Lab 2
Shelby McGarrah
sm52376

File Locations

quick sort results: in the tar, quickSortResults folder
y86 fib file: in the tar, y86Code folder
y86 quick sort file: in the tar, y86Code folder

Fibonacci Instructions

Nth number	Number of Instructions
0	9
1	12
2	36
3	63
4	114
5	192
6	321
7	528
8	864
9	1407
10	2286
11	3708
12	6009
13	9732
14	15756
15	25503
16	41274
17	66792
18	108081
19	174888
20	282984

Challenge Questions

I measured the average instructions per second using my fib program. With an n value of 30, the program was running 34806561 instructions, and it took an average of 3.4 seconds. So 34806561/ 3.4 is roughly 10 million (10237223) instructions per second.

To keep things simple, I just laid out my memory with the code at the bottom of the memory, at address 0. I placed the stack at the bottom of the memory, and grew it down towards 0. Of course, there isn't any real check to see if we're overriding code with our stack growth, but with such a simple simulator, and such a large memory space, I just stuck to a simple linear stack growth.

When I was coding in y86, I followed the typical x86 calling conventions. Though I took extra precautions and often pushed registers onto the stack to save them, even if they were caller saved, and I was in a callee function, for example. But keeping the consistent calling conventions allowed for a simpler exchange of information between methods.

The only y86 instruction that I didn't use was the xorq instruction. (I also didn't use iaddq or leaveq because they weren't specified in the book.) The only reason I didn't use xorq was because I never needed to. I used andq in its place if necessary. I also didn't use conditional move, just because conditional jumps made more sense to me at the time. It's easier to say "if this is true, jump here and continue", than "if this is true, move a value into a register and continue executing as normal". (I figured it would be easier to debug, too). But because of the y86's limited instruction set, I used the rest of the instructions consistently.

I took a lot of precautions to try and keep my code from breaking in the Fibonacci subroutine. I probably could have spared the extra pops and pushes, and so it definitely would have been more efficient to just pass values down the chain without saving them. I didn't even consider not using the traditional calling conventions, though, as I wanted to be able to debug my code more easily.

I didn't use any tools in particular to develop my simulator. I just used basic debugging techniques, print statements, and comparing return values. I developed what I thought was a fairly good testing rig for my simulator. It's able to compare the values in the registers, memory and whatnot, and determine if the exit state is what it should be. I also wrote up debugging comments as I went, and that provided me with most of the insight into the inner workings of my simulator that I needed to know. If I ever needed to examine what what happening at a particular spot in the execution, I'd just insert a halt, and see what the values of the state struct were. I left a lot of my tests in the tar file that I turned in (though I left out some of the messier ones). For my tester.c file, the expected results could be put into a file with the format:
As a side, my tester can be run with the following file format as the "expected results" file:

```
(
MEMORY
)
(registers)
0
0
0
0
0
0
0
0
0
```

0
0
0
0
0
0

(flags (z, s, o))
0
0
0
(status)
0
(pc)
0
And thus allowing for a check on the returned state struct.

Short File Descriptions

InitialState: file to initialize a state struct from a file input
instructions.h: define the number of instructions, registers, register names, etc.
state.h: contains the state struct that represents the machine of the simulator
tester.c: wrapper that allows the simulator to be tested in a cleaner, more straightforward way.
Util.h: helper methods to work with reading from a file
y86-assembler: program to take a y86 file and produce a file with address byte mappings
y86-simulator: program to take a file with address byte mappings and run the simulation.

Y86Code: folder with fib and quicksort files.
Tests: folder with a few of my test files, their expected results, and the y86 that produced them.
QuickSortResults: results from running quickSort on the given datasets

Brief Simulator Description

The main driver of the simulator is y86-simulator.c. It reads in from an input file of the form (address . Byte), and first builds an initial state struct with memory, registers, a pc, a status, and condition flags. Then the simulator starts processing instructions at memory location 0. An instruction will be read in from memory, and depending on what the instruction is, the state will be updated accordingly.

There are a few helper classes that I built, namely util which deals with file processing, and initialState, which will take care of the input file for the simulator. Thus the y86-simulator is only dealing with the method that are actually executing instructions.

All of the program's state is stored in a struct called state (which is in the state.h file). There, things like the memory length, the starting rsp and whatnot are indicated. (Though initializations of an instance of a state struct occur in initialState.h file.)

In the execute instruction method inside of the simulator file, you can see that instead of writing a switch statement to call whatever method necessary, I just got the instruction code, and used that to index into an array of function pointers. Then from there, the appropriate method will get called.

I also did my best to try and error handle, but because all the simulator sees are “1’s and 0’s”, the only time the simulator ever stops is if a halt instruction is read, or an invalid instruction is read. I also try to check that RSP doesn’t overflow whenever calls, returns, pushes, or pops alter the stack. Beyond that, if there is an expected register value (like 0xF), the simulator will print out a warning, but execution will continue.

When running both the fibonacci and quicksort algorithms, I used my tester.c file to wrap the call to the simulator, that way I could get the terminating state back and evaluate it. So I would call runSimulation, and the state struct that is returned contains the final state of the machine after running the program. I would then use that to evaluate the return value in rax for fibonacci, or the memory locations at 4096 for the quicksort algorithms.

I included snippets of the results for both programs (I’m not including all 1000 elements of the last test set? I’ll put that in a file inside of my turn-in tar):
quick sort on the length of 11:

```
4096   145
 4097   201
 4098   83
 4099   211
 4100   226
 4101   181
 4102   244
 4103   200
 4104   70
 4105   253
 4106   149
 4107   202
 4108   172
 4109   54
 4110   8
 4111   230
 4112   167
 4113   82
 4114   95
 4115   20
 4116   29
 4117   183
 4118   116
 4119   255
 4120   10
 4121   0
 4122   0
 4123   0
 4124   0
 4125   0
 4126   0
 4127   0
 4128   224
 4129   213
 4130   84
 4131   24
 4132   113
 4133   202
 4134   132
```

4135 0
4136 8
4137 2
4138 8
4139 180
4140 146
4141 157
4142 252
4143 8
4144 112
4145 137
4146 153
4147 99
4148 168
4149 163
4150 65
4151 32
4152 207
4153 252
4154 130
4155 196
4156 63
4157 223
4158 118
4159 35
4160 205
4161 103
4162 213
4163 25
4164 86
4165 253
4166 124
4167 57
4168 52
4169 38
4170 158
4171 43
4172 86
4173 72
4174 229
4175 111
4176 76
4177 231
4178 238
4179 90
4180 217
4181 36
4182 91
4183 113

results for the fibonacci series:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584 4181 6765

Just a note, because of all the pushing and recursion that's going on in the quicksort, it takes a long time to run the length of 1001. And due to the lateness of the datasets given, I unfortunately was unable to

debug the errors I had in the longer length ones. (I had tested on several smaller files, but I suppose there’s a problem with the recursive backtracking. C’est la vie)