Full Name: _____

SUNet Id: _____

I accept the letter and spirit of the honor code:

Signed: _____

Stanford CS149, Fall 2022

Midterm Solutions

Nov 15th, 2022

Instructions:

- Write your answers in the space provided below the problem. If your work gets messy, please clearly indicate your final answer.
- The exam has a maximum score of **100** points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.

Problem	Your Score	Possible Points
1		20
2		16
3		16
4		18
5		14
6		16
Total		100

Warm Up: Miscellaneous Short Problems

Problem 1. (20 points):

A. (4 pts) Consider a multi-core processor that runs at 2 GHz and has 4 cores. Each core can perform up to one 8-wide SIMD vector instruction per clock and supports hardware multi-threading with 4 hardware execution context per core. What is the maximum throughput of the processor in units of scalar floating point operations per second? Please show your calculations.

Solution: 2 billion clock/sec $\times$ 4 cores $\times$ 8 floating point ops per core per clock = 64 billion floating point ops/second. Even though the problem asked for an answer in units of ops/sec, we also accepted answers of 64 ops per clock.

B. (4 pts) Consider the processor from part A running a program that perfectly parallelizes onto many cores, makes perfect usage of SIMD vector processing, and has an arithmetic intensity of 4. (4 scalar floating ops per byte of data transferred from memory.) If we assume the processor has a memory system providing 64 GB/sec of bandwidth, is the program compute-bound or bandwidth bound on this processor? You may assume for math simplicity that 1 GB/sec is a one billion bytes per second (10^9). Please show your work.

Solution: The processor can perform 64 billion operations per second and the program has an arithmetic intensity of 4. Since every four ops the program performs a byte of data transfer, the program requires 16 GB/sec of bandwidth. The system provides significantly more bandwidth than this, so it is compute bound.

C. (4 pts) Consider the following piece of C code.

```
float A[VERY_LARGE];
float B[VERY_LARGE];
float C[VERY_LARGE];
float D[VERY_LARGE];
float E[VERY_LARGE];

for (int i=0; i<VERY_LARGE; i++)
  C[i] = A[i] * B[i];
for (int i=0; i<VERY_LARGE; i++)
  D[i] = C[i] + B[i];
for (int i=0; i<VERY_LARGE; i++)
  E[i] = D[i] - A[i];
```

Assume that `VERY_LARGE` is so large that the arrays are hundreds of MBs in size, and that the code is run on a single-core processor with a 8 MB cache. Please modify the program to maximize its arithmetic intensity. You only need to write to the output array `E`, you don't need to fill in `C` and `D` if it is not necessary. However, please DO NOT CHANGE the number of math operations performed. **If we assume the program before and after the modification is bandwidth bound, how much does your modification improve its performance?**

Solution: the old program had arithmetic intensity one op per three floats of memory traffic. The new program performs three ops per three floats of memory traffic (load from `A` and `B`, store to codeE). We accepted solutions that were in units of ops per byte or ops per float. Note that students needed to see that after the program transformation, the fact that `A[i]` and `B[i]` are accessed twice in the expression to compute `E[i]` does not mean both of those accessed generate memory traffic. The second access would certainly be a cache hit.

```
for (int i=0; i<VERY_LARGE; i++)
  E[i] = (A[i] * B[i]) + B[i] - A[i];
```

D. (4 pts) Consider a Spark program that consists of a sequence of four RDD operations executing on a cluster of 10 computers. If we assume that all RDD operations have **NARROW DEPENDENCIES**, please explain why this computation can be performed by Spark without any communication of RDD data between the machines.

Solution: Spark will partition the input RDD across all 10 machines. Since the operations have narrow dependencies, element i of an RDD can be computed using only element i from its parent RDD. That means that all machines can compute their part of the resulting RDDs using only local data, and so no inter-machine communication is necessary.

To receive full credit, an answer needed to clearly indicate that the computation can be scheduled in a way such that the data needed remains local to a machine. This is achieved by partitioning the RDD.

Note that this enables optimizations such as fusion, but stating that the operations can be fused is insufficient as it misses the why. Similarly, answers that restated the definition of narrow dependency, did not use the definition, or restated the problem information did not receive full credit.

E. (4 pts) Consider a cache that contains 32 KB of data, has a cache line size of 4 bytes, is fully associative (meaning any cache line can go anywhere in the cache), and uses an LRU (least recently used—the line evicted is the line that was last accessed the longest time ago) replacement policy. Please describe why the following code will take a cache miss on every data access to the array `A`.

```
const int SIZE = 1024 * 64;
float A[SIZE];
float sum = 0.0;
for (int reps=0; reps<32; reps++)
  for (int i=0, i<SIZE; i++)
    sum += A[i];
```

Solution: Each iteration of the inner loop iterates over 256 KB of data from `A`. (Note many students incorrectly determined the array was 64 KB, although this still allowed them to get the corect answer.) This means that by the time the code accesses the beginning of `A` in the next iteration of the outer loop, that data has been evicted from the cache. As a result, all accesses to `A` are cache misses. In the first iteration of the loop the misses are "cold misses" (the first time the data is accessed). In all subsequent iterations of the loop they are "capacity misses" because the program has accessed the data below, but the data has fallen out of the cache due to finite cache capacity.

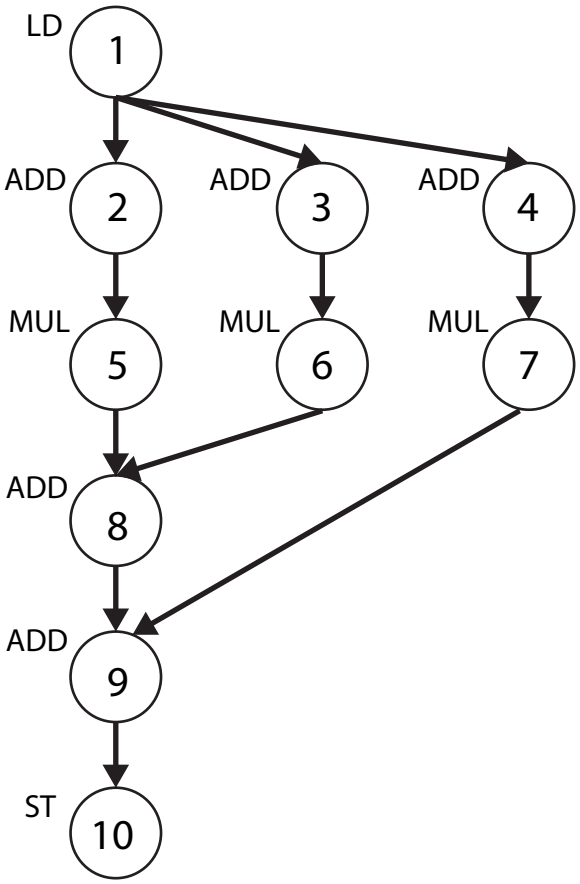
Dependency Graphs, ILP, and Superscalar/Multi-Threaded Execution

Problem 2. (16 points):

Consider the following sequence of scalar instructions running within a single thread.

```
1.  LD   R0 <- mem[R4]    // load memory address given by R4 into R0
2.  ADD  R1 <- R0, R0     // R1 = R0 + R0
3.  ADD  R2 <- R0, R0     // R2 = R0 + R0
4.  ADD  R3 <- R0, R0     // R3 = R0 + R0
5.  MUL  R1 <- R1, R1
6.  MUL  R2 <- R2, R2
7.  MUL  R3 <- R3, R3
8.  ADD  R1 <- R1, R2
9.  ADD  R1 <- R1, R3
10. ST   mem[R5] <- R1    // store R1 into memory at address given by R5
```

A. (4 pts) Please draw the instruction dependency graph for this instruction sequence.

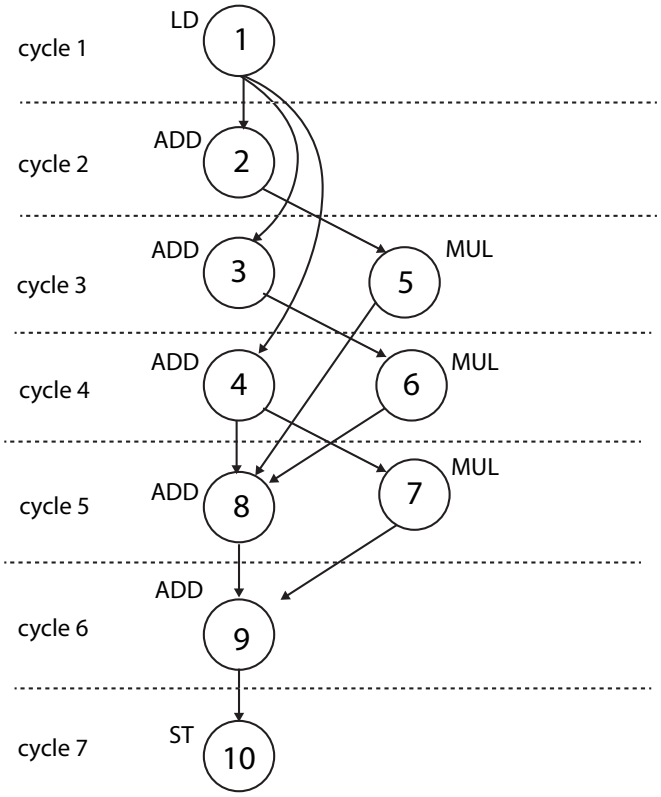


B. (4 pts) What is the maximum amount of instruction level parallelism (ILP) present in the program. Keep in mind that ILP is entirely a property of the program itself, not the machine it is run on.

Solution: The max ILP is 3. At two points in the program the instruction stream features three independent instructions that could be executed in parallel if a processor had the capability to do so.

C. (4 pts) Consider running this instruction stream on a single core, single-threaded processor that has superscalar execution capability to perform up to two instructions per clock, **PROVIDED THAT EXACTLY ONE INSTRUCTION IS A MUL**. In other words, the processor has two execution units (ALUs): one can execute add/load/store instructions and the other can execute only multiplications. Please assume all instructions (including loads/stores) individually complete in one cycle. You cannot modify the instructions in the program. What is the minimum number of cycles needed to execute this program? (Hint: think carefully about what instructions are allowed to run concurrently. The processing can run two instructions at the same time if they are independent and exactly one is a multiply.)

Solution: Seven cycles. The trick here was to notice that although the three independent adds must be serialized since there is only one unit that can perform them, some of the multiplies can issue at the same time as some of the adds. See the schedule below.



D. (4 pts) Now assume that the code above is changed so that it is: (1) running in a loop, where instructions 1-10 make up the body of the loop, and in each iteration of the loop the addresses stored in R4 and R5 are different. (2) Loop iterations are perfectly parallelized across `std::threads`. We are running this code on a **single core, multi-threaded processor that can process one instruction per clock** (arithmetic instructions, LDs, STs). However, from the moment a processor begins to execute a load instruction, there is a latency of 25 cycles before the data can be used by a dependent instruction. To be clear: if a core issues a LD in clock c, the LD occupies the core in clock c, but then the core cannot run an instruction that uses the value of the load until clock $c + 25$.

How many threads must be interleaved for the core to run at 100% efficiency? Please justify your answer.

Solution: Each time a thread stalls, it cannot make progress for 24 cycles (one cycle is the issuing of the load, followed by 24 cycles of waiting. Each thread in the system provides 10 cycles of latency hiding, so three threads are needed to hide latency. As a result a total of four threads are needed to run at peak rate. Note, the problem was designed so that students that used either 24 or 25 cycles of stall would get the same answer, so off-by-one issues did not matter.
The two main types of incorrect answers we saw were 1) 25 threads, which stems from not thinking about this problem in the steady state, which is what you should think for any throughput/utilization problem, and 2) 5 threads, which stems from thinking the processor in this question could leverage the maximum ILP present in this set of 10 instructions, while it can only handle one instruction at a time.

The patterns are pretty, but the SIMD efficiency may not be

Problem 3. (16 points):

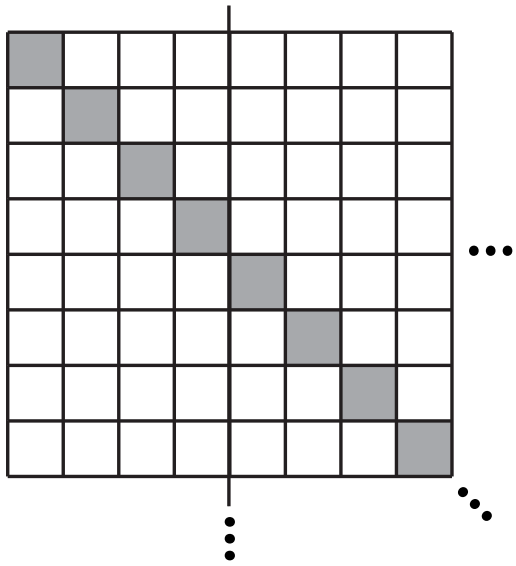
Consider the following ISPC code that processes a 16×16 input image (`input`) containing white, gray, or black pixels. It produces a 16×16 output image (`output`).

```
const int IMAGE_SIZE = 16;
void myfunction(uniform float* input, uniform float* output) {

  for (uniform int row=0; row<IMAGE_SIZE; row++) {
    for (uniform int col=0; col<IMAGE_SIZE; col+=programCount) {

      int idx = row*IMAGE_SIZE + col;
      float val = input[idx];    // load four bytes
      float result;

      if (!isWhite(val)) {
        if (isGray(val)) {
          result = foo1(val);    // 10 cycles of arithmetic
        } else {
          result = foo2(val);    // 10 cycles of arithmetic
        }
      } else {
          result = foo3(val);    // 10 cycles of arithmetic
      }
      output[idx] = result;      // store four bytes
    }
  }
```

The questions are on the next page...

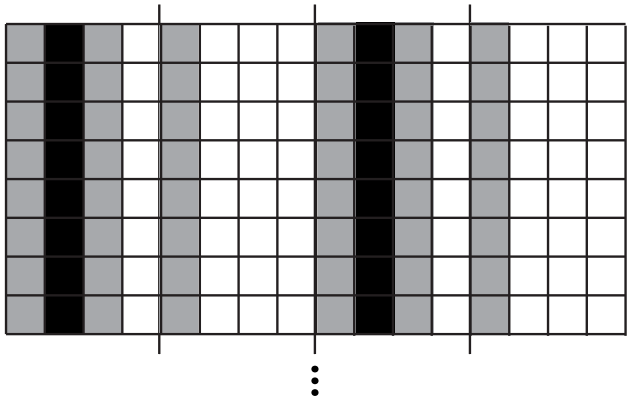
A. (4 pts) Consider running the code on the image below. **Note the full image is 16×16 pixels... the entire image is not shown in the figure but you can assume the pattern repeats as indicated, with only the diagonal being colored.**

Assume that the only arithmetic cycles we want you to count are the arithmetic instructions labeled in the code. (All conditionals and load/stores are "free".) What is the overall SIMD efficiency (50%, 100%, etc.) achieved when running the code on a 1 GHz single-core CPU with a **SIMD-width (and corresponding ISPC gang size) of 4**. In other words, the core can perform one 4-wide SIMD operation per clock. (Hint: start by computing the number of cycles needed to perform one row worth of the computation.)



*Solution: Each row of the image requires 16/4=4 iterations of the inner loop. Three of those iterations take 10 cycles, and 1 of the iterations takes 20 cycles due to divergence. As a result the overall efficiency of the computation is 80%. This can be computed in a number of ways: For example, one could compute the amount of time used by the actual program compared to a 100% efficient program that completes in 40 cycles: (10 + 10 + 10 + 10) / (10 + 10 + 10 + 20) = 0.8. Another approach would to be realize that the processor is running at 100% efficiency for 30 cycles and 50% efficiency for 20 cycles, therefore 100 * (30/50) + 50 * (20/50) = 80.*

B. (4 pts) What is the SIMD efficiency obtained when running the same code, using the same processor, but on the image below? Again, please keep in mind the image is 16×16 in size (e.g, the pattern of columns below just extends downward another 8 rows).



Solution: Due to the nested divergence, some iterations will take up to 30 cycles. The processing for the first four columns in a row takes 30 cycles, the second four columns takes 20 cycles, etc. So we have: (10+10+10+10) / (30 + 20 + 30 + 20) = 0.4, or 40% efficiency.

C. (4 pts) Now imagine that you are given a the following function:

`transpose(uniform float* input, uniform float* output)`

This function transposes the 16×16 input image and stores the result in a 16×16 output. (Recall that a transpose is `output[i][j] = input[j][i]`.) Write down pseudocode for how you would use calls to `myfunction` and `transpose` to produce a computation that **eliminates all execution divergence** when running `myfunction` **on the image from part B**. Yes, you can allocate temp buffers as needed.

Fill in pseudocode in `myNewFunction` below. Your solution should compute the same output image `output` as `myfunction(input, output)`.

Solution:

```
void myNewFunction(float* input, float* output) {
  float tmp1[IMAGE_SIZE*IMAGE_SIZE];
  float tmp2[IMAGE_SIZE*IMAGE_SIZE];
  transpose(input, tmp1);
  myfunction(tmp1, tmp2);
  transpose(tmp2, output);
}
```

D. (4 pts) Imagine that the code is run the same processor, but now with a memory system with a memory bandwidth of **8 bytes per clock**. Please assume that the `transpose` operation is **bandwidth bound** and the cost of the operation is equal to the time needed to transfer required data to and from memory. You should also assume that `myfunction` is **compute bound** and that the required time is a function of the number of cycles of arithmetic performed.

When running on the input image from part B Is the solution you proposed in part C faster or slower than the single call to `myfunction` used in part B? How much faster is it (in clocks)?

Please keep in mind that the processor:

- Can execute one four-wide SIMD operation per clock
- Is attached to a memory bus that can transfer 8 bytes of data per clock
- The input/output images are 16×16 elements in size.
- Each image element (a float) is 4 bytes

Hint: compute the amount of time needed to transpose the data, and then compute the amount of time (in clocks) needed to perform `myfunction` on this modified input.

Solution: A correct solution to part C incurs the cost of transposing the input and output images at the beginning and ending of the program, but receives the benefit of divergence-free execution in `myfunction`. The total number of bytes read/written is 256 elements×4 bytes/float×2=2048 bytes. At 8 bytes/clock, that's 256 clocks of "overhead". But the benefit of this computation is running `myfunction` in 16×40 clocks instead of 16×100 clocks. That's a savings of 16×60=960 clocks. It is definitely worth it to pre-transpose the data prior to doing the computation and then transpose it back at the end.

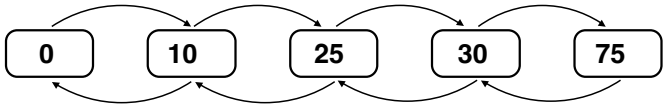
Two threads + a doubly Linked List = trouble

Problem 4. (18 points):

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which inserts a node by traversing the list starting from the head.
- `insert_tail`, which inserts a node by traversing the list **backwards starting from the tail**.

Insertions are the ONLY OPERATIONS on the data structure, and there are NO DELETES. Furthermore, you can assume that only two threads will ever be operating on the data structure at the same time, thread 1 is calling `insert_head`, and thread 2 calling `insert_tail`. (NOTE: these simplifications are important!)



Code for `insert_head` is given below. You can assume the code for `insert_tail` is similar. **NOTE THERE ARE NO LOCKS!**

```
struct Node {
  int value;
  Node* next, *prev;
};

void insert_head(Node* head, int value) {
  Node* n = new Node;
  n->value = value;

  Node* cur = head;

  // ignore insert at front of list case
  while ( /* position in list not found */ ) {

    if (n->value > cur->value &&
        n->value <= cur->next->value) {

      // link the new node forward/backward
      n->next = cur->next;
      n->prev = cur;

      cur->next->prev = n;  // link next back to n
      cur->next = n;        // link cur forward to n
      return;
    } else {
        cur = cur->next;    // step forward
    }
  }
}
```

Questions are on the next page...

A. (4 pts) Consider the case where thread 1 calls `insert_head(8)` and thread 2 concurrently calls `insert_tail(27)`. **Does the code generate correct output for all instruction interleavings?** Explain why. If not, draw one possible incorrect data structure that results and briefly describe the interleaving that causes this result.

Solution: The output is correct. Since modifications happen to different parts of the data structure, and thread 1 reads no variables written by thread 2 (and vice versa), the output is correct.

B. (4 pts) Consider the case where thread 1 calls `insert_head(27)` and thread 2 calls `insert_tail(26)`. **Does the code produce correct output for all instruction interleavings?** Explain why. If not, draw one possible incorrect data structure that results and briefly describe the interleaving that causes this result.

Solution: There are a lot of possible corruptions. Some valid ones are:

*(a) Losing one of the two inserted nodes (can be lost in either the forwards or backwards direction but not both). This will look something like 25 <-> 27 <- 26 <-> 30, but with 27 pointing forward to 30. This happens in the case where both to-be-inserted nodes update their links, and then the following instructions run in some interleaved manner. What happens next depends on which thread runs the first instruction to update the linked list first. If T1 runs `cur->next->prev = n` first, then T2 will insert its node between T1's node and T2's cur, and vice versa. In this case, the node that is inserted later will still have outbound links to 25 and 30. **Note:** The reason that we cannot fully lose a node is that the code to link one thread's inserted node into the linked list will affect the other thread's view of the list during insertion. In particular if T1 runs `cur->next->prev = n` before T2, it will update T2's "cur" node's next/prev pointer to point to T1's inserted node.*

(b) Inserting both nodes in an incorrect order, e.g. 25 <-> 27 <-> 26 <-> 30. This occurs if both threads pass the if statement check and then one thread completely finishes its insertion before the other one starts its insertion.

C. (4 pts) Consider a solution where threads use a 1-2-1 hand-over-hand locking strategy similar to the one discussed in class. (The thread grabs the lock for the next node it is traversing to in the list, so it holds two locks at once, and then releases the lock for the node it is leaving behind.) Assume that when inserting into the list, threads must hold a lock for the node before and after the future newly inserted node. Like in part C, consider the case where thread 1 calls `insert_head(27)` and thread 2 calls `insert_tail(26)`. What problem can happen now that the threads grab locks? Please describe the state of thread 1 and thread 2 that causes the problem (e.g., what locks are the threads holding?).

Solution: The code is not correct because deadlock can result. In this example, thread 1 might have the lock for node 25 and need the lock for node 30 to make progress. Thread 2 might have the lock for node 30 and need the lock for thread 25 to make progress.

D. (6 pts) Imagine that you had a function `trylock(Lock* l)` that locks the lock `l` if the lock is free, **but immediately returns false if the lock is not free.** (It does not block until the lock is acquired like a normal call to `lock(l)`.) Give an explanation of an implementation of `insert_head` that uses `trylock` to solve the problem you identified in the previous problem. To get full credit we require that your implementation must:

- Hold two locks: one for the node prior to and after new node `n`, when inserting `n`.
- Cannot perform repeated iteration through the list (e.g, it cannot restart from the beginning of the list if it fails to get the required locks)
- YOU DO NOT NEED TO WORRY ABOUT LIVELOCK.

Hint 1: keep in mind there are only inserts on the data structure, so a pointer cannot "disappear" out from underneath a thread. Hint 2: recall that if a thread has no locks on any nodes it has no guarantee what changes have been made to the data structure since it last checked. Nodes might have been added after it!

Your answer can be in pseudo code or words, just be clear about specifically what it does, and address all important cases.

Solution: The main idea is to release all locks when a trylock fails. Then repeat the following process until success.: Release the lock on the current node. At some time in the future, `lock()` (again) the current node, and recheck where the new node `n` should go in the list. Nodes might have been inserted after `cur` that `n` must now go after. Once the correct new position is found, attempt an insert if both required locks can be acquired. If not, repeat. This spinning is not particularly efficient, and does not protect against livelock or starvation, but in expectation it is a correct solution.

Load Linked / Store Conditional and Cache Coherence

Problem 5. (14 points):

A common set of instructions that enable atomic execution is load linked-store conditional (LL-SC). The idea is that when a processor loads from an address using a `load_linked` (LL) operation, the corresponding `store_conditional` (SC) to that address will succeed **only if no other writes to that address from any processor have intervened**.

Note that unlike `test_and_set` or `compare_and_swap`, which are single atomic operations, load linked and store conditional are two different operations... **each is atomic on its own, but the processor may execute other instructions in between a LL and a later SC.** Pseudocode for these instructions is given below.

```
int load_linked(int* addr) {
  return *addr;
}

// atomically perform this sequence
bool store_conditional(int* addr, int new_val) {
  if ( \* data in addr has not been written to by any processor *\
       \* since the last load_linked on addr                    *\ ) {
    *addr = new_val;
    return true;
  } else {
    return false;
  }
}
```

Consider the function `TryExchange()`, which is implemented using LL and SC as given below. `tryExchange` attempts to atomically read value of `x` and replace it with that value of `y`. It stores the old value at the address pointed to by `x` in the variable `z`, and **returns true if the atomic exchange succeeded.**

```
int TryExchange(int *x, int y, int *z) {
  *z = load_linked(x);
  return store_conditional(x, y);  // return true if swap actually occurred
}
```

A. (6 pts) Please implement a spin lock using `TryExchange`. (Your implementation can assume that calling threads behave reasonably and will not attempt to unlock a lock they they have not previously acquired, or lock a lock they already hold.)

Solution: The main idea of the solution to keep trying to perform a successful exchange. When the exchange succeeds we check to see if the value read from memory was 1. If so, then the lock was not free and we need to continue trying. If it is 0, that means the lock was taken.

```
void Lock(int* mylock) {
  int z;
  while (!(TryExchange(mylock, 1, &z) && z == 0));
}

void Unlock(int* mylock) {
  *mylock = 0;
}
```

B. (8 pts) Here is another way to implement a lock by directly using LL and SC. The lock is taken if the LL returns 0 and the SC succeeds, else the code tries again.

```
void Lock(int* mylock) {
  while (!(load_linked(mylock) == 0 && store_conditional(mylock, 1)));
}
```

We'd like you to analyze the cache coherence behavior of the two implementations of `Lock`: the one directly above, as well as your implementation based on `TryExchange`. Assume you have a multi-core processor that implements cache coherence using the MSI protocol. **(If needed, we provide a MSI state diagram on the next page.)** In addition to MSI, the system implements LL and SC as follows:

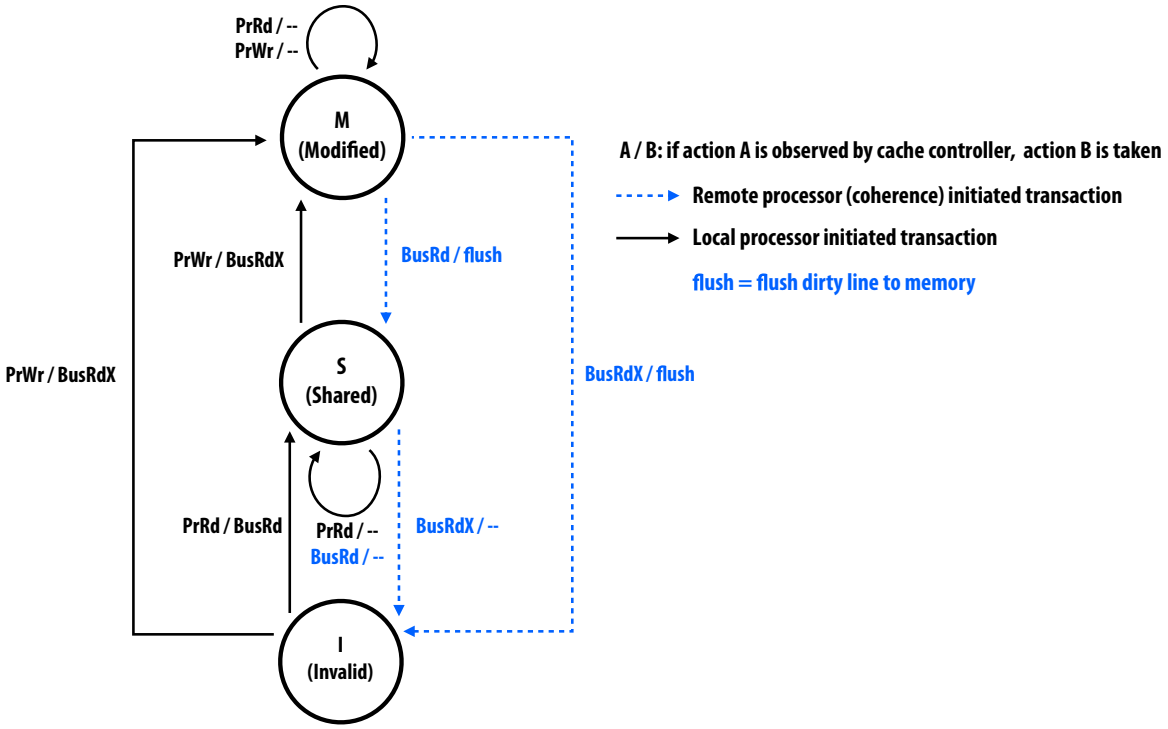
- LL is implemented by moving from the I state to the S state via BusRd (and setting an extra LL bit in the S state). If the line is already in M, it moves the line to S. (Note there is no need to issue any bus commands in this case, think about why!) If the line is already in S, no bus traffic is required.
- SC is implemented by checking that the line is in the S state in the local cache with the LL bit set. If it's not, the SC fails. If it is, then the processor moves the line from S to M and issues BusRdX and performs the update.

Assume that cache hits take 1 cycle (no bus traffic required), and bus transactions take 20 cycles (a cache miss is 1+20=21 total cycles). What's the performance difference between the two lock implementations assuming the while loop spins 100 times before the lock is zero? Please state any assumptions you make in your answer. Back-of-the-envelope calculations are fine, we aren't looking for a specific numerical answer, but you can give one if you want to.

Hint: remember that C code "early outs" if an AND expression cannot be true because the first term is false! (It only evaluates `A` when evaluating the expression `(A && B)` if `A` is false.)

Solution: The direct LL/SC lock spins on a bunch of LL operations until the lock is 0. It then executes the SC to try to take the lock. Therefore it executes at most one BusRd and then a bunch of cache hits. Then it executes a BusRdX to try to take the lock. (21 + 99 + 21 cycles). In other words, it behaves like the set and test-and-set lock we discussed in class.

In contrast, the `TryExchange`-based lock executes an SC in a loop while trying to acquire the lock (as part of the `tryExchange`. Now all threads trying to acquire the lock are generating writes (causing cache line invalidations, resulting in significantly more coherence traffic. (The LL might incur a miss for its BusRd (21 cycles) then the SC generates another BusRdX (21 more cycles) each iteration of the while loop.)


PrRd / --
PrWr / --
M
(Modified)
PrWr / BusRdX
BusRd / flush
S
(Shared)
PrWr / BusRdX
BusRdX / flush
PrRd / BusRd
PrRd / --
BusRd / --
BusRdX / --
I
(Invalid)
A / B: if action A is observed by cache controller, action B is taken
Remote processor (coherence) initiated transaction
Local processor initiated transaction
flush = flush dirty line to memory

Implementing Reader-Writer Locks

Problem 6. (16 points):

After the last problem you are hopefully quite familiar with LL-SC. In this problem, you will provide a simple implementation for read-write locks (which should remind you of the way invalidation-based cache coherence works) using the LL-SC primitives that were defined in the previous problem. (PLEASE SEE THE PREVIOUS PROBLEM FOR A DEFINITION OF LL-SC, BUT DO NOT NEED TO SOLVE THE PREVIOUS PROBLEM TO DO THIS PROBLEM.)

A read-write lock has the property that multiple threads may be holding the read lock, but **only one thread may be holding the write lock. If a thread is holding the write lock, no other thread may be holding a read lock.**

You may assume the follow simplifications:

- Sequentially consistent memory
- Threads will never call lock on a lock they hold, or unlock a lock they do not hold
- Your solutions may spin. We don't care about lock performance or fairness
- You may modify the `read_write_lock` struct if you wish, but you don't need to.

Hints:

- How do you implement atomic increment and decrement using LL/SC?
- You'll need some way to check to see if no other thread has the lock in either the read/write locked state.

The code to fill out is on the next page...

```
struct read_write_lock {
  // assume these two values are initialized to 0
  int num_readers;      // count of readers holding the lock
  int is_write_locked;  // 1 if there is a writer holding the lock
};
```

Solution: The gist of the solution is to force both readers and writers to hold the "writer" lock variable when determining if they can acquire the lock. Holding this variable means that no other thread can be writer, and no other threads can become readers. Readers immediately reset this variable to 0 once they've noted their reader status.

```
void write_lock(read_write_lock *l) {

  // Need to make sure there are no other writers.
  // This also prevents new readers
  while (true) {
    if (load_linked(l->is_write_locked) == 0 && score_conditional(&l->is_write_locked, 1))
      break;
  }

  // wait until all the readers are clear
  while (l->num_readers > 0);

  // now we have the writer lock and can proceed
}

void write_unlock(read_write_lock *l) {
  l->is_write_locked = 0;
}

void read_lock(read_write_lock *l) {

  // need to make sure there is not a current writer
  while (true) {
    if (load_linked(&l->is_write_locked) == 0 && store_conditional(&l->is_write_locked, 1))
      break;
  }

  // atomic increment count
  while (true) {
    int val = load_linked(&l->num_readers);
    if (store_conditional(&l->num_readers, val+1))
      break;
  }

  l->is_write_locked = 0;
}

void read_unlock(read_write_lock *l) {

  // atomic decrement count
  while (true) {
    int val = load_linked(&l->num_readers);
    if (store_conditional(&l->num_readers, val-1))
      break;
  }
}
```