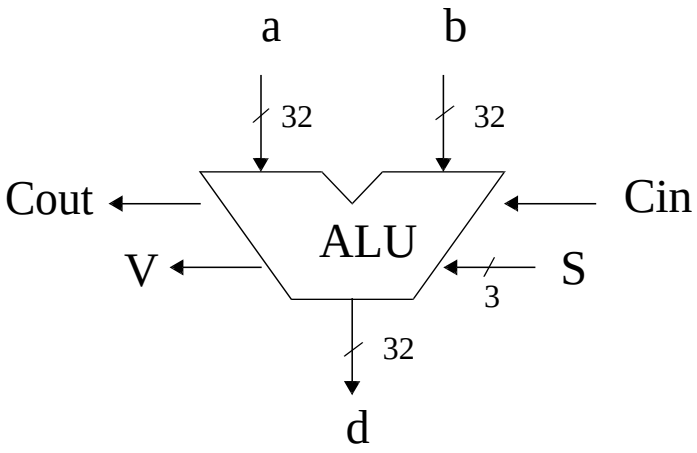
A 32-Bit ALU
Design Example

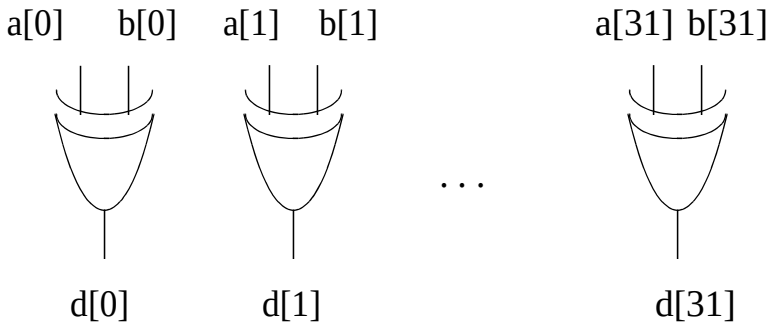
Let us try to synthesize a 32 bit ALU from a mostly behavioral description. A typical 32 bit ALU might look something like



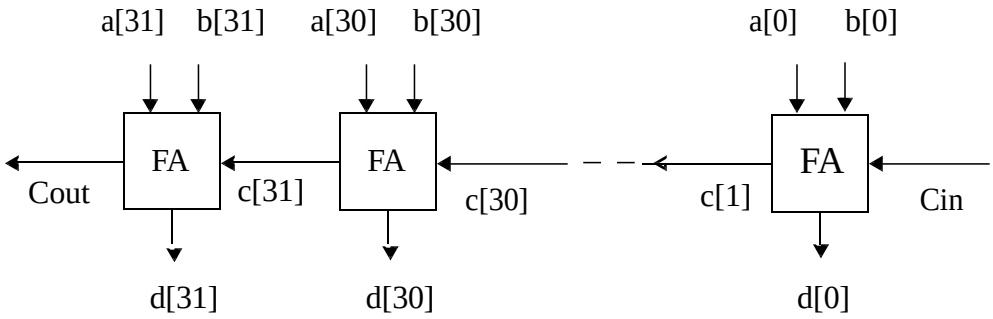
where the "d" output implements the function of "a" and "b" selected by the "S" control inputs. Note that the ALU is purely combinational logic and contains no registers and latches.

The arithmetic functions are much more complex to implement than the logic functions. Consider the circuitry needed to implement exclusive or

d = a ^ b



Contrast this with the circuit for the parallel adder..



32 bit parallel adder

The exclusive or operation requires the signals to travel through only one gate whereas the parallel adder circuit requires, in the worst case, the signal to travel through all 32 single bit adders. This has two significant consequences.

1. The worst case delay for the ALU is determined by the carry chain in the adder. It is worthwhile to use more sophisticated hardware design techniques to reduce this delay to improve the overall speed of the processor.
2. The synthesizer will spend a great deal of time trying to optimize (reduce the delay) of the carry chain for long carry chains. This is why it is impractical to synthesize a long carry chain out of random logic gates. Instead, certain well known solutions are provided for the optimizer to use for arithmetic operations. For example, the look ahead carry adder is automatically produced by the cadence tool when a "+" operator is used in behavioral Verilog.

Fast Adder Design. We will not use the "canned" solution from the "+" operator for the carry chain in our ALU design because we want to reuse the circuitry for other operations (like subtraction). A more thorough discussion of the design of high speed carry chains at the transistor level is provided in the VLSI courses.

The first step is to separate out the part of the circuit that computes the carry signals from the rest of the circuit. We do this by rewriting the logic equations for the full adder as follows.

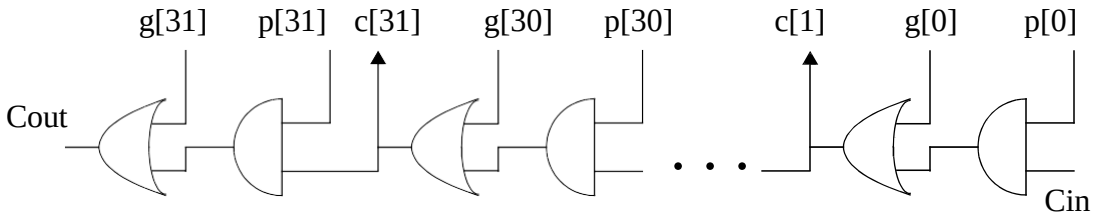
```
c[i+1] = (a[i] & b[i]) | ((a[i] | b[i]) & c[i])
       = g[i] | (p[i] & c[i])
```

where

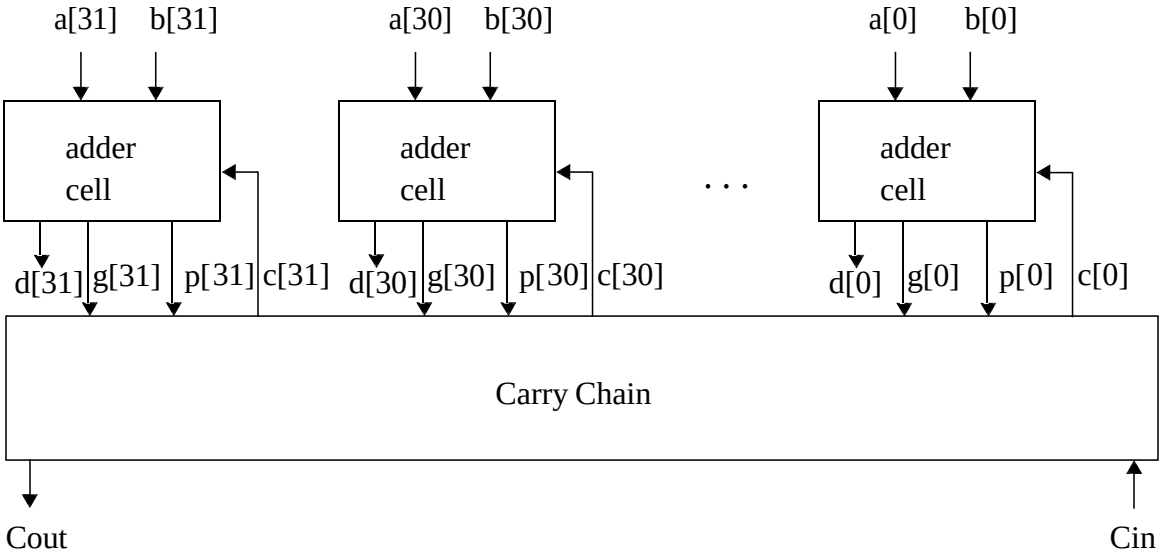
```
g[i] = a[i] & b[i]  // carry generate
p[i] = a[i] ^ b[i]  // carry propagate
```

Changing the definition of carry propagate to use exclusive or gives the same truth table for carry out.

Putting together this equation for all of the bit positions gives the following carry chain circuit.



When the carry chain circuit is used, the adder circuit becomes



Notice that the worst case delay is no longer through the adder cell and we can write a simple behavioral model for the adder cell.

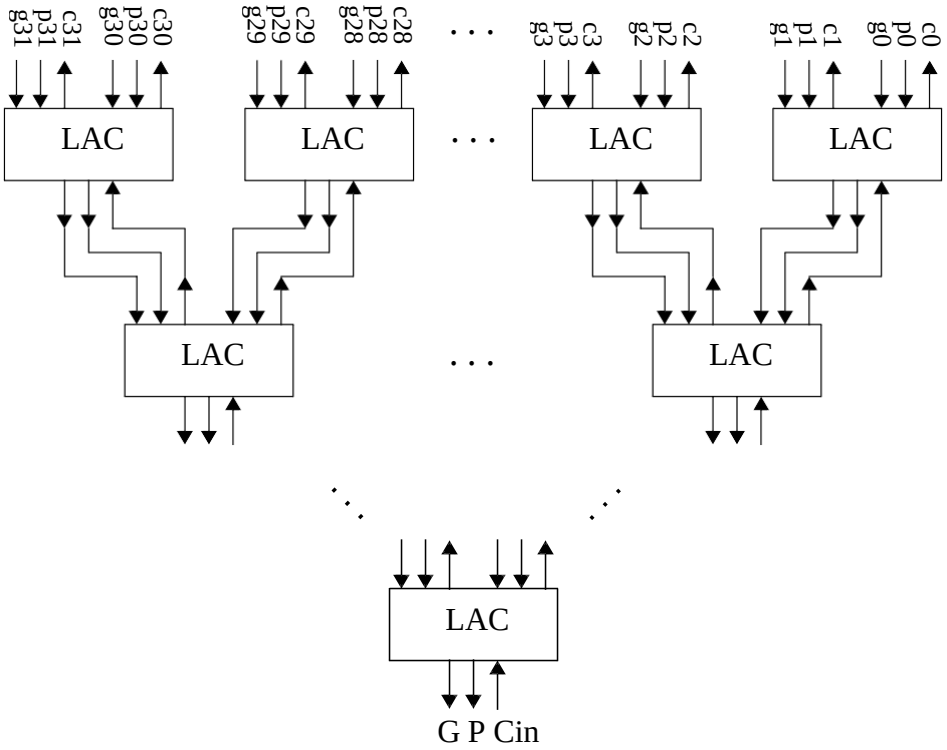
```
module add_cell (d, g, p, a, b, c);
    output d, g, p;
    input a, b, c;

    assign g = a & b;
    assign p = a ^ b;
    assign d = p ^ c;
endmodule
```

We cannot use a purely behavioral description for the carry chain because the synthesizer will spend too much time trying to optimize 32 levels of and-or gates. The optimizer will

try to find a circuit with a smaller worst case delay. We will use a combination of structural and behavioral descriptions. The structural description will divide the carry chain into smaller modules for which we can have a behavioral description that is easier to optimize.

Since the carry chain is connected as a linear chain, the delay for a 32-bit carry chain is about 32 times slower than the delay through a single bit. In general, the worst case delay for the carry chain will increase linearly with N, the number of bits in a word.

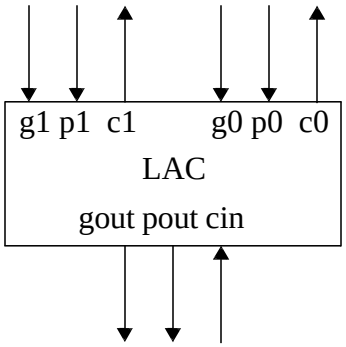
There are several well known techniques that give a delay proportional to log N if properly implemented. When N > 8, it is almost always faster to use one of these techniques. We will use a "look ahead carry" (LAC) technique to improve performance. The carry chain is implemented as a binary tree of look ahead carry cells.



For a 32-bit LAC adder, there are 5 rows of LAC circuits, each row with half the number of LAC circuits as the row above it. The worst case (longest) delay through the tree starts from the a, b inputs going down the tree branches through the generate and propagate signals to the root of the tree, and then coming back up the tree branches through the carry signals. There are 5 ($\log_2$ N in general) LAC's along each branch so that the worst case delay is proportional to log N. Thus, for the 32-bit adder, the worst case delay goes through 9 LAC's (5 down and 4 back up) which is much better than going through 32 pieces of the linear carry chain.

There is another version of this circuit that only traverses the tree once instead of twice (see the Advanced VLSI course).

Each LAC circuit is a copy of the following.



c1 = g0 | p0 & cin
c0 = cin
gout = g1 | p1 & g0
pout = p1 & p0

We must use a structural description for the LAC circuits in the tree. We can write a behavioral description of the LAC equations above and let the optimizer choose the best logic gates to implement the LAC module.

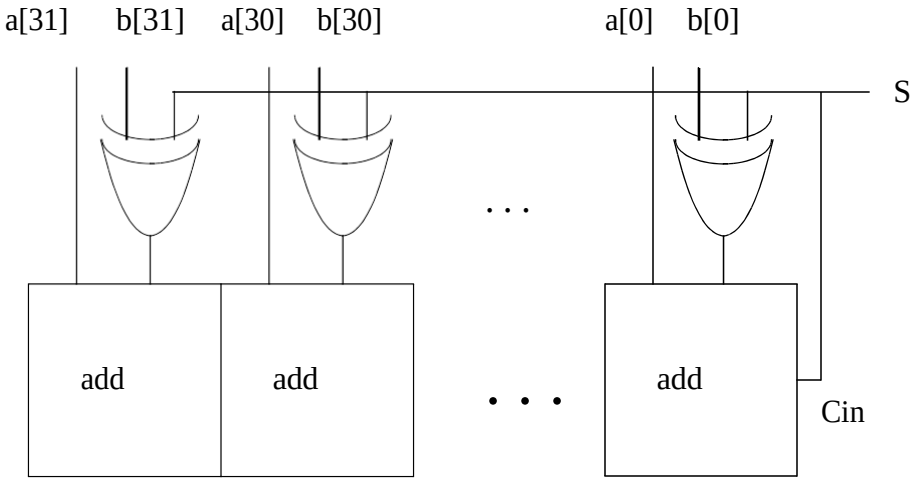
Adder/Subtracter Design. The adder can be turned into an adder subtracter by noting that

a - b = a + (-b)

where -b is formed (in the 2's complement representation) by inverting all of the b-bits and adding one to the lease significant bit. For the ALU, we want to control when the b-bits are inverted, i.e.

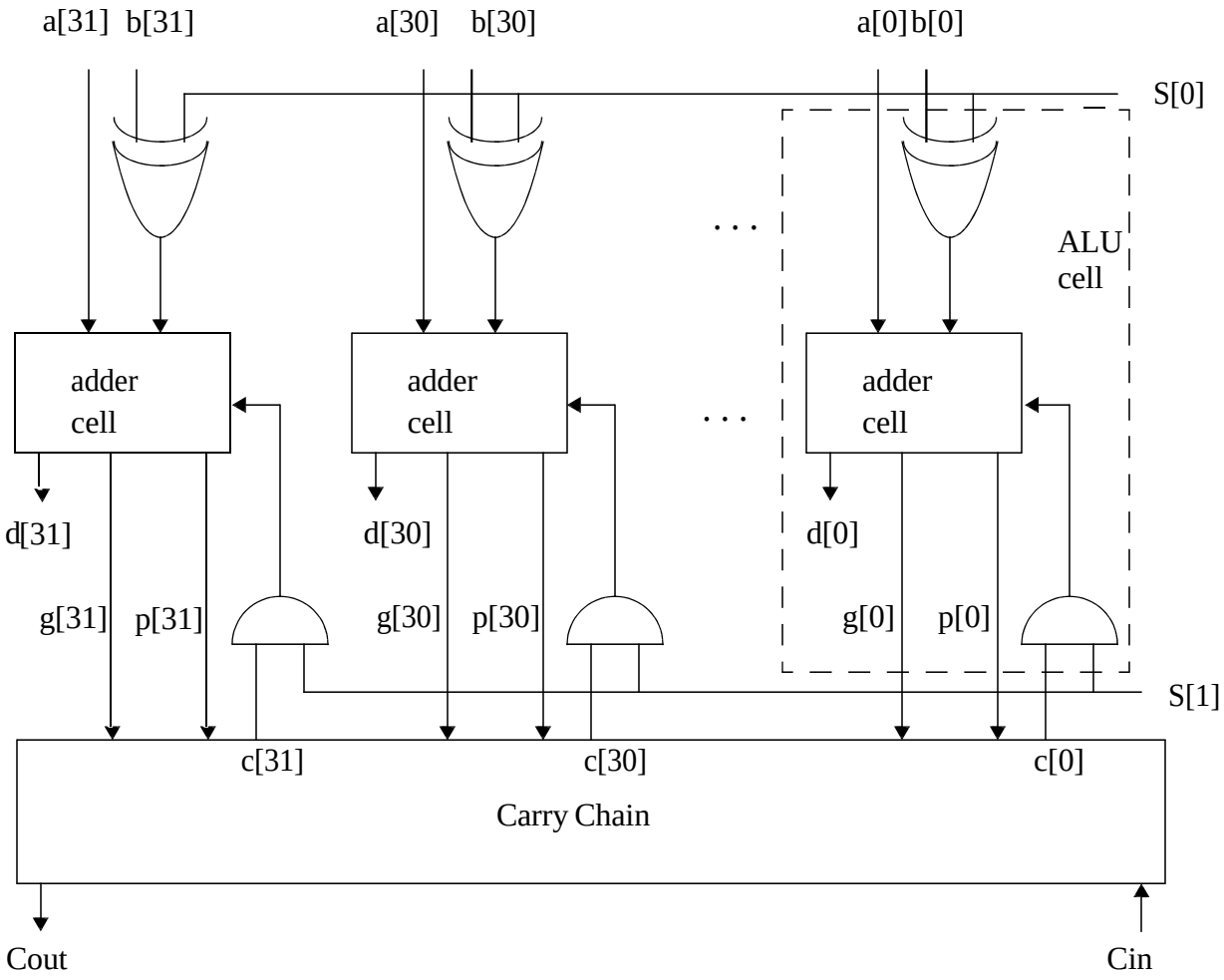
- addition: do not invert b-bits, carry in is zero
- subtraction: invert b-bits, carry in is one

The following circuit implements this behavior.



When S=0, the circuit adds; when S=1, the circuit subtracts.

Logic Operations. The logic operations are implemented by disabling the carry chain. Extra circuitry as needed can be added to the adder/subtracter cells to implement the desired logic functions for the ALU. The carry chain is disabled by forcing the carry in to each bit position to be '0' (not open circuit). The following circuit disables the carry this way.



The lines, S[1], S[0], select the following circuit functions.

S[1],S[0]	Function
00	xor
01	xnor
10	add
11	subtract

We now modify our earlier Verilog code for the add_cell module to make the Verilog for 1-bit of this ALU .

```
module alu_cell (d, g, p, a, b, c, S);
    output d, g, p;
    input a, b, c;
    input [1:0] S;

//internal nodes for b,c inputs to adder
    wire cint, bint;

    assign bint = S[0] ^ b;
    assign g = a & bint;
    assign p = a ^ bint;
    assign cint = S[1] & c;
    assign d = p ^ cint;
endmodule
```

Use internal node here so that b can be inverted

Use internal node here so that carry can be disabled

This one bit ALU can now be put together structurally in a higher level cell to make the entire ALU. It is no advantage to try a multibit behavioral description since we use a structural description for the carry chain.

Structural Hierarchy for the ALU

Using structural Verilog to limit the topological choices greatly reduces the amount of work the optimizer has to do. It also can prevent the optimizer from finding a true optimum circuit. For example, the topology of the linear carry chain circuit forces the optimizer to find a solution with delay proportional to N, rather than log N.

In the following example, we use purely behavioral code for the ALU cell and LAC circuits. Then we use structural Verilog to put together the circuits in the binary tree. Even though the optimizer is free to use any logic gates it wants out of the cell library, only a small part of the circuit is being optimized which allows the optimizer to find a good solution in a reasonably short period of time.

The first design project requires adding extra functionality to the alu_cell module.

```
module alu_cell (d, g, p, a, b, c, S);
    output d, g, p;
    input a, b, c;
    input [2:0] S;

    wire cint, bint;

    assign bint = S[0] ^ b;
    assign g = a & bint;
    assign p = a ^ bint;
    assign cint = S[1] & c;
    assign d = p ^ cint;
endmodule
```

extra select line added

Add extra functionality for d depending on S[2]

The lac module is the LAC circuit discussed earlier.

```
module lac (c, gout, pout, Cin, g, p);
    output [1:0] c;
    output gout, pout;
    input Cin;
    input [1:0] g, p;

    assign c[0] = Cin;
    assign c[1] = g[0] | ( p[0] & Cin );
    assign gout = g[1] | ( p[1] & g[0] );
    assign pout = p[1] & p[0];
endmodule
```

Now start building up the LAC tree by using structural Verilog to make a 2-level LAC.

```
module lac2 (c, gout, pout, Cin, g, p);
    output [3:0] c;
    output gout, pout;
    input Cin;
    input [3:0] g, p;

    wire [1:0] cint, gint, pint;

    lac leaf0(
        .c(c[1:0]),
        .gout(gint[0]),
        .pout(pint[0]),
        .Cin(cint[0]),
        .g(g[1:0]),
        .p(p[1:0])
    );

    lac leaf1(
        .c(c[3:2]),
        .gout(gint[1]),
        .pout(pint[1]),
        .Cin(cint[1]),
        .g(g[3:2]),
        .p(p[3:2])
    );

    lac root(
        .c(cint),
        .gout(gout),
        .pout(pout),
        .Cin(Cin),
```

```
        .g(gint),
        .p(pint)
    );
endmodule
```

Now make a 3-level LAC out of 2-level components and a single 1-level.

```
module lac3 (c, gout, pout, Cin, g, p);
    output [7:0] c;
    output gout, pout;
    input Cin;
    input [7:0] g, p;

    wire [1:0] cint, gint, pint;

    lac2 leaf0(
        .c(c[3:0]),
        .gout(gint[0]),
        .pout(pint[0]),
        .Cin(cint[0]),
        .g(g[3:0]),
        .p(p[3:0])
    );

    lac2 leaf1(
        .c(c[7:4]),
        .gout(gint[1]),
        .pout(pint[1]),
        .Cin(cint[1]),
        .g(g[7:4]),
        .p(p[7:4])
    );

    lac root(
        .c(cint),
        .gout(gout),
        .pout(pout),
        .Cin(Cin),
        .g(gint),
        .p(pint)
    );
endmodule
```

Make lac4 from 2 lac3's and a lac and make lac5 from 2 lac4's and a lac in the same manner as lac3 was made from 2 lac2's and a lac except for the different size busses.

The 5-level LAC implements the carry chain needed for the 32-bit ALU. Now connect the alu_cell's to the lac5.

```
module alu32 (d, a, b, Cin, S);
    output [31:0] d;
    input [31:0] a, b;
    input Cin;
    input [2:0] S;

    wire [31:0] c, g, p;

    alu_cell cell[31:0] (
        .d(d),
        .g(g),
        .p(p),
        .a(a),
        .b(b),
        .c(c),
        .S(S)
    );

    lac5 lac (
        .c(c),
        .gout( ),
        .pout( ),
        .Cin(Cin),
        .g(g),
        .p(p)
    );
endmodule
```

Makes 32 instances of alu_cell

These outputs are purposely left unconnected. Never leave inputs unconnected!

The array of alu_cell instances is equivalent to 32 instances as follows.

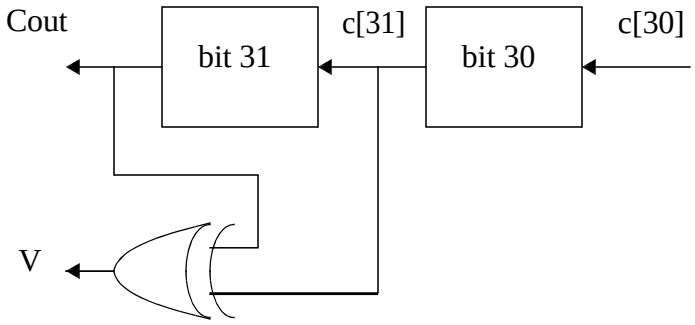
```
    alu_cell cell31 (
        .d(d[31]),
        .g(g[31]),
        .p(p[31]),
        .a(a[31]),
        .b(b[31]),
        .c(c[31]),
        .S(S)
    );

// 30 more instances here

    alu_cell cell0 (
        .d(d[0]),
```

```
        .g(g[0]),
        .p(p[0]),
        .a(a[0]),
        .b(b[0]),
        .c(c[0]),
        .S(S)
    );
```

Note that only the single bit lines in the alu_cell are expanded to connect to different bit of the 32-bit bus. The S bus is not expanded.

Overflow

Because there are a finite number of bits in the ALU, it is not always possible to represent the results of addition and subtraction operations. For unsigned numbers, a carry out, Cout, of 1 after an addition indicates that the sum was too large to represent in the number of bits in the operands. A Cout of 0 after a subtraction indicates that the difference was negative which cannot be represented unless the numbers are signed. For signed numbers, the overflow output, V, indicates when the 2's complement number for the result cannot be represented in the number of bits in the slice. The 2's complement overflow signal can be implemented as shown.



The extra circuitry to compute Cout and V should be added to the adder32.v file. If behavioral code is used, then the adder32.v module must be synthesized even though most of the code is structural.

```
module alu32 (d, Cout, V, a, b, Cin, S);  //final version
    output [31:0] d;
    output Cout, V;
    input [31:0] a, b;
    input Cin;
    input [2:0] S;

    wire [31:0] c, g, p;
    wire gout, pout;
```

```
    alu_cell cell[31:0] (
        .d(d),
        .g(g),
        .p(p),
        .a(a),
        .b(b),
        .c(c),
        .S(S)
    );

    lac5 lac (
        .c(c),
        .gout(gout),
        .pout(pout),
        .Cin(Cin),
        .g(g),
        .p(p)
    );

    assign Cout = gout | (pout & Cin);
    assign V = Cout ^ c[31];
endmodule
```