Final term exam of [IM20111201] アーキテクチャ学 (Computer Architecture)

C4TL1205 - LANDY Lucas (COLABS Student) - lucas.landy.t3@dc.tohoku.ac.jp

Table of Contents

Question (a)

The problem requires analyzing the execution of the given DAXPY loop using Tomasulo's algorithm in a single-issue MIPS pipeline. Specifically, we need to determine the total cycle count per iteration, stall cycles for each instruction, and when each instruction begins execution in the first three iterations.

Tomasulo's Algorithm

Tomasulo's algorithm is a dynamic scheduling technique used in out-of-order execution to improve processor performance by resolving data hazards dynamically at runtime. It allows instructions to be issued and executed out-of-order while still preserving program correctness by using reservation stations (RS) and a Common Data Bus (CDB) for operand communication. In traditional in-order execution, an instruction must wait until all previous instructions are completed before it can execute, leading to pipeline stalls. Tomasulo's algorithm allows instructions to execute as soon as their operands are available, even if previous instructions are still being processed. This technique reduces pipeline stalls caused by RAW (Read-After-Write) hazards, improves instruction throughput, and enhances Instruction-Level Parallelism (ILP).

- Key features relevant to this problem:
    1. Functional units are not pipelined → Once a unit is occupied, it must complete execution before processing another instruction.
    2. One result per cycle can be written to CDB → This can cause stalls if multiple instructions try to write results in the same cycle.
    3. Instructions issue if a reservation station is available.
    4. Execution starts only when all operands are ready.

Understanding the Given Code

The **DAXPY loop** (Double-Precision A·X Plus Y) is a common operation in numerical computing, performing:

$$Y(i) = a \cdot X(i) + Y(i)$$

Where:

- $X(i)$ and $Y(i)$ are vector elements.
- a is a scalar.
- The loop updates Y(i) sequentially.

Code Breakdown

```
DADDIU R4, R1, #800   # R4 = upper bound for X

Foo:
    L.D F2, 0(R1)      # Load X(i) into F2
    MUL.D F4, F2, F0   # F4 = a * X(i)
    L.D F6, 0(R2)      # Load Y(i) into F6
    ADD.D F6, F4, F6   # F6 = a * X(i) + Y(i)
    S.D F6, 0(R2)      # Store updated Y(i)
    DADDIU R1, R1, #8  # Increment X index
    DADDIU R2, R2, #8  # Increment Y index
    DSLTU R3, R1, R4   # Check if end of loop
    BNEZ R3, Foo       # Loop back if not done
```

The assembly implementation of the loop follows a structured sequence. First, it loads $X(i)$ from memory into register F2, then multiplies it by the scalar a, storing the result in F4. Next, it loads $Y(i)$ into F6, adds the scaled $X(i)$, and writes the updated value of $Y(i)$ back to memory. The integer registers R1 and R2 are incremented to point to the next elements in X and Y, and a comparison checks whether the loop should continue. This memory access pattern ensures sequential access, making it efficient for cache usage.
When executed under Tomasulo's Algorithm, several performance factors come into play. RAW (Read-After-Write) hazards occur because ADD.D depends on MUL.D, and S.D depends on ADD.D. Since the FP units are not pipelined, the MUL.D operation takes 15 cycles while ADD.D takes 10 cycles, causing stalls in execution. Additionally, since only one instruction can write to the Common Data Bus (CDB) per cycle, multiple operations completing simultaneously can create CDB contention, further delaying execution.
Optimizations such as pipelined floating-point units can reduce stalls, while register renaming eliminates WAW and WAR hazards, allowing greater instruction-level parallelism. Loop unrolling can further improve performance by reducing branch overhead and increasing instruction throughput. These techniques help minimize execution stalls and improve efficiency when executing the DAXPY loop using Tomasulo's dynamic scheduling approach.

Breakdown of Each Instruction

The DAXPY loop consists of floating-point and integer instructions, each playing a crucial role in updating vector Y. The floating-point operations involve loading data from memory, performing multiplication and addition, and storing results, while the integer operations handle loop control and pointer arithmetic. The table below summarizes each instruction, followed by a detailed breakdown.

Instruction	Description	Dependency
L.D F2, 0(R1)	Load $X(i)$ into $F2$	Memory latency
MUL.D F4, F2, F0	Compute $a \times X(i)$	RAW dependency on F2
L.D F6, 0(R2)	Load $Y(i)$ into $F6$	Memory latency
ADD.D F6, F4, F6	Compute $aX(i) + Y(i)$	RAW dependency on F4 and F6
S.D F6, 0(R2)	Store $Y(i)$	RAW dependency on F6

Instruction	Description	Dependency
DADDIU R1, R1, #8	Increment X index	Integer operation (1 cycle)
DADDIU R2, R2, #8	Increment Y index	Integer operation (1 cycle)
DSLTU R3, R1, R4	Compare indices	Integer operation (1 cycle)
BNEZ R3, Foo	Branch if needed	Integer operation (1 cycle)

Functional Unit Latencies

The latency of functional units (FUs) determines how long an instruction stays in the execution (EX) stage before producing a result. Since Tomasulo's Algorithm allows out-of-order execution, instructions can execute as soon as their operands are available, but execution time depends on how long a functional unit takes to process an instruction.

In this problem, the given latencies are as follows:

FU Type	Latency (Cycles)	No. of FUs
Integer	1	1
FP Adder (ADD.D)	10	1
FP Multiplier (MUL.D)	15	1
Load/Store (L.D / S.D)	1	-

Execution Schedule (First Iteration)

In a single-issue pipeline, only one instruction can issue per cycle, meaning instructions enter the pipeline sequentially and execute based on their dependencies and resource availability. Below is a breakdown of how the first three iterations of the loop execute in Tomasulo's Algorithm while handling latencies, stalls, and instruction overlaps.

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	L.D F2,0(R1)	1	2	2	3	Load takes 1 cycle
1	MUL.D F4,F2,F0	1	4-18	-	19	RAW on F2
1	L.D F6,0(R2)	2	5	5	6	Can execute independently
1	ADD.D F6,F4,F6	3	20-29	-	30	Waits for F4 (RAW)
1	S.D F6,0(R2)	4	31	31	-	Waits for F6
1	DADDIU R1,R1,#8	5	10	-	11	Integer op (1 cycle)

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	DADDIU R2,R2,#8	5	11	-	12	Integer op (1 cycle)
1	DSLTU R3,R1,R4	6	12	-	13	Integer op (1 cycle)
1	BNEZ R3, Foo	6	13	-	14	Integer op (1 cycle)
2	L.D F2,0(R1)	7	14	14	15	Next iteration starts earlier
2	MUL.D F4,F2,F0	7	16-30	-	31	Execution overlaps with prior ADD.D
2	L.D F6,0(R2)	8	15	15	16	Can execute alongside MUL.D
2	ADD.D F6,F4,F6	9	32-41	-	42	Waits for F4
2	S.D F6,0(R2)	10	43	43	-	Waits for F6
2	DADDIU R1,R1,#8	11	16	-	17	Integer op (1 cycle)
2	DADDIU R2,R2,#8	11	17	-	18	Integer op (1 cycle)
2	DSLTU R3,R1,R4	12	18	-	19	Integer op (1 cycle)
2	BNEZ R3, Foo	12	19	-	20	Integer op (1 cycle)
3	L.D F2,0(R1)	13	22	22	23	Execution overlaps further
3	MUL.D F4,F2,F0	13	24-38	-	39	Execution overlaps with prior ADD.D
3	L.D F6,0(R2)	14	23	23	24	Can execute alongside MUL.D
3	ADD.D F6,F4,F6	15	40-49	-	50	Waits for F4
3	S.D F6,0(R2)	16	51	51	-	Waits for F6
3	DADDIU R1,R1,#8	17	24	-	25	Integer op (1 cycle)
3	DADDIU R2,R2,#8	17	25	-	26	Integer op (1 cycle)

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
3	DSLTU R3,R1,R4	18	26	-	27	Integer op (1 cycle)
3	BNEZ R3, Foo	18	27	-	28	Integer op (1 cycle)

First Iteration (Processing X(1) and Y(1))

1. Cycle 1: `L.D F2, 0(R1)` issues, loading X(1) into F2. Since a load takes 1 cycle, execution happens in Cycle 2, and the value is written to the CDB in Cycle 3.
2. Cycle 1: `MUL.D F4, F2, F0` issues but cannot execute immediately due to a RAW hazard on F2 (waiting for `L.D`). It starts executing in Cycle 4 and takes 15 cycles (Cycles 4-18), completing in Cycle 19.
3. Cycle 2: `L.D F6, 0(R2)` issues and loads Y(1) into F6 in Cycle 5, writing it back in Cycle 6. This executes independently of `MUL.D`.
4. Cycle 3: `ADD.D F6, F4, F6` issues, but it must wait for MUL.D to finish (RAW dependency on F4). Since `MUL.D` finishes in Cycle 19, `ADD.D` starts in Cycle 20 and executes for 10 cycles (Cycles 20-29), writing to the CDB in Cycle 30.
5. Cycle 4: `S.D F6, 0(R2)` issues but must wait for `ADD.D` to finish (RAW dependency on F6). Since `ADD.D` writes to the CDB in Cycle 30, `S.D` executes in Cycle 31, storing the updated Y(1).
6. Cycle 5-6: The integer operations (`DADDIU R1/R2`, `DSLTU`, `BNEZ`) execute sequentially, completing by Cycle 14.

After Cycle 14, the processor moves to Iteration 2.

Second Iteration (Processing X(2) and Y(2))

1. Cycle 7: `L.D F2, 0(R1)` issues, loading X(2) in Cycle 14 and writing to CDB in Cycle 15.
2. Cycle 7: `MUL.D F4, F2, F0` issues but waits for F2. It executes from Cycle 16-30, writing its result in Cycle 31.
3. Cycle 8: `L.D F6, 0(R2)` issues, loading Y(2) in Cycle 15 and writing to CDB in Cycle 16.
4. Cycle 9: `ADD.D F6, F4, F6` issues but waits for F4. Since `MUL.D` completes in Cycle 31, `ADD.D` executes from Cycle 32-41, writing to CDB in Cycle 42.
5. Cycle 10: `S.D F6, 0(R2)` issues but must wait for F6. Since `ADD.D` writes in Cycle 42, `S.D` stores the updated Y(2) in Cycle 43.
6. Cycle 11-12: Integer operations execute sequentially, finishing in Cycle 20.

By Cycle 20, the processor moves to Iteration 3.

Third Iteration (Processing X(3) and Y(3))

1. Cycle 13: `L.D F2, 0(R1)` issues, loading X(3) in Cycle 22 and writing to CDB in Cycle 23.
2. Cycle 13: `MUL.D F4, F2, F0` issues but waits for F2. It executes from Cycle 24-38, writing its result in Cycle 39.
3. Cycle 14: `L.D F6, 0(R2)` issues, loading Y(3) in Cycle 23 and writing to CDB in Cycle 24.

4. Cycle 15: `ADD.D F6, F4, F6` issues but waits for F4. Since `MUL.D` completes in Cycle 39, `ADD.D` executes from Cycle 40-49, writing to CDB in Cycle 50.
5. Cycle 16: `S.D F6, 0(R2)` issues but must wait for F6. Since `ADD.D` writes in Cycle 50, `S.D` stores the updated Y(3) in Cycle 51.
6. Cycle 17-18: Integer operations execute sequentially, finishing in Cycle 28.

By Cycle 28, the processor is ready to start Iteration 4.

Key Observations in the Execution Schedule

1. Floating-Point Latency Bottlenecks Execution

- MUL.D (15 cycles) and ADD.D (10 cycles) dominate execution time.
- Since only one FP multiplier and one FP adder exist, each operation must wait for the previous one to finish.

2. Memory Loads Can Execute Independently

- `L.D F6, 0(R2)` is independent of `MUL.D`, so it starts earlier in each iteration without causing stalls.

3. CDB Contention Can Cause Stalls

- Only one instruction can write to the CDB per cycle, meaning that MUL.D, ADD.D, and S.D might have to wait even if they complete execution.

4. Integer Instructions Are Fast

- Integer operations (DADDIU, DSLTU, BNEZ) take only 1 cycle each, so they do not significantly impact performance.

How Execution Overlaps Between Iterations

From Iteration 2 onward, execution overlaps occur:

- Cycle 7: `L.D` from Iteration 2 starts before `ADD.D` from Iteration 1 finishes.
- Cycle 16: `MUL.D` from Iteration 2 starts before `ADD.D` from Iteration 1 completes.
- Cycle 24: `MUL.D` from Iteration 3 starts before `ADD.D` from Iteration 2 completes.

This parallel execution improves efficiency, but full parallelism is limited by functional unit constraints.

Conclusion: Execution Time Per Iteration

- First Iteration: 30-35 cycles (due to cold start).
- Subsequent Iterations: ~20-25 cycles per iteration (overlapping execution).
- Performance Bottlenecks: FP multiplier latency (MUL.D) and CDB contention.

Optimizations (if allowed) to improve execution:

- Pipelined FPUs → Reduce stall time for MUL.D and ADD.D.
- Register Renaming → Eliminate false dependencies for better ILP.
- Loop Unrolling → Reduces branch overhead and increases instruction parallelism.

Question (b)

In part (b), we assume the following modifications to the execution model:

1. Two-Issue Pipeline: The processor can issue two instructions per cycle.
2. Fully Pipelined Floating-Point Units: The FP multiplier and FP adder can start execution every cycle, meaning no stalls due to waiting for an available functional unit.
3. Other Assumptions Remain the Same: No forwarding, execution follows Tomasulo's algorithm.

Two-Issue Pipeline

A two-issue pipeline is an enhancement over a single-issue pipeline, allowing two instructions to be issued per cycle instead of one. This increases instruction throughput and improves performance by enabling parallel execution of independent operations. However, performance improvements depend on resource availability, instruction dependencies, and hardware constraints such as functional unit availability and result forwarding mechanisms.

In this setup, the processor attempts to issue two instructions per clock cycle if sufficient reservation stations, functional units, and operands are available. However, issuing two instructions simultaneously is not always possible due to:

- Data hazards: If one instruction depends on the result of another, it must wait before issuing.
- Structural hazards: If both instructions require the same functional unit, only one can issue.
- Control hazards: If the second instruction is a branch-dependent instruction, it may need to wait for the branch resolution.

In the DAXPY loop, a common dual-issue pair would be:

- L.D and MUL.D (Loading X(i) while computing a × X(i)).
- L.D and ADD.D (Loading Y(i) while performing addition).
- S.D and DADDIU (Storing Y(i) while updating memory pointers).

This allows better parallelism compared to a single-issue pipeline, where only one instruction is issued per cycle.

In this scenario, floating-point (FP) units are fully pipelined, meaning new FP operations can start execution before previous ones finish. This minimizes stalls and enables better instruction overlap. For example:

- MUL.D starts execution on Cycle 2. Since the FP multiplier is pipelined, another MUL.D can start a few cycles later while the first one is still processing.
- ADD.D can start immediately after MUL.D, rather than waiting for the FP unit to become available.
- Multiple floating-point instructions can be in different pipeline stages simultaneously, improving instruction-level parallelism (ILP).

The key benefit of pipelined FPUs is that instead of waiting for the entire execution latency of MUL.D (15 cycles) or ADD.D (10 cycles), subsequent instructions can enter the pipeline before the previous ones finish. This drastically reduces execution stalls compared to a non-pipelined system.

Despite having dual-issue capabilities and pipelined FPUs, the Common Data Bus (CDB) remains a performance bottleneck. The CDB is responsible for broadcasting computed results from functional units to

reservation stations and registers. However, only one result can be written to the CDB per cycle, leading to potential stalls in the following cases:

- MUL.D and ADD.D complete in the same cycle but only one can write its result to the CDB, forcing the other to stall.
- L.D completes at the same time as an FP operation, but the CDB can handle only one result per cycle, delaying subsequent instruction execution.
- Multiple stores (S.D) waiting for the CDB to forward data may also experience delays.

To optimize execution, out-of-order scheduling under Tomasulo's Algorithm helps reduce CDB contention by reordering instructions, but the single CDB remains a fundamental bottleneck in cases where multiple results are ready to write back simultaneously.

Functional Unit Latencies (Revisited)

Since pipelining is enabled, new instructions can start execution before previous ones finish, reducing stalls and improving instruction throughput. However, the number of available units limits execution parallelism, meaning that some instructions may have to wait for resources to free up. The table below summarizes the execution latencies and availability of each functional unit:

FU Type	Cycles in EX	Pipelined?	Number of Units
Integer (DADDIU, DSLTU, BNEZ)	1	Yes	1
FP Adder (ADD.D)	10	Yes	1
FP Multiplier (MUL.D)	15	Yes	1
Load (L.D)	1	Yes	-
Store (S.D)	1	Yes	-

Key Changes Compared to Part (a)

In this improved pipeline, MUL.D and ADD.D are now pipelined, allowing a new multiplication or addition instruction to start every cycle, even before the previous one finishes. This enables overlapping execution, reducing delays caused by long floating-point operation latencies. Additionally, the two-issue capability allows the processor to issue two instructions per cycle, improving instruction throughput and significantly reducing stalls compared to the single-issue pipeline in Part (a).

Execution Schedule (First Iteration)

In this two-issue pipeline, two instructions are issued per cycle whenever possible, reducing stalls and improving parallel execution. Floating-point units (FPUs) are pipelined, meaning multiple operations can be in progress simultaneously, overlapping execution stages. Below is a detailed breakdown of how the first three iterations of the loop execute, handling dependencies, stalls, and overlapping execution.

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First Issue

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	ADD.D F4,F0,F2	1	5	-	8	Wait for LD
1	S.D F4,0(R1)	2	3	9	-	Wait for ADD
1	DADDIU R1,R1,-8	2	4	-	5	Wait for ALU
1	BNEZ R1,R2,LOOP	3	6	-	-	Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10	-	13	Wait for LD
2	S.D F4,0(R1)	5	8	14	-	Wait for ADD
2	DADDIU R1,R1,-8	5	9	-	10	Wait for ALU
2	BNEZ R1,R2,LOOP	6	11	-	-	Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15	-	18	Wait for LD
3	S.D F4,0(R1)	8	13	19	-	Wait for ADD
3	DADDIU R1,R1,-8	8	14	-	15	Wait for ALU
3	BNEZ R1,R2,LOOP	9	16	-	-	Execution completed

First Iteration (Processing X(1) and Y(1))

1. Cycle 1:
   - `L.D F0, 0(R1)` issues (loads X(1) into F0).
   - `ADD.D F4, F0, F2` issues but waits for `L.D` to complete (RAW hazard).
2. Cycle 2:
   - `L.D` executes and accesses memory in Cycle 3.
   - `S.D F4, 0(R1)` issues but must wait for ADD.D.
   - `DADDIU R1, R1, -8` issues and executes in Cycle 4.
3. Cycle 3-4:
   - `ADD.D` now executes since `L.D` has completed.
   - `BNEZ R1, R2, LOOP` issues in Cycle 3 but must wait for DADDIU.
4. Cycle 5-8:
   - `ADD.D` completes execution in Cycle 8 and writes its result.

- `S.D` waits until `ADD.D` is finished and executes in Cycle 9.
- The first iteration completes by Cycle 9.

Second Iteration (Processing X(2) and Y(2))

1. Cycle 4:
   - `L.D F0, 0(R1)` issues again for the next iteration (X(2)).
   - `ADD.D F4, F0, F2` issues but must wait for `L.D` to complete.
2. Cycle 5-6:
   - `L.D` completes in Cycle 8, allowing `ADD.D` to begin execution in Cycle 10.
3. Cycle 7-13:
   - `ADD.D` executes from Cycle 10-13 and writes to CDB in Cycle 13.
   - `S.D` executes in Cycle 14 once `ADD.D` is done.

Third Iteration (Processing X(3) and Y(3))

1. Cycle 7:
   - `L.D F0, 0(R1)` issues again for X(3).
   - `ADD.D` waits for `L.D` to complete before execution in Cycle 15.
2. Cycle 8-18:
   - `ADD.D` executes from Cycle 15-18 and writes to CDB in Cycle 18.
   - `S.D` executes in Cycle 19 once `ADD.D` is done.
   - The third iteration completes at Cycle 19, and execution continues.

Key Observations and Execution Efficiency

- Instruction Overlapping: `L.D` starts before the previous `ADD.D` has finished, allowing better pipeline utilization.
- Reduced Stalls Due to Dual Issue: DADDIU and S.D execute alongside floating-point operations, reducing delays.
- CDB Bottleneck Still Exists: `ADD.D` and `S.D` must wait their turn to write to the CDB, causing minor stalls.

Performance Gain Analysis & Conclusion

The two-issue pipeline with pipelined floating-point units significantly improves execution efficiency compared to the single-issue pipeline. By allowing two instructions to issue per cycle and enabling overlapping execution, the iteration time is reduced from ~30-35 cycles to ~18-20 cycles, nearly doubling performance.

The key improvements include:

- Instruction-Level Parallelism (ILP): Independent loads, stores, and FP operations now execute in parallel.
- Pipelined FP Units: MUL.D and ADD.D start new executions before previous ones finish, reducing idle time.
- Reduced Stalls: Tomasulo's dynamic scheduling ensures that RAW dependencies are minimized, keeping the pipeline busy.
- Optimized Iteration Overlap: New loop iterations start execution before previous ones complete, maximizing throughput.

Despite these gains, CDB contention remains a limiting factor, as only one result can be written per cycle, occasionally causing minor stalls. However, the overall execution is much faster and more efficient, demonstrating how dual-issue capability and pipelined execution enhance Tomasulo's Algorithm's ability to exploit ILP.

Question (c)

In part (c), the problem asks us to find a sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to contention on the Common Data Bus (CDB).

CDB Contention

The Common Data Bus (CDB) is a critical component in Tomasulo's Algorithm, responsible for broadcasting computed results from functional units (FUs) to reservation stations and registers. It enables operand forwarding, allowing dependent instructions to begin execution as soon as their required values are available.

However, a key limitation of Tomasulo's Algorithm is that only one result can be written to the CDB per cycle. If multiple instructions complete execution simultaneously, only one can write back, while the others must wait, causing stalls. This is especially problematic in high-ILP scenarios, such as a two-issue pipeline with pipelined FPUs, where multiple instructions frequently complete in the same cycle.

For example, if a MUL.D and an ADD.D finish in the same cycle, but the CDB can only handle one result, the second instruction must stall, delaying subsequent dependent instructions. This contention can reduce the benefits of pipelining and dual-issue execution, making out-of-order scheduling and careful instruction balancing essential for minimizing stalls.

Key Causes of CDB Contention:

CDB contention occurs when multiple instructions attempt to write their results to the Common Data Bus (CDB) in the same cycle, but only one result can be written per cycle. This creates stalls, delaying execution and reducing the efficiency of pipelining and out-of-order execution. Below are the primary causes of CDB contention:

1. Multiple floating-point operations completing at the same time

- Since MUL.D (15 cycles) and ADD.D (10 cycles) are pipelined, a scenario may arise where an earlier ADD.D and a later MUL.D finish in the same cycle.
- Example: If `MUL.D F4, F2, F0` and `ADD.D F6, F4, F6` finish at the same time, only one can write to the CDB first, forcing the other to stall until the next cycle.

2. Load instructions completing together

- L.D (Load Double) takes 1 cycle to execute, but in a dual-issue pipeline, two loads can issue in the same cycle. If both L.D instructions complete simultaneously, only one can write back to CDB, while the other must wait.
- Example: `L.D F2, 0(R1)` and `L.D F6, 0(R2)` may complete at the same time, but since only one value can be broadcast, the second L.D has to stall.

3. Store instructions depending on values that must be written to the CDB first

- A store (S.D) needs its operand before writing to memory, but if the operand is waiting for an FP operation to write back first, the store must stall.
- Example: If `ADD.D F6, F4, F6` produces the value needed for `S.D F6, 0(R2)`, but another instruction is already using the CDB, the S.D must wait for ADD.D to write first.

These contention scenarios introduce stalls, reducing instruction throughput and execution efficiency. Optimizations such as reordering instructions, multiple CDBs, and better scheduling can help minimize contention.

Constructing a Problematic Sequence

To force a CDB stall, we need multiple instructions that finish execution in the same cycle and attempt to write to the Common Data Bus (CDB) simultaneously. Since only one result can be written per cycle, any additional instructions completing at the same time must stall, delaying dependent operations.

The following sequence demonstrates a scenario where CDB contention occurs:

```
L.D    F2, 0(R1)    # Load X(i) into F2
L.D    F4, 0(R2)    # Load Y(i) into F4
MUL.D  F6, F2, F0   # Compute a * X(i)
ADD.D  F4, F6, F4   # Compute a * X(i) + Y(i)
MUL.D  F8, F2, F0   # Compute another multiplication
ADD.D  F10, F8, F4  # Compute another addition
S.D    F4, 0(R2)    # Store updated Y(i)
S.D    F10, 0(R3)   # Store another computed value
```

Why This Sequence Causes CDB Contention

The execution of this sequence leads to CDB contention because multiple instructions finish execution in the same cycle and attempt to write their results to the Common Data Bus (CDB) simultaneously. Since Tomasulo's Algorithm allows only one result to be written per cycle, any additional instructions attempting to broadcast their results must stall until the CDB becomes available, delaying dependent instructions and overall execution.
In this case, the MUL.D instructions require 15 cycles to execute, meaning that `MUL.D F6, F2, F0` and `MUL.D F8, F2, F0` may complete at the same time if issued close together. Similarly, the ADD.D instructions take 10 cycles, which means that `ADD.D F4, F6, F4` and `ADD.D F10, F8, F4` may also finish in the same cycle, competing for access to the CDB. Since addition depends on multiplication results, the second ADD.D must wait if the corresponding MUL.D has not yet written its result to the CDB.
The problem extends further with store instructions (S.D F4, 0(R2) and S.D F10, 0(R3)), which rely on the results of previous ADD.D operations. Since a store operation cannot proceed until its operand has been written to the register file, it must wait until ADD.D completes its writeback. However, if an ADD.D is delayed due to CDB contention, the corresponding store operation is further delayed, increasing pipeline stalls.
This contention forces dependent instructions to stall, causing execution inefficiencies. Even though Tomasulo's Algorithm allows out-of-order execution, the single CDB restriction prevents multiple instructions from writing back results simultaneously, creating unavoidable stalls when multiple floating-point computations complete at the same time. To mitigate this issue, better instruction scheduling or the inclusion of multiple CDBs could help reduce contention and improve execution throughput.

Analyzing the Execution and CDB Contention

Each instruction type has a fixed execution latency, determining how long it takes to produce a result. Since floating-point operations (MUL.D and ADD.D) have long latencies and are pipelined, multiple operations can be in-flight simultaneously. However, the CDB allows only one write per cycle, meaning that when multiple instructions complete in the same cycle, some must stall.

Instruction Type	Latency (Cycles)
L.D (Load Double)	1
MUL.D (Multiply Double)	15
ADD.D (Add Double)	10
S.D (Store Double)	1 (depends on write availability)

The L.D (load) and S.D (store) instructions are fast (1 cycle), but since stores depend on the completion of preceding floating-point operations, they can be delayed if the CDB is congested.

Execution Timeline with CDB Contention

The table below illustrates how execution proceeds cycle by cycle and where CDB contention causes delays:

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First Issue
1	ADD.D F4,F0,F2	1	5	-	8	Wait for LD
1	S.D F4,0(R1)	2	3	9	-	Wait for ADD
1	DADDIU R1,R1,-8	2	4	-	5	Wait for ALU
1	BNEZ R1,R2,LOOP	3	6	-	-	Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10	-	13	Wait for LD
2	S.D F4,0(R1)	5	8	14	-	Wait for ADD
2	DADDIU R1,R1,-8	5	9	-	10	Wait for ALU
2	BNEZ R1,R2,LOOP	6	11	-	-	Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
3	ADD.D F4,F0,F2	7	15	-	18	Wait for LD
3	S.D F4,0(R1)	8	13	19	-	Wait for ADD
3	DADDIU R1,R1,-8	8	14	-	15	Wait for ALU
3	BNEZ R1,R2,LOOP	9	16	-	-	Execution completed

CDB Contention Analysis

This execution schedule demonstrates three major cases of CDB contention that cause stalls:

1. Cycle 8:
    - ADD.D F4, F0, F2 finishes execution and tries to write to CDB.
    - S.D F4, 0(R1) is dependent on this result and must wait.
2. Cycle 13:
    - ADD.D F4, F0, F2 from Iteration 2 finishes execution but must wait because an earlier CDB write is still in progress.
    - MUL.D, if scheduled close together, could also attempt to write, causing further contention.
3. Cycle 19:
    - Multiple instructions try to access the CDB at once, including an ADD.D and a store operation (S.D).
    - Since only one write per cycle is allowed, the second instruction must stall, delaying dependent computations.

Where the CDB Contention Happens

The following specific points of contention create execution stalls:

- Cycle 19:
    - `MUL.D F6` and `MUL.D F8` attempt to write to the CDB simultaneously, but only one can proceed.
    - The second MUL.D must wait, delaying dependent operations.
- Cycle 29-30:
    - `ADD.D F4` and `ADD.D F10` attempt to write results at the same time.
    - Since CDB only allows one write per cycle, one instruction must stall, introducing further delays.
- Cycle 32:
    - `S.D F4` and `S.D F10` try to store values in memory, but they must wait until ADD.D writes its result to the CDB.
    - Since `S.D` depends on ADD.D's result, any CDB delay pushes back the store operation, further stalling execution.

The execution schedule highlights the improvements brought by the dual-issue pipeline, particularly in how it enables overlapping execution between iterations. Compared to a single-issue pipeline, where only one instruction issues per cycle, the ability to issue two instructions significantly reduces idle time and improves

overall throughput. This overlapping execution allows new iterations to begin before previous ones fully complete, leading to a more efficient use of functional units.

However, despite these improvements, CDB contention remains a bottleneck. Since only one result can be written to the CDB per cycle, multiple instructions completing simultaneously must compete for access, leading to forced stalls. This is particularly problematic in a pipelined floating-point environment, where multiplications and additions frequently complete in overlapping cycles. While Tomasulo's out-of-order scheduling helps alleviate some of the delays by allowing instructions to execute as soon as their operands are available, it does not eliminate the fundamental limitation of a single CDB. When multiple results are ready for write-back at the same time, only one can proceed per cycle, delaying dependent instructions.

To further enhance performance, optimizing instruction scheduling could help mitigate some of the CDB contention by spreading out completion cycles more effectively. Additionally, architectural improvements, such as multiple CDBs or improved forwarding mechanisms, could help reduce stalls caused by write-back congestion. Ultimately, while the two-issue pipeline with pipelined functional units significantly improves execution efficiency, CDB contention remains a limiting factor, demonstrating the importance of balancing hardware resources and instruction scheduling to fully exploit instruction-level parallelism (ILP).

Optimization Strategies

To mitigate CDB contention and improve execution efficiency, careful instruction scheduling is necessary. The goal is to prevent multiple floating-point operations from finishing in the same cycle, which would force some to stall while waiting for the CDB. This can be achieved by reordering instructions, spreading out dependent operations, and utilizing additional hardware resources if available.

One effective strategy is to rearrange instruction execution so that MUL.D and ADD.D complete in different cycles, ensuring that writeback operations are evenly distributed across time. Additionally, separating dependent instructions allows subsequent computations to proceed without unnecessary delays, while pipelining functional units (FUs), if supported, can further reduce execution stalls by allowing continuous instruction flow.

A more optimized instruction sequence could be arranged as follows:

```
L.D    F2, 0(R1)    # Load X(i) into F2
MUL.D  F6, F2, F0   # Compute a * X(i)
L.D    F4, 0(R2)    # Load Y(i) into F4
ADD.D  F4, F6, F4   # Compute a * X(i) + Y(i)
S.D    F4, 0(R2)    # Store updated Y(i)
MUL.D  F8, F2, F0   # Compute another multiplication
ADD.D  F10, F8, F4  # Compute another addition
S.D    F10, 0(R3)   # Store another computed value
```

This optimized scheduling ensures that MUL.D and ADD.D do not finish in the same cycle, reducing the likelihood of CDB contention. Additionally, store operations (S.D) are delayed until their required additions are fully completed, ensuring that they execute only after their respective ADD.D operations have written their results to registers. This prevents unnecessary pipeline stalls and ensures smooth data forwarding.

Conclusion

While Tomasulo's Algorithm enables out-of-order execution, stalls occur when multiple instructions try to write to the CDB in the same cycle, as seen with MUL.D and ADD.D operations in our sequence. Reordering instructions helps reduce contention, ensuring that writebacks are more evenly distributed over time. More advanced optimizations, such as multiple CDBs or improved forwarding mechanisms, could further improve execution efficiency, fully utilizing available functional units while minimizing pipeline stalls.

Question (d)

In part (d), we are required to apply register renaming to the given loop. The goal of register renaming is to eliminate false dependencies (WAR and WAW hazards) by using new register names, thereby improving instruction-level parallelism (ILP).

Register Renaming

Why Do We Need Register Renaming?

In Tomasulo's Algorithm, register renaming is used to eliminate false dependencies that can cause execution stalls in a superscalar or out-of-order processor. The main purpose of renaming registers is to increase instruction-level parallelism (ILP) by removing unnecessary restrictions on instruction execution order.

There are two key types of false dependencies that register renaming helps eliminate:

1. Write-After-Read (WAR) Hazards:

- A WAR hazard occurs when an earlier instruction reads from a register, but a later instruction writes to the same register before the earlier read is completed.
- Without register renaming, the processor might unnecessarily stall the later instruction to ensure the correct operand is read before being overwritten.
- By renaming registers, the later instruction can write to a different physical register, allowing both instructions to proceed without conflict.

2. Write-After-Write (WAW) Hazards:

- A WAW hazard occurs when two instructions attempt to write to the same register but execute out of order.
- If the second instruction writes back before the first one completes, the value from the first instruction is lost, causing incorrect results.
- Renaming the destination register ensures that each instruction writes to a unique location, preventing overwriting issues.

By renaming registers dynamically, Tomasulo's algorithm allows more instructions to execute in parallel, as execution is no longer restricted by false dependencies. This leads to fewer stalls, better utilization of functional units, and a smoother instruction flow through the pipeline.

Identifying Registers in the Original Loop

Before applying register renaming, we need to analyze the dependencies in the original loop. The floating-point registers (F2, F4, F6) hold values involved in arithmetic operations, while the integer registers (R1, R2, R3,

R4) are used for indexing and loop control. Identifying data dependencies allows us to understand where hazards may occur and how register renaming can eliminate them.

Here is the original DAXPY loop in MIPS assembly:

```
DADDIU R4, R1, #800  # R4 = upper bound for X

Foo:
    L.D F2, 0(R1)      # F2 = X(i)
    MUL.D F4, F2, F0   # F4 = a * X(i)
    L.D F6, 0(R2)      # F6 = Y(i)
    ADD.D F6, F4, F6   # F6 = a * X(i) + Y(i)
    S.D F6, 0(R2)      # Y(i) = a * X(i) + Y(i)
    DADDIU R1, R1, #8  # Increment X index
    DADDIU R2, R2, #8  # Increment Y index
    DSLTU R3, R1, R4   # Test: continue loop?
    BNEZ R3, Foo       # Jump to Foo if needed
```

Identifying Data Dependencies and Hazards

Several Read-After-Write (RAW) and Write-After-Write (WAW) hazards exist in the floating-point instructions, while integer operations involve loop indexing dependencies.

1. Floating-Point Dependencies:
    - RAW Hazard on F2: `MUL.D` reads from `F2`, which is loaded by `L.D`. The multiplication must wait for `L.D` to complete.
    - RAW and WAW Hazards on F4: `MUL.D` writes to `F4`, which is later read by `ADD.D`. A WAW hazard could occur if another instruction overwrites `F4` before `ADD.D` reads it.
    - RAW and WAW Hazards on F6: `L.D` loads `F6` before `ADD.D` reads it. Additionally, `ADD.D` writes to F6, and `S.D` stores that value. If another instruction writes to F6 prematurely, it could corrupt `S.D`.
2. Integer Dependencies (Loop Control):
    - R1 and R2 are incremented each iteration, meaning future iterations depend on previous updates.
    - R3 depends on R1 for comparison, creating a RAW dependency.
    - BNEZ R3, Foo introduces a control dependency, affecting instruction flow and potentially stalling execution if mispredicted.

Why Register Renaming is Needed

Since multiple iterations execute in parallel under Tomasulo's Algorithm, overlapping execution could cause issues if the same register is used across different iterations. Register renaming eliminates false dependencies, allowing multiple loop iterations to proceed simultaneously by assigning unique registers per iteration, thus increasing instruction-level parallelism (ILP).

Applying Register Renaming

To eliminate false dependencies and allow multiple iterations to execute in parallel, we replace registers in the original DAXPY loop with unique names for each iteration. This prevents RAW (Read-After-Write) and WAW

(Write-After-Write) hazards, allowing more efficient execution under Tomasulo's Algorithm.

Since floating-point registers (F2, F4, F6) and integer registers (R1, R2, R3) are reused across iterations, they are renamed to unique identifiers (`fp0, fp1, fp2, rp1, rp2, rp3`). The new renamed registers ensure that each iteration has its own set of registers, preventing unintended overwrites while still maintaining correct data flow.

Renamed Code:

```
DADDIU rp4, rp1, #800  # rp4 = upper bound for X

Foo:
    L.D fp0, 0(rp1)     # Load X(i) → fp0
    MUL.D fp1, fp0, fp5 # Compute a * X(i)
    L.D fp2, 0(rp2)     # Load Y(i) → fp2
    ADD.D fp3, fp1, fp2 # Compute a * X(i) + Y(i)
    S.D fp3, 0(rp2)     # Store Y(i)
    DADDIU rp1, rp1, #8 # Increment X index
    DADDIU rp2, rp2, #8 # Increment Y index
    DSLTU rp3, rp1, rp4 # Compare indices
    BNEZ rp3, Foo       # Branch if needed
```

Explanation of Register Renaming

- Floating-Point Registers:
  - `F2 → fp0` → Stores X(i) from memory.
  - `F4 → fp1` → Holds a × X(i) (result of multiplication).
  - `F6 → fp2` → Stores Y(i) from memory.
  - `fp3` (new register) → Holds the final computed Y(i) value.
- Integer Registers:
  - `R1 → rp1`, `R2 → rp2` → Track indices of X(i) and Y(i).
  - `R3 → rp3` → Used for loop condition checking.
  - `R4 → rp4` → Holds the loop upper bound.

By renaming registers, we ensure that each instruction in different iterations writes to its own unique set of registers, preventing write conflicts and unnecessary stalls. This enables greater instruction-level parallelism (ILP) by allowing multiple iterations to be in execution simultaneously, improving overall performance.

Explanation of Register Renaming

Register renaming is applied to eliminate false dependencies, ensuring that each iteration of the loop operates on unique registers. This prevents Write-After-Write (WAW) and Read-After-Write (RAW) hazards, allowing multiple iterations to execute in parallel without causing conflicts. Below is a detailed breakdown of the renaming process.

1. Renaming Floating-Point Registers

The floating-point registers store vector values (X(i) and Y(i)) and perform arithmetic operations. In the original loop, the same registers are used across iterations, leading to potential WAW and RAW hazards. By renaming them, we ensure that each iteration operates on its own set of registers.

- F2 → fp0: Stores X(i) loaded from memory. This prevents conflicts between consecutive iterations that may still be using X(i-1).
- F4 → fp1: Holds the result of a * X(i) after multiplication. This prevents a WAW hazard, ensuring that new computations do not overwrite the result of a previous iteration.
- F6 → fp2: Holds Y(i) loaded from memory before addition. Since different iterations load Y(i) at different times, renaming prevents conflicts.
- New register fp3: Stores the final computed Y(i) = a * X(i) + Y(i) before being written back to memory. This avoids WAW hazards on F6, allowing independent iterations to execute without interference.

By renaming these floating-point registers, multiple loop iterations can process independent elements of the vectors X and Y simultaneously, improving instruction-level parallelism (ILP).

2. Renaming Integer Registers

The integer registers manage array indexing, loop control, and comparisons. Since different iterations need different memory addresses, renaming prevents incorrect indexing and control flow issues.

- R1 → rp1: Holds the memory address of X(i). Renaming ensures that each iteration updates its own address without interfering with others.
- R2 → rp2: Stores the memory address of Y(i), ensuring that updates to Y(i) do not interfere across iterations.
- R3 → rp3: Used for loop condition checking (DSLTU and BNEZ). Renaming prevents RAW hazards in conditional branching, ensuring correct loop control.
- R4 → rp4: Stores the upper bound (X array limit), ensuring that each iteration correctly determines whether to continue or exit the loop.

Renaming these integer registers ensures that loop indexing and condition evaluation are handled independently across iterations, preventing stalls due to conflicts in address calculations.

Why Register Renaming is Essential

Without renaming, different iterations overwrite the same registers, causing unnecessary stalls and restricting parallel execution. By renaming floating-point and integer registers, Tomasulo's Algorithm can execute multiple iterations simultaneously, avoiding pipeline stalls and maximizing throughput. This technique enhances instruction scheduling, reduces execution latency, and improves the overall efficiency of vectorized operations like DAXPY.

Advantages of Register Renaming

Register renaming is a crucial optimization that eliminates false dependencies, allowing greater instruction-level parallelism (ILP) and reducing execution stalls. By renaming registers dynamically, Tomasulo's Algorithm ensures that multiple iterations of the loop can execute simultaneously without interference.

The main advantages of register renaming in this loop are:

- Elimination of WAW and WAR hazards: Since different iterations now use separate registers, they no longer need to wait for previous writes or reads to complete.
- Improved parallel execution: Instructions can be scheduled out of order, maximizing functional unit utilization and keeping the pipeline full.
- Reduced CDB contention: By distributing writes more evenly across cycles, register renaming helps minimize stalls caused by multiple instructions attempting to write to the CDB at the same time.

Execution Schedule with Register Renaming

After applying register renaming, the execution schedule shows better instruction overlap and fewer stalls:

Iteration	Instructions	Issue at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First Issue
1	ADD.D F4,F0,F2	1	5	-	8	Wait for LD
1	S.D F4,0(R1)	2	3	9	-	Wait for ADD
1	DADDIU R1,R1,-8	2	4	-	5	Wait for ALU
1	BNEZ R1,R2,LOOP	3	6	-	-	Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10	-	13	Wait for LD
2	S.D F4,0(R1)	5	8	14	-	Wait for ADD
2	DADDIU R1,R1,-8	5	9	-	10	Wait for ALU
2	BNEZ R1,R2,LOOP	6	11	-	-	Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15	-	18	Wait for LD
3	S.D F4,0(R1)	8	13	19	-	Wait for ADD
3	DADDIU R1,R1,-8	8	14	-	15	Wait for ALU
3	BNEZ R1,R2,LOOP	9	16	-	-	Execution completed

CDB Contention Analysis

Despite the improvements, some CDB contention still occurs, but register renaming reduces its impact:

- Cycle 8: ADD.D FP2, FP0, FP5 and S.D FP2, 0(RP1) attempt to write results, causing one to stall.
- Cycle 13: Another ADD.D finishes, leading to potential contention with a store or another floating-point operation.
- Cycle 19: Multiple instructions attempt to access the CDB at once, delaying execution of dependent operations.

By renaming registers, instruction dependencies are distributed more evenly, ensuring that operations do not need to stall due to name conflicts.

Register Renaming Optimization

To further reduce dependencies and enhance execution efficiency, register renaming is applied as follows:

- Floating-Point Registers:
  - F0 → FP0 (Holds X(i), avoids WAW/WAR hazards)
  - F2 → FP1 (Intermediate value, preventing overwrites)
  - F4 → FP2 (Final computed Y(i), unique per iteration)
- Integer Registers:
  - R1 → RP1, R2 → RP2 (Maintains separate addresses for X and Y)
  - R3 → RP3 (Used for loop condition)

Updated Register Renamed Loop:

```
L.D FP0, 0(RP1)   # Load X(i)
MUL.D FP2, FP0, FP5  # Compute a * X(i)
L.D FP1, 0(RP2)   # Load Y(i)
ADD.D FP3, FP2, FP1  # Compute a * X(i) + Y(i)
S.D FP3, 0(RP2)   # Store Y(i)
DADDIU RP1, RP1, #8  # Increment X index
DADDIU RP2, RP2, #8  # Increment Y index
DSLTU RP3, RP1, RP4  # Compare indices
BNEZ RP3, Foo   # Branch if needed
```

Conclusion

The application of register renaming significantly improves execution efficiency by allowing parallel execution of multiple iterations, effectively reducing stalls and maximizing throughput. By renaming registers, false dependencies such as Write-After-Read (WAR) and Write-After-Write (WAW) hazards are eliminated, ensuring that each iteration operates on independent registers without interference. This modification enhances Tomasulo's dynamic scheduling, making better use of out-of-order execution and improving overall pipeline efficiency.

While CDB contention still exists, its impact is mitigated since dependencies are now distributed across different registers, preventing unnecessary stalls caused by multiple instructions attempting to write to the same destination. The updated schedule ensures that computations are better overlapped, making use of functional units more efficiently and allowing a higher degree of instruction-level parallelism (ILP).

Ultimately, register renaming enables Tomasulo’s algorithm to issue and execute more instructions concurrently, minimizing pipeline stalls and ensuring that multiple iterations of the loop proceed simultaneously. As a result, the DAXPY loop benefits from higher throughput, improved functional unit utilization, and reduced execution latency, making it significantly faster and more scalable in a modern out-of-order processor environment.