Chimera-2016-I Emulator Assignment

Practical 1 - Loading and storing

CANS Tech INC

Your task is to complete writing an Chimera-2016-I Emulator. In particular you need to write the code that emulates the Chimera-2016-I instructions. You will find a description of the instructions on Blackboard.

The Registers

in the Chimera-2016-I Microprocessor

First, lets see what registers are in the Chimera-2016-I Microprocessor...

...Registers are pieces of fast memory

Register A	- 8 bit Acummilator
	-
Flags	- Flags (individual bits)
	-
ProgramCounter	- 16 bit Program Counter
	-
StackPointer	- 16 bit Stack Pointer

The Registers

Register X	Register Y	- Two 8 bit Index Registers

Register B	Register C	- 4 8 bit General Registers
Register L	Register H	

The Registers

A Accumulator is where data can be stored and where calculations can happen.

The Flags are where the status of operations are stored, for example if a subtraction gives zero as a result then a Zero Flag is set.

A Index Register is used to address (point at) locations in memory where data is stored.

A General Purpose Register is used to store data for calculations

The 16 bit registers SP and PC point to locations in memory.

The Stack Pointer (SP) we will look at another time.

The Program Counter (PC) points to the next instruction (or part of) that is to be executed. As each instruction in memory is read and executed the PC is incremented.

The Addressing Modes

of the Chimera-2016-I Microprocessor

IMMEDIATE ADDRESSING (#)

The operand is the second byte for 8 bit instructions or the second byte for the lower byte and third byte for the higher byte represent the data for given instruction, no memory addressing is required.

IMPLIED ADDRESSING(impl)

A single byte instruction in which all of the data and operands are implied through the instruction itself.

ABSOLUTE ADDRESSING(abs)

In absolute addressing the second byte of an instruction represents the low order byte of an effective address. The third byte represents the high order byte of an effective address. The two bytes are added to allow full access to 65K of memory.

INDEXED ABSOLUTE ADDRESSING(abs,X)

In indexed absolute addressing the second byte and third byte of an instruction are used in conjunction with a index register (Register X or Register Y or Register XY). the second byte of the instruction represents the low order byte of an effective address. The third byte represents the high high byte of an effective address.

The result is added to the index register giving a result anywhere in memory. Any 16 bit carry is discarded.

ZERO PAGE ADDRESSING(zpg)

In zero page addressing the second byte of a instruction represents the low order byte of an effective address. The high order byte is fixed at 0 giving you access to the first 256 memory locations.

OFFSET ADDRESSING(rel)

In offset addressing, the second instruction byte of the instruction is used as a offset in conjunction with the programcounter. The offset is calculated by using the given byte as signed, resulting in -128 to +127. This offset is added to the contents of the programcounter giving the effective address within -128 to +127.

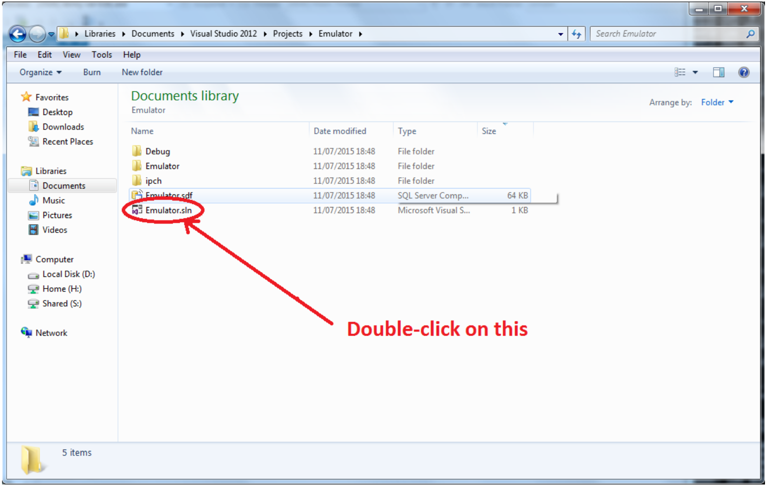
REGISTER ADDRESSING

In Register addressing the name of the desintation register (and the source where applicable) is stated in the instruction needing no addition bytes.

The Chimera-2016-I

Emulator code

Copy the Emulator.zip file from Blackboard and unzip it.
Copy the Emulator directory onto your home drive if it isn't already there.

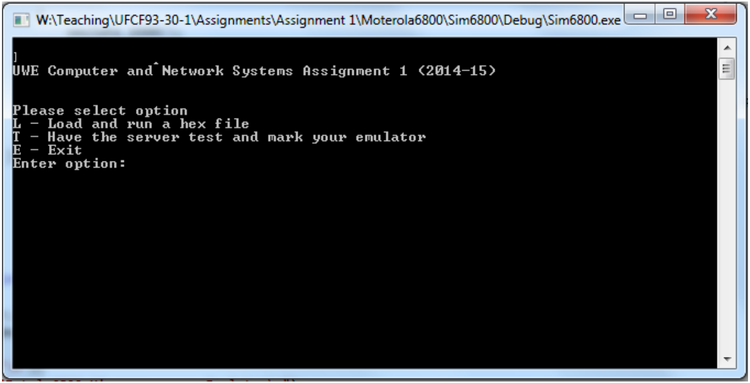

Libraries ▸ Documents ▸ Visual Studio 2012 ▸ Projects ▸ Emulator ▸
Search Emulator
File Edit View Tools Help
Organize ▾ Burn New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (D:)
Home (H:)
Shared (S:)
Network
Documents library
Emulator
Arrange by: Folder ▾
Name Date modified Type Size
Debug 11/07/2015 18:48 File folder
Emulator 11/07/2015 18:48 File folder
ipch 11/07/2015 18:48 File folder
Emulator.sdf 11/07/2015 18:48 SQL Server Comp... 64 KB
Emulator.sln 11/07/2015 18:48 Microsoft Visual S... 1 KB
Double-click on this
5 items

Eventually Visual Studio 2012 will open...

```
#include "stdafx.h"
#include <winsock2.h>

#pragma comment(lib, "wsock32.lib")


#define STUDENT_NUMBER     "12345678"

#define IP_ADDRESS_SERVER "127.0.0.1"

#define PORT_SERVER 0x1984 // We define a port that we are going to use.
#define PORT_CLIENT 0x1985 // We define a port that we are going to use.

#define WORD  unsigned short
#define DWORD unsigned long
#define BYTE  unsigned char

#define MAX_FILENAME_SIZE 500
#define MAX_BUFFER_SIZE   500

SOCKADDR_IN server_addr;
SOCKADDR_IN client_addr;
```

Replace with your student number

Build the project using the menu options at the top of the Visual Studio screen. Once it has built ok run it...



The option:

L - lets you load a .hex file and execute it (they can be found on Blackboard)
T - lets you obtain your current marks for implementing the 6800 instructions
E - lets you exit your program

Type E for now...

The code that you do need to change resides in the function...

void Group_1(BYTE opcode)

void Group_2_move(BYTE opcode)

These may or may not have some useful code already in there.

Implementing the LD Instruction

LD loads Accumulator A with the contents of an address in memory

6 different addressing modes are used with LD

- #
- abs
- abs,X
- abs,Y
- abs,XY
- zpg

LD

```
void Group_1(BYTE opcode) {
    BYTE HB = 0,LB = 0;
    WORD address = 0,data = 0;
    switch (opcode) {
        case 0x53: // LD immediate
            CODE ADDED HERE
            break;
    }
}
```

LD (#) - Hex: 0x43

data = fetch();
Registers[REGISTER_A] = data;

LD (abs) - Hex: 0x53

```
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Registers[REGISTER_A] = Memory[address];
}
```

LD (abs,X) - Hex: 0x63

```
address += Index_Registers[REGISTER_X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Registers[REGISTER_A] = Memory[address];
}
```

LD (abs,Y) - Hex: 0x73

```
address += Index_Registers[REGISTER_Y];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Registers[REGISTER_A] = Memory[address];
}
```

LD (abs,XY) - Hex: 0x83

```
address += (WORD)((WORD)Index_Registers[REGISTER_Y] « 8)
+ Index_Registers[REGISTER_X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Registers[REGISTER_A] = Memory[address];
}
```

LD (zpg) - Hex: 0x93

```
address += 0x0000 | (WORD)fetch();
if(address >= 0 && address < MEMORY_SIZE) {
    Registers[REGISTER_A] = Memory[address];
}
```

Implementing the STO Instruction

STO stores the contents of the Accumulator to memory

5 different addressing modes are used with STO

- abs
- abs,X
- abs,Y
- abs,XY
- zpg

STO (abs) - Hex: 0x04

```
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB << 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Memory[address] = Registers[REGISTER_A];
}
```

STO (abs,X) - Hex: 0x14

```
address += Index_Registers[REGISTER_X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB << 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Memory[address] = Registers[REGISTER_A];
}
```

STO (abs,Y) - Hex: 0x24

```
address += Index_Registers[REGISTER_Y];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Memory[address] = Registers[REGISTER_A];
}
```

STO (abs,XY) - Hex: 0x34

```
address += (WORD)((WORD)Index_Registers[REGISTER_Y] « 8)
+ Index_Registers[REGISTER_X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE) {
    Memory[address] = Registers[REGISTER_A];
}
```

STO (zpg) - Hex: 0x44

```
address += 0x0000 | (WORD)fetch();
if(address >= 0 && address < MEMORY_SIZE) {
	Memory[address] = Registers[REGISTER_A];
}
```

Implementing the MVR Instruction for Register B

MVR loads a general purpose register with the contents of an address in memory

1 different addressing modes are used with MVR
- #

MVR (#) - Hex: 0x27

data = fetch();
Registers[REGISTER_B] = data;

There is a MVR for every one of the General purpose registers...

...The rest you will have to do in your own time

Implementing the MV Instruction for Register A

First lets look at the MV block

0xA0 A, A	0xA1 B, A	0xA2 C, A	0xA3 L, A	0xA4 H, A	0xA5 M, A	
0xB0 A, B	0xB1 B, B	0xB2 C, B	0xB3 L, B	0xB4 H, B	0xB5 M, B	
0xC0 A, C	0xC1 B, C	0xC2 C, C	0xC3 L, C	0xC4 H, C	0xC5 M, C	
0xD0 A, L	0xD1 B, L	0xD2 C, L	0xD3 L, L	0xD4 H, L	0xD5 M, L	
0xE0 A, H	0xE1 B, H	0xE2 C, H	0xE3 L, H	0xE4 H, H	0xE5 M, H	
0xF0 A, M	0xF1 B, M	0xF2 C, M	0xF3 L, M	0xF4 H, M	0xF5	

Hex
destination, source

The source and destination are both offsets of the first MV instruction

Group_2_Move(BYTE opcode)

Find this function, your MV code will go in here

Inside the function you will need two variables fo the source and destination.

BYTE destination = opcode »4;
BYTE source = opcode & 0x0F;

You will also need two temperay variables for the registers

int destReg;
int sourceReg;

Now create a switch like the one that was in Group_1

```
switch(dest) {
    case 0x00:
        destReg = REGISTER_A;
        break;
    case 0x01:
        destReg = REGISTER_B;
        break;
.......
```

You will need something similar for source aswell

Once we have the source and destination registers we can complete the operation by making the destination equal the source:

Registers[destReg] = Registers[sourceReg];

But what if the source or desintation is REGISTER_M? if you remember back the the register list there is no register M

REGISTER_M is actually access to and from memory much like LD and STO but in a different way.
REGISTER_M refers to moving a registers contents to and from memory using the contents ofregisters L and H as the address
Register H is the high byte of the address and register L is the low byte, for example:

H contains 0x03 and L contains 0xC9
This would form the address 0x03C9

```
if(destReg == REGISTER_M){
    address = Registers[REGISTER_L];
    address += (WORD)Registers[REGISTER_H] « 4;
    if(address >= 0 && address <= MEMORY_SIZE){
        Memory[address] = Register[sourceReg];
    }
} else {
    Register[destReg] = Register[sourceReg];
}
```

You will need something similar for source aswell

Implementing the LODS Instruction

LODS loads StackPointer with the contents of an address in memory

6 different addressing modes are used with LODS

- #
- abs
- abs,X
- abs,Y
- abs,XY
- zpg

LODS (#) - Hex: 0x20

data = fetch();
StackPointer = data ; StackPointer += (WORD)fetch() « 8;

LODS (abs) - Hex: 0x30

```
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB << 8) + LB;
if(address >= 0 && address < MEMORY_SIZE - 1) {
	StackPointer = Memory[address];
	StackPointer += (WORD)Memory[address + 1] << 8;
}
```

LODS (abs,X) - Hex: 0x40

```
address += Index_Registers[REGISTER_X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB << 8) + LB;
if(address >= 0 && address < MEMORY_SIZE - 1) {
    StackPointer = Memory[address];
    StackPointer += (WORD)Memory[address + 1] << 8;
}
```

LODS (abs,Y) - Hex: 0x50

```
address += Index_Registers[REGISTER_Y];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE - 1) {
	StackPointer = Memory[address];
	StackPointer += (WORD)Memory[address + 1] « 8;
}
```

LODS (abs,XY) - Hex: 0x60

```
address += (WORD)((WORD)Index_Registers[REGISTER_Y] « 8) + Index_Registers[REG
X];
LB = fetch();
HB = fetch();
address += (WORD)((WORD)HB « 8) + LB;
if(address >= 0 && address < MEMORY_SIZE - 1) {
        StackPointer = Memory[address];
        StackPointer += (WORD)Memory[address + 1] « 8;
}
```

LODS (zpg) - Hex: 0x70

```
address += 0x0000 | (WORD)fetch();
if(address >= 0 && address < MEMORY_SIZE - 1) {
	StackPointer = Memory[address];
	StackPointer += (WORD)Memory[address + 1] « 8;
}
```

Don't forget to compile and run your program to check that it runs and see how many marks you have earned!

To be finished by next week LD, STO, MV LDX STOX LODY STOY TAY TYA LODS STOS MSA MVR instructions

You now must do this in your own time...

...and you must complete it before your next practical in one weeks time or you will fall behind!