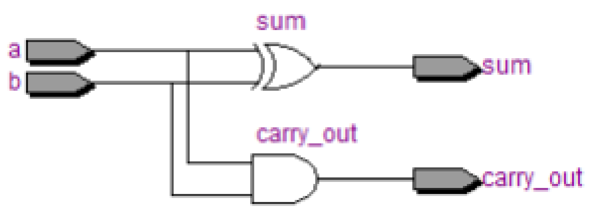
DSD Exercise



Arithmetic- and Logical Operators in VHDL

21-9-2015/PHM

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!!!

THE EXERCISE ITSELF

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_vector*s to *signed* or *unsigned*, to use the appropriate implementation found in numeric.std.

To use arithmetic, you must include:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

1) 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".

2) 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.

3) 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).

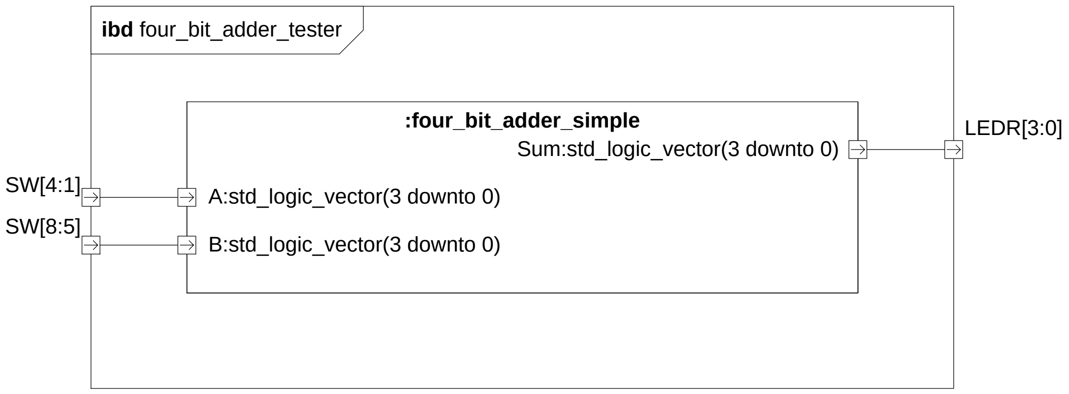

ibd four_bit_adder_tester
:four_bit_adder_simple
Sum:std_logic_vector(3 downto 0)
LEDR[3:0]
SW[4:1]
A:std_logic_vector(3 downto 0)
SW[8:5]
B:std_logic_vector(3 downto 0)

FIGURE 1: 4-BIT ADDER SIMPLE

4) 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.

5) Create a new architecture, "signed_impl", in your design entity (vhdl file). Create a signed version of the adder implemented in your "unsigned_impl" architecture. In the tester you can now select this specific implementation using the following syntax:

```
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));
```

6) 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?

7) 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.

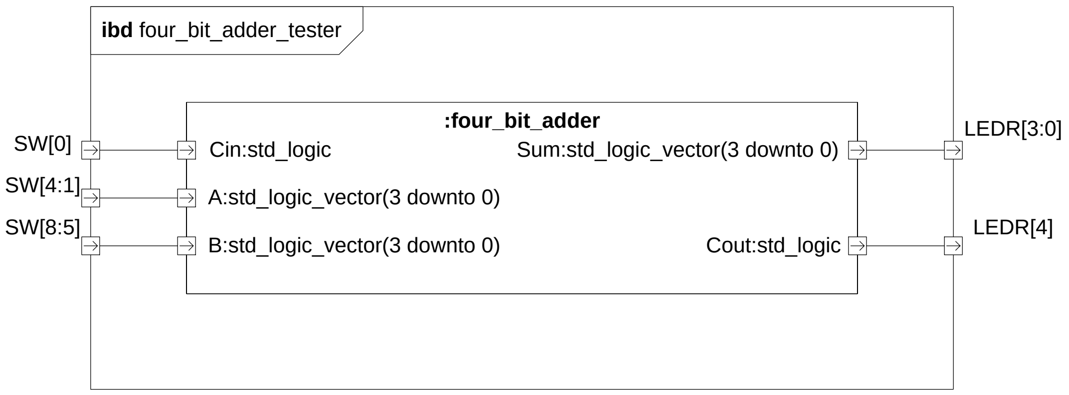

ibd four_bit_adder_tester
:four_bit_adder
SW[0]
Cin:std_logic
Sum:std_logic_vector(3 downto 0)
LEDR[3:0]
SW[4:1]
A:std_logic_vector(3 downto 0)
SW[8:5]
B:std_logic_vector(3 downto 0)
Cout:std_logic
LEDR[4]

FIGURE 2: 4-BIT ADDER WITH CARRY

8) 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 snippet below, it is your job to implement shift and rotate operations using the 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 shift_div;
```

1) Implement the following functionality using concatenation (& operator):

   a_shl: Shift a one-time to the left.

   a_shr: Shift a two times right.

   a_ror: Rotate a three times right.

2) 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?

3) 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:

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

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.

1) Create a component with the interface depicted in figure 3. 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?*

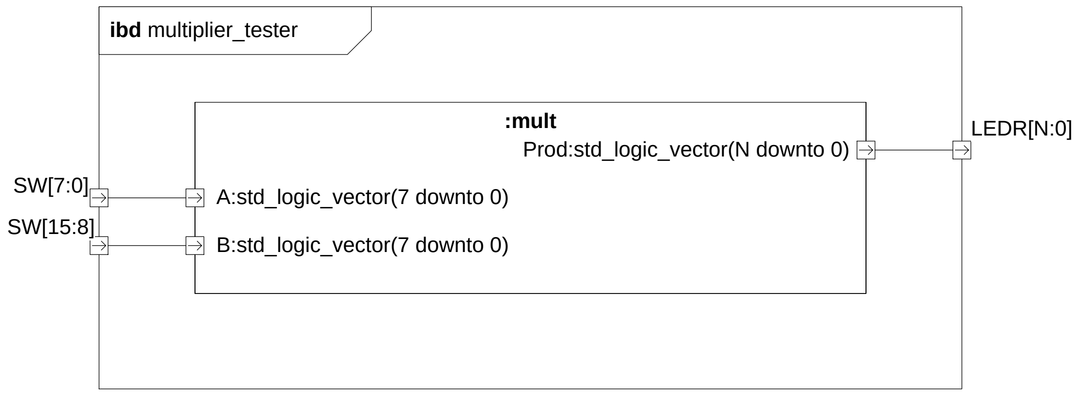

ibd multiplier_tester
:mult
Prod:std_logic_vector(N downto 0)
LEDR[N:0]
SW[7:0]
SW[15:8]
A:std_logic_vector(7 downto 0)
B:std_logic_vector(7 downto 0)

FIGURE 3: MULTIPLIER

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

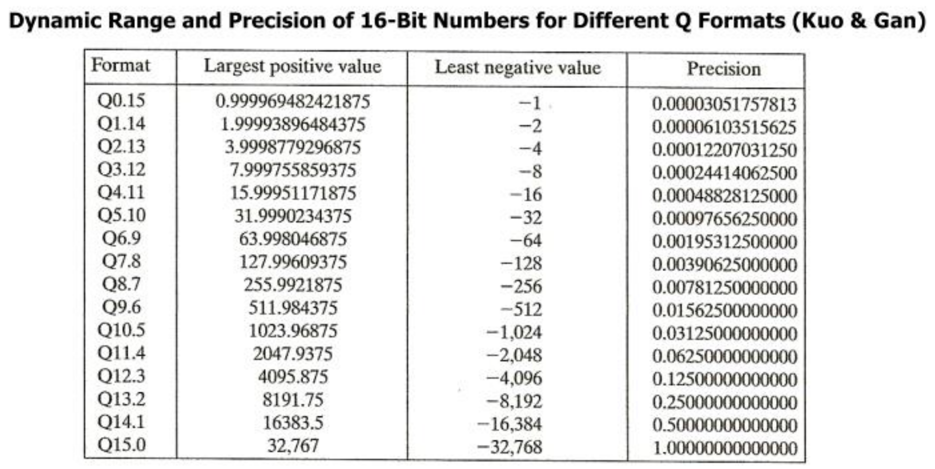
3) Modify the multiplier to use other bit sizes (7,6,5,4,3,2,1), compile, and note the number of logical elements used respectively. Plot the number of bits versus LEs used. *How do they correlate? Linearly? How does it scale? Does it correlate with the number of bits, the number of additions required, or..? (Hint: Binary Multiplier)*

4) 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?*

4) Fractional Fixed-Point Arithmetic

Fractional representation of numbers allows you to use fixed-point arithmetic on decimal numbers. It is often desired to use fixed-point arithmetic over floating-point due hardware resource demands (and hence to power consumption). Using floating point operations on a CPU that does not have a dedicated floating point unit (FPU), will slow down your computation significantly, because the CPU's ALU will have to be emulated it. Fractional numbers does not maintain full dynamic range like true floating point, but for most DSP applications it will suffice.

The fractional number format, also known as the Q-format (See Wikipedia: Q number format), uses the notation Qm.n, where m is the number of integer bits and n is the fractional portion. For signed numbers *m* may also contains the sign bit.



Dynamic Range and Precision of 16-Bit Numbers for Different Q Formats (Kuo & Gan)

Format	Largest positive value	Least negative value	Precision
Q0.15	0.999969482421875	−1	0.00003051757813
Q1.14	1.99993896484375	−2	0.00006103515625
Q2.13	3.9998779296875	−4	0.00012207031250
Q3.12	7.999755859375	−8	0.00024414062500
Q4.11	15.99951171875	−16	0.00048828125000
Q5.10	31.9990234375	−32	0.00097656250000
Q6.9	63.998046875	−64	0.00195312500000
Q7.8	127.99609375	−128	0.00390625000000
Q8.7	255.9921875	−256	0.00781250000000
Q9.6	511.984375	−512	0.01562500000000
Q10.5	1023.96875	−1,024	0.03125000000000
Q11.4	2047.9375	−2,048	0.06250000000000
Q12.3	4095.875	−4,096	0.12500000000000
Q13.2	8191.75	−8,192	0.25000000000000
Q14.1	16383.5	−16,384	0.50000000000000
Q15.0	32,767	−32,768	1.00000000000000

TABLE 1: UNSIGNED FRACTIONAL NUMBERS

As you can see in table 1, the Q-format allows you to use different ranges and resolutions with the same amount of bits, depending on your system needs. Range is $[-(2^m), 2^m-2^{-n}]$, resolution is 2^{-n}. Number of bits is n + m + sign bit.

For example 3.11 + 6.43 = 9.54 could be represented using fractionals. Representing 9.54 we'll need 2^6 (64>54) combinations for the fraction and 2^4 (16>9) to represent the integer part. A suitable format is therefore Q4.6. This results in 4+6+1 = 11 bits.

To convert floats to fractional you simply scale by the number of fractional bits (sign bit in parenthesis).

$$3.11 * 2^6 = 199 = (0)0011.000111$$

$$+\ 6.43 * 2^6 = 412 = (0)0110.011100$$

$$= 611 = (0)1001.100011 \implies 611 * 2^{-6} = 9.54$$

See https://www.altera.com/en_US/pdfs/literature/an/an083_01.pdf