EE 457 Lab 4 Cache Controller

1 Introduction

In shared memory multi-core systems, a main challenge is to keep the memory system coherent among all the cores within the MPSoC (Multi-processor System-on-Chip). As a result, diverse types of cache coherency protocols have been introduced to maintain coherency. The MSI protocol is the cornerstone protocol that most of the cache coherency protocols are based on. In this lab assignment, we emulate the memory system of a single core in part 1 and then enhance our cache to MSI protocol in part 2

2 What you will learn

This lab will help you:

- Apply cache mapping schemes to determine hit/miss.
- Design an MSI cache coherency implementation
- Further develop your Verilog description skills

3 Procedure

3.1 Part 1. Emulation of Cache (40 pts.)

In this part, we will emulate the behavior of the single processor memory system. The address space of the main memory for a single core byte-addressable processor is assumed to be 64 bytes. The internal data bus of the processor is 8 bits wide. Thus, 1 "word" is simply 1 byte. The cache block size is 2-bytes (i.e. 2 words) and the processor is enhanced with a direct-mapped cache with 4 blocks (i.e. 8 bytes). As a result, the physical and cache addresses will follow the format as shown in the Fig. 1. To facilitate single access block transfers to and from memory, the main memory is 16-bits wide and always performs a read/write of that data size. Thus, we only need the upper 5 bits of the processor address bus to address the 32 cache blocks (64 bytes) that exist in main memory. Cache organization is demonstrated in Fig. 2.

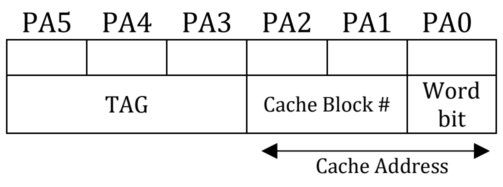


Fig. 1: Physical and Cache Address Format

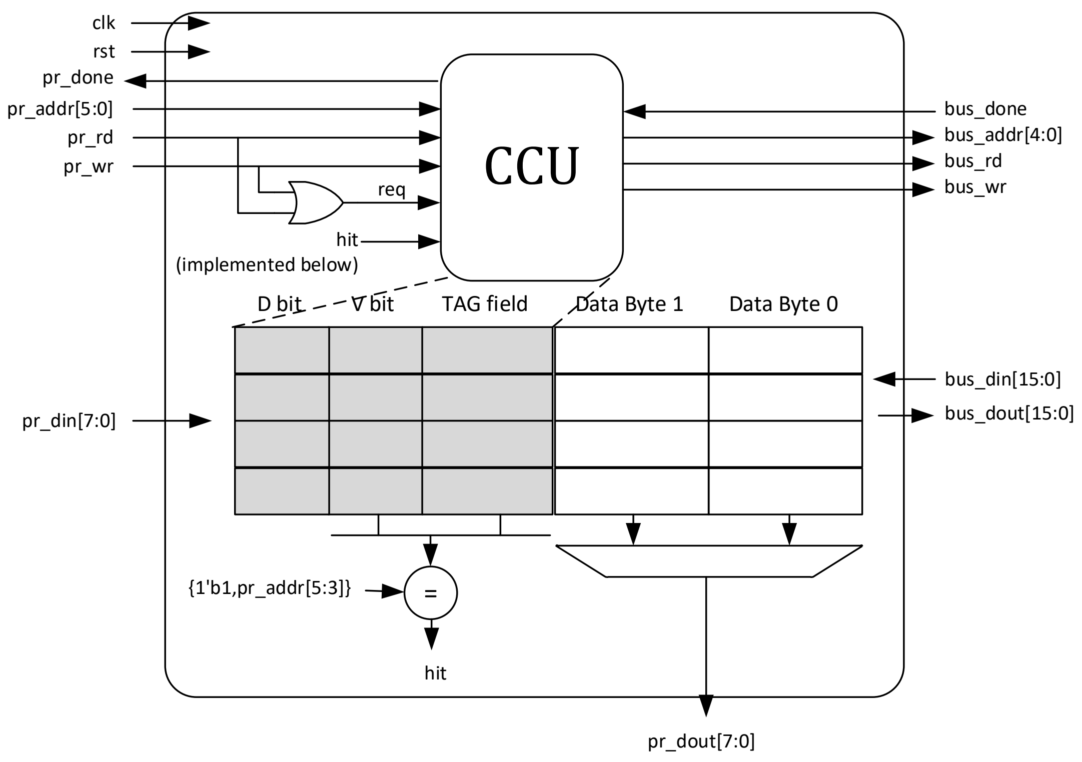


Fig. 2: Direct-mapped Cache Organization

The detailed explanation for signals used as cache ports are as follows:

pr_din[7:0]:	Data provided by the processor to be written in the cache.
pr_dout[7:0]:	Cache data provided to processor.
pr_addr[5:0]:	Address issued by the processor.
pr_rd:	Processor memory read request, issued when dealing with LW instruction.
pr_wr:	Processor memory write request, issued when dealing with SW instruction.
pr_done:	Cache operation is valid/complete. On a read the data should be on pr_din when cvalid is asserted so that the processor can use it as an enable to capture data. On a write cvalid indicates the cache operation has completed. The processor will maintain pr_rd and pr_wr until cvalid is true at a clock edge.
hit:	Internal cache control signal indicating a valid block with matching tag comparison.
bus_din[15:0]:	Full 2-byte cache block provided by the memory to be written in cache as a result of a cache miss.
bus_addr[4:0]:	Block (2-byte) address provided to memory by cache in case of a cache miss (to be used for writeback or fetch of a block)
bus_rd:	Bus read request by cache in case of a need to fetch.
bus_wr:	Bus write request by cache in case of a replacement.
bus_dout[15:0]:	Full 2-byte cache block provided by the cache to be written in the main memory in case of a writeback.

bus_done:	The memory operation is complete. If a memory read, then the data on mem_dout is valid and may be captured by the cache. There is a gap in the speed between memory and cache, which for the sake of this simulation we take to be 10 clocks in our test bench.
CLK:	system clock
Reset:	system reset (active high)

To check whether the block residing in the cache is the same as the one requested by the processor, the cache is indexed with pr_addr[2:1] (i.e. the 'cblk' field) and then the TAG of the cache block is compared with pr_addr[5:3]. V and D are valid and dirty bits, respectively. C.C.U. stands for Cache Control Unit and oversees coordination between processor and the bus (i.e. main memory). If a block is missed in the cache, the CCU will request the block from the bus and waits until memory provides the data to the cache. When the memory finishes the transaction requested by the core, it sets bus_done signal high for a clock so the cache knows that the transaction is done. The cache that we implement here is a write-back cache; therefore, if a block is dirty in the cache and the processor wants to update it with another block, the dirty block needs to be written in the main memory first. Note: While the valid and TAG bits would generally be stored in a separate TAG RAM, we will just store them as a simple register array in the CCU (this is why those bits are shaded and lines drawn in Fig. 2 to indicate that TAG, V, and D are actually stored in the CCU).

The memory has been designed completely and given to you as memory.v. Our memory is 10 times slower than the cache meaning it handles the memory requests issued to it after 10 processor clock cycles. If the request is a memory read, the data is read from it after 10 clocks. If the request is a write request, the data is written to it after 10 clocks. Once the request is performed by the memory, it sets the bus_done signal high for 1 CPU clock so the CCU knows the transaction is done. The memory initializes itself with a user provided text file as "datamem.txt". Each line in this file represents a byte in the memory and the line number represents the memory address starting from address 0. The format of the data in each line is in hex and each line in the file is initialized with the line number in hex starting from 0 for sake of simplicity (i.e., mem[i]= i , for example: Mem[11 hex]=11 hex = 8'b00010001)

Steps required to be done for Part1:

1. Complete the state diagram for CCU as provided in the Fig. 3 using the signals demonstrated for CCU ports in the Fig. 2. **(10 pts)**

   The description of the states in the Fig. 3 are as follows.

   Initial: Initial is the power-on state of the system. We clear the valid and dirty bits of the cache blocks as well as the memory control signals.

   Monitor: We want to support a **single-clock read/write on hits**. Thus we need to handle hits completely in this state using a Mealy approach. Only misses will cause us to transition to a new state. If we do have a miss, we may need to write back a block (if it is dirty) before fetching the desired block. If we indeed do have a miss, transition to an appropriate state where we can use a Moore-style approach to generate appropriate bus signals. To be more specific:

   - On a read hit, we can read the data immediately. The CCU need do nothing special and we can *stay in the **Monitor** state*.

- On a write hit, we perform the write on the specified **byte** of the cache block and update any necessary state (you should think about what bit(s) need to be updated). We can *stay in the* ***Monitor*** *state.*
- On a **miss** we must transition to either the **WB** or **Fetch** state. We must check if the current block in cache is dirty and write it back if so. We can initiate that operation by moving to the **WB** state where will generate the appropriate memory bus signals to write the block back and wait for the operation to complete. If the block is not dirty, we can move to the Fetch state where we can generate the appropriate memory bus signals to read the desired block.

WB: In this state, we generate the bus signals for the write/flush and wait for the memory operation to complete. However, once complete we need to start fetching the desired block from memory by moving to the **Fetch** state.

Fetch: In this state, we generate the bus signals to read/fetch the desired data (from the memory) until we see the bus_done signal asserted at which point we write the appropriate data into the cache. If the processor intended to write, be sure that specific byte from the processor is placed alongside the other byte read from memory to effectively perform the write at the same time as the cache captures the data. Also be sure to update the valid, tag, and dirty bits.

After servicing a read or write miss via the (**WB** and) **Fetch** state, we return to the Monitor state where the current request will now be a hit and the data sent back to the processor during the cycle in which we return to the **Monitor** state (i.e. you do not have to forward the appropriate data to the processor when bus_done is true in the WB and/or Fetch state).

2. Complete the Verilog skeleton provided to you as cache_p1.v based on the state diagram you designed for CCU. Only the state machine portion is blank and requires completion, though if you feel you need to change other signals you may. **(30 pts)**

3. The following memory references as shown in the Table 1 are included and show up in the testbench that we have provided to you. Fill in the Table 1 indicating the status of the cache upon each request and the required action taken by the CCU. **(10 pts)**

4. Compile you're your design as cache_p1.v . Then simulate it with the given test bench and verify the behavior of the cache you designed. To do so, simply use the .do file that we have provided to you as cache_p1.do. Note: Most transactions in our testbench will wait for the pr_done signal. Thus, if in the waveform you do not see new transactions happening after a certain point it is likely that you have not correctly implemented the logic for that transaction.

Table 1. Test bench memory references

Instruction	Hit	Miss	Dirty	CCU Transaction
LW @1				
SW @9, data=12				
SW @9, data=13				
SW @1, data=14				
LW@9				
LW@8				
SW@4, data=17				
LW@9				
LW@13				

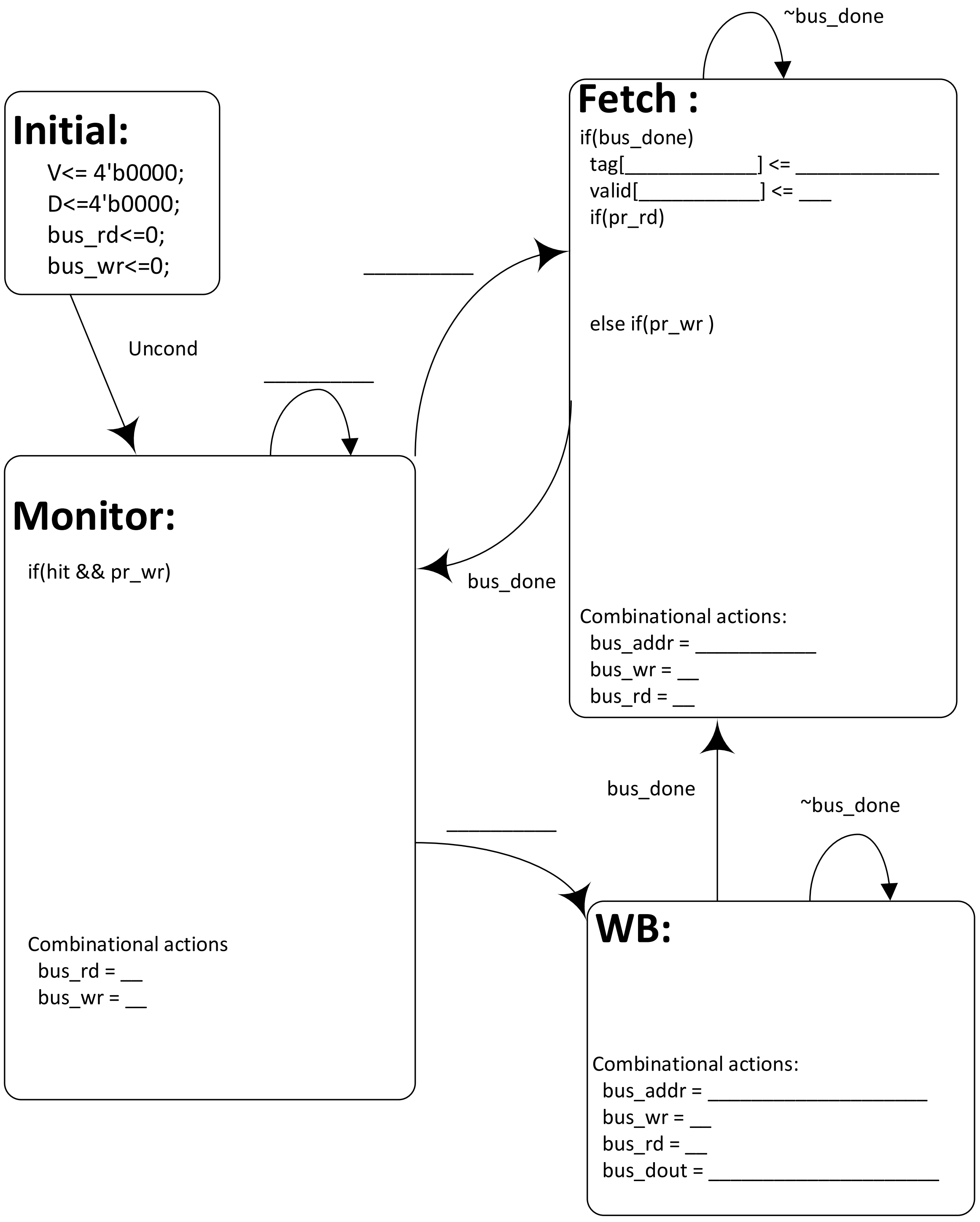
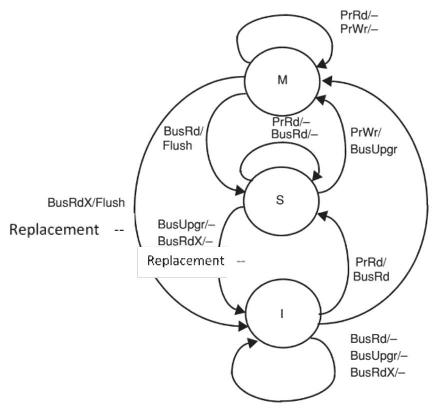


Fig. 3: CCU State Diagram

3.2 Part 2: Cache coherency protocol (60 pts)

In order to maintain a coherent memory system, we now enhance our cache control unit to implement an MSI cache coherency protocol. The MSI protocol and updated CCU diagram are shown in the Fig. 4 and Fig. 5, respectively.



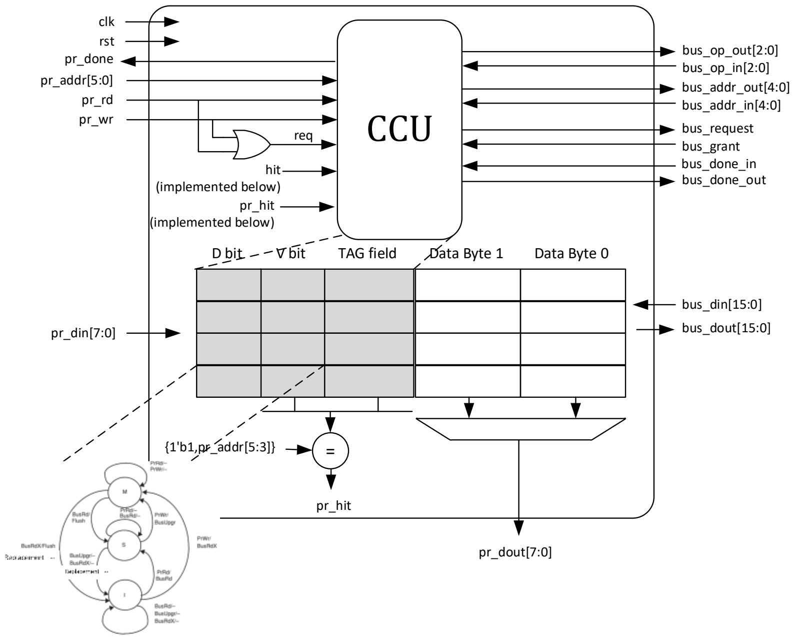


Fig. 4: MSI protocol state diagram **Fig. 5: MSI Finite State Machine Block Diagram**

Now we will have both an input and output version of signals now referred to as "bus_addr_in" and "bus_addr_out". In addition, we add 3-bit bus operation input/outputs to indicate the kind of bus operation (BusRd, BusRdX, Flush, BusUpgr, None} being perform by remote caches (bus_op_in) and by this local cache (bus_op_out).

Note that the MSI state diagram is PER cache block. We will still need the overall state machine developed in part 1 to control the operation sequence of WB, fetching, etc. But then for each cache block we will maintain its M, S, I state by using the Dirty and Valid bits. To do this we will use a 2-bit state encoding of the Dirty and Valid bits. (i.e. M = {D,V} = 11; S = {D,V} = 01; and I = {D, V} = 0,0). **<u>However, we will now call these 2-bits "msi_state" rather than dirty and valid.</u>**

Since we may receive requests from both the processor and the bus at the same time we will **CHOOSE to give priority to bus requests** over processor requests. Thus, if the bus operation is anything other than None you should handle it and ignore processor read and write requests until the bus operation is complete. To handle bus operations you will need to index the desired cache block using the bus address (in) and then compare the tags based on the desired bus address (in), as well as ensuring the block's state is valid. Our code skeleton will implement a **pr_hit** and **bus_hit** signal that you can use to help you. Your main task is to fill in the operations to be performed in each state

The **Monitor** state allows us to handle processor read and write hits and incoming BusUpgrades (i.e. bus requests to invalidate a cache block). If an incoming bus request requires us to flush modified data we can move to the **Flush** state (where we assume we can flush our cache data to the bus for a single cycle (setting the bus_op_out to BusFlush) and asserting the bus_done_out signal in the **Flush** State so that we stay for only one cycle (we mave modified the memory to write in a single cycle). **In both of these states (Monitor and Flush) the processor may still make a new request that causes a cache miss causing us to request the bus**. This request signal can be a Mealy-style output of the **Monitor** and **Flush** states. **We should stay in one of these two states until a bus_grant is given**. From there, we may need to evict the current block, writing it back if modified and then moving to the appropriate state to perform the desired bus operation. If we need to perform a BusUpgrade to invalidate others' version of a cache block, we assume that it can be done in a **single cycle** once the bus is granted (meaning we only need to stay in the BusUpgr state for 1 cycle and then return to Monitor). However, for BusRd and BusRdX we must wait until the bus_done_in signal is asserted before returning to Monitor.

Coherency operations are shown the Table 2 along with their encoding used in the skeleton provided to you. These transactions are used for **bus_op_in**, and **bus_op_out** signals.

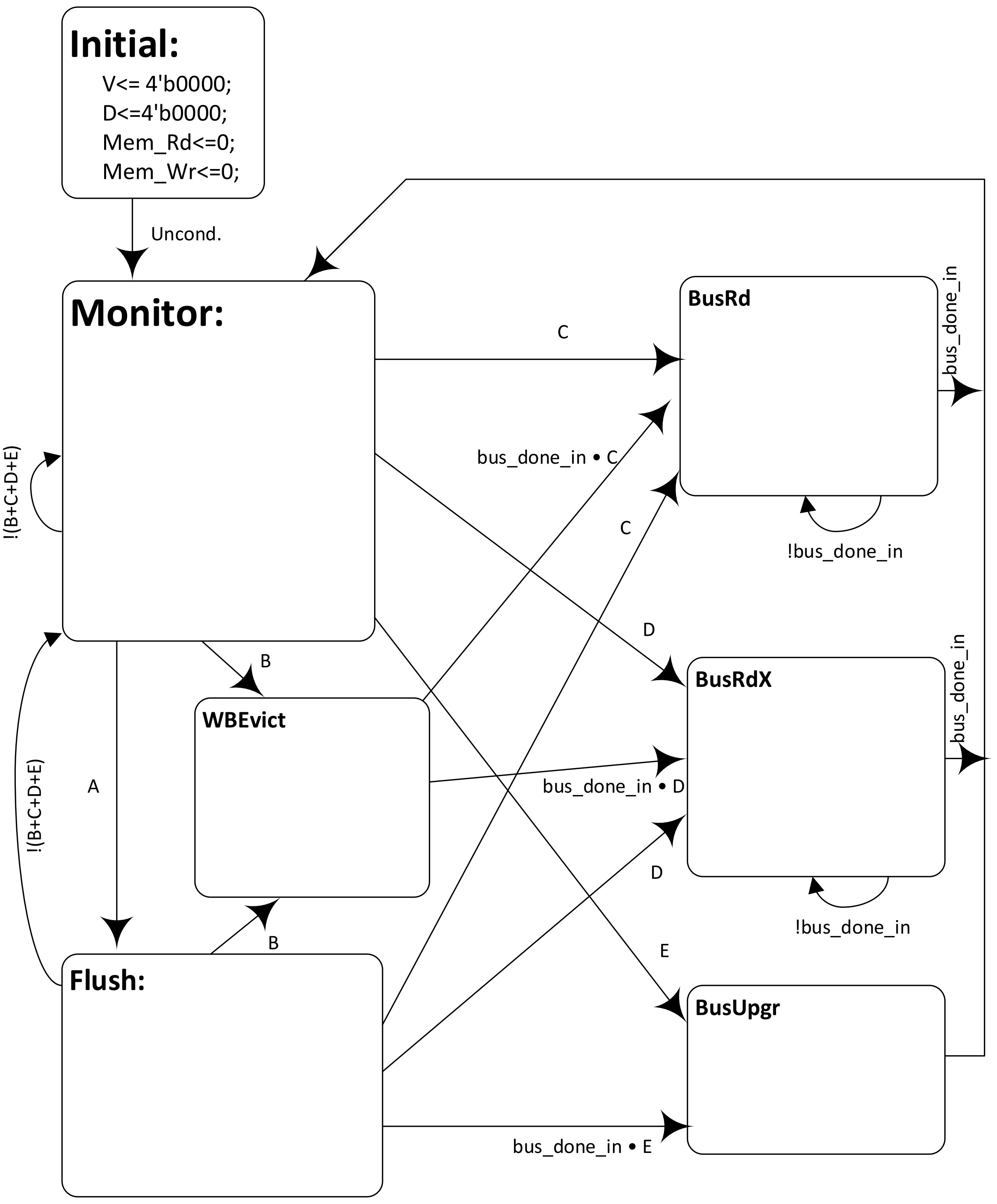
Table 2. Coherency Transactions

Coherency Transactions	Description	Encoding
None	Nothing	3’b000
BusRd	Read request for a block	3’b001
BusUpgr	Invalid other copies	3’b010
Flush	Our cache flushes/writesback a block on the bus	3’b011
BusRdX	Read block and invalidate other copies	3’b100

A state diagram for your controller is shown on the next page. You can use it as scratch work to plan what actions need to be taken in each step, but writing a full, exhaustive description of what to do in each state is likely less helpful and you can just pull up the code skeleton.

In the diagram below we define certain conditions (A, B, C, D, etc.) that should be used to transition to a new state to make the state diagram appear uncluttered. In the skeleton file we have defined signals with more meaningful names like `needToServiceBusReq` or `startBusRd` that you will need to complete. We use those signals to provide you some of the next state logic in the state machine implementation. You will need to add other actions to the state, but please read over the skeleton code and this handout a few times to start to see the connections before you begin coding in earnest.

You are free to alter the approach we've given in the skeleton as long as you implement a general design that works for any sequence of transactions. So if you don't like the way we've started the skeleton code, you can alter it.



Initial:
V<= 4'b0000;
D<=4'b0000;
Mem_Rd<=0;
Mem_Wr<=0;
Uncond.
Monitor:
!(B+C+D+E)
C
bus_done_in • C
C
D
B
A
!(B+C+D+E)
WBEvict
bus_done_in • D
D
B
E
Flush:
bus_done_in • E
BusRd
bus_done_in
!bus_done_in
BusRdX
bus_done_in
!bus_done_in
BusUpgr

Steps required to be done for Part2:

1. Complete the Verilog skeleton provided to you as cache_msi.v. Update the CCU state machine in the previous part so your cache supports MSI protocol (msi_state rather than valid and dirty bits). **(50 pts)**
2. The following memory references as shown in the following are included in the testbench that we have provided to you. Fill in the table indicating status of the each cache block upon each request and the required action to be taken on the bus. **(10 pts)**

Transaction #	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Instruction by Local Core	LW @1	SW @9	–	–	SW @9	SW @1	–
BUS_Req_in by a Remote Core	–	–	BusRd	BusRd	–	–	BusRdx
BUS_addr_in (Block addr) {byte addresses}	–	–	0 {0,1}	8 {8,9}	–	–	8 {8, 9}
Local Cache BLOCK 0, MSI							
Local Cache BLOCK 1, MSI							
Local Cache BLOCK 2, MSI							
Local Cache BLOCK 3, MSI							
BUS_Req_out							

Transaction #	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Instruction	–	LW @9	LW @9	–	LW @3	LW @2	–	–
BUS_Req_in	BusRdx	–	–	BusRdx	–	–	BusRd	BusUpgr
BUS_addr_in (Block addr) {byte addresses}	0 {0,1}	–	–	8 {8,9}	–	–	2 {2, 3}	2 {2,3}
Local Cache BLOCK 0, MSI								
Local Cache BLOCK 1, MSI								
Local Cache BLOCK 2, MSI								
Local Cache BLOCK 3, MSI								
BUS_Req_out								

3. Compile your design as cache_msi.v. Then simulate it with the given test bench and verify the behavior of the cache you designed. To do so, simply use the .do file that we have provided to you as cache_p2.do. You may (should) add more test cases to the end of the testbench. Again note that our testbench will not go on to another transaction until it verifies a previous transaction is correct (usually by looking at cvalid or the internal block state to check if it is the expected MSI value). If your testbench is not producing new transactions it is likely due to an error in your logic. We have added an integer counter to the waveform (signal i) so you can track which transaction the testbench is currently testing.