Chimera-2016-I Emulator Assignment

Practical 2 - Flags

CANS Tech INC

Flags are very important in all processors

They store status information between instruction executions

This means that information can be passed between instructions as they execute

Why would we want to do this?

Have you ever wondered what...

```
if (var1 >= var2) {
}
else {
}
```

...compiles to?

It re does something like this...

```
if (var1 >= var2) {
}
else {
}
```

```
LD
MVR
CMP
JCS else_label
JUMP ende_label
else_label
end_label
```

It actually does something like this...

LD
MVR
CMP
JCS else_label
JUMP ende_label
else_label
end_label

It is here that information needs to be past from one instruction to the next
The CMP needs to tell the JCS whether or not there was a carry
It does this using flags

Each mathematical and logical instruction sets the flags. Each conditional jump and conditional return reads the flags and jumps or returns appropriately.

The Chimera-2016-I Microprocessor has the following flags...

- Carry Flag (C)
- Zero Flag (Z)
- Negative Flag (N)
- Interupt Flag (I)

7	6	5	4	3	2	1	0
Z	-	I	-	N	-	-	C

Mnemonic	Description	Flags
CLC	Clear Carry flag	- - - - - - - - 0
SEC	Set Carry flag	- - - - - - - - 1
CLI	Clear Interupt flag	- - 0 - - - - -
SEI	Set Interupt flag	- - 1 - - - - -
CMC	Compliment carry flag	- - - - - - - - -

Status Register Ops

Zero flag

The zero flag is set to 1 if the result of a mathematical or logical operation gives a result that equal zero, otherwise it is set to 0.

Carry flag

The carry flag is set to 1 if the result of an addition is greater than 8-bits or when a borrow is required during subtraction, otherwise it is set to zero.The carry flag is also used during rotation instructions.

Negative flag

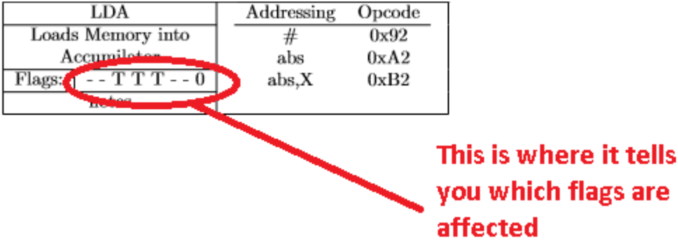
The negative flag is set to 1 if the most significant bit of the result is set to 1, otherwise it is set to 0.

Interrupt flag

The interrupt carry flag is used to enable/disable interrupts.

How do you know which opcodes set which flags?

Inside Chimera-2016-I documentation



LDA	Addressing	Opcode
Loads Memory into Accumulator	#	0x92
Flags: - - T T T - - 0	abs	0xA2
notes	abs,X	0xB2

Look for the file, Chimera-2016-I.pdf in more detail , on Blackboard. It contains a lot of information about each of the Chimera-2016-I instructions.

They tell you the following...

- What the instruction does
- The addressing mode used
- The flags that are modified

They even give you an example of the instruction in action...

Implementing the ADC instruction

ADC	
Register added to Accumulator with Carry	
Flags:	T - - - T - - T
notes	

Addressing	Opcode
A-B	0xB6
A-C	0xC6
A-L	0xD6
A-H	0xE6
A-M	0xF6

As we can see ADC first opcode is 0xB6

As always we need to add our new case to group_1

```
case 0xB6: // ADC A,B

break;
```

Remember that we are adding two bytes together... ...two 8-bit numbers. How big is the answer going to be?

It can be 9-bits!

As 9-bits do not fit into an 8-bit byte we can use 16-bit WORDs when we do the addition. You need to add the following code...

```
temp_word = (WORD)Registers[REGISTER_A] + (WORD)Registers[REGISTER_B];
```

Don't forget the carry

```
if ((Flags & FLAG_C) != 0)
{
    temp_word++;
}
```

Next we need to think of the Flags. Lets look at the Carry Flag. The Carry Flag gets set to 1 if the addition created a number larger that 8-bits (i.e. bit8 == 1). So we need to test for this so add...

```
if (temp_word >= 0x100)
{
    // Set carry flag
}
else
{
    // Clear carry flag
}
```

Do you remember how to set a single bit?

You use a bitwise OR.
Add...

Flags = Flags | FLAG_C;

Do you remember how to clear a single bit?

You use a bitwise AND.
Add...

Flags = Flags & (0xFF - FLAG_C);

Next we need to copy the result of the addition back into the Accumulator.
But we need to take into the account that the result of the addition is currently a 16-bit number.
Add...

```
Registers[REGISTER_A] = (BYTE)temp_word;
```

So far we have only dealt with the carry flag. Now it is time to deal with the others. The other flags are set based on the value of the result. A function has been provided to set the zero flag. Find...

```
void set_flag_z (BYTE inReg)
{
}
```

see if you can work out what its doing

Create...

```
void set_flag_n(BYTE inReg)
{
    BYTE reg;
    reg = inReg;
}
```

The Negative flag is set to 1 if the most significant bit (i.e. bit7) is set to 1, otherwise it is set to 0.
Add...

```
if ((reg & 0x80) != 0) // msbit set
{
    Flags = Flags | FLAG_N;
}
else
{
    Flags = Flags & (0xFF - FLAG_N);
}
```

```
case 0xB6:
param1 = Registers[REGISTER_A];
param2 = Registers[REGISTER_B];
temp_word = (WORD) param1 + (WORD)param2;
if ((Flags & FLAG_C) != 0){
    temp_word++;
}
if (temp_word >= 0x100){
    Flags = Flags | FLAG_C; // Set carry flag
}
else {
    Flags = Flags & (0xFF - FLAG_C); // Clear carry flag
```

```
}
set_flag_n((BYTE)temp_word);
set_flag_z((BYTE)temp_word);
Registers[REGISTER_A] = (BYTE)temp_word;
break;
```

Implementing the CMP instruction

The compare instruction is very similar to the add with carry instruction but for a few execeptions...

- It is subtract instead of addition
- The carry isn't added
- The data isn't written back!

```
case 0xBA:
param1 = Registers[REGISTER_A];
param2 = Registers[REGISTER_B];
temp_word = (WORD) param1 - (WORD)param2;
if (temp_word >= 0x100){
        Flags = Flags | FLAG_C; // Set carry flag
}
else {
        Flags = Flags & (0xFF - FLAG_C); // Clear carry flag
}
set_flag_n((BYTE)temp_word);
set_flag_z((BYTE)temp_word);
```

break;

Implementing the MSA instruction

MSA is a simile intruction that move the contents of the status register to the accumulator...

MSA impl 0x2D
Implied addressing means we don't need any additional data

Create...

```
case 0x2D: //MSA
        Registers[REGISTER_A] = Flags;
        break;
```

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

Don't forget to go back to last weeks instructions and update their flag setting

Now you should be able to implement
ADD, TAY, TYA, MSA, CLC, SEC, CLI, SEI, CMC,
on your own.