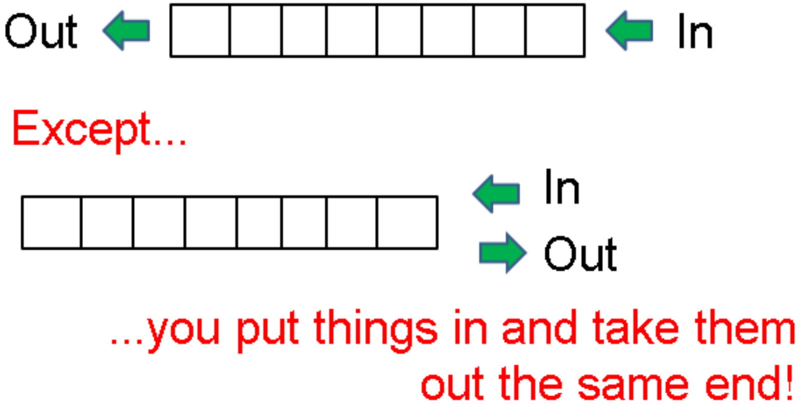
# Chimera-2016-I Emulator Assignment

## 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 PUSH Instruction

Once again inside the Group_1 function switch add

```
case 0xAE: // PUSH
    CODE HERE
    break;
```

| PUSH | | Addressing | Opcode |
|---|---|---|---|
| Pushes Register onto the Stack | | A | 0xAE |
| | | FL | 0xBE |
| Flags: | - - - - - - - - | B | 0xCE |
| notes | | C | 0xDE |
| | | L | 0xEE |
| | | H | 0xFE |

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 0xAF: // POP
    CODE HERE
    break;
```

| POP | | Addressing | Opcode |
|---|---|---|---|
| Pop the top of the Stack into the Register | | A | 0xAF |
| | | FL | 0xBF |
| | | B | 0xCF |
| Flags: | - - - - - - - - | C | 0xDF |
| notes | | L | 0xEF |
| | | H | 0xFF |

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 JUMP Instruction

Once again inside the Group_1 function switch add

```
case 0x38: // JUMP
    CODE HERE
    break;
```

| JUMP | | Addressing | Opcode |
|---|---|---|---|
| Loads Memory into ProgramCounter | | abs | 0x38 |
| 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 0x38: // JUMP 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 JMPR Absolute op-code

JMPR 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 JMPR instruction.

# Implementing the JMPR Instruction

Once again inside the Group_1 function switch add

```
case 0x07: // JMPR
    CODE HERE
    break;
```

<table>
<tr><td colspan="2">JMPR</td><td rowspan="4"><table><tr><th>Addressing</th><th>Opcode</th></tr><tr><td>abs</td><td>0x07</td></tr></table></td></tr>
<tr><td colspan="2">Jump to subroutine</td></tr>
<tr><td>Flags:</td><td>- - - - - - - -</td></tr>
<tr><td colspan="2">notes</td></tr>
</table>

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

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

}
```

JMPR works the same as JUMP 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 RT op-code

RT (return) does the opposite of JMPR

The RT 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 0x23: // RT
    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
PUSH, POP, JUMP, JMPR, RT, JCC, JCS, JNE, JEQ, JMI, JPL, CCC, CCS, CNE, CEQ, CMI, CPL,

It seems a lot but many follow...

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

# Questions?