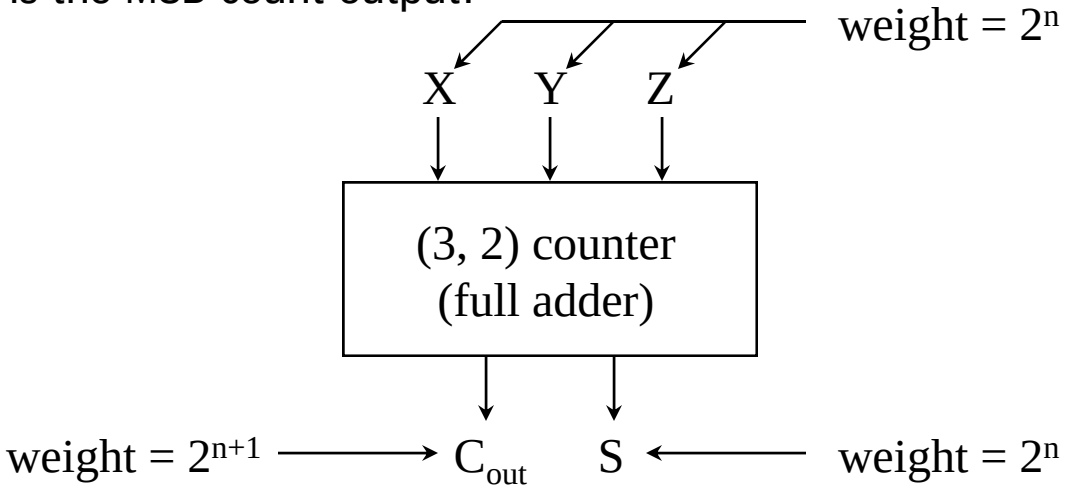
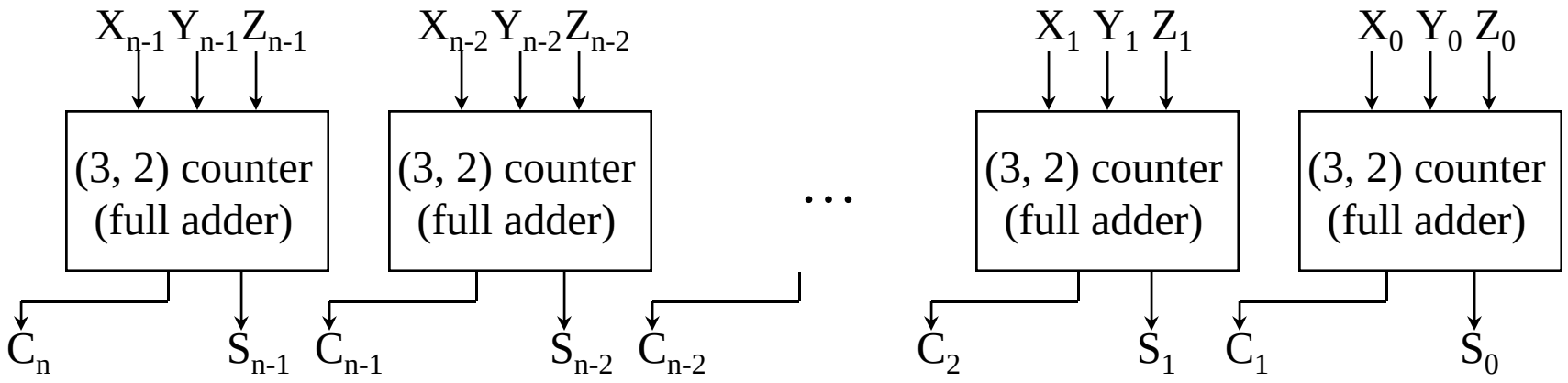
(3, 2) Counter

- An (m, n) counter takes as input m bits (all of the same power-of-2 weight) and produces an n-bit binary number whose value is the number of inputs that are equal to 1. In other words, it *counts* the number of 1s in the input and outputs the binary count value. The outputs of the counter have *different* power-of-2 weights. The weight of the LSB of the counter output is the same as the weight of each of the inputs, and the remaining bits have increasingly higher weights.
- The simplest and most widely used example is the (3, 2) counter. Of the 3 inputs, there can be either 0, 1, 2 or 3 inputs equal to 1. All four of these values can be represented as a 2-bit binary number. In fact, the (3, 2) counter is nothing but a full adder, where the sum is the LSB count output and the carry-out is the MSB count output:



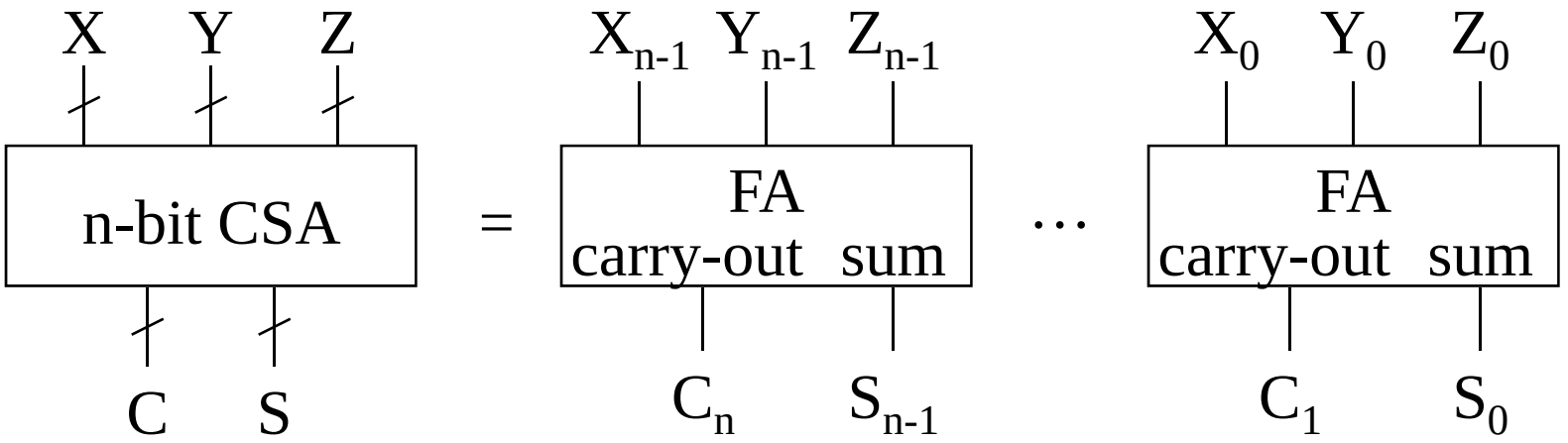
Carry-Save Adder

- In a multiplier, we have to add many partial products together in order to obtain the final product. We could just successively accumulate partial products using a cascade of standard high-speed adders in which we have a carry propagation. We refer to such adders as carry propagate adders (CPAs). However, this would be very slow due to the carry propagation delay in each CPA.
- A much better alternative is to successively reduce 3 input vectors to 2 output vectors, i.e. a sum vector and a carry vector. Each bit of these two vectors are computed independently of all other bits and there is no carry propagation between adjacent bit positions. The hardware does compression from 3 vectors X, Y and Z down to 2 vectors S and C is called a carry-save adder (CSA). It is composed of a parallel set of (3, 2) counters, i.e. a parallel set of full adders.



From CSAs to Wallace Trees

- When there are a large number of vectors to be compressed to 2 final vectors, we need many CSAs. They should be organized in a way that minimizes the delay (i.e., number of levels of CSA) and/or the number of CSAs required.
- An n-bit CSA can be conveniently denoted as follows:

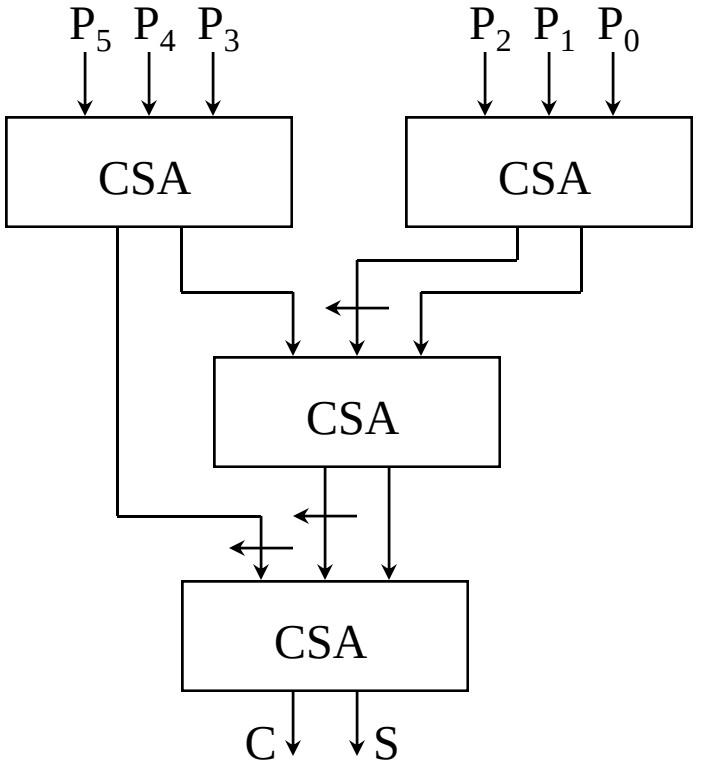


Ref: Vojin G. Oklobdzija, David Villeger and Simon S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers*, Vol. 45, No. 3, pp. 294-306, March, 1996.

Ref: P. Song and G. De Micheli, "Circuit and Architecture Trade-Offs for High Speed Multiplication," *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 9, pp. 1184-1198, Sept., 1991.

Wallace Tree Example

- For example, consider compressing 6 partial products P_0, P_1, ... P_5 to 2 vectors S and C. This can be done using 3 levels of CSAs.
- The left arrow on some CSA inputs means that that vector is shifted left by one bit position to account for the fact that it is a carry vector output of a prior CSA.
- This technique can be readily extended to a larger number of initial vectors. For example, we can compress 9 vectors to 2 using 4 levels of CSAs:
  level 1: 9 = 3 + 3 + 3 => 2 + 2 + 2 = 6
  level 2: 6 = 3 + 3 => 2 + 2 = 4
  level 3: 4 = 3 + 1 => 2 + 1 = 3
  level 4: 3 => 2



Wallace Tree Verilog Code: Part 1 of 2

```
module full_adder(a, b, cin, s, cout);

input  a, b, cin;
output s, cout;

assign s = a^b^cin;
assign cout = (a&b) | (b&cin) | (a&cin);

endmodule

// 8-bit carry-save adder
module csa(x, y, z, s, c);

input  [7:0] x, y, z;
output [7:0] s;
output [8:1] c;

full_adder fa0(x[0], y[0], z[0], s[0], c[1]);
full_adder fa1(x[1], y[1], z[1], s[1], c[2]);
full_adder fa2(x[2], y[2], z[2], s[2], c[3]);
full_adder fa3(x[3], y[3], z[3], s[3], c[4]);
full_adder fa4(x[4], y[4], z[4], s[4], c[5]);
full_adder fa5(x[5], y[5], z[5], s[5], c[6]);
full_adder fa6(x[6], y[6], z[6], s[6], c[7]);
full_adder fa7(x[7], y[7], z[7], s[7], c[8]);

endmodule
```

Wallace Tree Verilog Code: Part 2 of 2

```
// 6-input Wallace tree
module wallace(p0, p1, p2, p3, p4, p5, s, c);

input  [7:0] p0, p1, p2, p3, p4, p5;
output [8:0] s;
output [9:1] c;

wire   [7:0] s1, s2, s3, s4;
wire   [8:1] c1, c2, c3, c4;

csa csa1(p2, p1, p0, s1, c1);
csa csa2(p5, p4, p3, s2, c2);
csa csa3(s2, {c1[7:1], 1'b0}, s1, s3, c3);
csa csa4({c2[7:1], 1'b0}, {c3[7:1], 1'b0}, s3, s4, c4);
full_adder fa1(c1[8], c2[8], c3[8], s_msb, c_msb);
assign s = {s_msb, s4};
assign c = {c_msb, c4};

endmodule
```


Wallace Tree Testbench Code

```
module tb1wallace; // random unsigned inputs, decimal values including a check

reg  [7:0] p0, p1, p2, p3, p4, p5;  // 8-bit inputs (to be chosen randomly)
wire [8:0] s;                       // 9-bit sum output
wire [9:1] c;                       // 9-bit carry output
reg  [10:0] sval, check;            // 11-bit final sum and check values

// instantiate the 6-input Wallace tree
wallace wallace1(p0, p1, p2, p3, p4, p5, s, c);

// simulation of 50 random addition operations
initial repeat (50) begin
    // get new operand values and compute a check value
    p0 = $random; p1 = $random; p2 = $random;
    p3 = $random; p4 = $random; p5 = $random;
    check = p0 + p1 + p2 + p3 + p4 + p5;

    // compute and display the final sum value every 10 time units
    #10 sval = s + (2*c);
    $display($time, "  %d + %d + %d + %d + %d + %d = %d (%d)",
             p0, p1, p2, p3, p4, p5, sval, check);
end

endmodule
```

Wallace Tree Testbench Results

- A portion of the output produced is as follows:

```
 10   36 + 129 +   9 +  99 +  13 + 141 =  427 ( 427)
 20  101 +  18 +   1 +  13 + 118 +  61 =  312 ( 312)
 30  237 + 140 + 249 + 198 + 197 + 170 = 1191 (1191)
 40  229 + 119 +  18 + 143 + 242 + 206 =  957 ( 957)
 50  232 + 197 +  92 + 189 +  45 + 101 =  856 ( 856)
 60   99 +  10 + 128 +  32 + 170 + 157 =  596 ( 596)
 70  150 +  19 +  13 +  83 + 107 + 213 =  585 ( 585)
 80    2 + 174 +  29 + 207 +  35 +  10 =  457 ( 457)
 90  202 +  60 + 242 + 138 +  65 + 216 =  923 ( 923)
100  120 + 137 + 235 + 182 + 198 + 174 = 1046 (1046)

                          ...
```