# CET2113C – Advanced Digital Circuits

## Lab 5 – Introduction to Machine Language and Instruction Sets

### Objective

The objective of this lab is to understand General Purpose Processors and their Instruction Sets. You will be given the source code for an Instruction Set Simulator that simulates a basic set of instructions in a fictional processor that we call MDC1. You will be required to modify the source code, extending its instruction set to include additional instructions.

### Introduction

The instruction set of a processor describes the bit configurations allowed in the Instruction Register, indicating the atomic (or built-in) processor operations that the programmer may invoke. Each of these configurations forms an assembly instruction, and a sequence of these assembly instructions forms an assembly program.

Assembly instructions typically have two parts, an opcode (which stands for operations code) and operand fields. An opcode specifies the operation to take place during the instruction. Instructions are classified into three categories: Data transfer, and Arithmetic/Logical, and Branch instructions.

An Instruction Set Simulator is a program that simulates the execution of assembly instructions via a program that is compiled and runs in our computers. ISS.c is a c program that implements a very simple instruction set simulator. The instructions that have been implemented are listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Op-Code | 1st Operand | 2nd Operand |
| MOV Rn, direct | 0000 | Rn | direct |
| MOV direct,Rn | 0001 | Rn | direct |
| MOV @Rn, Rm | 0010 | Rn | Rm \_\_\_\_ |
| MOV Rn, #immed. | 0011 | Rn | immed. |
| ADD Rn, Rm | 0100 | Rn | Rm \_\_\_\_ |
| SUB Rn, Rm | 0101 | Rn | Rm \_\_\_\_ |
| JZ Rn, relative | 0110 | Rn | relative |

The ISS has 2048 bytes of program memory, 256 bytes of data memory and 16 registers. The instructions are loaded into memory in the main function as two part, two digit hex numbers that represent the complete instruction in binary. The main program then calls the simulator function to execute the program. This simulator function implements the instruction fetch cycle for a typical processor.

The program that is loaded into the main function is shown below. It implements a typical C loop, which is also shown below.

|  |  |
| --- | --- |
| //C Code  int total = 0;  for (int i=10; i!=0; i--)  total += i;  // next instructions... | // Equivalent Assembly Program  MOV RO, #0  MOV R1, #10  MOV R2, #1  MOV R3, #0  Loop: JZ R1, Next  ADD R0, R1  SUB R1, R2  JZ R3, Loop  Next: //next instruction |

### Procedure

1. Decode the existing assembly program into their respective binary as well as hex code.
2. Add the following instructions to the ISS. The INC instruction adds one to the operand (using register indirect, register direct, or direct addressing modes)
   1. INC @Rn // Register Indirect
   2. INC Rn // Register Direct
   3. INC direct // Direct
   4. DEC @Rn // Register Indirect
   5. DEC Rn // Register Direct
   6. DEC direct // Direct
3. Rewrite the above Assembly program incorporating these new instructions.
4. Decode the assembly program and test it in the instruction set simulator.
5. Write an assembly program to compute the Fibonacci sequence for the first 10 values of the sequence and store these values into the data memory beginning with the first memory location.
6. Decode the assembly program and test it in the instruction set simulator.

### Lab Report

For your lab report, please use the template provided. You must explain, in your own words, how the design of ISS, the improvements made by adding instructions, as well as the two assembly programs and binary decoding. Please include the contents of the data and program memory for each of the assembly programs.

//C code for the ISS

/\* Instruction Set Simulator

\* Key

\* Assembly Inst. First Byte Second Byte

\*

\* MOV Rn, direct 0000 Rn direct

\* MOV direct,Rn 0001 Rn direct

\* MOV @Rn, Rm 0010 Rn Rm \_\_\_\_

\* MOV Rn, #immed. 0011 Rn immed.

\* ADD Rn, Rm 0100 Rn Rm \_\_\_\_

\* SUB Rn, Rm 0101 Rn Rm \_\_\_\_

\* JZ Rn, relative 0110 Rn relative

\*

\*/

**#include** <stdio.h>

**typedef** **struct** {

**unsigned** **char** first\_byte, second\_byte;

} instruction;

instruction program[1024]; //instruction memory

**unsigned** **char** memory[256], reg[16]; //data memory

**int** **run\_program**(**int** num\_bytes) {

**int** pc = -1;

**unsigned** **char** fb, sb;

**while**( ++pc < (num\_bytes / 2) ) {

fb = program[pc].first\_byte;

sb = program[pc].second\_byte;

**switch**( fb >> 4 ) { // bit wise shift by 4 in the while loop

**case** 0: // MOV Rn, direct

reg[fb & 0x0f] = memory[sb];

**break**;

**case** 1: // MOV direct, Rn

memory[sb] = reg[fb & 0x0f];

**break**;

**case** 2: // MOV @Rn, Rm

memory[reg[fb & 0x0f]] = reg[sb >> 4];

**break**;

**case** 3: // MOV Rn, #immed.

reg[fb & 0x0f] = sb;

**break**;

**case** 4: // ADD Rn, Rm

reg[fb & 0x0f] += reg[sb >> 4];

**break**;

**case** 5: // SUB Rn, Rm

reg[fb & 0x0f] -= reg[sb >> 4];

**break**;

**case** 6: // JZ Rn, relative

**if** (reg[fb & 0x0f] == 0) {

pc = (0x0f & sb) - 1;

}

**else**

pc = pc;

**break**;

**default**: **return** -1;

}

}

**return** 0;

}

**void** **print\_memory\_contents**(){

**printf**("Printing memory\n");

**int** i;

**for**(i=0;i<256;i++){

**printf**("%i\t",memory[i]);

**if** ((i+1)%8 == 0){

**printf**("\n");

}

}

**return**;

}

**void** **print\_register\_contents**(){

**printf**("Printing registers\n");

**int** i;

**for**(i=0;i<16;i++){

**printf**("R%i : %i\n",i,reg[i]);

}

**return**;

}

**void** **initialize\_processor**(){

**int** i;

**for**(i=0;i<1024;i++){

program[i].first\_byte = 0x00;

program[i].second\_byte = 0x00;

}

**for**(i=0;i<256;i++){

memory[i] = 0x00;

}

**for**(i=0;i<16;i++){

reg[i] = 0x00;

}

}

**int** **main**() {

// initializes the data and program memory, and the registers

initialize\_processor();

// load program into program memory

// Instruction Binary Hex

// MOV R0, #0 0010000000000000 3000

program[0].first\_byte = 0x30;

program[0].second\_byte = 0x00;

program[1].first\_byte = 0x31;

program[1].second\_byte = 0x0A;

program[2].first\_byte = 0x32;

program[2].second\_byte = 0x01;

program[3].first\_byte = 0x33;

program[3].second\_byte = 0x00;

program[4].first\_byte = 0x61;

program[4].second\_byte = 0x08;

program[5].first\_byte = 0x40;

program[5].second\_byte = 0x10;

program[6].first\_byte = 0x51;

program[6].second\_byte = 0x20;

program[7].first\_byte = 0x63;

program[7].second\_byte = 0x04;

program[8].first\_byte = 0x38;

program[8].second\_byte = 0xff;

run\_program(18);

print\_memory\_contents();

print\_register\_contents();

**return** 0;

}