Introduction to Electronic System Design (DAT093)
Lab 1: Generics and two's complement arithmetic

Sven Knutsson, Lars Svensson

Version 2.1, September 7, 2018

1 Introduction

In Lab 0, you realized some adders and counters based on the two's-complement number representation. Here, we will extend these simple arithmetic circuits in three main ways:

1. We will use `generic` to parameterize our designs.
2. We will introduce overflow detection and saturation in our adders.
3. We will construct subtraction and multiplication hardware[1] based on experiences from the adder designs.

Don't be alarmed by the number of tasks; up to and including section 6, each step is rather small—if you find yourself considering a complex design for these, you should talk to your lab TA before spending much time. The remaining tasks are somewhat larger.

2 Preparation

Read through the entire lab PM and work through all tasks tagged **Preparation**. Most of these are small pen-and-paper design tasks. (Make a habit of starting

[1]The fourth basic mathematical operation, division, is notably absent; it is cumbersome to implement and rarely occurs in practical applications.

by sketching a hardware implementation whenever you write VHDL code! Your ideas of the desired hardware will often suggest a VHDL implementation that is clear, efficient, and less likely to be buggy than if you just write VHDL code without hardware in mind.)

3 A generic adder

Preparation: Read about the `generic` concept in VHDL; we will use it to create hardware descriptions where some properties are decided by compile-time parameterization. Also read about the `for–generate` construct.

The generic adder will use a `for–generate` construct to create the array of full adders (FAs), and a `generic` parameter to set the size of that array. Begin by introducing `for–generate` in a copy of the Lab-0 adder:

- Following the same procedure as in Lab 0, create a new project called `lab1a`. Copy[2] the VHDL files, test benches, and do files for the Ripple-Carry Adder (RCA) from `lab0b` into `lab1a`.
- Edit the RCA architecture to use one `for–generate` construct to create an array of FAs. Use `VECTOR` signal types where appropriate to represent the interconnections. Test your implementation using the same test bench and *do* file as in Lab 0; the behavior should be identical to that of the previous version.

As you have seen, the `for–generate` construct specifies the index range for the FA array. You will now *parameterize* this index range:

- Edit your ripple-carry adder entity to include a `generic` parameter `WIDTH` of type `integer`, which will control the adder wordlength. In the architecture, change the index range of the `for–generate` construct to use the value of `WIDTH` to set the number of FAs. Compile your description and fix compile-time errors, if any.
- Download the test bench for the 4-bit ripple adder from the course homepage. Also download the *do* file for the test bench. Verify that your design works according to the test bench. Fix any errors.

[2]Make sure to actually *copy* the files rather than just add the existing files to this project. You will edit these files later on; if you don't copy them now, you will overwrite the originals later!

How can you verify that your design is correct also for other values of `WIDTH`?

Are there limits on the value range of `WIDTH` that will yield a correct result?

4 Overflow detection

As is well known, the value range of a sum is in general larger than that of its terms. Therefore, an adder with the same wordlength for inputs and outputs will not be able to represent the sums of all possible combinations of input values—one more output bit would be needed. When the correct output cannot be represented, *overflow* has happened.

Several approaches may be taken to this problem. The first approach is to ignore it, as the adder in Section 3 did (the carry-out bit of the most-significant FA was not made available externally); this is simple to implement and maintains the property that $(a + b) - b$ gives the expected result, even if $a + b$ is too large to be represented (another benefit of two's complement!). The obvious drawback is that the sum of two positive values may appear as a negative value; this is known as "wrap-around" behavior.

A second approach is to detect when an overflow happens in order to take some action. The canonical way to test for overflow is to compare the carry outputs of the two most-significant FAs in the RCA: if they are not identical, an overflow has occurred.

Preparation: A hardware implementation of the canonical overflow detection may be quite small. Describe how to add this detection mechanism to your adder.

The VHDL implementation should be quite simple:

- Create a project and directory named `lab1b`.
- Download the test bench for the adder with overflow detection.
- Inspect the test bench in order to determine the entity name, port names, and generic names for the adder with overflow detection.
- Make *copies* of the VHDL files for the Section-3 adder as before. Modify the entity to correspond to the test bench. Compile the entity with the test bench and fix any errors.

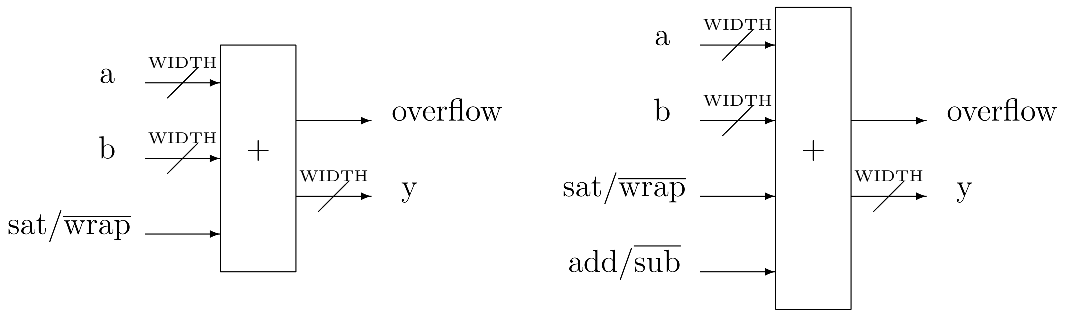
a
WIDTH
b
WIDTH
sat/$\overline{\text{wrap}}$
+
overflow
WIDTH
y
a
WIDTH
b
WIDTH
sat/$\overline{\text{wrap}}$
add/$\overline{\text{sub}}$
+
overflow
WIDTH
y

Figure 1: Illustrations of the entities for adder with optional saturation (as designed in Section 5), and for adder/subtractor (as in Section 6).

- Starting from the Section-3 adder architecture, implement the architecture for the new adder. Compile entity and architecture files with the test bench and fix any errors.
- Download the appropriate *do* file from the course home page. Simulate to verify that the behavior is as expected. Make sure to try both some input value combinations that should cause overflow and some that should not.

What are your limitations on `WIDTH` now?

5 Saturating adder

The overflow indication mechanism makes it possible to take action when overflow occurs. Here, that action will be to *saturate* the result: the output should be set to the largest-magnitude representable positive (negative) number for the given wordlength.

Preparation: What is the largest-magnitude positive and negative representable numbers in an `WIDTH`-bit two's-complement representation?

Preparation: Design hardware that will replace the sum with the saturated number when overflow is detected.

Preparation: Also introduce a one-bit input signal to select between saturating (when the bit is 1, or *set*) and wrap-around (when the bit is 0, or *cleared*) behavior.

The coding is similar to the previous examples:

- Create a project and directory named `lab1c`.
- Download test bench and *do* file for the saturating adder. Inspect the test bench to find out about name, ports, and generics of the new entity. Compare with the illustration in Figure 1.
- Create the entity for the saturating adder, for example by copying a previous file and editing it. Verify that your new entity fits with the test bench.
- Create the architecture for the saturating adder. Compile and fix any errors. Use test bench and *do* file to verify correctness.

6 Adder/subtractor

Subtraction can be viewed as an addition where one of the terms is first negated:

$$a - b \equiv a + (-b)$$

Fortunately, it is cheap to negate a two's-complement number: invert each of the bit values and add a 1 in the LSB position[3].

Preparation: Extend the saturating adder from Section 5 to also handle subtraction. An extra control signal is needed to select addition when set and subtraction when cleared. *Hint:* You will want to use the carry input of the least-significant FA.

Coding and implementation follows the now-familiar steps:

- Create a project and directory named `lab1d`.
- Download the appropriate test bench from the course homepage.
- Create the entity for the adder/subtractor, including the new port. Compare with the illustration in Figure 1. Verify that it works with the test bench.
- Implement the architecture of the adder/subtractor, starting from the saturating adder of Section 5.
- Download the appropriate *do* file and use it to verify your design.

The extra input signals have increased the number of cases that must be checked. With increasing hardware compexity, how will you handle the testing problem?

[3]`https://en.wikipedia.org/wiki/Two's_complement#Subtraction`

7 Serial adder/subtractor

The remaining sections of this lab will re-introduce the sequential (clocked) circuits encountered in the last part of Lab 0. The first task is to design and implement a *bit-serial* version of the adder/subtractor from Section 6.

In bit-serial processing, the addition (etc) operation is carried out *iteratively*, typically starting from the least significant bit. Only one full-adder is needed regardless of the wordlength; but the throughput is reduced compared with bit-parallel implementations, and some hardware must be added to pace the input and output signals. Figure 2 shows a conceptual illustration of a bit-serial adder. The full-adder carry-out signal is fed back to the carry-in port to be used in the next bit addition.

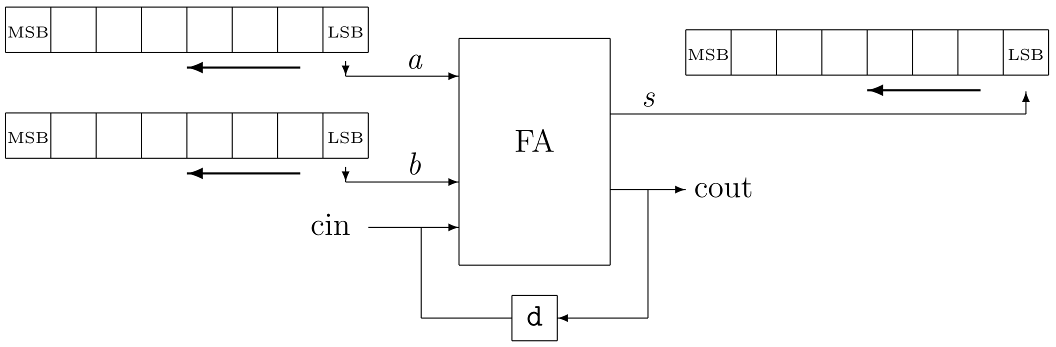

Figure 2: Conceptual illustration of simple bit-serial adder. The letter `d` marks a delay of one clock cycle. The input and output bits are selected in order, one per clock cyle, starting from the LSB. Mode-selecting control signals are omitted for clarity.

Figure 3 illustrates the entity for a serial adder/subtractor. In addition to the signals present already in Section 6, there is a clock signal and a control bit to decide when to start the calculation. An output signal that marks when computation is finished will simplify testing.

Preparation: Using Figure 2 and your Section-6 adder design as inspiration, design a bit-serial adder/subtractor according to Figure 3. The start signal should be *active-high*, that is, computation should start when the signal is set. The finished signal should also be active-high. Trigger on the rising edge of the clock. The reset signal should be synchronous (that is, it should take effect only on the next clock edge) and active-high.

Next, implement your design in VHDL as before:

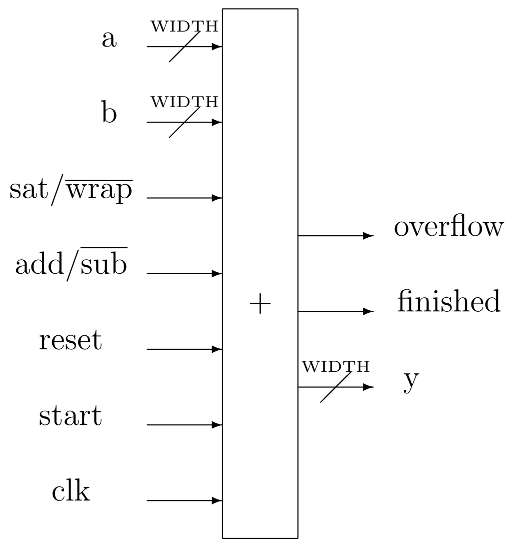

Figure 3: Illustration of entity for serial adder/subtractor with additional control signals.

- Create a project and directory named `lab1e`.
- Download the test bench from the home page.
- Create a VHDL entity in accordance with Figure 3. Compile with the test bench to verify consistency of interfaces.
- Design the architecture for the bit-serial adder/subtractor. Again, consider the desired hardware, and be aware that you cannot instantiate the FA as a component within the clocked process itself. Take care not to copy the result to the output until the entire calculation is completed. Compile to verify consistency with entity.
- Download the *do* file to test your design, and verify it as before.

8 Serial multiplier

$$
\begin{array}{rccccccccr}
 & a: & & & 1 & 1 & 0 & 1 & & 13 \\
 & b: & & & 1 & 0 & 0 & 1 & & 9 \\
\hline
 & & & & 1 & 1 & 0 & 1 & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
1 & 1 & 0 & 1 & & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 1 & & & 117
\end{array}
$$

Figure 4: Binary (not two's-complement) multiplication on paper. A one in a bit position in factor b means that the corresponding line in the summation is a (shifted) copy of a; a zero means that the corresponding line is cleared.

The bit-serial adder of the previous section produces one bit of the result in each clock cycle. With a similar principle, it is possible to build a serial multiplier implementation, where in subsequent cycles, one factor (the *partial product*) is added (or not) to the running sum, depending on the value of the subsequent bits of the other factor. Figure 4 illustrates the principle; a simplified view of an implementation is shown in Figure 5.

Preparation: The block marked "×" in Figure 5 generates the partial products as in Figure 4 from one 1-bit and one `WIDTH`-bit factor. It may be implemented very simply with logic gates; how?

Preparation: Starting from the serial-adder example, find a way to use clocked elements to implement the multiplier illustrated in Figure 5.

Preparation: A conceptually simple way to handle signed numbers is to inspect each factor and negate it if negative, and then negate the product if exactly one of the factors was inverted. Add such functionality to the multiplier.

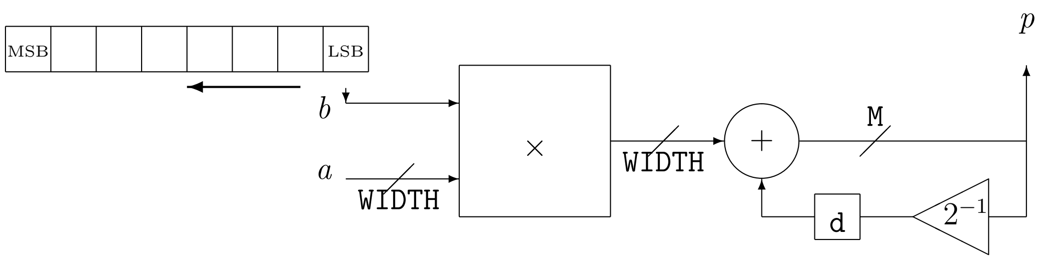

Figure 5: Conceptual illustration of serial multiplier, calculating $p = a \times b$.

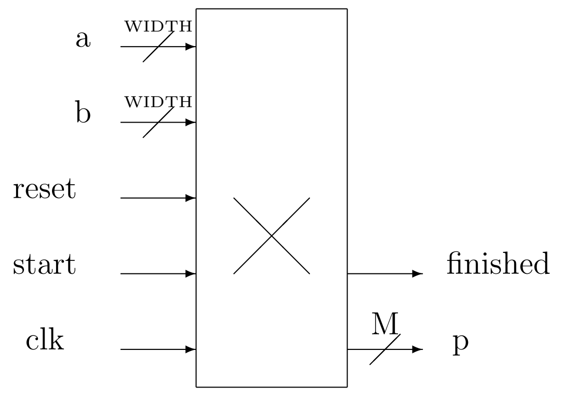

Figure 6: Interface of serial-multiplier entity.

The interface of the VHDL entity of the serial multiplier is illustrated in Figure 6. The signal ports are as for the bit-serial adder. Note that the wordlength of the product can be different from that of the factors.

Preparation: If both factors have wordlength `WIDTH`, how large must the product wordlength `M` be to avoid the potential for overflow?

The final implementation should proceed as for the previous designs:

- Create a project and directory named `lab1f`.
- Download the test bench from the homepage.
- Create a VHDL entity in accordance with Figure 6. Use the test bench to verify it, as before.
- Create a VHDL architecture in accordance with your preparations and with the entity. Compile with test bench.
- Download the *do* file that tests the multiplier with positive and negative factors on each input. Verify your design.

9 Wrap-up

After completing this lab session, you are expected to be able to carry out the following tasks:

- Use design hierarchy to reduce work and code amount for a given functionality.

- Declare and use `generic` parameters to make a design more reusable.
- Design simple combinational and sequential two's-complement arithmetic circuits.
- Discuss overflow, wrap-around, and saturation in two's-complement arithmetic.

A note about the example circuits: As arithmetic circuits are important in terms of both hardware amount and performance, there is a huge literature on how best to design them. The examples used here are not high-performance designs; they were chosen for simplicity, in order not to obscure the learning objectives listed above. Higher-performance architectures are discussed elsewhere.