Baugh-Wooley Multiplier Design

- To illustrate the mathematical transformation which is required, consider 4-bit signed operands X and Y and an 8-bit product P:

$$X \Leftrightarrow (x_3, x_2, x_1, x_0) \quad , \quad Y \Leftrightarrow (y_3, y_2, y_1, y_0) \quad , \quad P \Leftrightarrow (p_7, p_6, p_5, p_4, p_3, p_2, p_1, p_0)$$

- Because these are in two's complement form, we can compute their numerical values as:

$$X = -x_3 2^3 + \sum_{i=0}^{2} x_i 2^i \qquad Y = -y_3 2^3 + \sum_{j=0}^{2} y_j 2^j \qquad P = -p_7 2^7 + \sum_{i=0}^{6} p_i 2^i$$

- Also, since P = XY, we can write:

$$\begin{aligned} P &= (-x_3 2^3 + \sum_{i=0}^{2} x_i 2^i)(-y_3 2^3 + \sum_{j=0}^{2} y_j 2^j) \\ &= x_3 y_3 2^6 + \sum_{i=0}^{2} \sum_{j=0}^{2} x_i y_j 2^{i+j} - \sum_{i=0}^{2} x_i y_3 2^{i+3} - \sum_{j=0}^{2} x_3 y_j 2^{j+3} \end{aligned}$$

- Note that the first two terms are positive summands while the second two terms are negative summands. However, instead of subtracting, we can add the two's complements of those two terms.

Baugh-Wooley Multiplier - Cont.

- Consider the summation in the 3rd term. We can add 2 zero terms that don't change it:

$$\sum_{i=0}^{2} x_i y_3 2^{i+3} = 2^3[-0 \cdot 2^4 + 0 \cdot 2^3 + \sum_{i=0}^{2} x_i y_3 2^i]$$

- Instead of subtracting this, we can add its two's complement, which can be computed as the one's complement plus 1:

$$2^3[-1 \cdot 2^4 + 1 \cdot 2^3 + \sum_{i=0}^{2} \overline{x_i y_3} 2^i + 1]$$

- There are two possible cases:
  - If $y_3 = 0$, this simplifies to:

$$2^3[-2^3 + \sum_{i=0}^{2} 2^i + 1] = 2^3[-2^3 + (2^3 - 1) + 1] = 0$$

  - If $y_3 = 1$, this simplifies to:

$$2^3[-2^3 + \sum_{i=0}^{2} \overline{x}_i 2^i + 1]$$

Ref: K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley, 1979.

Baugh-Wooley Multiplier Design – Cont.

- These two sub-cases can be subsumed into the following single expression:

$$2^3[-2^3 + y_3 + \overline{y}_3 2^3 + \sum_{i=0}^{2} \overline{x}_i y_3 2^i]$$

- We can check this as follows:
  - For $y_3 = 0$, this reduces to: $2^3[-2^3 + 2^3] = 0$ ✓
  - For $y_3 = 1$, this reduces to: $2^3[-2^3 + 1 + \sum_{i=0}^{2} \overline{x}_i 2^i]$ ✓

- By symmetry (i.e., by reversing the roles of the x and y terms), we can immediately write down the corresponding expression for the two's complement of the 4th term as:

$$2^3[-2^3 + x_3 + \overline{x}_3 2^3 + \sum_{j=0}^{2} x_3 \overline{y}_j 2^j]$$

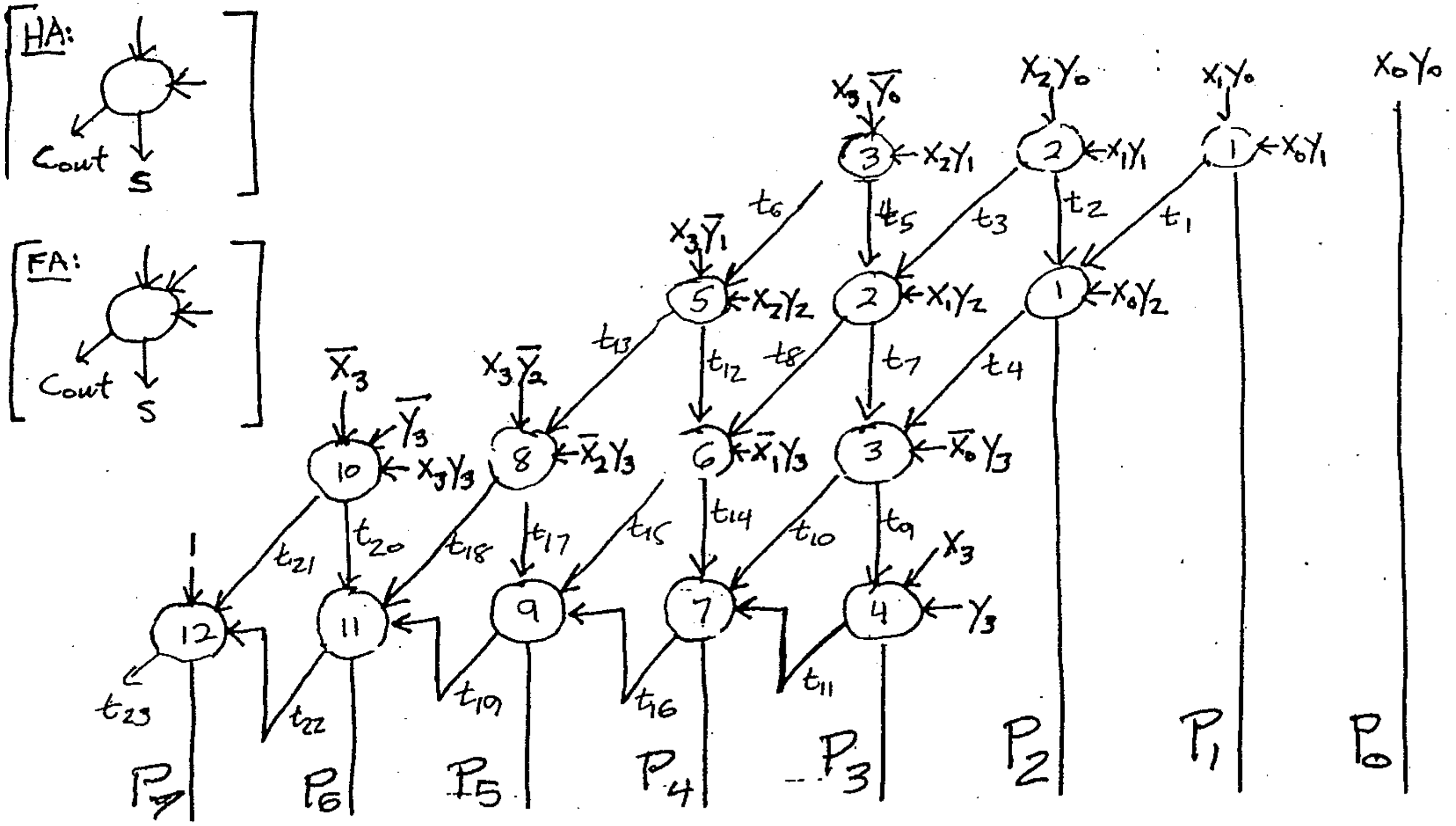
Baugh-Wooley Multiplier Design – Cont.

- Finally, replacing the negative summands by the addition of these two's complement forms in our original expression for P gives:

$$P = x_3y_32^6 + \sum_{i=0}^{2}\sum_{j=0}^{2}x_iy_j2^{i+j}$$
$$-2^6 + y_32^3 + \overline{y}_32^6 + \sum_{i=0}^{2}\overline{x}_iy_32^{i+3}$$
$$-2^6 + x_32^3 + \overline{x}_32^6 + \sum_{j=0}^{2}x_3\overline{y}_j2^{j+3}$$

- Note that we can further simplify $-2^6 - 2^6 = -2^7$, which, in turn, corresponds to a +1 in the 2^7 position because it is the MSB of an 8-bit two's complement vector.
- As a result, the above set of terms corresponds to a set of positive terms to be added using a set of half-adders and full-adders.
- The above mathematical transformations can be extended to an arbitrary operand sizes, such as 16-bit by 16-bit multiplication or 32-bit by 32-bit multiplication, etc.

Baugh-Wooley Multiplier Design – Cont.



Verilog Code for the Baugh-Wooly Multiplier: Part 1 of 3

```
// half adder component used in the multiplier
module half_adder(a, b, s, cout);

input  a, b;
output s, cout;

assign s = a^b;
assign cout = a&b;

endmodule

// full adder component used in the multiplier
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
```

Verilog Code for the Baugh-Wooly Multiplier: Part 2 of 3

```
// 4-bit by 4-bit Baugh-Wooley signed multiplier
module mult4bw(x, y, p);

input  [3:0] x, y;
output [7:0] p;

// constant logic-one value
supply1 one;

// internal nodes within the multiplier circuit
wire   t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12,
       t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23;
```

Verilog Code for the Baugh-Wooly Multiplier: Part 3 of 3

```
// structural description of the multiplier circuit
assign p[0] = x[0]&y[0];
half_adder ha1(x[1]&y[0], x[0]&y[1], p[1], t1);
half_adder ha2(x[2]&y[0], x[1]&y[1], t2, t3);
full_adder fa1(t2, t1, x[0]&y[2], p[2], t4);
half_adder ha3(x[3]&~y[0], x[2]&y[1], t5, t6);
full_adder fa2(t5, t3, x[1]&y[2], t7, t8);
full_adder fa3(t7, t4, ~x[0]&y[3], t9, t10);
full_adder fa4(t9, x[3], y[3], p[3], t11);
full_adder fa5(x[3]&~y[1], t6, x[2]&y[2], t12, t13);
full_adder fa6(t12, t8, ~x[1]&y[3], t14, t15);
full_adder fa7(t14, t10, t11, p[4], t16);
full_adder fa8(x[3]&~y[2], t13, ~x[2]&y[3], t17, t18);
full_adder fa9(t17, t15, t16, p[5], t19);
full_adder fa10(~x[3], ~y[3], x[3]&y[3], t20, t21);
full_adder fa11(t20, t18, t19, p[6], t22);
full_adder fa12(one, t21, t22, p[7], t23);

endmodule
```

An Exhaustive Testbench for the B-W Multiplier: Part 1 of 3

```
module tb9; // testbench for the 4-bit by 4-bit Baugh-Wooley signed multiplier
            // exhaustive checking of all 256 possible cases

reg  [3:0] x, y;        // 4-bit inputs (to be chosen randomly)
integer    xval, yval;  // numerical values of inputs x and y
wire [7:0] p;           // 8-bit output of the multiplier circuit
integer    pval;        // numerical value of the product
integer    check;       // value used to check correctness
integer    i, j;        // 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 4-bit by 4-bit Baugh-Wooley signed multiplier
mult4bw mult_instance(x, y, p);

// exhaustive simulation of all 256 possible cases
initial begin
    // initialize the counter variables
    num_correct = 0; num_wrong = 0;
```

An Exhaustive Testbench for the B-W Multiplier: Part 2 of 3

```
// loop through all possible cases and record the results
 for (i = 0; i < 16; i = i + 1) begin
x = i;
xval = -x[3]*8 + x[2:0];
for (j = 0; j < 16; j = j + 1) begin
        y = j;
    yval = -y[3]*8 + y[2:0];
        check = xval * yval;

          // compute and check the product
        #10 pval = -p[7]*128 + p[6:0];
        if (pval == check)
        num_correct = num_correct + 1;
        else
        num_wrong = num_wrong + 1;

        // following line is commented out, but is useful for debugging
        // $display($time, "  %d * %d = %d (%d)", xval, yval, pval, check);
    end
 end
```

An Exhaustive Testbench for the B-W Multiplier: 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 =         256, num_wrong =          0
```