Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.173 Fall 2010

Quiz 1

There are **7 questions** and **12 pages** in this quiz. Answer each question according to the instructions given. You have **80 minutes** to complete the quiz.

If you find a question ambiguous, be sure to write down any assumptions you make. **Please be neat and legible.** If we can't understand your answer, we can't give you credit!

Use the backs of pages if you need scratch space. You may also use them for answers, although you shouldn't need to. If you do use the blank sides for answers, make sure to clearly say so!

Before you start, please write your name CLEARLY in the space below.

This exam is open book.

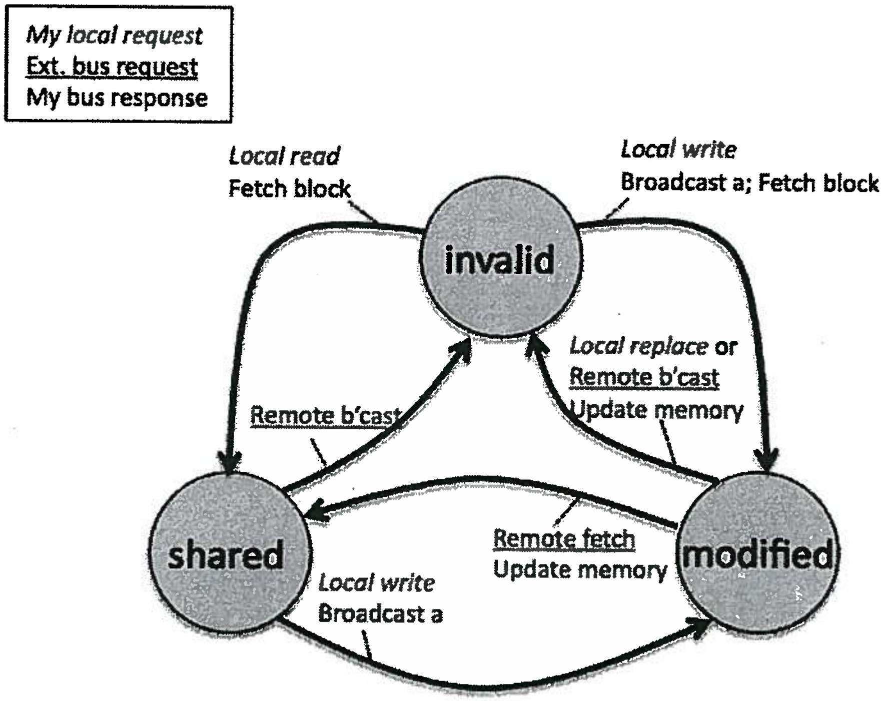
Do not write in the boxes below

1	2	3	4	5	6	7	Total
20 pts	15 pts	20 pts	15 pts	10 pts	15 pts	5 pts	100 pts

Name: SOLUTIONS

1. [20 points]: Cache coherence

Wrist Inc. has been selling a bus based multicore processor which uses the MSI cache coherence protocol characterized by the state machine shown below. "a" refers to the address of the location accessed by the local processor.



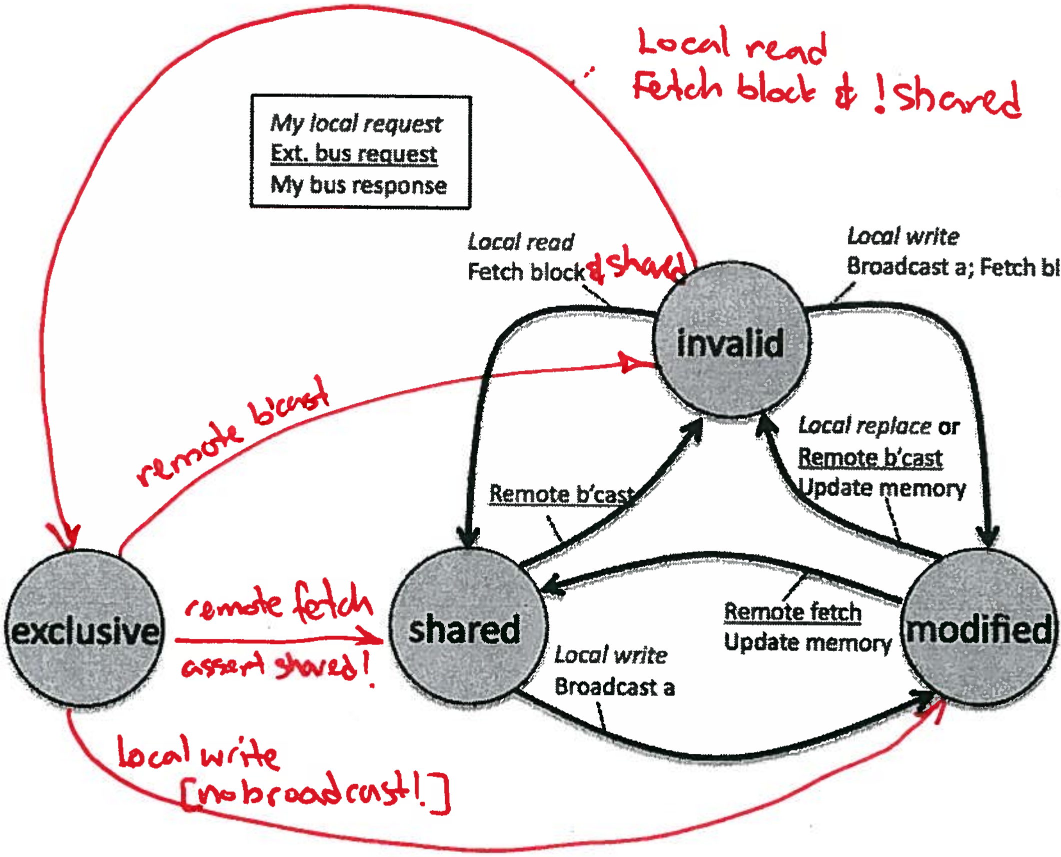
(a) (5 points) Briefly explain the purpose of the "Broadcast a" message a core places on the bus when it transitions from the `shared` state to the `modified` state.

Transition happens when local core is modifying a shared memory locations. Broadcast lets other caches know their copies are now stale and should be 'invalidated.

Wrist customers have been complaining that the Wrist bus often becomes a bottleneck, so Wrist engineers have come up with an idea to reduce bus traffic. They postulate that by splitting the `shared` cache block state into two states they can reduce bus traffic: `exclusive` (where an unmodified block lives in exactly one cache) and `shared` (where an unmodified block lives in more than one cache).

The engineers figure out that the added state requires two sets of changes. First, they have to add a new bus signal called `Shared/Not_Shared`. When a cache suffers a read miss and attempts to fetch the block from main memory, the `Shared/Not_Shared` bus signal is forced to the `Shared` state if any of the other caches have that block in their cache. Second, the state diagram must be updated.

(b) (10 points) Update the state diagram shown below by drawing the necessary transitions between the states `exclusive`, `shared`, `modified` and `invalid`, labeling each transition appropriately. You may have to modify some existing transitions and/or labels.

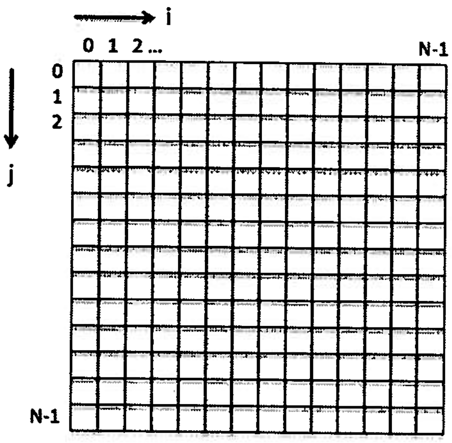


(c) (5 points) Briefly explain how including the `exclusive` state reduces bus traffic.

Writing to an exclusive cache line doesn't require a broadcast since no other cache needs to invalidate their copy.

2. [15 points]: Data partitioning

In this problem you will explore various was of creating an N-way partitioning a form of 2D Jacobi relaxation on a two-dimensional array A with dimensions $N \times N$ as shown below.



Unlike the familiar Jacobi relaxation in which we averaged 4 neighboring values according to the computation,

$$A[i,j] = (A[i,j+1] + A[i,j-1] + A[i-1,j] + A[i+1,j])/4$$

here the averaging uses 6 neighboring values according to the computation,

$$A[i,j] = (\underset{\text{next row}}{A[i,j+1]} + \underset{\text{prev row}}{A[i,j-1]} + \underset{\text{prev col}}{A[i-1,j]} + \underset{\text{prevprev col}}{A[i-2,j]} + \underset{\text{next col}}{A[i+1,j]} + \underset{\text{next next col}}{A[i+2,j]})/6$$

Assume a simple message passing machine model with N processors, in which a processor takes 1 cycle to send a word to a different processor. Similarly, receiving each word from a different processor also takes 1 cycle.

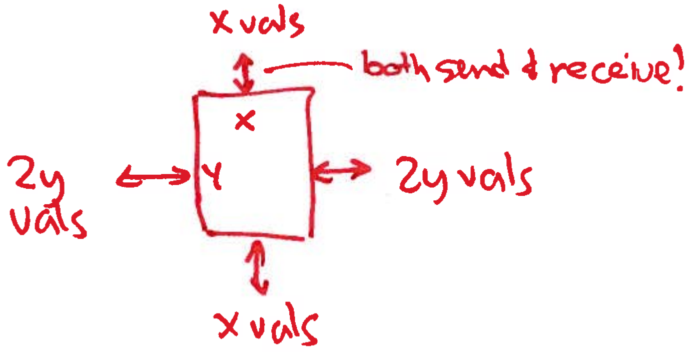
(a) (5 points) Partition by row: Suppose the data is partitioned so that each processor owns 1 row of data. Recall that each row has N words of data. Ignoring boundary effects, compute the number of cycles any given processor spends in sending or receiving data.

$$\left.\begin{array}{l}\text{send } \overset{1}{\text{row}} \text{ to N \& S neighbors : } 2N \text{ cycles} \\ \text{rcv } \overset{1}{\text{rows}} \text{ from N \& S neighbors: } 2N \text{ cycles}\end{array}\right\} 4N \text{ cycles}$$

(b) (10 points) Rectangular partitioning: Now suppose we partition the array among the N processors such that each processor gets a rectangular tile of data with x elements in the i dimension and y elements in the j dimension. Assume that $x \geq 2$, $y \geq 2$ and, for load balancing, assume that the following constraint is met:

$$N = xy$$

Compute the number of cycles in terms of x and y that any processor spends sending or receiving data for this rectangular partitioning.



total send + receive = $4x + 8y$ cycles

3. [20 points]: Software Barrier

Recall that a *barrier* ensures across all cores that *all* actions before the i^{th} barrier call are completed before *any* actions after the i^{th} barrier call. Grace has provided the following correct code of a barrier implementation that only sends messages to the next sequential core. In Grace's implementation core 2 plays a special role in initiating the messaging involved in implementing the barrier functionality. Grace assumes that sw_barrier is the only generator of message traffic.

```
sw_barrier()
{
  int me = corenum();     // my core number
  int next = (me == enetCorenum()-1) ? 2 : me+1;    // next core number
  IntercoreMessage dummy;    // an empty message

  // unless I'm core 2, wait until prev core sends us a message
  if (me != 2) while (message_recv(&dummy) == 0);

  // tell next core we've entered the barrier
  message_send(next,msgTypeBarrier,&dummy,0);

  // wait until prev core sends us a message.  Core 2 is waiting for
  // last core to enter the barrier, other cores are waiting for
  // earlier cores to message that they're exiting the barrier.
  while (message_recv(&dummy) == 0);

  // unless I'm the last core, tell next core we're exiting the barrier
  if (next != 2) message_send(next,msgTypeBarrier,&dummy,0);

  // all done! return from barrier
}
```

N messages

N-1 messages

(a) (4 points) If N cores are participating in the barrier, how many total messages are sent for the i^{th} invocation of sw_barrier?

total messages = $N + (N-1) = 2N-1$

Ben Bitdiddle doesn't want core 2 to be special so he deletes all the conditional executions based on the core number. In the copy of Grace's code below, Ben's deletions are shown using ~~strikeout~~ notation.

```
sw_barrier()
{
  int me = corenum();     // my core number
  int next = (me == enetCorenum()-1) ? 2 : me+1;    // next core number
  IntercoreMessage dummy;    // an empty message

  // ~~unless I'm core 2,~~ wait until prev core sends us a message
  ~~if (me != 2)~~ while (message_recv(&dummy) == 0);
```

Everyone waits here!

```
  // tell next core we've entered the barrier
  message_send(next,msgTypeBarrier,&dummy,0);

  // wait until prev core sends us a message.  Core 2 is waiting for
  // last core to enter the barrier, other cores are waiting for
  // earlier cores to message that they're exiting the barrier.
  while (message_recv(&dummy) == 0);

  // ~~unless I'm the last core,~~ tell next core we're exiting the barrier
  ~~if (next != 2)~~ message_send(next,msgTypeBarrier,&dummy,0);

  // all done! return from barrier
}
```

(b) (8 points) Sadly this code does not implement the desired functionality. To help Ben understand why give a specific scenario where the code above will fail.

As cores enter barrier, they loop waiting for a message - but no core sends a message!

⇒ enter first barrier, but never exit.

Disappointed that his last "improvement" failed, Ben makes another attempt. Not really understanding why Grace has implemented a two-phase process (first message indicates barrier entry, second message indicates barrier exit), he removes the second phase altogether. Again, in the copy of Grace's code below, Ben's deletions are shown using ~~strikeout~~ notation.

```
sw_barrier()
{
  int me = corenum();     // my core number
  int next = (me == enetCorenum()-1) ? 2 : me+1;    // next core number
  IntercoreMessage dummy;    // an empty message

  // unless I'm core 2, wait until prev core sends us a message
  if (me != 2) while (message_recv(&dummy) == 0);

  // tell next core we've entered the barrier
  message_send(next,msgTypeBarrier,&dummy,0);

  ~~// wait until prev core sends us a message.  Core 2 is waiting for~~
  ~~// last core to enter the barrier, other cores are waiting for~~
  ~~// earlier cores to message that they're exiting the barrier.~~
  ~~while (message_recv(&dummy) == 0);~~

  ~~// unless I'm the last core, tell next core we're exiting the barrier~~
  ~~if (next != 2) message_send(next,msgTypeBarrier,&dummy,0);~~

  // all done! return from barrier
}
```

(c) (8 points) Briefly explain the purpose of the second set of messages and give a specific scenario where the code above will fail.

The second set of messages ensures no core leaves the barrier until all cores have entered the barrier.

Without the second set of messages core 2 will exit barrier as soon as it sends its first message, even if no other cores have entered barrier!

4. [15 points]: Parallel TSP

Eliza is writing a parallel TSP program for the Beehive, and she has some questions for you.

Eliza keeps the current minimum known cost for branch-and-bound in a global 32-bit integer variable mincost. Each core reads `mincost` when deciding whether to abandon investigating a path, and updates `mincost` when it finds a shorter path. Eliza wonders whether she needs to protect uses of `mincost` with a lock. She is trying to choose between this locking code sequence:

```
icSema_P(costSema);
cache_invalidateMem(&mincost, sizeof(mincost));
if (current_cost < mincost) {
  mincost = current_cost;
  cache_flushMem(&mincost, sizeof(mincost));
}
icSema_V(costSema);
```

and this non-locking code sequence (identical except for the missing semaphore calls):

```
cache_invalidateMem(&mincost, sizeof(mincost));
if (current_cost < mincost){
  mincost = current_cost;
  cache_flushMem(&mincost, sizeof(mincost));
}
```

(a) (7 points) Describe a scenario in which the non-locking code sequence would result in slower TSP execution than the locking code sequence.

With no locking potential new mincost might get overwritten: Eg, old mincost = 100. A discovers mincost = 10, B discovers mincost < 99. A flushes, then B flushes ⇒ mincost = 99. So many additional searches may be performed unneccessarily.

(b) (8 points) Suppose `mincost` were a 64-bit integer instead of a 32-bit integer, the difference being that it takes two 32-bit memory accesses to read or write `mincost` and although the two words occupy consecutive memory locations, the words may not be in the same cache line. Suggest a potential *correctness* problem with the non-locking code sequence.

LSW MSW

Suppose mincost = $\boxed{0 \mid 2}$ and then A publishes new mincost: $\boxed{27 \mid 0}$ and B publishes $\boxed{0 \mid 1}$.

If the order of the flushes is A_{LSW}, B_{LSW}, B_{MSW}, A_{MSW} then the result is $\boxed{0 \mid 0}$ which will cause searches to abort that might have discovered the true min cost.

5. [10 points]: Shared Memory Synchronization

Fetch-and-inc is a wait-free synchronization operation that allows any processor to atomically increment a shared memory location, receiving the original value as the result of the operation. It is used to prevent the interleaving of read, add, and write operations from multiple processors trying to increment the location concurrently.

Finish this software implementation of fetch-and-inc. You should use only existing Beehive mechanisms.

```
// atomically increment location, returning old value
int fetch_and_inc(int *valptr)
{

   icSema_P(valLock);   //gain exclusive access

   cache_invalidateMem(valptr, sizeof(*valptr));  //remove stale value

   int rv = *valptr;


   *valptr = rv + 1;

   cache_flushMem(valptr, sizeof(*valptr));   // write to memory

   icSema_V(valLock);   // release the lock.


   return rv;
}
```

6. [15 points]: Verilog & Finite State Machines

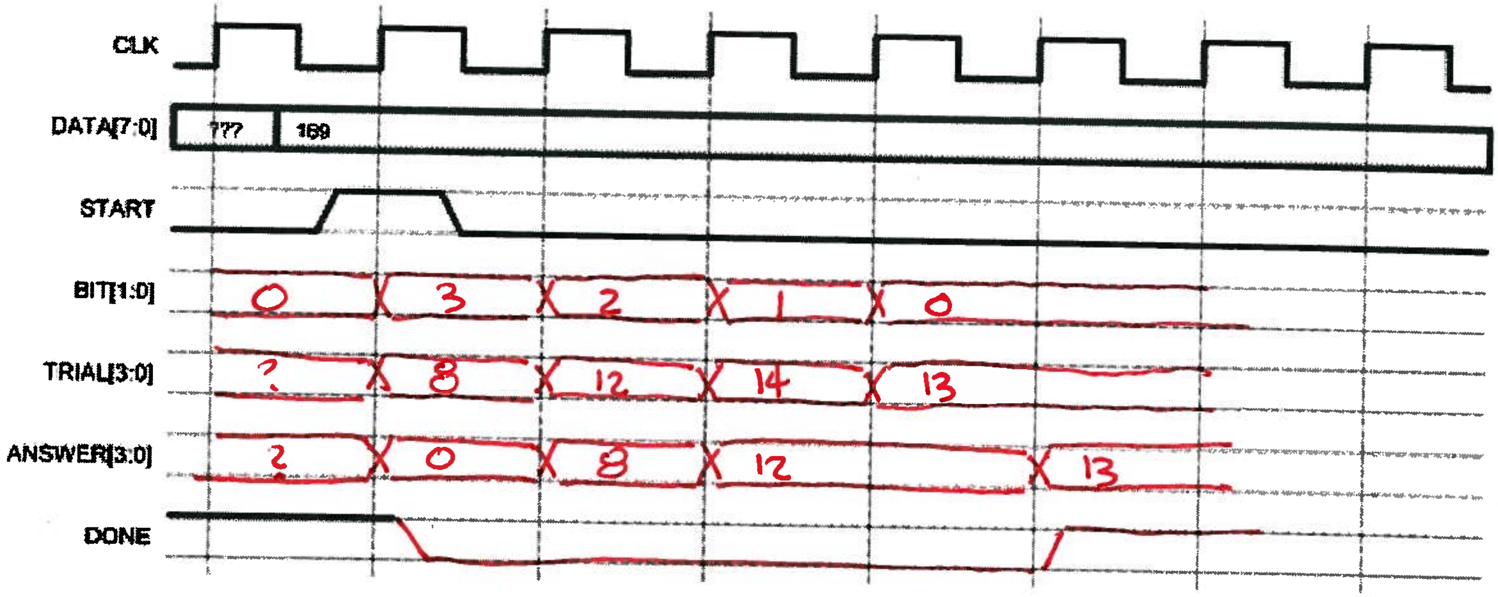
Consider the following Verilog module that iteratively computes the square root of an 8-bit integer value.

```
module sqrt(input clk,start,
            input [7:0] data,
            output reg [3:0] answer,
            output reg done);
  reg busy;
  reg [1:0] bit;

  wire [3:0] trial;
  assign trial = answer | (1 << bit);  // << is left shift

  always @ (posedge clk) begin
    if (busy) begin
      if (bit == 0) busy <= 0;
      else bit <= bit - 1;
      if (trial*trial <= data) answer <= trial;
    end
    else if (start) begin
      busy <= 1;
      answer <= 0;
      bit <= 3;
    end
  end

  assign done = ~busy;
endmodule
```

(a) (3 points) If start is asserted during cycle 100, during which clock cycle will done be asserted? Hint: the FSM always runs for a fixed number of clock cycles before asserting done.

cycle 100: start asserted
101: bit = 3
102: = 2
103 = 1
104 = 1
⇒ 105 done asserted

(b) (12 points) Please neatly complete the timing diagram below as the module computes the square root of 169:



7. **[5 points]:** Feedback!

Please take a couple of minutes to give us some feedback on the course. Has the course material been what you expected? How much time have you spent on the labs? Which course activities (lectures, labs, paper readings) have worked well? Which need re-engineering? Any feedback you wish to provide would be most welcome.

Thanks for your comments!

END OF QUIZ 1!