CPU Simulator Project

Professor Pfaffmann
CS 203
Lafayette College

by
Allie Mullan

11/14/2016

Hypothesis:

I hypothesize that my CPU Simulator design will carry out a machine code program with sufficient accuracy in a simplified representation of how a CPU actually works. I also hypothesize that my design is appropriately efficient. My project does contain certain design decisions that limit efficiency and flexibility.

Design:

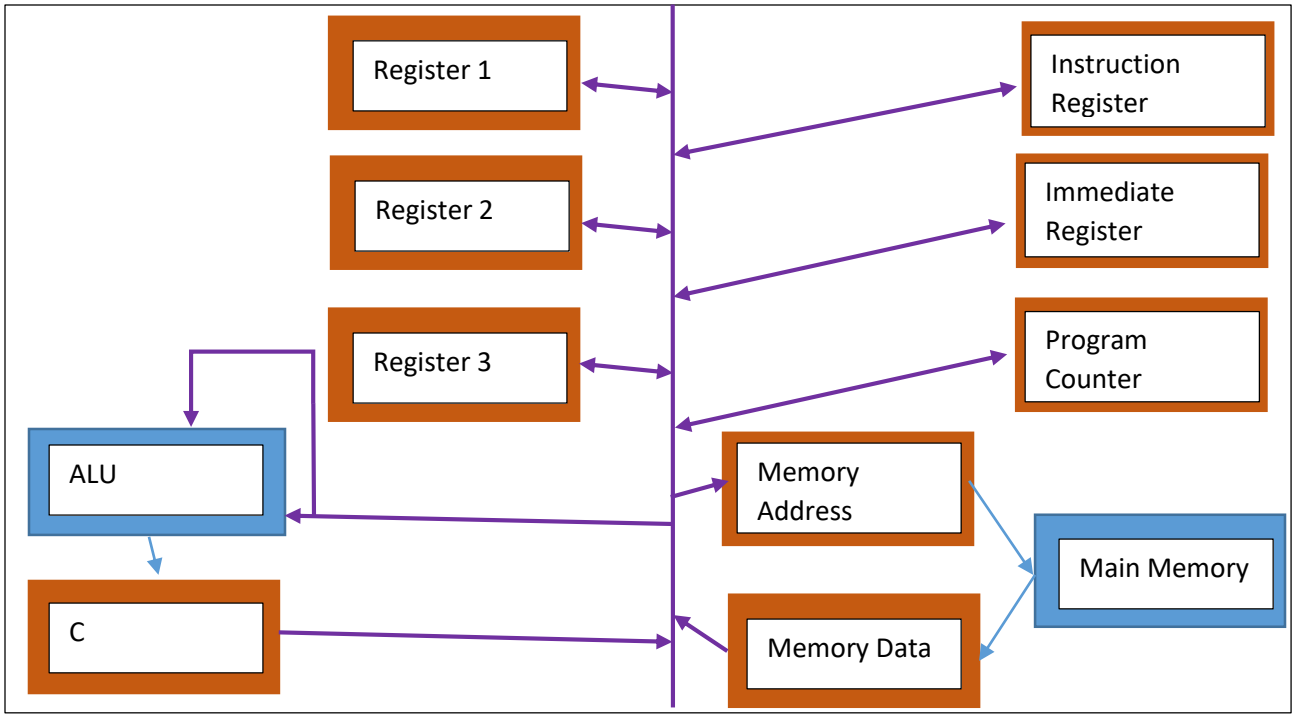

Figure 1(above): This is a visual representation of my CPU design. Register objects are shown in red while all other non-register objects are shown in blue. This example shows an implementation of the CPU with 3 general registers. The bus is represented by a purple arrow. Blue arrows represent the flow of information that does not go through the bus.

Config File

The config file is a text file that specifies all of the customizable aspects of the CPU. My config file specifies 4 different customizable aspects of the CPU. The first is word size. The word size refers to the length of instruction that the CPU can process at a time. Currently, the word size of my program is not customizable. It must remain at 16 because the program parses the instructions a certain way and does not allow for flexibility in the number of bits it uses for each section of the instruction. Ideally, I would incorporate flexibility into this aspect into the program.

The second customizable aspect is the bus size. This refers to how much information can fit in the bus. Currently, the way my bus is set up, the bus size in the config file does not matter. Ideally, I would like to pass the busSize variable into the constructor of the bus and have the bus keep track specifically of how many bits of it are filled up and how many are remaining. I would do this by including a bit check every time something was added to the bus. The number of bits of the item coming in would be calculated,

and then would be checked against the current capacity of the bus (which is the total capacity of the bus minus the number of already occupied bits). If the bus had enough room, the item would be added, otherwise the user would get an error print out. When something would be removed by the bus, it's current capacity would be increased by the number of bits in the item leaving the bus.

The third customizable aspect of the CPU is the ALU functionality. The ALU object has the capacity to perform a number of different operations such as addition and subtraction, however those operations can be turned on or off. The ALU functionality code is a 10-bit number of 1s and 0s. Each bit represents an operation that the ALU can do. For example, the bit at position 0 refers to the ALU's add operation, position 1, the subtract operation, and position 2, the multiply operation. If any particular bit is a 1, that means that the corresponding operation is "turned on," and if the bit is 0, it means the operation is "turned off." Figure 2 shows the ALU functionality code and which operation each bit in the code represents.

ALU Functionality Code Bit Representation

Add	Subtract	Multiply	Divide	Xor	Or	And	Sal	Sar	(no functionality)
Bit 0									Bit 9

Example: 0010100100 => In this instance, the ALU may use the operations multiply, xor, and sal.

Figure 2(above): Shows which operation each bit of the ALU functionality code represents. Each bit of the code is represented by a box. The bit is 1 if that operation is turned on, and 0 if it is turned off. An example ALU functionality code is provided.

The final customizable aspect of the CPU is the number of registers. This is the number of GeneralRegister objects that will be created when the CPUSimulator sets up the CPU and initializes all of its components.

If the user should want to adjust the config file, they should change only the numbers and not the Strings. In Figure 3, the pieces highlighted in yellow are the pieces that you can change to customize the program. The Strings cannot be changed.

wordSize 16

busSize 16

aluFunctionality 1111111111

noOfRegisters 6

Figure 3(above): A sample config file. The highlighted portions are what the user can change in order to customize the CPU Simulator. The Strings in front of the highlighted parts should not be changed.

RTN file

The RTN file is a file that contains the opcodes and the Register Transfer Notation for the CPU Simulator. The file is set up such as in Figure 4. The keyword “opcode” indicates the start of what is referred to in my program as the opcodeBlock. The opcode block contains the String name of the opcode and the Strings of the source and destination for each step of the RTN for that opcode. In my version of RTN, the String corresponding with the source object comes first, and the destination comes second.

My RTN file currently contains 11 opcodes- fetch, irmov, add, jmp, sal, sar, xor, and, or, sub, and rrmov. I planned on also implementing lea, mrmov, rmmov, and push and pop, but didn’t end up having time to implement every single opcode. The CPU can fully carry out instructions that use these 11 opcodes. If I were to add RTN instructions for the remaining opcodes, as well as indicate a 4 bit code for the opcodes in my instruction register, my program would theoretically be able to carry them out, though I have not tested this.

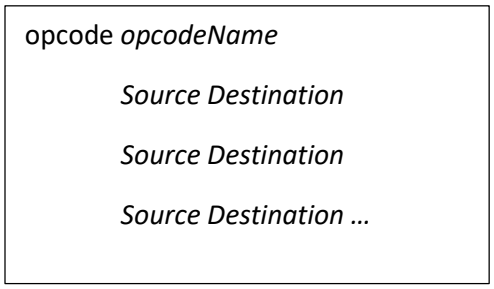

Figure 4(above): Shows the general format for the RTN file. Shows one “opcodeBlock” of opcode opcodeName and RTN instructions Source, Destination. The keyword ‘opcode’ followed by the actual name of the opcode comes first. The following lines contain one source and one destination separated by a single space.

Machine Code

The simulator includes 2 a files of machine code. These files are not customizable, but rather files that I created and hard-coded into the memory object. When the CPU Simulator is run, the MainMemory object parses the machine code file and saves each line to “main memory,” which in this case is a Hashmap in the MainMemory object. The Hashmap has a key of type Double and value of type Strings corresponding to a location in memory and the information stored at that address respectively. My first machine code file (machinecode.txt) contains 5 lines of code. Each line is a binary number of 16 bits. These 16 bits are the same 16 bits as the word size. This size cannot be altered because when my program parses the addresses, it uses a “.equals” comparison of two Strings to parse the instructions, and the Strings obviously must be of the same size for the program to register that they are equal and enter the necessary if statement.

The first machine code file (machinecode.txt) provides 5 lines of instructions. Figure 5 shows an assembly representation of machinecode.txt. The second file of machine code , called machinecode1.txt is represented in assembly in Figure 6.

```
0010011000111000 => irmov $24, %r3
0010011100100000 => irmov $28, %r1
0001100000111000 => add %r1, %r3
0010000001111100 => irmov $1, %r4
0011100000111110 => rrmov %r1, %r5
```

Figure 5(above): A translation of “machinecode.txt”. The machine code is on the left, and the corresponding assembly code is on the right of each machine code line.

```
0010000010100000 => irmov $2, %r1
0010011110110000 => irmov $30, %r2
0101110000100000 => sal %r2, %r1
0010000010111110 => irmov $2, %r5
0110100000111110 => sar %r1, %r5
```

Figure 6(above): A translation of “machinecode1.txt”. The machine code is on the left, and the corresponding assembly code is on the right of each machine code line.

CPU Components

The components of my CPU include the CPUSimulator, which is the main class, a parent class Registers, multiple child classes of Registers, and a number of other special functioning non-register objects. The program also includes several other helper classes. The Registers include the class GeneralRegister, which are the registers that are used explicitly in assembly code, such as %rax, %r1, and %r2. The other register classes represent the instruction register, the memory address register, the c register, the memory data register, the program counter, and an “immediate” register. The non-register objects are the ALU, the bus, and main memory.

There are also 3 helper classes- the ConfigFileReader class, the RTNReader class, and the GUI class.

Registers

The “Registers” Parent Class

I chose to use a parent class for the registers so that I could compare across classes. When executing RTN instructions, I generally had information moving from some source register to some destination register. Each register has the following methods: addContents(String x), to save a String to the register, getContents() which returns the contents, getFull() which returns whether or not the register currently has contents, and finally getName() which returns the String that corresponds with a String I chose for the name of the particular register.

The Instruction Register

The instruction register is a type of register that holds a String of contents, just like all the other registers. In the instruction register, this contents should be the next line of machine code that needs to be executed by the CPU simulator. The fetch RTN instruction gets the line from memory and places it in the instruction register. The instruction register has all the functions described in the parent Registers class, plus additional methods for parsing the instruction. I chose to put the instruction parsing here, because I wanted to take some functionality out of the CPU Simulator class. The instruction register has the method parseInstruction(). This method takes the line of machine code, which is the variable "contents," and splits it up according to the instruction format.

My instruction format (Figure 7) is hard coded into the program, allowing for simplicity for me as the coder, but sacrificing flexibility and function in my program, specifically word size. The first four bits of my instruction format correspond to the opcode. This means that there is a possibility for my program to be able to distinguish between 16 different opcodes. My project only utilizes 11. The next 6 bits correspond with "part A." Part A refers to both the source, and the first part of a line of assembly code after the opcode. Part B refers to the second part, as can be seen in Figure 7. The first bit of each part, shown shaded in Figure 7, is a 1 if the piece refers to a register, and 0 if it refers to a number.

The parseInstruction() method splits up the line of code into separate Strings and then calls methods to parse the Strings. The parsing is done in 3 methods – parseOpcode(), parsePartA(), and parsePartB(). These three methods take the relevant String and use if else statements to decode the String and determine what it is referring to. Then it saves Strings opcode, partA, and partB as different Strings depending on which if else statement was entered. These Strings are the literal names of the opcodes and registers or numbers, that can then be directly used in other String comparisons later on in the program execution. The remaining methods in the instruction register are getter methods.

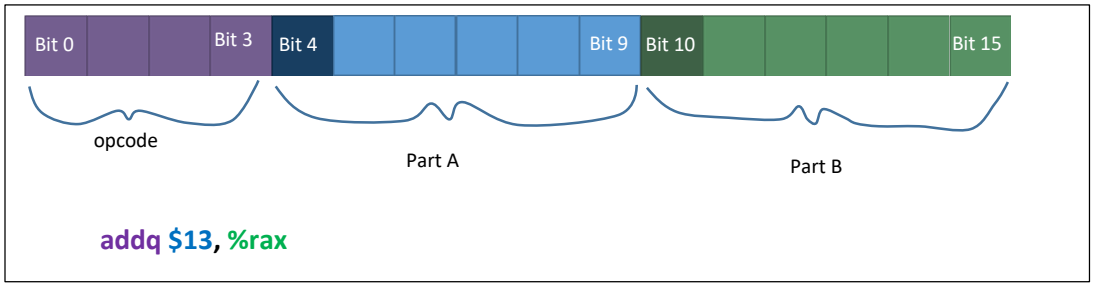

Figure 7(above): Details the instruction format. The purple bits correspond with the opcode number, the blue bits with the instruction part A, and the green bits with the instruction part B. The darker shaded bits correspond with the "register bit." A 0 in the register bit indicates that the piece of the address should be parsed as a number. A 1 in the register bit indicates that that piece of the address should be parsed as an address. Beneath, a color-coded line of assembly is provided, which corresponds with the instruction format diagram above.

The C Register

The C register is the register that holds the information coming from the ALU after it has executed an operation. It's String name is "CC" which corresponds to what it is referred to in the RTN file. The C register does not hold information from the bus. Instead, it is only used to hold the output from the ALU after the ALU has executed some kind of instruction. This instruction is passed directly from the ALU to the C register. The C register can then add that information to the bus to be moved elsewhere.

The Program Counter Register

The program counter register holds a value that corresponds to a location in memory from where the Memory Address register will pull from memory next. Technically it should hold a memory address, but in my program, for simplicity, it just uses a simple decimal Integer. With each fetch instruction executed, it is incremented by 1 so it will know to get the next line of code stored in memory. It's String name is "PC."

The General Registers

The general register is used in the CPU to represent the registers that can be directly utilized in assembly code, such as %rsi, %rdx, %rax, etc. The CPU Simulator generates as many of these registers as the config file specifies. They are named "Register " + some number which is unique to each register. They can hold a String which comes in from the bus, and can send their contents out into the bus as well.

The Immediate Register

The immediate register is the register where a number is put into if the machine code line specifies a number. If you have a line of assembly code that reads: irmovq $9, %rdx, then the 9 would be put in the instruction register. It is named "IM," not to be confused with the instruction register "IR."

The Memory Address Register

The memory address register holds a String that represents an address in memory. The memory address register is called "MA". When the RTN line M[MA] is called, whatever line of machine code that is stored in the main memory Hashmap at key of whatever is stored in MA will be pulled out and put into the memory data register.

The Memory Data Register

The memory data register holds a String that represents information that was pulled from some spot in main memory. The memory address register is called "MD".

Non-Register Objects

The ALU

The ALU is an object that takes an array of doubles called the ALU functionality code in the constructor. The ALU functionality code is a series of 1s and 0s that determines which of the ALU's operations can be used. The constructor saves these values and when each ALU operation

method is called, it performs a check to make sure that the operation's code is 1 before it can execute. The ALU works by having two variables called firstDouble and secondDouble. The ALU can take two pieces of information and preform an operation on them. My ALU has two methods for setting each of these variables respectively. When an operation is needed to be carried out, one would call the execute() method of the ALU. The execute() method takes in a String. This String should be the operator, such as "+", "-", "&", etc. The method uses String if else comparisons to determine which operator method it should call. It would check the input String, say "+", and then call the add() method, for example. In my code, I implemented an add() method, a subtract() method, a sar() method, sal() method, and() method, or() method, xor() method, divide() method, and multiply() method. Had I more time, I would have implemented more. The way my program is set up, I could include up to 16 opcodes.

The Bus

My bus holds data and a bus size. The bus size does not matter in my current implementation of the bus. The bus has an add method which adds a String to the bus' contents. The bus also has three move methods. The first moves the contents of the bus to a register. The second moves the contents to the first input of the ALU and the third to the third input of the ALU. I had to do this because the ALU is not a type of register and could not be used interchangeably with the Registers. In hindsight, I probably would have made a general object class rather than a register class because I did not actually have to distinguish between registers and non-registers, but rather had to use both types in many of the same ways.

Main Memory

The main memory contains a Hashmap. The Hashmap holds a key of type double, and a value of type String. The double refers to the memory location of the value, and the value refers to what information is stored at that location in memory. Main memory also parses a machine code file. The machine code file is hard coded in. In my submitted file, it contains the file "machinecode.txt" hard coded in, but I also included in my submission another machine code file, "machinecode1.txt" that can also be used to demonstrate the functionality of my CPU simulation. The main memory reads through the machine code file and saves each line of machine code into the Hashmap.

CPU Simulator Class

The CPU Simulator class acts as the main class. It contains a create method and a start method. The create method initializes all the registers and non-register cpu objects. It also begins running the GUI. It calls for the config file to be read, the rtn file to be read, and the machine code file to be read. It creates all of the CPU components, register, and non-register objects. One instance of each is created except for the GeneralRegister objects. The GeneralRegister objects are created in a loop and the number of registers created depends on what is specified in the config file. The start() method begins executing the machine code stored in main memory. It has a loop that calls fetch, then calls for the execution of whatever is in the instruction register. The fetch and instruction register execution are done in two separate ways. This is because I thought of two different ways to carry out RTN instructions. I got the first way to work for the fetch instruction, but then hit a wall and was unable to get it to work for the other opcodes. So I implemented a

very simple String comparison for the execution of the instruction register code. I parsed the source and destination, and then carried out the instructions specifically for that piece of individual RTN code. This makes for a very long and complicated set of if else statements and means that I have to look at the RTN file and make sure that every piece of RTN is represented exactly as a String in these if else statements. If I could do it over, I would make a CPU object parent class so that I could simply parse each CPU object that information is travelling between in the same way.

Helper Classes

The Config File Reader

This class reads and stores the info in the config file. It also contains getter methods for this information.

The RTN File Reader

This class reads and stores the info in the rtn file. It stores the RTN in a Hashmap with the String opcode as its key and an ArrayList of RTN as its value. Each time a new opcode is read from the machine code, the program goes to the RTN file reader and searches the Hashmap for the corresponding opcode String key and then gets the ArrayList of RTN instructions, from which it draws the appropriate sources and destinations to execute.

The GUI

The GUI is created during the setup in the CPU Simulator class. It creates a GUI with two buttons, “Step” and “Exit.” Exit stops the program and closes the GUI. The GUI also has a text box for each register object. The fields within the textbox show what is inside the register. When the registers get contents added to them, the corresponding field prints the String that was added to the register. I also have a TextArea that prints basic steps of the program as it executes. It will print the current opcode whose RTN it is executing, the source and destination of each step of the RTN instructions, and the final contents of the destination object after the RTN line is completed. Each of these pieces are in working order during the running of my program.

However, the “Step” button is not in working order. Ideally, the user would press this button each time they wanted the fetch or next step of machine code to be executed, then the program would execute the next step and then wait for the button to be pressed again for the following step to be executed.

I implemented this by having a method in CPUSimulator called waitForClick(). Once the program enters waitForClick(), it enters a while loop. It remains in the while loop until the “Step” button is clicked. Then, once the “Step” button is clicked, the program exits the while loop, and returns to executing the next step of the program. This was possible using a Boolean called “keepGoing” in the GUI class. keepGoing starts off as false and becomes true one the “Step” button is clicked. The while loop exits when keepGoing becomes true, and then once it exits the loop, it resets keepGoing back to false so that the next time waitForClick() is called, the while loop will be entered.

This however, only works the first 2-4 times that waitForClick() is called. Though the Boolean logic is being set up properly, as far as I can tell at this point, and the while loop is being entered and exited properly, after a certain number of calls, the program will exit the while loop and just freeze. I am not sure at this point how to fix this problem, however, if I were to continue working on it, I would look into different methods of pausing the execution of the program instead of a while loop.

Conclusion:

Putting it all Together

The basic execution of the program is as follows. The user inputs an array of String to the CPU Simulator class. The Strings are the names of the config file and rtn file respectively. The names of the files I used are configfile and rtnfile, so I inputted the Strings "configfile.txt", "rtnfile.txt." The CPU Simulator creates and reads the configfile and rtnfile. Then it creates the main memory, which reads a hard coded file of machine code. Then all the other CPU objects are instantiated. Program execution begins in the start() method that has a for loop that loops through the machine code program size, ie the number of lines of machine code in the machine code file. For each line, it calls the fetch instruction first. The fetch process uses the RTN fetch instructions to pull the first line of machine code stored in memory into the instruction register. Then the line is parsed in the instruction register. Parsing means splitting up the line of binary into three pieces. Each of the three pieces are put through if else statements which set a variable equal to a different String which represents the name of the opcode, the name of the register, or a number. Next, the CPU Simulator uses the opcode String from the instruction register to get the RTN instructions for that opcode. It loops through the pieces of the RTN instructions 2 at a time. The RTN instructions are stored in an ArrayList in order of source, destination, source, destination, and so on. With each iteration of the for loop it gets a source and a destination. Next, I used if and else loops to parse the source and destination String. In general, the if else statements determine which CPU object or register the RTN instruction is referring to, takes the contents of that object or register and adds it to the bus. The destination is then parsed in the same way, and in general, the bus deposits its contents in the destination object/register. There are some cases where the RTN instructions follow multiple parts or are not quite as simple as one source and destination, and so the instructions are a bit more complicated. This is why I had to hard code every case of RTN separately. The simulation ends when the last line of machine code is executed.

Issues

Overall, if I had to do the project over again, I would change the inheritance so that carrying out RTN instructions could be a simpler and interchangeable process, instead of a very long set of if else statements, with very specific instructions inside each one. I would create a parent class for CPU objects, that would refer to every object that can hold values, and not just registers. This would simply include all of the non register objects as well as the registers. Another thing I would like to change is the fact that I used Strings. I did this because when I scanned the addresses from the machine code file, they began as Strings and if I tried to parse them into Integers or doubles I would run into an error. Instead of figuring out how to resolve that error, I just left them as Strings. I also would have addressed the issue of word size and bus size. Neither of those variables actually made a difference in the program. I would also have

to somehow, when parsing the machine code lines, make it so variable lengths of Strings could be parsed as each part of the instruction format. I currently do not have an idea of how I could do this, besides converting the Strings to some kind of numerical value, however I could not do this without generating an error that I did not know how to get around. Another issue I would address is that of the GUI which I would make properly utilize the “Step” button by looking into a way to pause the program until the “Step” button is clicked. Finally, had I more time, I would expand my RTN and my machine code program to be able to do more.

References:

Stalling, William. "Instruction Sets: Characteristics and Functions." In *Computer Organization and Architecture: Designing for Performance*, 348-429. Boston: Pearson-Prentice Hall, 2016.

Pfaffmann, Jeffrey. *CS203 Project 1 Instructions*.

Pfaffmann Jeffrey. *Simple GUI.*