Laboratory

7

Low-Level Languages

Objective

- Study simple machine language and assembly language programs.

References

Software needed:

1) A web browser (Internet Explorer or Netscape)

2) Applet from the CD-ROM:

   a) Super simple CPU

Textbook reference: Chapter 7, pp. 201–223

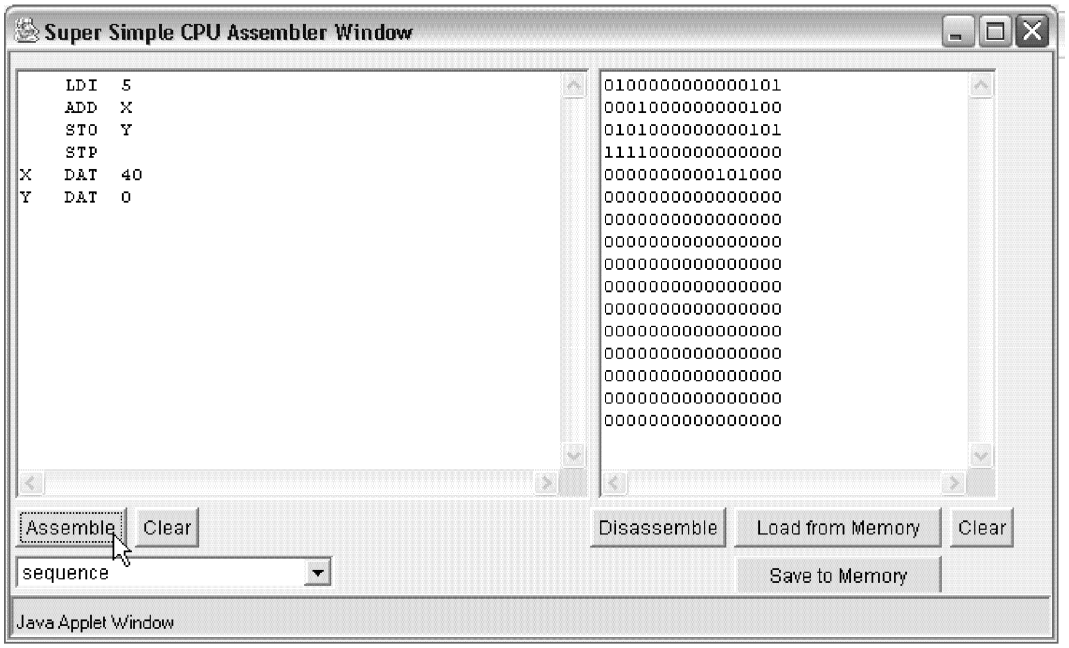
Background

Chapter 7, "Low-Level Programming Languages," discusses the basic concepts of machine and assembly programming. In this lab, we will use the same "Super simple CPU" applet that we used in Lab 5 (instead of Pep/7 that is discussed in the textbook). You should review Lab 5 for basic instructions on how to use the Super Simple CPU.

Activity

In Chapter 5, we studied the fetch-execute cycle while watching the Super simple CPU run a few programs. In this lab, we will look at a further refinement; assembler programming.

To begin, start the applet and click on the *Assembler Window* button. When the window appears, choose *Sequence* from the menu. This puts a short assembler program in the left window. Now click on *Assemble*. Here's what you should see:



```
    LDI  5
    ADD  X
    STO  Y
    STP
X   DAT  40
Y   DAT  0
```

```
0100000000000101
0001000000000100
0101000000000101
1111000000000000
0000000000101000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

The machine language equivalent of the assembler program appears in the right window. Since there are 16 words in the memory of the Super simple CPU, the assembler creates sixteen 16-bit values, one per memory word, padded out with 0s as necessary to make up 16 bits.

Assembler language (also called assembly language) is the human-readable encoding of machine language instructions. There is one assembler line per machine language instruction. The Super simple CPU is so basic and small that one instruction can fit neatly into just one memory word. As discussed in Lab 5, the first four bits are the opcode and the last 12 bits are the operand (see Lab 5 if you need a reminder).

Also remember from Lab 5 that you can see what the numerical 4-bit opcodes are and what they do from the help buttons on the main window of the applet, or from the opcode list in this manual on p. 69.

To demonstrate how machine language can be translated into assembler language, we'll start with the first machine language instruction that appears in the program we loaded into the CPU applet:

```
0100000000000101
```

(Believe it or not, many early programmers, including some still alive today, programmed computers in lines of binary machine language code just like this. Imagine how tricky it is to create, and especially to debug, a large program in machine language! That's exactly why assembler language was invented around 1952.)

Let's decode this instruction. First, assembler language replaces the four-digit opcode with its three-letter *mnemonic* (this strangely spelled word—the leading "m" is silent—comes from a Greek word meaning "to remember").

Looking up 0100 from the opcode list, we see it corresponds to LDI, the load immediate instruction. So the assembler language instruction should begin with LDI. Next, we convert the 12-digit binary operand (000000000101) into decimal, which gives us 5. So the complete assembler language equivalent of

```
0100000000000101
```

is `LDI 5`.

(If you wish, you can specify the operand as a binary number instead of a decimal number in the assembler code. However, to keep the Super simple CPU from getting confused as to whether `10` is "two" or "ten," you must put a `b` (or `B`) after a binary number. So, you could also have written `LDI 101b` and the applet would translate it into exactly the same machine instruction. Unlike the machine language instruction, you don't need to pad out the front of `101b` with 0s.)

From what we've seen so far, assembler language is merely a straightforward translation of machine language, easier to read, certainly, but of limited usefulness. The real power comes when addresses are encoded symbolically, using *identifiers*. An identifier is a descriptive word or phrase used by the programmer to aid in understanding the role of a memory address or data. That identifier is then used instead of referencing the memory address. Unlike opcodes, which are a defined set of instructions, a programmer can make up the identifiers to suit the situation—for example, you can replace numerical memory addresses with meaningful identifiers such as `SALARY` or `TOPOFLOOP` or `SALESTAX`. With identifiers, it's easier to understand the purpose of a line of code.

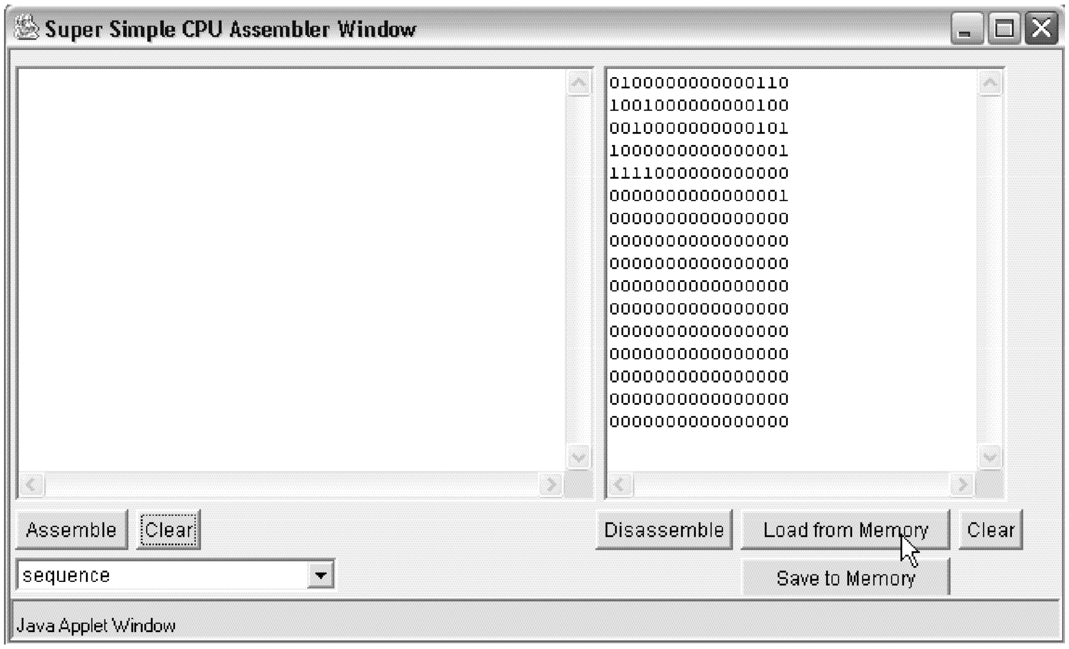
Let's look at a simple loop program, one that loads 20 into the accumulator, checks to see if it is 0, subtracts 1 if not, and continues. Close the assembler window, returning to the main window. Pull down *Example 5* from the examples menu.

Here's the program that will load into memory (the opcodes are separated from the operands here just to make it easier to read):

```
0100 000000010100
1001 000000000100
0010 000000000101
1000 000000000001
1111 000000000000
0000 000000000001
```

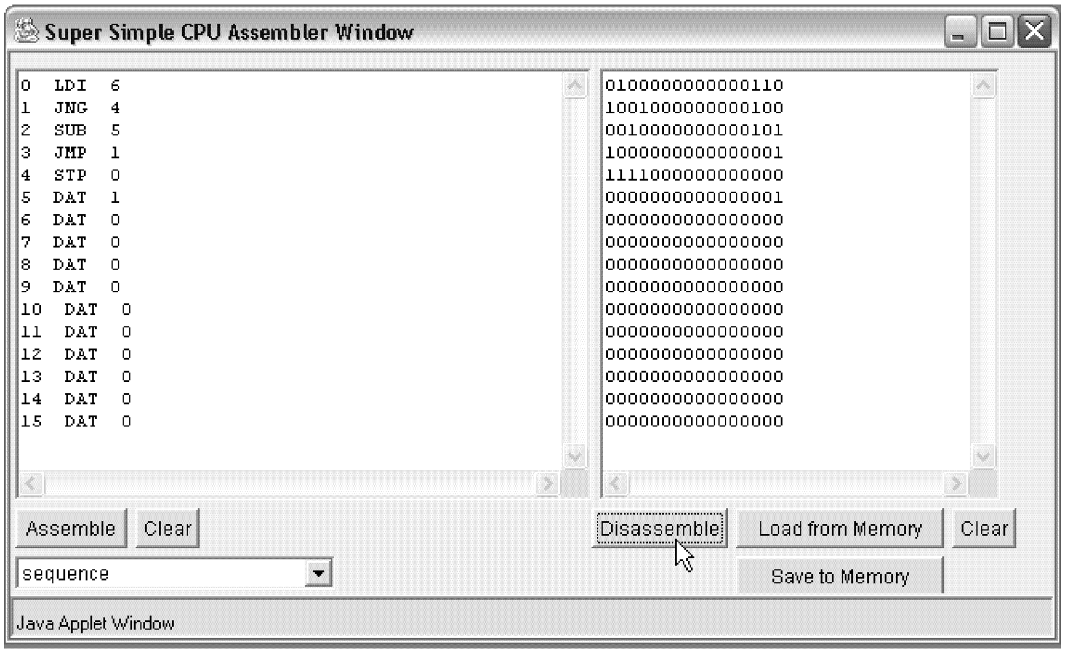
Even with the opcodes and operands separated, it's still not easy to decipher what this program does.

Click on *Assembler Window* in the main applet screen again. Once you see it, click on *Load from Memory*, as shown below:



```
0100000000000110
1001000000000100
0010000000000101
1000000000000001
1111000000000000
0000000000000001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

This copies the memory values from the Super simple CPU's memory into the machine language area of the assembler window. Now click on *Disassemble*, and let's see how smart the Super Simple CPU is. Can it reconstruct the original assembler program or not?



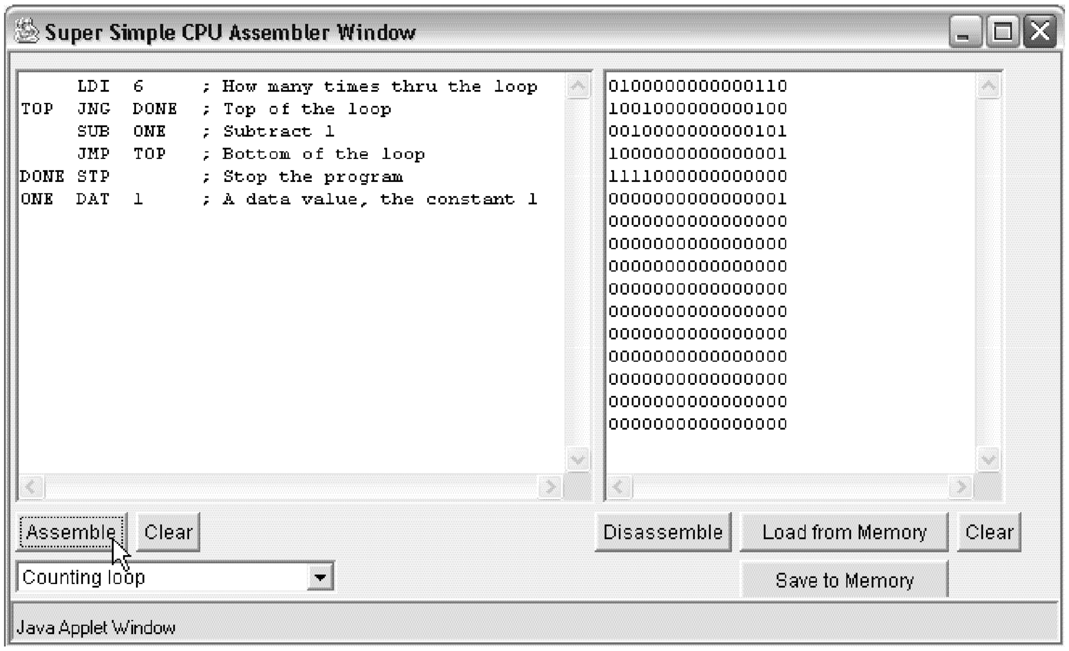
```
0  LDI  6
1  JNG  4
2  SUB  5
3  JMP  1
4  STP  0
5  DAT  1
6  DAT  0
7  DAT  0
8  DAT  0
9  DAT  0
10  DAT  0
11  DAT  0
12  DAT  0
13  DAT  0
14  DAT  0
15  DAT  0
```

```
0100000000000110
1001000000000100
0010000000000101
1000000000000001
1111000000000000
0000000000000001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

And the answer is ... well, not really. Sure, the *Disassemble* button translated the opcodes into mnemonics and converted the operands to decimal, but it left the addresses as they are. For example, the second instruction, `JNG 4`, jumps to memory word 4 if the accumulator is negative. But what is at word 4? A STP instruction, which will stop the computer. So address 4 could be better represented with an identifier that tells what its function is, like DONE or ENDOFLOOP.

Let's take a look at a version of this program in assembler that takes full advantage of the power of identifiers to create an easier-to-understand program. First, click on the two *Clear* buttons to clear your assembler windows. Then pull down the menu to *Counting Loop*.

The assembler program that appears in the left text area (shown below), is *much* more readable than the previous version—once you understand how assembler language uses identifiers.



```
     LDI  6     ; How many times thru the loop
TOP  JNG  DONE  ; Top of the loop
     SUB  ONE   ; Subtract 1
     JMP  TOP   ; Bottom of the loop
DONE STP        ; Stop the program
ONE  DAT  1     ; A data value, the constant 1
```

```
0100000000000110
1001000000000100
0010000000000101
1000000000000001
1111000000000000
0000000000000001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

Counting loop

To help you understand how assembler language uses identifiers, the chart below goes step by step through each line of the machine language version of this program, showing its corresponding assembler language version and explaining the identifiers used.

Memory Location	Machine Language Instruction	Assembly Language Instruction
0	`0100000000010100` Load `20` into the accumulator.	`LDI 20` `LDI`: The opcode. `20`: The value contained in the operand, which should be loaded into the accumulator.
1	`1001000000000100` Jump to memory location 4 if the value in the accumulator is < 0.	`TOP JNG DONE` `TOP`: Identifier given to this memory location, since it represents the top of a loop that first examines the contents of the accumulator to see if it is < 0. `JNG`: The opcode. `DONE`: The identifier given to memory location 4 (see next page).

2	`0010000000000101` Subtract the value in memory location 5 from the accumulator.	`SUB ONE` `SUB`: The opcode. `ONE`: Instead of referring to memory location 5, this instruction refers to an identifier called `ONE` (see below). Since `ONE` has a defined value of 1, 1 will be subtracted from the accumulator.
3	`1000000000000001` Jump to the instruction at memory location 1.	`JMP TOP` `JMP`: The opcode. `TOP`: Instead of referring to memory location 1, it refers to the identifier `TOP`. At this point the program jumps up to the top of the loop.
4	`1111000000000000` Stop the program.	`DONE STP` `DONE`: Identifier given to this memory location, since it contains the instruction to stop the program. `STP`: The opcode.
5	`0000000000000001` The value 1 is stored in this memory location, to be referred to by the program as needed.	`ONE DAT 1` `ONE`: Identifier for this chunk of data, since it has the value of 1. `DAT`: This is not one of the opcodes; instead, it is a *pseudo-op* that tells the assembler software that this is data. `1`: The value of the data that is to be linked to the identifier `ONE`.

Let's look at that final line of the assembler program:

```
ONE DAT 1
```

It doesn't create a machine instruction, because there is no machine opcode for `DAT`. Instead, `DAT` is what is known as a *pseudo-op*, a directive that is meaningful for the assembler software but that does not correspond to an opcode command.

Here, `ONE DAT 1` instructs the assembler to change the decimal number 1 into binary, store it into memory, and link the identifier `ONE` to the address of that word. `DAT` is called an assembler *pseudo-op* because there is no corresponding opcode called `DAT`. Rather, `DAT` is a directive for the assembler software, telling it what to do. Real assemblers have many pseudo-ops.

Click on *Assemble*, and the equivalent machine language program appears in the right text area, identical to what we saw before.

Writing assembler programs is an art as well as a craft. There are many different identifiers that can be used in place of machine addresses, but some make more sense than others. As with all programs, assembler programs should be written with *documentation* that will help future programmers decipher and modify the code, because useful programs undergo constant mutation as ever-greater demands are made on them. As the old programmer's lament goes: "If the program works, it must be changed!"

Exercise 1

Name ______________________________ Date ________________

Section ______________________________

1) Start the "Super simple CPU" applet.
2) Open the Assembler Window.
3) Load the GCD example and take a screenshot.
4) On your screenshot, draw arrows from the jumping instructions to their *target addresses*.
5) Change the second line from the bottom to read A DAT 21; A = 21 and change the bottom line to read B DAT 14; B = 14.
6) Click *Assemble*, then click *Save to Memory*. Go back to the main window.
7) Trace the program. In other words, pretend you are the computer and do one instruction at a time, writing down the values in IR and ACC at the end of each fetch-execute cycle.

   First, note the number in the PC, then click on the *1 Step* button. When that step finishes running, note the decoded IR (the mnemonic value in the box just beneath the IR box) and the value in the accumulator. Start a new line for the next step, noting the PC first, then once again clicking the *1 Step* button and noting the resulting IR and ACC values. Continue until the program is finished.

   To get you started, here are the first few lines of the trace done for you as an example. You should make sure that your results match these as you start out:

PC	IR	ACC
0	LOD 14	21
1	SUB 15	7
2	JZR 10	7
3	JNG 6	7
5	STO 14	7
6	JMP 0	7
0	LOD 14	...

8) Now click on *Log Window* and compare your trace with what the computer did as it ran the program. Did you get the same results? (We assume the computer was right, because, as everyone knows, computers *never* make mistakes!)

Exercise 2

Name ______________________________ Date ____________________

Section ______________________________

1) Start the "Super simple CPU" applet. Select *Example 3* (*Negative numbers*) from the pull-down menu.
2) Open the assembler window, click on *Load from Memory*, and then click *Disassemble*.
3) Take a screenshot of the assembler window.
4) Rewrite the assembler program, removing unneeded DAT lines and replacing addresses with identifiers. Be careful—there's a trap here! Several instructions shouldn't have identifiers as operands. Can you spot them? (Hint: Study the assembler program shown on the screenshot on the very first page of this lab as a model for what you are to do here. Make sure you understand what that program does and how it uses identifiers.)

Exercise 3

Name ______________________________ Date ____________________

Section ______________________________

1) Start the "Super simple CPU" applet.
2) Click on the "Assemble window" button. This brings up a blue window with several large text areas and some buttons.
3) In the left text area, type the following program. Make sure to put some spaces in front of the instruction mnemonic if there is no label. This means, for example, to hit the space bar several times before typing the "INP" in the first line.

```
      INP
      STO  A
      SUB  FIVE
      JZR  SHOW1
SHOW0 LDI 0
      OUT
      STP
SHOW1 LDI 1
      OUT
      STP
A     DAT  0
FIVE  DAT  5
```

4) Click on the Assemble button in the blue window. What do you see in the right text area? Can you make the correspondence between the mnemonics in the left text area with the binary codes in the right?
5) To transfer this program into the memory of the Super simple CPU, click on the button "Save to Memory." Close the blue window. Notice the binary version of the program has been copied into the memory cells. (The program is still there even though the blue window is invisible. Just click on "Assemble window" again to see it.)
6) Run the program. The first instruction will ask you for input. Type in 101, since the Super simple CPU requires your input to be binary. What is the output?
7) Click on Reset and Run the program again. This time, type in 111 and write down what the output is.

8) Look at the previous program and trace through the instructions. Imagine that you typed in 101. Circle all the instructions that are executed when this is what you inputted. Here's another copy of the program to mark up.

```
      INP
      STO  A
      SUB  FIVE
      JZR  SHOW1
SHOW0 LDI 0
      OUT
      STP
SHOW1 LDI 1
      OUT
      STP
A     DAT  0
FIVE  DAT  5
```

9) Two fundamental control structures in computer programming are *decisions* and *loops*. Does this program contain a decision? If so, which instructions trigger the taking of an alternate path?

10) Does this program contain a loop? If so, where is the instruction that causes the computer to jump back to an earlier spot?

Deliverables

Turn in your hand-written sheets for the questions requiring written answers. Also turn in the screenshots requested in the exercises.

Deeper Investigation

We have seen how the assembler software (which is part of the “Super Simple CPU” applet here, but is usually separate) translates assembler programs into machine language. Some of this process is straightforward, but some is not.

Think about what has to be done when translating the identifiers that replace memory addresses. How does the assembler do it? Would a table of identifier/address pairs be helpful? What happens to *forward references*? This refers to identifiers that are used as operands before we know what they turn into, as in the following:

```
    LOD A
    ADD B
    STO C
    STP
A   DAT 26
B   DAT 19
C   DAT 0
```

There are actually three forward references here, to A, B, and C.

Try to describe the algorithm that the assembler might use to translate an assembler program into machine language. Just use English to write your algorithm; by no means should you try to do it in assembler language!