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:

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

01 - [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.

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

11 - [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 register addresses and ANDs their values, storing the result in the first listed register

110 - [NOT] - Takes a register address and performs a bitwise not operation on the value at that address. The result is stored in the same register

111 - [EQU] - Takes a register address and tests if it equals zero, the value in the register will be 1 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 of bytes. The number of padding bits will always be strictly less than one byte.

The last byte in each function dictates how many operations are performed within the function

The assembly code given for each of these functions will use relative addresses and always assume that the address 0x00 will store the stack frame pointer, 0x01 will store the stack pointer, 0x02 will store the program counter and 0x03 will store any arguments to the function. If no argument exists 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:

```c
#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 commented for ease of reading.

```
0000000 # Padding for the whole file!
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 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|0000010|10|000 # MOVE the value of register 0
    # to the program counter, don't forget the program
    # counter increments!
    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 invalid inputs, excepting cases where a stack overflow occurs.