# Code documentation computer organization Final project

**208685784 Ori Avrahami 315416057, Ben Cohen**

## **Assembler** General Logic

1. Pass through the assembly source code – and check:
   1. If the current line is label, and update **labels\_array** if necessary.
   2. If the current line is **.word**  command, and updates words\_array.
2. Rewind to the beginning of the source code file.
3. Pass through the assembly source code – In this pass we Read each assembly line, decode it to hexadecimal and write it to **memin.txt** file.
4. Call **word\_to\_output** which write the .word commands to **memin.txt** file.
5. Close all the open files.

Section 1 - Elaboration

Run file\_flow function.

Using while loop, the program performs the next steps for each line in the assembly code:  
First it checks:

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

If one of the conditions above is true, the program will skip this line. otherwise, it checks if the line has a label in it. If so, it creates a new **label\_temp** object with:

* tmp\_label.label = the label we just got
* tmp\_label.index = commands\_counter (This is a global counter that counts how many of the lines we read are real commands)

And add it to the next empty index in labels\_array.

Eventually, check if the line is a '.word' command. If so, update words\_array and word\_index.

Section 3 - Elaboration

Run file\_flow function again.

In this run, the program checks if the line is a command. If so, run command\_to\_output that decode it to hexadecimal and write it to **memin.txt**.

Structs (defines in assembler.h file)

**label\_temp** - represents a label. Includes label and command index attributes.

Global and Static Variables

label\_counter - Holds the counter for labels**.**

num\_of\_lines **–** counts the num in the output file

word\_index **-** Holds the max non empty index in the data array.

commands\_counter - Holds the counter for the commands.

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

words\_array - An array that stores all the '.word' commands.

opcodes - An array of commands names.

registers - An array of registers names.

Helper Functions

num\_of\_opcode – Searches for the opcode's index in **opcodes\_arr**.

num\_of\_register – Searches for the register's index in **regs\_arr**.

num\_of\_label– Searches for the label's index in **g\_labels\_arr**.

check\_line\_with\_label – checks if line contains ':'

check\_if\_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)

command\_to\_output – parse the line and uses the previous codes to get the hexadecimal decoding.

word\_to\_output – Writes the memory data file.

command\_to\_output – simple for loop.

remove\_spaces – using isspace() to skip white spaces.

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

check\_line\_with\_label – returns the index of ':' or -1 if there isn't ':' or there is '#' before it.

check\_if\_word – 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 num\_of\_opcode. If it is then the line contains a command, otherwise it doesn't.

## **Simulator**

General Logic

1. Loading input files and save in temporary structs
2. Executing asm commands
3. Writing the output files
4. Close all 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 **COMMANDS\_ARRAY** – where each entry array represents an assembly command.
4. Similarly, we perform **load\_memory**() and **load\_disk\_file**() to read the data in the input file and diskin.txt respectively. The C objects contains them is **memory\_array** and **g\_disk.data** (a field in struct disk\_t).

Section 2 – Elaboration

The main part of the program is the **execute\_instructions**() function. This function simulates the fetch-decode-execute for each command.

* 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:
  + First we verify the command's validation using helper func.
  + Next, fetch instruction by reading **commands\_array[g\_pc]** (getting a command object), and increment the cycles by the commands type (R/I).
  + 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 **commands\_function\_array**. 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 **cpu\_registers** and **io\_registers** arrays and updating the relevant registers from the command object.
    - After command's execution – check if the command was lw/sw, and if so – update cycles by one.
  + Next, we call functions to update (if necessary) monitor, disk, timer or next irq2.
  + Then, 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 **io\_registers[irqreturn]** and update current **g\_pc** to **io\_registers [irqhandler]** – jumping to interrupt handler.
    - The **g\_in\_handler** flag will turn to false by the reti command once the interrupt handler returns.
  + Last, update g\_pc (pc\_counter) by different cases (if it isn't a jump/branch command, if handler happened and then set accordingly, if immediate value occurred and we don't want the imm-line to be read).

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, pollingon irq1status is needed).
    - Else (this disk isn’t busy and no command is set) the function returns.
* Leds
  + when "out\_cmd" and the relevant IO register is the leds register, we write the register value to the leds output file.
* Monitor
  + Check if the value in **monitorcmd** register is true and if so, update the monitor array at **monitoraddr** index to be **monitordata**. Then, we set the **monitorcmd** to be 0.
  + When writing to monitor's output file, we print the monitor's array values to the output file.
  + Monitor's output files includes both txt and YUV files for monitor's representation.
* Timer
  + Every time we execute a command, we check if the **timercurrent** value is greater than it's limit (**timemax)**. If so, and **timerenable** IO register is true, we increment the timer and check for timer interrupts. We set the relevant flag and reset the timer.

## **Fibo Assembly File**

General Logic

1. MAIN
   * Set $s0 as constant 0x800000 (if we decrease this number by one, we get the max size of positive in our case, so this is our boundary).
   * Set $s1 as the starting location in which we place the fib series.
   * Then, we add the first two fib elements and place them in their location
2. LOOP
   * Calculate the next fib element and set it to $t2
   * Branch if the element is greater than $s0 (that is explained above). If so, branch to Return
   * If not – set the last to elements as $t0, $t1
   * Increment $s1 (location)
   * Store the new fib element
   * Branch back to LOOP
3. RETURN
   * Ends the program

## **Square Assembly File**

General Logic

1. Two .word are used at the beginning in order to place the Location and Size of rect.
2. MAIN
   * loads top left corner from given address.
   * Loads size of rect from given address.
   * Initialize $s2 at constant 255 (for white color in monitor)
   * sets index\_1 ($to) 0.
   * sets index\_2 ($t1) 0.
   * Initialize $t2 as the costant difference from which to jump at the end of every end of rect in the current line, to it's beginning in the next line (in the monitor).
3. LOOP
   * if index\_1 is bigger than $t2 (that is explained above), jumps to end.
   * If index\_2 is bigger than size of rect – jumps to INC (out of shape).
   * Then, we do three operations regarding the pixel in the monitor (Read->Set-Draw)
   * Increment both indexes.
   * Jumps back to beginning of the LOOP
4. INC
   * increments index\_1 by 256 (the size of monitor) and then decrease it by the size of rect (in order for the jump to be until the start of the next line in the rect's position)
   * Init index\_2 back to zero
5. END
   * ends the program.

## **Disktest Assembly File**

General Logic

1. MAIN**:**
   * Set $a0 to 0, this register holds the first sector number.
   * Set $a1 to 1, this register holds the second sector number.
   * Start Disk\_test function (with the arguments above).
2. Disk\_test:
   * Set diskbuffer = 0, sets the memory cell saving the disk data to.
3. First\_read:
   * Store $ra in stack.
   * Check if the Disk is free to work. By jumping to Wait section.
   * When the disk is not busy, set ioregister[disksector] to the first address recived, and ioregister[diskcmd] to be on read mode (set to 1).
   * Now the first sector is written in the memory from the first location.
   * Set register $t0 = 7, will use as loop indicator.
4. First\_loop:
   * load the current word from memory and store it in $t1.
   * Sum += $t1.
   * Decrease the indicator by 1. And return to First\_loop label if i>=0.
   * After looping 8 times, save the sum in memory in location 0x100.
5. Sec\_read:
   * Check if the Disk is free to work. By jumping to Wait section.
   * When the disk is not busy, set ioregister[disksector] to the second address recived, and ioregister[diskcmd] to be on read mode (set to 1).
   * Now the second sector is written in the memory from the first location.
   * Set register $t0 = 7, will use as loop indicator.
6. Sec\_loop:
   * load the current word from memory and store it in $t1.
   * Sum += $t1.
   * Decrease the indicator by 1. And return to Sec\_loop label if i>=0.
   * After looping 8 times, save the sum in memory in location 0x101.
   * After summing the first 8 words in both 2 sectors, compare the sums and jump to Save\_first if first sum is bigger then second.
   * save the first sector 8 words sum in 0x102.
7. Save\_first:
   * save the second sector 8 words sum in 0x102.
8. end\_loop:
   * load $ra address from stack
   * empty stack
   * branch back to main section to terminate properly.

RETURN:

* + halts the program

1. WAIT
   * Get 'diskstatus'.
   * If 'diskstatus'==0 return to where we left off in FOR (We do it with the $ra register)
   * If 'diskstatus'==1 return to the start of WAIT (This implements a busy wait).