Due: TBD

Acknowledgement: This homework is prepared with the help of course material provided by Prof. Arvind from MIT.

1 Introduction

This lab is used as an introduction to simple combinational circuits and Bluespec System Verilog (BSV). Even though BSV contains higher level functions to create circuits, this lab will focus on using low level gates to create blocks that are used in higher level circuits such as adders. This will enable you to see how digital hardware can be generated by the BSV compiler.

In this lab you will build and compare multi-bit adders. First, you will implement a **ripple-carry adder** like the one you saw in the lecture. Then, you will write a polymorphic multiplexer using for-loops and use it to implement a **carry select adder**.

1.1 Adders

Adders are essential building blocks for digital systems. There are many different adder architectures that provide different combinations of area, speed and power features. There is no single architecture that dominates all other adders in all the areas. Therefore hardware designers choose adders based on system area, speed, and power constraints.

1.1.1 Full Adder

Full adder is the basic component for building arithmetic blocks in digital systems. As shown in Figure 1a, it has three 1-bit inputs (a, b, cin) and two 1-bit outputs (s, cout). The logic in the full adder is defined as:

$s = (a \oplus \veebar b) \oplus \veebar c_{in}$

$c_{out} = a.b + (a \oplus \veebar b).c_{in}$

When you add more than one-bit numbers, then the carry output from one bit addition is carry input to the next higher order bit.

1.1.2 Ripple-Carry Adder

As in the long-hand manual addition of two n-bit numbers, the carry ripples from least significant to the most significant bit of the result. A ripple-carry adder simply simulates this manual process and is amongst the simplest adder architectures.

A ripple-carry adder is made up of a chain of full adder blocks connected through the carry chain. A 4-bit ripple carry adder can be seen in Figure 1b.

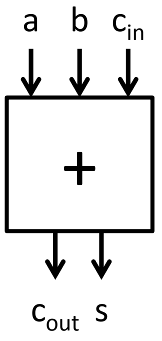
While it is a straightforward implementation of the long-hand manual addition, its performance is affected by doing all addition operations in series even when all input bits are available as the carry ripples through a series of 1-bit full adder circuits.

1.1.3 Carry-Select Adder

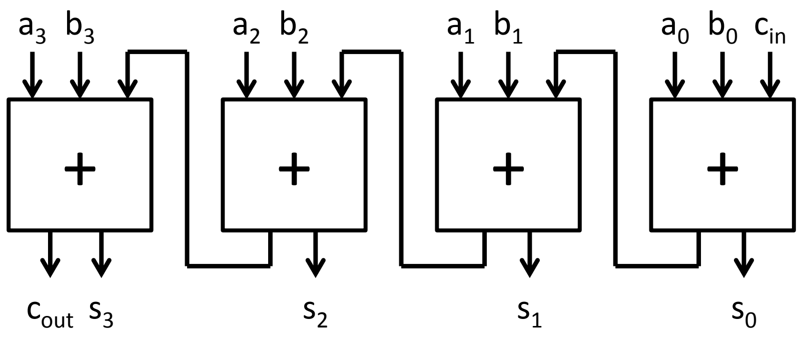
Keep in mind that both “a” and “b” inputs to the adder are available nearly immediately when the adder operation begins. A carry-select adder (CSA) makes use of this fact to compute sums simultaneously by assuming carry to be both 0 and 1.

Thus, the carry select adder adds prediction or speculation to the ripple carry adder to speed up execution. It computes the bottom bits the same way the ripple carry adder computes them, but it differs in the way it computes the top bits. Instead of waiting for the carry signal from the lower bits to be computed, it computes two possible results for the top bits: one results assumes there is no carry from the lower bits and the other assumes there is a bit carried over. Once that carry bit is calculated, a mux is used to select the top bits that correspond to the carry bit. An 8-bit carry select adder can be seen in Figure 2.

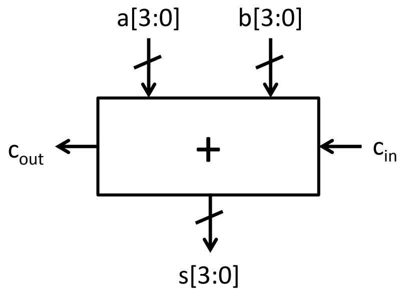
Since a CSA computes additions by assuming and selecting between the two values of input carry, the final output is provided by a multiplexer circuit discussed next.



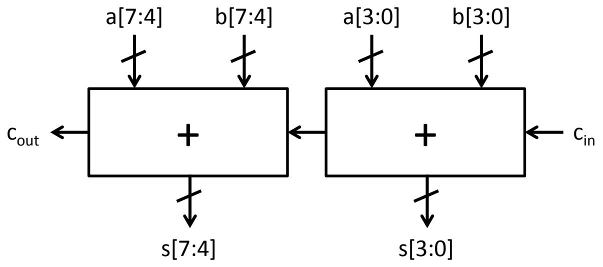
(a) Full adder



(b) 4-bit ripple carry adder built from full adders



(c) Symbol for 4-bit adder



(d) 8-bit ripple carry adder

Figure 1: Construction of a 4-bit adder and an 8-bit adder from full adder blocks

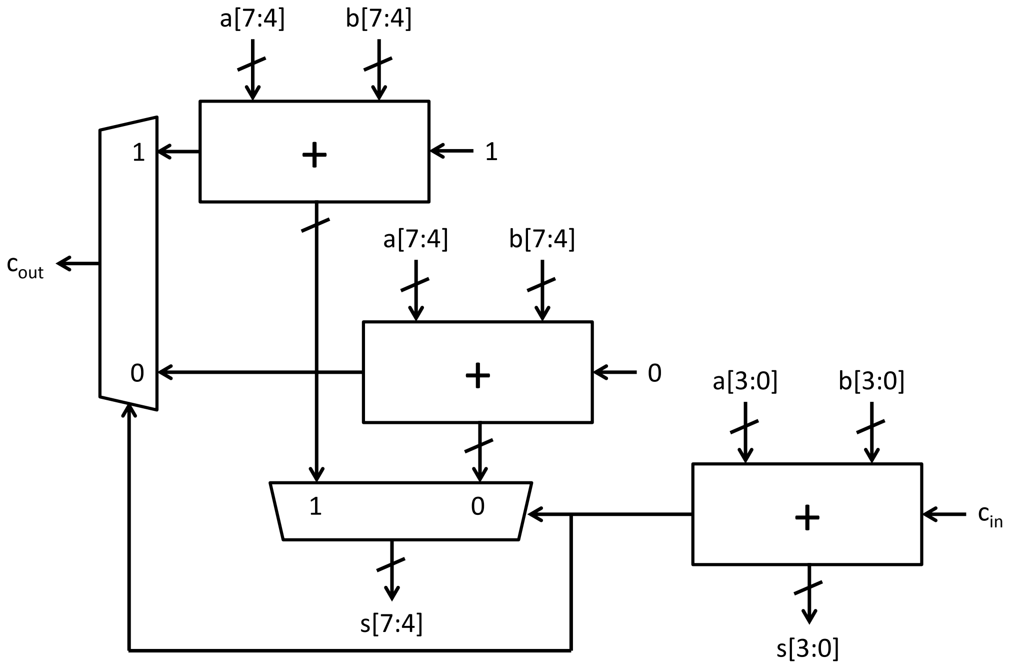


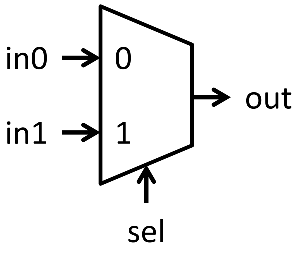
Figure 2: 8-bit carry select adder

1.2 Multiplexers

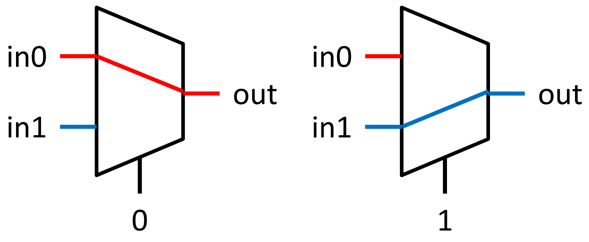
Multiplexers (or muxes for short) are blocks that are used to select between multiple signals. A multiplexer has multiple data inputs `in`*N*, a select input `sel`, and a single output `out`. The value of `sel` determines which input is shown on the output.

The muxes in this lab are all 2-way muxes. That means there will be two inputs to select between (`in0` and `in1`) and `sel` will be a single bit. If the `sel` is 0, then `out` = `in0`. If the `sel` is 1, then `out` = `in1`.

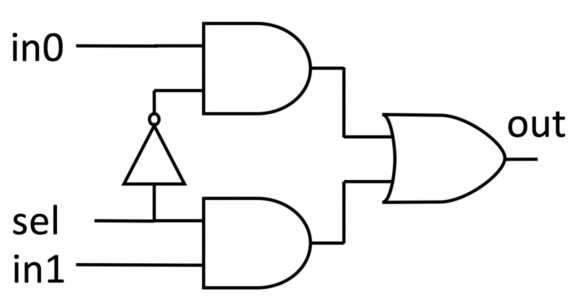
Figure 3a shows the symbol used for a mux, and figure 3b shows pictorially the function of a mux.



(a) Multiplexer symbol



(b) Multiplexer functionality



(c) Construction of 1 bit multiplexer

Figure 3: Symbol, functionality, and construction of 1 bit multiplexer

2 Lab Assignment

2.1 Guidelines

1. You are not allowed to use Bluespec built-in and (&), or (|), not (∼), xor (∧) and add (+) operators. Use the following functions for and1, or1, xor1 and not1. You can use + operator for incrementing loop variable.

```
function Bit#(1) and1(Bit#(1) a, Bit#(1) b);
    return a & b;
endfunction

function Bit#(1) or1(Bit#(1) a, Bit#(1) b);
    return a | b;
endfunction

function Bit#(1) xor1( Bit#(1) a, Bit#(1) b );
    return a ^ b;
endfunction

function Bit#(1) not1(Bit#(1) a);
    return ~ a;
endfunction
```

2. Use following functions for 1 bit full adder sum bit and carry bit

```
function Bit#(1) fa_sum( Bit#(1) a, Bit#(1) b, Bit#(1) c_in );
    return xor1( xor1( a, b ), c_in );
endfunction

function Bit#(1) fa_carry( Bit#(1) a, Bit#(1) b, Bit#(1) c_in );
    return or1( and1( a, b ), and1( xor1( a, b ), c_in ) );
endfunction
```

2.2 Building a 4-bit Ripple Carry Adder

Exercise 1

Complete the code in `Adders.bsv` for `add4` by using a for loop to properly connect all the instances of `fa_sum` and `fa_carry`. Use fa_sum and fa_carry functions for writing `add4` code.

```
function Bit#(5) add4( Bit#(4) a, Bit#(4) b, Bit#(1) c_in );
    // ...
endfunction
```

Check the correctness of the code by running the Add4 testbench:

```
$ make add4simple
$ ./simAdd4Simple
```

2.3 Building an 8-bit Ripple Adder

Exercise 2

Complete the code in `Adders.bsv` for `add8` by using `add4` function that you wrote in previous exercise.

```
function Bit#(9) add8( Bit#(8) a, Bit#(8) b, Bit#(1) c_in );
    // ...
endfunction
```

Check the correctness of the code by running the RCA testbench:

```
$ make rcasimple
$ ./simRcaSimple
```

2.4 Building an 8-bit Carry Select Adder

2.4.1 Building an n-bit Multiplexer

The first step in constructing our carry select adder is to build a basic multiplexer from gates. Let us first examine `Multiplexer.bsv`.

```
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    return (sel == 0)? a: b;
endfunction
```

The first line begins a definition of a new function called multiplexer1. This multiplexer function takes several arguments which will be used in defining the behavior of the multiplexer. This multiplexer operates on single bit values, the concrete type `Bit#(1)`. Later we will learn how to implement polymorphic functions, which can handle arguments of any width.

This function uses C-like constructs in its definition. Simple code, such as the multiplexer can be defined at the high level without implementation penalty. However, because hardware compilation is a difficult, multi-dimensional problem, tools are limited in the kinds of optimizations that they can do.

The `return` statement, which constitutes the entire function, takes two input and selects between them using `sel`. The `endfunction` keyword completes the definition of our multiplexer function. You should be able to compile the module.

Exercise 3

Referring to Figure 3c, using the and, or, and not gates, re-implement the function `multiplexer1` in `Multiplexer.bsv`. (The required functions, called `and1`, `or1` and `not1`, respectively, are provided)

```
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    ....
endfunction
```

Check the correctness of the code by running the multiplexer1 testbench:

```
$ make mux1simple
$ ./simMux1Simple
```

Exercise 4

Complete the implementation of the function `multiplexer4` in `Multiplexer.bsv` using for loops and `multiplexer1`.

```
function Bit#(4) multiplexer4(Bit#(1) sel, Bit#(4) a, Bit#(4) b);
    ....
endfunction
```

Check the correctness of the code by running the multiplexer testbench:

```
$ make muxsimple
$ ./simMuxSimple
```

Exercise 5

Complete the definition of the function `multiplexer\_n` in `Multiplexer.bsv`. Verify that this function is correct by replacing the original definition of multiplexer4 to only have: `return multiplexer\_n(sel, a, b);`. This redefinition allows the test benches to test your new implementation without modification.

Check the correctness of the code by running the multiplexer_n testbench:

```
$ make muxnsimple
$ ./simMuxNSimple
```

2.4.2 Building 8-bit Carry Select Adder using 4-bit Adders

We will now move on to building adders. The fundamental cell for adding is the full adder which is shown in Figure 1a. This cell adds two input bits and a carry in bit, and it produces a sum bit and a carry out bit. `Adders.bsv` contains two function definitions to describe the behavior of the full adder. `fa_add` computes the add output of a full adder, and `fa_carry` computes the carry output. These functions contain the same logic as the full adder presented in lecture 2.

An adder that operates on 4-bit numbers can be made by chaining together 4 full adders as shown in Figure 1b. This adder architecture is known as a ripple carry adder because of the structure of the carry chain. To generate this adder without writing out each of the explicit full adders, a for loop can be used similar to `multiplexer5`.

Exercise 6

Referring to Figure 2, in `Adders.bsv` complete the code for `cs_add8`, you must use `add4` and `multiplexer_n` for this exercise, you are not allowed to use `multiplexer1` for this exercise.

```
function Bit#(9) cs_add8( Bit#(8) a, Bit#(8) b, Bit#(1) c_in );
    // ...
endfunction
```

Check the correctness of the code by running the CSA testbench:

```
$ make csasimple
$ ./simCsaSimple
```

3 Bluespec Reference

3.1 Static Elaboration

Many muxes in real world systems are larger than 1-bit wide. We will need multiplexers that are larger than a single bit, but writing the code to manually instantiate 32 single-bit multiplexers to form a 32-bit multiplexer would be tedious. Fortunately, BSV provides constructs for powerful static elaboration which we can use to make writing the code easier. Static elaboration refers to the process by which the BSV compiler evaluates expressions at compile time, using the results to generate the hardware. Static elaboration can be used to express extremely flexible designs in only a few lines of code.

In BSV we can use bracket notation (`[]`) to index individual bits in a wider Bit type, for example `bitVector[1]` selects the second least significant bit in `bitVector` (`bitVector[0]` selects the least significant bit since BSV's indexing starts at 0). We can use a for-loop to copy many lines of code which have the same form. For example, to aggregate the and1 function to form a 5-bit and function, we could write:

```
function Bit#(5) and5(Bit#(5) a, Bit#(5) b);
    Bit#(5) aggregate;
    for(Integer i = 0; i < 5; i = i + 1) begin
        aggregate[i] = and1(a[i], b[i]);
    end
    return aggregate;
endfunction
```

The BSV compiler, during its static elaboration phase, will replace this for loop with its fully unrolled version.

```
aggregate[0] = and1(a[0], b[0]);
aggregate[1] = and1(a[1], b[1]);
aggregate[2] = and1(a[2], b[2]);
aggregate[3] = and1(a[3], b[3]);
aggregate[4] = and1(a[4], b[4]);
```

3.2 Polymorphism and Higher-order Constructors

So far, we have implemented two versions of the multiplexer function, but it is easy to imagine needing an n-bit multiplexer. It would be nice if we did not have to completely re-implement the multiplexer whenever we want to use a different width. Using the for-loops introduced in the previous section, our multiplexer code is already somewhat parametric because we use a constant size and the same type throughout. We can do better by giving a name (`N`) to the size of the multiplexer using `typedef`. Our new multiplexer code looks something like:

```
typedef 5 N;
function Bit#(N) multiplexerN(Bit#(1) sel, Bit#(N) a, Bit#(N) b);
    // ...
    // code from multiplexer5 with 5 replaced with N (or valueOf(N))
    // ...
endfunction
```

The `typedef` gives us the ability to change the size of our multiplexer at will. The `valueOf` function introduces a small subtlety in our code: `N` is not an Integer but a *numeric type* and must be converted to an Integer before being used in an expression. Even though it is improved, our implementation is still missing some flexibility. All instantiations of the multiplexer must have the same type, and we still have to produce new code each time we want a new multiplexer. However in BSV we can further parameterize the module to allow different instantiations to have instantiation-specific parameters. This sort of module is polymorphic, the implementation of the hardware changes automatically based on compile time configuration. Polymorphism is the essence of design-space exploration in BSV.

The truly polymorphic multiplexer can be started as follows:

```
// typedef 32 N; // Not needed
function Bit#(n) multiplexer\_n(Bit#(1) sel, Bit#(n) a, Bit#(n) b);
```

The variable `n` represents the width of the multiplexer, replacing the concrete value `N` (=32). In BSV *type variables* (`n`) start with a lower case whereas concrete types (`N`) start with an upper case.