COMPUTER ARCHITECTURE
TOMASULO ALGORITHM SIMULATION

G SWATHI CS24M1007
A MANASA REDDY CS24M1013

Use the following information related to the functional units of the Dynamic Instruction Scheduling - In-order Issue, Out-of-order Execution, and out-of-order commit processor using Tomasulo Algorithm.

64-bit Processor,
32 registers and each register is 64-bit

Functional Units
Integer Addition/Subtraction (IADD Rdst, Rsrc1, Rsrc2) - 6 CC
Integer Multiplication (IMUL Rdst, Rsrc1, Rsrc2) - 12 CC
Floating point Addition/Subtraction (FADD Rdst, Rsrc1, Rsrc2) - 18 CC
Floating point Multiplication (FADD Rdst, Rsrc1, Rsrc2) - 30 CC
Floating point Division (FDIV Rdsr, Rsrc1, Rsrc2) - 40 cc
Load (LD Rdst, Mem) - 1 CC
Store (ST Mem, Rsrc) - 1 CC
Logic Unit (AND/OR/XOR Rdst, Rsrc1, Rsrc2) - 1 CC
No Operation (NOP) 1CC

Implement and Test with the Assembly program as input to the processor - Show How Instruction packet is generated and How it is executed in Processor. Number clock cycles program takes to execute. Illustrate each clock cycle operation.

TOMASULO ALGORITHM:

Tomasulo's algorithm is a **dynamic scheduling algorithm** used in modern processors to allow **in order issue, out-of-order execution and out-of-order commit** while handling data hazards efficiently. The algorithm uses reservation stations, **register renaming**, and a **common data bus (CDB)** to mitigate Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW) dependencies. Unlike static scheduling, which stalls the pipeline when hazards occur, Tomasulo's approach allows independent instructions to proceed by dynamically resolving operand dependencies. Reservation stations hold instructions until their operands are available, and register renaming prevents false dependencies by ensuring each instruction has a unique destination. The execution of instructions is based on operand availability rather than program order, improving efficiency and resource utilization.

One of the key features of Tomasulo's algorithm is the Common Data Bus (CDB), which allows instructions to broadcast their results to all waiting units, enabling immediate forwarding and reducing stalls. This is particularly useful in floating-point units, where latency is high. Widely used in modern out-of-order processors, Tomasulo's method enhances instruction-level parallelism, leading to significant performance improvements over traditional in-order execution.

ASSUMPTIONS:

1. Given that there are 32 registers. It is assumed that 16 registers can hold integer values and 16 registers can hold floating point values.
2. Reservation Stations: The following assumptions are made regarding the reservation stations and buffers which is equal to the number of hardware components available.
    a. Integer Operations:
    Adder: 2 reservation stations
    Multiplier: 2 reservation stations
    Logical Operations: 2 reservation stations

    b. Floating Point Operations:
    Adder: 4 reservation stations
    Multiplier: 4 reservation stations
    Logical Operations: 2 reservation stations

    c. Load and Store: 2 buffers each
3. No control hazards.

ABOUT THE CODE:

<u>Data Structures in the Implementation:</u>

1. Instruction Class

- **Purpose:** Represents an individual operation (arithmetic, memory load/store, etc.) along with its operands, timing information (issue, start, complete, write-back cycles), and the reservation station that will handle it.

- **Attributes:**
    - op: Operation code (e.g., FADD, IADD, LD).
    - `dest`, `src1`, `src2`: Destination and source registers.
    - `address`: For memory instructions.
    - Timing fields (`issue`, `start`, `complete`, `write`) capture the progress of the instruction through the pipeline.
    - `res_station`: The name of the reservation station allocated for the instruction.

2. ReservationStation Class

- **Purpose:** Acts as a holding area for instructions waiting for execution. It holds operands (or tags indicating pending results), the operation code, the destination register, and additional bookkeeping details.

- **Attributes:**
    - `name` and `op_type`: Identifier and type of operation the RS is dedicated to.
    - `busy`: Indicates if the station is in use.
    - `op`: The operation currently assigned.
    - `dest`: Destination register.
    - `Vj`, `Vk`: Values for operands when available.
    - `Qj`, `Qk`: Tags that point to reservation stations whose results are still pending.
    - `exec_remaining`: Remaining cycles required for execution.
    - `broadcasted`: Flag to check if the result has been broadcasted on the CDB.
    - `result`: Stores the computed result.

3. Register Files

- **FP_registers and int_registers:**
    - Mapped as dictionaries for floating-point and integer registers respectively.
    - Each register is associated with a value and a tag (if the register is waiting on a pending result). This tag is used for register renaming, ensuring that the correct result is forwarded once available.

4. Load and Store Buffers

- These are specialized reservation stations for memory operations.
    - **Load Buffers:** Hold load instructions (LD). They track the memory address and destination register.
    - **Store Buffers:** Hold store instructions (ST). They track the source register (whose value is to be stored) and the memory address.

5. Grouping of Reservation Stations

- Different groups of reservation stations are created for various operation types:
    - **FP arithmetic (FADD, FSUB) and FP multiplication/division (FMUL, FDIV).**
    - **FP logical operations (FAND, FOR, FXOR, FNOT).**
    - **Integer arithmetic (IADD, ISUB) and multiplication/division (IMUL, IDIV).**
    - **Integer logical operations (IAND, IOR, IXOR, INOT).**

These groups ensure that operations are issued to the appropriate RS and help manage functional units separately.

How the Code Replicates the Tomasulo Algorithm

1. Instruction Issue

- **Process:**
  - The simulator checks if there is a free reservation station (or load/store buffer) for the instruction.
  - For memory instructions (LD/ST), it selects a free buffer; for others, it finds a free RS based on the operation type.

- **Register Renaming:**
  - Once an instruction is issued, the destination register is updated with a tag indicating the RS handling the operation. This prevents false dependencies and allows for out-of-order execution.

2. Execution Phase

- **Operand Availability:**
  - Before execution, the RS checks whether the necessary operands (or their tags) are ready.
  - If an operand is not available (indicated by a tag in `Qj` or `Qk`), the RS waits.

- **Timing:**
  - The simulator uses the `exec_remaining` attribute to simulate execution cycles. Execution begins after the issue cycle and only decrements if operands are ready.

3. Write-Back and Result Broadcasting

- **Result Computation:**
  - Once an RS finishes executing (i.e., `exec_remaining` reaches zero), it computes the result using the `compute_result` method.

- **Broadcasting**
  - The result is then broadcast on the common data bus (CDB) to all waiting RS entries. If another RS is waiting on the result (tracked by `Qj` or `Qk`), it updates its operand value.

- **Register Update:**

- The register file is updated with the computed result if the register's tag matches the RS name. This ensures that only the latest result is written back.

4. Maintaining Pipeline State

- **Cycle-Based Simulation:**
    - The simulation is driven by cycles (`current_cycle`), with each cycle simulating the write-back, issue, and execution stages.

- **State Logging:**
    - At the end of each cycle, the simulator prints the state of instructions, reservation stations, register files, and a snapshot of memory. This provides a clear view of the pipeline's progress and how instructions move through various stages.

5. Memory Handling

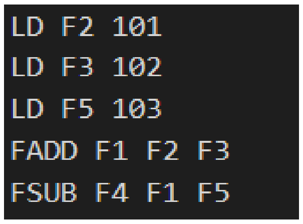
- **Load/Store Buffers:**
    - Memory operations are issued to specialized buffers. Loads retrieve data from memory while stores write the computed values back to memory.
    - The code handles the addressing and value forwarding similar to how a hardware implementation would use a dedicated unit for memory accesses.

HANDLING DEPENDENCIES

In modern computer architectures, dependency handling plays a crucial role in ensuring correct and efficient execution of instructions. Dependencies arise due to the use of shared resources such as registers and memory locations.

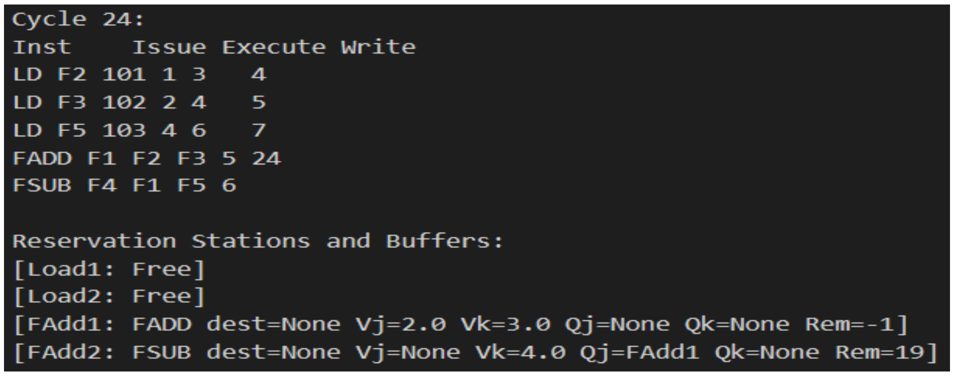
1. RAW: RAW dependency occurs when an instruction depends on the result of a previous instruction before it can execute. This dependency must be resolved to maintain the correct execution order. Operand forwarding is used to mitigate RAW dependencies, allowing an instruction to use a result before it is officially written back to a register.

INPUT:



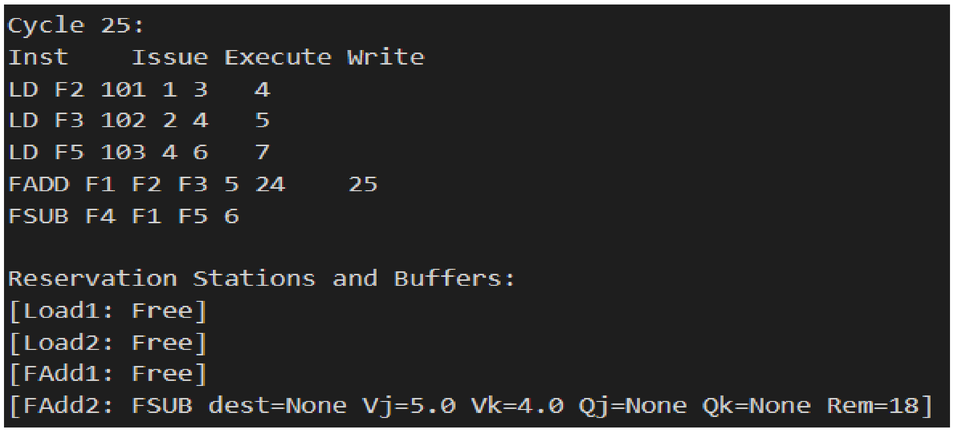
```
LD F2 101
LD F3 102
LD F5 103
FADD F1 F2 F3
FSUB F4 F1 F5
```

OUTPUT:



```
Cycle 24:
Inst    Issue Execute Write
LD F2 101 1 3   4
LD F3 102 2 4   5
LD F5 103 4 6   7
FADD F1 F2 F3 5 24
FSUB F4 F1 F5 6

Reservation Stations and Buffers:
[Load1: Free]
[Load2: Free]
[FAdd1: FADD dest=None Vj=2.0 Vk=3.0 Qj=None Qk=None Rem=-1]
[FAdd2: FSUB dest=None Vj=None Vk=4.0 Qj=FAdd1 Qk=None Rem=19]
```

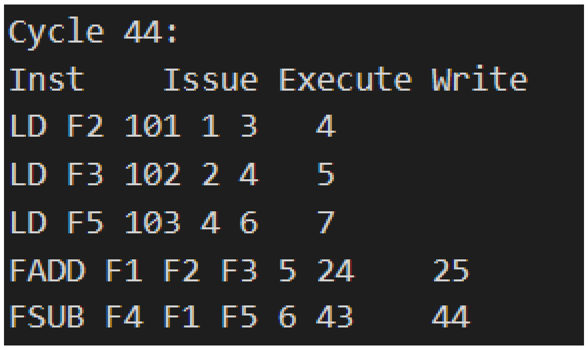


```
Cycle 25:
Inst    Issue Execute Write
LD F2 101 1 3   4
LD F3 102 2 4   5
LD F5 103 4 6   7
FADD F1 F2 F3 5 24    25
FSUB F4 F1 F5 6

Reservation Stations and Buffers:
[Load1: Free]
[Load2: Free]
[FAdd1: Free]
[FAdd2: FSUB dest=None Vj=5.0 Vk=4.0 Qj=None Qk=None Rem=18]
```

FSUB depends on FADD's result which is computed by the end of 24th clock cycle. Instead of waiting for FADD to write its result to the register file, the value is forwarded directly to FSUB in cycle 25, allowing it to proceed without unnecessary stalls. This forwarding effect can be seen in the remaining time for the FSUB operation. This technique optimizes execution efficiency by reducing pipeline delays.

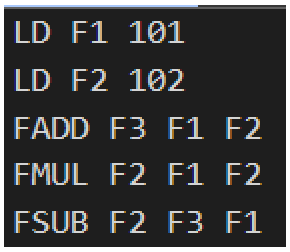
FINAL OUTPUT:



```
Cycle 44:
Inst    Issue Execute Write
LD F2 101 1 3   4
LD F3 102 2 4   5
LD F5 103 4 6   7
FADD F1 F2 F3 5 24    25
FSUB F4 F1 F5 6 43    44
```

2. WAR: A Write After Read (WAR) dependency occurs when an instruction needs to read a value from a register before a later instruction writes to that same register. The issue arises if the write operation happens too soon, overwriting the value before it is read, leading to incorrect results.

INPUT:

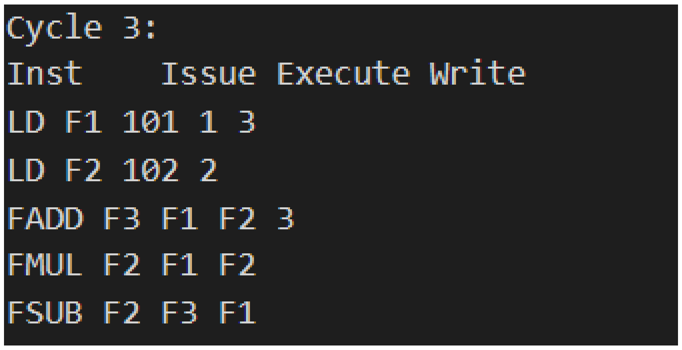


```
LD F1 101
LD F2 102
FADD F3 F1 F2
FMUL F2 F1 F2
FSUB F2 F3 F1
```

In the fourth line of the above assembly code, the result is written into register F2 which is being read in the third line. This is an example of WAR dependency.

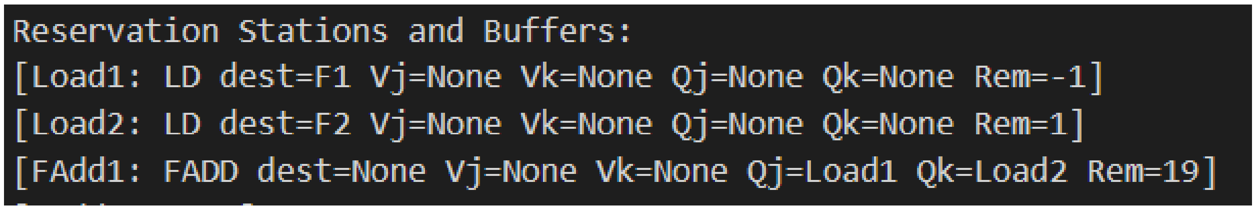
In scoreboard approach, when there is a WAR dependency the FMUL instruction would be stalled until the FADD instruction reads the operands. This dependency is overcome by using register renaming technique.

OUTPUT:

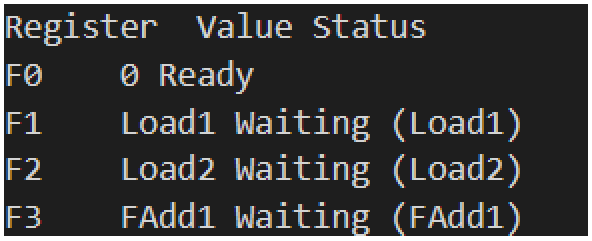


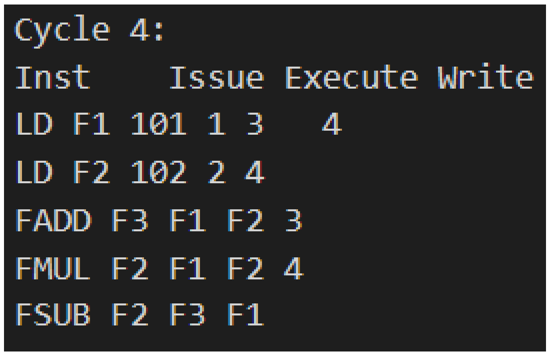
```
Cycle 3:
Inst    Issue Execute Write
LD F1 101 1 3
LD F2 102 2
FADD F3 F1 F2 3
FMUL F2 F1 F2
FSUB F2 F3 F1
```

In clock cycle 3 when the FADD instruction is issued, the register F2 contains the value present in the memory location 101 which is 2 as below. As the load operation is not completed yet, the tag of the reservation station that provides the result is stored in the instruction to get result while broadcast.

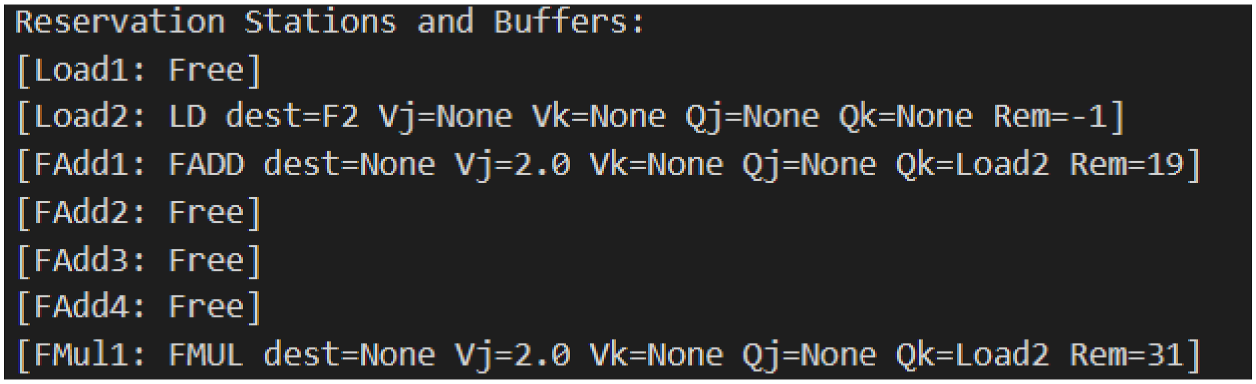


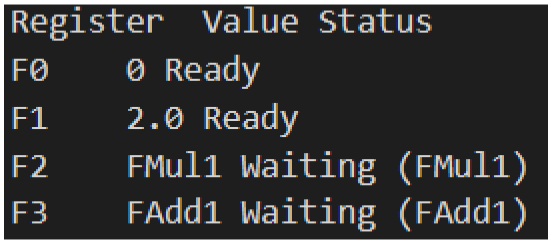
```
Reservation Stations and Buffers:
[Load1: LD dest=F1 Vj=None Vk=None Qj=None Qk=None Rem=-1]
[Load2: LD dest=F2 Vj=None Vk=None Qj=None Qk=None Rem=1]
[FAdd1: FADD dest=None Vj=None Vk=None Qj=Load1 Qk=Load2 Rem=19]
```


```
Register  Value Status
F0    0 Ready
F1    Load1 Waiting (Load1)
F2    Load2 Waiting (Load2)
F3    FAdd1 Waiting (FAdd1)
```

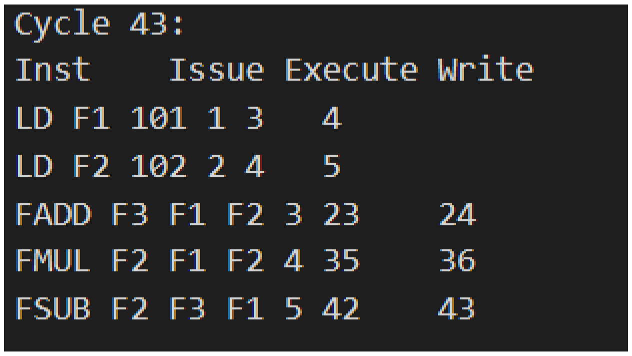

```
Cycle 4:
Inst    Issue Execute Write
LD F1 101 1 3   4
LD F2 102 2 4
FADD F3 F1 F2 3
FMUL F2 F1 F2 4
FSUB F2 F3 F1
```

In clock cycle 4 after the issual of 4th instruction, as FADD needs to write into F2, the register status is updated by the tag of the reservation station computing this result. The result which has to be loaded into F2 from memory is broadcasted through the CDB when it is available. Based on the tag present (in this case it is Load2). So, the register F2 is renamed by the tag of Load2 reservation station for FADD instruction and FMUL reservation station for writing it back to F2. The changes are shown below.


```
Reservation Stations and Buffers:
[Load1: Free]
[Load2: LD dest=F2 Vj=None Vk=None Qj=None Qk=None Rem=-1]
[FAdd1: FADD dest=None Vj=2.0 Vk=None Qj=None Qk=Load2 Rem=19]
[FAdd2: Free]
[FAdd3: Free]
[FAdd4: Free]
[FMul1: FMUL dest=None Vj=2.0 Vk=None Qj=None Qk=Load2 Rem=31]
```

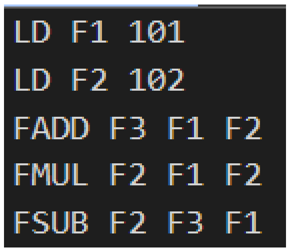

```
Register  Value Status
F0    0 Ready
F1    2.0 Ready
F2    FMul1 Waiting (FMul1)
F3    FAdd1 Waiting (FAdd1)
```

FINAL OUTPUT:



```
Cycle 43:
Inst    Issue Execute Write
LD F1 101 1 3   4
LD F2 102 2 4   5
FADD F3 F1 F2 3 23    24
FMUL F2 F1 F2 4 35    36
FSUB F2 F3 F1 5 42    43
```

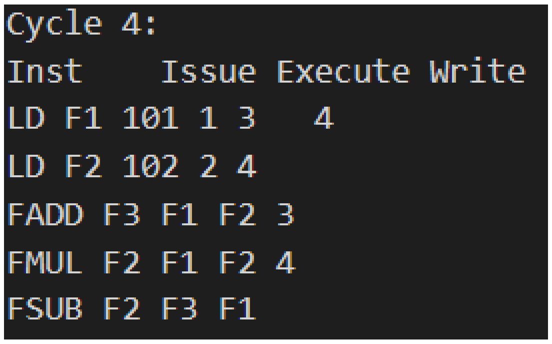
3. <u>WAW:</u>
INPUT:



```
LD F1 101
LD F2 102
FADD F3 F1 F2
FMUL F2 F1 F2
FSUB F2 F3 F1
```

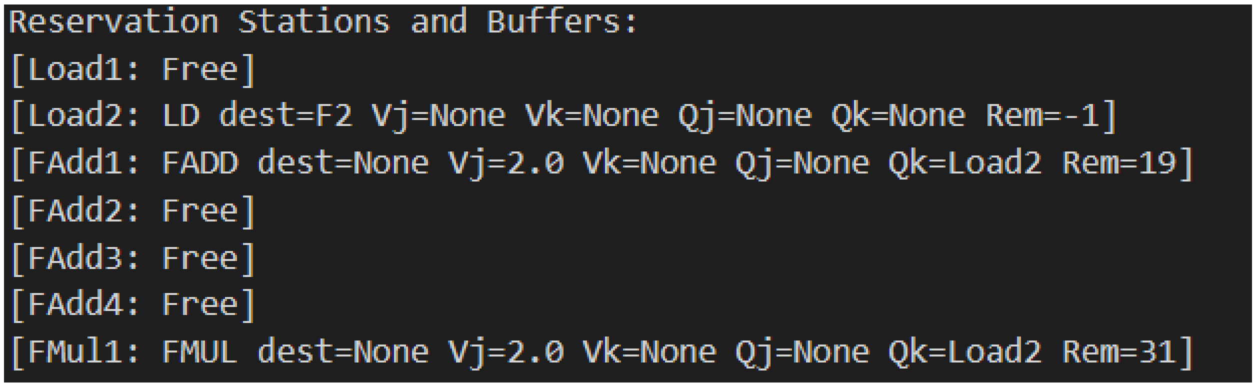
In the fifth line of the above assembly code, the result is written into register F2 which is being used to write a result in the fourth line. This is an example of WAW dependency. If these dependencies are not handled properly, data race conditions might occur which might lead to inconsistent computational results. In Scoreboard technique, the fifth instruction would be stalled until the fourth instruction has written back. But in Tomasulo algorithm, register renaming technique is used.

OUTPUT:

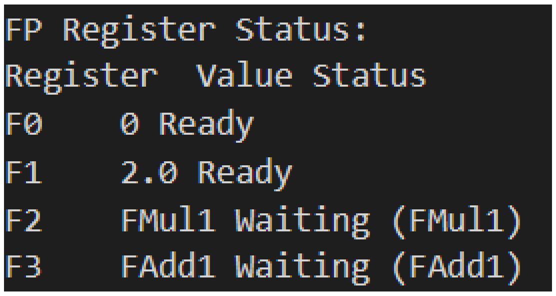


```
Cycle 4:
Inst    Issue Execute Write
LD F1 101 1 3   4
LD F2 102 2 4
FADD F3 F1 F2 3
FMUL F2 F1 F2 4
FSUB F2 F3 F1
```

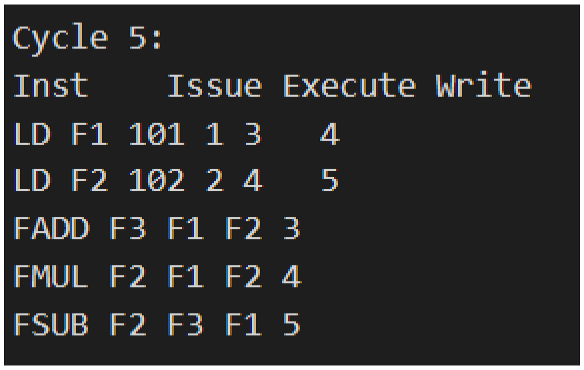
In the fourth clock cycle, when the fifth instruction is not yet issued, the register F2 is waiting for the result from FMul1 reservation station which contains the instruction present in the fourth line of the input code as shown below.



```
Reservation Stations and Buffers:
[Load1: Free]
[Load2: LD dest=F2 Vj=None Vk=None Qj=None Qk=None Rem=-1]
[FAdd1: FADD dest=None Vj=2.0 Vk=None Qj=None Qk=Load2 Rem=19]
[FAdd2: Free]
[FAdd3: Free]
[FAdd4: Free]
[FMul1: FMUL dest=None Vj=2.0 Vk=None Qj=None Qk=Load2 Rem=31]
```

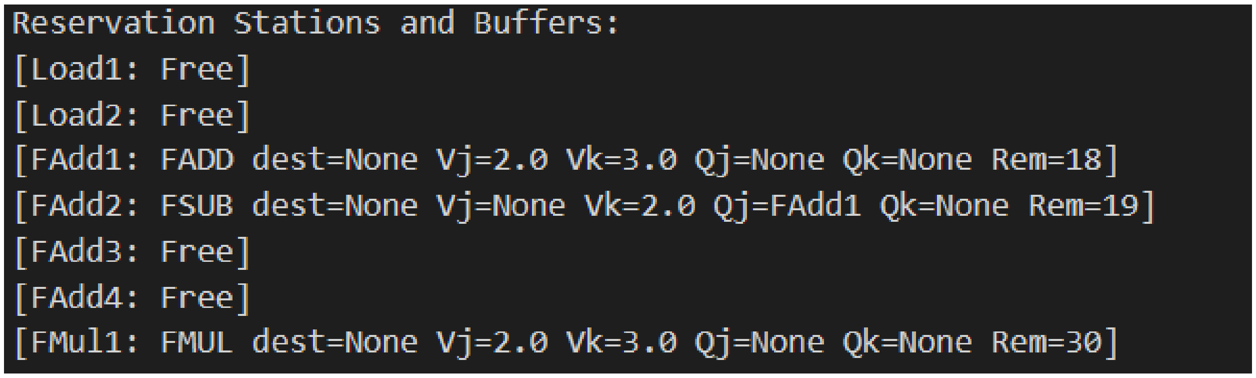


```
FP Register Status:
Register  Value Status
F0    0 Ready
F1    2.0 Ready
F2    FMul1 Waiting (FMul1)
F3    FAdd1 Waiting (FAdd1)
```

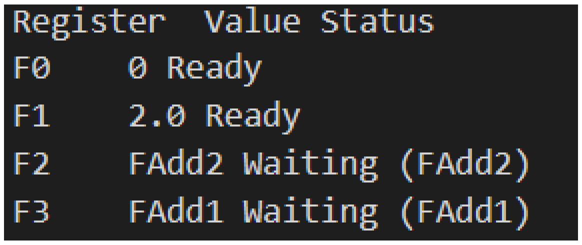


```
Cycle 5:
Inst    Issue Execute Write
LD F1 101 1 3   4
LD F2 102 2 4   5
FADD F3 F1 F2 3
FMUL F2 F1 F2 4
FSUB F2 F3 F1 5
```

In clock cycle 5, when FSUB instruction is issued, the write in F2 by FMul1 is overwritten by FAdd2 reservation stations result. The tag of FAdd2 is now present in the F2 register indicating this. If F2 is further read in the code, tag of FAdd2 reservation station is used so that CDB can broadcast the data to the necessary reservation stations and buffers. Hence, register renaming is performed by renaming F2 with the tag of the structural unit that produces the desired result.



```
Reservation Stations and Buffers:
[Load1: Free]
[Load2: Free]
[FAdd1: FADD dest=None Vj=2.0 Vk=3.0 Qj=None Qk=None Rem=18]
[FAdd2: FSUB dest=None Vj=None Vk=2.0 Qj=FAdd1 Qk=None Rem=19]
[FAdd3: Free]
[FAdd4: Free]
[FMul1: FMUL dest=None Vj=2.0 Vk=3.0 Qj=None Qk=None Rem=30]
```


```
Register  Value Status
F0    0 Ready
F1    2.0 Ready
F2    FAdd2 Waiting (FAdd2)
F3    FAdd1 Waiting (FAdd1)
```

SAMPLE INPUT
A sample assembly code of 22 lines is given as input to the code. The assembly code is as follows.

```
LD R0 10
LD R1 11
IADD R2 R1 R0
IMUL R3 R1 R0
IADD R4 R2 R3
ST 12 R2
ST 13 R3
ST 14 R4
LD F0 100
LD F1 101
FADD F2 F0 F1
FMUL F3 F2 F0
FADD F4 F2 F3
FMUL F5 F2 F4
FMUL F2 F5 F4
FAND F3 F2 F5
FOR F6 F2 F5
ST 102 F2
ST 103 F3
ST 104 F4
ST 105 F5
ST 106 F6
```

The code has structural hazards as well as all kinds of dependencies.

OUTPUT
The output of the entire simulation is written into a .txt file.
For each cycle during the execution, the following are written into the txt file.

1. Execution table
2. Reservation Stations status
3. Load Store buffers status
4. Register status

The output txt file obtained for the above assembly code is also attached.

FURTHER IMPROVEMENTS

To evolve the current simulation into a more advanced and realistic model of modern out-of-order processors, several improvements can be considered

1. **Reorder Buffer (ROB):**
   Add a reorder buffer to allow in-order commit of instructions. This would help to maintain precise exceptions and support speculative execution, letting instructions complete out-of-order while still committing in program order.

2. **Branch Prediction and Speculative Execution:**
   Integrate branch prediction to handle control hazards. With speculative execution, the simulator could begin processing instructions from predicted branches and later rollback if the prediction was incorrect.

3. **Multiple Issue and Superscalar Support:**
   Extend the simulator to issue more than one instruction per cycle. This requires managing several functional units and reservation stations concurrently, which can also be supported by multiple ROB entries.

4. **Improved Memory Disambiguation:**
   Enhance load and store handling by adding memory dependency checking. A more advanced system might include a dedicated memory disambiguation unit to better resolve potential conflicts between loads and stores.

5. **Dynamic Scheduling Enhancements:**
   Incorporate more sophisticated hazard detection techniques to manage not only RAW hazards (which Tomasulo already addresses) but also structural hazards. This could involve dynamically allocating or reusing reservation stations based on the current workload.

6. **Common Data Bus (CDB) Improvements:**
   Although the simulation already forwards results, you could refine how results are broadcast to reduce the contention on the CDB. This might involve partitioning the CDB or simulating multiple broadcast channels.

7. **Detailed Timing and Power/Area Metrics:**
   Incorporate more detailed cycle-level performance metrics, such as execution unit utilization or energy consumption estimates, to analyze the trade-offs of advanced scheduling techniques.

8. **Speculative State Recovery:**
   Implement mechanisms for precise state recovery in case of mis-speculations. This

includes saving register and memory states prior to speculative execution and restoring them if needed.

Each of these improvements would add layers of complexity that mirror more advanced and realistic implementations of the Tomasulo algorithm in modern processors.

CONCLUSION:
Handling RAW, WAR and WAW dependencies efficiently is essential in achieving high performance in modern processors. Techniques such as operand forwarding, instruction scheduling, register renaming, and out-of-order execution help minimize stalls and improve instruction throughput. By implementing these strategies, processors can execute instructions with minimal interference while maintaining program correctness.