DSD EXERCISE

ARITHMETIC AND LOGICAL OPERATORS IN VHDL

AARHUS UNIVERSITY
SCHOOL OF ENGINEERNG
PHM, CEF

JUNE 2018

Document History

2015-09-25: PHM, initial version.
2018-01-26: CEF, converted to LaTeX.
2018-02-17: CEF, fixed comma error in shift_div entity.
2018-06-07: CEF, added DSP figure made some DSP setup clarification.
2018-08-30: CEF, fixed dash in heading.
2018-10-18: CEF, moved some figues and tables.

Goals

The goals for this exercise are:

- Use signals and the composite type `std_logic_vector`.
- Use and understand the signed and unsigned logic types.
- Use most common arithmetic operators.
- Use the concatenation operator for shift and rotate operations.

Prerequisites

- Have Quartus II up and running.
- That you have read THE DSD EXERCISE GUIDELINES!

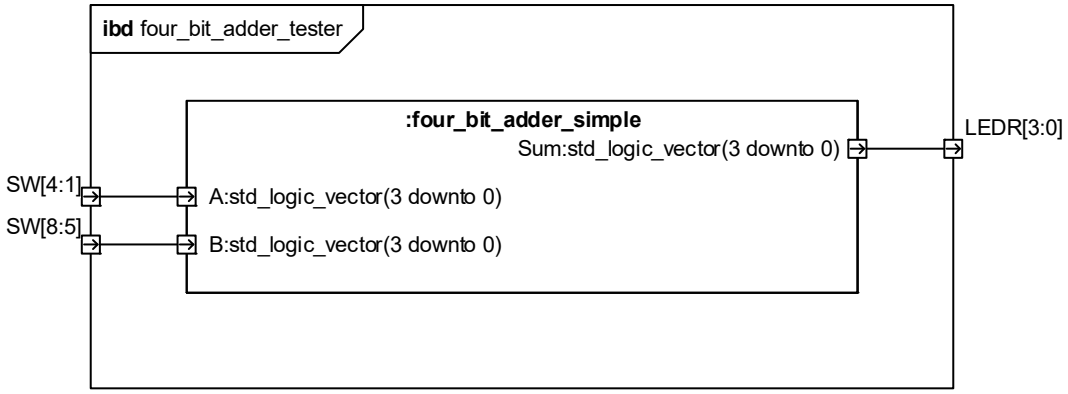

Fig. 1: *4-bit adder simple.*

1 Signed and Unsigned Arithmetic

VHDL has built-in operators for arithmetic that make use of ex. adders a lot simpler, than what we experienced in the previous exercise. The `numeric.std` library allows you to perform unsigned and signed arithmetic. `std_logic` or `std_logic_vector` types are typically used for ports. These types represent raw data and no information about its numeric representation (signed or unsigned). In you solution you must be careful to cast the `std_logic_vectors` to signed or unsigned, to use the appropriate implementation found in `numeric.std`. To use arithmetic, you must include the source code table 1.

a) Start out by implementing a 4-bit unsigned adder with an interface as depicted in figure 1. Use the '+' operator and the appropriate `unsigned()` / `std_logic_vector()` functions to cast between types. Name your architecture "`unsigned_impl`".

b) Verify your solution using functional simulation, note the output of the simulation, how does it comply? You may use the input waveform from previous exercises or create a new.

c) Create a tester and Instantiate the new adder in it. You can use assignment->import assignments to import the `de2_assignments.qsf` found on Black Board file sharing (tool->altera).

d) Build your project and test it on the DE2 board. You can have multiple architectures to a single design entity. This allows you to have different implementations for the same design entity, we try this next.

e) Create a new architecture, "`signed_impl`", in your design entity (vhdl file). Create a signed version of the adder implemented in your "`unsigned_impl`" archi-

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
```

Tab. 1: *VHDL header for arithmetics.*

```
i1: entity work.four_bit_adder_simple(signed_impl)
  port map(
    A => SW(4 downto 1),
    B => SW(8 downto 5),
    sum => LEDR(3 downto 0)
  );
```

Tab. 2: *4-bit adder entity instantiation.*

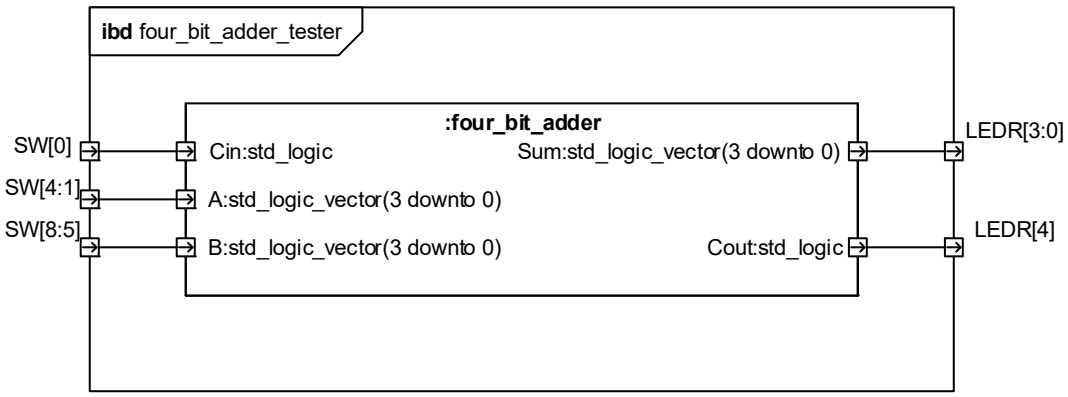

Fig. 2: *4-bit adder with carry.*

tecture. In the tester you can now select this specific implementation using the syntax found in source code table 2.

f) Build and download. How does the binary output compute? How does it compare with the unsigned version? Why? When do you think it can be beneficial to have multiple architectures (implementations) for a single design entity?

g) The adder in figure 1 easily overflows. We need additional carry in- and out logic to support larger numbers. You must extend the unsigned adder to support this as depicted in figure 2. Note that you must use the `resize()` function to change the bit size of vectors before adding them together. Also note that `std_logic` ports are not vectors and therefore cannot use `resize()`, concatenation must be used instead.

Update the tester. Build, download and test. Modify the signed version and compare the results. How do they compare with the previous results? Do you see any effect of using signed/unsigned? Why?

2 Concatenation

`Std_logic_vector` is an array type of `std_logic`. The concatenation operator '`&`' allows you to manipulate arrays by appending bits together to form arrays. This is similar to the C++ stream '<<' operator (`&&` in java/Phyton). In this case it is just bits rather than streams that we are working on. Given the interface in the source code table 3, it is your job to implement shift and rotate operations using the concatenation operator.

a) Implement the following functionality using concatenation (`&` operator):

```
entity shift_div is
  port (
    a : in std_logic_vector(7 downto 0);
    a_shl, a_shr, a_ror: out std_logic_vector(7 downto 0)
  );
end;
```

Tab. 3: *Interface definition for the shift entity.*

`a_shl`: Shift a one-time to the left.

`a_shr`: Shift a two times right.

`a_ror`: Rotate a three times right.

b) Check the implanted design in the Technology Map Viewer and note the number of LEs used. How are these kind of operations implemented in an FPGA?

c) Download and test the design. Use red and green led's for output and switches for input. Remember to take a photo for documentation.

3 Multiplication

The VHDL standard supports all normal arithmetic operators. You can investigate the library file, which is in plain VHDL, to see which operators are implemented for which types. The library file is located at:

`C:\altera\13.0sp1\quartus\libraries\vhdl\ieee\numeric_std.vhd`

(assuming `C:\altera\13.0sp1\` as install directory) Using the operators comes at a cost: resource usage! Addition and substraction is cheap, one LE per bit, as we have learned already. But how about multiplication and division?!

The FPGA has built-in DSP blocks that it will seek to use when feasible. The DSP blocks can perform multiplication/division, but is a limited resource in the FPGA. To investigate how multiplication is implemented with logical elements in the FPGA, we'll turn off the DSP blocks for now. In the Assignments Editor add the assignment shown in table 4, also take a look at figure 3 that shows a setup for a particular solution; your entity names will different.

This will force the fitter to use Logic Elementes, even for functions that could be implemented in DSP blocks. For multiplication this means that it will be implemented using additions. We'll now create a new component that performs multiplication of two inputs using the '*' operator.

Status	From	To	Assignment Name	Value	Enabled
		*	DSP Block Balancing	Logic Elements	Yes

Tab. 4: *Turn off the FPGA DSP block in the Assignment Editor. Open the 'Assignment Editor' in Quartus, goto the end in the table and put in a new entry; the Enable column means if an rule is active and the Entity column must contain the name of your particular entity name, see also figure 3.*

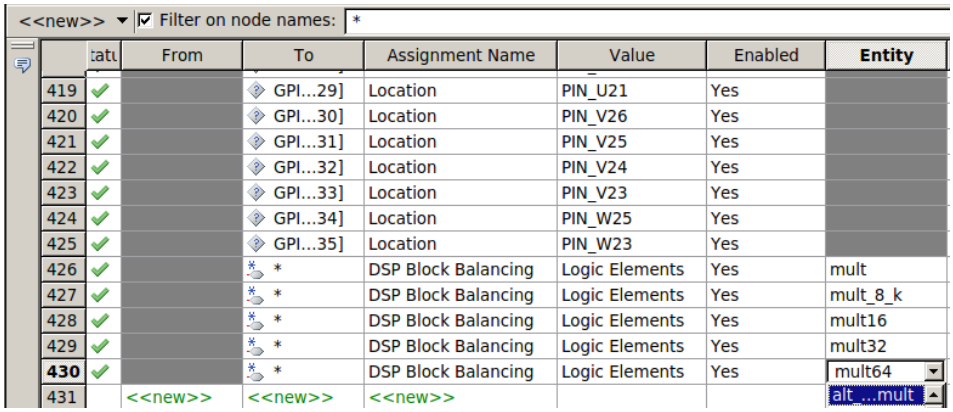

Fig. 3: *A dump of my 'Assignment Editor' when disabling the DSP block for some particular entities, like 'mult' or 'mult_8_k'.*

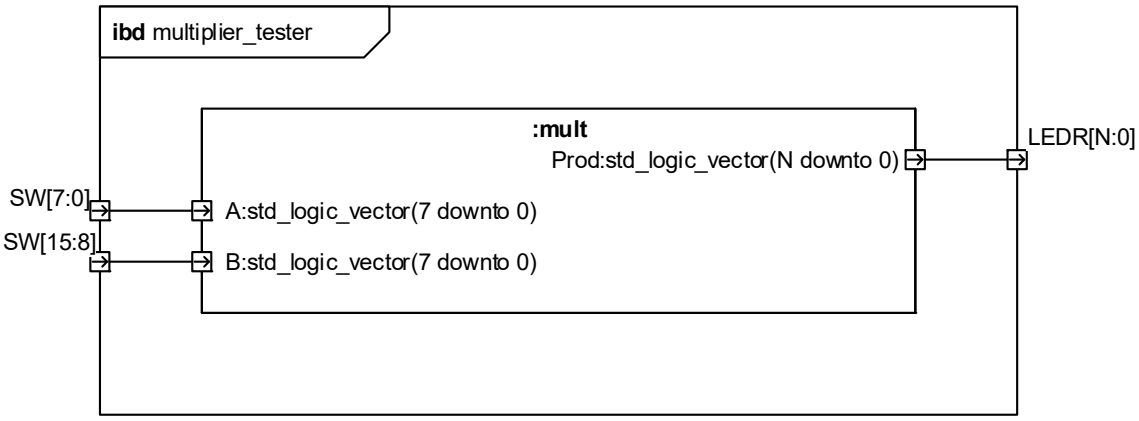

Fig. 4: *Multiplier.*

a) Create a component with the interface depicted in figure 4. The component must multiply the two inputs and output the result. Use the '*' operator and dataflow style. How does the number of bits on the in- and output correlate?

b) Compile and test the multiplier on the DE2-board. Note the number of Logical Elements used (compilation report).

c) Modify the multiplier to use other bit sizes (32,16,8,4,3,2,1), compile, and note the number of logical elements used respectively. Plot the number of bits versus LEs used. How do they correlate? Linearly? How does it scale? Does it correlate with the number of bits, the number of additions required, or..? (Hint: Binary Multiplier).

d) Multiply 'A' with constants: (2,3,4,7,8,10). The '*' operator allows you to multiply with an integer directly. Again, note the number of LEs used. How does it correlate? Are there special cases where multiplication becomes really cheap, if so, why?