# Problem M16.1: Transactional Memory (Spring 2015 Quiz 4, Part B)

Ben Bitdiddle wants to implement a transactional memory system with pessimistic conflict detection in a two-core processor. This system has the following characteristics:

- When a transaction starts, it is assigned a unique global timestamp.
- The memory system tracks the set of addresses read or written by each transaction (i.e., its **read set** and **write set**).
- For every transactional load, the memory system checks whether this load reads an address in the **write set** of any other transaction, and declares a conflict if so.
- For every transactional store, the memory system checks whether this store writes an address in the **read set or write set** of any other transaction, and declares a conflict if so.
- On a conflict, the transaction with the later timestamp aborts.
- An aborted transaction restarts execution 10 cycles later.

Ben runs a program with two types of transaction: X and Y, shown below.

| Cycle relative to start | Transaction X |
|---|---|
| Cycle 0 | Starts |
| Cycle 10 | Read B |
| Cycle 20 | Read A |
| Cycle 30 | Write A |
| Cycle 40 | Ends |

| Cycle relative to start | Transaction Y |
|---|---|
| Cycle 0 | Starts |
| Cycle 10 | Read B |
| Cycle 20 | Read A |
| Cycle 30 | Read B |
| Cycle 40 | Ends |

## Problem M16.1.A

Suppose the system is executing two transactions: a type X transaction that starts at cycle 0 and receives timestamp 0, and a type Y transaction that starts at cycle 5 and receives timestamp 5. Is there a conflict between these two transactions? If so, at what cycle does this conflict happen?

**Problem M16.1.B**

Ben implements conflict detection by extending a conventional MSI coherence protocol. Furthermore, drawing inspiration from the delay invalidation cache coherence protocol in Quiz 3, Ben wants to optimize his transactional memory system as follows:

- When a core receives an abort for its currently running transaction, it delays the abort until the next local cache miss. If the transaction finishes without additional misses, it will commit successfully.

With this optimization, assume the same scenario as in the previous question: a type X transaction that starts at cycle 0 and receives timestamp 0, and a type Y transaction that starts at cycle 5 and receives timestamp 5. Are any of these transactions aborted? If so, when do aborts happen?

Does this optimization always provide correct transactional semantics? Explain your answer in one or two sentences.

**Problem M16.1.C**

Ben believes this optimization works well and always needs fewer cycles to complete transactions. Is he correct? If so, explain why this always improves performance with one or two sentences. Otherwise, provide an example where this optimization causes a transaction to finish later.

## Problem M16.2: Transactional Memory (Spring 2016 Quiz 4, Part D)

You are designing a hardware transactional memory (HTM) system that uses pessimistic concurrency control (i.e., on each load/store, the HTM checks for conflicting accesses to the same address made by other transactions). Comment on whether the following conflict resolution policies suffer from either livelock (i.e., the system may reach a state where *no single transaction* makes forward progress) or starvation (i.e., the system may reach a state where *at least one transaction* does not make forward progress). State your reasoning.

1. **Requester wins**: Upon a conflict, the transaction whose request initiated the conflict check is granted access to the data, and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

2. **Timestamp-based, retain timestamp on abort**: Each transaction is assigned a unique timestamp when it first begins execution. Timestamps are monotonically increasing. Upon a conflict, if the requesting transaction's timestamp is lower than the timestamps of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original timestamp when they restart execution.

3. **Timestamp-based, discard timestamp on abort**: Like the previous policy, except that aborted transactions discard their previous timestamp and acquire a new one when they restart execution.

4. **Random-number-based, retain random number on abort**: Each transaction is assigned a unique random number when it first begins execution. Upon a conflict, if the requesting transaction's random number is lower than the random numbers of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original random number when they restart execution.

5. **Random-number-based, discard random number on abort**: Like the previous policy, except that aborted transactions discard their previous random number and acquire a new one when they restart execution.

## Problem M16.3: Transactional Memory (Spring 2020 Quiz 4, Part C)

Ben Bitdiddle is designing a hardware transactional memory (HTM) system. He is concerned about three potential issues arising in his system:

1. *Deadlock*: Some transactions stay stalled indefinitely on a cyclic waiting pattern, so they neither commit nor abort.

2. *Livelock*: Some transactions can execute, but no transaction ever commits (e.g., due to repetitive aborts and re-execution). Thus, the system does not make forward progress.

3. *Starvation*: Some transactions can commit, but at least one other transaction is prevented from committing indefinitely. Thus, one or a subset of transactions does not make forward progress.

Ben wants to classify each of the 4 HTM systems in Questions 1 to 4 as one of four types, according to the forward progress guarantees they provide:

**A.** May deadlock
**B.** May livelock, but cannot deadlock
**C.** May starve, but cannot deadlock or livelock
**D.** Cannot deadlock, livelock, or starve

For **Questions 1 to 4, write down the letter A, B, C, or D and explain your choice**. You can either explain intuitively why an issue cannot arise, or use an example to show that the system suffers from the issue.

When you choose a particular option, you only need to explain the issues it differentiates between. For example, if you choose B, you should explain why the system cannot deadlock, and describe an example of how it may livelock.

## Problem M16.3.A

**HTM 1**: Optimistic conflict detection, lazy versioning, and *committer-wins* resolution policy. Assume there is enough capacity for versioning (so transactions do not overflow speculative buffers, e.g., the L1 cache).

The *committer-wins* resolution policy works as follows. Upon a conflict, the committing transaction wins and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

## Problem M16.3.B

**HTM 2**: Pessimistic conflict detection, lazy versioning, and *requester-wins* resolution policy. Assume there is enough capacity for versioning.

The *requester-wins* resolution policy works as follows. Upon a conflict, the transaction that triggers the detection (the requester) wins and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

**Problem M16.3.C**

**HTM 3**: Pessimistic conflict detection, eager versioning, and *requester-stalls* resolution policy. Assume there is enough capacity for versioning.

The *requester-stalls* resolution policy works as follows. Upon a conflict, the transaction that triggers the detection (the requester) stalls until the conflicting transactions abort or commit. After aborting, transactions immediately restart execution.

**Problem M16.3.D**

**HTM 4**: Pessimistic conflict detection, lazy versioning, and *oldest-wins* resolution policy. Assume there is enough capacity for versioning.

The *oldest-wins* resolution policy works as follows. Each transaction is assigned a unique, monotonically increasing timestamp when it first begins execution. Upon a conflict, if the requesting transaction's timestamp is lower than the timestamps of all other conflicting transactions, the requesting transaction commits and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted. After aborting, transactions immediately restart execution. Aborted transactions retain their original timestamp when they restart execution.

**Problem M16.3.E**

Consider HTM 1, HTM 2, HTM 3, and HTM 4.
Which HTM(s) may suffer from serialization bottlenecks when running only non-conflicting transactions concurrently? Point out such bottlenecks for each HTM design.

**Problem M16.3.F**

Consider an HTM system. Suppose **N** identical transactions are running simultaneously on N cores. All the transactions read and write to the same memory location, causing conflicts between any pair of them.

Does the HTM design guarantee that all the transactions can commit in the end? If so, what is the maximum number of aborts?

Answer the questions above for each of HTM 1, HTM 2, and HTM 3, respectively