Laboratory 4: Memory

(This lab is partly taken from Pitchaya Sitti-Amorn, Ph.D.)

Objectives

1. Able to implement memory in HDL
2. Able to instantiate internal FPGA memory

Background

We would be soon implementing our first processor in the next Lab!. Now, you should have understanding of how to implement a FSM with Verilog. What you are missing is how to implement memory model on Verilog such that you can use it on your very first processor. We will be looking at read-only-memory (ROM), random-access-memory (RAM), and first-in-first-out (FIFO)

ROM

The first kind of memory you are going to be implement is read-only-memory (ROM). So far, you have been using only Verilog for synthesizing register (D-Flipflop). However, it's costly to implement memory using purely registers. Field-programmable gate array (FPGA) manufactures often include block of memory inside the FPGA such that you can use.
Typical ROM instantiation looks like the following

```
module rom case(
      (* synthesis , rom block = "ROM CELLXYZ01" *)
      output reg [3:0] z ,
      input wire [2:0] a); // address- 8 deep memory

always@* begin // @(a)
      case (a)
      3'b000: z = 4'b1011;
      3'b001: z = 4'b0001;
      3'b100: z = 4'b0011;
      3'b110: z = 4'b0010;
      3'b111: z = 4'b1110;
      default : z =4'b0000 ;
      endcase
end
endmodule // rom case
```

The code above would generate ROM with 8 address, each address is 4 bits.

You might notice the synthesis suggestion keyword (* synthesis, rom_block = "ROM_CELLXYZ01" *). This tell the synthesis tools to try to use the dedicated ROM inside the FPGA instead of implementing as a block of registers. The keywords may differ from

one FPGA vendor from another. Note that most synthesis tools nowadays are smart enough to detect the access pattern that you can remove that out.

Second, the ROM in is actually asynchronous. You could make it a synchronous ROM by adding a clock, i.e.

```
module rom case(
      (* synthesis , rom block = "ROM CELLXYZ01" *) input clk ,
      output reg [3:0] z ,
      input wire [2:0] a); // address- 8 deep memory
always@(posedge clk)
      begin
      case (a)
      3'b000: z = 4'b1011;
      3'b001: z = 4'b0001;
      3'b100: z = 4'b0011;
      3'b110: z = 4'b0010;
      3'b111: z = 4'b1110;
      default : z =4'b0000 ;
      endcase
end
endmodule // rom case
```

Having the data inside your program is extremely inconvenient especially if you have a large file set of data. Verilog allows you to "read" data from a file.

```
// Verilog-2001 style
// ROM module using two dimensional arrays with
// memory defined in text file with $readmemb or $readmemh
// NOTE: This style can lead to simulation/synthesis mismatch
//       if the content of data file changes after synthesis
module rom_2dimarray_initial_readmem (
  output wire [3:0] z,
  input  wire [2:0] a);
  // declares a memory rom of 8 4-bit registers.
  //The indices are 0 to 7
  (* synthesis, rom_block = "ROM_CELL XYZ01" *)
  reg    [3:0] rom[0:7];
  // NOTE:  To infer combinational logic instead of a ROM, use
  // (* synthesis, logic_block *)
  initial \$readmemb("rom.data", rom);
  assign z = rom[a];
endmodule
```

The rom.data would look like this

```
1011 // addr=0
1000 // addr=1
0000 // addr=2
1000 // addr=3
0010 // addr=4
0101 // addr=5
1111 // addr=6
1001 // addr=7
```

RAM

Another useful primitive for the FPGA is random-access-memory (RAM). Again, as in ROM case, we could have implemented RAM as a set of registers, but it's expensive and costly to do so. Typical FPGAs have dedicated area for RAMs, (BlockRAM for Xilinx, Memory Block for Altera, etc.) The following code will generate RAM with 128x8 bits.

```
module SinglePortRAM (
inout wire [7:0] d, // Data In and Out
input wire [6:0] addr , // Address
input wire oe , // Output Enable
input wire clk , we) ;
(* synthesis , ram block *)

reg [7:0] mem [127:0];

always @(posedge clk)
if(we)
      mem[addr] <= d;

assign d = oe ? mem[addr] : 8'bZ;
endmodule
```

You may see the inout port in the example. The idea is that the port can be used as both input and output (at different time). To read, you have to assign Z to the wire before reading the data. To write, just connect register to the wire. This line "assign d = oe ? mem[addr] : 8'bZ; " explains such connection.

Block RAM

As you can see that we can write a HDL code to generate registers, and we can potentially implement a memory using it. However, a FPGA has a small number of these CLB, and it is a bit over kill since these logic can be do much greater things than being just memory. So, most FPGA vendor has a specialize memory units that we can use on these FPGAs. Each vendor has a different name, but for Xilinx, we call it Block RAMs. Typically, each bRAMs has a size of 10-20Kbit depending on the FPGA, and each FPGA may have from ten to thousands of these bRAMs.

There are several ways to initiate bRAMs on Xilinx, but in general Xilinx Synthesis step can recognize if you are going to generate a bRAMs from a pattern following pattern

```
parameter RAM WIDTH = <ram width >;

parameter RAM ADDR BITS = <ram addr bits >;

reg [RAMWIDTH-1:0] <ram name> [(2**RAMADDRBITS)- 1:0]; reg [RAM WIDTH-1:0]
<output  data >;

<reg or wire> [RAMADDRBITS-1:0] <address>;

<reg or wire> [RAMWIDTH-1:0] <input data >; always @(posedge <clock>)

if (<ram enable>) begin

if (<write enable>) begin

<ram name>[<address >] <= <input data >; <output data> <= <input data >;

end else

<output data> <= <ram name>[<address >]; end
```

Note that bRAMs is not the only type of construct that the Synthezier can recoginize. There are many other types of construct. Some of these are even more complicated. So most FPGA vendors has a so called language template. For Xilinx, you can find these out in the menu by going to Tools > Language Templates. For memory, it is in Verilog > Synthesis Constructs > Coding Examples > RAM > BlockRAM. We recommend you explore these constructs.

Most FPGA has what it called Distributed RAM. This is essentially use the logic gate and registers to implement the memory. Usually, Distributed RAM is faster than bRAMs, but it is smaller. 1

Note that there is also another method for instantiating the bRAMs. This is through the use of Block Design. We recommend you take a look at this document https://www.xilinx.com/support/documentation/university/Vivado-Teaching/Digital-Design/2014x/docs-pdf/Vivado_tutorial.pdf for more information about how to use IP Integrator and Block Design.

Exercises

1. Design and build a circuit to work as a Stack (LIFO). The user can use two push buttons in order to PUSH (BTNU) or POP (BTNC). Use 8 switches on the board as a value. When a user hit a PUSH button, it will store the value from the switches to the stack. When the user hit the POP switch, it will display the value from the top of stack in the two hex displays on the left. The other hex displays is used to display the number of elements currently in the stack. The stack can keep up to 256 elements. If the stack is full, hitting the PUSH button should not do anything.

   We recommend using a PUSH button as a reset (BTND).

2. Read an input from 5 binary switches. (You are welcome to use any switch. If you have no preference, use SW[4..0].) This should give you a number ranging from 0 to 31. Use distributed memory or bRAMs as the ROM for converting binary to 2 BCD for displaying on seven-segment display. Use 5 bit binary as an address. The output can be either 2x4 BCD for applying to BCDtoSevenSegment (as used in the previous lab) or 2x8 for feeding directly to seven segment display.

   Note: You may want to initialize the memory from data. Please see the language template for more information

3. Use Block Design to create a simple calculator 4-bit calculators. You will assign 4 switches as the 4-bits input for A and another 4 switches for B. You will assign 4 push buttons to do 4 different operations.
   a. When BTNU is pushed, you will display the result of A+B in based 10 using the three 7-segment displays.
   b. When BTNL is pushed, you will display the result of A-B in based 10 using the three 7-segment displays.
   c. When BTND is pushed, you will display the result of A*B in based 10 using the three 7-segment displays.
   d. When BTNR is pushed, you will display the result of A/B in based 10 using the three 7-segment displays.

4. Please answer the following questions and submit (in PDF format) to CourseVille on Friday before 23:59 (midnight).
   a. Explain your ROM for mapping 5-bit binary to 2-digit BCDs (or 2x8 bits seven segment displays depending on your design in Exercise.2).