**Name**

**Advanced Programming in C++**

**Lab Exercise 4/5/2021**

**Simulating a Simpletron Microprocessor**

Let us create a computer we will call the Simpletron. As it’s name implies, it is a simple machine. The Simpletron runs programs in the only language it understands, that is, Simpletron Machine Language (SML).

The Simpletron contains an accumulator – a special register in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number such as +3364, -1293, +0007, -0001, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, 02, … 99.

Before running an SML program, we must load or place the program into memory. The first instruction of every SML program is always placed in location 00.

Each instruction written in SML occupies one word of the Simpletron’s memory (i.e. are signed four-digit decimal numbers). We shall assume that the sign of an SML instruction is always plus, but the sign of data word may be either plus or minus. Each location in the Simpletron’s memory may contain either an instruction, a data value used by the program, or an unused area of memory. The first two digits of each SML instruction are the operation code, which specifies the operation to be performed.

Operation Code Meaning

Input/Output operations

#define READ 10 Read a word from terminal into specific location

#define WRITE 11 Write a word from a specific location in memory to terminal

Load/Store operations

#define LOAD 20 Load a word from a specific location in memory into accumulator

#define STORE 21 Store a word from the accumulator to a specific location in memory

Arithmetic operations

#define ADD 30 Add a word from a specific memory location to the accumulator

#define SUBTRACT 31 Subtract a word from a specific memory location from the accumulator

#define DIVIDE 32 Divide a word from a specific memory location into the accumulator

#define MULTIPLY 33 Multiply a word from a specific memory location by the accumulator

Note: results are left in the accumulator

Transfer of control operations

#define BRANCH 40 Branch to a specific memory location

#define BRANCHNEG 41 Branch to a specific memory location if accumulator negative

#define BRANCHZERO 42 Branch to a specific memory location if accumulator zero

#define HALT 43 Halt the program

Example 1

Location Number Instruction

00 +1007 Read A

01 +1008 Read B

02 +2007 Load A

03 +3008 Add B

04 +2109 Store C

05 +1109 Write C

06 +4300 Halt

07 +0000 Variable A

08 +0000 Variable B

09 +0000 Result C

The SML program reads two numbers from the keyboard and computes and prints their sum. The instruction +1007 reads the first number from the keyboard and stores it in location 07 (initialized to 0000). Then +1008 reads the next number into location 08. The load instruction +2007, puts the first number in the accumulator. And the add instruction +3008 adds the second number to the number in the accumulator. The store instruction +2109, places the value in the accumulator in memory location 09. The write instruction +1109 takes the number in memory location 09 and sends it to the screen. The halt instruction +4300

Example 2

Location Number Instruction

00 +1009 Read A

01 +1010 Read B

02 +2009 Load A

03 +3110 Subtract B

04 +4107 Branch negative to 07

05 +1109 Write A

06 +4300 Halt

07 +1110 Write B

08 +4300 Halt

09 +0000 Variable A

10 +0000 Variable B

This SML program reads two numbers from the keyboard and prints the larger value. Note the use of the instruction +4107 is a conditional transfer of control much the same as C++’s if statement.

Now that we know the SML language let us write SML programs to perform the following tasks:

1. Use a sentinel controlled loop to read positive numbers and compute and print their sum
2. Use a counter controlled loop to read seven numbers, some positive and some negative, compute their sum and print their average
3. Read a series of numbers and determine the largest number. The first number read indicates how many numbers should be processed

Now that we have a fundamental understanding of SML, let us write a program in C that will simulate the operation of the Simpletron. Your Simpletron simulator will turn your computer into a Simpletron. You will then be able to actually run, test, and debug SML programs. When you run you Simpletron Simulator it should begin printing

\*\*\* Welcome to Simpletron \*\*\*

\*\*\* Please enter your program one instruction \*\*\*

\*\*\* (or data word) at a time. I will type the \*\*\*

\*\*\* memory location number and a question \*\*\*

\*\*\* mark (?). You then type the word to be \*\*\*

\*\*\* in that location. Type the sentinel –9999 \*\*\*

\*\*\* to stop entering your program \*\*\*

Simulate the memory of the Simpletron with a one-dimensional array called **Memory** that has 100 elements. If the simulator is running, the dialog of example 2 would look as such:

00 ? +1009

01 ? +1010

02 ? +2009

03 ? +3110

04 ? +4107

05 ? +1109

06 ? +4300

07 ? +1110

08 ? +4300

09 ? +0000

10 ? +0000

11 ? -9999

\*\*\* Program loading completed \*\*\*

\*\*\* Program execution begins \*\*\*

The SML program has been loaded into the array **Memory**. Now the Simpletron executes your SML program. Execution begins with the instruction in location 00, and like C continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable **Accumulator** to represent the accumulator register. Use the variable **InstructionCounter** to keep track of the location in memory that contains the instruction being performed. Use the variable **OperationCode** to indicate the operation currently being performed, i.e. the left two digits of the instruction word. Use the variable **Operand** to indicate the memory location to which the current instruction operates. Thus **Operand** is the rightmost two digits of the instruction currently being performed. In microprocessor terminology, this is called direct addressing. You will bring the instruction from memory into the **InstructionRegister**, then “pick off” the left two digits and place them in **OperationCode** and “pick off” the right two digits and place them in **Operand**. When the Simpletron begins execution, the special registers are initialized as follows:

# Accumulator +0000

InstructionCounter 00

InstructionRegister +0000

OperationCode 00

Operand 00

Now let us walk through the execution of the first SML instruction from

example 2, +1009 in memory location 00. This is called the instruction execution cycle.

The variable InstructionCounter tells us the memory location of the next instruction to be performed. We fetch the contents of that location from Memory by using the C statement

InstructionRegister = Memory[InstructionCounter];

The operation code and the operand are extracted from the instruction register by the statements

OperationCode = InstructionRegister/100;

Operand = InstructionRegister %100;

Now the Simpletron must now determine if the operation code is a read, write load etc. A switch can be used to differentiate between the twelve operations in SML. In the switch structure, the behavior of various SML instructions is simulated as follows

read: cin >> Memory[Operand];

load: Accumulator = Memory[Operand];

add: Accumulator += Memory[Operand];

*Other instructions are left to you*

halt: cout << “\*\*\* Simpletron Execution Terminated \*\*\*”;

Once program execution is terminated, you will want to “dump” the contents of all memory locations as well as the contents of each register to the screen. Note that a dump after executing a Simpletron program will show the actual values of instructions and data values at the moment of execution termination.

## Sample Screen Dump

# REGISTERS

Accumulator +0000

InstructionCounter 00

InstructionRegister +0000

OperationCode 00

Operand 00

MEMORY:

0 1 2 3 4 5 6 7 8 9

1. +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

1. +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

## Program Control

After each instruction is executed, we must increment the InstructionCounter as such

prepare the Simpletron to execute the next instruction. This is done with an increment as such

InstructionCounter++;

which will now make InstructionCounter hold the Memory “address” of the next instruction. This will work fine as long as we are executing sequential instructions. The question is what happens when we encounter a branch instruction? If a branch instruction occurs, we must set the InstructionCounter to the value of the Operand as such

InstructionCounter = Operand;

For example, the instruction BRANCHZERO would be simulated as such

if (Accumulator == 0)

InstructionCounter = Operand;

## Error and Exception Handling

There are various errors that could be encountered during program execution. During program loading, each number stored in the Simpletron’s memory must fall within the range of –9999 to +9999. You could use a while loop to test as each number is entered to make sure it is a valid instruction word. During execution of the program, several problems can occur such as:

Attempts to divide by 0

Invalid operation codes

Accumulator overflows

Serious errors such as this are called fatal errors. When a fatal error is detected your simulator should print an error message such as:

\*\*\* Attempt to divide by zero \*\*\*

\*\*\* Simpletron execution abnormally terminated \*\*\*

and print a complete register and memory dump as previously discussed. Once you have your simulation completed, test run the SML language you wrote earlier. As an improvement to your program, modify the program to bring the program into your array **Memory** from a disk file. This would more closely simulate actual program execution.

Your task for today is to build a Simpletron class. Your class should be defined in a file called “Simpletron.h” and implemented in a file called “Simpletron.cpp”. In your class, you are to have the following global defines:

#define READ 10

#define WRITE 11

#define LOAD 20

#define STORE 21

#define ADD 30

#define SUBTRACT 31

#define DIVIDE 32

#define MULTIPLY 33

#define BRANCH 40

#define BRANCHNEG 41

#define BRANCHZERO 42

#define HALT 43

You should have the following private members:

int Memory[100];

int Accumulator;

int InstructionCounter;

int InstructionRegister;

int OperationCode;

int Operand;

You should have the following private member functions:

void Greeting(void);

void InitializeMemory(void);

void InitializeRegisters(void);

void LoadProgram(void);

You should have the following public member functions:

void Execute(void);

void DumpRegisters(void);

void DumpMemory(void);

The constructor function should call Greeting( ), InitializeMemory( ), InitializeRegisters( ), and LoadProgram( ).

Your main function should test the class as follows (be sure to #include “Simpletron.h”:

int main()

{

Simpletron test;

test.Execute();

test.DumpRegisters();

test.DumpMemory();

return 0;

}

Here are some of your function definitions:

Simpletron::Simpletron()

{

Greeting();

InitializeRegisters();

InitializeMemory();

LoadProgram();

}

void Simpletron::Greeting(void)

{

cout << "Welcome to the Simpletron\nLow Performance Microprocessor” << endl;

cout << "Hit any key to commence program execution\n\n\n”;

getch(); // need to include <conio.h>

}

void Simpletron::InitializeMemory(void)

{

int i;

for (i=0; i<100; i++)

Memory[i] = 0;

}

void Simpletron::InitializeRegisters(void)

{

Accumulator=0;

InstructionCounter=0;

InstructionRegister=0;

OperationCode=0;

Operand=0;

}

void Simpletron::LoadProgram(void)

{

ifstream infile(“program.txt”); // be sure program.txt is stored in the project folder

int instruction;

int i;

instruction = 1;

i = 0;

while (instruction != 0)

{

infile >> instruction;

Memory[i] = instruction;

i++;

}

}

void Simpletron::Execute(void)

{

do

{

InstructionRegister = Memory[InstructionCounter];

OperationCode = InstructionRegister / 100;

Operand = InstructionRegister % 100;

switch (OperationCode)

{

case READ: printf("Enter a number: ");

cin >> Memory[Operand];

InstructionCounter++;

break;

case WRITE: cout << "Result: " << Memory[Operand] << endl;

InstructionCounter++;

break;

case LOAD: Accumulator = Memory[Operand];

InstructionCounter++;

break;

case STORE: Memory[Operand] = Accumulator;

InstructionCounter++;

break;

case ADD: Accumulator += Memory[Operand];

InstructionCounter++;

break;

case SUBTRACT: Accumulator -= Memory[Operand];

InstructionCounter++;

break;

case DIVIDE: if (Memory[Operand] == 0)

{

cout <<"Attempt to divide by 0\n");

cout << "Simpletron execution abnormally terminated\n";

OperationCode = 43;

break;

}

else

{

Accumulator /= Memory[Operand];

InstructionCounter++;

break;

}

case MULTIPLY: Accumulator \*= Memory[Operand];

InstructionCounter++;

break;

case BRANCH: InstructionCounter = Operand;

break;

case BRANCHNEG: if (Accumulator < 0)

InstructionCounter = Operand;

else

InstructionCounter++;

break;

case BRANCHZERO: if (Accumulator == 0)

InstructionCounter = Operand;

else

InstructionCounter++;

break;

case HALT: cout << "\n\n\*\*\*Simpletron Execution Terminated\*\*\*\n\n\n";

break;

} /\*End of switch\*/

}while (OperationCode != HALT);

}

void Simpletron::DumpRegisters(void)

{

cout << "\n\nREGISTERS\n";

cout << "Accumulator\t\t\t" << Accumulator << endl;

cout << "Instruction Counter\t\t" << InstructionCounter << endl;

cout << "Instruction Register\t\t" << InstructionRegister << endl;

cout << "Operation Code\t\t\t" << OperationCode << endl);

cout << "Operand\t\t\t\t" << Operand << endl);

}

void Simpletron::DumpMemory(void)

{

int i;

cout << "\n\nMEMORY\n";

for (i = 1; i <= 100; i++)

{

cout << Memory[i-1] << " \t";

if (i % 10 == 0)

cout << endl;

}

}

Here is a program to test your code:

1007

1008

2007

3008

2109

1109

4300

0

You should place it in a text file and store it as “program.txt” and store it in your project folder.

**Programming Assignment**

Use any text editor to write and save these programs. The program needs to be stored in the same folder as main.cpp. You will need to edit the LoadProgram function to reference the correct filename. The original program code is stored in a file called program.txt. At the end of this document you will find potential solutions for these 2 programs (program1.txt and program2.txt). Note: be careful when saving a text document that it does not add an extra .txt at the end (program1.txt.txt) by saving as All Files (\*.\*) in the Save File dialog box.

1. Write a program in SML that will print the numbers from 1 to 10 on the screen.
2. Write a program that will allow the user to enter two numbers and displays the product of those two numbers.

**Note: The pages that follow are source code from a working project.// Simpletron Project.cpp**

#include "Simpletron.h"

int main()

{

Simpletron test;

test.Execute();

test.DumpRegisters();

test.DumpMemory();

return 0;

}

**//Simpletron.h**

//Author: Mr. Messa

//Date: April 24, 2008

//Class Definition file for the Simpletron Class

//This class represents an abstraction of a microprocessor

#ifndef SIMPLETRON\_H

#define SIMPLETRON\_H

#define READ 10

#define WRITE 11

#define LOAD 20

#define STORE 21

#define ADD 30

#define SUBTRACT 31

#define DIVIDE 32

#define MULTIPLY 33

#define BRANCH 40

#define BRANCHNEG 41

#define BRANCHZERO 42

#define HALT 43

class Simpletron

{

private:

//Member variables

int Memory[100];

int Accumulator;

int InstructionCounter;

int InstructionRegister;

int OperationCode;

int Operand;

//Member functions (private)

void Greeting(void);

void InitializeMemory(void);

void InitializeRegisters(void);

void LoadProgram(void);

public:

//Member functions (public)

Simpletron();

void Execute(void);

void DumpRegisters(void);

void DumpMemory(void);

};

#endif

**//Simpletron.cpp**

#include "Simpletron.h"

#include <iostream>

#include <fstream>

#include <conio.h>

using namespace std;

Simpletron::Simpletron()

{

Greeting();

InitializeRegisters();

InitializeMemory();

LoadProgram();

}

void Simpletron::Greeting(void)

{

cout << "Welcome to the Simpletron\nLow Performance Microprocessor" << endl;

cout << "Hit any key to commence program execution\n\n\n";

getch(); // need to include <conio.h>

}

void Simpletron::InitializeMemory(void)

{

int i;

for (i=0; i<100; i++)

Memory[i] = 0;

}

void Simpletron::InitializeRegisters(void)

{

Accumulator=0;

InstructionCounter=0;

InstructionRegister=0;

OperationCode=0;

Operand=0;

}

void Simpletron::LoadProgram(void)

{

ifstream infile("program.txt"); // be sure program.txt is stored in the project folder

int instruction;

int i;

instruction = 1;

i = 0;

while (instruction != 0)

{

infile >> instruction;

Memory[i] = instruction;

i++;

}

}

void Simpletron::Execute(void)

{

do

{

InstructionRegister = Memory[InstructionCounter];

OperationCode = InstructionRegister / 100;

Operand = InstructionRegister % 100;

switch (OperationCode)

{

case READ: cout << "Enter a number: ";

cin >> Memory[Operand];

InstructionCounter++;

break;

case WRITE: cout << "Result: " << Memory[Operand] << endl;

InstructionCounter++;

break;

case LOAD: Accumulator = Memory[Operand];

InstructionCounter++;

break;

case STORE: Memory[Operand] = Accumulator;

InstructionCounter++;

break;

case ADD: Accumulator += Memory[Operand];

InstructionCounter++;

break;

case SUBTRACT: Accumulator -= Memory[Operand];

InstructionCounter++;

break;

case DIVIDE: if (Memory[Operand] == 0)

{

cout <<"Attempt to divide by 0\n";

cout << "Simpletron execution abnormally terminated\n";

OperationCode = 43;

break;

}

else

{

Accumulator /= Memory[Operand];

InstructionCounter++;

break;

}

case MULTIPLY: Accumulator \*= Memory[Operand];

InstructionCounter++;

break;

case BRANCH: InstructionCounter = Operand;

break;

case BRANCHNEG: if (Accumulator < 0)

InstructionCounter = Operand;

else

InstructionCounter++;

break;

case BRANCHZERO: if (Accumulator == 0)

InstructionCounter = Operand;

else

InstructionCounter++;

break;

case HALT: cout << "\n\n\*\*\*Simpletron Execution Terminated\*\*\*\n\n\n";

break;

} /\*End of switch\*/

}while (OperationCode != HALT);

}

void Simpletron::DumpRegisters(void)

{

cout << "\n\nREGISTERS\n";

cout << "Accumulator\t\t\t" << Accumulator << endl;

cout << "Instruction Counter\t\t" << InstructionCounter << endl;

cout << "Instruction Register\t\t" << InstructionRegister << endl;

cout << "Operation Code\t\t\t" << OperationCode << endl;

cout << "Operand\t\t\t\t" << Operand << endl;

}

void Simpletron::DumpMemory(void)

{

int i;

cout << "\n\nMEMORY\n";

for (i = 1; i <= 100; i++)

{

cout << Memory[i-1] << "\t";

if (i % 10 == 0)

cout << endl;

}

}

//program1.txt

2030

1130

3030

2131

1131

3030

2132

1132

3030

2133

1133

3030

2134

1134

3030

2135

1135

3030

2136

1136

3030

2137

1137

3030

2138

1138

3030

2139

1139

4300

1

0

program2.txt

1007

1008

2007

3308

2109

1109

4300

0