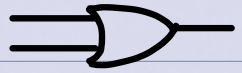
Lecture 1 - 8th Jan

AND:  OR:  NOT: 

Lecture 2: 9th Jan

Decimal to Binary Conversion:

↳ $23_{10} = (?)_2$.

Algorithm: keep dividing by radix until quotient = 0.

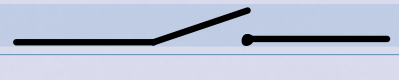
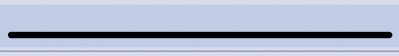
↳

	quotient	remainder
$23 \div 2$	11	1 (least significant bit)
$11 \div 2$	5	1
$5 \div 2$	2	1
$2 \div 2$	1	0
$1 \div 2$	0	1 (most significant bit)

$\therefore 23_{10} = (1\ 0\ 1\ 1\ 1)_2$

Binary Logic

- two-valued logic → eg. true/false, on/off, etc.
- binary logic variables → can take on two discreet values 0 or 1.
  - ↳ 0 is false/closed/off, 1 is true/open/on.

- false / closed / off :  (0)
- true / open / on :  (1)

Binary Logic Functions

↳ variables can only have values of 0 or 1

- Expressions of binary variables or other functions.
- Output a value of 0 or 1.
- Example: $f = x + y$
- Functions can be defined using truth tables.

Logic Operators

- needed to make logic functions and manipulate them
- Basic operators are AND, OR, and NOT.

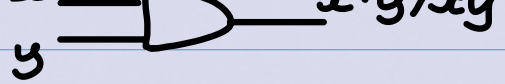
↓

Symbols for Logic Operators:

- AND : $\cdot$ → $x \cdot y$ or simply xy
- OR : $+$ → $x + y$
- NOT : $'$ or $\bar{}$ → $\bar{x}$ or x'

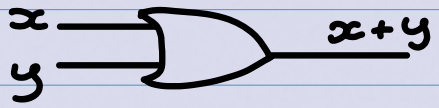
- Operator Precedence: () → NOT → AND → OR

Logic Gates, their Symbols, and Truth Tables

- AND: 

x	y	x·y / xy
0	0	0
0	1	0
1	0	0
1	1	1

• OR:



x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

• NOT: x —▷○— $\bar{x}/x'$

x	$\bar{x}/x'$
0	1
1	0

Representation of Logic Functions

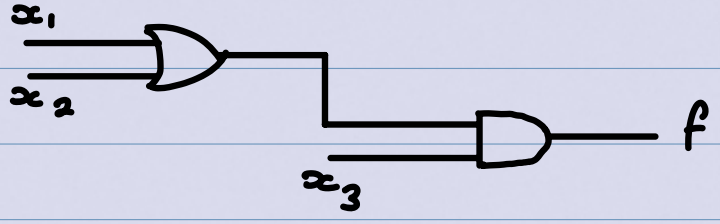
• Algebraic:

↳ example: $f = (x_1 + x_2) \cdot x_3$

truth table:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

• schematic:



Lecture 3: 10th Jan

Boolean Algebra: defined by a set of elements, B, together with two binary operations + and · that satisfy the following postulates:

↓

- P1: $x+y \in B$ and $x \cdot y \in B$
- P2: $x + 0 = x$ and $x \cdot 1 = 1$
- P3: $x + y = y + x$ and $x \cdot y = y \cdot x$
- P4: $x \cdot (y+z) = x \cdot y + x \cdot z$ and $x + y \cdot z = (x+y) \cdot (x+z)$

- P5: $\forall a \in B, \exists a' \in B$ s.t. $a + a' = 1$ and $a \cdot a' = 0$.
- P6: $\exists a, b \in B$ s.t. $a \neq b$.

take $B = \{0, 1\}$, and all postulates hold!

Synthesis: the implementation of a circuit.

Cost of a circuit: # gates + # inputs to the gates

- Using boolean algebra, a function can be manipulated leading to multiple designs for the same function. The costs can vary.

Minterms and Maxterms

- For a function of n variables, a (logical) product term in which each of the n variables appears once is called a minterm

- For a function of n variables, a (logical) sum term in which each of the n variables appears once is called a minterm

↳ denoted by lowercase m

Examples: $x_1' x_2 x_3 x_4$ → minterm

$x_1 x_2' x_3$ → product

↳ denoted by uppercase M

$x_1 + x_2 + x_3 + x_4$ → Maxterm

$x_1 + x_3$ → sum

For a particular input pattern, its associated minterm is 1 while all other minterms evaluate to 0, and its associated maxterm is 0 while all other maxterms evaluate to 1.

- Minterms and maxterms are dual to each other. (↗ 1s become 0s and ANDs become ORs.)

Canonical Sum of Products

↳ given a truth table, it's always possible to write a logic expression for the function by taking logical OR of the minterms for which the function outputs true (1).

Note: the expression must have minterms only to be a canonical SOP.

Example: write the canonical SOP for the following truth table:

	x_1	x_2	x_3	f
0	0	0	0	0
1	0	0	1	1 → $m_1 = x_1' x_2' x_3$
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1 → $m_4 = x_1 x_2' x_3'$
5	1	0	1	1 → $m_5 = x_1 x_2' x_3'$
6	1	1	0	1 → $m_6 = x_1 x_2 x_3'$
7	1	1	1	0

$\therefore f = m_1 + m_4 + m_5 + m_6$

$\therefore f = x_1' x_2' x_3 + x_1 x_2' x_3' + x_1 x_2' x_3' + x_1 x_2 x_3'$

Canonical Product of Sums

↳ given a truth table, it's always possible to write a logic expression for the function by taking logical AND of the maxterms for which the function outputs false (0).

Note: the expression must have maxterms only to be a canonical POS.

Example: write the canonical POS for the following truth table:

	x_1	x_2	x_3	f
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$\therefore f = M_0 \cdot M_2 \cdot M_3 \cdot M_7$

$M_0 = x_1 + x_2 + x_3$

$M_2 = x_1 + x_2' + x_3$

$M_3 = x_1 + x_2' + x_3$

$M_7 = x_1' + x_2' + x_3'$

$\therefore f = (x_1 + x_2 + x_3) \cdot (x_1 + x_2' + x_3) \cdot (x_1 + x_2' + x_3) \cdot (x_1' + x_2' + x_3')$

- Canonical SOP/POS is very often not the lowest cost expression, but they can usually be simplified to a better one!

↳ For example, the above canonical POS can be simplified as follows:

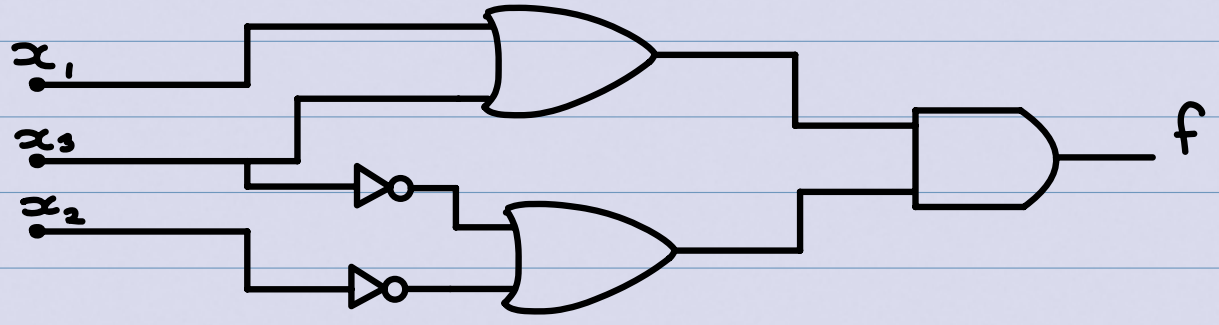
↳ $(x_1 + x_2 + x_3) \cdot (x_1 + x_2' + x_3) \cdot (x_1 + x_2' + x_3) \cdot (x_1' + x_2' + x_3')$

$= (x_1 + x_3) \cdot (x_2' + x_3')$ (using rule 11b).

↳ This has a lower cost than canonical POS!

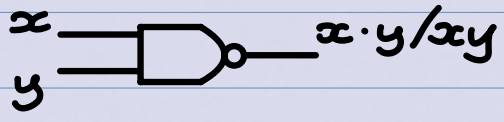
We can the implement this into a schematic:

$(x_1 + x_3) \cdot (x_2' + x_3')$:



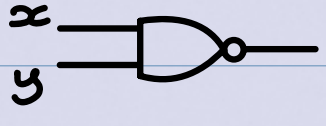
Synthesis with NAND and NOR:

- **NAND:**

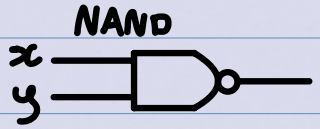


x	y	x·y /xy
0	0	1
0	1	1
1	0	1
1	1	0

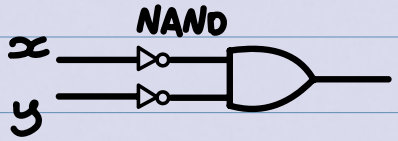
- **NOR:**



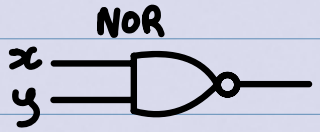
x	y	(x+y)'
0	0	1
0	1	0
1	0	0
1	1	0



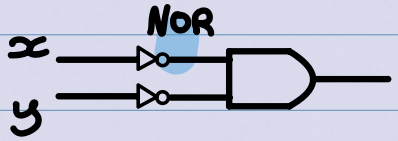
is the same as



↳ $\overline{(xy)} = \bar{x} + \bar{y}$!



is the same as



↳ $\overline{(x + y)} = \bar{x}\bar{y}$!

Gray Code → a binary numeral system where two successive values differ in only one bit. It's also known as the reflected binary code.

1-bit Gray Code:

0
1

2-bit Gray Code:

0	0
0	1
1	1
1	0

3-bit Gray Code:

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

Given n-bit gray code, it's easy to write the (n+1)-bit Gray code.

↳ Example: let n=2. The (2+1)-bit Gray code can be attained as follows:

1) write the two bit gray code with a 0 in front (in red)
2) write the two bit gray code in reverse order with a 1 in front (in blue)

↳

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

Gray Code Based Simplification

→ Example :

x_1	x_2	f	
0	0	1	if the function had only these two 1s, then $f = x_1'$ since x_2 has no effect when $x_1 = 0$.
0	1	1	
1	0	0	this 1 implies that f should also have $x_1 x_2$!
1	1	0	

$\therefore f = x_1' + x_1 x_2$ covers all 3 1s for the function, and is simpler than the canonical SOP.

Example :

x_1	x_2	f	
0	0	1	if the function had these two 1s only, then $f = x_1'$ since x_2 has no effect when $x_1 = 0$.
0	1	1	
1	0	1	if the function had these two 1s only, then $f = x_2$ since x_1 has no effect when $x_2 = 1$.
1	1	0	

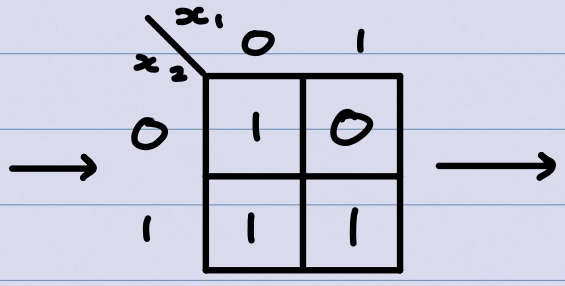
Karnaugh (K) Maps

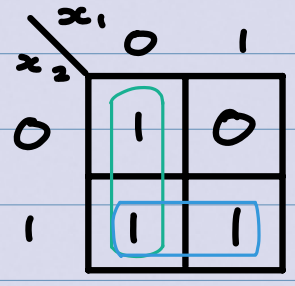
- Simplification of logic expressions using boolean algebra is not very systematic. An alternative approach is using K-maps.
- K-maps only work for functions of upto 5 variables!
- Alternative representation of a truth table!
- K-maps use the gray code!

- While a truth table is tabular, the K-map is arranged as a grid of cells.
- The "coordinates" of a cell represents the function's input values.
- The content of any cell is the function's output value for the corresponding inputs.
- The rows and columns of a K-map are labelled in a way so that 1 variable changes between adjacent rows and columns (gray code)

K-map for 2 variables example:

x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1





for the green rectangle, f is 1 when x_1 is 0, regardless of what x_2 is.

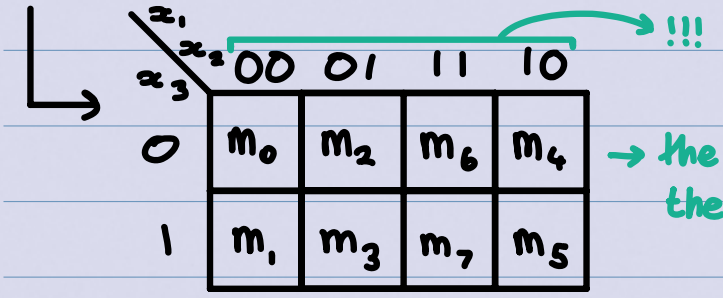
↳ ∴, this term would be x_1'.

for the blue rectangle, f is 1 when x_2 is 1, regardless of what x_1 is.

↳ ∴, this term would be x_2.

↳ ∴, $f = x_1' + x_2$

When setting up a K-map, only one variable can change between each row/column (gray code)



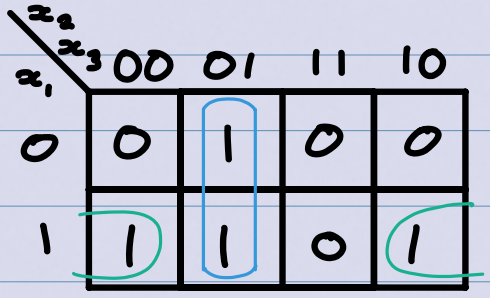
In general, for an n-variable function a rectangle encompassing 1s is in:

- 1 cell corresponds to a product term of n variables (ie, minterm)
- 2 adjacent cells correspond to a product of (n-1) variables
- 4 (ie 2^2) adjacent cells correspond to a product of (n-2) variables
- 8 (ie 2^3) adjacent cells correspond to a product of (n-3) variables.

Example:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

→



↳ ∴, $f = x_2' x_3 + x_1 x_3'$

- Columns and rows wrap around from left to right and top to bottom.

K-map locations of minterms for 4-variable functions:

x_3x_4 \ x_1x_2	00	01	11	10
00	m_0	m_4	m_{12}	m_8
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

Example: simplify $\sum m(2,3,9,10,11,13)$ using K-maps:

x_3x_4 \ x_1x_2	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	1	0	0	1
10	1	0	0	1

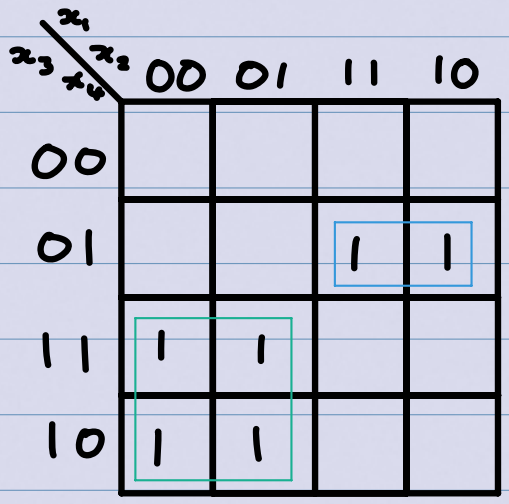
→ $f = x_1 x_3' x_4 + x_2' x_3$

Example: simplify $\sum m(0,2,3,6,7,8,10,15)$ using K-maps:

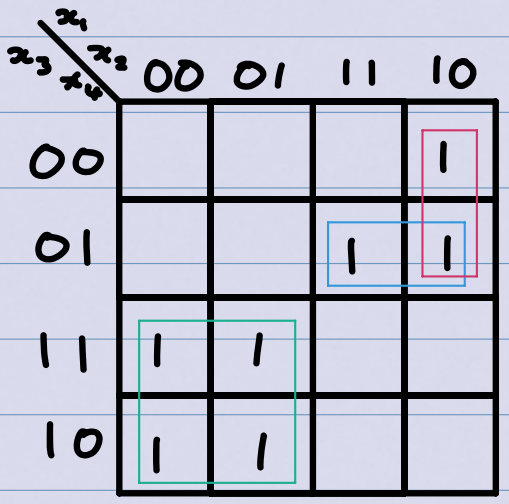
x_3x_4 \ x_1x_2	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	1	0
10	1	1	0	1

→ $f = x_2 x_3 x_4 + x_1' x_3 + x_2' x_4'$

Example K-map for 5-variable functions:



$x_5 = 0$



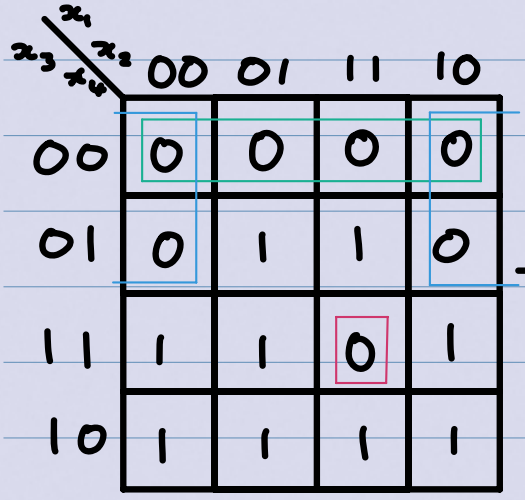
$x_5 = 1$

$$f = x_1'x_3 + x_1x_3'x_4 + x_1x_2'x_3'x_5$$

Minimization of POS forms:

- Very similar to sum-of-products minimization. The differences are:
  - ↳ we try to encompass the 0s in the K-maps
  - ↳ we AND together the resulting sum terms

Example: $f(x_1, \dots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$



$\longrightarrow f = (x_3 + x_4) \cdot (x_2 + x_3) \cdot (x_1' + x_2' + x_3' + x_4')$

Don't Cares: when a subset of inputs don't matter (ie, we don't care what the output is).

↳ Rather than using a 1 or 0, we use a "d" to mark a don't care as the output.

Example: $f(x_1,\dots,x_4) = \sum m(2,4,5,6,10) + D(12,13,14,15)$

x_3x_4 \ x_1x_2	00	01	11	10
00	0	1	d	0
01	0	1	d	0
11	0	0	d	0
10	1	1	d	1

↓ SOP implementation

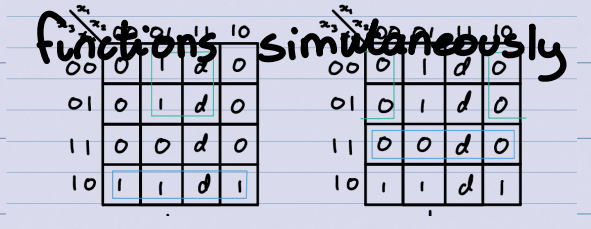
x_3x_4 \ x_1x_2	00	01	11	10
00	0	1	d	0
01	0	1	d	0
11	0	0	d	0
10	1	1	d	1

↓ POS implementation

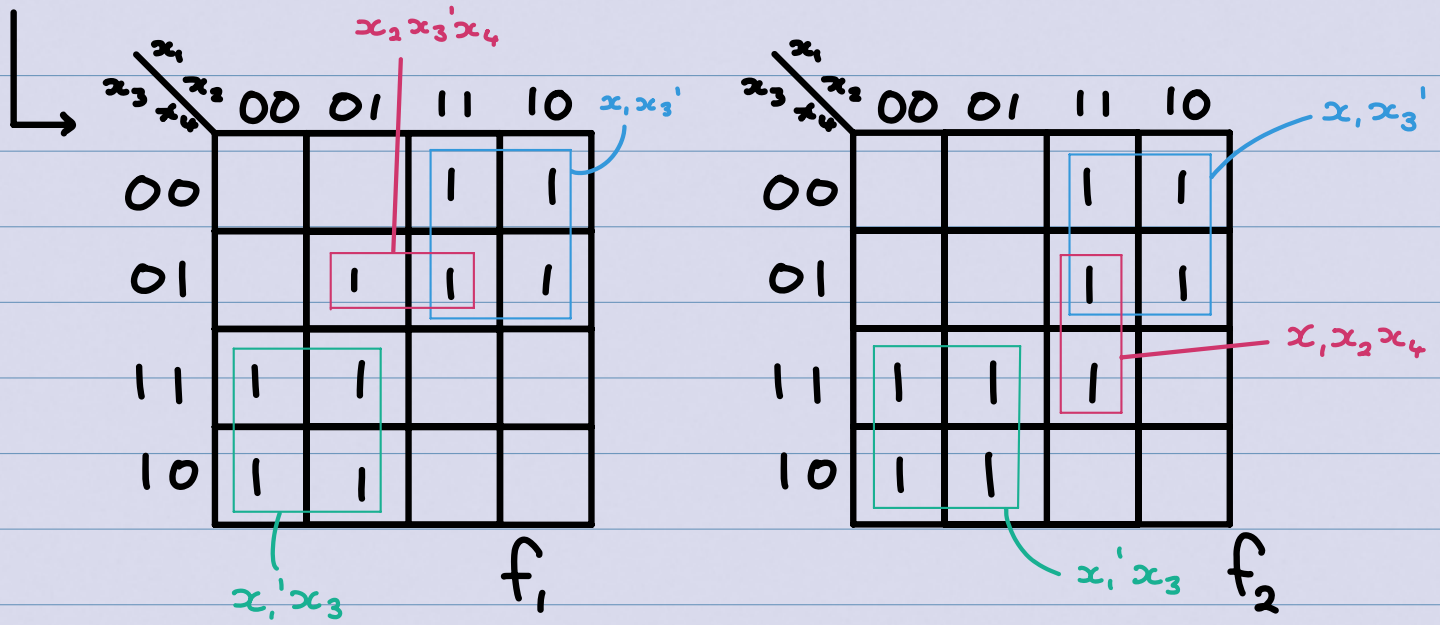
- d can act as either a 0 or 1, we can pick whichever makes it easier for us!

Multiple-Output Circuits:

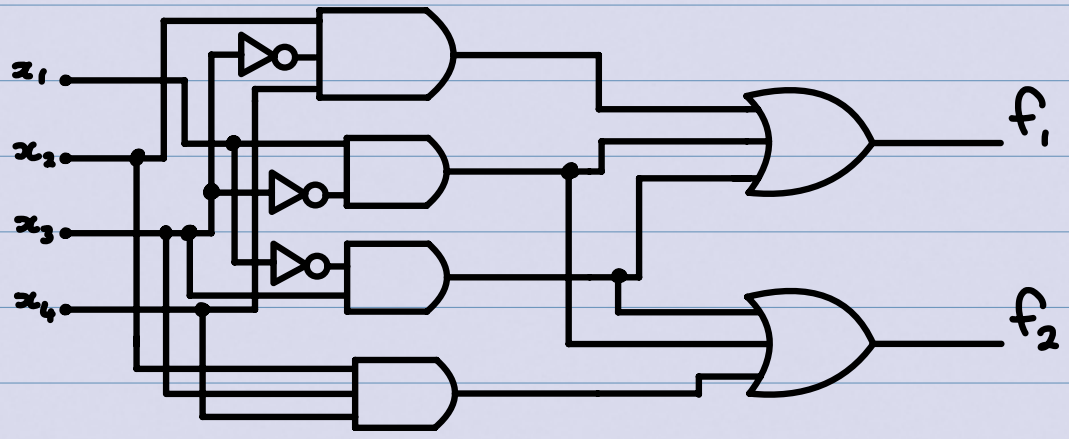
- Suppose we want to implement two or more functions of the same variables. It will often be a lower cost implementation to consider both functions simultaneously instead of individually.



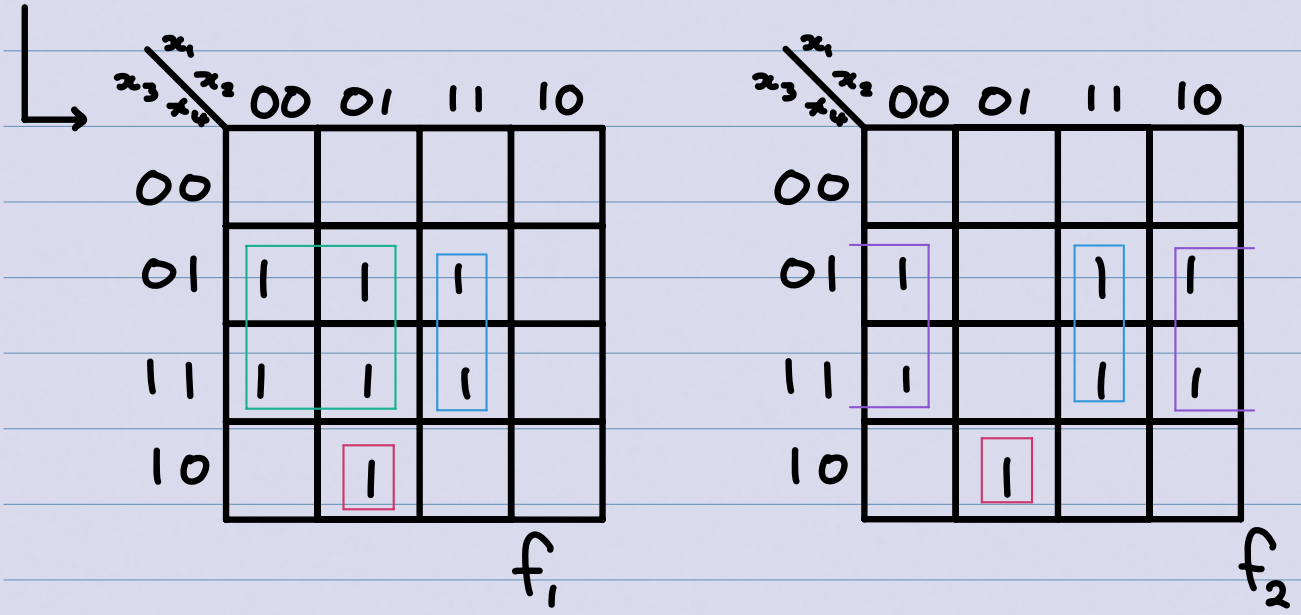
Example:



$$f_1 = x_1'x_3 + x_1x_3' + x_2x_3'x_4$$
$$f_2 = x_1'x_3 + x_1x_3' + x_1x_2x_4$$

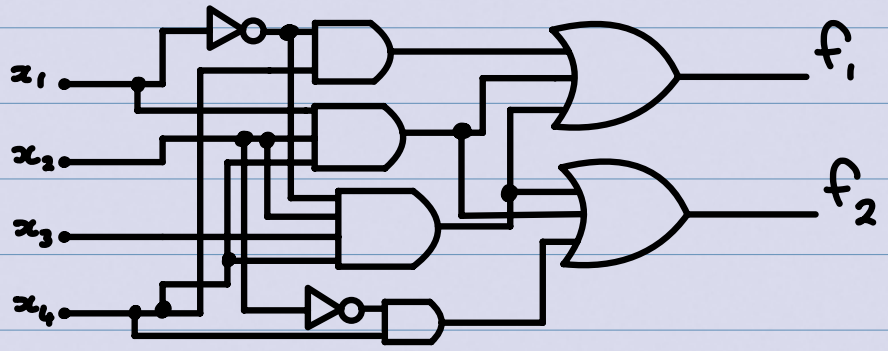


Example:



$f_1 = x_1'x_4 + x_1x_2x_4 + x_1'x_2x_3x_4$

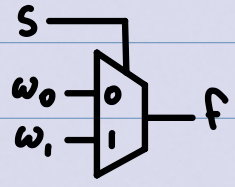
$f_2 = x_2'x_4 + x_1x_2x_4 + x_1'x_2x_3x_4$



Multiplexers

↳ a multiplexer circuit has a number of data inputs, one or more select inputs, and one output

Multiplexer symbol:

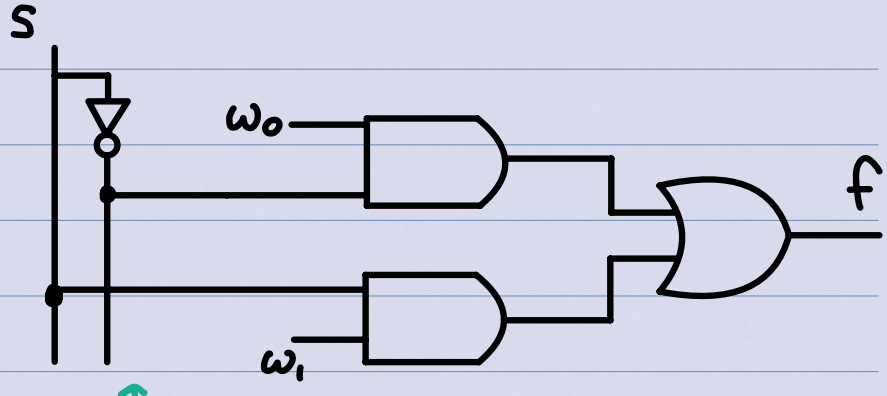


Multiplexer truth table:

s	f
0	w_0
1	w_1

$f = s'w_0 + sw_1$

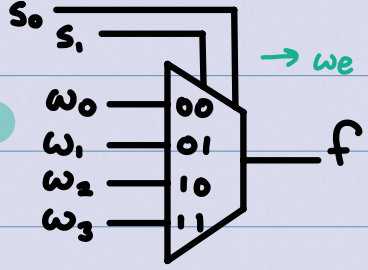
Sum-of-products circuit:



two-to-one multiplexer ↗

4-to-1 multiplexers

Symbol :



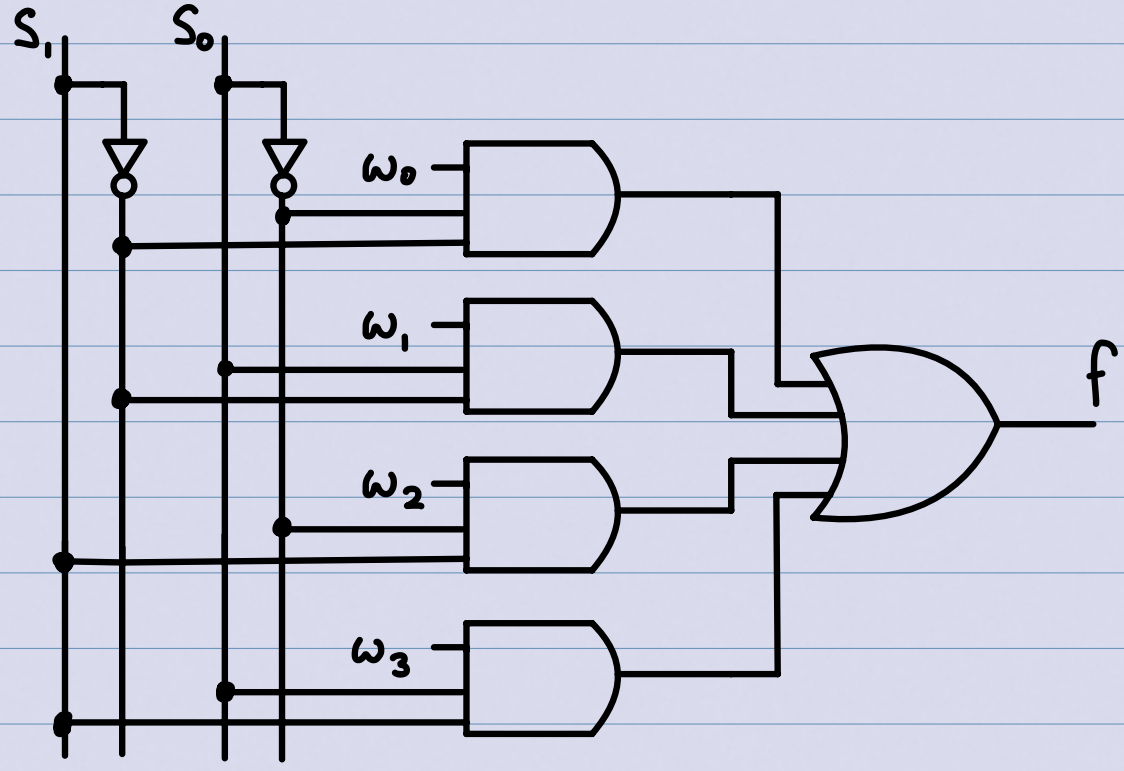
→ we now need a 2-bit input for the selector!

Truth Table :

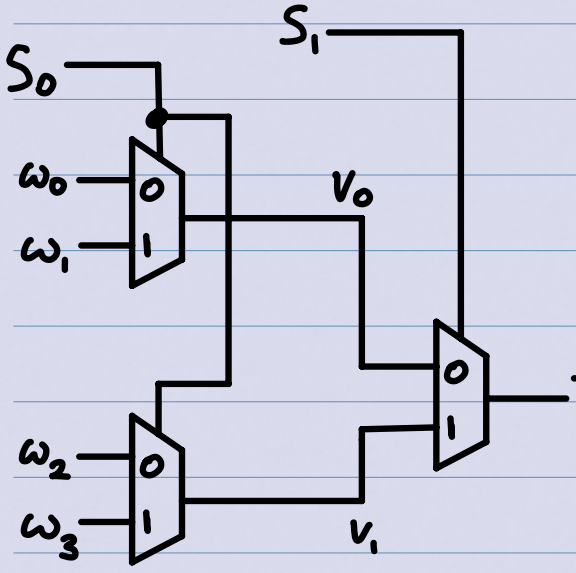
S_1	S_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

$f = S_1' S_0' w_0 + S_1' S_0 w_1 + S_1 S_0' + S_0 S_1 w_3$

Circuit :



Making a 4-to-1 multiplexer using 2-to-1 multiplexers:



Algebraically,

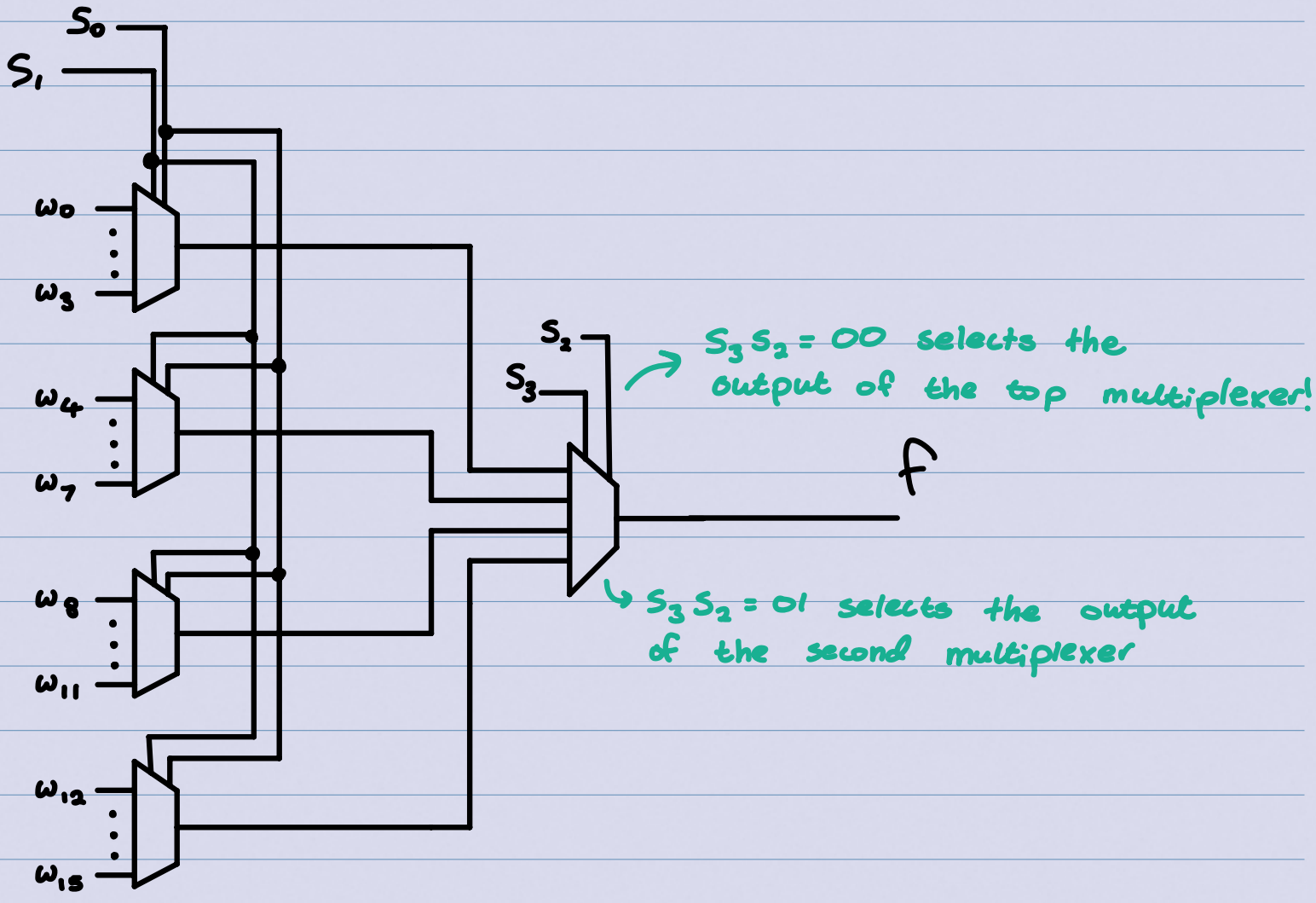
$f = S_1' V_0 + S_1 V_1$

$= S_1' (S_0' w_0 + S_0 w_1) + S_1 (S_0' w_2 + S_0 w_3)$

$f = S_0' S_1' w_0 + S_0 S_1' w_1 + S_0' S_1 w_2 + S_0 S_1 w_3$

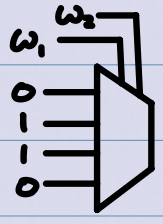
↳ Same as definition!

16-to-1 multiplexer using 4-to-1 multiplexers:



Synthesis of logic functions using multiplexers:

XOR $\left(\begin{array}{cc|c} w_1 & w_2 & f \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}\right)$ can of course be implemented simply using a 4-to-1 mux:

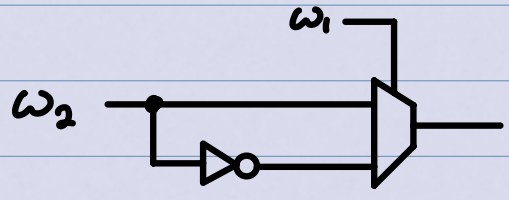


!

However, it can also be done more efficiently:

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\bar{w}_2$

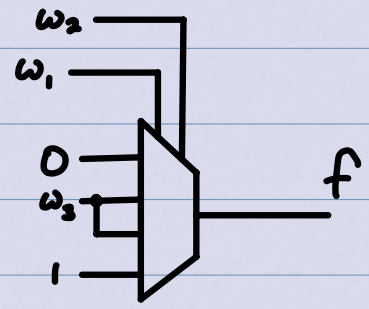


!

3- Variable Example :

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

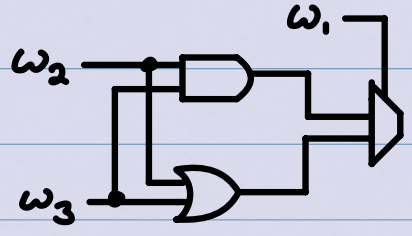
w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1



OR

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	$w_2 w_3$
1	$w_2 + w_3$



Shannon's expansion: Any bodean function $f(w_1, \dots, w_n)$ can be written in the form:

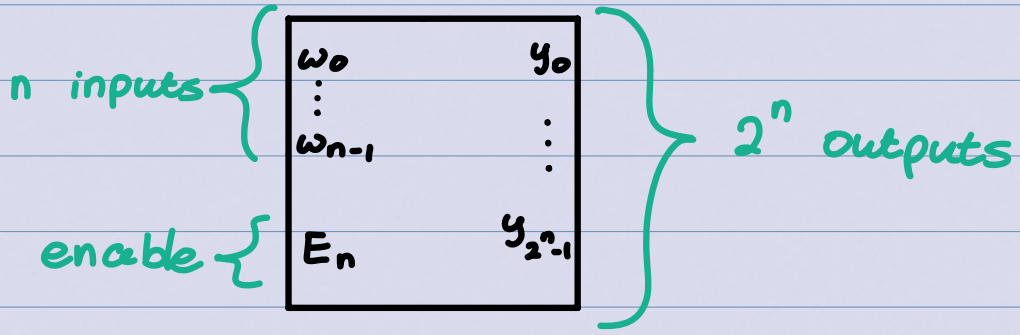
$$f(w_1, \dots, w_n) = w_1' \cdot f(0, \dots, w_n) + w_1 \cdot f(1, \dots, w_n)$$

Decoders:

- A binary decoder has **n inputs** and **2^n outputs**.
- Only **one output is asserted at a time**, and each output corresponds to one valuation of the inputs.
- A decoder typically has an enable input, E_n.

If **$E_n = 0$**, then **none of the outputs are asserted.**

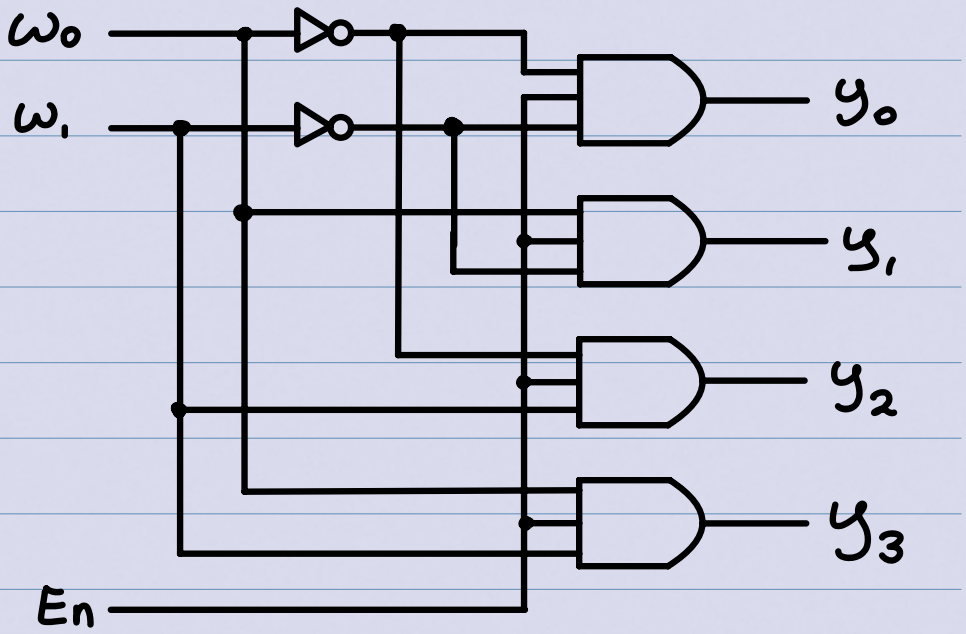
If $E_n = 1$, then the valuation of $w_{n-1} \dots w_1 w_0$ determines which output is asserted.



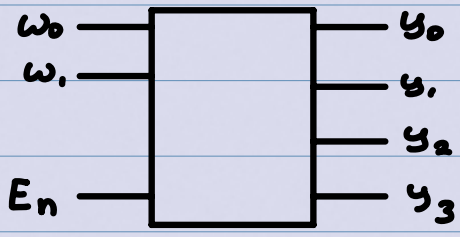
Truth Table example:

E_n	w_1	w_0	y_0	y_1	y_2	y_3
0	d	d	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Circuit Example:

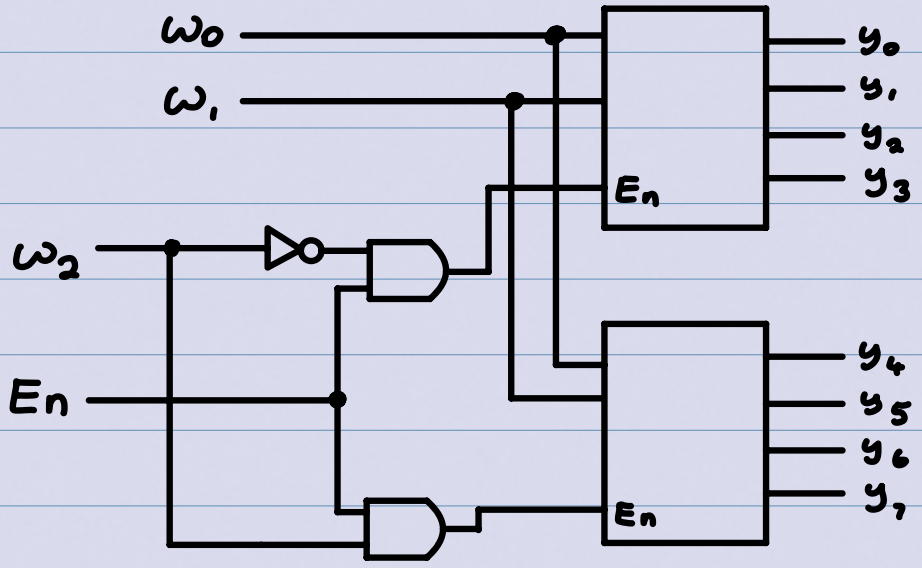


Symbol Example:

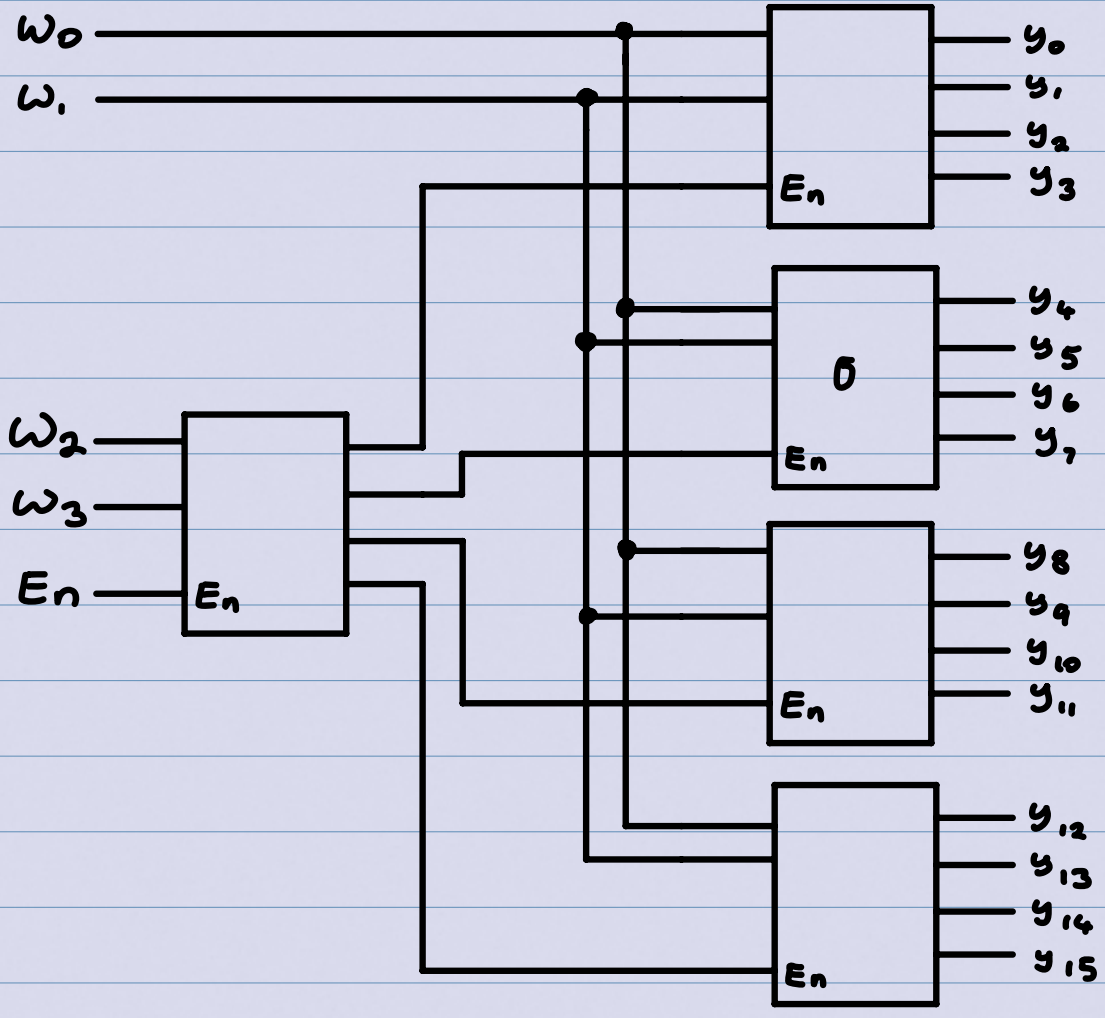


Note: $y_0 = E_n w_1' w_0'$, $y_1 = E_n w_1' w_0$, $y_2 = E_n w_1 w_0'$, ...

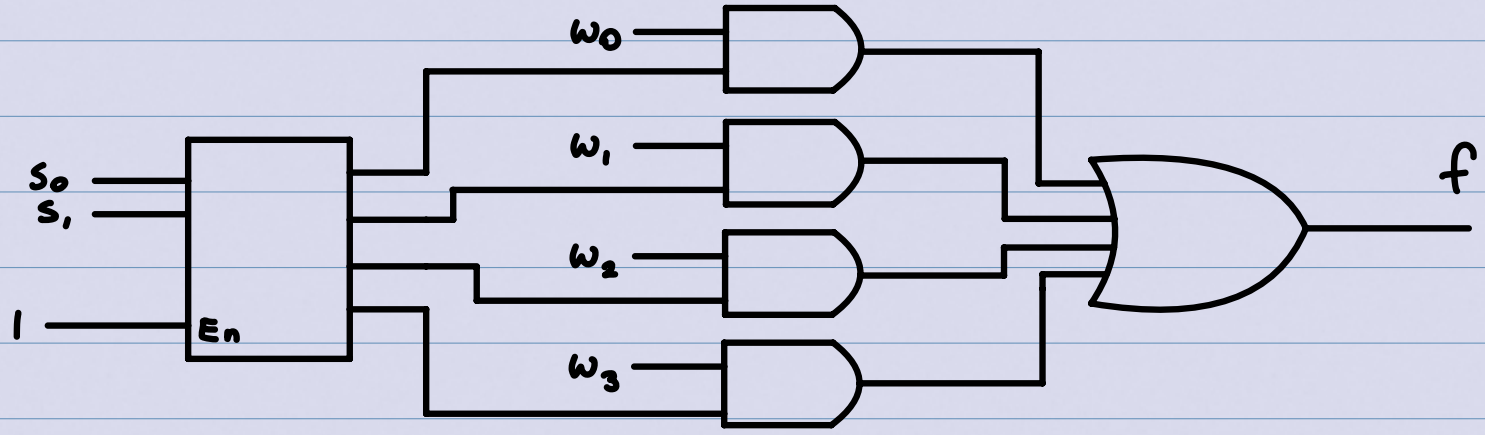
Making a 3-to-8 decoder using 2-to-4 decoders:



4-to-16 decoder using 2-to-4 decoders:



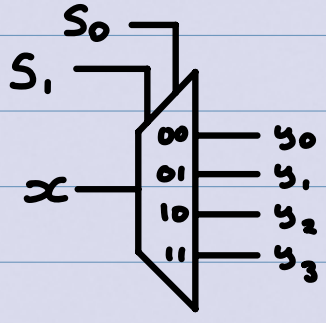
Using a decoder to make a 4-to-1 multiplexer:



De-multiplexers:

- A de-multiplexer has only **one data input**, **n select inputs**, and **2^n outputs**.
- Performs the opposite operation as a multiplexer, it **places the input value at one of the outputs specified by the select inputs.**

- **Symbol:**



→ $y_0 = x\, S_1'\, S_0'$

$y_1 = x\, S_1'\, S_0$

$y_2 = x\, S_1\, S_0'$

$y_3 = x\, S_1\, S_0$

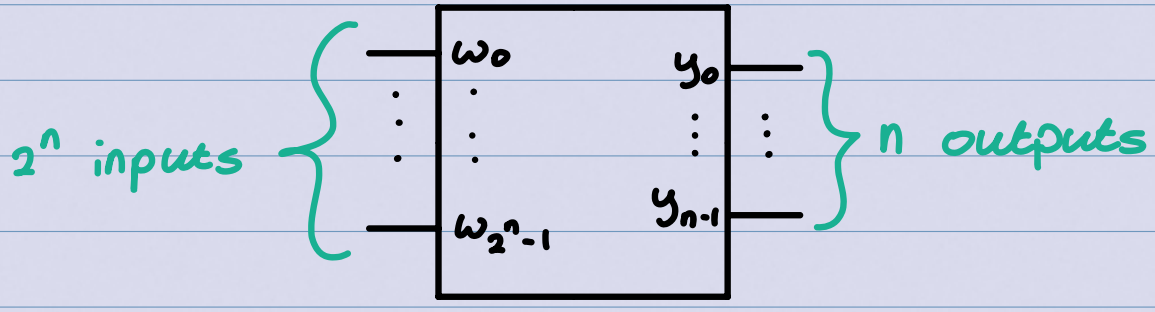
- **Truth Table:**

S_1	S_0	y_0	y_1	y_2	y_3
0	0	x	0	0	0
0	1	0	x	0	0
1	0	0	0	x	0
1	1	0	0	0	x

↳ these are for a 1-to-4 de-mux!

Encoders:

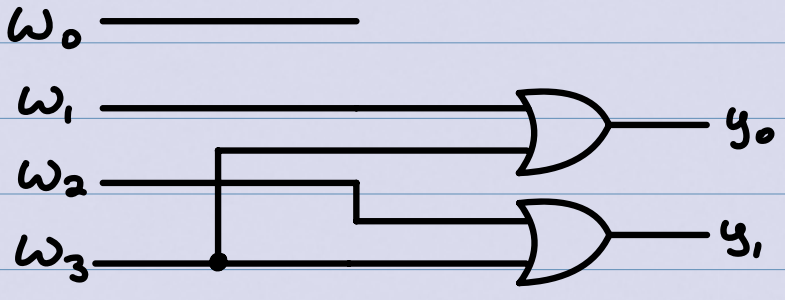
- A binary encoder encodes information from its 2^n inputs into an n-bit code
- One of the inputs should be one, and the outputs present the binary number that identifies which input is 1.



- Truth Table:

w_3	w_2	w_1	w_0	y_1	y_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

- Circuit:

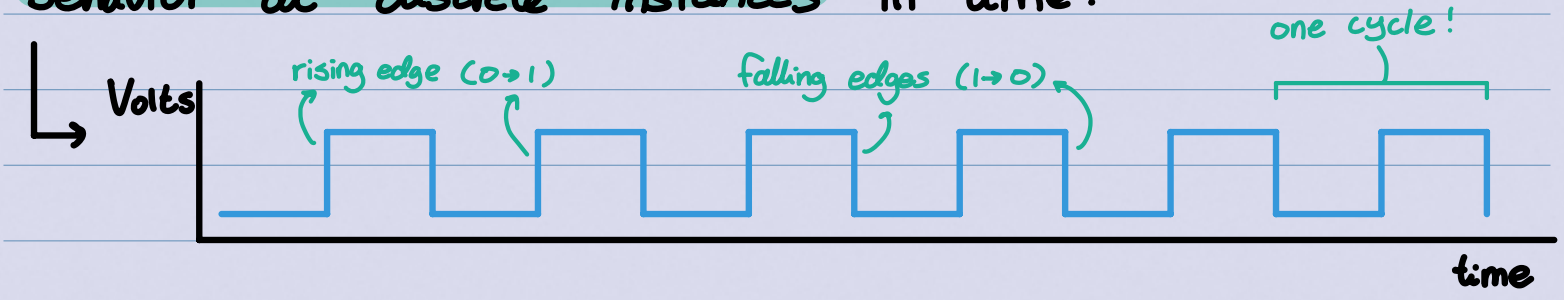


- Circuits in which outputs depend solely on present inputs (not any past signals) are known as combinational circuits.
- Circuits in which outputs also depend on past

Signals are known as sequential circuits.
↳ Sequential circuits have 'memory' or storage elements that can store logic values.

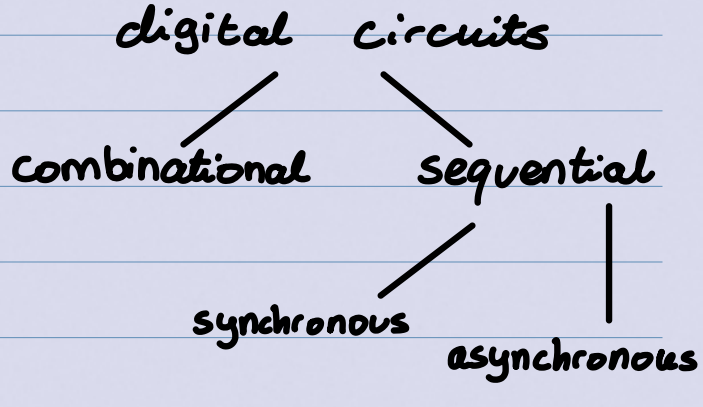
- There are two types of sequential circuits:
  ↳ Synchronous sequential circuits: circuit behavior (ie signals) changes only at discrete instances
  ↳ Asynchronous sequential circuits: circuit behavior can change at any point in time.

- A clock signal is used to change the circuit behavior at discrete instances in time.



↳ time duration of one cycle: period (t) [s/μs/ns...]
↳ clock frequency is the reciprocal of t: $f = 1/t$ [Hz]

Types of digital circuits:



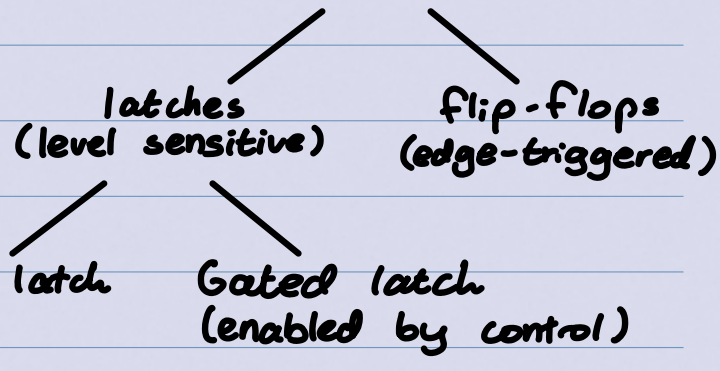
Storage Elements:
↳ Latches and Flip-Flops!

- Latches
  ↳ Basic Latch: responds with changes in the signal level of input data (level sensitive)

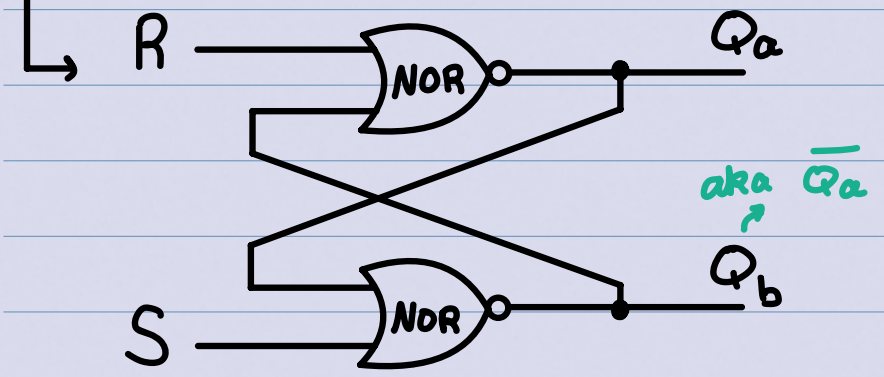
↳ Gated Latch: a basic latch with an enable type control input (eg, a clock).
  ↳ if the control is not asserted, the gated latch does not respond to changes in the signal level of input data.
  ↳ if the control is asserted, a gated latch acts like a level sensitive basic latch!
↳ For a periodic control signal like a clock, the state can change multiple times during a single cycle in response to changes in the signal level of input data.

- Flip-Flops:
  - Has an enable type control input (eg, clock) and can be viewed to have to gated latches.
  - The state of a flip-flop responds to input data signals only at one of the edges (rising or falling) of the control signal.
  - Since edges are almost instant in time, a flip-flop can't change its state more than once in a single cycle.

Types of storage elements:



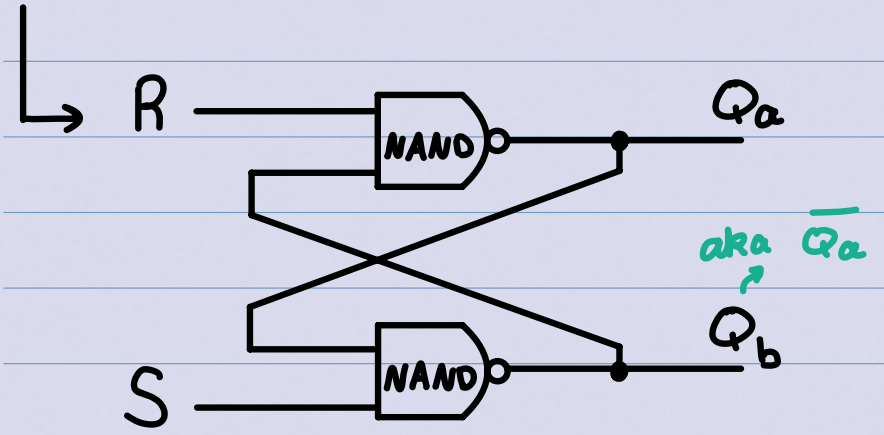
Basic latch (NOR):



S	R	
0	0	no change!
0	1	reset Qa to 0
1	0	set Qa to 1
1	1	Qa, Qb = 1

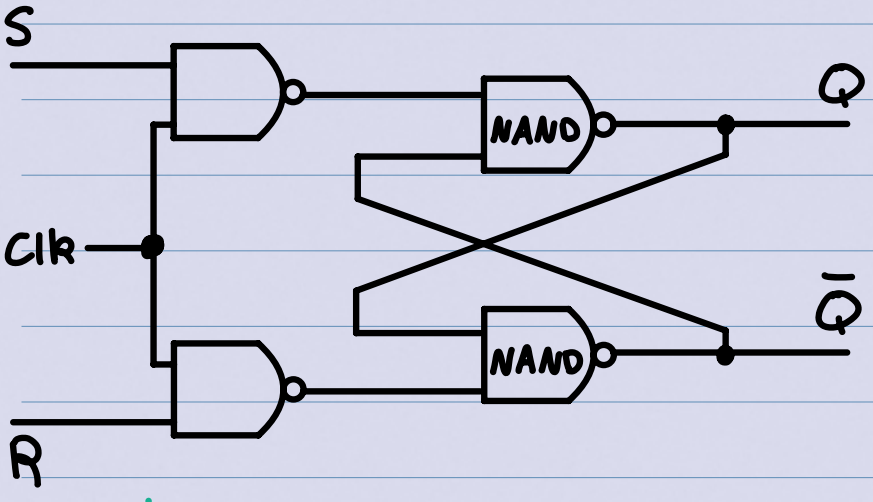
this should not happen

Basic latch (NAND):



S	R	
0	0	Qa, Qb = 1
0	1	reset Qa to 0
1	0	set Qa to 1
1	1	no change!

Gated Latch with NAND:



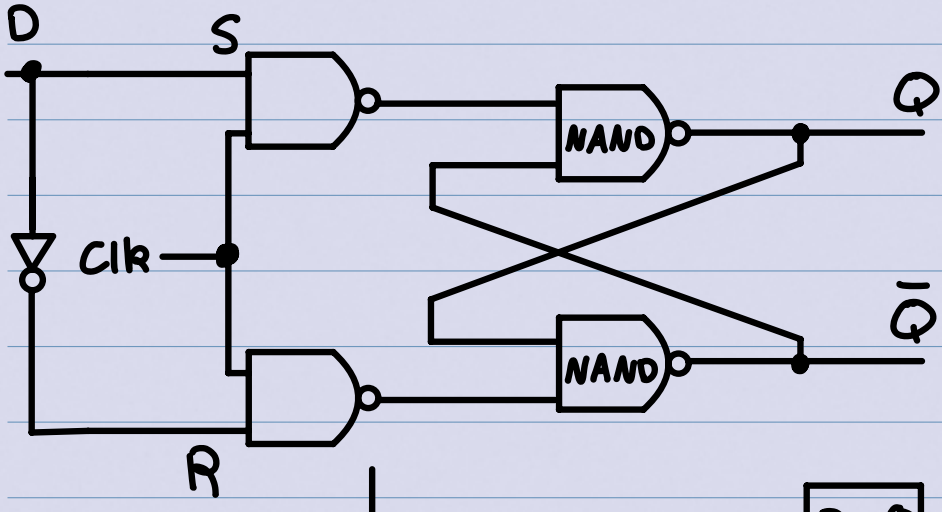
Clk	S	R	
0	d	d	no change!
1	0	0	no change!
1	0	1	reset Qa to 0
1	1	0	set Qa to 1
1	1	1	not allowed!

Use NAND for the enable gate, not AND!!!

However, the case S = R = 1 is still possible!

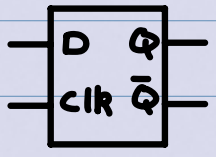
Gated D Latch with NAND:

Q(t+1) is the next state



Clk	D	Q(t+1)
0	d	no change!
1	0	reset Q to 0
1	1	set Q to 1

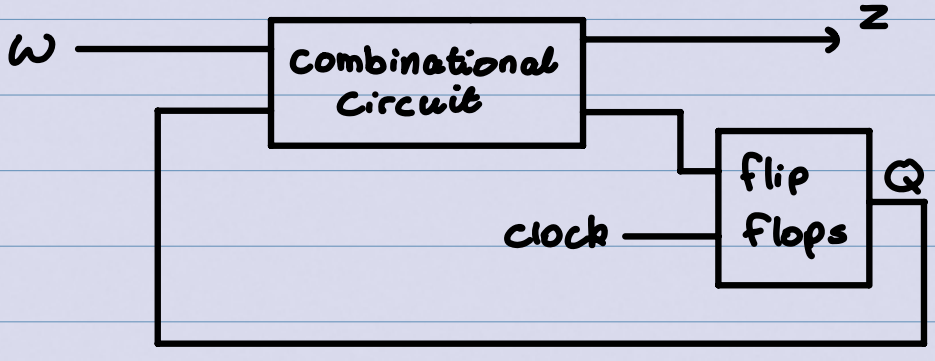
↳ Symbol:



Set 6:

A synchronous sequential circuit, aka a finite state machine, is made using combinational logic and flip-flops

↳ general form:

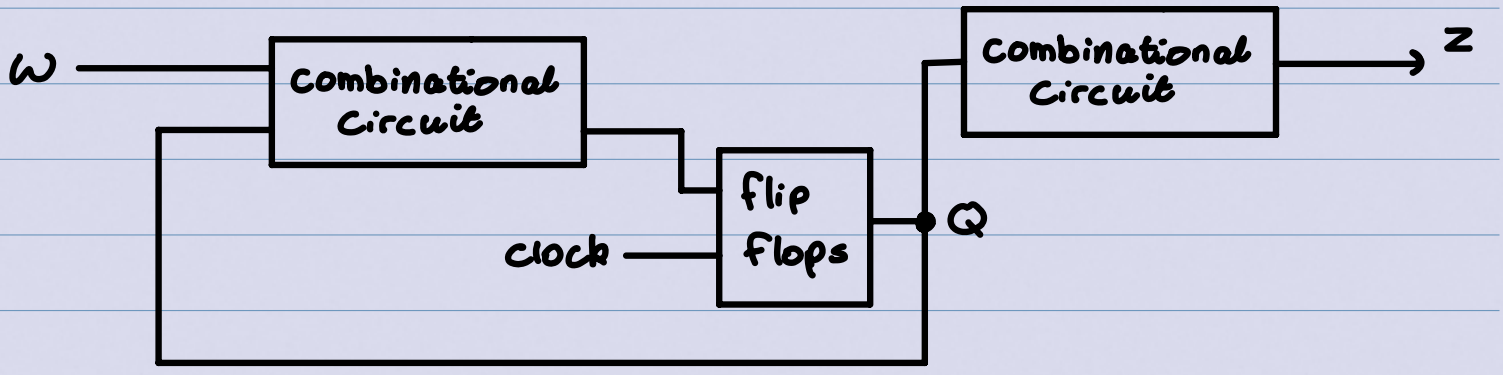


w: circuit input
Q: current state
z: circuit output

Moore Machine:

- A special case where circuit output (z) depends solely on the current state.
  - ↳ output changes predictably + synchronously with the clock (ie - at clock edges)

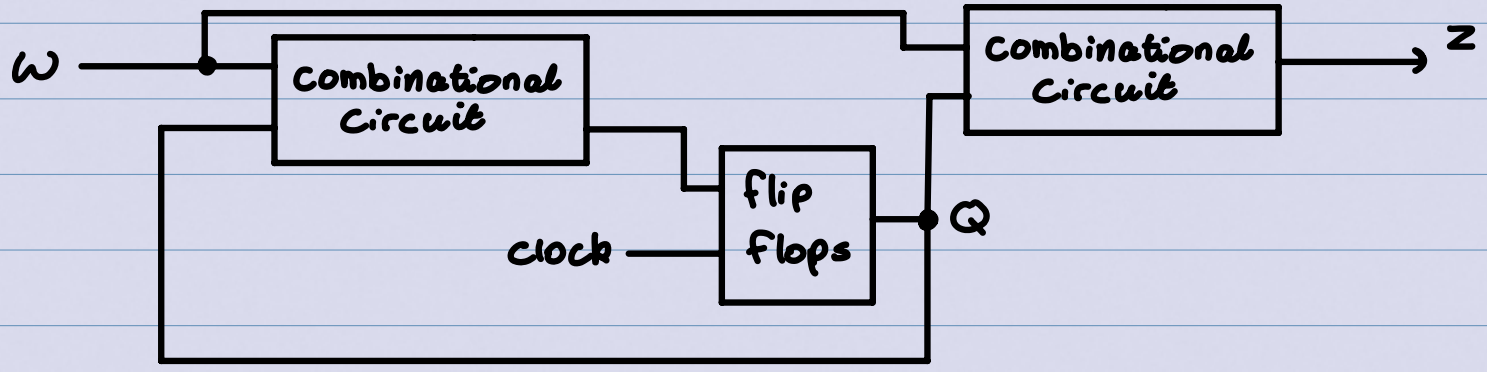
General Circuit:



Mealy Machine:

- When the output depends on both the current state (Q) AND the circuit input (w).

↳ Output changes asynchronously with the clock



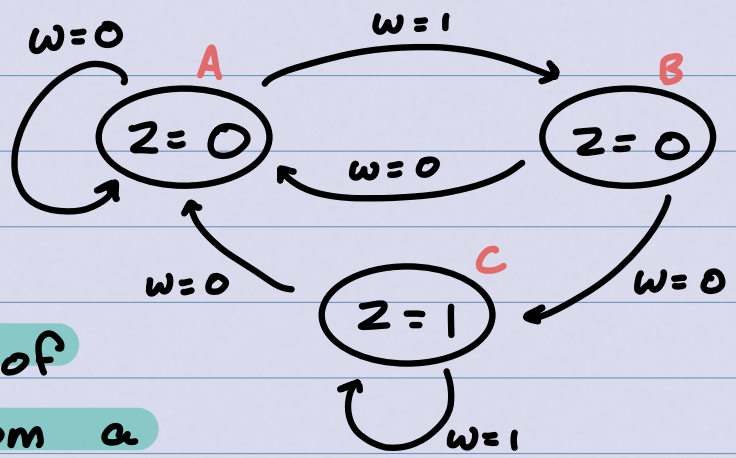
Example of a Moore Machine:

- one input w and one output z
- all changes occur at rising edges of clock
- output (z) = 1 iff w was 1 for the last two clock cycles.

↳

clock	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w	0	1	0	1	1	0	1	1	1	0	1
z	0	0	0	0	0	1	0	0	1	1	0

State diagram:



Note: the number of arcs originating from a node is 2^n, where n is the number of inputs!

State Table / Transition Table:

present state	Next State w=0	Next State w=1	output
A	A	B	0
B	A	C	0
C	A	C	1

now, let A = 00, B = 01, and C = 11 (no D = 11). also, let $Y_2 Y_1$ represent the next state.

present state $y_2 y_1$	next state w=0 $Y_2 Y_1$	next state w=1 $Y_2 Y_1$	output z
A 0 0	00	01	0
B 0 1	00	10	0
1 0	x	x	x
C 1 1	00	10	1

↳ using K-maps:

$y_2 y_1$ / Y_1	00	01	11	10
0	0	0	0	d
1	0	0	0	d

$w = 0$ $\quad Y_2$

$y_2 y_1$ / Y_1	00	01	11	10
0	0	0	0	d
1	0	1	1	d

$w = 1$ $\quad Y_2$

↳ we see that $Y_2 = w y_1$!!

$y_2 y_1$ / Y_2	00	01	11	10
0	0	0	0	d
1	0	0	0	d

$w = 0$ $\quad Y_1$

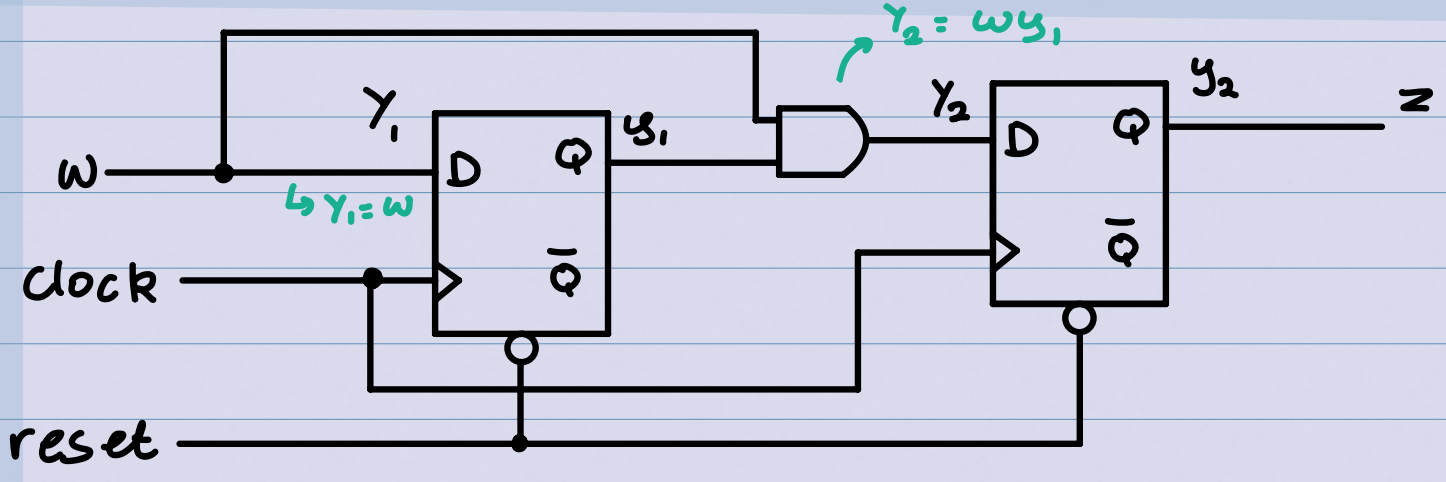
$y_2 y_1$ / Y_2	00	01	11	10
0	1	d	d	d
1	d	1	1	d

$w = 1$ $\quad Y_1$

↳ we see that $Y_1 = w$!

from the table, we see that $z = y_2$.

∴, this circuit can be constructed:



State Minimization:

- A synchronous sequential circuit that has n states requires at least $\log_2 n$ flip-flops.
  - ↳ By reducing the number of states, we can reduce the number of flip-flops.

- If two states are equivelant, we can combine them into a single state!
  - ↳ two states are equivelent if:
    - ∀ circuit inputs, the states give the same output
    - ∀ circuit inputs, the states give the same (or an equivelent) next state.

State Minimization Through Partitioning:

↳ 1) partition states according to circuit outputs
2) ∀ partition, iteratively do the following:
   - for any input pattern, if different states in a partition result in a transition to another partition, then those states are NOT equivelent; and we seperate them into smaller partitions
3) continue partitioning until all states in any partition transition to the same other partition for any input pattern
4) once no more partitioning is required, identification of the equivelent states is done

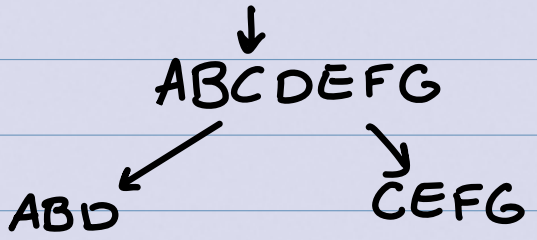
Example: minimize this state table:

Present state	Next state (NS) w=0	w=1	Output (z)
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

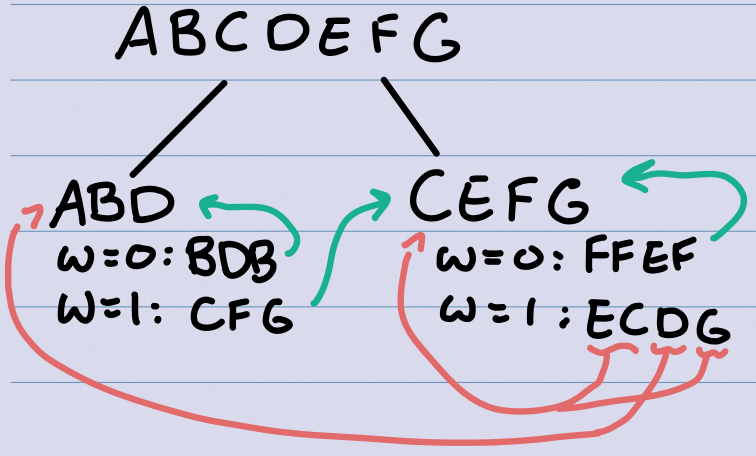
→ first, let's organize by output!

z=1 for P.S = A, B, D

z=0 for P.S = C, E, F, G.

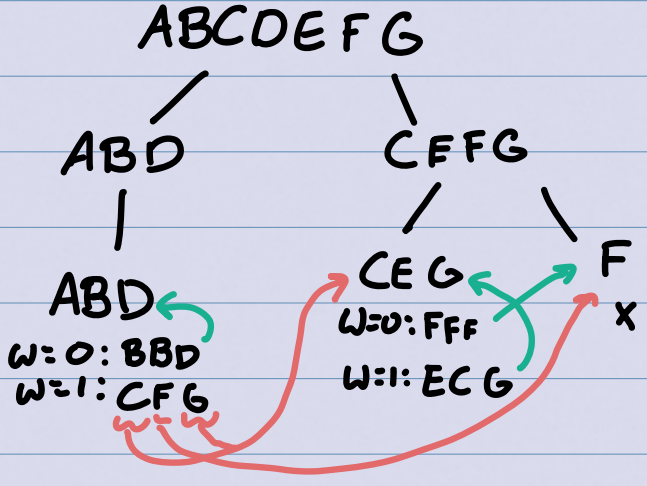


now, let's organize by next state:

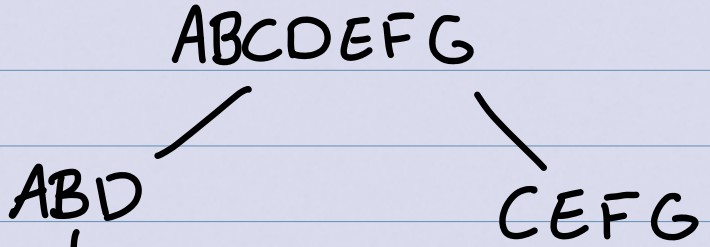


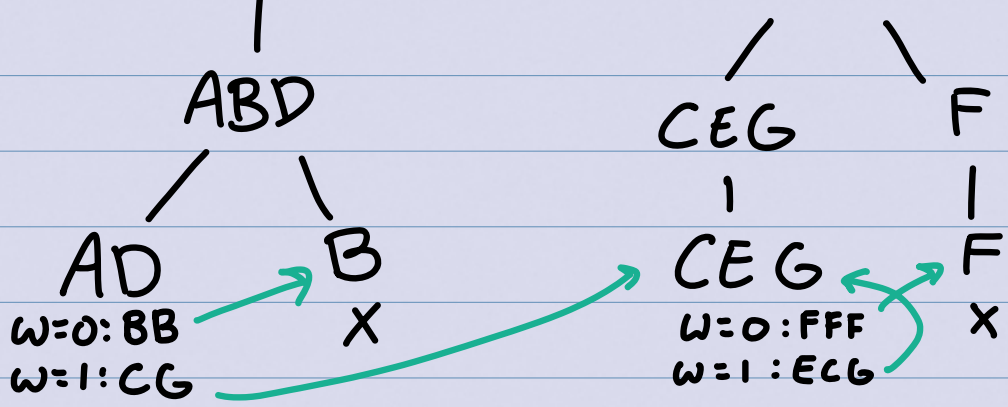
the D does not fit w/ the rest! ∴, since it's in 3rd position, must former partition 3rd thing → in this case, F.

↳



→ F doesn't fit, so must partition B out.





↳ A and D are equiv, CEG are equiv!

works now! re-writing the table:

Present state	Next state (NS) w=0	Next state (NS) w=1	Output (z)
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

→

Present state	Next state (NS) w=0	Next state (NS) w=1	Output (z)
A	B	C	1
B	~~D~~ A	F	1
C	F	~~E~~ C	0
~~D~~	~~B~~	~~G~~	~~1~~
~~E~~	~~F~~	~~C~~	~~0~~
F	~~E~~ C	~~D~~ A	0
~~G~~	~~F~~	~~G~~	~~0~~

↳ final minimized table:

PS	NS w=0	NS w=1	Z
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

FF excitation tables

D	$Q(t+1)$
0	0
1	1

1) DFF characteristic table

T	$Q(t+1)$
0	$Q(t)$
1	$Q'(t)$

3) TFF characteristic table

D	$Q(t)$	$Q(t+1)$
0	0	0
1	0	1
0	1	0
1	1	1

2) DFF excitation table
[Clearly, $Q(t+1) = D$]

T	$Q(t)$	$Q(t+1)$
0	0	0
1	0	1
0	1	1
1	1	0

4) TFF excitation table
[Note that $Q(t+1) = TQ'(t) + T'Q(t)$, i.e., if $Q(t)$=0, then $Q(t+1) = T$, else $Q(t+1) = T'$]

FF excitation tables (contd.)

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

5) JKFF characteristic table

J	K	$Q(t)$	$Q(t+1)$
0	X	0	0
1	X	0	1
X	0	1	1
X	1	1	0

7) JKFF excitation table (short form)

J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

6) JKFF excitation table

Note that:
- $Q(t+1) = J\,Q'(t) + K'\,Q(t)$
- I.e.,
  - if $Q(t)$=0, then $Q(t+1) = J$ and
  - if $Q(t)$=1, then $Q(t+1) = K'$

↳ D FF: N.S = D → D is output off D-FF

T FF: N.S = T if P.S = 0, else N.S = T'.

JK FF: if P.S = 0 then N.S = J, else N.S = K'.

asynch circuit is in stable state if all internal signals stop changing.

we will always assume that:

- only one input can change at a time
- input only changes after circuit is stab

State reduction in asynch:

- partitioning equivelent states
- merging compatible states

↳ partitioning: equivelent states are same as for synch circuit, but also;

two states (rows) in a flow table are potentially equivelent if any unspecified entries are in the same N.S column.

↳ don't remove the don't cares!

Example: minimize the states in this flow table:

↓

Present State	Next state DN = 00	01	10	11	Output z
A	(A)	B	(C)	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	(F)	–	0
E	A	(E)	–	–	1
F	A	–	(F)	–	1

Initial flow table

(ABD)(CEF)
(A)(BD)(CEF)
(A)(B)(D)(CEF)
(A)(B)(D)(CF)(E)

∴ :

P.S	NS DN= 00	01	10	11	Z
A	(A)	B	C	–	
B	D	(B)	–	–	
C	A	–	(C)	–	
D	(D)	E	C	–	
E	A	(E)	–	–	

Merging :

↳ compatible states: two states (rows) are compatible if there are no state conflicts for any input valuation. ;e :

↳ both rows have same output
AND
∀ input valuation, one of the following must be true:

1) both rows have the same successor
2) both rows are stable
3) the successor of at least 1 row is unspecified

Steps for merging:

1) identify compatible state pairs
2) draw a merger diagram
3) choose best merging possibility

Example:

Present state	Next state $w_2 w_1$ = 00	01	10	11	Output z
A	(A)	H	B	–	0
B	F	–	(B)	C	0
C	–	H	–	(C)	1
D	A	(D)	–	E	1
E	–	D	G	(E)	1
F	(F)	D	–	–	0
G	F	–	(G)	–	0
H	–	(H)	–	E	0

· first, let's look at rows with output z=0
↳ A, B, F, G, H.

A	(A)	H	B	–	0
B	F	–	(B)	C	0

→ not compatible, A ≠ F

A	(A)	H	B	–	0
F	(F)	D	–	–	0

→ not compatible, H ≠ D

A	(A)	H	B	–	0
G	F	–	(G)	–	0

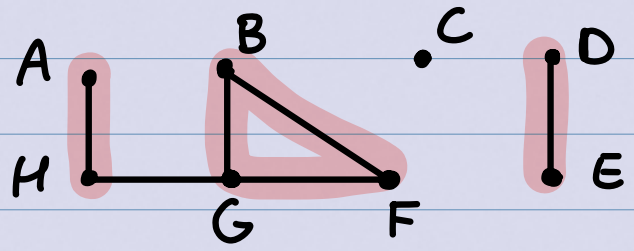
→ not compatible, A ≠ F

A	(A)	H	B	–	0
H	–	(H)	–	E	0

→ compatible: → (A, H).

... → (A, H), (B, F), (B, G), (F, G), (G, H).
(D, E).

Merger diagram:



$(A, H) \to A$, $(B, G), (B, F), (F, G) \to B$

$C \to C$, $(D, E) \to D$.

new table:

Present state	Next state $w_2 w_1$ = 00	01	10	11	Output z
A	(A)	H	B	–	0
B	B	–	(B)	C	0
C	–	H	–	(C)	1
D	A	(D)	–	D	1
E	–	D	G	(E)	1
F	B	(F)	D	–	–	0
G	B	F	–	(G)	–	0
H	–	(H)	–	E	0

A	A	A	B	–
B	B	–	B	C
C	–	A	–	C
D	A	D	B	D
E	–	D	B	D
F	B	D	–	–
G	B	–	B	–
H	–	A	–	D

$(A, H) \to A$:

A	A	A	B	D
B	B	–	B	C
C	–	A	–	C
D	A	D	B	D
E	–	D	B	D
F	B	D	–	–
G	B	–	B	–

(F, G):

A	A	A	B	D
B	B	–	B	C
C	–	A	–	C
D	A	D	B	D
E	–	D	B	D
F	B	D	B	–

(D,E):

A	A	A	B	D	0
B	B	D	B	C	0
C	-	A	-	C	1
D	A	D	B	D	1

→ final merged table!!

Race Conditions

↳ occurs in ASC when 2 or more state variables change in response to a change in the input value

- unequal circuit delays may not allow multiple state variable changes at the same time

- Assume 2 state variables change
  - ↳ if the circuit reaches the same final stable state regardless of the order in which the state variables change, then the race is non-critical
  - ↳ if it reaches the same final stable state depending on the order in which the state variables change, then the race is critical!

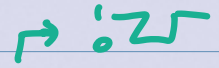
- We must avoid critical races!!

hazard: a momentary unwanted switching transient at a circuit's output (eg - glitch)

↳ occur due to unequal propogation delays

↳ can be static or dynamic

static-1 hazard: occurs when output is 1 and should remain 1, but temporarily switches to 0 due to a change in input



static-0 hazard: occurs when output is 0 and should remain 0, but temporarily switches to 1 due to an input change.

dynamic hazard: occurs when an input change and circuit output should change from 0→1 or 1→0, but temporarily flips between values.