Assembler & Simulator documentation

Assembler:

Static values:

- Static arrays showing the opcodes, register names and Hex values according to the index that the values are sitting in
- Static counter of the number of labels in the Assembly code
- Static 2D array that in each index sits the label's name and its PC address in HEX
- Static array representing the DMEM for .word commands

Methods:

- `int get_reg_num(char *reg_name)` – returns the numeric value of the register according to its name
- `int get_opcode_num(char *opcode)` – returns the numeric value of the opcode according to its name
- `bool is_hex(char* value)` – returns true if the string represents an hex value (starts with 0X or 0x)
- `int dec_from_string(char* str)` – returns the decimal value of a string, whether it is represented in HEX or decimal
- `void get_hex_from_int(unsigned int num, char* hex, int num_of_bytes)` – receives a number, a result string and updates the calue of this string in HEX representation of the number according to the argument num_of_bytes. Includes negative numbers and sign extension.
- `void remove_last_char(char* str)` – updates the argument str without its last char
- `char* check_label(char * ptr)` – checks if a line is a label according if the 1st word ends with ':'. If so, it returns the label, else returns NULL
- `int get_label_index(char *label)` – returns the index in the label table if the label exists there by running over the labels table. Else return -1
- `void write_dmemin(char* file_name)` – writes the dmem file according to the format by running over the dmem array.
- `handle_word_cmd(char* address, char* data)` – writes the data in the correct address in the dmem array.
- `int main(int argc, char** argv)` – the body of the Assembler. Going twice through the Assebly code. The 1st run updates the labels table, and counts the amount of labels we have in the Assembly code. In the 2nd run we are going over each line and writing it in the imem file according to the format. We assume that the input is coorect. We are using all the functions mentioned above to recognize labels and convert string to int/HEX according to what we need. We are going over each word and handle a line in Assembly code one word at a timr according to its position in the format provided.

Simulator:

Static values:

- `static unsigned int pc` - Counter for the PC we are currently processing
- `static int tot_instructions_done` - Counter of the instructions the Simulator processed
- `static int total_lines` - Counter of the lines in the imemin file
- `static int next_irq2` - Static number for next irq2
- `static int disk_timer -` Static int for the disk timer
- `static int interrupt_routine` - Static int for the interrupt routine, used as indicator.
- `static int proc_regs[REGSNUM]`-Static array of 16 ints to represent the processor registers
- `static unsigned int hw_regs[HWREGS]`-Static array of 22 unsigned ints to represent the HW registers
- `static char instructions[MAXPC][6]`- Static array of strings to represent the instructions cache. We save each instruction in its PC
- `static int memory[MEMSIZE]` - Static array the represent the memory
- `static int monitor[PIXELS_X][PIXELS_Y]` - Static 2D array to represent the monitor
- `static int disk[SECTOR_NUMBER][SECTOR_SIZE]` - Static 2D array to represent the Disk
- `const static char hwReg[HWREG_NUM][HWREG_MAX_LENGTH]`- Static array for the HW registers names

Static files: some of the files were easier to operate when defined as static file . All of them are defined in the project's instructions.

- `FILE *irq2in;`
- `FILE *hwRegTraceFile;`
- `FILE *leds_file;`
- `FILE *trace_file;`
- `FILE *monitorYuv;`

Methods:

- `bool is_immediate(char* inst)` – returns true iff the instruction uses the $imm register
- `void get_hex_from_int(unsigned int num, int num_of_bytes, char* hex)` – same as the assembler
- `bool is_positive_imm(char* imm_hex)` – returns true iff the immediate value is >= 0
- `int update_imm(char* imm_hex)` – returns the integer value of the immediate provided to the method. Uses the is_positive_imm and shifts accordingly to get the correct representation.
- `int update_instructions(char* file_name)` – goes over the imemin file and updates all the instructions in an array by pc. Returns the number of lines in imemin file so we know when to exit the main loop.
- `void init_memory(char* file_name)` – initiates the DMEM according to the dmemin file
- `void write_hwRegTrace(char cmd, int ioReg, int value)` – updates the hwregtrace and leds files according to the format whenever needed
- `void write_dmem_out(char* file_name)` – writes the dmemout file according to format
- `void update_trace()` – updates the trace file in every instruction processed according to the format
- `void write_cycles(char* file_name)` – writes the cycles output file according to format
- `void write_regout(char* file_name)` - writes the regout output file according to format
- `void init_disk(char* file_name)`- a function to initiate the hard disk array, wich simulate a real hard disk. The function copies the diskin file data to work with on the simulator.
- `void write_diskout(char* file_name)` – the function writes diskout file and its called on the end of the simulator run. The function copies the data on the disk array into the file with respect to the blocks and sectors.
- `void write_sector()` – the function writes on disk interrupt the data from the diskbuffer to its destination address , function called on "out" op code instructions.
- `void read_sector()` – The function read from disk to a register of the processor when "in" op code instruction is called, with respect to the disk interrupt rules as defined on the project.
- `void disk_handler()` – The function is responsible for the disk interrupts, it will call write/read sector function is the disk is available for work as defined in the project.
- `void init_monitor()` – initiate the monitor array, wich simulate a real monitor to have 0 in all it's pixels.
- `void monitor_cmd()` – update a monitor pixel. The function knows wich pixel acroding to monitorx/y registers, and the value is saved in register monitor data.
- `void write_monitor_file(char* file_name)` – The function is called right before the simulator ends, we print to the monitor file , the monitor values at the end of the simulation.
- `void timer_handler()` – a function to manage timer interrupts like defined in the project.
- `static int check_signal()` – the function check irq as defined in the project.
- `static void move_to_interrupt_Routine()` – if the simulator get any interrupt the function saves the current pc in the dedicated hwregister, updates the pc variable and update `interrupt_routine = 1`, interrupt routine indicator as explained above.

- `void irq2_handler()`- The function manage the irq2 file interrupts by changing irq2status register to be 1 on the suitable clock cycle as defined on the project.
- `void interrupt_handler()` – every run on the main's while the function is called to handle interrupts by checking the interrupts status, and handle them according to wich interrupt is the relevant on the clock cycle.
- `void clock_counter()` – The function is responsible to update the clock cycle every time we enter the main's while, on imm instructions is called twice as defined on the project.
- `bool handle_cmd(int pc_index, bool is_imm) – receives a pc and bool for is_imm and performs the instruction in the specified PC according to the instructions received in the project. Return True if we had a branch taken in the method so we will not update the PC when not needed.`
- `int main(int argc, char** argv[]) – calls the init and writes methods. Running a while loop as long as we did not finish the imemin file. The main loop handles commands, checking interrupts and updates the relevant files.`