Digital Logic

Analog vs Digital

Sampling & quantisation

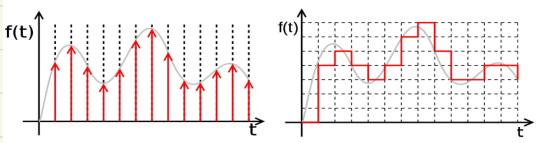
- Changes in a continuous manner
  - ↳ Day to night
  - ↳ Temperature increase

- Δ of quantity are discrete
  - ↳ 12:34 pm → 12:35 pm

f(t) is an analog signal continuously varying with time (gray curve)

Sampling f(t) at periodic intervals will generate the discrete time signal (red arrows)

Quantisation of the discrete time signal will produce the digital signal (red lines)



Logic Gates

Boolean Constants
- ↳ 2 values
  - TRUE / FALSE
  - Logic HI / logic LO
  - 1 / 0

waveform diagrams

1 0 0 1 1

Boolean Laws

$a + ab = a(1+b)$
$= a$

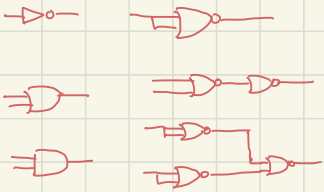
$a + a'b = (a+ab) + a'b$
$= a + b(a+a')$
$= a + b$

$ab + a'c + bc = ab + a'c + bc(a+a')$
$= a(b+bc) + a'(c+bc)$
$= ab + a'c$

$X = a + b$
$X' = (a+b)'$
$= a' \cdot b'$

$Y = a\,b$
$Y' = (a \cdot b)'$
$= a' + b'$

NOR



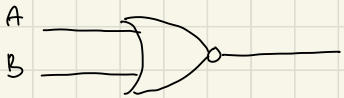
$(a' + b')' = a \cdot b$

NAnd

$X = (AB)'$

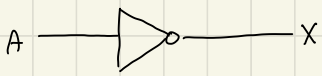
NOR

$X = (A+B)'$



Not Gate

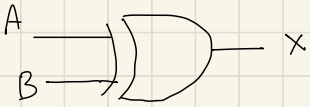
$X = A'$

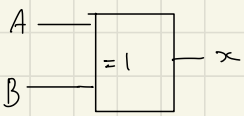


XOR Exclusive - OR Gate

$X = AB' + A'B$

$X = A \oplus B$





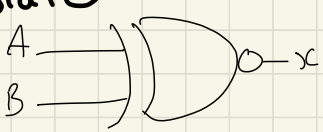
When $A \neq B$, $X = 1$

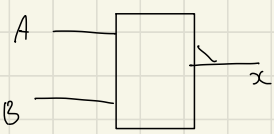
$A = B$, $X = 0$

XNOR Exclusive-NOR Gate

$X = AB + A'B'$

$X = (A \oplus B)'$





When $A = B$, $x = 1$

$A \neq B$, $x = 0$

XOR Application

Compare two circuits : Bitwise comparator

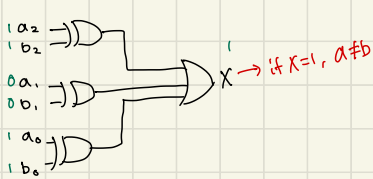
$\underline{a}$ $\underline{b}$

3Bits output 1 when $a \neq b$

// let $\underline{a}$: $a_1 a_2 a_0$

let $\underline{b}$: $b_1 b_2 b_0$

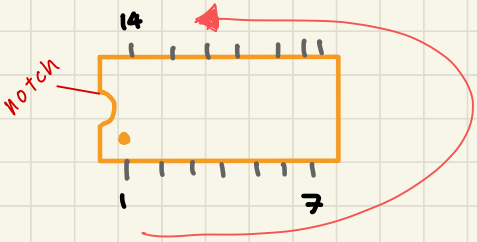
Comparing Multi-bit input



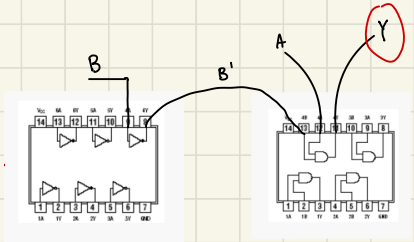
$A \oplus B \oplus C$ = Only = 1 when odd number of ones!

Physical Logic Devices

integrated circuits



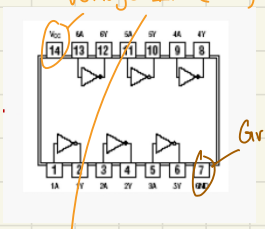
Create $Y = AB'$



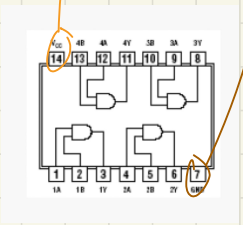
Voltage IN (VCC)

Ground

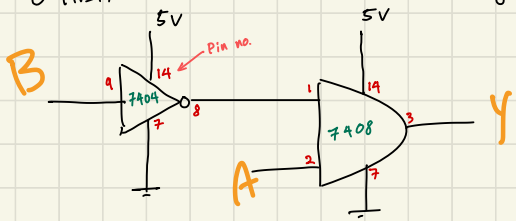
7404
Hex - NOT
↓
6 not gates



7408
QUAD - AND
↓
4 and gates



① first, Create circuit connection diag.



Parity Bit

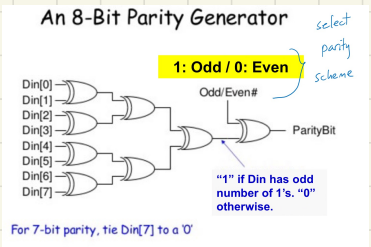
Even parity : Add a bit that makes it Even

Odd parity : Add a bit that makes it Odd

Even : 011011

Odd : 111011

now become odd num. of ones



If odd number of

Digital Arithmetic

- Begin with LSB

$10111_2 + 1010_2 = ?$

$1 + 1 = 2_{10} \rightarrow 10_2$

$10000_2 = 33_{10}$

Digital circuits can also do this

> HALF adder (HA)

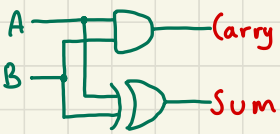
- logic circuit that adds two bits

Carry : $A \cdot B$

Sum : $A \oplus B$

MSB LSB

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



> Full Adder (FA)

Carry input

Inputs			Outputs	
A	B	Cin	Cout	Sum
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

$1 + 1 + 1 = 3$

$11_2 = 3$

Sum : $A \oplus B \oplus C$

Carry out : $A \cdot B + B \cdot C + A \cdot C$

As long as two inputs are 1, carry out = 1

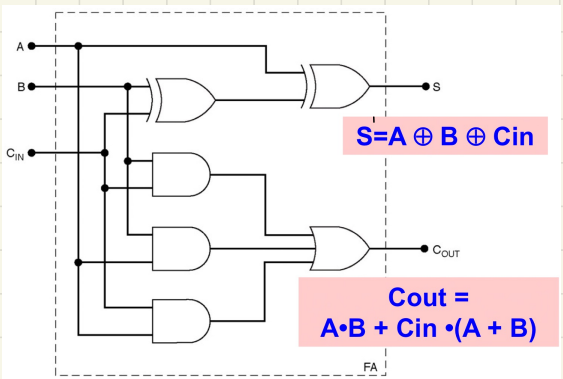
Carry Out : $A \cdot B + B \cdot C + A \cdot C$

$= A \cdot B + (A'B + AB)C + (B'A + BA)C$

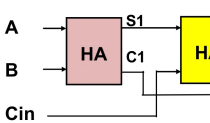
$= A \cdot B + CAB + CBA + CA'B + CB'A$

$= A \cdot B + CAB + C(A'B + AB')$

$= A \cdot B + C(A \oplus B)$



Full adder circuit implemented using 2 half-adders and an OR gate



$S1 = A \oplus B$

$C1 = A \cdot B$

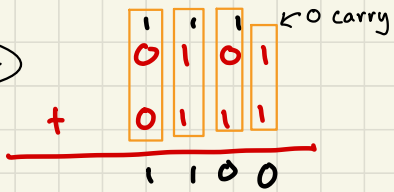
$S2 = (A \oplus B) \oplus Cin$

$C2 = Cin \cdot (A \oplus B)$

Binary Addition of : $0101_2 + 0111$

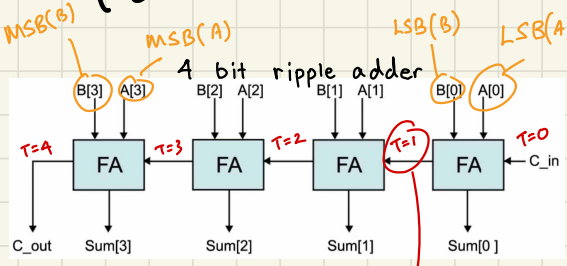
Need 4 FA...

each FA has 3 values, the carry and the 2 bits



Parallel Adder : Ripple adder, where full adders are cascaded...

* propogation Delay



Signed numbers

① Sign Magnitude System

Sign bit = MSB : 0 for +ve numbers
MSB : 1 for -ve numbers

Range For N bits

$-(2^{N-1}-1)$ to $+(2^{N-1}-1)$

14 : 0 1110
-14 : 1 1110

② 2's complement system

Sign bit = MSB : 0 for zero & +ve
MSB : 1 for -ve

Range For N bits

$-(2^{N-1})$ to $+(2^{N-1}-1)$

14 : 0 1110
-14 : 1 0010

① Invert every bit .... perform 1's complement

1110 → 0111

② Add (arithmetic) 1 to it

```
  0111
+    1
------
  0010
```

1110 is the 2's complement of 0010

Shortcut method for 2's complement

① From LHS, copy 0s until you hit the first 1 bit
② Invert the remaining bits

This comes from +1176

0 1 0 0 1 0 0 1 1 0 0 0

1 0 1 1 0 1 1 0 0 0

-1176

Two's compliment

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

Always 100000....

-1 is all ones

0000

Sign bit=0, the rest all one

Decimal	2's comp
$-(2^{N-1})$	1000…00
$-(2^{N-1}) + 1$	1000…01
$-(2^{N-1}) + 2$	1000…10
.	.
.	.
-1	1111…11
0	0000…00
1	0000…01
2	0000…10
.	.
.	.
$(2^{N-1}) - 2$	0111…10
$(2^{N-1}) - 1$	0111…11

Decimal Values

Special Bit Patterns

$-(2^{N-1})$	100…000, N-1 zeros
-1	All ones
0	All zeros
$2^{N-1} - 1$	with N-1 ones

Converting 2's Comp to Binary

+ve, Perform normal conversion 01001 = 9

-ve, Peform 2's Comp to convert to +ve 10011 = 01101
Peform normal conversion = -13

Sign extension

When there are more bits necessary to represent binary,
> MSB will be filled with sign bits.
101 = 1101 = 11101 011 = 0011 = 00011

Also, 2's compliment will △ -ve/+ve w/o △ magnitude
> except for most -ve value (10000)
> -8 in 4-bit system
-16 " 5-bit "
-128 " 8-bit "

} Because 8, 16 & 128 don't exist

Why use 2's ?

- Same circuit for addition & subtraction

☆ Universability !
Less space !
Cheaper !

$A - B = A + (-B)$: Just add it together !

2's compliment

ADDITION IN 2's

+10	0	1 0 1 0	(augend)
+3	0	0 0 1 1	(addend)
+13	0	1 1 0 1	(sum)

Sign bit must be aligned

Only look at relevant bits

+10		0	1 0 1 0
-3		1	1 1 0 1
+7	1	0	0 1 1 1

Ignore carry out

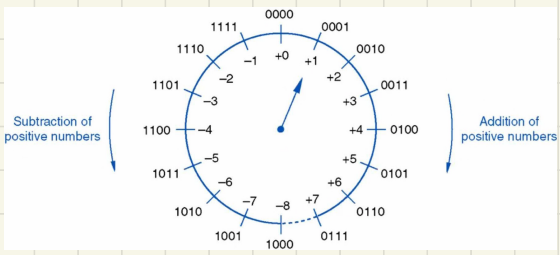
sign bit

+10		0	1 0 1 0
-3		1	1 1 0 1
+7	1	0	0 1 1 1

1 1 0 1 Sign extension

Ignore carry out

sign bit



Arithmetic Overflow

- When operation between two numbers produces result that exceeds current bit. 4-bit : (-8 to 7) cannot do $-7-5 = \underbrace{-12}_{\text{5-bit}}$

	1	0	1	0	1 0
+	0	1	0	0	0
	1	0	0	1	0

Case 1

$x - y$ Always OK

Case 2

$-x - y$ Overflow! Sign bit +ve

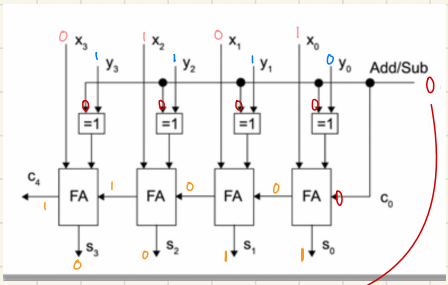
Case 3

$x + y$ Overflow! Sign bit -ve

Using full adders

Addition

$x_3x_2x_1x_0 + y_3y_2y_1y_0$

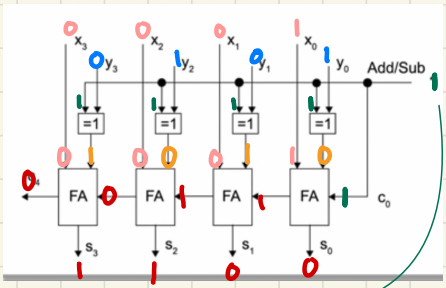


0 1 0 1

1 1 1 0

1 0 0 1 1

Subtraction



0 0 0 1

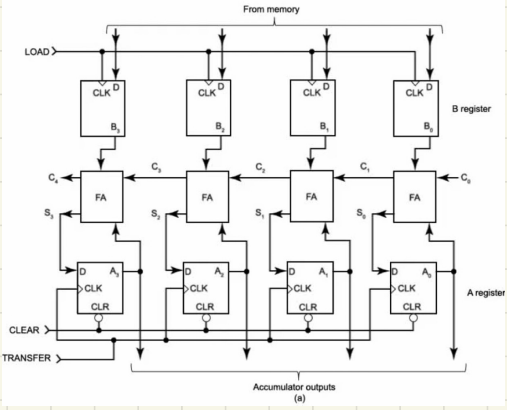
0 1 0 1

0 1 1 1

1 1 0 0 → 0 1 0 0 = -4

Register

- Memory storage to store bits
- Timing signals control circuit sequence



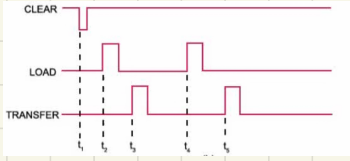
t1: CLEAR* clears the contents of A register to 0's

t2: PGT of first LOAD pulse transfers operand X from memory into B register

t3: PGT of first TRANSFER pulse transfers FA output (=X) into A register

t4: PGT of second LOAD pulse transfers operand Y from memory into B register

t5: PGT of second TRANSFER pulse transfers FA output (=X+Y) into A register



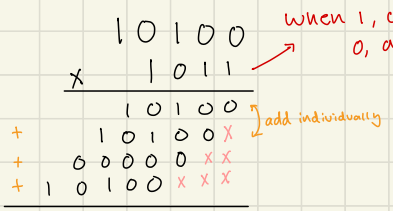
Overflow if

B[3] = A[3] but ≠ S[3]

OR

C_out ≠ C_in

Binary Multiplication



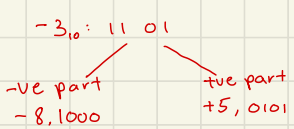
* multiplying two numbers of m-bits will not exceed 2m-bits of memory.

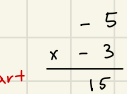
-Ve Binary

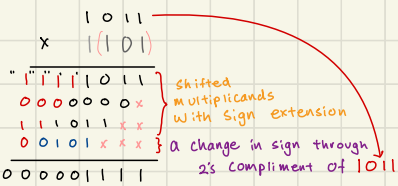
Option One: Convert signed t unsigned, then multiply

Option Two: If multiplier is -ve

1) Treat value as sum of 2 parts







BCD addition

adding bcd, the result may be more than 9

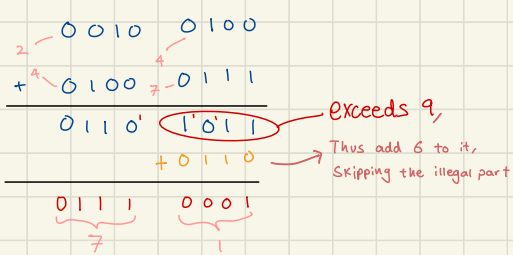
> a correction needs to be made by adding 6 (0110)

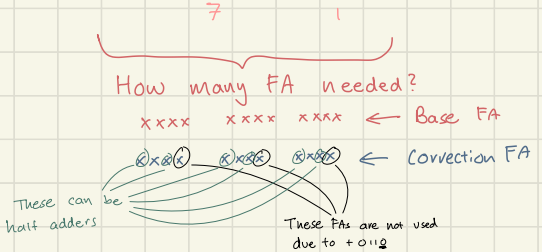
Two Steps

The correction involves two steps:

- **a carry of decimal value 1 is brought forward and added to the next higher digit**
- **the decimal value 6 is added to the sum to obtain the correct BCD digit**

$24_{10} + 47_{10} = 71_{10}$





Combinational logic circuit

- No memory, it is a "reactive" circuit
- Unlike sequential circuits that have memory
- USE TRUTH TABLE, then can impliment

Self study review

Boolean Expression

- also known as
  - ↳ Boolean Equation
  - ↳ logic function

Canonical Form
1. Sum of minterm expression
2. Product of maxterm expression

Minterm: All possible combinations formed by AND

Maxterm: All possible combinations formed by OR

Using AND, what result gives m0 = 1

Convert number to dec = M2

Using OR, what will make M2 = 0

inputs			minterms		maxterms	
X	Y	Z				
0	0	0	X' • Y' • Z'	m0	X + Y + Z	M0
0	0	1	X' • Y' • Z	m1	X + Y + Z'	M1
0	1	0	X' • Y • Z'	m2	X + Y' + Z	M2
0	1	1	X' • Y • Z	m3	X + Y' + Z'	M3
1	0	0	X • Y' • Z'	m4	X' + Y + Z	M4
1	0	1	X • Y' • Z	m5	X' + Y + Z'	M5
1	1	0	X • Y • Z'	m6	X' + Y' + Z	M6
1	1	1	X • Y • Z	m7	X' + Y' + Z'	M7

For N-inputs, there will be 2^N minterms

e.g. 4 inputs: a, b, c, d
- 13 in decimal = 1101 in binary
- Then minterm m13 = a b c' d
- maxterm M13 = a' + b' + c + d'
- 2 in decimal = 0010 in binary
- Then minterm m2 = a' b' c d'
- maxterm M2 = a + b + c' + d

Minterm: Achieve 1. any 0s will prime

Maxterm: Achieve 0 any 1s will prime

inputs			minterms		maxterms		F
X	Y	Z					
0	0	0	X' • Y' • Z'	m0	X + Y + Z	M0	0
0	0	1	X' • Y' • Z	m1	X + Y + Z'	M1	1
0	1	0	X' • Y • Z'	m2	X + Y' + Z	M2	1
0	1	1	X' • Y • Z	m3	X + Y' + Z'	M3	0
1	0	0	X • Y' • Z'	m4	X' + Y + Z	M4	1
1	0	1	X • Y' • Z	m5	X' + Y + Z'	M5	0
1	1	0	X • Y • Z'	m6	X' + Y' + Z	M6	0
1	1	1	X • Y • Z	m7	X' + Y' + Z'	M7	1

$F = X'Y'Z + X'YZ' + XY'Z' + XYZ$
$= \Sigma_{XYZ}(1, 2, 4, 7)$ **SOm**

$F = (X+Y+Z)(X+Y'+Z')(X'+Y+Z')(X'+Y'+Z)$
$= \pi_{XYZ}(0, 3, 5, 6)$ **POM**

How to compute?

1. Find relevant mn ∑ Mn
2. For minterms, Look for output = 1
   - $m1 + m2 + m4 + m7$, $\sum_{xyz}(1, 2, 4, 7)$
   - ↳ Odd term, active High
3. For maxterms, Look for output = 0
   - $(M_0)(M_3)(M_5)(M_6)$, $\pi_{xyz}(0, 3, 5, 6)$
   - ↳ even term, active Low

Standard Form of Boolean Exp

Sum of Product

- Minimization : Reduce logic gates & inputs on each logic gate.

① Compress sum of minterms into produt term (no need all variables XYZ)

example:
This is a sum-of-minterms expression:
f(x, y, z) = xyz' + xyz + x'y'z + xy'z

Simplifying, we get
f(x, y, z) = xy (z' + z) + (x' + x) y'z
= xy + y'z

Product of sum

① Compress maxterms into Sum term

example:
This is a product-of-maxterms expression:

f(x, y, z) = (x+y'+z')(x+y'+z)(x'+y'+z)(x'+y+z)

Simplifying, we get
f(x, y, z) = (x + y')(x' + z)

These are neither SOP nor POS expressions:

f = (xy)'z + xz' — not a product term

f = xy(x' + z)' — not a sum term

f = (xy + z)(x' + y) — not a sum term

WAYS TO SIMPLIFY

- Algebraic method
- Karnaugh map (K-map)
- Quine-McCluskey method (Q-M method or tabulation method)
- Heuristic methods, e.g. Espresso-II

Karnaugh Map

1) Based on $AB + AB' = A(B+B') = A$

$(A+B)(A+B') = AA + AB + AB' + BB'$
$= A + A + 0$
$= A$

- K-map squares are labelled such that adjacent squares differ only in one variable.
- SOP expression for output X can be obtained by ORing together those squares that contain a 1.
- Can also obtain POS expression by ANDing together those squares that contain a 0.
- Note the correspondence with SOm (think 1) and POM (think 0).

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

X = A'B'C'D + A'BC'D + ABC'D + ABCD

X	C, D: 00	01	11	10
A, B: 00	0	1	0	0
01	0	1	0	0
11	0	1	1	0
10	0	0	0	0

look! Only one var change:

C,D = 0,0
C,D = 0,1 ↓ Change
C,D = 1,1 ↓ one at a
C,D = 1,0 ↓ Time

loop neighboring Ones

A, B \ C, D	00	01	11	10
00	0	1	0	1
01	0	1	0	0
11	0	1	1	0
10	1	0	0	1

A'C'D

B'CD'

This loop wraps around

AB'D'

ABD

This loop wraps around too

Loop 4 neighboring Ones

A, B \ C, D	00	01	11	10
00	1	1	1	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

A'B'

BD

B'D'

Loop 8 neighboring Ones

A, B \ C, D	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

D

- Only can loop 2^n neighbours together.
- No diagonal looping
- Loops can be more than once, "double dip"
- Similarly, looping zeros to obtain POS is possible, giving a sum term

Don't care conditions

Example: Design a logic circuit whose input is a BCD digit, and whose output goes HIGH if the input is smaller than 6_{10}

Z < 6

A	B	C	D	Z	
0	0	0	0	1	0
0	0	0	1	1	1
0	0	1	0	1	2
0	0	1	1	1	3
0	1	0	0	1	4
0	1	0	1	1	5
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	X	cannot be BCD digit.
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	

X denotes "don't care"

Z \ CD	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

AB

$Z = A'B' + A'C'$

Z \ CD	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

AB

$Z = B'C + A'C'$

Z \ CD	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

AB

$Z = A' (B'+C')$

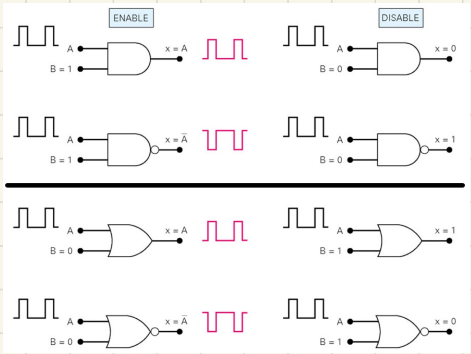
Treat X as one & zero: it doesn't matter!

Enable / Disable circuits

Enable: Output can Δ based on Input

Disable / Inhibit: Output cannot Δ, fixed @ 0 or 1

Basically, it's like a "child lock"



Inputs		Output	Effect of output
Child Safe	Unlock	Open	
0	X	0	Door closed
1	0	0	Door closed
1	1	1	Door opened

X: "don't care", i.e. can be 0 or 1

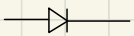
- **Circuit is enabled when ChildSafe=1; the output Open changes with input Unlock.**
- **Circuit is disabled when ChildSafe=0; Open is stuck at 0.**

	Enable input	When Enabled	When Disabled
AND	Active High	non-inverted	0
NAND	Active High	inverted	1
OR	Active Low	non-inverted	1
NOR	Active Low	inverted	0

Digital Circuits

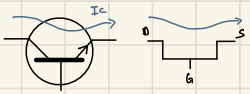
- Transistors, resistors & diodes on silicone chip.

Diodes



- One direction
- Includes LED

Transistor



- Can be used as switch

TTL → old, but

1) Simple fabrication
2) Low power dissipation
3) Transistor only
4) packing density
5) Greater fan out

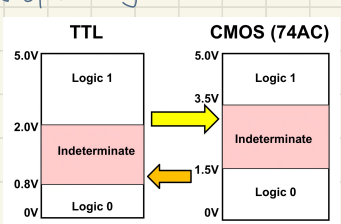
TTL family

- Vcc @ 5 volts
- Unconnected (floating) will act as logic 1, 1.4 to 1.8 volt

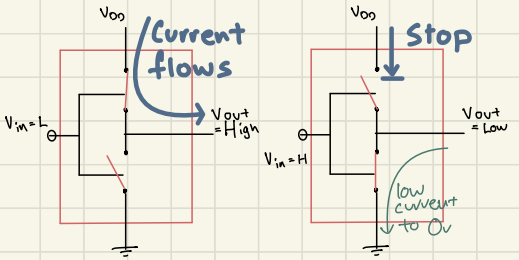
CMOS family

- V_{DD} @ 3v to 18v (use 5v?)
- Floating pins are BAD ↳ Overheating. All must connect to ground/V_{DD}

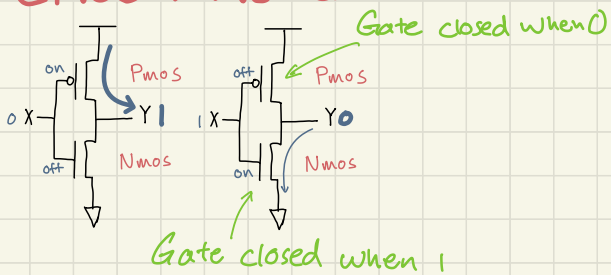
→ Some cmos incompatible ☆ operating in indeterminate



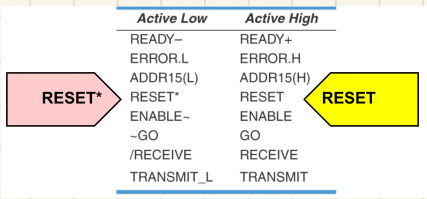
TTL Inverter



CMOS Inverter



Active high / Active low



for this mod,
reset* = low
reset = high

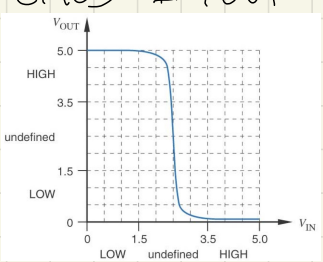
Signal name	Effect/result when signal=0	Effect/result when signal=1	Active high/low
Subtract	Add Y to X	**Subtract** Y from X	Active high
Add*	**Add** Y to X	Subtract Y from X	Active low
Unmute	Mute a speaker	**Unmute** a speaker	Active high
Unmute*	**Unmute** a speaker	Mute a speaker	Active low
Read	Write data	**Read** data	Active high
Write*	**Write** data	Read data	Active low

Active high or active low depends on the signal name and effect/result

	Active High	Active Low
asserted	1	0
negated	0	1

Transfer Characteristics

CMOS IN/OUT



☆ current is similar

V_{OH} min out volt for High e.g. 3.5v and above

V_{IH} min input volt for High e.g 2.0v

V_{OL} max output volt for Low

V_{OH} max input volt for Low

Basically: $V_{OH} > V_{IH}$ output > input for High

$V_{OL} < V_{IL}$ output < input for Low

Noise Margin

V_{OH} → Noise, ↑↓ Voltage → V_{IH}

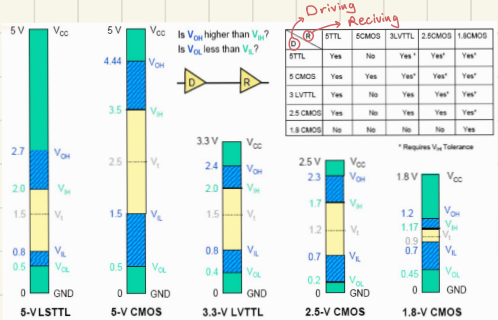
↳ V_{IL} (noise dips V_{OH})

High (noise margin) = V_{OH} (min) − V_{IH} (min)

= 3.5 − 2.0

= 1.5 volt margin

☆ for overall, will be smallest of the two



Fan Out (current related)

: How much gates can you connect to one output.
Current is "spread" amongst ICs

Dc fan out = $\frac{I_{OH}}{I_{IH}}$

Always, $I_{out} \geq I_{IN}$

↑ connections will ↑ noise ⅗ ↓ switching speed

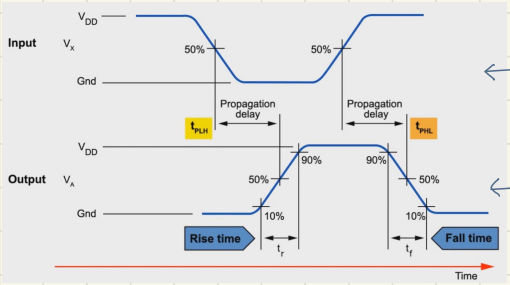
Dynamic Switching

P, Power Dissipation: When Ic switches between 1 ⅗ 0, power consumed
When not swiching, Low dissipation

$P = CV^2f$ C = Constant f = Swiching frequency
V = Power supply voltage

propogation delay: time to transition between input ⅗ output (nano seconds)

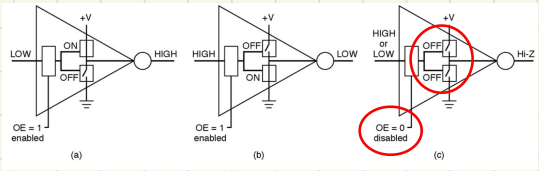
t_{PHL} = High to Low time t_{PLH} = Low to High time



← Identify from 50% point

← Fall time from 10% to 90%

Tristate Output

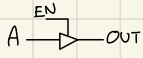


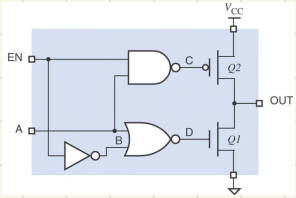
3 logic states
① logic 0
② logic 1
③ High impedance (Hi-Z)
→ Like an "open" circuit

EN	A	B	C	D	Q1	Q2	OUT
L	L	H	H	L	off	off	Hi-Z
L	H	H	H	L	off	off	Hi-Z
H	L	L	H	H	on	off	L
H	H	L	L	L	off	on	H

Enable High

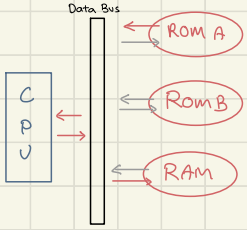
CMOS Tristate Buffer



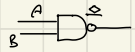


Advantage

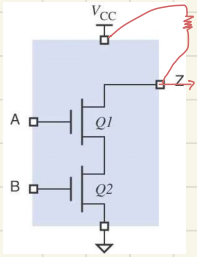
→ Can connect output together
↳ Only one output is enabled (max)
↳ Bus Contention, damage to devices
(used to connect memory modules, share bus)



Open Collector / Drain output

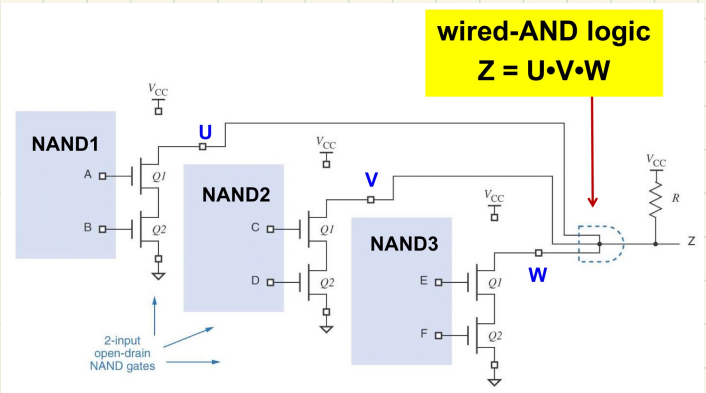


Pull-up resistor to make output = 1 when A/B = 0



A	B	Q1	Q2	Z
L	L	off	off	open
L	H	off	on	open
H	L	on	off	open
H	H	on	on	L

When A & B are high, Z = 0



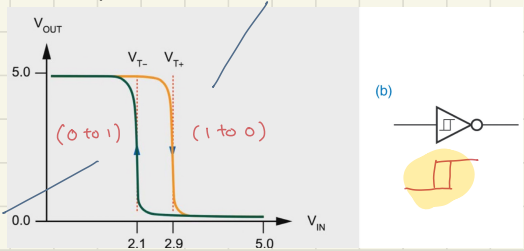
Constructing "AND" gates using only wires?

1) If any of the drains are open, provides flow through ground.

2) When all are "closed", current has to flow through Z

Schmitt-Trigger Inputs

VT+

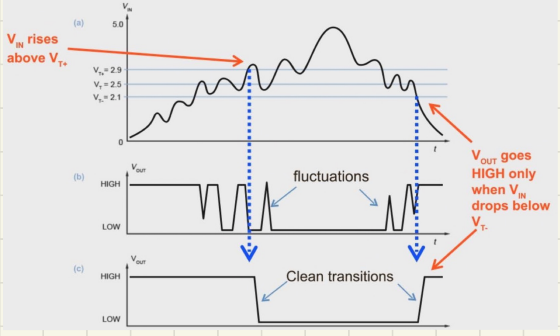


VT-

Input volt rise above VT+, Vout = Low

Input volt drops below VT-, Vout = High

☆ Hysteresis

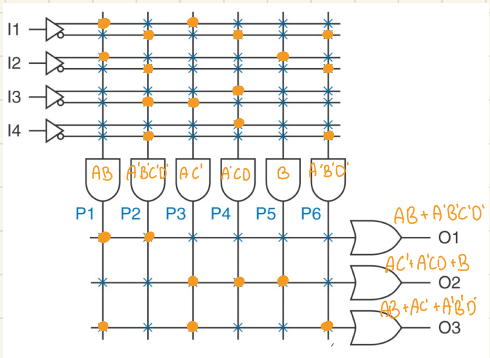


Usefull when Vin rises slowly, or fluctuating due to noise
Clean transition, Stable

☆ Useful: when output connected to state sensitive component that "activates" stuff

Combinational PLA

PLD = Programmable logic devices
PLA = Programmable logic array
PAL = Programmable array logic
CPLD = Complex PLG
FPGA = Field Programmable Gate Array



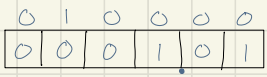
Fixed-Point Numbers

→ Small numbers / integrs

loose decimal places

n number of bits . m number of bits

→ Predetermine number of bits for front & back



The system stores the decimal

Floating-Point Numbers

$S \times R^E$

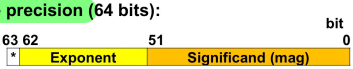
Where

- S is the *significand*, usually a signed fractional fixed-point number
- R is the *radix* (e.g. 2 for binary, 8 for octal, 10 for decimal, etc.)
- E is the *exponent*, usually a signed fixed-point integer

Example: the IEEE Standard 754 on floating-point arithmetic

Single precision (32 bits):

Double precision (64 bits):



- * is the sign bit of significand
- radix is 2
- significand is in sign-magnitude notation
- exponent is in 2's complement notation

Fixed Point

0101.1111

Floating Point

1.01111×2^{2}

1) First One is removed, as don't need to store bcuz all numbers will start with 1
2) Take Radix, 0010
3) Take first 4 bits, 0111
4) Number = 0010 0111

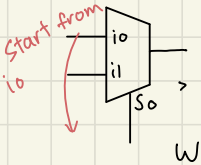
Problems of floating Point

1) It is an "estimate" or range

Combinational Circuits

– The input determine the output

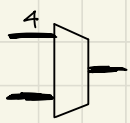
Multiplexer



when $S_0 = 0 \rightsquigarrow i0$

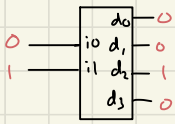
$= 1 \rightsquigarrow i1$

when line is thick



Bus of 4-bit

Decoder



Sequential Circuits

Verilog (Hardware Design Language)

We declare a module in Verilog using the **module** keyword and a list of ports:

Something like a function

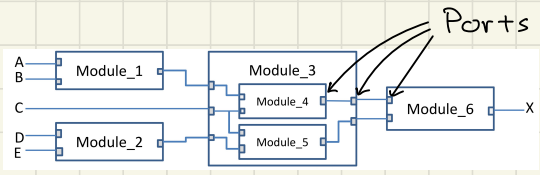
```
module somename (port1, port2, port3);
```

- The above declaration describes a module with three **ports**, each a single wire

We can indicate the direction of the ports using the keywords **input** and **output**:

```
module somename (input port1, port2,
                 output port3);
```

- The above describes a module with two inputs and one output




```
module some_name (
     input a, b, c,
     output X);

  // Describe your circuit here

endmodule
```

VL Structural Design

- This allows us to ***structurally*** describe any circuit we might otherwise use a circuit diagram for
- The principles learnt in the first half of the course allow us to describe any circuit in terms of these gates

X	Y	C_i	C_o	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

Truth table for Full Adder

$S = X \oplus Y \oplus C_i$

$C_o = X.Y + X.C_i + Y.C_i$

Minimize using Boolean Algebra or K-Maps

```
module full_adder (
      input X, Y, Ci,
      output S, Co);

  // Describe your structural
  // design for full adder here

endmodule
```

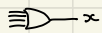
Steps

A module definition – *done*
Some gates
Wires to connect those gates

Primitives Available

and(y, a, b)

or(z, a, b, c, d)



Wires (one bit)

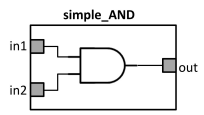
```
wire int_signal
```

Case sensitive
end with ;

For example:

A simple module that contains just a two-input ***and*** gate would be written like this:

```
module simple_AND (
       input in1, in2,
       output out);

       and (out, in1, in2);

endmodule
```



Multibit Wire

Verilog has a special construct for handling multi-bit signals (wires). **Formed by specifying a range:**

```
wire [31:0] databus;
```

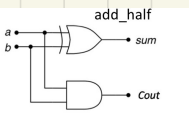
Also for specifying multi-bit module ports

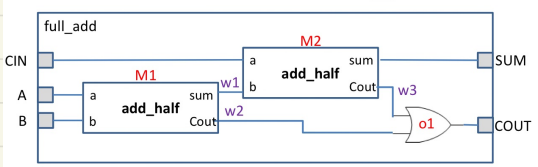
```
module add16 (input [15:0] a, b,
              output [15:0]sum,
              output cout);

    // add module internals here

endmodule
```

Module instantiation

- Something like functions

```
module add_half (input a, b,
                 output sum, Cout);
  xor g1 (sum, a, b);
  and g2 (Cout, a, b);
endmodule
```





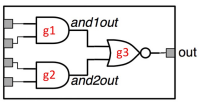

```
module full_add (input A, B, CIN,
                 output SUM, COUT);
    wire  w1, w2, w3;

    add_half M1 (A, B, w1, w2);
    add_half M2 (CIN, w1, SUM, w3);
    or o1 (COUT, w3, w2);
endmodule
```

Using it together:

Let us consider an ***and-or-inverter***

Boolean equation: out = ((a•b)+(c•d))'

- Note you can also use of a unique ***identifier*** for each gate (this can help in testing):



Gate Identifiers

```
module andorinv (input a, b, c, d,
                 output out);
    wire and1out, and2out;
    and g1 (and1out, a, b);
    and g2 (and2out, c, d);
    nor g3 (out, and1out, and2out);
endmodule
```

Using Single Assign

$x = (a\ b) + ((c\ b')\ (d + e)')'$

```
assign x = (a & b) | ~((~b & c) & ~(d | e));
```

Ordered instantiation

```
module add_half (input a, b,
                 output sum, Cout);
  xor g1 (sum, a, b);
  and g2 (Cout, a, b);
endmodule
```

```
add_half M1 (A, B, w1, w2);
```

```
add_half M2 (CIN, w1, SUM, w3);
```

Named instantiation (Better)

```
module add_half (input a, b,
                 output sum, Cout);
  xor g1 (sum, a, b);
  and g2 (Cout, a, b);
endmodule
```

```
add_half M1 (.a(A),
             .b(B),
             .sum(w1),
             .Cout(w2));
```

```
add_half M2 (.a(CIN),
             .b(w1),
             .sum(SUM),
             .Cout(w3));
```

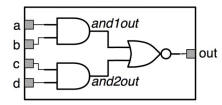
Verilog Assignments

- We can use a range of operators:
  - **&&** : and (bitwise is: **&**)
  - **||** : or (bitwise is: **|**)
  - **!** : not (bitwise is: **~**)
  - **^** : xor (bitwise is: **^**)

Operator	Name
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~&	Bitwise NAND
~|	Bitwise NOR

a 01001
b 11010
a&B 01000

- The and-or-inverter example previously shown:
  - The ***assign statement*** allows the use of more complex operators and operands – more on this later



Instantiating gate primitives

```
wire and1out, and2out;

nor g3 (out, and1out, and2out);
and g2 (and2out, c, d);
and g1 (and1out, a, b);
```

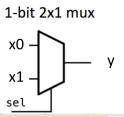
```
assign out = ~((a&b) | (c&d));
```

Continuous assignment

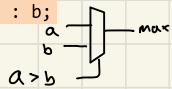
Conditional Assignment

```
assign y = sel ? x1 : x0; // a multiplexer
```

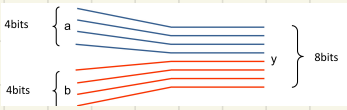
The signal **y** will be connected to **x1** **if sel is 1, else** it is connected to **x0**




```
assign max = (a>b) ? a : b;
```



Concatenation




```
wire [3:0] a, b;
wire [7:0] y;
...
assign y = {a, b};
```

Replication

assign z = {4{c}} → cccc

Literal Assign

- <size>'<radix><value>
  - <size> is the width **in bits**
  - <radix>: b for binary, o for octal, h for hex, d for decimal
  - <value>: the number you want, with as many optional underscores as needed (for readability)

Examples:

- 4'b0000 (4 binary bits "0 0 0 0")
- 8'h4F (= 8'b01001111)
- 8'b0100_1111 (Same as above. Note the use of the underscore)
- 1'b1 (a single "1")

→ assigning literal to

1) Larger bus, MSB padded with 0

2) Smaller bus, MSB cut off

Paremeter

```
module some_mod #(parameter SIZE=8, WIDTH=16)(
  input  [SIZE-1:0] X, Y,
  output [WIDTH-1:0] Z);
```

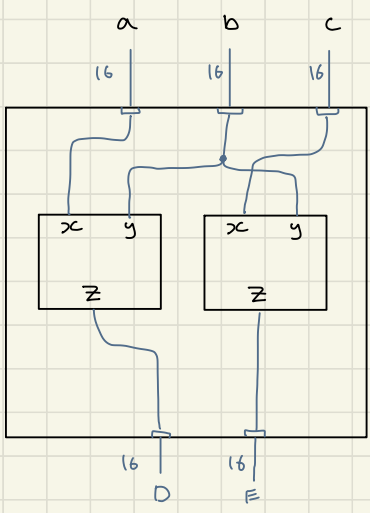
- It is also possible to redefine a **parameter**
  - Consider the following 8-bit module

```
module submod #(parameter SIZE=8) (
  input  [SIZE-1:0] X, Y, output [SIZE-1:0] Z);
  // some statements in here
endmodule
```

- Then, note the change in the parameter in the submodule instantiation below — redefine
  - Now using two 16-bit submodules

```
module top_mod #(parameter SIZE=16) (
  input  [SIZE-1:0] a, b, c,  output [SIZE-1:0] D, E);

  submod #(.SIZE(SIZE)) U1 (.X(a), .Y(b), .Z(D));
  submod #(.SIZE(SIZE)) U2 (.X(c), .Y(b), .Z(E));

endmodule
```

In practice, it may be better to call them different names to avoid confusion



```
module adder #(parameter SIZE=32)(
        input Cin,
        input [SIZE-1:0] A, B,
        output Cout,
        output [SIZE-1:0] Sum);

        assign {Cout, Sum} = A+B+Cin;

endmodule
```

Note: 1. How the concatenation operator handles the 33-bit result produced by A+B+Cin
2. The use of the arithmetic addition operator (+) to generate the sum.

Behavioural Modeling

Combinational Always Block

```
always @ (a, b)
  begin
    x = a & b;
    y = a | b;
  end
```

Sensitivity List

Procedural statements

The **always** keyword starts a block

The *sensitivity list* **must** contain the names of any signals that affect the output of the block

always @ *

Just use *

Reg (Behaves like variable in code)

Signals you assign to from within an ***always*** *block*

- Must be declared as being of type: **reg**

reg is synonymous with wire, but these signals can be assigned to from inside an always block, **wire**s cannot!

A **wire** is simply a *connection*, it holds no value of its own

The **reg** type is more like a *variable* in programming, as we will see

Note: you *cannot* assign to a **reg** signal using an ***assign*** *statement*, or connect it to module instance outputs.

```
module temp (input a, b,
             output out);
  wire w1;

  assign w1 = a & b;
  assign out = w1;

endmodule
```

```
module temp (input a, b,
             output out);
  reg w1;

  always @ *
    w1 = a & b;

  assign out = w1;
endmodule
```

```
module temp (input a, b,
             output reg out);
  reg w1;

  always @ *
    w1 = a & b;
    out = w1
  assign out = w1;
endmodule
```

If internal assignment

Declare **reg** signals:

- You can set an *initial value* when you declare **reg** signals

```
reg a, b;
reg [3:0] x = 4'b0000;
```

You can also declare your module outputs as **reg** if you plan to assign to them directly from within an **always** block:

```
module sel_one (input [5:0] a, b, c, d,
                input [1:0] sel,
                output reg [5:0] sigsel);
```

If Else

```
always @ *
begin
  if (alarm == 1'b1)
    begin
      if (after_hours)
        siren = 1'b0;
      else
        siren = 1'b1;
      light = 1'b1;
    end
  else
    begin
      siren = 1'b0;
      light = 1'b0;
    end
end
```

If only (if else), no need begin & end

Case Statement

```
always @ *
case (sel)
    2'b00   : y = a;
    2'b01   : y = b;
    2'b10   : y = c;
    default : y = 4'b1010;
endcase
```

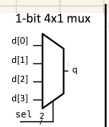
Default for any exception

```
always @ *
case (sel)
    2'd0    : y = a;
    2'd1    : begin
                y = b;
                z = c;
              end
    2'd2    : y = c;
    default : y = 4'b1010;
endcase
```

There is no need for a break in each branch (unlike some software languages)

```
module mux4 (output reg  q,
             input [3:0] d,
             input [1:0] sel);

  always @ * begin
     case (sel)
        2'b00 : q = d[0];
        2'b01 : q = d[1];
        2'b10 : q = d[2];
        2'b11 : q = d[3];
     endcase
  end
endmodule
```

1-bit 4x1 mux



Note the begin and end. These are not needed as a case is a single statement. But makes it easier to read!!

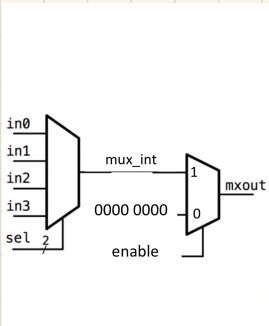
- Example: 3-to-8 decoder:

Input	Output
000 →	00000001
001 →	00000010
010 →	00000100
011 →	00001000
...	
111 →	10000000

Reg!!

```
module decoder3_8(output reg [7:0] d_out,
                  input  [2:0] ival);
always @ *
  case(ival)
     3'b000  :  d_out = 8'b00000001;
     3'b001  :  d_out = 8'b00000010;
     3'b010  :  d_out = 8'b00000100;
     3'b011  :  d_out = 8'b00001000;
     3'b100  :  d_out = 8'b00010000;
     3'b101  :  d_out = 8'b00100000;
     3'b110  :  d_out = 8'b01000000;
     3'b111  :  d_out = 8'b10000000;
  endcase
endmodule
```

Example

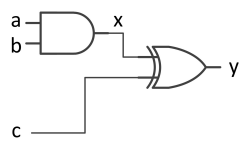



```
module mux_4_32(output [7:0] mxout,
                input  [7:0] in3, in2,
                input  [7:0] in1, in0,
                input  [1:0] sel,
                input        enable);
  reg [7:0] mux_int;
  assign mxout = enable ? mux_int : 8'd0;
  always @ *
    begin
      case(sel)
        2'b00   :  mux_int = in0;
        2'b01   :  mux_int = in1;
        2'b10   :  mux_int = in2;
        2'b11   :  mux_int = in3;
      endcase
    end
endmodule
```

always assign all values, or else the loop will "store" a value and it will no longer be a combi circuit as output not 100% reliant on inputs provided

Statement Order

```
always @ *
begin
     x = a & b;
     y = x | c;
     y = x ^ c;
end
```



Cancelled: Last is prioritized.

Latch

One way to fix this is to use a default assignment at the top of the **always** block:

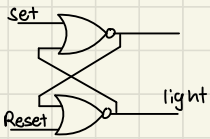
```
always @*
begin
   y = x;
   if(valid) begin
      c = a | b;
      y = z;
   end
   else
      c = a;
end
```

The default is overwritten by any subsequent assignment
Assignments at the top of an **always** block are hence a great way of remembering to assign to all your signals

Sequential Circuits

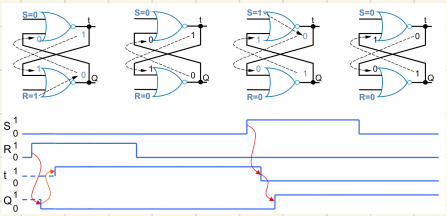
– Can store information

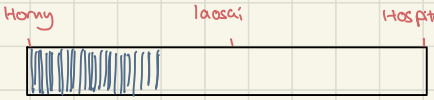
SR Latch



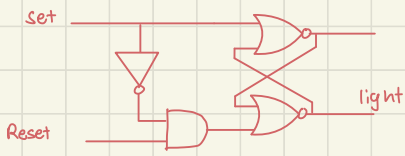
– Most basic to store

S	R	Light
1	0	1
0	1	0
0	0	store



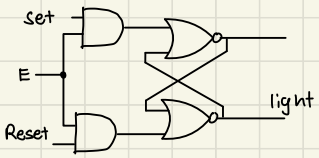


☆ If both asserted & de-asserted at the same time, the circuit will oscillate and flicker



↳ However these inputs will have a propogation delay as the set inverts & the AND output turns to zero

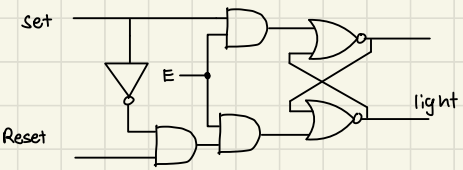
Enable SR Latch (call button)



- Only works when E = 1

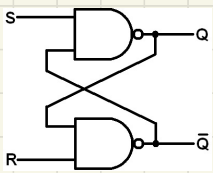
E	S	R	Q+	Function
0	X	X	Q	Store
1	0	0	Q	Store
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	?	Undefined

Gate Enable / Level sensitive SR Latch (Enable call button)

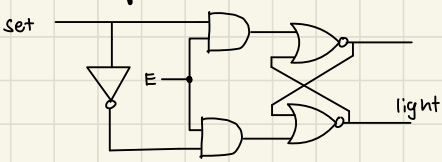


- Only works when E = 1

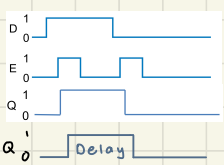
E	S	R	Q+	Function
0	X	X	Q	Store
1	0	0	Q	Store
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	?	Undefined



Transparent Enabled D-latch ("storage" w. enable)



E	D	Q+
0	X	Q
1	0	0
1	1	1



Cock (∇) Clock

Frequency ⇝ f (gHz, mHz)

Period ⇝ One cycle $f = \frac{1}{p}$

Example: If clock ⇝ 5 kHz,

Period = $\frac{1}{5\times10^{3}} = 2\times10^{-4}$

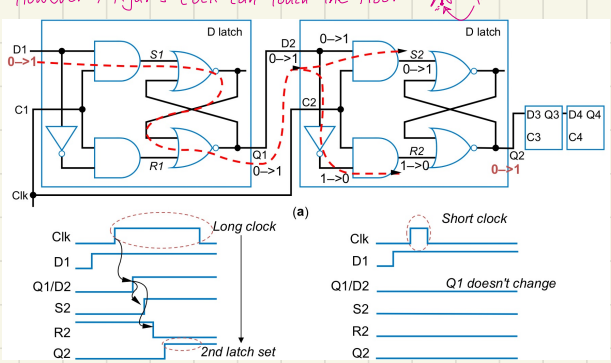
mili	$m = 1\times10^{-3}$
micro	$\mu = 1\times10^{-6}$
nano	$n = 1\times10^{-9}$

Long Cock vs Short Coc

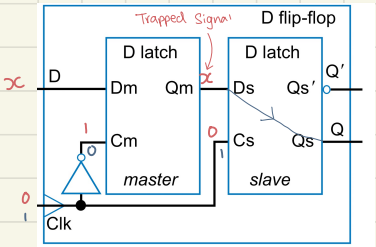
The long cock can penetrate circuit deeper

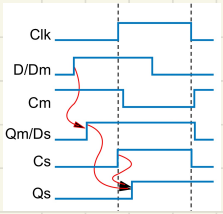
The short cock cannot, fails to turn on the circuitK

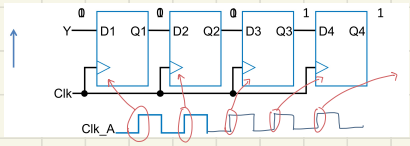
However, Ryan's cock can touch the floor



Edge Triggered D Flip Flop

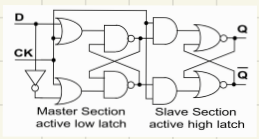






★ Positive edge trigger

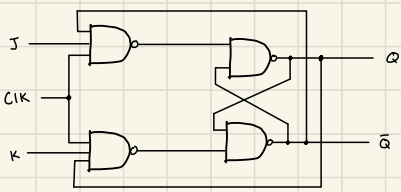
Negative edge trigger

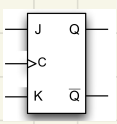


Active Low to eliminate propagation when ▷o

↳ Basically only activate when edge trigger, store that value

JK Flip Flop





CLK	J	K	Q(t+1)
↑	0	0	Q
↑	0	1	0
↑	1	0	1
↑	1	1	Q'
	X	X	Q

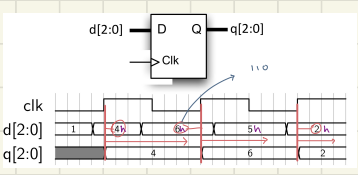
When both 0, Q stored

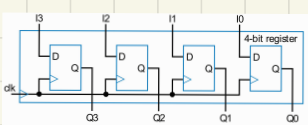
01 , Q=0

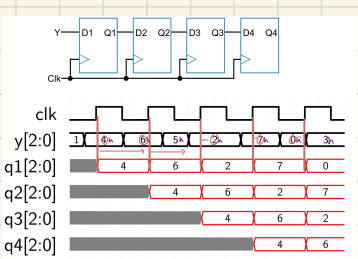
10 , Q=1

11 , Q=flip over

Register

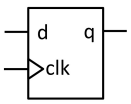






Sequential Verilog

Flip flops



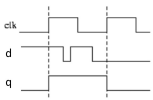

```
module simplereg (input d, clk,
                  output reg q);

always@(posedge clk)
    q <= d;
endmodule
```

inside assignment

Special

for -ve, use negedge

non-blocking assignment
- Order does not matter
- Imp. for syncro. always block

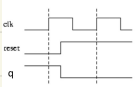


- This creates a 1-bit register/D flip-flop with input *d* and output *q*

Reset Asynchronous / Syncronous

- Reset anytime

- For an *asynchronous* reset, we would need to add the reset signal to the sensitivity list:

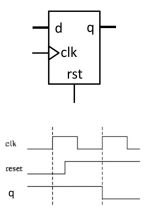



```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk or posedge rst)
begin
    if(rst)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

rst go from 0 to 1

- Reset on clock ↑↓

- A register with *synchronous* reset:

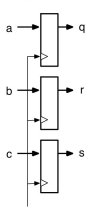



```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk)
begin
    if(rst) // same as (rst==1'b1)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

If active low, if (!rst)

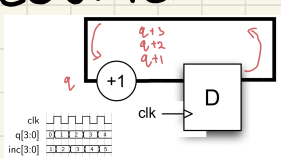
Registers

- We can create multiple registers by including multiple assignments
- Each *assignment* in a synchronous always block results in a *register*




```
module multireg (input [7:0] a, b, c,
                 input clk, rst,
                 output reg [7:0] q, r, s);

always@(posedge clk)
begin
    if(!rst) begin
        q <= 8'b0000_0000;
        r <= 8'b0000_0000;
        s <= 8'b0000_0000;
    end else begin
        q <= a;
        r <= b;
        s <= c;
    end
end

endmodule
```

Counter

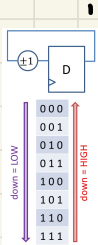



```
module simplecnt (input clk, rst,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        q <= q + 1'b1;
end

endmodule
```

At each rising edge, we pass through the incremented value of the current count

The data width can be any number of bits

UP & Down Counter



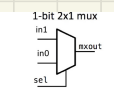

```
module simplecnt (input clk, rst, down,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        if(down)
            q <= q - 1'b1;
        else
            q <= q + 1'b1;
end

endmodule
```

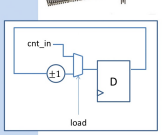
Input time

Synchronous Counters in Verilog

```
module simplecnt (input clk, rst,
                  input down, load,
                  input [3:0] cnt_in,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        if(load)
            q <= cnt_in;
        else
            if(down)
                q <= q - 1'b1;
            else
                q <= q + 1'b1;
end
endmodule
```

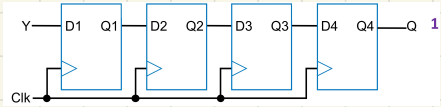
1-bit 2x1 mux





Note: Synthesis tool may not instantiate a multiplexer

Shift Register

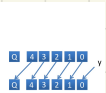



```
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```

```
module shiftreg (input  clk, y,
                 output reg q_out);

reg [3:1] q;

always@(posedge clk)
begin
    q[1]   <= y;
    q[3:2] <= q[2:1];
    q_out  <= q[3];
end

endmodule
```

For many reg

```
module shiftreg (input clk, y,
                 output reg Q);

reg [4:0] q;

always@(posedge clk)
begin
    q[0]   <= y;
    q[4:1] <= q[3:0];
    Q      <= q[4];
end

endmodule
```




```
module shiftreg (input clk, y,
                 output q_out);

reg [4:0] q;

always@(posedge clk)
begin
    q[0]   <= y;
    q[4:1] <= q[3:0];
end

assign q_out = q[4];

endmodule
```

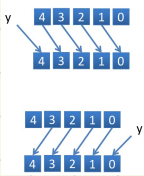
Shift on command

```
module shiftreg2 (input clk, y, sh,
                  output q_out);

reg [4:0] q;

always@(posedge clk)
begin
    if (sh) begin
        q[0]   <= y;
        q[4:1] <= q[3:0];
    end
end

assign q_out = q[4];

endmodule
```

Direction, Output word, Input Word

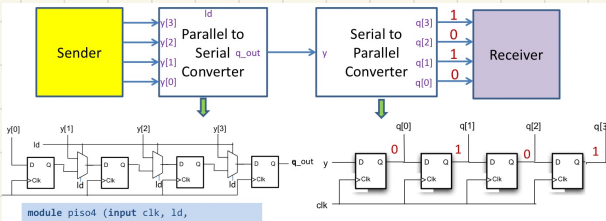
We can also add the capability to shift in the other direction

When the *rt* input is high, the new sample is registered at the MSB, and the rest shift right




```
module shiftreg5 (input clk, y, sh, rt, ld,
                  input [4:0] ld_val,
                  output q_out,
                  output [4:0] q_word);
reg [4:0] q;

always@(posedge clk)          load value in
begin                                  Reg
    if (ld) q <= ld_val; else
    if (sh) begin
        if (rt) begin
            q[4] <= y; q[3:0] <= q[4:1];
        end else begin
            q[0] <= y; q[4:1] <= q[3:0];
        end
    end
end
                          Choose Out
assign q_out = rt ? q[0] : q[4];
assign q_word = q;      output all

endmodule
```

Sender to Receiver



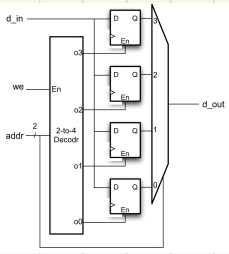

```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

```
module sipo4 (input clk, y,
              output reg [3:0] q);
always@(posedge clk)
begin
    q[0]   <= y;
    q[3:1] <= q[2:0];
end

endmodule
```

Memory

In order to store a value, we must assert *we* (write enable) and provide an address

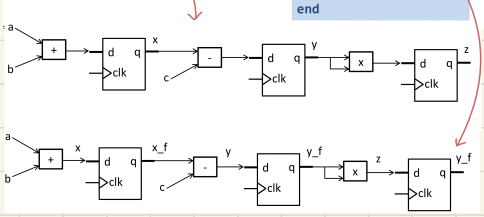
The decoder outputs a 1 to enable the corresponding register to store whatever is on *d_in* at the clock edge



- Some prefer to code with the combinational and synchronous aspects separated

```
always@(posedge clk)
begin
    x <= a + b;
    y <= x - c;
    z <= y * y;
end
```

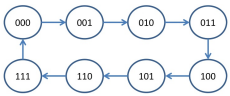
```
always@(posedge clk)
begin
    x_f <= x;
    y_f <= y;
    z_f <= z;
end

always@*
begin
    x = a + b;
    y = x_f - c;
    z = y_f * y_f;
end
```



States

Sequences and States

- We can use a figure to show these transitions:



0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

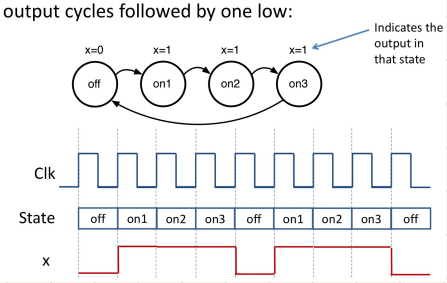
This is a called *state transition diagram*

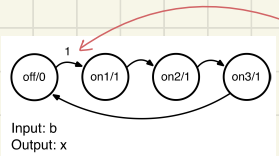
- Each node is a possible *state* of the system
- It shows the movement from one state to the next
- In this case, the states are simply labeled with their output values

Finite State Machines

Example

An always-on state machine that produces three high output cycles followed by one low:





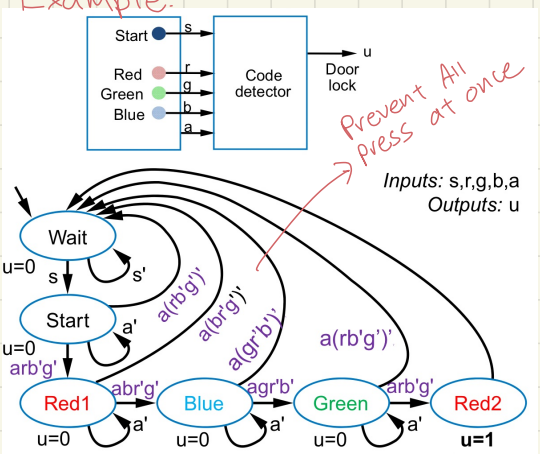
Can use a ledgend. here, only when b=1, it can "exit" off.

write the FSM information in a state transition table

Current State	Next State b=0	Next State b=1	Output
off	off	on1	0
on1	on2	on2	1
on2	on3	on3	1
on3	off	off	1

Current State	b	Next State	Output x
off	0	off	0
off	1	on1	0
on1	0	on2	1
on1	1	on2	1
on2	0	on3	1
on2	1	on3	1
on3	0	off	1
on3	1	off	1

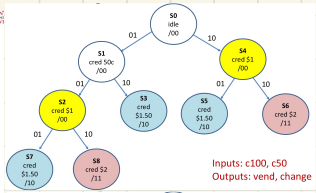
Example:

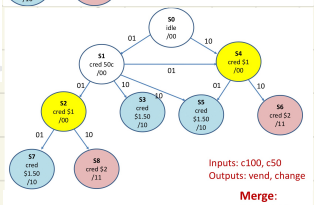


Mealy Machines

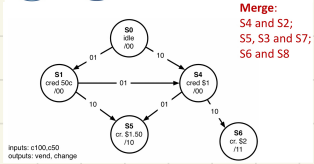
Another Example

- We want to design an FSM for a vending machine, that accepts only $1 and 50c coins, dispensing a drink that costs $1.50, and change if necessary
- This can be a circuit with two inputs, c100, for a $1 coin inserted and c50 for a 50c coin
- We assume only 1 input is ever high, and it tells us what coin was inserted
- The FSM has two outputs, vend, to release a drink, and change, to give 50c of change



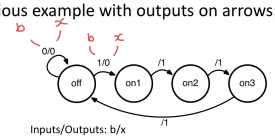


Merge:
S4 and S2;
S5, S3 and S7;
S6 and S8



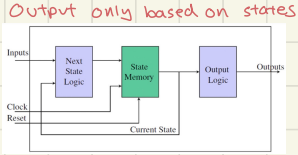
Mealy Machines

- For Mealy machines, we can't draw the outputs inside the state circles, so we add them to the arrows instead, after a slash
- The output written on an arrow tells us what the output should be during the emitting state, when the input values match those on the same arrow
- A previous example with outputs on arrows:



Inputs/Outputs: b/x

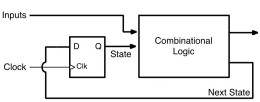
This is moore:
Output only based on states



Basic Struc For Vlog

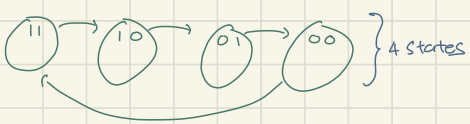
Mapping to Implementation

- Hence, the basic structure looks like this:



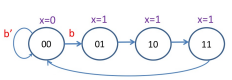
- The combinational logic the following based on the current state and inputs
  - Next State
  - Outputs
- We must turn the states into some binary representation, then we can use those values and the inputs to write the logic to produce the next state and outputs

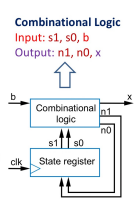
How many states are needed?



For m states, $\log_2 m$ bits needed

- We now have a value of s1, s0 for each state
- We can use this to build a table that captures the transitions to ns1, ns0, and the output

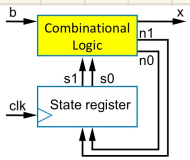




Current State: **s[1:0]**
Next State: **n[1:0]**

- Now determine the combinational logic:

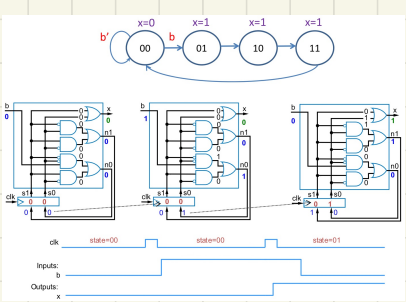
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
off	0	0	0	0	0	0
	0	0	1	0	0	1
on1	0	1	0	1	1	0
	0	1	1	1	1	0
on2	1	0	0	1	1	1
	1	0	1	1	1	1
on3	1	1	0	1	0	0
	1	1	1	1	0	0



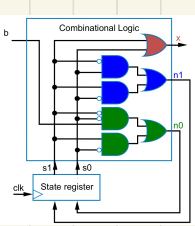
x = s1 + s0
(note that x=1 if s1=1 or s0=1)

n1 = s1' s0b' + s1' s0b + s1s0' b' + s1s0' b
n1 = s1' s0 + s1s0'

n0 = s1' s0' b + s1s0' b' + s1s0' b
n0 = s1' s0' b + s1s0'



x = s1 + s0
n1 = s1' s0 + s1s0'
n0 = s1' s0' b + s1s0'



```
module pulse3 (input b,
               input clk, rst,
               output x);

wire n1, n0; – next state
reg  s1, s0; – current state

assign n1 = s1 ^ s0;
assign n0 = (~s1&~s0&b) | (s1&~s0);
assign x = s1 | s0;

always @ (posedge clk)
begin
    if(rst) begin
       s1 <= 1'b0;
       s0 <= 1'b0;
    end else begin
       s1 <= n1;
       s0 <= n0;
    end
end
endmodule
```

State Register
the assign on clock part

Parameter

Verilog Constants

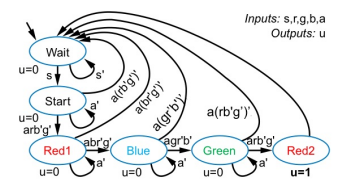
- First we need to encode the states:
  - Wait = 000, Start = 001, Red1 = 010, Blue = 011, Green = 100, Red2 = 101
- As we assign each state a binary value, we'd still rather not deal with binary numbers in our Verilog code.
- We can use the Verilog parameter keyword to declare named constants:

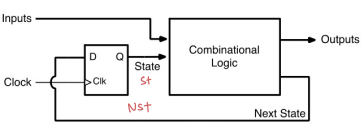
```
parameter st1 = 2'b00, st2 = 2'b01, st3 = 2'b10;
```

- We can then use these names in place of the values anywhere in the code.

```
nst = st2;
```

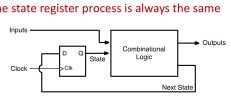
A More Complex Example






```
module seqdet (input s, r, g, b, a,
               input clk, rst,
               output u);

parameter waite=3'b000, start=3'b001, red1=3'b010,
blue=3'b011, green=3'b100, red2=3'b101;

reg [2:0] nst, st;

assign u = (st == red2);          → Logic to drive output

always @ (posedge clk)
begin
    if(rst)
        st <= waite;
    else
        st <= nst;
end                               } State register

always @ *
begin
    nst = st;   If a', will maintain @ itself if not change
    case(st)
        waite: if(s) nst = start;
        start: if(a)
                  if(r&~b&~g) nst = red1;
                  else nst = waite;
        red1: if(a)
                  if(b&~r&~g) nst = blue;
                  else nst = waite;
        blue: if(a)
                  if(g&~r&~b) nst = green;
                  else nst = waite;
        green: if(a)
                   if(r&~g&~b) nst = red2;
                   else nst = waite;
        red2:  nst = waite;

        default: nst = waite;
    endcase
end                               } Logic to determine next state
endmodule
```

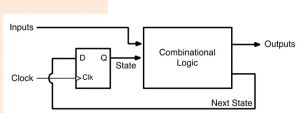
- On the previous slide:
  - We declare two 3-bit reg signals (since both will be assigned from always blocks)
  - The output is high only when the current state is Red2 (101), so we can assign it directly
  - The state register process is always the same



- We still need to describe the combinational logic to determine the next state (based on current state and inputs)

- So the generalized FSM structure looks like this:

```
module seqdet (input /* FSM inputs */
               input clk, rst,
               output /* FSM outputs */);

parameter st1 = … /* state names and assignments */

reg [2:0] nst, st;  /* next and current state signal
*/

/* state register always block
   always the same structure
   st <= nst, with a reset to intial */

/* state transition always block
   generally uses a case statement
   assigning to nst based on st and inputs */

/* output assignment can be separate using
   assign or always block, or combined in
   state transition always block */
endmodule
```



Recap (Structural Design)

Ports

```
module (input in1, in2,
        output out3);
endmodule
```

Wires

- wire
  - `wire thing;`
  - `wire [3:0] that;`
  - `assign thing = 1'b0`
- reg
  - `reg what;`
  - `reg [3:0] whatte;`
  - → only can be assigned from always @ * block

Gates

```
and a1 (out3, in1, in2);
xor xo1 (out4, in3, in4);
```

output first

- also, cannot assign to same output

Wire manipulation

`assing thing = in1[1]`

Specific bit selection

Ordered instantiation

```
module top (input [1:0] sel,
            input full,
            output [3:0] queue);
decoder2to4 (.a(sel), .enable(~full), .one-hot(queue));
```

Assign Statements

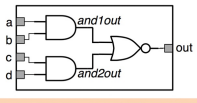
- We can use a range of operators:
  - **&&** : and (bitwise is: **&**)
  - **||** : or (bitwise is: **|**)
  - **!** : not (bitwise is: **~**)
  - **^** : xor (bitwise is: **^**)

Operator	Name
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~&	Bitwise NAND
~|	Bitwise NOR

Imp

not imp

- The and-or-inverter example previously shown:
  - The ***assign statement*** allows the use of more complex operators and operands – more on this later



Instantiating gate primitives

```
wire and1out, and2out;
nor g3 (out, and1out, and2out);
and g2 (and2out, c, d);
and g1 (and1out, a, b);
```

```
assign out = ~((a&b) | (c&d));
```

Continuous assignment

`assign y = x ? a : b;`

If x = 1, y = a

x = 0, y = b

Recap (Behavioural Design)

Combinational Always Block

Basic

```
always @ *
begin
    x = a & b
    y = x ^ c
end
```

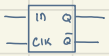
Case

```
always @ *
  begin
  case (x)
    2'b00 : q = d[0];
    2'b01 : q = d[1];
    2'd2  : q = d[2];
    2'd3  : q = d[3];
  endcase
  end
```

must always apply for ALL CASES or else a LATCH is produced.

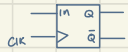
Latches ∴ sequential Circuits

Latch



When clock = 1 (or 0, if -ve trigger)
Q exactly matches input

Flip-Flop



On posedge clk (or negedge clk)
Q will copy in. **Once**

```
reg q1, q2, q3
always @ (posedge clk)
begin
  if (rst)
    begin
      q1 <= 1'b0
      q2 <= 1'b0
      q3 <= 1'b0
    end
  else
    begin
      q1 <= y
      q2 <= q1
      q3 <= q2
    end
end
```

Using non-blocking assignment for clk