Introduction to Simple Computer Part II

Hui Chen

Computer & Information Science

CUNY Brooklyn College

Fall 2023

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.

Introducing MARIE

- Bring together the ideas discussed to this point using a very simple model computer.
- MARIE Machine Architecture (Really Intuitive and Easy, or MARIE)
  - designed for the singular purpose of illustrating basic computer system concepts.
- This system is very simple with limited potential.
- Nonetheless, a deep understanding of its functions will enable you to comprehend system architectures that are much more complex.

Characteristics

- 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.

MARIE’s Registers: (1) – (3)

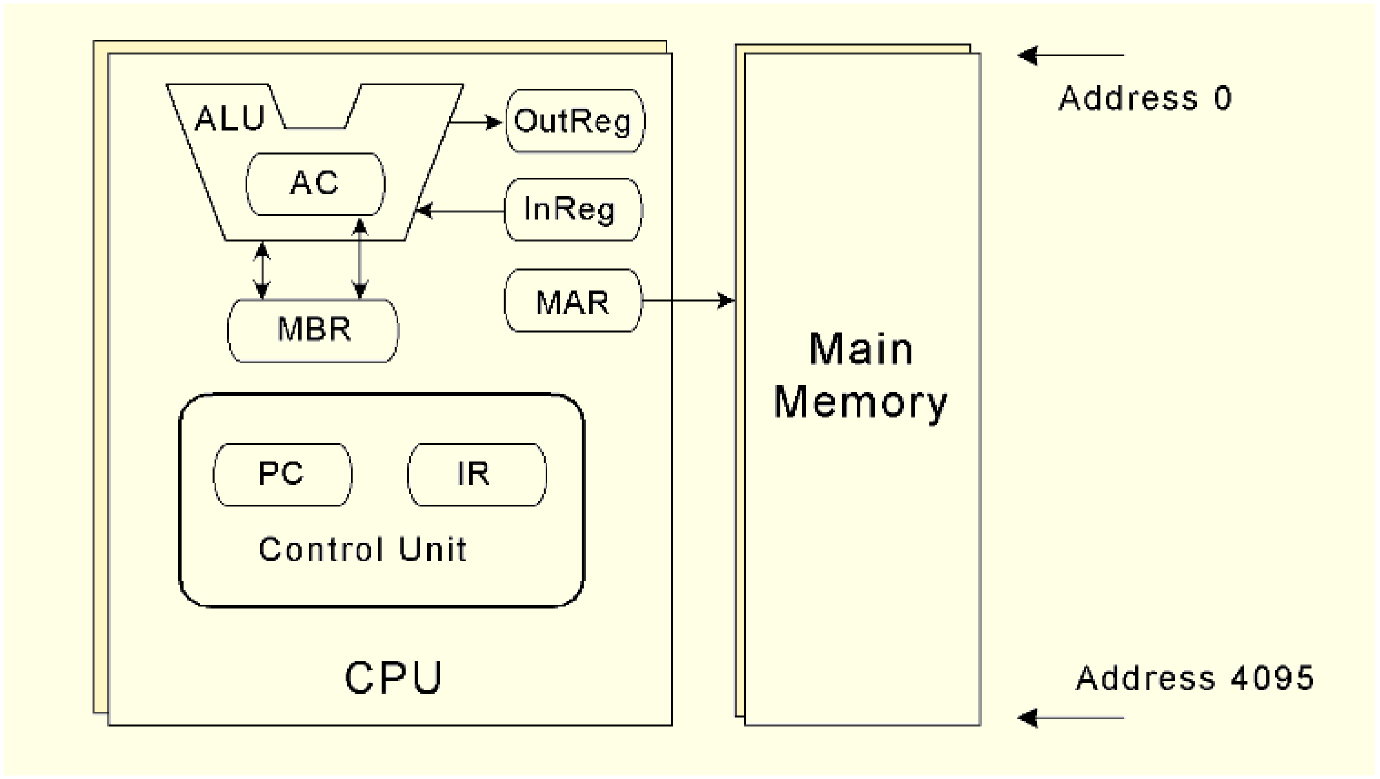
- MARIE’s seven registers are:
  - (1) Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
  - (2) Memory address register, MAR, a 12-bit register that holds the memory address of an instruction or the operand of an instruction.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

MARIE’s Registers: (4) – (7)

- (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.

MARIE Architecture

- This is the MARIE architecture shown graphically.

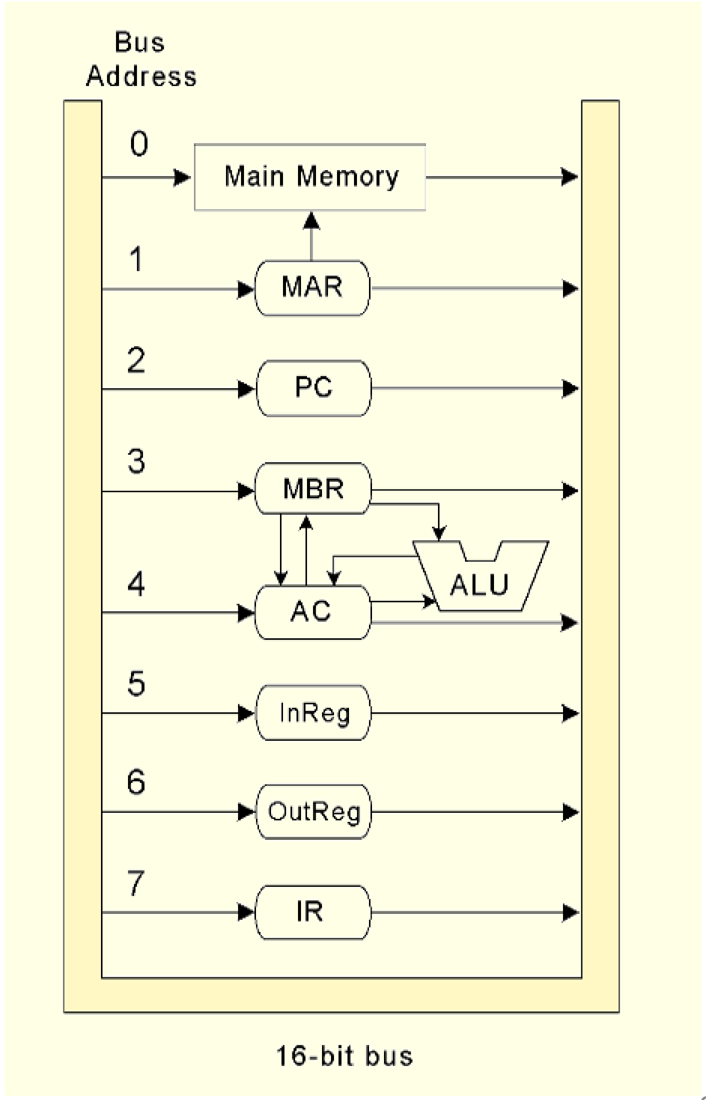


MARIE Architecture

- 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.

MARIE’s 16-bit BUS

- This is the MARIE data path show graphically.

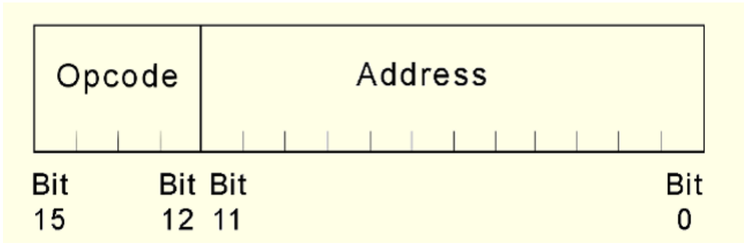


Instruction Set

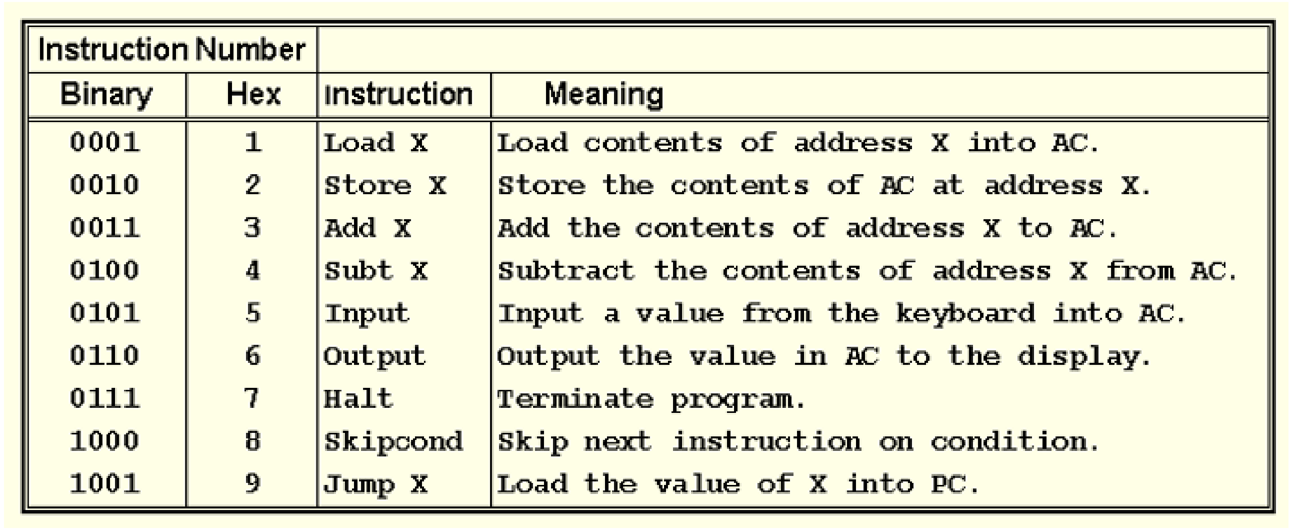
- A computer’s instruction set architecture (ISA) specifies
  - the format of its instructions, and
  - the primitive operations that the machine can perform.
- The ISA is an interface between a computer’s hardware and its software.
- Some ISAs include hundreds of different instructions for processing data and controlling program execution.
- The MARIE ISA consists of only 13 instructions.

Format of MARIE Instruction

- This is the format of a MARIE instruction:



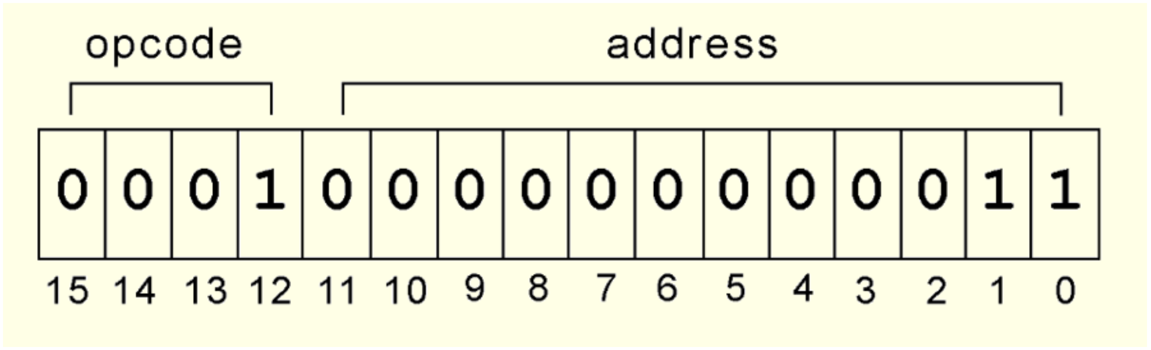
- The fundamental MARIE instructions are:



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.

Instruction Example: LOAD

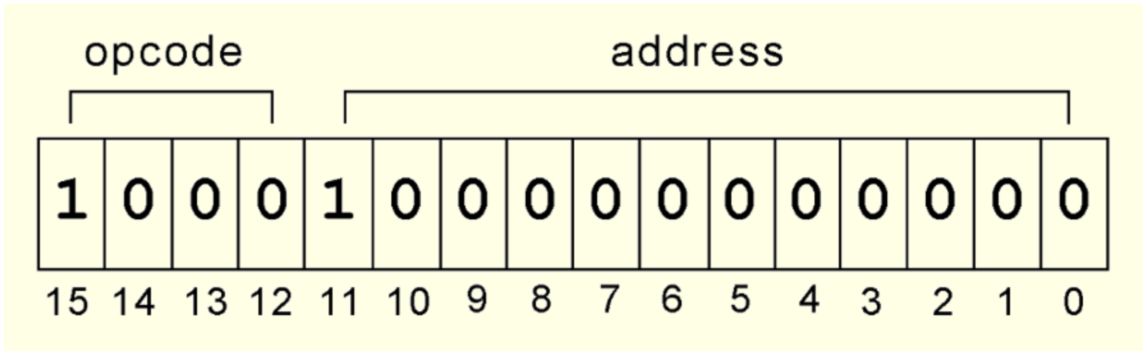
- This is a bit pattern for a **`LOAD`** instruction as it would appear in the IR:



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

Instruction Example: SKIPCOND

- 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.

Microoperations and RTL

- 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.

RTL: Example

- 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
```

More about SKIPCOND using RTL

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

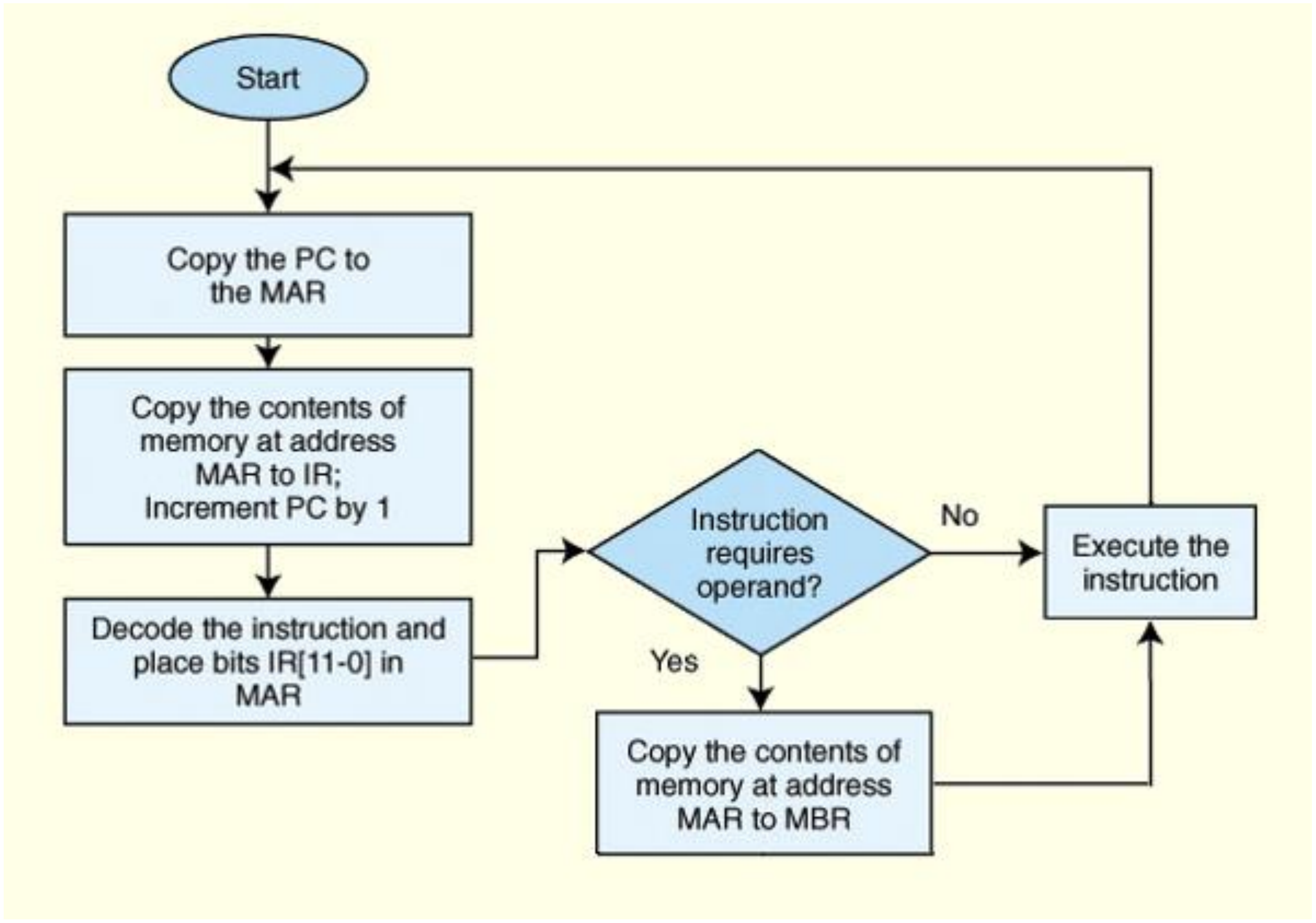
```
If IR[11 - 10] = 00 then
    If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
    If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 11 then
    If AC > 0 then PC ← PC + 1
```

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.

Instruction Processing: Float Chart

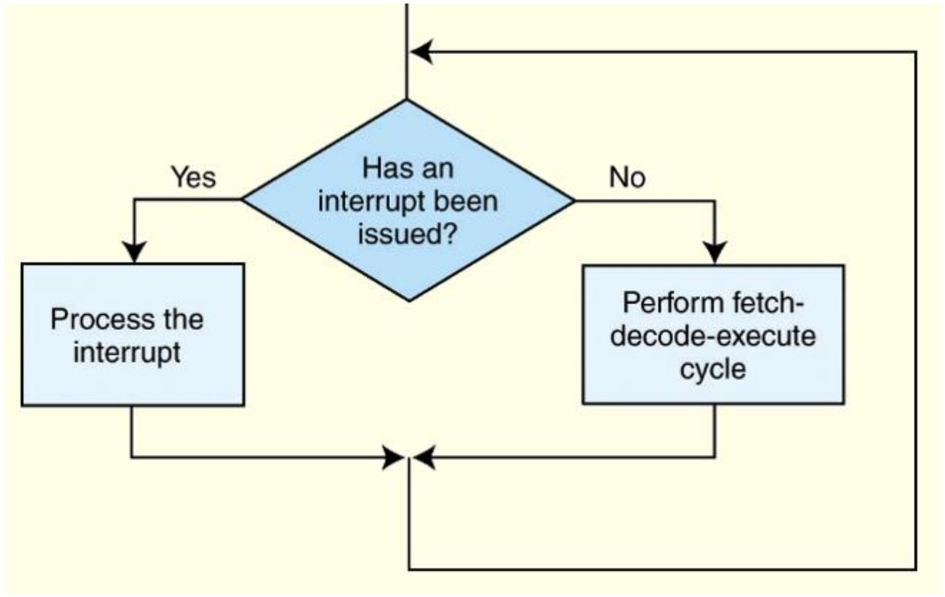

Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR; Increment PC by 1
Decode the instruction and place bits IR[11-0] in MAR
Instruction requires operand?
No
Yes
Execute the instruction
Copy the contents of memory at address MAR to MBR

Instruction Processing: Interrupts

- 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*.

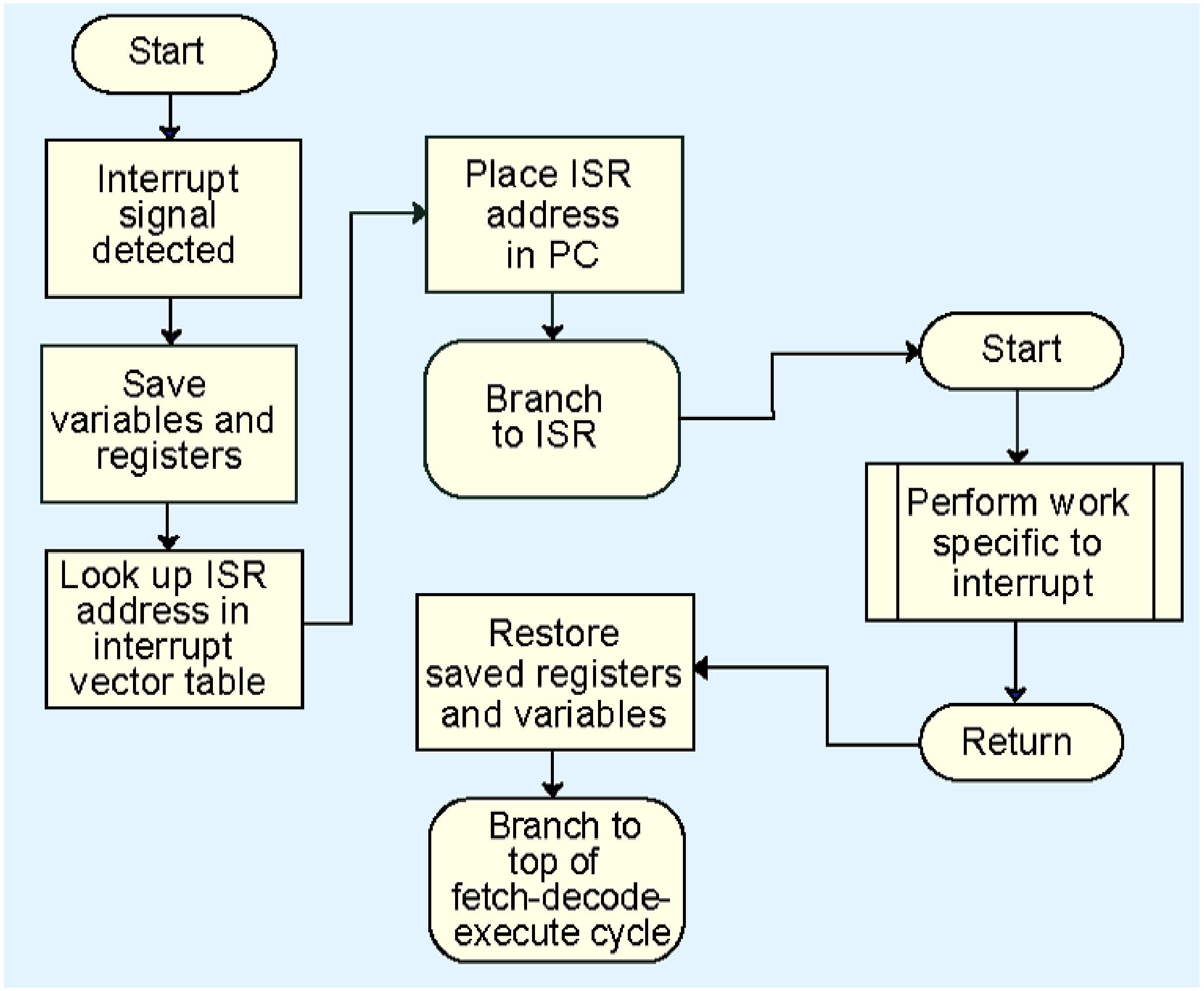
Modifying Instruction Processing for Interrupts

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


Has an interrupt been issued?
Yes
No
Process the interrupt
Perform fetch-decode-execute cycle

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

Interrupt Processing


Start
Interrupt signal detected
Save variables and registers
Look up ISR address in interrupt vector table
Place ISR address in PC
Branch to ISR
Start
Perform work specific to interrupt
Return
Restore saved registers and variables
Branch to top of fetch-decode-execute cycle

More about Interrupts Processing

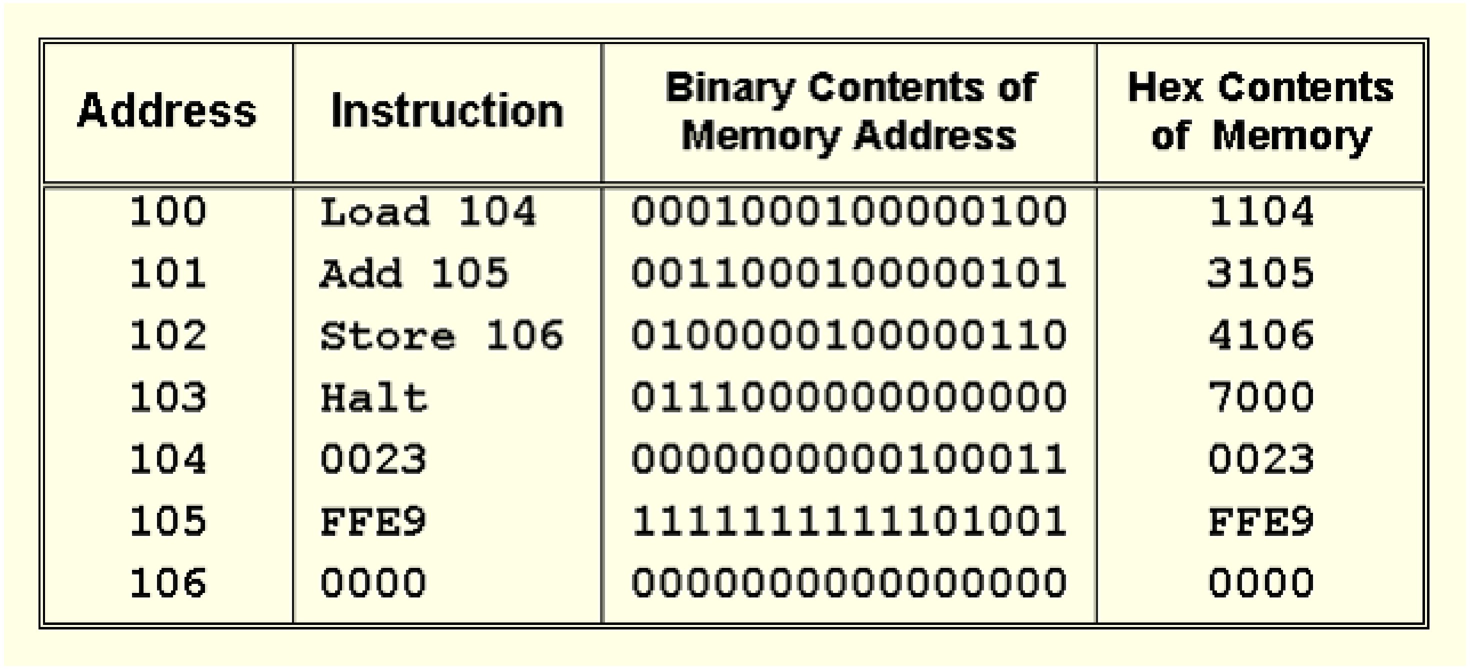
- For general-purpose systems, it is common to disable all interrupts during the time in which an interrupt is being processed.
    - Typically, this is achieved by setting a bit in the flags register.
- Interrupts that are ignored in this case are called *maskable*.
- *Nonmaskable* interrupts are those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

Interrupts and I/O

- 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 later.
- 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.

A Simple Program on MARIE

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

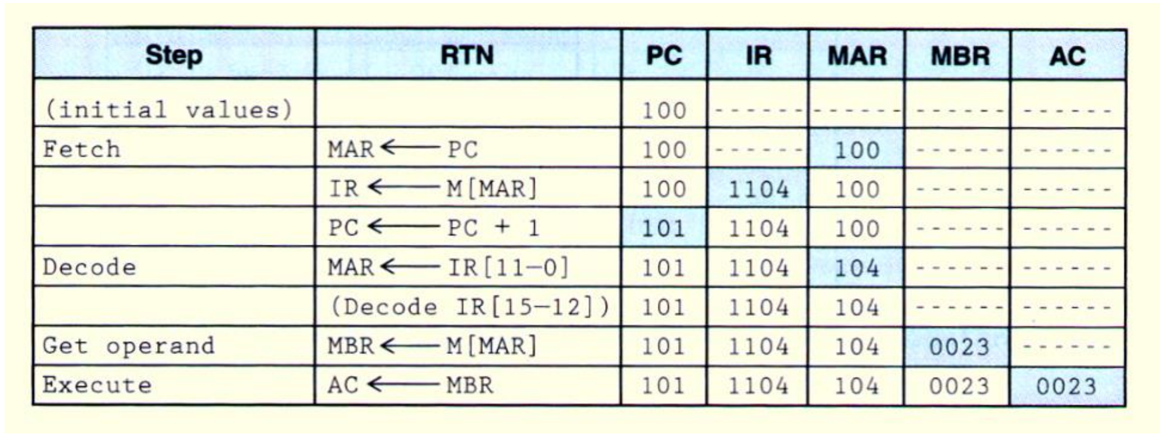


Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000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

All numbers are hexadecimals

Running the Simple Program: 1st Instruction

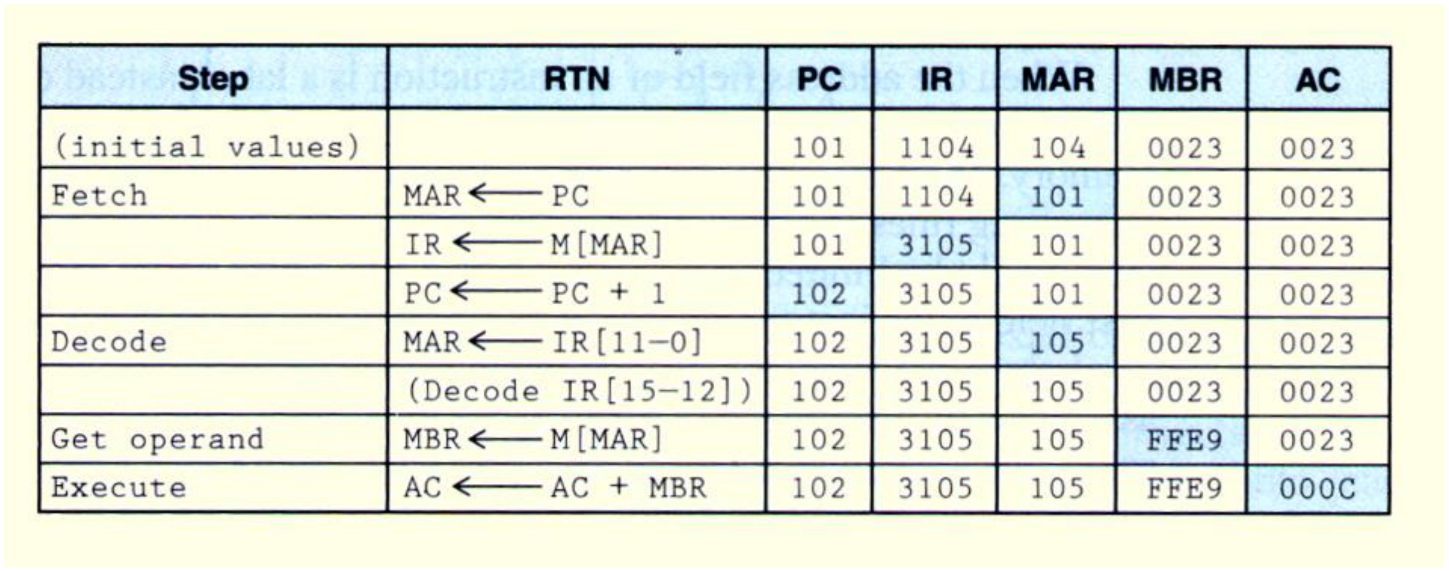
- Let’s look at what happens inside the computer when our program runs.
- This is the **`LOAD 104`** instruction:



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

Running the Simple Program: 2nd Instruction

- 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

Introducing 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.

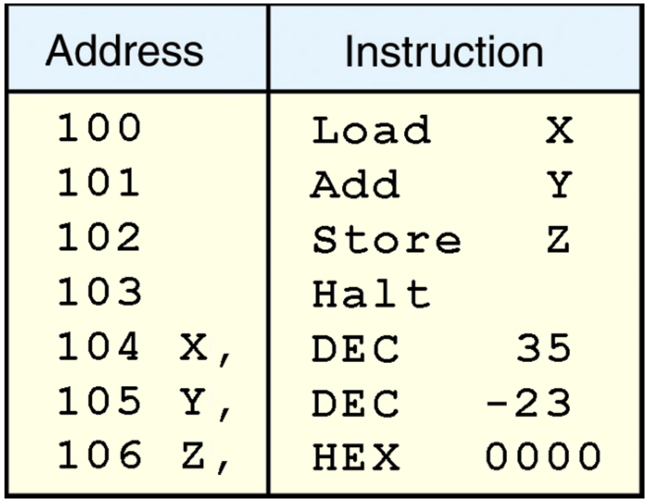
Assembling Process

- Assemblers create an *object program file* from mnemonic *source code* in two passes.

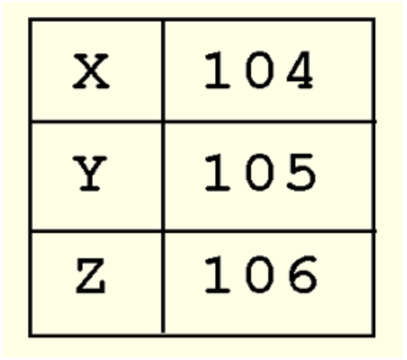
1. During the first pass, the assembler assembles as much of the program as it can, while it builds a *symbol table* that contains memory references for all symbols in the program.
2. During the second pass, the instructions are completed using the values from the symbol table.

Example: Pass 1

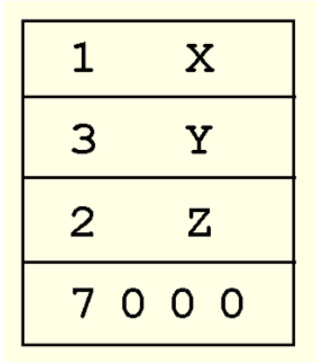
- Consider our example program at the right.
  - Note that we have included two directives **HEX** and **DEC** that specify the radix of the constants.
- The first pass, creates a symbol table and the partially-assembled instructions as shown.



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



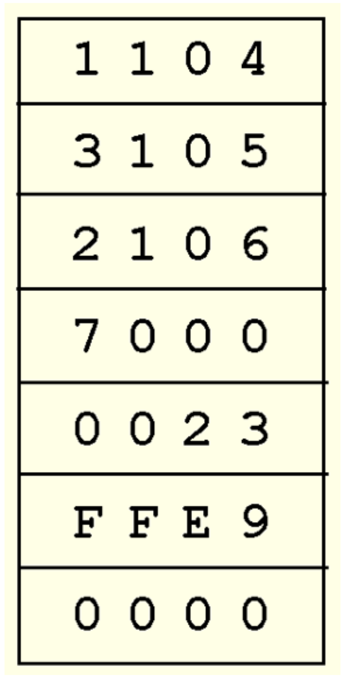
X	104
Y	105
Z	106



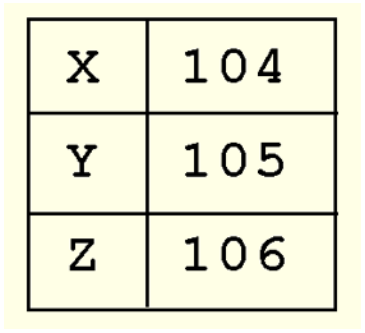
1	X
3	Y
2	Z
7 0 0 0	

Example: Pass 2

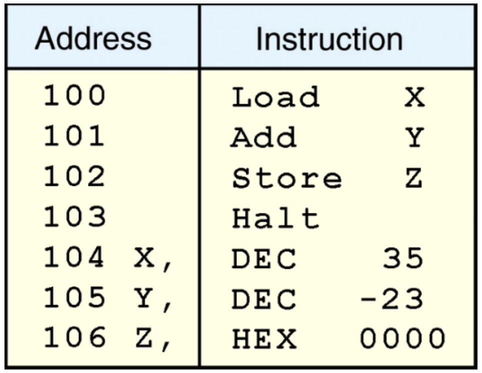
- After the second pass, the assembly is complete.



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

Indirect Addressing

- 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.

Extending Our Instruction Set with Indirect Addressing

- We have included three indirect addressing mode instructions in the MARIE instruction set.
    - LOADI X
    - STOREI X
    - ADDI X
    - JNS X
    - CLEAR

LOADI X

- In RTL :

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

STOREI X

- In RTL :

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

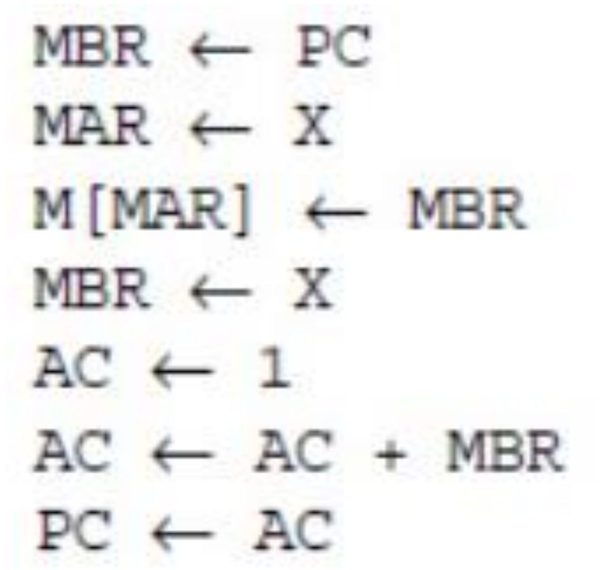
ADDI X

- 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
```

Subroutines

- 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
```

CLEAR

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

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

A Discussion on Decoding

- 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.
  - Hardwired controllers implement this program using digital logic components.
  - With microprogrammed control, a small program is placed into read-only memory in the microcontroller.

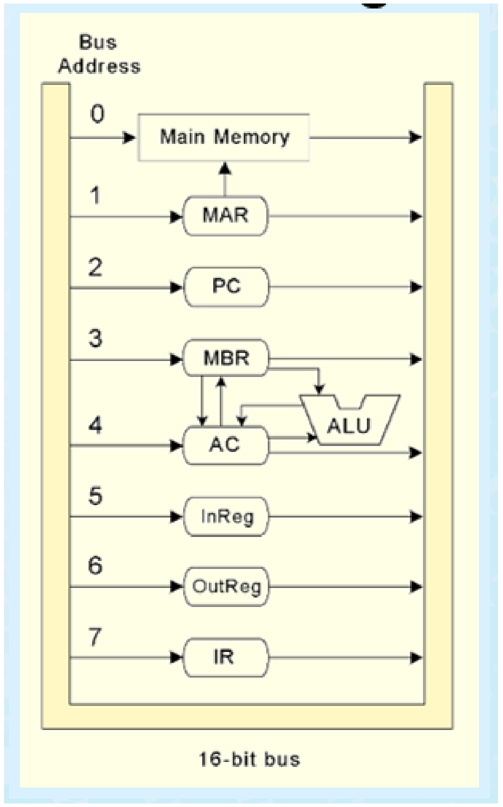
Control Unit

- 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:

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

Decoding

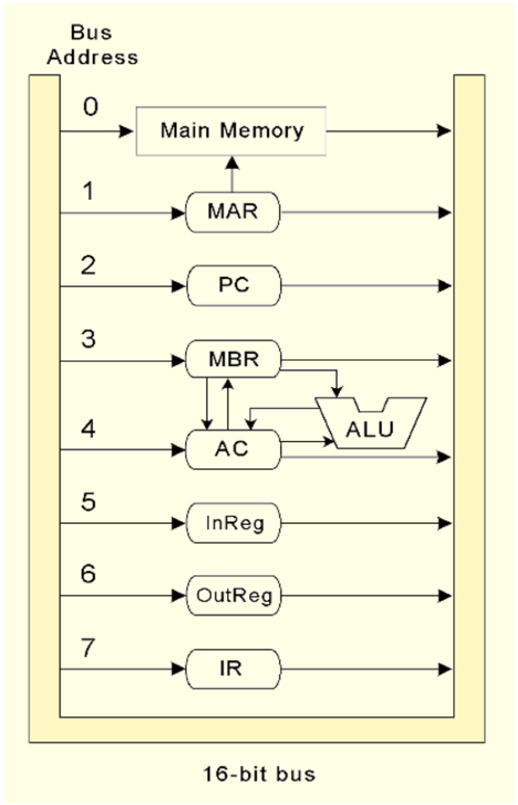
- Each of MARIE's registers and main memory have a unique address along the datapath.
- The addresses take the form of signals issued by the control unit.



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

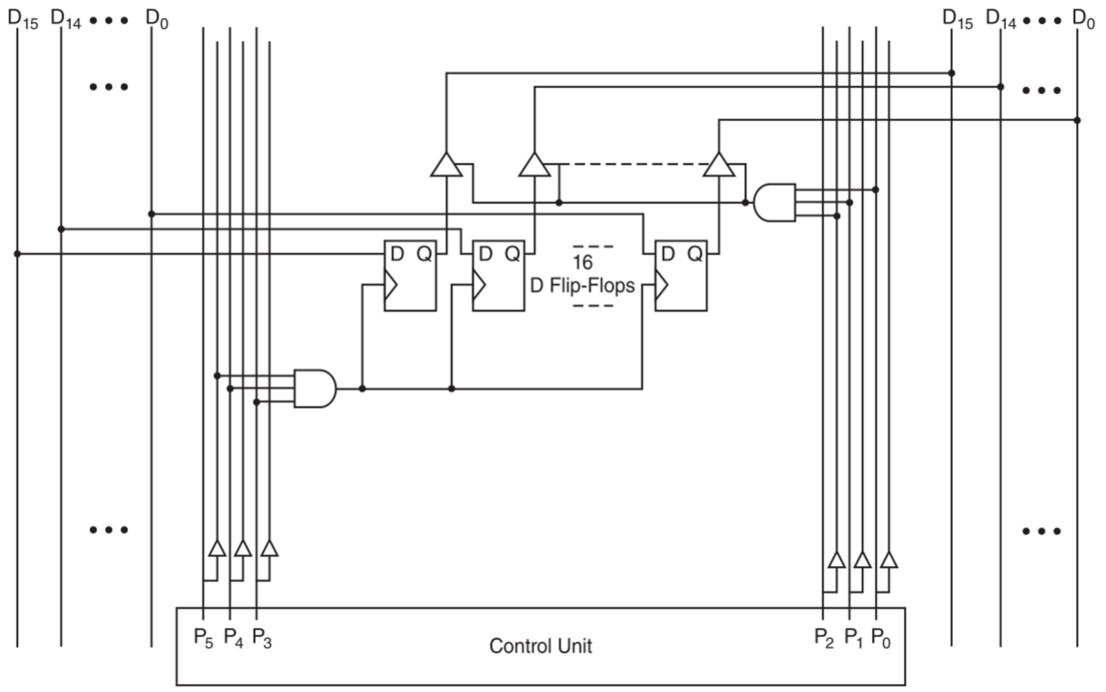
Control Signals

- 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.

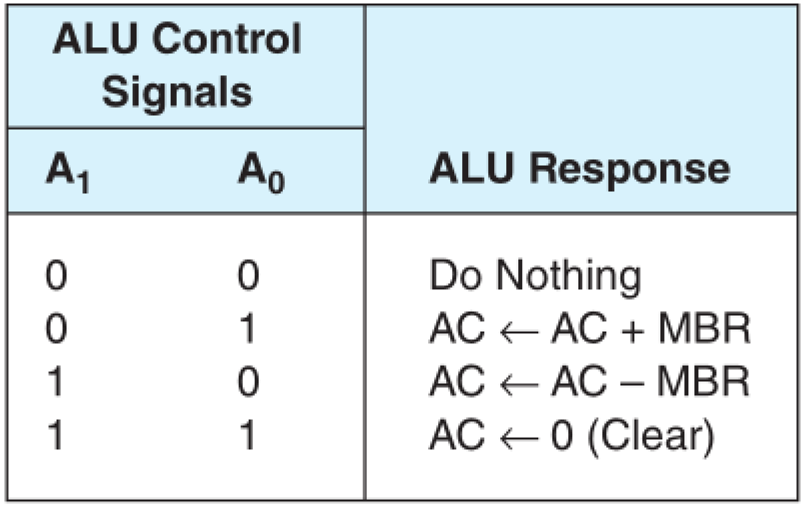
Decoding: Example Circuit



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

Control Signals

- 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		
A_1	A_0	ALU Response
0	0	Do Nothing
0	1	$AC \leftarrow AC + MBR$
1	0	$AC \leftarrow AC - MBR$
1	1	$AC \leftarrow 0$ (Clear)

Control Signaling: Example

- 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.

Control Signaling: Example

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

$P_3\ P_2\ P_1\ P_0\ T_3$: 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]

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

Timing: Example

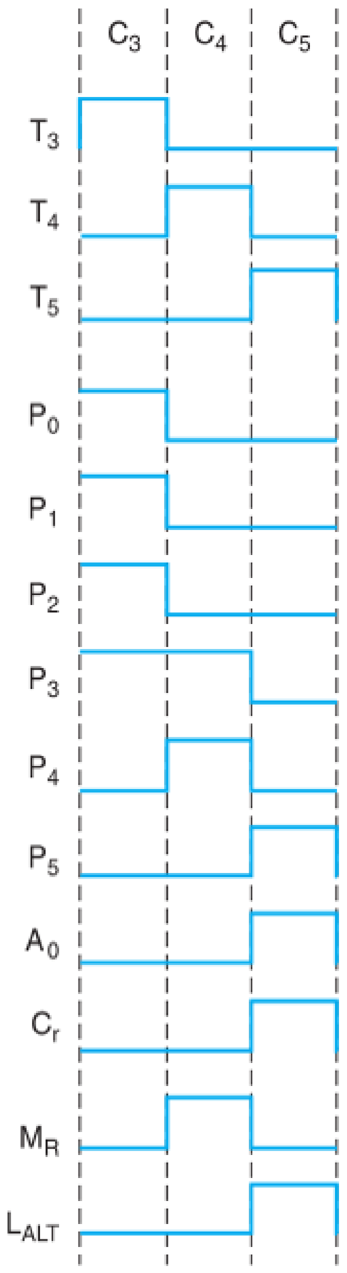
- Notice the concurrent signal states during each machine cycle: C_0 through C_3.

$P_3\ P_2\ P_1\ P_0\ T_3$: 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]

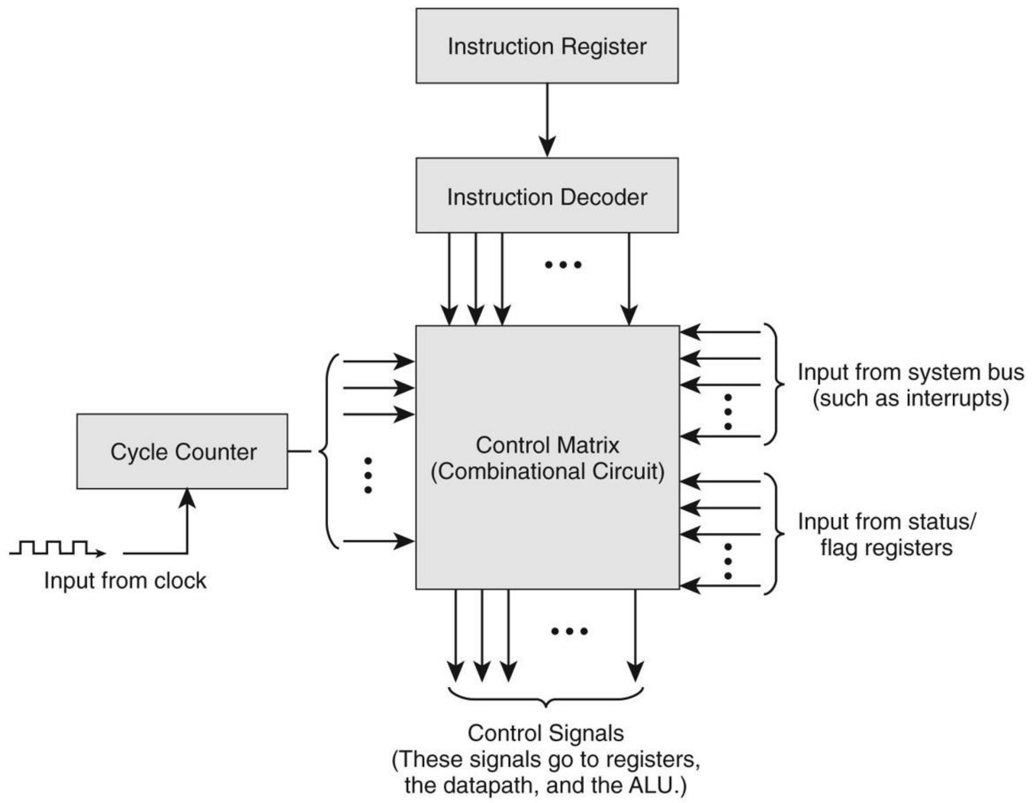


Implementing Decoder and Control Signals: Hardwired

- 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.

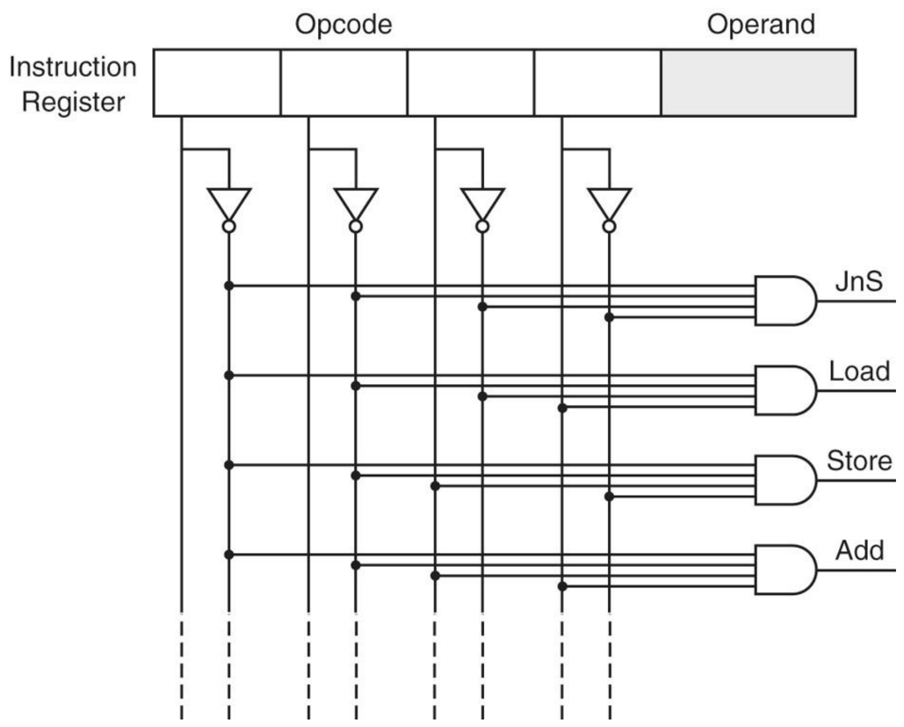
A block diagram of a hardwired control unit is shown on the following slide.

MARIE Decoding: Hardwired


Instruction Register
Instruction Decoder
Control Matrix (Combinational Circuit)
Cycle Counter
Input from clock
Input from system bus (such as interrupts)
Input from status/ flag registers
Control Signals (These signals go to registers, the datapath, and the ALU.)

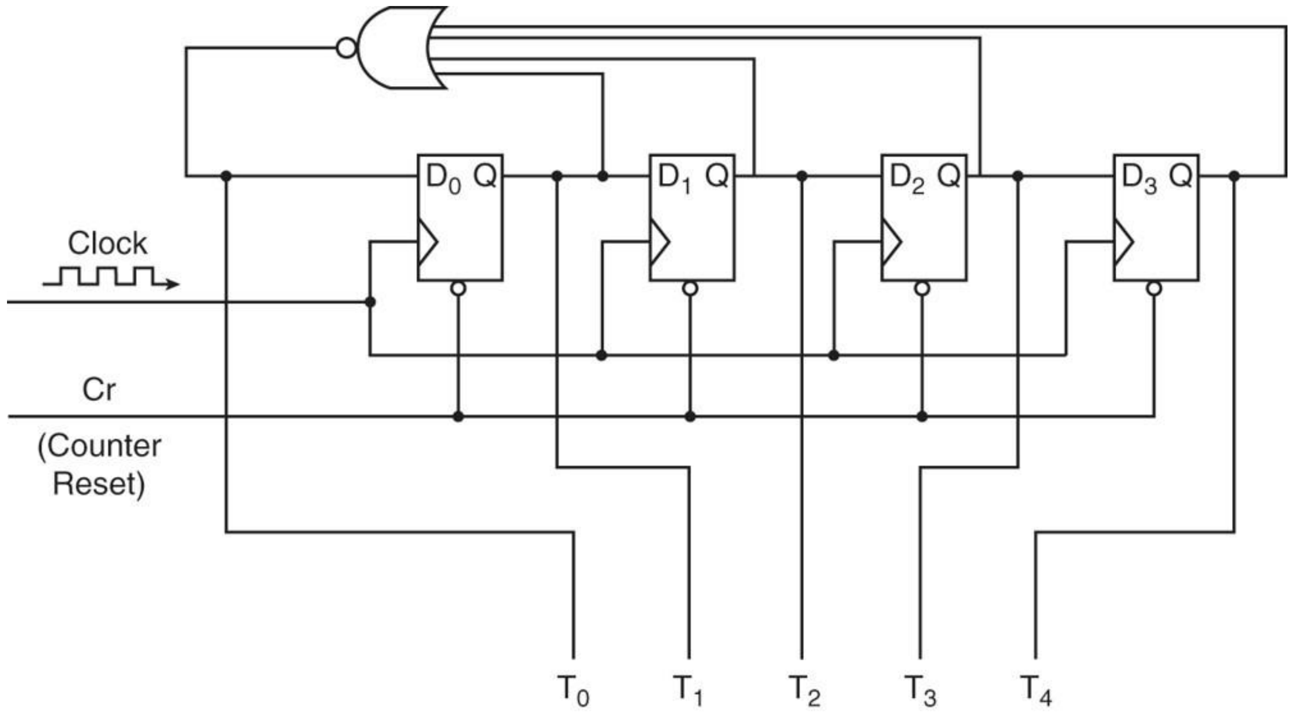
MARIE’s Instruction Decoder

- MARIE's instruction decoder. (Partial.)



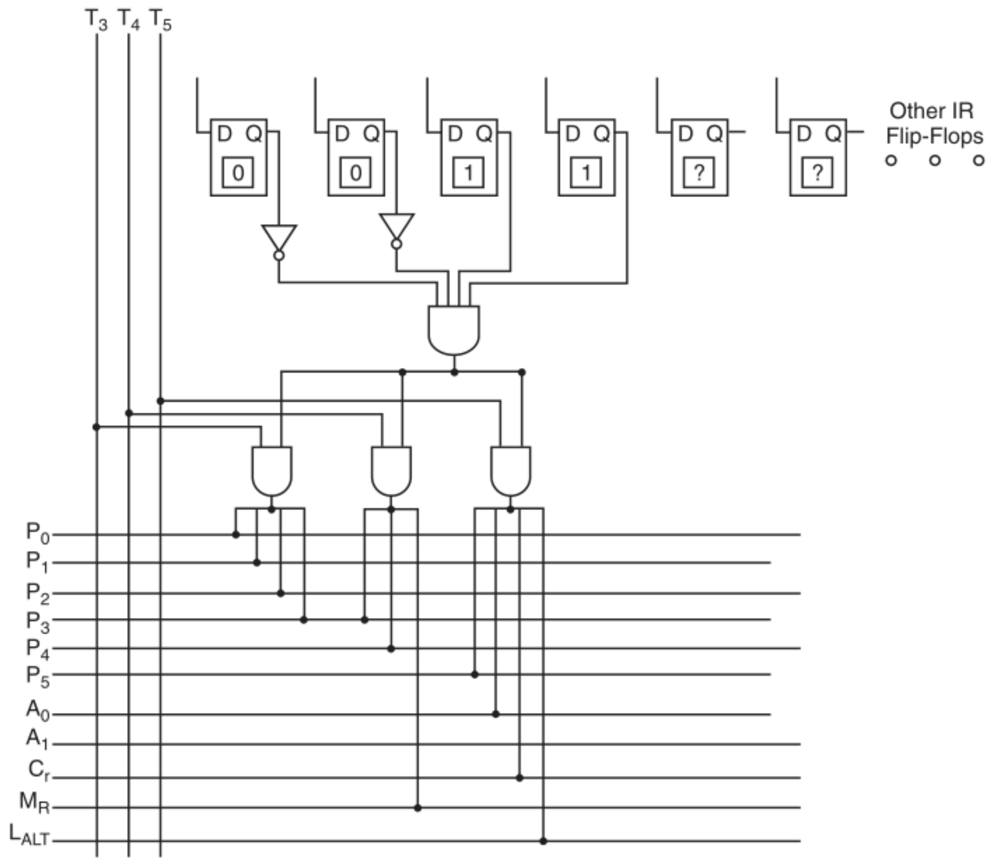
Ring Counter

- A ring counter that counts from 0 to 5



MARIE’s ADD Instruction

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

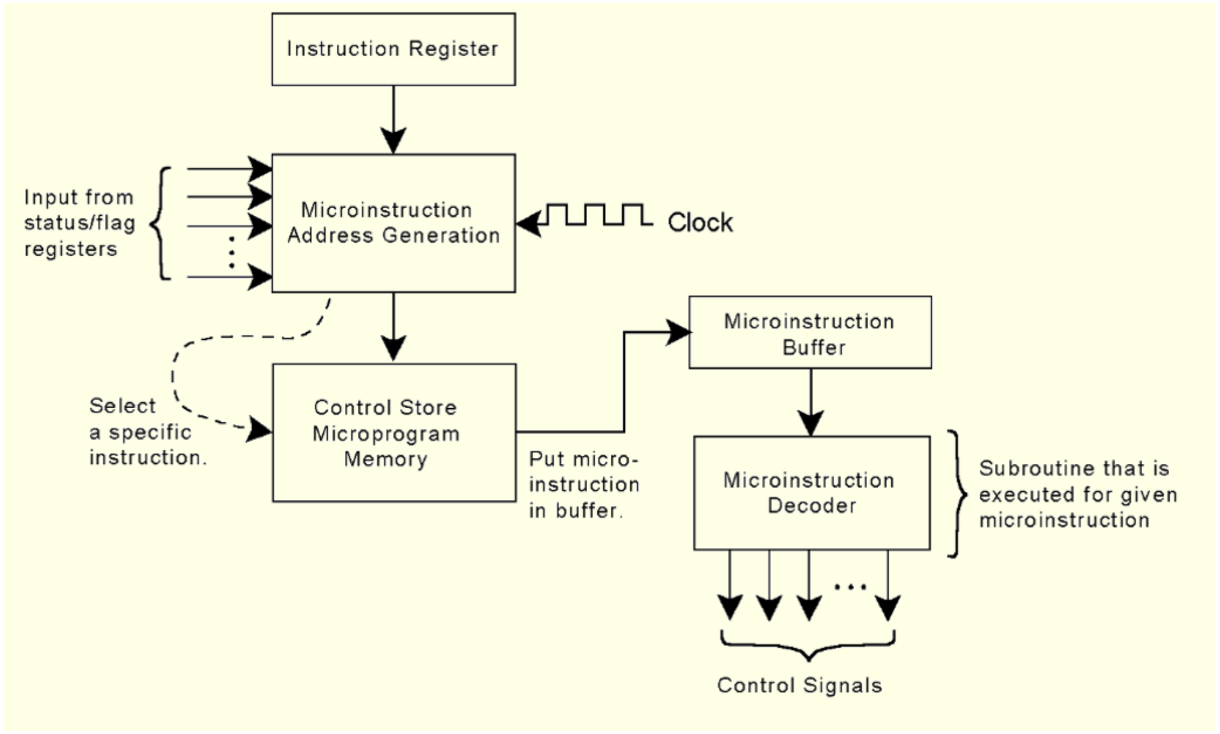


Implementing Decoder and Control Signals: Microprogrammed Control

- 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.

Generic Microprogramming Control Unit

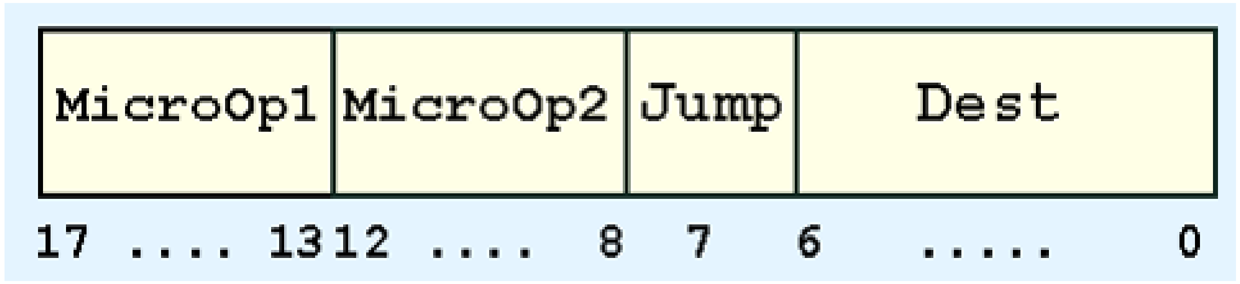
- This is how a generic microprogrammed control unit might look.



Instruction Register
Input from status/flag registers
Microinstruction Address Generation
Clock
Select a specific instruction.
Control Store Microprogram Memory
Put micro-instruction in buffer.
Microinstruction Buffer
Microinstruction Decoder
Subroutine that is executed for given microinstruction
Control Signals

Microprogrammed Control: Example

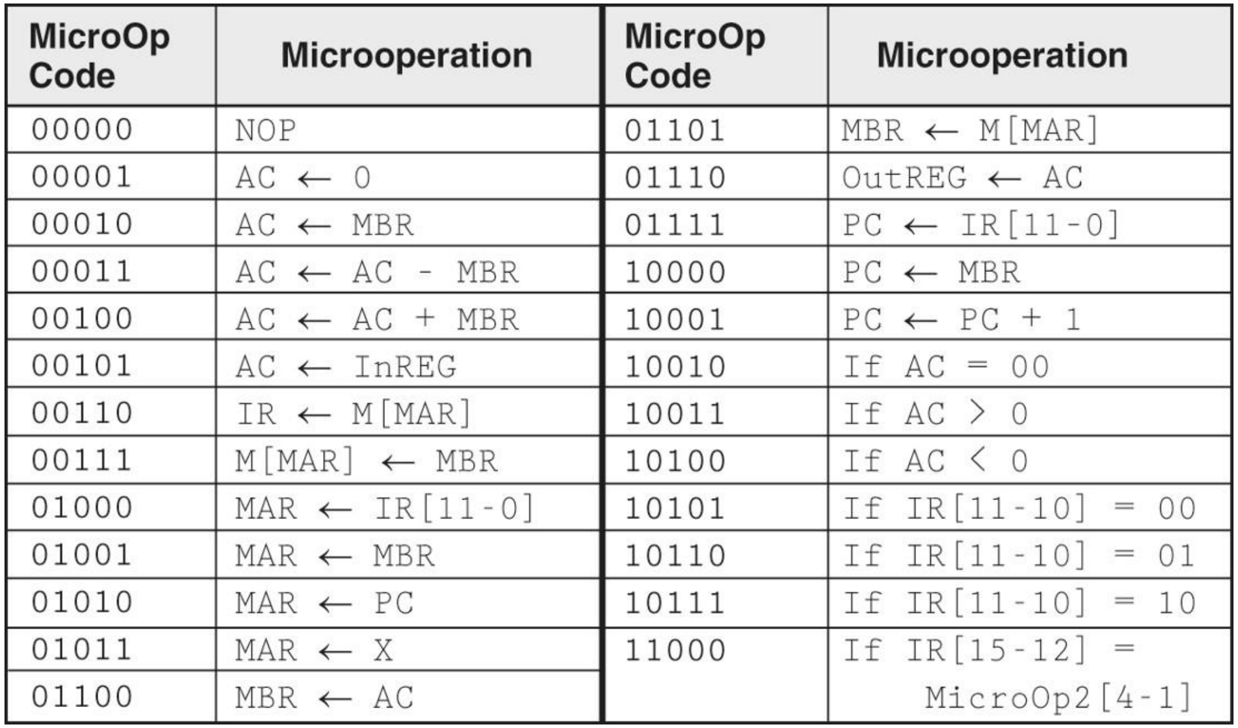
- If MARIE were microprogrammed, the microinstruction format might look like this:



- **`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.

MARIE’s Microoperation Codes

- 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		

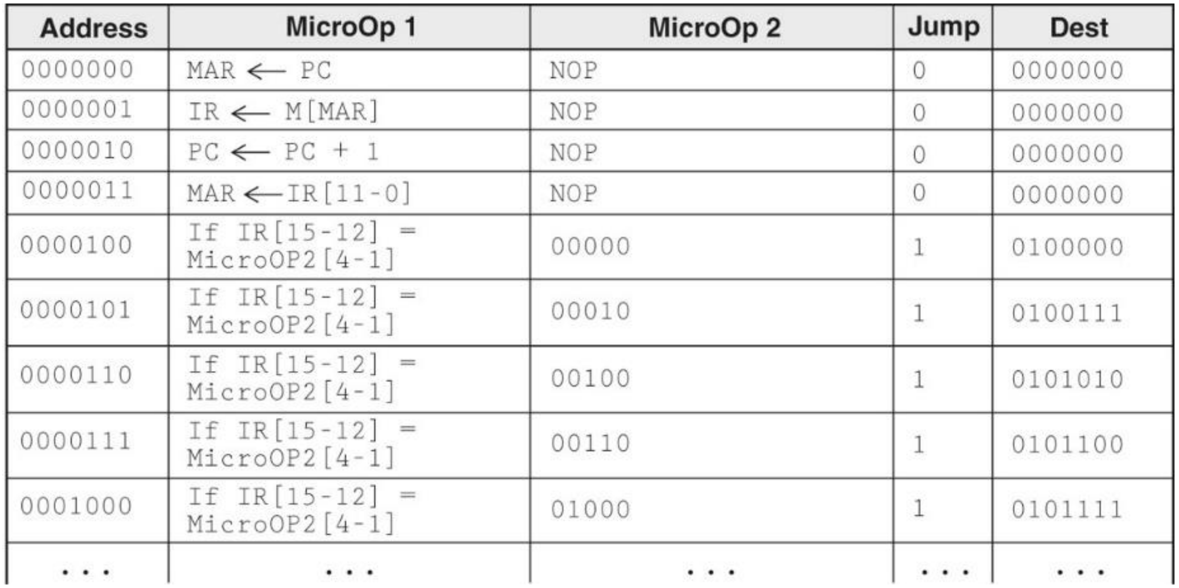
MARIE's Microprogram

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

Address	MicroOp 1	MicroOp 2	Jump	Dest
0000000	MAR $\leftarrow$ PC	NOP	0	0000000
0000001	IR $\leftarrow$ M[MAR]	NOP	0	0000000
0000010	PC $\leftarrow$ PC + 1	NOP	0	0000000
0000011	MAR $\leftarrow$ 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
. . .	. . .	. . .	. . .	. . .

MARIE's Microprogram

- 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
. . .	. . .	. . .	. . .	. . .

Discussion: Microprogrammed Control

- 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).

Real-World Architectures

- MARIE shares many features with modern architectures but it is not an accurate depiction of them.
- 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 later if time permits.

Intel Architecture (1)

- 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?

Intel Architecture (2)

- 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.

Intel Architecture (3)

- 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.

Intel Architecture (4)

- 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.

MIPS Architecture (1)

- 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.

MIPS Architecture (2)

- 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?

Summary

- Control units can be microprogrammed or hardwired.
  - Hardwired control units give better performance, while microprogrammed units are more adaptable to changes.
- 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.