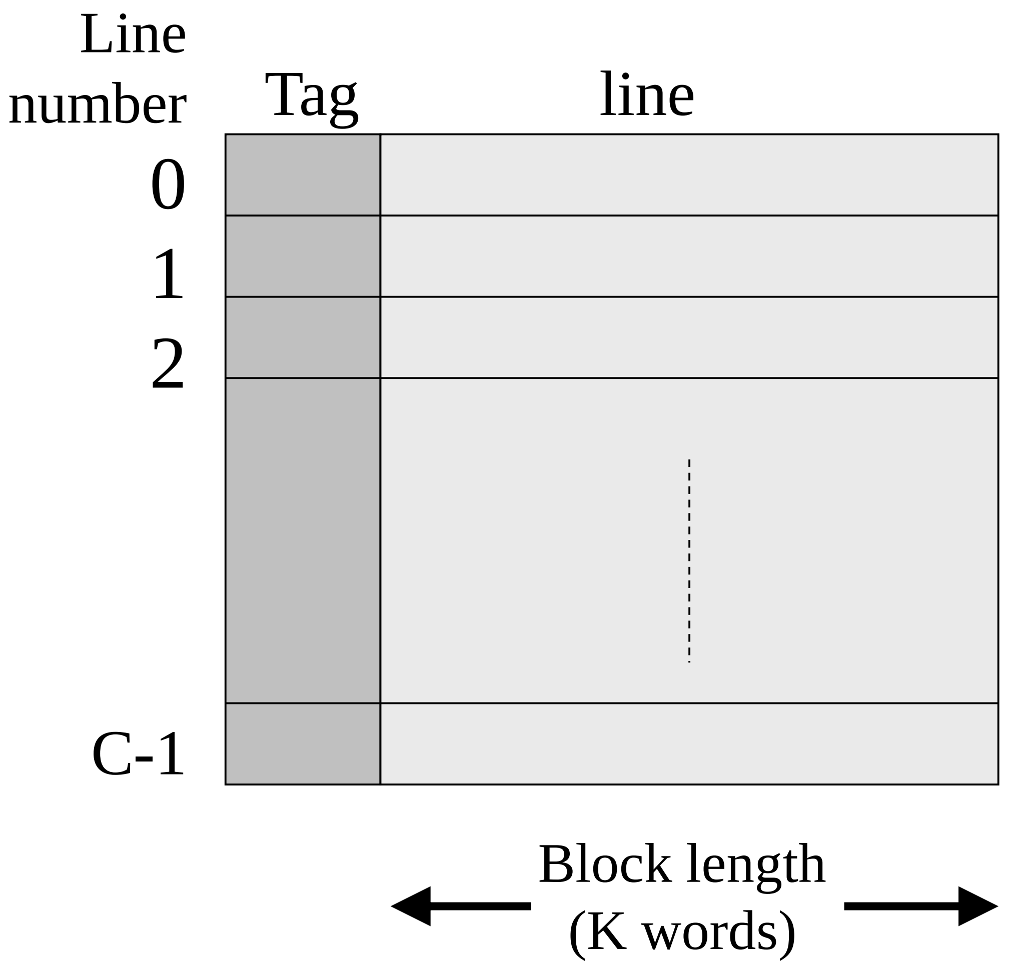
Few terms

- Cache hit rate, h
  - number of times the CPU found the required data in cache
  - Expressed in percentage
- Cache miss, (1-h)
  - number of times the CPU did not find the required data in cache
  - Expressed in percentage; typical numbers 3%, 10% etc.
- Average access time, $\bar{t} = h * t_c + (1 - h) * (t_m + t_c)$
- Cache efficiency, $\Lambda = \frac{t_c}{\bar{t}}$

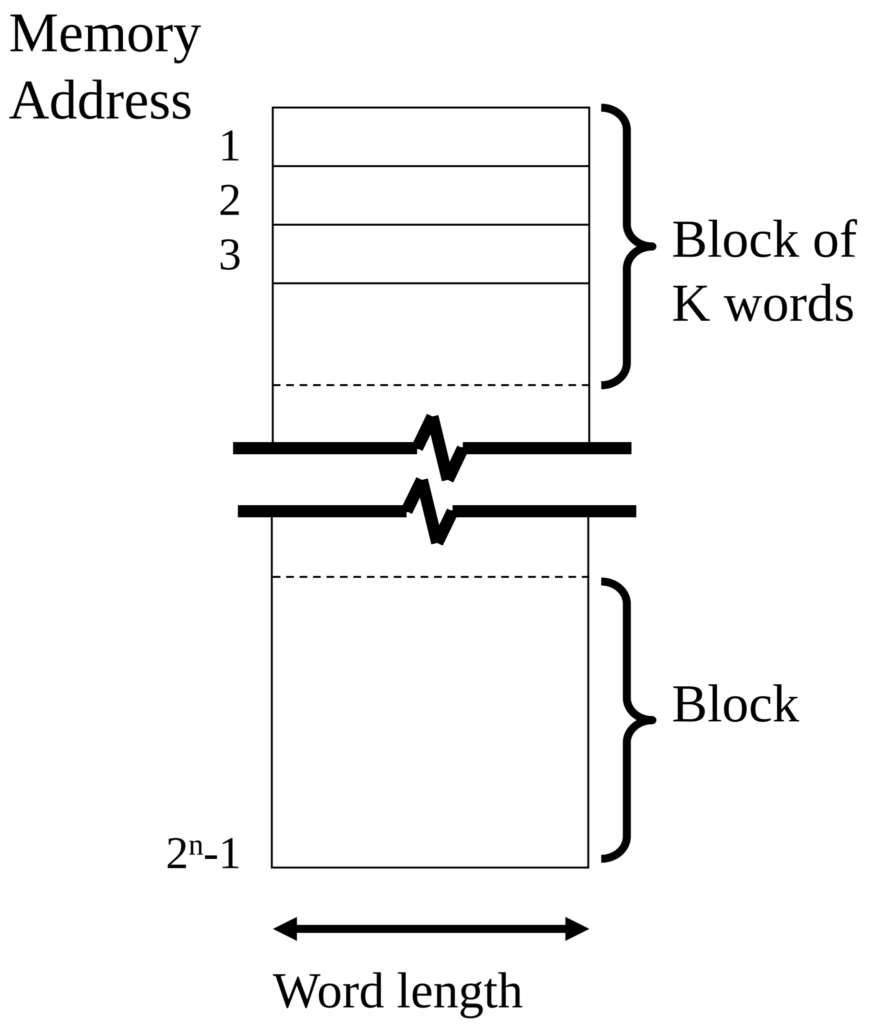
Cache Structure

- Cache is organized as ‘lines’ of k number of consecutive memory locations
- Memory is divided into ‘blocks’ of k number of consecutive memory locations
- ‘tag’ bits for each line of cache;
- tags store info (id) about the corresponding line

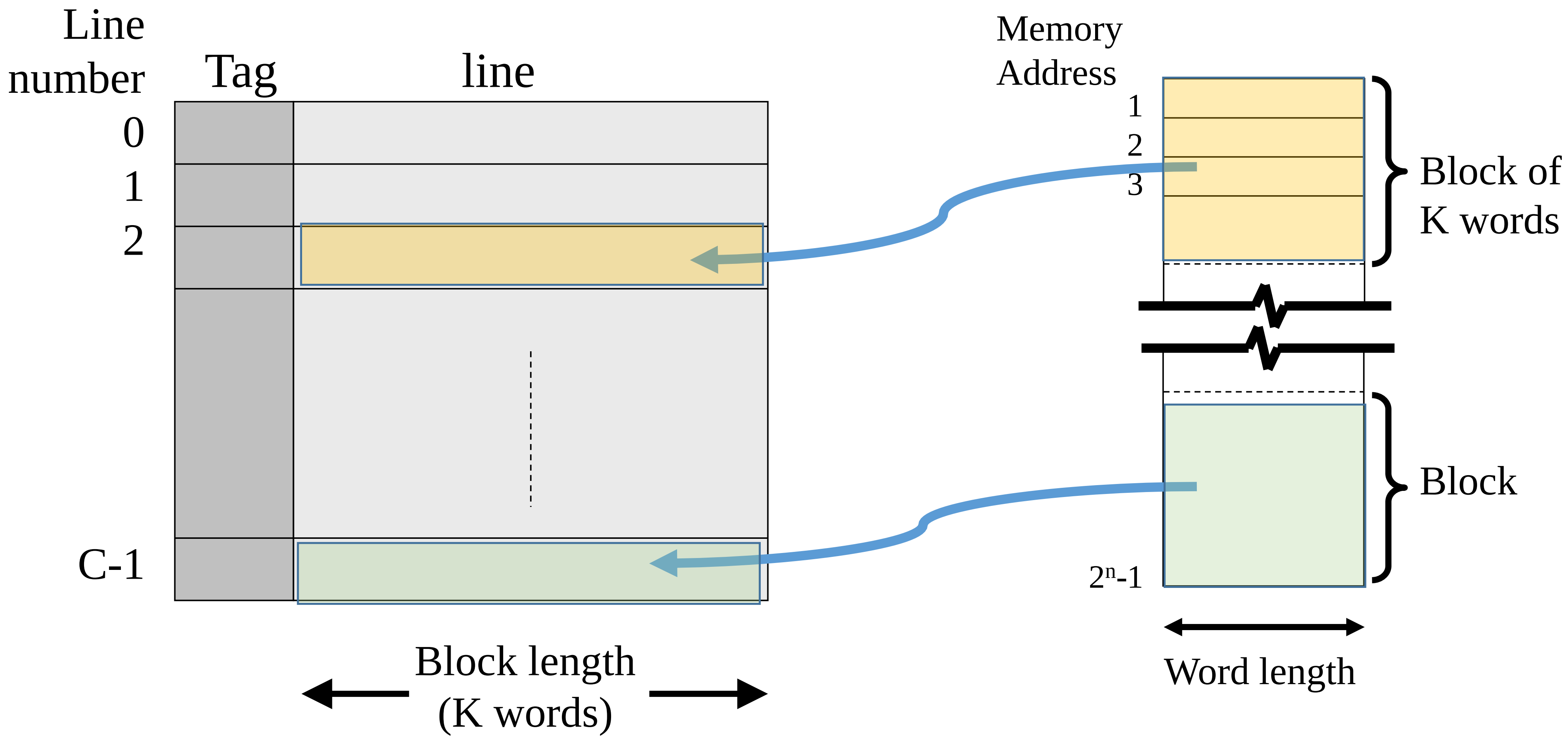
Cache Structure



Memory divided into blocks



Cache Structure



how are these lines and blocks linked and tracked?

Block Placement in cache line - Mapping

Where can a block be placed in cache?

We have 3 options as below.

1. Anywhere in cache - fully associative (or associative mapped)
2. In one predetermined place - direct-mapped
3. In a limited set of places - set-associative mapped
   - Hybrid of direct mapped and fully associative

Block Placement in cache line - Mapping

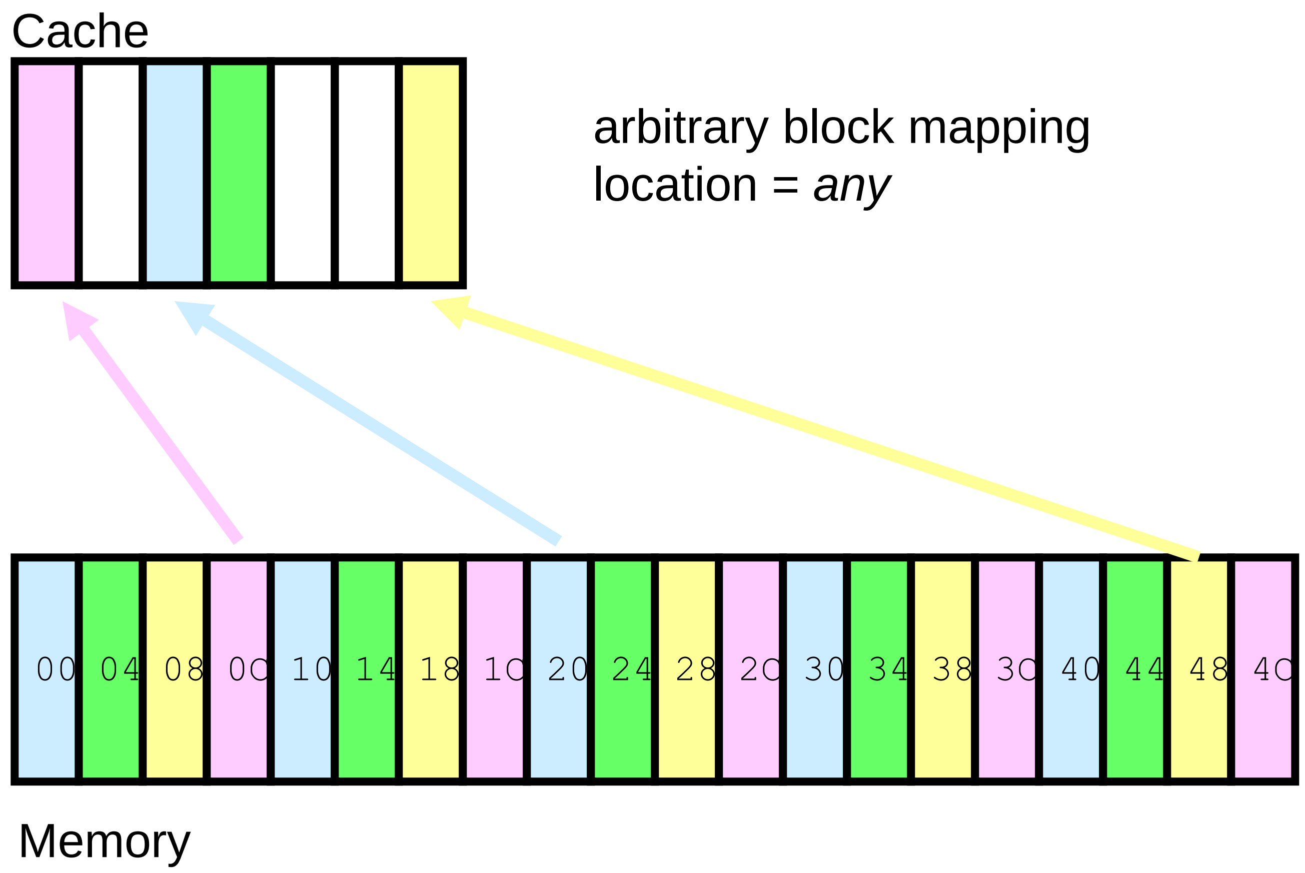
- Where can block be placed in cache?
  1. Anywhere in cache - fully associative
     - Compare tag to every block in cache
  2. In one predetermined place - direct-mapped
     - Use part of address to calculate block location in cache
     - Compare cache block with tag to check if block present
  3. In a limited set of places - set-associative
     - Hybrid of direct mapped and fully associative
     - Use portion of address to calculate set (like direct-mapped)
     - Place in any block in the set
     - Compare tag to every block in set

Fully-associative Mapping

- A main memory block can load into any line of cache
- Every line's tag must be examined for a match
- Cache searching gets expensive and slower
- Memory address is interpreted as:
  - Least significant w bits = word position within block
  - Most significant s bits = tag used to identify which block is stored in a particular line of cache

Tag – s bits	Word – w bits

Fully Associative Block Placement



arbitrary block mapping
location = *any*

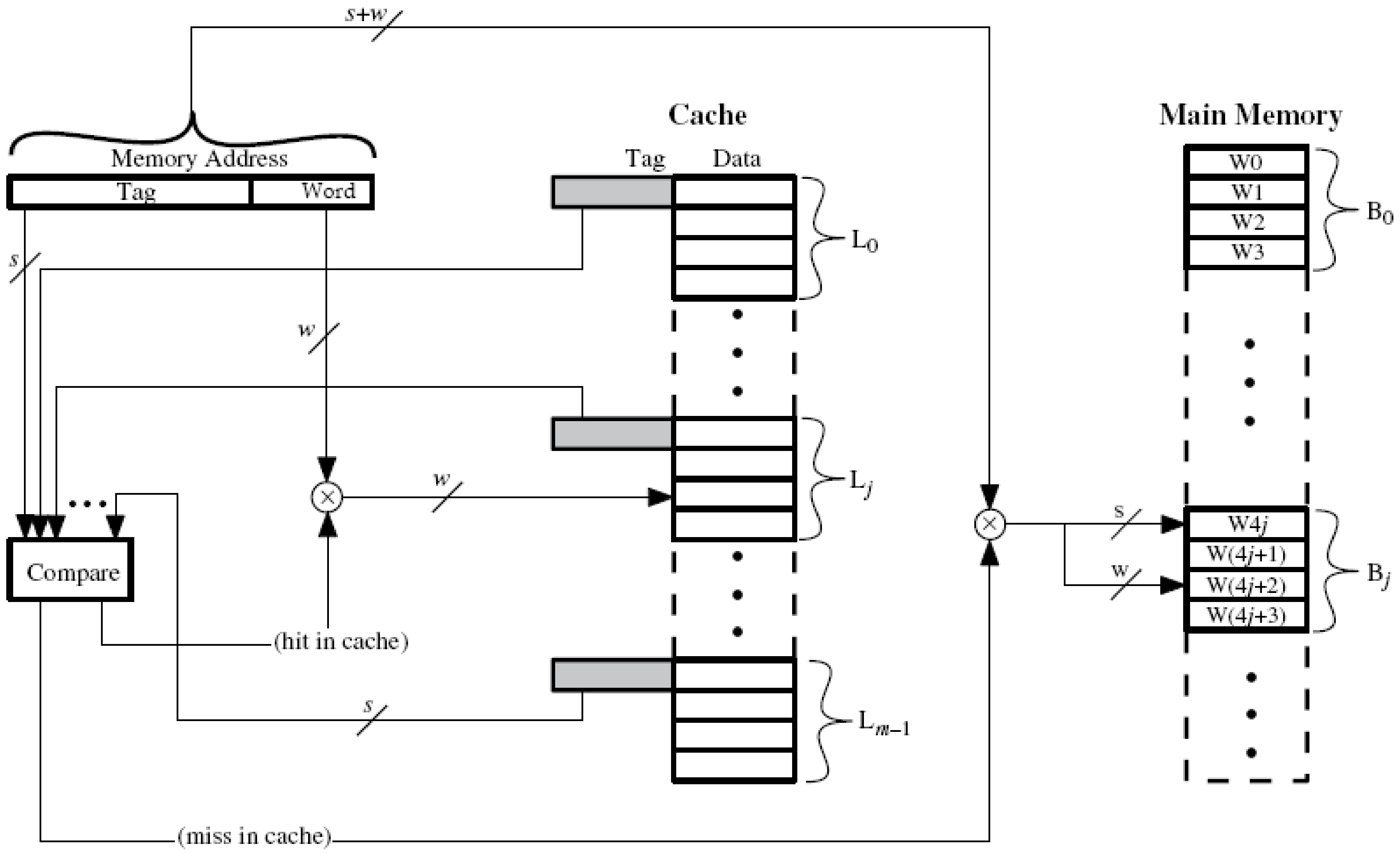
We can see that there are 4 bytes (32 bits) in each block.

Associative Mapping Address Structure Example

Tag – s bits (22 in example)	Word – w bits (2 in ex.)

- Let the address bits be 24 bits (address space?)
- 22 bit tag stored with each 32 bit block of data
- How many blocks are there in this example?
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which of the four 8 bit words is required from 32 bit data block

Fully Associative Cache Organization



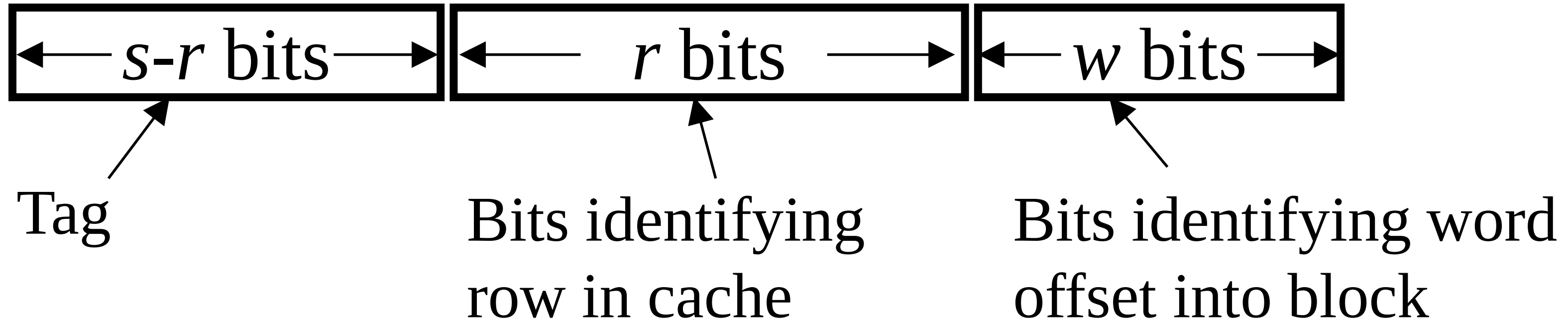
Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable locations = 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 (can be of any capacity)
- Size of tag = s bits
- Low value cache miss is expected

Direct Mapping Address Structure

Each main memory address (*s*+*w* bits) can by divided into three fields

- Least Significant *w* bits identify unique word within a block
- Remaining bits (*s*) specify which block in memory. These are divided into two fields
  - Least significant *r* bits of these *s* bits identifies which line in the cache
  - Most significant *s*-*r* bits uniquely identifies the block within a line of the cache


s-r bits
r bits
w bits
Tag
Bits identifying row in cache
Bits identifying word offset into block

Direct Mapping Cache Line Table

number of lines = m
Number of blocks in main memory = 2^S

(S is the number of bits required to identify a block in main memory.)

Cache line	Main Memory blocks held
0	$0, m, 2m, 3m \ldots 2^s - m$
1	$1, m+1, 2m+1 \ldots 2^s - m+1$
$m-1$	$m-1, 2m-1, 3m-1 \ldots 2^s - 1$

Direct Mapping

- Each block of main memory maps to only one cache line – i.e. if a block is in cache, it will always be found in the same place.
- Line number is calculated using the following function

$$i = j \text{ modulo } m$$

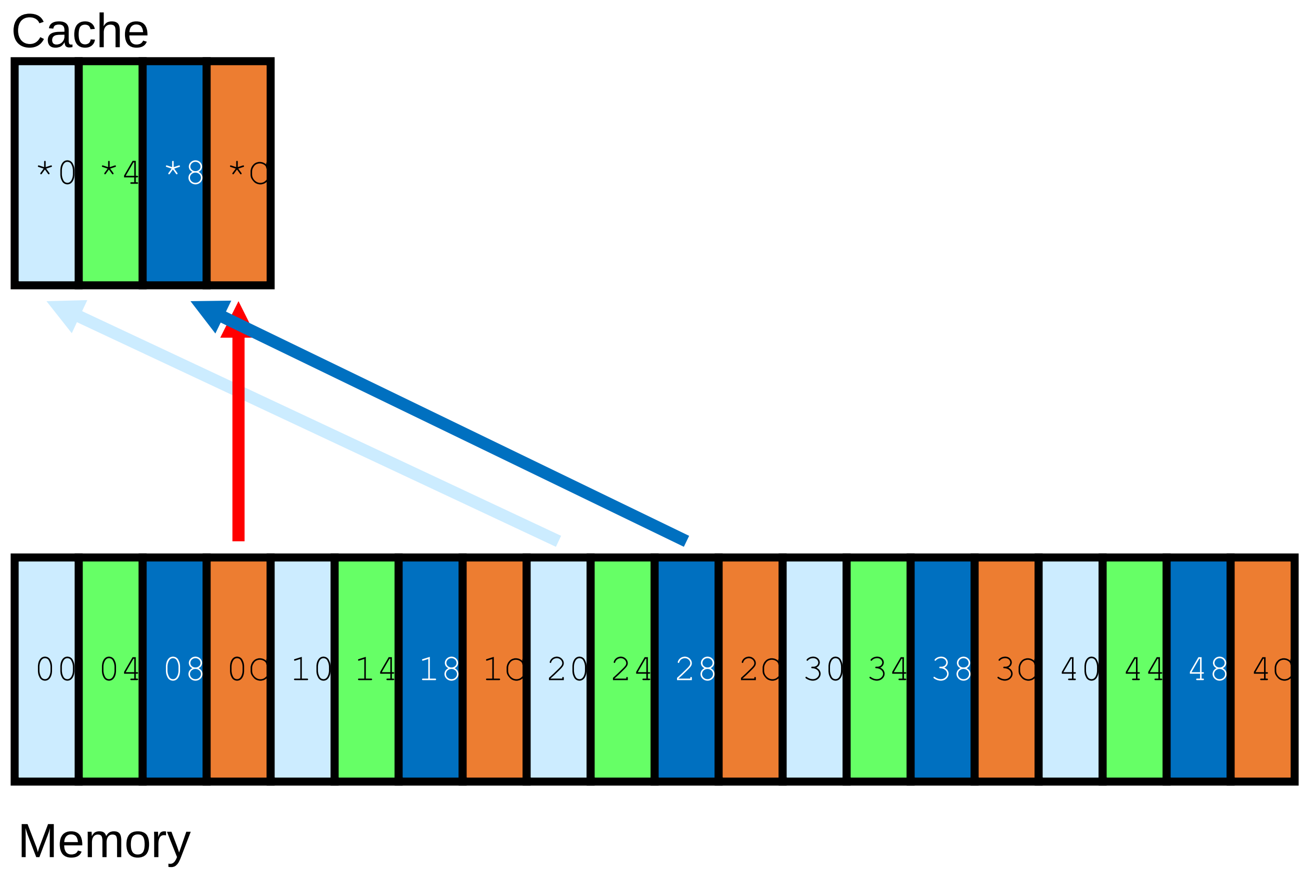
where

i = cache line number

j = main memory block number

m = number of lines in the cache

Direct Mapping illustration



Direct mapping - illustration

Consider, main memory address of 8 bits and block size of 4 bytes

- 256 bytes of main memory
- 64 blocks in main memory

Consider, 8 lines cache, each has a length of 4 bytes

- 32 bytes of cache
- Need 3 bits to identify a line in the cache
- Need 2 bits to identify a byte in a line

tag	line	w
3	3	2

A look at cache

Line number	Tag (3 bits)	Cache line
0	xyz	4 bytes of data in each line
1	pqr	
2	abc	
3		
….		
7		

Block numbers that can sit in each line

Pre determined 8 blocks can sit here

Pre determined 8 blocks can sit here

⋮

Pre determined 8 blocks can sit here

Direct mapping- illustration

tag	line	w
3	3	2

memory address (*s*+*w* bits)
S bits specify which block in memory.
Least significant *r* bits of these *s* bits identifies which line in the cache

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

00h	Block 0, line 0
04h	Block 1, line 1
08h	Block 2, line 2
0Eh	Block 3, line 3
28h	Block 10, line 2
30h	Block 12, line 4
78h	Block 30, line 6
7Ch	Block 31, line 7
FFh	Block 63, line 7

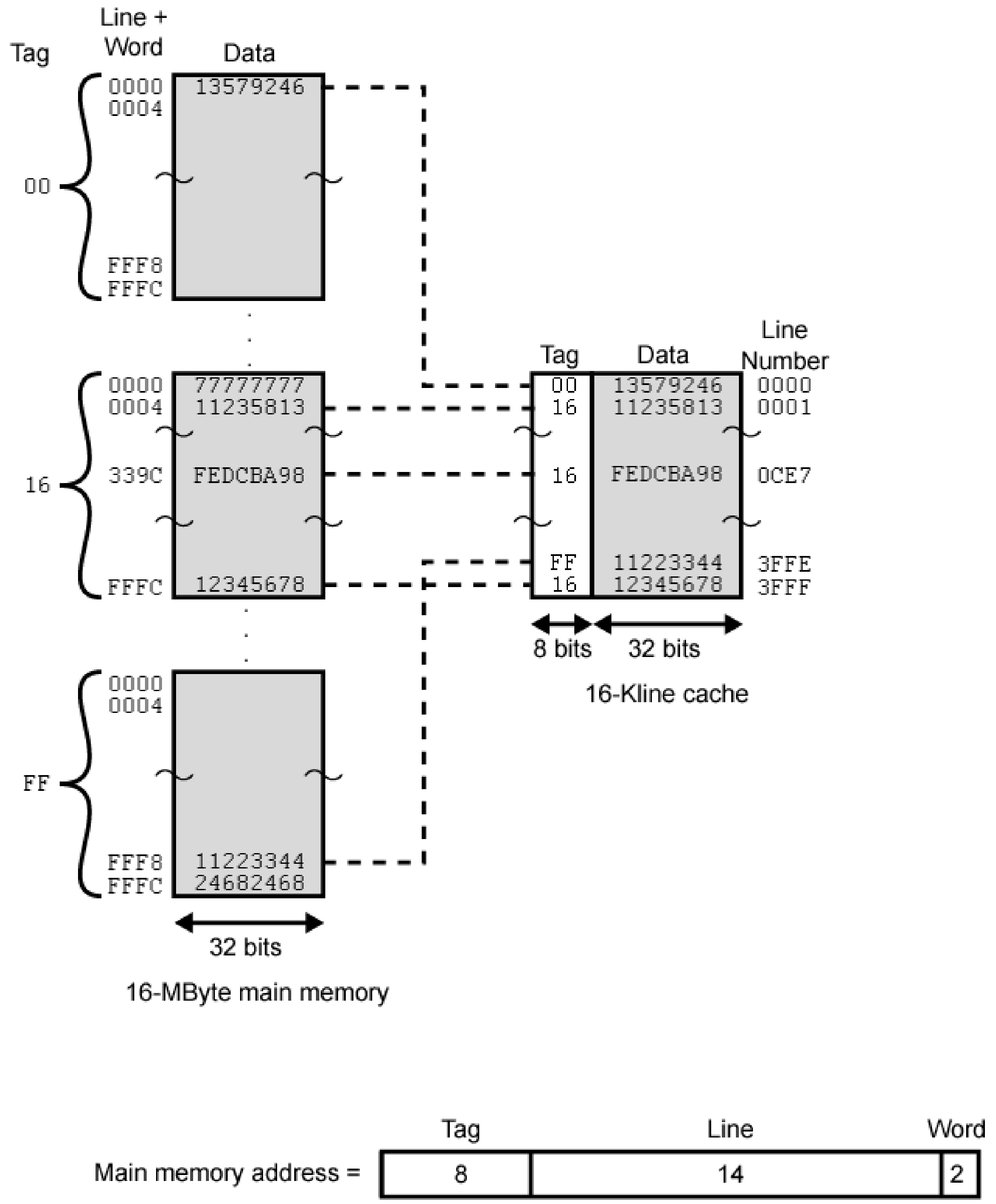
00h
04h
08h
0Eh
28h
30h
78h
7Ch
FFh

Direct Mapping Address Structure Example

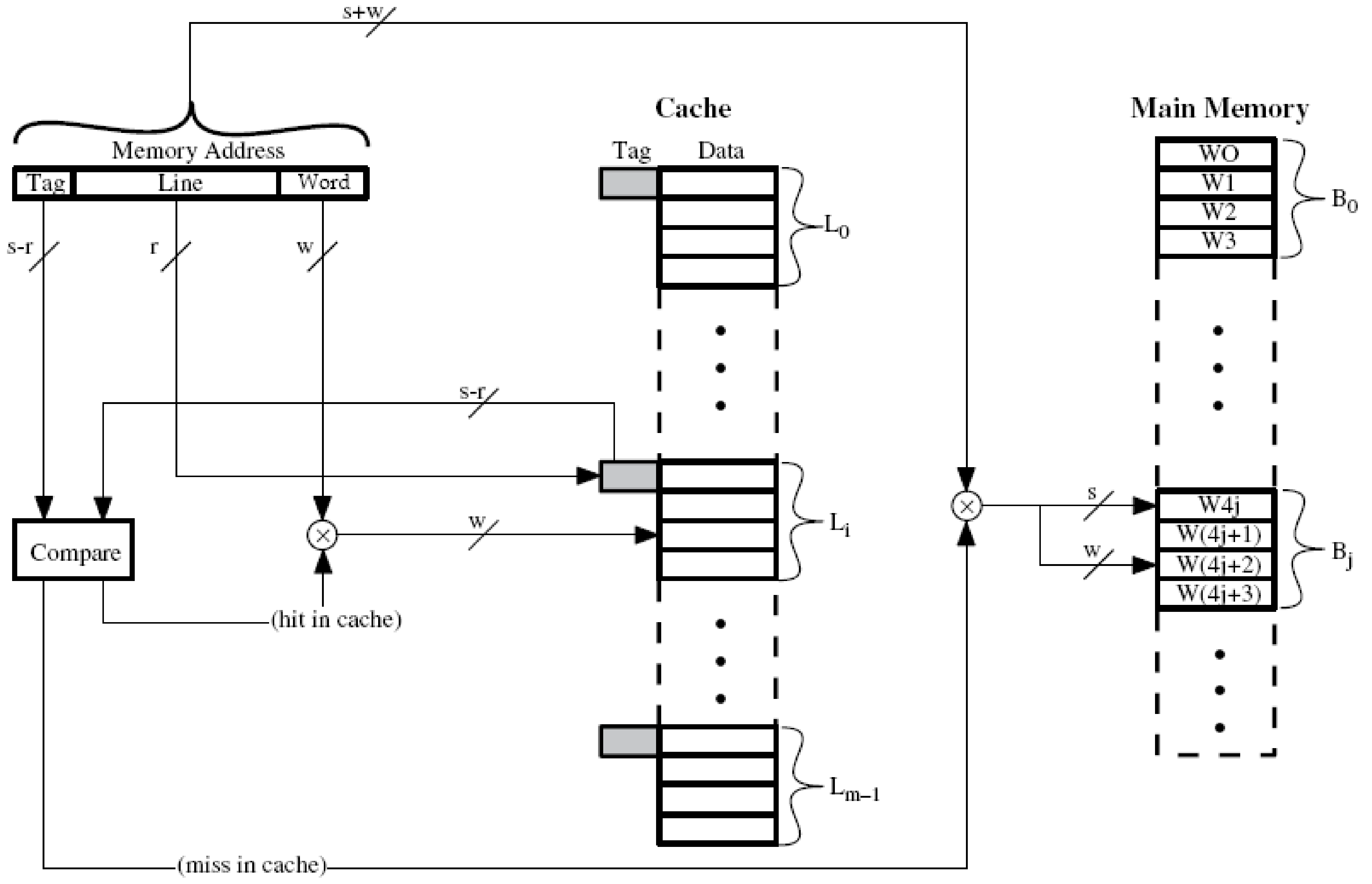
Tag s-r	Line or slot r	Word w
8	14	2

- 24 bit address
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
- 8 bit tag
- 14 bit slot or line
- No two blocks in the same line have the same tag
- Check contents of cache by finding line and comparing tag

Direct mapping



Direct Mapping Cache Organization



Direct Mapping Summary

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

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 (thrashing)

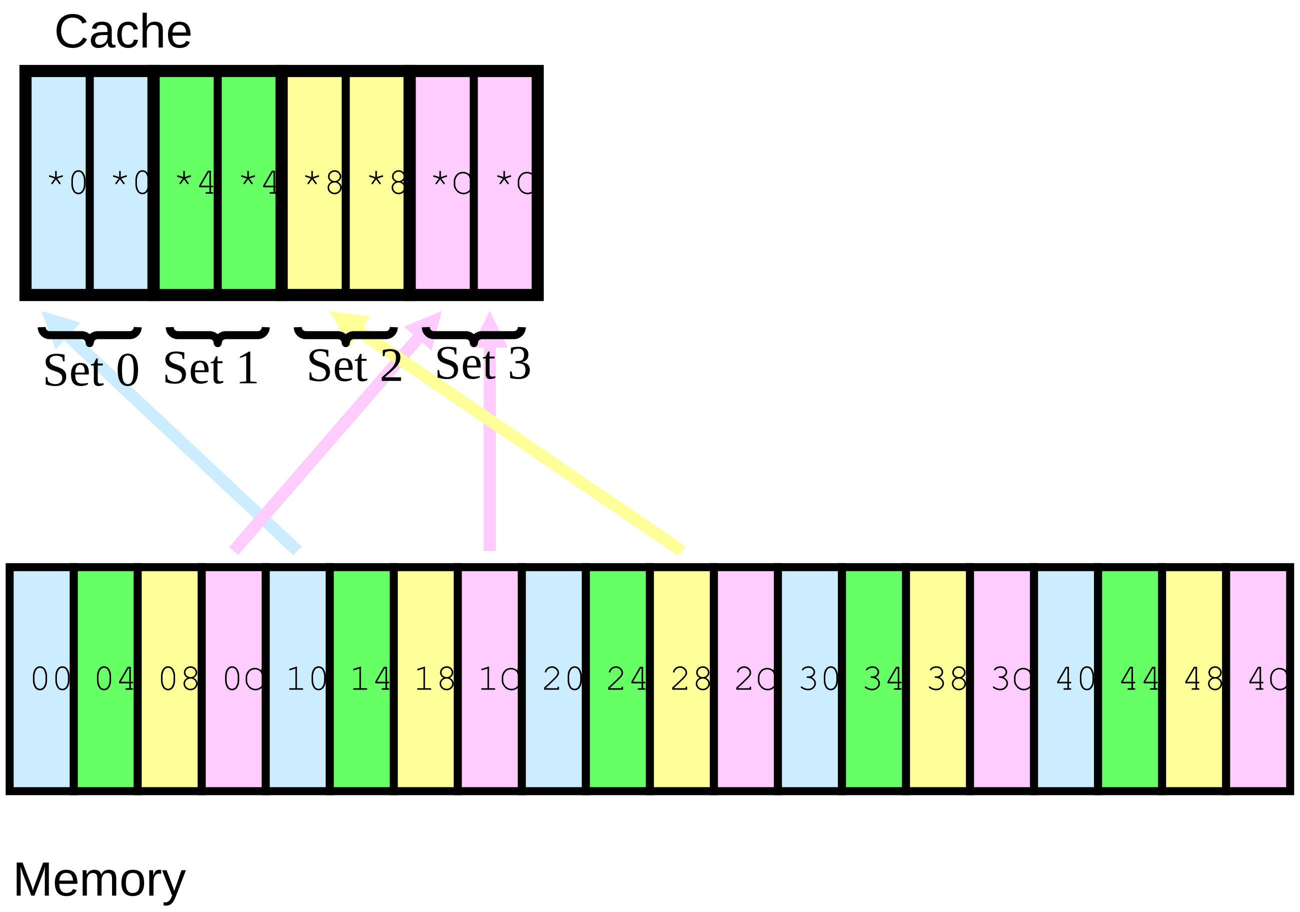
Set Associative Mapping

- Address length is $s + w$ bits
- Cache is divided into a number of sets, $v = 2^d$
- k blocks (or lines) can be contained within each set
- k lines in a cache is called a k-way set associative mapping
- Number of lines in a cache $= v \bullet k = k \bullet 2^d$
- Size of tag = $(s\text{-}d)$ bits
- It is viewed as many set of associative mapping.

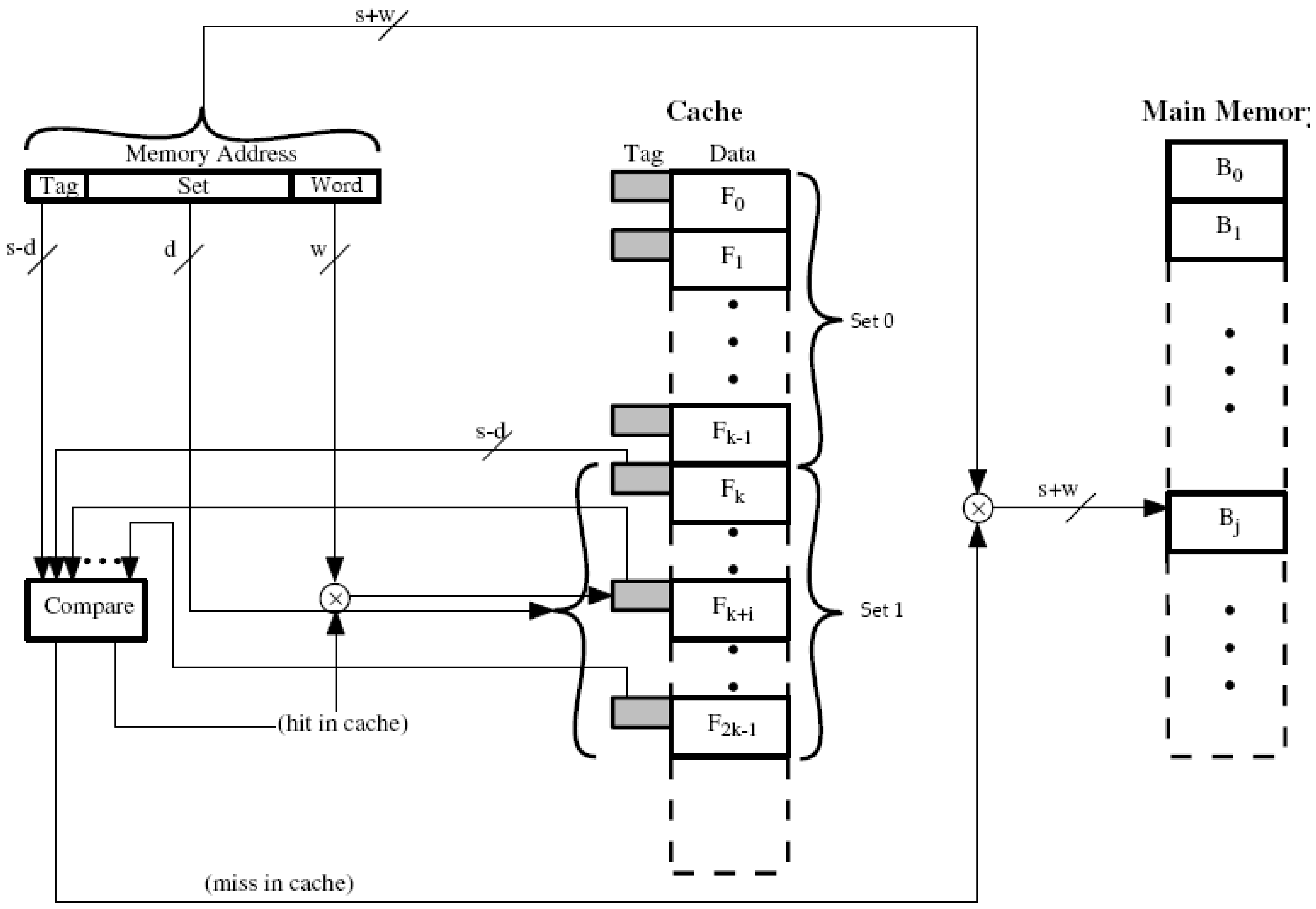
Set Associative Mapping

- Hybrid of Direct and Associative
  k = 1, this is direct mapping
  v = 1, this is associative mapping
- Each set contains a number of lines, basically the total number of lines in cache divided by the number of sets
- A given block maps to any line within its specified set – e.g. Block B can be in any line of set i.
- 2 lines per set is the most common organization.
  - Called 2-way associative mapping
  - A given block can be in one of 2 lines in only one specific set
  - Significant improvement over direct mapping

Set-Associative Block Placement



K-Way Set Associative Cache Organization



Previous direct mapping example- will be used for next mapping illustration

- 24 bit address lines
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
- 8 bit tag
- 14 bit slot or line
- No two blocks in the same line have the same tag
- Check contents of cache by finding line and comparing tag

2 way set associative mapping

- Divides the 16K lines into 2 sets of 8k with 2 lines in each set
- This requires a 13 bit set number
- With 2 word bits, this leaves 9 bits for the tag
- Blocks beginning with the addresses 000000_{16}, 008000_{16}, 010000_{16}, 018000_{16}, 020000_{16}, 028000_{16}, etc. map to the same set, Set 0.
- Blocks beginning with the addresses 000004_{16}, 008004_{16}, 010004_{16}, 018004_{16}, 020004_{16}, 028004_{16}, etc. map to the same set, Set 1.

Tag 9 bits	Set 13 bits	Word 2 bits

Set Associative Mapping Address Structure

Tag 9 bits	Set 13 bits	Word 2 bits

- Note that there is one more bit in the tag than for this same example using direct mapping.
- Therefore, it is 2-way set associative
- Use set field to determine cache set to look into
- Compare tag field to see if we have a hit

Direct Mapping Exercise

What cache line number will the following addresses be stored at? what will the address range of each block they belong to? Consider a cache with 4K lines of 16 words to a block in a 256 Meg memory space. (how many address bits?)

a) $9ABCDEF_{16}$
b) 1234567_{16}

Tag s-r	Line or slot r	Word w
12 bits	12 bits	4 bits

Cache line number: a) CDEh b) 456h

Address range: (a) 9ABCDE0h to 9ABCDEFh (b) 1234560h to 123456Fh

Direct Mapping Exercise

Show the cache memory with tag bits and line number (in binary) which will be occupied by the following addresses? Consider a cache with 4K lines of 16 words to a block in a 256 Meg memory space.

a) $8ABCDAF_{16}$
b) $81A3457_{16}$

How many lines do we have in total?
What is the size of cache in bytes?

Tag s-r	Line or slot r	Word w
12 bits	12 bits	4 bits

If next address is 81A3458H, will this be a hit or miss?

Ans: a) tag bits: 1000 1010 1011 corresponding to line no. 1100 1101 1010
b) tag bits: 1000 0001 1010 corresponding to line no. 0011 0100 0101

Direct Mapping Exercise

Assume that a portion of the cache showing tags and line number looks like the table below. Which of the following addresses are contained in the cache?

a) $438EE8_{16}$ b) $F18EFF_{16}$ c) $6B8EF3_{16}$ d) $AD8EF3_{16}$

miss hit miss hit

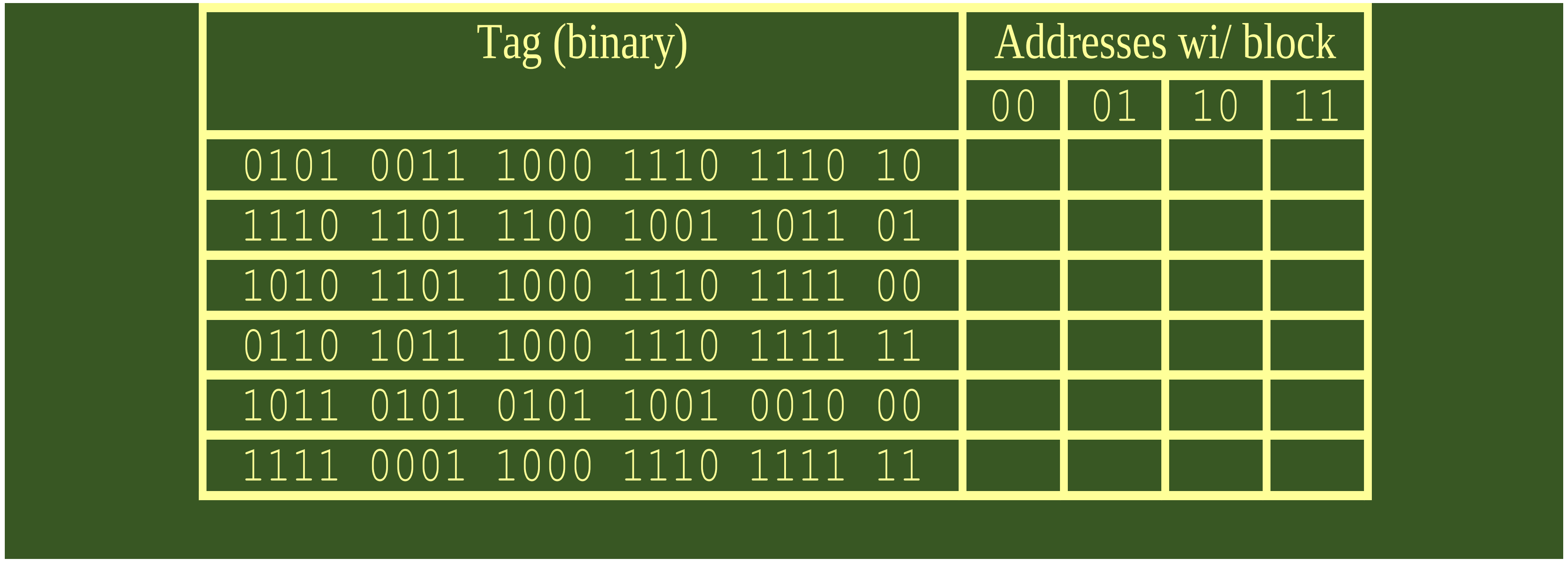
Tag (binary)	Line number (binary)	Addresses wi/ block			
		00	01	10	11
0101 0011	1000 1110 1110 10				
1110 1101	1000 1110 1110 11				
1010 1101	1000 1110 1111 00				
0110 1011	1000 1110 1111 01				
1011 0101	1000 1110 1111 10				
1111 0001	1000 1110 1111 11				

Fully Associative Mapping Exercise

Assume that a portion of the cache with the tag values looks like the table below. Which of the following addresses are contained in the cache?

a) $438EE8_{16}$ b) $F18EFF_{16}$ c) $6B8EF3_{16}$ d) $AD8EF3_{16}$

miss hit miss hit



Tag (binary)	Addresses wi/ block			
	00	01	10	11
0101 0011 1000 1110 1110 10				
1110 1101 1100 1001 1011 01				
1010 1101 1000 1110 1111 00				
0110 1011 1000 1110 1111 11				
1011 0101 0101 1001 0010 00				
1111 0001 1000 1110 1111 11				

Set Associative Mapping Exercise

For each of the following addresses, answer the following questions based on a 2-way set associative cache with a total of 4K lines, each line containing 16 words, with the main memory of size 256 Meg memory space:

a) What is the cache set number the identified block be associated with?

b) What will the tag be?

c) What will the address range of the block in which the given address be located?

1. $9ABCDEF_{16}$
2. 1234567_{16}

Tag s-r	Set s	Word w
13	11	4

Ans.: for 9ABCDEFH

set number is : 10011011110

tag: 1001101010111

9ABCDE0H to 9ABCDEFH

For 1234567H

set number is : 10001010110

tag: 0001001000110

1234560H to 123456FH

RISC and CISC

Speed v/s Space

Introduction

- CISCs were the first kind of processors.
- Innovations in RISC architectures are based on a close analysis of a large set of widely used programs.
- One of the main concerns of RISC designers was to maximize the efficiency of pipelining.
- Both RISCs and CISCs try to solve the same problem. CISCs are going the traditional way of implementing more and more complex instructions. RISCs try to simplify the instruction set.

CISC

- In the 1970’s, memory was expensive and small in size, so people designed computers that would pack as many action as possible in a single instruction this saved memory space, but added complexity.
- Example: adding 2 numbers which are in memory and storing the result back to memory. Something like, a = a + b;
- CISC – Complex Instruction Set Computing
- RISC – Reduced Instruction Set Computing
  - Less Number of instructions
  - Less Variety of addressing modes
  - Less Size of each instruction
  - More Number of registers
- BCD add instruction is a CISC if it does both add and correct it in one instruction.

CISC features or characteristics

- Microprogrammed control unit
- Large number of instructions (200-500)
- Instructions can do more than 1 thing (that is, an instruction could carry out 2 or more actions)
- Many addressing modes
- Instructions vary in length and format
- This was the typical form of architecture until the mid 1980s, RISC has become more popular since then although most architectures remain CISC

RISC

- Hardwired control unit
- Instruction lengths fixed (usually 32 bits long)
- Instruction set size is limited (80-100 instructions)
- Instructions rely mostly on registers; register to register operations. Memory accessed only on loads and stores
- Few addressing modes
- Easy to pipeline for great speedup in performance
- CPU takes less silicon area to implement and remaining area can be used for providing more number of registers.

RISC

Pure RISC machines have the following features

1. **All RISC instruction codes have the same number of bits** (Typically 32 bit)
   - ❑CISC instructions can vary and have no fixed length.
   - ❑Because **RISC** have fixed instruction code, this makes it a lot easer to be pipelined, thus making it faster

2. **The RISC instruction set includes only very simple operations that can ideally be executed in a small number of clock cycles.**
   - ❑These instructions can then be moved through a pipeline quickly and not hold the pipeline up
   - ❑for **BCD** add instruction, we should make add and correction two instructions to make it **RISC** machine and have it easily pipelined

RISC

3. **RISC instructions for reading data from memory include only a single operand load instruction and a single operand store instruction.**
   - ❑Because of this **RISC** machines are referred to as “**load and store**” machines
   - ❑Most **CISC** machines have a single instruction to load the operand from memory and then add operand to a register, note that we doing two things with a single instruction. Just like in **BCD** add instruction
   - ❑Two advantages of a simple load and store
     - ❑machine code only takes a few bits
     - ❑machine code can be quickly decoded, which makes pipelines easier to design

RISC

4. **The RISC instruction set is designed so that compilers can efficiently translate high-level language constructs to instruction codes for the machine.**
   - ❑Compiler writers took little advantage of powerful **CISC** instructions because they were very machine specific and they are not very portable this ended up having compilers that acted like they are working for a **RISC** machine even though they were working for **CISC** machine
   - ❑A problem with pure **RISC** is that it takes many small instructions to do anything, which uses a lot of memory; **this excessive use of memory is called as "code bloat"**
   - ❑A pure RISC machine also requires a greater memory bandwidth because we are constantly doing lot of fetch, load and store operations.
   - ❑ microprocessors are usually designed using both **RISC** and **CISC** approach, **(a combination of both), in order to get best of both worlds.**

Arguments for CISC

- A rich instruction set should simplify the compiler design by having instructions which match the HLL statements.
- Many powerful instructions are supported, making the assembly language programmer’s job much easier.
- Generally micro-coded CU, and hence easy for updating and adding new instructions in next generation processors. Easy to have backward compatibility of the instructions (processor).

Comparing RISC and CISC.

CISC

- Large number of instructions- 150 to 300. (more can be counted by considering variety of addressing modes)
- Employs a variety of data types and a large number of addressing modes.
- Variable length instruction formats.
- Instructions can manipulate operands in memory.

RISC

- Fewer number of instructions. Less than 100.
- Supports less variety of addressing modes.
- Fixed length instructions. 32 bits usually. Easy to decode instruction formats.
- Mostly register-register operations. Memory access is by LOAD and STORE instructions.

Comparing RISC and CISC.

CISC

- Number of cycles per instruction (CPI) varies from 1 to 20 depending on the instruction.
- 8 to 32 GPRs and no support is available for parameter passing and function calls.
- Micro-programmed control unit.
- Complexity is in hardware.
- Primary goal is to complete a task in as few lines of assembly as possible.

RISC

- Number of CPI is 1 as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and uniform instruction format.
- 64 to 512 GPRs are available that are helpful for parameter passing in function calls.
- Hardwired control unit.
- Complexity is in software.
- Primary goal is to speedup individual instruction.

The Performance Equation

The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \underset{\textcircled{1}}{\frac{\text{cycles}}{\text{instruction}}} \times \underset{\textcircled{2}}{\frac{\text{instructions}}{\text{program}}}$$

The CISC approach

- minimizes the number of instructions per program (2)
- sacrificing the number of cycles per instruction. (1)

RISC does the opposite

- reduces the cycles per instruction (1)
- sacrificing number of instructions per program (2)

Virtual memory

1. Accommodate multi-programming with limited RAM!!!
2. Give all programs unlimited memory space !!!

A few questions to think about

- can a program written for 8051, run on 8085 or 8086 or PowerPC or any other processor? (your question should be??)
- What about the memory size of the program; Can it fit in the given address space?
- should I think about the available RAM capacity on the target system before I write a program? (available RAM can be less than the max capacity)
- can I run multiple programs on the same processor?
  - Since program must be loaded onto RAM for it to run, what about the memory requirement of each of the programs; how can I serve all the programs with the available RAM (fixed capacity)?
  - will the programs share the available RAM?
- can I load my program at any location; is there a specific location demand from the program or am I free to chose any location that is available?
  - What if I use direct addresses for my data addressing and code addressing?
  - Relocatable code

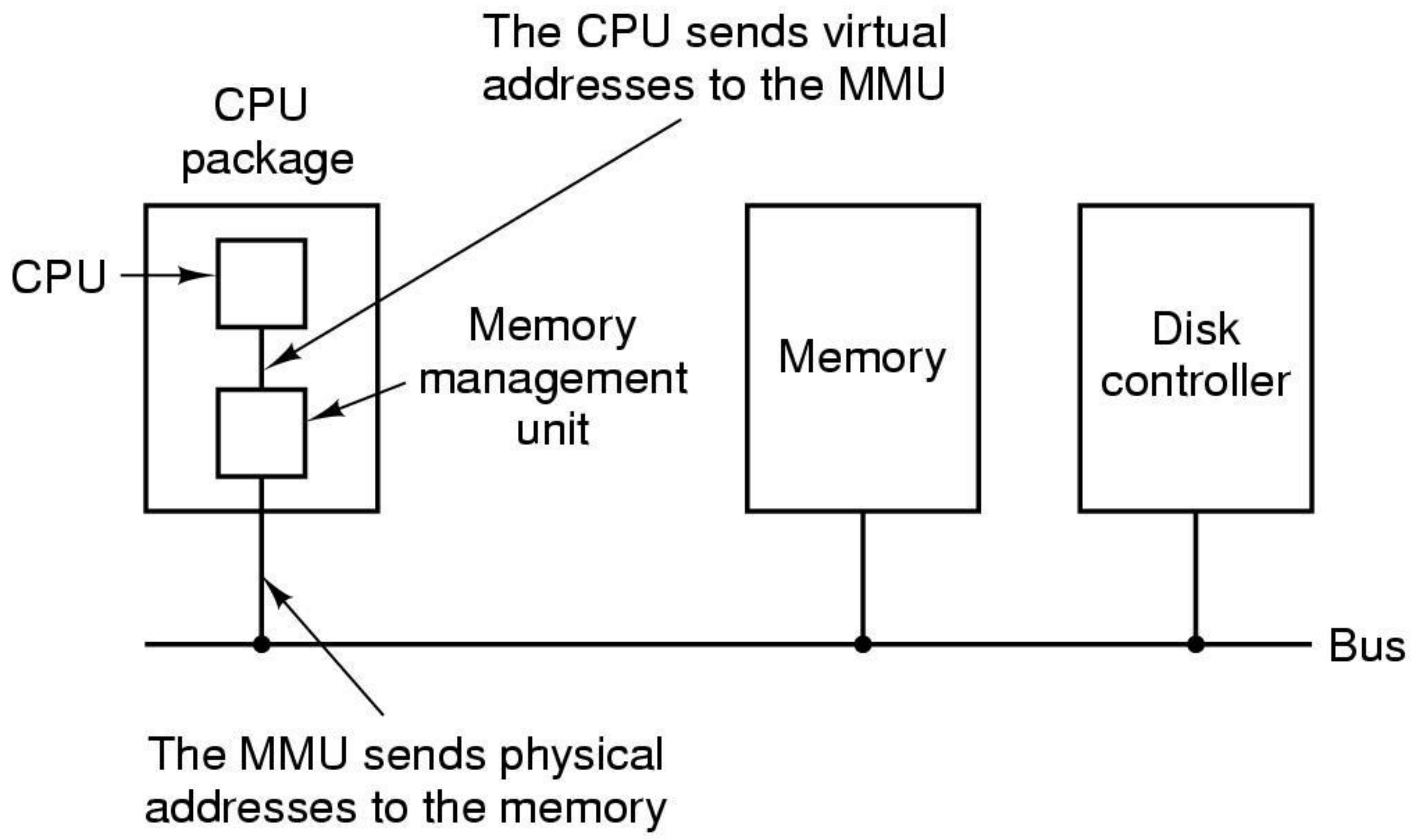
Virtual Memory

- Virtual memory achieves a complete separation of logical and physical address-spaces
- Today, typically a virtual address is 32 bits, this allows a process to have "*virtually*" 4GB of addressable memory
    - Physical memory is much smaller than this, and varies from machine to machine

Terminology in Virtual Memory

- Each process has its own private "virtual address space" (e.g., 2^{32} Bytes); program uses "virtual addresses" for locating data and code
- Each computer has a "physical address space" (e.g., 128 MegaBytes DRAM); also called "real memory"
- Address translation: mapping virtual addresses to physical addresses
  - Allows multiple programs to use (different chunks of physical) main memory at the same time
  - Also allows some chunks of virtual memory to be represented on disk, not in main memory (to exploit memory hierarchy)

Virtual Memory

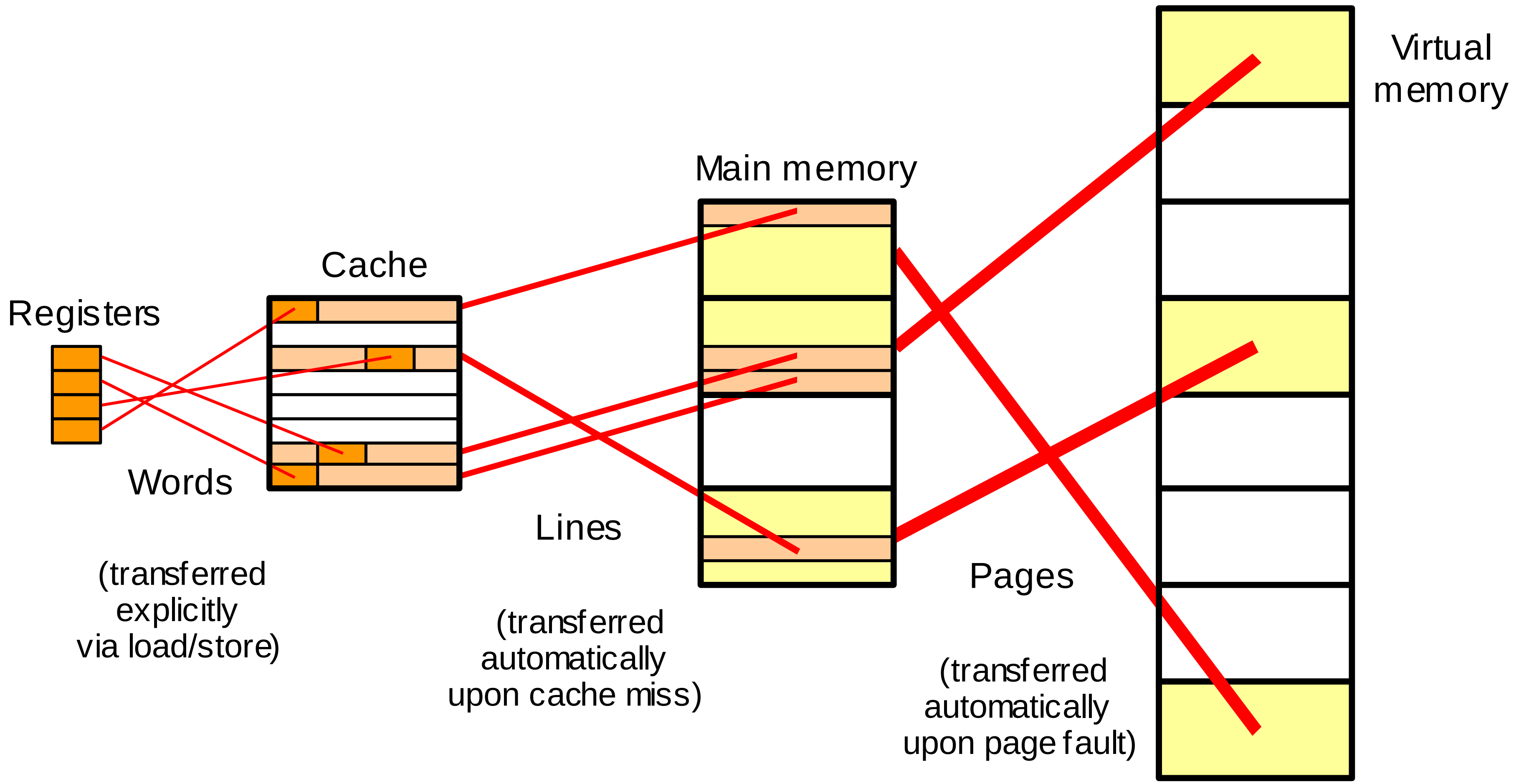

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Position and function of MMU

Virtual Memory

- Uses main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by supporting code relocation (i.e., the code can be loaded anywhere in main memory)
- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
  - During run-time, each virtual address must be translated to a physical address (an address in main memory)

Memory Hierarchy: The Big Picture


Virtual memory
Main memory
Cache
Registers
Words
Lines
Pages
(transferred explicitly via load/store)
(transferred automatically upon cache miss)
(transferred automatically upon page fault)

Data movement in a memory hierarchy.

Virtual Memory

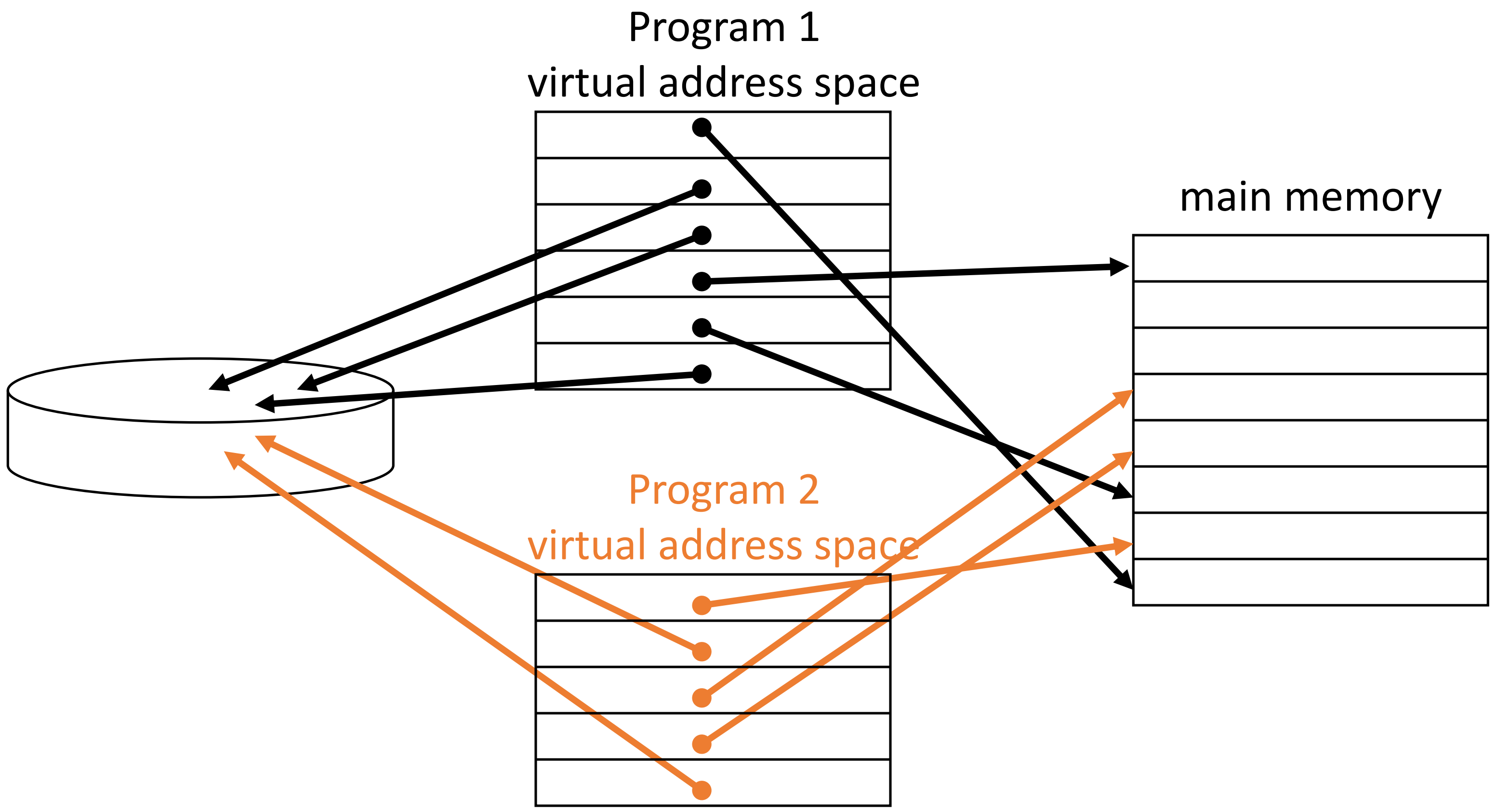
- A process may be broken up into chunks that do not need to be located contiguously in main memory
- No need to load all chunks of a process in main memory
- A process may be swapped in and out of main memory such that it occupies different regions at different points of time
- **<u>Advantages</u>**:
    - More processes can be maintained in main memory
    - With so many processes in main memory, it is very likely that a process will be in the Ready state at any particular time (*CPU efficiency*)
    - A process may be larger than all of main memory

Comparing cache and VM

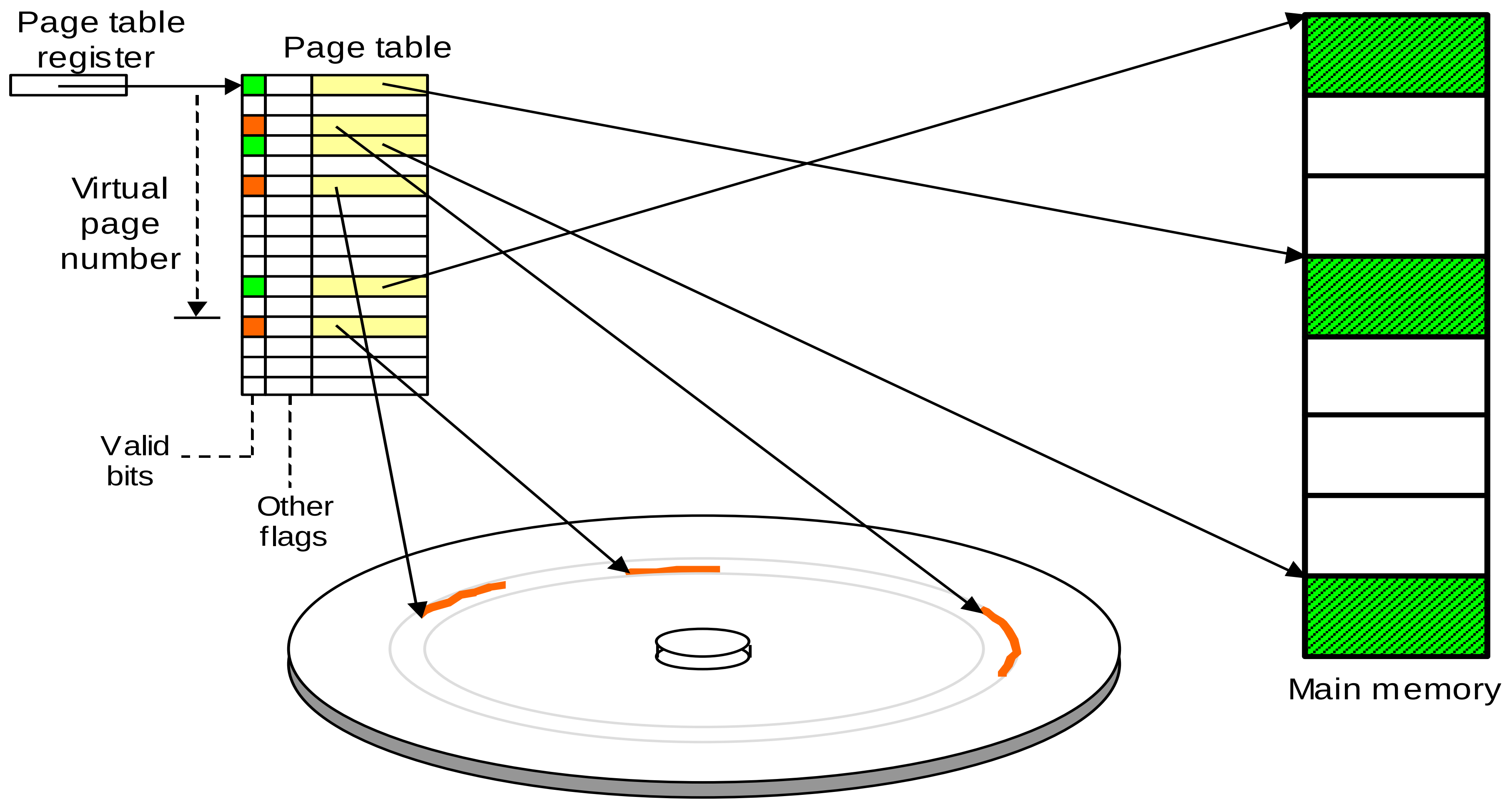
- Caching
  - **To address bottleneck between CPU and Memory**
  - **Direct**
  - **Associative**
  - **Set Associate**
- Virtual Memory Systems
  - **To make programs to think that they have whole of memory that they need**
  - **To address size constraint between Memory & Disk**
  - **Pages**
  - **Translation Lookaside Buffer**

Two Programs Sharing Physical Memory

- A program's address space is divided into pages (page has a fixed size)
- The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table (page table not shown in the diagram below)



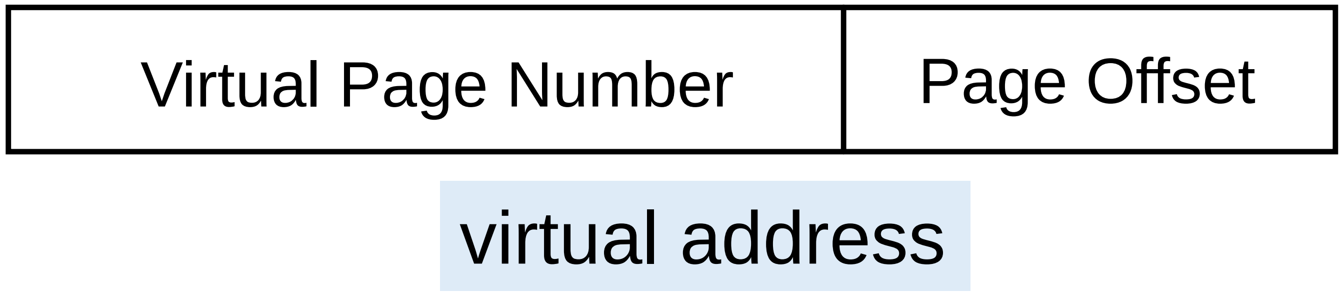
Page Tables and Address Translation



The role of page table in the virtual-to-physical address translation process.

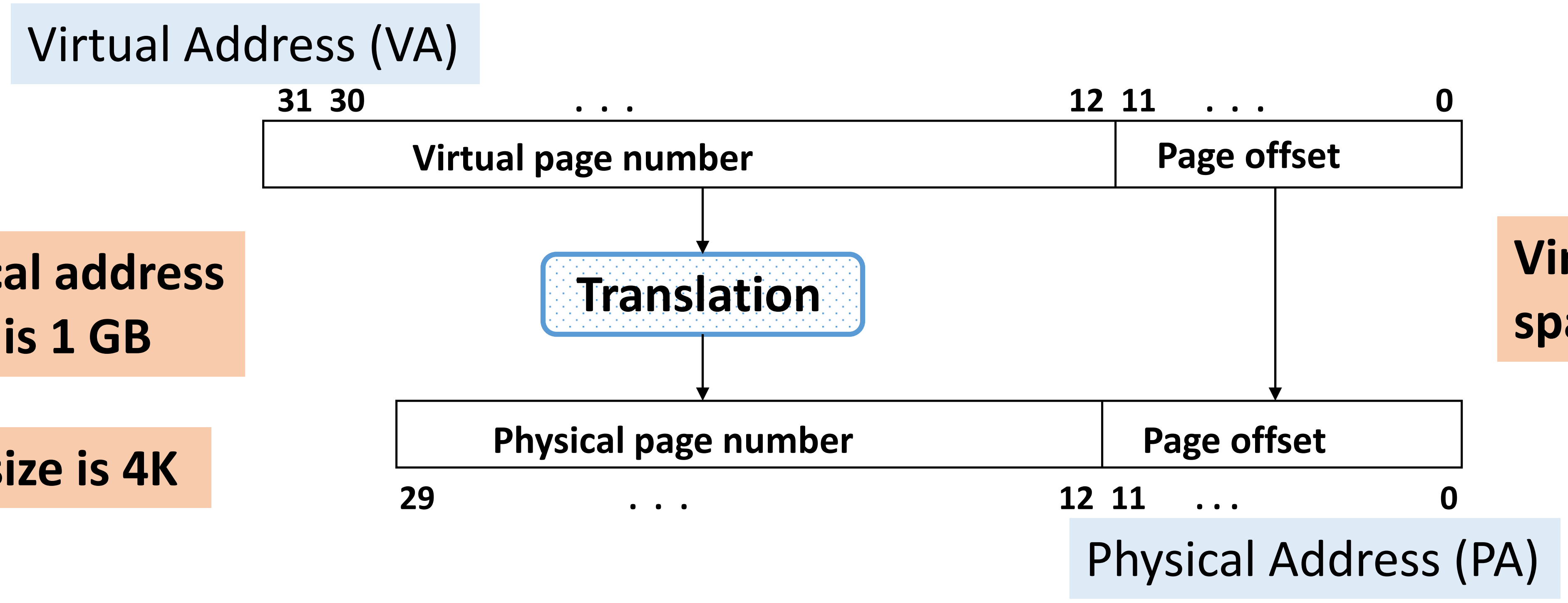
How to Perform Address Translation?

- Virtual Memory divides memory into equal sized pages
- Virtual Memory address is divided into two fields;
  - Virtual page number
  - Page offset
- offsets within the pages do not change



Address Translation

- A virtual address (logical address or programmer's address) is translated to a physical address (main memory address) by a combination of hardware and software



Physical address space is 1 GB

Page size is 4K

Virtual address space is 4 GB

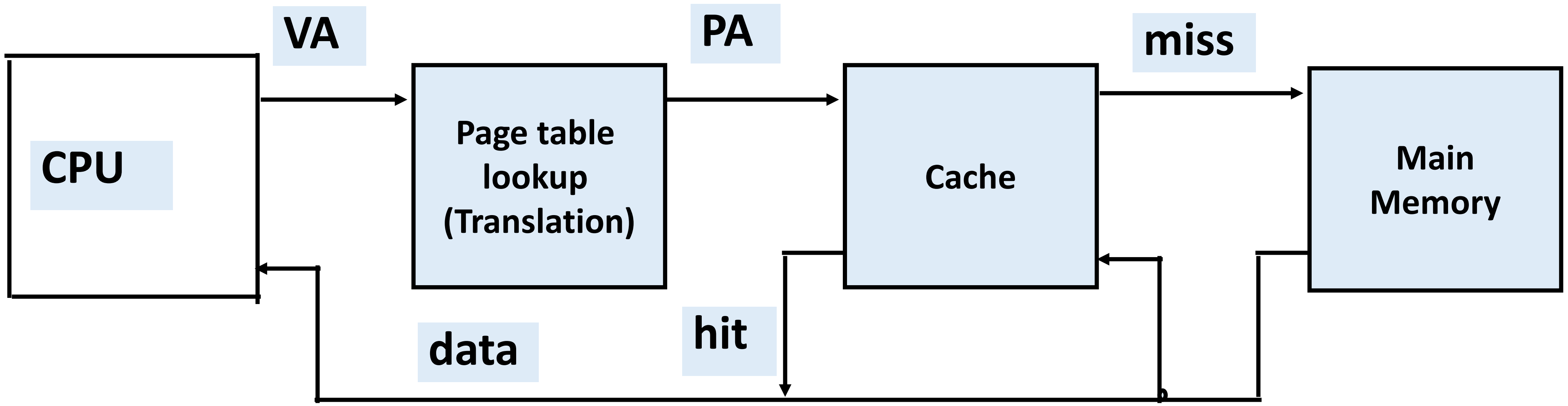
- So each memory request *first* requires an address translation from the virtual space to the physical space
- A virtual memory miss (i.e., when the page is not in main memory) is called a ***page fault***

How to Translate Fast?

- Problem: Virtual Memory requires two memory accesses!
    - Page Table is in physical memory! => 2 main memory accesses!
    - one to translate Virtual Address into Physical Address (page table lookup)
    - one to transfer the actual data (cache hit)

- Observation: since there is locality in pages of data, there must be locality in virtual addresses of those pages!
- Why not create a cache of virtual to physical address translations to make translation fast? (smaller is faster)
- such a "page table cache" is called a Translation Lookaside Buffer, or TLB

Virtual Addressing with a Cache



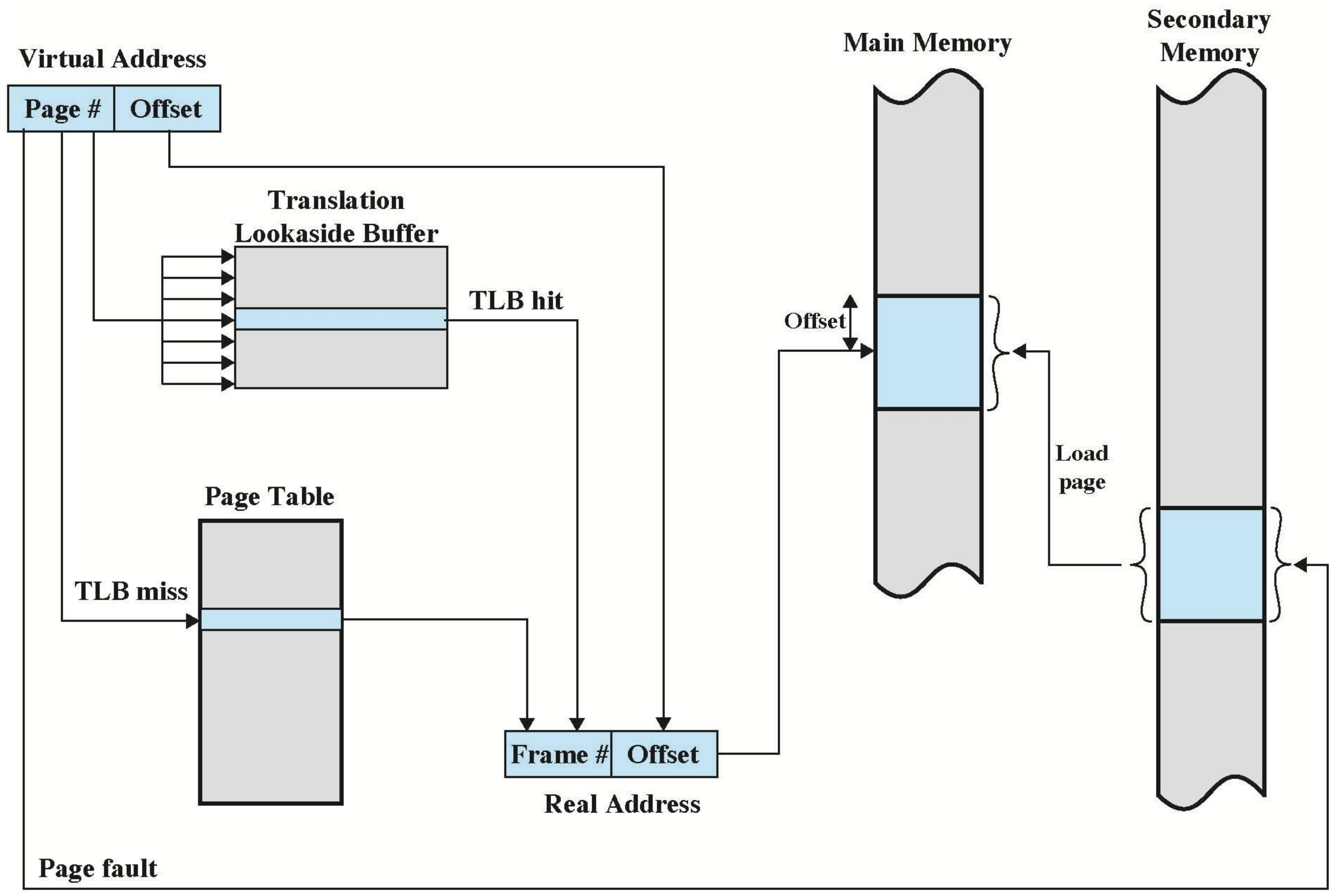
- Translation Lookaside Buffer (TLB) –a small cache that keeps track of recently used address mappings to avoid accessing page table

Translation Lookaside Buffer

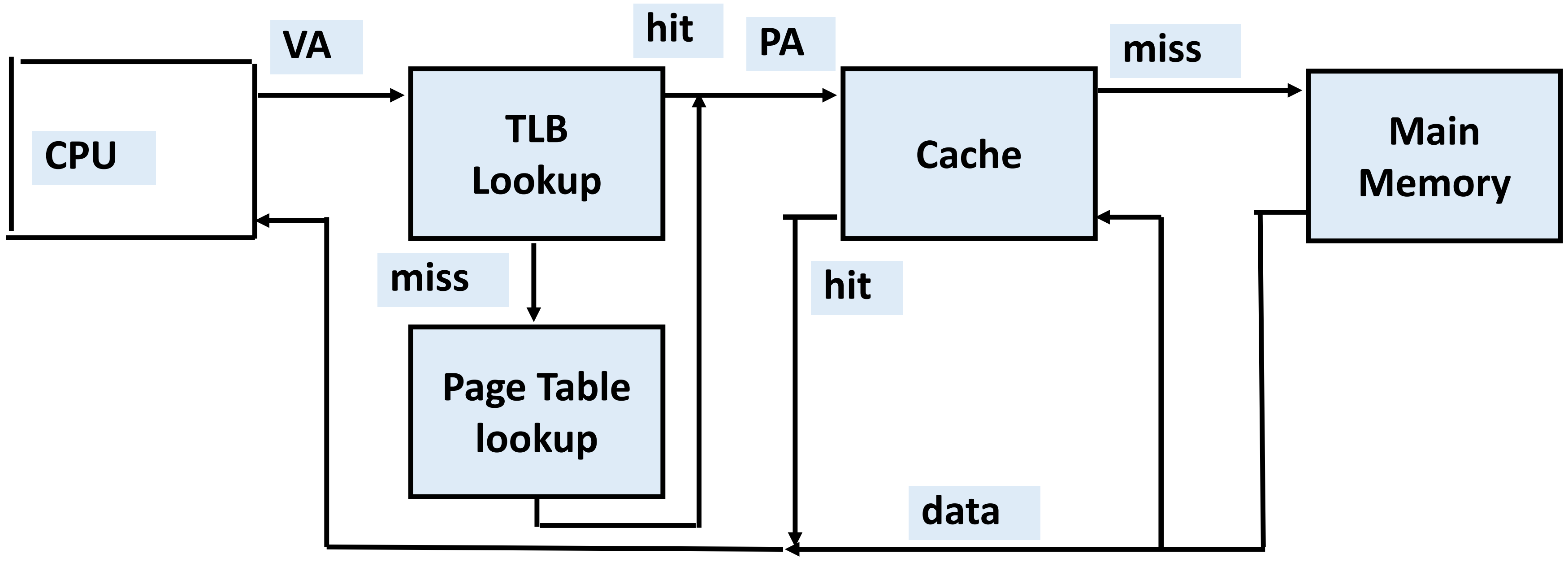
- Functions same way as a memory cache. TLB stores most recently used page table entries. Stores virtual page numbers and the mapping for it. Uses associative mapping (i.e. any virtual page number maps to any TLB index)

- Given a virtual address, processor searches the TLB first.
- If page table entry is present (a ***hit***), the frame number is retrieved and the real address is formed.
- If page table entry is not found in the TLB (a ***miss***), the virtual page number is used to index the page table, and TLB is updated to include the new page entry.

TLB working showing TLB hit, TLB miss and page fault



A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB.
    - Takes 10’s of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it’s a true page fault
    - Takes 1,000,000’s of cycles to service a page fault
- TLB misses are much more frequent than true page faults

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	(Hit) DC	Hit	Yes – this is what we want! although the page table is not checked if the TLB hits
Hit	Hit	Miss	Yes – TLB hits, but data is not in cache; cache miss
Miss	Hit	Hit	Yes – TLB miss, PA is in page table
Miss	Hit	Miss	Yes – TLB miss, PA is in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

I/O organization and I/O communication

How I/O devices are addressed and communicate.

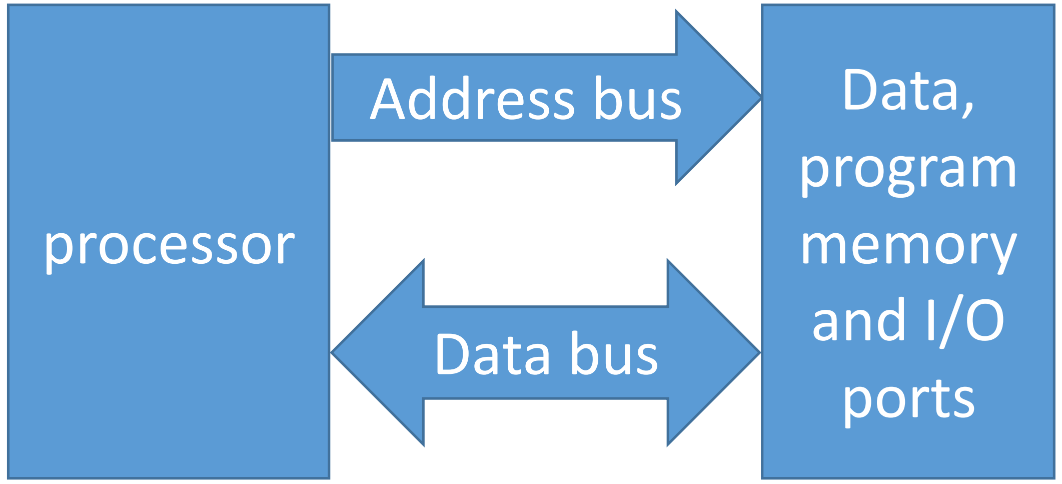
Programmed IO

Interrupt I/O

DMA

I/O mapping

- How to differentiate between I/O address and memory address?
- How 8051 is doing it?
- We have 2 options while designing the processor
  - Shared address space
  - Dedicated address space
  - Each processor supports one of the options



I/O mapping

- Shared address space
  - Same instruction for both memory and I/O access
  - Also called as memory mapped I/O
  - Will have same size address bus for both I/O and memory address

- Dedicated address space
  - Separate instructions for memory and I/O access
  - Also called as I/O mapped I/O or standard I/O
  - Usually will have 8 bit addresses; sufficient

Ports and I/O addresses

- Port is an access point where we connect I/O devices
- Every port will have an address
- I/O device address will be the address of the port that is used for accessing it
    - If we connect keyboard to port-A then keyboard address is the same as port-A's address ; for example 8020H
- Port addresses are decided during design of the hardware system (or the processor) and they remain fixed

Standard I/O vs Memory-Mapped I/O (Examples)

Standard I/O:

- IN yyyy
- OUT yyyy
- yyyy is the address of the I/O port.
- Data will flow between the port address and accumulator.

Memory-Mapped I/O:

- LDA yyyy
- STA yyyy
- yyyy is the address of the I/O port.
- Data will flow between the port address and accumulator.

- MOV Rn, yyyy
- MOV xxxx, Rn

Methods of I/O communication

1. Programmed I/O; Use a program for I/O communication
   I. Unconditional
   II. Conditional

2. Interrupt I/O;

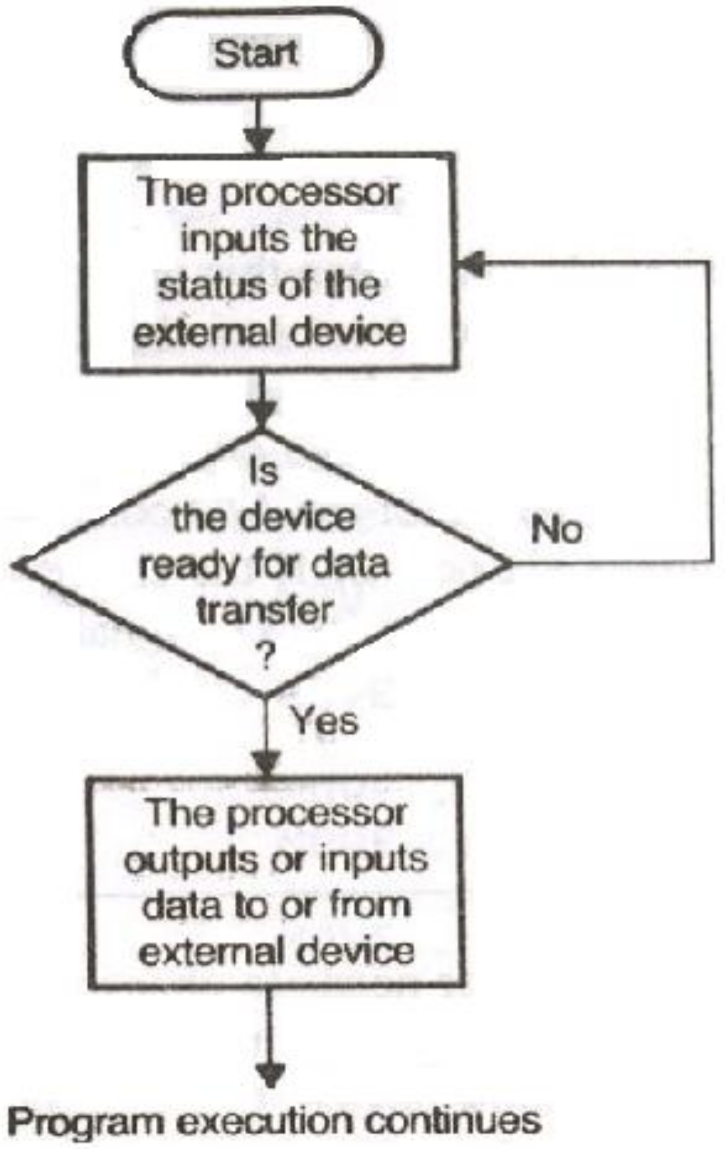
   will it not use a program for I/O communication?

3. DMA; use dedicated hardware for communication between I/O and memory

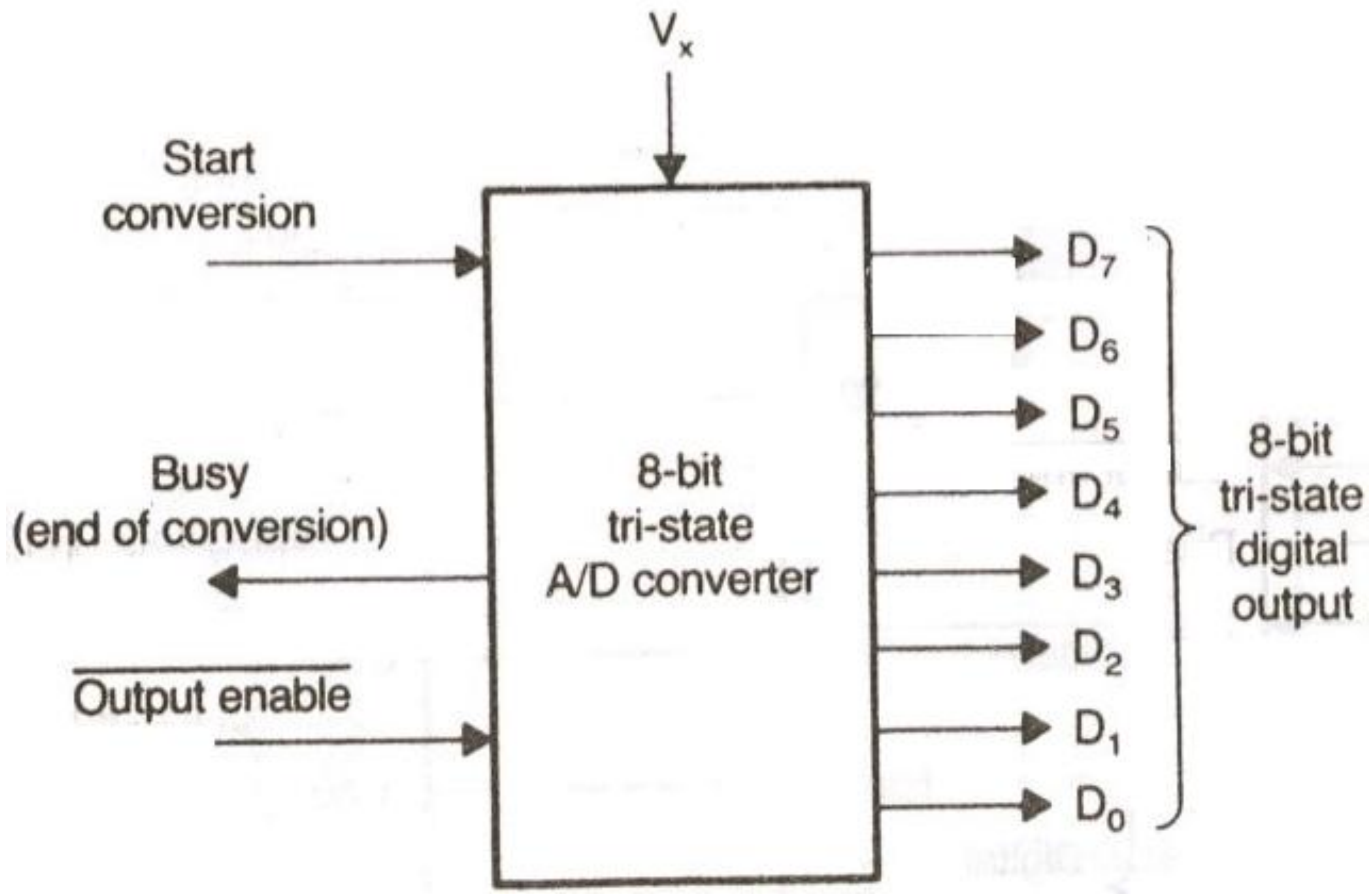
Programmed I/O

- Unconditional I/O
  - Program will not check for any conditions (status of I/O device) before initiating data transfer activity
  - LED switching on/off, DAC etc.
- Conditional I/O
  - Program will initiate data transfers only after status condition is indicating for data transfer activity
  - ADC, Keyboard etc.

Conditional Programmed I/O


Start
The processor inputs the status of the external device
Is the device ready for data transfer ?
No
Yes
The processor outputs or inputs data to or from external device
Program execution continues

Working steps of a basic ADC



V_x
Start conversion
Busy (end of conversion)
$\overline{\text{Output enable}}$
8-bit tri-state A/D converter
D_7
D_6
D_5
D_4
D_3
D_2
D_1
D_0
8-bit tri-state digital output

Example of Conditional Programmed I/O

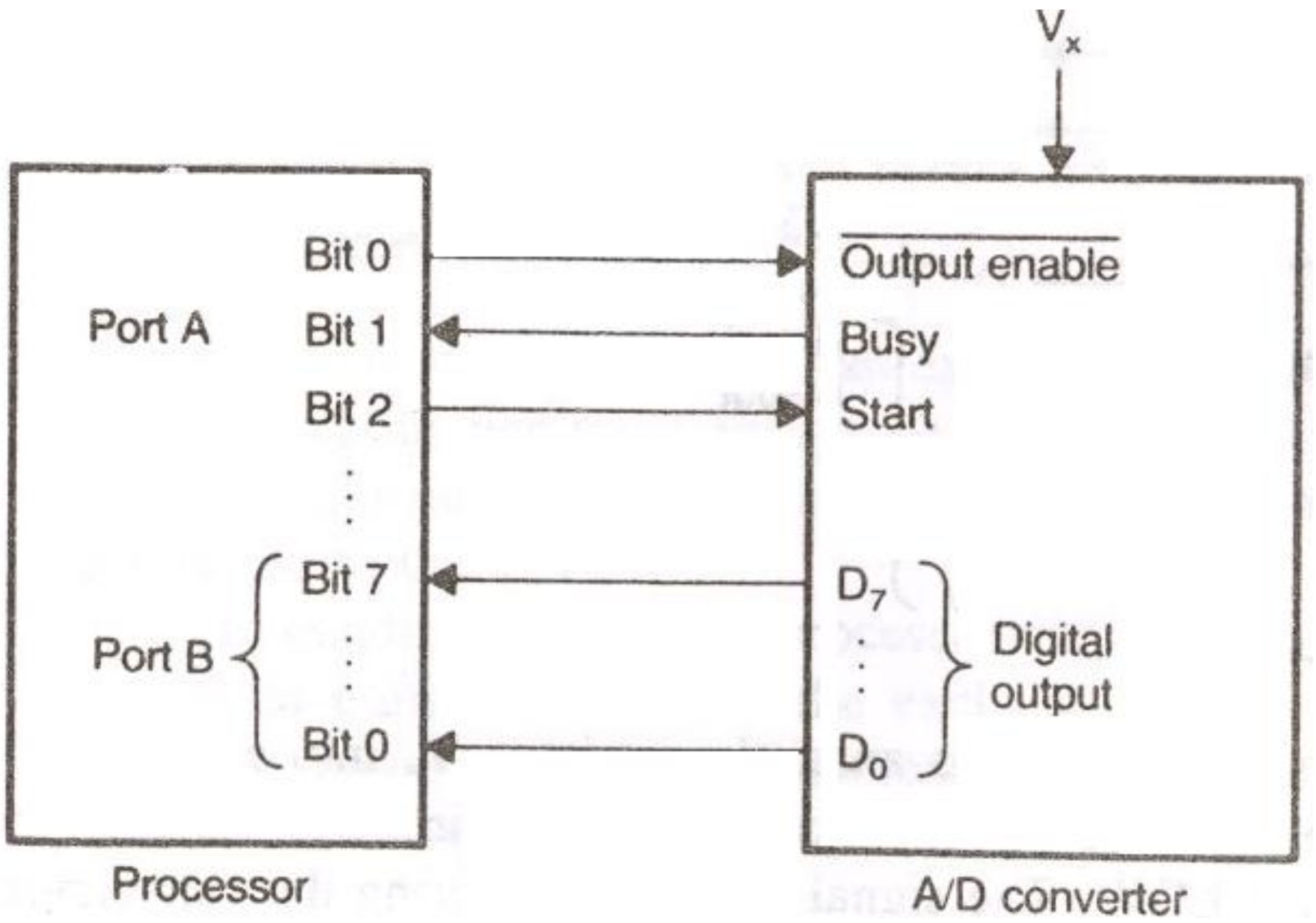
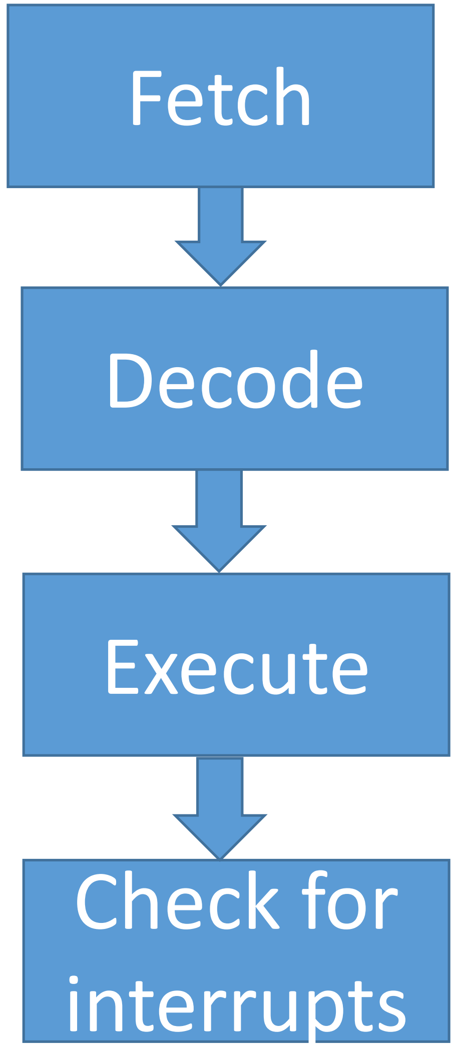


Figure: Interfacing an A/D Converter to an 8-bit Processor

INTERRUPTS

- An *interrupt* is an event (external or internal) that signals the processor to inform that the interrupting device needs its service (‘attention’).
- CPU can execute any (main) program, without wasting time by *not waiting* for the event to occur.
- quick response to events, needed for real-time control application in a multi-programming environment.

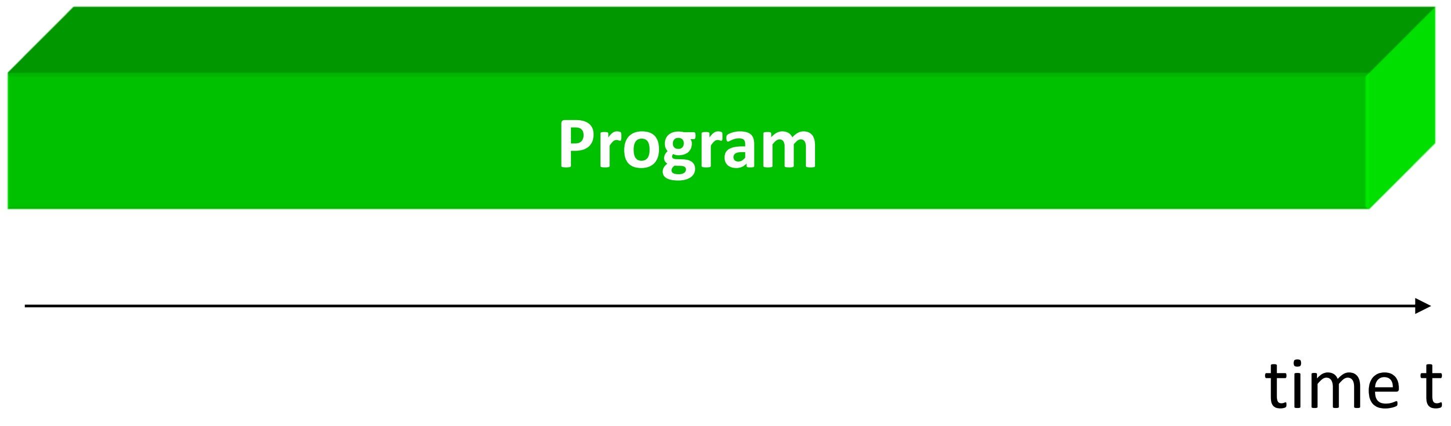
When will processor check for interrupts?



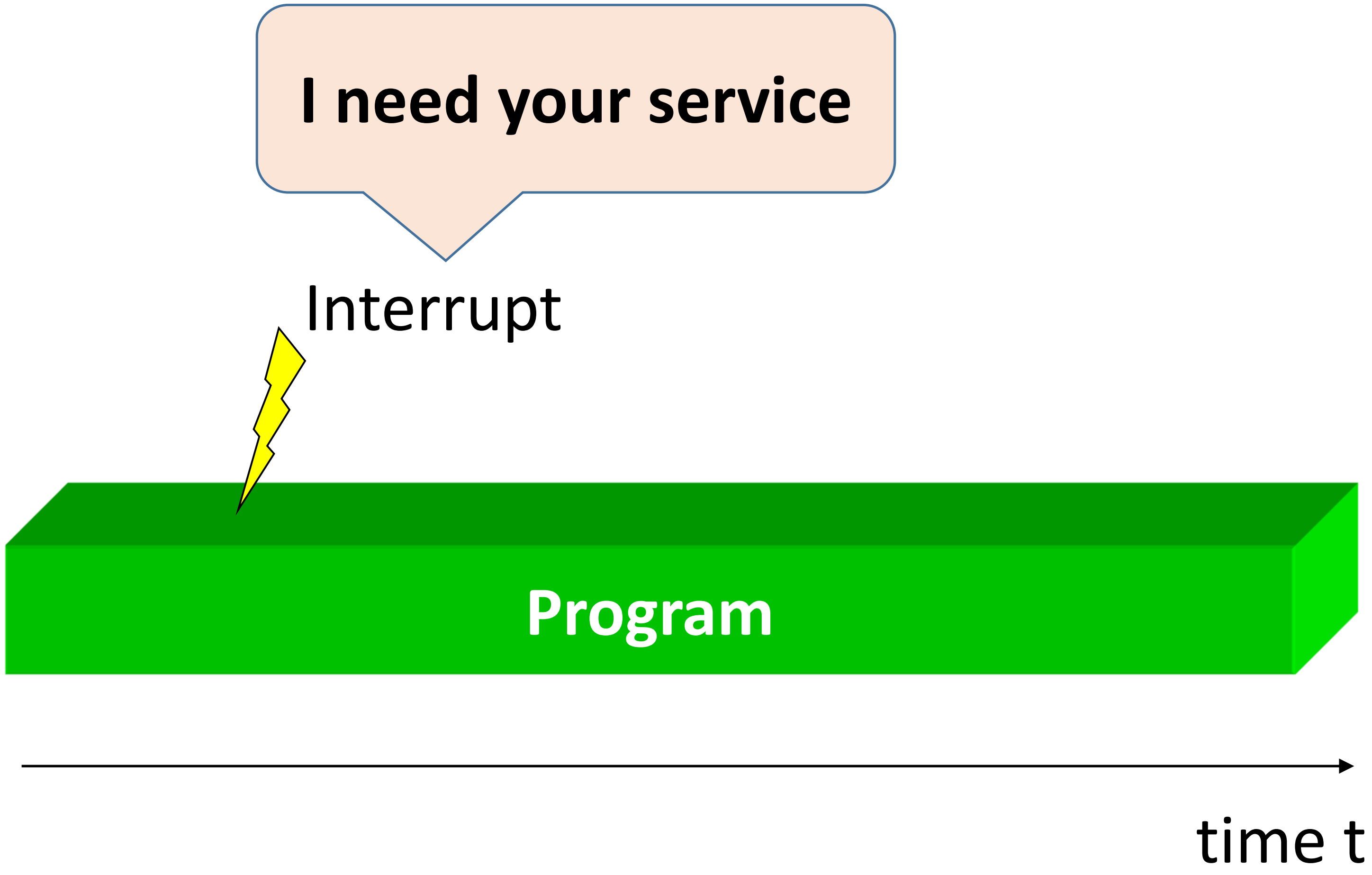
Remember the Instruction cycle?

If there is any interrupt, then start the program to service (handle) the interrupt and then come back to the program which was left on the mid way.

Interrupts

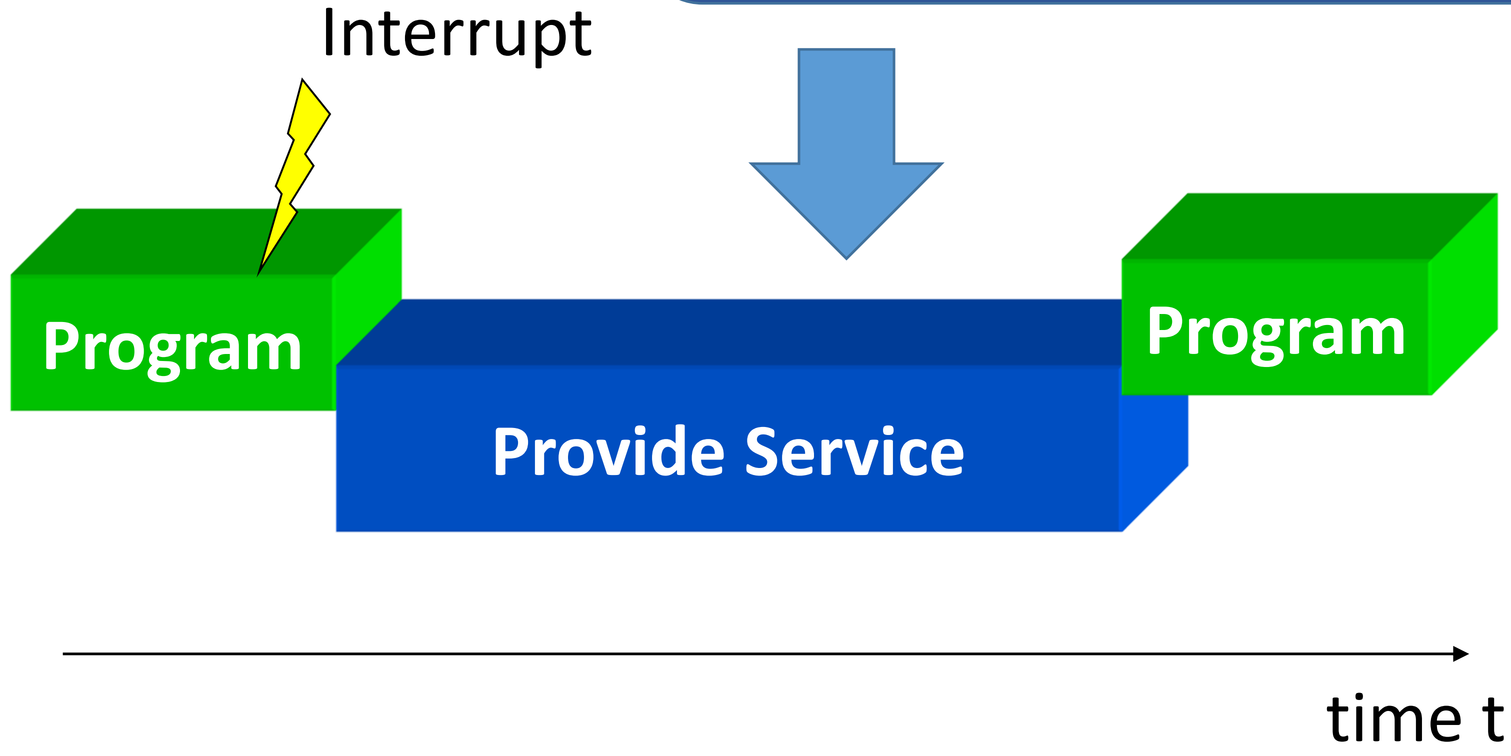


Interrupts

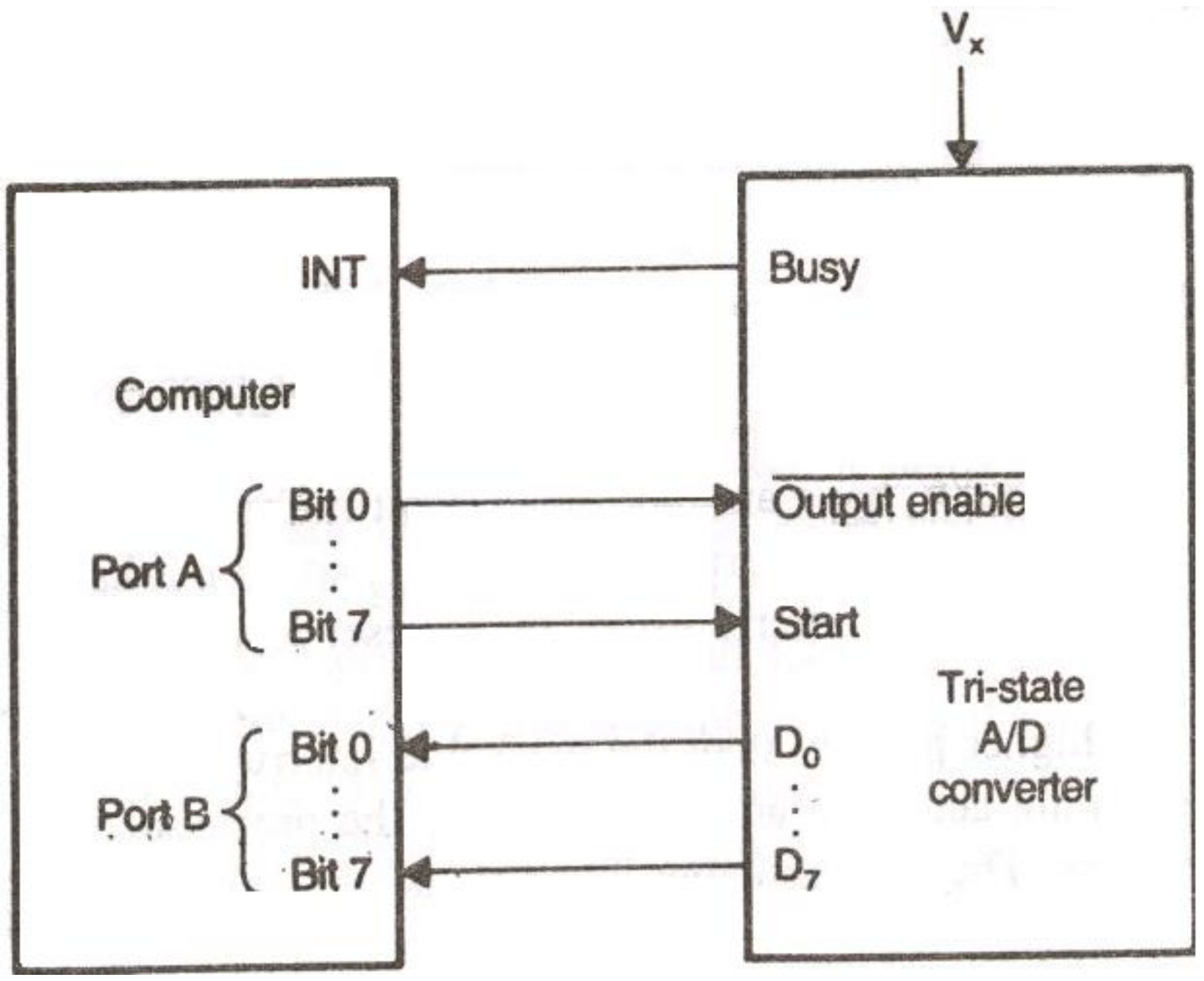


Interrupts

ISR, is a *short program* which will get executed in order to provide the service requested by the interrupting I/O device



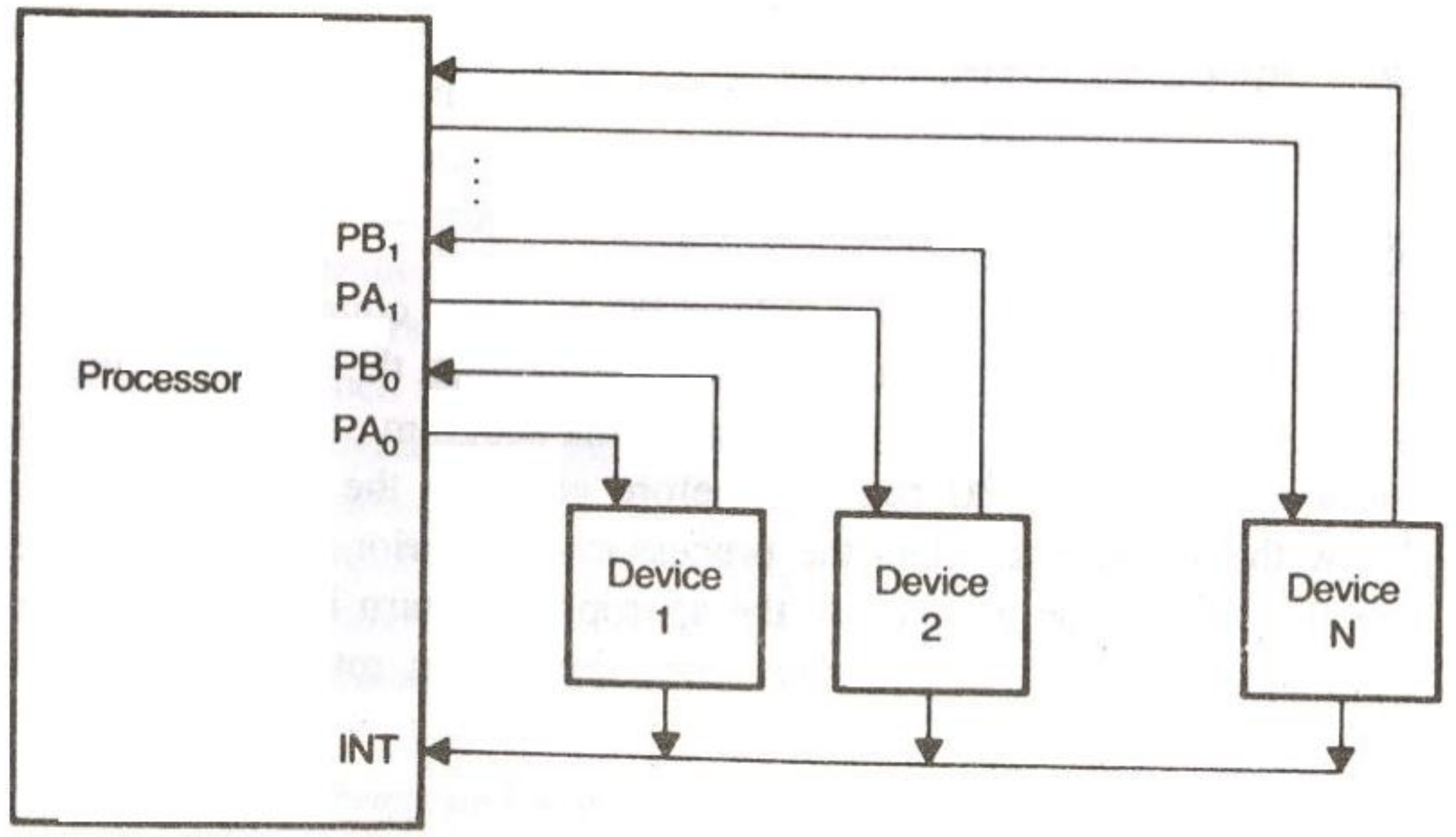
Interrupt I/O



How many interrupt signal pins should a processor have?

Can we handle all interrupts with only one INT pin?

Interrupt I/O: Polled Interrupts

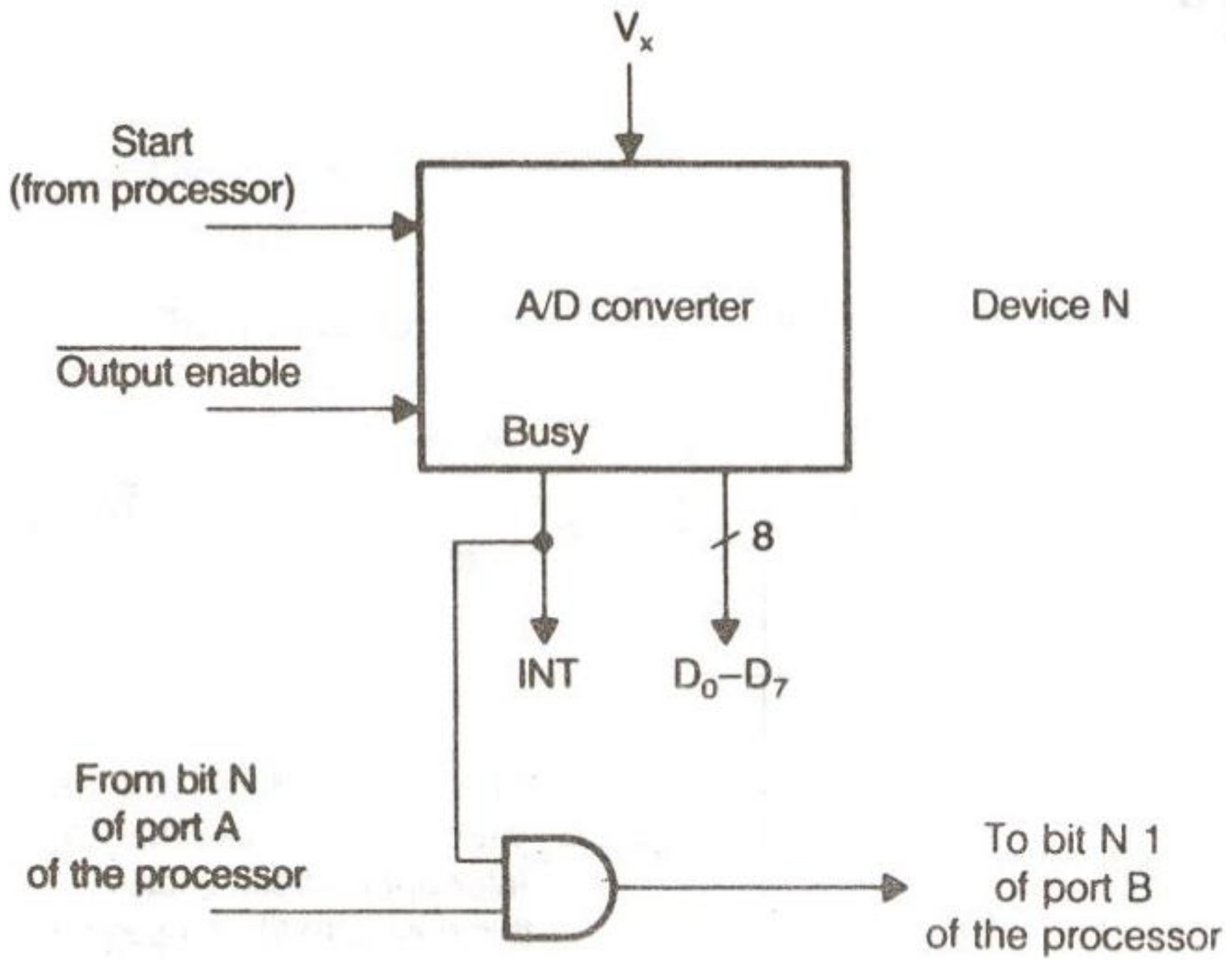


How to handle more than one device under interrupt system and processor has only one interrupt signal?

Three programs are involved in this method:

1. Main
2. Polling of interrupts
3. ISR

Interrupt I/O: Polled Interrupts

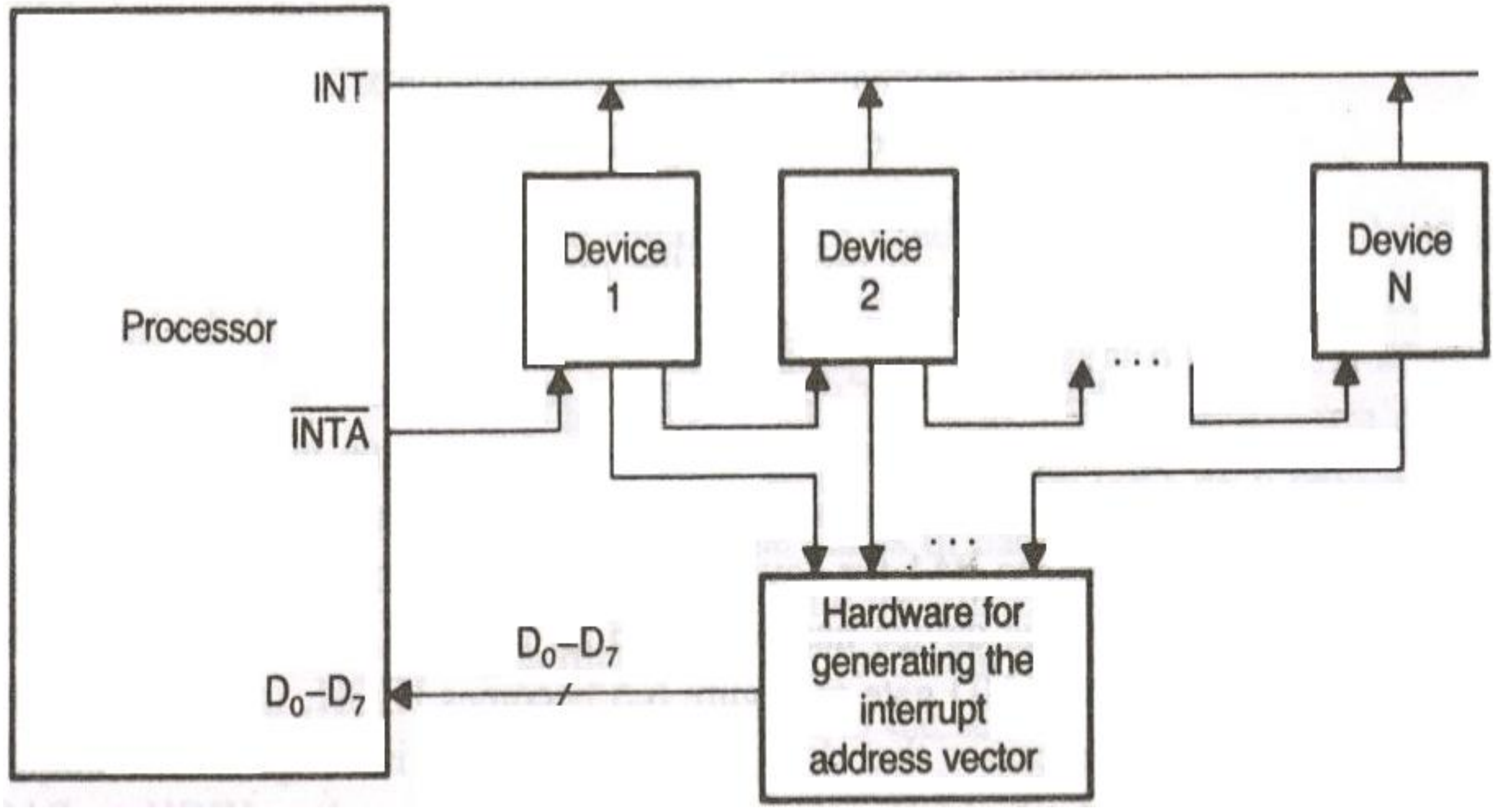


A logic '1' is sent and if logic '1' is received then that is the device which has interrupted.

This is program based polling and hence slower.

Time taken for polling itself may be too much for large number of devices.

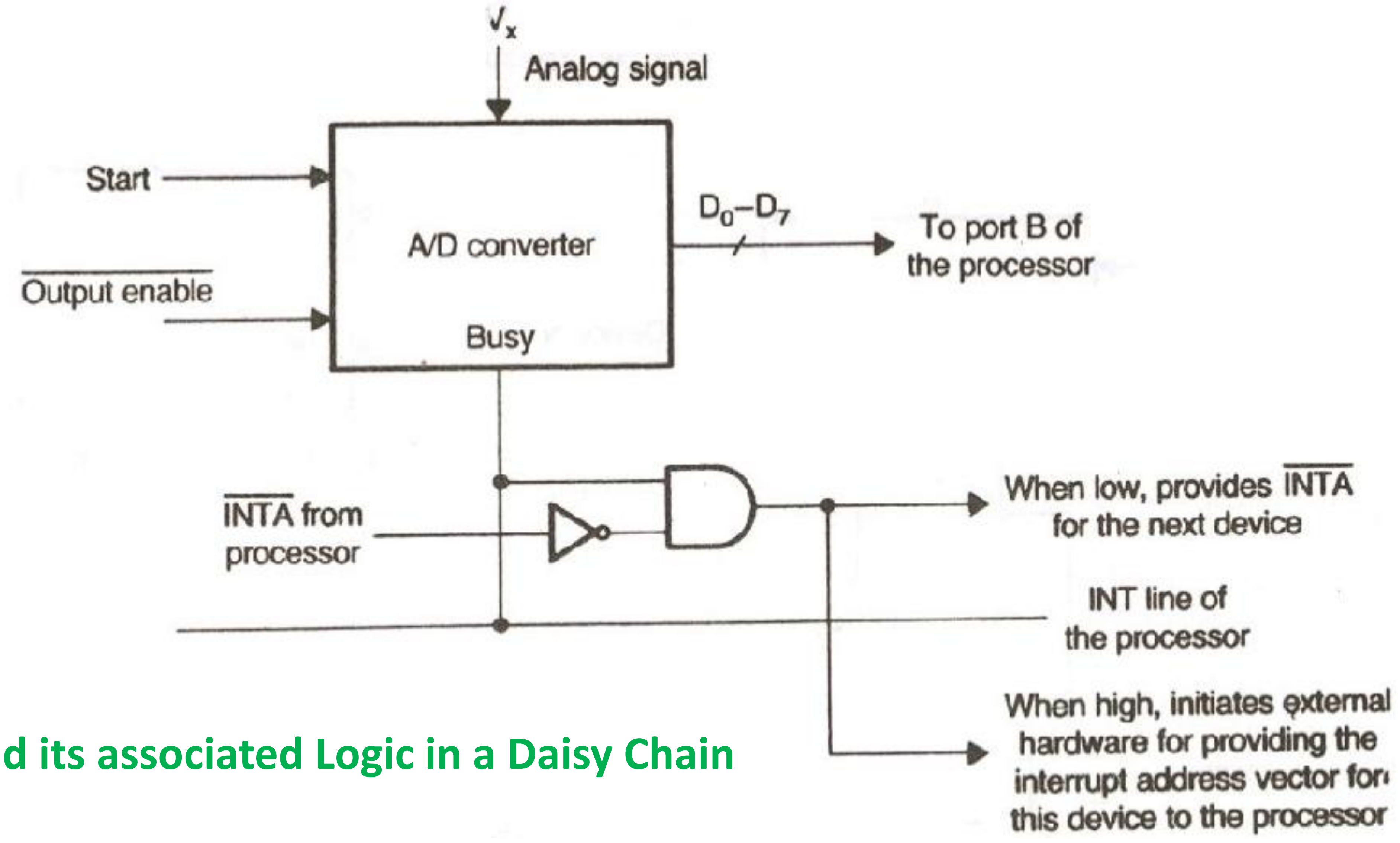
Interrupt I/O: Daisy Chain Interrupt



Hardware method of polling.

Faster than polled interrupts.

Interrupt I/O: Daisy Chain Interrupt



V_x
Analog signal
Start
$\overline{\text{Output enable}}$
A/D converter
Busy
D_0–D_7
To port B of the processor
$\overline{\text{INTA}}$ from processor
When low, provides $\overline{\text{INTA}}$ for the next device
INT line of the processor
When high, initiates external hardware for providing the interrupt address vector for this device to the processor

A device and its associated Logic in a Daisy Chain

Few comparing points

- Daisy chain has fixed priority
  - High priority is given to *electrically* closest device to $\overline{\text{INTA}}$ signal
- Fast as there is no code that needs to run

- Polled interrupt system has flexible priority
  - High priority is given to the device which is checked first by the polling program
- Slow as there is a polling program that needs to run

Few points about interrupts

- Maskable and non – maskable
- External (hardware), internal (also called as exceptions) and software interrupts (by programmer in programs)
- Software interrupts are similar to CALL and RET
- Vector address (it’s a bit sequence or an ID); specifies ISR address
- Vectored and non-vectored interrupts
- Nesting of interrupts
- Use EI and DI instructions to avoid nesting

Interrupt Overheads

Interrupt arrives

Complete current instruction execution

Save essential register information

Vector to ISR

Interrupt Latency

Save additional register information

Execute core body of ISR

Restore additional register information

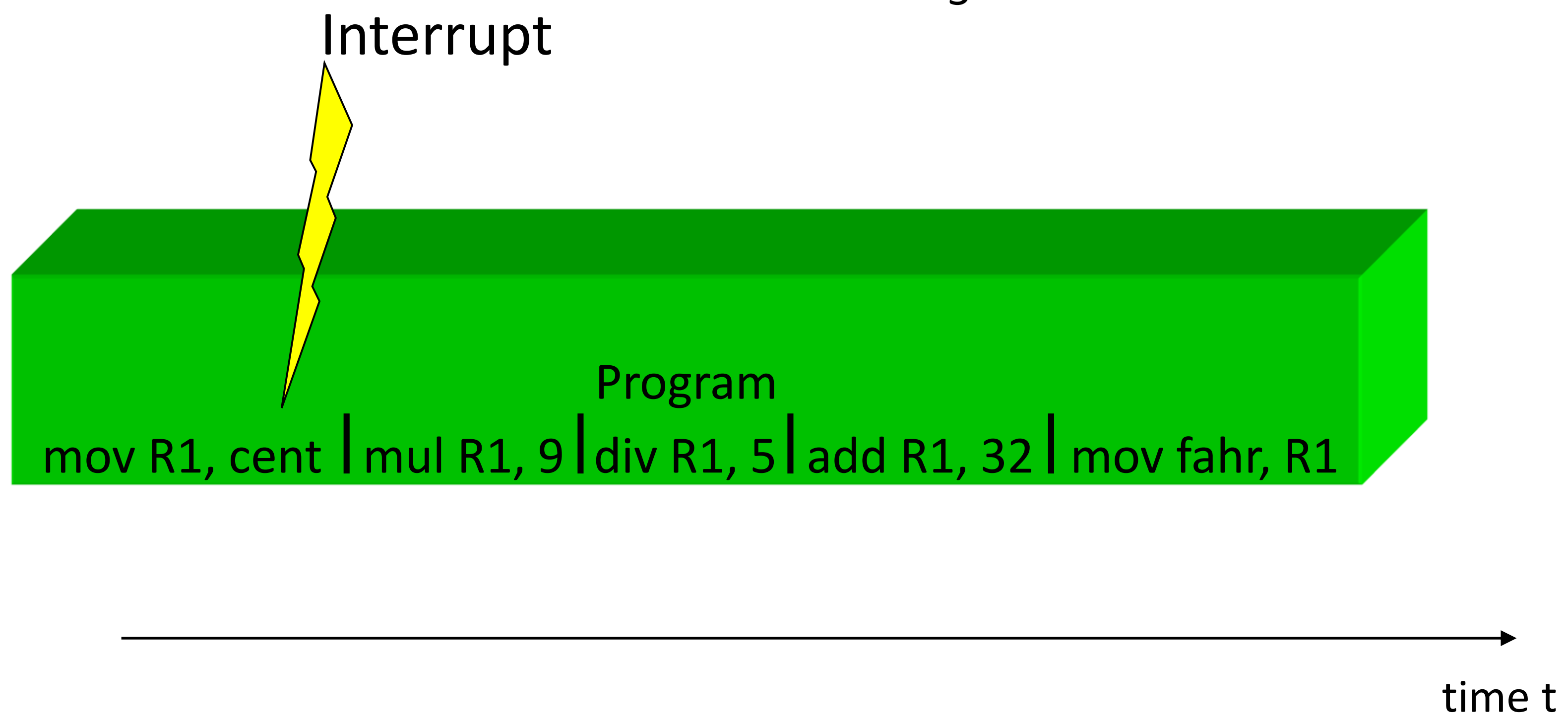
restore essential registers and return from interrupt

Resume task

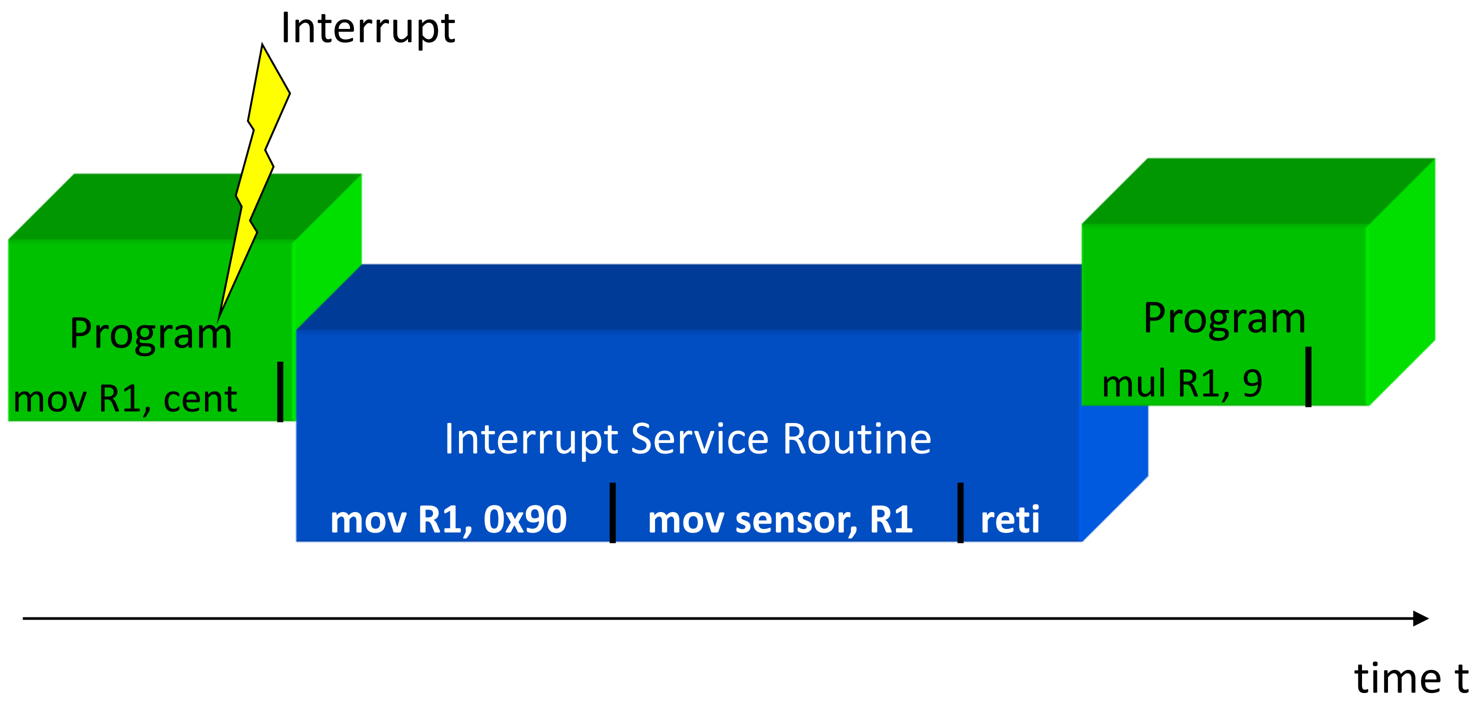
Interrupt Termination

Interrupts

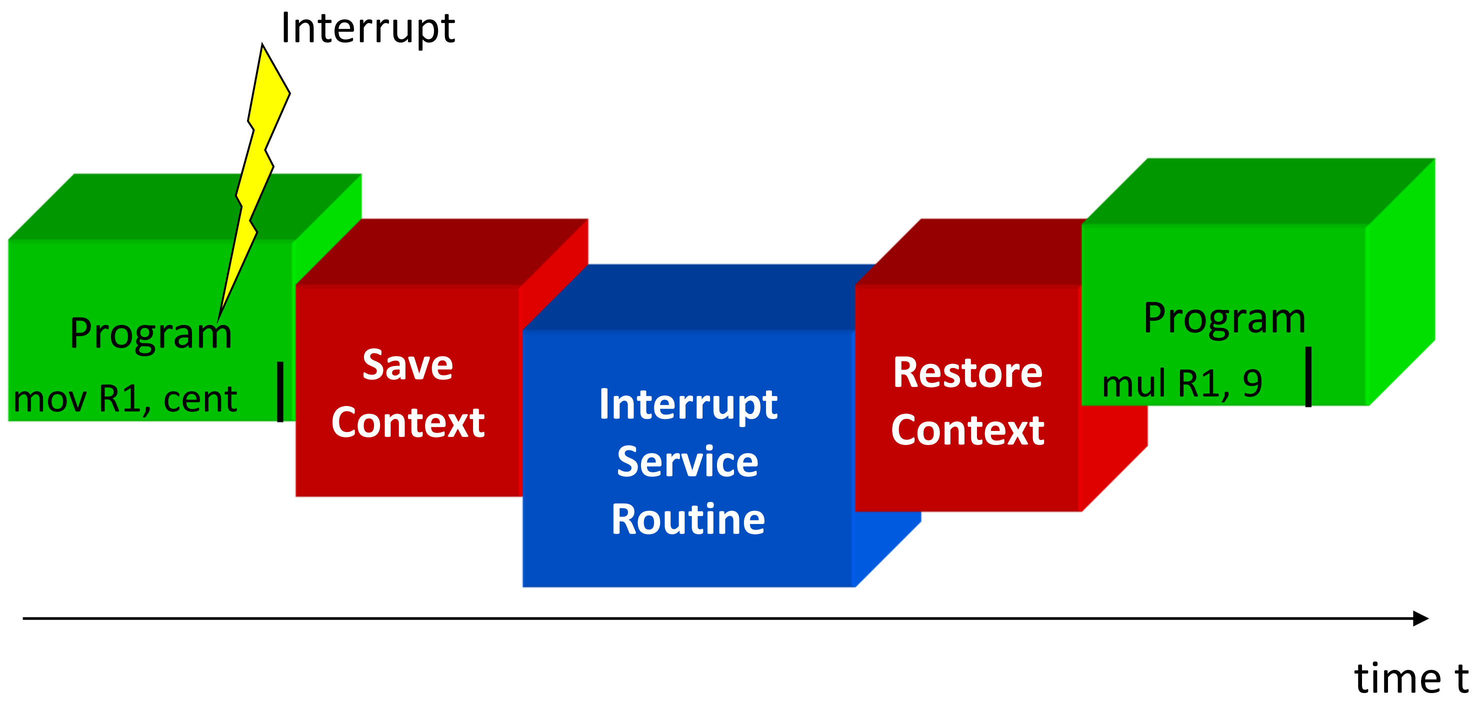
$$fahr = \left(cent * \frac{9}{5}\right) + 32$$



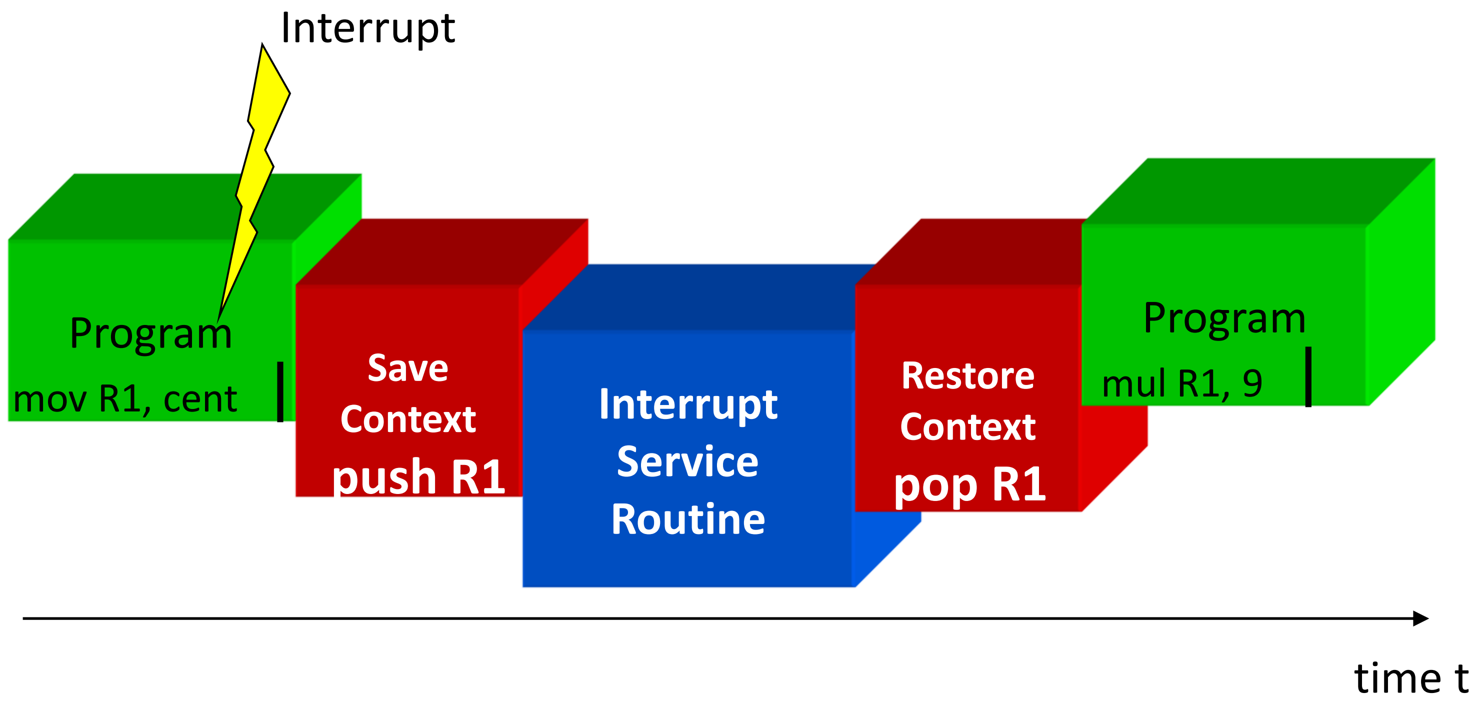
Interrupts


Interrupt
Program
mov R1, cent
Interrupt Service Routine
mov R1, 0x90
mov sensor, R1
reti
Program
mul R1, 9
time t

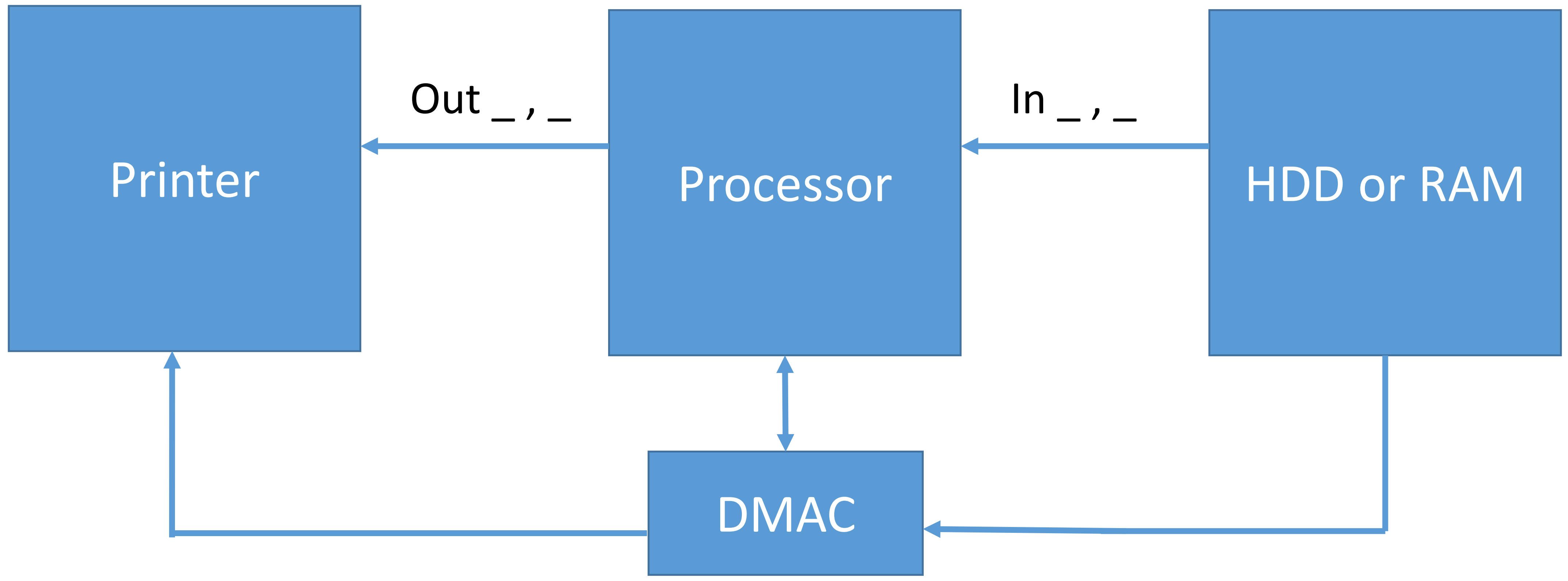
Interrupts


Interrupt
Program
mov R1, cent
Save Context
Interrupt Service Routine
Restore Context
Program
mul R1, 9
time t

Interrupts


Interrupt
Program
mov R1, cent
Save Context
push R1
Interrupt Service Routine
Restore Context
pop R1
Program
mul R1, 9
time t

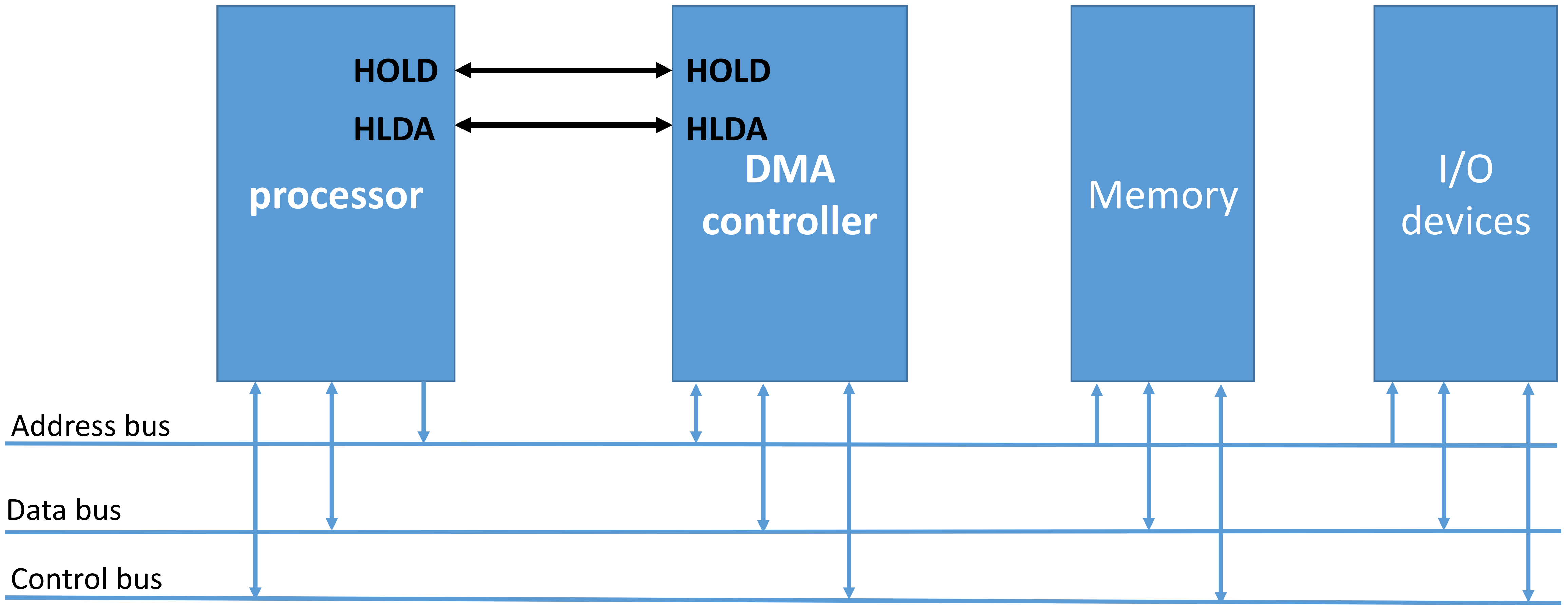
Need for Direct Memory Access, DMA



DMA concept and few terms

- Bus master and slave devices
- Bus request and bus grant
- Tristate or disconnect from the bus

DMA basic concept


HOLD
HLDA
processor
HOLD
HLDA
DMA controller
Memory
I/O devices
Address bus
Data bus
Control bus