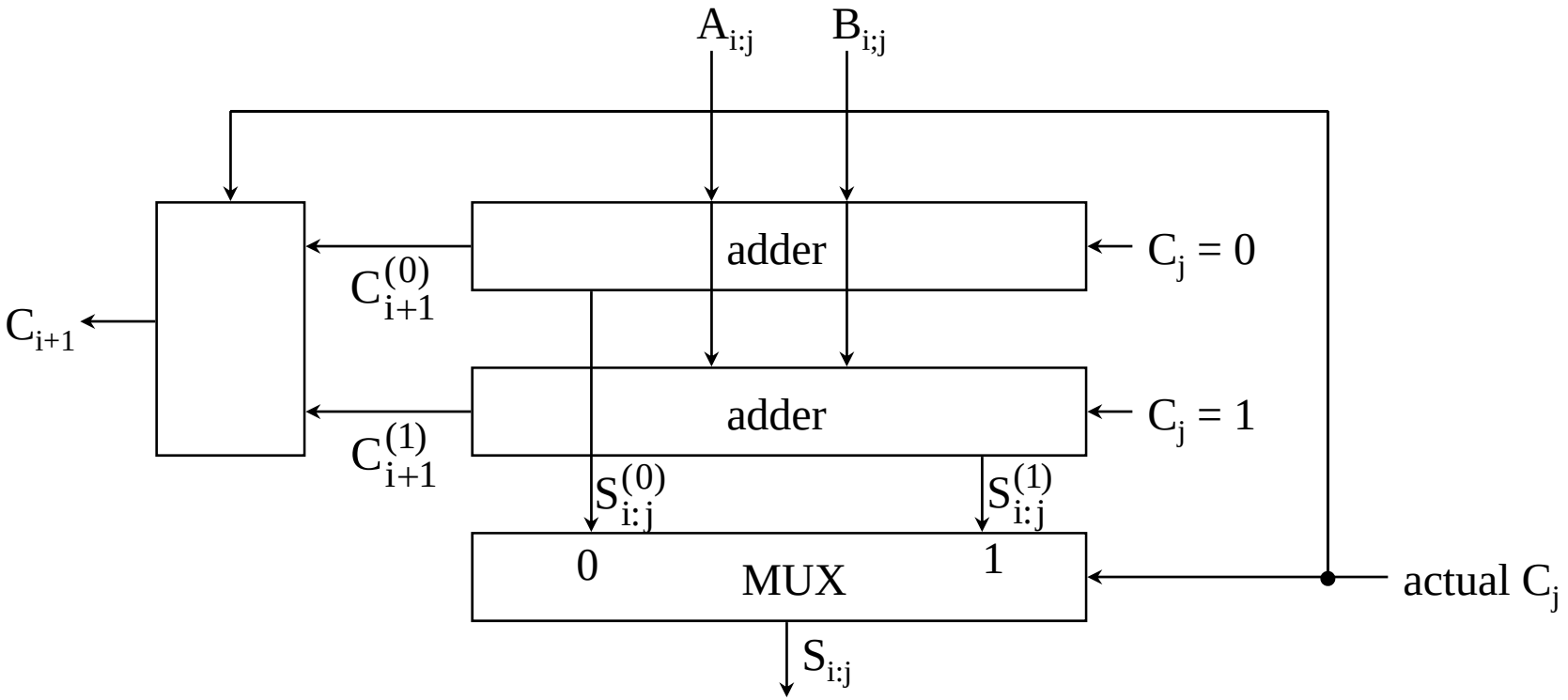
Architecture of Carry-Select Adders

- Calculate sums and carry out for a group of bits twice (in parallel), assuming that the carry in to the group is a 0 and also assuming that the carry in is a 1. When the carry in arrives, use it to select the correct sum and carry out bits.
- In a carry-select architecture, group-level carries ripple between groups. In some designs, the number of bits within a group is increased for the higher-order groups to balance the delays.



© 2011 by Gerald E. Sobelman

Carry Look-Ahead (CLA) Adders

- Given two multi-bit operands A and B, we define the functions P_i (propagate), G_i (generate) and $Psum_i$ (partial sum) at each bit position as follows: (Note that some references define propagate using the exclusive OR function.)

$$P_i = A_i + B_i, \quad G_i = A_i B_i, \quad Psum_i = A_i \oplus B_i$$

- Assume that the carry into bit position i is C_i. Then, the sum at bit position i, S_i, and carry into the next higher bit position, C_{i+1}, are given by:

$$S_i = Psum_i \oplus C_i, \; C_{i+1} = G_i + P_i C_i$$

- A CLA adder uses the fact that these functions can be computed at all bit positions in parallel, and this information can be used to speed up the carry (and hence the sum) computations.

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$$

$$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0)$$

$$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

© 2011 by Gerald E. Sobelman

Carry Look-Ahead (CLA) Adders - Continued

- The previous equations determine the first four carries in parallel. While the previous set of substitutions could be continued to higher-order carries, they become impractical due to the the large number of terms involved and large fan-in gates that would be required.
- One alternative is to construct a hierarchical structure consisting of *groups, blocks* and *sections*. For example, we can define the group generate and propagate functions as:

$$G_{3:0} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \ , \ P_{3:0} = P_3P_2P_1P_0$$

- Using these functions, the equation for C_4 can be written as follows:

$$C_4 = G_{3:0} + P_{3:0}C_0$$

- Similarly, we can define group generate and propagate functions for other, non-overlapping 4-bit groups:

$$G_{i+3:i} = G_{i+3} + P_{i+3}G_{i+2} + P_{i+3}P_{i+2}G_{i+1} + P_{i+3}P_{i+2}P_{i+1}G_i$$
$$P_{i+3:i} = P_{i+3}P_{i+2}P_{i+1}P_i$$

© 2011 by Gerald E. Sobelman

Carry Look-Ahead (CLA) Adders - Continued

- Given the definitions of these group generate and propagate functions, we can create the following group carries:

$$C_8 = G_{7:4} + P_{7:4}C_4 = G_{7:4} + P_{7:4}\left(G_{3:0} + P_{3:0}C_0\right)$$
$$= G_{7:4} + P_{7:4}G_{3:0} + P_{7:4}P_{3:0}C_0$$
$$C_{12} = G_{11:8} + P_{11:8}C_8 = G_{11:8} + P_{11:8}\left(G_{7:4} + P_{7:4}G_{3:0} + P_{7:4}P_{3:0}C_0\right)$$
$$= G_{11:8} + P_{11:8}G_{7:4} + P_{11:8}P_{7:4}G_{3:0} + P_{11:8}P_{7:4}P_{3:0}C_0$$
$$C_{16} = G_{15:12} + P_{15:12}C_{12}$$
$$= G_{15:12} + P_{15:12}\left(G_{11:8} + P_{11:8}G_{7:4} + P_{11:8}P_{7:4}G_{3:0} + P_{11:8}P_{7:4}P_{3:0}C_0\right)$$
$$= G_{15:12} + P_{15:12}G_{11:8} + P_{15:12}P_{11:8}G_{7:4}$$
$$+ P_{15:12}P_{11:8}P_{7:4}G_{3:0} + P_{15:12}P_{11:8}P_{7:4}P_{3:0}C_0$$

- This can be extended to another level of hierarchy (i.e., *blocks*) by forming block generate and propagate functions, which allows us to get block carries. For example:

$$G_{15:0} = G_{15:12} + P_{15:12}G_{11:8} + P_{15:12}P_{11:8}G_{7:4} + P_{15:12}P_{11:8}P_{7:4}G_{3:0}$$
$$P_{15:0} = P_{15:12}P_{11:8}P_{7:4}P_{3:0} \Rightarrow C_{16} = G_{15:0} + P_{15:0}C_0$$

© 2011 by Gerald E. Sobelman

Parallel Prefix Adders

- Parallel prefix adders make use of recursive relations between group propagate and generate functions. Consider 3 bit positions i, m and j, where i > m > j:

$$P_{i:j} = \left(P_i P_{i-1} \cdots P_m\right)\left(P_{m-1} P_{m-2} \cdots P_j\right) = P_{i:m} P_{m-1:j}$$

$$\begin{aligned} G_{i:m} &= G_i + P_i G_{i-1:m} \\ &= G_i + P_i\left(G_{i-1} + P_{i-1} G_{i-2:j}\right) = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2:m} \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_{m+1} G_m \end{aligned}$$

$$\begin{aligned} G_{i:j} &= \left(G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_{m+1} G_m\right) + \left(P_i P_{i-1} \cdots P_m\right) G_{m-1:j} \\ &= G_{i:m} + P_{i:m} G_{m-1:j} \end{aligned}$$

- Define an operator "$\circ$" which implements the recursive relations for $P_{i:j}$ and $G_{i:j}$, i.e.:

$$\begin{aligned} (P_{i:m}, G_{i:m}) \circ (P_{m-1:j}, G_{m-1;j}) &= (P_{i:m} P_{m-1:j}, G_{i:m} + P_{i:m} G_{m-1;j}) \\ &= (P_{i:j}, G_{i:j}) \end{aligned}$$

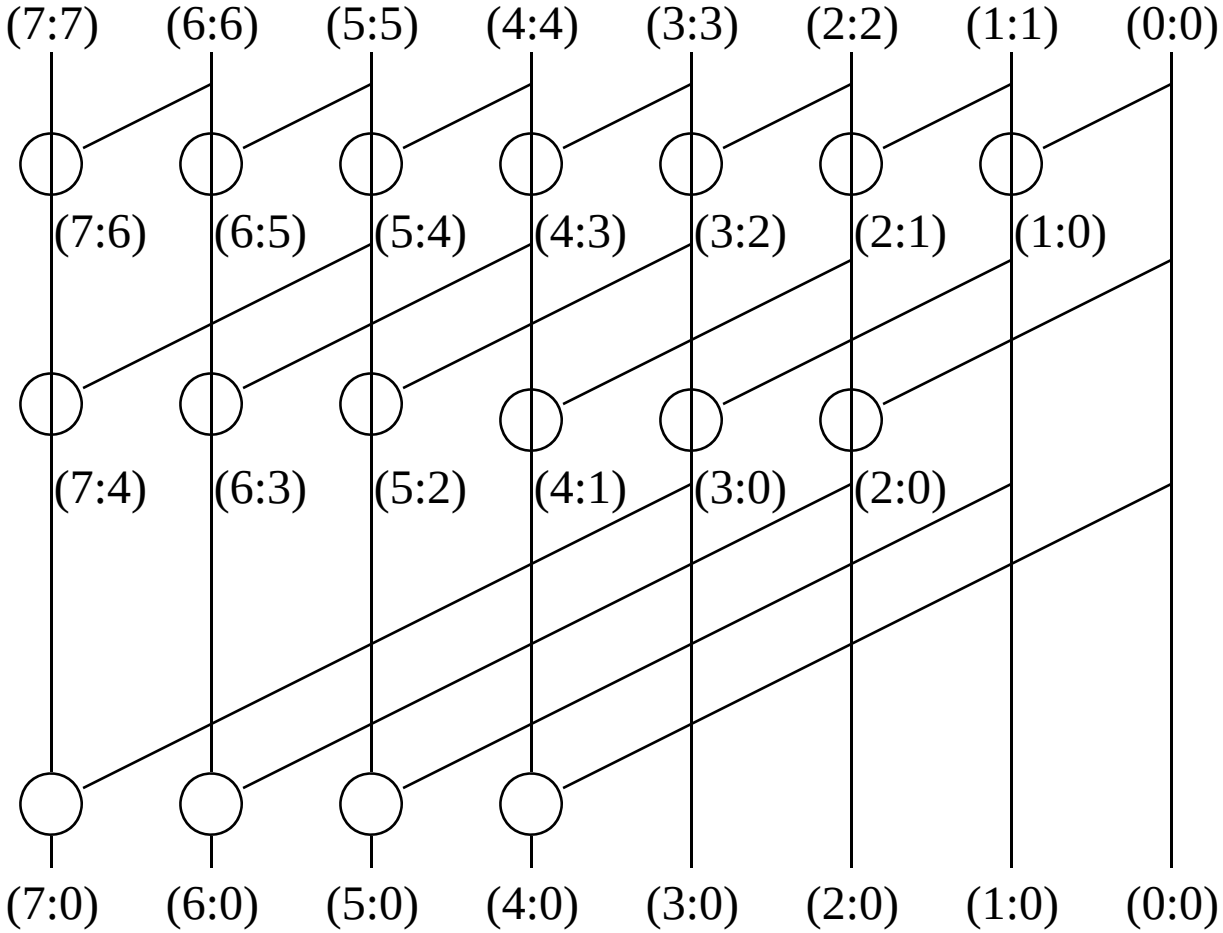
- It can be shown that this operator is *associative* (i.e., a series of $\circ$ operators can be evaluated in any order). This provides a great deal of flexibility in devising various adder architectures, as illustrated on the following pages.

Simon Knowles, "A Family of Adders," *15th IEEE Symposium on Computer Arithmetic,* pp. 277-284, 2001.

© 2011 by Gerald E. Sobelman

Kogge-Stone (KS) Architecture

- The n-bit KS design achieves the minimum logical depth of $\log_2 n$ levels. It combines values at spans of distance 2^0, 2^1, 2^2, ... , 2^{n-1}. The 8-bit KS design is:



© 2011 by Gerald E. Sobelman

Verilog Code for the Kogge-Stone Adder: Part 1 of 4

```
module opo(p1, g1, p2, g2, p3, g3);

input  p1, g1, p2, g2;
output p3, g3;

assign p3 = p1 & p2;
assign g3 = g1 | (p1 & g2);

endmodule

module kogge_stone(a, b, c0, s, c8);

input  [7:0] a, b;
input  c0;
output [7:0] s;
output c8;

wire   [7:0] p, g, psum;
wire   c1, c2, c3, c4, c5, c6, c7;
```

© 2011 by Gerald E. Sobelman

Verilog Code for the Kogge-Stone Adder: Part 2 of 4

```
// form the propagate, generate and psum signals from the input operands
or(p[0], a[0], b[0]);
or(p[1], a[1], b[1]);
or(p[2], a[2], b[2]);
or(p[3], a[3], b[3]);
or(p[4], a[4], b[4]);
or(p[5], a[5], b[5]);
or(p[6], a[6], b[6]);
or(p[7], a[7], b[7]);

and(g[0], a[0], b[0]);
and(g[1], a[1], b[1]);
and(g[2], a[2], b[2]);
and(g[3], a[3], b[3]);
and(g[4], a[4], b[4]);
and(g[5], a[5], b[5]);
and(g[6], a[6], b[6]);
and(g[7], a[7], b[7]);

xor(psum[0], a[0], b[0]);
xor(psum[1], a[1], b[1]);
xor(psum[2], a[2], b[2]);
xor(psum[3], a[3], b[3]);
xor(psum[4], a[4], b[4]);
xor(psum[5], a[5], b[5]);
xor(psum[6], a[6], b[6]);
xor(psum[7], a[7], b[7]);
```

© 2011 by Gerald E. Sobelman

Verilog Code for the Kogge-Stone Adder: Part 3 of 4

```
// first level of operator o
opo opo01(p[7], g[7], p[6], g[6], p76, g76);
opo opo02(p[6], g[6], p[5], g[5], p65, g65);
opo opo03(p[5], g[5], p[4], g[4], p54, g54);
opo opo04(p[4], g[4], p[3], g[3], p43, g43);
opo opo05(p[3], g[3], p[2], g[2], p32, g32);
opo opo06(p[2], g[2], p[1], g[1], p21, g21);
opo opo07(p[1], g[1], p[0], g[0], p10, g10);

// second level of operator o
opo opo08(p76, g76, p54, g54, p74, g74);
opo opo09(p65, g65, p43, g43, p63, g63);
opo opo10(p54, g54, p32, g32, p52, g52);
opo opo11(p43, g43, p21, g21, p41, g41);
opo opo12(p32, g32, p10, g10, p30, g30);
opo opo13(p21, g21, p[0], g[0], p20, g20);

// third level of operator o
opo opo14(p74, g74, p30, g30, p70, g70);
opo opo15(p63, g63, p20, g20, p60, g60);
opo opo16(p52, g52, p10, g10, p50, g50);
opo opo17(p41, g41, p[0], g[0], p40, g40);
```

© 2011 by Gerald E. Sobelman

Verilog Code for the Kogge-Stone Adder: Part 4 of 4

```
// form the carry signals
assign c1 = g[0] | (p[0] & c0);
assign c2 = g10 | (p10 & c0);
assign c3 = g20 | (p20 & c0);
assign c4 = g30 | (p30 & c0);
assign c5 = g40 | (p40 & c0);
assign c6 = g50 | (p50 & c0);
assign c7 = g60 | (p60 & c0);
assign c8 = g70 | (p70 & c0);

// form the sum signals
xor(s[0], psum[0], c0);
xor(s[1], psum[1], c1);
xor(s[2], psum[2], c2);
xor(s[3], psum[3], c3);
xor(s[4], psum[4], c4);
xor(s[5], psum[5], c5);
xor(s[6], psum[6], c6);
xor(s[7], psum[7], c7);

endmodule
```

© 2011 by Gerald E. Sobelman

A Testbench for the Kogge-Stone Adder

```
module tb1ks; // random unsigned inputs, decimal values including a check

reg  [7:0] a, b;  // 8-bit inputs (to be chosen randomly)
reg  c0;          // carry input  (to be chosen randomly)
wire [7:0] s;     // 8-bit sum output
wire c8;          // carry out of the MSB position
reg  [8:0] check; // 9-bit sum value used to check correctess

// instantiate the 8-bit Kogge-Stone adder
kogge_stone ks1(a, b, c0, s, c8);

// simulation of 50 random addition operations
initial repeat (50) begin
    // get new operand values and compute a check value
    a = $random; b = $random; c0 = $random;
    check = a + b + c0;

    // compute and display the sum every 10 time units
    #10 $display($time, "  %d + %d + %d = %d (%d)", a, b, c0, {c8, s}, check);
end

endmodule
```

© 2011 by Gerald E. Sobelman

A Portion of the Testbench Output

```
 10     36 + 129 + 1 = 166 (166)
 20     99 +  13 + 1 = 113 (113)
 30    101 +  18 + 1 = 120 (120)
 40     13 + 118 + 1 = 132 (132)
 50    237 + 140 + 1 = 378 (378)
 60    198 + 197 + 0 = 395 (395)
 70    229 + 119 + 0 = 348 (348)
 80    143 + 242 + 0 = 385 (385)
 90    232 + 197 + 0 = 429 (429)
100    189 +  45 + 1 = 235 (235)
          ...
```

© 2011 by Gerald E. Sobelman

An Exhaustive Testbench for the K-S Adder: Part 1 of 3

```
module tb2ks; // testbench for the 8-bit unsigned Kogge-Stone adder
              // exhaustive checking of all 256*256*2 possible cases

reg  [7:0] a, b;  // 8-bit operands
reg  c0;          // carry input
wire [7:0] s;     // 8-bit sum output
wire c8;          // carry output
reg  [8:0] check; // 9-bit value used to check correctess
integer    i, j, k;     // loop variables
integer    num_correct; // counter to keep track of the number correct
integer    num_wrong;   // counter to keep track of the number wrong

// instantiate the 8-bit Kogge-Stone adder
kogge_stone ks1(a, b, c0, s, c8);

// exhaustive checking
initial begin
    // initialize the counter variables
    num_correct = 0; num_wrong = 0;
```

An Exhaustive Testbench for the K-S Adder: Part 2 of 3

```
// loop through all possible cases and record the results
for (i = 0; i < 256; i = i + 1) begin
    a = i;
    for (j = 0; j < 256; j = j + 1) begin
        b = j;
        for (k = 0; k < 2; k = k + 1) begin
            c0 = k;
            check = a + b + c0;

            // compute and check the product
            #10 if ({c8, s} == check)
               num_correct = num_correct + 1;
            else
               num_wrong = num_wrong + 1;

// following line is for debugging
// $display($time, "  %d + %d + %d = %d (%d)", a, b, c0, {c8, s}, check);

        end
    end
end
```

© 2011 by Gerald E. Sobelman

An Exhaustive Testbench for the K-S Adder: Part 3 of 3

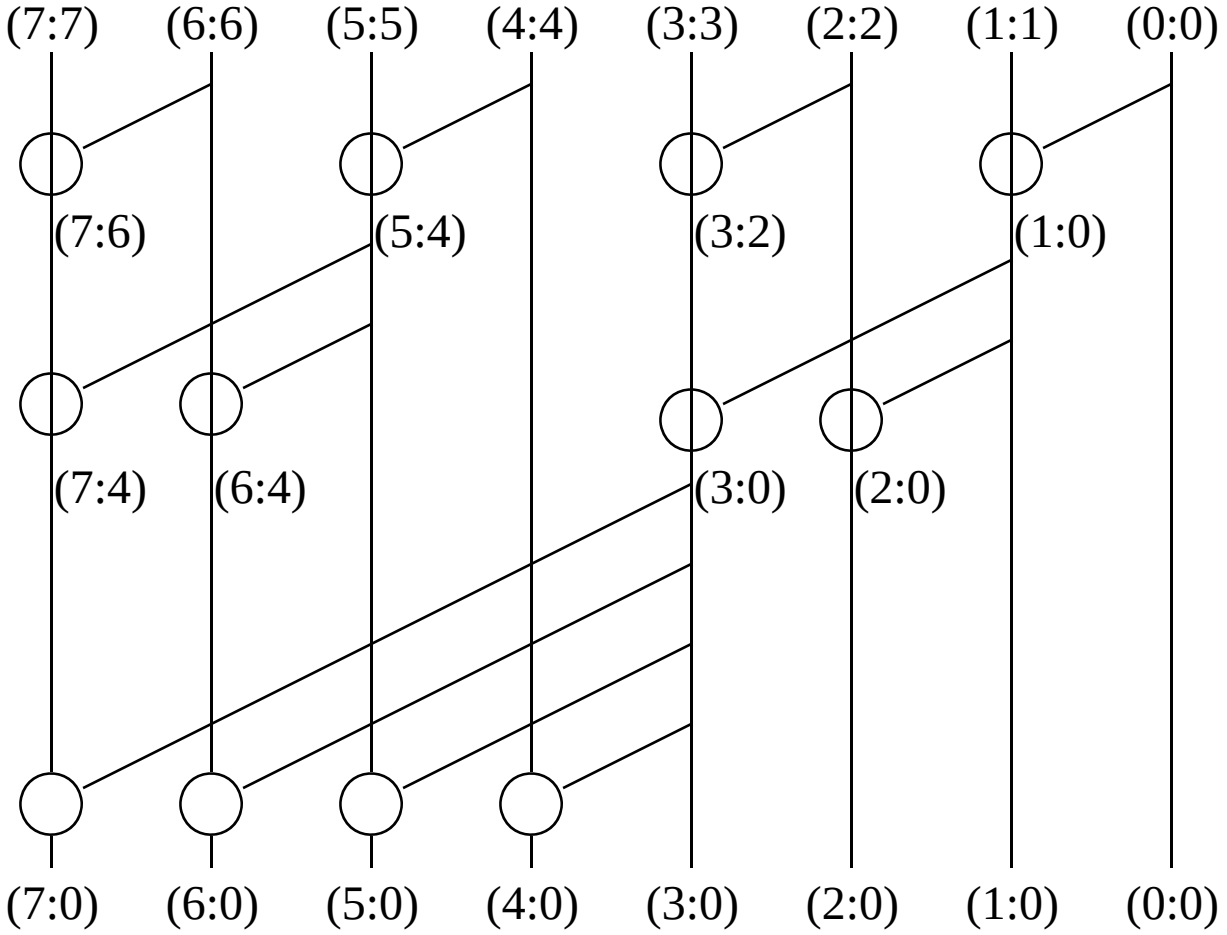
```
    // print the final counter values
    $display("num_correct = %d, num_wrong = %d", num_correct, num_wrong);

end

endmodule
```

- The output produced by this testbench is:

```
num_correct =      131072,        num_wrong =           0
```

Ladner-Fischer (LF) Architecture

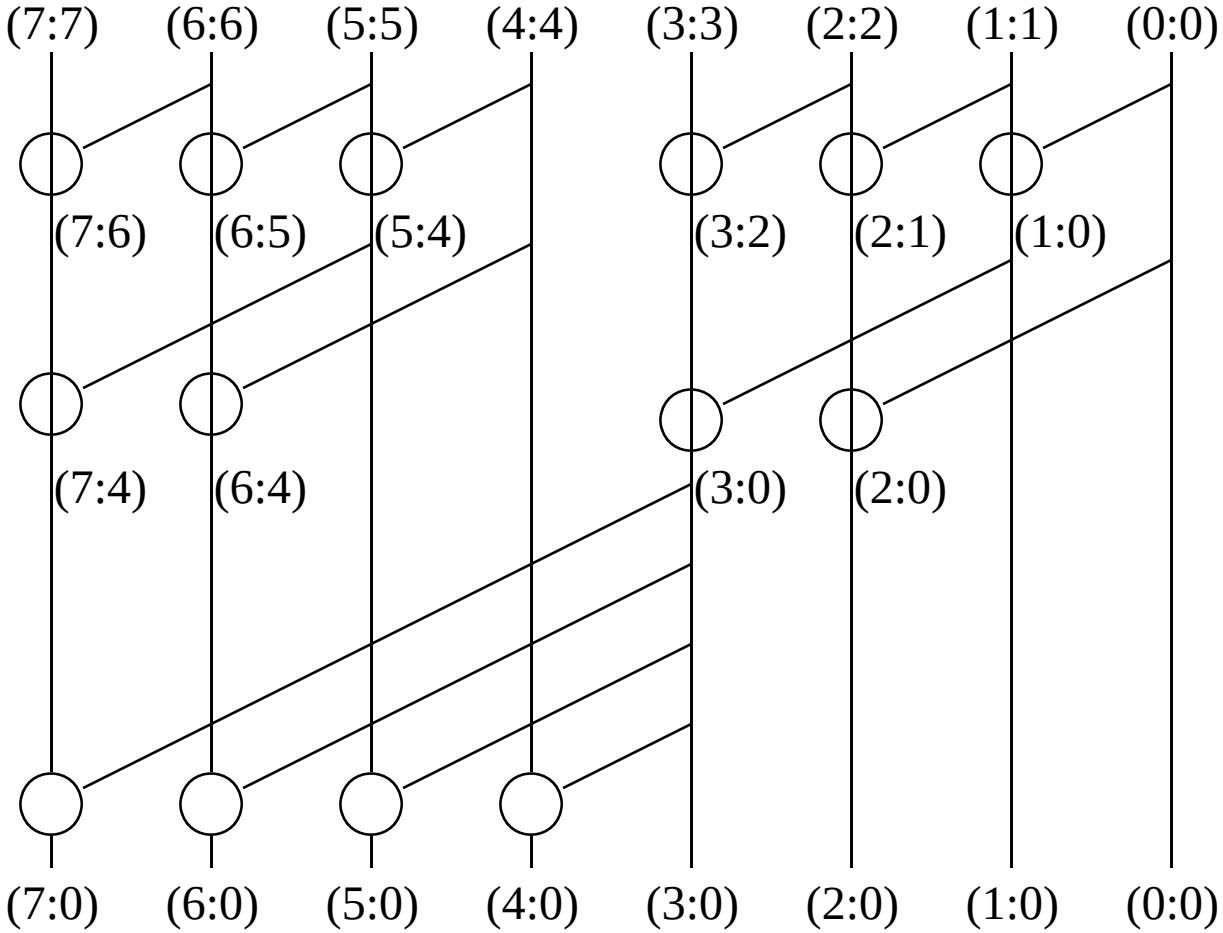
- The n-bit LF design also achieves the minimum logical depth of $\log_2 n$ levels. It uses higher fan-out configurations. The 8-bit LF design is shown below:



© 2011 by Gerald E. Sobelman

Knowles Architecture

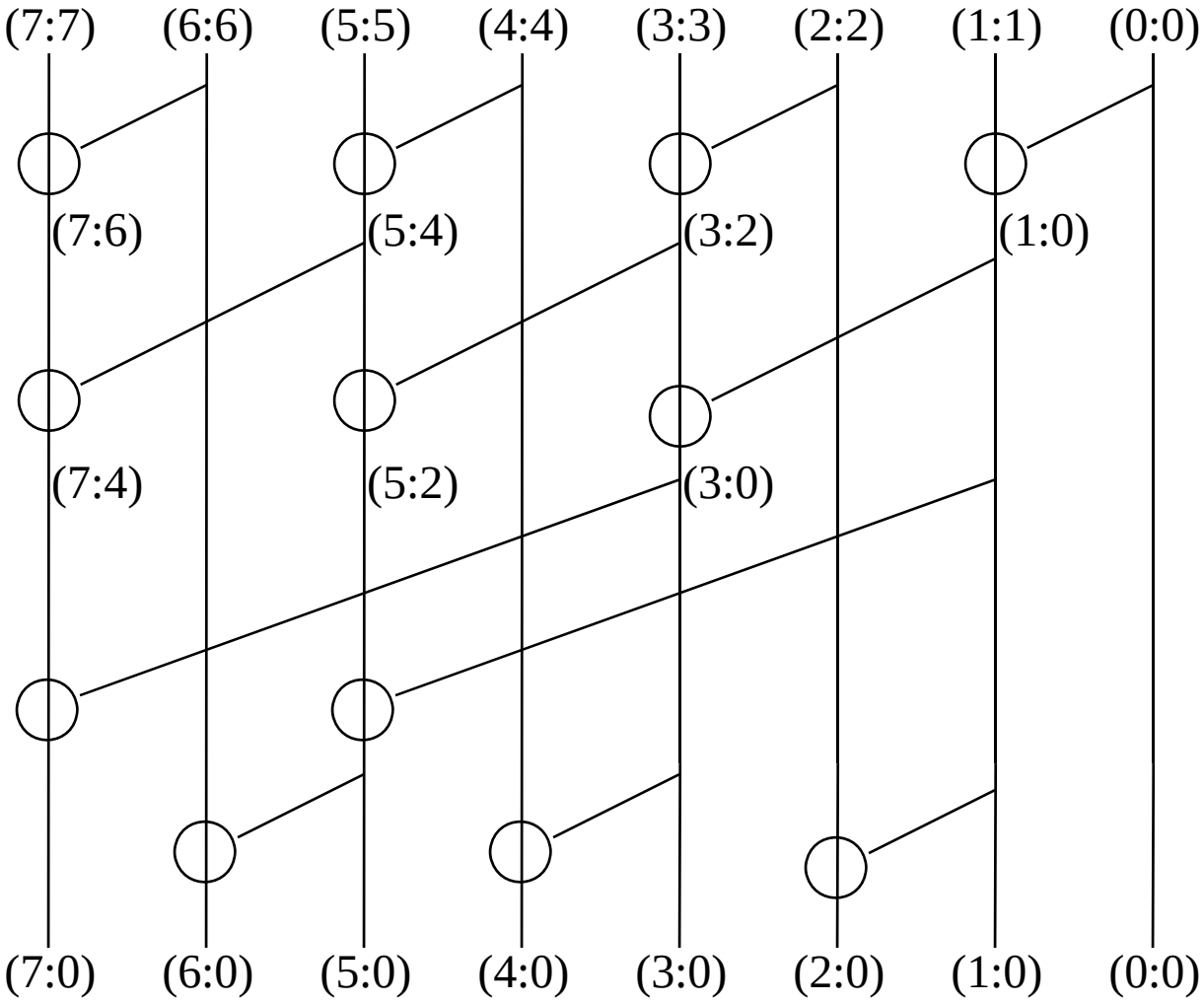
- Knowles has shown that the KS and LF designs are limiting cases of a set of minimum-depth designs. There are 3 other possible 8-bit designs, such as:



© 2011 by Gerald E. Sobelman

Han-Carlson (HC) Architecture

- One additional stage at the end is used, but it has low fanout and few devices.



© 2011 by Gerald E. Sobelman