Lab 2: Multiplexers and Adders

Due: 11:59:59pm, Sunday April 30th, 2017

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 consumption. There is no single architecture that dominates all others in all aspects. Therefore hardware designers choose adders based on system area, speed, and power constraints.

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, c_{in}) and two 1-bit outputs (s, c_{out}). The logic in the full adder is defined as:

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

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

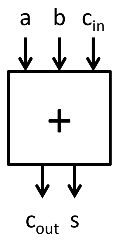
where ., +, and $\oplus$ represent logic AND, OR, and XOR operations, respectively. When you add multiple-bit numbers, then the carry output from each one-bit addition is carry input to the next higher-order one.

Ripple-Carry Adder

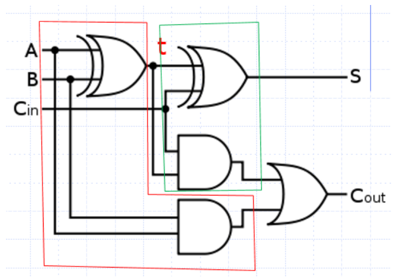
As in the long-hand manual addition of two n-bit numbers, the carry ripples from least significant to the most significant bit of the result. A ripple-carry adder simply simulates this manual process and is amongst the simplest adder architectures.

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 1d.

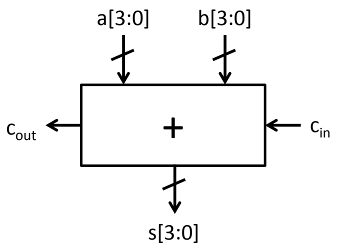
While it is a straightforward implementation of the long-hand manual addition, its performance is affected by doing all addition operations in series even when all input bits are available as the carry ripples through a series of 1-bit full adder circuits.



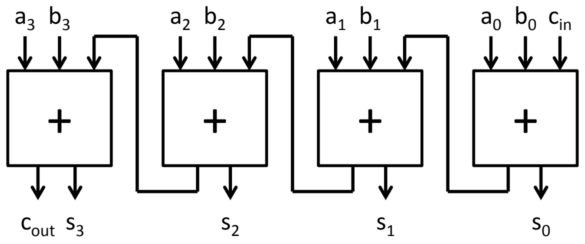
(a) Full adder



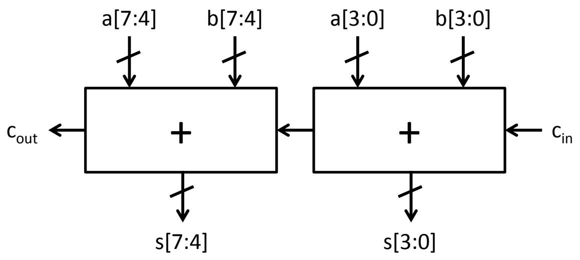
(b) Gate-level implementation of full adder



(c) Symbol for 4-bit adder



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



(e) 8-bit ripple carry adder

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

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 the 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 result 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.

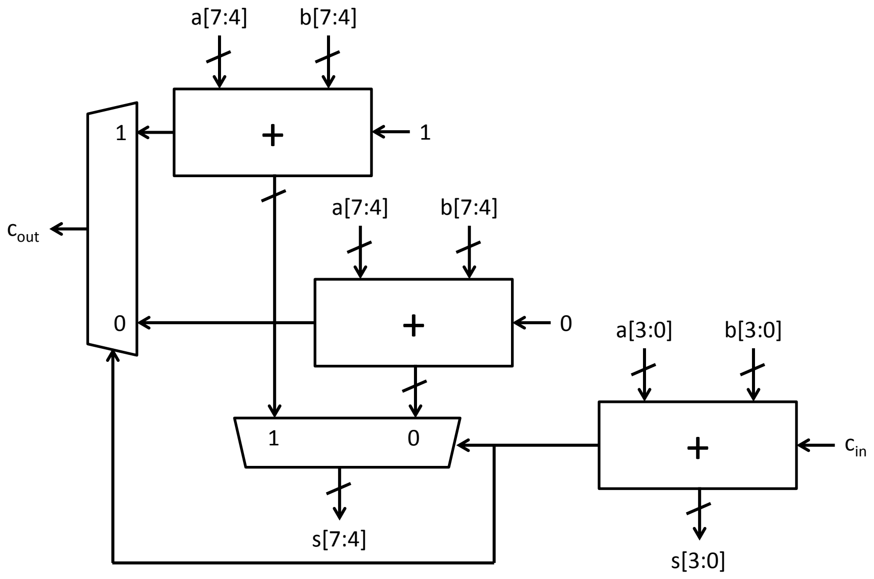
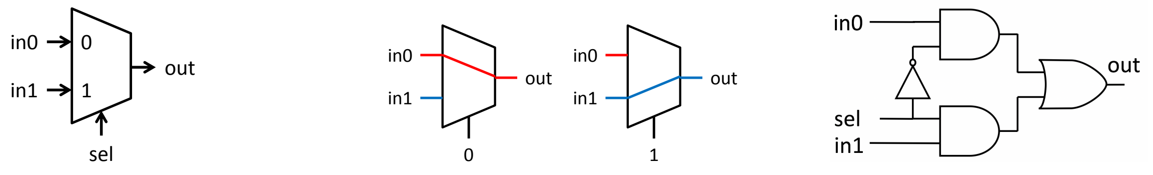


Figure 2: 8-bit carry select adder

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.

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 transferred the output. The muxes in this lab are all 2-way muxes. That means there will be two inputs to select from (`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

3 Lab Assignment

First, make a copy of the source code using the following command

```
$ cp -r  /home/linux/ieng6/cs140g/public/lab2 .
```

3.1 Guidelines

Please follow these instructions strictly. You will get zero marks if the given instructions are not followed for an exercise.

1. You are NOT allowed to use `if` statements, including the ternary operator (`?:`).
2. You are NOT allowed to use BSV built-in AND (&), OR (|), NOT (∼), and XOR (∧) operators. Instead, use the functions `and1`, `or1`, `xor1` and `not1` defined in `BasicGates.bsv`.
3. You are NOT allowed to use BSV built-in ADD (+) operator, except for incrementing loop variables.

3.2 Building a 4-bit Ripple Carry Adder

Exercise 1: 10 Points

Referring to Figure 1b, complete the code in `Adders.bsv` for `fa` function using only the functions in `BasicGates.bsv` (`and1`, `or1`, `xor1` and `not1`). Check the correctness of the code by running the `fa` testbench:

```
$ make fa
$ ./simFA
```

Exercise 2: 5 Points

Complete the code in `Adders.bsv` for `add4` by using the `fa` function for four single-bit full adders you implemented in the previous exercise. Check the correctness of the code by running the `add4` testbench:

```
$ make add4
$ ./simAdd4
```

3.3 Building an 8-bit Ripple Adder

Exercise 3: 5 points

Complete the code in `Adders.bsv` for `add8` using the `add4` function that you wrote in the previous exercise. Check the correctness of the code by running the `rca` testbench:

```
$ make rca
$ ./simRCA
```

3.4 Building an 8-bit Carry Select Adder

3.4.1 Building an 1-bit Multiplexer

The first step in constructing our carry select adder is to build a basic multiplexer from gates. Multiplexer is defined as follows

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

Exercise 4: 10 Points

Referring to Figure 3c, write a gate-level implementation of the function `multiplexer1` in `Multiplexer.bsv` using functions `and1`, `or1` and `not1` only. Check the correctness of the code by running the `mux1` testbench:

```
$ make mux1
$ ./simMux1
```

3.4.2 Static Elaboration

We will need multiplexers that are larger than a single bit, but writing the code to manually instantiate multiple single-bit multiplexers to form a multiple-bit multiplexer would be tedious. BSV provides constructs for powerful static elaboration which we can used 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 instead of many lines of code which have the same form. For example, to aggregate four `and1` functions to form a 4-bit `and` function, we could write:

```
function Bit#(4) and4(Bit#(4) a, Bit#(4) b);
    Bit#(4) aggregate=0;
    for(Integer i = 0; i < 4; 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]);
```

Exercise 5: 10 Point

Complete the implementation of the function `multiplexer4` in `Multiplexer.bsv` using `for` loop(s) and `multiplexer1`. Check the correctness of the code by running the `mux4` testbench:

```
$ make mux4
$ ./simMux4
```

3.4.3 Polymorphism

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 4 N;
function Bit#(N) multiplexerN(Bit#(1) sel, Bit#(N) a, Bit#(N) b);
    // code from multiplexer4 with 4 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 parametrize 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 function have the following signature:

```
// typedef 4 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` (= 4). In BSV *type variables* (`n`) start with a lower case whereas concrete types (`N`) start with an upper case.

Exercise 6: 5 Points

Complete the definition of the function `multiplexer_n` in `Multiplexer.bsv`. Check the correctness of the code by running the `muxn` testbench:

```
$ make muxn
$ ./simMuxN
```

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

Exercise 7: 35 points

Referring to Figure 2, in `Adders.bsv` complete the code for `cs_add8` using `add4` and `multiplexer_n` functions. You are NOT allowed to use `multiplexer1` for this exercise. Check the correctness of the code by running the `csa` testbench:

```
$ make csa
$ ./simCSA
```

Exercise 8: 20 points

Answer the following questions in `Lab2.txt` in the required format. You will get zero marks if you use incorrect formatting. Use the following area and delay values for your calculations (see Appendix for the details). Calculate the area and delay of the Carry Select Adder and Ripple Carry Adder using the area and delay values of the optimized Full Adder with constant carry_in when possible. Assume that wires takes zero area and zero delay.

		`c_in` not constant	`c_in=1`	`c_in=0`
Area of Full Adder		7	5	3
Maximum propagation delay of Full Adder	from `c_in` to `c_out`	2	N/A	N/A
	from `a` to `c_out`	4	3	1
	from `b` to `c_out`	4	3	1
	from `c_in` to `s`	2	N/A	N/A
	from `a` to `s`	4	3	2
	from `b` to `s`	4	3	2

Table 1: Area and delay for Full Adder, N/A = Not Applicable.

Area of `multiplexer1`		4
Maximum propagation delay of `multiplexer1`	from `in0` to `out`	2
	from `in1` to `out`	2
	from `sel` to `out`	3

Table 2: Area and delay for `multiplexer1`

Exercise 8.1: 2 points

Compute the total area of 8-bit Ripple Carry Adder.

Exercise 8.2: 4 points

Compute the maximum propagation delay of 8-bit Ripple Carry Adder from any input to C_out.

Exercise 8.3: 6 points

Compute the total area of 8-bit Carry Select Adder.

Exercise 8.4: 6 points

Compute the maximum propagation delay of 8-bit Carry Select Adder from any input to C_out.

Exercise 8.5: 1 points

Which adder has least area? Write 0 for RCA and 1 for CSA.

Exercise 8.6: 1 points

Which adder has smallest delay? Write 0 for RCA and 1 for CSA.

4 Submission

Write your solution for exercises 1,2,3, and 7 in `Adder.bsv`, exercise 4,5, and 6 in `Multiplexer.bsv` and exercise 8 in `Lab2.txt`. Submission instructions are the same as Lab 1 (Piazza @ 68). To submit your assignment, simply run the following command from `lab2/` directory:

```
$ bundleP2 <your-email>
```

Submission verification

Refer to Lab 1.

5 Appendix: area and delay calculation

Table 3 shows the area and delay of different gates in their respective units. Area and delay of the full adder (Table 1) and `multiplexer1` (Table 2) are calculated using Table 3 values.

gate	area	delay
`not`	1	1
`and`	1	1
`or`	1	1
`xor`	2	2

Table 3: Area and delay of `not`, `and`, `or` and `xor` gates

5.1 Area Calculation

Area of full adder shown in Figure 1b is sum of area of all gates. Using the values from Table 3, area of full adder is $2 + 2 + 1 + 1 + 1 = 7$. Now if `c_in` is constant then we can reduce some gates using the following rules.

```
and(a,1)=and(1,a)=a
and(a,0)=and(0,a)=0
or(a,1)=or(1,a)=1
or(a,0)=or(0,a)=a
xor(a,0)=xor(0,a)=a
xor(a,1)=xor(1,a)=not(a)
```

For example if `c_in` is 1, then `and(t,c_in)` can be reduce to just a wire `t` and `xor(t,1)` can be reduced to `not(t)` gate. Hence the area of full adder with `c_in=1` is $2 + 1 + 1 + 1 = 5$.

5.2 Delay Calculation

To calculate the maximum propagation delay from an input to an output, we pick the longest path along which the delay is maximum. Delay of a path is the sum of the delays of all gates along that path. For example, maximum propagation delay from `c_in` to `c_out` of a full adder is $1+1 = 2$. Similarly, the delay from `a` to `c_out` is $2+1+1 = 4$.

As we saw in previous section, if some of the inputs are constant, then some gates can be reduced to simpler gates or wires. In the case of full adder with `c_in=1` discussed above, maximum propagation delay from `a` to `c_out` is $2 + 1 = 3$.