Digital Logic Circuits

Introduction

Reference

- Text book: V. P. Nelson, H. T. Nagle, B. D. Carroll and J. D. Irwin, *Digital Logic Circuit Analysis and Design,* Prentice Hall, 1995, ISBN 0-13-463894-8.

Course Objective

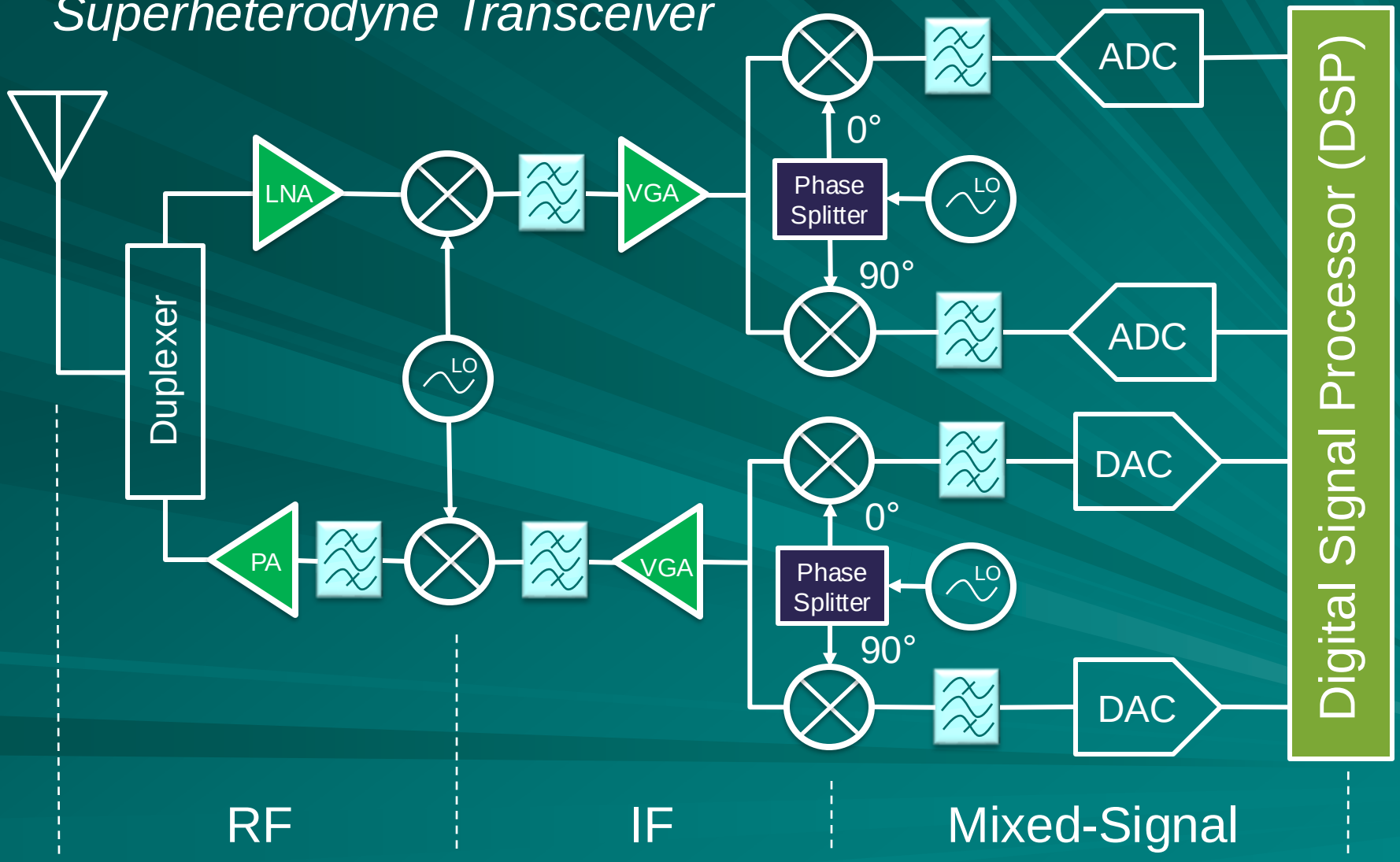
- This course introduces the student to the design of digital logic circuits, both combinational and sequential, and the design of digital systems in a hierarchical, top-down manner. The student is also introduced to the use of computer-aided design tools used to develop digital circuits.

Electronic Systems

- Example 1: Cell phone transceiver.
- Example 2: Digital computer.

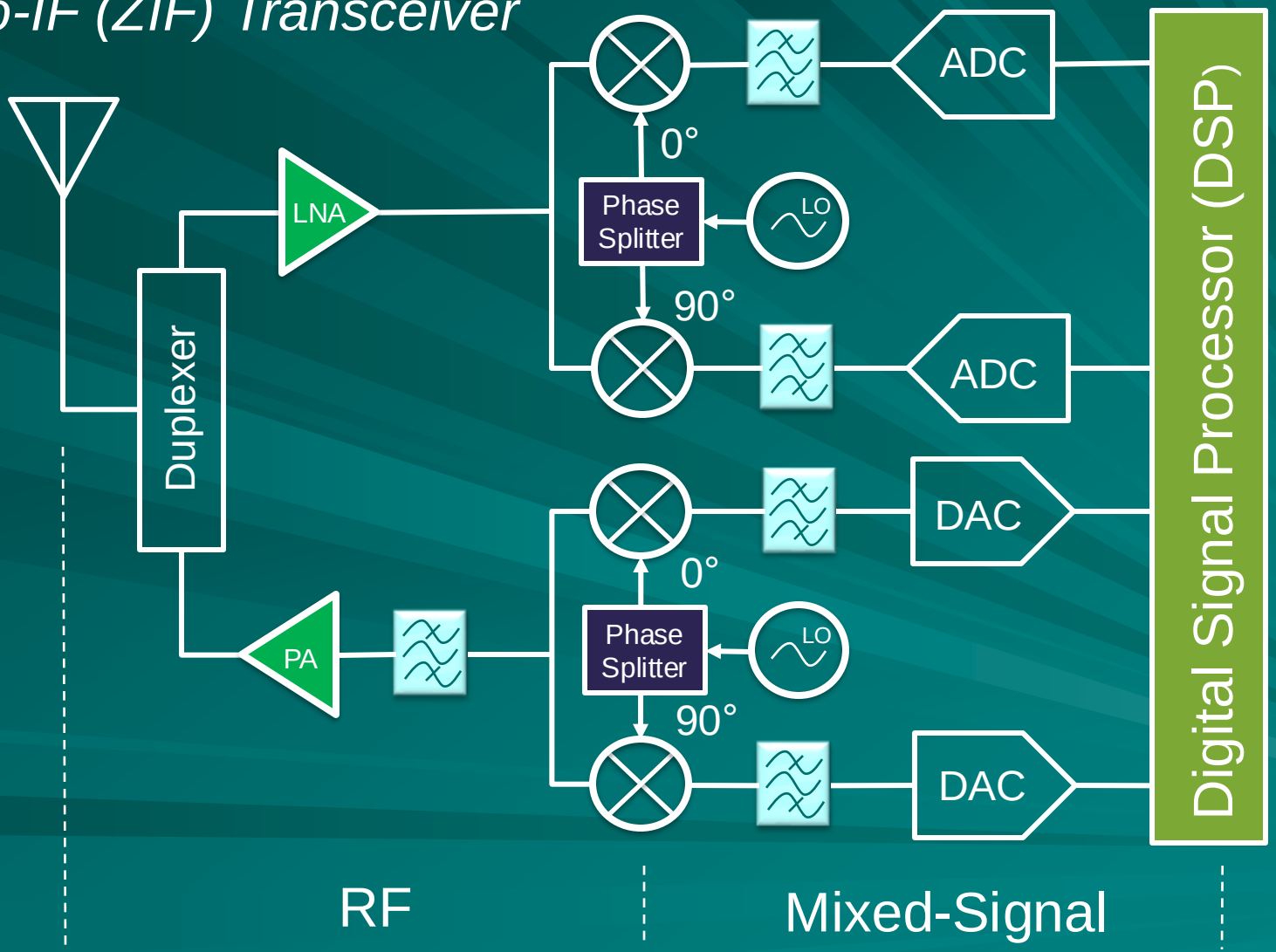
An RF Communications System

Superheterodyne Transceiver

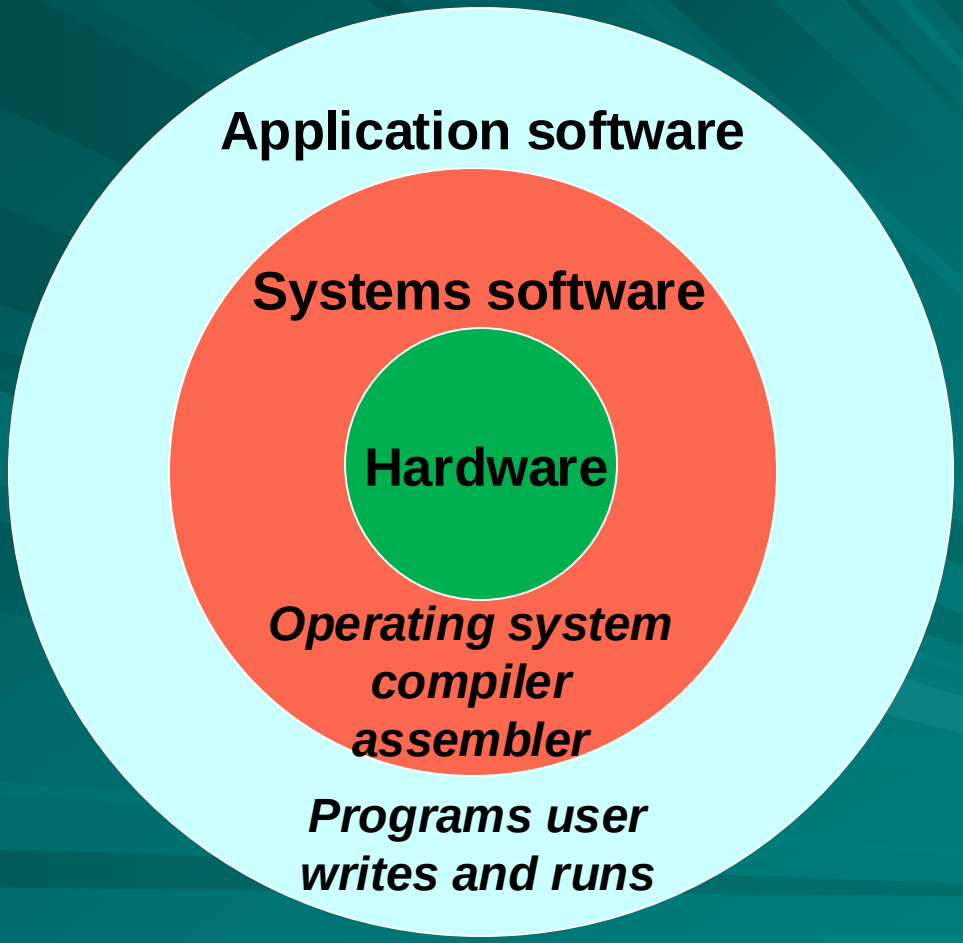


An Alternative RF Communications System

Zero-IF (ZIF) Transceiver



The Concept of a Computer



User

Software

Compiler

Assembler

Application software, a program in C:

```
swap (int  v[ ], int  k)
{int  temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

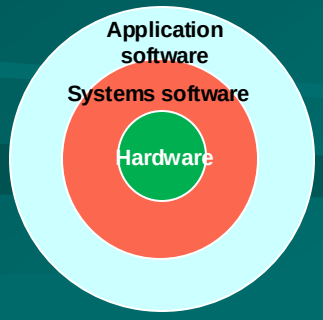
MIPS compiler output, assembly language program:

```
swap;
        muli    $2,     $5, 4
        add     $2,     $4, $2
        lw      $15,    0 ($2)
        lw      $16,    4 ($2)
        sw      $16,    0 ($2)
        sw      $15,    4 ($2)
        jr      $31
```

MIPS binary machine code:

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

32-bit words stored in memory



Machine instructions

Binary Machine Code

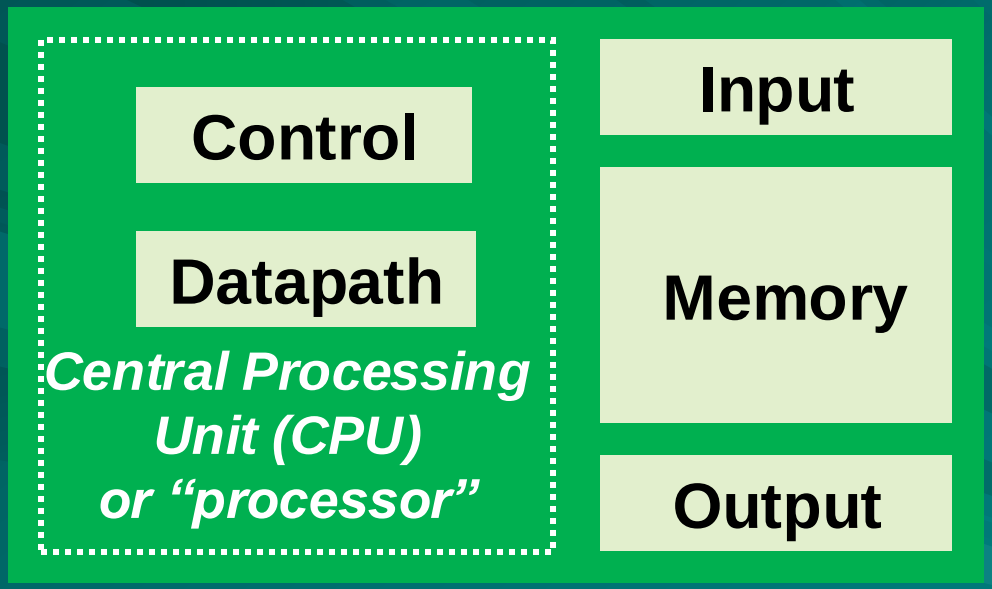
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000

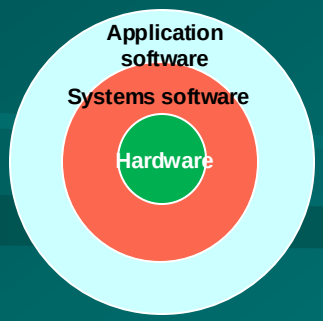


Instruction code (opcode)

Encoded data

The Hardware of a Computer



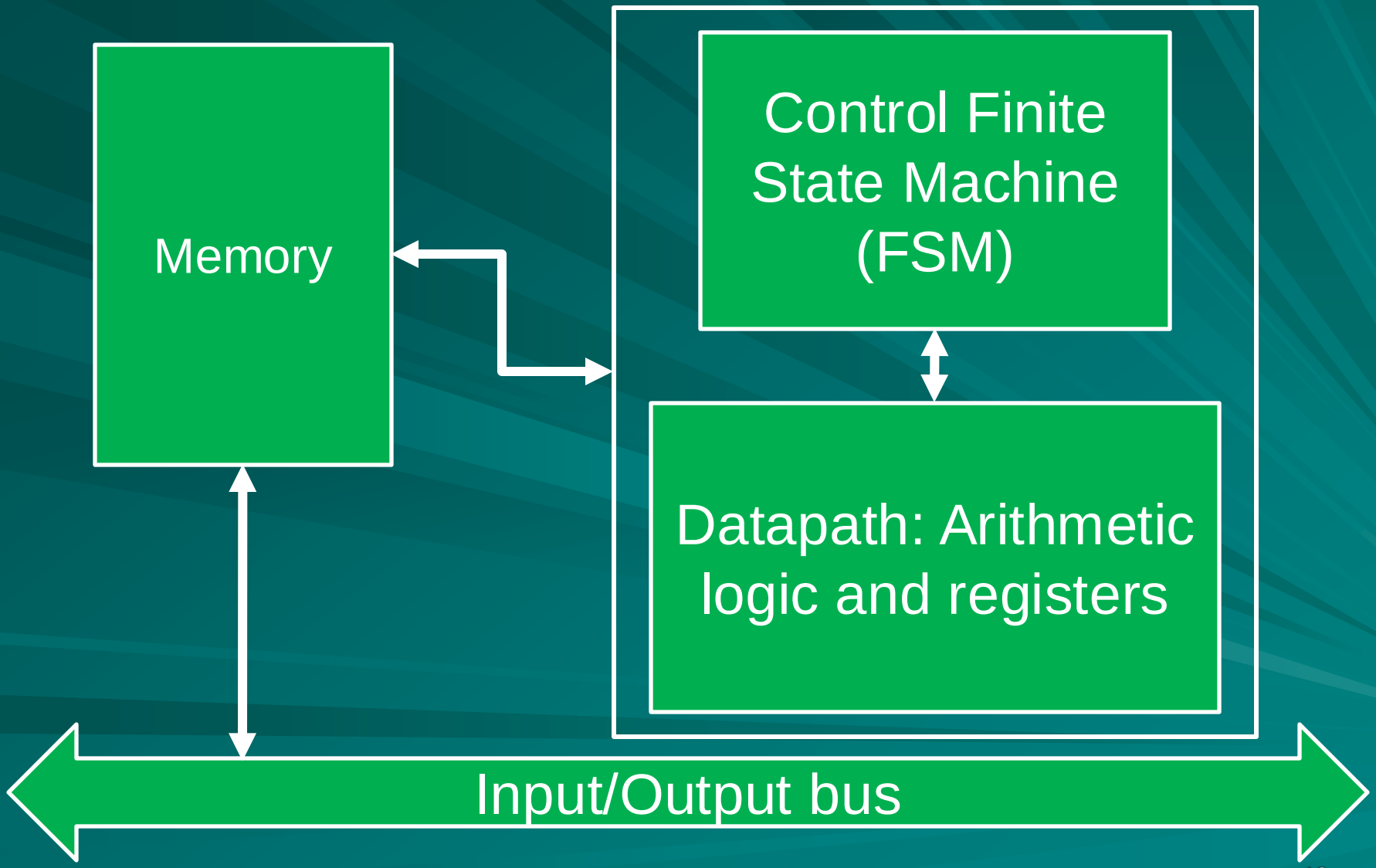


FIVE PIECES

Hardware Processes Machine Code

- User program is translated into binary machine code by compiler and assembler and is stored in memory.
- Control unit reads program from memory, one word at a time (fetch operation).
- Control unit deciphers the instruction bits of program word and configures datapath logic, which processes data and saves results in memory (decode and execute operations).

Digital Hardware of Computer



To Find Out More

- J. Rogers, C. Plett and F. Dai, *Integrated Circuit Design for High-Speed Frequency Synthesis*, Boston: Artech House, 2006.
- D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, the Hardware/Software Interface*, *Fourth Edition*, San Francisco, California: Morgan Kaufman Publishers (Elsevier), 2009.

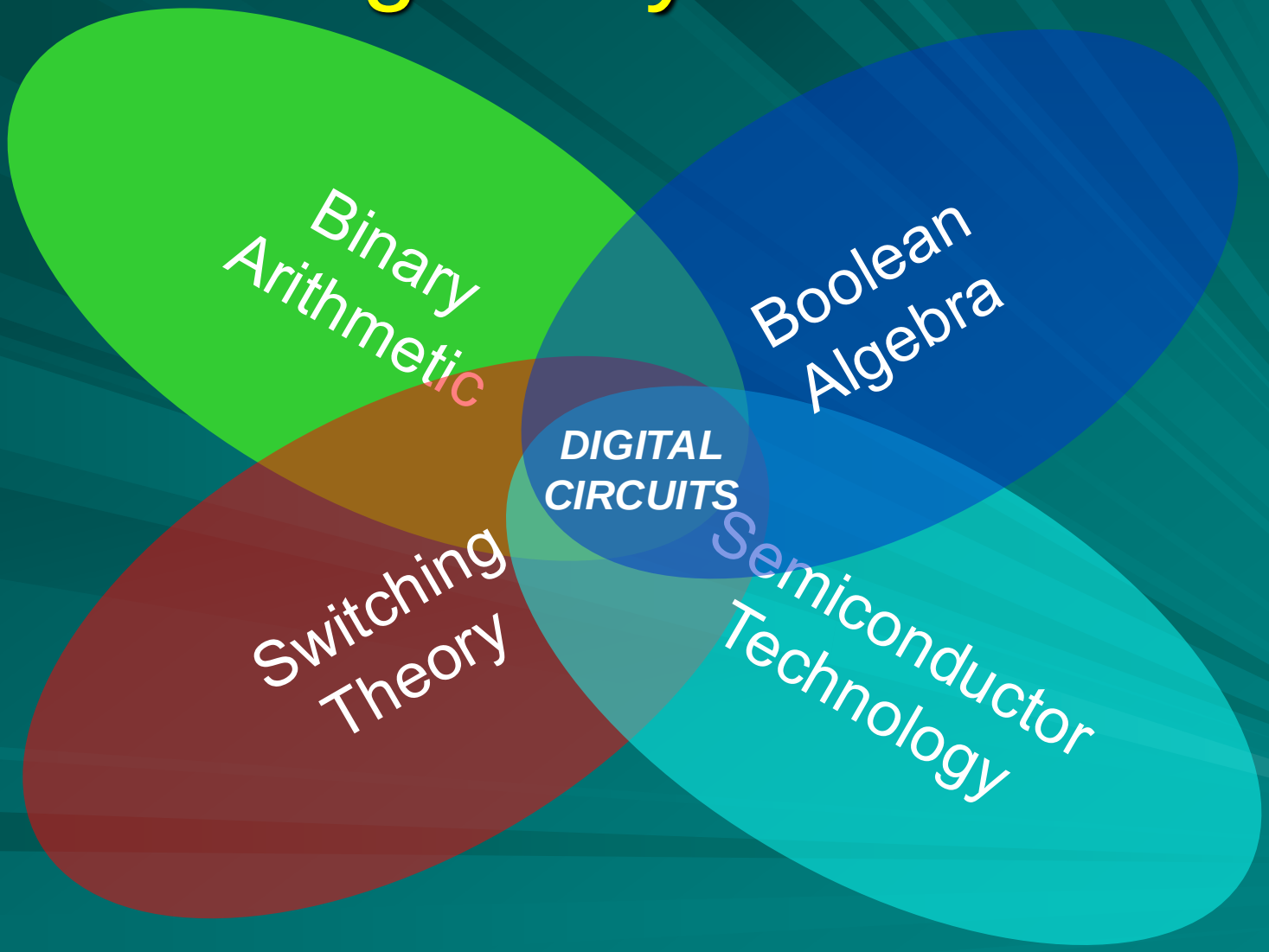
Exercises

- Identify radio frequency (RF), mixed (analog and digital) signal, and digital components in a communications system.
- Which parts in computing and communications systems contain digital arithmetic logic circuits?
- Where is the binary machine code stored in a digital computer?
- What is the difference between sequential and combinational circuits? Which category does the control of a computer belong to?

Answers

- Components of a communications system:
  - RF: Antenna, duplexer, filter, mixer, local oscillator.
  - Mixed-signal: ADC, DAC.
  - Digital: DSP.
- Arithmetic logic is contained in DSP and datapath.
- Machine code is stored in a memory.
- Sequential and combinational circuits:
  - A sequential circuit contains memory. Its output is determined by the input and the content of the memory.
  - A combinational circuit contains no memory. Its output depends entirely upon the input.
  - Control of a computer is a sequential circuit.

Digital Systems

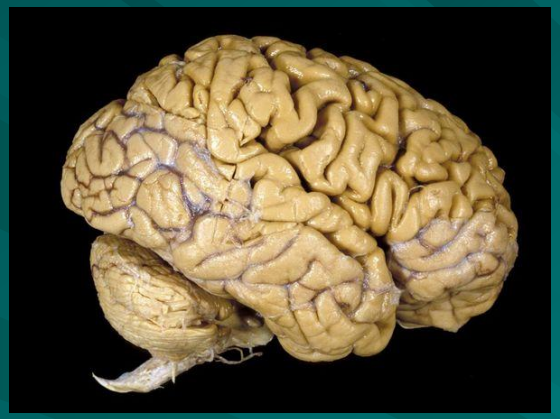


Next

Binary Arithmetic

Why Binary Arithmetic?

3 + 5



8

0011 + 0101



1000

Digital Logic Circuits

Binary Arithmetic (Chapter 1)

Why Binary Arithmetic?

- Hardware can only deal with binary digits, 0 and 1.
- Must represent all numbers, integers or floating point, positive or negative, by binary digits, called *bits*.
- Can devise electronic circuits to perform arithmetic operations: add, subtract, multiply and divide, on binary numbers.

Positive Integers

- Decimal system: made of 10 digits, {0,1,2, . . . , 9}

$$41 = 4\times10^1 + 1\times10^0$$

$$255 = 2\times10^2 + 5\times10^1 + 5\times10^0$$

- Binary system: made of two digits, {0,1}

$$00101001 = 0\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$$

$$= 32 + 8 + 1 = 41$$

11111111 = 255, largest number with 8 binary digits, 2^8-1

Base or Radix

- For decimal system, 10 is called the base or radix.
- Decimal 41 is also written as 41_{10} or 41_{ten}
- Base (radix) for binary system is 2.
- Thus, $41_{ten} = 101001_2$ or 101001_{two}
- Also, $111_{ten} = 1101111_{two}$

  and $111_{two} = 7_{ten}$
- What about negative numbers?

Signed Magnitude – What Not to Do

- Use fixed length binary representation
- Use left-most bit (called *most significant bit* or MSB) for sign:

  0 for positive

  1 for negative

- Example: $+18_{ten} = 00010010_{two}$

  $-18_{ten} = 10010010_{two}$

Difficulties with Signed Magnitude

- Sign and magnitude bits should be differently treated in arithmetic operations.
- Addition and subtraction require different logic circuits.
- Overflow is difficult to detect.
- "Zero" has two representations:

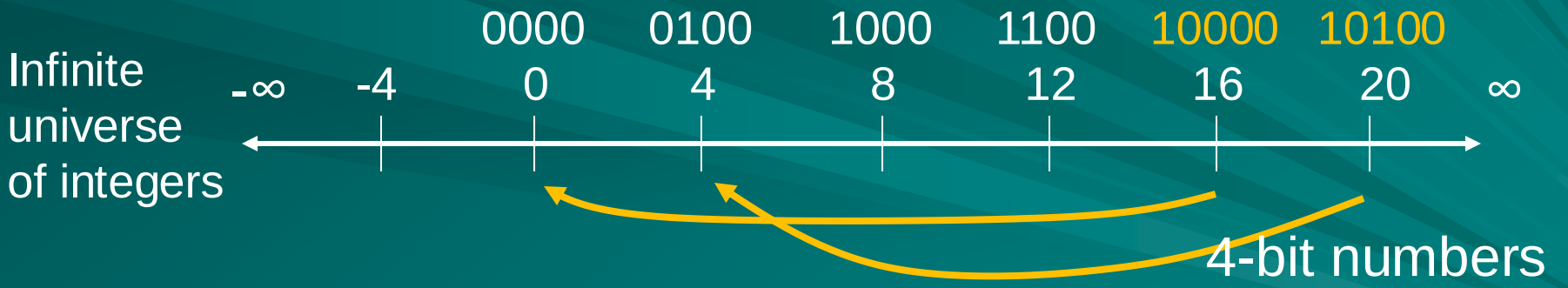
  $+ 0_{ten} = 00000000_{two}$

  $- 0_{ten} = 10000000_{two}$

- ***Signed-integers are not used in modern computers.***

Problems with Finite Math

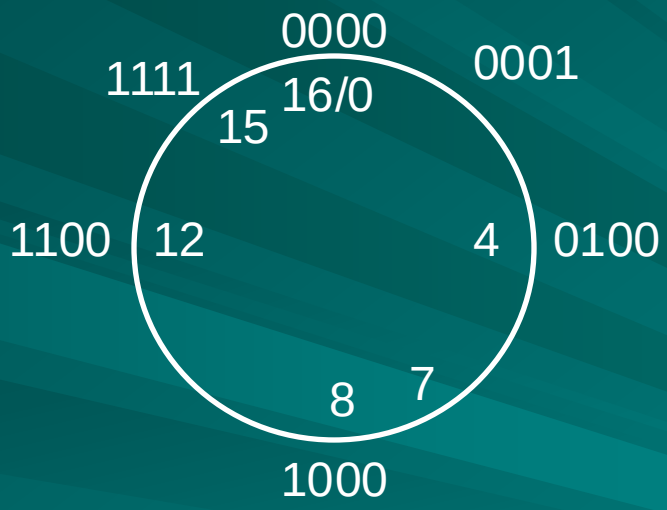
- Finite size of representation:
  - Digital circuit cannot be arbitrarily large.
  - Overflow detection – easy to determine when the number becomes too large.

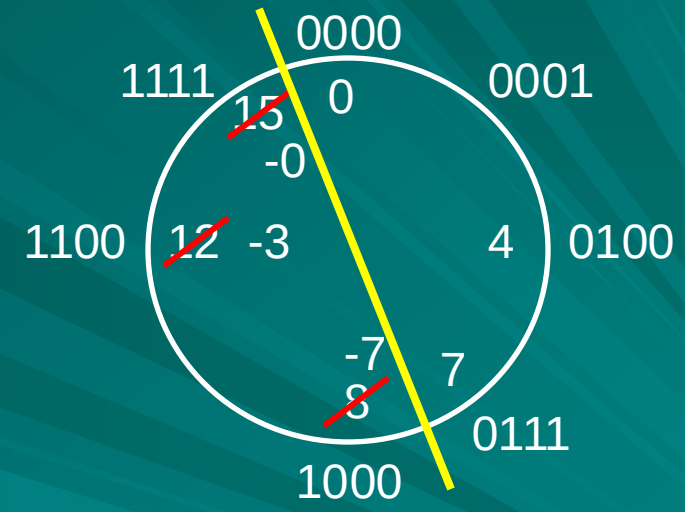


- Represent negative numbers:
  - Unique representation of 0.

4-bit Universe

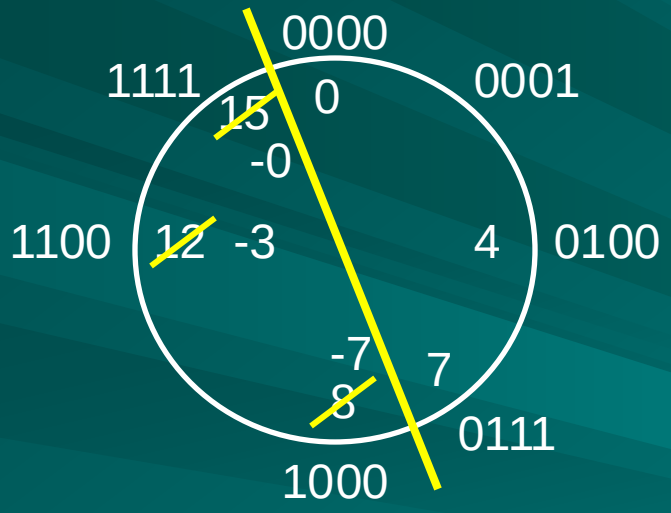
Modulo-16 (4-bit) universe





Only 16 integers: 0 through 15, or – 7 through 7

One Way to Divide Universe
1's Complement Numbers

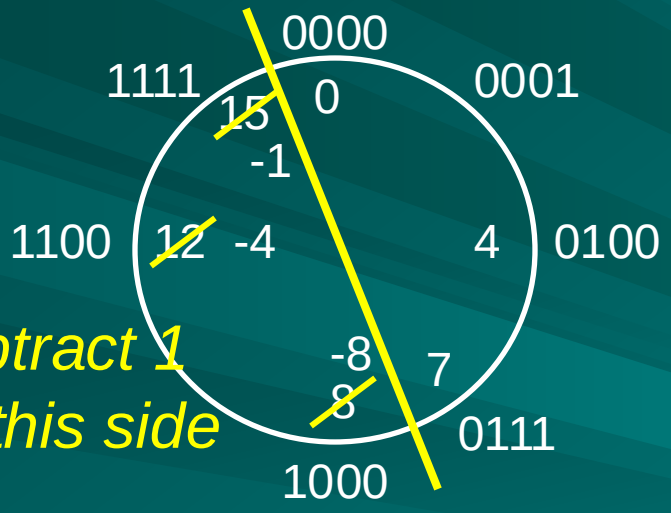


Decimal magnitude	Binary number	
	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

Negation rule: invert bits.

Problem: 0 ≠ – 0

Another Way to Divide Universe 2's Complement Numbers



Negation rule: invert bits and add 1

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

Integers With Sign – Two Ways

- Use fixed-length representation, but no explicit sign bit:
  - 1's complement: To form a negative number, complement each bit in the given number.
  - 2's complement: To form a negative number, start with the given number, subtract one, and then complement each bit, *or*

    first complement each bit, and then add 1.
- 2's complement is the preferred representation.

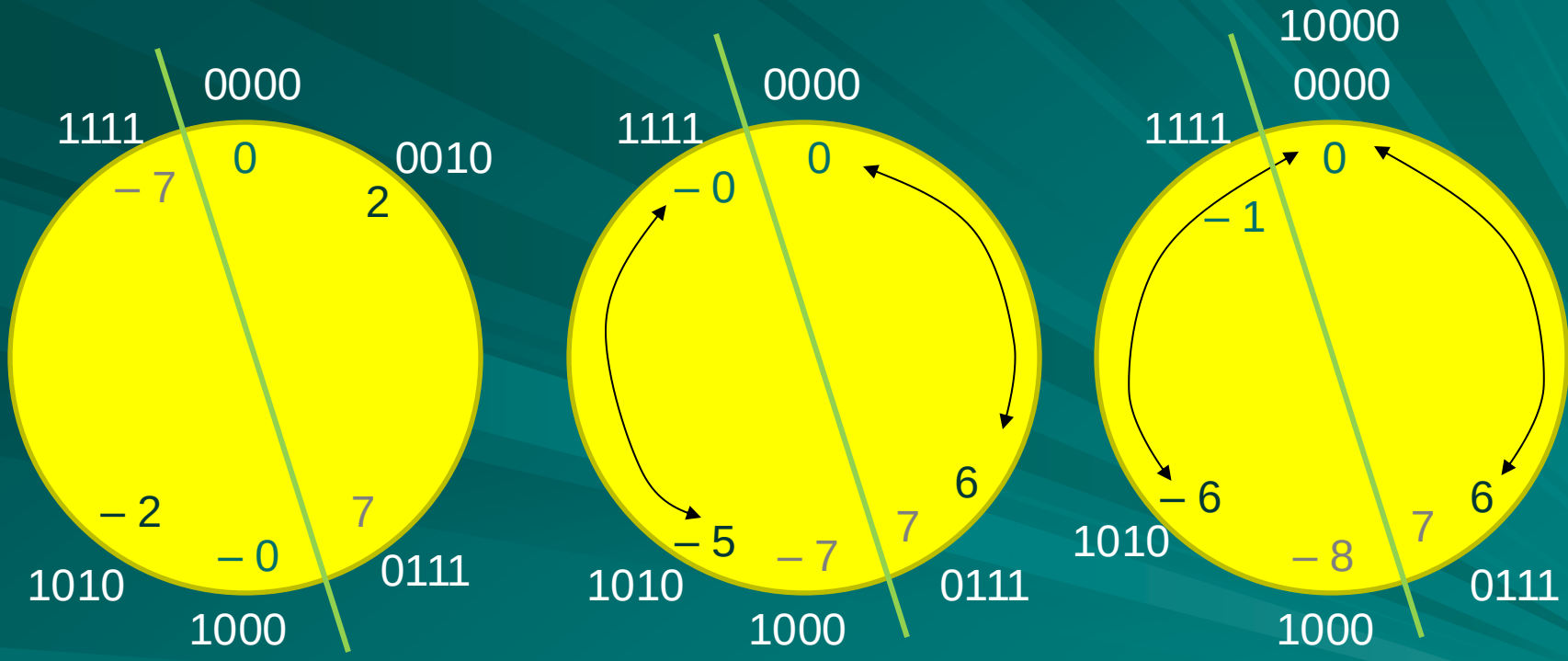
2’s-Complement Integers

- Why not 1’s-complement? *Don’t like two zeros.*
- Negation rule:
  - Subtract 1 and then invert bits, or
  - Invert bits and add 1
- Some properties:
  - Only one representation for 0
  - Exactly as many positive numbers as negative numbers
  - Slight asymmetry – there is one negative number with no positive counterpart

General Method for Binary Integers with Sign

- Select number (n) of bits in representation.
- Partition 2^n integers into two sets:
  - 00…0 through 01…1 are $2^n/2$ positive integers.
  - 10…0 through 11…1 are $2^n/2$ negative integers.
- Negation rule transforms negative to positive, and vice-versa:
  - Signed magnitude: invert MSB (most significant bit)
  - 1's complement: Subtract from $2^n - 1$ or 1…1 (same as "inverting all bits")
  - 2's complement: Subtract from 2^n or 10…0 (same as 1's complement + 1)

Three Systems (n = 4)



1010 = – 2

Signed magnitude

1010 = – 5

1’s complement integers

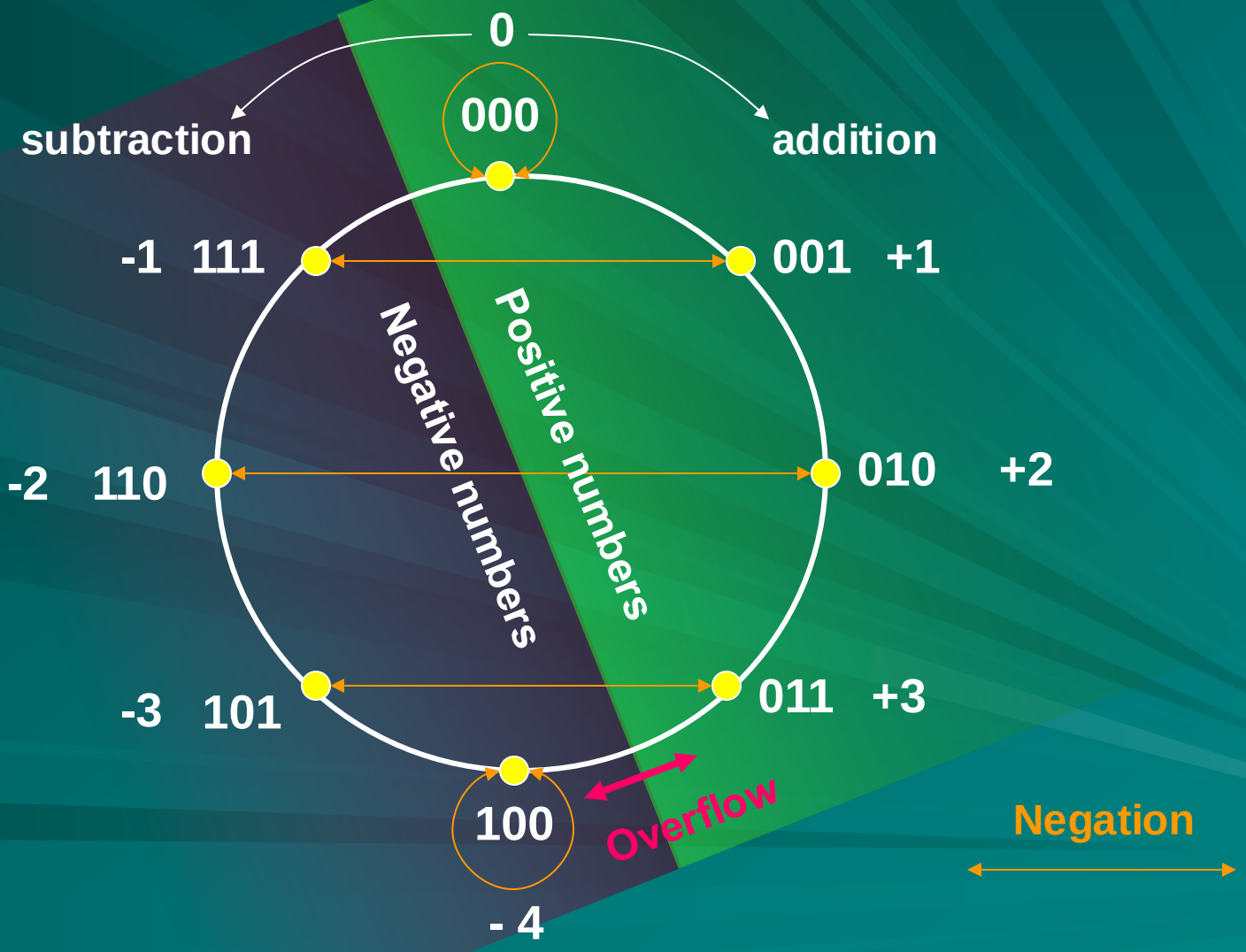
1010 = – 6

2’s complement integers

Three Representations

Sign-magnitude	1's complement	2's complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = - 0	100 = - 3	100 = - 4
101 = - 1	101 = - 2	101 = - 3
110 = - 2	110 = - 1	110 = - 2
111 = - 3	111 = - 0	111 = - 1
		(Preferred)

2’s Complement Numbers (n = 3)



2's Complement n-bit Numbers

- Range: -2^{n-1} through $2^{n-1} - 1$
- Unique zero: 00000000 0
- Negation rule: see slide 11 or 13.
- Expansion of bit length: stretch the left-most bit all the way, e.g., 11111101 is still 101 or – 3. Also, 00000011 is same as 011 or 3.
- Most significant bit (MSB) indicates sign.
- Overflow rule: If two numbers with the same sign bit (both positive or both negative) are added, the overflow occurs if and only if the result has the opposite sign.
- Subtraction rule: for A – B, add – B and A.

Summary

- For a given number (n) of digits we have a finite set of integers. For example, there are $10^3 = 1{,}000$ decimal integers and $2^3 = 8$ binary integers in 3-digit representations.
- We divide the finite set of integers $[0, r^n - 1]$, where radix r = 10 or 2, into two equal parts representing positive and negative numbers.
- Positive and negative numbers of equal magnitudes are complements of each other: x + complement (x) = 0.

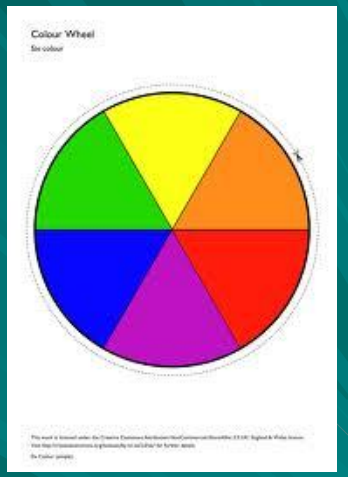
Summary: Defining Complement

- Decimal integers:
  - 10's complement: – x = Complement (x) = 10^n – x
  - 9's complement: – x = Complement (x) = 10^n – 1 – x
  - For 9's complement, subtract each digit from 9
  - For 10's complement, add 1 to 9's complement
- Binary integers:
  - 2's complement: – x = Complement (x) = 2^n – x
  - 1's complement: – x = Complement (x) = 2^n – 1 – x
  - For 1's complement, subtract each digit from 1
  - For 2's complement, add 1 to 1's complement

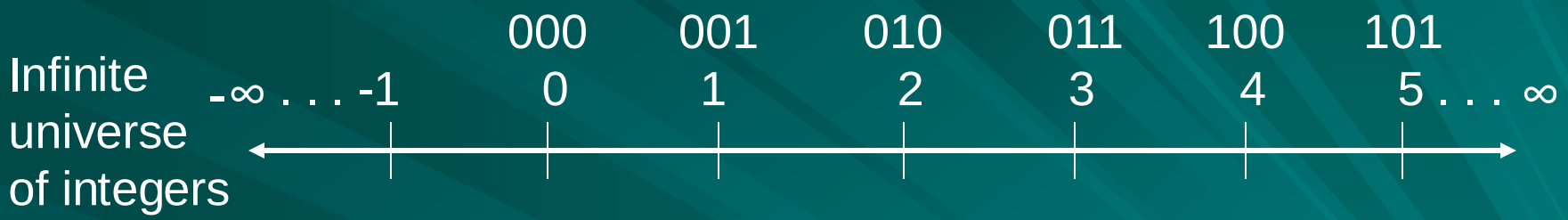
Understanding Complement

- Complement means “something that completes”:

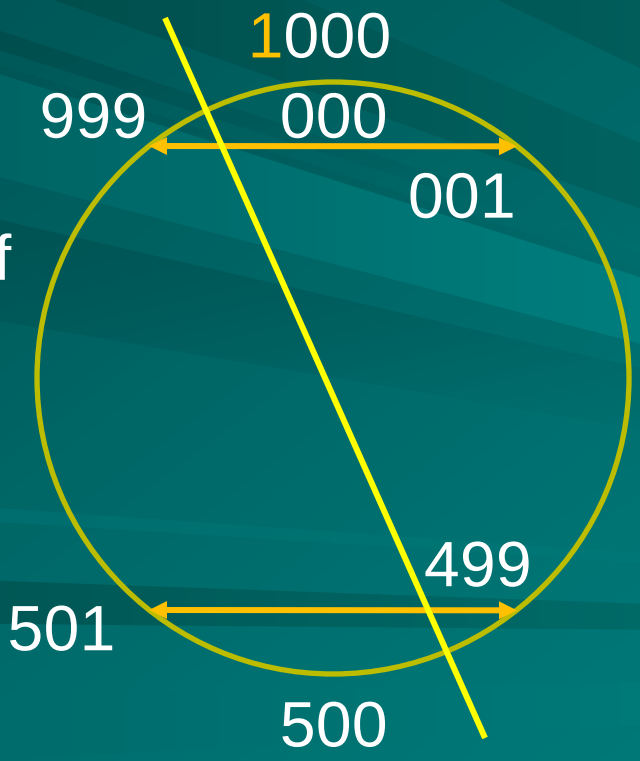
  e.g., X + complement (X) = “Whole”.
- Complement also means “opposite”, e.g., complementary colors are placed opposite in the primary color chart.
- Complementary numbers are like electric charges. Positive and negative charges of equal magnitudes annihilate each other.

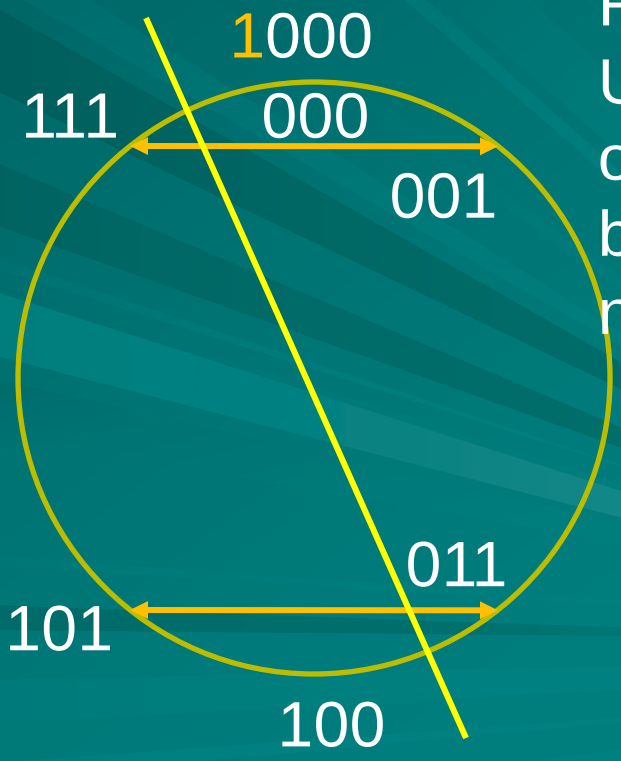


2's-Complement Numbers



Finite Universe of 3-digit Decimal numbers





Finite Universe of 3-bit binary numbers

Examples of Complements

- Decimal integers (r = 10, n = 3):
  - 10's complement: – 50 = Compl (50) = 10^3 – 50 = 950; 50 + 950 = 1,000 = 0 (in 3 digit representation)
  - 9's complement: – 50 = Compl (50) = 10^n – 1 – 50 = 949; 50 + 949 = 999 = – 0 (in 9's complement rep.)
- Binary integers (r = 2, n = 4):
  - 2's complement: – 5 = Complement (5) = 2^4 – 5 = 11_{10} or 1011; 5 + 11 = 16 = 0 (in 4-bit representation)
  - 1's complement: – 5 = Complement (5) = 2^4 – 1 – 5 = 10_{10} or 1010; 5 + 10 = 15 = – 0 (in 1's complement representation)

2's-Complement to Decimal Conversion

$$b_{n-1}\, b_{n-2} \ldots b_1\, b_0 = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i\, b_i$$

8-bit conversion box

-128	64	32	16	8	4	2	1

Example

-128	64	32	16	8	4	2	1
1	1	1	1	1	1	0	1

– 128 + 64 + 32 + 16 + 8 + 4 + 1 = – 128 + 125 = – 3

For More on 2's-Complement





Donald E. Knuth (1938 -)





Abu Abd-Allah ibn Musa al'Khwarizmi (~780 – 850)

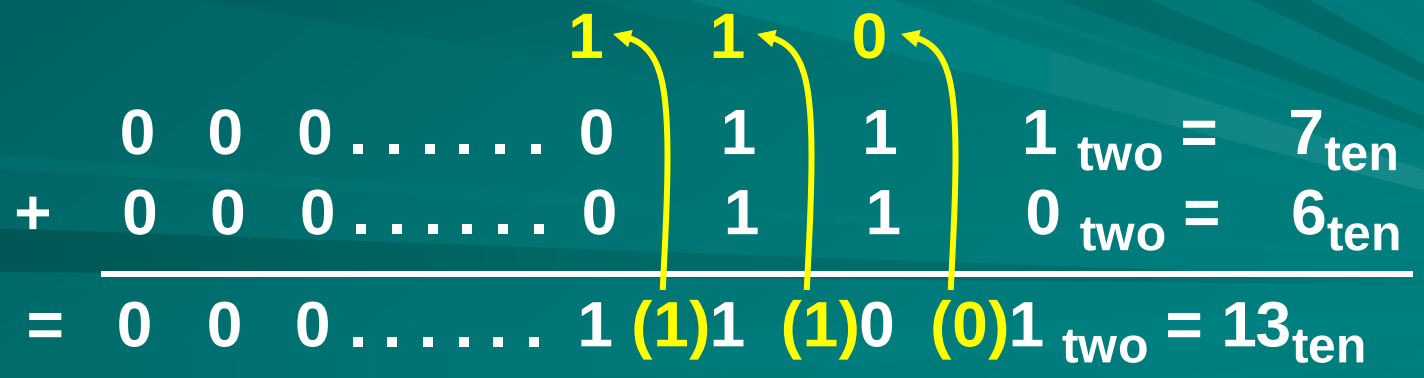
- Chapter 4 in D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms, Volume II, Second Edition*, Addison-Wesley, 1981.
- A. al'Khwarizmi, *Hisab al-jabr w'al-muqabala*, 830.
- **Read: A two part interview with D. E. Knuth, *Communications of the ACM* (CACM), vol. 51, no. 7, pp. 35-39 (July), and no. 8, pp. 31-35 (August), 2008**.

Addition

- Adding bits:
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = (1) 0

carry

- Adding integers:



Subtraction

- Direct subtraction

$$\begin{array}{rcccccccl} & 0 & 0 & 0 \ldots\ldots 0 & 1 & 1 & 1_{two} = & 7_{ten} \\ - & 0 & 0 & 0 \ldots\ldots 0 & 1 & 1 & 0_{two} = & 6_{ten} \\ \hline = & 0 & 0 & 0 \ldots\ldots 0 & 0 & 0 & 1_{two} = & 1_{ten} \end{array}$$

- Two's complement subtraction by adding

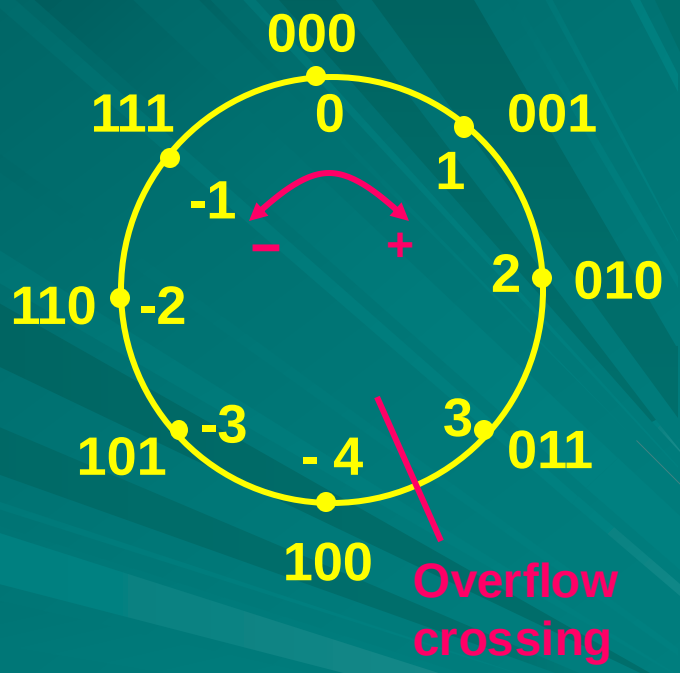
$$\begin{array}{rcccccccl} & 1 & 1 & 1 \ldots\ldots 1 & 1 & 0 & & \\ & 0 & 0 & 0 \ldots\ldots 0 & 1 & 1 & 1_{two} = & 7_{ten} \\ + & 1 & 1 & 1 \ldots\ldots 1 & 0 & 1 & 0_{two} = - & 6_{ten} \\ \hline = & 0 & 0 & 0 \ldots\ldots 0 & (1) & 0 \; (1) \; 0 \; (0) & 1_{two} = & 1_{ten} \end{array}$$

Overflow: An Error

- Examples: Addition of 3-bit integers (range - 4 to +3)
  - -2-3 = -5

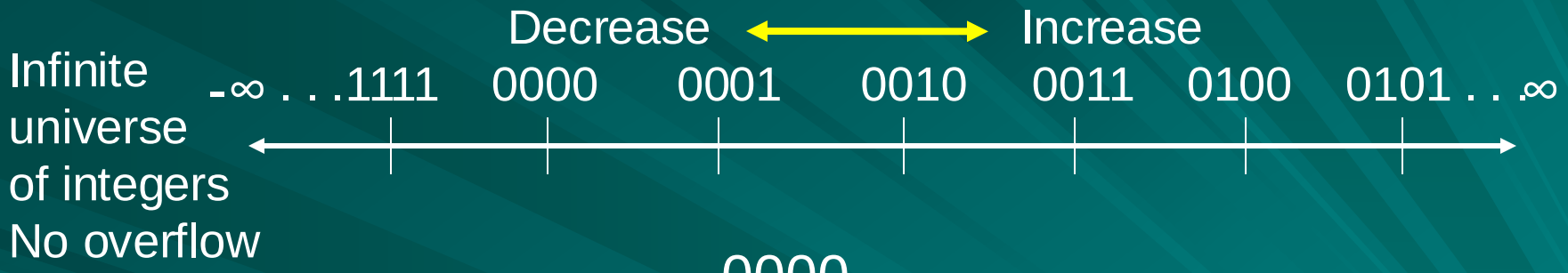
    110 = -2
    + 101 = -3
    = 1011 = 3 (error)
  - 3+2 = 5

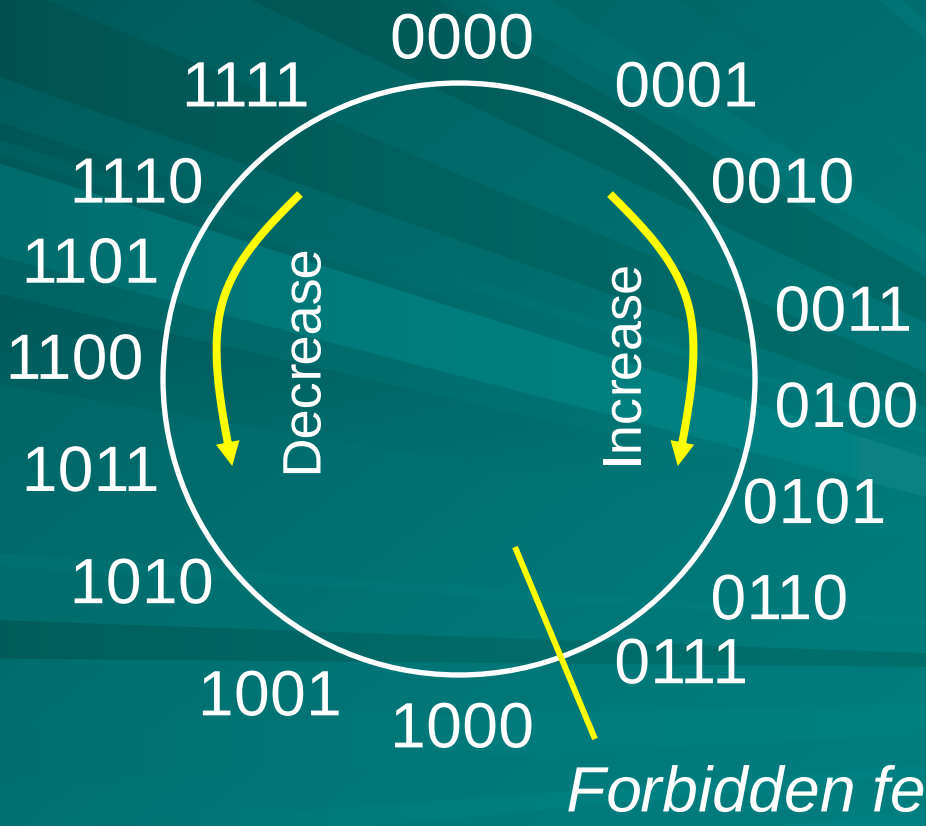
    011 = 3
    010 = 2
    = 101 = -3 (error)



- **Overflow rule: If two numbers with the same sign bit (both positive or both negative) are added, the overflow occurs if and only if the result has the opposite sign.**

Overflow and Finite Universe

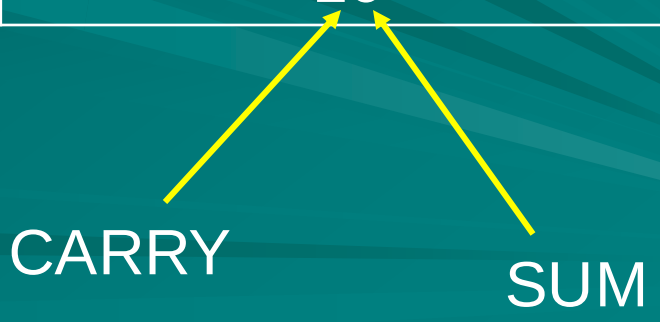




Finite Universe of 4-bit binary integers

Adding Two Bits

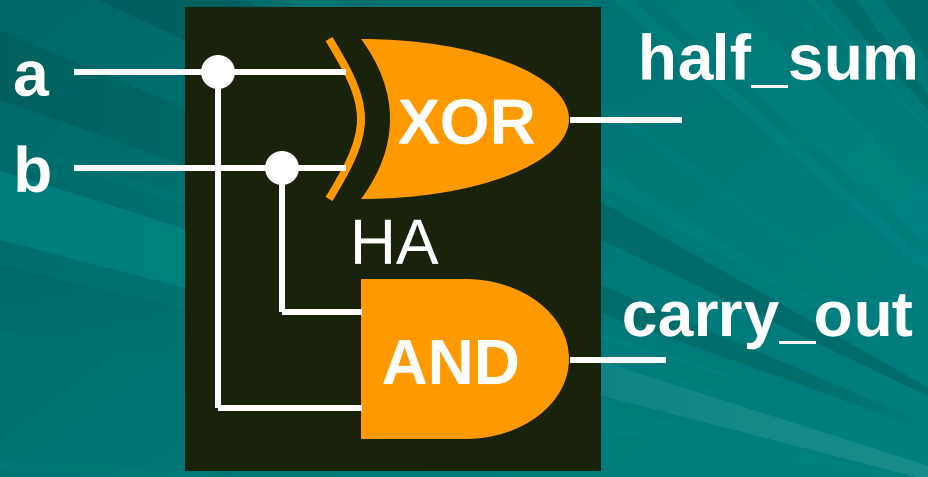
a	b	s = a+b	
		Decimal	Binary
0	0	0	00
0	1	1	01
1	0	1	01
1	1	2	10



Half-Adder Adds two Bits

- Adding two bits:

a	b	half_sum h_s(a, b)	carry_out c_o(a, b)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



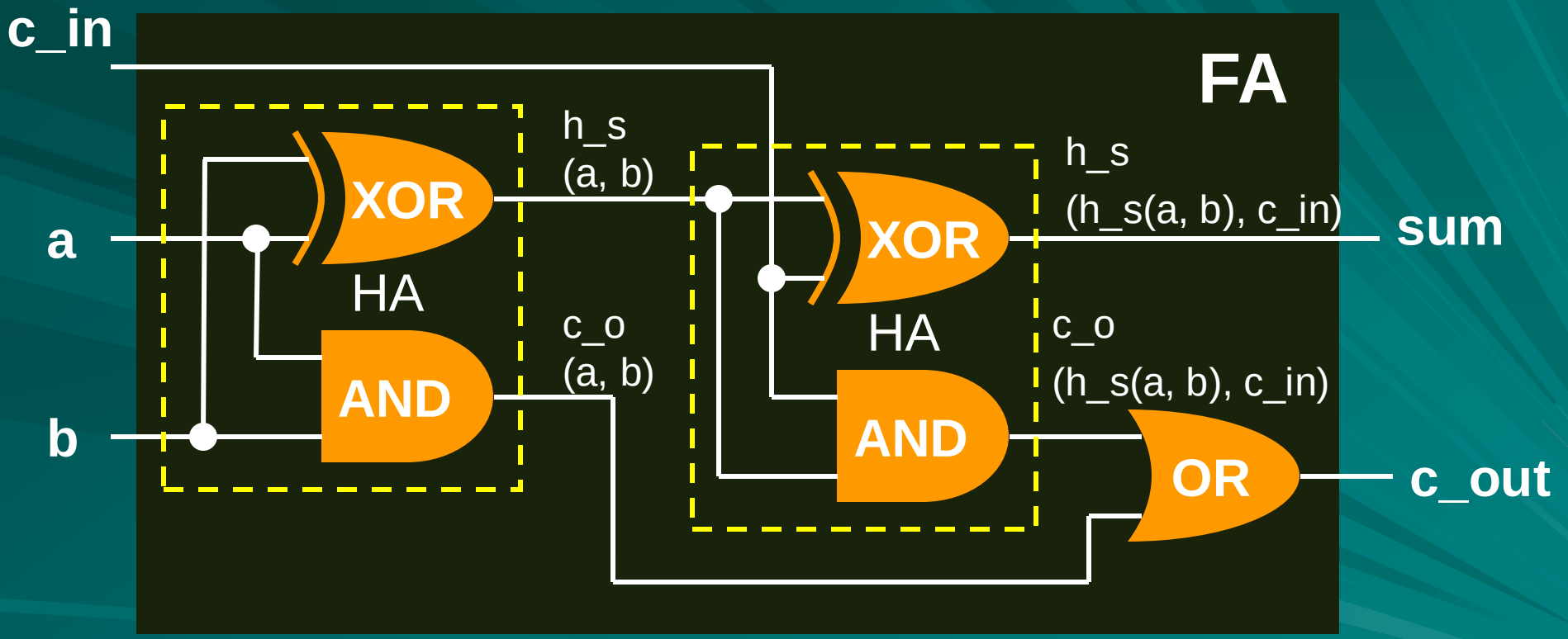
Adding Three Bits

a	b	c	s=a+b+c	
			Decimal	Binary
0	0	0	0	00
0	0	1	1	01
0	1	0	1	01
0	1	1	2	10
1	0	0	1	01
1	0	1	2	10
1	1	0	2	10
1	1	1	3	11

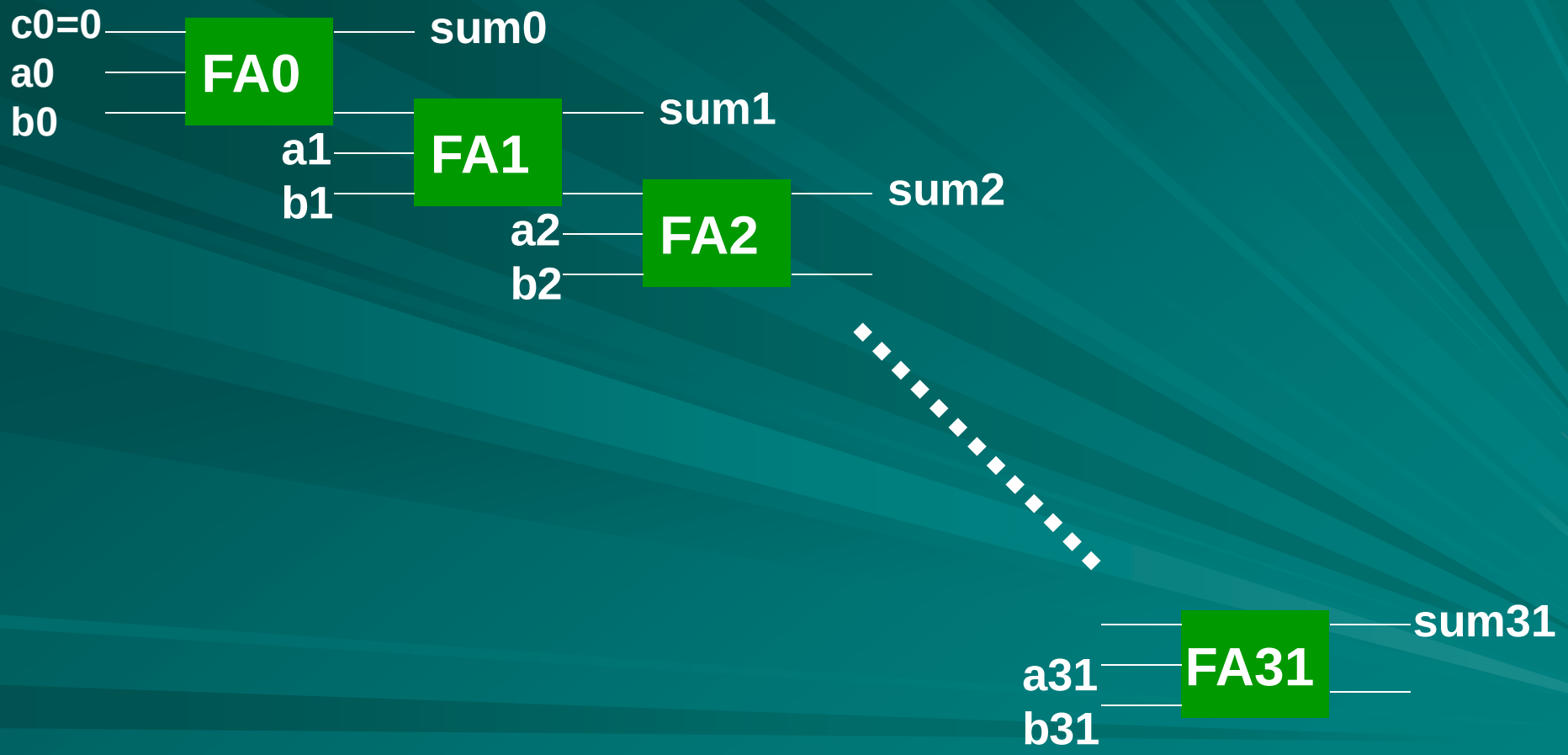
CARRY

SUM

Full-Adder Adds Three Bits



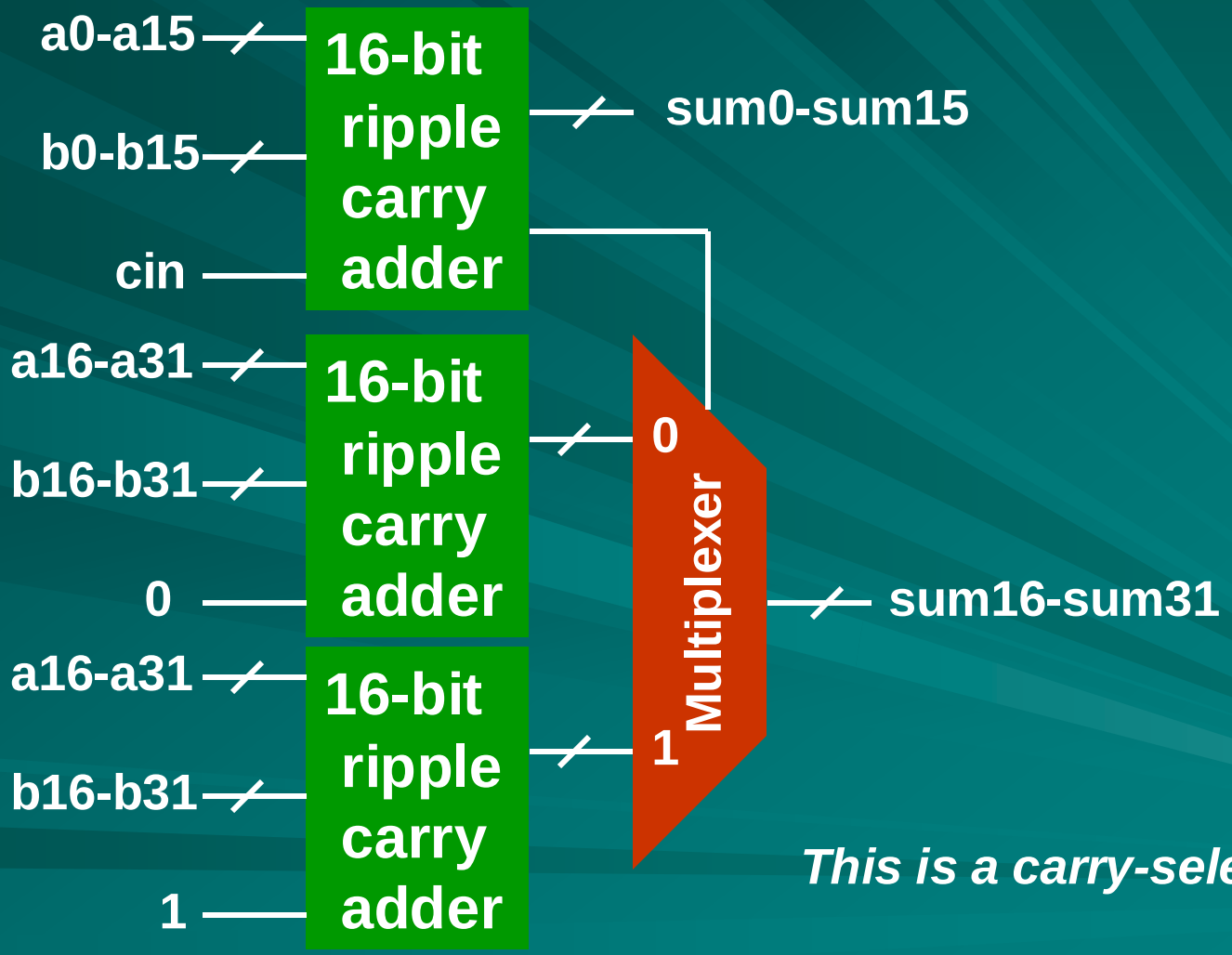
32-bit Ripple-Carry Adder



How Fast is Ripple-Carry Adder?

- Longest delay path (critical path) runs from (a0, b0) to sum31.
- Suppose delay of full-adder is 100ps.
- Critical path delay = 3,200ps
- Clock rate cannot be higher than $1/(3{,}200\times10^{-12})$ Hz = 312MHz.
- Must use more efficient ways to handle carry.

Speeding Up the Adder



This is a carry-select adder

Fast Adders

- In general, any output of a 32-bit adder can be evaluated as a logic expression in terms of all 65 inputs.
- Number of levels of logic can be reduced to $\log_2 N$ for N-bit adder. Ripple-carry has N levels.
- More gates are needed, about $\log_2 N$ times that of ripple-carry design.
- Fastest design is known as carry lookahead adder.

N-bit Adder Design Options

Type of adder	Time complexity (delay)	Space complexity (size)
Ripple-carry	O(N)	O(N)
Carry-lookahead	$O(\log_2 N)$	$O(N \log_2 N)$
Carry-skip	$O(\sqrt{N})$	O(N)
Carry-select	$O(\sqrt{N})$	O(N)

Reference: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, San Francisco, California, 1990, page A-46.

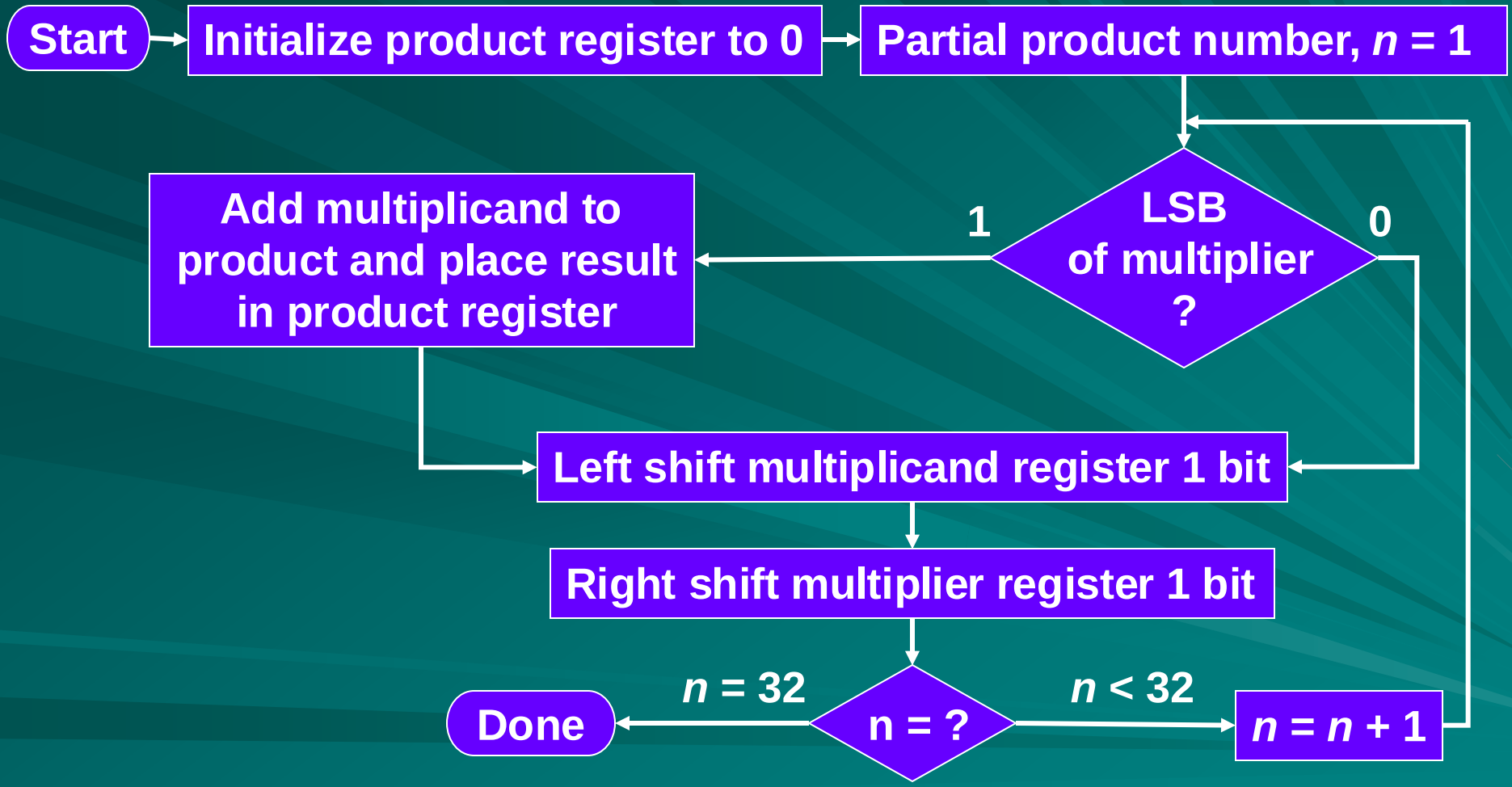
Binary Multiplication (Unsigned)

$1\ 0\ 0\ 0_{two}$	$= 8_{ten}$	**multiplicand**
$1\ 0\ 0\ 1_{two}$	$= 9_{ten}$	**multiplier**

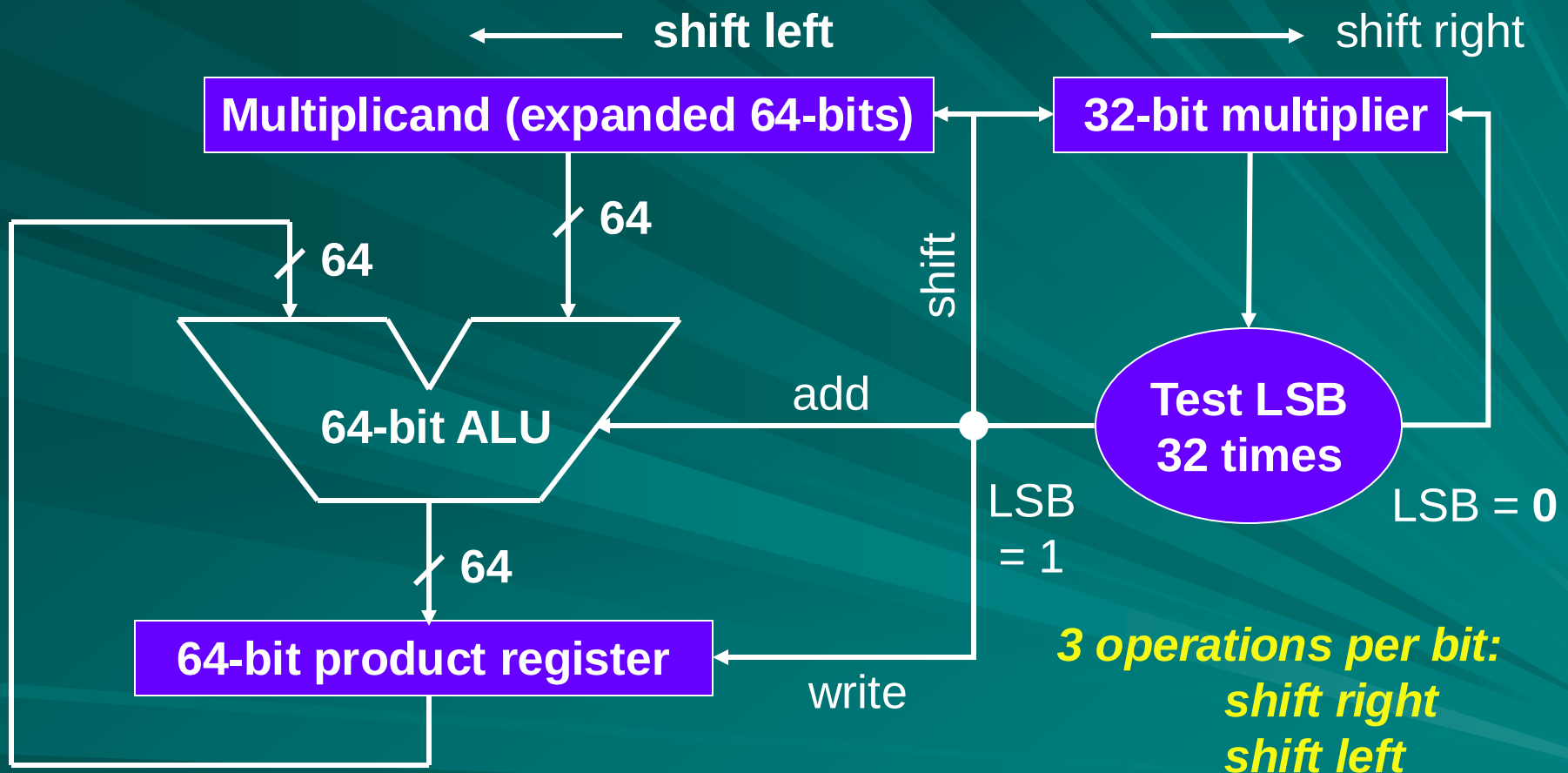
```
    1 0 0 0
  0 0 0 0                    partial products
0 0 0 0
1 0 0 0
-----------------
1 0 0 1 0 0 0   = 72ten
```

Basic algorithm: For n = 1, 32, only If *n*th bit of multiplier is 1, then add multiplicand × 2^{n-1} to product

Multiplication Flowchart

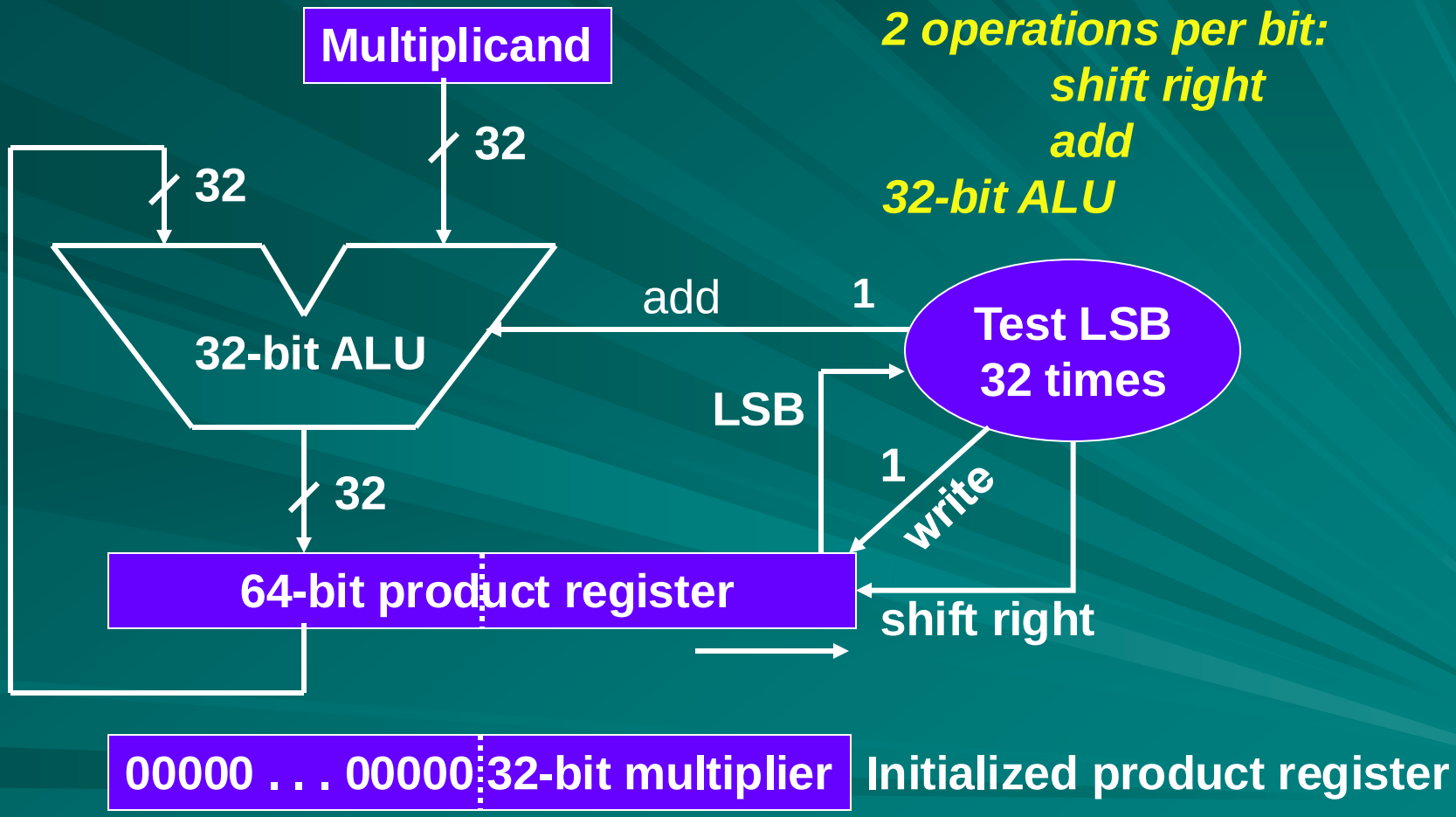


Serial Multiplication



3 operations per bit:
shift right
shift left
add
Need 64-bit ALU

Serial Multiplication (Improved)



2 operations per bit:
shift right
add
32-bit ALU

Example: $0010_{two} \times 0011_{two}$

$0010_{two} \times 0011_{two} = 0110_{two}$, i.e., $2_{ten} \times 3_{ten} = 6_{ten}$

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	LSB =1 => Prod = Prod + Mcand	0010	0010 0011
	Right shift product	0010	0001 0001
2	LSB =1 => Prod = Prod + Mcand	0010	0011 0001
	Right shift product	0010	0001 1000
3	LSB = 0 => no operation	0010	0001 1000
	Right shift product	0010	0000 1100
4	LSB = 0 => no operation	0010	0000 1100
	Right shift product	0010	0000 0110

Multiplying with Signs

- Convert numbers to magnitudes.
- Multiply the two magnitudes through 32 iterations.
- Negate the result if the signs of the multiplicand and multiplier differed.
- Alternatively, the previous algorithm will work with some modifications. See B. Parhami, *Computer Architecture*, New York: Oxford University Press, 2005, pp. 199-200.

Example 1: $1010_{two} \times 0011_{two}$

$1010_{two} \times 0011_{two} = 101110_{two}$, i.e., $-6_{ten} \times 3_{ten} = -18_{ten}$

Iteration	Step	Multiplicand	Product
0	Initial values	11010	00000 0011
1	LSB = 1 => Prod = Prod + Mcand	11010	11010 0011
	Right shift product	11010	11101 0001
2	LSB = 1 => Prod = Prod + Mcand	11010	10111 0001
	Right shift product	11010	11011 1000
3	LSB = 0 => no operation	11010	11011 1000
	Right shift product	11010	11101 1100
4	LSB = 0 => no operation	11010	11101 1100
	Right shift product	11010	11110 1110

Example 2: $1010_{two} \times 1011_{two}$

$1010_{two} \times 1011_{two} = 011110_{two}$, i.e., $-6_{ten} \times (-5_{ten}) = 30_{ten}$

Iteration	Step	Multiplicand	Product
0	Initial values	11010	00000 1011
1	LSB =1 => Prod = Prod + Mcand	11010	11010 1011
	Right shift product	11010	11101 0101
2	LSB =1 => Prod = Prod + Mcand	11010	10111 0101
	Right shift product	11010	11011 1010
3	LSB = 0 => no operation	11010	11011 1010
	Right shift product	11010	11101 1101
4	LSB =1 => Prod = Prod – Mcand*	00110	00011 1101
	Right shift product	11010	00001 1110

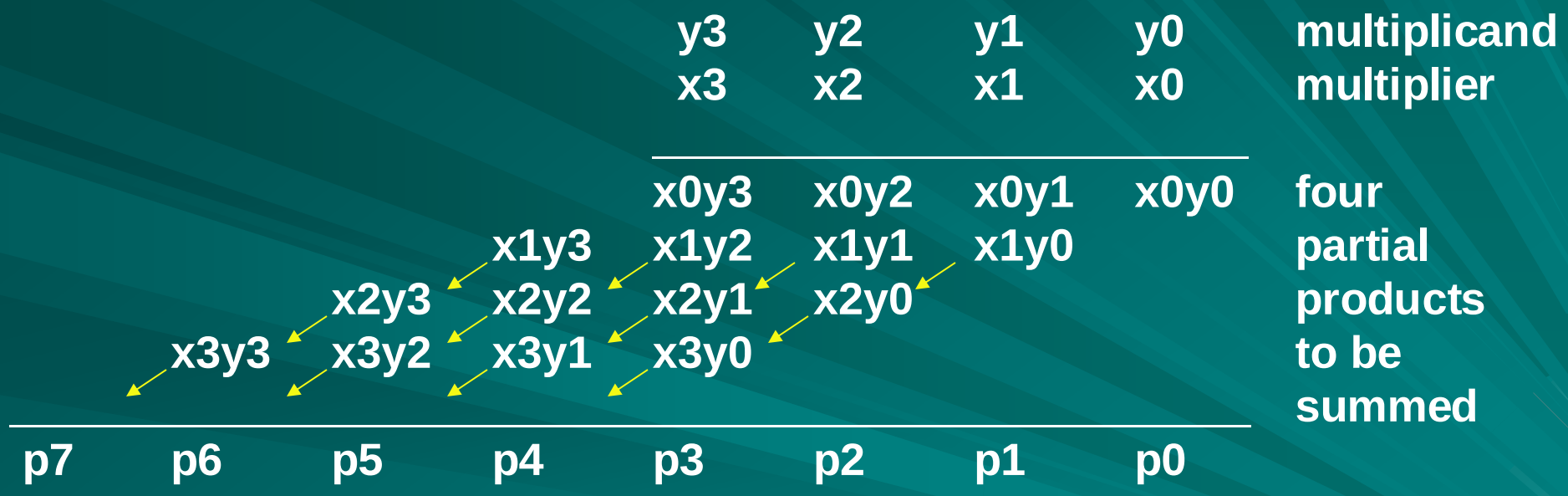
*Last iteration with a negative multiplier in 2's complement.

Adding Partial Products

				y3	y2	y1	y0	multiplicand
				x3	x2	x1	x0	multiplier
				x0y3	x0y2	x0y1	x0y0	four
			carry← x1y3	x1y2	x1y1	x1y0		partial
		carry← x2y3	x2y2	x2y1	x2y0			products
carry←	x3y3	x3y2	x3y1	x3y0				to be
								summed
p7	p6	p5	p4	p3	p2	p1	p0	

Requires three 4-bit additions. Slow.

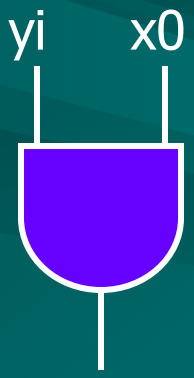
Array Multiplier: Carry Forward



Note: Carry is added to the next partial product (carry-save addition). Adding the carry from the final stage needs an extra (ripple-carry stage. These additions are faster but we need four stages.

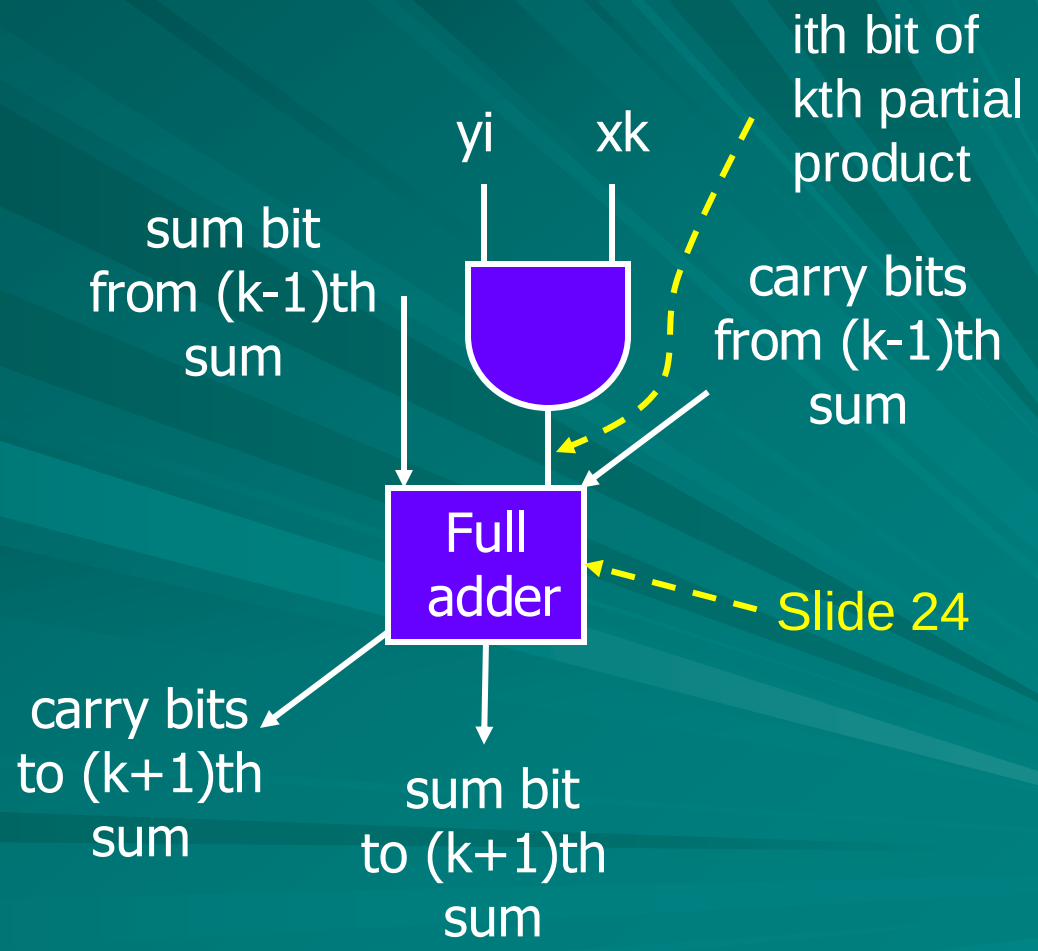
Basic Building Blocks

- Two-input AND
- Full-adder

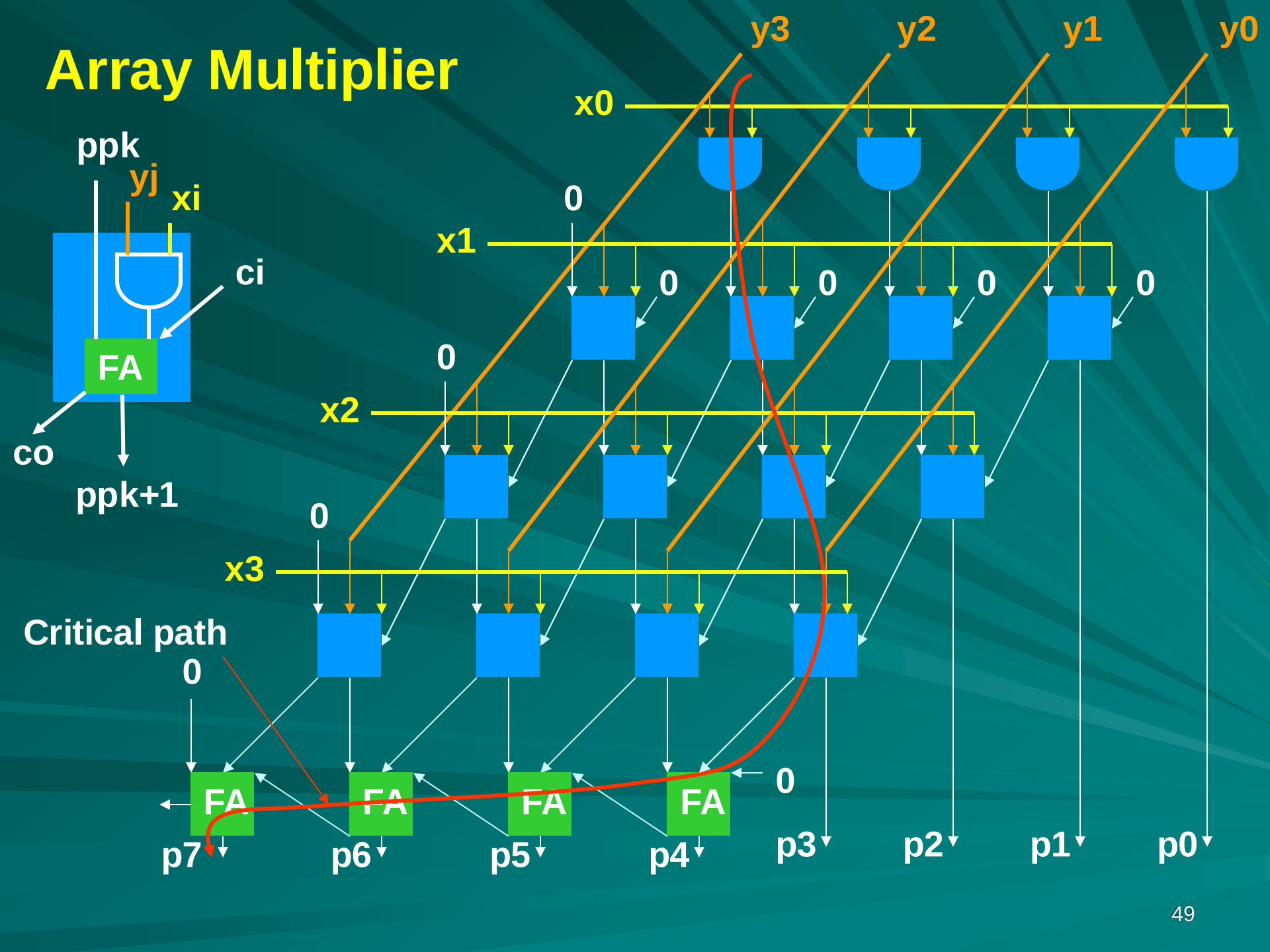


$p_{0i} = x_0 y_i$

0th partial product



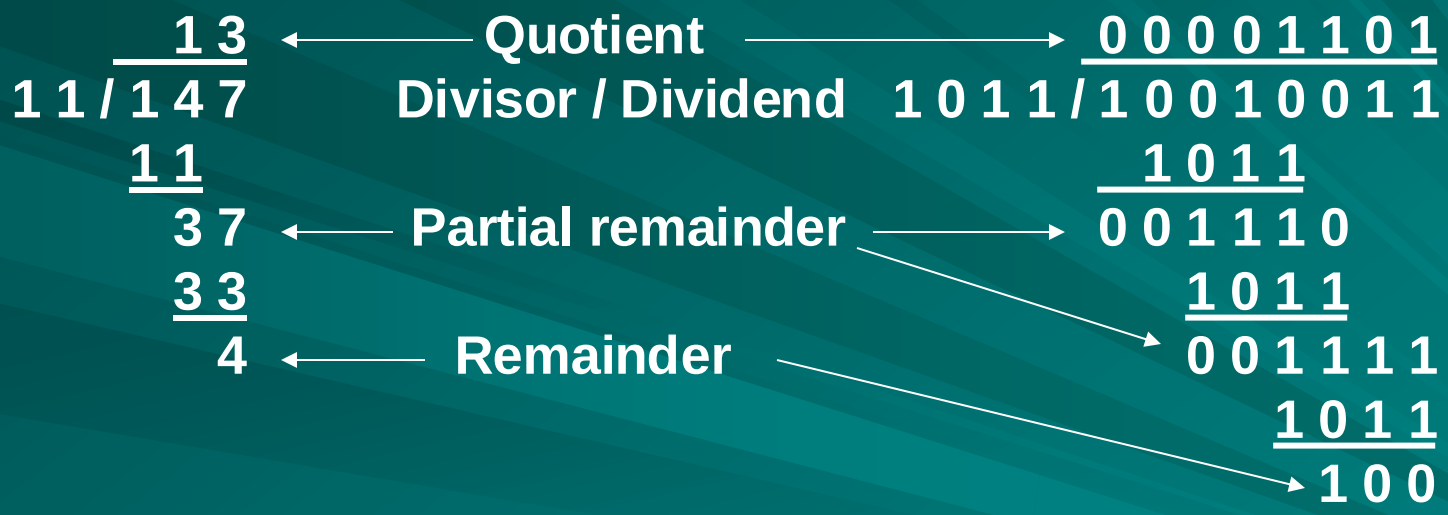
Array Multiplier



Types of Array Multipliers

- Baugh-Wooley Algorithm: Signed product by two's complement addition or subtraction according to the MSB's.
- Booth multiplier algorithm
- Tree multipliers
- Reference: N. H. E. Weste and D. Harris, *CMOS VLSI Design, A Circuits and Systems Perspective, Third Edition*, Boston: Addison-Wesley, 2005.

Binary Division (Unsigned)



4-bit Binary Division (Unsigned)

Dividend: 6 = 0110
Divisor: 4 = 0100
– 4 = 1100

6
— = 1, remainder 2
4



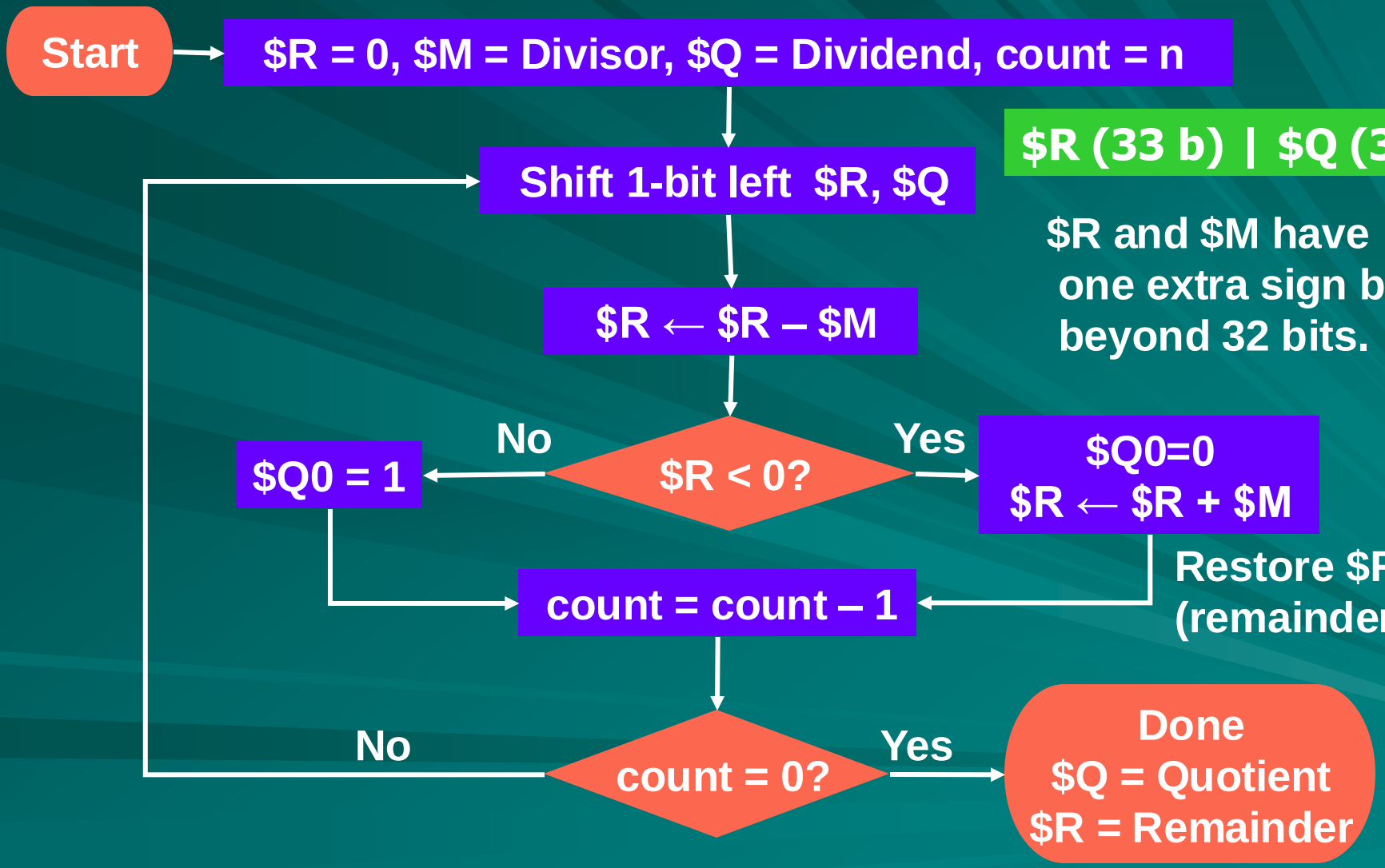
negative → quotient bit 0
→ restore remainder

negative → quotient bit 0
→ restore remainder

negative → quotient bit 0
→ restore remainder

positive → quotient bit 1

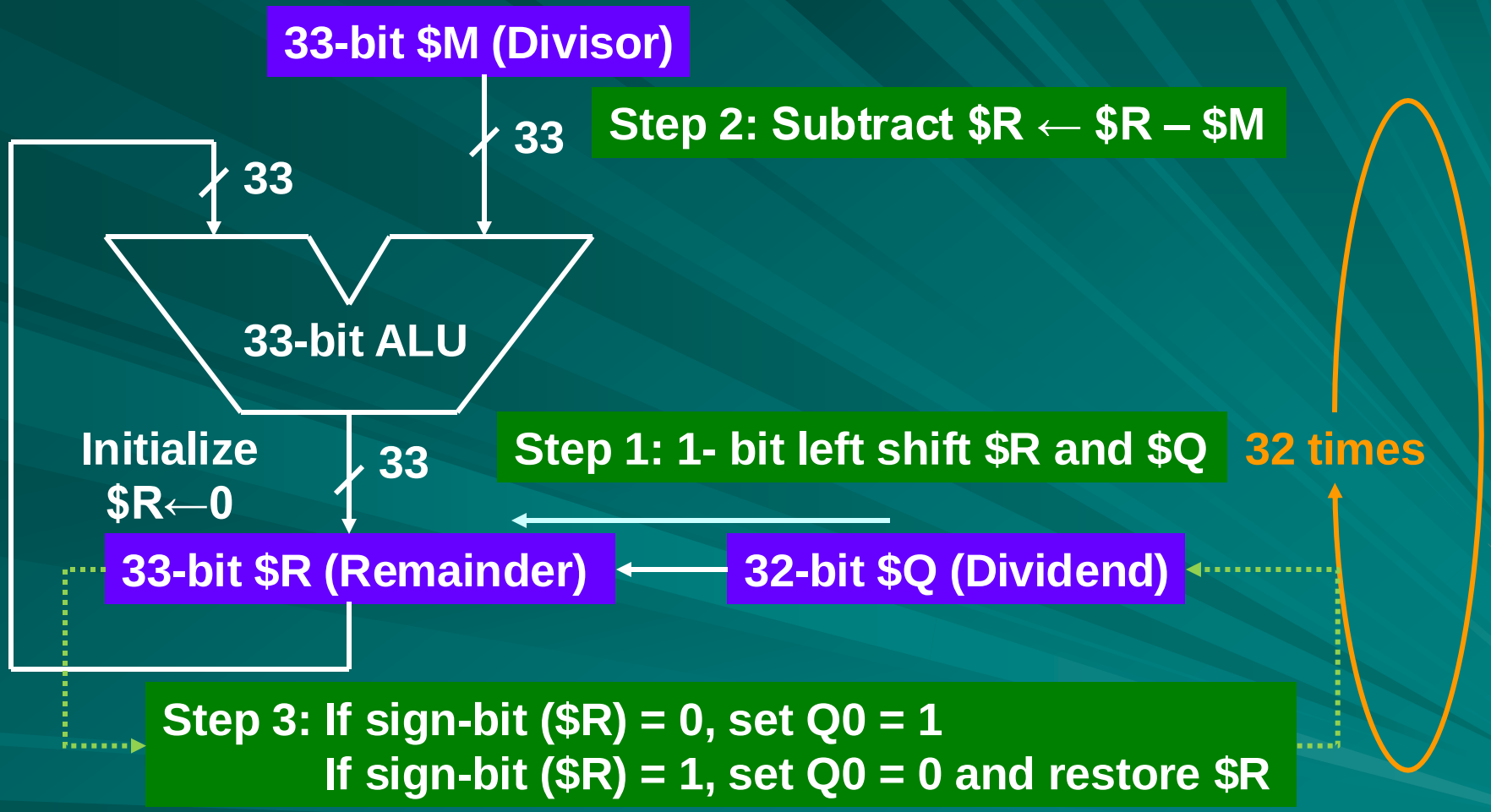
32-bit Binary Division Flowchart



4-bit Example: 6/4 = 1, Remainder 2

count	Actions	n	$R, $Q	$M = Divisor
4	Initialize	4	0 0 0 0 0 | 0 1 1 0	0 0 1 0 0
	Shift left $R, $Q	4	0 0 0 0 0 | 1 1 0 0	0 0 1 0 0
	Add – $M (11100) to $R	4	1 1 1 0 0 | 1 1 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	3	0 0 0 0 0 | 1 1 0 0	0 0 1 0 0
3	Shift left $R, $Q	3	0 0 0 0 1 | 1 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	3	1 1 1 0 1 | 1 0 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	2	0 0 0 0 1 | 1 0 0 0	0 0 1 0 0
2	Shift left $R, $Q	2	0 0 0 1 1 | 0 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	2	1 1 1 1 1 | 0 0 0 0	0 0 1 0 0
	Restore, add $M (00100) to $R	1	0 0 0 1 1 | 0 0 0 0	0 0 1 0 0
1	Shift left $R, $Q	1	0 0 1 1 0 | 0 0 0 0	0 0 1 0 0
	Add – $M (11100) to $R	1	0 0 0 1 0 | 0 0 0 0	0 0 1 0 0
	Set LSB of $Q = 1	0	0 0 0 1 0 | 0 0 0 1	0 0 1 0 0
0			Remainder | Quotient	

Division



V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization, Fourth Edition*, New York: McGraw-Hill, 1996.

Example: 8/3 = 2, Remainder = 2

Iteration 1

Initialize $R = 0 0 0 0 0 $Q = 1 0 0 0 $M = 0 0 0 1 1

Step 1, L-shift $R = 0 0 0 0 1 $Q = 0 0 0 0

Step 2, Add – $M = 1 1 1 0 1

$R = 1 1 1 1 0

Step 3, Set Q0 $Q = 0 0 0 0

Restore + $M = 0 0 0 1 1

$R = 0 0 0 0 1

Iteration 2

Step 1, L-shift $R = 0 0 0 1 0 $Q = 0 0 0 0 $M = 0 0 0 1 1

Step 2, Add – $M = 1 1 1 0 1

$R = 1 1 1 1 1

Step 3, Set Q0 $Q = 0 0 0 0

Restore + $M = 0 0 0 1 1

$R = 0 0 0 1 0

Example: 8/3 = 2 (Remainder = 2) (Continued)

Iteration 3

$R = 0 0 0 1 0 $Q = 0 0 0 0 $M = 0 0 0 1 1

Step 1, L-shift $R = 0 0 1 0 0 $Q = 0 0 0 0 $M = 0 0 0 1 1

Step 2, Add – $M = 1 1 1 0 1

$R = 0 0 0 0 1

Step 3, Set Q0 $Q = 0 0 0 1

Iteration 4

Step 1, L-shift $R.Q = 0 0 0 1 0 $Q = 0 0 1 0 $M = 0 0 0 1 1

Step 2, Add – $M = 1 1 1 0 1

$R = 1 1 1 1 1

Step 3, Set Q0 $Q = 0 0 1 0 *Final quotient*

Restore + $M = 0 0 0 1 1

$R = 0 0 0 1 0 *Remainder*

Note "Restore $R" in Steps 1, 2 and 4. This method is known as *the RESTORING DIVISION. An improved method, NON-RESTORING DIVISION, is possible (see Hamacher, et al.)*

Non-Restoring Division

- Avoid unnecessary addition (restoration).
- How it works?
  - Initially $R contains dividend $\times 2^{-n}$ for n-bit numbers. Example (n = 8):

Dividend	00101101
$R, $Q	00000000 00101101

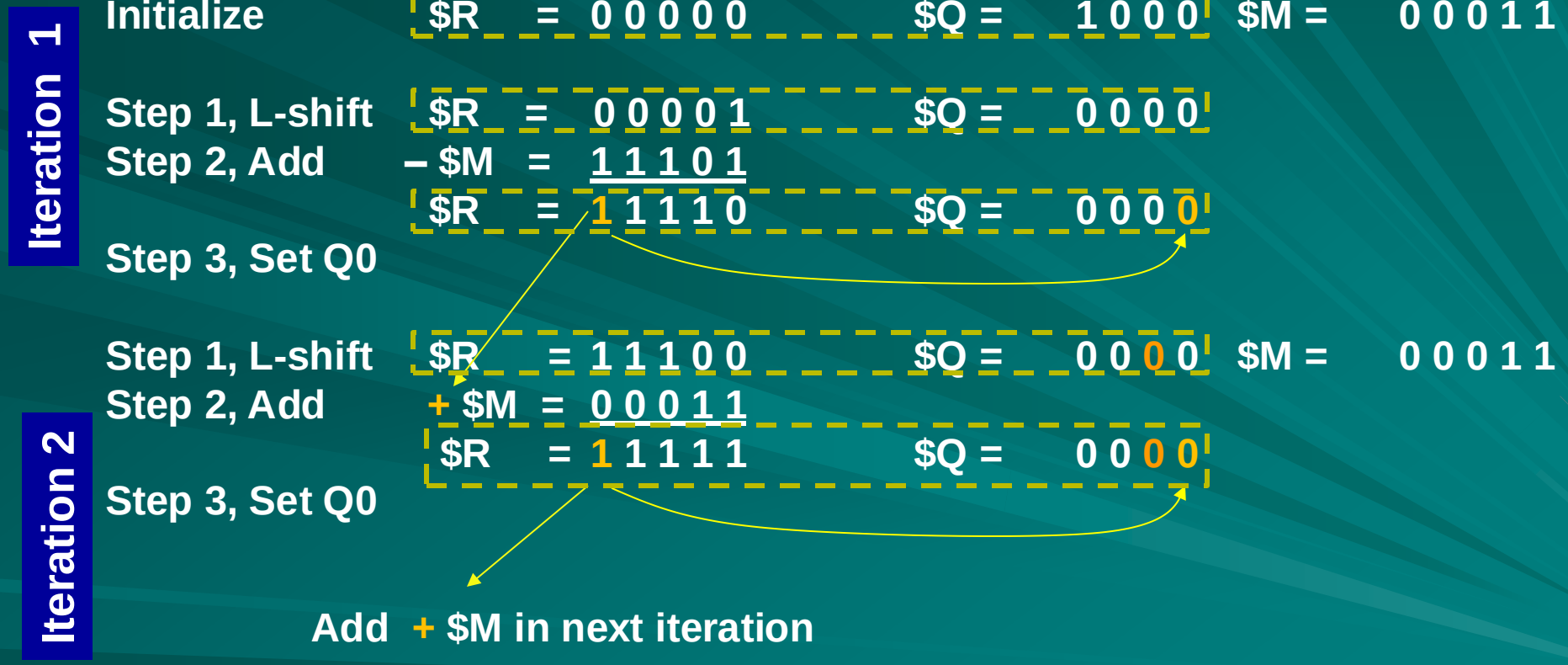
  - In some iteration after left shift, suppose $R = x and divisor is y
  - Subtract divisor, $R = x – y
  - Restore: If $R is negative, add y, $R = x
  - Next step: Left shift, $R = 2x+b, and subtract y, $R = 2x – y + b

How It Works: Last two Steps

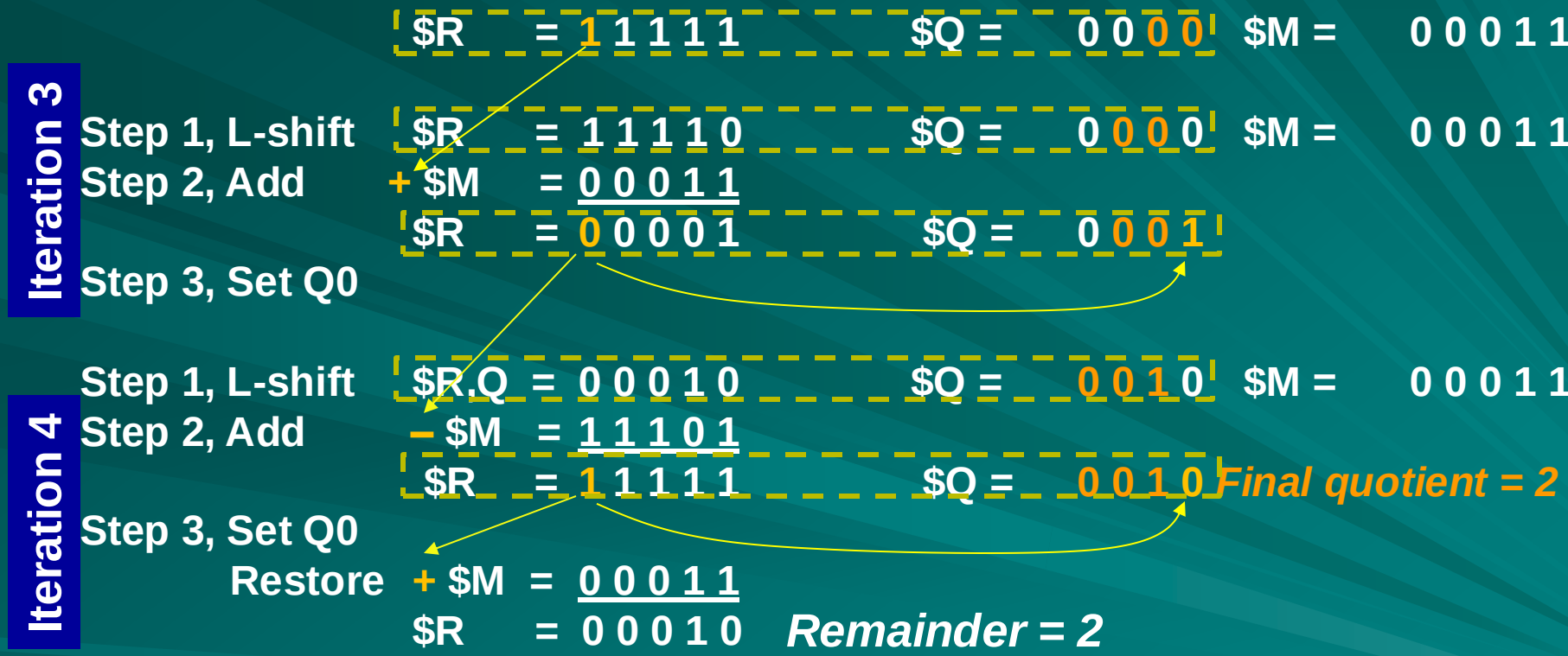
- *Suppose we do not restore and go to next step:*
- Left shift, $R = 2(x – y) + b = 2x – 2y + b, and **add** y, then $R = 2x – 2y + y + b = 2x – y + b (same result as with restoration)

- Non-restoring division
  - Initialize and start iterations same as in restoring division by subtracting divisor
  - In any iteration after left shift and subtraction/addition
    - If $R is positive, subtract divisor (y) in next iteration
    - If $R is negative, add divisor (y) in next iteration
  - After final iteration, if $R is negative then restore it by adding divisor (y)

Example: 8/3 = 2, Remainder = 2
Non-Restoring Division



Example: 8/3 = 2 (Remainder = 2) Non-Restoring Division (Continued)



See, V. C. Hamacher, Z. G. Vranesic and Z. G. Zaky, *Computer Organization, Fourth Edition*, McGraw-Hill, 1996, Section 6.9, pp. 281-285.

Signed Division

- Remember the signs and divide magnitudes.
- Negate the quotient if the signs of divisor and dividend disagree.
- There is no other direct division method for signed division.

Symbol Representation

- Early versions (60s and 70s)
  - Six-bit binary code (Control Data Corp., CDC)
  - EBCDIC – extended binary coded decimal interchange code (IBM)
- Presently used –
  - ASCII – American standard code for information interchange – 7 bit code specified by American National Standards Institute (ANSI), see Table 1.11 on page 63; an eighth MSB is often used as parity bit to construct a byte-code.
  - Unicode – 16 bit code and an extended 32 bit version

ASCII

- Each byte pattern represents a character (symbol)
  - Convenient to write in hexadecimal, e.g., with even parity,
    - 00000000 0_{ten} 00_{hex} null
    - 01000001 65_{ten} 41_{hex} A
    - 11100001 225_{ten} $E1_{hex}$ a
  - Table 1.11 on page 63 gives the 7-bit ASCII code.
  - C program – *string* – terminating with a null byte (odd parity):

01000101	01000011	01000101	10000000
69_{ten} or 45_{hex}	67_{ten} or 43_{hex}	69_{ten} or 45_{hex}	128_{ten} or 80_{hex}
E	C	E	(null)

Error Detection Code

- Errors: Bits can flip due to noise in circuits and in communication.
- Extra bits used for error detection.
- Example: a parity bit in ASCII code

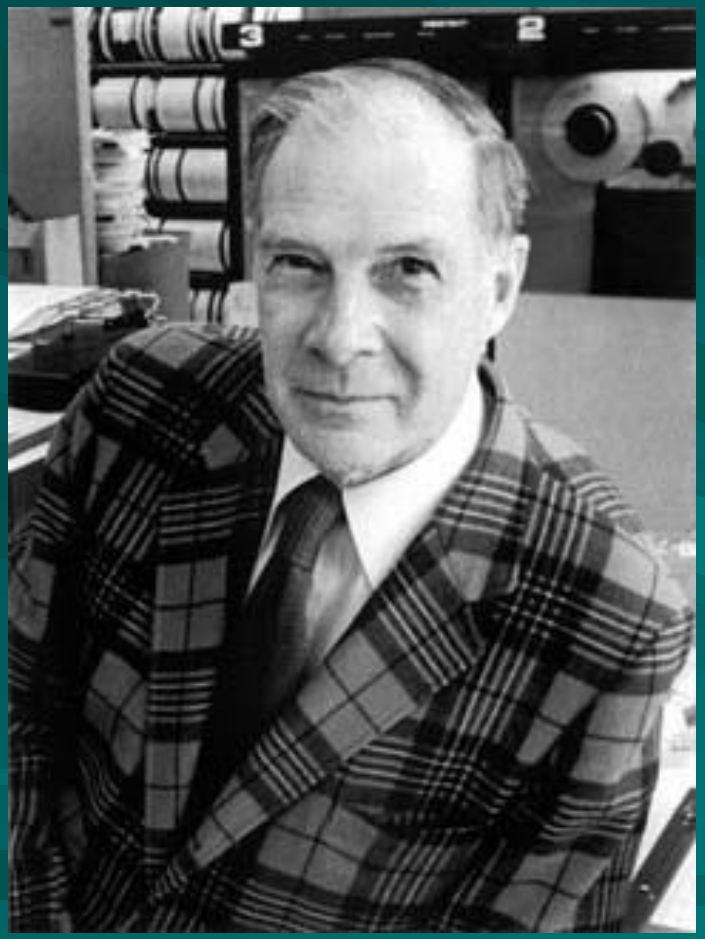


Even parity code for A (even number of 1s) 01000001

Odd parity code for A (odd number of 1s) 11000001

Single-bit error in 7-bit code of “A”, e.g., 1000101, will change symbol to “E” or 1000000 to “@”. But error will be detected in the 8-bit code because the error changes the specified parity.

Richard W. Hamming



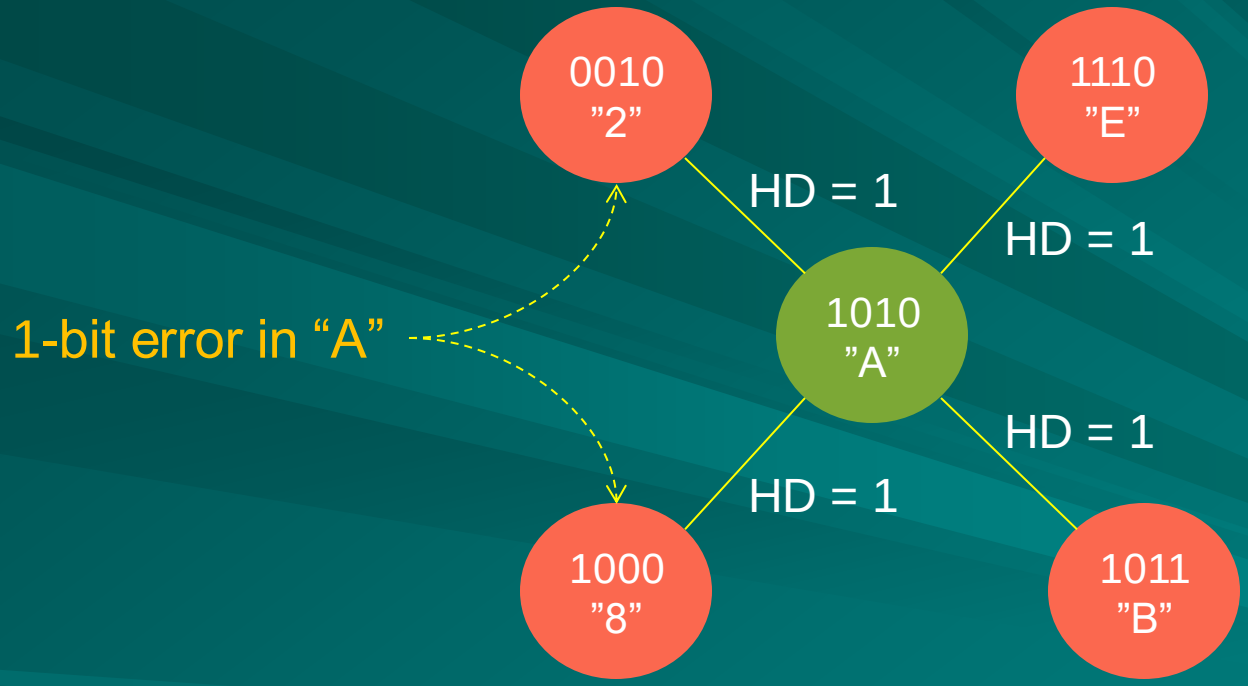
1915 -1998

- Error-correcting codes (ECC).
- Also known for
  - Hamming distance (HD) = Number of bits two binary vectors differ in
  - Example:

    HD(1101, 1010) = 3
  - Hamming Medal, 1988

The Idea of Hamming Code

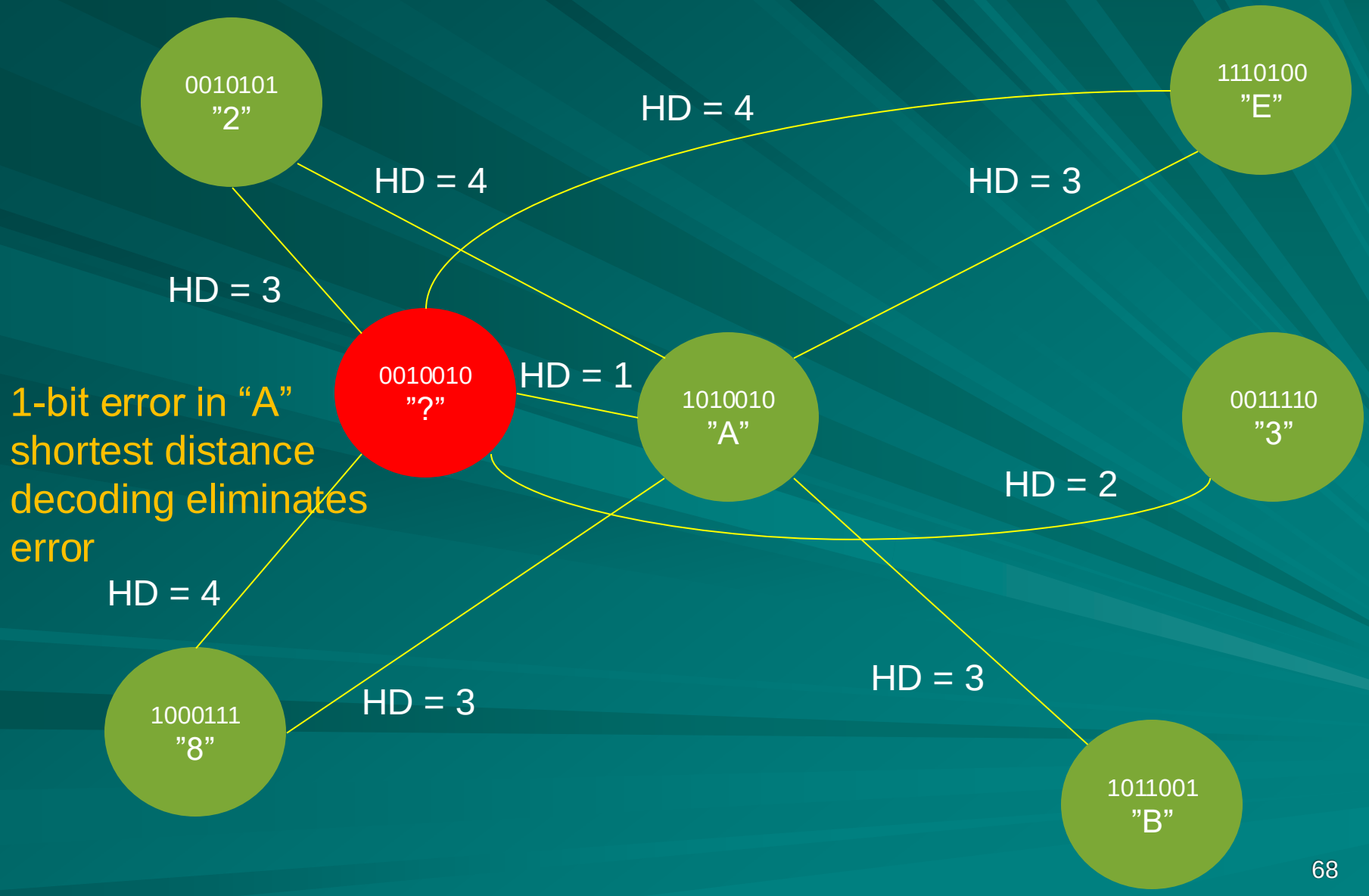
Code space contains 2^N possible N-bit code words:



Error not correctable. Reason: No redundancy.
Hamming’s idea: Increase HD between valid code words.

N = 4 Code	Symbol
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Hamming's Distance ≥ 3 Code



Minimum Distance-3 Hamming Code

Symbol	Original code	Odd-parity code	ECC, HD ≥ 3
0	0000	10000	0000000
1	0001	00001	0001011
2	0010	00010	0010101
3	0011	10011	0011110
4	0100	00100	0100110
5	0101	10101	0101101
6	0110	10110	0110011
7	0111	00111	0111000
8	1000	01000	1000111
9	1001	11001	1001100
A	1010	11010	1010010
B	1011	01011	1011001
C	1100	11100	1100001
D	1101	01101	1101010
E	1110	01110	1110100
F	1111	11111	1111111

Original code: Symbol “0” with a single-bit error will be Interpreted as “1”, “2”, “4” or “8”.

Reason: Hamming distance between codes is 1. A code with any bit error will map onto another valid code.

Remedy 1: Design codes with HD ≥ 2. Example: Parity code. Single bit error detected but not correctable.

Remedy 2: Design codes with HD ≥ 3. For single bit error correction, decode as the valid code at HD = 1.

For more error bit detection or correction, design code with HD ≥ 4.

Integers and Real Numbers

- Integers: the *universe* is infinite but discrete
  - No fractions
  - No numbers between consecutive integers, e.g., 5 and 6
  - A countable (finite) number of items in a finite range
  - Referred to as fixed-point numbers
- Real numbers – the *universe* is infinite and continuous
  - Fractions represented by decimal notation
    - Rational numbers, e.g., 5/2 = 2.5
    - Irrational numbers, e.g., 22/7 = 3.14159265 . . .
  - *Infinite* numbers exist even in the smallest range
  - Referred to as floating-point numbers

Wide Range of Numbers

- A large number:

  $976{,}000{,}000{,}000{,}000 = 9.76 \times 10^{14}$

- A small number:

  $0.000000000000000976 = 9.76 \times 10^{-14}$

Scientific Notation

- Decimal numbers
  - 0.513×10^5, 5.13×10^4 and 51.3×10^3 are written in scientific notation.
  - 5.13×10^4 is the *normalized* scientific notation.
- Binary numbers
  - Base 2
  - Binary point – multiplication by 2 moves the point to the right.
  - Normalized scientific notation, e.g., $1.0_{two} \times 2^{-1}$

Floating Point Numbers

- General format

$$\pm 1.bbbbb_{two} \times 2^{eeee}$$

or $$(-1)^S \times (1+F) \times 2^E$$

- Where
  - S = sign, 0 for positive, 1 for negative
  - F = fraction (or mantissa) as a binary integer, 1+F is called *significand*
  - E = *exponent* as a binary integer, positive or negative (two's complement)

Binary to Decimal Conversion

Binary $(-1)^S (1.b_1b_2b_3b_4) \times 2^E$

Decimal $(-1)^S \times (1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + b_4 \times 2^{-4}) \times 2^E$

Example: -1.1100×2^{-2} (binary) $= -(1 + 2^{-1} + 2^{-2}) \times 2^{-2}$

$= -(1 + 0.5 + 0.25)/4$

$= -1.75/4$

$= -0.4375$ (decimal)

William Morton (Velvel) Kahan

Architect of the IEEE floating point standard



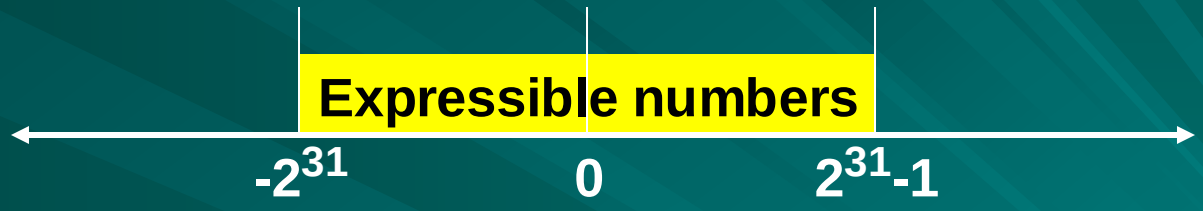
1989 Turing Award Citation:

For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations."

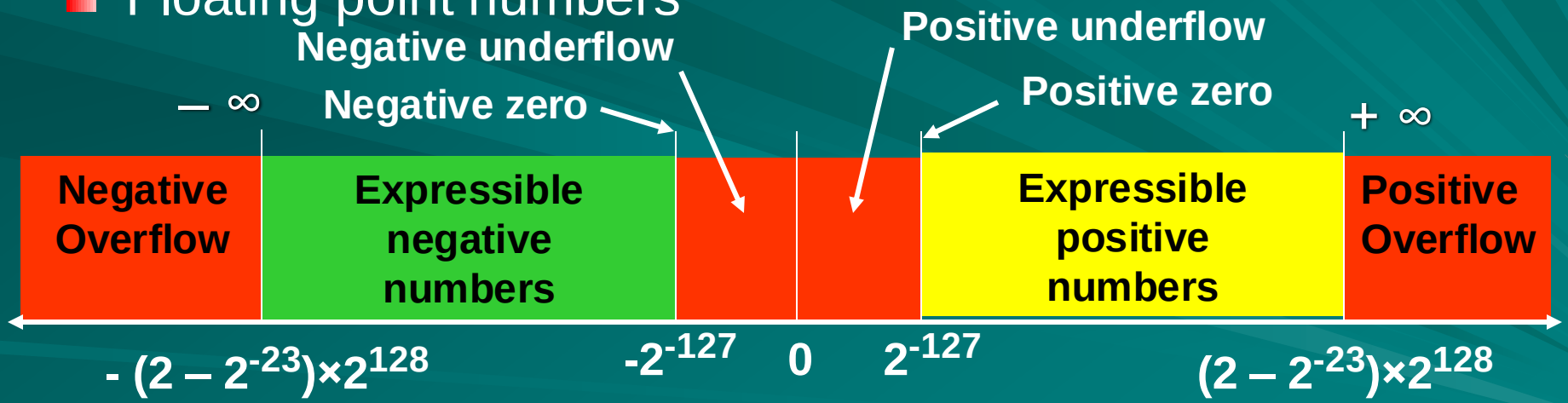
b. 1933, Canada
Professor of Computer Science, UC-Berkeley

Numbers in 32-bit Formats

- Two's complement integers



- Floating point numbers



- Ref: W. Stallings, *Computer Organization and Architecture*, Sixth Edition, Upper Saddle River, NJ: Prentice-Hall.

IEEE 754 Floating Point Standard

- Biased exponent: true exponent range [-126,127] is changed to [1, 254]:
  - *Biased exponent* is an 8-bit positive binary integer.
  - True exponent obtained by subtracting 127_{ten} or 01111111_{two}
- First bit of significand is always 1:

$$\pm 1.bbbb \ldots b \times 2^{E}$$

  - 1 before the binary point is implicitly assumed.
  - Significand field represents 23 bit fraction after the binary point.
  - Significand range is [1, 2), to be exact $[1, 2 - 2^{-23}]$

Examples

Biased exponent (0-255), bias 127 (01111111) to be subtracted

$1.1010001 \times 2^{10100}$ = 0 10010011 10100010000000000000000 = 1.6328125×2^{20}

$-1.1010001 \times 2^{10100}$ = 1 10010011 10100010000000000000000 = -1.6328125×2^{20}

$1.1010001 \times 2^{-10100}$ = 0 01101011 10100010000000000000000 = 1.6328125×2^{-20}

$-1.1010001 \times 2^{-10100}$ = 1 01101011 10100010000000000000000 = $-1.6328125 \times 2^{-20}$

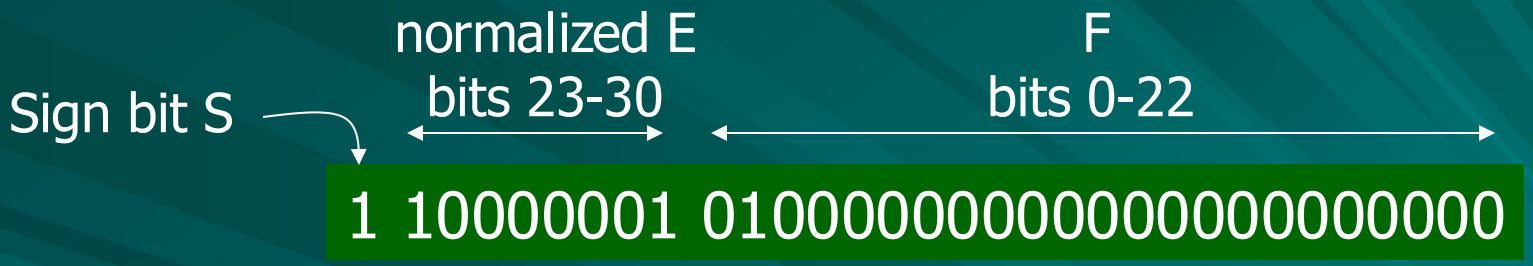
1.0
0.5
0.125
0.0078125
1.6328125

8-bit biased exponent
107 – 127
= – 20

Sign bit

23-bit Fraction (F)
of significand

Example: Conversion to Decimal



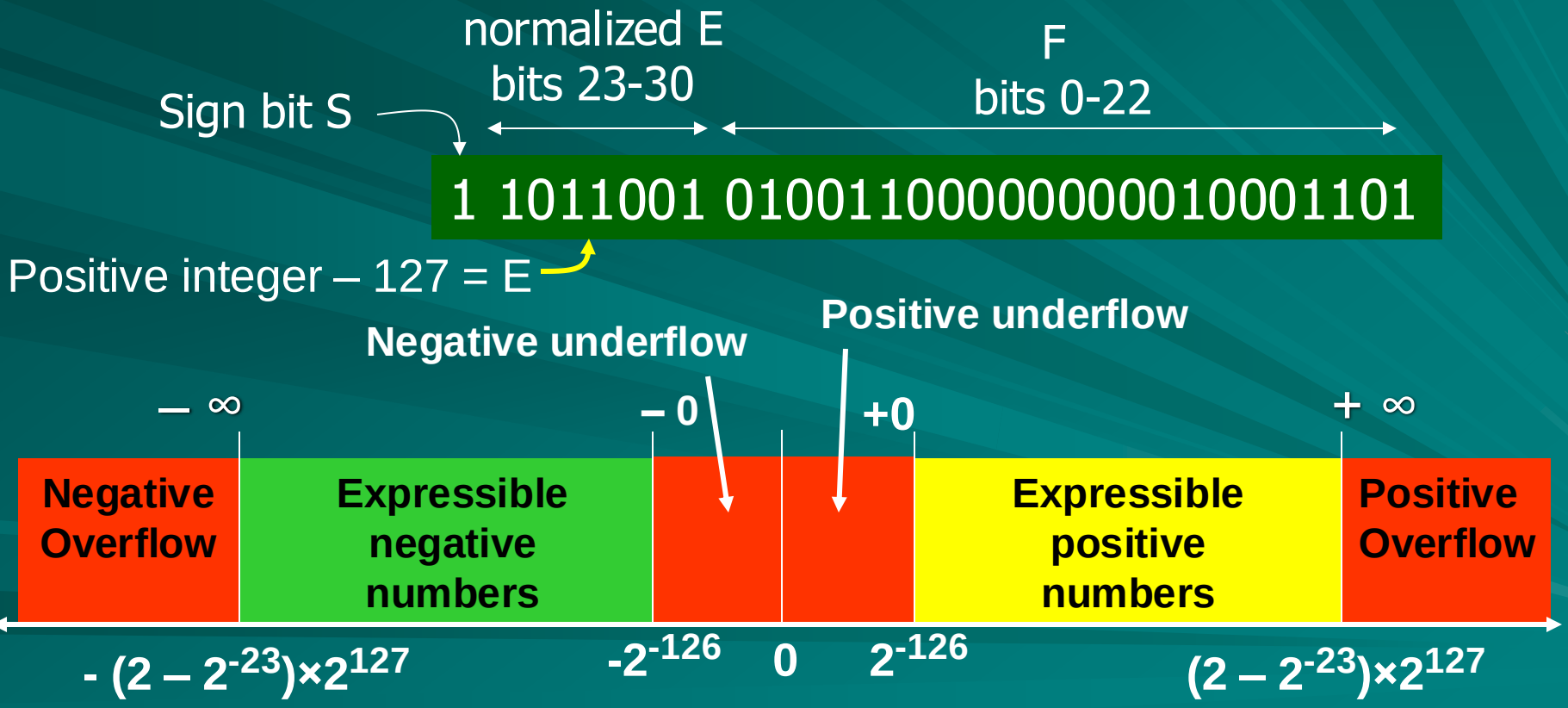
- Sign bit is 1, number is negative
- Biased exponent is $2^7+2^0 = 129$
- The number is

$(-1)^S \times (1 + F) \times 2^{(\text{exponent} - \text{bias})}$

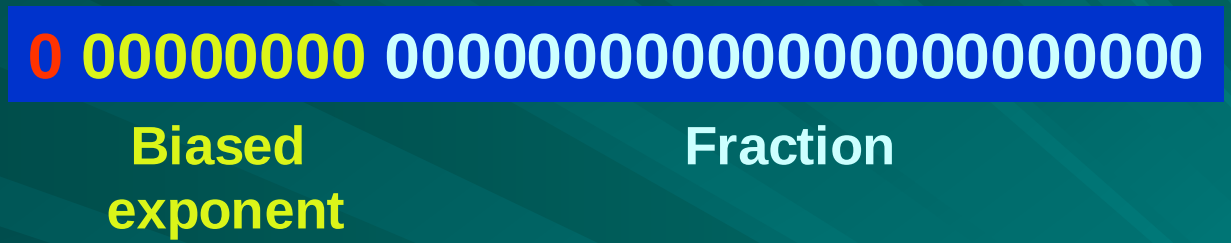
$= (-1)^1 \times (1 + F) \times 2^{(129 - 127)}$
$= - 1 \times 1.25 \times 2^2$
$= - 1.25 \times 4$
$= - 5.0$

IEEE 754 Floating Point Format

- Floating point numbers

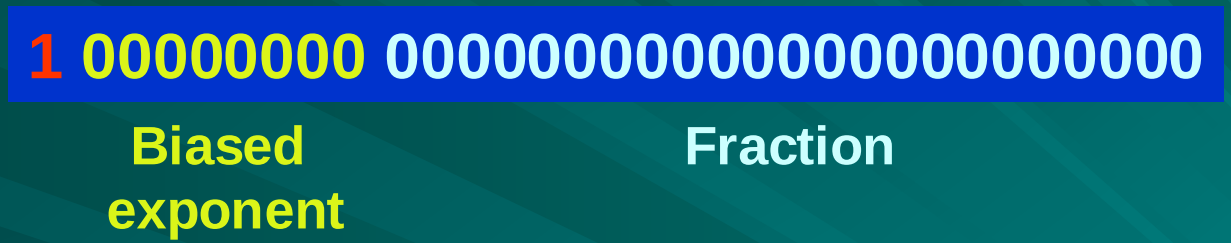


Positive Zero in IEEE 754



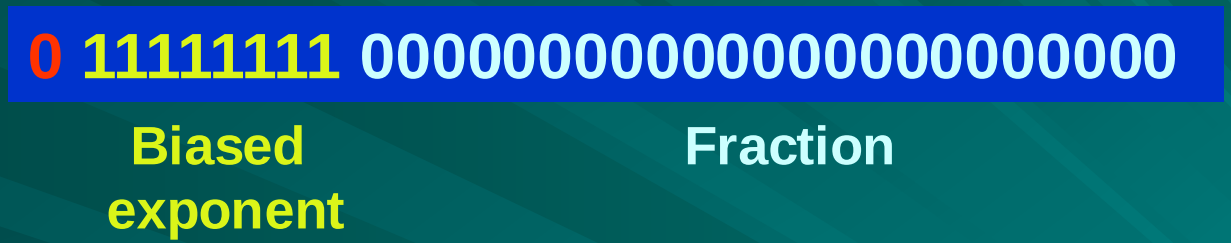
- $+ 1.0 \times 2^{-127}$
- Smallest positive number in single-precision IEEE 754 standard.
- Interpreted as *positive zero*.
- True exponent less than –126 is *positive underflow*; can be regarded as zero.

Negative Zero in IEEE 754



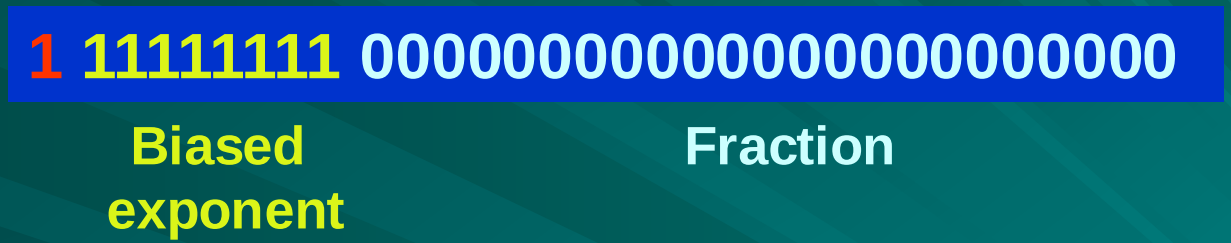
- -1.0×2^{-127}
- Smallest negative number in single-precision IEEE 754 standard.
- Interpreted as *negative zero*.
- True exponent less than –126 is *negative underflow*; may be regarded as 0.

Positive Infinity in IEEE 754



- $+ 1.0 \times 2^{128}$
- Largest positive number in single-precision IEEE 754 standard.
- Interpreted as $+ \infty$
- If true exponent = 128 and fraction ≠ 0, then the number is greater than ∞. It is called *"not a number" or NaN* and may be interpreted as ∞.

Negative Infinity in IEEE 754



- -1.0×2^{128}
- Smallest negative number in single-precision IEEE 754 standard.
- Interpreted as - ∞
- If true exponent = 128 and fraction ≠ 0, then the number is less than - ∞. It is called *"not a number" or NaN* and may be interpreted as - ∞.

Addition and Subtraction

0. Zero check
   - Change the sign of subtrahend, i.e., convert to summation
   - If either operand is 0, the other is the result
1. Significand alignment: right shift significand of smaller exponent until two exponents match.
2. Addition: add significands and report error if overflow occurs. If significand = 0, return result as 0.
3. Normalization
   - Shift significand bits to normalize.
   - report overflow or underflow if exponent goes out of range.
4. Rounding

Example (4 Significant Fraction Bits)

- Subtraction: $0.5_{ten} - 0.4375_{ten}$
- Step 0: Floating point numbers to be added

  $1.000_{two} \times 2^{-1}$ and $-1.110_{two} \times 2^{-2}$
- Step 1: Significand of lesser exponent is shifted right until exponents match

  $-1.110_{two} \times 2^{-2} \longrightarrow -0.111_{two} \times 2^{-1}$
- Step 2: Add significands, $1.000_{two} + (-0.111_{two})$

  Result is $0.001_{two} \times 2^{-1}$

```
 01000
+11001
 00001
```

2's complement addition, one bit added for sign

Example (Continued)

- Step 3: Normalize, $1.000_{two} \times 2^{-4}$

  No overflow/underflow since

  $127 \geq \text{exponent} \geq -126$

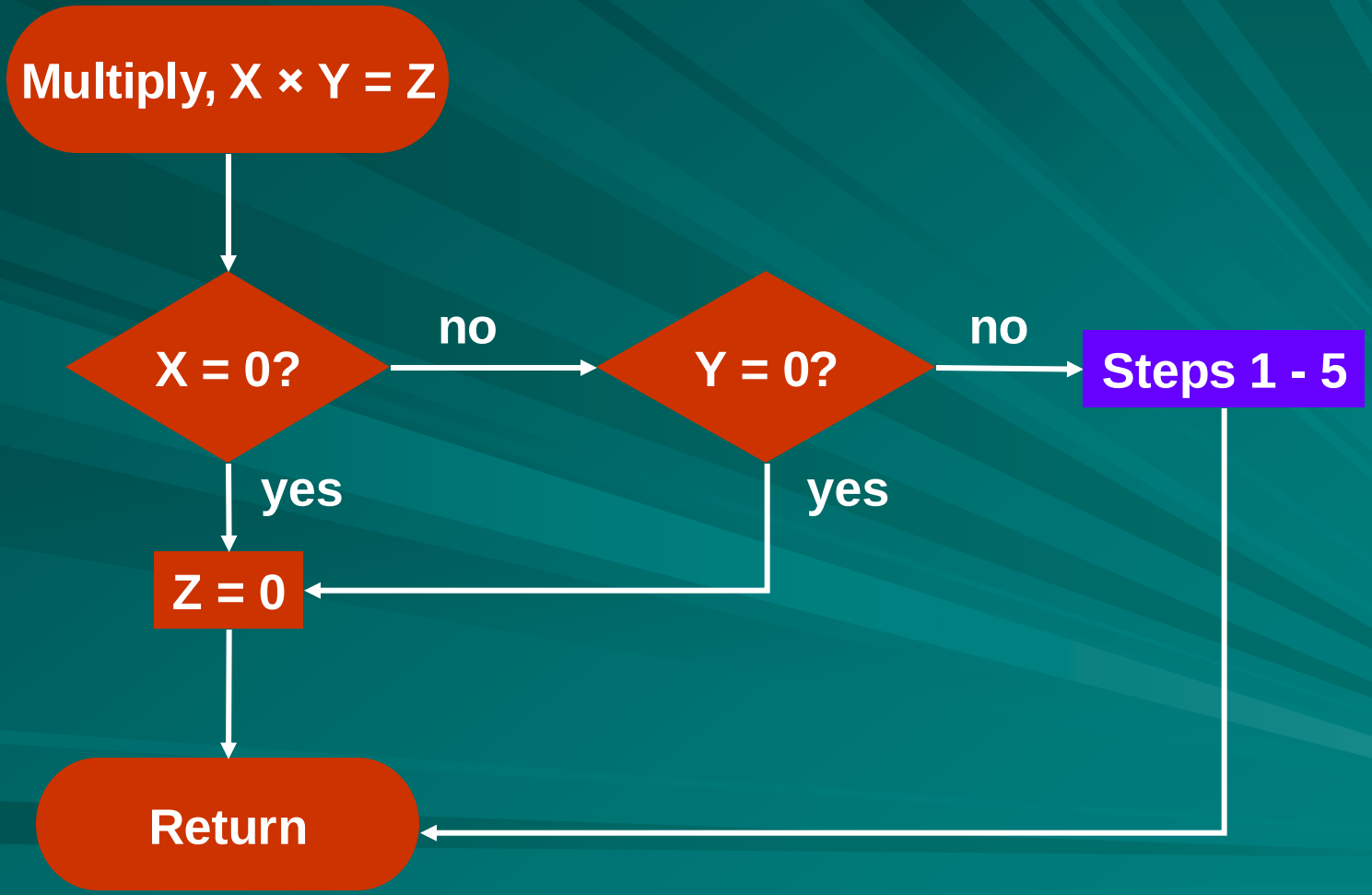
- Step 4: Rounding, no change since the sum fits in 4 bits.

$$1.000_{two} \times 2^{-4} = (1+0)/16 = 0.0625_{ten}$$

FP Multiplication: Basic Idea

1. Separate sign
2. Add exponents (integer addition)
3. Multiply significands (integer multiplication)
4. Normalize, round, check overflow/underflow
5. Replace sign

FP Multiplication: Step 0



FP Multiplication Illustration

- Multiply 0.5_{ten} and -0.4375_{ten} ($answer = -0.21875_{ten}$) or
- Multiply $1.000_{two} \times 2^{-1}$ and $-1.110_{two} \times 2^{-2}$
- Step 1: Add exponents

$$-1 + (-2) = -3$$

- Step 2: Multiply significands

```
  1.000
 ×1.110
  0000
 1000
1000
1000
1110000        Product is 1.110000
```

FP Mult. Illustration (Cont.)

- Step 3:
  - Normalization: If necessary, shift significand right and increment exponent.

    Normalized product is 1.110000×2^{-3}
  - Check overflow/underflow: $127 \geq \text{exponent} \geq -126$
- Step 4: Rounding: 1.110×2^{-3}
- Step 5: Sign: Operands have opposite signs,

  Product is -1.110×2^{-3}

(Decimal value = – (1+0.5+0.25)/8 = – 0.21875_{ten})

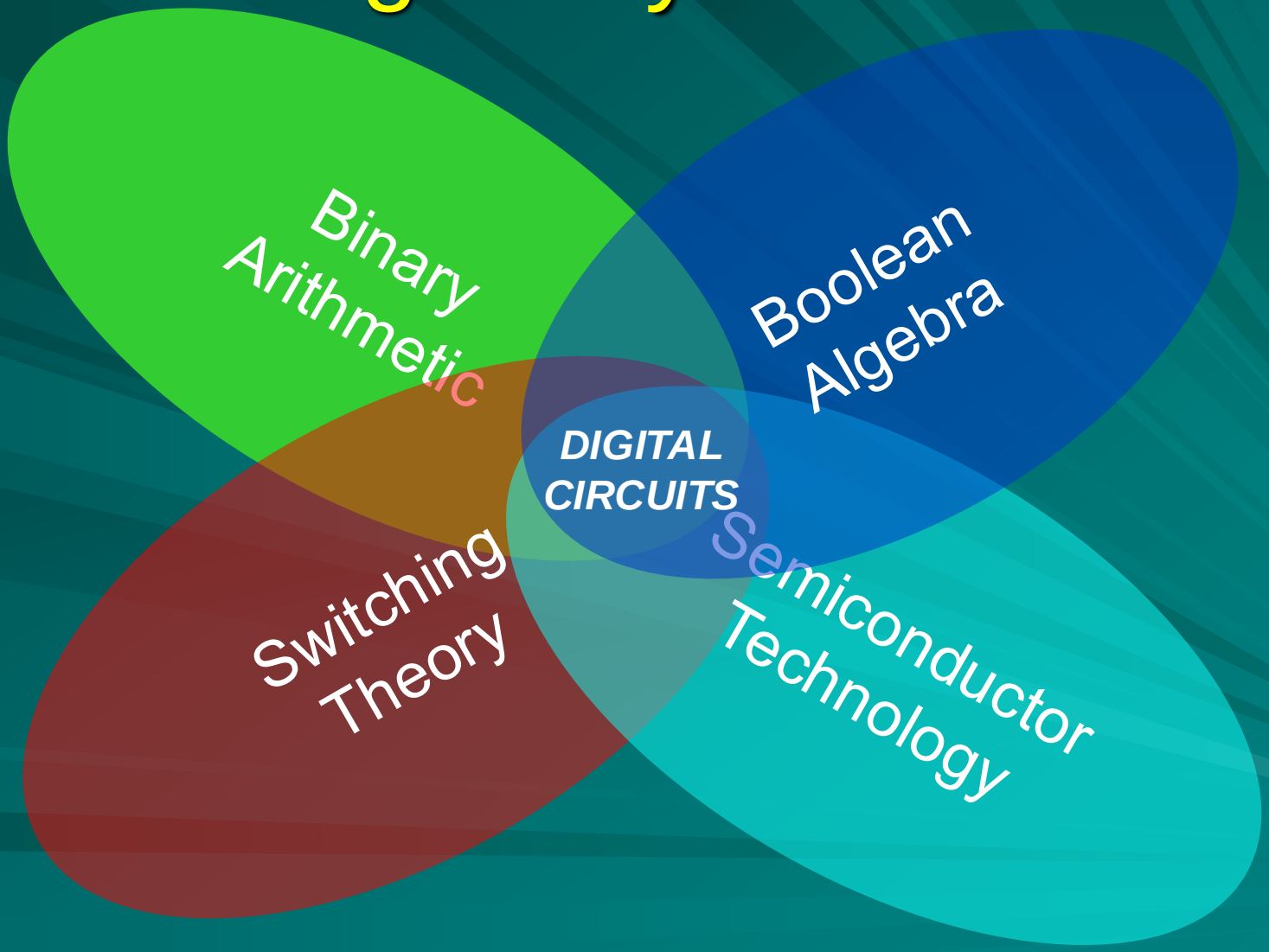
FP Division: Basic Idea

- Separate sign.
- Check for zeros and infinity.
- Subtract exponents.
- Divide significands.
- Normalize and detect overflow/underflow.
- Perform rounding.
- Replace sign.

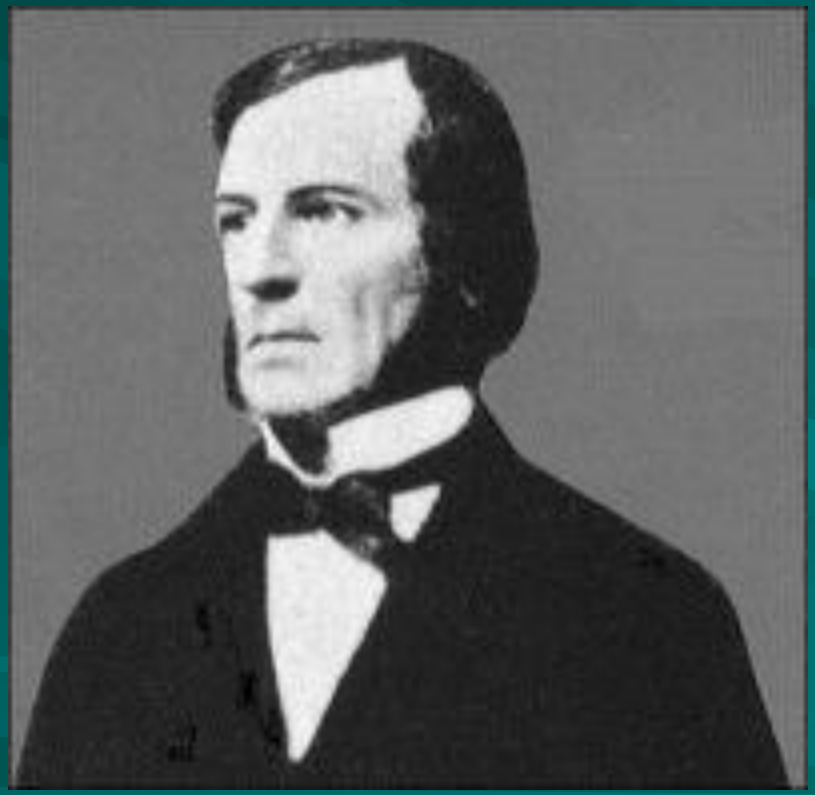
Digital Logic Circuits

Boolean Algebra (Chapter 2)

Digital Systems



George Boole, 1815-1864



- Born, Lincoln, England
- Professor of Math., Queen's College, Cork, Ireland
- Book, *The Laws of Thought*, 1853
- Wife: Mary *Everest* Boole

An Axiom or Postulate

- A self-evident or universally recognized truth.
- An established rule, principle, or law.
- A self-evident principle or one that is accepted as true without proof as the basis for argument.
- A postulate – Understood as the truth.

Boolean Algebra
Postulate 1: Set and Operators

- Define a set K containing two or more elements.
- Define two binary operators:
  - +, also called “OR”
  - ·, also called “AND”
  - Such that for any pair of elements, a and b in K, a + b and a·b also belong to K

Example

- a: {students in “digital circuits” course}
- b: {students in “computer systems” course}
- 1: {all EE juniors}
- 0: {null set}
- K: {a, b, 1, 0, a+b, a·b}
- Postulate 1:
  - a + b = 1
  - a · b = {full-time EE students}

Postulate 2: Identity Elements

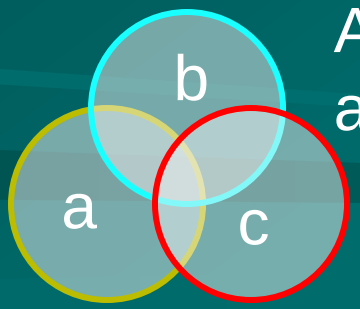
- There exist 0 and 1 elements in K, such that for every element a in K
  - $a + 0 = a$
  - $a \cdot 1 = a$
- Definitions:
  - 0 is the identity element for + operation
  - 1 is the identity element for · operation
- Remember, 0 and 1 here should not be misinterpreted as 0 and 1 of ordinary algebra.

Postulate 3: Commutativity

- Binary operators + and · are commutative.
- That is, for any elements a and b in K:
  - $a + b = b + a$
  - $a \cdot b = b \cdot a$
- Example:
  - $a + b = b + a$ = {all EE students}
  - $a \cdot b = b \cdot a$ = {all full-time EE students}

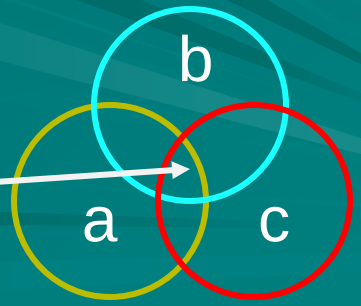
Postulate 4: Associativity

- Binary operators + and · are associative.
- That is, for any elements a, b and c in K:
  - a + (b + c) = (a + b) + c
  - a · (b · c) = (a · b) · c
- Example: EE department has three courses with student groups a, b and c



All EE students:
a + (b + c)

EE students in all EE courses: a · (b · c)



Postulate 5: Distributivity

- Binary operator + is distributive over · and · is distributive over +.
- That is, for any elements a, b and c in K:
  - $a + (b \cdot c) = (a + b) \cdot (a + c)$
  - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- Remember dot (·) operation is performed before + operation:

$$a + b \cdot c = a + (b \cdot c) \neq (a + b) \cdot c$$

Postulate 6: Complement

- A unary operation, *complementation*, exists for every element of K.
- That is, for any elements a in K:

$$a + \bar{a} = 1$$

$$a \cdot \bar{a} = 0$$

- Where, 1 is identity element for ·

  0 is identity element for +

Example

A set contains four elements:

x = {φ}, null set y = {1, 2}

z = {3, 4, 5} w = {1, 2, 3, 4, 5}

Define two operations: union (+) and intersection (·):

+	x	y	z	w
x	x	y	z	w
y	y	y	w	w
z	z	w	z	w
w	w	w	w	w

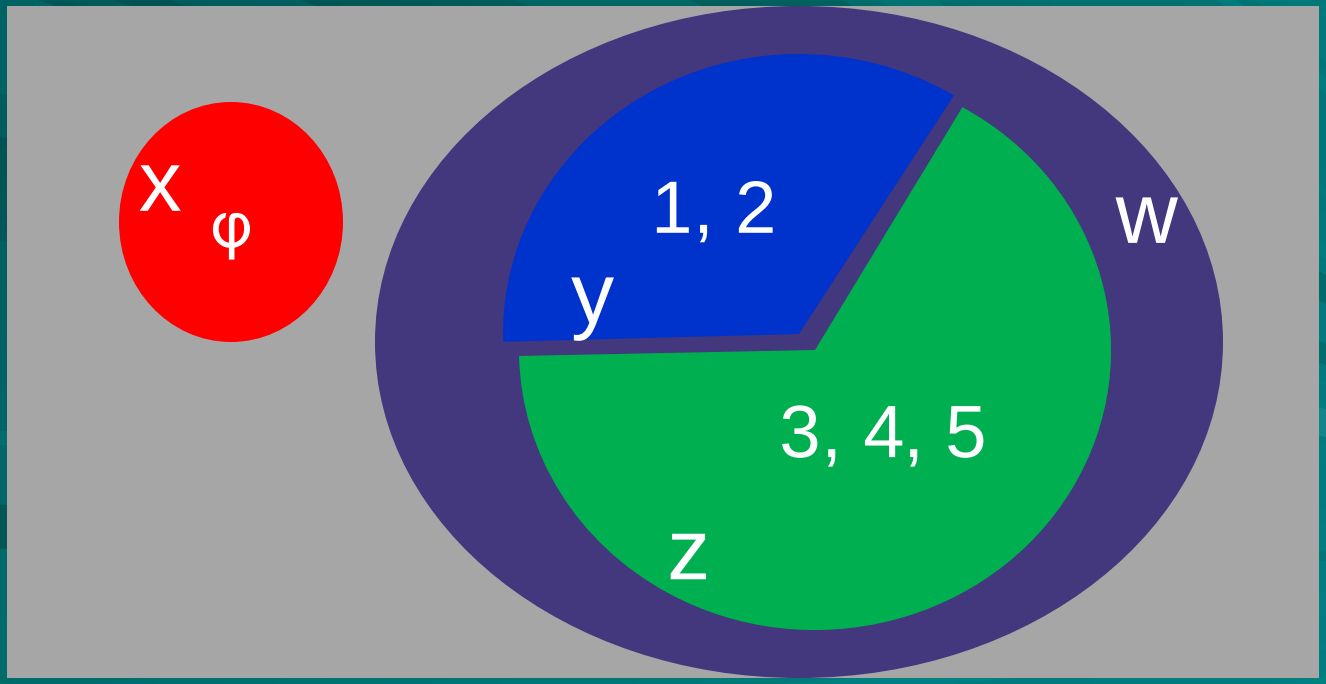
·	x	y	z	w
x	x	x	x	**x**
y	x	y	x	**y**
z	x	x	z	**z**
w	x	y	z	**w**

Verify Postulates 1, 2 and 3

1. Union and intersection, used as binary operators on a pair of elements, produce a result within the same set.
2. Identity elements are x for union (+) and w for intersection (·). $x \equiv 0$; $w \equiv 1$.
3. Commutativity is verified from the symmetry in the function tables for the two operators

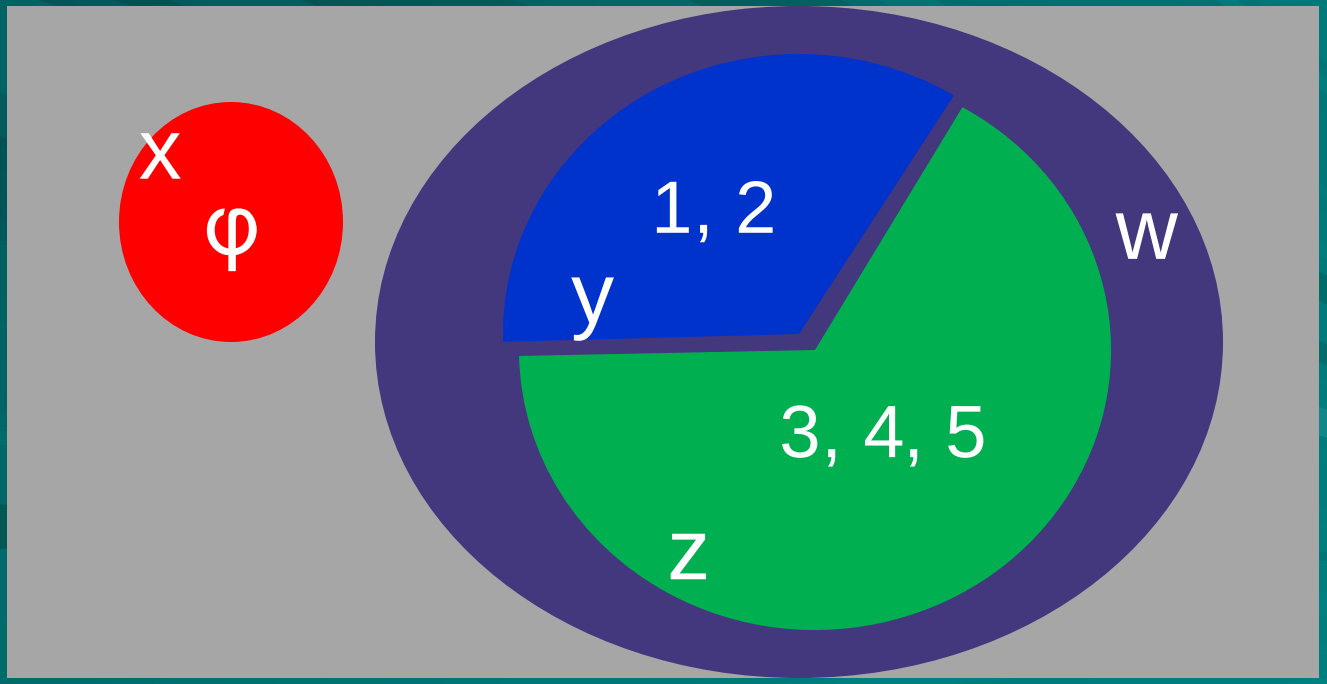
Postulate 4: Associativity

- Examine the Venn diagram. For any group of elements, intersections or unions in any order lead to the same result.



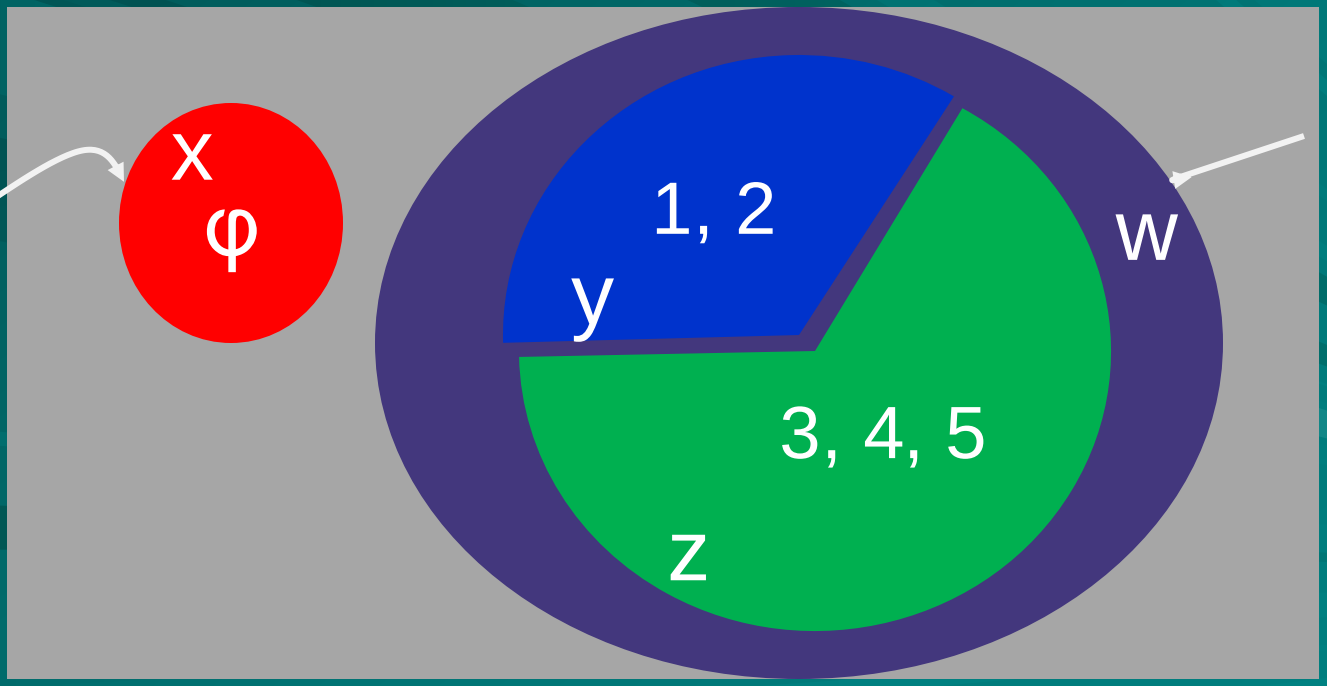
Postulate 5: Distributivity

- To verify distributivity, examine the Venn diagram for distributivity over union and intersection.



Postulate 6: Complements

- Any element + its complement = Identity for ·
- Any element · Its complement = Identity for +
- Verifiable from Venn diagram.



Conclusion

- Because all six postulates are true for our example, it is a Boolean algebra.

The Duality Principle

- Each postulate of Boolean algebra contains a pair of expressions or equations such that one is transformed into the other and vice-versa by interchanging the operators, $+ \leftrightarrow \cdot$, and identity elements, $0 \leftrightarrow 1$.
- The two expressions are called the duals of each other.

Examples of Duals

Postulate	Duals	
	Expression 1	Expression 2
1	$a, b, a + b \in K$	$a, b, a \cdot b \in K$
2	$a + 0 = a$	$a \cdot 1 = a$
3	$a + b = b + a$	$a \cdot b = b \cdot a$
4	$a + (b + c) = (a + b) + c$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
5	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
6	$a + \bar{a} = 1$	$a \bullet \bar{a} = 0$

Examples of Duals

Expressions: $A \cdot B$ $A + B$



Equations:

$A + (BC) = (A+B)(A+C)$ $\overset{\text{duals}}{\longleftrightarrow}$ $A\,(B+C) = AB + AC$

Note: $A \cdot B$ is also written as AB.

Properties of Boolean Algebra

- Properties stated as theorems.
- Provable from the postulates (axioms) of Boolean algebra.

Theorem 1: Idempotency (Invariance)

- For all elements a in K: a + a = a; a a = a.
- Proof:

a + a	=	(a + a)1,	(identity element)
	=	(a + a)(a + ā),	(complement)
	=	a + a ā,	(distributivity)
	=	a + 0,	(complement)
	=	a,	(identity element)

Theorem 1: Idempotency

- For all elements a in K: a + a = a; a a = a.
- Proof:

a a	=	(a a) + 0,	(identity element)
	=	(a a) + (a ā),	(complement)
	=	a (a + ā),	(distributivity)
	=	a 1,	(complement)
	=	a,	(identity element)

Theorem 2: Null Elements Exist

- $a + 1 = 1$, for + operator.
- $a \cdot 0 = 0$, for $\cdot$ operator.
- Proof:

$$
\begin{aligned}
a + 1 &= (a + 1)1, && \text{(identity element)} \\
&= 1(a + 1), && \text{(commutativity)} \\
&= (a + \bar{a})(a + 1), && \text{(complement)} \\
&= a + \bar{a}\,1, && \text{(distributivity)} \\
&= a + \bar{a}, && \text{(identity element)} \\
&= 1, && \text{(complement)}
\end{aligned}
$$

Similar proof for $a\,0 = 0$.

Theorem 2: Null Elements Exist

- $a + 1 = 1$, for + operator.
- $a \cdot 0 = 0$, for $\cdot$ operator.
- Proof:

$$
\begin{aligned}
a + 1 &= (a + 1)1, && \text{(identity element)} \\
&= 1(a + 1), && \text{(commutativity)} \\
&= (a + \bar{a})(a + 1), && \text{(complement)} \\
&= a + \bar{a}\,1, && \text{(distributivity)} \\
&= a + \bar{a}, && \text{(identity element)} \\
&= 1, && \text{(complement)}
\end{aligned}
$$

Similar proof for $a\,0 = 0$.

Theorem 2: Null Elements Exist

- a + 1 = 1, for + operator.
- a · 0 = 0, for · operator.
- Proof:

$$
\begin{aligned}
a\,0 &= (a\,0) + 0, && \text{(identity element)} \\
&= 0 + (a\,0), && \text{(commutativity)} \\
&= (a\,\bar{a}) + (a\,0), && \text{(complement)} \\
&= a(\bar{a} + 0), && \text{(distributivity)} \\
&= a\,\bar{a}, && \text{(identity element)} \\
&= 0, && \text{(complement)}
\end{aligned}
$$

Theorem 3: Involution Holds

- $\bar{\bar{a}} = a$
- Proof: $a + \bar{a} = 1$ and $a\,\bar{a} = 0$, (complements)

  or $\bar{a} + a = 1$ and $\bar{a}\,a = 0$, (commutativity)

  i.e., a is complement of $\bar{a}$

  Therefore, $\bar{\bar{a}} = a$

Theorem 4: Absorption

- $a + a\,b = a$
- $a\,(a + b) = a$
- Proof:

$$
\begin{aligned}
a + a\,b &= a\,1 + a\,b, && \text{(identity element)} \\
&= a(1 + b), && \text{(distributivity)} \\
&= a\,1, && \text{(Theorem 2)} \\
&= a, && \text{(identity element)}
\end{aligned}
$$

Similar proof for $a\,(a + b) = a$.

Theorems 5, 6 and 7 (p. 86-87)

- Theorem 5:

$$a + \bar{a}b = a + b$$

$$a(\bar{a} + b) = ab$$

- Theorem 6:

$$ab + a\bar{b} = a$$

$$(a + b)(a + \bar{b}) = a$$

- Theorem 7:

$$ab + a\bar{b}c = ab + ac$$

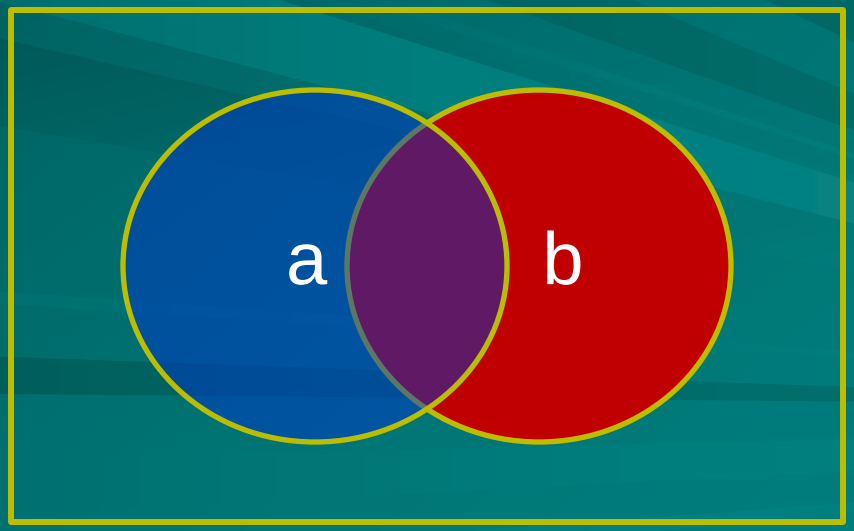
$$(a + b)(a + \bar{b} + c) = (a + b)(a + c)$$

Proving Theorem 5

$$a + \bar{a}b = a + b$$

$$a(\bar{a} + b) = ab$$

Using Venn diagram



Theorem 8: DeMorgan's Theorem

$$\overline{a+b} = \bar{a} \cdot \bar{b}, \quad \textit{for proof, see page 88}$$

$$\overline{a \cdot b} = \bar{a} + \bar{b}, \quad \textit{for proof, see page 88}$$

Generalization of DeMorgan's Theorem:

$$\overline{a+b+\cdots+z} = \bar{a} \cdot \bar{b} \cdots \bar{z}$$

$$\overline{a \cdot b \cdots z} = \bar{a} + \bar{b} + \cdots + \bar{z}$$

Martians and Venusians

- Suppose Martians are blue and Venusians are pink.
- An Earthling identifying itself: “I am not blue or pink.” $\overline{\text{blue} + \text{pink}} = \overline{\text{blue}} \cdot \overline{\text{pink}}$
- Meaning: “I am not blue and I am not pink.”
- Or: “I am not a Martian and I am not a Venusian.”

Tall, Dark and Handsome

- “He did not appear to be tall, dark and handsome.”

$$\overline{\text{tall} \cdot \text{dark} \cdot \text{handsome}} = \overline{\text{tall}} + \overline{\text{dark}} + \overline{\text{handsome}}$$

- Meaning: “He was not tall or he was not dark or he was not handsome.”
- Equivalently: “He was short or he was pale or he was ugly.”
- Perhaps, not the fellow we were looking for.

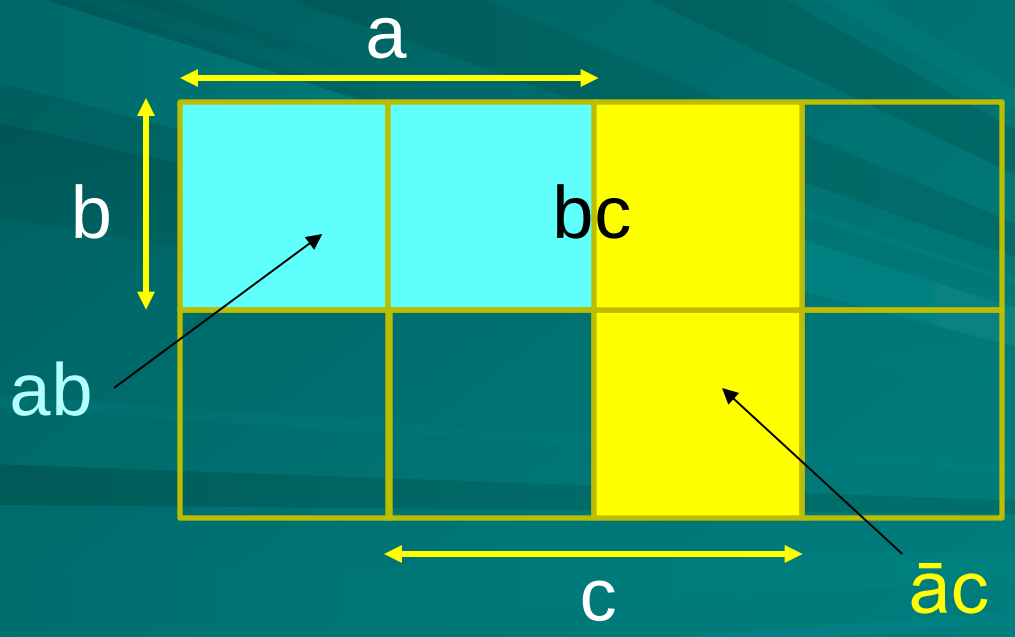
Theorem 9: Consensus

$$ab + \bar{a}c + bc = ab + \bar{a}c$$

$$(a + b)(\bar{a} + c)(b + c) = (a + b)(\bar{a} + c)$$

See page 90.

First case for *union* and *intersection:*



Next, Switching Algebra

- Set K contains two elements, {0, 1}, also called {false, true}, or {off, on}, etc.
- Two operations are defined as, + ≡ *OR*, · ≡ *AND.*

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

Digital Logic Circuits

Switching Algebra (Chapter 2)

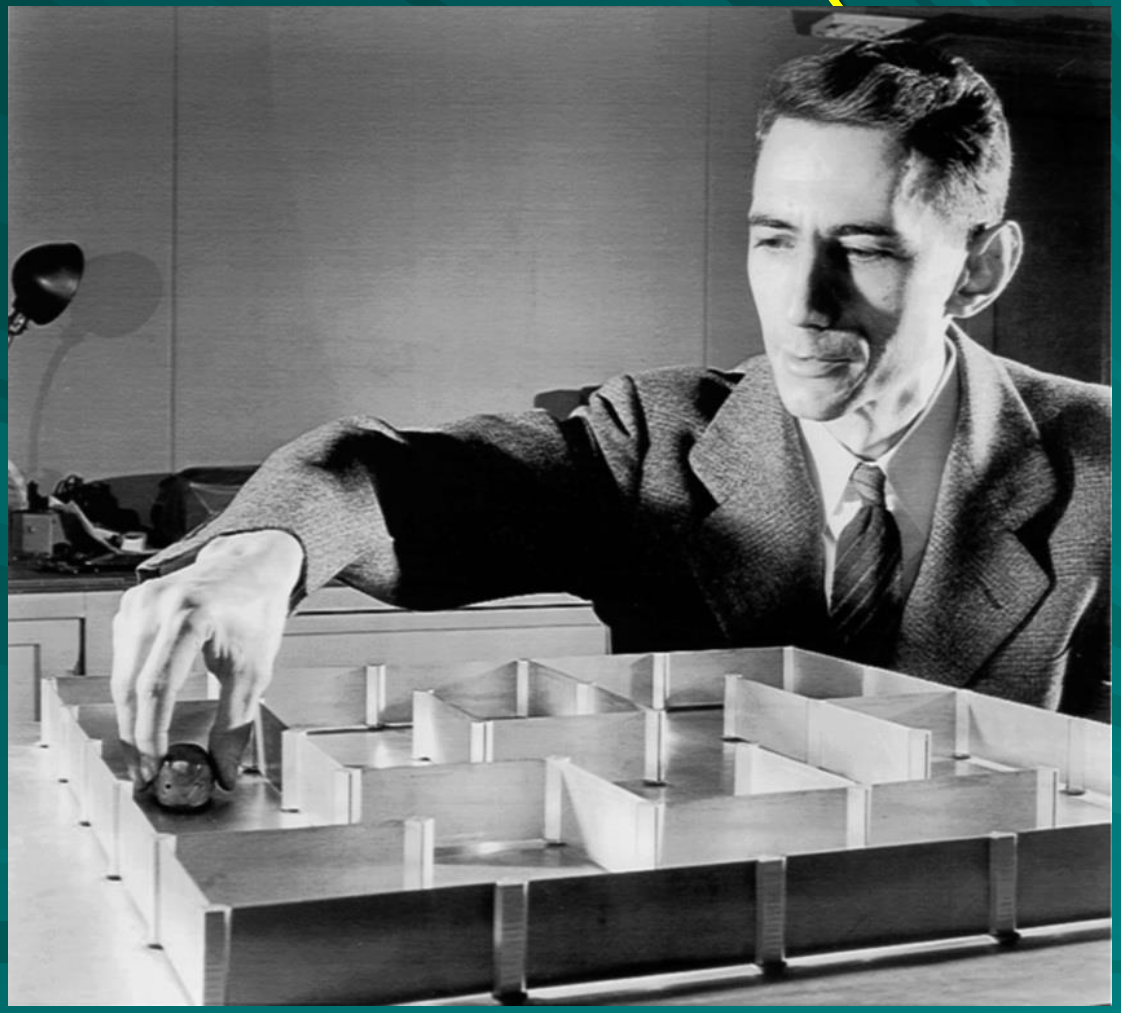
Switching Algebra

- A Boolean algebra, where
  - Set K contains just two elements, {0, 1}, also called {false, true}, or {off, on}, etc.
  - Two operations are defined as, + ≡ *OR*, · ≡ *AND.*

+	0	1
0	0	1
1	1	1

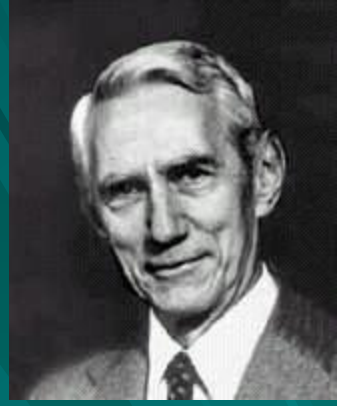
·	0	1
0	0	0
1	0	1

Claude E. Shannon (1916-2001)



http://www.kugelbahn.ch/sesam_e.htm

Shannon's Legacy



- *A Symbolic Analysis of Relay and Switching Circuits*, Master's Thesis, MIT, 1940. Perhaps the most influential master's thesis of the 20th century.
- *An Algebra for Theoretical Genetics,* PhD Thesis, MIT, 1940.
- Founded the field of Information Theory.
- C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication,* University of Illinois Press, 1949. A "must read."

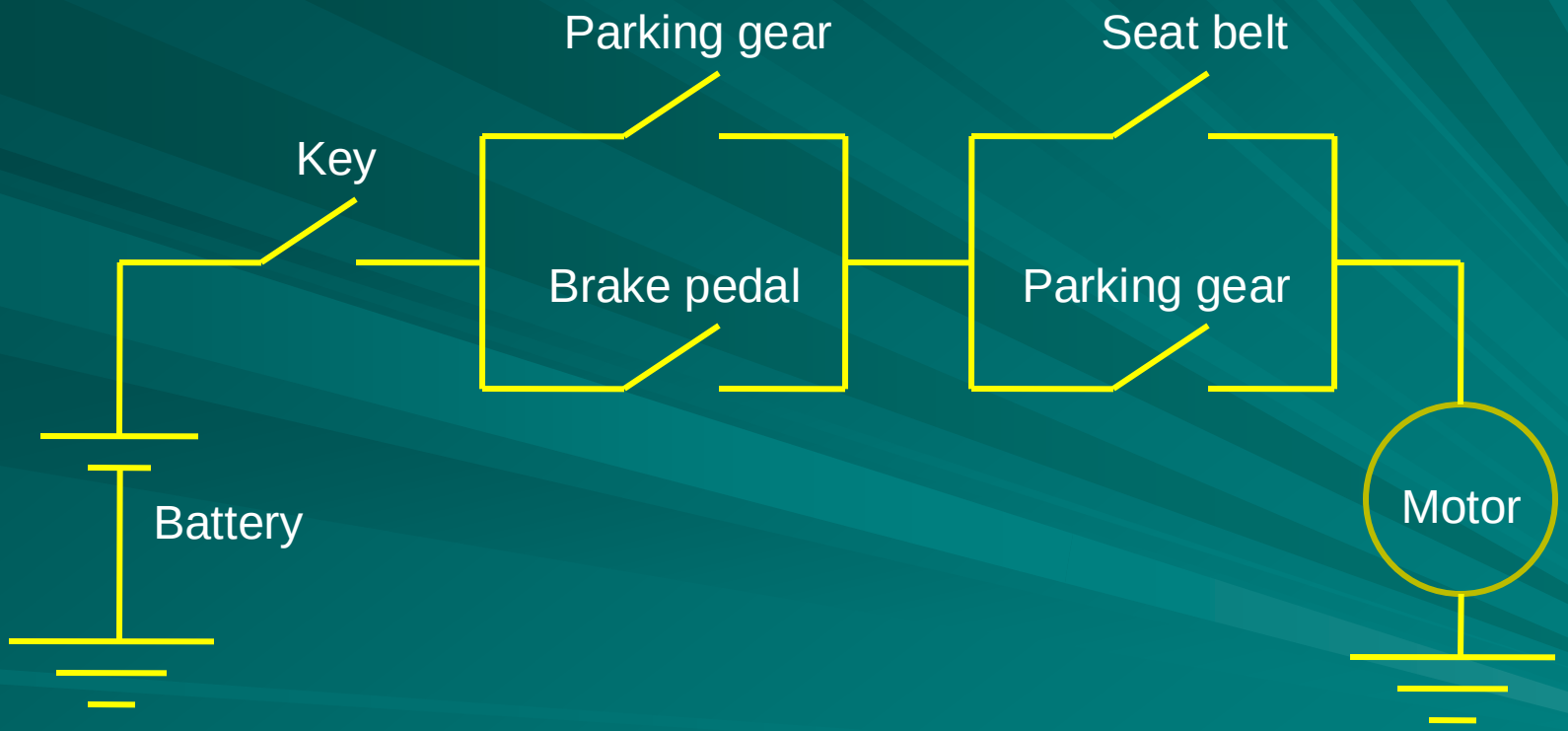
Switching Devices

- Electromechanical relays (1940s)
- Vacuum tubes (1950s)
- Bipolar transistors (1960 - 1980)
- Field effect transistors (1980 -)
- Integrated circuits (1970 -)
- Nanotechnology devices (future)

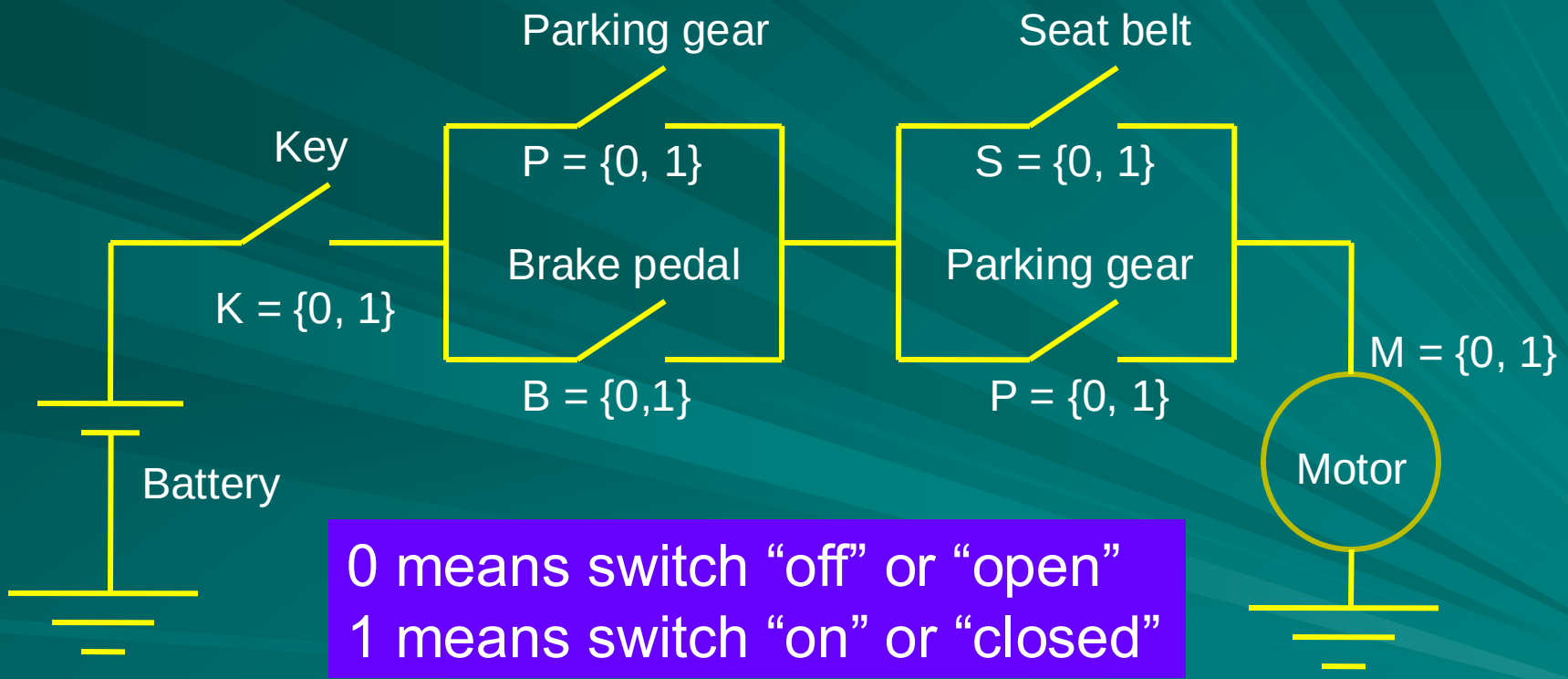
Example: Automobile Ignition

- Engine turns on when
- Ignition key is applied AND
  - Car is in parking gear OR
  - Brake pedal is on
- AND
  - Seat belt fastened OR
  - Car is in parking gear

Switching logic

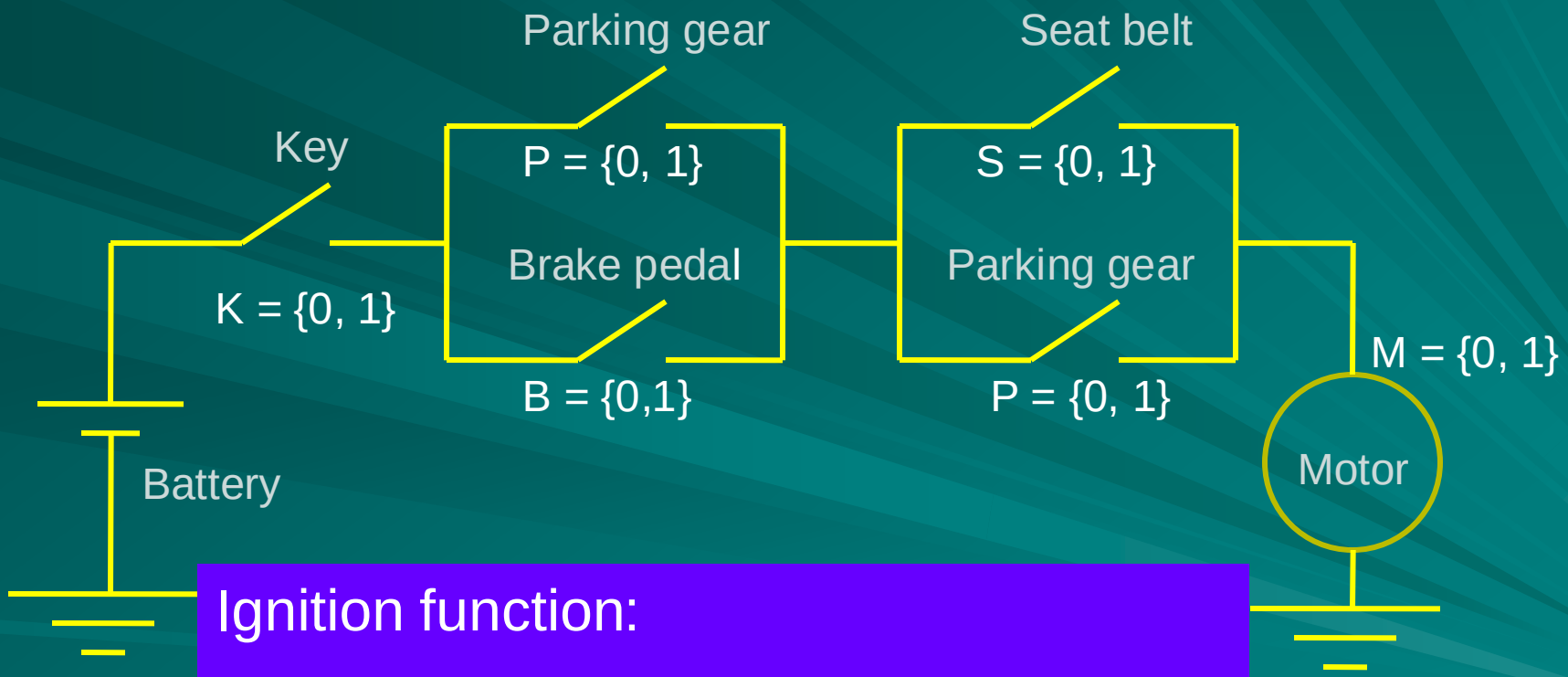


Define Boolean Variables



0 means switch “off” or “open”
1 means switch “on” or “closed”

Write Boolean Function



Ignition function:

M = K AND (P OR B) AND (S OR P)
= K(P + B)(S + P)

Simplify Boolean Function

M = K AND (P OR B) AND (S OR P)

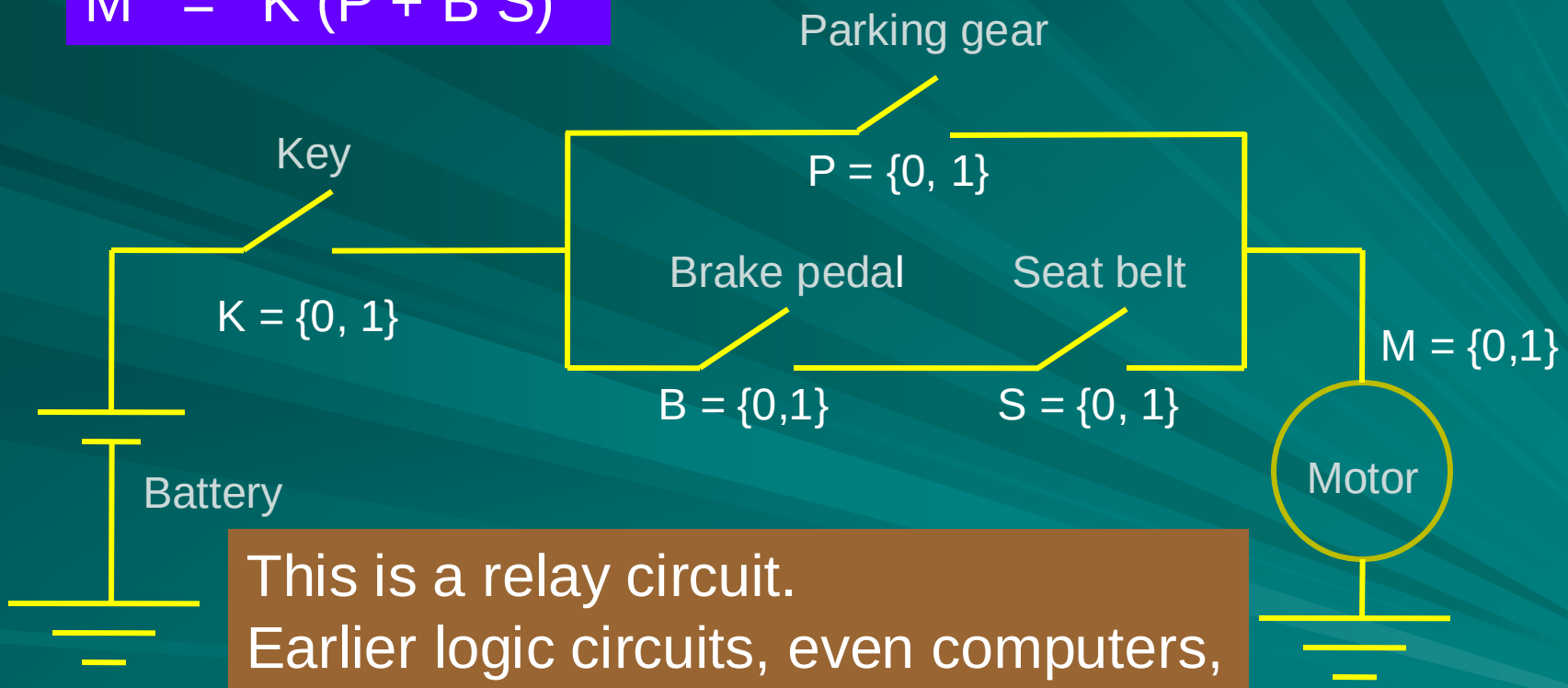
= K(P + B)(S + P)

= K(P + B)(P + S) *Commutativity*

= K (P + B S) *Distributivity*

Construct an Optimum Circuit

$$M = K(P + B\,S)$$



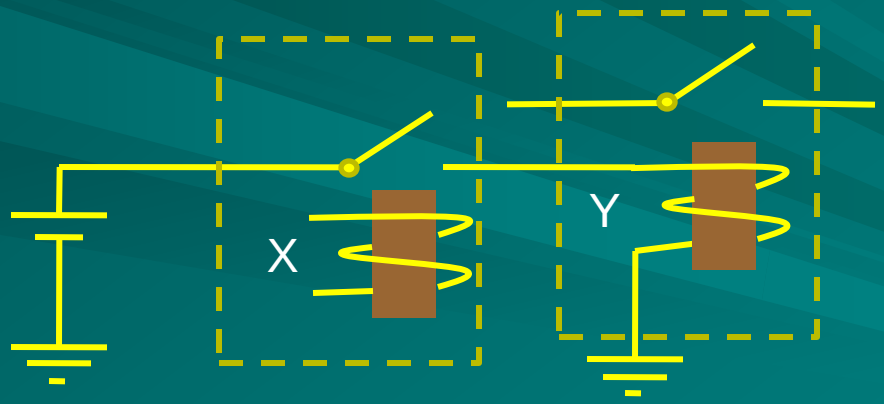
This is a relay circuit.
Earlier logic circuits, even computers, were built with relays.

Implementing with Relays

- An electromechanical relay contains:
  - Electromagnet
  - Current source
  - A switch, spring-loaded, normally open or closed
- Switch has two states, open (0) or closed (1).
- The state of switch is controlled by “not applying” or “applying” current to electromagnet.

One Switch Controlling Other

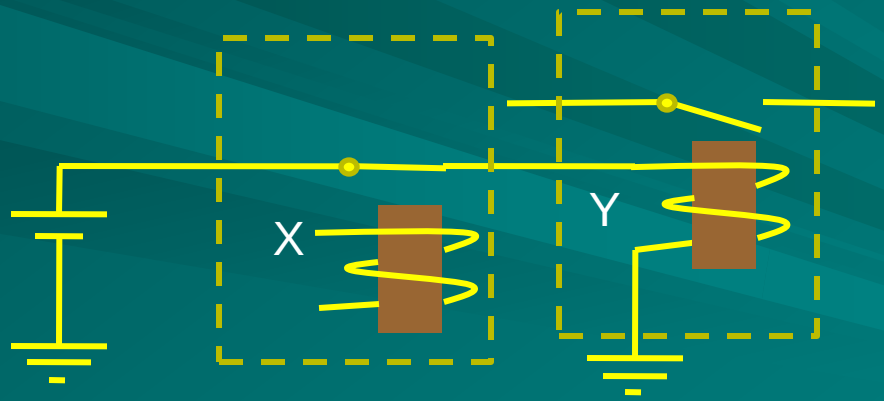
- Switches X and Y are normally open.
- Y cannot close unless a current is applied to X.



Y = X

Inverting Switch

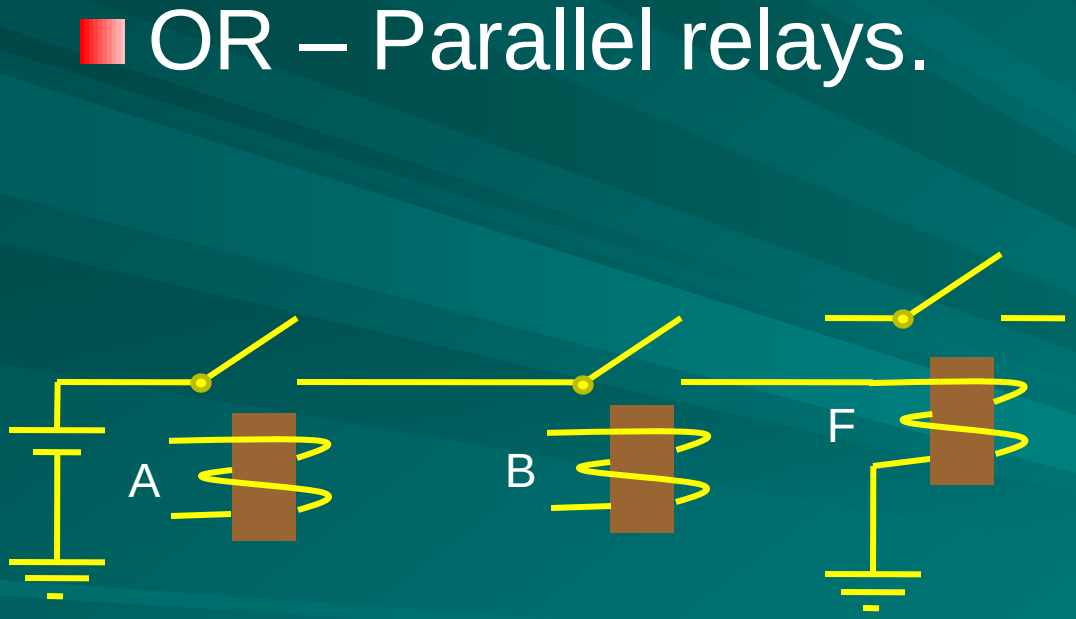
- Switch X is normally closed and Y is normally open.
- Y cannot open unless a current is applied to X.



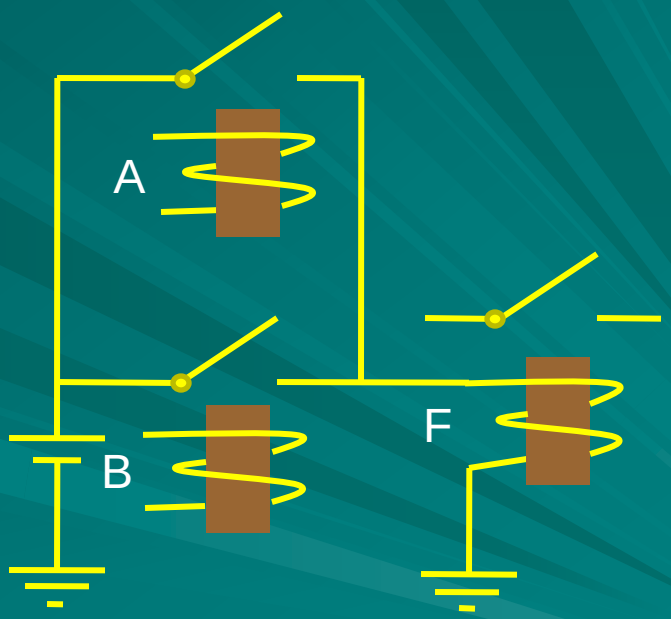
$$Y = \overline{X}$$

Boolean Operations

- AND – Series connected relays.
- OR – Parallel relays.

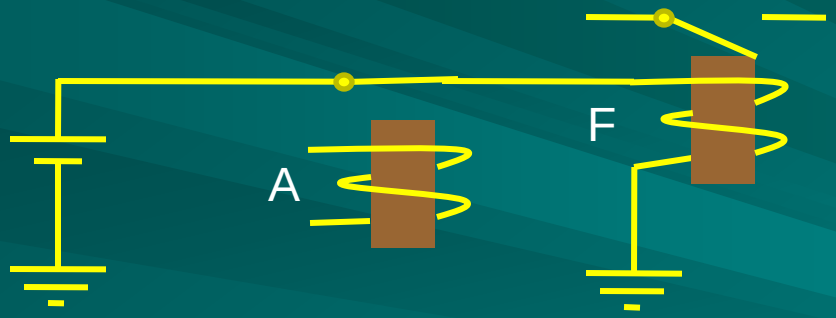


F = A B

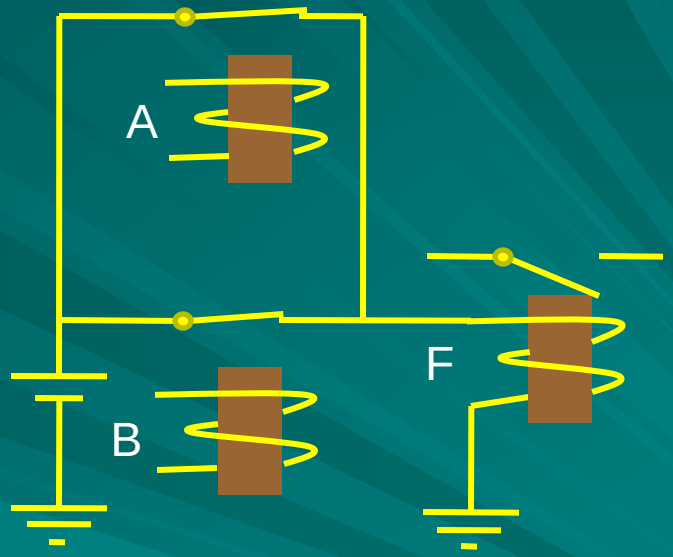


F = A + B

Complement (Inversion)



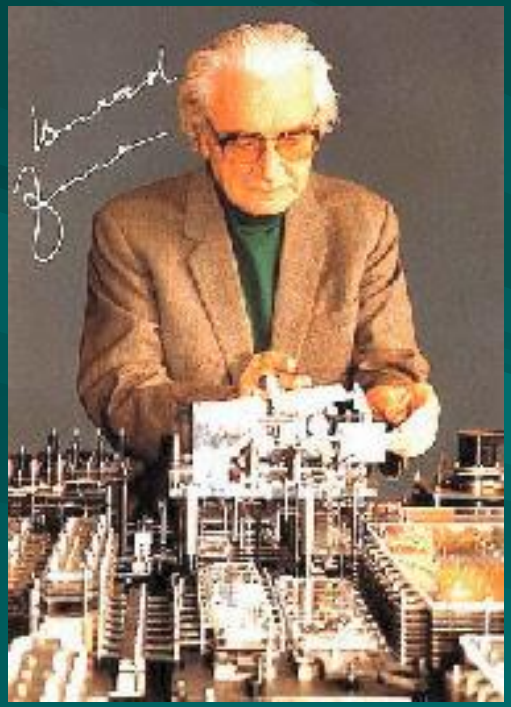
$F = \overline{A}$



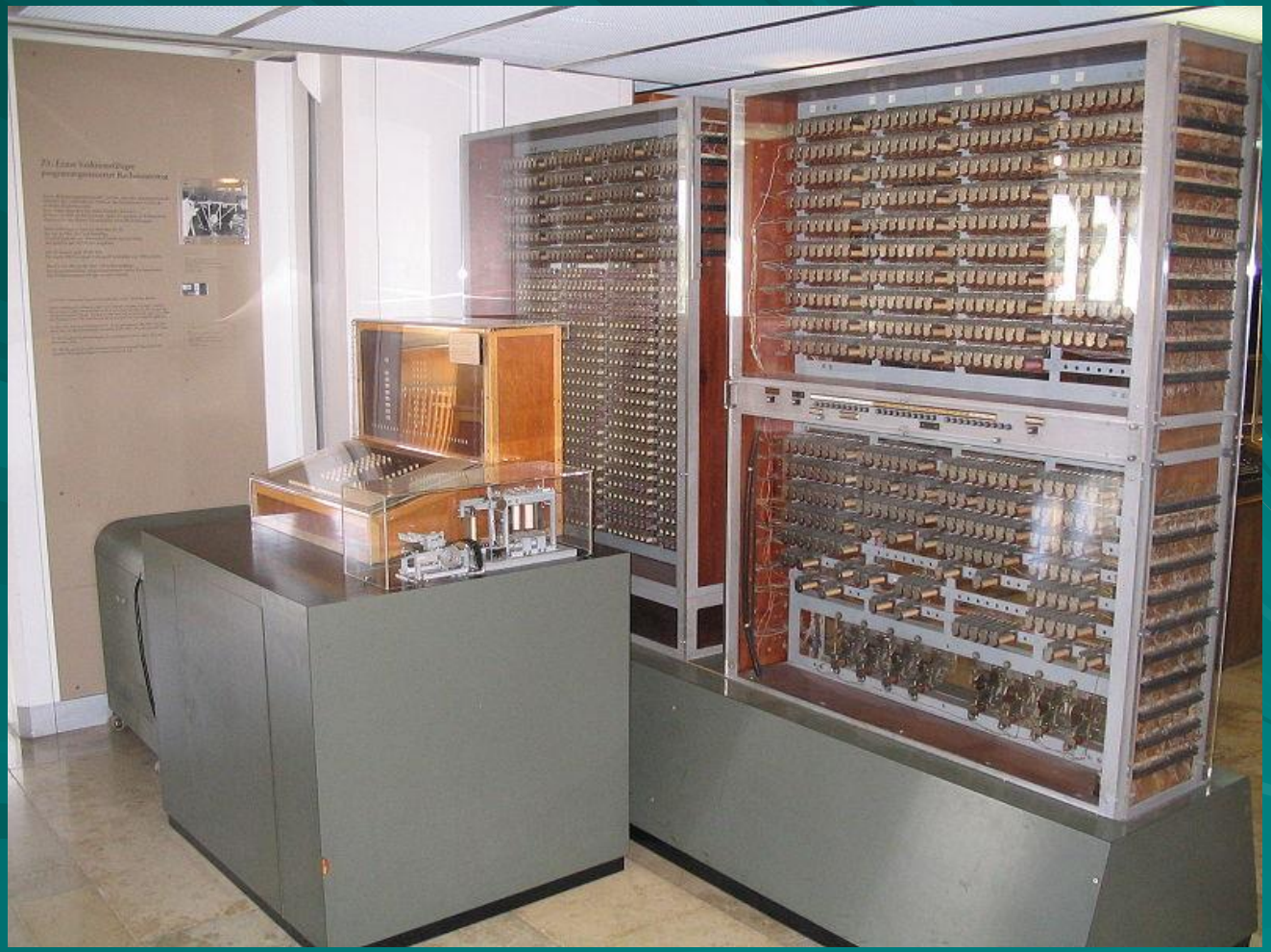
$F = \overline{A + B}$

$= \overline{A} \cdot \overline{B}$

Relay Computers
Conrad Zuse (1910-1995)

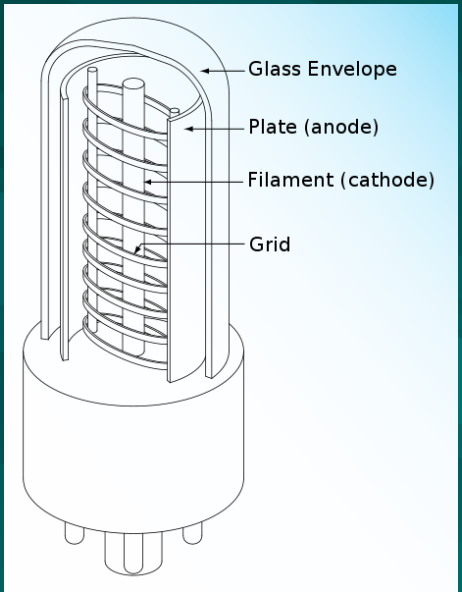


Z1 (1938)

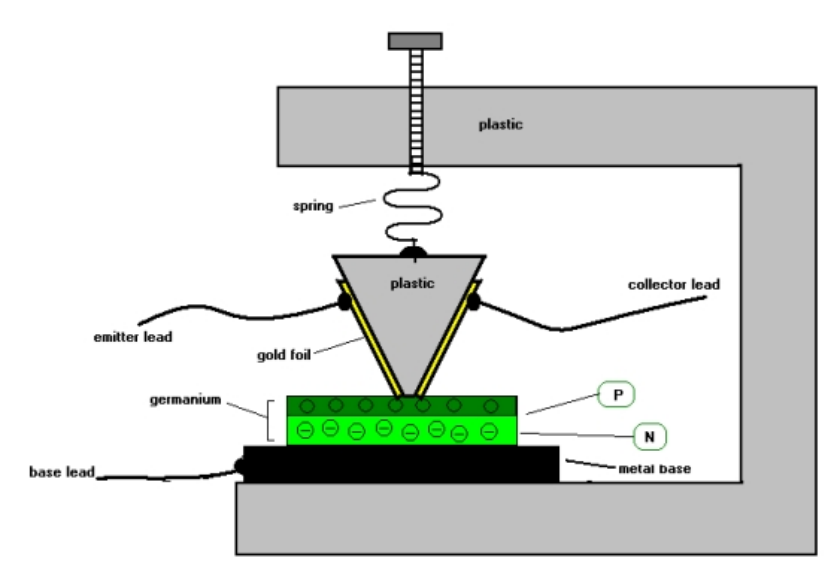


Z3 (1941)

Electronic Switching Devices



Electron Tube
Fleming, 1904
de Forest, 1906

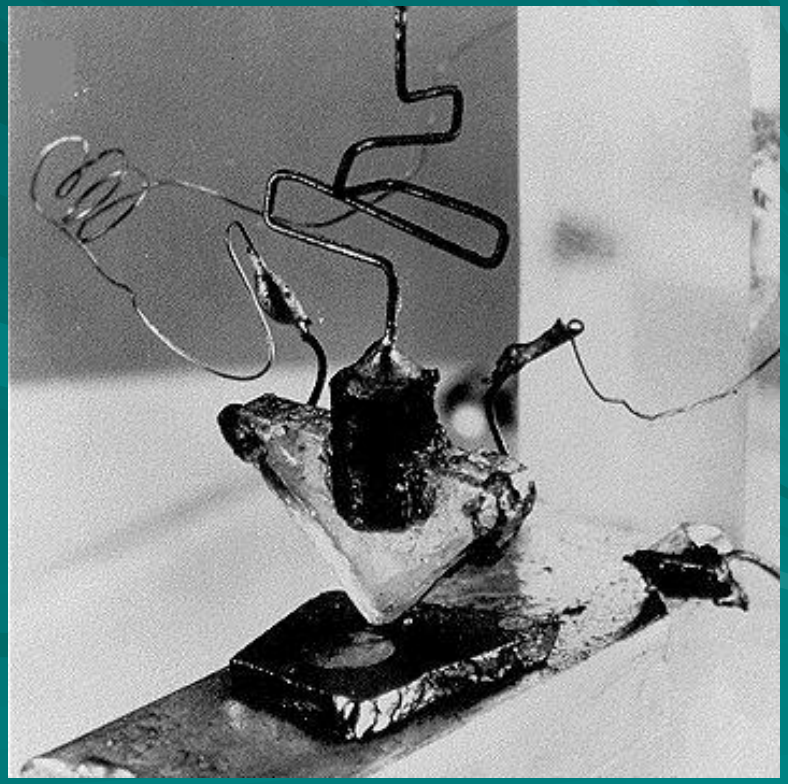


Point Contact Transistor
Bardeen, Brattain, Shockley, 1948

Transistor, 1948

The thinker, the tinkerer, the visionary and the transistor
John Bardeen, Walter Brattain, William Shockley
Nobel Prize, 1956



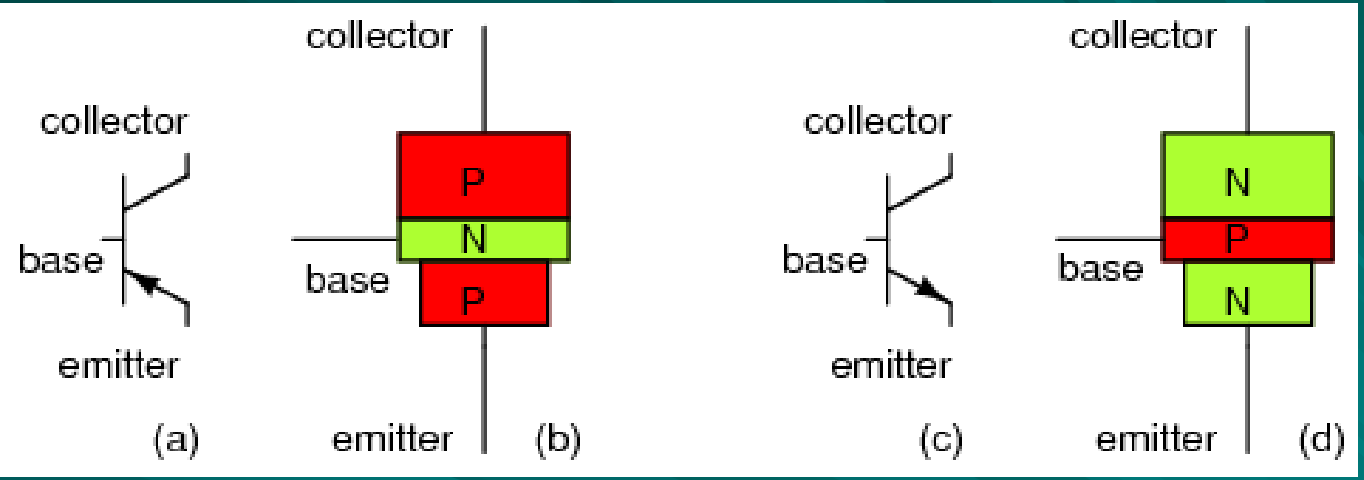


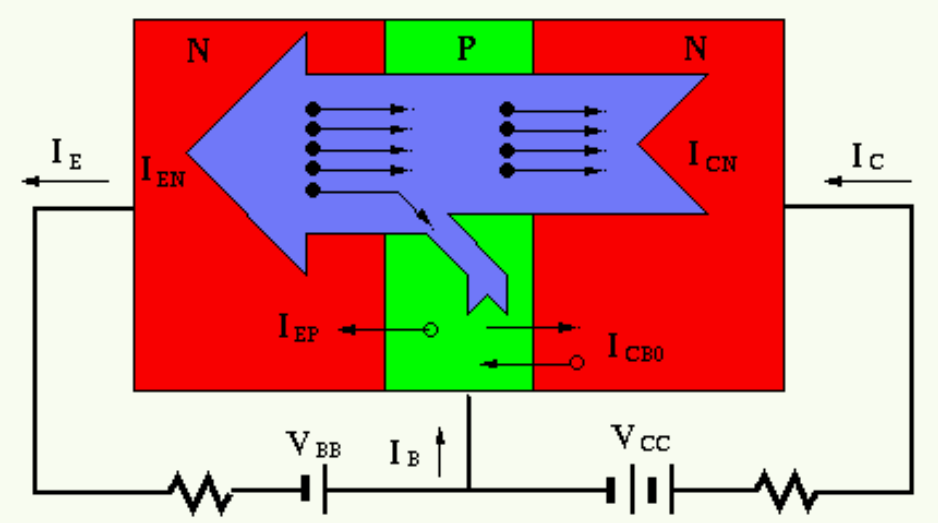
Bell Laboratories, Murray Hill, New Jersey




microelectronics group
A replica of the first transistor, invented at Bell Labs. December 23, 1947
Lucent Technologies
Bell Labs Innovations
50 Years and Counting...

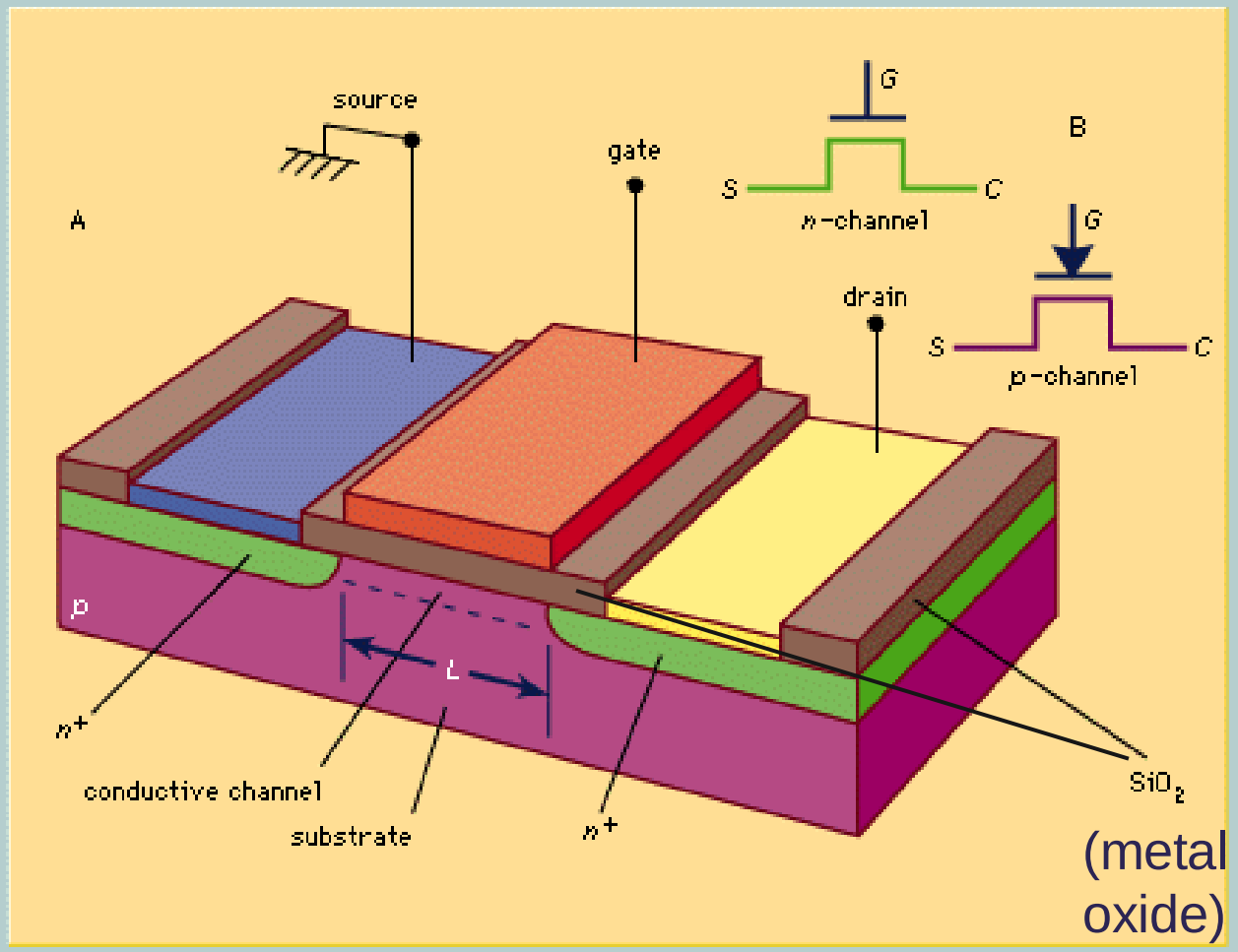
Bipolar Junction Transistor (BJT)





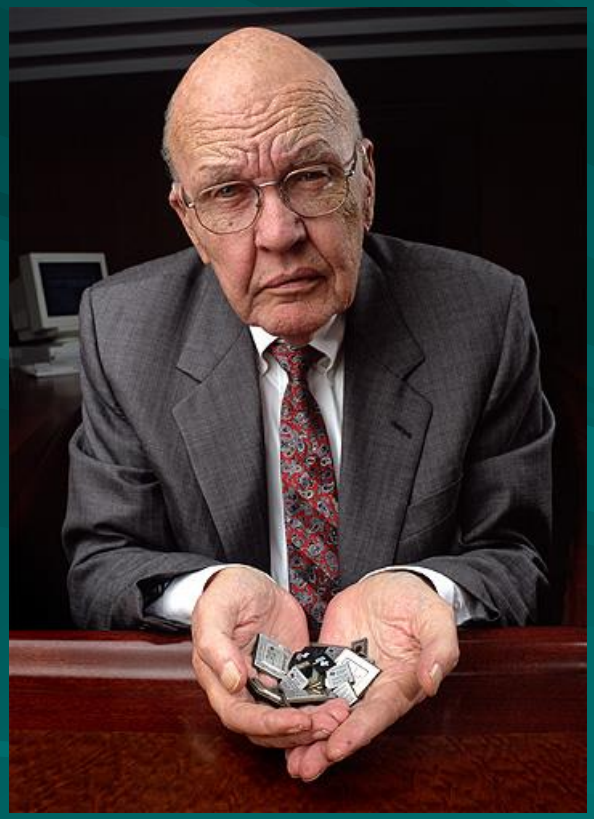
Field Effect Transistor (FET)

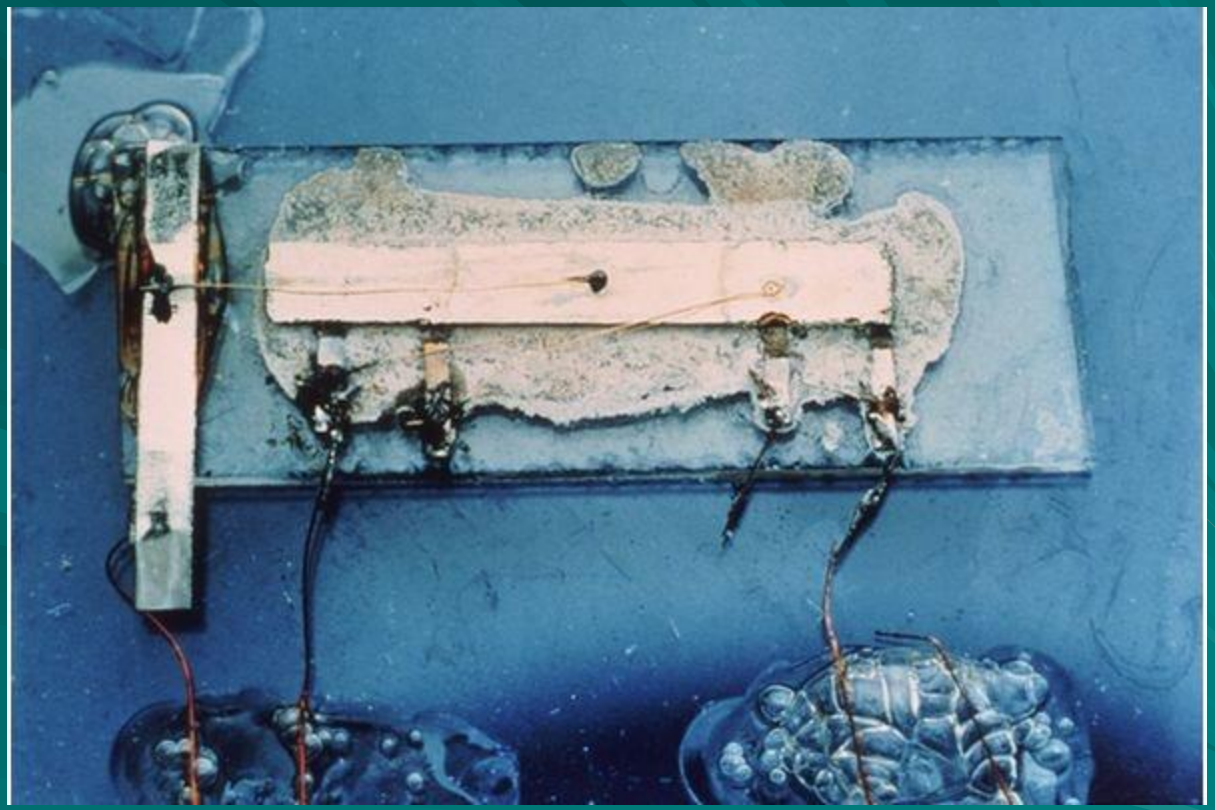
a.k.a.
metal oxide semiconductor (MOS) FET.



©1995 Encyclopaedia Britannica, Inc.

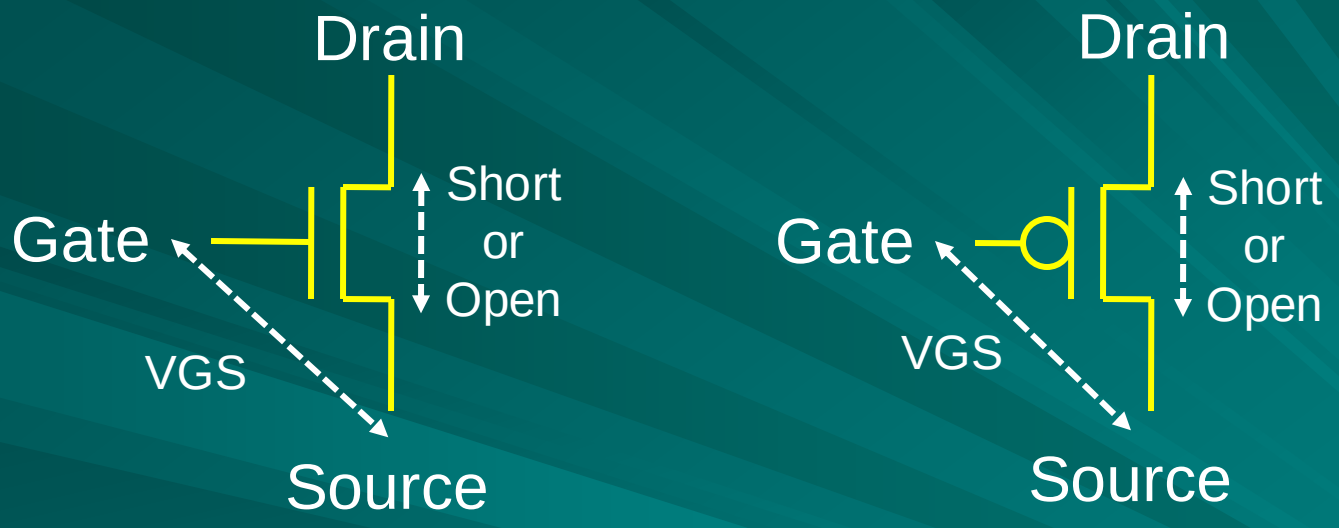
Integrated Circuit (1958)





Jack Kilby (1923-2005), Nobel Prize, 2000

MOSFET (Metal Oxide Semiconductor Field Effect Transistor)



NMOSFET

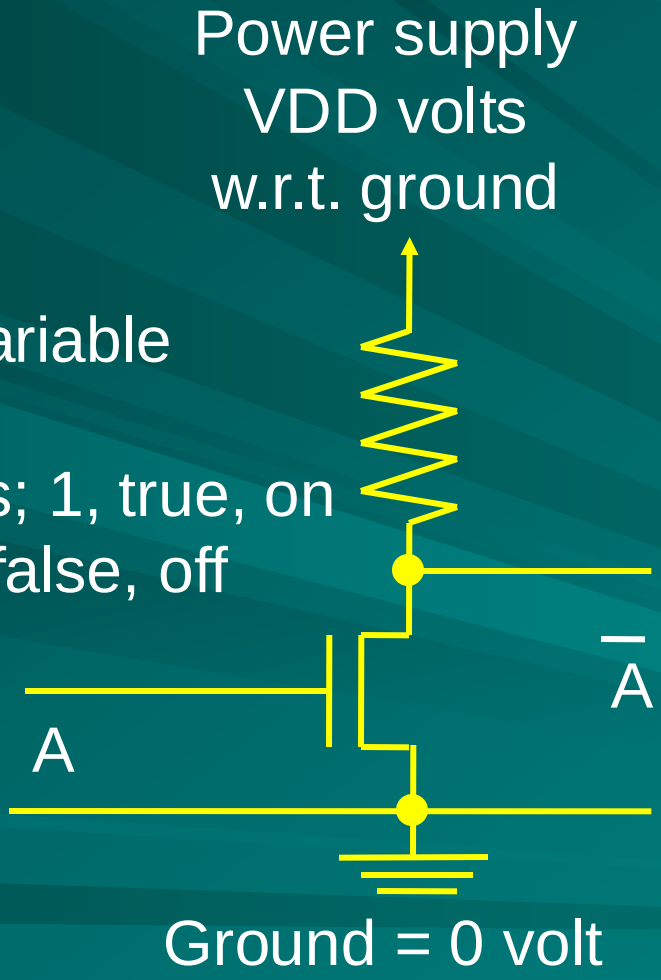
VGS = 0, open
VGS = high, short

PMOSFET

VGS = 0, short
VGS = high, open

Reference:
R. C. Jaeger and T. N. Blalock, *Microelectronic Circuit Design, Third Edition*, McGraw Hill.

NMOSFET Gate (Early Design)



A: Boolean variable

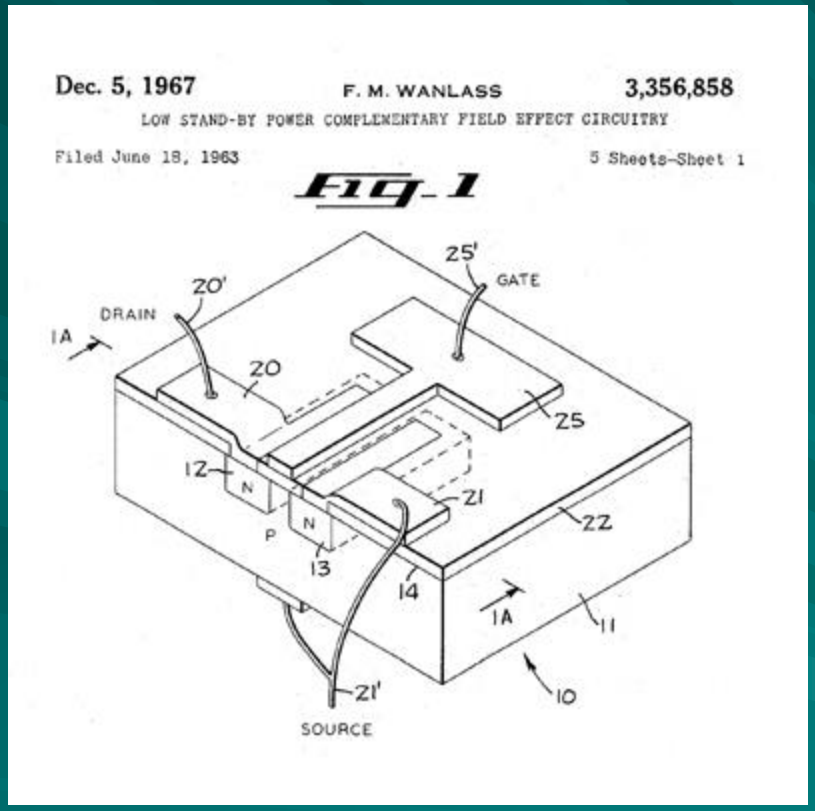
A = VDD volts; 1, true, on
A = 0 volt; 0, false, off

Problem: When A = 1, current leakage causes power dissipation.

Solution: Complementary MOS design proposed by

F. M. Wanlass and C.-T. Sah, "Nanowatt Logic Using Field-Effect Metal-Oxide Semiconductor Triodes," *International Solid State Circuits Conference Digest of Technical Papers*, Feb 20, 1963, pp. 32-33.

CMOS Circuit



Wanlass, F. M. "Low Stand-By Power Complementary Field Effect Circuitry." *U. S. Patent 3,356,858* (Filed June 18, 1963. Issued December 5, 1967).

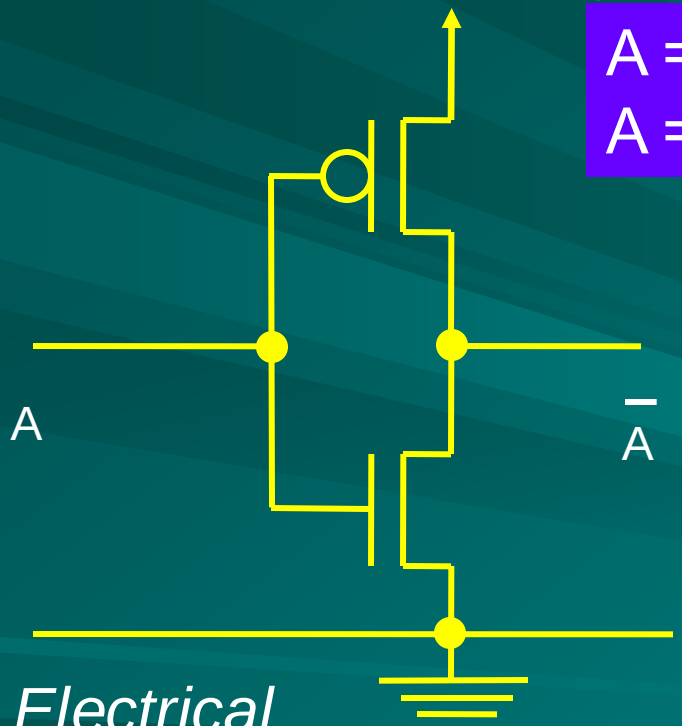
CMOS Logic Gate: Inverter

Power supply

VDD = 1 volt; voltage depends on technology.

A = VDD = 1 volt is state “1”

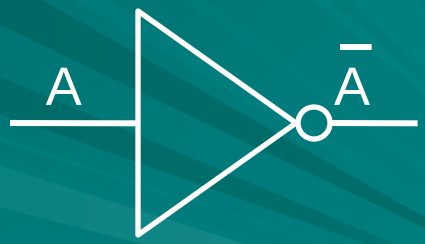
A = GND = 0 volt is state “0”



Electrical Circuit

Truth Table	
A	$\overline{A}$
0	1
1	0

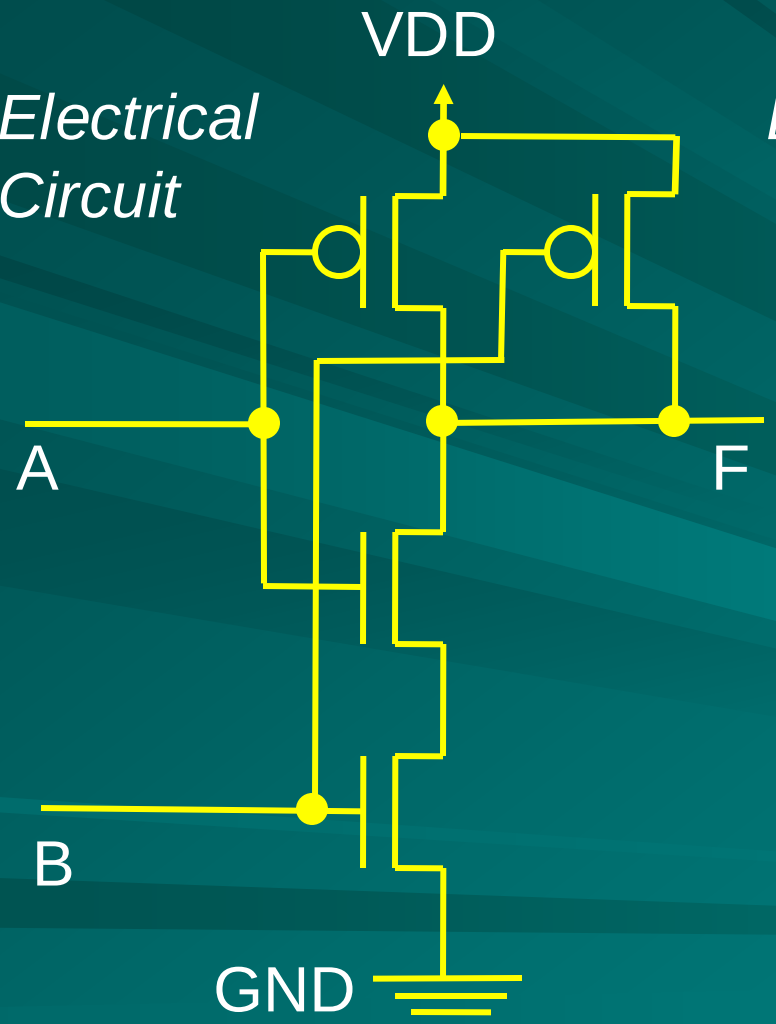
Boolean Function



Symbol

CMOS Logic Gate: NAND

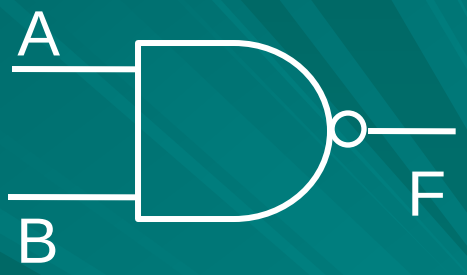
Electrical Circuit



Boolean Function

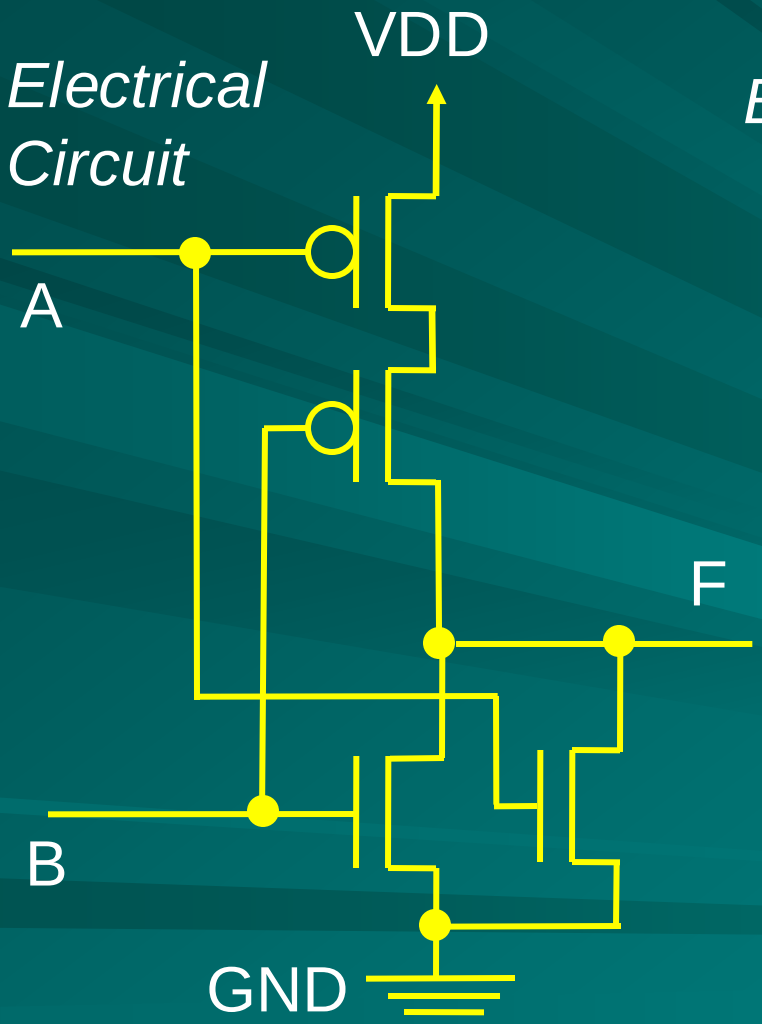
Truth Table		
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Symbol



CMOS Logic Gate: NOR

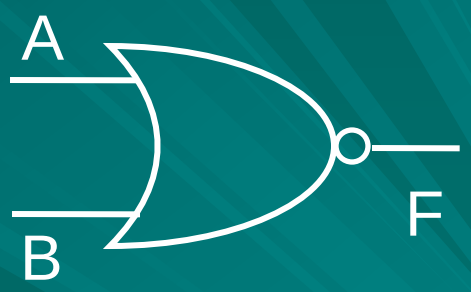
Electrical Circuit



Boolean Function

Truth Table		
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

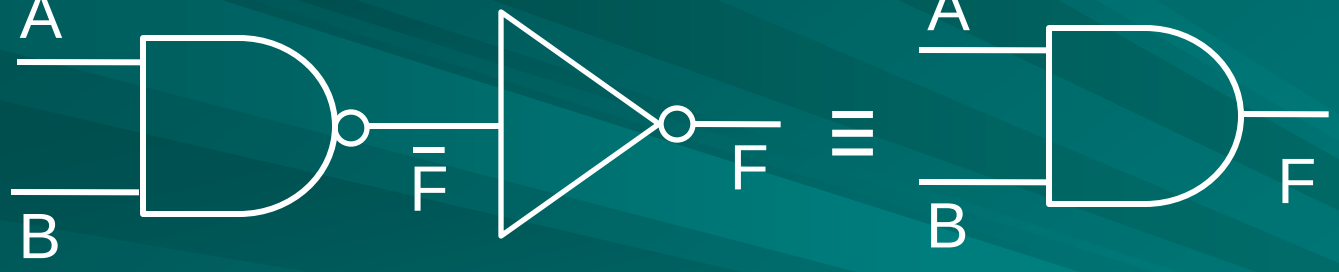
Symbol



CMOS Logic Gate: AND

Boolean Function

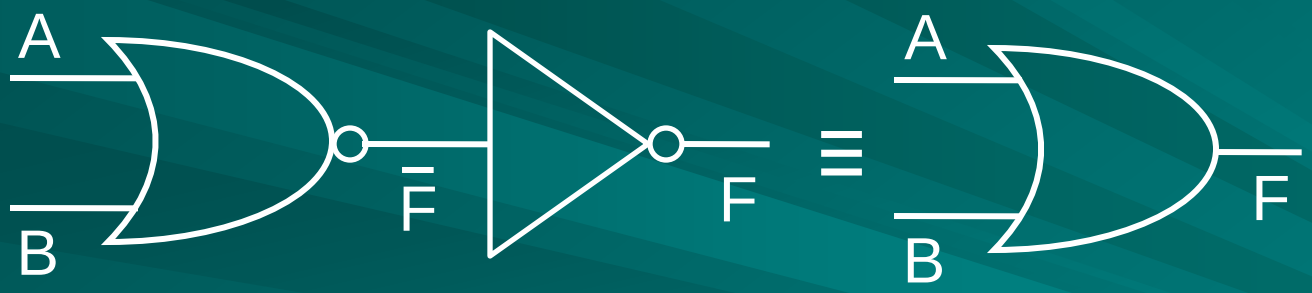
Symbol



Truth Table		
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

CMOS Logic Gate: OR

Symbol



Boolean Function

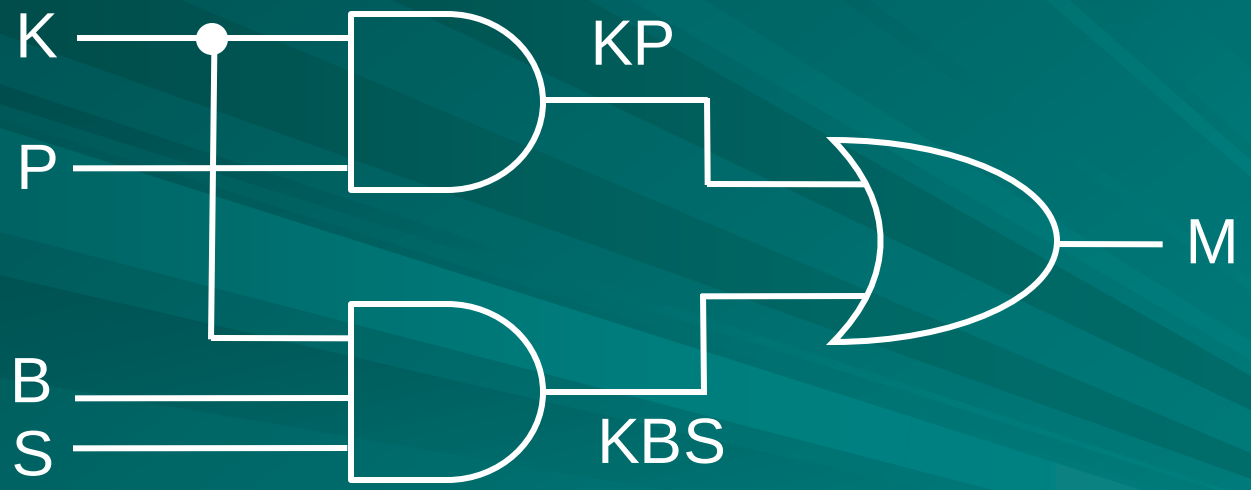
Truth Table		
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

CMOS Gates

Logic function	Number of transistors	
	1 or 2 inputs	N inputs
NOT	2	-
AND	6	2N + 2
OR	6	2N + 2
NAND	4	2N
NOR	4	2N

Optimized Ignition Logic

$$M = K(P + BS)$$
$$= KP + KBS$$



3 gates, 20 transistors. *Can we reduce transistors?*

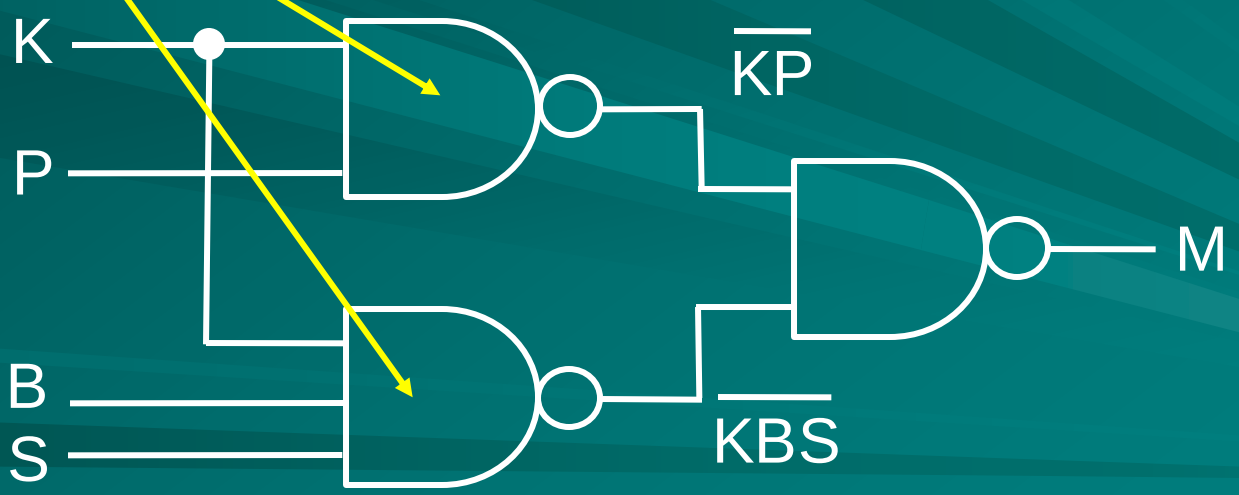
Further Optimization

$M = K(P + BS) = KP + KBS$ (Distr. law)

$= \overline{\overline{KP + KBS}}$ (Theorem 3, involution)

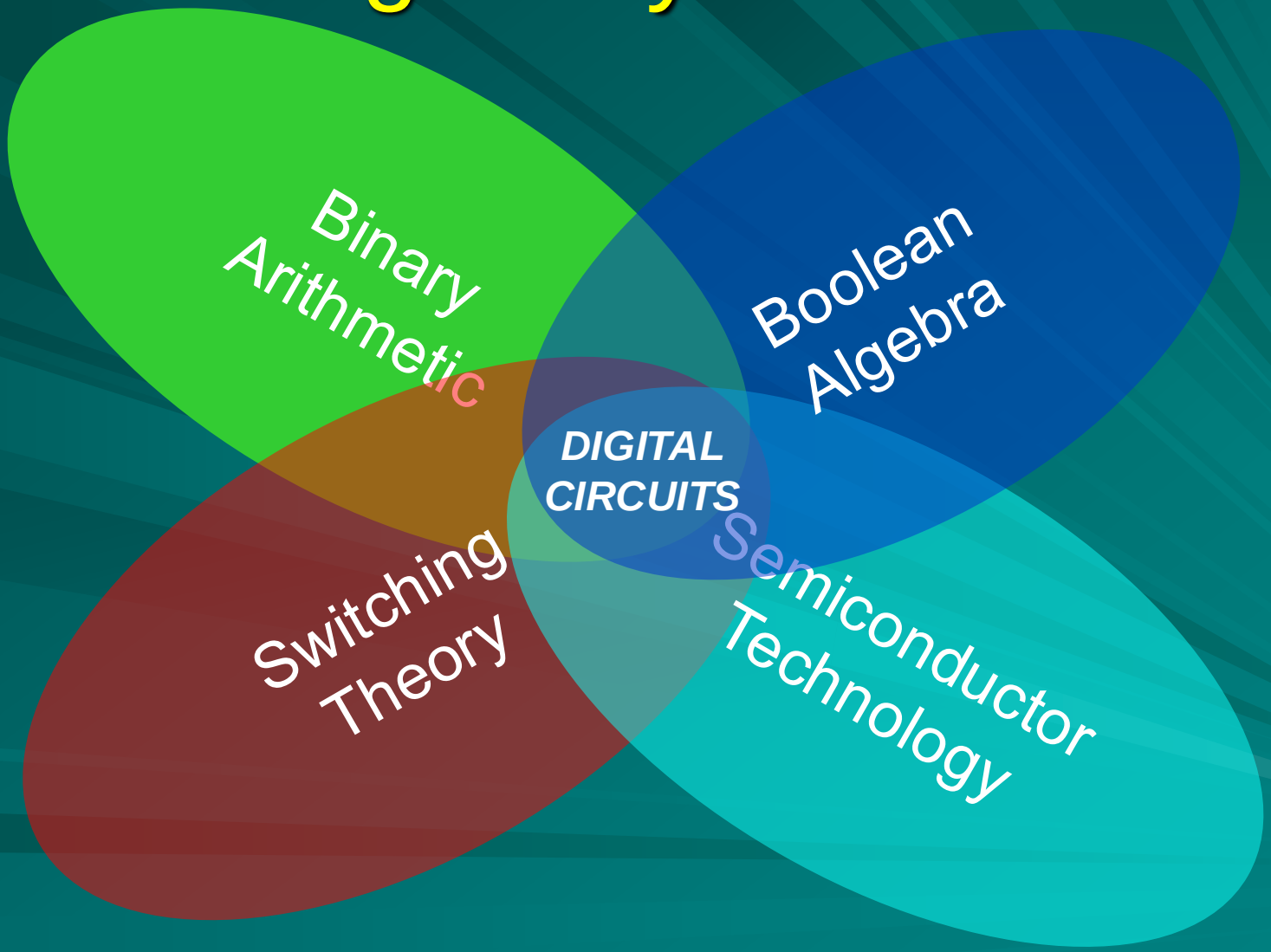
$= \overline{\overline{KP} \cdot \overline{KBS}}$ (De Morgan's theorem)

NAND gates
4+6 transistors



3 gates, 14 transistors.

Digital Systems



Digital Logic Design

- Representation of switching function:
  - Truth table
  - Canonical forms
  - Karnaugh map
- Logic minimization: Minimize number of literals.
- Technology mapping: Implement logic function using predesigned gates or building blocks from a technology library.

Truth Table

- Truth table is an exhaustive description of a switching function. Contains 2^n input combinations for n variables.
- Example: $f(A,B,C) = A\,B + \bar{A}\,C + A\,\bar{C}$

$2^n = 8$ rows

n Input variables, n = 3			Output
A	B	C	f(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

How Many Switching Functions?

- Output column of truth table has length 2^n for n input variables.
- It can be arranged in 2^{2^n} ways for n variables.
- Example: n = 1, single variable.

Input	Output functions			
A	F1(A)	F2(A)	F3(A)	F4(A)
0	0	0	1	1
1	0	1	0	1

Definitions

- Boolean variable: A variable denoted by a symbol; can assume a value 0 or 1.
- Literal: Symbol for a variable or its complement.
- Product or product term: A set of literals, ANDed together. Example, $a\ b\ \bar{c}$.
- Cube: Same as a product term.
- Sum: A set of literals, Ored together. Example, $a + b + \bar{c}$.

More Definitions

- SOP (sum of products): A Boolean function expressed as a sum of products.

  Example: $f(A,B,C) = A\,B + \overline{A}\,C + A\,\overline{C}$

- POS (product of sums): A Boolean function expressed as a product of sums.

  Example:

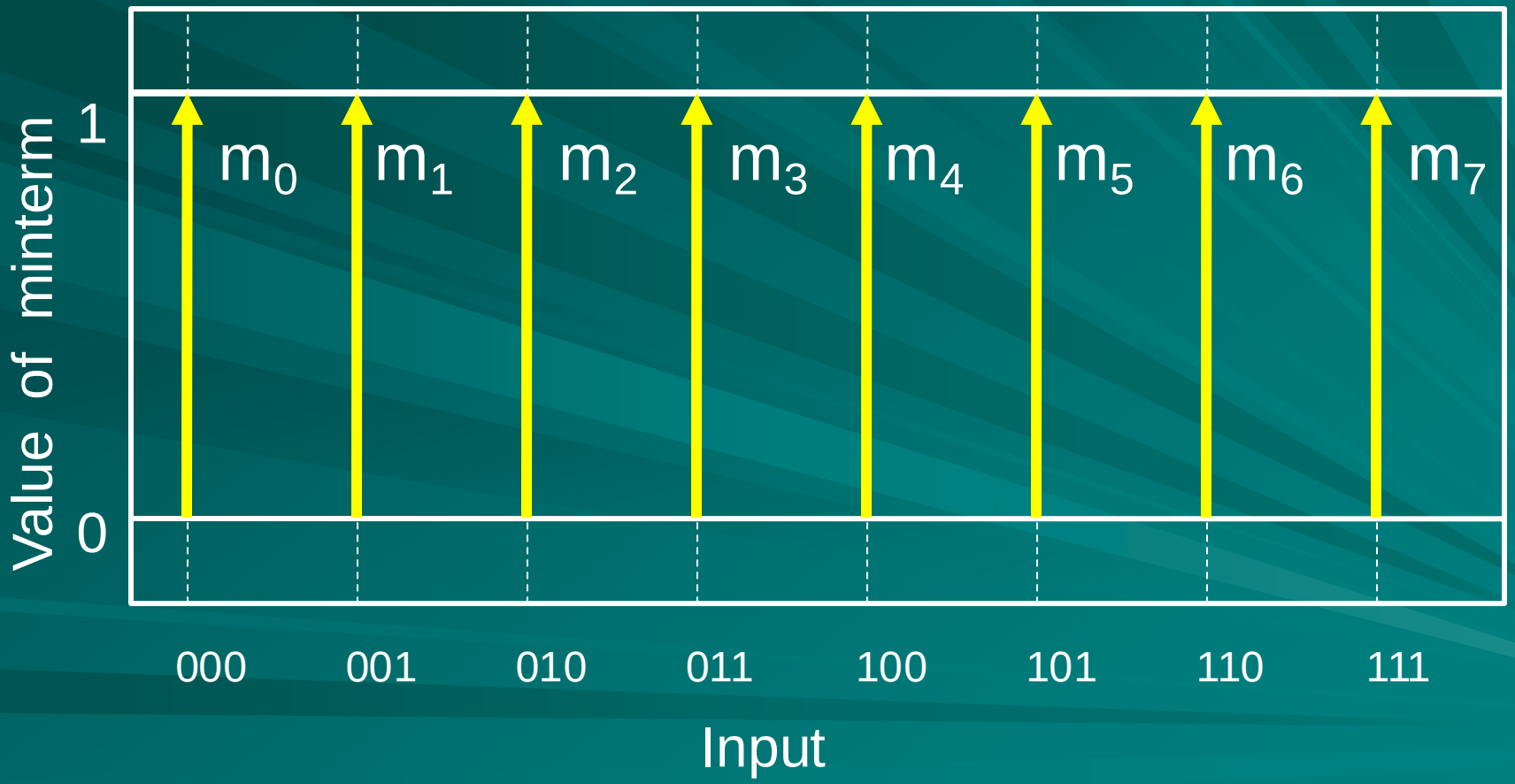
  $f(A,B,C) = (\overline{A} + \overline{B} + \overline{C})\,(\overline{A} + B + \overline{C})\,(A + \overline{B} + C)$

Minterm

- A product term in which each variable is present either in true or in complement form.
- For n variables, there are 2^n unique minterms.

Minterm	Product
m_0	$\bar{A}\ \bar{B}\ \bar{C}$
m_1	$\bar{A}\ \bar{B}\ C$
m_2	$\bar{A}\ B\ \bar{C}$
m_3	$\bar{A}\ B\ C$
m_4	$A\ \bar{B}\ \bar{C}$
m_5	$A\ \bar{B}\ C$
m_6	$A\ B\ \bar{C}$
m_7	$A\ B\ C$

Minterms are Canonical Functions



Canonical SOP Form
a.k.a. Disjunctive Normal Form (DNF)

- A Boolean function expressed as a sum of minterms.
- Example: $f(A,B,C) = A\,B + \overline{A}\,C + A\,\overline{C}$

$$= \overline{A}\;\overline{B}C + \overline{A}BC + A\;\overline{B}\;\overline{C} + AB\;\overline{C} + ABC$$

$$= m_1 + m_3 + m_4 + m_6 + m_7 = \sum m(1, 3, 4, 6, 7)$$

Truth table with row numbers

Row No.	A	B	C	f(A,B,C)
0	0	0	0	0
1	**0**	**0**	**1**	**1**
2	0	1	0	0
3	**0**	**1**	**1**	**1**
4	**1**	**0**	**0**	**1**
5	1	0	1	0
6	**1**	**1**	**0**	**1**
7	**1**	**1**	**1**	**1**

Maxterm

- A summation term in which each variable is present either in true or in complement form.
- For n variables, there are 2^n unique maxterms.

Maxterm	Sum
M_0	$A + B + C$
M_1	$A + B + \overline{C}$
M_2	$A + \overline{B} + C$
M_3	$A + \overline{B} + \overline{C}$
M_4	$\overline{A} + B + C$
M_5	$\overline{A} + B + \overline{C}$
M_6	$\overline{A} + \overline{B} + C$
M_7	$\overline{A} + \overline{B} + \overline{C}$

Canonical POS Form
a.k.a. Conjunctive Normal Form (CNF)

- A Boolean function expressed as a product of maxterms.
- Example: $f(A,B,C) = A\,B + \overline{A}\,C + A\,\overline{C}$

$$= (A + B + C)(A + \overline{B} + C)(\overline{A} + B + \overline{C})$$

$$= M_0\,M_2\,M_5 = \Pi\, M(0, 2, 5)$$

Truth table with row numbers

Row No.	A	B	C	f(A,B,C)
0	**0**	**0**	**0**	**0**
1	0	0	1	1
2	**0**	**1**	**0**	**0**
3	0	1	1	1
4	1	0	0	1
5	**1**	**0**	**1**	**0**
6	1	1	0	1
7	1	1	1	1

Canonical Forms are Unique

- A canonical form completely defines a Boolean function. That is, for every input the canonical form specifies the value of the function.
- To determine canonical form:
  - Construct truth table and sum minterms corresponding to 1 outputs, or multiply maxterms corresponding to 0 outputs.
  - Alternatively, use Shannon's expansion theorem (see Section 2.2.3, page 101).
- Two Boolean functions are identical if and only if their canonical forms are identical.

Why Generate Canonical Form?

- Example: Are the following Boolean Functions Identical?

$$F1 = \overline{A}\,B\,D + A\,B\,D + A\,C\,D$$

$$F2 = A\,C\,D + \overline{A}\,B\,D + B\,\overline{C}\,D$$

$$F3 = A\,\overline{B}\,C\,D + B\,D$$

- Generate canonical forms, e.g., minterms, and compare.

Algebraic Procedure

- Expand each term to contain all variables
- Remember
  - Postulate 6: Complement

$$a + \bar{a} = 1$$

$$a \cdot \bar{a} = 0$$

  - Postulate 2: Identity elements

    a + 0 = a, 0 is identity element for +

    a · 1 = a, 1 is identity element for dot (·)

$$F1 = \overline{A}BD + ABD + ACD$$

$$= \overline{A}BD(C + \overline{C}) + ABD(C + \overline{C}) + ACD(B + \overline{B})$$

$$= \overline{A}BCD + \overline{A}B\overline{C}D + ABCD + AB\overline{C}D + ABCD + A\overline{B}CD$$

$$= m_7 + m_5 + m_{15} + m_{13} + m_{11}$$

$$= \sum m(5, 7, 11, 13, 15)$$

Similarly,

$$F2 = \sum m(5, 7, 11, 13, 15)$$

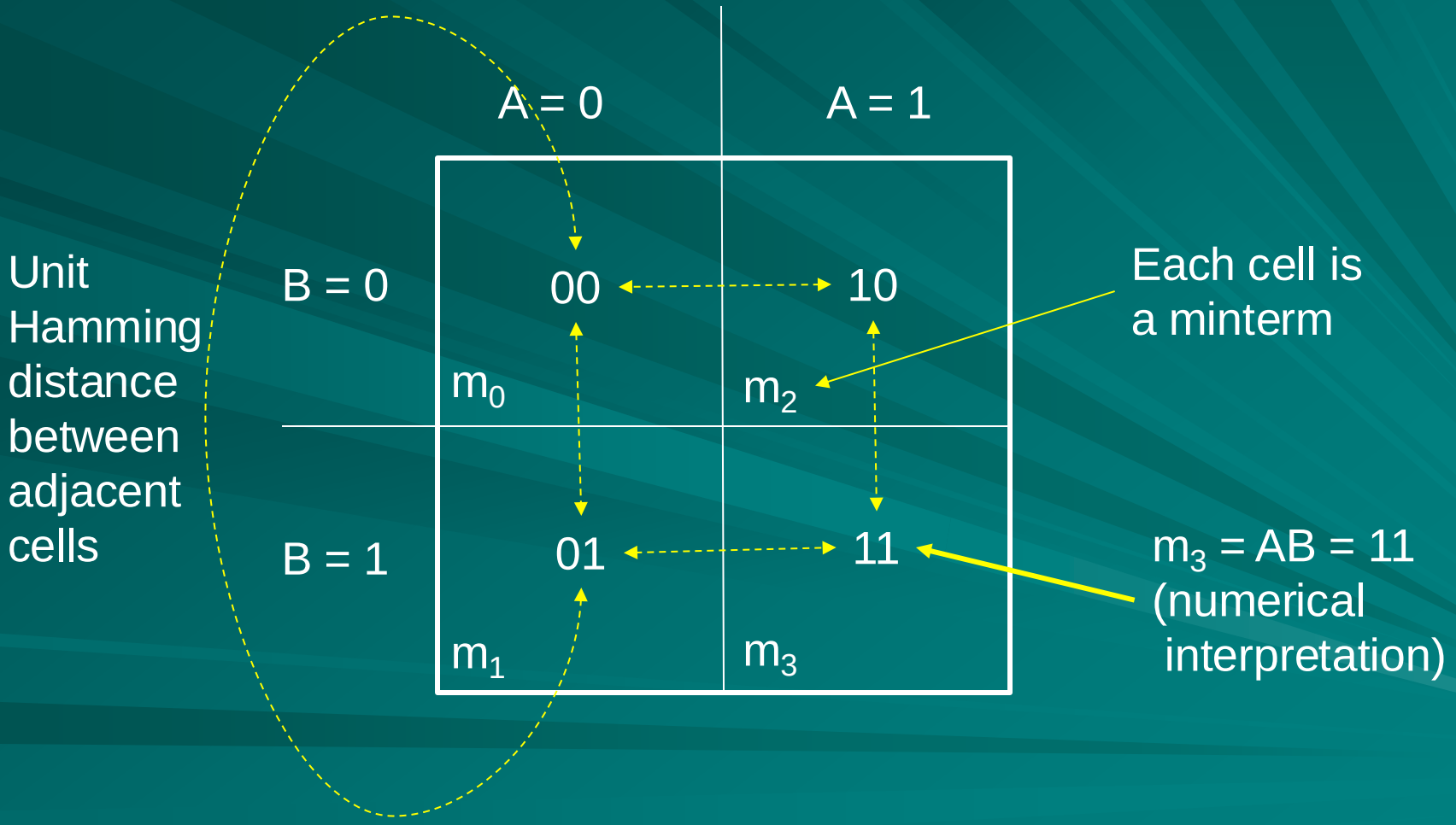
$$F3 = \sum m(5, 7, 11, 13, 15)$$

Hence, three functions are identical.

Karnaugh Map

- 1952: Edward M. Veitch invented a graphical procedure for digital circuit optimization.
- 1953: Maurice Karnaugh perfected the map procedure:
  - "The Map Method for Synthesis of Combinational Logic Circuits," *Trans. AIEE,* pt I, 72(9):593-599, November 1953.

Karnaugh Map: 2 Variables, A, B



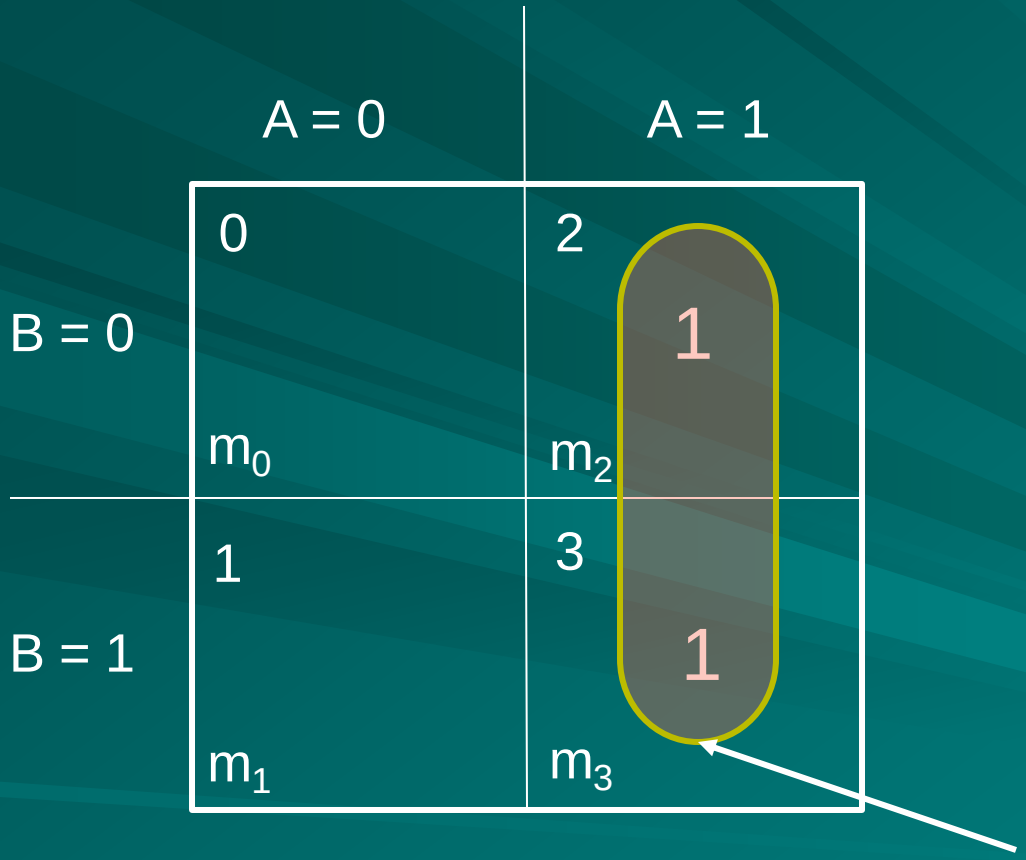
Representing a Function

	A = 0	A = 1
B = 0	0 m_0	2 1 m_2
B = 1	1 m_1	3 1 m_3

Place 1 in cells corresponding to minterms in canonical form. For example, see $F = A\,B + A\,\overline{B}$ represented on the left.

Truth Table		
A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

Grouping Adjacent Minterms



Adjacent cells differ in one variable, which is eliminated.

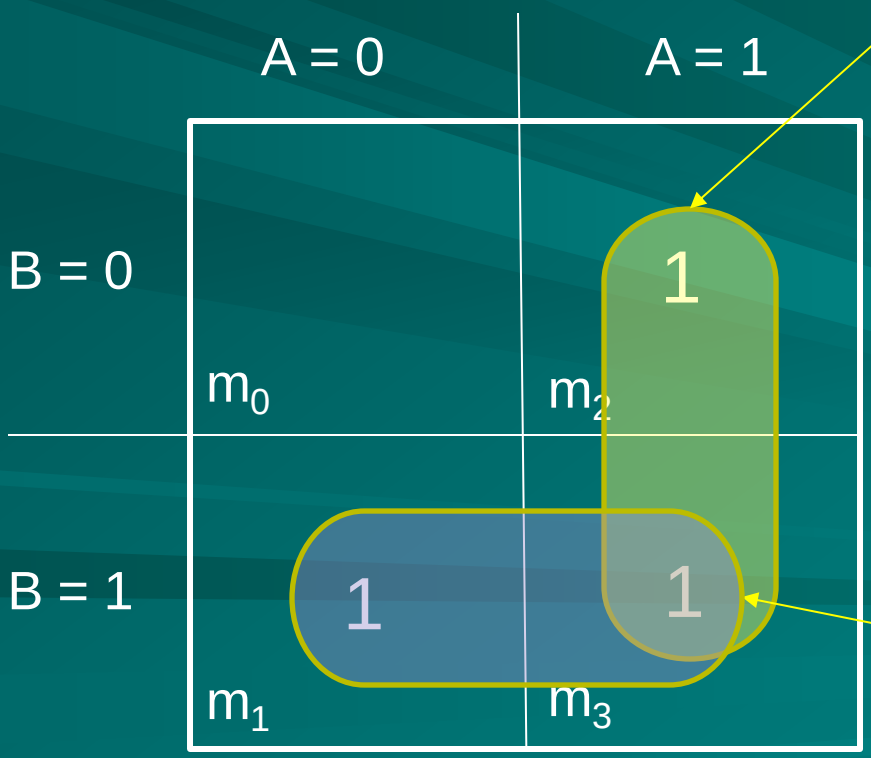
For example,

$F = A\,B + A\,\overline{B}$

$= A(B + \overline{B}) = A$

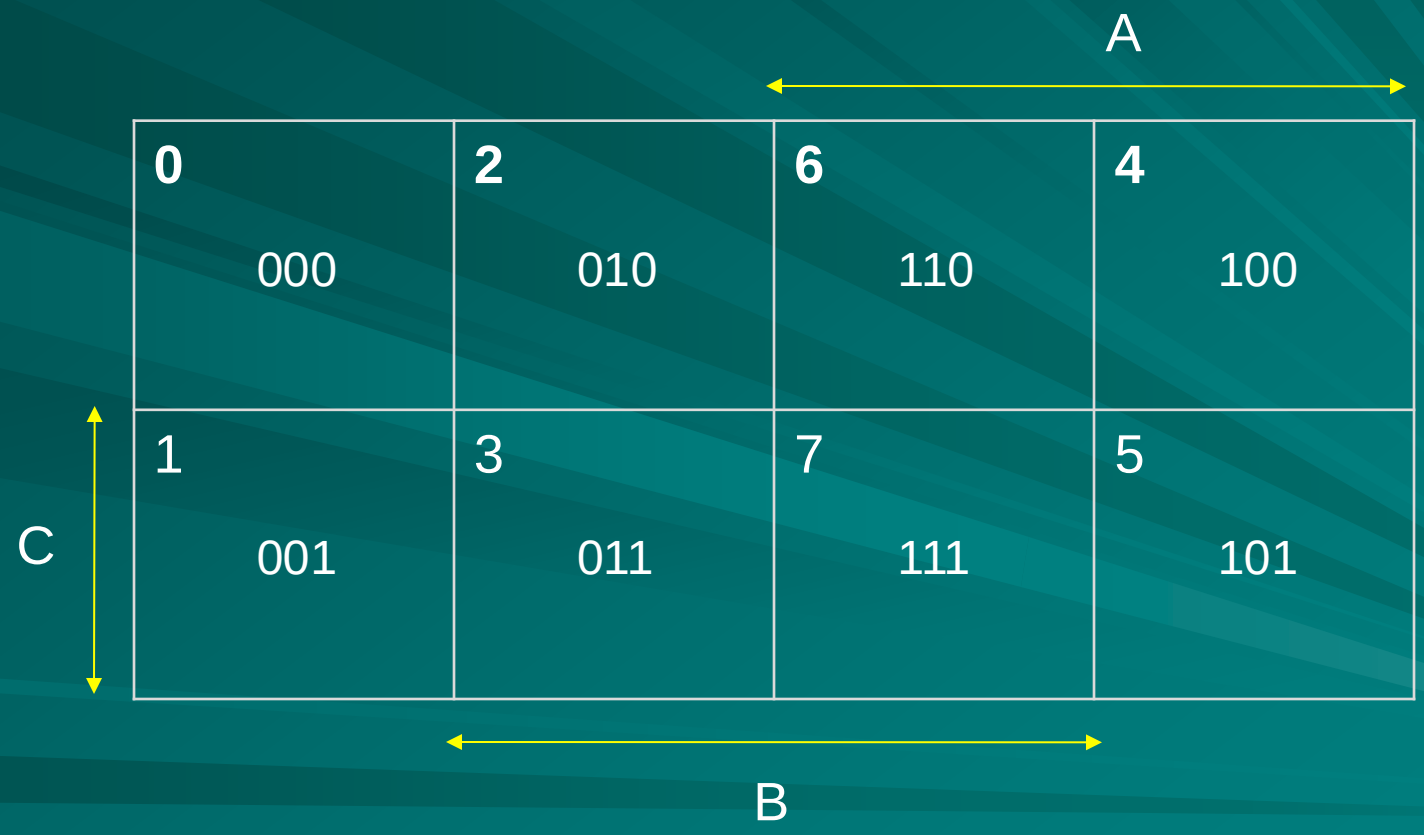
Karnaugh Map Minimization

- Canonical SOP form represented on map
- Example: F = A $\bar{B}$ + A B + $\bar{A}$ B
- Find minimal cover (fewest groups of largest sizes),
- F = A + B





Karnaugh Map: 3 Variables, A, B, C



Check unit Hamming distance between adjacent cells.

Synthesizing a Digital Function

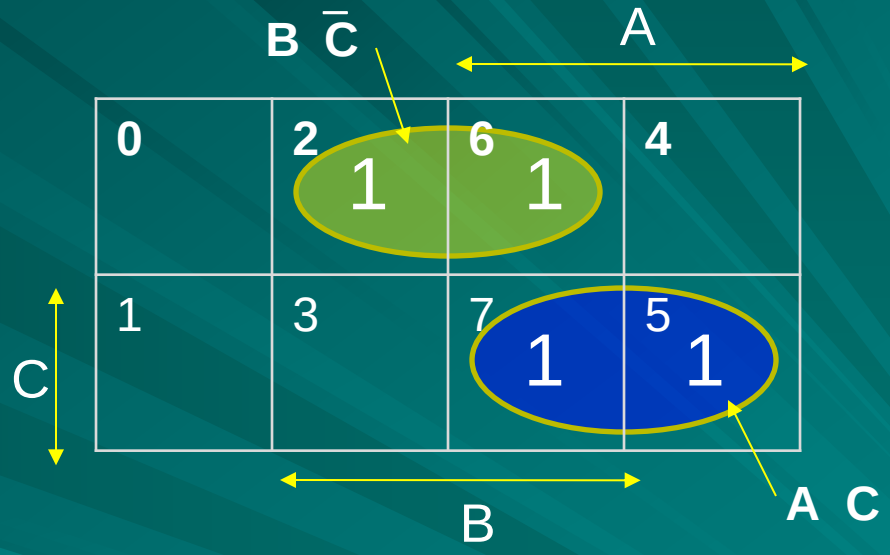
- Start with specification.
- Create a truth table from specification.
- Minimize (SOP with fewest literals):
  - Either write canonical SOP
  - Reduce using postulates and theorems
  - Or find largest cubes from Karnaugh map
- Minimized SOP gives a two-level AND-OR circuit.
- NAND or NOR circuit for CMOS technology can be found using de Morgan's theorem.

Example: Multiplexer

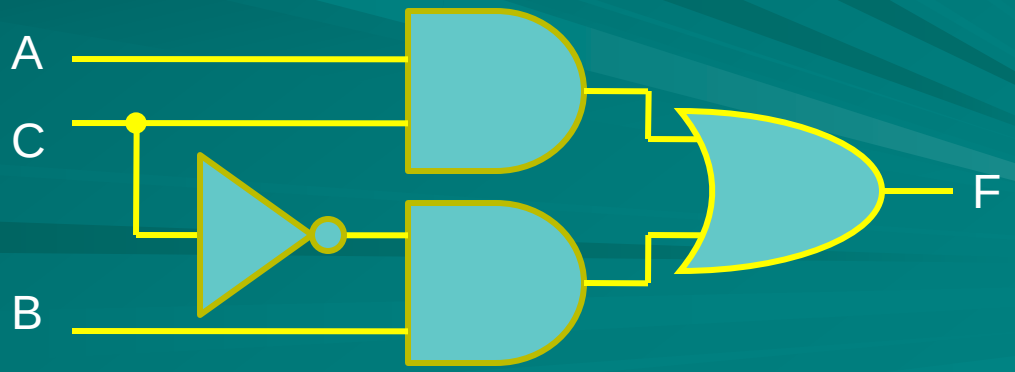
- Inputs: A, B, C
- Output: F
- Function:
  - F = A, when C = 1
  - F = B, when C = 0

3-Input Function: Multiplexer

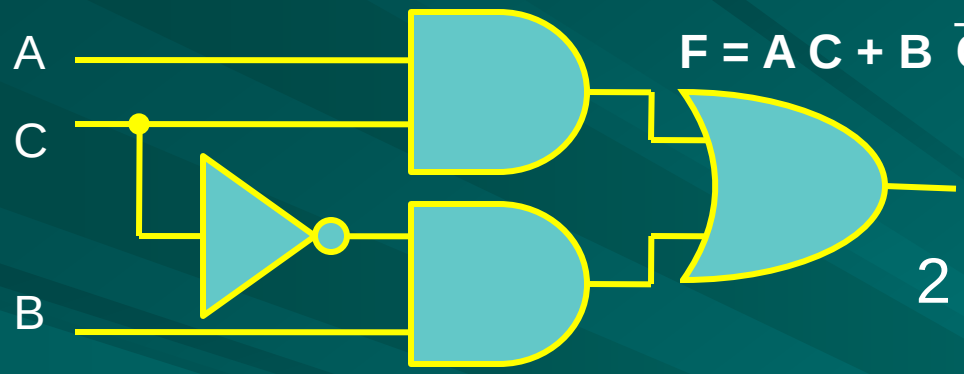
Truth Table				
row	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



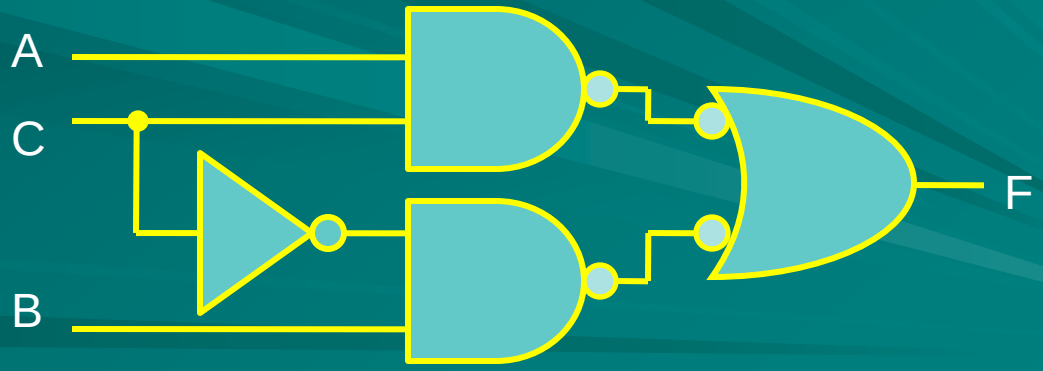
$F = A\,C + B\,\bar{C}$



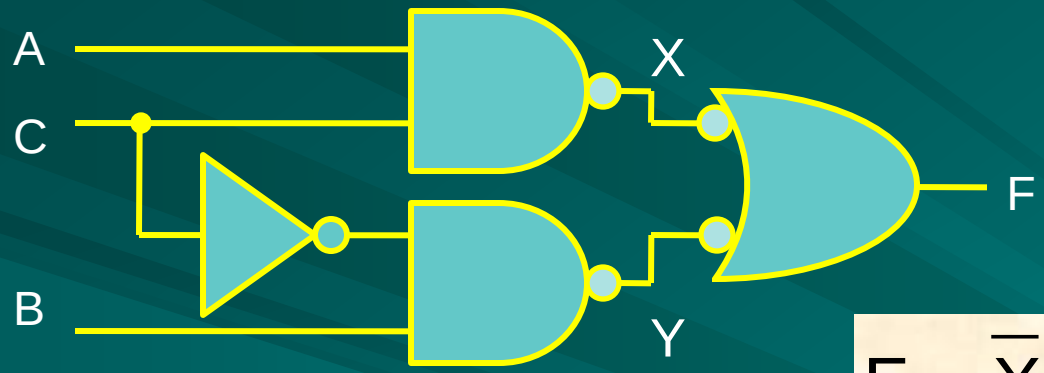
Technology Optimization



2 + 6 + 6 + 6 = 20 transistors

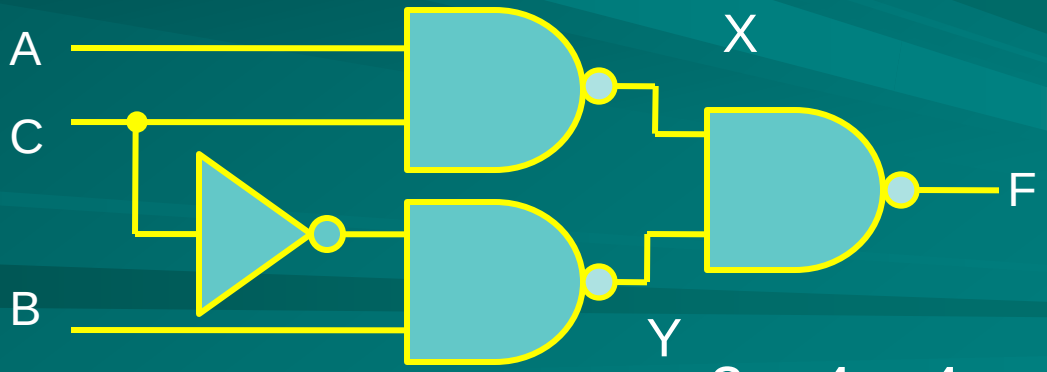


Optimized Multiplexer



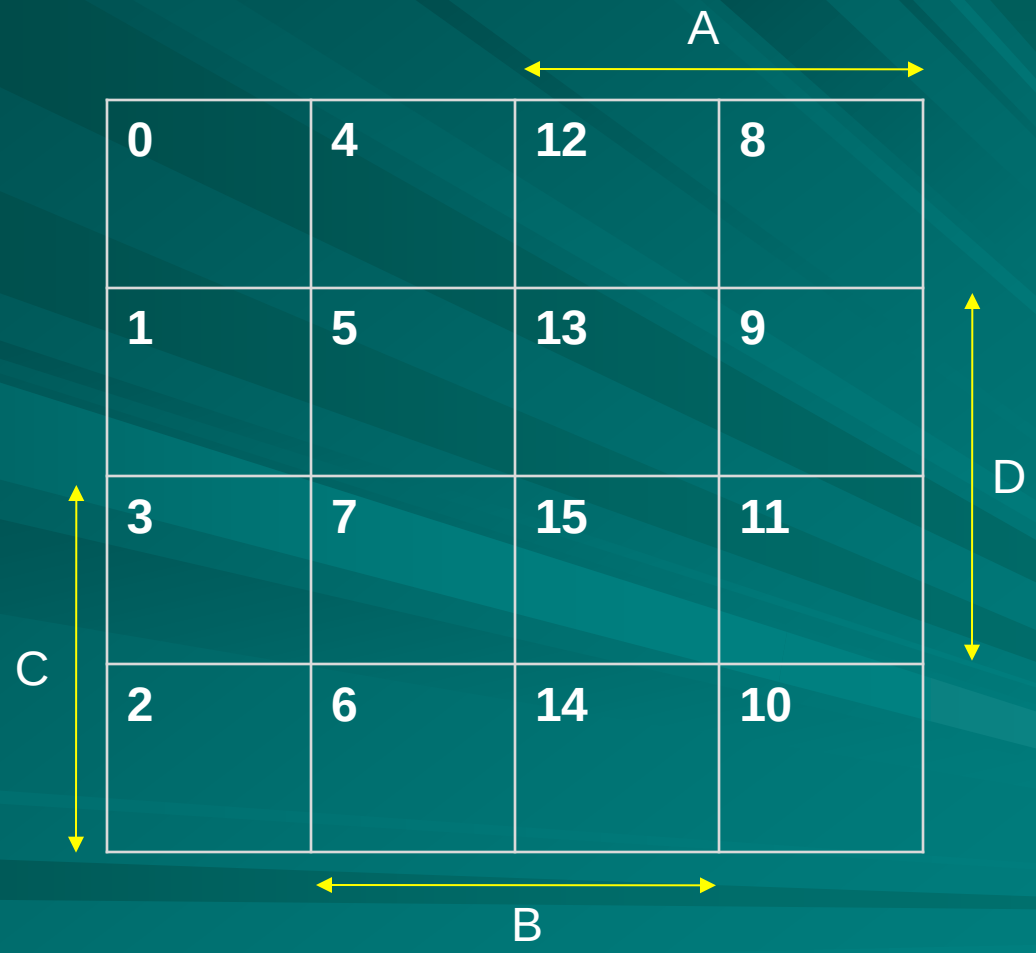
$$F = \overline{X} + \overline{Y} = \overline{X \cdot Y}$$

from de Morgan's theorem



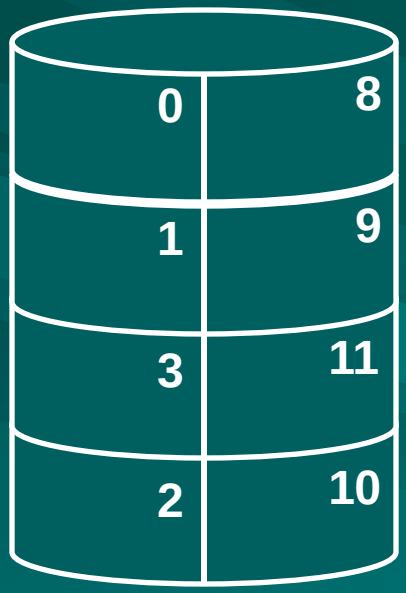
2 + 4 + 4 + 4 = 14 transistors

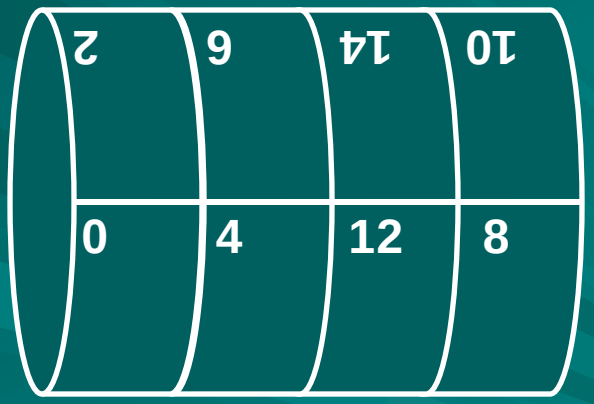
Karnaugh Map: 4 Variables, A, B, C, D



Check unit Hamming distance between adjacent cells.

Adjacency of Edge Cells





Reexamine Three Functions from Slide 48

- Example: Are the following Boolean Functions Identical?

$$F1 = \overline{A}\,B\,D + A\,B\,D + A\,C\,D$$

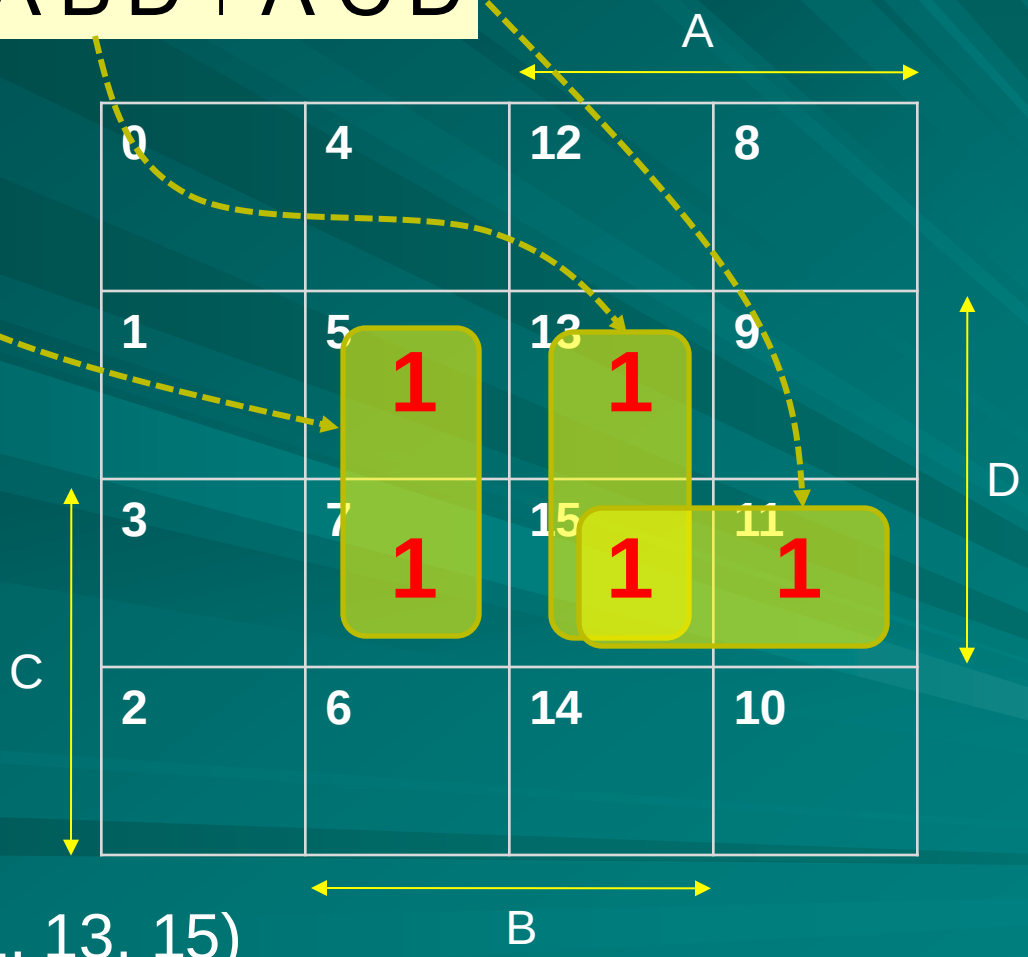
$$F2 = A\,C\,D + \overline{A}\,B\,D + B\,\overline{C}\,D$$

$$F3 = A\,\overline{B}\,C\,D + B\,D$$

- This time generate Karnaugh maps, and compare.

Karnaugh Map of F1

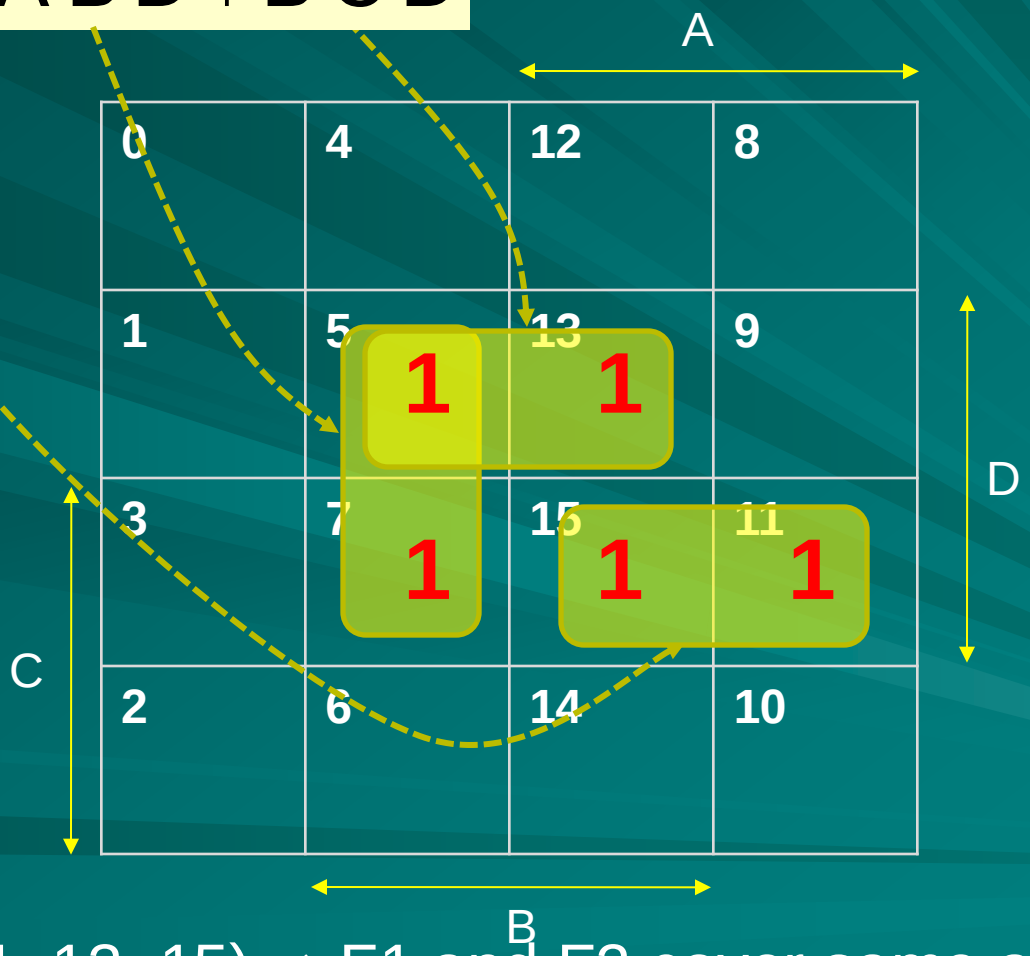
$$F1 = \overline{A}\,B\,D + A\,B\,D + A\,C\,D$$



F1 = Σm(5, 7, 11, 13, 15)

Karnaugh Map of F2

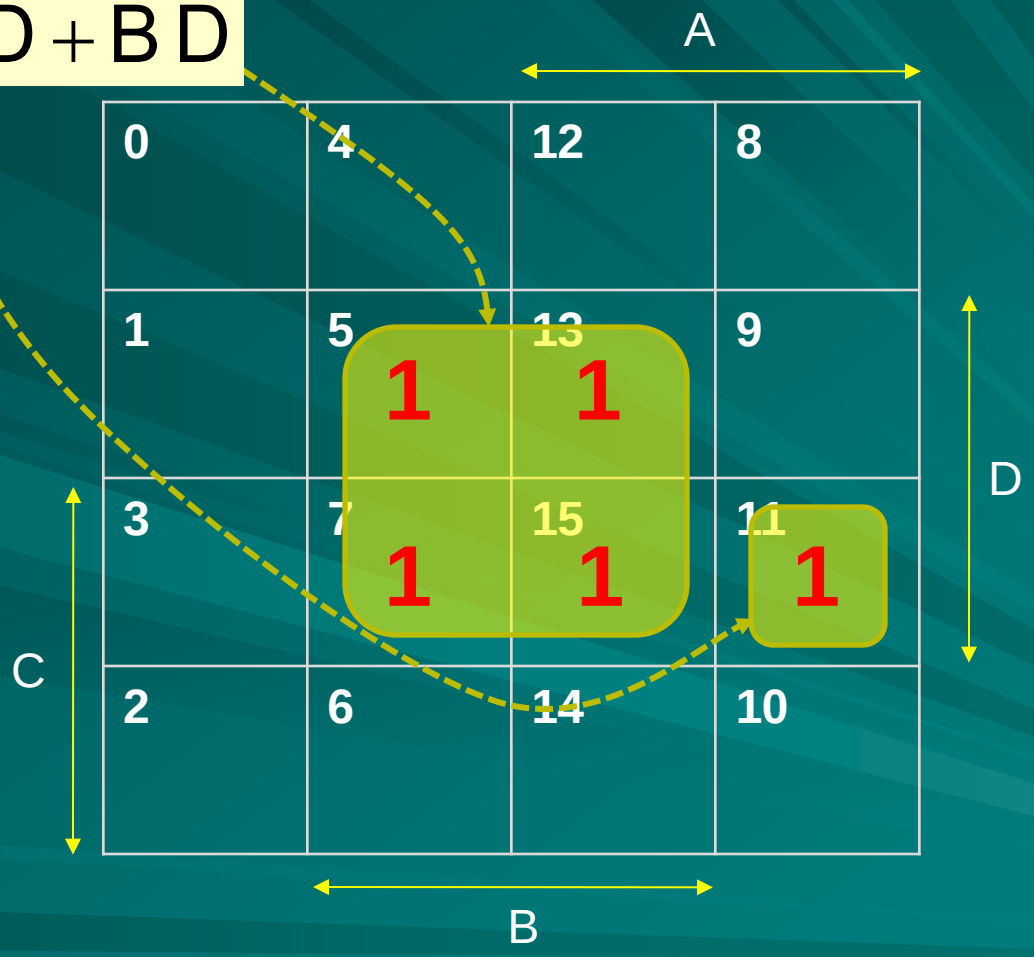
$$F2 = A\,C\,D + \overline{A}\,B\,D + B\,\overline{C}\,D$$



F2 = Σm(5, 7, 11, 13, 15) ⇒ F1 and F2 cover same area on map.

Karnaugh Map of F3

$$F3 = A\,\bar{B}\,C\,D + B\,D$$



F3 = Σm(5, 7, 11, 13, 15) ⇒ F1, F2 and F3 cover exactly the same area on map, hence, they are identical.

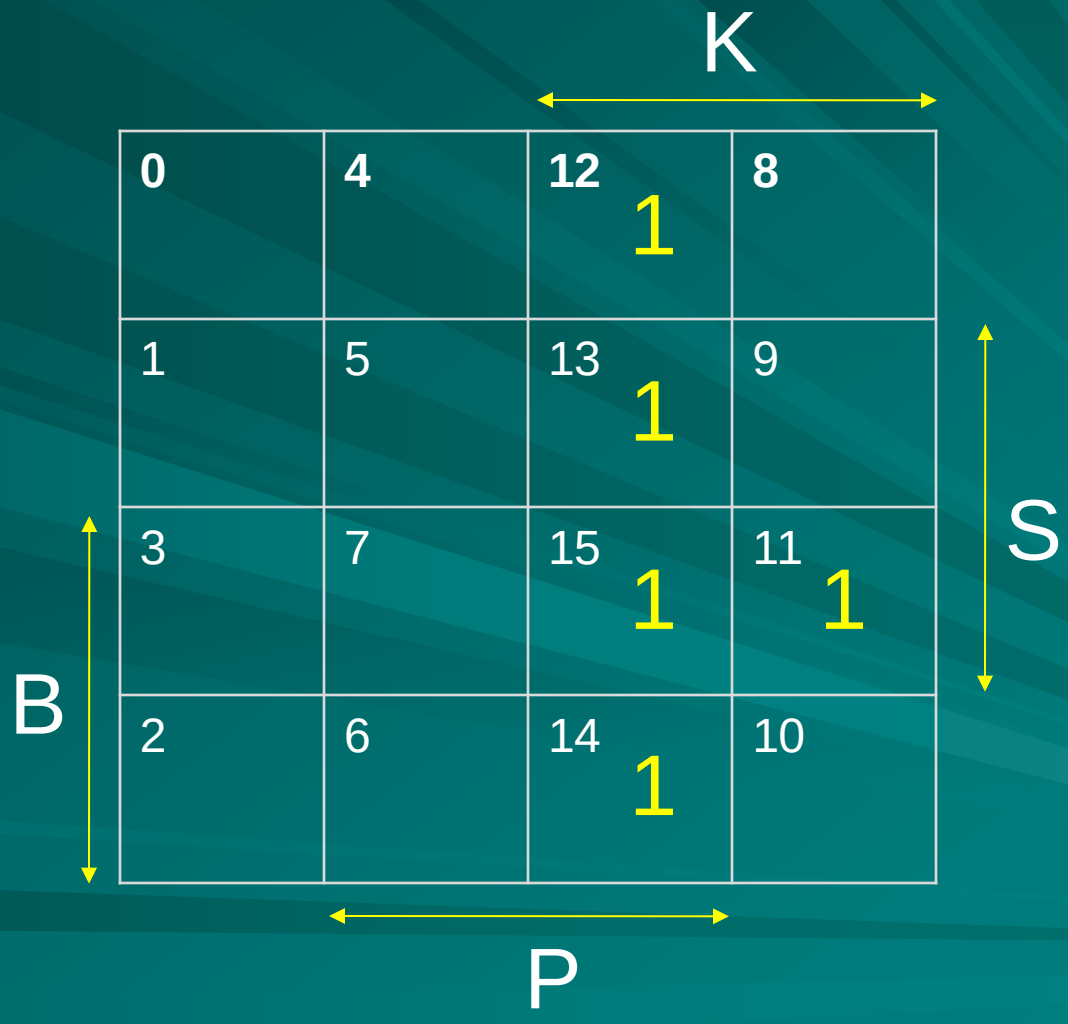
Ignition Function

M = K AND (P OR B) AND (S OR P)

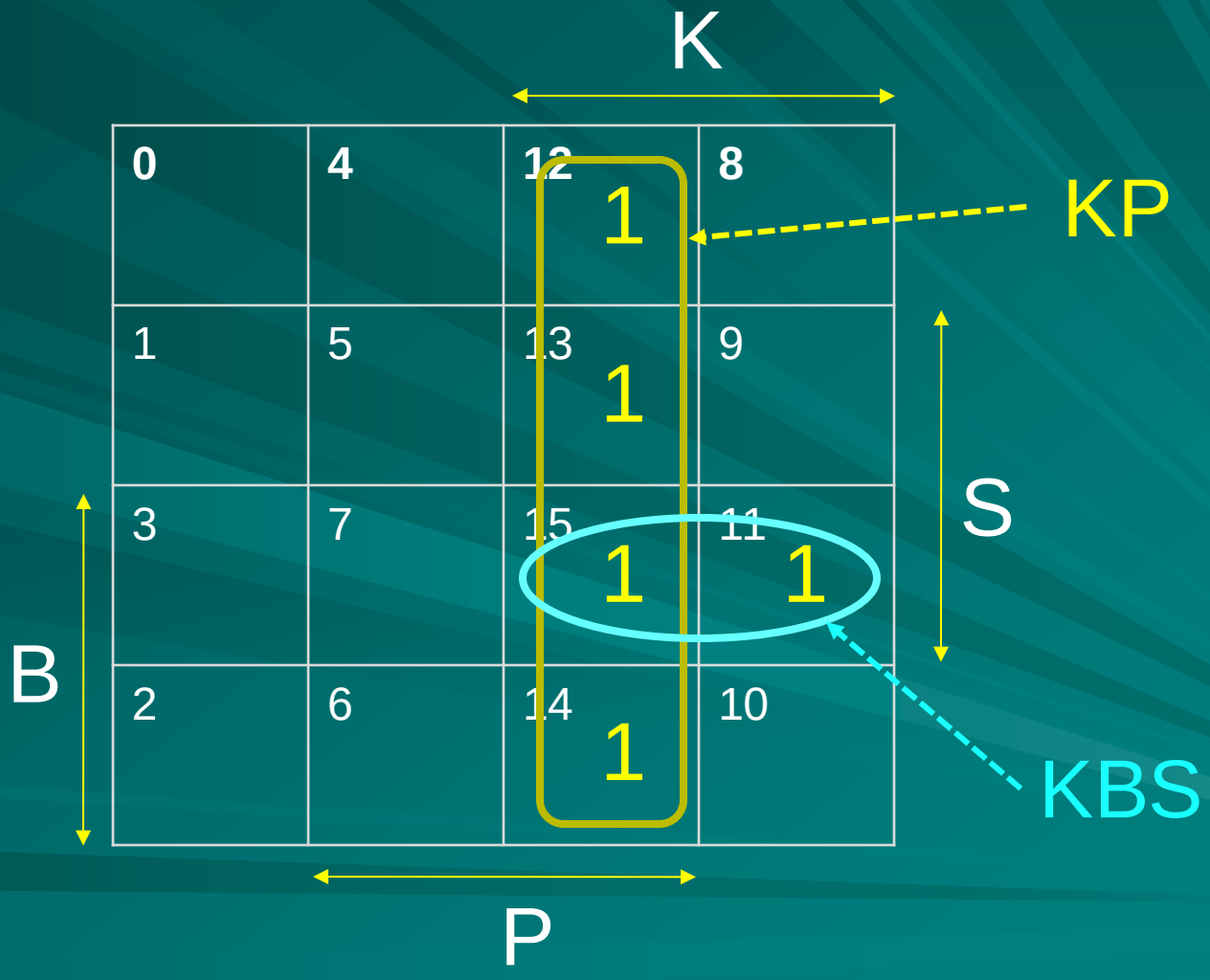
= K(P + B)(S + P)

Minterm	K	P	B	S	M
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Karnaugh Map: M(K, P, B, S)

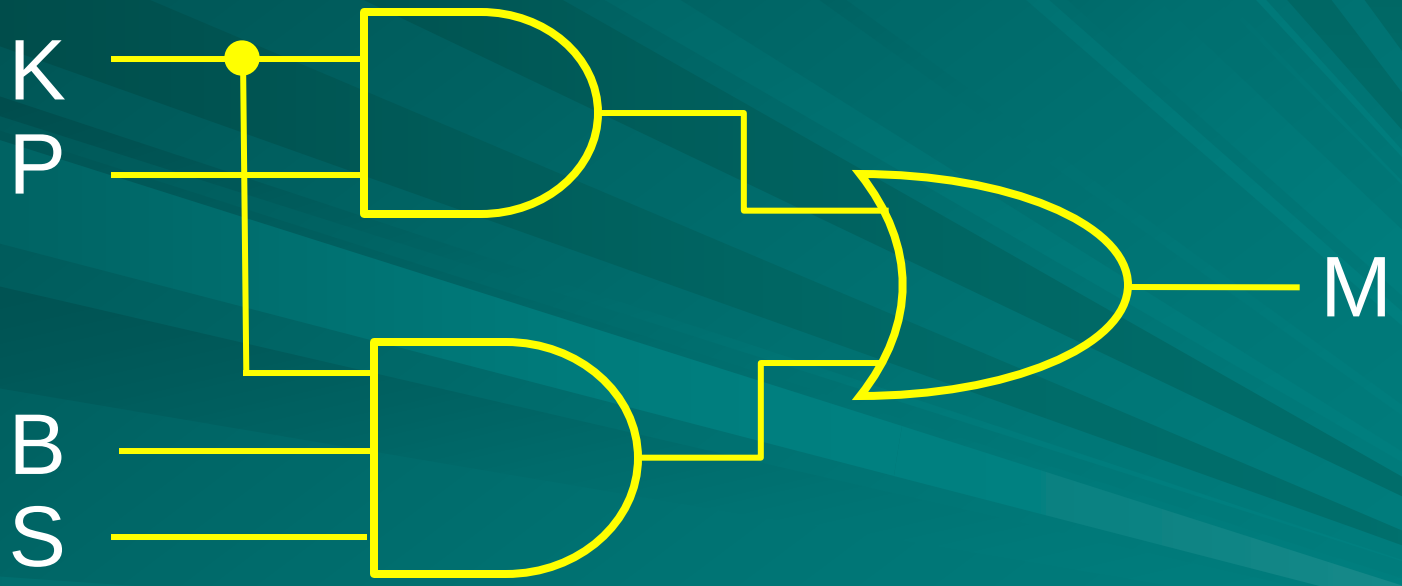


Karnaugh Map: Minimum Cover



Minimized Function

M = KP + KBS



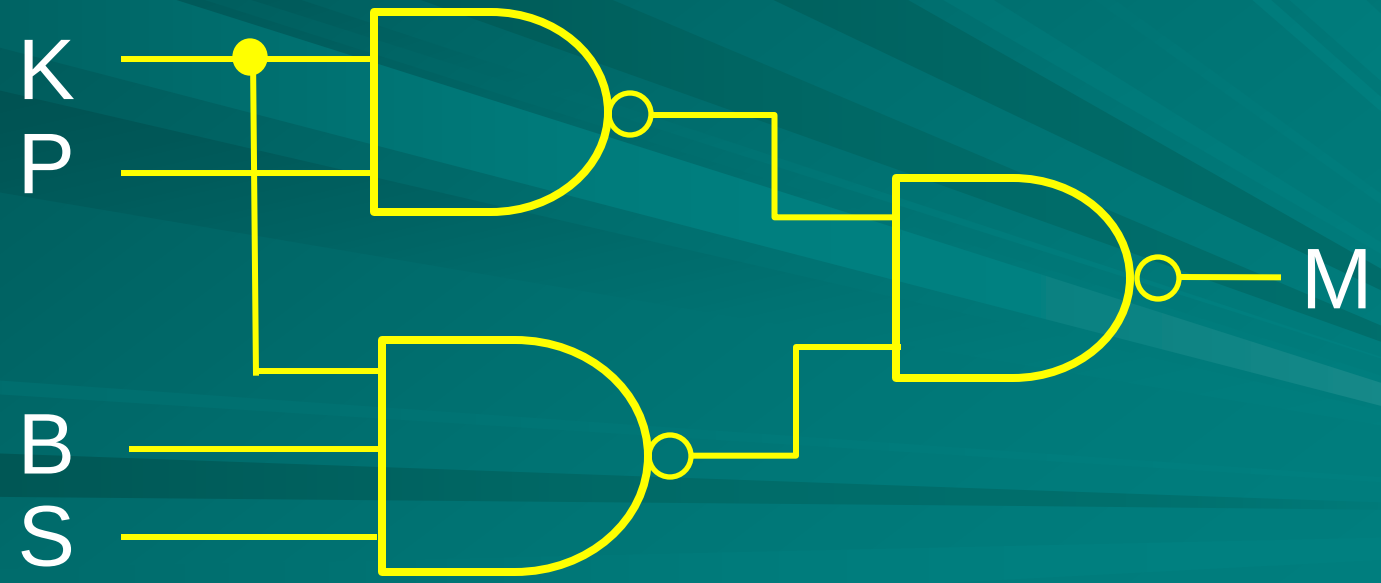
Using Inverting Gates

Because They Need Fewer Transistors

$$M = \overline{KP + KBS}$$

$$= \overline{\overline{KP} \cdot \overline{KBS}}$$

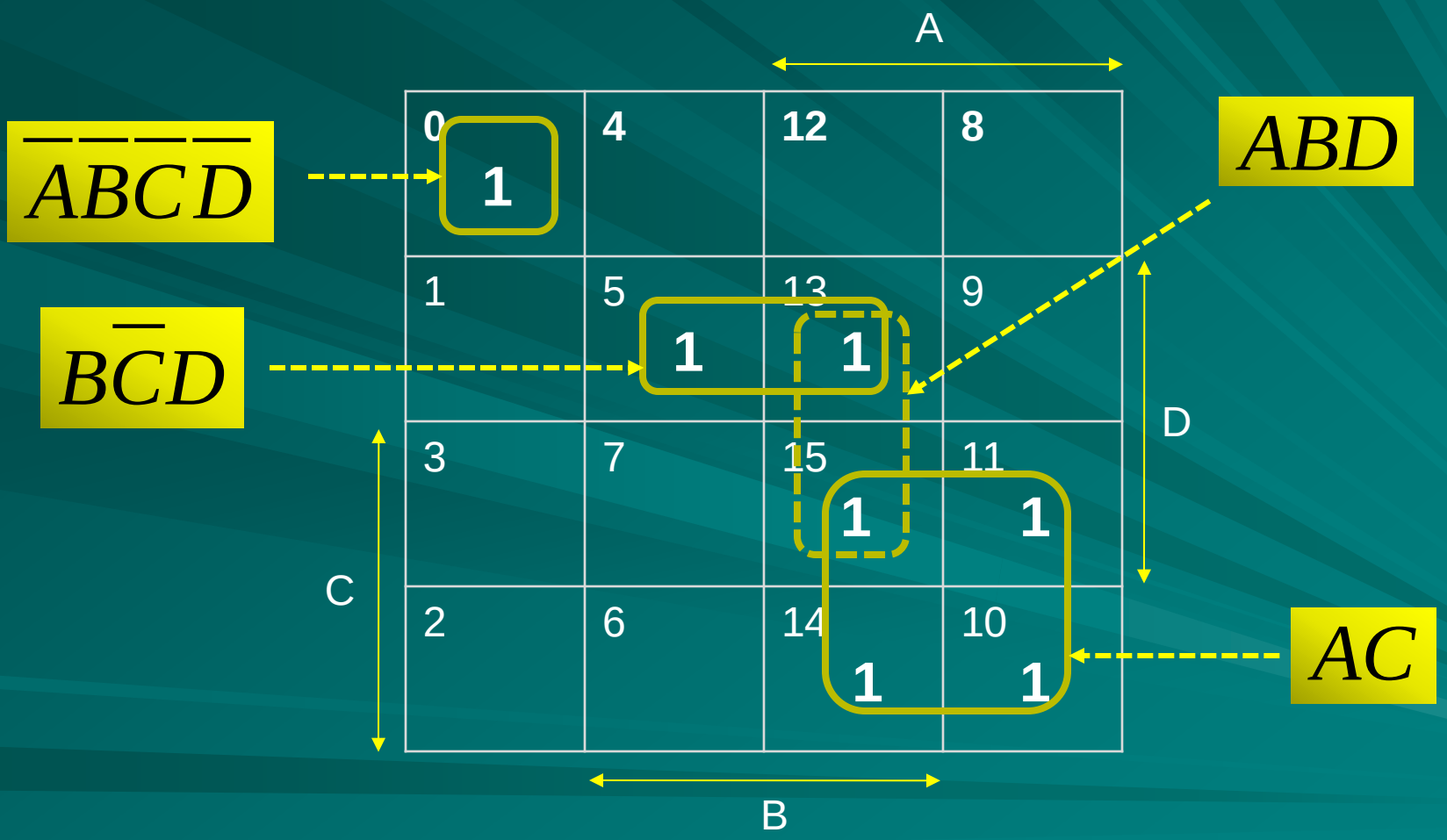
Using de Morgan's Theorem



Karnaugh Map on the Web

http://courseware.ee.calpoly.edu/~rsandige/KarnaughExplorer.html

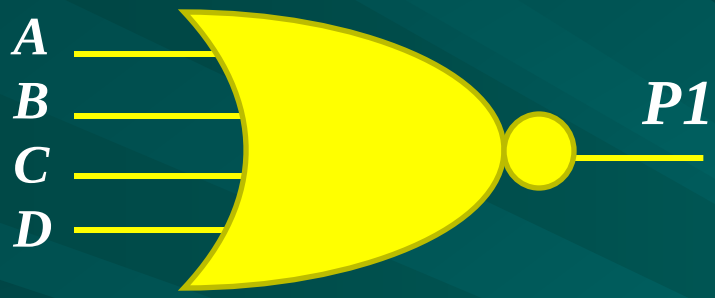
Minterm Clusters, Cubes or Products



Larger clusters are products with fewer variables.

Product

$$P1 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} = \overline{A + B + C + D}$$



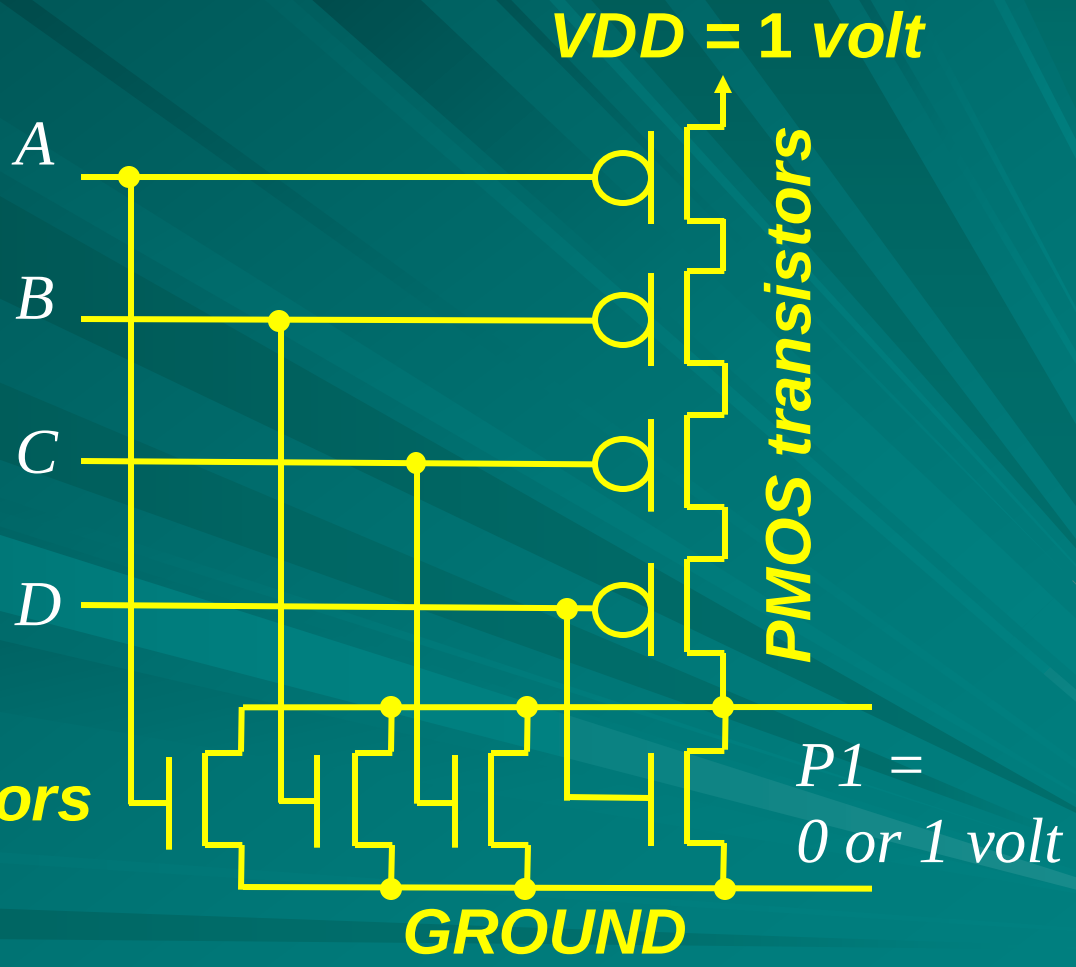
Signals are voltages w.r.t. GROUND

A = 0 or 1 volt

B = 0 or 1 volt

C = 0 or 1 volt

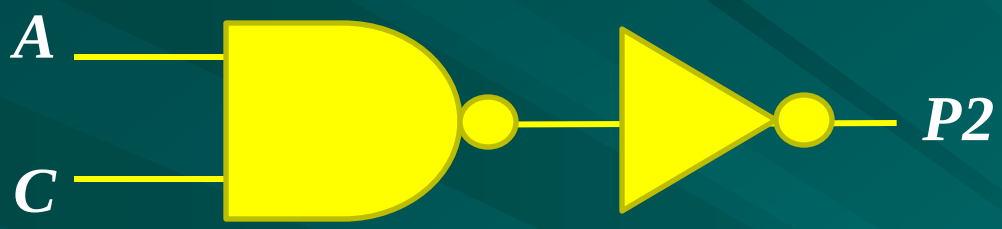
D = 0 or 1 volt



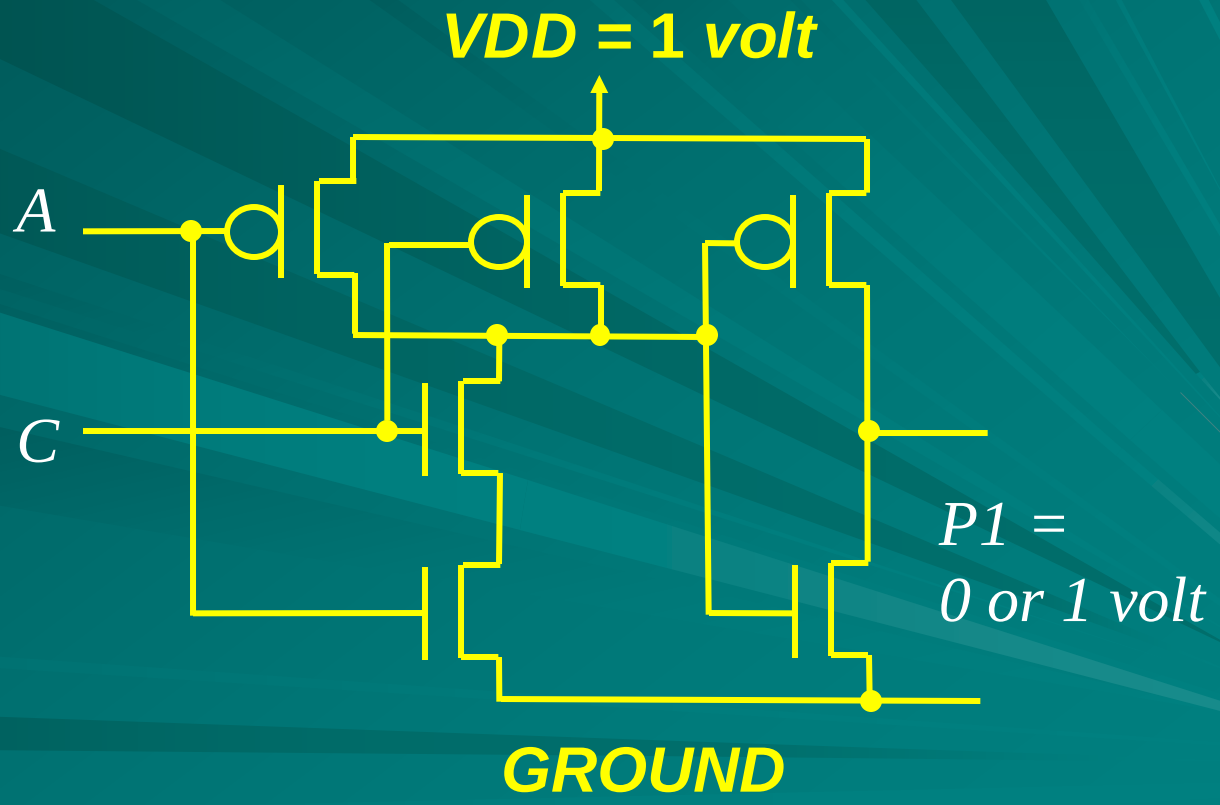
More variables in a product mean more transistors (hardware).

Product

$$P2 = AC = \overline{\overline{AC}}$$



Signals are voltages w.r.t. GROUND
A = 0 or 1 volt
B = 0 or 1 volt
C = 0 or 1 volt
D = 0 or 1 volt



Fewer variables in a product mean fewer transistors (hardware).

Observations

- A larger minterm cluster is a product with fewer variables; requires fewer transistors.
  - Each gate input needs two transistors.
- Smaller number of clusters is more efficient:
  - Fewer gates to generate products.
  - Fewer inputs for the OR gate to produce the function.
- Direct minterm implementation is most inefficient.

INTRODUCTION TO VHDL

HDLs in Digital System Design

- Model and document digital systems
  - Hierarchical models
    - System, RTL (Register Transfer Level), gates
  - Different levels of abstraction
    - Behavior, structure
- Verify circuit/system design via simulation
- Synthesize circuits from HDL models

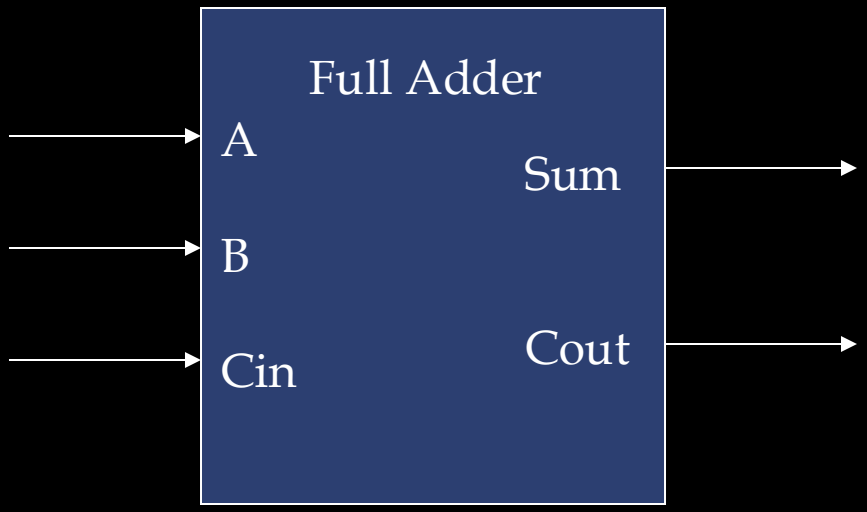
Hardware Description Languages

- VHDL = VHSIC Hardware Description Language (VHSIC = Very High Speed Integrated Circuits)
  - Developed by DOD from 1983 – based on ADA
  - IEEE Standard 1076-1987/1993/200x
  - Based on the ADA language
- Verilog – created in 1984 by Philip Moorby of Gateway Design Automation (merged with Cadence)
  - IEEE Standard 1364-1995/2001/2005
  - Based on the C language
  - IEEE P1800 "System Verilog" in voting stage & will be merged with 1364

Anatomy of a VHDL model

- "Entity" describes the external view of a design/component
- "Architecture" describes the internal behavior/structure of the component
- Example:

1-bit full adder



Entity Inputs/Outputs

- External view comprises input/output signals ("ports")
- A "port" is defined by its signal name, direction and type:

  port_name: direction data_type;

  - direction:
    - in - driven into the entity from an external source
    - out - driven from within the entity
    - inout - bidirectional – drivers within the entity and external
  - data_type: any scalar or aggregate signal type

IEEE Standard 1164 Data Types

- Type std_logic data values:
  'U', 'X' – uninitialized/unknown
  '0', '1' – strongly-driven 0/1
  'L', 'H' – weakly-driven 0/1 (resistive)
  'Z', 'W' - strong/weak "floating"
  '-' - don't care
- Type std_logic_vector is array of std_logic
- Include package:
  library IEEE;
  use IEEE.std_logic_1164.all;

Entity format

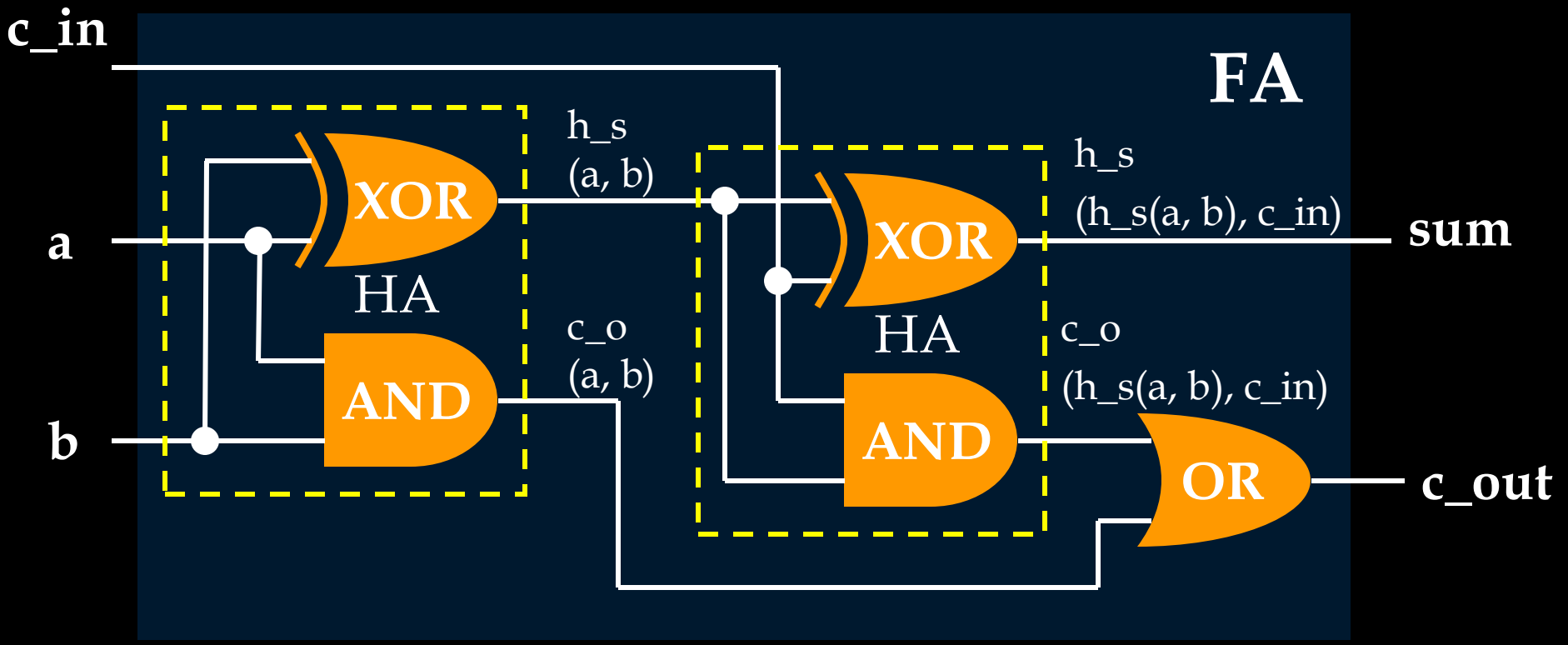
ENTITY *entity_name* IS

GENERIC (optional)
(*generic_name*: type :=*default_value*;
…
generic_name: mode signal_type);

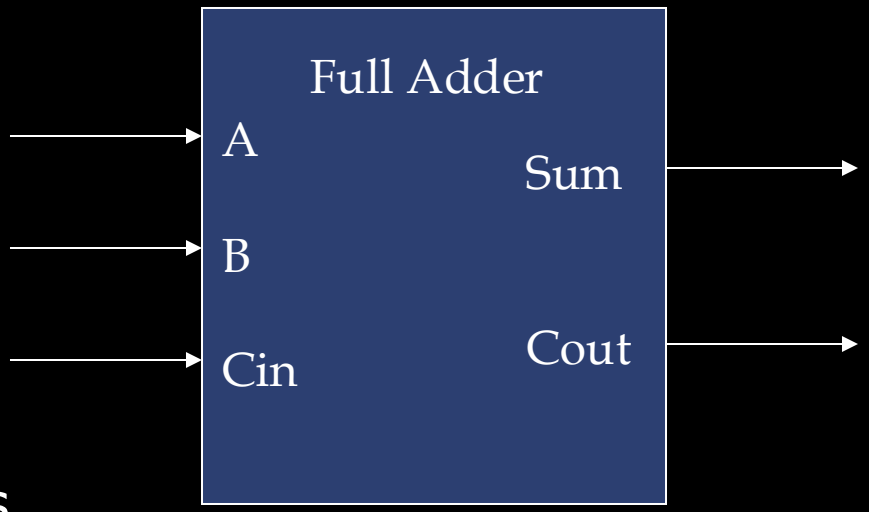
PORT
(*signal_name*: mode signal_type;
…
signal_name: mode signal_type);

END ENTITY *entity_name*;

Full-Adder Adds Three Bits



Entity example
(1-Bit Full Adder)




```
ENTITY Full_adder IS
   PORT (                    -- I/O ports
         a:    IN STD_LOGIC;      -- a input
         b:    IN STD_LOGIC;      -- b input
         cin:  IN STD_LOGIC;      -- carry input
         sum:  OUT STD_LOGIC;  -- sum output
         cout: OUT STD_LOGIC);  -- carry output
END Full_adder ;
```

Architecture format

ARCHITECTURE *architecture_name* OF *entity_name* IS

-- data type definitions (ie, states, arrays, etc.)

-- internal signal declarations

-- component declarations

-- function and procedure declarations

BEGIN

-- *behavior of the model is described here using:*

-- component instantiations

-- concurrent statements

-- processes

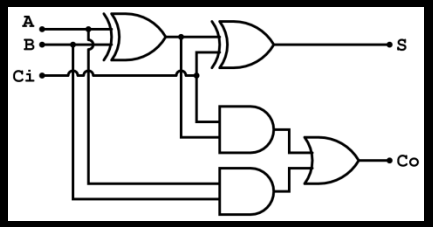
END ARCHITECTURE *architecture_name*;

Dataflow architecture example

```
ARCHITECTURE dataflow OF Full_adder IS
BEGIN
    sum <= a xor b xor cin;
   cout <= (a and b) or (a and cin) or
                (b and cin);
END dataflow;
```

Structural architecture example

```
ARCHITECTURE structure OF Full_adder IS
        COMPONENT xor IS              -- declare component to be used
                PORT (x,y: IN STD_LOGIC;
                        z: OUT STD_LOGIC);
        END COMPONENT xor;
        COMPONENT or IS               -- declare component to be used
                PORT (x,y: IN STD_LOGIC;
                        z: OUT STD_LOGIC);
        END COMPONENT or;

        COMPONENT and IS              -- declare component to be used
                PORT (x,y,z: IN STD_LOGIC;
                        p: OUT STD_LOGIC);
        END COMPONENT xor;

        SIGNAL x1,x2,x3,x4: STD_LOGIC;          -- signal internal to this component
BEGIN
        G1: xor PORT MAP (a, b, x1); -- instantiate 1st  xor gate
        G2: xor PORT MAP (x1, Cin, Sum);  -- instantiate 2nd xor gate
        G3: or PORT MAP (a, b, x2);   -- instantiate 1st  or gate
        G4: or PORT MAP (a, Cin, x3);  -- instantiate 2nd or gate
        G5: or PORT MAP (b, Cin, x4);                -- instantiate 3rd  or gate
        G6: and PORT MAP (x2, x3, x4, Cout);  -- instantiate and gate

END structure;
```



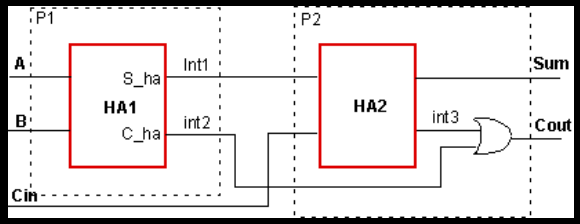
Full-adder

Alternative structural architecture example

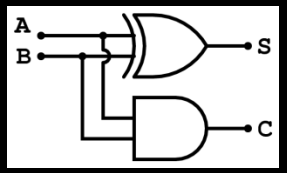
```
ARCHITECTURE structural OF Full_adder IS

    COMPONENT half_adder
    PORT(a,b : IN STD_LOGIC;
    sum, carry : OUT STD_LOGIC);
    END COMPONENT;

    COMPONENT or_2
    PORT(a,b : IN STD_LOGIC;
    c : OUT STD_LOGIC);
    END COMPONENT;

    SIGNAL int1, int2, int3 : STD_LOGIC;

    BEGIN

    H1: half_adder port map(a=>A, b=>B,
    sum=>int1, carry=>int3);
    H2: half_adder port map(a=>s1, b=>C_in,
    sum=>sum, carry=>s2);
    O1: or_2 port map(a=> int2, b=>int3,
    c=>C_out);

    END structural;
```

```
ENTITY half_adder IS

PORT (a,b : IN STD_LOGIC ;
sum,carry : OUT STD_LOGIC);

END half_adder;

ARCHITECTURE dataflow OF
half_adder IS

BEGIN

sum<= a xor b;
carry <= a and b;

END dataflow;
```

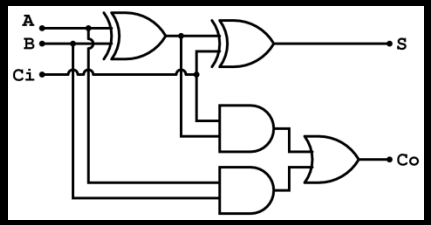
```
ENTITY or_2 IS

PORT (a,b : IN STD_LOGIC ;
c : OUT STD_LOGIC);

END or_2;

ARCHITECTURE dataflow OF
or_2 IS

BEGIN

c<= a or b;

END dataflow;
```



Full-adder



Each Half-adder

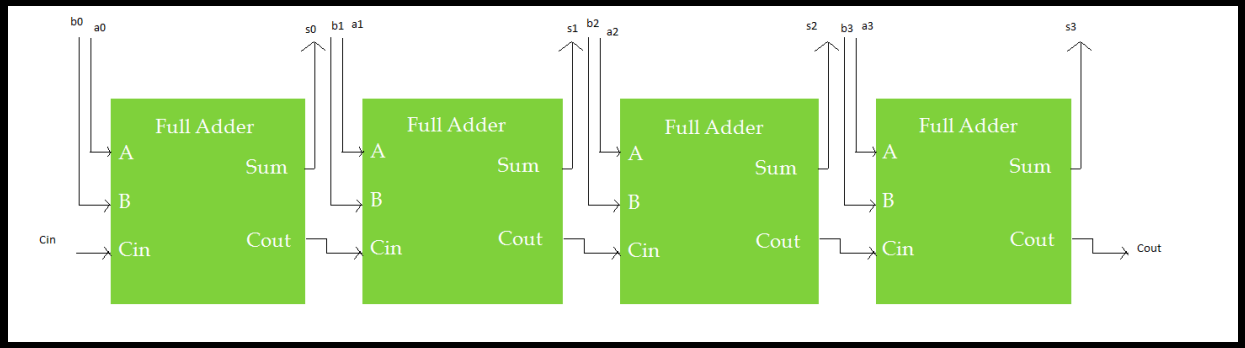


Full-adder

Extending full-adder circuit to multiple-bit adition

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY adder_4bit IS
    PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        Cin : IN STD_LOGIC;
            sum: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
            Cout: OUT STD_LOGIC);
    END adder_4bit;

ARCHITECTURE structural OF adder_4bit IS

    SIGNAL c: STD_LOGIC_VECTOR (4 DOWNTO 0);

COMPONENT Full_adder
        PORT(a, b, c: IN STD_LOGIC;
sum, carry: OUT STD_LOGIC);
        END COMPONENT;
```

```
BEGIN

        FA0: Full_adder
            PORT MAP (a(0), b(0), Cin, sum(0), c(1));
        FA1: Full_adder
            PORT MAP (a(1), b(1), C(1), sum(1), c(2));
        FA2: Full_adder
            PORT MAP (a(2), b(2), C(2), sum(2), c(3));
        FA3: Full_adder
            PORT MAP (a(3), b(3), C(3), sum(3), c(4));

        Cout <= c(4);

END structural;
```



Behavioral architecture example

```
ARCHITECTURE behavioral OF Full_adder
    IS
BEGIN
    Sum: PROCESS(a, b, cin)
    BEGIN
        sum <= a xor b xor cin;
    END PROCESS Sum;
    Carry: PROCESS(a, b, cin)
    BEGIN
        cout <= (a and b) or (a and cin) or
    (b and cin);
    END PROCESS Carry;
END behavioral;
```

```
ARCHITECTURE behavioral OF
    Full_adder IS
BEGIN
    PROCESS(a, b, cin)
    BEGIN
        sum <= a xor b xor cin;
        cout <= (a and b) or (a and cin)
    or (b and cin);
    END PROCESS;
END behavioral;
```

VHDL "Process" Construct

- Allows conventional programming language methods to describe circuit behavior
- Supported language constructs ("sequential statements") – only allowed within a process:
  - variable assignment
  - if-then-else (elsif)
  - case statement
  - while (condition) loop
  - for (range) loop

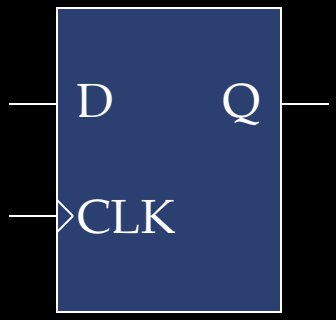
Process Format

[label:] process (*sensitivity list*)
declarations
begin
sequential statements
end process;

- Process statements executed once at start of simulation
- Process halts at "end" until an event occurs on a signal in the "sensitivity list"

Using a "process" to model sequential behavior

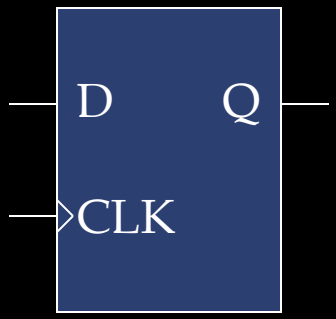
```
ENTITY dff IS
  PORT (d,clk:  IN STD_LOGIC;
        q:      OUT STD_LOGIC);
END dff;

ARCHITECTURE behavioral OF dff IS
BEGIN
    PROCESS(clk)   -- "process sensitivity list"
    BEGIN
        IF (clk'event and clk='1') THEN
                q <= d AFTER 1 ns;
        END IF;
    END PROCESS;
END behavioral;
```



- Process statements executed sequentially (sequential statements)
- clk'event is an attribute of signal clk which is TRUE if an event has occurred on clk at the current simulation time

Alternative to sensitivity list

```
ENTITY dff IS
  PORT (d,clk: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
ARCHITECTURE behavioral OF dff IS
BEGIN
      PROCESS   -- no "sensitivity list"
      BEGIN
          WAIT ON clk; -- suspend process until event on clk
          IF (clk='1') THEN
                    q <= d AFTER 1 ns;
          END IF;
      END PROCESS;
END behavioral;
```



- Other "wait" formats:

```
WAIT UNTIL (clk'event and clk='1')
WAIT FOR 20 ns;
```

- Process executes endlessly if no sensitivity list or wait statement!

Sequential statements in process

- if-then-elsif-else *statement*

```
if condition then
    (... sequence of statements...)
elsif condition then
    (... sequence of statements...)
else
    (... sequence of statements...)
end if;
```

- case *statement*

```
case expression is
    when choices => sequence of statements
    when choices => sequence of statements
    ...
    when others => sequence of statements
end case;
```

Sequential statements in process

- while *loop*

  [*label:*] while *condition* loop
  ... *sequence of statements* ...
  end loop [*label*];

- for *loop*

  [*label:*] for *loop_variable* in *range* loop
  ... *sequence of statements*...
  end loop [*label*];

Behavioral architecture example using conditional statements

```
ARCHITECTURE functional OF
    Full_adder IS
    BEGIN
        PROCESS(A,B,Cin)
        BEGIN
        If (Cin = '0' and A = '0' and B =
        '0' ) then
        sum<= '0'; Cout <= '0';
        elsif(Cin = '0' and A = '0' and B =
        '1') then
        sum <= '1' ; Cout <= '0';
        elsif(Cin = '0' and A = '1' and B =
        '0' ) then
        sum <= '1' ; Cout <= '0';
        elsif(Cin = '0' and A = '1' and B =
        '1' ) then
        sum<= '0'; Cout <= '1';
        elsif(Cin = '1' and A = '0' and B =
        '0' ) then
        sum <= '1' ; Cout <= '0';
        elsif(Cin = '1' and A = '0' and B =
        '1' ) then
        sum<= '0'; Cout <= '1';
        elsif(Cin = '1' and A = '1' and B =
        '0' ) then
        sum<= '0'; Cout <= '1';
        elsif(Cin = '1' and A = '1' and B =
        '1' ) then
        sum <= '1' ; Cout <= '1';
        else
        sum <= 'X' ; Cout <= 'X';
        end if;
    END PROCESS;
    END functional;
```

Sequential architecture example

```
ENTITY counter_4bit IS
  PORT(Ld, Clr, Clk: IN
       STD_LOGIC;
       D: IN STD_LOGIC_VECTOR (3
        DOWNTO 0);
       Cout: OUT STD_LOGIC;
       Qout: OUT
        STD_LOGIC_VECTOR (3
        DOWNTO 0));
END counter_4bit;
```

```
ARCHITECTURE behavioral OF counter_4bit IS
SIGNAL Q: STD_LOGIC_VECTOR(3 DOWNTO
0);
BEGIN
  Qout <= Q;
  Cout <= Q(3) and Q(2) and Q(1) and Q(0);
  PROCESS(Clk)
  BEGIN
    IF Clk'event and Clk = '1' THEN
      IF Clr = '0' THEN
                Q <= "0000";
      ELSIF Ld = '0' THEN
                Q <= D;
      ELSE
                Q <= Q + 1;
      END IF;
    END IF;
  END PROCESS;
END behavioral;
```

References

- VHDL mini-reference on *Prof. Nelson's* website
  - http://www.eng.auburn.edu/department/ee/mgc/vhdl.html
- VHDL resources on *Prof. Stroud's* website
  - http://www.eng.auburn.edu/~strouce/elec4200.html
- VHDL resources on Prof. Agrawal's website
  - http://www.eng.auburn.edu/~agrawvd/COURSE/E6200_Fall10/course.html
- http://esd.cs.ucr.edu/labs/tutorial/
- http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html#_Toc526061344
- http://www.vhdl.org/
- http://www.doulos.com/knowhow/vhdl_designers_guide/
- http://www.altera.com/support/examples/vhdl/vhdl.html
- http://www.vhdl-online.de/tutorial/
- http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m12_23/sld006.htm
- http://www.doc.ic.ac.uk/~ih/teaching/lectures/comparch/logic/adder/

Using Modelsim

- Modelsim PE (Student Edition) can be downloaded from

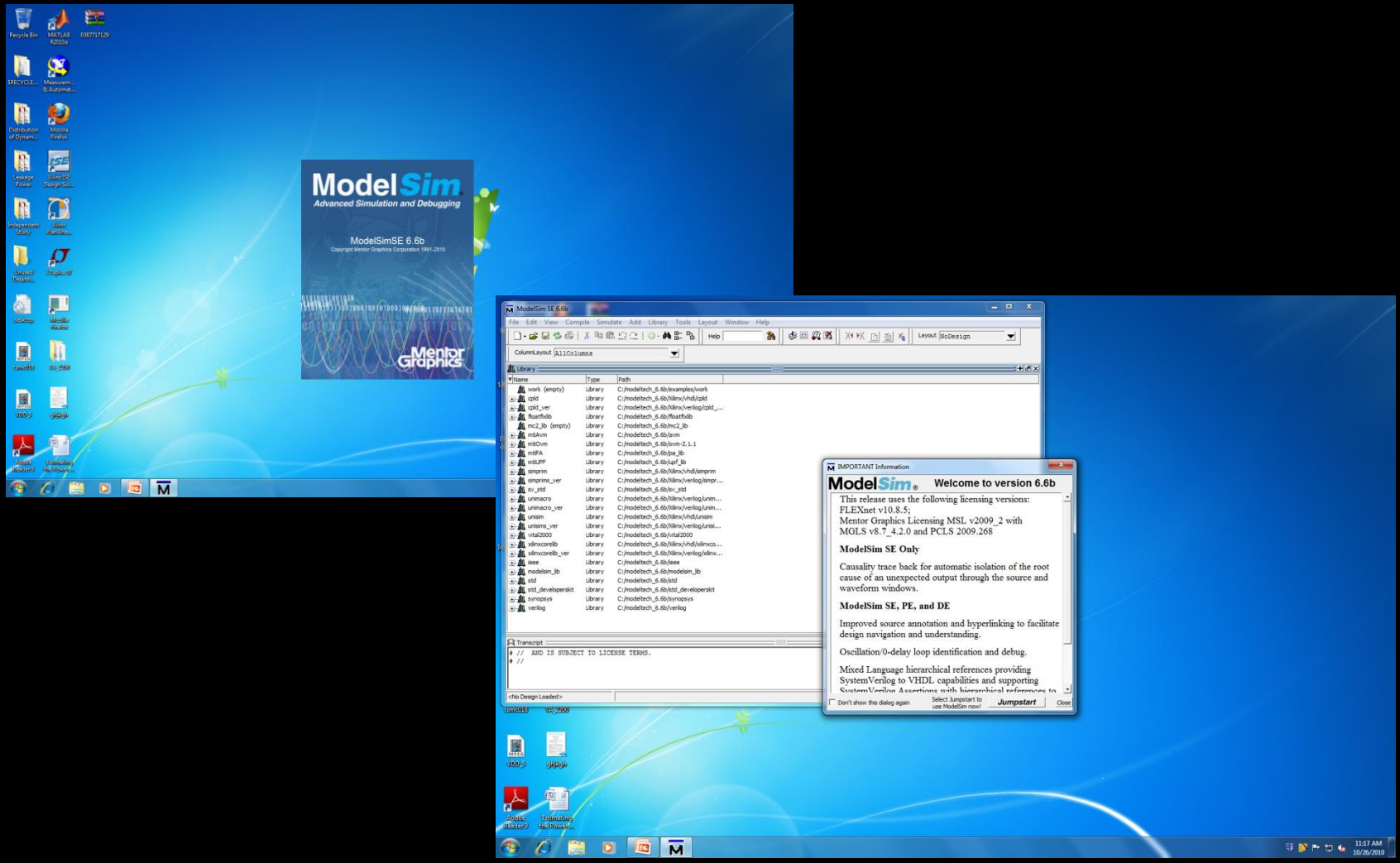
http://model.com/content/modelsim-pe-student-edition-hdl-simulation?quicktabs_4=1#quicktabs-4

- Modelsim is installed on the Windows7 platform in Lab 310, Broun Hall.

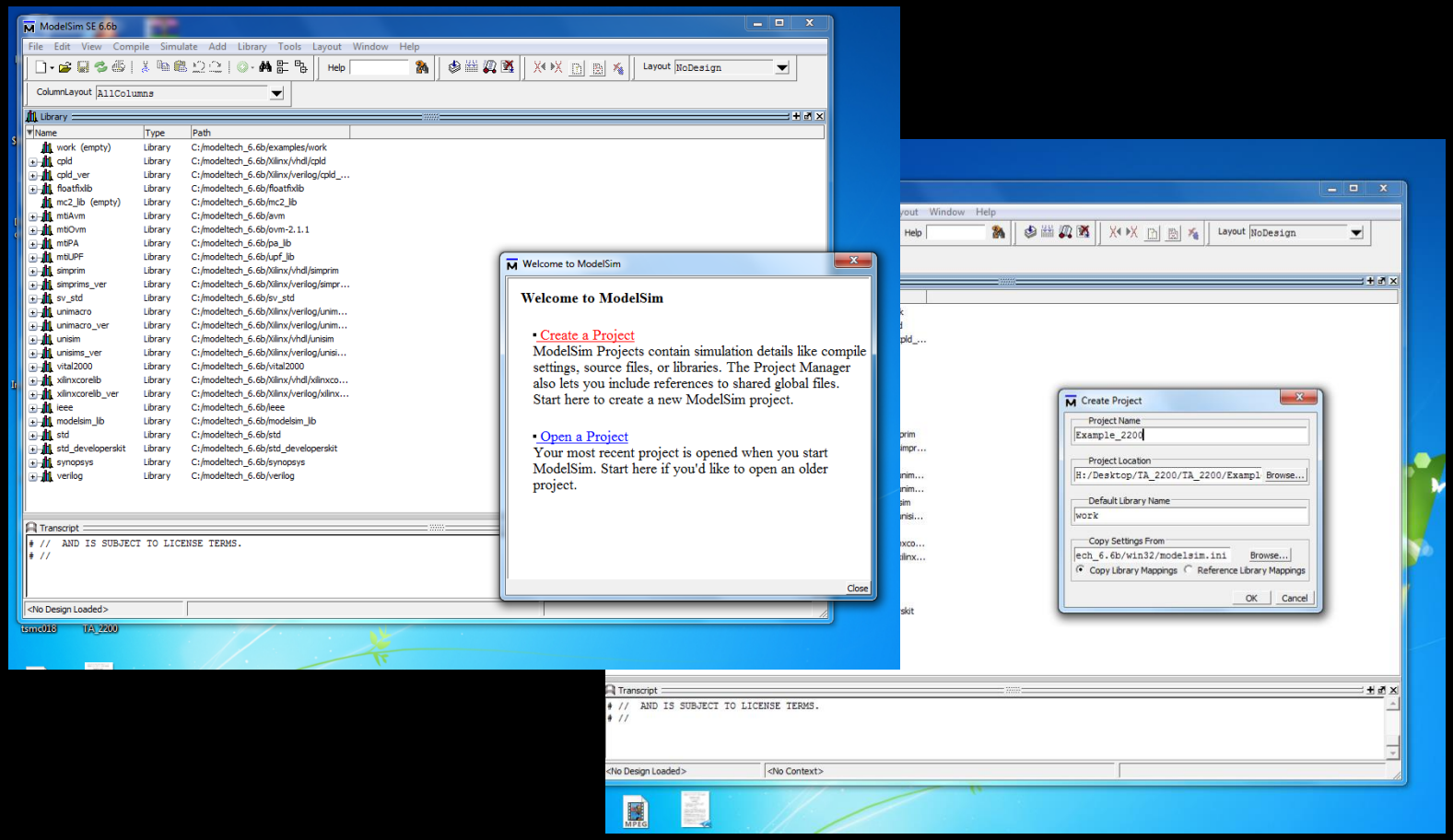
Using Modelsim



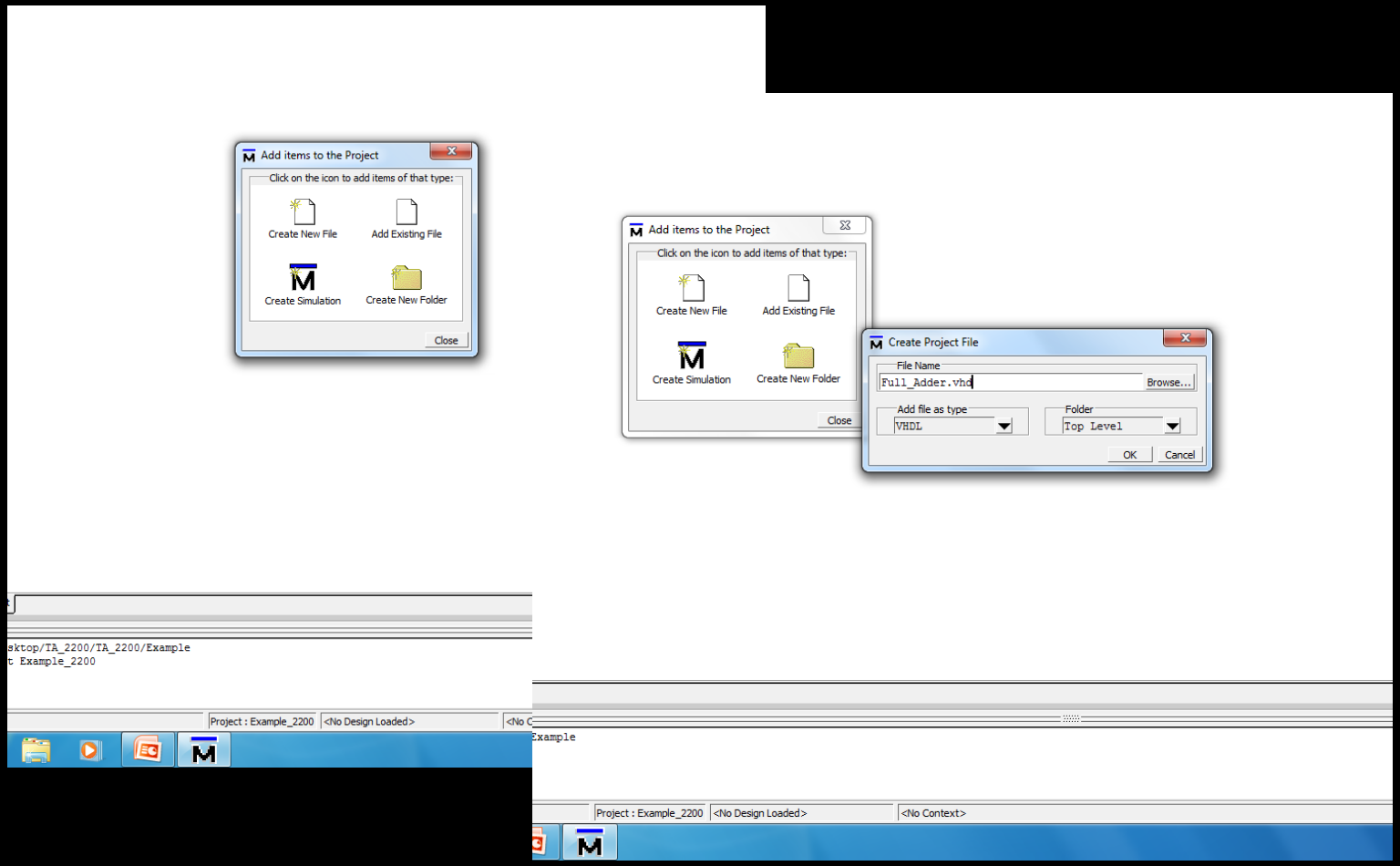
Using Modelsim



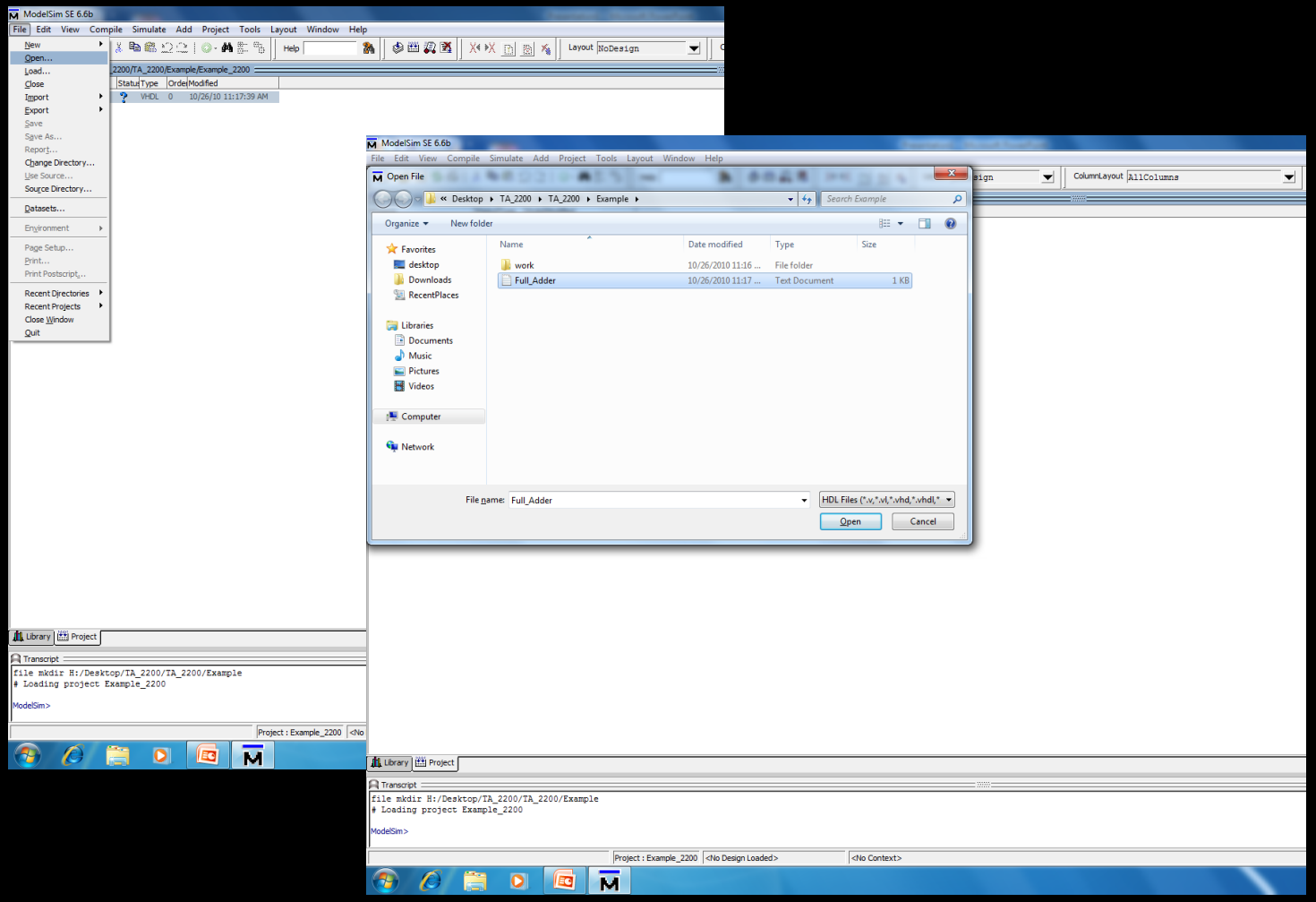
Using Modelsim



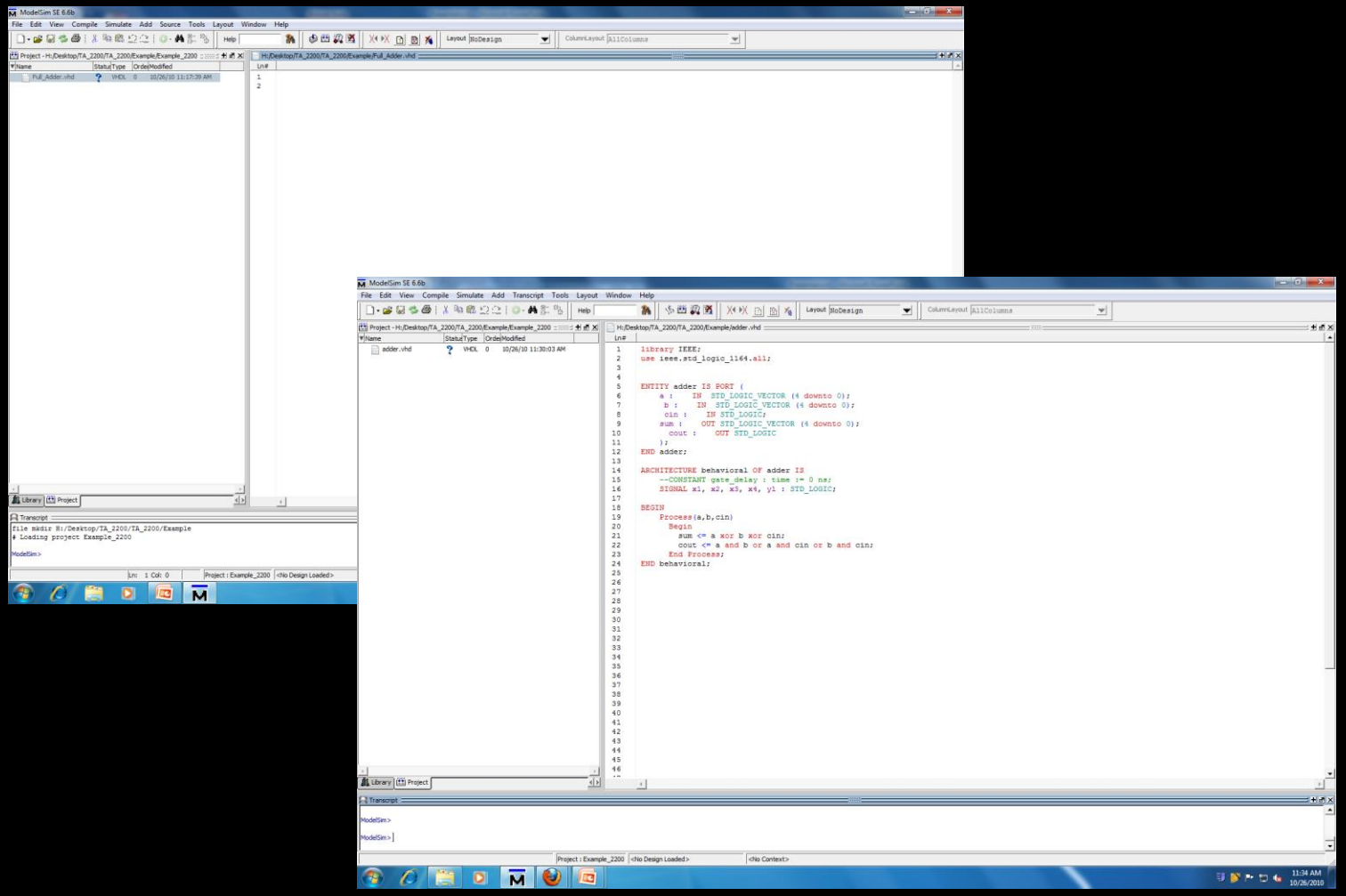
Using Modelsim



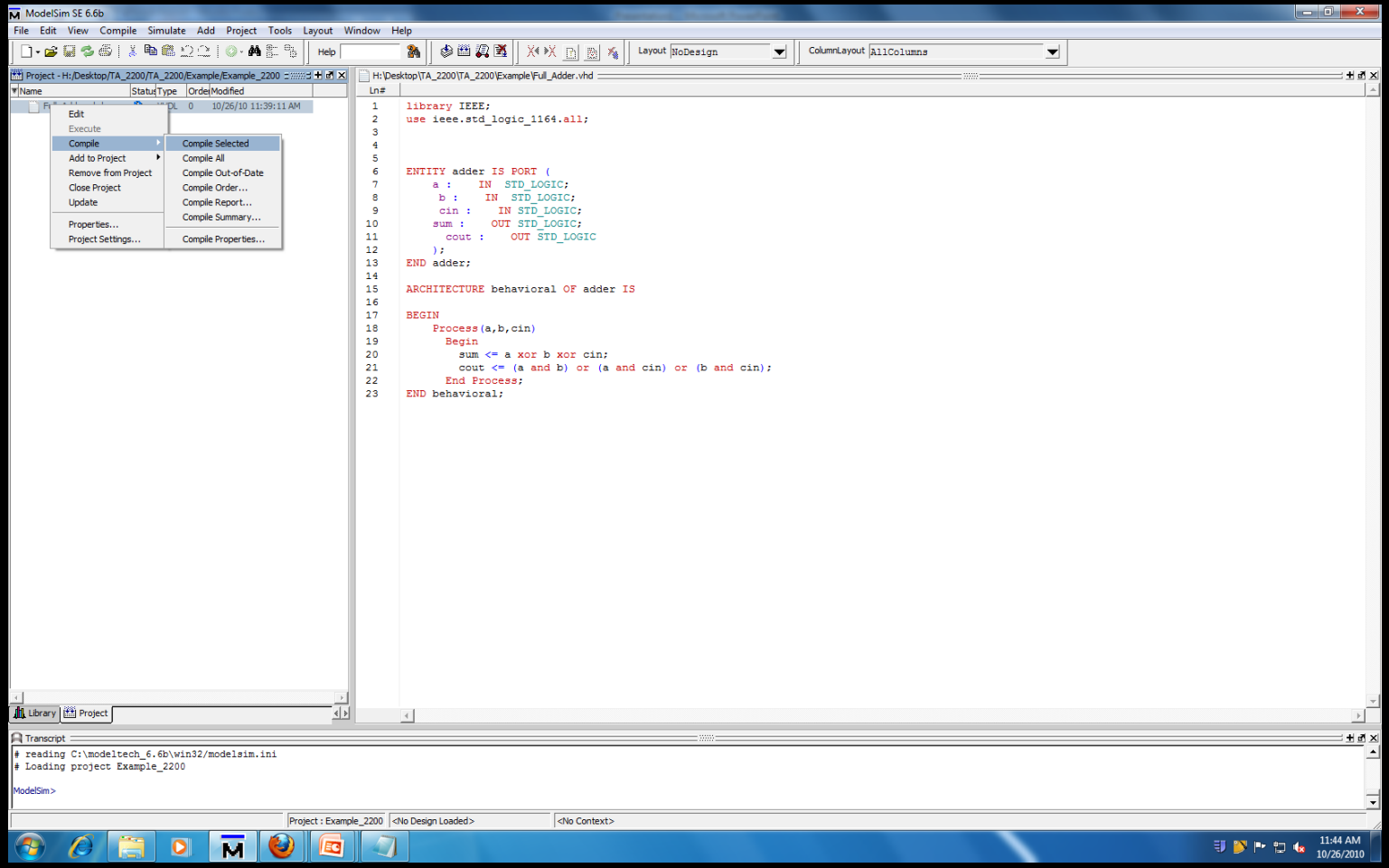
Using Modelsim



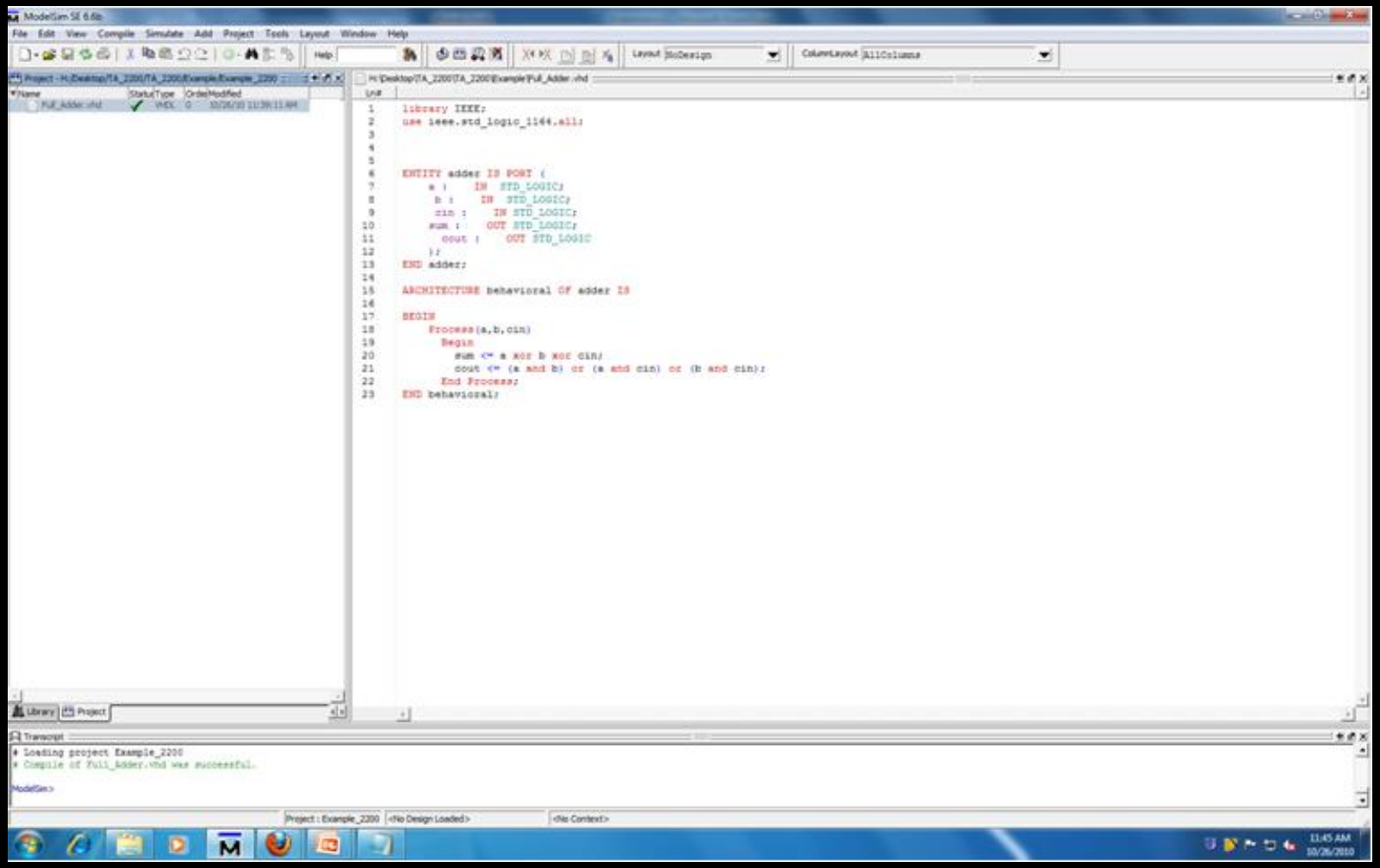
Using Modelsim



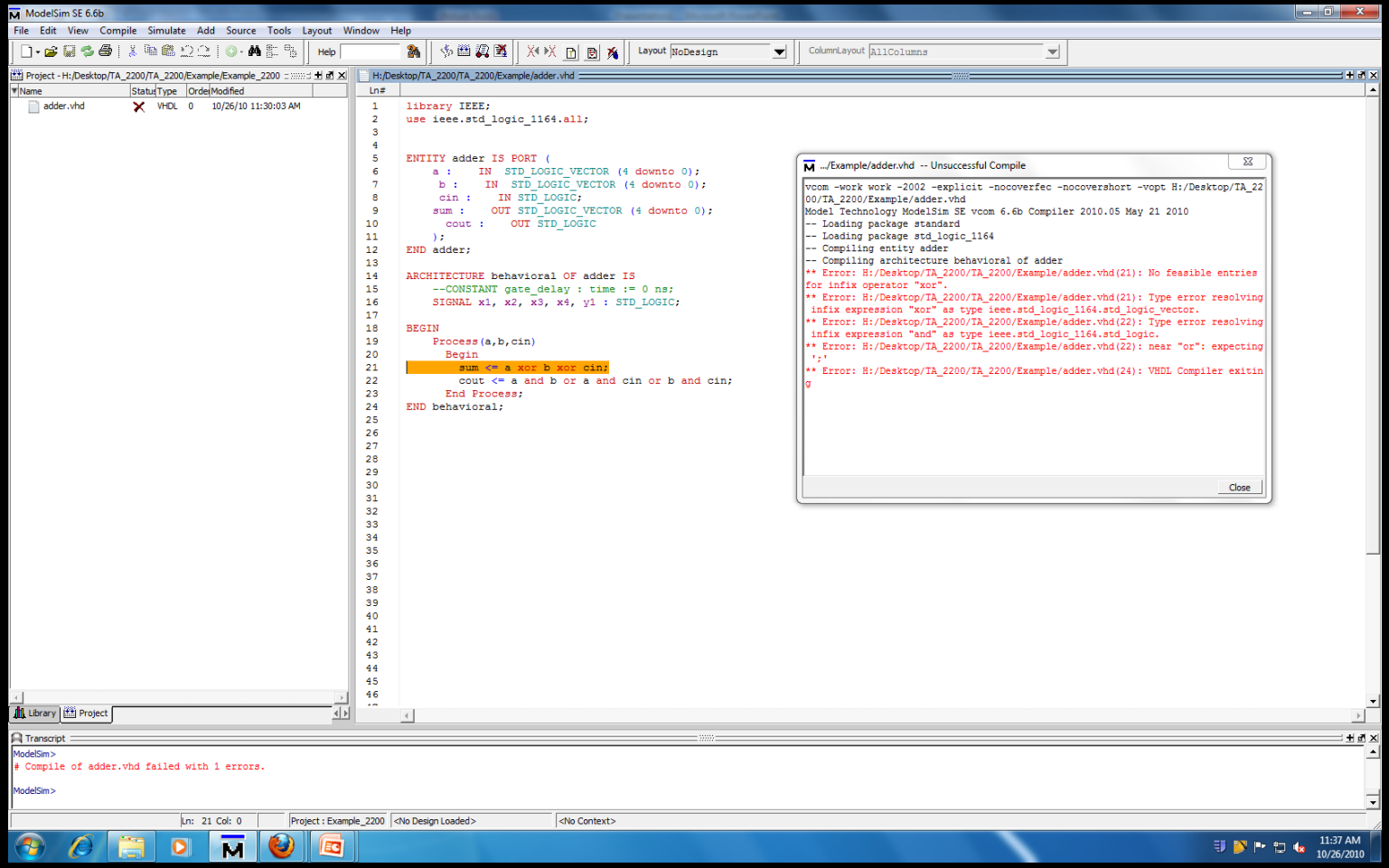
Using Modelsim



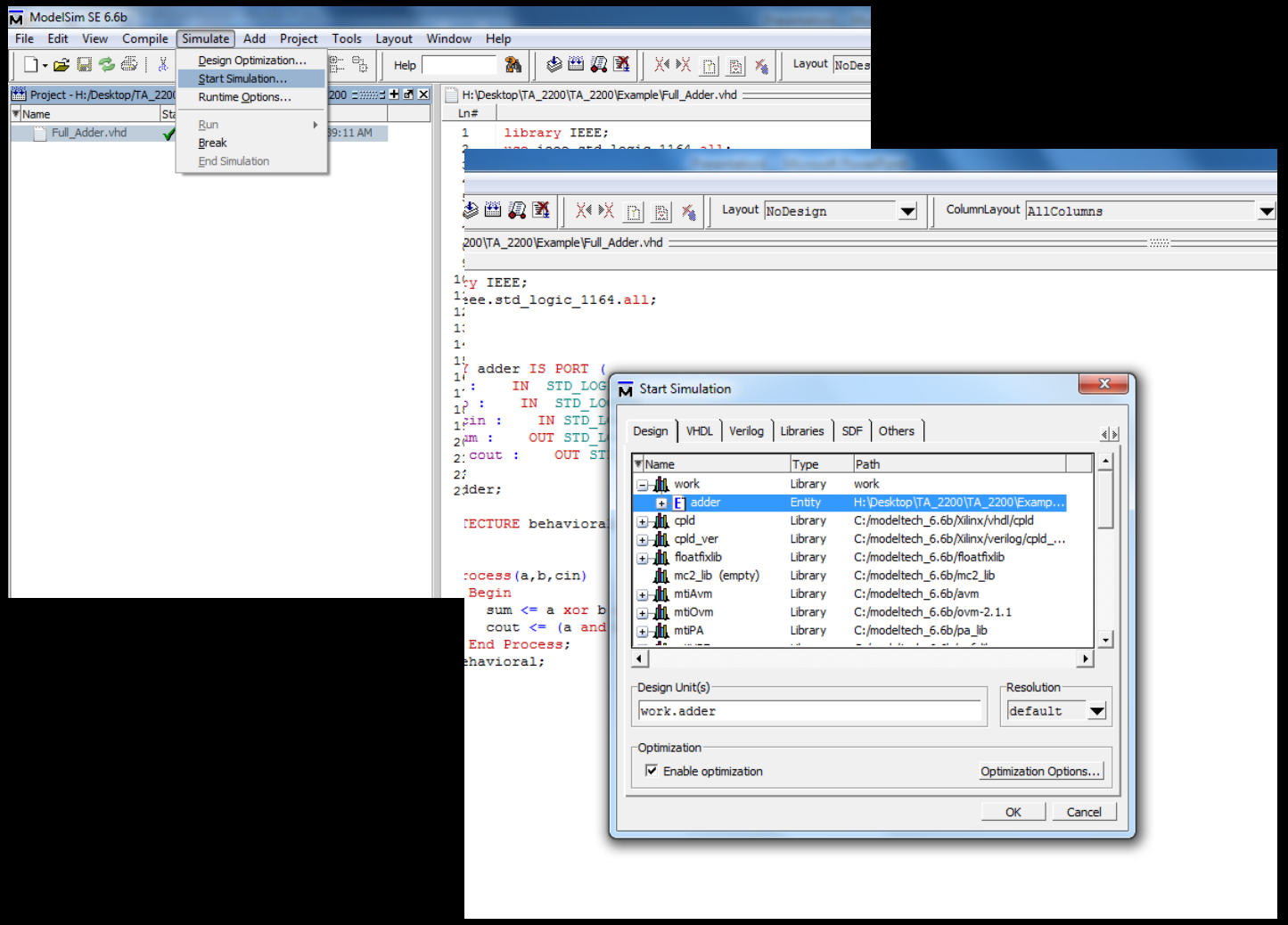
Using Modelsim



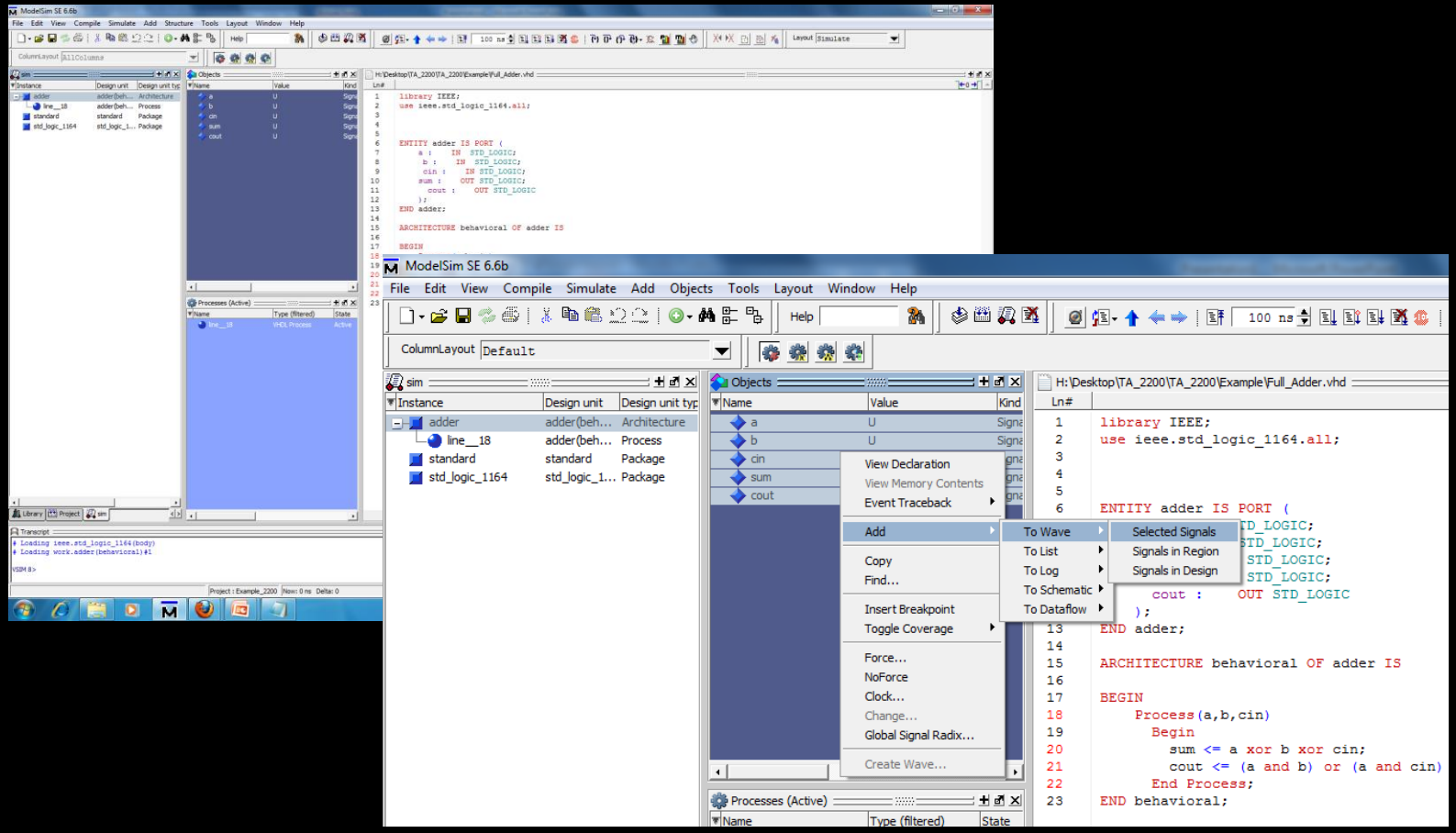
Using Modelsim



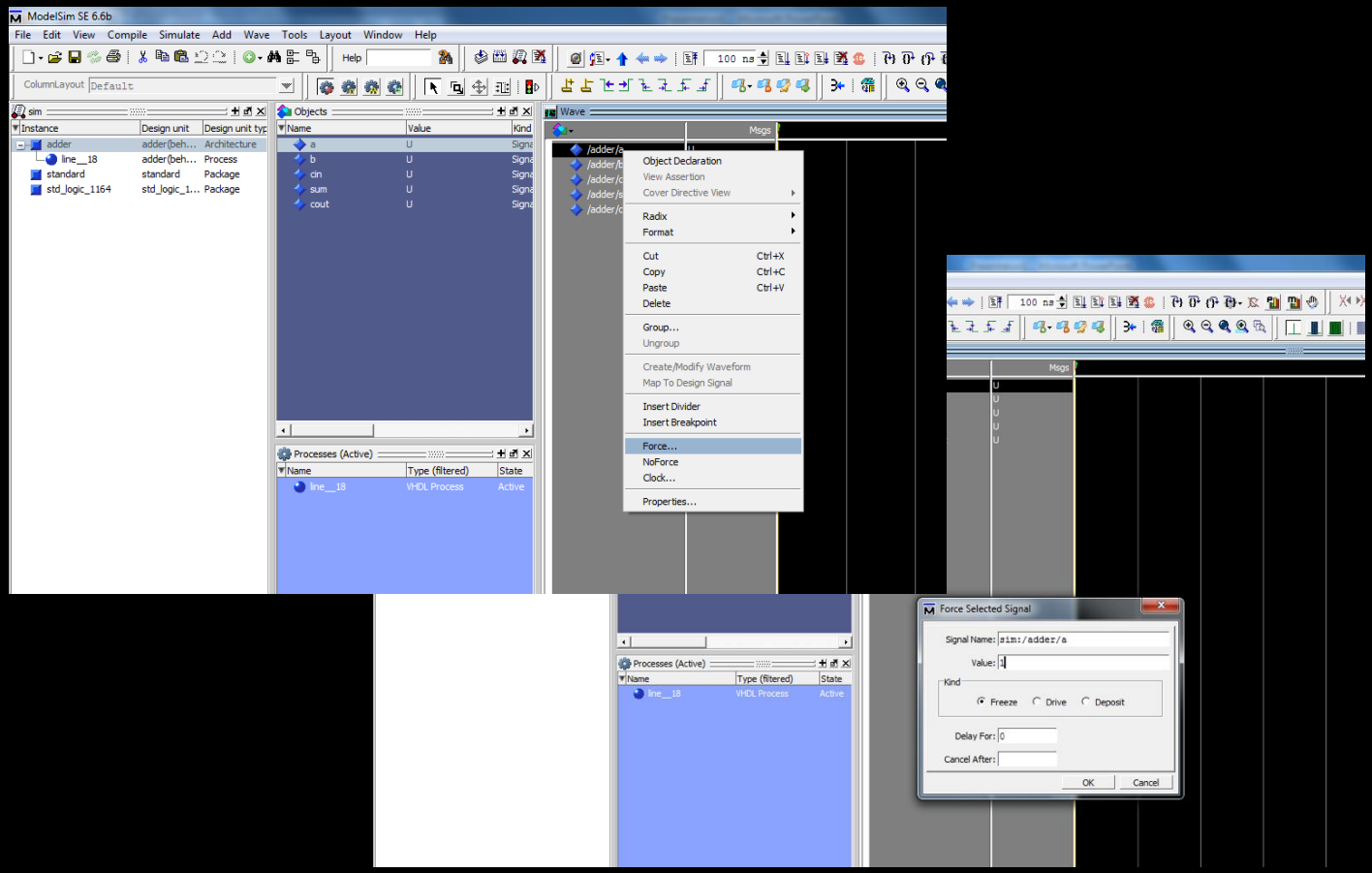
Using Modelsim



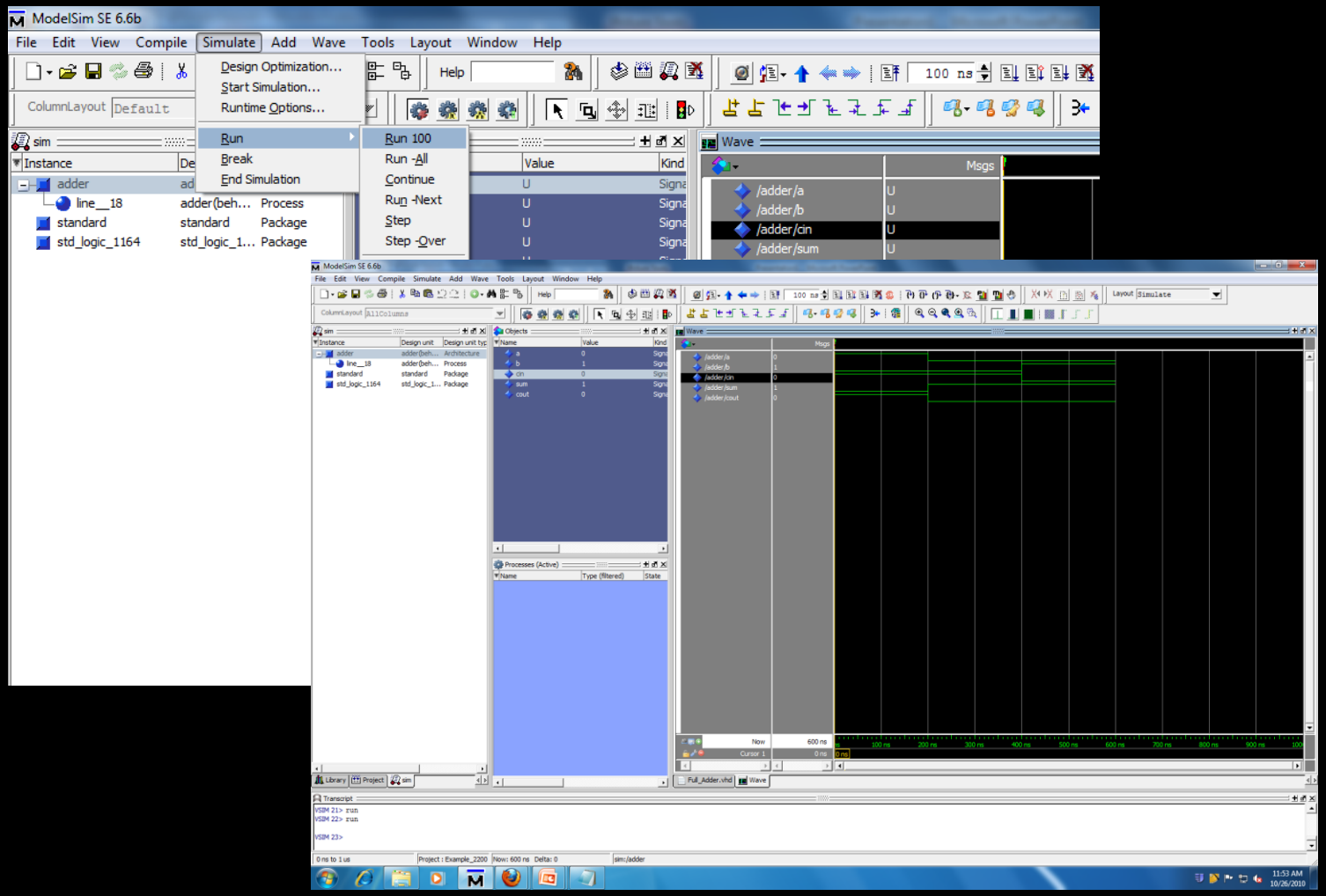
Using Modelsim



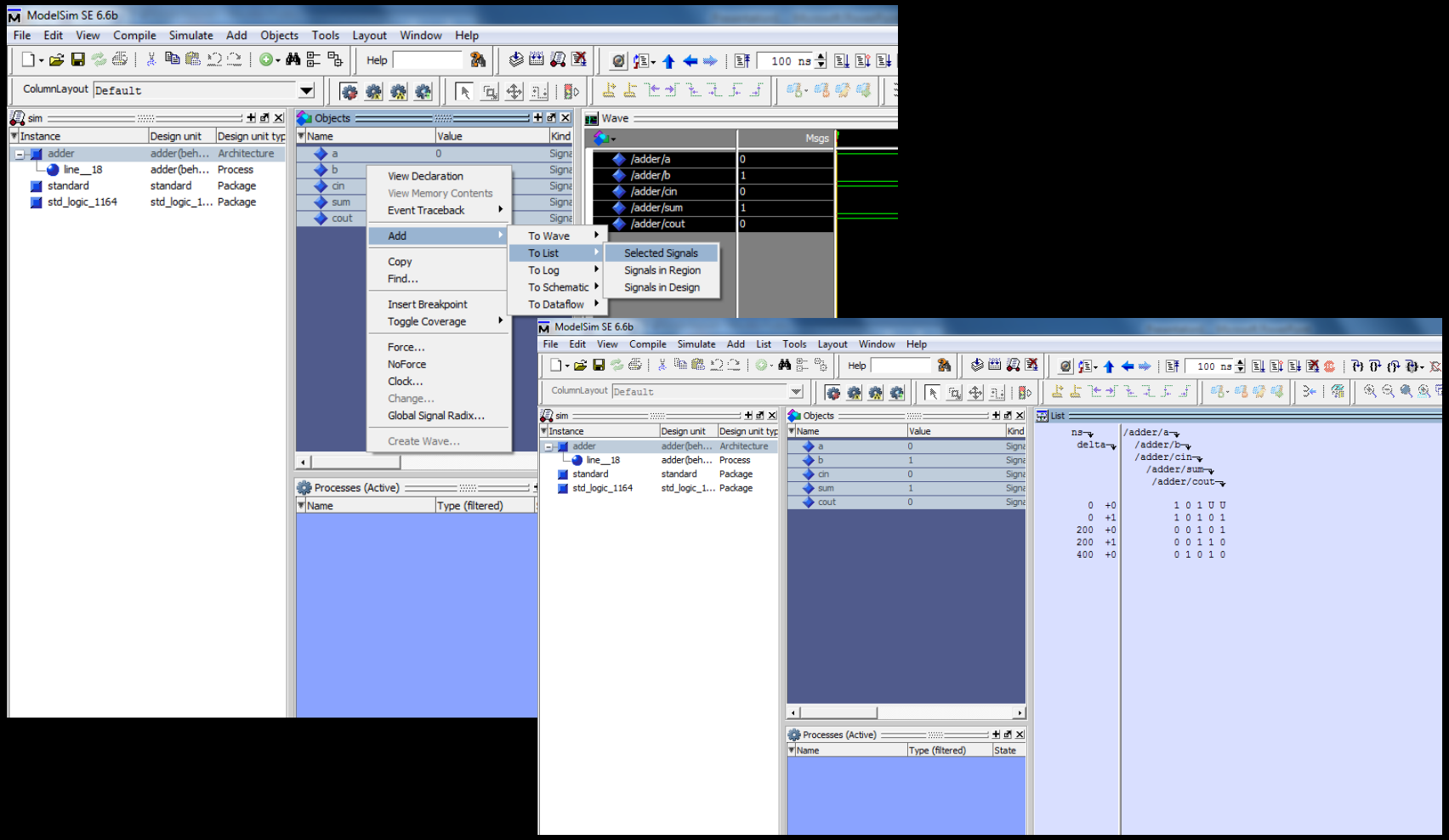
Using Modelsim



Using Modelsim



Using Modelsim

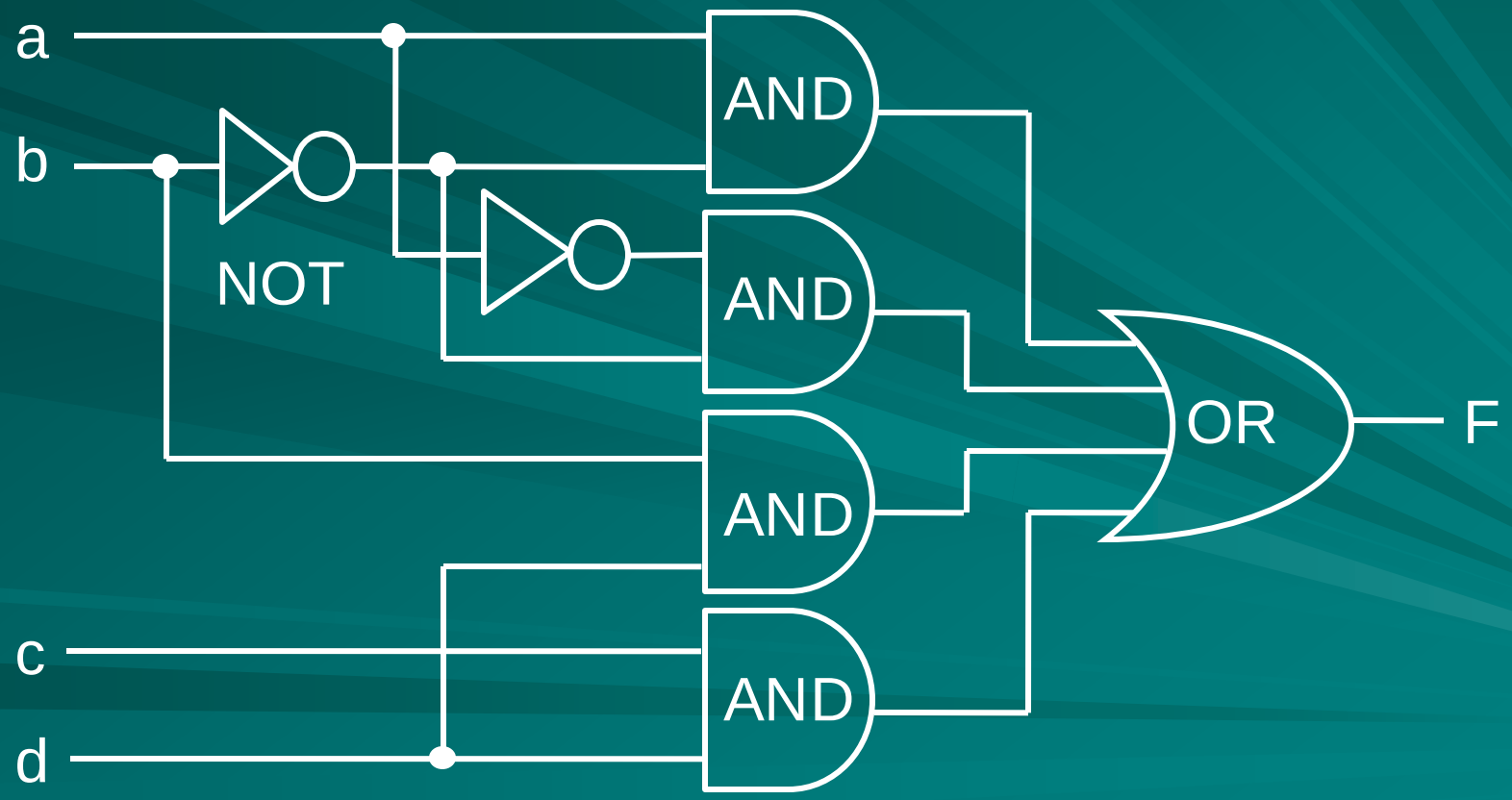


Digital Logic Circuits

Logic Minimization (Chapter 3)

Understanding Minimization . . .

- Logic function: $F = a\bar{b} + \bar{a}\bar{b} + bd + cd$



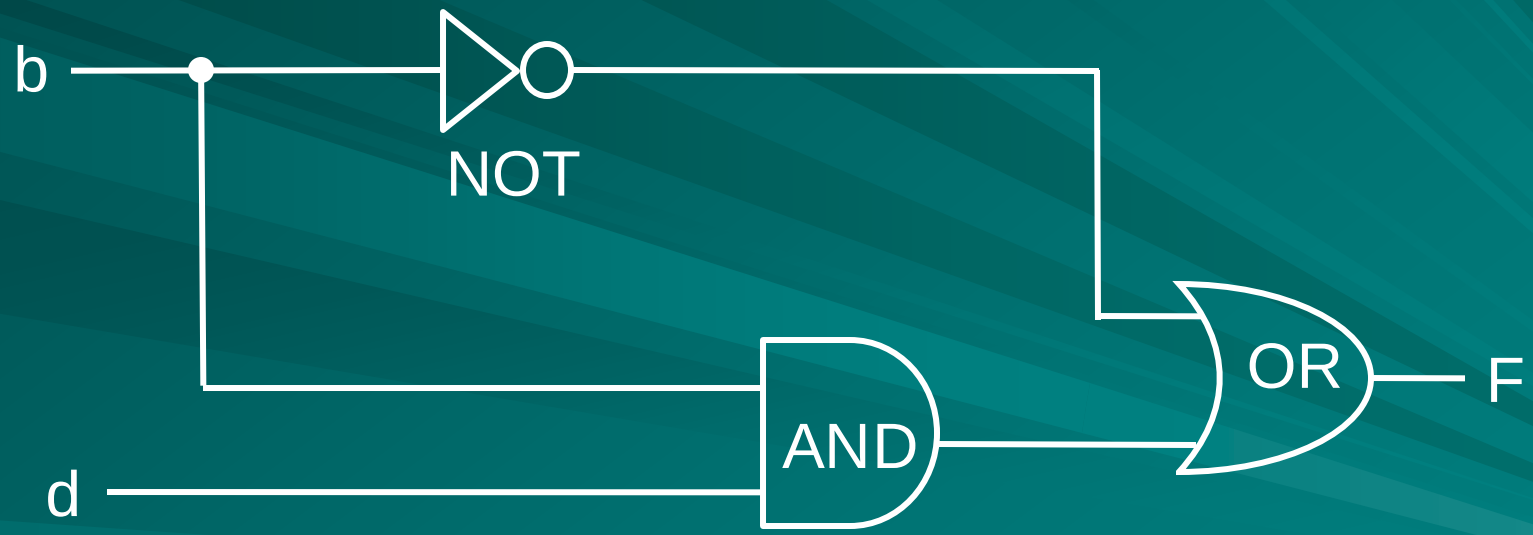
. . . Understanding Minimization

- Reducing products:

$F = a\bar{b} + \bar{a}\bar{b} + bd + cd$	Distributivity
$= \bar{b}(a + \bar{a}) + bd + cd$	Complementation
$= \bar{b}1 + bd + cd$	Identity
$= \bar{b}(c + \bar{c}) + bd + cd$	Complementation
$= bd + \bar{b}c + cd + \bar{b}\bar{c}$	Distribitivity
$= bd + \bar{b}c + \bar{b}\bar{c}$	Consensus theorem
$= bd + \bar{b}(c + \bar{c})$	Distributivity
$= bd + \bar{b}$	Complement, identity

. . . Understanding Minimization

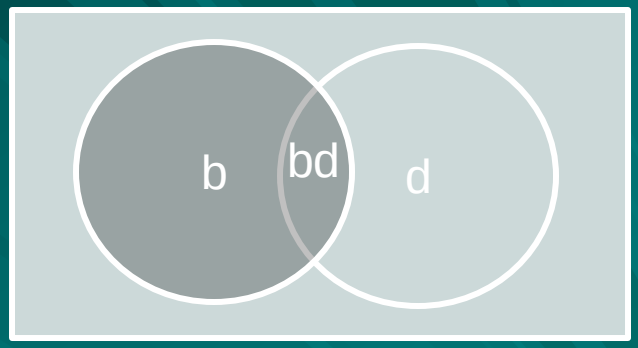
- Reduced SOP: $F = bd + \overline{b}$



. . . Understanding Minimization

- Reducing literals:

$$F = bd + \bar{b}$$
$$= d + \bar{b}$$



Absorption theorem



Exercise: This circuit uses 8 transistors in CMOS technology. Can you redesign it with 6 transistors? (*Hint: Use de Morgan's Theorem*.)

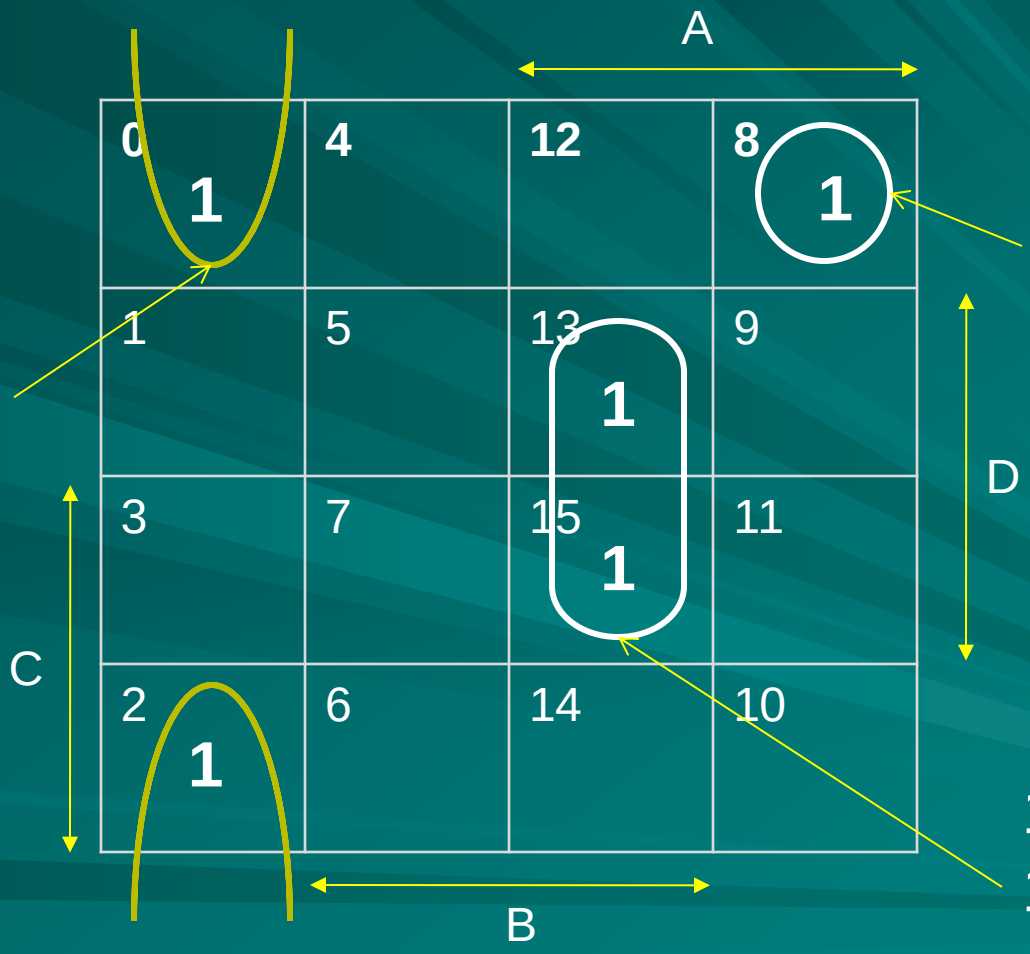
Logic Minimization

- Generally means
  - In SOP form:
    - Minimize number of products (reduce gates) and
    - Minimize literals (reduce gate inputs)
  - In POS form:
    - Minimize number of sums (reduce gates) and
    - Minimize literals (reduce gate inputs)

Product or Implicant or Cube

- Any set of literals ANDed together.
- Minterm is a special case where all variables are present. It is the largest product.
- A minterm is also called a 0-implicant of 0-cube.
- A 1-implicant or 1-cube is a product with one variable eliminated:
  - Obtained by combining two adjacent 0-cubes
  - $ABCD + ABC\,\bar{D} = ABC(D + \bar{D}) = ABC$

Cubes (Implicants) of 4 Variables



What is this?

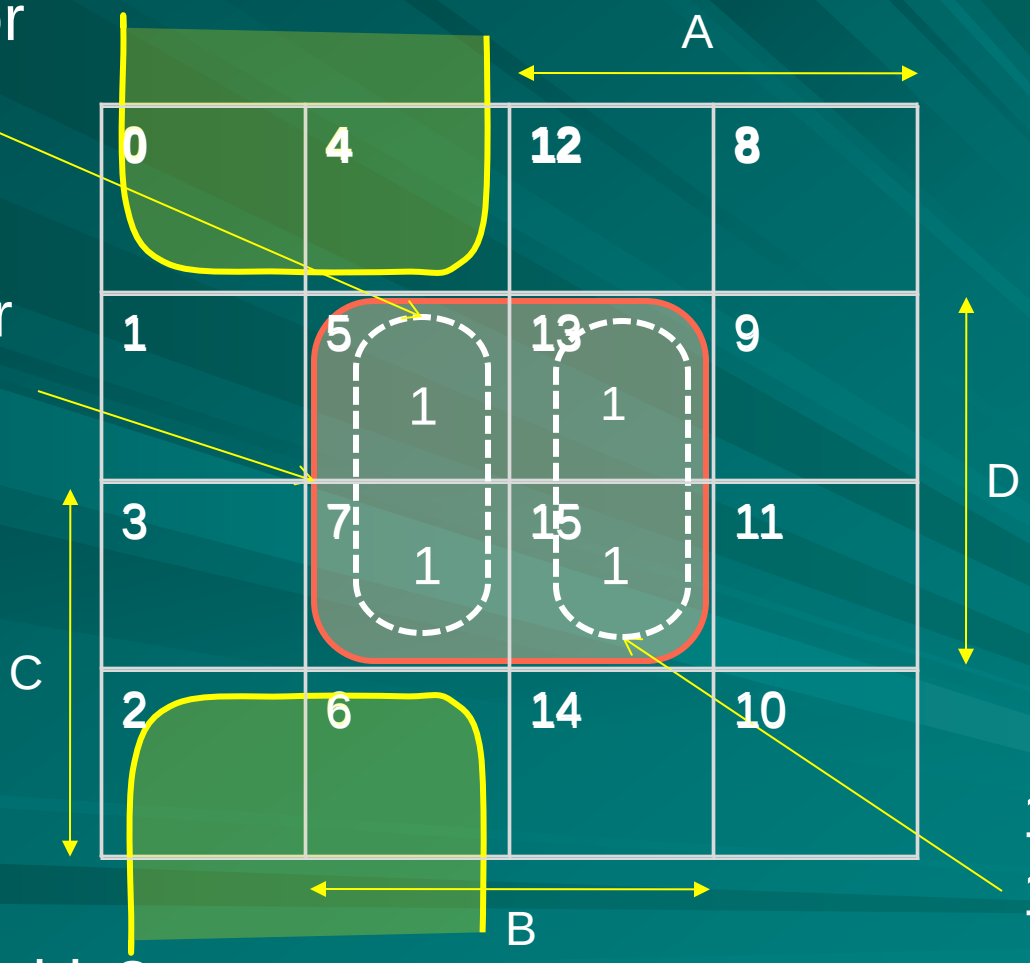
Minterm or
0-implicant or
0-cube
$A\ \overline{B}\ \overline{C}\ \overline{D}$

1-implicant or
1-cube
ABD

Growing Cubes, Reducing Products

1-implicant or
1-cube
$\bar{A}BD$

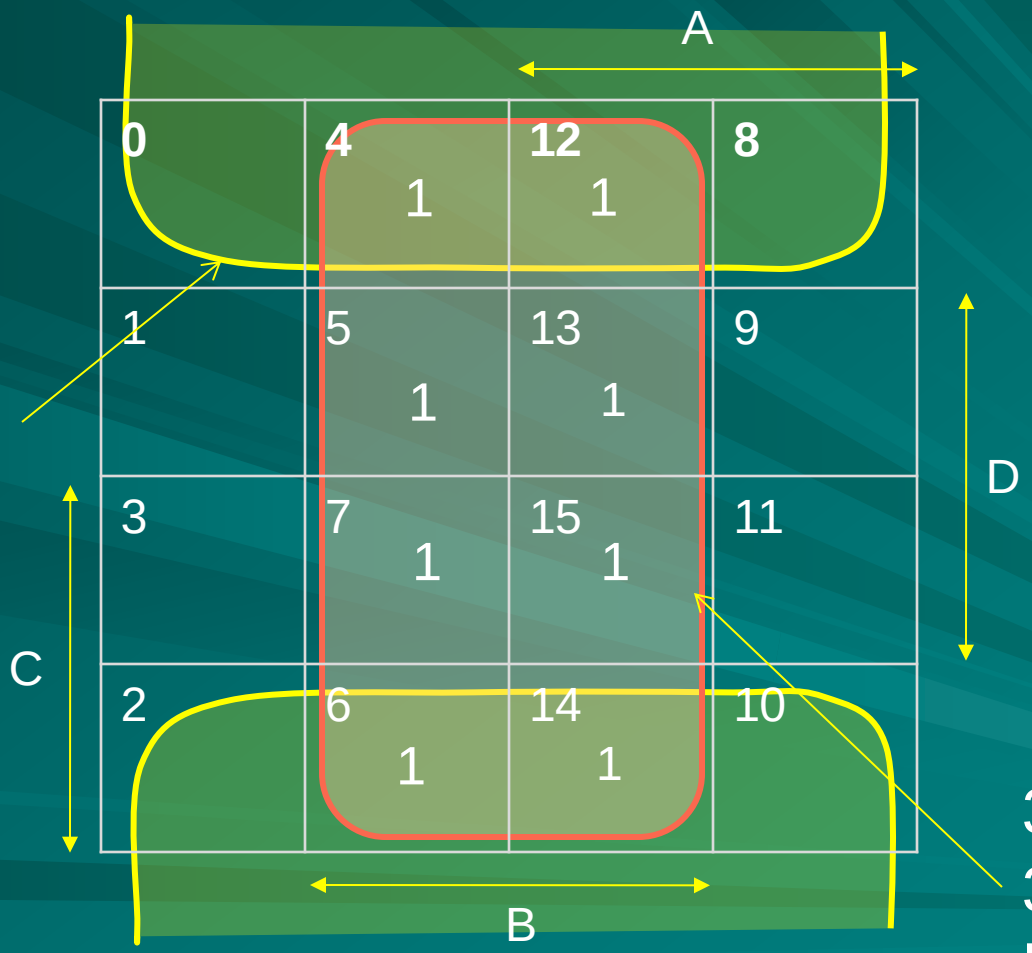
2-implicant or
2-cube
BD



What is this?

1-implicant or
1-cube
ABD

Largest Cubes or Smallest Products



What is this?

3-implicant or
3-cube
B

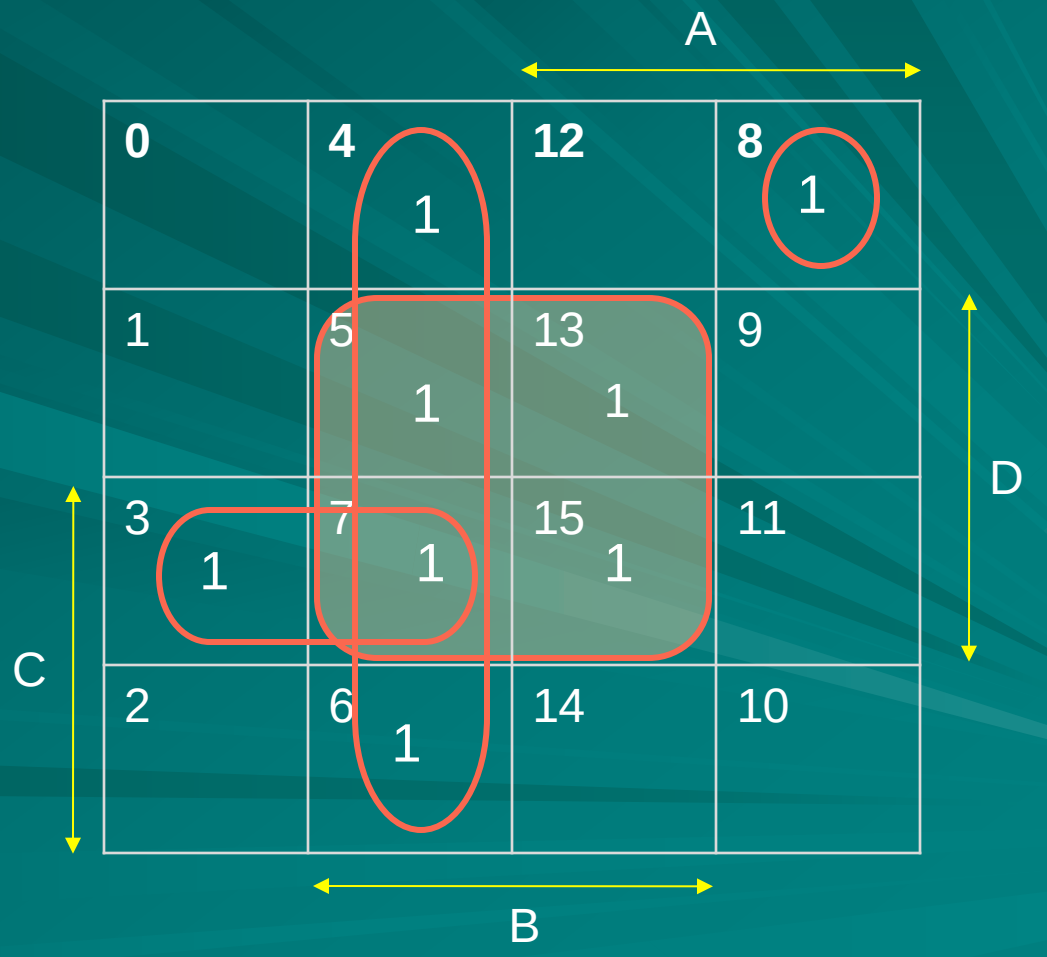
Implication and Covering

- A larger cube covers a smaller cube if all minterms of the smaller cube are included in the larger cube.
- A smaller cube implies (or subsumes) a larger cube if all minterms of the smaller cube are included in the larger cube.

Implicants of a Function

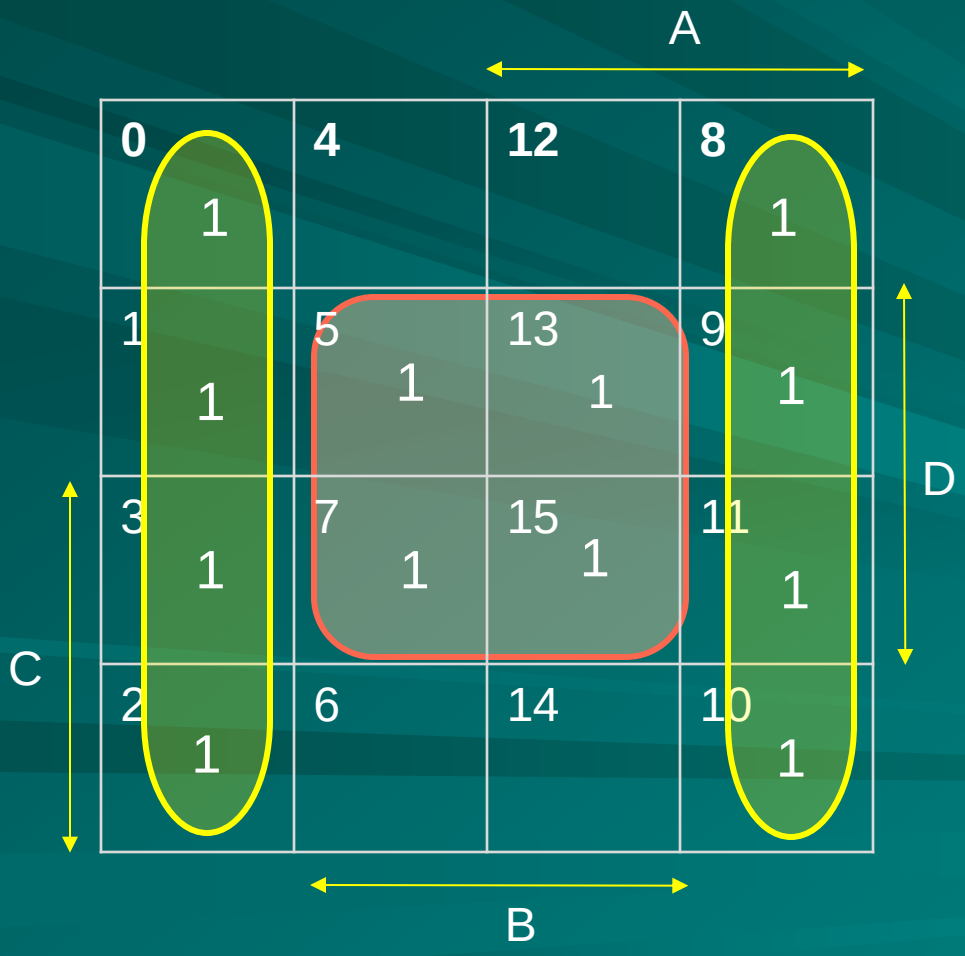
- Minterms, products, cubes that imply the function.

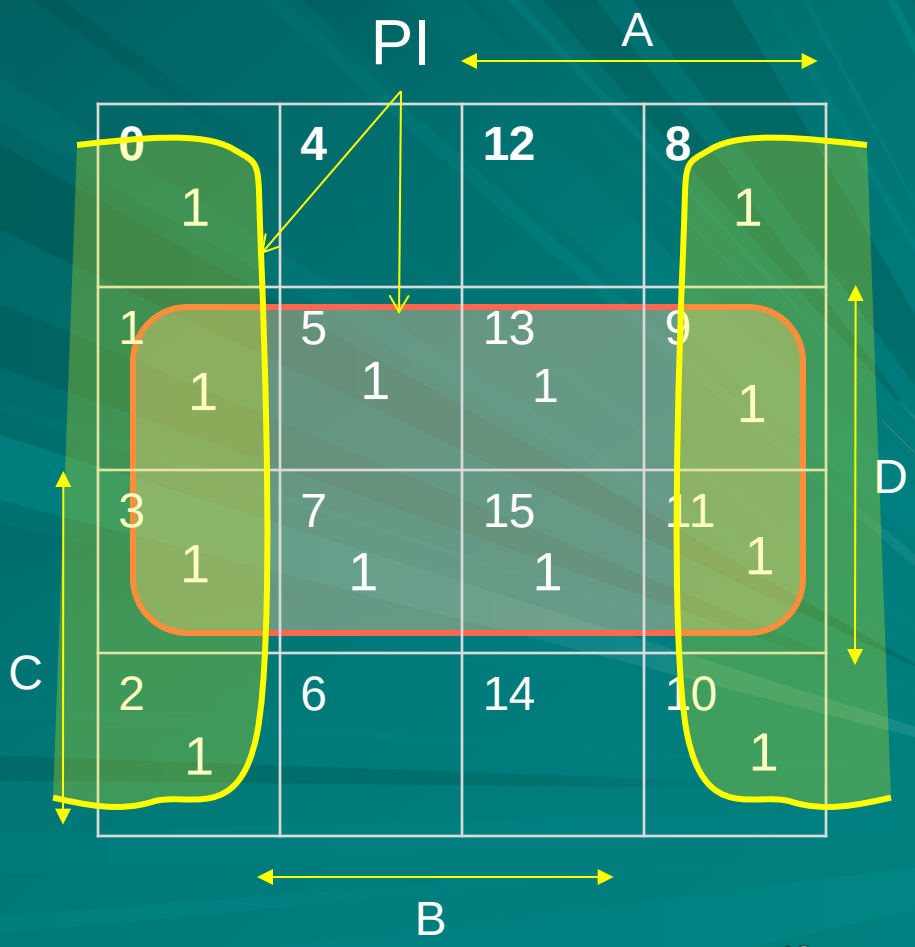
$$F = \overline{A}B + BD + \overline{A}CD + A\overline{B}\overline{C}\overline{D}$$



Prime Implicant (PI)

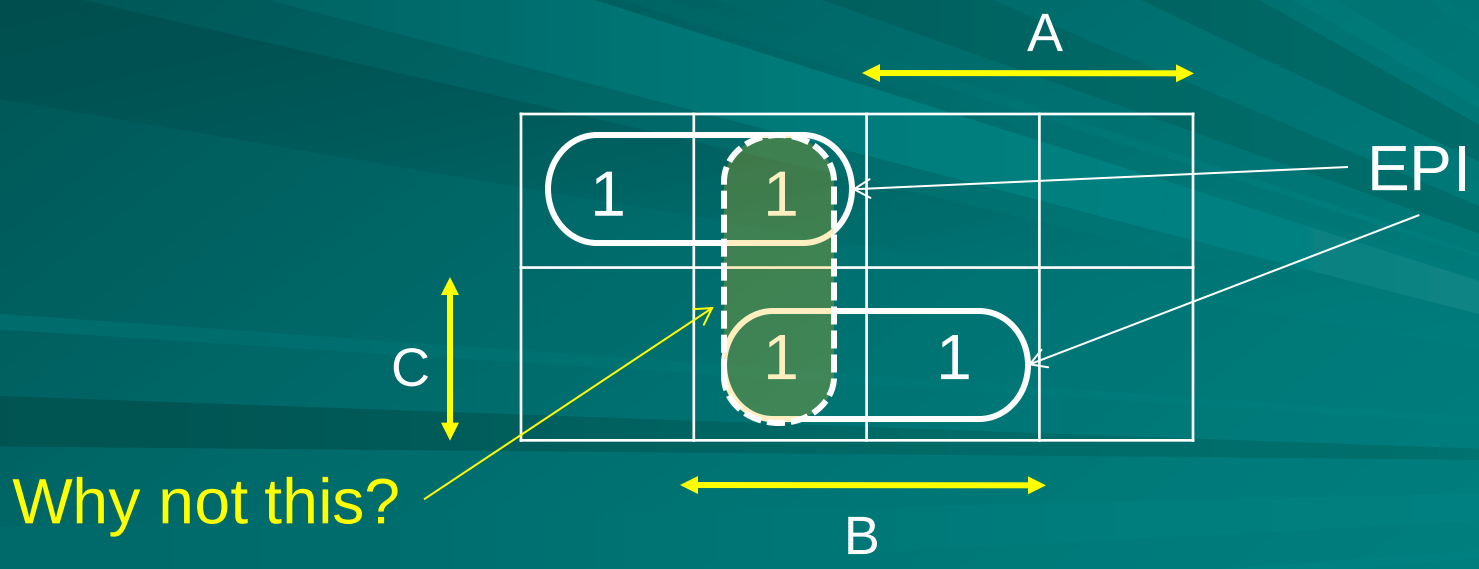
- A cube or implicant of a function that cannot grow larger by expanding into other cubes.





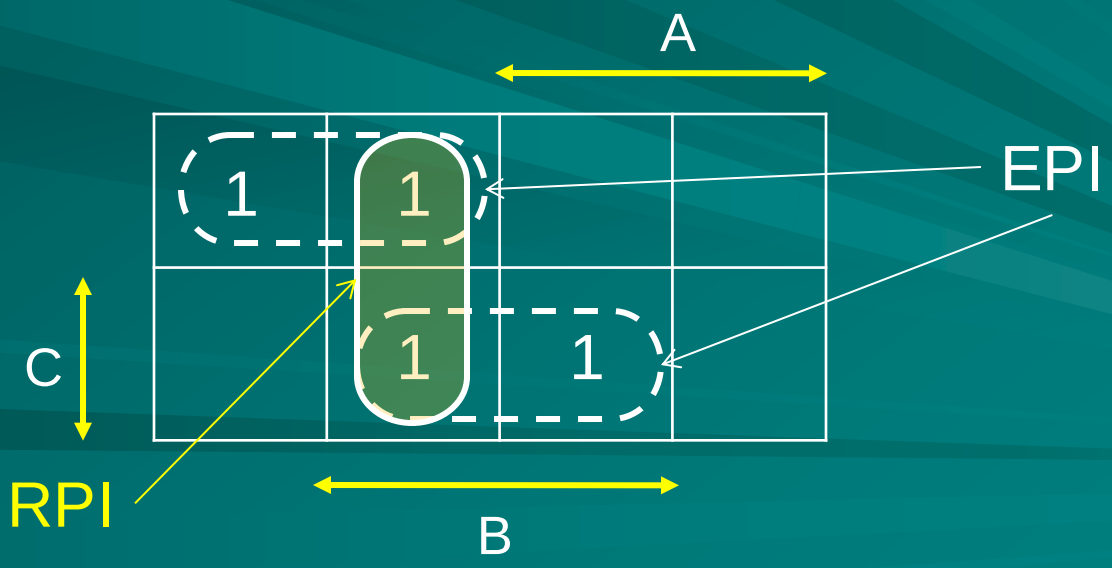
Essential Prime Implicant (EPI)

- If among the minterms subsuming a prime implicant (PI), there is at least one minterm that is covered by this and only this PI, then the PI is called an essential prime implicant (EPI).
- Also called essential prime cube (EPC).



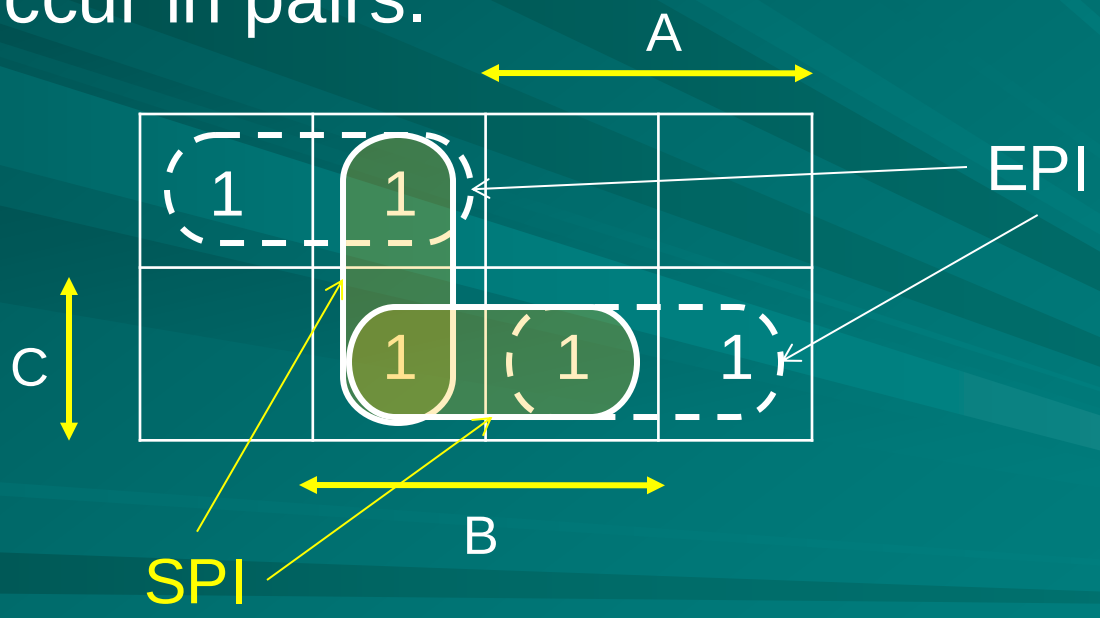
Redundant Prime Implicant (RPI)

- If each minterm subsuming a prime implicant (PI) is also covered by other essential prime implicants, then that PI is called a redundant prime implicant (RPI).
- Also called redundant prime cube (RPC).



Selective Prime Implicant (SPI)

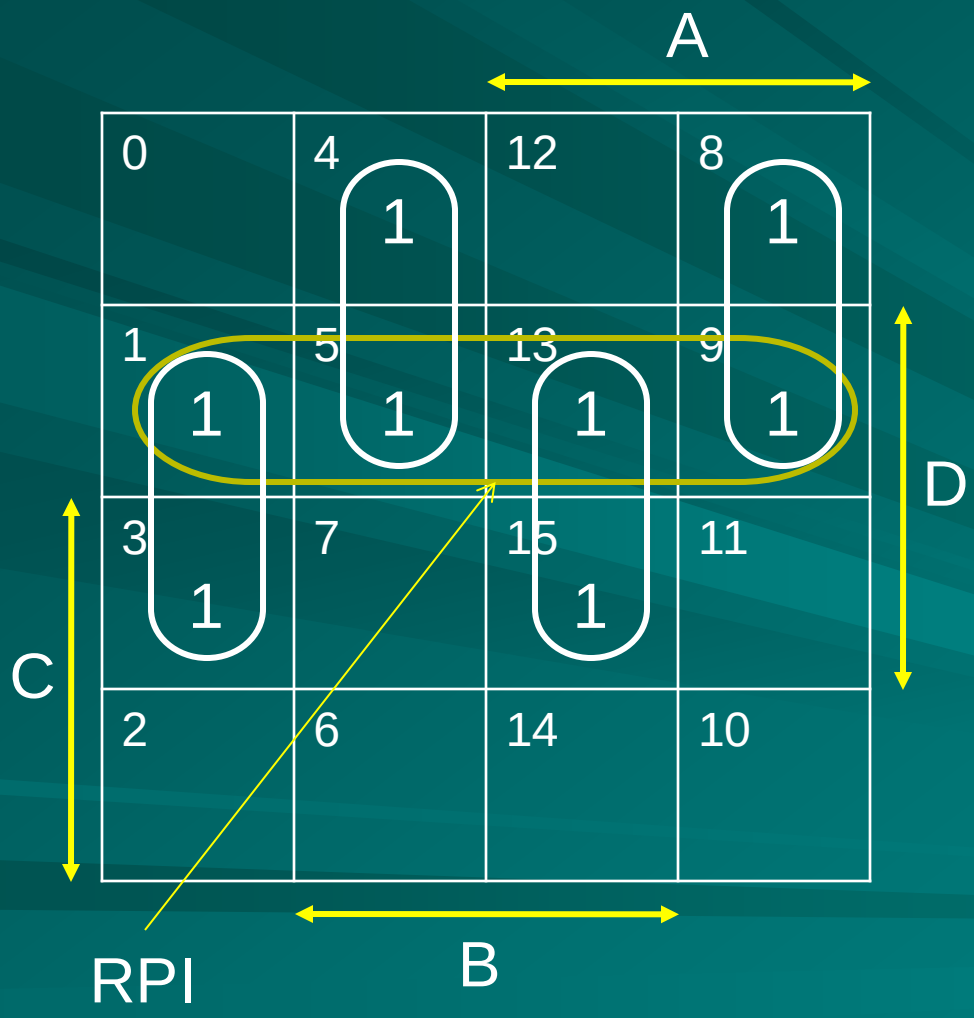
- A prime implicant (PI) that is neither EPI nor RPI is called a selective prime implicant (SPI).
- Also called selective prime cube (SPC).
- SPIs occur in pairs.



Minimum Sum of Products (MSOP)

- Identify all prime implicants (PI) by letting minterms and implicants to grow.
- Construct MSOP with PI only :
  - Cover all minterms
  - Use only essential prime implicants (EPI)
  - Use no redundant prime implicant (RPI)
  - Use cheaper selective prime implicants (SPI)
  - A good heuristic – Choose EPI in ascending order, starting from 0-implicant, then 1-implicant, 2-implicant, . . .

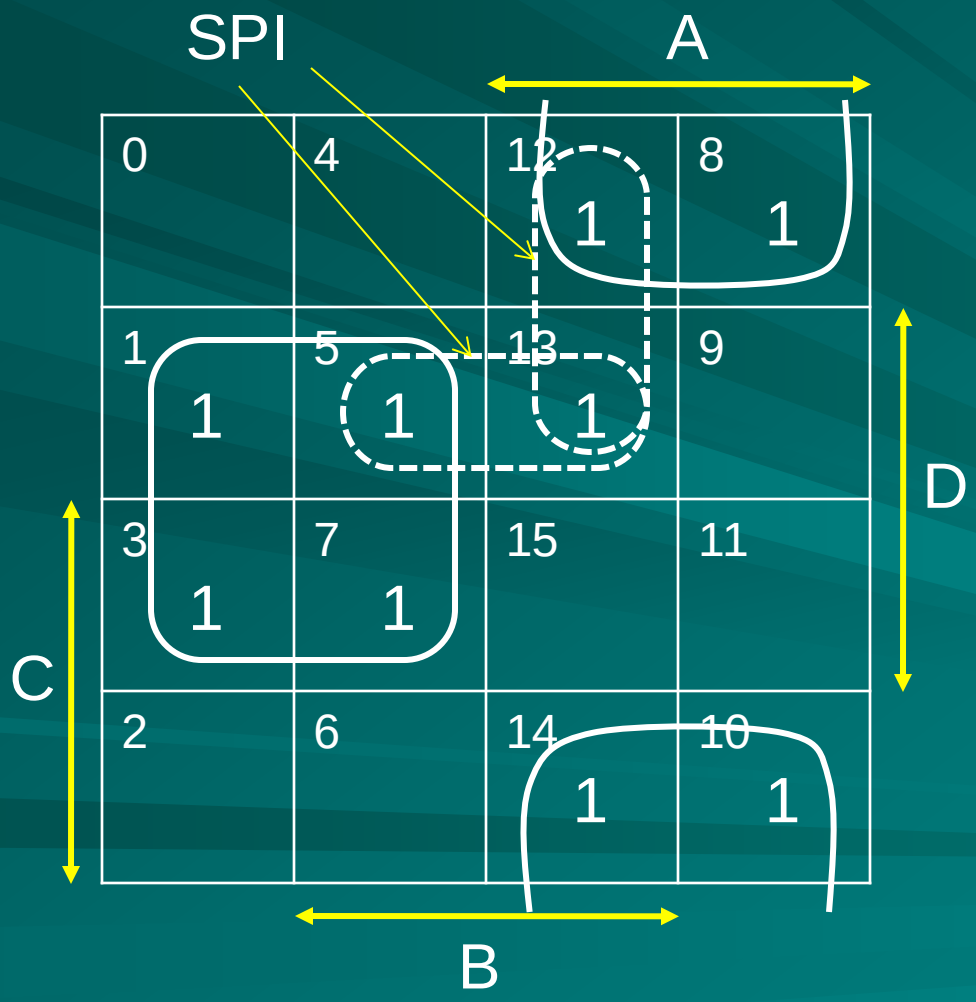
Example: F=∑m(1,3,4,5,8,9,13,15)



MSOP:

$$F = \bar{A}\,\bar{B}\,D + \bar{A}\,B\,\bar{C} + A\,B\,D + A\,\bar{B}\,\bar{C}$$

Example: F=∑m(1,3,5,7,8,10,12,13,14)

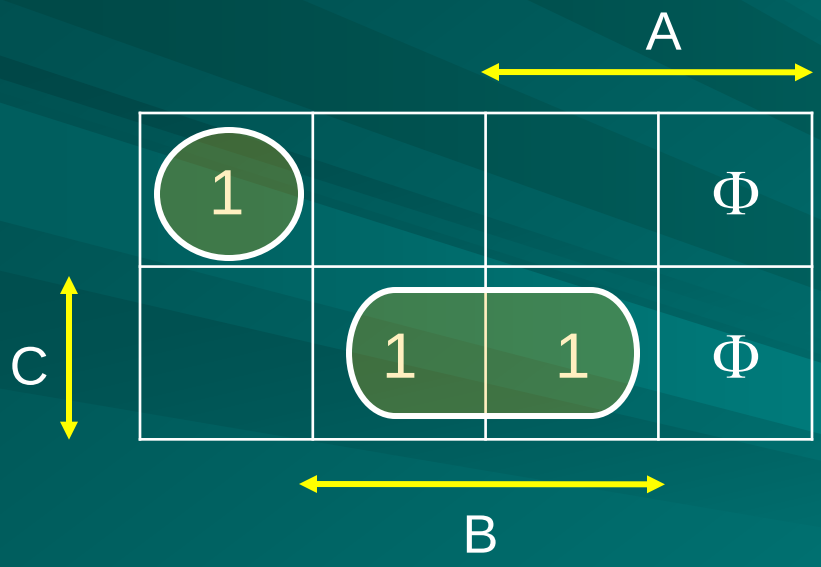


MSOP:

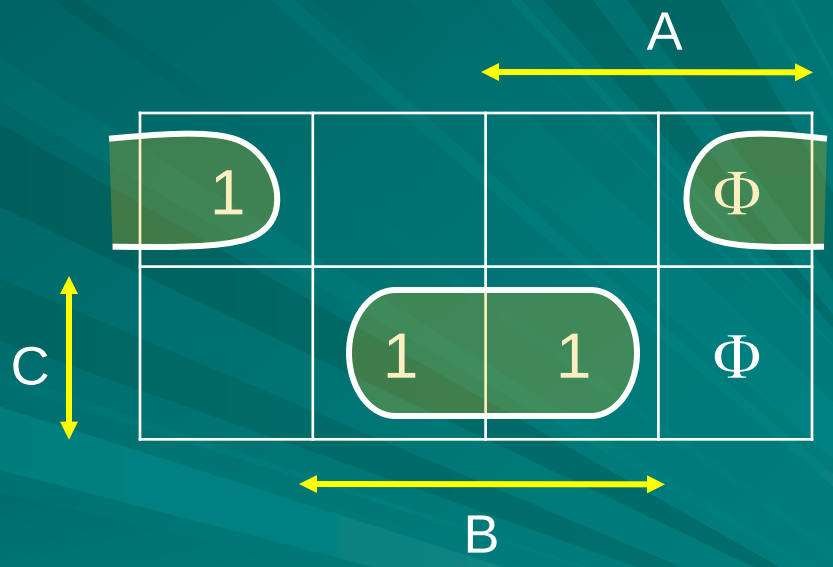
$F = \bar{A}D + A\bar{D} + AB\bar{C}$

Functions with Don't Care Minterms

- $F(A,B,C) = \sum m(0,3,7) + d(4,5)$
- Include don't care minterms when beneficial.



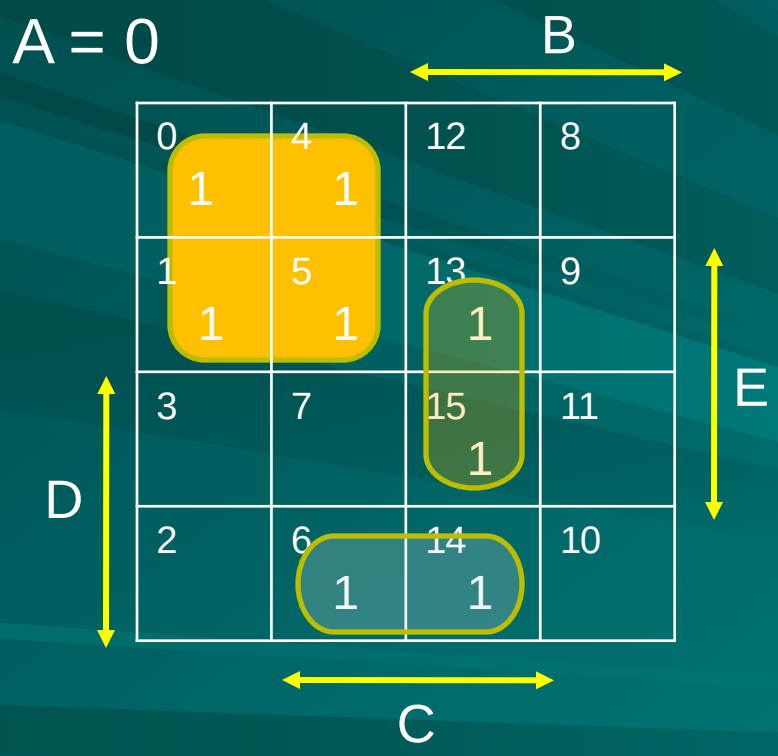
$F = B\,C + \overline{A}\;\overline{B}\;\overline{C}$

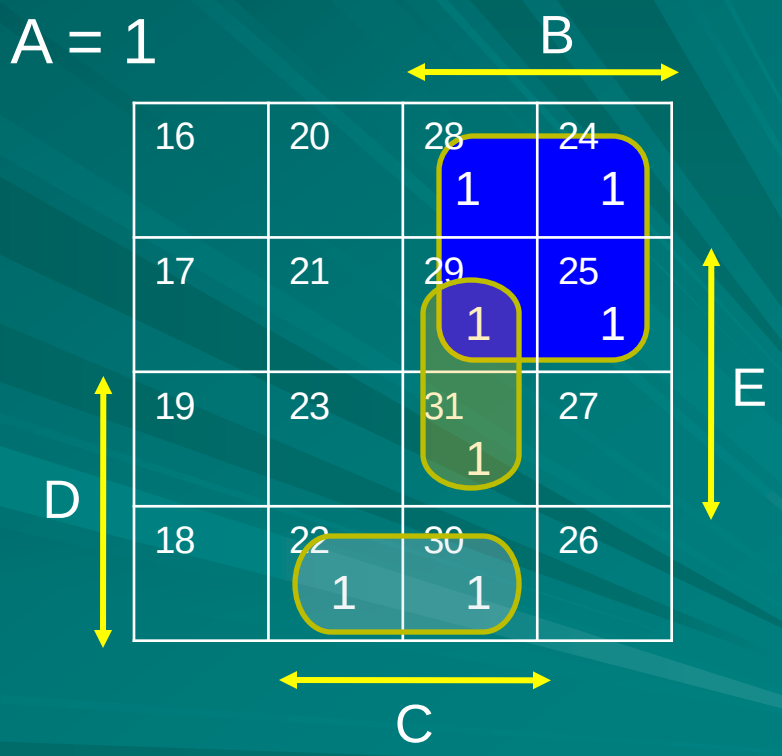


$F = B\,C + \overline{B}\;\overline{C}$

Five-Variable Function

- F(A,B,C,D,E)
  = ∑m(0,1,4,5,6,13,14,15,22,24,25,28,29,30,31)





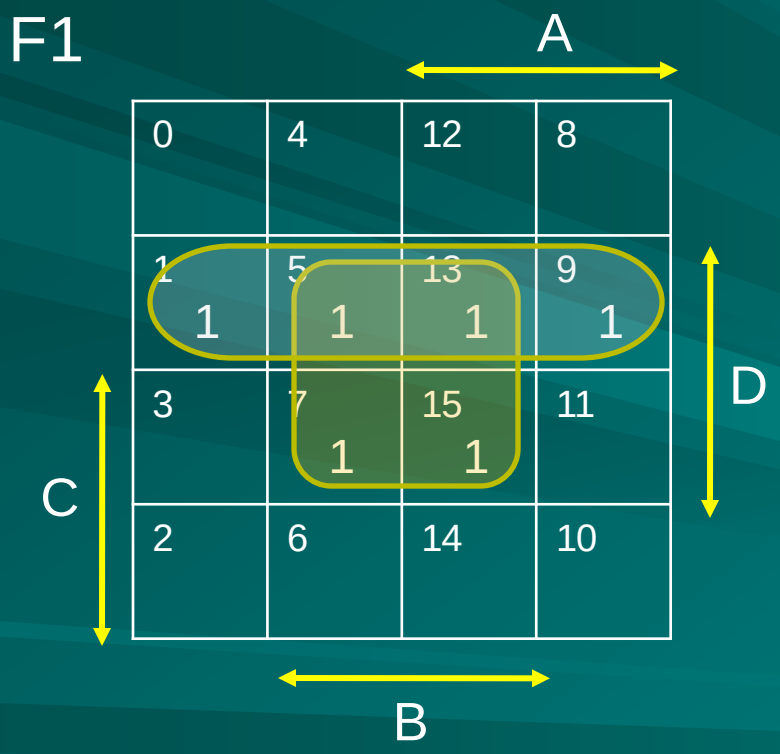
$$F = \bar{A}\,\bar{B}\,\bar{D} + A\,B\,\bar{D} + B\,C\,E + C\,D\,\bar{E}$$

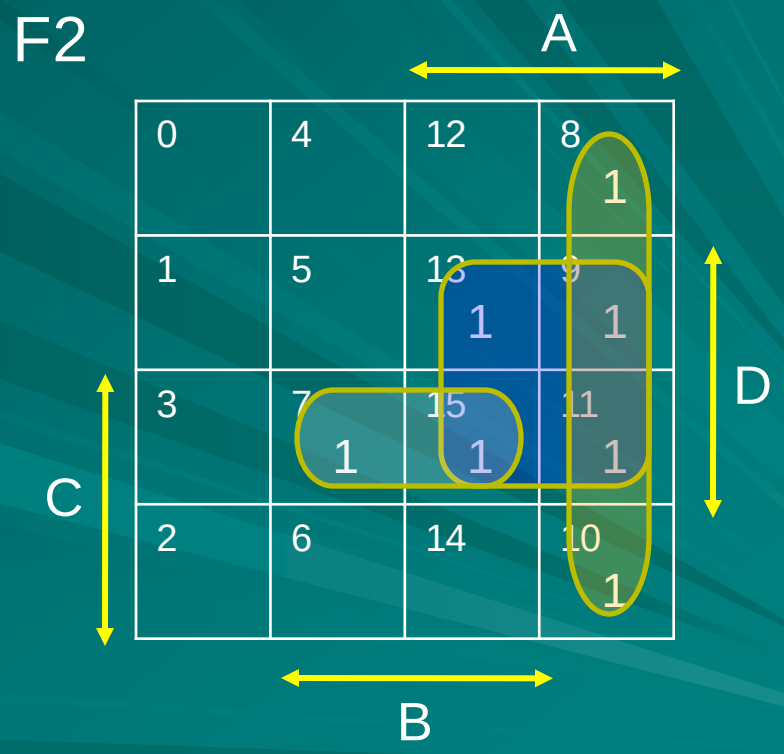
Multiple-Output Minimization

Inputs				Outputs	
A	B	C	D	F1	F2
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	1

Individual Output Minimization

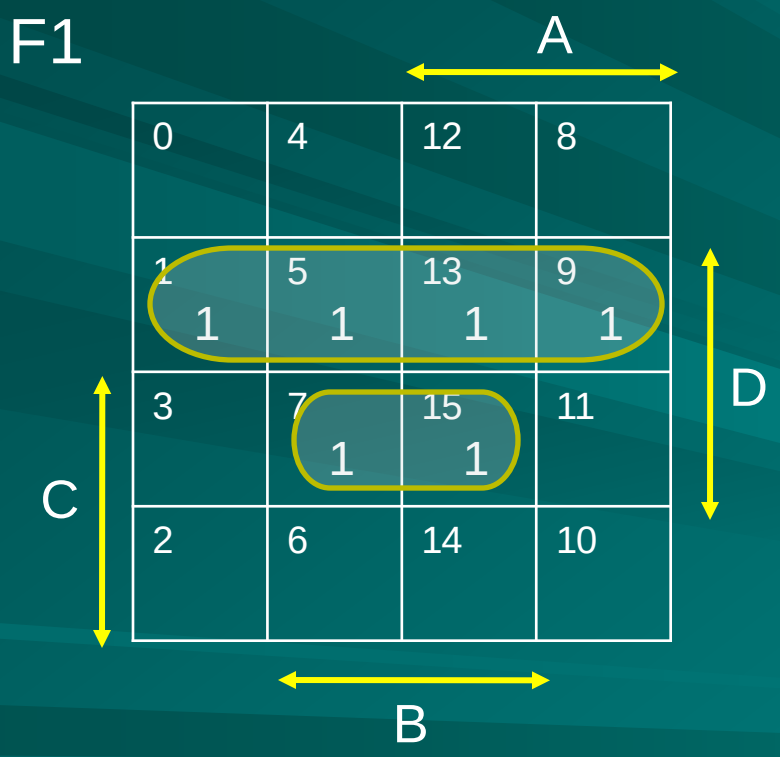
Need five products.

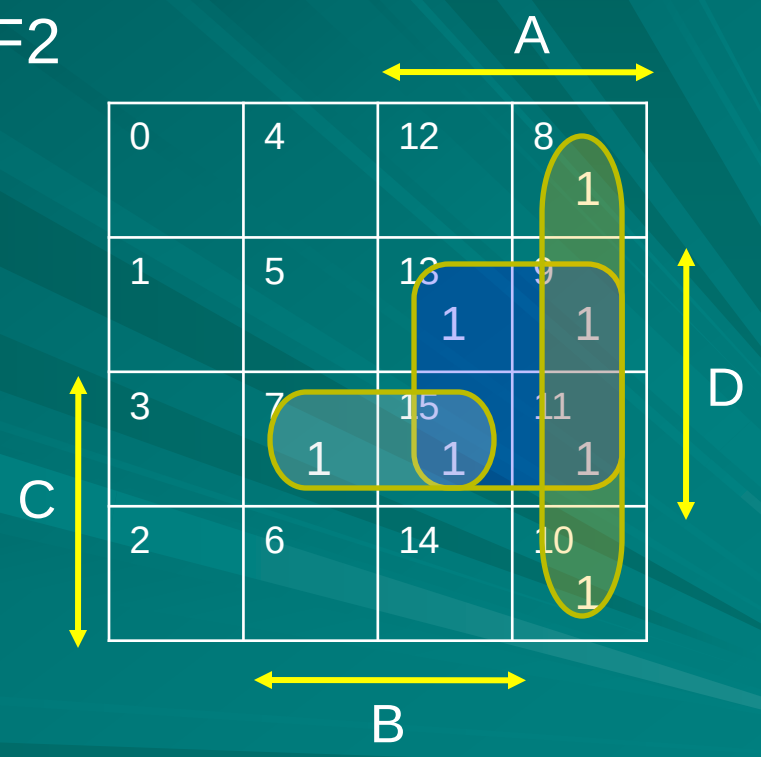




Global Minimization

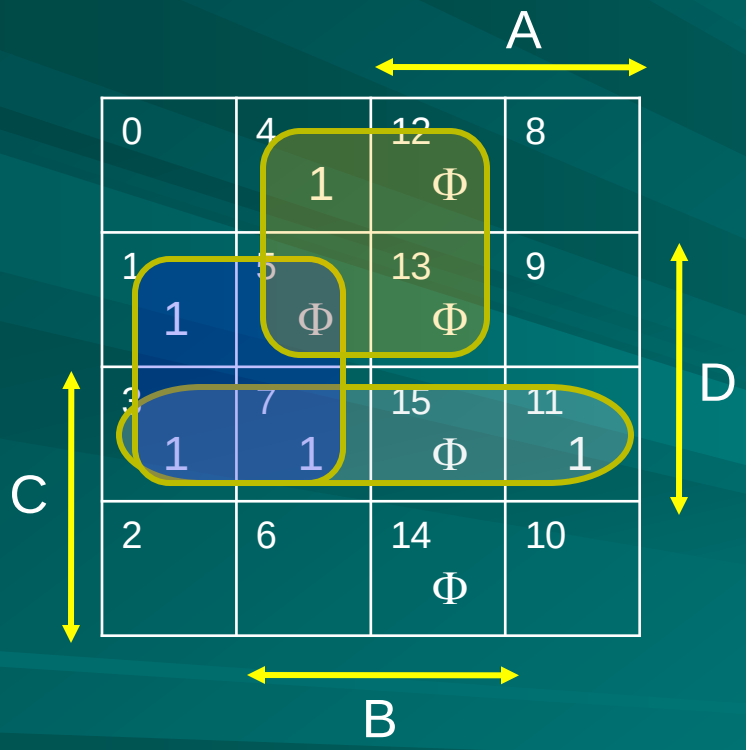
Need four products.



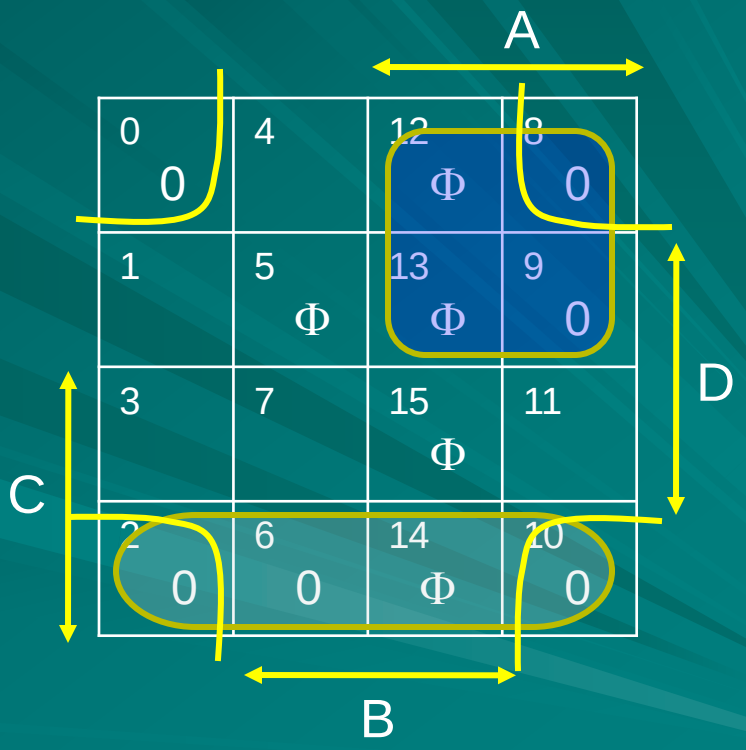


Minimized SOP and POS

- F(A,B,C,D) = $\sum$ m(1,3,4,7,11) + d(5,12,13,14,15)
  = $\prod$ M(0,2,6,8,9,10) D(5,12,13,14,15)



$F = B\,\bar{C} + \bar{A}D + C\,D$



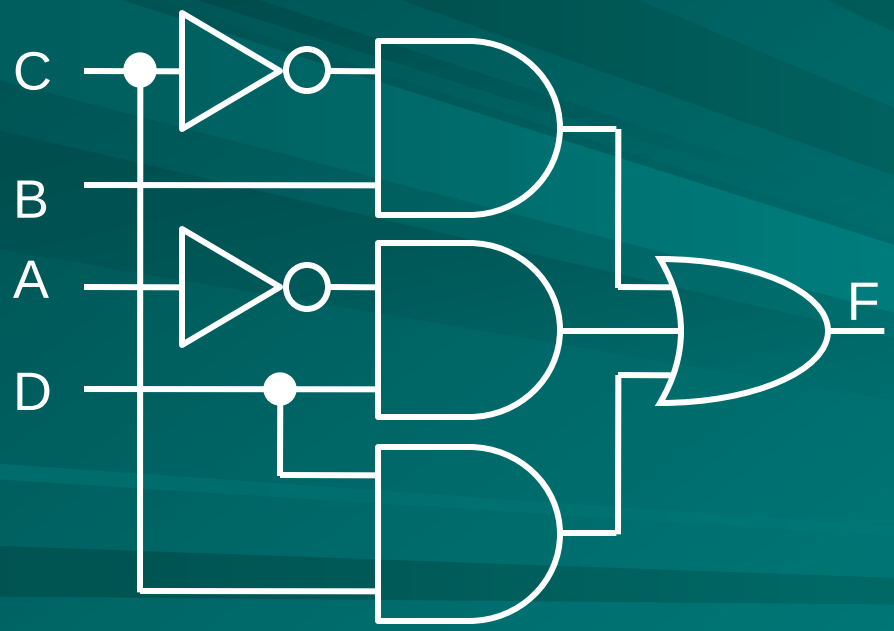
$\bar{F} = \bar{B}\,\bar{D} + C\,\bar{D} + A\,\bar{C}$

$F = (B + D)(\bar{C} + D)(\bar{A} + C)$

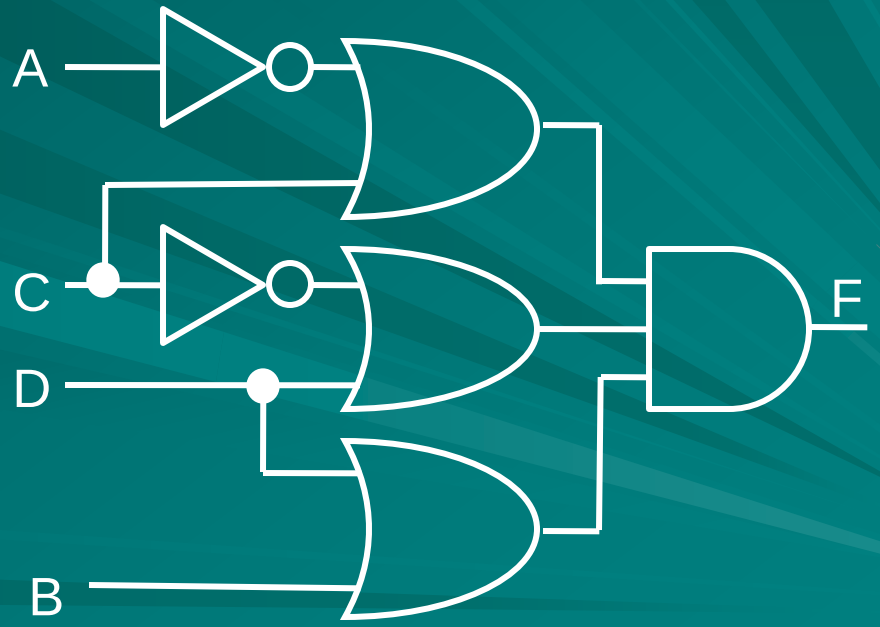
SOP and POS Circuits

- F(A,B,C,D) = $\sum$ m(1,3,4,7,11)+d(5,12,13,14,15)
  = $\prod$ M(0,2,6,8,9,10) D(5,12,13,14,15)

F = B $\bar{C}$ + $\bar{A}$D + C D



F = (B + D)($\bar{C}$ + D)($\bar{A}$ + C)

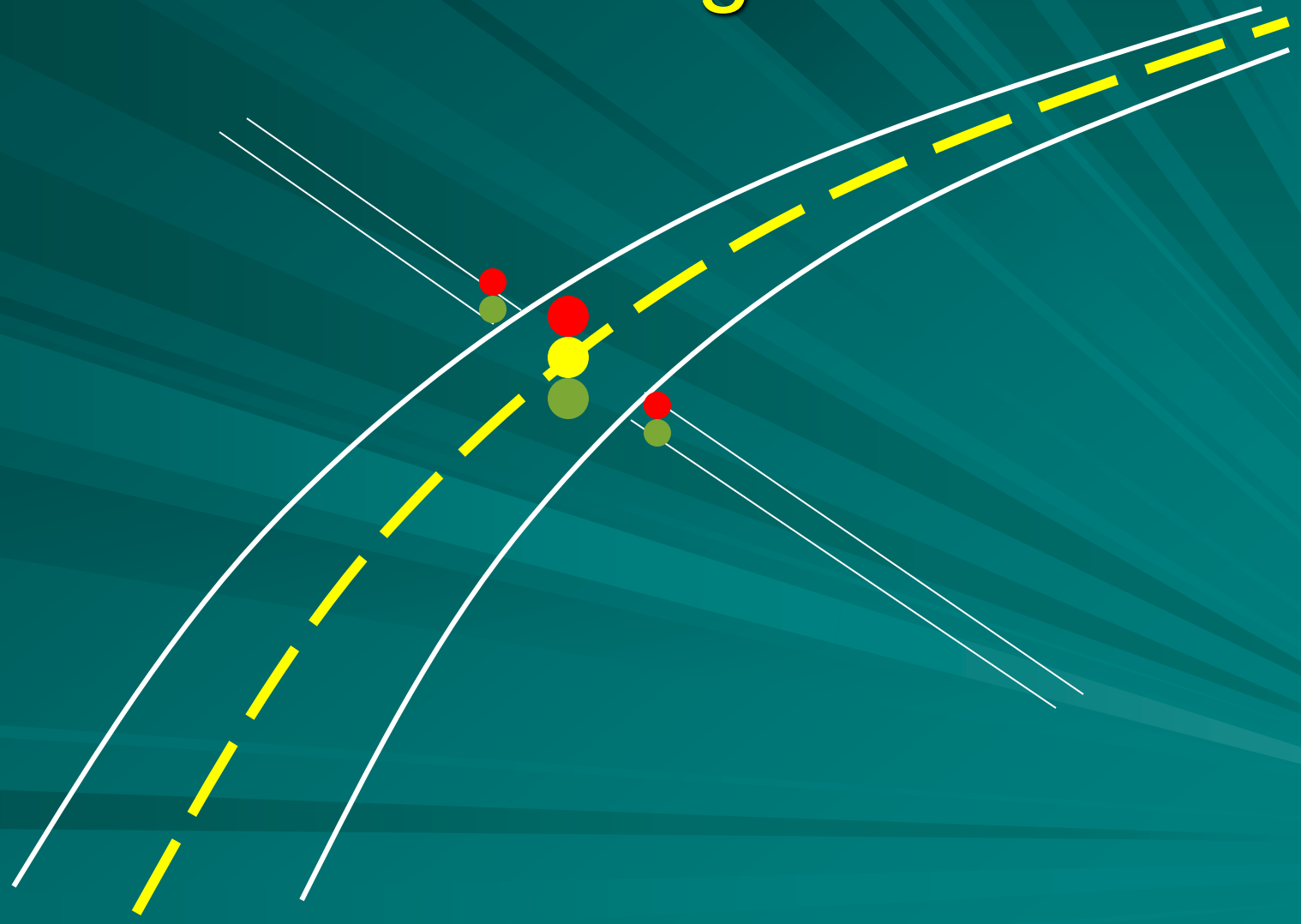


Are two circuits functionally Identical?

How Don’t Cares Occur

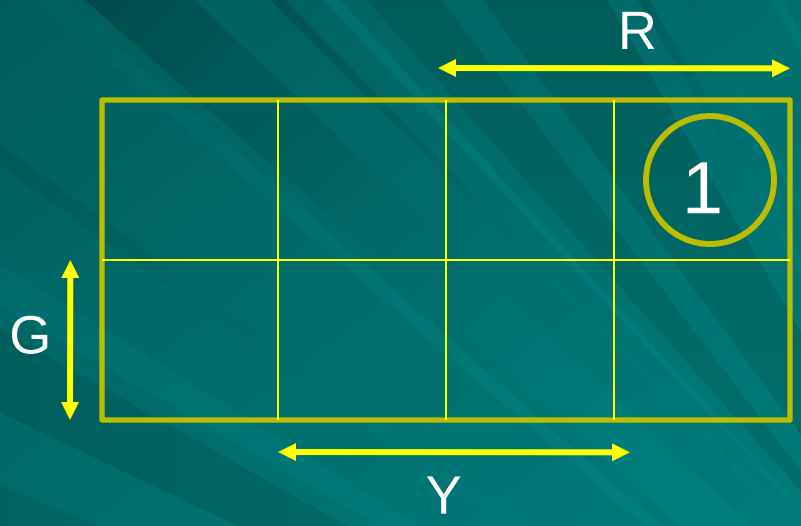
- Consider two roads crossing:
  - Highway with traffic signal, red (R), yellow (Y) or green (G).
  - Rural road with red (r) or green (g) signal.
- Here R, Y, G, r and g are Boolean variables; a 1 implies light is on, 0 means light is off.
- Highway signals R, Y and G are controlled by a computer; only one light can be on.
- We only need to devise a digital circuit to obtain g, because r = $\overline{g}$.

Traffic Signals

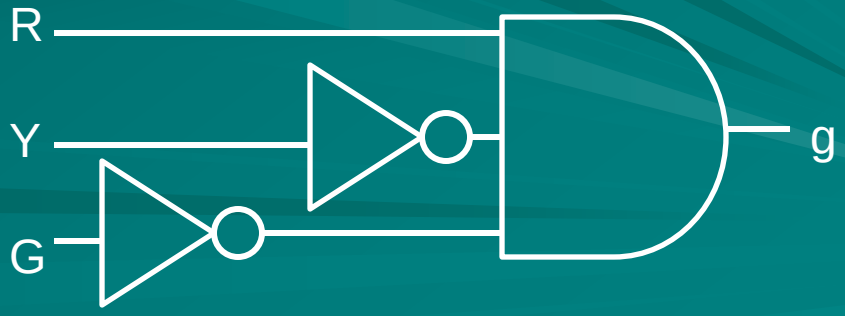


Completely Specified Function

Truth Table				
minterm	R	Y	G	g
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

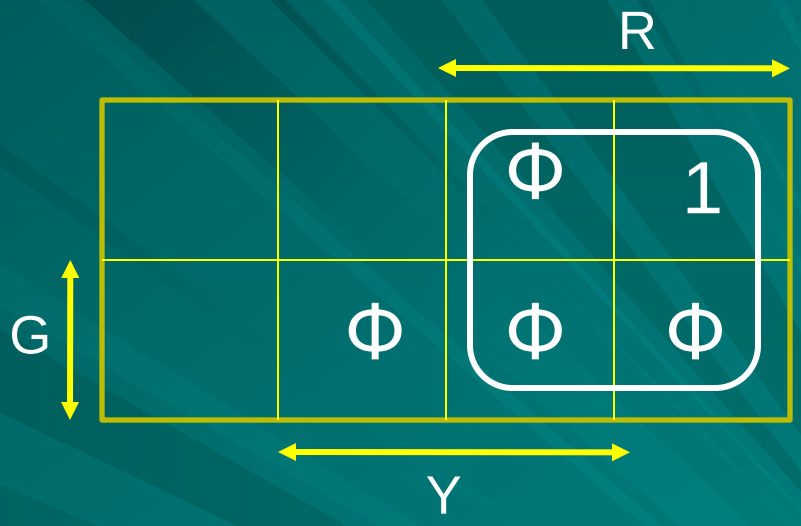


$$g = R\,\bar{Y}\,\bar{G}$$



Incompletely Specified Function

Truth Table				
minterm	R	Y	G	g
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	Φ
4	1	0	0	1
5	1	0	1	Φ
6	1	1	0	Φ
7	1	1	1	Φ

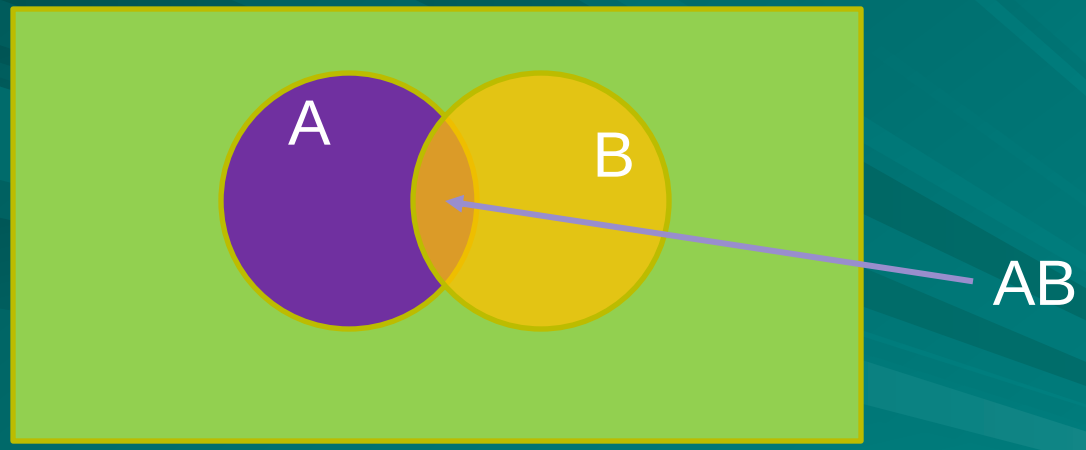


$g = R$



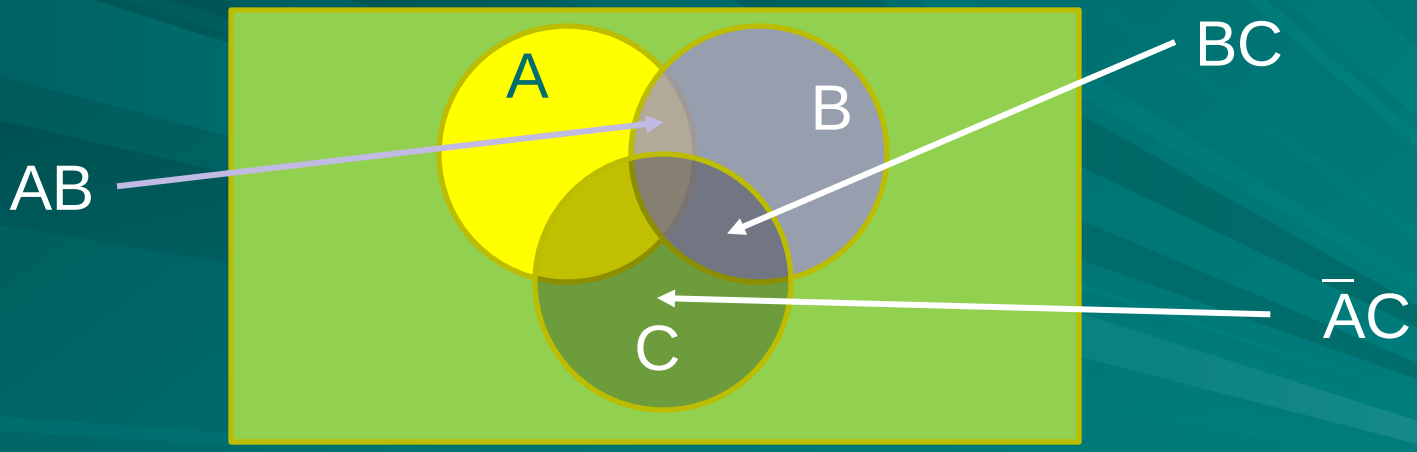
Absorption Theorem

- For two Boolean variables: A, B
- A + A B = A
- Proof:



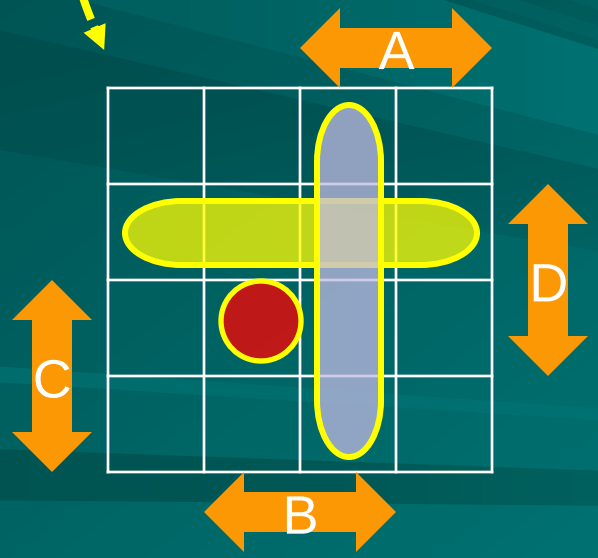
Consensus Theorem

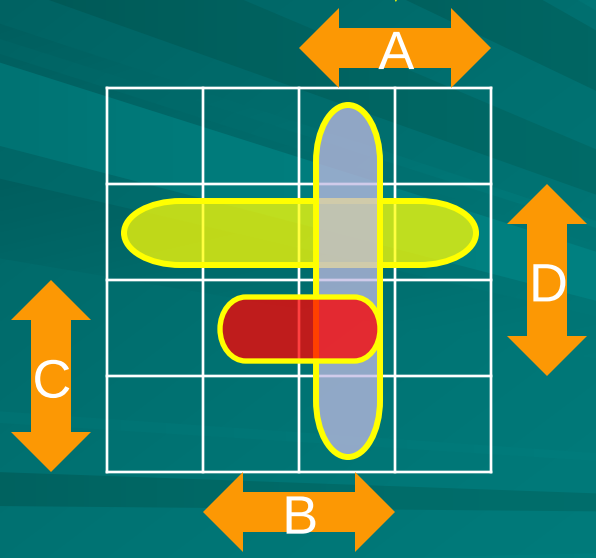
- For three Boolean variables: A, B, C
- $A\,B + \bar{A}\,C + B\,C = A\,B + \bar{A}\,C$
- Proof:

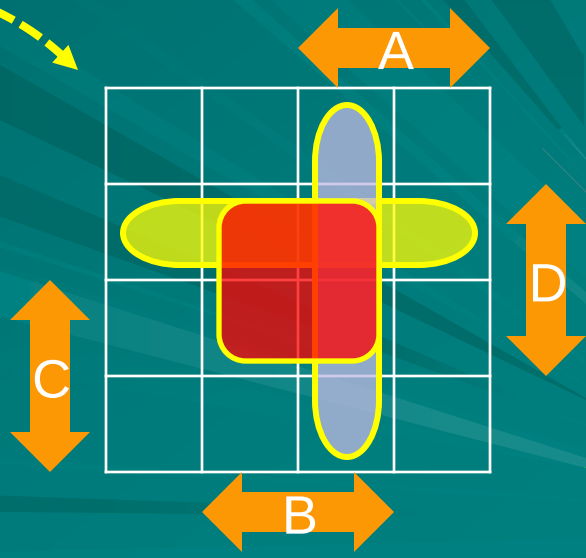


Growing Implicants to PI

- F = $AB + \bar{C}D + \bar{A}BCD$ *initial implicants*
  = $AB + \bar{A}BCD + BCD + \bar{C}D$ *consensus th.*
  = $AB + BCD + \bar{C}D$ *absorption th.*
  = $AB + BCD + \bar{C}D + BD$ *consensus th.*
  = $AB + \bar{C}D + BD$ *absorption th.*







Identifying EPI

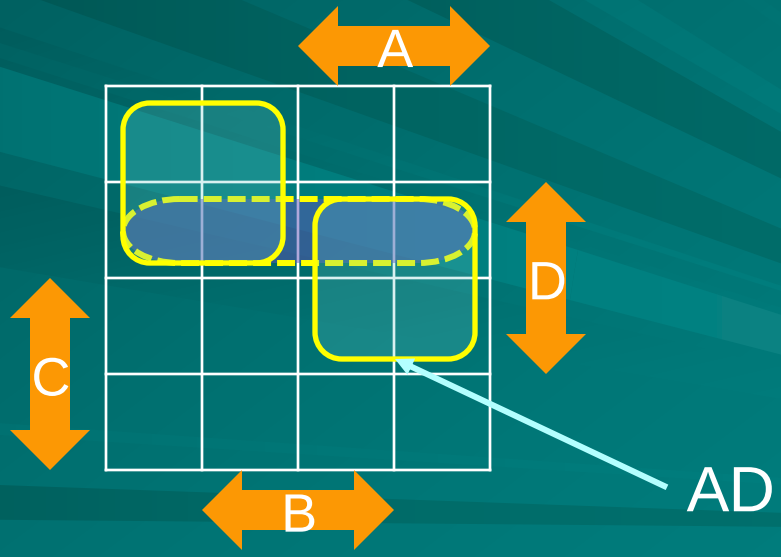
- Find all prime implicants.
- From prime implicant SOP, remove a PI.
- Apply consensus theorem to the remaining SOP.
- If the removed PI is generated, then it is either an RPI or an SPI.
- If the removed PI is not generated, then it is an EPI

Example

- PI SOP: F = $AD + \bar{A}\,\bar{C} + \bar{C}D$
- Is AD an EPI?

  F – {AD} = $\bar{A}\,\bar{C} + \bar{C}D$, no new PI can be generated

  Hence, AD is an EPI.



Example (Cont.)

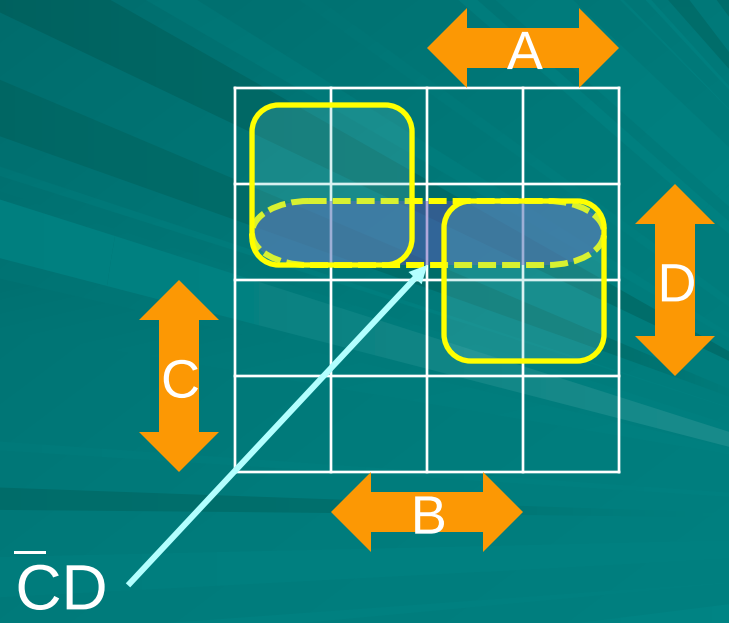
- PI SOP: F = $A\,D + \bar{A}\,\bar{C} + \bar{C}D$
- Is $\bar{C}D$ an EPI?

$F - \{\ \bar{C}D\ \} = A\,D + \bar{A}\,\bar{C}$

$= A\,D + \bar{A}\,\bar{C} + \bar{C}\,D$ (Consensus theorem)

Hence $\bar{C}\,D$ is not an EPI

(it is an RPI)



Finding MSOP

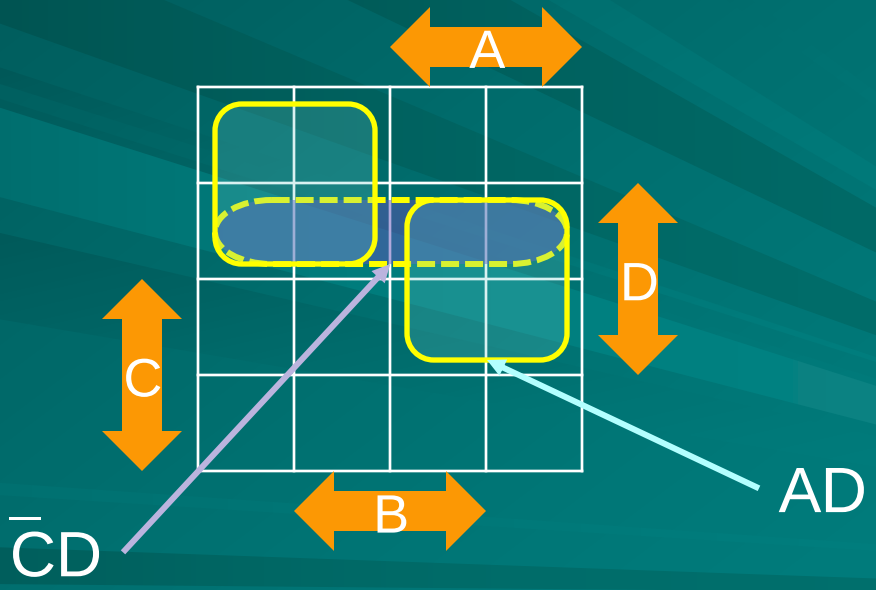
1. Start with minterm or cube SOP representation of Boolean function.
2. Find all prime implicants (PI).
3. Include all EPI's in MSOP.
4. Find the set of uncovered minterms, {UC}.
5. MSOP is minimum if {UC} is empty. *DONE.*
6. For a minterm in {UC}, include the largest PI from remaining PI's (non-EPI's) in MSOP.
7. Go to step 4.

Finding Uncovered Minterms, {UC}

- {UC} = ({PI} # {PMSOP}) # {DC}
- Where:
  - {PI} is set of all prime implicants of the function.
  - {PMSOP} is any partial SOP.
  - {DC} is set of don't care minterms.
  - Sharp (#) operation between Boolean expressions X and Y, X # Y, is the set of minterms covered by X that are not covered by Y.

Example: # (Sharp) Operation

- AD # $\bar{C}D$ = {A B C D, A $\bar{B}$ C D}
- $\bar{C}D$ # AD = { $\bar{A}$ B $\bar{C}$ D, $\bar{A}$ $\bar{B}$ $\bar{C}$ D}



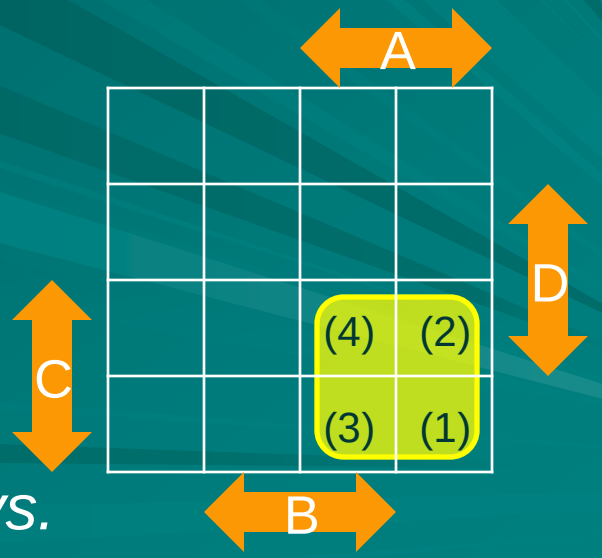
Minterms Covered by a Product

- A product from which k variables have been eliminated, covers 2^k minterms.
- Example: For four variables, A, B, C, D

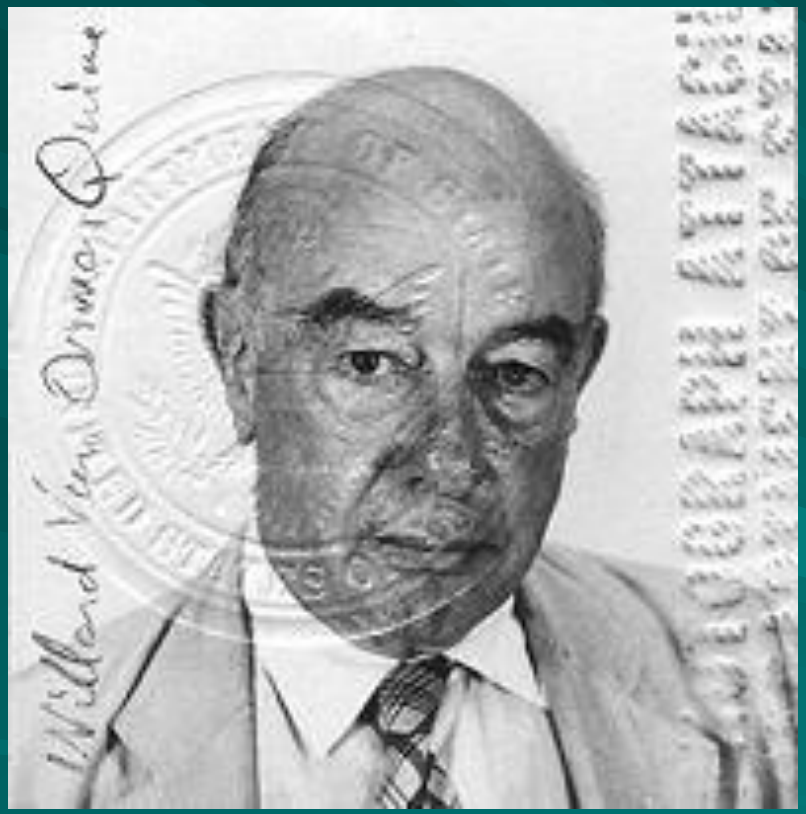
  Product AC covers $2^2 = 4$ minterms:

  1) A $\overline{B}$ C $\overline{D}$
  2) A $\overline{B}$ C D
  3) A B C $\overline{D}$
  4) A B C D

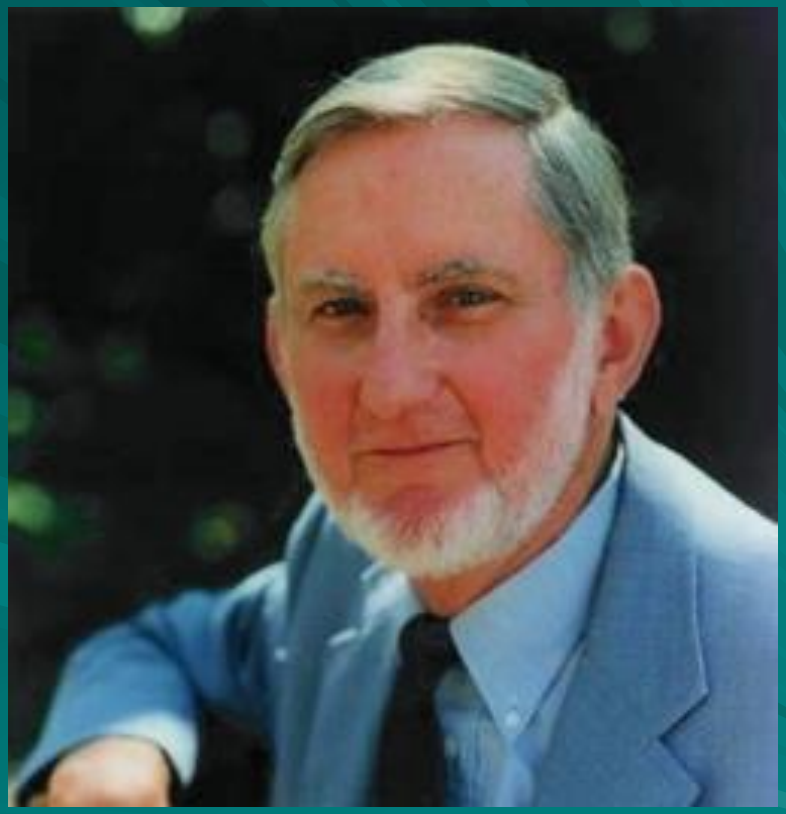
Obtained by inserting the eliminated variables in all possible ways.



Quine-McCluskey



Willard V. O. Quine
1908 – 2000



Edward J. McCluskey
b. 1929

Quine-McCluskey Tabular Minimization Method

- W. V. Quine, “The Problem of Simplifying Truth Functions,” *American Mathematical Monthly*, vol. 59, no. 10, pp. 521-531, October 1952.
- E. J. McCluskey, “Minimization of Boolean Functions,” *Bell System Technical Journa*l, vol. 35, no. 11, pp. 1417-1444, November 1956.
- Textbook, Section 3.9, pp. 211-225.

Q-M Tabular Minimization

- Minimizes functions with many variables.
- Begin with minterms:
  - Step 1: Tabulate minterms in groups of increasing number of true variables.
  - Step 2: Conduct linear searches to identify all prime implicants (PI).
  - Step 3: Tabulate PI's vs. minterms to identify EPI's.
  - Step 4: Tabulate non-essential PI's vs. minterms not covered by EPI's. *Select* minimum number of PI's to cover all minterms.
- MSOP contains all EPI's and *selected* non-EPI's.

$F(A,B,C,D) = \sum m(2,4,6,8,9,10,12,13,15)$

- Q-M Step 1: Group minterms with 1 true variable, 2 true variables, etc.

Minterm	ABCD	Groups
2	0010	1: single 1
4	0100	
8	1000	
6	0110	2: two 1's
9	1001	
10	1010	
12	1100	
13	1101	3: three 1's
15	1111	4: four 1's

Q-M Step 2

- Find all implicants by combining minterms, and then combining products that differ in a single variable: For example,
  - 2 and 6, or $\overline{A}\ \overline{B}\ C\ \overline{D}$ and $\overline{A}\ B\ C\ \overline{D} \rightarrow \overline{A}\ C\ \overline{D}$, written as 0 – 1 0.
- Try combining a minterm (or product) with all minterms (or products) listed below in the table.
- Include resulting products in the next list.
- If minterm (or product) does not combine with any other, mark it as PI.
- Check the minterm (or product) and repeat for all other minterms (or products).

Step 2 Executed on Example

List 1			List 2			List 3		
Minterm	ABCD	PI?	Minterms	ABCD	PI?	Minterms	ABCD	PI?
2	0010	X	2, 6	0-10	PI2	8,9,12,13	1-0-	PI1
4	0100	X	2,10	-010	PI3			
8	1000	X	4,6	01-0	PI4			
6	0110	X	4,12	-100	PI5			
9	1001	X	8,9	100-	X			
10	1010	X	8,10	10-0	PI6			
12	1100	X	8,12	1-00	X			
13	1101	X	9,13	1-01	X			
15	1111	X	12,13	110-	X			
			13,15	11-1	PI7			

Step 3: Identify EPI's

Covered by EPI →				x	x		x	x	x
Minterms →	2	4	6	8	9	10	12	13	15
PI1 is EPI				x	x		x	x	
PI2	x		x						
PI3	x					x			
PI4		x	x						
PI5		x					x		
PI6				x		x			
PI7 is EPI								x	x

Step 4: Cover Remaining Minterms

Remaining minterms →	2	4	6	10
PI2	x		x	
PI3	x			x
PI4		x	x	
PI5		x		
PI6				x

Integer linear program (ILP), available from Matlab and other sources: Define integer {0,1} variables, xk = 1, select PIk; xk = 0, do not select PIk.

Minimize $\sum_k$ xk, subject to constraints: x2 + x3 ≥ 1

x4 + x5 ≥ 1

x2 + x4 ≥ 1

x3 + x6 ≥ 1

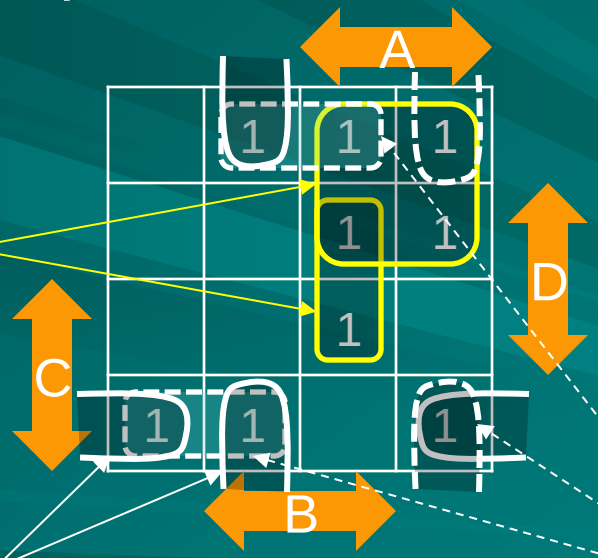
A solution is x3 = x4 = 1, x2 = x5 = x6 = 0

Linear Programming (LP)

- A mathematical optimization method for problems where some “cost” depends on a large number of variables.
- An easy to understand introduction is:
  - S. I. Gass, *An Illustrated Guide to Linear Programming*, New York: Dover Publications, 1970.
- Very useful tool for a variety of engineering design problems.
- Available in software packages like Matlab.
- Courses on linear programming are available in Math, Business and Engineering departments.

Q-M MSOP Solution and Verification

- F(A,B,C,D) = PI1 + PI3 + PI4 + PI7
  = 1-0- + -010 + 01-0 + 11-1
  = A $\bar{C}$ + $\bar{B}$ C $\bar{D}$ + $\bar{A}$ B $\bar{D}$ + A B D
- See Karnaugh map.

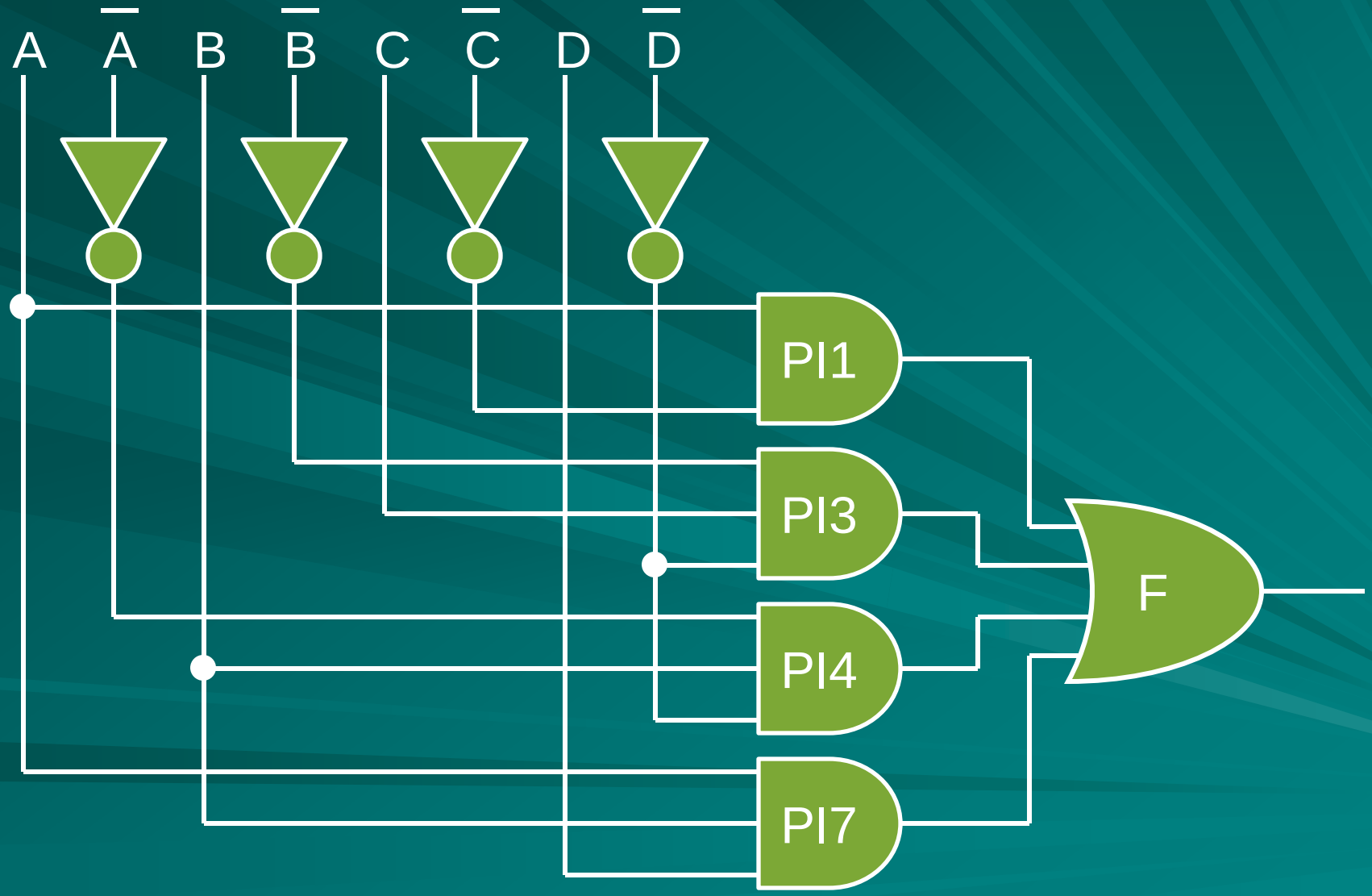


EPI's in MSOP

Non-EPI's in MSOP

Non-EPI's not in MSOP

Minimized Circuit



QM Minimizer on the Web

- http://www.mathcs.bethel.edu/~gossett/DiscreteMathWithProof/QuineMcCluskey.html
- Note: This minimizer may not always select the best from the selective prime implicants.

Further Reading

- Incompletely specified functions: See Example 3.25, pages 218-220.
- Multiple output functions: See Example 3.26, pages 220-222.

Digital Logic Circuits

Logic Synthesis (Chapters 2-5)

Logic Synthesis

- Definition: To design a logic circuit such that it meets the specifications and can be economically manufactured:
  - Performance – meets delay specification, or has minimum delay.
  - Cost – uses minimum hardware, smallest chip area, smallest number of gates or transistors.
  - Power – meets power specification, or consumes minimum power.
  - Testablility – has no redundant (untestable) logic and is easily testable.

Synthesis Procedure

- Minimization – Obtain MSOP or MPOS. This is also known as two-level minimization because the result can be implemented as a two-level AND-OR or NAND-NAND or NOR-NOR circuit.
- Technology mapping – Considering design requirements, transform the minimized form into one of the technologically realizable forms:
  - Programmable logic array (PLA)
  - Standard cell library
  - Field programmable gate array (FPGA)
  - Others . . .

References on Synthesis

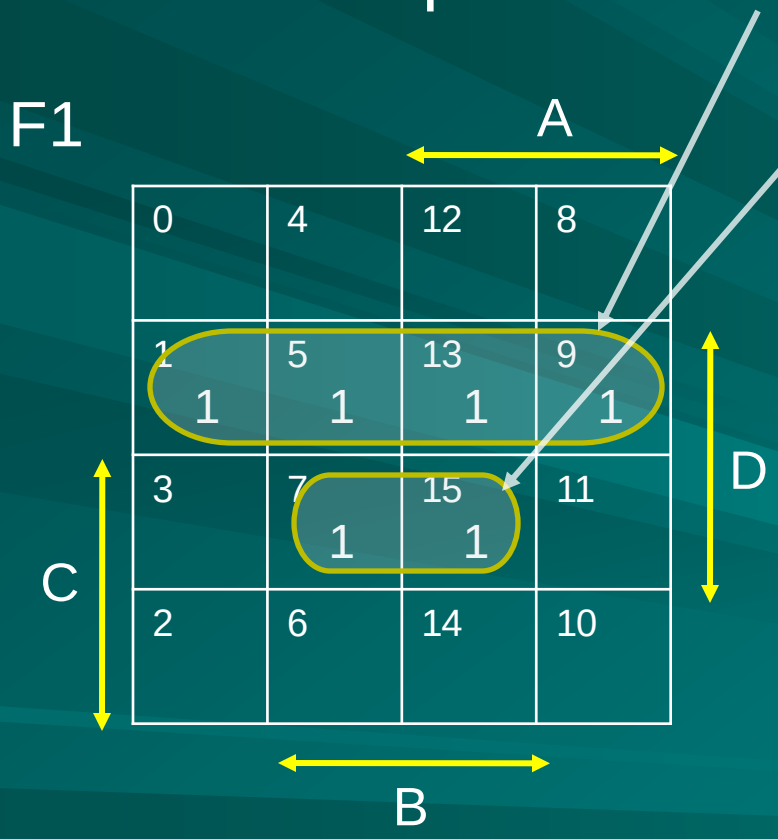
- G. De Micheli, *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill, 1994.
- S. Devadas, A. Ghosh and K. Keutzer, *Logic Synthesis*, New York: McGraw-Hill, 1994.

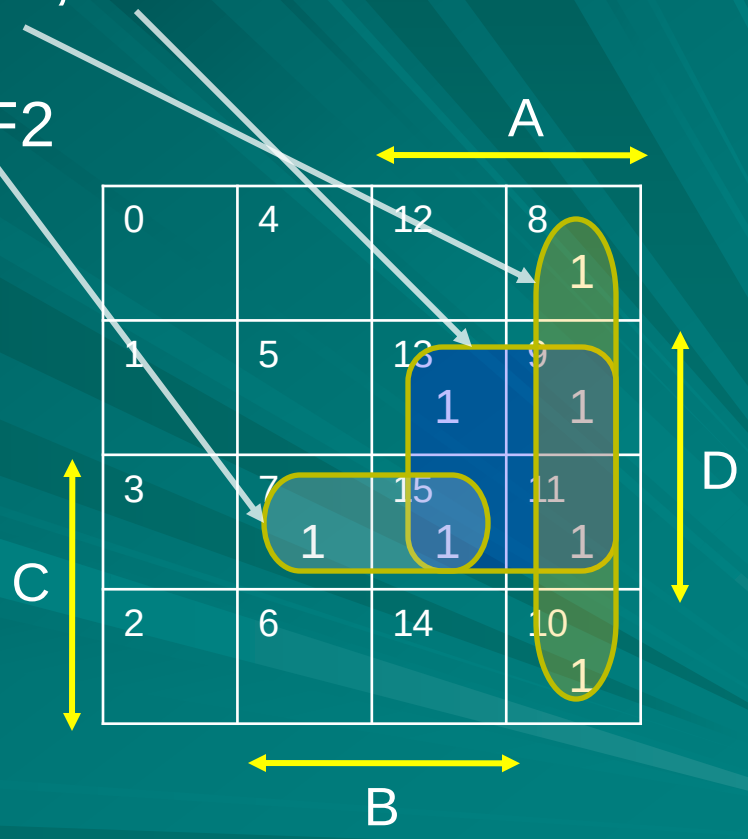
Programmable Logic Array (PLA)

- A direct implementation of multi-output function as a two-level circuit in MOS technology.
- PLA styles:
  - NAND-NAND
  - NOR-NOR
- Textbook, Chapter 5.

Example: Two-Output Function

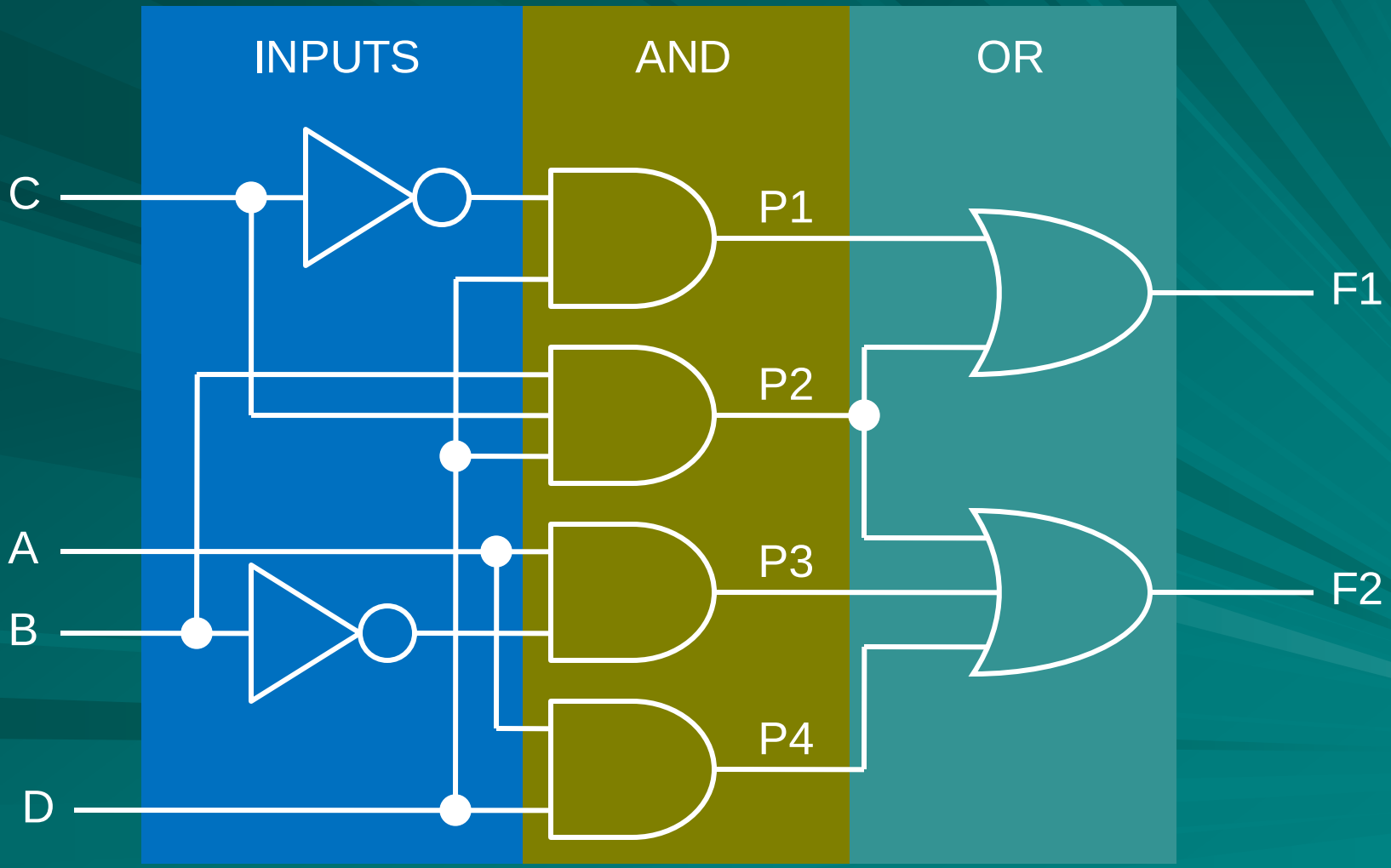
Need four products: P1, P2, P3, P4



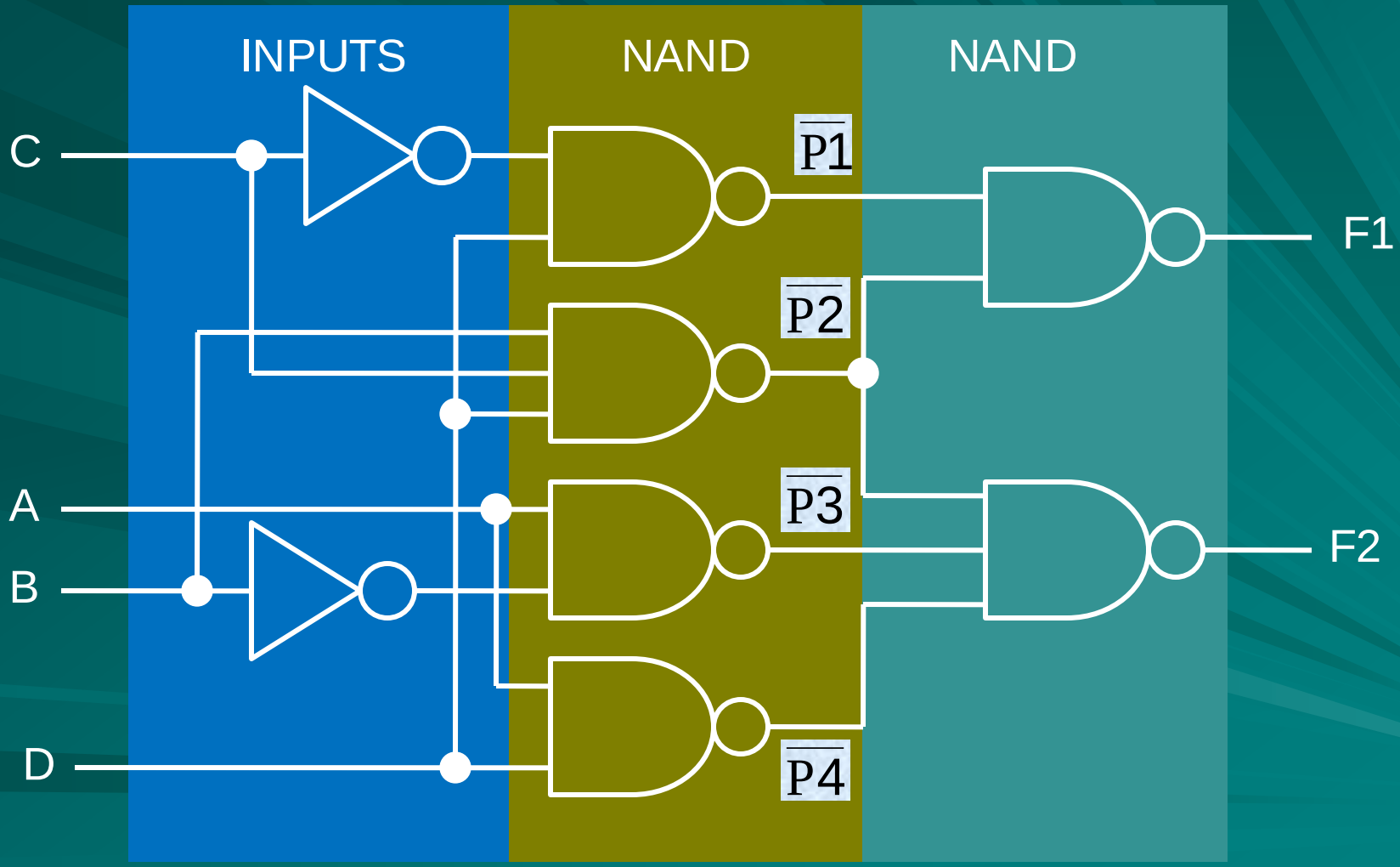


Two-Level AND-OR Implementation

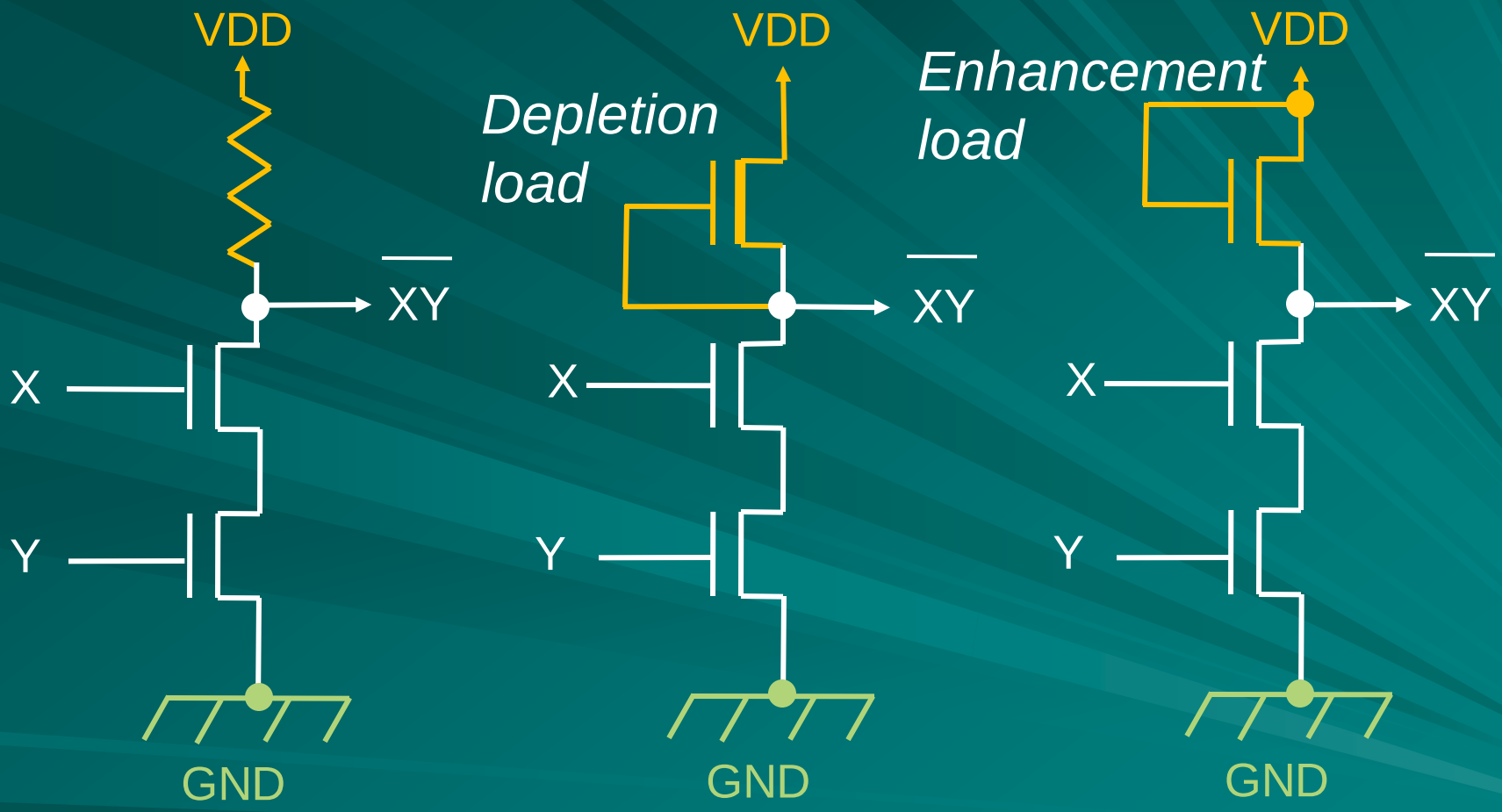
- Also known as technology-independent circuit.



NAND-NAND Implementation

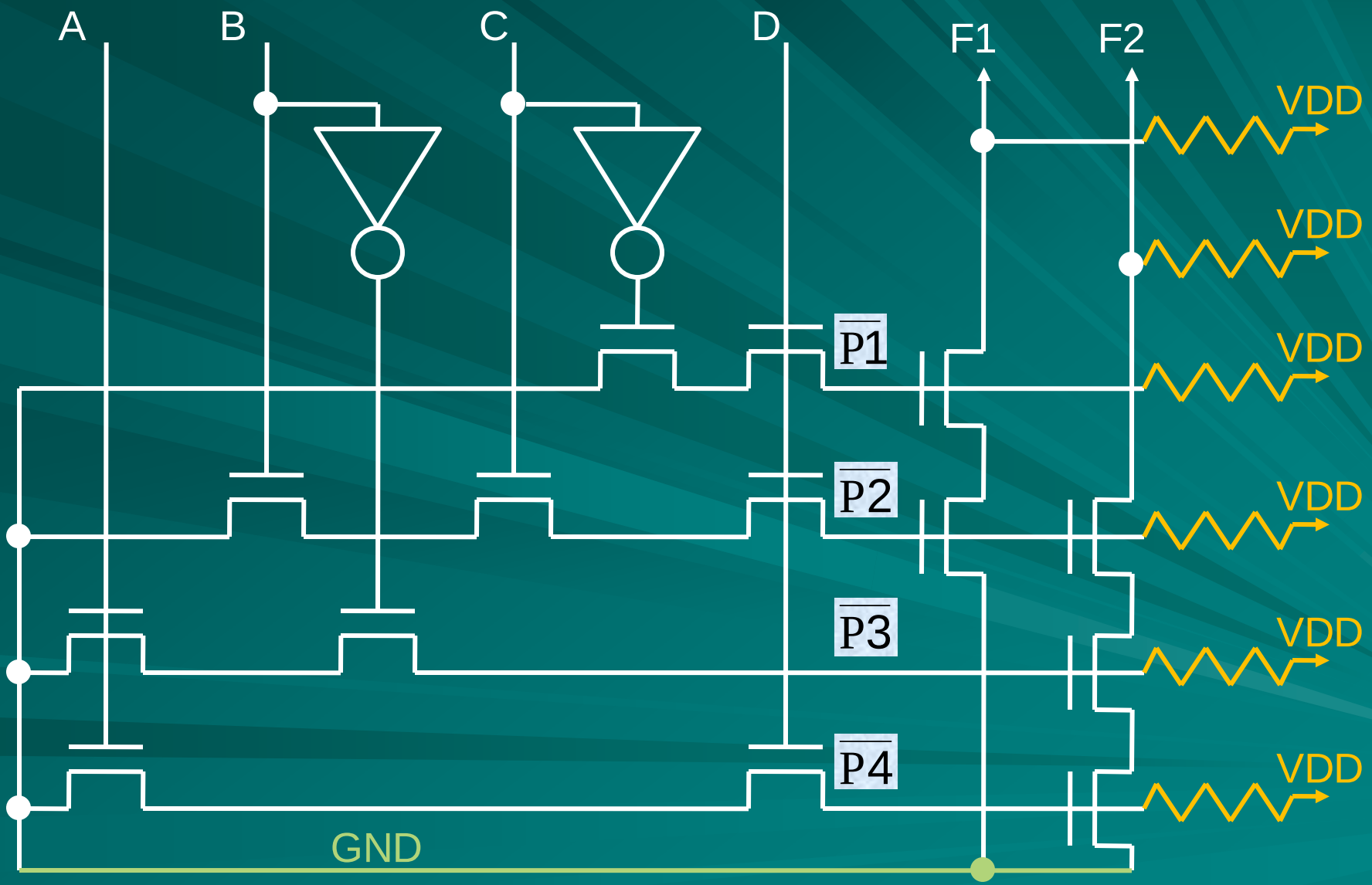


A NAND Gate in nMOS Technology

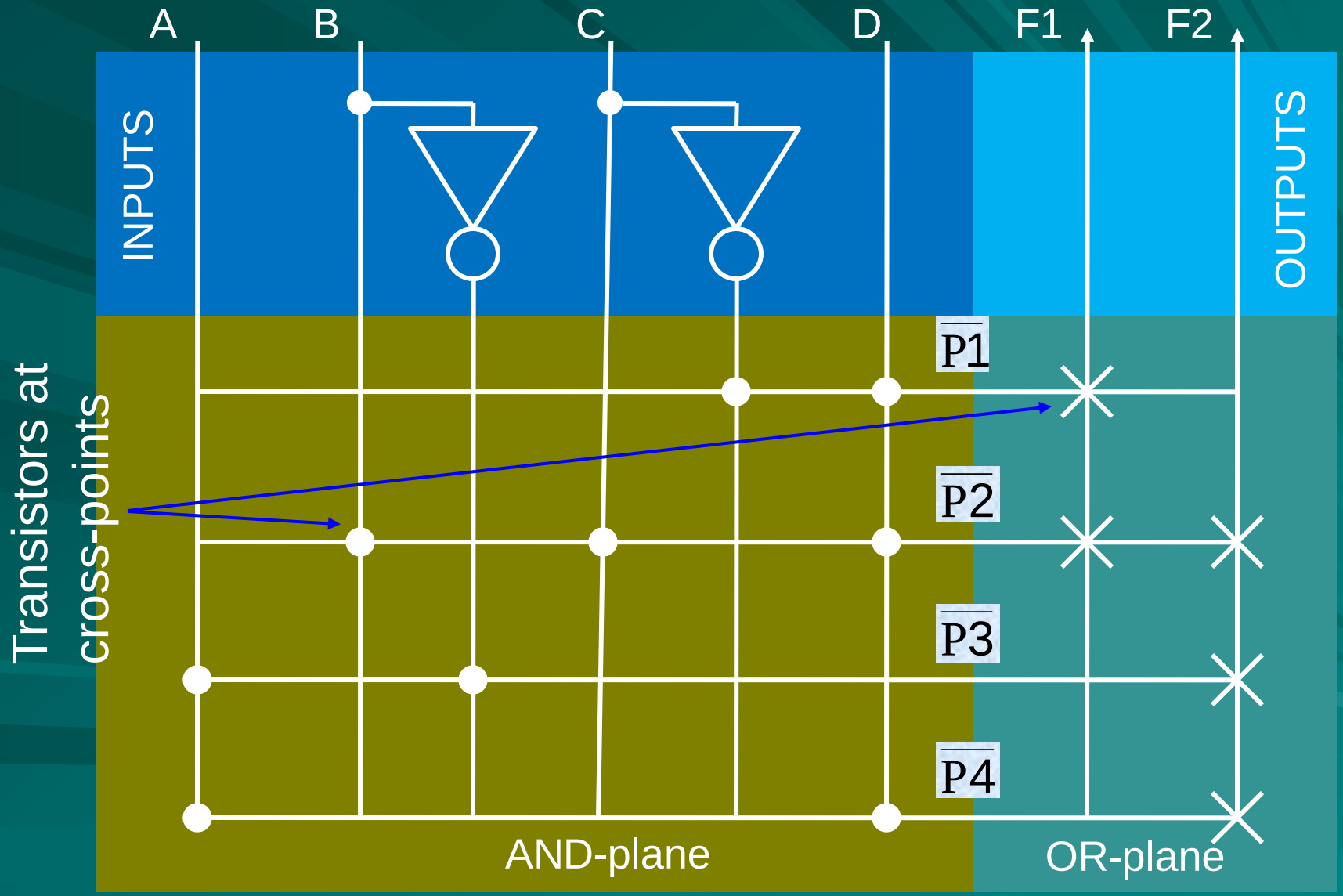


R. C. Jaeger and T, N. Blalock, Microelectronic Circuit Design, Boston: McGraw-Hill, 2008, Section 6.8.2.

NAND-NAND PLA



NAND-NAND PLA SCHEMATIC



Standard-Cell Design

- Obtain two-level minimized form.
- Map the design onto predesigned building blocks called standard cells (technology mapping).
- Standard-cell library contains predesigned logic cells in the technology of manufacture. Examples of technology:
  - 90 nanometer CMOS
  - 65 nanometer CMOS
  - 45 nanometer CMOS
  - . . .
- This is known as application-specific integrated circuit (ASIC).

Technology Mapping

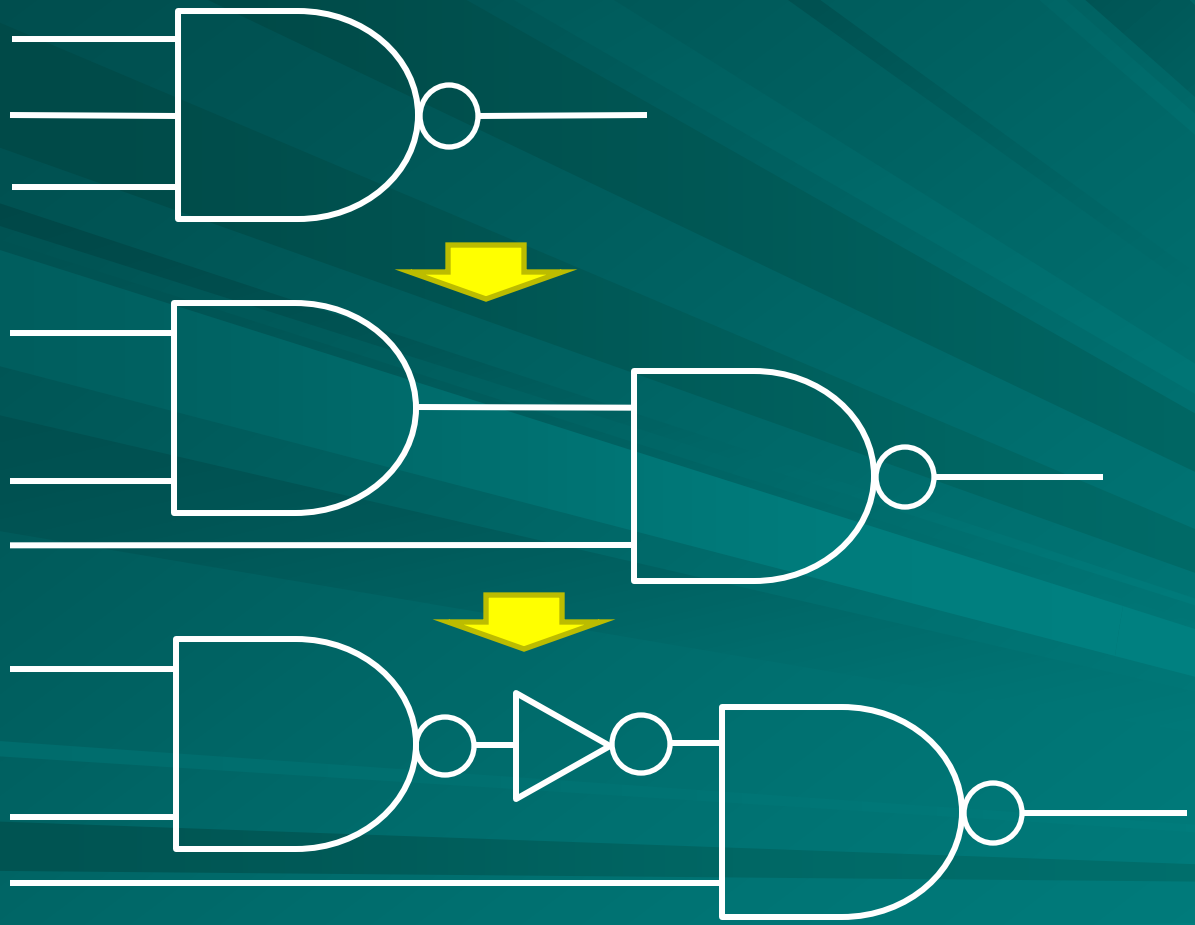
- Find a common logic element, e.g., two-input NAND gate or inverter (one-input NAND).
- MSOP is converted into NAND-NAND circuit.
- Split larger input gates into two-input NAND gates and inverters.
- Cover the circuit with standard cells, also split into two-input NAND gates and inverters (graph-matching).

A Typical Cell Library

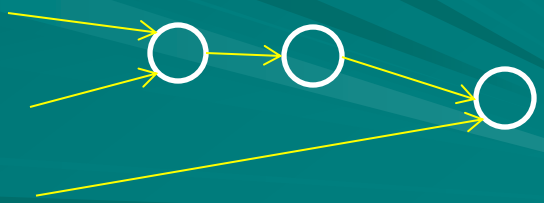
Name	Area units (cost)	Inputs	Output function, Z
Inverter	2	A	$Z = \overline{A}$
NAND2	3	A, B	$Z = \overline{AB}$
NAND3	4	A, B, C	$Z = \overline{ABC}$
NAND4	5	A, B, C, D	$Z = \overline{ABCD}$
AOI21	4	A, B, C	$Z = \overline{AB + C}$
OAI21	4	A, B, C	$Z = \overline{(A + B)C}$
AOI22	5	A, B, C, D	$Z = \overline{AB + CD}$
XOR	4	A, B	$Z = A\overline{B} + \overline{A}B$

S. Devadas, A. Ghosh and K. Keutzer, *Logic Synthesis*, New York: McGraw-Hill 1994, Section 7.7, pp. 185-198.

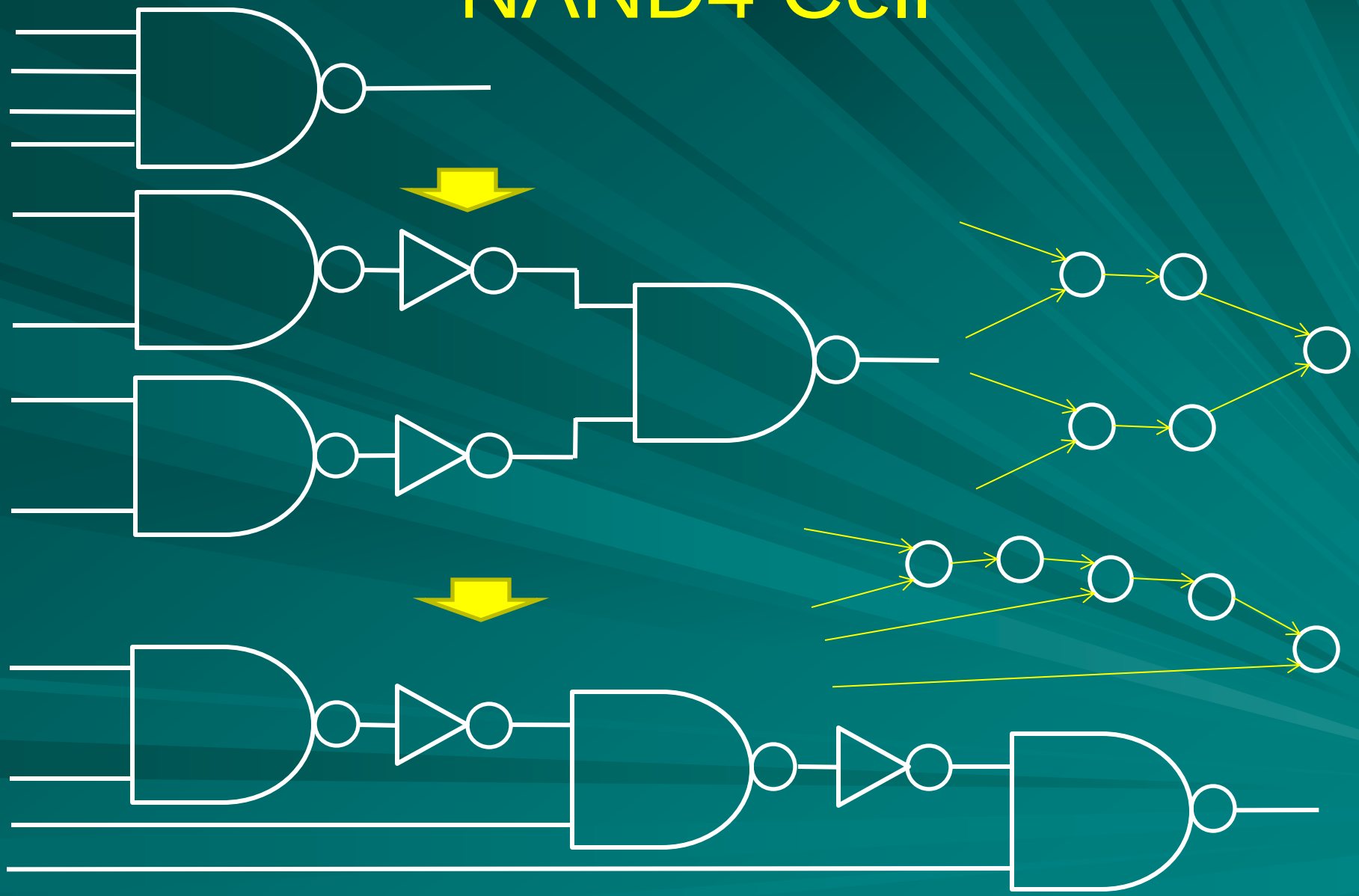
NAND3 Cell



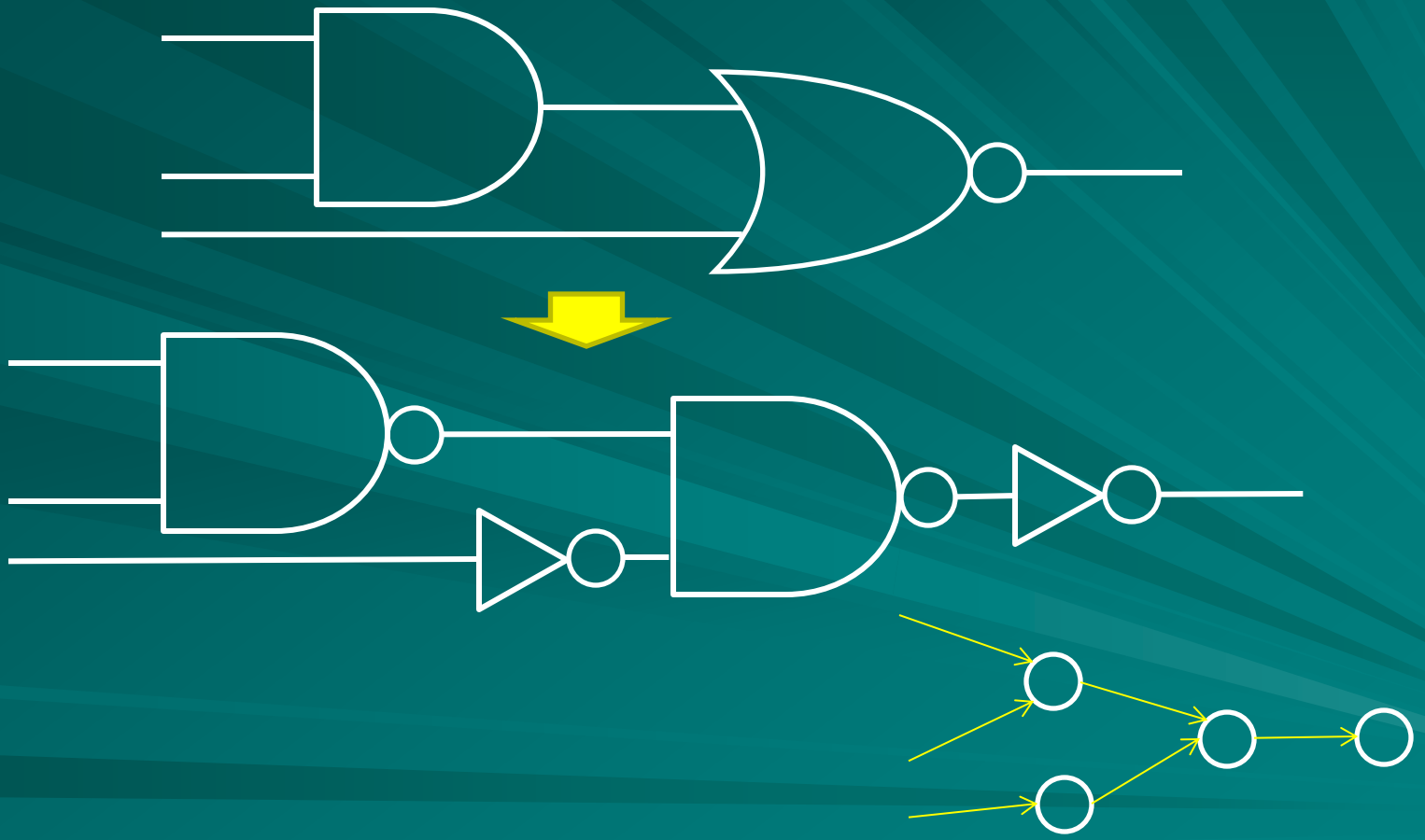
Directed Acyclic Graph (DAG) (tree)



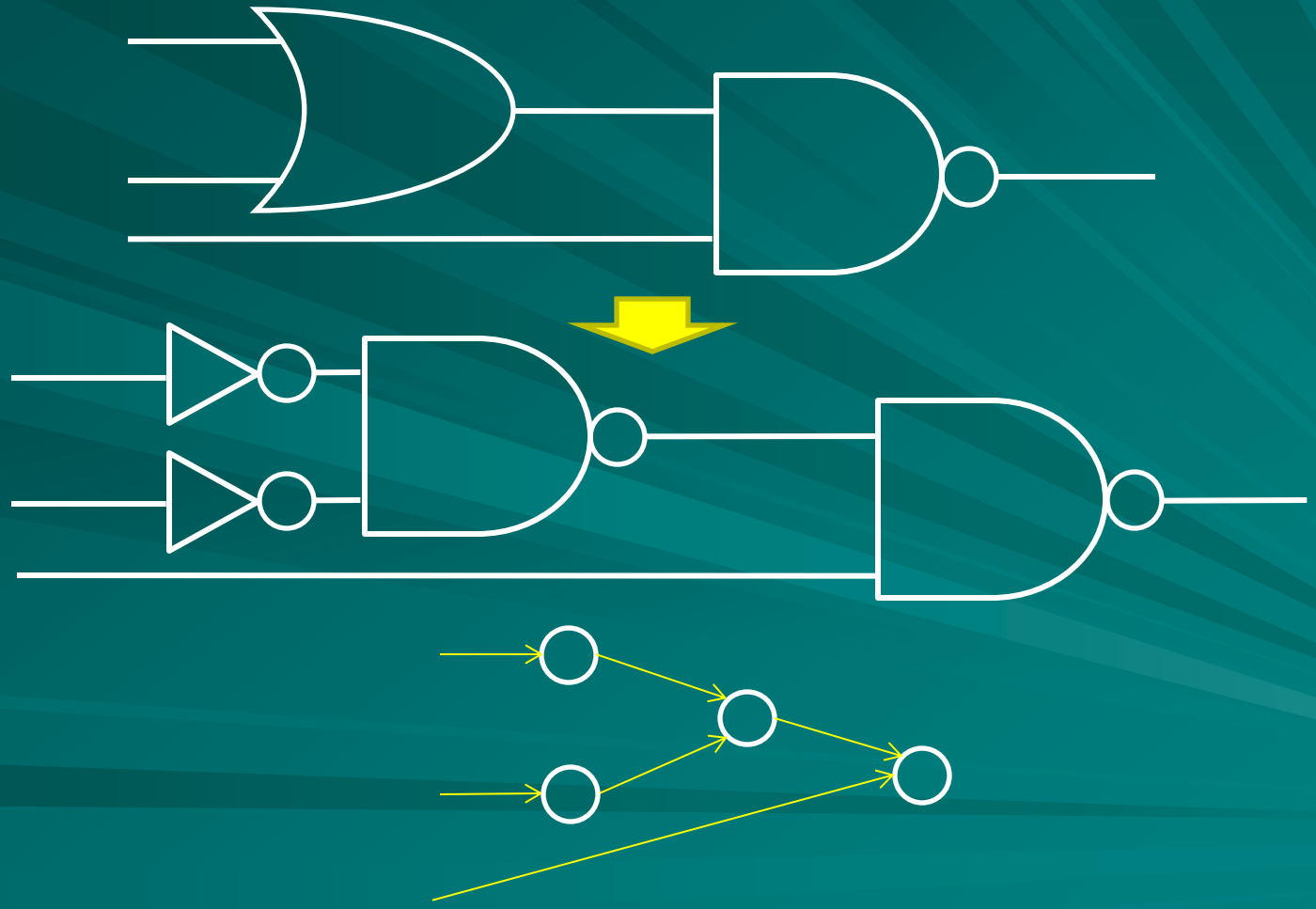
NAND4 Cell



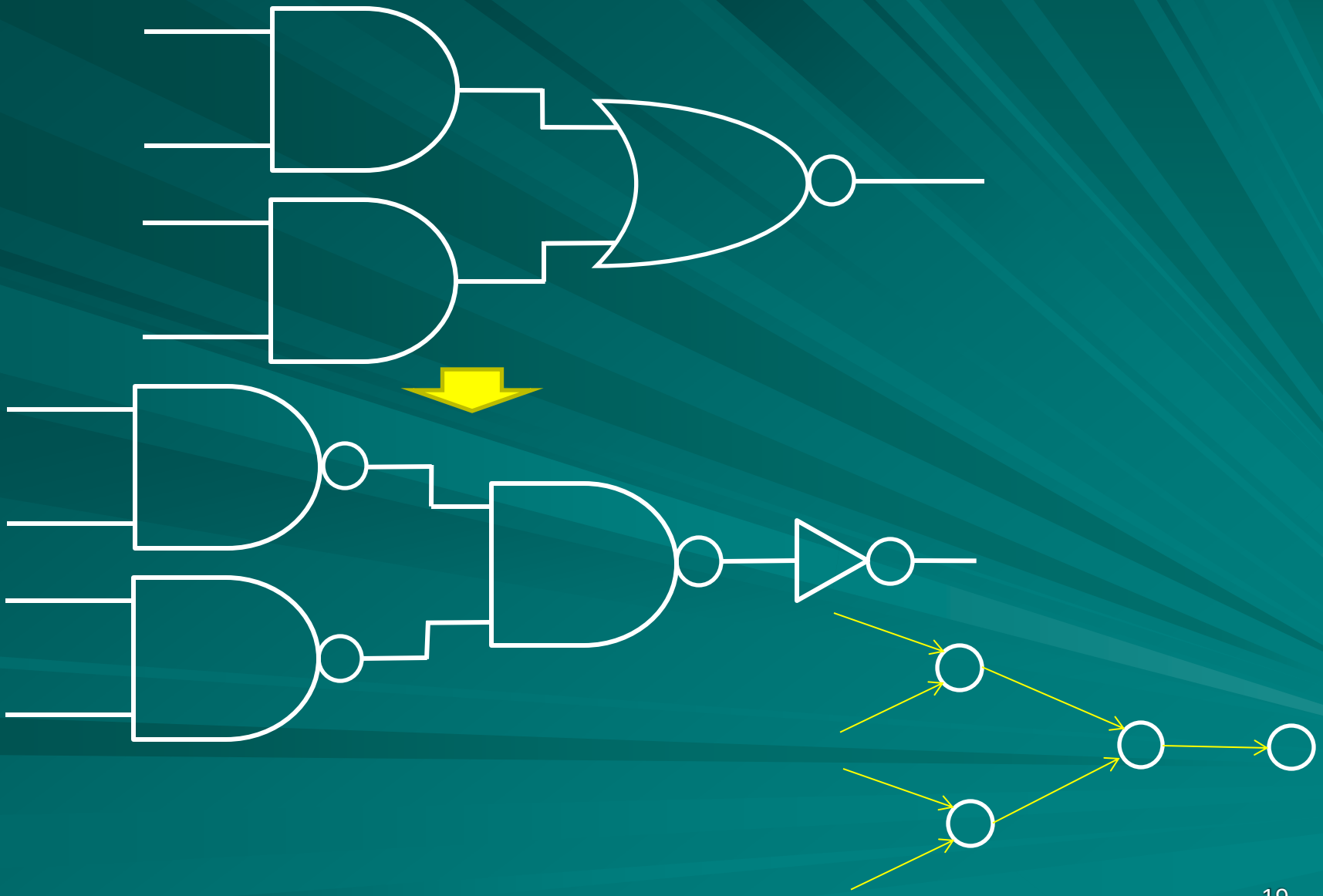
AOI21 Cell



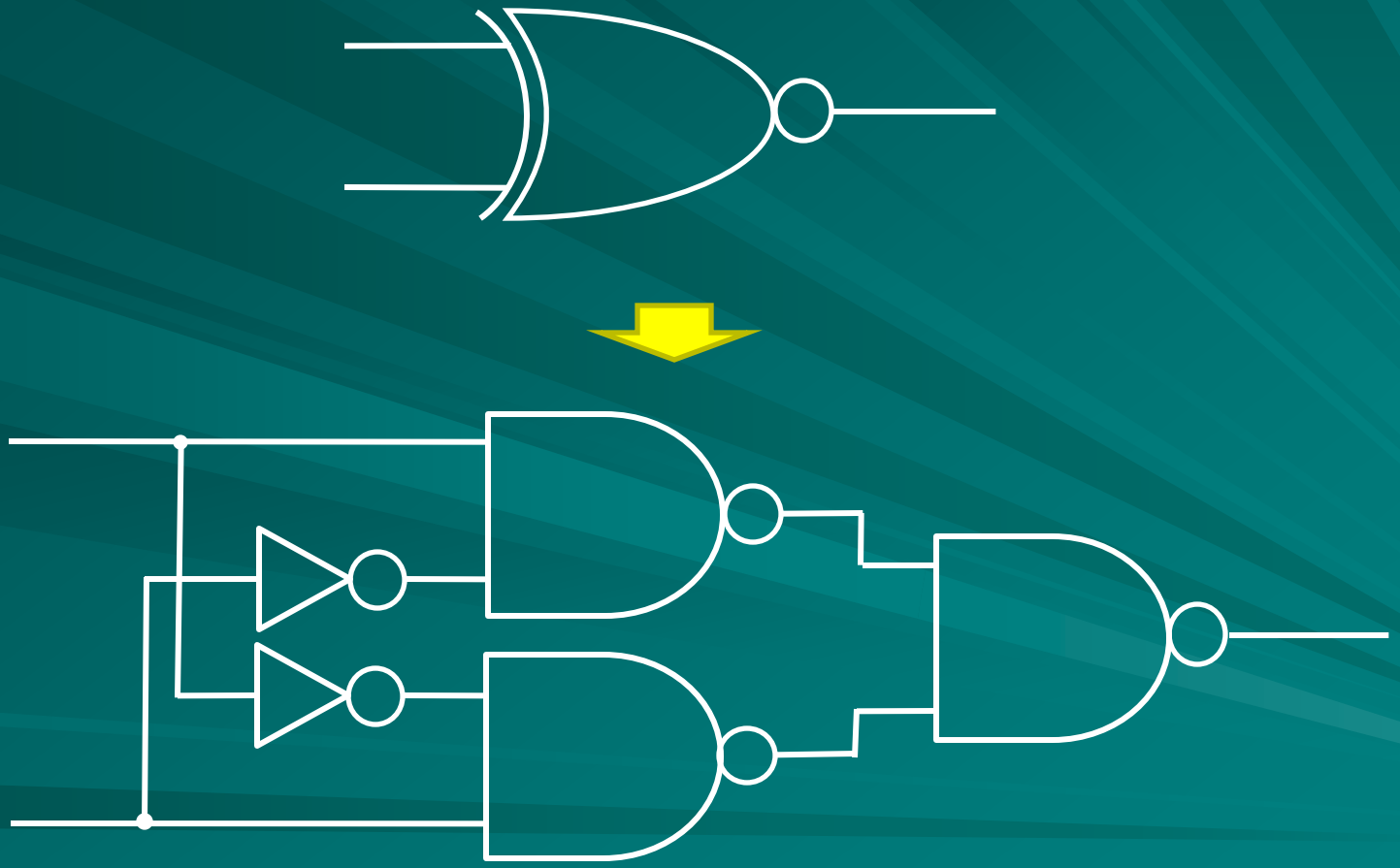
OAI21 Cell



AOI22 Cell



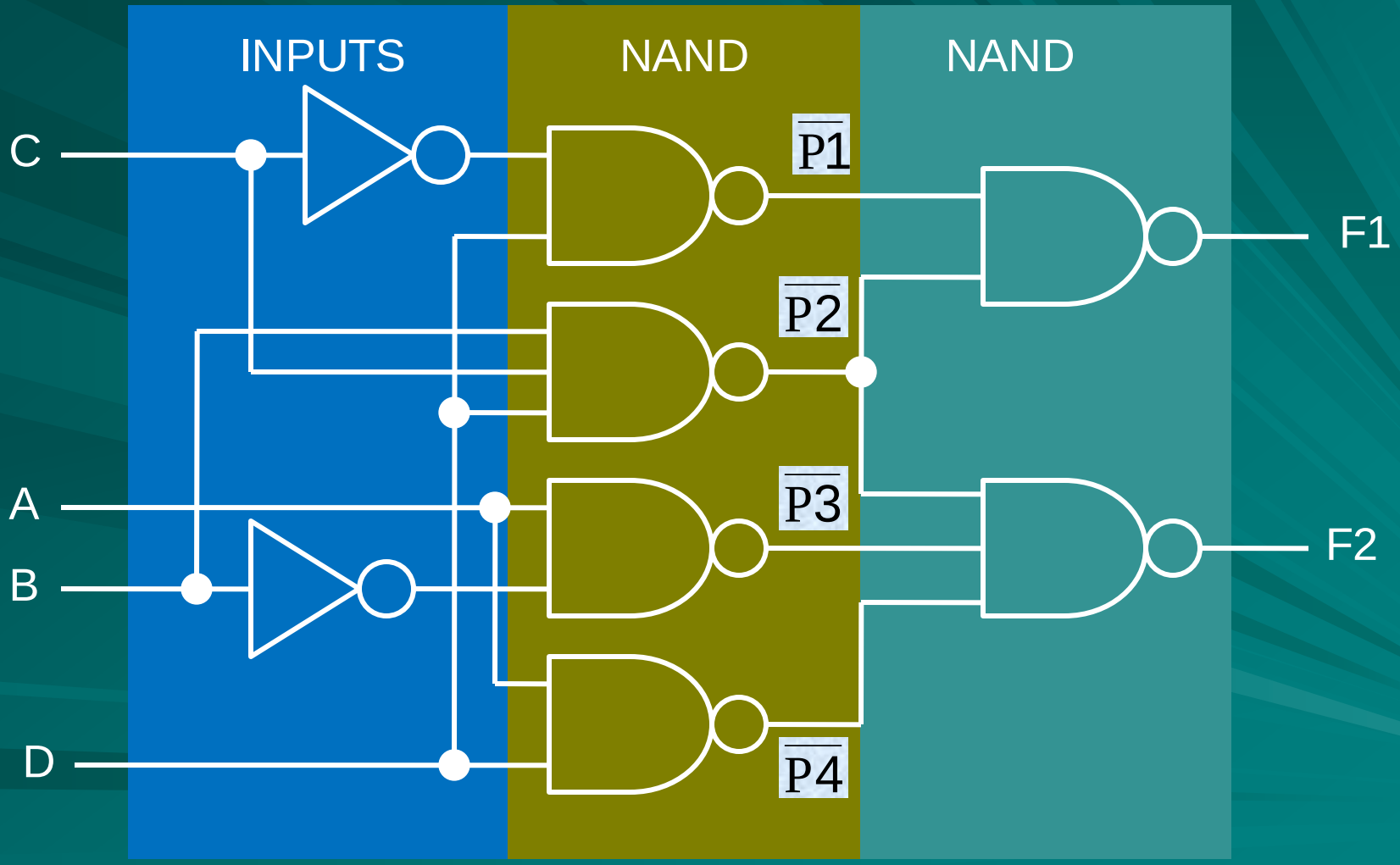
XOR Cell



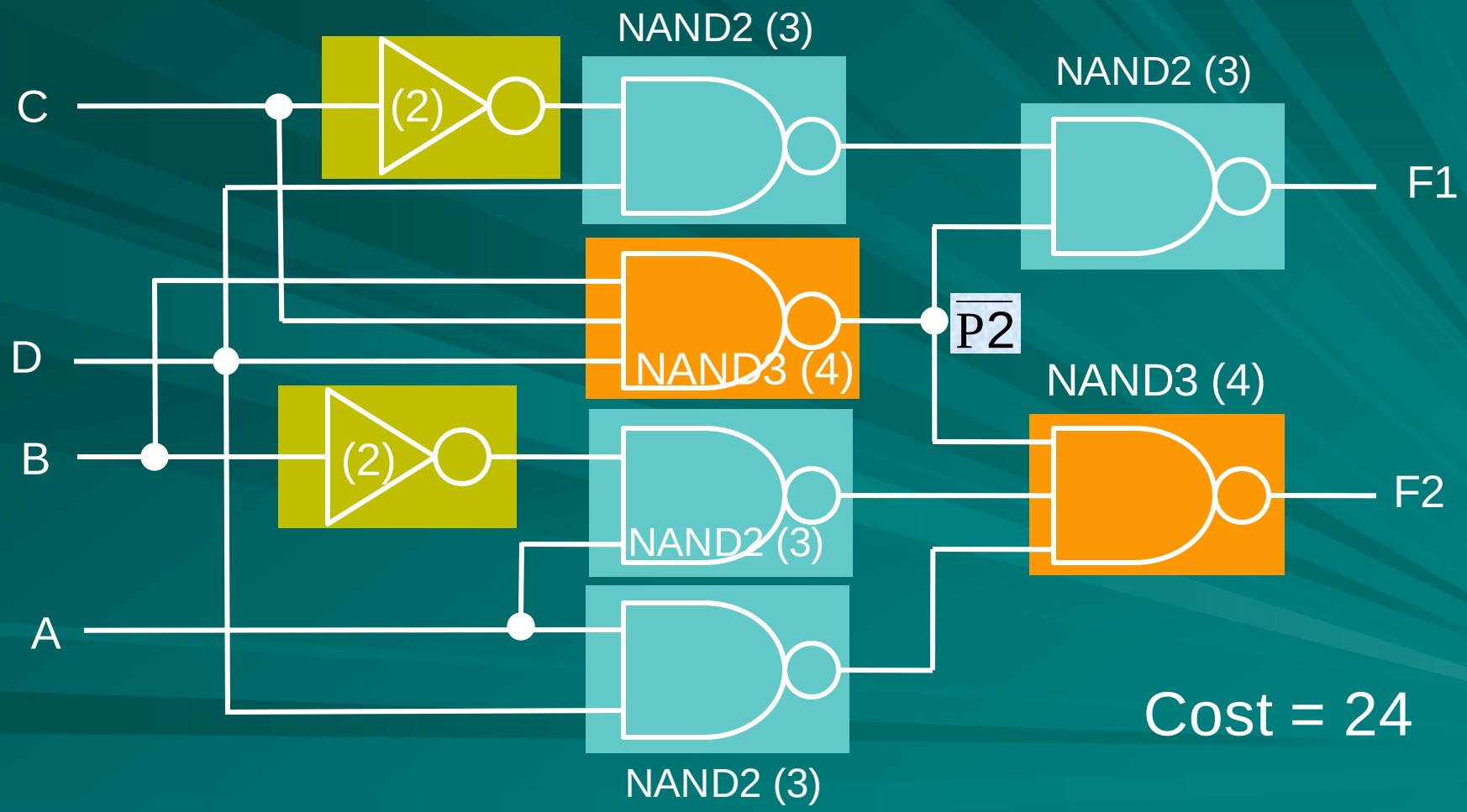
Technology Mapping Procedure

- Obtain MSOP.
- Convert to two-level AND-OR circuit.
- Transform to two-level NAND-NAND circuit.
- Transform to two-input NAND and inverter tree network.
- Perform an optimal pattern matching to obtain a minimum cost tree covering.

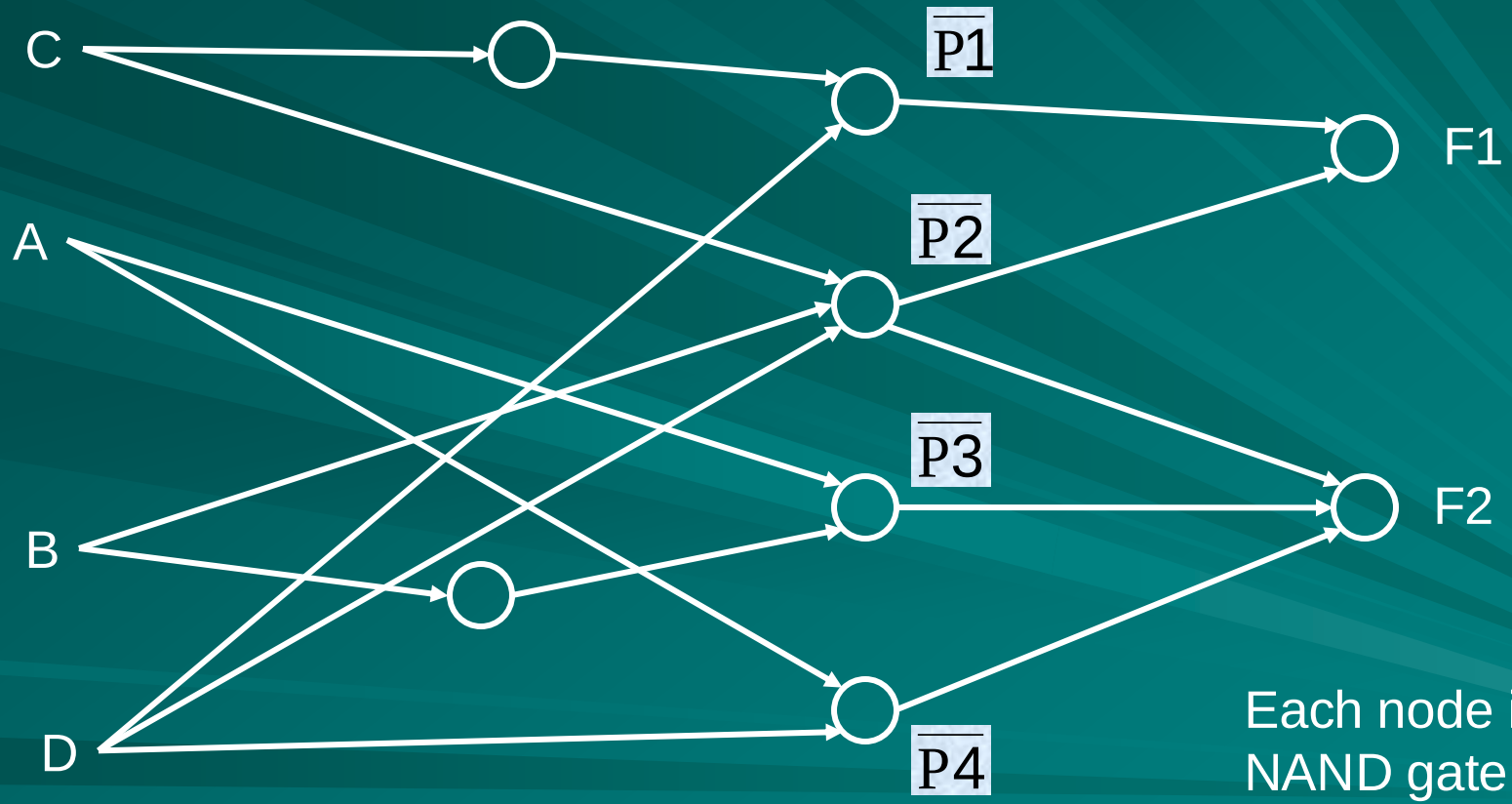
Previous Example: 2-Level NAND



A Simple Technology Mapping

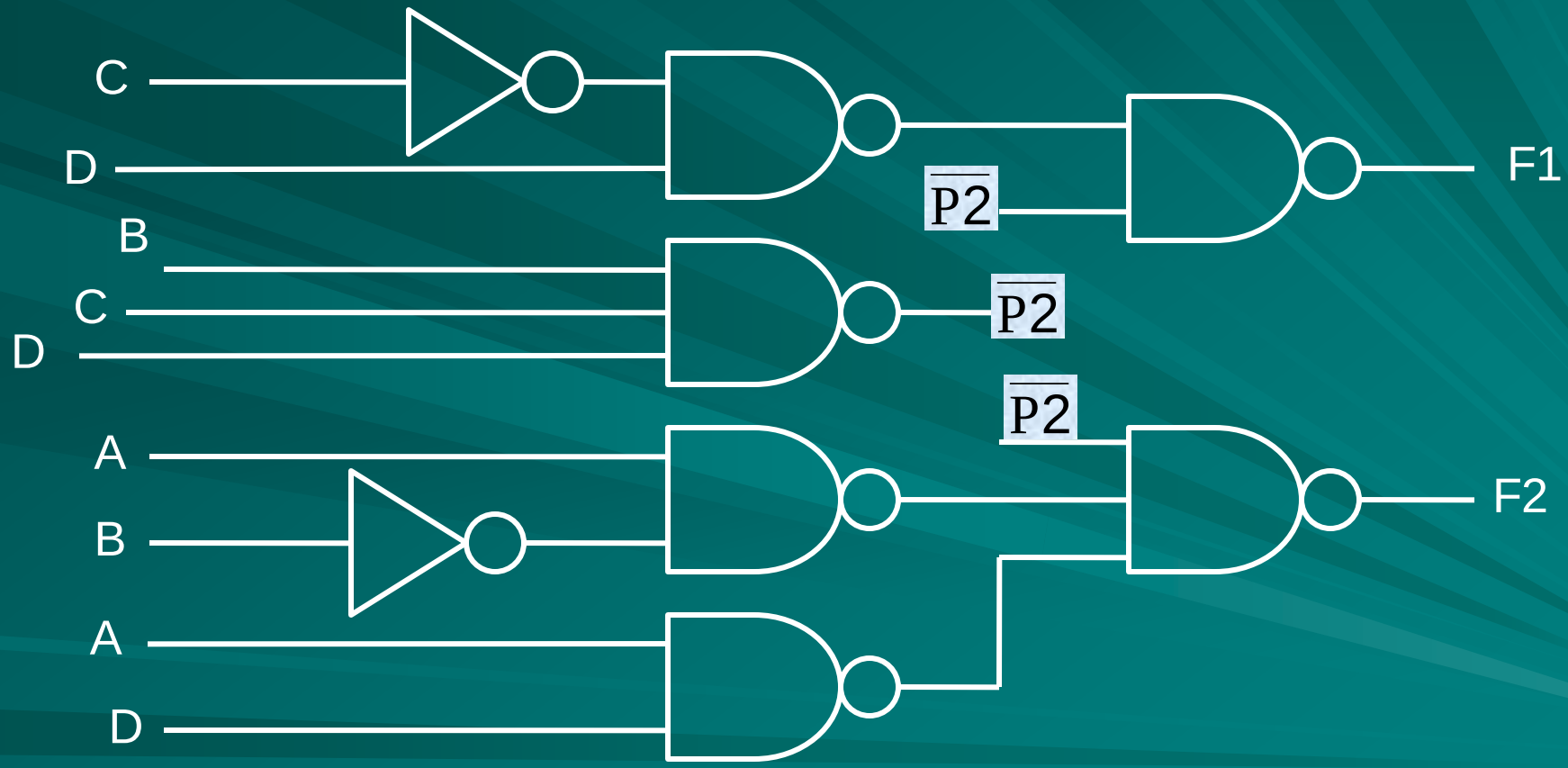


Circuit is a Directed Acyclic Graph (DAG)

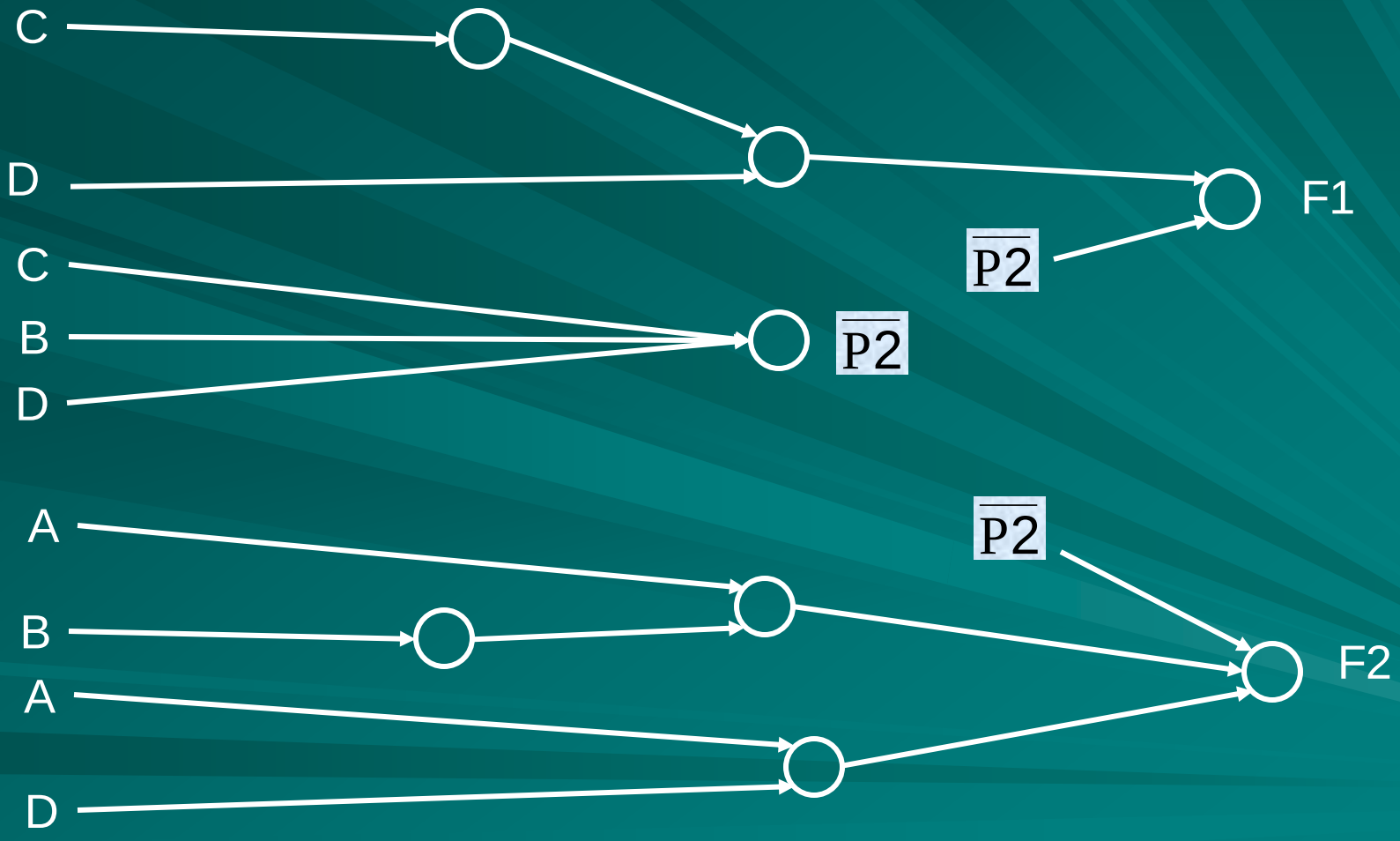


Each node is a NAND gate.

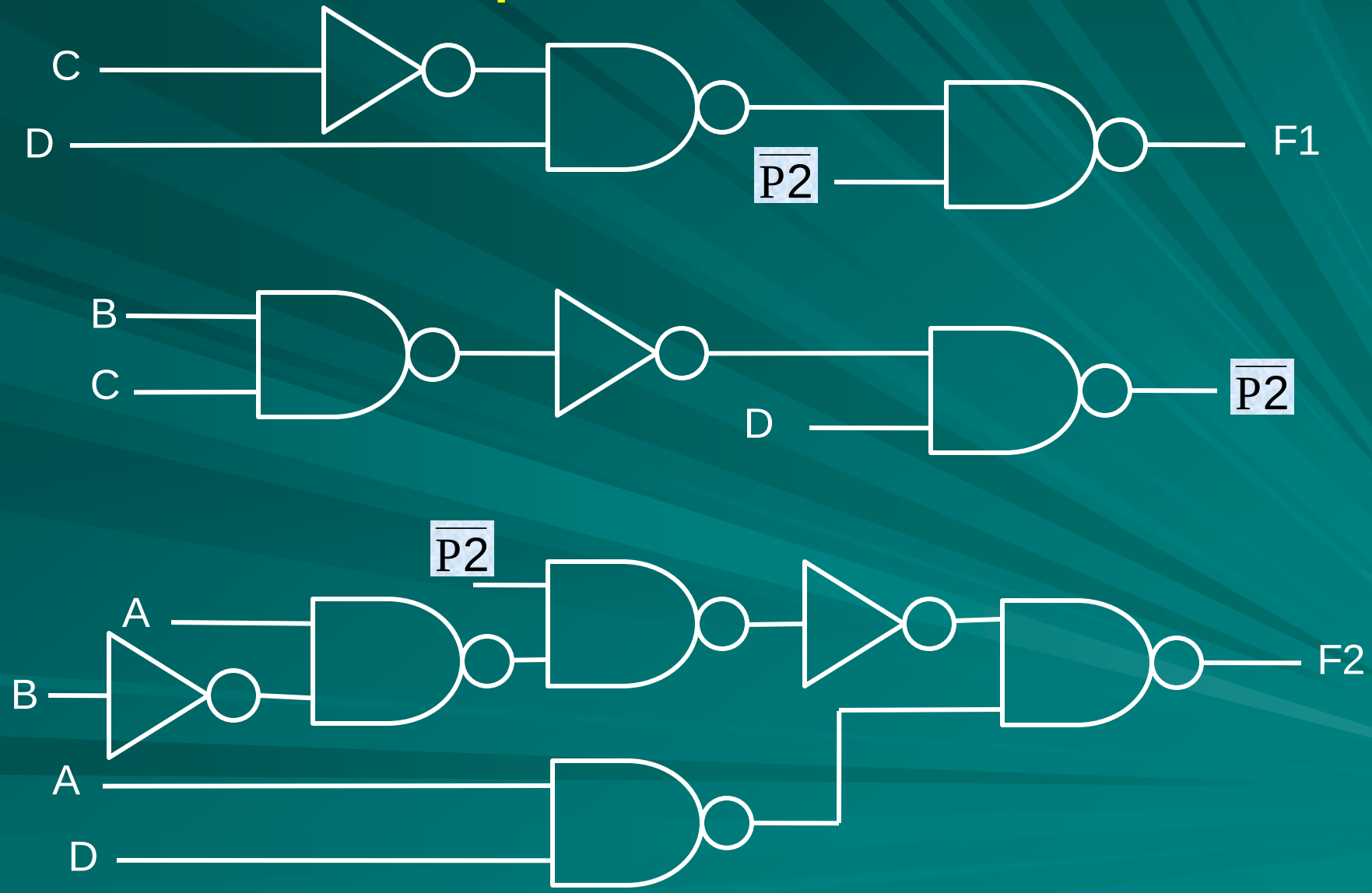
Splitting into a Forest of Trees



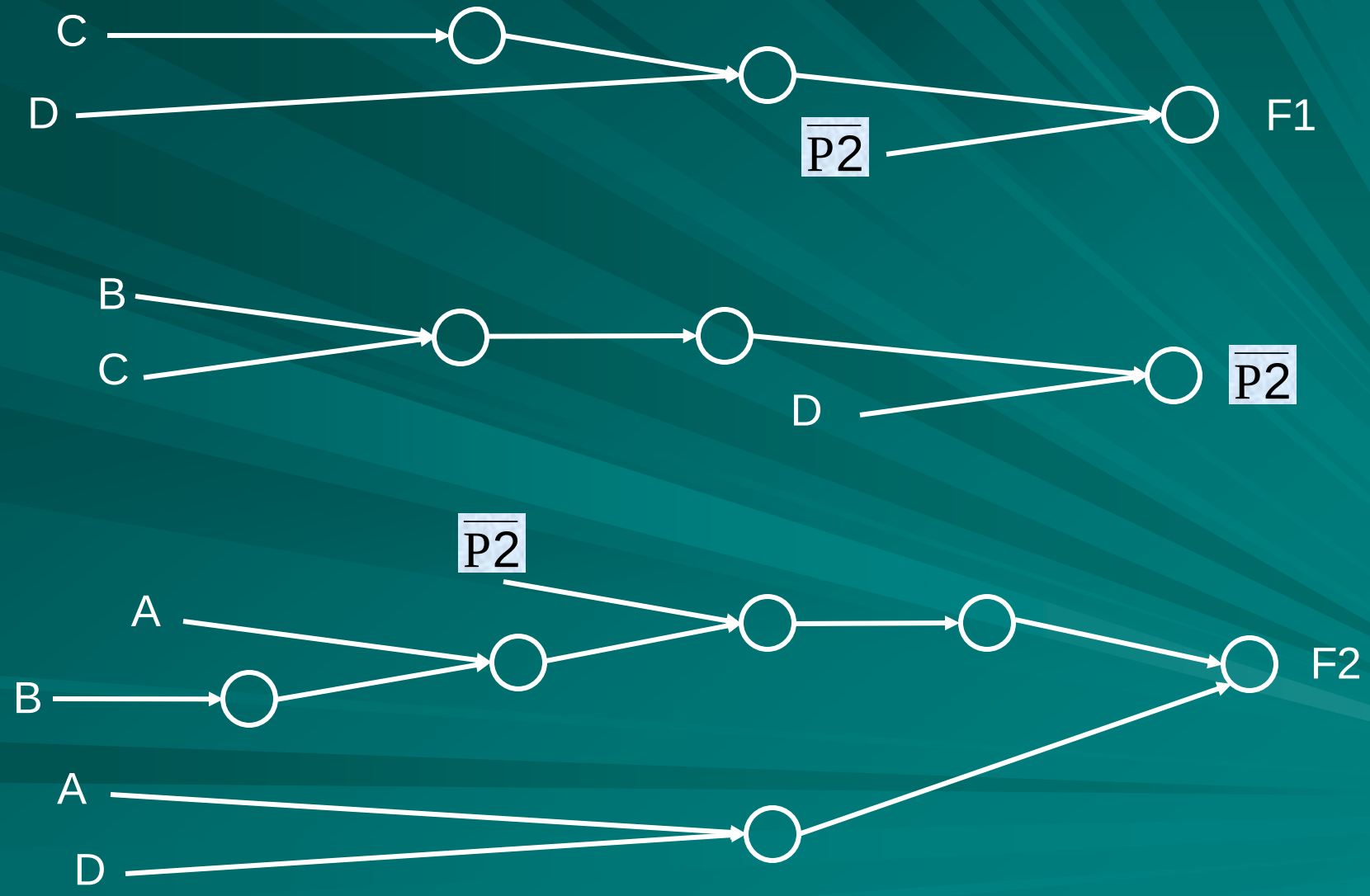
Splitting DAG into Trees (Forest)



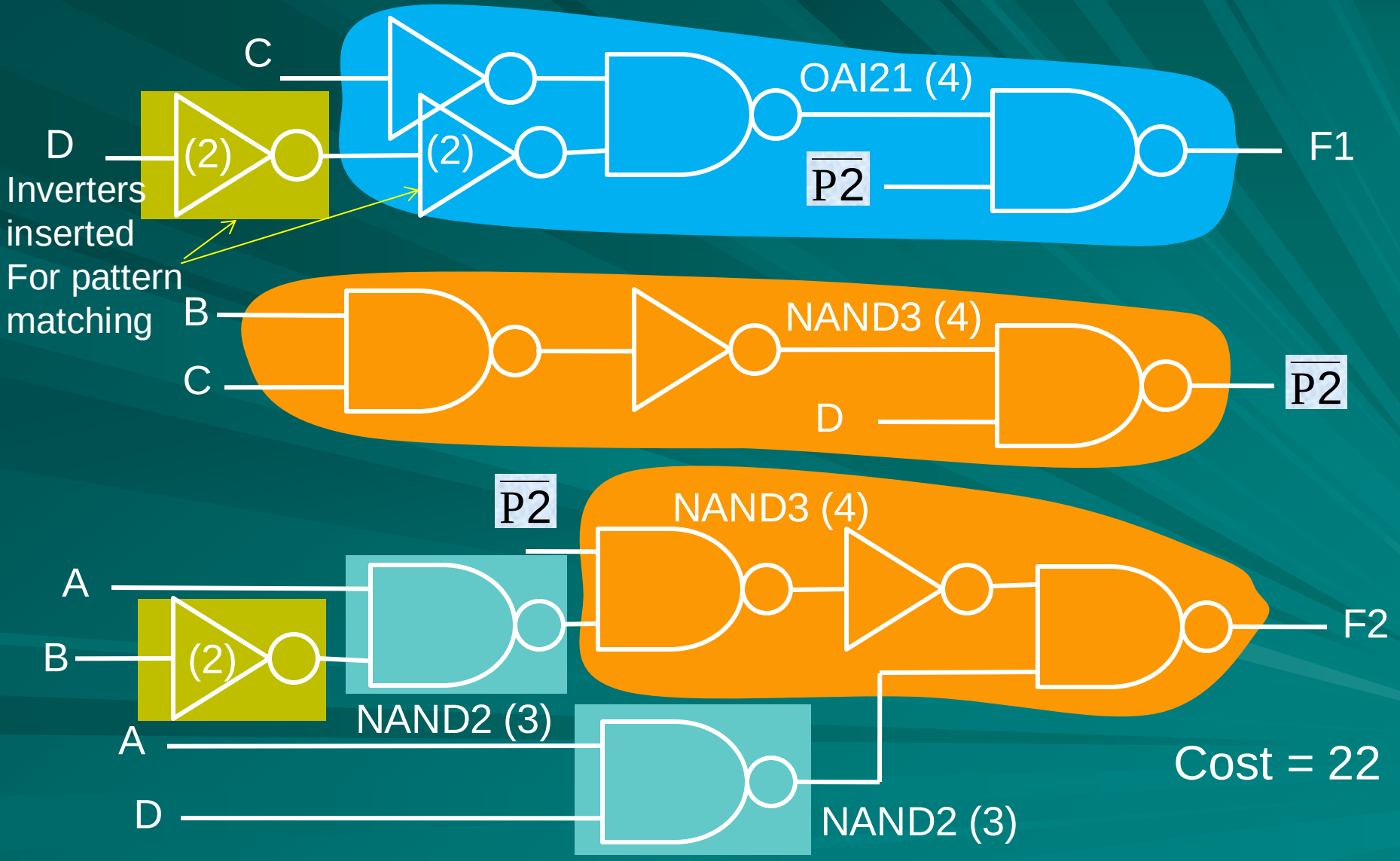
Two-Input NAND Trees



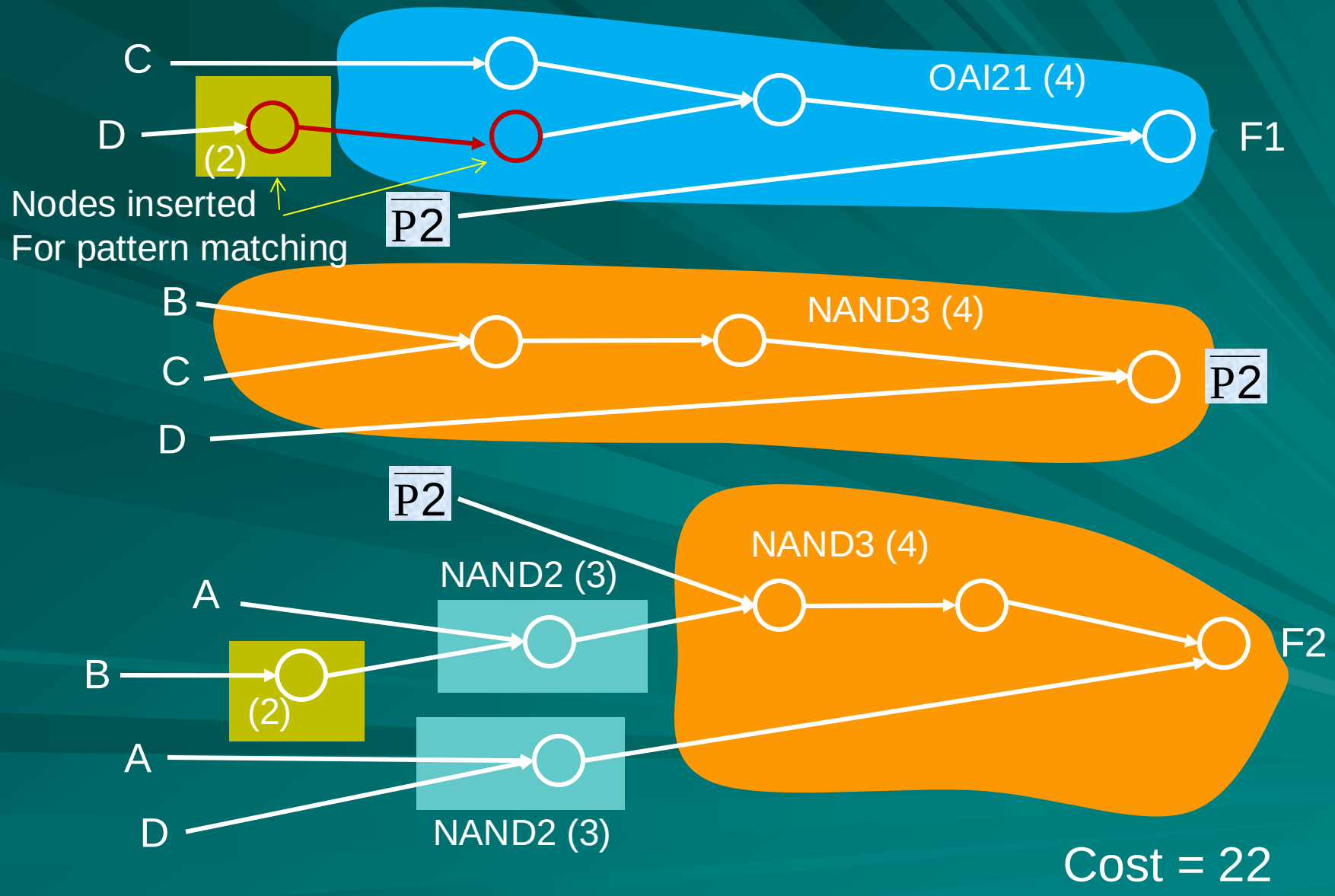
Alternatively, in Graph Format



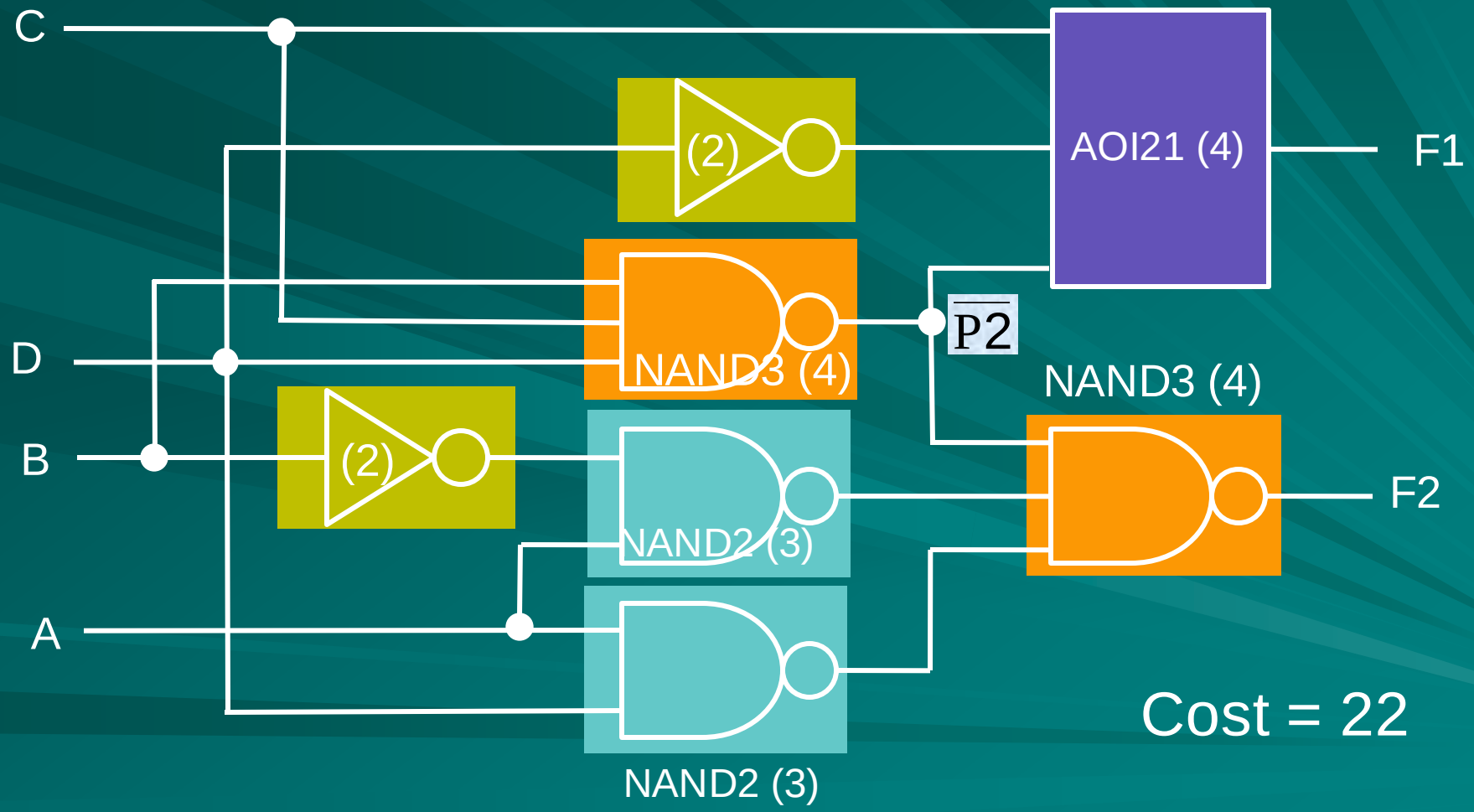
An Improved Technology Mapping



Alternatively, in Graph Format



Improved Technology Mapping



Original Reference

- K. Keutzer, “DAGON: Technology Binding and Local Optimization by DAG matching,” *Proc. 24th Design Automation Conf.,* 1987, pp. 341-347.

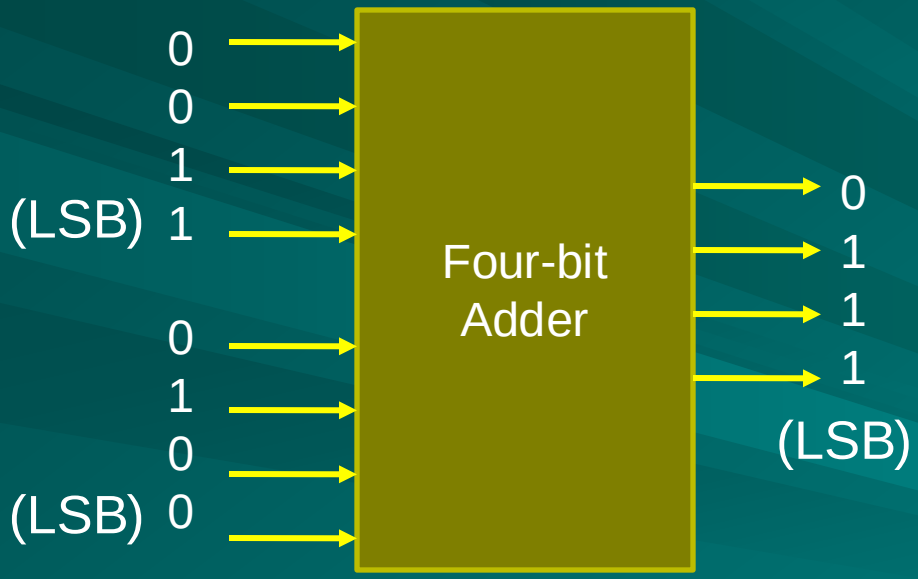
Digital Logic Circuits

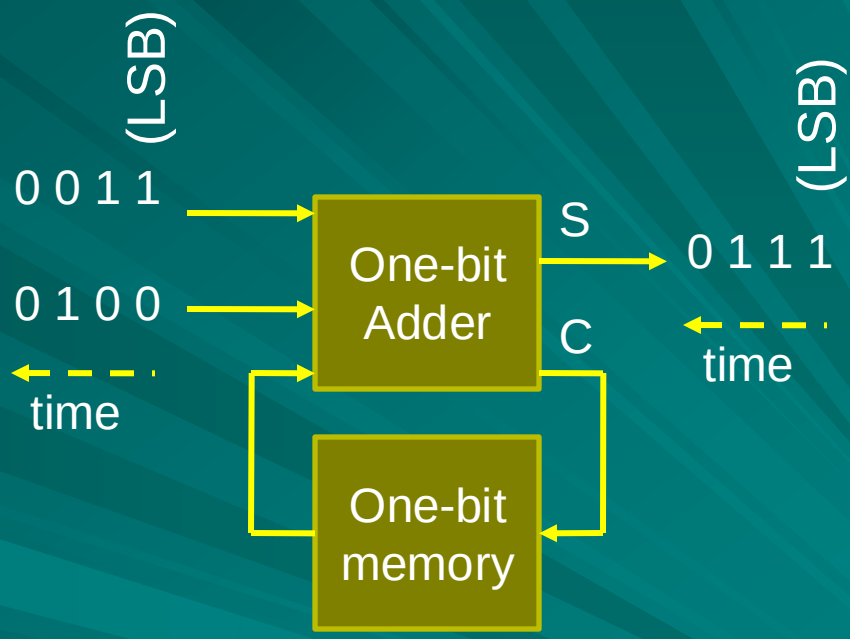
Sequential Circuits (Chapter 6)

Combinational vs. Sequential

- Combinational circuit:
  - Output is a function of input
  - No memory
  - Example: parallel adder
- Sequential circuit:
  - Output is a function of input and something else stored in the circuit
  - Internal memory
  - Example: serial adder

Parallel and Serial Adders

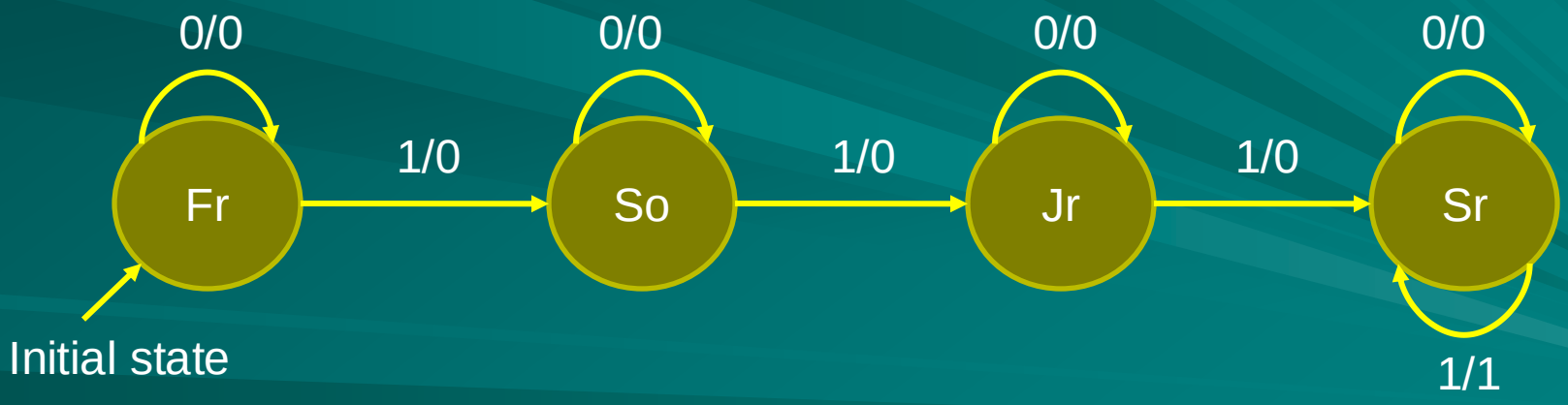




1. Memory initialized to 0 (initial carry = 0)
2. Time synchronization of Inputs, output, and memory (clock)

Another Example of Sequential Circuit

- Four-year degree program:
  - Student can be in four states (Fr, So, Jr, Sr)
  - One-bit yearly input, 1 (completed) or 0 (in progress)
  - Output = 1 (degree completed), 0 (in progress)
- State diagram:



State Table or Excitation Table

Input	Present State	Next State	Output
0	Fr	Fr	0
0	So	So	0
0	Jr	Jr	0
0	Sr	Sr	0
1	Fr	So	0
1	So	Jr	0
1	Jr	Sr	0
1	Sr	Sr	1

Initial State: Fr

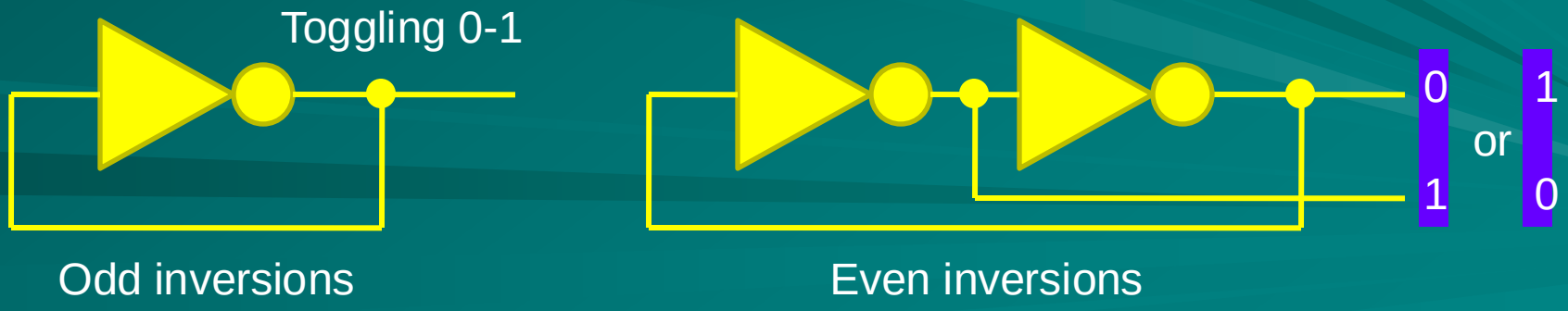
State Table (Alternative Form)

Next state/output

Present state	Inputs 0	Inputs 1
Fr	Fr/0	So/0
So	So/0	Jr/0
Jr	Jr/0	Sr/0
Sr	Sr/0	Sr/1

When Is Circuit Not Combinational?

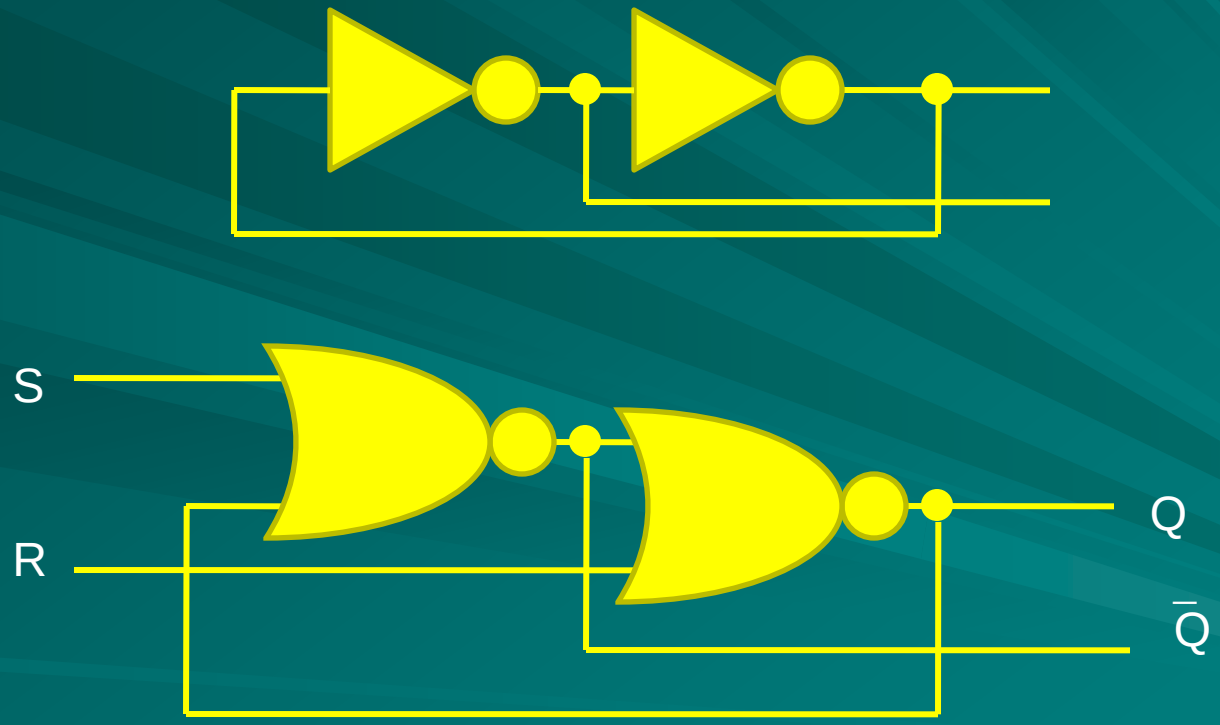
- When the present input does not completely control output.
- For a logic circuit without feedback, input uniquely determines the output.
- Examples of non-combinational (sequential) circuits:



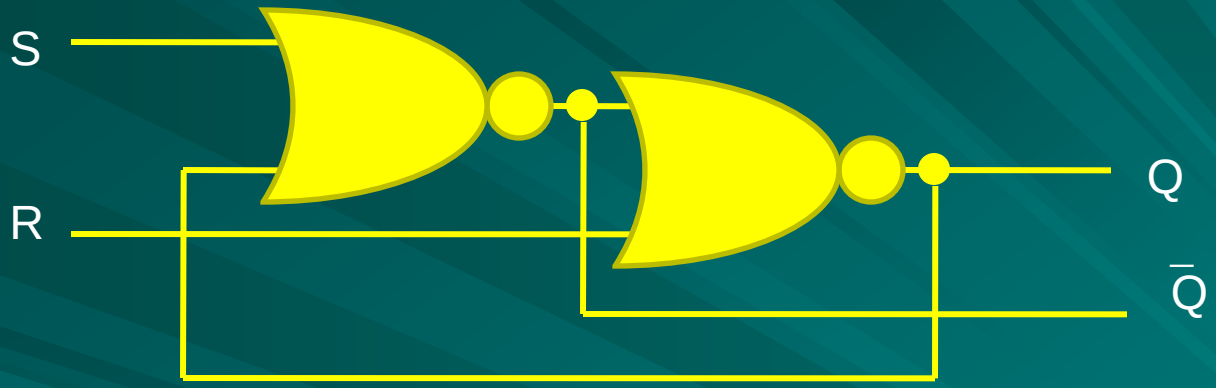
SR Latch: Basic Sequential Circuit

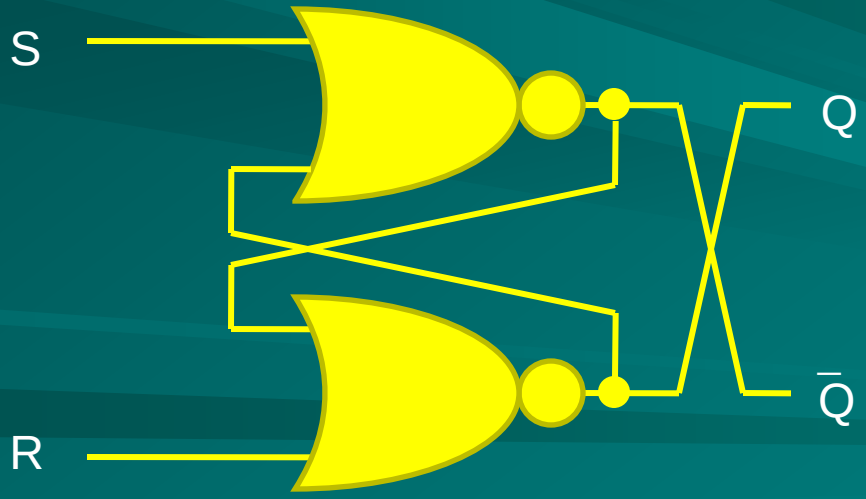
- Feedback loop with even number of inversions (no oscillation?).
- Output(s): two sets of logic values from the loop.
- Input functions:
  - To control loop logic values
  - To set the loop in “input control” or “store” state

Adding Inputs to Feedback Loop

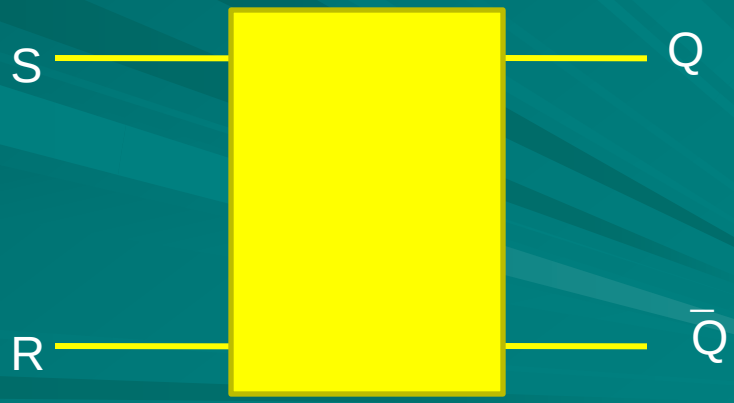


NOR Set-Reset (SR) Latch





Also drawn as

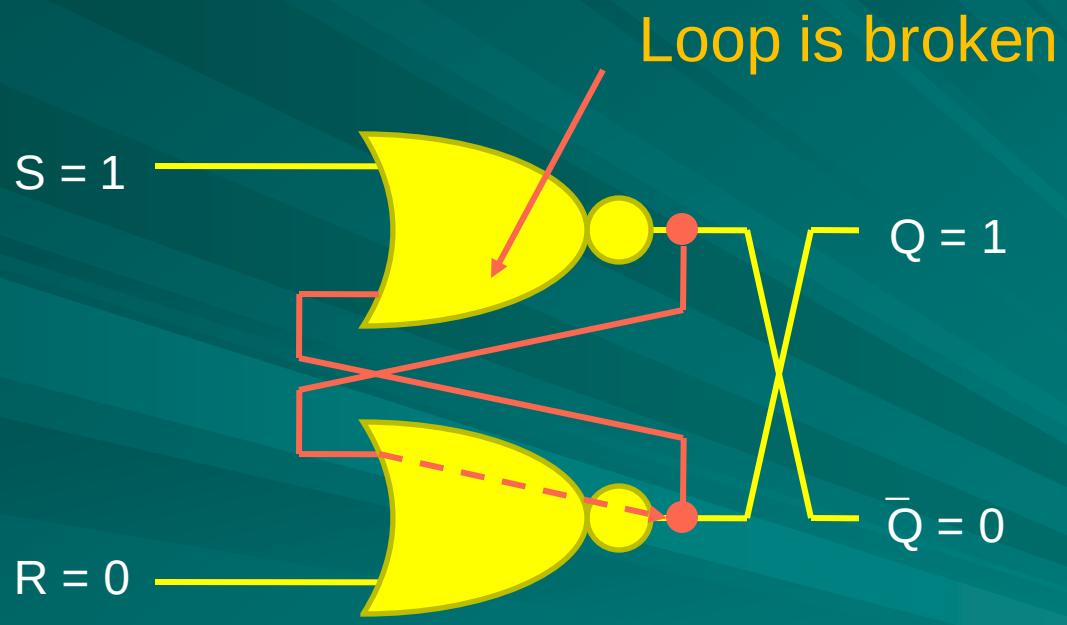


Symbol used in Logic schematics

States of Latch

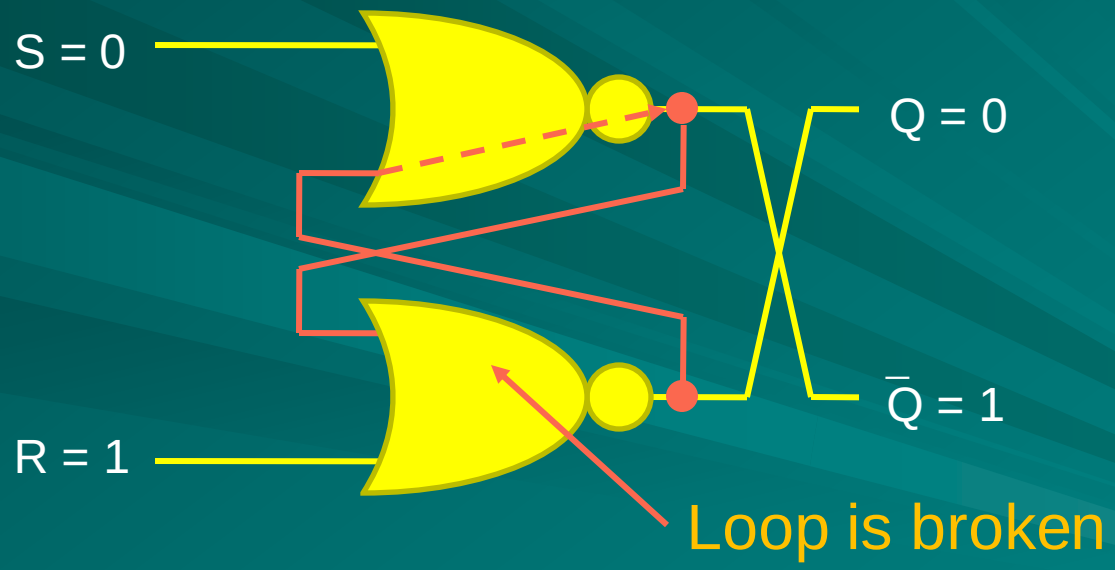
State	S	R	Q	$\overline{Q}$
Set	1	0	1	0
Reset	0	1	0	1
Store	0	0	Prev. Q	Prev. $\overline{Q}$
Illegal	1	1	0	0

The “Set” State



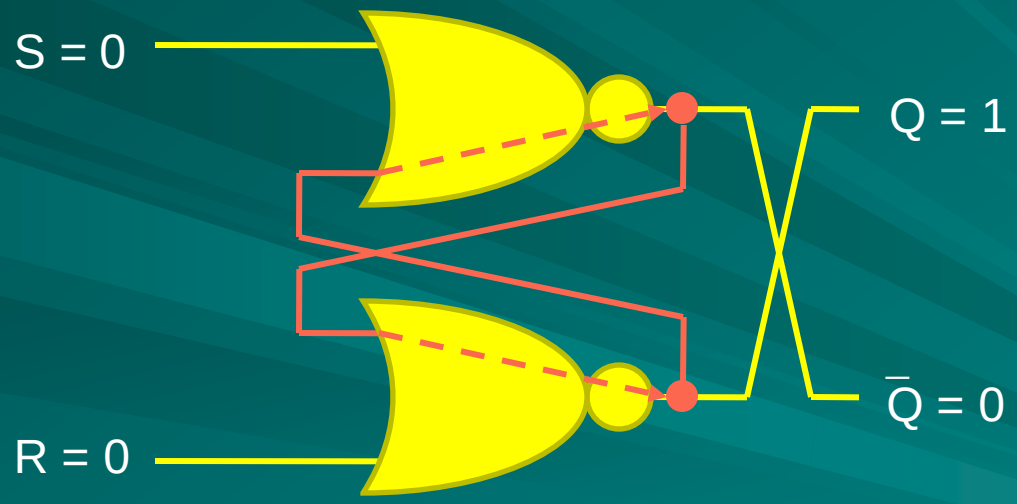
Behavior is combinational.

The “Reset” State



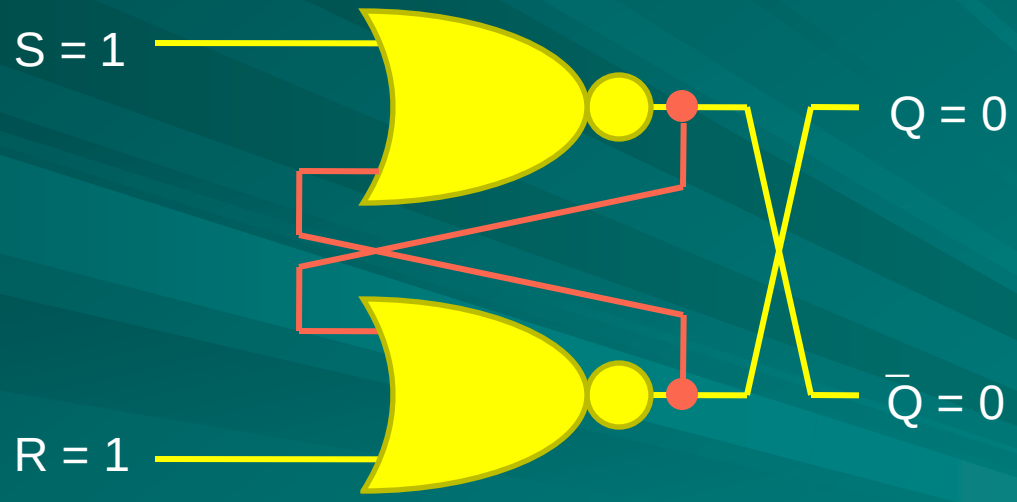
Behavior is combinational.

The “Store” State



Loop is activated; behavior is sequential.

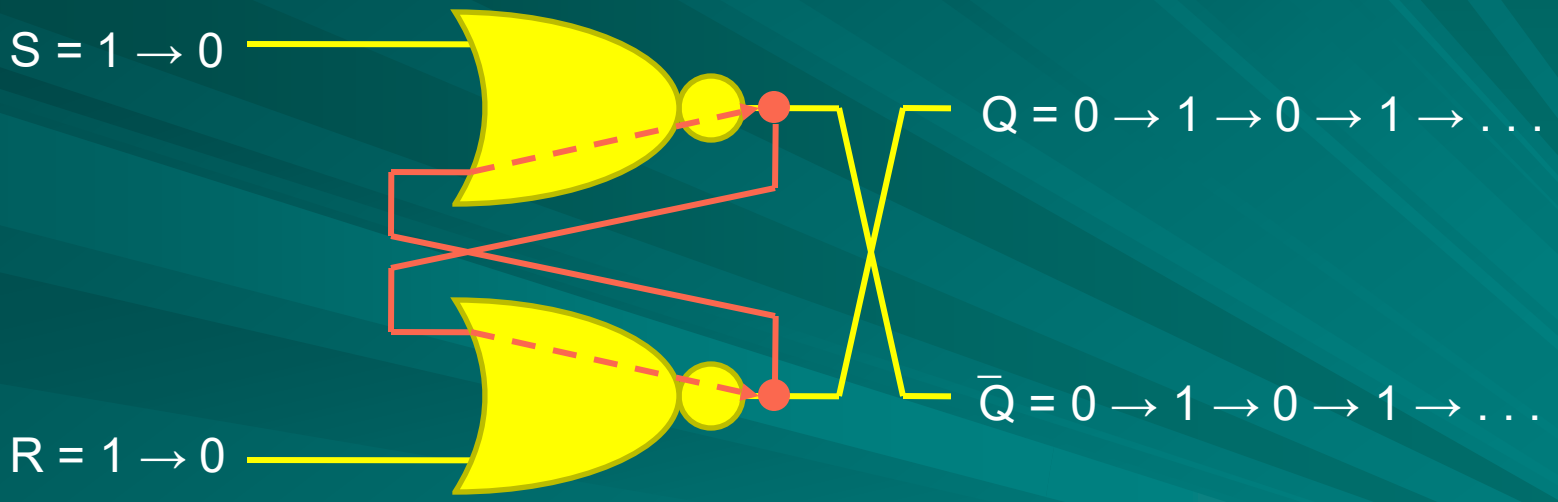
The “Illegal” State



Loop is broken in two places and inconsistent values inserted.

“Illegal” State Cannot Be Stored

Assume two gates have equal delays.



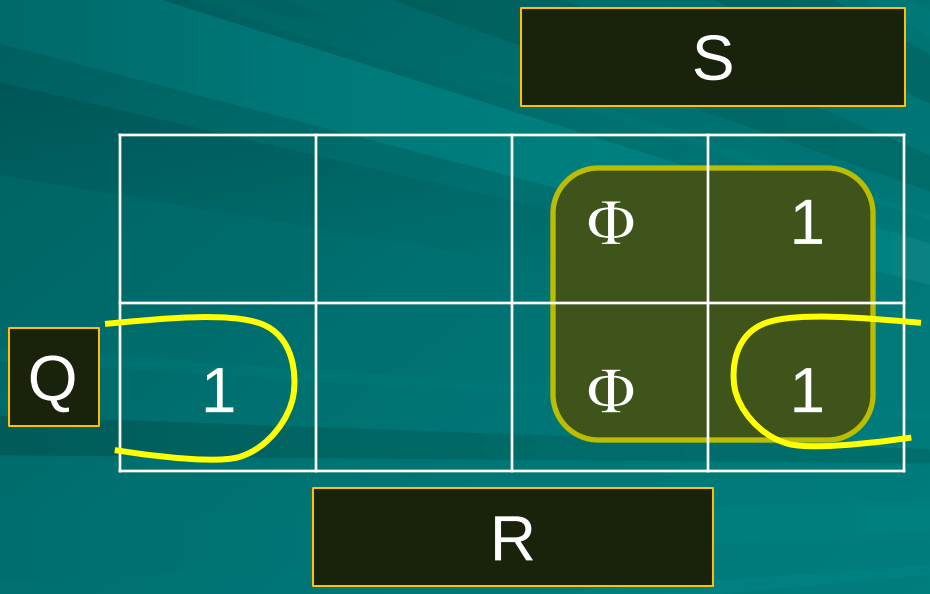
Output oscillates with a period of loop delay. For unequal gate delays, faster gate will settle to 1 *and slower gate to* 0*. This is known as RACE CONDITION.*

Excitation Table of SR Latch

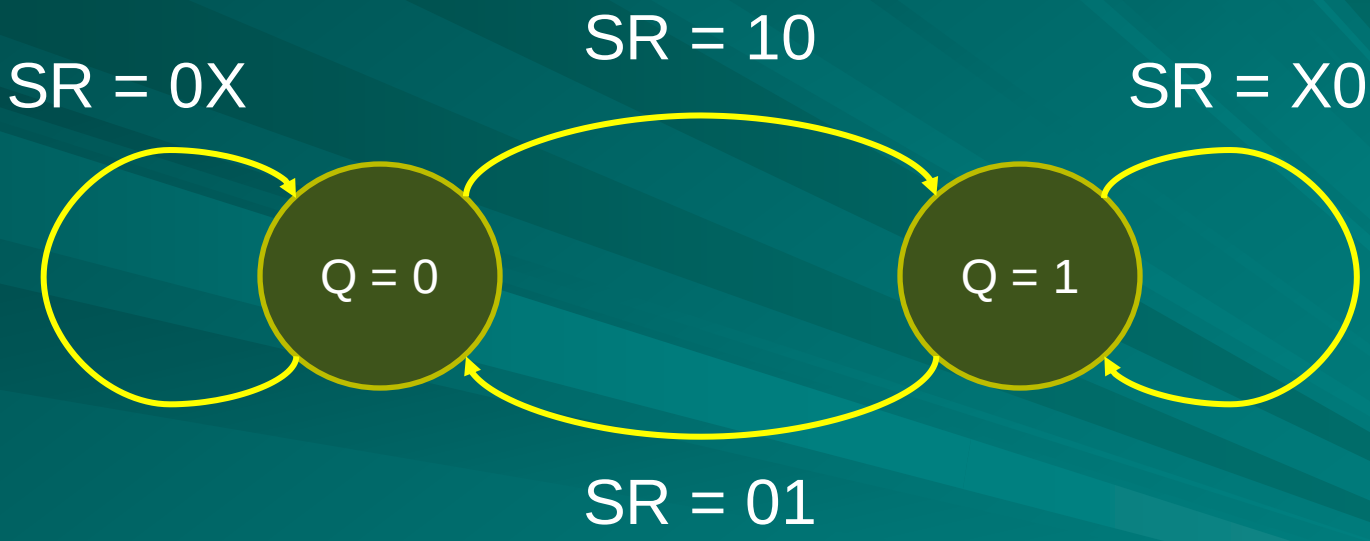
Excitation inputs		Present state	Next state	Name of State
S	R	Q	Q*	
0	0	0	0	Store
0	0	1	1	
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	Illegal	Race condition
1	1	1	Illegal	

Characteristic Equation for SR Latch

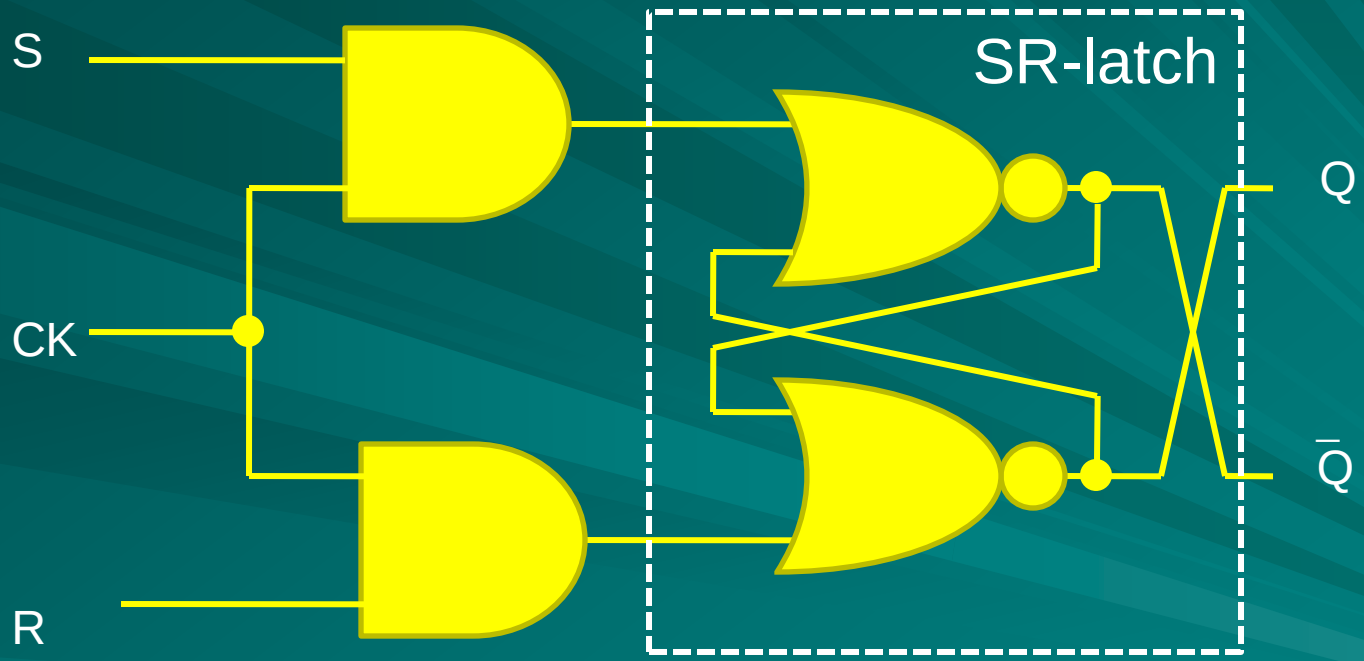
- Next-state function:
  - Treat illegal states as don't care
  - Minimize using Karnaugh map
- Characteristic equation, $Q^* = S + \bar{R}Q$



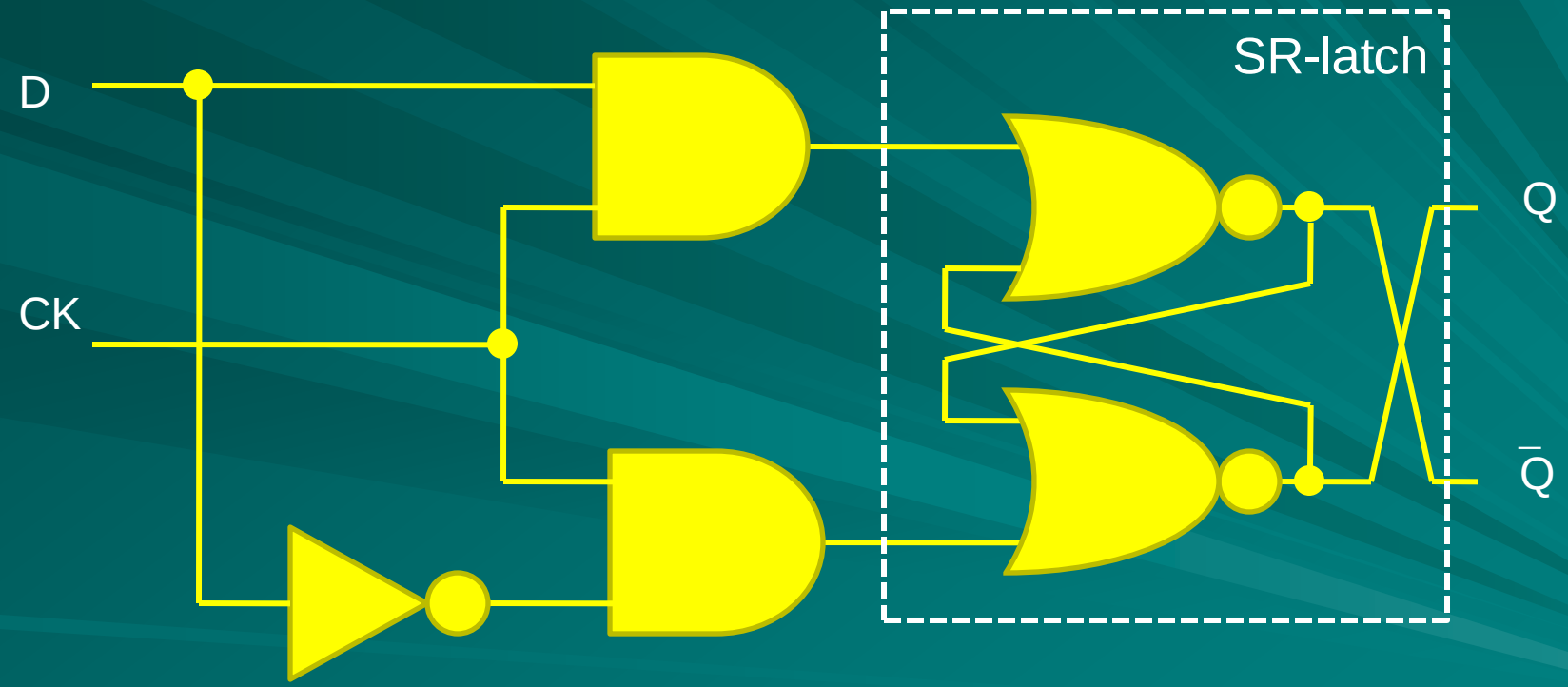
State Diagram of SR Latch



Clocked SR Latch



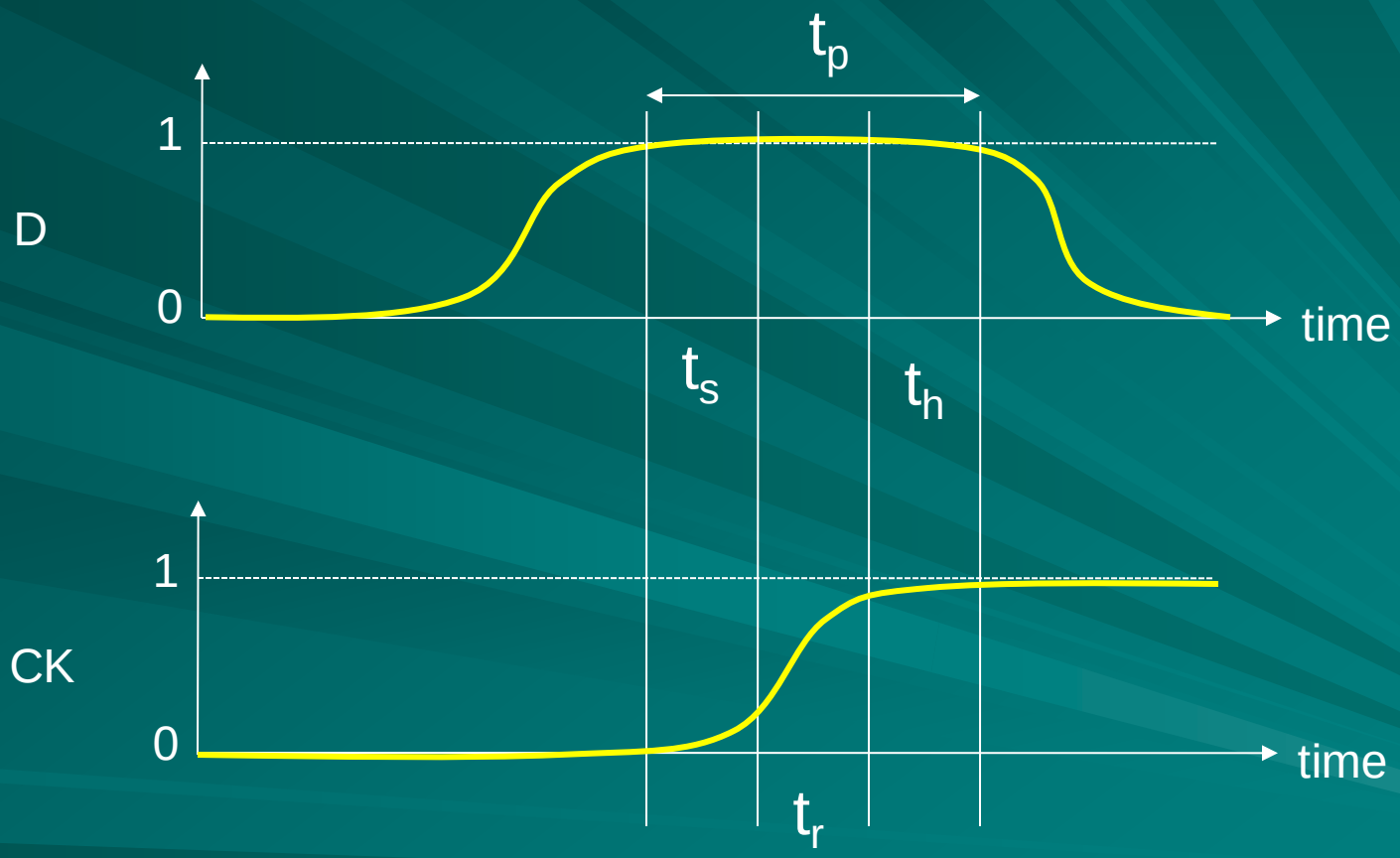
Delay Latch or D-Latch



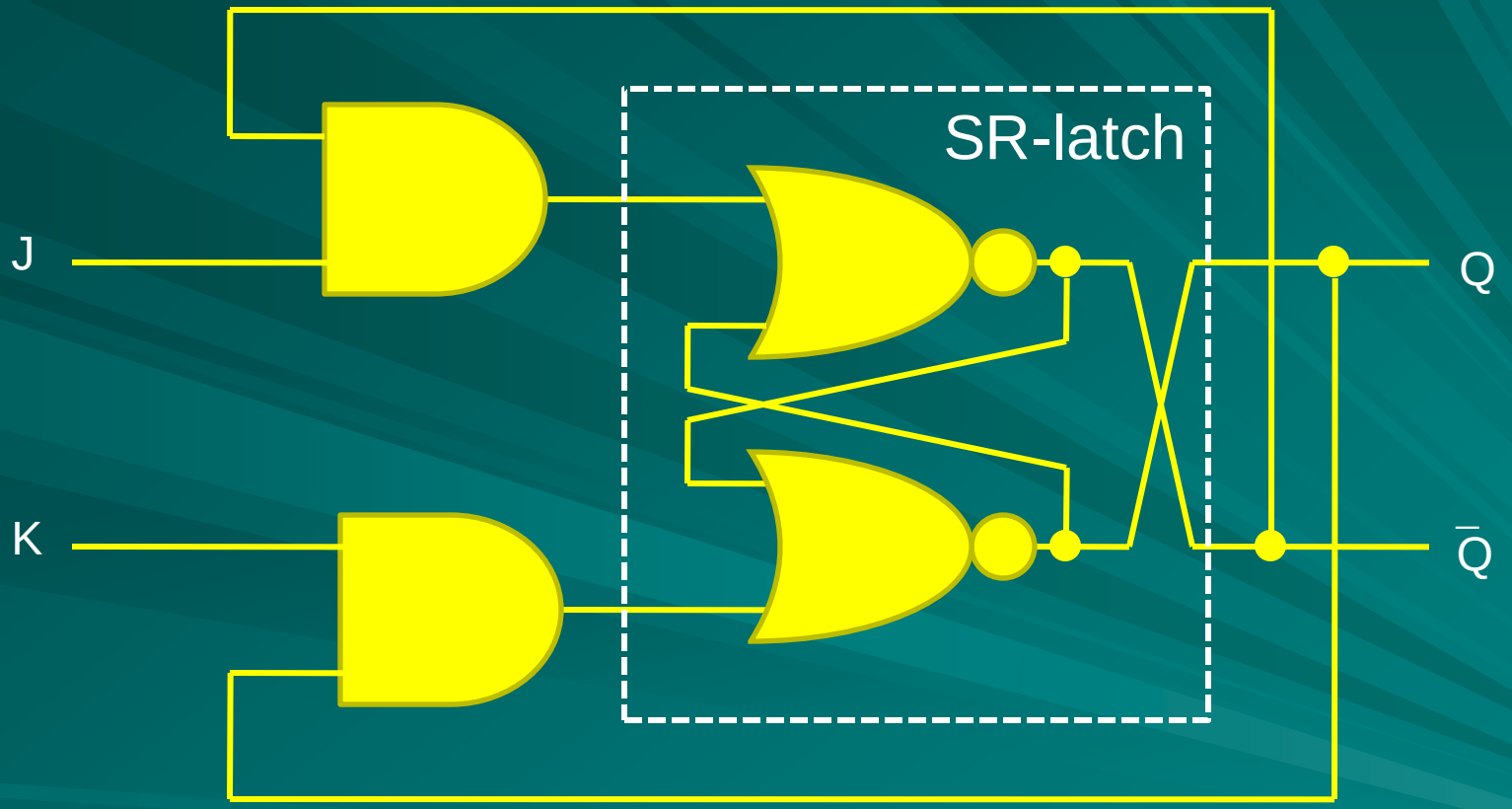
Setup and Hold Times of Latch

- Signals are synchronized with respect to clock (CK).
- Operation is ***level-sensitive***:
  - CK = 1 allows data (D) to pass through
  - CK = 0 holds the value of Q, ignores data (D)
- *Setup time* is the interval before the clock transition during which data (D) should be stable (not change). This will avoid any possible race condition.
- *Hold ti*me is the interval after the clock transition during which data should not change. This will avoid data from latching incorrectly.

Latch Inputs



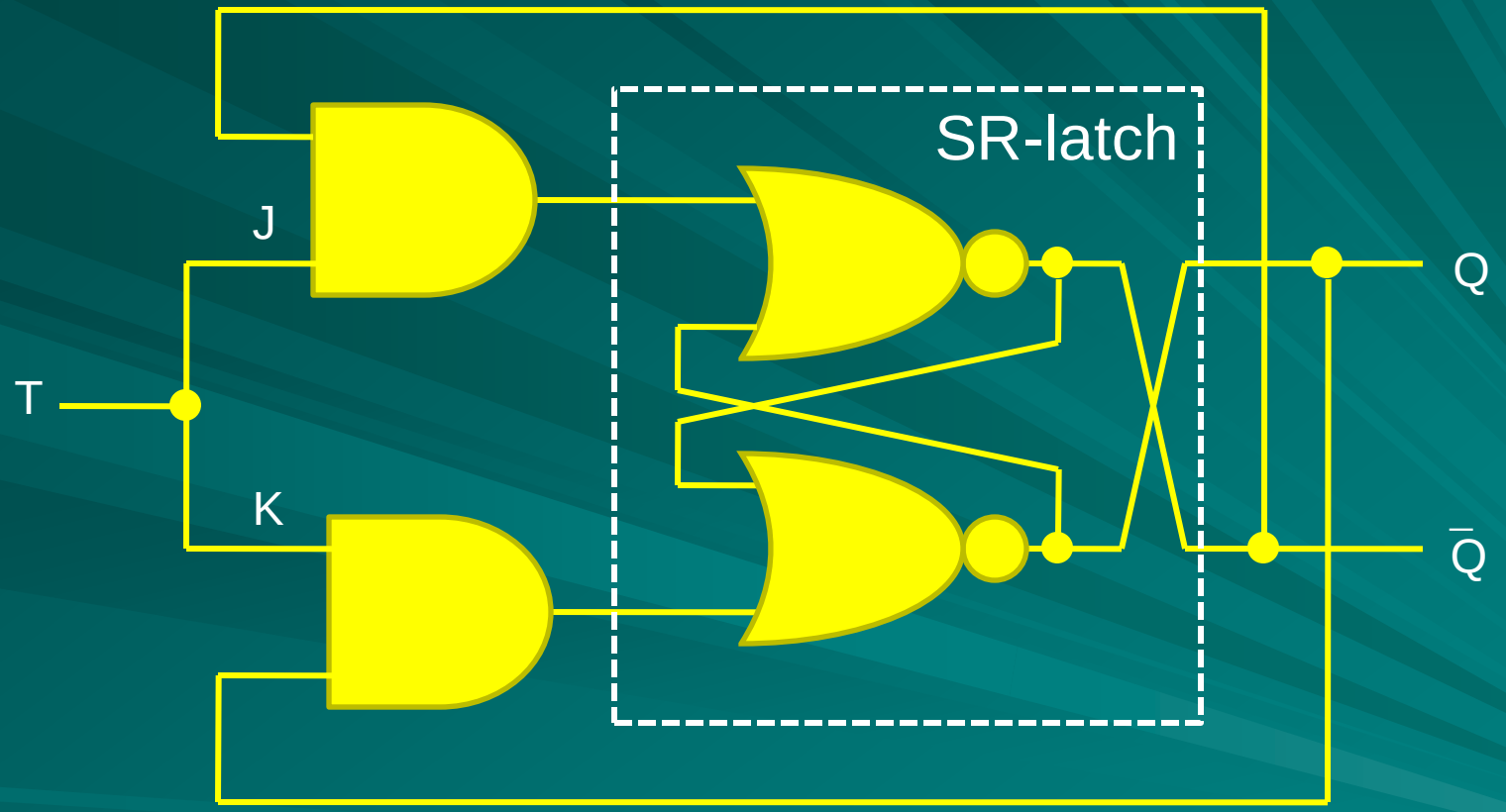
JK-Latch



Characteristic Equation, $Q = J\ \bar{Q}^* + \bar{K}\ Q^*$

Where Q = present state, Q* = previous state

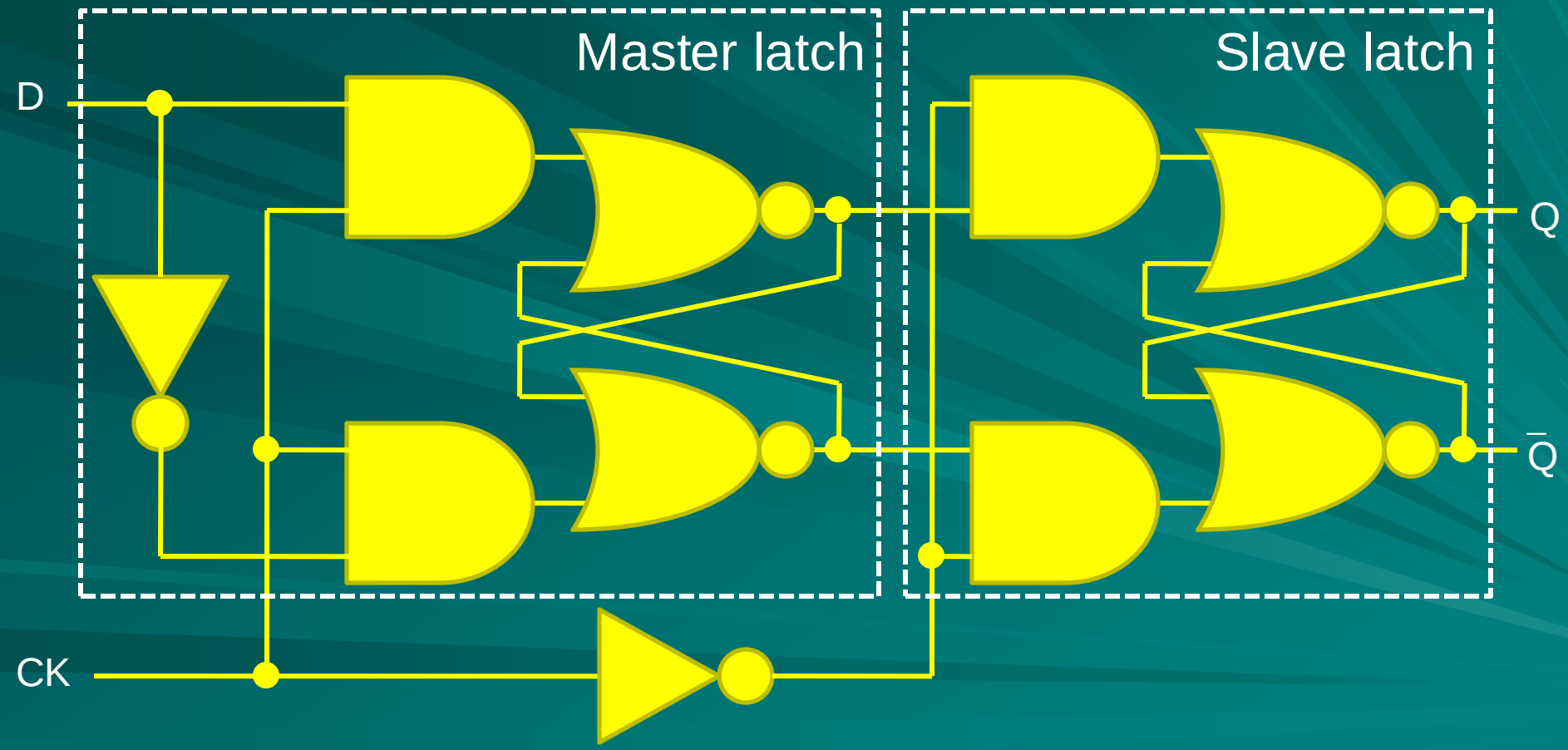
T-Latch (Toggle Latch)



Characteristic Equation, $Q = T\ \bar{Q}^* + \bar{T}\ Q^*$

Where Q = present state, Q* = previous state

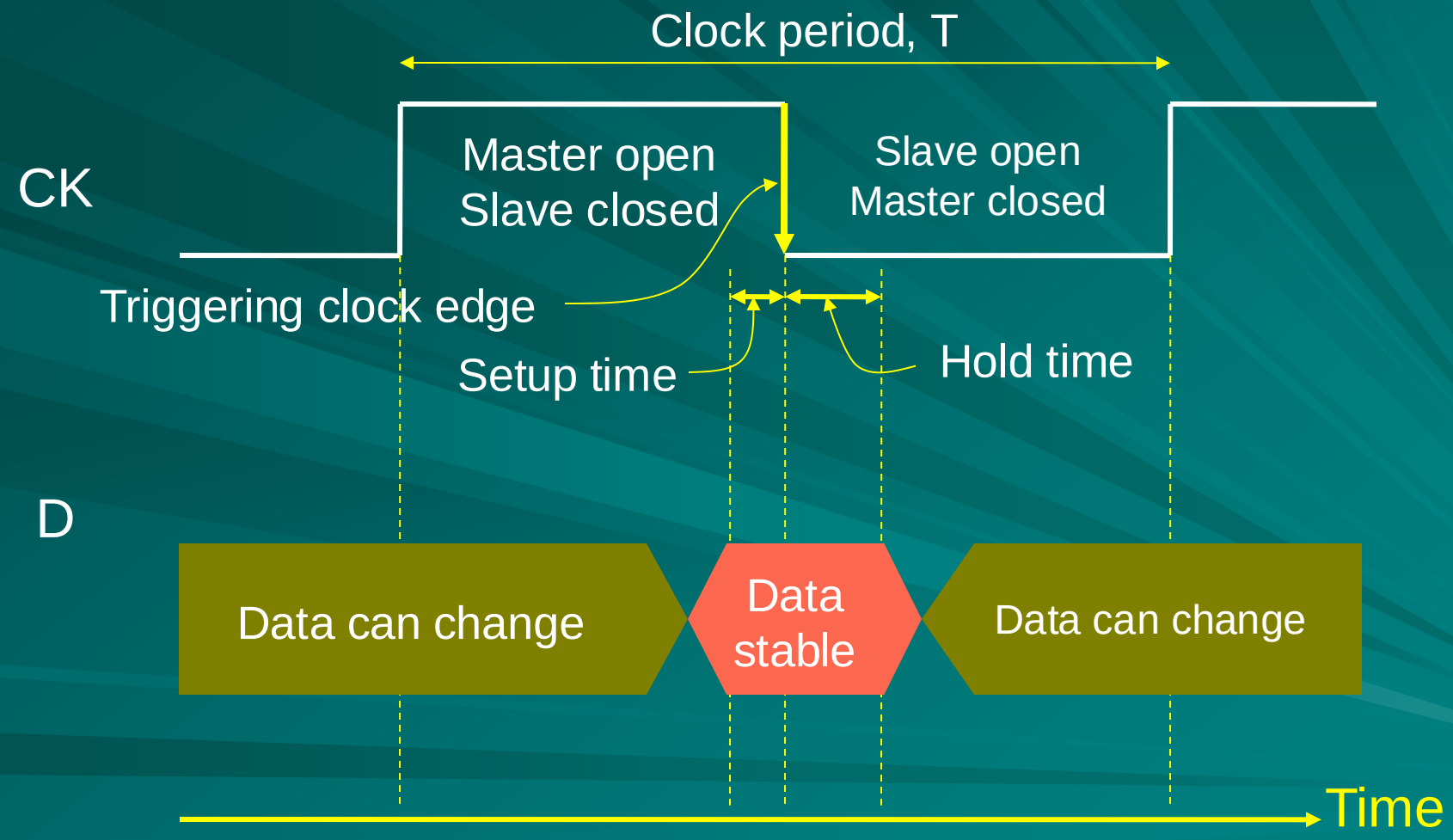
Master-Slave D-Flip-Flop



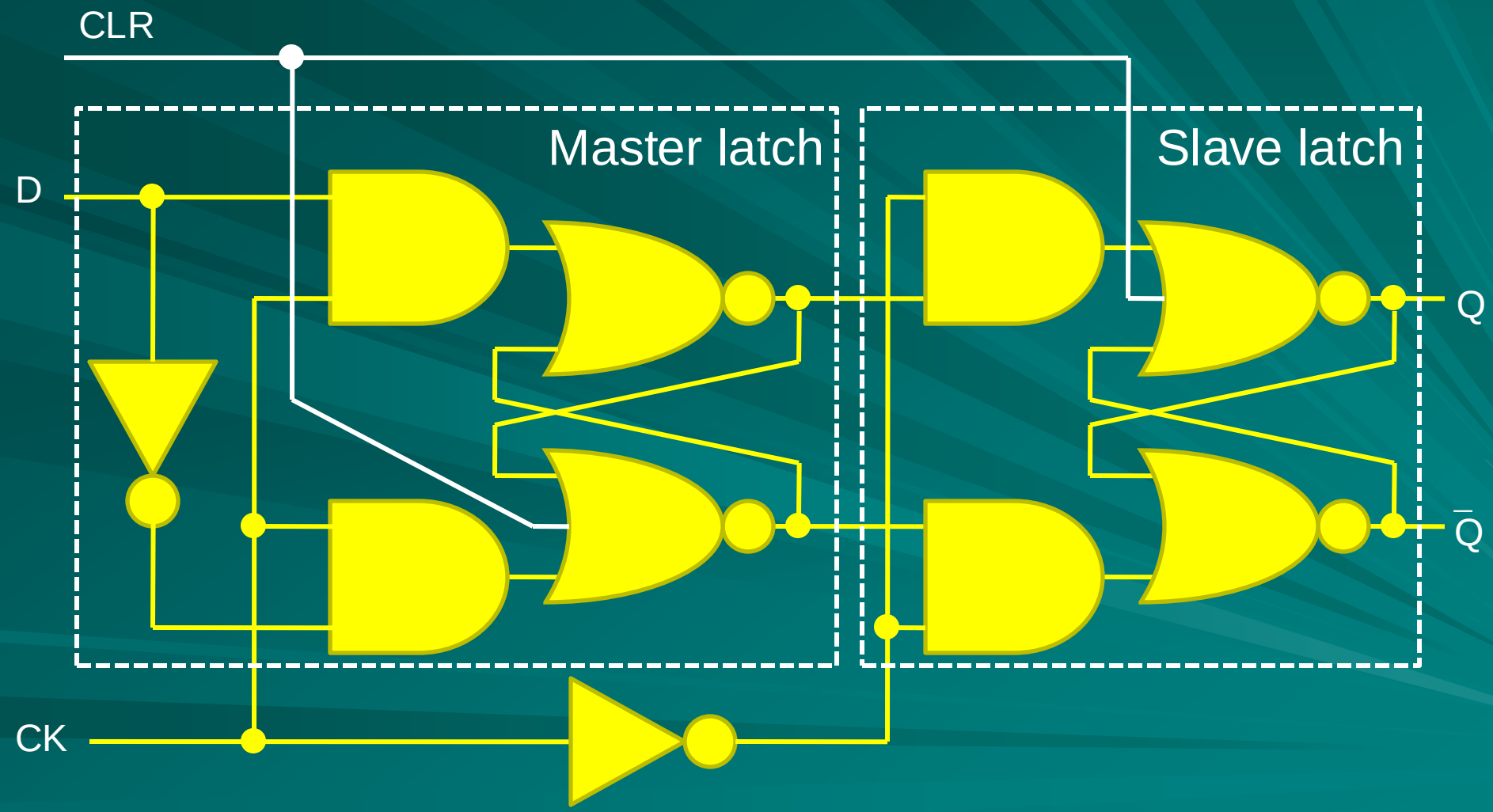
Master-Slave D-Flip-Flop

- Uses two level-sensitive clocked D-latches.
- Transfers data (D) with one clock period delay.
- Operation is ***edge-triggered***:
  - Negative edge-triggered, CK = 1→0, Q = D (previous slide)
  - Positive edge-triggered, CK = 0→1, Q = D

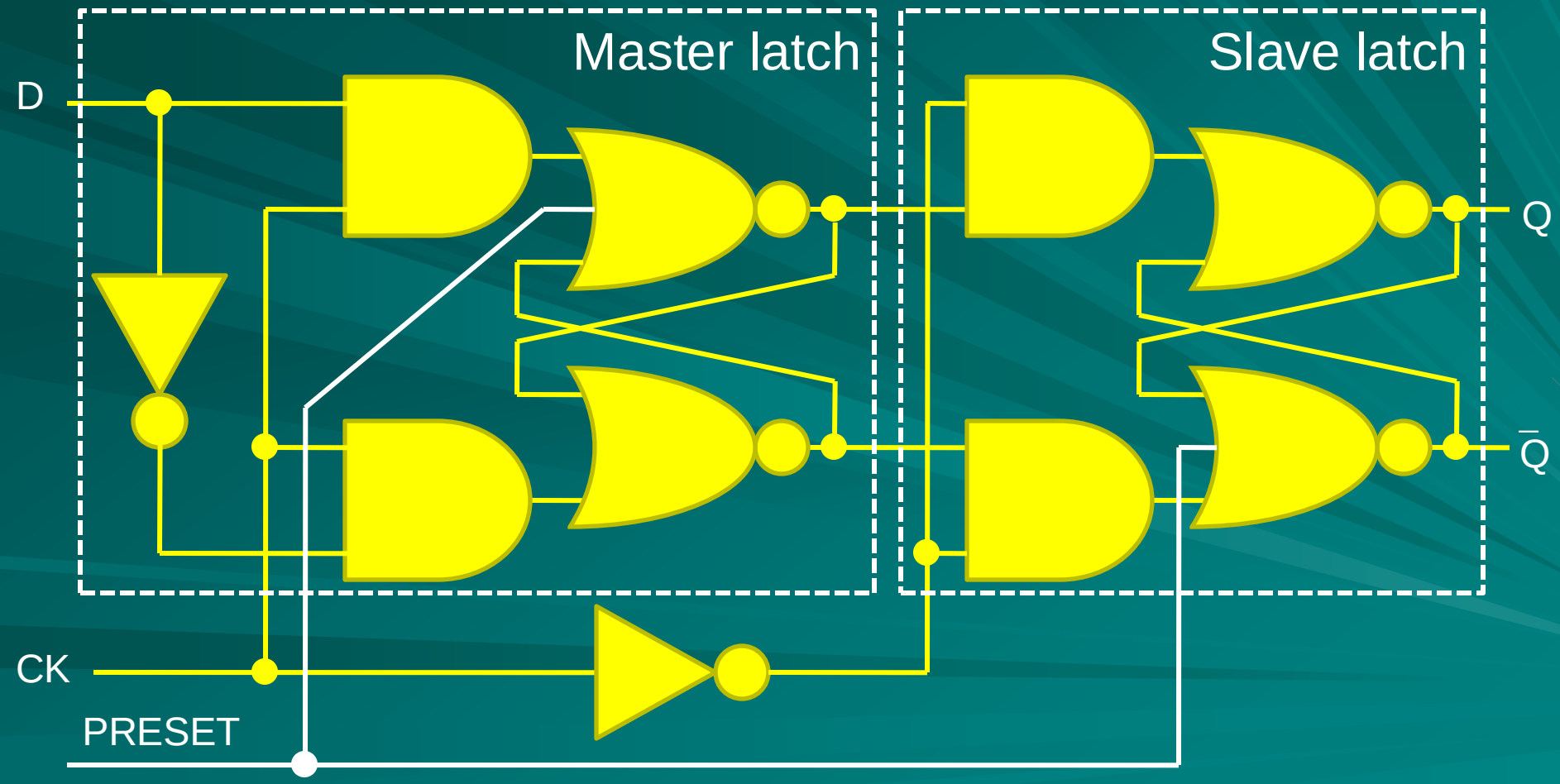
Negative-Edge Triggered D-Flip-Flop



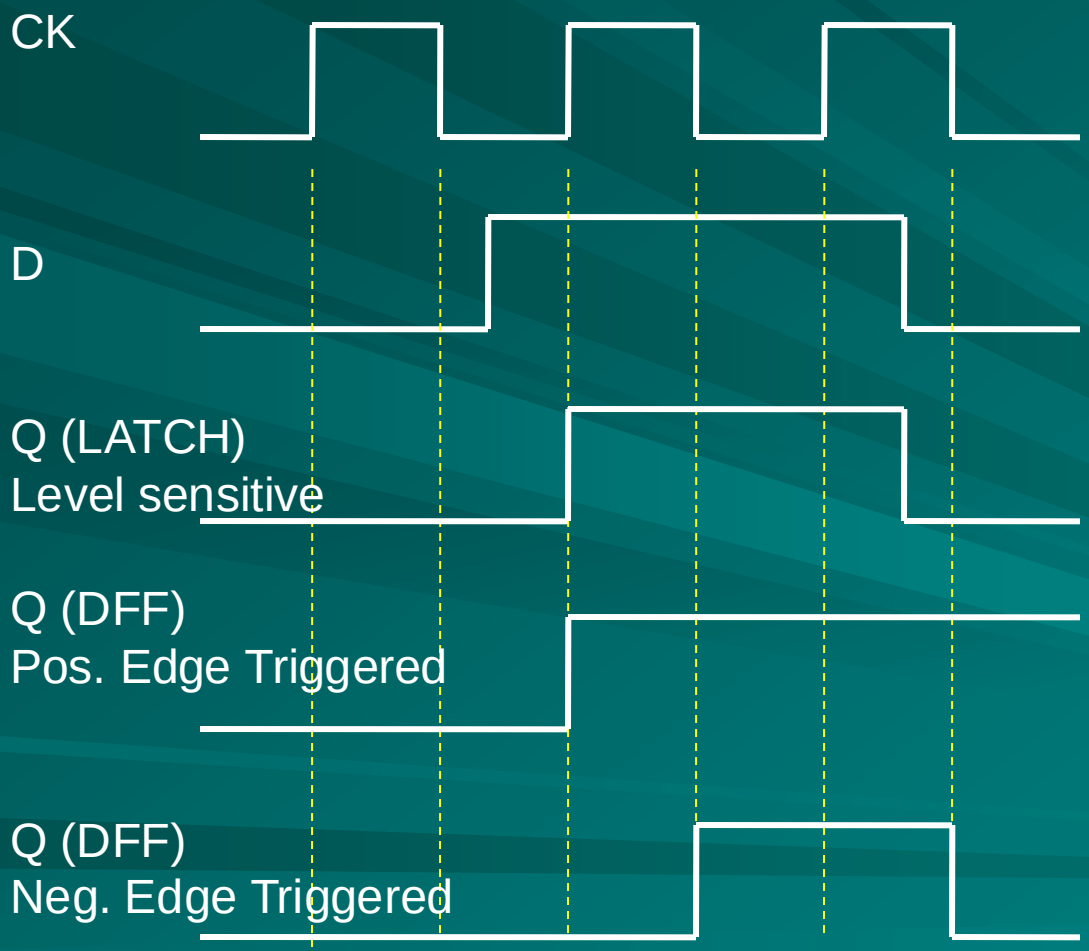
D-Flip-Flop With CLEAR

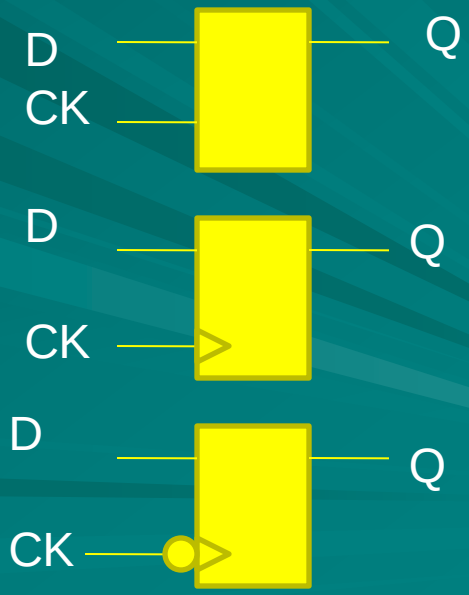


D-Flip-Flop With PRESET



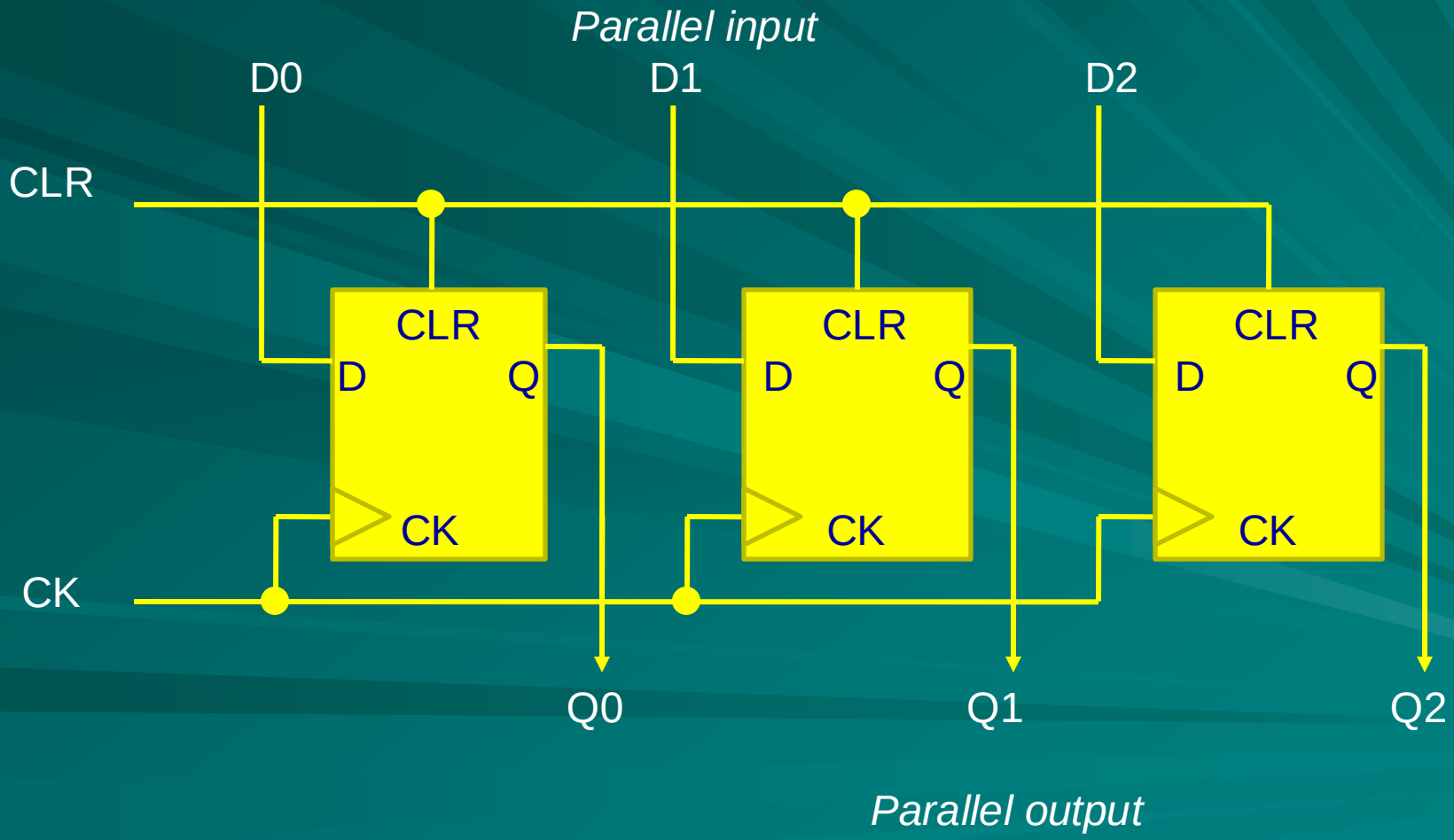
Symbols for Latch and D-Flip-Flops





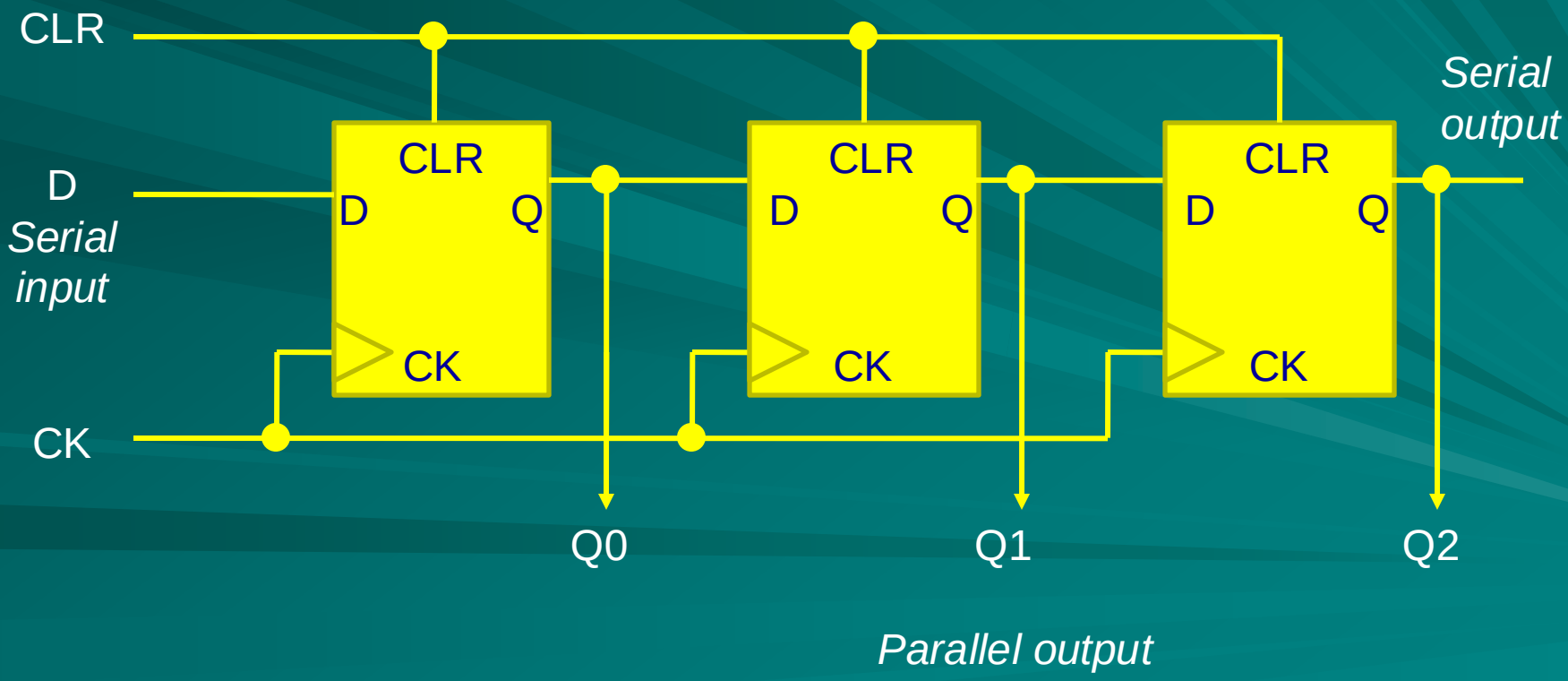
Register (3-Bit Example)

- Stores parallel data



Shift Register (3-Bit Example)

- Stores serial data (parallel output)
- Delays data (serial output)



Digital Logic Circuits

Finite State Machines (FSM) (Chapter 7-10)

Two Types of Digital Circuits

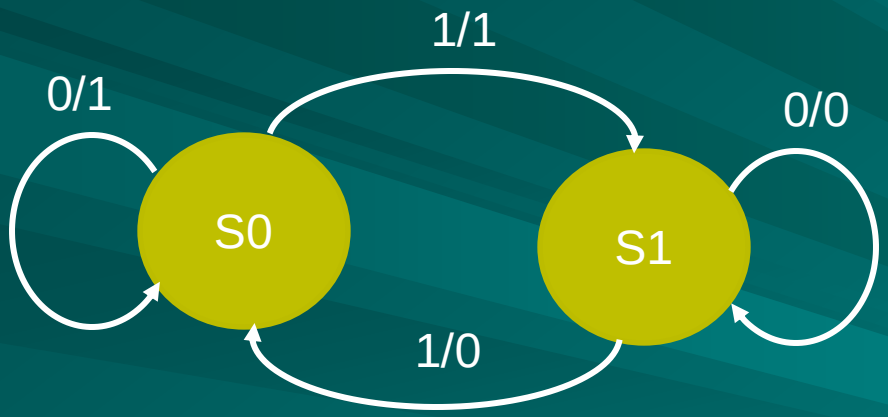
1. Output depends uniquely on inputs:
   - Contains only logic gates, AND, OR, . . .
   - No feedback interconnects
2. Output depends on inputs and memory:
   - Contains logic gates, latches and flip-flops
   - May have feedback interconnects
   - Contents of flip-flops define internal state; N flip-flops provide 2^N states; finite memory means finite states, hence the name “finite state machine (FSM)”.
   - Clocked memory – synchronous FSM
   - No clock – asynchronous FSM

Textbook Organization

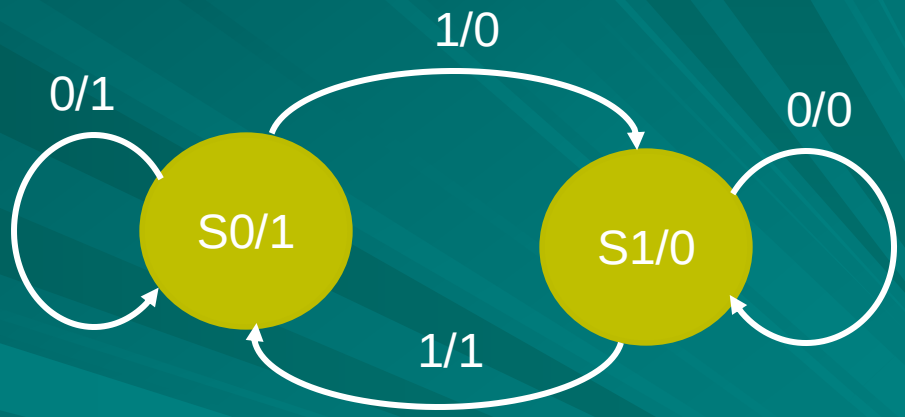
- Chapter 6: Sequential devices – latches, flip-flops.
- Chapter 7: Modular sequential logic – registers, shift registers, counters.
- Chapter 8: Specification and analysis of FSM.
- Chapter 9: Synchronous (clocked) FSM design.
- Chapter 10: Asynchronous (pulse mode) FSM design.

Mealy and Moore FSM

- Mealy machine: Output is a function of input and the state.
- Moore machine: Output is a function of the state alone.



Mealy machine



Moore machine

G. H. Mealy, “A Method for Synthesizing Sequential Circuits,” *Bell Systems Tech. J.*, vol. 34, pp. 1045-1079, September 1955.
E. F. Moore, “Gedanken-Experiments on Sequential Machines,” *Annals of Mathematical Studies*, no. 34, pp. 129-153 ,1956, Princeton Univ. Press, NJ.

Example 8.17: Robot Control

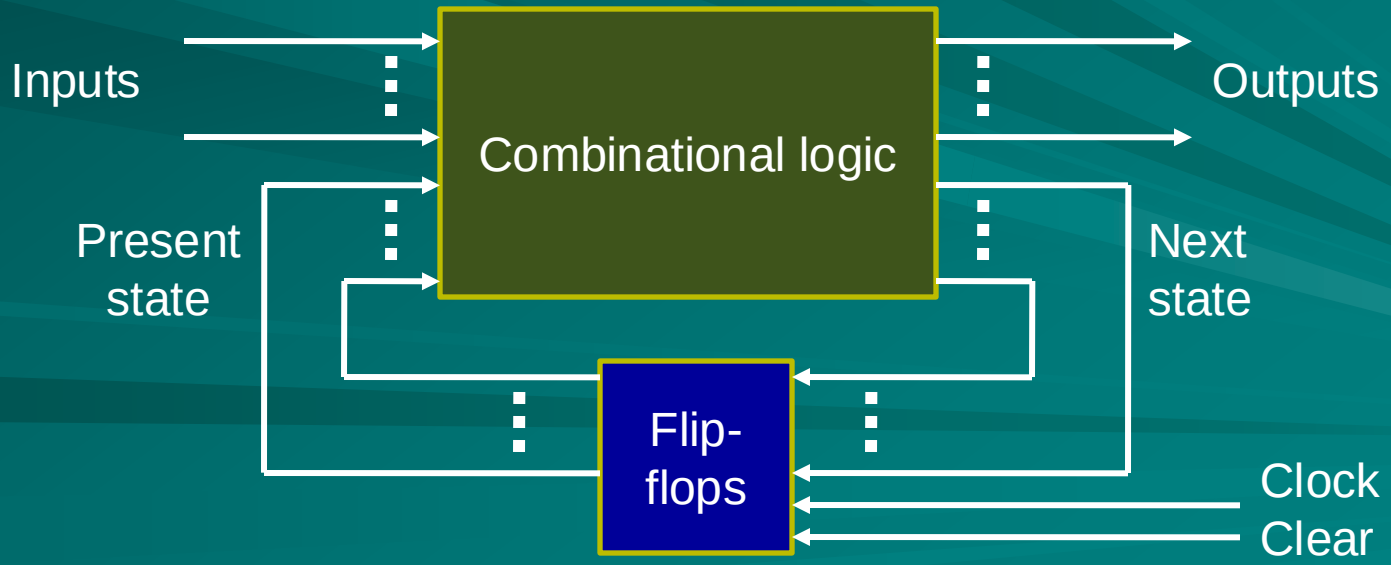
- A robot moves in straight line, encounters obstacle and turns right or left until path is clear; on successive obstacles right and left turn strategies are used.
- Define input: One bit
  - X = 0, no obstacle
  - X = 1, an obstacle encountered
- Define outputs: Two bits to represent three possible actions.
  - Z1, Z2 = 00 no turn
  - Z1, Z2 = 01 turn right by a predetermined angle
  - Z1, Z2 = 10 turn left by a predetermined angle
  - Z1, Z2 = 11 *output not used*

Example 8.17: Robot Control (Continued . . . 2)

- Because turning strategy depends on the action for the previous obstacle, the robot must remember the past.
- Therefore, we define internal memory states:
  - State A = no obstacle detected, last turn was left
  - State B = obstacle detected, turning right
  - State C = no obstacle detected, last turn was right
  - State D = obstacle detected, turning left

Realization of FSM

- The general hardware architecture of an FSM, known as Huffman model, consists of:
  - Flip-flops for storing the state.
  - Combinational logic to generate outputs and next state from inputs and present state.
  - Clock to synchronize state changes.
  - Initialization hardware to set the machine in prespecified state.



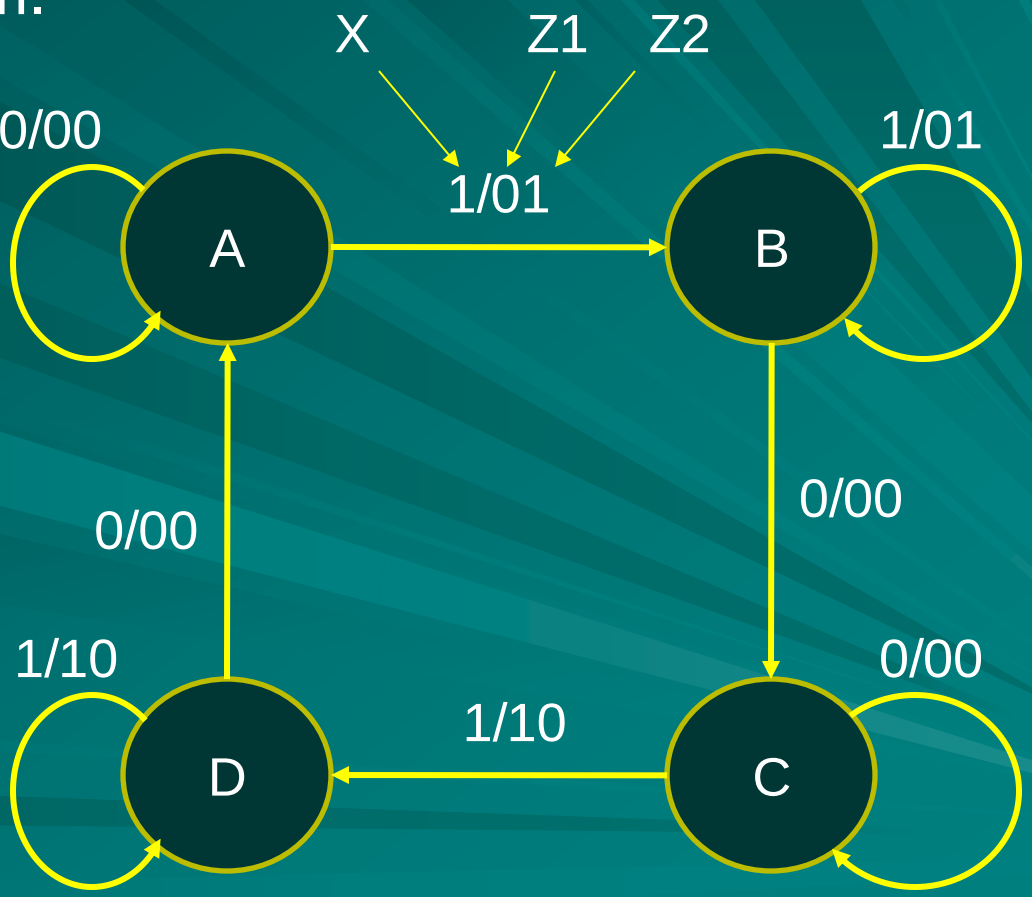
Example 8.17: Robot Control (Continued . . . 3)

- Construct state diagram.

A: no obstacle, last turn was left
B: obstacle, turn right
C: no obstacle, last turn was right
D: obstacle, turn left

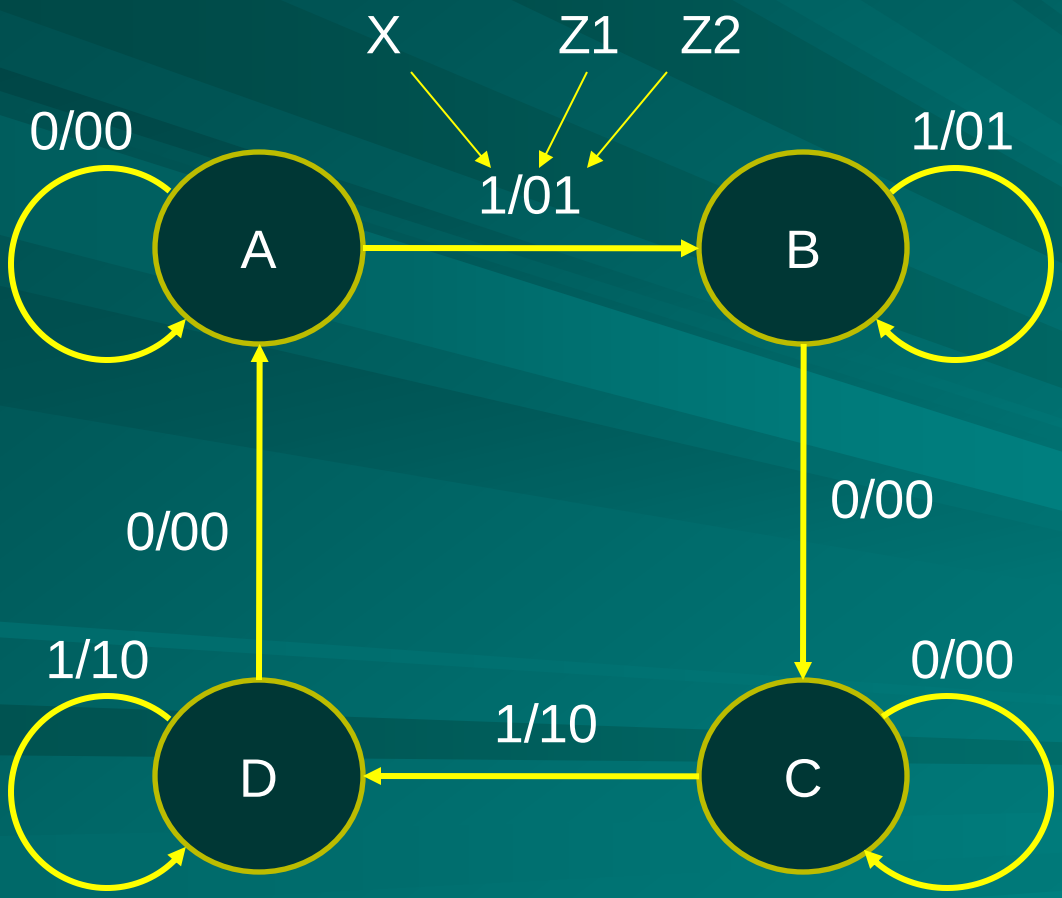
Input: X = 0, no obstacle
X = 1, obstacle

Outputs:
Z1, Z2 = 00, no turn
Z1, Z2 = 01, right turn
Z1, Z2 = 10, left turn



Example 8.17: Robot Control (Continued . . . 4)

- Construct state table.



Present state	X = 0	X = 1
A	A/00	B/01
B	C/00	B/01
C	C/00	D/10
D	A/00	D/10

Next state

Outputs Z1, Z2

Example 8.17: Robot Control (Continued . . . 5)

- State assignment: Each state is assigned a unique binary code. Need $\log_2 4 = 2$ binary state variables to represent 4 states.
- Let memory variables be Y1,Y2:

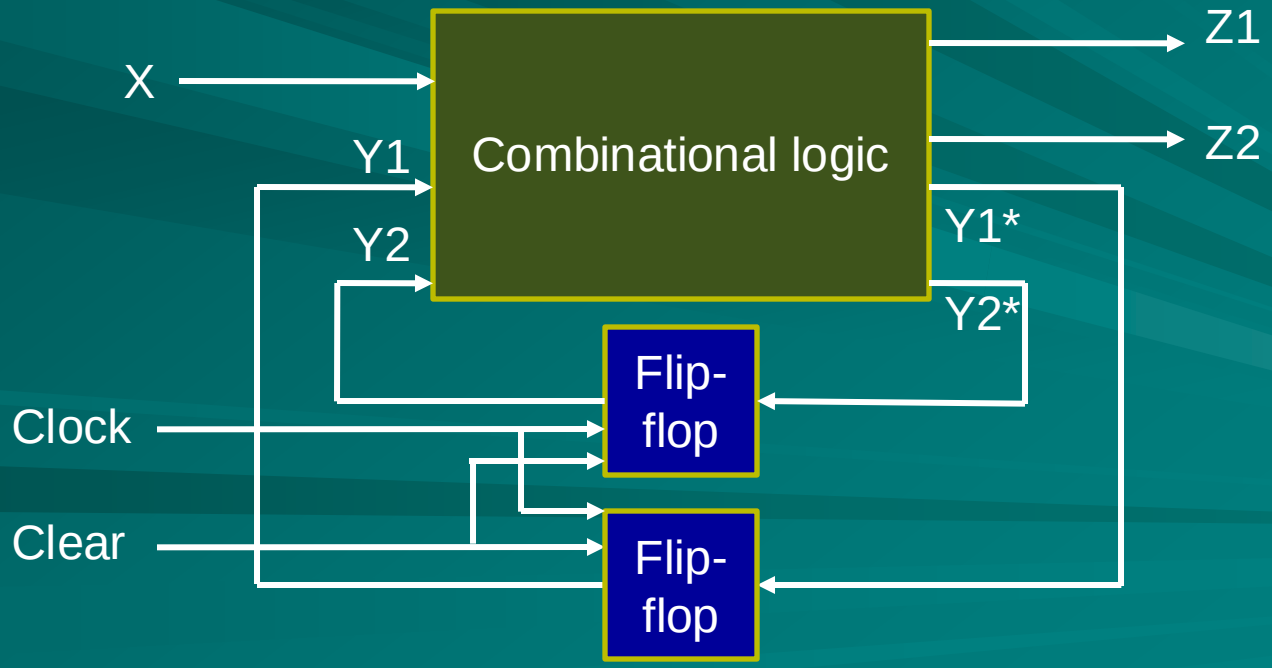
A: {Y1,Y2} = 00; B: {Y1,Y2} = 01; C: {Y1,Y2} = 11, D: {Y1,Y2} = 10

Present state \ X	0	1
A	A/00	B/01
B	C/00	B/01
C	C/00	D/10
D	A/00	D/10

Y1 Y2 \ X	0	1
00	00/00	01/01
01	11/00	01/01
11	11/00	10/10
10	00/00	10/10

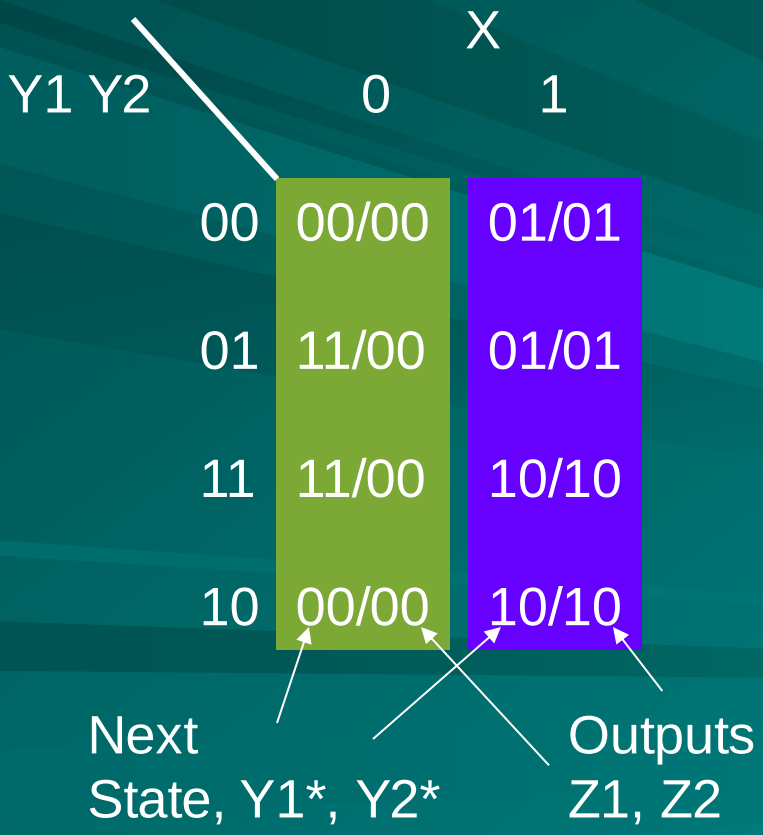
Realization of FSM

- Primary input: X
- Primary outputs: Z1, Z2
- Present state variables: Y1, Y2
- Next state variables: Y1*, Y2*



Example 8.17: Robot Control (Continued . . . 6)

- Construct truth tables for outputs, Z1 and Z2, and excitation variables, Y1 and Y2.



Input	Present state		Outputs		Next state	
X	Y1	Y2	Z1	Z2	Y1*	Y2*
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	1	0	1	0	1	0
1	1	1	1	0	1	0

Example 8.17: Robot Control (Continued . . . 7)

- Synthesize logic functions, Z1, Z2, Y1*, Y2*.

Input	Present state		Outputs		Next state	
X	Y1	Y2	Z1	Z2	Y1*	Y2*
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	1	0	1	0	1	0
1	1	1	1	0	1	0

$Z1 = XY1\ \bar{Y}2 + XY1\ Y2 = XY1$

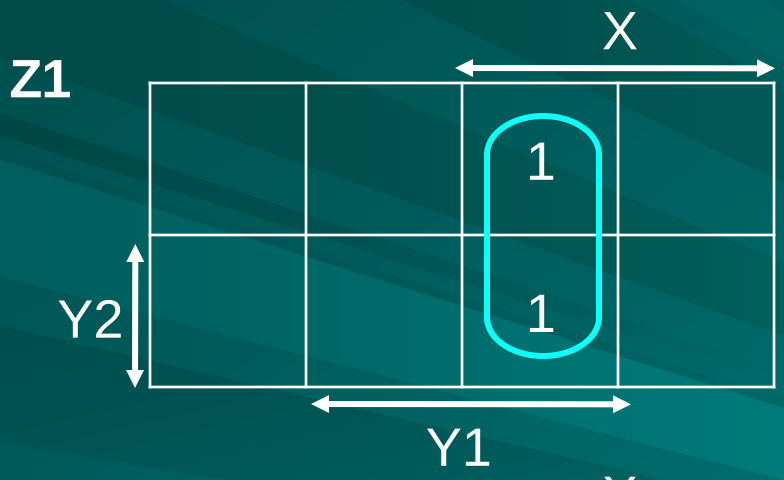
$Z2 = X\ \bar{Y}1\ \bar{Y}2 + X\ \bar{Y}1\ Y2 = X\ \bar{Y}1$

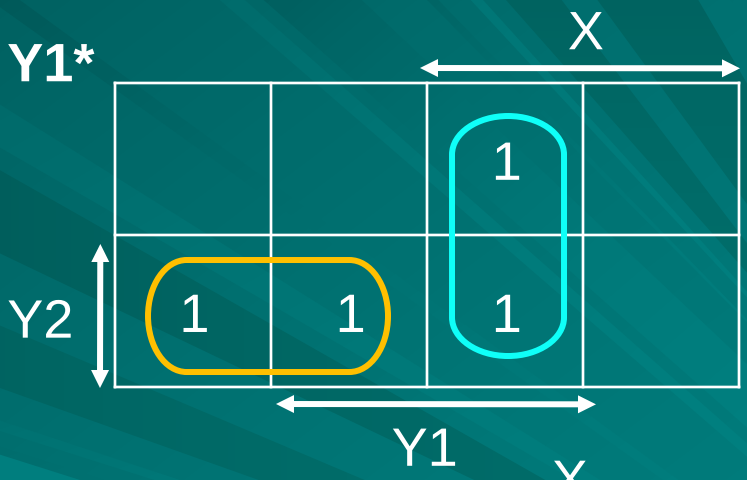
$Y1^* = \bar{X}\ \bar{Y}1\ Y2 + \ldots$

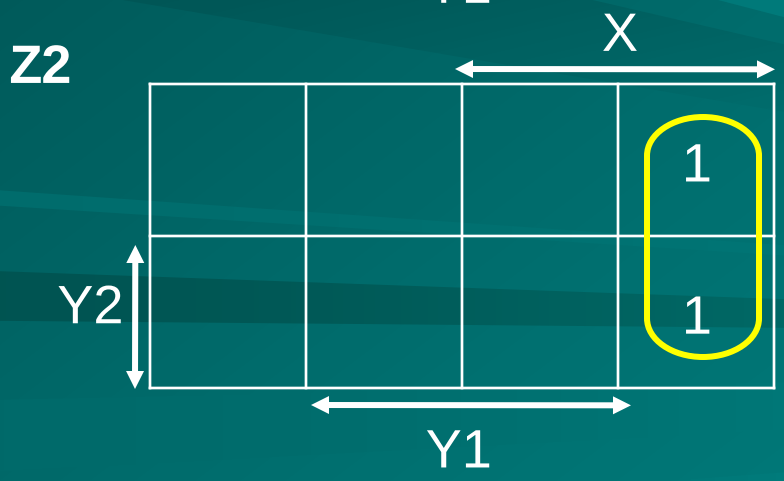
$Y2^* = \bar{X}\ \bar{Y}1\ Y2 + \ldots$

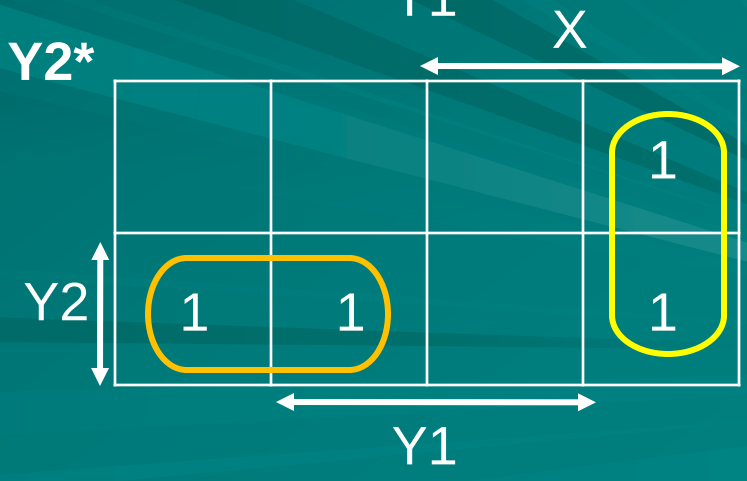
Example 8.17: Robot Control (Continued . . . 8)

- Synthesize logic functions, Z1, Z2, Y1*, Y2*.



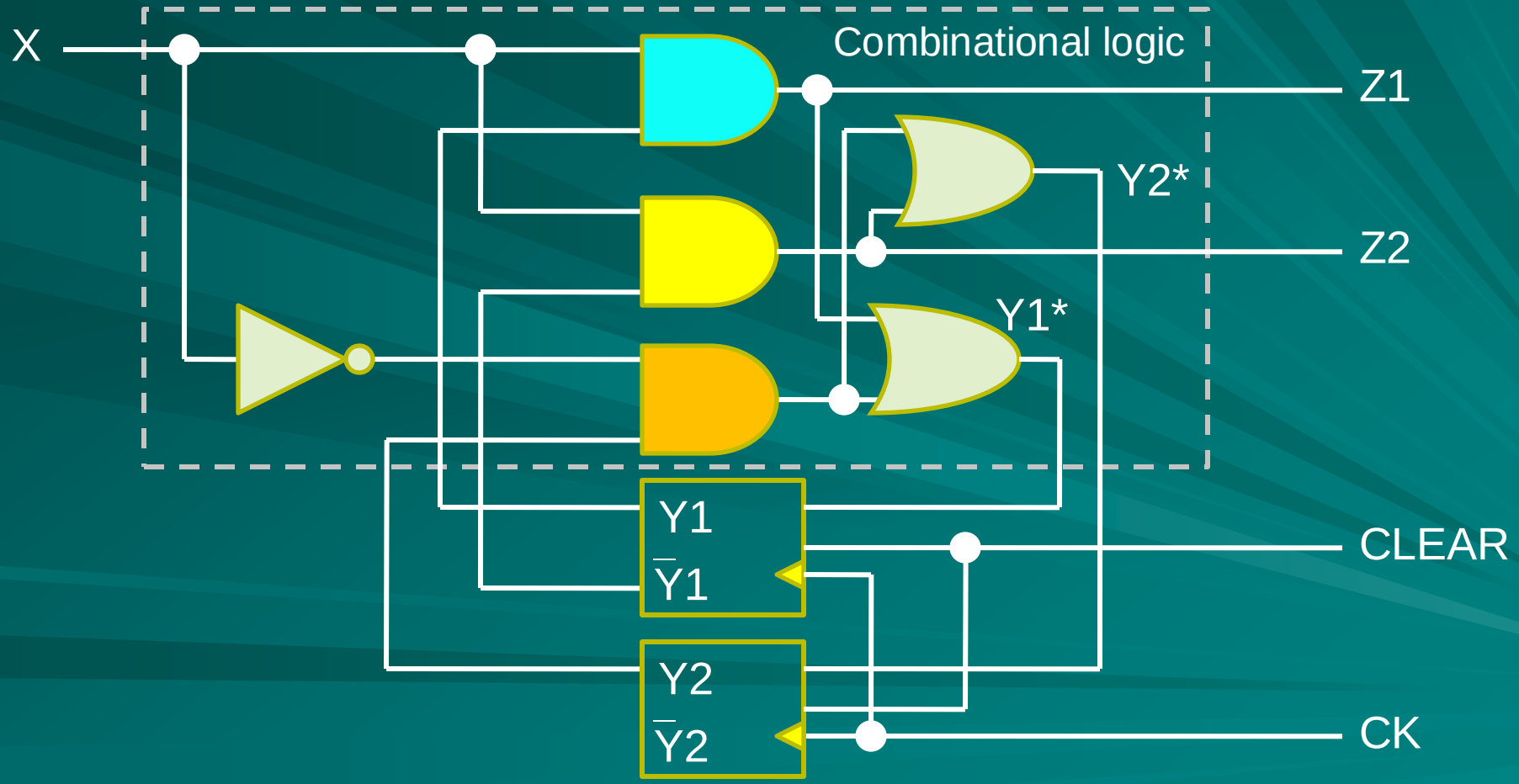






Example 8.17: Robot Control (Continued . . . 9)

- Synthesize logic and connect memory elements (flip-flops).

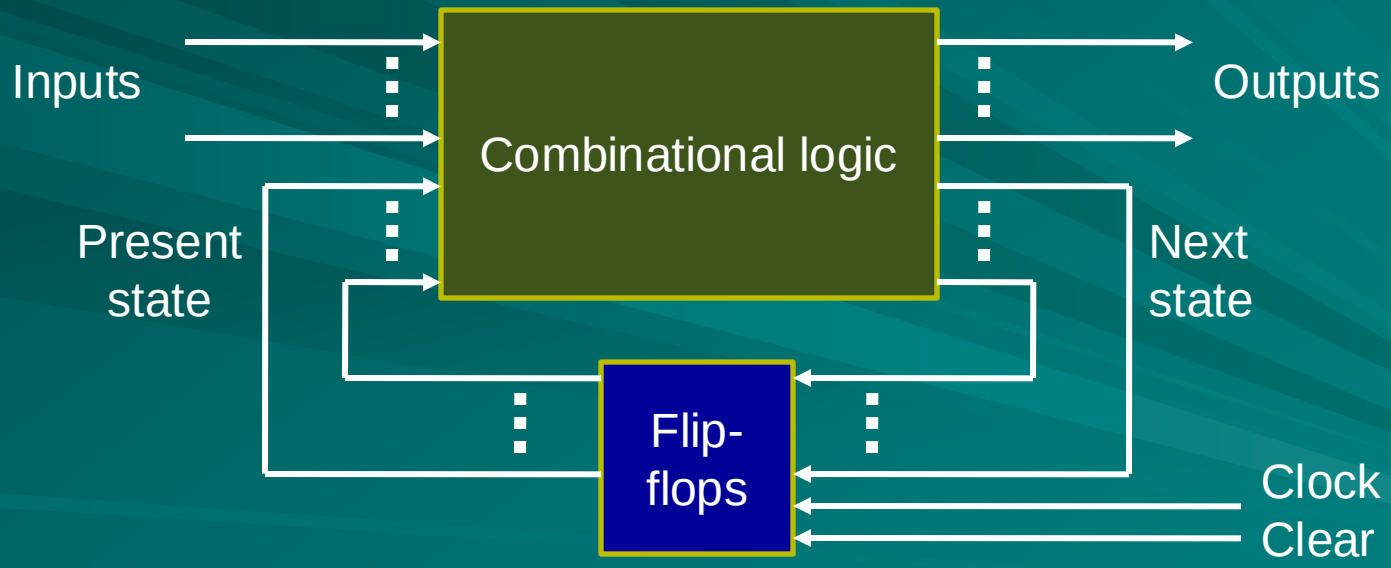


Steps in FSM Synthesis

- Examine specified function to identify inputs, outputs and memory states.
- Draw a state diagram.
- Minimize states (see Section 9.1).
- Assign binary codes to states (Section 9.4).
- Derive truth tables for state variables and output functions.
- Minimize multi-output logic circuit.
- Connect flip-flops for state variables. *Don’t forget to connect clock and clear signals.*

Architecture of an FSM

- The Huffman model, containing:
  - Flip-flops for storing the state.
  - Combinational logic to generate outputs and next state from inputs and present state.



D. A. Huffman, “The Synthesis of Sequential Switching Circuits, *J. Franklin Inst*., vol. 257, pp. 275-303, March-April 1954.

State Minimization

- An FSM contains flip-flops and combinational logic:
  - Number of flip-flops, $N_{ff} = \lceil \log_2 N_s \rceil$, N_s = #states
  - Size of combinational logic depends on state assignment.

Ceiling operator

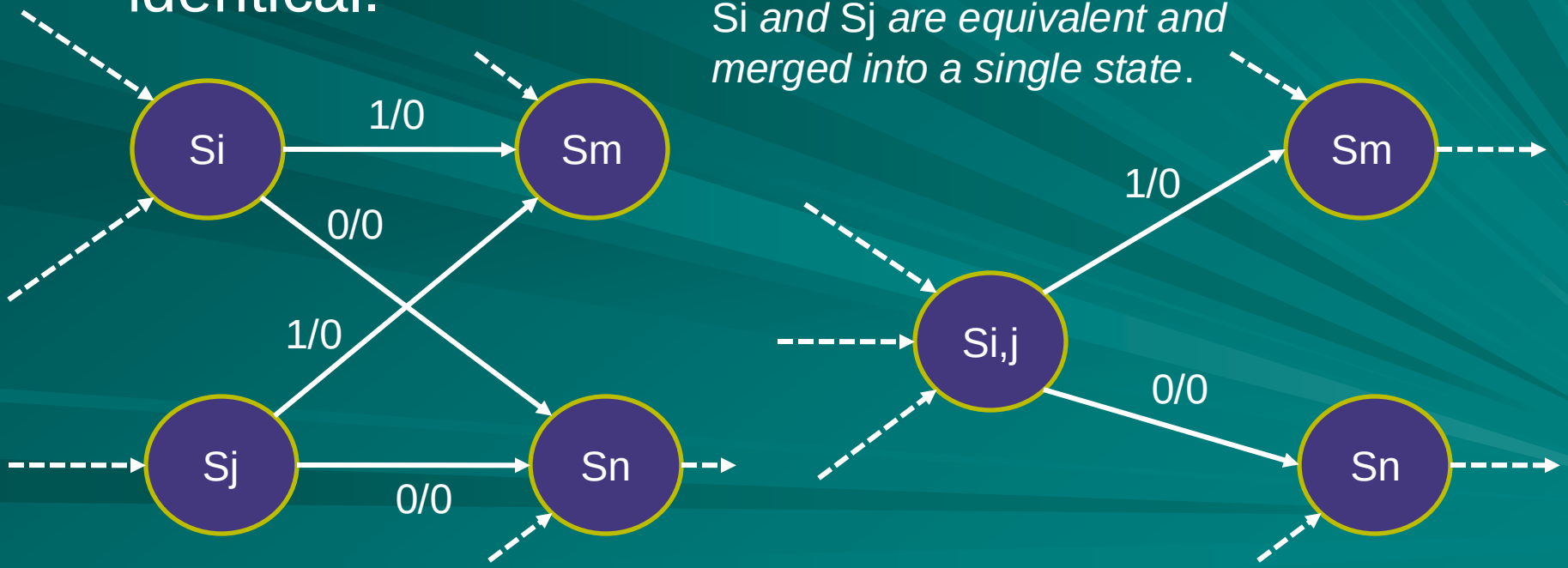
- Examples:

1. $N_s = 16$, $N_{ff} = \lceil \log_2 16 \rceil = 4$

2. $N_s = 17$, $N_{ff} = \lceil \log_2 17 \rceil = \lceil 4.0875 \rceil = 5$

Equivalent States

- Two states of an FSM are *equivalent* (or *indistinguishable*) if for each input they produce the same output and their next states are identical.



Minimizing States

- Example: States A . . . I, Inputs I1, I2, Output, Z

A and D are equivalent

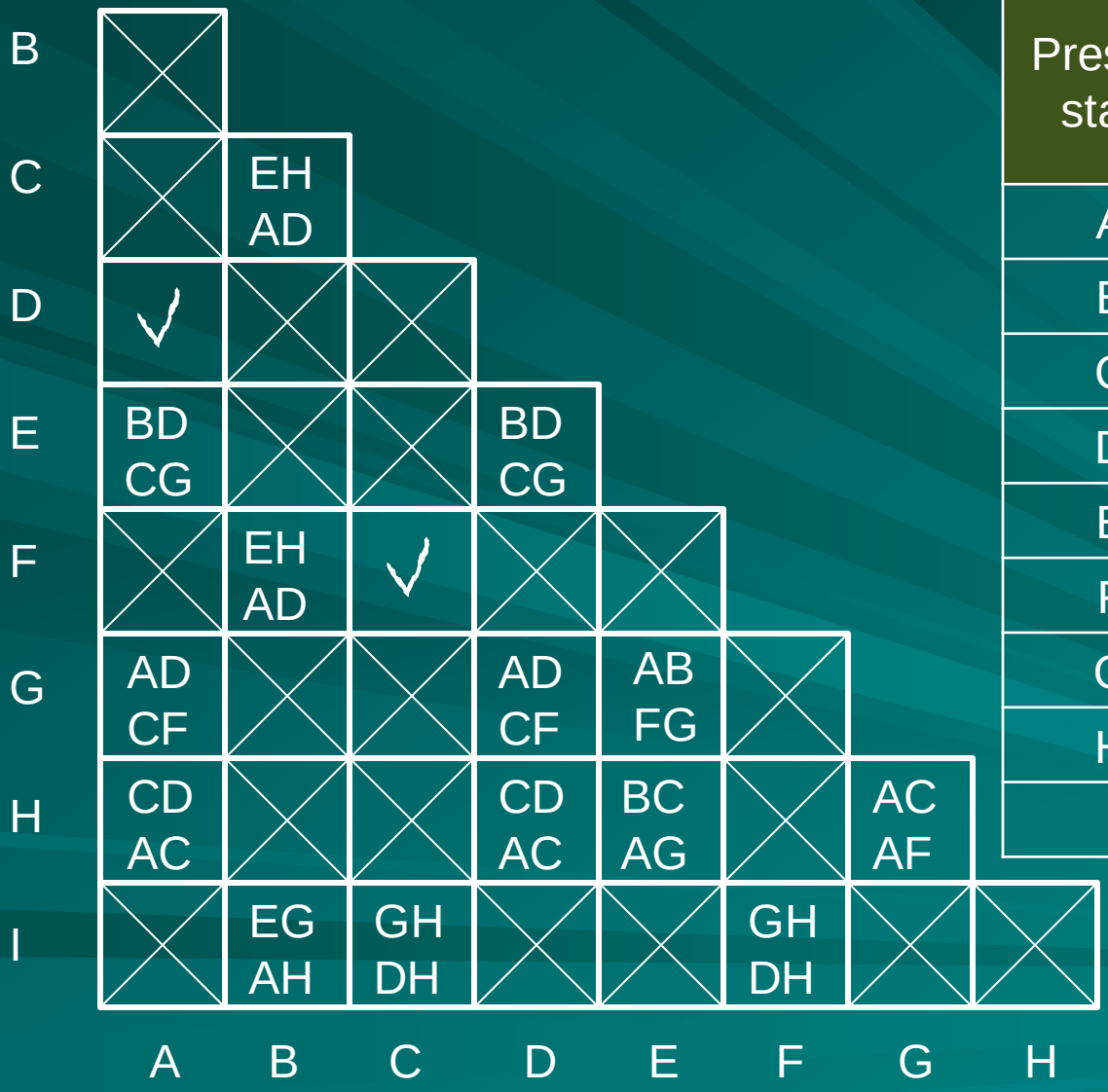
A and E produce same output

Q: Can they be equivalent?

A: Yes, if B and D were equivalent and C and G were equivalent.

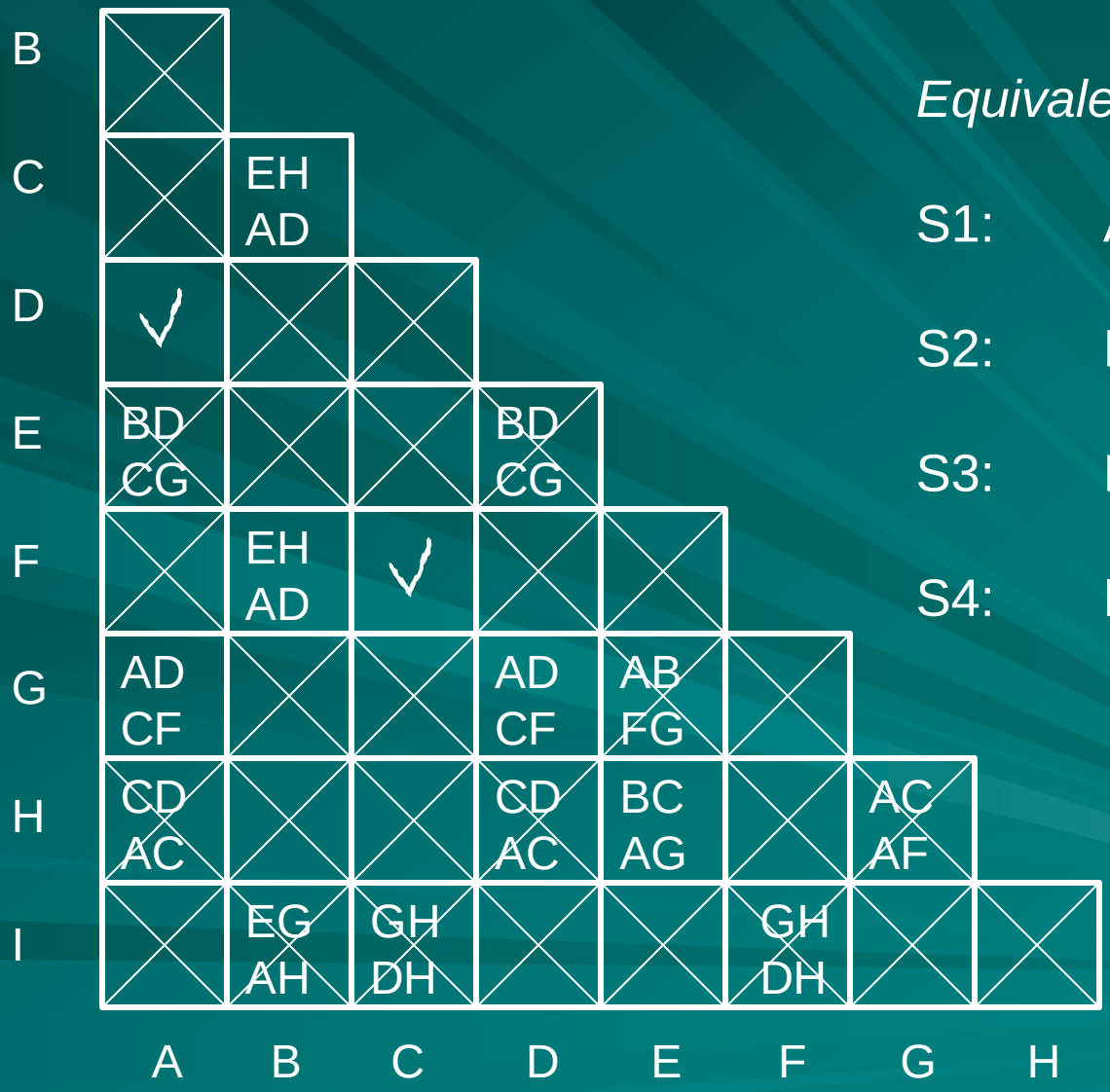
Present state	Next state, output (Z)	
	Input	
	I1	I2
A	D / 0	C / 1
B	E / 1	A / 1
C	H / 1	D / 1
D	D / 0	C / 1
E	B / 0	G / 1
F	H / 1	D /1
G	A / 0	F / 1
H	C / 0	A / 1
I	G / 1	H / 1

Implication Table Method



Present state	Next state, output (Z)	
	Input	
	I1	I2
A	D / 0	C / 1
B	E / 1	A / 1
C	H / 1	D / 1
D	D / 0	C / 1
E	B / 0	G / 1
F	H / 1	D / 1
G	A / 0	F /1
H	C / 0	A / 1
I	G / 1	H / 1

Implication Table Method (Cont.)



Equivalent states:

S1: A, D, G

S2: B, C, F

S3: E, H

S4: I

Minimized State Table

Original

Present state	Next state, output (Z)	
	Input	
	I1	I2
A	D / 0	C / 1
B	E / 1	A / 1
C	H / 1	D / 1
D	D / 0	C / 1
E	B / 0	G / 1
F	H / 1	D / 1
G	A / 0	F / 1
H	C / 0	A / 1
I	G / 1	H / 1

Minimized

Present state	Next state, output (Z)	
	Input	
	I1	I2
S1 = (A, D, G)	S1 / 0	S2 / 1
S2 = (B, C, F)	S3 / 1	S1 / 1
S3 = (E, H)	S2 / 0	S1 / 1
S4 = I	S1 / 1	S3 / 1

Number of flip-flops is reduced from 4 to 2.

State Assignment

- State assignment means assigning distinct binary patterns (codes) to states.
- N flip-flops generate 2^N codes.
- While we are free to assign these codes to represent states in any way, the assignment affects the optimality of the combinational logic.
- Rules based on heuristics are used to determine state assignment.

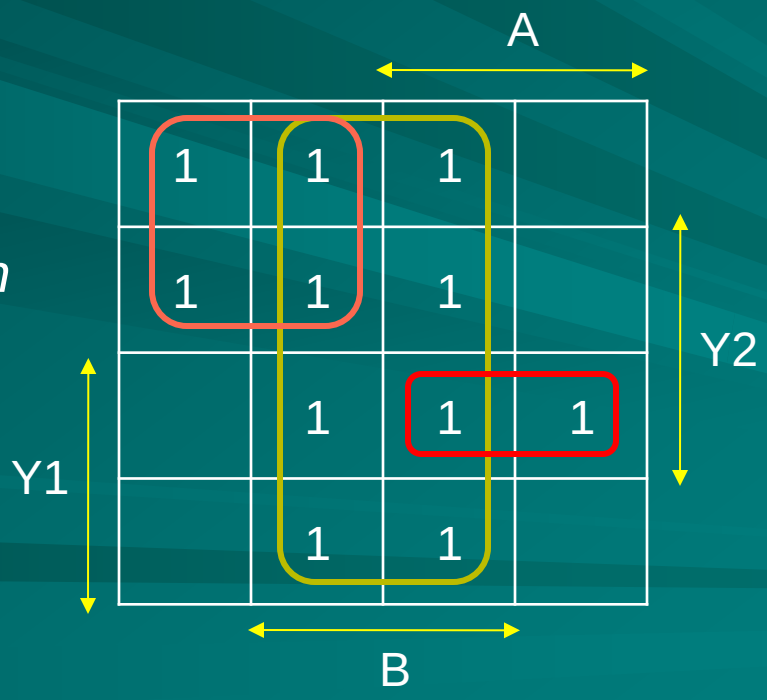
Criteria for State Assignment

- Optimize:
  - Logic gates, or
  - Delay, or
  - Power consumption, or
  - Testability, or
  - Any combination of the above
- Up to 4 or 5 flip-flops: can try all assignments and select the best.
- More flip-flops: Use an existing heuristic (one discussed next) or invent a new heuristic.

The Idea of Adjacency

- Inputs are A and B
- State variables are Y1 and Y2
- An output is F(A, B, Y1, Y2)
- A next state function is G(A, B, Y1, Y2)

Karnaugh map of output function or next state function



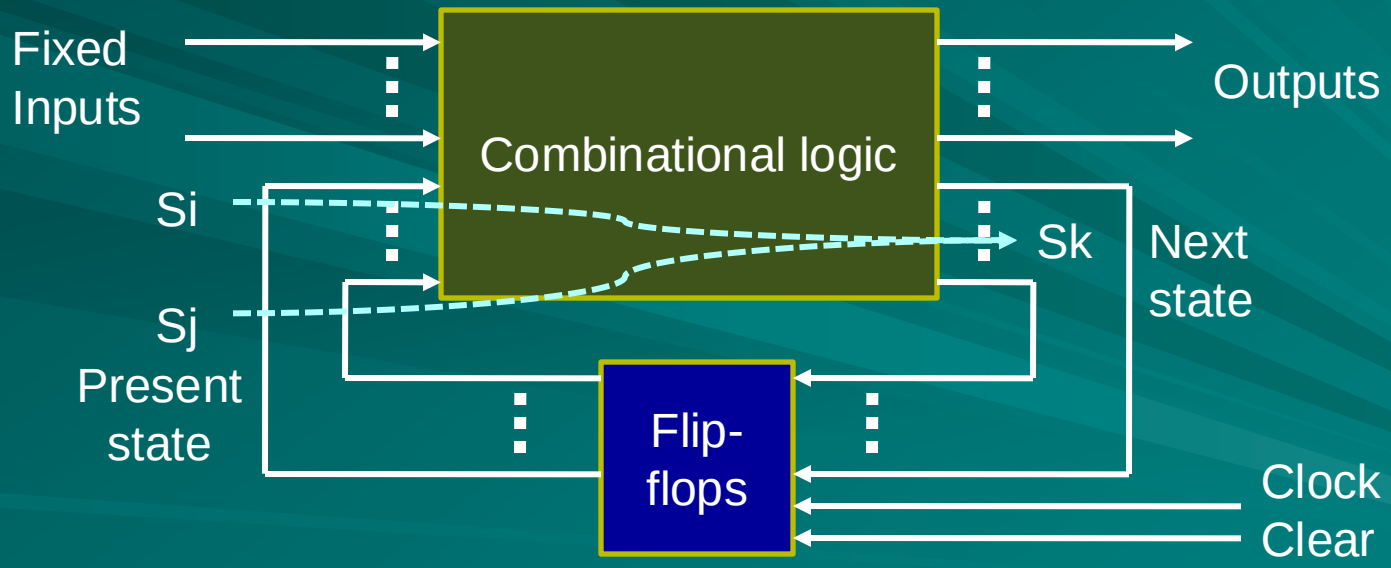
- Larger clusters produce smaller logic function.
- Clustered minterms differ in one variable.

Size of an Implementation

- Number of product terms determines number of gates.
- Number of literals in a product term determines number of gate inputs, which is proportional to number of transistors.
- Hardware α (total number of literals)
- Examples of four minterm functions:
  - F1 = $ABCD + \bar{A}\,\bar{B}\,\bar{C}\,\bar{D} + \bar{A}\,\bar{B}CD + \bar{A}B\,\bar{C}D$ has 16 literals
  - F2 = $ABC + \bar{A}\,\bar{C}D$ has 6 literals

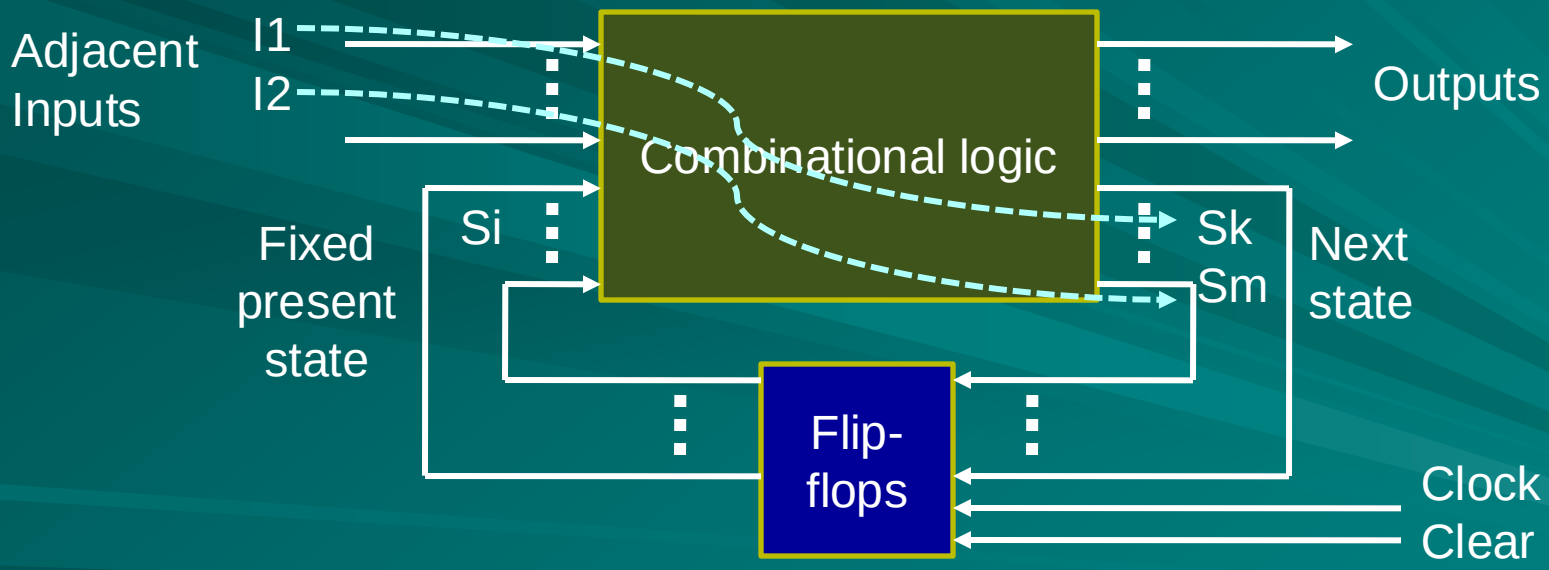
Rule 1

- States that have the same next state for some fixed input should be assigned logically adjacent codes.



Rule 2

- States that are the next states of the same state under logically adjacent inputs, should be assigned logically adjacent codes.

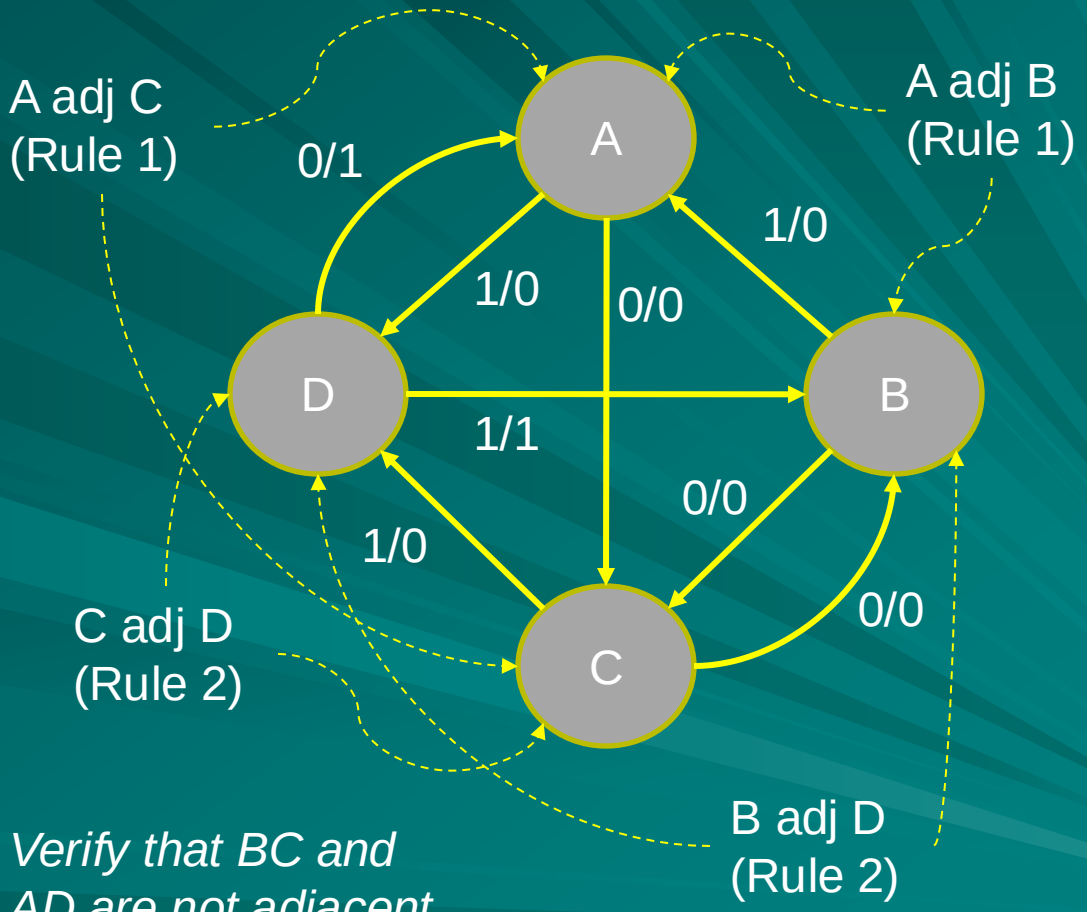


Example of State Assignment

Present state	Next state, output (Z)	
	Input, X	
	0	1
A	C, 0	D, 0
B	C, 0	A, 0
C	B, 0	D, 0
D	A, 1	B, 1

Figure 9.19 of textbook

	0	1
0	A	B
1	C	D



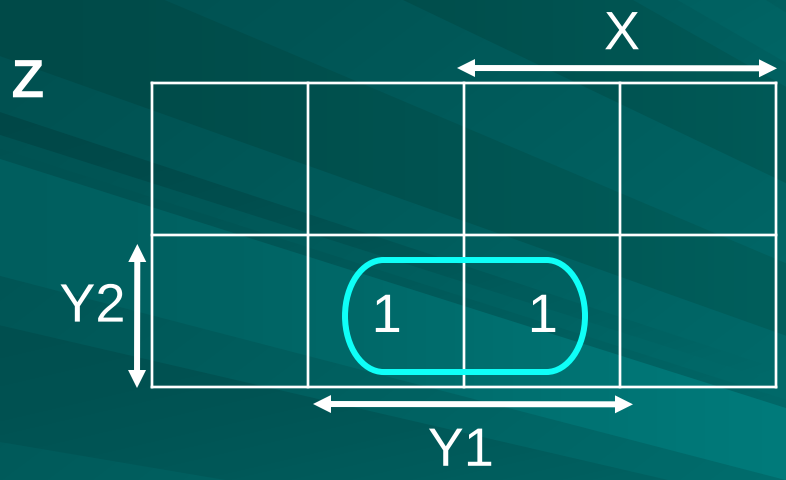
Verify that BC and AD are not adjacent.

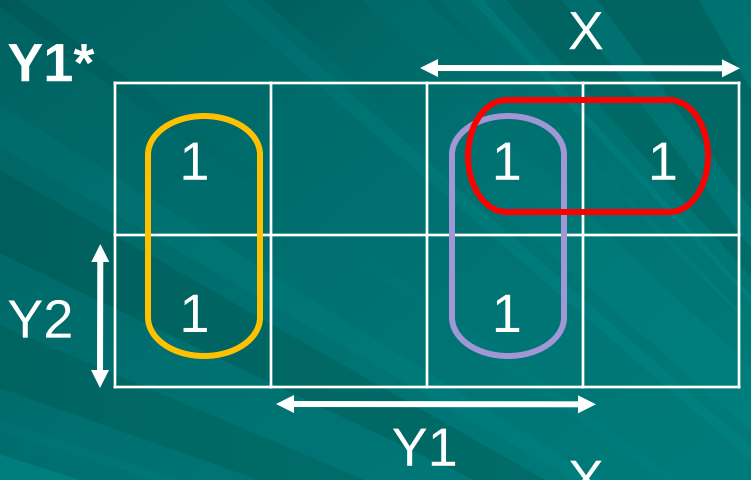
A = 00, B = 01, C = 10, D = 11

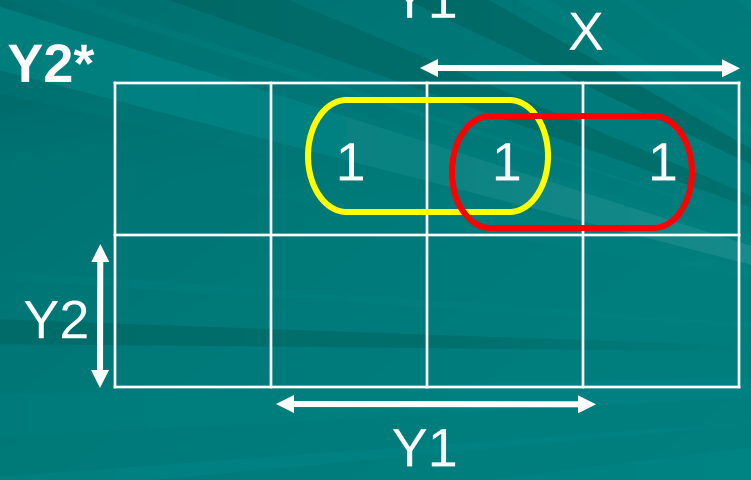
Present state Y1, Y2	Next state, output Y1*Y2*, Z	
	Input, X 0	1
A = 00	10 / 0	11 / 0
B = 01	10 / 0	00 / 0
C = 10	01 / 0	11 / 0
D = 11	00 / 1	01 / 1

Input	Present state		Output	Next state	
X	Y1	Y2	Z	Y1*	Y2*
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	0	1	1
1	1	1	1	1	0

Logic Minimization for Optimum State Assignment

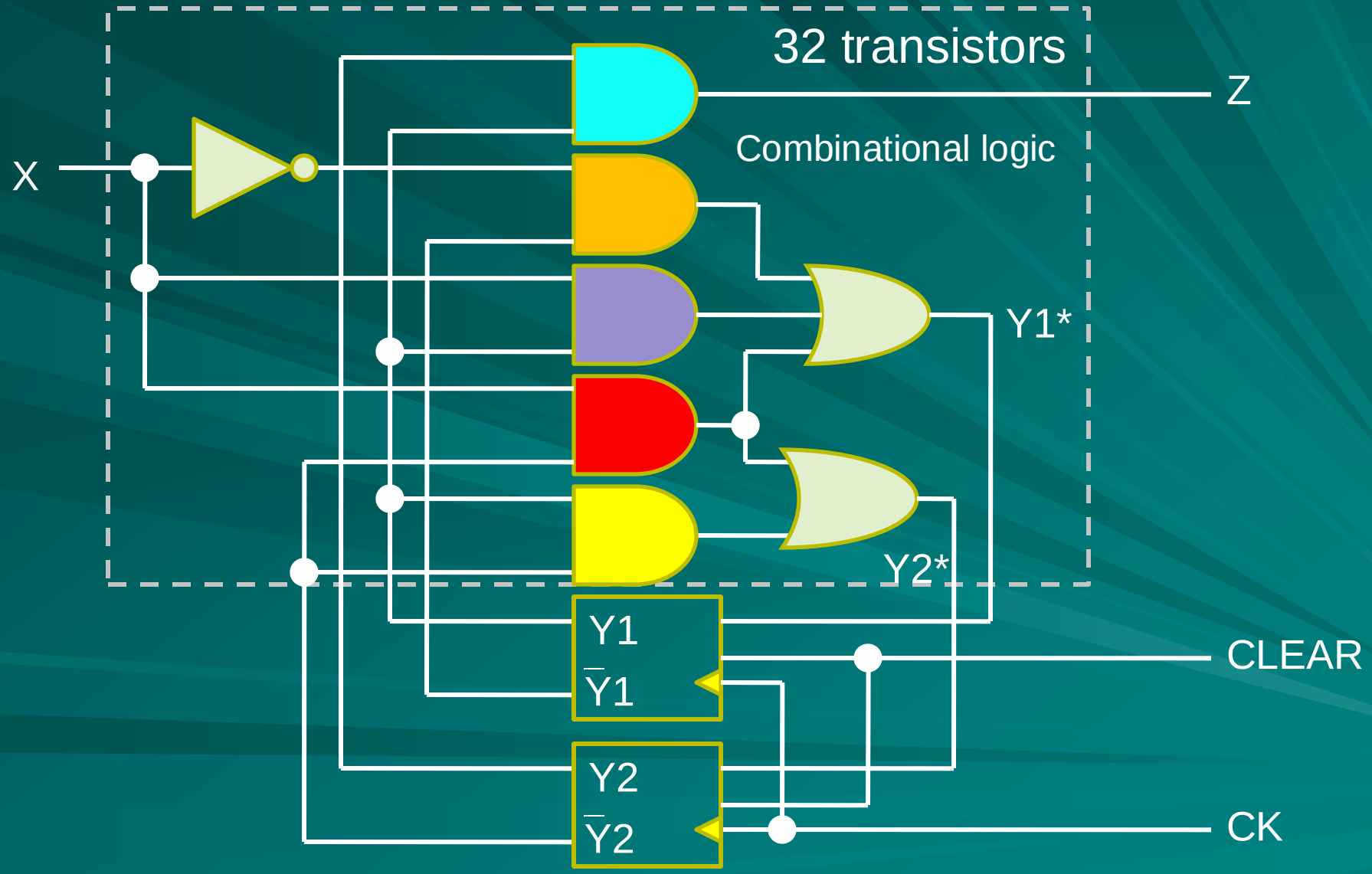






Result: 5 products, 10 literals.

Circuit for Optimum State Assignment

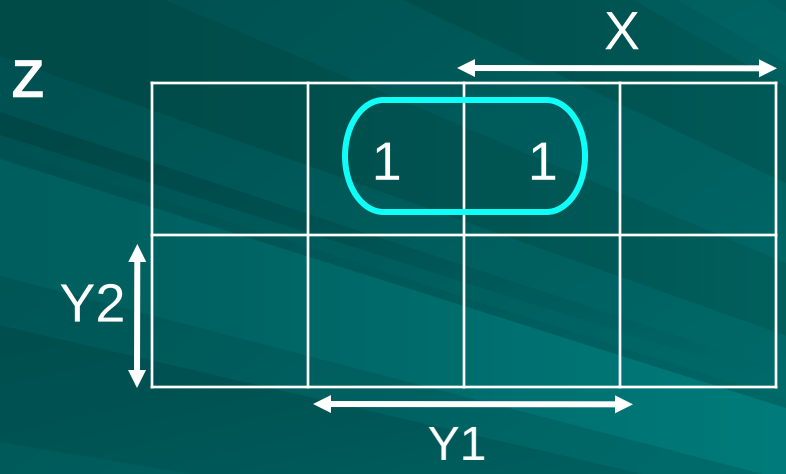


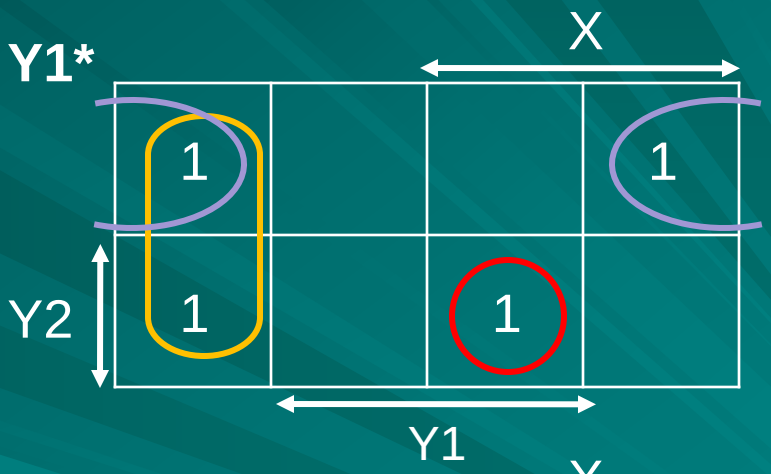
Using an Arbitrary State Assignment: A = 00, B = 01, C = 11, D = 10

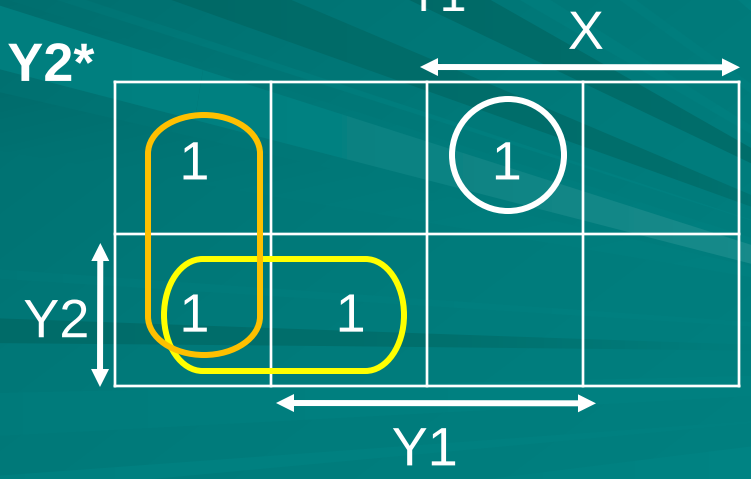
Present state Y1, Y2	Next state, output Y1*Y2*, Z	
	Input, X 0	1
A = 00	11 / 0	10 / 0
B = 01	11 / 0	00 / 0
C = 11	01 / 0	10 / 0
D = 10	00 / 1	01 / 1

Input	Present state		Output	Next state	
X	Y1	Y2	Z	Y1*	Y2*
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0

Logic Minimization for Arbitrary State Assignment

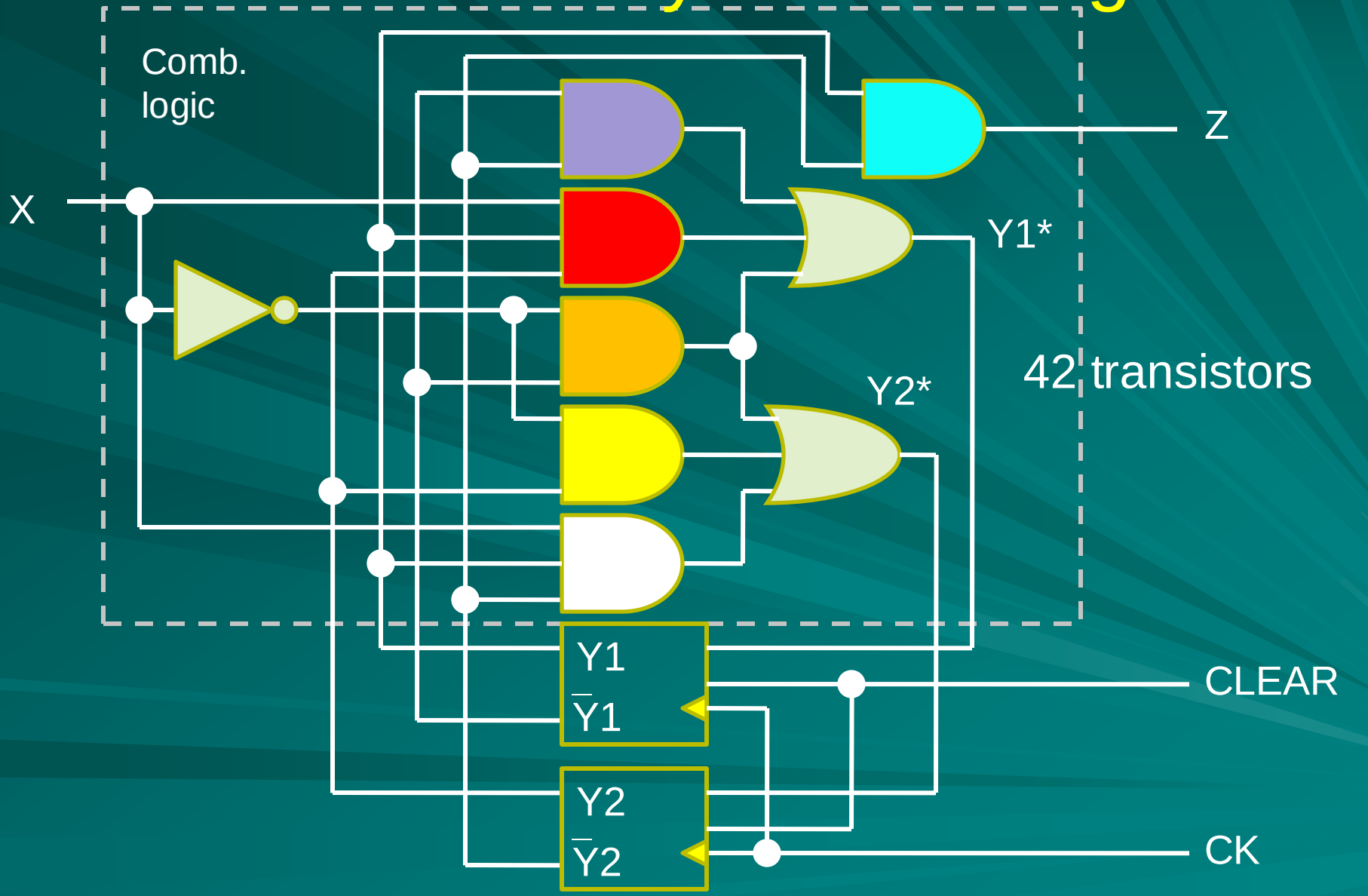






Result: 6 products, 14 literals.

Circuit for Arbitrary State Assignment



Find Out More About FSM

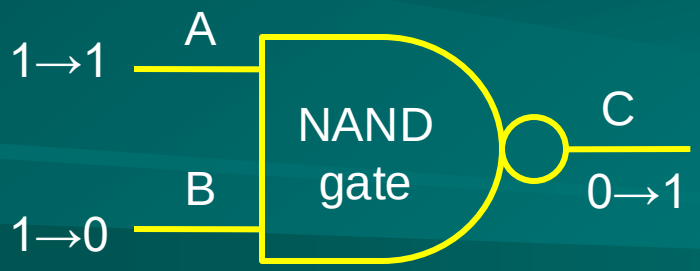
- State minimization through partioning (Section 9.2.2).
- Incompletely specified sequential circuits (Section 9.3).
- Further rules for state assignment and use of implication graphs (Section 9.4).
- Asynchronous or fundamental-mode sequential circuits (Chapter 10).

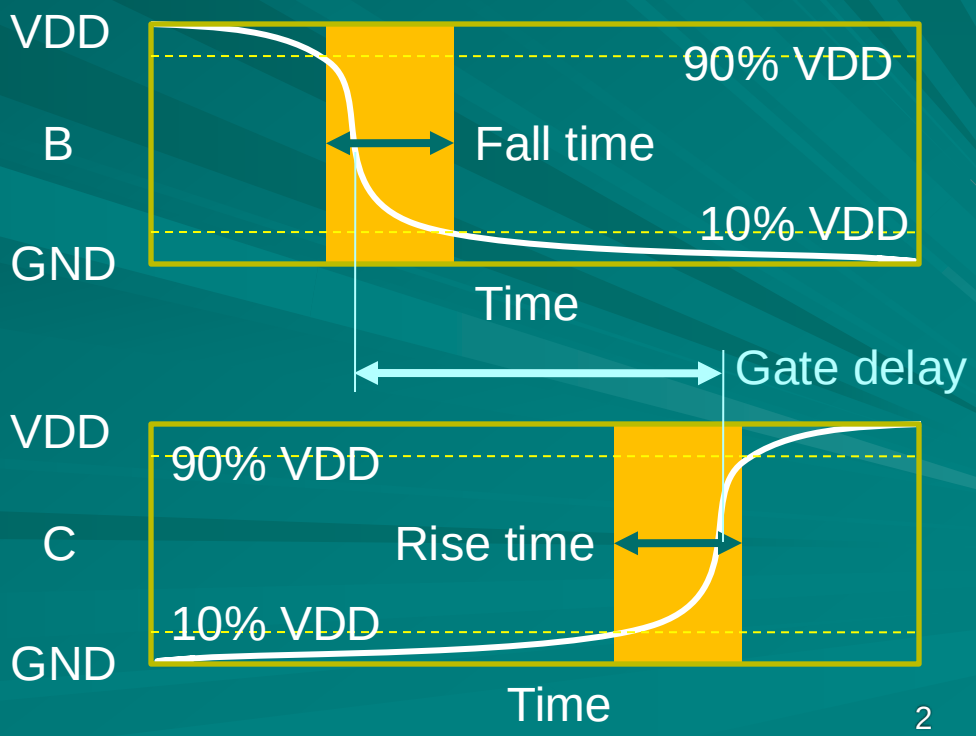
Digital Logic Circuits

Delay and Power

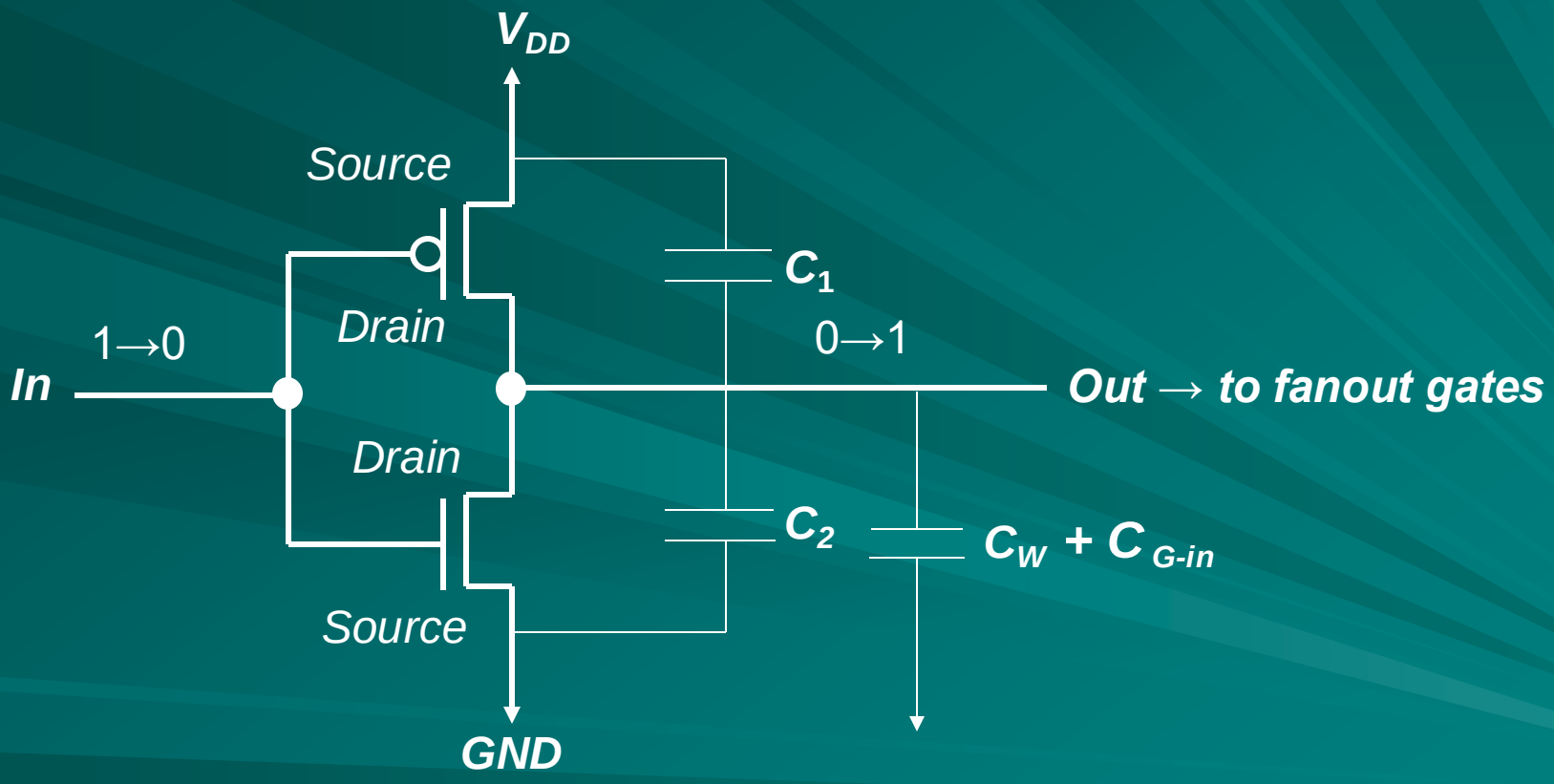
Delay: Definitions

- Rise time is the time a signal takes to rise from 10% to 90% of its peak value.
- Fall time is the time a signal takes to drop from 90% to 10% of its peak value.
- Delay of a gate is the time interval between the input crossing 50% of peak value and the output crossing 50% of peak value.



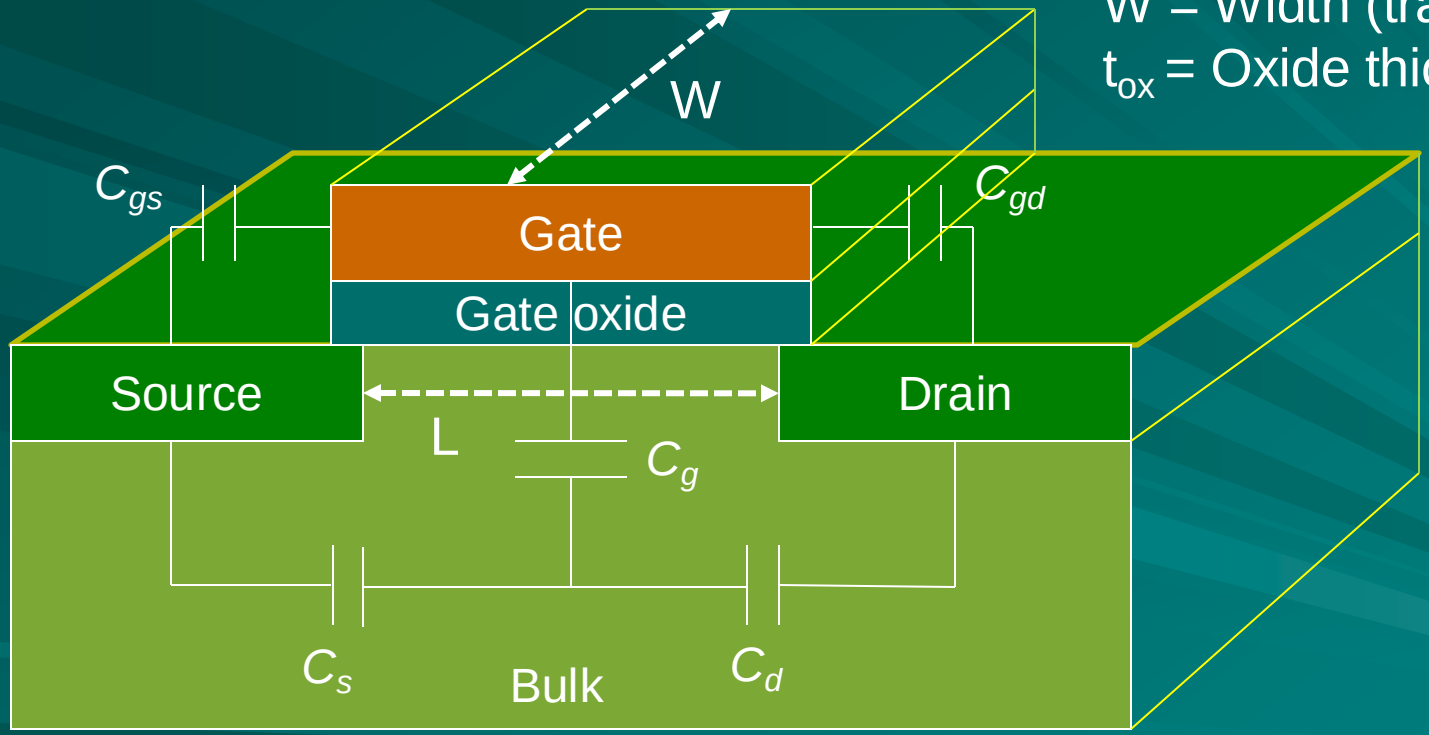


Consider Delay of Inverter (Other Gates are Similar)



Capacitances in MOSFET

L = Channel length (fixed)
W = Width (transistor size)
t_{ox} = Oxide thickness



R. C. Jaeger and T. N. Blalock, *Microelectronic Circuit Design*, Boston: McGraw-Hill, 2008.

Gate Capacitance

$$C_g = \varepsilon_{ox} WL / t_{ox}$$

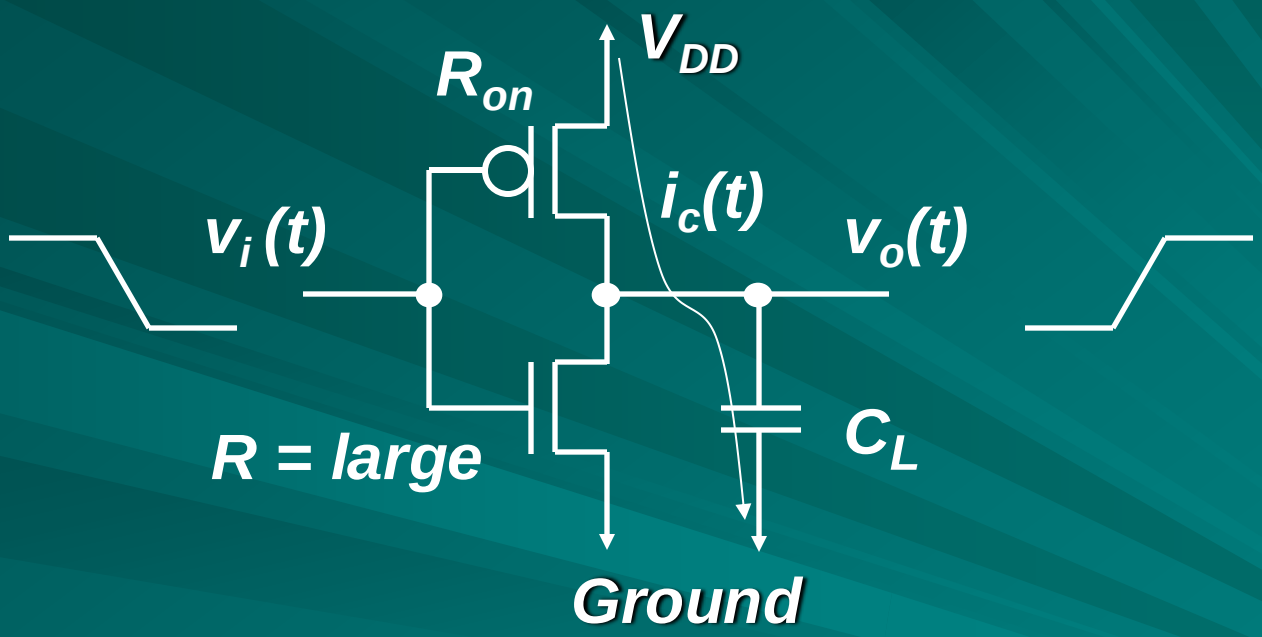
$$= C_{permicron} W$$

$$C_{permicron} = \frac{\varepsilon_{ox}}{t_{ox}} L$$

where $\varepsilon_{ox} = 3.9\varepsilon_0$ for Silicon dioxide

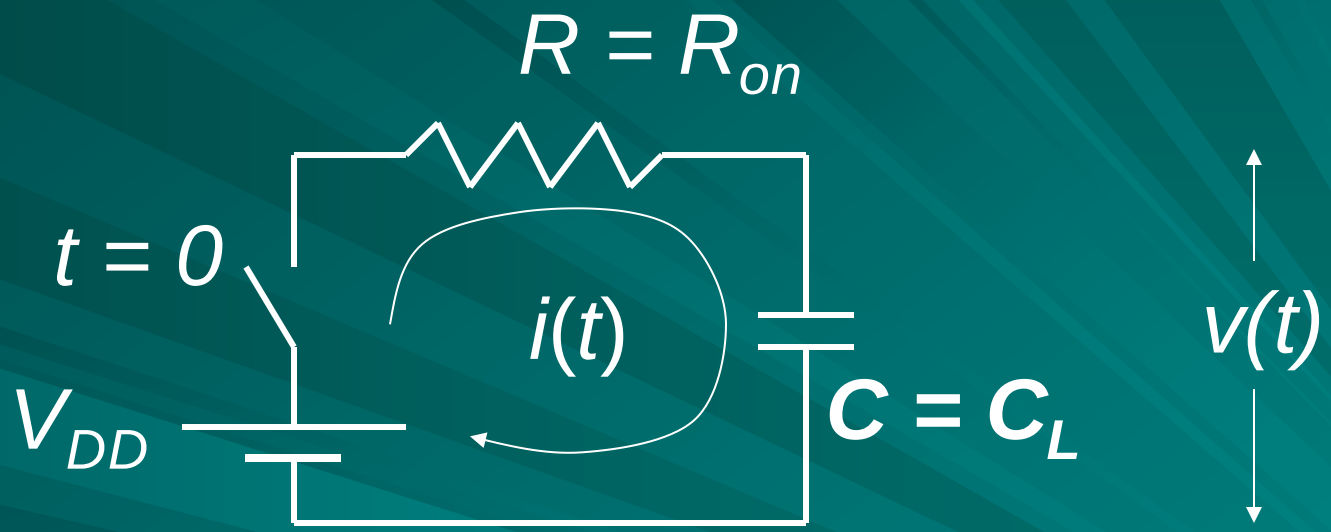
$= 3.9 \times 8.85 \times 10^{-14}$ F/cm

Propagation Delay of a Transition



C_L = Total load capacitance for gate; includes transistor capacitances of driving gate + routing capacitance + transistor capacitances of driven gates; obtained by layout analysis.

Charging of a Capacitor



Charge on capacitor, $q(t)$ = $C\,v(t)$

Current, $i(t)$ = $dq(t)/dt$ = $C\,dv(t)/dt$

$$i(t) = C\,dv(t)/dt = [V_{DD} - v(t)]\,/R$$

$$\int \frac{dv(t)}{V_{DD} - v(t)} = \int \frac{dt}{RC}$$

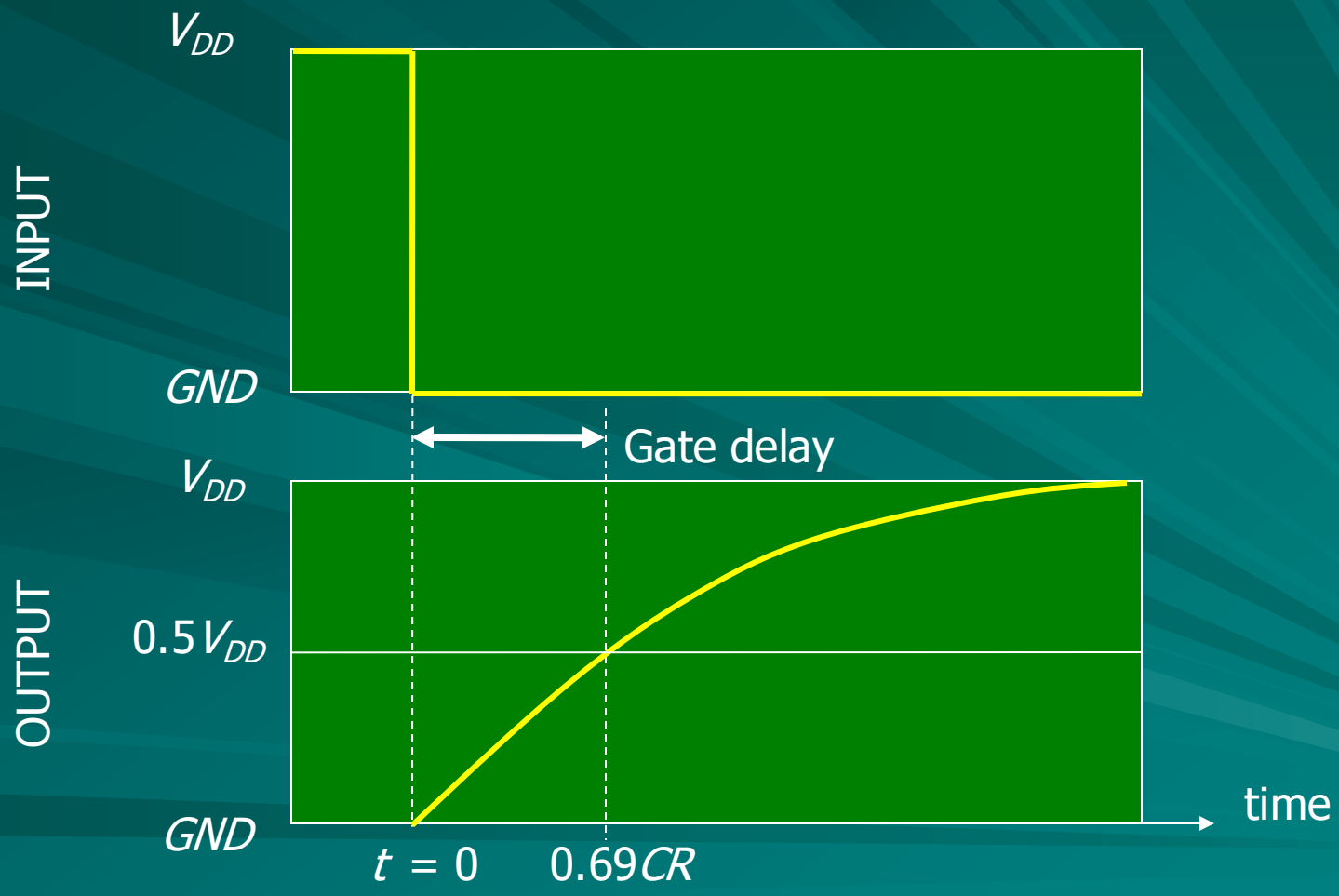
$$\ln [V_{DD} - v(t)] = \frac{-t}{RC} + A$$

Initial condition, $t = 0$, $v(t) = 0 \rightarrow A = \ln V_{DD}$

$$v(t) = V_{DD}\left[1 - \exp\left(\frac{-t}{RC}\right)\right] = 0.5V_{DD}$$

$$t = 0.69\,RC$$

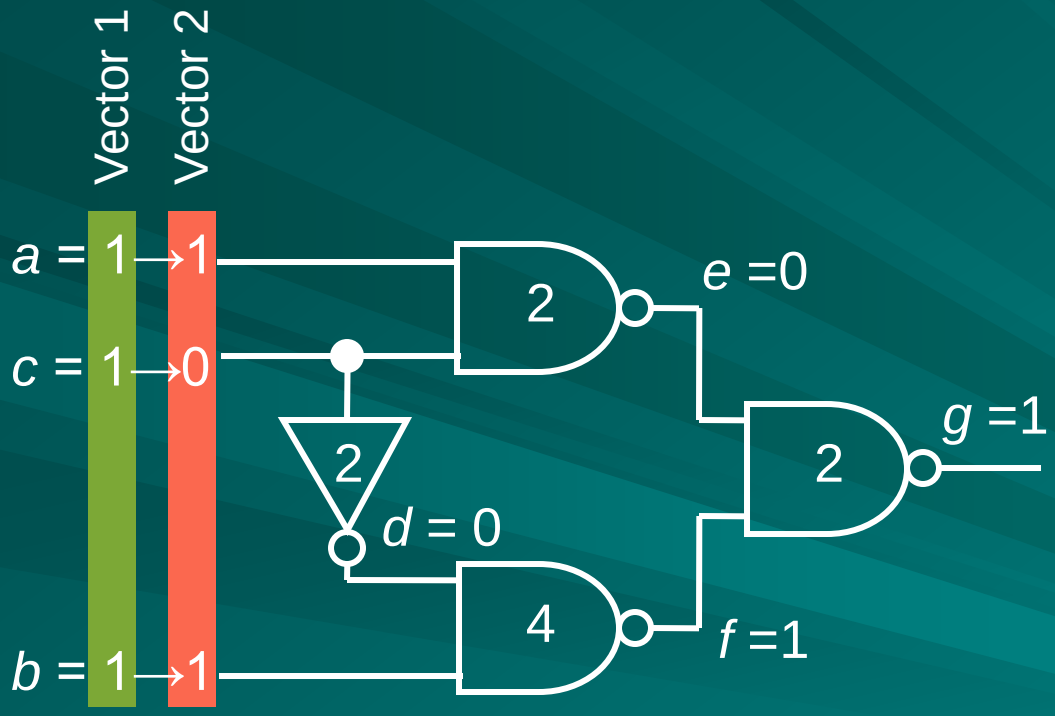
Inverter: Idealized Input



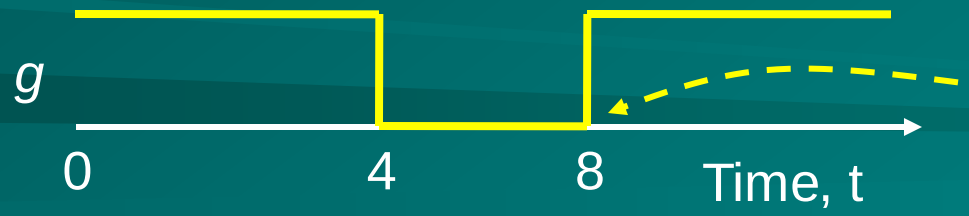
Large Circuit Timing Analysis

- Determine gate delays:
  - From layout analysis, or
  - Use approximate delays:
    - Neglect routing
    - Gate delay increases in proportion to number of fanouts
    - Delay decreases in proportion to gate size increase
- Purpose of analysis is to verify timing behavior – speed of operation.
- Methods of analysis:
  - Circuit simulation – most accurate, expensive (Spice program)
  - Event-driven logic simulation – efficient, accurate
  - Static timing analysis (STA) – most efficient, approximate

Event-Driven Simulation Example
Textbook, Section 2.7



Time stack	Scheduled events	Activity gates
t = 0	*c* = 0	*d*, *e*
1		
2	*d* = 1, *e* = 1	*f*, *g*
3		
4	*g* = 0	
5		
6	*f* = 0	*g*
7		
8	*g* = 1	



Input changes should be separated at least by 8 units of time.

Simpler Forms of Simulation

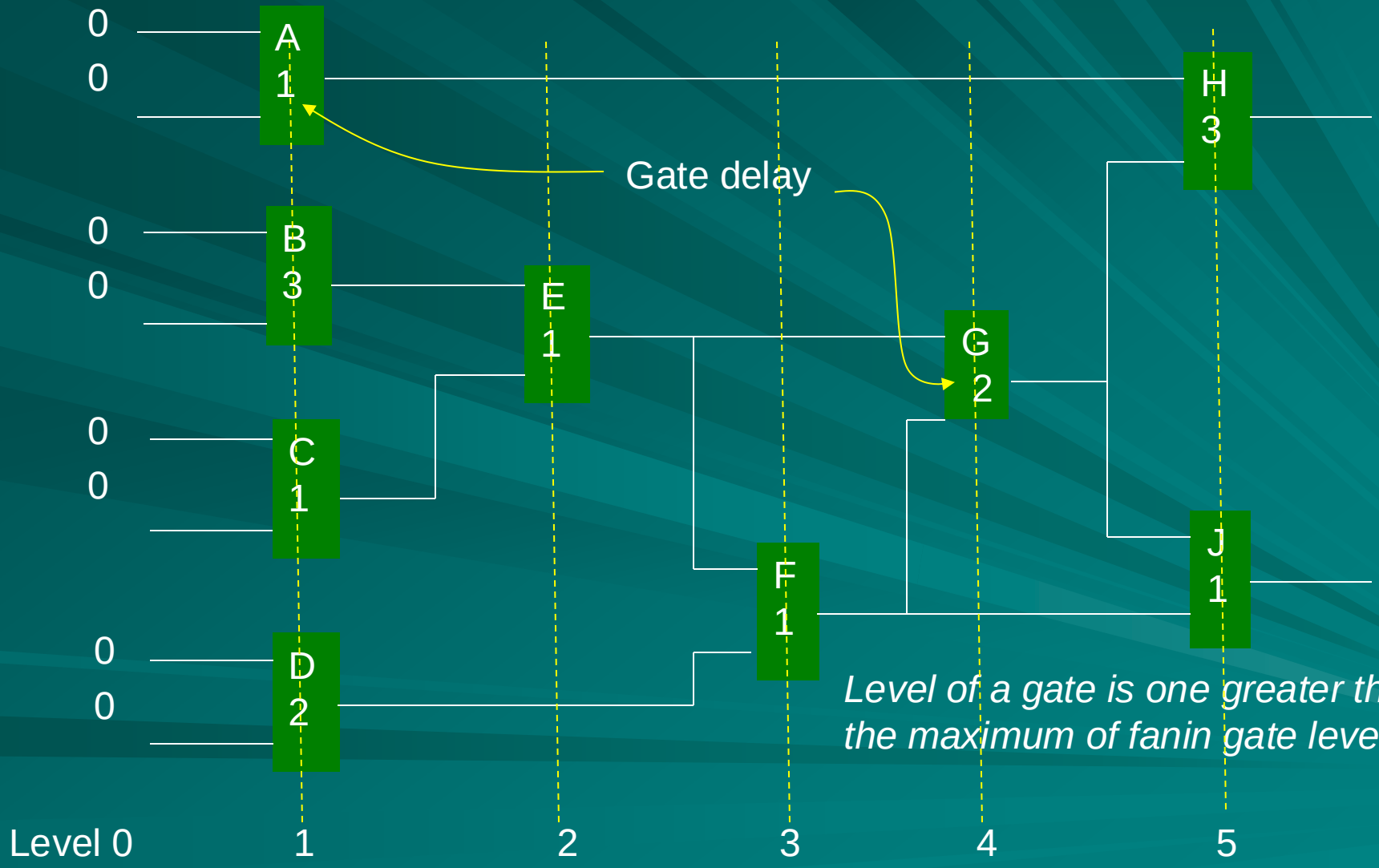
- Zero-delay simulation:
  - All gates are assumed to have zero delay
  - Suitable for combinational (feedback-free) circuits
  - Verifies logic function but not timing
- Unit-delay simulation:
  - All gates are assumed to have one unit of delay
  - Suitable for combinational as well as for feedback (sequential) circuits

Static Timing Analysis (STA)

- Combinational logic for critical path delays.
- Circuit represented as an acyclic directed graph (DAG).
- Gates characterized by delays; same as for event-driven simulation.
- No input vectors are used – worst-case anlysis – static analysis (simulation is dynamic).

Example

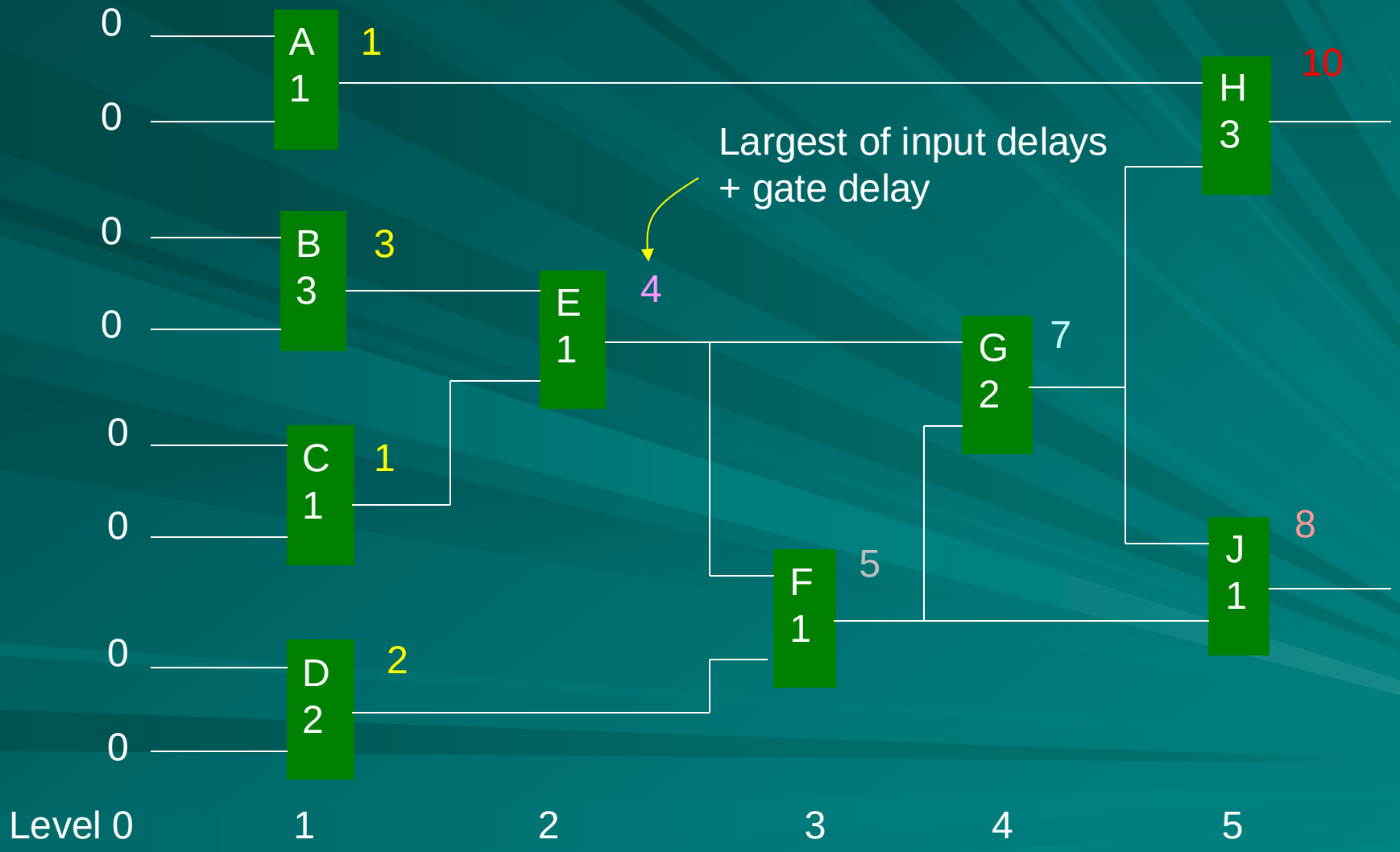
Levelize graph. Initialize arrival times at primary inputs to 0.



Level of a gate is one greater than the maximum of fanin gate levels

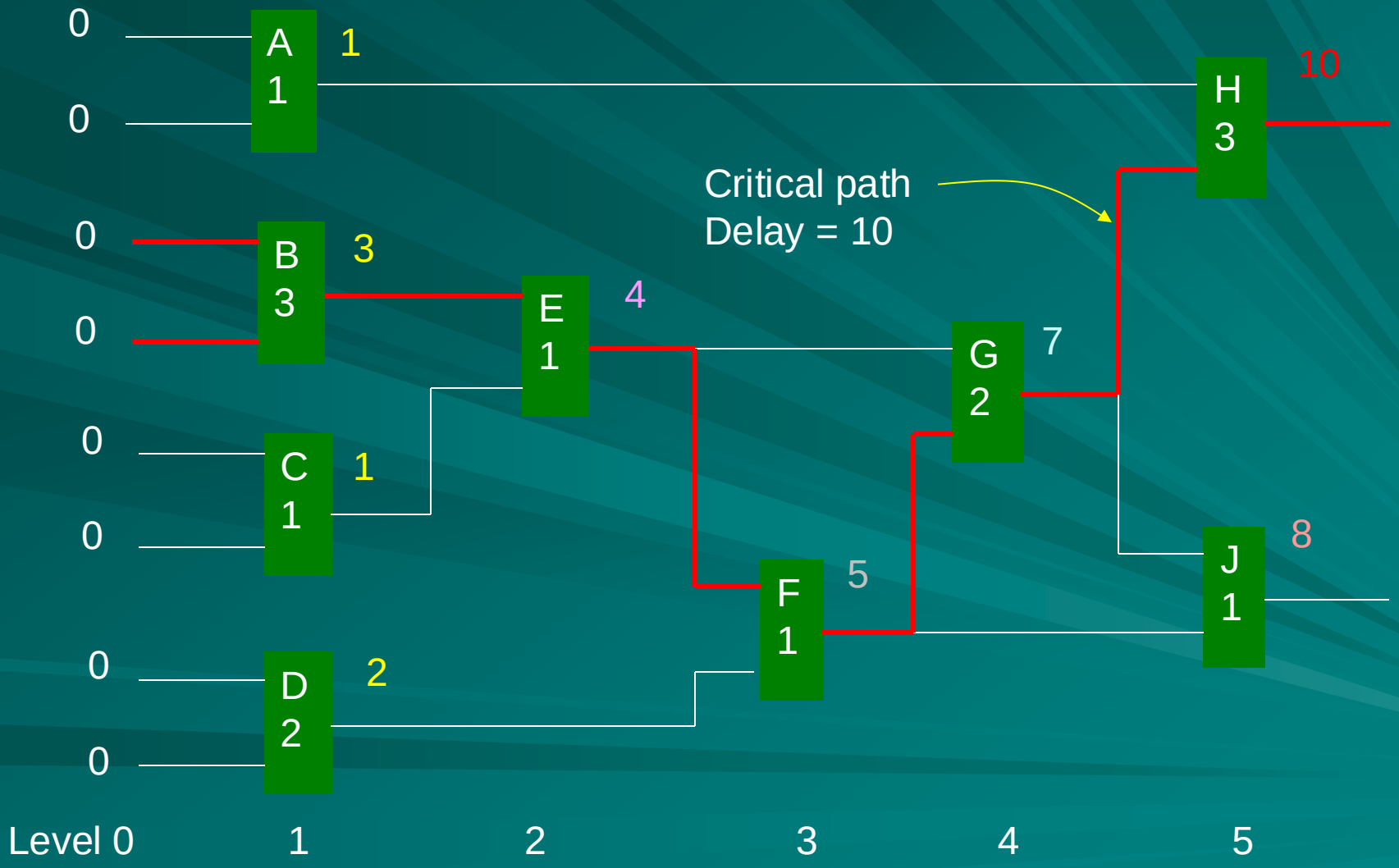
Example (Cont.)

Determine output arrival time when all input arrival times are known.



Example (Cont.)

Trace critical path from the output with longest arrival time.



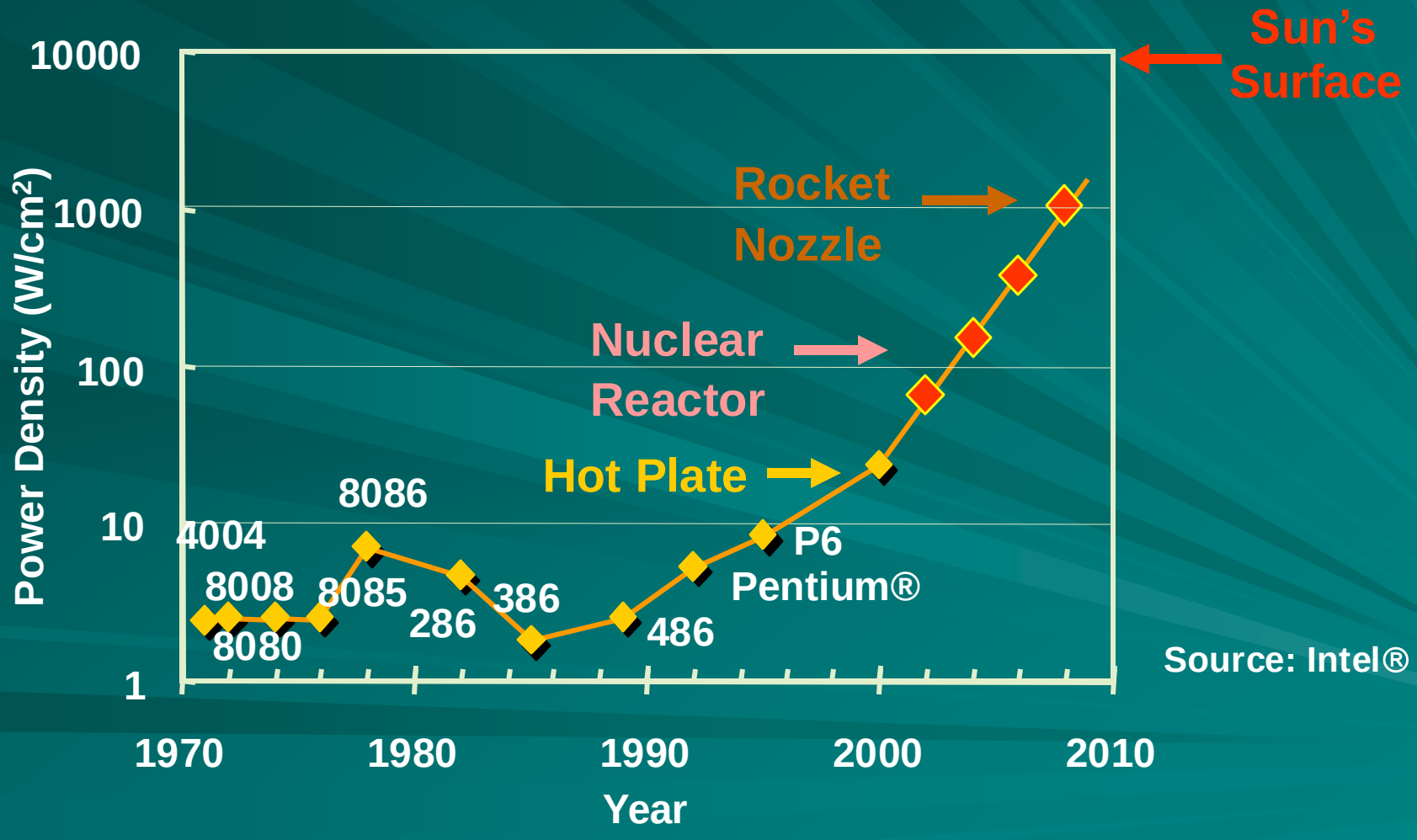
Path Analysis Algorithms for Directed Acyclic Graphs (DAG)

- Graph size: n = |V| + |E|, for |V| vertices and |E| edges.
- Levelization: O(n) (linear-time) algorithm finds the maximum (or minimum) depth.
- Path counting: $O(n^2)$ algorithm. Number of paths can be exponential in n.
- Finding all paths: Exponential-time algorithm.
- Shortest (or longest) path between two nodes:
  - Dijkstra's algorithm: $O(n^2)$
  - Bellman-Ford algorithm: $O(n^3)$

References

- Delay modeling, simulation and testing:
  - M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, Springer, 2000.
- Analysis and Design:
  - G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
  - N. Maheshwari and S. S. Sapatnekar, *Timing Analysis and Optimization of Sequential Circuits*, Springer, 1999.
- PrimeTime (Static timing analysis tool):
  - H. Bhatnagar, *Advanced ASIC Chip Synthesis, Second Edition*, Springer, 2002

VLSI Chip Power Density



ISSCC, Feb. 2001, Keynote



Patrick P. Gelsinger
Senior Vice President
General Manager
Digital Enterprise Group
INTEL CORP.

"Ten years from now, microprocessors will run at 10GHz to 30GHz and be capable of processing 1 trillion operations per second – about the same number of calculations that the world's fastest supercomputer can perform now.

"Unfortunately, if nothing changes these chips will produce as much heat, for their proportional size, as a nuclear reactor. . . ."

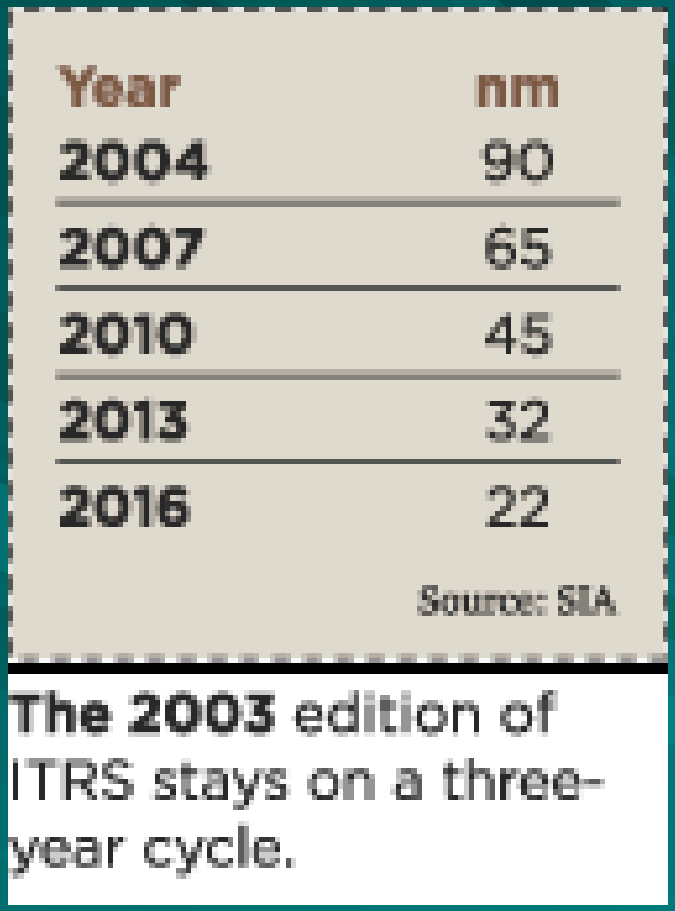
SIA Roadmap for Processors (1999)

Year	1999	2002	2005	2008	2011	2014
Feature size (nm)	180	130	100	70	50	35
Logic transistors/cm^2	6.2M	18M	39M	84M	180M	390M
Clock (GHz)	1.25	2.1	3.5	6.0	10.0	16.9
Chip size (mm^2)	340	430	520	620	750	900
Power supply (V)	1.8	1.5	1.2	0.9	0.6	0.5
High-perf. Power (W)	90	130	160	170	175	183

Untrue predictions.

Source: *http://www.semichips.org*

Silicon Technology Trend



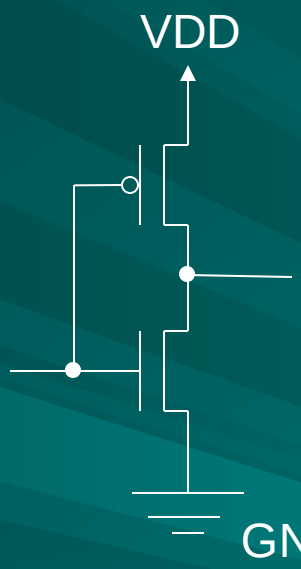
Year	nm
2004	90
2007	65
2010	45
2013	32
2016	22

Source: SIA

The 2003 edition of ITRS stays on a three-year cycle.

Source: *http://www.eetimes.com/story/OEG20040123S0041*

CMOS Logic (Inverter)



No current flows from power supply! Where is power consumed?

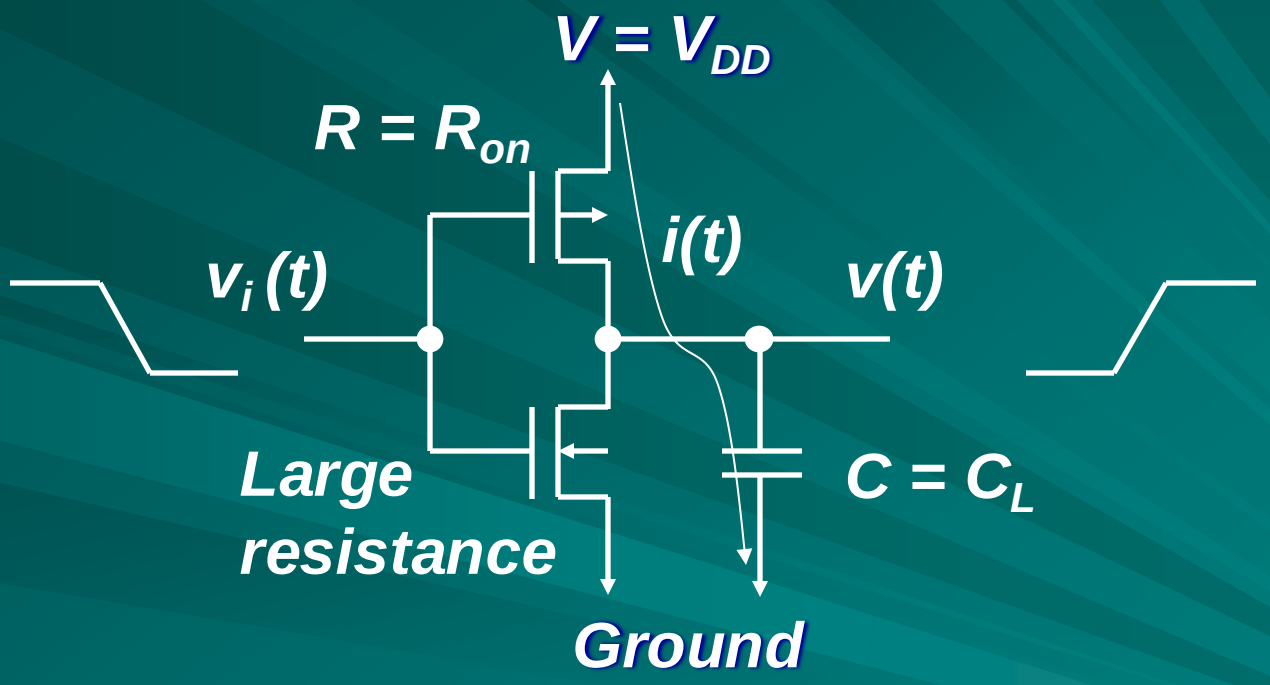
F. M. Wanlass and C. T. Sah, “Nanowatt Logic using Field-Effect Metal-Oxide-Semiconductor Triodes,” *IEEE International Solid-State Circuits Conference Digest*, vol. IV, February 1963, pp. 32-33.

Components of Power

- Dynamic, when output changes
  - Signal transitions (major component)
    - Logic activity
    - Glitches
  - Short-circuit (small)
- Static, when signal is in steady state
  - Leakage (used to be small)

$$P_{total} = P_{dyn} + P_{stat}$$
$$= P_{tran} + P_{sc} + P_{stat}$$

Power of a Transition: P_{tran}



C = Total load capacitance for gate; includes transistor capacitances of driving gate + routing capacitance + transistor capacitances of driven gates; obtained by layout analysis.

Charging of a Capacitor

- From Slide 8:

$$v(t) = V\left[1 - exp\left(\frac{-t}{RC}\right)\right]$$

$$i(t) = C\frac{dv(t)}{dt} = \frac{V}{R}exp\left(\frac{-t}{RC}\right)$$

Total Energy Per Charging Transition from Power Supply

$$E_{trans} = \int_0^\infty V\, i(t)\, dt = \int_0^\infty \frac{V^2}{R} \exp\left(\frac{-t}{RC}\right) dt$$

$$= CV^2$$

Energy Dissipated Per Transition in Resistance

$$R \int_0^\infty i^2(t)\, dt = R \frac{V^2}{R^2} \int_0^\infty exp\left(\frac{-2t}{RC}\right) dt$$

$$= \frac{1}{2} CV^2$$

Energy Stored in Charged Capacitor

$$\int_0^\infty v(t)\, i(t)\, dt = \int_0^\infty V\left[1-\exp\left(\frac{-t}{RC}\right)\right]\frac{V}{R}\exp\left(\frac{-t}{RC}\right) dt$$

$$= \frac{1}{2} CV^2$$

Transition Power

- Gate output rising transition
  - Energy dissipated in pMOS transistor = $CV^2/2$
  - Energy stored in capacitor = $CV^2/2$
- Gate output falling transition
  - Energy dissipated in nMOS transistor = $CV^2/2$
- Energy dissipated per transition = $CV^2/2$
- Power dissipation:

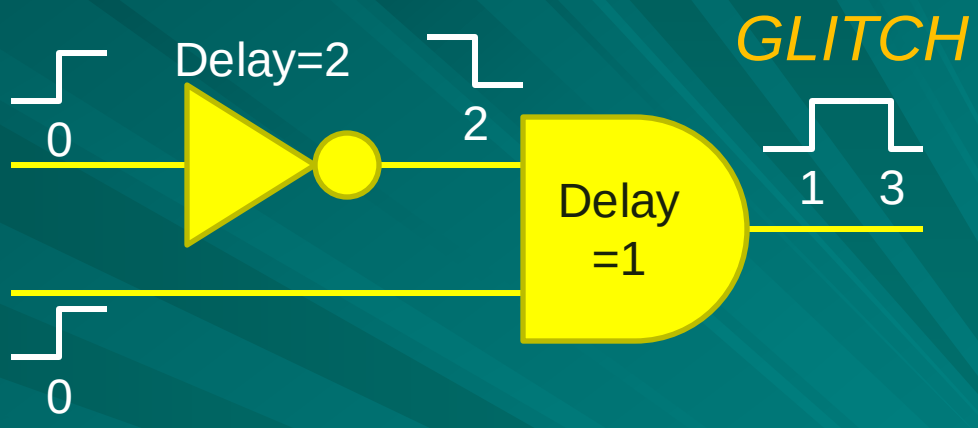
$$P_{trans} = E_{trans}\, \alpha\, f_{ck} = \alpha\, f_{ck}\, CV^2/2$$

α = *activity factor*

f_{ck} = *clock frequency*

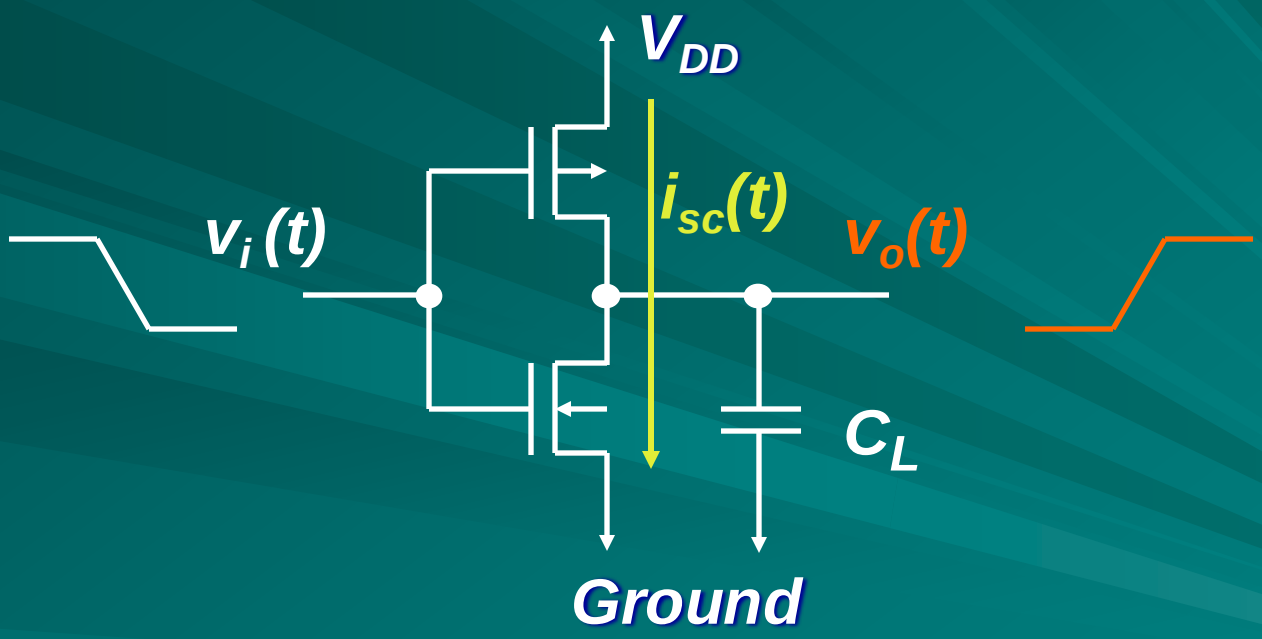
Components of Power

- Dynamic
  - Signal transitions
    - Logic activity
    - Glitches
  - Short-circuit
- Static
  - Leakage



$$P_{total} = P_{dyn} + P_{stat}$$
$$= P_{tran} + P_{sc} + P_{stat}$$

Short Circuit Power of a Transition: P_{sc}



Short-Circuit Power

- Increases with rise and fall times of input.
- Decreases for larger output load capacitance; large capacitor takes most of the current.
- Small, about 5-10% of dynamic power; momentary shorting of supply and ground during opening and closing of transistor switches.

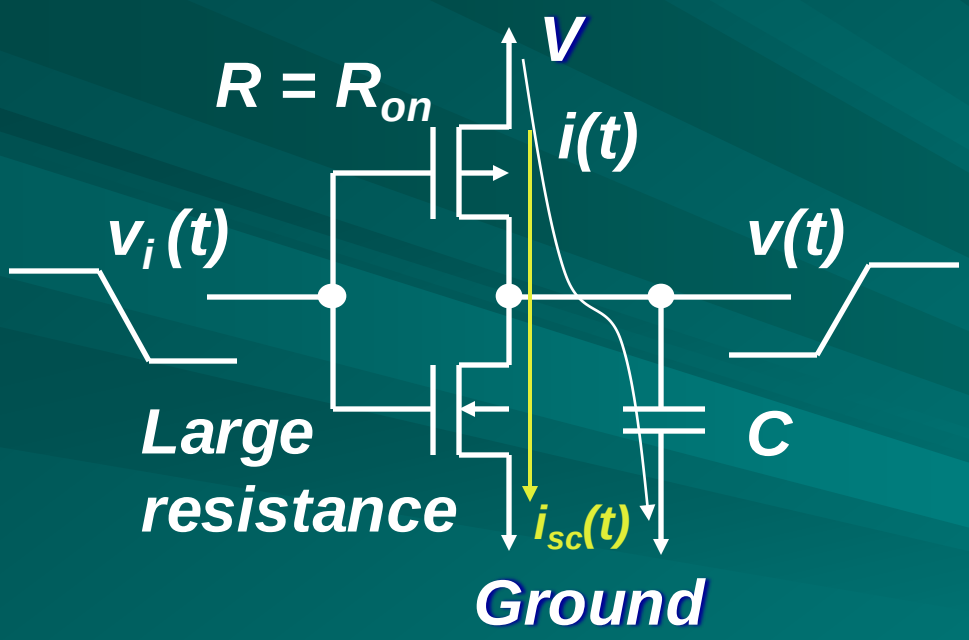
Components of Power

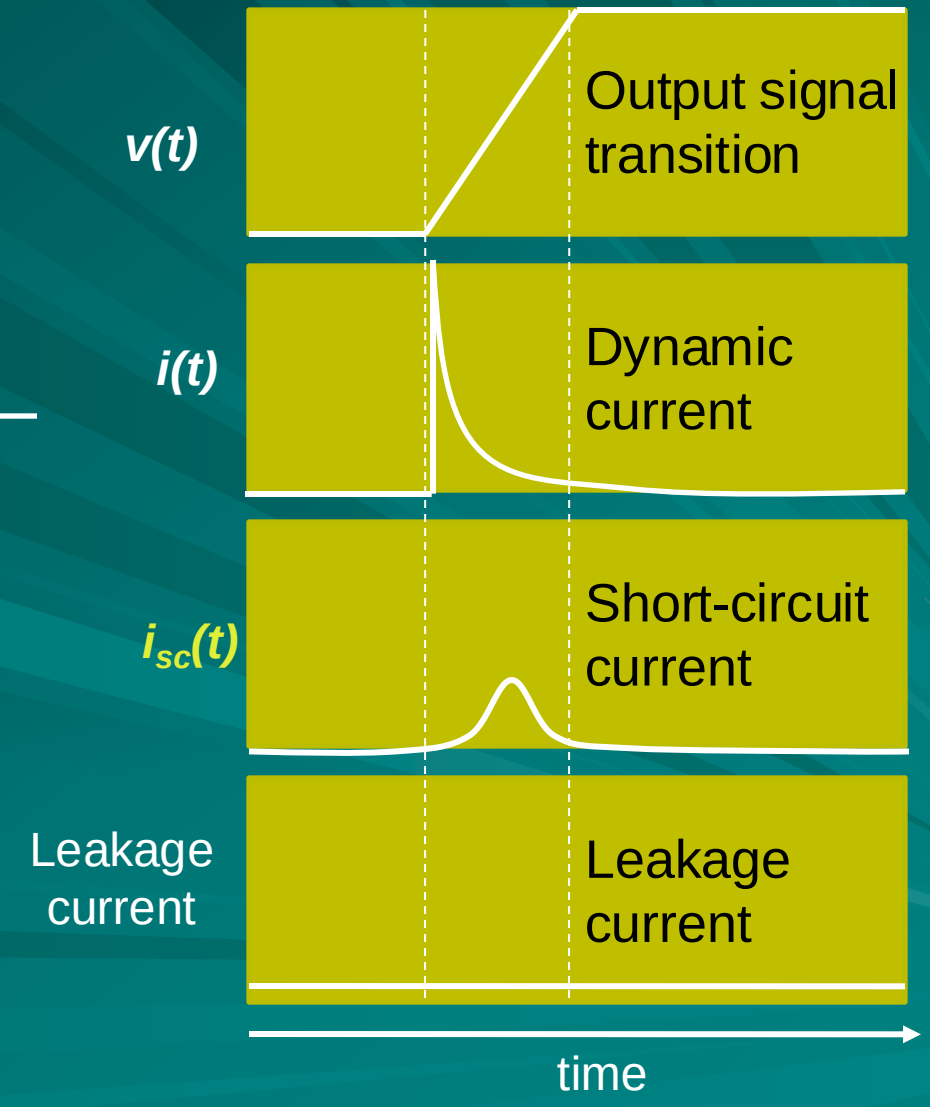
- Dynamic
  - Signal transitions
    - Logic activity
    - Glitches
  - Short-circuit
- Static
  - Leakage

Static (Leakage) Power

- Reason: Resistance of an open-circuit transistor switch is large but not infinite.
- Leakage power as a fraction of the total power increases as clock frequency drops. *Turning supply off in unused parts can save power*.
- For a gate it is a small fraction of the total power; it can be significant for very large circuits.
- Static power increases as feature size is scaled down; controlling leakage is an important aspect of transistor design and semiconductor process technology.

CMOS Gate Power

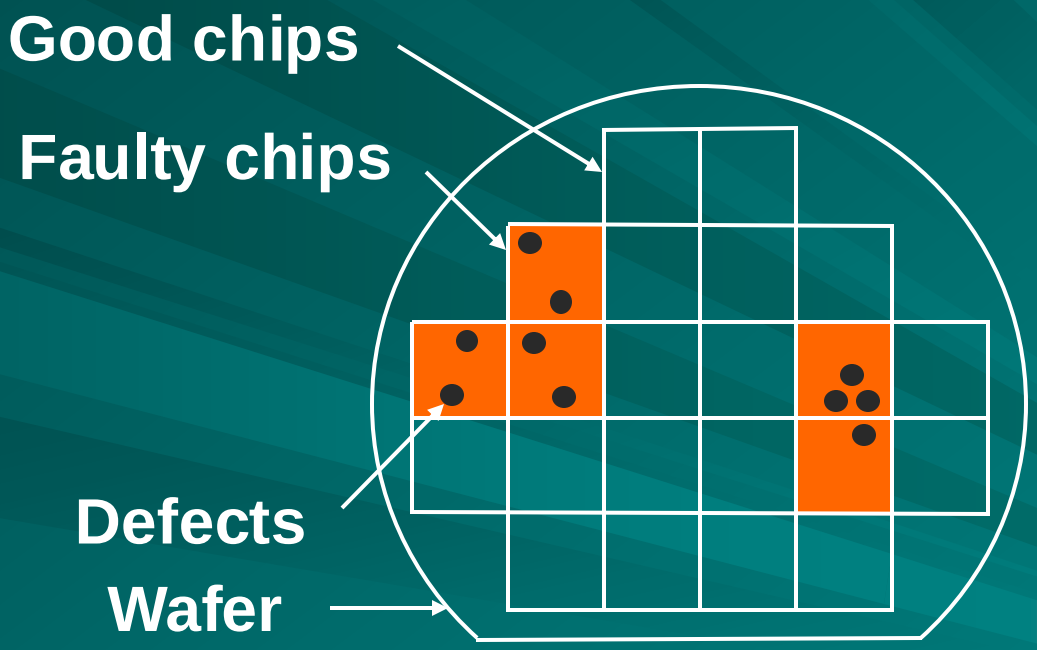




Digital Logic Circuits

Logic Testing (Chapter 12)

Circuit Fabrication and Defects



Wafer yield = 17/22 = 0.77

VLSI Chip Yield

- A chip is an area on a Silicon wafer that contains thousands, sometimes millions, of transistors and interconnects .
- A manufacturing defect is a finite chip area with electrically malfunctioning circuitry caused by errors in the fabrication process.
- A chip with no manufacturing defect is called a good chip.
- Fraction (or percentage) of good chips produced in a manufacturing process is called the *yield*. Yield is denoted by symbol *Y*.
- Cost of a chip:

$$\frac{\text{Cost of fabricating and testing a wafer}}{\text{Yield x Number of chip sites on the wafer}}$$

Yield

- Manufacturing yield is a function of three parameters:
  - Chip area, A
  - Defect density, d
  - Fault clustering parameter, α (*alpha*)
- $Y = (1 + Ad/\alpha)^{-\alpha}$
- Typical values:
  - A = 1 cm^2 for a processor chip
  - d = 1 defect/cm^2 for a matured process
  - α = 0.2 – 0.5

Yield Equation

Y = Prob (zero defect on a chip) = p (0)

$$Y = (1 + Ad / \alpha)^{- \alpha}$$

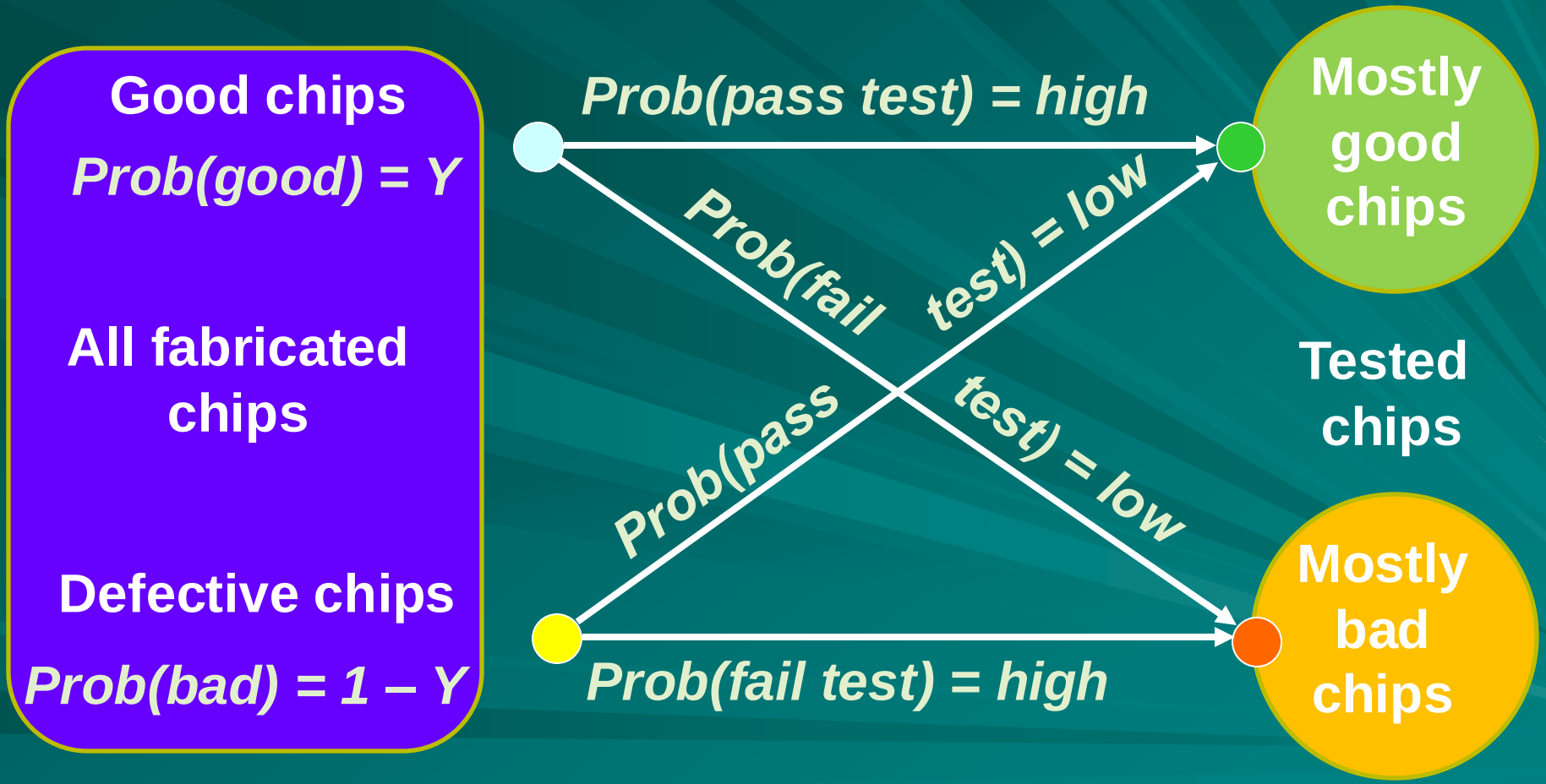
Example: Ad = 0.5, α = 0.5, Y = 0.71

Unclustered defects: $\alpha = \infty$, $Y = e^{- Ad}$

Example: Ad = 0.5, $\alpha = \infty$, Y = 0.61

too pessimistic !

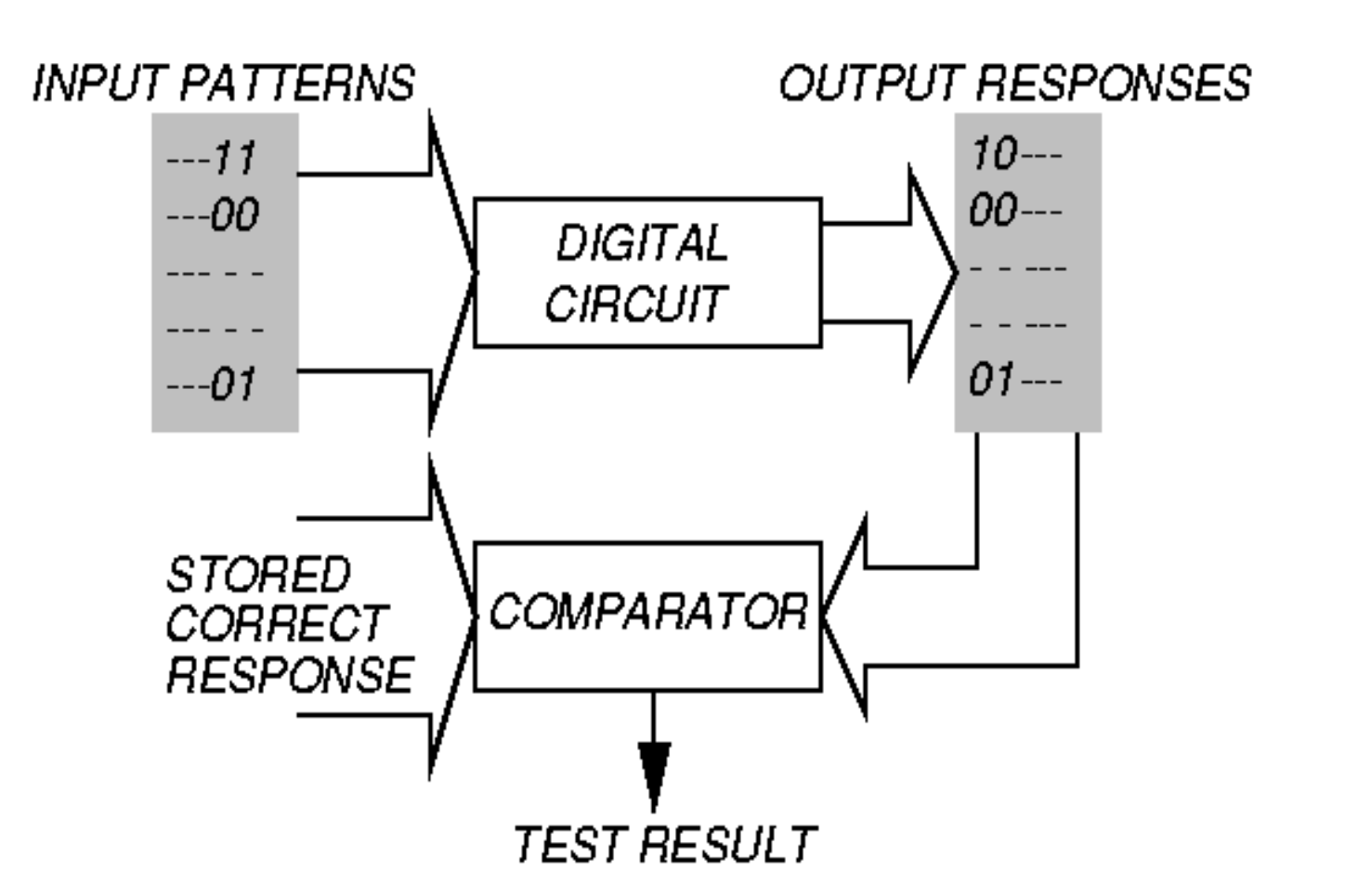
Testing Separates Good from Bad



Defect Level or Reject Ratio

- *Defect level* (DL) is the ratio of faulty chips among the chips that pass tests.
- DL is measured as *parts per million* (ppm).
- DL is a measure of the effectiveness of tests.
- DL is a quantitative measure of the manufactured product quality. For commercial VLSI chips a DL greater than 500 ppm is considered unacceptable.

How to Test?


INPUT PATTERNS
---11
---00
--- - -
--- - -
---01
DIGITAL CIRCUIT
OUTPUT RESPONSES
10---
00---
- - ---
- - ---
01---
STORED CORRECT RESPONSE
COMPARATOR
TEST RESULT

ADVANTEST Model T200GS Automatic Test Equipment (ATE), Broun 318



Why Compromise on Testing?

- Complexity and cost.
- Example: A digital circuit with 64 inputs (such as a 32-bit adder).
- There are 2^{64} = 18446744073709551616 possible input patterns or vectors.
- If we can apply one pattern in 1ns, then time to test the circuit with all patterns is
  - 2^{64} x 10^{-9} /3600 = 5124095.6 hours
  - Or 213504 days
  - Or 585 years

So, We Model Faults

- A fault model is not necessarily a real defect; model is an abstraction.
- Modeled faults should be countable; preferably, number of faults should be linear in circuit size.
- A modeled fault should be theoretically analyzable:
  - Algorithms to generate test inputs
  - Algorithms to simulate modeled faults

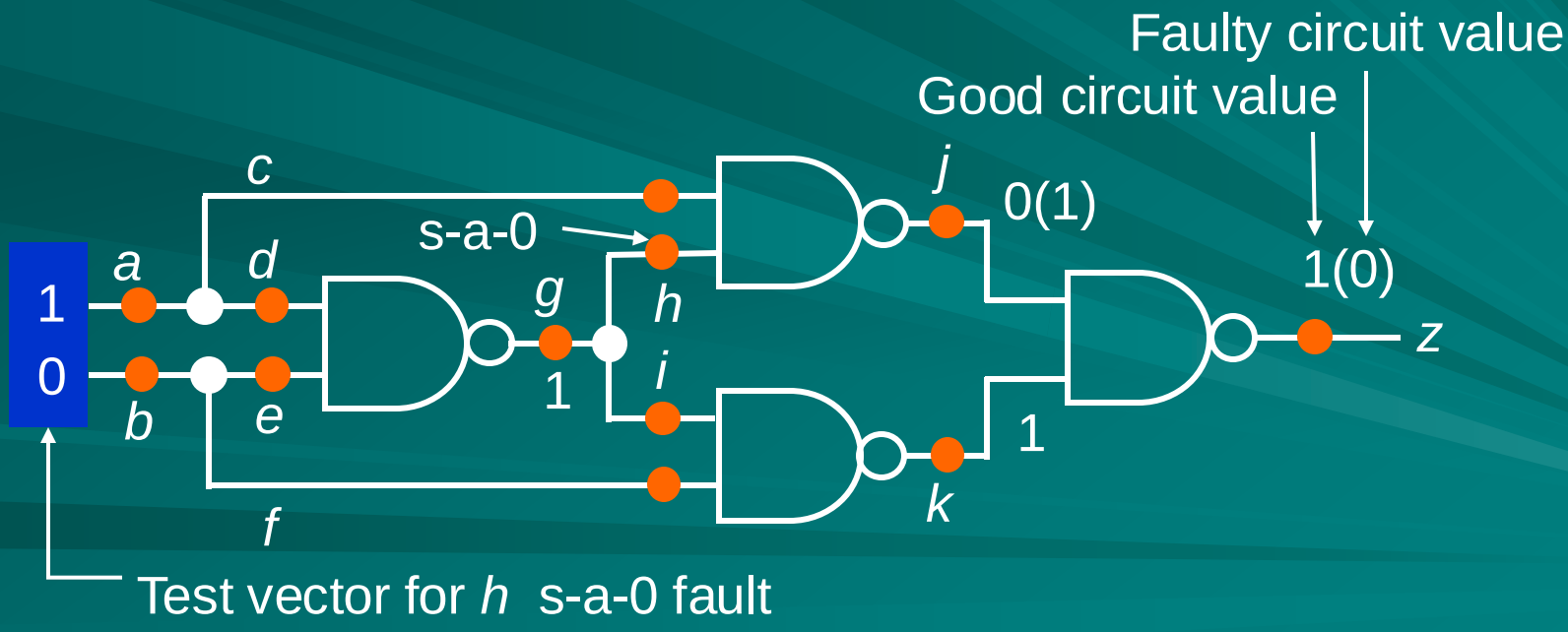
Common Fault Models

- Single stuck-at faults
- Transistor open and short faults
- Memory faults
- PLA faults (stuck-at, cross-point, bridging)
- Functional faults (processors)
- Delay faults (transition, path)
- Analog faults
- For more details of fault models, see

  M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, Springer, 2000.

Single Stuck-at Fault

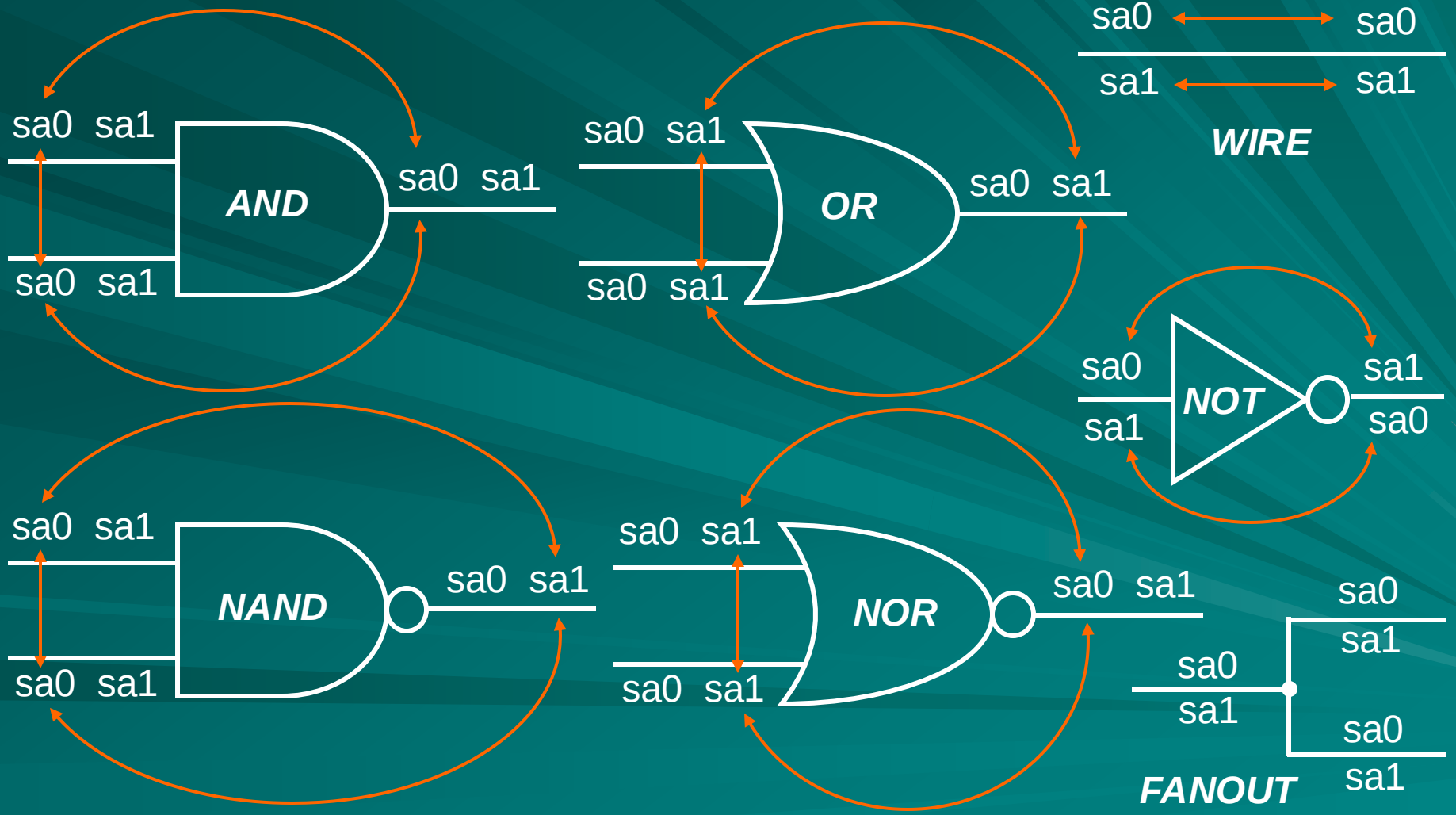
- Three properties define a single stuck-at fault
  - **Only one line is faulty**
  - **The faulty line is permanently set to 0 or 1**
  - **The fault can be at an input or output of a gate**
- Example: XOR circuit has 12 fault sites (●) and 24 single stuck-at faults



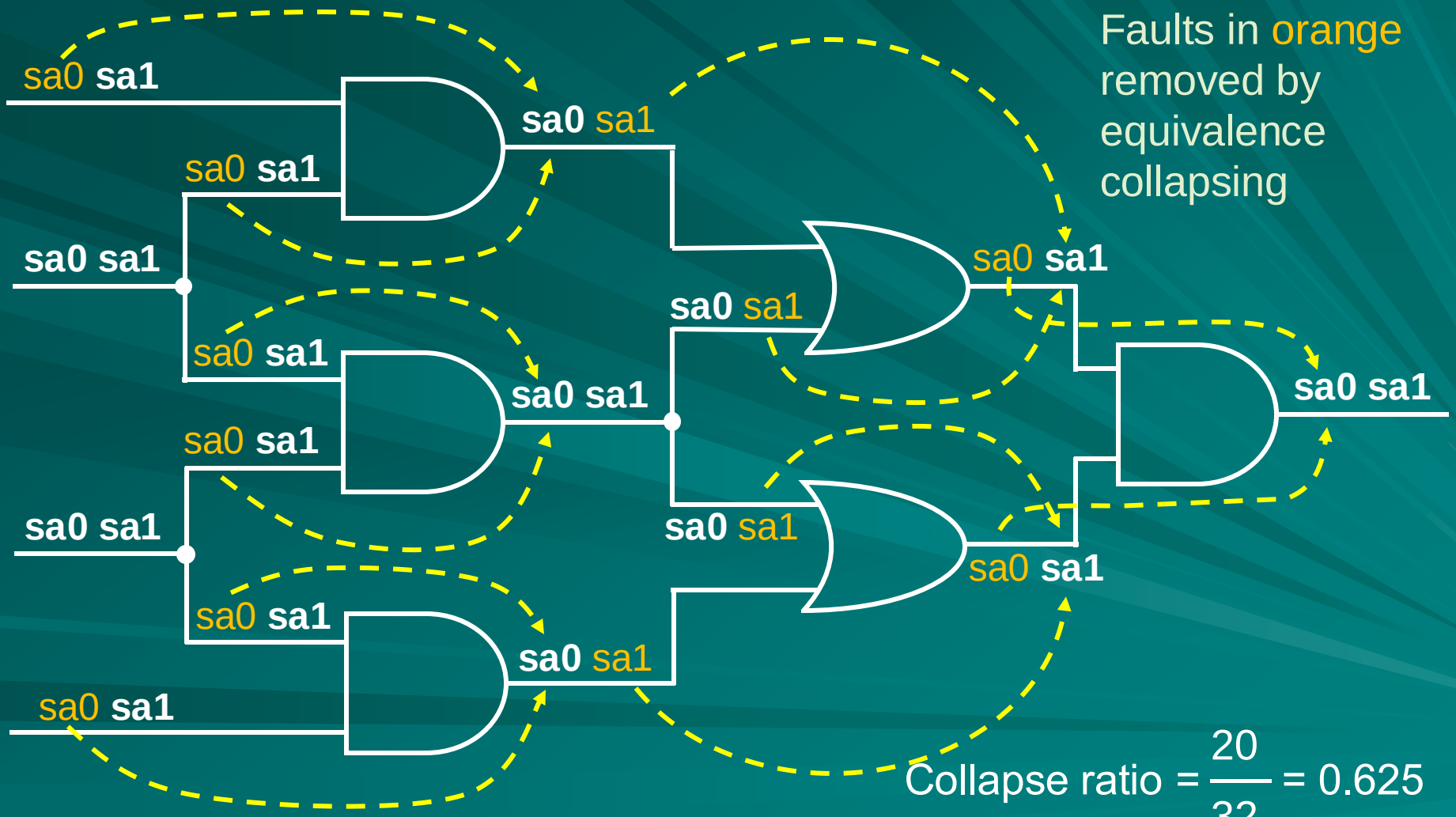
Fault Equivalence

- Number of fault sites in a Boolean gate circuit is
  = #PI + #gates + # (fanout branches)
- Fault equivalence: Two faults f1 and f2 are equivalent if the corresponding faulty functions are identical.
- If faults f1 and f2 are equivalent then any test that detects f1 also detects f2, and vice-versa.
- Fault equivalence checking rule: Faults f1 and f2 are equivalent iff the corresponding faulty functions are identical.
- Fault collapsing: All single faults of a logic circuit can be divided into disjoint equivalence subsets, where all faults in a subset are mutually equivalent. A collapsed fault set contains one fault from each equivalence subset.

Fault Equivalences Around Gates



Equivalence Collapsing Example



Faults in orange removed by equivalence collapsing

Collapse ratio = $\frac{20}{32}$ = 0.625

Exercise 1

- The cost of a chip is $1.00 when its yield is 50%. What will be its cost if you increased the yield to 80%.
- What is the total number of single stuck-at faults, counting both stuck-at-0 and stuck-at-1, in the following circuit?
- Which faults are left after equivalence fault collapsing? What is the collapse ratio?



Answers to the Exercise

- The cost of a chip is US$1.00 when its yield is 50%. What will be its cost if you increased the yield to 80%.

Assume a wafer has *n* chips, then

$$\text{Chip cost} = \frac{\text{wafer cost}}{0.5 \times n} = \$1.00$$

$$\text{Wafer cost} = 0.5n \times \$1.00 = 50n \text{ cents}$$

For yield = 0.8, chip cost = wafer cost/(0.8*n*) = 50*n*/(0.8*n*) = 62.5 cents

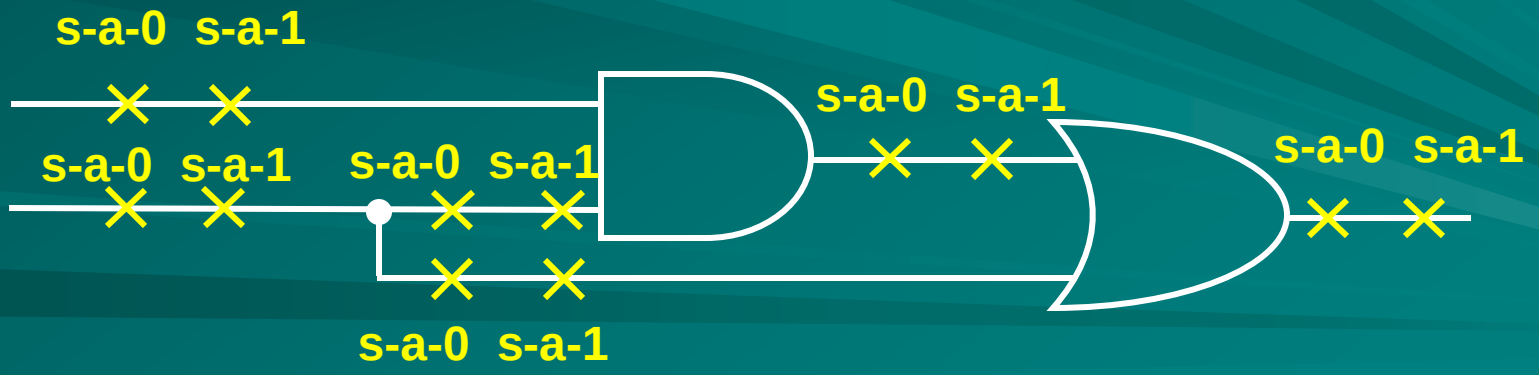
Answers Continued

- What is the total number of single stuck-at faults, counting both stuck-at-0 and stuck-at-1, in the following circuit?

Counting two faults on each line,

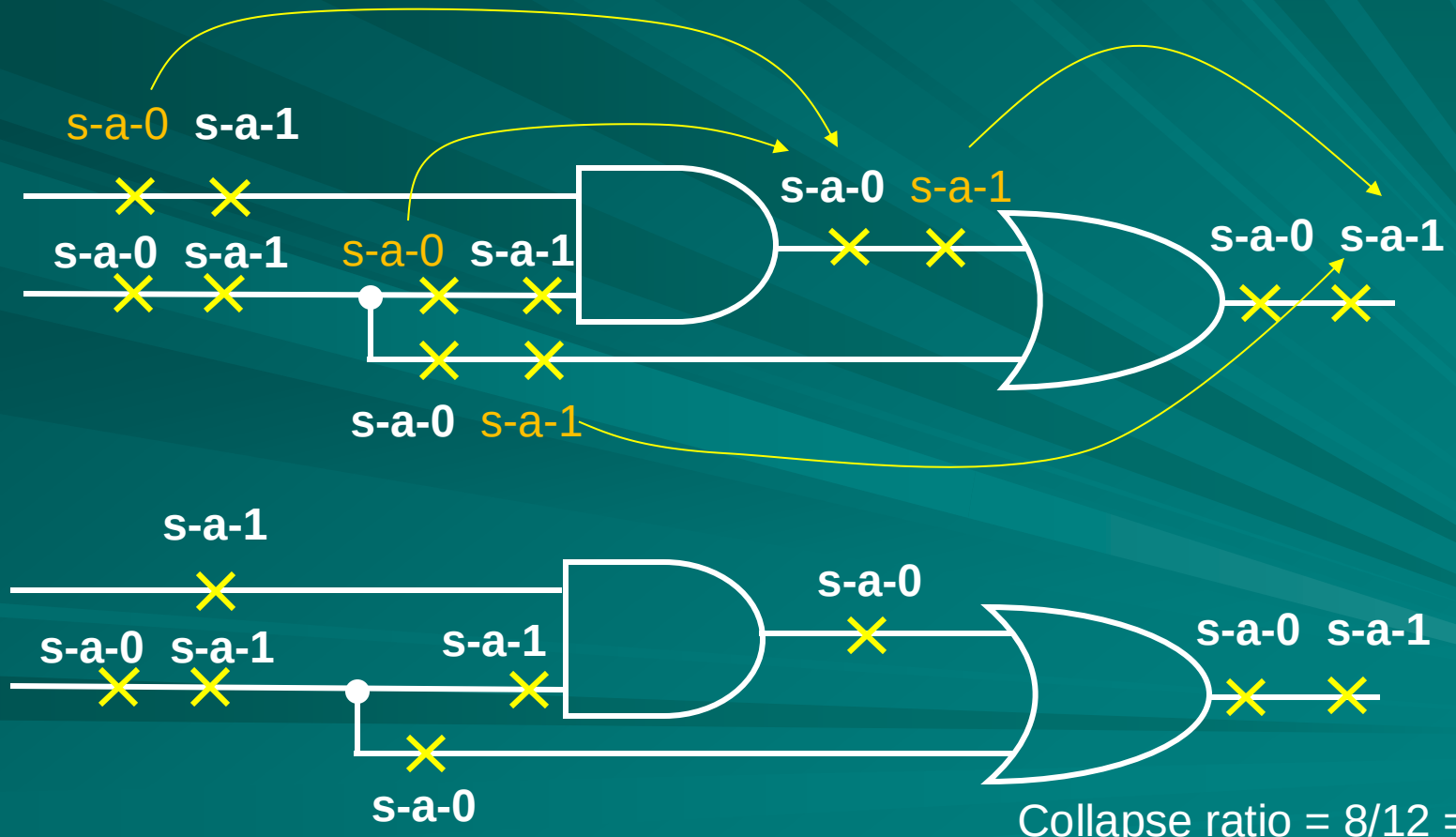
Total number of faults = 2 × (#PI + #gates + #fanout branches)

= 2 × (2 + 2 + 2) = 12



Answers Continued

- How many faults are left after equivalence fault collapsing?

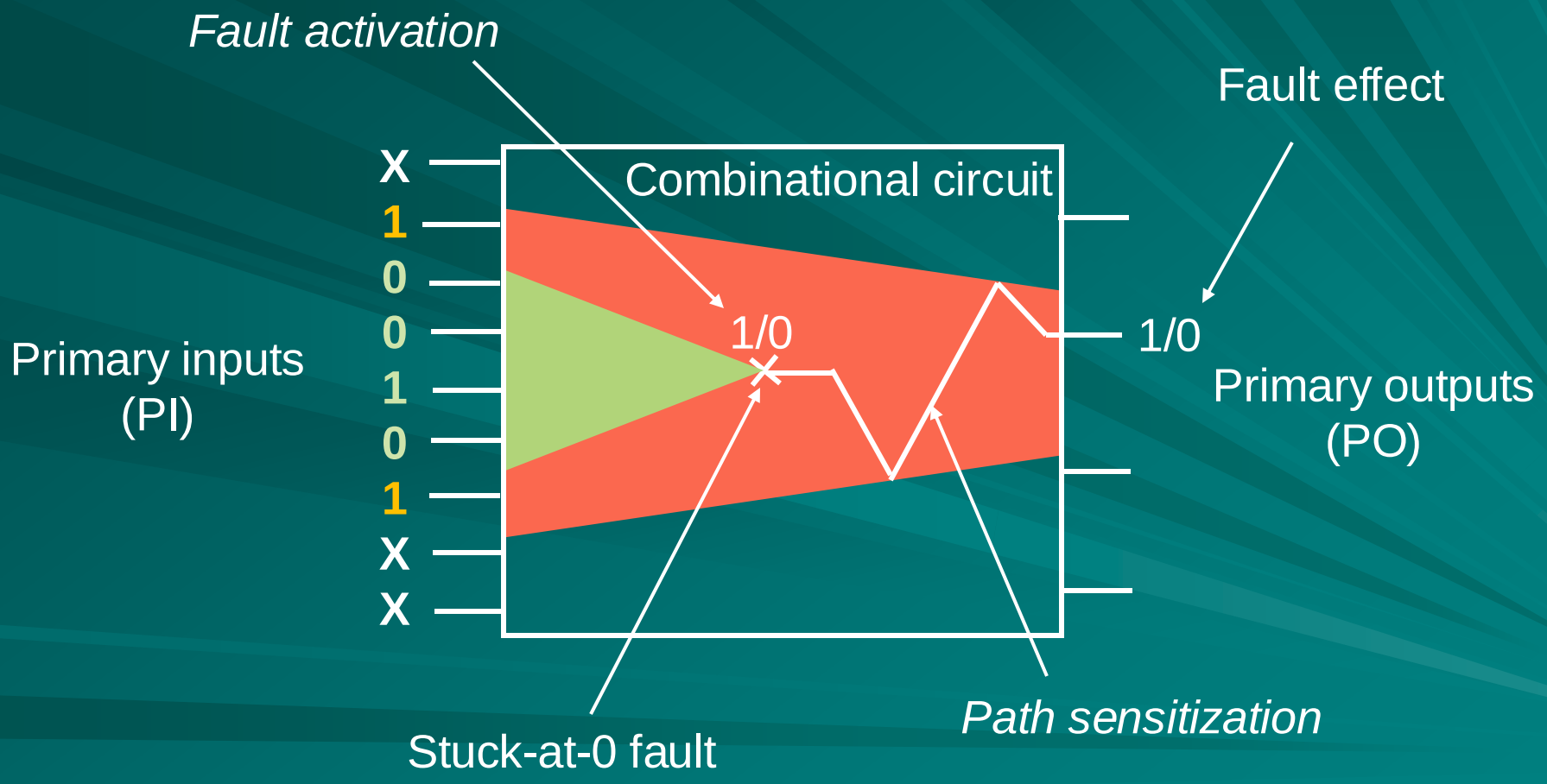


Collapse ratio = 8/12 = 0.67

ATPG Problem

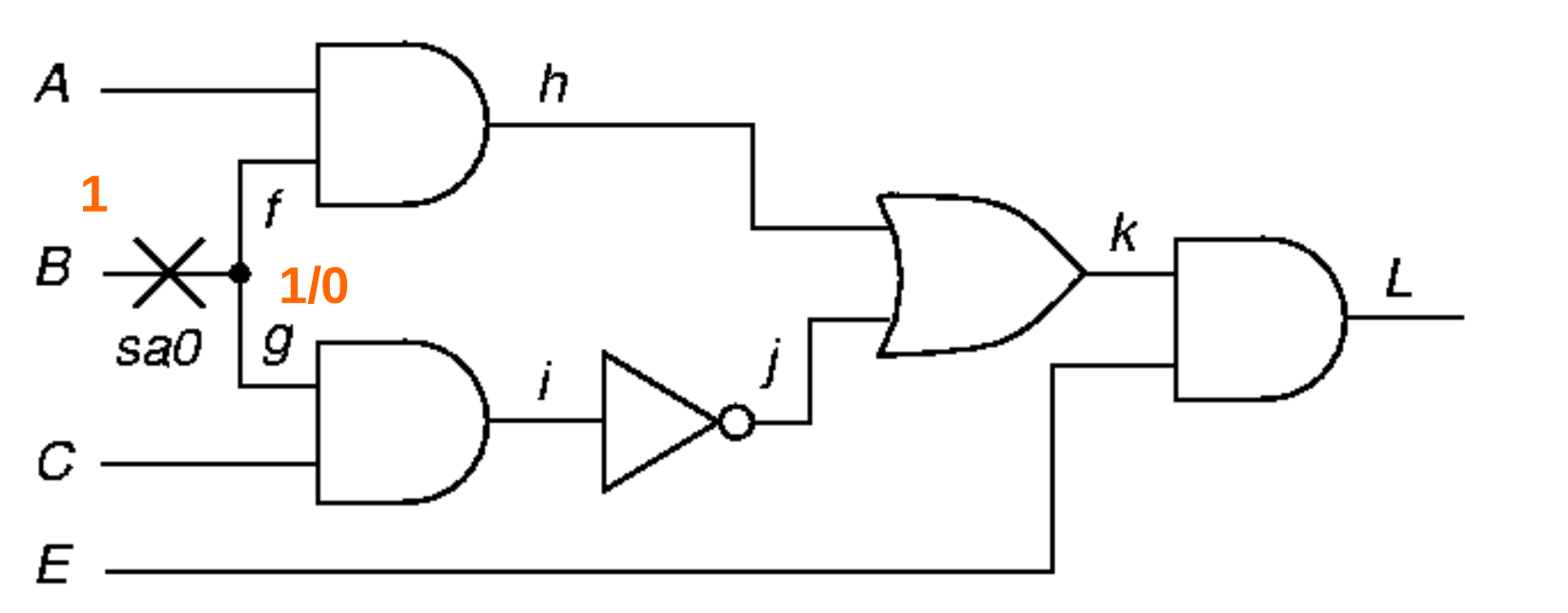
- ATPG: Automatic test pattern generation
  - *Given*
    - A circuit (usually at gate-level)
    - A fault model (usually stuck-at type)
  - *Find*
    - A set of input vectors to detect all modeled faults.
- Core solution: Find a test vector for a given fault.
- Combine the “core solution” with a fault simulator into an ATPG system.

What is a Test?



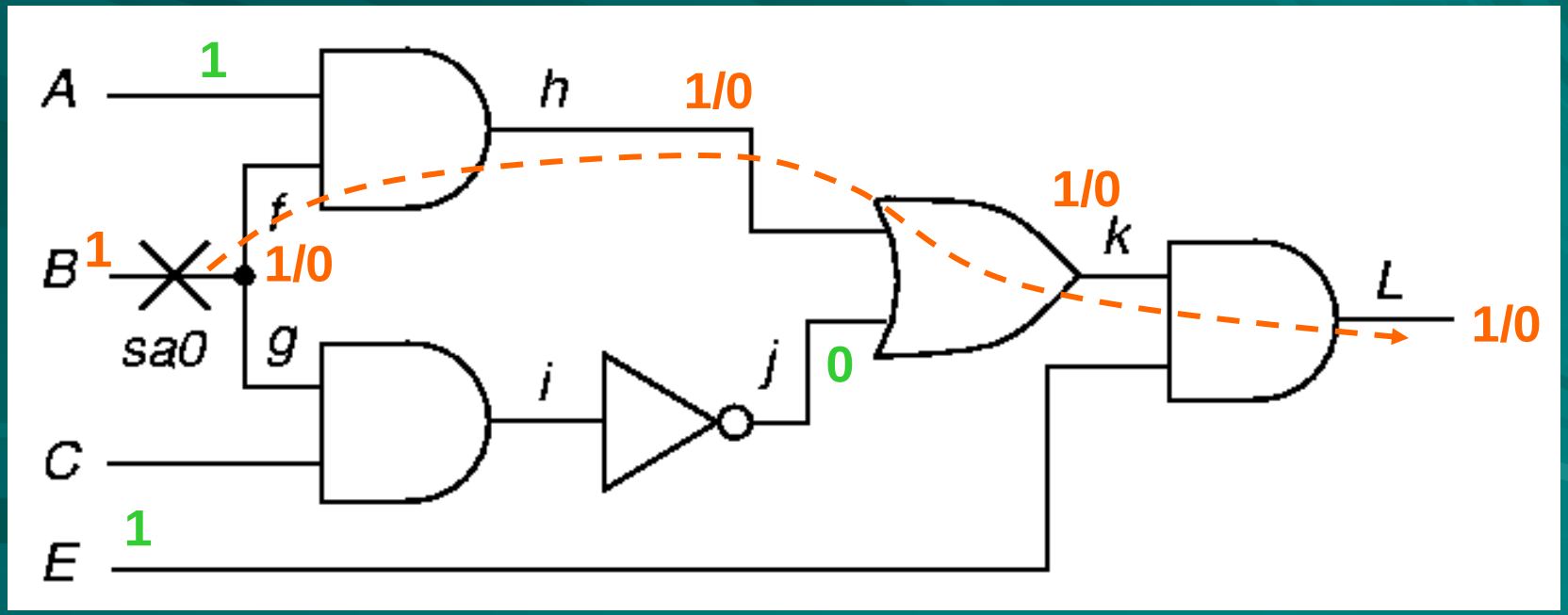
An ATPG Example

1. Fault activation
2. Path sensitization
3. Line justification



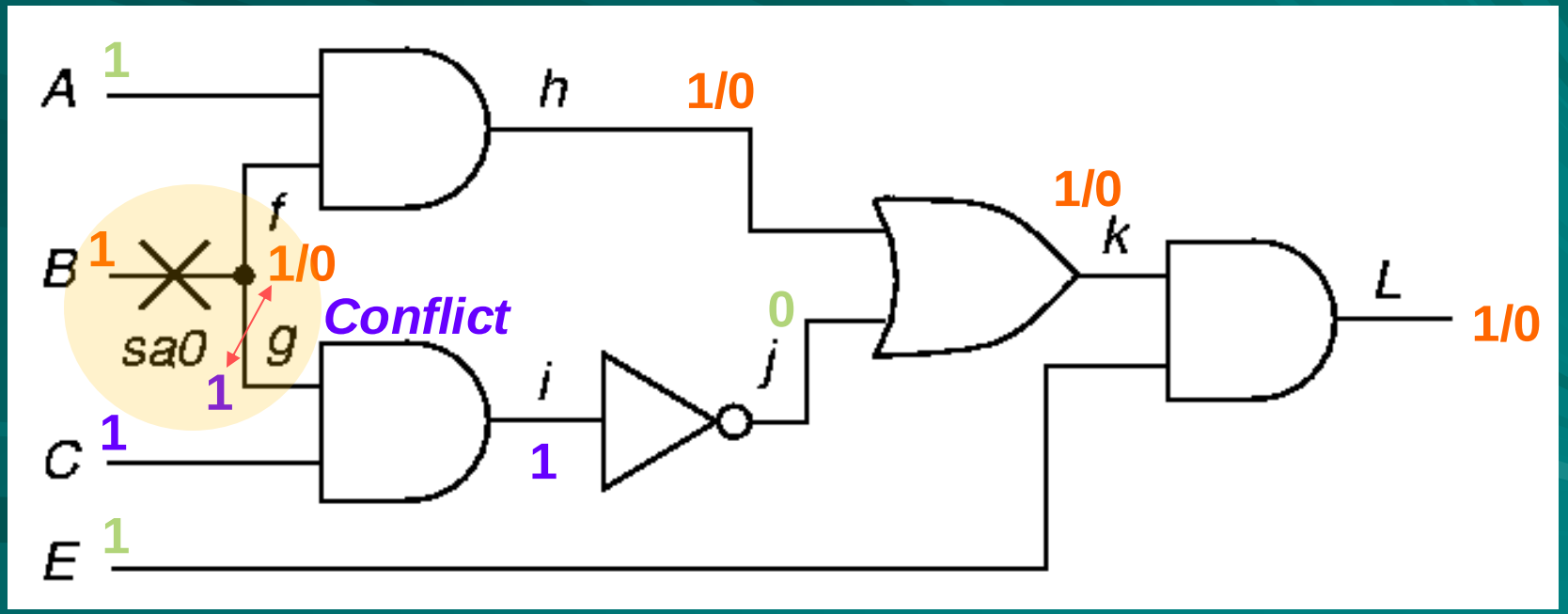
ATPG Example (Cont.)

1. Fault activation
2. Path sensitization
3. Line justification



ATPG Example (Cont.)

1. Fault activation
2. Path sensitization
3. Line justification



ATPG Example (Cont.)

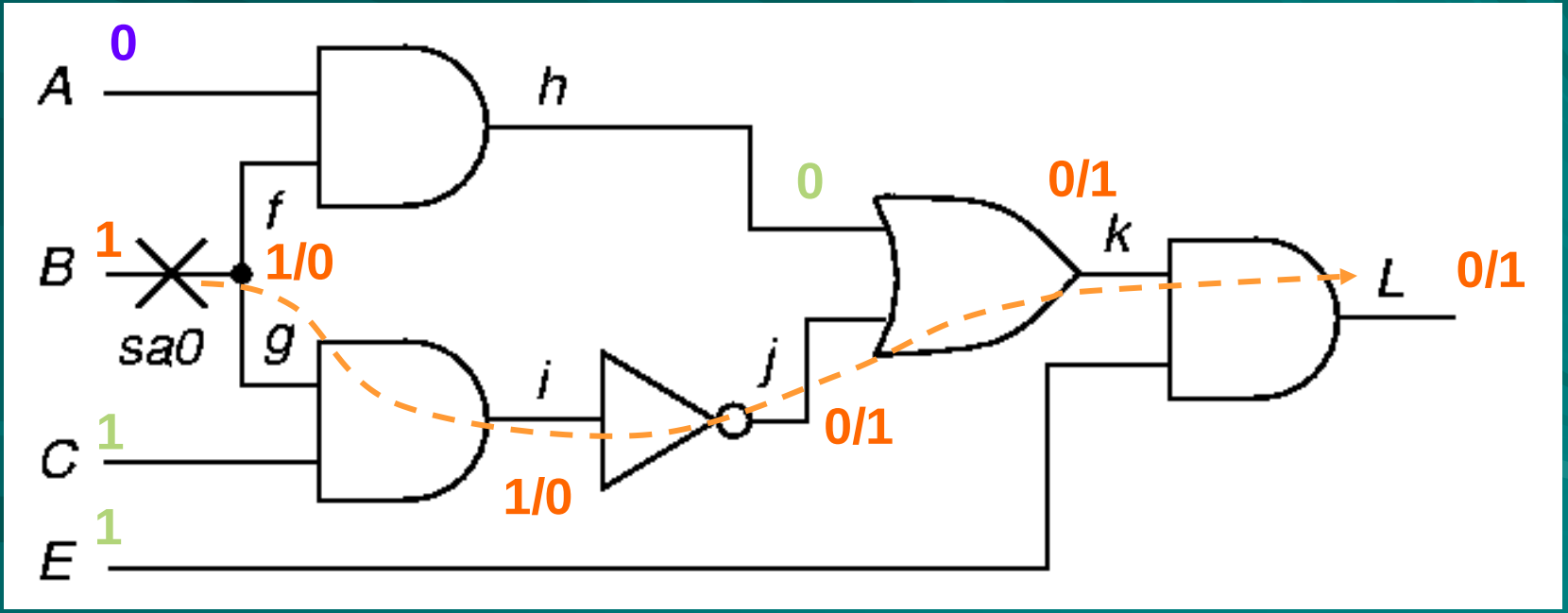
1 Fault activation

2 Path sensitization

3 Line justification

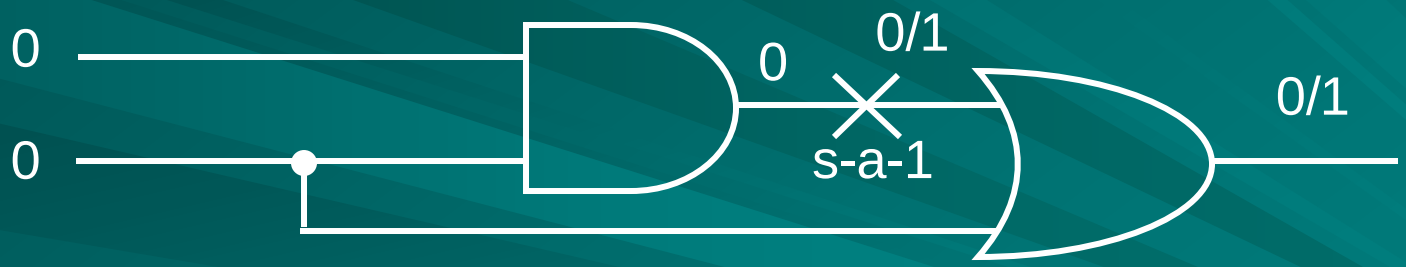
Backtrack

Test found



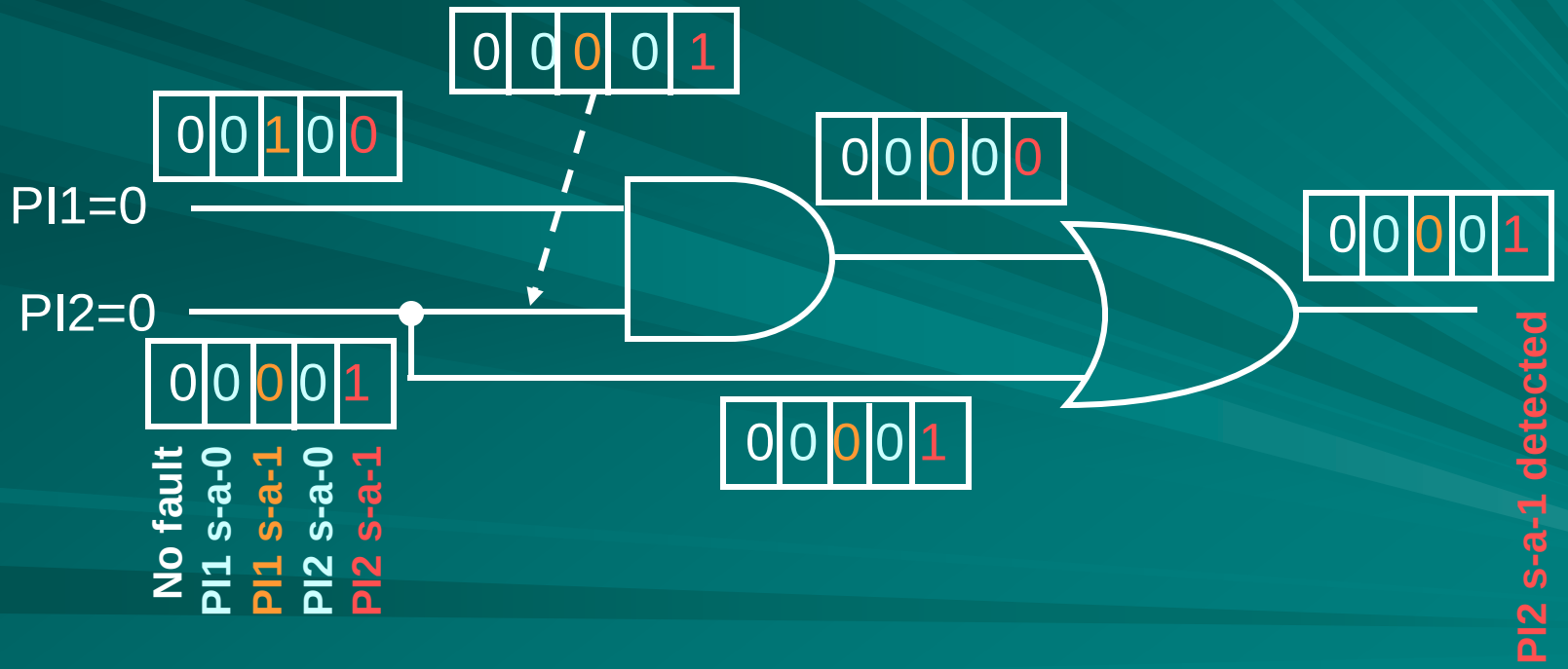
Exercise 2: Find a Test for a Fault

- A test for the stuck-at-1 fault shown in the diagram is 00.

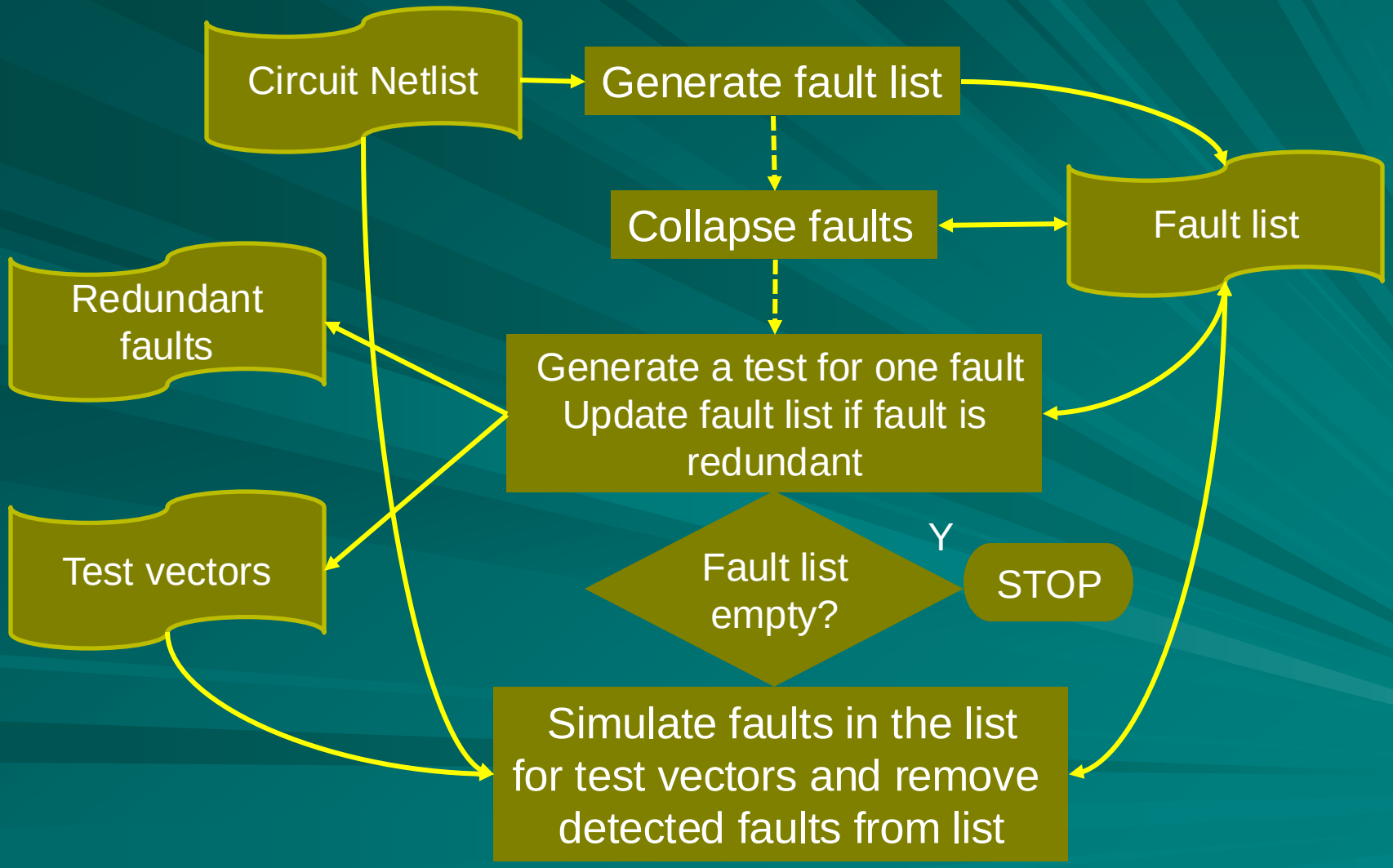


Fault Simulation: Find Faults Detected by a Vector

- Fault simulation of four PI faults is illustrated below.
  Fault PI2 s-a-1 is detected by the 00 test input.



A Test Generation System



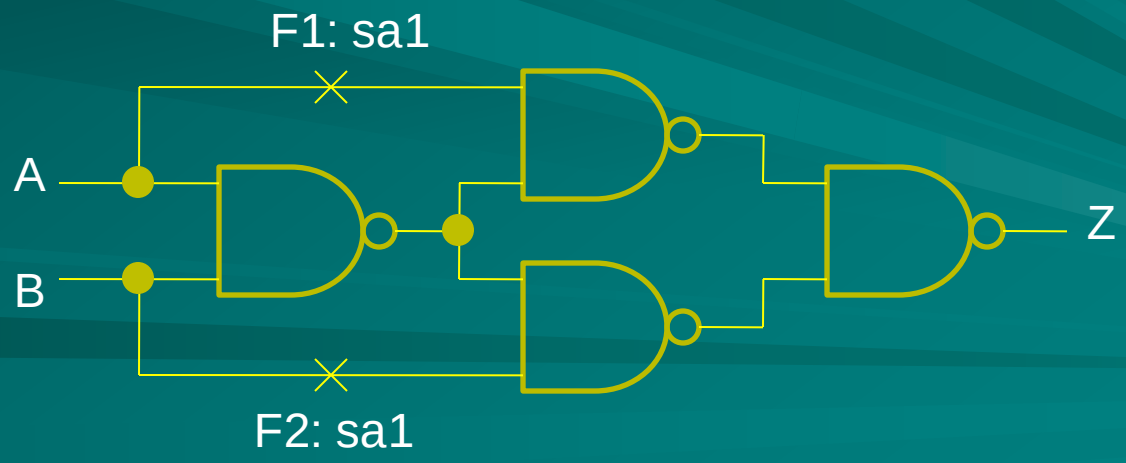
Exercise 3: Find Tests for All Faults



Proving Two Faults to be Equivalent

- Gate-level collapsing can find some, but not all equivalences. This is called *structural collapsing*.
- *Functional collapsing*: Two faults are equivalent iff the two faulty functions are identical.
- Example:

$$Z(F1) = Z(F2) = \overline{A} + \overline{B}$$

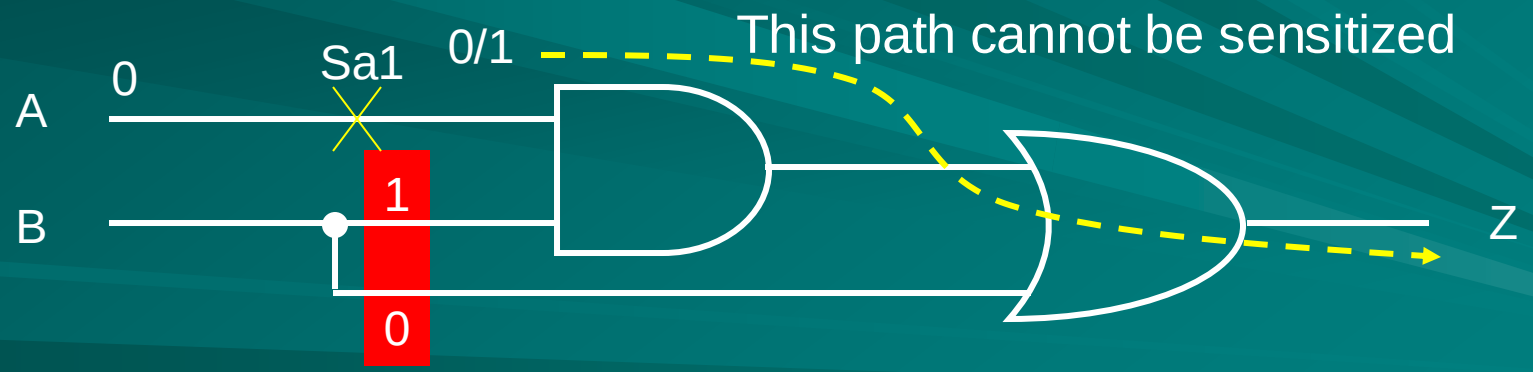


Redundant Fault

- Definition: Faulty function is identical to the fault-free function.
- No test can be found for a redundant fault.
- When a redundant fault is present in a circuit, the function is not changed.
- A circuit with a redundant fault can be simplified.

Finding a Redundant Fault

- Method 1: Compare the faulty function to the fault-free function.
- Method 2: Try to generate a test and find that no test is possible.
- Example: Z = B, Z(A Sa1) = B

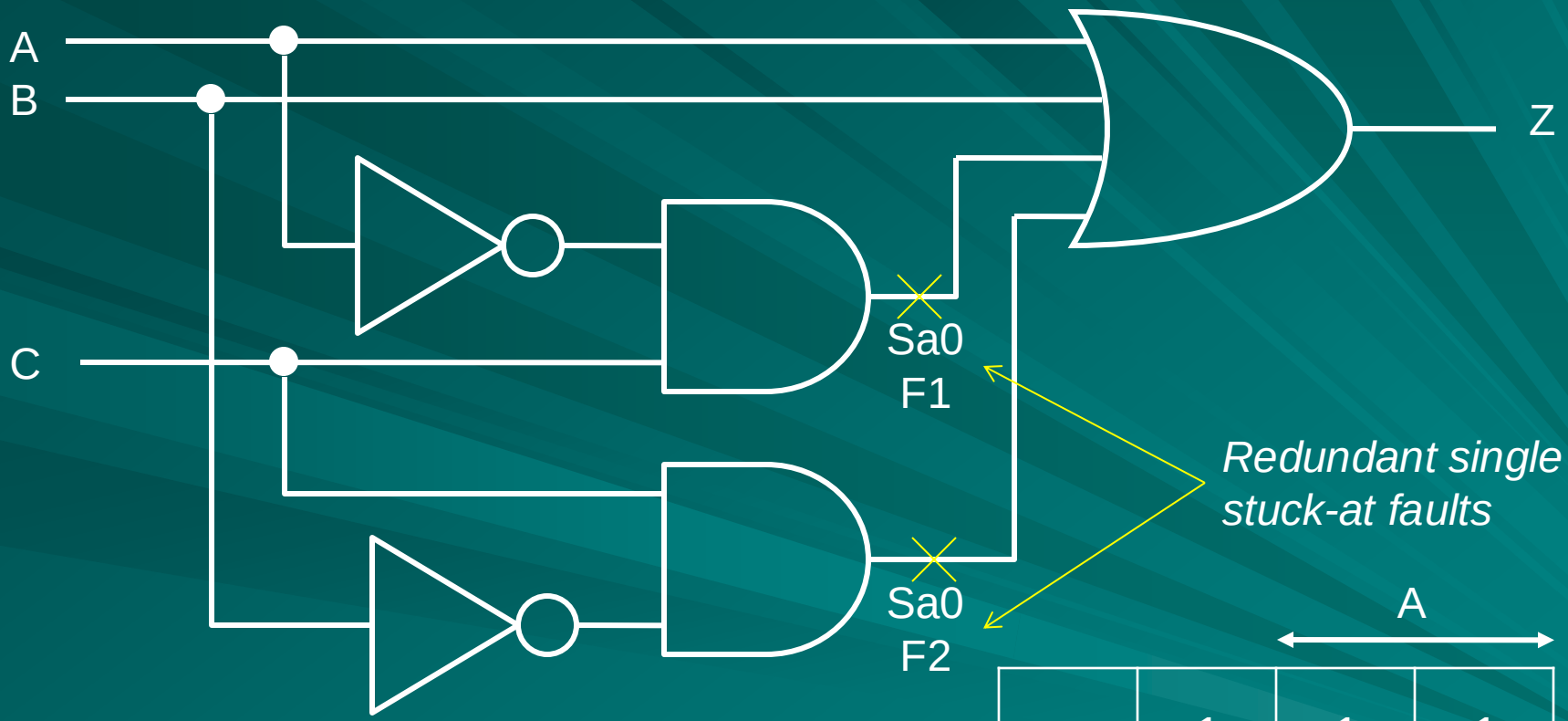


Removing Redundancy

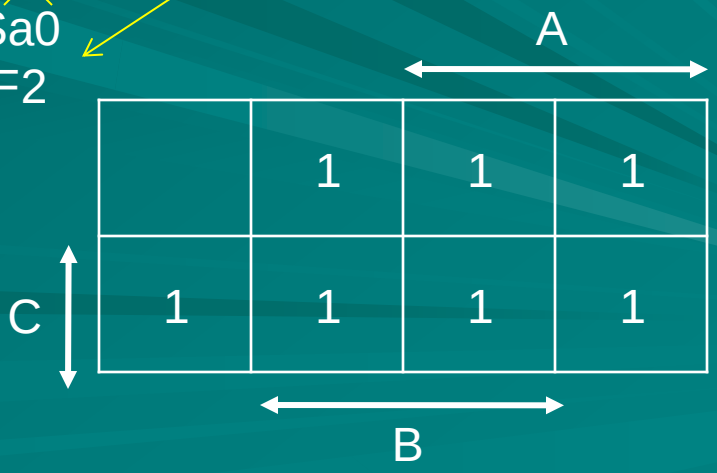
1. Find a redundant fault and insert it in the circuit (*stop if circuit is irredundant*).
2. Remove any unnecessary lines and gates.
3. Go to step 1.

   Caution: Only one fault can be inserted at one time, because insertion of a fault creates a new circuit.

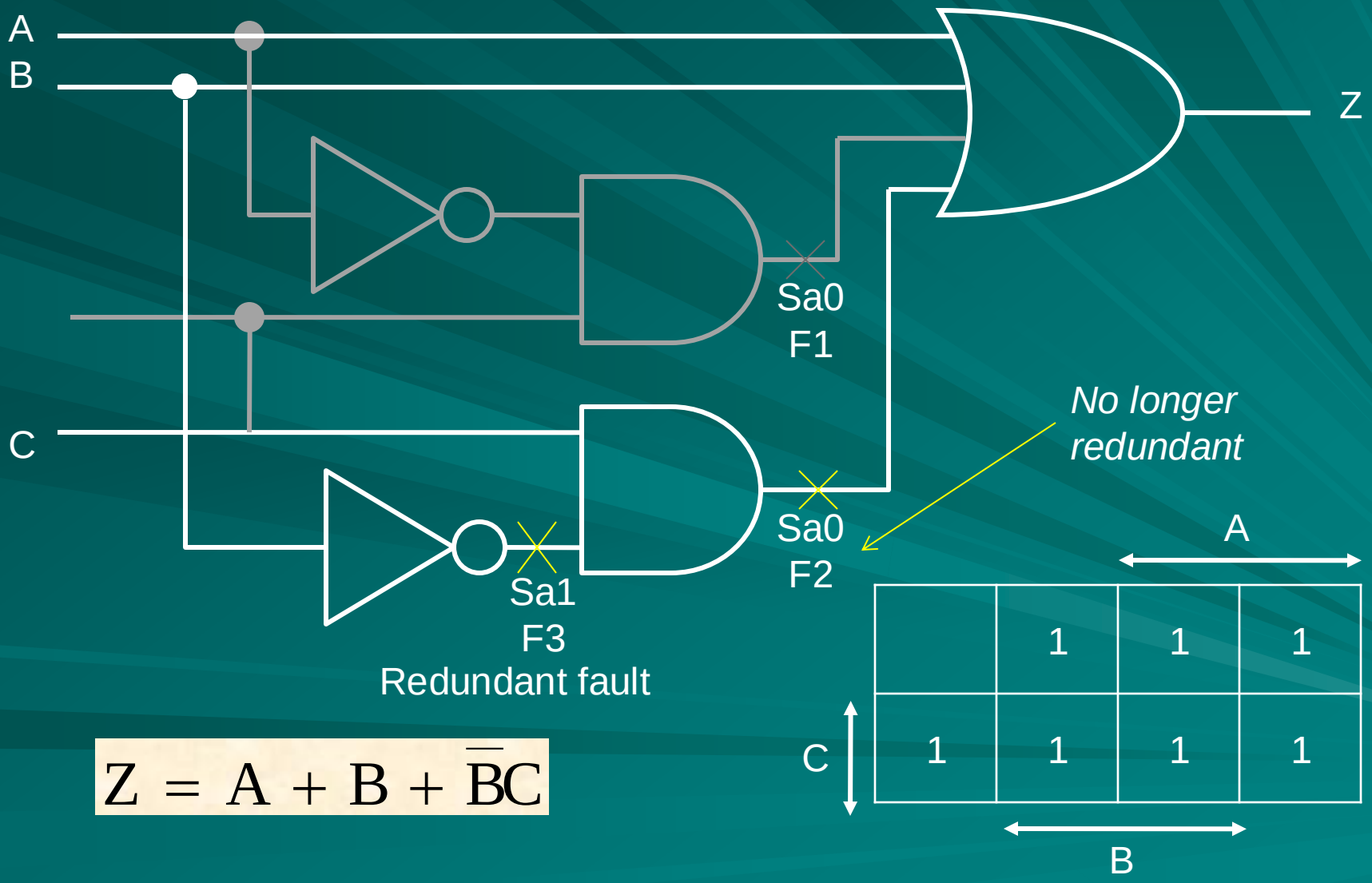
Example: Redundancy Removal



$$Z = A + B + \overline{A}C + \overline{B}C$$



F1 Removed



$$Z = A + B + \overline{B}C$$

F3 Removed

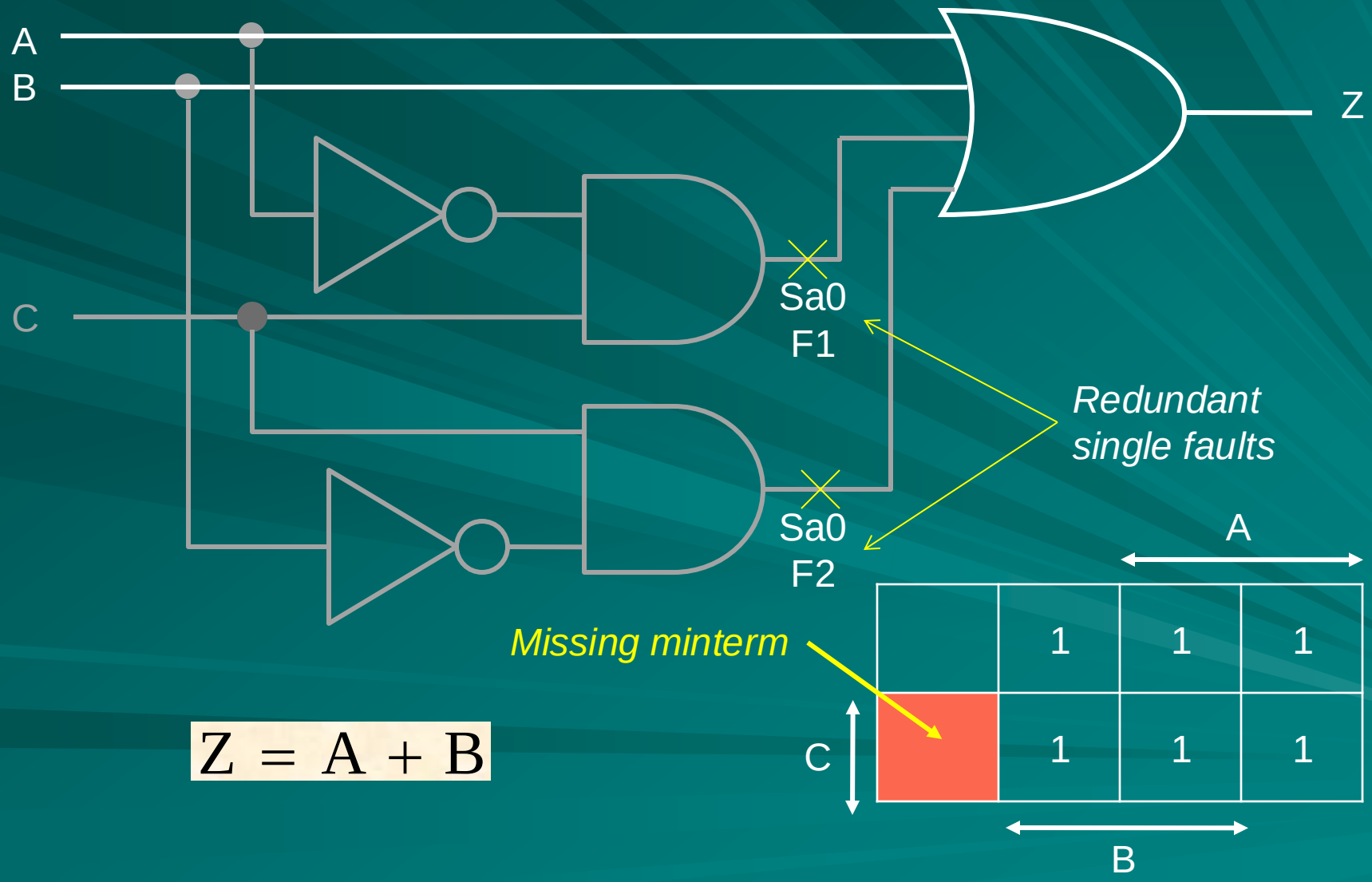


$$Z = A + B + C$$

C \ A		1	1	1
	1	1	1	1

B

Removing F1 and F2 (Incorrect)



$$Z = A + B$$

Verilog Overview

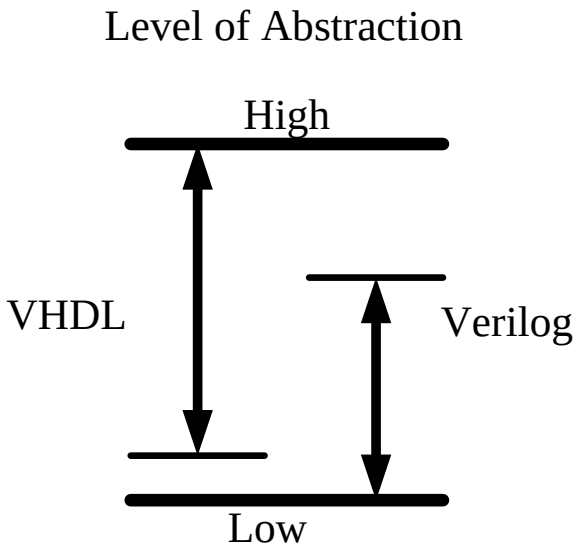
An overview of the Verilog HDL.

Lecture Overview

- A perspective.
- Some Verilog Basics
- A couple of Verilog models

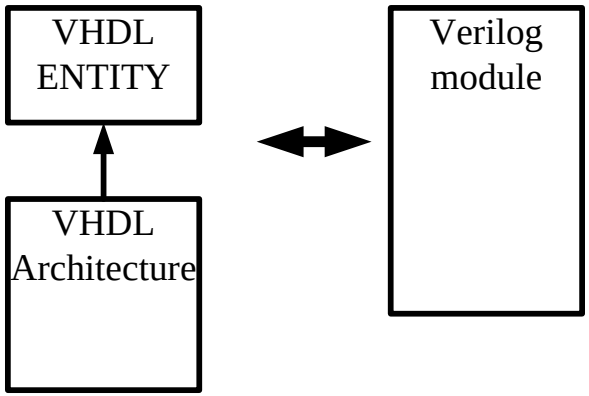
Perspective

- Verilog is another HDL modeling language
- Verilog has some aspects that give it a better low end.
- VHDL has some data types and capabilities that give it capabilities at the abstract description of components and systems



VHDL and Verilog

- In VHDL have the Entity design unit which has many possible architecture
  - Entity is interface and port specification
  - Architecture is the functional specification
- In Verilog have the module
  - Module has interfact and port specification and then the functional specification in one code unit



The Verilog module

- At the leaf of an architecture
  - module generic_unit (r,g0,g1,g2,g3,a,b);
  - output r;
  - input g0,g1,g2,g3,a,b;
  - assign r = (~a & ~b & g0) | (~a &b & g1) |
  - (a & ~b & g2) | (a & b & g3);
  - endmodule

Its testbench

```
module vtb();
    wire gout;
    reg a,b,g0,g1,g2,g3;

    generic_unit G1 (gout,g0,g1,g2,g3,a,b);

    initial begin
        #100 $finish;
    end

    initial begin
        #10 a=0; b=0; g0=1; g1=0; g2=0; g3=0;
        #10 a=0; b=0; g0=0; g1=0; g2=0; g3=0;
        #10 a=0; b=1; g0=0; g1=1; g2=0; g3=0;
        #10 a=0; b=1; g0=0; g1=0; g2=0; g3=0;
        #10 a=1; b=0; g0=0; g1=0; g2=1; g3=0;
        #10 a=1; b=0; g0=0; g1=0; g2=0; g3=0;
        #10 a=1; b=1; g0=0; g1=0; g2=0; g3=1;
        #10 a=1; b=1; g0=0; g1=0; g2=0; g3=0;
    end

endmodule
```

- The Testbench to apply tests to the unit
- Note the differences
  - No configuration
  - No declaration
  - Signals which retain a value are typed reg or wire
  - The #10 gives a 10ns delay
  - No signals in or out of this module like the VHDL testbench

Another cut at the design

- module generic_unit_m2(r,g0,g1,g2,g3,a,b);
- output r;
- input g0,g1,g2,g3,a,b;
- wire t1 = ~a & ~b & g0;
- wire t2 = ~a &b & g1;
- wire t3 = a & ~b & g2;
- wire t4 = a & b & g3;
- assign r = t1 | t2 | t3 | t4;
- endmodule

Multiple ways to do things

- As just illustrated – there are multiple ways to do the same thing
- The modules in the preceding slides are wired into vtb and simulate with the same results
- This module could be written at least 4 other ways
  - You could write it with an IF statement
  - You could write it with a CASE statement

Another Leaf Unit

```
-- the carry unit
module carry (pin,kin,cin,cout);
    input     pin,kin,cin;
    output    cout;
    assign cout = (pin & cin) | (~pin & ~kin);
endmodule
```

And a structural slice

- Can hierarchically model just like VHDL
  - module slice (r,cout,a,b,pctl,kctl,rctl,cin);
  - output r, cout;
  - input a,b,cin;
  - input [3: 0] pctl,kctl,rctl;
  - wire pint,kint;
  - generic_unit punit (pint,pctl[0],pctl[1],pctl[2],
  - pctl[3],a,b);
  - generic_unit kunit (kint,kctl[0],kctl[1],kctl[2],
  - kctl[3],a,b);
  - carry cunit (cout,pint,kint,cin);
  - generic_unit runit (r,rctl[0],rctl[1],rctl[2],
  - rctl[3],pint,cin);
  - endmodule

Hierarchy

- This slice could then be instantiated into the next higher level – the 8-bit architecture
- Verilog does not have a generate statement – would have to wire it up explicitly
- Levels of hierarchy are not limited

Behavioral modeling and the always block

- The alu behavioral model could also be done in Verilog
  - module alu_beh (a,b,cin,alu_op,r);
  - input [7:0] a,b;
  - input cin;
  - input alu_op;
  - output r;
  - // now start the modeling

Modeling

- always @ (a or b or cin or alu_op)
- begin
- case (alu_op)
- 0: begin r = a; end //op_a
- 1: begin r = b; end //op_b
- 2: begin r = ~a end //op_not a
- 3: begin r = a & b //op_a_and_b
- 12: begin r = //do binary addition
- endcase
- end

No enumeration type

- But can emulate it
  - parameter op_a=0,op_b=1,op_AandB=3. … op_aplusb=7, … ;
- And use a case statement similar to VHDL
  - case (alu_op)
  - op_a: begin … end
  - op_b: begin … end

The always block

- The always block executes whenever one of the signals in the list has a transition according to the way it is written
- Could also write it with ANDs such that all the signals used must have a transition

The register set

- VHDL Model for a refresher
- The Entity

```
LIBRARY  IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY  registers is
  PORT ( ABUS,BBUS : INOUT std_logic_vector;
        Aload,Bload : IN  std_logic;
        Adrive,Bdrive : IN std_logic;
        AregNo,BregNo : IN integer);
END  registers;
```

The architecture

```
ARCHITECTURE  behavioral OF registers IS

BEGIN

  regs : PROCESS (Aload,Bload,Adrive,Bdrive)
    TYPE reg_set_type is array (0 to 15) of std_logic_vector (ABUS'RANGE);
    VARIABLE reg_set : reg_set_type;
    CONSTANT HighImp : std_logic_vector (15 downto 0) := "ZZZZZZZZZZZZZZZZ";
   BEGIN
    IF (Aload'EVENT and Aload = '1') -- Latch A bus into register
     THEN  reg_set(AregNo) := ABUS;
    END IF;
    IF (Bload'EVENT and Bload = '1') -- Latch B bus into register
     THEN  reg_set(BregNo) := BBUS;
    END IF;
    IF (Adrive'EVENT)
     THEN
      IF (Adrive = '0') -- Drive A register onto ABUS
        THEN  ABUS <= reg_set(AregNo);
        ELSE  ABUS <= HighImp;
      END IF;
    END IF;
    IF (Bdrive'EVENT)
   THEN
       IF (Bdrive = '0') -- Drive B register onto BBUS
         THEN  BBUS <= reg_set(BregNo);
         ELSE  BBUS <= HighImp;
       END IF;
    END IF;

   END PROCESS regs;
```

Verilog first cut

```
module reg_set (abus,bbus,aload,bload,adrive,
                                bdrive,aregno,bregno);
    inout [15:0]   abus,bbus;
    input          aload,bload,adrive,bdrive;
    input   [3:0]   aregno,bregno;
    reg  [15:0]  areg0,areg1,areg2,areg3,areg4,
                    areg5,…areg14.areg15;
    reg [15:0] breg0,breg1,breg2,breg3,breg4,
                    breg5,…breg14.breg15;
    always@(aload or bload or adrive or bdrive)
      begin
```

continued

- if (aload = =1)
- case (aregno)
- 4’b0000 : areg0 = abus;
- 4’b0001 : areg1 = abus;
- 4’b0010 : areg2 = abus;
- 4’b0011 : areg3 = abus;
- 4’b1111 : areg15 = abus;
- endcase
- if (bload = =1) //same

Continued 2

```
if (adrive = = 1)
    case (aregno)
        4'b0000:   abus = areg0;
        …
    endcase
// drive of b similar
endmodule
```

Some general Verilog features

- Propagation Delays
  - Single Delay: and #3 G1 (y,a,b,c);
  - Rise/Fall Delay   and #(3,5) G2 (y,a,b)
  - Rise/Fall/Turnoff   buff0 #(3,6.5) (y,x_in,en)
  - Rise/Fall/Turnoff with Min:typ:Max
  - buff1 #(3:4:5,4:5:6,7:8:9) (y,x_in,en);

Verilog features

- Built in value set for built in logic type
  - 0 1 x z
  - Organized as registers, nets (wires), and memories
- Does have integer and real types
- Does have procedures but few references to be found as to their use. They are declared in the module where used. Verilog does not have packages.
- Several cites on web had similar figures to comparison figure given earlier

State Machines

- The T Bird taillight problem as an example
  - module tlight_cntrl(rts,lts,haz,clk,lc,lb,la,rc,rb,ra);
  - input rts,lts,haz,clk;
  - output lc,lb,la,rc,rb,ra;
  - reg [2:0] state,next_state;
  - parameter S_idle = 0, S_l1=1,S_l2=2,S_l3=3,
  - S_r1=4,S_r2=5,S_r3=6,S_lr3=7;

  - always @ (posedge clk)
  - begin
  - state = next_state;
  - end

T Bird cont 2

```
always @ (state or rts or lts or has)
 begin
  case (state)
    S_idle:  if (haz=1 | (lts=1&rts=1))
                        next_state=S_lr3;
             else if (haz=0&lts=0&rts=1)
                        next_state=S_r1;
             else if (haz=0&lts=1&rts=0)
                        next_state=S_l1;
             else    next_state=S_idle;
    S_l1 :   if (haz=1) next_state=S_lr3; else next_state=S_l2;
    S_l2 :   if (haz=1) next_state=S_lr3; else next_state=S_l3;
    S_l3 :   next_state=S_idle;
    S_r1 :   if (haz=1) next_state=S_lr3; else next_state=S_r2;
    S_r2 :   if (haz=1) next_state=S_lr3; else next_state=S_r3;
    S_r3 :   next_state=S_idle;
    S_lr3 :  next_state=S_idle;
  endcase
 end
```

The output

```
always @ (state)
  case(state)
    S_idle : begin lc=0;lb=0;la=0;ra=0;rb=0;rc=0; end
    S_l1:    begin lc=0;lb=0;la=1;ra=0;rb=0;rc=0; end
    S_l2:    begin lc=0;lb=1;la=1;ra=0;rb=0;rc=0; end
    S_l3:    begin lc=1;lb=1;la=1;ra=0;rb=0;rc=0; end
    S_r1:    begin lc=0;lb=0;la=0;ra=1;rb=0;rc=0; end
    S_r2:    begin lc=0;lb=0;la=0;ra=1;rb=1;rc=0; end
    S_r3:    begin lc=0;lb=0;la=0;ra=1;rb=1;rc=1; end
    S_lr3:   begin lc=1;lb=1;la=1;ra=1;rb=1;rc=1; end
  endcase

endmodule
```