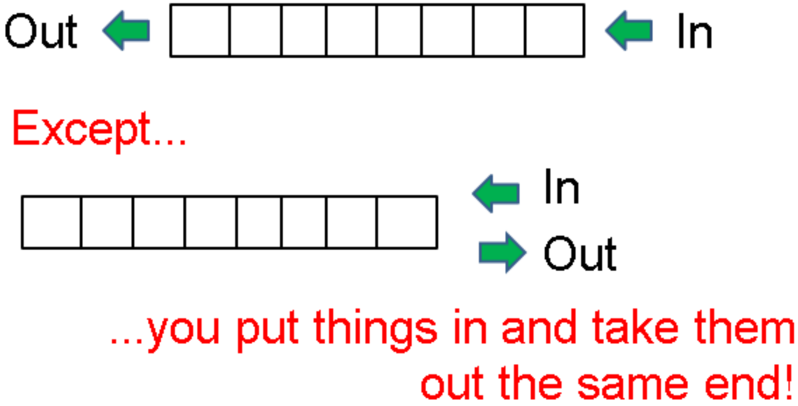
Chimera-2018-A Emulator Assignment

Practical 3 - Stack

CANS Tech INC

The Stack is a very important construct in computers

The stack can be thought of as a strange type of queue...



Whereas a normal queue is referred to as a
First In First Out (FIFO) queue

a stack is referred to as a
Last In First Out (LIFO) queue

Examples of Stacks...

Picture source: Internet

In the computer the stack resides in memory

The top of the stack is pointed to by the Stack Pointer (SP) Register in the CPU

We...
Push data onto the stack

And we...
Pull (Pop) data off of the stack

When data is pushed onto the stack the following happens...

1. The stack pointer is decremented
2. Memory[StackPointer] = data

When data is pulled off the stack the following happens...

1. data = Memory[StackPointer]

2. The stack pointer is incremented

Implementing the PSH Instruction

Once again inside the Group_1 function switch add

```
case 0xE5: // PSH
      CODE HERE
      break;
```

PSH		Addressing	Opcode
Pushes Register onto the Stack		A	0xE5
		FL	0xE6
Flags:	- - - - - - - -	B	0xE7
notes		C	0xE8
		D	0xE9
		E	0xEA
		F	0xEB

Firstly we need to check that the address held in the stack pointer is valid...

```
if ((StackPointer >= 1) && (StackPointer < MEMORY_SIZE)){

}
```

But why StackPointer >= 1?

Next we copy the data and decrement the stack pointer ...

StackPointer - -;
Memory[StackPointer] = Registers[REGISTER_A];

Giving...

```
if ((StackPointer >= 1) && (StackPointer < MEMORY_SIZE)){

StackPointer - -;
Memory[StackPointer] = Registers[REGISTER_A];
}
```

Implementing the POP Instruction

Once again inside the Group_1 function switch add

```
case 0xF5: // POP
    CODE HERE
    break;
```

POP	Addressing	Opcode
Pop the top of the Stack into the Register	A	0xF5
	FL	0xF6
	B	0xF7
Flags: - - - - - - - -	C	0xF8
notes	D	0xF9
	E	0xFA
	F	0xFB

Firstly we need to check that the address held in the stack pointer is valid...

```
if ((StackPointer >= 0) && (StackPointer < MEMORY_SIZE - 1)){

}
```

Notice the difference this time?

Next we copy the data and decrement the stack pointer ...

Registers[REGISTER_A] = Memory[StackPointer];
StackPointer + + ;

Giving...

```
if ((StackPointer >= 0) && (StackPointer < MEMORY_SIZE - 1)){

Registers[REGISTER_A] = Memory[StackPointer];
StackPointer + + ;
}
```

Compile and run your code to see how many marks you have!

Implementing the JP Instruction

Once again inside the Group_1 function switch add

```
case 0x05: // JP
    CODE HERE
    break;
```

JP		Addressing	Opcode
Loads Memory into ProgramCounter		abs	0x05
Flags:	- - - - - - - -		
notes			

First we need to get the address of the function that we are going to call...

lb = fetch();
hb = fetch();
address = ((WORD)hb « 8) + (WORD)lb;
This is exactly the same we did for loading.

And then set the Program Counter to its new value...

ProgramCounter = address;

Giving...

```
case 0x05: // JP abs
    lb = fetch();
    hb = fetch();
    address = ((WORD)hb « 8) + (WORD)lb;
    ProgramCounter = address;
    break;
```

Compile and run your code to see how many marks you have!

But beware! Now that you are implimenting intructions that effect the ProgramCounter your program may go insane!

Implementing the CALL Absolute op-code

CALL pushes the contents of the program counter (the address of the next sequential instruction) onto the stack and then jumps to the address specified in the CALL instruction.

Implementing the CALL Instruction

Once again inside the Group_1 function switch add

```
case 0xF3: // CALL
    CODE HERE
    break;
```

CALL		Addressing	Opcode
Jump to subroutine		abs	0xF3
Flags:	- - - - - - - -		
notes			

Next we need to validate the address in the stack pointer...

```
if ((StackPointer >= 2) && (StackPointer < MEMORY_SIZE)){

}
```

CALL works the same as JP but before we jump to a new location we push the current Program

StackPointer - -;
Memory[StackPointer] = (BYTE)((ProgramCounter » 8) & 0xFF);
StackPointer - -;
Memory[StackPointer] = (BYTE)(ProgramCounter & 0xFF);

Giving...

```
hb = fetch();
lb = fetch();
address = ((WORD)hb « 8) + (WORD)lb;
if ((StackPointer >= 2) && (StackPointer < MEMORY_SIZE)){
StackPointer - - ;
Memory[StackPointer] = (BYTE)((ProgramCounter » 8) & 0xFF);
StackPointer - - ;
Memory[StackPointer] = (BYTE)(ProgramCounter & 0xFF);
}
ProgramCounter = address;
```

Implementing the RTN op-code

RTN (return) does the opposite of CALL

The RTN instruction pulls two bytes of data off the stack and places them in the program counter register.

Program execution resumes at the new address In the program counter.

Once again inside the Group_1 function switch add

```
case 0x0F: // RTN
    CODE HERE
    break;
```

Next we need to validate the address in the stack pointer...

```
if ((StackPointer >= 0) && (StackPointer < MEMORY_SIZE - 2)){

}
```

Next we need to pull the address off of the stack...

```
lb = Memory[StackPointer];
StackPointer++;
hb = Memory[StackPointer];
StackPointer++;
```

Then set up the program counter with the new address...

```
ProgramCounter = ((WORD)hb << 8) + (WORD)lb;
```

Compile and run your code to see how many marks you have!

Now you can implement
PSH, POP, JP, CALL, RTN, JCC, JCS, JNE, JEQ, JVC, JVS, JMI, JPL, JLS, JLT, RCC, RCS, RNE, REQ, RVC, RVS, RMI, RPL, RHI, RLE,

It seems a lot but many follow...

If (flag set or not set)
{
Jump/Branch...
}

Questions?