# Code documentation for computer ARCHITECTURE SCOREBOARD project

Liron Cohen 207481268, Ron Federman 209339290

## **Solution Main Parts** The solution includes the following main parts:

## main

The main file contains the main run of the program. It includes the following parts:

1. Open the input and output files and validate them using the **parsing\open\_and\_validate\_file** method.
2. Create the simulation struct using the **scoreboard\get\_simulation** method.
3. Initialize the simulation using the **init\init\_simulation** method.
4. In the first clock cycle, try to **scoreboard\fetch**. If succeeding, advance the pc and the clock cycle.
5. While the program is not halted and there is a positive number of active instructions, enter the main loop that contains:
   1. Try to **scoreboard\fetch**. If succeeding, advancing the pc.
   2. Call **scoreboard\execute\_all** and **scoreboard\issue**.
   3. If the trace unit is busy, write its values to the traceunit output file.
   4. Call **scoreboard\cycle\_end** and advance the clock cycle.
6. Write the values to the memout file, the regout file, and the traceinst file.
7. Free memory.
8. Close files.

## global header file

The main header file contains the following structures:

* **opcode\_e** - an enum that contains the possible opcodes (LD, ST…)
* **reg\_e** - an enum that contains the registers numbers (F1, F2...)
* **unit\_state\_e** - an enum that contains the possible states of a unit (IDLE, READ\_OPERANDS, EXEC, WRITE\_RESULT).
* **op\_config\_t** - defines an operation from configuration. Includes the number of units and the unit delay cycles.
* **unit\_id\_t** - defines a unit id. Includes the unit's opcode, index and a representing string.
* **inst\_trace\_t** - defines the values that are relevant to instruction trace (unit id, unit, cycle issued, cycle read operands…)
* **inst\_t** - defines an instruction. Includes its opcode, dst, src0, src1, imm, trace and raw instruction.
* **bool\_ff** - defines an old value and a new value of a FF containing Boolean value.
* **unit\_ptr\_ff** - defines an old value and a new value of a FF containing a unit pointer.
* **reg\_ff** - defines an old value and a new value of a FF containing a register name.
* **float\_uint** - a struct that includes a uint32 value of a number and its float corresponding value.
* **float\_uint\_ff** - defines an old value and a new value of a FF containing a float\_uint.
* **unit\_t** - defines a unit. Includes the unit id, Fi, Fj, Fk, Qj, Qk, Rj, Rk, busy, state, exec count, active instruction and exec result.
* **config\_t** - defines the configuration. Includes the trace unit and the configurations of all units.
* **reg\_val\_status** - defines the value and status of a register. The value if float\_uint and the status is a pointer to the relevant unit for updating the register, if there is any.
* **regs\_str[]** - an array that defines the names of the registers.
* **opcode\_str[]** - an array that defines the name of the opcodes.

## Input Parsing

**Parsing** includes the following methods:

* **open\_and\_validate\_file** - opens the file and makes sure it succeeds.
* **close file** - closes the file and makes sure it succeeds.

**memin** includes the following method:

* **load\_memin** - loads the memin input file to the memory struct.

**cfg** includes the following methods:

* **load\_configution** - gets a configuration file and a configuration struct.Reads the lines of the files, and parses them into the configuration struct using the methods below.
* **cfg\_str\_param\_type** - gets a parameter string and returns the relevant parameter type it defines (unit num, unit delay or trace unit).
* **cfg\_str\_operation** - gets a parameter string and returns the relevant operation it defines (add, sub, mult…).
* **cfg\_str\_num\_param** - gets a parameter string and takes the relevant number from the string (without the definition and '=' sign).
* **cfg\_str\_set\_trace\_unit** - gets a parameter string and parses the trace unit operation and index into the configuration.

## Initialization

**init** includes the following methods:

* **init\_simulation** -
  + loading memory from memin input file.
  + setting clock cycle, pc and counters to 0, and halted to false.
  + loading the configuration from file.
  + initializing the units using the **init\_units** method.
  + setting the trace unit.
  + create an array to store the current writing addresses for every cycle using the **create\_current\_writing\_addresses** method.
  + initializing registers values and statuses using the **init\_regs\_status\_values** method.
  + initializing the instruction queue using the **inst\_queue\init\_instruction\_queue** method.
* **init unit** -
  + for every operation, allocate the required memory according to num of units in configuration.
  + set up the unit values (index, operation, initial state as IDLE…).
* **init\_regs\_status\_values** -
  + set up the initial registers values to be as their index.
  + set up their status to be NULL (not waiting to any unit to write the value).
* **create\_current\_writing\_addresses** -
  + allocating the array of the current writing address (updated every cycle) as the total number of LD and ST units (each one can write to or read from at most one address at a time).

## Instruction Queue

## Operations

## Scoreboard

## Output Files

**output\_files** includes the following method:

* **write\_memout\_file** - write the values for the memout file. will be written at the end of execution.
* **write\_regout\_file** - write the values for the regout file. will be written at the end of execution.
* **write\_traceinst\_file** - write the values for the traceinst file. will be written at the end of execution.
* **write\_traceunit\_file** - write the values for the traceunit file. will be written in every cycle.

## Disposal

**free\_memory** includes the following method:

* **free\_units\_memory** - free all of the dynamically allocated memory for the units.
* **free\_current\_writing\_addresses\_array** - free the dynamically allocated memory for the array of current writing addresses.

## **Check Files**

## **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