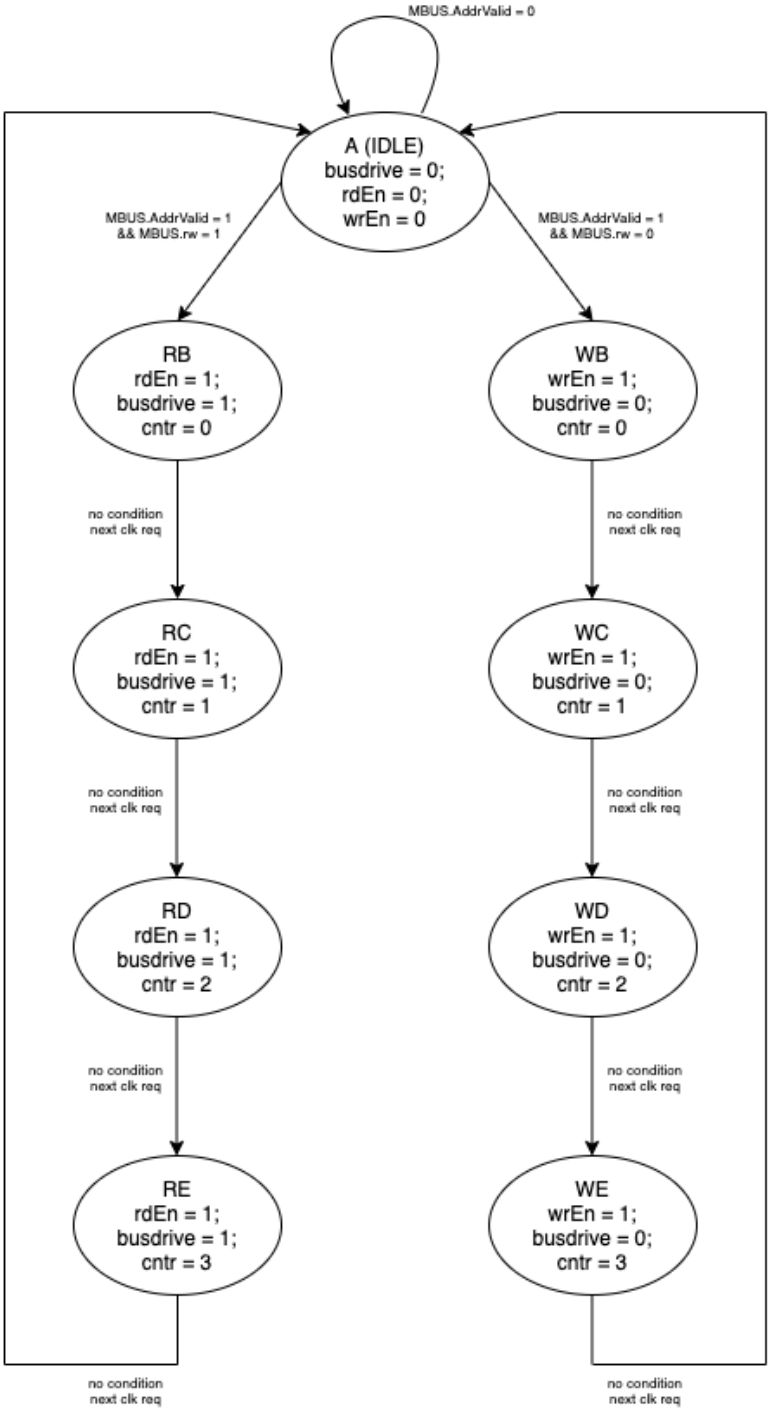
ECE 571 - Fall 21 - Homework 3

Yashodhan Wagle PSU ID: 930027080

Design Report:
Memory Controller Implementation:



FSM implementation:

- A is the idle state where it waits for the MBUS.AddrValid and MBUS.rw. The AddrValid signal tells if the data asserted on the bus is valid or not. The rw signal decides whether the fsm will enter the write or read mode.
- Once the FSM is in the read or write states, the rdEn or wrEn are asserted respectively and the busdrive signal depending on if it is reads or writes.
- Each state (represented by states B, C, D, E) in the write and read side of the FSM writes or reads to the next data in a burst of 4.
- There is no condition to cycle between the B, C, D, E as it is not bound by any condition but the next cycle is required to write or read to the next data.

Memory Array Implementation:

```
parameter   MEMADDRBITS = 12;               // memory array address bits
parameter   MEMSIZE = 2**MEMADDRBITS;       // memory size based on memory address

// declare the memory array
logic [BUSWIDTH-1:0]    M[MEMSIZE]; // memory array. Only one page declared
```

As we can see from the above code, even though the buswidth is 16 bits, the memory array implemented is only one page (12 bits) and thus any references to pages other than PAGE are illegal. This needs to be implemented in the code.

Memory Initialization:

```
initial begin: init_array
   // clear the memory locations
   foreach (M[i]) begin
       M[i] = '0;
   end
end: init_array
```

Code for initializing memory to a 0 array.

For Read:

```
assign MBUS.AddrData = ((page == PAGE) && busdrive) ? DataOut : 'z;  // data from the
memory array (reads)
assign addr = baseaddr + cntr;

// read a location from memory.
```

```
always_comb begin
   if (rdEn == 1'b1)
       DataOut = M[addr];
end
```

From the above code we can determine for reads, the MBUS.AddrData is asserted with a valid address in the state A and DataOut is connected to MBUS.AddeData if the requested data is from PAGE. addr is determined by the baseaddr provided by the CPU and cntr shows the burst of data so it increments in every consecutive state starting from 0 to 3. This addr is provided to the memory array M and thus in return gets mapped to DataOut. We also need to assert the rdEn for the comb block to work and busdrive since the memory is the one providing valid data which is done in the FSM.

For Write:

```
assign DataIn = MBUS.AddrData;      // data to the memory array (writes)
assign addr = baseaddr + cntr;

// write a location in memory
// We used an always block instead of always_ff because we are using an initial
// block to initialize the memory and we can't drive M[] from two procedural blocks
// using always_ff
always @(posedge MBUS.clk) begin
   if (wrEn == 1'b1) begin
       M[addr] <= DataIn;
   end
end // write a location in memory
```

Similarly for the write, the addr is addition of baseaddr provided by the BUS and cntr increments for every cycle. We write to the memory on every other cycle thus a sequential block is used for the same. Similarly we need to assert wrEn and deassert (tristate output) busdrive since CPU is providing valid data which is done in FSM.

Processor Implementation:

The processor here functions as a testbench that performs reads and writes with task constructs and providing test cases for the same.

Write Requests:

```
// bus driver for AddrData bus
assign MBUS.AddrData = busdrive ? ad : 'z;

@(posedge clk)          //stateA
       busdrive<=1;
       ad[15:12]<=page;
       ad[11:0]<=baseaddr;
@(posedge clk);
       MBUS.AddrValid<=0;
       ad<=data0;
```

State A needs to assert busdrive to further assert a valid address and valid address bit. The subsequent data to be written are taken into a buffer and labelled from data0 to data3 (State WB is included). The CPU will assert the busdrive signal for the whole request. The 4 MSB bits are used to determine the page and others are baseaddr.

Read Requests:

```
// drive the address onto AddrData
@(posedge clk);  // state A in the timing diagram
busdrive <= 1;
MBUS.AddrValid <= 1;
MBUS.rw <= 1;

@(posedge clk); // state B, C, D, and E in the timing diagram
MBUS.AddrValid <= 0;
MBUS.rw <= 0;
busdrive <= 0;
```

Similarly for reads, the signals that are to be asserted are shown above. In reads, the CPU drives the busdrive only for A state since it won't be providing valid data on the bus after the A state (i.e. RB, RC, RD, RE). The address decoding is the same for reads as well.

Testbench Implementation:

Both read and write tasks contain a display function that determines the correctness of the memory controller on the CPU side by performing the same function in the task itself.
For reads, the function compares data of CPU side memory (TBMem) to that of the memory array (M) on the memory controller side.
For writes, the function checks if the provided page is valid and then writes it to CPU side memory (TBMem) to maintain a copy of the correct memory array on CPU side.

Testcase Implementation:

```
$display("Writing to the first 16 locations of the memory");
```

Implementing 4 CPU_wrReqs to given memory locations which result in 16 writes since a burst of 4 is implemented.

```
$display("\nReading them from the memory");
```

Implementing 4 CPU_rdReqs to random memory locations which have not been written to yet. Thus the expected output is 0.

```
$display("writing to the top of the memory and read it back");
```

Implementing a CPU_wrReq to a location and implementing a CPU_rdReq to the same location. The expected output is the same as written during the prior.

```
$display("Write to the top of the memory and read it back using wraparound
technique");
```

This is the same instr implementation as before but here the baseaddr is not aligned (4 byte aligned). This should not affect the output since the use of %4 function makes it aligned. The expected output is the same as written during the prior.

```
$display("write to the first 4 locations at the wrong page");
```

Implementing a CPU_wrReq to a location on the wrong page (not MEMPAGE1) and implementing a CPU_rdReq to the same baseaddr but to MEMPAGE1. This invalidates the CPU_wrReq instruction since the only one page is implemented in the memory array which is assumed to be MEMPAGE1. Thus the expected output should not be what was written just before but what already is in the memory.