Lab 6: Tomasulo's Algorithm

The goal of this lab assignment is to implement Tomasulo's algorithm, an instruction scheduling algorithm that allows for out-of-order (OOO) execution and parallelization of operations on multiple functional units.

This write up describes the required modules you have to implement. We also suggest you refer to the lecture for more information.

Reservation Station:

The reservation station will include stations for Add/Sub, Mult/Div, Load, and Store. Upon issuing an instruction, it should find an available slot within the corresponding station.

Name	Busy	Op	Vj	Vk	Qj	Qk	RemainCycle
Load0	false						
Load1	false						
Add0	false						
Add1	false						
Add2	false						
Mult0	false						
Mult1	false						
Store0	false						
Store1	false						

Fig1: An example of the Reservation Stations

Instructions in reservation stations wait for their operands to be resolved. Each entry in the reservation stations has a Busy status, the operation (referred as Op), the value of source operands (referred as Vj and Vk), and the reservation station that produces the source register values not currently available (referred as Qj, Qk).

Each entry in the reservation stations is equipped with an execution countdown, initiated to the duration of the operation being executed. The countdown begins only when all source operands are ready.

Additionally each station should also record the cycle in which the instruction was issued to it.

Let's consider the simplest example where a Load instruction (which takes 2 cycles) has just been issued on cycle 1 and there are no dependencies. Then, the execution of this Load instruction begins on cycle 2 and completes on cycle 3. In cycle 4, this loaded value will be broadcasted to any registers and reservation stations through the Common Data Bus (CDB).

As another example, consider that a Mult instruction (which takes 10 cycles) is issued at cycle 3 but doesn't commence execution at cycle 4 due to the unavailability of one of its inputs. Assuming that, at cycle 5, another instruction writes the required result (via CDB broadcasting), making Mult executable. Thus at

cycle 5, Mult starts counting down from 10, and its execution completes at cycle 5+10=15, writing the result via the CDB on cycle 16.

Common Data Bus (CDB):

The CDB is used to broadcast completed data. Through the CDB, it communicates to all reservation stations and registers the origin of the valid data, i.e., indicating "from which" station the data originates.

Register Result Status:

Field	F0	F1	F2	F3
Qi	Mult0	Load1		Load0

Fig2: An example of the register status

The floating-point registers are denoted as F0, F1, F2, etc. Each register entry includes a pointer indicating the associated reservation station and a valid-bit, signaling whether the register's data is ready (dataRdy).

When we issue a new instruction to a reservation station, we check the source and destination registers.
If all source registers (operands) are ready, they are assigned to Vj / Vk of the reservation station. Otherwise, the pointer to the reservation station is placed in Qj / Qk.
For the destination register, the current reservation station is assigned to it.

For example, if an instruction requires register F1 as input, but F1 is not ready and is waiting for the result of "Load1" station, then we can put "Load1" to Qj (or Qk). Upon "Load1" writes the result, Qj (or Qk) is cleared and data is written to Vj (or Vk).
Also if the current instruction uses the "Add1" station, and its destination is F4, then we assign "Add1" to F4, and mark F4 as not data-valid until the result is available.

Implementation:

In your code, you can follow these steps for your simulator.

In each cycle:

1. Read the Common Data Bus (CDB), and update registers and reservation stations.
    a. Suppose in the previous cycle, an operation has completed its execution in a particular station. Hence in this cycle, the resultant data is broadcasted to CDB and we should monitor the CDB and update any registers and reservation stations accordingly (write result).
2. For each reservation station,
    a. If all sources are ready, decrease its remaining_cycle (timer) by one.

   b. If the remaining_cycle = 0, it implies that the execution is completed. Broadcast the result to CDB in the next cycle. Since the CDB can only process one data each cycle, if multiple stations complete at the same time, process the result of the instruction which was *issued first*, and leave the other one to the next cycle. (the first issued instruction will broadcast its results first).
3. Issue a new instruction if a free reservation station is available.
   a. Examine the register result status.
      i. If the source registers (operands) already hold valid data, assign them to Vj / Vk.
      ii. Otherwise, if the data is not yet available (i.e., waiting for the result from another reservation station), set the pointers to the corresponding reservation stations in Qj / Qk (e.g. "Add1" in Qj means the instruction waits for Add1 to finish and write the result).
      iii. For the destination register: if the current station name is "Add2", it needs to assign "Add2" to the destination register. This indicates that the register is waiting for station "Add2" to complete and write the result.

Your simulator should record the cycle at which an instruction is issued, completes execution, and writes its result. Also it should record the status for all registers each cycle. Refer to the later sections for a detailed outline of the simulator skeleton.

Examples of how to handle RAW, WAR and WAW:

RAW:

ADD **F2** F0 F1
SUB F3 **F2** F1

The SUB instruction waits for the value of F2 provided by ADD to be available before initiating the execution. Suppose ADD completes its execution at cycle 1 and writes the result at cycle 2. In this case, SUB will start execution at cycle 2 and complete execution at cycle 4. SUB writes its result at cycle 5.

WAR:

ADD F2 **F0** F1
SUB **F0** F1 F5

ADD was issued before SUB and was already in the reservation station with the proper value of F0 (either valid data or a pointer to another station). There will be no hazard when issuing SUB because it overrides the F0 register with SUB's reservation station.
ADD has already used the F0, and the SUB can safely overwrite it without causing conflicts.

WAW:

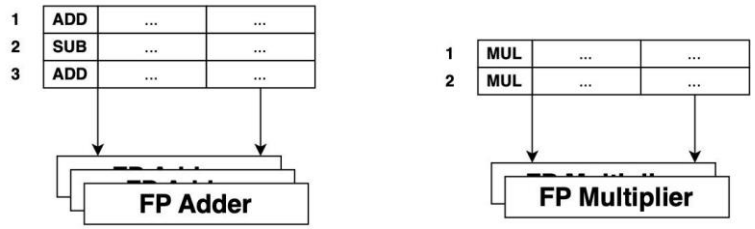
DIV **F2** F0 F4
ADD F3 **F2** F1
SUB **F2** F1 F5

ADD is issued and waits for the value of F2 provided by DIV. In the station of ADD, the Qj points to the station of DIV. SUB is issued after ADD and overwrites F2 with the station of SUB.

Suppose all inputs of SUB are available. Even though SUB completes execution before ADD and writes to register F2 directly, ADD can still get correct data as it uses the pointer related to DIV's reservation station. Moreover, when DIV finishes execution, it doesn't override the value that SUB already wrote in F2 as the register no longer points to DIV's reservation station.

Functional units:

We assume that for each row in a reservation station, there is a corresponding functional unit as shown in the diagram below. We assume this condition for each type of reservation station (LOAD, STORE, ADD, MULT).



Therefore, if multiple instructions of the same type are waiting for a particular value and that value has just been issued on the CDB, then multiple instructions *could* begin executing based on the availability of the functional units. Still, only one value is allowed to be written to the CDB at the end.

(Note that as we present parallelism here, we don't assume pipelines in the functional units. That could be different from the real-world scenario and here we just take it as an easier case.)

In the skeleton code:

Instruction (trace.txt)

You will have to support ADD, SUB, MULT, LOAD, STORE, and DIV instructions. The latency of each instruction is mentioned in the table below.

Instruction	Latency (Cycles)
ADD, SUB, LOAD, STORE	2
MULT	10
DIV	40

For example, if ADD is issued at cycle 1, then it should be completed at cycle 3. At cycle 4, the result is broadcasted through CDB and dependents are resolved (thus they can start execution at cycle 4).

The instructions will be read from trace.txt as string. Here is an example:

```
LOAD F6 34 0
LOAD F2 45 0
MULT F0 F2 F6
SUB F8 F6 F2
DIV F10 F0 F6
ADD F6 F8 F2
```

It follows the format of [Op] [Destination] [Source 1] [Source 2], e.g., a STORE instruction looks like: "STORE F6 40 0".
To simplify, we make some specific adjustments for the instructions:

1. The LOAD and STORE operation will only use immediate numbers for addresses. Thus we don't need to consider adding the regular registers.
2. We don't implement actual memory in the simulator, so no need to consider memory hazards. In other words, all LOAD and STORE will be completed successfully in 2 cycles after issuing.
3. For consistency, STORE also has one cycle of "writing result", but it won't change the register result status (i.e., you don't need to put something like "STORE0" to register).

Hardware configuration (config.txt)

We can define the size of each part in our simulator. Each line represents the number of:

```
Reservation Station size of Load (default = 3)
Reservation Station size of Store (default = 3)
Reservation Station size of Add (default = 3)
Reservation Station size of Mult (default = 3)
Floating Point Registers size (default = 12)
```

The Reservation Stations have name indexing from 0 to its (size-1).
For example, if the size of RS Load, RS Store, RS Add and RS Mult are (1, 2, 3, 4), then they can be indexed by name: Load0, Store0, Store1, Add0, Add1, Add2, Mult0, Mult1, Mult2, Mult3.

Similarly, if the Register size is 4, the registers have names: F0, F1, F2, F3.

Output (trace.out.txt)
We will print all register result status each 5 cycles. And at the end of the program, we print the instruction status. Here is an example:

```
Cycle 5:
F0: Mult0, dataRdy: N,
F1: , dataRdy: N,
F2: Load1, dataRdy: Y,
F3: , dataRdy: N,
F4: , dataRdy: N,
F5: , dataRdy: N,
```

```
F6: Load0, dataRdy: Y,
F7: , dataRdy: N,
F8: Add0, dataRdy: N,
F9: , dataRdy: N,
F10: Mult1, dataRdy: N,
F11: , dataRdy: N,

…

Instruction Status:
Instr0: Issued: 1, Completed: 3, Write Result: 4,
Instr1: Issued: 2, Completed: 4, Write Result: 5,
Instr2: Issued: 3, Completed: 15, Write Result: 16,
Instr3: Issued: 4, Completed: 7, Write Result: 8,
Instr4: Issued: 5, Completed: 56, Write Result: 57,
Instr5: Issued: 6, Completed: 10, Write Result: 11,
```

We have already given the structure for the register in the skeleton. It should print out which reservation station it's waiting for, and whether its data is ready.

After all instructions have been executed, the simulator prints the status for each instruction, including when it's issued / executed completed / writing result.

Your Assignment:

1. We have provided the skeleton code in the file tomasulosimulator.cpp. Implement the functions where you see "TODO: implement".
    a. A Makefile has been provided for you to compile the source code (do not modify the Makefile)
    b. We will be compiling your code with: `g++ version 11.3.0.` Please make sure that your code is compatible with `g++.`
    c. You can compile your design by typing "make" which will create an executable called "tomasulosimulator.out". Run the executable by calling: "./tomasulosimulator.out config.txt trace.txt**"**.
    d. Calling the binary "./tomasulosimulator.out" expects the "trace.txt" file in the same directory and produces one output file name "trace.out.txt".
    e. We have provided the correct output results for the provided test case in the "expected_results" directory. Make sure you pass this test case before submitting your code. However, we encourage you to write your own sequence of requests and check your design.
2. You will be submitting your assignments on Gradescope. **You must upload ONLY one file on Gradescope: "tomasulosimulator.cpp". Do not zip the file or upload any other file.**