Bruno E. Gracia Villalobos

EE 4513

Assignment # 3

Design of a Single-Port 8B Synchronous RAM with Reset in Verilog

September 14, 2019

Introduction

This report entails the design, simulation, and synthesis of a single-port, byte-addressable, 8B synchronous Random-Access Memory (RAM) with Reset in an 8 rows X 8-Bit configuration using the Verilog HDL. The RAM takes in a 3-bit address, 8-bit data, and three control signals of 1-bit width: write-enable, read-enable, and reset. As a single port design, the output is a single 8-bit data bus.

The RAM design is developed to be of variable size and can be defined to support M rows X N width using *parameter* data types. To write to the RAM, the log2(N) sized input address bus must be defined to the desired row to modify, the N-bit data input bus with the data to write, and the write-enable control signal set high. As for reading from the RAM, the same address bus must be set accordingly as well as the read-enable control signal set high. For write and read operations, data is written and sent to the output at the active edge of the clock respectively. However, when both write and read signals are high, logic gives preference to write requests over read requests to handle the contention.

The last capability of the RAM is the reset function—the operation sets a total of M X N bits to GND at the positive edge of the clock given an active high control signal. It is important to mention the reset takes priority over read and write control signals and can impact the operation of the RAM if left at a logic high level. For this reason, careful attention must be placed to the control signals sent to the RAM.

For the design approach, the RAM is developed at RTL to define the sequential logic with a procedural block. In addition, the design encapsulates a single module and therefore does not specify a top-down or bottom-up approach. The subsequent sections entail of the design process, simulation and synthesis of the RAM.

Procedure

```
module MEMORY_SYNC
#(parameter WIDTH = 8, parameter ROWS = 8) (
input CLK,

input [WIDTH-1:0] write_data,       //the data to write to memory address
input [$clog2(WIDTH)-1:0] address,  //the address in memory to write or read data

input read_enable,                  //control signal to enable read of data
input write_enable,                 //ctrl signal to enable write of data
input reset,                        //ctrl signal to enable reset of data

output reg [WIDTH-1:0] read_data    //the data read from memory
);
```

Figure 1: 8B RAM portlist

The first step in creating the 8B Synchronous RAM is to setup the portlist, shown in Figure 1. As explained previously, a single-port design transfers 6 inputs into a single output register with a width of 8 bits. Little endian is used for the write-data and address wire buses and the read-data register.

Both write_data and read_data ports are given a size of WIDTH using little endian. To define the address size, the Verilog system function *$clog2* is used to compute number of bits needed to

represent the number of ROWS. The control signals read_enable, write_enable, and reset are all given a width of 1-bit.

To make the variable size of the RAM, two *parameters* are defined: WIDTH and ROWS. WIDTH defines the input and output data width and ROWS defines the number of entries in the RAM. For an 8B configuration with byte-addressable memory, WIDTH and ROWS are defined to be both 8 to give a total number of 8 rows X 8-bits = 8B. in Figure 2, the RAM is defined as a reg array with a little-endian data but big endian addressing—this is to give the design a more intuitive approach when indexing: memory[0] is the first entry, memory[1] is the second entry, and so on.

```
reg [WIDTH-1:0] memory [0:ROWS-1]; //create array of # ROWS with WIDTH sized entries
```

Figure 2: RAM as a reg array

```
if(reset) begin //set all rows of memory to GND
    for(i=0; i<ROWS; i = i+1) begin
        memory[i] = 0;
    end
end
```

Figure 3: Reset for loop

Now with the storage capacity setup, the next step is to define a procedural *always* block with a positive edge CLK as the sensitivity list. Within the block, the reset signal is assessed first to give it priority and executes a *for* loop to set every entry in the RAM to ground as shown in Figure 3. Both read_enable and write_enable are handled next.

Three cascading conditional statements are used to define the behavior of the read and write control signals. To prevent a contention case as mentioned earlier, the first statement gives the read operation priority when both control signals are high, and the output register is set to the RAM contents at the specified address. If this is not the case, the next statement checks the assertion of the read_enable signal on its own to carry out a read operation. Finally, if the above are not true, then logically a write operation is called, but we cannot assume this as the write_enable control signal can have a value of X or Z and therefore is checked for assertion. Figure 4 displays the cascading conditional statements.

```
if(read_enable & write_enable) begin
    read_data = memory[address]; //give priority to read over write
end
else if(read_enable) begin
    read_data = memory[address];
end
else if(write_enable) begin
    memory[address] = write_data;
end
```

Figure 4: Cascading conditional statements

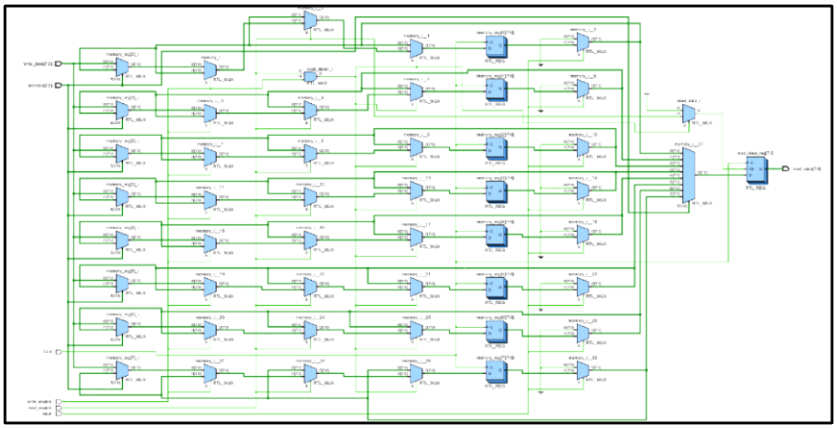


Figure 5: RAM elaborated design at RTL

Once the design is built, the design is elaborated to verify the desired logic is interpreted by the Verilog compiler. Figure 5 shows the RAM schematic at RTL. Surprisingly, the design is inefficient due to the large amount of MUXes and the lack of an RTL_RAM inference.

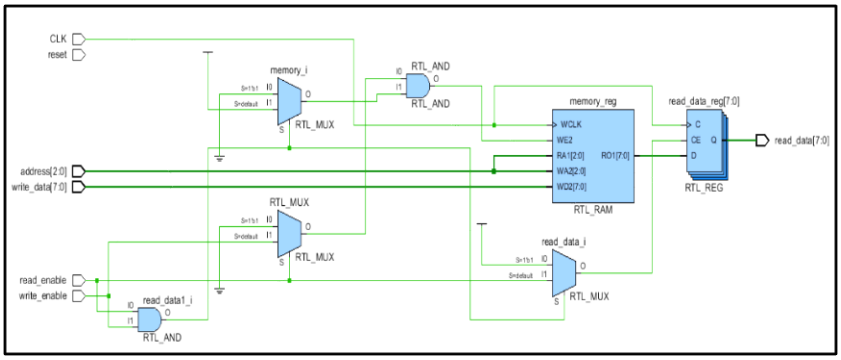


Figure 6: RTL_RAM inference without reset operation

Testing the design further, the reset operation is found to be the culprit. Deleting the reset conditional within the always procedural block realizes an RTL_RAM inference when elaborating the design as shown in Figure 6, and the reset control signal is disconnected.

After restoring the reset operation inside the procedural block, the design is simulated and synthesized next.

Observation & Results

Time	Register Transfers	read_data
0ns	Address = 3'b000 Write_enable = 1'b1 Write_data = 8'hA4	XX
10ns	Read_enable = 1'b1 Write_enable = 0	XX
15ns		8'hA4
20ns	Read_enable = 1'b0 Write_enable = 1'b1 Write_data = 8'h11 Address = 3'b100	8'hA4
30ns	Read_enable = 1'b1	8'hA4
35ns		8'h11
40ns	Reset = 1'b1	
45ns		8'h00

A testbench is developed to instantiate the RAM as the Unit Under Test (UUT) with a WIDTH value of 8, and a ROWS value of 8 for an 8B configuration. The CLK signal is set to period of 10ns. An *initial* procedural block is used to carry out the necessary register transfers to test the UUT as shown in the to the left.

Figure 7 displays the output of the behavioral simulation conducted with the register transfers in the table. The RAM is written to and read from two times and is also reset. First, from 0 → 15 ns, value A4 is written to address 0 and read out at time 15ns. Next, from 20 → 35 ns, value 11 is written to address 4 and read out at time 35ns. Lastly, the reset control signal is set high at t=40ns and the output register is 00 at the following active edge—45ns. Essentially, consecutive write and reads take 15ns to be in the output register—a mere two clock cycles.

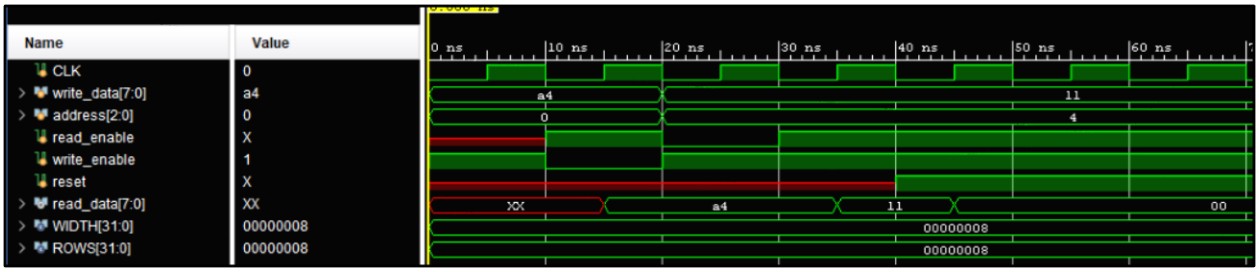


Figure 7: RAM behavioral simulation

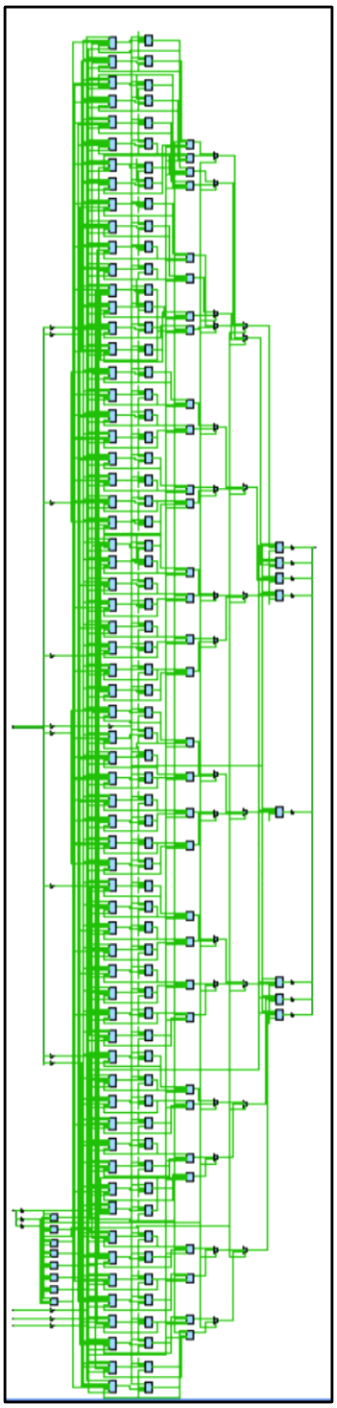


Figure 9: RAM synthesized design with reset operation

LUT	FF	BRAMs	URAM	DSP	LUTRAM	IO	GT	BUFG	MMCM	PLL	PCIe
100	72	0.00	0	0	0	23	0	1	0	0	0

Figure 8: Resource usage by RAM with reset operation

Following the simulation, the design is synthesized to verify for efficiency. As discovered in the *procedure* section, the reset operation inhibits the design from being inferred as an RTL_RAM. Two different synthesis runs will be used to elaborate this discovery: one with the reset operation defined, and one without it. First, the design is synthesized with the reset operation to meet guidelines for this assignment and is displayed in Figure 10 developed by the XST.

At first glance, the design seems excessive with the mere number of components used as displayed in Figure 9 above. With 100 LUTs and 72 FFs, the design is surely highly inefficient when the Artix-7 FPGA has RAM resources to use. Investigating the logic synthesized, the reset signal is shown as an input to the LUT6 in Figure 8—used due to the number of inputs. This means the output is a function of the reset signal, and therefore the large network of logic is needed to handle this single bit. In other words, rather than writing data directly to the RAM, this design checks first for the assertion of the reset bit, which renders the desired operation unfulfilled. Certainly, the reset operation is a resource intensive operation, and belongs best in a ROM instead where entries are only read from.

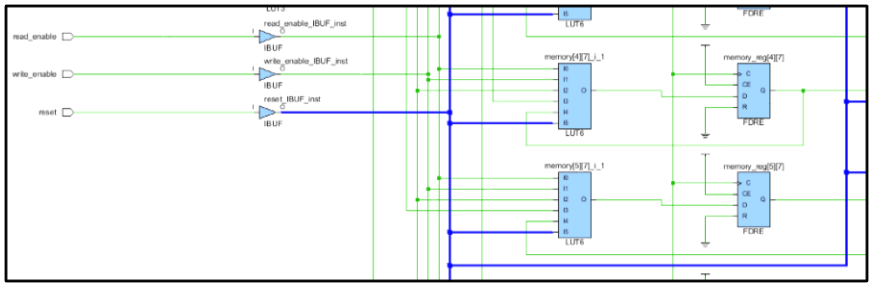
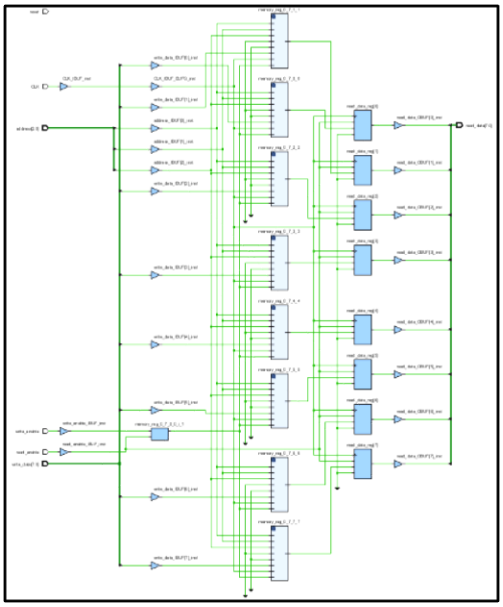


Figure 10: Reset control signal as input to LUT6

For learning purposes, the design is synthesized without the reset operation. Figure 11 to the right displays the XST output for the RAM module with the reset control signal shown in the top left disconnected. To understand the layout, the signal path for a single input data bit is inspected.

Figure 11: RAM synthesized without reset operation



With this approach, it can be seen the XST makes use now of the RAM32X1S resources within the Artix-7 FPGA. A LUT2 is used to selectively enable the RAM to read or write, and the FF after the RAM output is used to "gate" the output register from displaying results until the read_enable is asserted. This simplified signal path is shown in Figure 12 below.

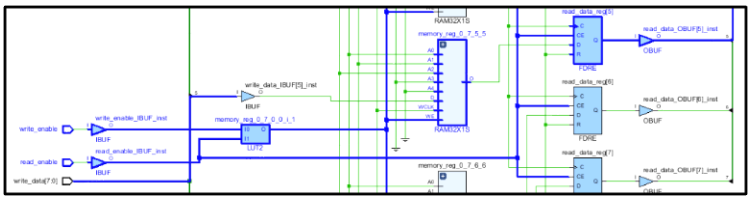


Figure 12: Highlighted cells for simplified signal path for a single bit

It is interesting to take note of the XST's decision to use the RAM32X1S primitive over the RAM64X1S primitive. Since pins A3 and A4 are grounded, the address select bus is effectively only 3 bits, giving a RAM space of 8 entries, which is what the ROWS *parameter* is specified to be at by default. With this in mind, each RAM32X1S cell provides an output data width of 1 bit and cascading 8 cells together as shown in Figure 11 gives the design an effective RAM space of 8 entries X 8-bits. Since this size of RAM cell is the smallest in the XST primitive manual, the synthesizer came up with this clever approach in defining an 8B RAM space—at a better rate of efficiency than the one in Figure 10, but without the reset operation.

An additional observation arises about the operation of the XST. Upon further investigation in the XST manual, the Artix-7 FPGA can use RAM resources in a block and distributed manner, which essentially means using dedicated RAM components over LUT based RAM respectively. In the design without the reset operation, distributed RAM is used as confirmed by the property of one of the RAM32X1S cells as shown in Figure 13.

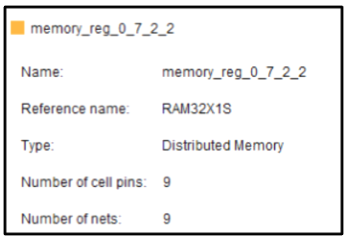


Figure 13: RAM32X1S Leaf Cell

This finding confirms the XST's manual definition on the types of distributed and block RAM available to Series 7 FPGAs, with synchronous read mapping to either block or distributed RAM:

> The type of inferred RAM depends on its description.
> - RAM descriptions with an *asynchronous* read generate a distributed RAM macro.
> - RAM descriptions with a *synchronous* read generate a block RAM macro. In some cases, a block RAM macro can actually be implemented with distributed RAM. The decision on the actual RAM implementation is done by the macro generator.

In addition, the distributed RAM is chosen because the XST claims there are speed improvements over block RAM when the effective memory size is small as shown in Figure 14.

To achieve better design speed, XST implements small RAMs and ROMs using distributed resources. RAMs and ROMs are considered *small* if their sizes follow the rules shown in the following table.

Rules for Small RAMs and ROMs

Devices	Size (bits) * Width (bits)
Virtex®-4	<= 512
Virtex-5	<= 512

Figure 14: XST Manual Distributed RAM

Lastly, one final question is posed, why is the RAM32X1S cell declared to not be of sequential design in the properties section as shown in Figure 15? As declared in the previous investigation of the differences between distributed and block RAM, the most logical explanation is the use of LUTs in the latter, which are combinational by design.

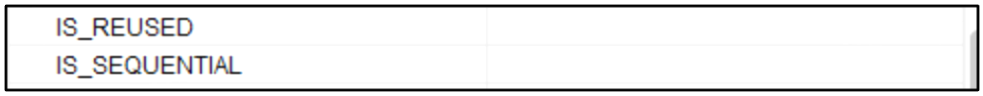


Figure 15: Properties of RAM32X1S Leaf Cell

Conclusion

The design of the 8B Synchronous RAM with reset allowed the learning of more efficient Verilog practices. For example, the procedural *always* block made use of a *casez* statement at first to assess read, write, and reset control signals without worry for Hi-Z values. This approach leaves room for error because every possible combination must be contemplated, and preliminary simulations could not verify the design for correct operation. Conditional statements were used instead to only verify the assertion of a control signal, rather than the de-assertion as needed in the *casez* statement.

Furthermore, the differentiation of the RAM with and without the reset operation allowed to learn the Vivado IDE at a deeper level and informed how the XST calculates the necessary resources to use on an FPGA. With the former utilizing MUXes and the latter utilizing actual RAM cells, this project allowed the learning of inferred cell design for porting to an FPGA. It is evident the reset operation created a design inefficacy and further research must be conducted for a better implementation by making use of the RAM cells within the FPGA.

Code

MEMORY_SYNC.V

```
`timescale 1ns / 1ps

module MEMORY_SYNC
```

```
#(parameter WIDTH = 8, parameter ROWS = 8) (
input CLK,

input [WIDTH-1:0] write_data,       //the data to write to memory address
input [$clog2(WIDTH)-1:0] address,  //the address in memory to write or read data

input read_enable,                  //control signal to enable read of data
input write_enable,                 //ctrl signal to enable write of data
input reset,                        //ctrl signal to enable reset of data

output reg [WIDTH-1:0] read_data    //the data read from memory
);

reg [WIDTH-1:0] memory [0:ROWS-1]; //create array of # ROWS with WIDTH sized entries

integer i=0; //loop variable to reset memory
always@(posedge CLK) begin

    if(reset) begin //set all rows of memory to GND
        for(i=0; i<ROWS; i = i+1) begin
            memory[i] = 0;
        end
    end

    if(read_enable & write_enable) begin
        read_data = memory[address]; //give priority to read over write
    end
    else if(read_enable) begin
        read_data = memory[address];
    end
    else if(write_enable) begin
        memory[address] = write_data;
    end

end
endmodule
```

MEMORY_SYNC_SIM.V

```
`timescale 1ns / 1ps
module MEMORY_SYNC_SIM();


parameter WIDTH = 8;
parameter ROWS = 8;

reg CLK;
initial CLK = 1'b0;
always #5 CLK = ~CLK;

reg [WIDTH-1:0] write_data;
reg [$clog2(WIDTH)-1:0] address;
reg read_enable;
reg write_enable;
reg reset;

wire [WIDTH-1:0] read_data;

MEMORY_SYNC #(WIDTH, ROWS) UUT(
.CLK(CLK),
.write_data(write_data),
.address(address),
.read_enable(read_enable),
.write_enable(write_enable),
.reset(reset),

.read_data(read_data)
);

initial begin
//write A4 to address 0
write_data = 8'hA4;
address = 3'b000;
write_enable = 1'b1;
#10 write_enable = 1'b0;
```

```
//read address 0
read_enable = 1'b1;
#10 read_enable = 1'b0;

//write 11 to address 4
write_data = 8'h11;
address = 3'b100;
write_enable = 1'b1;
#10 write_enable = 1'b0;

read_enable = 1'b1;
write_enable = 1'b1; //check if both high

#10 reset = 1'b1; //has priority over everything
end
endmodule
```