Multipliers & Pipelining

- Combinational multiplier
- Two's complement multiplier
- Smaller multipliers, faster multipliers
- Latency & Throughput
- Pipelining to increase throughput
- Retiming

Lab #3 due tonight, report next Tuesday, no LPSets this week

Unsigned Multiplication

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

AB_i called a "partial product"

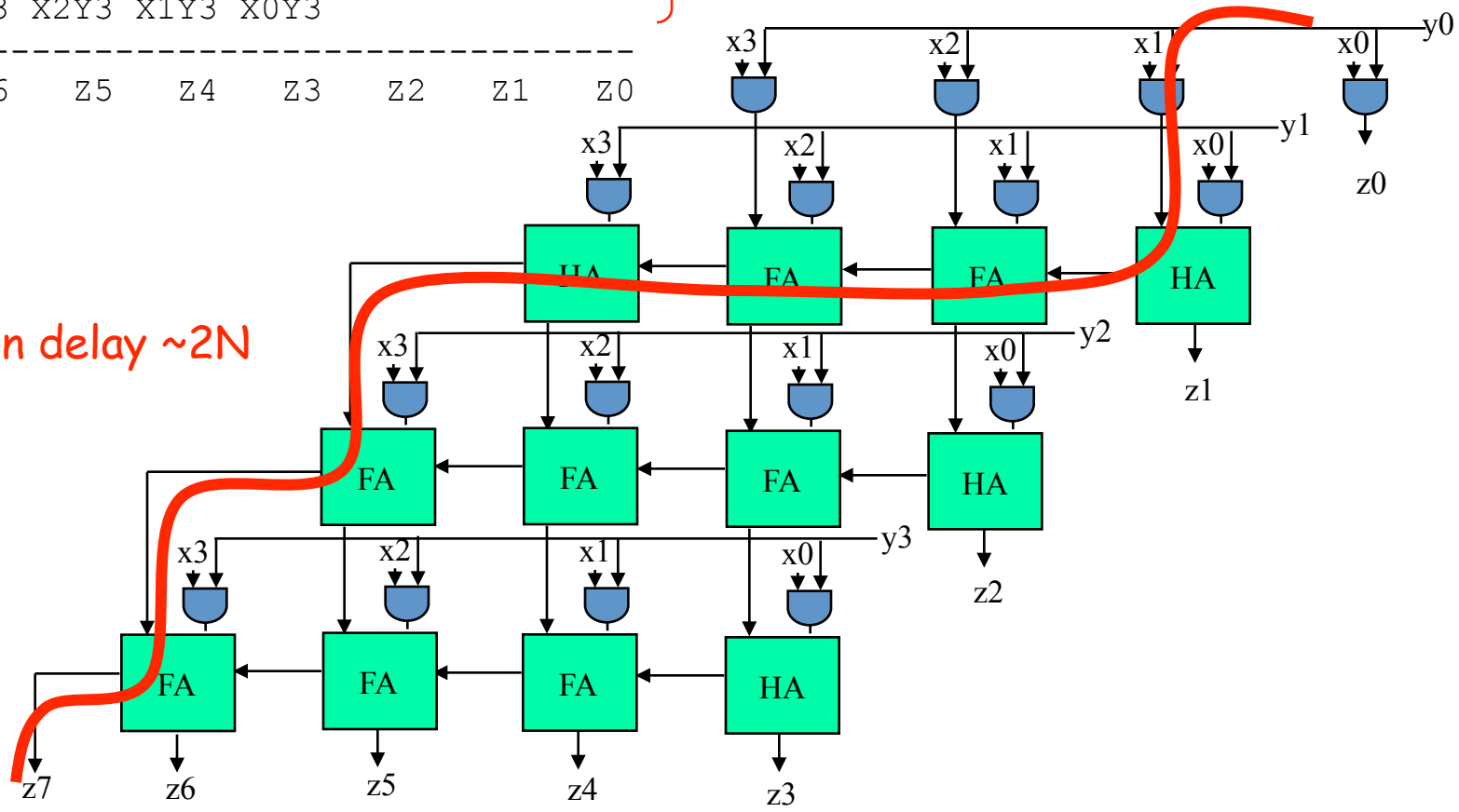
Multiplying N-bit number by M-bit number gives (N+M)-bit result

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
```

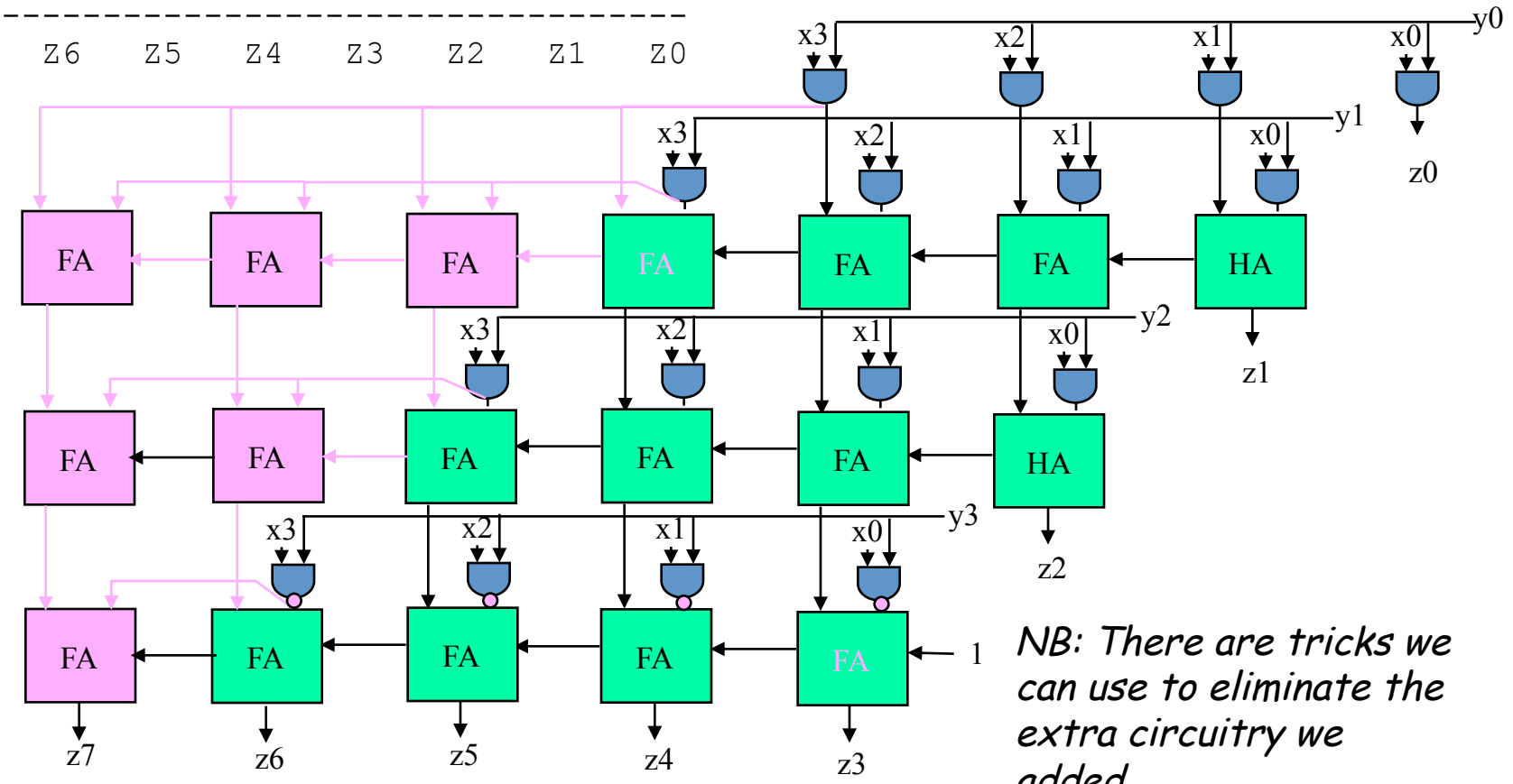
Partial products, one for each bit in multiplier (each bit needs just one AND gate)



- 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!

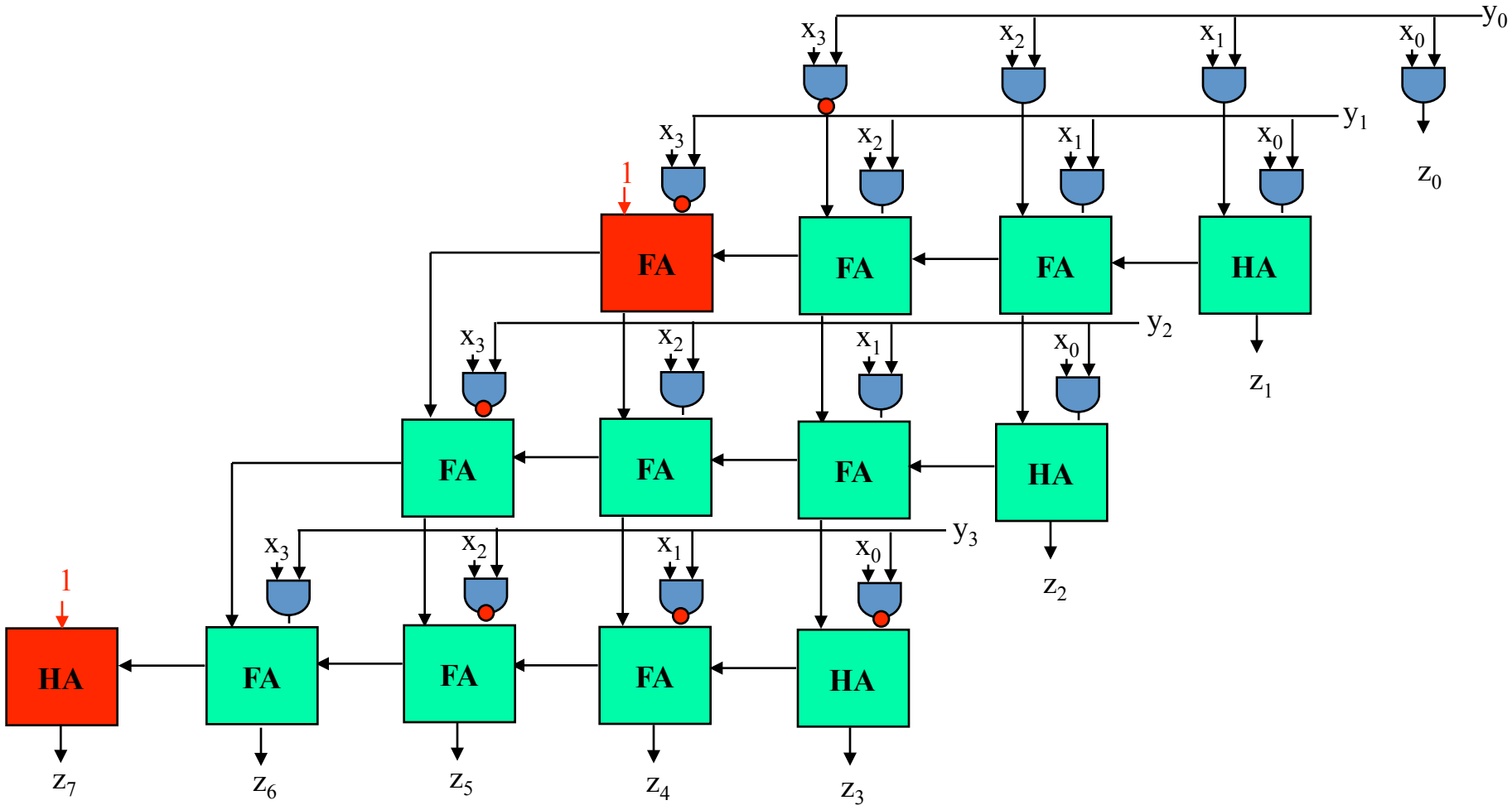
```
                              ____
                        ____  X3Y0  X2Y0  X1Y0  X0Y0
+                 ____  X3Y1  X2Y1  X1Y1  X0Y1
+           ____  X2Y2  X1Y2  X0Y2
+     X3Y3  X2Y3  X1Y3  X0Y3
+                             1
-         1     1     1     1
```

Step 4: finish computing the constants…

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

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

2's Complement Multiplication



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:

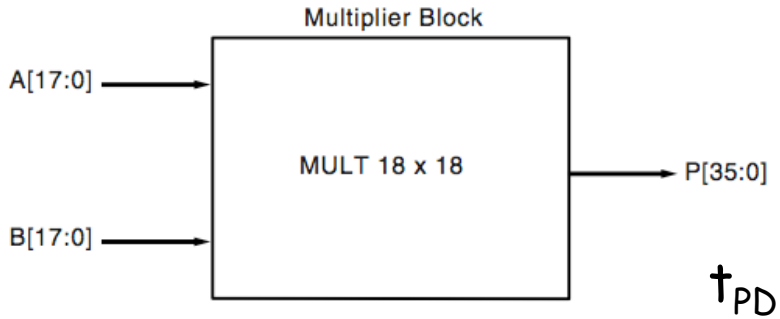
```
wire signed [9:0] a,b;
wire signed [19:0] result = a*b;  // signed multiplication!
```

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

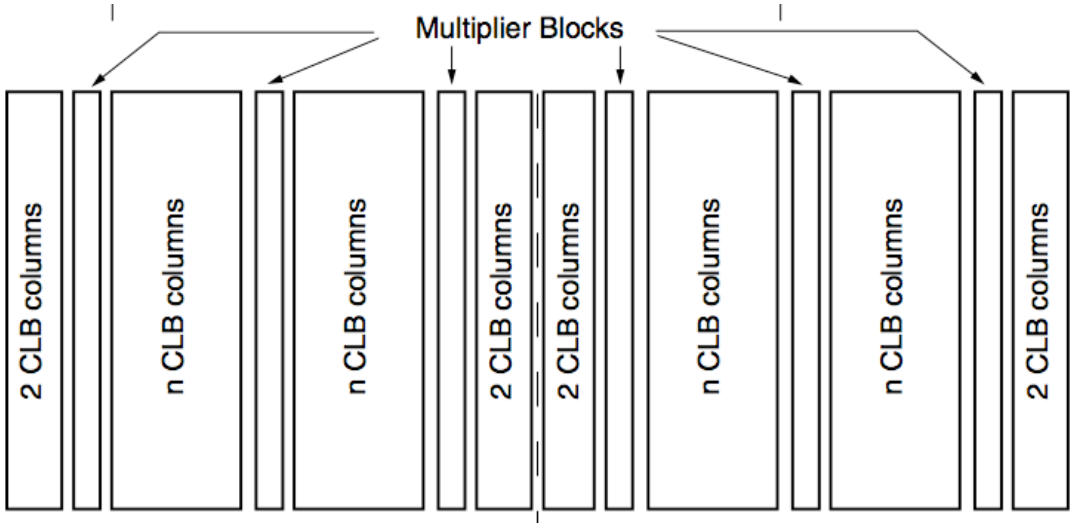
Multiplication on the FPGA

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



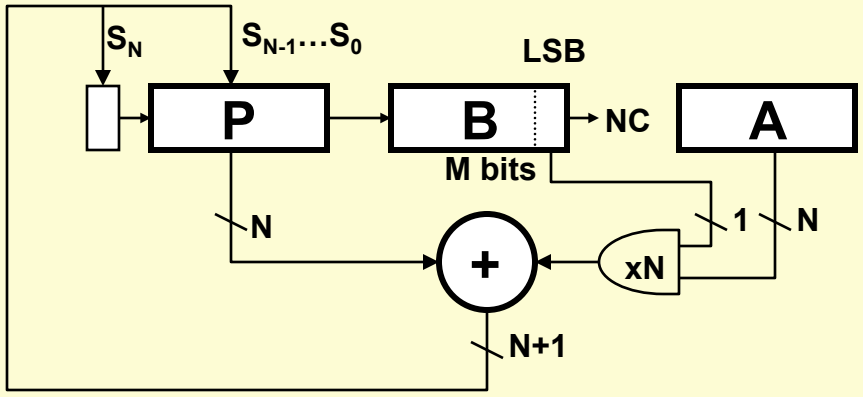
$t_{PD} \approx 10ns$

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



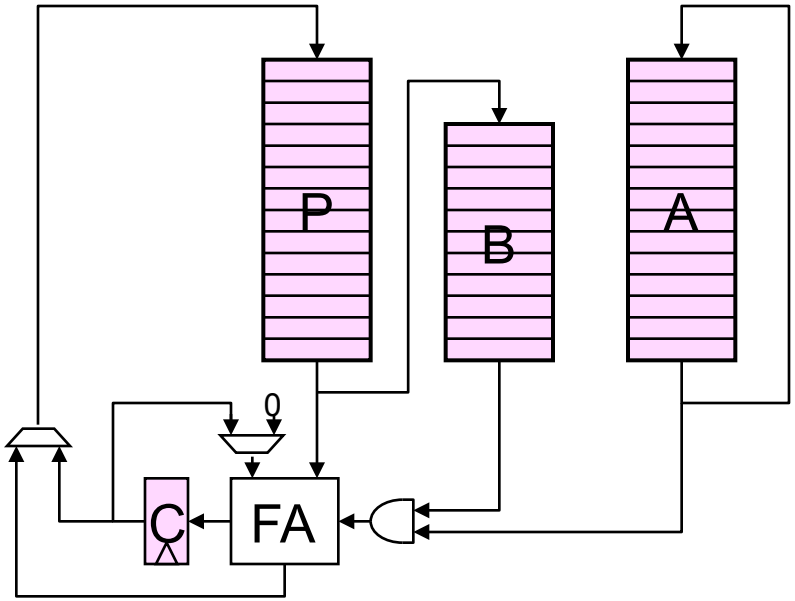
Sequential Multiplier

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

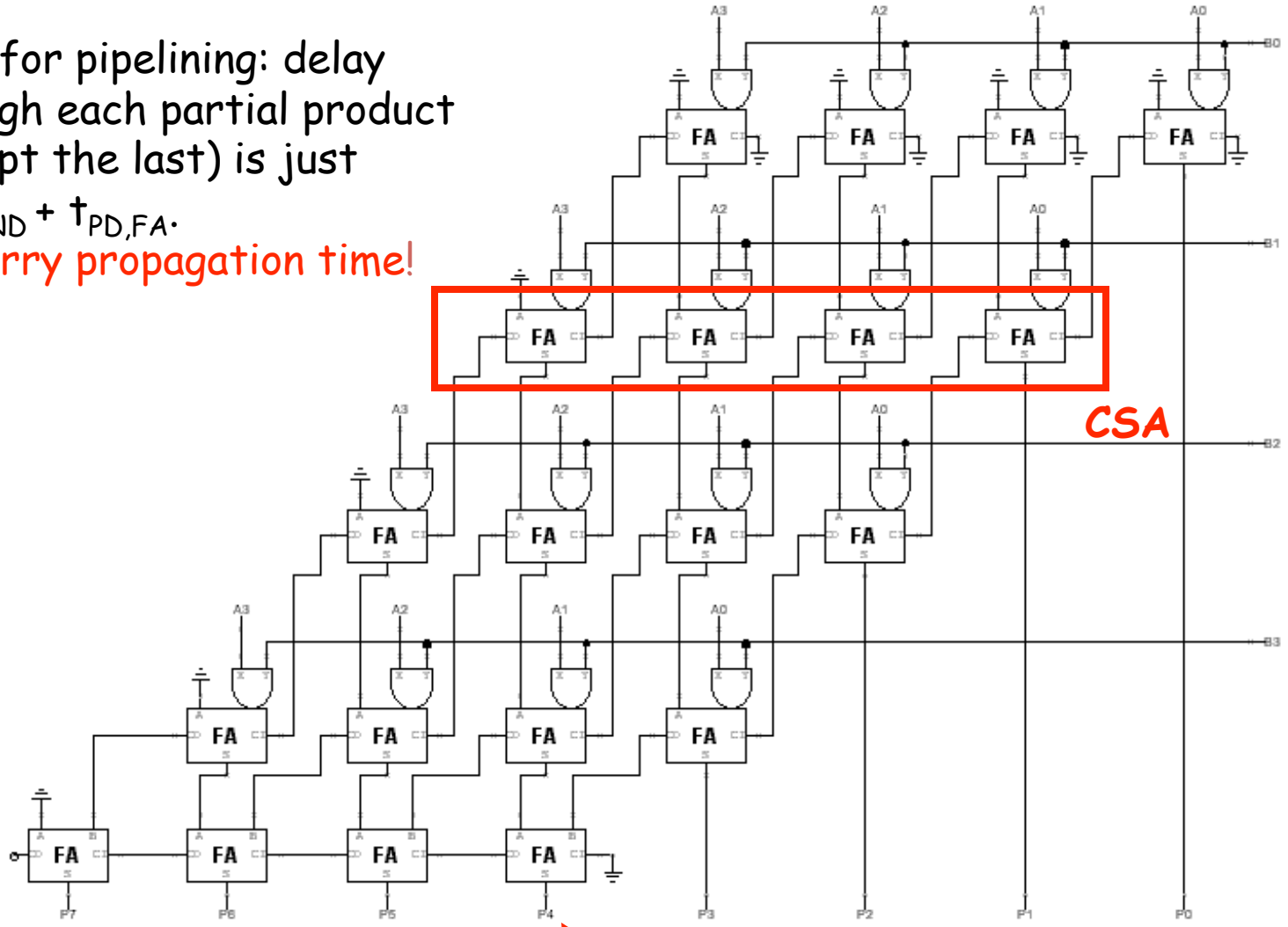



```
Init: P = 0; Load A,B

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
```

Useful building block: Carry-Save Adder

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

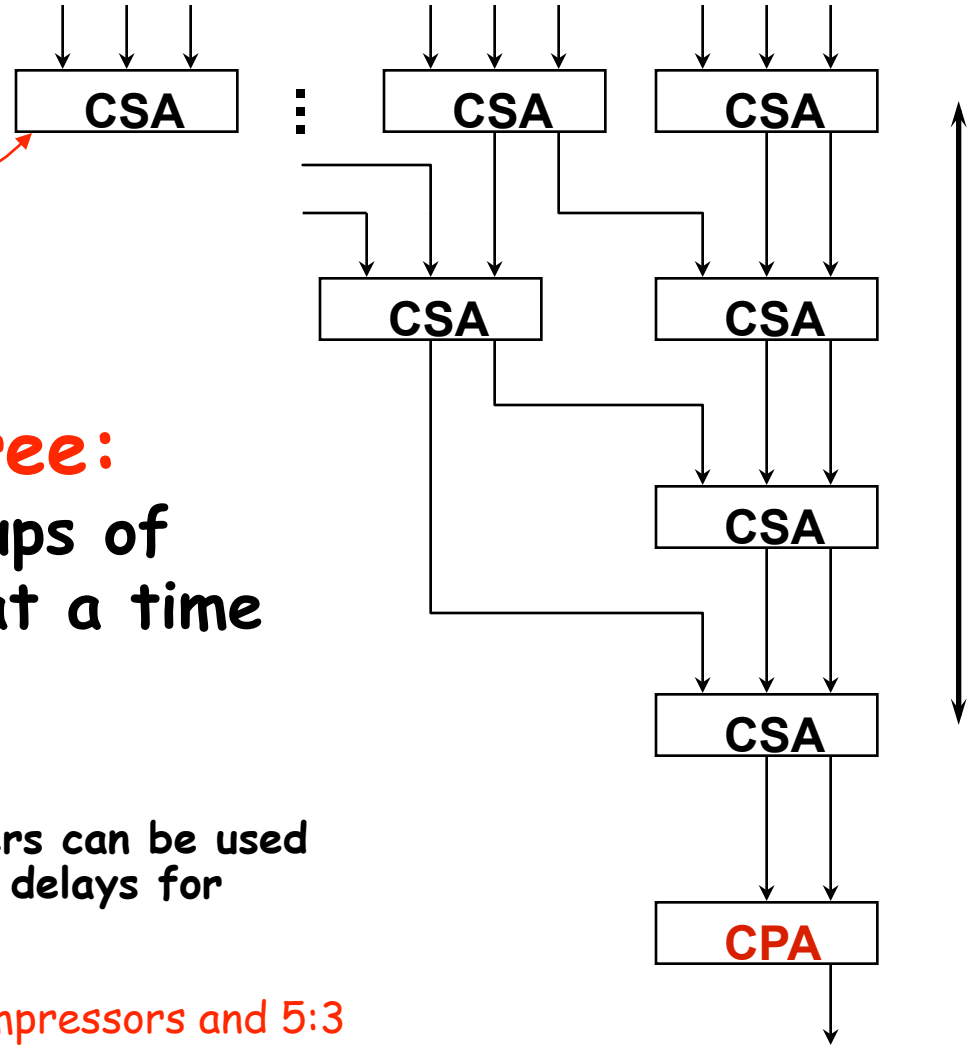
No carry propagation time!



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

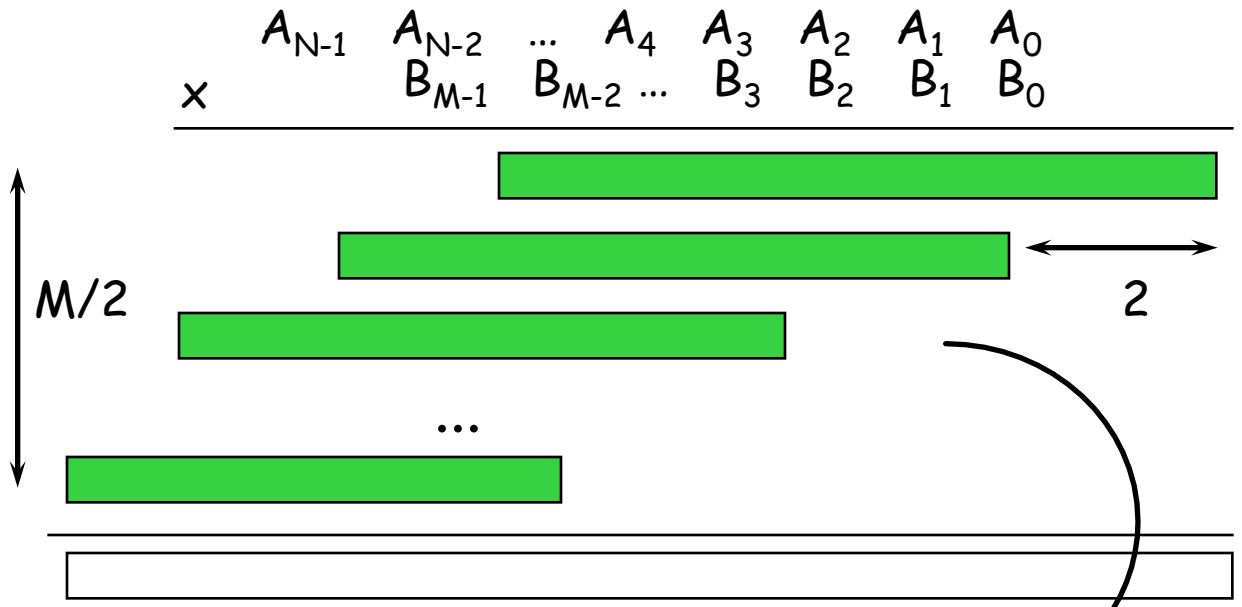
- If one of the operands is a constant, make it the multiplier (B in the earlier examples). For each "1" bit in the constant we get a partial product (PP) – may be noticeably fewer PPs than in the general case.
  - For example, in general multiplying two 4-bit operands generates four PPs (requiring 3 rows of full adders). If the multiplier is say, 12 (4'b1100), then there are only two PPs: 8*A+4*A (requiring 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.

$$\ldots 011110 \ldots = \ldots 100000 \ldots - \ldots 000010 \ldots = \ldots 010\overline{0}10 \ldots$$

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 rows 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

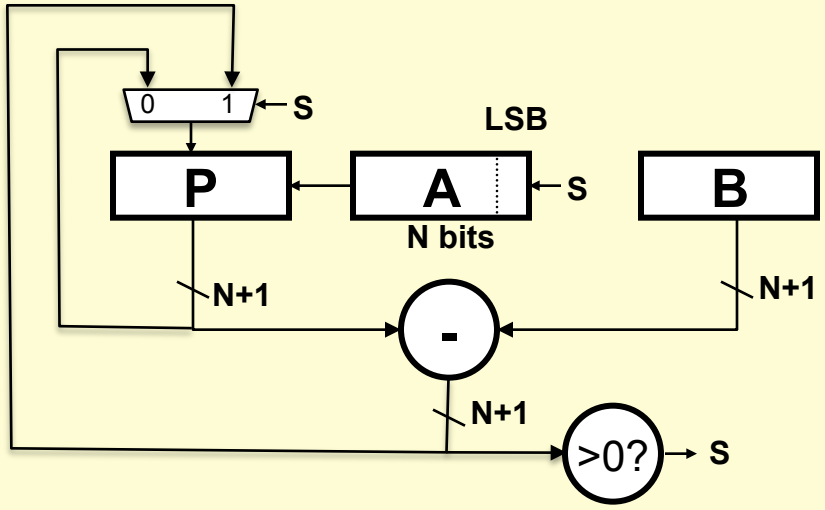
from previous bit pair

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!

Sequential Divider

Assume the Dividend (A) and the divisor (B) have N bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction each cycle and then cycle the circuit N times. This circuit works on unsigned operands; for signed operands one can remember the signs, make operands positive, then correct sign of result.




```
Init: P←0, load A and B
Repeat N times {
   shift P/A left one bit
   temp = P-B
   if (temp > 0)
     {P←temp, A_LSB←1}
   else A_LSB←0
}
Done: Q in A, R in P
```

Performance Metrics for Circuits

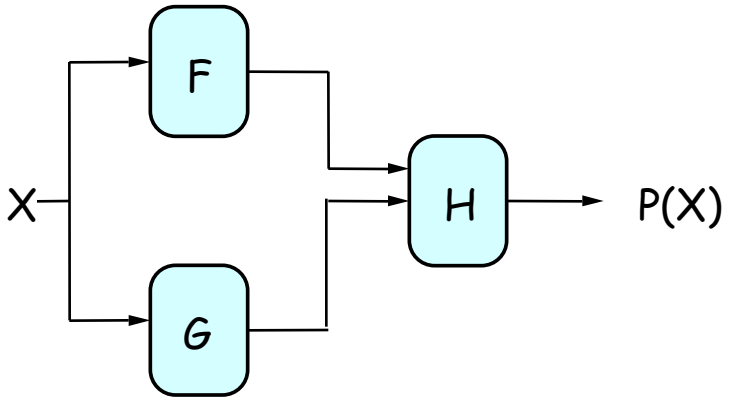
Circuit Latency (L): time between arrival of new input and generation of corresponding output.

For combinational circuits this is just t_{PD}.

Circuit Throughput (T): Rate at which new outputs appear.

For combinational circuits this is just $1/t_{PD}$ or 1/L.

Performance of Combinational Circuits

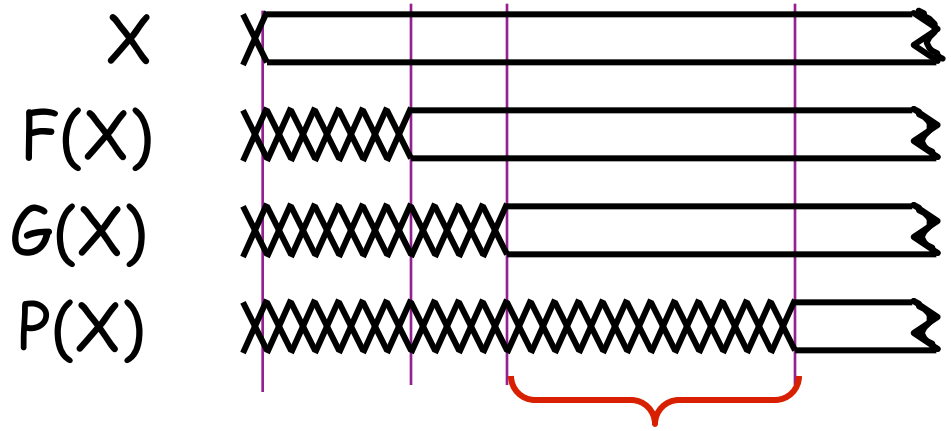


For combinational logic:

$L = t_{PD}$,

$T = 1/t_{PD}$.

We can't get the answer faster, but are we making effective use of our hardware at all times?

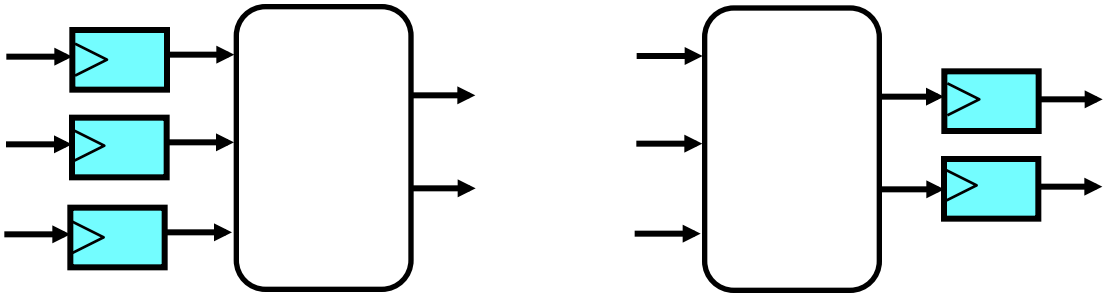


F & G are "idle", just holding their outputs stable while H performs its computation

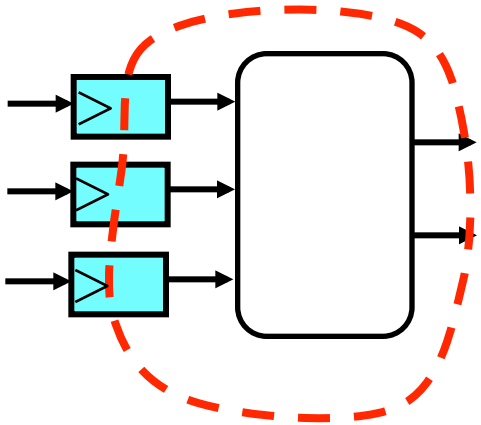
Retiming: A very useful transform

Retiming is the action of moving registers around in the system

- Registers have to be moved from ALL inputs to ALL outputs or vice versa

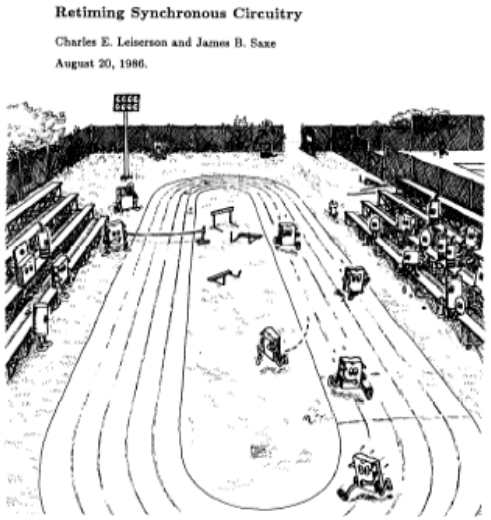


Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of the edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.

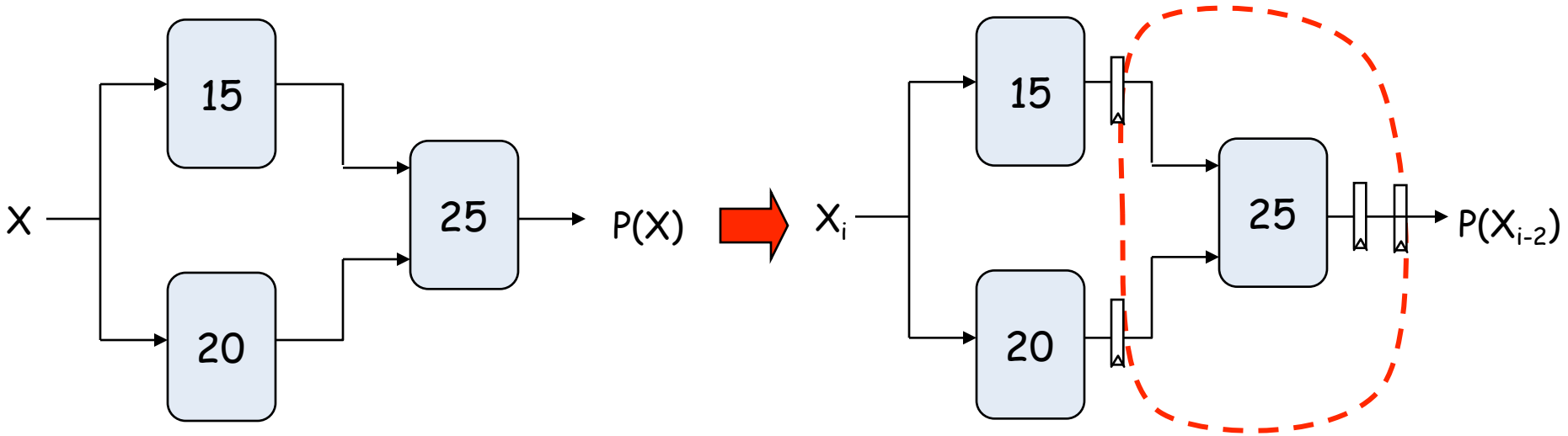


Benefits of retiming:

- Modify critical path delay
- Reduce total number of registers



Retiming Combinational Circuits aka "Pipelining"

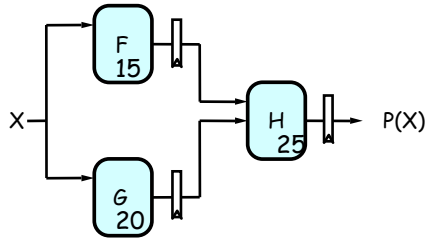


L = 45
T = 1/45

Assuming ideal registers:
i.e., $t_{PD} = 0$, $t_{SETUP} = 0$ →

$t_{CLK} = 25$
$L = 2*t_{CLK} = 50$
$T = 1/t_{CLK} = 1/25$

Pipeline diagrams



Clock cycle →

Pipeline stages ↓

	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	...
F Reg		$F(X_i)$	$F(X_{i+1})$	$F(X_{i+2})$	...
G Reg		$G(X_i)$	$G(X_{i+1})$	$G(X_{i+2})$	
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

DEFINITION:

a *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

ALWAYS:

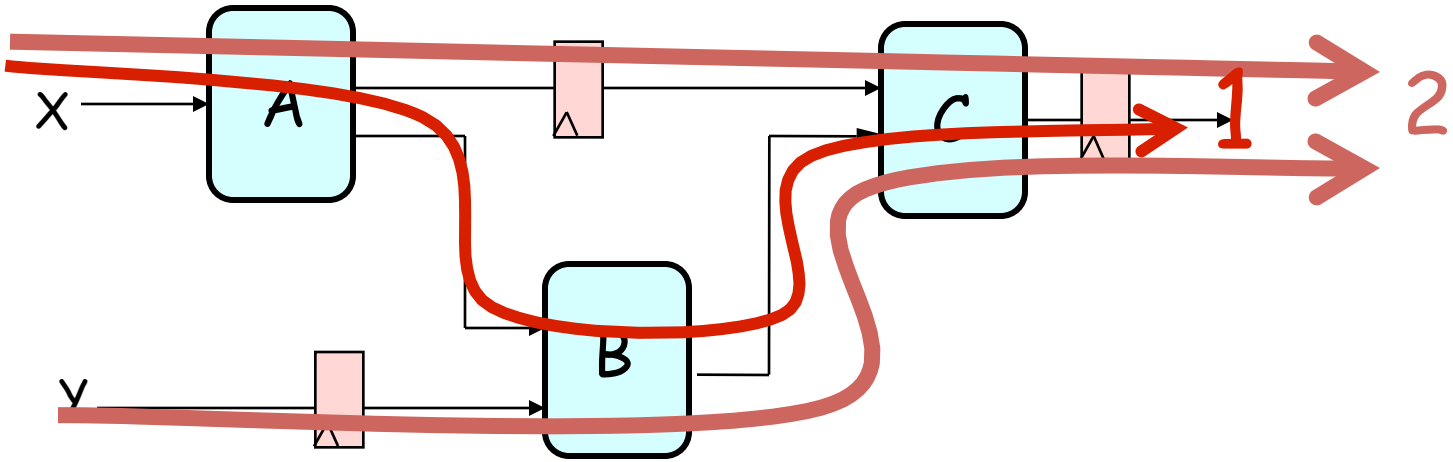
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

The LATENCY of a K-pipeline is K times the period of the clock common to all registers.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Ill-formed pipelines

Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline? __none__

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

A pipelining methodology

Step 1:

Add a register on each output.

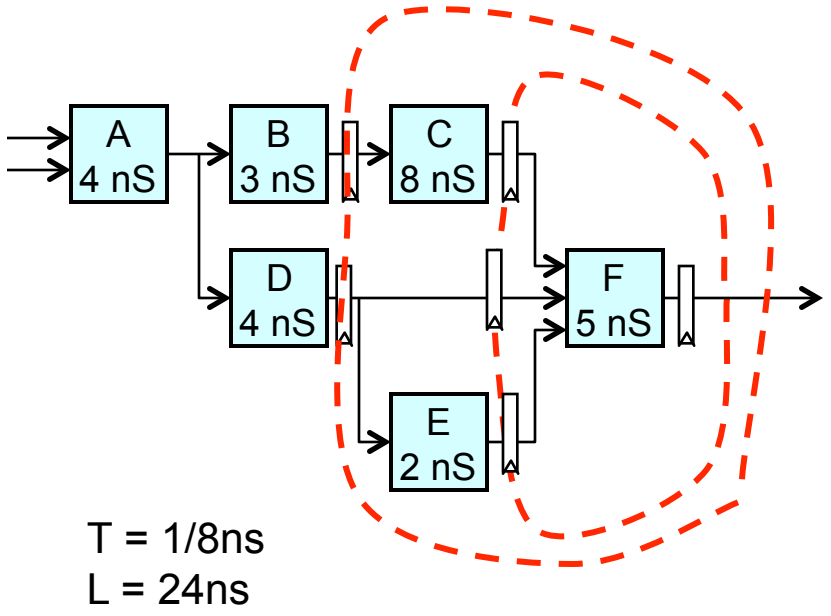
Step 2:

Add another register on each output. Draw a cut-set contour that includes all the new registers and some part of the circuit. Retime by moving regs from all outputs to all inputs of cut-set.

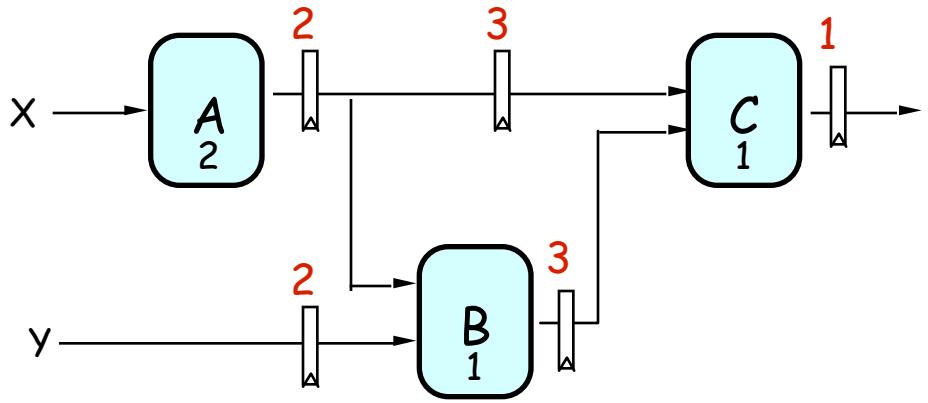
Repeat until satisfied with T.

STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Pipeline Example



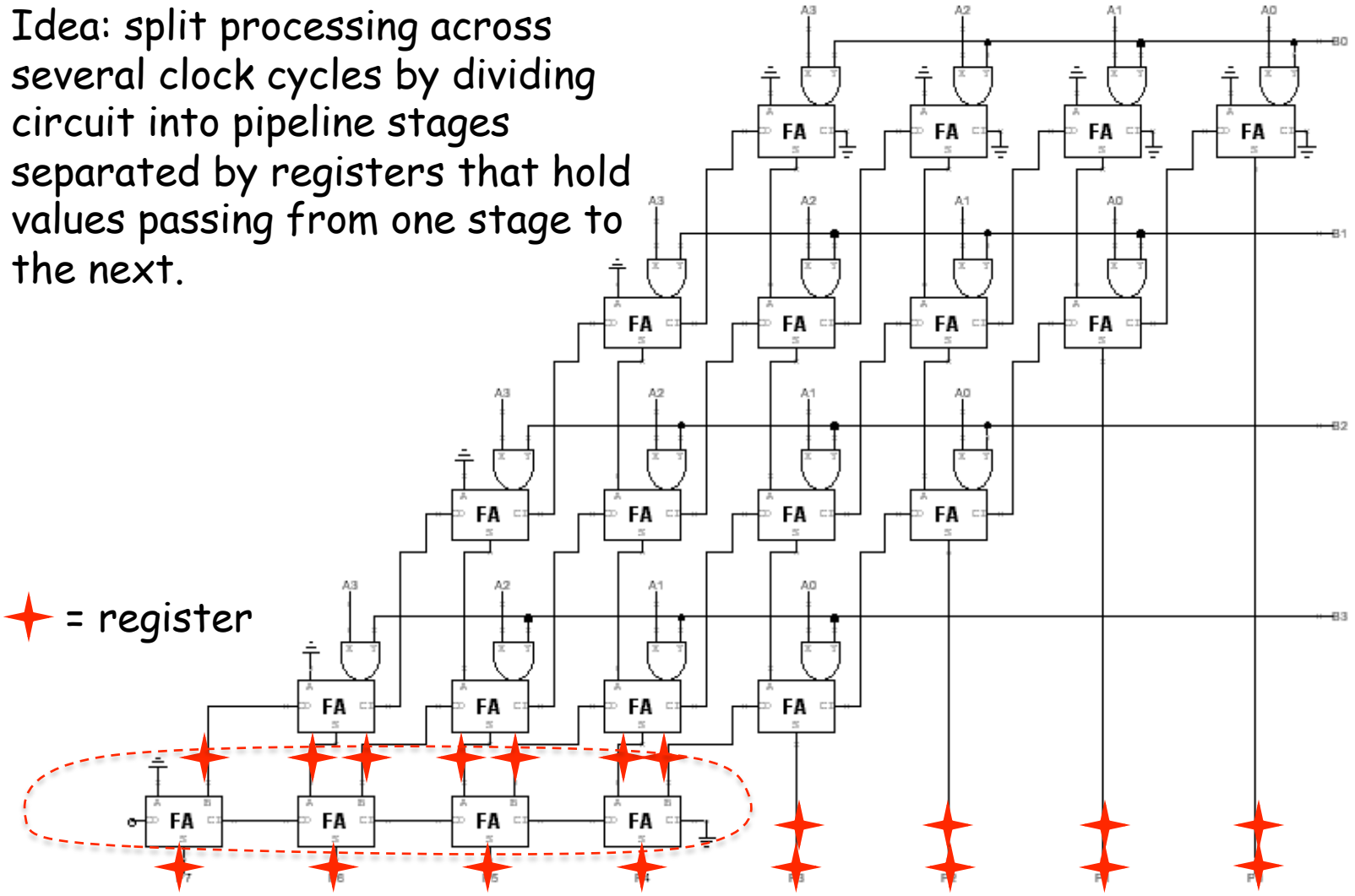
	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well -formed.

Increasing Throughput: Pipelining

Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.



= register

Throughput = $1/4t_{PD,FA}$ instead of $1/8t_{PD,FA}$)

How about $t_{PD} = 1/2t_{PD,FA}$?

= register