Basics of Digital Logic Design

Dr. Arjan Durresi
Louisiana State University
Baton Rouge, LA 70810
Durresi@Csc.LSU.Edu

These slides are available at:

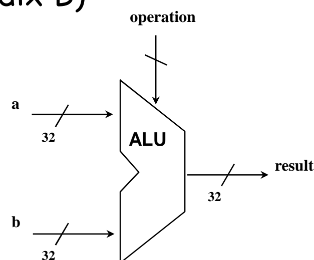
http://www.csc.lsu.edu/~durresi/CSC3501_07/

- Gates
- Boolean Algebra
- Karnough Maps
- Latches and Flip-Flops
- Registers

Lets Build a Processor

- Almost ready to move into chapter 5 and start building a processor
- First, let's review Boolean Logic and build the ALU we'll need

  (Material from Appendix B)



Review: Boolean Algebra & Gates

- Problem: Consider a logic function with three inputs: A, B, and C.

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

- Show the truth table for these three functions.
- Show the Boolean equations for these three functions.
- Show an implementation consisting of inverters, AND, and OR gates.

Signals, Logic Operations and Gates

- Rather than referring to voltage levels of signals, we shall consider signals that are logically 1 or 0 (or asserted or de-asserted)
- Logic operation

NOT

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

AND

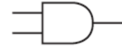
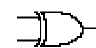
A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

- Gates  
- **Output**
- **is 1 if:** **Input is 0** | **Both inputs are 1s** | **At least one input is 1** | **Inputs are not equal**

Gates

- Gates are simplest digital logic circuits, and they implement basic logic operations (functions).
- Gates are designed using few resistors and transistors.
- Gates are used to build more complex circuits that implement more complex logic functions.

Classification of Logic Functions/Circuits

- Combinational logic functions (circuits):
  - any number of inputs and outputs
  - outputs **yi** depend only on current values of inputs **xi**
- Logic equations may be used to define a logic function.
- Example: A logic function with 4 inputs and 2 outputs

$$y_1 = (x_1 + (x_2 * x_3)) + ((\overline{x_3} * x_4) * x_1)$$
$$y_2 = (\overline{x_1} + (x_2 * x_4)) + ((x_1 * x_2) * \overline{x_3})$$

- "*" used for "and", "+" used for "or"
- For sequential functions (circuits):
  - outputs depend on current values of inputs and some internal states.
- Any logic function (circuit) can be realized using only and, or and not operations (gates).
- nand and nor operations (gates) are universal.

Basic Laws of Boolean Algebra

Identity laws: $A + 0 = A$
$A * 1 = A$

• Inverse laws: $A + \overline{A} = 1$
$A * \overline{A} = 0$

Zero and one laws: $A + 1 = 1$
$A * 0 = 0$

Commutative laws: $A + B = B + A$
$A * B = B * A$

Associative laws: $A + (B + C) = (A + B) + C$
$A * (B * C) = (A * B) * C$

Distributive laws : $A * (B + C) = (A * B) + (A * C)$
$A + (B * C) = (A + B) * (A + C)$

DeMorgan's laws: $\overline{(A + B)} = \overline{A} * \overline{B}$
$\overline{(A * B)} = \overline{A} + \overline{B}$

Simple Circuit Design: Example

- Given logic equations, it is easy to design a corresponding circuit

$$y_1 = (x_1 + (x_2 * x_3)) + ((\overline{x_3} * x_4) * x_1) = x_1 + (x_2 * x_3) + (\overline{x_3} * x_4 * x_1)$$
$$y_2 = (\overline{x_1} + (x_2 * x_4)) + ((x_1 * x_2) * \overline{x_3}) = \overline{x_1} + (x_2 * x_4) + (x_1 * x_2 * \overline{x_3})$$

Truth Tables

- Another way (in addition to logic equations) to define certain functionality
- Problem: their sizes grow exponentially with number of inputs.

inputs			outputs	
x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

What are logic equations corresponding to this table?

$y_1 = x_1 + x_2 + x_3$

$y_2 = x_1 * x_2 * x_3$

Design corresponding circuit.

Logic Equations in Sum of Products Form

- Systematic way to obtain logic equations from a given truth table.

inputs			outputs	
x_1	x_2	x_3	y_1	y_2
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

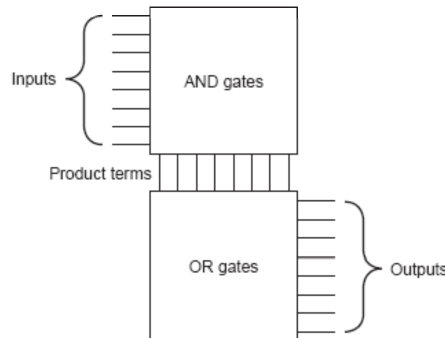
- A product term is included for each row where y_i has value 1
- A product term includes all input variables.
- At the end, all product terms are ored

$$y_1 = \overline{x_1} * \overline{x_2} * \overline{x_3} + \overline{x_1} * \overline{x_2} * x_3 + \overline{x_1} * x_2 * x_3 + x_1 * x_2 * x_3$$

$$y_2 = \overline{x_1} * \overline{x_2} * \overline{x_3} + \overline{x_1} * \overline{x_2} * x_3 + \overline{x_1} * x_2 * \overline{x_3} + x_1 * \overline{x_2} * \overline{x_3} + x_1 * \overline{x_2} * x_3$$

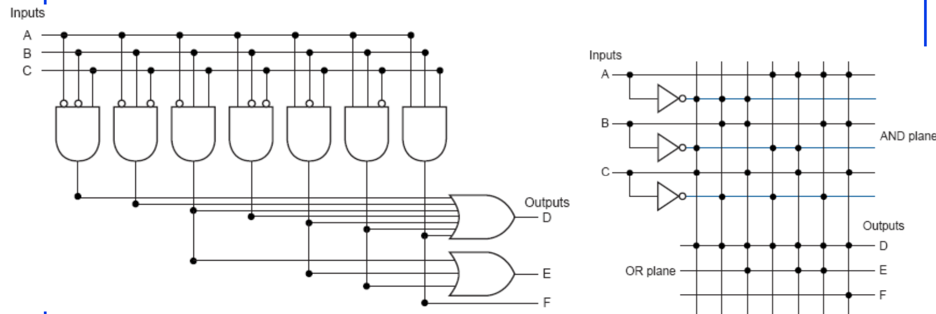
Programmable Logic Array - PLA

- PLA – structured logic implementation



PLA

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1



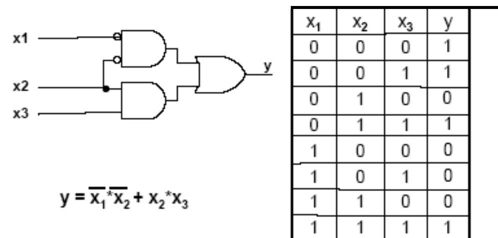
- Implementing a logic function

ROM

- **Logic functions can be implemented with:**
  - **Read-only memory (ROM).**
  - **Programmable ROMs (PROMs)**
- A ROM has a set of input address lines and a set of outputs.
- There are *m* input lines for 2^m addressable entries, called the *height*
- The number of bits in each addressable entry is equal to the number of output bits and is sometimes called the *width* of the ROM.
- A ROM can encode a collection of logic functions directly from the truth table.
- For example, if there are *n* functions with *m* inputs, we need a ROM with *m* address lines (and 2^m entries), with each entry being *n* bits wide.

Circuit - Logic Equation - Truth Table

- For the given logic circuit find its logic equation and truth table.



x_1	x_2	x_3	y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Note that y column above is identical to y_1 column Slide 11.
- Thus, the given logic function may be defined with different logic equations and then designed by different circuits.

Minimization Applying Boolean Laws

- Consider one of previous logic equations:

$$y_1 = \overline{x_1}*\overline{x_2}*\overline{x_3} + \overline{x_1}*\overline{x_2}*x_3 + \overline{x_1}*x_2*x_3 + x_1*x_2*x_3$$

$$= \overline{x_1}*\overline{x_2}*(\overline{x_3} + x_3) + x_2*x_3*(\overline{x_1} + x_1)$$

$$= \overline{x_1}*\overline{x_2} + x_2*x_3$$

- But if we start grouping in some other way we may not end up with the minimal equation.

Minimization Using Karnough Maps

- Provides more formal way to minimization
- Includes 3 steps
  - 1. Form Karnough maps from the given truth table. There is one Karnough map for each output variable.
  - 2. Group all 1s into as few groups as possible with groups as large as possible.
  - 3. each group makes one term of a minimal logic equation for the given output variable.
- *Forming Karnough maps*
- The key idea in the forming the map is that horizontally and vertically adjacent squares correspond to input variables that differ in one variable only. Thus, a value for the first column (row) can be arbitrary, but labeling of adjacent columns (rows) should be such that those values differ in the value of only one variable.

Minimization Using Karnough Maps

- *Grouping (*This step is critical)
- When two adjacent squares contain 1s, they indicate the possibility of an algebraic simplification and they may be combined in one group of two. Similarly, two adjacent pairs of 1s may be combined to form a group of four, then two adjacent groups of four can be combined to form a group of eight, and so on. In general, the number of squares in any valid group must be equal to 2^k. Note that one 1 can be a member of more than one group and keep in mind that you should end up with as few as possible groups which are as large as possible.
- *Finding Product Terms*
- The product term that corresponds to a given group is the product of variables whose values are constant in the group. If the value of input variable x_i is 0 for the group, then x_i is entered in the product, while if x_i has value 1 for the group, then x_i is entered in the product.

Minimization Using Karnough Maps

Example 1: Given truth table, find minimal circuit

x_1	x_2	x_3	y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

x3 \ x1 x2	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$y = \overline{x_1}*\overline{x_2} + x_2*x_3$

Minimization Using Karnough Maps

Example 2:

x_3 \ x_1 x_2	00	01	11	10
0	1	1	0	1
1	1	0	0	1

$y = \overline{x_1}*\overline{x_3} + \overline{x_2}$

Example 3:

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

$y = \overline{x_1}*\overline{x_2}*\overline{x_3} + \overline{x_1}*x_2*x_4 + x_2*x_3*x_4$

Example 4:

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

$y = x_1*\overline{x_4} + \overline{x_2}*x_3*x_4 + \overline{x_1}*x_2*\overline{x_3}*x_4$

Don't Cares

- Often in implementing some combinational logic, there are situations where we do not care what the value of some output is
  - because another output is true or
  - because a subset of the input combinations determines the values of the outputs.
- Such situations are referred to as *don't cares*.
- Don't cares are important because they make it easier to optimize the implementation of a logic function.

Don't cares

- The full truth table, without don't cares:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

- The truth table written with output don't cares

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

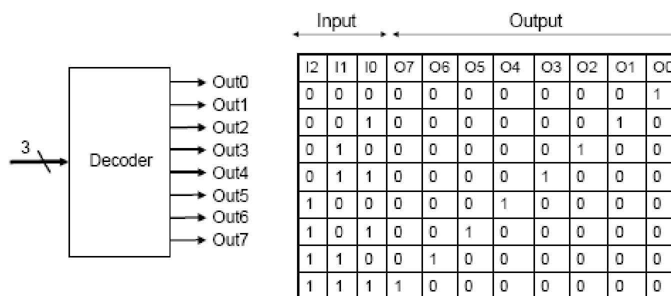
- The truth table with the input don't cares,

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

Decoders

- A full decoder with n input has 2^n outputs. Let inputs be labeled In_0, In_1, In_2,..., In_{n-1}, and let outputs be labeled Out_0, Out_1,..., Out_{2n-1}
- A full decoder functions as follows: Only one of outputs has value 1 (it is active) while all other outputs have value 0. The only output set to 1 is one labeled with the decimal value equal to the (binary) value on input lines
- In general, a decoder with n inputs may have fewer than 2^n outputs. Sometime those are called partial decoders. Decoders with only one output are common.

3-Input Full Decoder



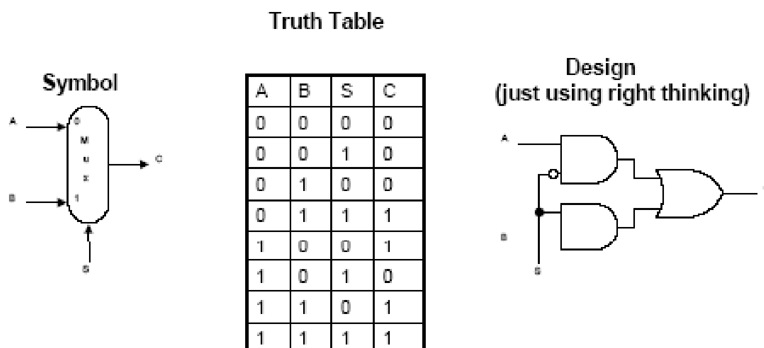
I2	I1	I0	O7	O6	O5	O4	O3	O2	O1	O0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Multiplexers

- A basic multiplexer has only one output line z. There are two sets of input lines: data lines and select lines.
- Let a number of data lines be N, labeled d_0, d_1, d_2,... d_{N-1}. There are m select lines, labeled s_0, s_1,..., s_{m-1}. m is such that any of data lines can be referenced (selected) by a decimal value on select lines. Thus, m has to satisfy the following inequality: $2^{m-1} < N \le 2^m$.
- A multiplexer functions as follows: Output z has the value of the data input line labeled by the value on select lines.

Simplest Basic Multiplexer

- Simplest mux is one with 2 data input lines and 1 select line.



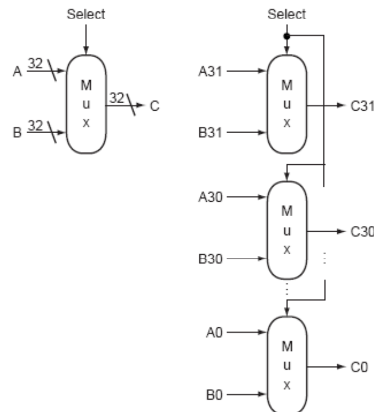
A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

3-Data Multiplexer Truth Table

do	d1	d2	s1	s0	z	
0	0	0	0	0	0	
0	0	0	0	1	0	
0	0	0	1	0	0	
0	0	0	1	1	d	This input not allowed
0	0	1	0	0	0	
0	0	1	0	1	0	
0	0	1	1	0	1	
0	0	1	1	1	d	This input not allowed
0	1	0	0	0	0	
0	1	0	0	1	1	
0	1	0	1	0	0	
0	1	0	1	1	d	This input not allowed
0	1	1	0	0	0	
-	-	-	-	-	-	
1	1	1	1	0	1	
1	1	1	1	1	d	

Complex Multiplexer

- Instead N single data lines and one output line as in a basic mux, a complex mux has N sets of data lines and one set of output lines and each set has K lines.



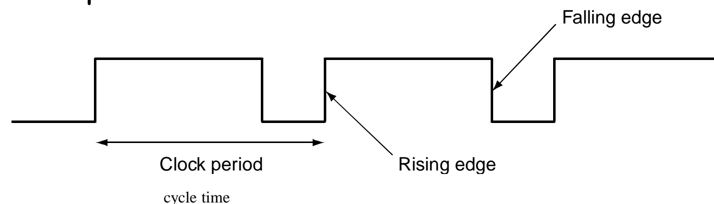
a. A 32-bit wide 2-to-1 multiplexor b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

Memory Elements: Flip-flops, Latches, and Registers

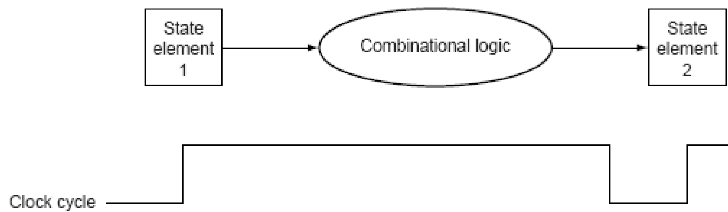
- All memory elements store state:
  - the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element.
- All logic blocks containing a memory element contain state and are sequential.

State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?

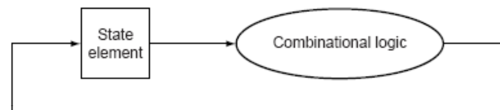


Clocking Methodology



- In an edge-triggered methodology, either the rising edge or the falling edge of the clock is *active* and causes state changes to occur.
- To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize, then the clock edge samples those value for storage in the state elements.

Clocking Methodology



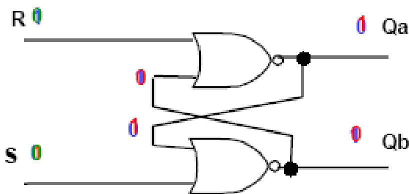
- It is possible to have a state element that is used as both an input and output to the same combinational logic block

Latches and Flip-flops

- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
  - "logically true" - could mean electrically low
- Flip-flop: state changes only on a clock edge (edge-triggered methodology)
  - A clocking methodology defines when signals can be read and written — wouldn't want to read a signal at the same time it was being written

R-S Latch (set-reset latch): Simplest Sequential Circuit

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0



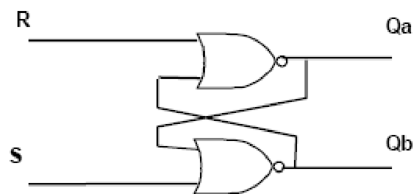
1. Let us start with: S = 0 & R = 1 → Qa = 0 & Qb = 1
2. Let us now change R to 0: S = 0 & R = 0 → Qa = 0 & Qb = 1, i.e. no change
3. Let us now change S to 1: S = 1 & R = 0 → Qa = 1 & Qb = 0
4. Let us now change S to 0: S = 0 & R = 0 → Qa = 1 & Qb = 0, i.e. no change

Thus, for steps 2 and 4 inputs are identical while outputs are different, i.e. we have a sequential circuit.

R-S latch is an *unclocked* memory element

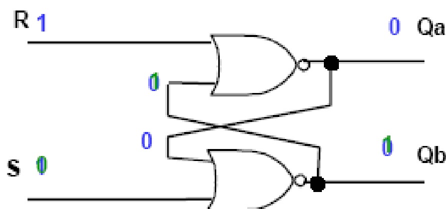
R-S Latch Characteristics

- R-S letch is a memory element that “remembers” which of two inputs has most recently had value 1:
- Outputs Qa =1 & Qb = 0 indicate that S is currently or was 1 last
- Outputs Qa =0 & Qb = 1 indicate that R is currently or was 1 last



R-S Latch Characteristics (continued)

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0



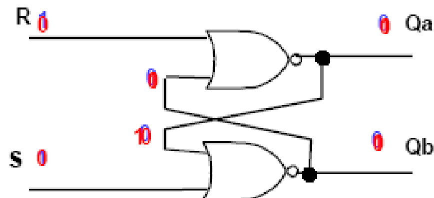
5. Let us consider case: **S** = 1 & R = 1 → Qa = 0 & Qb = 0

Qa = 0 & Qb = 0 indicate currently S = 1 and R = 1

6a. Let us now change **S** to 0: **S** = 0 & R = 1 →Qa = 0 & Qb = 1,

R-S Latch Characteristics (continued)

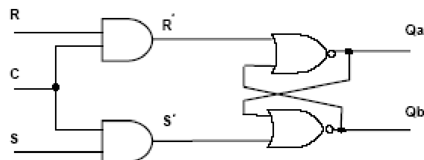
A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0



5. Let us again consider case: S = 1 & R = 1 → Qa = 0 & Qb = 0

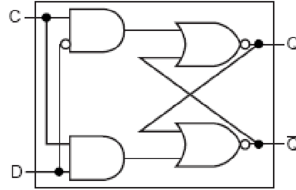
6b. Let us now change S and R simultaneously to 0: S = 0 & R = 0
→ Unstable state

Gated R-S Latch



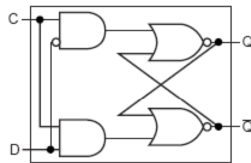
- C input is write enable (not a clock)
- When C = 1, a gated R-S latch behaves as an ordinary R-S latch.
- When C = 0, changes in R and S do not influence outputs.
- Note that the case R'=1 & S'=1 is still possible, and the unstable state can be reached easily.

(Gated) D-Latch



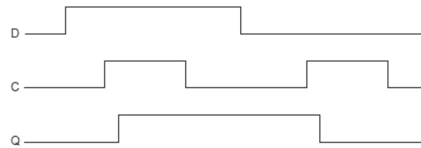
- Two inputs:
  - the data value to be stored (D)
  - the write enable signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement (often unused)
- Note: The case R'=1 & S'=1 is not possible.

D-Latch Functioning



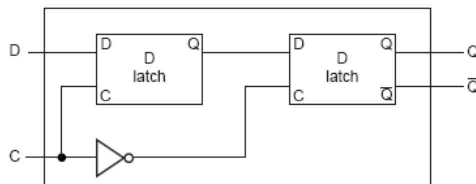
- When C=1, D-latch state (and Q-output) is identical to D-input, i.e. any change in the value of D-input is immediately followed by the change of Q-output.
- When C=0, D-latch state is unchanged and it keeps the value it had at the time when C input changed from 1 to 0.

D-Latch Functioning



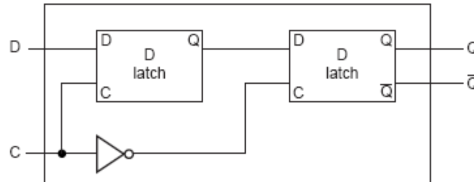
- Operation of a D latch assuming the output is initially deasserted.
- When the clock, *C*, is asserted, the latch is open and the *Q* output immediately assumes the value of the *D* input.

D Flip-Flop



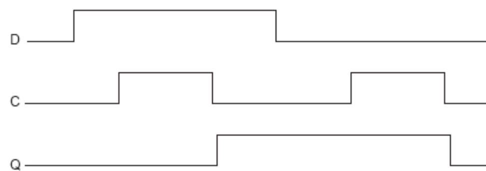
- Two inputs:
  - the data value to be stored (D)
  - the write enable signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement (often unused)

D Flip-Flop Functioning



- When C changes its value from 1 to 0, i.e. on the falling edge, D-flip flop state (and Q output) gets the value D-input has at that moment,
- During all other times, D-flip flop state is unchanged and it keeps the value it had at the time of the falling edge of C-input.
- There is a critical period T_{cr} around the falling edge of C during which value on D should not change. T_{cr} is split into two parts, the setup time before the C edge, and the hold time after the C edge.

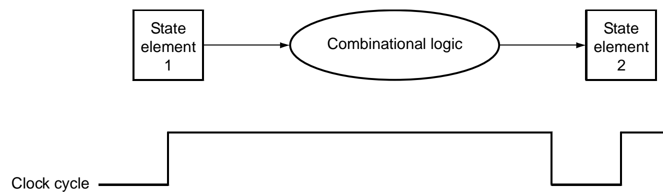
D Flip-Flop Functioning



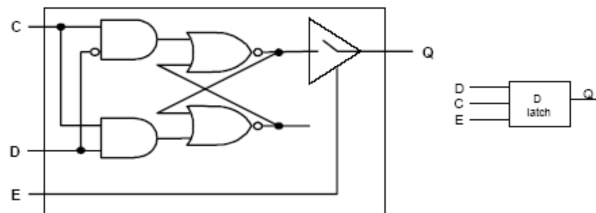
- Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted.
- When the clock input (*C*) changes from asserted to deasserted, the *Q* output stores the value of the *D* input.
- Compare this behavior to that of the clocked D latch shown in slide 41.
- In a clocked latch, the stored value and the output, *Q*, both change whenever *C* is high, as opposed to only when *C* transitions.

Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

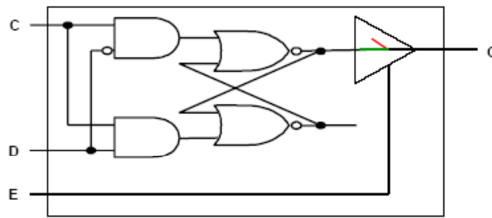


Three State D-Latch



- Three inputs:
  - **the data value to be stored (D)**
  - **the write enable signal (C) indicating when to read & store D**
  - **the read enable signal (E) indicating when internal state is provided on the output**
- One output:
  - **the value of the internal state (Q)**

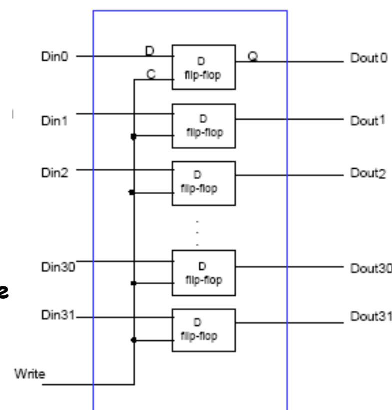
Three State D-Latch Functioning



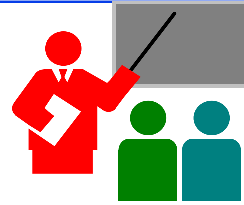
- Storing (writing) is performed as in the case of the (ordinary) Dlatch)
- When E=1 (enable read), then the switch is closed, and Q has value (0 or 1) that has been stored (written) into the D-latch
- When E=0 (disable read), then the switch is open, and Q is in the high impendence state (the third possible "value" on the output).

Design of 32-bit Register

- D flip-flop as a building block
- Thus, 32-bit register has:
  - 33 inputs and
  - 32 outputs
- There are two operations on a register:
  - read and
  - Write
- Read operation:
  - register content is always available on Dout0-Dout31
- Write operation:
  - provide desired values on Din0-Din31
  - generate falling edge on the write line
  - Recall critical period T_{cr} around falling edge.



Summary



- Gates
- Boolean Algebra
- Karnough Maps
- Latches and Flip-Flops
- Registers