ISA Project Documentation:

Assembler:

The assembler contains two modules - pass1 and pass2 and the assembler c file "asm.c" which has the main function.

General operation of the assembler code:

asm.exe gets a program.asm file as an input and prints memin.txt as output. First, asm.c runs pass1 to get all the labels addresses from the assembly code, and then runs pass 2 to translate the program.asm code lines to machine language. Finally, the translation is printed into memin.txt file.

Modules:

Pass1:

Structs:

Label: holds the label name and address.

```
typedef struct Label {
        char name[MAX_LABEL];
        char address[ADDRESS_LEN+1];
}label;
```

Functions:

```
void create_labels_array(FILE* f, label labels_array[], int* label_count, int*
out_lines_num);
```

Create labels array: gets the input file of assembly code, iterates over it and creates an array of struct Label, which holds all of the labels in the code by its name and address.

```
bool check_line_valid(char* line, int line_index);
```

check line valid: check code's lines' validity and returns true is so.

```
bool read_label_from_line(char* line, char* label);
```

read label from line: reads the label, if exists, from a given line , and adds it to the labels' array. Return true if a label was found.

```
int hex_to_char(int dec);
```

hex to char returns the ascii char value of hex bit.

```
void dec_to_n_chars_hex(char* hex_char, int dec, int n);
```

decimal to n chars hex converts a decimal number to an n-sized char in hex representation.

Pass2:

Structs:

```
typedef struct Command {
        char opcode[REG_SIZE];
        char rd[REG_SIZE];
        char rt[REG_SIZE];
        char rs[REG_SIZE];
        char label[MAX_LABEL];
        char type[TYPE_SIZE];
        char imm[REG_SIZE];
        char word_address[MAX_LABEL];
        char word_val[MAX_LABEL];
}command;
```

Command: holds a command line arguments (registers, opcode) and additional data (command type , label, word for word command).

<u>Functions:</u>

```
void second_pass(FILE* f, char* argv[], label labels_array[], int label_count);
```

second pass: iterates over the assembly input file, parse each code line, and prints it translated to machine language into the memin.txt file.

```
void initialize_command(command* cmd);
```

initialize the command struct to be ready to receive new command.

```
bool parse_line(char* line, command* cmd, label labels_array[], int pass_num, int
label_count);
```

parse line takes a single code line, check for labels appearance, reads the arguments from the line and updates command struct attributes.

```
bool clean_line(char* line, int pass_num);
```

clean line returns true if the line is both a label and a command and cleans the line from commas

```
bool find_if_label_in_line(char* line);
```

return true if a single code line contains label.

```
bool is_cmd_dotword(char* line);
```

returns true if the command line is ".WORD" type.

```
int register_number(char* reg);
```

gets a register as a string and returns its' int number representation.

```
int opcode_number(char* opcode);
```

gets an opcode as a string and returns its' int number representation.

```
bool is_reg_imm(char* reg);
```

returns true if the register string argument was $imm.

```
void update_command_type(command* cmd, char* line, int arg_num, char* opcode, char*
reg1, char* reg2, char* reg3);
```

update the command type attribute of a command struct type.

```
void update_command(command* cmd, char* opcode, char* reg1, char* reg2, char* reg3,
char* temp);
```

update command object type attributes: registers, opcode, and labels (except type).

```
void update_imm(command* cmd, label* labels_array, int label_count);
```

Updates the IMM value to hex.

```
bool is_hex(char* str);
```

returns true if a constant in a code line is hex (0xHEXNUM).

```
void print_line(FILE* out_f, command* cmd, int* line_num);
```

prints a machine code line to memin.txt file.

```
void print_word(FILE* out_f, command* cmd, int* lines_num);
```

prints a .WORD command to memin.txt file.

```
bool is_cmd_i_type(command* cmd);
```

returns true if a command is I type.

Simulator:

The simulator contains three modules - command_handle, IO_handle, files_printer and the main simulator c file "sim.c".

General operation of the simulator code:

The simulator c file "sim.c" is divided into mainly four parts: initialize area (files, main memory, hard disk, monitor and structs), fetch-decode-execute loop, output files area and closing files.

command_handle module contains functions and structs mainly regarding to the fetch decode execute loop, and some help functions to convert between hexadecimal and decimal.

IO_handle module is responsible for managing the input and output devices, and interrupts' routines.

Print_files module contains functions that print the output files by the project demands.

the main simulator file includes all the above modules, and as mentioned, runs the fetch – decode – execute loop until the program stops by halt command.

Modules:

Command handle module:

Structs:

```
typedef struct Command {
        int opcode;
        int rt;
        int rd;
        int rs;
        int format;
        int imm;
}Command;
```

Command struct holds attributes of integers type that represent a parsed command line.

```
typedef struct GlobalParam {
        int clk;
        int pc;
        int halted;
        int interupt_in_proccess;
        int irq;
}GP;
```

GlobalParam holds attributes of global parameters and flags that are used to manage the CPU operation.

For example when "halted" goes high, the cpu knows we reached and of program.

Functions:

```
void initialize_global_params(GP *gp);
```

Initializes the global parameters at the beginning of the main program.

```
void initialize_main_memory(char* argv[], char main_memory[][LINE_SIZE]);
```

Initializes the main memory at the beginning of the main program. It updates the main memory array by the memin.txt and initializes to zero the rest of the memory lines.

```
void initialize_command(Command* cmd);
```

Initializes a declared command object.

```
bool is_hex_negative(char* hex_str);
```

checks whether a hex number represented as string is negative.

```
void decode(char main_memory[][LINE_SIZE], Command* cmd, int* registers_arr, int*
IO_regirsters_arr, GP* gp);
```

Decodes a line code from main memory into the separate attributes it holds such as opcode and registers.

```c
void update_command(Command* cmd, char* opcode, char* rd, char* rs, char* rt);
```

Updates a command struct object by the decoded attributes of a line code.

```c
void execute_command(char main_memory[][LINE_SIZE], Command* cmd,int *registers_arr,int
*IO_regirsters_arr, GP* gp,FILE *hwregtrace_file);
```

Executes the command by its' type and opcode.

```c
void increment_gp(Command* cmd, GP* gp, int *io_registers);
```

Increments the global parameters clock and pc depending on the type of the command.

```c
**
int hex_to_char(int dec);


void dec_to_n_chars_hex(char* hex_char, int dec, int n);
```

**The last two functions are depicted in the assembler part in pass1 module.

IO Handle module:

Structs:

```c
typedef struct Monitor {
        int frame_buffer[MONITOR_SIZE][MONITOR_SIZE];
        int last_value_col;
        int last_value_row;
}Monitor;
```

Monitor holds the frame buffer grid, and values that help to print the monitor.txt properly.

```c
typedef struct Irq2_Manage {
        int irq2_current_clock;
        bool interrupts_left;
        bool current_irq2_pending;
}irq2_manage;
```

Irq2_Manage holds attributes that manage the irq2 interrupts process.

```c
typedef struct Hard_Disk {
        int memmory[SECTOR_SIZE][SECTOR_SIZE];
        int start_clock;
}hard_disk;
```

Hard disk contains the hard disks' sectors, and a start clock value to control the time it takes to write or read operation to be done.

Functions:

```c
void set_irq_value(GP* gp, int *io_registers);
```

Sets irq value by the logic mentioned in the project demands:
irq = (irq0enable & irq0status) | (irq1enable & irq1status) | (irq2enable & irq2status).

```c
void set_irq2_status(irq2_manage *irq2, int *io_registers );
```

Uses the irq2_manage to set irq2status io register at the suitable clock count.

```c
void read_irq2in_data(irq2_manage *irq2, FILE* irq2in);
```

Reads and updates from the irq2in.txt into irq2_manage the current interrupt waiting to occur.

```c
void initialize_irq2_manage(irq2_manage *irq2);
```

Initializes irq2_manage struct object.

```c
void allow_interupt_begin(GP* gp, int* io_registers);
```

Allows interrupt to begin by update the IRQRETURN io register and the pc pointer to the interrupt routine address. It also raises a flag that an interrupt is in process, so no other interrupt routine would be allowed.

```
void init_frame_buffer(Monitor *moni);
```

initializes the monitor frame buffer to zero values.

```
void set_frame_buffer(Monitor* moni, int* io_registers, GP* gp);
```

If a write to monitor command took place, it will update the monitor's frame buffer at the given address.

```
void timer_update(int *io_registers);
```

Updates the timer count when enabled and resets it when the count reached maximum value.

```
void initialize_hard_disk(hard_disk *hd, char** argv);
```

Updates the hard disk memory from the diskin.txt file and initializes the hard disk start clock to zero (the value that helps determine the time it takes for a writ or read to occur).

```
void hard_disk_operation(hard_disk* hd, int* io_registers, char
main_memmory[][LINE_SIZE]);
```

Writes to the hard disk from main memory or reads from hard disk to main memory subject to the instructions in the project. Updates the disk io registers when an action is finished.

<u>Files printer module:</u>

<u>Functions:</u>

```
void print_leds(Command* cmd, int* registers, int* io_registers, FILE* leds_file);
void print_display7seg(Command* cmd, int* registers, int* io_registers, FILE*
display7seg_file);
void print_monitor(Monitor* moni, char** argv);
void print_cycles_file(int* io_registers, char** argv);
void print_main_memmory(char main_memmory[][LINE_SIZE], char** argv);
void print_regout_file(int* registers, char** argv);
void print_trace_file(Command* cmd, int* registers, GP* gp, FILE* trace_file);
void print_hwregrace_file(Command* cmd, int* registers, int* io_registers, FILE*
hwregtrace_file);
void print_diskout_file(hard_disk* hd, char** argv);
```

All the above functions print the file mentioned in the functions' name as an output of the simulator, like the project instructions demands.

<u>General Macro Definitions:</u>

```
// General Parameters

#define LINE_SIZE 6
#define MEM_SIZE 4096
#define NUM_OF_REGS 16
#define NUM_OF_IO_REGS 23

// Command formats
#define R_FORMAT 0
#define I_FORMAT 1
#define M_FORMAT 2 // for memmory commands (3 clocks)

// Registers (need to add them all)
#define IMM_REG 1

// opcodes
#define ADD 0
#define SUB 1
#define MUL 2
#define AND 3
#define OR  4
#define XOR 5
#define SLL 6
#define SRA 7
#define SRL 8
#define BEQ 9
```

```c
#define BNE 10
#define BLT 11
#define BGT 12
#define BLE 13
#define BGE 14
#define JAL 15
#define LW  16
#define SW  17
#define RETI 18
#define IN 19
#define OUT 20
#define HALT 21



// IO VALUES

#define MONITOR_SIZE 256
#define NUM_OF_IRQ2 10
#define MAX_CLOCK_SIZE 9
#define LEDS_HEX 8
#define SECTOR_SIZE 128
#define PIXEL_SIZE 2
#define DISK_RW_TIME 1024


// IO REGISTERS

#define IRQ0ENABLE 0
#define IRQ1ENABLE 1
#define IRQ2ENABLE 2
#define IRQ0STATUS 3
#define IRQ1STATUS 4
#define IRQ2STATUS 5
#define IRQHANDLER 6
#define IRQRETURN 7
#define CLKS 8
#define LEDS 9
#define DISPLAY7SEG 10
#define TIMERENABLE 11
#define TIMERCURRENT 12
#define TIMERMAX 13
#define DISKCMD 14
#define DISKSECTOR  15
#define DISKBUFFER  16
#define DISKSTATUS  17
#define RESERVED1  18
#define RESERVED2  19
#define MONITORADDR  20
#define MONITORDATA  21
#define MONITORCMD  22
```