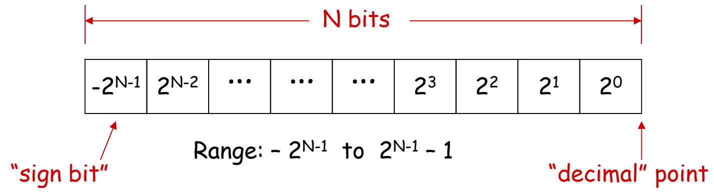
Arithmetic Circuits & Multipliers

- Addition, subtraction
- Performance issues
  - -- ripple carry
  - -- carry bypass
  - -- carry skip
  - -- carry lookahead
- Combinational multiplier
- Two's complement multiplier
- Smaller multipliers, faster multipliers

Signed integers: 2's complement


N bits
-2^{N-1} 2^{N-2} ••• ••• ••• 2^3 2^2 2^1 2^0
"sign bit"
Range: -2^{N-1} to $2^{N-1}-1$
"decimal" point

8-bit 2's complement example:

11010110 = $-2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

By moving the implicit location of "decimal" point, we can represent fractions too:

1101.0110 = $-2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

Sign extension

Consider the 8-bit 2's complement representation of:

```
42 = 00101010      -5 = ~00000101 + 1
                      =  11111010 + 1
                      =  11111011
```

What is their 16-bit 2's complement representation?

```
42 = 0000000000101010

-5 = 1111111111111011
```

$$-5 = \underbrace{11111111}\,11111011$$

Extend the MSB (aka the "sign bit") into the higher-order bit positions

Using Signed Arithmetic in Verilog

Use care with signed arithmetic!

```
wire signed [7:0] total;
wire [3:0] counter; // max value 15, counting widgets off the mfg line
wire signed [5:0] available;

assign total = available + counter; // does this give the correct answer?
```

NO! counter = 4'b1111 is treated as -1. Need to "append" a leading zero

```
assign total = available + {1'b0, counter};  // or use $unsigned()
assign total = available + $unsigned(counter);
```

Adder: a circuit that does addition

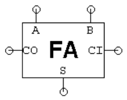
Here's an example of binary addition as one might do it by "hand":

Carries from previous column: 1 1 0 1

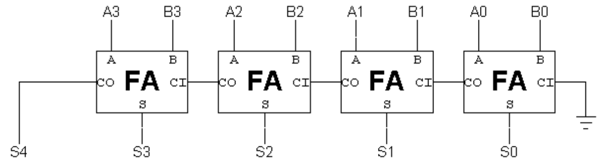
```
   1101
+  0101
  10010
```

Adding two N-bit numbers produces an (N+1)-bit result

If we build a circuit that implements one column:

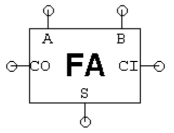


we can quickly build a circuit to add two 4-bit numbers…



"Ripple-carry adder"

"Full Adder" building block



A	B	C	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The "half adder" circuit has only the A and B inputs

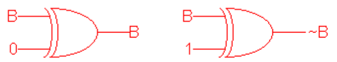
$$S = A \oplus B \oplus C$$

$$\begin{aligned} CO &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \\ &= (\overline{A} + A)BC + (\overline{B} + B)AC + AB(\overline{C} + C) \\ &= BC + AC + AB \end{aligned}$$

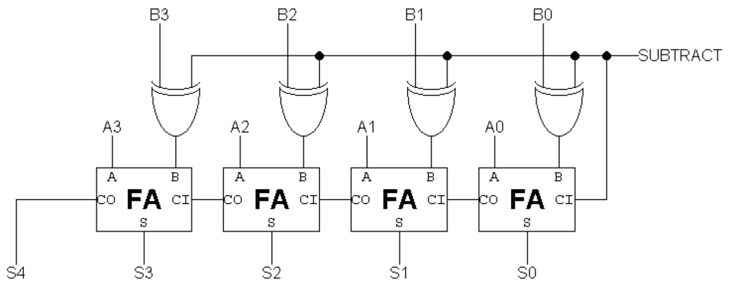
Subtraction: A-B = A + (-B)

Using 2's complement representation: –B = ~B + 1

~ = bit-wise complement



So let's build an arithmetic unit that does both addition and subtraction. Operation selected by *control input*:



Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001 *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

$$V = A_{N-1}B_{N-1}\overline{S_{N-1}} + \overline{A_{N-1}}\,\overline{B_{N-1}}S_{N-1}$$

$$V = COUT_{N-1} \oplus CIN_{N-1}$$

To compare A and B, perform A–B and use condition codes:

Signed comparison:

LT	N⊕V
LE	Z+(N⊕V)
EQ	Z
NE	~Z
GE	~(N⊕V)
GT	~(Z+(N⊕V))

Unsigned comparison:

LTU	C
LEU	C+Z
GEU	~C
GTU	~(C+Z)

Condition Codes in Verilog

Z (zero): result is = 0

N (negative): result is < 0

C (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

```
wire signed [31:0] a,b,s;
wire z,n,v,c;
assign {c,s} = a + b;
assign z = ~|s;
assign n = s[31];
assign v = a[31]^b[31]^s[31]^c;
```

Might be better to use sum-of-products formula for V from previous slide if using LUT implementation (only 3 variables instead of 4).

Modular Arithmetic

The Verilog arithmetic operators (+,-,*) all produce full-precision results, e.g., adding two 8-bit numbers produces a 9-bit result.

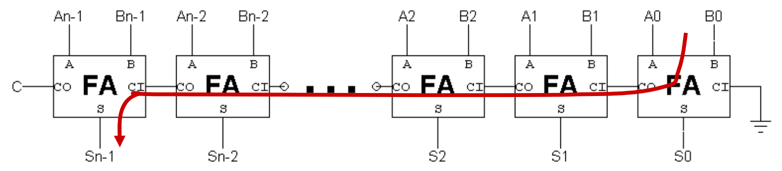
In many designs one chooses a "word size" (many computers use 32 or 64 bits) and all arithmetic results are truncated to that number of bits, i.e., arithmetic is performed modulo $2^{\text{word size}}$.

Using a fixed word size can lead to *overflow*, e.g., when the operation produces a result that's too large to fit in the word size. One can

- Avoid overflow: choose a sufficiently large word size
- Detect overflow: have the hardware remember if an operation produced an overflow – trap or check status at end
- Embrace overflow: sometimes this is exactly what you want, e.g., when doing index arithmetic for circular buffers of size 2^N.
- "Correct" overflow: replace result with most positive or most negative number as appropriate, aka *saturating arithmetic*. Good for digital signal processing.

Speed: t_{PD} of Ripple-carry Adder

$C_O = AB + AC_I + BC_I$



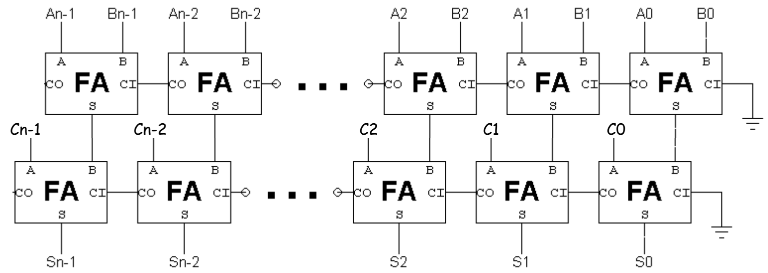
Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = \underbrace{(N-1)*(t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{\text{CI}_{N-1}\text{ to S}_{N-1}} \approx \Theta(N)$$

$$\boxed{t_{adder} = (N-1)t_{carry} + t_{sum}}$$

Θ(N) is read "order N" : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

How about the t_{PD} of this circuit?



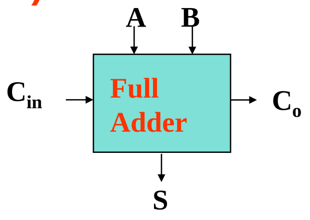
Is the t_{PD} of this circuit = 2 * $t_{PD,\text{N-BIT RIPPLE}}$?

Nope! t_{PD} of this circuit = $t_{PD,\text{N-BIT RIPPLE}} + t_{PD,FA}$!!!

Alternate Adder Logic Formulation

How to Speed up the Critical (Carry) Path?
(How to Build a Fast Adder?)

A	*B*	*C_i*	*S*	*C_o*	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate



Generate (G) = AB

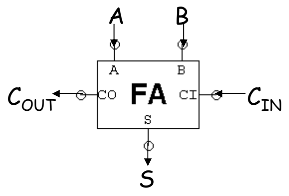
Propagate (P) = A ⊕ B

$$\boxed{\begin{aligned} C_o(G,P) &= G + PC_i \\ S(G,P) &= P \oplus C_i \end{aligned}}$$

Note: can also use P = A + B for C_o

Faster carry logic

Let's see if we can improve the speed by rewriting the equations for C_{OUT}:



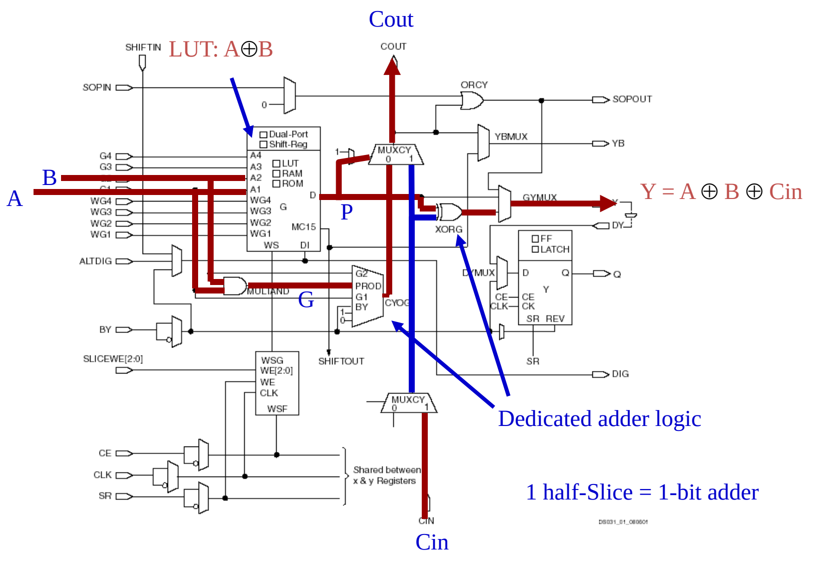
$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$
$$= AB + (A + B)C_{IN}$$
$$= \underbrace{G}_{\text{generate}} + \underbrace{P}_{\text{propagate}}\, C_{IN}$$

where G = AB
P = A + B

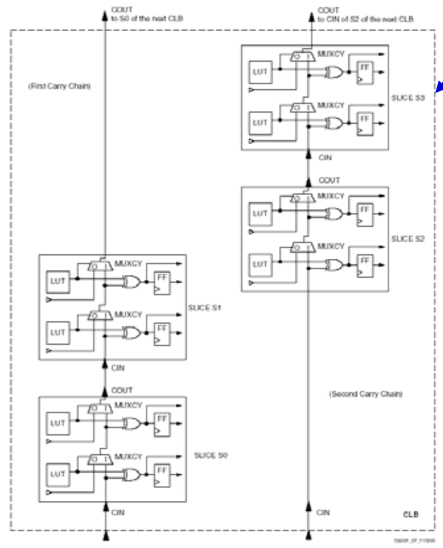
```
module fa(input a,b,cin, output s,cout);
  wire g = a & b;
  wire p = a ^ b;
  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Actually, P is usually defined as P = A^B which won't change C_{OUT} but will allow us to express S as a simple function : $S = P\hat{\ }C_{IN}$

Virtex II Adder Implementation

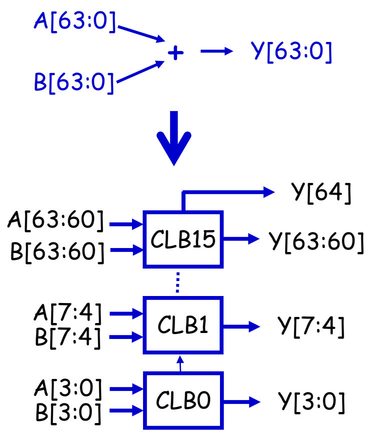


Virtex II Carry Chain



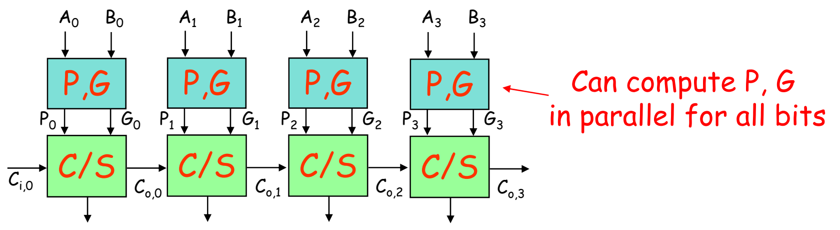
1 CLB = 4 Slices = 2, 4-bit adders

64-bit Adder: 16 CLBs

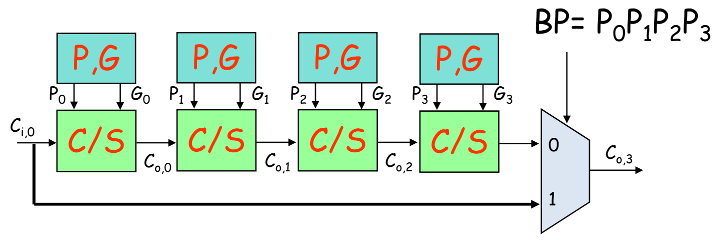


CLBs must be in same column

Carry Bypass Adder



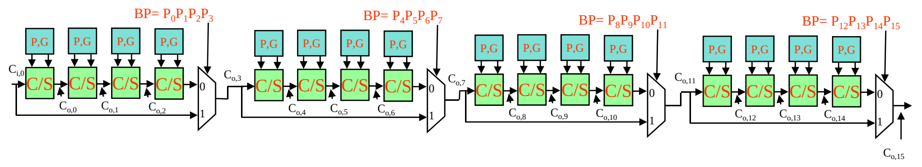
A_0 B_0 A_1 B_1 A_2 B_2 A_3 B_3
P,G
P_0 G_0 P_1 G_1 P_2 G_2 P_3 G_3
C/S
$C_{i,0}$ $C_{o,0}$ $C_{o,1}$ $C_{o,2}$ $C_{o,3}$
Can compute P, G in parallel for all bits



BP= $P_0P_1P_2P_3$
P,G
P_0 G_0 P_1 G_1 P_2 G_2 P_3 G_3
C/S
$C_{i,0}$ $C_{o,0}$ $C_{o,1}$ $C_{o,2}$ $C_{o,3}$
0
1

Key Idea: if $(P_0\, P_1\, P_2\, P_3)$ then $C_{o,3} = C_{i,0}$

16-bit Carry Bypass Adder

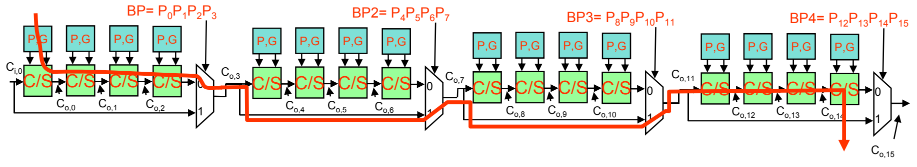


BP= $P_0P_1P_2P_3$
BP= $P_4P_5P_6P_7$
BP= $P_8P_9P_{10}P_{11}$
BP= $P_{12}P_{13}P_{14}P_{15}$
P,G
C/S
$C_{i,0}$ $C_{o,0}$ $C_{o,1}$ $C_{o,2}$ $C_{o,3}$ $C_{o,4}$ $C_{o,5}$ $C_{o,6}$ $C_{o,7}$ $C_{o,8}$ $C_{o,9}$ $C_{o,10}$ $C_{o,11}$ $C_{o,12}$ $C_{o,13}$ $C_{o,14}$ $C_{o,15}$

What is the worst case propagation delay for the 16-bit adder?

Assume the following for delay each gate:
P, G from A, B: 1 delay unit
P, G, C_i to C_o or Sum for a C/S: 1 delay unit
2:1 mux delay: 1 delay unit

Critical Path Analysis



BP= $P_0P_1P_2P_3$
BP2= $P_4P_5P_6P_7$
BP3= $P_8P_9P_{10}P_{11}$
BP4= $P_{12}P_{13}P_{14}P_{15}$
P,G
C/S
$C_{i,0}$ $C_{o,0}$ $C_{o,1}$ $C_{o,2}$ $C_{o,3}$ $C_{o,4}$ $C_{o,5}$ $C_{o,6}$ $C_{o,7}$ $C_{o,8}$ $C_{o,9}$ $C_{o,10}$ $C_{o,11}$ $C_{o,12}$ $C_{o,13}$ $C_{o,14}$ $C_{o,15}$

For the second stage, is the critical path:

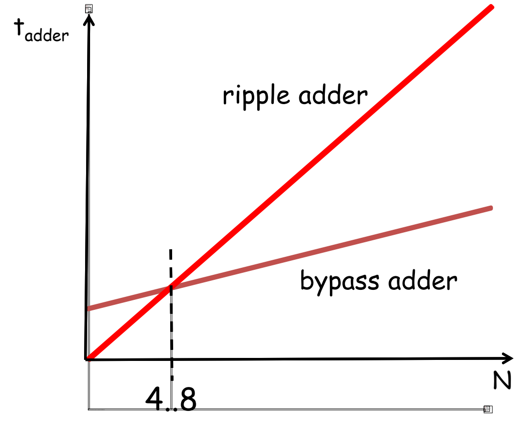
BP2 = 0 or BP2 = 1 ?

Message: Timing analysis is very tricky – Must carefully consider data dependencies for ***false paths***

Carry Bypass vs Ripple Carry

Ripple Carry: $t_{adder} = (N-1)\, t_{carry} + t_{sum}$

Carry Bypass: $t_{adder} = 2(M-1)\, t_{carry} + t_{sum} + (N/M-1)\, t_{bypass}$



M = bypass word size

N = number of bits being added

Carry Lookahead Adder (CLA)

- Recall that $C_{OUT} = G + P\,C_{IN}$ where G = A&B and P = A^B
- For adding two N-bit numbers:

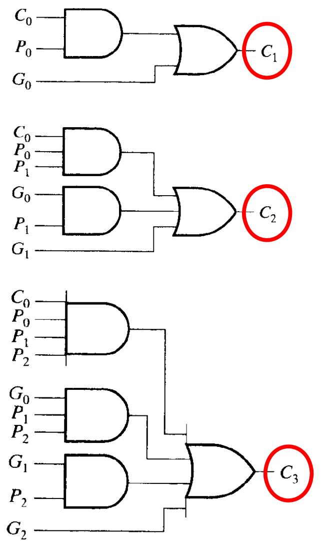
$$\begin{aligned} C_N &= G_{N-1} + P_{N-1}C_{N-1} \\ &= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}C_{N-2} \\ &= \underbrace{G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0C_{IN}} \end{aligned}$$

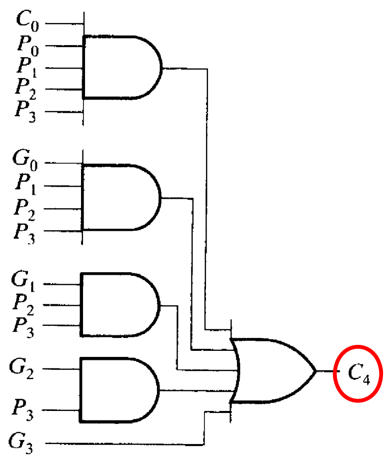
C_N in only 3 gate delays* :
1 for P/G generation, 1 for ANDs, 1 for final OR

*assuming gates with N inputs

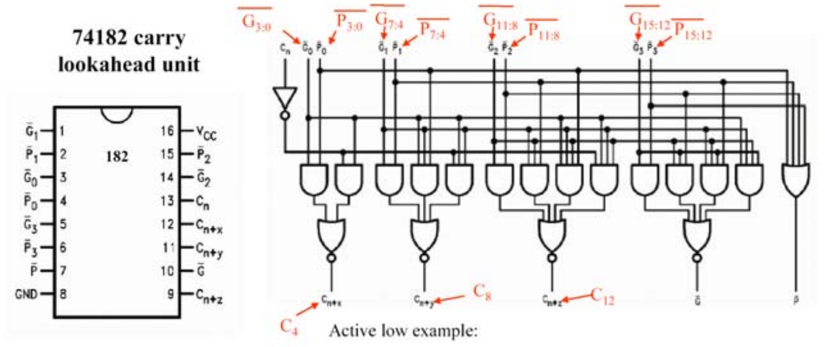
- Idea: pre-compute all carry bits as f(Gs,Ps,C_{IN})

Carry Lookahead Circuits





The 74182 Carry Lookahead Unit



Active low example:

$$\begin{aligned} C_{n+x} &= \overline{\overline{\overline{G0}\cdot\overline{P0}} + \overline{\overline{G0}\cdot\overline{C_n}}} \\ &= \overline{\overline{\overline{G0}\cdot\overline{P0}}}\cdot\overline{\overline{\overline{G0}\cdot\overline{C_n}}} \\ &= (G0+P0)\cdot(G0+C_n) = G0 + P0C_n \end{aligned}$$

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

$C_{n+y} = C_8 = G_{7:4} + P_{7:4}G_{3:0} + P_{7:4}P_{3:0}C_{i,0} = G_{7:0} + P_{7:0}C_n$

$C_{n+z} = C_{12} = G_{11:8} + P_{11:8}G_{7:4} + P_{11:8}P_{7:4}G_{3:0} + P_{11:8}P_{7:4}P_{3:0}C_n$
$= G_{11:0} + P_{11:0}C_n$

- high speed carry lookahead generator
- used with 74181 to extend carry lookahead beyond 4 bits
- correctly handles the carry polarity of the 181

Block Generate and Propagate

G and P can be computed for groups of bits (instead of just for individual bits). This allows us to choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

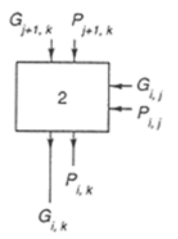
$C_{J+1} = G_{IJ} + P_{IJ}C_I$

$G_{IK} = G_{J+1,K} + P_{J+1,K}\,G_{IJ}$

$P_{IK} = P_{IJ}\,P_{J+1,K}$

"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"

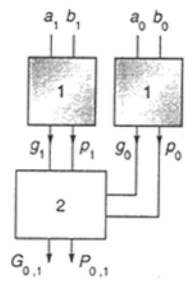
where I < J and J+1 < K



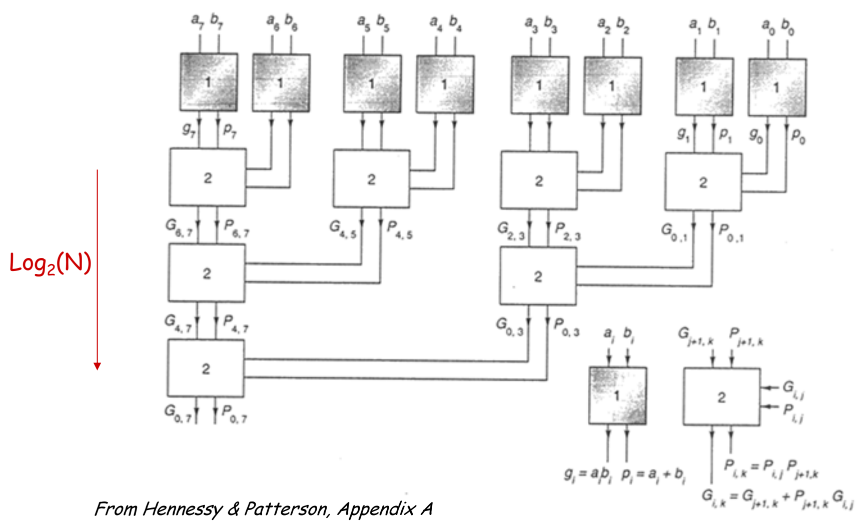
Hierarchical building block

P/G generation

1st level of lookahead

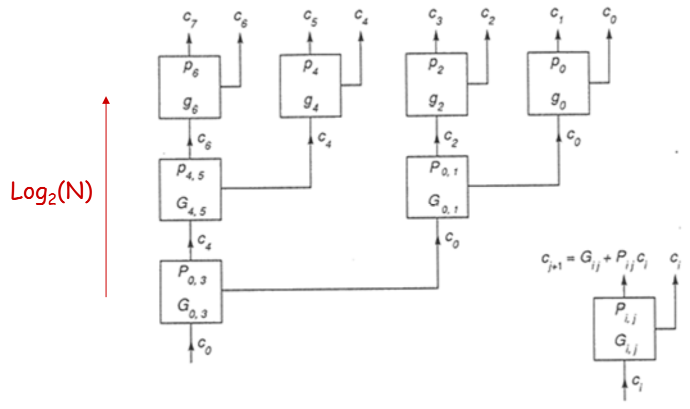


8-bit CLA (P/G generation)

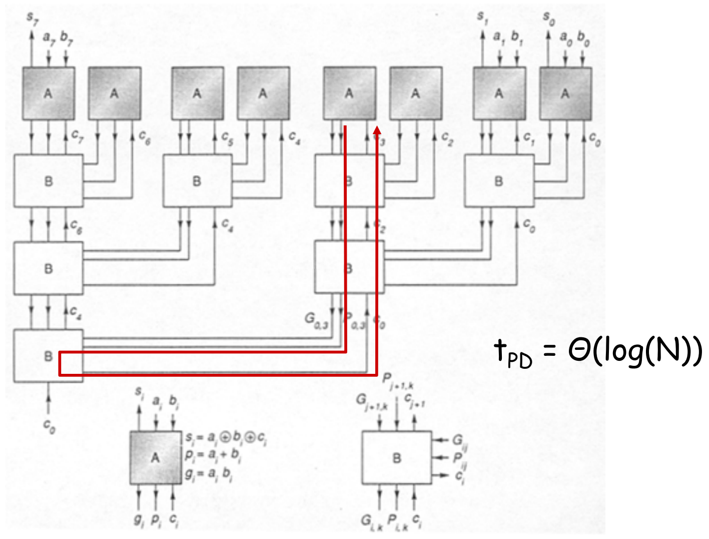

Log2(N)
$g_i = a_ib_i$ $p_i = a_i + b_i$
$P_{i,k} = P_{i,j}P_{j+1,k}$
$G_{i,k} = G_{j+1,k} + P_{j+1,k}G_{i,j}$

From Hennessy & Patterson, Appendix A

8-bit CLA (carry generation)



8-bit CLA (complete)



Unsigned Multiplication

$$
\begin{array}{rcccccccc}
 & & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$

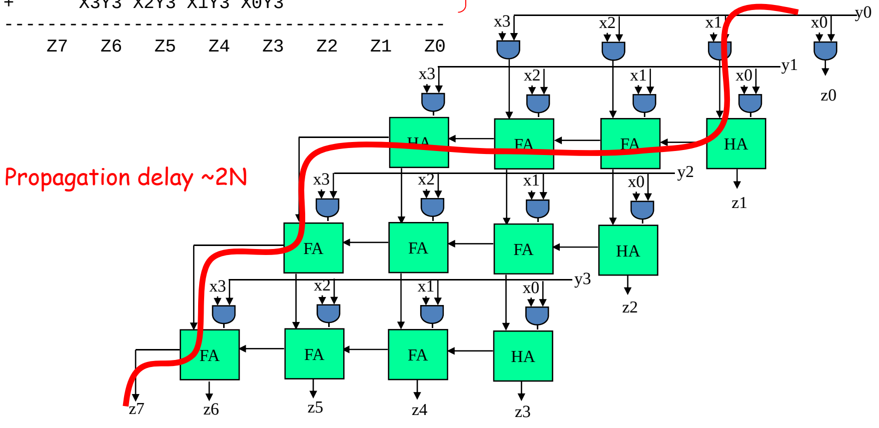
AB_i called a "partial product" $\rightarrow$

Multiplying N-bit number by M-bit number gives (N+M)-bit result

Easy part: forming partial products
(just an AND gate since B_I is either 0 or 1)
Hard part: adding M N-bit partial products

Combinational Multiplier (unsigned)

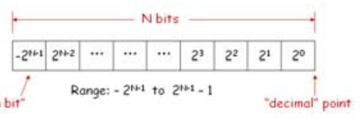
```
                       X3   X2   X1   X0   ← multiplicand
                    *  Y3   Y2   Y1   Y0   ← multiplier
                    --------------------
                     X3Y0 X2Y0 X1Y0 X0Y0  }
  +             X3Y1 X2Y1 X1Y1 X0Y1       }  Partial products, one for each bit in
  +        X3Y2 X2Y2 X1Y2 X0Y2            }  multiplier (each bit needs just one
  +   X3Y3 X2Y3 X1Y3 X0Y3                 }  AND gate)
  ---------------------------------------
     Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```



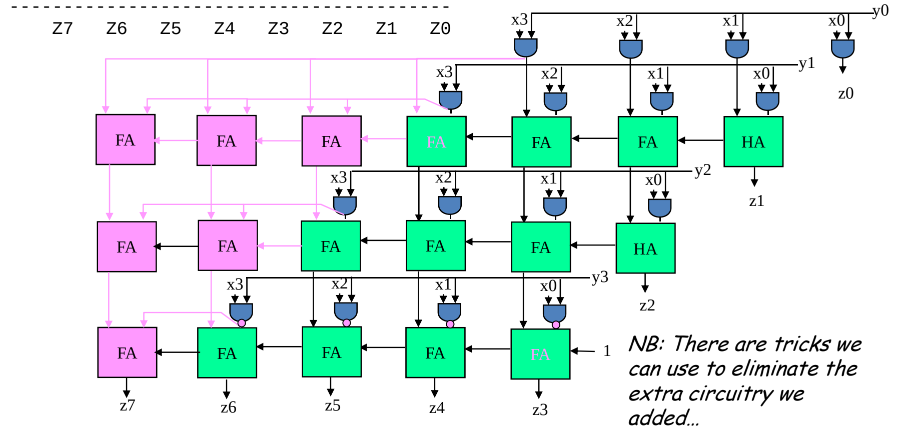
➢ Propagation delay ~2N

Combinational Multiplier (signed!)

```
                       X3   X2   X1   X0
                    *  Y3   Y2   Y1   Y0
                    --------------------
  X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
+ X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
+ X3Y2 X3Y2 X3Y2 X2Y2 X1Y2 X0Y2
- X3Y3 X3Y3 X2Y3 X1Y3 X0Y3
-----------------------------------------
   Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```



N bits
-2^{N-1} 2^{N-2} ··· ··· ··· 2^3 2^2 2^1 2^0
"sign bit"
Range: -2^{N-1} to $2^{N-1}-1$
"decimal" point



NB: There are tricks we can use to eliminate the extra circuitry we added…

2's Complement Multiplication

(Baugh-Wooley)

Step 1: two's complement operands so high order bit is -2^{N-1}. Must sign extend partial products and subtract the last one

```
                           X3   X2   X1   X0
                        *  Y3   Y2   Y1   Y0
                        --------------------
  X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
+ X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
+ X3Y2 X3Y2 X3Y2 X2Y2 X1Y2 X0Y2
- X3Y3 X3Y3 X2Y3 X1Y3 X0Y3
-----------------------------------------
   Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```

Step 2: don't want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

```
  X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
+                         1
+ X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
+                    1
+ X3Y2 X3Y2 X3Y2 X2Y2 X1Y2 X0Y2
+               1
+ X3Y3 X3Y3 X2Y3 X1Y3 X0Y3   }
+                         1   } -B = ~B + 1
+      1
-      1    1    1    1
```

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

```
                     X3Y0 X2Y0 X1Y0 X0Y0
+               X3Y1 X2Y1 X1Y1 X0Y1
+          X2Y2 X1Y2 X0Y2 X0Y2
+     X3Y3 X2Y3 X1Y3 X0Y3
+                       1
-        1    1    1    1
```

Step 4: finish computing the constants…

```
                     X3Y0 X2Y0 X1Y0 X0Y0
+               X3Y1 X2Y1 X1Y1 X0Y1
+          X2Y2 X1Y2 X0Y2 X0Y2
+     X3Y3 X2Y3 X1Y3 X0Y3
+   1               1
```

Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

Baugh Wooley Formulation –The Math

no insight required

Assuming X and Y are 4-bit twos complement numbers:

$$X = -2^3x_3 + \sum_{i=0}^{2} x_i2^i \qquad Y = -2^3y_3 + \sum_{i=0}^{2} y_i2^i$$

The product of X and Y is:

$$XY = x_3y_32^6 - \sum_{i=0}^{2} x_iy_32^{i+3} - \sum_{j=0}^{2} x_3y_j2^{j+3} + \sum_{i=0}^{2}\sum_{j=0}^{2} x_iy_j2^{i+j}$$

For twos complement, the following is true:

$$-\sum_{i=0}^{3} x_i2^i = -2^4 + \sum_{i=0}^{3} \overline{x_i2^i} + 1$$

The product then becomes:

$$XY = x_3y_32^6 + \sum_{i=0}^{2} \overline{x_iy_3}2^{i+3} + 2^3 - 2^6 + \sum_{j=0}^{2} \overline{x_3y_j}2^{j+3} + 2^3 - 2^6 + \sum_{i=0}^{2}\sum_{j=0}^{2} x_iy_j2^{i+j}$$

$$= x_3y_32^6 + \sum_{i=0}^{2} \overline{x_iy_3}2^{i+3} + \sum_{j=0}^{2} \overline{x_3y_j}2^{j+3} + \sum_{i=0}^{2}\sum_{j=0}^{2} x_iy_j2^{i+j} + 2^4 - 2^7$$

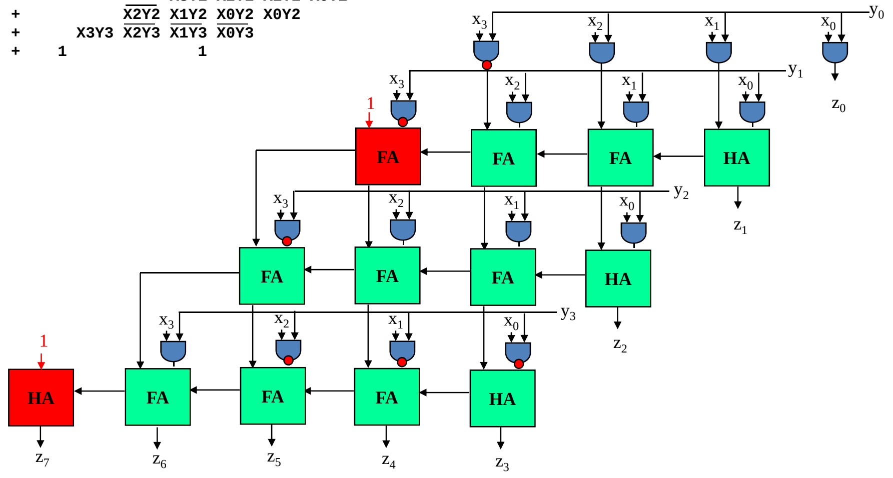
$$= -2^7 + x_3y_32^6 + (\overline{x_2y_3} + \overline{x_3y_2})2^5 + (\overline{x_1y_3} + \overline{x_3y_1} + x_2y_2 + 1)2^4$$

$$+ (\overline{x_0y_3} + \overline{x_3y_0} + x_1y_2 + x_2y_1)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 + (x_0y_1 + x_1y_0)2$$

$$+ (x_0y_0)2^0$$

2's Complement Multiplication

$$\begin{array}{lcccccccc}
 & & & & & \overline{X3Y0} & X2Y0 & X1Y0 & X0Y0 \\
+ & & & & \overline{X3Y1} & X2Y1 & X1Y1 & X0Y1 & \\
+ & & & \overline{X2Y2} & X1Y2 & X0Y2 & X0Y2 & & \\
+ & & X3Y3 & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & \\
+ & 1 & & & 1 & & & &
\end{array}$$



Multiplication in Verilog

You can use the "*" operator to multiply two numbers:

```
wire [9:0] a,b;
wire [19:0] result = a*b;    // unsigned multiplication!
```

If you want Verilog to treat your operands as signed two's complement numbers, add the keyword signed to your wire or reg declaration:

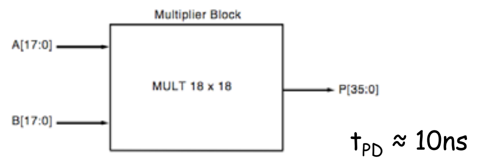
```
wire signed [9:0] a,b;
wire signed [19:0] result = a*b;  // signed multiplication!
```

Remember: unlike addition and subtraction, you need different circuitry if your multiplication operands are signed vs. unsigned. Same is true of the >>> (arithmetic right shift) operator. To get signed operations all operands must be signed.

To make a signed constant: 10'sh37C

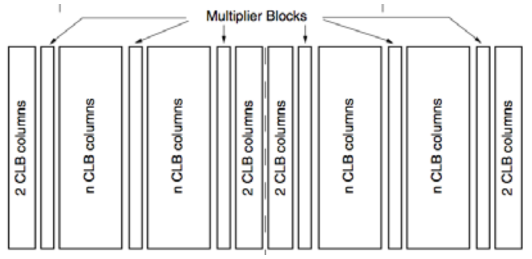
Multiplication on the FPGA

Hardware multiplier block: two 18-bit twos complement (signed) operands



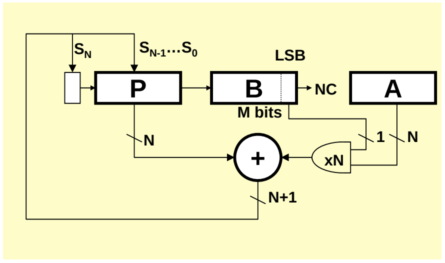
$t_{PD} \approx 10ns$

In the XC2V6000: 6 columns of mults, 24 in each column = 144 mults



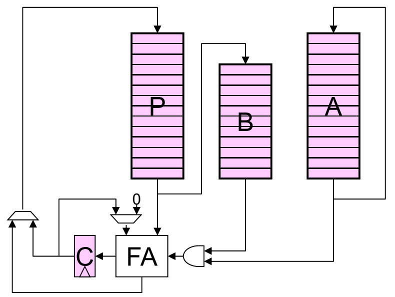
Sequential Multiplier

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:



```
Init: P←0, load A and B

Repeat M times {
   P ← P + (B$_{LSB}$==1 ? A : 0)
   shift P/B right one bit
}

Done: (N+M)-bit result in P/B
```

Bit-Serial Multiplication



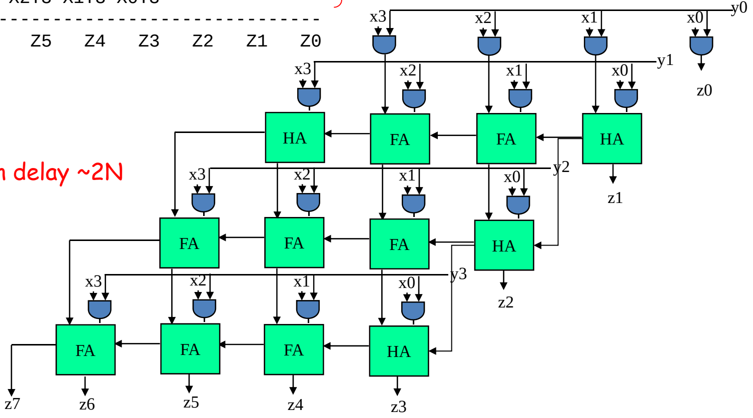
```
Init: P = 0; Load A,B

Repeat M times {
  Repeat N times {
    shift A,P:
    Amsb = Alsb
    Pmsb = Plsb + Alsb*Blsb + C/0
  }
  shift P,B: Pmsb = C, Bmsb = Plsb
}

(N+M)-bit result in P/B
```

Combinational Multiplier (unsigned)

```
             X3   X2   X1   X0   ← multiplicand
         *   Y3   Y2   Y1   Y0   ← multiplier
         ------------------------
            X3Y0 X2Y0 X1Y0 X0Y0
+      X3Y1 X2Y1 X1Y1 X0Y1
+   X3Y2 X2Y2 X1Y2 X0Y2
+ X3Y3 X2Y3 X1Y3 X0Y3
-------------------------------------
  Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```

Partial products, one for each bit in multiplier (each bit needs just one AND gate)

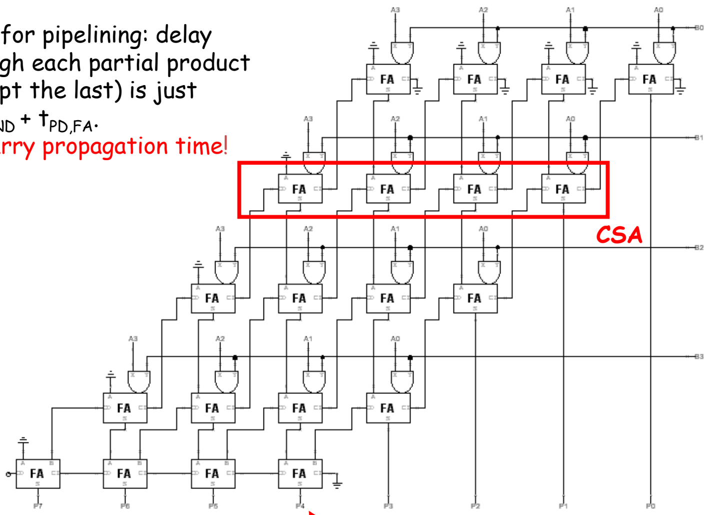
➢ Propagation delay ~2N



Useful building block: Carry-Save Adder

Good for pipelining: delay through each partial product (except the last) is just $t_{PD,AND} + t_{PD,FA}$.

No carry propagation time!



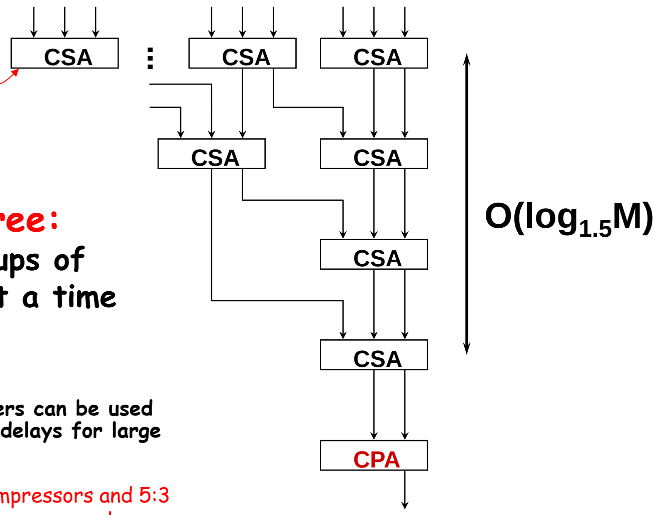
Last stage is still a carry-propagate adder (CPA)

Wallace Tree Multiplier

This is called a 3:2 counter by multiplier hackers: counts number of 1's on the 3 inputs, outputs 2-bit result.

Wallace Tree:

Combine groups of three bits at a time



Higher fan-in adders can be used to further reduce delays for large M.

4:2 compressors and 5:3 counters are popular building blocks.

Multiplication by a constant

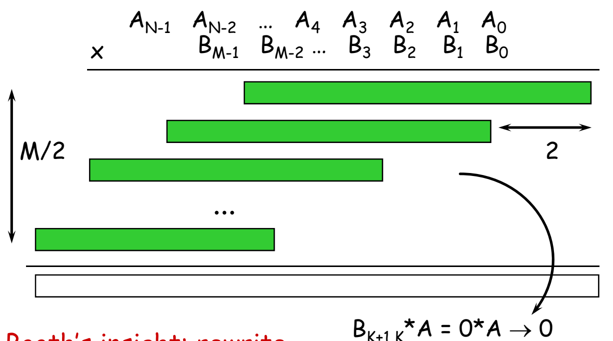
- If one of the operands is a constant, make it the multiplier (B in the earlier examples). For each "1" bit in the constant we get a partial product (PP) – may be noticeably fewer PPs than in the general case.
  - For example, in general multiplying two 4-bit operands generates four PPs (3 rows of full adders). If the multiplier is say, 12 (4'b1100), then there are only two PPs: 8*A+4*A (only 1 row of full adders).
  - But lots of "1"s means lots of PPs… can we improve on this?
- If we allow ourselves to subtract PPs as well as adding them (the hardware cost is virtually the same), we can re-encode arbitrarily long contiguous runs of "1" bits in the multiplier to produce just two PPs.

…011110… = …100000… – …000010… = …010001̄0…

where $\bar{1}$ indicates subtracting a PP instead of adding it. Thus we've re-encoded the multiplier using 1,0,-1 digits – aka *canonical signed digit* – greatly reducing the number of additions required.

Booth Recoding: Higher-radix mult.

Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would halve the number of columns and halve the latency of the multiplier!



A_{N-1} A_{N-2} … A_4 A_3 A_2 A_1 A_0
x B_{M-1} B_{M-2} … B_3 B_2 B_1 B_0
M/2
2
…

$B_{K+1,K}$*A = 0*A → 0
= 1*A → A
= 2*A → 4A – 2A
= 3*A → 4A – A

Booth's insight: rewrite 2*A and 3*A cases, leave 4A for next partial product to do!

Booth recoding

On-the-fly canonical signed digit encoding!

current bit pair   from previous bit pair

B_{K+1}	B_K	B_{K-1}	action	
0	0	0	add 0	
0	0	1	add A	
0	1	0	add A	
0	1	1	add 2*A	
1	0	0	sub 2*A	
1	0	1	sub A	← -2*A+A
1	1	0	sub A	
1	1	1	add 0	← -A+A

A "1" in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

Summary

- Performance of arithmetic blocks dictate the performance of a digital system
- Architectural and logic transformations can enable significant speed up (e.g., adder delay from $O(N)$ to $O(\log_2(N))$)
- Similar concepts and formulation can be applied at the system level
- Timing analysis is tricky: watch out for false paths!
- Area-Delay trade-offs (serial vs. parallel implementations)