Tel Aviv University

16.01.2023

COMPUTER ORGANIZATION PROJECT

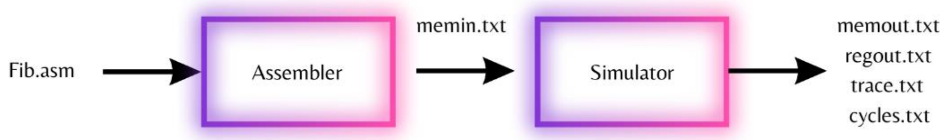
Lior Ben Ishay - 313135261

Keren Druker - 208678847

Table of Contents

Description

The purpose of this coding assignment is to create an assembler and a simulator for a RISC processor called SIMP as follow:



The Assembler will translate the assembler code written in text in assembly format to machine code.

The simulator should be able to read a program written in machine code and simulate the execution of the program on the SIMP processor.

The SIMP processor has a clock frequency of 512 Hz and contains 16 registers, each 32 bits wide (which are numbered as described below):

Registers Number	Registers Name	Purpose
0	$zero	Constant zero
1	$imm	Sign extended imm
2	$v0	Result value
3	$a0	Argument register
4	$a1	Argument register
5	$a2	Argument register
6	$a3	Argument register
7	$t0	Temporary register
8	$t1	Temporary register
9	$t2	Temporary register
10	$s0	Saved register
11	$s1	Saved register
12	$s2	Saved register
13	$gp	Global pointer (static data)
14	$sp	Stack pointer
15	$ra	Return address

There are two instruction formats: R format and I format.

R format			
19:12	11:8	7:4	3:0
opcode	rd	Rs	rt

I format			
19:12	11:8	7:4	3:0
opcode	rd	rs	rt
imm			

The processor supports 18 instructions:

Opcode Number	Register Name	Purpose
0	add	R[rd] = R[rs] + R[rt]
1	sub	R[rd] = R[rs] – R[rt]
2	mul	R[rd] = R[rs] * R[rt]
3	and	R[rd] = R[rs] & R[rt]
4	or	R[rd] = R[rs] | R[rt]
5	xor	R[rd] = R[rs] ^ R[rt]
6	sll	R[rd] = R[rs] << R[rt]
7	sra	R[rd] = R[rs] >> R[rt], arithmetic shift with sign extension
8	srl	R[rd] = R[rs] >> R[rt], logical shift
9	beq	if (R[rs] == R[rt]) pc = R[rd]
10	bne	if (R[rs] != R[rt]) pc = R[rd]
11	blt	if (R[rs] < R[rt]) pc = R[rd]
12	bgt	if (R[rs] > R[rt]) pc = R[rd]
13	ble	if (R[rs] <= R[rt]) pc = R[rd]
14	bge	if (R[rs] >= R[rt]) pc = R[rd]
15	jal	R[rd] = next instruction address, pc = R[rs]
16	lw	R[rd] = MEM[R[rs]+R[rt]], with sign extension
17	sw	MEM[R[rs]+R[rt]] = R[rd] (bits 19:0)
18	Halt	Halt execution, exit simulator

<u>**Additional information:**</u>

The **main memory** is 4096 lines, each containing 20 bits.

The **program counter** (PC) register is 12 bits wide, and it points to the relevant line in the code.

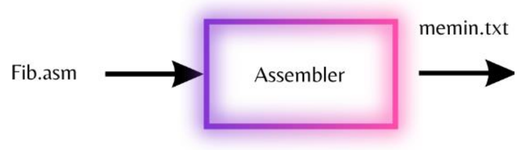
Cycle execution time:

- An R format instruction (except sw and lw) will take one clock cycle.
- An I format instruction (except sw and lw) will take two clock cycle (one to load the instruction and one to load the imm).
- lw and sw will take an additional clock cycle to read or write the data to/from memory.

Basic Assumptions:

1. The project is written in the C programming language. The assembler and simulator are different programs each in a different library that compiles and runs separately.
2. The code contains comments that explains its operation.
3. The code tested on Visual Studio 2017 + 2019, community.
4. The maximum line size of the input file is 300.
5. The maximum label size is 50.
6. Labels must begin with a letter followed by either letters or numbers.
7. Ignoring whitespace (spaces, tabs).
   There may be spaces or tabs and the input is valid.
8. Support hexadecimal digits in lower case and upper case.

The Assembler:



Input :

The input file program.asm contains the assembly program.

Output :

the output file memin.txt contains the memory image (as described above) and is input to the simulator.

Main key points:

1. We run over the assembly code twice.
   first for picking the labels store them and give them an immediate value.
   second run for translating the instruction and print them to memin.txt.
2. To make the main function as short as possible, we used help functions and implement them later in the code.
3. **Based on the TA's instructions, the memin.txt will be implement as follows:**

Line number	Memory part
0 - 2047	Machine code instructions
2048 - 4095	.word memory

In other words, the **memin.txt is divided to 2 parts**

rows **0-2047** are **instructions memory** (I-type, R-type and immediate) or 00000 after the end of the assembly code.

rows **2048-4095** are **data memory** for .word and 00000 otherwise.

Documentation:

The assembler is using the following libraries:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
```

Defined variables, based on the assumptions listed above:

```
#define  TRUE 1;
#define  FALSE 0;
#define  MAX_LINE_LEN 301// The maximum line size of the input file
is 300.
#define  MAX_LABEL_LEN 51 // the maximum lable size is 50.
#define  MAX_MEMO_LINES 4096
#define  DATA_MEMO_START 2048
#define  NUM_OF_REG 16
#define  MAX_VAR_IN_INST 7
```

We used enum to define the opcodes and registers as described in the SIMP processor:

```
enum OPCODE {
       add, sub, mul, and, or , xor, sll, sra, srl, beq, bne, blt, bgt,
ble, bge, jal, lw, sw, halt,
};

enum REGISTER {
       $zero, $imm, $v0, $a0, $a1, $a2, $a3, $t0, $t1, $t2, $s0, $s1,
$s2, $gp, $sp, $ra,
};
```

We created 3 structs for the instructions as they are described in the SIMP processor:

```
typedef struct Rfromat
{
       char op; // 8 bits
       char rd; // 4 bits
       char rs; // 4 bits
       char rt; // 4 bits
}Rfromat;

typedef struct Ifromat
{
       char op; // 8 bits
       char rd; // 4 bits
       char rs; // 4 bits
       char rt; // 4 bits
       int imm; // 20 bits.
}Ifromat;

typedef struct word
{
       int address;
       int data;
}word;
```

As mentioned above, we created a struct to store the current label if the line is a label:

```
typedef struct label // a struct to organize labels information
{
	// name of the label
	char name[MAX_LABEL_LEN];
	// val the immediate value related
	int val;
} label;
```

Store all the labels in an array of labels.

We count the PC (I type: PC+=2 , R type PC+=1) so when we run into a label, we know the relative PC of it and that’s the value of the label.

```
// an array to store the labels in order to use then in the 2nd run.
label labels[MAX_MEMO_LINES];
int label_counter = 0; // number of labels

// we need to store also the word memory - .word
int words[MAX_MEMO_LINES / 2] = {0};
int word_addrs_count = 0;
int max_word_address = 0;

uint16_t PC = 0; // cant be negative maximum 12 bits
```

Help Functions:

To make things easier and make the code more readable, we used the following help functions:

```
/****************************************************************
             help function declarations
****************************************************************/
void remove_white_spaces(char *str); // remove " " , "\t", "\n"
void remove_tabs(char* str);
Rfromat *read_instraction_R(char *line); //set the R struct
according to the line instruction
Ifromat *read_instraction_I(char *line); //set the I struct
according to the line instruction
int is_instraction(char *line); //return True if the line is an
instruction and false otherwise.
int is_I_instraction(char *line); // return True if the line is an I
type instruction and false otherwise.
char GetOpcode(char *op); // translate the opcode to a number
char GetRegister(char *reg); // translate the register to a number
int read_label_imm(char *str); // translate the label to an
immediate
void print_inst_I(FILE *machine, Ifromat *I); //print an I type
instruction to memin.txt
void print_inst_R(FILE *machine, Rfromat *R); //print an R type
instruction to memin.txt
word *read_instraction_word(char *line); //set the word struct
according to the line instruction
void print_inst_words(FILE *machine, word *words); // print the

value to memin.txt the data part.
```

The main goal of those functions is to distinguish between different types of instructions, as well as getting only the relevant information on each line.

Each function is implemented after the Main function and contains explanations along the code.

First Run:

```
// fill the lable array so we can use them
void firstRun(FILE *assembly)
{
	char line[MAX_LINE_LEN]; // a place to store the line
	char* label_name;
	int i = 0;

	while (!feof(assembly))
	{
		fgets(line, MAX_LINE_LEN, assembly); // reads a line 
from assembly
		remove_white_spaces(line); // removes spaces and tabes.
		if (strcmp(line, "\n") == 0) continue;  // empty line
		if (line[0] == '#') continue; // all row is comment
		if (strstr(line, ":") != NULL)  // If there are ':' in 
the line which means it contains a lable
		{
			if (strstr(line, "#") != NULL)
			{   // chacks if the ':' are in a comment
				if ((strstr(line, ":")) > (strstr(line, 
"#"))) continue;
			}


			label_name = strtok(line, ":"); // sparates the 
label name before the ":"
			if (label_name != NULL)
			{
				labels[label_counter].val = PC;
				strcpy(labels[label_counter].name , 
label_name);
				label_counter++;
			}

		}

		// if the line has "$" its an instruction if its is not 
after a "#"
		if (is_instraction(line))
			PC++;
		if (is_I_instraction(line))
			PC++;
	}
}
```

In the first run the function gets a pointer to the file that representing the assembly code.
It runs until end of the file and checks for: empty line, labels, comments, instructions.
It reads each line in the assembly code using `fgets` function (included in `stdio.h`), removes whitespaces using the `remove_white_spaces` function and skips tabs.
Checking if the line contains a Label is done by searching for ":", and if so, removes comments and ":" with the functions `strstr` and `strtok` (included in `string.h`)
Distinguishing between instruction and comment is done by `is_instraction` and `is_I_instraction` functions and increment the counter accordingly (by 1 for R-type and by 2 for I-type).

Second run:

```
void second_run(FILE* assembly, FILE *machine)
{
        int RowInd = 0;
        int R_format = FALSE;
        char line[MAX_LINE_LEN]; // a place to store the line
        // char op[6];
        // char reg[6];
        // int opcode;
        Rfromat *Rinst;
        Ifromat *Iinst;
        word *w;
        //inst instraction;
        // int j;
        while (!feof(assembly))
        {
                fgets(line, MAX_LINE_LEN, assembly); // reads a line
from assembly


                if (strcmp(line, "\n") == 0) continue;  // empty line
                if (line[0] == '#') continue; // all row is comment

                if (strstr(line, ".word") != NULL)
                {
                        if ((strstr(line, "#") == NULL) ||
                                (strstr(line, "#") != NULL) &&
(strstr(line, ".word") < (strstr(line, "#"))))
                        {

                                w = read_instraction_word(line);
                                words[w->address] = w->data;
                                if (w->address > max_word_address)
                                {
                                        max_word_address = w->address;
                // update max address
                                }
                                continue;
                        }
                }

                else if (is_instraction(line))  // a label will be
skipped
                {
                        remove_white_spaces(line);  // removes spaces
and tabes.

                        if (strstr(line, "$imm") != NULL) // Iformat
                        {
                                Iinst = read_instraction_I(line); //
ignore the comment and read the instraction into a Iformat struct
                                print_inst_I(machine, Iinst);
                        }
                        else //Rformat
                        {
                                Rinst = read_instraction_R(line); //
ignore the comment and read the instraction into a Rformat struct
                                print_inst_R(machine, Rinst);
                        }
                }

        }
```

```
		int line_num = PC;
		for (line_num; line_num <= 2048 ; line_num++)
		{
			fprintf(machine, "%05X\n", 0);
		}

		print_inst_words(machine, words);
	}
```

In the second run the function gets a pointer to the file that representing the assembly code and a pointer to the assembler output file – memin.txt.
In a similar way to the first run, it runs until end of the file and checks for: empty line, labels, comments, type of instruction.
It reads each line in the assembly code using `fgets` function (included in `stdio.h`), removes whitespaces using the `remove_white_spaces` function and skips tabs.
Checking if starts with ".word" and if so, read the address and value and stores it in the array of words.
Distinguishing between R and I instructions and execute properly using:
`read_instraction_I` , `print_inst_I` , `read_instraction_R` , `print_inst_R`.
At the end of the function it writes the output file (memin.txt) based on the array words, and pads the rest of the memory with zeros.

memin.txt. is divided to 2 parts
rows 0-2047 are instructions data (I-type, R-type and immediate) or 00000 after the end of the assembly code.
rows 2048-4095 are data memory for .word and 00000 otherwise.

Main:

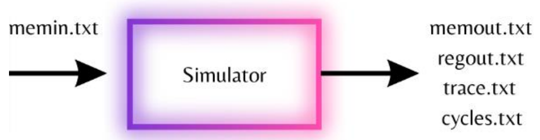
```
int main(int argc, char* argv[])
{

	FILE *assembly;
	FILE *machine;
	FILE *StartFile;
	assembly = fopen(argv[1], "r");
	if (assembly == NULL)
	{
		printf("Error: opening file failed\n");
		return 1;
	}
	machine = fopen(argv[2], "w");
	if (machine == NULL)
	{
		printf("Error: opening file failed\n");
		return 1;
	}

	firstRun(assembly);
	rewind(assembly);
	second_run(assembly, machine);

	StartFile = assembly;
	fclose(assembly);
	fclose(machine);

}
```

The main function's input is:

> **`argc`** - an integer representing the number of command line arguments.
>
> **`argv`** - an array of character pointers representing the command line arguments.

Open the assembly code file `(FILE *assembly)` and create an output file `(FILE *machine)`.
for both of the files, the program is looking for error in the opening process; if error has been found, the main returns 1 which indicating failure.
If the files opened correctly, then `firstRun` function is executed, as described above, find and store all the labels.
Then it uses `rewind` (included in `stdio.h`) to set the pointer back to the start of the file.
Then, `second_run` function is executed, as described above, reads the assembly code again and write the machine code in the output file – memin.txt.
At the end of the function, the input and output files are closing and returning 0 – which indicates success.

The Simulator:



Input :

The input file memin.txt that received from the Assembler and contains the memory image (as described above).

Output :

memout.txt - is an output file in identical format to memin.txt that contains the contents of the memory at the end of the run.
regout.txt - is an output file that contains the contents of the registers R2-R15 at the end of the run.
Each row will contain 8 hexadecimal digits representing a 32-bit number (the contents of those respective registers).

trace.txt - is an output file that contains a row of text for each instruction that was executed by the processor in the following format:

PC INST R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15

cycles.txt - is an output file which contains the number of clock cycles that transpired during the run(in decimal).

Main key points:

1. We chose to store the memin.txt in an array, that hold each line in a separate cell.

   This way, jump instructions can be done more easily without reading the input file again and again.
2. to make thing easier, we defined a struct that contains all the instructions part divided.

   The struct defined as follows:

```
typedef struct Instruction {
        int opcode;
        int rd;
        int rs;
        int rt;
        int imm;
} instruction;
```

3. The simulator is based on the method of:

   Fetch -> Decode -> Execute

   Following this method, we defined 3 main functions in the Simulator part:

   `read_file` – to read the memin.txt and store it in memin_array as described above.

   This function is equivalent to the **Fetch part.**

   `read_instructions` – decode the relevant line and store the data in the instruction struct before execution.

   This function is equivalent to the **Decode part**.

   `execute` – The function performs various operations based on the value of the **opcode** field, and using the data stored in the instruction struct, line by line.

   This function is equivalent to the **Execute part**.

Documentation:

Defines:

To make things more clear, we defined the following sizes:

```
#define True 1;
#define False 0;
#define MAX_LINE 5 // maximum line length in memin.txt.
#define MAX_LABEL 50 // the maximum lable size is 50.
#define MAX_MEMO_LINES 4096 // maximum number of lines in the
memory.
#define NUM_OF_REG 16 // number of registers.
#define MAX_DATA_MEMO 2048
#define MAX_INST_MEMO 2048
```

Functions:

There are 3 main functions in the simulator part:

```
void read_file(FILE* file) {
             char line[MAX_LINE + 1];
             int i = 0;
             char temp;
             // read the file line by line:
                   for (i = 0; i < MAX_INST_MEMO ; i++)
                   {
                         fgets(line, MAX_LINE + 1, file); // get line
       from file.
                         strncpy(memin_array[i], line, MAX_LINE); // copy
       the line into the array of memin_array.
                         temp = getc(file); // remove '\n' from the end
       of the line.
                   }

                   for (i = 0; i < MAX_DATA_MEM0 ; i++)
                   {
                         fgets(line, MAX_LINE + 1, file); // get line
       from file.
                         memory[i] = strtol(line, NULL, 16); // copy the
       data into the memory array.
                         temp = getc(file); // remove '\n' from the end
       of the line.
                   }
}
```

This function reads memin.txt, one line at a time, and stores each line as a string in an array of strings called memin_array.
The function reads the file by using the fgets function, as we saw earlier in the project.
The strncpy function then copies the line into the i'th element of the memin_array array, and stop after reading n chars (n = MAX_LINE=5).
The function also reads the newline character at the end of the line, using getc and discards it.
The loop continues until the end of the file is reached, which is determined by the feof function.

```
int read_instructions(int PC, instruction* instr) {
            int num;
            int imm_check = 0;
            char op_code[3] = { memin_array[PC][0], memin_array[PC][1],
      '\0' }; // read the opcode
            char rd[2] = { memin_array[PC][2], '\0' }; // read the rd
            char rs[2] = { memin_array[PC][3], '\0' }; // read the rs
            char rt[2] = { memin_array[PC][4], '\0' }; // read the rt

            instr->imm = 0; // initialize imm
            instr->opcode = (int)strtol(op_code, NULL, 16); // convert
      the opcode to decimal
            instr->rd = (int)strtol(rd, NULL, 16); // convert the rd to
      decimal
            instr->rs = (int)strtol(rs, NULL, 16); // convert the rs to
      decimal
            instr->rt = (int)strtol(rt, NULL, 16); // convert the rt to
      decimal.

            // check if the instruction is R-type or I-type:
            if (instr->rs == 1) { // rs is immidiate
                  imm_check = 1; // indicate that the instruction is I-
      type.
                  char imm[6] = { memin_array[PC + 1][0], memin_array[PC
      + 1][1], memin_array[PC + 1][2], memin_array[PC + 1][3],
      memin_array[PC + 1][4], '\0' };
                  num = (int)strtol(imm, NULL, 16); // convert the imm to
      decimal and sign extend it
                  if (imm[0] == 'F') {
                        num |= 0xfff00000;
                  }
                  instr->imm = num;
                  reg[instr->rs] = instr->imm;
            }

            if (instr->rt == 1) { // rt is immidiate
                  imm_check = 1;
                  char imm[6] = { memin_array[PC + 1][0], memin_array[PC
      + 1][1], memin_array[PC + 1][2], memin_array[PC + 1][3],
      memin_array[PC + 1][4], '\0' };
                  num = (int)strtol(imm, NULL, 16); // convert the imm to
      decimal and sign extend it
                  if (imm[0] == 'F') {
                        num |= 0xfff00000;
                  }
                  instr->imm = num;
                  reg[instr->rt] = instr->imm;
            }
            if (instr->rd == 1) { // rd is immidiate
                  imm_check = 1;
                  char imm[6] = { memin_array[PC + 1][0], memin_array[PC
      + 1][1], memin_array[PC + 1][2], memin_array[PC + 1][3],
      memin_array[PC + 1][4], '\0' };
                  num = (int)strtol(imm, NULL, 16); // convert the imm to
      decimal and sign extend it
                  if (imm[0] == 'F') {
                        num |= 0xfff00000;
                  }
                  instr->imm = num;
                  reg[instr->rd] = instr->imm;
            }

            return imm_check;
}
```

This function reads an instruction stored in memory (memin_array) at a specific location (specified by the PC or program counter) and parse it into its component parts (opcode, rd, rs, and rt) and then stores these parts in a instr struct that we saw earlier.

The function also uses the strtol() function (included in `string.h`) to convert the hexadecimal opcode, rd, rs, and rt to decimal values.

It also checks whether the instruction is R-type or I-type by checking the value of rs, rt, and rd.
If any of these values are 1, it indicates that the instruction is I-type and the immediate value is stored in the .imm field of the struct.
Additionally, the imm value in **2’s complement** is sign-extended and stored in the relevant register.
to make sure we get the 2’s complement representation of the number we used the following:

```
if (imm[0] == 'F') {
                num |= 0xfff00000;
            }
```

Finally, the function returns a value indicating whether the instruction is of I-type (returns 1) or R-type (returns 0).

```c
int execute(instruction* instr)
{
	switch (instr->opcode) {
	case 0x00: // add
		reg[instr->rd] = reg[instr->rs] + reg[instr->rt];
		return 0;
	case 0x01: // sub
		reg[instr->rd] = reg[instr->rs] - reg[instr->rt];
		return 0;
	case 0x02: // mul
		reg[instr->rd] = reg[instr->rs] * reg[instr->rt];
		return 0;
	case 0x03: // and
		reg[instr->rd] = reg[instr->rs] & reg[instr->rt];
		return 0;
	case 0x04: // or
		reg[instr->rd] = reg[instr->rs] | reg[instr->rt];
		return 0;
	case 0x05: // xor
		reg[instr->rd] = reg[instr->rs] ^ reg[instr->rt];
		return 0;
	case 0x06: // sll
		reg[instr->rd] = reg[instr->rs] << reg[instr->rt];
		return 0;
	case 0x07: // sra
		reg[instr->rd] = reg[instr->rs] >> reg[instr->rt];
		return 0;
	case 0x08: // srl
		reg[instr->rd] = reg[instr->rs] >> reg[instr->rt];
		return 0;
	case 0x09: // beq
		if (reg[instr->rs] == reg[instr->rt]) {
			PC = reg[instr->rd];
			return 1;
		}
		return 0;
	case 0x0A: // bne
		if (reg[instr->rs] != reg[instr->rt]) {
			PC = reg[instr->rd];
			return 1;
		}
		return 0;
	case 0x0B: // blt
		if (reg[instr->rs] < reg[instr->rt]) {
			PC = reg[instr->rd];
			return 1;
		}
		return 0;
	case 0x0C: // bgt
		if (reg[instr->rs] > reg[instr->rt]) {
			PC = reg[instr->rd];
			return 1;
		}
		return 0;
	case 0x0D: // ble
		if (reg[instr->rs] <= reg[instr->rt]) {
			PC = reg[instr->rd];
			return 1;
		}
		return 0;
	case 0x0E: // bge
		if (reg[instr->rs] >= reg[instr->rt]) {
			PC = reg[instr->rd];
```

```
                        return 1;
                }
                return 0;
        case 0x0F: // jal
                reg[instr->rd] = (instr->rs == 1 || instr->rt == 1) ? PC + 2
: PC + 1;
                PC = reg[instr->rs];
                return 1;
        case 0x10: // lw
                reg[instr->rd] = memory[(reg[instr->rs] + reg[instr->rt])%
MAX_DATA_MEM0];
                return 0;
        case 0x11: // sw
                memory[(reg[instr->rs] + reg[instr->rt]) % MAX_DATA_MEM0] =
reg[instr->rd];
                return 0;
        case 0x12: // halt
                PC = -5;
                return 1;
        }
}
```

This is the 3rd main function in the simulator.
It takes as input a pointer to an instruction structure, explained above.
Based on the value of the opcode The function executes the appropriate action.
The supported operations are those listed at the beginning of the file for the SIMP processor, and are executed based on the table in page 3.

For branch instructions, such as beq, bne, blt, bgt, ble, and bge, the value of the program counter (PC) is changed based on the values in registers rs and rt.

The jal instruction also updates the value of the PC, and stores the current value of the PC in register 15 ($ra) for returning.

All the jumps return 1 so it will indicate that a jump has occurred.

There are also load and store instructions, lw and sw, which can read from or write to a memory location using a base address stored in a register and an offset.

Finally, there is a halt instruction which sets the value of the PC to a sentinel value (-5 in this case) to indicate that the program has finished executing.

Main:

The function starts with open the input and output files, and check for errors in the opening process, in a similar way to the Main function in the assembler.

```
int main(int argc, char* argv[]) {
	FILE* memin = NULL;
	FILE* memout = NULL;
	FILE* regout = NULL;
	FILE* trace = NULL;
	FILE* cycles = NULL;
	char temp[MAX_LINE+1] = {0};
	int num;
	int jump=0;


	// open files:
	memin = fopen(argv[1], "r");
	// check if memin file was opened:
	if (memin == NULL) {
		printf("Error: memin file was not opened");
		exit(1);
	}

	memout = fopen(argv[2], "w");
	// check if memout file was opened:
	if (memout == NULL) {
		printf("Error: memout file was not opened");
		exit(1);
	}

	regout = fopen(argv[3], "w");
	// check if regout file was opened:
	if (regout == NULL) {
		printf("Error: regout file was not opened");
		exit(1);
	}

	trace = fopen(argv[4], "w");
	// check if memin file was opened:
	if (trace == NULL) {
		printf("Error: Cannot open file\n");
		return False;
	}

	cycles = fopen(argv[5], "w");
	// check if memin file was opened:
	if (cycles == NULL) {
		printf("Error: Cannot open file\n");
		return False;
	}

	// read memin file into memin_array:
	read_file(memin);
```

Then, execute the `read_file` function, that reads a file, one line at a time, and stores each line as a string in an array of strings called `memin_array`, according to the explanation above.

The next part is executed while PC is found within the memin memory:

```
	while (PC > -1 && PC < 2048) {

		// write to trace.txt file:
		memcpy(temp, memin_array[PC], 5);
		temp[MAX_LINE] = '\0';
		fprintf(trace, "%03X %s 00000000 %08x %08x %08x %08x
%08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x\n", PC,
memin_array[PC], reg[1], reg[2], reg[3], reg[4], reg[5], reg[6],
reg[7], reg[8], reg[9], reg[10], reg[11], reg[12], reg[13], reg[14],
reg[15]);

		imm_check = read_instructions(PC, instr);

		jump = execute(instr);

		if (imm_check == 1)
			cycles_count+=2; // add 2 cycle to the cycles
counter due to constant loading
		else
			cycles_count += 1; // // add 1 cycle to the
cycles counter

		if (jump)
			continue; // so we won't increase the PC.


		//update PC & cycles_count:
		if (imm_check == 1) {
			PC = PC + 2;
			imm_check = 0;
		}
		else {
			PC = PC + 1;
		}
		if ((instr->opcode == 0x10) || (instr->opcode == 0x11))
cycles_count++; // add 1 cycle to the cycles counter due to lw / sw
instructions
	}
```

The program then enters a while loop that continues until the PC (program counter) is indicates reaching the end of the memory or the program reads an Halt instruction.

Within this loop, the `execute()` function is called to execute the instruction, and the trace, cycles, and register files are updated with the appropriate information. The PC and the `cycles_count` are incremented, according to the conditions listed above.

This is the last part of the Main function:

```
            // print memory to memout.txt file:
            for (int i = 0; i < MAX_MEMO_LINES  / 2; i++) {
                  memcpy(temp, memin_array[i], 5);
                  temp[MAX_LINE] = '\0';
                  fprintf(memout, "%05X\n", strtol(temp , NULL , 16));
            }

            for (int i = 0; i < MAX_MEMO_LINES / 2; i++) {
                  num = memory[i];
                  fprintf(memout, "%05X\n",num);
            }


            // print registers to regout.txt file:
            for (int i = 2; i < NUM_OF_REG; i++) {
                  num = reg[i];
                  fprintf(regout, "%08x\n", num);
            }

            // print cycles to cycles.txt file:
            fprintf(cycles, "%d\n", cycles_count);


            // close files:
            fclose(memin);
            fclose(memout);
            fclose(regout);
            fclose(trace);
            fclose(cycles);


            // return 0 if no errors:
            return False;
      }
```

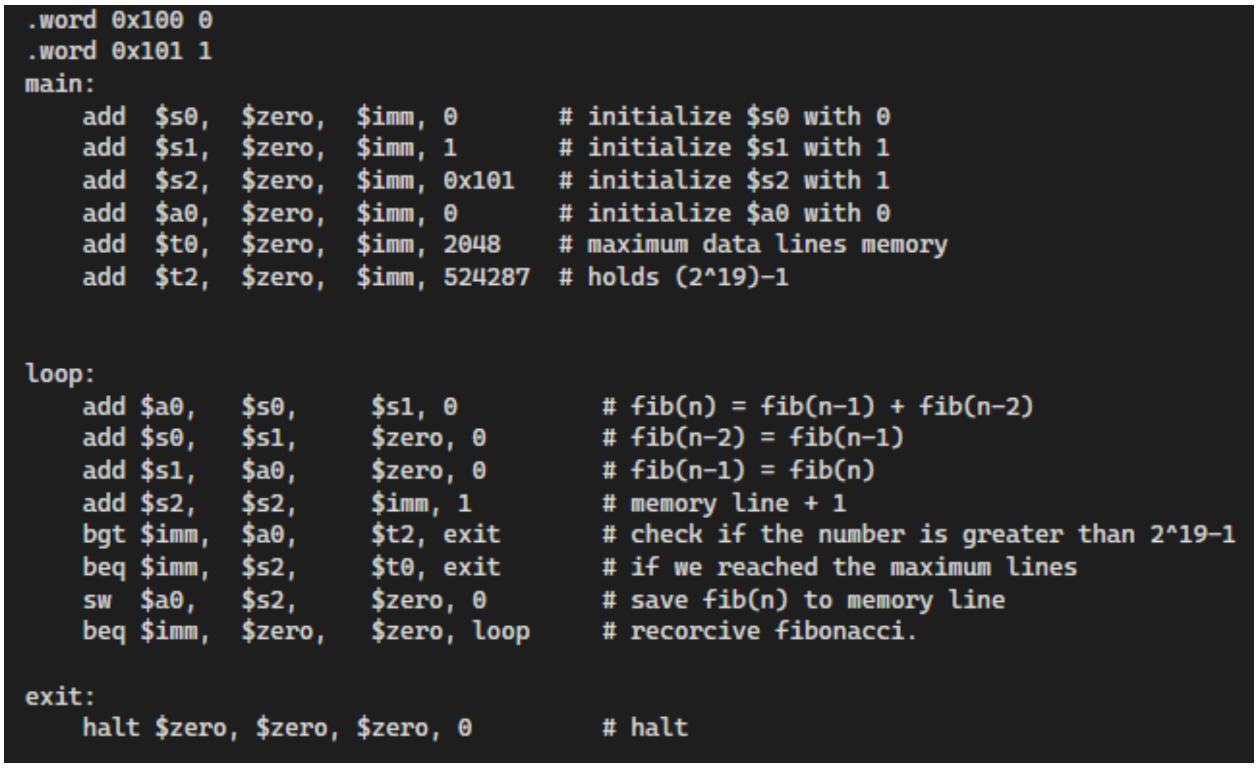
In this part the function prints the memory to memout.txt using memcpy function (included in `stdio.h`), the contents of the registers to the regout.txt file, and the value of the total cycles to the cycles.txt file.

Finally, the function closes all the files and return 0 that indicates success.

Test File - Fibonacci:

In order to test the code, we created a Fibonacci code in Assembly language as the input file.

The code goes as follows:



```
.word 0x100 0
.word 0x101 1
main:
    add  $s0,  $zero,  $imm, 0       # initialize $s0 with 0
    add  $s1,  $zero,  $imm, 1       # initialize $s1 with 1
    add  $s2,  $zero,  $imm, 0x101   # initialize $s2 with 1
    add  $a0,  $zero,  $imm, 0       # initialize $a0 with 0
    add  $t0,  $zero,  $imm, 2048    # maximum data lines memory
    add  $t2,  $zero,  $imm, 524287  # holds (2^19)-1


loop:
    add $a0,   $s0,     $s1, 0         # fib(n) = fib(n-1) + fib(n-2)
    add $s0,   $s1,     $zero, 0       # fib(n-2) = fib(n-1)
    add $s1,   $a0,     $zero, 0       # fib(n-1) = fib(n)
    add $s2,   $s2,     $imm, 1        # memory line + 1
    bgt $imm,  $a0,     $t2, exit      # check if the number is greater than 2^19-1
    beq $imm,  $s2,     $t0, exit      # if we reached the maximum lines
    sw  $a0,   $s2,     $zero, 0       # save fib(n) to memory line
    beq $imm,  $zero,   $zero, loop    # recorcive fibonacci.

exit:
    halt $zero, $zero, $zero, 0        # halt
```

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1.

Main:

In the main, the code initializes several registers:

- $s0 is initialized with 0 and it holds the current Fibonacci number.
- $s1 is initialized with 1 and it holds the next Fibonacci number.
- $s2 is initialized with 256 and it holds the memory line index.
- $a0 is initialized with 0 and it holds the temporary value of Fibonacci number.
- $t0 is initialized with 2048 and it holds the maximum data lines memory.
- $t2 is initialized with 524287 and it holds 2^19-1, the maximum value that can be represented with 20 bits 2's complement.

Loop:

The core of the code is in the loop where it executes the following steps:

- **add $a0, $s0, $s1, 0 -** it adds fib(n-2) and fib(n-1) numbers and store the result in fib(n)
- **add $s0, $s1, $zero, 0 -** it updates the fib(n-2) number with fib(n-1) number.
- **add $s1, $a0, $zero, 0 -** it updates fib(n-1) number with fib(n) number.
- **add $s2, $s2, $imm, 1 -** it increments the memory line index.
- **bgt $imm, $a0, $t2, exit** - it checks if the fib(n) number is greater than 2^19-1 if true it jumps to exit.
- **beq $imm, $s2, $t0 exit -** it checks if the memory line index is equal to the maximum data lines memory if true it jumps to exit.
- **sw $a0, $s2, $zero, 0 -** it stores the fib(n) number in the current memory line index.
- **beq $imm, $zero, $zero, loop -** it goes to the loop again.

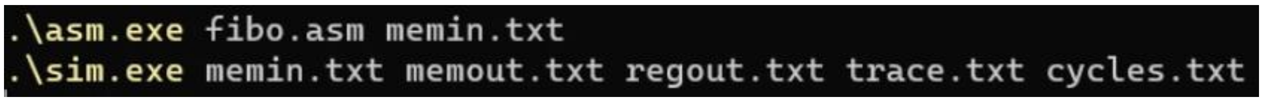
The output of this program is a sequence of Fibonacci numbers stored in memory, starting from 0 and 1, and incrementing until it reaches the maximum value that can be represented with 20 bits 2's complement (524287), or until it reaches the maximum data lines.

How to run the program:

Each folder contains the relevant files:

sim.exe , asm.exe, memin.txt, memout.txt, regout.txt, trace.txt, cycles.txt

To run the program on the CMD:



```
.\asm.exe fibo.asm memin.txt
.\sim.exe memin.txt memout.txt regout.txt trace.txt cycles.txt
```

Assembler:

The files relevant to the Assembler are located in:

Project_computer -> sim -> Debug.

Simulator:

The files relevant to the Simulator are located in:

Project_computer -> asm -> Debug.

Fibonacci:

The files to the whole program are located in:

Project_computer -> fibo