CHALMERS
Department of Computer Science and Engineering
2010-08-30

DAT091 Introduction to Electronic System Design
Laboratory assignment 2
Basic arithmetics

Introduction

Many of our FPGA applications involve manipulation of data using the four elementary mathematical functions addition, subtraction, multiplication and division. In this assignment we will have a look at different ways to implement three of these functions in hardware. We will exclude division.

Preparations

Prepare the assignments by sketching the structure of your code. What are the inputs and outputs? Separate the code into suitable blocks. If the code is sequential, draw a flowchart. Think of how to test your code thoroughly, select suitable input values, and write `do` files for the simulation of your designs in **QuestaSim**.
In all the designs make sure they work properly for both positive and negative numbers, that is for signed numbers. Make the solutions general by using generics to define the word lengths so that you don´t have to rewrite the whole code if you want to change the word length.

Test and simulation

Plan in advance how to simulate your designs and how to check that the results are correct. To do this find a series of input values for the simulation that can check a sufficient number of results and use these values when you further on (lab assignment 4) write test benches for your designs. Make sure that your tests cover the combinations of data values that can occur, for example different combinations of positive and negative values that give positive and negative results. When you test saturation and overflow make sure to test both the upper and the lower limit of the allowed value span.

CHALMERS UNIVERSITY OF TECHNOLOGY
Sven Knutsson
Department of Computer Science and Engineering
Division of Computer Engineering
Mail address: SE-412 96 Gothenburg
Sweden
Email: svenk@chalmers.se
Web: www.chalmers.se

Prepare how to test your design in advance. We test our designs by simulating them in **QuestaSim**. To be able to demonstrate and reproduce your results you shouldn´t add your signals to the simulation by hand and you shouldn´t input stimuli by hand but you should do both things using a **QuestaSim** batch file, a do file.

2.1 Addition

2.1.1Vector addition

Use behavioral VHDL to implement an adder that adds two std_logic vectors a and b of length n to give an output vector y of the same length as a result, *Figure 2.1*. Remember to make the design general by using generics. Hint: Start by using a fixed word length, for example 8 bits and then move to a generic approach.

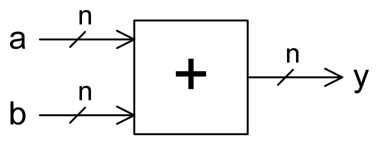

Figure 2.1 Vector adder

2.1.2 Structural adder

2.2.1.1 Ripple carry adder

Start by writing the code for a one bit full adder, *Figure 2.2*.
Use this full adder as a component that is reused a number of times to implement the n bit adder in *Figure 2.1* as a ripple carry adder. It takes one bit of extra thought to realize how to do this using generics.

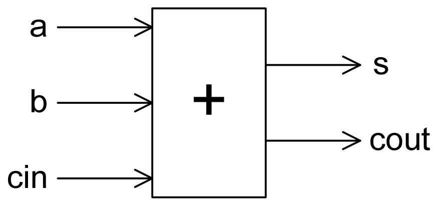

Figure 2.2 One bit full adder

2.1.2.1.1 Testing the ripple carry adder

Write a do file that tests the design for different combinations of positive and negative input vectors that should give both positive and negative results. Test what happens at overflow, that is if the result is too large to fit (positively or negatively) within the number of bits set by the generic constant.

2.1.2.2 Overflow indication

Extend the code with an overflow bit to indicate if the result is too large to fit (positively or negatively) within the number of bits set by the generic constant, *Figure 2.3*.

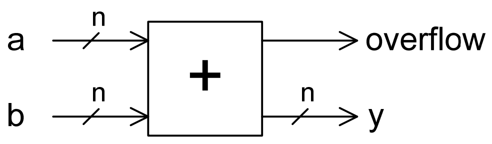

Figure 2.3 Vector adder with overflow

2.1.2.2.1Testing the ripple carry adder with overflow indication

Test the ripple carry adder with overflow indication the same way as you tested the ripple carry adder in *Paragraph 2.1.2.1.1*.

2.1.2.3 Ripple carry adder with saturate/wrap around control

Keep the overflow bit and introduce a new control bit that makes it possible to choose what should happen if the result overflows, *Figure 2.4*. When the bit is set the result should saturate at overflow and when it is cleared the result should wrap around.

2.1.2.3.1Testing the ripple carry adder with saturate/wrap around control

Test the ripple carry adder with saturate/wrap around control the same way as you tested the ripple carry adder in *Paragraph 1.2.1.1* but do the tests twice, once with the control bit set for wrap around and once with the control bit set for saturation.

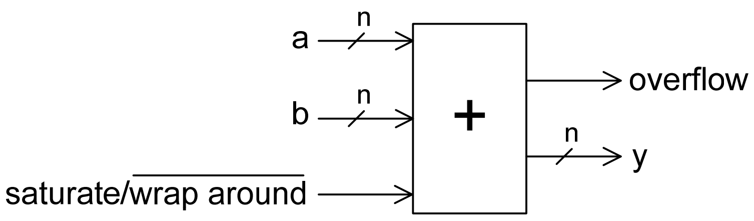

Figure 2.4 Vector adder with saturate or wrap around

2.2 Subtraction

2.2.1 Ripple adder/subtracter

Rewrite the code from *Paragraph 2.1.2.3* to implement a circuit that can do both addition and subtraction. The choice of mathematical function is done using a control bit, *Figure 2.5*.

You can do this by using the one bit full adder in *Figure 2.2* and some extra logic or and by designing a one bit full adder/subtracter, *Figure 2.6*. The first approach is recommended.

Hint: You can use the fact that subtraction can be implemented as the addition between a and the 2-complement of b?

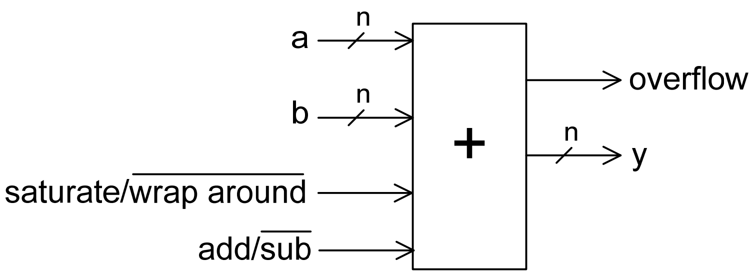

Figure 2.5 Vector adder/subtracter

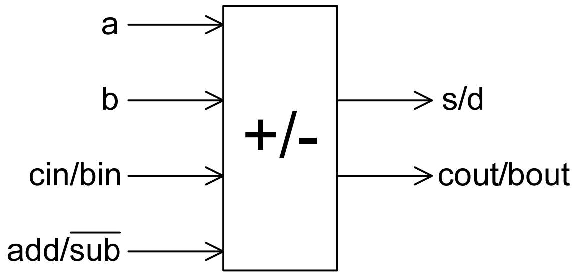

Figure 2.6 One bit full adder/subtracter

2.2.1.1 Testing the ripple adder/subtracter

Test the ripple adder/subtractor the same way as you tested the ripple carry adder with saturate/wrap around control but do it both in addition and in subtraction mode.

2.2.2 Serial adder/subtracter

We can serialize the adder/subtracter by introducing a clock to control the flow and use this to shift the input vector bits into the adder/subtracter and use one single one bit ad-

der/subtracter (or a single one bit adder with some extra logic) to calculate the result bit by bit, *Figure 2.7*.
To make the code work you will need a control bit to start the calculation and it is also good to have an indicator bit to tell when the calculation is finished. Don´t output any temporary results, that is don´t update the output before the calculation is finished. The interface of the serial adder/subtracter should follow *Figure 2.8*.

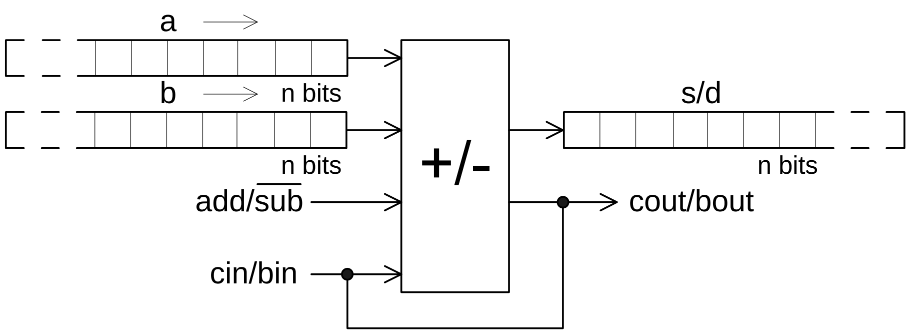

Figure 2.7 Serial adder/subtracter

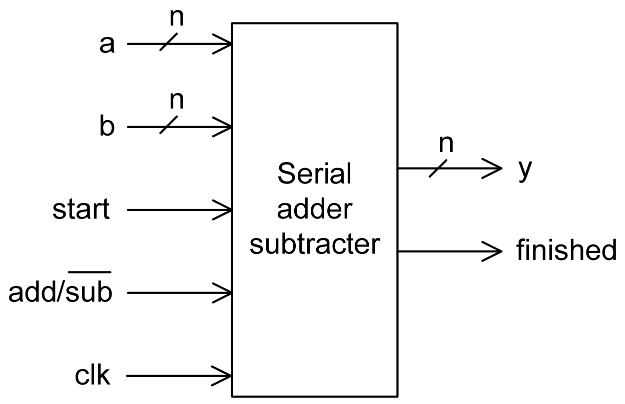

Figure 2.8 Interface of the serial adder/subtracter

2.2.1 Testing the ripple adder/subtracter

Test the ripple adder/subtracter the same way as you tested the ripple carry adder with saturate/wrap around control but do it both in addition and in subtraction mode.

2.3 Multiplication

2.3.1 Vector multiplication

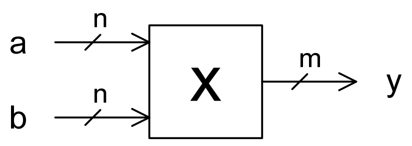

Figure 2.9 One bit full adder/subtracter

Use behavioral VHDL to implement an eight (8) bit multiplier for the multiplication of two vectors a and b to give the output vector y as a result, *Figure 2.9*. The function should work for both positive and negative numbers.
How many bits do you need for the result to make sure that you have full accuracy? Is there any difference if

you interpret your numbers as integers or as fractional numbers?

2.3.1.1Testing the multiplier

Test the multiplier by writing a `do` file that gives different combinations of positive and negative values as input stimuli.

2.3.2 Multiplication by successive addition

We can implement the multiplier in a serial way similar to the way we calculate multiplication using pen and paper. We take digits one by one from vector b and multiply vector a with that digit, shift the result to the right to give it the weight of the current digit from b and add it to the accumulated result. Since the digits from vector b in the binary case only can be one (1) or zero (0) we don´t need to do the multiplication but it is only a question of deciding if the value should be added to the accumulating sum or not, not forgetting the shift, *Figure 2.10*. We call this *multiplication by successive addition*.

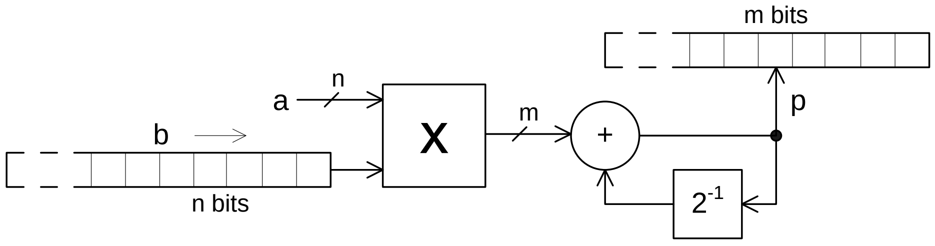

Figure 2.10 Multiplication by serial addition

Implement a multiplier that uses successive addition to implement the function. Since we are doing the calculation bit by bit we need to introduce a clock to control the calculation flow. We will also need a signal to start the calculation and it is recommended to have an indicator to tell when the result is finished. Don´t output any temporary results, that is don´t update the output before the calculation is finished.

The interface of the serial adder/subtracter should follow *Figure 2.11*.

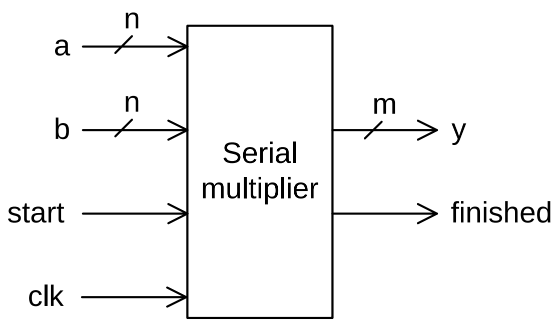

Figure 2.11 Interface of the serial multiplier

2.3.2.1 Testing the multiplier

Test the multiplier by writing a `do` file that gives different combinations of positive and negative values as input stimuli. You will of course also need to include the clock signal, the start signal and the finished signal in the test.

Conclusions

We have studied a number of ways to implement the elementary mathematical operations addition, subtraction and multiplication. Some of the implementations are parallel while others are serial. If we only need a few operations the parallel approach might be the preferable choice since most FPGA circuits have some dedicated adder/subtracters and multipliers but if

we need more operations they will have to be created from scratch. By serializing the design we can save hardware but we will lose speed.
How do the different approaches affect the throughput and latency of the circuit? Think about the difference between throughput and latency.