Hardware Verification for a Computation Storage Module

By: Ali Badry

Table of Contents

List of Figures

List of Tables

➢ Design

In order to satisfy the condition of having an RTL that targets the following platforms: (Simulation – Emulation – FPGA), synthesizable RTL should be written, not just behavioral constructs. The language used to write the RTL is System Verilog.

The description of the module may sound easy, but dealing with memories requires paying extra attention. Most of current memories' designs use single port or dual port memories, that's why we used configurable dual port memory. Thus, block diagram should be designed first to describe and determine the used hardware in our system.

Computation storage module

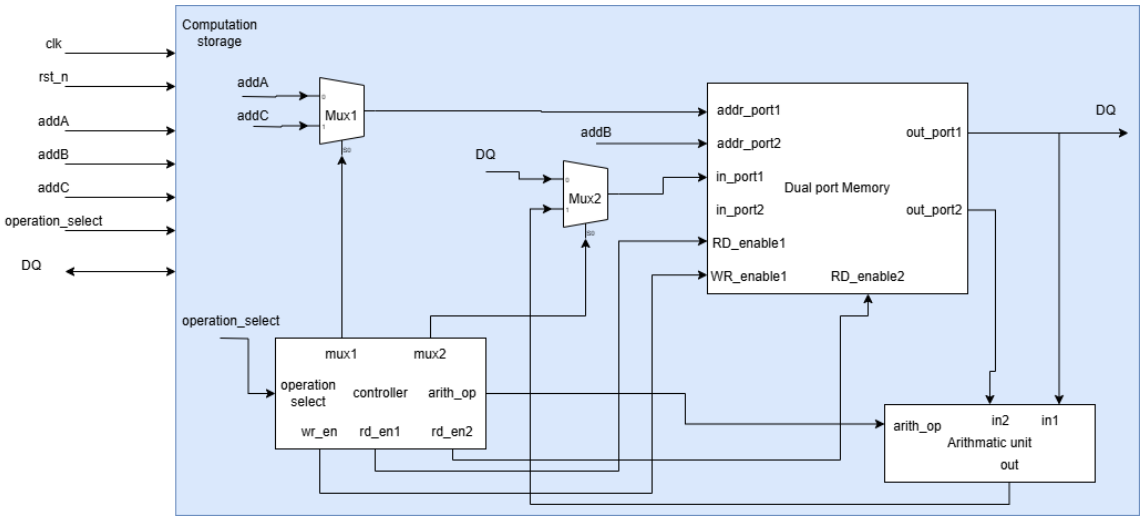

Figure 1. Computation storage module block diagram

Computation storage module parameters

PORT NAME	DEFAULT VALUE	DESCRIPTION
MEM_WIDTH	8	Memory word length
MEM_DEPTH	16	Number of memory entries
NO_OPERATIONS	4	Number of supported commands

Table 1. Computation storage module parameter list

Computation storage module ports

PORT NAME	PORT WIDTH	PORT DIRECTION	DESCRIPTION
CLK	1	Input	System main clock
RST_N	1	Input	Negative edge global reset
ADDA	Log2(MEM_DEPTH)	Input	Input port 1 memory address
ADDB	Log2(MEM_DEPTH)	Input	Input port 2 memory address
ADDC	Log2(MEM_DEPTH)	Input	Output port 1 memory address
OPERATION_SELECT	2	Input	Determines memory operation
DQ	MEM_WIDTH	In/out	Memory modifier/reader bus

Table 2. Computation storage module port list

System Description:

Commands description may be straight forward; however, special handling should be considered due to the constraint of the dual port memory, *which is read in a cycle, write in another cycle*. We can't read, process and write in the same cycle.

The controller handles the input and output of the memory not to violate its functionality. It also determines the arithmetic operation done at a specific time. If the operation requires reading and writing, the handling is done as following: first insert the reading addresses (which is in our case, address A and address B) in a single clock cycle, then it is followed by processing and writing in the next clock cycle.

This gives a delay of 2 clock cycles which is acceptable. Therefore, the inputs should remain constant at the input ports for at least 2 clock cycles to ensure the correctness of processing.

The DQ port requires special handling as it is bidirectional. Thus, we control it by an enable signal that makes it output in case of the first operation by driving the output of port 1 memory. However, it remains unconnected in the rest of the operation it be driven by an external source.

A direct testbench is written to make sure that the main functionalities of the system works (only 4 test cases for the 4 operations are written). UVM is test bench is also written and described later in detail in the verification section.

Dual port memory

It doesn't have special implementation to show, just normal memory that has independent ports that can read and write simultaneously (shown in Figure 6).

Dual port memory parameters

PORT NAME	DEFAULT VALUE	DESCRIPTION
MEM_WIDTH	8	Memory word length
MEM_DEPTH	16	Number of memory entries

Table 3. Dual port memory parameter list

Dual port memory ports

PORT NAME	PORT WIDTH	PORT DIRECTION	DESCRIPTION
CLK	1	Input	System main clock
RST_N	1	Input	Initiate memory output
ADDR_PORT1	Log2(MEM_DEPTH)	Input	Input port 1 memory address
ADDR_PORT2	Log2(MEM_DEPTH)	Input	Input port 2 memory address
IN_PORT1	MEM_WIDTH	Input	Input memory port 1
IN_PORT2	MEM_WIDTH	Input	Input memory port 2
RD_ENABLE1	1	input	Port 1 read enable
RD_ENABLE2	1	input	Port 2 read enable
WR_ENABLE1	1	input	Port 1 write enable
WR_ENABLE2	1	input	Port 2 write enable
OUT_PORT1	MEM_WIDTH	Output	Output memory port 1
OUT_PORT2	MEM_WIDTH	Output	Output memory port 2

Table 4. Dual port memory port list

Arithmetic unit

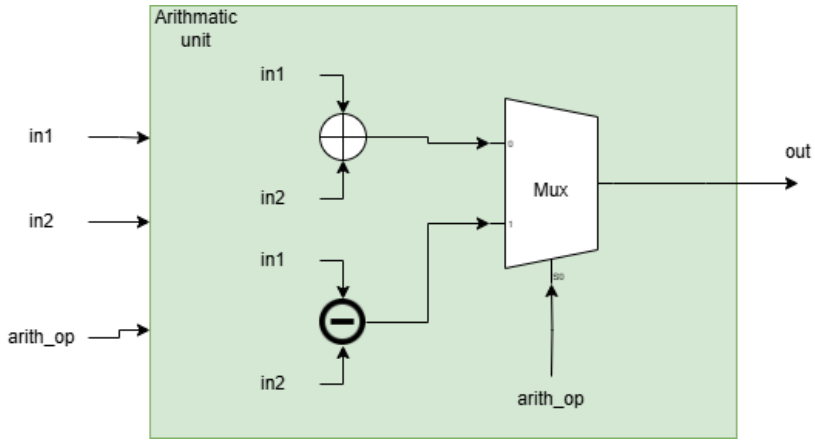

Figure 2. Arithmetic unit block diagram

Arithmetic unit parameters

PORT NAME	DEFAULT VALUE	DESCRIPTION
WIDTH	8	Input and output word length

Table 5. Arithmetic unit parameter list

Arithmetic unit ports

PORT NAME	PORT WIDTH	PORT DIRECTION	DESCRIPTION
IN1	WIDTH	Input	First operand
IN2	WIDTH	Input	Second operand
ARITH_OP	1	Input	Operation selection
OUT	WIDTH	Output	Operation result

Table 6. arithmetic unit port list

Controller

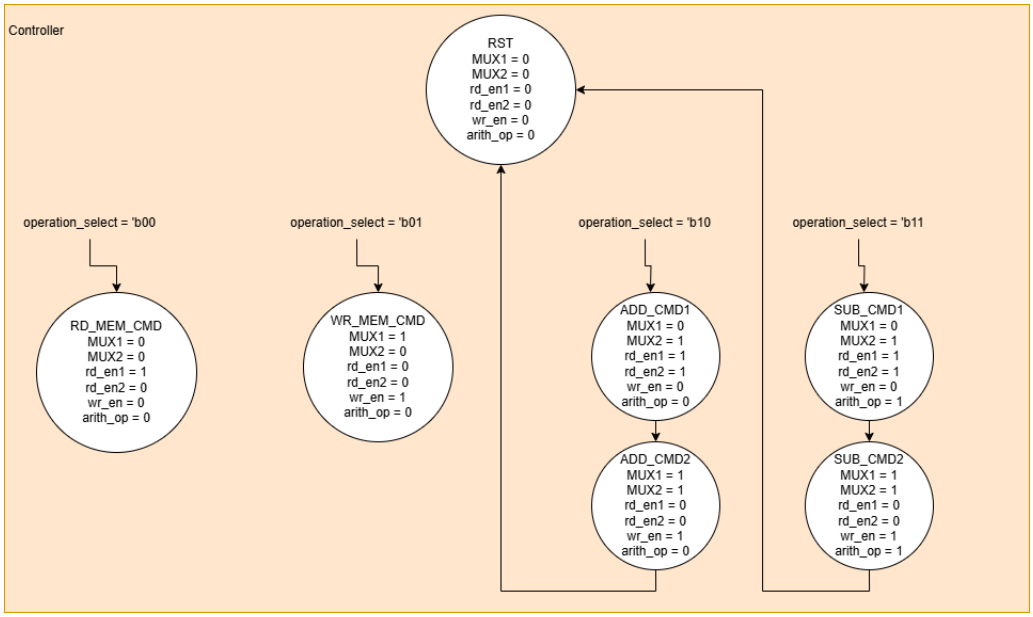

Figure 3.Controller finite state machine

Controller ports

PORT NAME	PORT WIDTH	PORT DIRECTION	DESCRIPTION
CLK	1	Input	System clock
RST_N	1	Input	Negative edge global reset
OPERATION SELECT	2	Input	Operation selection
RD_EN1	1	Output	Port 1 read enable
RD_EN2	1	Output	Port 2 read enable
WR_EN1	1	Output	Port 1 write enable
MUX1	1	Output	Multiplexer 1 selection line
MUX2	1	Output	Multiplexer 2 selection line
ARITH_OP	1	Output	Arithmetic operation selection

Table 7. Controller port list

Direct testbench simulation

```
# at time                    0,  address A: 0, address B: 0, address C: 0, operation select: 0, DQ: 00
# at time                10000,  address A: 8, address B: 0, address C: 0, operation select: 0, DQ: 00
# at time                25000,  address A: 8, address B: 0, address C: 0, operation select: 0, DQ: 55
# RD_CMD_MEM (test1) passed successfully!
# at time                30000,  address A: 8, address B: 0, address C: 1, operation select: 1, DQ: ff
# WR_CMD_MEM (test2) passed successfully!
# at time                50000,  address A: a, address B: 5, address C: d, operation select: 2, DQ: ff
# ADD_CMD (test3) passed successfully!
# at time                80000,  address A: c, address B: 3, address C: 7, operation select: 3, DQ: ff
# SUB_CMD (test4) passed successfully!
# ** Note: $finish    : CS_TB.sv(106)
#    Time: 156 ns  Iteration: 0  Instance: /CS_TB
```

Figure 4.Direct testbench simulation results

The 4 test cases injected by the direct testbench simulated on QuestaSim passed successfully as shown in the previous figure, which gives a first intuition on the functionality of the system. Later in the UVM testbench, more testcases should be held to see further the block's overall functionality.

Hardware synthesis

To check if the system is hardware compatible, FPGA synthesis and implementation using Vivado is used, to see whether it contains unsythesizable hardware constructs or not. The FPGA used in the process is ZYNQ UltraScale+ ZCU104 with constraint of 120MHz frequency to observe timing and resources compatibility. The implementation completed successfully as shown in the following figure without any timing violation.

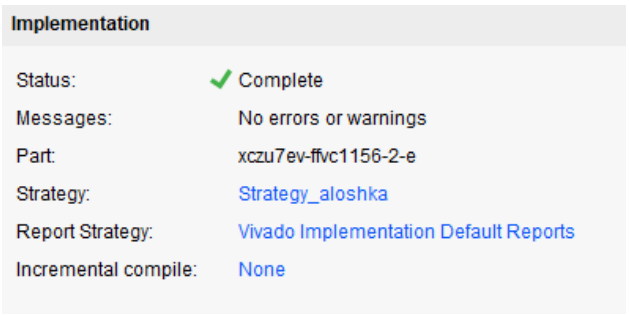

Figure 5. Implementation results

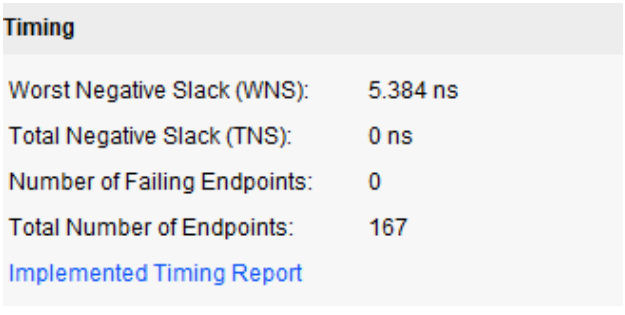

Figure 6.Implementation timing results

Scripts

There are two scripts to make the simulation easier and faster:

1. **Python script:** this script generates memory contents and save them in an external text file named “MEM_content.txt” to initialize the memory with random values. The width and the depth of the memory are configurable to facilitate any future changes in the memory’s size.
2. **DO script:** this script automates the compilation, wave generation and simulation processes. What is needed to be done is opening QuestatSim tool in the design’s directory, then type “do run.do” to automate the simulation process.

➢ Verification

Introduction

As we saw earlier, the design mainly deals with a memory element that sends/receives single or multiple data at the same time. It also contains a processing element that has only two operations (+/-).

Hence, we try to mainly test those four operations with different conditions or arrangement to observe the system's transitions and flow from one operation to the other. We should also test the range of acceptable data that will not cause the results to overflow when dealing with the arithmetic operations. How the memory will handle various locations access.

We should also consider the quality of the written RTL with code coverage by observing if there is dead code, dummy transitions or unreachable states.

Verification environment

Our UVM testing environment (shown in figure 2) consists of various components. The components are:

1. **UVM Test:** it gathers the environments in single place and responsible for starting sequences.
2. **UVM Environment:** it contains the agent(s) in a single component.
3. **UVM Agent:** it is responsible for interacting with DUT, scoreboard and subscriber. It contains the driver, sequencer and monitor.
4. **UVM Sequencer:** it passes the sequence items to the driver to apply the test cases to the DUT.
5. **UVM Driver:** it converts the transactions received from the sequencer to DUT's pin level through the virtual interface connected to it.
6. **UVM Monitor:** it captures the data from the DUT through virtual interface, converts it to transactions to send it to the scoreboard and subscriber.
7. **UVM Scoreboard:** it captures the transactions from the monitor and compare it with the reference model or reference data to determine success or failure.
8. **UVM Subscriber:** it captures the transactions from the monitor for coverage.

The flow would start by building the UVM components first. In consequence, we connect them with each other. Then, we apply our sequence items to drive the DUT and observe the output or the changes in the DUT. Finally, we perform code and functional coverage to measure the quality of the verification environment.

Features to be verified

- **Memory reading:** to make sure that with sending a valid memory address, we should read the right content from the preloaded memory.
- **Memory writing:** to write in the right location provided by the specified port address after getting a direct data from the input port or storing the result of the arithmetic processing element.
- **Performing addition operation:** by sending the adding command to the computation storage module, we should make sure that the stored element is the result of adding the 2 operands.
- **Performing subtraction operation:** by sending the subtraction command to the computation storage module, we should make sure that the stored element is the result of subtracting the 2 operands.
- **Ability to transit from one command to all the other commands:** this feature tests the flexibility of the system by sending randomizing the sent operation command and observing the output of each 2 or more sequence commands.
- **Data overflow handling:** we should store large values in the memory to see how the arithmetic unit will handle this data.

- **Out of range address:** Although our memory depth's size is a power of 2 (as most of real hardware memory elements are), we should try to instantiate memory of size not a power of 2 and observe how it will handle out of range input addresses.
- **Simultaneously memory read and write using the 2 ports:** dual port memory gives the ability to read or write 2 different data at the same time using the 2 ports.
- **Inability to read and write from memory using 1 port:** we shouldn't be able to read and write multiple data at the same time using 1 port only.
- **Check the stability of the in/out port:** only 1 direction should be driven by the DQ in/out port not to give chance for the data to be corrupted whether in the memory or the output of the module.

Test plan

Table 8 shows test cases with following criteria:

- Goal of each test case.
- Derived inputs to execute the test case.
- Expected output from the DUT at each test case.

Test Name	**addA**	**addB**	**addC**	**Operation select**	**DQ (input)**	**others**	**Expected result**	**Goal**
Test 01	8	x	x	0	Z	-	DQ = MEM[8]	Read from memory at any location indicated by addA
Test 02	x	x	1	1	0xff	-	MEM[1] = 0xff	Write to memory at location indicated by addC
Test 03	10	5	13	2	Z	-	MEM[13] = MEM[5]+MEM[10]	Addition operation on operands at location addA and addB. Then storing in location addC
Test 04	12	3	7	3	Z	-	MEM[7] = MEM[12]-MEM[3]	Subtraction operation on operands at location addA and addB. Then storing in location addC
Test 05	4	0	11	0 => 2	Z	-	DQ = MEM[4] => MEM[11] = MEM[4]+MEM[0]	Trying a new sequence of reading then addition.
Test 06	6	15	2	1 => 3	0x00 => Z	-	MEM[2] = 0xaa => MEM[2] = MEM[6]-MEM[15]	Trying a new sequence of write then addition.
Test 07	1	9	9 =>1 =>12	1=>1=>2	0xff => 0xff => Z	-	MEM[9] = 0xff => MEM[1] = 0xff => MEM[12] = MEM[1]+MEM[9]	Seeing what the result is when data is very large to process by arithmetic unit
Test 08	20	25	18	3	Z	-	Memory should read and write in overflown memory addresses	Seeing what the result is when memory addresses are larger than the bus width
Test 09	Rand	Rand	Rand	Rand	Rand	Rand	-	Number of random inputs to catch any escaped bugs

Table 8. Test cases

Test Name	addA	addB	addC	Operation select	DQ (input)	others	Expected result	Goal
Test 09	8	x	10	x	0xcc	Rd_en1 = 1, Wr_en1 = 1	MEM[8] = 0xcc	Memory reaction when reading and writing simultaneously from port1. Only read command should execute.
Test 10	x	13	3	x	0x22	Rd_en2=1, Wr_en1=1	DQ = MEM[13], MEM[3] = 0x22	Memory reaction when reading and writing simultaneously from 2 ports. Both read and write command should execute.

Table 9.Future test cases

Code and functional coverage

Code coverage:

To measure the percentage of executed code, toggling and expressions exercised, code coverage is used to achieve such thing. In the simulation, branch, statement, toggle, condition and FSM are activated to measure how much of the RTL is executed.

The achieved percentage was nearly 85% after due to unexercised reset condition in the system in each state of the FSM as shown in the following figure.

```
Total Coverage By Instance (filtered view): 85.11%
```

Figure 7. Total code coverage

Functional coverage:

Functional coverage mainly focuses on inputs and output spans as well as the cross coverage between multiple inputs. In this coverage implementation, the focus will not be on finite state transitions, rather it would be on input operation selection transition. It would also be on spanning the memory addresses and hitting the extremes of the input data ports (i.e. highest value and lowest value), as well as cross covering between the input addresses. The functional coverage code is shown in the figure below.

```
//---------covergroup definition--------------//
covergroup coverage_subscriber; //could use sensitivity list to sample at every posedge of clock
    cover1: coverpoint seq_item_in_subscriber.addA {bins bin1[] = {[0:15]};}
    cover2: coverpoint seq_item_in_subscriber.addB {bins bin2[] = {[0:15]};}
    cover3: coverpoint seq_item_in_subscriber.addC {bins bin3[] = {[0:15]};}
    cover4: coverpoint seq_item_in_subscriber.DQ_in {bins bin4[] = {8'hff,8'h00};}
    cover5: coverpoint seq_item_in_subscriber.operation_select {
            bins bin4[] = (0,1,2,3 => 0,1,2,3);}
    cross cover1,cover2;
    cross cover1,cover3;
    cross cover3,cover2;
endgroup
```

Figure 8.Functional coverage code

The result of functional coverage was 97.36% after applying 7 directed test cases and 1000 constrained random test cases.

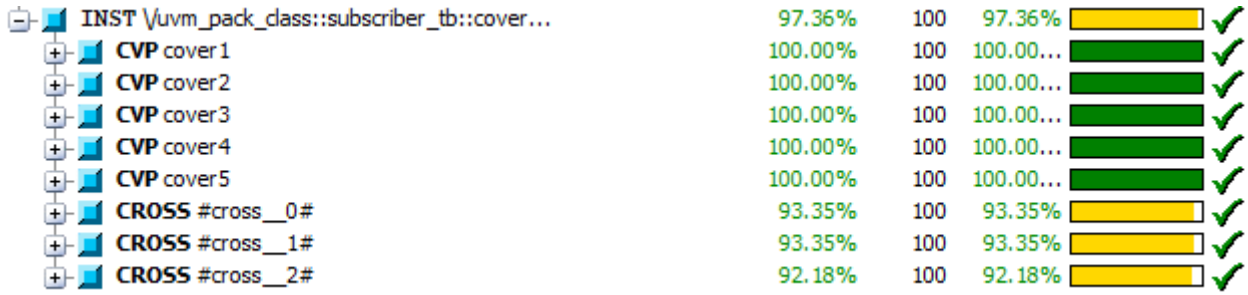

Figure 9. Functional coverage results

The total coverage percentage (code coverage + functional coverage) is 87%

Verification results

After applying around 7 directed test cases and 1000 constrained random test cases, the result of the simulation was **zero** errors after comparing the DUT operation results with the expected memory reading/storing results.

```
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 30456: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO uvm_pack_class.sv(395) @ 30456: uvm_test_top.env_in_test.scoreboard_in_env [report_phase] total errors occured:          0 from total cases:       1015
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    5
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     1
# ** Note: $finish    : C:/questasim64_2021.1/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 30456 ns  Iteration: 60  Instance: /top_tb
```

Figure 10. Simulation results