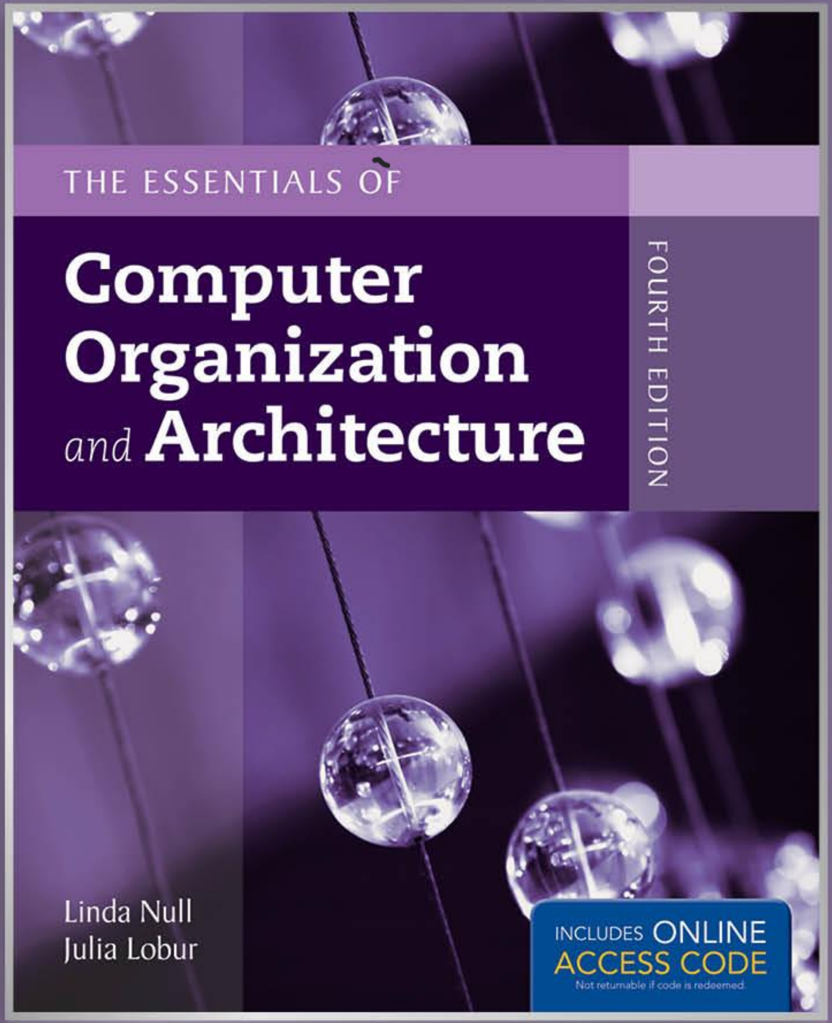
Chapter 4

MARIE: An Introduction to a Simple Computer



4.1 Introduction

- Chapter 1 presented a general overview of computer systems.
- In Chapter 2, we discussed how data is stored and manipulated by various computer system components.
- Chapter 3 described the fundamental components of digital circuits.
- Having this background, we can now understand how computer components work, and how they fit together to create useful computer systems.

4.2 CPU Basics

- The computer's CPU fetches, decodes, and executes program instructions.
- The two principal parts of the CPU are the *datapath* and the *control unit*.
  - The datapath consists of an arithmetic-logic unit and storage units (registers) that are interconnected by a data bus that is also connected to main memory.
  - Various CPU components perform sequenced operations according to signals provided by its control unit.

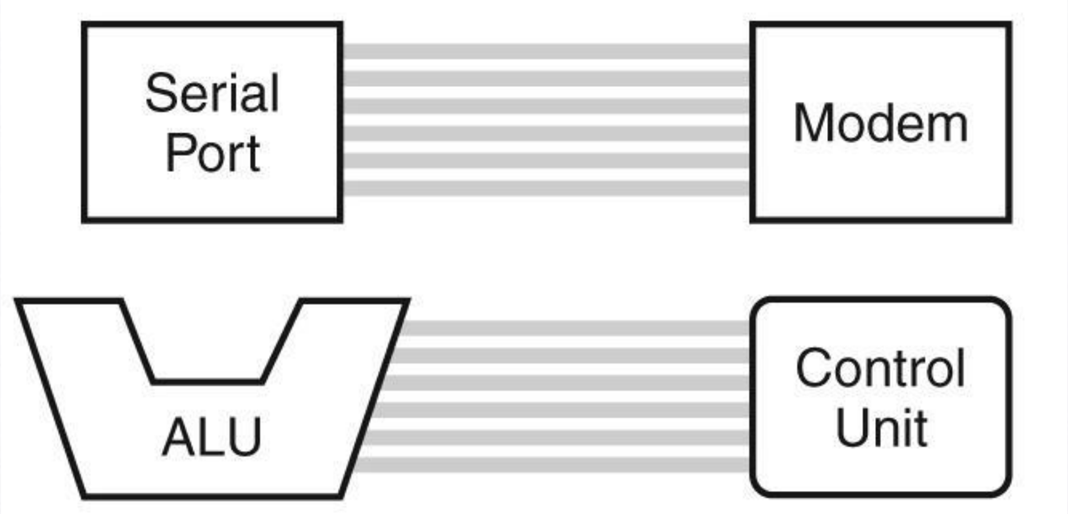
4.2 CPU Basics

- Registers hold data that can be readily accessed by the CPU.
- They can be implemented using D flip-flops. Static
  - A 32-bit register requires 32 D flip-flops. Same for ram
- The arithmetic-logic unit (ALU) carries out logical and arithmetic operations as directed by the control unit.
- The control unit determines which actions to carry out according to the values in a program counter register and a status register.

4.3 The Bus

- The CPU shares data with other system components by way of a data bus. USB
  - A bus is a set of wires that simultaneously convey a single bit along each line.
- Two types of buses are commonly found in computer systems: *point-to-point*, and *multipoint* buses.

These are point-to-point buses:



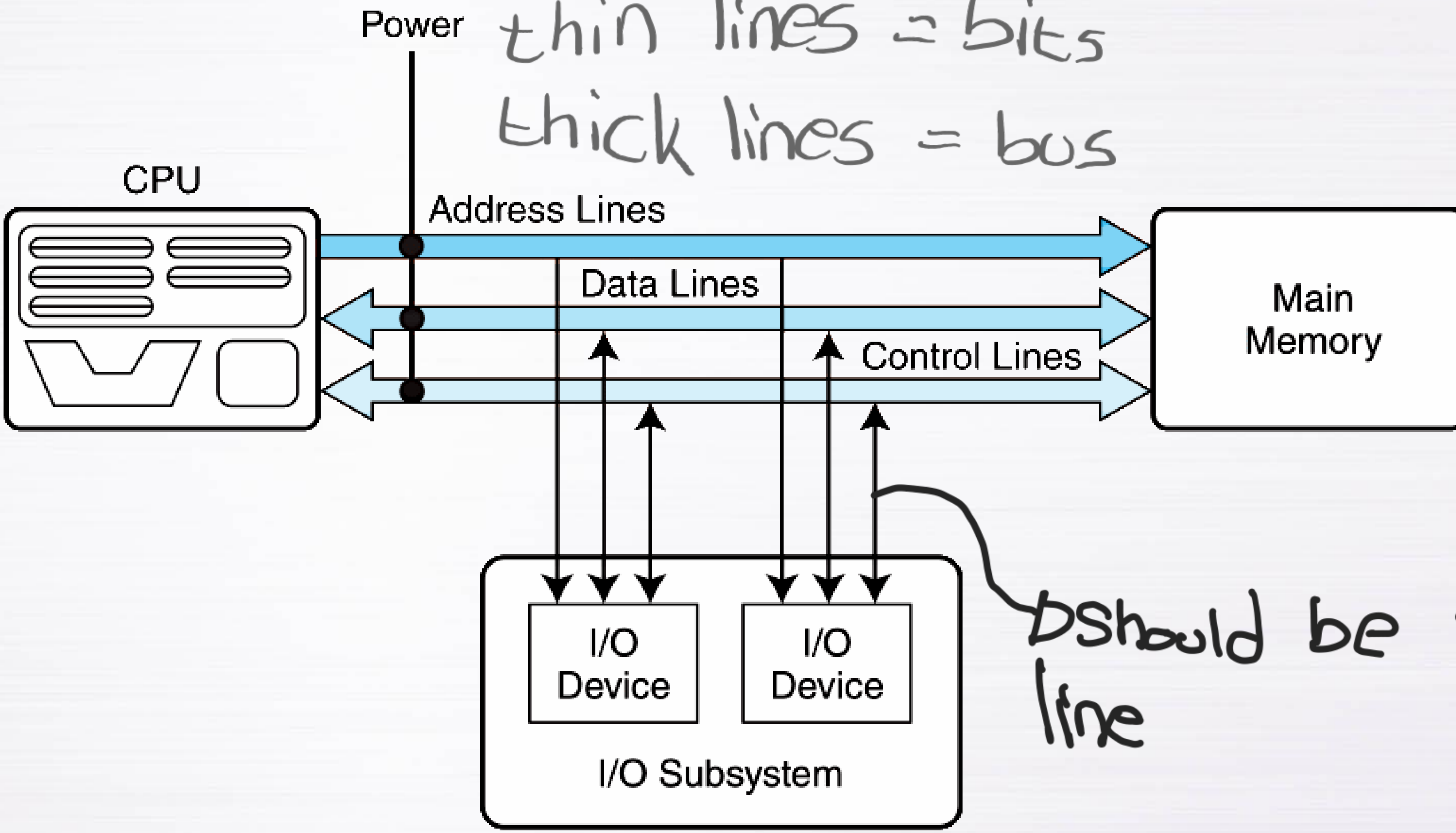
4.3 The Bus

- Buses consist of data lines, control lines, and address lines.
- While the data lines convey bits from one device to another, control lines determine the direction of data flow, and when each device can access the bus.
- Address lines determine the location of the source or destination of the data.

The next slide shows a model bus configuration.

4.3 The Bus

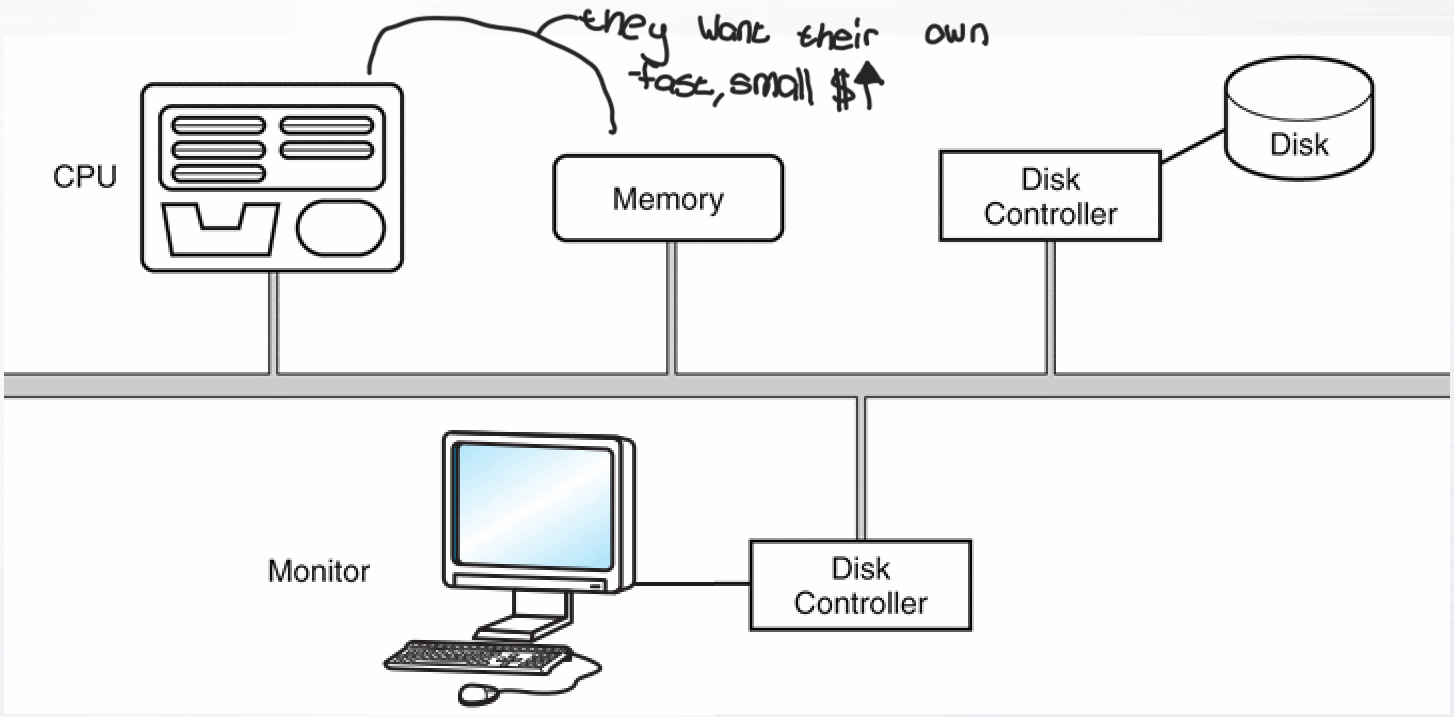
thin lines = bits
thick lines = bus



should be thic line

4.3 The Bus

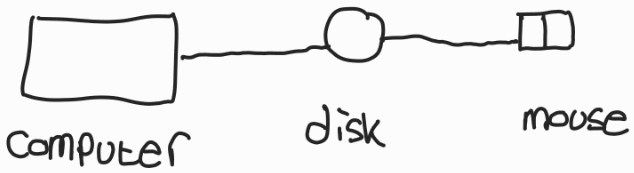
- A multipoint bus is shown below.
- Because a multipoint bus is a shared resource, access to it is controlled through protocols, which are built into the hardware.

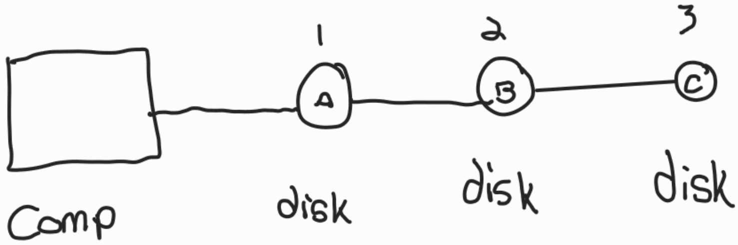


4.3 The Bus

- In a master-slave configuration, where more than one device can be the bus master, concurrent bus master requests must be arbitrated.
- Four categories of bus arbitration are:
  - **Daisy chain**: Permissions are passed from the highest-priority device to the lowest.
  - **Centralized parallel**: Each device is directly connected to an arbitration circuit.
  - **Distributed using self-detection**: Devices decide which gets the bus among themselves.
  - **Distributed using collision-detection:** Any device can try to use the bus. If its data collides with the data of another device, it tries again.

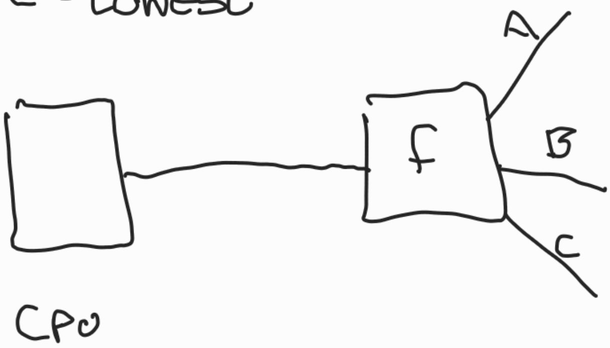
Contention Controll Protocoll





SCSI

A = Highest priority
C = Lowest



you can program
CPU to code
Priority

4.4 Clocks

- Every computer contains at least one clock that synchronizes the activities of its components.
- A fixed number of clock cycles are required to carry out each data movement or computational operation.
- The clock frequency, measured in megahertz or gigahertz, determines the speed with which all operations are carried out.
- Clock cycle time is the reciprocal of clock frequency.
  - An 800 MHz clock has a cycle time of 1.25 ns.

4.4 Clocks

- Clock speed should not be confused with CPU performance.
- The CPU time required to run a program is given by the general performance equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

  - We see that we can improve CPU throughput when we reduce the number of instructions in a program, reduce the number of cycles per instruction, or reduce the number of nanoseconds per clock cycle.

We will return to this important equation in later chapters.

RiSC reduce ARM

CiSC complex

4.5 The Input/Output Subsystem

- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces. address bus
- I/O can be memory-mapped-- where the I/O device behaves like main memory from the CPU's point of view.
- Or I/O can be instruction-based, where the CPU has a specialized I/O instruction set.

We study I/O in detail in chapter 7.

4.6 Memory Organization

- Computer memory consists of a linear array of addressable storage cells that are similar to registers.
- Memory can be byte-addressable, or word-addressable, where a word typically consists of two or more bytes.
- Memory is constructed out of RAM chips, often referred to in terms of length × width.
- If the memory word size of the machine is 16 bits, then a 4M × ~~16~~ RAM chip gives us 4 megabytes of 16-bit memory locations.

8

4.6 Memory Organization

- How does the computer access a memory location corresponding to a particular address?
- We observe that 4M can be expressed as $2^2 \times 2^{20} = 2^{22}$ words.
- The memory locations for this memory are numbered 0 through $(2^{22} - 1)$.
- Thus, the memory bus of this system requires at least 22 bit address lines.

4.6 Memory Organization

- Physical memory usually consists of more than one RAM chip.
- Access is more efficient when memory is organized into banks of chips with the addresses interleaved across the chips
- With low-order interleaving, the low order bits of the address specify which memory bank contains the address of interest.
- Accordingly, in high-order interleaving, the high order address bits specify the memory bank.

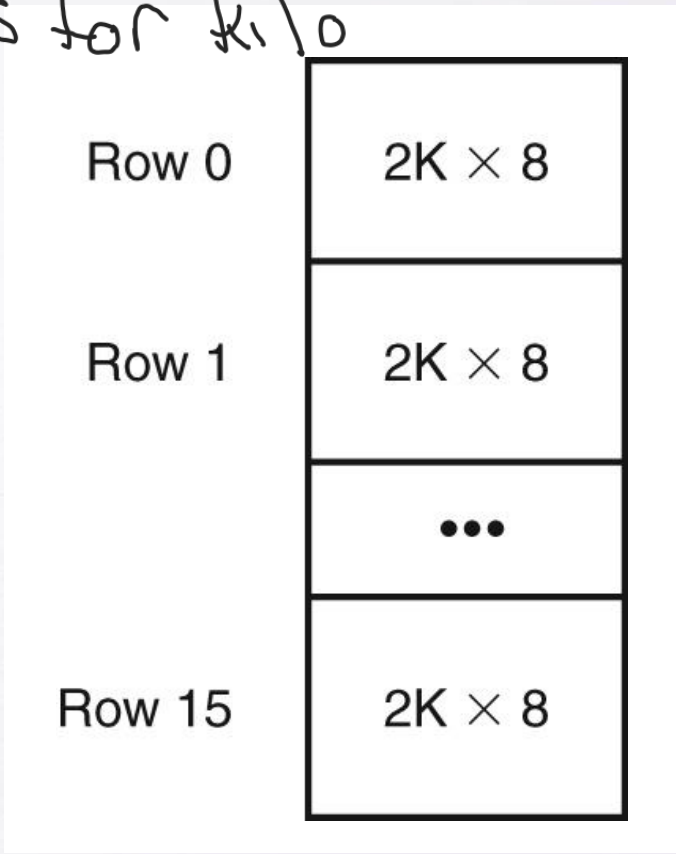
The next two slides illustrate these two ideas.

2k = 11 bits

kilo = 10   2 = 1 = 11

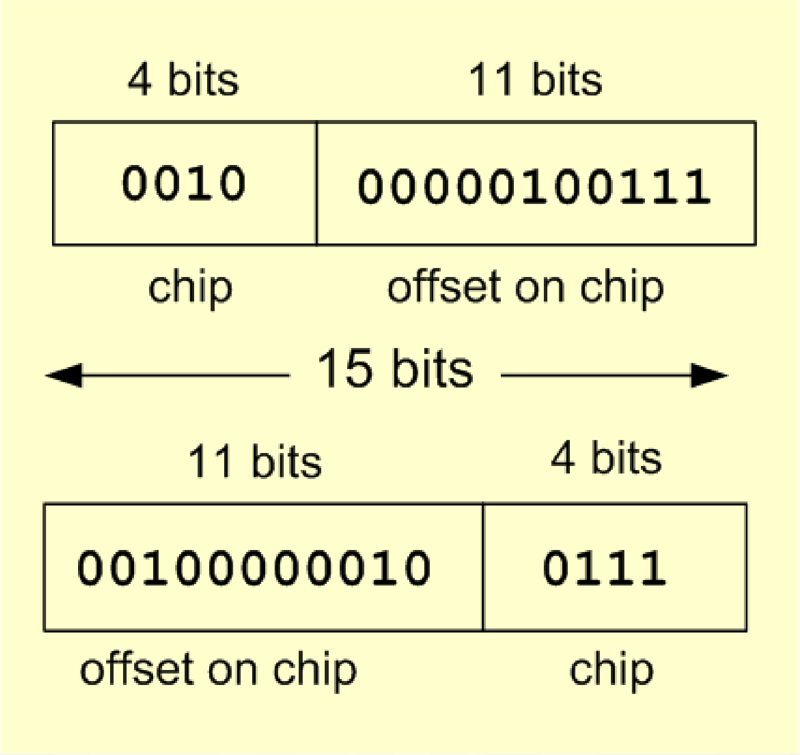
4.6 Memory Organization

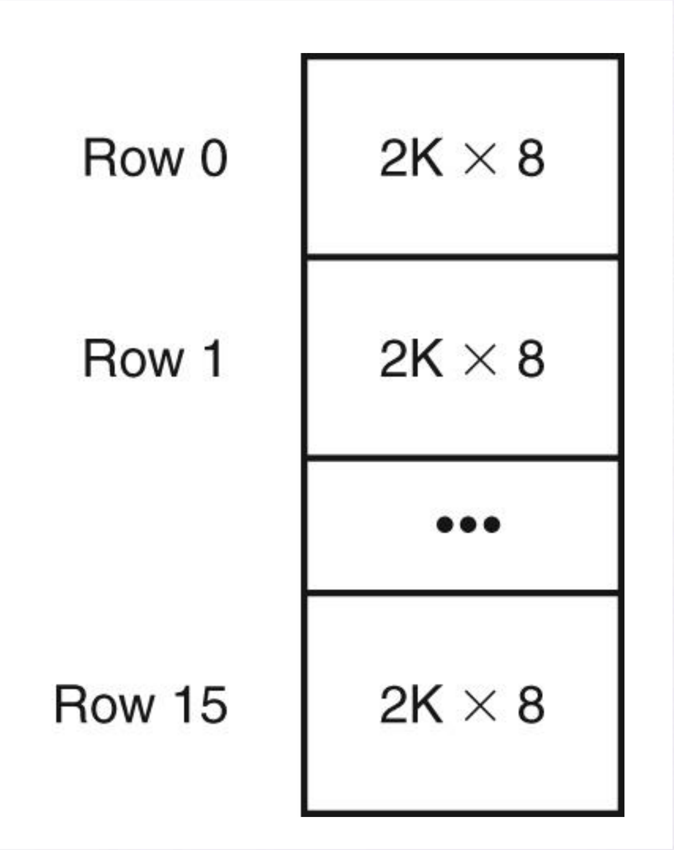
- Example: Suppose we have a memory consisting of 16 2K x 8 bit chips. $2^5 = 32$ is for kilo
  - Memory is $32K = 2^5 \times 2^{10} = 2^{15}$
  - 15 bits are needed for each address.
  - We need 4 bits to select the chip, and 11 bits for the offset into the chip that selects the byte.



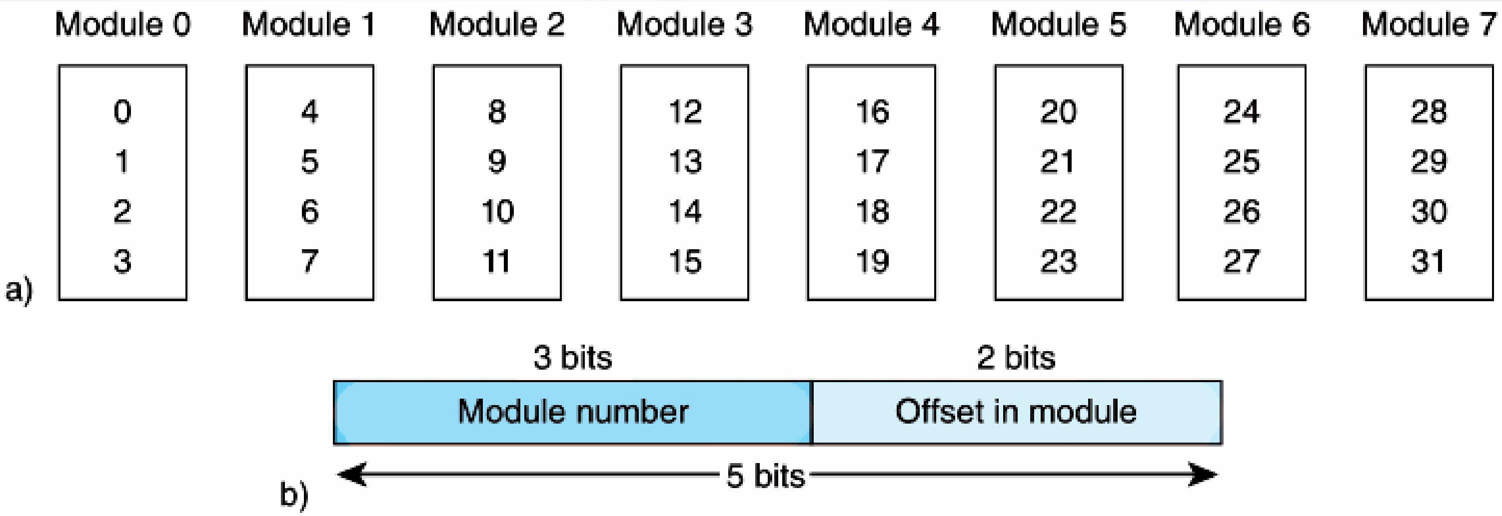
4.6 Memory Organization

- In high-order interleaving the high-order 4 bits select the chip.
- In low-order interleaving the low-order 4 bits select the chip.





4.6 Memory Organization

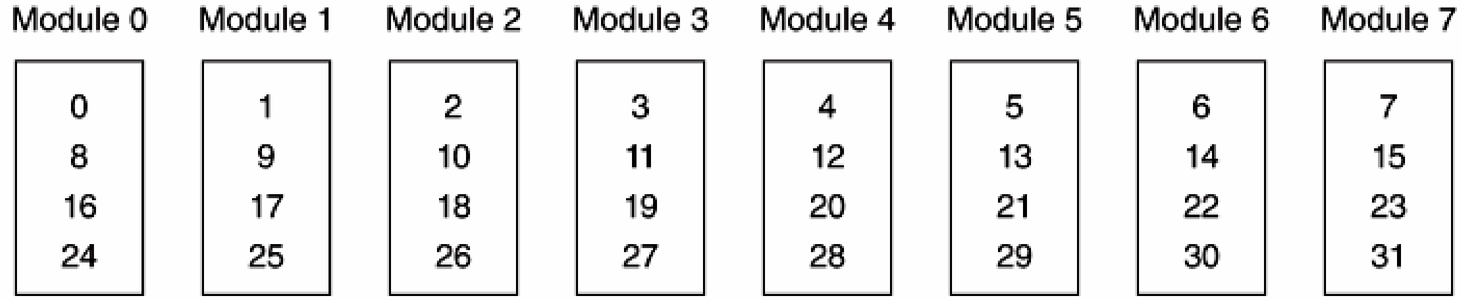


c)

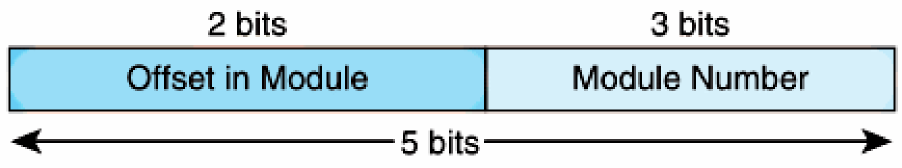
Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Module Number	Offset in Module
Module 0	0	00000	000 00	0	0
	1	00001	000 01	0	1
	2	00010	000 10	0	2
	3	00011	000 11	0	3
Module 1	4	00100	001 00	1	0
	5	00101	001 01	1	1
	6	00110	001 10	1	2
	7	00111	001 11	1	3

a) High-Order Memory Interleaving b) Address Structure c) First Two Modules

4.6 Memory Organization



a)



b)

Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Offset in Module	Module Number
Module 0	0	00000	00 000	0	0
	8	01000	01 000	1	0
	16	10000	10 000	2	0
	24	11000	11 000	3	0
Module 1	1	00001	00 001	0	1
	9	01001	01 001	1	1
	17	10001	10 001	2	1
	25	11001	11 001	3	1

c)

a) Low-Order Memory Interleaving b) Address Structure c) First Two Modules

4.6 Memory Organization

- EXAMPLE 4.1 Suppose we have a 128-word memory that is 8-way low-order interleaved
  - which means it uses 8 memory banks; $8 = 2^3$
- So we use the low-order 3 bits to identify the bank.
- Because we have 128 words, we need 7 bits for each address ($128 = 2^7$)

4.7 Interrupts

- The normal execution of a program is altered when an event of higher-priority occurs. The CPU is alerted to such an event through an interrupt.
- Interrupts can be triggered by I/O requests, arithmetic errors (such as division by zero), or when an invalid instruction is encountered.
- Each interrupt is associated with a procedure that directs the actions of the CPU when an interrupt occurs.
  - Nonmaskable interrupts are high-priority interrupts that cannot be ignored.

4.8 MARIE

- We can now bring together many of the ideas that we have discussed to this point using a very simple model computer.
- Our model computer, the Machine Architecture that is Really Intuitive and Easy, MARIE, was designed for the singular purpose of illustrating basic computer system concepts.
- While this system is too simple to do anything useful in the real world, a deep understanding of its functions will enable you to comprehend system architectures that are much more complex.

4.8 MARIE

The MARIE architecture has the following characteristics:

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 4K words of word-addressable main memory.
- 16-bit data words.
- 16-bit instructions, 4 for the opcode and 12 for the address.
- A 16-bit arithmetic logic unit (ALU).
- Seven registers for control and data movement.

4.8 MARIE

MARIE's seven registers are:

- Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
- Memory address register, MAR, a 12-bit register that holds the memory address of an instruction or the operand of an instruction.
- Memory buffer register, MBR, a 16-bit register that holds the data after its retrieval from, or before its placement in memory.

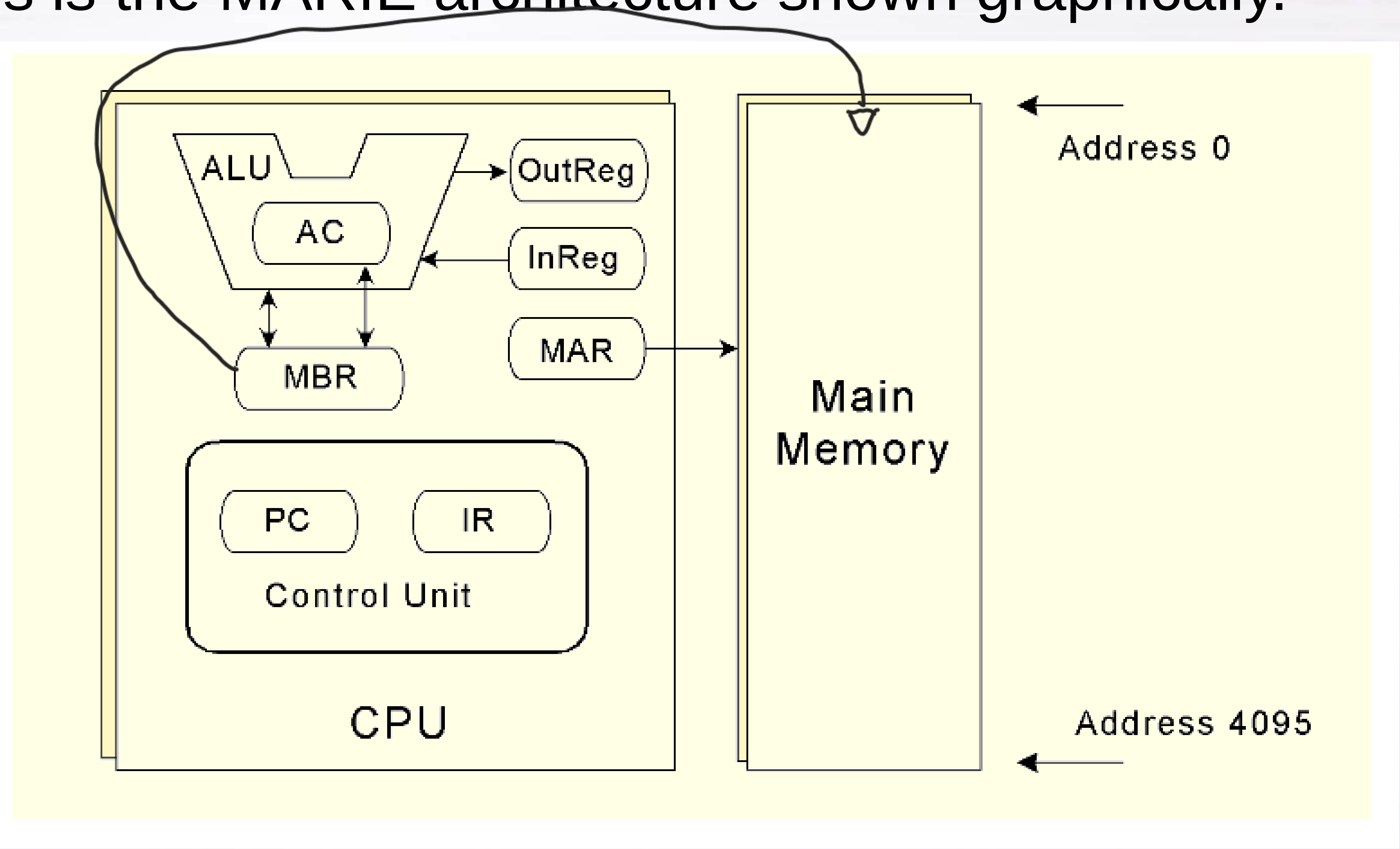
4.8 MARIE

MARIE’s seven registers are:

- Program counter, PC, a 12-bit register that holds the address of the next program instruction to be executed.
- Instruction register, IR, which holds an instruction immediately preceding its execution.
- Input register, InREG, an 8-bit register that holds data read from an input device.
- Output register, OutREG, an 8-bit register, that holds data that is ready for the output device.

4.8 MARIE

This is the MARIE architecture shown graphically.

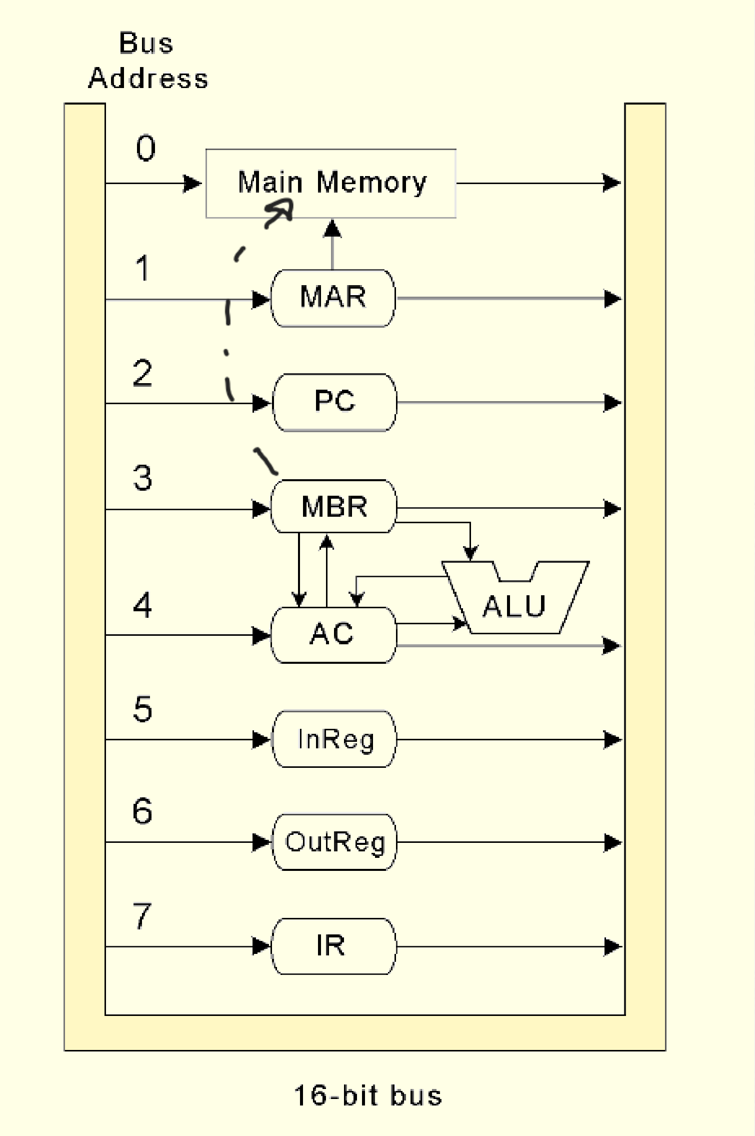


4.8 MARIE

- The registers are interconnected, and connected with main memory through a common data bus.
- Each device on the bus is identified by a unique number that is set on the control lines whenever that device is required to carry out an operation.
- Separate connections are also provided between the accumulator and the memory buffer register, and the ALU and the accumulator and memory buffer register.
- This permits data transfer between these devices without use of the main data bus.

4.8 MARIE

This is the MARIE data path shown graphically.

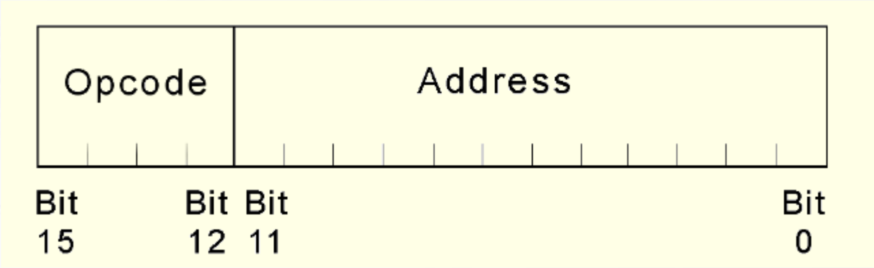


4.8 MARIE

- A computer's instruction set architecture (ISA) specifies the format of its instructions and the primitive operations that the machine can perform.
- The ISA is an interface between a computer's hardware and its software.
- Some ISAs include hundreds of different instructions for processing data and controlling program execution.
- We will be looking at nine instructions.

4.8 MARIE

- This is the format of a MARIE instruction:

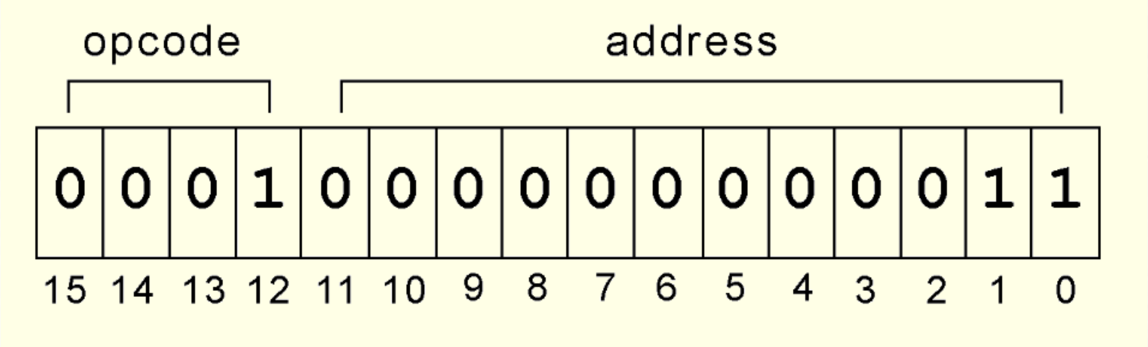


- The fundamental MARIE instructions are:

Instruction Number			
Binary	**Hex**	**Instruction**	**Meaning**
0001	1	Load X	Load contents of address X into AC.
0010	2	Store X	Store the contents of AC at address X.
0011	3	Add X	Add the contents of address X to AC.
0100	4	Subt X	Subtract the contents of address X from AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate program.
1000	8	Skipcond	Skip next instruction on condition.
1001	9	Jump X	Load the value of X into PC.

4.8 MARIE

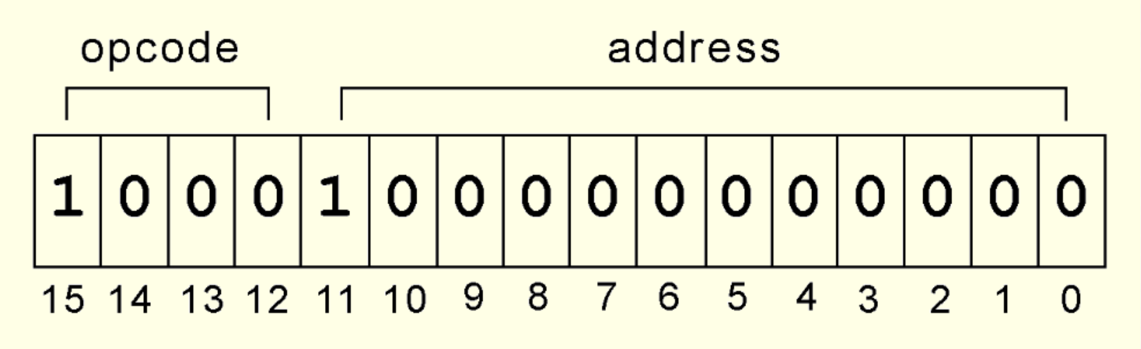
- This is a bit pattern for a **`LOAD`** instruction as it would appear in the IR:



- We see that the opcode is 1 and the address from which to load the data is 3.

4.8 MARIE

- This is a bit pattern for a **`SKIPCOND`** instruction as it would appear in the IR:



- We see that the opcode is 8 and bits 11 and 10 are 10, meaning that the next instruction will be skipped if the value in the AC is greater than zero.

4.8 MARIE

- Each of our instructions actually consists of a sequence of smaller instructions called *microoperations*.
- The exact sequence of microoperations that are carried out by an instruction can be specified using *register transfer language (RTL).*
- In the MARIE RTL, we use the notation M[X] to indicate the actual data value stored in memory location X, and $\leftarrow$ to indicate the transfer of bytes to a register or memory location.

4.8 MARIE

- The RTL for the **`LOAD`** instruction is:

  $\mathbf{MAR} \leftarrow \mathbf{X}$

  $\mathbf{MBR} \leftarrow \mathbf{M[MAR]}$

  $\mathbf{AC} \leftarrow \mathbf{MBR}$

- Similarly, the RTL for the **`ADD`** instruction is:

  $\mathbf{MAR} \leftarrow \mathbf{X}$

  $\mathbf{MBR} \leftarrow \mathbf{M[MAR]}$

  $\mathbf{AC} \leftarrow \mathbf{AC + MBR}$

4.8 MARIE

- Recall that **`SKIPCOND`** skips the next instruction according to the value of the AC.
- The RTL for this instruction is the most complex in our instruction set:

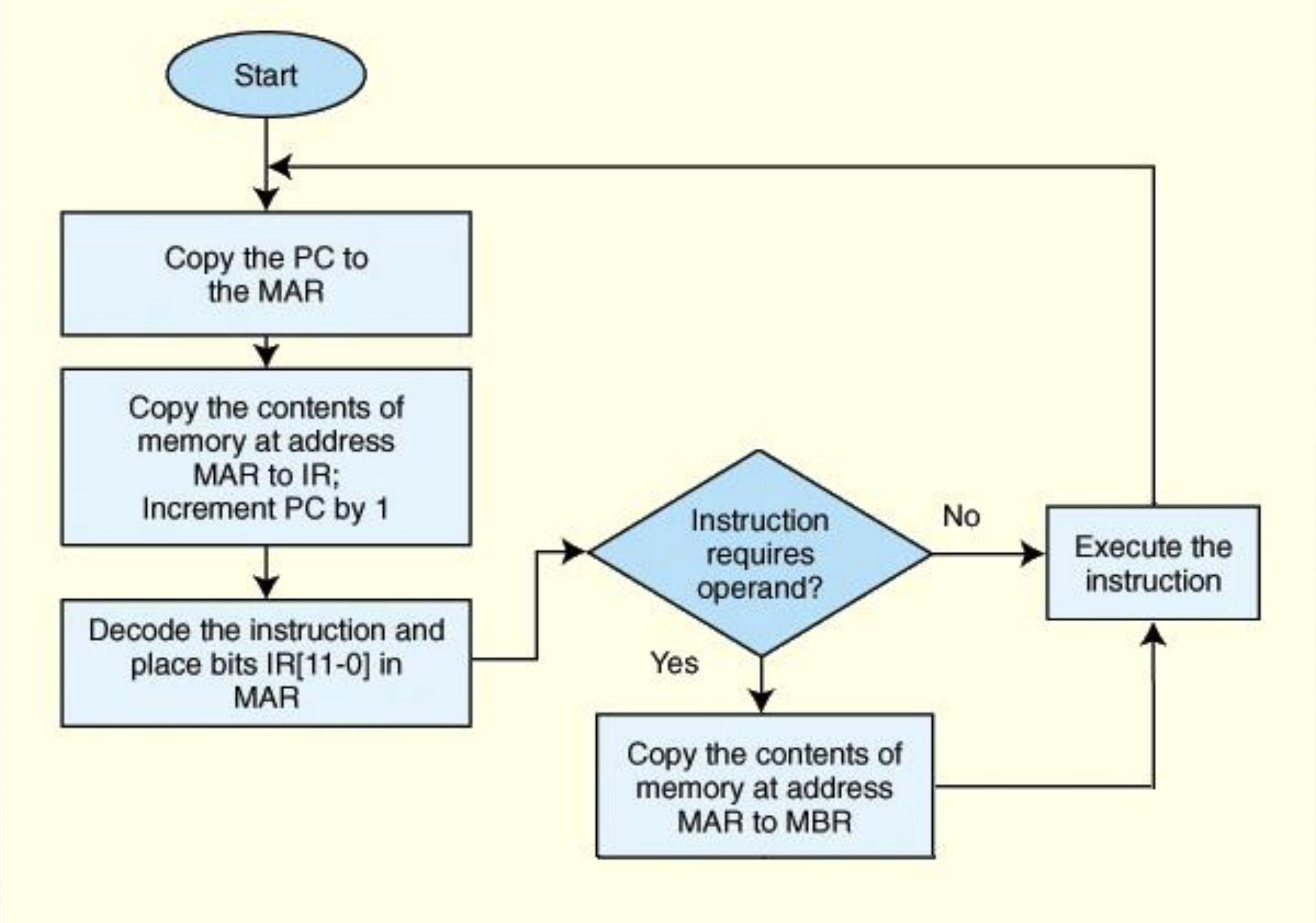
```
If IR[11 - 10] = 00 then
        If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
        If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 11 then
        If AC > 0 then PC ← PC + 1
```

4.9 Instruction Processing

- The *fetch-decode-execute cycle* is the series of steps that a computer carries out when it runs a program.
- We first have to *fetch* an instruction from memory, and place it into the IR.
- Once in the IR, it is *decoded* to determine what needs to be done next.
- If a memory value (operand) is involved in the operation, it is retrieved and placed into the MBR.
- With everything in place, the instruction is *executed*.

The next slide shows a flowchart of this process.

4.9 Instruction Processing

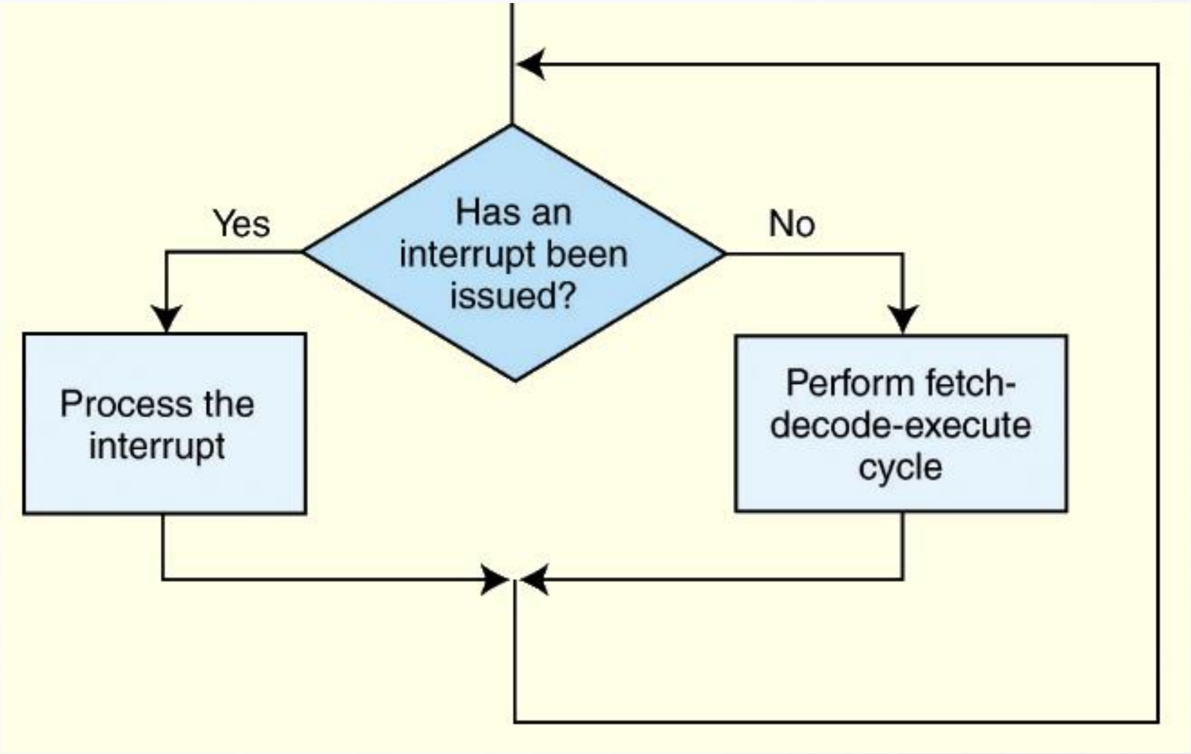


4.9 Instruction Processing

- All computers provide a way of interrupting the fetch-decode-execute cycle.
- Interrupts occur when:
  - A user break (e.,g., Control+C) is issued
  - I/O is requested by the user or a program
  - A critical error occurs
- Interrupts can be caused by hardware or software.
  - Software interrupts are also called *traps*.

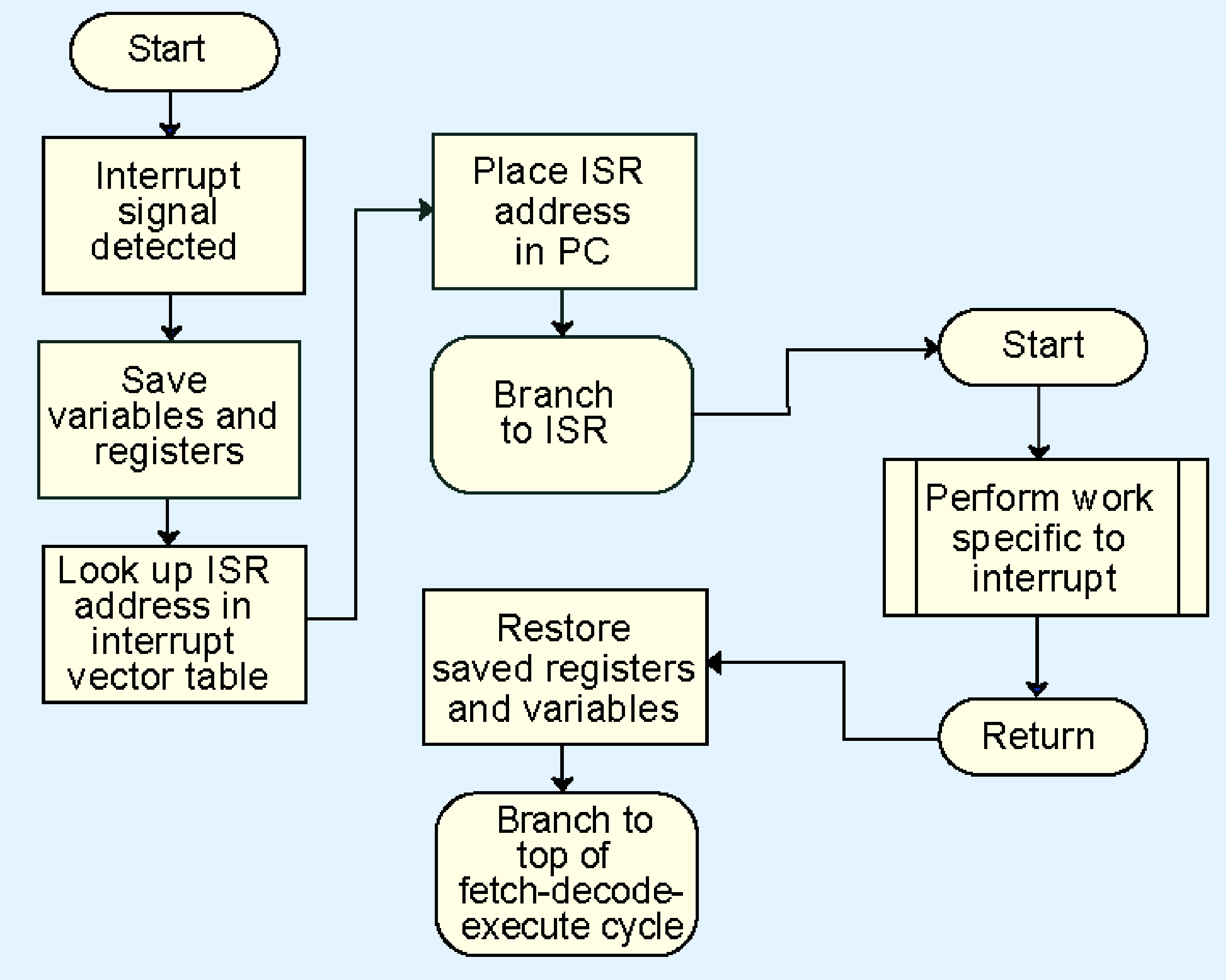
4.9 Instruction Processing

- Interrupt processing involves adding another step to the fetch-decode-execute cycle as shown below.



The next slide shows a flowchart of "Process the interrupt."

4.9 Instruction Processing



4.9 Instruction Processing

- For general-purpose systems, it is common to disable all interrupts during the time in which an interrupt is being processed.
  - Typically, this is achieved by setting a bit in the flags register.
- Interrupts that are ignored in this case are called *maskable*.
- *Nonmaskable* interrupts are those interrupts that must be processed in order to keep the system in a stable condition.

4.9 Instruction Processing

- Interrupts are very useful in processing I/O.
- However, interrupt-driven I/O is complicated, and is beyond the scope of our present discussion.
  - We will look into this idea in greater detail in Chapter 7.
- MARIE, being the simplest of simple systems, uses a modified form of programmed I/O.
- All output is placed in an output register, OutREG, and the CPU polls the input register, InREG, until input is sensed, at which time the value is copied into the accumulator.

4.10 A Simple Program

- Consider the simple MARIE program given below. We show a set of mnemonic instructions stored at addresses 0x100 – 0x106 (hex):

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

4.10 A Simple Program

- Let's look at what happens inside the computer when our program runs.
- This is the **`LOAD 104`** instruction:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ⟵ PC	100	------	100	------	------
	IR ⟵ M[MAR]	100	1104	100	------	------
	PC ⟵ PC + 1	101	1104	100	------	------
Decode	MAR ⟵ IR[11–0]	101	1104	104	------	------
	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR ⟵ M[MAR]	101	1104	104	0023	------
Execute	AC ⟵ MBR	101	1104	104	0023	0023

4.10 A Simple Program

- Our second instruction is **`ADD 105`**:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ⟵ PC	101	1104	101	0023	0023
	IR ⟵ M[MAR]	101	3105	101	0023	0023
	PC ⟵ PC + 1	102	3105	101	0023	0023
Decode	MAR ⟵ IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ⟵ M[MAR]	102	3105	105	FFE9	0023
Execute	AC ⟵ AC + MBR	102	3105	105	FFE9	000C

4.11 A Discussion on Assemblers

- Mnemonic instructions, such as **`LOAD 104`**, are easy for humans to write and understand.
- They are impossible for computers to understand.
- *Assemblers* translate instructions that are comprehensible to humans into the machine language that is comprehensible to computers
  - We note the distinction between an assembler and a compiler: In assembly language, there is a one-to-one correspondence between a mnemonic instruction and its machine code. With compilers, this is not usually the case.

4.11 A Discussion on Assemblers

- Assemblers create an *object program file* from mnemonic *source code* in two passes.
- During the first pass, the assembler assembles as much of the program as it can, while it builds a *symbol table* that contains memory references for all symbols in the program.
- During the second pass, the instructions are completed using the values from the symbol table.

4.11 A Discussion on Assemblers

- Consider our example program at the right.
  - Note that we have included two directives **HEX** and **DEC** that specify the radix of the constants.
- The first pass, creates a symbol table and the partially-assembled instructions as shown.

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

X	104
Y	105
Z	106

1	X
3	Y
2	Z
7 0 0 0	

4.11 A Discussion on Assemblers

- After the second pass, the assembly is complete.

1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0

X	104
Y	105
Z	106

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

4.12 Extending Our Instruction Set

- So far, all of the MARIE instructions that we have discussed use a *direct addressing mode*.
- This means that the address of the operand is explicitly stated in the instruction.
- It is often useful to employ a *indirect addressing*, where the address of the address of the operand is given in the instruction.
  - If you have ever used pointers in a program, you are already familiar with indirect addressing.

4.12 Extending Our Instruction Set

- We have included three indirect addressing mode instructions in the MARIE instruction set.
- The first two are **`LOADI X`** and **`STOREI X`** where specifies the address of the operand to be loaded or stored.
- In RTL :

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← MBR
```

`LOADI X`

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← AC
M[MAR] ← MBR
```

`STOREI X`

4.12 Extending Our Instruction Set

- The **`ADDI`** instruction is a combination of **`LOADI X`** and **`ADD X`**:
- In RTL:

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← AC + MBR
```

`ADDI X`

4.12 Extending Our Instruction Set

- Another helpful programming tool is the use of subroutines.
- The jump-and-store instruction, **`JNS`**, gives us limited subroutine functionality. The details of the **`JNS`** instruction are given by the following RTL:

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

4.12 Extending Our Instruction Set

- Our first new instruction is the **`CLEAR`** instruction.
- All it does is set the contents of the accumulator to all zeroes.
- This is the RTL for **`CLEAR`**:

$$\mathbf{AC} \leftarrow \mathbf{0}$$

- We put our new instructions to work in the program on the following slide.

4.12 Extending Our Instruction Set

```
100 |          LOAD Addr
101 |          STORE Next
102 |          LOAD Num
103 |          SUBT One
104 |          STORE Ctr
105 |Loop  LOAD Sum
106 |          ADDI Next
107 |          STORE Sum
108 |          LOAD Next
109 |          ADD One
10A |          STORE Next
10B |          LOAD Ctr
10C |          SUBT One
10D |          STORE Ctr
```

```
10E |          SKIPCOND
   000
10F |          JUMP Loop
110 |          HALT
111 |Addr   HEX 117
112 |Next   HEX 0
113 |Num    DEC 5
114 |Sum    DEC 0
115 |Ctr    HEX 0
116 |One    DEC 1
117 |          DEC 10
118 |          DEC 15
119 |          DEC 2
11A |          DEC 25
11B |          DEC 30
```