Hardware Transactional Memory

- transactions normally associated with databases
- in this context, think of a transaction as the atomic update of a number of memory locations [eg. atomic update of a data structure]
- a transaction is a finite sequence of machine instructions that read and write memory locations, executed by a single thread, satisfying the following properties:
  - serializability: transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another
  - committed transactions are never observed by different threads to execute in different orders
  - atomicity: each transaction makes a sequence of tentative changes [NOT visible to other threads] to memory and the architectural state [CPU registers] and then either
    - COMMITS - making its tentative changes visible to other threads
    - ABORTS - causing its tentative changes to be discarded

Hardware Transactional Memory

- *Transactional Memory: Architectural Support of Lock-Free Data Structures*
  Maurice Herlihy and J. Eliot B. Moss
  Proceedings of the 20th Annual International Symposium on Computer Architecture
  1993

- motivations
  - lock free operations on a data structure will not be prevented if other threads stall mid execution
  - avoids common problems with mutual exclusion
  - out performs best known locking techniques

- takes advantage of the first level cache and the cache coherency protocol

- tentative changes made to the first level cache [and architectural state] ONLY

- tentative changes made visible atomically on a successful commit

Hardware Transactional Memory

- typical transactional code

start transaction	start transaction
< UPDATE SHARED DATA STRUCTURE >	*< UPDATE SHARED DATA STRUCTURE >*
commit transaction	commit transaction
retry on failure	retry on failure

- will describe Intel Transactional Synchronization eXtension [TSX]
- implemented 20 years after original Herlihy and Moss paper
- support for hardware lock elision [HLE] and restricted transactional memory [RTM]
- first Haswell CPU with TSX released Jun-13 [Aug-14 bug reported in first implementation]
- NOT all later CPUs support TSX [need to test CPUID.07H.EBX.RTM [bit 11] = 1]

Intel TSX

- 4 *new* assembly language instructions for RTM
  - xbegin transaction begin
  - xend transaction end
  - xabort transaction abort
  - xtest test if in a transaction
- example transactional code [IA32/x64 assembly language]

```
    xbegin L0
    < INSTRUCTIONS TO UPDATE SHARED DATA STRUCTURE >
    xend
    < HERE ON SUCCESSFUL COMMIT >

L0: < HERE ON ABORT > [eax contains RTM abort status]
```

- eager conflict detection
- transaction fails as soon as a conflict is detected

Intel RTM

- why does a transaction abort?
- instructions inside a transaction read and write memory locations
- transaction read set and write set
- transaction will abort if any other CPU...
  - reads a location in its write set
  - writes to a location in its read or write set
- transactions may also abort due to hardware limitations, context switches, interrupts, page faults, update of PTE Accessed and Dirty bits, ...
- MUST provide a non transactional execution path that can be executed if a transaction fails *continuously*

Intel RTM...

- RTM abort status in eax

eax bit	
0	set if abort caused by XABORT instruction
1	transaction may succeed on retry [always clear if bit 0 set]
2	set if another logical processor conflicts with read or write set
3	set if internal buffer overflowed
4	set if debug breakpoint was hit
5	set if abort occurred during a nested transaction
6:23	reserved
24:31	ABORT argument [only valid if bit 0 set]

- NB: an aborted transaction can return 0 in eax [NO bits set]

TSX Instrinsics

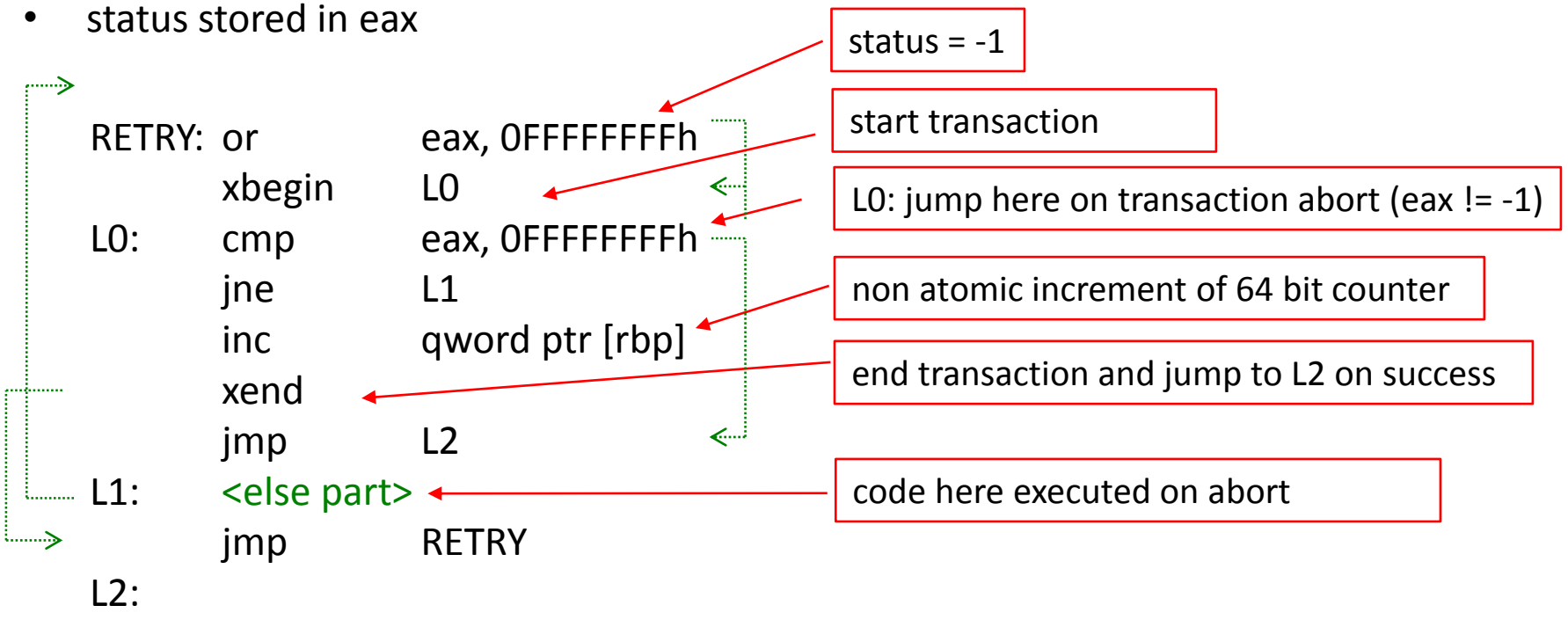
- _xbegin() and _xend() intrinsics
- not so easy to follow without examining generated code
- consider following code to increment a shared global variable *g* using a transaction

```
while (1) {                                 // keep trying
   int status = _xbegin();                  // set status = -1 and start transaction
   if (status == _XBEGIN_STARTED) {         // status == XBEGIN_STARTED == -1
      (*g)++;                               // non atomic increment of shared global variable
      _xend();                              // end transaction
      break;                                // break on success
   } else {                                 //
      …                                     // code here executed if transaction aborts
   }                                        //
}
```

- no code provide here for non transactional path, BUT it is required
- non transactional path could update data structure using a lock (hopefully a rare event)

TSX Instrinsics...

- examine generated code using debugger
- NB: can't debug [single step] code in the body of transaction
- status stored in eax



- NB: works because if transaction aborts, eax will not be -1

TSX Instrinsics...

- void _xabort(const unsigned int imm)

  forces transaction to abort
  the low 8 bits of imm will be returned in bits 24:31 of RTM abort status

- unsigned char _xtest(void)

  returns 1 if currently executing a transaction, otherwise 0

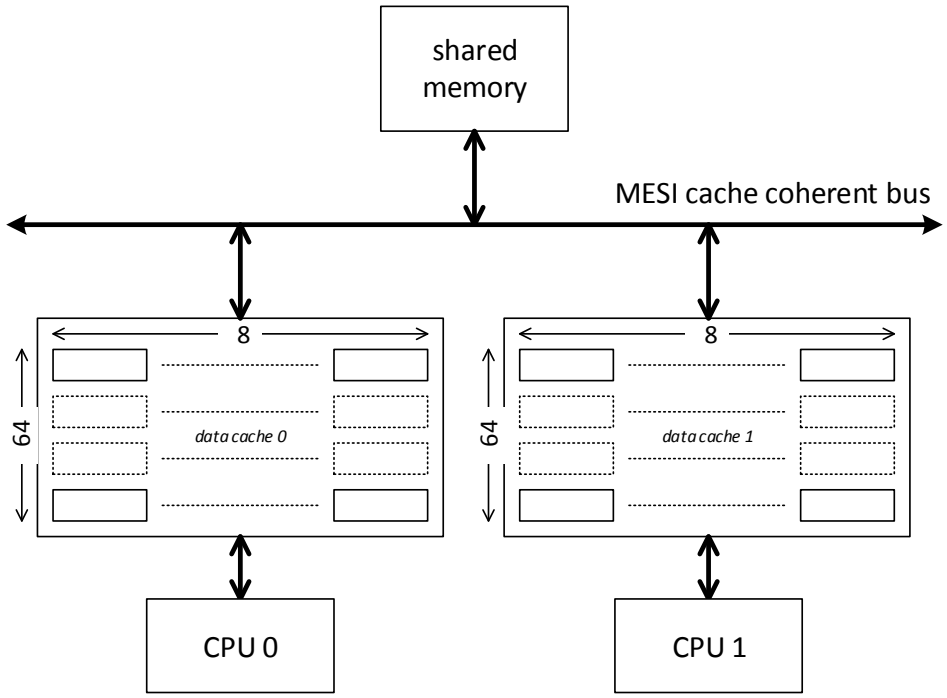
- transactions can be nested up to an implementation limit [MAX_RTM_NEST_COUNT]

```
xbegin L0       // nesting count 1
xbegin L1       // nesting count 2 [L1 ignored if nesting count != 1]
xend            // nesting count 1
xend            // nesting count 0
```

- transaction only committed if nesting count is 0 [partial support of nested transactions]

TSX Level 1 Cache Support

- Haswell level 1 data cache
- 32K L=64 K=8 N=64
- 512 cache lines [8 x 64]
- each hyper-threaded CPU has its own L1 cache
- MESI cache coherency



- cache line states Modified, Exclusive, Shared and Invalid
- additional T bit which is set if cache line is part of a transaction

TSX Level 1 Cache Support...

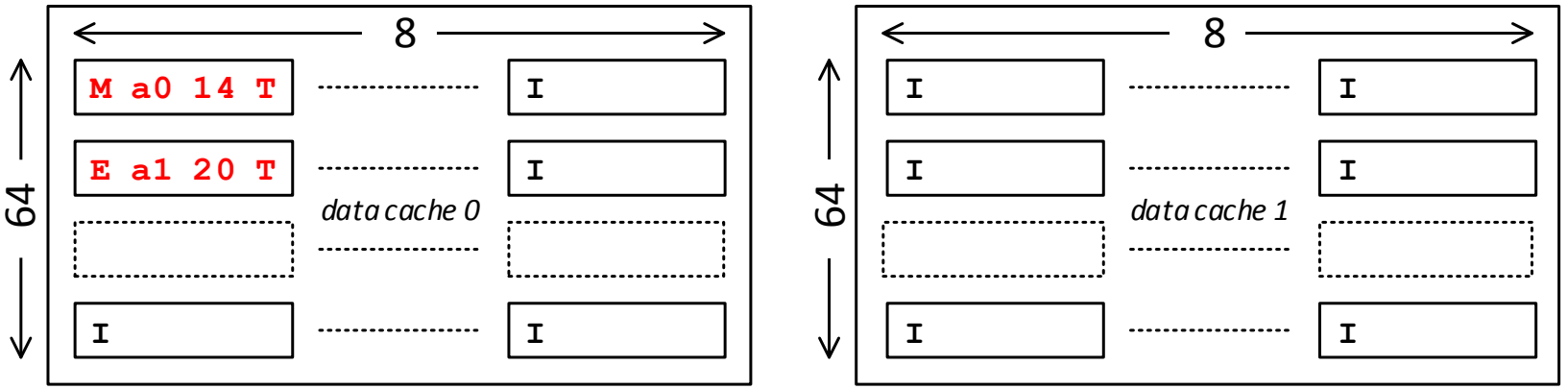
- consider the following transaction [assume initially a0 = 10 and a1 = 20]

```
xbegin
    a0 += 4;      // add 4
    a1 -= 4;      // subtract 4
xend
```

- simulates atomically transferring €4 from one bank account to another
- transaction involves two memory locations a0 and a1
- transactions can be executed concurrently [will abort if a conflict detected]
- assume *address of a0* maps to level 1 data cache set 0 and *a1* to set 1
- assume ALL cache lines initially Invalid with T = 0

TSX Level 1 Cache Support...

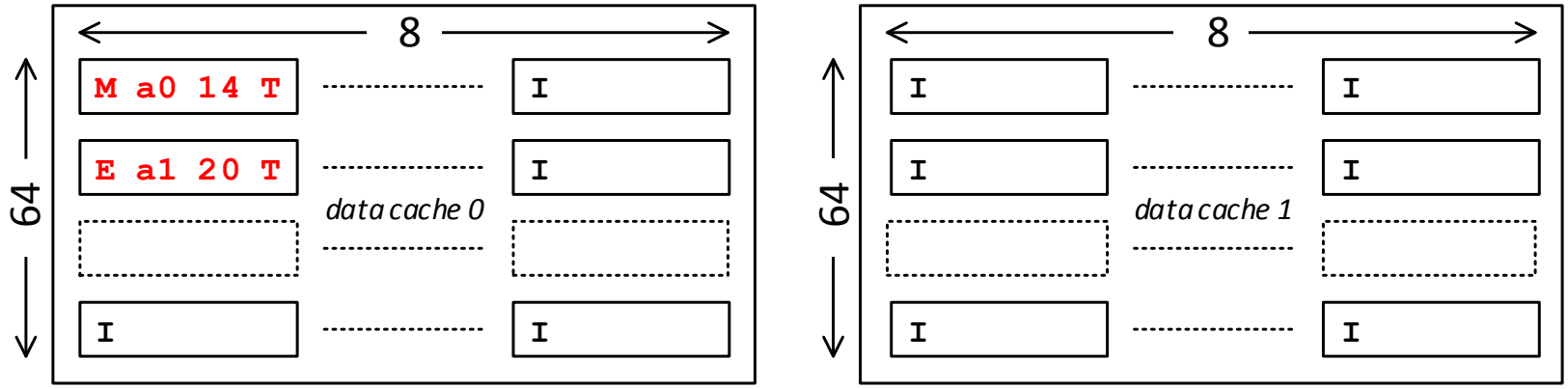
- CPU0 starts transaction
- CPU0 reads a0 into cache [Exclusive] and sets T bit [a0 added to transaction read set]
- CPU0 writes a0 [a0 += 4] in cache ONLY [Modified] [a0 added to transaction write set]
- CPU0 reads a1 into cache [Exclusive] and sets T bit [a1 added to transaction read set]



- CPU0 writes a1 [a1 -= 4] in cache ONLY [Modified] [a1 added to transaction write set]
- xend executed and...
- transaction commits by clearing T bits [*instantaneously*] [and its read and write sets]
- modified cache lines are now visible and accessible [Modified]

TSX Level 1 Cache Support...

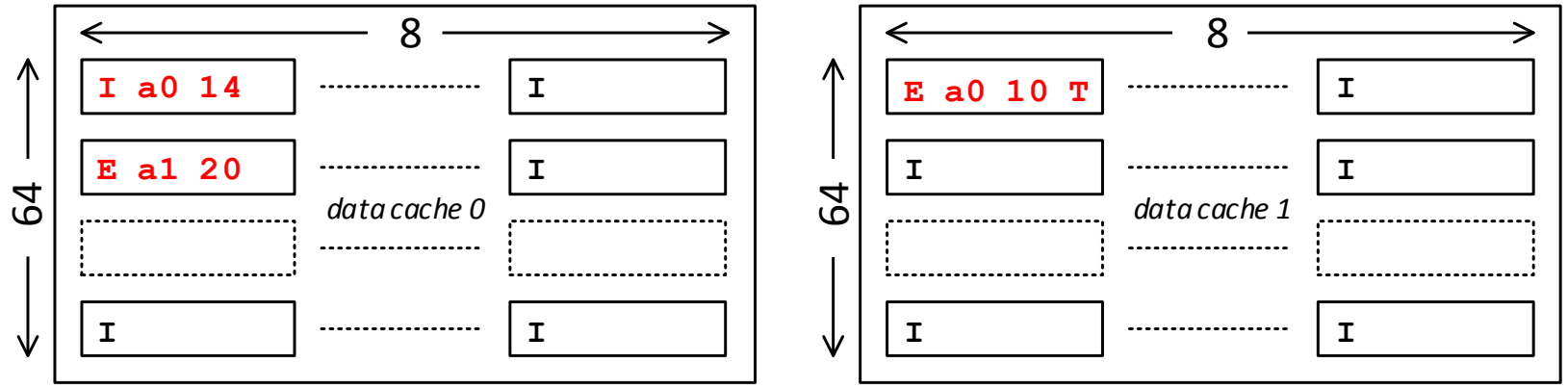
- imagine CPU0 and CPU1 execute the transaction concurrently and that...
- CPU0 is *ahead of* CPU1 and is about to write to a1 when CPU1 starts its transaction
- assume that the data caches are in the following state



- CPU1 tries to read a0 into its cache
- CPU0 detects a conflict because CPU1 is attempting to read a Modified cache line that is part of its transaction [T and Modified bits set meaning that a0 is a member of CPU0's write set]

TSX Level 1 Cache Support...

- CPU0 aborts transaction by invalidating all Modified cache lines involved in the transaction [those marked with a T bit] and clearing all T bits
- CPU1 will read a0 directly from memory [CPU0 will NOT intervene to supply data]]



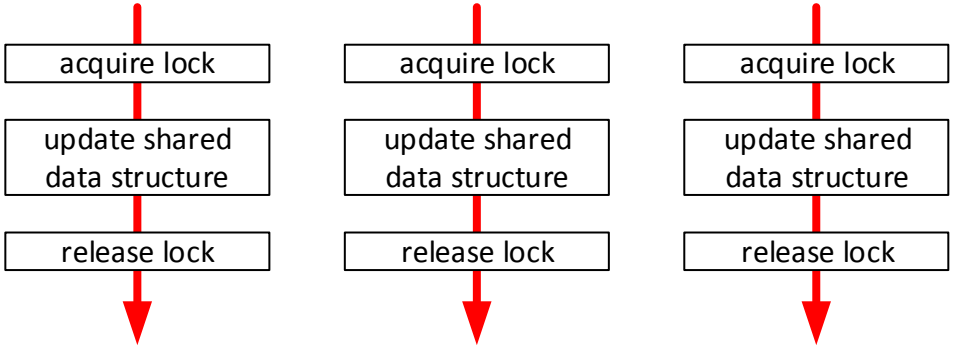
- even though CPU0 had nearly completed its transaction, it is CPU0 that aborts
- CPU0 would also abort if CPU1 reads a0 outside of a transaction
- CPU0 detects a conflict if another CPU reads a location in its write set or writes to a location in either its read or write set

TSX Level 1 Cache Support...

- CPU detects conflicts at the granularity of a cache line
- replacement [eviction] of a cache line in the write set causes a transaction to abort
- replacement [eviction] of cache lines in the read set are tracked by unspecified implementation specific hardware and may not cause an abort [victim cache??]
- since the Haswell level 1 cache is 8 way, a transaction that writes to 9 locations which map to the same set will always abort
- remember that a hyper-threaded CPU share the first level cache [thus reducing the effective size of a thread's read and write set]
- how exactly is a modified cache line, which subsequently becomes part of a transaction, handled? must write to memory before being overwritten as part of a transaction so original value can read from memory if transaction aborts

Hardware Lock Elision [HLE]

- makes use of transactional memory to speculatively update a shared data structure that is normally protected by a lock



- can be easily retro-fitted to existing code base [by modifying lock code]
- instead of acquiring lock, update shared data structure speculatively
- use transactional memory to detect conflicts
- if conflict detected, re-execute by acquiring lock for real

Hardware Lock Elision...

- two *new* TSX instructions needed to support HLE

  xacquire – used as a prefix to the instruction acquiring lock
  xrelease – used as a prefix to the instruction releasing lock

- xacquire must precede XCHG or a LOCK prefix
- xrelease must precede XCHG, a LOCK prefix, MOV *mem, reg* or MOV *mem, imm*
- xacquire and xrelease are treated as NOPs on CPUs which do not support TSX

- how does HLE work?

- IF XACQUIRE EXECUTED NORMALLY IT WILL ELIDE (ALTER) THE FOLLOWING INSTRUCTION

- IF XACQUIRE EXECUTED AS THE RESULT OF A TRANSACTION ABORT IT IS IGNORED

Hardware Lock Elision...

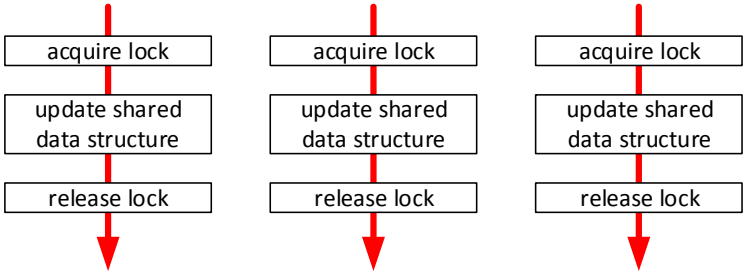
- normal execution of xacquire starts a transaction
- following instruction executed with elision [normally an instruction to obtain lock]
- writing of lock not visible externally [has side effect of reducing bus traffic]
- address of lock, its original value and new value saved in an internal elision buffer
- address of lock added to the transaction readset
- other CPUs will continue to read the lock as being free [unless they have also obtained the lock with elision], but this CPU will see the lock as taken [reads new value from elision buffer]

- if NO conflicts detected while updating the shared data structure…

- xrelease commits the transaction and ALL changes become visible *instantaneously*
- instruction following xrelease will not write to the lock if it is going to overwrite it with its original value [original value saved in elision buffer] also reducing bus traffic
- other CPUs will NOT observe the write and hence their transactions will NOT abort

Hardware Lock Elision...

- if multiple threads obtain the lock by elision and then do not interfere with each other, updates to the shared data structure can occur in parallel



- without HLE there is NO parallelism

- if a conflict is detected, the transaction aborts and the xacquire instruction is re-executed, but ignored, resulting in the following lock instruction also being executed normally [without elision]

- writing to the lock without elision results in conflicting transactions being aborted as it writes to the readset of conflicting transactions

Hardware Lock Elision...

- Sample assembly language code

jump here on transaction abort, but xacquire is ignored

```
        mov     eax, 1          ; eax = 1
retry:  xacquire                ; xacquire prefix hint
        xchg    eax, lock       ; exchange eax and lock in memory
        test    eax, eax        ; test eax if lock free [0] …
        jz      locked          ; jmp to locked otherwise…
wait:   pause                   ; causes transaction to abort
        cmp     lock, #1        ; should get here outside of a transaction
        je      wait            ; wait until lock free
        jmp     retry           ; retry using HLE

locked: < UPDATE SHARED DATA STRUCTURE>

        xrelease                ; xrelease prefix hint
        mov     lock, 0         ; clear lock
        ret                     ; return
```

Hardware Lock Elision...

- what is the function of the pause instruction?
- if the lock is already set when executing the lock instruction with elision, the thread will spin waiting for the lock to become free **INSIDE A TRANSACTION**
- when the lock is freed by the thread holding the lock [written with 0], the waiting threads will abort and then try to obtain the lock without elision
- NOT good as thread might have been able to update the shared data structure transactionally by obtaining lock with elision
- obtaining lock without elision inhibits parallelism
- can easily get into a state where the lock is always obtained without elision unless there is a break when no threads are trying to obtain the lock

Hardware Lock Elision...

- the pause instruction causes transaction to abort
- the instruction to obtain the lock is then re-executed without elision and if the lock is still taken the “do while” loop will be executed non transactionally
- when the lock is freed, an attempt is made to obtain lock with elision
- approach reduces the number of times lock taken without elision
- what happens if locked freed before pause executed? there is a race, the consequence of which is that the lock will be obtained without elision
- Tutorial 3 will help determine the effectiveness of HLE locks

HLE Instrinsics

- Microsoft VC++ HLE intrinsics

```
long _InterlockedExchange_HLEAcquire(long* addr, long v)  // acquire lock using HLE

void _Store_HLERelease(long *addr, long v);               // release lock using HLE
```

- equivalent Microsoft VC++ for previous assembly language code

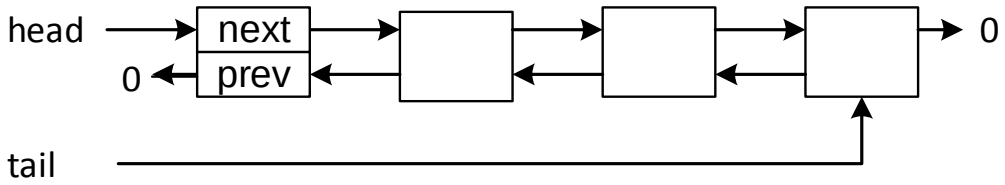
```
while (_InterlockedExchange_HLEAcquire(&lock, 1)) {
    do {
        _mm_pause();                          // aborts transaction
    } while (lock == 1);
}

< UPDATE SHARED DATA STRUCTURE >

_Store_HLERelease(&lock, 0);                  // release lock
```

Transaction Code Example

- doubly linked list
- head and tail



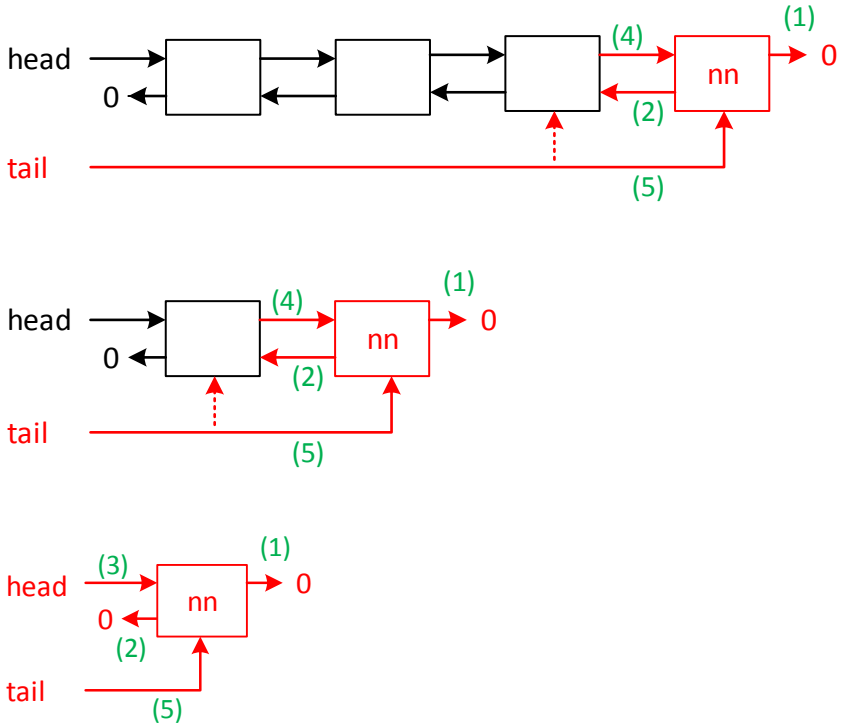
- operations to add and remove an item from head or tail
- when list NOT empty, operation modifies head or tail, but NOT both
- when list empty, operation modifies head and tail

- difficult to extract parallelism using locks
- protecting list with a single lock means that concurrent operations at either end of list are NOT possible

- extracting concurrency straightforward with transactions

Transaction Code Example...

```
void DLList::addTail(Node *nn) {
    xbegin();
    nn->next = NULL;          // (1)
    nn->prev = tail;          // (2)
    if (tail == NULL) {
        head = nn;            // (3)
    } else {
        tail->next = nn;      // (4)
    }
    tail = nn;                // (5)
    xend();
}
```



- similar code needed for addHead(), removeHead() and removeTail()
- extracts maximum concurrency

- MUST provide a non transactional path

Implementing Transactional and Non Transactional Paths

- consider the following approach
  - use the same code for transactional and non transactional paths
  - delay between attempts
  - if transaction continues to fail after a given number of attempts, update data structure using a lock
  - need to read lock in transactional path [add lock to readset] so that if any other thread sets the lock, the transaction will be aborted
  - need to consider races between transactional and non transactional paths

Sample Code for Transactional and Non Transactional Paths

```
void DLList::addTail(Node *nn) {

    int state = TRANSACTION;                    // TRANSACTION = 0 LOCK = 1
    int attempt = 1;                            // number of attempts
    while (1) {
        UINT status =_XBEGIN_STARTED;           // initialise status _XBEGIN_STARTED = -1
        if (state == TRANSACTION) {             // if state == TRANSACTION ...
            status = _xbegin();                 // execute transactionally ...
        } else {                                // otherwise ...
            while (InterlockedExchange(&lock, 1)) {   // obtain testAndtestAndSet lock and ...
                do {                            // execute non transactionally
                    _mm_pause();
                } while (lock == 1);
            }
        }
        if (status == _XBEGIN_STARTED) {        // test status
            if (state == TRANSACTION && lock)   // if executing transactionally, add lock to readset so transaction will abort if lock obtained by another thread
                _xabort(0xA0);                  // ALSO abort immediately if lock already set

            < UPDATE SHARED DATA STRUCTURE >

            if (state == TRANSACTION) {         // if executing transactionally ...
                _xend();                        // end transaction ...
            } else {                            // otherwise ...
                lock = 0;                       // release lock
            }
            break;
        } else {                                // HERE on a transaction abort
            if (lock) {                         // if lock set ...
                do {                            // wait until released
                    _mm_pause();
                } while (lock);
            } else {                            // otherwise
                volatile UINT64 wait = attempt << 4;   // initialise wait and delay by ...
                while (wait--);                 // decrementing until zero (NB: initially wait should be > 0)
            }
            if (++attempt >= MAXATTEMPT)        // increment attempt and if greater than or equal MAXATTEMPT
                state = LOCK;                   // execute non transactionally by obtaining lock
        }
    } // while
}
```

- one approach
- back off and number of attempts need to be tuned

Using an HLE Lock

declared as part of DLLIist class

intrinsic

```
void DLList::addTail(Node *nn) {
    while (_InterlockedExchange_HLEAcquire(&lock, 1)) {
        do {
          _mm_pause();          // aborts transaction
        } while (lock == 1);
    }
    nn->next = NULL;
    nn->prev = tail;
    if (tail == NULL) {
        head = nn;
    } else {
        tail->next = nn;
    }
    tail = nn;
    _Store_HLERelease(&lock, 0);
}
```

which is better??
RTM more flexible, probably a poor strategy to give up after only one attempt at trying to make change with a transaction as per HLE

intrinsic

xrelease
mov lock, 0

Learning Outcomes

- you are now able to
  - explain exactly what a transaction is in this context
  - describe the operation of the Intel TSX instruction set
  - explain how the level 1 cache detects conflicts between transactions
  - write lockless algorithms using RTM transactions
  - write lockless algorithms using Hardware Lock Elision (HLE)