Table of Contents

[Brief description of the project 2](#_Toc449972373)

[Final state of the program 2](#_Toc449972374)

[Breakdown of Instruction Execution 3](#_Toc449972375)

[Sample output of Register Display and Memory Dump 5](#_Toc449972376)

[Testing 6](#_Toc449972377)

[Conclusions 6](#_Toc449972378)

[Appendix 7](#_Toc449972379)

[Rakul’s VCPU 7](#_Toc449972380)

[VCPU header 7](#_Toc449972381)

[VCPU MENU 9](#_Toc449972382)

[Load File 12](#_Toc449972383)

[Write File 13](#_Toc449972384)

[HEX Dump 14](#_Toc449972385)

[Memory Modify 15](#_Toc449972386)

[Display Registers 16](#_Toc449972387)

[Reset Registers 17](#_Toc449972388)

[Instruction Cycle 18](#_Toc449972389)

[Fetch 19](#_Toc449972390)

[Execute Instruction 20](#_Toc449972391)

[Load Registers 23](#_Toc449972392)

[Store Registers 24](#_Toc449972393)

[Flag Handler 25](#_Toc449972394)

[Check Carry 26](#_Toc449972395)

[Check Branch 27](#_Toc449972396)

# Brief description of the project

For my CENG 356 final project I have created a C program to perform registry level simulation of a CPU based on the ARM Cortex M0+ architecture. The CPU’s memory size is 16Kbytes and uses 32-bit registers. The CPU currently fetches 32-bits of instructions (2x 16-bit instructions) and executes the instructions accordingly. It also has other features such as load file, memory dump, modify memory and more, we provided more detail of functionalities in “Final Program State” section. For most reliable CPU use, we would recommend creating the instruction file using Hex Editor.

# Final state of the program

I have implemented the following functionalities to our Virtual CPU:

**Memory Dump:**

This function implements memory dump to our Virtual CPU. The interface code prompts

the user for an offset and length to be dumped from the CPU memory. The data dumped will

depend on what has been placed in memory by the load file function.

**Load File to Memory:**

This function prompts the user for the name of a file and reads it into a buffer. The

function will be passed a pointer to the buffer and a max number of bytes to be read from the file

(size of buffer) and will return the number of bytes successfully read. The program will then

display the number of bytes read from the file (in decimal and hex) and if the file was truncated.

**Memory Modify:**

This function asks the user to enter the starting address. Display the entered address and

the existing value at that location and prompt the user for a new value. If the input is a period (.),

function exits, if it is a hex value, change the value and increments the address, else increments

the address.

**Quit:**

Ends CPU menu loop and finishes CPU simulation.

**Display Registers:**

This function displays the content of all registers and status of all flags.

**Write File:**

This function prompts the user for the name of a file, prompts for the number of bytes to

write and creates the file. The function will be passed a pointer start of the data to be put in the

file.

**Reset Registers:**

This function clears the content of all registers and status of all flags.

**Help:**

This function lists the CPU commands and what they do.

**Trace:**

The trace command calls the the instructionCycle function and displayRegister function.

It determines and executes the instruction and determines if new line of instruction is to be

fetched. Depending on the current instruction, instructionCycle executes the execute function

and/or fetch function.

# Breakdown of Instruction Execution

Our CPU handles the execution of each 16-bit instruction through masking specified bits.

Bellow, we specify how our instruction execution is determined:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 0 | 0 | 0 | 0 | Operation | | | | Rn | | | | Rd | | | | Data Processing |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0 | 0 | 1 | 0 | L | B |  |  | Rn | | | | Rd | | | | Load/Store |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0 | 1 | OpCode | | 8-bit value | | | | | | | | Rd | | | | Immediate operations |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 0 | 0 | condition | | | | 8-bit relative address | | | | | | | | Conditional Branch |
|  |  |  |  |  | | | |  | | | | | | | |  |
| 1 | 1 | 0 | K | Offset12 | | | | | | | | | | | | Unconditional Branch |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  | Stop |

Data Processing: Bits 12-15 specify the instruction type (0000)

* Operations: Bits 8-11 specify Operation type
* ADD: add (0100)
* SUB: subtract (0010)
* ADC: add with carry (0101)
* Rn: Bits 4-7 specify the register that is used as the second operand
* Rd: Bits 0-3 specify the register that is used as one of the operands and as the destination of the result

Load Store: Bits 12-15 specify the instruction type (0010)

* L: Bit 11 specifies load or store
* Load: bit set (1)
* Store: bit not set (0)
* B: Bit 10 specifies if you are load/store a byte or a word
* Transfer word: Bit not set (0)
* Transfer byte: Bit set (1)
* Rn: Bits 4-7 specify the value that is used as the memory address in the transfer
* Rd: Bits 0-3 specify the Source/Destination Register

Immediate Operation: Bits 14-15 specify the instruction type (01)

* OpCode: Bits 12-13 specify OpCode
* MOV: move (00)
* CMP: compare (01)
* ADD: add (10)
* SUB: subtract (11)
* 8-bit value: Bits 4-11 specify the 8-bit immediate value
* Rd: Bits 0-3 specify the Source/Destination Register

Conditional Branch: Bits 12-15 specify the instruction type (1000)

* Condition: Bits 8-11 specify condition
* EQ: Equal (0000)
* NE: Not Equal (0001)
* CS: Unsigned Higher or Same (0010)
* CC: Unsigned Lower (0011)
* MI: Negative (0100)
* PL: Positive (0101)
* HI: Unsigned Higher (1000)
* LS: Unsigned Lower or Same (1001)
* AL: Always (1110)
* 8-bit relative address: Bits 0-7 specify the offset as an 8-bit relative address. The 8-bit signed value is added to the Program counter

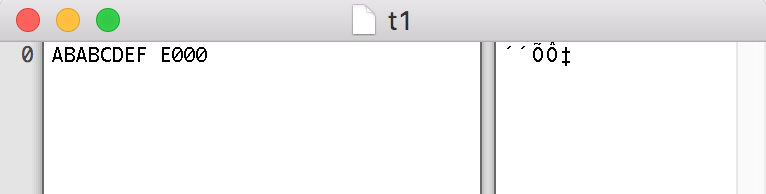
Unconditional Branch: Bits 13-15 specify the instruction type (110)

* K: Bit 12 specified link bit
* Branch: Bit not set (0)
* Branch with Link: Bit set (1)
* Offset12: Bits 0-11 specify the offset that is a 12-bit absolute memory location

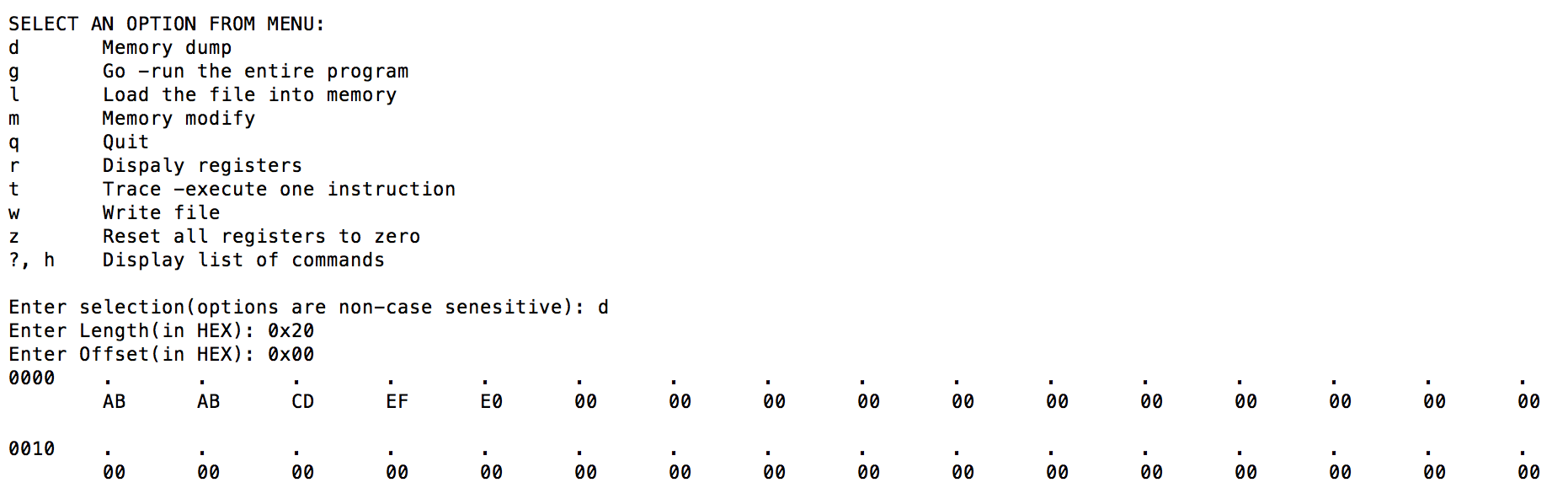
Stop: Bits 13-15 specify Stop instruction (111). Sets an internal stop flag which stops further instructions from being fetched

# Sample output of Register Display and Memory Dump

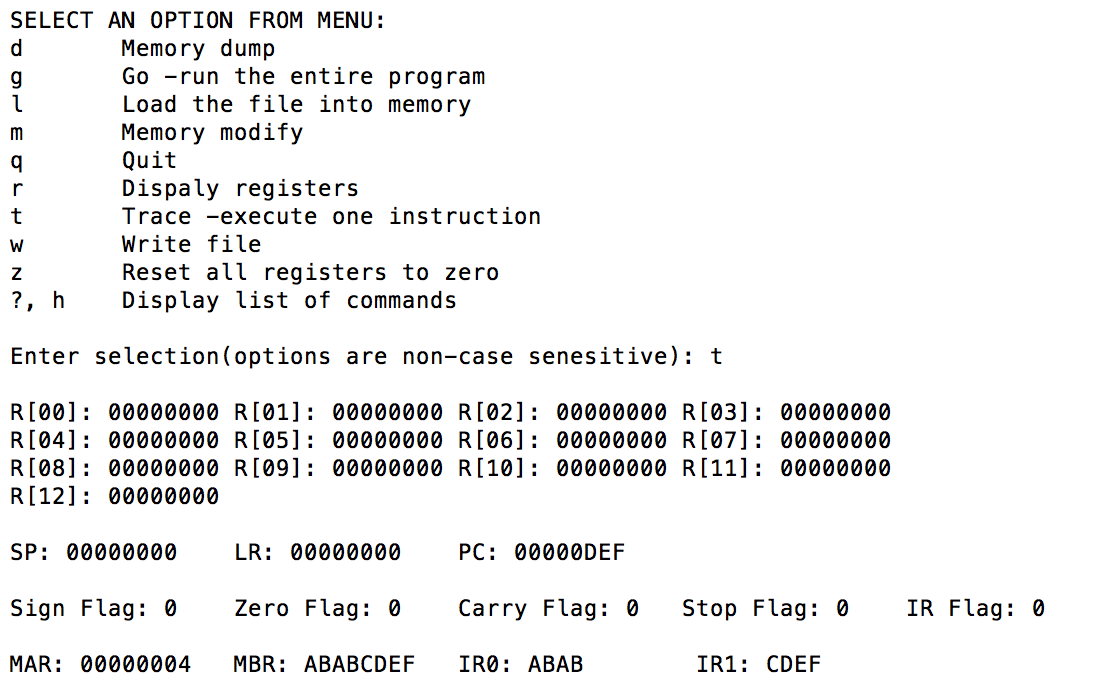
Content of HEX file loaded to CPU:



CPU displaying memory dump for file loaded:



CPU executing first trace and displaying registers:



# Testing

We tested the following scenarios to ensure the CPU’s correct functionality:

* CPU menu was tested to see if its constantly looping unless user quits
  + Checked if user input called correct function
* Load file is capable loading file to memory correctly and displaying file size
  + Determines if file was truncated
  + Determines if file does not exist
* Write file correctly created file of specified size
* HEX dump displayed correct length of data from specified offset
  + Non printable characters were tested to see if they were printed as ‘.’
  + Ensured maximum 16 columns of data was printed in hex dump
* Memory modify was check to see if it starts from specified offset and if offset was valid
  + Replaces existing value with entered hex value
  + Exits when user enter ‘.’
* Display registers displayed current state of all registers and flags
* Reset registers cleared all registers and flags
* Instructions cycle, checks and updates IR flag
  + If IR flag 0 fetches instruction and executes
  + If IR flag 1 executes instruction
* Fetch reads instruction to IR and MBR and update PC and mar
* Execute functions were tested successfully for correct functionality
  + Load word
  + Store word
  + Immediate ADD- add 2 to register 2 (HEX: 6022)
  + Immediate SUB- subtract 1 from register 2 (HEX: 7012)
  + Immediate MOV- move 1 to register 1 (HEX: 4011)
  + Immediate CMP- compare 6 with value in register 1 (HEX: 5061)
  + Data Process ADD- add registers 1 and 2 (HEX: 0412)
  + Data Process SUB- subtract registers 1 and 2 (HEX: 0212)
  + Data Process ADC
  + Unconditional Branch- unconditional CS branch to absolute location 1 (HEX: C001)
  + Conditional Branch- conditional CS branch to relative offset 1 (HEX: 8201)
  + Stop – HEX: E000

# Conclusions

To conclude the project, we were able to utilize our understanding of the ARM Cortex

M0+ architecture and C programing skills to create a functional Virtual CPU. We have

successfully designed and built our CPU to function according to the project requirements. Our

next goal is to continue to implement more functionality to our CPU, upon free time available.

Overall, the project was an excellent learning experience where we able to apply our course

understanding.

# Appendix

## Rakul’s VCPU

rmVCPU.h

### VCPU header

//Rakul Mahenthiran Virtual CPU header file

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <limits.h>

#include <float.h>

#include <ctype.h>

#include <stdint.h>

//General Defines

#define CPU\_MEM 0x4000

#define REG\_BIT\_SIZE 0x20

#define MAX32 0xFFFFFFFF

#define TOTAL\_REG\_NUM 0x10

#define IR1\_MASK 0xFFFF

#define SHIFT\_BYTE 8

#define SHIFT\_2\_BYTES 16

#define SHIFT\_3\_BYTES 24

#define THUMB 0x2

#define MASK\_BYTE 0xFF

//Special Rgisters Index

#define SP\_REG 0xD

#define LR\_REG 0xE

#define PC\_REG 0xF

//Imediate Instructions Values

#define MOV 0x0

#define CMP 0x1

#define ADD 0x2

#define SUB 0x3

//Data Process Instructions Values

#define DP\_ADD 0x4

#define DP\_SUB 0x2

#define DP\_ADC 0x5

//CPU Registers

static uint32\_t registers[TOTAL\_REG\_NUM];

static uint32\_t mar;

static uint32\_t mbr;

static uint32\_t ir;

static uint32\_t alu;

static uint16\_t cur\_inst;

//Flags

static uint8\_t sign\_flag;

static uint8\_t zero\_flag;

static uint8\_t carry\_flag;

static uint8\_t stop\_flag;

static uint8\_t ir\_flag;

#define IR0 (unsigned)ir >> 16

#define IR1 ir & IR1\_MASK

//Execute Function Types

#define PROCESS\_DATA 0x0

#define LOAD\_STORE 0x1

#define IMEDIATE1 0x2

#define IMEDIATE2 0x3

#define CONDITIONAL\_BRANCH 0x4

#define UNCONDITONAL\_BRANCH 0x6

#define STOP 0xE000

//Branch Conditions

#define EQ 0x0

#define NE 0x1

#define CS 0x2

#define CC 0x3

#define MI 0x4

#define PL 0x5

#define HI 0x8

#define LS 0x9

#define AL 0xE

//CPU Functions

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

void writeFile(void \*memory);

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

void memModify(void \*memptr,unsigned location);

void displayRegisters();

void zeroRegisters();

void instructionCycle(void \*memptr);

void fetch(void \*memptr);

void execute(void \*memptr);

uint32\_t loadRegisters(uint32\_t marVal, void \*memptr);

void storeRegisters(uint32\_t marVal, uint32\_t mbrVal, void \*memptr);

void flagHandler(uint32\_t alu);

int isCarry(uint32\_t op1,uint32\_t op2, uint8\_t c);

int checkBranch();

rmVCPU.c

### VCPU MENU

#include "rmVCPU.h"

int main(){

unsigned char memory[CPU\_MEM];

unsigned int bytesRead;

unsigned length, offset, memModAddress;

char option;

int loopRun = 1;

//Menu Format

printf("\n\n--Name: Rakul Mahenthiran--\n--Date: Mar 23, 2016--\n--Student#: 822240982--\n\n");

while(loopRun == 1){

printf("\nSELECT AN OPTION FROM MENU: ");

printf("\nd\tMemory dump");

printf("\ng\tGo -run the entire program");

printf("\nl\tLoad the file into memory");

printf("\nm\tMemory modify");

printf("\nq\tQuit");

printf("\nr\tDispaly registers");

printf("\nt\tTrace -execute one instruction");

printf("\nw\tWrite file");

printf("\nz\tReset all registers to zero");

printf("\n?, h\tDisplay list of commands");

printf("\n\nEnter selection(options are non-case senesitive): ");

scanf("%s", &option);

switch(option){

case 'd':

case 'D':

printf("Enter Length(in HEX): 0x");

scanf("%X", &length);

printf("Enter Offset(in HEX): 0x");

scanf("%X", &offset);

if((offset + length) >= CPU\_MEM)

printf("\nThe length and offset specified is out of memory boudary.\n");

else

memDump(memory, offset, length);

break;

case 'g':

case 'G':

printf("\n\nGo function yet to be implemented.");

break;

case 'l':

case 'L':

bytesRead = loadFile(memory, CPU\_MEM);

if(bytesRead != -1) { //checks file was opened

printf("Read %u bytes (0x%04x in hex)\n", bytesRead, bytesRead);

}

break;

case 'm':

case 'M':

printf("\n\nEnter starting address to modify(in HEX): 0x");

scanf("%X", &memModAddress);

if(memModAddress >= CPU\_MEM)

printf("\nThe offset specified is out of memory boudary.\n");

else

memModify(memory, memModAddress);

break;

case 'q':

case 'Q':

printf("\nGoodbye!\n");

loopRun = 0;

break;

case 'r':

case 'R':

displayRegisters();

break;

case 't':

case 'T':

instructionCycle(memory);

displayRegisters();

break;

case 'w':

case 'W':

writeFile(memory);

break;

case 'z':

case 'Z':

zeroRegisters();

break;

case '?':

case 'h':

case 'H':

printf("\nHelp Menu ->");

printf("\nEnter 'd' or 'D' for memory dump");

printf("\nEnter 'g' or 'G' to go -run the entire program");

printf("\nEnter 'l' or 'L' to load the file into memory");

printf("\nEnter 'm' or 'M' for memory modify");

printf("\nEnter 'q' or 'Q' to quit");

printf("\nEnter 'r' or 'R' to dispaly registers");

printf("\nEnter 't' or 'T' to trace -execute one instruction");

printf("\nEnter 'w' or 'W' to write file");

printf("\nEnter 'z' or 'Z' to reset all registers to zero");

printf("\nEnter '?', 'h' or 'H' to display list of commands");

printf("\n\nEnter selection: ");

break;

default:

printf("\nError, invalid option entered!\n");

}

while(option != '\n') {

scanf("%c", &option);

}

}

return(0);

}

### Load File

//Load file function

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

FILE \*fp;

char filename[24];

unsigned int readCount, fileSize;

printf("Enter Filename to load: ");

scanf("%s", filename);

fp = fopen(filename, "r");

if (fp != NULL) {

//checking file original size read to see if truncated

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

fileSize = ftell(fp);

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

if(fileSize > max){

printf("File has been truncated.\n");

}

//bytes read

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

fclose(fp);

}

else{

printf("\n\tERROR, file does not exist.\n");

return(-1);

}

return (readCount);

}

### Write File

//Wite file

void writeFile(void \*memory) {

char filename[24];

unsigned int fileSize;

int i;

printf("Enter Filename to write: ");

scanf("%s", filename);

printf("Enter File Size: ");

scanf("%d", &fileSize);

FILE \*fp;

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

fwrite(memory, 1, fileSize, fp);

fclose(fp);

}

### HEX Dump

//Hex Dump

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

{

char\* p = memptr;

int rows;

int x,y,z;

unsigned int decOffset = ("%d", offset);

unsigned int decLength = ("%d", length);

int remainder;

//determine how many rows need to printed in hex dump

remainder = decLength % 16;

if(remainder == 0)

rows = decLength / 16;

else

rows = (decLength / 16) + 1;

//looping through each row

for(x = 0; x < rows; x++){

//print offset at start of row

printf("%04X\t", decOffset + (x \* 16));

//print all 16 ascii columns for row

for(y = 0; y < 16; y++){

if((y + (x \* 16)) < decLength){

if(isprint(((char\*)p)[decOffset + y + (x \* 16)])){ //printable char

printf("%c\t",p[decOffset + y + (x \* 16)]);

}

else{ // other char

printf(".\t");

}

}

}

printf("\n\t");

//print all 16 hex columns for row

for(z = 0; z < 16; z++){

if((z + (x \* 16)) < decLength){

if((z + (x \* 16)) < decLength){

printf("%02X\t",0xFF & ((char\*)p)[decOffset + z + (x \* 16)]);

}

}

}

printf("\n\n");

}

}

### Memory Modify

//Memory Modify

void memModify(void \*memptr,unsigned location)

{

char\* p2 = memptr;

unsigned char value[256];

unsigned char exit = '.';

unsigned hexVal = 0x00;

while(1){

getchar(); //get rid of \n from scanf

//print offset and value at offset

printf("Address location 0x%04x contains hex value: 0x%02X", location, p2[location]);

printf("\nEnter value to replace with (in hex): 0x");

fgets(value,sizeof(value)-1,stdin);

//end loop if '.' is detected

if(value[0] == exit)

break;

sscanf(value," %X",&hexVal);

//change values at mem location

(((char\*)memptr)[location]) = hexVal;

//end loop if address full

if(++location == 16384)

break;

}

}

### Display Registers

//Dispaly Registers

void displayRegisters()

{

int i;

printf("\n");

for(i =0; i < TOTAL\_REG\_NUM; i++){

if(i < 13){

printf("R[%02d]: %08X\t", i, registers[i]);

}

else if(i == 13){

printf("\n\nSP: %08X\t", registers[i]);

}

else if(i == 14){

printf("LR: %08X\t", registers[i]);

}

else{

printf("PC: %08X\t", registers[i]);

}

if(((i+1)%4) == 0){

printf("\n");

}

}

printf("\nSign Flag: %d\tZero Flag: %d\tCarry Flag: %d\tStop Flag: %d\tIR Flag: %d", sign\_flag, zero\_flag, carry\_flag, stop\_flag, ir\_flag);

printf("\n\nMAR: %08X\tMBR: %08X\tIR0: %04X\t IR1: %04X\t\n", mar, mbr, IR0, IR1);

}

### Reset Registers

//Zero all Registers

void zeroRegisters(){

int i;

for(i = 0; i < TOTAL\_REG\_NUM; i++){

registers[i] = 0;

}

sign\_flag = 0;

zero\_flag = 0;

carry\_flag = 0;

stop\_flag = 0;

ir\_flag = 0;

mar = 0;

mbr = 0;

ir = 0;

cur\_inst = 0;

}

### Instruction Cycle

//depending on ir\_flag executes instruction and determine next fetch

void instructionCycle(void \*memptr){

if(ir\_flag == 0){

fetch(memptr);

cur\_inst = IR0;

execute(memptr);

ir\_flag = 1;

}

if(ir\_flag == 1){

cur\_inst = IR1;

execute(memptr);

ir\_flag = 0;

}

}

### Fetch

//Fetches 32-bits of instruction from file loaded. Fetches the 32-bits, 8 –bits at a time, as specified by the ARM Cortex M0 architecture. Updates MAR and PC

void fetch(void \*memptr){

int i;

mar = registers[PC\_REG];

for(i = 0; i < 4; i++, mar++){

mbr = mbr << 8;

mbr += \*((unsigned char\*)memptr + mar);

}

ir = mbr;

registers[PC\_REG] += 4;

}

### Execute Instruction

//Masks specified bits from instruction fetched and determines the instruction to be executed and its parameters

void execute(void \*memptr){

//Load Store Instruction

if(((unsigned)cur\_inst >> 13) == LOAD\_STORE){

//Load

if((cur\_inst >> 11) & 0x1){

//Load a Byte

if((cur\_inst >> 10) & 0x1){

registers[cur\_inst & 0xF] = loadRegisters(registers[(cur\_inst >> 4) & 0xF], memptr);

registers[cur\_inst & 0xF] = registers[cur\_inst & 0xF] & MASK\_BYTE;

}

//Load a Word

else{

registers[cur\_inst & 0xF] = loadRegisters(registers[(cur\_inst >> 4) & 0xF], memptr);

}

}

//Store

else{

mbr = registers[cur\_inst & 0xFF];

//Store a Byte to register

if((cur\_inst >> 10) & 0x1){

mar = registers[(cur\_inst >> 4) & 0xF];

mbr = registers[cur\_inst & 0xF];

\*((unsigned char\*)memptr + mar) = (unsigned char)mbr & MASK\_BYTE;

}

//Store a Word to Register

else{

storeRegisters(registers[(cur\_inst >> 4) & 0xF], registers[cur\_inst & 0xF], memptr);

}

}

}

//Imediate Instruction

else if((((unsigned)cur\_inst >> 13) == IMEDIATE1) | (((unsigned)cur\_inst >> 13) == IMEDIATE2)) {

//Move Imediate Value

if(((cur\_inst >> 12) & 0x3) == MOV){

registers[cur\_inst & 0xF] = ((cur\_inst >> 4) & 0xFF);

flagHandler(registers[cur\_inst & 0xF]);

}

//Compare Imediate Value

else if(((cur\_inst >> 12) & 0x3) == CMP){

alu = registers[cur\_inst & 0xF] + ~((cur\_inst >> 4) & 0xFF) + 1;

flagHandler(alu);

carry\_flag = isCarry(registers[(cur\_inst >> 12) & 0x3], ~((cur\_inst >> 4) & 0xFF), 0);

}

//Add Imediate Value

else if(((cur\_inst >> 12) & 0x3) == ADD){

alu = registers[cur\_inst & 0xF] + ((cur\_inst >> 4) & 0xFF);

flagHandler(alu);

carry\_flag = isCarry(registers[cur\_inst & 0xF], ((cur\_inst >> 4) & 0xFF), 0);

registers[cur\_inst & 0xF] = alu;

}

//Subtract Imediate Value

else if(((cur\_inst >> 12) & 0x3) == SUB){

alu = registers[cur\_inst & 0xF] + ~((cur\_inst >> 4) & 0xFF) + 1;

flagHandler(alu);

carry\_flag = isCarry(registers[cur\_inst & 0xF], ~((cur\_inst >> 4) & 0xFF), 1);

registers[cur\_inst & 0xF] = alu;

}

}

//Data Process Instruction

if(((unsigned)cur\_inst >> 13) == PROCESS\_DATA){

if(((cur\_inst >> 8) & 0xF) == DP\_ADD){

alu = registers[cur\_inst & 0xF] + registers[(cur\_inst >> 4) & 0xF];

flagHandler(alu);

carry\_flag = isCarry(registers[cur\_inst & 0xF], ~registers[(cur\_inst >> 4) & 0xF], 0);

registers[cur\_inst & 0xF] = alu;

}

else if(((cur\_inst >> 8) & 0xF) == DP\_SUB){

alu = registers[cur\_inst & 0xF] + ~registers[(cur\_inst >> 4) & 0xF] + 1;

flagHandler(alu);

carry\_flag = isCarry(registers[cur\_inst & 0xF], ~registers[(cur\_inst >> 4) & 0xF], 1);

registers[cur\_inst & 0xF] = alu;

}

else if(((cur\_inst >> 8) & 0xF) == DP\_ADC){

alu = registers[cur\_inst & 0xF] + registers[(cur\_inst >> 4) & 0xF] + carry\_flag;

flags(alu);

carry\_flag = iscarry(registers[cur\_inst & 0xF], registers[(cur\_inst >> 4) & 0xF], carry\_flag);

registers[cur\_inst & 0xF] = alu;

}

}

//Unonditional Branch Instruction

else if(((unsigned)cur\_inst >> 13) == UNCONDITONAL\_BRANCH){

if((cur\_inst >> 12) & 0x1){

registers[LR\_REG] = registers[PC\_REG];

}

registers[PC\_REG] = cur\_inst & 0xFFF;

// Make sure the IR flag is not still HI after the PC has changed.

ir\_flag = 0;

}

//Conditional Branch Instruction

else if(((unsigned)cur\_inst >> 13) == CONDITIONAL\_BRANCH){

// Check condition codes and flags

if(checkBranch()){

// Add relative address making sure its a signed 8-bit value

alu = registers[PC\_REG] + (int8\_t)(cur\_inst & 0xFF);

if(ir\_flag != 0){

ir\_flag = 0;

alu = alu + ~THUMB + 1;

}

registers[PC\_REG] = alu;

}

}

//Stop

else if(cur\_inst == 0xE000){

stop\_flag = 1;

}

}

### Load Registers

//Load Registers

uint32\_t loadRegisters(uint32\_t marVal, void \*memptr){

int i;

mar = marVal;

for(i = 0; i < 4; i++, mar++){

mbr = mbr << 8;

mbr += \*((unsigned char\*)memptr + mar);

}

return mbr;

}

### Store Registers

//Store Registers

void storeRegisters(uint32\_t marVal, uint32\_t mbrVal, void \*memptr){

mar = marVal;

mbr = mbrVal;

\*((unsigned char\*)memptr + mar++) = (unsigned char)(mbr >> 24 & 0xFF);

\*((unsigned char\*)memptr + mar++) = (unsigned char)(mbr >> 16 & 0xFF);

\*((unsigned char\*)memptr + mar++) = (unsigned char)(mbr >> 8 & 0xFF);

\*((unsigned char\*)memptr + mar) = (unsigned char)mbr & 0xFF;

}

### Flag Handler

//Flag Handler

void flagHandler(uint32\_t alu){

if(alu == 0){

zero\_flag = 1;

}else{

zero\_flag = 0;

}

sign\_flag = (alu & 0x80000000) >> 31;

}

### Check Carry

//Check Carry

int isCarry(uint32\_t op1,uint32\_t op2, uint8\_t c){

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

return(1);

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

}

### Check Branch

//Check condition code and flags to determine if branch needed

int checkBranch(){

if(((cur\_inst >> 8) & 0xF) == EQ){

if(zero\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == NE){

if(zero\_flag == 0){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == CS){

if(carry\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == CC){

if(!carry\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == MI){

if(sign\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == PL){

if(!sign\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == HI){

if(carry\_flag && zero\_flag == 0){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == LS){

if(carry\_flag == 0 || zero\_flag){

return 1;

}

}else if(((cur\_inst >> 8) & 0xF) == AL){

return 1;

}

return 0;

}