Or Bahari 204356315
Or Shahar 208712471
Daniel Kinderman 205684590

# ISA Project Documentation

## Assembler

Assembler: asm.exe

The assembler program has input file *.asm program and an output file memin.txt, which will be used as an input file for the simulator.

The assembler opens the file and scans the *.asm program twice: in the first scan it saves the names and addresses of all the labels found in the file, and the second scan reads each line in the file, translates it to a command in machine language according to the SIMP instructions, and saves it to memin.txt.

1) First scan – **parse_labels(FILE* asm_prog)**: takes each line in the *.asm program and does the following:
    - Converts the line from a string to a list of arguments and removes comments.
    - Checks if the line has a label (if colons ':' are present).
    - If there is a line it's name and address are stored (according to the current PC counter) in the fields of the struct Label.
    - Calculates the matching opcode index and updates the PC accordingly.
    - We rewind back to the beginning of the file to start our next scan.
2) Second scan - **parse_instructions(FILE* asm_prog)**: takes each line in the *.asm program and does the following:
    - Converts the line from a string to a list of arguments and removes comments.
    - Calculates the matching opcode index.
    - If the opcode is ".word" it stores the value of the data argument in the address location in the Memory list (which will be later written into memin.txt).
    - else – it calculates the matching register index for rd, rs, rt, and for the immediate argument and parses the matching label address or the immediate number.
    - Calculates the SIMP command from the arguments and stores it in Memory.
    - Updates the last "line" in Memory which has any data that should be written to memin.txt.
3) **write_to_memory(FILE* memin)** – takes each command/data from Memory and writes it as a 32-bit hexadecimal word to the corresponding line in memin.txt.

## Simulator

Simulator: sim.exe
main file: main.c. main function: **main(int argc, char** argv)**.

the project is divided to modules.
**cpu module:**

DATA:

each opcode/register/IOregister is defined as an enum with its correct "SIMP" index.

instruction is a struct containing rd,rs,rt,imm and opcode values, already parsed from memory ("memin.txt").

Or Bahari 204356315
Or Shahar 208712471
Daniel Kinderman 205684590

cpu structure holds the **current** PC counter, parsed instruction, registers, IOregisters, memory and irq status.

operation is a pointer to an array of functions. every command operator is a function defined in operators module.

OPERATION:

cpu is intiliazed in function named: ***sim_init*()**.

each clock cycle, the cpu fetches an instruction using **fetch_address(cpu)**.

each (legal) instruction will be executed by **executeInstruction(cpu)**. otherwise, “ignore and continue”.

**<u>operators module</u>:**

OPERATION:

contains all the commands defined in SIMP processor, each command is implemented in a different function.

**<u>filesManager module:</u>**

DATA:

irq2 structure contains all the data required to handle irq2 interrupt.

OPERATION:

Parses memin.txt and writes memout.txt

Parses diskin.txt and writes diskout.txt

write cycles.txt

initializing irq2 structure.

**<u>main module:</u>**

Operation:

write_trace(cpu,trace_file_desc) writes a trace line for trace.txt.

**main(int argc, char** argv)** ‘pseudo code’:

1. initializes cpu.

2. Parses memin.txt, diskin.txt, initializing irq2.txt and starts parsing. Note that irq2.txt is being parsed during the entire program run, as answered in the course forum.

3. open outfiles.

Or Bahari 204356315
Or Shahar 208712471
Daniel Kinderman 205684590

4. for every clock cycle until HALT command:

   * check and handle interrupt

   * fetch correct address

   * write trace

   * handles leds.txt, display.txt and hwregout.txt if IN or OUT commands

   * update irq0status and irq2status (note that irq0status is updated inside **executeInstruction(cpu)**

   * execute instruction

   * handle disk request

5. write out files and close files.

6. free all dynamic allocated memory.

## Test Programs in SIMP "Assembly"

**leds.asm**: We enable interrupt irq0, set timermax=255 and set interrupt handler. Then we set the first led and loop ("Loop") until leds=0 that leads us to program termination. Every 256 cycles interrupt irq0 occurs ("Int_handler") and shifts leds to the left (saving led values in temporary register $t0 and then write its content to leds register).

**summat.asm**: In the program, we load from memory the data of the first array, add it to the data of the second array, and store it in the address of the third array. The second array is located 16 addresses above the address of the first array. The third array is in located 32 addresses above the first array. we load the correct value of each element of the matrices from memory and store them in the appropriate address in the third array. We only used one register to indicate the array address.

**clock.asm:** In this program, we write to the 7-segment display register. We compared very large values (which represent the time) that cannot be stored in the $imm register. Therefore, they were loaded into memory and were used indirectly. We wait for a minute (which is 256 cycles). We know when a minute is passed by sampling irq0status. After a minute we raise the value of the register that represents the time and we save it to memory. we denote Critical Time as 19:59:59. Once reached, we load the new time from memory (20:00:00), save it to the display and keep going. We have already saved the time in a way that should be presented and so it should not have been translated.

**disktest.asm**: In this program, we coded reading from disk and writing to disk. Each time from one disk location to another (different sectors). We worked with one registrar which represents the sector we

Or Bahari 204356315
Or Shahar 208712471
Daniel Kinderman 205684590

read from / write to. It was possible to work with one registrar since the reading and writing process is: 0->4->1->5->2->6->3->7->4. Red numbers are representing – sectors that we read from. Blue numbers are representing – sectors that we write to. 4 (black number) – stop condition. Each time we added and subtracted the value of a register: +4_-3_+4_-3_+4_-3_+4_-3_+4. To sample if the disk is busy - we used registrar 17 and checked its value. We also used a register which value indicates whether we last wrote or read.

**bubble.asm**: We run 2 loops over the entire array. first loop run over all the elements in the array. The second (nested) loop runs over all elements from 0 until the element we are currently sampling in loop 1. If we find an element that is with wrong order with the next element- we will swap between them.

**qsort.asm**: pseudo-code explanation: we choose the lowest index value as pivot. In each recursion: partition process for the array using pivot value, quicksort left partition recursively, and quicksort right partition recursively. To manage the recursion functionality(“Quicksort”) - we put $sp at 1000 and every recursive call we decrease it by 3. We made room for $ra and two registers indicating the size of the array p, r. We wrote the partition(“PartitionBegin”) with an external section that has 2 loops supporting the required element replacement. To manage the return from the recursive function we clear the ‘memory’ by adding 3 to $sp (“END”).