Joshua Mularczyk, Ethan Stong, Christian Williams

CPTR 380

Final Paper

15 March 2022

Histogram Equalization Routine

For this project, we decided to design a custom 8-bit architecture CPU (Central Processing Unit) with a Histogram Equalization Routine in mind to spark our ideas. Our architecture aimed for a simple 16X16 greyscale image with up to 256 shades. The architecture has 16 8-bit registers and is a store/load instruction set (no Move instruction). There are 16 instructions that run on single cycle design (no pipeline). As the architecture is simple, we are assuming that the programmer made no errors so there is no error detection or exceptions. Max Program Size is 256 instructions (Trying to figure out how to increase).

**Project tasks**

Created a custom-designed instruction set

Create assembler

Design a CPU to execute instruction set

Implement this hardware by describing it in C++

**Specifications**

* 8 Bit architecture
* Instructions will be 16 bits wide
* 16 OP codes(4 bits)
  + That way we can use 8 bits for our address and then the rest of the bits for opcode etc.
  + Not all OP codes get used
* Address up to 256 bytes of memory
* Registers will be 8 bits
* No direct operations on memory (do a lw then and add, not same step)
* Only Loading from memory. Any other load needs to occur by using an add instruction

**Instruction Format**

* First byte is opcode (first 4 bits)
* Second byte is register (second 4 bits)
* Third and fourth byte are the data or address (8bits)

**Instruction Format Display**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Type** | **Opcode** | | **DestReg** | **SrcReg1** | **SrcReg2** | |
| R | 4 bits | | 4 bits | 4 bits | 4 bits | |
| **Type** | | **Opcode** | **Regw/address** | **Unused** | |
| B | | 4 bits | 4 bits | 8 bits | |
| **Type** | | **Opcode** | **Register** | **Address** | |
| I | | 4 bits | 4 bits | 8 bits | |

**Instruction Library**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Opcode** | **Format** | **Address** |
| Load |  |  |  |
|  | LM | LM R1, 0x45 | 0x2 |
| Store |  |  |  |
|  | ST | ST R1, 0x45 | 0x3 |
| Branch |  |  |  |
|  | BEQ | BEQ R1, R2, R3 | 0x6 |
|  | BGT | BGT R1, R2, R3 | 0x7 |
|  | BLT | BLT R1, R2, R3 | 0x8 |
|  | B | B R3 | 0x9 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Opcode** | **Format** | **Address** |
| Add |  |  |  |
|  | ADDI | ADDI R1, R2, 5 | 0xA |
|  | ADD | ADD R2, R2, R3 | 0xB |
| Subtract |  |  |  |
|  | SUBI | SUBI R1, R2, 5 | 0xC |
|  | SUB | SUB R1, R2, R3 | 0xD |
| Shift |  |  |  |
|  | SL | SL R1, R2, 10 | 0xE |
|  | SR | SR R1, R2, 10 | 0xF |

|  |  |  |  |
| --- | --- | --- | --- |
| Logic |  |  |  |
|  | AND | AND R1, R2, R3 | 0x0 |
|  | OR | OR R1, R2, R3 | 0x1 |
| End |  |  |  |
|  | END | END | 0x4 |

**Assembler**

The assembler for this custom CPU was designed in Python. The purpose of the assembler is to take a .txt file that is filled out, line by line, with assembler instruction code. It then takes this code and turns it into a list of binary numbers displayed in an output file line by line.

**Part I.**

The code displayed below allows access to the assembler input text file. It then processes the file and separates all the instructions into tokens. In this code, a token is anything separated by whitespace.

file = "assemblerInput.txt"

filehandle = open("assemblerInput.txt","r")

with filehandle:

for line in filehandle:

tokenized\_line = line.split()

**Part II.**

The second part of the assembler, represented by the code below, is in charge of deciding what OPCODE leads each line. Once it reads the first token of each line, it uses and if-else statement to determine what to do based on what instruction it is. The general idea of what happens in each if-statement is: the register tokens, address tokens, and immediate tokens are all read and added into an array as decimal values. These values are then all converted into binary.

if tokenized\_line[0].upper() =="LM":

r=int(tokenized\_line[1][1])

hex = tokenized\_line[2]

hex\_list = hex.split("x")

v=int(hex\_list[(1)])

b=[2,r,v>>4,v&0xF]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l,]

**Part III.**

The final part of the assembler is sending these lines of binary into the output text file. This is displayed in the code below.

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

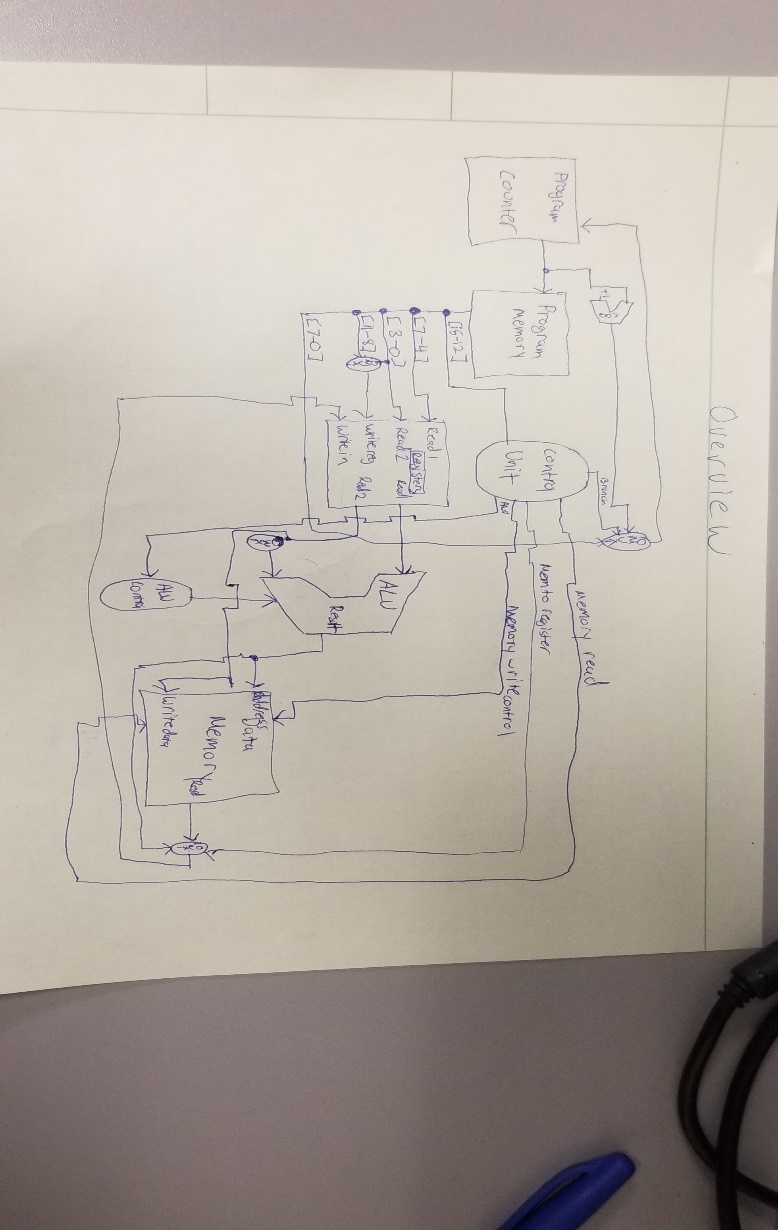
f.write(line)

f.write(' ')

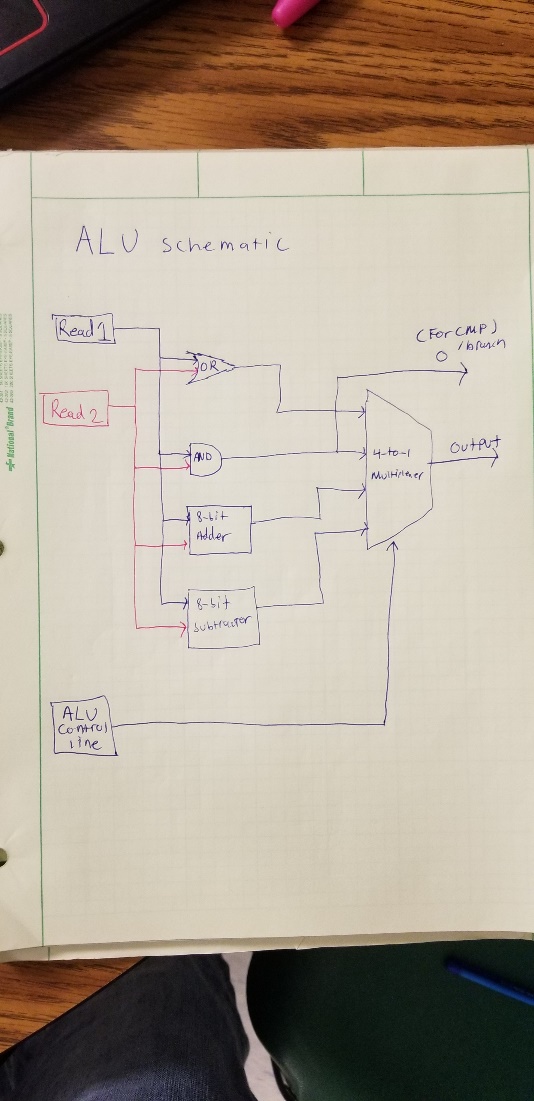
f.write('\n')

\*The full code is displayed in the code section at the end

-Overview Schematic for CPU



-ALU schematic



**Hardware implementation**

It was implemented using C++ where a block in the diagram (generally) correlates with a function in C++. The code is running a single cycle at a time with no concurrent tasks for more accurate simulation.

The architecture is a single cycle 8-bit computer. While this design was aimed at histogram equalization it could certainly be used for other purposes given its’ general, albeit, simple instruction set.

The ALU (code below) has 7 operations that are controlled by a bus coming from the control unit.

The control unit translates the opcode into different control lines for multilple components.

**VHDL:**

I have taken this time and have watched many tutorials and beginner to the VHDL language videos. As the CPU has been designed, I have made modeled small pieces with VHDL. Below is the model of one of the many muxes that we have will used, specifically the mux the chooses between our next address or the jump address for the PC. Now that we have our computer designed, I can now go and model some of the larger pieces in the PC that we were debating over in previous meetings.

**Example of a Mux (our PC mux in progress):**

library ieee;

use ieee.std\_logic\_1164.all;

--MUX to determine next Address

entity mux is

    port(

        choice  : in std\_logic;

        PCplus4 : in std\_logic\_vector(15 downto 0);

        Offset  : in std\_logic\_vector(15 downto 0);

        output : out std\_logic\_vector(15 downto 0)

    );

    end entity mux;

architecture sum of mux is

begin

   output <= Offset when (choice = '1') else PCplus4;

   end sum;

**C++ Implementation of the CPU:**

**Part I.**

In order for the CPU to do what we need, we read in the machine language instructions output of the Assembler and store them in the program memory. To do this, the AssemblerToProgramMem function would open the Assembler output and read the lines into a temporary array and then separates the values in the line and stores them into new array for processing.

ifstream file("Test.txt");

    if (file.is\_open())

    {

        while (!file.eof())

        {

            k = 0;

            // reads line in file

            getline(file, data[i], '\n');

            // copies line that is read

            toArray.str((data[i]));

            // stores copied in to array for bitset

            while (k < 4)

            {

                toArray >> array[k];

                k++;

            }

These values in this array were then trimmed to get rid of the leading “0b” so they could then be converted into integers. And stored into the official program memory.

for (int p = 0; p < 4; p++)

            {

                array[p] = array[p].erase(0, 2);

            }

-------------------------------------------------------------------------------------------------------------------------------

if (b < 255)

                {

                    while (a < 4)

                    {

                        bitset<8> bs4(array[a]);

                        programMem[b][a] = static\_cast<int>(bs4.to\_ulong());

                        a++;

                    }

                    a = 0;

                }

            }

            i++;

            b++;

Now that these values are in program memory, the can now be read out to in the decode stage using the MemtoRegisters function.

void MemtoRegisters(){

    opcode = (programMem[programCounter][0]);

    thirdReg = programMem[programCounter][1];

    if ((opcode == 2)||(opcode = 3)){

        address = programMem[programCounter][2];

    }

    else if (opcode == 9) {

//Address assigned based on register input

}

else {

        firstReg = programMem[programCounter][2];

        thirdReg = programMem[programCounter][3];

    }

}

**Part II.**

The ALU the takes the information gathered from these reads and performs the operation signified by the ALUControl which is given by the opcode. After the operation is done, the result is stored into the proper register or memory address

controlUnit(opcode);

// cout << opcode << endl;

ALUresult = ALU(firstReg, secondReg);

if (store == true)

{

switch (thirdReg)

{

// assign the result to the proper register

}

}

**Part III.**

This process is repeated and new data is put in the registers and operations are performed until the system reads in the end instruction from the instruction memory and the program stops.

**Code:**

**Assembler (Python)**

file = "assemblerInput.txt"

filehandle = open("assemblerInput.txt","r")

with filehandle:

for line in filehandle:

tokenized\_line = line.split()

if tokenized\_line[0].upper() =="LM":

r=int(tokenized\_line[1][1])

hex = tokenized\_line[2]

hex\_list = hex.split("x")

v=int(hex\_list[(1)])

b=[2,r,v>>4,v&0xF]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l,]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="ST":

r=int(tokenized\_line[1][1])

hex = tokenized\_line[2]

hex\_list = hex.split("x")

v=int(hex\_list[(1)])

b=[3,r,v>>4,v&0xF]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="BEQ":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[6,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="BGT":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[7,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="BLT":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[8,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="B":

r=int(tokenized\_line[1][1])

b=[9,r]

i=bin(b[0])

j=bin(b[1])

m=[i,j]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="ADDI":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3])

b=[10,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="ADD":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[11,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="SUBI":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3])

b=[12,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="SUB":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[13,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="SL":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3])

b=[14,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="SR":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3])

b=[15,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="AND":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[0,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="OR":

r=int(tokenized\_line[1][1])

v=int(tokenized\_line[2][1])

x=int(tokenized\_line[3][1])

b=[1,r,v,x]

i=bin(b[0])

j=bin(b[1])

k=bin(b[2])

l=bin(b[3])

m=[i,j,k,l]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

elif tokenized\_line[0].upper()=="END":

b=[4]

i=bin(b[0])

m=[i]

print(m)

lines = m

with open('assemblerOutput.txt', 'a') as f:

for line in lines:

f.write(line)

f.write(' ')

f.write('\n')

**CPU Implementation (C++)**

#include <fstream>

#include <string>

#include <iostream>

#include <sstream>

#include <bitset>

using namespace std;

bool load, store, branch, add, sub, addi, subi, andLine, orLine, memWrite, memRead, memToRegister, bType, lessThan, gThan, beq, toBranch;

bool endProg = false;

int ALUresult;

int regOne = 0, regTwo = 0, regThree = 0, regFour = 0, regFive = 0, regSix = 0, regSeven = 0, regEight = 0, regNine = 0, regTen = 0, regEleven = 0, regTwelve = 0, regThirteen = 0, regFourteen = 0, regFifteen = 0, regSixteen = 0;

int programMem[256][4];

int mainMemory[128];

int memoryRetrieve(int address);

void memoryStore(int toStore, int address);

int programCounter = 0;

int opcode;

int firstReg;

int secondReg;

int thirdReg;

int address;

int ALU(int inputA, int inputB);

void AssemblertoProgramMem();

void addressPlusOne();

void MemtoRegisters();

void controlUnit(int opCode)

{

load = false;

store = false;

branch = false;

add = false;

addi = false;

sub = false;

subi = false;

andLine = false;

orLine = false;

memWrite = false;

memRead = false;

memToRegister = false;

bType = false;

gThan = false;

lessThan = false;

beq = false;

toBranch = false;

switch (opCode)

{

case 2:

load = true;

memToRegister = true;

memRead = true;

break;

case 3:

store = true;

memWrite = true;

break;

case 4:

endProg = true;

break;

case 6:

beq = true;

break;

case 7:

gThan = true;

break;

case 8:

lessThan = true;

case 9:

branch = true;

case 10:

addi = true;

break;

case 11:

add = true;

break;

case 12:

subi = true;

break;

case 13:

sub = true;

break;

case 0:

andLine = true;

break;

case 1:

orLine = true;

break;

}

}

int main()

{

AssemblertoProgramMem();

programCounter = 0;

while (endProg == false)

{

MemtoRegisters();

controlUnit(opcode);

// cout << opcode << endl;

ALUresult = ALU(firstReg, secondReg);

if (store == true)

{

switch (thirdReg)

{

case 1:

memoryStore(regOne, address);

break;

case 2:

memoryStore(regTwo, address);

break;

case 3:

memoryStore(regThree, address);

break;

case 4:

memoryStore(regFour, address);

break;

case 5:

memoryStore(regFive, address);

break;

case 6:

memoryStore(regSix, address);

break;

case 7:

memoryStore(regSeven, address);

break;

case 8:

memoryStore(regEight, address);

break;

case 9:

memoryStore(regNine, address);

break;

case 10:

memoryStore(regTen, address);

break;

case 11:

memoryStore(regEleven, address);

break;

case 12:

memoryStore(regTwelve, address);

break;

case 13:

memoryStore(regThirteen, address);

break;

case 14:

memoryStore(regFourteen, address);

break;

case 15:

memoryStore(regFifteen, address);

break;

case 16:

memoryStore(regSixteen, address);

break;

}

}

if (load == true)

{

switch (firstReg)

{

case 1:

regOne = memoryRetrieve(address);

break;

case 2:

regTwo = memoryRetrieve(address);

break;

case 3:

regThree = memoryRetrieve(address);

break;

case 4:

regFour = memoryRetrieve(address);

break;

case 5:

regFive = memoryRetrieve(address);

break;

case 6:

regSix = memoryRetrieve(address);

break;

case 7:

regSeven = memoryRetrieve(address);

break;

case 8:

regEight = memoryRetrieve(address);

break;

case 9:

regNine = memoryRetrieve(address);

break;

case 10:

regTen = memoryRetrieve(address);

break;

case 11:

regEleven = memoryRetrieve(address);

break;

case 12:

regTwelve = memoryRetrieve(address);

break;

case 13:

regThirteen = memoryRetrieve(address);

break;

case 14:

regFourteen = memoryRetrieve(address);

break;

case 15:

regFifteen = memoryRetrieve(address);

break;

case 16:

regSixteen = memoryRetrieve(address);

break;

}

}

if (branch == true || beq == true || lessThan == true || gThan == true)

{

programCounter = address;

}

else

{

switch (thirdReg)

{

case 1:

regOne = ALUresult;

break;

case 2:

regTwo = ALUresult;

break;

case 3:

regThree = ALUresult;

break;

case 4:

regFour = ALUresult;

break;

case 5:

regFive = ALUresult;

break;

case 6:

regSix = ALUresult;

break;

case 7:

regSeven = ALUresult;

break;

case 8:

regEight = ALUresult;

break;

case 9:

regNine = ALUresult;

break;

case 10:

regTen = ALUresult;

break;

case 11:

regEleven = ALUresult;

break;

case 12:

regTwelve = ALUresult;

break;

case 13:

regThirteen = ALUresult;

break;

case 14:

regFourteen = ALUresult;

break;

case 15:

regFifteen = ALUresult;

break;

case 16:

regSixteen = ALUresult;

break;

}

addressPlusOne();

}

}

cout << mainMemory[1] << endl;

cout << mainMemory[2] << endl;

}

int ALU(int inputA, int inputB)

{

int result;

if (load != true || store != true)

{

int dataOne;

int dataTwo;

switch (inputA)

{

case 1:

dataOne = regOne;

break;

case 2:

dataOne = regTwo;

break;

case 3:

dataOne = regThree;

break;

case 4:

dataOne = regFour;

break;

case 5:

dataOne = regFive;

break;

case 6:

dataOne = regSix;

break;

case 7:

dataOne = regSeven;

break;

case 8:

dataOne = regEight;

break;

case 9:

dataOne = regNine;

break;

case 10:

dataOne = regTen;

break;

case 11:

dataOne = regEleven;

break;

case 12:

dataOne = regTwelve;

break;

case 13:

dataOne = regThirteen;

break;

case 14:

dataOne = regFourteen;

break;

case 15:

dataOne = regFifteen;

break;

case 16:

dataOne = regSixteen;

break;

}

if (add == true || sub == true || andLine == true || orLine || true || beq == true || gThan == true || lessThan == true)

{

switch (inputB)

{

case 1:

dataTwo = regOne;

break;

case 2:

dataTwo = regTwo;

break;

case 3:

dataTwo = regThree;

break;

case 4:

dataTwo = regFour;

break;

case 5:

dataTwo = regFive;

break;

case 6:

dataTwo = regSix;

break;

case 7:

dataTwo = regSeven;

break;

case 8:

dataTwo = regEight;

break;

case 9:

dataTwo = regNine;

break;

case 10:

dataTwo = regTen;

break;

case 11:

dataTwo = regEleven;

break;

case 12:

dataTwo = regTwelve;

break;

case 13:

dataTwo = regThirteen;

break;

case 14:

dataTwo = regFourteen;

break;

case 15:

dataTwo = regFifteen;

break;

case 16:

dataTwo = regSixteen;

break;

}

}

if (addi == true)

{

result = dataOne + inputB;

}

if (subi == true)

{

result = dataOne - inputB;

}

if (add == true)

{

result = dataOne + dataTwo;

}

if (sub == true)

{

result = dataOne - dataTwo;

}

if (andLine == true)

{

result = dataOne & dataTwo;

}

if (orLine == true)

{

result = dataOne | dataTwo;

}

if (beq == true)

{

if (dataOne == dataTwo)

{

toBranch = true;

}

}

if (lessThan == true)

{

if (dataOne < dataTwo)

{

toBranch = true;

}

}

if (gThan == true)

{

if (dataOne > dataTwo)

{

toBranch = true;

}

}

}

return result;

}

int memoryRetrieve(int address)

{

return mainMemory[address];

}

void memoryStore(int toStore, int address)

{

mainMemory[address] = toStore;

}

void AssemblertoProgramMem();

void addressPlusOne();

void AssemblertoProgramMem()

{

string line;

int i = 0;

int k = 0;

string data[256];

string array[4];

int b = 0;

int a = 0;

stringstream toArray("");

ifstream file("Test.txt");

if (file.is\_open())

{

while (!file.eof())

{

k = 0;

// reads line in file

getline(file, data[i], '\n');

// copies line that is read

toArray.str((data[i]));

// stores copied in to array for bitset

while (k < 4)

{

toArray >> array[k];

k++;

}

// resets the stream

stringstream().swap(toArray);

for (int p = 0; p < 4; p++)

{

array[p] = array[p].erase(0, 2);

}

if ((array[0] == "10") || (array[0] == "11") || (array[0] == "1001"))

{

array[2] = array[2] + array[3];

array[3] = "";

if (b < 255)

{

while (a < 4)

{

bitset<8> bs4(array[a]);

programMem[b][a] = (bs4.to\_ulong());

a++;

}

a = 0;

}

}

else

{

if (b < 255)

{

while (a < 4)

{

bitset<8> bs4(array[a]);

programMem[b][a] = (bs4.to\_ulong());

a++;

}

a = 0;

}

}

i++;

b++;

}

}

return;

}

void addressPlusOne()

{

programCounter++;

}

void MemtoRegisters()

{

opcode = programMem[programCounter][0];

thirdReg = programMem[programCounter][1];

if ((opcode == 2 || opcode == 3))

{

address = programMem[programCounter][2];

}

else if (opcode == 9)

{

thirdReg = programMem[programCounter][1];

switch (thirdReg)

{

case 1:

address = regOne;

break;

case 2:

address = regTwo;

break;

case 3:

address = regThree;

break;

case 4:

address = regFour;

break;

case 5:

address = regFive;

break;

case 6:

address = regSix;

break;

case 7:

address = regSeven;

break;

case 8:

address = regEight;

break;

case 9:

address = regNine;

break;

case 10:

address = regTen;

break;

case 11:

address = regEleven;

break;

case 12:

address = regTwelve;

break;

case 13:

address = regThirteen;

break;

case 14:

address = regFourteen;

break;

case 15:

address = regFifteen;

break;

case 16:

address = regFifteen;

break;

}

}

else

{

firstReg = programMem[programCounter][2];

secondReg = programMem[programCounter][3];

}

}