A Sample Machine Architecture and Machine Language

Machine Architecture

The machine has 16 general-purpose registers numbered 0 through F (hexadecimal). Each register is one byte (8 bits) long. For identifying registers within instructions, each register is assigned the unique four-bit pattern that represents its register number. This, register 0 is identified by 0000 (hexadecimal 0), and register 4 is identified by 0100 (hexadecimal 4).

Main memory consists of 256 cells. Each cell contains eight bits of data. Since there are 256 cells in memory, each cell is assigned a unique address consisting of an integer in the range of 0-255. An address can therefore be represented by a pattern of eight bits ranging from 00000000 to 11111111 (or a hexadecimal value in the range of 00 to FF).

Floating point values are assumed to be stored in the format shown as follows:

S E E E M M M M

The first bit is a sign bit. The next three bits store the exponent, with a bias of 4. The remaining four bits store the mantissa, with a hidden 1 bit.

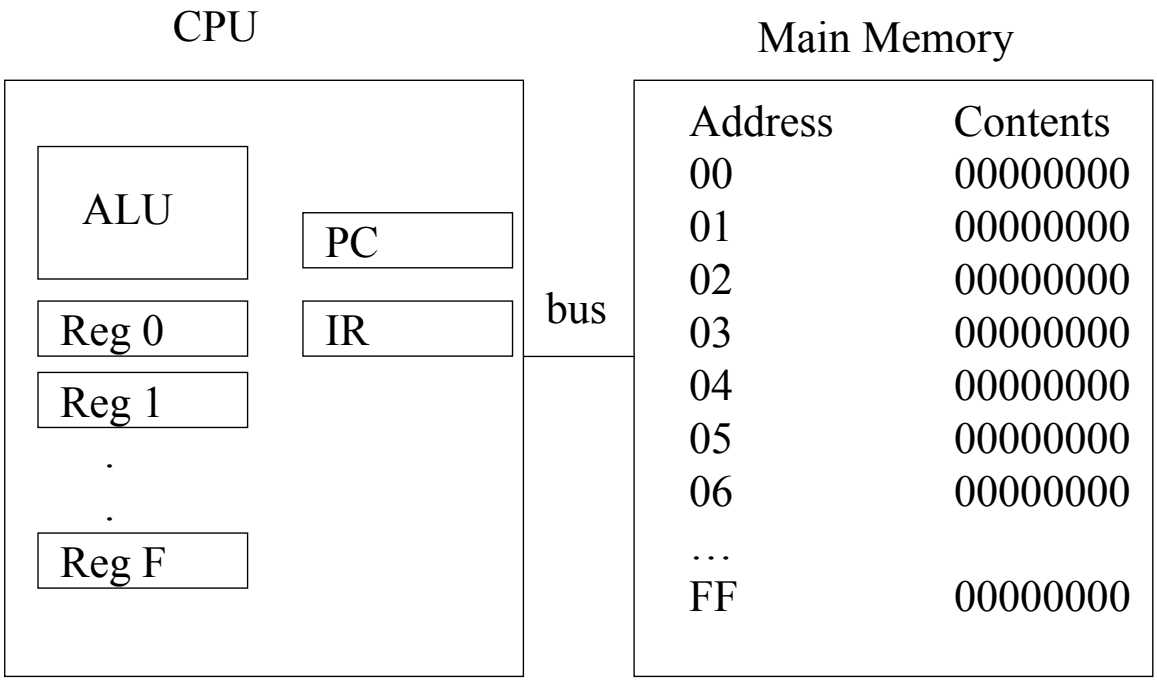
Machine Language

Each machine instruction is two bytes long. The first four bits consist of the opcode, the last 12 bits make up the operand field. The following table lists the instructions in hexadecimal notation together with a short description of each. The letters R, S, and T are used in place of hexadecimal digits in those fields representing a register identifier that varies depending on the particular application of the instruction. The letters X and Y are used in lieu of hexadecimal digits in variable fields not representing a register.

OpCode	Operand	Description
1	RXY	LOAD the register R with the bit pattern found in memory cell whose address is XY. E.g., 14A3 would cause the contents of memory cell at address A3 to be placed in register 4.
2	RXY	IMMEDIATE LOAD the register R with the bit pattern XY. E.g., 20A3 would cause the value A3 to be placed in register 0.
3	RXY	STORE the bit pattern found in register R in the memory cell whose address is XY. E.g., 35B1 would cause the contents of register 5 to be placed in the memory cell whose address is B1.
4	0RS	MOVE the bit pattern found in register R to register S. Note that not all bits in the operand are used. E.g., 40A4 would cause the contents of register A to be copied to register 4.

5	RST	ADD the bit patterns in register S and T as though they were two's complement numbers and store the result in register R. E.g. 5726 would cause the values in registers 2 and 6 to be added and placed in register 7.
6	RST	ADD the bit patterns in register S and T as though they were floating point numbers and store the result in register R. E.g. 6726 would cause the values in registers 2 and 6 to be added as floating point numbers and placed in register 7.
7	RST	LOGICALLY OR the bit patterns in registers S and T and place the result in register R. E.g., 7CB4 would cause the result of ORing the contents of registers B and 4 to be placed in register C.
8	RST	LOGICALLY AND the bit patterns in registers S and T and place the result in register R. E.g., 8045 would cause the result of ANDing the contents of registers 4 and 5 to be placed in register 0.
9	RST	EXCLUSIVE OR the bit patterns in registers S and T and place the result in register R. E.g., 95F3 would cause the result of EXCLUSIVE Oring the contents of registers F and 3 to be placed in register 5.
A	R0X	ROTATE the bit pattern in register R one bit to the right X times. Each time place the bit that started at the low-order end to the high-order end. E.g., A403 would cause the contents of register 4 to be rotated 3 bits to the right in a circular fashion.
B	RXY	JUMP to the instruction located in the memory cell at address XY if the bit pattern in register R is equal to the bit pattern in register 0. Otherwise, continue with the normal sequence of execution. E.g., B43C would first compare the contents of register 4 with register 0. If identical, the execution sequence would be altered so the next instruction executed is the one at location 3C in memory. Otherwise, program execution continues as normal with the next sequential instruction.
C	000	HALT instruction, stop execution. E.g. C000 stops execution.

An overview of the architecture is shown below. We are given the ALU, registers from 0-F, a program counter, and instruction register inside the CPU. Memory is addressed from 00 to FF where each location stores a byte.



A few comments are in order regarding the instruction format. Each instruction occupies a total of 16 bits. The op-code for each instruction occupy the first 4 bits. The remaining bits specify the operands.

This machine has two ADD instructions: one for adding two's complement representations and one for adding floating-point representations. This distinction results from the fact that the operands are just bits. We don't have any associated type stored with the operand to know what kind of data we're working with. Consequently, we must perform different activities with the ALU when adding floating point numbers than when adding integers.

The operand field consists of three hexadecimal digits (12 bits) and in each case clarifies the general instruction given by the opcode. For example, if the first hex digit is 1 (the opcode for loading from memory), the next hex digit of the instruction indicates which register is to be loaded and the last two hex digits indicate which memory cell is to provide the data. Thus, the instruction 1347 (hex) translates to the statement "LOAD register 3 with the contents of the memory cell at address 47."

A subtle distinction exists between our machine's two LOAD instructions. Here we see that the opcode 1 refers to the instruction that loads a register with the contents of a memory cell, while opcode 2 refers to the instruction that loads a register with a constant, particular value. In this case the operand contains the actual value we are loading.

An interesting situation occurs in the case of the JUMP instruction, opcode B. If the first register contains the same pattern as register 0, the machine jumps to the instruction at the address indicated by the last two hexadecimal digits of the operand. Otherwise, the execution continues as normal. This is called a conditional jump, since we may branch or

we may not branch. However, if the first hexadecimal digit of the operand field is 0, the instruction requests that register 0 be compared with register 0. Since a register is always equal to itself, the jump is always taken. This is called an unconditional jump, and would be coded with the first two digits B0.

Example: Adding values stored in memory

Consider the following sequence of steps (an algorithm) to add values together:

1. Get one of the values to be added from memory and place it in a register
2. Get the other value to be added from memory and place it in another register
3. Activate the addition circuitry with the registers loaded from steps 1 and 2 as inputs and another register designated to hold the result
4. Store the result in memory
5. Stop

Assuming that the values to be added are in two's complement notation at memory addresses 6C and 6D and the sum is to be placed in 6E, we can accomplish this with the following machine code:

1. 156C
2. 166D
3. 5056
4. 306E
5. C000

Don't forget that these hex codes are just ways to represent binary data. The binary bits are what is actually stored in the computer to represent this program.

Sample Exercises:

1. Describe the following machine instructions in English
    a. 368A
    b. BADE
    c. 803C
    d. 40F4
2. What is the difference between the instructions 15AB and 25AB?
3. Translate the following instructions to machine code.
    a. LOAD register 3 with hex value 56
    b. ROTATE register 5 three bits to the right
    c. JUMP to the instruction located at F3 if the contents of register 7 equal the contents of register 0
    d. AND the contents of register A with the contents of register 5 and leave the result in register 0
4. Write the machine code to sum the numbers from 1 to 50 (32 in hex). Here is some pseudocode:

```
        Set SUM and COUNTER to some registers in memory
        SUM ← 0
        COUNTER ← 1
        Load R0 ← 51 (decimal)
Label:  Set SUM ← SUM + COUNTER
        Set COUNTER ← COUNTER + 1
        If (COUNTER equals 51) goto Exit
        Goto Label
Exit:   Store SUM to someplace in memory we want the result
```

Solutions to Sample Machine Language Problems:

1)

a. 368A - Stores the bit pattern in register 6 into memory location 8A

b. BADE - If the contents of Register 0 equals the contents of Register A, then set the program counter to DE so that we start executing the instruction at memory location DE

c. 803C - Logically AND the contents of register 3 with register C and store the result in register 0

d. 40F4 - Copy the contents of register F to register 4

2) What is the difference between the instructions 15AB and 25AB?

15AB - Loads register 5 with the contents of memory location AB. For example, if memory location AB contains the number 3, then register 5 gets the value 3.

25AB - Loads register 5 with the bit pattern AB. Register 5 gets the value AB (i.e. the bit pattern 10101011)

3) Translate the following instructions to machine code.

a. LOAD register 3 with hex value 56 - 2356

b. ROTATE register 5 three bits to the right - A503

c. JUMP to the instruction located at F3 if the contents of register 7 equal the contents of register 0 - B7F3

d. AND the contents of register A with the contents of register 5 and leave the result in register 0 - 80A5

4) Write the machine code to sum the numbers from1 to 50 (32 in hex). Here is some pseudocode:

```
        Set SUM and COUNTER to some registers in memory
        SUM ← 0
        COUNTER ← 1
        Load R0 ← 51 (decimal)
Label:  Set SUM ← SUM + COUNTER
        Set COUNTER ← COUNTER + 1
        If (COUNTER equals 51) goto Exit
        Goto Label
Exit:   Store SUM to someplace in memory we want the result
```

Let's use register F to hold SUM and register E to hold COUNTER. The pseudocode can now be converted into machine code. Let's say our code starts at address 0:

Address	Machine Code	
0:	1F00	// Loads 0 into register F, or SUM ← 0
1:	1E01	// Loads 1 into register E, or COUNTER ← 1
2:	1101	// Loads 1 into register 1
3:	1033	// Load 33 (hex) into register 0, or R0 ← 51
4:	5FFE	// SUM ← SUM + COUNTER
5:	5EE1	// COUNTER ← COUNTER+1 (reg 1 equals 1)
6:	BE08	// If COUNTER equal 51 goto address 08

7:	B004	// If Reg 0 equal Reg 0 goto address 04
		// i.e. always go to address 04, because reg 4 will
		// always equal itself

8:	3FFF	// Store SUM someplace, in this case, to memory // address FF
9:	C000	// HALT