THE UNIVERSITY OF SYDNEY

COMP2017 / 9017 Assignment 1

Due: April 19th, 10pm AEST (Week 6 Thursday)

This assignment is worth 4% of your final assessment

Task Description

In this assignment you will be implementing and performing operations on a virtual stack. You will need to emulate this virtual stack to store and reference variables and stack frame contexts before reading a set of pseudo assembly instructions that dictate the operations that should be performed on the stack. Your program will take a single command line argument being the name of the file containing your pseudo assembly code.

Before attempting this assignment it would be a good idea to familiarise yourself with the stack, registers, stack frames, stac t you will need to track your pw mory and handle files.

Command Format

You will be provided with a series of commands as operations taking the form:

([Address][Address_type])[Address][Address_type][Operation Code]

A series of operations form a function and will have some associated unique 4 bit label.

The second address and address type field is optional and will not be required for unary operations. The address type specifies whether it is a stack address, a value, a register address or a pointer to another stack address. Stack addresses are 7 bits long, register addresses are 3 bits long and values are 8 bits long and pointer addresses are also 7 bits long.

The address type itself is two bits long and specifies the type of the preceding address.

- **00** - value: 1 byte long. The value in the preceding 8 bits should be interpreted as a single byte value.
- **01** - register address: 3 bits long. This address refers to one of the eight fixed registers
- **10** - stack address: 7 bits long. This refers to an address on the stack containing a single byte.
- **11** - pointer valued: 7 bits long. This refers to an address on the stack containing a single byte that itself refers to another address on the stack. This is useful for accessing the stack pointer. The registers cannot be accessed using this pointer.

Note that using the stack pointer to allocate a variable should result in the value of the stack pointer being incremented.

The opcodes associated with these pseudo-assembly instructions are detailed below.

Opcodes:

000 - [MOV] - Pushes the value at some point in memory to another point in memory (register or stack)

001 - [CAL] - Calls another function, the first argument is a single byte (using the '00' type) representing what function is being called, the second argument is the stack address that starts a sequence of arguments to be passed to the function. When the function returns, the return value should be placed on the next available stack address.

010 - [POP] - Pops memory from the stack, to be returned to the calling function. Your implementation should manage the storage of this value.

011 - [RET] - Terminates the current function, this is guaranteed to always exist at the end of each function. There may be more than one RET in a function.

100 - [ADD] - Takes two register addresses and adds the values, storing the result in the first listed register

101 - [AND] - Takes two re t
listed register

110 - [NOT] - Takes a register address and performs a bitwise not ope
address. The result is stored in the same register

111 - [EQU] - Takes a register address and tests if it equals zero, the va
if it is 0, or 0 if it is not. The result is stored in the same register.

You will need to read each of the op-codes and implement the operation on the memory specified.

Each function is defined with a one byte header dictating the label of the function and the number of arguments, and a one byte tail specifying the number of instructions in the function. The function with the label 0 is the main function and should be executed first.

```
[Padding bits]
[function label (4 bits)]|[number of arguments (4 bits)]
    [OPCODE]
    [OPCODE]
    ...
    [RET]
    [Number of instructions (1 byte)]
[function label (4 bits)]|[number of arguments (4 bits)]
    [OPCODE]
    [OPCODE]
    ...
    [RET]
    [Number of instructions (1 byte)]
```


The first few bits of the file are padding bits to ensure that the total number of bits in the file accumulates to a whole number o n one byte.

The last byte in each functio on

The assembly code given for each of these functions will use relative a that the address 0x00 will store the stack frame pointer, 0x01 will store store the program counter and 0x03 will store any arguments to the fun then 0x03 will be unused.

You have a fixed set of registers, each with an associated three bit label (0-7). Each register can store a single byte at a time. You may note that some of the operations can only be performed on the registers.

Examples

Remember that the addresses used within each function are relative, you will need to translate these appropriately when calling across multiple functions.

The following assembly function moves the values 3 and 5 to separate registers before adding them, moving the value to the stack and returning that value.

Some equivalent C code might look like this:

```
#define BYTE unsigned char // Because all values are 1 byte
int main()
{
        BYTE r1, r2;
        BYTE s3;
        r1 = 3; // Storing the value 3 at register 1
        r2 = 5; // Storing the value 5 at register 2
        r1 = r1 + r2; // Storing the value 3 at register 1
        s3 = r1; // Store the register value on the stack
        return s3; // Return the value at the stack address
}
```


The code is split and comme


```
0000000 # Padding f
0000|0000 # Function 0 with 0 arguments
    00000101|00|000|01|000 # MOVE the valu
    00000011|00|001|01|000 # MOVE the valu
    000|01|001|01|100 # ADD registers 0 and 1
    000|01|0000011|10|000 # MOVE register 0 to 0x03
    0000011|10|010 # POP the value at 0x03
    011 # Return from the function
    00000110 # 6 instructions in this function
```


This function should return the value ‘8’.

The binary representation of the file will look like this:

```
00000000000000000000101000000100000000011000010100000001000001110000000001110010011000001100
```

Notice the padded zero bits at the **start** of the file.

And when you try to read the file (for example using vim) it should look like this:

```
\x00\x00\x00\n\x04\x01\x85\x01,\x088\x03\x93\x06
```

This can also be re-written by taking advantage of the stack pointer. Remember that the stack pointer should increment each time it is called.

```
# Some number of bits of padding
0000|0000 # Function 0 with 0 arguments
    00000101|00|000|01|000 # MOVE the value 5 to register 0
    00000011|00|001|01|000 # MOVE the value 3 to register 1
    000|01|001|01|100 # ADD register 1 to register 0
    000|01|0000001|11|000 # MOVE register 0 to the next
    # address specified by the stack pointer (0x03 here)
    0000011|10|010 # POP the value at 0x03 containing `8'
    011 # Return from the function
    00000101 # 6 instructions in this function
```

And we can also begin to concern ourselves with implementing loops or switches by changing the value of the program counter.

```
# Some number of bits of padding
0000|0000 # Function 0 with 0 arguments
    00000010|00|001|01|000 # MOVE the value 2 to register 1
    0000010|10|000|01|000 # MOVE the value of the
    # program counter to register 0
    00000011|00|010|01|000 # MOVE the value 3 to register 2
    001|01|010|01|100 # ADD register 2 to register 1
    000|01|0000
    # to the program
    # counter incre
    0000011|10|010 # POP the value at 0x03
    011 # Return from the function
    00000111 # 7 instructions in this function
```


Note that as the program counter is a single byte, this indicates that the maximum number of instructions in a function is 255. Also note that the above program will never terminate. (You will not be given test cases that do not terminate, but even in this minimalist language you can encounter infinite loops).

Of course given our limited range of stack addresses, we can also have stack overflows:

```
# Some number of bits of padding
0000|0000 # Function 0 with 0 arguments
    000|01|0000001|11|000 # MOVE register 0 to the next
    # address specified by the stack pointer (0x03 here)
    000|01|0000001|11|000 # MOVE register 0 to the next
    # address specified by the stack pointer (0x04 here)
    000|01|0000001|11|000 # MOVE register 0 to the next
    # address specified by the stack pointer (0x05 here)
    ...
    ...
    0000011|10|010 # POP the value at 0x03
    011 # Return from the function
    10000010 # 130 instructions in this function
```

When run this should display:

```
``Stack Overflow!''
```

Helpful Hints

Start by reading in the operation code from the above examples, you will need to use bitwise operations here and should refer to the relevant tutorial sheet. Switch and case statements will be useful here.

You don't know the number of padding bits at the start of the file in advance!

As you do not know the sizes of some of the objects before you read the associated files, you will need to dynamically allocate memory.

Spend some time working out how the stack pointer, the frame pointer and the program counter work before getting the rest of the program written.

You will not be tested on the internal state of your stack and as a result you have a degree of flexibility in your implementation.

Don't forget to free any allocated memory!

You will not be given invali

Evaluation and Submission

You will submit your assignment using Edstem. You are encoura s and files to check the validity of your program while also submitting it to E

> **4 Marks** are based on automatic marking of the assessment, automatic test cases include test cases that are both visible and hidden on the Edstem assignment submission page.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment specification or if your code is unnecessarily or deliberately obfuscated.

Academic declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Information Technologies, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of IT for the purpose of future plagiarism checking.