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:
    a. Try to **scoreboard\fetch**. If succeeding, advancing the pc.
    b. Call **scoreboard\execute_all** and **scoreboard\issue**.
    c. If the trace unit is busy, write its values to the traceunit output file.
    d. 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.

Test programs

Test program 1 – Calculating Pi

In this program we used the simulator to calculate an approximation for pi.

We used Leibniz formula:

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots\right)$$

Since the simulator doesn't include branching we wrote the program without loops, here is a snippet from the start of the program and the registers values in the end of execution:

```
ADD F7 F0 F4
ADD F5 F0 F1
ADD F5 F5 F2
DIV F6 F4 F5
SUB F7 F7 F6
ADD F5 F5 F2
DIV F6 F4 F5
ADD F7 F7 F6
ADD F5 F5 F2
DIV F6 F4 F5
SUB F7 F7 F6
....
```

$F0$	0.000000
$F1$	1.000000
$F2$	2.000000
$F3$	3.000000
$F4$	4.000000
$F5$	513.000000
$F6$	0.007797
$F7$	3.145484
$F8$	8.000000
$F9$	9.000000
$F10$	10.000000
$F11$	11.000000
$F12$	12.000000
$F13$	13.000000
$F14$	14.000000
$F15$	15.000000

We can see that in the end of the execution $F7$ contains a two-digit approximation for pi.

$F5$ contains the last denominator used for calculating the sum.

$F6$ contains the lase calculated element in the series which is $\frac{4}{513}$.

For each of the recurring set of 3 instructions we have the following dependencies between the registers which are handled correctly by the scoreboard mechanism:

Calculate $F5$ → Calculate $\frac{4}{F5}$ to $F6$ → Calculate the new sum by add/sub $F6$ to/from $F7$.

Test program 2 – loads and stores

The goal of this programs is to verify the functionality of the load and stores from the same address and the different possible dependencies.

```
ST F5 30
LD F2 30
ADD F2 F5 F2
ST F2 39
LD F4 39
ST F9 39
ST F11 39
LD F3 39
ST F6 40
HALT
```

- The first LD and ST instructions use the same address, hence $F2$ will read the value of $F5$ from memory (value of 5.0). Next, we add 5+5 and save the result in $F2$ ($F2 = 10.0$).
- Next, we store the value calculated for $F2$ and load it into to $F4$ ($F4 = 10.0$).
- We store the value of $F9$ and then $F11$ to address 39, $F11$ should override it and then we load the updated value to $F3$ ($F3 = F11 = 11$).
- We can see that all the registers have their correct value at the end of the program.
- The memout file also contains the right results
  - $MEM[30] = 5$
  - $MEM[39] = 11$
  - $MEM[40] = 6$

$F0$	0.000000
$F1$	1.000000
$F2$	10.000000
$F3$	11.000000
$F4$	10.000000
$F5$	5.000000
$F6$	6.000000
$F7$	7.000000
$F8$	8.000000
$F9$	9.000000
$F10$	10.000000
$F11$	11.000000
$F12$	12.000000
$F13$	13.000000
$F14$	14.000000
$F15$	15.000000

Test program 3 – Parallelism

The goal of this program is to check parallel execution of many instructions which is supported by the simulator.

The relevant configurations are:

- 10 ADD units
- 10 MULT units
- Add delay is 2 cycles.
- Mult delay is 7 cycles.

The traceinst file:

```
04222000 0 MULT0 1 2 8 9
04333000 1 MULT1 2 3 9 10
04444000 2 MULT2 3 4 10 11
04555000 3 MULT3 4 5 11 12
04666000 4 MULT4 5 6 12 13
04777000 5 MULT5 6 7 13 14
04888000 6 MULT6 7 8 14 15
04999000 7 MULT7 8 9 15 16
02223000 8 ADD0 10 11 12 13
02445000 9 ADD1 12 13 14 15
02667000 10 ADD0 14 15 16 17
02889000 11 ADD1 16 17 18 19
02224000 12 ADD2 17 18 19 20
02668000 13 ADD0 18 20 21 22
02226000 14 ADD1 21 23 24 25
```

```
MULT F2 F2 F2
MULT F3 F3 F3
MULT F4 F4 F4
MULT F5 F5 F5
MULT F6 F6 F6
MULT F7 F7 F7
MULT F8 F8 F8
MULT F9 F9 F9
ADD F2 F2 F3
ADD F4 F4 F5
ADD F6 F6 F7
ADD F8 F8 F9
ADD F2 F2 F4
ADD F6 F6 F8
ADD F2 F2 F6
HALT
```

We can see that the MULT instructions are not being stalled since they are independent and can run in parallel. For the add operations we can see some of them are getting stalled in the issue phase in order to avoid write after write: for example, the MULT instruction for $F2$ is finishing its write-result in cycle 9, so the ADD that will write to $F2$ is stalled to issue in cycle 10.