CPU Simulator Lab

CIS 203 Computer Science II

Brian C. Ladd

Spring 2023

1 Note to Students

Several questions you must answer come before our explanation. This is on purpose. For you to get the most out of the learning with this lab, *please* do your best to answer the question, in writing, **before** reading past the checkpoint. Your check mark for answering the question depends on your making an honest effort to think about what is being asked; your mind will be primed to remember our explanation after engaging with the question. You will be able to link your thinking and our thinking for better understanding.

2 Pre-Lab

Before the lab you will need to watch several short video clips about the inner workings of a modern computer:

- What's a CPU?
- Fetch, Decode, Execute
- Assembly Language
- Primitive Data Types `float` or `double` can be ignored.
- Unsigned (Positive) Binary Numbers
- What is Abstraction?
- Why use Abstraction?
- Abstraction/Reality
- Creating an Abstract Model

Testing referred to in these videos is UK A Levels testing. You will not be expected to write code in any assembly language. *Abstraction* is an underlying idea in all of computer science.

3 Student Learning Outcomes (or What Should You Know?)

- Describe the architecture of a modern, stored-program computer and how it reflects the abstract properties of **digital**, **binary**, and **general-purpose**.
- Describe the **fetch-decode-execute** cycle and relate it to the parts of a computer.
- Relate the fetch-decode-execute cycle for machine instructions to the **running-time** of a program.
- Explain the difference between **hardware** and **software** in relation to the computer being general-purpose.

4 What is a computer?

Before you proceed, write down, in you notes, your answer to that question. Consider what *every* computer has in common and distill the definition down into no more than three sentences.

[ ] Checkpoint 1: Show us your answer to the question: What is a computer?

4.1 What is a program?

Pretend you are sitting at your computer. It is on and you are not running any application programs. You want to surf the Web. What do you do? And what does the computer do in response?

Picture what you would do to start the Web browser of your choice. After you command the computer to run it, what does the computer do? Go right up to the point where the browser is ready for you to enter an URL address.

[ ] Checkpoint 2: Show us your description of what happens when the computer begins running an application program.

4.2 A General-purpose Computer

Your computer is **general-purpose**. This means that the **hardware** is not a video player or a word processor. Instead, the hardware is an **emulator** that takes a detailed description of **special-purpose** computer (the video player or word processor) and follows the definition to emulate the special-purpose computer.

It its core, the hardware stores and interprets zeros and ones, binary numbers. Those binary numbers can encode integers, audio and video, and even **instructions** for the hardware to execute.

When you write a Java program, the compiler places the variables and instructions into specific memory locations by the compiler. The compiler reads your Java program file and writes another file to the hard drive with instructions and variables laid out so they can be read directly into memory. So, how does Firefox start on a Linux computer with XWindows?

Double click on icon the operating system associates the icon with a particular compiled program, `firefox`.

Load the program the operating system reads the contents of the file into the computer's memory.

Run the program the hardware has its own variable (a *register*) where it keeps track of the address, in memory, of the next instruction it should execute. So, to start the program, the operating system just points this register, the *program counter* to the spot where the compiler put the first instruction of the compiled program.

But how can just two symbols, {0, 1} *encode* all of these things.

[ ] Checkpoint 3: Show us your answer to that question. [Give this your best shot.]

4.3 A Binary Computer

How can two symbols, {0, 1} *encode* all of these things? Think about the Western European alphabet. It can be used to encode the complete works of William Shakespeare, *The Three Musketeers*, and even the Java programs you wrote for CS 1. In order for you to read any of these documents, you must know **what** language they are in (English, French, Java) and know (be able to *decode*) that language.

Similarly, the computer stores numbers in memory. To use the values correctly, it needs to know which locations hold instructions, `int` values, `char` values, or `String` objects. You give your variables types to tell the Java compiler how your program interprets the contents of the memory where the variable is stored.

The *alphabet analogy* above is one way to model combining low-level elements in different orders and proportions to get many different results. Can *you* come up with a different one?

Think about **cooking** for a start. You could construct a similar analogy between binary encoding and cooking with a limited number of ingredients.

[ ] Checkpoint 4: Show us your analogy for binary data encoding.

5 Fetch-decode-execute

5.1 Tracing Java Programs

```
1  // Sequential Instructions
2  public static void main(String [] args) {
3    int x = 11;
4    int y = 2 * x;
5    int z = x + 7 * y;
6    System.out.println(String.format("%d %d %d", x, y, z));
7  }
```

How does the above Java program execute? You could have put your finger on a line, figured out what it did, and then updated your model of the value of the variables.[1] Then you would advance to the next executable line and do it again. That *cycle* of behavior could be defined as:

fetch read the Java instruction currently pointed to

decode figure out what the Java instruction says to do

execute perform the instruction

Now, stop using your finger. Instead use a pencil and a box labeled *Pointer at Code* (`PC`). You could then extend the **fetch** stage as follows:

fetch read the Java instruction currently in `PC` and add `1` to `PC`.

Using the line number labels, write down the values of `PC` as the program executes. Also write down the output of the program.

What happens if the program is more complicated? What if there is *selection* (an `if` statement)?

```
1  // Selection Instruction: selection.java
2  public static void main(String [] args) {
3    int x = Integer.parseInt(args[0]); // read first argument as int
4    if ((x % 2) == 0)
5      System.out.println("Even");
```

[1] By using your *finger* you are simulating a *digital* computer. Terrible professor joke. Sorry.

```
6      else
7        System.out.println("Odd");
8      System.out.println("Bye!")
9    }
```

Think about using PC to trace this program.[2] The rules of our current fetch-decode-execute are not sufficient: sometimes we need to load a new address into PC rather than just increment the old value.

In this code either line 4 is followed by 5 and then the PC goes to 8 (*past* the `else` clause) **or** line 4 is followed by line 7 and then the PC goes to 8; one path or the other is chosen depending on whether `x` is even or odd.

We can extend `fetch-decode-execute` so that the instruction can update PC (if necessary) in the `execute` phase.

fetch read the Java instruction currently in PC and add 1 to PC

decode figure out what the Java instruction says to do

execute perform the instruction; update PC if necessary

Trace the values of PC when `java selection 2` is run. Also note the output. Then trace the values of PC when `java selection 1` is run and note the output.

[ ] Checkpoint 5: Show us your traces and output for both runs of the program.

5.2 Tracing Machine Instructions

Start up the CPU Simulator in a browser; you will have to have JavaScript enabled.

The cyan box on the right of the page is the *random-access memory* (RAM) with 15 bytes of memory. The salmon box on the left is the *central processing unit* (CPU). Notice that the red arrows connecting them: the *Control Bus* permits the CPU to send commands to and read status from the RAM; the *Address Bus* connects a particular byte in the RAM to the *Data Bus* for the CPU to load (from the RAM to the CPU) or store (from the CPU to the RAM).

In this simulation, input and output are pop-up boxes in the browser for user input when the program is running or putting values in the bytes if you click on RAM cells or registers.

Go to the Settings section of the Web page and find the Import/Export button. Click it and paste the following into the text box, replacing whatever was there before. This is our Add two numbers example (a little more complicated than the one on the Webpage). Make sure there are no spaces *before* what you paste and then press the Import button.

```
91 3f 91 3e 1f 92 00 00 00 00 00 00 00 00 00
```

Notice that the values stored in the first six locations in the RAM have changed from all zeros to different **binary** numbers. For the moment, leave it showing binary but in Settings you *can* change the number base (Binary is base 2, Denary is base 10, Hex is base 16).

Look *inside* the CPU and you will find the *registers*; a register is a named, one-byte storage location.[3]

- ACC:: *Accumulator* where one operand comes from for each math instruction and where the result is placed. For example, if you Add with the address 1000, ACC becomes ACC plus the contents of RAM[8].
- CIR:: *Current instruction register* where the fetched instruction is stored to be available throughout the cycle.

[2]There is no error checking. If you ran it without any arguments, the program would crash.

[3]The registers are like *variables* (each has a name and a value stored in it) and the RAM is like an *array* (there is one name, RAM, and individual values are found with an index to the memory location.)

- MAR:: *Memory address register* is the RAM index for fetching a byte from RAM into MDR or writing the MDR to a location in the RAM.
- MDR:: *Memory data register* is the value read from (on a fetch or Load operation) or written to (on a Store operation) RAM.
- PC:: *Program counter* (almost *place in code*) is the index in RAM where the next instruction is to come from.

When the program is loaded, the PC is initialized to the location where the **compiler**, **operating system**, and **hardware designer** have agreed program code is to begin. In this machine, that location is 00000000.

Use the Step button in the lower right of the Webpage to step through the program's first instruction. Read the status box at the bottom of the screen and pay attention to the fetch-decode-execute cycle presented there.

The PC is, initially, 00000000 in binary or 0 in base 10[4]. The PC is copied to the MAR at the first step of the fetch phase.

On the next step, the MAR communicates the address to the RAM through the Address Bus; this sets the RAM up to read or write RAM[0] (depending on the control signals sent from the CPU). The simulation highlights RAM[0] (where that 0 comes from the MAR).

On the next step, the contents of RAM[0] is communicated to the MDR through the Data Bus. The simulation highlights the data register and copies the 10010001 value from the RAM to the register across the Data Bus.

On the next step, the MDR is copied to the CIR. The CIR holds the encoded machine instruction for the duration of the decode and execute phases. It is a separate register than the MDR because the execute phase of a load or store instruction makes use of the MDR and the CPU would lose the instruction before execution was finished.

The last step of the fetch phase follows, incrementing the PC. All instructions in this example run *sequentially* (the execute phase never changes the PC) so the next instruction is always fetched from the address one higher than where the previous instruction was found.

The decoding phase begins at the next step by splitting the eight bits[5] in the CIR into two four bit values, the *opcode* (the leftmost four bits) and the *operand* (the rightmost four bits). The *operand* is used to look up what instruction is to be performed inside the Decode Unit.

The CIR value of 10010001 is split into the opcode of 1001 and the operand of 0001. The last two lines in the decoder say this opcode encodes either Input or Output.

On the next step, the operand of 0001 decodes as Input. The input instruction pops up a dialog and you can type in a *decimal* number. This time around, type in ”7”. The binary value for 7 is placed in the ACC as 00000111.

On the last step for this instruction, the CPU checks if any other part of the computer needs servicing. Since none does, the CPU begins the fetch-decode-execute cycle again.

Use the step button to watch the instruction in RAM[1] fetched into the MDR and then copied into the CIR. Do not step past the message “**Fetch** the *Control Unit* increments the *Program Counter*”. The PC should read 00000010.

Write down, in your notes, the name of the instruction that is about to be decoded: look at the CIR, break it into halves, and figure out what the *opcode* means. Without worrying about what it means, write down the *operand* next to the name of the opcode in your notes.

[ ] Checkpoint 6: Show us the name of the opcode and the operand you just wrote down. The operand is treated as an *address*: to what *register* in the CPU must the operand be copied during the execute phase?

[4]The UK seems to use *denary* as the adjective for a base 10 number; in the US, *decimal* is more typical.

[5]bit = **b*inary dig*it**

Step through the instruction until the message is about checking for interrupts (the end of the whole fetch-decode-execute loop).

During the execute phase just completed, at what index in the RAM was the ACC stored? You can find it by matching the value in the ACC with the `Value` in the RAM cell and then look at the `Address` next to it.[6]

Go ahead and push the `Run` button. Every two seconds, the simulation will step through the phases and instructions. You can watch the PC increment by 1 at the end of the fetch phase, look at the opcode as each instruction is decoded, and then watch the values move around in the registers and RAM as each executes.

If you want to run the same program again without reloading it from the Web (important after you make changes to the program later), just click on the `Reset CPU` button in the top row of buttons. **Beware** that if you reset the RAM, all the values in memory (instructions and variables) will be lost.

5.3 Uncompiling

What did the higher-level program that produced this program look like? We will use a Java-like syntax, simplifying things so we do not need a `class` or a `public static void main` line. Our code will become instructions starting at address 0 and variables stored in the highest addresses.

The program looked something like this:

```
int firstNumber; // assigned to RAM[15]
int secondNumber; // assigned to RAM[14]
firstNumber = Input();
secondNumber = Input();
Output(secondNumber + firstNumber);
```

The compiler read the declaration of the variable `firstNumber` and, as noted in the comment, assigned it the location of `RAM[15]`. This means that in the assembly and machine language, the address 15 will "mean" `firstNumber`. Similarly, location `RAM[14]` is where `secondNumber` is stored.

The code was compiled into the seven instructions in RAM locations 0-6. If we write names for opcodes and decimal operands, the assembly language looks like this:

```
0   Input
1   Store 15
2   Input
3   Store 14
4   Add 15
5   Output
6   End
```

Find the address where the `Add` instruction is stored. Write the address down in your notes.

Click on the RAM cell containing the `Add` instruction so that you can edit it. Using the `Decode Unit` as a guide, change the instruction so that it *subtracts* `firstNumber` from `secondNumber` rather than adding; all arithmetic takes place between the value in a RAM cell (the *operand*) and the value in ACC. Remember that an instruction is a byte[7] that has four bits of opcode and four bits of operand. You want to change the assembly program to:

```
0   Input
1   Store 15
2   Input
3   Store 14
4   Subtract 15
5   Output
6   End
```

[6] If you want the *decimal* value of the address, try switching the simulator to `Denary` and see what the address becomes. You should also see the value in the ACC and RAM location switch to 7, the value you gave as input.

[7] a *byte* is a collection of eight bits

Run the program, entering the value 15 for the first number and 31 for the second. Make note of the result displayed in your notes.

Take a glance at the "Java" version of the program. Think about what part of the program you changed by changing the `Add`.

[ ] Checkpoint 7: Show us your modified program. Be ready to point out the changed instruction. What was the result when you entered 15 and 31? Was it the value you expected? Can you use the "Java" listing above to explain why the output is correct?

6 Selection

Our `Add two numbers` example program and your modification of it demonstrates that the simulated CPU can run a *sequence* of instructions in a row; interesting computations require CPU support for *selection* of different instructions, depending on the data being processed.

6.1 Find the Biggest of Two Numbers

Think about the "Java" version of our program. How could we modify it to output the larger of the two values the user entered?

CS 1 to the rescue: use an `if` statement and output one of the two variables, depending on which is larger. Excellent.

```
int firstNumber = 31; // assigned to RAM[15]
int secondNumber = 42; // assigned to RAM[14]
firstNumber = Input();
secondNumber = Input();
if (secondNumber - firstNumber >= 0)
  Output(secondNumber);
else
  Output(firstNumber);
```

That might not be exactly what you would have written, especially in the Boolean expression in the `if` statement. You probably would have put the two variables on either side of the comparison operator. That will be addressed below.

6.1.1 How can the CPU select one of two paths?

Go to the `Settings` section of the Web page and find the `Import/Export` button. Click it and paste the following into the text box, replacing whatever was there before. This is our `Find the Biggest` example (a little more complicated than the one on the Webpage).

```
91 3f 91 3e 2f 89 5f 92 6b 5e 92 00 00 00 1f 2a
```

Looking at this as the more readable *assembly language* with all 16 memory locations shown, the loaded program looks like

```
0  Input                 ; input -> Acc
1  Store           15    ; store Acc -> RAM[15] firstNumber
2  Input                 ; input -> Acc
3  Store           14    ; store Acc -> RAM[14] secondNumber
4  Subtract        15    ; Acc - RAM[15] -> Acc (secondNumber - firstNumber)
5  Branch ACC >= 0  9    ; goto 9 if Acc non-negative
6  Load            15    ; RAM[15] -> Acc firstNumber
7  Output                ; Acc -> output
8  Branch Always   11    ; goto 11 ALWAYS
9  Load            14    ; RAM[14] -> Acc secondNumber
```

```
Output                ; Acc -> output
End                   ; End instruction
0                     ; Unused location
0                     ; Unused location
42                    ; secondNumber
31                    ; firstNumber
```

All the numbers are given in decimal. Before moving on: write down in your notes the values of `firstNumber` and `secondNumber` *before* execution begins. Remember that the variable *names* are gone but their specific RAM locations contain their values.

In the simulator, you can enter a values into the registers, just as you can into the RAM locations. Pretend that you are the processor and you want to print the bigger of the two variable values. Put the address of the "right" instruction in the `PC`.[8] Use `Step` or `Run` to watch the program execute from your chosen location check the output is `42`, not `31` (or anything else).

Once the program gives the right output, `Reset CPU` and put the correct value in `PC` again.

[ ] Checkpoint 8: Show us your register values in the simulator (the value in the `PC`) and then run it for us so we can see the output is the larger number. Also show us where you noted the values of the two variables.

6.1.2 A quick detour

Did you notice that the instructions for the machine were different than the "Java" statements. This is because the instructions the CPU knows are very, very simple and a "Java" statement might have to become more than one machine instruction. To get input for `firstNumber` and store it in the right location requires two instructions; displaying the value for `secondNumber` also requires two instructions.

The simplicity of the instructions is even more pronounced when it comes to selection. This machine has only three instructions to change the sequential execution:

`Branch Always` As used above, at the end of showing one value, to skip over showing the second value, the instruction at `8` resets the `PC` during the execute phase from `9` (= `8+1`; the value set at the end of the fetch phase) to `11`. `11` is the address of the first instruction **after** output of the other variable.

`Branch on Zero` Same as `Branch Always` **but** it only resets the `PC` if the `Acc` register is equal to zero.

`Branch Non-negative` Same as `Branch Always` **but** it only resets the `PC` if the `Acc` register is non-negative. We are not talking about the representation of negative numbers, just realize that some `int` values are negative.

What happens if the conditional branches do **not** replace the value in the `PC`? Then the value set at the end of the fetch remains and the program continues *sequentially* without taking the branch.

6.1.3 Comparison with Subtraction

In the previous checkpoint you set the value of the `PC` to `6` to output `firstNumber` and to `9` to output `secondNumber`. We need a conditional branch in instruction `5` to have the CPU set the `PC` depending on which number is bigger.

Pause and think about that for a second. Pretend that the `Acc` has a *negative* number if `firstNumber` is bigger.[9] Then a `Branch Non-negative` would set the `PC` to its operand if `Acc` is zero or positive.

[8]Again, if typing it in in decimal is easier, use the `Denary` button in `Settings`.

[9]Look back at the "Java" program above for spoilers on how we will get such a value in the accumulator.

Take a moment and think about that. Write down in your notes the *operand* value the `Branch Non-negative` should have. Using the `Decode Unit` and the `Binary` button in the settings, write down the binary for the whole **instruction** `Branch Non-negative X` where `X` is the operand value you determined above.

If no instruction in the computer stores a value to `RAM[12]`, what happens to the value in that byte? Similarly, what happens to the value `10010001=_2 stored in =RAM[0]` if no instruction stores the accumulator to `RAM[0]`?

Compare the instruction in `RAM[4]` to the subtraction in the "Java" program. How do you know that the correct value is stored in `Acc` even though `secondNumber` is **not** loaded before `firstNumber` is subtracted from the `Acc`.

[ ] Checkpoint 9: Show us your translation the `Branch Non-negative` instruction. Be prepared to explain what happens to the values in RAM addresses between `Store` instructions that address them. Be prepared to explain why `RAM[4]` calculates `secondNumber - firstNumber`, even without loading `secondNumber` before the subtraction.

7 Levels of Abstraction

7.1 Digital, Binary, General-purpose Computers

A computer is complex. It can be argued that modern *central processing units* (CPUs) are among the most complicate artifacts ever constructed by humankind; it can also be argued that large computer *programs* are at least as complicated. How can a human being understand a chip with more than twenty *billion* transistors?

Short answer: no human being can. Instead they either understand the function of some subset of those transistors **at the transistor level** or they elide some of the details and understand more of the chip with less resolution.

The longer answer is **abstraction**. Abstraction is "the process of leaving out of consideration one or more properties of a complex object so as to attend to others." [Websters 1913] Computers can be defined at multiple levels, each level leaving out (taking for granted) all the levels below and only focusing on one set of features.

When you learned to program in Java, for example, you could ignore how the operating system interacts with the hardware, how the hard drive works, or how data is stored inside the *random-access memory* (RAM). Instead you focused on defining Java classes, compiling and running them, and figuring out where the semicolons go.

In this lab you have looked at a lower level of the comupter, much closer to the silicon.[10] You watched the *bits* and *bytes* move through the RAM and CPU of a simple, simulated computer.

You saw how the values stored in the computer are **digital**: any one of the eight bits in a byte can be *on* or *off*, never halfway between; a byte can therefore hold 2^8 or 256 different values.

The values in the memory (registers and RAM) are just numbers. You saw that numbers can be expressed in different *bases* such as **decimal** and **binary** (and you used the `Settings` buttons to have the computer do the conversion for you).

Take a moment and speculate in your notes about why computers use **binary** to actually store the numbers, rather than a base of 3[11] or a base of 10.

Finally, you saw that a byte just holds a sequence of bits. If it contains the sequence `00111111` and it is loaded into the `Acc`, then it is interpreted as the number `00111111`$_2$ `or 145`$_{10}$. If, instead, it is at the location referred to by the `PC` and it is fetched and decoded, then it is the instruction `Store` (see `Decode Unit` for opcode `0011`) to location 1111$_2$ or 15$_{10}$.

[10]Also known as the *raw iron* of a computer.

[11]There actually was a ternary computer built in Russia at the end of the 1950's. It is weird.

The ability to encode numbers and instructions in the same bytes makes the computer **general-purpose**. If you put different instructions in the RAM, the computer can interpret the data contents of the RAM as numbers, strings, audio, or even 3-D terrain for a video game.

[ ] Checkpoint 10: Using the London Underground map/satellite image example of **abstraction**, explain to us the abstraction between *machine language* and *Java*. Also, show us your speculation on why computers use **binary** rather than another base.