Lab III: Memory Access Interface

Group: CS3710

Members: Steen Sia, Stephan Stankovic, Jon Pilling, Jeremy Wu

Abstract:

The design of the memory access interface was not too complicated in comparison to the previous implementations of the ALU and Regfile. In the process of creating the memory, our team opted to utilize the template that verilog has already implemented for us. The template we decided to go with was the true dual port ram dual clock.

```
// Quartus Prime Verilog Template
// True Dual Port RAM with dual clocks

module true_dual_port_ram_dual_clock
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk_a, clk_b,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	always @ (posedge clk_a)
	begin
		// Port A
		if (we_a)
		begin
			ram[addr_a] <= data_a;
			q_a <= data_a;
		end
		else
		begin
			q_a <= ram[addr_a];
		end
	end

	always @ (posedge clk_b)
	begin
		// Port B
		if (we_b)
		begin
			ram[addr_b] <= data_b;
			q_b <= data_b;
		end
		else
		begin
			q_b <= ram[addr_b];
		end
	end

endmodule
```

True Dual Port Ram Dual Clock

One of the changes we made from the template is tweaking the data width and address width from 8 and 6, to 16 and 10 respectively. Now having established our memory block size, we began to create a sequence of continuous assignments at every positive edge of the clock in order to test that memory is being written on to registers. Furthermore, we want to ensure that we can read from those registers as well. In reference to the website, "Modeling Memories and FSM…," we used the simply memory example where it reads from a text file. As shown below, the statement "$readmemb("memory.txt", ram)", is a function that takes in the file to be read and the register that acts as a block of memory.

```
// Quartus Prime Verilog Template
// True Dual Port RAM with dual clocks
module memory// true_dual_port_ram_single_clock
#(parameter DATA_WIDTH=16, parameter ADDR_WIDTH=10)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	initial
	begin
		$readmemb("memory.txt", ram);
	end

	always @ (posedge clk)
	begin
		if (we_a && we_b)
			begin
				if (addr_a == addr_b)
					begin
						ram[addr_a] <= data_a;
						q_a <= data_a;
					end
				else
					begin
						ram[addr_a] <= data_a;
						ram[addr_b] <= data_b;
						q_a <= data_a;
						q_b <= data_b;
					end
			end
		else if (we_a)
			begin
				ram[addr_a] <= data_a;
				q_a <= data_a;
			end
		else if (we_b)
			begin
				ram[addr_b] <= data_b;
				q_b <= data_b;
			end
		else
		begin
			q_a <= ram[addr_a];
			q_b <= ram[addr_b];
		end
	end

endmodule
```

Current Version of True Dual Port RAM with Dual Clocks

Inside this our memory.txt file, we have written random binary values to be stored in the first couple addresses, and used a jump command (@ 0x64) to start storing values after address 100.

```
1100111111001111
1010101011001111
@64
01011010011001111
1111111111001111
```

Memory.txt file

Before moving on, the code above essentially takes in two enable signals, write enable A and write enable B. We have four different conditions in which the following can happen: both write enables are active, write enable A is active and B is inactive, write enable A is inactive and B is active, and neither enable signals are active. For the first condition, we check if both write enables are active. In this case, a check must be made where if the addresses to be written into A and B are the same, we simply choose to write into the address in A and ignore B. Otherwise, unique addresses are determined and we shall put values in the registers A and B respectively. Second, our condition takes in write enable A and puts the value into the A register upon the next positive edge. Third, it operates the same as the second condition, but write enable B is checked and value to be written is into the B register. Lastly, if no enable signals were given, we simply retain the previous values of registers A and B. The picture below confirms that values have been set to the appropriate registers. We see that the mif file has created 2 blocks of RAM which is 512 x 2 = 1024 words. In the text, we wanted to put values in the first couple of registers, jump at address 100 in hex and write a couple of values after that. We can see below that 0, 1, 100, and 101 have values written respectively.

memory.ram0_memory_e411fb78.hdl.mif

```
-- begin_signature
-- memory
-- end_signature
WIDTH=16;
DEPTH=1024;

ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;

CONTENT BEGIN
    1023 :  XXXXXXXXXXXXXXXX;
    1022 :  XXXXXXXXXXXXXXXX;
    1021 :  XXXXXXXXXXXXXXXX;
    1020 :  XXXXXXXXXXXXXXXX;
    1019 :  XXXXXXXXXXXXXXXX;
    1018 :  XXXXXXXXXXXXXXXX;
    1017 :  XXXXXXXXXXXXXXXX;
    1016 :  XXXXXXXXXXXXXXXX;
    1015 :  XXXXXXXXXXXXXXXX;
    1014 :  XXXXXXXXXXXXXXXX;
    1013 :  XXXXXXXXXXXXXXXX;
    1012 :  XXXXXXXXXXXXXXXX;
    1011 :  XXXXXXXXXXXXXXXX;
    1010 :  XXXXXXXXXXXXXXXX;
    1009 :  XXXXXXXXXXXXXXXX;
    1008 :  XXXXXXXXXXXXXXXX;
    1007 :  XXXXXXXXXXXXXXXX;
    1006 :  XXXXXXXXXXXXXXXX;
    1005 :  XXXXXXXXXXXXXXXX;
    1004 :  XXXXXXXXXXXXXXXX;
    1003 :  XXXXXXXXXXXXXXXX;
    1002 :  XXXXXXXXXXXXXXXX;
    1001 :  XXXXXXXXXXXXXXXX;
    1000 :  XXXXXXXXXXXXXXXX;
    999 :   XXXXXXXXXXXXXXXX;
    998 :   XXXXXXXXXXXXXXXX;
    997 :   XXXXXXXXXXXXXXXX;
    996 :   XXXXXXXXXXXXXXXX;
```

Memory.mif file creating 1024 words

```
    9 : XXXXXXXXXXXXXXXX;
    8 : XXXXXXXXXXXXXXXX;
    7 : XXXXXXXXXXXXXXXX;
    6 : XXXXXXXXXXXXXXXX;
    5 : XXXXXXXXXXXXXXXX;
    4 : XXXXXXXXXXXXXXXX;
    3 : XXXXXXXXXXXXXXXX;
    2 : XXXXXXXXXXXXXXXX;
    1 : 0001000100010001;
    0 : 0001000100010001;
END;
```

Register 0 and 1 with values

```
   101 :   0001000100010001;
   100 :   0001000100010001;
   99 :    XXXXXXXXXXXXXXXX;
   98 :    XXXXXXXXXXXXXXXX;
   97 :    XXXXXXXXXXXXXXXX;
   96 :    XXXXXXXXXXXXXXXX;
   95 :    XXXXXXXXXXXXXXXX;
   94 :    XXXXXXXXXXXXXXXX;
   93 :    XXXXXXXXXXXXXXXX;
   92 :    XXXXXXXXXXXXXXXX;
   91 :    XXXXXXXXXXXXXXXX;
   90 :    XXXXXXXXXXXXXXXX;
```

Register 100 and 101 with values

Ultimately, our memory finite state machine emulates this procedure by covering all four cases mentioned above and displaying them into the hex 7-segment.

```
module memory_fsm(out0, out1, out2, out3, out4, out5);

reg clk, reset;


output wire [6:0] out0, out1, out2, out3, out4, out5;

reg we_a, we_b;
reg [15:0] data_a, data_b;
reg [9:0] addr_a, addr_b;
reg [2:0] state;

wire [15:0] q_a, q_b;

parameter resets = 3'b000;
parameter first = 3'b001;
parameter second = 3'b010;
parameter third = 3'b011;
parameter fourth = 3'b100;

memory m(
   .clk(clk),
   .we_a(we_a),
   .we_b(we_b),
   .addr_a(addr_a),
   .addr_b(addr_b),
   .data_a(data_a),
   .data_b(data_b),
   .q_a(q_a),
   .q_b(q_b)
);

initial
   begin
      clk = 1;
      reset = 1;
      #20
      reset = 0;
      #200
      $finish;
   end
```

Memory_FSM 1 (Inputs, Outputs, Registers)

```
always@ (state)
begin
   case(state)
      resets:
         begin
            we_a = 0;
            we_b = 0;
            addr_a = 10'b0;
            addr_b = 10'b0;
            data_a = 16'b0;
            data_b = 16'b0;

         end

      first:
         begin
            we_a = 1;
            we_b = 0;
            addr_a = 10'b0;
            addr_b = 10'b0;
            data_a = 16'b0000_0000_0000_0001;
            data_b = 16'b0;

         end

      second:
         begin
            we_a = 0;
            we_b = 1;
            addr_a = 10'b0;
            addr_b = 10'b00000_00001;
            data_a = 16'b0;
            data_b = 16'b0000_0000_0000_0010;

         end

      third:
         begin
            we_a = 1;
            we_b = 1;
            addr_a = 10'b00000_00000;
            addr_b = 10'b00000_00001;
            data_a = 16'b0000_0000_0000_0111;
            data_b = 16'b0000_0000_0000_1111;

         end
```

Memory_FSM 2 (Each State Specification)

```
always@ (posedge clk)
begin
   state <= resets;
   case(state)
      resets:
         if (reset)
            state <= resets;
         else
         begin
            state <= first;
         end
      first:
         if (reset)
            state <= resets;
         else
         begin
            state <= second;
         end
      second:
         if (reset)
            state <= resets;
         else
         begin
            state <= third;
         end
      third:
         if (reset)
            state <= resets;
         else
         begin
            state <= fourth;
         end
      fourth:
         if (reset)
            state <= resets;
         else
         begin
            state <= fourth;
         end

      default:
         state <= resets;
   endcase

end
```

Memory_FSM 3 (Progression of States)

```
      fourth:
         begin
            we_a = 1;
            we_b = 1;
            addr_a = 10'b00000_00000;
            addr_b = 10'b00000_00000;
            data_a = 16'b0000_0000_0000_0001;
            data_b = 16'b0000_0000_0000_1111;

         end

      default:
         begin
            we_a = 0;
            we_b = 0;
            addr_a = 16'b0;
            addr_b = 16'b0;
            data_a = 16'b0;
            data_b = 16'b0;

         end

   endcase
end

always@ (*)
   #5 clk <= ~clk;

 hexTo7Seg first0(q_a[3:0],out0);
 hexTo7Seg second0(q_a[7:4],out1);
 hexTo7Seg third0(q_a[11:8],out2);
 hexTo7Seg fourth0(q_b[3:0],out3);
 hexTo7Seg fifth0(q_b[7:4],out4);
 hexTo7Seg sixth0(q_b[11:8],out5);

endmodule
```

Memory_FSM 4(Cont. And HexTo7Seg