Chimera-2018-A 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 {
}
```

```
LDA
LD
CMP
JCS else_label
JP ende_label
else_label
end_label
```

It actually does something like this...

LDA
LD
CMP
JCS else_label
JP 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-2018-A Microprocessor has the following flags...

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

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

Mnemonic	Description	Flags
CLC	Clear Carry flag	- - - - - - - 0
SEC	Set Carry flag	- - - - - - - 1
CLI	Clear Interupt flag	- - - - 0 - - -
STI	Set Interupt flag	- - - - 1 - - -
SEV	Set Overflow flag	- 1 - - - - - -
CLV	Clear Overflow flag	- 0 - - - - - -

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.

Overflow flag

The overflow flag is set to 1 if there is an overflow or borrow on the most significant bit. The overflow flag is important when signed arithmetic is being used.

How do you know which opcodes set which flags?

Inside Chimera-2018-A documentation

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

This is where it tells you which flags are affected

Look for the file, Chimera-2018-A.pdf in more detail , on Blackboard. It contains a lot of information about each of the Chimera-2018-A 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 ADD instruction

ADD		Addressing	Opcode
Register added to Accumulator with Carry		A-B	0x10
Flags:	T T - - - - T T	A-C	0x11
notes		A-D	0x12
		A-E	0x13
		A-F	0x14

As we can see ADD first opcode is 0x10

As always we need to add our new case to group_1

case 0x10: // ADD 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);
}
```

Create...

```
void set_flag_v(BYTE in1, BYTE in2, BYTE out1)
{
    BYTE reg1in;
    BYTE reg2in;
    BYTE regOut;
    reg1in = in1;
    reg2in = in2;
    regOut = out1;
```

You need to add code for the Overflow Flag here (ask your PAL leaders for help, they have a copy of the code that you need)

```
}
```

```
case 0x10:
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);
set_flag_v(param1, param2, (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 execptions...

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

```
case 0x30:
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);
```

```
set_flag_v(param1, param2, (BYTE)temp_word);
break;
```

Implementing the CSA instruction

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

CSA impl 0xD5
Implied addressing means we don't need any additional data

Create...

```
case 0xD5: //CSA
        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 MAX, MXA, CSA, CLC, SEC, CLI, STI, SEV, CLV,

on your own.