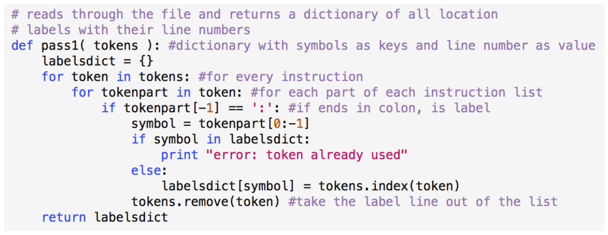
Zena's CS232 Project 8

For this project, we were able to work in whatever language we liked to create an assembler, which is essentially what translates user-input instructions into machine code (ones and zeroes). I made my assembler in python.

Task 1 was to download the template and get familiar with what it does. The main thing it does is convert the string from the file it reads in into tokens, which are separated based on where the space is. The output of the tokenizer function is a list of lists, which is essentially a list of all lines in the file. The spaces are turned into commas, which separate the contents of each line. This means that when referring to tokens[0][0], it would take the first line, and the first string on that line, which is, incidentally, either the operation or a label. And speaking of labels, this brings us to the second task.

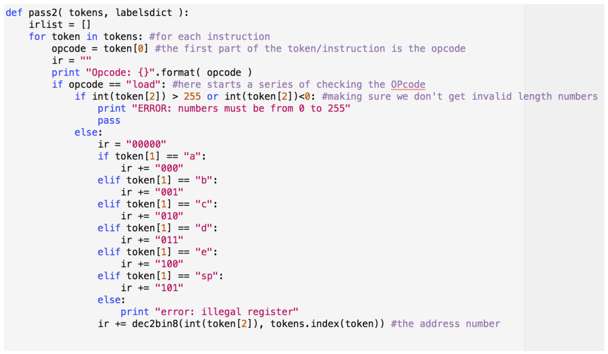
Task 2 was to make the pass1 function, which would take in any labels, which are places that the programmer can return to through a branch, now indicated by string instead of being a fixed address. The way I wrote pass 1 is shown here:



```python
# reads through the file and returns a dictionary of all location
# labels with their line numbers
def pass1( tokens ): #dictionary with symbols as keys and line number as value
    labelsdict = {}
    for token in tokens: #for every instruction
        for tokenpart in token: #for each part of each instruction list
            if tokenpart[-1] == ':': #if ends in colon, is label
                symbol = tokenpart[0:-1]
                if symbol in labelsdict:
                    print "error: token already used"
                else:
                    labelsdict[symbol] = tokens.index(token)
                tokens.remove(token) #take the label line out of the list
    return labelsdict
```

I first read in the list of lists, "tokens", then create a dictionary where I will store any labels and their line numbers I find in the tokens. The way I decide if something is a label is by looping over every instruction and every string in the instruction, and checking if the last index (-1) has a colon or not. If it does, I store it in the dictionary. And if the dictionary already has the label, then I give an error message. Since pass2 is about translating instructions into binary, it does not need the labels anymore, and so I get rid of the line that had the label within the loop. I return the dictionary so I can pass it into pass2 along with the tokens.

Pass2's job is to translate characters into a list of binary instructions. This then gets printed line-by-line into a file, which I'll get to later. I made a series of if's to check what the operation (index 0 of each list within the tokens list) is, printing an error message if it was something we don't check for, or an undefined operation. Then, within each of those, what I checked depended on which operation it was. Most would have 1, 2, or 3 more indexes of the line to check. Return, for example, would not, though, because it does not need any registers specified like operations such as addition, loading, or subtraction would need. If a user had a letter like "a" within the correct operation, the string "000" would be written to the instruction-representing string that was being built throughout the pass2 function, because a corresponds to "000" on the tables. At the end of pass2, I would append the instruction to an instruction list, then print the instruction. I also had the opcode print above the binary instruction. The general structure of pass2 is pictured here:

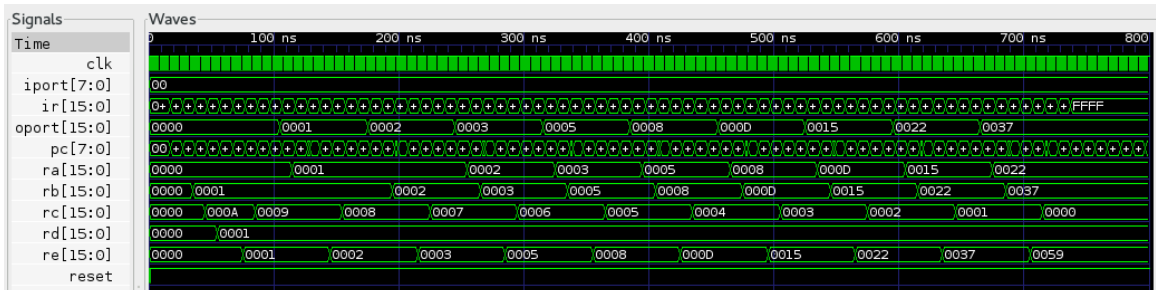


```python
def pass2( tokens, labelsdict ):
    irlist = []
    for token in tokens: #for each instruction
        opcode = token[0] #the first part of the token/instruction is the opcode
        ir = ""
        print "Opcode: {}".format( opcode )
        if opcode == "load": #here starts a series of checking the OPcode
            if int(token[2]) > 255 or int(token[2])<0: #making sure we don't get invalid length numbers
                print "ERROR: numbers must be from 0 to 255"
                pass
            else:
                ir = "00000"
                if token[1] == "a":
                    ir += "000"
                elif token[1] == "b":
                    ir += "001"
                elif token[1] == "c":
                    ir += "010"
                elif token[1] == "d":
                    ir += "011"
                elif token[1] == "e":
                    ir += "100"
                elif token[1] == "sp":
                    ir += "101"
                else:
                    print "error: illegal register"
                ir += dec2bin8(int(token[2]), tokens.index(token)) #the address number
```

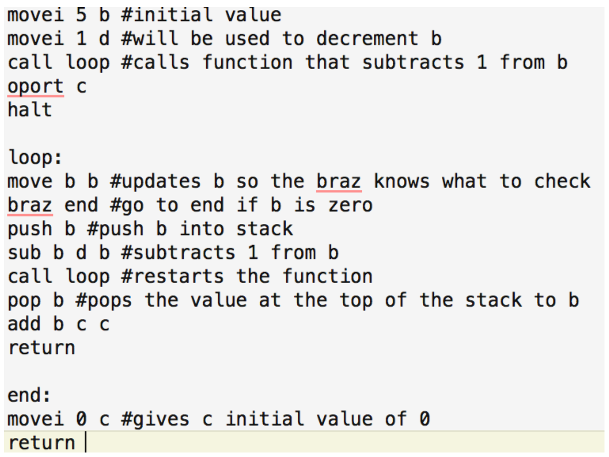
The fourth task (the pass2 function was the third) was to make the main function that would actually open a file to read in, then write to a newly-generated one. The reading was done for us; the filename would be defined on argv[1], and the output of tokenizing that file was stored to the variable tokens. After closing that file, I set labels equal to the returned value from passing the tokens through pass1 and the instruction list (irlist) equal to the output of passing the tokens and labels through pass2. I set filename equal to the input filename minus the extension .txt and plus ".mif", so that the name of the file would have its extension be changed to ".mif". Writing required opening the file as a mif using "with open() as mif", which Will told me to use when the way I was doing it before resulted in error messages saying the file was closed when I tried to write to it.

And what I did write to it was each instruction from the binary instruction list. Before each instruction, I also needed to put the line number, which required decimal to hex conversion. Each time through the instruction-line-writing loop, a variable would increment by one then be passed into the write method. After each line, I also had to tell the program to skip a line. Then, out of the loop, I also needed the other details like depth, width, end, etc. to make the file readable by vhdl.

Task 5 was to write the fibonacci sequence as assembly. I compiled it and tested it with gtkwave. The output looked the same as last time because the generated mif file was, naturally, the same. Here it is, with the oport showing the sequence like last time:

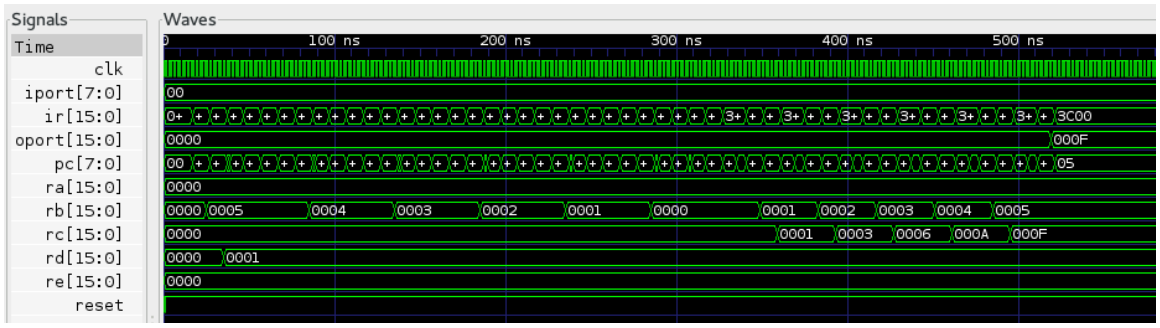


Task 6 was to make a recursive function in assembly, my process for which is shown here:

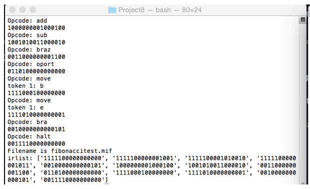


```
movei 5 b #initial value
movei 1 d #will be used to decrement b
call loop #calls function that subtracts 1 from b
oport c
halt

loop:
move b b #updates b so the braz knows what to check
braz end #go to end if b is zero
push b #push b into stack
sub b d b #subtracts 1 from b
call loop #restarts the function
pop b #pops the value at the top of the stack to b
add b c c
return

end:
movei 0 c #gives c initial value of 0
return
```

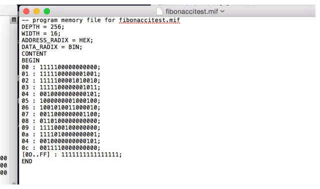
It decrements b from 5 to 0, then brings it back up to 5 again, but while adding its value to c and storing the result in c this time around. The gtkwave for this is shown here:



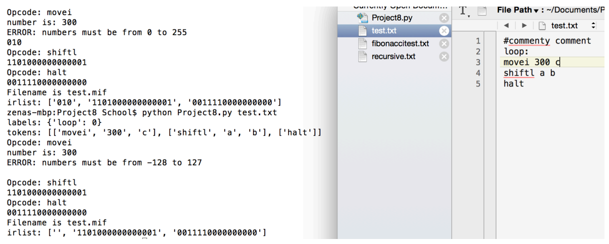
I did the second extension, which is to point out errors and suggest what the user do. For instance, operations that can take in a number can only use a number from 0 to 255 (-128 to 127 with two's complement); I made it so the instruction couldn't be stored if the number was invalid, and then printed a statement telling this information to the user. I also had each operation and its instruction print so you can tell what you're getting the error for. I printed out the filename, tokens, and instruction list, too. Pictures displaying these functions are below:



```
Opcode: add
1000000001000100
Opcode: sub
1001010011000010
Opcode: braz
0011000000001100
Opcode: oport
0110100000000000
Opcode: move
token 1: b
1111000100000000
Opcode: move
token 1: e
1111010000000001
Opcode: bra
0010000000000101
Opcode: halt
0011110000000000
Filename is fibonaccitest.mif
irlist: ['1111100000000000', '1111100000001001', '1111100001010010', '1111100000
001011', '0010000000000101', '1000000001000100', '1001010011000010', '0011000000
001100', '0110100000000000', '1111000100000000', '1111010000000001', '0010000000
000101', '0011110000000000']
```

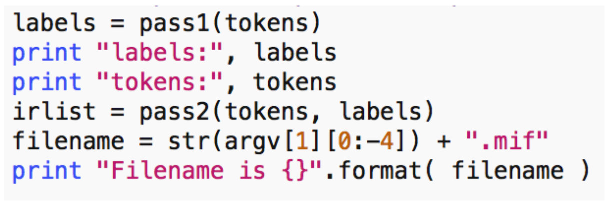


```
-- program memory file for fibonaccitest.mif
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 1111100000000000;
01 : 1111100000001001;
02 : 1111100001010010;
03 : 1111100000001011;
04 : 0010000000000101;
05 : 1000000001000100;
06 : 1001010011000010;
07 : 0011000000001100;
08 : 0110100000000000;
09 : 1111000100000000;
0a : 1111010000000001;
04 : 0010000000000101;
0c : 0011110000000000;
[0D..FF] : 1111111111111111;
END
```



```
Opcode: movei
number is: 300
ERROR: numbers must be from 0 to 255
010
Opcode: shiftl
1101000000000001
Opcode: halt
0011110000000000
Filename is test.mif
irlist: ['010', '1101000000000001', '0011110000000000']
zenas-mbp:Project8 School$ python Project8.py test.txt
labels: {'loop': 0}
tokens: [['movei', '300', 'c'], ['shiftl', 'a', 'b'], ['halt']]
Opcode: movei
number is: 300
ERROR: numbers must be from -128 to 127

Opcode: shiftl
1101000000000001
Opcode: halt
0011110000000000
Filename is test.mif
irlist: ['', '1101000000000001', '0011110000000000']
```

```
#commenty comment
loop:
movei 300 c
shiftl a b
halt
```



```python
labels = pass1(tokens)
print "labels:", labels
print "tokens:", tokens
irlist = pass2(tokens, labels)
filename = str(argv[1][0:-4]) + ".mif"
print "Filename is {}".format( filename )
```

This project was great because it helped me remember a lot of python from last semester. I learned more about reading in files, which I wanted to understand since it seems interesting and very important. I also had a lot of fun figuring out what list parts I needed to use where. It was great to have a break from vhdl, as well, although I didn't miss the indentation errors.

Thanks to: Professor Maxwell, Melody, Will, Tatsuya