Lab 3 Report — Block RAM

Computer Design Laboratory ECE 3710
Fall 2021
The University of Utah

Jacob Peterson
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Brady Hartog
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Isabella Gilman
Computer Engineering 2023
University of Utah
Salt Lake City, UT

Nate Hansen
Computer Engineering 2023
University of Utah
Salt Lake City, UT

***Abstract*—This paper details the design and synthesis of a simple Verilog module for a 1024-byte instantiation of Block RAM on a Cyclone V FPGA. This module will be used as a component for a model CPU that resembles the CR16. As such, we ensured that this BRAM module would be dual-port and allow for simultaneous read and write operations. We demonstrated the correct read-write functionality of our BRAM in multiple ways. We used a Verilog testbench to demonstrate correct operation of the virtual hardware, and we built a simple FSM to demonstrate the correct behavior of the memory in hardware. We are prepared to expand the capacity of this module should 1024 bytes prove insufficient for our purposes. This preliminary demonstration is meant only to show correct operation of the module. Adjustments to parameterization or behavior may be made at a later date as we better understand what we need the CPU to accomplish.**

I. Introduction

The Cyclone V offers many options for data storage. We need our CR16 to be able to pre-load files to begin program execution when we synthesize and flash the FPGA. From the beginning we have been designing CPU datapath elements for 16-bit words, so our memory will store data in that width. We chose to use two M10K memory blocks for the purposes of this lab. The section "BRAM Hardware Details on Cyclone V" justifies this decision.

Intel Quartus Prime provides various templates of code that will synthesize as Block RAM on the FPGA. We utilized one of these templates to code our RTL design and implement data preloading from a file. We added some customization to the template to make the module easier to use when we integrate it with the CR16. We tested the RTL in simulation using a testbench, and we tested its capability on the board using a simple FSM module with hard-coded reads/writes. Our tests are designed to cover all the possible behaviors of a true dual-port RAM–two simultaneous reads, two simultaneous writes, and a simultaneous read/write. Our module is demonstrably operable in simulation and on the FPGA.

Finally, we review the synthesis reports provided by Quartus Prime to analyze the M10K's resource use and gather information that will help us to make decisions about future trade-offs with our design. This information will guide our design practice as we move on to the full integration of datapath elements into the completed CR16 CPU.

II. BRAM Hardware Details on Cyclone V

We determined that the RAM block should have true dual-port capabilities to integrate seamlessly with the functionality of our CR16 CPU. True dual-port mode is only available on the Cyclone V FPGA using the M10K memory block. For the purposes of this lab, we will instantiate two units of 512x16-bit M10K memory. We will only use two units for testing purposes, but we have the option to include as many as 397 units based on FPGA capacity [1]. It makes sense to use 16-bit words for a 16-bit PC. However, we have capability to expand the scheme to as many as 20 bits. This is good to keep in mind in the event that we need to augment instruction length to add capability to the CPU when we put it to work. The volume capabilities of each M10K unit are detailed in the table in Fig. 1.

Table 2–4. M10K Block Mixed-Width Configurations (True Dual-Port Mode)

Port B	Port A							
	8K x 1	4K x 2	2K x 4	2K x 5	1K x 8	1K x 10	512 x 16	512 x 20
8K x 1	Yes	Yes	Yes	—	Yes	—	Yes	—
4K x 2	Yes	Yes	Yes	—	Yes	—	Yes	—
2K x 4	Yes	Yes	Yes	—	Yes	—	Yes	—
2K x 5	—	—	—	Yes	—	Yes	—	Yes
1K x 8	Yes	Yes	Yes	—	Yes	—	Yes	—
1K x 10	—	—	—	Yes	—	Yes	—	Yes
512 x 16	Yes	Yes	Yes	—	Yes	—	Yes	—
512 x 20	—	—	—	Yes	—	Yes	—	Yes

Fig. 1. Table for width configurations of the M10K memory unit in true dual-port mode. [1]

Based on the documentation, it is unclear where this memory block exists on the physical FPGA. We will be able to interpret its resource occupation from the synthesis reports after coding and synthesizing an RTL instance. Once we have that information, we can determine the possible trade-offs that come with expanding the memory capacity and adding future functional elements like an instruction decoder, program counter, and datapath combination. Many of these decisions will be made after the CR16 has been implemented and we have a more complete perspective.

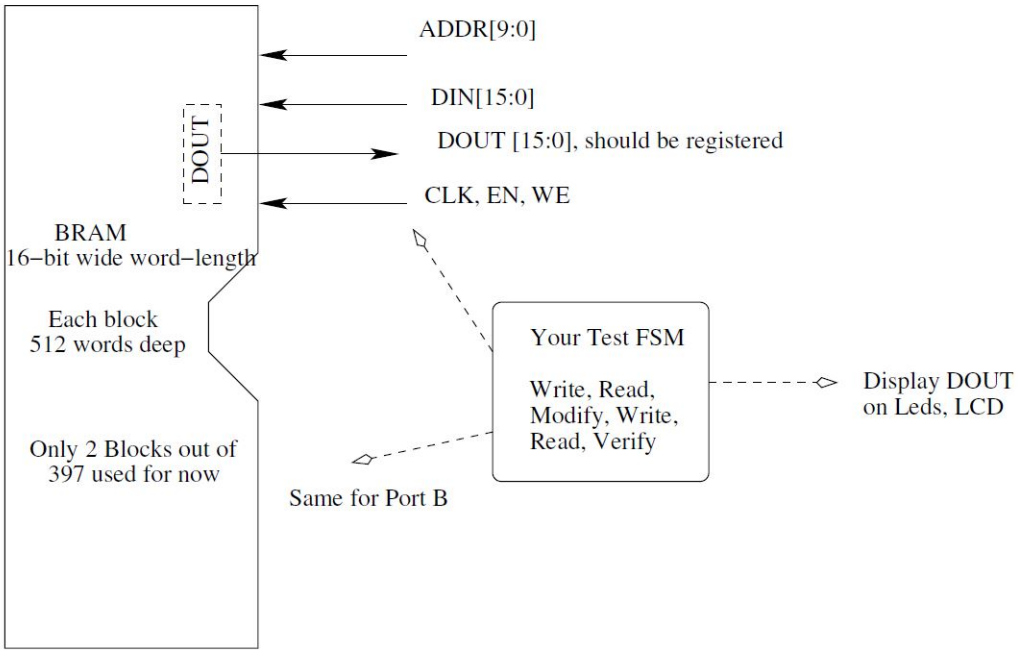

Fig. 2. Block diagram for 2 units of 512x16-bit Block RAM with true dual-port capabilities.

For now, we will follow a general model as outlined by the block diagram in Fig. 2.

III. BRAM RTL

The RTL for our BRAM module was adapted from a template generated by Quartus Prime. It supports simultaneous reading and writing via a dual-port interface with I/O data lines and address lines for each port. The module is completely parameterized allowing for a variable data width and address width. It also supports initializing the BRAM upon FPGA programming or flashing given a UTF-8-encoded initialization file via the `readmemh` Verilog system task. This initialization file simply contains n-bit hex values on every line where each line corresponds to the data at a given memory address. Additionally, the parameter list allows address offsetting to be used so that a module instantiation with a BRAM initialization file can be pre-loaded at a specific address in BRAM. If no initialization file is given, then the entirety of BRAM is initialized to zeros.

The register array contained within the BRAM module consists of a 2-dimensional array where the first dimension is an unpacked array representing a word and the second dimension is a packed array containing the bits of a word. There are two `always` blocks sensitive to the `posedge` of the clock signal—one block for port A and one for port B. If the `I_WRITE_ENABLE` signal on the given port is asserted, then the value from the data inputs are written to the register array at the given address and the output data is latched to its value from the previous clock cycle. If the `I_WRITE_ENABLE` signal on the given port is reset, then the output data lines get the word contained in register array at the given address.

According to synthesis reports, the body of this BRAM module gets "synthesized away" upon compilation and synthesis within Quartus Prime. Since the BRAM already exists on the FPGA, there is no need to synthesize this module completely and utilize logic elements, but instead, connections are made between the BRAM on the FPGA and the `input` and `output wires` declared in a `bram` module instantiation.

The signature and body of our BRAM module is shown in Figure 3.

IV. BRAM Top

To demonstrate a functional BRAM module, we created a top-level module with an FSM that steps through various sequences of instructions that read and write data to BRAM. The lower 8 bits of the output data lines are displayed in hexadecimal on the 7-segment displays on the FPGA board and a button is used to step through the FSM states. A BRAM initialization file containing arbitrary data at specific memory addresses is used to confirm that the BRAM is properly initialized with the desired data upon programming or flashing. A state machine is constructed with the following states. Note that the numbers shown on the 7-segment displays are used to validate data integrity and correct FSM and BRAM functionality. Also note that the first 5 states are used to step through memory addresses at locations with known data given in the initializing file.

- `S0`: Reset state (read 0th memory address on both ports (shows 0303 on 7-segments)).

```
module bram
       #(parameter P_BRAM_INIT_FILE = "",
         parameter integer P_BRAM_INIT_FILE_START_ADDRESS = -1,
         parameter integer P_BRAM_INIT_FILE_END_ADDRESS = -1,
         parameter integer P_DATA_WIDTH = 16,
         parameter integer P_ADDRESS_WIDTH = 10)
       (input I_CLK,
        input [P_DATA_WIDTH - 1 : 0] I_DATA_A, I_DATA_B,
        input [P_ADDRESS_WIDTH - 1 : 0] I_ADDRESS_A, I_ADDRESS_B,
        input I_WRITE_ENABLE_A, I_WRITE_ENABLE_B,
        output reg [P_DATA_WIDTH - 1 : 0] O_DATA_A, O_DATA_B);

reg [P_DATA_WIDTH - 1 : 0] ram [0 : 2 ** P_ADDRESS_WIDTH - 1]; // 2D register array for RAM

// Initialize entire BRAM to zeros if 'P_BRAM_INIT_FILE' is empty or
// read contents of 'P_BRAM_INIT_FILE' file to BRAM.
integer index;
initial begin
    if (P_BRAM_INIT_FILE == "")
        for (index = 0; index < 2 ** P_ADDRESS_WIDTH; index++)
            ram[index] = {P_DATA_WIDTH{1'd0}};
    else
        if (P_BRAM_INIT_FILE_START_ADDRESS != -1 && P_BRAM_INIT_FILE_END_ADDRESS != -1)
            $readmemh(P_BRAM_INIT_FILE, ram,
                      P_BRAM_INIT_FILE_START_ADDRESS, P_BRAM_INIT_FILE_END_ADDRESS);
        else if (P_BRAM_INIT_FILE_START_ADDRESS != -1)
            $readmemh(P_BRAM_INIT_FILE, ram, P_BRAM_INIT_FILE_START_ADDRESS);
        else
            $readmemh(P_BRAM_INIT_FILE, ram);
end

// Port A
always @(posedge I_CLK) begin
    if (I_WRITE_ENABLE_A) begin
        ram[I_ADDRESS_A] <= I_DATA_A;
        O_DATA_A <= O_DATA_A;
    end
    else
        O_DATA_A <= ram[I_ADDRESS_A];
end

// Port B
always @(posedge I_CLK) begin
    if (I_WRITE_ENABLE_B) begin
        ram[I_ADDRESS_B] <= I_DATA_B;
        O_DATA_B <= O_DATA_B;
    end
    else
        O_DATA_B <= ram[I_ADDRESS_B];
end
endmodule
```

Fig. 3. The BRAM module signature and body.

- `S1`: Read 1st memory address on port A (shows 0x02 on the 2 right 7-segments).
- `S2`: Read 2nd memory address on port B (shows 0x01 on the 2 left 7-segments).
- `S3`: Read 1021st memory address on port A (shows 0x21 on the 2 right 7-segments).
- `S4`: Read 1022nd memory address on port B (shows 0x22 on the 2 left 7-segments).
- `S5`: Read 1023rd memory address on port A (shows 0x23 on the 2 right 7-segments).
- `S6`: Overwrite 1023rd memory address via port A with 0xAA.
- `S7`: Read overwritten 1023rd memory address on port B while simultaneously writing to 0th memory address via port A (shows AA23 on 7-segments).
- `S8`: Read updated 0th memory address on port A and port B (shows 0101 on 7-segments).

The `bram` module instantiation was clocked to the 50Mhz FPGA clock which creates seamless state transitions and demonstrates that there were no issues with propagation delays when using a high frequency clock.

V. Testbench

The testbench is designed to test the read and write abilities of port A and B in the Block RAM. Before testing, the first eight words in memory are initialized to values 1-8. From here, four separate tests demonstrate the read/write abilities of our memory access interface.

1) Test 1 reads the values stored at addresses 0-7 in memory on port A.
2) Test 2 tests whether ports A and B are able to read from memory simultaneously. For this test, port A attempts to read from addresses 0-5, while port B read from 2-7.
3) Test 3 is designed to test the write ability on port A. During this test, port A overwrites the existing data in memory, then port B reads the value and verifies its correctness. For this test, the value at addresses 0-7 get overwritten by the value of the address index multiplied by 2.
4) Test 4 is similar to test 3 and uses port B to overwrite the existing data in memory, then reads the values using port A. For this test, the value at addresses 0-7 get overwritten by the value of the address index multiplied by 3.

VI. Compilation Report

We selected some key figures from the Quartus Prime compilation report on our Block RAM interface. These figures are shown in Fig. 4 through Fig. 6. Some key observations include the following:

- The analysis and synthesis summary of Fig. 4 shows that our design uses 16,384 block memory bits in total. We expected to have: (2 BRAM modules) × (512 words/module) × (16 bits/word) = 16,384 bits in total, so this result is verified. The result is also verified by Fig. 6.
- The RAM usage details of Fig. 5 verify that our design contains two BRAM modules and that each is in dual-port mode with a 16-bit word width.

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Thu Oct 07 16:37:21 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CR16
Top-level Entity Name	bram_top
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	9
Total pins	31
Total virtual pins	0
Total block memory bits	16,384
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Fig. 4. High-level statistics of the Block RAM interface when synthesizing *bram_top.sv* as the top-level module.

Analysis & Synthesis RAM Summary

Name
bram:i_bram|altsyncram:ram_rtl_0|altsyncram_25u1:auto_generated|ALTSYNCRAM
bram:i_bram|altsyncram:ram_rtl_1|altsyncram_68r1:auto_generated|ALTSYNCRAM

Type	Mode	Port A Depth	Port A Width	Port B Depth	Port B Width
AUTO	Simple Dual Port	1024	16	1024	16
AUTO	Simple Dual Port	1024	16	1024	16

Size	MIF
16384	db/CR16.ram0_bram_99bdc352.hdl.mif
16384	db/CR16.ram0_bram_99bdc352.hdl.mif

Fig. 5. Details of RAM usage for the Block RAM interface.

Analysis & Synthesis Resource Usage Summary	
Resource	Usage
Estimate of Logic utilization (ALMs needed)	21
Combinational ALUT usage for logic	38
-- 7 input functions	0
-- 6 input functions	0
-- 5 input functions	0
-- 4 input functions	30
-- <=3 input functions	8
Dedicated logic registers	9
I/O pins	31
Total MLAB memory bits	0
Total block memory bits	16384
Total DSP Blocks	0
Maximum fan-out node	write_enable_a
Maximum fan-out	16
Total fan-out	340
Average fan-out	2.72

Fig. 6. Details of resource usage for the Block RAM interface.

VII. Conclusion

The BRAM module is an essential component for efficient CPU functionality. By demonstrating correct functionality, we can be confident that our CPU will properly fetch instructions, write values, and navigate throughout the program based on coded control flow. We've shown correct operation in simulation and practice, so we are confident that the data will transfer properly. Although we were unable to run a proper instance of the Quartus Timing Analyzer, we are confident

that there are no hold/setup time violations because the FSM we used to demonstrate on the FPGA was able to write and read on a single clock cycle. If this is an issue in the future, we can modify our FSM to handle setup, hold, and write-back. Based on the results we have seen so far, we are ready to begin integrating all of these elements together into one integrated datapath managed by a single CPU Finite State Machine.

REFERENCES

[1] ALTERA. (2012) Cyclone v device handbook.