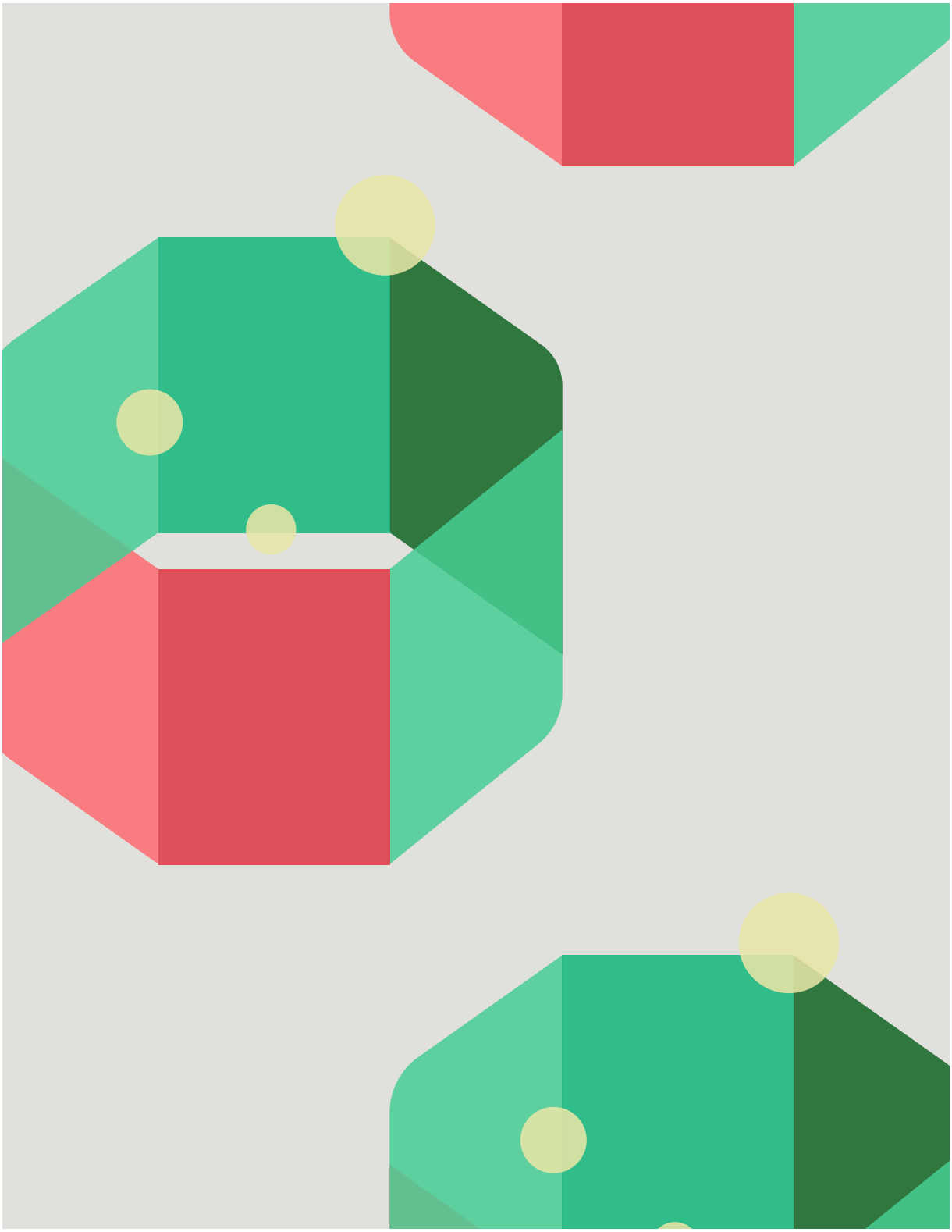


Decimal Numbers are base 10

$5374 = 5 \cdot 10^3 + 3 \cdot 10^2 + 7 \cdot 10^1 + 4 \cdot 10^0$

$10^3\ 10^2\ 10^1\ 10^0$

$$A = \sum_{i=0}^{n-1} a_i\, 10^i$$

Binary numbers are base 2

$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$

$2^3\ 2^2\ 2^1\ 2^0$

$$A = \sum_{i=0}^{n-1} a_i\, 2^i$$

Decimal to Binary conversion

method #1

- Find the largest power of 2 that fits, subtract, if not zero, repeat

method #2

- Divide by 2, put the remainder to the left of the binary representation

ex:

53_{10}

method #1

$53_{10} - 32 = 21$

$21 - 16 = 5$

$5 - 4 = 1$

$1 - 1 = 0$

1 1 0 1 0 1

$2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

method #2

	remainder
$53/2 = 26\ldots$	1
$26/2 = 13$	0
$13/2 = 6\ldots$	1
$6/2 = 3$	0
$3/2 = 1\ldots$	1
$1/2 = 0\ldots$	1

110101_2

Binary values and Range

N- digit decimal numbers

10^N values

Range : $[0, 10^N - 1]$

ex: 3 digit decimal number

$10^3 = 1000$ possibilities

Range : $[0, 999]$

N-digit binary numbers

2^N values

Range : $[0, 2^N - 1]$

ex: 3 digit binary number

$2^3 = 8$ possibilities

Range : $[0, 7] = [000_2, 111_2]$

Hexadecimal Numbers are base 16

0 1 2 3 . . . 9 10 11 12 13 14 15

9 A B C D E F

Hexadecimal to Binary conversion

$4AF_{16}$ (also written as 0x4AF) to binary

0100 1010 1111

ex:

01101011_2

0110_2 1011_2

$6B_{16}$

Hexadecimal to Decimal conversion

$4AF_{16}$ to decimal

$16^2 \cdot 4 + 16^1 \cdot 10 + 16^0 \cdot 15 = 1199_{10}$

Bits

least significant bit

10010110

most significant bit

Bytes & Nibbles

byte

10010110

nibble

Bytes

CEBF9AD7

most s. byte

least s. byte

Large Powers of 2

- 2^{10} = 1 kilo ≈ 1000
- 2^{20} = 1 mega ≈ 1 mil
- 2^{30} = 1 giga ≈ 1 bil

Estimating Powers of 2

- what is the value of 2^{24}? $2^4 \cdot 2^{20} \approx$ 16 mil
- How many values can a 32-bit variable represent? $2^2 \cdot 2^{30} \approx$ 4 bil

Signed Binary Numbers

2 ways to represent

Sign/Magnitude Numbers

1 sign bit, N-1 magnitude bits

+ → 0 − → 1

addition doesn't work ✗

ex: +6 = 0110

−6 = 1110

Two's Complement Numbers

most sig. bit is a negative power of 2

addition works ✓

ex: $1111 = -2^3 + 2^2 + 2^1 + 2^0 = -1$

Two's Complement Numbers : Flipping the sign

Method: (1) Invert the bits (2) Add 1

ex: Flip the sign of $3_{10} = 0011_2$ → $1100 + 1 = 1101 = -3_{10}$

ex: Take the complement of $6_{10} = 0110_2$ → $1001 + 1 = 1010_2 = -6_{10}$

ex: Two's complement 1001_2 decimal → $-2^3 + 2^0 = -7_{10}$

Two's Complement Numbers : Addition

ex: Add $6 + (-6)$ using two's complements

$$\begin{array}{r} {\scriptstyle 111} \\ 0110 \\ +\ 1010 \\ \hline 10000 \end{array}$$

ex: Add $-2 + 3$ using two's complements

$$\begin{array}{r} 1110 \\ +\ 0011 \\ \hline 10001 \end{array}$$

Overflow

* Carry out the most significant
* Overflow can occur when two numbers being added have the same sign bit

Increasing Bit Width

Sign extension

Number value stays same

ex: (3_{10})

0011 → 00000011

(-5_{10})

1011 → 11111011

Zero-extension

value changes for (-)'s

ex: (3_{10})

0011 → 00000011

(-5_{10})

1011 → 00001011

Binary Multiplication

very similar to normal

```
  0101
x 0111
------
  0101
 0101
0101
0000
-------
0100011
```

$5 \cdot 7 = 35$

useful for multiplying numbers

Logical and Arithmetic Shift

Logical left & right shift

(A<<N) (A>>N)

Move A to left / right by N positions, extend with 0s

ex: XXX01 <<3 = 01000

Arithmetic Shift

(A>>>N)

Move A to right, except for sign bit, by N positions, extend with sign bit

ex: 10XXX >>>3 = 11110

Shifters as multipliers & Dividers

$A << N = A \cdot 2^N$

ex: $00001 << 2 = 00100$

$(1 \cdot 2^2 = 4)$

$A >>> N = A / 2^N$

ex: $10000 >>> 2 = 11100$

$(-16 / 2^2 = -4)$

So far we saw... →

Number Systems

- Positive
  - ↳ Unsigned binary
- Negative
  - ↳ Two's complement
  - ↳ Sign / magnitude numbers

Fractional Numbers have 2 common notations:

1. Fixed Point: restricted range, faster computation
2. Floating Point: binary point floats to the right of the most sig. 1 (scientific notation). larger range of values but more expensive (more power)

Fixed Point

Integer bits	Fraction bits

Floating Point

Sign	Exponent	Mantissa

$$\pm M \times 2^E$$

(1) Fixed Point

Using 4 integer and 4 fraction bits : 0110. 1100

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

ex:

7.5625_{10} to binary

0111

$0.5625 \cdot 2 = 1.125$ Digit = 1 $p = p - 1 = 0.125$

$0.125 \cdot 2 = 0.250$ Digit = 0

$0.250 \cdot 2 = 0.5$ Digit = 0

$0.5 \cdot 2 = 1$ Digit = 1 $p = p - 1 = 0$

0111.1001

Signed Fixed Point Numbers

Sign / magnitude

$-7.5_{10} = 11111000$

Two's complements

1. +7.5 01111000
2. Invert bits 10000111
3. Add 1 to lbs + 1

10001000

② Floating-Point Numbers

Binary point FLOATS → 11010.1 → 1.10101

Step 1

a. convert decimal to binary

$228_{10} = 11100100_2$

b. binary "scientific notation"

11100100 → $1.1100100_2 \times 2^7$

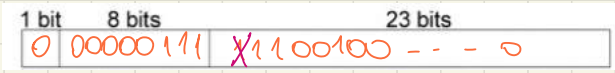
c. Fill each of 32 bits

+ 7 as in 2^7

Step 2

a. mantissa always starts with a 1 (bc scientific not.) therefore no need to store the **1**.

↳ "implicit leading 1"



Store only fraction bits in the 23-bit field

Step 3 think: how can you represent (−) exponents?

biased exponents = bias + exponent

bias = 127 (01111111_2)

exponent 7 ⟶ 127 + 7 = 134 = 10000110_2

$[0, 126] < 0$

$[127, 255] \geq 0$ } $[-126, 127]$

think of it as:
7 is the 134 th number in this range

FINAL:

Sign Biased exp. Fraction

↳ in hexadecimal: 0x43640000

Numbers with Fractions

Unsigned

N bits → 2^N values
R: $[0, 2^N - 1]$

Value "A"
→ $\lfloor \log_2 A \rfloor + 1$ bits are required

Two's complement

N bits → 2^{N-1} values
R: $[-2^{N-1}, 2^{N-1} - 7]$

Value "A"
→ $\lfloor \log_2 |A| \rfloor + 2$ bits

ex: Convert -58.25_{10} to floating point notation

$-58.25_{10} = 111010.01_2$

1.1101001×2^5

1 bit	8 bits	23 bits
1	10000100	11010010 ... 0

(-)

$127 + 5 = 132 \rightarrow$ exponent

formula to convert it back

$(-1)^{sign} \cdot 2^{(exponent-127)} \cdot 1.F$

ex: Convert -0.25_{10} to floating point notation

$0.25_{10} = 0.01_2 \rightarrow 1.0 \times 2^{-2}$

$127 + (-2) = 125$

1 bit	8 bits	23 bits
1	01111101	0000 0

(-)

ex: Convert 11000010011010010000000000000000 to decimal representation

$(-1)^{sign} \cdot 2^{(exponent-127)} \cdot 1.F$

$= -1^1 \times 2^{(132-127)} \times 1.1101001 = -1 \times 2^5 \times 1.1101001 = -58.25_{10}$

Remember! Implicit leading 1

How do we store other special cases

Number	Sign	Exponent	Fraction
0	Any	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	Any	11111111	non-zero
denormals	Any	00000000	F ≠ 0 $(-1)^s \times 2^{-126} \times 0.F$

«Not a Number» e.g.: $\sqrt{-1}$

Floating Point Precision

Single - precision format 32 - bit

Double - precision format 64 - bit

Half - precision format 16 - bit

Precision	Num. Bits	Num. Sign Bits	Num. Exponent Bits	Num. Fraction Bits	Bias	Minimum Positive Normal Value	Maximum Value	Minimum Positive Subnormal Value
Single	32	1	8	23	127	2^{-126}	$(2-2^{-23}) \times 2^{127}$	2^{-149}
Double	64	1	11	52	1023	2^{-1022}	$(2-2^{-52}) \times 2^{1023}$	2^{-1074}
Half	16	1	5	10	15	2^{-14}	$(2-2^{-10}) \times 2^{15}$	2^{-24}

Floating Point : Rounding

Rounding modes

Up ↑ (ceiling)

Toward 0 (truncation)

Down ↓ (floor)

To nearest **(default)**

Overflow: too large (round to ∞)

Underflow: too small (round to 0)

ex:

Example: round 1.100101 (1.578125) to only 3 fraction bits

- Down: 1.100
- Up: 1.101
- Toward zero: 1.100 ← if negative ↑ if positive ↓ (so literally towards zero)
- To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Floating Point Addition

Keep in mind that the results of flooding point operations may not be the exact result of the decimal since the numbers that are represented with floating points are limited.

Step 1

Decompose Operands

(cant add implicit 1)

ex: X Y

Add $3.75 + 5.125$ in floating point notation.

$3.75_{10} = 11.11_2 \rightarrow X.111 \times 2^1$ 1+127=128

0	1000 0000	111000...000
sign	exp	mantissa

$5.125_{10} = 101.001_2 \rightarrow X.01001 \times 2^2$ 2+127=129

0	1000 0001	01001000...000
sign	exp	mantissa

Step 2 Equalize Exponents (shift mantissas)

X → exp: 128 ⇝ shift right by 1

Y → exp: 129

$111000...0 >> 2 = 0111000...0$ (23 bits)

Step 3 Convert from sign magnitude to 2's complement

In this exercise there are no negative numbers however if there were, we would invert the bits & add 1.

Step 4 Add mantissas

Both now have exp = 129
So we can add mantissas
Note that regardless of signs, this step works since 2's comp.

$$\begin{array}{r} 0.1111000000\ldots \\ +\ 1.0100100000\ldots \\ \hline \cancel{1}0.0011100000 \end{array}$$

Step 5 Convert from 2's complement to sign magnitude

In this exercise there are no negative numbers however if there were, we would invert the bits & add 1.

Step 6 Normalize Result

129 + 1 = 130 (new exponent)

$$0\ \underbrace{1000\ 0010}_{\text{exp}}\ \underbrace{000\ 1110\ 0000\ \ldots\ 0000}_{\text{fraction}} = 8.875_{10}$$

Why biased exponent?

Easier to compute & store ?

Floating - Point Multiplication

1. Sum the exponents and subtract the bias
2. Multiply mantissas
3. Normalize (if needed)

ex. here

Floating - Point Subtraction and Division are ... → not common, opposite of (+) & (x)

How to subtract **binary numbers**

method 1

$0_{10} = 2_{10}$

$$\begin{array}{r} 1101 \\ -\ 1010 \\ \hline 0011 \end{array}$$

method 2

work on two's complement and sum

ex: 13-10?

$13 = 01101$

$10 = 01010$

$-10 = 10101$

$+\ 1$

10110

$13 + (-10) = 01101 + 10110 = 100011 = 00011$

* Overflow only happens if numbers have the same sign.

method 3

ex: 4.25 - 1.5 = ?

```
  0100.01
+ 1110.10
----------
 10010.11
= 0010.11 = 2.75_10
```

```
4.25 =  0100.01
1.5  =  01.1
-------------
        10.0
           1
         +
-------------
-1.5  =  10.1
```

on two's complement you always add 1 to the rightmost digit, doesn't matter if fraction

How to subtract / add hexa. numbers

ex:

```
  4 0x11 = 17_10
  51BB
- A3B
------
  4780
```

ex:

```
  +0x1 = 16_10
  4780
+ A3B
------
  51BB
```

Exercises

1. Turn 14 & 11 into base 2 and multiply.

$14_{10} = 1110$

$\times\ 11_{10} = 1011$

154

```
  1110
x 1011
------
  1110
  1110
 0000
+1110
--------
10011010
```

2. Sum and subtract 1110010 & 1010001

```
  11
 1110010
+1010001
---------
10000011
```

$1110010_2 = 72_{10}$

$1010001_2 = 81_{10}$

$72 - 81 = -9$

```
 1110010
-1010001
---------
 0100001
```

3. Given A = −24 and B = 37, turn them into 2-compl. base 2 calculate A+B and A−B and check results on base 10.

$A = -24_{10} = 11101000$

$B = 37_{10} = 00100101$

```
  1 1
 11101000
+00100101
---------
 10001101
```

$= 13_{10}$

```
  11 11
 11 10 1000   -24
+11 01 1011
-----------
 11000011
```

$= -61_{10}$

```
  11011010
+         1
-----------
 11011011
```

$-37_{10} = 11011011$

4. Consider X = 0x45A00000 and Y = 0xC5100000, convert to IEEE 754 notation and add them. Show as hexadecimal

X = 0x45A00000 = 0100 0101 1010 000 ... 0

Y = 0xC5100000 = 1100 0101 0001 000 ... 0

X { sign = 0 (+) exponent = 10001011 = 139

139 − 127 = 12

fraction = .01 → 1.010 = 1.25_{10}

$= 1.25 \times 2^{12}$

Y { sign = 1 (−) exponent = 10001010 = 138

138 − 127 = 11

fraction = .001 → 1.001 = 1.125_{10}

$= -1.125 \times 2^{11}$

$X = 1.010 \times 2^{12}$

$Y = 1.001 \times 2^{11} = 0.1001 \times 2^{12}$

Xm = 1.010 → 01.0100

Ym = 0.1001 → 00.1001 → 11.0110

it became positive, we shift so its (-) again

11.0110 + 1 = 11.0111

01.0100 (1.25)

+ 11.0111 (-0.5625)

100.1011 → $0.1011 \times 2^{12} = 1.011 \times 2^{11}$

sign = 0 exp. = 11 + 127 = 138_{10} = 10001010 } hexa : 0x45300000

fract. = 01100..0

5. X = 111 and Y = 78, convert to 2's complement, add and subtract them, convert to hexa decimal.

X = 01101111 Y = 01001110 → −Y = 10110010 (10110001 + 1)

$X+(-Y) = 01101111 + 10110010 = 100100001 = 33_{10}$

$= 0x21$

$X+Y = 01101111 + 01001110 = 10111101 = 189_{10}$

$= 0xBD$

6. Compute W = X + Y (X = 0x3EAB, Y = 2E73) in base 16. Convert to base 10 and verify your answer.

$$\begin{array}{r} 3EAB \\ +\ 2E73 \\ \hline 6D1E \end{array} = 6\times16^3 + 13\times16^2 + 1\times16^1 + E\times16^0 = 27934_{10}$$

7. Compute W = X − Y (X = 0x51BB, Y = 0xA3B) in base 16. Consider it as a rational IEEE half-precision, multiply by 1100011000000000.

$$\begin{array}{r} 51BB \\ -\ A3B \\ \hline 4780 \end{array}$$

0	10001	1110000000
sign	5 exp	10 fract.

real exp. = 17 − 15 = 2

mantissa = 1.111

0	10001	1110000000	mantissa = 1.111
1	10001	1000000000	mantissa = 1.1

exp → 10001 + 10001 = 100010 (sum the exponent)

100010 − 0001111 = 0100010 (subtract the bias)

0100010 + 1110001 = X010011 = 19_{10}

overflow between a (+) and a (−) doesnt matter

mantissas → 1.111 × 1.1 = 1111 + 1111 = 101101. → 10.1101

→ Scientific notation

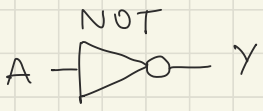
$1.01101 \times 2^1 \times 2^{19}$

$= 1.01101 \times 2^{20}$

Logic Gates

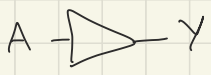
Simple digital circuits working on binary variables.

Single-input logic gates



Gate

BUF



$Y = \bar{A}$ (complement)

Boolean equation

$Y = A$

A	Y
0	1
1	0

A	Y
1	1
0	0

Truth table

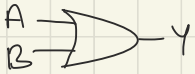
Two - input logic gates

AND

$Y = A \cdot B$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR



$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR NAND (not and) NOR (not or) XNOR

Some extra 2-inp. logic gates

XOR

$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

1 iff one of the inputs = 1

NAND

$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR

$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

$Y = \overline{A \oplus B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Multiple Input Logic Gates

NOR3

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

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

it is true when all inputs are 0

true when all inputs 1 are

AND3

$Y = ABC$

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3-input XOR gate (A⊕B⊕C)

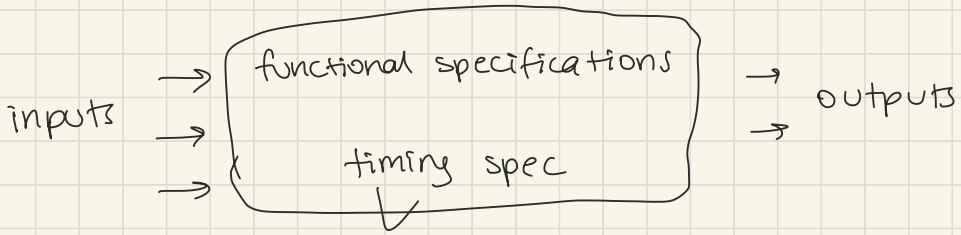
A	B	C	Y (Out)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

example

Odd parity

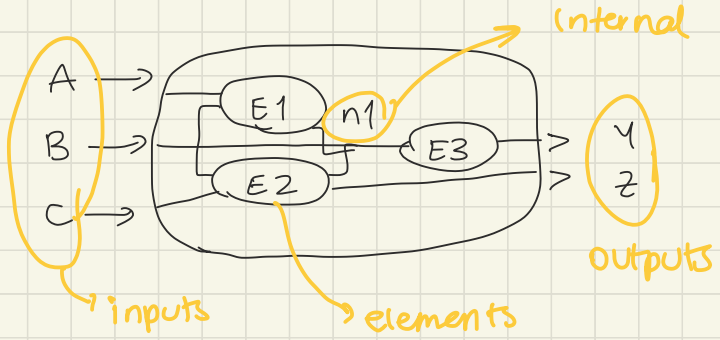
If you have an odd number of 1 bits, output is 1.

Logic Circuits



delay between inputs changing and outputs responding

ex:



Combinational Logic

memoryless

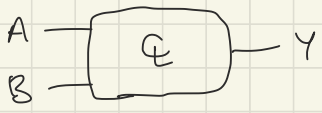
Outputs determined by current inputs

Sequential Logic

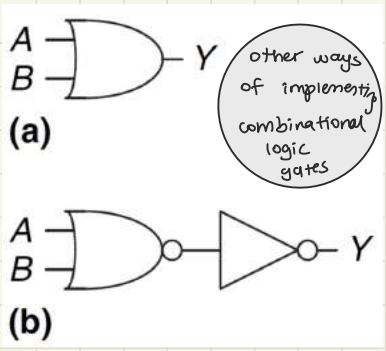
Has memory

Outputs determined by previous and current inputs

→ CL: Combinational Logic



$Y = F(A, B) = A + B$



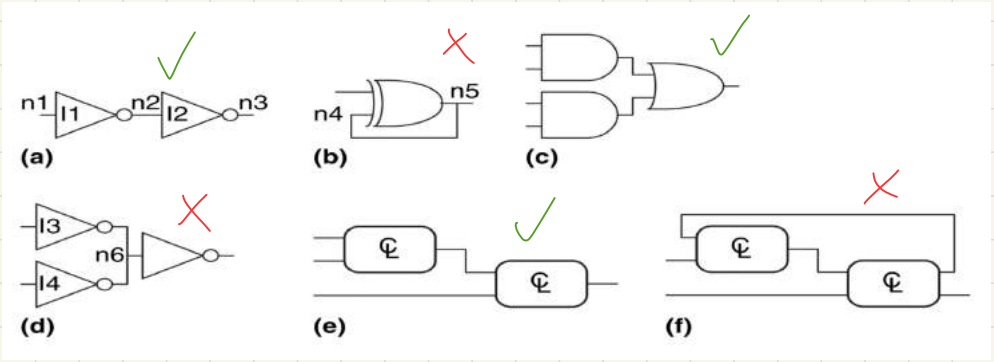
Rules of Combinational Composition

Every element is combinational (inp → out)

Every node is either an input or connects to **exactly** 1 output

No cyclic paths (no paths returning)

ex:



Boolean Equations

Every truth table can be expressed as a boolean equation

ex:

$S = F(A, B, C_{in})$

$C_{out} = F(A, B, C_{in})$

$S = A \cdot B \cdot C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Some Definitions

OR → known as sum AND → known as product

Complement inverse of a variable (bar over letter)

$A = 0$, $\bar{A} = 1$

Literal variable or its complement $(A, \bar{A}, \bar{B}, B, \bar{C}, C)$

Implicant AND of one or more literals

$AB\bar{C}$, $\bar{A}C$, BC

Minterm product with all input variables

$AB\bar{C}$, $\overline{ABC}$, ABC

Maxterm sum of all input variables

$(A + \bar{B} + C)$, $(\bar{A} + B + \bar{C})$, $(\bar{A} + \bar{B} + C)$

Operator Presedence

$Y = (A \text{ OR } B) \text{ AND } C$ vs $Y = A \text{ OR } (B \text{ AND } C)$

Sum of Products

* each row has **minterms**
* table with p inputs → $n = 2^p$
* all equations can be written in SOP form
* we build minterms that always give 1
* we take the rows where the output is 1.

A	B	Y	minterm	minterm name
0	0	0	$\bar{A}\,\bar{B}$	m_0
0	1	1	$\bar{A}\,B$	m_1
1	0	0	$A\,\bar{B}$	m_2
1	1	1	$A\,B$	m_3

$$Y = F(A,B) = \bar{A}B + AB = \Sigma(1,3)$$

Product of Sums

* each row has **maxterms**
* table with p inputs → $n = 2^p$
* all equations can be written in POS form
* we build maxterms that always give 0
* we take the rows where the output is 0

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \bar{B}$	M_1
1	0	0	$\bar{A} + B$	M_2
1	1	1	$\bar{A} + \bar{B}$	M_3

$$Y = F(A,B) = (A+B)(\bar{A}+B) = \Pi(0,2)$$

SOP and POS are logically equivalent!!

SOP produces a shorter equation when the output is TRUE on only a few rows of a truth table

POS is shorter when the output is FALSE on only a few rows of a truth table

ex:

Going to cafeteria for lunch

→ won't eat lunch ($\bar{E}$)

→ cafeteria not open ($\bar{O}$)

→ if they only serve corndogs (C)

O	C	E
0	0	0
0	1	0
1	0	1
1	1	1

SOP
(sum of products)

	O	C	E	minterms
m_0	0	0	0	$\bar{O} \cdot \bar{C} = 1$
m_1	0	1	0	$\bar{O}\, C$
m_2	1	0	1	$O\, \bar{C}$
m_3	1	1	0	$O\, C$

$E = O\bar{C} = \Sigma(2)$

POS
(product of sums)

O	C	E	maxterms
0	0	0	$O + C = 0$
0	1	0	$O + \bar{C}$
1	0	1	$\bar{O} + C$
1	1	0	$\bar{O} + \bar{C}$

$E = (O+C) + (O+\bar{C}) + (\bar{O}+\bar{C})$

$= \Pi(0,1,3)$

Boolean Algebra

Axioms and theorems simplify boolean equations

Duality → ANDs vs ORs (·) (+) 0's and 1's interchanged

Boolean Theorems with one variable

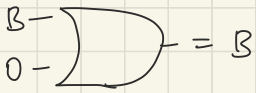
Number	Theorem	Dual	Name
T1	$B \cdot 1 = B$	$B + 0 = B$	Identity
T2	$B \cdot 0 = 0$	$B + 1 = 1$	Null Element
T3	$B \cdot B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$		Involution
T5	$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$	Complements

Dual: Replace: · with +
0 with 1

T_1 Identity Theorem

$B \cdot 1 = B$

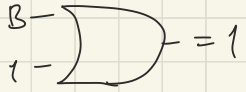
$B + 0 = B$



T_2 Null Element

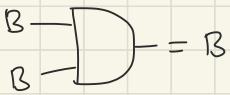
$B \cdot 0 = 0$

$B + 1 = 1$

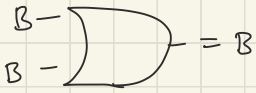


T_3 Idempotency

$B \cdot B = B$



$B + B = B$



T_4 Involution (Double Negation)

$\overline{\overline{B}} = B$

T_5 Complements

$B \cdot \overline{B} = 0$

$B + \overline{B} = 1$

Boolean theorems with several variables

#	Theorem	Dual	Name
T6	B·C = C·B	B+C = C+B	Commutativity
T7	(B·C) · D = B · (C·D)	(B + C) + D = B + (C + D)	Associativity
T8	B · (C + D) = (B·C) + (B·D)	B + (C·D) = (B+C)·(B+D)	Distributivity
T9	B · (B+C) = B	B + (B·C) = B	Covering (or Absorption)
T10	(B·C) + (B·$\overline{C}$) = B	(B+C) · (B+$\overline{C}$) = B	Combining

$\longrightarrow \quad \bar{A}B + A = B + A$

Warning: T8' differs from traditional algebra:
OR (+) distributes over AND (·)

Number	Theorem	Dual	Name
T11	(B·C) + ($\overline{B}$·D) + (C·D) = (B·C) + ($\overline{B}$·D)	(B+C) · ($\overline{B}$+D) · (C+D) = (B+C) · ($\overline{B}$+D)	Consensus

ex:

Prove

Number	Theorem	Name
T9	B· (B+C) = B	Covering

by using:

#1 – method of perfect induction

B	C	B+C	B · (B+C)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

#2- method of proof through theorems

$$
\begin{aligned}
B\cdot(B+C) &= B\cdot B + B\cdot C && \text{T8: Distributivity}\\
&= B + B\cdot C && \text{T3: Idempotency}\\
&= B\cdot(1+C) && \text{T8: Distributivity}\\
&= B\cdot 1 && \text{T2: Null element}\\
&= B && \text{T1: Identity}
\end{aligned}
$$

Simplifying an Equation

ex:

Prove that $PA + \bar{A} = P + \bar{A}$

method #1

$$
\begin{aligned}
PA + \bar{A} &= PA + (\bar{A} + \bar{A}P) && \text{T9' Covering } (B+(BC)=B)\\
&= PA + P\bar{A} + \bar{A} && \text{T6 Commutativity}\\
&= P(A+\bar{A}) + \bar{A} && \text{T8 Distributivity}\\
&= P(1) + \bar{A} && \text{T5' Complement}\\
&= P + \bar{A} && \text{T1 Identity}
\end{aligned}
$$

$PA + \bar{A} = (\bar{A} + A)(\bar{A} + P)$ — T8' Distributivity

$= 1\,(\bar{A} + P)$ — T5' Complement

$= \bar{A} + P$ — T1 Identity

method #2

ex:

Prove T11 using axioms and theorems

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

$$= BC \cdot 1 + \bar{B}D \cdot 1 + CD \cdot 1$$

$$= BC\,(\bar{B} + B) + \bar{B}D\,(\bar{B} + B) + CD\,(\bar{B} + B)$$

$$= \cancel{BC\bar{B}}^{\,0} + BCB + \bar{B}D\bar{B} + \cancel{B\bar{B}D}^{\,0} + CD\bar{B} + CDB$$

$$= BC + \bar{B}D + CD\bar{B} + CDB$$

$$= BC(1 + D) + \bar{B}D\,(1 + C)$$

$$= BCD + \bar{B}DC$$

How to approximate binary numbers

Method 1 – Convert to decimal

0.11001_2 Can either become 0.11_2 or 1.00_2

$0.11_2 = 0.75_{10} \rightarrow$ 0.11001_2 is closer to

$1.00_2 = 1.00_{10} \rightarrow$ 0.75_{10} than 1.00_{10}

rounded to $0.75_{10} = 0.11_2$

method 2 - Formula way

Check the first digit that we drop:
- If it is a 0, we round down (i.e., just drop the extra bits)
- If it is a 1:
  - If any of the following digits is a 1, we round up
  - Otherwise, the number is equidistant and we round to the number with 0 as LSB

Examples:
0.11001_2: the first digit (among those we drop), is 0. We round to 0.11_2
0.11101_2: the first digit (among those we drop), is 1. There is another 1 afterwards (the LSB). We round to 1.00_2
0.11100_2: the first digit (among those we drop), is 1. There are no other 1 afterwards. The number is equidistant, we pick the one with 0 as LSB (i.e., 1.00_2)

Converting POS to SOP

method 1

$Y = (A + C + D + E)(A + B)$

$= \cancel{AA} + \cancel{AB} + \cancel{AC} + BC + \cancel{AD} + BD + \cancel{AE} + BE$

$= A(1 + B + C + D + E) + BC + BD + BE$

$= A + BC + BD + BE$

method 2

apply when T8'
$W + XZ = (W + X)(W + Z)$
is possible

$Y = (A + C + D + E)(A + B)$

Let $X = (C + D + E)$

$Y = (A + X)(A + B)$

$= A + XB = A + BC + BD + BE$

Converting SOP to POS

ex:

$Y = (A + \bar{B}CDE)$

apply when T8'
$w + xz = (w + x)(w + z)$
is possible

Let $X = \bar{B}C$, $Z = DE$

$Y = (A + XZ)$

$= (A + X)(A + Z) = (A + \bar{B}C)(A + DE)$

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

ex:

$Y = AB + \bar{C}DE + F$

method 2

apply when T8'
$w + xz = (w + x)(w + z)$
is possible

$Y = (AB + \underbrace{\bar{C}}_{X}\underbrace{DE}_{Z}) + F$

$= (\underbrace{AB}_{Y} + \underbrace{\bar{C}}_{Z}) + (AB + DE) + F$

$=$

1

1

1

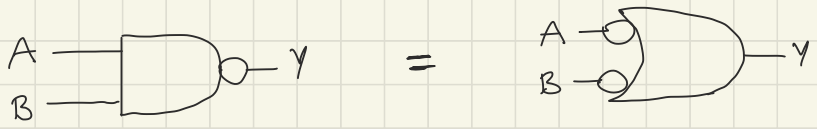
DeMorgan's Theorem

#	Theorem	Dual	Name
T12	$\overline{B_0 \cdot B_1 \cdot B_{2\ldots}} = \overline{B_0} + \overline{B_1} + \overline{B_{2\ldots}}$	$\overline{B_0 + B_1 + B_{2\ldots}} = \overline{B_0} \cdot \overline{B_1} \cdot \overline{B_{2\ldots}}$	DeMorgan's Theorem

The complement of product is the sum of complements ★

The complement of sum is the product of complements

ex: $Y = \overline{AB} = \overline{A} + \overline{B}$



ex: $Y = \overline{A+B} = \overline{A} \cdot \overline{B}$

ex: $Y = \overline{(A + \overline{BD})\,\overline{C}} = \overline{(A + \overline{BD})} + \overline{\overline{C}} = (\overline{A} \cdot \overline{\overline{BD}}) + C$

$= (\overline{A} \cdot BD) + C = \overline{A}BD + C$

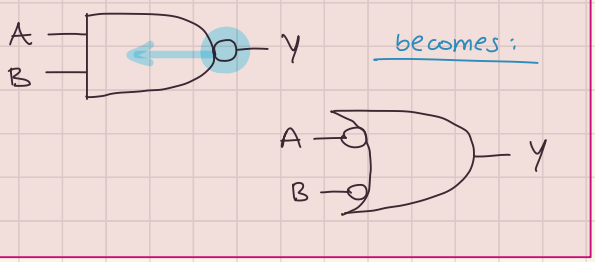
ex: $Y = \overline{(\overline{ACE} + \overline{D}) + B} = \overline{(\overline{ACE} + \overline{D})} \cdot \overline{B} = \overline{\overline{ACE}} \cdot \overline{\overline{D}} \cdot \overline{B}$

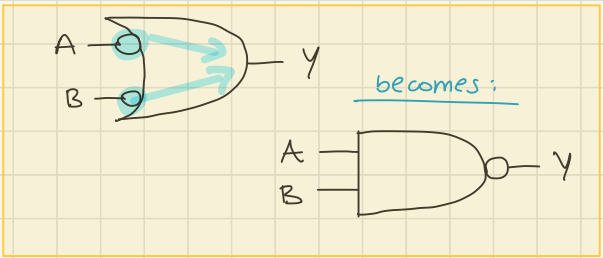
$= (\overline{\overline{AC}} + E) \cdot D\overline{B} = A\overline{B}CD + \overline{B}D\overline{E}$

Bubble Pushing

Backward

- Body changing
- Adds bubbles to input

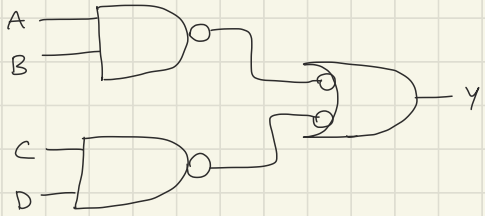




Forward

- Body changing
- Adds bubbles to output

ex: What is the Boolean expression of this circuit?



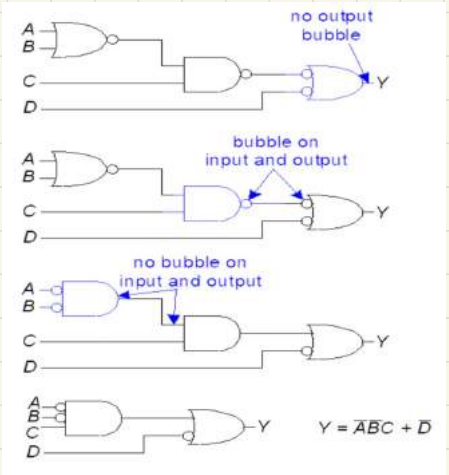
$$Y = \overline{\overline{AB}} + \overline{\overline{CD}}$$

$$= AB + CD$$

Bubble Pushing Rules

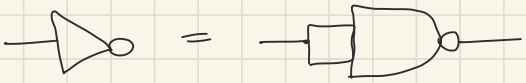
- Begin at output, work towards input
- We dont want bubbles on output
- Cancel out bubbles (ex: $\bar{\bar{A}} = A$)

ex:



Completeness

NAND (and NOR) is functionally complete (universal)

NOT 

AND 

OR 

CMOS is a type of transistors fabrication process

CMOS logic prefers the use of NAND and NOR's.

ex:

Bubble pushing for CMOS Logic

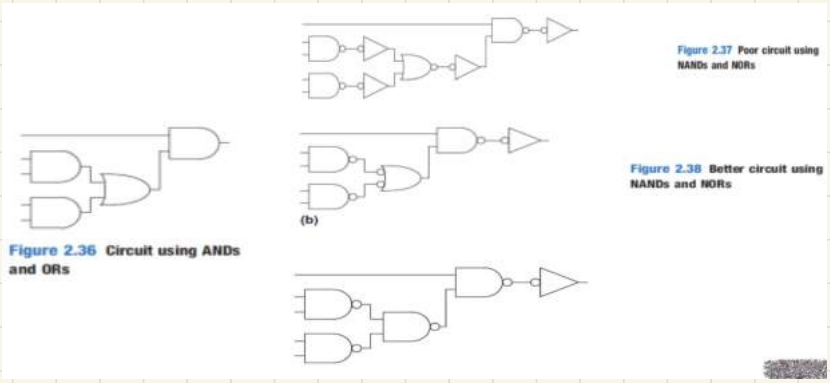


Figure 2.36 Circuit using ANDs and ORs

(b)

Figure 2.37 Poor circuit using NANDs and NORs

Figure 2.38 Better circuit using NANDs and NORs

ex: Solve the De-Morgan example

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

$$= \bar{A}BC + \bar{B} + C + BC = BC(\bar{A}+1) + \bar{B} + C$$

$$= BC + \bar{B} + C = C(B+1) + \bar{B} = C + \bar{B}$$

ex: $\overline{(A+B+C)}D + AD + B = \bar{A}\cdot\bar{B}\cdot\bar{C}\cdot D + AD + B$

$$= \bar{A}\bar{B}\bar{C}D + AD + B + \bar{A}BC\bar{D}$$

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

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

$$A \oplus C = (\bar{A}+\bar{C})(A+C)$$

ex: $ABCD + \bar{A}B\bar{C}D + \overline{(\bar{B}+D)}E$

$$ABCD + \bar{A}B\bar{C}D + B\cdot\bar{D}\cdot\bar{E} = ABCD + \bar{A}B\bar{C}D + B\bar{D}E$$

$$= BD(AC + \bar{A}\bar{C}) + B\bar{D}E$$

$= (\bar{A}+\bar{C})(A+C)$

From SOP – POS to NOR – NAND

NAND
$Y = \overline{AB}$

Since $\bar{\bar{A}} = A$,

$$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

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

to convert to NAND we add double negation and distribute it

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

this is now in NAND form

$$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

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

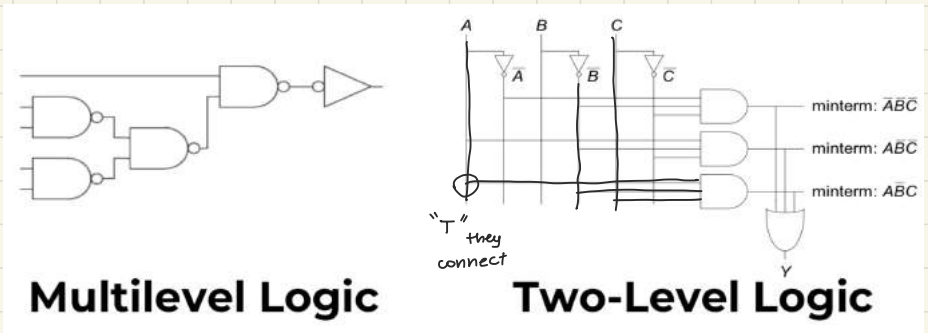
double negation on each

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

NOR form

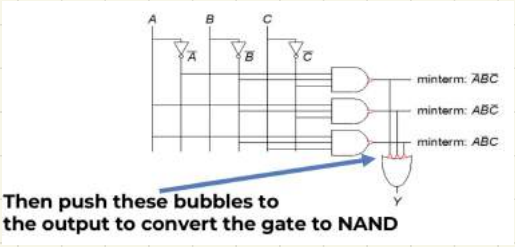
NAND's and NOR's are required because of irl applications of logic gates

From Logic to Gates



Two-Level Logic : ANDs followed by ORs

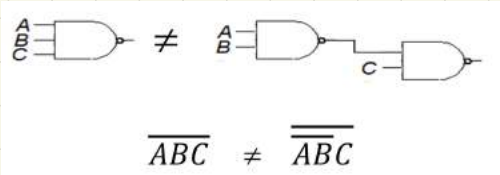
Also known as programable logic array (PLA)



Can be transformed into a NAND – NAND network

Caveat

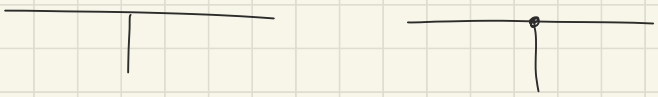
NAND and NOR ports are NOT associative



Two-Level Logic – Circuit Schematic Rules

Inputs on the left (top)
Outputs on the right (bottom)
Gates flow from left → right

Straight wires at all times



connects

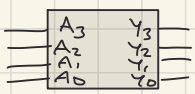
connects

doesn't connect

Multiple-Output Circuits

Example: Priority Circuits

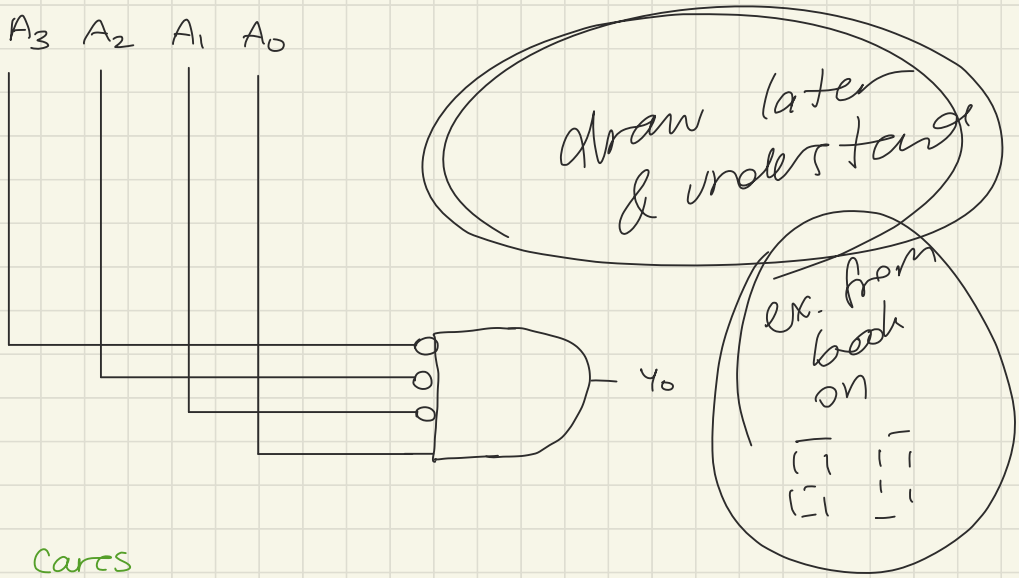
Output is 1 according to the most prioritized input



priority circuit

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

priority circuit hardware



Don't Cares

The circuit above was hard to write down
We use Dont Care to make the truth table more compact

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

don't care about the others

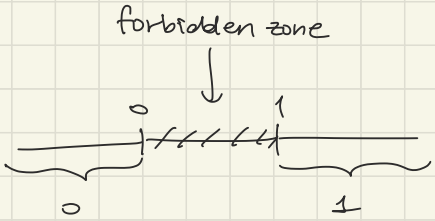
Truth table easier to read → easier to build
→ easier to compute

X and Z

Circuit Voltage

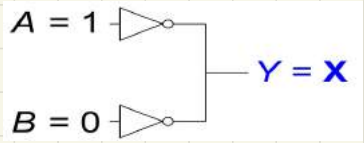
0 and 1 are discrete values

ex: 0V is 0
ex: 1.2V is 1



Contention : X

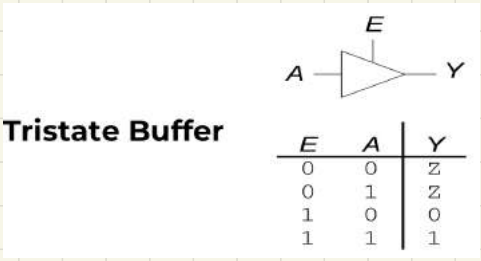
— Circuit tries to decide between 1 and 0
— This is a bug, cannot happen in real life
— Happens if wrong circuit



* X is used for contention and "don't care"
look at context to tell them apart

Floating : Z

— Neither 0 nor 1.
— Output can be 0, 1 or anything inbetween

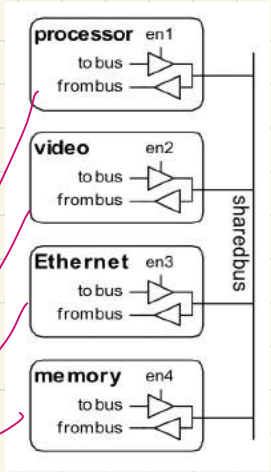


* A voltmeter will NOT indicate whether or not a node is floating

Tristate Bus

Floating nodes used in tristate busses

Used in cases with multiple components

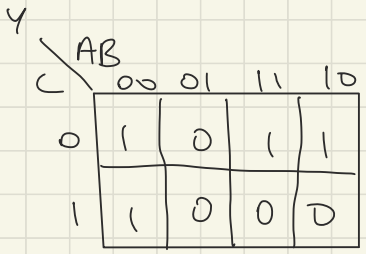


many different drivers but one is active at once

K-Maps (Karnaugh maps)

Minimized boolean expressions - represents a truth table through 2D structure

Can be done for SOP and POS ✓



Rules

#1 - Only change one bit

ex:

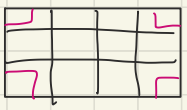
00 01 10 11 ✗

00 01 11 10 ✓

#2 - Circle only powers of 2

ex:

#3 - Wrapping corners is possible

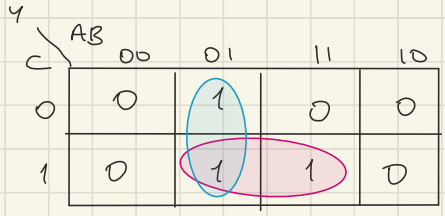


ex:

Draw the 3-input K-map for the following

Truth Table

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	**1**
0	1	1	**1**
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	**1**



$\bar{A}BC + \bar{A}B\bar{C} + \bar{A}BC + ABC$

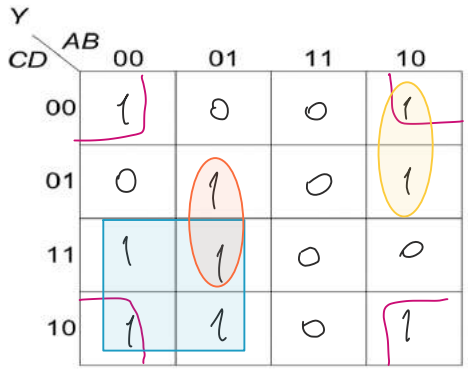
$= \bar{A}B + BC$

ex:

Draw the 4-input K-map for the following

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

1st 2nd 4th 3rd



* ones that dont change are the result

$\bar{A}\bar{B}CD + \bar{A}\bar{B}C\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D}$

$\bar{A}C(\cdots)$

$= \bar{A}C$

$\bar{A}\bar{B}CD + \bar{A}\bar{B}C\bar{D}$

$= A\bar{B}C(\cdots)$

$= A\bar{B}\bar{C}$

$\bar{A}BCD + \bar{A}BC\bar{D}$

$= \bar{A}BD(\cdots)$

$= \bar{A}BD$

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

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

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

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

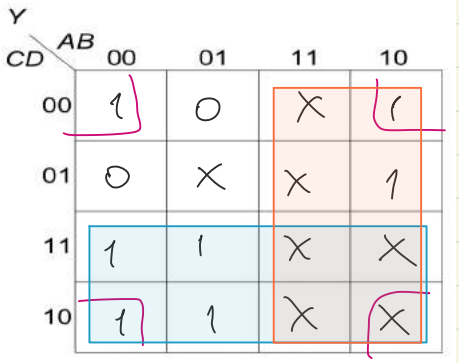
$= \bar{B}\bar{D}$

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

K-Maps With Don't Cares

ex: Draw the 4-input K-map with don't cares

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



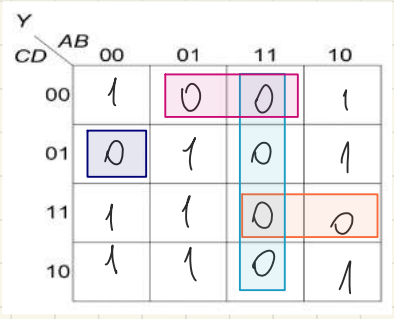
$\bar{B}\bar{D} + C + A$

★ Point of information ★ duality ★

So far we've done SOP, POS is possible too.

0s instead of 1s, each square maxterm } POS form
each covering represents sum

ex:

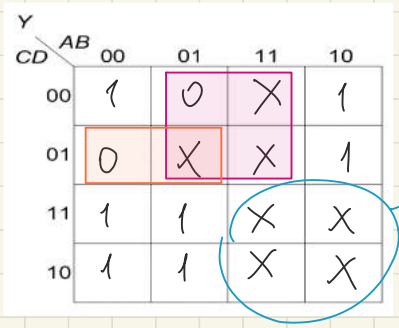


pos form

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

ex:

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



no squares with all don't cares

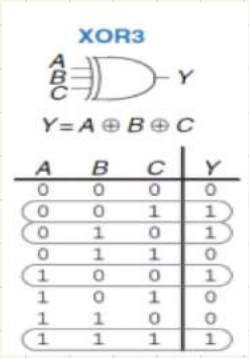
the result would be the same anyway

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

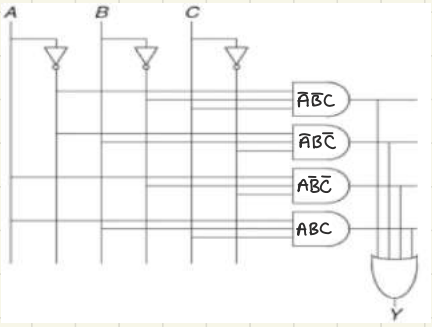
Quine - Mc Cluskey Algorithm

- Method of prime implicants
- Used for minimization of Boolean functions, functionally identical to K-maps but tabular form → + efficiency

Multilevel Combinational Logic



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



Questions

1- How many AND gates do you need to build an 8-input XOR, using an AND/OR 2-levels network?

256 out { 128 outputs → 1 ⇝ 128 AND gates
128 outputs → 0

2- How many inputs do the AND gates have ?

8 inputs overall → 1 for each → answer : 8

3- How many inputs does the OR gate have ?

128 → one for each AND

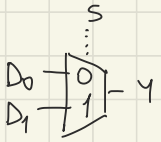
Combinational Building Blocks :

1. Multiplexer (Data Selector)

* Output assigned to only one of the inputs
* $\log_2 N$ bit select inputs (control signal)

ex: if D_0, D_1, D_2, D_3 then S_0, S_1

$$Y = \bar{S}D_0 + SD_1$$



2:1 MUX

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

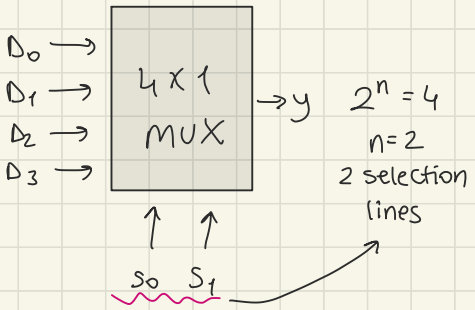
S	Y
0	D_0
1	D_1

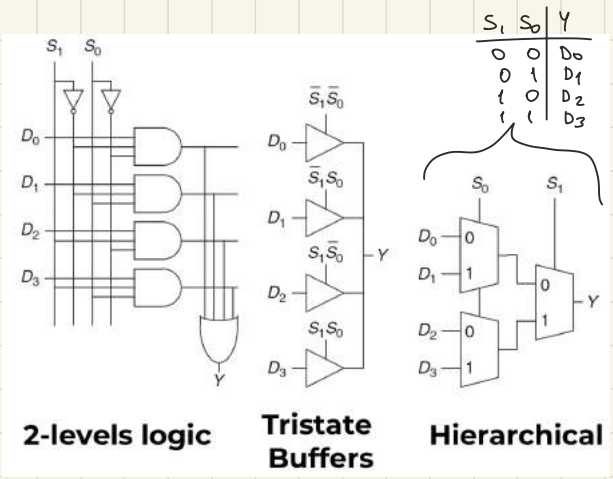
POI : some examples

S_0	S_1			
0	0	→	00_2 → = 0 →	I_0
1	1	→	11_2 → = 3 →	I_3
1	0	→	10_2 → = 2 →	I_2

multiple S's come together to correspond to one input

ex: 4 to 1 MUX





Logic: when S is zero, input zero (D_0)'ll be sent to the outp

Questions

1- How many AND gates do you need to build an 8-input MUX, using an AND/OR 2-levels circuit?

8 → one for each input

2- How many inputs do the AND gates have?

4 → the input itself + 3 selection lines

formula: $\log_2 N + 1 = \log_2 8 + 1 = 3 + 1 = 4$

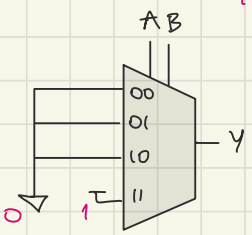
3- How many inputs does the OR gate have?

8 → one for each AND gate

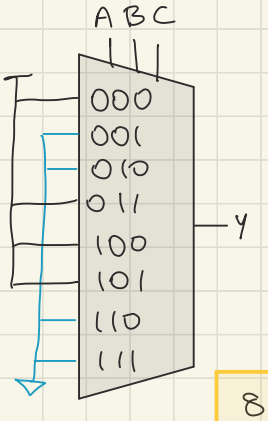
Using MUX as a lookup table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$



A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



8X1 MUX

$Y = \bar{A}\bar{B}\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C$

$= \bar{B}\bar{C} + \bar{A}BC + A\bar{B}C$

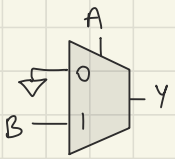
$= \bar{B}(\bar{C} + AC) + \bar{A}BC \quad = \quad \bar{A}BC + \bar{B}\bar{C} + A\bar{B}$

Logic using MUX

ex:

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	Y
0	0
1	B

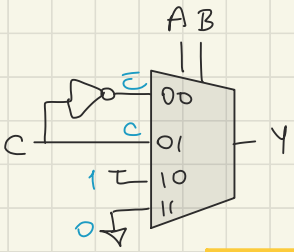


2X1 MUX

ex:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

A	B	Y
0	0	$\bar{C}$
0	1	C
1	0	1
1	1	0



4x1 MUX

Shannon Expansion Theorem

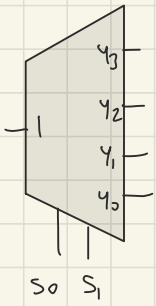
Used to expand any Boolean equation (f)

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$$

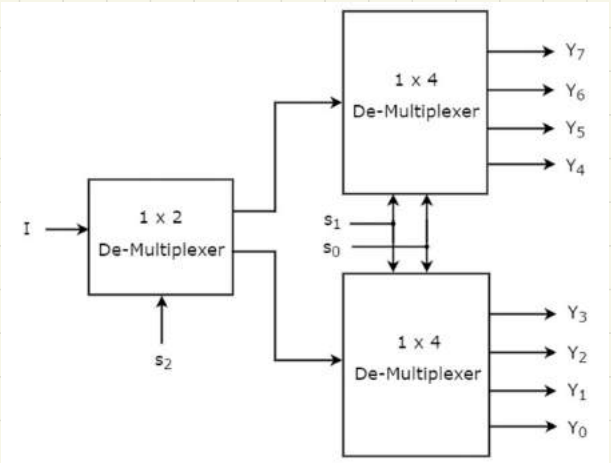
Demultiplexer (DEMUX)

Input goes on one of the outputs

N outputs → $\log_2 N$ bits



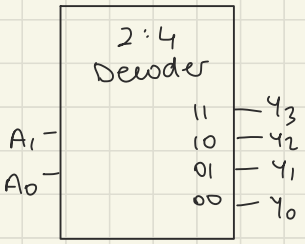
ex: 1 x 8 DEMUX



2. Decoder

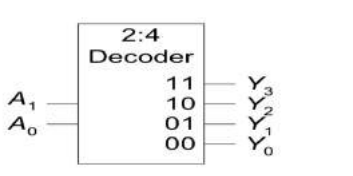
N inputs, 2^N outputs

One-hot outputs → only one output is HIGH at once

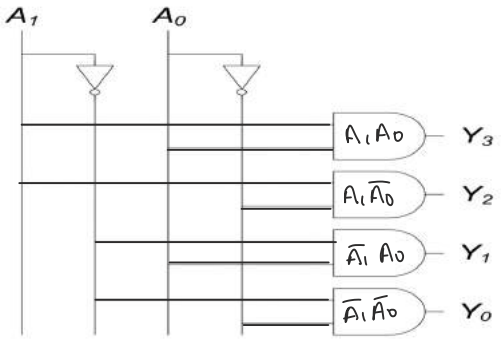


A_1 A_0	Y_3	Y_2	Y_1	Y_0
0 0	0	0	0	1
0 1	0	0	1	0
1 0	0	1	0	0
1 1	1	0	0	0

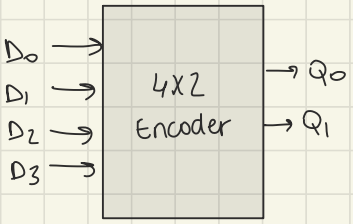
ex: Decoder implementation on multilevel logic



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



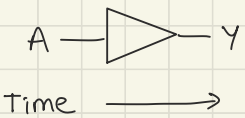
Encoders

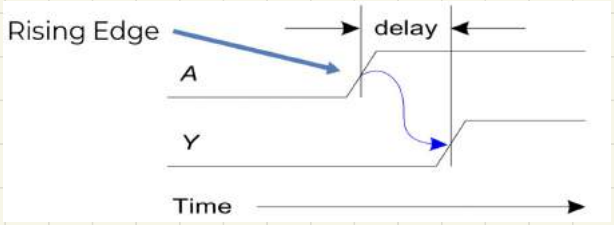


$D_3 D_2 D_1 D_0$	$Q_1 Q_0$
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
0 0 0 0	x x

Timing

Delay: time between input change and output changing





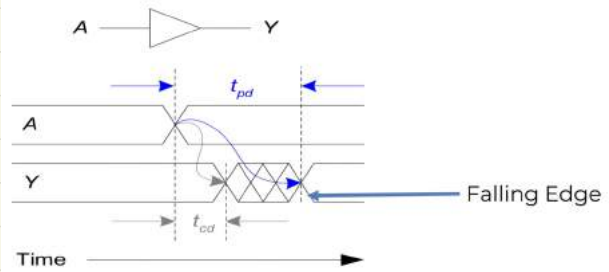
The moment where A changes into Y is called "delay"

The rising edge is the middle of the delay

Propagation & Contamination Delay

Propagation Delay: t_{pd} = max delay from input change to when outputs REACH final value

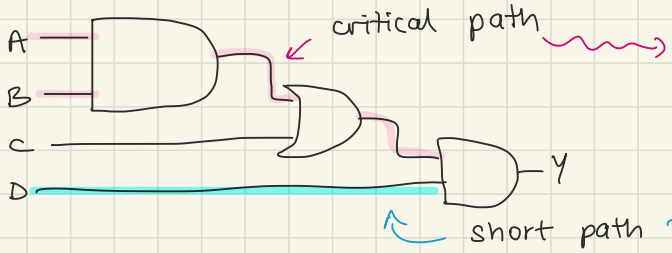
Contamination Delay: t_{cd} = min delay from input change to when an output STARTS to change



Delay is caused by physical limitations

t_{pd} may not always be equal to t_{cd}

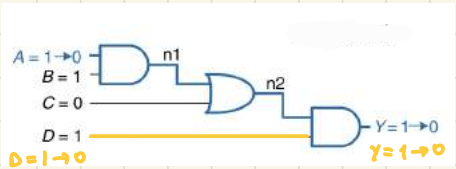
Critical (Long) & Short Paths

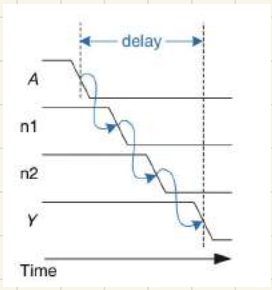


$t_{pd} = 2t_{pd\_AND} + t_{pd\_OR}$

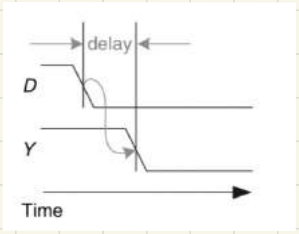
$t_{cd} = t_{cd\_AND}$

ex: Page 90 from book





Critical path



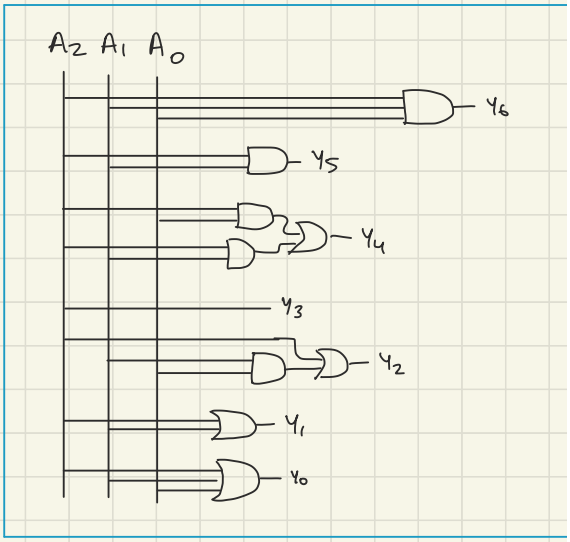
Short path

ex:

Exercise 2.38 An *M*-bit *thermometer code* for the number *k* consists of *k* 1's in the least significant bit positions and $M - k$ 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has *N* inputs and 2^N–1 outputs. It produces a 2^N–1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output, and sketch a schematic.

A_2	A_1	A_0	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	1
0	1	1	0	0	0	0	1	1	1
1	0	0	0	0	0	1	1	1	1
1	0	1	0	0	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

"thermometer" bc increasing



note: can be done by K-maps

$Y_0 = A_0 + A_1 + A_2$

$Y_1 = A_2 + A_1$

$Y_2 = A_2 + A_1 \cdot A_0$

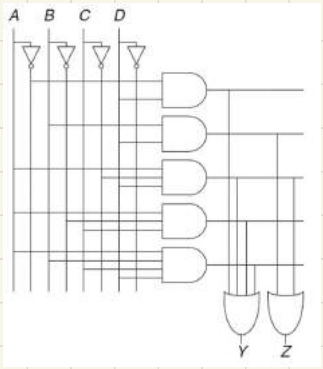
$Y_3 = A_2$

$Y_4 = A_2 A_1 + A_2 A_0$

$Y_5 = A_2 A_1$

$Y_6 = A_2 A_1 A_0$

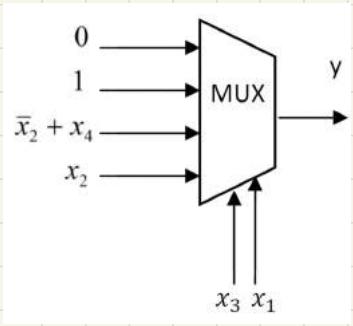
ex: 2.24- Write the Boolean eq. for the following circuit (no need to minimize it)



$$Y = \bar{A}D + A\bar{C}D + AB\bar{C} + ABCD$$

$$Z = BD + A\bar{C}D$$

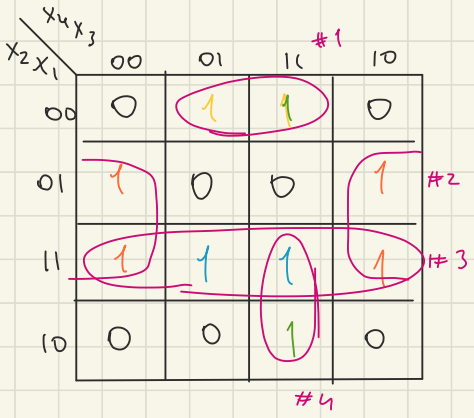
ex: Convert MUX to SOP



x_3	x_1	Y
0	0	0
0	1	1
1	0	$\bar{x}_2 + x_4$
1	1	x_2

$$Y = 0(\bar{x}_3\bar{x}_1) + 1(\bar{x}_3 x_1) + (\bar{x}_2 + x_4)(x_3\bar{x}_1) + x_2(x_3 x_1)$$

$$= \bar{x}_3 x_1 + x_3\bar{x}_2\bar{x}_1 + x_4 x_3\bar{x}_1 + x_3 x_2 x_1$$



#1 #2 #3 #4

$$x_3\bar{x}_2\bar{x}_1 + \bar{x}_3 x_1 + x_2 x_1 + x_4 x_3 x_2$$

ex: 2.21 in the book example for Ben's point

Y	BC 00	01	11	10
A 0	0	1	1	0
1	0	1	1	0

$Y = C$

Chapter 3

State: info about a circuit

Latches & Flip Flops

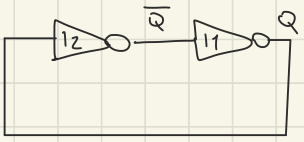
Latches & flip-flops are state elements that store 1 bit.

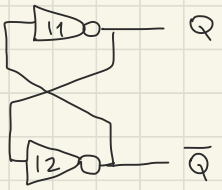
→ Has short term memory

→ Present output depends on present input as well as past output(s)

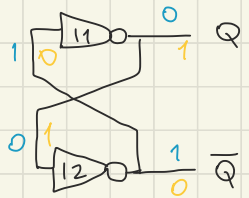
- Bistable Circuit
- SR Latch
- D Latch
- D flip-flop

1 Bistable Circuit is the foundation for other state elements





Consider the two cases:



→ $Q = 0 \Rightarrow$ then $\overline{Q} = 1$ (consistent)

→ $Q = 1 \Rightarrow$ then $\overline{Q} = 0$ (consistent)

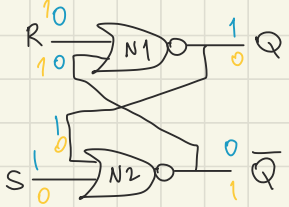
* There are **no inputs** to control the state
* N variables → $\log_2 N$ bits of info. stored

2 SR (Set / Reset) Latch nor gate!

The latch is controlled by pulses only

> set S to 1 if you want to SET Q to 1, reset 1 if u want Q=0

Consider all 4 cases:



Set the output

$S = 1, R = 0 \longrightarrow$ then $Q = 1$ and $\overline{Q} = 0$

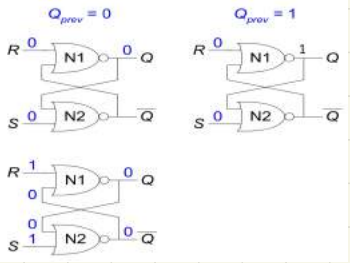
$S = 0, R = 1 \longrightarrow$ then $Q = 0$ and $\overline{Q} = 1$

Reset the output

$S = 0, R = 0$

$S = 1, R = 1 \longrightarrow$

- **S = 0, R = 0:**
  then $Q = Q_{prev}$
  Memory! output depends on prev. value
- **S = 1, R = 1:**
  then $Q = 0, \overline{Q} = 0$
  Invalid state



Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

unreachable (invalid) state

watch the video!

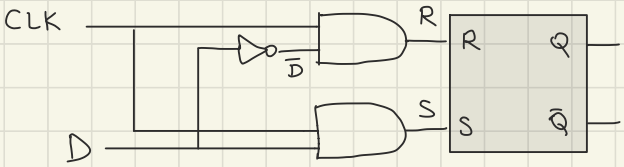
3 D Latch

CLK : when the out. changes
D : what the out. changes to

CLK = 1 } transparent
D → Q

CLK = 0 } opaque
Qprev

D Latch Circuit



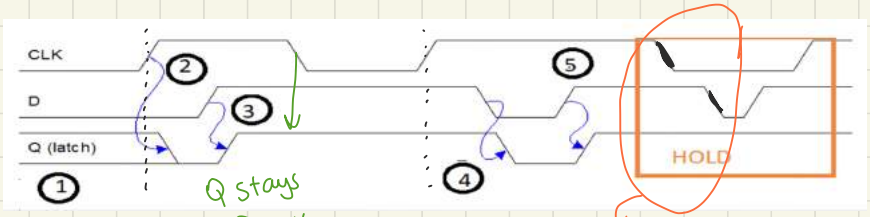
CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prev}	$\bar{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

D Latch Waveform

② transparent D = Q

③ CLK = 1, D 0→1, D = Q

⑤ CLK = 1, D 1→0, D = Q



① (power just turned on) Q is unknown

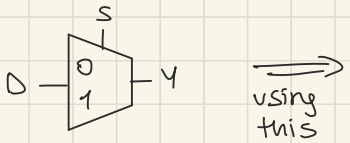
Q stays at Qprev when CLK=0

④ CLK = 1, D 1→0, D = Q

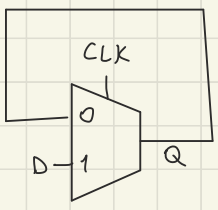
HOLD : CLK = 0, D 1→0, D ≠ Q

Q doesn't follow

ex: Draw a D Latch starting from a 2:1 MUX



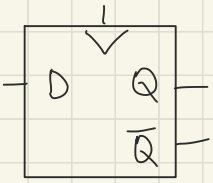
using this



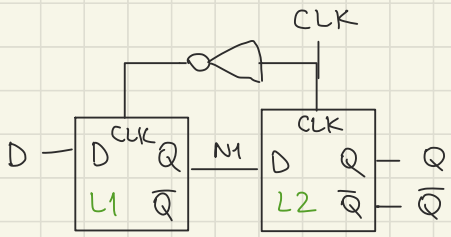
4 D Flip-Flop

D=Q only on rising edge of CLK

↳ edge triggered



D Flip-Flop Internal Circuit



when CLK = 0

L1 CLK = 1
D → N1
transparent

L2 CLK = 0
Q prev
opaque

when CLK = 1

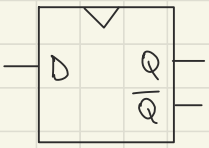
L1 CLK = 0
Q prev
opaque

L2 CLK = 1
N1 → Q
transparent

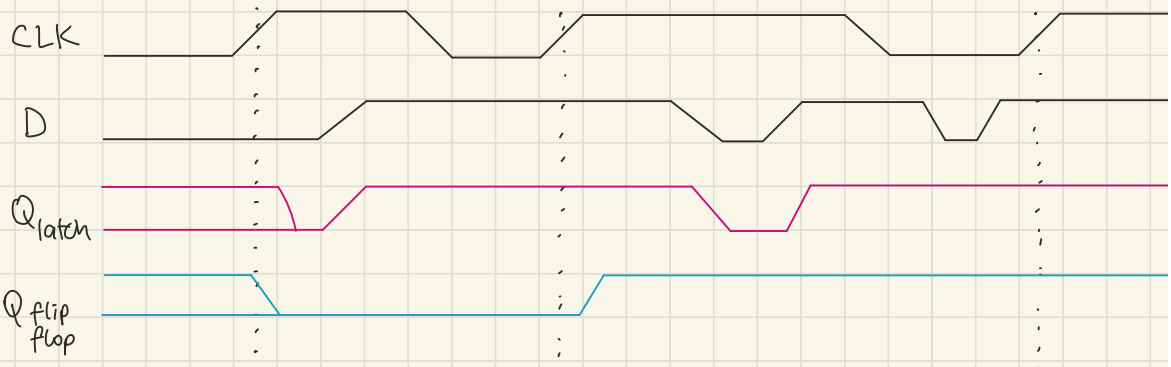
ex:

D Latch vs D Flip-Flop

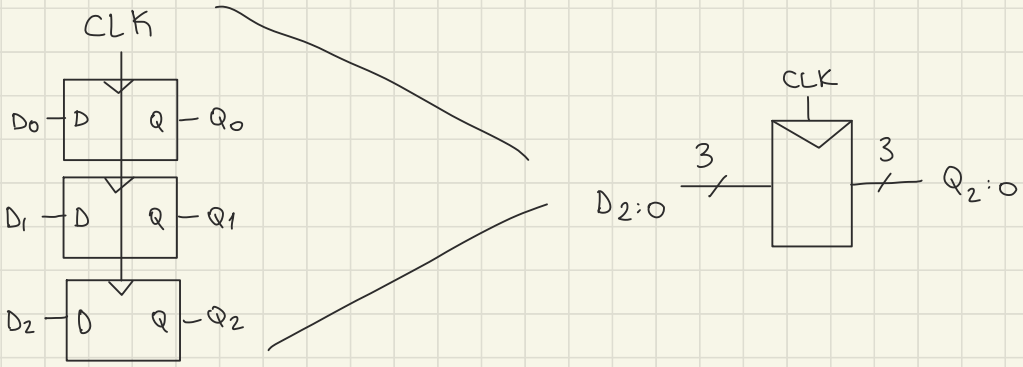
D = Q as long as CLK = 1 else Q prev



D=Q only on rising edge



5 Registers : Multi-bit Flip Flops

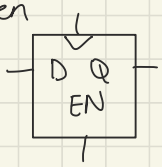


Flip Flops

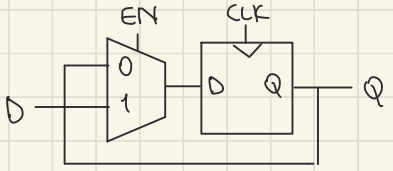
1 Enabled Flip-Flop

the ENable controls when Data is stored

EN=1 : D passes to Q
EN=0 : Previous state

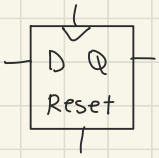


Internal circuit



2 Resettable Flip-Flops

R=1 : Q forced to 0
R=0 : normal D flip flop

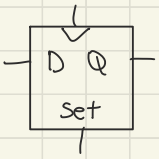


2 types

2.1 Synchronous ↓ resets at clock edge only

2.2 Asynchronous ↓ resets immediately when R=1

(3) Settable Flip-Flops

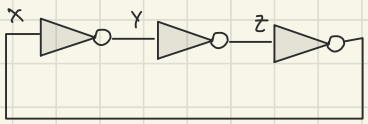


S=1 : Q forced to 1
S=0 : normal D flip flop

Synchronous Logic Design

Sequential circuit : all circuits that aren't combinational

Some problematic circuits : (1) Ring oscilator

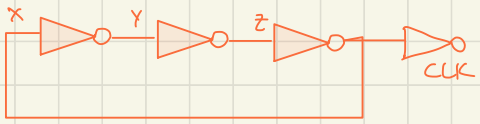


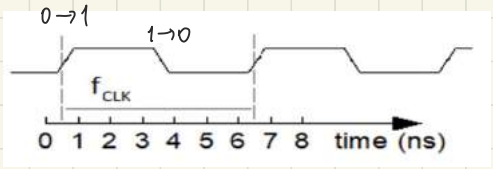
⇒ no inputs, 3 outputs, tpd of each inverter 1ns

Astable (unstable) circuit (x=0 → y=1, z=0 → x=1) oscillates

cyclic path: output fed back into input

ex: Assuming $t_{cd} = t_{pd} = 1$: calculate the frequency of CLK

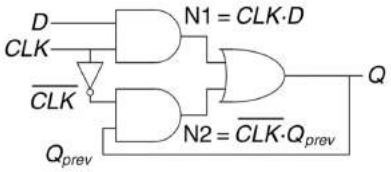


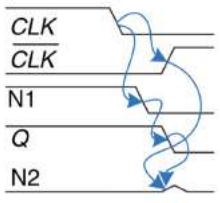


3ns for X, Y, Z

(2) Race Conditions

$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$





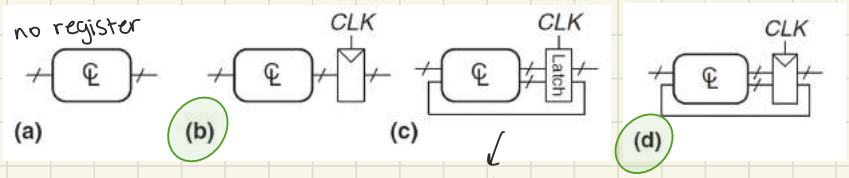
Synchronous Sequential Circuits

- Breaks cyclic paths by inserting registers
  - o Combirational circuit + registers
- Registers contain state of the system
- System synced to CLK
- If CLK is slow, inputs to all registers settle before the next CLK edge and all races are eliminated

Rules of synchronous sequential circuit composition:

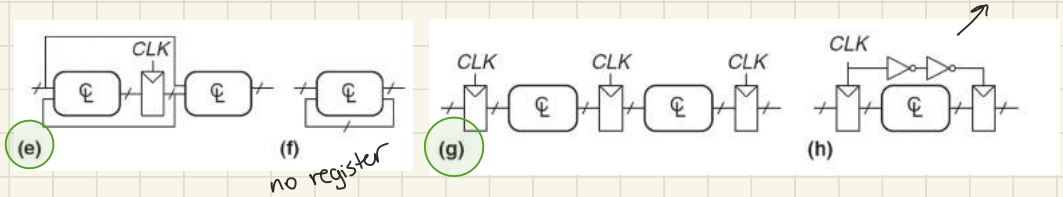
Elements either (at least 1) register or a comb. circuit
All registers receive the same clock signal
Each cyclic path contains atleast 1 register

ex: Which of the following are synchronous sequential circuits ?



every cyclic path atleast 1 register

all registers must receive same clock



no register

Finite State Machines

Consists of :

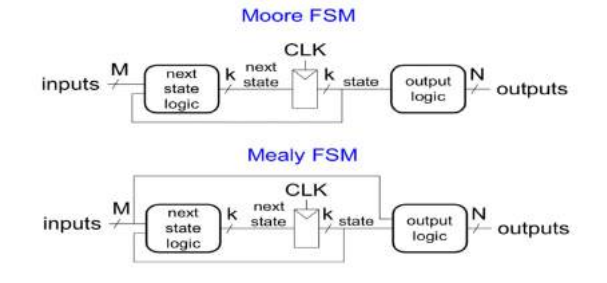
- State register : stores curr state & loads next state at CLK edge
- Combinational logic : computes next state and computes the outputs

Next state determined by current state and inputs

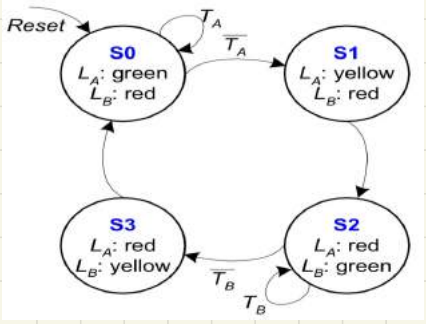
2 types differ in output logic :

Moore FSM : out depends on current state

Mealy FSM : out depends on current state and inputs



Moore FSM



$S_{0:3}$ states

outputs labeled in each state

green : 00
yellow : 01
red : 10
} we encode using binary

FSM Output Table

Current		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$L_{A1} = S_1$ $\quad L_{B1} = \overline{S_1}$

$L_{A0} = \overline{S_1} S_0$ $\quad L_{B0} = S_1 S_0$

State Transition Table and Encoded State Transition Table

Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

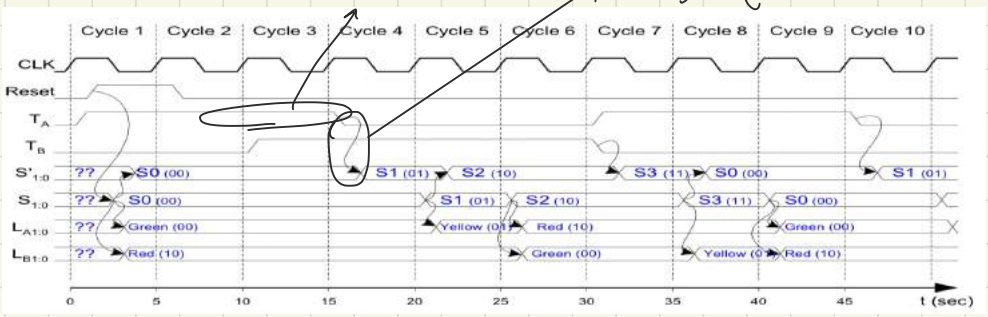
$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\,\overline{S_0}\,\overline{T_B}$$

Moore FSM Timing Diagram

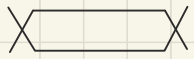
there's traffic on road A

as soon as TA=0 lights start changing (with delay ofc)



Every 5 seconds CLK 0→1

State only changes on rising edge of CLK (bc. D flipflops)

 ← this is bc multiple outputs

Finite State Machine (FSM) Initialization

→ The initial state of a FSM is not defined when switched on it can start from any of the 2^N (# of bits) possible states

→ To solve this problem → Reset signal exists!

FSM Unreachable States

→ In a 3 state FSM ($S_0 = 00$, $S_1 = 01$, $S_2 = 10$) 11 is unreached

→ The unused state still contribute in output, but are never reached

FSM Encoding

Binary Encoding

i.e. for four states 00, 01, 10, 11

One-hot encoding

One state bit per state, one bit HIGH at once
i.e. for four states 0001, 0010, 0100, 1000
Requires more flip flops however often next state and output logic are simpler

Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

State	Encoding
S0	0001
S1	0010
S2	0100
S3	1000

Current State				Inputs		Next State			
S_3	S_2	S_1	S_0	T_A	T_B	S'_3	S'_2	S'_1	S'_0
0	0	0	1	0	X	0	0	1	0
0	0	0	1	1	X	0	0	0	1
0	0	1	0	X	X	0	1	0	0
0	1	0	0	X	0	1	0	0	0
0	1	0	0	X	1	0	1	0	0
1	0	0	0	X	X	0	0	0	1

$$S'_3 = S_2\overline{T_B} \qquad S'_1 = S_0\overline{T_A}$$
$$S'_2 = S_1 + S_2 T_B \qquad S'_0 = S_0 T_A + S_3$$

ex.

Moore vs. Mealy FSM – The snail

If the snail goes over 01 it smiles

out 1 = smile 0 = no smile

states

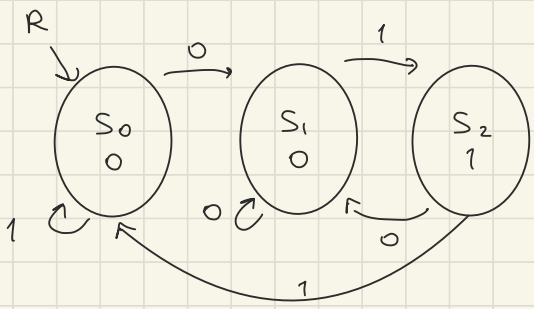
00	0
01	1
10	0
11	0

1 is seen, previous was 0 = 1 S_2

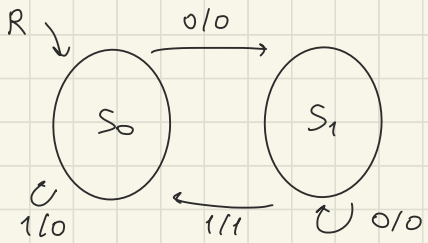
0 is read = 0 S_1

1 is seen, previous was 1 = 0 S_0

Moore



Mealy



Moore State transition table

S_0	00
S_1	01
S_2	10

encodings

S_1 S_0	A	S_1' S_0'
0 0	0	0 1
0 0	1	0 0
0 1	0	0 1
0 1	1	1 0
1 0	0	0 1
1 0	1	0 0

S_0 saw 0 go to S_1

$S_1' = S_0 A$

$S_0' = \overline{A}$

Moore Output

S_1 S_0	Y
0 0	0
0 1	0
1 0	1
1 1	X

$Y = S_1$

Mealy State transition and Output table

S_0	0
S_1	1

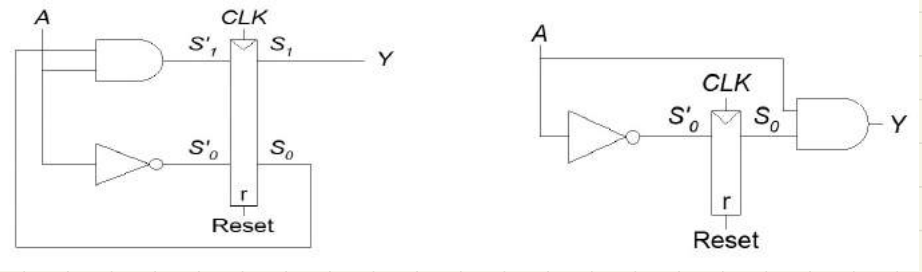
encodings

current state S_0	input A	next s. S'_0	output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

$S'_0 = \overline{A}$

$Y = S_0 A$

Moore vs Mealy Schematic



ex. done

Moore vs Mealy Comparison

Moore

- more states (typically)
- safer bc. output changes at edge

Mealy

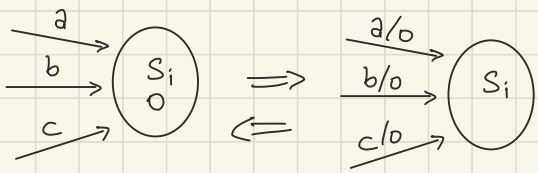
- fewer states (typically)
- reacts faster, doesn't wait for CLK
- asynchronous outputs can be dangerous

Synchronous Mealy: avoids glitchy outputs, safer, obeys CLK

Glitches

Glitches occur when a single input causes problems with multiple outputs. It is a timing error, if waited enough, the output settles to the right answer

Moore ⇄ Mealy Conversion



FSM Design Procedure

1. Identify ins & outs
2. Sketch state transition diagram
3. Write state transition table
4. Select state encodings
5. for moore:
   a. Write state transition table
   b. Write output table

   for mealy:
   Write a combined state tr. and out. table
6. Select encoding
7. Write equations (boolean for next st. & out.)
8. Sketch!

See Lecture 14 for ex!

Deriving a FSM from a schematic

Almost the reverse process of FSM design

1. Examine circuit → inputs ? → outputs? → state bits ?
2. Write next state and output equations
   ↳ create tables
3. Reduce and eliminate unreachables
4. Assign a name to each valid state
5. Rewrite next state and output tables with state names
6. Draw state transition
7. State what the FSM does

ex:

Derive the following FSM

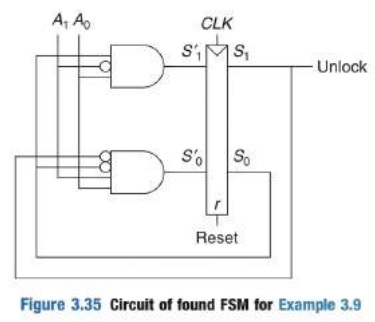


Figure 3.35 Circuit of found FSM for Example 3.9

→ Moore (doesn't depend on inp.s)

$U = S_1$ $\quad S_1' = S_0 \bar{A}_1 A_0$ $\quad S_0' = \bar{S}_1 \bar{S}_0 A_1 A_0$

2^4 combinations

next state table →

S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
		,			
		1			
		1			

→ output table

S_1	S_0	U
0	0	0
0	1	0
1	0	1
1	1	1

Table 3.17 Next state table derived from circuit in Figure 3.35

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S_1'	Next State S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

group all

unreachable bc. next state is never 1 1

Table 3.18 Output table derived from circuit in Figure 3.35

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1
1	1	1

reduce table

Table 3.19 Reduced next state table

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S_1'	Next State S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Table 3.20 Reduced output table

	Current State S_1	Current State S_0	Output *Unlock*
S_0	0	0	0
S_1	0	1	0
S_2	1	0	1

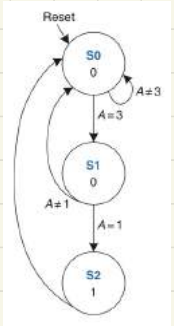
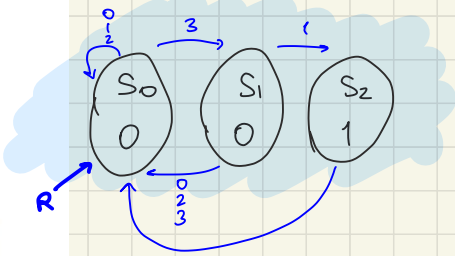


Table 3.21 Symbolic next state table

Current State S		Input A		Next State S'
S0	00	0	00	S0
S0		1	01	S0
S0		2	10	S0
S0		3	11	S1
S1	01	0		S0
S1		1		S2
S1		2		S0
S1		3		S0
S2	10	X		S0

Table 3.22 Symbolic output table

Current State S	Output *Unlock*
S0	0
S1	0
S2	1



Timing

→ Flip flop samples D at clock edge

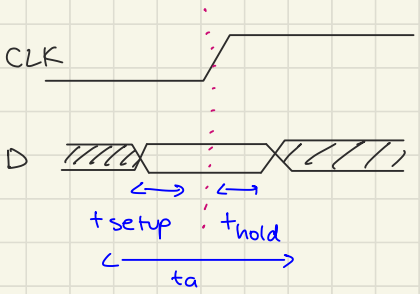
→ T_c is the clock period (from rising edge to another)

→ $f = \frac{1}{T_c}$ measured in Hertz 1Hz = 1 cycle per second

→ 1GHz = 1000MHz
1MHz = 1000KHz
1KHz = 1000Hz

→ D must be stable when sampled

Input Timing Constraints



Setup time: t_{setup} = data must be stable for a period of time before rising edge

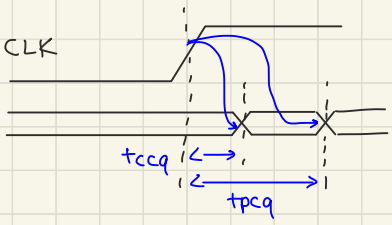
Hold time: t_{hold} = time after clock edge must be stable

Apperture time: $t_a = t_{setup} + t_{hold}$

Output Timing Constraints

Propagation delay: t_{pcq} = time after CLK edge out is guaranteed stable

Contamination delay: t_{ccq} = time after edge out might not be stable

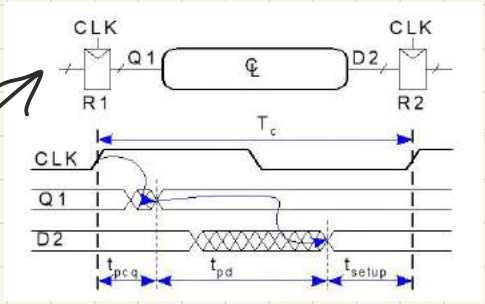


The Dynamic Discipline

The dynamic discipline (3.5.1) states that:
The inps. of a synchronous sequential circuit must be stable during the (setup and hold) apperture time around CLK edge

System Timing → Setup Time Constraint

The delay between registers has a min and a max delay, depending on the circuit elements



setup time constraint

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

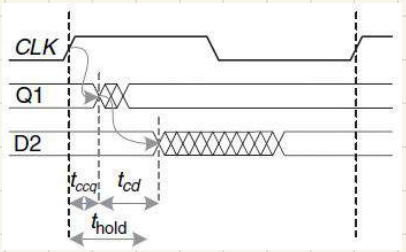
$(t_{pcq} + t_{setup})$ → depend on the flipflop

System Timing → Hold Time Constraint

longer path for setup time
shortest path for hold time

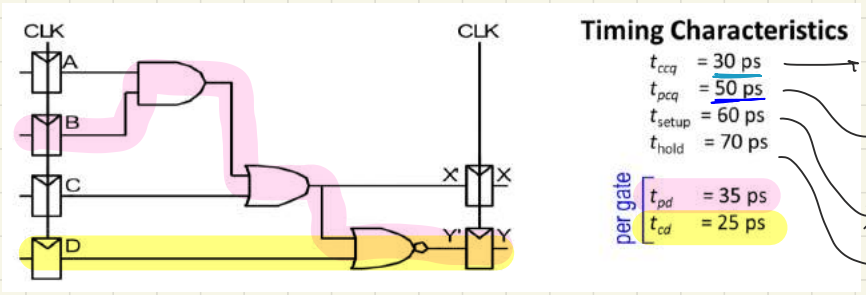
* buffers can be added to meet hold criteria

$$t_{hold} < t_{ccq} + t_{cd}$$



ex:

Timing Analysis



$t_{pd} = 3 \times 35 = 105$ ps
$t_{cd} = 1 \times 25 = 25$ ps

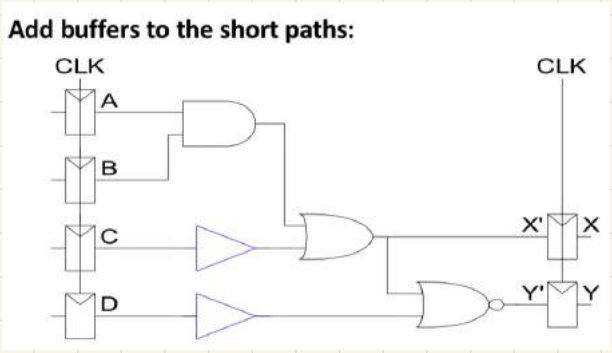
Setup time constraint

$T_c \geq 50 + 105 + 25 = 215$ ps

$f_c = \frac{1}{T_c} = 4.65$ GHz

hold time constraint $(30+25)$ ps > 70ps ✗

* add a BUF gate for extra delay



$t_{pd} = 3 \times 35 = 105$ ps
$t_{cd} = 2 \times 25 = 50$ ps

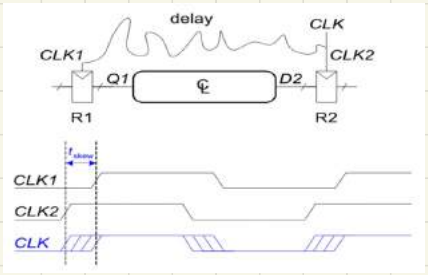
T_c didnt change however

↙

hold time constraint $(30+50)$ ps > 70ps ✓

Clock Skew

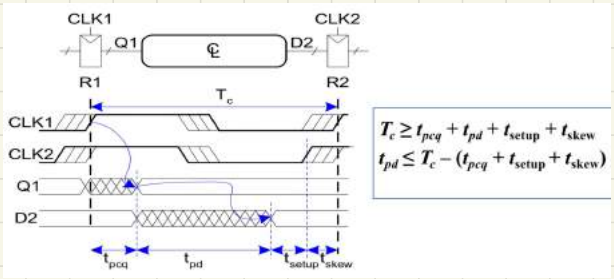
the clock doesnt arrive at all registers at the same time



Skew: difference between clock edges

Perform worst case analysis to check for dynamic discipline violations

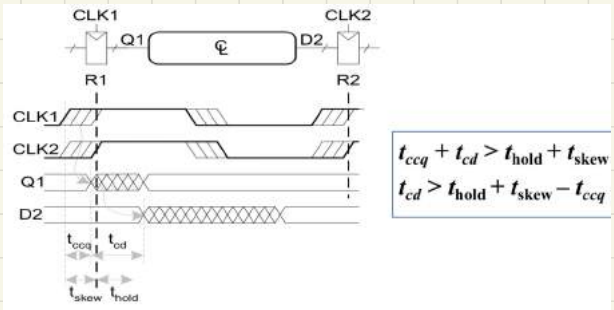
Setup Time Constraint with Skew



Worst case:

CLK2 earlier than CLK1

Hold Time Constraint with Skew



Worst case:

CLK2 later than CLK1

Parallelism

Speed → latency and throughput of information

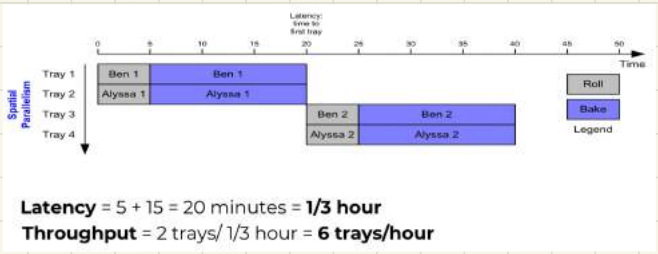
Token: group of inputs → group of outputs

Latency: time for one token to pass

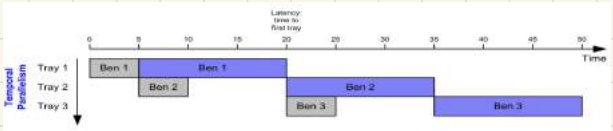
Throughput: number of tokens per unit time
↳ can be improved by processing more tokens at the same time (parallelism)

2 types of parallelism:

Spatial parallelism:
multiple copies of hardware perform multiple tasks at once



Latency = 5 + 15 = 20 minutes = **1/3 hour**
Throughput = 2 trays/ 1/3 hour = **6 trays/hour**



Latency = 5 + (10) +15 = 30 minutes = **1/2 hour**
Throughput = 1 trays/ 1/4 hour = **4 trays/hour**

Using both techniques, the throughput would be **8 trays/hour**

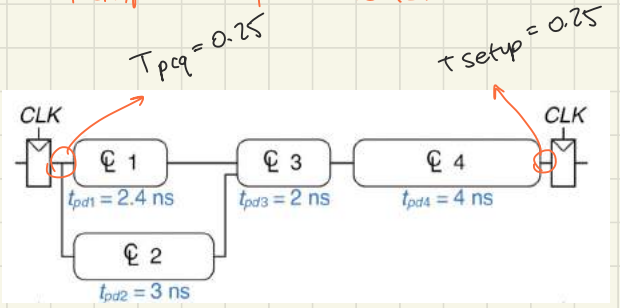
Temporal parallelism:
tasks broken into multiple stages, different stages work at the same time (pipelining)

Generalization

- Latency = L $\Rightarrow$ Throughput = $\frac{1}{L}$
- Spatial parallelism with N copies $\Rightarrow$ throughput = $\frac{N}{L}$
- Temporal parallelism
  - ↳ ideal case N stages equal length $\Rightarrow$ throughput = $\frac{N}{L}$
  - ↳ in practice this is rarely the case. longest stage with latency L_1 $\Rightarrow$ throughput = $\frac{1}{L_1}$

ex:

Temporal parallelism example



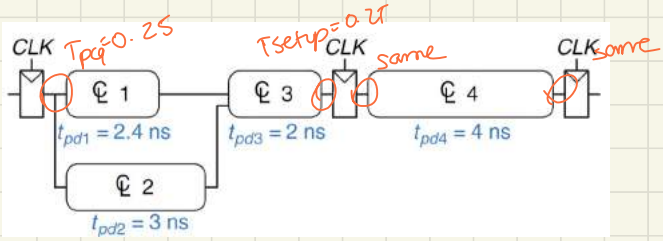
$T_c = 3+2+4+0.25+0.25 = 9.5\,ns$

Latency = 9.5 ns

Throughput = $\frac{1}{9.5}$ ns

= 105 MHz

ex:



$3+2+0.25+0.25 = 5.5$

$4+0.25+0.25 = 4.5$

Latency = $5.5 \cdot 2 = 11$

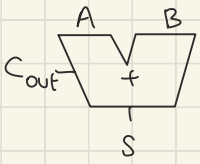
Throughput = $\frac{1}{5.5}$

Chapter 5

Things we've seen so far are digital building blocks (gates, multiplexers, decoders, logic arrays etc.)

These building blocks are used to build a microprocessor.

1 Bit Adder (Half Adder)

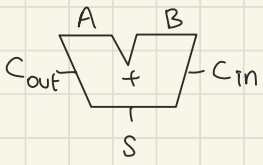


A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

1 Bit Adder (Full Adder)



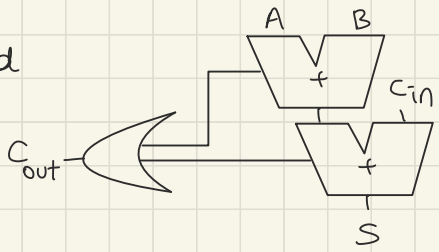
$C_{out} = AB + AC_{in} + BC_{in}$

$S = A \oplus B \oplus C_{in}$

Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adders From Half Adders

The full adder can be formed of 2 half adders



Multibit Adders (CPAs)

3 types:

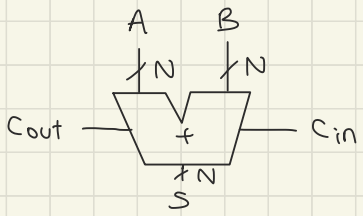
1- Ripple-carry
slow

2- Carry-lookahead
fast

3- Prefix
faster

faster for large adders but require more hardware

CPA general symbol

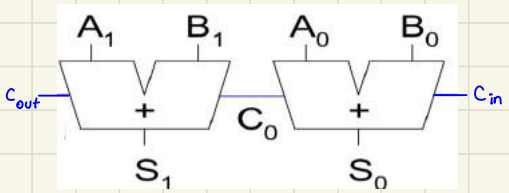


#1 Ripple-Carry Adder (CLA)

Chains 1-bit full adders

Carry ripples through the chain

```
  010
 1010
+0011
------
 1101
```

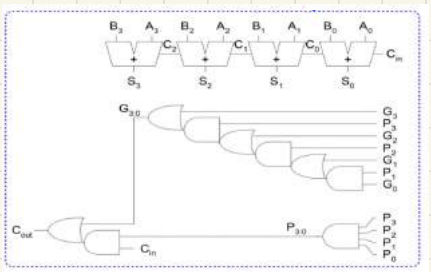


* Slow because carry signal goes through every bit

#2- Carry Lookahead Adder

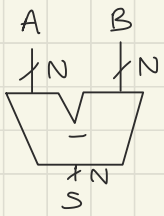
solves this problem by dividing the adder in blocks (ex: 32 bit adder divided into 4-bit blocks)

Looks ahead accross blocks instead of waiting to ripple accross all



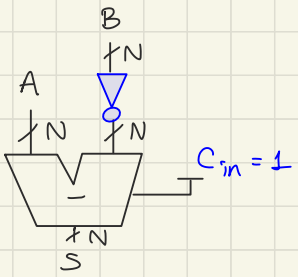
Subtracter

Symbol



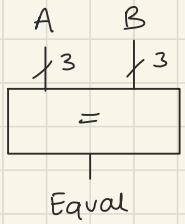
Implementation

$A + (-B)$

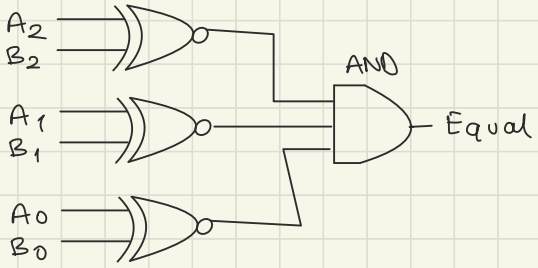


Comparator : Equality

Symbol



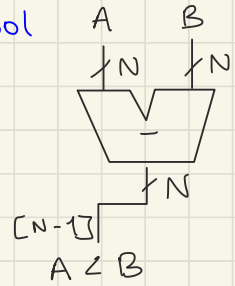
1 if A = B , else 0



Implementation

Comparator : Less Than

Symbol



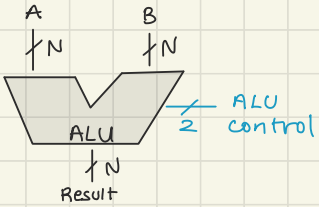
If result is negative (sign bit = 1)

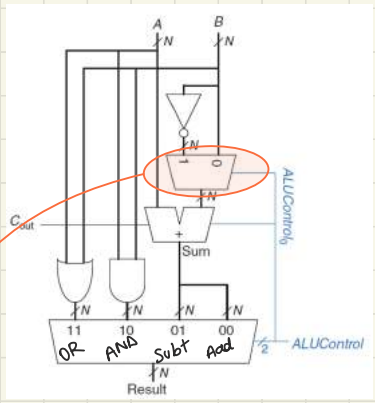
$A - B < 0 \quad \therefore \quad A < B$

* Does not work with overflows

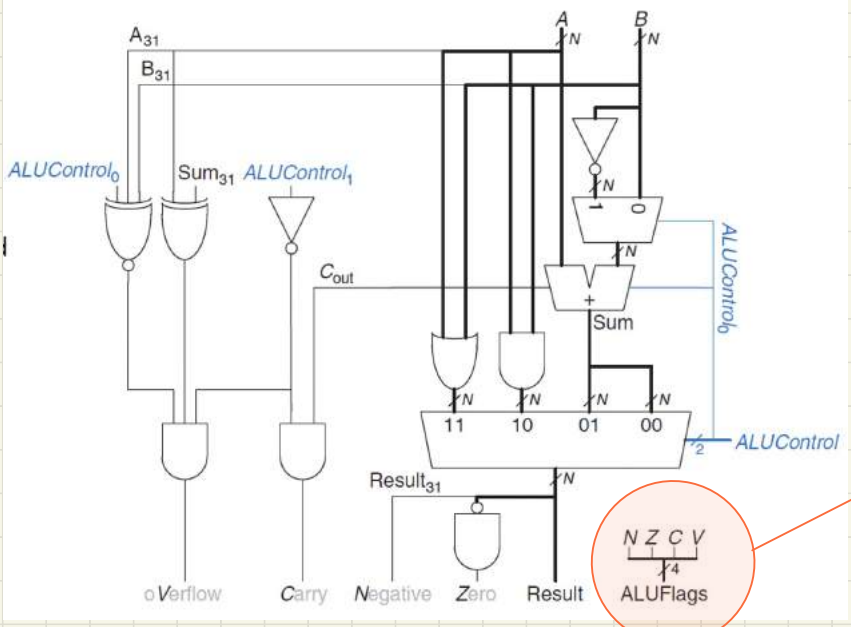
Arithmetic Circuits – Arithmetic / Logical Unit (ALU)

Performs multiple operations
(AND, OR, addition, subtraction)





useful for Subt. since
01 ⇒ −B + A



extra outputs "flags"

Shifters and Rotators

Logical Shifters

Shifts value to L/R fills spaces with zeros

11001 >> 2 = 00110
11001 << 2 = 00100

Arithmetic Shifting

Same as logical shift but right shift extended by sig. bit.

11001 >>> 2 = 11110
11001 <<< 2 = 00100

Rotator

Rotates bits in a circle

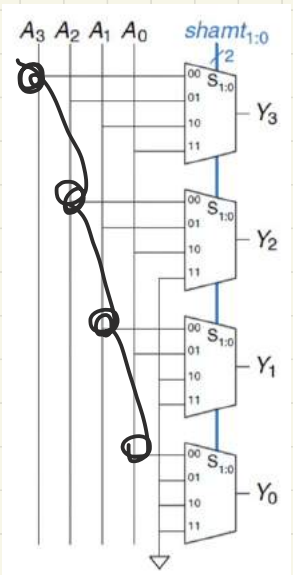
11001 ROR 2 = 01011

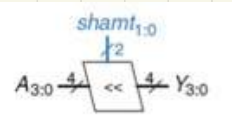
11001 ROL 2 = 01011

Remember that shifters act as multipliers and dividers!

$A << N = A \times 2^N$
$A >>> N = A / 2^N$

Shifter Design

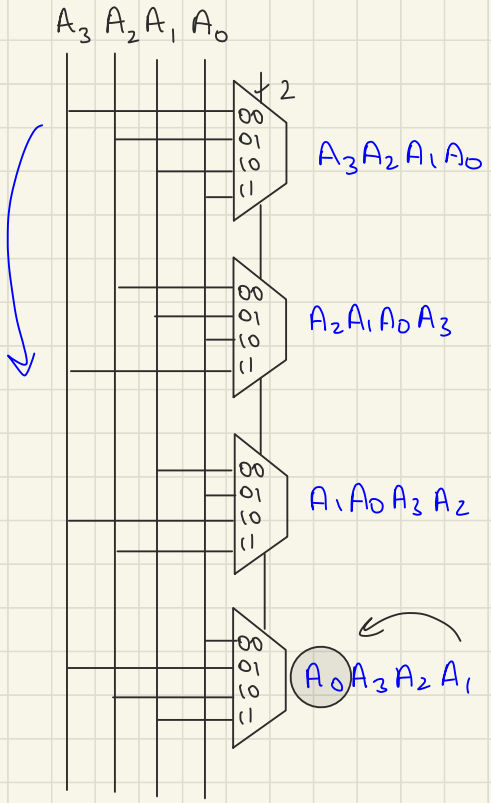




logical left shift design

ex:

Exercise 5.18 – Design 4-bit left and right rotators and sketch your design



1100 ROR 2 = 0011

1100 ROL 2 = 0011

left rotator

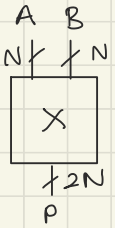
came to the left

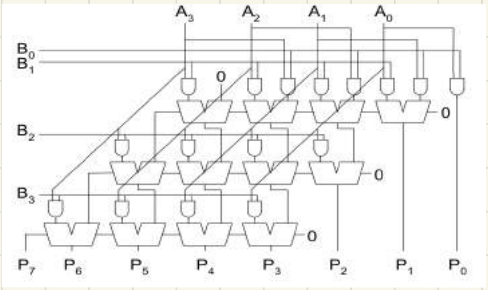
Multipliers

```
  0101
x 0111
------
  0101
 0101
0101
0000
-------
0100011
```

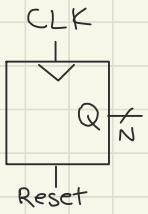
Shifted partial products are summed to form result

Symbol





Counters



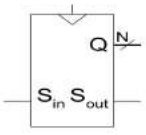
- Reset sets to zero
- Increments on each clock edge up to 2^N
- Cycles through each number
  - ex: N = 3 ⟹ 000, 001, 010, 011 ... 111, 000 ...

Shift Registers

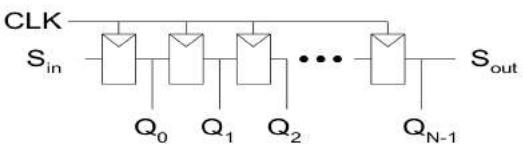
Shifts a new bit in and a bit out in each CLK edge
Converts serial to parallel
Input provided serially (one bit at a time)

~~Shift register~~
=
~~Shifter~~

Symbol:



Implementation:

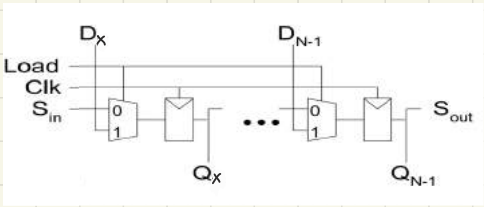


Shift Register with Parallel Load

Load = 0 shift register

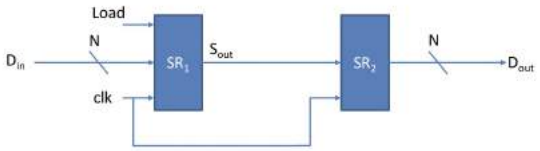
Load = 1 flip flops loaded

↳ now can act as serial → parallel or opposite



Serial Communication

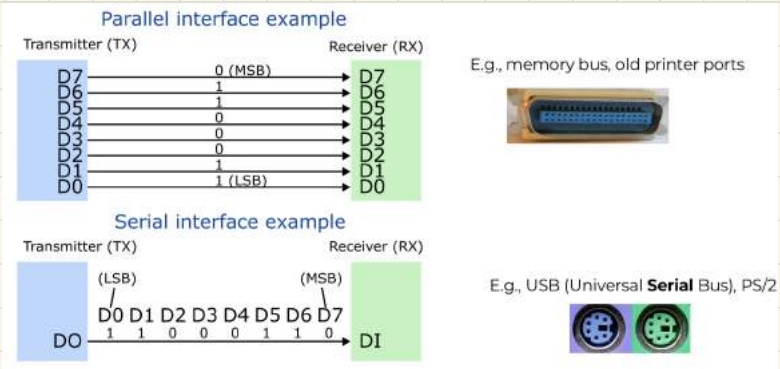
- The shift registers can be used to make a serial transmitter
- N data bits D_{in} are sent to the shifter register SR_1
- The clock and S_{out} signals are sent to the SR_2 receiver
- SR_2 is used to reconstruct the D_{out} data



Why serial communication instead of parallel?

- Physics! (Avoids crosstalk effects)
- Synchronization (all parallel bits must arrive at the same time)
- Even if it transfers less data per cycle, it can have a much smaller cycle compared to parallel communication

Serial vs Parallel Communication



Division

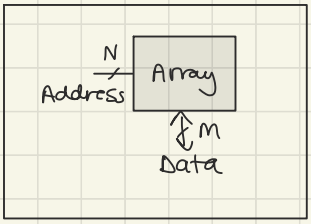
If need be to divide by 2^x → arithmetic shift
Else → a divider is used → division is expensive

Memory Arrays

Registers and flip-flops → can store small amount of data
memory arrays → can store **big** amount of data

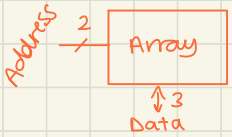
3 types:

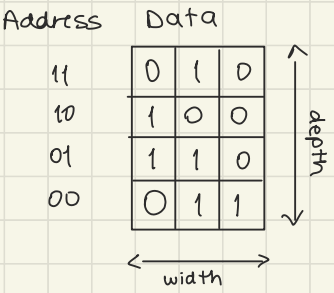
1 – Dynamic random access memory (DRAM)
2 – Static random access memory (SRAM)
3 – Read only memory (ROM)



ex:

2-dimensional
3-bit word



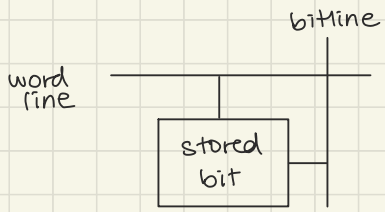


N address bits and *M* data bits:

- **Depth:** number of rows (number of words) = 2^N *(With N bits, we can express 2^N addresses)*
- **Width:** number of columns (size of word) = *M*
- **Array size:** depth × width = $2^N \times M$

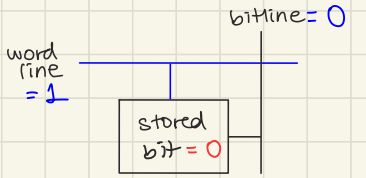
Memory Array Bit Cells

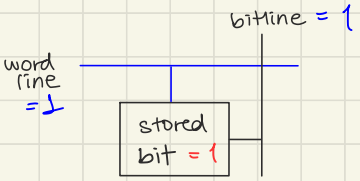
Wordline
- single row in memory array
- corresponds to unique address
- only one wordline HIGH at once



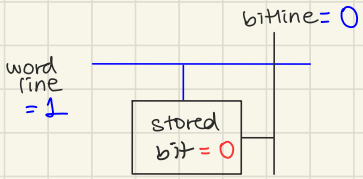
* wordline = 1 ⇒ stored bit transferred to/from bitline
* wordline = 0 ⇒ bitline disconnected
* circuit depends on memory type

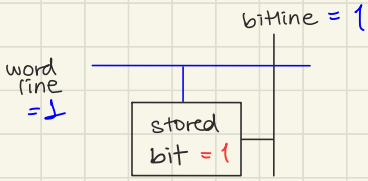
Read



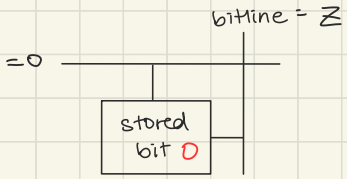


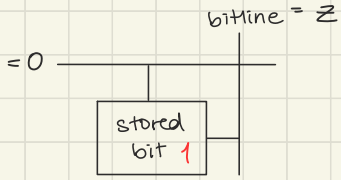
Write

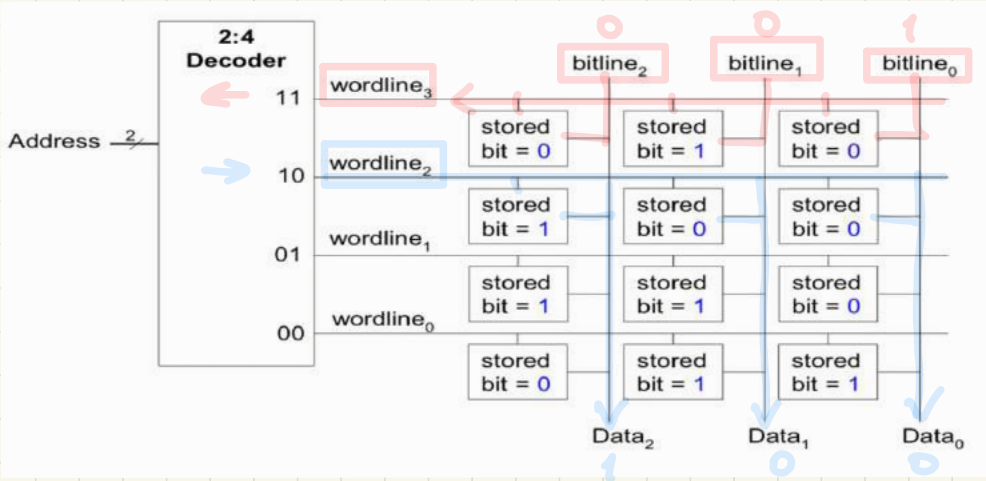




Nothing







How to read

decoder → $wordline_2 = 1$

$wordline_2$ → 100 goes on data lines

How to write

to write 001 to $wordline_3$

bitlines → set to 001

$wordline_3$ → 1

Types of memory

RAM:

Volatile (data lost without power)

Read & written fast

main memory in computer

ROM:

Non-volatile (retains data after powerloss)

Read fast, written slow

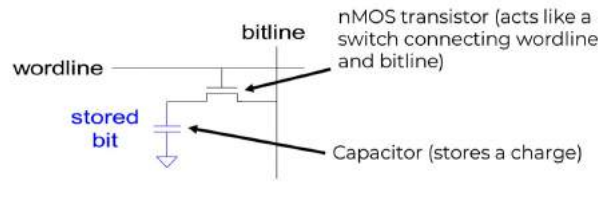
Thumb drives, digital cameras etc.

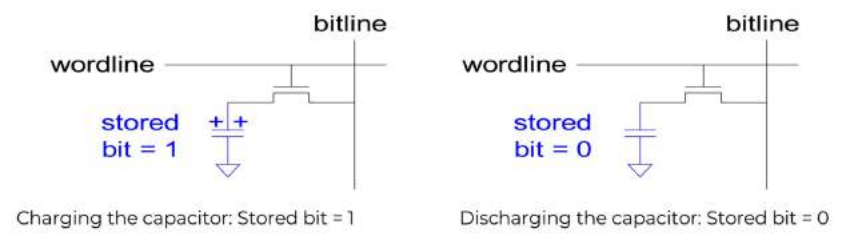
RAM

#1 – DRAM (Dynamic random access memory)

→ Data bits stored on capacitor

→ Dynamic bc. value is rewritten periodically





reading: data transferred from capacitor to bitline → reading destroys stored value

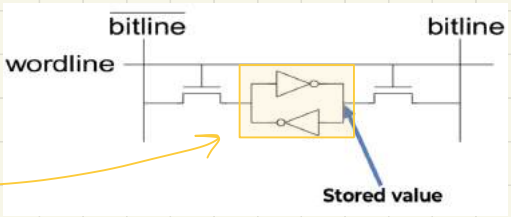
data must be rewritten after every read → even when not read, capacitors slowly discharge

∴ refreshing necessary

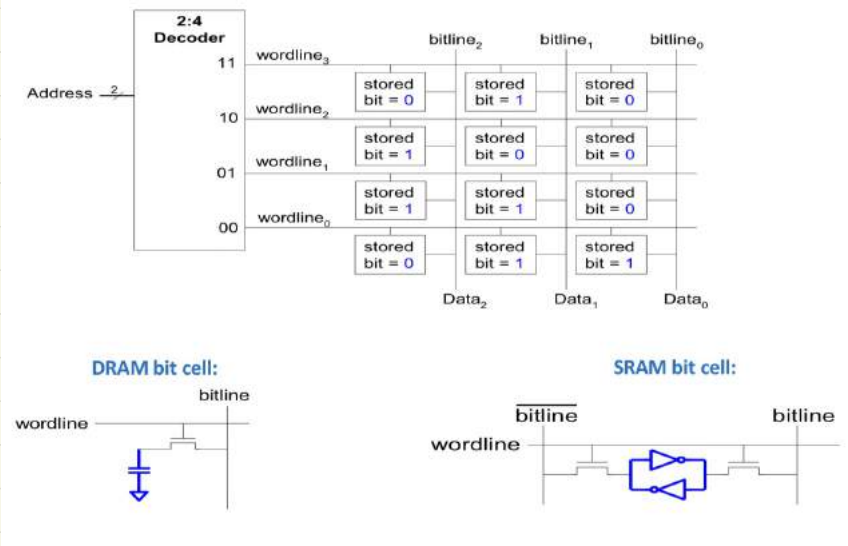
#2 – SRAM (Static random access memory)

→ Data stored on cross-coupled inverters

→ Static bc. no need to refresh data (cross coupled inverter already restores value)



Memory Arrays Review



Tradeoffs

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

* more transistors
  ↳ more power, area, cost

* DRAM has higher latency because charge must move from capacitor to bitline

Memory Ports

A memory can have multiple ports

Port 1 → reads data from A1 to RD1
Port 2 → " " " A2 to RD2
Port 3 → writes WD3 to A3 if WE3=1

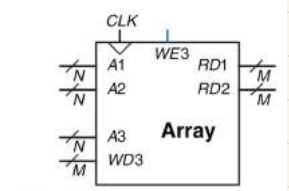


Figure 5.44 Three-ported memory

Register Files

Temporary variables stored in register files, usually SRAM arrays

↓

smaller area compared to a flip flop array

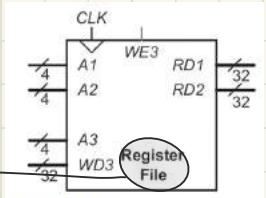
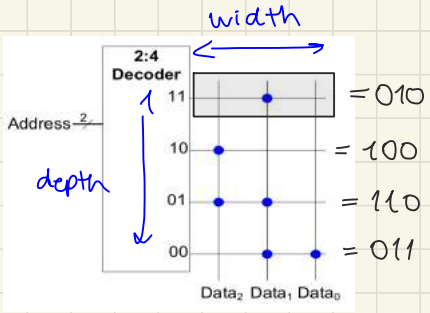


Figure 5.48 16 × 32 register file with two read ports and one write port

ROM

Dot notation

A dot notates that there is no transistor (i.e. = 1)

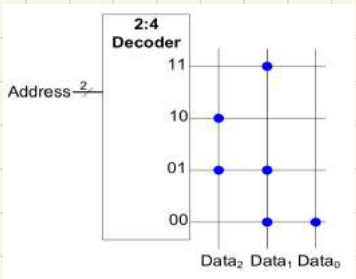


State is persistent without power (is determined by wiring)

ROM's are built from transistors (cost effective)

Logic using PLA / ROM / FFGA

ROM Logic



Although primarily used to store, ROMs can perform combinational logic functions

$Data_2 = A_1 \oplus A_0$

$Data_1 = \overline{A_1} + A_0$

$Data_0 = \overline{A_1}\ \overline{A_0}$

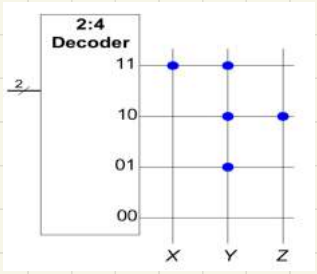
ex:

Logic with ROMs : Implement the following using a $2^3 \times 3$ bit ROM

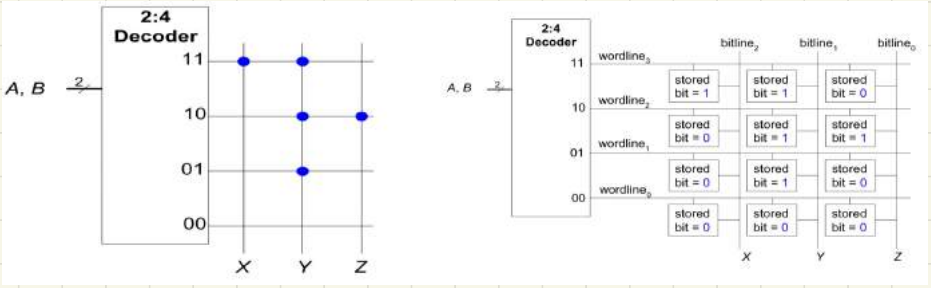
$X = AB$
$Y = A + B$
$Z = A\bar{B}$

A	B	X	Y	Z
0	0	0	0	0
0	1	0	1	0
1	0	0	1	1
1	1	1	1	0

$\Rightarrow$



Logic with any memory Array



Logic

PLA (Programmable Logic Arrays)

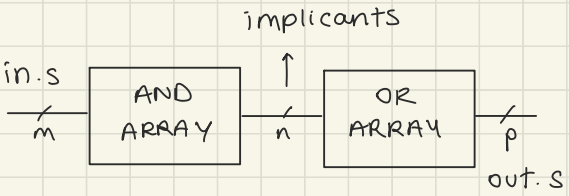
- AND array followed by OR array
- Comb. logic only
- Fixed internal connections

FPGAs (Field Prog able Gate Arrays)

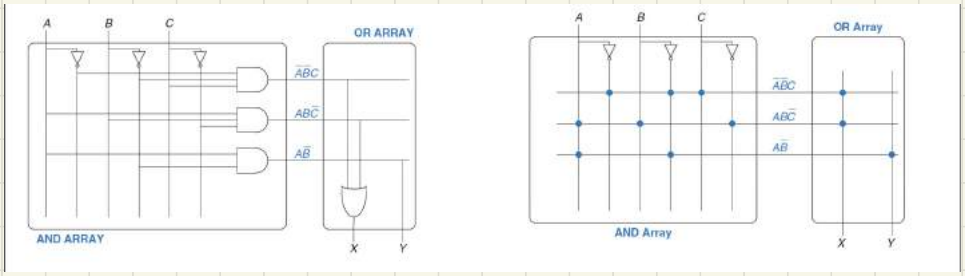
- Array of Logic El.s (LEs)
- Comb. and Seq. logic
- Programmable internal connections

PLAs

general structure :



Dot Notation



ROM can be viewed as a special case of PLAs

FPGAs

An array of reconfigurable gates

Is composed of :

LEs : (Logic Elements) performs logic

I/O Blocks : Input / Output elements

Programmable interconnection : connects LEs & IOEs

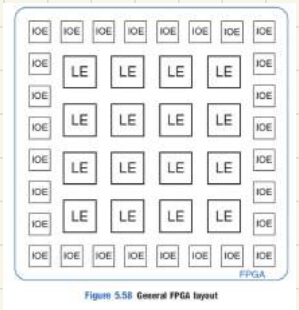


Figure 5.58 General FPGA layout

* more powerful than PLAs (can also perform multilevel and seq. logic)

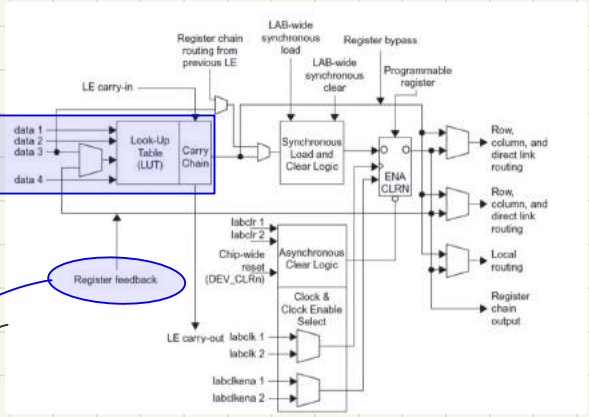
LE (Logic Elements)

Composed of → **LUTs** (lookup tables) performs comb. logic

Composed of → **flip-flops** performs seq. logic

Composed of ↓ **multiplexers** connects the two

ex: **Logic Element**



Has a 4 input LUT

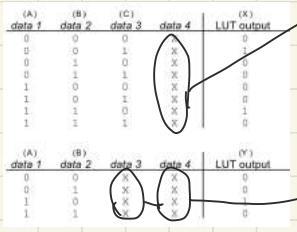
1 bit register

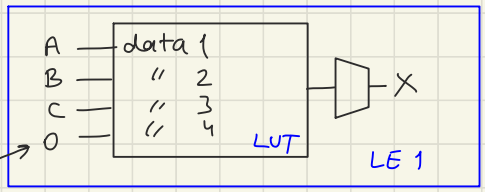
It can perform one combinational and/or registered function of up to 4 variables

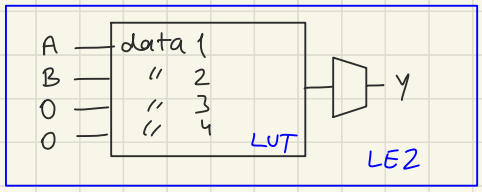
LE Configuration

$X = \bar{A}\bar{B}C + AB\bar{C}$

$Y = A\bar{B}$







Steps for LE config.

1- We need 2 LE's (one for x and y)
2- Set the LUTs with proper values
3- Set the select signals on multiplexers so that the output from the LUT is selected

End of chapter note!

Remember that next chapter is about HDL and that CAD (computer aided design) tools are used to bring these to life.

Enter the design using schematic entry or an HDL
Simulate the design
Synthesize design (create the configuration bitstream)

Nothing is done manually!

Chapter 4

Hardware Description Languages (HDL)

Designers of hardware use Computer-Aided Design (CAD) tool to produce the most optimal product

In general you first simulate and then synthetize

HDL is a rich language, not all comments are synthetized (ex: print → only for simulation)

SystemVerilog Modules

A module takes in x inputs and outputs y outputs

2 types :

→ Behavioral describes what a module does (ex: full adder)

→ Structural describes the inner circuit, how it's built from simpler modules (ex: ripple carry from full adders)

Behavioral SystemVerilog

ex:

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

SystemVerilog

name of module

define module

```
module example(input logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

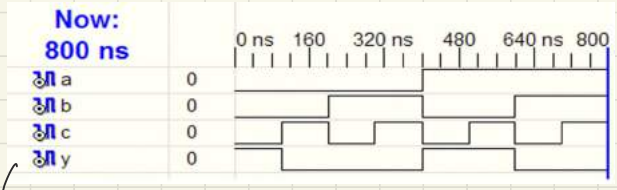
boolean variables (0 or 1)

end definition

combinatial logic

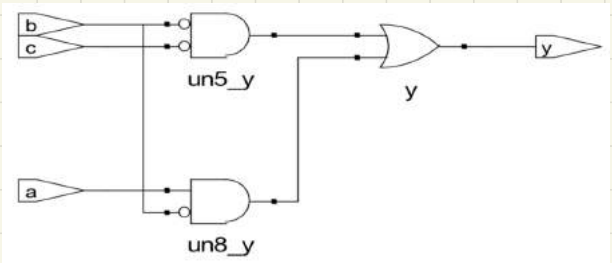
Simulation

★ Waveform used to check if true



↳ y = 1 when abc = 000, 100, 101

Synthesis



done!

you can now build it irl

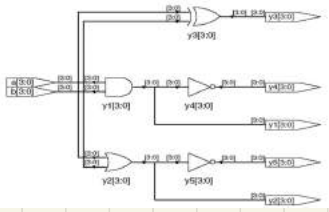
SystemVerilog Syntax

- → Case sensitive (reset ≠ Reset)
- → Names cannot start with numbers (2mux invalid)
- → whitespaces ignored
- → Comments ⟶ // single line & /* multiline comment */

Not executed in order! Executed when input available

Bitwise Operators

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

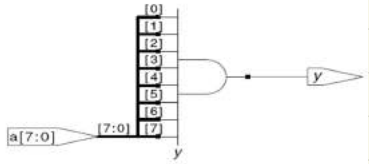


Reduction Operators

```
module and8(input  logic [7:0] a,
            output logic       y);
   assign y = &a;
   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

size of y not specified → default 1 bit

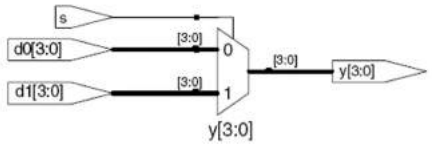
Similar operators for ORs, XORs, etc...



Conditional Assignment

We don't have "if"s in hardware → use MUX's

```
module mux2(input  logic [3:0] d0,
                   logic [3:0] d1,
            input  logic       s,
            output logic [3:0] y);
  assign y = s ? d1 : d0;
endmodule
```



`? :` is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

Internal Variables

- It can be useful to break a complex circuit into intermediate steps
- HDL assign instructions take place concurrently
- E.g., full adder

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

We can define:

$$P = A \oplus B$$
$$G = AB$$

Thus rewriting it as:

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in}$$

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;  // internal nodes
  assign p = a ^ b;
  assign g = a & b;
  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

create internal variables

define internal variables

Precedence

when in doubt use parenthesis!

~	NOT
|	XOR
~|	NOR

Numbers

N'Bvalue → value, base

↓ # of bits

eg: 8'hAB = hexadecimal AB
= 10101011_2
= 171_{10}

Bit Swizzling

[2:1] → order reversed

[3:1] => bits 3, 2, 1

[0] just the 0'th bit

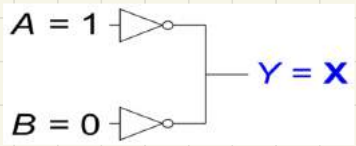
```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// underscores (_) are used for formatting only to make
// it easier to read. SystemVerilog ignores them.

// if y is a 12-bit signal, the above statement produces:
// y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

y[0] = 1'b1;
y[1] = 1'b0;
```

Contention : X

we saw this before!

→ circuit tries to draw output to 0 and 1

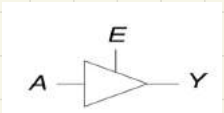
→ usually indicates a bug



Floating : Z

→ output neither 1 nor 0

→ a voltmeter won't indicate floating



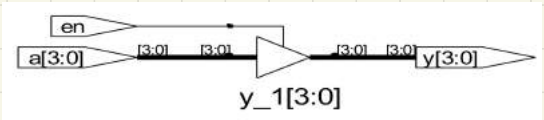
E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Tristate buffer

Z: Floating Output

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
  assign y = en ? a : 4'bz;
endmodule
```

4 bits



SystemVerilog Logic Datatype

→ 0, 1, z (floating), x (invalid)

→ x indicates an invalid logic level, x in simulation is almost always a bug

→ Remember to set/reset flip-flops before requesting output

- If all the tristate buffers driving a bus are simultaneously OFF, the bus will **float**, indicated by *z*.
- If a gate receives a floating (z) input, it may produce an *x* output when it can't determine the correct value

Truth Tables with Undefined and Floating Values

→ SystemVerilog can determine the output sometimes despite unknowns

&	A=0	A=1	A=z	A=x
B=0	0	0	0	0
B=1	0	1	x	x
B=z	0	x	x	x
B=x	0	x	x	x

zero and anyth. is zero

|	A=0	A=1	A=z	A=x
B=0	0	1	z	x
B=1	1	1	1	1
B=z	x	1	x	x
B=x	x	1	x	x

one or anyth. is one

Delays

Helps with debugging

Helpful for predicting speed in simulation

Delays are ignored during synthesis

↳ delay of a gate depends on t_{pd} and t_{cd}

ex:

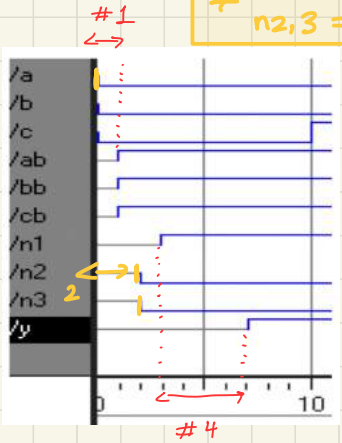
$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

this is executed in time step #1

meaning : wait 4 time units from the moment inputs are available

★ since a=0 n2,3 = 0



Structural modelling

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule
```

3 input AND

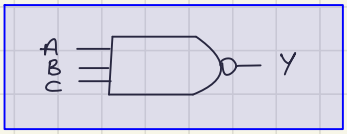
```
module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule
```

NOT gate

```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                      // internal signal

  and3 andgate(a, b, c, n1);     // instance of and3
  inv  inverter(n1, y);          // instance of inv
endmodule
```

3 input NAND

in ABC → out $\overline{ABC}$



---

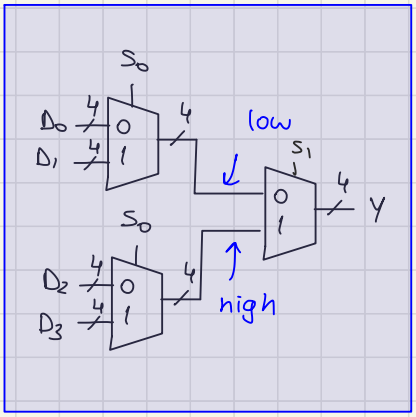
example 1 : 4-bit wide 4:1 MUX starting from 4-bit wide 2:1 MUX

```
module mux2(input  logic [3:0] d0,
                   logic [3:0] d1,
            input  logic       s,
            output logic [3:0] y);
  assign y = s ? d1 : d0;
endmodule
```

4 bit wide 2:1 MUX

```
module mux4(input logic [3:0] d0, d1, d2, d3,
            input logic [1:0] s,
            output logic [3:0] y);
  logic [3:0] low, high;
  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

use 3 of these



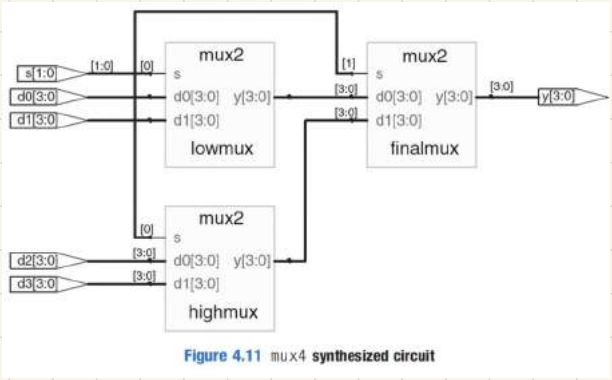
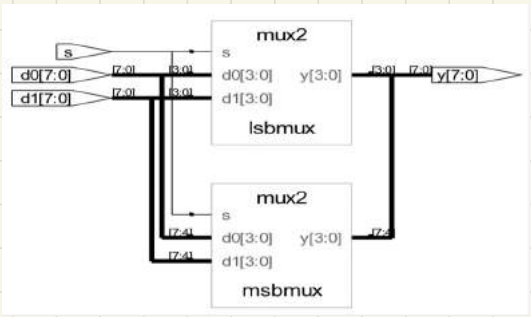


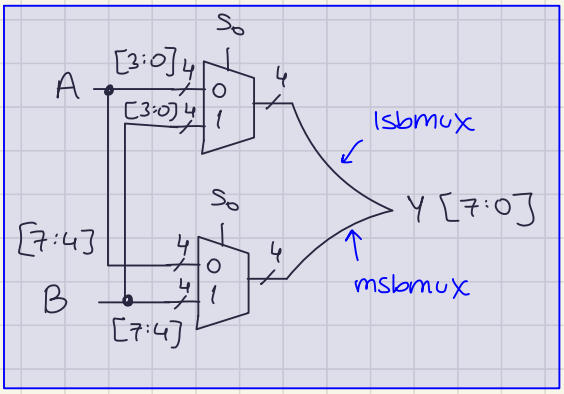
Figure 4.11 mux4 synthesized circuit

Start by the simplest unit when building something complex !

example 2: 8-bit wide 2:1 mux starting from 4-bit wide 2:1 mux

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```





split 8-bit input into 2 4-bits

Sequential Logic

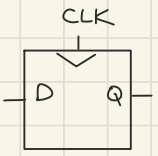
SystemVerilog uses idioms to describe latches, flip-flops and FSMs

Always Statement

```
always @(sensitivity list)
  statement;
```

When the event in sensitivity list occurs, statement is executed

D Flip Flop




```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;

endmodule
```

everytime there's positive (raising) edge of CLK, execute q <= d

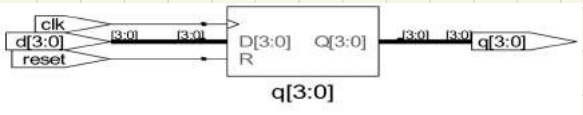
q gets d

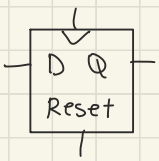
remember : D flip-flop — D=Q only on rising edge

Resettable D Flip-Flop (Synchronous)

resets at clock edge only

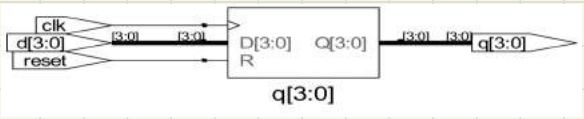
```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```





Resettable D Flip-Flop (Asynchronous) → resets immediately when R = 1

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



same synthesis as sync

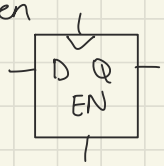
difference : reacts on positive edge of R

D Flip-Flop with Asynchronous Reset and Enable

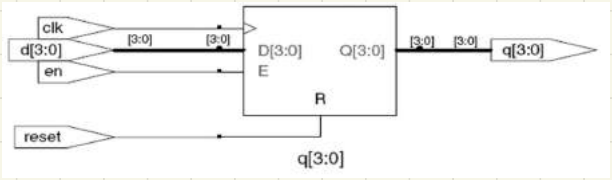
the ENable controls when Data is stored

EN=1 : D passes to Q

EN=0 : Previous state



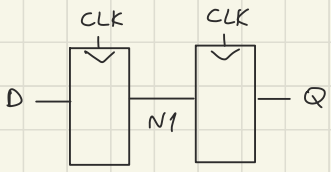

```
module flopren(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```

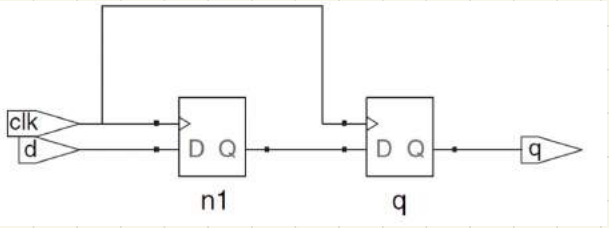


Multiple Registers

```
module sync(input logic clk,
            input logic d,
            output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

passes on



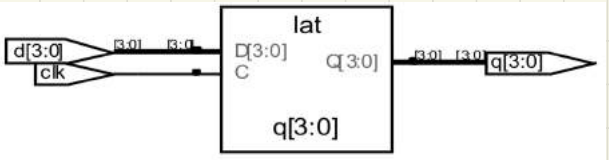


since n1 is neither an input nor an output, it is defined within the module

D Latch


```
module latch(input  logic        clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;

endmodule
```

* a bit problematic use only if you need it
* raises errors



remember D-Latch :

D = Q as long as CLK=1 else Qprev

Combinational Logic with System Verilog

"always" statement is used

if/else and case/casez are used

ex: always statement

```
module inv(input logic [3:0] a,
           output logic [3:0] y);
  assign y = ~a;
endmodule

module inv(input logic [3:0] a,
           output logic [3:0] y);
  always_comb
    y = ~a;
endmodule
```

in this example always_comb means anytime a changes, ↓ always (a)

$y = \overline{a}$

ex: always with multiple statements

```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb          // need begin/end because there is
    begin              // more than one statement in always
      y1 = a & b;      // AND
      y2 = a | b;      // OR
      y3 = a ^ b;      // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

" case "

case statement implies combinational logic iff all possible input combinations described

* remember to use " default " statement

ex: if statement

Priority circuit:

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)

```
module priorityckt(input logic [3:0] a,
                   output logic [3:0] y)
  always_comb
    if (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else y = 4'b0000;
endmodule
```

elif

Blocking vs. Nonblocking Assignments

=

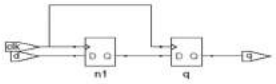
occurs in order

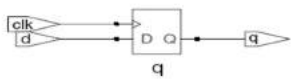
<=

occurs simultaneously with others

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```

```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```





very important

Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)

```
always_ff @ (posedge clk)
      q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (`assign`)

```
assign y = a & b;
```

- **More complicated combinational logic (e.g., in case *if*, *case*, *casez* are needed):** use `always_comb` and blocking assignments (`=`)

- Assign a signal in **only one** always statement or continuous assignment statement (if you are not using 'Z' values).

very important

FSM with SystemVerilog

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic y);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:       nextstate = S1;
         S1:       nextstate = S2;
         S2:       nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign y = (state == S0);
endmodule
```

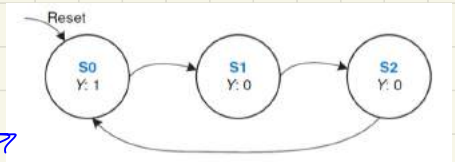
flipflop

resettable flipflop

reset signal or start case

if state is S0 → y=1
else → y=0

enumerates S0 – 00
S1 – 01
S2 – 10

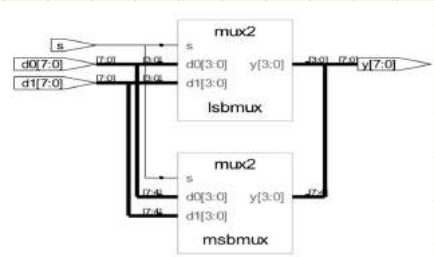


FSM: divide by 3
output =1 every 3 clock cycle

Parametrized Modules

Structural modelling

8-bit wide 2:1 mux starting from 4-bit wide 2:1




```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

Defining parametrized modules allow for there to be no fixed-width inputs/outputs

Parametric 2:1 mux:

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

width becomes a variable

and is used as a variable

Instance with 8-bit bus width (uses default):

```
mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

Testbenches

→ HDL that tests another module : device under test (dut)

→ Not synthesizeable

types :
- → simple
- → self-checking
- → self-checking with testvectors

ex: Testbench Example :

Write SystemVerilog code to implement the following function in hardware

$y = \bar{b}\bar{c} + \bar{a}b$ and name it "sillyfunction"

```
module sillyfunction (input logic a,b,c
                      output logic y);

  assign y = ~b & ~c | ~a & b;

endmodule
```

✓

Simple Testbench & Self checking Testbench

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

```
module testbench2();
  logic a, b, c, y;
  // instantiate device under test
  Sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1; #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("010 failed.");
    c = 1; #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1; #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("110 failed.");
    c = 1; #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

Testbench with Testvectors

Checking all cases with SystemVerilog can be error-prone

Testvector file : inputs and all expected outputs

ex: test.tv contains vectors of

000_1
001_0
⋮
111_0

} abc_y expected

1. Generate clock for assigning inputs, reading outputs
2. Read testvectors file into array
3. Assign inputs, expected outputs
4. Compare outputs with expected outputs and report errors

Lecture 23

left at testbench clock

4. Compare with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y, yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```

selects variables one after the other

binary

4. Compare with Expected Outputs

```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                   vectornum, errors);
          $finish;
      end
    end
endmodule

// === and !== can compare values that are 1, 0, x, or z.
```

i'm done

XXXX

Adder

```
module adder #(parameter N = 8)
              (input logic [N-1:0] a, b,
               input logic cin,
               output logic [N-1:0] s,
               output logic cout);

  assign {cout, s} = a + b + cin;
endmodule
```

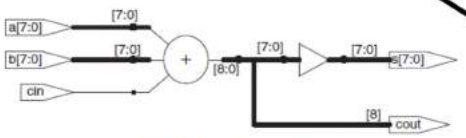


Figure 5.8 Synthesized adder

SystemVerilog provides a + operator to specify a CPA. Synthesis tools will select the cheapest (smallest) design that meets the timing requirements.

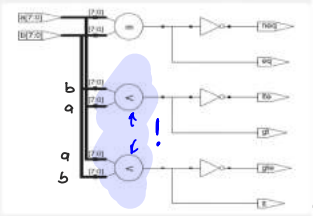
full adder

Cout S

11010

Comparators

```
module comparator #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt,
  gte);

  assign eq = (a == b);
  assign neq = (a != b);
  assign lt = (a < b);
  assign lte = (a <= b);
  assign gt = (a > b);
  assign gte = (a >= b);
endmodule
```



systemverilog simplifies a lot of things

ex:

Counters

```
module counter #(parameter N = 8)
                (input logic clk,
                 input logic reset,
                 output logic [N-1:0] q);
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= q + 1;
endmodule
```

(nonblocking) in sequential circuits you should use nonblocking operator

q = q+1

Shift Register With Parallel Load

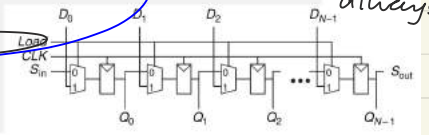
```
module shiftreg #(parameter N = 8)
                 (input logic clk,
                  input logic reset, load,
                  input logic sin,
                  input logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic sout);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (load) q <= d;
    else q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

reset is 1 bit

8 bit d

8 bit q

flip flops everytime we need resettable flip flops, always operator



if we are not resetting nor loading: shifted

RAM (With synchronous writes)

```
module ram #(parameter N = 6, M = 32)
            (input logic clk,
             input logic we,
             input logic [N-1:0] adr,
             input logic [M-1:0] din,
             output logic [M-1:0] dout);

   logic [M-1:0] mem [2**N-1:0];

   always_ff @(posedge clk)
     if (we) mem [adr] <= din;

   assign dout = mem[adr];
endmodule
```

write enable

n bit address

n bit data

data out

each is m bits

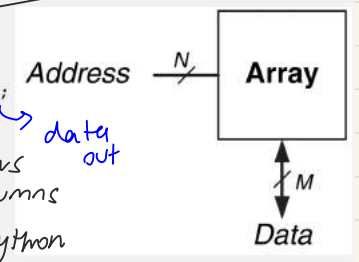
2^N rows

N columns

writing (if we=1)

reading

like matrices in python



ROM

```
module rom(input logic [1:0] adr,
           output logic [2:0] dout):

always_comb
  case(adr)
     2'b00: dout = 3'b011;
     2'b01: dout = 3'b110;
     2'b10: dout = 3'b100;
     2'b11: dout = 3'b010;
  endcase
endmodule
```

Is going to be synthetized into logic gates or into an array depending on what's the best solution (depends on the size)