ALU with selectable inputs and outputs

This page was intentionally left blank

DISCLAIMER

Author: Dragos Constantin Doncean
Email: doncean@asicart.com
Mobile: +40-740-936997

Table of Contents

1. INTRODUCTION

The purpose of this document is to provide the necessary specifications for a core called “ALU with selectable inputs and outputs”. Specifications will refer to:

– functional description of the core
– design architecture
– design architecture description
– verification environment architecture
– verification environment description
– methodologies applied for a design's test

The proposed core is a didactical project in Verilog. It's about a core containing an ALU with three selectable inputs and two selectable outputs. The design itself is be suited for Verilog beginners willing to make the next step by building a circuit having practical requirements and that is a little more complex than the ones presented in Verilog books. In the end, the interested persons will have tested all the operator types and operator symbols existing in Verilog and will have closely observed the way Verilog performs operations with the provided operands.

The source code contains tests, ranging from direct tests and random tests to improved tests. I will present the advantages and disadvantages of each one of them. Improved tests will be a very good introduction to design verification concepts like BFMs, monitors, collectors and checkers. These verification environment components will be written in/adapted to Verilog, since a design verification language (DVL) like Vera is not widely used in technical universities or even more, not at home.

Included in source code there is a Makefile and a Perl script for running individual tests. You will surely find them useful when building a design verification environment from scratch for your own use or for a future big project.

By studying all the files in this project, understanding, modifying and improving them, you will have an idea about how a design verification looks like. Also, you will have an idea about the required steps for building a design according to specifications.

Also, experienced engineers, trainers or university academic staff and students will find this project interesting as an exercise for new learners.

2. DESIGN

2.1 Design architecture

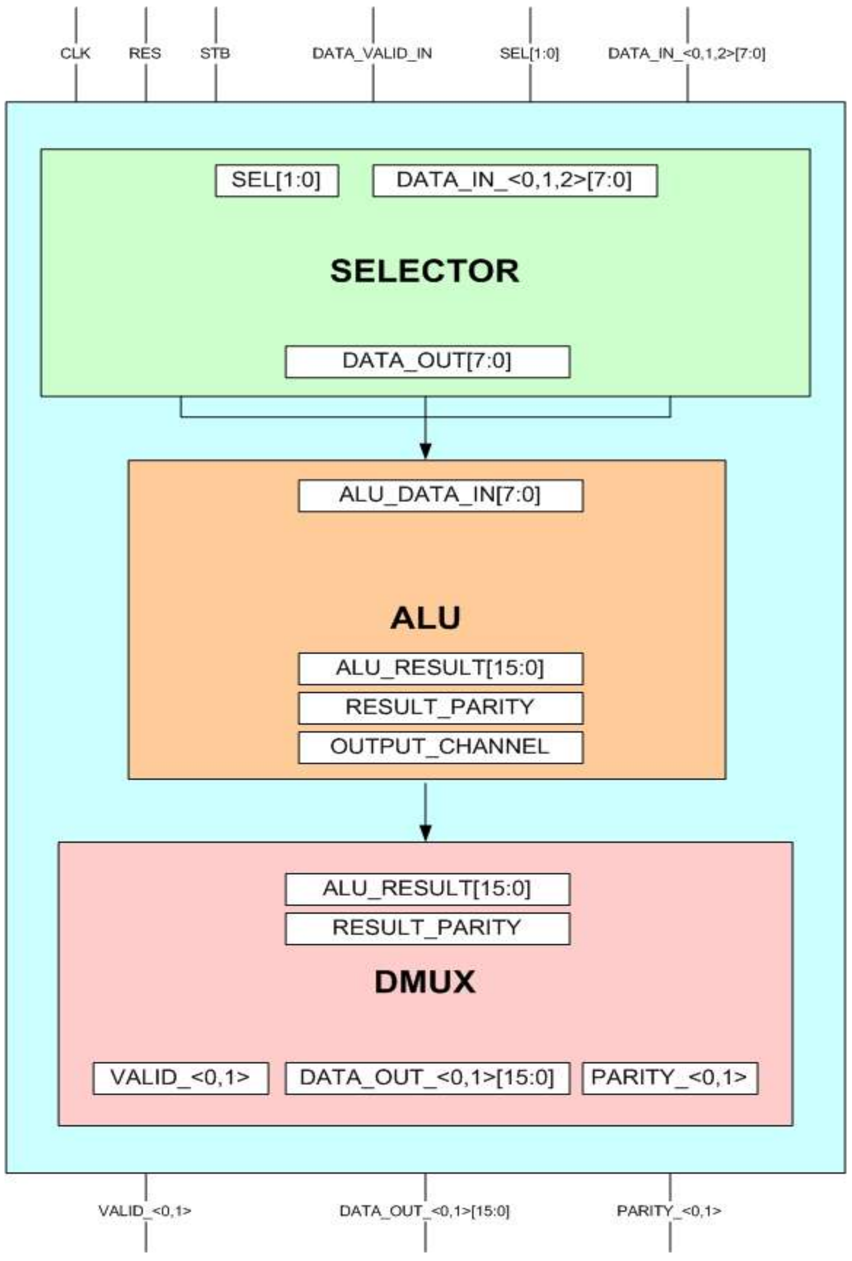

Fig. 1 – Design architecture for "ALU with selectable inputs and outputs"

2.2 Design architecture description

2.2.1. Functional description of the core

2.2.1.1.Features

- 3 input channels
- 2 output channels
- ALU performing all the operations known by Verilog
- parity calculation of ALU results
- serial data input
- serial data output
- selectable inputs depending on separate selection bits
- selectable outputs depending on selection bits from serial inputs
- signaling of valid output data

2.2.2 Functional description

2.2.2.1. Clock

The core performs its operations on the positive edge of the clock only.

2.2.2.2 Reset

The reset is asynchronous and is active high.

2.2.2.3 Input

This core should collect data from three input sources. The active source is marked by select bits. Data is being given serially on the active input, in 8 bits granularity. The first byte decodes the operator type, operation symbol and output channel. The following bytes (up to three) represent the operands. A signal should indicate that there is an active read operation.

The beginning of a data read operation should be indicated by a strobe signal. When that signal is high, the transaction's attributes should be valid (input channel selection and the first data byte).

There must be a delay of 3 clock cycles minimum between assertion of the strobe signal between two transactions in order for the circuit to function properly.

2.2.2.4 ALU

ALU should take data serially every positive clock edge from the input logic and should decode the first byte in order to see the operation that it should perform. Depending on the decoded bits, it sends the result to the output logic on its output bus. The output is 16 bits wide, the maximum necessary in order to transmit the full result.

Also, ALU calculates the parity for the result, a single bit being calculated as the XOR of all the result's bits.

2.2.2.5 ALU operations

The core's ALU should perform all possible Verilog operations according to the following table:

Operator Type	*Operator Symbol*	*Operation Performed*	*Number of Operands*
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	substract	two
	%	modulus	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two

Operator Type	*Operator Symbol*	*Operation Performed*	*Number of Operands*
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	right shift	two
	<<	left shift	two
Concatenation	{ }	concatenation	any number
Replication	{ { } }	replication	any number
Conditional	?:	conditional	three

Table 1. Operator Types and Symbols

The above table shows all the Verilog operator types and operator symbols . Our ALU will implement the exact operations, with the following modifications for simplifying the first version of this core:

– ***Operator Type*** = Concatenation => ***Number of Operands*** = two
– ***Operator Type*** = Replication => ***Number of Operands*** = two

The codifications for the above operations are:

Operator Type	***Codification (decimal)***	***Operator Symbol***	***Operation Performed***	***Codification (decimal)***
Arithmetic	0	*	multiply	0
		/	divide	1
		+	add	2
		-	substract	3
		%	modulus	4
Logical	1	!	logical negation	0
		&&	logical and	1
		||	logical or	2
Relational	2	>	greater than	0
		<	less than	1
		>=	greater than or equal	2
		<=	less than or equal	3
Equality	3	==	equality	0
		!=	inequality	1
		===	case equality	2
		!==	case inequality	3

Operator Type	*Codification (decimal)*	*Operator Symbol*	*Operation Performed*	*Codification (decimal)*
Bitwise	4	~	bitwise negation	0
		&	bitwise and	1
		|	bitwise or	2
		^	bitwise xor	3
		^~	bitwise xnor	4
		~^	bitwise xnor (2nd op)	5
Reduction	5	&	reduction and	0
		~&	reduction nand	1
		|	reduction or	2
		~|	reduction nor	3
		^	reduction xor	4
		^~	reduction xnor (1st op)	5
		~^	reduction xnor (2nd op)	6
Shift	6	>>	right shift	0
		<<	left shift	1
Concatenation	7	{ }	concatenation	0
Replication	8	{ { } }	replication	0
Conditional	9	?:	conditional	0

Table 2. Decimal Codifications of Operator Types and Symbols

The overall codifications in binary are:

Operator Type	***Codification (binary)***	***Operator Symbol***	***Codification (binary)***
Arithmetic	4'b0000	*	3'b000
		/	3'b001
		+	3'b010
		-	3'b011
		%	3'b100
Logical	4'b0001	!	3'b000
		&&	3'b001
		||	3'b010
Relational	4'b0010	>	3'b000
		<	3'b001
		>=	3'b010
		<=	3'b011
Equality	4'b0011	==	3'b000
		!=	3'b001
		===	3'b010
		!==	3'b011
Bitwise	4'b0100	~	3'b000
		&	3'b001
		|	3'b010
		^	3'b011
		^~	3'b100 (1st op)
		~^	3'b101 (2nd op)

Operator Type	*Codification (binary)*	*Operator Symbol*	*Codification (binary)*
Reduction	4'b0101	&	3'b000
		~&	3'b001
		|	3'b010
		~|	3'b011
		^	3'b100
		^~	3'b101 (1st op)
		~^	3'b110 (2nd op)
Shift	4'b0110	>>	3'b000
		<<	3'b001
Concatenation	4'b0111	{ }	3'b000
Replication	4'b1000	{ { } }	3'b000
Conditional	4'b1001	?:	3'b000

Table 3. Overall Binary Codifications of Operator Types and Symbols

2.1.2.6 Output

The output logic takes the ALU result and its parity and, depending on the output channel that it's selected, activates a signal that shows which is the current active data source on the output. That signal is valid for one cycle. At the same time with the output channel validation, output data and parity should be valid on the data bus.

3. VERIFICATION

3.1 Verification environment architecture

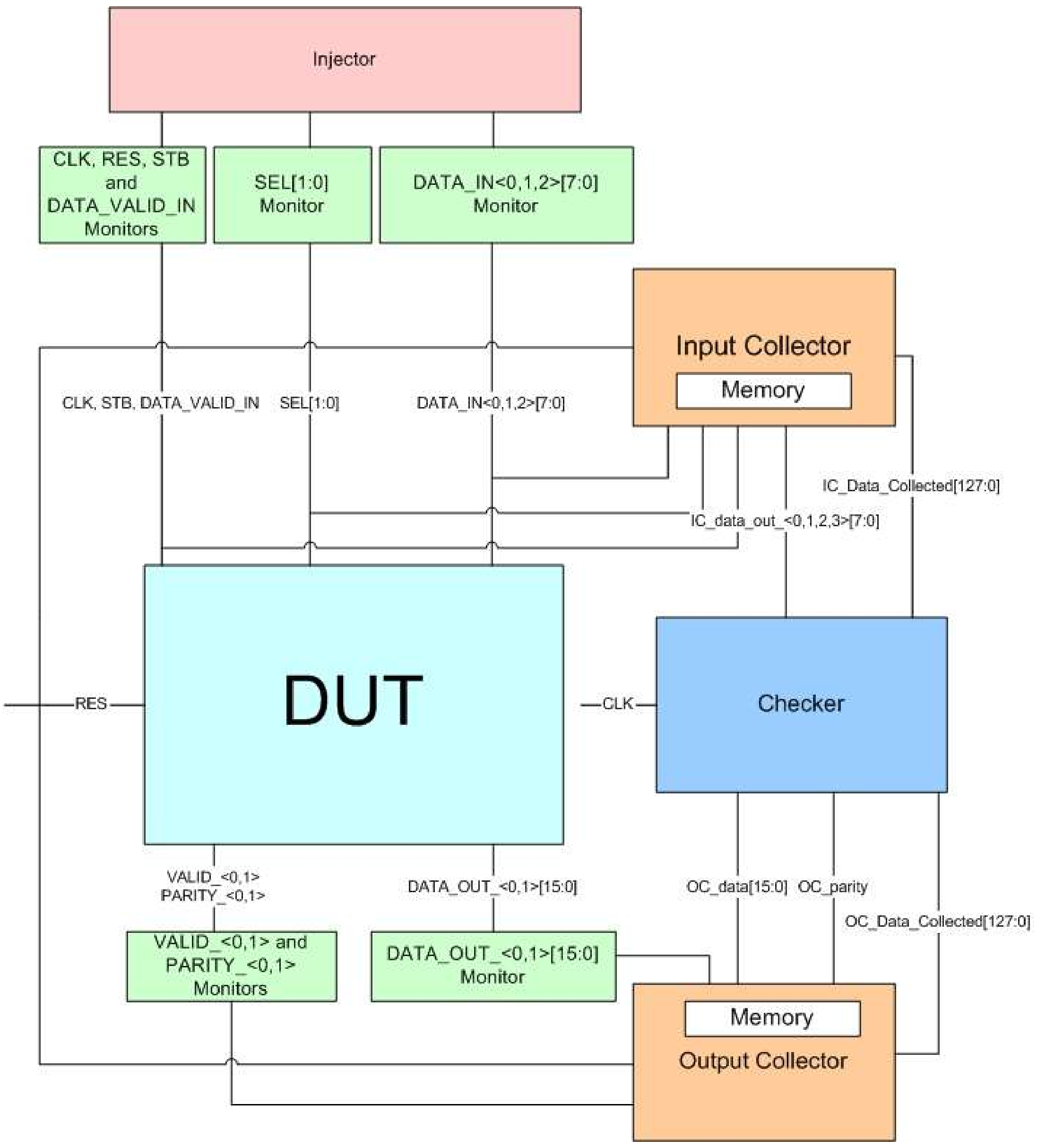

Fig. 2 – Verification environment architecture for "ALU with selectable inputs and outputs"

3.2 Verification environment description

As one can see in the above image, the verification of a circuit (called DUT – Design Under Test) is done using several verification components, built according to their role.

3.2.1 DUT

DUT is the design itself. In our case, it represents the Verilog code that was used to create the core with its hierarchy. In DUT are instantiated the modules that compund the circuit: input logic (SELECTOR), ALU and output logic (DMUX).

3.2.2 Injectors

Also call BFMs (Bus Functional Model), mainly provides input data. They control the signals that must be sent to the DUT. To simplify things, one can say that the injector of the present project is 100% active because it only transmits data and it doesn't have waiting times conditioned by design signals. But in reality, because of complexity of digital circuits, there are no 100% active injectors or 100% passive injectors. In our case, the injector controlls the following signals: CLK, RES, STB, SEL, DATA_VALID_IN, DATA_IN_0, DATA_IN_1 and DATA_IN_2.

3.2.3 Monitors

They follow the values of the signals or buses and check if they don't break the rules described in specifications.

- Ex:
  - STB, RES, SEL must not be 'X' or 'Z' during the simultation
  - DATA_OUT_0, PARITY_0 must not be 'X' or 'Z' after reset
  - DATA_IN[9:0] must not contain invalid data (for ex. 1xx0_zzzz)

3.2.4 Collectors

They follow the input and output of a certain design path and they collect transactions.

Depending on what they collect, data are stored in memory/files according to the structures built based on specifications. Retained data are further transmitted to the checkers in order to check for differences that may have appeared.

They have 100% knowledge about the communication protocol.

3.2.5 Checkers

They receive data from collectors (in this case from the collectors from the input and output of the core) and check if output data was matched to input data according to specifications.

They compare the design's with their result by using their own calculations. Also, they check for “lost” transactions or for transactions in progress.

They have no knowledge about the communication protocol.

Considering our core, the design verification environment contains:

- an injector that provides input data and values for the control signals
- monitors for all the signals
- one collector for input
- for simplicity, just one collector data outputs; this one will memorize the output according to the validation signal
- a checker for inadvertencies checking: input vs. output; it will caluclate the result according to the decoded input bits, the same way as ALU should do it

Transactions' transmitting to the checker will be done using IC_Data_Collected[127:0] (IC = Input_Collector) and OC_Data_Collected[127:0] (OC = Output_Collector) signals. Each collector will have a 16 bytes memory. Signalling of a transaction's collection will be done by asserting the corresponding bit of IC(OC)_Data_Collected[i].

Data will be effectively transmitted using special busses created between collectors and checker.

The checker will check on every clock if IC_Data_Collected[i] == OC_Data_Collected [i] == 1. If this condition is true, then it will perform data comparison between the circuit's input and output.

The circuit should be checked by using both valid (good machine) and invalid (bad machine) data.

One should use two levels of severity: WARNING and ERROR.

3.2.6 Other components

There are other components of a verification environment that are not depicted above. Such an environment should contain a script for running tests and Makefiles for compiling the source code.

3.3 Methodologies applied for a design's test

When running tests in order to verify digital designs, there are a few approaches:

3.3.1. Directed tests

They are useful only when verifying small and very small circuits in order to fully cover all the input and output combinations. It is useful when covering test values that were not reached in regressions.

3.3.2 Random tests

They are a better solution compared to directed tests because data is generated randomly. That way, the possibility of introducing into the design transactions the engineer wasn't thinking about is higher. The amount of code to be written is smaller compared to a directed test. Ideally, the verification you use should offer a constraint solver for you to create complex scenarios (constrained randomizations, dependable randomizations, several randomization phases etc).

3.3.3 Complex/improved tests

Considering the complexity of today's digital circuits, the use of directed tests or random tests that simplify code for directed tests is not effective. Visually checking of inputs and outputs is time consuming and inefficient. The need for better tests and automatic ways of checking a design in obvious.

Being given these considerations (there are many more), I created an improved test that fulfills the above requirements. This test type uses random generated transactions, instantiates the verification components I talked about (the other tests don't use them at all) and writes the results into files. I chose to give “.out” extension to the output files in order for the users to recognize the verification components' logs. You don't need to check only visually the circuit's behaviour now.

This is the type of test you should use for a proper verification.

4. FILES

4.1 List of project's files and directories

List of project's files and directories in alphabetical order:

1. **doc**
   - alu.png
   - ASICArt.jpg
   - Design_and_verification_env.jpg
   - Design_and_verification_env.vsd
   - dmux.png
   - dut.jpg
   - dut.jpg
   - dut.vsd
   - selector.png
   - Spec.sxw
   - Spec.pdf
   - Spec.doc
2. **makefile**
   - Makefile
3. **rtl**
   - alu.v
   - dmux.v
   - dut.v
   - selector.v
4. **run**
   - checker.out
   - clk_gen.out
   - clk_monitor.out
   - data_in_bfm.out
   - data_in_monitor.out

- data_out_monitor.out
- data_valid_in_monitor.out
- directed_test.v -> ../tests/directed_test.v
- improved_test.log
- improved_test.v -> ../tests/improved_test.v
- input_collector.out
- Makefile -> ../makefile/Makefile
- output_collector.out
- parity_0_monitor.out
- parity_1_monitor.out
- random_test.v -> ../tests/random_test.v
- res_bfm.out
- res_monitor.out
- RunTest.pl -> ../scripts/RunTest.pl
- sel_monitor.out
- stb_monitor.out
- sv_files -> ../sv_files
- valid_monitor.out
- verilog.log
- **waves**
    - **waves_directed_test**
    - **waves_improved_test**
    - **waves_random_test**

5. **scripts**
    - *RunTest.pl
6. **sv_files**
    - simvision_directed_test.sv
    - simvision_improved_test.sv
    - simvision_random_test.sv
7. **tests**
    - directed_test.v
    - improved_test.v
    - random_test.v

8. **verif_env**
   1. **bfms**
      - clk_gen.v
      - data_in_bfm.v
      - res_bfm.v
   2. **checker**
      - checker.v
   3. **collectors**
      - input_collector.v
      - output_collector.v
   4. **monitors**
      - clk_monitor.v
      - data_in_monitor.v
      - data_out_monitor.v
      - data_valid_in_monitor.v
      - parity_monitor.v
      - res_monitor.v
      - sel_monitor.v
      - stb_monitor.v
      - valid_monitor.v

5. RUNNING THE PROJECT

Before actually running the test, you may have to do these things first:

– specify for your operating system where the Verilog simulator executable file is
– modify in makefile/Makefile the RUN_COMMAND variable, depending on the name of your Verilog simulator

1. Create a separate directory for this project
2. **cd <work_dir>**
3. **cd run** – enter the "run" directory of the project
4. **RunTest.pl <test_name>** - it may be directed_test.v, random_test.v or improved_test.v
5. look at .log files or .out files - depending on which type of test you ran - using a text viewer

6. EXAMPLE WAVEFORMS

For a better visualization of the waveforms, please look at the .png files located in the “doc” directory of the project.

6.1 SELECTOR

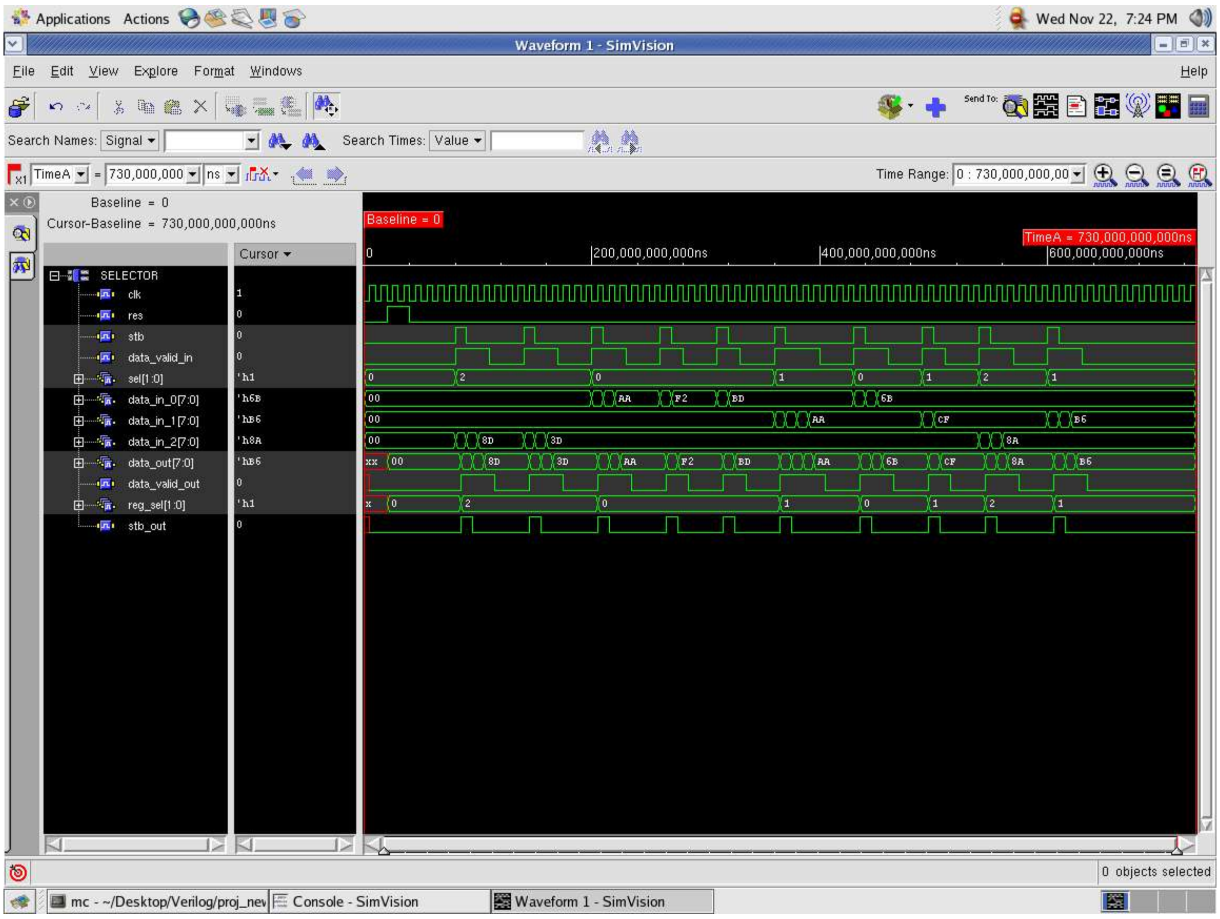

Fig. 3 – SELECTOR module waveforms

6.2 ALU

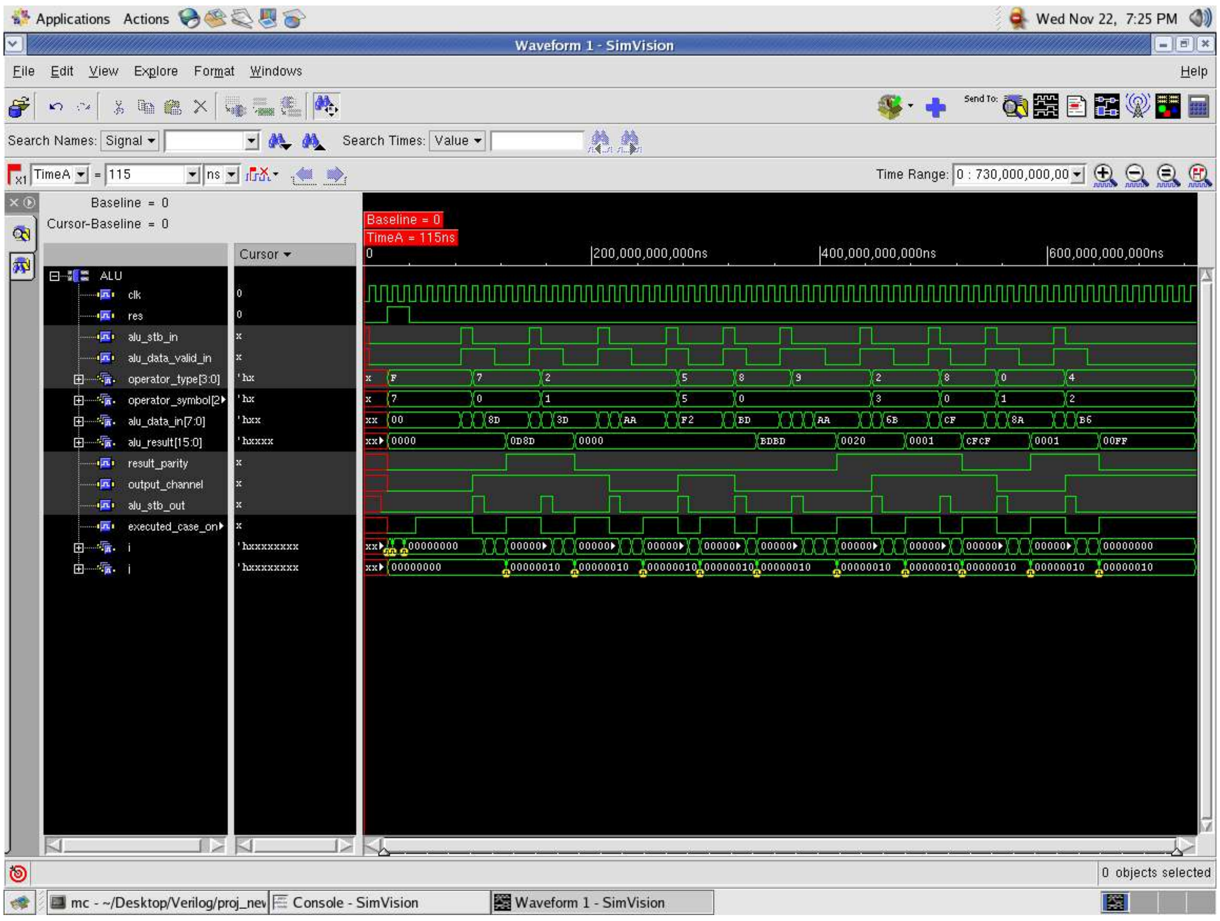

Fig. 4 – ALU module waveforms

6.3 DMUX

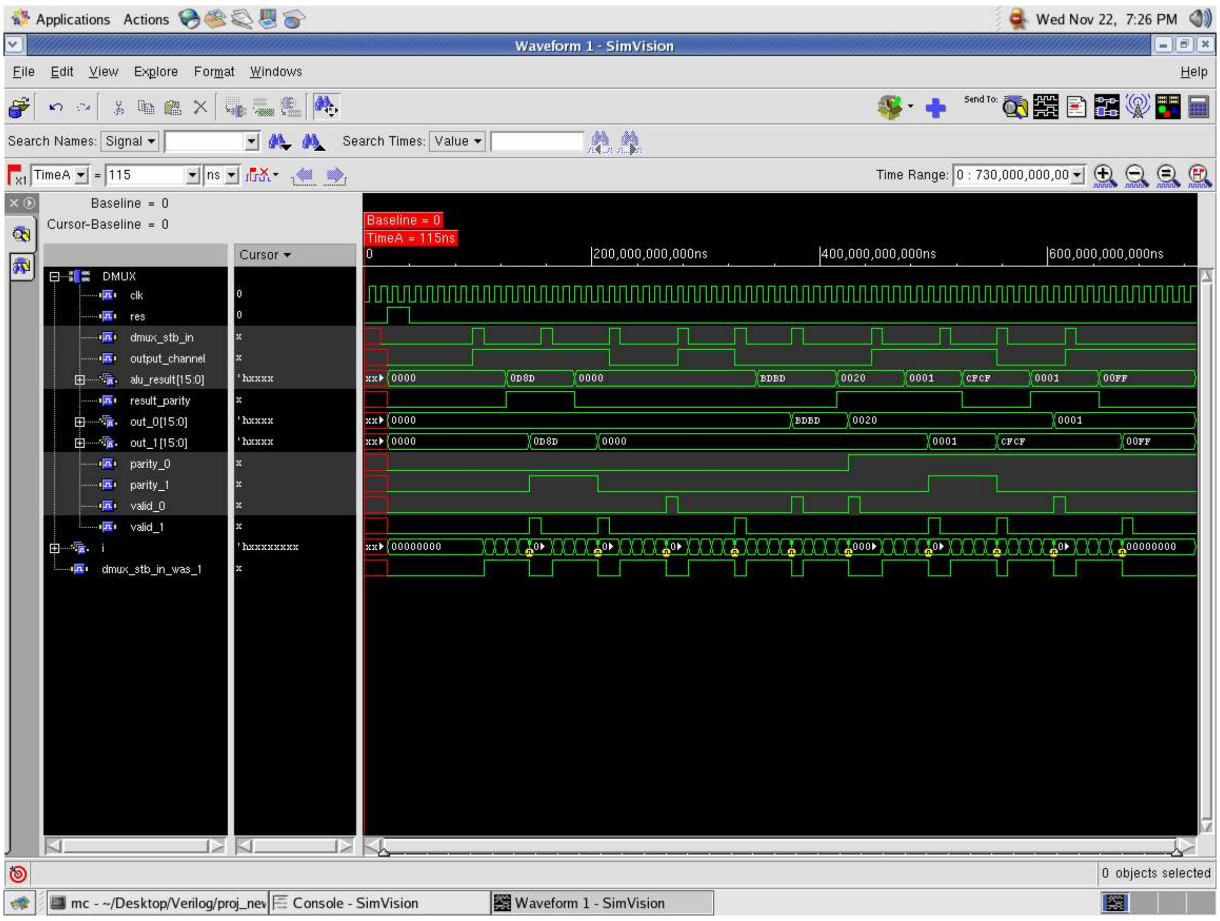

Fig. 5 – DMUX module waveforms

7. EXERCISES

The exercises I propose may also be considered future steps in improving this project:

1. Make the circuit work with any number of operands for operations like Concatenation and Replication
2. Modify the existing verification environment so one can run tests with a user given seed.
3. Build a regression script in Perl to run different random test types with different random seeds.

8. REFERENCES

1. Palnitkar, Samir - "Verilog HDL – A Guide to Digital Design and Synthesis"
2. http://www.hsrl.rutgers.edu/ug/make_help.html

9. ERRATA