EITF35: Introduction to Structured VLSI Design

Part 2.1.1: Combinational circuit

Liang Liu
liang.liu@eit.lth.se

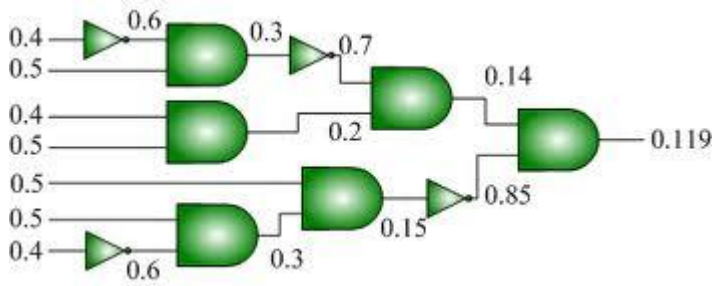
Why Called “Combinational” Circuits?

Combination

- In mathematics a combination is a way of selecting several things out of a larger group
- Select two fruits out of APPLE, PEAR, and ORANGE
- In a combination the **order** of elements is **irrelevant**

Combinational Circuits

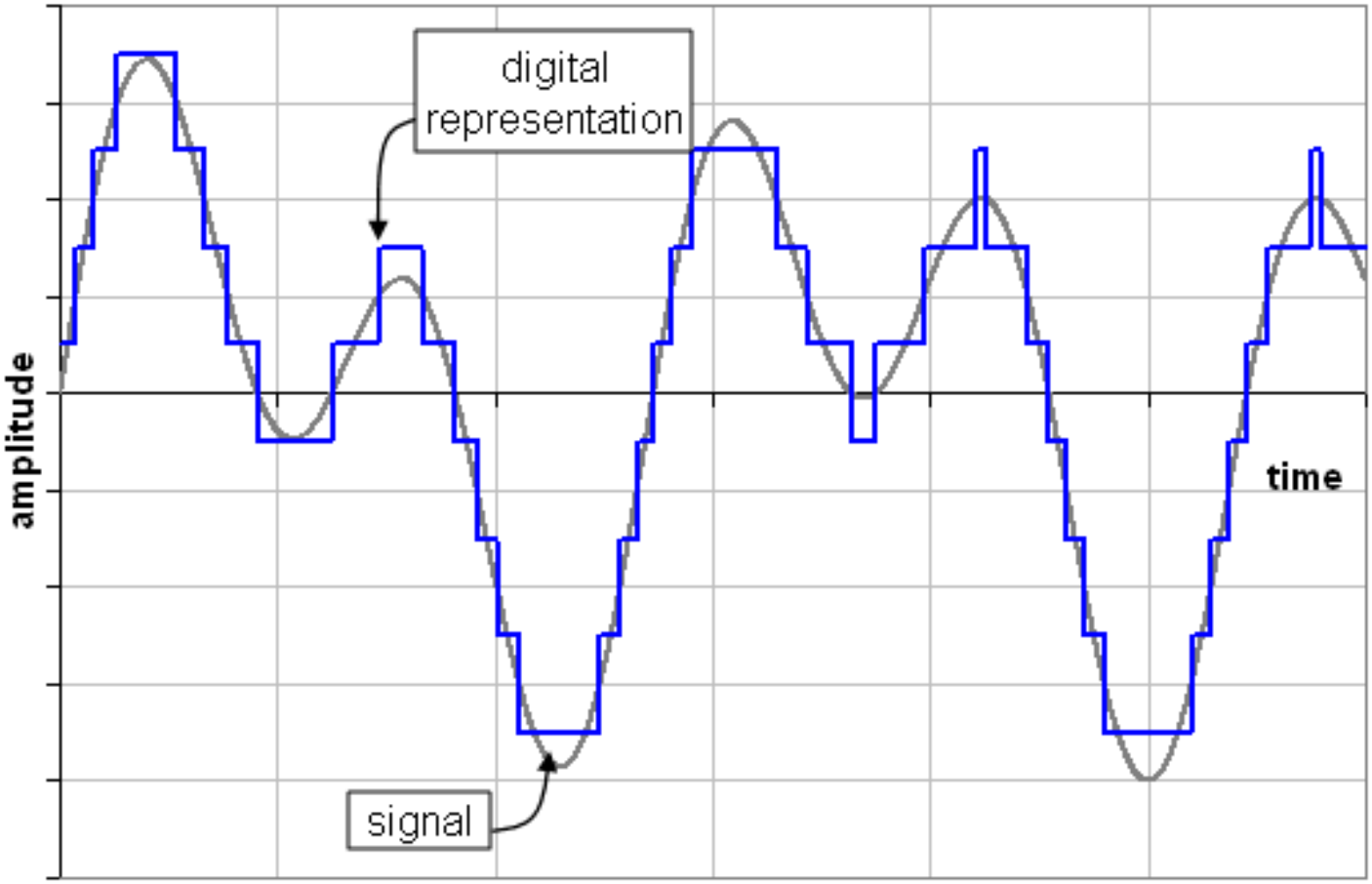
- ***time-independent logic***, where the output is a pure function of the present input only.
- the **order** of inputs doesn't matter for the outputs.



Overview

- Fixed-Point Representation
- Add/Subtract
- Multiplication
- Timing & Techniques to Reduce Delay

‘Digital’- quantization



Data Representation

Unsigned

- Uses the two numbers from 0 to 1
- Every column represents a power of 2
- Unsigned integer:

$$\sum_{i=0}^{n-1} \text{bit}_i 2^i$$

Signed (Two's complement)

- *N*-bit number is defined as the complement with respect to 2^N, in other words the result of **subtracting the number from 2^N**
- 2's complement of a binary number involves inverting all bits and **adding 1**

$$\text{bit}_{n-1}(-2^{n-1}) + \sum_{i=0}^{n-2} \text{bit}_i 2^i$$

$$11110100_2 = -12_{10}$$

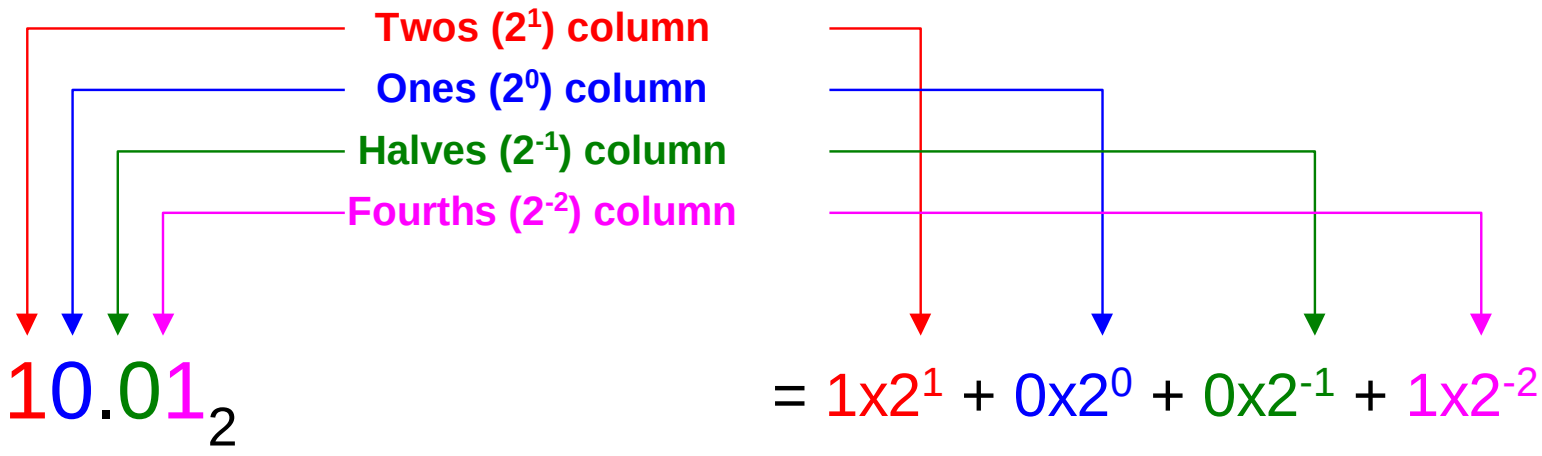
Sign bit **2's complement**

8-bit Signed/Unsigned Integers

Signed overflow ↑	-128	1000 0000	
	-127	1000 0001	
	…	…	
		1111 1100	
MSB defines sign		1111 1101	
	-2	1111 1110	
	-1	1111 1111	
Signed integers	0	0000 0000	0
	1	0000 0001	1
	2	0000 0010	2
	3	0000 0011	3
	…	…	…
	126	0111 1110	126 Unsigned integers
Signed overflow ↓	127	0111 1111	127
		1000 0000	128
		1000 0001	129
		…	…
		1111 1110	254
		1111 1111	255 Unsigned overflow ↓

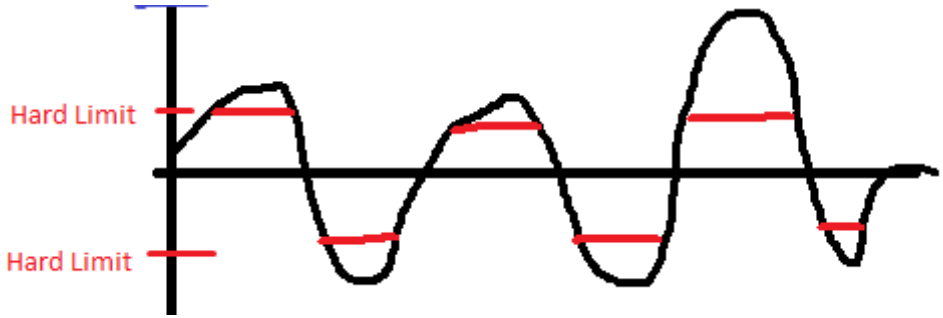
General Fixed-Point Representation

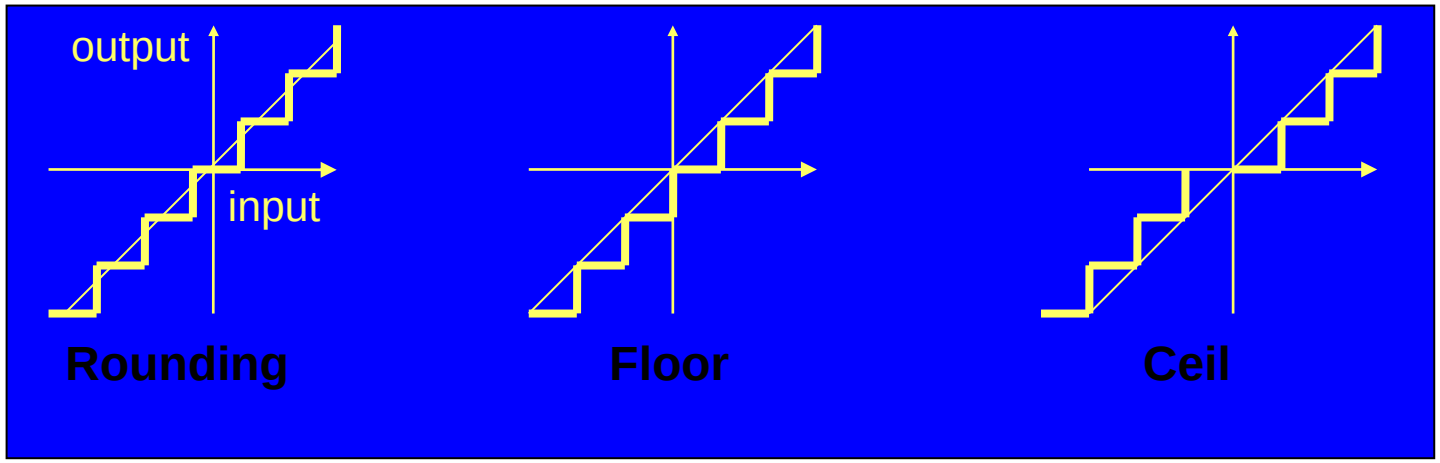
- ***Qm.n* notation**
  - m bits for integer portion, n bits for fractional portion
  - Total number of bits N = m + n + 1, for signed numbers
- **Example: 16-bit number (N=16) and Q2.13 format**
  - 2 bits for integer portion,13 bits for fractional portion,1 signed bit (MSB)
- **Special cases:**
  - 16-bit integer number (N=16) => Q15.0 format
  - 16-bit fractional number (N = 16) => Q0.15 format



Finite Word-Length Effect

- **Overflow**
  - Saturation
- **Quantization error**
  - Round
  - Truncation





round(0.51)=1 ***floor(0.51)=0*** ***ceil(0.49)=1***

Exercise (1 min)

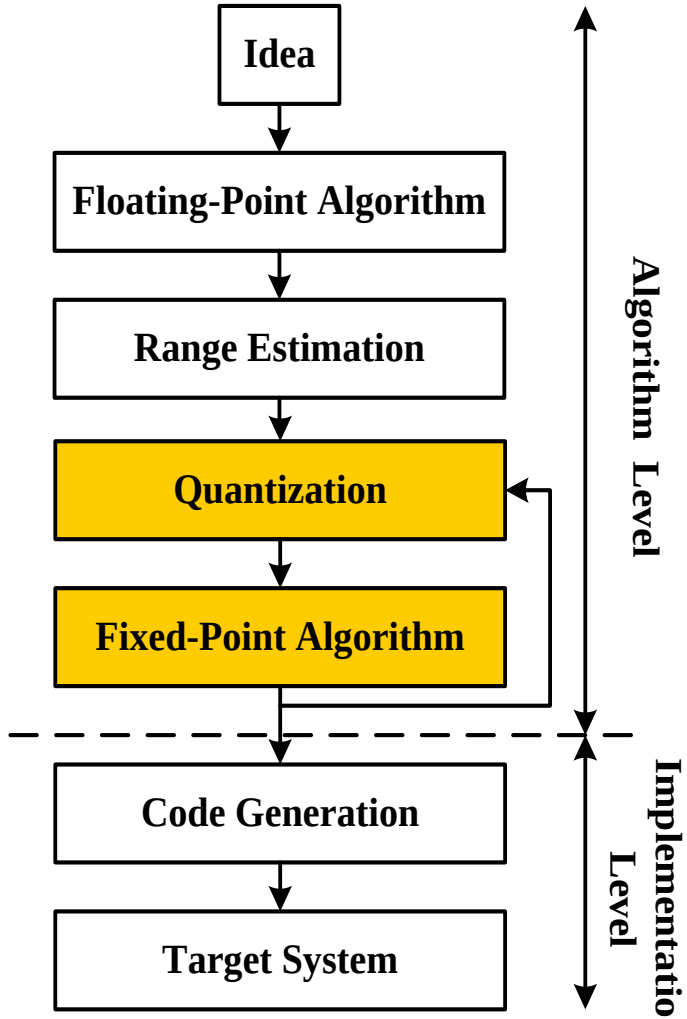
- 1.234, only 3 bits for the fraction part
- Do Round Truncation
- Do Floor Truncation

*round(1.234*2^3)/2^3=1.25*

*floor(1.234*2^3)/2^3=1.125*

Fixed-Point Design

- **Digital signal processing algorithms**
  - Often developed in floating point
  - Later mapped into **fixed point** for digital hardware realization
- **Fixed-point digital hardware**
  - Lower area
  - Lower power
  - Higher quantization error



Design Trade-off

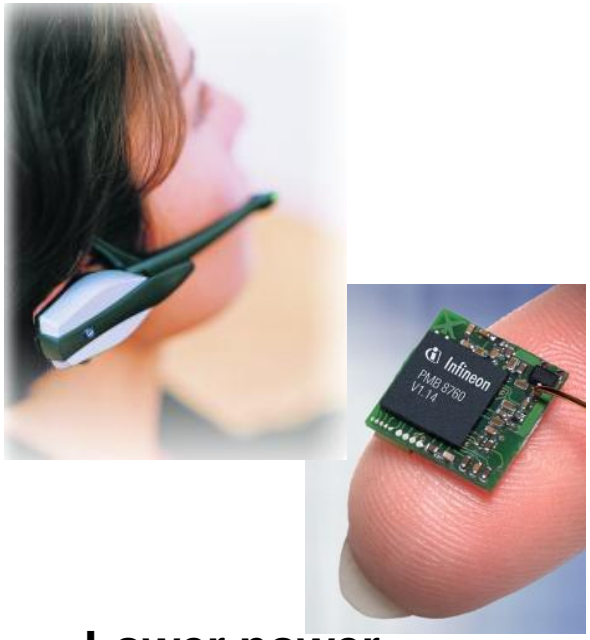
Implement the best HW realization. Best??

Flexibilty
Complexity

- Processors
- FPGAs

Low power
Low cost
Flexibilty

- Processors
- Dedicated HW



Lower power
Lower cost

- Dedicated HW
- Processors

Design Trade-off

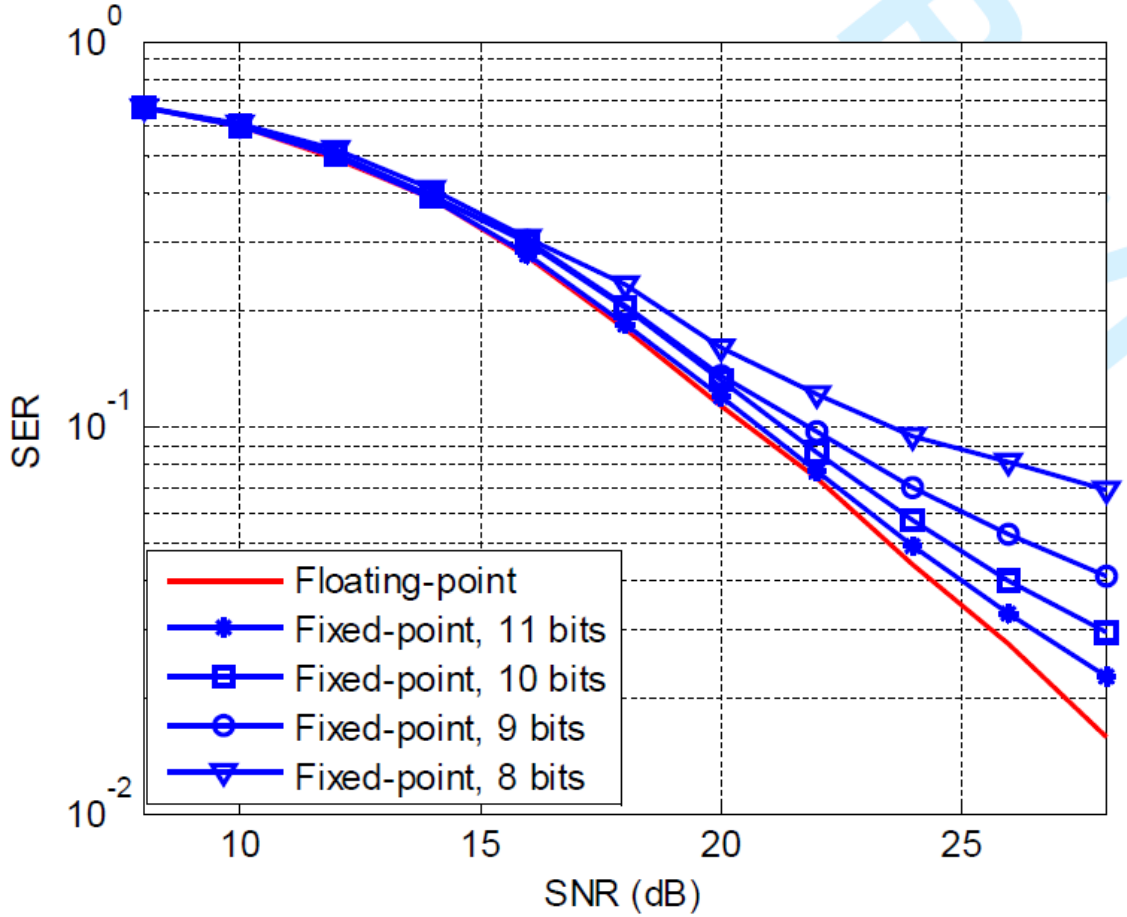
Implement the best HW realization. Best??

Different applications, different demands...
Thus, ”*just good enough*” is the best in engineering.

Try to find a balance between effort and cost!

Optimum Word-Length

- Range Analysis
- Fixed-point Simulation



Will learn more in DSP-Design course

QoS requirements of different applications

High-Quality Voice → BER ≤ 2×10^{-3}

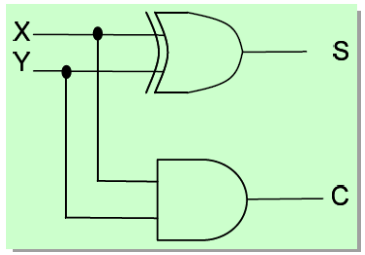
Video Program → BER ≤ 2×10^{-4}

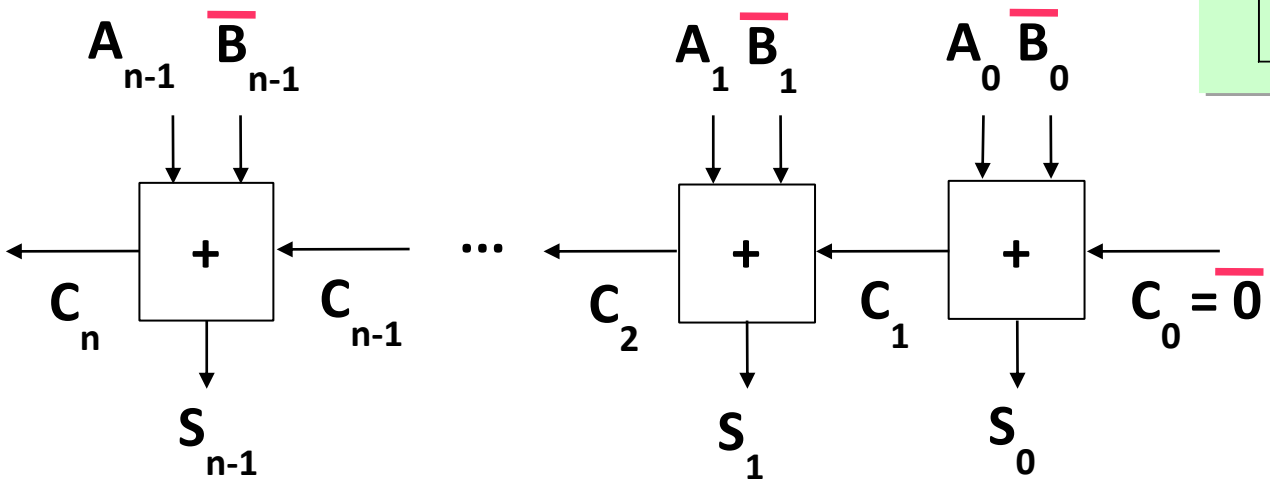
File Transfer → BER < 10^{-6}

Overview

- Fixed-Point Representation
- **Add/Subtract**
- Multiplication
- Timing&Techniques to Reduce Delay

Add/Subtract





- The HW for sum/difference (S) does not care about signed/unsigned
- Unsigned overflow = Carry-out & add
- Signed overflow = $C_n \oplus C_{n-1}$
- True sign = $S_{n-1} \oplus$ signed overflow = $(A_{n-1} \oplus B_{n-1} \oplus C_{n-1}) \oplus (C_n \oplus C_{n-1}) = A_{n-1} \oplus B_{n-1} \oplus C_n$

Unsigned Overflow Examples

- 4-Bit unsigned addition

10+6 = 16, outside [0..15]

$$\begin{array}{rr} & 1010 \\ & \underline{+0110} \\ C_4=1 & 0000 \end{array}$$

$C_n = C_4 = 1$ & add $\Leftrightarrow$ Unsigned overflow

Carry-out & add $\Leftrightarrow$ Unsigned overflow

Signed Overflow Example

4-Bit signed addition

6+7 = 13, outside [-8..7]

$$\begin{array}{rr} & 0110 \\ & +0111 \\ \hline C_4=0 & 1101 \end{array}$$

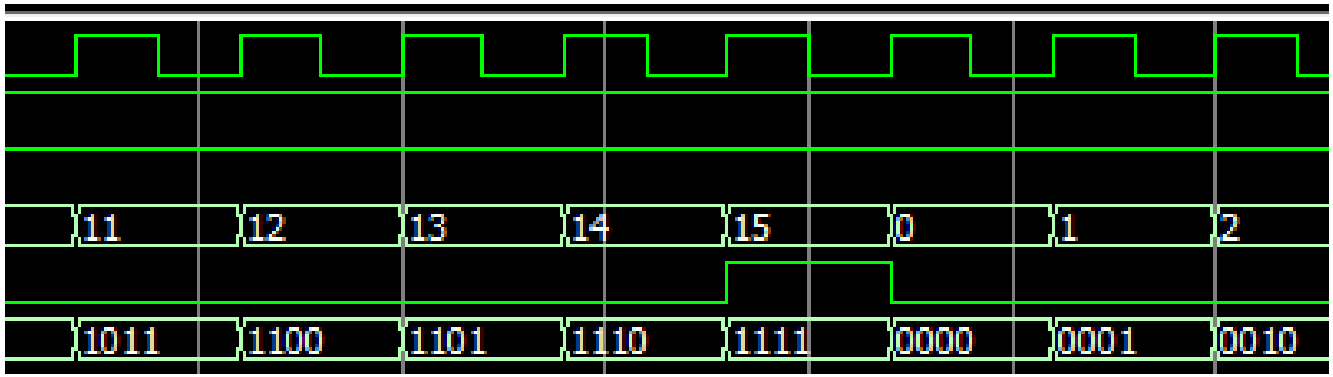
$C_3 = 1$

$C_n \oplus C_{n-1} = C_4 \oplus C_3 = 0 \oplus 1 = 1 \Leftrightarrow$
Carry-outs different $\Leftrightarrow$ Signed overflow

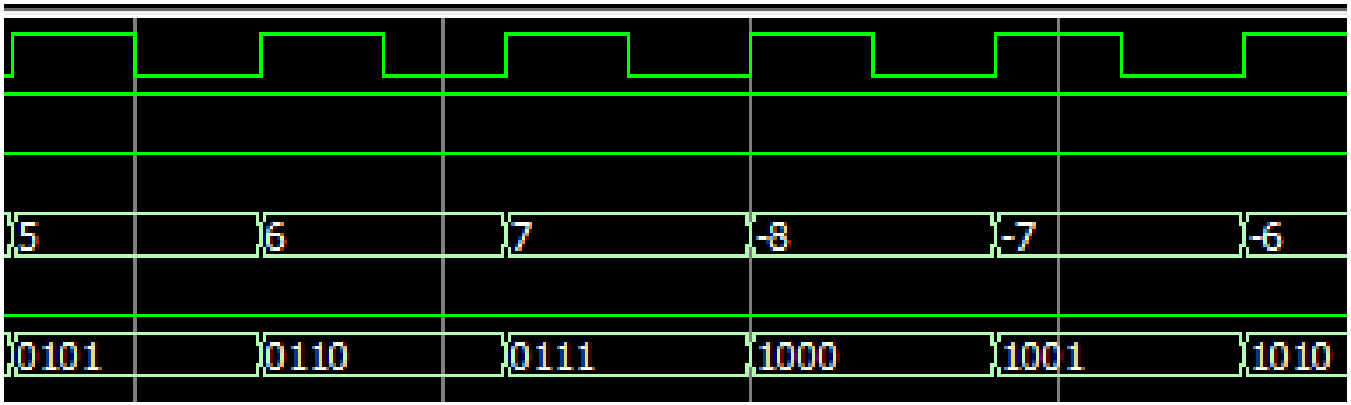
$S_{n-1} \oplus$ signed overflow =
$A_{n-1} \oplus B_{n-1} \oplus C_n = A_3 \oplus B_3 \oplus C_4 = 0 \oplus 0 \oplus 0 = 0 \Leftrightarrow$ True sign = Positive/zero

Overflow in Hardware

- **Hardware does not take care of the overflow for you**
  - Unsigned



  - Signed



Two's Complement Signed Extension

- **To add two numbers, we must represent them with the same number of bits: 0100+11100**
  - If we just pad with zeroes on the left:

4-bit		8-bit	
0100	(4)	00000100	(still 4)
1100	(-4)	00001100	(12, not -4)

  - Instead, replicate the MS bit -- the sign bit:

4-bit		8-bit	
0100	(4)	00000100	(still 4)
1100	(-4)	11111100	(still -4)

- **Extend one sign-bit to protect overflow**

Decimal Mark in Hardware

- **Matlab aligns the decimal mark automatically**

  1.32+100.2343= 101.5543

- **Hardware does NOT**
  - Decimal mark is just a concept

    01.100+001.01=?

    10001

  - You need to align the decimal mark manually

    001.100+001.010=010.110

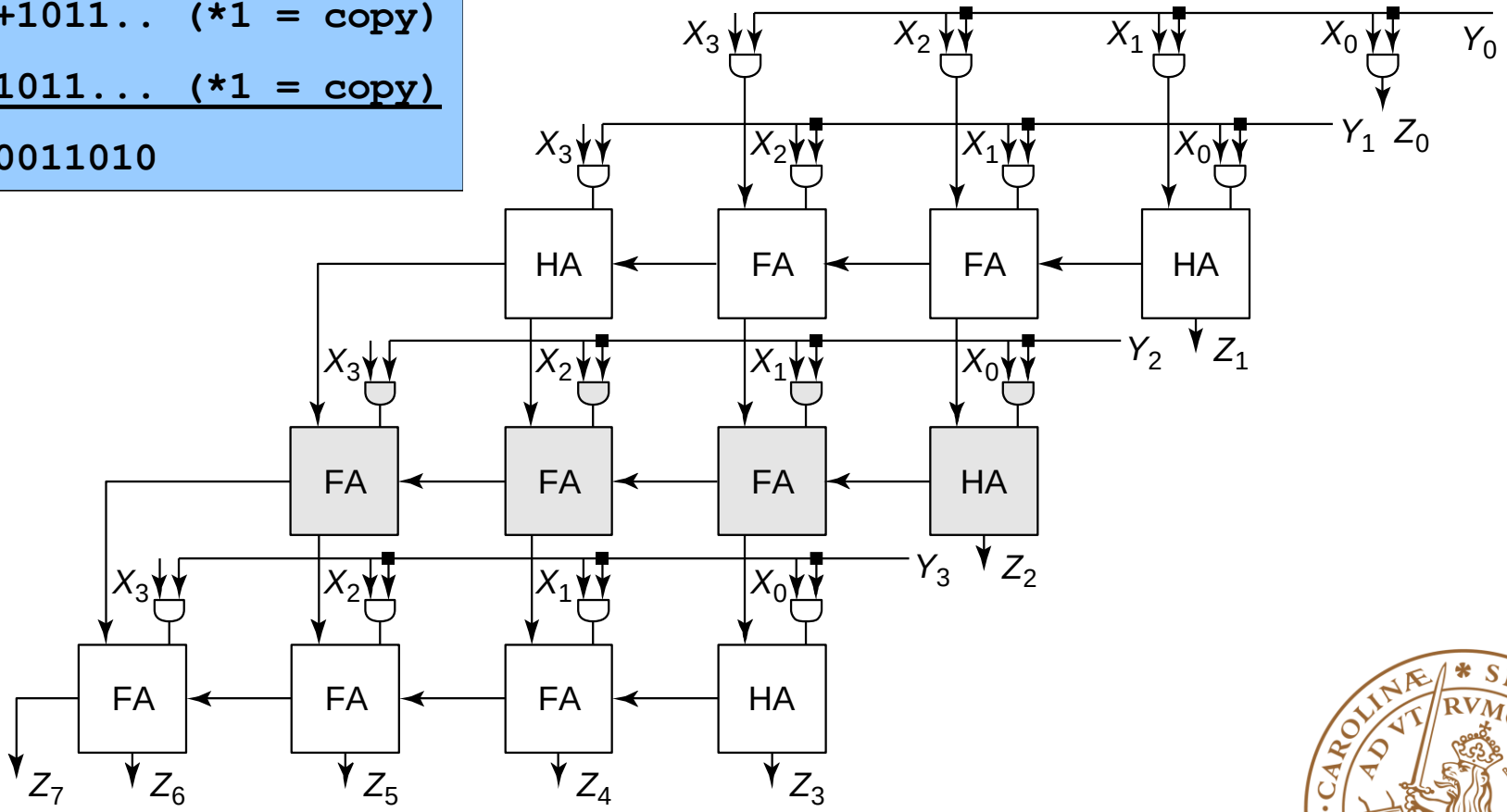
Overview

- Fixed-Point Representation
- Add/Subtract
- **Multiplication**
- Timing & Techniques to Reduce Delay

Array Multiplier (unsigned)

```
      1011 * 1110
      ---------
      0000  (*0 = zero)
     +1011.  (*1 = copy)
    +1011..  (*1 = copy)
   +1011...  (*1 = copy)
   ---------
   10011010
```

- **Direct Mapping**
  - ***Horizontal :*** partial product using AND
  - ***Vertical :*** shift-add of partial product



Don't Forget ... Signed Multiplication

1 0 1 1	-5x
0 0 1 1	+3

?

1 1 1 1 0 0 0 1	-15

Signed Multiplication

- **Either transform to multiply of non-negative integers:**
  - Record signs and negate any negative factors.
  - Perform unsigned multiplication.
  - Negate product if signs above differ.

abs(-5)=5
abs(3)=3

```
          0 1 0 1     +5x
          0 0 1 1     +3
          ---------
          0 1 0 1
        0 1 0 1
      0 0 0 0
    0 0 0 0
    0 0 0 1 1 1 1     +15
    ---------------
```

-1*1*15=-15

Signed Multiplication

- Or directly perform signed multiplication:
  - **Multiplier**: positive
  - **Multiplicand**: positive or negative
  - **Sign extend** the partial products when adding up

```
          1 0 1 1    -5x
        0 0 1 1      +3
      ---------
1 1 1 1 1 0 1 1
1 1 1 1 0 1 1
0 0 0 0 0 0
0 0 0 0 0
-----------------
1 1 1 1 0 0 0 1    -15
```

Signed Multiplication

- **Or directly perform signed multiplication:**
  - Multiplier & Multiplicand: positive or negative
  - Sign extend the partial products when adding up
  - Subtract instead of adding last partial product

Ex:

```
  - 5                1011    multiplicand
x - 3              x 1101    multiplier
-----              ------
  15               00000     partial product
                   11011     shifted multiplicand
                   ------
Added bit using    111011    partial product
sign extension     00000     shifted multiplicand
                   ------
                  1111011    partial product
                   11011     shifted multiplicand
                   ------
                 11100111    partial product
                   00101     shifted and negated multiplicand
                   ------
                 00001111    product
```

Multiplier in Xilinx FPGA

Embedded Multipliers

- 18-bit embedded multipliers (signed and unsigned)
- Using Embedded Multipliers in Spartan-3 FPGAs

  http://www.xilinx.com/support/documentation/application_notes/xapp467.pdf

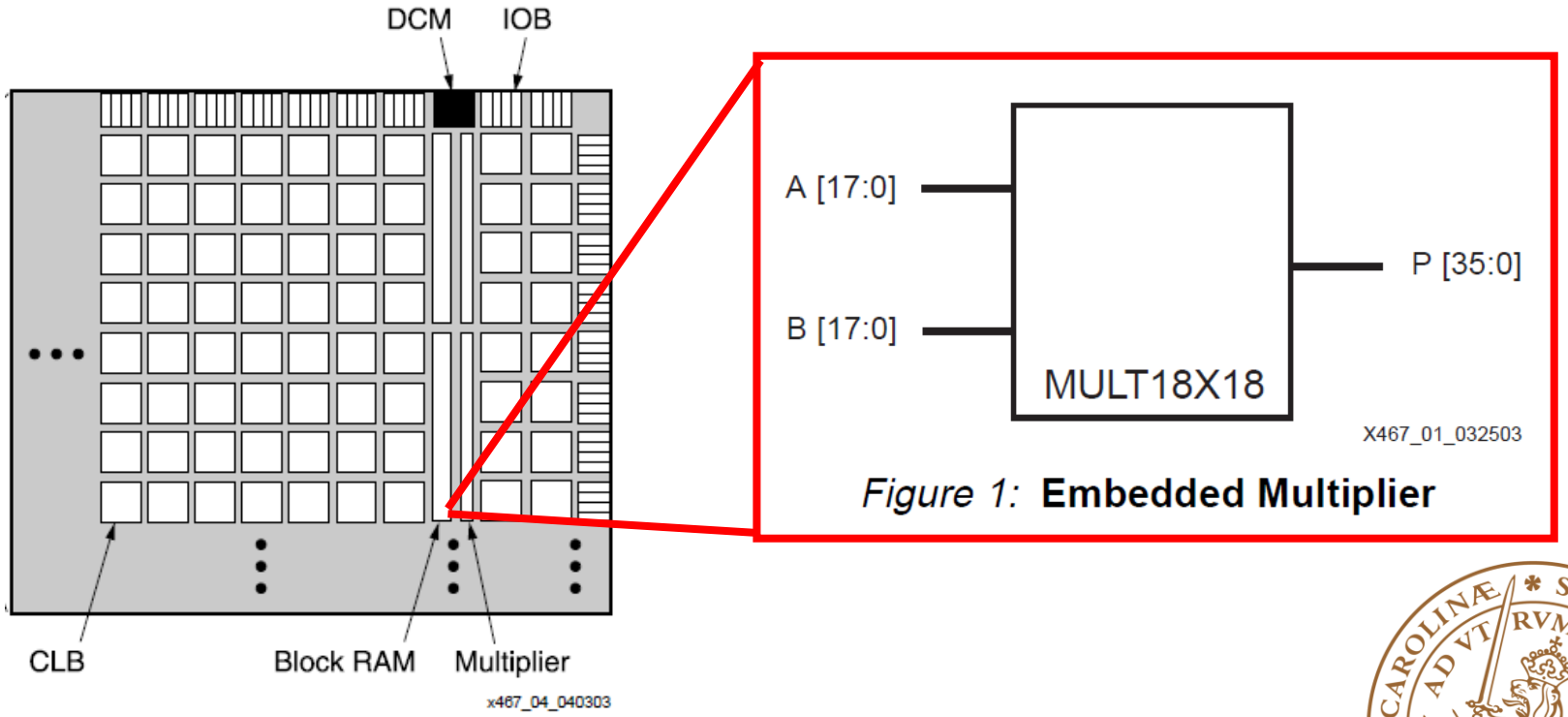


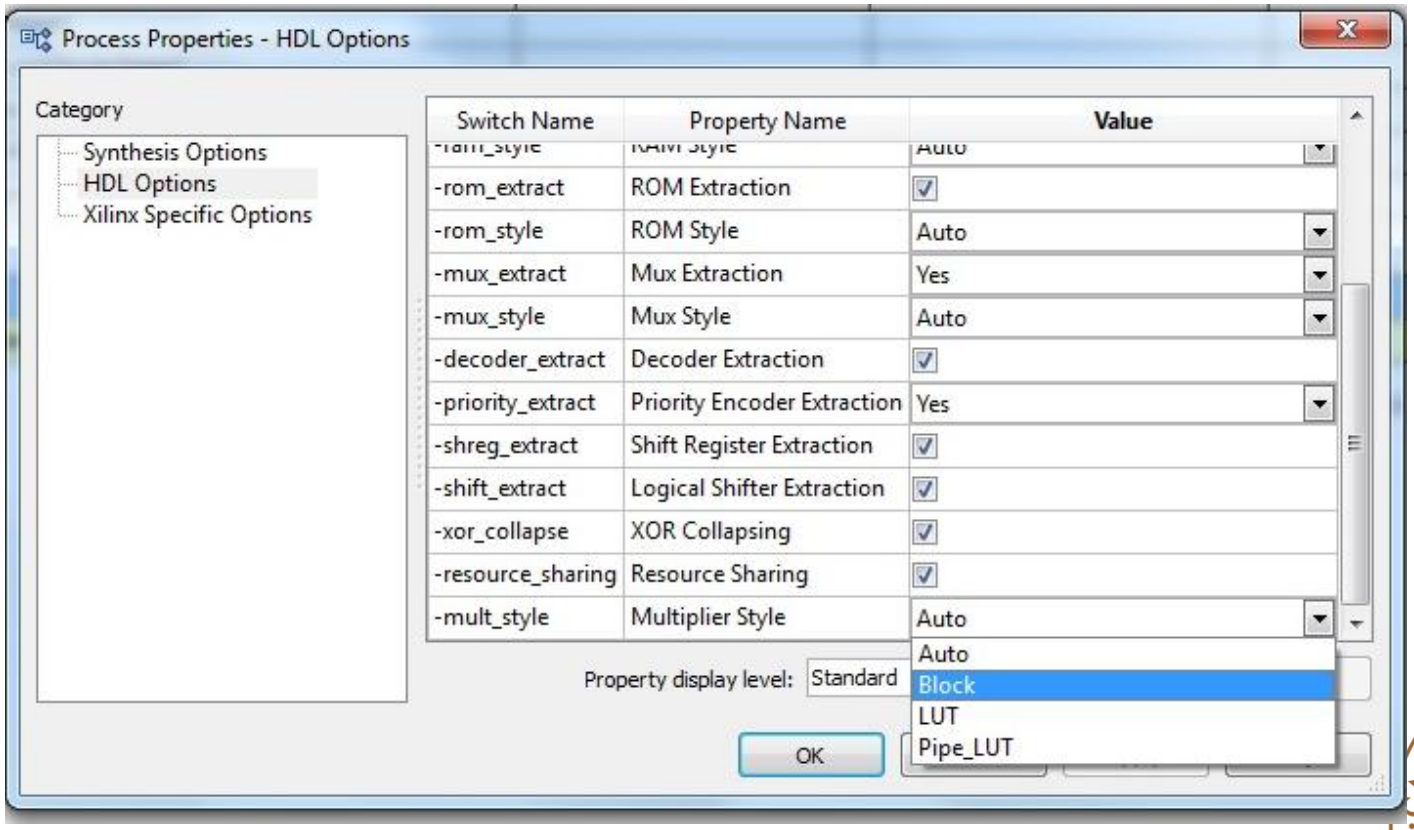
Figure 1: **Embedded Multiplier**

Multiplier in Xilinx FPGA

Embedded Multipliers

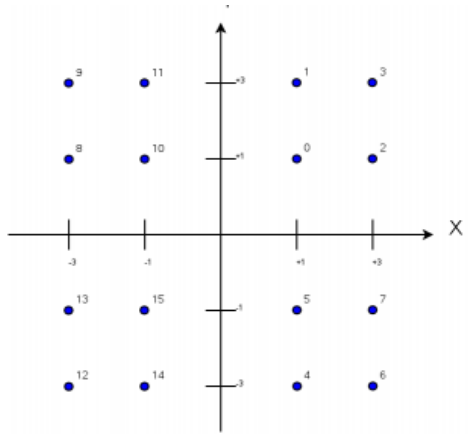
- 18-bit embedded multipliers (signed and unsigned)
- Using Embedded Multipliers in Spartan-3 FPGAs

  http://www.xilinx.com/support/documentation/application_notes/xapp467.pdf



Multiplication by a Constant

- **Examples:**
  - **Twiddle factor** in FFTs
  - **Constellation points** in wireless communication



$$y_j = \sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n} jk} x_k$$

- **The synthesis engine may be not smart enough to do some evident optimizations.**
- **As a designer you need to assure that multiplications with a small constant is accomplished by a number of shifts & adds**

Some numerical examples:

2 (10_2): multiplicand << 1

3 (11_2): multiplicand << 1 + multiplicand

5 (101_2): multiplicand << 2 + multiplicand

255 (11111111_2): ?

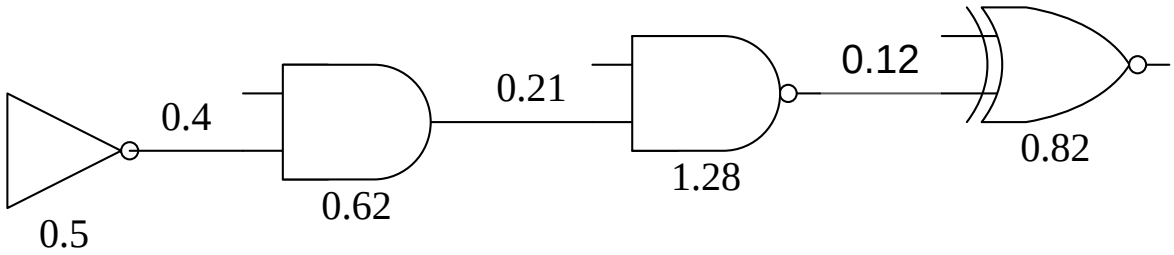
multiplicand << 8 – multiplicand

Overview

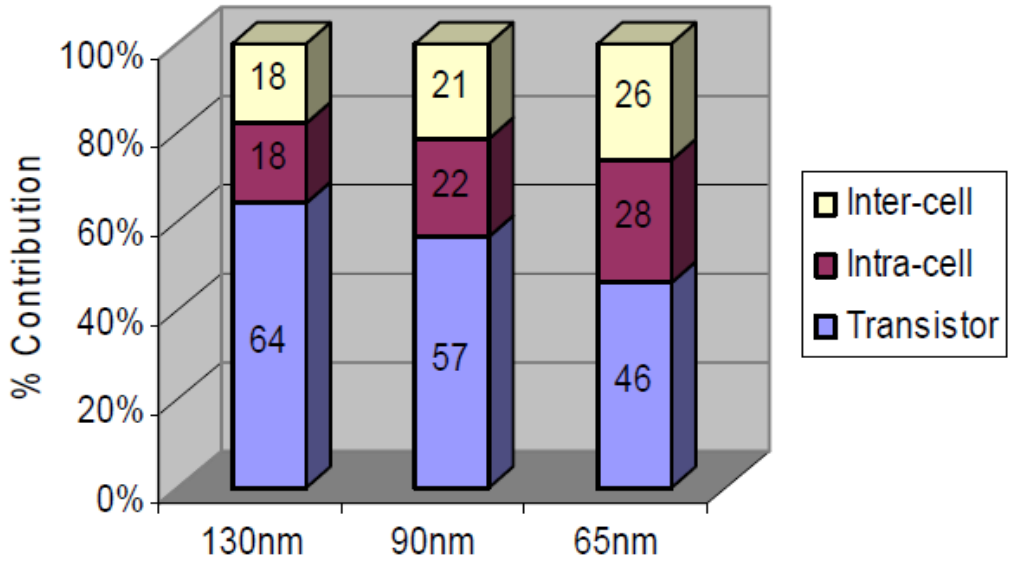
- Fixed-Point Representation
- Add/Subtract
- Multiplication
- **Timing & Techniques to Reduce Delay**

Combinational Circuit Timing

- **Path delay = cell delay + net delay**

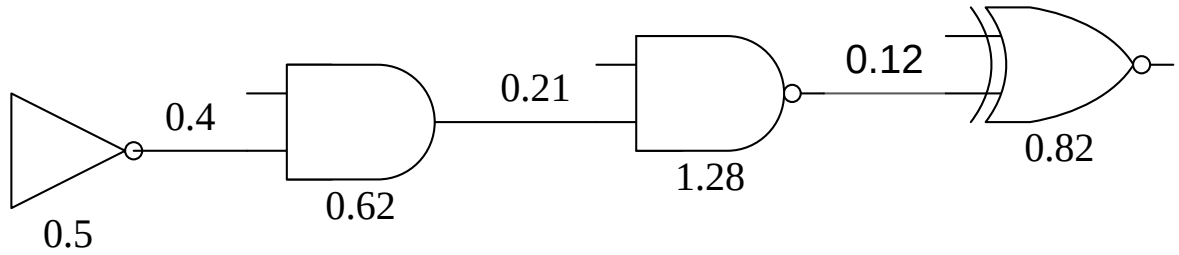


Path Delay = 0.5+0.4+0.62+0.21+1.28+0.12+0.82=3.95 ns

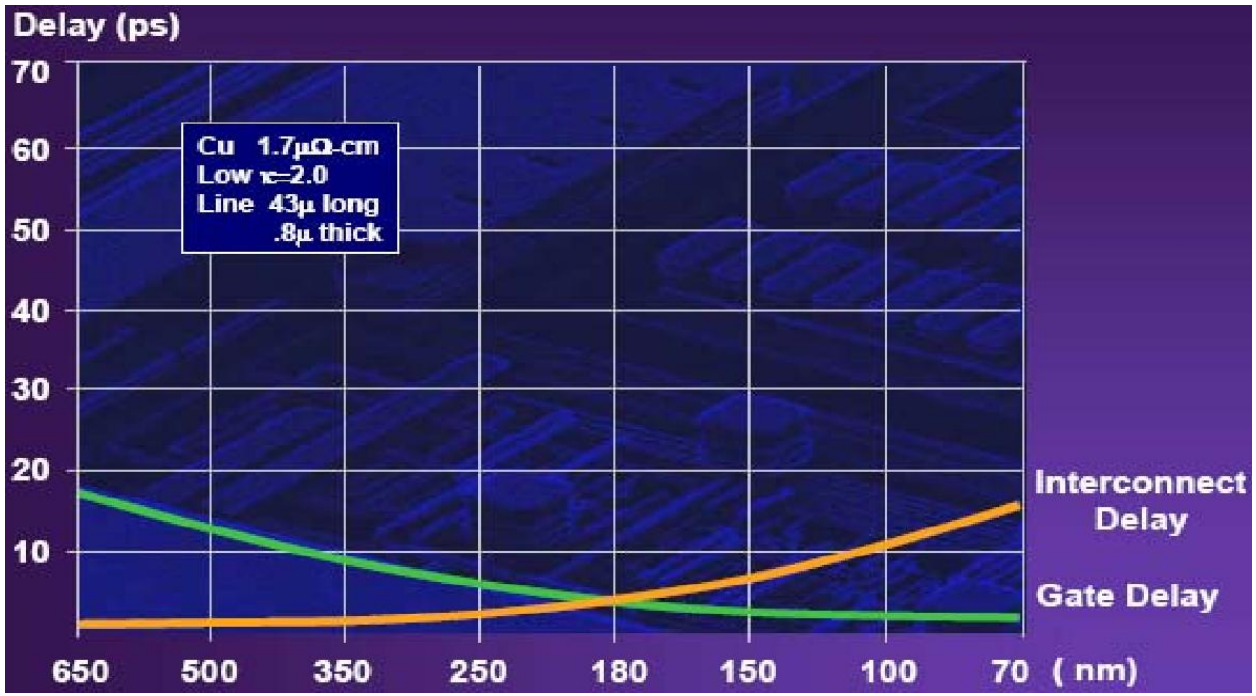


Combinational Circuit Timing

- **Path delay = cell delay + net delay**

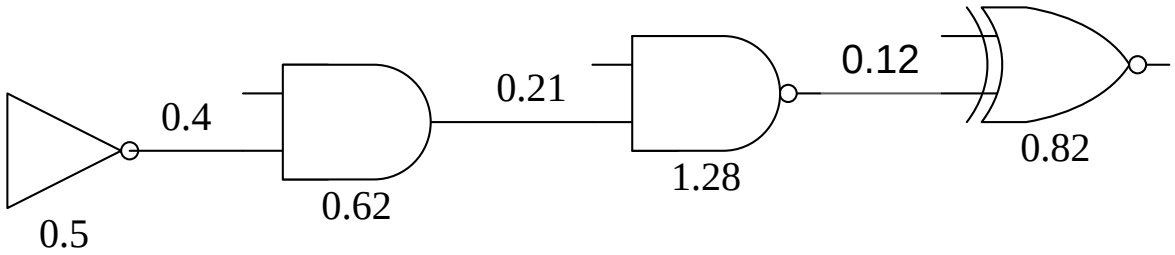


Path Delay = 0.5+0.4+0.62+0.21+1.28+0.12+0.82=3.95 ns



Combinational Circuit Timing

- **Path delay = cell delay + net delay**

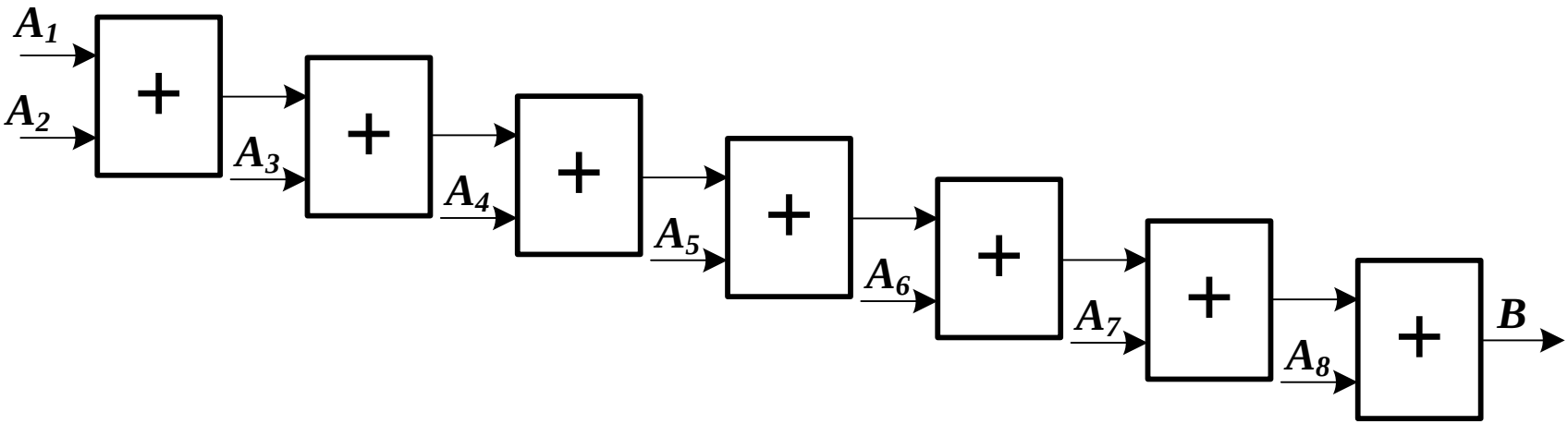


Path Delay = 0.5+0.4+0.62+0.21+1.28+0.12+0.82=3.95 ns

- **How to reduce processing delay**
  - Reduce cell delay? Standard-cell library
  - Reduce net delay? Place & Route
  - **Or we can change the architecture**

Example1: Higher-Level Adder Chain

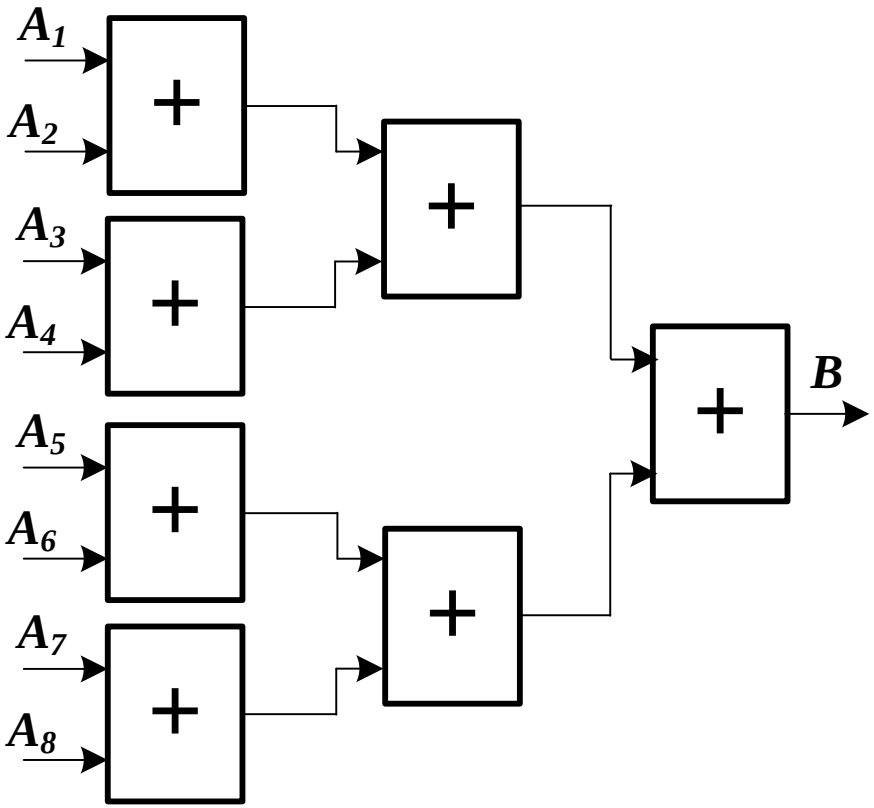
- **Calculate:** $B = A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8$



Cascaded-Chain

Higher-Level

$$B = [(A_1 + A_2) + (A_3 + A_4)] + [(A_5 + A_6) + (A_7 + A_8)]$$



Tree

Cascade vs. Tree

Comparison of n-input adder

- Cascading chain:
  - Area: (n-1) full adder
  - Delay: (n-1)
  - Flexibility: easy to modify (scale)
- Tree:
  - Area: (n-1) full adder
  - Delay: $\log_2^n$
  - Flexibility: not so easy to modify
- Software should be able to do the conversion automatically

Thanks!