Arithmetic Circuits & Multipliers

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

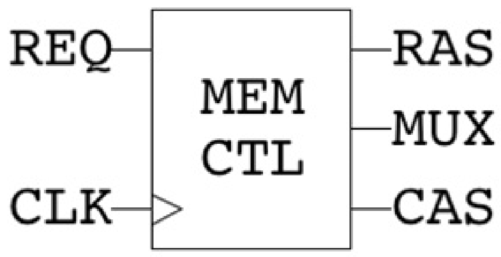
Handouts

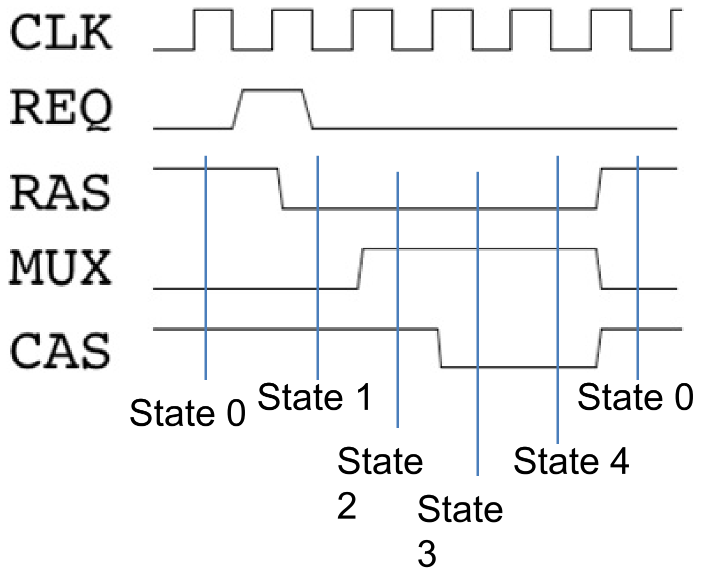
- lecture slides,

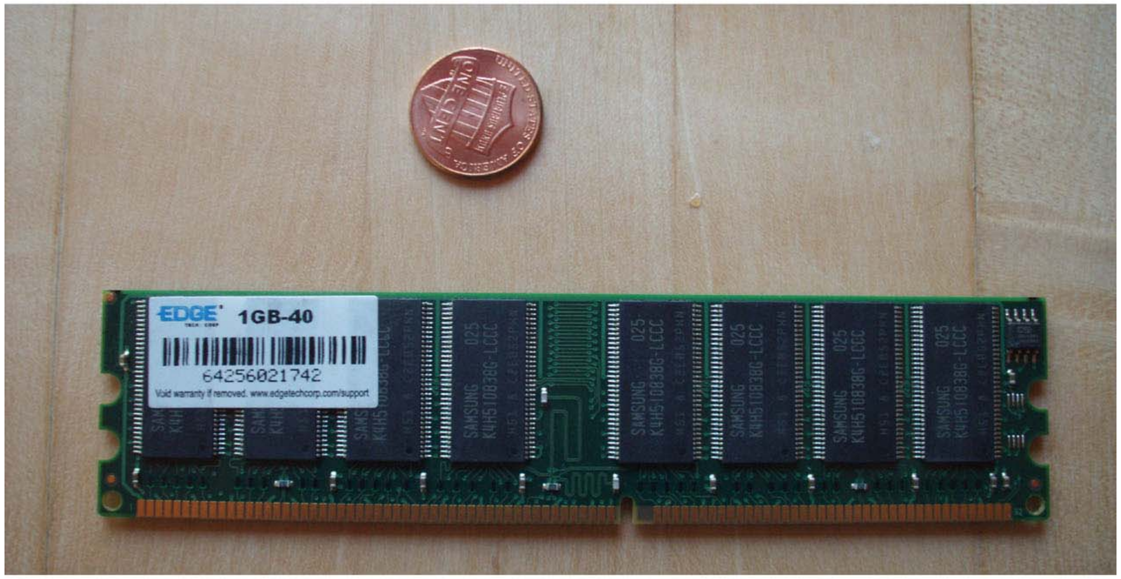
Reminder: Lab #3 due Tue/Wed
Pizza ~~Wed~~ 6p. Thu 6p

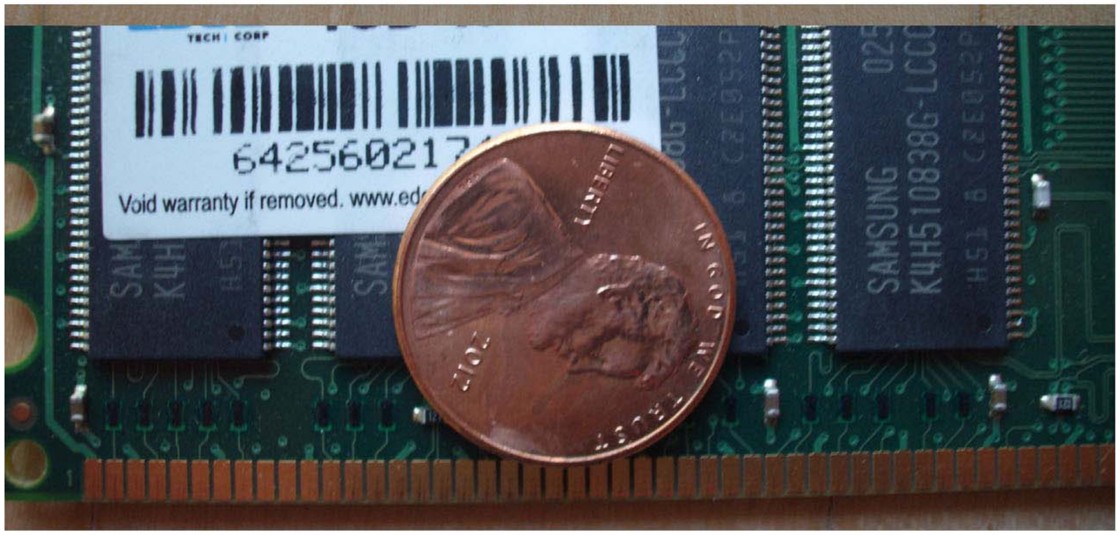
Sign up for Lab 3 Checkoff

Memory Controller

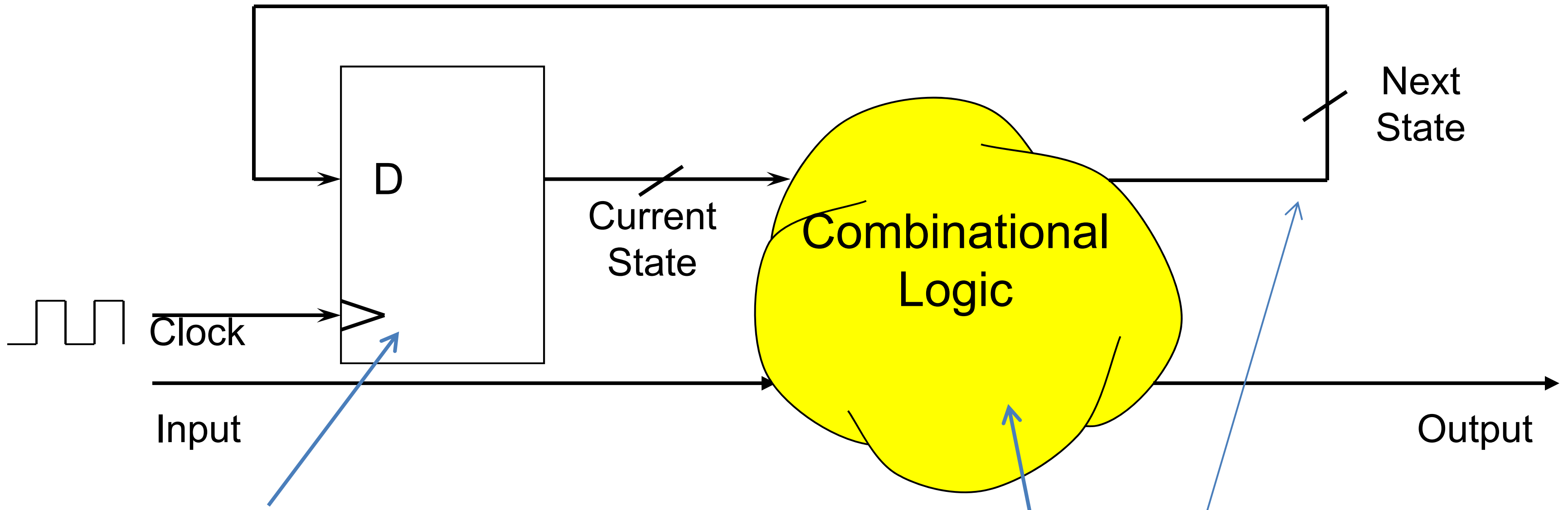





EDGE
1GB-40
64256021742
Void warranty if removed. www.edgetechcorp.com/support
SAMSUNG
K4H510838G-LCCC


TECH | CORP
642560217
Void warranty if removed. www.ed
SAMSUNG
K4H510838G-LCCC
LIBERTY
IN GOD WE TRUST
2012

FSM

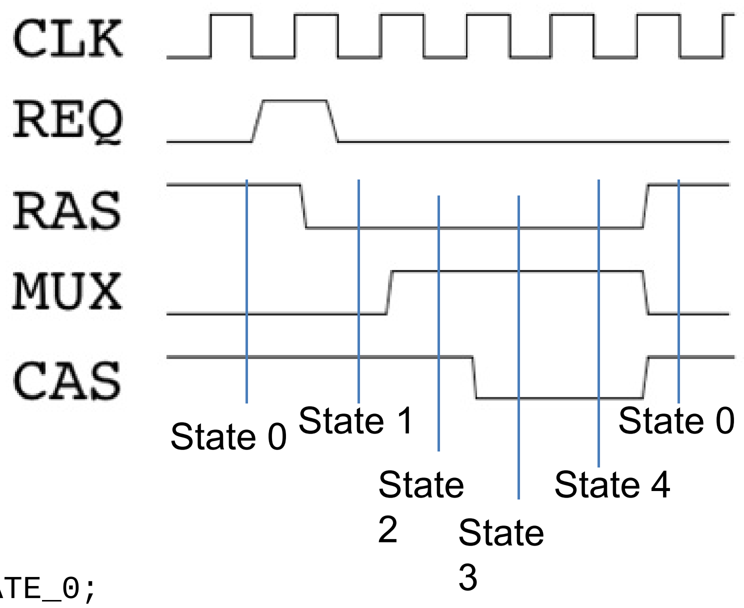


```
always_ff @(posedge clock)
   state <= next_state;
```

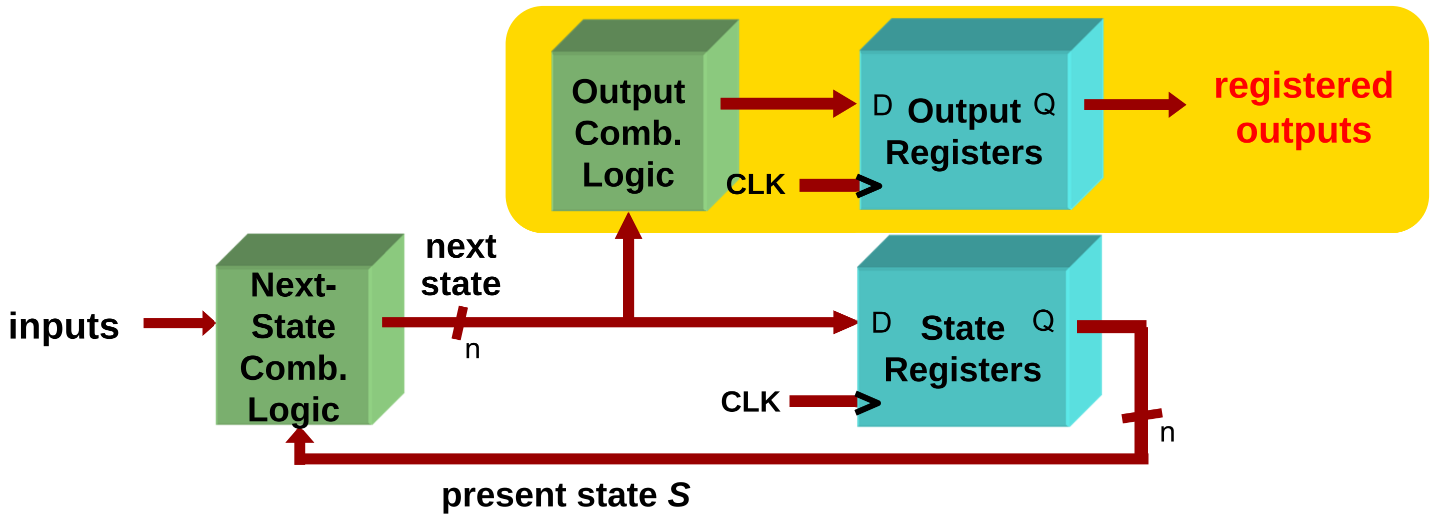
```
always_comb @
  begin // logic to determine next_state
     case (state)
    state_1:  next_state = . . .
    state_2:  next_state = . . .
    .  .  .
         default: next_state = STATE_0;
     endcase
  end
```

Glitchy Solution

```
module (
  input req, clk,
  output reg ras, mux, cas
  );

logic [3:0] state, next_state:

parameter [3:0] STATE_0 = 0;  // 0000
parameter [3:0] STATE_1 = 1;  // 0001
parameter [3:0] STATE_2 = 2;  // 0010
parameter [3:0] STATE_3 = 3;  // 0011
parameter [3:0] STATE_4 = 4;  // 0100

always_ff @(posedge clk)  state <= next_state;

always_comb begin
  case (state)
     STATE_0: next_state = req ? STATE_1 : STATE_0;
     STATE_1: next_state = STATE_2;
     STATE_2: next_state = STATE_3;
     STATE_3: next_state = STATE_4;
     STATE_4: next_state = STATE_0;
     default: next_state = state_0;
  endcase
end

assign ras = !((state==STATE_1)||(state==STATE_2)||(state==STATE_3)||(state==STATE_4));
assign mux = (state==STATE_2)||(state==STATE_3)||(state==STATE_4);
assign cas = !((state==STATE_3)||(state==STATE_4));

endmodule
```



Registered FSM Outputs are Glitch-Free

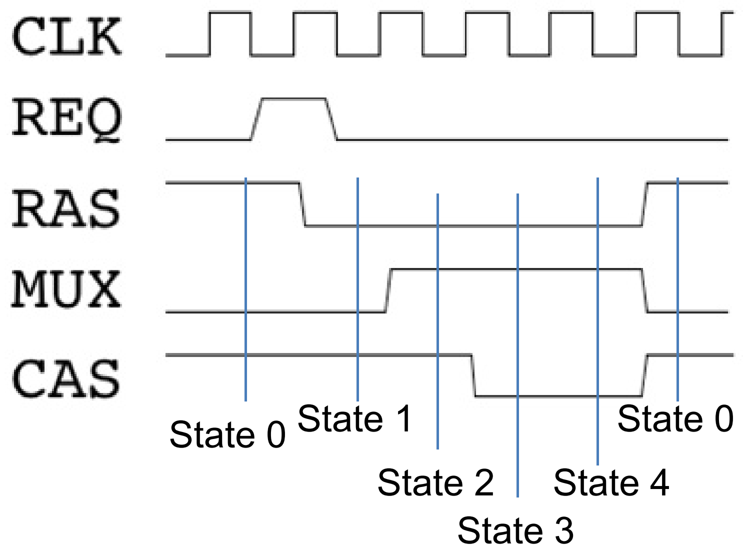


- Move output generation into the sequential always block
- Calculate outputs based on <u>next</u> state
- Delays outputs by one clock cycle. Problematic in some application.

```
reg DC,DN,DD;

 // Sequential always block for state assignment
 always_ff @ (posedge clk or negedge reset) begin
   if (!reset)   state <= IDLE;
   else if (clk) state <= next;

   DC <= (next == GOT_30c || next == GOT_35c ||
          next == GOT_40c || next == GOT_45c ||
          next == GOT_50c);
   DN <= (next == RETURN_5c);
   DD <= (next == RETURN_20c || next == RETURN_15c ||
          next == RETURN_10c);
 end
```

Glitchy Solution

```
module (
  input req, clk,
  output reg ras, mux, cas
  );

logic[3:0] state, next_state:

parameter [3:0] STATE_0 = 0;  // 0000
parameter [3:0] STATE_1 = 1;  // 0001
parameter [3:0] STATE_2 = 2;  // 0010
parameter [3:0] STATE_3 = 3;  // 0011
parameter [3:0] STATE_4 = 4;  // 0100

always_ff @(posedge clk)  state <= next_state;

always_comb begin
  case (state)
     STATE_0: next_state = req ? STATE_1 : STATE_0;
     STATE_1: next_state = STATE_2;
     STATE_2: next_state = STATE_3;
     STATE_3: next_state = STATE_4;
     STATE_4: next_state = STATE_0;
     default: next_state = state_0;
  endcase
end
```



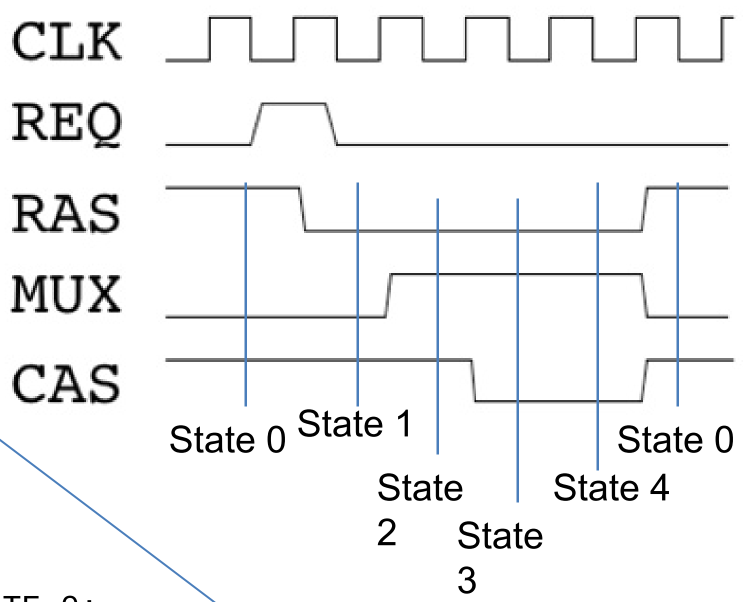
```
always_ff @(posedge clk) begin
  ras <= !((next_state==STATE_1)||(next_state2) . .
.
```

```
assign ras = !((state==STATE_1)||(state==STATE_2)||(state==STATE_3)||(state==STATE_4));
assign mux = ~~(state==STATE_2)||(state==STATE_3)||(state==STATE_4);~~
assign cas = ~~!((state==STATE_3)||(state==STATE_4));~~

endmodule
```

Another Glitch Free Solution

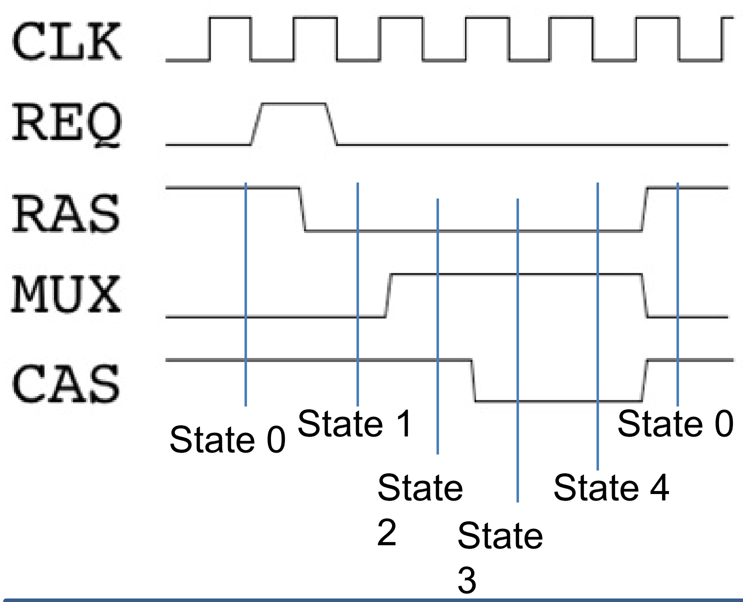
```
module (
  input req, clk,
  output reg ras, mux, cas
  );

logic [3:0] state, next_state:

parameter [3:0] STATE_0 = 4'b1010;
parameter [3:0] STATE_1 = 4'b0010;
parameter [3:0] STATE_2 = 4'b0110;
parameter [3:0] STATE_3 = 4'b0100;
parameter [3:0] STATE_4 = 4'b0101;

always_ff @(posedge clk)  state <= next_state;

always_com begin
  case (state)
     STATE_0: next_state = req ? STATE_1 : STATE_0;
     STATE_1: next_state = STATE_2;
     STATE_2: next_state = STATE_3;
     STATE_3: next_state = STATE_4;
     STATE_4: next_state = STATE_0;
     default: next_state = STATE_0;
  endcase
end

assign {ras, mux, cas}  = {state[3],state[2],state[1]};

endmodule
```

ras
mux
cas



Hint: You will need four bits for your state variable.

Alternative Verilog



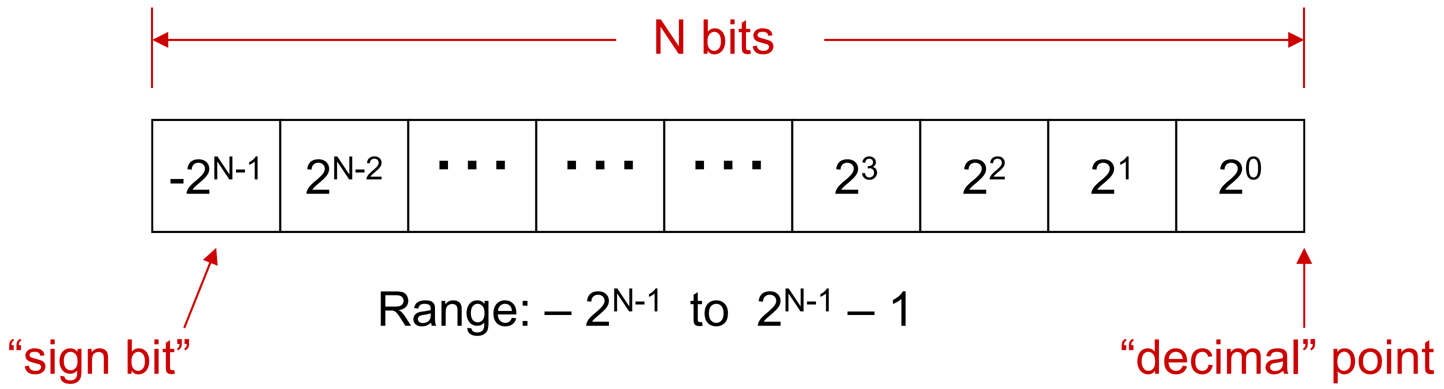
```
module (
  input req, clk,
  output reg ras, mux, cas
  );

logic [3:0] state, next_state:

parameter [3:0] STATE_0 = 4’b1010;
parameter [3:0] STATE_1 = 4’b0010;
parameter [3:0] STATE_2 = 4’b0110;
parameter [3:0] STATE_3 = 4’b0100;
parameter [3:0] STATE_4 = 4’b0101;
```

```
always_ff @(posedge clk)  state <= next_state;

always_comb begin
  case (state)
     STATE_0: next_state = req ? STATE_1 : STATE_0;
     STATE_1: next_state = STATE_2;
     STATE_2: next_state = STATE_3;
     STATE_3: next_state = STATE_4;
     STATE_4: next_state = STATE_0;
     default: next_state = STATE_0;
  endcase
end
```

```
// next_state not needed

always_ff @(posedge clk)  begin
  case (state)
     STATE_0: state <= req ? STATE_1 : STATE_0;
     STATE_1: state <= STATE_2;
     STATE_2: state <= STATE_3;
     STATE_3: state <= STATE_4;
     STATE_4: state <= STATE_0;
     default: state <= STATE_0;
  endcase
end
```

```
assign {ras, mux, cas}  = {state[3],state[2],state[1]};

endmodule
```

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
```

$$-5 = \underbrace{11111111}_{\uparrow}11111011$$

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

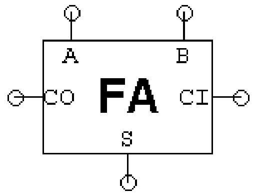
Adder: a circuit that does addition

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

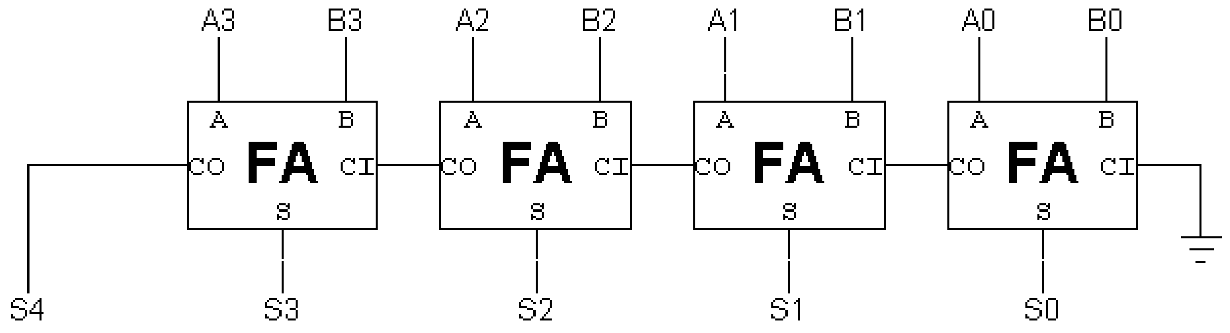
Adding two N-bit numbers produces an (N+1)-bit result

```
 1 1 0  1   ← Carries from previous column
   1101
+  0101
  -----
  10010
```

If we build a circuit that implements one column:

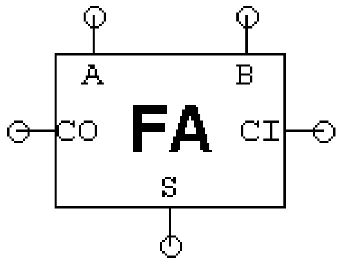


we can quickly build a circuit to add two 4-bit numbers…



"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

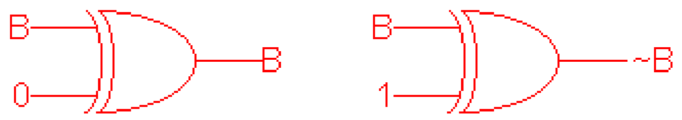
$$S = A \oplus B \oplus C$$

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

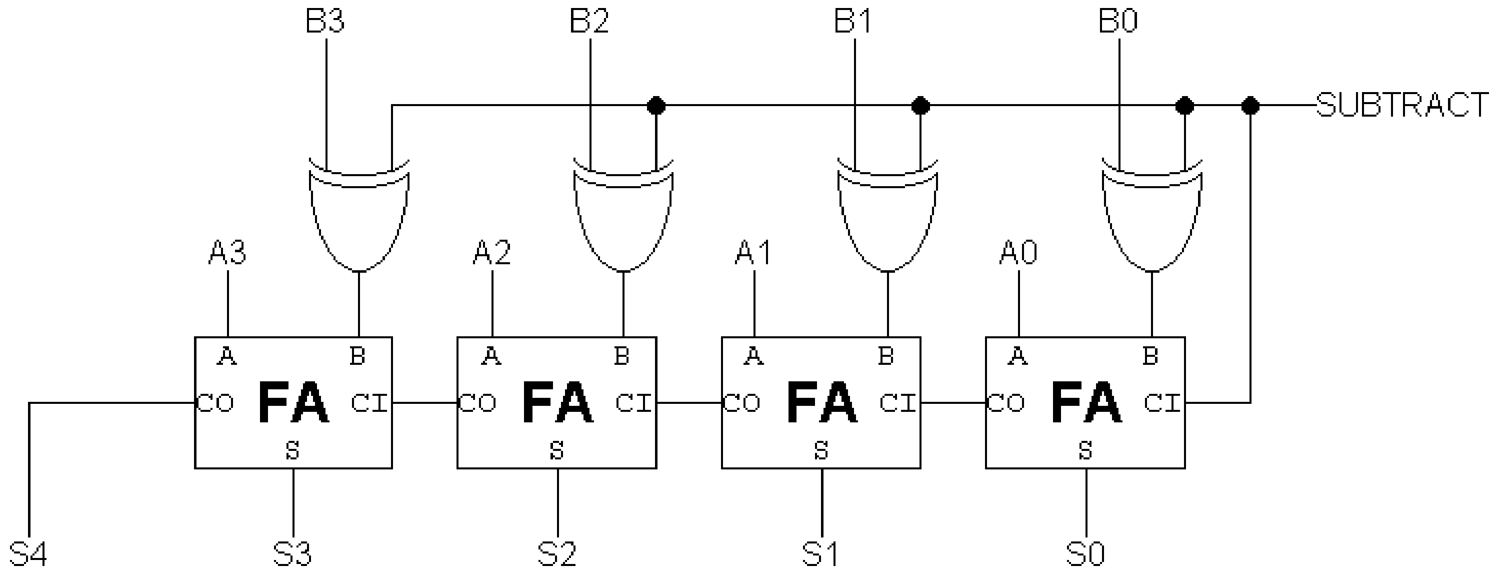
Subtraction: A-B = A + (-B)

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.

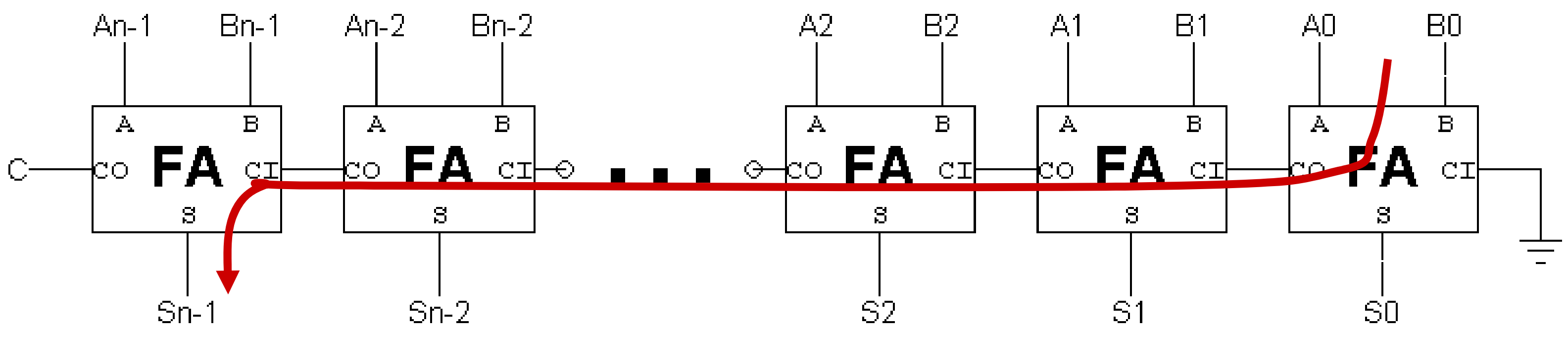
In many designs one chooses a "word size" (many computers use 32 or 64 bits) and all arithmetic results are truncated to that number of bits, i.e., arithmetic is performed modulo $2^{\text{word size}}$.

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

$C_O = AB + AC_I + BC_I$



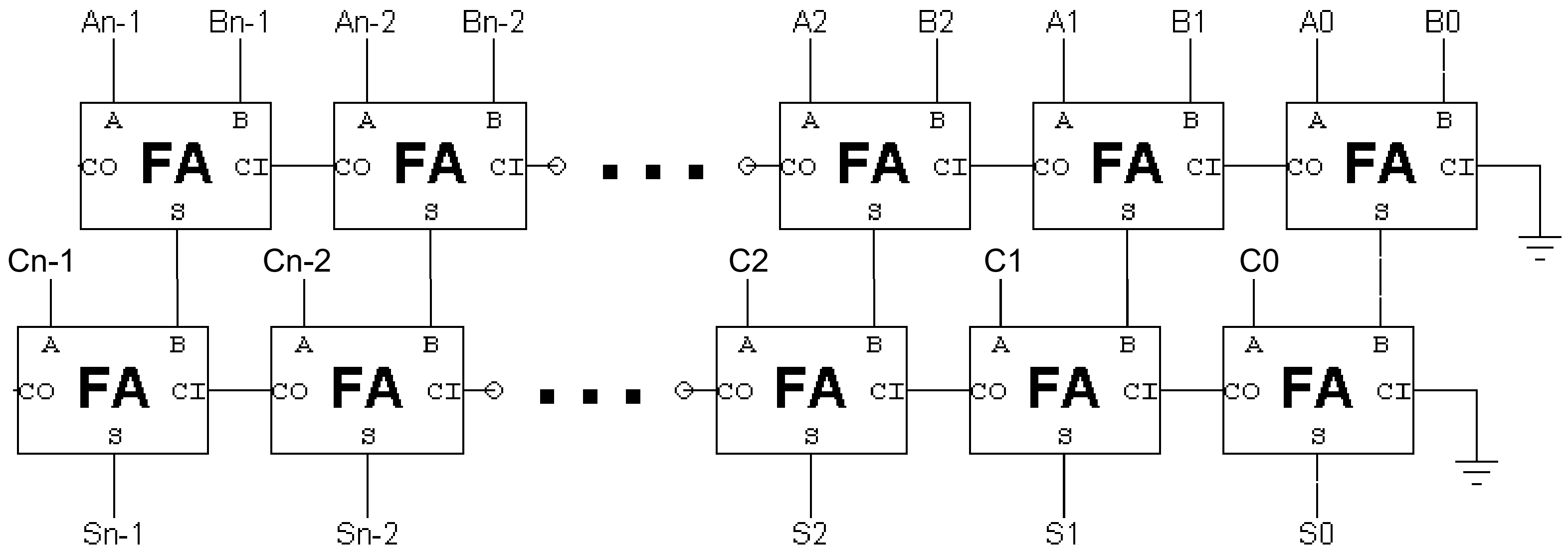
Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

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

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

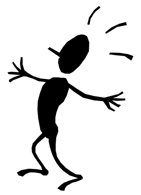
Θ(N) is read "order N" : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

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

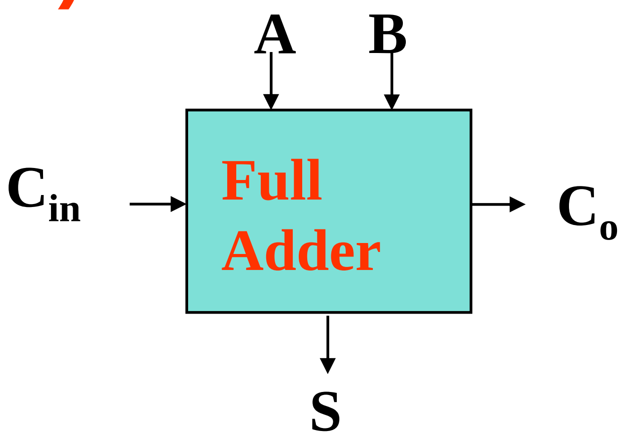


Timing analysis is tricky!

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

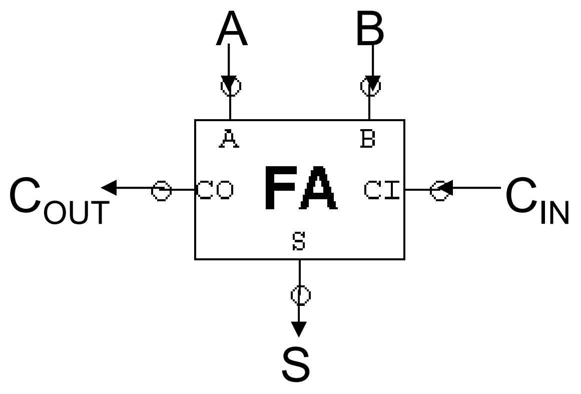
Propagate (P) = A ⊕ B

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

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}:



$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$

$$= AB + (A + B)C_{IN}$$

$$= \underset{\text{generate}}{G} + \underset{\text{propagate}}{P}\, C_{IN}$$

where $G = AB$

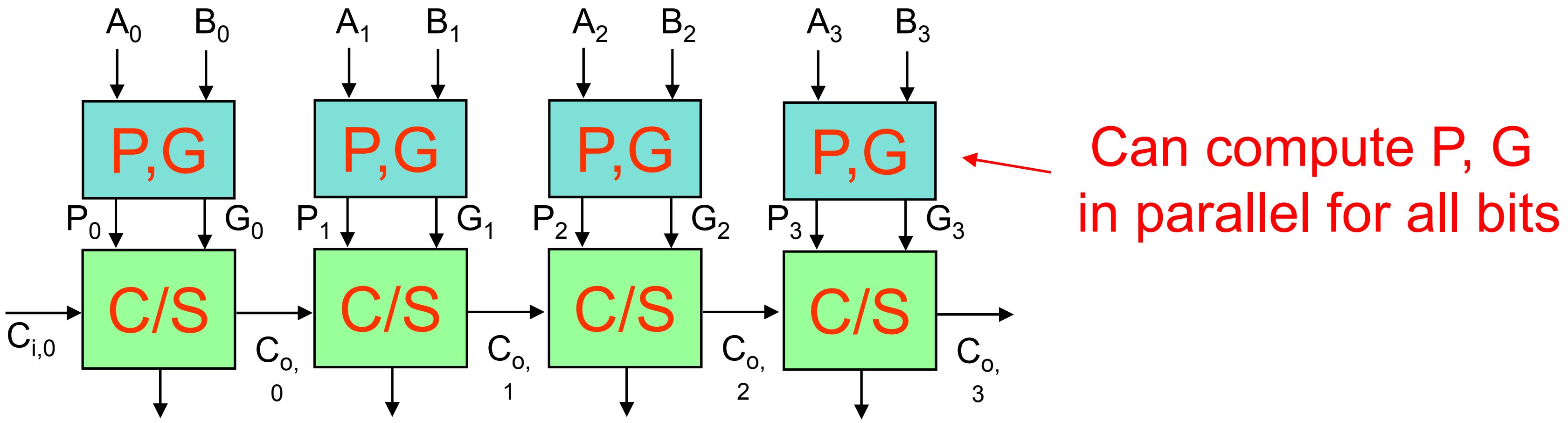
$P = A + B$

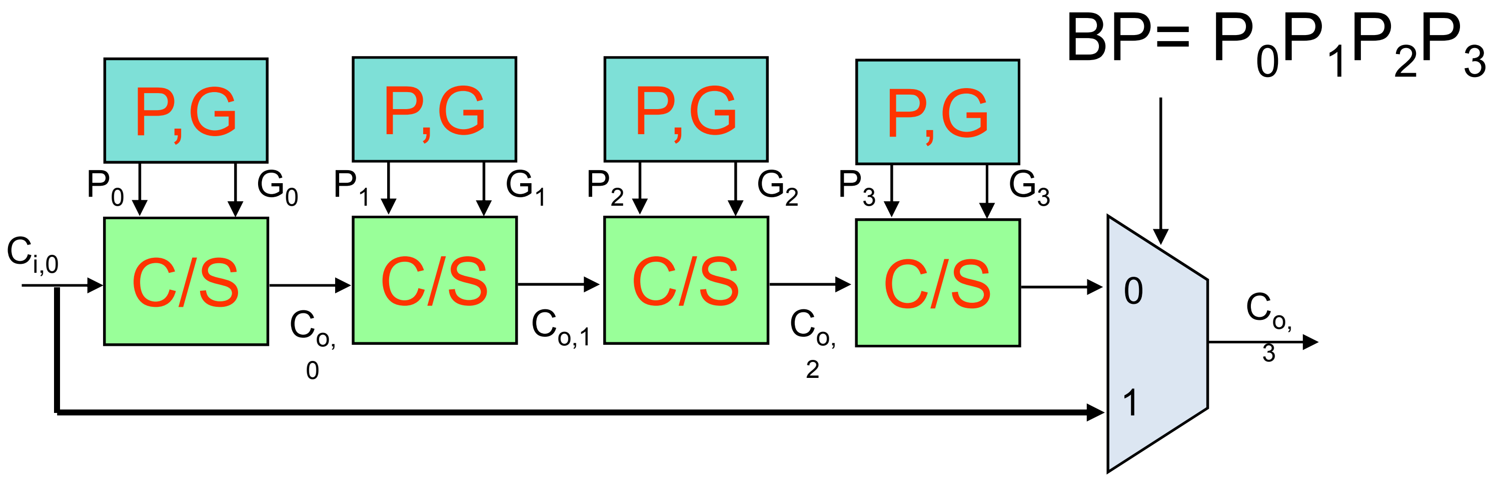
```
module fa(input a,b,cin, output s,cout);
  wire g = a & b;
  wire p = a ^ b;
  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

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 :

$S = P$^C_{IN}

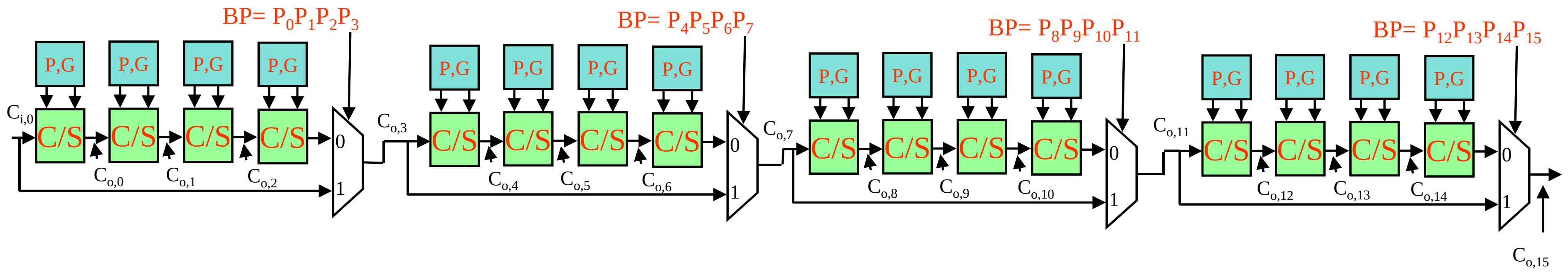
Carry Bypass Adder





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

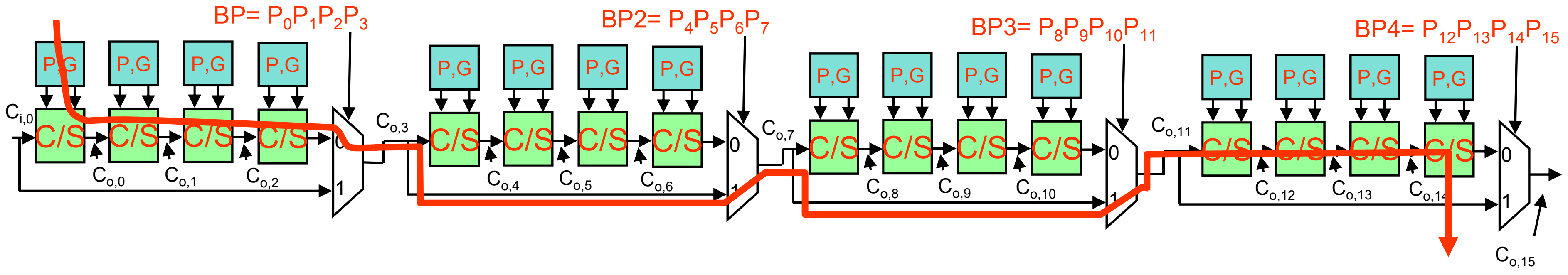
16-bit Carry Bypass Adder



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:

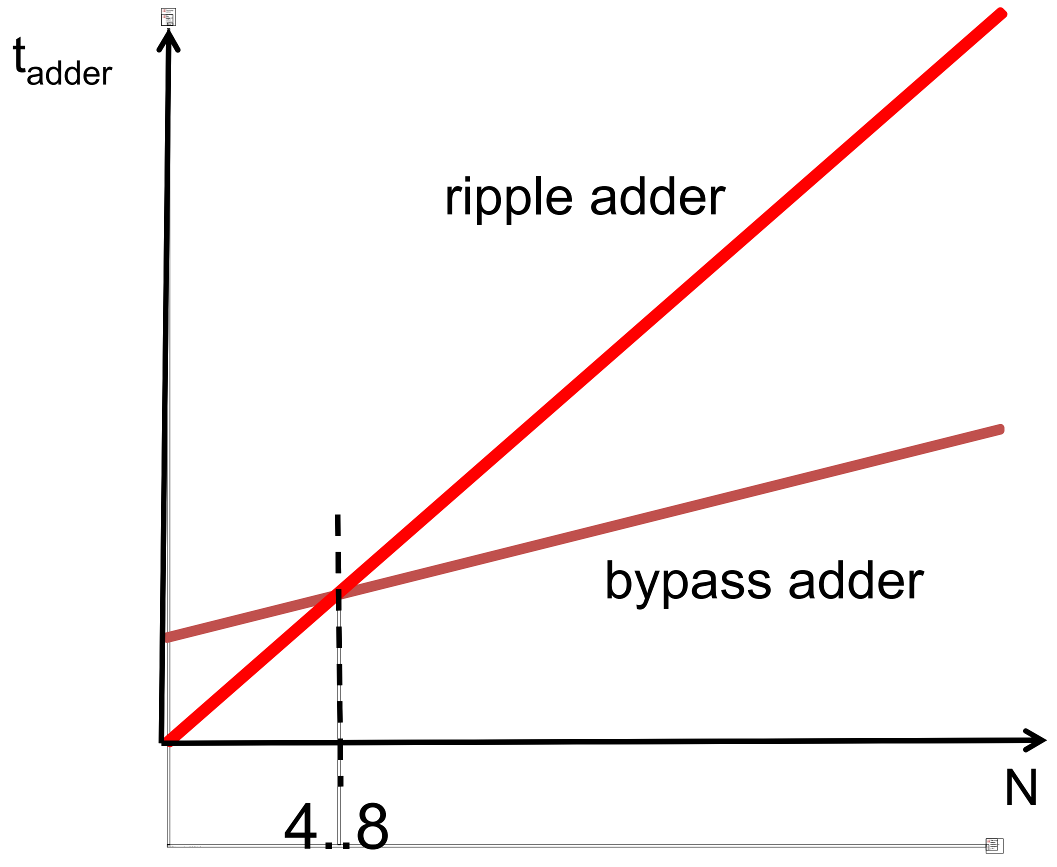
BP2 = 0 or BP2 = 1 ?

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 $P = A\hat{}B$

- For adding two N-bit numbers:

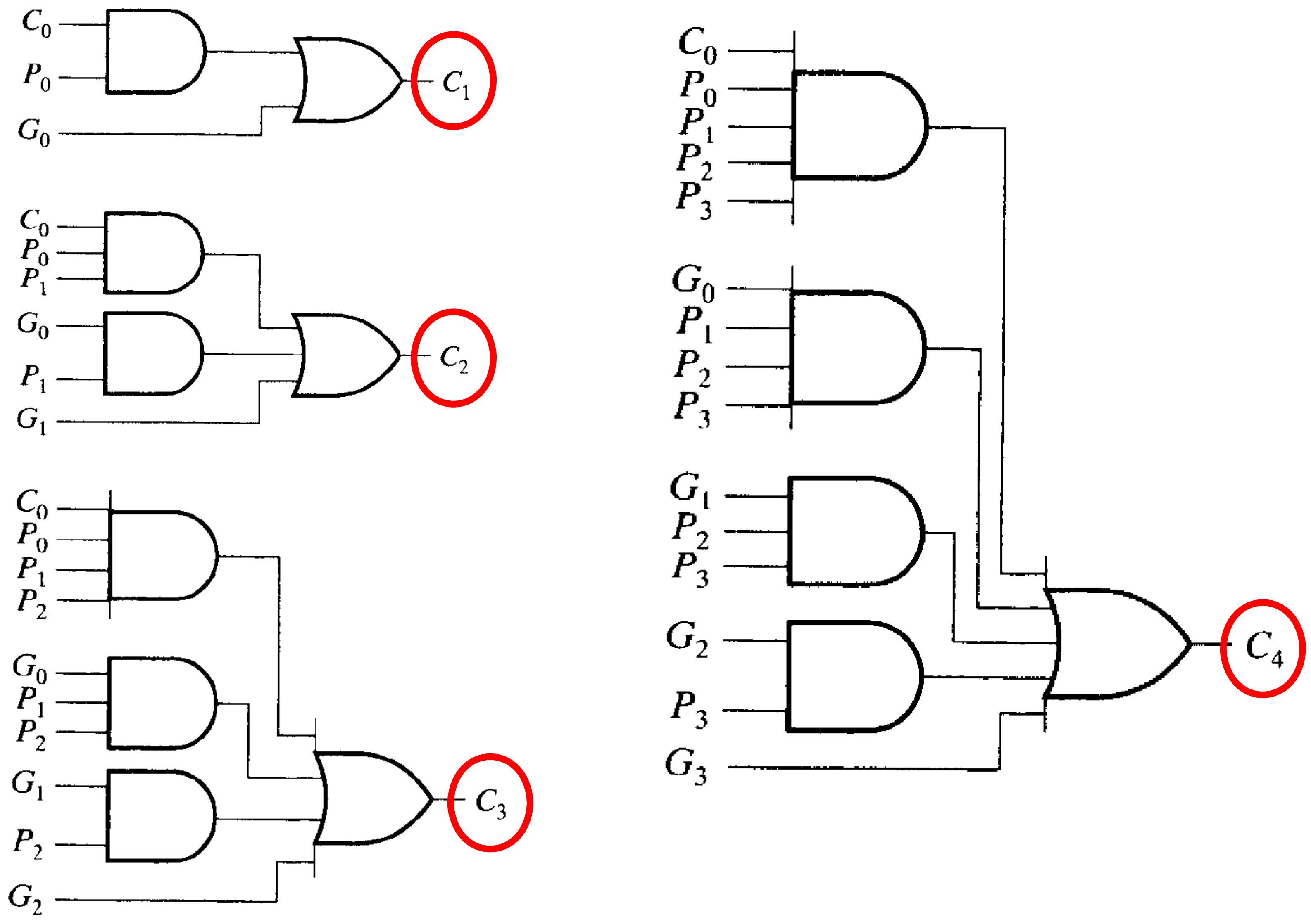
$$
\begin{aligned}
C_N &= G_{N-1} + P_{N-1}C_{N-1} \\
&= G_{N-1} + P_{N-1}\,G_{N-2} + P_{N-1}\,P_{N-2}C_{N-2} \\
&= \underbrace{G_{N-1} + P_{N-1}\,G_{N-2} + P_{N-1}\,P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0C_{IN}}
\end{aligned}
$$

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

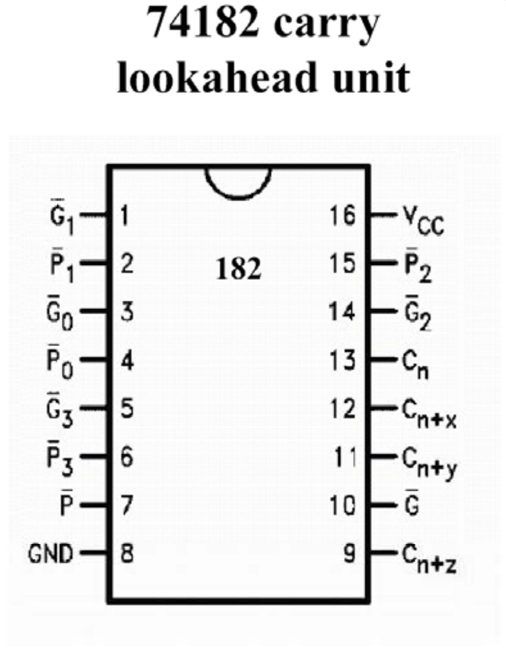
*assuming gates with N inputs

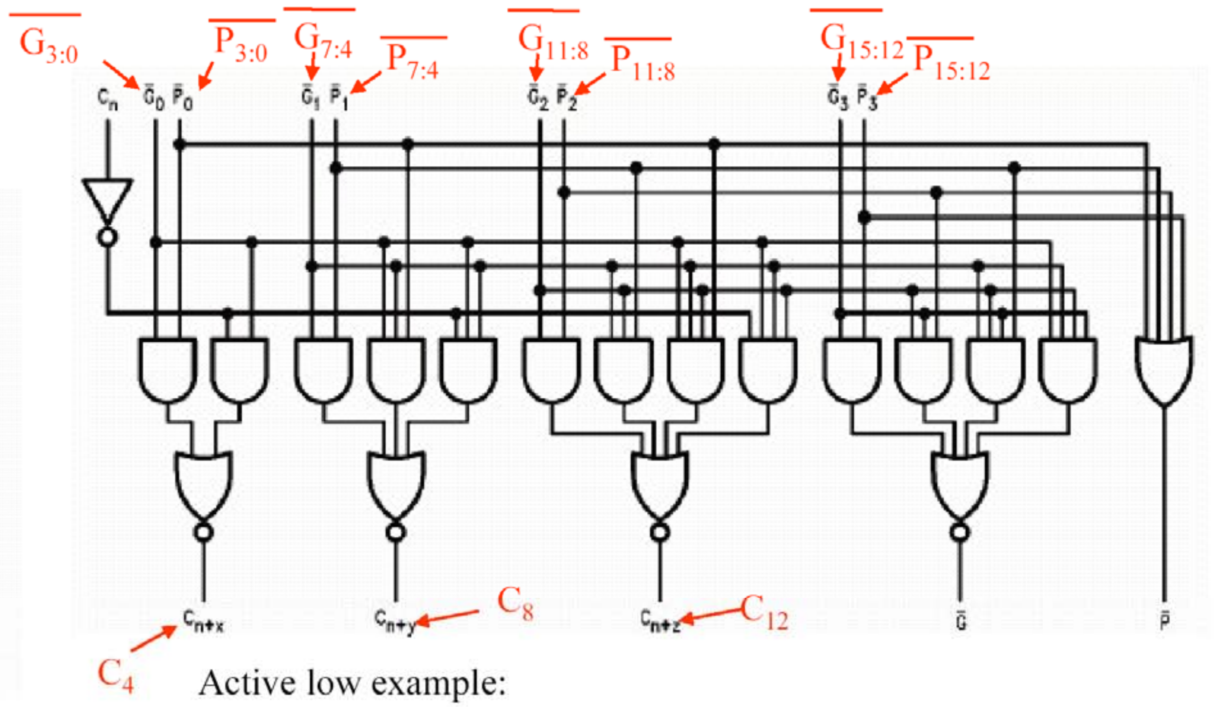
- Idea: pre-compute all carry bits as $f(Gs,Ps,C_{IN})$

Carry Lookahead Circuits



The 74182 Carry Lookahead Unit





Active low example:

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

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

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

➢ $C_4 = G_{3:0} + P_{3:0}C_n$

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

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

- 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:

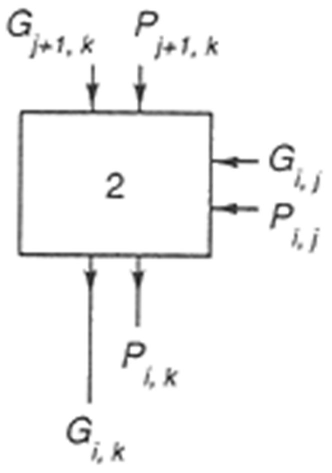
$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

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

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

where I < J and J+1 < K

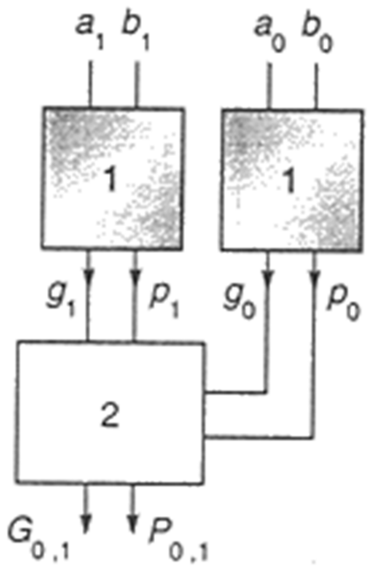
"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"



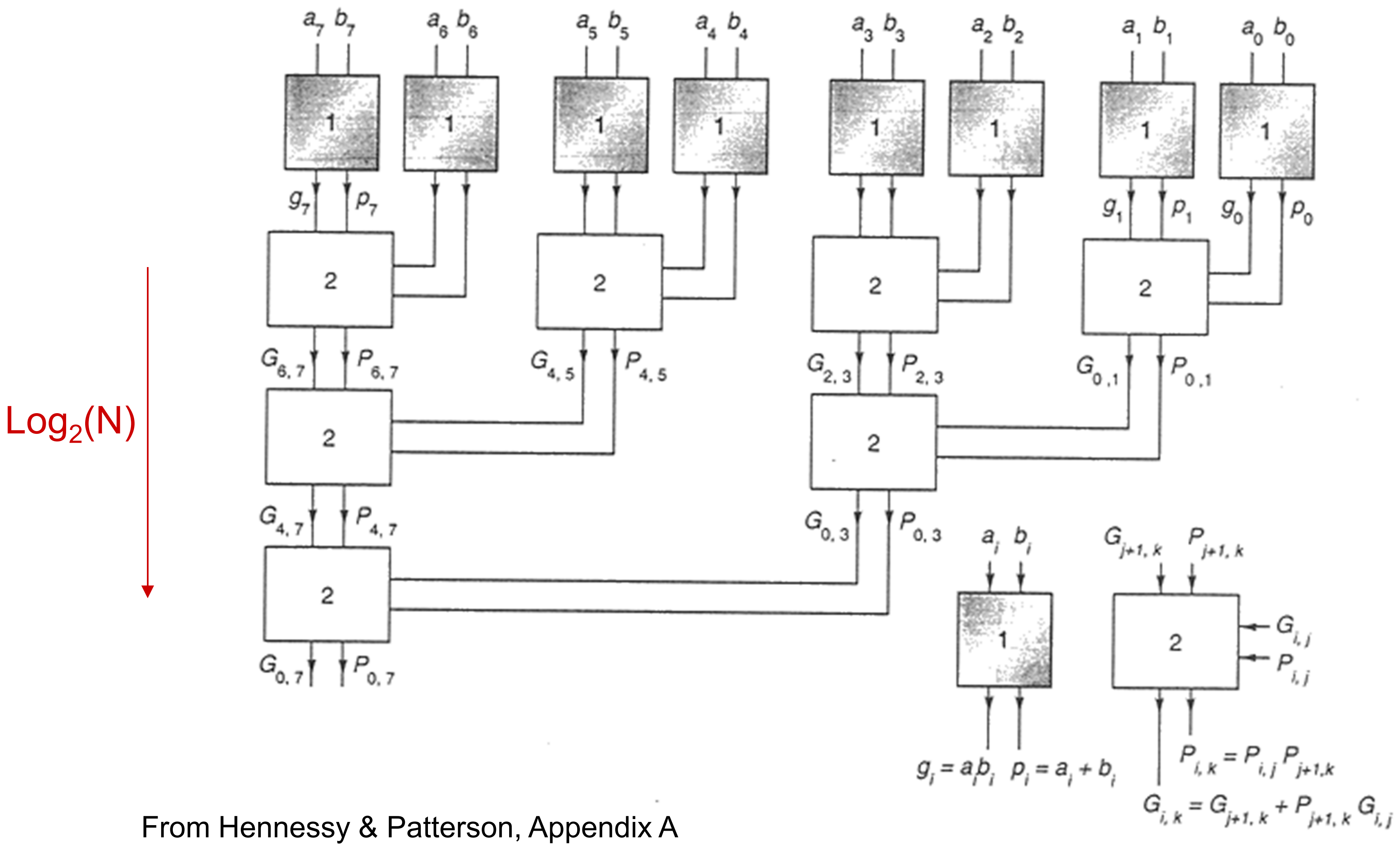
Hierarchical building block

P/G generation

1st level of lookahead

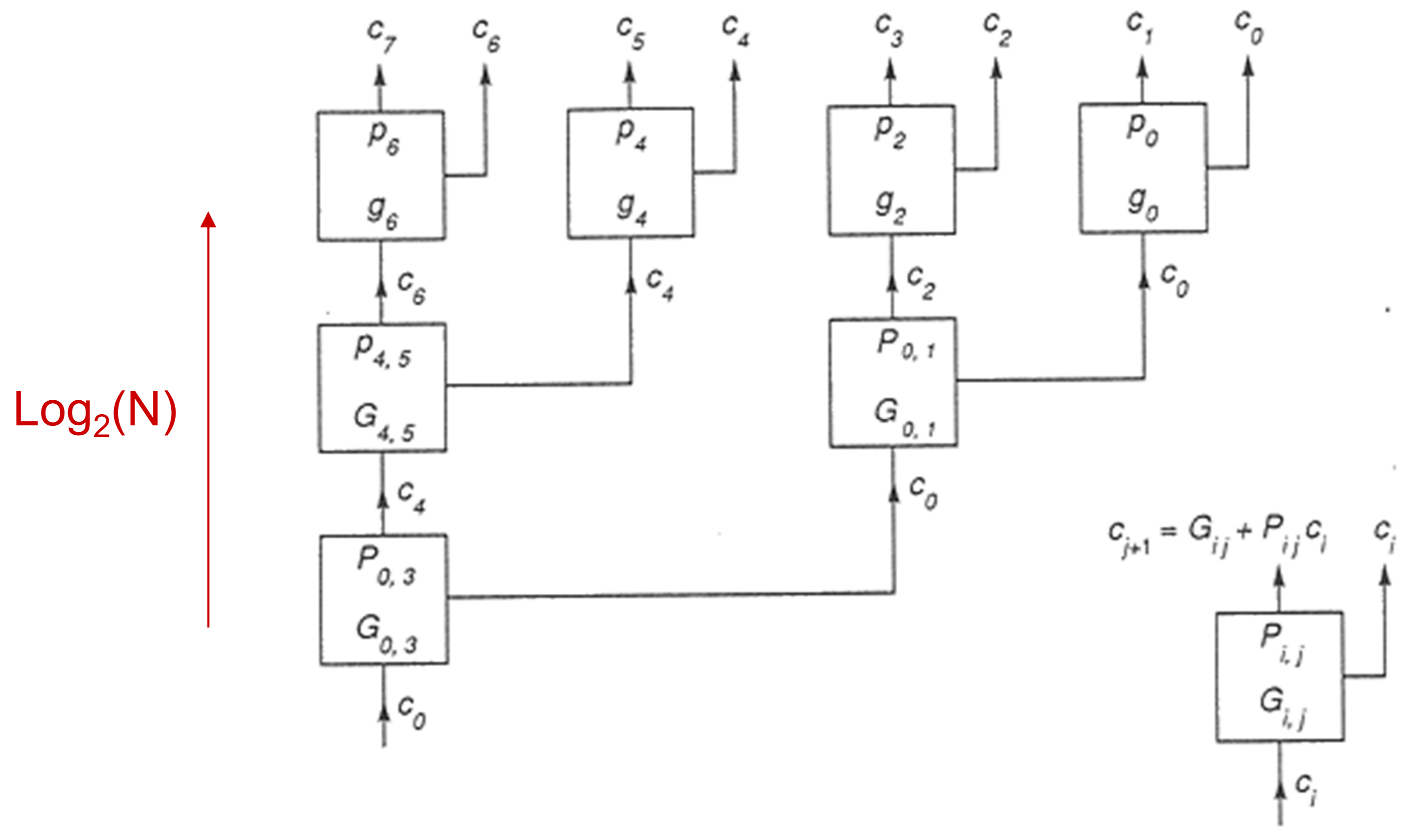


8-bit CLA (P/G generation)

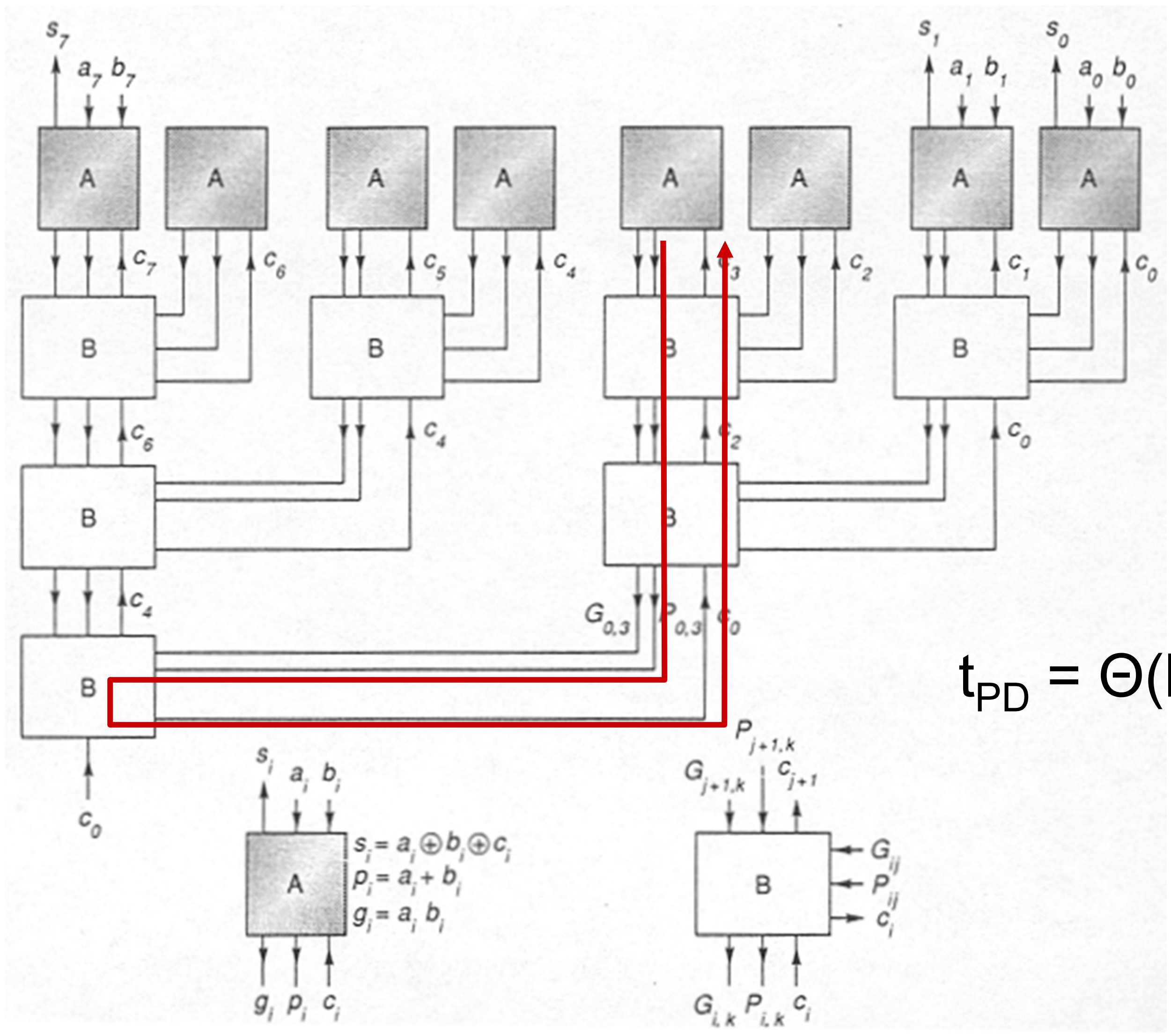


From Hennessy & Patterson, Appendix A

8-bit CLA (carry generation)



8-bit CLA (complete)



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

Unsigned Multiplication

AB_i called a “partial product” $\longrightarrow$

$$
\underbrace{
\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
}_{\text{Multiplying N-bit number by M-bit number gives (N+M)-bit result}}
$$

Easy part: forming partial products

(just an AND gate since B_I is either 0 or 1)

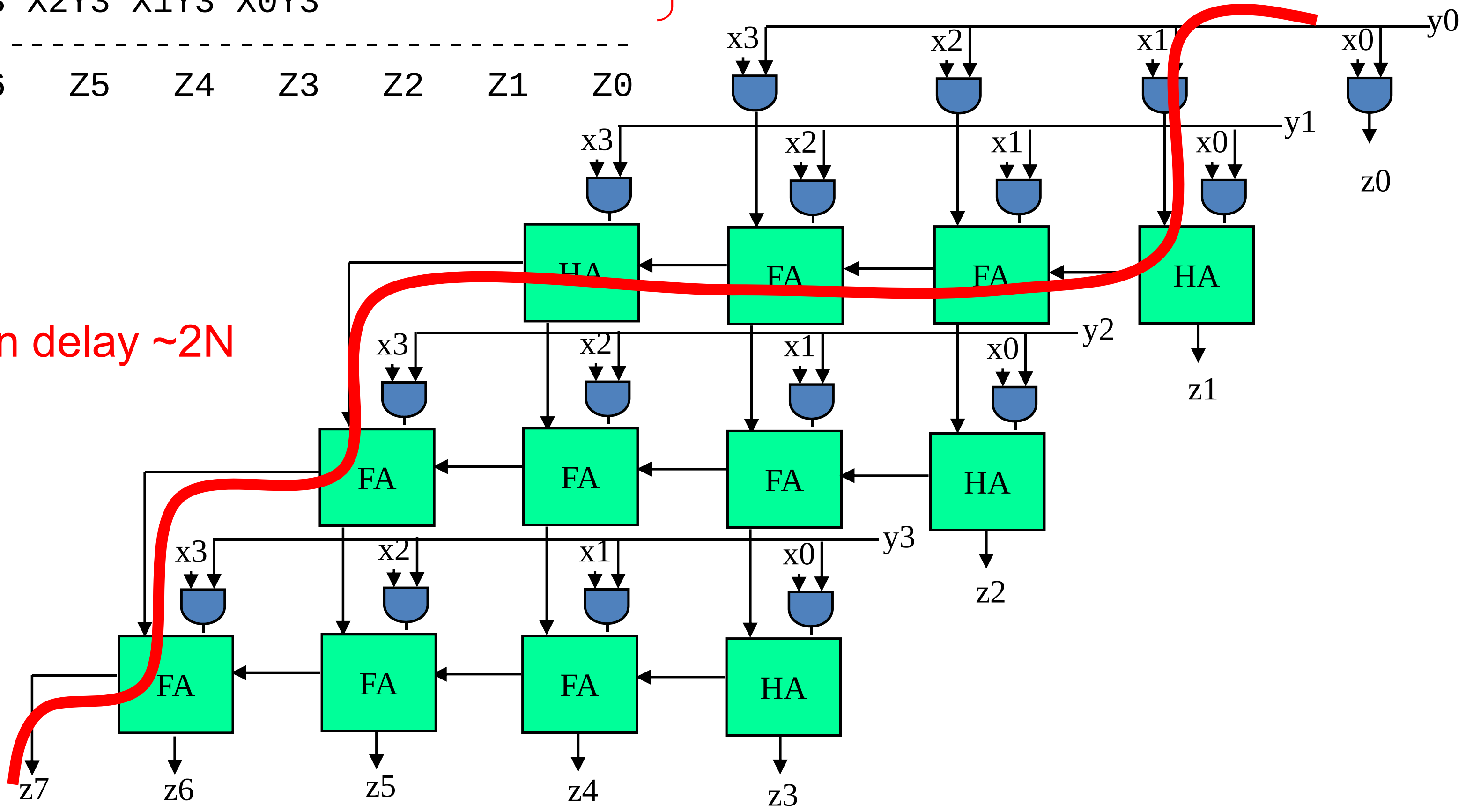
Hard part: adding M N-bit partial products

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
```

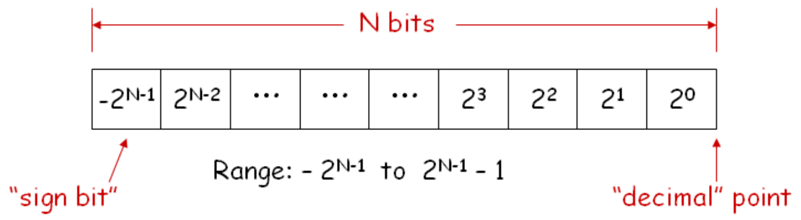
Partial products, one for each bit in multiplier (each bit needs just one AND gate)

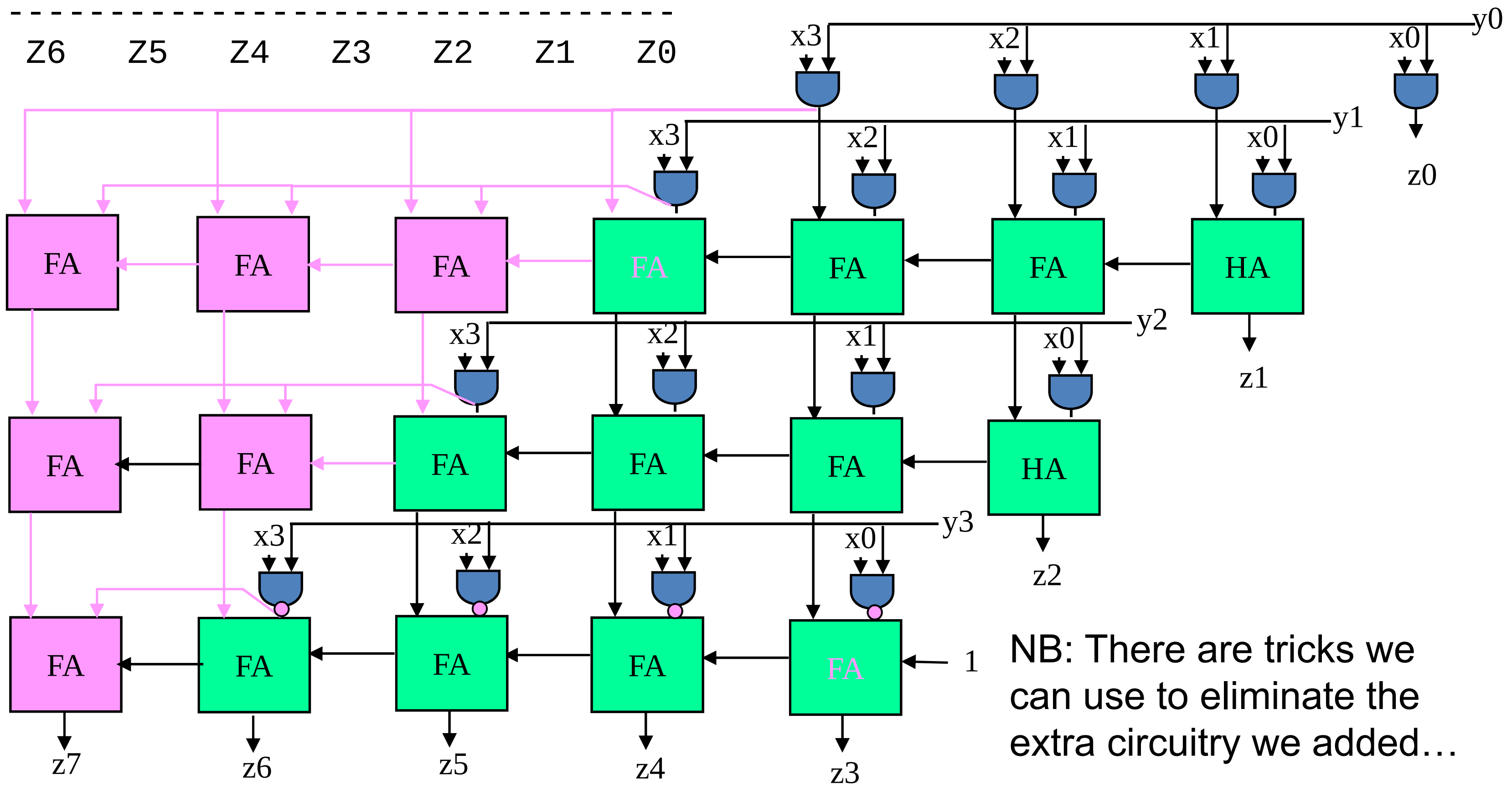
- Propagation delay ~2N



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

$$\begin{array}{ccccccccc}
 & & & & & X3 & X2 & X1 & X0 \\
 & & & & * & Y3 & Y2 & Y1 & Y0 \\
\hline
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & \\
- & X3Y3 & X3Y3 & X2Y3 & X1Y3 & X0Y3 & & & \\
\hline
 & Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0
\end{array}$$

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).

$$\begin{array}{ccccccccc}
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & & & & & 1 & & & \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & \\
+ & & & & 1 & & & & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & \\
+ & & & 1 & & & & & \\
+ & \overline{X3Y3} & \overline{X3Y3} & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & \\
+ & & & & & 1 & & & \\
+ & & 1 & & & & & & \\
- & & 1 & 1 & 1 & 1 & & &
\end{array}$$

$\left.\right\}$ $-B = \sim B + 1$

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

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

Step 4: finish computing the constants…

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

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:

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

The product then becomes:

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

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

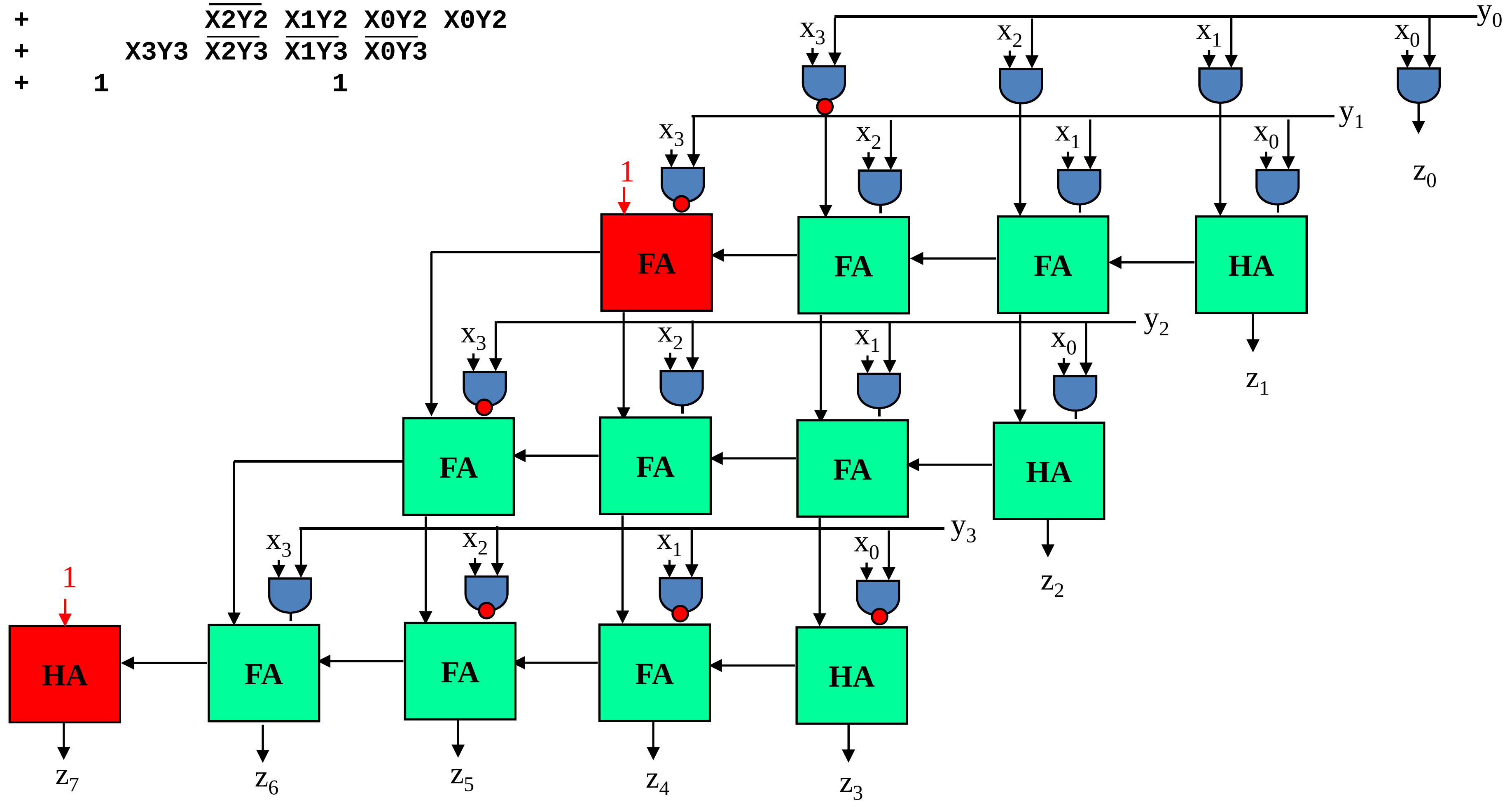
$$= -2^7 + x_3 y_3 2^6 + (\overline{x_2 y_3} + \overline{x_3 y_2}) 2^5 + (\overline{x_1 y_3} + \overline{x_3 y_1} + x_2 y_2 + 1) 2^4$$

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

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

2’s Complement Multiplication

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



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!
```

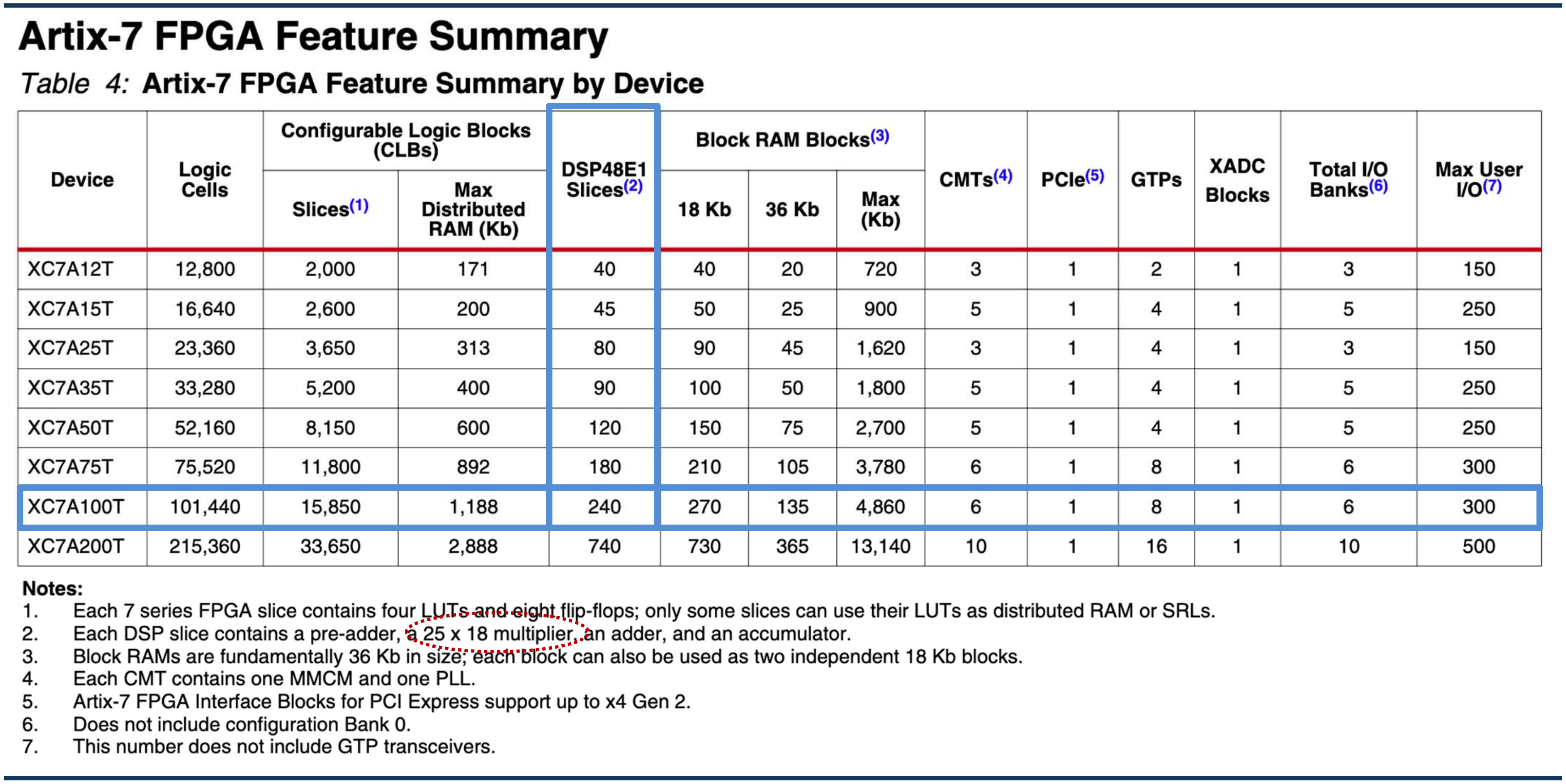
If you want Verilog to treat your operands as signed two’s complement numbers, add the keyword signed to your wire or reg declaration:

```
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

Artix-7 Details



Artix-7 FPGA Feature Summary

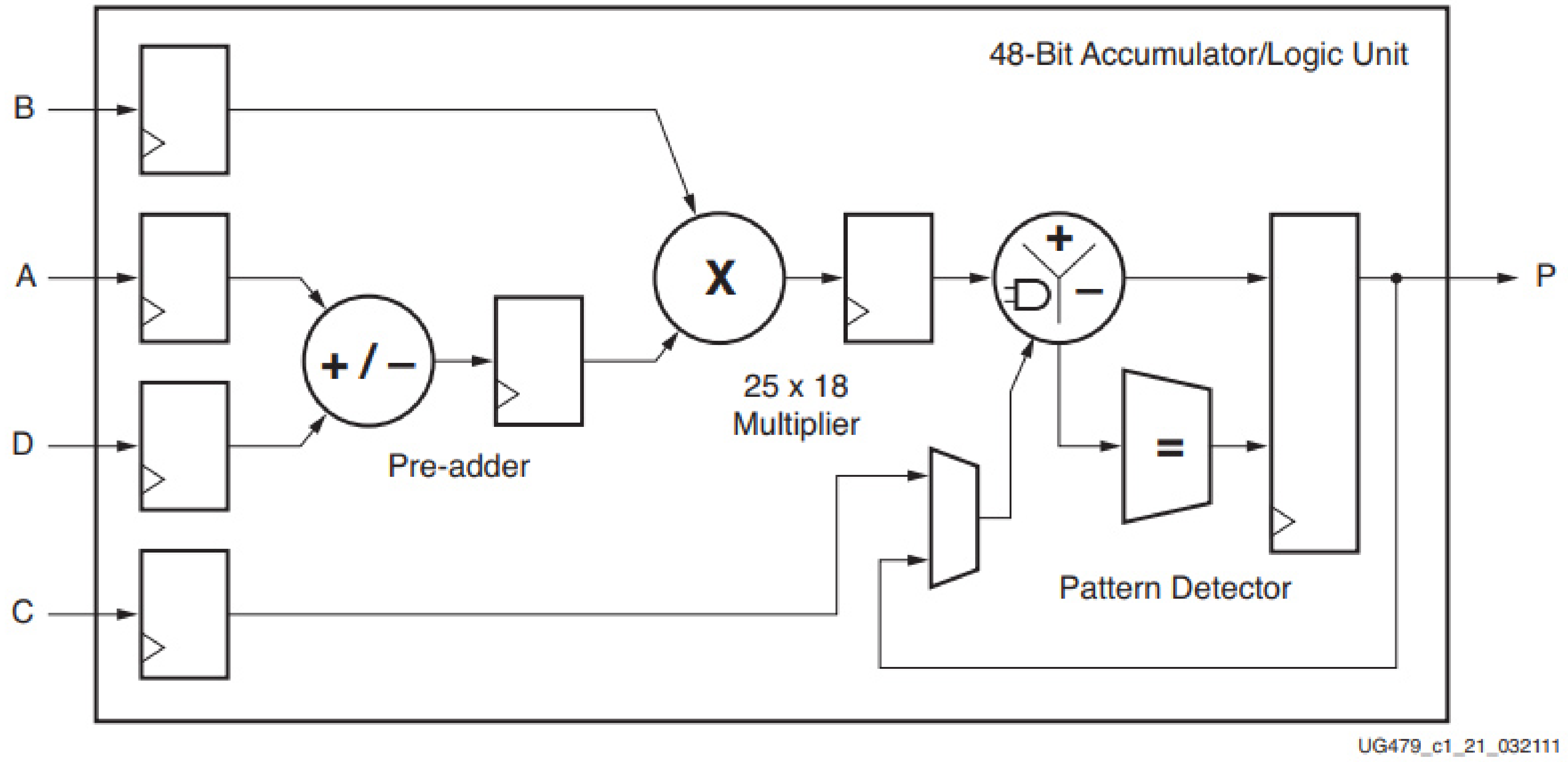
Table 4: **Artix-7 FPGA Feature Summary by Device**

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices(2)	Block RAM Blocks(3)			CMTs(4)	PCIe(5)	GTPs	XADC Blocks	Total I/O Banks(6)	Max User I/O(7)
		Slices(1)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7A12T	12,800	2,000	171	40	40	20	720	3	1	2	1	3	150
XC7A15T	16,640	2,600	200	45	50	25	900	5	1	4	1	5	250
XC7A25T	23,360	3,650	313	80	90	45	1,620	3	1	4	1	3	150
XC7A35T	33,280	5,200	400	90	100	50	1,800	5	1	4	1	5	250
XC7A50T	52,160	8,150	600	120	150	75	2,700	5	1	4	1	5	250
XC7A75T	75,520	11,800	892	180	210	105	3,780	6	1	8	1	6	300
XC7A100T	101,440	15,850	1,188	240	270	135	4,860	6	1	8	1	6	300
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500

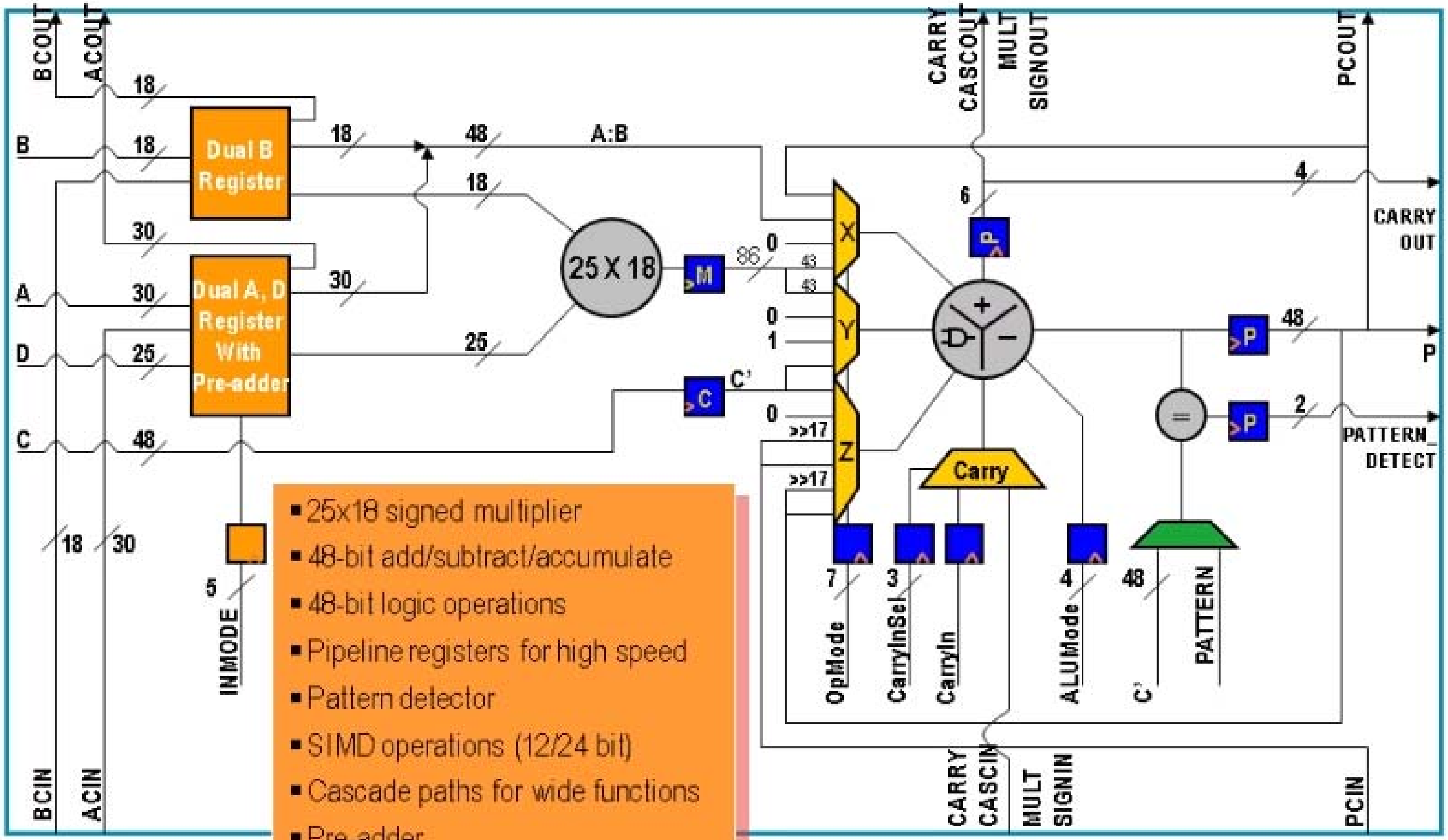
Notes:
1. Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
2. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
3. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
4. Each CMT contains one MMCM and one PLL.
5. Artix-7 FPGA Interface Blocks for PCI Express support up to x4 Gen 2.
6. Does not include configuration Bank 0.
7. This number does not include GTP transceivers.

Multiplier tpd = 3.97ns

Slice Overview



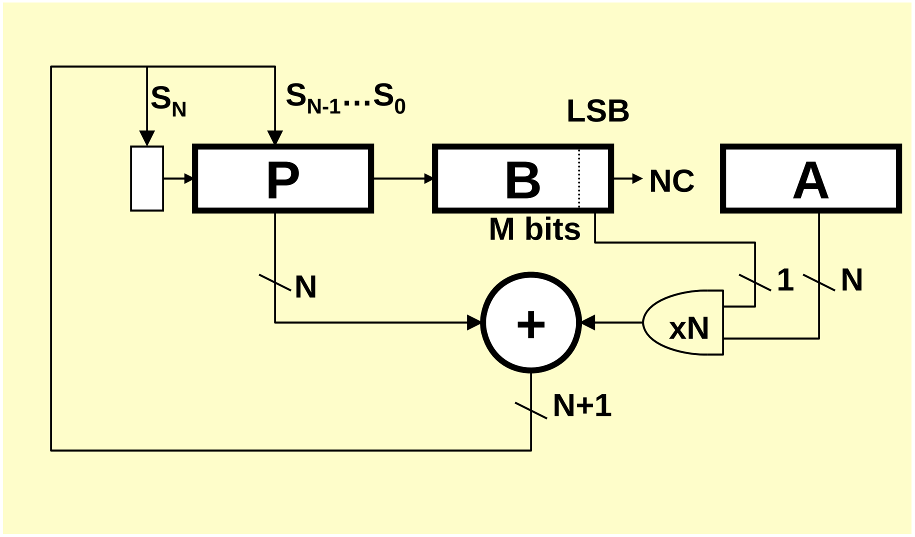
7 Series DSP48E1 Slice



- 25x18 signed multiplier
- 48-bit add/subtract/accumulate
- 48-bit logic operations
- Pipeline registers for high speed
- Pattern detector
- SIMD operations (12/24 bit)
- Cascade paths for wide functions
- Pre-adder

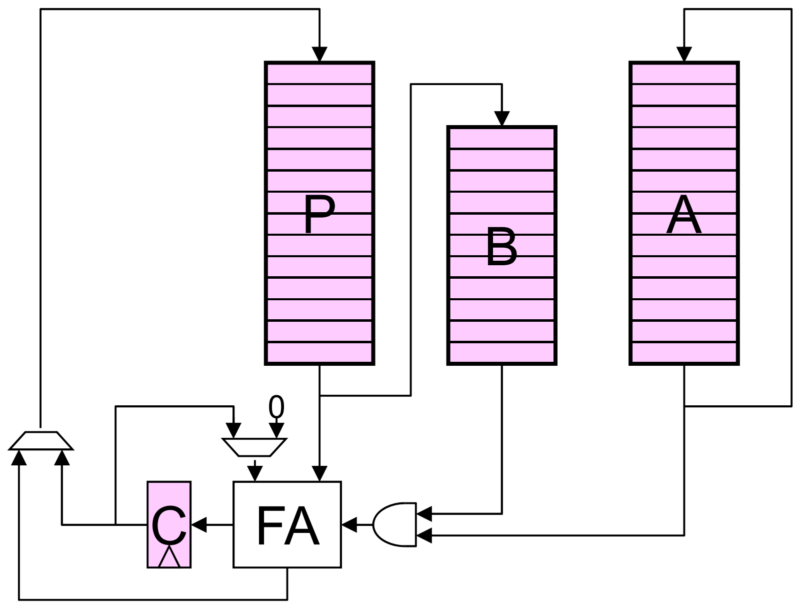
Sequential Multiplier

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)

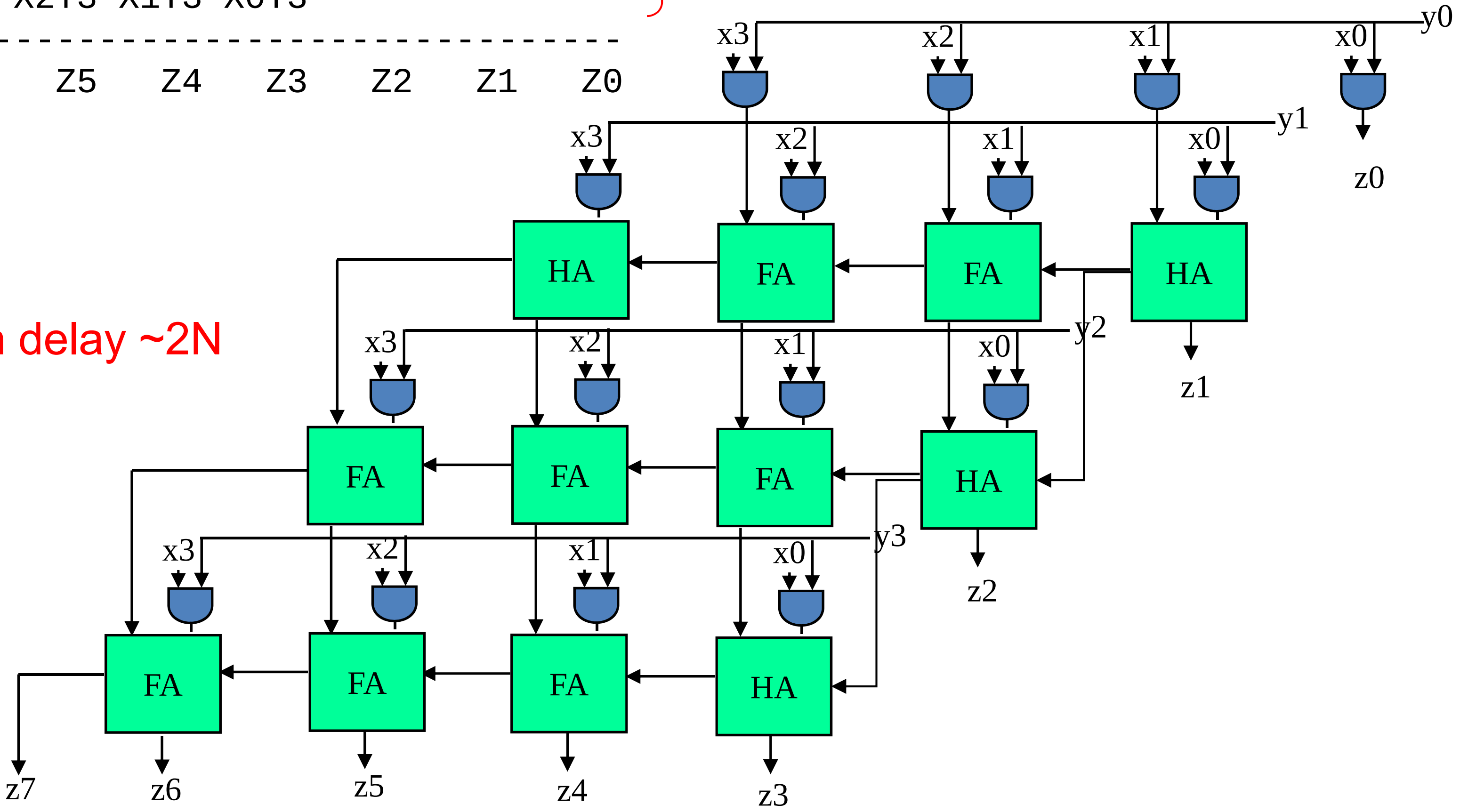
```
                     X3    X2    X1    X0
                  *  Y3    Y2    Y1    Y0
                  --------------------
                   X3Y0  X2Y0  X1Y0  X0Y0
+             X3Y1  X2Y1  X1Y1  X0Y1
+       X3Y2  X2Y2  X1Y2  X0Y2
+  X3Y3  X2Y3  X1Y3  X0Y3
--------------------------------------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

multiplicand

multiplier

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

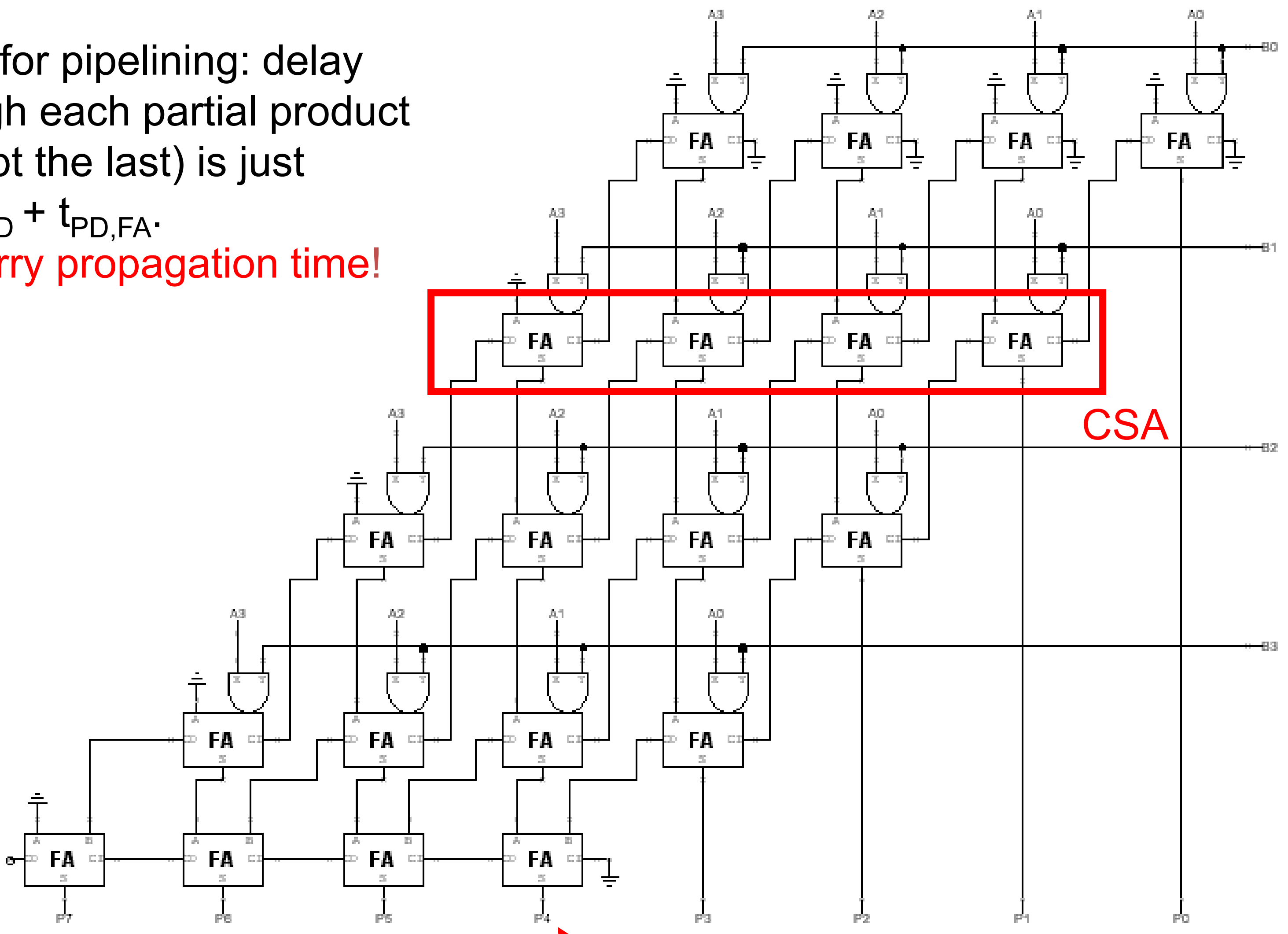
- Propagation delay ~2N



Useful building block: Carry-Save Adder

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

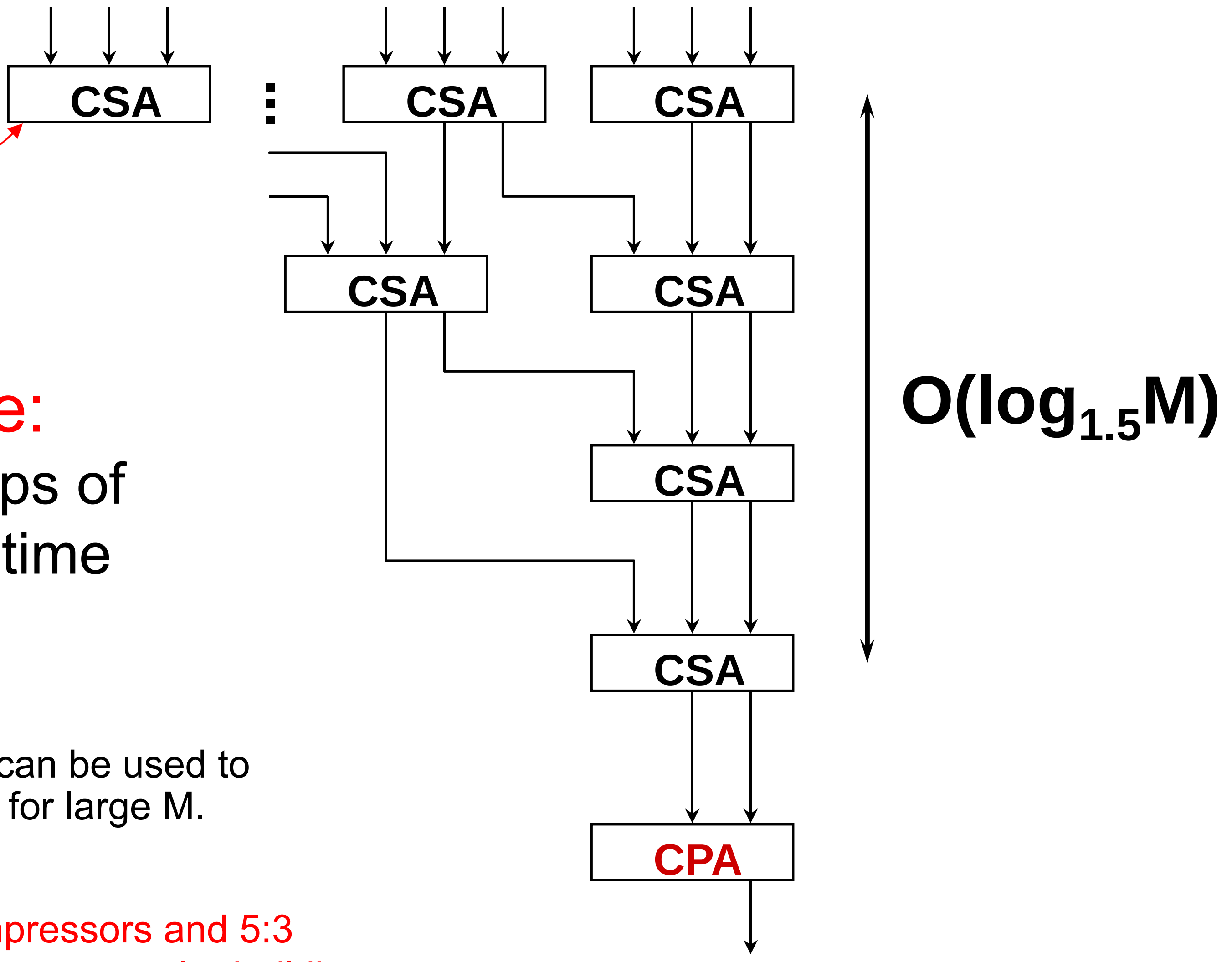
No carry propagation time!



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.



$O(\log_{1.5} M)$

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.

Artix-7 FPGA 3-2 Compressor

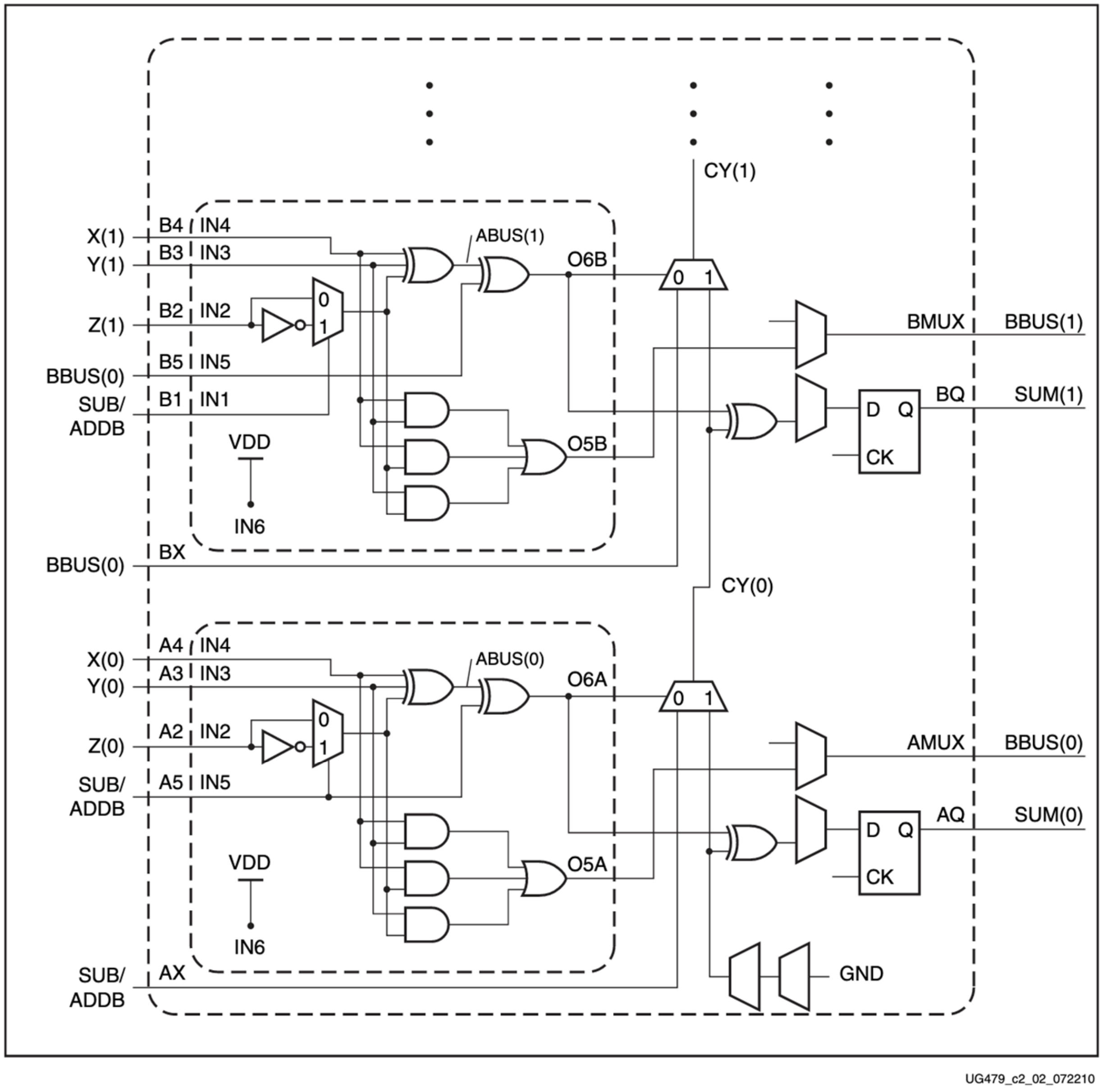


Figure 3-2: **Ternary Add/Sub with 3:2 Compressor**

Wallace Tree *
Four Bit Multiplier

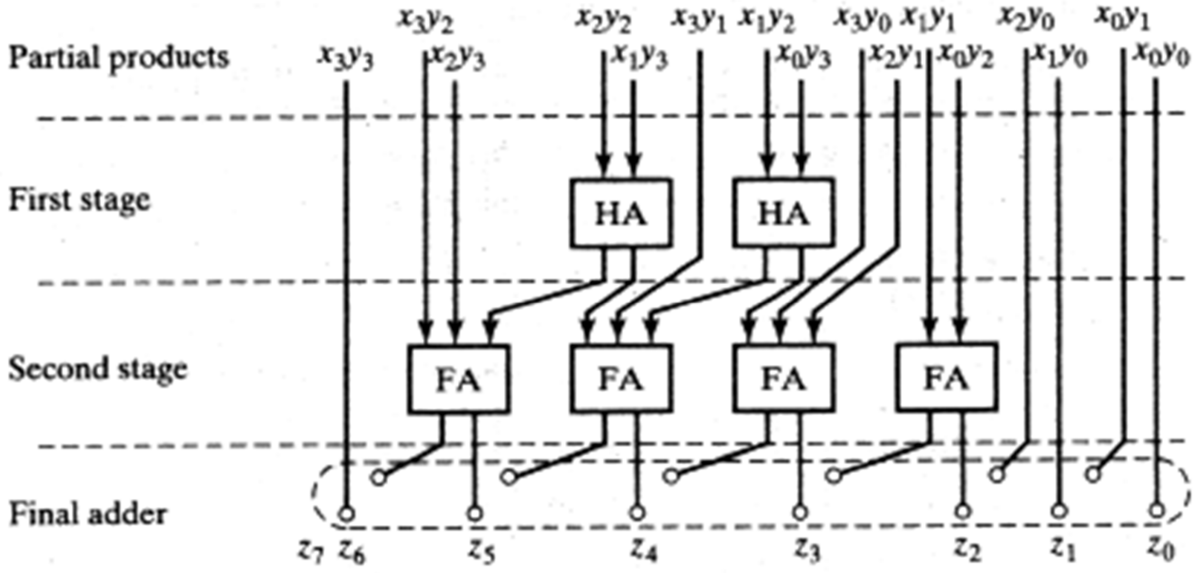


Figure 11-35 Wallace tree for four-bit multiplier.

*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?

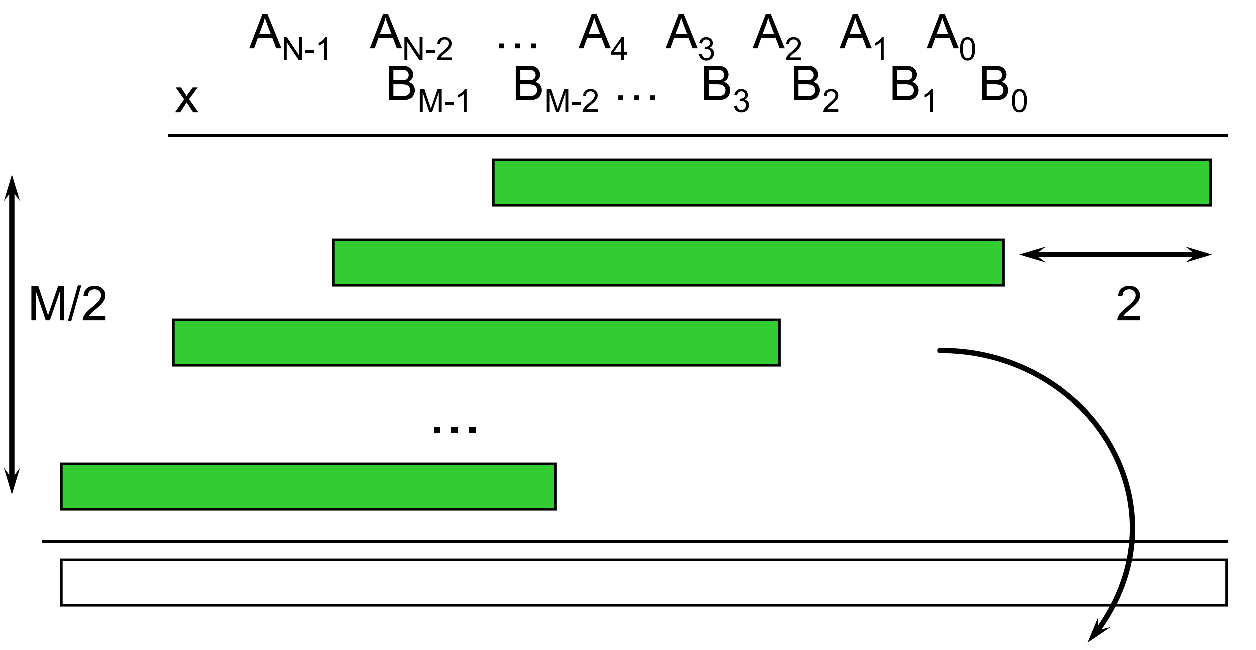
- If we allow ourselves to subtract PPs as well as adding them (the hardware cost is virtually the same), we can re-encode arbitrarily long contiguous runs of "1" bits in the multiplier to produce just two PPs.

$$\ldots011110\ldots = \ldots100000\ldots - \ldots000010\ldots = \ldots010001\bar{1}0\ldots$$

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)

Lab 4 Car Alarm - Design Approach

- Read lab/specifications carefully, use reasonable interpretation
- Use modular design – don't put everything into lab4_main.sv
- 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

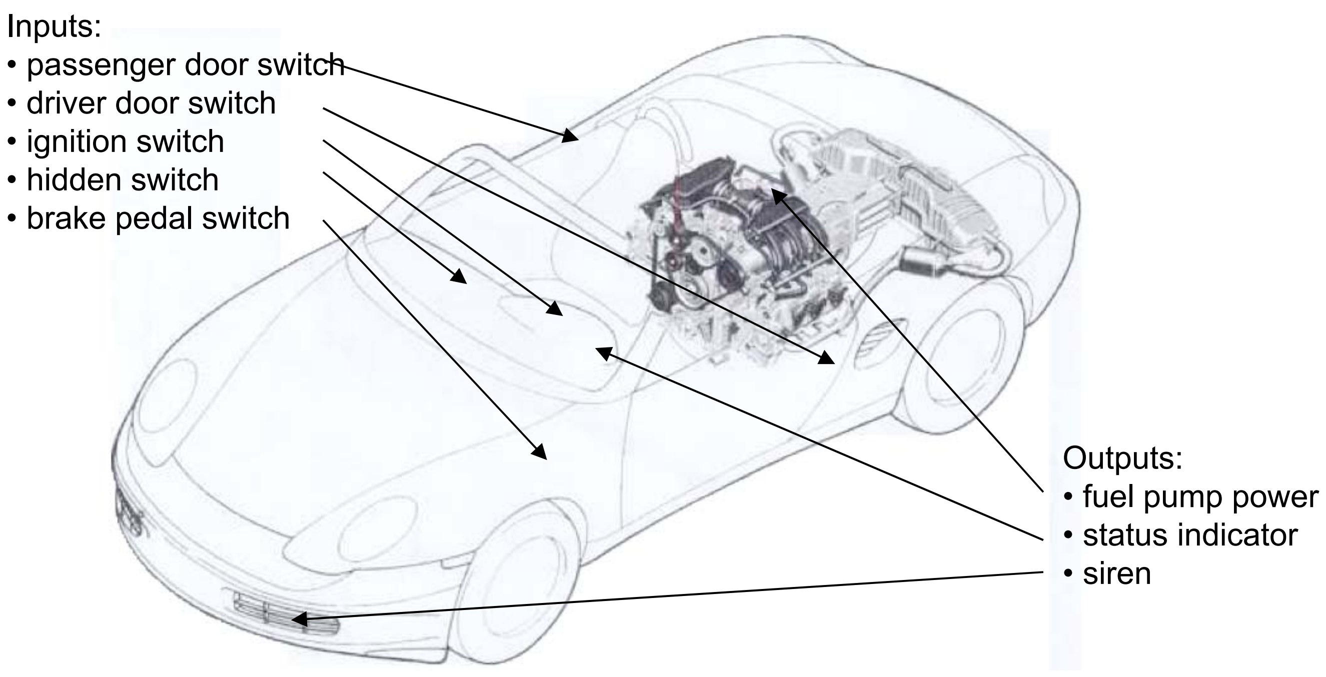
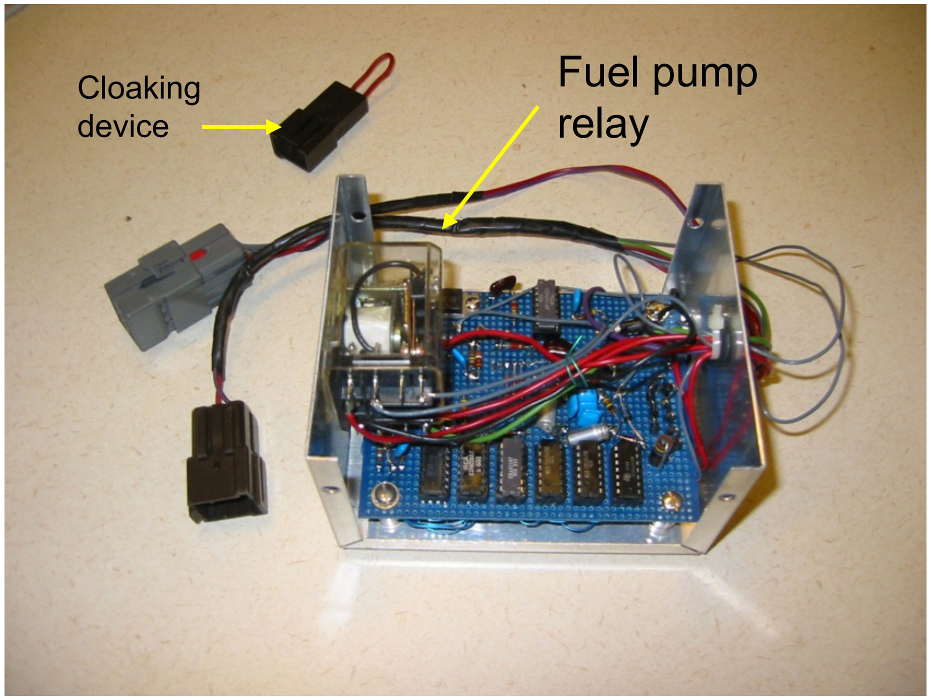


Figure 1: System diagram showing sensors (inputs) and actuators (outputs)

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

- Add parameter for fast debug mode for the one hz clock. This will allow for viewing signals in simulation or ILA without waiting for 25 million clock cycles. Avoids recomplilations.

```
module lab4_main.sv
timer  #(.DIVISOR(25_000_000)  my_timer(…)
// ------------------------------------------------

 module timer #(parameter DIVISOR=3)  (input clk_25mhz, ….
  // defaults to 3 clocks cycles
  …
  endmodule
```

Debugging Hints – Lab 4

- Implement a speedy debug mode for the one hz clock. This will allow for viewing signals on the ILA or simulation without waiting for 25 million clock cycles. Avoids recomplilations.

```
assign speedy = sw[6];
always_ff @ (posedge clk) begin
       if (count == (speedy ? 3 : 24_999_999)) count <= 0;
       else count <= count +1;
    end
assign one_hz = (count == (speedy ? 3 : 24_999_999)) ;
```

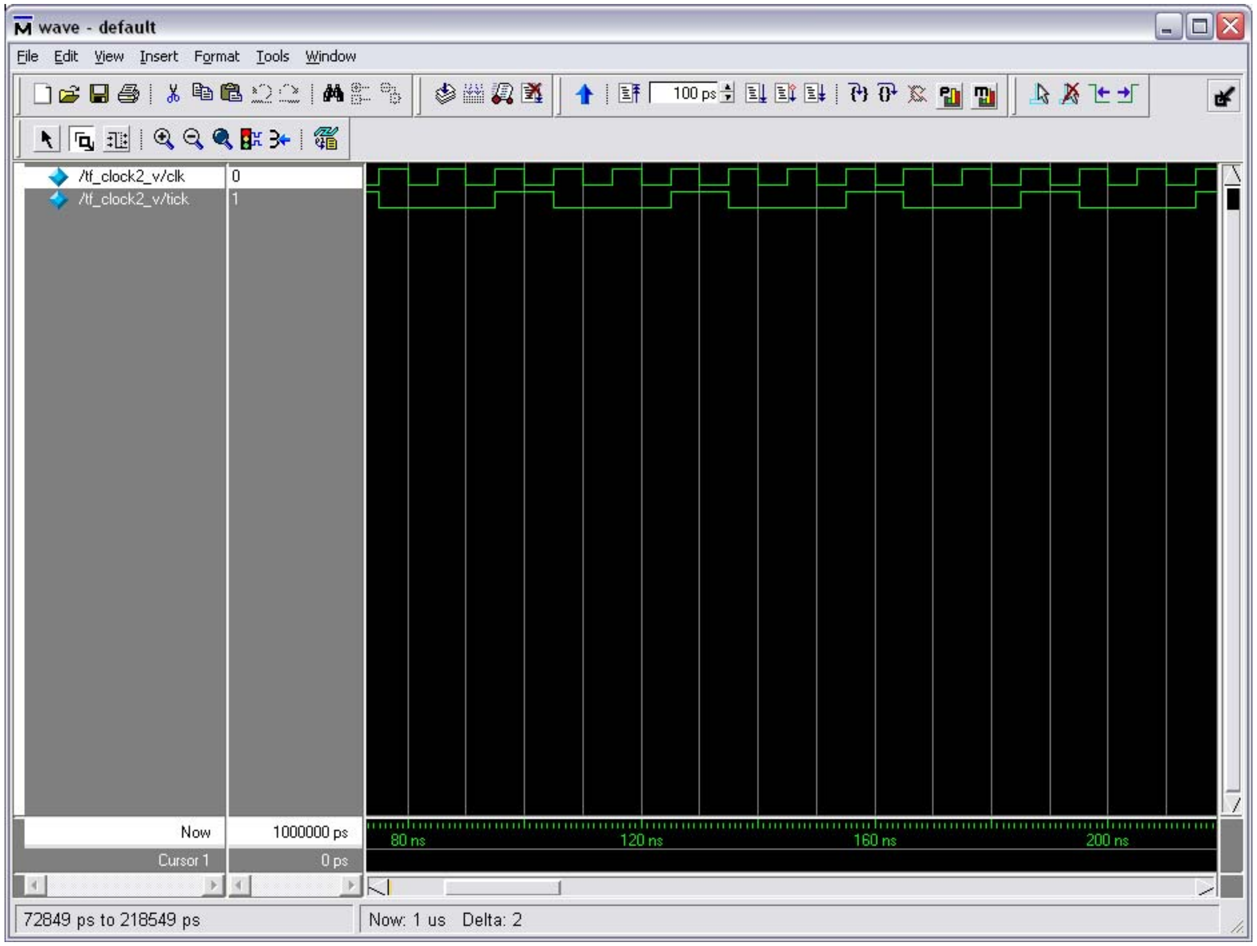
. . . Or use parameters . . .

```
module timer #(parameter DIVISOR=3)  (input clk_25mhz, ….
  // defaults to 3 clocks cycles
  …
```

One Hz Ticks in Simlulation

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;  . . .
```

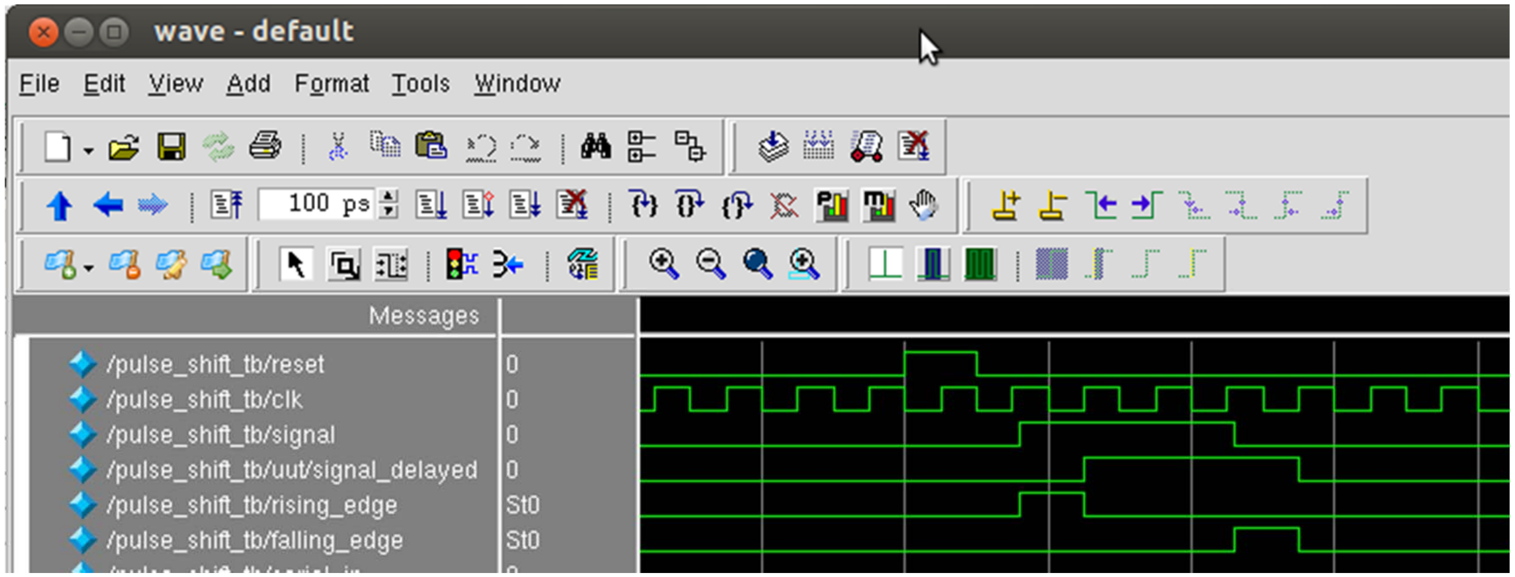


/tf_clock2_v/clk	0
/tf_clock2_v/tick	1
Now	1000000 ps
Cursor 1	0 ps

72849 ps to 218549 ps

Now: 1 us   Delta: 2

Edge Detection



```
/pulse_shift_tb/reset                 0
/pulse_shift_tb/clk                   0
/pulse_shift_tb/signal                0
/pulse_shift_tb/uut/signal_delayed    0
/pulse_shift_tb/rising_edge           St0
/pulse_shift_tb/falling_edge          St0
```

```
logic signal_delayed;

always_ff @(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.

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.”

```
module stage_1 (input clk_in,
               input rst_in,
               input[7:0] s1_in,
               output logic [8:0] s1_out
                );

    //Generate two options: (See LPset 3)
```
$y_{n1} = x_n \oplus x_{n-1}$ for $7 \geq n \geq 1$ $\quad y_{n2} = \overline{(x_n \oplus x_{n-1})}$

```
    //Identify transitions in each:
```
$t_{n-1} = y_n \oplus y_{n-1}$ for $7 \geq n \geq 1$

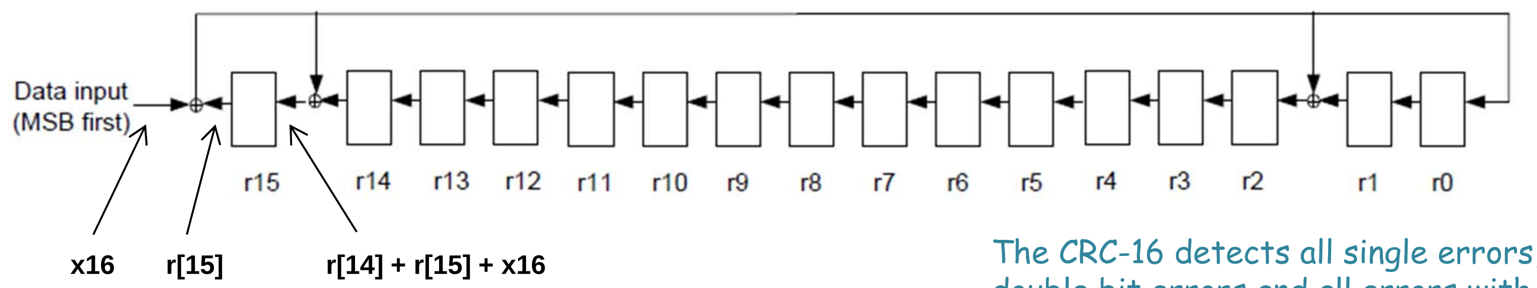
```
    //Tally the transitions in each situation:
```
$assign\ sum = t_0 + t_1 \ldots t_8$

```
//Based on tallies, choose one with fewer (or equal) and
//produce correct output
```
assign s1_out = (sum1<=sum2)? …

```
endmodule
```

Cyclic redundancy check - CRC

CRC16 (x16 + x15 + x2 + 1)



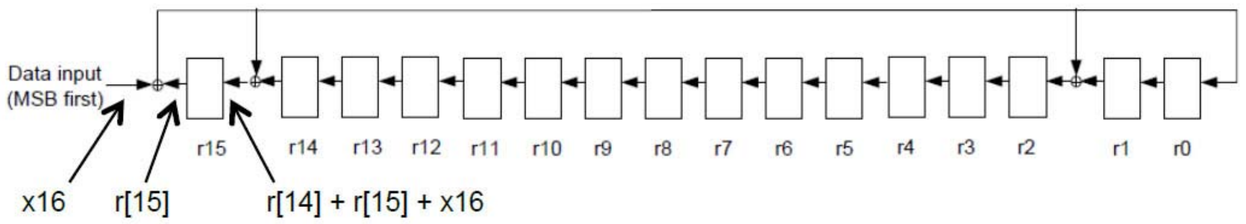
The CRC-16 detects all single errors, all double bit errors and all errors with burst less than 16 bits in length.

- Each "r" is a register, all clocked with a common clock. Common clock not shown
- As shown, for register r15, the output is r[15] and the input is the sum of r[14], r[15] and data input x16, etc
- The small round circles with the plus sign are adders implemented with XOR gates.
- Initialize r to 16'hFFFF at start

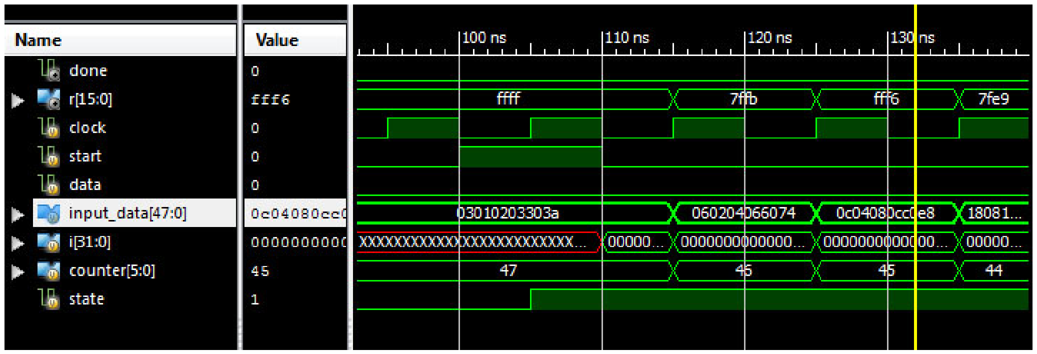
CRC Solution

CRC16: x16+x15+x2+1

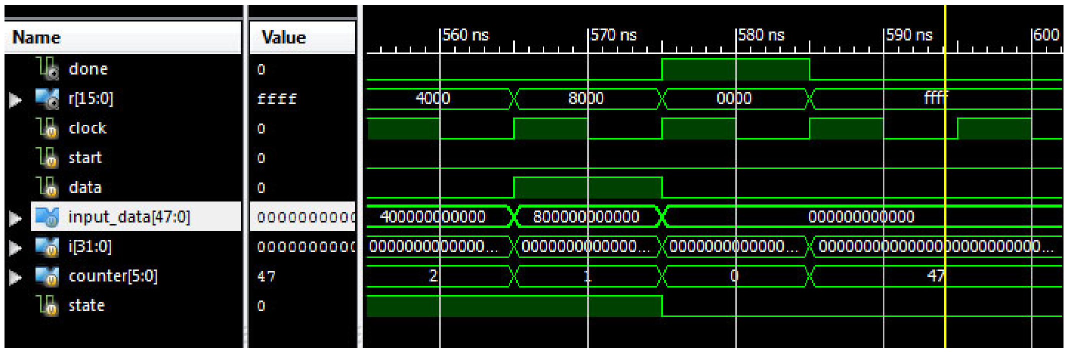
```
module lpset6(
    input clock,
    input start,
    input data,
    output done,
    output reg [15:0] r
    );
    parameter IDLE=0;
    parameter CRC_CALC=1;

    wire x16 = data;
    reg state=0;
    reg [5:0] counter=0; //my counter

    always @ (posedge clock) begin
       case (state)
        IDLE: begin



        end

        CRC_CALC: begin







         end

       endcase
      end

      assign done = (counter == 0);
endmodule
```



Data input
(MSB first)
r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r1
r0
x16
r[15]
r[14] + r[15] + x16



Name
Value
done 0
r[15:0] fff6
clock 0
start 0
data 0
input_data[47:0] 0c04080cc0
i[31:0] 000000000
counter[5:0] 45
state 1
100 ns
110 ns
120 ns
130 ns
ffff
7ffb
fff6
7fe9
03010203303a
060204066074
0c04080cc0e8
18081...
XXXXXXXXXXXXXXXXXXXXXXXXXX...
00000...
0000000000000...
0000000000000...
00000...
47
46
45
44



Name
Value
done 0
r[15:0] ffff
clock 0
start 0
data 0
input_data[47:0] 000000000
i[31:0] 000000000
counter[5:0] 47
state 0
560 ns
570 ns
580 ns
590 ns
600
4000
8000
0000
ffff
400000000000
800000000000
000000000000
0000000000000...
0000000000000...
0000000000000...
0000000000000000000000000...
2
1
0
47