Lab 5 – Cache Controller with Coherency

❖ Introduction

In the shared memory multi-core systems, one the main challenges is to keep the memory system coherent among all the cores within the MPSoC[1]. As a result, diverse types of cache coherency protocols have been introduced to maintain coherency in the memory system in the so-called devices. 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.

❖ 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 64B. The internal data bus of the processor is 8 bits wide. The cache block size is 2-bytes (16-bits = a "word") 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 words (64 bytes). Cache organization has been demonstrated in Fig. 2.

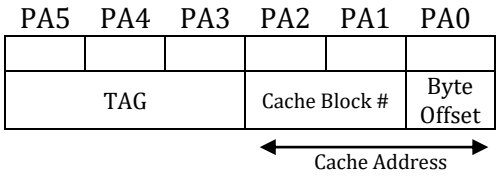


Fig. 1: Physical and Cache Address Format

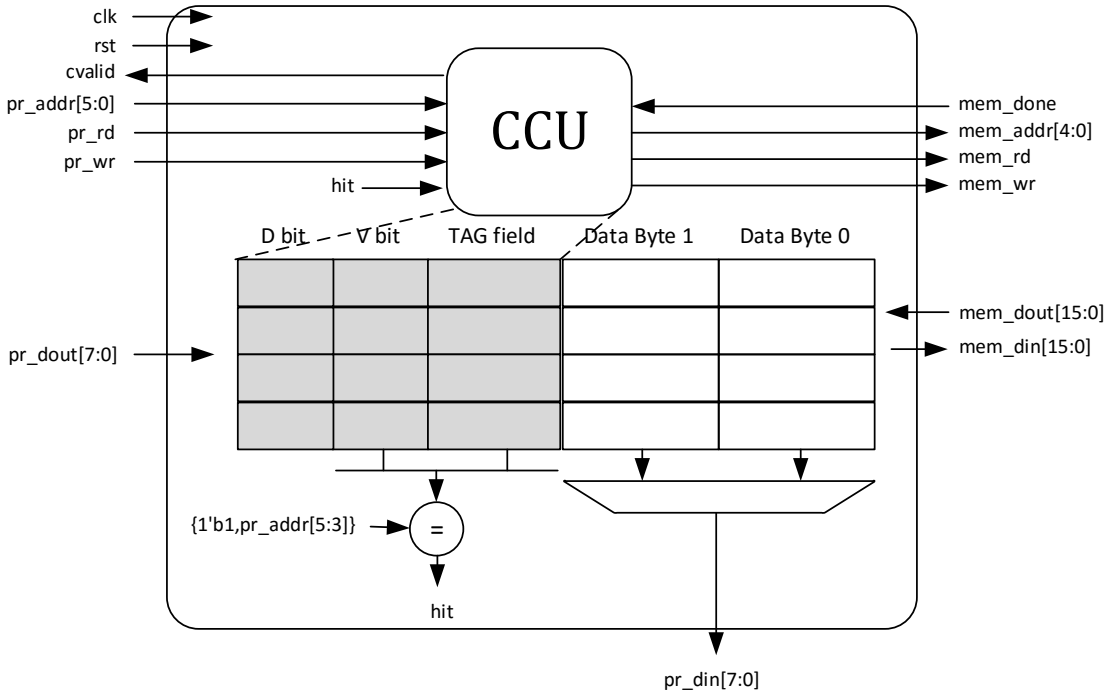


Fig. 2: Direct-mapped Cache Organization

[1] Multi-Core System on a Chip

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

pr_dout[7:0]: Data provided by the processor to be written in the cache.

pr_din[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.

cvalid: 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.

mem_dout[15:0]: Full 2-byte cache block provided by the memory to be written in cache as a result of a cache miss.

mem_addr[5:0]: Word (2-byte) address provided to memory by cache in case of a cache miss (to be used for writeback or fetch of a block)

mem_rd: Memory read request by cache in case of a read miss.

mem_wr: Memory write request by cache in case of a replacement.

mem_din[15:0]: Full 2-byte cache block provided by the cache to be written in the main memory in case of a writeback.

mem_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 is in charge of coordination between processor and main memory. If a block is missed in the cache, the CCU will request the block from the main memory and waits until memory provides the data to the cache. When the memory finishes the transaction requested by the core, it sets mem_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 mem_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_{hex}, for example: Mem[11]=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 it 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.

   StartProcess:

   We want to support a **single-clock read/write on hits**. Thus we need to handle hits completely in this state. 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, assume Mealy style operation and use this state to initiate the block writeback or fetch. To be more specific:

   - On a read hit, we can read the data immediately. The CCU need do nothing special.
   - On a write hit, we perform the write on the specified **byte** of the cache block and update any necessary state.
   - On a miss we must check if the current block in cache is dirty and write it back if so. We can initiate that operation immediately and then move to WaitForFlush. If the block is not dirty we can start the fetching of the desired block and then move to WaitForFetch.

   WaitForFlush:

   In this state, we wait for the memory operation to complete. However, once complete we need to start fetching the desired block from memory. We can start that operation in a Mealy fashion when we see the memory is about to complete. Then we can move to WaitForFetch to wait for completion of the fetch operation.

   WaitForFetch:

   In this state, we again use a Mealy style approach to complete the fetch operation by writing the appropriate data into the cache when we see the memory data is available for reading. If the processor intended to write, be sure that specific byte is placed alongside the other byte read from memory. Also be sure to update the valid, tag, and dirty bits.

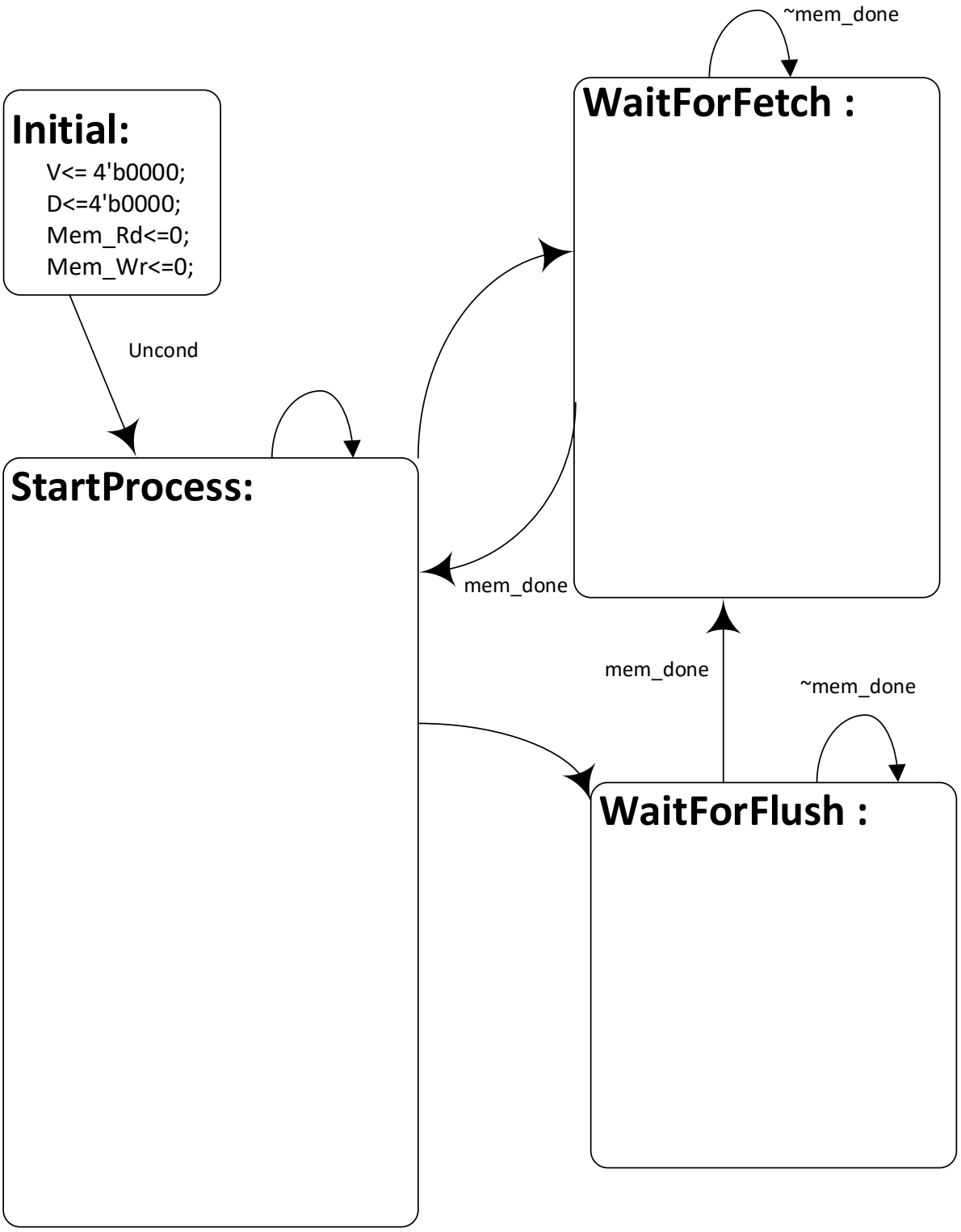


Fig. 3: CCU State Diagram

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. **(15 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. **(5 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 transaction in our testbench will wait for the cvalid signal. So 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. **(10 pts)**

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				

Part 2: Cache coherency protocol (50 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.

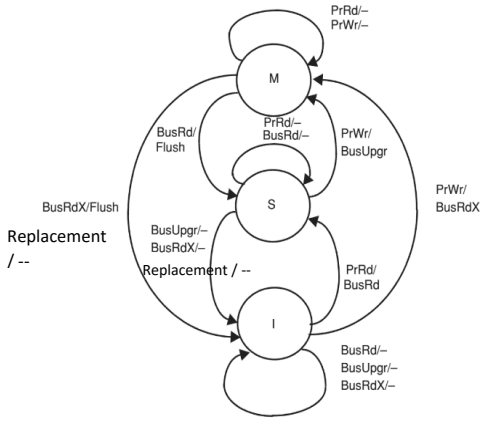


Fig. 4: MSI protocol state diagram

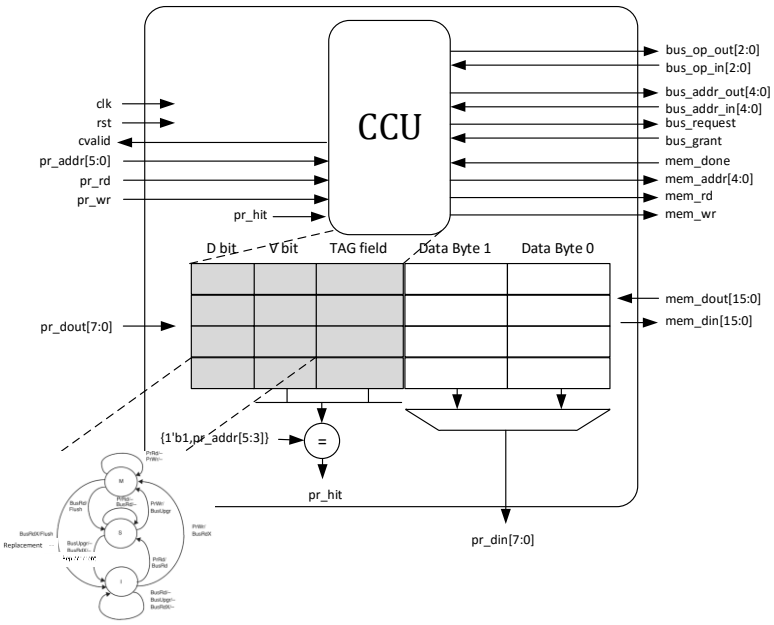


Fig. 5: MSI Finite State Machine Block Diagram

Now some of the memory signals (namely the address lines) will be 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 flushing, fetching, etc. But then for each

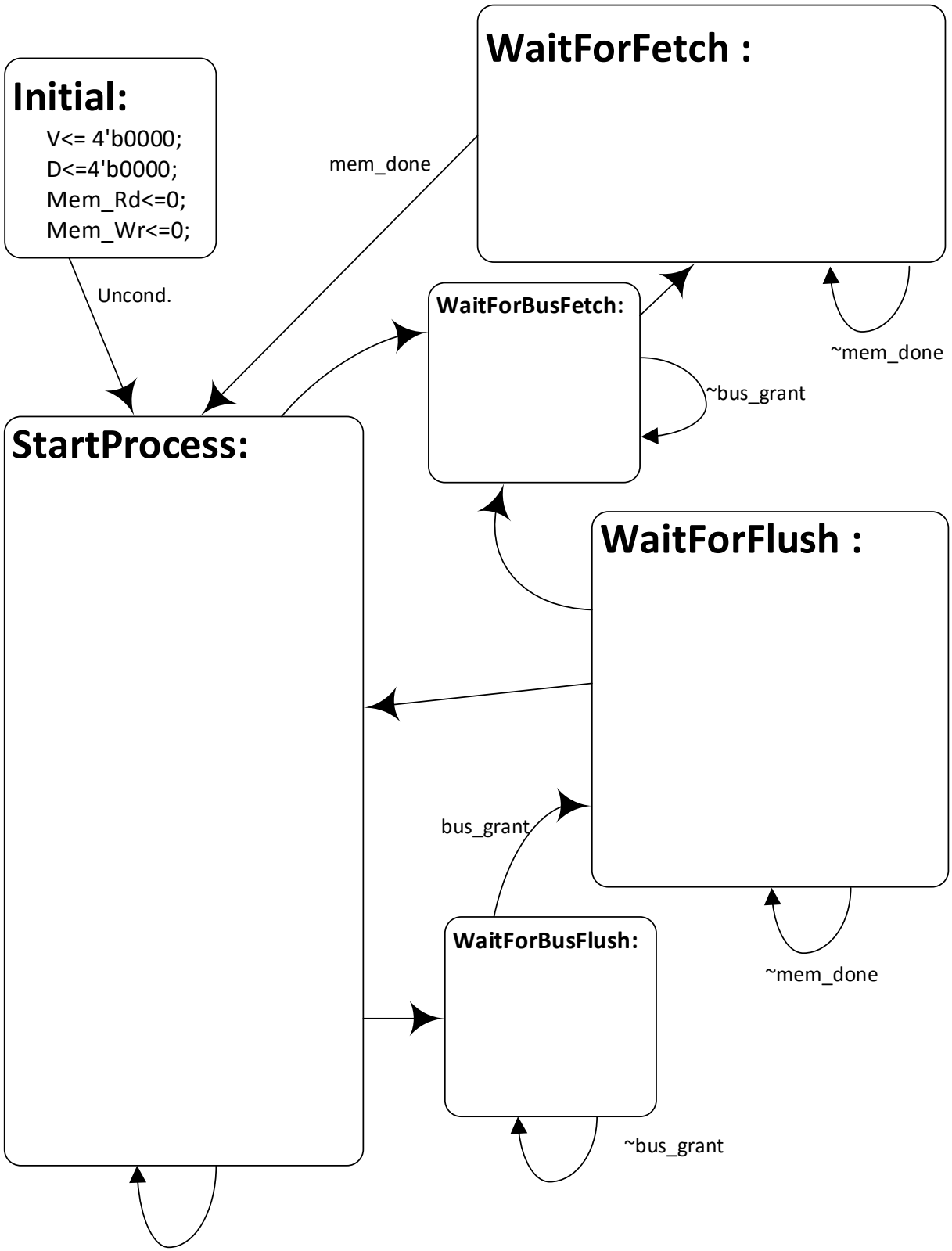
cache block we will maintain its M, S, I state by using the Dirty and Valid bits. To do this we will use a 2-bit state encoding of the Dirty and Valid bits. (i.e. M = {D,V} = 11; S = {D,V} = 01; and I = {D, V} = 0,0). **However, we will now call these 2-bits "msi_state" rather than dirty and valid.** In addition, since the bus may be busy we need to request the bus before we output onto it. Thus we will add 2 states (called WaitForBusFlush and WaitForBusFetch) to our overall state machine. These states assume you have asserted 'bus_request' and are used to wait until 'bus_grant' is returned as a 1. (Note: Our testbench is "easy" on you in that it always asserts bus_grant…but you might want to test your design by deasserting bus_grant in the testbench and ensure your cache controller waits until the bus_grant is asserted.

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	Supply a block on the Shared BUS	3'b011
BusRdX	Read block and invalidate other copies	3'b100

We will now need to handle cache operations for both the processor and bus. We will not make you do so simultaneously but instead assume any bus request has priority 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 (again use a Mealy approach that allows you to update values in a state only when a signal like 'mem_done' or 'bus_grant' are asserted.


Initial:
V<= 4'b0000;
D<=4'b0000;
Mem_Rd<=0;
Mem_Wr<=0;
Uncond.
WaitForFetch :
mem_done
~mem_done
WaitForBusFetch:
~bus_grant
StartProcess:
WaitForFlush :
bus_grant
~mem_done
WaitForBusFlush:
~bus_grant

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). Only the state machine portion is blank and requires completion, though if you feel you need to change other signals you may. **(30 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. **(5 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}	
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.

❖ Review Questions (10 pts)

1- Is it beneficial to design CCU as a Mealy or Moore machine in the cache design? Explain your reasons in details. **(5 pts)**

2- Are the mutual exclusive and all-inclusive properties for FSMs violated in the state diagram of Fig. 4a elaborating MSI protocol? Explain your answer in details. **(5 pts)**