Kardelen Demiral 2018400027

Kardelen Erdal 2018400024

CMPE230 PROJECT 2 REPORT

HOW DID WE HANDLE OUR PROBLEMS?

The first problem was to implement an assembler. It was actually quite straight-forward so with the help of professor Ozturan's converter code, we implemented it in a short period of time. We iterated the file 2 times. In the first one we put label names with their locations into a dictionary. In the second one we basically interpreted every line into a machine code. We read the code in the input file line by line and using funcions like "split" and "strip", easily tokenized the lines. For every line we determined the addressing mode, operation type and the operand, then called the converter function with this data. Throughout the process, we tried to state the errors in the .asm file. But since we don't have much clarification on errors, we just did the ones that comes our minds.

The second problem was basically simulating a CPU. At first, we didn't quite get what to do and acted if this was an easier problem than it actually is. Forgetting the fact that memory blocks are 8bits, we filled a 64k length memory with 16 bit integers in decimal. However, we realized that it was a big mistake. After thinking thoroughly, we decided to implement it in a more real-like way. We put the registers in an array and since they hold 16 bit integers, we let the them keep integers in decimal. For the memory, we put the data in every block as 8 bit binary strings. This way, we could finally store and retrieve the data correctly. For example when we need to store a value in an address xxxx, firstly we converted it into a 16 bit binary string, then split it into two, put them in the addresses xxxx and xxxx+1. For this program, we wrote a function called "*get_value*" and made use of it in several operations. It basically returns the value to use in operations considering the addressing mode. For example if the addressing mode is 11, it returns the value in the index of the memory where the operand specifies. We also have a function called "*store*". Again considering the addressing mode it puts the value in the register A to the location the operand specifies. We also have other functions like "*get_inst_add_type*" which returns the instruction and addressing types of a statement. The other functions are simple and just their names are enough explain them.

The other thing we were confused hence had to change our code several times is the subtracting operation. Since negative integers are not recognized by our CPU, we had to do this operation in a different way in SUB, DEC and CMP operations. We first did the subtraction in decimal and if the result is negative, we took the two's complement of it, then converted it into a positive decimal number. This actually worked fine but since we thought this way the flags could be set wrongly, we decided to do this operation as firstValue + not(secondValue) +1. If the value of firstValue + not(secondValue) +1 is greater than 65535, we subtracted 65536 from this value, and set the carry flag. We had some trouble about DEC operation too but at the end we decided to do it like SUB. The other operations were easy compared to these so we implemented them without any confusion.

If you want to use our READ operation, you should give characters one by one, pressing enter button.

Kardelen Demiral 2018400027

Kardelen Erdal 2018400024

POSSIBLE ERRORS

For error checking, we wrote functions in both programs. For syntax errors, we wrote a funciton called checkLine in the first program. It checks every line if they are written correctly in accordence to our assembly language syntax. If not, it gives a syntax error and exits the program. We called this function in the first iteration of the file. For runtime errors, we considered 3 errors. Invalid instructions, memory limit exceedings and popping from an empty stack. For the first one, in the function were we get the instruction type from the machine code, we checked if it is greater then 1C. If that so, we gave an error. For the second one we wrote a "memoryCheck" function. It is called whenever we the CPU has to reach the memory. If the location exceeds the length of the memory, it gives an error and exits the program. For the last one, we checked if the stack pointer points the end of the memory when POP instruction is being executed.

CONCLUSION

There are a couple of things about the project that we are disappointed. We've learned Regex in Python and we should have used it since it would be really helpful with the first part. However, since we did not spend so much time with Regex, and also Python generally, we did not even try to use Regex. We would feel much stronger if we had used it. Also, there were so much ambiguous points especially about carry flag. It was so ambiguous that even in the 14 testcases, there are not even one JC or JNC instruction. So, we are not sure if we handle it in right way or not. I wish there would be much clearer description, or clear answers to our questions. Nevertheless, we completed the project (still have questions on our minds).

It was a fun project. We learned a lot about CPU executions, memory and python!