LABORATORY 7

Computer Cycling

OBJECTIVE

- Learn about the fetch-execute cycle of computers.

REFERENCES

Software needed:

1) Super Simple CPU app from the Lab Manual website (Super Simple CPU.jar)

BACKGROUND

Review these topics from your textbook, lecture notes, or online resources:

- The stored program concept, memory
- ALU, input and output units, control unit
- Fetch-decode-execute cycle, machine code instructions, opcodes
- Number representation, binary numbers, two's-complement notation

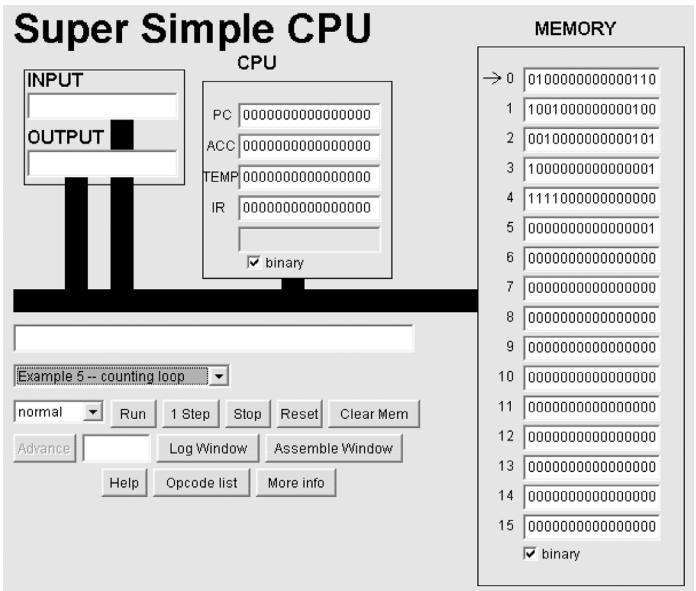
ACTIVITY

Computers are unbelievably complex, but at the lowest level they are surprisingly simple. A program consists of thousands, millions, or billions of instructions, all performed blindingly fast. But each individual instruction is quite simple.

If each instruction is simple and easy to understand, why does putting thousands of them together result in software that would tax the intelligence of Albert Einstein? People have been trying to figure that out for years, because figuring it out might give us clues to understanding, writing, and maintaining that software. The difficulty seems to be in what some call the "semantic gap" between problems that computer scientists want to solve using computers and the tools with which they solve them—tools such as software, programming languages, and even hardware. If the tools are too simple, we will have to use a lot of them in complex ways to get the job done. If the tools are more complex, they can do more in one step and are easier to describe. Think of how much work goes into preparing a meal, but how easy it is to place an order at a restaurant. In this lab, we will study the instructions of a very simple computer and then watch programs run in this computer.

Start the Super Simple CPU app. The input and output devices are merely text areas to the left. The CPU and memory are clearly marked, though the memory of this computer is ridiculously small: 16 words of memory, each 16 bits long, a total of about 0.0000305 megabytes!

Select Example 5 from the pull-down menu to load the example program, and then click the *Run* button. After a few seconds, you should see this:



As the program runs, it cycles through the basic instruction cycle, which is commonly called the fetch-execute cycle. This app displays the steps in the cycle in the blue text area in the middle.

There are three registers in this CPU:

PC	The program counter
ACC	The accumulator
IR	The instruction register

The PC register contains the address of the next instruction. It goes up by 1 after most instructions, but some, such as the JMP instruction, simply rewrite it entirely.

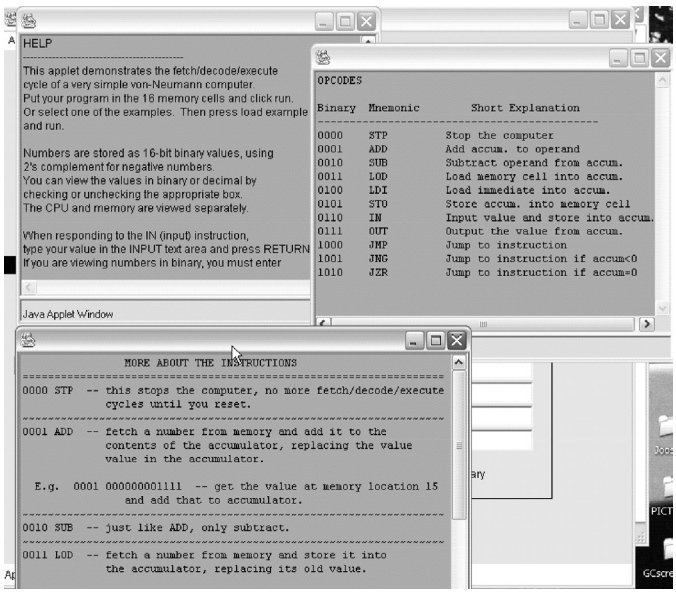
ACC is the accumulator, kind of like the one-number screen on many calculators. Values are added into it or subtracted from it, depending on whether the instruction is ADD or SUB. It also serves as the way station for values coming in from the input device and going out to the output device.

IR is the place where the current instruction is kept and decoded. In the app, the decoded values from the current instruction are displayed below IR. The mnemonic of the instruction, a short alphabetic name for the instruction, is shown along with the operand in decimal.

The contents of memory and the CPU's registers are usually displayed in binary, but they can be shown in decimal by unchecking the box. This change of display does not affect the values and can be done any number of times, even when a program is running.

Numbers are stored as 16-bit values in two's-complement notation. If the number has a 1 in the leftmost bit, it will be thought of as negative.

There is built-in help for this app. Click on the *Help* button near the bottom of the screen to bring up a window explaining the basic operation of the app. There are also buttons that summarize the opcodes and give more detailed explanations of them.



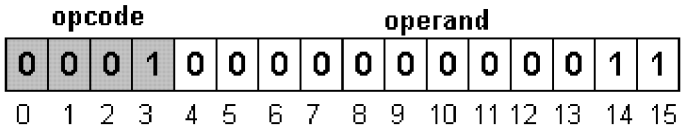
OPCODES

Binary	Mnemonic	Short Explanation
1111	STP	Stop the computer
0001	ADD	Add accumulator to operand
0010	SUB	Subtract operand from accumulator
0011	LOD	Load memory cell into accumulator
0100	LDI	Load immediate into accumulator
0101	STO	Store accumulator memory cell
0110	INP	Input value and store accumulator
0111	OUT	Output value from accumulator
1000	JMP	Jump to instruction
1001	JNG	Jump to instruction if accumulator < 0
1010	JZR	Jump to instruction if accumulator = 0

MORE ABOUT THE INSTRUCTIONS

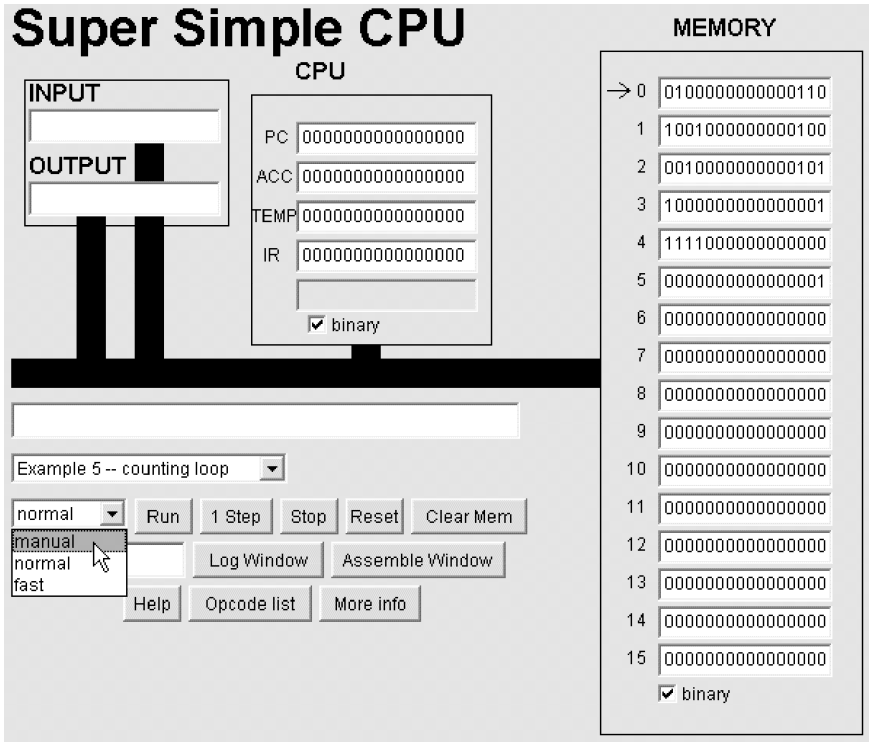
1111 SIP	This stops the computer; no more fetch/decode/execute cycles until you reset.
0001 ADD	Fetch a number from memory and add it to the contents of the accumulator, replacing the value in the accumulator. (E.g., 0001000000001111: Get the value at memory location 15 and add that to the accumulator.)
0010 SUB	Fetch a number from memory and subtract it from the contents of the accumulator, replacing the value in the accumulator.
0011 LOD	Fetch a number from memory and store it in the accumulator, replacing the accumulator's old value. (E.g., 0011000000001111: Get the value at memory location 15 and store that value in the accumulator.)
0100 LDI	Load immediate; the value to be put in the accumulator is the operand (the rightmost 12 bits of the instruction); do not go to memory like LOD. (E.g., 0100000000001111: Store the value 15 in the accumulator.)
0101 STO	Store the accumulator's value in memory at the indicated location. (E.g., 010 100000000 1111: Store the accumulator's value in memory location 15.)
0110 INP	Ask the user for one number and store that in the accumulator.
0111 OUT	Copy the value in the accumulator to the output area.
1000 JMP	Jump to the instruction at the indicated memory address. (E.g., 1000000000001111: Put the value 15 into the PC, which will cause the next instruction to be taken from location 15 of the memory.)
1001 JNG	Jump to the instruction at the indicated memory location if the accumulator's value is negative; otherwise, just add 1 to the PC. (E.g., 1001000000001111: Put the value 15 into the PC, if accumulator < 0; otherwise, go to the next instruction.)
1010 JZR	Jump to the instruction at the indicated memory location if the accumulator's value is zero; otherwise, just add 1 to the PC. (E.g., 10 1000000000 1111: Put the value 15 into the PC, if accumulator = 0; otherwise, go to the next instruction.)

Each instruction in this super-simple computer has two parts: opcode and operand.

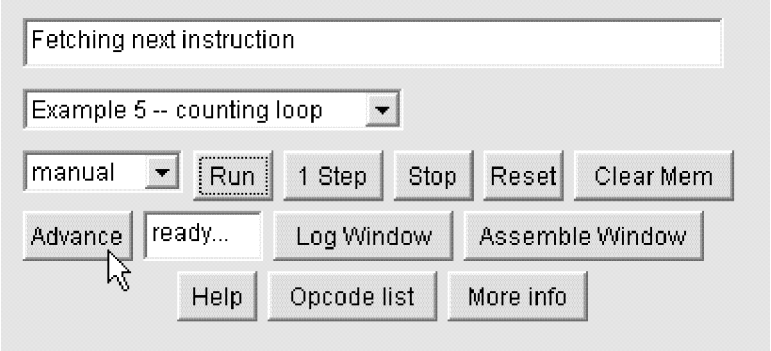


The opcode is the first four bits and represents the instructions. For example, 0001 is the ADD instruction, as the opcode reference list above shows. The operand can be several different things depending upon which opcode you are using. It is often the address of a memory cell, used by ADD, SUB, LOD, and STO. If the instruction is LDI (*load immediate*), the operand is a 12-bit constant that is put directly into the accumulator. The jumping instructions (JMP, JNG, JZR) use the 12-bit operand as the value that is stored in the PC register, thus causing control to jump to that spot in memory. A few instructions (STP, IN, OUT) ignore the operand altogether.

This CPU app allows you to watch each instruction in almost microscopic detail. If the app is running, press the *Stop* and *Reset* buttons. Then pull down the speed choice from the menu next to *Run* and select *manual*.



This function allows you to control the steps of each and every instruction. Click on *Run* to start the program. Each instruction begins and does one part of the fetch-execute cycle before stopping. When it stops, the word "ready..." appears in the box next to the *Advance* button. Information on what is about to happen is provided in the blue box above the buttons.



Using the *Advance* button to walk through instructions takes some time, but it allows you to carefully observe each part of each instruction. In addition, addresses and values that flow between the CPU and memory, and between the Input/Output unit and the CPU, are animated in yellow binary numbers through the red bus.

The normal speed disables the *Advance* button but still animates the instructions, while the fast speed does not show the animations, allowing the program to run quickly.

In addition to advancing each instruction, you can "single-step" a program. Single-stepping refers to running one complete instruction and then stopping the program by clicking on the *1 Step* button. Many debuggers for real programming languages also have a single-stepping option so you can dig into a problematic program and ferret out its bugs. Think of single-stepping as a sort of time microscope: You amplify the delay time between instructions to a very large amount so you can see what is going on.

There are a number of example programs, ranging from ridiculously short and simple to somewhat complicated. The greatest common divisor is fairly complex to understand, and it almost fills up the entire 16-word memory.

Load the GCD program (Example 6). Memory cells 0 through 10, inclusive, contain the program. Memory cell 14 contains the first integer value (in the example it is 18, which is 100102), and cell 15 contains the second integer (in the example it is 24, which is 110002).

It is extremely difficult to determine exactly what the program does by just looking at the contents of the memory. In fact, it is impossible to tell from memory alone whether a location contains a 16-bit integer or an instruction. So how does a computer "know" what is in a memory word? It never really does; only the PC determines that. If the memory word's contents are fetched into the IR and treated like an instruction, then they *are* an instruction.

Wait a minute! This opens the door to a terrible problem! What if a program "jumps" into a section of data? Can that happen? What will be the result? The answer to the second question is certainly yes, a computer can jump into data. This is one of the major implications of the *stored program concept*.

One downside of this ability to jump into data is that an improperly designed program can get the wrong jump instruction (because an improperly designed programmer wrote it that way!), start interpreting data as instructions, and probably do something terrible, like erase all of the memory or launch nuclear missiles. Hardware designers have invented ways to assure that this won't happen. Sadly, mean-spirited people can use this ability to change programs to write evil things called viruses.

One benefit of the ability to jump into data is that a program can create a new program, or even rewrite part of itself, and then execute it right away. Since programs are really data, this capability was seen by some early pioneers as a way for programs to learn and improve. It hasn't quite worked out that way, simply because learning is a hugely complex process (as you well know!), but the spirit of this dynamic execution is used in many aspects of computer science today.

The purpose of the Super Simple CPU app is not to turn you into a programmer. Rather, it is meant to allow you to see the fetch-execute cycle wend its path through a typical CPU. In the exercises, you will study a few instructions individually. The larger programs were included so that you can get a feel for how a complete program looks and acts at this low level.

EXERCISE 1

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app.
2) Select the first example by pulling down the examples list and selecting *Example 1—sequence*.
3) The program is three instructions long, in memory addresses 0, 1, and 2. Write down the three instructions or take a screenshot.

4) Click on the *Opcodes list or More info* button to get a key to the opcodes.
5) Decode the three instructions. Look at the first four bits of each memory cell and find those bit patterns in the opcodes key. Write down the corresponding three-letter mnemonic.

6) How many fetch-execute cycles will this program perform when you run it?

EXERCISE 2

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app.
2) Load and run the second example, which contains an *Input* and an *Output* instruction. Notice that when the app inputs, you are supposed to type a binary number in the *Input* box (which turns yellow) and press *Enter*. The program will output the number you entered.
3) Now we'll modify the program: In place of the STP instruction in word 2, encode a STO (store) instruction that will store the accumulator value in memory cell 15.
4) Since you replaced the STP instruction in step 3, you should create a new STP instruction in word 3, to make sure your program halts.
5) Run your program. When it finishes, you should see your input number in the memory cell labeled "15."
6) Take a screenshot of your program.

EXERCISE 3

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app, or press the *Reset* and *Clear Mem* buttons if it's already running.
2) Type the following instructions into memory cells 0 and 1. (The 16-bit numbers are broken with spaces to make them easier to read. Do not put spaces in your numbers when you type them in!)

```
0100 0000 0000 1110
0011 0000 0000 1110
```

3) In cell 14, type the following:

```
0000 0000 0001 0101
```

4) Run the program by single-stepping. Press the *1 Step* button. After the first instruction finishes (the blue box will say "Updating PC"), write down its three-letter mnemonic and the value in the accumulator.

5) Press *1 Step* again to run the second instruction and write down the mnemonic and the accumulator's value.

6) Take a screenshot.
7) Explain the difference between the two instructions. (Hint: They are both loading instructions, which means they cause a new number to be put into the accumulator. But there is a difference.)

8) Why do you suppose there is no “store immediate” instruction?

9) What sequence of regular instructions could be used to do the same thing? That is, how could we store a specific number, say 133, into a specific memory location, say memory cell 13?

EXERCISE 4

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app.
2) Load the counting loop example (Example 5).
3) Run the app and write down the PC after each instruction.
4) Take a screenshot and circle all the jumping instructions.
5) Be a human "disassembler." Translate the computer instructions in cells 1 to 5 into mnemonics. Remember, only the first 4 bits of each instruction is the opcode. To see the opcode list, click on the *Opcode list* button.

Mem cell	Mnemonic
0	_____
1	_____
2	_____
3	_____
4	_____

EXERCISE 5

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app.
2) Load the GCD example (Example 6). (A blast from your mathematical past: GCD is short for "greatest common divisor.")
3) Run it so you can see that it works. (You might want to speed it up! To do this, pull down the speed list and select "fast.")
4) Click the *Reset* button. Click on the binary checkbox at the bottom of memory so that decimal numbers appear in memory instead of binary. In location 14, type 16; in location 15, type 28. What do you think the GCD of these numbers will be?

5) Run the program. When it is done, take a screenshot. Find the memory cell that has the GCD and circle it.
6) What is the GCD of 30 and 31? You could use the app to confirm your suspicions. Click on these buttons: *Stop*, *Reset*. Then type "30" into cell 14, and type "31" into cell 15. Press the *Run* button.

EXERCISE 6 (THIS ONE IS CHALLENGING!)

Name _____ Date _____

Section _____

1) Start the Super Simple CPU app.
2) Load the GCD example (Example 5).
3) *Disassemble* the program in memory. This means write a human-readable version by decoding the instructions.

   When writing the program, put the memory address to the left, followed by the two- or three-letter mnemonic for the opcode. Follow that with a number if it is a jump or LDI instruction, or an arithmetic instruction (ADD or SUB). You could go one step further and replace 14 with a variable name of your choosing, and 15 with another variable name. For example, A and B will work.

   Here's the disassembled version of Example 4, "copy a number":

```
0      LDI      13
1      LOD      13
2      STD      12
3      STP
```

DELIVERABLES

Turn in your handwritten sheets showing your answers to the exercises, along with the screenshots as requested.

DEEPER INVESTIGATION

We have been very sneaky with the Super Simple CPU app. We have taught you some elementary programming! It is hard to examine a programmable device like a computer without talking about programming just a little bit. Perhaps getting your feet wet in this shallow water before officially learning the ins and outs of programming lessens the fear, or whets the appetite! Ask any programmer: Sure, programming can be frustrating, infuriating, or irritating at times, but above all, when you get it, *it is fun! Lots of fun!*

If every program is ultimately made up of thousands of individual, simple CPU instructions, as we saw in the Super Simple CPU app, what makes software so complex? Let's investigate by analogy, turning to other disciplines.

Think about a field of study that you enjoy: music, chemistry, politics, linguistics, geology. In most fields, there are basic elements out of which more complex things are built. Music has tones and rhythms. Chemistry has atoms and electron valence shells. Where does complexity come from? For example, in chemistry there are only so many different elements (kinds of atoms), and there are only a few different ways they can attach to one another and stick together. Why, then, are there some 100,000 compounds?

Write a few paragraphs in which you speculate on how complex things can arise out of combining a few basic elements. How can you put basic elements together to achieve complex systems?