ViRtual CPU Final Report

By: Richard Clapham | n00663650

4/20/2016

Table of Contents

[Description 2](#_Toc448852576)

[Final State of Program 2](#_Toc448852577)

[Sample output of Register display and Memory Dump 3](#_Toc448852578)

[Memory Dump 3](#_Toc448852579)

[Display Registers 3](#_Toc448852580)

[Testing 4](#_Toc448852581)

[Conclusion 4](#_Toc448852582)

[Appendix 5](#_Toc448852583)

[Main.h 5](#_Toc448852584)

[Main.c 9](#_Toc448852585)

# Description

The virtual CPU project is designed to simulate an ARM x86 processor at a register level. The CPU has 16 kilobytes of memory that the CPU uses to store instructions. The CPU is then capable of executing the instructions from memory and displaying how each register would be affected after an instruction is executed.

When the CPU is first started it will reset all registers to zero. It will then allow the user to choose a command. The main command being used is the trace command in which it will execute all the commands loading into memory one at a time and display the registers after each instruction. The instruction cycle starts by determining which instruction register to use. (This information is stored in a flag called IR\_FLAG). If the flag is zero it will begin the fetch phase of the instruction cycle. In the fetch stage the memory address register (MAR) will be set equal to the program counter. It will then load a total of 32 bits into the memory buffer register (MBR). (It moves 8 bits at a time because in the simulation the CPU can only move 8 bits at a time from memory to the MBR). It will then load the MBR into the instruction register (IR) and finally the fetch command will then increase the program counter by 4 bytes. After the fetch has completed the virtual CPU will then execute the command in instruction register 0 (IR0) and then change the IR\_FLAG to 1. The next time the trace command is called the IR\_FLAG will be equal to 1 so instead of having to fetch data the CPU simply needs to execute the command that was already loaded into instruction register 1 (IR1). Finally, the IR\_FLAG will then be set to 0 and the instruction cycle will then be capable of beginning again.

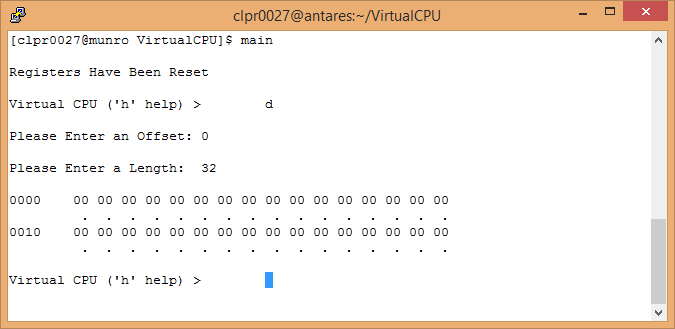
# Final State of Program

The final program appears to be functioning correctly with no errors that I am currently aware of. The virtual CPU meets all the requirements of the CPU reference document given to us. Also, each function meets its intended purpose the commands and their purpose are as follows:

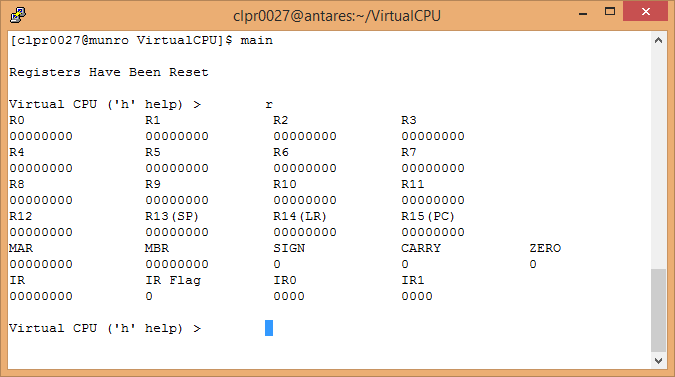
* Dump Memory – Displays the contents of memory in hexadecimal and ASCII
* Go – Runs all the commands in memory until either maximum memory is reached or the stop flag is set
* Load File – Receives a file from the user and loads the contents into memory. This will truncate the file it is larger than maximum memory.
* Memory Modify – Allows the user to enter an offset and then proceed to modify the value at that specific location in memory. Receives in two hexadecimal values and will check to ensure the values entered are in hexadecimal.
* Quit – Terminates the program
* Display Registers – Displays the current contents of each register
* Trace – Executes one instruction and then displays the registers
* Write File – Creates a file with a size and name specified by the user.
* Reset Registers – Sets all the registers to zero
* Help – Displays a list of all commands available to the user

# Sample output of Register display and Memory Dump

## Memory Dump



## Display Registers



# Testing

The testing of the project was to simply use a hex editor to create a file. (The hex editor I used was HxDen). The file would then be given the commands in hex editor such as E000. (This is a sample of a stop command and should simply just set the STOP flag). The file would then be loaded into memory and executed. If the instruction that was executed operated according to the CPU reference document, then I considered the that portion of the program working correctly.

# Conclusion

In conclusion, I have created a well-documented program that is designed to simulate an ARM x86 processor at a register level. The virtual CPU I created appears to function correctly with no errors that I am currently aware of. The program was developed in both visual studio 2015 and occasionally using a gcc compiler.

In conclusion, the virtual CPU project was a lot of work and programming however, it was very rewarding to have completed the project and I believe it will assist in finding future employment in software development. This project allowed me to get a much greater understanding of how a CPU functions then when I first entered this program.

# Appendix

## Main.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Name: Richard Clapham

\* Student #: 821-490-125

\* Student #: n00663650

\* Date: 4/7/2016

\* Virtual CPU project Header

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef MAIN

#define MAIN

#include <stdint.h>

#include <stdio.h>

#include <stdbool.h>

#include <sys/types.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#define MEMORY\_MAX 0x3E80 //Determines maximum memory size (16000)

#define MAX32 0xFFFFFFFF

#define HEX\_COLUMN 16

#define REG\_NUM 16 //Determines amount of registers

/\* Defines all registers and flags \*/

#define SP R[13] //Stack Pointer

#define LR R[14] //Link Register

#define PC R[15] //Program Counter

#define IR IR //Instruction Register Combination of IR0 & IR1

#define ALU ALU //Arithmetic Logic Unit

#define MAR MAR //Memory Address Register

#define MBR MBR //Memory Buffer Register

#define IR0 IR0 //Instruction Register 0

#define IR1 IR1 //Instruction Register 1

#define SIGN SIGN //Sign Flag

#define CARRY CARRY //Carry Flag

#define ZERO ZERO //Zero Flag

#define STOP\_FLAG STOP\_FLAG //Flag Set by Stop Instruction

#define IR\_FLAG IR\_FLAG //Flag to determine the active Instruction Register

/\* Registers holds all hardware except for memory 32-bit \*/

typedef struct{

uint32\_t IR,

ALU,

R[REG\_NUM],

MAR,

MBR;

/\* Instruction Registers 16-bit \*/

uint16\_t IR0,

IR1;

/\* Condition Code Registers 8-bit \*/

uint8\_t SIGN,

CARRY,

ZERO;

} registers;

registers regs;

/\* STOP and whichever IR is Active Flag \*/

bool STOP\_FLAG,

IR\_FLAG;

unsigned Current\_IR;

int Current\_TYPE;

/\* Flag Checks \*/

#define ISZERO(x) (x == 0) ? 1 : 0

#define ISSIGN(x) ((0x8000 & x) == 0x8000) ? 1 : 0

/\* Determining Instruction Type \*/

#define DATA\_PROCESS 0x0 //Data Processing

#define LOAD\_STORE 0x1 //Load / Store

#define IMMEDIATE\_1 0x2 //Immediate Option 1

#define IMMEDIATE\_2 0x3 //Immediate Option 2

#define CON\_BRANCH 0x4 //Conditional Branch

#define PUSH\_PULL 0x5 //Push / Pull

#define UN\_BRANCH 0x6 //Unconditional Branch

#define T\_STOP 0x7 //Stop

/\* Mask Definitions \*/

/\* Data Process \*/

#define DATA\_OP ((Current\_IR & 0xF00) >> 8) //Operation

#define DATA\_RN ((Current\_IR & 0xF0) >> 4) //(Secondary Operand)

#define DATA\_RD (Current\_IR & 0x000F) //Destination (Primary Operand)

/\* Data Process Codes \*/

#define DATA\_ADC 0x5 //Add with Carry

#define DATA\_ADD 0x4 //Add / Add Immediate

#define DATA\_AND 0x0 //And

#define DATA\_BIC 0xE //Bit Clear

#define DATA\_CMP 0xA //Compare / Compare Immediate

#define DATA\_EOR 0x1 //Exclusive Or

#define DATA\_LSL 0x7 //Logical Shift Left

#define DATA\_LSR 0x6 //Logical Shift Right

#define DATA\_MOV 0xD //Move / Move Immediate

#define DATA\_MVN 0xF //Move Not

#define DATA\_ORR 0xC //Or

#define DATA\_ROR 0xB //Rotate Right

#define DATA\_SUB 0x2 //Subtract / Subtract Immediate

#define DATA\_SXB 0x3 //Sign Extend Byte

#define DATA\_TEQ 0x9 //Test Equivalence

#define DATA\_TST 0x8 //Test Bits

/\* Load / Store \*/

#define LOAD\_STORE\_B ((Current\_IR & 0x800) >> 11) //Byte / Word Bit (0 = transfer word, 1 = transfer byte)

#define LOAD\_STORE\_L ((Current\_IR & 0x400) >> 10) //Load / Store Bit (0 = Store to Memory, 1 = Load from Memory)

#define LOAD\_STORE\_RN ((Current\_IR & 0x0F0) >> 4) //Memory Address

#define LOAD\_STORE\_RD (Current\_IR & 0x000F) //Source / Destination Register

/\* Immediate Operations \*/

#define IMM\_OP ((Current\_IR & 0x3000) >> 12) //Operation

#define IMM\_VALUE ((Current\_IR & 0x0FF0) >> 4) //Value

#define IMM\_RD (Current\_IR & 0x000F) //Source / Destination Register

/\* Immediate Operations Code \*/

#define IMM\_MOV 0x0 //Move

#define IMM\_CMP 0x1 //Compare

#define IMM\_ADD 0x2 //Add

#define IMM\_SUB 0x3 //Subtract

/\* Conditional Branch \*/

#define CBRANCH\_COND ((Current\_IR & 0xF00) >> 8)

#define CBRANCH\_ADDR (Current\_IR & 0x00FF)

/\* Conditional Branch Codes \*/

#define CBRANCH\_EQ 0x0000 //Equal (Zero Set)

#define CBRANCH\_NE 0x0001 //Not Equal (Zero Clear)

#define CBRANCH\_CS 0x0002 //Unsigned Higher or Same (Carry Set)

#define CBRANCH\_CC 0x0003 //Unsigned Lower (Carry Clear)

#define CBRANCH\_MI 0x0004 //Negative (Sign Set)

#define CBRANCH\_PL 0x0005 //Positive (Sign Clear)

#define CBRANCH\_HI 0x0008 //Unsigned Higher (Carry Set & Zero Clear)

#define CBRANCH\_LS 0x0009 //Unsigned Lower or Same (Carry Clear || Zero Set)

#define CBRANCH\_AL 0x000E //Always (No Flags)

/\* Push / Pull \*/

#define PUSH\_PULL\_L ((Current\_IR & 0x0800) >> 11) //Load / Store Bit (0 = Push, 1 = Pull)

#define PUSH\_PULL\_R ((Current\_IR & 0x0100) >> 8) //Additional Push / Pull Bit (0 = No Extra Push or Pulls, 1 = Pull PC / Push LR)

#define PUSH\_PULL\_H ((Current\_IR & 0x0400) >> 10) //High / Low Bit (0 = Low Registers, 1 = High Registers)

#define PUSH\_REG\_LIST (Current\_IR & 0x00FF)

/\* Unconditional Branch \*/

#define UBRANCH\_LINK\_K ((Current\_IR & 0x1000) >> 12) //Link Bit (0 = Branch (BRA), 1 = Branch with Link (BRL))

#define UBRANCH\_OFFSET (Current\_IR & 0x0FFF)

/\* My Functions \*/

void error(char \*msg);

void execute(uint16\_t instruction, uint16\_t type, void \*memptr);

void fetch(void \*memptr);

void instructionCycles(void \*memptr);

int isCarry(unsigned long op1, unsigned long op2, unsigned C);

void flagsCheck(uint32\_t register\_1, uint32\_t register\_2, int flag\_Check);

void memDump(void \*memptr, unsigned offset, unsigned length);

void go();

int loadFile(void \*buffer, unsigned int max);

void memmod(void \*memptr, unsigned offset);

void quit();

void displayRegisters();

void trace(void \*memptr);

void writeFile(void \*memory);

void resetRegisters();

void help();

int main(int argc, char \*argv[]);

#endif

## Main.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Name: Richard Clapham

\* Student #: 821-490-125

\* Student #: n00663650

\* Date: 4/7/2016

\* Virtual CPU project

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "main.h"

char cmd = 'q';

char buffer[16000];

/\*\*

Used to display error messages currently not being used

\*\*/

void error(char \*msg)

{

perror(msg);

exit(0);

}

/\*\*

Function designed to execute the commands given

@param instruction - Receives the current instruction

@param type - Receives and determines the type of instruction

@param memptr - Receives the Virtual CPU's buffer

\*\*/

void execute(uint16\_t instruction, uint16\_t type, void \*memptr)

{

Current\_IR = instruction;

Current\_TYPE = type;

//Data Process

if (type == DATA\_PROCESS) {

if(DATA\_OP == DATA\_ADC){ //Add with Carry

regs.ALU = regs.R[DATA\_RD] + DATA\_RN + regs.CARRY;

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 1);

}

else if(DATA\_OP == DATA\_ADD){ //Add / Add Immediate

regs.ALU = regs.R[DATA\_RD] + regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 1);

}

else if(DATA\_OP == DATA\_AND){ //And

regs.ALU = regs.R[DATA\_RD] & regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_BIC){ //Bit Clear

regs.ALU = regs.R[DATA\_RD] & ~regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_CMP){ //Compare / Compare Immediate

regs.ALU = regs.R[DATA\_RD] & ~regs.R[DATA\_RN] + 1;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_EOR){ //Exclusive Or

regs.ALU = regs.R[DATA\_RD] ^ regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_LSL){ //Logical Shift Left

regs.ALU = regs.R[DATA\_RD] << regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 1);

}

else if(DATA\_OP == DATA\_LSR){ //Logical Shift Right

regs.ALU = regs.R[DATA\_RD] >> regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 1);

}

else if(DATA\_OP == DATA\_MOV){ //Move / Move Immediate

regs.R[DATA\_RD] = regs.R[DATA\_RN];

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_MVN){ //Move Not

regs.R[DATA\_RD] = ~regs.R[DATA\_RN];

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_ORR){ //Or

regs.ALU = regs.R[DATA\_RD] | regs.R[DATA\_RN];

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_ROR){ //Rotate Right

for (int i = 0; i < regs.R[DATA\_RN]; i++)

{

regs.CARRY = regs.R[DATA\_RD] & 0x0001;

if (regs.CARRY)

regs.ALU = (regs.R[DATA\_RD] >> 1) | 0x80000000;

}

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_SUB){ //Subtract / Subtract Immediate

regs.ALU = regs.R[DATA\_RD] + ~regs.R[DATA\_RN] + 1;

regs.R[DATA\_RD] = regs.ALU;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_SXB){ //Sign Extend Byte

regs.R[DATA\_RD] = (signed)regs.R[DATA\_RN];

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_TEQ){ //Test Equivalence

regs.ALU = regs.R[DATA\_RD] ^ regs.R[DATA\_RN] + 1;

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else if(DATA\_OP == DATA\_TST){ //Test Bits

regs.ALU = regs.R[DATA\_RD] & regs.R[DATA\_RN];

flagsCheck(regs.R[DATA\_RD], regs.R[DATA\_RN], 0);

}

else{

perror("Operation not found!");

}

}

//Load / Store

else if (type == LOAD\_STORE) {

uint32\_t mask = 0xFF000000;

if(LOAD\_STORE\_B == 0){ //Transfer Word

if(LOAD\_STORE\_L == 0){ //Store to Memory

regs.MAR = regs.R[LOAD\_STORE\_RN];

regs.MBR = regs.R[LOAD\_STORE\_RD];

for(int i = 0; i < 4; i++){

/\* Move MSB first into memory, then the rest \*/

((uint8\_t \*)memptr)[regs.MAR++] = (regs.MBR & mask) >> (8 \* (3 - i));

mask = mask >> 8;

}

}

else if(LOAD\_STORE\_L == 1){ //Load from Memory

regs.MAR = regs.R[LOAD\_STORE\_RN];

regs.MBR = regs.R[LOAD\_STORE\_RD];

((uint8\_t \*) memptr)[regs.MAR] = regs.MBR;

}

}

else if(LOAD\_STORE\_B == 1){ //Transfer Byte

if(LOAD\_STORE\_L == 0){ //Store to Memory

regs.MAR = regs.R[LOAD\_STORE\_RN];

for(int i = 0; i < 4; i++){

regs.MBR = regs.MBR << 8;

regs.MBR += ((uint8\_t \*) memptr)[regs.MAR++];

}

regs.R[LOAD\_STORE\_RD] = regs.MBR;

}

else if(LOAD\_STORE\_L == 1){ //Load from Memory

regs.MAR = regs.R[LOAD\_STORE\_RN] & 0x000000FF;

regs.MBR = ((uint8\_t \*) memptr)[regs.MAR];

regs.R[LOAD\_STORE\_RD] = regs.MBR;

}

}

}

//Immediate Operations

else if (type == IMMEDIATE\_1 || type == IMMEDIATE\_2) {

if (IMM\_OP == IMM\_MOV) { //Move

regs.ALU = IMM\_VALUE;

regs.R[IMM\_RD] = regs.ALU;

flagsCheck(regs.R[IMM\_RD], IMM\_VALUE, 0);

}

else if (IMM\_OP == IMM\_CMP) { //Compare

regs.ALU = regs.R[IMM\_RD] - IMM\_VALUE;

flagsCheck(regs.ALU, IMM\_VALUE, 1);

}

else if (IMM\_OP == IMM\_ADD) { //Add

regs.ALU = regs.R[IMM\_RD] + IMM\_VALUE;

regs.R[IMM\_RD] = regs.ALU;

flagsCheck(regs.R[IMM\_RD], IMM\_VALUE, 1);

}

else if (IMM\_OP == IMM\_SUB) { //Subtract

regs.ALU = regs.R[IMM\_RD] + ~IMM\_VALUE + 1;

regs.R[IMM\_RD] = regs.ALU;

flagsCheck(regs.R[IMM\_RD], IMM\_VALUE, 1);

}

}

//Conditional Branch

else if (type == CON\_BRANCH) {

/\* Use hold value to sign CBRANCH\_ADDR \*/

int8\_t temp = CBRANCH\_ADDR;

int32\_t offset = (int32\_t)temp;

/\* Decrement PC to reset the branch fetch \*/

regs.PC -= 2;

if(CBRANCH\_COND == CBRANCH\_EQ){ //Equal (Zero Set)

if (regs.ZERO == true)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_NE) { //Not Equal (Zero Clear)

if (regs.ZERO == false)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_CS) { //Unsigned Higher or Same (Carry Set)

if (regs.CARRY = true)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_CC) { //Unsigned Lower (Carry Clear)

if (regs.CARRY = false)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_MI) { //Negative (Sign Set)

if (regs.SIGN == true)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_PL) { //Positive (Sign Clear)

if (regs.SIGN == false)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_HI) { //Unsigned Higher (Carry Set & Zero Clear)

if (regs.CARRY == true && regs.ZERO == false)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_LS) { //Unsigned Lower or Same (Carry Clear || Zero Set)

if (regs.CARRY == false && regs.ZERO == true)

regs.PC += offset;

}

else if (CBRANCH\_COND == CBRANCH\_AL) { //Always (No Flags)

regs.PC += offset;

}

regs.IR = 0;

IR\_FLAG = 0;

}

//Push / Pull

else if (type == PUSH\_PULL) {

uint8\_t current\_Register;

uint32\_t mask = 0xFF000000;

if (regs.SP == 0)

regs.SP = MEMORY\_MAX;

fprintf(stdout, "PUSH\_PULL\_L: %X\nPUSH\_PULL\_R: %X\nPUSH\_PULL\_H: %X\n", PUSH\_PULL\_L, PUSH\_PULL\_R, PUSH\_PULL\_H);

if(PUSH\_PULL\_L == 0){ // PUSH

if (PUSH\_PULL\_R == 0) { // No extra Push / Pull

if (PUSH\_PULL\_H == 0) { // Low Registers

current\_Register = 7;

for (int i = 0x80; i > 0;){

if ((PUSH\_REG\_LIST & i) != 0){

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

else if (PUSH\_PULL\_H == 1) { // High Registers

current\_Register = 12;

for (int i = 0x80; i > 0;){

if ((PUSH\_REG\_LIST & i) != 0){

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

}

else if (PUSH\_PULL\_R == 1) { // Extra Push / Pull

if (PUSH\_PULL\_H == 0) { // Low Registers

current\_Register = 7;

for (int i = 0x80; i > 0;) {

if ((PUSH\_REG\_LIST & i) != 0) {

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

else if (PUSH\_PULL\_H == 1) { //High Registers

current\_Register = 12;

for (int i = 0x80; i > 0;) {

if ((PUSH\_REG\_LIST & i) != 0) {

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

}

else if (PUSH\_PULL\_L == 1) { // PULL

if (PUSH\_PULL\_R == 0) { // No Extra Push / Pull

if (PUSH\_PULL\_H == 0) { // Low Registers

current\_Register = 7;

for (int i = 0x80; i > 0;) {

if ((PUSH\_REG\_LIST & i) != 0) {

instruction = 0x28D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

regs.SP -= 0x0004;

}

current\_Register--;

i = i >> 1;

}

}

else if (PUSH\_PULL\_H == 1) { // High Registers

current\_Register = 12;

for (int i = 0x80; i > 0;) {

if ((PUSH\_REG\_LIST & i) != 0) {

instruction = 0x28D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

regs.SP -= 0x0004;

}

current\_Register--;

i = i >> 1;

}

}

}

else if (PUSH\_PULL\_R == 1) { // Extra Push / Pull

if (PUSH\_PULL\_H == 0) { // Low Registers

current\_Register = 7;

for (int i = 0x80; i > 0;){

if ((PUSH\_REG\_LIST & i) != 0){

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

else if (PUSH\_PULL\_H == 1) { // High Registers

current\_Register = 12;

for (int i = 0x80; i > 0;)

{

if ((PUSH\_REG\_LIST & i) != 0)

{

regs.SP -= 0x0004;

instruction = 0x20D0 + current\_Register;

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

current\_Register--;

i = i >> 1;

}

}

type = LOAD\_STORE;

execute(instruction, type, buffer);

}

}

}

//Unconditional Branch

else if (type == UN\_BRANCH) {

if (UBRANCH\_LINK\_K == 0){ // Branch (BRA)

regs.PC = UBRANCH\_OFFSET;

}

else{ // Branch with Link (BRL)

regs.LR = regs.PC;

regs.PC = UBRANCH\_OFFSET;

}

}

//Stop

else if (type == T\_STOP) {

STOP\_FLAG = 1;

}

else {

perror("Type Not Found!");

}

}

/\*\*

Function designed to fetch the first command from memory

@param memptr - Receives the Virtual CPU's buffer

\*\*/

void fetch(void \*memptr)

{

regs.MAR = regs.PC;

for(int i = 0; i < 4; i++){

regs.MBR = regs.MBR << 8;

regs.MBR = regs.MBR + ((char\*)memptr)[regs.MAR++];

}

regs.IR = regs.MBR;

regs.PC = regs.PC + sizeof(uint32\_t);

}

/\*\*

Responsible for controlling the fetch and execute functions

@param memptr - Receives the Virtual CPU's buffer

\*\*/

void instructionCycles(void \*memptr)

{

int type = 0;

if (IR\_FLAG == 0){

fetch(buffer);

regs.IR0 = (regs.IR >> 16);

regs.IR1 = regs.IR;

type = (regs.IR0 & 0xE000) >> 13;

execute(regs.IR0, type, buffer);

IR\_FLAG = 1;

}

else{

type = (regs.IR1 & 0xE000) >> 13;

execute(regs.IR1, type, buffer);

IR\_FLAG = 0;

}

}

/\*\*

isCarry()- determine if carry is generated by addition: op1+op2+C

@param op1 - receives register 1

@param op2 - receives register 2

@param C - receives the current carry flag

@return int - returns 1 or 0 depending if the carry is set

\*\*/

int isCarry(unsigned long op1,unsigned long op2, unsigned C)

{

if ((op2 == MAX32) && (C == 1))

return 1; // special case where op2 is at MAX32

return((op1 > (MAX32-op2-C))?1:0);

}

/\*\*

Checks to see if flags are set

@param register\_1 - Recieves the destination register (primary operand)

@param register\_2 - Recieves the (secondary operand)

@param flag\_Check - Determines the type of flag check to run (0 = No carry, 1 = Carry)

\*\*/

void flagsCheck(uint32\_t register\_1, uint32\_t register\_2, int flag\_Check)

{

if (flag\_Check == 0) {

if (ISSIGN(register\_1))

regs.SIGN = 1;

else if (!ISSIGN(register\_1))

regs.SIGN = 0;

if (ISZERO(register\_1))

regs.ZERO = 1;

if (!ISZERO(register\_1))

regs.ZERO = 0;

}

else if (flag\_Check == 1) {

if (ISZERO(register\_1))

regs.ZERO = 1;

else if (!ISZERO(register\_1))

regs.ZERO = 0;

if (ISSIGN(register\_1))

regs.SIGN = 1;

else if (!ISSIGN(register\_1))

regs.SIGN = 0;

if (isCarry(register\_1, register\_2, regs.CARRY))

regs.CARRY = 1;

else if (!isCarry(register\_1, register\_2, regs.CARRY))

regs.CARRY = 0;

}

}

/\*\*

Allows the user to view a portion of the memory

@param memptr - Receives the Virtual CPU's buffer

@param offset - Receives the point in memory to begin viewing

@param length - Receives the length of memory to view

\*\*/

void memDump(void \*memptr, unsigned offset, unsigned length)

{

/\* Ensure offset is less than 16000 if not it defaults to 0 \*/

if(offset >= MEMORY\_MAX){

offset = 0x0000;

}

/\* Ensures that the maximum length is 16000 \*/

if(length > MEMORY\_MAX){

length = MEMORY\_MAX;

}

unsigned int end = offset + length;

/\* Ensures that if it is a valid offset and valid length it won't display data

that is not inside the buffer \*/

while (end > MEMORY\_MAX) {

end = MEMORY\_MAX;

}

for (int i = offset; i < end; i += HEX\_COLUMN)

{

/\* Prints the current location in memory \*/

if (i != offset) putchar('\n'); {

fprintf(stdout, "%04X\t", i);

}

for (int j = i; j < (i + HEX\_COLUMN); j++)

{

/\* If the length is reached before the end of the line exit loop \*/

if (j == (length + offset))

{

fprintf(stdout, "\n\t");

for (int k = i; k <= (length + offset - 1); k++)

{

//This if statement cause the error with visual studio (runtime error only)

if (isprint(((char\*)memptr)[k])) {

fprintf(stdout, " %c ", ((char\*)memptr)[k]);

}

else {

fprintf(stdout, " . ");

}

}

break;

}

/\* Printf the Hex values \*/

fprintf(stdout, "%02X ", ((unsigned char\*)memptr)[j]);

/\* Printf the ASCII character of hex values if possible, else print '.' \*/

if (j == (i + HEX\_COLUMN) - 1)

{

fprintf(stdout, "\n\t");

for (int k = i; k < (i + HEX\_COLUMN); k++)

{

//This if statement cause the error with visual studio (runtime error only)

if (isprint(((char\*)memptr)[k])) {

fprintf(stdout, " %c ", ((char\*)memptr)[k]);

}

else {

fprintf(stdout, " . ");

}

}

}

}

}

fprintf(stdout, "\n");

}

void go(void \*memptr)

{

STOP\_FLAG = 0;

while (STOP\_FLAG == 0 && regs.PC < MEMORY\_MAX) {

instructionCycles(&memptr);

}

displayRegisters();

//fprintf(stdout, "\nGo has not been implemented\n");

}

/\*\*

Requests a file name from the user and then attempts to load the file into memory

@param buffer - Receives the Virtual CPU's buffer

@param max - Receives the Virtual CPU's maximum buffer size

@return int - Returns The file size (Returns -1 if file does not exist)

\*\*/

int loadFile(void \*memptr, unsigned int max)

{

FILE \*fp;

char filename[50];

int readCount;

int check = 0;

/\* Prompts user for File Name \*/

fprintf(stdout, "\nPlease Enter a File name:\t");

fscanf(stdin, "%s", filename);

/\* Attempts to open file with given name will state if file is truncated or if it does not exist\*/

fp = fopen(filename, "r");

if(fp != NULL)

{

fseek(fp, 0L, SEEK\_END);

check = ftell(fp);

fseek(fp, 0L, SEEK\_SET);

readCount = fread(memptr,1,max,fp);

fclose(fp);

if(check > 16000){

fprintf(stdout, "The File was Truncated");

}else{

fprintf(stdout, "The File was not Truncated");

}

}

else

{

fprintf(stdout, "FILE DOES NOT EXIST\n");

return(-1);

}

return readCount;

}

/\*\*

Allows the user to modify individual bytes in memory based off of a given location

uses "." to exit

@param memptr - Receives the Virtual CPU's buffer

@param offset - Receives the point in memory to begin modifying

\*\*/

void memmod(void \*memptr, unsigned offset)

{

/\* Ensure offset is less than 16000 if not it defaults to 0 \*/

if (offset >= MEMORY\_MAX) {

offset = 0x0000;

}

char user\_input[sizeof(int) + 1];

unsigned int value;

while (1)

{

/\* Displays information about current offset \*/

fprintf(stdout, "Address\t0x%04X\n", offset);

fprintf(stdout, "Value:\t(%02X)\n", ((unsigned char\*)memptr)[offset]);

fprintf(stdout, "Please Enter a Valid 2 Digit Hex Number, '.' to exit:\t");

fgets(user\_input, sizeof(int) + 1, stdin);

/\* Checks if user\_input is a '.' or '.\n' then exits function if user entered a '.' \*/

if (strcmp(user\_input, ".\n") == 0 || strcmp(user\_input, ".") == 0) {

fprintf(stdout, "Exiting Memory Modify...\n");

break;

}

/\* Checks if user\_input is a valid 2 digit hexidecimal number \*/

else if (!isxdigit(user\_input[0]) || !isxdigit(user\_input[1])){

fprintf(stdout, "Enter a valid 2 digit hex number.\n\n");

continue;

}

/\* If user\_input is valid hexidecimal value then modify memory and increment the offset \*/

else{

sscanf(user\_input, "%X", &value);

fprintf(stdout, "New value is: %02X\n\n", value);

((char\*)memptr)[offset] = value;

offset++;

}

}

}

/\*\*

Simply Displays messages that the program is exiting

\*\*/

void quit()

{

fprintf(stdout, "\nExiting Program...\n\n");

}

/\*\*

Simply Displays the current value of all the registers

\*\*/

void displayRegisters()

{

fprintf(stdout, "R0\t\t R1\t\t R2\t\t R3\n");

fprintf(stdout, "%08X\t %08X\t %08X\t %08X\n", regs.R[0], regs.R[1], regs.R[2], regs.R[3]);

fprintf(stdout, "R4\t\t R5\t\t R6\t\t R7\n");

fprintf(stdout, "%08X\t %08X\t %08X\t %08X\n", regs.R[4], regs.R[5], regs.R[6], regs.R[7]);

fprintf(stdout, "R8\t\t R9\t\t R10\t\t R11\n");

fprintf(stdout, "%08X\t %08X\t %08X\t %08X\n", regs.R[8], regs.R[9], regs.R[10], regs.R[11]);

fprintf(stdout, "R12\t\t R13(SP)\t R14(LR)\t R15(PC)\n");

fprintf(stdout, "%08X\t %08X\t %08X\t %08X\n", regs.R[12], regs.SP, regs.LR, regs.PC);

fprintf(stdout, "MAR\t\t MBR\t\t SIGN\t\t CARRY\t\t ZERO\n");

fprintf(stdout, "%08X\t %08X\t %01X\t\t %01X\t\t %01X\n", regs.MAR, regs.MBR, regs.SIGN, regs.CARRY, regs.ZERO);

fprintf(stdout, "IR\t\t IR Flag\t IR0\t\t IR1\n");

fprintf(stdout, "%08X\t %01X\t\t %04X\t\t %04X\n", regs.IR, IR\_FLAG, regs.IR0, regs.IR1);

}

/\*\*

Responsible for running the instruction cycle and then displaying the registers

@param memory - Receives the Virtual CPU's buffer

\*\*/

void trace(void \*memptr)

{

instructionCycles(&memptr);

displayRegisters();

}

/\*\*

Function that creates a file of specified size

@param memory - Receives the Virtual CPU's buffer

\*\*/

void writeFile(void \*memptr)

{

char filename[50];

int fileSize = 0;

/\* Prompts user for File Name \*/

fprintf(stdout, "\nPlease Enter a File name:\t");

fscanf(stdin, "%s", filename);

getchar();

/\* Prompts user for File Size\*/

fprintf(stdout, "Please Enter A File Size:\t");

fscanf(stdin, "%d", &fileSize);

/\* Creates a file of the designated name and size \*/

FILE \*fp;

fp = fopen(filename, "w+");

for(int i = 0; i < fileSize; i++){

fprintf(fp, "\0");

}

fwrite(memptr, 1, fileSize, fp);

fclose(fp);

fprintf(stdout, "File Created Successfully\n");

}

/\*\*

Simply resets the value of all the registers to 0

\*\*/

void resetRegisters()

{

regs.IR = 0x00000000;

regs.ALU = 0x00000000;

for(int i = 0; i < REG\_NUM; i++){

regs.R[i] = 0x00000000;

}

regs.MAR = 0x00000000;

regs.MBR = 0x00000000;

regs.IR0 = 0x0000;

regs.IR1 = 0x0000;

regs.SIGN = 0x00;

regs.CARRY = 0x00;

regs.ZERO = 0x00;

STOP\_FLAG = false;

IR\_FLAG = false;

fprintf(stdout, "\nRegisters Have Been Reset\n");

}

/\*\*

Simply Displays the main menu when called

\*\*/

void help()

{

fprintf(stdout, "\nMain Menu\n");

fprintf(stdout, "-----------------------------------\n");

fprintf(stdout, "d)\tDump Memory\n");

fprintf(stdout, "g)\tGo - Run the Entire Program\n");

fprintf(stdout, "l)\tLoad a File into Memory\n");

fprintf(stdout, "m)\tMemory Modify\n");

fprintf(stdout, "q)\tQuit\n");

fprintf(stdout, "r)\tDisplay Registers\n");

fprintf(stdout, "t)\tTrace Execute One Instruction\n");

fprintf(stdout, "w)\tWrite File\n");

fprintf(stdout, "z)\tReset All Registers to Zero\n");

fprintf(stdout, "h)\tDisplay List of Commands\n");

}

/\*\*

Main Function that is responsible for receiving the users choice

@param argc - Currently not being used

@param argv - Currently not being used

\*\*/

int main(int argc, char \*argv[])

{

resetRegisters(); //ARM x86 resets registers when started

/\* While loop for main prompt checks user input \*/

while(1)

{

fprintf(stdout, "\nVirtual CPU ('h' help) >\t");

fscanf(stdin, "%s", &cmd);

if(cmd == 'd' || cmd == 'D'){

unsigned int myOffset = 0;

unsigned int myLength = 0;

fprintf(stdout, "\nPlease Enter an Offset:\t");

fscanf(stdin, "%d", &myOffset);

getchar();

fprintf(stdout, "\nPlease Enter a Length:\t");

fscanf(stdin, "%d", &myLength);

fprintf(stdout, "\n");

memDump(buffer, myOffset, myLength);

}

else if(cmd == 'g' || cmd == 'G'){

go(buffer);

}

else if(cmd == 'l' || cmd == 'L'){

int bytesRead;

bytesRead = loadFile(buffer, MEMORY\_MAX);

if(bytesRead != -1){

fprintf(stdout, "\nRead %u bytes (0x%04x in hex)\n", bytesRead, bytesRead);

}

}

else if(cmd == 'm' || cmd == 'M'){

unsigned int myOffset = 0;

fprintf(stdout, "\nPlease Enter an Offset:\t");

fscanf(stdin, "%d", &myOffset);

fprintf(stdout, "\n");

getchar();

memmod(buffer, myOffset);

}

else if(cmd == 'q' || cmd == 'Q'){

quit();

break;

}

else if (cmd == 'r' || cmd == 'R'){

displayRegisters();

}

else if(cmd == 't' || cmd == 'T'){

trace(buffer);

}

else if(cmd == 'w' || cmd == 'W'){

writeFile(buffer);

}

else if(cmd == 'z' || cmd == 'Z'){

resetRegisters();

}

else if(cmd == 'h' || cmd == 'H'){

help();

}

else {

fprintf(stdout, "\nCommand Invalid ('h' help)\n");

}

while(cmd != '\n'){

fscanf(stdin, "%c", &cmd);

}

}

return 0;

}