# 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:
   1. Read each assembly line, decode it to hexadecimal and write it to **imemin.txt** file.
   2. 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 array represents an assembly command. (The array is named g\_cmd\_arr).
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 containg 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 indicateing 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).

Device implementation:

* Disk
  + Represented by the disk\_t struct declared in simulator.h
    - Data field is a byte matrix sized 128X256 (num of sectores times size of sector.
    - time\_in\_cmd holds the time since the disk started performing it’s current job. Each job takes 1024 clock cycles.
  + Each clock cycle 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, pollingon irq1status is needed).
    - Else (this disk ins’t busy and no command is set) the function returns.

### 

## **Binom Assembly File**

General Logic

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 went 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