1. 4-Way Traffic Light Controller

The project is aimed at designing a traffic light controller that controls 4 traffic lights at a 4-point crossing.
Let us suppose the 4-point crossing is the junction of 2 roads: one running in the North-South (NS) direction and another running in the East-West (EW) direction. Each of these roads is again a 2-lane road; one for traveling up (E or N) and another for traveling down (W or S). The traffic light controller should be implemented to avoid collisions and minimize waiting time for vehicles.

The main module should tie together 2 sensors, a timer and 4 traffic lights (1 for each lane).

The sensor will be used for detecting traffic pressure. At each clock tick, it non-deterministically outputs a value of 1/2/3 for zero/light(1-5 cars)/heavy traffic(above 5 cars) respectively.

For each of the 2 roads, traffic should be allowed alternatively and traffic in the 2 orthogonal roads should also be allowed alternatively. Hence, an instance of valid traffic allowance should look like: N->S, E->W, S->N, W->E.

The timer sub-module implements a timer that outputs "long" and "longer" timeouts.

A traffic light on the NS road stays green for at least time for light traffic and "longer" time for heavy traffic. Any time after this time, if there is a car waiting on the EW road, then the other light turns green. The other light remains green until there are no more cars on the EW road, or until the "long" or "longer" time expires. The yellow light for both directions stays yellow for time. Note that only a single timer is used for both the EW and NS controllers. In theory, this could lead to conflicts; as implemented, such conflicts are avoided. From the START state, the timer produces the signal after a nondeterministic amount of time. The signal remains asserted until the timer is reset (via the signal). From the SHORT state, the timer produces the signal or "longer" after a nondeterministic amount of time depending on the traffic pressure. The signal or "longer" remains asserted until the timer is reset. Notice that the use of non-determinism in the description of the timer models an infinite number of actual implementations, each with a different set-up for the time periods.

The EW lights stay RED until they are enabled by the NS control. At this point, it resets the timer, and any 1 of the lanes moves to GREEN. It stays in GREEN until there are no cars, or the long or longer timer expires. At this point, it moves to YELLOW and resets the timer. It stays in YELLOW until the short timer expires. At this point, it moves to RED and enables the NS controller.

The NS lights stay RED until they are enabled by the EW control. At this point, it resets the timer, and moves to GREEN. It stays in GREEN until there is heavy traffic pressure on the EW road or the long or longer timer expires. At this point, it moves to YELLOW and resets the timer. It stays in YELLOW until the short timer expires. At this point, it moves to RED and enables the EW controller.

Design your own functionalities in the test-bench for simulating the components as well as the overall system. Also, please make valid/necessary assumptions (if you require).

2.TCP Connection Establishment

The Internet's Transmission Control Protocol is probably the most widely used transport layer protocol that offers a reliable, connection-oriented, byte-stream service and is also the most carefully tuned. It has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time. Finally, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers. In addition to the above features, TCP also implements a highly tuned (and still evolving) congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from over-running the receiver, but so as to keep the sender from overloading the network. The aim of this project is to implement the 3-way handshake protocol for a TCP connection establishment and the connection teardown phase.

Following are the steps for connection and termination of a TCP connection:
1. The client application opens a connection to the server by sending a TCP segment which only the header is present (no data). This header contains a flag SYN stands for "Synchronize" and the TCP port number the server (application). The client is in SYN_SENT state (SYN sent).
2. The server (application) is listening (listen) and on receipt of the SYN from the client, it responds with a SYN and ACK flag. The server is then in SYN_RCVD (SYN received) state.
3. The client receives the server's TCP segment with SYN ACK indicators and moves in status ESTABLISHED. He also sends a response ACK to the server that also passes in status ESTABLISHED.
This exchange in three phases (three-way handshake) completes the establishment of the TCP connection that can now be used to exchange data between the client and server.
4. In the event that a connection request arrives on the server and that no application is listening on the requested port, a segment with flag RST (reset) is sent to the client by the server, the connection attempt is immediately terminated.
Once a connection has been established, the communicating parties keep listening to their respective ports for incoming packets until there is a connection termination request from any of them.
5. As a TCP connection is bidirectional (full duplex) , the connection termination process should be made in both directions of the communication. The client as well as the server can send a segment with FIN flag, this means an end to sending data. Receiving a segment with FIN indicates that the other end will not send more data. The connection then is half closed.
6. The other party acknowledges the FIN with ACK, informs the application for the release of this connection and when done then sends a FIN segment to the termination initiator which in turn, acknowledges it with an ACK flag.

All segments exchanged between client and server are numbered sequentially. Each side of the connection initiates and maintains the sequence of segments sent.

Design two protocol controllers, one for the server and one for the client that generate appropriate flags on receipt of the different kinds of packets described above. Make valid assumptions wherever necessary.

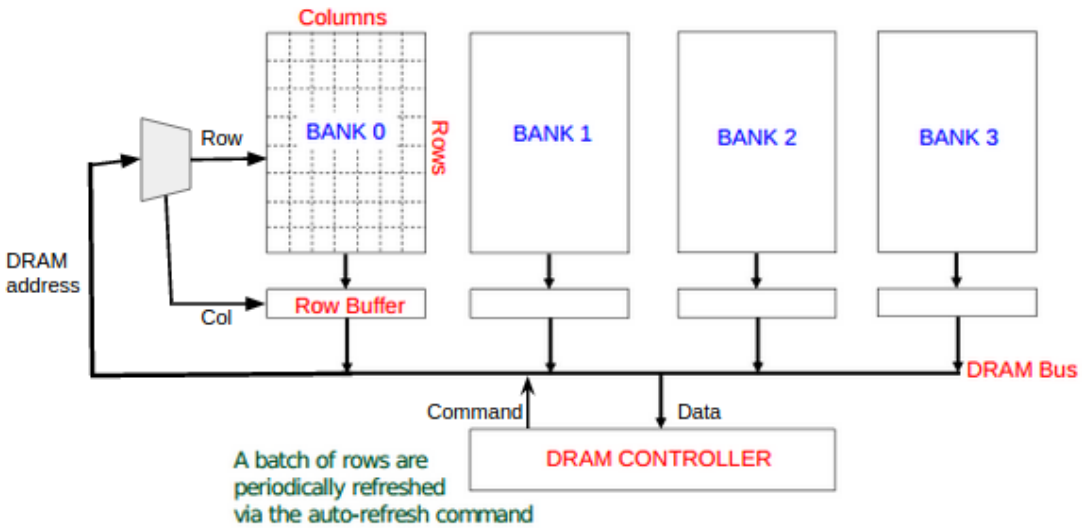
3. DRAM controller:

A DRAM controller is an on chip component of the processor, responsible for communicating with off chip DRAM systems. Data is actually stored inside multiple banks in interleaved fashion. All banks of the DRAM can be accessed parallely. Each bank has a two dimensional matrix of bit cells(capacitor) and a row buffer.
Below are the steps/commands to read data from DRAM :

a) RA (row access): An address in DRAM gets divided into a tuple <Bankid, rowid, colid>. Bankid selects the bank; while the rowid selects a row and all the bit cells in the row are stored in the row buffer.
b) CA (column access): A column is selected by the colid and the bit corresponding to that column stored in the row buffer is sent out of the bank.
c) PRE (precharge): When all the required columns are read out, the entire row in the row buffer is written back into the original row position of the bank (since the capacitors leak the charge over time which may corrupt the data).

When a request of different row arrives at controller, first the existing row inside the row buffer needs to pre precharged before the new row's RA step begins.Assuming the number of banks is 8. Each bank has a 128 X 8 matrix of bit cells, i.e each bank can hold 1024 bits of data. Write a verilog code to implement this design.
Besides the above mentioned functionality, there are additional commands that need to be supported.

REFRESH: Even though all rows of the DRAM bank are not accessed, still because of the leaky nature of capacitors, each row of the DRAM bank needs to be periodically refreshed. During the refresh time of a row, same row access is prohibited. Assume refresh rate is 125ms.



Constraints:

a) Since inside the DRAM there is no additional command buffer, all the commands need to be sent one by one (sequentially).
b) Each command has its own execution delay. So the controller has to send the commands in such a way that all such delay constraints of DRAM can be maintained and no command gets ignored.
c) Periodically REFRESH command needs to be issued

d) Though each bank can be accessed simultaneously but at a time only one bank can transfer data to DRAM controller via DRAM bus.

4. Vending Machine:

Design a vending machine where a user can order a tea, cookies ,coffee, 3 varieties of candies and 4 varieties of chocolates which cost $5, $20, $7, $10/-, $20/-, $25/-, $ 30, $ 10, $25 and $50/-, respectively, for each of these items. We assume that the amount of money that can be inserted at a time is $1, $ 2, $ 5, $ 10, or by credit/debit card. So, if the user wants to enter a higher amount, then they have to enter it multiple times. The user is also provided with a return money option so that at the end of their transaction they can take the remaining money (if more was inserted). The user can make 3 transactions at a time and they can take the change at the end of their operation.
In case of multiple options to return change the vending machine must choose the one that has the minimum number of coins to be dispatched. Eg: $ 20/- can be dispatched as 1 $ 20/- note, two ten dollar notes or or 20 one dollar coins. The vending machine must give priority for $.20/- note as it requires only one note to be dispatched and in case $20 rupee notes are not available then it should go for the second option.
Design your own functionalities in the test-bench for simulating the components as well as the overall system. Also, please make valid/necessary assumptions (if you require).

Inputs:

1. m0: input indicating $.10/- has been inserted
2. M1: input indicating that $.1/- has been inserted
3. M2: input indicating that $. 2/- has been has inserted
4. M3: input indicating that $.5/- has been inserted.
5. M4: input indicating that a credit card has been swiped/
6. M5: input indicating that a debit card was used.
7. ti: input indicating that user wants tea
8. wi: input indicating that user wants a cookie
9. ci: input indicating that user wants coffee
10. B1: input indicating that user wants candy typ1
11. B2: input indicating that user wants candy type 2
12. B3: input indicating that user wants candy type 3
13. C1: input indicating that user wants chocolate type 1
14. C2: input indicating that user wants chocolate type 2
15. C3: input indicating that user wants chocolate type 3
16. C4: input indicating that user wants chocolate type 4
17. rr: input indicating that change should be dispatched (end of his/her transaction)

Outputs:

1. so[10]: output indicating that a particular item is out of stock (shelf empty)
2. T0: Indicates tea has been served.
3. Wo: output indicating that water has been served to the user
4. Co: output indicating that coffee has been served to the user
5. Bo[3]: output indicating that candy X has been served to the user
6. C[4]: output indicating that chocolate type X has been served to the user
7. RRo: output indicating the returned money
    1. RR0 = 0, if no money is returned

2. RRo = 1, if $10/- is returned
3. RRo = 2, if $ 20/- is returned
4. RRo=3,if $2/- is returned
5. RRo =4 if $5/- is returned
6. RRo=5 if $1/- is returned

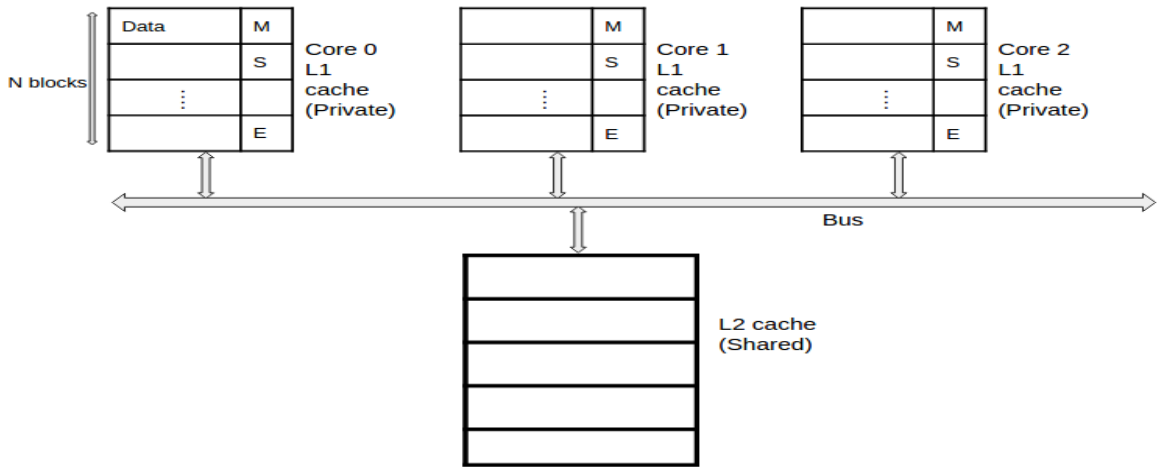
5. Cache coherence protocol:

In modern computer architecture, memory hierarchy (L1->L2->..->Main Memory) plays an important role for improving the performance of the system. However in multi core system data consistency becomes an issue for the shared data. So cache coherence maintains the uniformity/ consistency of shared data stored inside the local cache of each core. A number of protocols have been evolved to support this coherence over a period of time. In this assignment you have to implement and verify one of the commonly used protocols; MESI protocol.

We will explain the protocol with an example:

In the below figure, there are 3 cores each with a Private L1 cache and a shared L2 cache; connected by a bus. Each L1 cache is divided into n blocks.Each block has two parts- data and its state. To maintain the coherence each block can be one of the four states (i,e M, E, S, I).

a) Initially when a program starts executing, all the blocks in L1 are in invalid or I state.
b) Any read or write access to the block in I state results cache miss and initiates a request into the bus.
c) The request must be serviced by either one of the other cores or the L2 cache, based on the state of the requested block of each core.
d) A cache block in M (modified) state at core C means, core C has updated (written) the block recently and no other cores along with L2 has the same copy.
e) A cache block in S (shared) state at core C means, core C has the same copy of the block which at least one of the two other cores and L2 has.
f) A cache block in E (exclusive) state at core C means only core C and L2 have the copy of the block.
g) So when core D sends a read request for a block which is in E state at core C, C will provide



the data to D, instead of L2. In that case, that block state at core C remains E, however at D the state becomes S.

h) If a block is in S state at both core C and D and core C wants to write the data into the same block; then the state of the block at core C will become M from S and I from S at core D (i,e C will invalidate the data core D has).
i) Only at the time of block replacement, the data value will be written back to L2 cache, provided that the replaced block is in M state - write back cache replacement policy.

a) Write the verilog code to implement MESI protocol with 4 cores with private L1 and one shared L2 cache. Use the following specifications for implementing the MESI protocol.

Design specifications:

a) Number of cores : 4
b) Number of L2 cache :1
c) Number of blocks in each L1 cache: 4
d) Number of blocks in L2 cache: 8
e) Address mapping in L1 cache: Direct Mapping

Inputs:

Read(R) or Write(W) requests generated by each core on different addresses.

Eg- Core 0 -R- <16, ->
Core 1- R- <16, ->
Core 2- W- <16, 120> ….

<Address, Data value>

Outputs:

a) State of each block (tuple of two information < coherence state, data value>) in each core's L1 cache.
b) State of each block (only data value) in L2 cache.