CS-160 Lab #4: Machine Language

General

Read through the "Background" material below, then download the Lab #4 question set and answer the questions. Turn in the questions using the instructions posted on the class web site.

For ALL Word processing documents, you must submit your documents in one of the following formats: MS-Word (NOT Works), RTF (most word processors can save in this format), or Open Document (used by the freely available Open Office suite). They will be returned ungraded if submitted in any other format.

Concepts

This lab helps you to gain an understanding of program execution flow (the CPU execution cycle) and what the the language of the CPU (machine language) is like.

Background

Note: Original source of the background contained below is from the "CS160 Worksheets" by Daniel Balls of the CS department at Oregon State University; updated and revised by Mitch Fry (CS, Chemeketa Community College).

Machine Languages

In this worksheet we will explore how the major components of a computer work together to perform simple tasks. We begin with the architecture of an extremely primitive computer, which is a slight modification of the computer described in the ninth edition of J. Glenn Brookshear's *Computer Science: An Overview*, Addison-Wesley (2007).

The central processing unit (CPU) is the circuitry that controls how information (in the form of bit patterns) is manipulated and stored. The CPU is made up of two parts—the logic/arithmetic unit and the control unit. The logic/arithmetic unit has the ability to perform simple mathematical calculations in addition to other operations. The control unit contains many individual cells, called registers, of which there are two types: general-purpose registers and special-purpose registers. The function of general-purpose registers is to temporarily store bit patterns that will be manipulated by the logic/arithmetic unit. There are two special-purpose registers: the instruction register and the program counter. The instruction register contains the instruction the control unit is currently carrying out. The program counter contains the address of the next instruction to be performed.

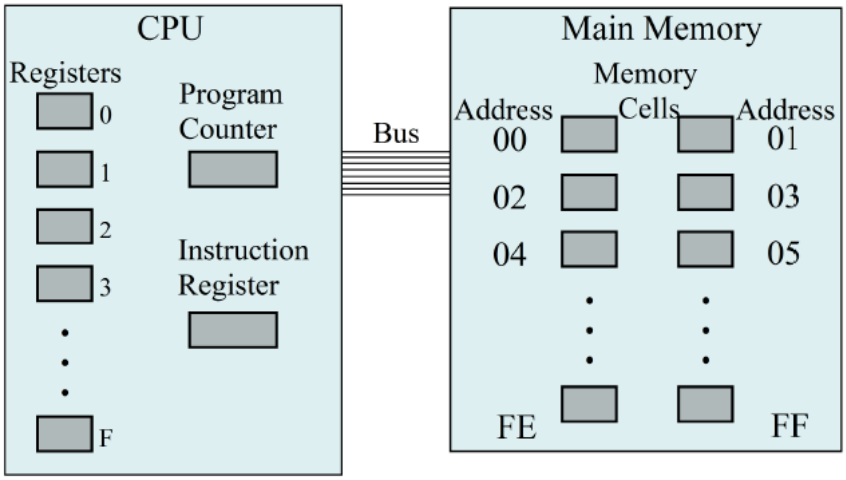
Another critical part of computer architecture is the main memory. The main memory consists of cells. Each cell has a unique address and contains 8 bits. These bits may correspond to a program instruction or to a data value. Bit patterns are transferred to and from the main memory to the CPU through a series of wires called a bus.

A Sample Machine

We will discuss how the CPU and main memory of a simple computer work together to perform basic functions. Our machine has 16 general-purpose registers, numbered in hexadecimal notation from 0 to F. Main memory consists of 256 cells—each having as its address a hexadecimal number between 00 and FF, as shown in the diagram below.

Computer programmers typically write programs in a high-level language, but in order for computer programs to be executed on a computer, they must be translated into low-level instructions called *machine language*. We will deal with the operations of our computer at the machine language level. In our simple computer, a machine instruction is a sequence of 16

bits that directs the CPU to perform a particular task. An example of a machine instruction may be 0010 1101 0011 1100. We'll use hexadecimal notation to simplify the reading and writing of these instructions. In this system the example instruction above would be written 2D3C. We'll now explain what purpose each of these four digits has in the machine language.



The first hexadecimal digit (2 in our example) in the machine instruction is called the operation code, or op-code, for short. The digit in the op-code field indicates which basic function is to be performed. These basic functions can be classified into three categories: data relocation, arithmetic/logic, and control. We will discuss selected functions from each of these categories.

An important data relocation function is LOAD. There are two types of LOAD function. One of the LOAD instructions copies a value from the main memory and puts it into a specified general-purpose register. The other kind of LOAD function —distinguished from the previous LOAD function by a different op-code (see below) — puts a constant value into a specified general-purpose register. Another data relocation function, MOVE, copies a value from one register and puts it into another register. The STORE function copies a value from a register and writes it into a specific location in the main memory.

There are two ADD functions in the computer's arithmetic/logic repertoire. One ADD function adds integers represented in two's complement notation. The other ADD function adds two floating-point numbers. There are also logical functions that the CPU can perform on bit patterns. For example, the NOT function returns the complement of the number that is given to it. Other examples of logical functions are described below.

One of the control functions the computer utilizes is the JUMP function. In the normal sequence of events, the computer executes instructions in order; the program counter advances from 0 to 2 to 4, and so on. The JUMP instruction may set the program counter to an arbitrary address.

Op-code	Operand	Description of Function	Example
1	RXY	LOAD bit pattern found in memory cell XY into register R.	***17D2*** loads bit pattern found in memory cell D2 into register 7.
2	RXY	LOAD the bit pattern XY into register with address R.	***2D3C*** loads the bit pattern 00111100 into register D.
3	RXY	STORE the bit pattern found in the register R in memory cell XY.	***3A03*** stores the bit pattern in register A into memory cell 03.
4	0RS	MOVE (copy) bit pattern from register R to register S.	***403F*** copies bit pattern in register 3 into register F.
5	RST	ADD integers in registers S and T (two's complement form) and place result in register R.	***5B64*** adds integers in registers 6 and 4; puts result in register B.
6	RST	ADD floating-point numbers in registers S and T and place result in register R.	***6AE1*** adds the floating point numbers in registers E and 1; puts result in register A.
7	RST	Combine bit pattern in register S and register T using the OR operator and place result in register R.	***7D29*** combines bit patterns found in register 2 and 9 using the OR operator and places the result in register D.
8	RST	Combine bit pattern in register S and register T using the AND operator and place result in register R.	***81F2*** combines bit patterns found in register F and 2 using the AND operator and places the result in register 1.
9	RST	Combine bit pattern in register S and register T using the EXCLUSIVE OR operator and place result in register R.	***94C3*** combines bit patterns found in register C and 3 using the XOR operator and places the result in register 4.
A	R0X	ROTATE bit pattern in register R to the right X times.	***A907*** rotates bit pattern in register 9 to the right 7 times.
B	RXY	JUMP to instruction located in memory cell XY IF the bit pattern in register R is EQUAL to bit pattern in register 0; otherwise continue to next instruction.	***BDA7*** compares bit pattern in register D with that in register 0. If they are equal, program jumps to instruction located in cell A7; if they are not, the program moves to the next instruction.
C	000	HALT the program's execution.	***C000*** causes program stop.
D	RXY	JUMP to instruction XY IF the bit pattern in register R is LESS THAN ZERO; otherwise continue to next instruction.	***D18D*** skips to instruction located in cell 8D if bit pattern in register 1 begins with 1; otherwise, the program moves to the next instruction.
E	0RS	NOT—negate each bit in pattern found in register S and put resulting bit pattern in register R.	***E03C*** finds complement of bit pattern in register C and puts the result in register 3.

One JUMP occurs when the value in specified register is equal to the value in register 0. The second JUMP occurs when the value in a specified register is less than zero. Another control instruction is the HALT command, which causes the execution of the program to stop.

For our purposes, each of these (and several other) commands has been assigned a specific hexadecimal digit, as shown in the **Op-Code** portion in the above table. Next to each op-code in the table is a description of the operand that provides details necessary for the instruction to be carried out.

In our example instruction (2D3C), the first hexadecimal digit 2 indicates that the machine is to load a particular bit pattern. The other hexadecimal digits (D3C) in the instruction are known as the operand. The letter D in this case represents the address of the register into which the bit pattern will be loaded. The 3C signifies that the bit pattern to be loaded is 00111100. Taken together, the instruction 2D3C indicates the machine will load the bit pattern 00111100 into the general-purpose register with address D.

Another example of an instruction is 87F2. The op-code 8 instructs the machine to combine two bit-patterns into one using the AND function. The hexadecimal digits F and 2, tell us to combine the bit patterns stored in those two registers and place them in register 7. For the next few paragraphs we will assume that bit pattern 01101110 is in register F and the bit pattern 11110011 is in register 2. To combine these two patterns into one using the AND operation, we first stack one bit pattern on top of the other so each of the eight digits is aligned:

```
  0 1 1 0 1 1 1 0
  1 1 1 1 0 0 1 1
```

Next we'll start at the right (though this is arbitrary) and, according to rules of the AND operation, place a one under this column of numbers if they are both one, and a zero if they aren't both one. The same procedure is applied to each of the remaining seven columns of numbers, and the result is represented below:

```
      0 1 1 0 1 1 1 0
AND   1 1 1 1 0 0 1 1
      ---------------
      0 1 1 0 0 0 1 0
```

The instruction 87F2 would combine 0110110 and 11110011 (which are the bit patterns in registers F and 2, respectively) using the AND operation and put the resulting bit pattern, 01100010, into register 7.

The combining of two bit patterns into one can also be performed by other logical operators—OR, XOR, etc. You may want to verify that the result of combining the bit patterns in registers F and 2 using the OR operation will yield the bit pattern 11111111, while the resulting bit pattern using the XOR operation is 10011101. The NOT operator requires only one bit pattern and simply negates each of the bits in the pattern. For example, the instruction E07F will place the bit pattern 10010001 in register 7, which you may also want to verify.

The ROTATE function rotates the bits in a bit pattern a certain number of positions, in a circular fashion. For example, the instruction AF04 will rotate the bit pattern in register F four positions to the right. If the original pattern is 01101110, the resulting bit pattern is 11100110, as shown below.

```
0 1 1 0 1 1 1 0  ⇒  1 1 1 0 0 1 1 0
original byte       rotated byte
```

In order to help you become more familiar with the intricacies of machine language, we will walk through a few examples. First, let's add two values that are stored in the main memory and place the sum back in the main memory. Suppose 02 and 08 are the values of the bytes stored in the main memory cells with addresses F0 and F1, respectively. Our objective is to write a machine language program that adds these two values and places their sum in the main

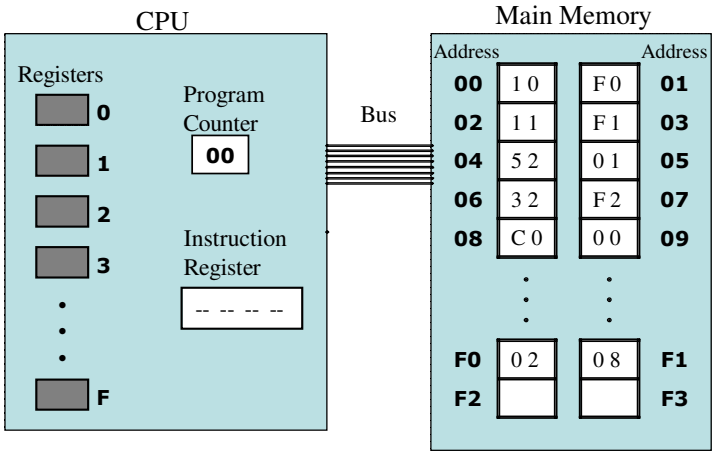
memory cell with address F2. The steps that we will need the computer to perform in order to realize this objective are outlined below:

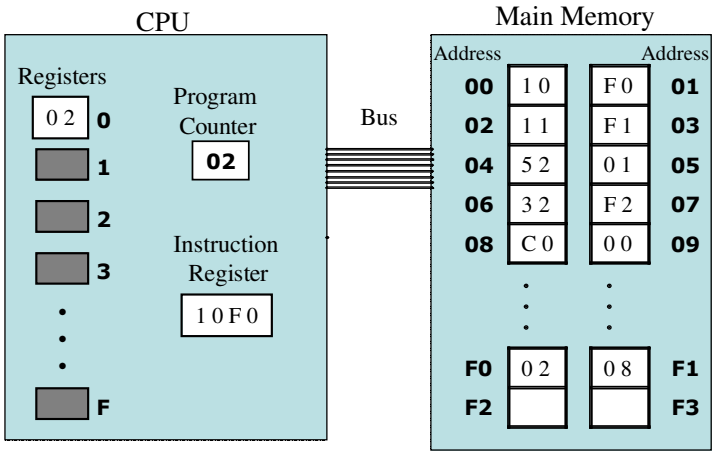
1. Load the value of the memory cell with address F0 into register 0.
2. Load the value of the memory cell with address F1 into register 1.
3. Add the values in register 0 and register 1 and place the result in register 2.
4. Store the value of register 2 into the memory cell with address F2.
5. Halt the program.

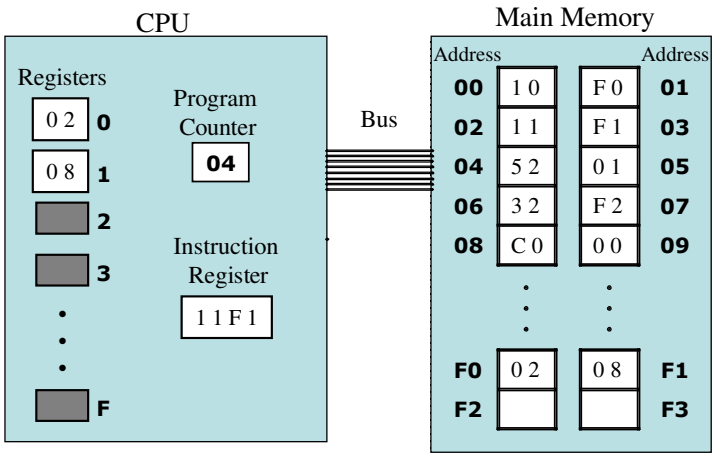
For each of the steps above, there is an associated two-byte instruction written in machine language that the CPU can use. These instructions are:

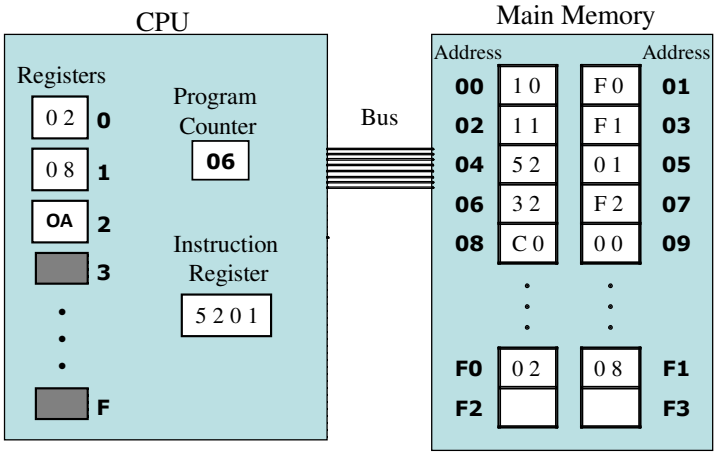
10F0
11F1
5201
32F2
C000

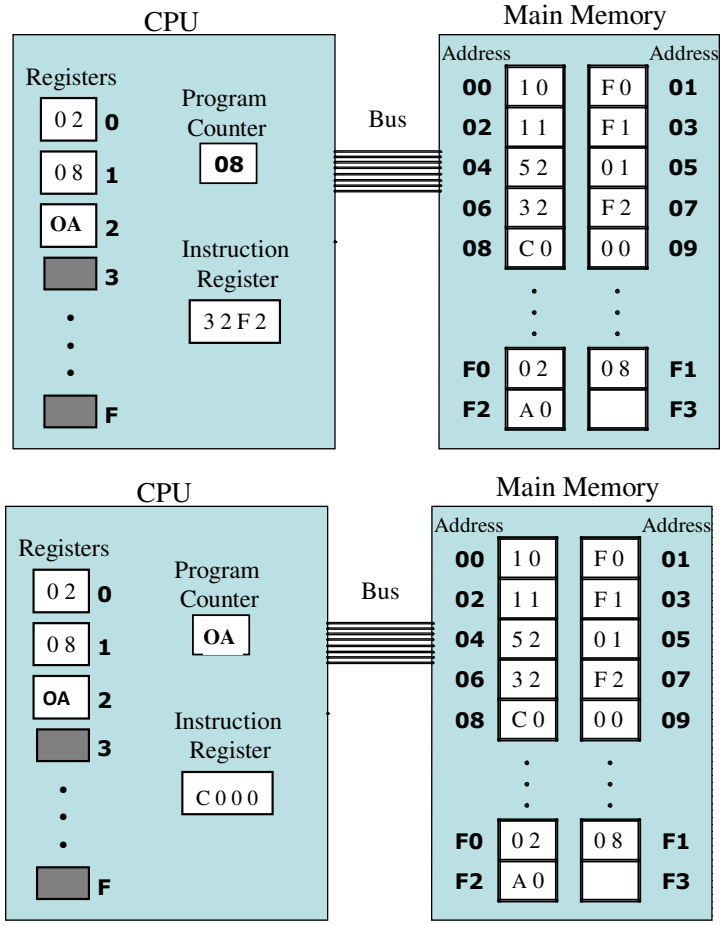
We will now represent the changes that occur in the main memory and CPU as these instructions are carried out at the machine level. Notice that the machine language instructions are stored in the main memory in cells 00 to 09. As the CPU progresses through the five instructions, watch the program counter, the instruction register, the effected registers and main memory cells change accordingly.




CPU
Registers
0 2 0
1
2
3
F
Program Counter
02
Instruction Register
1 0 F 0
Bus
Main Memory
Address
Address
00 1 0 F 0 01
02 1 1 F 1 03
04 5 2 0 1 05
06 3 2 F 2 07
08 C 0 0 0 09
F0 0 2 0 8 F1
F2 F3


CPU
Registers
0 2 0
0 8 1
2
3
F
Program Counter
04
Instruction Register
1 1 F 1
Bus
Main Memory
Address
Address
00 1 0 F 0 01
02 1 1 F 1 03
04 5 2 0 1 05
06 3 2 F 2 07
08 C 0 0 0 09
F0 0 2 0 8 F1
F2 F3


CPU
Registers
0 2 0
0 8 1
OA 2
3
F
Program Counter
06
Instruction Register
5 2 0 1
Bus
Main Memory
Address
Address
00 1 0 F 0 01
02 1 1 F 1 03
04 5 2 0 1 05
06 3 2 F 2 07
08 C 0 0 0 09
F0 0 2 0 8 F1
F2 F3



There are a couple of items that should be noted. Since each instruction is two bytes long, each line of instruction takes up two memory cells; consequently, the program counter only identifies the first address of the cell containing the line of instruction. This address is always an even number Also, due to space constraints, only those cells that are necessary for the program are shown in the graphical representations. This convention will be used in the following exercises.

In our final example we will illustrate the characteristic of one of the two JUMP functions in our arsenal. We'll begin by assuming that the value of register 0 is known to have the bit pattern 06 and the value of register 5 is 0. Also, for the time being, we'll assume the value of the memory cell with address C3 is unknown. We'd like to subtract the value of C2 from 06, the value in register 0, and place the result in the memory cell C3. If the value of C2 is also 06, the difference is clearly 0, and in this case we don't want to have to run the rest of the program. First we will create a list of instructions that will accomplish the objective. They are as follows:

1. Load the contents of memory cell C2 into register 1.
2. JUMP to the last line of instruction if the value of C2 is equal to the value in register 0; otherwise, move on to step three.
3. Load the value 00000001 into register 2.

4. Compute the logical NOT of the bit pattern in register 1 and place the result into register 3.

5. Add the integers in registers 2 and 3 as if they were in two's complement form and place the result in register 4.
6. Add the integers in registers 0 and 4 as if they were in two's complement form and place the result in register 5.
7. Store the value of register 5 in the memory cell with address C3.
8. End the program.

We'll assume that these 8 machine instructions are located in the memory cells with addresses 00 through 0F. Here is the resulting machine language program.

```
1 1 C 2
B 1 D C
2 2 0 1
E 0 3 1
5 4 2 3
5 5 0 4
3 5 C 3
C 0 0 0
```