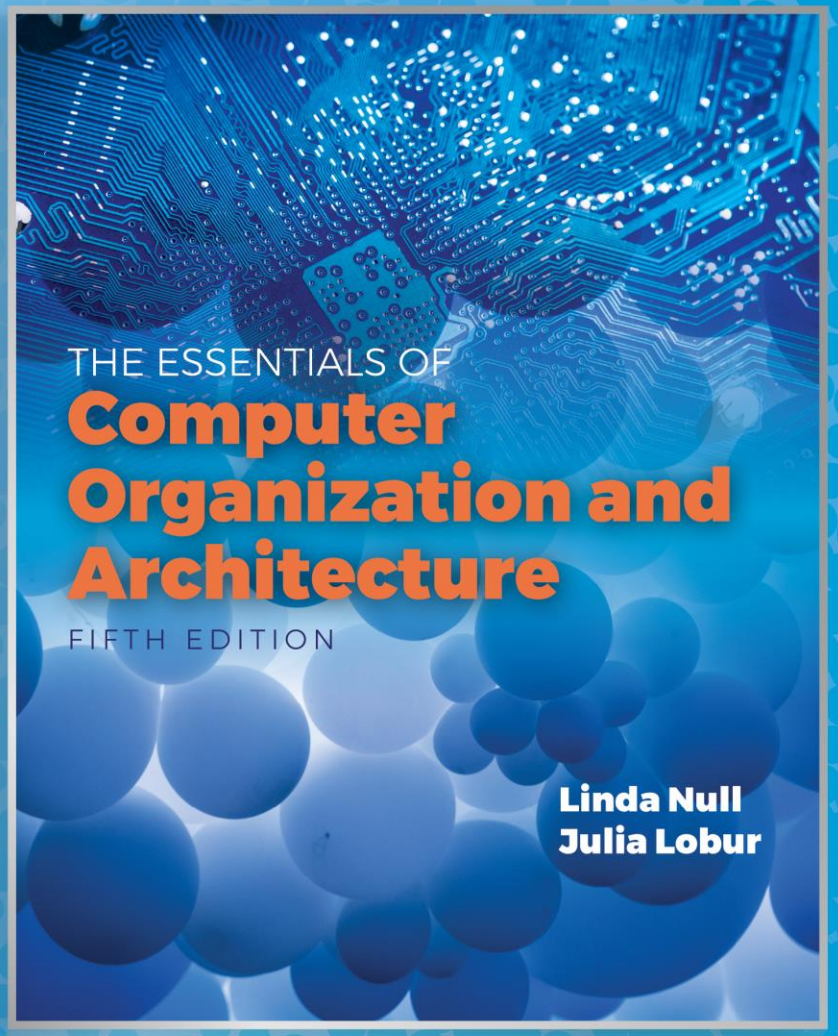
Chapter 4

MARIE: An Introduction to a Simple Computer



Objectives

- Learn the components common to every modern computer system.
- Be able to explain how each component contributes to program execution.
- Understand a simple architecture invented to illuminate these basic concepts, and how it relates to some real architectures.
- Know how the program assembly process works.

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 (1 of 2)

- 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.

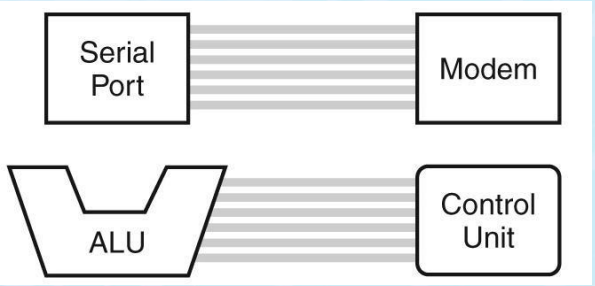
4.2 CPU Basics (2 of 2)

- Registers hold data that can be readily accessed by the CPU.
- They can be implemented using D flip-flops.
  - A 32-bit register requires 32 D flip-flops.
- 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 (1 of 5)

- The CPU shares data with other system components by way of a data bus.
  - 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:

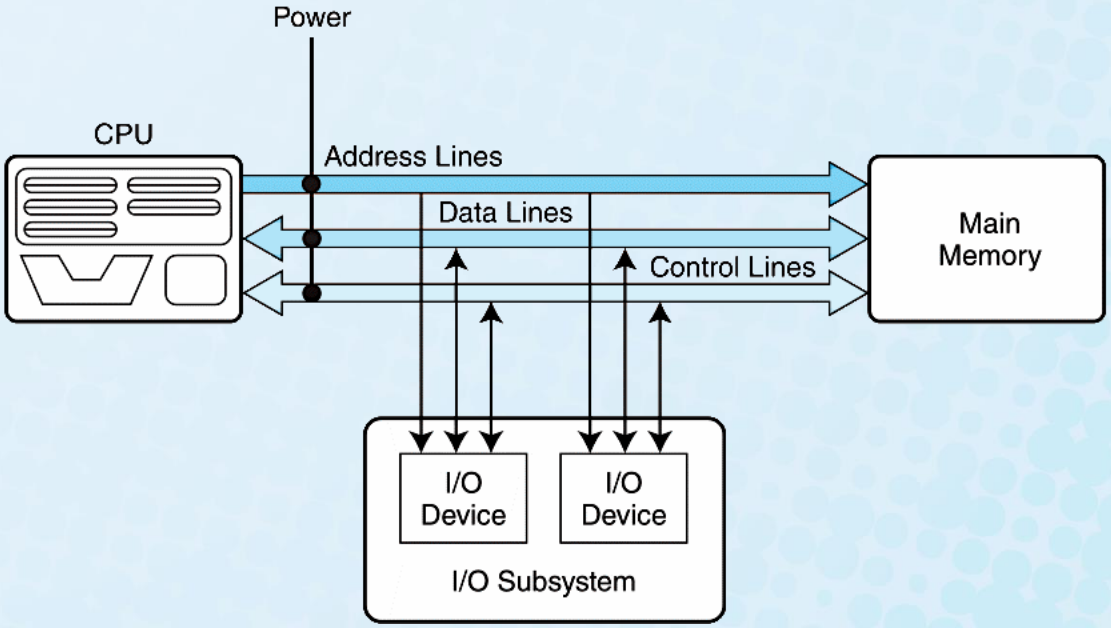


4.3 The Bus (2 of 5)

- 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.

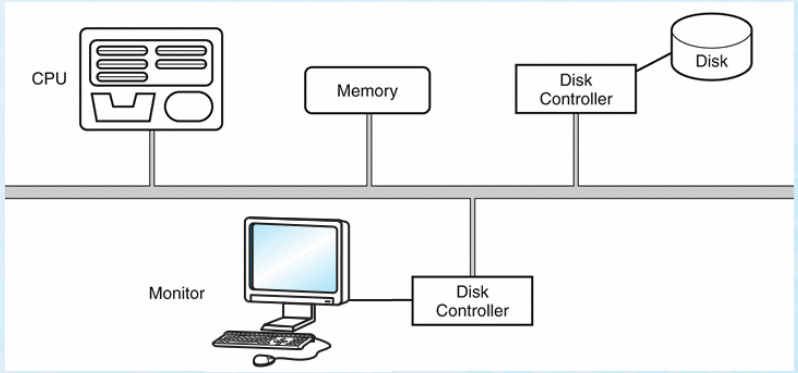
The next slide shows a model bus configuration.

4.3 The Bus (3 of 5)



4.3 The Bus (4 of 5)

- 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 (5 of 5)

- 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.

4.4 Clocks (1 of 2)

- 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 (2 of 2)

- 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.

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.
- 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 (1 of 8)

- 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 of RAM chips, often referred to in terms of length $\times$ width.
- If the memory word size of the machine is 16 bits, then a 4M $\times$ 16 RAM chip gives us 4 mega 16-bit memory locations.

4.6 Memory Organization (2 of 8)

- How does the computer access a memory location corresponds 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 address lines.
  - The address lines "count" from 0 to $2^{22} - 1$ in binary. Each line is either "on" or "off" indicating the location of the desired memory element.

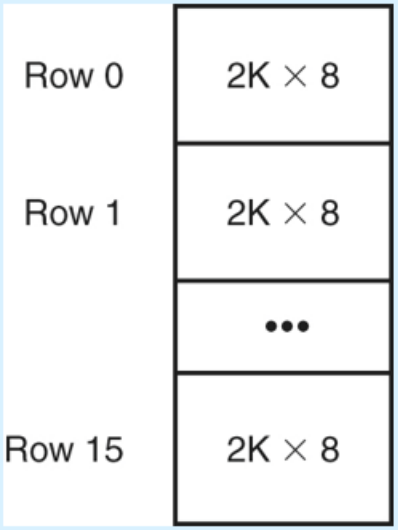
4.6 Memory Organization (3 of 8)

- 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.

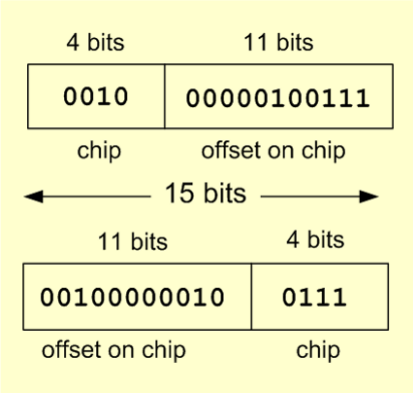
4.6 Memory Organization (4 of 8)

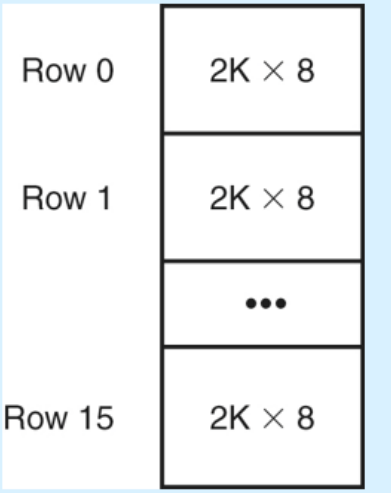
- Example: Suppose we have a memory consisting of 16 2K x 8 bit chips.
  - Memory is 32K = $2^5 \times 2^{10} = 2^{15}$
  - 15 bits are needed for each address.
  - We need 4 bits to select the chip, and 11 bits for the offset into the chip that selects the byte.



4.6 Memory Organization (5 of 8)

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



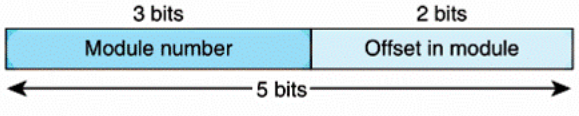


4.6 Memory Organization (6 of 8)

a)

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

b)



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

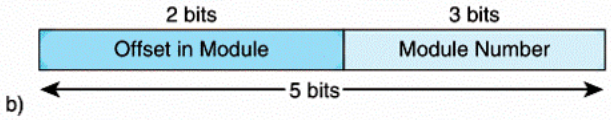
a) High-Order Memory Interleaving b) Address Structure c) First Two Modules

4.6 Memory Organization (7 of 8)

a)

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

b)



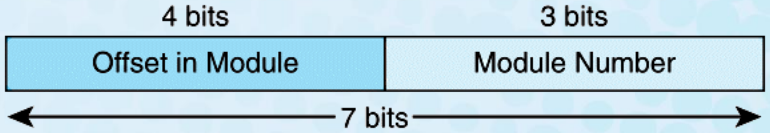
c)

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

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

4.6 Memory Organization (8 of 8)

- 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 (1 of 14)

- 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 (2 of 14)

- 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 (3 of 14)

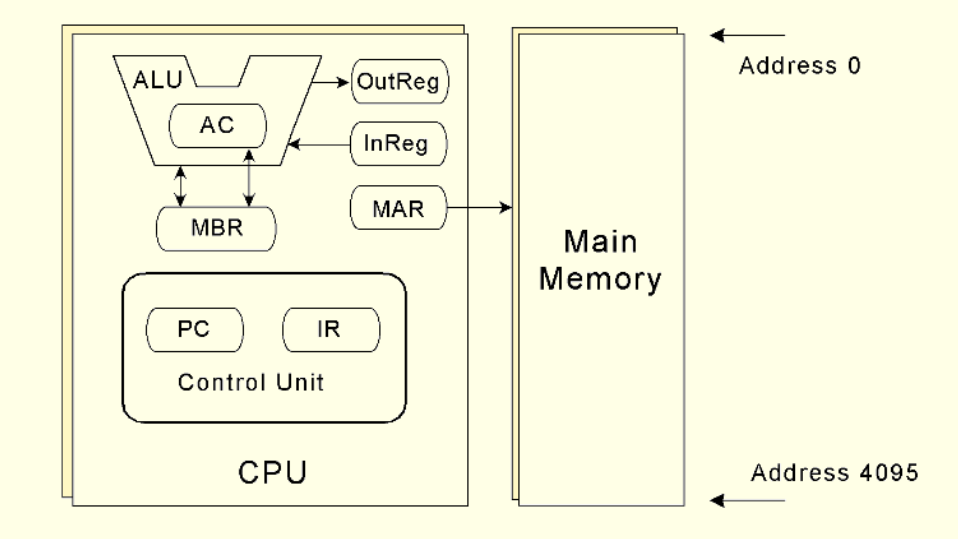
- MARIE's seven registers are:
  - (1) Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
  - (2) Memory address register, MAR, a 12-bit register that holds the memory address of an instruction or the operand of an instruction.
  - (3) 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 (4 of 14)

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

4.8 MARIE (5 of 14)

- This is the MARIE architecture shown graphically.

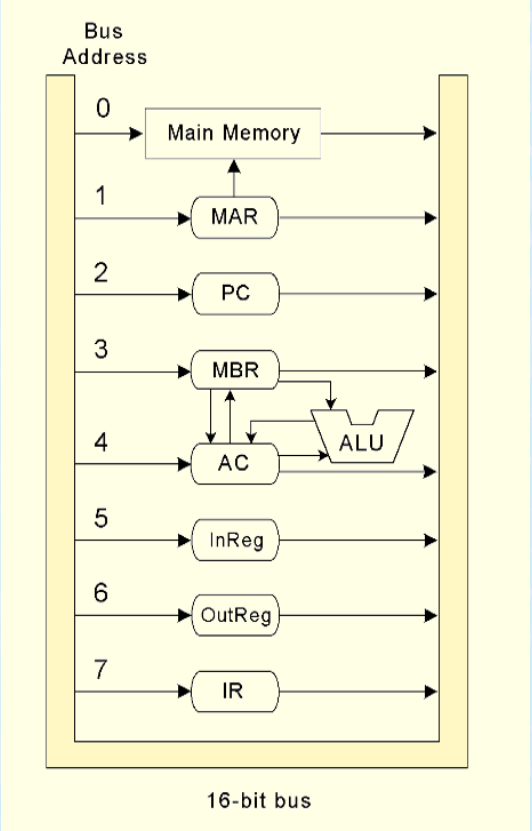


4.8 MARIE (6 of 14)

- 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 (7 of 14)

- This is the MARIE data path shown graphically.

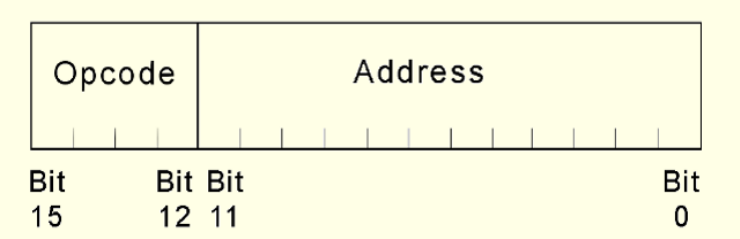


4.8 MARIE (8 of 14)

- 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.
- The MARIE ISA consists of only 13 instructions.

4.8 MARIE (9 of 14)

- This is the format of a MARIE instruction:

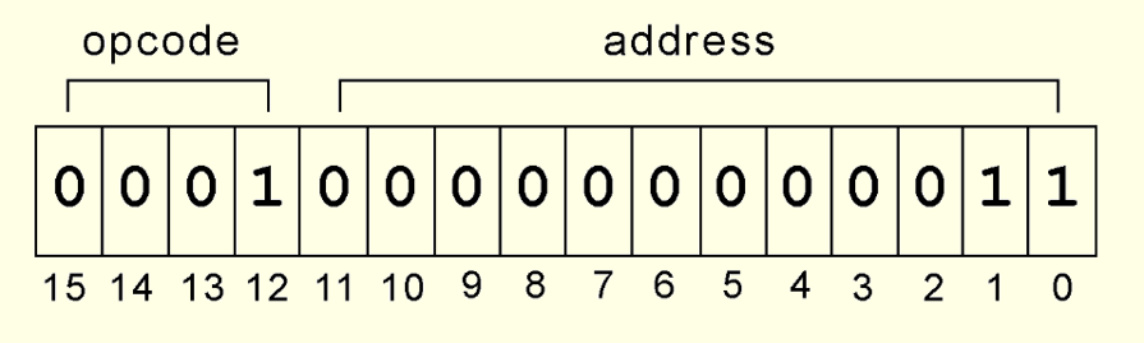


- The fundamental MARIE instructions are:

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 (10 of 14)

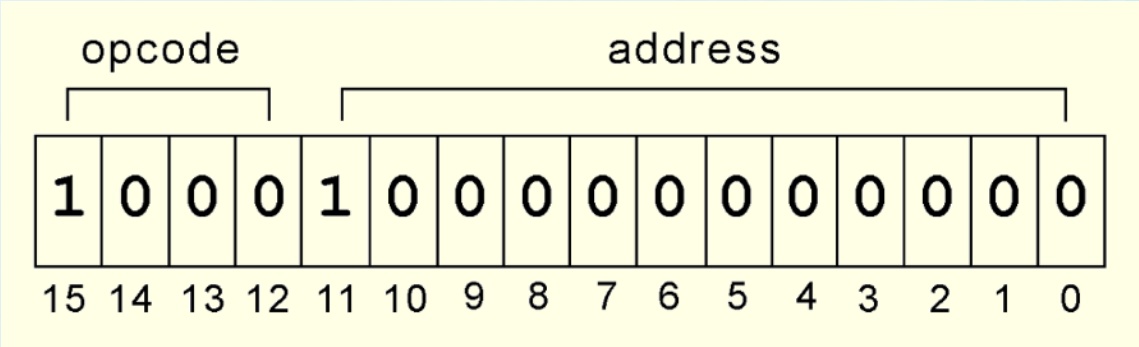
- This is a bit pattern for a **LOAD** instruction as it would appear in the IR:



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

4.8 MARIE (11 of 14)

- 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.

What is the hexadecimal representation of this instruction?

4.8 MARIE (12 of 14)

- 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 ← to indicate the transfer of bytes to a register or memory location.

4.8 MARIE (13 of 14)

- The RTL for the LOAD instruction is:

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

- Similarly, the RTL for the ADD instruction is:

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

4.8 MARIE (14 of 14)

- 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:

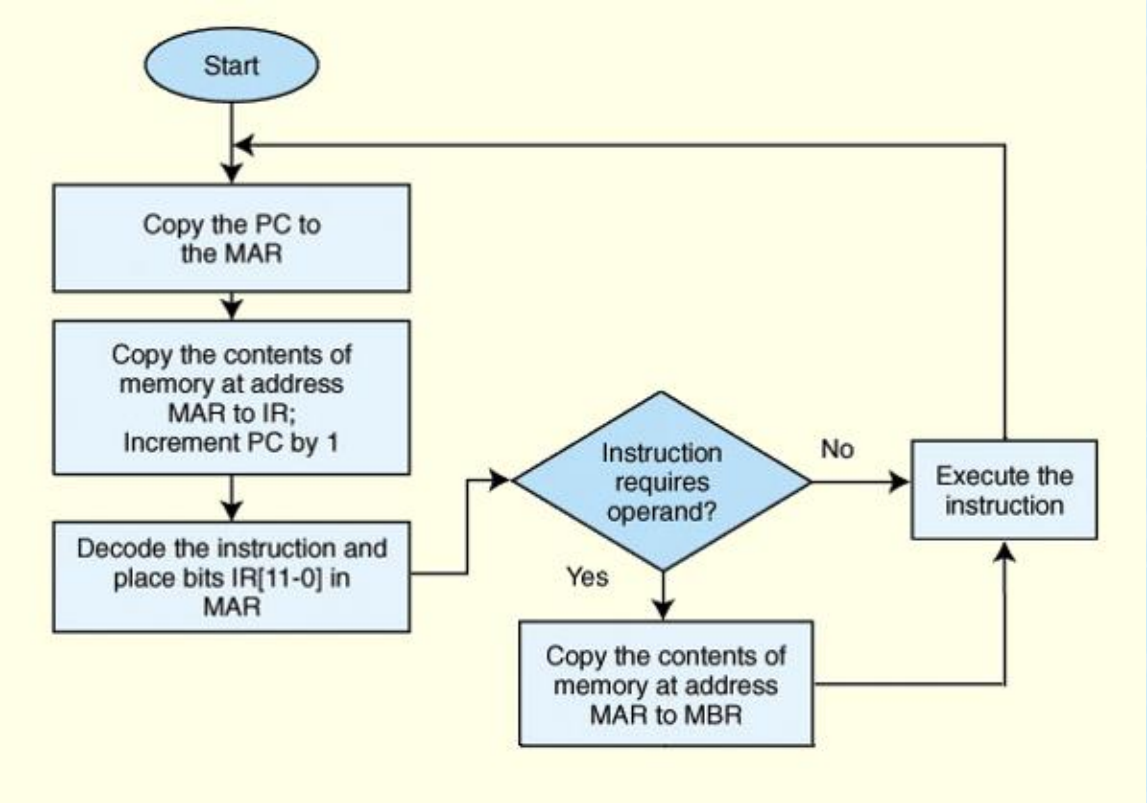
```
If IR[11 - 10] = 00 then
      If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
      If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 10 then
      If AC > 0 then PC ← PC + 1
```

4.9 Instruction Processing (1 of 7)

- 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 (2 of 7)

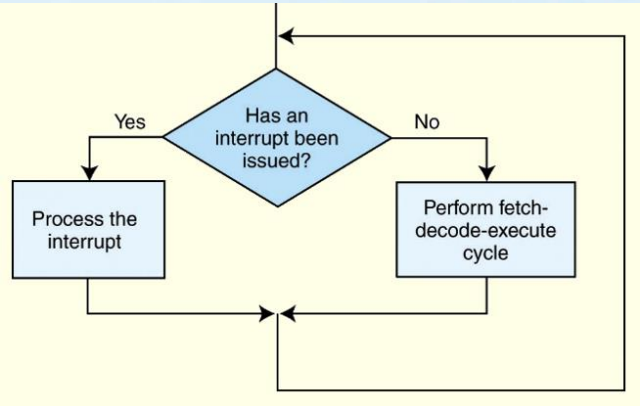


4.9 Instruction Processing (3 of 7)

- All computers provide a way of interrupting the fetch-decode-execute cycle.
- Interrupts are asynchronous and indicate some type of service is required.
- 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*.

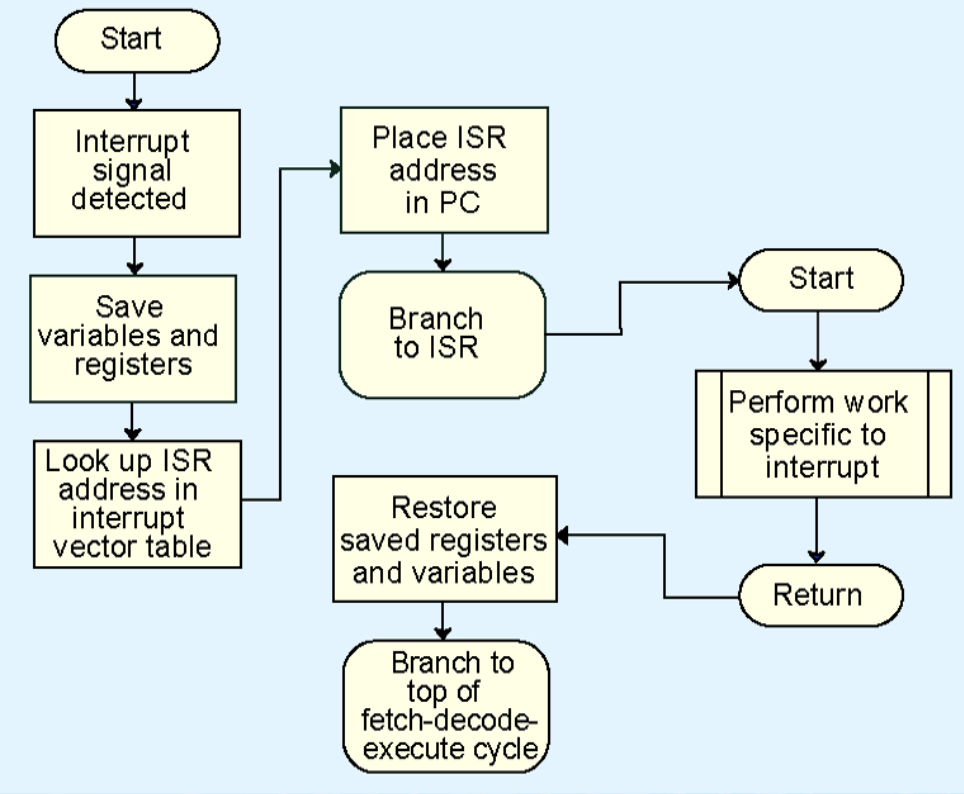
4.9 Instruction Processing (4 of 7)

- 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 (5 of 7)



4.9 Instruction Processing (6 of 7)

- 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 (7 of 7)

- 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 (1 of 3)

- 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 (2 of 3)

- 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 (3 of 3)

- 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 (1 of 4)

- 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 (2 of 4)

- 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 (3 of 4)

- 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
7000	

4.11 A Discussion on Assemblers (4 of 4)

- After the second pass, the assembly is complete.

1104
3105
2106
7000
0023
FFE9
0000

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 (1 of 6)

- 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 (2 of 6)

- 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 (3 of 6)

- The **`ADDI X`** where specifies the address of the operand to be added.
- In RTL:

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

`ADDI X`

4.12 Extending Our Instruction Set (4 of 6)

- 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
```

Does JNS permit recursive calls?

4.12 Extending Our Instruction Set (5 of 6)

- 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**:

AC ← 0

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

4.12 Extending Our Instruction Set (6 of 6)

Example 4.2 on the textbook and Ex4_1.mas in the MARIE simulator package: Using loop to add five numbers.

```
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 20
119 |       DEC 2
11A |       DEC 25
11B |       DEC 30
```

MARIE Assembly Program: Example 4.3

Ex4_2.mas in the MARIE simulator package

If X = Y then
    X = X × 2
else
    Y = Y - X

```
            ORG 100
If,         Load    X       /Load the first value
            Subt    Y       /Subtract the value of Y, store result in AC
            Skipcond 400    /If AC=0, skip the next instruction
            Jump    Else    /Jump to Else part if AC is not equal to 0
Then,       Load    X       /Reload X so it can be doubled
            Add     X       /Double X
            Store   X       /Store the new value
            Jump    Endif   /Skip over the false, or else, part to end of if
Else,       Load    Y       /Start the else part by loading Y
            Subt    X       /Subtract X from Y
            Store   Y       /Store Y-X in Y
Endif,      Halt            /Terminate program (it doesn't do much!)
X,          Dec     12      /Load the loop control variable
Y,          Dec     20      /Subtract one from the loop control variable
            END
```

MARIE Assembly Program: Example 4.4

Ex4_3.mas in the MARIE simulator package

```
/ This program traverses a string and outputs each
/ character. The string is terminated with a null.
/ Note: By changing the output window control setting
/ to "no linefeeds" the text will print in a single
/ line, rather than in a column of single characters.
        ORG 100
Getch,  LoadI       Chptr   / Load the character found at address chptr.
        Skipcond    400     / If the character is a null, we are done.
        Jump        Outp    / Otherwise, proceed with operation.
        Halt
```

MARIE Assembly Program: Example 4.4

```
Outp,    Output          / Output the character.
         Load    Chptr   / Move pointer to next character.
         Add     One
         Store   Chptr
         Jump    Getch
One,     Hex     0001
Chptr,   Hex     10B
String,  Dec     072     / H
         Dec     101     / e
         Dec     108     / l
         Dec     108     / l
         Dec     111     / o
         Dec     032     / [space]
         Dec     119     / w
         Dec     111     / o
         Dec     114     / r
         Dec     108     / l
         Dec     100     / d
         Dec     033     / !
         Dec     000     / [null]
         END
```

MARIE Assembly Program: Example 4.5

Ex4_4.mas in the MARIE simulator package

```
/This example illustrates the use of a simple subroutine to double the value stored at X

        ORG     100
        Load    X       / Load the first number to be doubled.
        Store   Temp    / Use Temp as a parameter to pass value to Subr.
        JnS     Subr    / Store the return address, and jump to the procedure.
        Store   X       / Store the first number, doubled
        Load    Y       / Load the second number to be doubled.
        Store   Temp
        JnS     Subr    / Store the return address and jump to the procedure.
Loop,   Store Y         / Store the second number doubled.
        Halt            / End program.
X,      DEC     20
Y,      DEC     48
Temp,   DEC     0
```

MARIE Assembly Program: Example 4.5

```
Subr,   HEX     0       / Store return address here.
        Load    Temp    / Actual subroutine to double numbers.
        Add     Temp    / AC now holds double the value of Temp.
        JumpI   Subr    / Return to calling code.
        END
```

4.13 A Discussion on Decoding (1 of 21)

- A computer's control unit keeps things synchronized, making sure that bits flow to the correct components as the components are needed.
- There are two general ways in which a control unit can be implemented: *hardwired control* and *microprogrammed* control.
  - With microprogrammed control, a small program is placed into read-only memory in the microcontroller.
  - Hardwired controllers implement this program using digital logic components.

4.13 A Discussion on Decoding (2 of 21)

- Your text provides a complete list of the register transfer language for each of MARIE's instructions.
- The microoperations given by each RTL define the operation of MARIE's control unit.
- Each microoperation consists of a distinctive signal pattern that is interpreted by the control unit and results in the execution of an instruction.
  - Recall, the RTL for the Add instruction is:

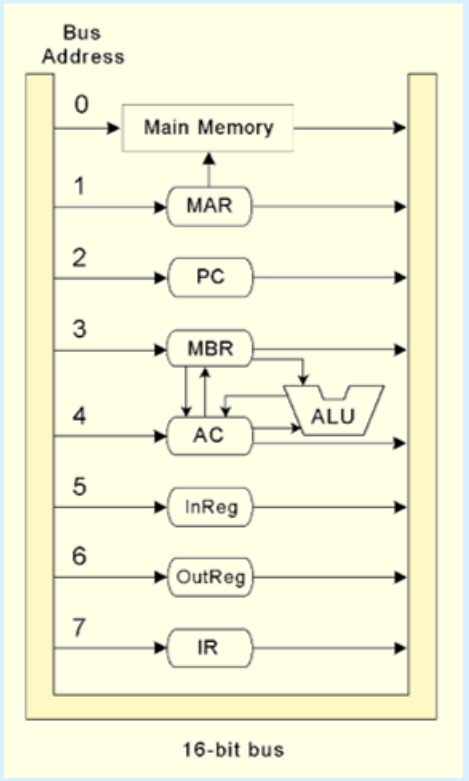
$$\mathbf{MAR \leftarrow X}$$

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

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

4.13 A Discussion on Decoding (3 of 21)

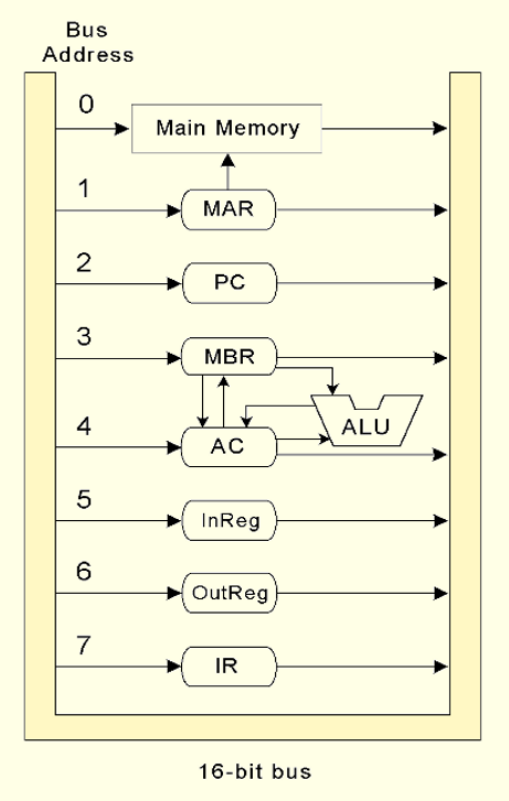
- Each of MARIE's registers and main memory have a unique address along the datapath.
- The addresses take the form of signals issued by the control unit.



How many signal lines does MARIE's control unit need?

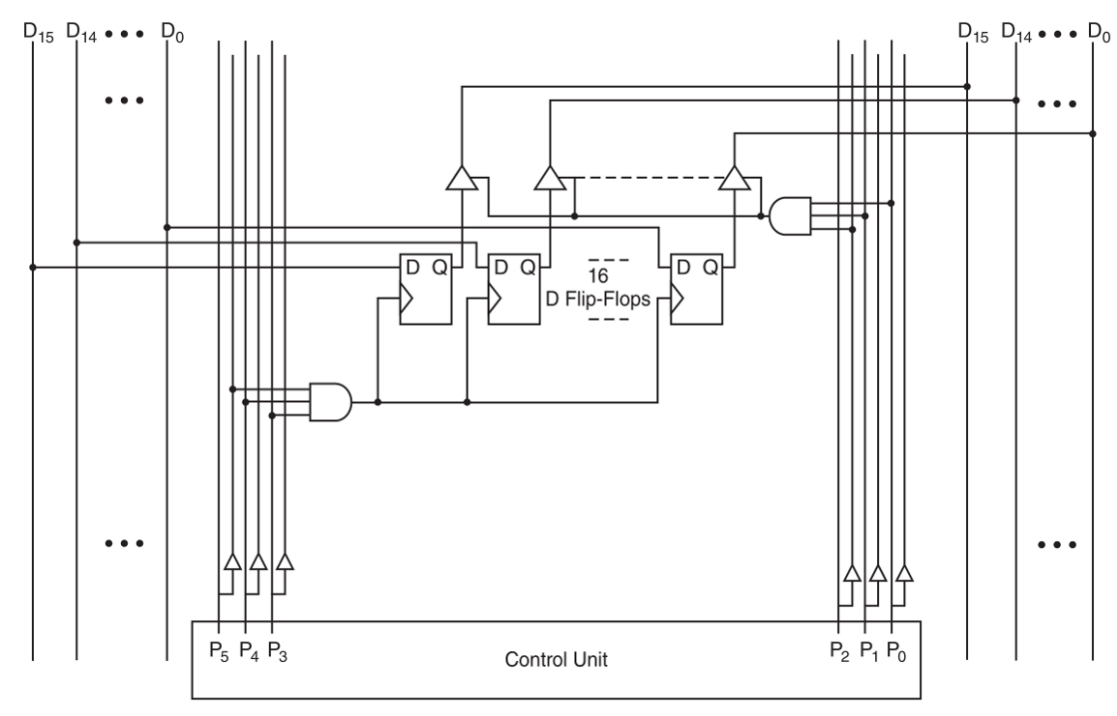
4.13 A Discussion on Decoding (4 of 21)

- Let us define two sets of three signals.
- One set, P_2, P_1, P_0, controls reading from memory or a register, and the other set consisting of P_5, P_4, P_3, controls writing to memory or a register.



The next slide shows a close up view of MARIE's MBR.

4.13 A Discussion on Decoding (5 of 21)



The register MBR is enabled for reading when P_0 and P_1 are high, and enabled for writing when P_3 and P_4 are high.

4.13 A Discussion on Decoding (6 of 21)

- Careful inspection of MARIE's RTL reveals that the ALU has only three operations: add, subtract, and clear.
  - We will also define a fourth "do nothing" state.
- The entire set of MARIE's control signals consists of:
  - Register controls: P_0 through P_5, M_R , and M_W.
  - ALU controls: A_0 through A_1 and L_{ALT} to control the ALU's data source.
  - Timing: T_0 through T_7 and counter reset C_r

ALU Control Signals		ALU Response
A_1	A_0	
0	0	Do Nothing
0	1	AC ← AC + MBR
1	0	AC ← AC – MBR
1	1	AC ← 0 (Clear)

4.13 A Discussion on Decoding (7 of 21)

- Consider MARIE's Add instruction. Its RTL is:

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

- After an Add instruction is fetched, the address, X, is in the rightmost 12 bits of the IR, which has a datapath address of 7.
- X is copied to the MAR, which has a datapath address of 1.
- Thus we need to raise signals P_0, P_1, and P_2 to read from the IR, and signal P_3 to write to the MAR.

4.13 A Discussion on Decoding (8 of 21)

- Here is the complete signal sequence for MARIE's Add instruction:

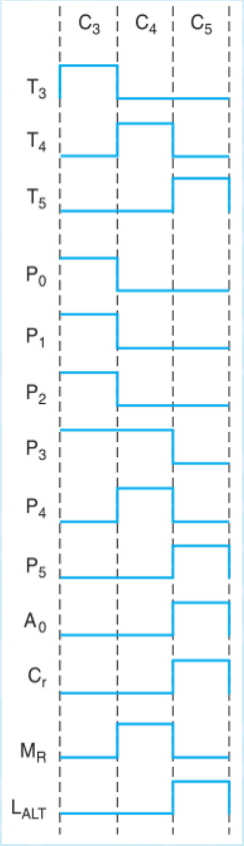
$$P_3\ P_2\ P_1\ P_0\ T_3 \quad : \quad MAR \leftarrow X$$
$$P_4\ P_3\ T_4\ M_R \quad : \quad MBR \leftarrow M[MAR]$$
$$C_r\ A_0\ P_5\ T_5\ L_{ALT} \quad : \quad AC \leftarrow AC + MBR$$
$$[Reset\ counter]$$

- These signals are ANDed with combinational logic to bring about the desired machine behavior.
- The next slide shows the timing diagram for this instruction.

4.13 A Discussion on Decoding (9 of 21)

- Notice the concurrent signal states during each machine cycle: C_3 through C_5.

$P_3\ P_2\ P_1\ P_0\ T_3$: MAR ← X
$P_4\ P_3\ T_4\ M_R$: MBR ← M[MAR]
$C_r\ A_0\ P_5\ T_5\ L_{ALT}$: AC ← AC + MBR
[Reset counter]

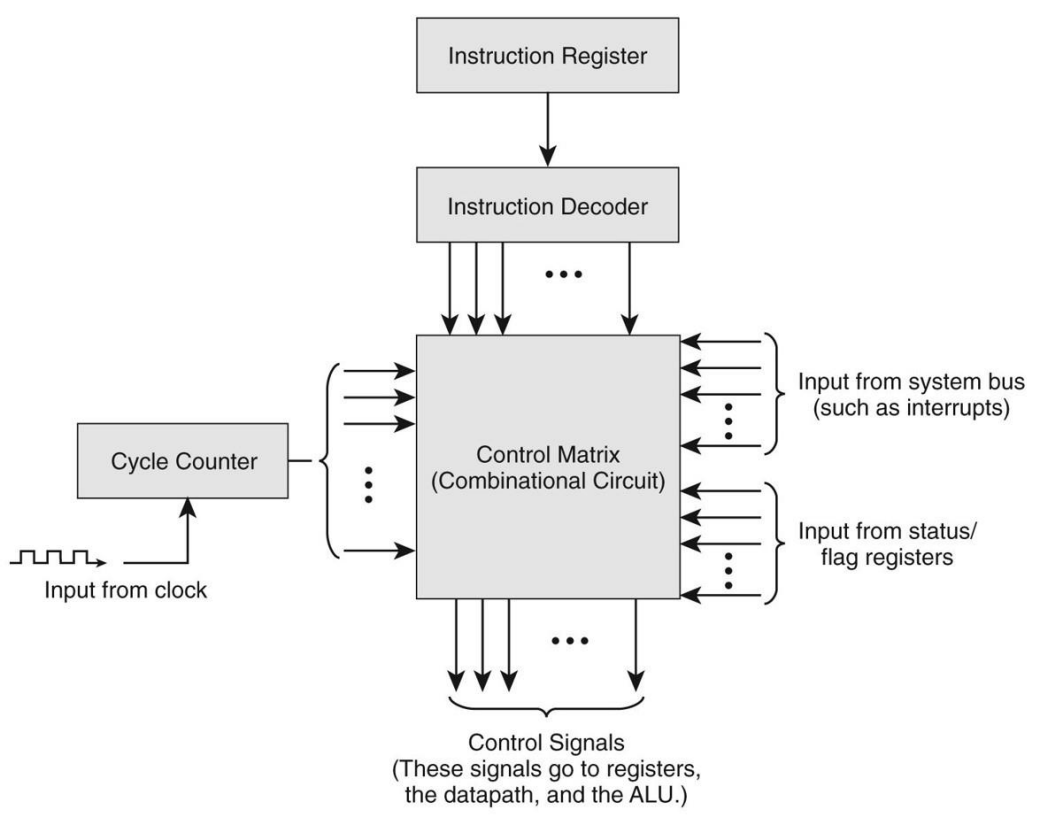


4.13 A Discussion on Decoding (10 of 21)

- We note that the signal pattern just described is the same whether our machine used hardwired or microprogrammed control.
- In *hardwired control*, the bit pattern of machine instruction in the IR is decoded by combinational logic.
- The decoder output works with the control signals of the current system state to produce a new set of control signals.

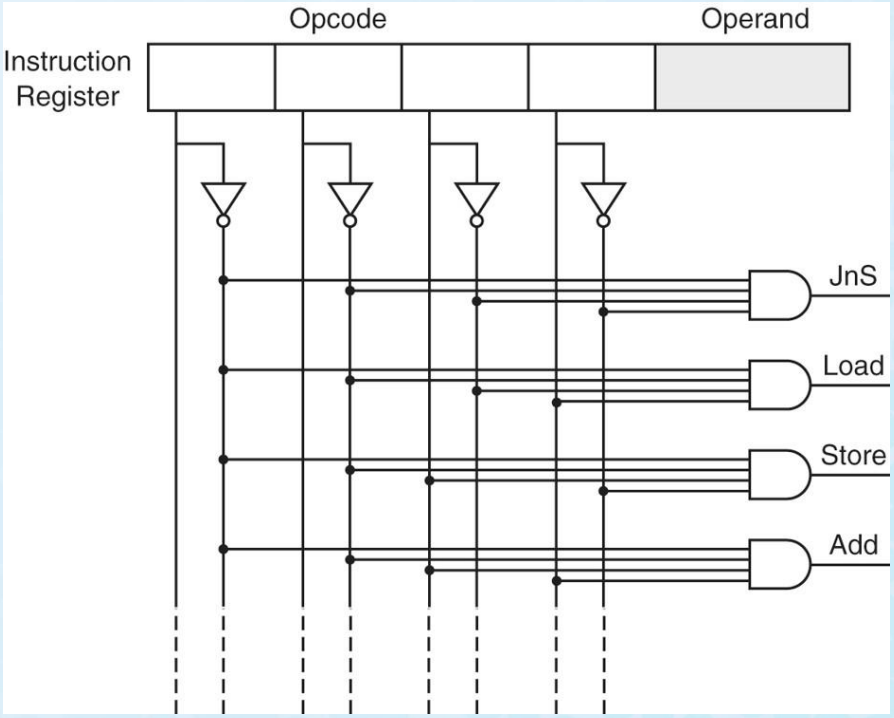
A block diagram of a hardwired control unit is shown on the following slide.

4.13 A Discussion on Decoding (11 of 21)



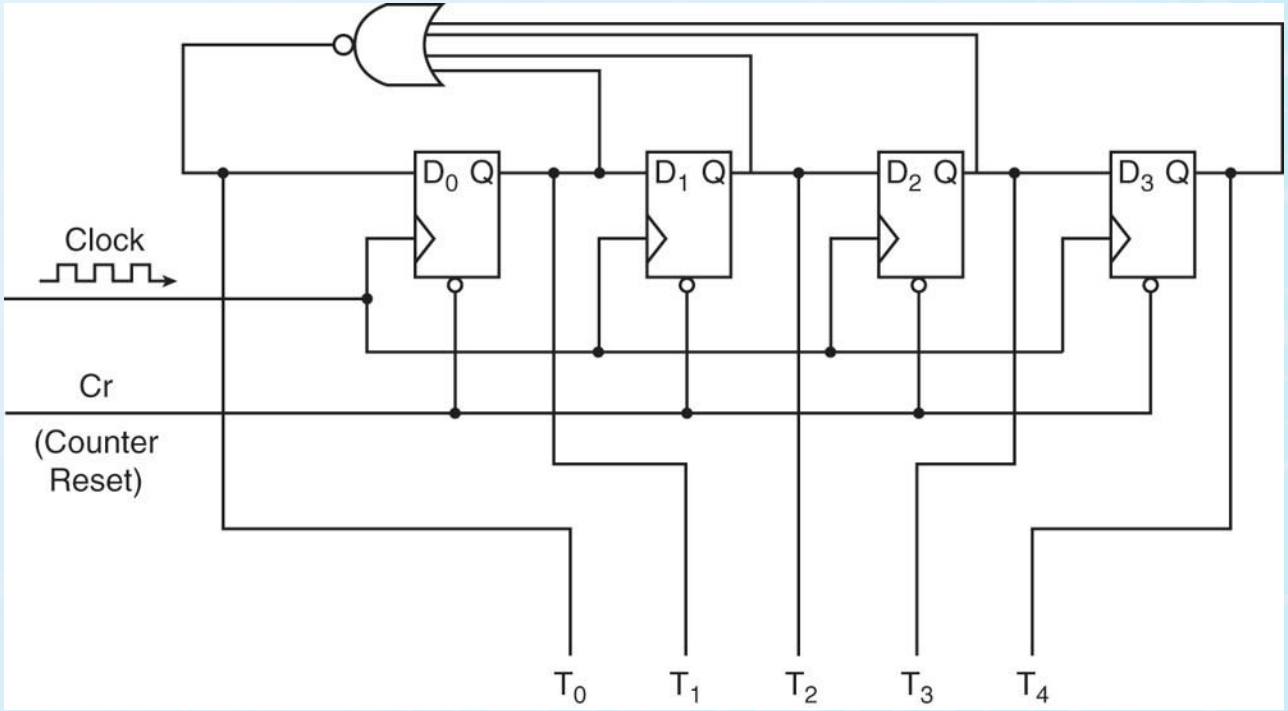
4.13 A Discussion on Decoding (12 of 21)

- MARIE's instruction decoder. (Partial.)



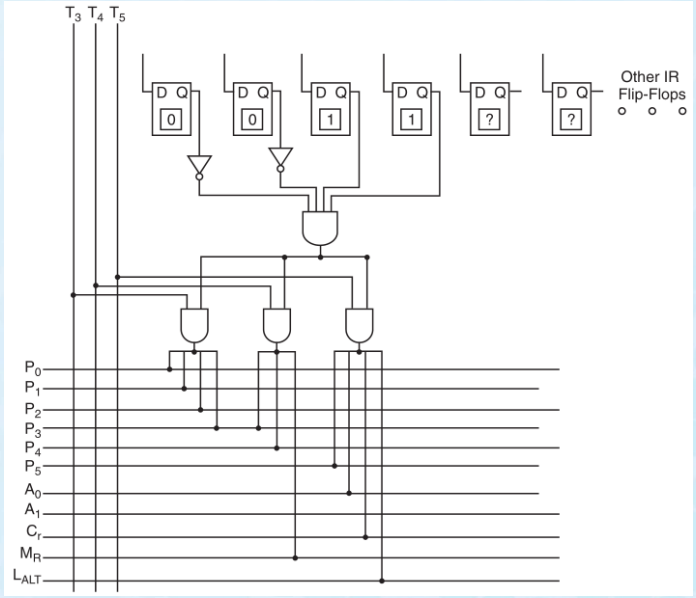
4.13 A Discussion on Decoding (13 of 21)

- A ring counter that counts from 0 to 5



4.13 A Discussion on Decoding (14 of 21)

- This is the hardwired logic for MARIE's Add = 0011 instruction.

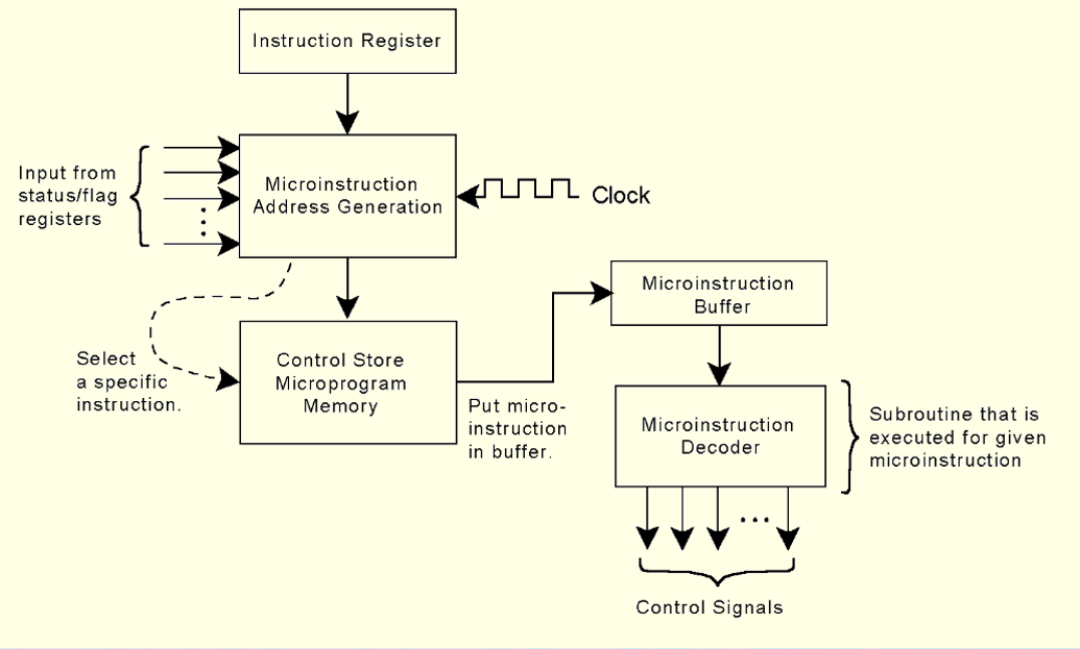


4.13 A Discussion on Decoding (15 of 21)

- In microprogrammed control, instruction microcode produces control signal changes.
- Machine instructions are the input for a microprogram that converts the 1s and 0s of an instruction into control signals.
- The microprogram is stored in firmware, which is also called the control store.
- A microcode instruction is retrieved during each clock cycle.

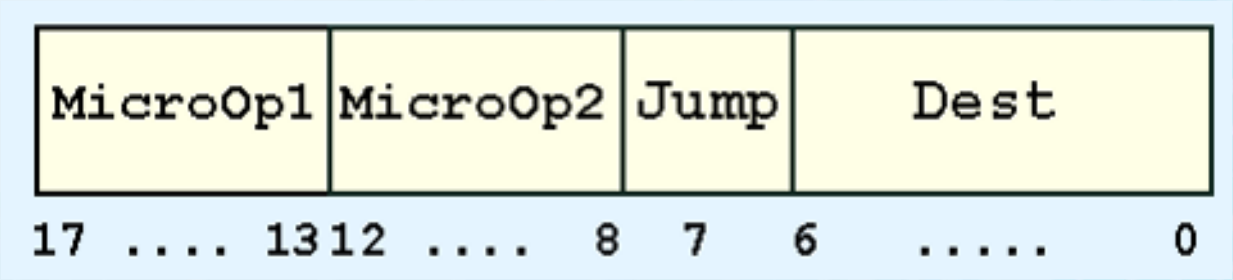
4.13 A Discussion on Decoding (16 of 21)

- This is how a generic microprogrammed control unit might look.



4.13 A Discussion on Decoding (17 of 21)

- If MARIE were microprogrammed, the microinstruction format might look like this:



- **`MicroOp1`** and **`MicroOp2`** contain binary codes for each instruction. **`Jump`** is a single bit indicating that the value in the **`Dest`** field is a valid address and should be placed in the microsequencer.

4.13 A Discussion on Decoding (18 of 21)

- The table below contains MARIE's microoperation codes along with the corresponding RTL:

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01101	MBR ← M[MAR]
00001	AC ← 0	01110	OutREG ← AC
00010	AC ← MBR	01111	PC ← IR[11-0]
00011	AC ← AC - MBR	10000	PC ← MBR
00100	AC ← AC + MBR	10001	PC ← PC + 1
00101	AC ← InREG	10010	If AC = 00
00110	IR ← M[MAR]	10011	If AC > 0
00111	M[MAR] ← MBR	10100	If AC < 0
01000	MAR ← IR[11-0]	10101	If IR[11-10] = 00
01001	MAR ← MBR	10110	If IR[11-10] = 01
01010	MAR ← PC	10111	If IR[11-10] = 10
01011	MAR ← X	11000	If IR[15-12] = MicroOp2[4-1]
01100	MBR ← AC		

4.13 A Discussion on Decoding (19 of 21)

- The first nine lines of MARIE's microprogram are given below (using RTL for clarity):

Address	MicroOp 1	MicroOp 2	Jump	Dest
0000000	MAR ← PC	NOP	0	0000000
0000001	IR ← M[MAR]	NOP	0	0000000
0000010	PC ← PC + 1	NOP	0	0000000
0000011	MAR ←IR[11-0]	NOP	0	0000000
0000100	If IR[15-12] = MicroOP2[4-1]	00000	1	0100000
0000101	If IR[15-12] = MicroOP2[4-1]	00010	1	0100111
0000110	If IR[15-12] = MicroOP2[4-1]	00100	1	0101010
0000111	If IR[15-12] = MicroOP2[4-1]	00110	1	0101100
0001000	If IR[15-12] = MicroOP2[4-1]	01000	1	0101111
. . .	. . .	. . .	. . .	. . .

4.13 A Discussion on Decoding (20 of 21)

- The first four lines are the fetch-decode-execute cycle.
- The remaining lines are the beginning of a jump table.

Address	MicroOp 1	MicroOp 2	Jump	Dest
0000000	MAR ← PC	NOP	0	0000000
0000001	IR ← M[MAR]	NOP	0	0000000
0000010	PC ← PC + 1	NOP	0	0000000
0000011	MAR ←IR[11-0]	NOP	0	0000000
0000100	If IR[15-12] = MicroOP2[4-1]	00000	1	0100000
0000101	If IR[15-12] = MicroOP2[4-1]	00010	1	0100111
0000110	If IR[15-12] = MicroOP2[4-1]	00100	1	0101010
0000111	If IR[15-12] = MicroOP2[4-1]	00110	1	0101100
0001000	If IR[15-12] = MicroOP2[4-1]	01000	1	0101111
...	...	...	...	...

4.13 A Discussion on Decoding (21 of 21)

- It's important to remember that a microprogrammed control unit works like a system-in-miniature.
- Microinstructions are fetched, decoded, and executed in the same manner as regular instructions.
- This extra level of instruction interpretation is what makes microprogrammed control slower than hardwired control.
- The advantages of microprogrammed control are that it can support very complicated instructions and only the microprogram needs to be changed if the instruction set changes (or an error is found).

4.14 Real-World Architectures (1 of 7)

- MARIE shares many features with modern architectures but it is not an accurate depiction of them.
- In the following slides, we briefly examine two machine architectures.
- We will look at an Intel architecture, which is a CISC machine and MIPS, which is a RISC machine.
  - CISC is an acronym for complex instruction set computer.
  - RISC stands for reduced instruction set computer.

We delve into the “RISC versus CISC” argument in Chapter 9.

4.14 Real-World Architectures (2 of 7)

- The classic Intel architecture, the 8086, was born in 1979. It is a CISC architecture.
- It was adopted by IBM for its famed PC, which was released in 1981.
- The 8086 operated on 16-bit data words and supported 20-bit memory addresses.
- Later, to lower costs, the 8-bit 8088 was introduced. Like the 8086, it used 20-bit memory addresses.

What was the largest memory that the 8086 could address?

4.14 Real-World Architectures (3 of 7)

- The 8086 had four 16-bit general-purpose registers that could be accessed by the half-word.
- It also had a flags register, an instruction register, and a stack accessed through the values in two other registers, the base pointer and the stack pointer.
- The 8086 had no built in floating-point processing.
- In 1980, Intel released the 8087 numeric coprocessor, but few users elected to install them because of their high cost.

4.14 Real-World Architectures (4 of 7)

- In 1985, Intel introduced the 32-bit 80386.
- It also had no built-in floating-point unit.
- The 80486, introduced in 1989, was an 80386 that had built-in floating-point processing and cache memory.
- The 80386 and 80486 offered downward compatibility with the 8086 and 8088.
- Software written for the smaller-word systems was directed to use the lower 16 bits of the 32-bit registers.

4.14 Real-World Architectures (5 of 7)

- Intel's Pentium 4 introduced a brand new NetBurst architecture.
- Speed enhancing features include:
  - Hyperthreading
  - Hyperpipelining
  - Wider instruction pipeline
  - Execution trace cache (holds decoded instructions for possible reuse) multilevel cache and instruction pipelining.
- Intel, along with many others, is marrying many of the ideas of RISC architectures with microprocessors that are largely CISC.

4.14 Real-World Architectures (6 of 7)

- The MIPS family of CPUs has been one of the most successful in its class.
- In 1986 the first MIPS CPU was announced.
- It had a 32-bit word size and could address 4GB of memory.
- Over the years, MIPS processors have been used in general purpose computers as well as in games.
- The MIPS architecture now offers 32- and 64-bit versions.

4.14 Real-World Architectures (7 of 7)

- MIPS was one of the first RISC microprocessors.
- The original MIPS architecture had only 55 different instructions, as compared with the 8086 which had over 100.
- MIPS was designed with performance in mind: It is a *load/store* architecture, meaning that only the load and store instructions can access memory.
- The large number of registers in the MIPS architecture keeps bus traffic to a minimum.

How does this design affect performance?

Conclusion (1 of 2)

- The major components of a computer system are its control unit, registers, memory, ALU, and data path.
- A built-in clock keeps everything synchronized.
- Control units can be microprogrammed or hardwired.
- Hardwired control units give better performance, while microprogrammed units are more adaptable to changes.

Conclusion (2 of 2)

- Computers run programs through iterative fetch-decode-execute cycles.
- Computers can run programs that are in machine language.
- An assembler converts mnemonic code to machine language.
- The Intel architecture is an example of a CISC architecture; MIPS is an example of a RISC architecture.