

2026

CS 332

Subject 1

NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



Module 1

The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details

This is a bluish note!

*Operating
Systems:
Internals
and Design
Principles*

Chapter 1 Computer System Overview

Ninth Edition
By William Stallings

Operating System

- Exploits the hardware resources of one or more processors
- Provides a set of services to system users
- Manages I/O modules that manage secondary memory and I/O devices

Basic Elements of a computer

Processor

60-70 years ago

A 'Computer' could

first barely fit into a building

then it could hardly fit on a single floor of a building

then it could fit into a room

then in the 80s it could fit into a computer box or laptop

and nowadays it fits into a phone or is a phone

System Bus is the 'highway'/'cables' connecting all other elements (components)

Processor : CPU (Central Processing Unit) i.e execution unit

Processor : Microprocessor (miniaturized CPU)

Processor : a core of a multicore system containing multiple CPUs i.e. execution units.

I/O Modules

Main Memory

System Bus

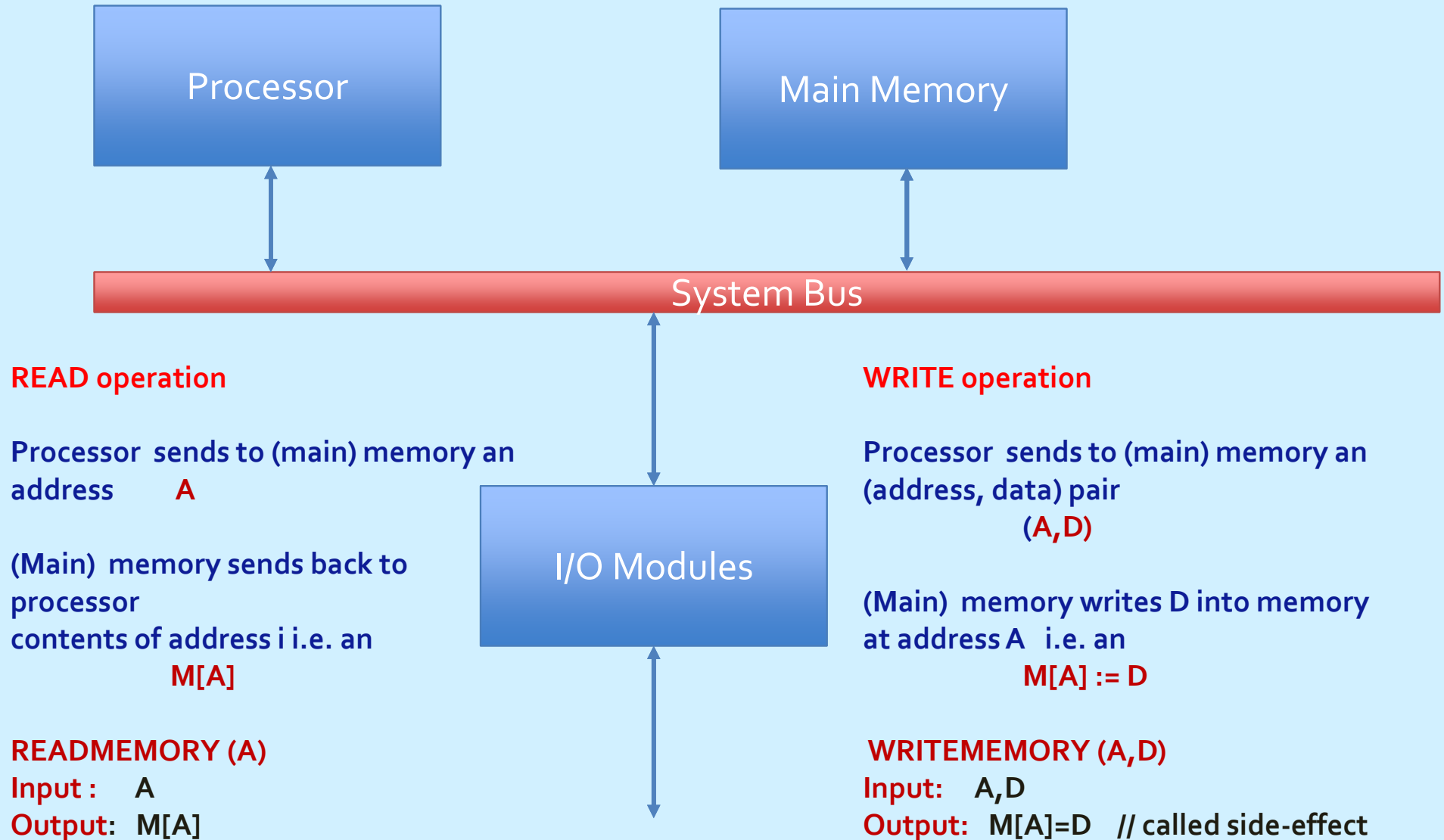


Component 1: (Main) Memory and Secondary Memory

Main Memory

- Stores data and programs
- Typically volatile
 - Contents of the memory is lost when the computer is shut down
- Referred to as *real memory* or *primary memory* or just *memory*

Read & Write from/to MM





Register: Small named memory inside Processor (aka CPU)

(The collection of registers of a processor is referred to as the register file or register set)

Cache: Slower memory than registers faster than MM.

(Memory inside or outside of CPU's die, chip, socket enclosure but on the same motherboard: location varies with CPU generation.)

Main Memory (MM): Main or Primary memory, one of the components of a computer outside of CPU and controlled by SystemBus

(We might refer to it as DRAM or SRAM depending on technology used; RAM means random access memory i.e. $M[A]$ takes same time to retrieve the byte independently of how complicated A looks like (eg whether A is 0, or A is 9 or A is 0xFF01234F4E88BCDE.)

Secondary Memory (SM): Any other memory that requires communication through I/O modules (e.g. a variety of drives some of them known as disk drives)

Buffer: Memory needed to regulate difference in speed between two devices (e.g. CPU and I/O devices)

Note 1: One might consider a cache a type of buffer memory between the CPU and main memory

Note 2: A buffer between CPU (fast dev.) and an I/O device (slow dev.) might be in the I/O module or available inside the device.

Note 3: CPUs use multiple caches. The closest to the CPU is the Level-1 (L1), the next closest Level-2 (L2) and so on.

The Memory Hierarchy

- Going down the hierarchy:
 - Decreasing cost per bit
 - Increasing capacity
 - Increasing access time
 - Decreasing frequency of access to the memory by the processor

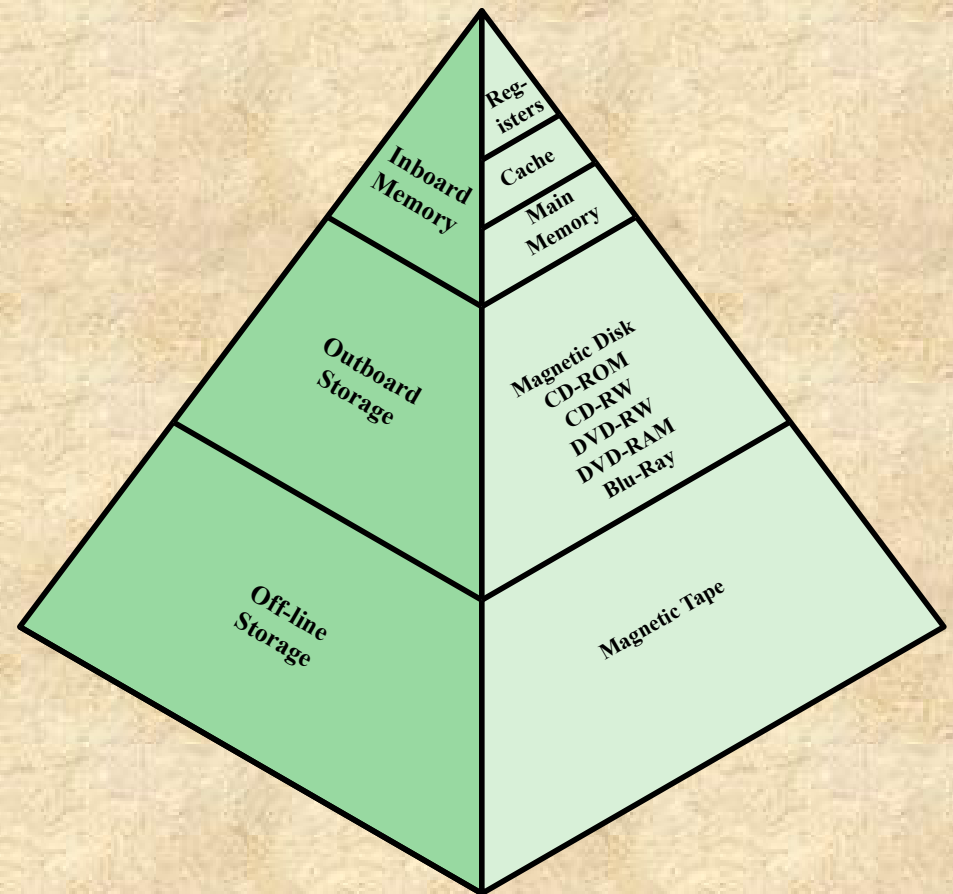


Figure 1.14 The Memory Hierarchy



Main memory M is sometimes referred to as RAM

RAM means Random Access Memory; Random Access means time to get $M[0]$ or $M[1000000]$ or $M[1000000000]$ is the same and independent of the 'complexity' of the 'address' (0 is simple/smaller than 1000000 which is simpler than 1000000000 plays no role).

RAM can be DRAM (Dynamic RAM), SRAM (Static RAM)

DRAM

uses 1 transistor per bit (and 1 capacitor); CHEAP; periodic refresh (<1ms)
After every read, read bit need to be rewritten; volatile; minimal persistence (1-10s); EMI (ElectroMagneticInterference) sensitive; Used for MainM.

SRAM

uses 4-6 transistors per bit; x10 to x20 faster than DRAM; no refresh; can be non-volatile but is usually volatile with some remanence (preservation of data); non EMI sensitive; Used for MainM in space apps or caches in CPUs.

SRAM 10x to 100x more expensive than DRAM



Fast vs Furious

Latency:
Throughput:

Time to retrieve the first byte
 Time to retrieve additional (sometimes closely located) byte(s)

Example 1.

Hard disk drive latency is ~20ms. Throughput is 10,000,000B/s or more i.e. 100ns/B.

Example 2.

(Main) memory latency is 80-120ns. Throughput can be faster ~ 100GiB/s for some Intel CPUs

with capable motherboards i.e. 0.01-0.02ns. This is a theoretical measure only.

Level	Typical size	Latency
Lo: CPU registers	8-32	1 cycle
L1: level-1 cache (instructions & data)	32KiB each (L1I, L1D)	1-5 cycles
L2: level-2 cache	256-4096KiB	15 cycles
L3: level-3 cache	4-128MiB	45 cycles
L4: level-4 cache (Intel Broadwell)	128MiB	
SRAM (L2 or L3 use it)	1-16MiB	Lo-L2 cache levels
DRAM (MM or ... L4)	several GiB	45cycles+60ns~80ns
SSD	hundrds of GiB	0.1-0.2ms
HDD	several TiB	15-20ms for most HDD (less for server HDD)

Note 1. For a 1GHz CPU 45 cycles is 45ns ; for a 2GHz cpu it is roughly 23ns. Thus 45cycles+60ns is roughly 80-100ns.

Memory Relationships

Faster
access time
= greater
cost per bit

Greater capacity
= smaller cost per
bit

Greater
capacity =
slower access
speed



For the past 50 or so many years,

we build Large, **Slow**, memory units that are **Inexpensive**

building Large, **Fast**, memory units is still **Expensive**

We combine Smaller, **Fast** memory units that are **Expensive**

with Large, **Slow** memory units that are **Inexpensive**

to get the illusion of having something

faster than **Slow** but slower than **Fast**



more Definitions

Source code:

Written in high level language (C,Java,etc). Resides on SM

(Exampe: `int i=0, sum=0; for (i=0 ; i <= 20 ; i++) sum += i;`

'Assembly':

```
loop1      xor cx,cx    // i=0
           xor ac,ac    //sum=0
           myadd ac,cx
           inc cx
           cmp cx, 20
           jle loop1
```

High level program converted into machine code (executable file) using a compiler infrastructure. The machine code is binary but can be viewed as a sequence of mnemonics for machine code instructions or generated independently prior to machine code generation. Resides then on SM.

Machine code:

0/1s in memory representing or indicating instructions and data; Resides on SM. At execution time it is moved to MM!

```
130  11000010 //XOR :1100 reg cx: 0010
131  11000001 // XOR :1100 reg ac:0001
132  01000011 // ADD ac= ac+cx
133  10000010 // ?
134  11110010 // tricky part cmp xcx
135  00010100 // 20 is here
136  11100000 // jump i.e. goto where?
137  10000100 // 0b10000100 better be 132!
```

(Fragment on the left shows a fragment of main memory in binary a byte at a time. It describes bytes starting with address 130 through 137; comments are to show what assembly instruction maps to correspond byte or sequence of bytes!

Note: The 'assembly' code or 'machine code' might resemble Intel 8086 code but it is not! It might resemble Textbook instructions or register names but they are not! It's a figment of my 'scientific freedom' to make things presentable and relatable to a later discussion

We usually refer to the machine code as the program or the executable or executable file.

In order to run the 'program' (aka executable machine code file) we need to load it into the main memory and direct the CPU to the first instruction to be executed and then ask the CPU to do so! Then the program becomes a program in execution that WE WILL CALL a process!



Component 2: Processor

Processor

Controls the
operation of the
computer

Performs the
data processing
functions

Referred to as
the *Central
Processing Unit*
(CPU)

Microprocessor

- Invention that brought about desktop and handheld computing
- Contains a processor on a single chip
- Fastest general purpose processors
- Multiprocessors
- Each chip (socket) contains multiple processors (cores)

CPU and its unit(s)

CPU (old times)

Central Processing Unit; contains a (data) processing unit also known as execution unit and was known as **processor**

CPU (nowadays)

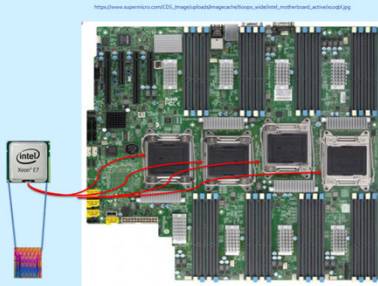
It contains multiple processing units known as cores and collectively known as **multicore processor**

(Micro)processor:

Miniaturized CPU on a single (micro)chip

Die and (micro)chip:

A microchip contains one or more dies (silicon integrated circuits) covered with epoxy inside a plastic or ceramic housing with gold-plated connectors. Die can contain processing units and/or memory.



Socket:

Motherboard receptacle (of a chip); 4 shown

Motherboard:

Houses CPU, MainMemory, System bus, I/O modules

SoC:

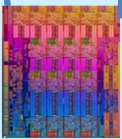
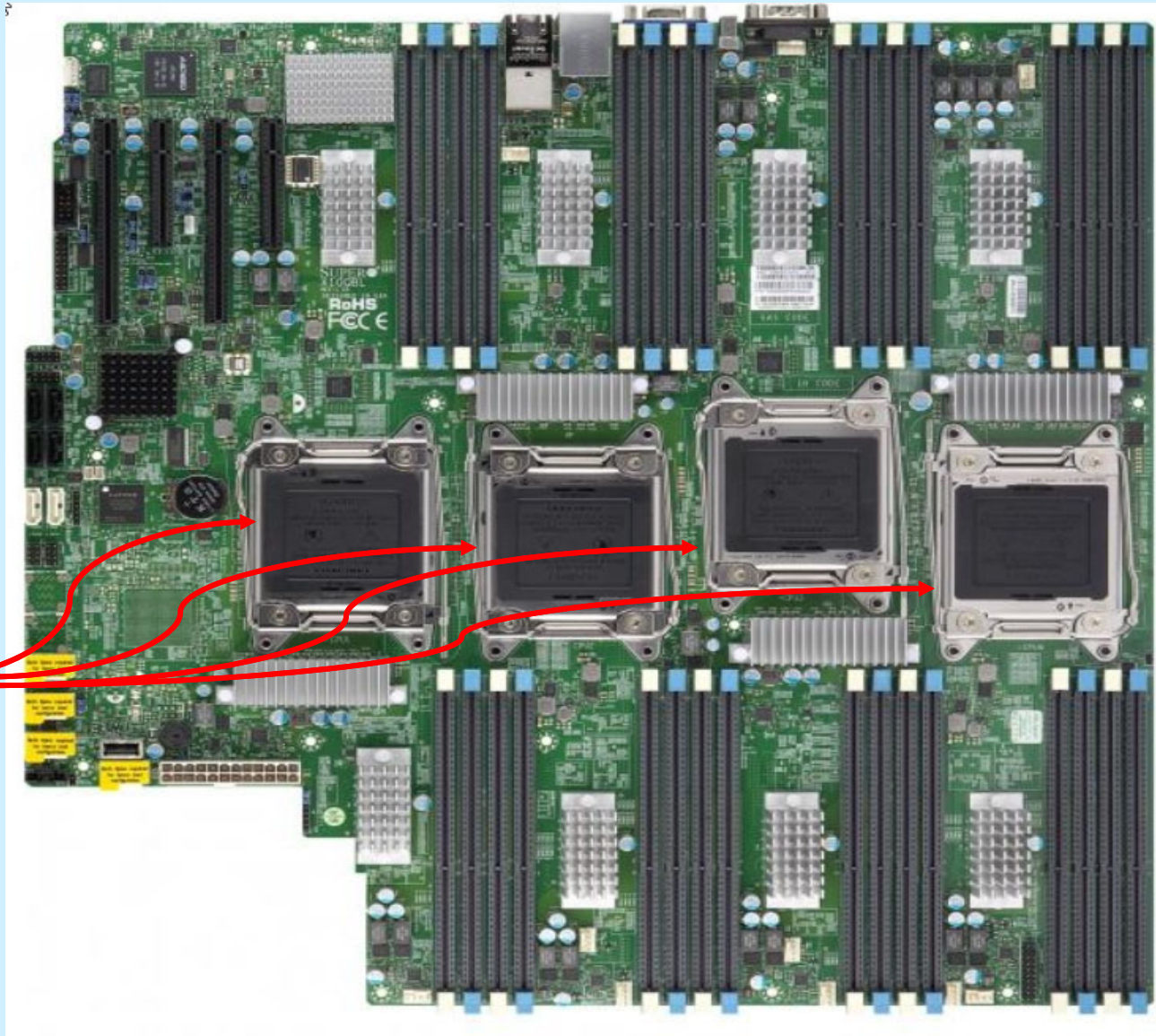
System on a chip : 'computer ON a chip'

A motherboard

https://www.supermicro.com/CDS_Image/uploads/imagecache/600px_wide/intel_motherboard_active/x10qbl.jpg

This motherboard
houses
4 of the Xeon
processors
shown in the previous
two slides
and also on the next
slide

Motherboard has four
(square) **sockets**





more definitions

Processor (generic)

Uniprocessor:

Computer:

Multicomputer:

Multiprocessor:

Multiprocessor system:

One execution unit InOneChip; short for microprocessor

One execution unit /processor (= one CPU with one exec unit)

A 'box' containing Processor(s), MM, system bus, plus more

More than one computers ($\#Computer > 1$)

More than one processors. ($\#Computer = 1 \wedge \#Processor > 1$)

Multicomputer $\vee$ Multiprocessor

(in extreme cases also a computer with one multicore processor)

Symmetric Multiprocessor (SMP): A computer with more than one processors of the same type all sharing the same main memory ($\#Computer = 1 \wedge \#Processor > 1 \wedge SameType$)

Multicore processor: A processor with more than one execution units known as cores i.e. ($\#Processor = 1 \wedge \#ExecUnit > 1 \wedge OnOneChip$) ExecUnit and core as synonym nowadays

Note: $\wedge$: conjunction , $\vee$ disjunction

Multicomputer: ($\#Computer > 1$)

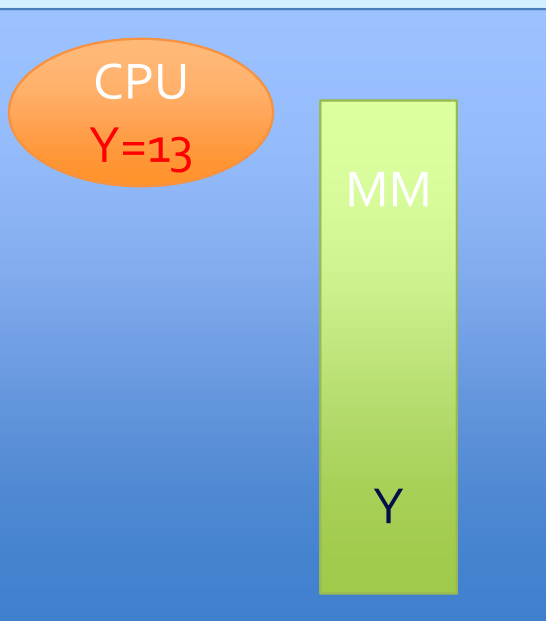
Multicore processor: ($\#Processor == 1$)
&& ($\#ExecUnit > 1$) && OnOneChip

Textbook uses term 'multicore computer' : it means 'multicore processor'

THEN

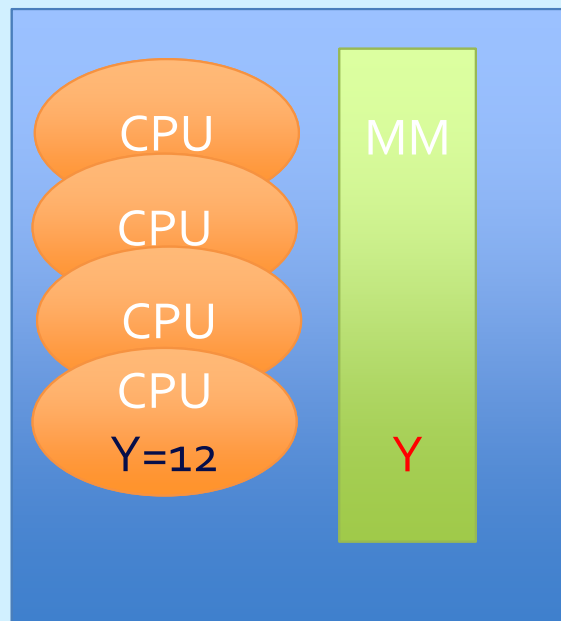
30-70 years ago

Computer: uniprocessor



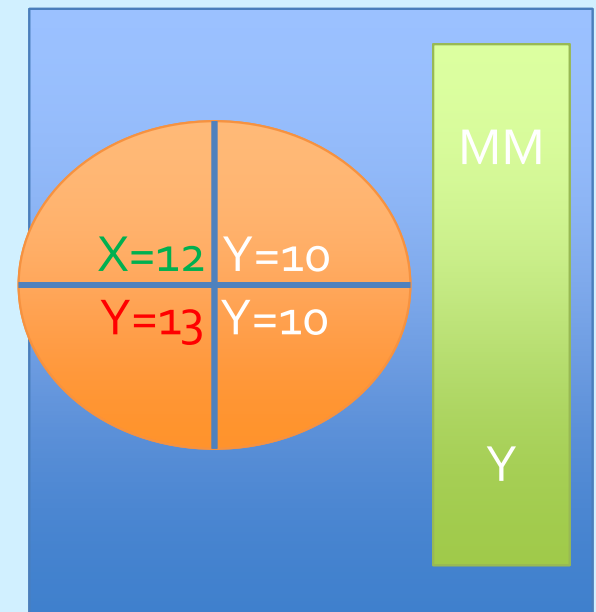
30years ago

Computer: SMP



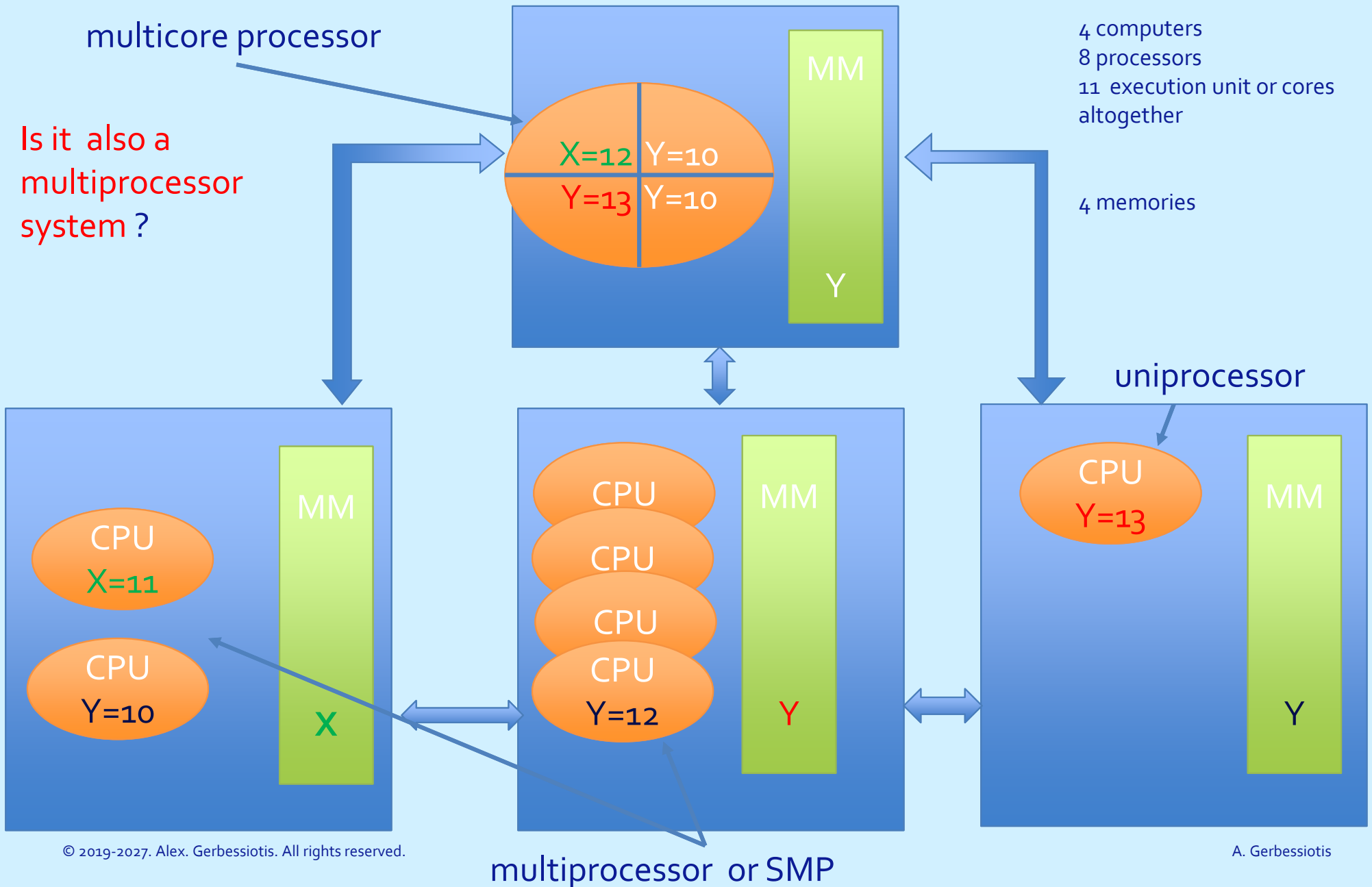
15 years ago

Computer: multicore processor



1 computer with 1 processor (1 execution unit total), 1 main memory ; a Uniprocessor.

NOW: multicomputer

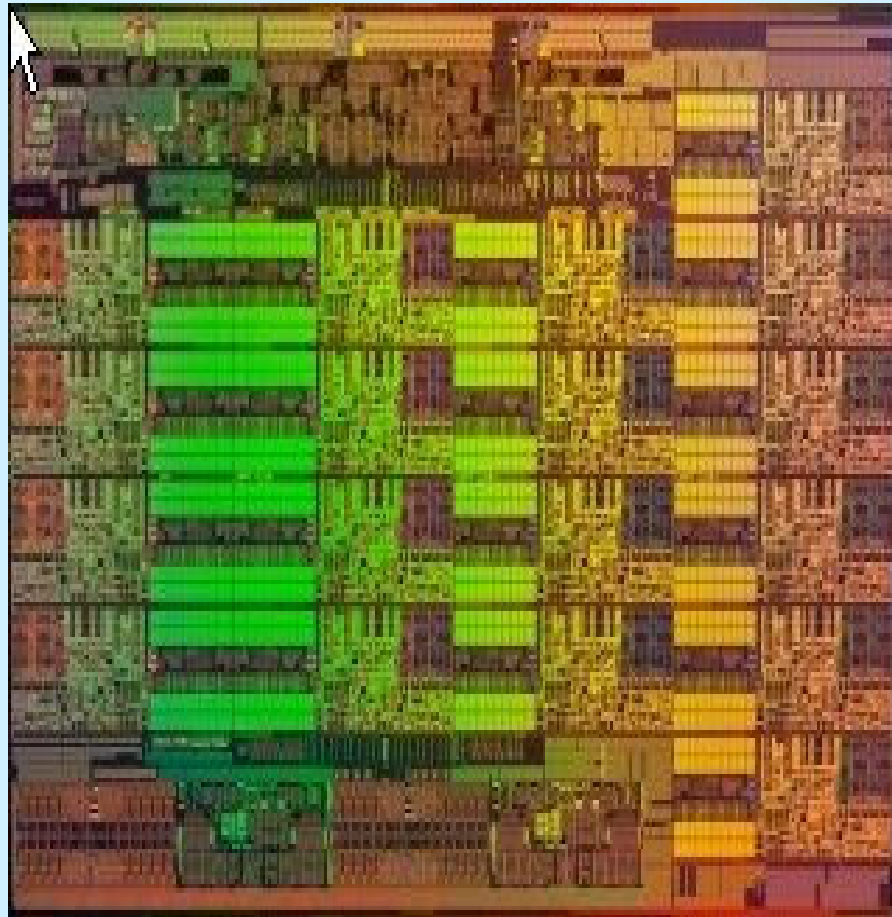
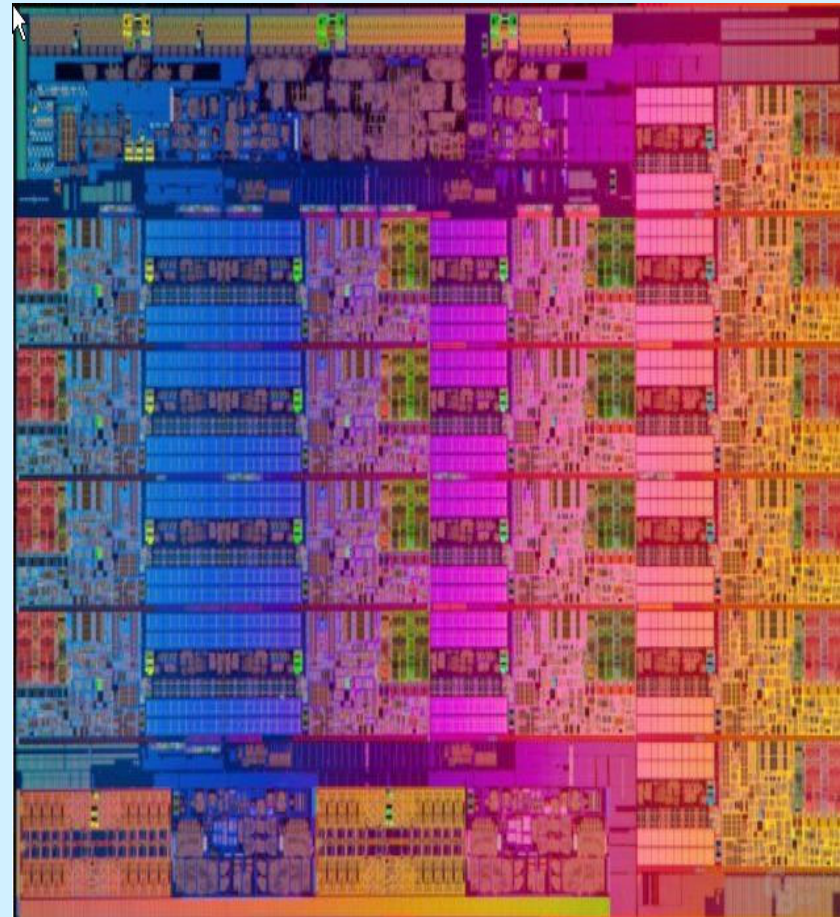


Multicore Computer

- Also known as a chip multiprocessor
- Combines two or more processors (cores) on a single piece of silicon (die)
 - Each core consists of all of the components of an independent processor
- In addition, multicore chips also include L2 cache and in some cases L3 cache

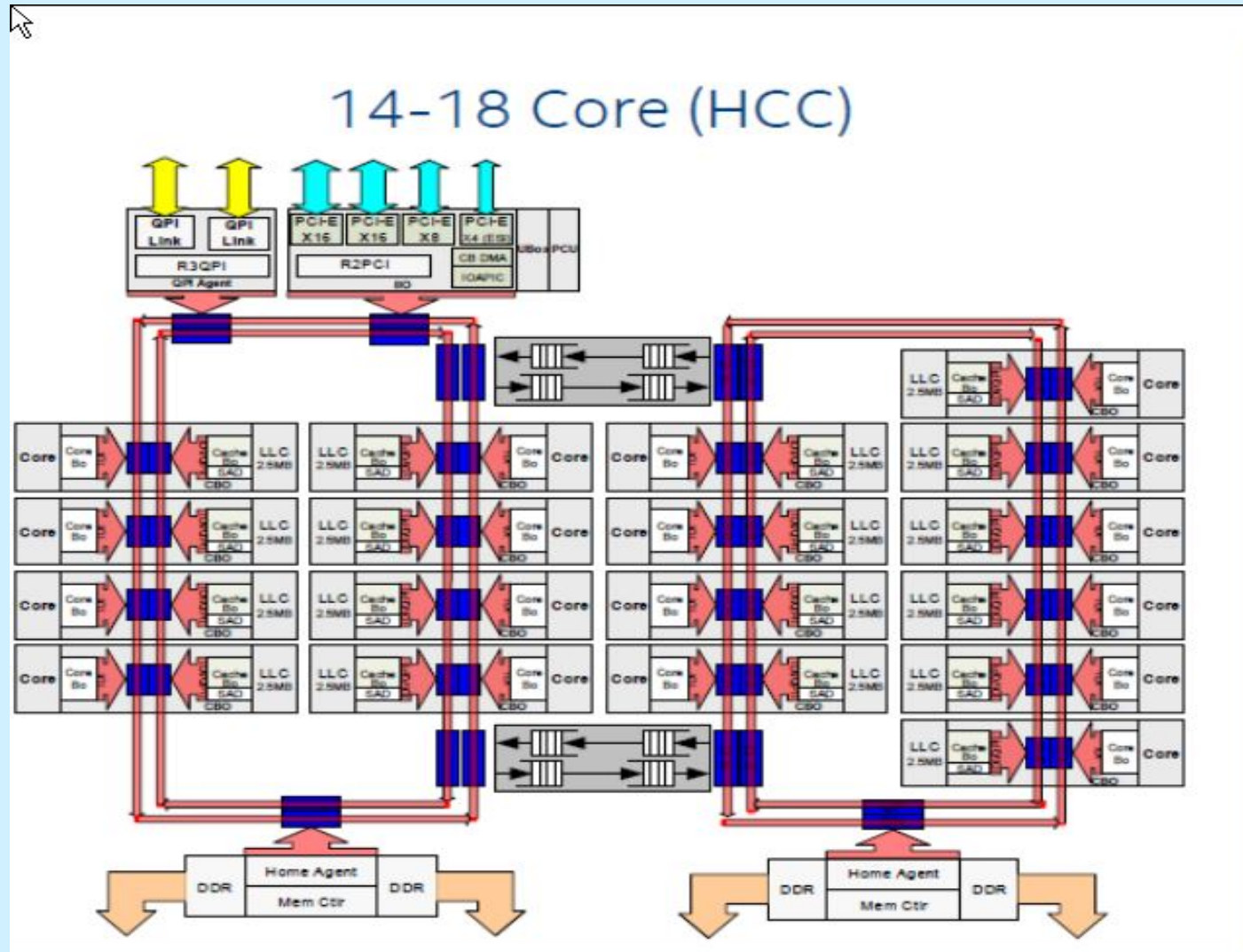


Multi-core or Multicore processor



same as before

<http://techfr2.blogspot.com/2017/07/Xeon.e7-8800.html>



Symmetric Multiprocessors (SMP)

- A stand-alone computer system with the following characteristics:
 - Two or more similar processors of comparable capability
 - Processors share the same main memory and are interconnected by a bus or other internal connection scheme
 - Processors share access to I/O devices
 - All processors can perform the same functions
 - The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels

SMP Advantages

Performance

- A system with multiple processors will yield greater performance if work can be done in parallel

Scaling

- Vendors can offer a range of products with different price and performance characteristics

Availability

- The failure of a single processor does not halt the machine

Incremental Growth

- An additional processor can be added to enhance performance



Uniprocessor: 1CPU

Job 'SMP' consists of three tasks A,B,C Requirement: B,C after A completes

Completion Time= 30u
 Turnaround Time= 30u

CPU-0



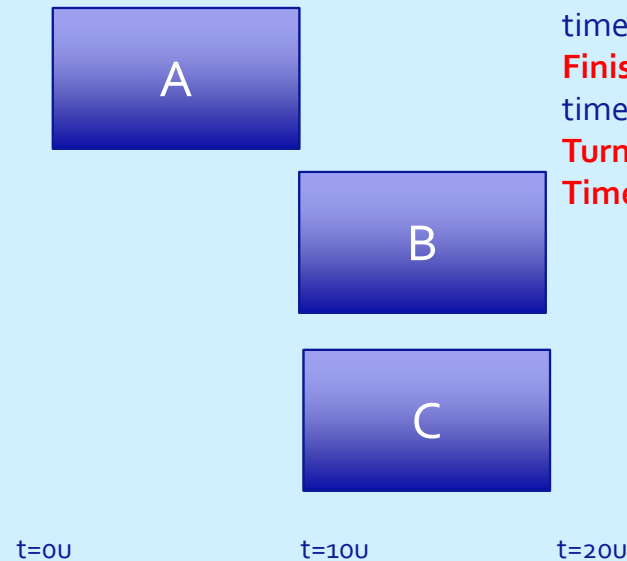
SMP: 3CPUs

CPU-0

CPU-1

CPU-2

Completion Time=20u
 Turnaround Time= 20u



Definitions

Start time: here of job 'SMP' consisting of three tasks A,B,C as noted.

Arrival time might be a synonym of start time; usually Arrival time precedes Start time

Finish time or Completion time: time last task of job completes its execution.

Turnaround time = Completion Time - Start Time (or Arrival Time).



Instructions and Data and their Architectural Separation!



Harvard Architecture:

Separate Memory for Data and Programs (Instructions)

Von-Neumann Architecture:

Same memory for Data and Programs (Instructions)

Modern architectures are ... (mostly sort of) ...

In fact they are

L₁ (Intel, AMD) cache is

L₂, L₃ cache and others are

von-Neumann

hybrid von-Neumann

Harvard (separate L₁D from L₁I)

von-Neumann (unified)

yet modern programs use separate areas in main memory to store (1) programs and (2) different types of data. Do not forget though that it is in the same Memory!



Component 2: Processor and Component 1: Main Memory and their Architectural Relationship (and proximity)



HARDWARE (cores, processors, execution units)

execute **INSTRUCTIONS** that operate on a stream of **DATA**

With the next slide we offer a taxonomy of **HARDWARE**

based on

Instructions (I) and **Data (D)**



Flynn's Taxonomy based on instruction execution(I) and data stream(D)

SISD (Single-Instruction Single-Data)

Microprocessors of 20-30 years ago

SIMD (Single-Instruction Multiple-Data)

Data-parallelism in 'Supercomputers' CM-1, CM-2, CM-200, Cray X-MP, commodity processors Intel MMX, and graphical processor units eg GPUs

SIMT (Single-Instruction Multiple-Thread) model as in NVIDIA's Compute Unified Device Architecture (CUDA) GPUs.

MISD (Multiple-Instruction Single-Data)

Systolic arrays , Pipeline

MIMD (Multiple-Instruction Multiple-Data)

MPMD (Multiple-Program Multiple-Data)

SPMD (Single-Program Multiple-Data) : CM-5, Cray T3D/T3E, SGI Power Challenge,
PC clusters and other multicomputers

Graphical Processing Units (GPU's)

- Provide efficient computation on arrays of data using Single-Instruction Multiple Data (SIMD) techniques pioneered in supercomputers
- No longer used just for rendering advanced graphics
 - Also used for general numerical processing
 - Physics simulations for games
 - Computations on large spreadsheets

GPU : NVIDIA Turing



<https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>

(# computers =1), (#processors =?), (#cores = ?)



72 SM
 (SM: Streaming Multiprocessor)
 Each SM has 64 CUDA cores!
 Total number of cores: 72x64



Component 1: Main Memory and architectural relationship In a multi*



1. Shared Memory (eg SMP) :

All processors (execution units) share the same and one memory.

Example. The processors of an SMP system but also the cores of a multicore CPU.

UMA: Uniform Memory Access. All processors retrieve one byte e.g. $M[o]$ in the same amount of time

2. Distributed Memory (eg Multicomputer, Multiprocessor system):

Each computer has its own memory and makes it available to other processors/execution units of the multicomputer. One processor can access $M[o]$ from its own memory, but also $M[o]$ from another processor call it processor 5. We view memory as a pair of (ProcessorID, Address) rather than as Address. Implication: every processor (in fact participating execution unit) has a unique ID.

NUMA: Non-Uniform Memory Access. Processors retrieve one byte e.g. $M[Address]$ in a varying amount of time depending on proximity of the memory containing the byte and the speed of access between the computer of the client processor (generating the memory request) and the computer of the server processor (satisfying the memory request).

3. Distributed Shared Memory (eg Multicomputer, Multiprocessor system):

Each computer has its own memory and makes it available to other processors/execution units of the multicomputer in the form of shared memory let us call it $DSM[o]$. There is only one address o across all processors. Thus $DSM[o]$ resolves uniquely in a (ProcessorID, Address)

cc-NUMA : Cache Coherent Non-Uniform Memory Access. A cached shared variable is modified by one processor and that shared variable is requested by another processor

A multicomputer is an interconnected system!



Component 3: I/O Modules

Component 4: System Bus

I/O Modules

A I/O module is not a device; A I/O module communicates with I/O devices

Move data
between the
computer and
its external
environment

Secondary
memory devices
(e.g. disks)

Communications
equipment

Terminals

System Bus

- Provides for communication among processors, main memory, and I/O modules



Shared Memory-based communication

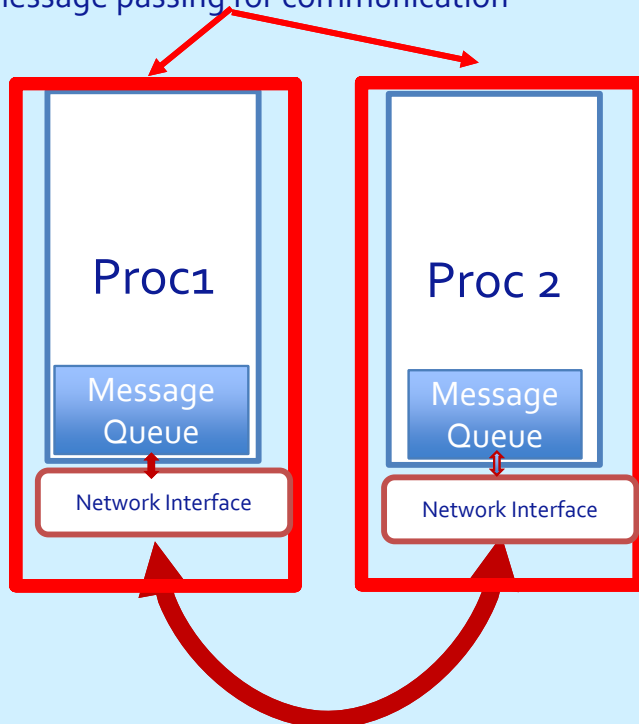
Use shared memory of an SMP for processors to communicate with each other directly. No 'message' is formed for this communication.

Message passing

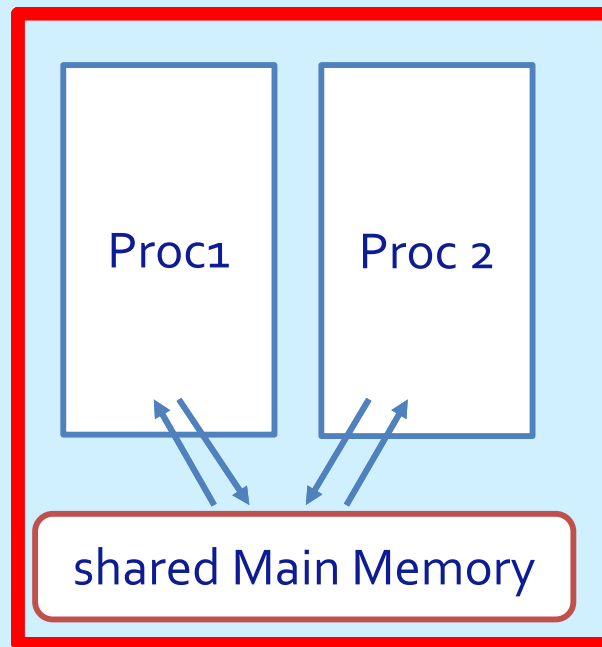
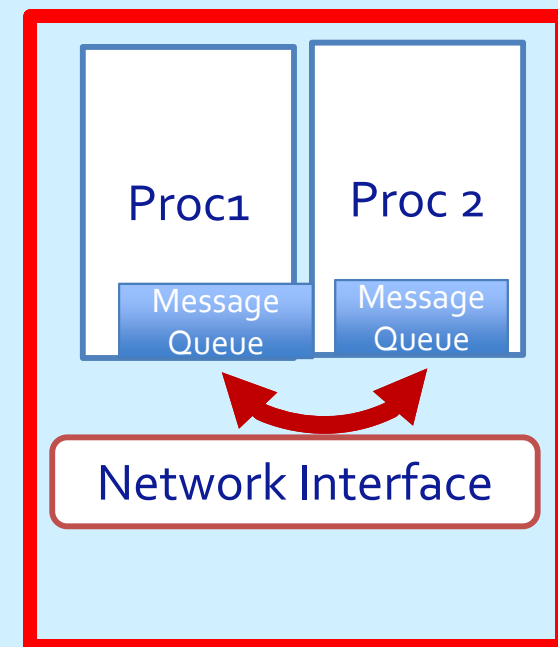
Communication of processors in a multiprocessor system and in particular a multicomputer is realized by a processor (source) sending a message to a another processor (destination) by using the regular network interface cards (networking) of the comptuers hosting the two processors or a specialized network that is fast and optimized for the particular communication.

It is possible that message passing is realized using shared memory based communication. A 'message' is formed in this communication.

A multicomputer (two computers) utilizing message passing for communication



Shared Memory communication on an SMP



Review 1.1



- 1.1.1 File x.c is stored on Secondary Memory
- 1.1.2 After gcc x.c -o x, file x is stored on Secondary Memory
- 1.1.3 When we do ./x program(process) x is in Main Memory
(process: program in execution!)
- 1.1.4 Laptop has 1 processor with 8 cores.
Processor is a multicore processor
- 1.1.5 In 1999 I bought a computer (one box) with two identical CPU sharing the same main memory. My system was an SMP
- 1.1.6. I have a cluster with 16 computers. Each computer is of the same type as any other.
System is a Multicomputer



Components 1, 2, 3

and their interaction

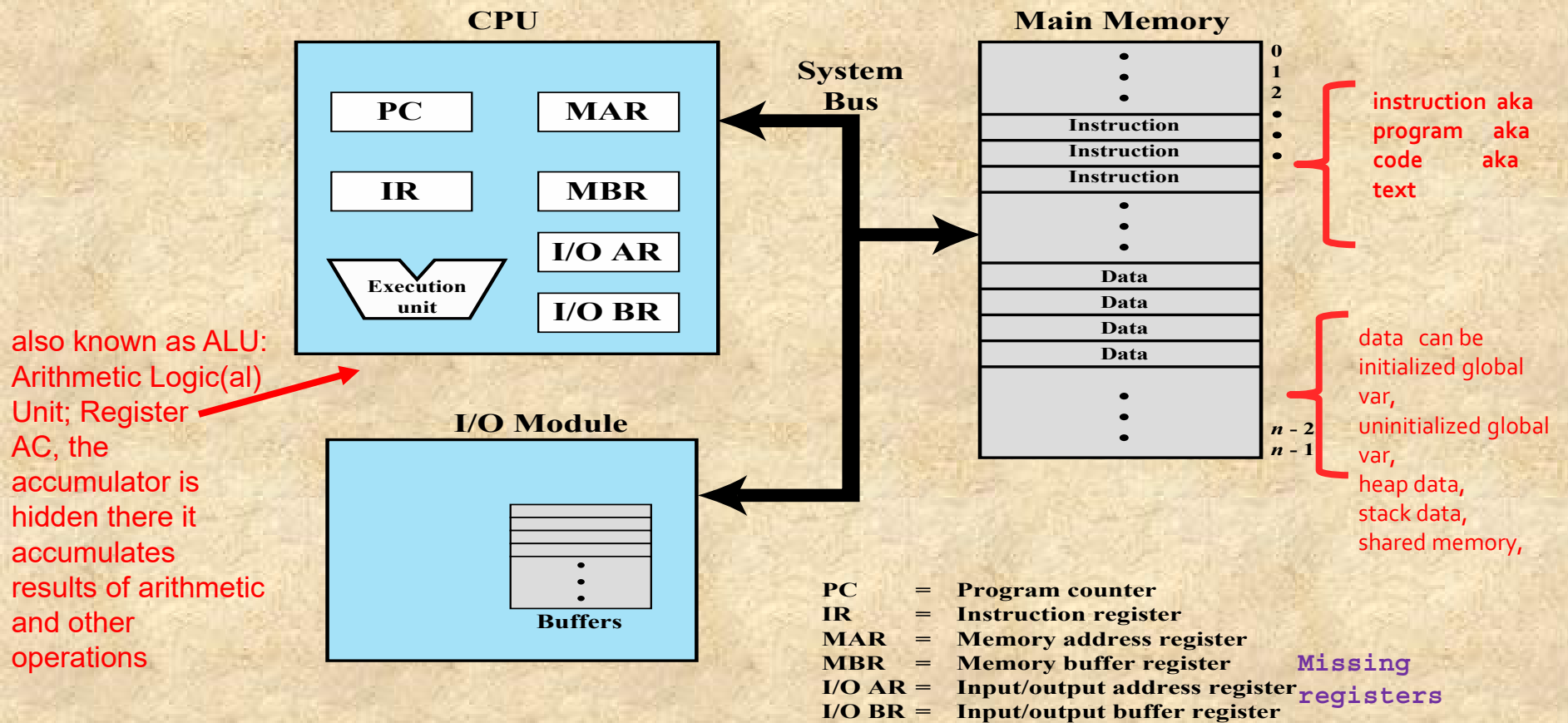


Figure 1.1 Computer Components: Top-Level View



Register: Fast Named memory on the CPU (Few registers with names!)

Register use speeds up computations : No need for CPU to access memory
(1ns instruction cycle vs 80-100ns for M[A])

Registers (textbook architecture) are

PC : Program Counter (address of instruction to be **fetch**ed **NEXT**)

PSW : Processor status word i.e. processor's state (control center)

IR : Instruction Register (holds instruction that is being executed)

AC : Accumulator (part of ALU/Execution Unit, usually)

MAR, MBR : Memory Address and Memory Buffer Register to communicate with main memory. For a read operation from memory MAR stores A and after the read operation MBR will store M[A].

I/O AR , I/O BR : I/O Address and I/O Buffer Register to communicate with I/O modules

BP,SP : introduced (stack pointer) Base stack Pointer, top of Stack Pointer
(hidden inside ALU/Execution Unit...)

10 registers total ...



Textbook Architecture

PC

AC

IR

SP

BP

MAR, MBR, I/O AR, I/O BR

PSW

~ 10 registers

Intel

register file: collection of registers

16bit: no prefix, 32bit: E prefix, 64bit: R prefix

(Intel mnemonics B: base as in BX, C: counter as in CX, S or D: source/destination as in RSI RDI)

PC or RIP: Instruction Pointer

RAX

RBX, RCX, RDX, RSI, RDI

RSP

RBP

R8-R15

RFLAGS,

XMM0-XMM15 (MMX registers)

~ 18 register+ 16 special purpose =

~34 registers

530	00FD
531	0044
532	000F
533	0001
534	00FF
535	001D
536	0026
537	0003

BTW: Intel has more 'hidden' registers!

Processor State

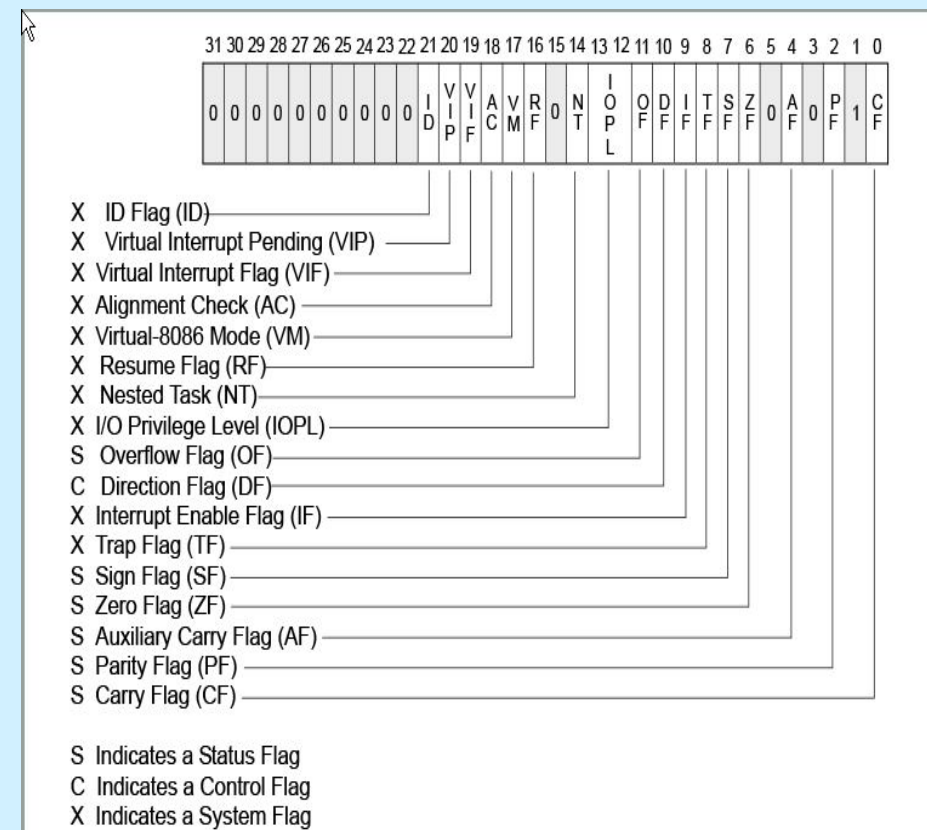
Textbook's PSW (Processor status word)

contains similar info
 to RFLAGS (64-bit) or
 EFLAGS (32-bit or lower
 32-bit of RFLAGS)

Intel's RFLAGS (previously EFLAGS)

<https://sipb.mit.edu/iap/6.828/slides/lec-1b.html#/eflags-register>

See also Page 126



Instruction Execution

- A program consists of a set of instructions stored in memory

Processor reads
(fetches) instructions
from memory

The diagram consists of two red rounded rectangular boxes connected by a white arrow pointing from left to right. The first box contains the text 'Processor reads (fetches) instructions from memory' and the second box contains 'Processor executes each instruction'. The entire diagram is set against a light beige background with a subtle pattern.

Processor executes
each instruction

```

START:                PC=init-value;           //OS loader decides or User program
                    while (1) {
/* Fetch Stage */      IR = M[PC]; PC++;
/* Execute Stage */    exec(IR) ;if IR==HALT goto HALT-ADDRESS
                    }
HALT-ADDRESS:         exit()

```

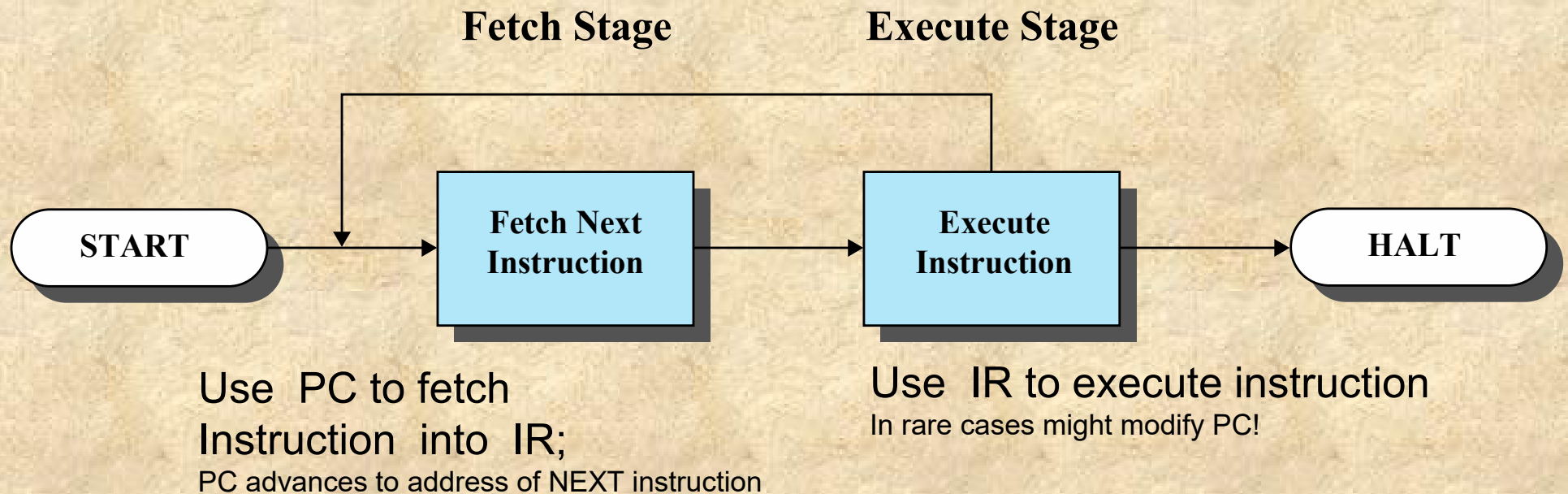


Figure 1.2 Basic Instruction Cycle

2 stage execution: Fetch+Execute

```

START:                PC=init-value;           //OS loader decides or User program
START-WHILE:          while (1) {
/* Fetch Stage */      IR = M[PC];PC++;
/* Execute Stage */    exec(IR) ;if IR==HALT goto HALT-ADDRESS;
/* Interrupt Stage*/   if InterruptsDisabled goto START-WHILE
                       checkForInterrupt; initInterruptHandler;
                       }
HALT-ADDRESS:          exit()

```

Fetch Stage Execute Stage Interrupt Stage

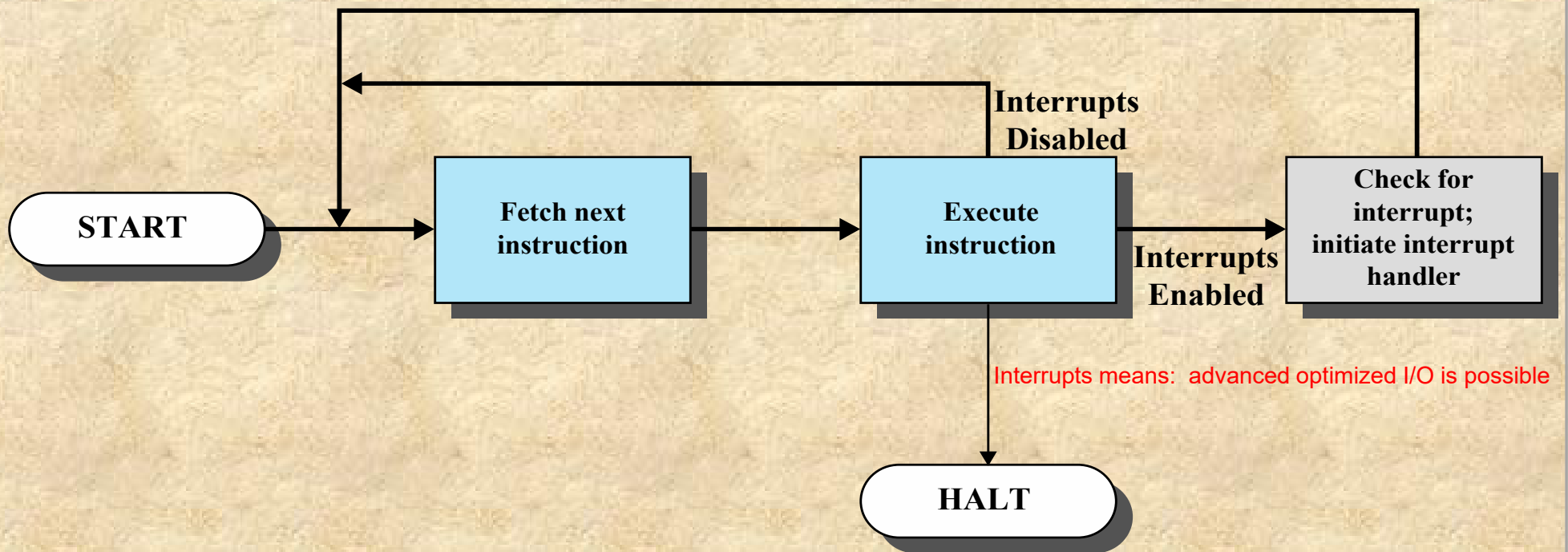


Figure 1.7 Instruction Cycle with Interrupts

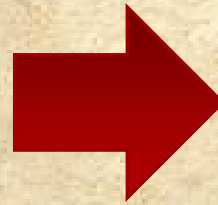
Modern-era: Selective 2-stage or 3-stage execution
based on Interrupts Disabled(2-stage) or Enabled (3-stage)

Instruction Fetch and Execute

- The processor fetches an instruction from memory
- Typically the program counter (PC) holds the address of the next instruction to be fetched
 - PC is incremented after each fetch **to point to the next instruction to be executed (in the next cycle)**

Instruction Register (IR)

Fetches instruction is loaded into Instruction Register (IR)



- Processor interprets the instruction and performs required action:
 - Processor-memory
 - Processor-I/O
 - Data processing
 - Control

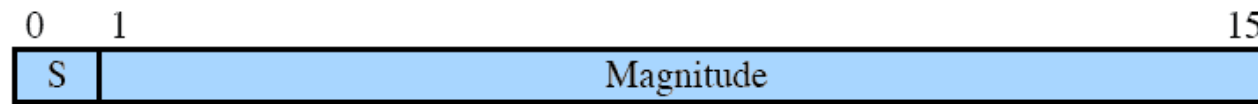
Note 1: One instruction cannot fit into one byte. An instruction will need two bytes.

ISA

Instruction
set
architecture



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0x1

0x2

0x5

0001 = Load AC from memory

0010 = Store AC to memory

0101 = Add to AC from memory

$AC = M[Address]$

$M[Address] = AC$

$AC = AC + M[Address]$

(d) Partial list of opcodes

Opcode: 4bit
i.e. 1 hex digit
Address: 12bit
i.e. 3 hex digit

Total 16bit

Data: 16bit
i.e.
4 hex digit

12 bit address
mean we can
address
 $2^{12}B$ i.e. all
of memory is
4096 two-byte
word (half
words)
i.e. a
total of
 $8192B = 8KiB$

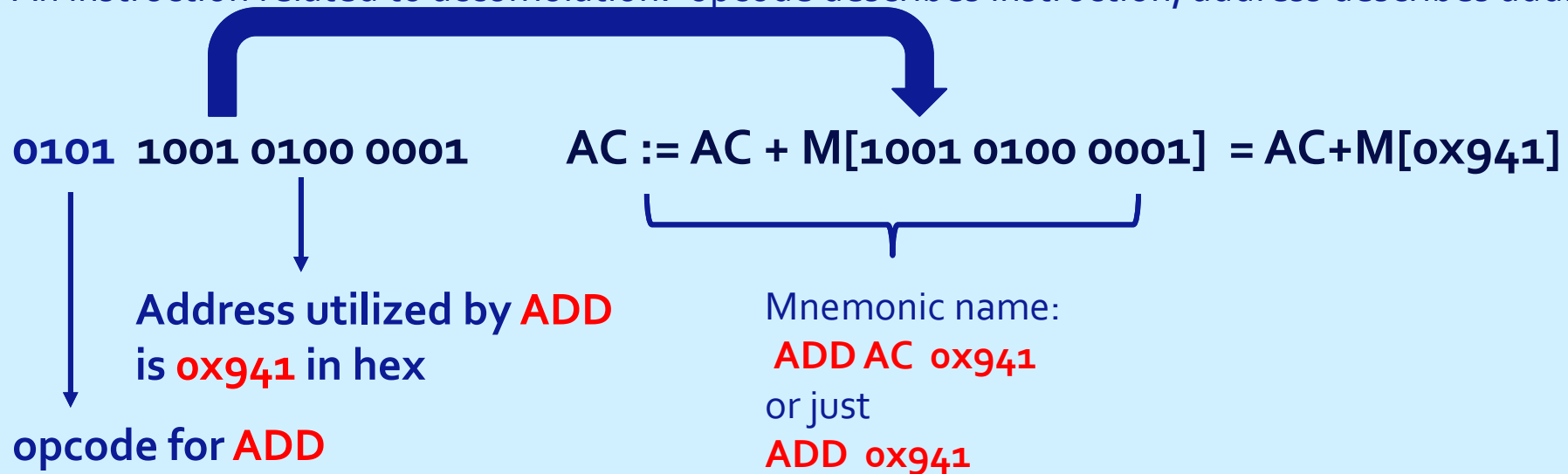
Figure 1.3 Characteristics of a Hypothetical Machine

AC is called an accumulator because it is capable of doing an accumulation :

accumulation: $AC := AC \text{ op } M[\text{address}]$

accumulation operator is an instruction with code op, where op can be +, -, *, etc but also AND, OR, XOR.

An instruction related to accumulation: opcode describes instruction, address describes address





0001 = Load AC from memory i.e. $AC=M[Address]$

LOAD AC Address

0010 = Store AC to memory i.e. $M[Address]=AC$

STOR AC Address

0101 = Add to AC from memory $AC=AC+M[Address]$

ADD AC Address

Instruction is 16bit (2B): Leftmost 4 bit is opcode; Rightmost 12bit is Address!

Additional List of Opcodes ... continues Figure 1.3 of textbook

0011 = Load AC from I/O

$AC=I/O[Address]$

(from what device number/port?)

0111 = Store AC to I/O

$I/O[Address]=AC$

(to what device number/port?)

Address next to opcode for above identifies a particular device's number/port

Task: Evaluate 2+3 (Use Fig 1.4)

0001 1001 0100 0000

LOAD AC 0x940 // load into AC a 3

0101 1001 0100 0001

ADD AC 0x941 // Add to AC a 2

// contents of AC is a ?

Review 1.2



- 1.2.1 How many bit for a hexadecimal digit? 1bit $2^{*1}=2$ states;
4bit $2^{*4}=16$ states
Thus 4bit to represent
0..9,a..f or 0..9,A..F
- 1.2.2 How many bit for an opcode? 4bit
- 1.2.3 How many possible opcodes? $2^{*4} = 16$
- 1.2.4 How many bit for an address? $16 - \# \text{bit of opcode} = 16 - 4 = 12 \text{ bit}$
- 1.2.5 How much memory can be addressed? $2^{*12} = 4096$
- 1.2.6 How many 'byte' is an addressable memory? Traditionally 1B
But per textbook 16bit = 2B
- 1.2.7 How many bytes of Memory can we address? $2^{*12} * 2B = 8192B = 8KiB !$
- 1.2.8 Memory location 301 in slide 52
5941 is 4hex digit=16bit
indicates a 5941. What is it? **First hex digit 5= 0b0101**
- 1.2.9 What is the binary form of address 301?
(Hint look at bottom of previous slide) **is opcode for ADD; for ADD**
rest of opcode, 941, is an
address!
- THUS NO ANSWER for 1.a.9

everything is
in
hex

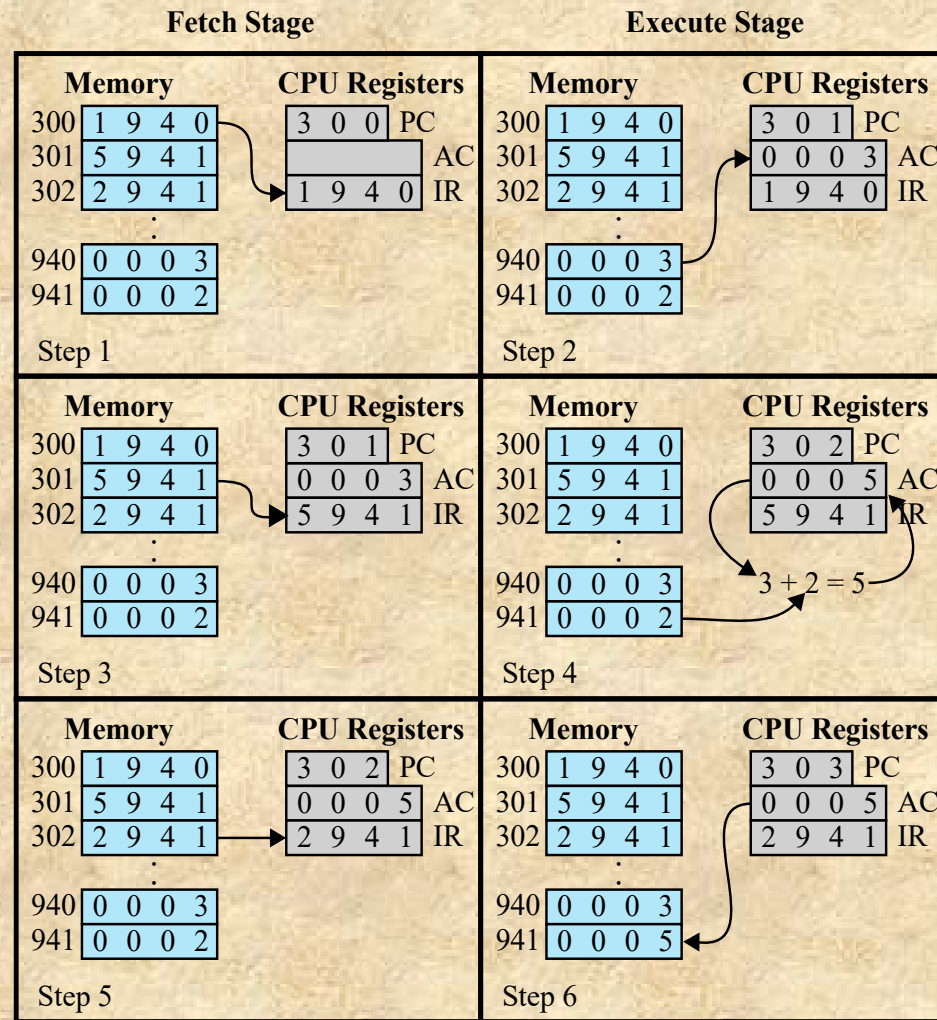


**Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)**



Textbook

register file

PSW, (processor state)

PC

IR

AC,

MAR, MBR,

I/O AR, I/O BR

BP, SP

plus

text + data (includes
stack)

Intel

register file

RFLAGS, (processor state)

RIP

IR (not accessible)

RAX,

RBX, RCX, RDX

RSP, RBP

RSI, RDI

R8-R15

XMM0-XMM15

plus

text + data (includes
stack)



To suspend execution of a **processor**,

- a. wait for current instruction to complete (its execution)
- b. save/store 'register file' (selected registers) into MEMORY (STACK)
- c. along with processor state (**PSW**)

To resume execution of a **processor**

- a. wait for current instruction to complete
- b. load/restore 'register file' from MEMORY (STACK)
- c. along with restored processor state (**PSW**)

Modern microprocessors have set of instructions for step (b) and (c) of suspend and resume (eg save PC and PSW, save everything else)

register file: contents (values) of all relevant (selected) registers used in program's execution



To suspend execution of a program ,

- a. wait for current instruction of program to complete
- b. save/store 'register file' (selected registers) into MM or SM
- c. store processor state (PSW)
- d. save/store text+data into MM (already) or SM (if needed)

To resume execution of a program,

- a. wait for current instruction to complete
- b. load/restore 'register file' from MM or SM
- c. restore processor state (PSW)
- d. load/restore text+data from MM (not needed) or SM (if needed)
- e. PC is set already; ready to go (execute)

Modern microprocessors have set of instructions for step (b) and (c) of suspend and resume (eg save PC and PSW, save everything else)

Review 1.3



- | | | |
|-------|---|---|
| 1.3.1 | How many registers does a modern CPU have? (a) ≤ 64 , (b) 128-1024, (c) > 1024 | (a) |
| 1.3.2 | What are the most important registers to be saved /restored first during a suspension/resumption of activity? | PC, PSW |
| 1.3.3 | What register defines the processor's state? | PSW |
| 1.3.4 | What is a process state? | Register file (PC,PSW,etc)
code(text, program)
data, stack, heap, etc |
| 1.3.5 | What is another name for 'process'? | Program in execution |
| 1.3.6 | What is a 'program in execution' state? | See 1.3.4 (and 1.3.5) |



Module 2

I/O

Interrupts

- Mechanism by which other modules may interrupt the normal sequencing (context) of the processor
- Provided to improve processor utilization
 - Most I/O devices are slower than the processor
 - Processor must pause to wait for device
 - Wasteful use of the processor

Table 1.1 Classes of Interrupts

An interrupt is a hardware interrupt!

Program

Trap or Faults are
'Software interrupts'
also known as
Exceptions

Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.

Timer

An interrupt is a
hardware interrupt!
Timer is a piece of
hardware

Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.

I/O

An interrupt is a
hardware interrupt!
I/O Device is a piece of
hardware

Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.

Hardware failure

An interrupt is a hardware interrupt. A hardware failure generated interrupt is by a failing piece of hardware.

Generated by a failure, such as power failure or memory parity error.

Secondary Memory

**Also
referred to
as auxiliary
memory**

- **External**
- **Nonvolatile**
- **Used to store
program and
data files**

I/O Techniques

- When the processor encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module

Three techniques are possible for I/O operations:

Programmed
I/O

Interrupt-
Driven I/O

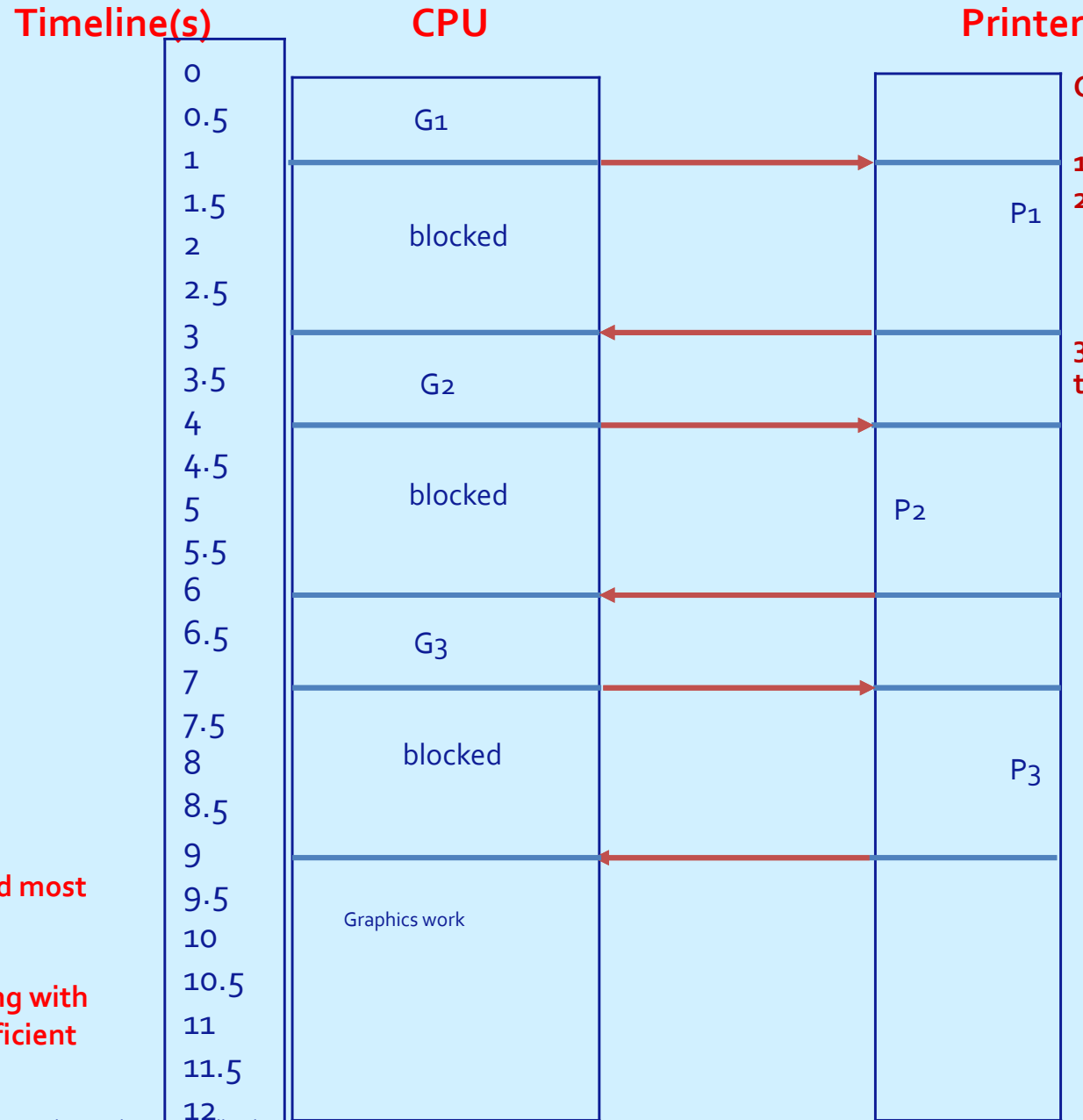
Direct Memory
Access (DMA)

Programmed I/O

also known as Blocked I/O

- The I/O module performs the requested action then sets the appropriate bits in the I/O status register
- The processor periodically checks the status of the I/O module until it determines the instruction is complete
- With programmed I/O the performance level of the entire system is severely degraded

Programmed (Blocked) I/O



Consider the task (or job):

Print 3 pages, one page at a time

1. Prepare a page to be printed : 1s G-task

2 Print page : 2s P-task

For an interrupt-based I/O

3 Resuming from an interrupt: 1/2s I-task

Other work: Graphics 3s non printing

Blocked I/O only G-tasks and P-tasks

While Printer is printing (eg P1) CPU is blocked (i.e. idle)

CPU utilization [0-12s] : $6/12 = 50\%$

Turnaround time of the Print Task/Job

involving all three pages is 9s and

CPU utilization [PrintJ] : $3/9 = 33\%$

Programmed/ Blocked I/O implies no interrupts are supported (or available) thus there is need for I-task execution!

1940s and most of 1950s

computing with very inefficient I/O

Interrupt-Driven I/O

Processor issues an I/O command to a module and then goes on to do some other useful work

The processor executes the data transfer and then resumes its former processing

The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor

More efficient than Programmed I/O but still requires active intervention of the processor to transfer data between memory and an I/O module

Interrupt-Driven I/O Drawbacks

- Transfer rate is limited by the speed with which the processor can test and service a device
- The processor is tied up in managing an I/O transfer
 - A number of instructions must be executed for each I/O transfer

Interrupt I/O

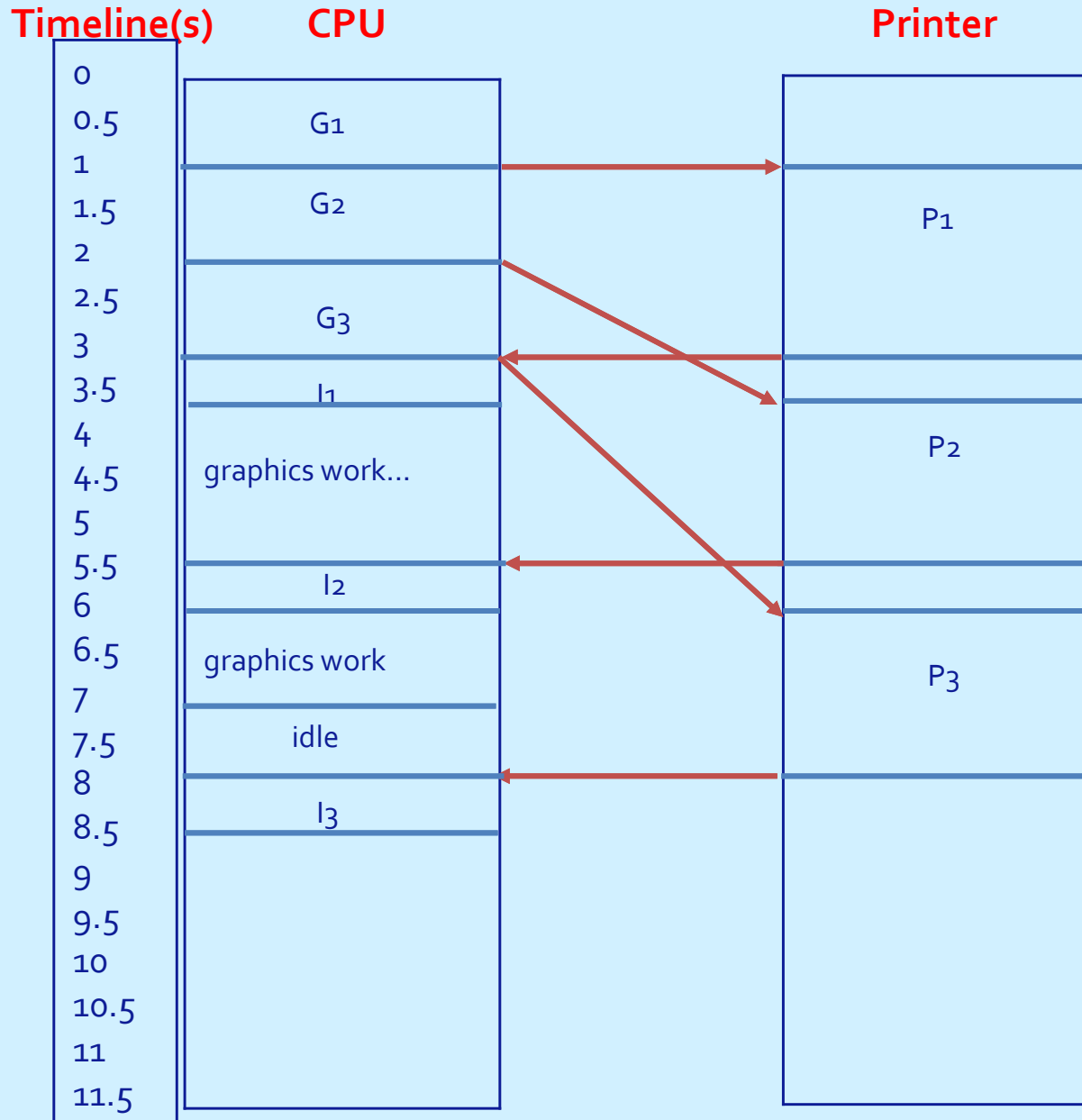
Turnaround,
 Elapsed
 sometimes used
 interchangeably.

Turnaround =
 Completion Time
 minus
 Start time.

If Start time = 0
 then
 Turnaround time
 is
 Completion time.

**Interrupt
 handling first**

**Maybe a HW
 question...**



Interrupt I/O

In addition to G-task, P-task, I-task overhead...

While Printer is printing (eg P1)
 CPU is working on G2, and then
 on G3 instead of being blocked.
 There are idle moments as well

CPU utilization [0-8.5s] : $6/8.5 = 70\%$ with graphics work included

Note : we exclude interrupt I-tasks even if they use the CPU; it is overhead. BTW idle is idle.

Turnaroundtime of the Print Job involving all three pages is 8.5s better than gs before

Direct Memory Access (DMA)

- Performed by a separate module on the system bus or incorporated into an I/O module

When the processor wishes to read or write data it issues a command to the DMA module containing:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read/write
- The number of words to be read/written

Direct Memory Access

- Transfers the entire block of data directly to and from memory without going through the processor
 - Processor is involved only at the beginning and end of the transfer
 - Processor executes more slowly during a transfer when processor access to the bus is required
- More efficient than interrupt-driven or programmed I/O

DMA



Turnaround,
Elapsed
sometimes used
interchangeably.

Turnaround =
Completion Time
minus
Start time.

If Start time = 0
then
Turnaround time
is
Completion time.

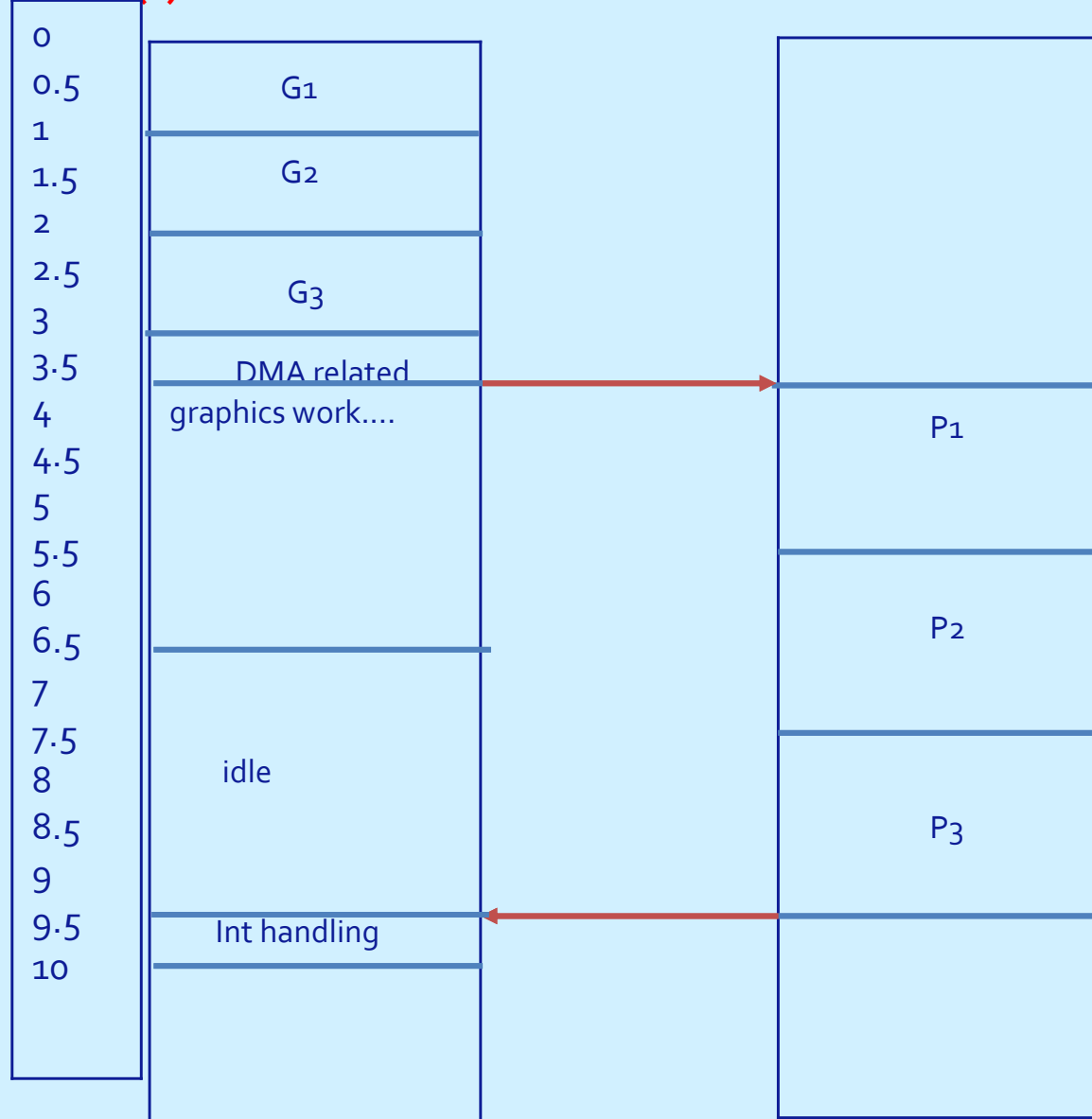
**Interrupt
handling first**

**Maybe a HW
question...**

Timeline(s)

CPU

Printer



Interrupt I/O

In addition to G-task, P-task, I-task overhead...

While Printer is printing (eg P1)
CPU is working on G2, and then
on G3 instead of being blocked.
There are idle moments as well

CPU utilization (0-10s) : $6/10 = 60\%$

Turnaround time of the Print
Job involving all three pages is
10s ; in the same time we do
the graphics work as well... for
free

Food for thought: What if
graphics work is 6sec not 3sec
for all three cases?

```

START:          PC=init-value;          //OS loader decides or User program
START-WHILE:    while (1) {
/* Fetch Stage */      IR = M[PC];PC++;
/* Execute Stage */    exec(IR) ;if IR==HALT goto HALT-ADDRESS;
/* Interrupt Stage*/   if InterruptsDisabled goto START-WHILE
                        checkForInterrupt; initInterruptHandler;
                        }
HALT-ADDRESS:    exit()

```

Fetch Stage Execute Stage Interrupt Stage

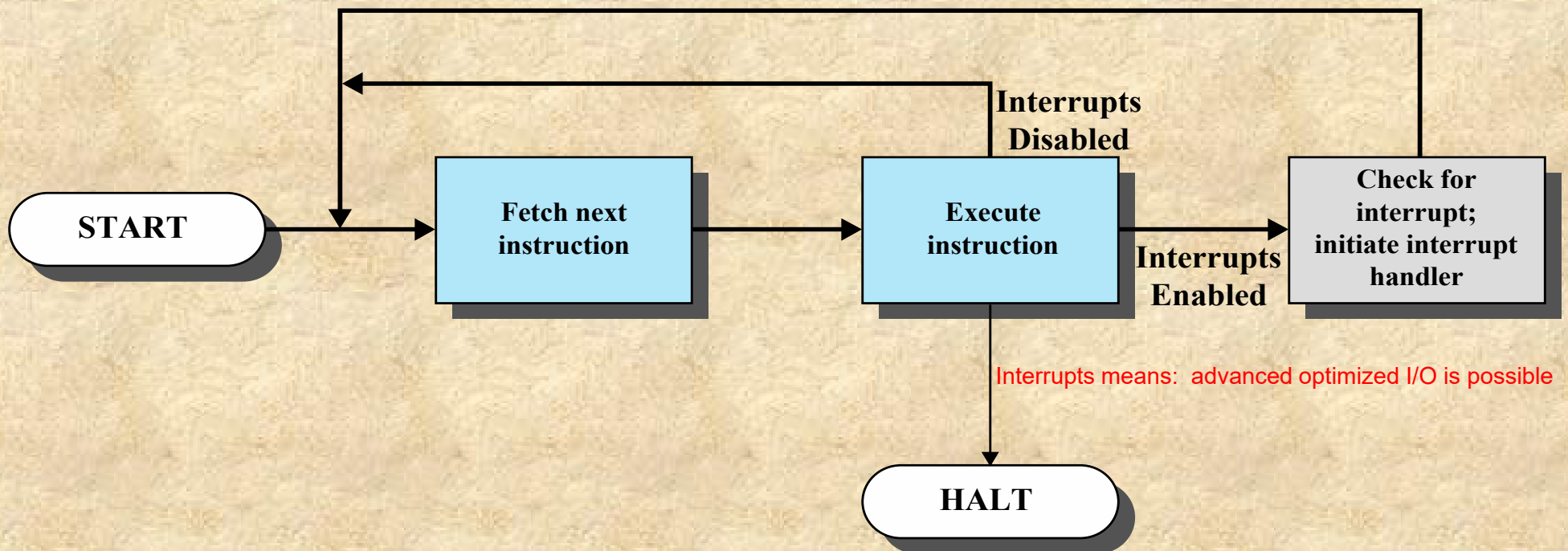


Figure 1.7 Instruction Cycle with Interrupts

Modern-era: Selective 2-stage or 3-stage execution
based on Interrupts Disabled(2-stage) or Enabled (3-stage)

Context 1 **User Program** **context switch** **Interrupt Handler** **Context 2**

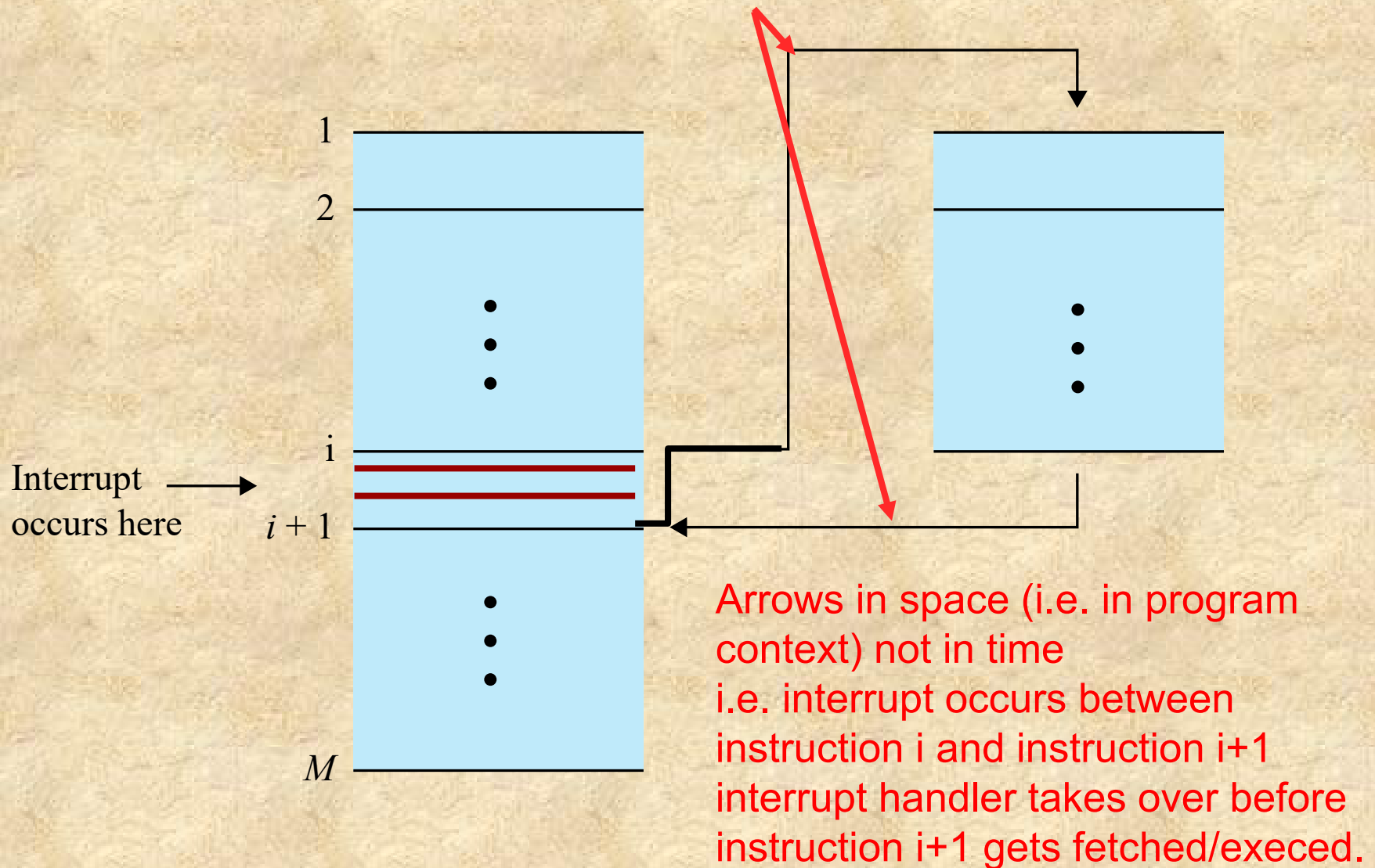
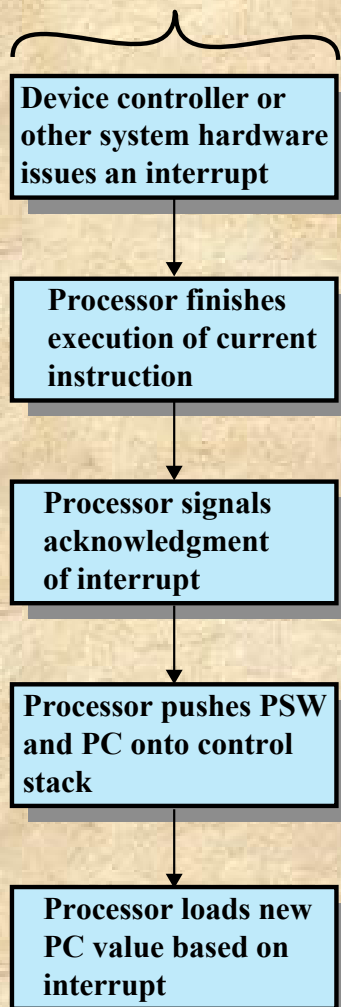


Figure 1.6 Transfer of Control via Interrupts

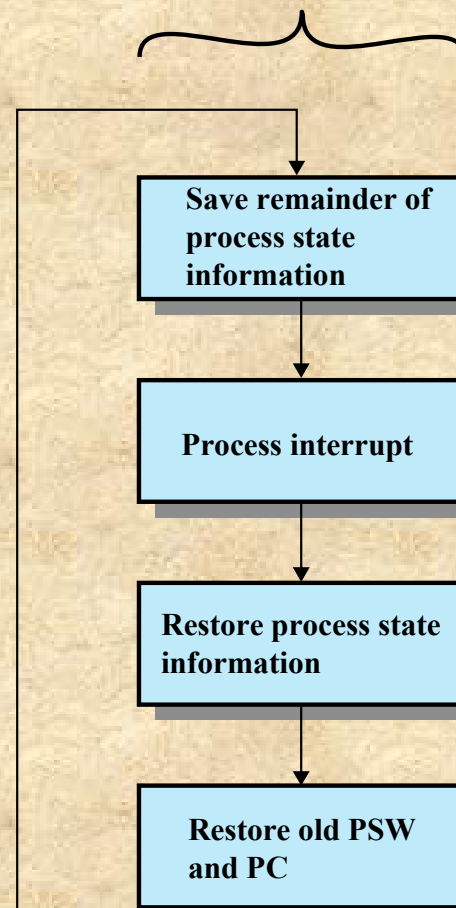
Hardware



Processor state first
`sPUSH(PSW);sPUSH(PC)`
Value of PC pushed is $i+1$

PC = 0 ?
old times interrupt handler
was at address 0!

Software



`PUSH Register file - PC - PSW`
(in practice selected registers are saved)

`POP Processor file - PC - PSW`

`sPOP(PSW); sPOP(PC)`
(i.e. PC = $i+1$)

Figure 1.10 Simple Interrupt Processing

ALARM!!
Fig 1.11
contradicts
Fig 1.10

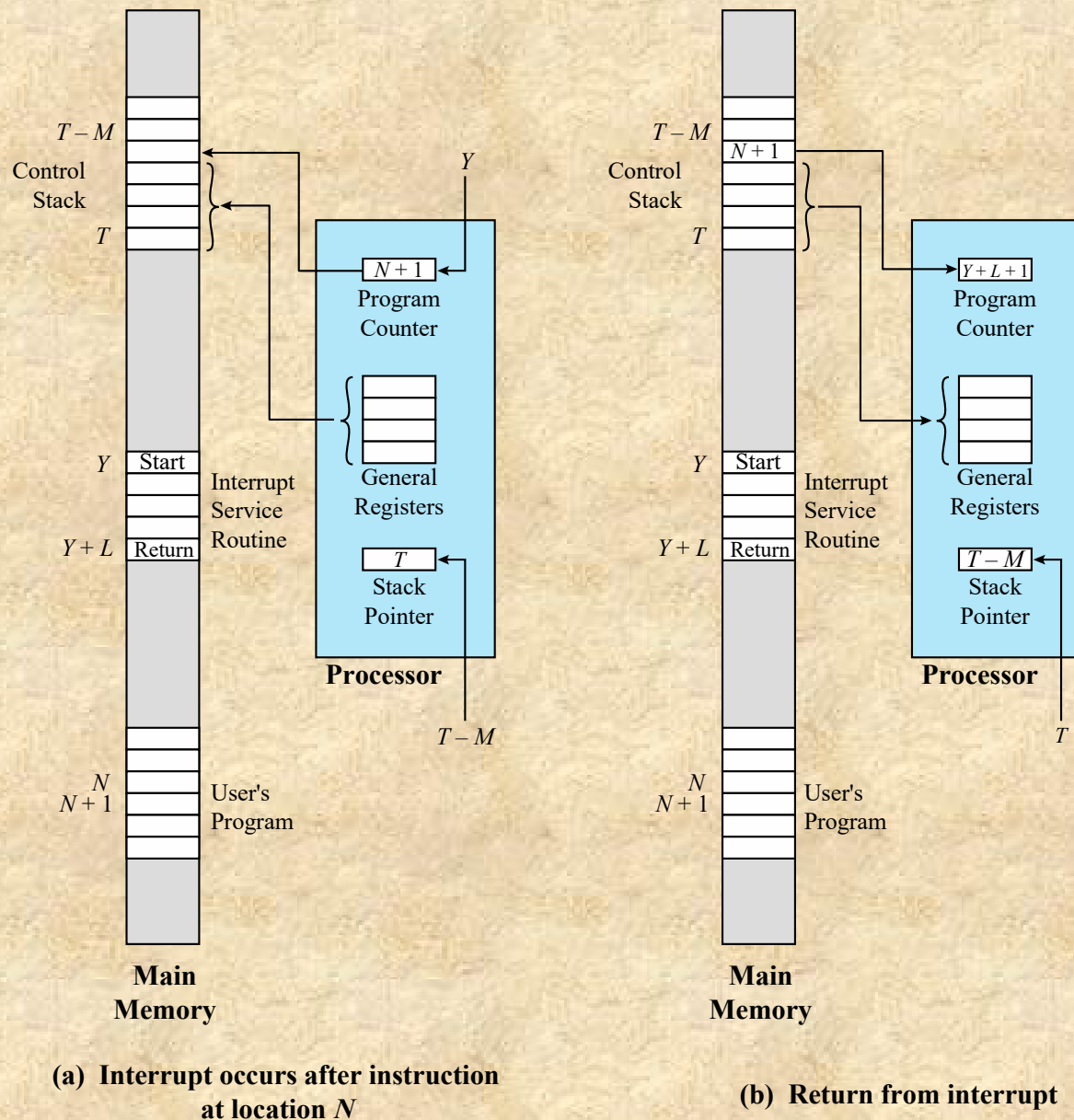


Figure 1.11 Changes in Memory and Registers for an Interrupt



A program about to become a program in execution (i.e. a process) has

text (also known as **instructions** or **code**),

initialized data

uninitialized data (also known as **bss**)

heap data for dynamically allocated data

stack data for function call invocation

shared (memory) data to be shared

i.e. `int *x= new int(1);`

i.e. `z= f(x,t);`

i.e. `creat_sh_mem(8)`

The stack above sometimes it being referred to as

user stack (data)

since it stored info re the execution of regular user function calls in user-mode (Chap 2) . Other function calls (eg system calls supported by OS) are executed in kernel-mode (aka privileged mode, monit-mode, system-mode) by a separate

kernel stack (data)

as explained in Chapters 2 and 3

Q1 Who turns an (executable) program into a program in execution (i.e. process)? Ans: The loader which loads the program from secondary memory into main memory.

Q2 Who 'owns' or 'controls' the loader? The OS

Multiple Interrupts

An interrupt occurs while another interrupt is being processed

- e.g. receiving data from a communications line and printing results at the same time

Two approaches:

- Disable interrupts while an interrupt is being processed
- Use a priority scheme



Module 3

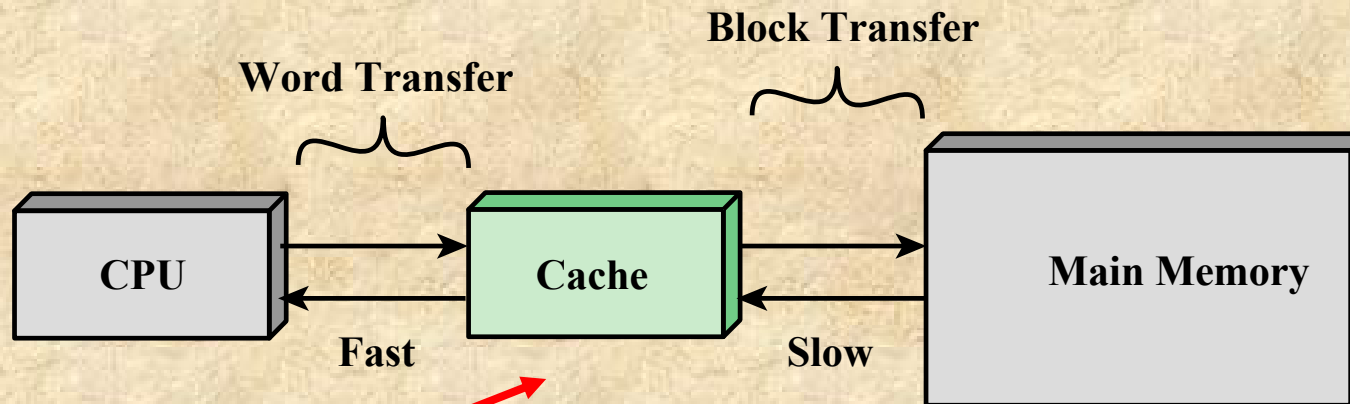
Caches



Is Cache associated with
Component 2: Processor
or
Component 1: Main Memory ?

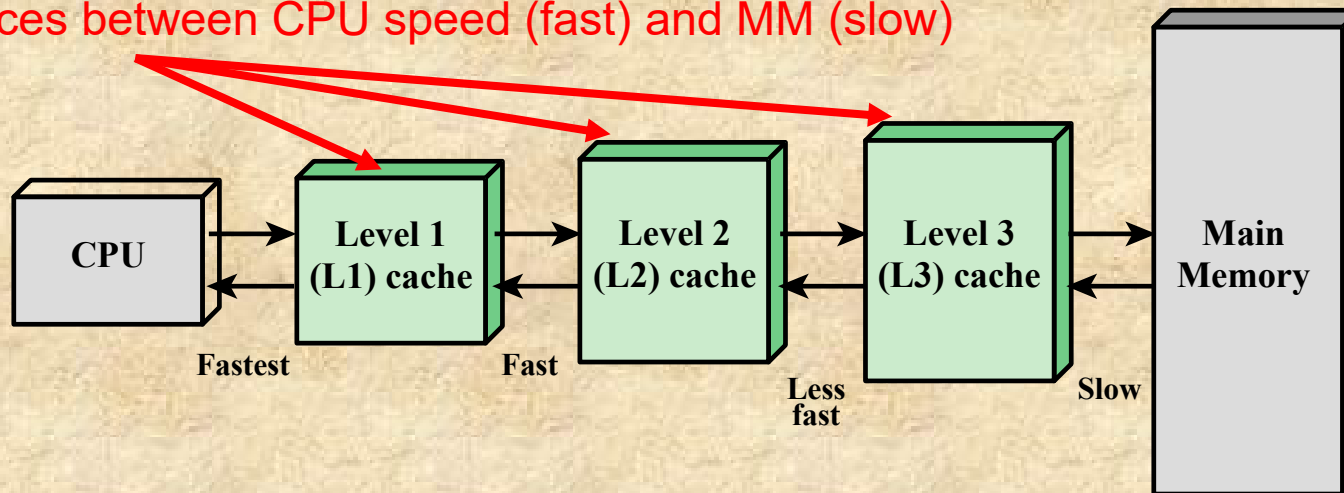
Memory Hierarchy

- Design constraints on a computer's memory
 - How much?
 - How fast?
 - How expensive?
- If the capacity is there, applications will likely be developed to use it
- Memory must be able to keep up with the processor
- Cost of memory must be reasonable in relationship to the other components



(a) Single cache

Cache is buffer (auxiliary memory) that evens out differences between CPU speed (fast) and MM (slow)

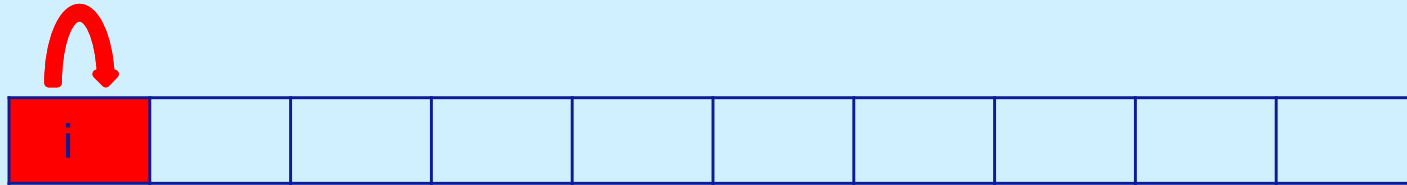


(b) Three-level cache organization

Figure 1.16 Cache and Main Memory

Principle of Locality

- Memory references by the processor tend to cluster **i.e. future memory accesses are close to past accesses**
- Data is organized so that the percentage of accesses to each successively lower level is substantially less than that of the level above
- Can be applied across more than two levels of memory



Temporal:

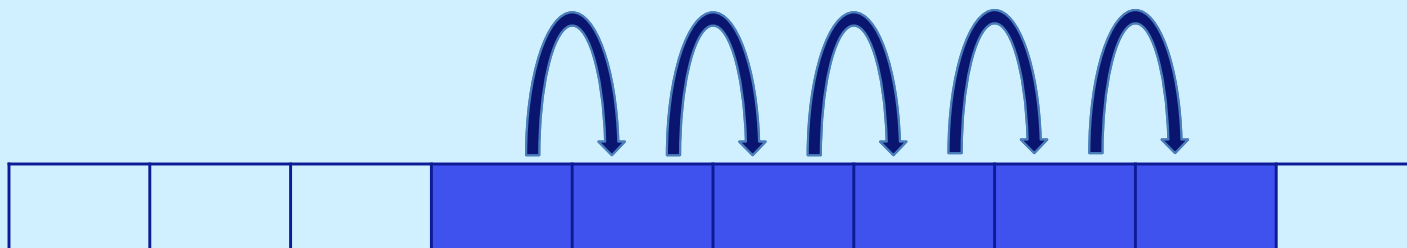
for(; i < n ;)

for(; ; i++)

[i]

Spatial:

... M[10] M[11] M[?] ...





Locality of (memory) reference can be about

(a) data, and

```
for (i=0 ; i < n ; i++ ) sum = sum +M[i];
```

(b) programs

```
temp = a ; a=b ; b = temp;
```

It can also be

Temporal Locality

recently referenced data more likely to
be referenced again (sum, i)

Spatial Locality

see above (temp,a,b and M[i-1]..M[i].. M[i+1] etc)

Issue/Problem: Register access : 1 cycle, Mem access: 100cycle

Questions



- Question 1:** How many times is `i < n` instruction executed?
- Question 2:** How many times is `i++` instruction executed?
- Question 3:** How many times is `i` read in `i < n` and `i++`, `M[i]`
- Question 4:** How many times is `i` written in `i < n`, `i++`, `i = 0`, `M[i]`
- Question 5:** Do we need `i < n` say in Intel's L1I?
- Question 6:** Repeat 5 for the other instructions
- Question 7:** Do we need `i` in Intel's L1D?
- Question 8:** Repeat 7 for `n`
- Question 9:** the loop's `i` is currently equal to `n-50`. Give 8 addresses of data that should be in the cache (eg L1D) in the next 8 cycles!

Cache Memory

No Intel CPU instruction for L1[A] or L2[A] or L3[A]

- Invisible to the OS
- Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
- Processor execution is limited by memory cycle time
- Exploit the principle of locality with a small, fast memory



Cache (memory)

Cache: Fast, Expensive, Small-in-size memory, close to the CPU than Main Memory (RAM); sometime on the CPU's chip

Instruction Cache: Cache that holds only instructions (eg. Intel's L1I cache)

Data cache: Cache that holds only data (eg Intel's L1D cache)

[Separation of data and instructions : Harvard model]

Unified cache: Cache that holds both instructions and data (L2, L3, and it is also the case for main memory nowadays)

[Same space for data and instructions: von-Neumann model]



Issue/Problem: Register access : 1 cycle, Mem access: 100cycle

3.1.1	We execute 1000 instructions. How many cycles do they take, if instructions do not reference MM?	1000 cycles
3.1.2	We execute 1000 instructions. Each one of them references during its execution stage MM. How many cycles to complete them?	100,000 cycles
3.1.3	Elapsed time of 3.1.1	1000 cycles
3.1.4	Elapsed time of 3.1.2	100,000 cycles.
3.1.5	CPU utilization for 3.1.1	$1000/1000 = 100\%$
3.1.6	CPU utilization for 3.1.2	$1000/100,000 = 1\%$

Conclusion: Modern CPUs face the 1% nightmare scenario!

Some definitions



Note: Below we assume we have a two-level memory structure: fast and slow. Alternatively, we could use those definitions for cache and main-memory, main-memory and disk, or other multi-hierarchical structure (L1, L2 and main-memory). Next page it is L1(fast), L2(slow)

Hit: An access where data is located in the 'fast' (fast memory)

Miss: An access where data is NOT located in the 'fast' (it's in slow memory)

Hit time(HT): Time to access the 'fast' (fast)

Miss time Penalty(MT): Time to move data to the 'fast' and then
CPU (usually fast + slow)

Hit ratio (HR): fraction of time data is located in the 'fast'

Miss ratio(MR): 1 minus hit ratio (or 100 minus hit ratio(%))



Example 1: L1 and L2

Some data from a previous slide. Register(CPU): 1 cycle, L1: 5 cycle, L2: 15 cycle CPU → L1 → L2

Example 1. Simple Two level hierarchy : L1 (fast) and L2 (slow)

A memory reference is either in L1 or L2. Thus 100% of the time we need no access to Main Memory. For an address D, we first access/check L1; if there, DONE; else we have wasted an L1 access and we head to L2; it will always be there.

95% of memory references are to L1 (Hit ratio: $HR = 95\%$ or 0.95)

5% of memory references are to L2 (Miss ratio: $MR = 5\%$ or $0.05 = 1 - HR$)

Hit time (HT): 5 cycle (L1),

Miss time penalty (MT): 5 + 15 cycle (L1 + L2)

Average # cycles is [Effective Memory access rate] = $95\% * 5 + 5\% * (5 + 15) = 5.75$ cycle

5.75 is closer to the 5 of L1 and much smaller than the 15 of L2

Note : register speed is 1 cycle (i.e. 5X faster than L1!)

Effective Memory access time (in ns)

Effective Memory access rate (in cycle)

May be used interchangeably!



Example 2: L1, L2, MM

Example 2: Not very simple : three-level hierarchy!

L1 cache: 95% hit , 5 cycle , L2 cache: 99% hit , 15cycle, L2 MT : 45 cycle, MainMemory: 200cycle

95% hit into L1: 5cycle total: 5 cycle

5% miss wasted L1 access: 5cycle ---> 99% hit into L2: 15cycle total: 5+15cycle

1% miss wasted L2 access: 15cycle, L2 miss serv: 45cycle + MM total: 5+15+45+200cycle

Eff Mem access rate = (L1hit) + (L1missL2hit) + (L1missL2missMM) =

$$= 0.95 * 5[L1hit] + 0.05 * 0.99 * (5+15) [L1missL2hit]$$

$$+ 0.05 * 0.01 * (5+15+45+200) = 5.8725$$

Answer is still close to L1 than L2 or MM

95% in L1 (fast)

5% not in L1 but 99% of 5% in L2 (slower)

5% not in L1 but 1% of 5% in MM (slowest)



Let L1 HR $h_1 = 50\%$ and HT $t_1 = 1\text{cycle}$.

CPU $\leftrightarrow$ L1 $\leftrightarrow$ L2 $\leftrightarrow$ L3 $\leftrightarrow$ MM

L2 HR $h_2 = 75\%$ and HT $t_2 = 10\text{cycles}$.

L3 HR $h_3 = 90\%$ and HT $t_3 = 60\text{cycles}$.

MM HR $h_4 = 100\%$ and HT $t_4 = 120\text{cycles}$.

What is the effective memory access rate EMAR?

Show that $EMAR = t_1 + (1-h_1)t_2 + (1-h_1)(1-h_2)t_3 + (1-h_1)(1-h_2)(1-h_3)t_4$.

Proof. Hint : Answer is ...15cycles (if I am not wrong)...

$$EMAR = T_1$$

$$T_1 = h_1 \cdot t_1 + (1-h_1)(t_1 + T_2)$$

$$T_2 = h_2 \cdot t_2 + (1-h_2)T_3$$



1. A cache stores blocks of memory never individual bytes (eg pages of memory using Subject 0 notation)
 Let address of memory be $x=32\text{bit}$ (8hex digit). Let D be an arbitrary address of 32-bit.
 PageNumber $\rightarrow$ BlockNumber
 PageOffset $\rightarrow$ BlockOffset
2. Block of memory of size K (block length) is stored in a line of the 'cache'. Individual bytes of a block (K of them) addressable using $\lg K$ rightmost bit of address.
3. Number of lines in a cache is C thus an index uses $\lg C$ bit. Line number is similar to a Page number of Subject 0 but uses bit on the left of the $\lg K$ bit of offset.
4. Line number: $\lg C$ bit, Block Offset: $\lg K$ bit
 What about the remaining $x - \lg C - \lg K$ bit? They become a tag and stored in cache line along the K -long block!
5. **PROBLEM:** I want to store in the cache the block containing address $0x99912349$.

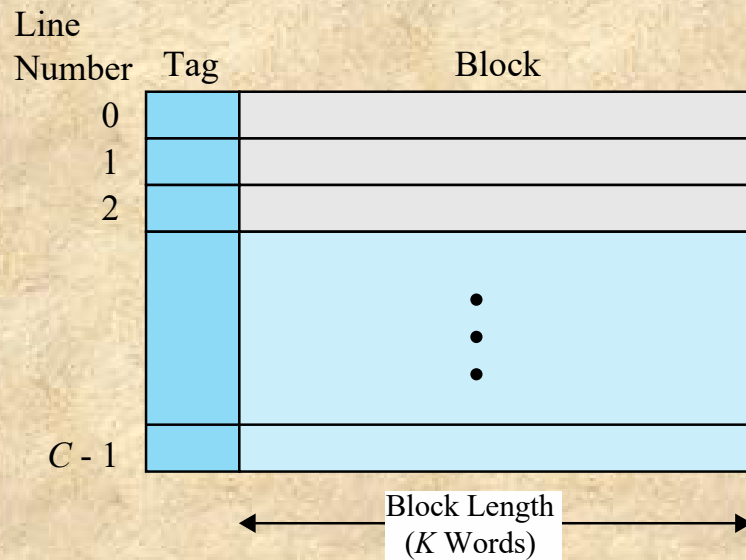
1. $x=32\text{-bit}$ Let $D: 0x \text{ 4FF12345 } 32\text{bit}$
 $D: 01001111111100010010001101000101$

2 Let $K=16$; $\lg K=4$ (1hex digit)
 Subj. 0 BlockNumber: $4FF1234$ 28bit
 BlockOffset: 5 4bit

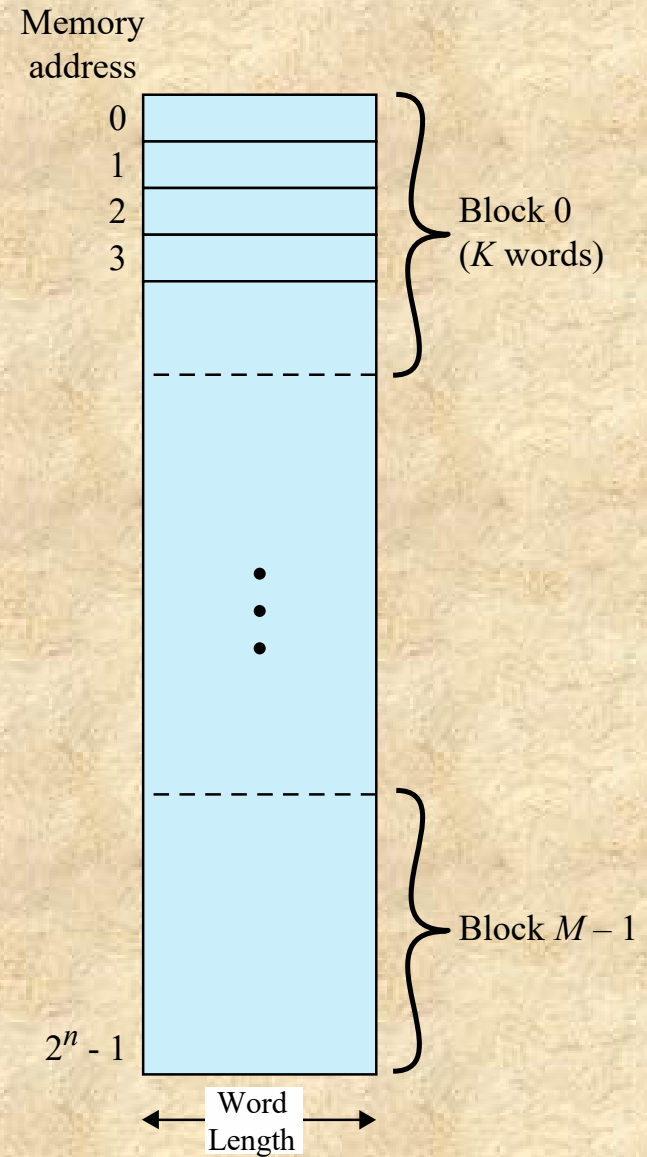
3. Let $C=4096$, $\lg C=12$ (3hex digit)
 Cache line index: 234 12bit
 aka Cache line offset

4. $32 - 4 - 12 = 16\text{bit}$ for tag
 Tag is: $4FF1$ 16bit
 In line $0x234$ of the cache we store a block of 16B starting with the byte at address $0x4FF12340$ and ending with the byte at $0x4FF1234F$ plus the tag $4FF1$. Cache line length is $16B + 4 \times 4\text{bit}$
 for tag = $16 \times 8 + 16 = 144\text{bit}$

5. $0x99912341$ must be stored in cache line 234 as well which is occupied as it stores the contents of $4FF12345$. **Solution:** Have an extra copy of this 'cache architecture' or evict previous block



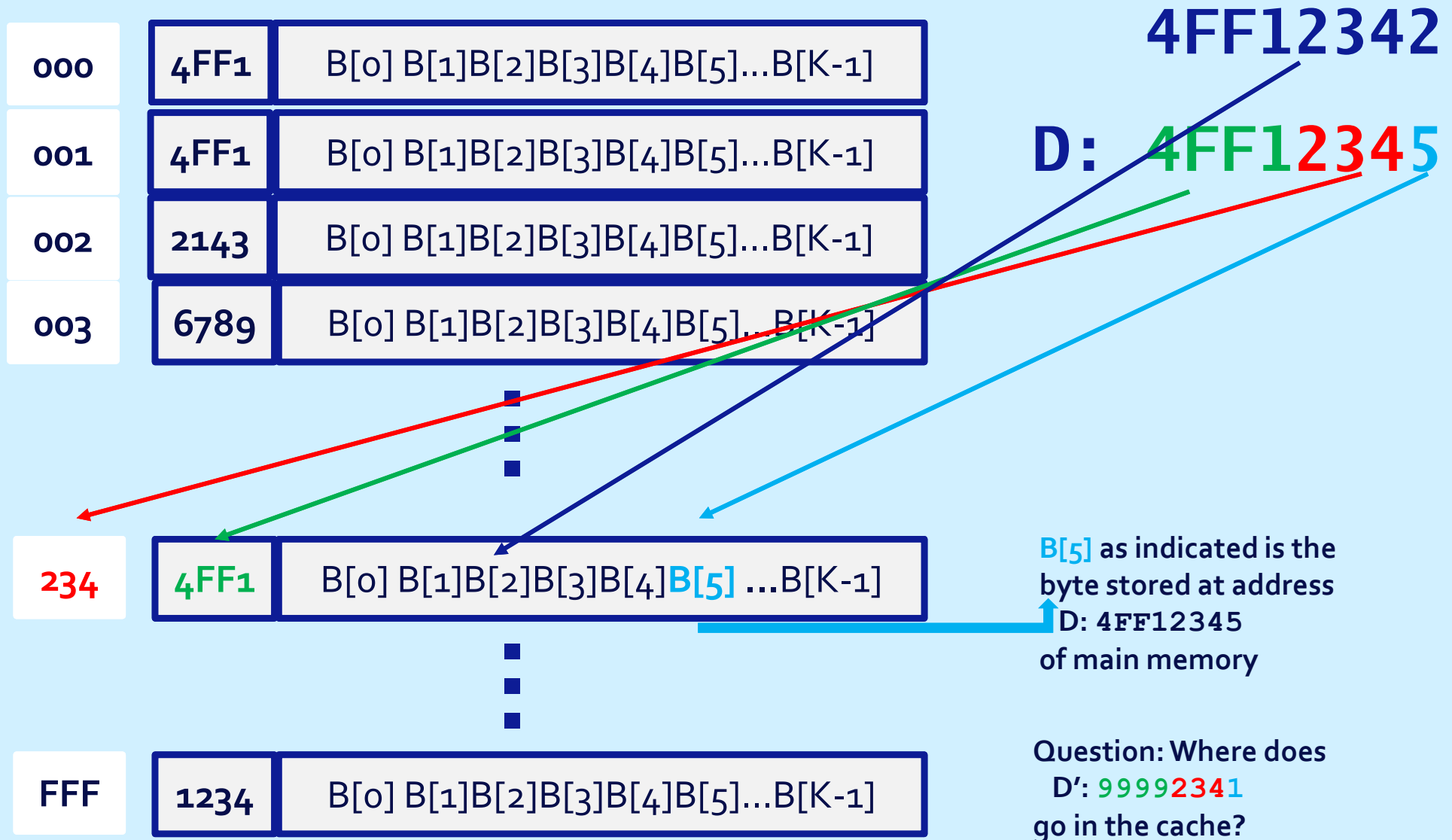
(a) Cache



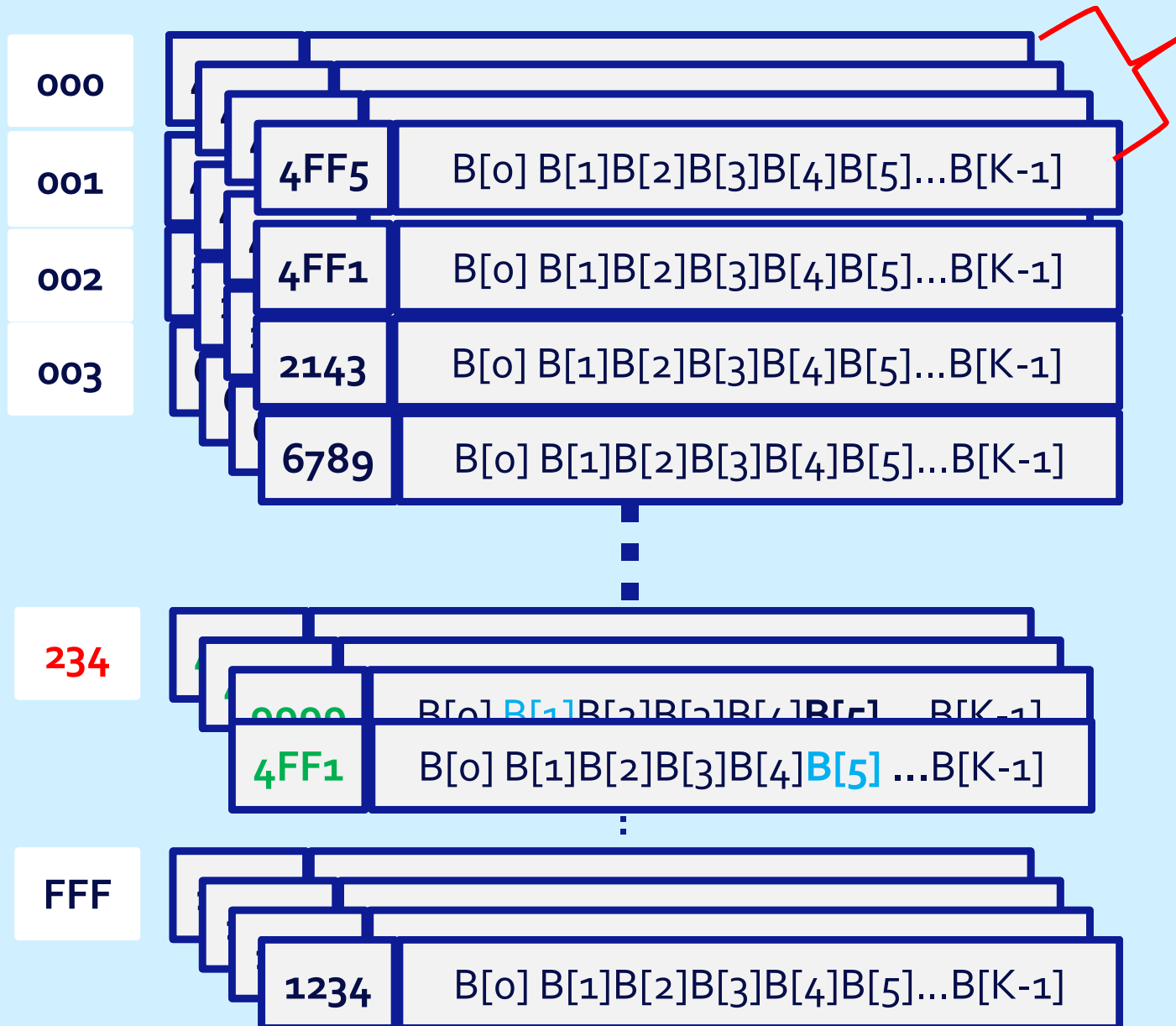
(b) Main memory

Figure 1.17 Cache/Main-Memory Structure

Cache organization



Cache organization

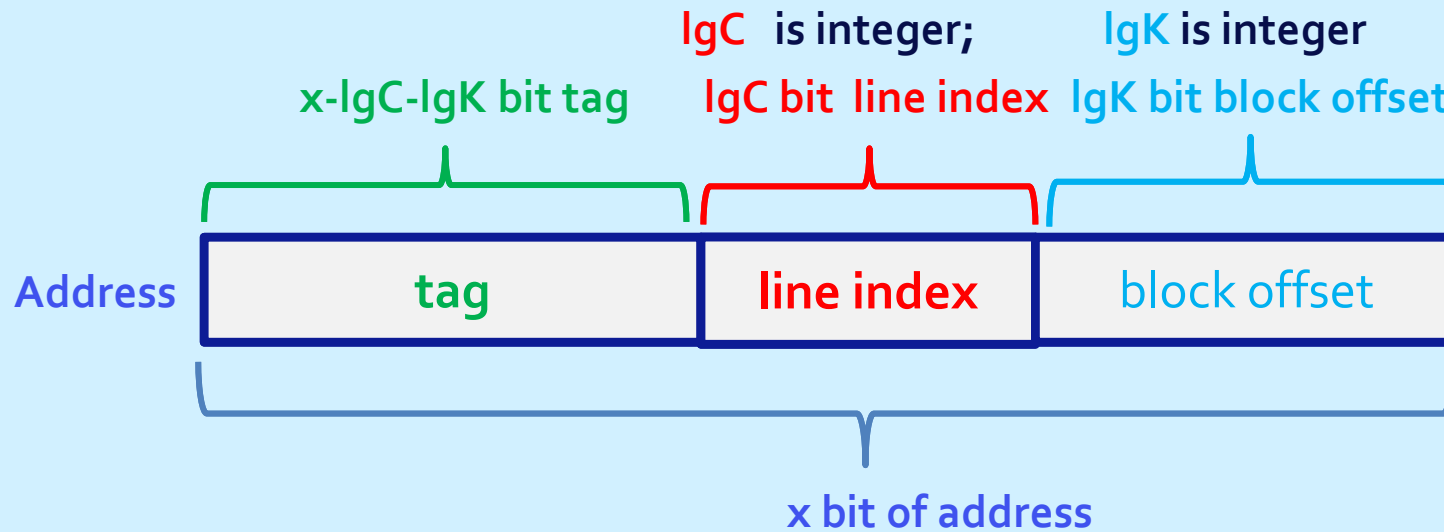


A=4 in this
 example. i.e.
 4-way cache

Three-dimensional entity in the general case: $C \times K \times A$

	X	#bit of address of architecture (e.g. 32bit)	
rows:	C	(number of lines or #sets)	dimension-1
columns:	K	(block length or block size or 'line size')	dimension-2
depth:	A	(associativity or ways or #blocks/set)	dimension-3

(C, K , must be powers of two; A not necessarily) $\Rightarrow \lg C, \lg K$ are both integer!





Nominal size of cache (#B): $C \times K \times A$

Real size of cache (#B or #bit): $C \times K \times A + C \times A \times \text{ceil}[\{(\text{\#bit for tag}) + (\text{\#bit for flags})\}/8]$

$C \times K \times A \times 8 + C \times A \times \{(\text{\#bit for tag}) + (\text{\#bit for flags})\}$

1. Cache organization is A copies of a matrix of C rows and K 'byte columns'; every row contains K bytes plus some tag information associated with the K bytes plus some flag information. Thus the row is not just K byte long but a bit more ($\text{\#bit for tag} + \text{\#bit for flags}$).
2. Per subject o a byte of memory say of address D belongs to a page of size K (from now on block length) and of a given page number (from now on line index) and page offset (from now on block offset). The rest of D (that is neither a line index nor block offset) is the tag of D
3. The line index part of D is used to locate all relevant rows of all A matrices. Each row's tag is compared to the tag of D . If none matches the tag of D , then D is not in the cache. If one matches the tag of D , then we retrieve from that matrix and that row (line index) the column indexed by the block offset!

Review 3.2



Intel Pentium 4 (32-bit) L1D cache: Cache size 8KiB, 4-way associative, 64B block size

3.2.1. What is the x?

x=32 (Intel P4 32-bit)

3.2.2. How much is CxKxA ?

8KiB = $8 \times 1024\text{B} = 8192\text{B}$

3.2.3. How much is A?

4-way $\Rightarrow A=4$

3.2.4. How much is K?

64B (block size/length)

3.2.5. How much is lgK?

$\lg 64 = 6$

3.2.6. How much is C?

$C \times K \times A / (K \times A) =$

$8192 / (64 \times 4) = 32$

3.2.7. How much is lgC?

$\lg 32 = 5$

3.2.8. How many bit for a tag?

$x - \lg C - \lg K = 32 - 5 - 6 = 21$

3.2.9 Let's say each line use 8-bit flags

How many bit per line of the cache?

$K \times 1\text{B} \times 8\text{bit} / B + \text{tag} + \text{flag} =$

$64 \times 8 + 21 + 8 = 541\text{bit}$

3.2.10 How many bit in 8KiB? (nominal bit size)

$8\text{KiB} \times 8\text{bit} / B = 65536\text{bit}$

3.2.11 Real cache size in bit?

$C \times A \times 541\text{bit} = 69248\text{bit}$

Example



(With reference to the architecture of Review 3.2)

3.3.1. Let $D = 00101010 \ 10010000 \ 00000000 \ 11000010$

3.3.2. What the block number (in hex), what is the block offset(in hex)?

3.3.3. What is the tag, line index, block offset in binary?

3.3.4. What is the tag and line index in hex?

Answer is on next page...

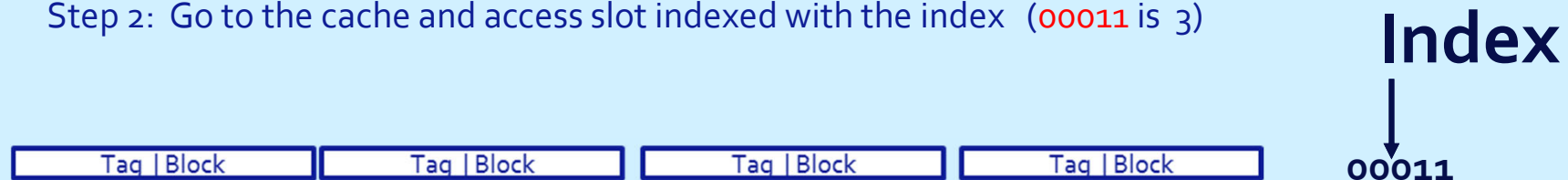
Input : For the previous example if we have a 32-bit address

00101010 10010000 00000000 11000010

Step 1: Isolate **tag(21 bit)** ; **index(5 bit)** ; **blockoffset(6 bit)**

00101010 10010000 00000000 11000010

Step 2: Go to the cache and access slot indexed with the index (00011 is 3)



Step 3: For that slot compare the tag (00101010100100000000) with each one of the four tags available for the slot. If a match (hit) go to Step 4 else a miss is recorded go to Step 5

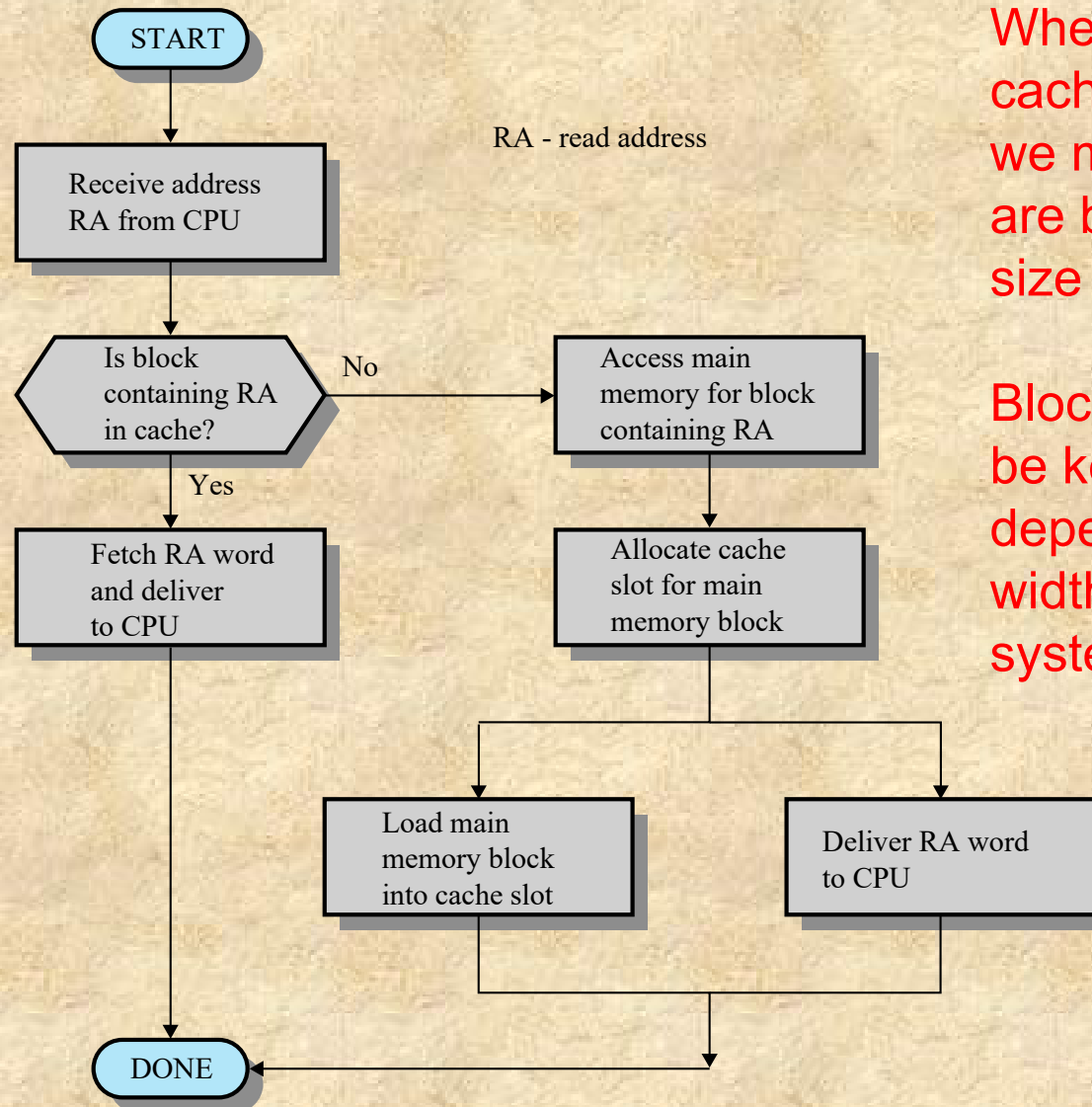
Step 4: (Hit) Retrieve the block and from the block, retrieve the data starting with block offset (000010) [the third byte of the block]

Step 5: (Miss) deal with the miss

Offset

Tag

Note: Depending on the cache design other information might be stored in the cache slot if available (eg flag/valid bits) along the tag.

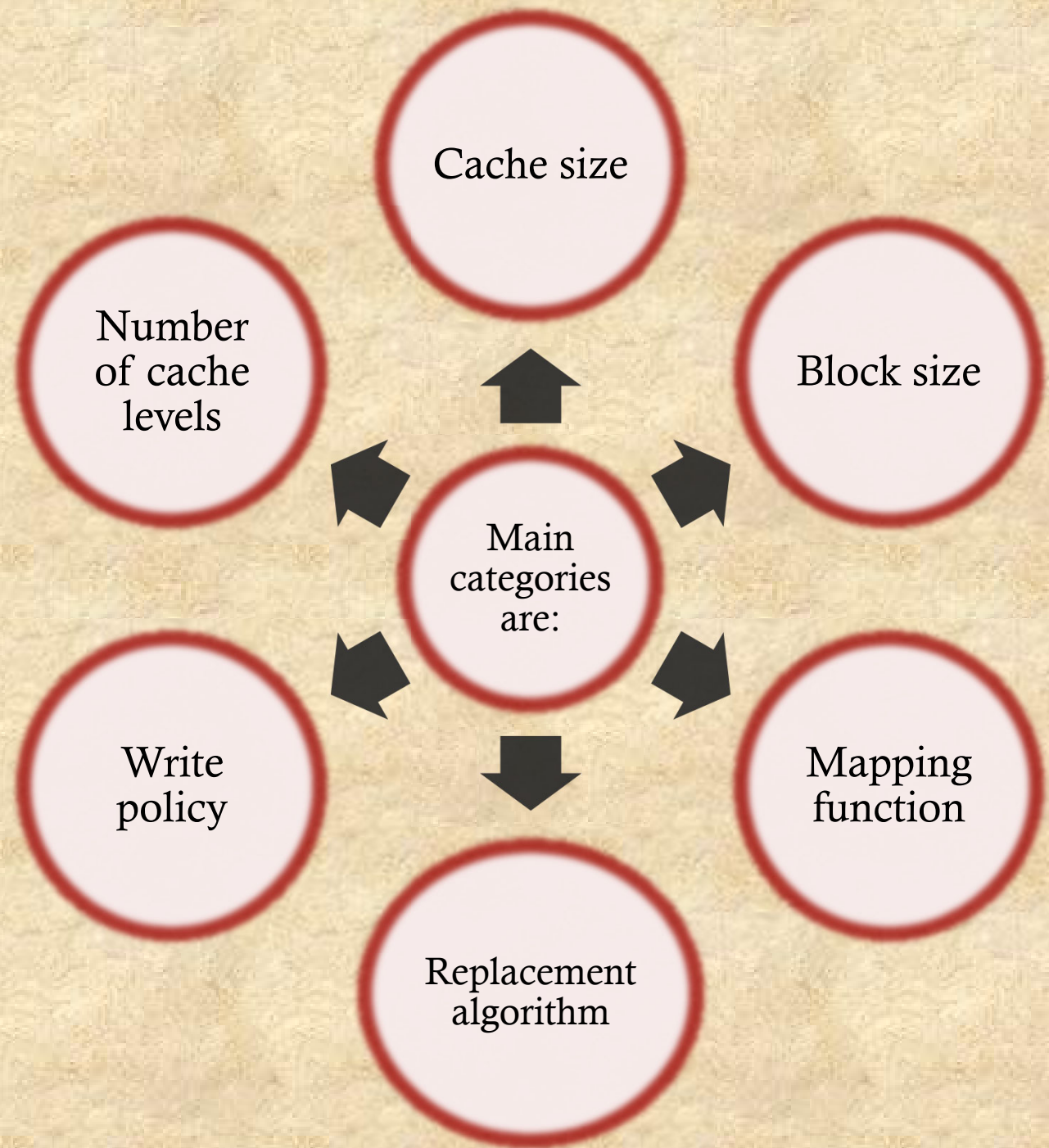


When we load the cache from MM we move data that are blocklength in size

Blocklength should be kept small depending on the width (#bit) of the system bus

Figure 1.18 Cache Read Operation

Cache Design



Cache and Block Size

Cache Size

Small caches have significant impact on performance

Block Size

The unit of data exchanged between cache and main memory

Mapping Function

- Determines which cache location the block will occupy

Two constraints affect design:

When one block is read in, another may have to be replaced

The more flexible the mapping function, the more complex is the circuitry required to search the cache

Replacement Algorithm

- Least Recently Used (LRU) Algorithm
 - Effective strategy is to replace a block that has been in the cache the longest with no references to it
 - Hardware mechanisms are needed to identify the least recently used block
 - Chooses which block to replace when a new block is to be loaded into the cache

Write Policy

Dictates when the memory write operation takes place

- Can occur every time the block is updated
- Can occur when the block is replaced
 - Minimizes write operations
 - Leaves main memory in an obsolete state



Cache Write Policy

- Write-through** : both cache and main memory get updated
- Write-back** : only cache gets updated (dirty bit on to indicate so); main memory not updated... When to update?
- Write-miss** : An address being written into, is not in cache ...
When we update main memory, do we load it into the cache?
- Write-around**
- Write-no-allocate** : Main memory changes, cached does not !

```
sum=0
```

```
for(i = 0 ; i < n ; i++)  
  for(j=0; j < n ; j++)  
    sum += A[i][j];
```

```
sum=0;
```

```
for (j=0 ; j < n ; j++)  
  for(i=0 ; i < n ; i++)  
    sum += A[i][j]
```

Summary

- Basic Elements
- Evolution of the microprocessor
- Instruction execution
- Interrupts
 - Interrupts and the instruction cycle
 - Interrupt processing
 - Multiple interrupts
- The memory hierarchy
- Cache memory
 - Motivation
 - Cache principles
 - Cache design
- Direct memory access
- Multiprocessor and multicore organization
 - Symmetric multiprocessors
 - Multicore computers



Names

High-level (in all names you may attach the word file)

Code:	C/C++/Java source code
Source Code:	Correct way to call "Code"
Assembly (or Assembled output):	Use a compiler to convert source code into assembly
Executable:	Use assembler to convert assembly into executable NOTE: NOWADAY we use a compiler infrastructure to generate an executable from source code (though assembly is generated by we do not see it.)
Source code:	Has a sequence of statements Example: <code>sum=0; for(i=0;i<n;i++) sum+=A[i];</code>
Instructions:	CPU instructions; a statement is split into one or more cpu instructions Example: <code>j++</code> becomes <code>MOVE AC, M[j]</code> (actions right-to-left) <code>INCREMENT AC</code> (AC:accumulator) <code>STORE M[j], AC</code>
SISD:	Execution one CPU instruction at a time on one piece of data Example: <code>ADD AC, M[j]</code>
SIMD:	Execution one CPU instruction at a time on more data Example: <code>ADD AC, M[0:20]</code>
SIMT:	Execution one statement (CPU instruction) by multiple threads Example: <code>_threadKernel () { for(i=0;i<10;i++) sum +=A[threadID+i];</code>



Cache : Paging

Paging: L is MM address

s : page size [power of 2]

$L = \langle \text{Page \#}, \text{Page Offset} \rangle = \langle P, T \rangle$

Caching: L is MM address

K: block length (or size) [power of 2]

C: line length [power of 2]

Wayness of cache: #of copies of each line

$L = \langle \text{block \#}, \text{block offset} \rangle = \langle \text{block\#}, T \rangle$

$L = \langle \text{Tag}, \text{Line\#}, \text{blockoffset} \rangle$

LineIndex=Line# = Line number = Line offset (aliases)

Calculations using integer division (quotient+remainder)

Page # = L / s

Page Offset = $L \% s$

Block # = L / K

Block Offset = $L \% K$

Line # = $\text{Block\#} \% C$

Tag = $\text{Block\#} / C$



Cache : Paging

Paging: L is MM address

s : page size [power of 2]

$L = \langle \text{Page \#}, \text{Page Offset} \rangle = \langle P, T \rangle$

Caching: L is MM address

K: block length (or size) [power of 2]

C: line length [power of 2]

Wayness of cache: #of copies of each line

$L = \langle \text{block \#}, \text{block offset} \rangle = \langle \text{block\#}, T \rangle$

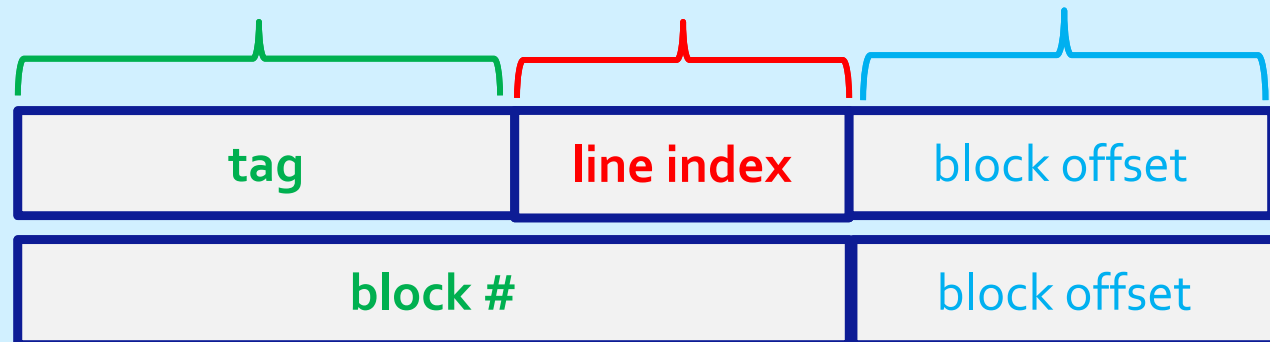
$L = \langle \text{Tag}, \text{LineIndex}, \text{blockoffset} \rangle$

LineIndex=Line# = Line number = Line offset (aliases)

Calculations using bit manipulation

$x - \lg C - \lg K$ bit tag $\lg C$ bit line index $\lg K$ bit block offset

L:



NJIT

A white curved line that starts under the 'N', arches over the 'J' and 'I', and ends under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department