2.2 THE MARIE Instruction Set Architecture

MARIE has a very simple, yet powerful, instruction set. The *instruction set architecture* (***ISA***) of a machine specifies the instructions that the computer can perform and the format for each instruction. The ISA is essentially an interface between the software and the hardware.

<u>We mentioned previously that each instruction for MARIE consists of 16 bits:</u>

- ➢ The most significant 4 bits, bits 12–15, make up the *opcode* that specifies the instruction to be executed (which allows for a total of 16 instructions).
- ➢ The least significant 12 bits, bits 0–11, form an address, which allows for a maximum memory size of 2^{12}-1. The instruction format for MARIE is shown in Figure 2.2.

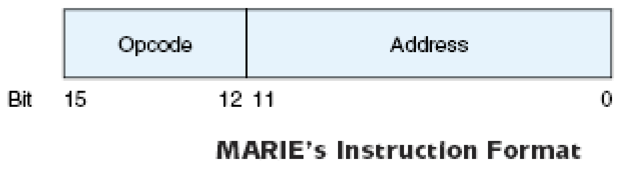


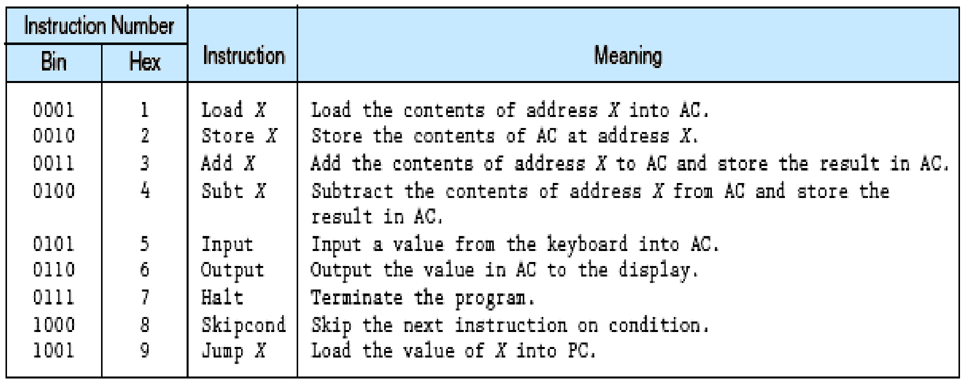
Figure 2.2: MARIE's Instruction Format

<u>Most ISAs consist of instructions for</u>:

- ✓ Processing data.
- ✓ Moving data.
- ✓ Controlling the execution sequence of the program.

MARIE's instruction set consists of the instructions shown in Table 2.2.

Table 2.1: MARIE's Instruction Set



Instruction Number		Instruction	Meaning
Bin	Hex		
0001	1	Load *X*	Load the 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 and store the result in AC.
0100	4	Subt *X*	Subtract the contents of address *X* from AC and store the result in 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 the program.
1000	8	Skipcond	Skip the next instruction on condition.
1001	9	Jump *X*	Load the value of *X* into PC.

2.2.1 Instruction set illustration

2.2.1.1The Load instruction

- ✓ Allows us to move data from memory into the CPU (via the MBR and the AC).
- ✓ All data (which includes anything that is *not* an instruction) from memory must move first into the MBR and then into either the AC or the ALU; there are no other options in this architecture.

<u>Notice that</u>

- The **Load** instruction does not have to name the AC as the final destination; this register is *implicit* in the instruction. Other instructions reference the AC register in a similar fashion.
- A transfer from one register to another always involves a transfer onto the bus from the source register, and then a transfer off the bus into the destination register. However, for the sake of clarity, we do not include these bus transfers, assuming that you understand that the bus must be used for data transfer.

2.2.1.2 The Store instruction

→Allows us to move data from the CPU back to memory.

2.2.1.3 The Add and Subt instructions

● Add and subtract, respectively, the data value found at address *X* to or from the value in the AC. The data located at address *X* is copied into the MBR where it is held until the arithmetic operation is executed.

2.2.1.4 Input and Output

- Allow MARIE to communicate with the outside world.
- Input and output are complicated operations. In modern computers, input and output are done using ASCII bytes. This means that if you type in the number 32 on the keyboard as input, it is actually read in as the ASCII character "3" followed by "2." These two characters must be converted to the numeric value 32 before they are stored in the AC.

●**<u>We are glossing over a very important concept</u>:**
How does the computer know whether an input/output value is to be treated as numeric or ASCII, if everything that is input or output is actually ASCII? The answer is that the computer knows through the context of how the value is used.

In MARIE, we assume numeric input and output only. We also allow values to be input as decimal and assume there is a "magic conversion" to the actual binary values that are stored.

2.2.1.5 The Halt command

● causes the current program execution to terminate.

2.2.1.6 The Skipcond instruction

- Allows us to perform conditional branching (as is done with "while" loops or "if" statements).
- When the **Skipcond** instruction is executed, the value stored in the AC must be inspected. Two of the address bits (let's assume we always use the two address bits closest to the opcode field, bits 10 and 11) specify the condition to be tested.

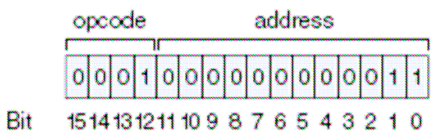
Table 2.3

Bit 11	Bit 10	condition
0	**0**	Skip if the AC is negative
0	**1**	Skip if the AC is equal to 0
1	**0**	Skip if the AC is greater than 0
1	**1**	Nothing

- By "skip" we simply mean jump over the next instruction. This is accomplished by incrementing the PC by 1, essentially ignoring the following instruction, which is never fetched. The **Jump** instruction, an unconditional branch, also affects the PC.
- This instruction causes the contents of the PC to be replaced with the value of *X*, which is the address of the next instruction to fetch.
- We wish to keep the architecture and the instruction set as simple as possible. Once you gain familiarity with how the machine works, we will extend the instruction set to make programming easier.

<u>Let's examine the instruction format used in MARIE</u>.

- Suppose we have the following 16-bit instruction:

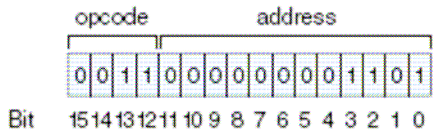


Load address 3

- The leftmost 4 bits indicate the opcode, or the instruction to be executed.
- 0001 is binary for 1, which represents the **Load** instruction.
- The remaining 12 bits indicate the address of the value we are loading, which is address 3 in main memory.
- This instruction causes the data value found in main memory, address 3, to be copied into the AC.

Consider another instruction:

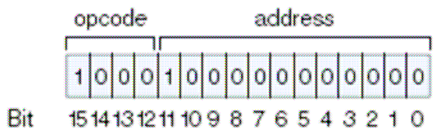
ADD address 13



- The leftmost four bits, 0011, are equal to 3, which is the **Add** instruction.
- The address bits indicate address 00D in hex (or 13 decimal). We go to main memory, get the data value at address 00D, and add this value to the AC.
- The value in the AC would then change to reflect this sum.

One more example follows:

SKIP if AC >=o



- The opcode for this instruction represents the **Skipcond** instruction.
- Bits ten and eleven (read left to right, or bit eleven followed by bit ten) are 10, indicating a value of 2. This implies a "skip if AC greater than or equal to 0." If the value in the AC is less than zero, this instruction is ignored and we simply go on to the next instruction. If the value in the AC is greater than or equal to zero, this instruction causes the PC to be incremented by 1, thus causing the instruction immediately following this instruction in the program to be ignored (keep this in mind as you read the following section on the instruction cycle).

We will be writing programs using this limited instruction set. Would you rather write a program using the commands **Load**, **Add**, and **Halt**, or their binary equivalents 0001, 0011, and 0111? Most people would rather use the

instruction name, or *mnemonic*, for the instruction, instead of the binary value for the instruction. Our binary instructions are called *machine instructions*. The corresponding mnemonic instructions are what we refer to as *assembly language instructions*. There is a one-to-one correspondence between assembly language and machine instructions. When we type in an assembly language program (i.e., using the instructions listed in Table 2.2),
we need an assembler to convert it to its binary equivalent. We discuss assemblers in next sections.