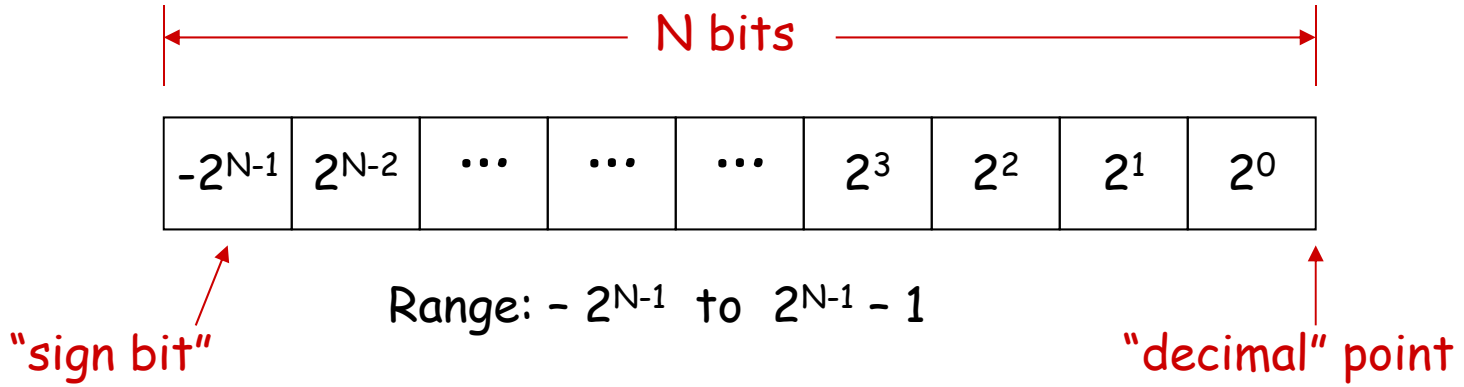
Arithmetic Circuits & Multipliers

- Addition, subtraction
- Performance issues
  -- ripple carry
  -- carry bypass
  -- carry skip
  -- carry lookahead
- Multipliers

Reminder: Lab #3 due tonight!
Pizza Wed 6p

Signed integers: 2's complement



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:

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

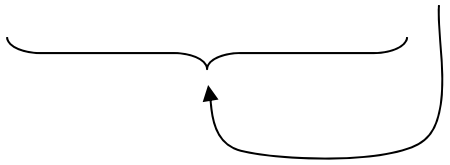
Sign extension

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
```



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

Adder: a circuit that does addition

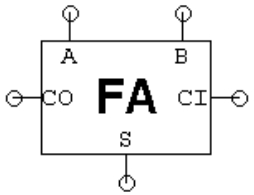
Here's an example of binary addition as one might do it by "hand":

Carries from previous column

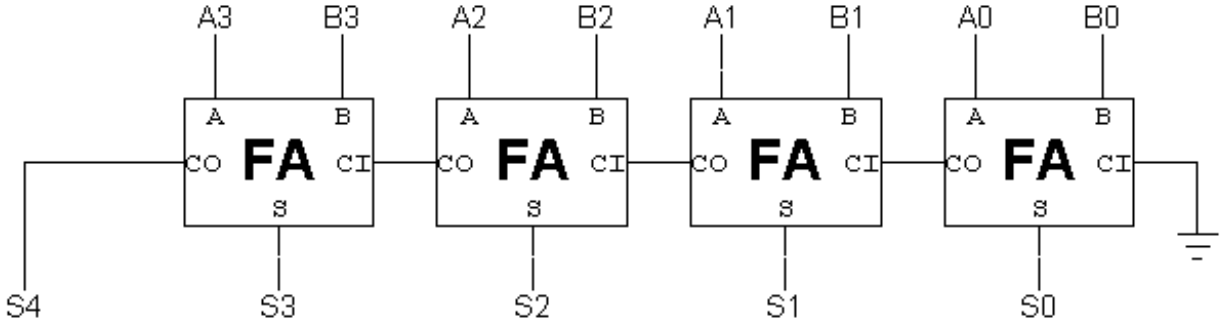
```
  1 1 0 1
    1101
+   0101
-------
   10010
```

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

If we build a circuit that implements one column:

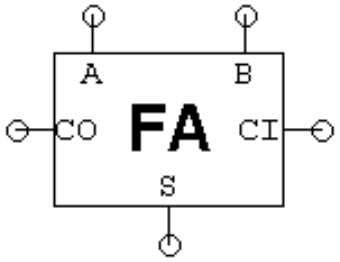


we can quickly build a circuit to add two 4-bit numbers…



"Ripple-carry adder"

"Full Adder" building block



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

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

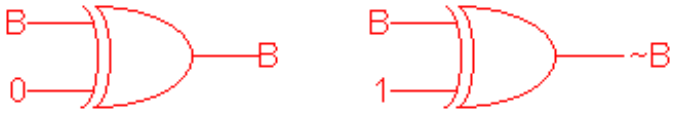
$$S = A \oplus B \oplus C$$

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

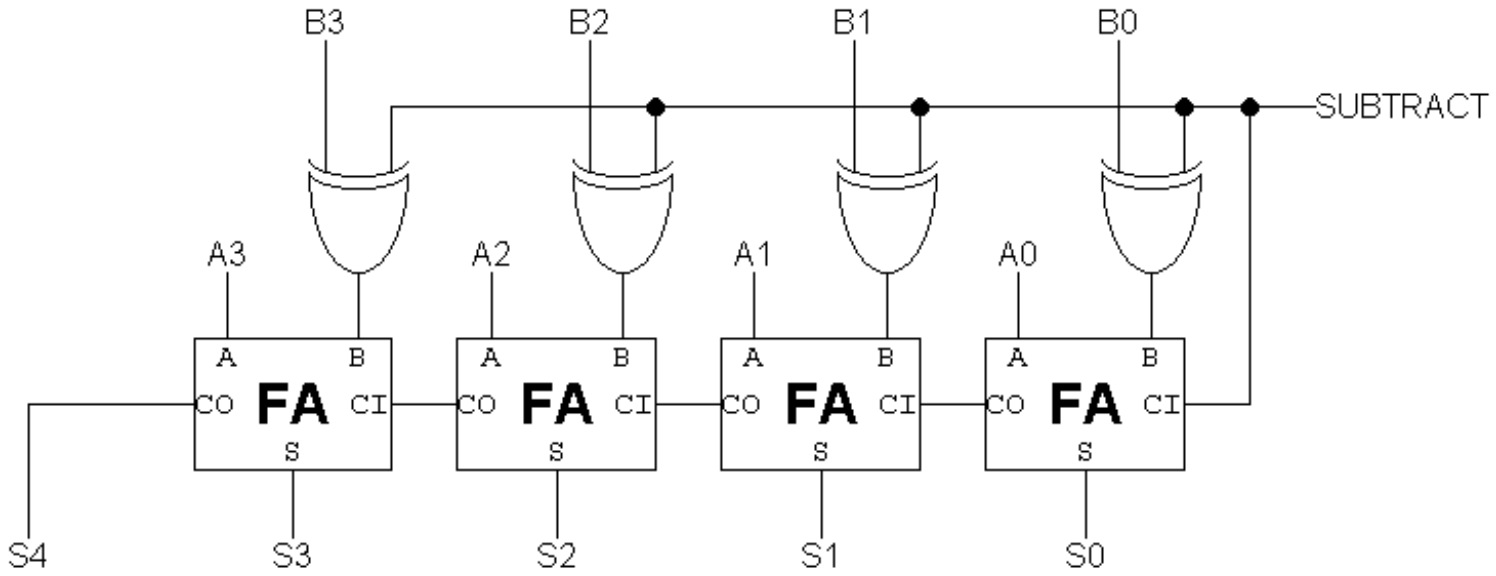
Subtraction: A-B = A + (-B)

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.

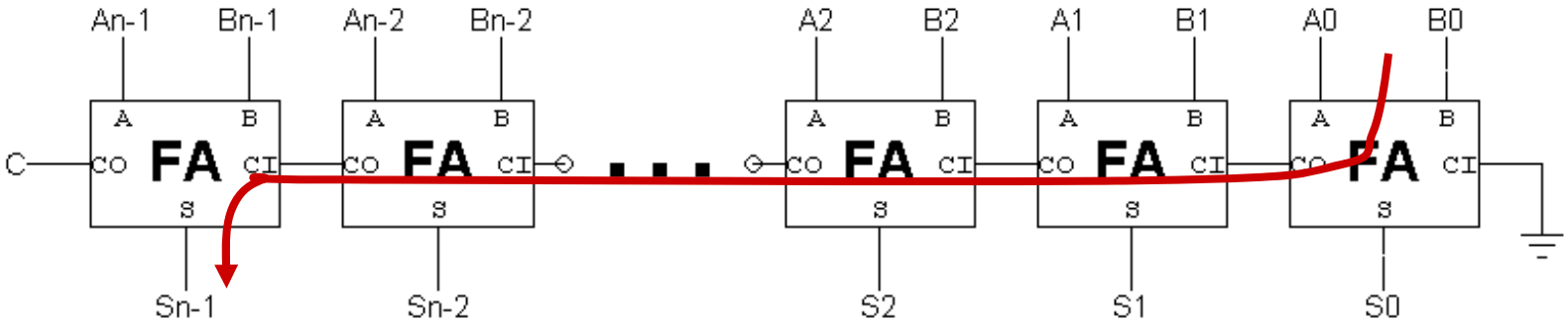
In many designs one chooses a "word size" (many computers use 32 or 64 bits) and all arithmetic results are truncated to that number of bits, i.e., arithmetic is performed modulo $2^{\text{word size}}$.

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.

$$t_{PD} = (N-1)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

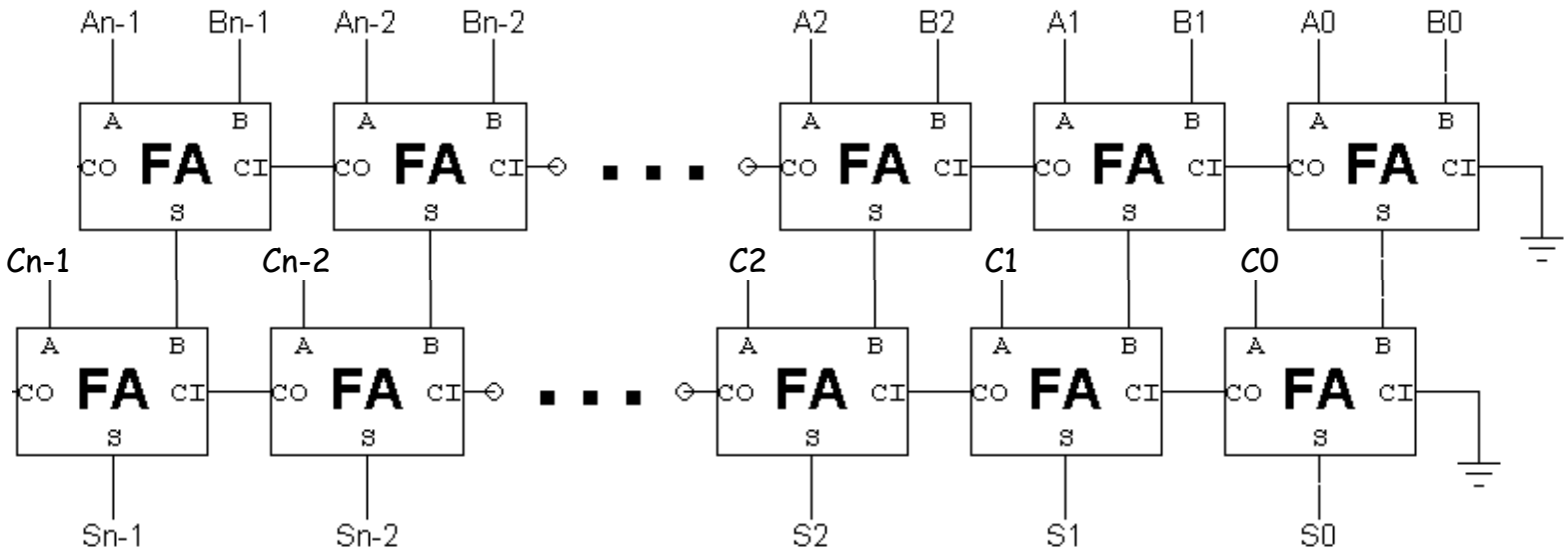
CI to CO

CI_{N-1} to S_{N-1}

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

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 * $t_{PD,N\text{-BIT RIPPLE}}$?

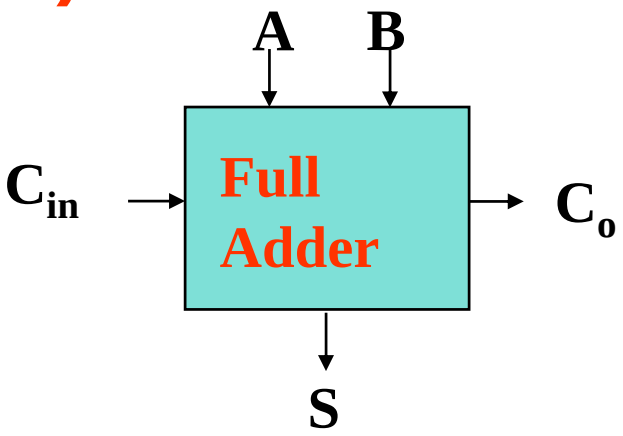
Nope! t_{PD} of this circuit = $t_{PD,N\text{-BIT RIPPLE}}$ + $t_{PD,FA}$!!!

— Timing analysis is tricky!

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

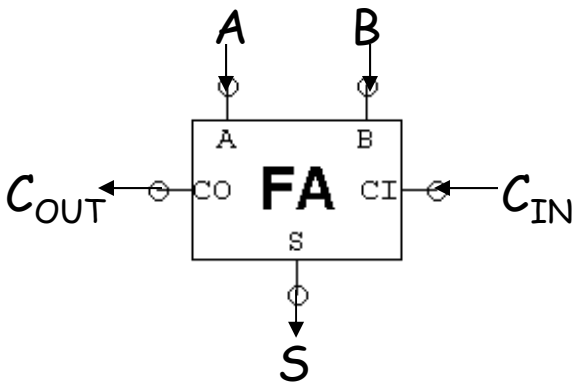
$$C_o(G,P) = G + PC_i$$

$$S(G,P) = P \oplus C_i$$

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}:



$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$
$$= AB + (A + B)C_{IN}$$
$$= G + P\,C_{IN}$$

generate propagate

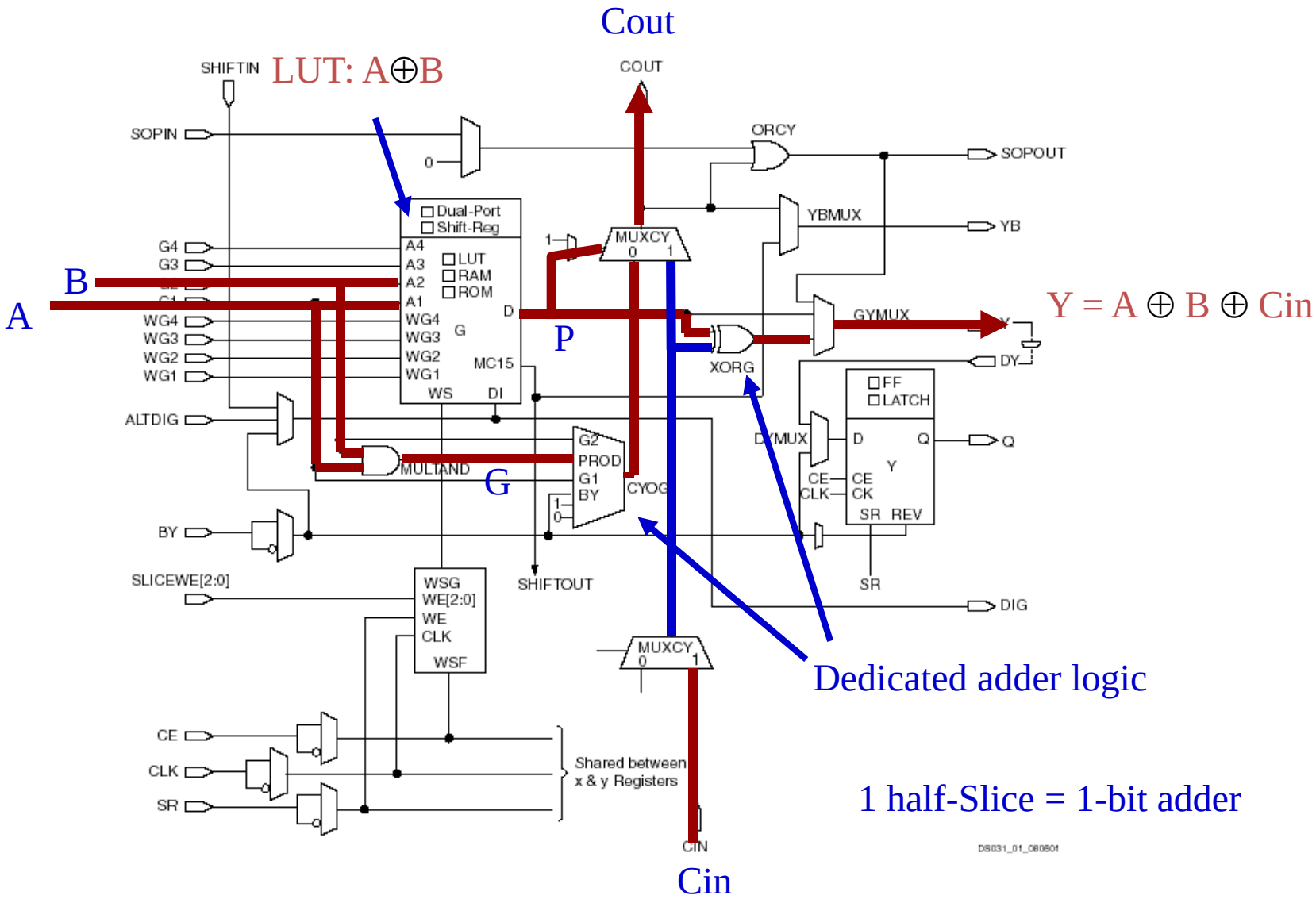
where $G = AB$
$P = A + B$

```
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
```

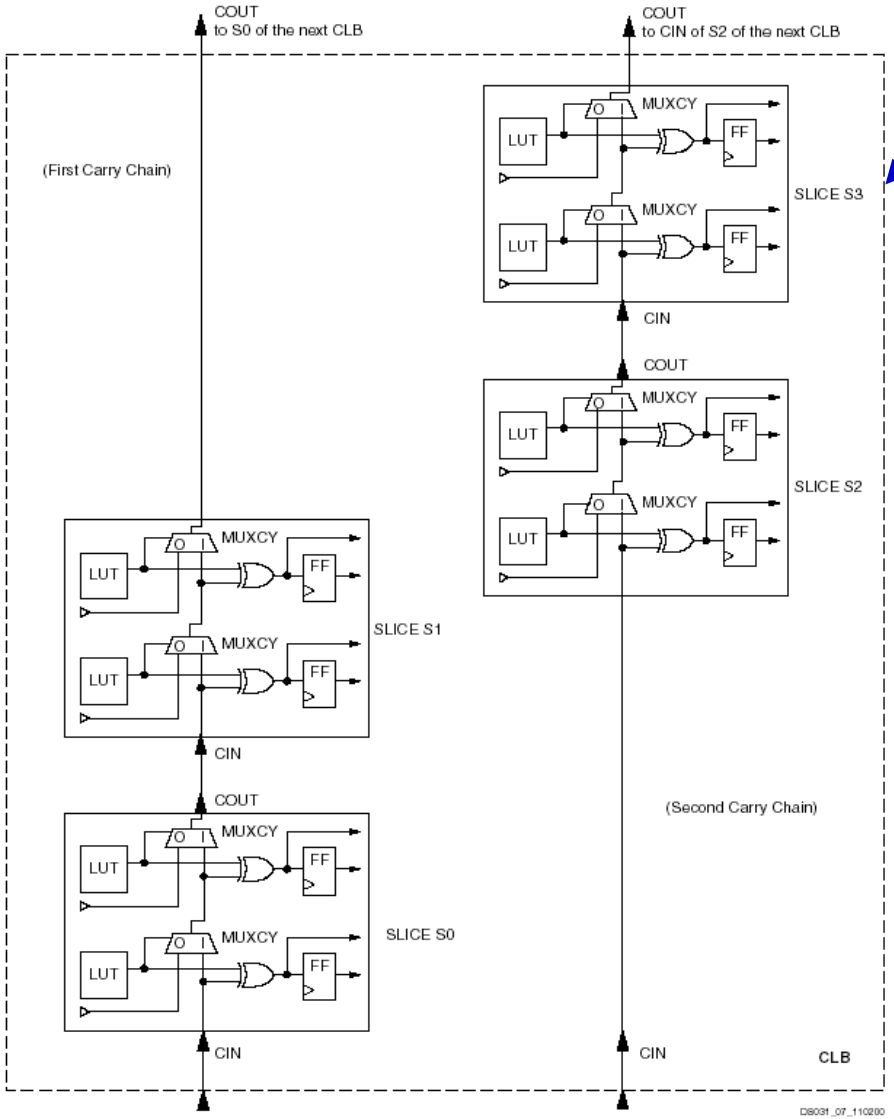
Actually, P is usually defined as P = A^B which won't change C_{OUT} but will allow us to express S as a simple function :

$S = P \hat{} C_{IN}$

Virtex II Adder Implementation

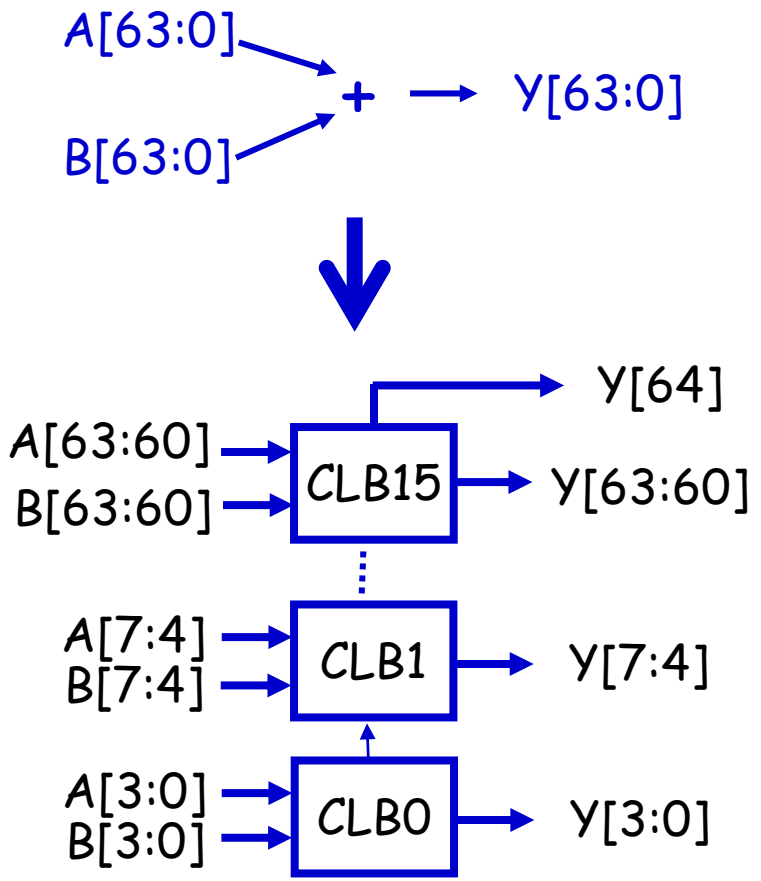


Virtex II Carry Chain



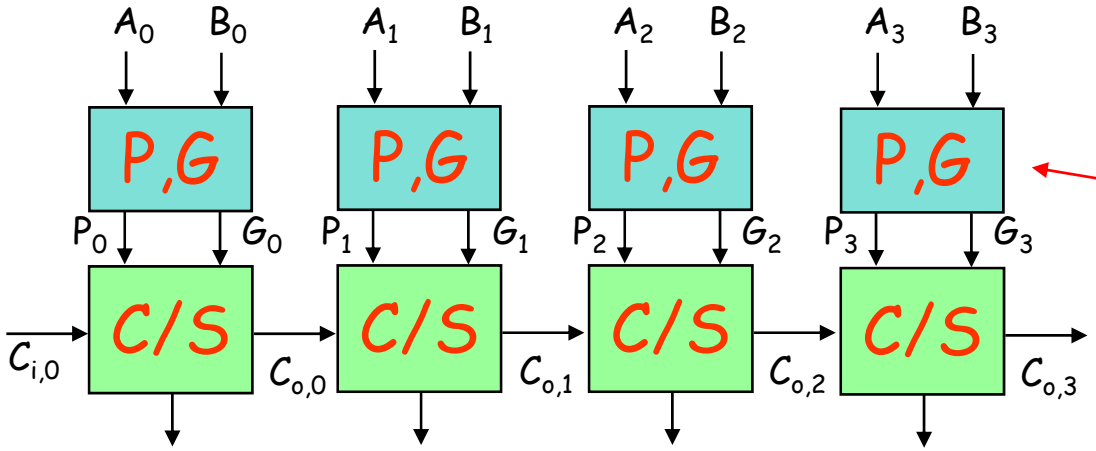
1 CLB = 4 Slices = 2, 4-bit adders

64-bit Adder: 16 CLBs

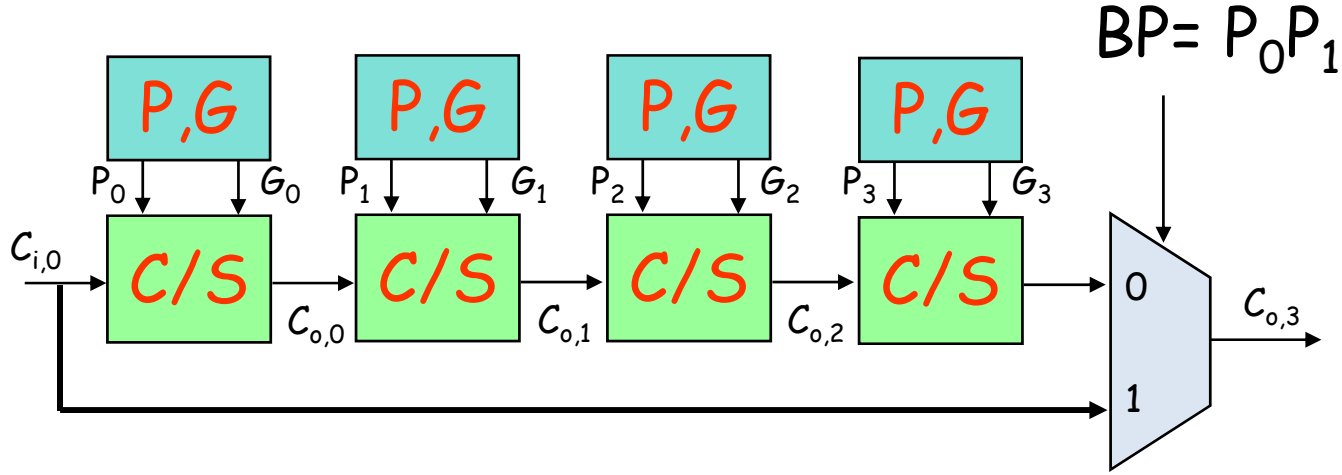


CLBs must be in same column

Carry Bypass Adder

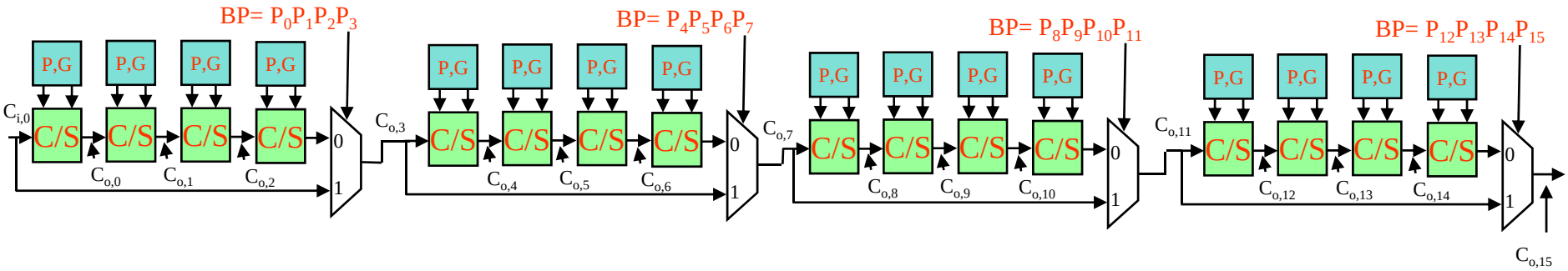


Can compute P, G in parallel for all bits



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

16-bit Carry Bypass Adder



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

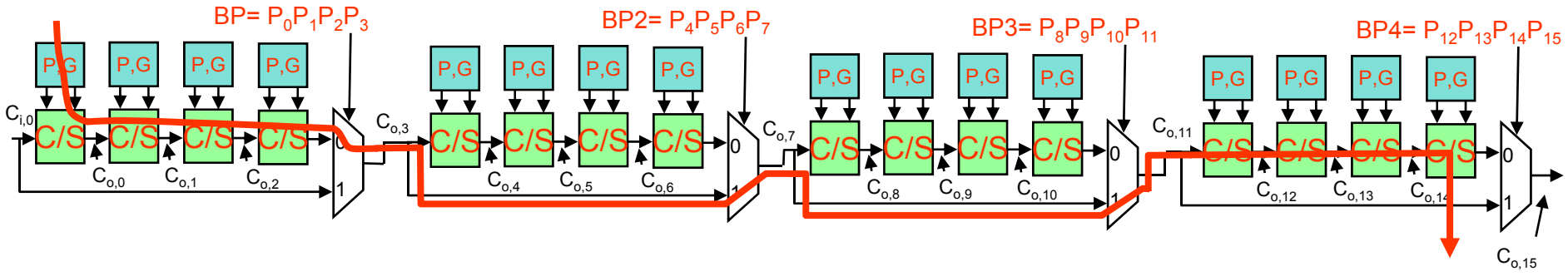
Assume the following for delay each gate:

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



For the second stage, is the critical path:

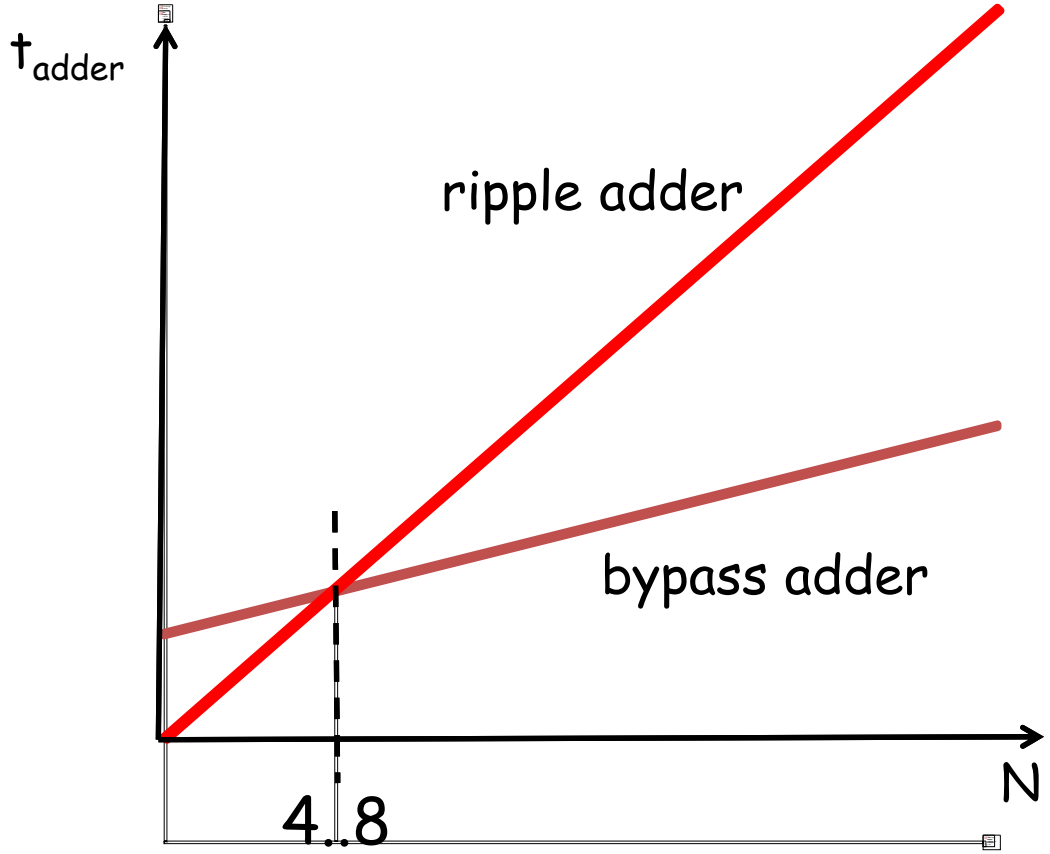
BP2 = 0 or BP2 = 1 ?

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:

$$C_N = G_{N-1} + P_{N-1}C_{N-1}$$

$$= G_{N-1} + P_{N-1}\,G_{N-2} + P_{N-1}\,P_{N-2}C_{N-2}$$

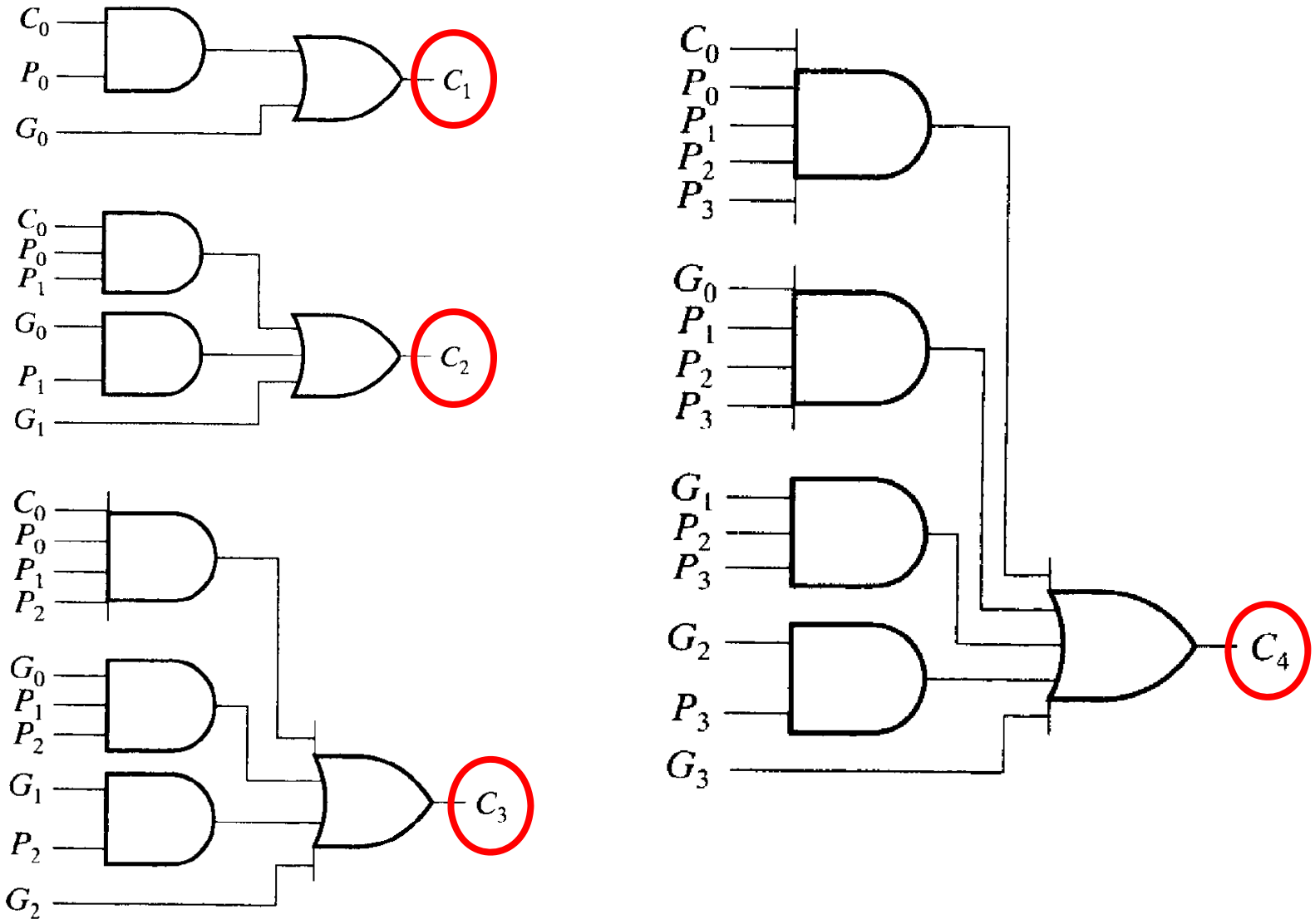
$$= G_{N-1} + P_{N-1}\,G_{N-2} + P_{N-1}\,P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0 C_{IN}$$

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

*assuming gates with N inputs

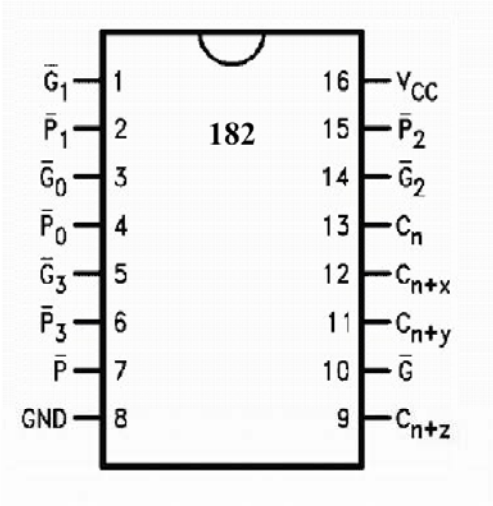
- Idea: pre-compute all carry bits as f(Gs,Ps,C_{IN})

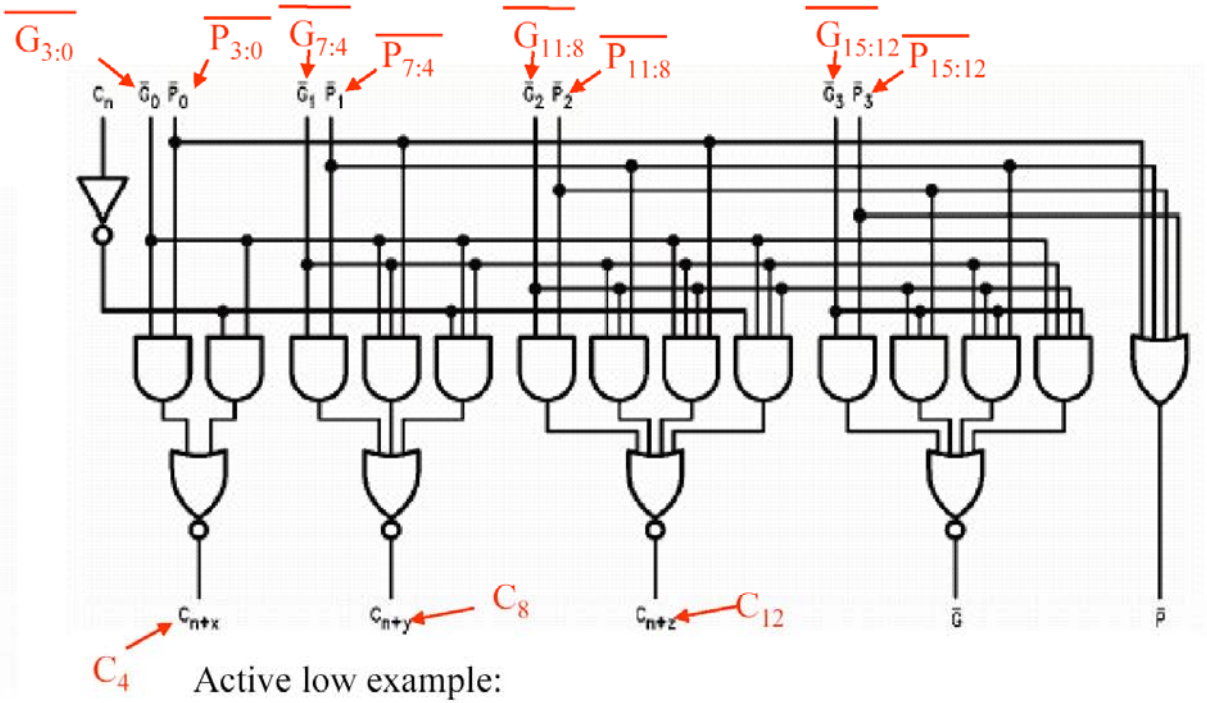
Carry Lookahead Circuits



The 74182 Carry Lookahead Unit

74182 carry lookahead unit





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

Active low example:

$$C_{n+x} = \overline{\overline{G0}\cdot\overline{P0} + \overline{G0}\cdot\overline{C_n}}$$

$$= \overline{\overline{G0}\cdot\overline{P0}\cdot\overline{G0}\cdot\overline{C_n}}$$

$$= (G0+P0)\cdot(G0+C_n) = G0 + P0C_n$$

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

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

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

$$= G_{11:0} + P_{11:0}C_n$$

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:

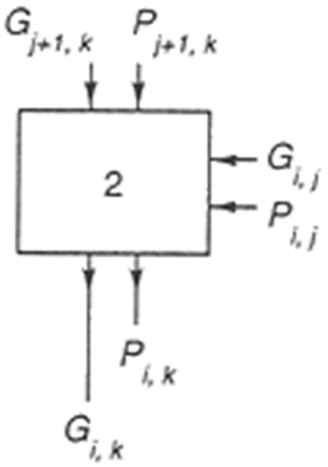
$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

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

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

"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"

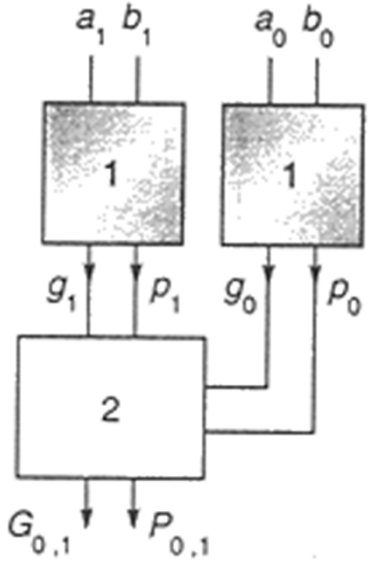
where I < J and J+1 < K



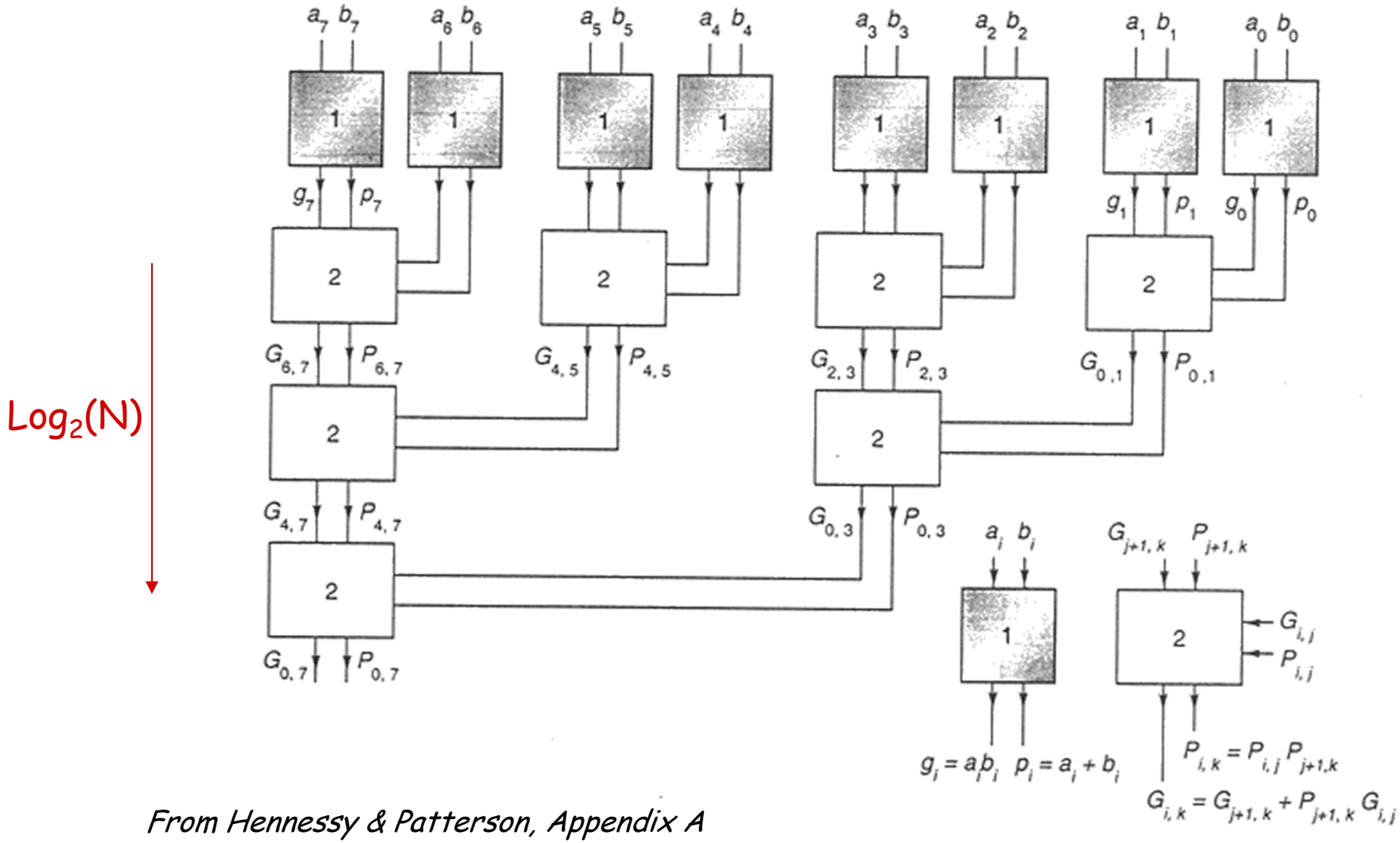
Hierarchical building block

P/G generation

1st level of lookahead

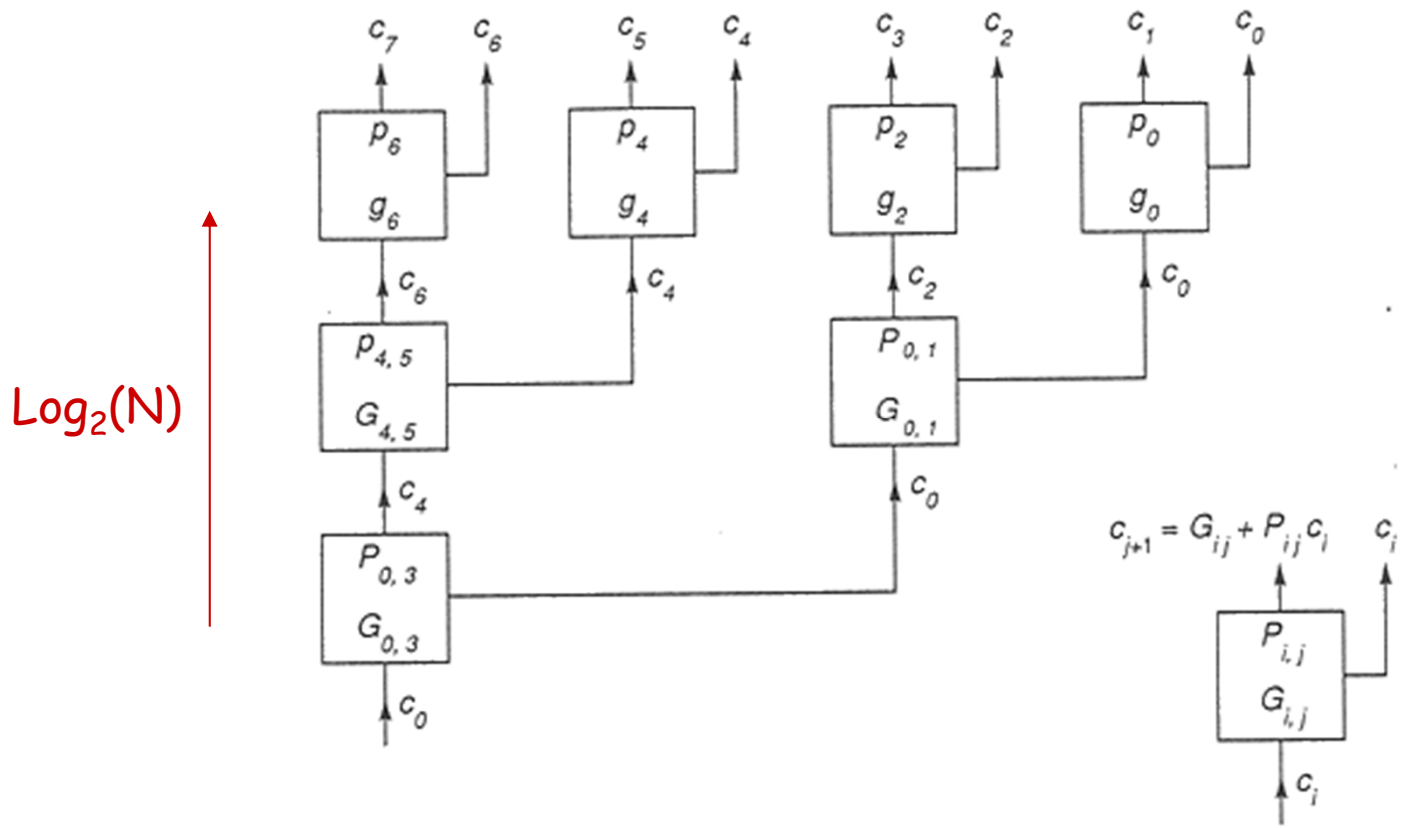


8-bit CLA (P/G generation)

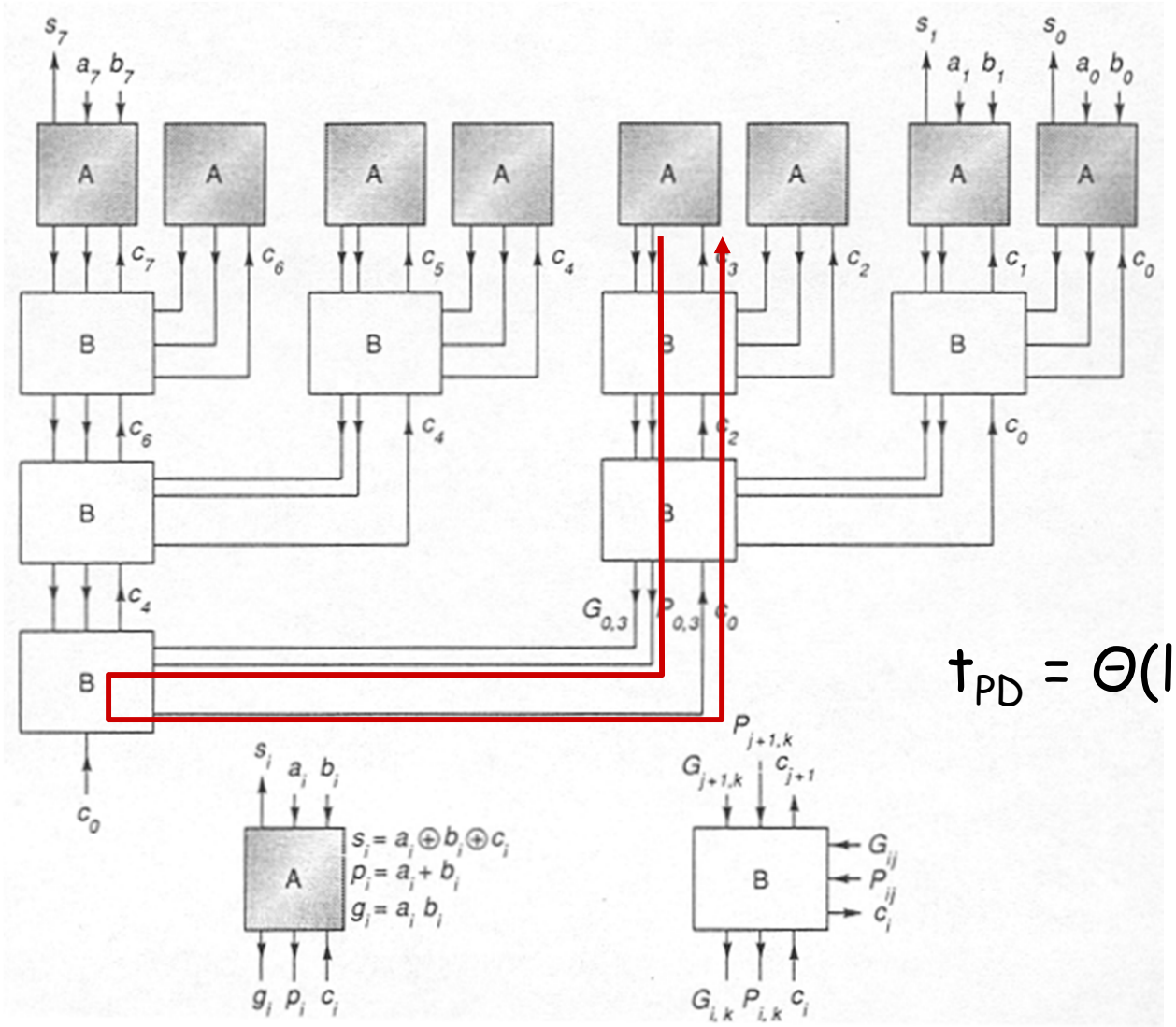


From Hennessy & Patterson, Appendix A

8-bit CLA (carry generation)



8-bit CLA (complete)



$t_{PD} = \Theta(\log(N))$

Unsigned Multiplication

AB_i called a "partial product"

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

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

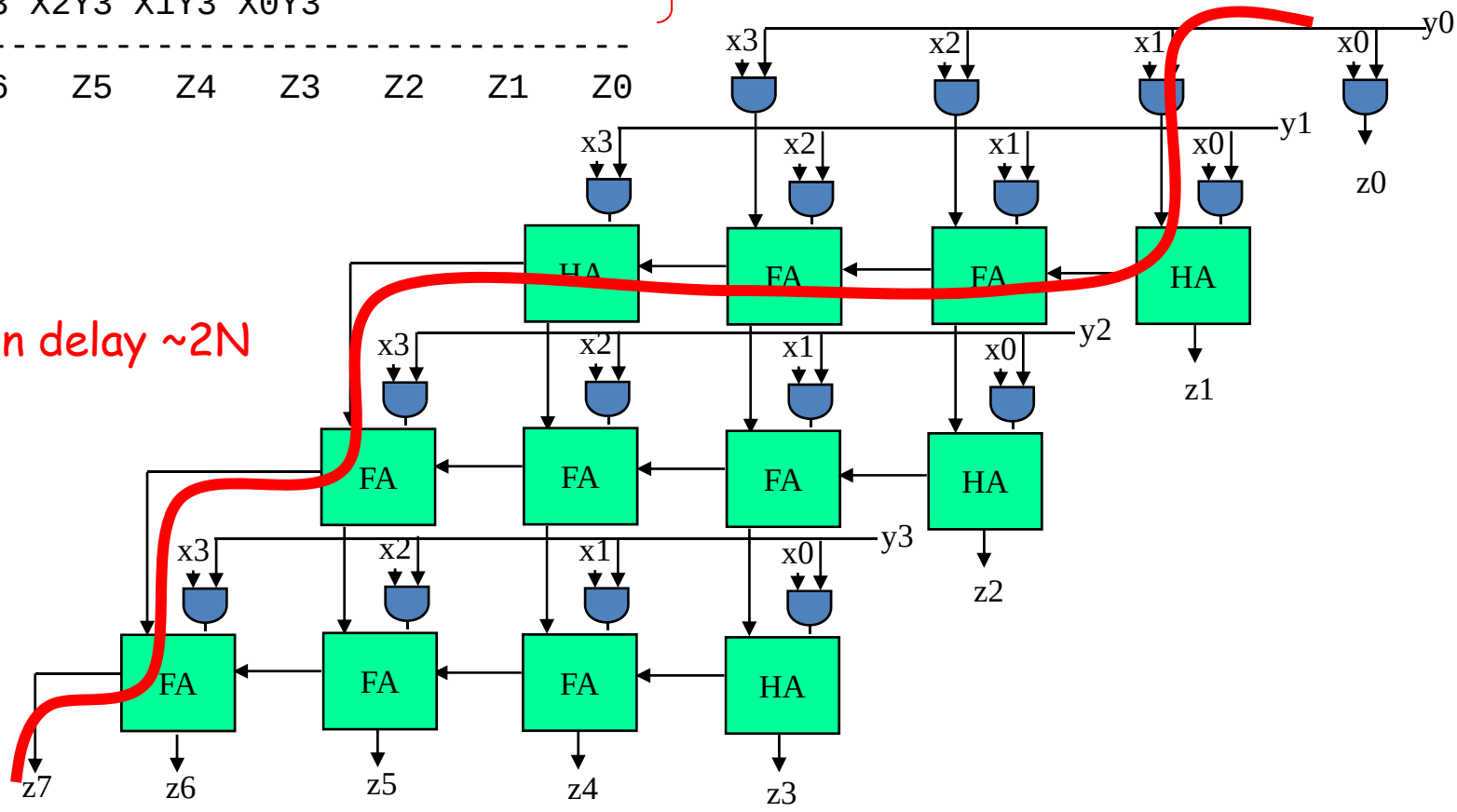
Easy part: forming partial products
(just an AND gate since B_I is either 0 or 1)
Hard part: adding M N-bit partial products

Combinational Multiplier (unsigned)

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

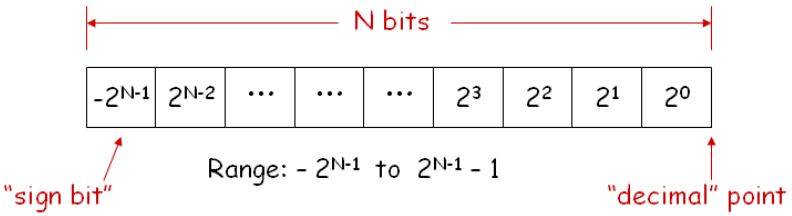
Partial products, one for each bit in multiplier (each bit needs just one AND gate)

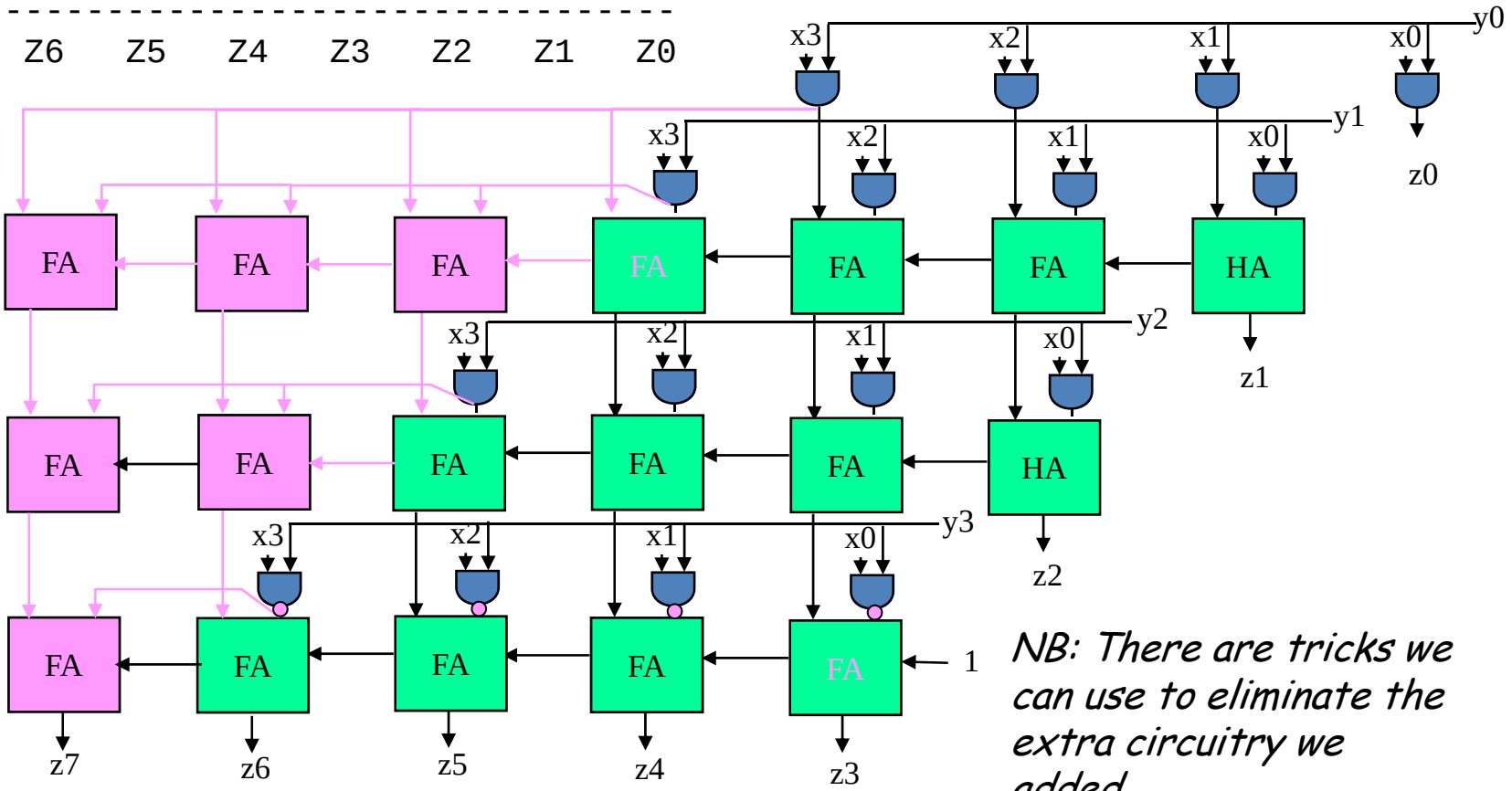
- Propagation delay ~2N



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
```





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      } -B = ~B + 1
+                         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:

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

The product of X and Y is:

$$XY = x_3 y_3 2^6 - \sum_{i=0}^{2} x_i y_3 2^{i+3} - \sum_{j=0}^{2} x_3 y_j 2^{j+3} + \sum_{i=0}^{2}\sum_{j=0}^{2} x_i y_j 2^{i+j}$$

For twos complement, the following is true:

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

The product then becomes:

$$XY = x_3 y_3 2^6 + \sum_{i=0}^{2} \overline{x_i y_3} 2^{i+3} + 2^3 - 2^6 + \sum_{j=0}^{2} \overline{x_3 y_j} 2^{j+3} + 2^3 - 2^6 + \sum_{i=0}^{2}\sum_{j=0}^{2} x_i y_j 2^{i+j}$$

$$= x_3 y_3 2^6 + \sum_{i=0}^{2} \overline{x_i y_3} 2^{i+3} + \sum_{j=0}^{2} \overline{x_3 y_j} 2^{j+3} + \sum_{i=0}^{2}\sum_{j=0}^{2} x_i y_j 2^{i+j} + 2^4 - 2^7$$

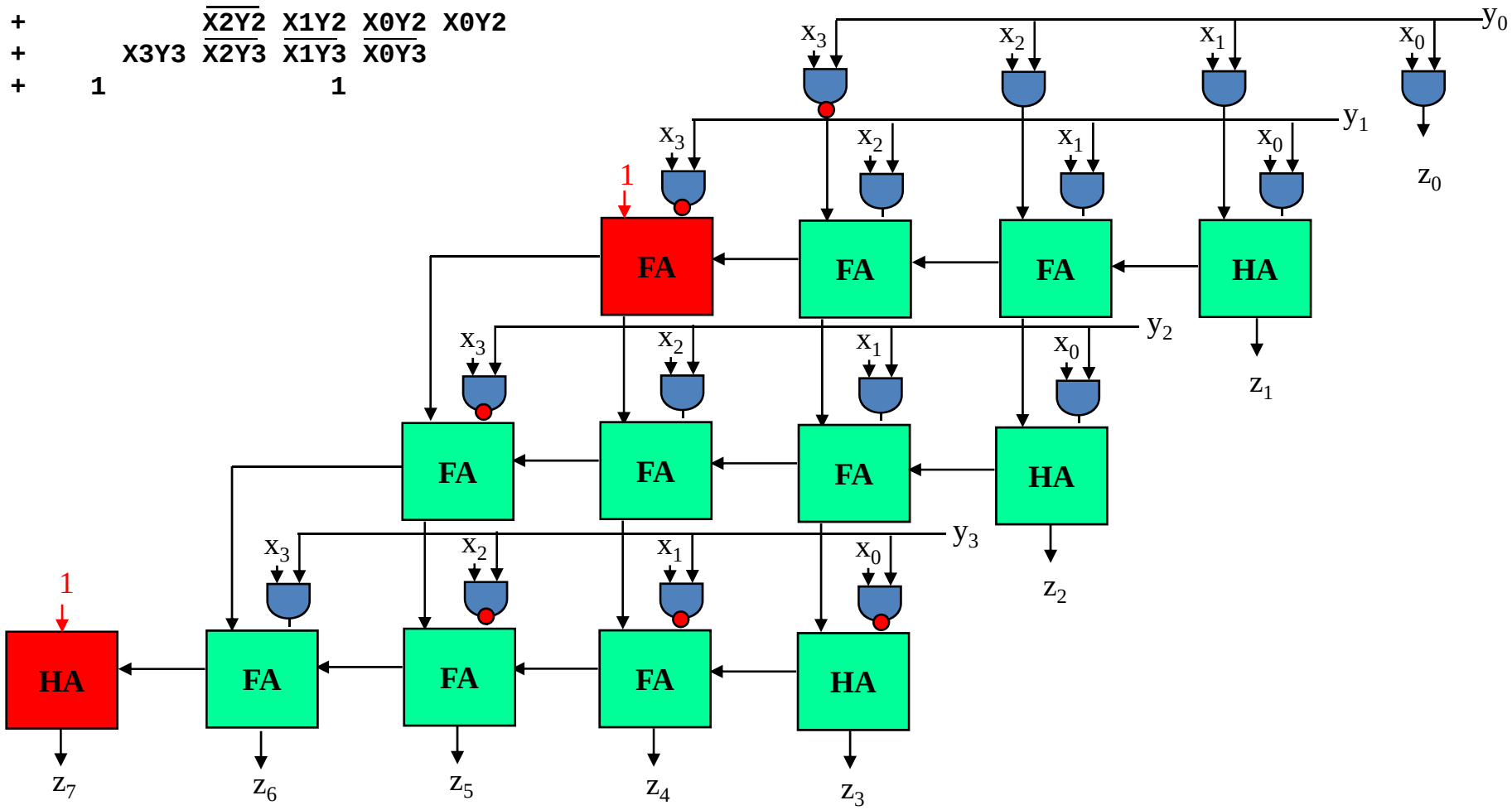
$$= -2^7 + x_3 y_3 2^6 + \overline{(x_2 y_3} + \overline{x_3 y_2})2^5 + \overline{(x_1 y_3} + \overline{x_3 y_1} + x_2 y_2 + 1)2^4$$

$$+ \overline{(x_0 y_3} + \overline{x_3 y_0} + x_1 y_2 + x_2 y_1)2^3 + (x_0 y_2 + x_1 y_1 + x_2 y_0)2^2 \; 1$$

$$+ (x_0 y_1 + x_1 y_0)2^1 + (x_0 y_0)2^0$$

2's Complement Multiplication

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



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:

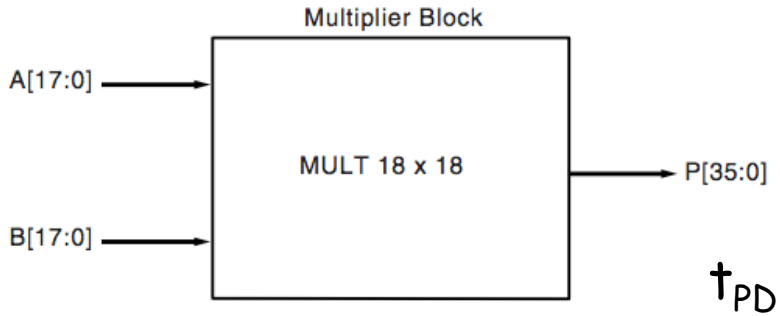
```
wire signed [9:0] a,b;
wire signed [19:0] result = a*b;  // signed multiplication!
```

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

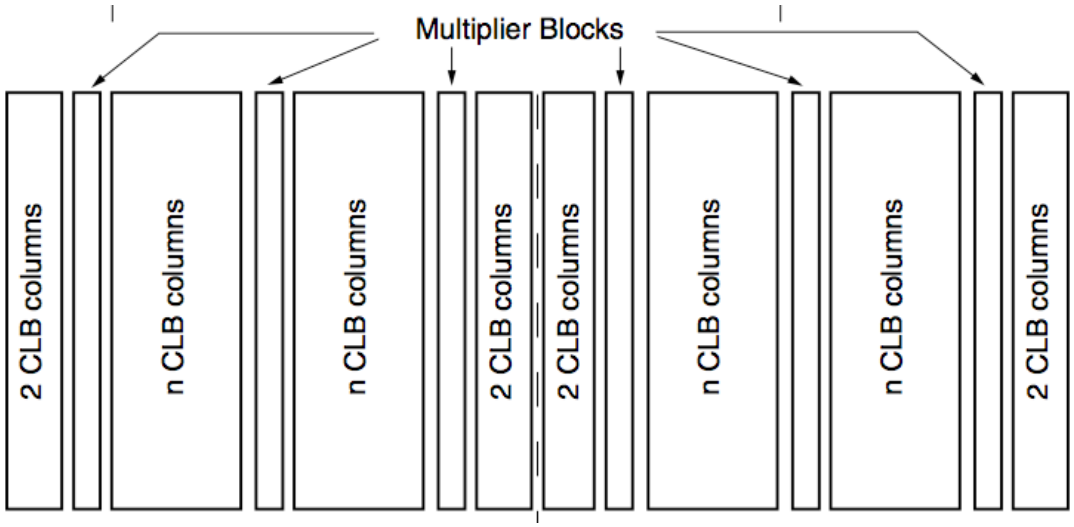
Multiplication on the FPGA

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



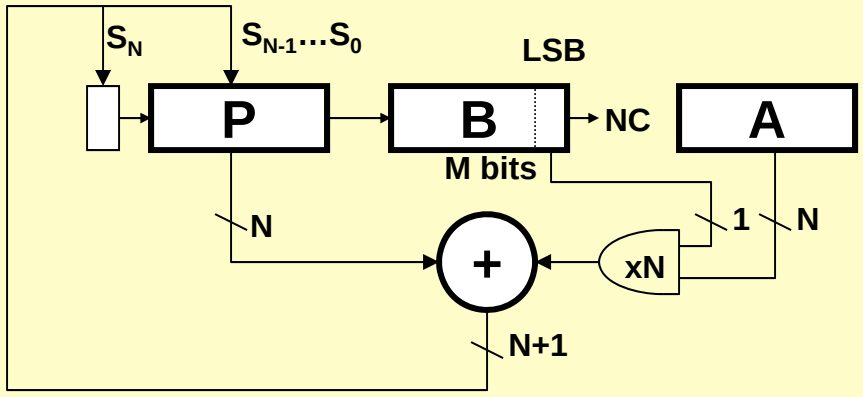
$t_{PD} \approx 10ns$

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



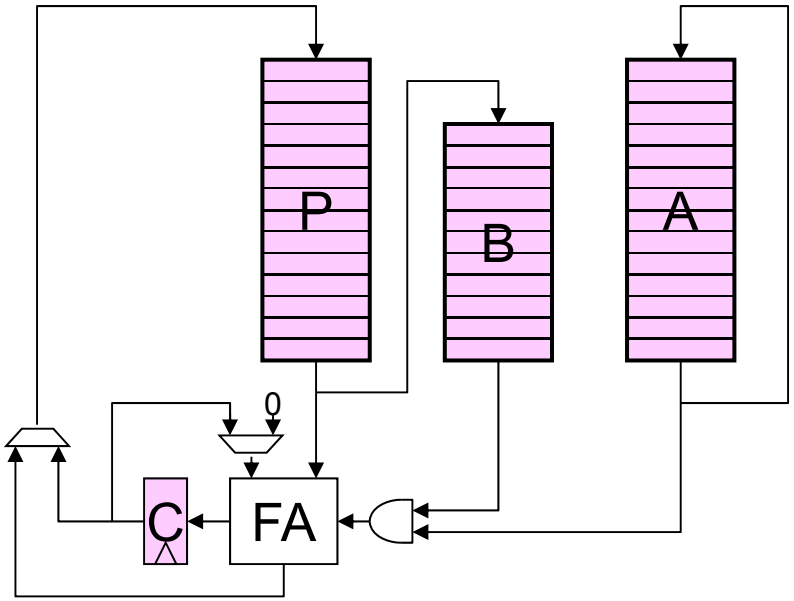
Sequential Multiplier

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



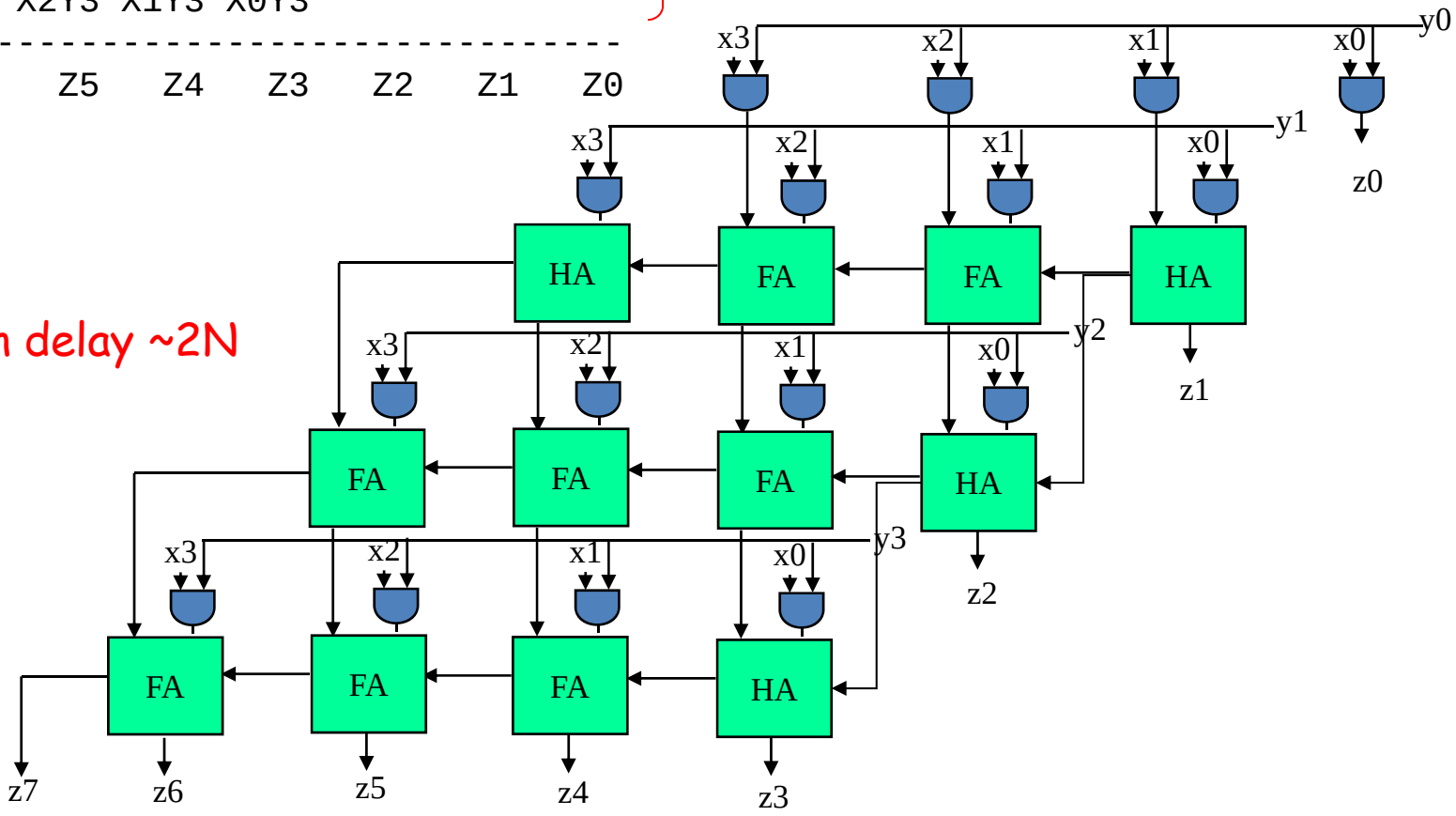

```
Init: P = 0; Load A,B

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)

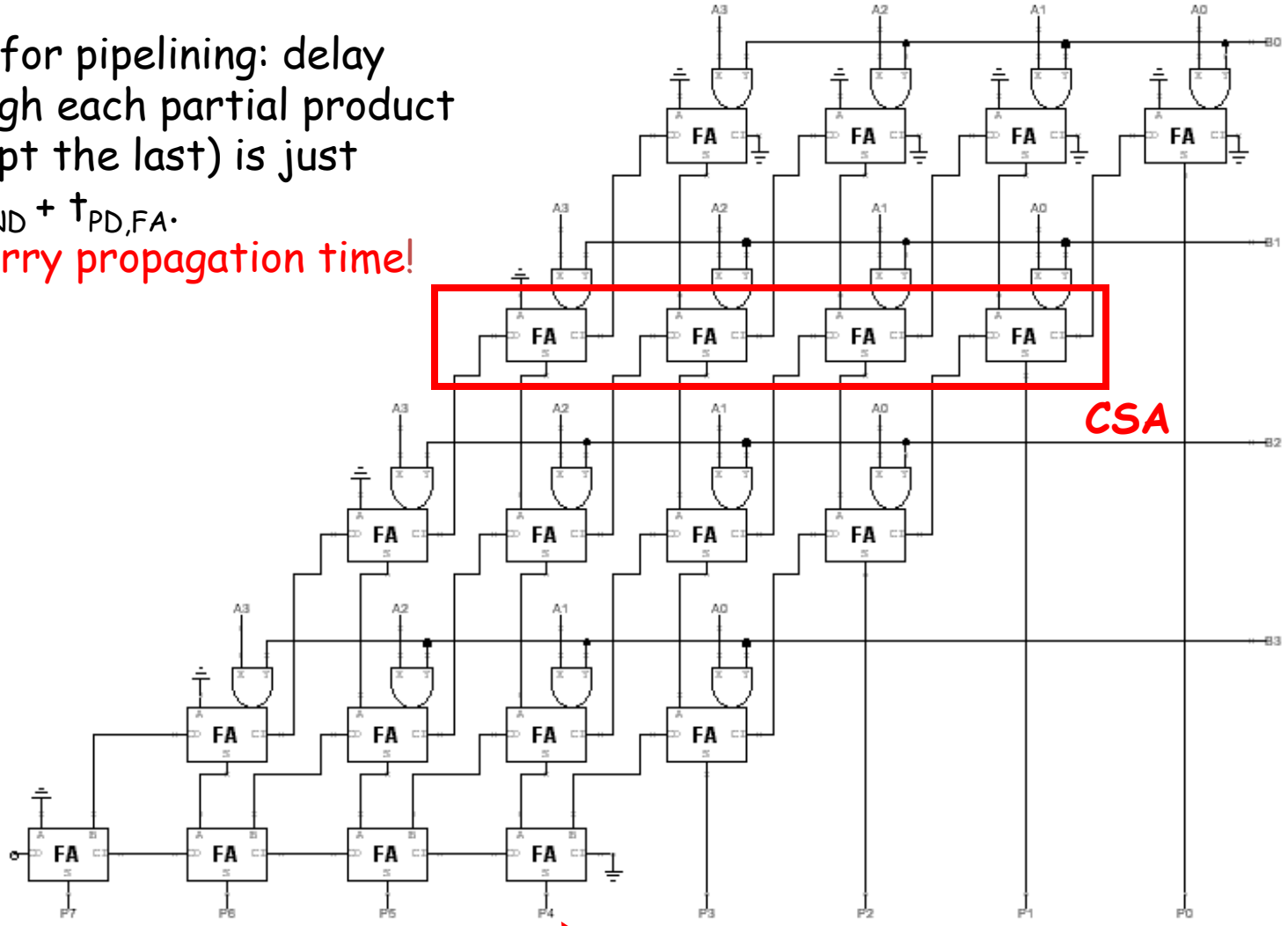
- Propagation delay ~2N



Useful building block: Carry-Save Adder

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

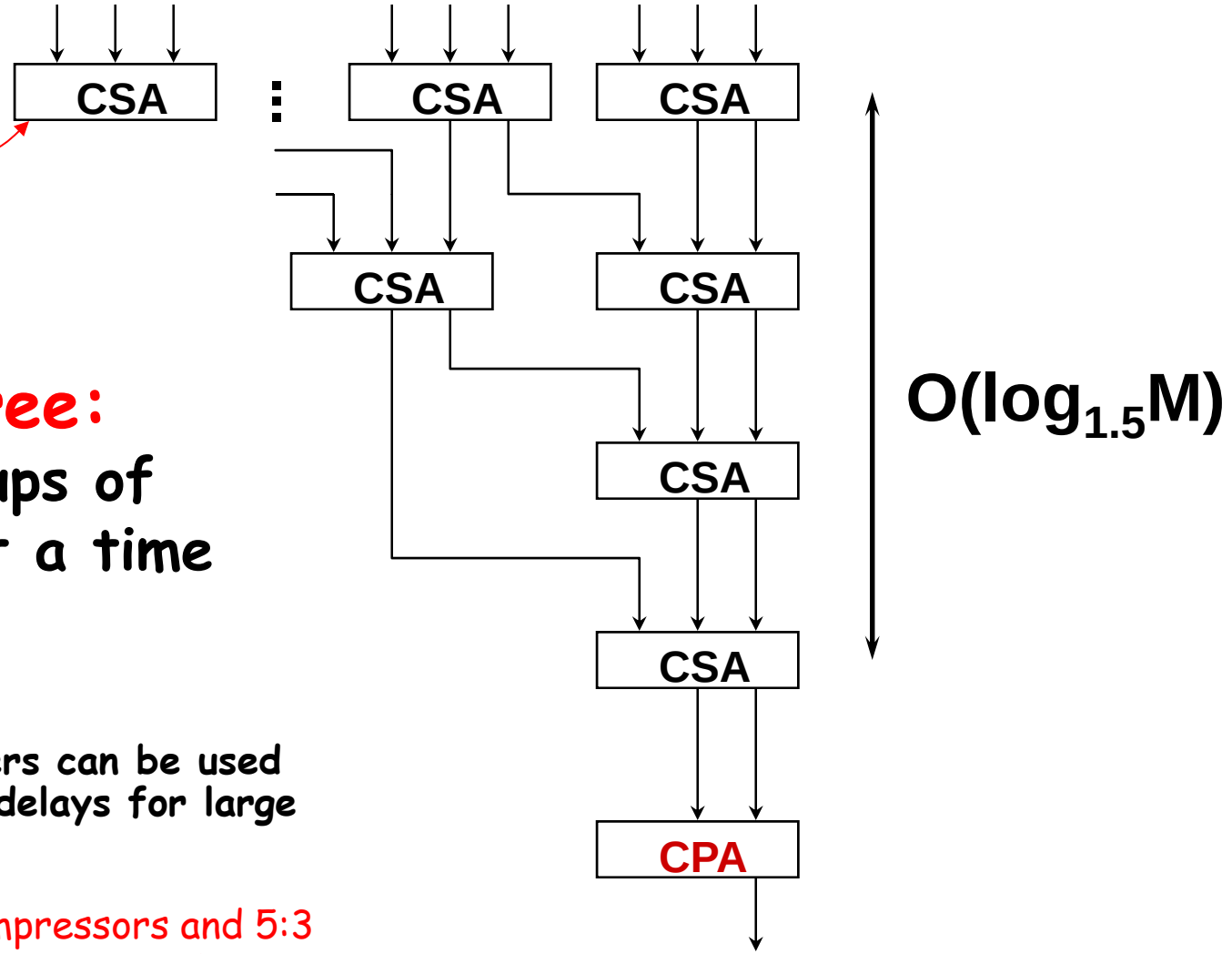
No carry propagation time!



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.

Wallace Tree *
Four Bit Multiplier

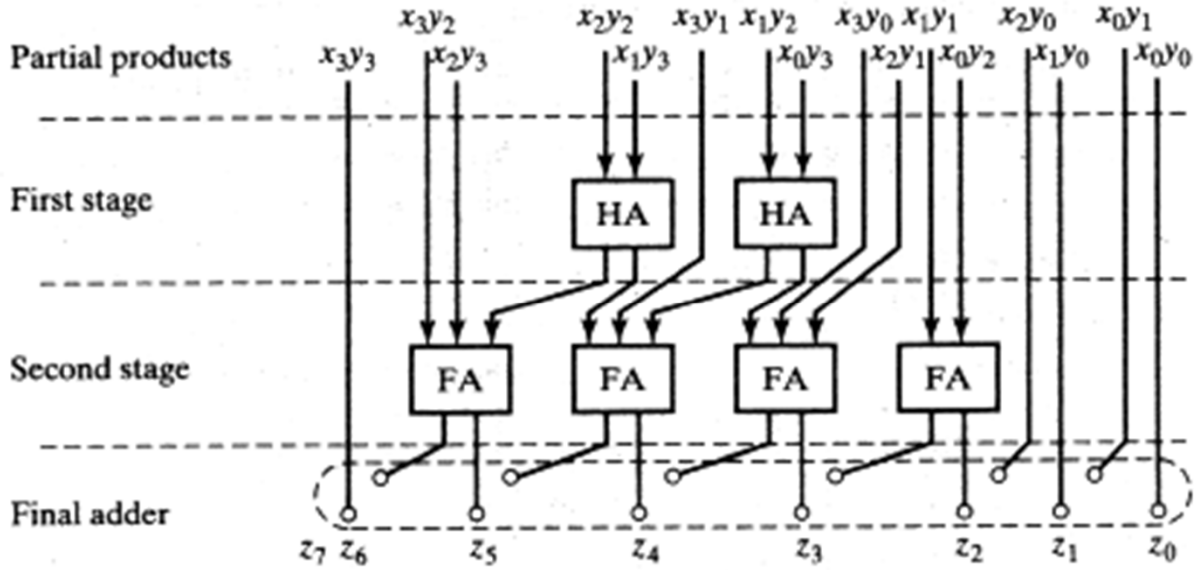


Figure 11-35 Wallace tree for four-bit multiplier.

*Digital Integrated Circuits
J Rabaey, A Chandrakasan, B Nikolic

Multiplication by a constant

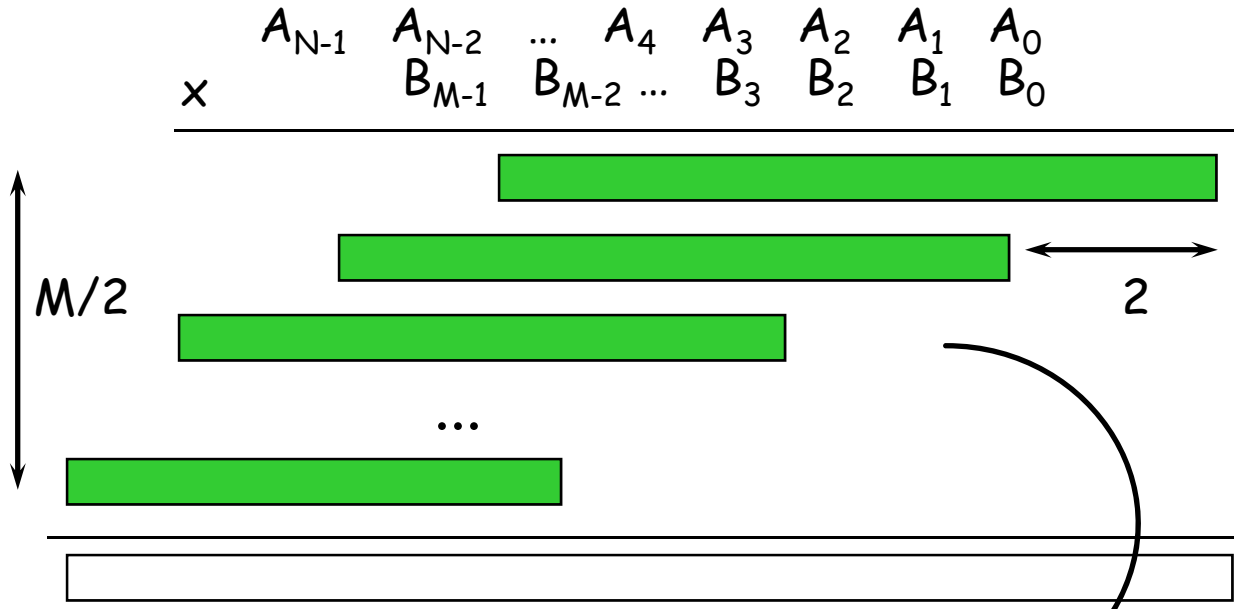
- If one of the operands is a constant, make it the multiplier (B in the earlier examples). For each "1" bit in the constant we get a partial product (PP) – may be noticeably fewer PPs than in the general case.
  - For example, in general multiplying two 4-bit operands generates four PPs (3 rows of full 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.

$$\ldots 011110\ldots = \ldots 100000\ldots - \ldots 000010\ldots = \ldots 010001\overline{0}\ldots$$

where $\overline{1}$ 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!



$B_{K+1,K}*A = 0*A \rightarrow 0$
$= 1*A \rightarrow A$
$= 2*A \rightarrow 4A - 2A$
$= 3*A \rightarrow 4A - A$

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 $\mathcal{O}(N)$ to $\mathcal{O}(\log_2(N))$
- 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)

Lab 4 Car Alarm - Design Approach

- Read lab/specifications carefully, use reasonable interpretation
- Use modular design – don't put everything into labkit.v
- Design the FSM!
  - Define the inputs
  - Define the outputs
  - Transition rules
- Logical modules:
  - fsm.v
  - timer.v // the hardest module!!
  - siren.v
  - fuel_pump.v
- Run simulation on each module!
- Use hex display: show state and time
- Use logic analyzer in Vivado

Car Alarm – Inputs & Outputs

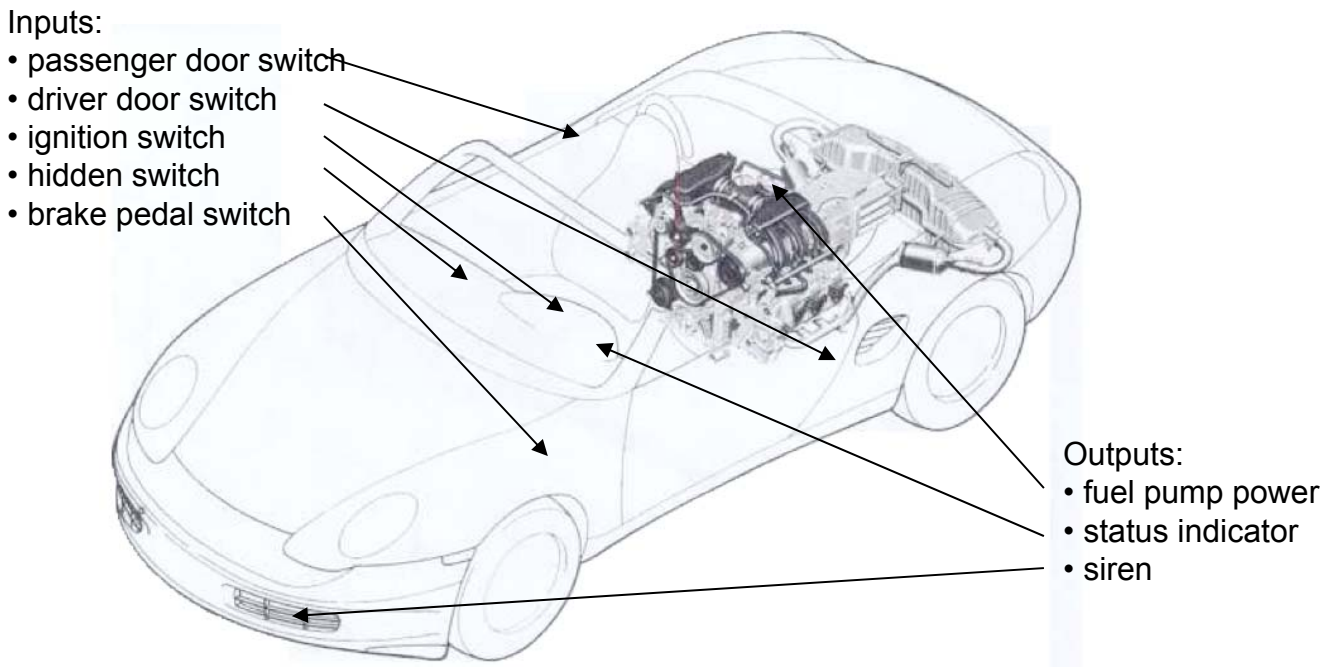
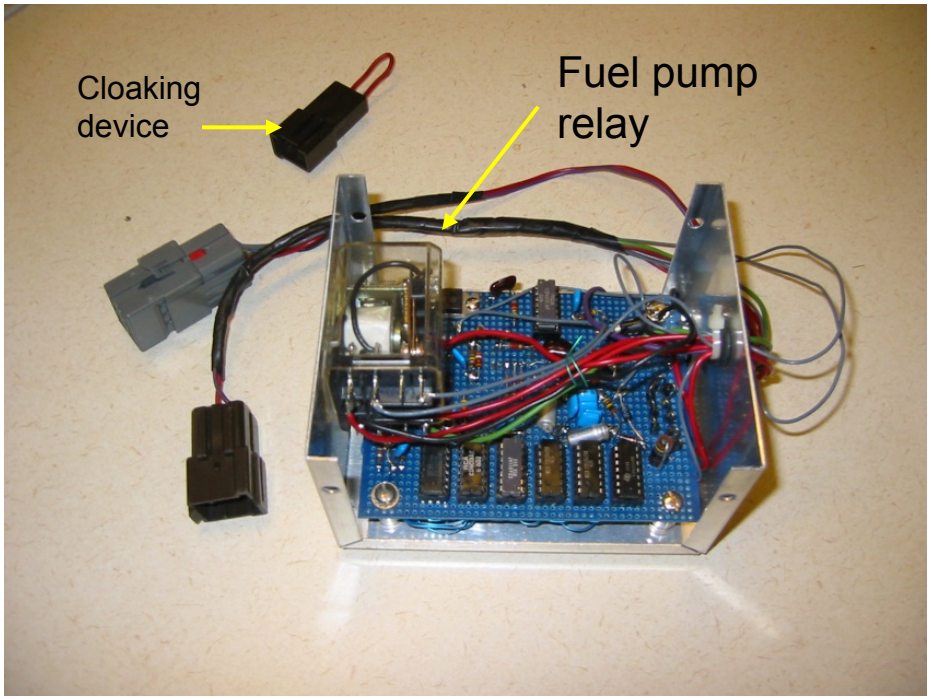


Figure 1: System diagram showing sensors (inputs) and actuators (outputs)

Car Alarm – CMOS Implementation



- Design Specs
  - Operating voltage 8-18VDC
  - Operating temp: -10C +65C
  - Attitude: sea level
  - Shock/Vibration
- Notes
  - Protected against 24V power surges
  - CMOS implementation
  - CMOS inputs protected against 200V noise spikes
  - On state DC current <10ma
  - Include T_PASSENGER_DELAY and Fuel Pump Disable
  - System disabled (cloaked) when being serviced.

Debugging Hints – Lab 4

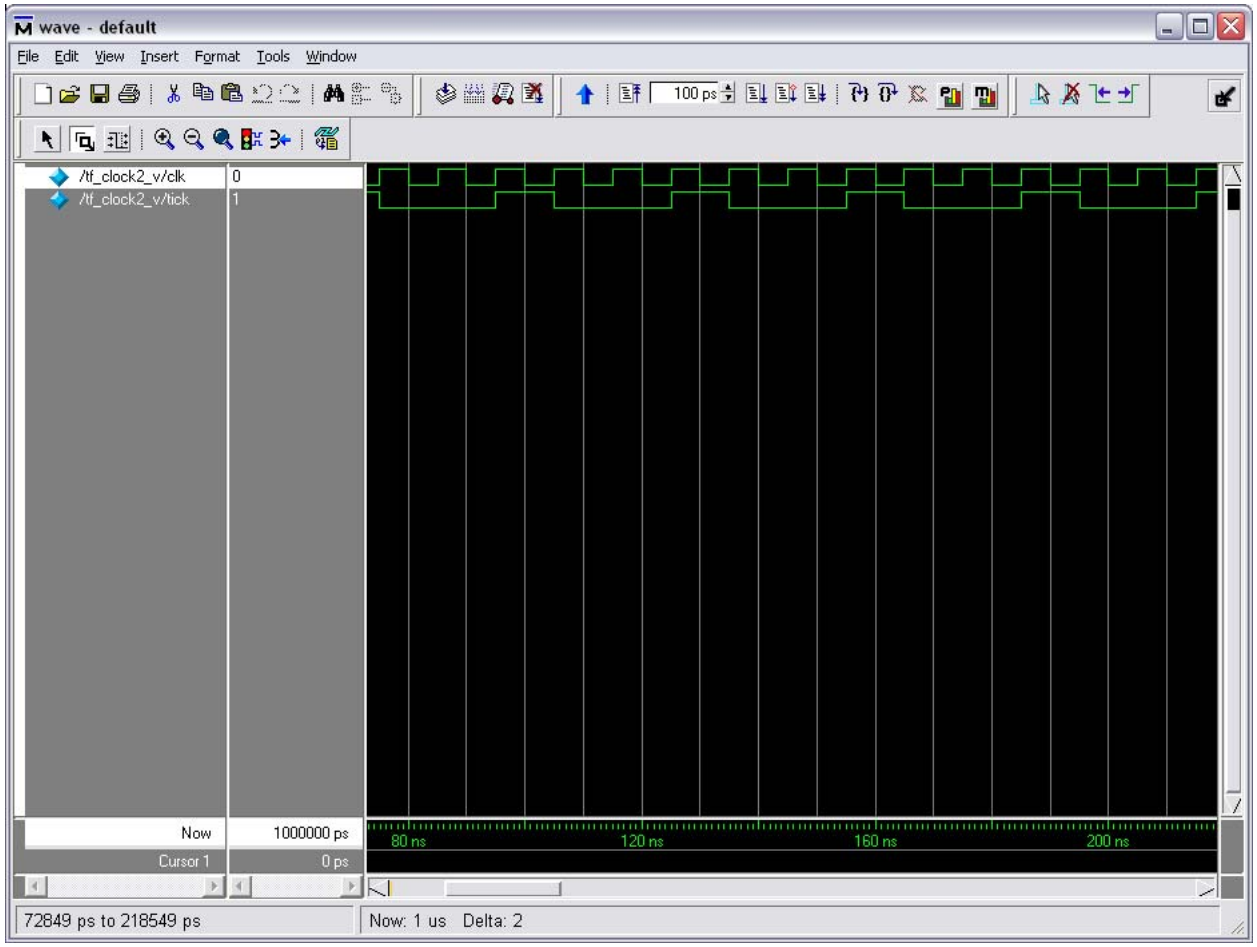
- Implement a warp speed debug mode for the one hz clock. This will allow for viewing signals on the logic analyzer or Modelsim without waiting for 27/25 million clock cycles. Avoids recomplilations.

```
assign warp_speed = sw[6];
always @ (posedge clk) begin
    if (count == (warp_speed ? 3 : 26_999_999)) count <= 0;
    else count <= count +1;
  end

assign one_hz = (count == (warp_speed ? 3 : 26_999_999)) ;
```

One Hz Ticks in Modelsim

To create a one hz tick, use the following in the Verilog test fixture:

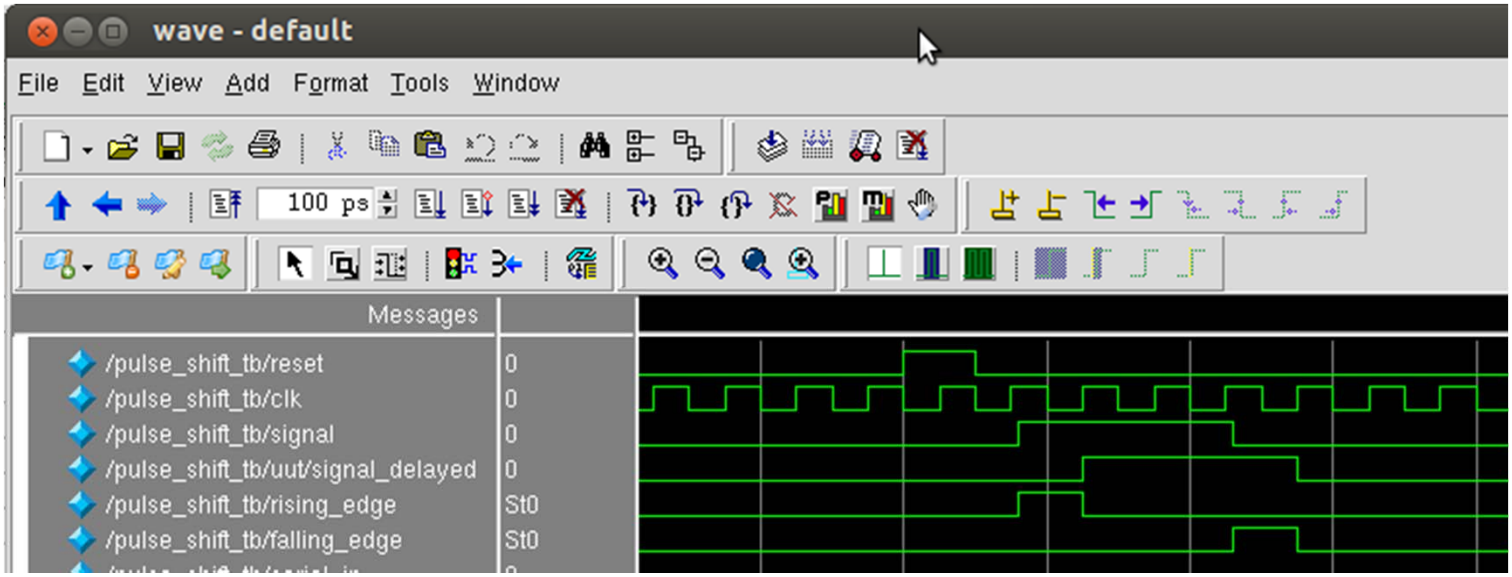
```
always #5 clk=!clk;
always  begin
      #5 tick = 1;
      #10 tick = 0;
      #15;
 end

  initial begin
      // Initialize Inputs
      clk = 0;
      tick = 0;  . . .
```



For Loops, Repeat Loops in Simulation

```
integer i;    // index must be declared as integer
integer irepeat;

// this will just wait 10ns, repeated 32x.
// simulation only! Cannot implement #10 in hardware!
        irepeat =0;
        repeat(32)  begin
        #10;
        irepeat = irepeat + 1;
        end


// this will wait #10ns before incrementing the for loop
     for (i=0; i<16; i=i+1)  begin
         #10;     // wait #10 before increment.
          // @(posedge clk);
          // add to index on posedge
        end

// other loops:  forever, while
```

Edge Detection



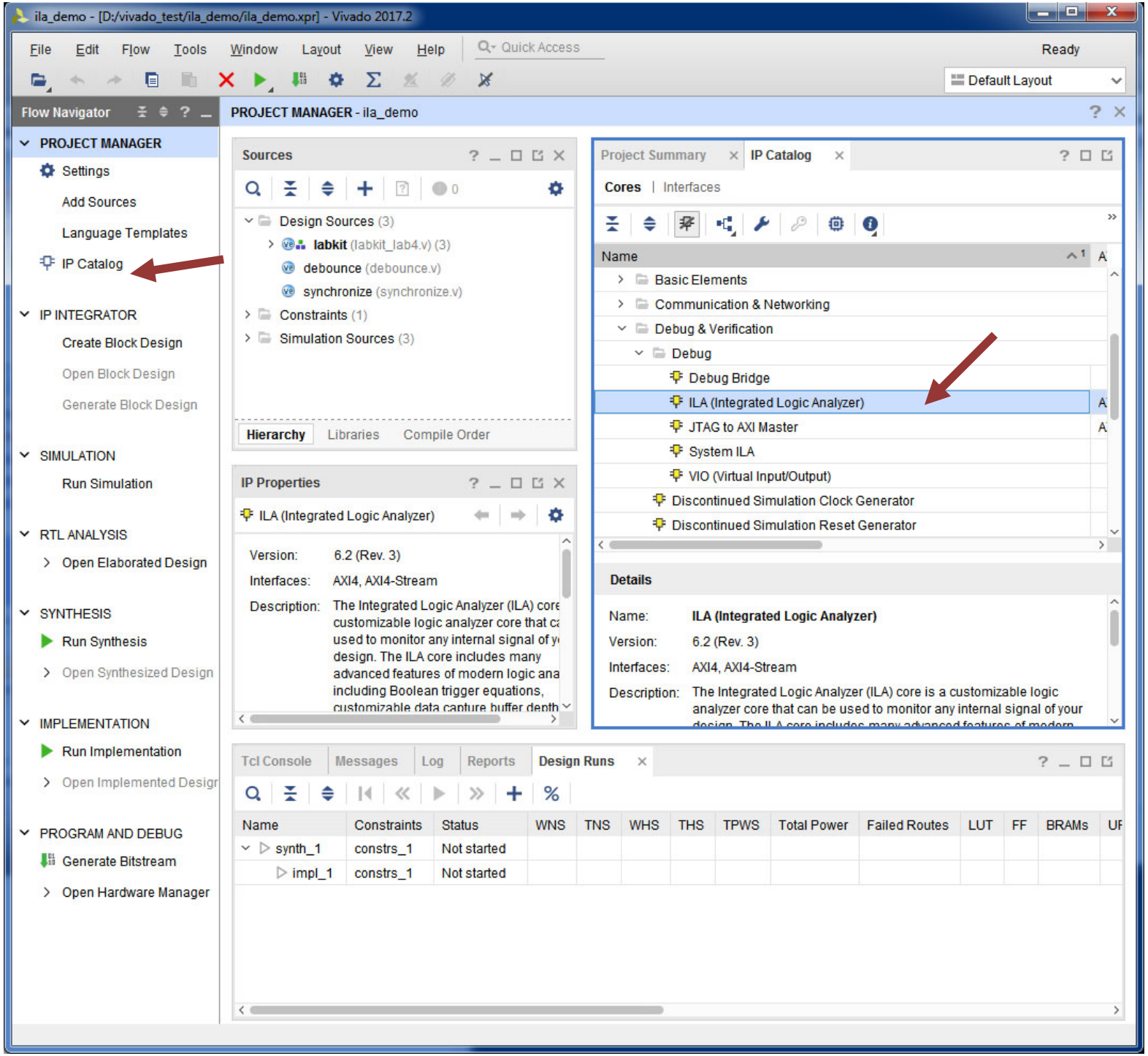

```
reg signal_delayed;

always @(posedge clk)
        signal_delayed <= signal;

assign rising_edge = signal && !signal_delayed;
assign falling_edge = !signal && signal_delayed;
```

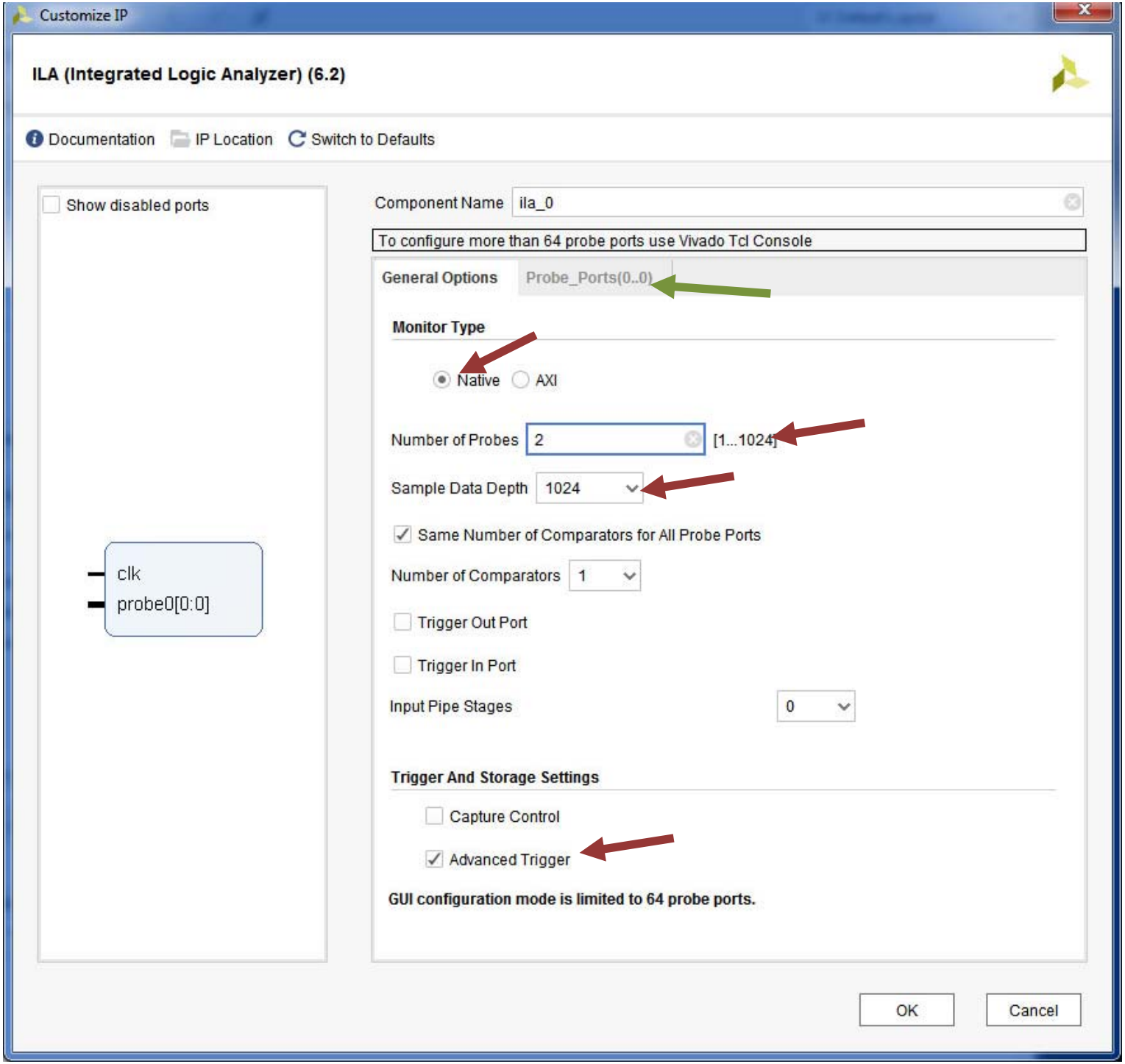
Vivado ILA

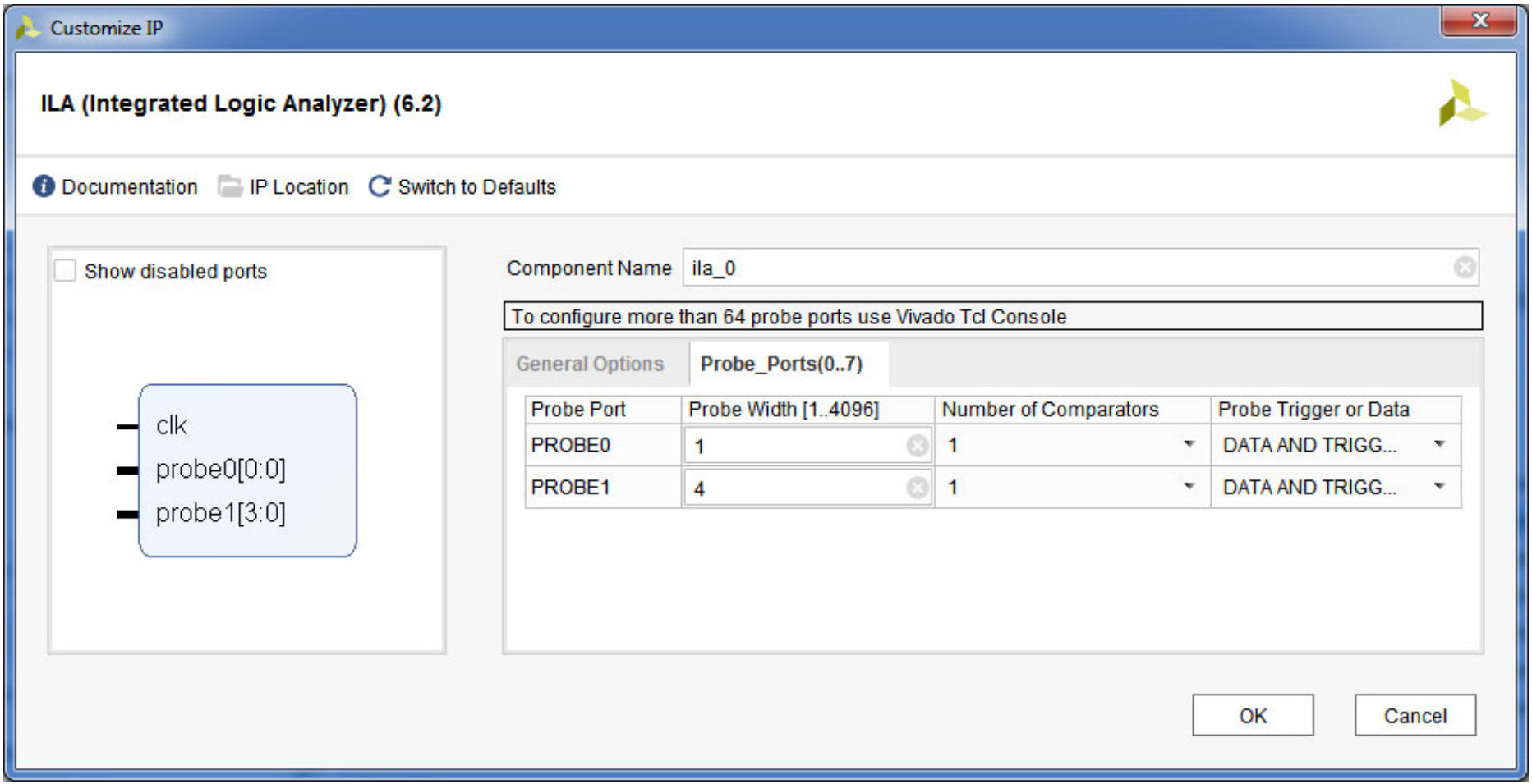
- Integrated Logic Analyzer (ILA) IP core
  - logic analyzer core that can be used to monitor the internal signals of a design
  - includes many advanced features of modern logic analyzers
    - Boolean trigger equations,
    - edge transition triggers ...
  - no physical probes to hook up!

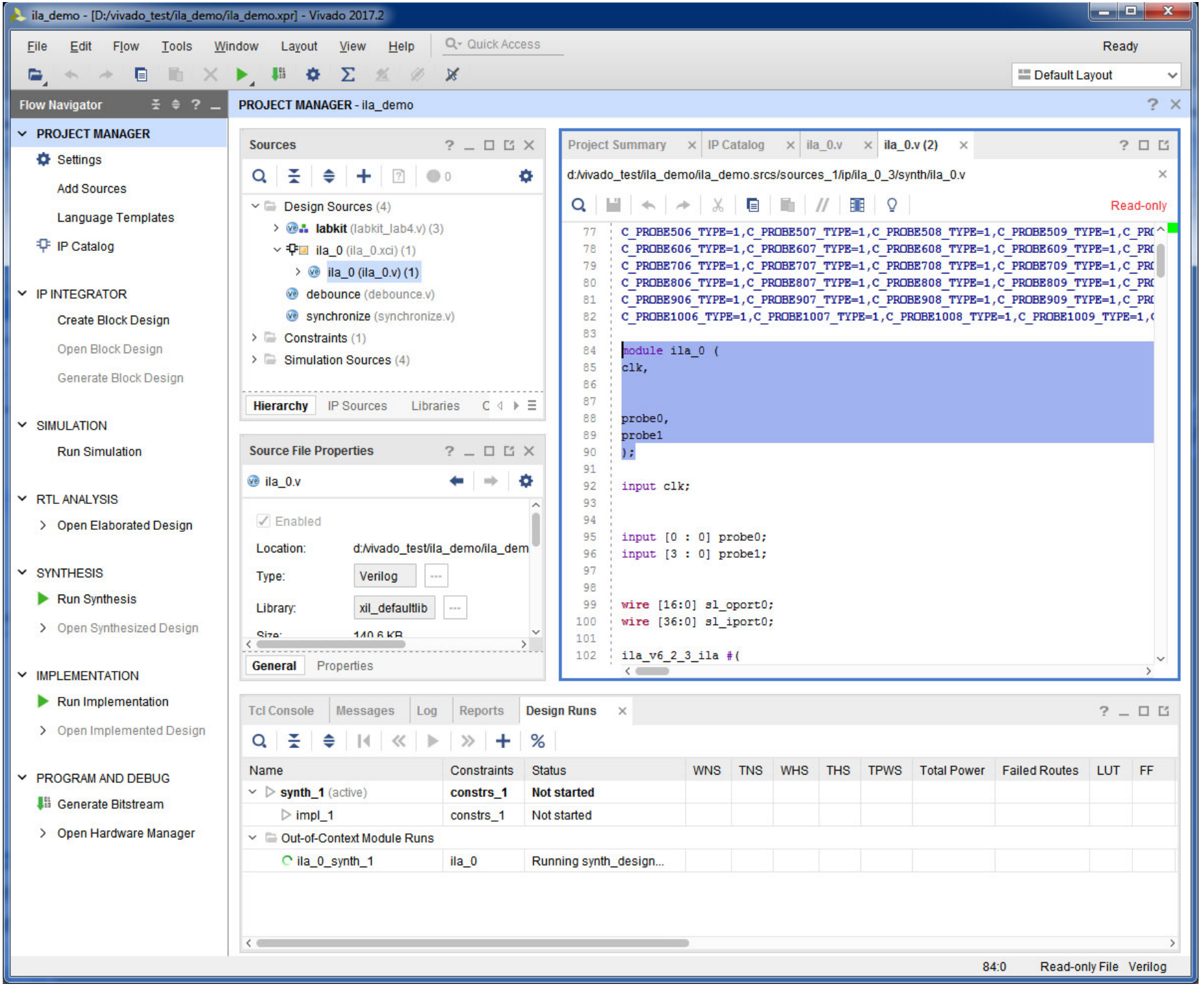
- Bit file must be loaded on target device. Not simulation.

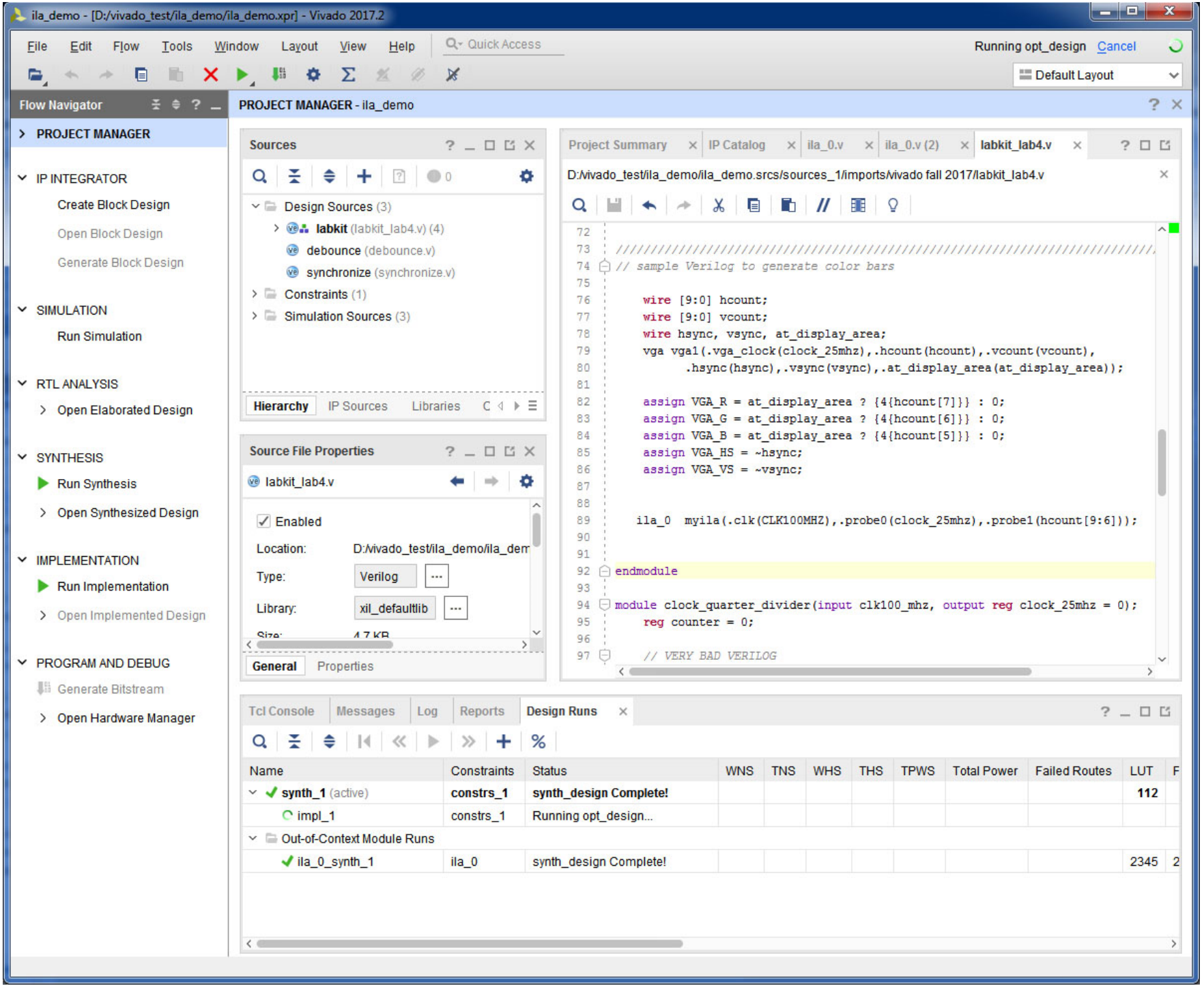
- Tutorial
  http://web.mit.edu/6.111/www/f2017/handouts/labs/ila.html

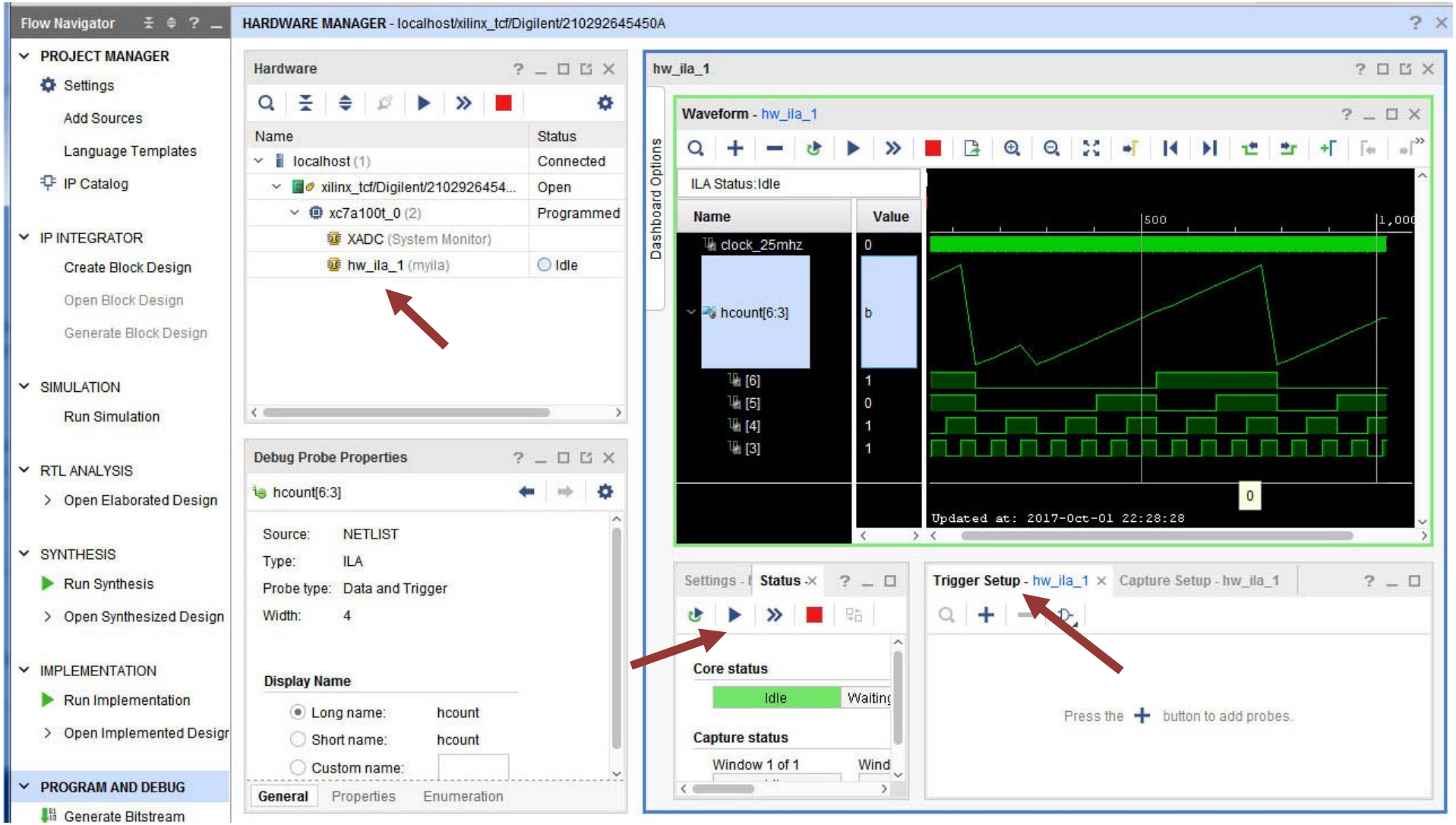

ila_demo - [D:/vivado_test/ila_demo/ila_demo.xpr] - Vivado 2017.2
File Edit Flow Tools Window Layout View Help
Quick Access
Ready
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
PROJECT MANAGER - ila_demo
Sources
Design Sources (3)
labkit (labkit_lab4.v) (3)
debounce (debounce.v)
synchronize (synchronize.v)
Constraints (1)
Simulation Sources (3)
Hierarchy Libraries Compile Order
IP Properties
ILA (Integrated Logic Analyzer)
Version: 6.2 (Rev. 3)
Interfaces: AXI4, AXI4-Stream
Description: The Integrated Logic Analyzer (ILA) core
customizable logic analyzer core that ca
used to monitor any internal signal of y
design. The ILA core includes many
advanced features of modern logic ana
including Boolean trigger equations,
customizable data capture buffer depth
Project Summary
IP Catalog
Cores | Interfaces
Name
Basic Elements
Communication & Networking
Debug & Verification
Debug
Debug Bridge
ILA (Integrated Logic Analyzer)
JTAG to AXI Master
System ILA
VIO (Virtual Input/Output)
Discontinued Simulation Clock Generator
Discontinued Simulation Reset Generator
Details
Name: ILA (Integrated Logic Analyzer)
Version: 6.2 (Rev. 3)
Interfaces: AXI4, AXI4-Stream
Description: The Integrated Logic Analyzer (ILA) core is a customizable logic analyzer core that can be used to monitor any internal signal of your
Tcl Console Messages Log Reports Design Runs
Name Constraints Status WNS TNS WHS THS TPWS Total Power Failed Routes LUT FF BRAMs
synth_1 constrs_1 Not started
impl_1 constrs_1 Not started


Customize IP
ILA (Integrated Logic Analyzer) (6.2)
Documentation
IP Location
Switch to Defaults
Show disabled ports
clk
probe0[0:0]
Component Name
ila_0
To configure more than 64 probe ports use Vivado Tcl Console
General Options
Probe_Ports(0..0)
Monitor Type
Native
AXI
Number of Probes
2
[1...1024]
Sample Data Depth
1024
Same Number of Comparators for All Probe Ports
Number of Comparators
1
Trigger Out Port
Trigger In Port
Input Pipe Stages
0
Trigger And Storage Settings
Capture Control
Advanced Trigger
GUI configuration mode is limited to 64 probe ports.
OK
Cancel


Customize IP
ILA (Integrated Logic Analyzer) (6.2)
Documentation
IP Location
Switch to Defaults
Show disabled ports
clk
probe0[0:0]
probe1[3:0]
Component Name
ila_0
To configure more than 64 probe ports use Vivado Tcl Console
General Options
Probe_Ports(0..7)
Probe Port
Probe Width [1..4096]
Number of Comparators
Probe Trigger or Data
PROBE0
1
1
DATA AND TRIGG...
PROBE1
4
1
DATA AND TRIGG...
OK
Cancel


ila_demo - [D:/vivado_test/ila_demo/ila_demo.xpr] - Vivado 2017.2
File Edit Flow Tools Window Layout View Help
Quick Access
Ready
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
PROJECT MANAGER - ila_demo
Sources
Design Sources (4)
labkit (labkit_lab4.v) (3)
ila_0 (ila_0.xci) (1)
ila_0 (ila_0.v) (1)
debounce (debounce.v)
synchronize (synchronize.v)
Constraints (1)
Simulation Sources (4)
Hierarchy IP Sources Libraries
Source File Properties
ila_0.v
Enabled
Location: d:/vivado_test/ila_demo/ila_dem
Type: Verilog
Library: xil_defaultlib
General Properties
Project Summary
IP Catalog
ila_0.v
ila_0.v (2)
d:/vivado_test/ila_demo/ila_demo.srcs/sources_1/ip/ila_0_3/synth/ila_0.v
Read-only
77 C_PROBE506_TYPE=1,C_PROBE507_TYPE=1,C_PROBE508_TYPE=1,C_PROBE509_TYPE=1,C_PR
78 C_PROBE606_TYPE=1,C_PROBE607_TYPE=1,C_PROBE608_TYPE=1,C_PROBE609_TYPE=1,C_PR
79 C_PROBE706_TYPE=1,C_PROBE707_TYPE=1,C_PROBE708_TYPE=1,C_PROBE709_TYPE=1,C_PR
80 C_PROBE806_TYPE=1,C_PROBE807_TYPE=1,C_PROBE808_TYPE=1,C_PROBE809_TYPE=1,C_PR
81 C_PROBE906_TYPE=1,C_PROBE907_TYPE=1,C_PROBE908_TYPE=1,C_PROBE909_TYPE=1,C_PR
82 C_PROBE1006_TYPE=1,C_PROBE1007_TYPE=1,C_PROBE1008_TYPE=1,C_PROBE1009_TYPE=1,
83
84 module ila_0 (
85 clk,
86
87
88 probe0,
89 probe1
90);
91
92 input clk;
93
94
95 input [0 : 0] probe0;
96 input [3 : 0] probe1;
97
98
99 wire [16:0] sl_oport0;
100 wire [36:0] sl_iport0;
101
102 ila_v6_2_3_ila #(
Tcl Console Messages Log Reports Design Runs
Name Constraints Status WNS TNS WHS THS TPWS Total Power Failed Routes LUT FF
synth_1 (active) constrs_1 Not started
impl_1 constrs_1 Not started
Out-of-Context Module Runs
ila_0_synth_1 ila_0 Running synth_design...
84:0 Read-only File Verilog


ila_demo - [D:/vivado_test/ila_demo/ila_demo.xpr] - Vivado 2017.2
File Edit Flow Tools Window Layout View Help
Quick Access
Running opt_design Cancel
Default Layout
Flow Navigator
PROJECT MANAGER
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
PROJECT MANAGER - ila_demo
Sources
Design Sources (3)
labkit (labkit_lab4.v) (4)
debounce (debounce.v)
synchronize (synchronize.v)
Constraints (1)
Simulation Sources (3)
Hierarchy IP Sources Libraries
Source File Properties
labkit_lab4.v
Enabled
Location: D:/vivado_test/ila_dem
Type: Verilog
Library: xil_defaultlib
General Properties
Project Summary IP Catalog ila_0.v ila_0.v (2) labkit_lab4.v
D:/vivado_test/ila_demo/ila_demo.srcs/sources_1/imports/vivado fall 2017/labkit_lab4.v
72
73 //
74 // sample Verilog to generate color bars
75
76 wire [9:0] hcount;
77 wire [9:0] vcount;
78 wire hsync, vsync, at_display_area;
79 vga vga1(.vga_clock(clock_25mhz),.hcount(hcount),.vcount(vcount),
80 .hsync(hsync),.vsync(vsync),.at_display_area(at_display_area));
81
82 assign VGA_R = at_display_area ? {4{hcount[7]}} : 0;
83 assign VGA_G = at_display_area ? {4{hcount[6]}} : 0;
84 assign VGA_B = at_display_area ? {4{hcount[5]}} : 0;
85 assign VGA_HS = ~hsync;
86 assign VGA_VS = ~vsync;
87
88
89 ila_0 myila(.clk(CLK100MHZ),.probe0(clock_25mhz),.probe1(hcount[9:6]));
90
91
92 endmodule
93
94 module clock_quarter_divider(input clk100_mhz, output reg clock_25mhz = 0);
95 reg counter = 0;
96
97 // VERY BAD VERILOG
Tcl Console Messages Log Reports Design Runs
Name Constraints Status WNS TNS WHS THS TPWS Total Power Failed Routes LUT
synth_1 (active) constrs_1 synth_design Complete! 112
impl_1 constrs_1 Running opt_design...
Out-of-Context Module Runs
ila_0_synth_1 ila_0 synth_design Complete! 2345


Flow Navigator
HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210292645450A
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/2102926454...
Open
xc7a100t_0 (2)
Programmed
XADC (System Monitor)
hw_ila_1 (myila)
Idle
Debug Probe Properties
hcount[6:3]
Source: NETLIST
Type: ILA
Probe type: Data and Trigger
Width: 4
Display Name
Long name: hcount
Short name: hcount
Custom name:
General
Properties
Enumeration
hw_ila_1
Dashboard Options
Waveform - hw_ila_1
ILA Status: Idle
Name
Value
clock_25mhz
0
hcount[6:3]
b
[6]
1
[5]
0
[4]
1
[3]
1
500
1,000
0
Updated at: 2017-Oct-01 22:28:28
Settings - h
Status
Core status
Idle
Waiting
Capture status
Window 1 of 1
Wind
Trigger Setup - hw_ila_1
Capture Setup - hw_ila_1
Press the + button to add probes.

Student Comments

- "All very reasonable except for lab 4, Car Alarm. Total pain in the ass. "
- "The labs were incredibly useful, interesting, and helpful for learning. Lab 4 (car alarm) is long and difficult, but overall the labs are not unreasonable."