# 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:

* Open the input and output files and validate them using the **parsing\open\_and\_validate\_file** method.
* Create the simulation struct using the **scoreboard\get\_simulation** method.
* Initialize the simulation using the **init\init\_simulation** method.
* In the first clock cycle, try to **scoreboard\fetch**. If succeeding, advance the pc and the clock cycle.
* 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.
* Write the values to the memout file, the regout file, and the traceinst file.
* Free memory.
* 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, exec result and is executed.
* **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.
* **address\_entry**  - defines an address in memory. includes the address and the index of the relevant store unit.

## 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 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 ST units (each one can write to or read from at most one address at a time). Initializing the values to be ADDRESS\_INVALID.

## Instruction Queue

The 16-long instructions queue includes the following structure:

* **inst\_queue\_t** - defines the instructions queue. Includes the instruction buffer that contains instructions in memory, head index, tail index, and Boolean values for is full and is empty.

**inst\_queue** includes the following methods:

* **init\_instruction\_queue** - setting memory for the queue and initializing its values.
* **is\_full** - checks if the queue is full.
* **is\_empty** - checks if the queue is empty.
* **top** - if the queue is not empty, returns the instruction in the head of it.
* **enqueue** - if the queue is not full, add an instruction into its tail.
* **dequeue** - if the queue is not empty, removes the value in its head and returns it.

## Operations

**operations** file includes the following methods:

* **perform\_instruction** - gets an instruction to preform and calls the relevant operation method according to its opcode.
* **ld\_op, st\_op, add\_op, sub\_op, mult\_op, div\_op, halt\_op** - preforming the operation.
* **update\_regs** - if it’s a LD operation updates the destination register float value, and otherwise updates the destination register integer value.

## Scoreboard

The scoreboard includes the following structure:

* **simulation\_t** - defines the simulation and includes:
  + the memory array.
  + issued instructions array.
  + number of issued instructions.
  + number of finished instructions.
  + the instructions queue.
  + registers values and status.
  + configuration struct.
  + operational units as defined in the configuration.
  + clock cycle counter.
  + pointer to the trace unit.
  + the active store addresses array.
  + the active store addresses array size.
  + the current pc.
  + is halted.

The scoreboard includes the following methods:

* **execute\_all** -
  + Going over all of the busy units, and calling the relevant method according to the unit's state.
* **fetch** -
  + If the program is halted returns false (not fetched).
  + If the instructions queue is not full, enqueues the instruction in the memory in index pc.
* **issue** -
  + Retrieving instructions queue's top.
  + Decoding the instruction using **parse\_line\_to\_inst** method.
  + If it's a halt instruction, mark the program as halted.
  + If the destination register is busy, waiting (handling WAW).
  + Finding free operational unit. If there is no free unit, waiting.
  + If everything is okay so far, dequeueing instruction from the queue.
  + Updating the scoreboard values of the assigned unit after issue, including handling the case when writing and using of a register in the same cycle. Using the **update\_scoreboard\_after\_issue** method.
  + Assigning the unit to the instruction, using the **assign\_unit\_to\_inst** method.
  + Updating the issued instructions array.
  + If it's a ST instruction, inserting the new store address to the active addresses buffer.
  + After a successful issue setting the next state to read\_operands.
* **read\_operands** -
  + If both registers are ready (Rj and Rk), marks them as not ready.
  + If they are not ready, returns false (didn't read operands yet).
  + After a successful issue setting the next state to exec.
  + Updating the instruction trace.
  + Calling exec once (as the first cycle of exec is the cycle of the read operands).
* **exec** -
  + Decrementing the exec counter of the unit.
  + Performing the instruction in order to get the right values for the operation using the **try\_perform\_instruction** method.
    - Checking if the writing address collides using the **is\_address\_collide** method. Stalling the execution of a memory instruction that will cause a collision or is dependent on earlier memory access not yet finished.
    - Calling the **preform\_instruction** method.
    - If it's a successful ST operation, removing its address from the active addresses buffer.
    - Saving the operation result in the exec\_result field and the actual register will be updated in the write-result phase.
  + If exec counter is finished setting the next state to write\_result.
* **write\_result** -
  + Checking for write after read (WAR) - avoid writing to the destination register if any unit didn't read it yet, using the **is\_there\_unit\_pending\_read\_operand** method. The method is going over all of the busy units and checks if they are waiting for the destination register.
  + Resolving read after write (RAW) by updating the R fields to true in the units waiting for this write and clearing the Q fields, using the **update\_pending\_units** method.
  + Updating the value in the regs array according to the instruction result.
  + Updating the instructions trace.
  + Marking unit as not busy, and dest reg as not pending for any unit.
  + Incrementing the finished instructions counter.
* **cycle\_end** -
  + **update\_ff\_regs** - going over all of the registers and updating their old values to be their new values.
  + **update\_ff\_units** - going over all of the operational units and updating their old values to be their new values.

## Output Files

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

* **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 methods:

* **free\_units\_memory** - free all of the dynamically allocated memory for the units.
* **free\_address\_buff** - 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