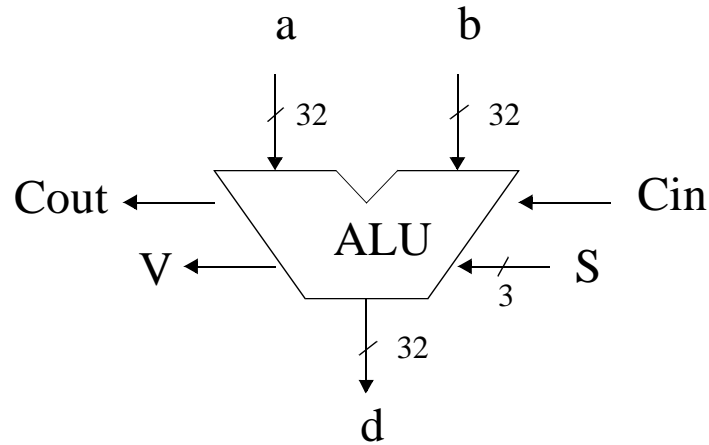


A 32-Bit ALU Design Example

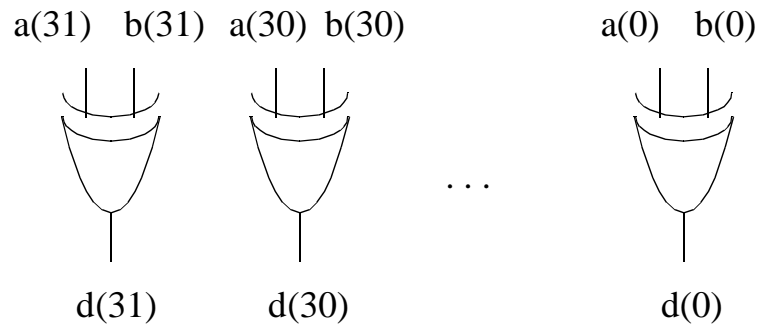
A typical 32 bit ALU might look something like



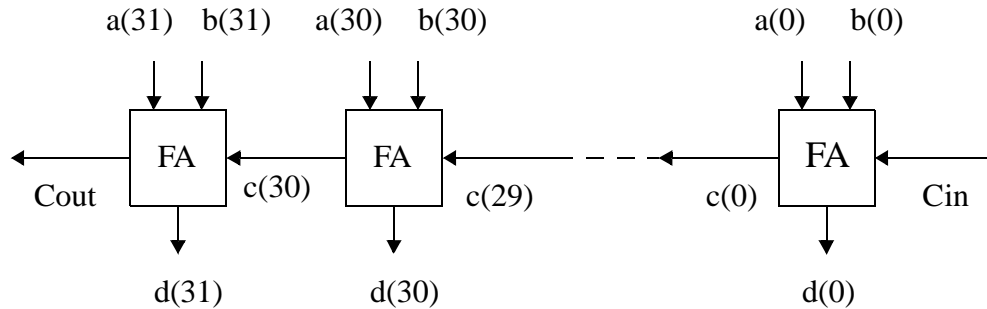
where the “d” output implements the function of “a” and “b” selected by the “S” control inputs. Note that the ALU is purely combinational logic and contains no registers and latches.

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

$$d \leq a \text{ xor } b$$



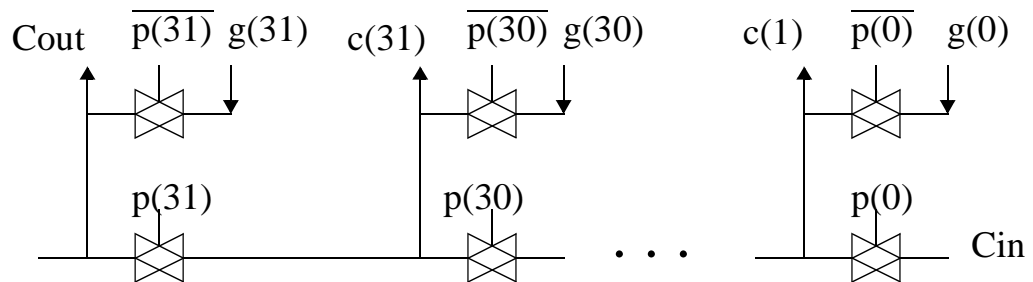
Contrast this with the circuit for the parallel adder.



The xor 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.
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 all 32 bits at once.

Fast Adder Design. The design of high speed carry chains is beyond the scope of this course (take the VLSI course). We will approximate the performance of a high speed carry chain with the following Manchester Carry Chain circuit.

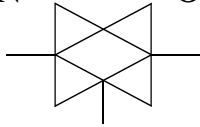


where the carry generate, g, and carry propagate, p, functions are defined as follows.

$$g(i) \leq a(i) \text{ and } b(i)$$

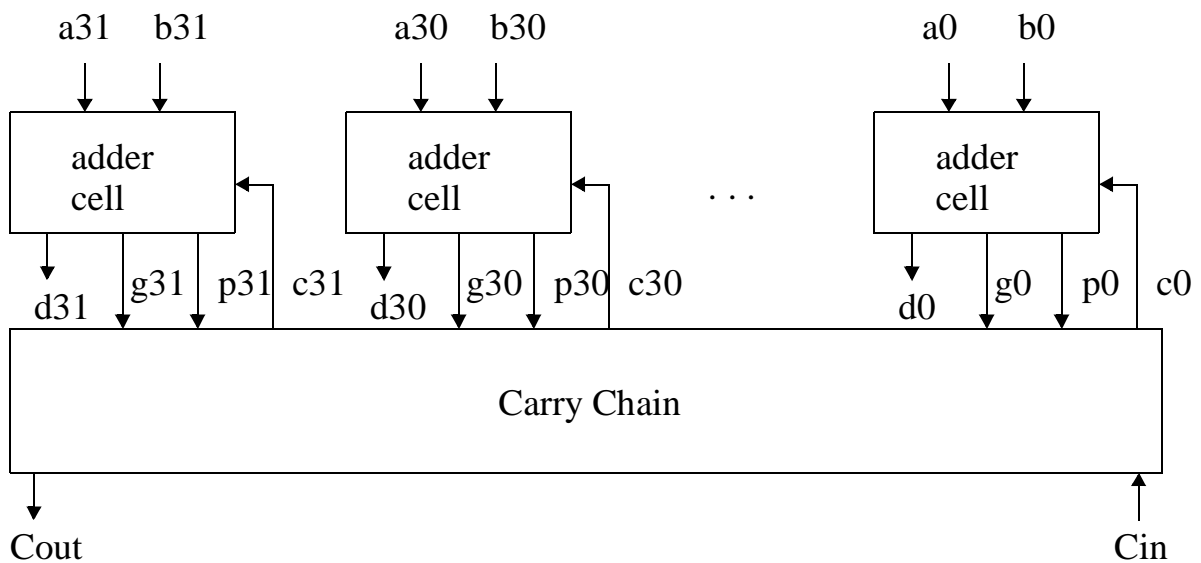
$$p(i) \leq a(i) \text{ xor } b(i)$$

The transmission gate is a controllable switch as the following table indicates. The Z stands for a high impedance (open switch) output.

<div style="display: inline-block; text-align: center;"> IN  OUT CONTROL </div>	control	in	out
	0	0	Z
	0	1	Z
	1	0	0
	1	1	1

Note: the delay predicted by the simulator for the transmission gate circuit is unrealistically short and more sophisticated techniques are really used.

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

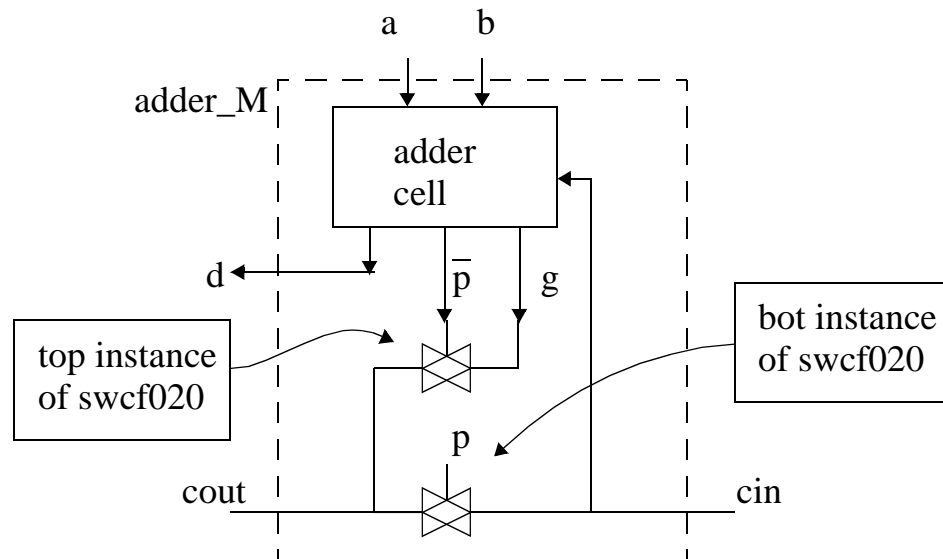


Notice that the worst case delay is no longer through the adder cell and we can write a simple behavioral model to use with the synthesizer.

```
entity add_cell is port(
  signal a,b,c: in std_logic;
  signal d,g,p: out std_logic;
end add_cell;
```

```
architecture behav of add_cell is
begin
  g <= a and b;
  p <= a xor b;
  d <= p xor c;
end behav;
```

We cannot use a behavioral description for the carry chain because the synthesizer does not use the transmission gate when it synthesizes circuits. We must use a structural description to put in the transmission gates. Since the Manchester carry chain is the same for each bit position, we can use the following mixture of behavior and structural description.



```
entity add_M is port(
    signal a,b,cin: in std_logic;
    signal d,cout: out std_logic);
end add_M;
```

```
architecture combo of add_M is
    signal p,pnot,g: std_logic;
    component swcf020 port(
        signal DATA1: in std_logic;
        signal CTL2: in std_logic;
        signal O: out std_logic);
    end component;
```

begin

```
g <= a and b;  
p <= a xor b;  
pnot <= not p;  
d <= p xor cin;
```

behavioral description of add function

```
top: swcf020    ↙
port map(
    DATA1 => g,
```

Structural description of carry chain

$$);$$

```
bot: swcf020
port map(
    DATA1 => cin,
    CTL2 => p,
    O => cout
);
```

end combo;

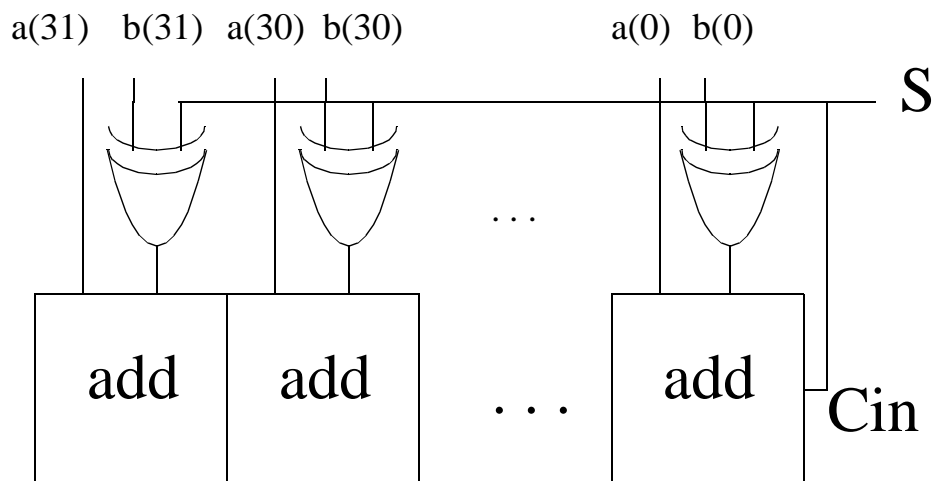
Adder/Subtractor Design. The adder can be turned into an adder subtracter by noting that

$$a - b = a + (-b)$$

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

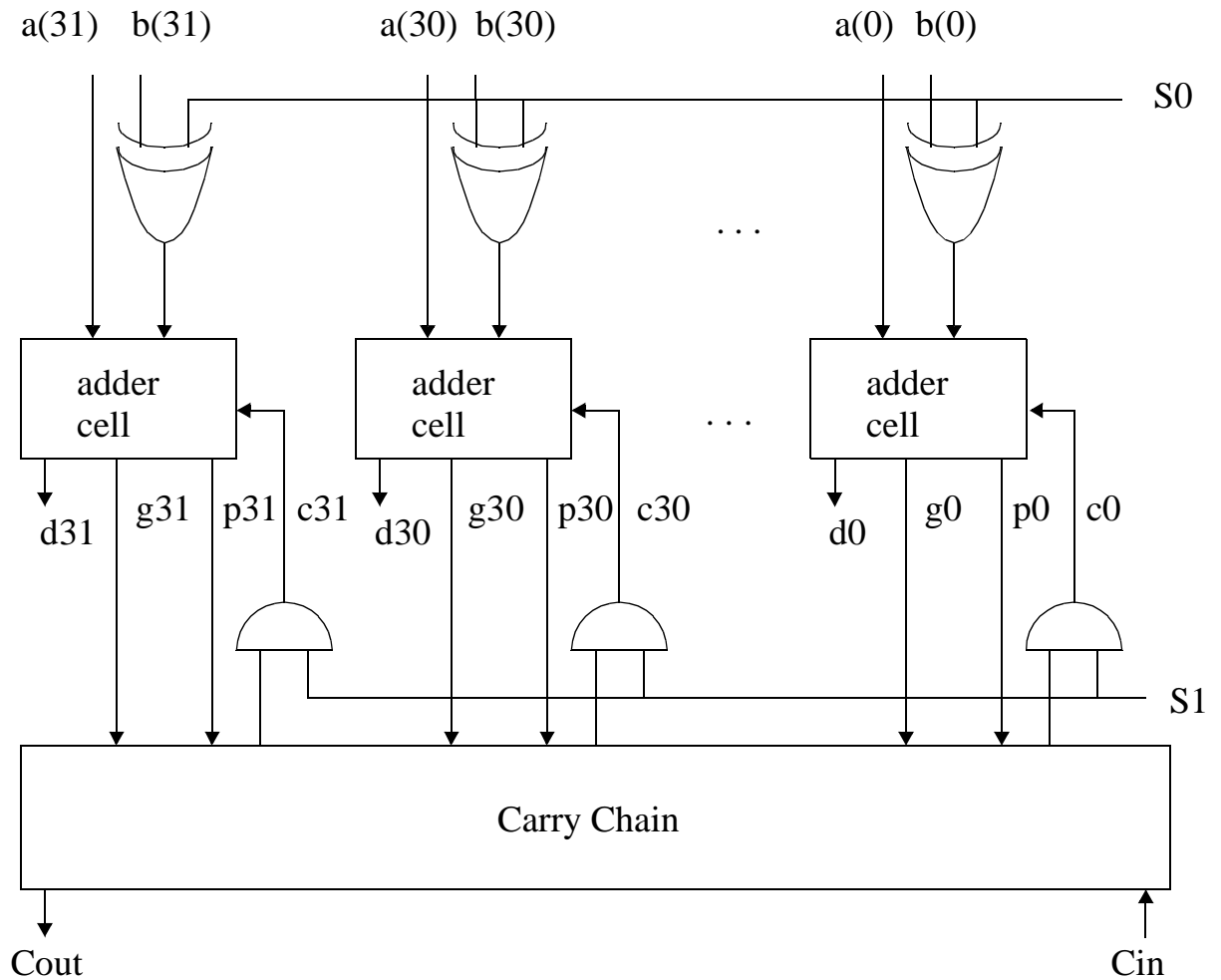
- addition: do not invert b-bits, carry in is zero
- subtraction: invert b-bits, carry in is one

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/subtractor 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, $S(1), S(2)$, select the following circuit functions.

$S1, S0$	Function
00	xor
01	xnor
10	add
11	subtract

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

```

entity ALU_cell is port(
    signal S: in std_logic_vector(1 downto 0);
    signal a,b,cin: in std_logic;
    signal d,cout: out std_logic);
end ALU_cell;

architecture combo of ALU_cell is
    signal p,pnot,g: std_logic;
    --internal nodes for b,c inputs to adder
    signal c,bint: std_logic
    component swcf020 port(
        signal DATA1: in std_logic;
        signal CTL2: in std_logic;
        signal O: out std_logic);
    end component;

begin
    bint <= S(0) xor b;
    g <= a and bint;
    p <= a xor bint;
    pnot <= not p;
    d <= p xor c;
    c<=S(1) and cin;

    top: swcf020
    port map(
        DATA1=>g,
        CTL2=>pnot,
        O=>cout
    );

    bot: swcf020
    port map(
        DATA1=>cin,
        CTL2=>p,
        O=>cout
    );
end combo;

```

Use internal node here so that b can be inverted

Use internal node here so that carry can be disabled

The carry chain description remains the same

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 must use a structural description for the carry chain.

Synthesizing the ALU

Lets suppose that we make a 32 bit ALU out of 4 bit slices as

```
entity alu32 is port(
  signal S: in std_logic_vector(2 downto 0);
  signal a,b: in std_logic_vector(31 downto 0);
  signal d: out std_logic_vector(31 downto 0);
  signal Cin: in std_logic;
  signal Cout: out std_logic;
  signal V: out std_logic);
end alu32;
```

only lower case
in entity name

architecture combo of alu32 is

```
  signal c: std_logic_vector(7 downto 0);
  component alu4 port(
    signal S: in std_logic(2 downto 0);
    signal a,b: in std_logic(3 downto 0);
    signal d: out std_logic_vector(3 downto 0);
    signal Cin: in std_logic;
    signal Cout: out std_logic;
    signal V: out std_logic);
  end component;
```

begin

```
  slice7: alu4
  port map(
    S => S,
    a => a(31 downto 28),
    b => b(31 downto 28),
    d => d(31 downto 28),
    Cin => c(7),
    Cout => Cout
    V => V
  );
```

Last slice done separately
so that V can be connected

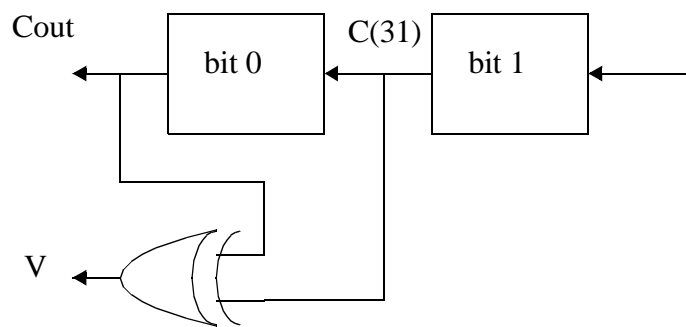
```
  c(0) <= Cin;
  gen026: for i in 0 to 6 generate
    slice: alu4 port map(
      S=>S,
      Cin=>c(i),
      Cout=>c(i+1)
      a=>a(4*i+3 downto 4*i),
      b=>b(4*i+3 downto 4*i),
      d=>d(4*i+3 downto 4*i),
    );
  end generate;
```

Never leave inputs
unconnected!!

V (an output) is
purposely left
unconnected

end combo;

The overflow output, V, from each slice indicates when the 2's complement number for the result cannot be represented in the number of bits in the slice. When we wire the slices together, the overflows from the three least significant slices can be ignored because the other slices provide more bits. The overflow from the most significant slice cannot be ignored since there are no more significant bits in other slices. The 2's complement overflow signal can be implemented as shown.



$$V \leq Cout \text{ xor } C(31);$$

Let us suppose further that alu4 is a structural combination of 4 alu_cell's in a similar manner as alu32 is formed from alu4's.

We can now use the vhdl synthesizer program to synthesize the ALU in a variety of ways.

1. Synthesize entire ALU at once with the following commands:

```
vsyn -vhdl alu_cell
vsyn -vhdl alu4
vsyn -vhdl alu32
vsyn -synth alu32
```

2. Synthesize only 4 bits of the ALU and combine 8 slices together to make the 32 bit ALU:

```
vsyn -cleanup
vsyn -vhdl alu32
vsyn -synth alu32
vsyn -vhdl alu_cell
vsyn -vhdl alu4
vsyn -synth alu4
```

Makes a
"black box"
for alu4

3. Synthesize only a single bit of the ALU and put the 32 bits together to make the 32 bit ALU:

```
vsyn -cleanup
vsyn -vhdl alu32
vsyn -synth alu32
vsyn -vhdl alu4
vsyn -synth alu4
vsyn -vhdl alu_cell
vsyn -synth alu_cell
```

Makes a
“black box”
for alu4

Makes a
“black box”
for alu_cell

Beware a common error. Make sure your logic functions are completely specified. Any unspecified outputs cause the synthesizer to generate latches which should not be there. To avoid these errors, always use “else” with “when”

```
x <= y when z else w;
```

and always set a default value for all outputs at the beginning of a process block.

```
sample: process(x,y,z)
begin
    out1 <= x;
    out2 <= y;
    if ...
```

Once you have synthesized the ALU, simulate it with the following commands.

```
vsm alu32
viewsim alu32 -alu32.cmd -nographics
```

or click on the appropriate icons in the powerview window if you are using graphics.