Virtual CPU Report

**Table of Contents**

[Introduction 3](#_Toc449034485)

[Registers 3](#_Toc449034486)

[Condition code register 4](#_Toc449034487)

[Instruction Flowchart 4](#_Toc449034488)

[Instruction Format 5](#_Toc449034489)

[OpCodes for Data Processing Instructions 6](#_Toc449034490)

[OpCodes for Immediate Instructions 6](#_Toc449034491)

[Branch Condition Codes 7](#_Toc449034492)

[Instruction Set Coding Sheet 8](#_Toc449034493)

[Sample output 9](#_Toc449034494)

[Testing 10](#_Toc449034495)

[Conclusion 11](#_Toc449034496)

[Appendix 11](#_Toc449034497)

[main.cpp 11](#_Toc449034498)

[myMenu.h 12](#_Toc449034499)

[myMenu.cpp 12](#_Toc449034500)

[myFunctions.h 14](#_Toc449034501)

[myFunctions.cpp 15](#_Toc449034502)

[myCPU.h 21](#_Toc449034503)

[myCPU.cpp 22](#_Toc449034504)

[makefile 27](#_Toc449034505)

# Introduction

This project consisted of simulating a simple CPU through a program written in C. It simulates the virtual CPU at a register level which allows the user to transfer data between registers and between registers and memory. The vCPU has 16 registers that are visible to the user and 16KB of memory.

The file that is loaded must be an ASCII based text file containing 4 character instructions representing 4 hex characters. Newline characters must be represented by 2 bytes (0x0A and 0x0D). The program follows big endian architecture and negative values are represented using 2’s compliment. Debugging provides unassembled instructions per bytecode.

# Registers

The following table shows the different registers, some of which are available to the user and some which are used by the vCPU.

|  |  |  |  |
| --- | --- | --- | --- |
| **Visible to the User** | **Register** | **Size (bits)** | **Purpose** |
| Yes | R0-R12 | 32 | General Purpose |
| Yes | R13 | 32 | Stack Pointer |
| Yes | R14 | 32 | Link Register |
| Yes | R15 | 32 | Program Counter |
| No | MBR | 32 | Memory Buffer |
| No | MAR | 32 | Memory Address |
| No | IR0/1 | 16 | Instruction Register |
| No | CCR | 8 | Condition Codes |

# Condition code register

The condition codes are represented using an 8 bit register and organized as follows.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Purpose** | Reserved | Debug | Reserved | Active IR | Stop | Carry | Zero | Sign |

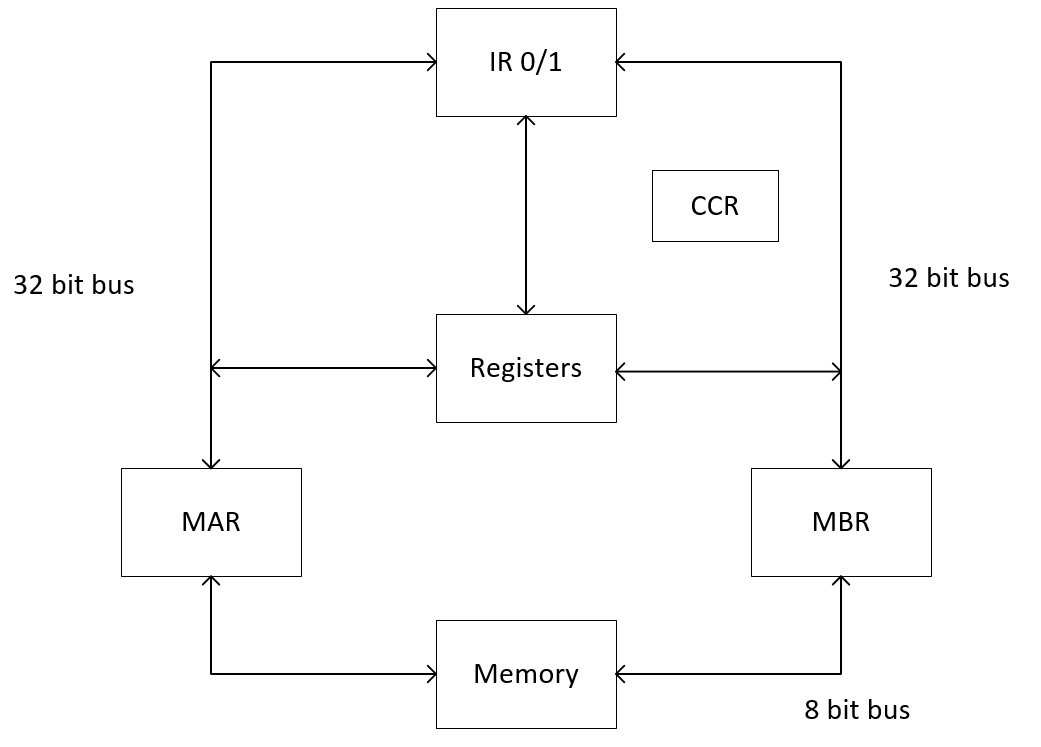
# Instruction Flowchart

The chart below shows the flow of information in the vCPU. An instruction cycle consists of one fetch and execute cycle.

During the fetch command:

1. MAR receives the address in PC
2. The instruction at MAR in Memory is copied to MBR 1 byte at a time.
3. The active IR is then updated with the instruction

During the execute command, the instruction held in the active IR is executed and the IR flag is toggled. A trace command consists of one instruction cycle while the go command continuously performs instruction cycles until the stop flag is set.



# Instruction Format

The following table shows how each bytecode instruction is constructed.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |  |
|  | 0 | 0 | 0 | 0 | Operation | | | | Rn | | | | Rd | | | | Data Processing |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 0 | 0 | 1 | 0 | L | B | X | X | 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 | 12-bit Offset | | | | | | | | | | | | Unconditional Branch |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Stop |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Data Processing Instructions

* The Operation field specifies the logical or arithmetic operation.
* The Rd register is used as one of the operands and as the destination of the result.
* The Rn register is used as the second operand.

Load/Store Instructions

* Rn - value is used as the memory address in the transfer
* Rd - Source/Destination Register
* L - Load/Store bit: 0 = Store to memory , 1 = Load from Memory
* B - Byte/Word bit: 0 = transfer Word, 1= transfer byte

Immediate Instructions

* The immediate value is an 8-bit value (zero extended).

Conditional Branch Instructions

* The offset is an 8-bit relative address. The 8-bit signed value is added to the Program counter.

Unconditional Branch Instruction

* The offset is a 12-bit absolute memory location

Stop Instruction

* Sets an internal stop flag which stops further instructions from being fetched.
* Used to return control to user interface when a program is run.

# OpCodes for Data Processing Instructions

The following codes define which Data processing instruction is required

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Code** | **Description** | **Flags (ncz)** |
| AND | 0000 | Rd = Rd AND Rn | n\_z |
| SUB | 0001 | Rd = Rd – Rn | n\_z |
| ADD | 0100 | Rd = Rd + Rn | ncz |
| MOV | 1101 | Rd = Rn | n\_z |

# OpCodes for Immediate Instructions

The following code defines which Immediate Instruction is required.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Code** | **Description** | **Flags (ncz)** |
| MOV | 00 | Rd = immediate value | n\_z |

# Branch Condition Codes

The codes define which condition is checked for each conditional branch.

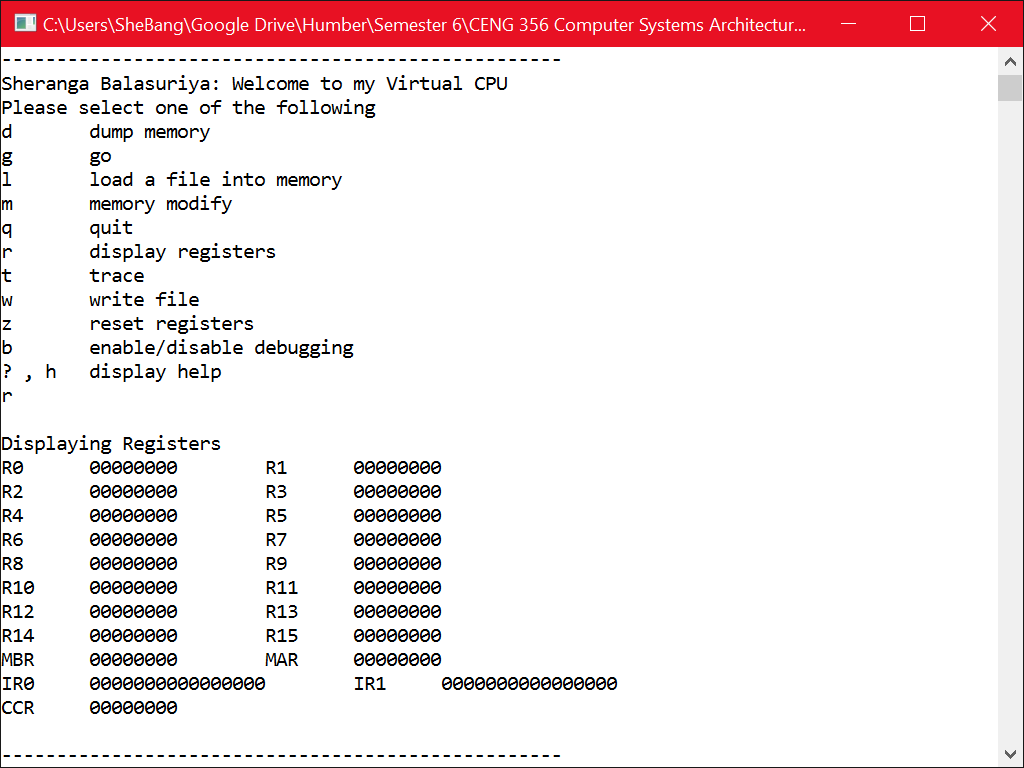
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Suffix** | **Code (bits)** | **Code (hex)** | **Flags** | **Meaning** |
| EQ | 0000 | 0 | Z set | Equal |
| NE | 0001 | 1 | Z clear | Not Equal |
| CS | 0010 | 2 | C set | unsigned higher or same |
| CC | 0011 | 3 | C clear | unsigned lower |
| MI | 0100 | 4 | N set | negative |
| PL | 0101 | 5 | N clear | positive |
| HI | 1000 | 8 | C set and Z clear | unsigned higher |
| LS | 1001 | 9 | C clear or Z set | unsigned lower or same |
| AL | 1110 | E | Ignored | Always |

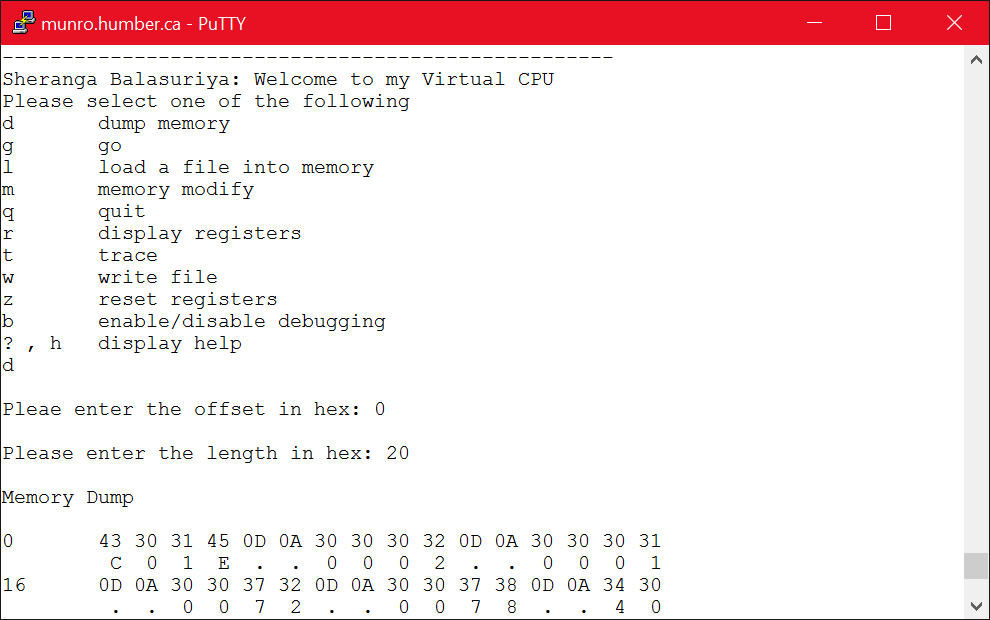
# Instruction Set Coding Sheet

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Opcode** | **Instruction** | **Coding (Hex)** | **Description** | **Flags** | **Example** |
| ADD | Add | 04nd | Rd := Rd + Rn | ncz | ADD r1,r2 |
| AND | And | 00nd | Rd := Rd AND Rn | n-z | AND r3,r12 |
| BRA | Branch | Cooo | PC := offset | --- | BRA next |
| BXX | Conditional branch | 8xoo | PC := PC+offset if true | --- | BNE again |
| LDR | Load register | 28nd | Rd := [Rn] | --- | LDR r2,[r7] |
| MOV | Move immediate | 4iid | Rd := immediate | n-z | MOV r1,#3A |
| STP | Stop | E000 | Set internal Stop flag | --- | STP |
| STR | Store register | 20nd | [Rn] := Rd | --- | STR r2,[r7] |
| SUB | Subtract | 02nd | Rd := Rd - Rn | ncz | SUB r1,r2 |
| Notes  ooo - offset – 12-bit absolute memory address  oo - 8-bit relative address  n - Rn register number  d - Rd register number (destination)  ii - 8 bit immediate value  x - Condition Code | | | | | |

# Sample output

The original code was written in Visual Studio 2015 in x86 on an x64 Windows 10 machine. It was also tested on an x86 Linux machine with minor adjustments to the for loops for C99 compliance. The following screenshots show the display registers and memory dump commands.





# Testing

The following program was used to test the virtual CPU. It is a simple program that loops 2 times, decrementing 1 each loop. It add the decrement to a random variable and at the end stores the final value in memory. It uses each available instruction in the current form of the vCPU.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; loopSample – sample program for VCPU-2016

; Loop skeleton

; Sheranga Balasuriya, April 14, 2016

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0000 C01E Start BRA MAIN

0006 0002 LoopLen dw 5 ; # of times to loop

000C 0001 LoopDec dw 1 ; loop decrement

0012 0072 Aptr dw Num1

0018 0078 Bptr dw Result

001E 4060 MAIN MOV R0, #LoopLen ; load counter pointer

0024 2802 LDR R2, [R0] ; load counter value

002A 40C0 MOV R0, #LoopDec ; load decrement pointer

0030 2803 LDR R3, [R0] ; load loop decrement

0036 4120 MOV R0, #Aptr ; load pointer

003C 2804 LDR R4, [R0] ; load num1 pointer

0042 2845 LDR R5, [R4] ; load sample number

0048 0435 AGAIN ADD R5, R3 ; sample operation

004E 0232 SUB R2, R3 ; decrement counter

0054 81F3 BNE AGAIN ; do again if not zero

005A 4180 MOV R0, #Bptr ; load pointer address

0060 2806 LDR R6, [R0] ; load pointer

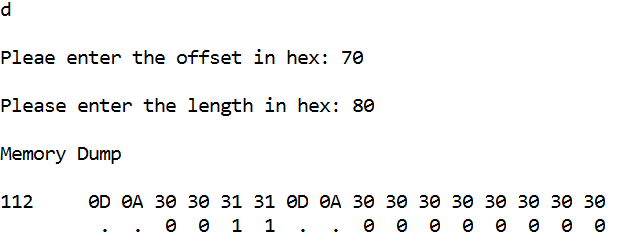
0066 2056 STR R5, [R6] ; store sample result

006C E000 STP

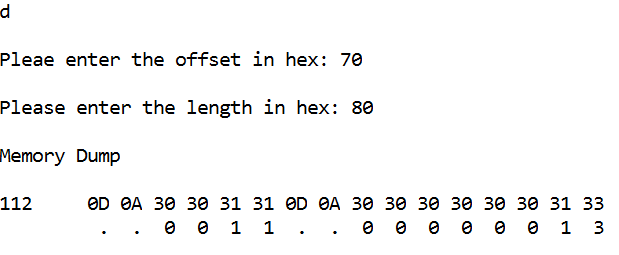
0072 0011 Num1 dw 0011

0078 0000 Result dw 0000

The following screenshot shows memory before running program



The following screenshot shows memory after running program. The program looped twice adding 1 to 0x11 each time. The end result (0x13) was then store at 0x78.



# Conclusion

While the basic program works there are many instructions left to be added. Using a binary file instead of an ASCII based text file would have been more efficient. As it stands the virtual CPU is a good representation of how real CPU’s work, especially ARM based architecture which is mainly based on register to register instructions.

# Appendix

The following is a full listing of the code for this virtual CPU. The code is separated into 3 main components. myMenu consists of the menu the user interacts with and its functionality. myFunctions consists of the functionality behind each command the user can invoke. myCPU contains the vCPU functionality. The registers are declared as a structure while the memory is declared as a character array. The included makefile was used for compilation purposes on the UNIX environment.

## main.cpp

#include "myMenu.h"

int main() {

menu();

return 0;

}

## myMenu.h

#pragma once

#ifndef MYMENU\_H

#define MYMENU\_H

#include <stdint.h>

// constants

#define MEMSIZE 16000

// function prototypes

void menu(void);

// declare memory

extern char memory[MEMSIZE];

// declare registers

// Reserved, Debugging, Reserved, IR, Stop, Carry, Zero, Sign

struct Registers

{

uint32\_t gen[16];

uint32\_t mbr;

uint32\_t mar;

uint16\_t ir[2];

uint8\_t ccr;

};

#endif // !MYCPU\_H#pragma once

## myMenu.cpp

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include "myMenu.h"

#include "myFunctions.h"

#include "myCPU.h"

/\*\*

Main menu

\*\*/

void menu() {

int instrReturn;

char choice;

int bytesRead;

unsigned offset, length;

// initialize memory, declared in myMenu.h

char memory[MEMSIZE];

// initialize registers and reset, declared in myMenu.h

struct Registers myRegisters;

reset(&myRegisters);

while (1) {

// display menu

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

fprintf(stdout, "\nSheranga Balasuriya: Welcome to my Virtual CPU\nPlease select one of the following");

fprintf(stdout, "\nd\tdump memory");

fprintf(stdout, "\ng\tgo");

fprintf(stdout, "\nl\tload a file into memory");

fprintf(stdout, "\nm\tmemory modify");

fprintf(stdout, "\nq\tquit");

fprintf(stdout, "\nr\tdisplay registers");

fprintf(stdout, "\nt\ttrace");

fprintf(stdout, "\nw\twrite file");

fprintf(stdout, "\nz\treset registers");

fprintf(stdout, "\nb\tenable/disable debugging");

fprintf(stdout, "\n? , h\tdisplay help\n");

fscanf(stdin, " %1c", &choice);

flush();

// check choice taken

switch (choice) {

case 'd':

case 'D':

// get offset from user

fprintf(stdout, "\nPleae enter the offset in hex: ");

fscanf(stdin, "%x", &offset);

flush();

// get length from user

fprintf(stdout, "\nPlease enter the length in hex: ");

fscanf(stdin, "%x", &length);

flush();

memDump(memory, offset, length);

break;

case 'g':

case 'G':

go(&myRegisters,memory);

break;

case 'l':

case 'L':

bytesRead = loadFile(memory, MEMSIZE);

if (bytesRead > 0)

fprintf(stdout, "\n%d(0x%X) bytes read into memory", bytesRead, bytesRead);

break;

case 'm':

case 'M':

modify(memory);

break;

case 'q':

case 'Q':

return;

break;

case 'r':

case 'R':

showRegisters(&myRegisters);

break;

case 't':

case 'T':

instrReturn = instructionCycle(&myRegisters, memory);

showRegisters(&myRegisters);

break;

case 'w':

case 'W':

writeFile(memory);

break;

case 'z':

case 'Z':

reset(&myRegisters);

break;

case 'b':

case 'B':

debug(&myRegisters);

break;

case '?':

case 'h':

case 'H':

help();

break;

default:

fprintf(stdout, "\nIncorrect entry, try again");

}

}

}

## myFunctions.h

#pragma once

#ifndef MYFUNCTIONS\_H

#define MYFUNCTIONS\_H

#include <stdint.h>

// define: verify active ir

#define ACTIVE\_IR ((myRegisters->ccr >> 5) & 1)

// function prototypes

void flush(void);

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

void go(struct Registers \* myRegisters, void \* memory);

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

void modify(void \* memory);

void writeFile(void \* memory);

void reset(struct Registers \* myRegisters);

void showRegisters(struct Registers \* myRegisters);

void printBinary16(uint16\_t x);

void printBinary8(uint8\_t x);

int instructionCycle(struct Registers \* myRegisters, void \* memory);

void fetch(struct Registers \* myRegisters, void \* memory);

void debug(struct Registers \* myRegisters);

void help(void);

#endif

## myFunctions.cpp

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include "myMenu.h"

#include "myFunctions.h"

#include "myCPU.h"

/\*\*

flush the input buffer

\*\*/

void flush() {

int c;

while ((c = getchar()) != '\n' && c != EOF);

}

/\*\*

dump the contents of memory

memptr: pointer to memory

offset: number of bytes offset to start

length: number of bytes to show

\*\*/

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

fprintf(stdout, "\nMemory Dump\n");

// check if offset too large

if ((offset > MEMSIZE) || (offset<0)) {

fprintf(stdout, "\nOffset error, printing from beginnig.");

offset = 0;

}

// check if length too large

if (((offset + length) > MEMSIZE) || (length < 1)) {

fprintf(stdout, "\nLength too large, dumping all memory starting from offset.");

length = MEMSIZE - offset;

}

int i = offset;

char data;

while (i < length) {

// print line of hex code

fprintf(stdout, "\n%d\t", i);

for (int i2 = 0;i2 < 16;i2++) {

if ((i + i2) == length)

break;

// retrieve character

data = \*((char \*)memptr + i + i2);

// print character

fprintf(stdout, "%.2X ", data);

}

// print line of ASCII code

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

for (int i2 = 0;i2 < 16;i2++) {

if ((i + i2) == length)

break;

// retrieve character and verify if printable

data = \*((char \*)memptr + i + i2);

if ((data < 33) || (data > 126))

data = '.';

// print character

fprintf(stdout, " %c ", data);

}

i += 16;

}

}

/\*\*

Execute instructions in memory until stop

myRegisters: pointer to register structure

memory: pointer to memory

\*\*/

void go(struct Registers \* myRegisters, void \* memory) {

fprintf(stdout, "\nRun");

while (instructionCycle(myRegisters, memory) != 2);

}

/\*\*

Load contents of file into memory

memory: pointer to memory

max: maximum bytes to read

returns: number of bytes read

\*\*/

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

FILE \*fp;

char fileName[21];

long fileSize;

size\_t result;

fprintf(stdout,"\nLoad");

//prompt for filename

fprintf(stdout, "\nPlease enter file name (20 character max): ");

fscanf(stdin, "%20s", fileName);

flush();

//open file and read to memory

fp = fopen(fileName, "rb");

if (fp == NULL) {

fprintf(stdout, "Error opening file %s", fileName);

return -1;

}

else {

//determine file size

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

fileSize = ftell(fp);

if (fileSize > max)

fprintf(stdout, "File is %ld bytes long, only %d(x%X) bytes will be read",fileSize,max-1,max-1);

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

//read contents of file into memory

result = fread(memory, sizeof(char), max - 1, fp);

fclose(fp);

return result;

}

}

/\*\*

Modify contents of memory

memory: pointer to memory

\*\*/

void modify(void \* memory) {

unsigned address;

char data;

char update[3] = { '\0' };

fprintf(stdout, "\nMemory Modify\n");

// prompt user for starting address

do {

fprintf(stdout, "\nPlease enter a starting address in hex(0-%x): ", MEMSIZE);

fscanf(stdin, "%x", &address);

flush();

} while ((address < 0) || (address > MEMSIZE));

fprintf(stdout, "\nEnter Hex value to update (0-FF), 2E to exit, any other value to move to next address");

fprintf(stdout, "\naddr\tvalue\tupdated\n");

// show specified address

while (1) {

// reset input string

update[0] = '\0';

update[1] = '\0';

// verify if memory is printable

data = \*((char \*)memory);

if ((data < 33) || (data > 126))

data = '.';

// prompt user for updated value

fprintf(stdout, "%.2X\t%c\t", address, data);

fscanf(stdin, "%2s", &update);

flush();

// check what user has entered

if ( (isxdigit(update[0])) && (isxdigit(update[1])) ) {

\*(char \*)memory = strtol(update,NULL,16);

memory = (char \*)memory + sizeof(char);

address++;

}

else if (strcmp(update,".")==0) {

return;

}

else {

memory = (char \*)memory + sizeof(char);

address++;

}

}

}

/\*\*

Write the contents of memory to a file

memory: pointer to memory

\*\*/

void writeFile(void \* memory) {

FILE \*fp;

char fileName[21];

int nBytes = 0;

size\_t result;

fprintf(stdout,"\nWrite");

//prompt for filename

fprintf(stdout, "\nPlease enter file name (20 character max): ");

fscanf(stdin, "%20s", fileName);

flush();

//prompt for number of bytes to write

do {

fprintf(stdout, "Please enter number of bytes to write (less than %d): ",MEMSIZE);

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

flush();

} while (nBytes>MEMSIZE);

//open file and write memory

fp = fopen(fileName, "wb");

if (fp == NULL) {

fprintf(stdout, "Error opening file %s", fileName);

return;

}

else {

result = fwrite(memory, sizeof(char), nBytes, fp);

fclose(fp);

}

//check if successfully written

if (result > 0)

fprintf(stdout, "%zu Bytes successfully written to %s", result, fileName);

else

fprintf(stdout, "Error writing to file");

}

/\*\*

Reset registers

myRegisters: pointer to register structure

\*\*/

void reset(struct Registers \* myRegisters) {

fprintf(stdout, "\nResetting Registers");

for (int i = 0;i < 16;i++)

myRegisters->gen[i] = 0;

myRegisters->mbr = 0;

myRegisters->mar = 0;

myRegisters->ir[0] = 0;

myRegisters->ir[1] = 0;

myRegisters->ccr = 0;

}

/\*\*

Show registers

myRegisters: pointer to register structure

\*\*/

void showRegisters(struct Registers \* myRegisters) {

fprintf(stdout, "\nDisplaying Registers");

// display general registers

for (int i = 0;i < 16;i += 2) {

fprintf(stdout, "\nR%d\t%.8X\tR%d\t%.8X", i, myRegisters->gen[i], i + 1, myRegisters->gen[i + 1]);

}

// display rest of registers

fprintf(stdout, "\nMBR\t%.8X\tMAR\t%.8X",myRegisters->mbr,myRegisters->mar);

fprintf(stdout, "\nIR0\t");

printBinary16(myRegisters->ir[0]);

fprintf(stdout, "\tIR1\t");

printBinary16(myRegisters->ir[1]);

// TODO split ccr flag prints

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

printBinary8(myRegisters->ccr);

}

// TODO combine binary prints

/\*\*

Print Binary represenation of 16 bit Int

x: int to be printed

\*\*/

void printBinary16(uint16\_t x) {

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

if (x & 0x8000)

fprintf(stdout, "1");

else

fprintf(stdout, "0");

x = x << 1;

}

}

/\*\*

Print Binary represenation of 8 bit Int

x: int to be printed

\*\*/

void printBinary8(uint8\_t x) {

for (int i = 0; i < 8;i++) {

if (x & 0x80)

fprintf(stdout, "1");

else

fprintf(stdout, "0");

x = x << 1;

}

}

/\*\*

Instruction Cycle

Consists of one fetch and execute command

myRegisters: pointer to register structure

memory: pointer to memory structure

\*\*/

int instructionCycle(struct Registers \* myRegisters, void \* memory) {

fetch(myRegisters, memory);

return execute(myRegisters, memory);

}

/\*\*

Fetch new instruction from memory

myRegisters: pointer to register structure

memory: pointer to memory structure

\*\*/

void fetch(struct Registers \* myRegisters, void \* memory) {

char value;

// reset mbr and active ir

myRegisters->mbr = 0;

myRegisters->ir[ACTIVE\_IR] = 0;

// testing

//myRegisters->gen[0] = 0xA;

//myRegisters->gen[1] = 0xC;

// load new memory address

myRegisters->mar = myRegisters->gen[15];

memory = (char \*)memory + (myRegisters->mar \* sizeof(char));

// load 4 bytes into mbr from memory at mar

for (int i = 0; i < 4;i++, myRegisters->mar++) {

value = \*((char \*)memory);

myRegisters->mbr += strtol(&value, NULL, 16);

memory = (char \*)memory + sizeof(char);

if (i<3)

myRegisters->mbr = myRegisters->mbr << 4;

}

// load instruction into active ir

myRegisters->ir[ACTIVE\_IR] = myRegisters->mbr;

}

/\*\*

Enable/disable debugging features

myRegisters: pointer to register structure

\*\*/

void debug(struct Registers \* myRegisters) {

if ((myRegisters->ccr >> 6) && 0x1)

fprintf(stdout, "debugging disabled");

else

fprintf(stdout, "debugging enabled");

myRegisters->ccr ^= 1 << 6;

}

/\*\*

Display help

\*\*/

void help() {

fprintf(stdout,"\nHelp");

fprintf(stdout, "\nPlease choice from one of options.");

fprintf(stdout, "\nDump Memory: dump user defined area of memory");

fprintf(stdout, "\nRun: run program starting at memory 0x0");

fprintf(stdout, "\nLoad: load user defined file");

fprintf(stdout, "\nModify: directly modify user specified area of memory");

fprintf(stdout, "\nQuit: end virtual CPU program");

fprintf(stdout, "\nDisplay: show registers");

fprintf(stdout, "\nTrace: execute one instruction at a time starting at 0x0");

fprintf(stdout, "\nWrite: write contents of memory to file");

fprintf(stdout, "\nReset: reset all the registers to zero");

fprintf(stdout, "\nDebugging: enable showing of debugging information");

fprintf(stdout, "\nHelp: shows this description");

// TODO add more help

}

## myCPU.h

#pragma once

#ifndef MYCPU\_H

#define MYCPU\_H

// define current instruction

#define CIR myRegisters->ir[ACTIVE\_IR]

// define instructions

#define STP 0xE000

#define BRA 0xC000

#define MOV 0x4000

#define ADD 0x0400

#define LDR 0x2800

#define STR 0x2000

#define AND 0x0000

#define BXX 0x8000

#define SUB 0x0200

// define operands

#define BRA\_OP 0xFFF

#define RD 0xF

#define RN 0xF0

#define IMM 0xFF0

// define conditional branches

#define BRA\_EQ 0x80

#define BRA\_NE 0x81

#define BRA\_CS 0x82

#define BRA\_CC 0x83

#define BRA\_MI 0x84

#define BRA\_PL 0x85

#define BRA\_HI 0x88

#define BRA\_LS 0x89

#define BRA\_AL 0x8E

// define other

#define MAX32 0xFFFFFFFF

// function prototypes

int execute(struct Registers \* myRegisters, void \* memory);

void flags(struct Registers \* myRegisters, int reg);

void updatePC(struct Registers \* myRegisters, uint8\_t x);

#endif // !MYCPU\_H

## myCPU.cpp

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include "myCPU.h"

#include "myMenu.h"

#include "myFunctions.h"

/\*\*

execution of one instruction

myRegisters: pointer to register structure

memory: pointer to memory structure

\*\*/

int execute(struct Registers \* myRegisters, void \* memory) {

// TODO check flags

//printf("\nExecute");

char value;

void \* memory2;

// check which instruction

if (CIR == STP) {

// stop instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nstop");

myRegisters->ccr |= 1 << 4;

return 2;

}

else if ((CIR & BRA) == BRA) {

// branch instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nbranch");

myRegisters->gen[15] = CIR & BRA\_OP;

// flip ir flag and return

myRegisters->ccr ^= 1 << 5;

return 0;

}

else if ((CIR & MOV) == MOV) {

// move instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nmove");

myRegisters->gen[CIR & RD] = (CIR & IMM) >> 4;

myRegisters->gen[15] = myRegisters->mar + 2;

// check and set flags

flags(myRegisters, CIR & RD);

}

else if ((CIR & LDR) == LDR) {

// load instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nload");

memory = (char \*)memory + myRegisters->gen[((CIR & RN) >> 4)];

myRegisters->gen[CIR & RD] = 0;

// load new data from memory

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

value = \*((char \*)memory);

myRegisters->gen[CIR & RD] += strtol(&value, NULL, 16);

memory = (char \*)memory + sizeof(char);

if (i<3)

myRegisters->gen[CIR & RD] <<= 4;

}

// check and set flags

flags(myRegisters, CIR & RD);

}

else if ((CIR & STR) == STR) {

// store instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("store");

memory2 = (char \*)memory + myRegisters->gen[CIR & RD];

uint32\_t num = myRegisters->gen[((CIR & RN) >> 4)];

sprintf((char \*)memory2, "%X", (num & 0xF0000000)>>28);

sprintf((char \*)memory2 + 1, "%X", (num & 0xF000000)>>24);

sprintf((char \*)memory2 + 2, "%X", (num & 0xF00000) >> 20);

sprintf((char \*)memory2 + 3, "%X", (num & 0xF0000) >> 16);

sprintf((char \*)memory2 + 4, "%X", (num & 0xF000) >> 12);

sprintf((char \*)memory2 + 5, "%X", (num & 0xF00) >> 8);

sprintf((char \*)memory2 + 6, "%X", (num & 0xF0) >> 4);

sprintf((char \*)memory2 + 7, "%X", num & 0xF);

}

else if ((CIR & BXX) == BXX) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nbranch: %x", CIR >> 8);

// conditional branch instruction

if ((CIR >> 8) == BRA\_NE) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nbranch not equal");

if (!(myRegisters->ccr >> 1) & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_CS) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nunsigned higher or same");

if ((myRegisters->ccr >> 2) & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_CC) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nunsigned lower");

if (!(myRegisters->ccr >> 2) & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_MI) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nnegative");

if (myRegisters->ccr & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_PL) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\npositive");

if ((myRegisters->ccr) & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_HI) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nunsigned higher");

if (((myRegisters->ccr >> 2) & 1) && (!(myRegisters->ccr >> 1) & 1))

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_LS) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nunsigned lower or same");

if (!((myRegisters->ccr >> 2) & 1) && ((myRegisters->ccr >> 1) & 1))

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

else if ((CIR >> 8) == BRA\_AL) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nalways");

updatePC(myRegisters, CIR & 0xFF);

}

else if ((CIR >> 8) == BRA\_EQ) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nbranch if equal");

if ((myRegisters->ccr >> 2) & 1)

updatePC(myRegisters, CIR & 0xFF);

else

myRegisters->gen[15] = myRegisters->mar + 2;

}

myRegisters->ccr ^= 1 << 5;

return 0;

}

else if ((CIR & ADD) == ADD) {

// add instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nadd");

// check and set carry flag

if ((myRegisters->gen[((CIR & RN) >> 4)] == MAX32) ||

(myRegisters->gen[CIR & RD] > (MAX32 - myRegisters->gen[((CIR & RN) >> 4)])))

myRegisters->ccr |= 1 << 3;

// add registers

myRegisters->gen[CIR & RD] = myRegisters->gen[((CIR & RN) >> 4)] + myRegisters->gen[(CIR & RD)];

// check and set flags

flags(myRegisters, CIR & RD);

}

else if ((CIR & SUB) == SUB) {

// sub instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nsub");

//check and set negative flag

if((myRegisters->gen[CIR & RD])>myRegisters->gen[((CIR & RN) >> 4)])

myRegisters->ccr |= 1 << 1;

// sub registers

myRegisters->gen[CIR & RD] = myRegisters->gen[(CIR & RD)] - myRegisters->gen[((CIR & RN) >> 4)];

// check and set flags

flags(myRegisters, CIR & RD);

}

else if ((CIR & AND) == AND) {

// and instruction

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nand");

myRegisters->gen[CIR & RD] &= myRegisters->gen[((CIR & RN) >> 4)];

// check and set flags

flags(myRegisters, CIR & RD);

}

else {

if ((myRegisters->ccr >> 6) && 0x1)

printf("General Instruction Error");

return 1;

}

// flip ir flag, update pc, and return

myRegisters->ccr ^= 1 << 5;

myRegisters->gen[15] = myRegisters->mar + 2;

return 0;

}

/\*\*

Set the flags

myRegisters: pointer to register structure

reg: register that was used

\*\*/

void flags(struct Registers \* myRegisters, int reg) {

// check and set zero flag

if (myRegisters->gen[reg] == 0)

myRegisters->ccr |= 1 << 1;

else

myRegisters->ccr &= ~(1 << 1);

// check and set negative flag

if ((myRegisters->gen[reg] & 0x80000000) == 0x80000000)

myRegisters->ccr |= 1 << 0;

else

myRegisters->ccr &= ~(1 << 0);

}

/\*\*

Update PC counter

myRegisters: pointer to register structure

x: offset for PC

\*\*/

void updatePC(struct Registers \* myRegisters, uint8\_t x) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nrelative address: %X", x);

// check if negative

if ((x & 0x80) == 0x80) {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\nnegative: %X", ~x);

myRegisters->gen[15] -= (~x & 0xFF);

}

else {

if ((myRegisters->ccr >> 6) && 0x1)

printf("\npositive");

myRegisters->gen[15] += x;

}

}

## makefile

main: main.o myFunctions.o myMenu.o myCPU.o

gcc - g - o main main.o myFunctions.o myCPU.o myMenu.o

main.o : main.c

gcc - g - c main.c

myFunctions.o : myFunctions.c myFunctions.h

gcc - g - c myFunctions.c

myMenu.o : myMenu.c myMenu.h

gcc - g - c myMenu.c

myCPU.o : myCPU.c myCPU.h

gcc - g - c myCPU.c