Baylor University

Department of

Electrical and Computer Engineering

ELC 5311 (graduates), 4311(undergrad)

Advanced Digital Logic Laboratories

Keith Schubert
Professor
Department of Electrical and Computer Engineering
Baylor University

Contents

Lab 1

Barrel Shifter

Verilog has a built in left and right shift for both arithmetic and logical:

Operation	Verilog	Exmple
left shift logical	`<<`	`assign a = b <<2;`
left shift arithmetic	`<<<`	`assign a = b <<<2;`
right shift logical	`>>`	`assign a = b >>2;`
right shift arithmetic	`>>>`	`assign a = b >>>2;`

So why do we need more? There is no circulant version, and circulant is quite useful in many operations, such as multiplication, division, and various encryption techniques.

Remember that when you shift something there is always an empty spot left. That empty spot has to be filled with something, so the different shift styles explain how to fill the missing bits. Logical always fills with zero, regardless of the direction. Arithmetic fills with the sign bit, if it is a left shift, or zeros if it is a right shift. In a circulant shift, the bits that shift out, wrap around and become the bits shifted in. You can make any shift from a logical, though it takes some extra work and thus time also. For instance a left circulant shift by n of an N bit number can be made by doing a left logical shift by n, a right logical shift by $N - n$, then taking a bitwise or of the two results.

This is not as fast, or efficient as a dedicated circuit, particularly since doing a left and right shift then taking the or, would use three commands, thus three cycles of computation. This is very expensive. We will proceed by building a right shift, then making a left or right shift from it. We will then look at alternate ways to build a right shift that work better when parameterizing.

1.1 Shifting Right

Done in class. Make sure you make the extendable version using the generate statement.

1.2 Shifting Left

This is the assignment to undergrads and grads. Take what we did in class and make a version to left shift. There are two common ways:

1. Invert the input on the number to be shifted and feed into a right shifter then invert the output. As mentioned in class this will work and does not take extra time (the inversion is a wire reorder)

2. Recode the right shift as a left shift.

You must do the second.

1.3 Shift Left or Right

If you are a grad student, you must now make a circuit that can shift left or right. There are several ways to do this. One of the easiest is to make both and add a mux to select which to use. You can also note that the first way of building the left shift above can be used to reduce the shifter units by adding a conditional flip before and after the Right Shift unit. The first is likely the best choice in this case as you have to build the left shift as mentioned for the undergrads, then do this combination based off it.

1.4 Assignment

1. You must build the units above (upload .sv files)
   (a) Undergrads and Grads: left shift
   (b) Grads: Combine left shift with right shift from class.
2. Write a testbench as described in class and test your units from above. (upload sv files)
3. Program your board and take a picture of it working.
4. Create a report that describes the project, design rational, testing rational, and results (screenshots and pictures of board running the code). (upload pdf)
5. Make sure to include pictures of your timing diagram (from test) and the picture of the functioning board.

Lab 2

Adders and Complexity

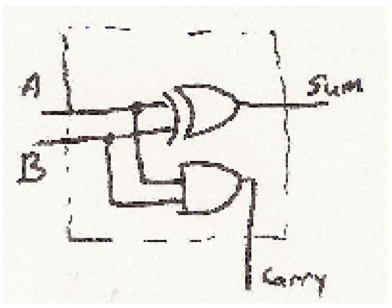
2.1 Introduction

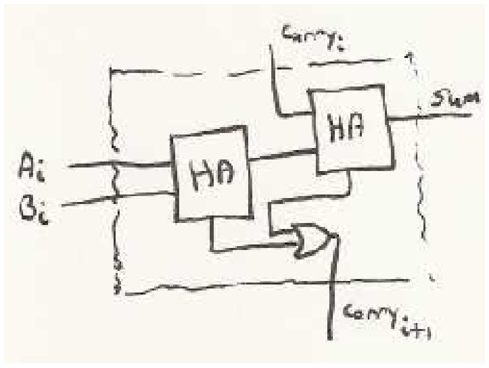
Addition is a fundamental part of any ALU and can be easily produced in Verilog by just using "+". You will get an adder that is inferred for your FPGA. The actual adder varies wildly from simple ripple adders, to specialized pre-built hardware blocks. We are going to explicitly build three different adders with very different complexities and compare them.

2.1.1 Ripple Adders

This is the technique that is covered in digital logic. Basically, full bit adders, see Figure 2.1, are created and cascaded together. The carry bit from the previous full adder must arrive before the result is added. The resulting valid carries thus ripple down to the most significant bit (hence the name). Adding n bit numbers, thus takes the propagation time of $n + 1$ levels of logic, i.e. it is O(n) in time to calculate addition. Thus if 32 bit numbers are added on fast logic (1ns per stage/gate) the process would take 33ns. This is way too slow. On the bright side, none of the gates take more than 2 inputs so the size of the gates is O(1).

Figure 2.1: (left) Half Adder, (right) Full Adder





2.2 Generate

The Verilog generate command is a powerful tool. Consider the code to generate a ripple carry adder below.

Listing 2.1: Ripple adder

```
`timescale 1ns / 1ps

module ripple#(parameter N=8)(
    input logic [N-1:0] x,
    input logic [N-1:0] y,
    input logic mode,
    output logic [N-1:0] sum,
    output logic cout
    );
    logic [N:0] carry;
    logic [N-1:0] ty;

    assign carry[0]=mode;
    assign cout=carry[N];

    generate
    genvar i;
    for (i=0;i<N;i++) begin:fa
        assign ty[i]= y[i]^mode;
        assign carry[i+1]= x[i]&ty[i] | x[i]&carry[i] | ty[i]&carry[i];
        assign sum[i]= x[i]^ty[i]^carry[i];
    end
    endgenerate

endmodule
```

2.3 Board

You need to implement your ripple carry adder/subtractor on the Nexus 4 DDR board. This will require using the switches for two numbers, so you will have to create memory with buttons to read into them. You will also need buttons to select addition or subtraction. Consider the code below.

Listing 2.2: Ripple adderboard

```
`timescale 1ns / 1ps

module RCadd_board(
    input logic CLK100MHZ,
    input logic [15:0] SW,
    input logic BTNL,
    input logic BTNR,
    input logic BTNU,
    input logic BTND,
    output logic [15:0] LED
    );
    logic [15:0] A, nA, B, nB;
    logic addsub, naddsub;
    logic cout;

    always_ff@(negedge CLK100MHZ) begin
        A<=nA;
        B<=nB;
        addsub<=naddsub;
    end

    always_comb begin
        nA=A;
        nB=B;
        naddsub=addsub;
        if (BTNL)
            nA=SW;
        if (BTNR)
            nB=SW;
        if (BTNU)
            naddsub=0;
        if (BTND)
            naddsub=1;
    end

    ripple#(.N(16)) addsubunit (
        .x(A),
        .y(B),
        .mode(addsub),
        .sum(LED),
        .cout(cout)
    );

endmodule
```

2.4 Testing

Create a testbench for the ripple carry adder that has the following features:

1. unit under test properly set up with inputs and outputs that are logic variables
2. a clock with a 10 ns cycle time run in an **always** block
3. an **initial** block that sets up the variables
4. an **initial** block that runs the test cases

2.5 Assignment

1. You must build the units above (upload .sv files)
   (a) Undergrads and Grads: for a 16-bit ripple carry adder

   (b) Grads: do one of the extra options we discussed in class.

2. Write a testbench using tasks and test your units from above. (upload sv files)
3. Program your board and take a picture of it working.
4. Create a report that describes the project, design rational, testing rational, and results (screenshots and pictures of board running the code). (upload pdf)
5. Make sure to include pictures of your timing diagram (from test) and the picture of the functioning board.

Lab 3

Carry Look-Ahead Adders

3.0.1 Carry Look-Ahead

Carry look-ahead (CLA) is also referred to as carry lookahead and lookahead carry. Don't worry about the naming differences. Assume $x + y = z$. Pre-generate all carries with 2-level logic. Usually form (g,p,c) generate, propagate, carry.

$$
\begin{aligned}
G_i &= x_i \cdot y_i && (3.1)\\
P_i &= x_i + y_i && (3.2)\\
C_i &= G_i + P_i \cdot C_{i-1} && (3.3)\\
&= G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdot C_{i-2}) && (3.4)\\
&= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-2} && (3.5)\\
&= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot G_{i-2} + \ldots + P_i \cdot P_{i-1} \cdot \ldots \cdot P_0 \cdot C_{in} && (3.6)\\
&= G_i + \sum_{j=1}^{i} \left(\prod_{k=j}^{i} P_k \right) G_{i-1} + \left(\prod_{j=0}^{i} P_j \right) C_{in} && (3.7)
\end{aligned}
$$

This method is very fast (regardless of size it take 5 levels of logic) but requires large gates for problems of reasonable size (even 16 or 32 bit numbers) and thus has problems with fan-in, fan-out, and size.

Often blocks of a number are handled with lookahead, and the blocks are connected in some fashion (for example ripple) to get the net result (i.e. just like single bit adds from a full adder are connected to propagate the carry bit, blocks or 4, 8, or more could be handled lookahead then connected to propagate the carry bit between them to handle a larger number, say 32 bits). Even better than cascading (ripple connection) the adders, is to us group carry-lookahead, in which each of the carry-lookahead adders output their group propagate and group generate variables to a circuit that generates the carry-in bits for each group. It takes 5 logic levels to generate the carries to each individual carry-lookahead adder, and each adder then takes 5 levels of logic to get the result, for a total of 10 levels of logic. For the example of adding 32 bit numbers with fast logic, it would take 10ns with group carry-lookahead adders (probably four or eight bits in a group).

Example 1: Specify the equations of a two bit binary adder with carry in (i.e.: one equation for each of the sum bits and one equation for the carry out). Put the equations in sum of products form.

Sol: Let the two numbers to be added be A_1A_0 and B_1B_0. Let the resulting sum be S_1S_0. Let the carries be C_{in} and C_{out}. Finally, let C_0 be the carry from the first bit added (saves writing).

$$S_0 = A_0 \oplus B_0 \oplus C_{in}$$

$$\begin{aligned} C_0 &= A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in} \\ S_1 &= A_1 \oplus B_1 \oplus C_0 \\ C_{out} &= A_1 \cdot B_1 + A_1 \cdot C_0 + B_1 \cdot C_0 \end{aligned}$$

Putting this in sum of products form yields

$$\begin{aligned} S_0 &= A_0' \cdot B_0' \cdot C_{in} + A_0' \cdot B_0 \cdot C_{in}' + A_0 \cdot B_0' \cdot C_{in}' + A_0 \cdot B_0 \cdot C_{in} \\ S_1 &= A_1' \cdot B_1' \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) + A_1' \cdot B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in})' + \\ &\quad\; A_1 \cdot B_1' \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in})' + A_1 \cdot B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\ &= A_1' \cdot B_1' \cdot A_0 \cdot B_0 + A_1' \cdot B_1' \cdot A_0 \cdot C_{in} + A_1' \cdot B_1' \cdot B_0 \cdot C_{in} \\ &\quad\; + A_1' \cdot B_1 \cdot (A_0' \cdot B_0' + A_0' \cdot C_{in}' + B_0' \cdot C_{in}') \\ &\quad\; + A_1 \cdot B_1' \cdot (A_0' \cdot B_0' + A_0' \cdot C_{in}' + B_0' \cdot C_{in}') \\ &\quad\; + A_1 \cdot B_1 \cdot A_0 \cdot B_0 + A_1 \cdot B_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_1 \cdot B_0 \cdot C_{in} \\ &= A_1' \cdot B_1' \cdot A_0 \cdot B_0 + A_1' \cdot B_1' \cdot A_0 \cdot C_{in} + A_1' \cdot B_1' \cdot B_0 \cdot C_{in} \\ &\quad\; + A_1' \cdot B_1 \cdot A_0' \cdot B_0' + A_1' \cdot B_1 \cdot A_0' \cdot C_{in}' + A_1' \cdot B_1 \cdot B_0' \cdot C_{in}' \\ &\quad\; + A_1 \cdot B_1' \cdot A_0' \cdot B_0' + A_1 \cdot B_1' \cdot A_0' \cdot C_{in}' + A_1 \cdot B_1' \cdot B_0' \cdot C_{in}' \\ &\quad\; + A_1 \cdot B_1 \cdot A_0 \cdot B_0 + A_1 \cdot B_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_1 \cdot B_0 \cdot C_{in} \\ C_{out} &= A_1 \cdot B_1 + A_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\ &\quad\; + B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\ &= A_1 \cdot B_1 + A_1 \cdot A_0 \cdot B_0 + A_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_0 \cdot C_{in} \\ &\quad\; + B_1 \cdot A_0 \cdot B_0 + B_1 \cdot A_0 \cdot C_{in} + B_1 \cdot B_0 \cdot C_{in} \end{aligned}$$

3.1 Four Bit (Block) Carry Look-Ahead Module

We will build our 4-bit carry look-ahead adder from a 4-bit ripple carry adder/subtractor. In the ripple carry version, we would have four full adders that pass their carry bits, and the subtraction part is handled by **xor**'ing each bit of the second number with cin[1]. To turn this into a carry look ahead unit, we will break the ripple carry connection, and add a carry look ahead unit that will supply the carry bits for all the full adders but the first, as well as the carry out. See figure fig-4bitcla.

Since we want to re-use the testing system from our ripple carry adder/subtractor, pass the same interface for the carry look-ahead (CLA) adder/subtractor. Inside the cla adder/subtractor, generate the full adders to add each bit of D.a and the xor of D.b with D.cin. Do not put the xor in the full adder[2]. Create a module for the carry look-ahead unit (4 bit), and instantiate it in the cla adder/subtractor.

We will construct the cla unit using generate eq. 3.1, propagate eq. 3.2, and the simple formula for carry eq. 3.7. Note that it is easiest to use the reducing and and reducing or constructs in System Verilog (bitwise **and** or bitwise **or** operation with only one operand on the right of the operation). I advise making a temporary array (compressed usually works best with vivado) and doing the reducing **and** then storing it. You can then combine it with a reducing **or**.

[1] 1note **xor**'ing the second number with cin, performs a selective 1's complement on the number based on the value of cin. This effectively turns cin into the mode (add on 0, subtract on 1). The number that has been 1's complemented becomes 2's complemented as cin also provides a +1 since it is the carry in to the least significant bit.

[2] It is confusing for two reasons: 1) it is not what anyone will expect in a full adder, 2) you will have to pass two carrys in one for the **xor** and a separate one for the traditional cin (they are only the same on the first full adder).

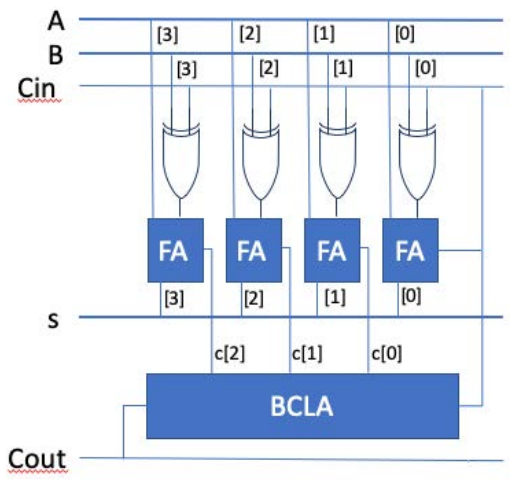


Figure 3.1: 4 bit carry look-ahead adder

3.2 Sixteen Bit Block Carry Look-Ahead

The 16 bit version of the CLA Adder/subtractor will be much like the 4 bit version. Both use one full adder for each bit of the numbers to be added or subtracted, see figure 3.2. Both use xor gates and carry in to perform the one's complement on the number to be subtracted. The full adders will have one modification though, they will now make the generate and propagate signals for each bit and pass them to the CLA unit(s) inside the CLA adder/subtractor. This is very important as in order to make the CLA tree structure without making new units for each level of the tree the previous level must combine the propagate and generate signals to make a block propagate (BP) and block generate (BG) signals for that block. When that is done the same 4 bit BCLA unit can be used for every level, and will even handle the generation of BP and BG for each level. The tree structure is pretty simple, four propagate and generate signals and one carry in come into each BCLA and four carry outs are returned along with a BP and BG to the next level down. BP and BG are created by dividing eq. 3.7 for the final carry into those terms that don't multiply cin (BG) and those that do (BP):

$$BG = G_3 + \sum_{i=1}^{3} \left(\prod_{j=i}^{3} P_j \right) G_{i-1} \tag{3.8}$$

$$BP = \prod_{i=0}^{3} P_i \tag{3.9}$$

thus we can write eq. 3.7 for the final carry as $C(3) = BG + BP \cdot C_{in}$.

3.3 Assignment

1. You must build the units above (upload .sv files)
   (a) Undergrads: for a 16-bit **ripple** block carry lookahead, with 4-bit blocks.
   (b) Grads: for a 16-bit **tree** block carry lookahead, with 4-bit blocks.

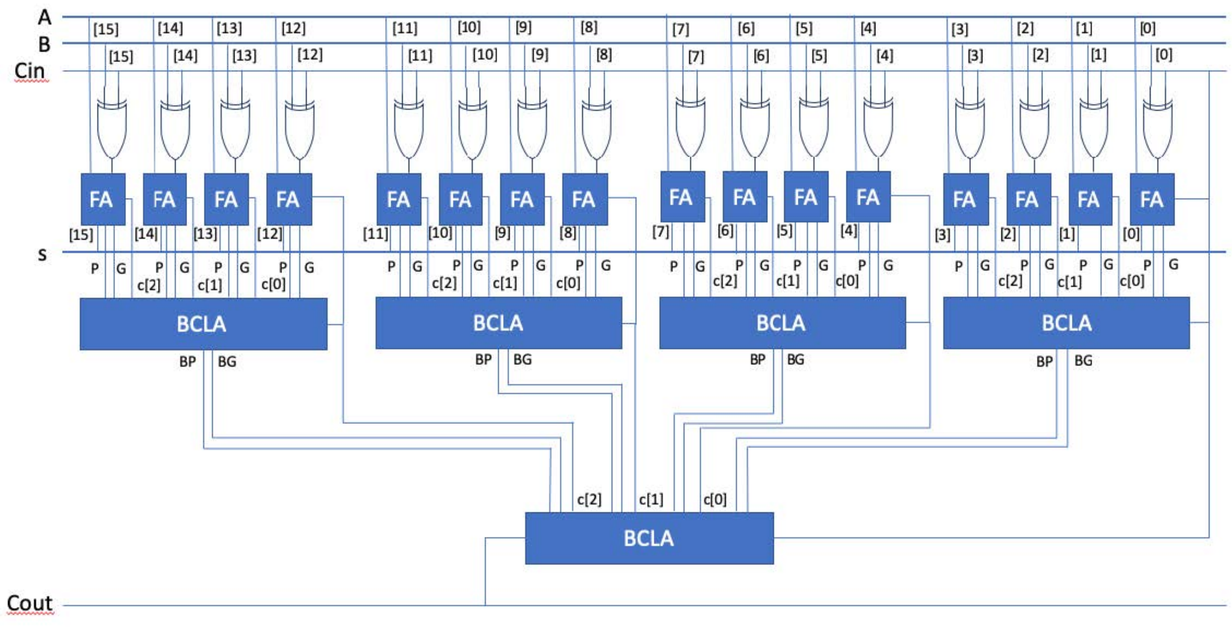


Figure 3.2: 16-bit carry look ahead adder, tree connections

2. Write a testbench using tasks and test your units from above. (upload sv files)
3. Program your board and take a picture of it working.
4. Create a report that describes the project, design rational, testing rational, and results (screenshots and pictures of board running the code). (upload pdf)
5. Make sure to include pictures of your timing diagram (from test) and the picture of the functioning board.

Lab 4

Conditional Sum Adders

4.1 Introduction

4.1.1 Conditional Sum

Conditional sum is a divide and conquer algorithm, and hence exploits binary tree parallelism. The algorithm works by calculating both possible results for each bit (if carry in was 1 or 0), then performing paired conditional concatenation using the actual carry bit of the lower number, see Figure 4.1.

1. form conditional terms for each digit in summation $\rightarrow$ (digit with carry, digit without carry) = $(x_i + y_i + 1, x_i + y_i)$
2. group by twos from right and for both conditional values in the right parenthesis form the result as follows:
   (a) the leftmost bit of the two terms on the right are the carry bits used to select the term on the left
   (b) concatenate the appropriate term on the left (picked by carry) with each term on right after removing the parity bits of the right terms
3. continue pairings until only 1 term remains. pick right number if $c_{in} = 0$ else pick left.

Example 2: Add $x = 0110$ and $y = 1111$ by conditional sum and indicate if overflow occurred.

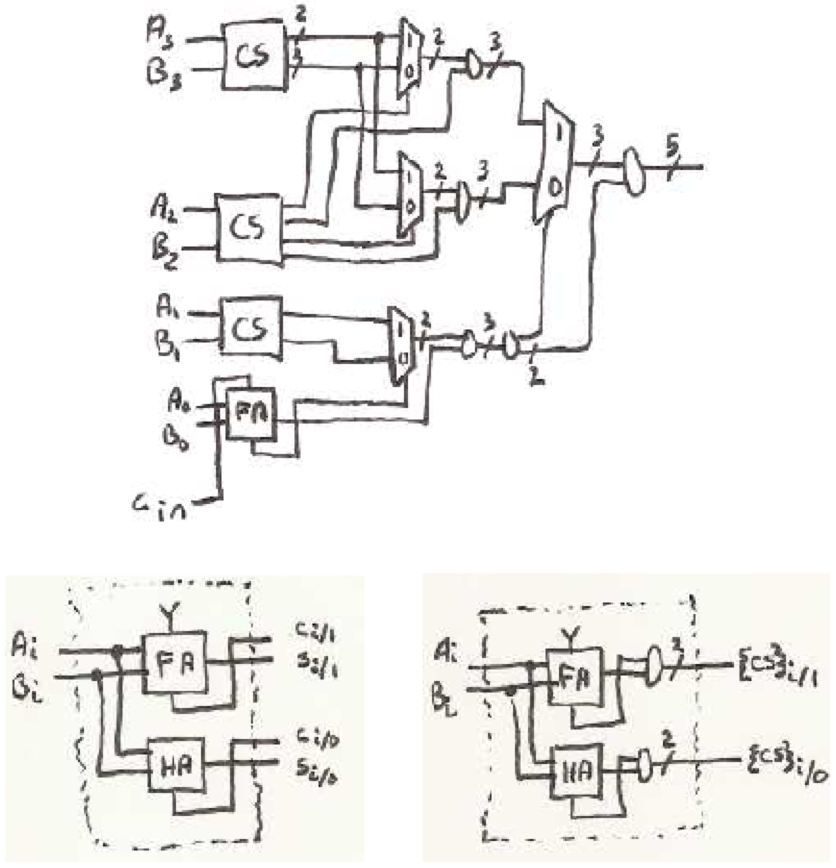
$$\begin{tikzcd}
0+1 \arrow[d] & & 1+1 \arrow[d] & & 1+1 \arrow[d] & & 0+1 \arrow[d] \\
(10,01) \arrow[dr] & & (11,10) \arrow[dl] & & (11,10) \arrow[dr] & & (10,01) \arrow[dl] \\
& (101,100 \arrow[drr] & & & & (110,101) \arrow[dll] & \\
& & & (10110,10101) & & & \\
& & & 1\ 0101 & & &
\end{tikzcd}$$

No overflow occurred (added a positive and negative number).

Example 3: Calculate $7 - 8$ by conditional sum.

$7 = 0111$ and $-8 = 1000$

Figure 4.1: Conditional Sum Adder (above), and its sub-blocks (below, left and right).



$$\begin{tikzcd}[column sep=tiny]
0 & & 1 & & 1 & & 1 \\
+1 & & 0 & & 0 & & 0 \\
(10,01) \arrow[dr] & & (10,01) \arrow[dl] & & (10,01) \arrow[dr] & & (10,01) \arrow[dl] \\
& (100,011) \arrow[drr] & & & & (100,011) \arrow[dll] & \\
& & & (10000,01111) & & &
\end{tikzcd}$$

Since this was done as addition no carry-in was set so the solution is $\underline{\;0 \mid 1111\;}$ or -1 in signed base ten.

Example 4: Add by conditional sum $x = 01100110$ and $y = 00110011$.

$$\begin{tikzcd}[column sep=tiny]
0+0 \arrow[d] & 1+0 \arrow[d] & 1+1 \arrow[d] & 0+1 \arrow[d] & 0+0 \arrow[d] & 1+0 \arrow[d] & 1+1 \arrow[d] & 0+1 \arrow[d] \\
(01,00) \arrow[dr] & (10,01) \arrow[d] & (11,10) \arrow[dr] & (10,01) \arrow[d] & (01,00) \arrow[dr] & (10,01) \arrow[d] & (11,10) \arrow[dr] & (10,01) \arrow[d] \\
& (010,001) \arrow[dr] & & (110,101) \arrow[dl] & & (010,001) \arrow[dr] & & (110,101) \arrow[dl] \\
& & (01010,01001) \arrow[drr] & & & & (01010,01001) \arrow[dll] & \\
& & & & (010011010,010011001) & & & \\
& & & & 0\,10011001 & & &
\end{tikzcd}$$

4.2 Complexity

Why go through this? First, by a folk theorem of Dr. Alan Laub, "*What is hard for us tends to be easy for computers (and vice versa).*" In reality this process is really easy for a computer to do. Second, the process is highly parallel, so it can be done very fast. If the numbers to be added are n bits long this takes $2(\log_2(n)+1)$ levels of logic, much better than the $n+1$ levels of logic required by ripple calculations. Thus it is O(log(n)) in time complexity. For example, for adding the 32 bit numbers considered already, conditional sum would take $2(\log_2(32)+1) = 12$ levels of logic, so on the fast logic described it would be 12ns, a huge improvement.

4.3 Recursive Build

Keeping track of everything in this tree structure is very challenging and thus error prone. We will therefore develop a recursive description, which greatly simplifies tree based algorithms.

Before we start, define $BT = B\hat{}c_{in}$. This can be done in one assign statement, since B has N bits and c_{in} only has one, and you didn't use the reducing form, Verilog will conclude you meant to **xor** c_{in} with each bit of B separately.

As Shifu said, "There is now a level zero." For us that level is adding or subtracting 2 bits without a carry coming in. We will build a special version of a full adder for this, that outputs the carry and sum of A_i and B_i and those bits +1. Call it CSFA (conditiional sum full adder). In the top level conditional sum adder, we will have a generate statement and this will be our basis clause.

We now just have to write the inductive clause, which for us will have two parts. First we need to instantiate two copies of conditional sum adder with half the bits each, to handle the upper and lower parts of A and B. We will then instantiate two muxes for combining the four outputs of the two half-sized conditional summer adders. Combining is done based on the carry values generated for each of two outputs on the smaller conditional sum adder.

The final step is to use carry in to select which of the final two conditional sum adds is correct. This is just another mux, and technically completes the 2's complement or not 2's complement.

4.4 Assignment

1. You must build the units above (upload .sv files)
   (a) Undergrads and Grads: for a 16-bit conditional sum adder.
   (b) Grads: do one of the extra options we discussed in class.
2. Write a testbench using tasks and test your units from above. (upload sv files)
3. Program your board and take a picture of it working.
4. Create a report that describes the project, design rational, testing rational, and results (screenshots and pictures of board running the code). (upload pdf)
5. Make sure to include pictures of your timing diagram (from test) and the picture of the functioning board.

Appendix A

Github

Git is a version control system originally developed by Linus Torvalds to do the Linux OS development. Github is a git web service that I will be using to develop the labs. You can get the manual and code from:

`https://github.com/KeithEvanSchubert/Advanced_Digital_Logic.git`