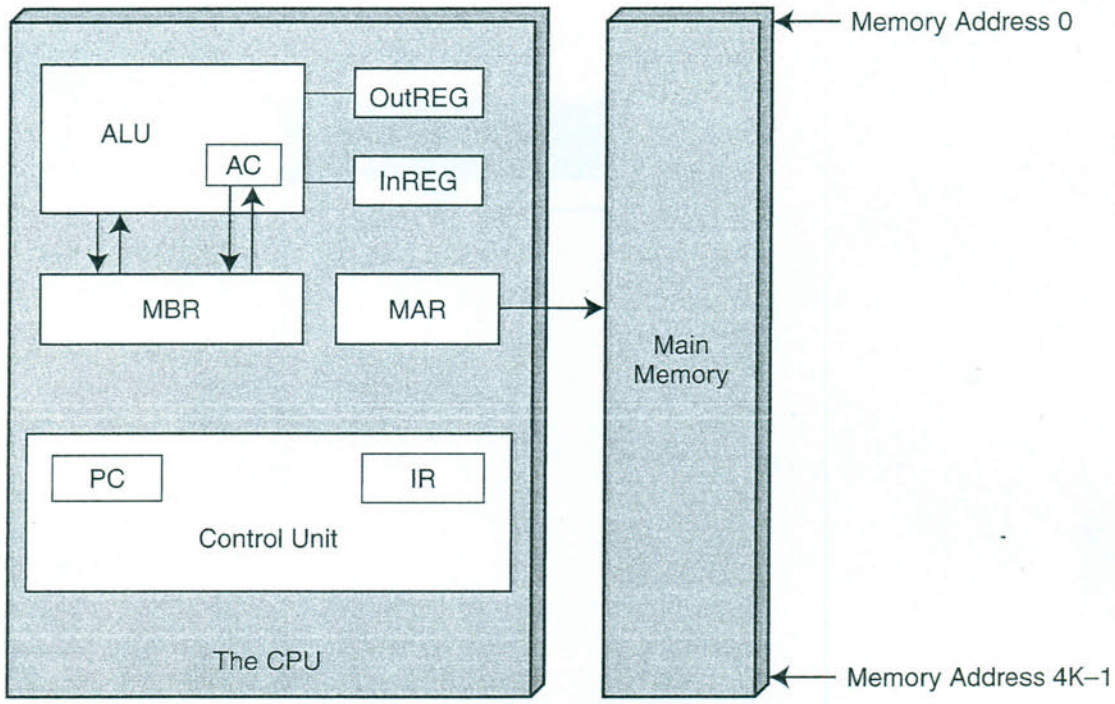


FIGURE 4.8 MARIE's Architecture

are referenced implicitly in an instruction, as we see when we describe the instruction set for MARIE in Section 4.8.3.

In MARIE, there are seven registers, as follows:

- **AC:** The **accumulator**, which holds data values. This is a **general-purpose register** and holds data that the CPU needs to process. Most computers today have multiple general-purpose registers.
- **MAR:** The **memory address register**, which holds the memory address of the data being referenced.
- **MBR:** The **memory buffer register**, which holds either the data just read from memory or the data ready to be written to memory.
- **PC:** The **program counter**, which holds the address of the next instruction to be executed in the program.
- **IR:** The **instruction register**, which holds the next instruction to be executed.
- **InREG:** The **input register**, which holds data from the input device.
- **OutREG:** The **output register**, which holds data for the output device.

The MAR, MBR, PC, and IR hold very specific information and cannot be used for anything other than their stated purposes. For example, we could not store an arbitrary data value from memory in the PC. We must use the MBR or the AC to store this arbitrary value. In addition, there is a **status** or **flag register** that holds information indicating various conditions, such as an overflow in the ALU. However, for clarity, we do not include that register explicitly in any figures.

MARIE is a very simple computer with a limited register set. Modern CPUs have multiple general-purpose registers, often called **user-visible registers**, that perform functions similar to those of the AC. Today's computers also have additional registers; for example, some computers have registers that shift data values and other registers that, if taken as a set, can be treated as a list of values.

MARIE cannot transfer data or instructions into or out of registers without a bus. In MARIE, we assume a common bus scheme. Each device connected to the bus has a number, and before the device can use the bus, it must be set to that identifying number. We also have some pathways to speed up execution. We have a communication path between the MAR and memory (the MAR provides the inputs to the address lines for memory so the CPU knows where in memory to read or write), and a separate path from the MBR to the AC. There is also a special path from the MBR to the ALU to allow the data in the MBR to be used in arithmetic operations. Information can also flow from the AC through the ALU and back into the AC without being put on the common bus. The advantage gained using these additional pathways is that information can be put on the common bus in the same clock cycle in which data are put on these other pathways, allowing these events to take place in parallel. Figure 4.9 shows the datapath (the path that information follows) in MARIE.

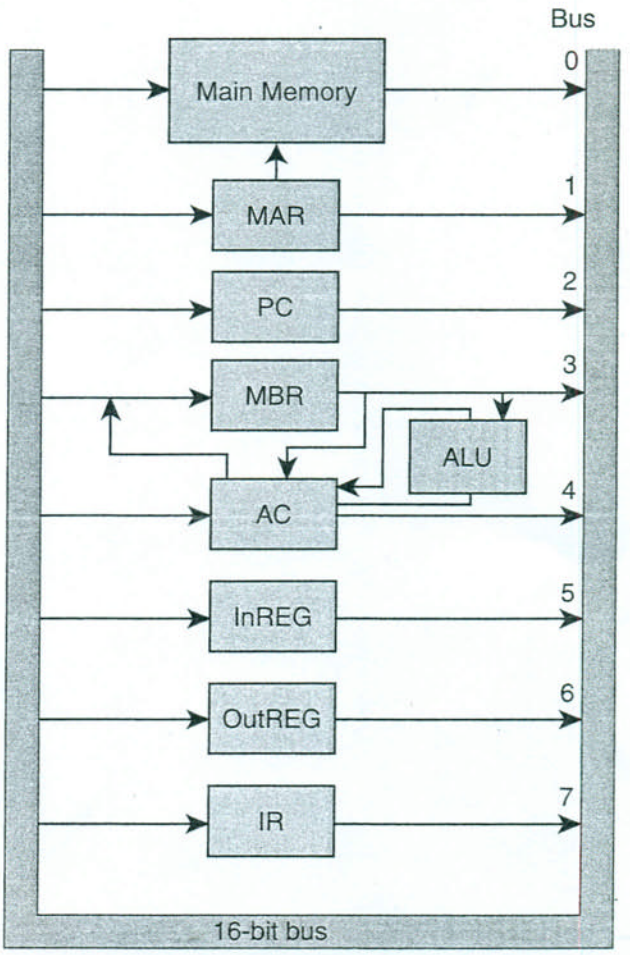


FIGURE 4.9 Datapath in MARIE

4.8.3 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. Some ISAs include hundreds of instructions. We mentioned previously that each instruction for MARIE consists of 16 bits. The most significant 4 bits, bits 12 through 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 through 11, form an address, which allows for a maximum memory size of $2^{12}-1$. The instruction format for MARIE is shown in Figure 4.10.

Most ISAs consist of instructions for processing data, moving data, and controlling the execution sequence of the program. MARIE's instruction set consists of the instructions shown in Table 4.2.

The `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. Notice that 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. The `Store` instruction allows us to move data from the CPU back to memory. 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. `Input` and `Output` allow MARIE to communicate with the outside world.

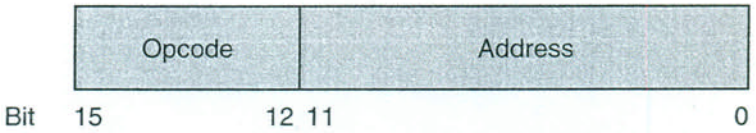


FIGURE 4.10 MARIE's Instruction Format

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.`

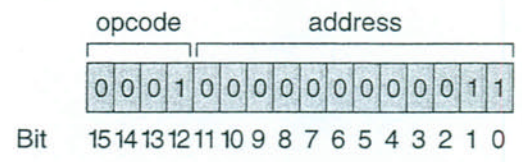
TABLE 4.2 MARIE's Instruction Set

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. Because we are focusing on how a computer works, we are going to assume that a value input from the keyboard is "automatically" converted correctly. We are glossing over a very important concept: How does the computer know whether an I/O 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. In reality, these are issues that must be addressed if a computer is to work properly.

The `Halt` command causes the current program execution to terminate. 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. If the two address bits are 00, this translates to "skip if the AC is negative." If the two address bits are 01 (bit eleven is 0 and bit ten is 1), this translates to "skip if the AC is equal to 0." Finally, if the two address bits are 10 (or 2), this translates to "skip if the AC is greater than 0." 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.

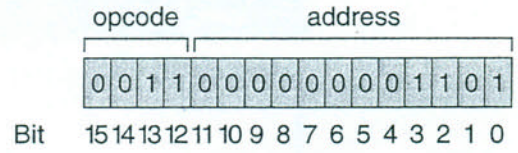
We wish to keep the architecture and the instruction set as simple as possible and yet convey the information necessary to understand how a computer works. Therefore, we have omitted several useful instructions. However, you will see shortly that this instruction set is still quite powerful. Once you gain familiarity with how the machine works, we will extend the instruction set to make programming easier.

Let's examine the instruction format used in MARIE. Suppose we have the following 16-bit instruction:

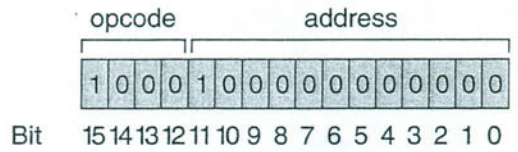


The leftmost four bits indicate the opcode, or the instruction to be executed. 0001 is binary for 1, which represents the `Load` instruction. The remaining 12 bits

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:



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:



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 0." If the value in the AC is less than or equal to zero, this instruction is ignored and we simply go on to the next instruction. If the value in the AC is greater than 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).

These examples bring up an interesting point. 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 4.2), we need an assembler to convert it to its binary equivalent. We discuss assemblers in Section 4.11.

4.8.4 Register Transfer Notation

We have seen that digital systems consist of many components, including arithmetic logic units, registers, memory, decoders, and control units. These units are interconnected by buses to allow information to flow through the system. The instruction set presented for MARIE in the preceding sections constitutes a set of

machine-level instructions used by these components to execute a program. Each instruction appears to be very simplistic; however, if you examine what actually happens at the component level, each instruction involves multiple operations. For example, the `Load` instruction loads the contents of the given memory location into the AC register. But, if we observe what is happening at the component level, we see that multiple "mini-instructions" are being executed. First, the address from the instruction must be loaded into the MAR. Then the data in memory at this location must be loaded into the MBR. Then the MBR must be loaded into the AC. These mini-instructions are called **microoperations** and specify the elementary operations that can be performed on data stored in registers.

The symbolic notation used to describe the behavior of microoperations is called **register transfer notation** (**RTN**) or **register transfer language** (**RTL**). We use the notation $M[X]$ to indicate the actual data stored at location X in memory, and $\leftarrow$ to indicate a transfer of information. In reality, 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.

We now present the register transfer notation for each of the instructions in the ISA for MARIE.

Load *X*

Recall that this instruction loads the contents of memory location X into the AC. However, the address X must first be placed into the MAR. Then the data at location M[MAR] (or address X) is moved into the MBR. Finally, this data is placed in the AC.

MAR $\leftarrow$ *X*

MBR $\leftarrow$ M[MAR]

AC $\leftarrow$ MBR

Because the IR must use the bus to copy the value of X into the MAR, before the data at location X can be placed into the MBR, this operation requires two bus cycles. Therefore, these two operations are on separate lines to indicate they cannot occur during the same cycle. However, because we have a special connection between the MBR and the AC, the transfer of the data from the MBR to the AC can occur immediately after the data is put into the MBR, without waiting for the bus.

Store *X*

This instruction stores the contents of the AC in memory location X:

MAR $\leftarrow$ *X*, MBR $\leftarrow$ AC

M[MAR] $\leftarrow$ MBR

Add *X*

The data value stored at address *X* is added to the AC. This can be accomplished as follows:

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

Subt *X*

Similar to `Add`, this instruction subtracts the value stored at address *X* from the accumulator and places the result back in the AC:

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

Input

Any input from the input device is first routed into the InREG. Then the data is transferred into the AC.

```
AC ← InREG
```

Output

This instruction causes the contents of the AC to be placed into the OutREG, where it is eventually sent to the output device.

```
OutREG ← AC
```

Halt

No operations are performed on registers; the machine simply ceases execution of the program.

Skipcond

Recall that this instruction uses the bits in positions 10 and 11 in the address field to determine what comparison to perform on the AC. Depending on this bit combination, the AC is checked to see whether it is negative, equal to 0, or greater than 0. If the given condition is true, then the next instruction is skipped. This is performed by incrementing the PC register by 1.