Scoreboard Project Documentation:

General Overview:

The scoreboard algorithm is a dynamic scheduling technique used to improve the performance of a processor by allowing it to execute multiple instructions in parallel and OOO. The algorithm works by maintaining a record, or "scoreboard," of the state of each instruction and the resources it uses, such as functional units, registers and memory.

The scoreboard simulator routine is divided into five main steps:

1. Fetch: the processor retrieves the instruction from memory to a queue.
2. Issue: the processor checks for a suitable free unit to see if the necessary resources are available and issues the instruction into a free functional unit for execution.
3. Read Operands: the processor reads the operands required for the instruction from registers or memory.
4. Execute: the instruction is executed using the operands and allocated resources.
5. Write Result: the instruction generates a result and writes it to a register or memory location.

   This routine repeats itself until all instructions were executed, meaning the queue is empty, and all the results were written back to the memory or registers.

Additionally, there are two more steps/actions the simulator executes:

1. Resource Allocation/Deallocation: of the functional units defined by the user configurations.
2. Logs: In this step, the relevant logs are being updated/created to reflect the correctness of the simulator.

Design:

The simulator (sim.c) is built of 5 modules:

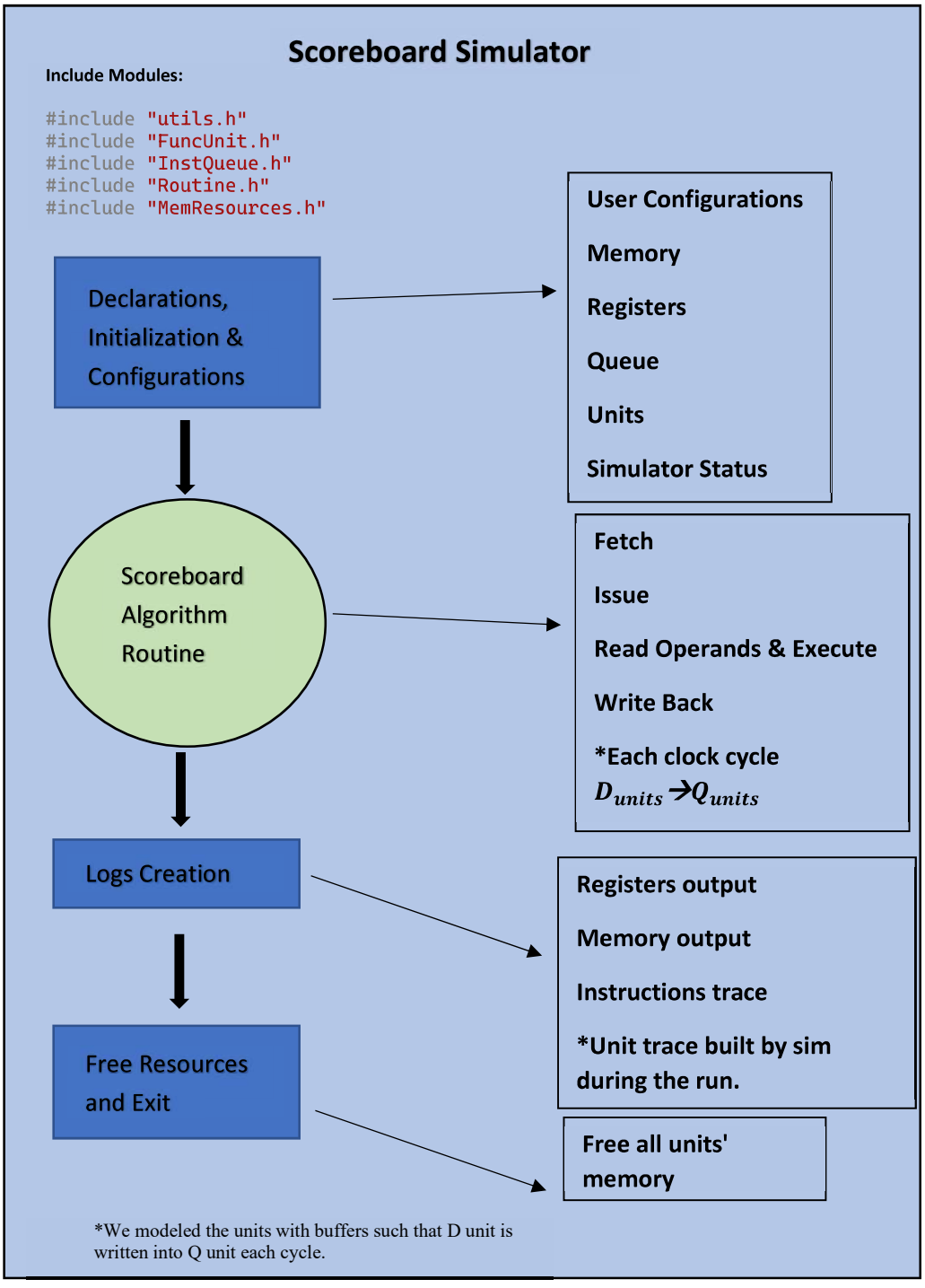
1. Routine: defines the top-level general routine functions - Fetch, Issue, Read Operands & Execute and Write Result and general structs for simulator status.
2. MemResources: defines the memory and registers structs and holds their functionality.
3. InstQueue: instructions queue module for a cyclic array-based queue structure and general functions (enqueue, dequeue, is_empty, is_full).
4. FuncUnit: functional units module defines the different units' structures and functions, and also unit-resources (memory and registers) interfaces` functions derivates from the algorithm.
5. Utils: general utils and user configuration struct and functions.

Simulator Implementation:

On execution the **sim.c** generates several steps:

1. Initializes instances and resources, updates them from user inputs and arguments and sets the user configurations.
2. Start and repeats the routine of scoreboard algorithm architecture.
3. When queue is empty, and all the units are empty the routine stops.
4. Print Logs of memory, registers, and instructions trace.
5. Free allocated memory and exits.

Top level block diagram for simplicity:



Modules Specifications:

Routine Module:

Structures:

1. Hazard – WAW and RAW hazards flag.
2. InstState – values of the insttrace.txt log.
3. Simstatus – holds general values for tracing and managing the simulator such as clock cycle, pc, halt flag**,** waw hazard flag, units done flag and an array of "InstState" to trace instruction for the instruction log.

Functions:

Routine functions:

1. InitSimStatus – initializes the Simstatus structure.
2. Fetch – fetches an instruction from memory to queue.
3. Issue – issue instruction into a free unit, updating all relevant fields.
4. ReadOpsAndExec – wait for operands to be ready, reads them and starts execution.
5. WriteBack (Write Result) – write to registers after execution and update status/value wherever needed.
6. CheckAllUnitsDone – check if all units finished, as an indication to finish simulator if also halt flag has already raised.
7. print_instrace – creates the instrace.txt file.

Shared Routine and Functional Units sub functions:

1. update_unit_fields – updates given unit fields as part of the issue step.
2. issue_to_unit – issue a given instruction to unit.
3. unit_write_back - wait for all other units to be ready to handle WAR hazard and perform write back on each unit.
4. UnitTypeWriteBack – perform unit_write_back iteratively on each unit.
5. unit_read_ops_and_exec – read operands and execute for a given unit.

Utils Module:

Structures:

1. ConfField – holds values that are used to parse user configurations file.
2. UserConf – holds user configurations from cfg.txt

Functions:

1. GetArgs – get user arguments.
2. SetConfig – set user configurations to UserConf.
3. getline – get line from memin.txt
4. dec_to_n_chars – converts a decimal number to hex string.

InstQueue Module:

Enum:

1. Opcodes enumeration {LD, ST, ADD, SUB, MUL, DIV, HALT}.

Structures:

1. Instruction – holds instruction fields (op, dst, src0 etc.).
2. Instruction Queue – cyclic array queue of instructions.

Functions:

1. CreateInst – create an instruction based on memory word.
2. InitQueue.
3. isEmpty.
4. enQueue.
5. deQueue.
6. isFull.
7. current_opcode – checks the opcode of the first instruction in the queue.
8. copy_inst – create a copy of an instruction.

MemResources Module:

Structures:

1. F2U – holds a float or an unsigned 32-bit value.
2. Mem – an array holds 4096 lines of F2U objects.
3. Reg – has a value and a tag.

Functions:

Regs functions**:**

1. InitRegs – initializes registers as required.
2. PrintRegs – print regout.txt
3. RegHasTag – return true if the register has a tag.
4. GetRegData – returns register data.
5. GetRegNum – return register name.

Mem functions:

1. InitMem – initializes memory as required from memin.txt.
2. PrintMem – prints memout.txt.

FuncUnit Module:

Structures:

1. Unit – holds unit fields (busy, name, Fi, Fj, Qj etc.) and extra values to trace instructions and save result.
2. UnitsTable – holds pointer to each function units` array (add array, sub array and etc.) and also the units configurations data.

Functions:

General purpose:

1. InitUnit – initialize a unit.
2. ResetUnit – reset unit after write back.
3. CreateUnitArray – create number of units type in array.
4. CreateUnitsTable – create the table of all units.
5. FreeUnitsTable – free memory allocated to all units.
6. unit_NS_update – update the Q table from the D table on clock positive edge.

Issue step:

7. search_free_unit – checks for available unit for instruction.
8. get_free_unit – returns available unit.
9. update_unit_reg_name – updates the units reg names.

Read operands and execute step:

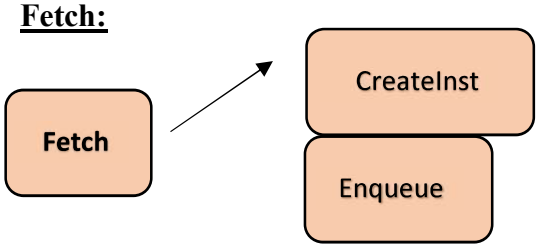
10. is_unit_busy – check if busy.
11. check_ready_for_store - takes care of multiple stores for the same address.
12. check_ready_for_load - takes care of load for a busy write address.
13. exec_add
14. exec_sub
15. exec_mul
16. exec_div
17. exec_ld
18. exec_st

Write result step:

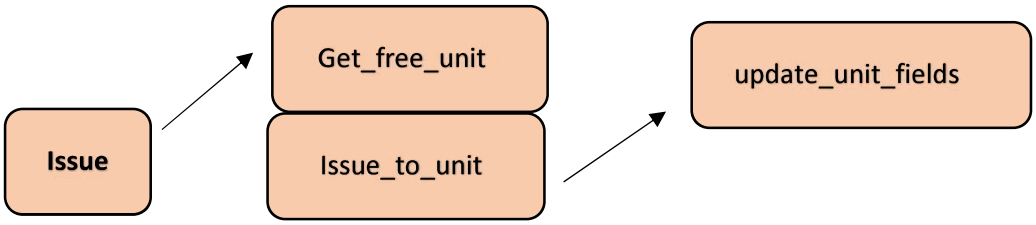
19. wait_unit_type_ready_for_wb – iterates over same type units and waits until all are ready for write back, in case of WAR hazard.
20. wait_other_unit_ready_for_wb - checks for WAR hazard on a given unit.
21. perform_wb_on_unit_type – iterates over all the units that might be waiting for registers to be ready.
22. perform_wb_on_other_unit – update other units as part of the write back just in case they are waiting for registers to be ready.

Sub Routines Functions Dependencies:

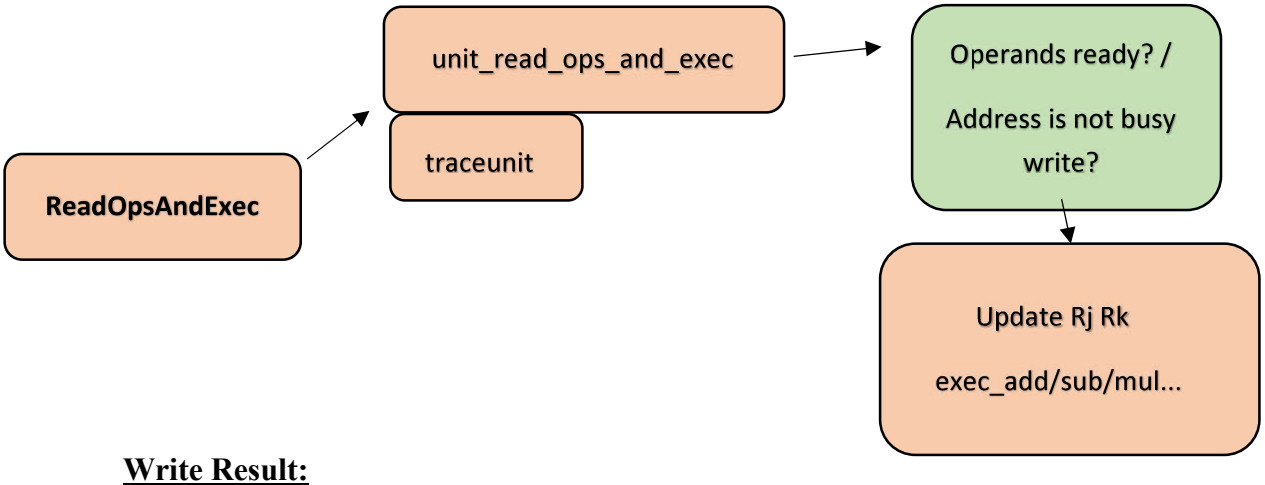
Fetch:



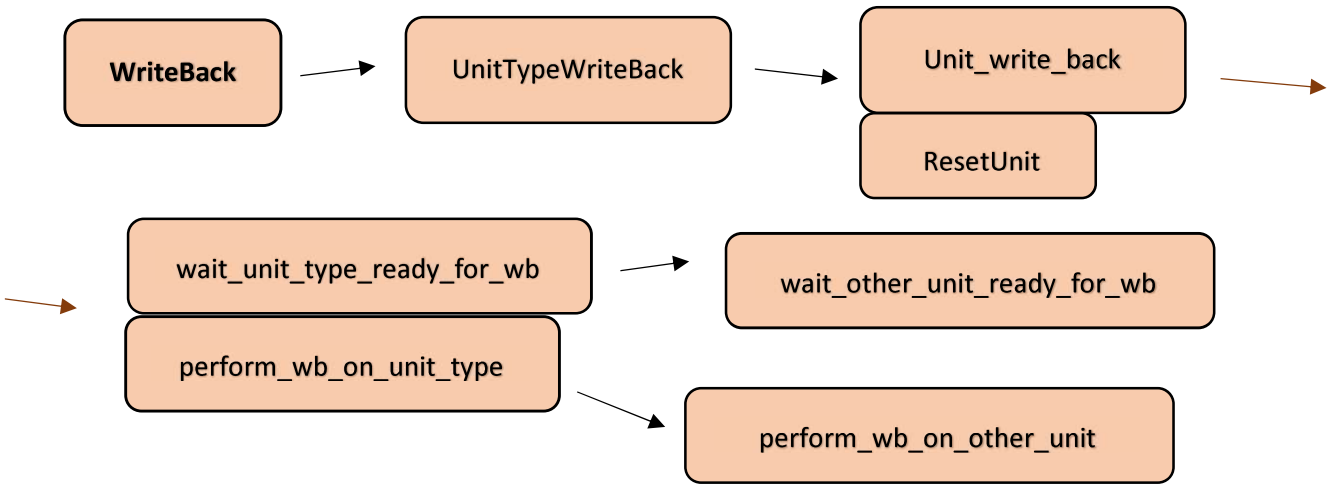
Issue:



Read Operands and Execute:



Write Result:



Tests Libraries:

1. Store and load to same address hazard test case:

Check that the simulator deals well with parallel writing/reading to/from same address at the same time. First the code checks LD after ST to same address, then ST after LD to same address and finally ST after ST to the same address.

```
ST    F0    F0    F5    0x20    //MEM[0x20] = F5
LD    F3    F0    F0    0x20    //F3 = MEM[0x20] - LD after ST of the same address
ADD   F7    F7    F2    0x00     //F5 = F5+F3
LD    F2    F0    F0    0x17     //F2 = MEM[0x17]
ST    F1    F0    F5    0x17    //MEM[0x17] = F5 - ST after LD of same address
ST    F0    F0    F5    0x95    //MEM[0x95] = F5
ST    F6    F0    F6    0x95    //MEM[0x95] = F6 - ST after ST of same address
HALT
```

2. Write after read hazard (taken from scoreboard.pdf on Moodle):

Check that the algorithm deals with the WAR hazard, such that a "young" instruction that finished execution, will not perform write back to a register that older instructions are using its value. In the next test, SUB won't write to register F3 until the two "younger" ADD commands would finish using F3.

```
ADD   F2    F5    F3    0x00    //F2=F5+F3
MUL   F5    F2    F3    0x00    //F5=F2+F3
ADD   F2    F5    F3    0x00    //F2=F5+F3
ADD   F4    F5    F3    0x00    //F4=F5+F3
SUB   F3    F6    F1    0x00    //F3=F6+F1 this instruction should wait on WR
ST    F0    F0    F1    0X11    //MEM[0X1i] = Fi
ST    F0    F0    F2    0X12
ST    F0    F0    F3    0X13
ST    F0    F0    F4    0X14
ST    F0    F0    F5    0X15
ST    F0    F0    F6    0X16
ST    F0    F0    F7    0X17
ST    F0    F0    F8    0X18
HALT
```

3. Special Test Case:

Check the special case in which an issued instruction uses the same register at the same clock cycle of an instruction that finished write result. The second LD instruction finishes the write back step to register F2 at the same clock cycle in which the SUB instruction is being issued.

```
LD      F6      F0      F0      0X22    // F6 = MEM[0X22]
LD      F2      F0      F0      0X2D    // F2 = MEM[0X2D]
MUL     F0      F2      F4      0X00    // F0 = F2 * F4
SUB     F8      F6      F2      0X00    // F8 = F6 * F2
DIV     F10     F0      F6      0X00    // F10 = F0 / F6
ADD     F6      F8      F2      0X00    // F6 = F8 + F2
ST      F0      F0      F0      0X15    // MEM[0X15] = F0
ST      F0      F6      F0      0X16    // MEM[0X16] = F6
ST      F0      F10     F0      0X17    // MEM[0X17] = F10
ST      F0      F8      F0      0X18    // MEM[0X18] = F8
HALT
```