CODE DOCUMENTATION FOR COMPUTER ORGANIZATION PROJECT

Shaked Lubin 208728139, Liron Cohen 207481268, Ron Federman 209339290

Assembler

General Logic

1. Pass through the assembly source code – In this pass we only check at which lines there are labels and update **g_labels_arr**.
2. Rewind to the beginning of the source code file.
3. Pass through the assembly source code – In this pass we:
    a. Read each assembly line, decode it to hexadecimal and write it to **imemin.txt** file.
    b. For each **.word** command we update **g_data_memory**
4. Write **g_data_memory** to **dmemin.txt** file.
5. Close all the open files.

Section 1 - Elaboration

Runs **pass_over_file** function.

First the function checks:

1. If the line is empty by checking if the first letter is '\n'
2. If the line is a comment by calling **is_line_comment**

If 1 or 2 is true then we skip this row, otherwise we check if the line has a label in it. If the line does contain a label, then we create a new **label_t** object with:

- label = the label we just got
- cmd_index = **g_command_counter** (This is a global counter that counts how many of the lines we read are real commands)

Now we add the new **label_t** to the next empty spot in **g_labels_arr**.

Section 3 - Elaboration

In this part we use the same function as in part 1, the only difference is that we check if a line is **.word** command and if so we update **g_data_memory**. We also check if a line is a real command and if so run **decode_cmds_to_output_file** to decode it to hexadecimal and write it to **imemin.txt**.

Structs (defines in assembler.h file)

label_t - represents a label. Includes label and command index attributes.

Global and Static Variables

g_max_memory_index - Holds the max non empty index in the data array.

g_command_counter - Holds the counter for the commands.

g_label_count - Holds the counter for labels.

g_data_memory - An array that stores all the '.word' commands.

g_labels_arr - An array that stores all the labels and their indexes in the code

opcodes_arr - An array of commands names.

regs_arr - An array of registers names.

Helper Functions

get_opcode_num – Searches for the opcode's index in **opcodes_arr**.

get_reg_num – Searches for the register's index in **regs_arr**.

get_label_num – Searches for the label's index in **g_labels_arr**.

does_line_contain_label – checks if line contains ':'

is_label – gets immediate value and checks if the first char is a letter or not (to know if the immediate gets a value or a label)

decode_cmds_to_output_file – parse the line and uses the previous codes to get the hexadecimal decoding.

add_data_to_memory – use sscanf() to get the value and address for **g_data_memory**, also update **g_max_memory_index** for when we write the file (so we won't write the empty memory).

write_memory_file – simple for loop.

clear_leading_white_spaces – using isspace() to skip white spaces.

is_line_comment – checks if the first letter is '#' (we already cleared white spaces at this point).

line_has_label – returns the index of ':' or -1 if there isn't ':' or there is '#' before it.

is_line_word_command – checks if first letter is '.' (already cleared white spaces at this point).

line_has_command – checks if the first word in the line (after the label) is a valid opcode using get_opcode_num. If it is then the line contains a command, otherwise it doesn't.

Simulator

General Logic

1. Loading input files
2. Executing assembly commands
3. Writing output files
4. Close files

Section 1 – Elaboration

1. First, we set up the files from the command line arguments.
2. For irq2 file we read the first line and store it in **g_next_irq2**.
3. Next perform **load_instructions**() function to read imemin.txt into an array of **asm_cmd_t** – each entry in the **g_cmd_arr** array represents an assembly command.
4. Similarly, perform **load_data_memory**() and **load_disk_file**() to read the dmemin.txt and diskin.txt representing the data memory and disk contents respectively. The C objects contains them is **g_dmem** and **g_disk.data** (a field in struct disk_t).

Section 2 – Elaboration

The main part of the program is the **exec_instructions**() function. This function simulates the fetch-decode-execute for each clock cycle of the cpu.

- A global flag **g_is_running** indicating the program is running – we initialize it to true. The only function to set it to false is **halt_cmd**() which correspond to the halt assembly command.
- Each iteration (clock cycle):
  - First, check whether the cpu is being interrupted: this is the case when both the cpu isn't currently handling an interrupt and one of the 3 interrupts is enabled and signaled.
    - If the conditions hold, set **g_in_handler** to true indicating we're inside an interrupt handler, save current pc to **g_io_regs[irqreturn]** and update current **g_pc** to **g_io_regs[irqhandler]** – jumping to interrupt handler.
    - The **g_in_handler** flag will turn to false by the reti command once the interrupt handler returns.
  - Next, fetch instruction by reading **g_cmd_arr[g_pc]** (getting a command object).
  - Execute command includes two steps:
    - Update immediate registers to hold the immediate value from the command (after sign extension).
    - Execute the command by accessing a global array of functions pointers named **cmd_ptr_arr** in **index cmd->opcode**. Each entry in the array is a function pointer to perform the corresponding opcode from the command struct.

        - Each cmd function (add, sub, beq, etc..) access the **g_cpu_regs** and **g_io_regs** arrays and updating the relevant registers from the command object.
    - Next, we call functions to update (if necessary) monitor, disk, timer or next irq2.
    - Increment **g_io_regs[clks]** updating the clock cycles counter.
    - Last, check if the last command **isn't** a jump or brunch command (reti is included as a jump), if so then advance **g_pc** by 1. (if the command is jump or branch the command itself would handle the pc update).

Devices implementation:

- Disk
    - Represented by the **disk_t** struct declared in simulator.h
        - Data field is a byte matrix sized 128X512 (number of sectors times size of sector).
        - time_in_cmd holds the time since the disk started performing its current job. Each job takes 1024 clock cycles.
    - Each clock cycles the **update_disk**() function is called:
        - If a read/write command has been set in **g_io_regs[diskcmd]** and the disk isn't currently busy:
            - Mark the disk as busy.
            - Get relevant disk sector and g_dmem buffer from **g_io_regs[disksector]** and **g_io_regs[diskbuffer]** respectively.
            - For read command perform a memcpy() from **g_disk.data** to **g_dmem**.
            - For write command perform a memcpy() from **g_dmem** to **g_disk.data**.
        - Else, if the disk is busy:
            - Increment **time_in_cmd** field in the disk struct.
            - If the **time_in_cmd** reached 1024 the disk finished:
                - Mark it as not busy.
                - Reset **g_io_regs[diskcmd]**.
                - Indicate an interrupt by setting **g_io_regs[irq1status]** to 1. (An interrupt would only happen if the program has set the **g_io_regs[irq1enable]** to 1 (If not, polling on irq1status is needed).
        - Else (this disk isn't busy and no command is set) the function returns.
- Leds
    - Every time there's an out cmd and the relevant IO register is the leds register, we write the register value to the leds output file.
- Monitor

  - Every time we execute a command, we check if the value in **monitorcmd** register is true and if so, we update the monitor array at **monitoraddr** index to be **monitordata**. Then, we set the **monitorcmd** to be 0.
  - By the project's instructions, when we an 'in' cmd with **monitorcmd** IO register, we set the output value to be 0.
  - When we write the monitor's output file, we print the monitor's array values to the output file.
- <u>Timer</u>
  - Every time we execute a command, we check if the value in **timerenable** IO register is true and if so, we increment the timer and check for timer interrupts. We set the relevant flag and reset the timer.

Binom Assembly File

<u>General Logic</u>

1. MAIN
   - sets starting point of stack at 2048.
   - saves n and k to $a0 and $a1.
   - jumps and links to binom with n and k.
   - saves the answer at required address in memory.
   - ends the program.
2. BINOM
   - adjusts stack for 4 items.
   - saves $s0, $s0, $a1 and $ra to stack.
3. INFUNC
   - if k = 0 or n = k jumps to BASE.
   - calls binom with n-1 and k-1, answer is stored in $v0 and then in $s0.
   - calls binom with n-1 and k, answer is stored in $v0.
   - sums the results and jumps to RET.
4. BASE
   - sets $v0 to 1.
5. RET
   - loads $s0, $s0, $a1 and $ra from stack.
   - adjusts stack for 4 items.
   - returns.

Circle Assembly File

General Logic

1. MAIN
    - loads R (radius value) from memory and save its value squared.
    - sets index to 0.
    - sets $s1 to save monitor size (256 squared) and $s2 to save the constant 255.
2. LOOP
    - if index is bigger or equals to monitor size, jumps to end.
    - saves row value (i) to be index / 256.
    - saves column value (j) to be index % 255.
    - subtracts 128 from both values to calculate distance from the middle of the monitor.
    - calculates the two value squared and compares their sum to the radius squared.
    - If out of circle, jumps to inc.
    - else, sets pixel address as index, pixel color to white and draws it.
3. INC
    - increments the index value.
    - jumps to loop.
4. END
    - ends the program.

Mulmat Assembly File

General Logic

1. MAIN
   - Sets $s0 to point to the first cell of the first matrix A.
   - Sets $s1 to point to the first cell of the second matrix B.
   - Sets $s3 to point to the first cell of the output matrix C.
   - Saves to our stack the initial row and column counters values.
2. LOOP
   - Gets from the stack the current row and column counters values.
   - Calculates the current pointers to the relevant cells in A, B and C.
3. CALC
   - First, we check if we got to the end of A or B, if so, we need to jump to COND. If not, we continue.
   - Gets the relevant values from A and B.
   - Calculates the multiplication of the current values from A and B.
   - Adds the result to the current cell in C.
   - Advance to the next cell in A row.
   - Advance to the next cell in B column.
   - Returns to the start of CALC.
4. COND
   - Store the value we got in the current cell in C.
   - Advance to the next row and column (in the counters).
   - If we haven't gone through all the rows and columns we jump to RESET. Otherwise, we end the program.
5. RESET
   - Set $s0, $s1 and $s2 back to their initial values.
   - Jumps back to LOOP.

Disktest Assembly File

General Logic

1. MAIN
    - Sets $t0 to 7, this register will hold the sector number.
    - Sets $s0 to 0, this will be our buffer.
    - Sets 'diskbuffer' to $s0.
2. FOR
    - If we got to a negative sector number, we jump to RETURN.
    - Sets $t2 to be the next sector number ($t2=$t0+1).
    - We jump to WAIT until the disk isn't busy and then we will continue from this point (Using jal command and updating $ra).
    - We read the data from sector number $t0 to our buffer.
    - We jump to WAIT until the disk isn't busy and then we will continue from this point (Using jal command and updating $ra).
    - We write the data to sector number $t2 from our buffer.
    - Update the sector number - $t0=$t0-1.
    - Return to the start of FOR.
3. WAIT
    - Get 'diskstatus'.
    - If 'diskstatus'==1 we return to the start of WAIT (This implements a busy wait).
    - If 'diskstatus'==0 we return to where we left off in FOR (We do it with the $ra register)
4. RETURN
    - Halts