

Chapter 2 Digital Design, Simulation and Implementation

This chapter discusses Digital Circuit Design, Simulation and Implementation

2.1 Digital Design Methodologies

Logic circuits are classified into two types, "combinational" and "sequential." A combinational logic circuit is one whose outputs depend only on its current inputs. The output of a sequential logic circuit depends not only on the current inputs, but also on the past sequence of inputs, possibly arbitrarily far back in time.

A combinational circuit may contain an arbitrary number of logic gates and inverters but no feedback loops. A feedback loop is a signal path of a circuit that allows the output of a gate to propagate back to the input of that same gate; such a loop generally creates sequential circuit behaviour.

In combinational circuit analysis, we start with a logic diagram, and proceed to a formal description of the function performed by that circuit, such as a truth table or a logic expression. In design/synthesis, we do the reverse, starting with a formal description and proceeding to a logic diagram.

Definitions

A *literal* is a variable or the complement of a variable. Examples: X , Y , X' , Y' .

A *product term* is a single literal or a logical product of two or more literals. Examples: W' , $X.Y.Z$, $X.Y'.Z$, $W'.Y'.Z'$.

A *sum-of-products* expression is a logical sum of product terms. Example: $Z'+W.X.Y+X.Y'.Z+W'.Y'.Z$.

A *product-of-sums* expression is a logical product of sum terms. Example: $Z'.(W+X+Y).(X+Y'+Z).(W'+Y'+Z)$.

The *canonical sum* of a logic function is a sum of the product terms corresponding to truth-table rows (input combinations) for which the function produces a 1 output. Function F of Table 2.1 can be described below:

$$\begin{aligned} F &= \Sigma_{X,Y,Z} (0,2,4,5,7) \\ &= X'.Y'.Z' + X'.Y.Z' + X.Y'.Z' + X.Y'.Z + X.Y.Z \end{aligned}$$

X	Y	Z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.1

Hence the notation $\Sigma_{X,Y,Z} (0,2,4,5,7)$ is the sum of product terms corresponding 0, 2, 4, 5, and 7 with variables X, Y, and Z.

The *canonical product* of a logic function is a product of the sum terms corresponding to input combinations for which the function produces a 0 output. Function F in Table 2.1 can also be described using canonical product:

$$\begin{aligned} F &= \Pi_{X,Y,Z} (1,3,6) \\ &= (X+Y+Z').(X+Y'+Z').(X'+Y'+Z) \end{aligned}$$

Hence the notation $\Pi_{X,Y,Z} (1,2,6,7)$ is a the product of sum terms corresponding to 1, 2, 6 and 7 with variables X, Y, and Z.

In Summary, there are five possible representations for a combinational logic function:

- A truth table.
- Sum of Product
- Product of Sum
- Canonical Sum
- Canonical Product

Switching-Algebra

The following three laws are the same for Boolean algebra as they are for ordinary algebra:

1. *Commutative law of addition and multiplication:*

eg. $A + B = B + A$

$$AB = BA$$

2. *Associative law of addition and multiplication:*

eg. $A + (B + C) = (A + B) + C$

$$A(BC) = (AB)C$$

3. *Distributive law:*

eg. $A(B+C) = AB + AC$

$$(A+B)(C+D) = AC + AD + BC + BD$$

These three laws hold true for any number of variables.

In addition to the basic Commutative, Associative, and Distributive laws, several Boolean identities and theorems are very useful in simplifying Boolean logic equations and logic circuits. They are listed below:

Rule 1 $A \bullet B \bullet C \bullet 0 = 0$

Rule 2 $A \bullet 1 = A$

Rule 3 $A + 0 = A$

Rule 4 $A + B + C + 1 = 1$

Rule 5 $A \bullet A \bullet A = A$

Rule 6 $A + A + A = A$

Rule 7 $A \bullet A' = 0$

Rule 8 $A + A' = 1$

Rule 9 $(A')' = A$

Rule 10 $A + A'B = A + B$
 $A' + AB = A' + B$

DeMorgan's Theorem is another useful theorem:

$$(X_1 + X_2 + X_3 + \dots + X_n)' = X_1' \cdot X_2' \cdot X_3' \cdot \dots \cdot X_n'$$

$$(X_1 \cdot X_2 \cdot X_3 \cdot \dots \cdot X_n)' = X_1' + X_2' + X_3' + \dots + X_n'$$

Example:

Use DeMorgan's theorem to convert the following SOP expression to POS. (Solution will be given during lecture.)

$$(A'.B.C + A.B'.C + A.B.C')$$

Combinational logic circuit design usually starts with a description of the problem. We call this *circuit description*. For example, we may be asked to design a three bits even number detector. A logic function described in this way can be designed directly from the canonical sum or product expression.

$$F = \sum_{N_2, N_1, N_0} (2, 4, 6)$$

This circuit can be synthesise directly using AND, NOT and OR gate. This is shown in Figure 2.1.

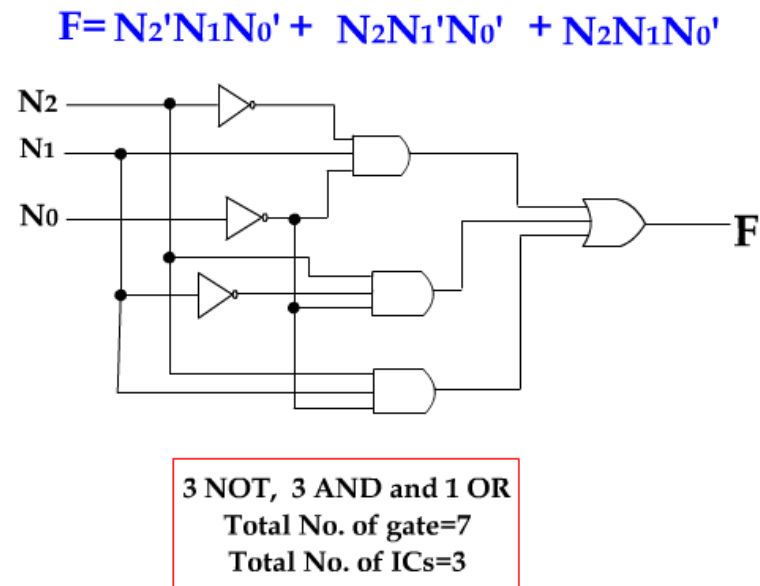
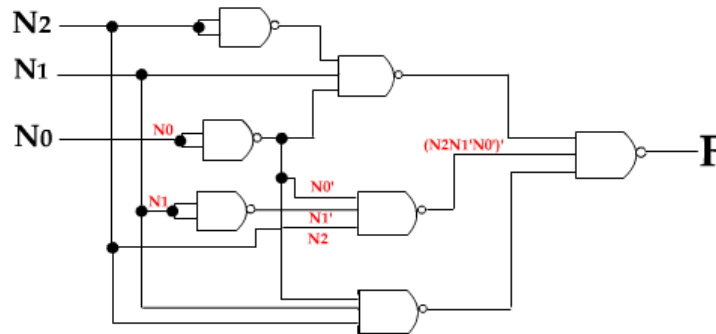


Figure 2.1 Even number detector using mixture of gates

However, for most technology, NAND and NOR gates are faster. Therefore, we may want to manipulate the equations so that the circuit consists of NAND or NOR gate only. This process is call *circuit manipulations*.

$$F = (N_2'N_1N_0' + N_2N_1'N_0' + N_2N_1N_0)''$$

$$= [(N_2'N_1N_0)' \cdot (N_2N_1'N_0)']'$$



7 NAND
Total No. of gate=7
Total No. of ICs=2

Figure 2.2 Even number detector using NAND gates only

Finally, in order to use minimal number of gates, we can perform a combinational *circuit minimisation* using K-map.

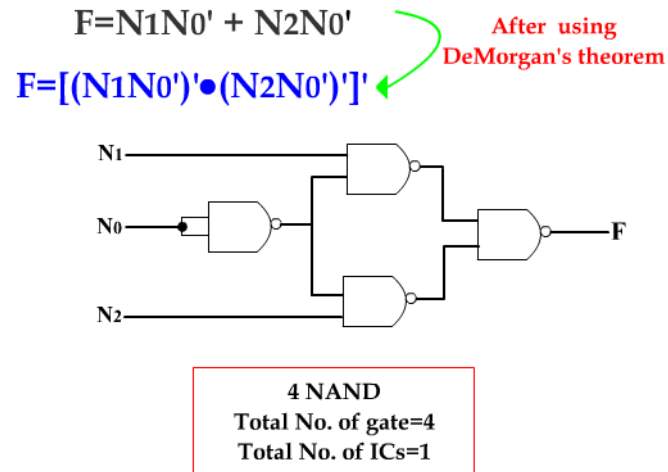
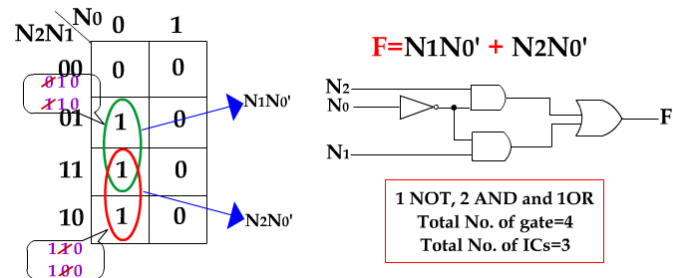


Figure 2.3 K-map and simplified circuit

Karnaugh Maps

The Karnaugh map (K-map) is a graphical device used to simplify a logic equation or to convert truth table to its corresponding logic circuit in a simple, orderly process. K-map's practical usefulness is limited to six variables. This topic will be limited to problems with up to four inputs, since five- and six-input problems are too involved and are best done by a computer program.

The labelling of input values on the K-map is done to assure that there is only one input variable that changes between adjacent cells. Two, three, four-variables K-maps are shown in Figure 2.4.

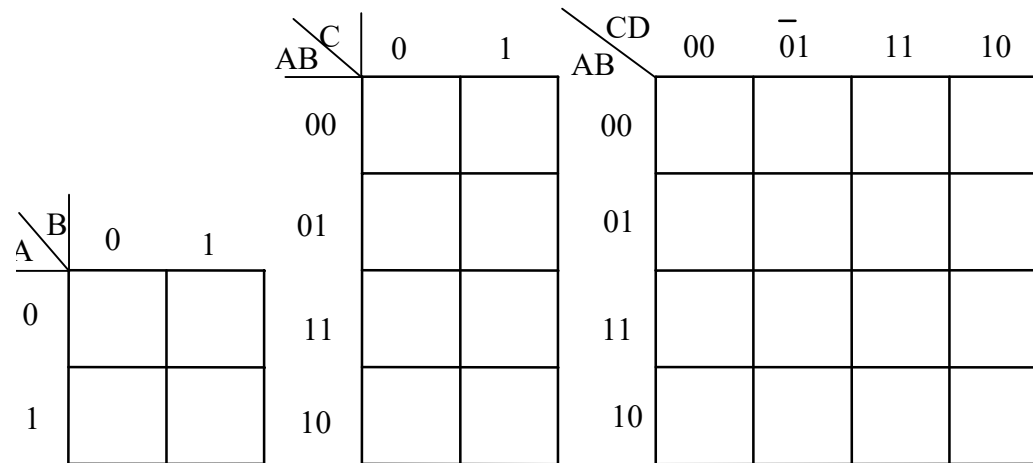


Figure 2.4 Two-variable, three-variable, and four-variable Karnaugh maps

Rules for Grouping Cells for Simplification:

You can choose to group the **1s** or the **0s** that are in adjacent cells according to the following rules, by drawing a loop around those cells:

1. Adjacent cells are cells that differ by only a single variable.
2. The **1s** or **0s** in adjacent cells must be combined in groups of 1,2,4,8,16, and so on.
3. Each group of **1s** or **0s** should be maximised to include the largest number of adjacent cells as possible in accordance with rule 2.
4. Every **1s** or **0s** on the map must be included in at least one group. There can be overlapping groups if they include noncommon 1s or 0s.

Example:

Simplified the equation $X = \prod_{ABC}(3,4,5,6,7)$ by circling the 0s.

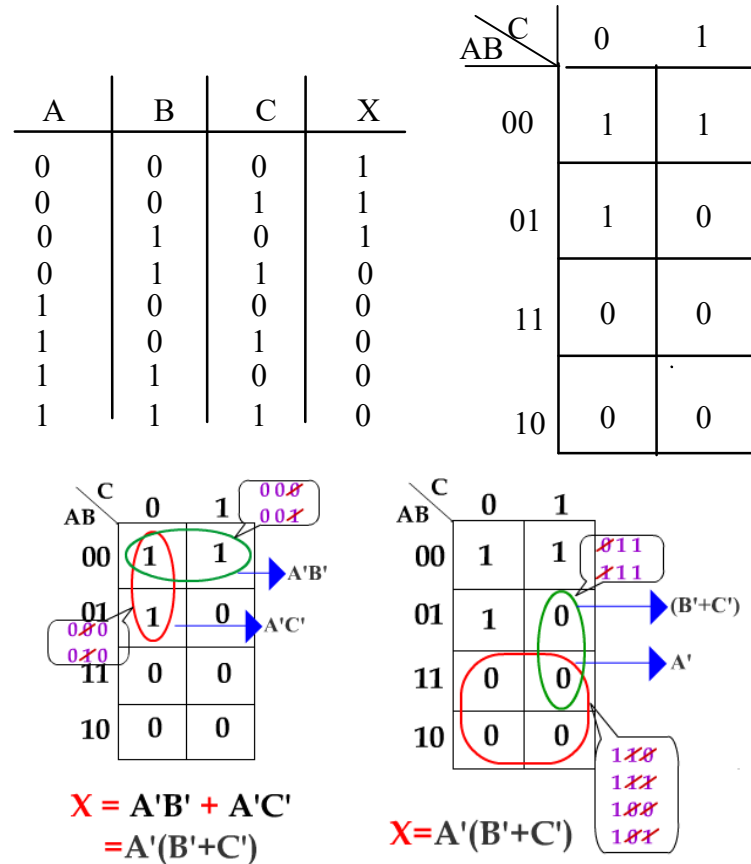
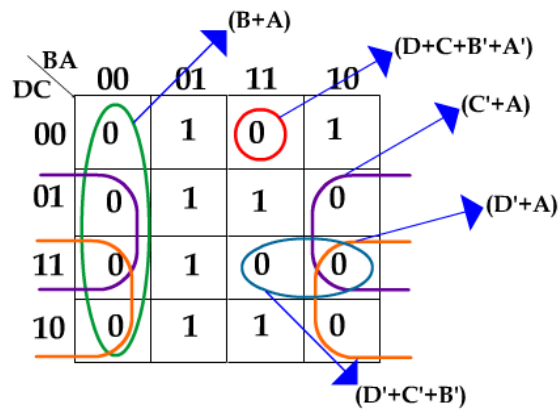


Figure 2.5 K-map with three input variable

Example:

Simplified the term from the K-map shown in Figure 2.6 by using the rules of grouping 0s cell for simplification and simplifying expression rules.

		BA			
		00	01	11	10
DC	00	0	1	0	1
	01	0	1	1	0
	11	0	1	0	0
	10	0	1	1	0



$$F = (B+A)(D+C+B'+A')(C'+A)(D'+A)(D'+C'+B')$$

Figure 2.6 Four inputs K-map

"Don't Care" Conditions

"Don't Care" condition is defined as input conditions for which there are no specified output level. It is usually because these input condition will never occur.

When the input combinations will not occur, the output states are filled in on the truth table and in the K-map as a X, and are referred to as don't care states. The "don't care" conditions should be considered as a 0 or 1 to produce K-map that yields the simplest expression.

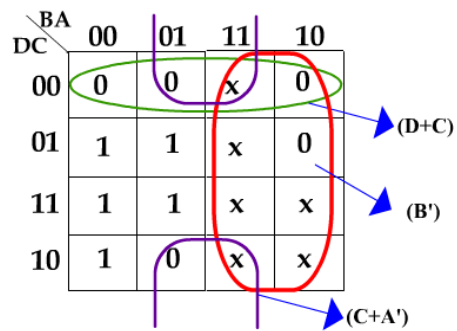
Example:

BA DC	00	01	11	10
00	0	0	x	0
01	1	1	x	0
11	1	1	x	x
10	1	0	x	x

BA DC	00	01	11	10	
00	0	0	x	0	→ CB'
01	1	1	x	0	
11	1	1	x	x	
10	1	0	x	x	→ DA'

F = CB' + DA'

	BA	00	01	11	10
DC		00	01	11	10
00		0	0	x	0
01		1	1	x	0
11		1	1	x	x
10		1	0	x	x



$$F = (D+C)(B')(C+A')$$

Figure 2.7 Using of don't care state

2.2 Case Study on Combinational Circuit Design

0to9decoder Design (See Lab 4)

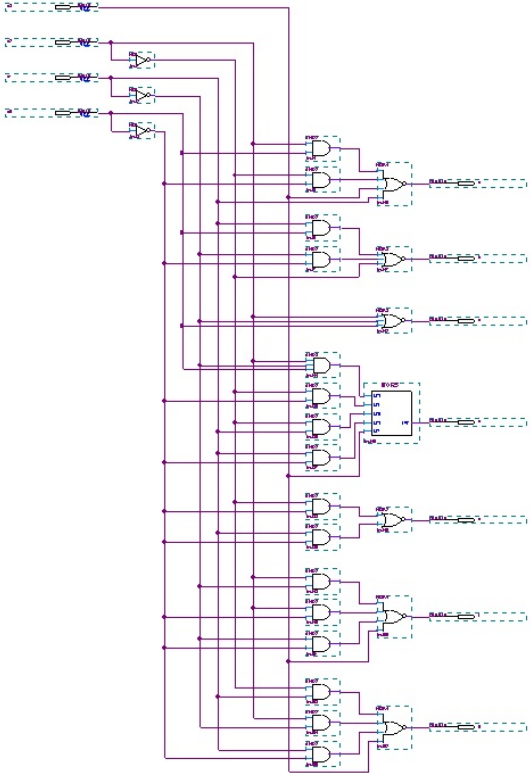


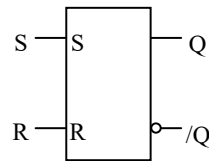
Figure 2.8 Sample *decoder (0to9decoder)* circuit

Finite State Machine Design

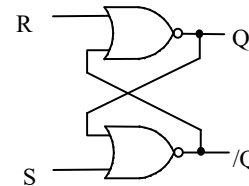
Latches and Flip-Flops

The most commonly used latches and flip-flops are:

S-R latch



Logic symbol



Logic circuit

S	R	Q	/Q	Operation
0	0	Q ₀	/Q	Hold (no change)
0	1	0	1	Reset
1	0	1	0	Set
1	1	?	?	Invalid (Ambiguous)

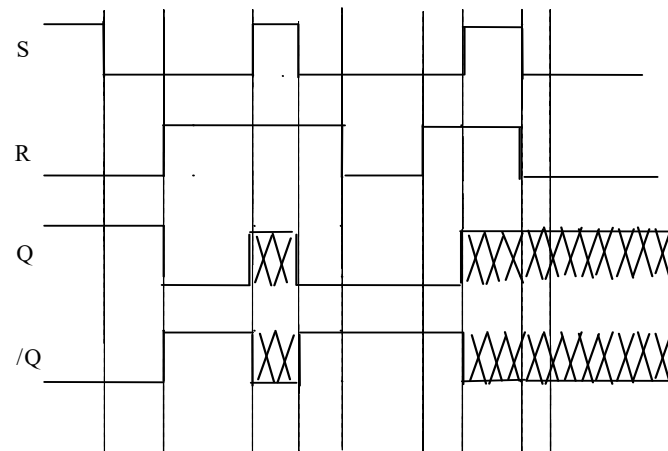
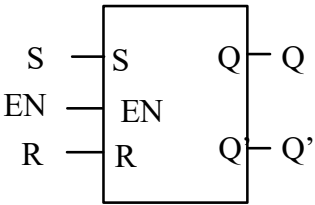
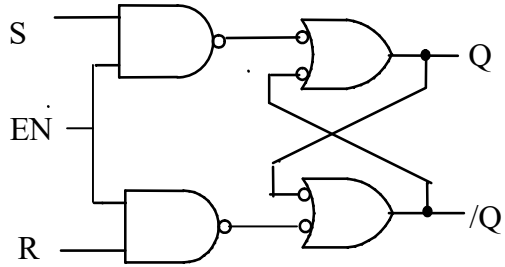


Figure 2.9 Logic symbol, logic circuit, function table and waveform for S-R latch

S-R latch with enable pin



Logic Symbol



Logic Circuit

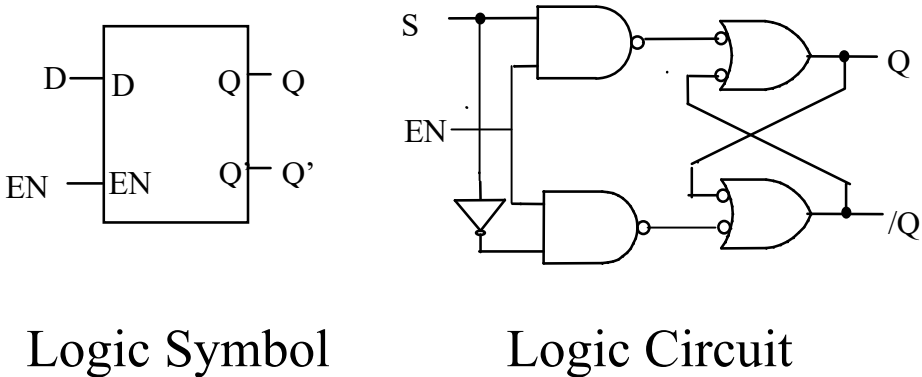
S	R	E N	Q	/Q	Operation
0	0	1	Q_0	/Q	Hold (no change)
0	1	1	0	1	
1	0	1	1	0	Reset
1	1	1	?	?	Set
x	x	0	Q_0	/Q	Invalid (Ambiguous) Hold (no change)

"X" indicates "don't care"

Function Table

Figure 2.10 Logic symbol, circuit and function table for S-R latch with enable pin

D latch



Logic Symbol

Logic Circuit

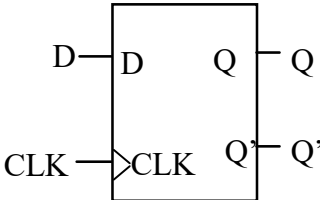
EN	D	Q	\bar{Q}
0	X	Q_0	\bar{Q}_0
1	0	0	1
1	1	1	0

"X" indicates "don't care"

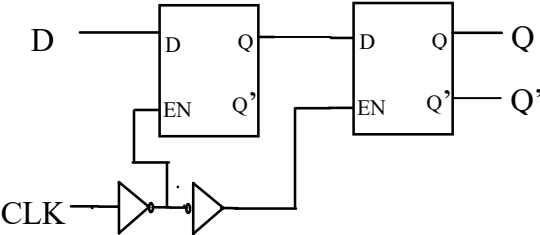
Function Table

Figure 2.11 Logic symbol, logic circuit and function table for S-R latch

Edge-triggered D flip-flop



Logic Symbol



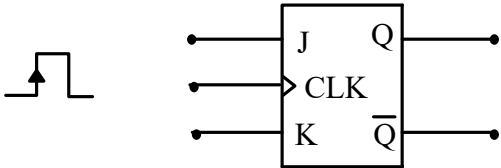
Logic Circuit

CLK	D	Q	\bar{Q}
↑ ↑ No positive clk edge	X	Q_0	\bar{Q}_0
	0	0	1
	1	1	0

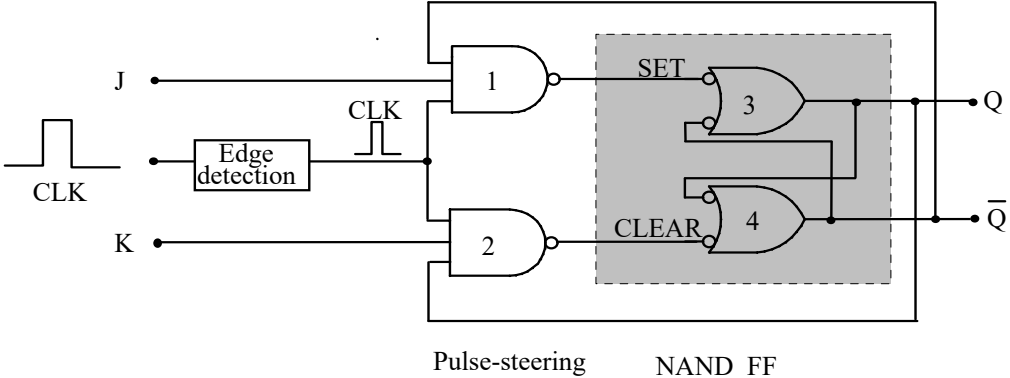
Function Table

Figure 2.12 Logic symbol, logic circuit and function table for D flip-flop

Edge-triggered J-K flip-flop



Logic Symbol



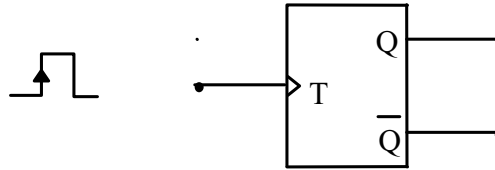
Logic Circuit

J	K	CLK	Q (output)
0	0	↑	Q_0 (no change)
0	1	↑	0 (reset)
1	0	↑	1 (set)
1	1	↑	\bar{Q}_0 (toggle)

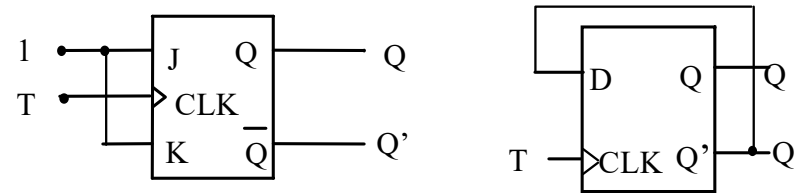
Function Table

Figure 2.13 Logic symbol, logic circuit and function table for JK flip-flop

T flip-flop



Logic Symbol

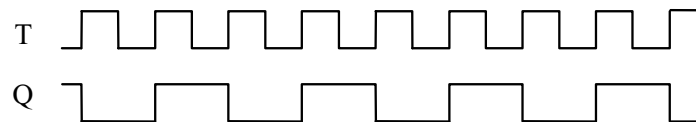


Two possible logic circuit for T flip-flop

↑

T	Q	\bar{Q}
No positive edge	Q_0	\bar{Q}_0
	\bar{Q}_0	Q_0

Function Table



Waveform for T flip-flop

Figure 2.14 Logic symbol, circuit and function table and waveform for T flip-flop

Presetting and Clearing of flip-flops and latches

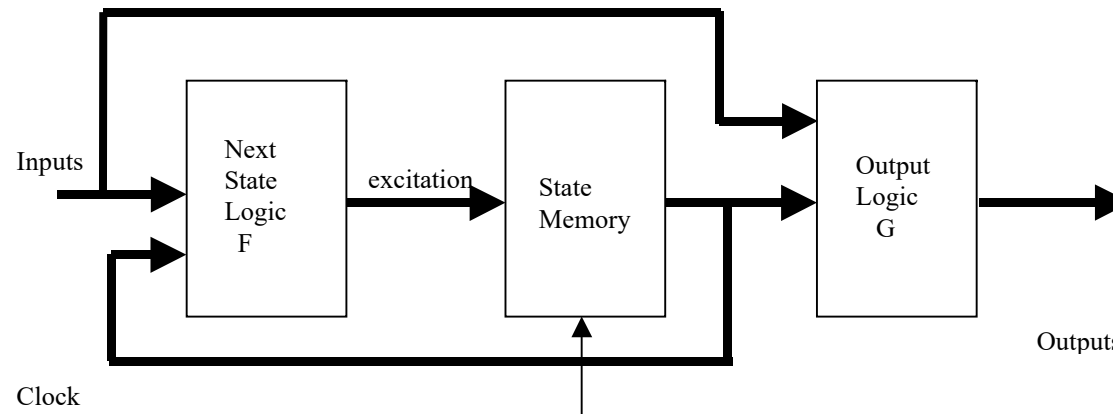
Most flop-flops and latches come with a Clear and Preset pins for the user to do asynchronous set or reset. The Clear and Preset pins can be active HIGH or active LOW.

Clocked synchronous State Machines

"State machines" is a generic name given to a sequential circuit; "clocked" refers to the fact that their storage elements (flip-flops) employ a clocked input; and "synchronous" means that all of the flip flops use the same clock signal.

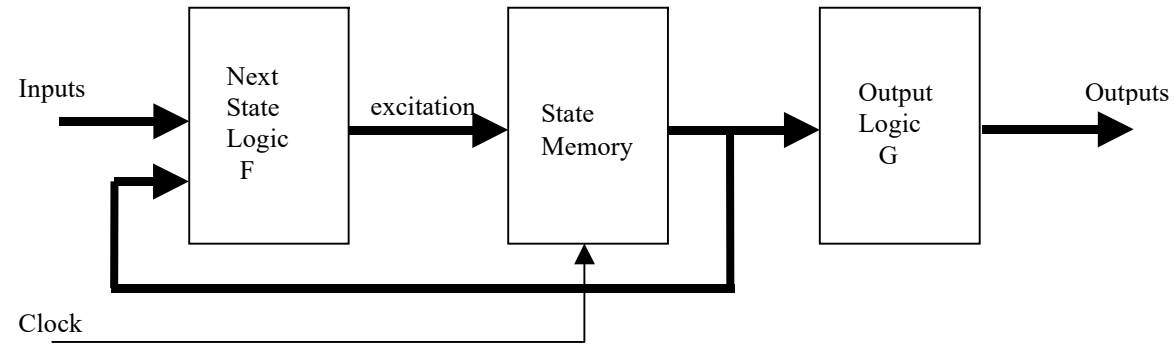
In state machines circuits, all changes at the output will take place under the control of a periodic sequence of pulses called a clock. Each clock pulse will permit the circuit to either remain in the present state (present set of flip-flop values) or move to another state (a new set of flip-flop values). The advantage of clocked sequential circuits is that glitches that occur due to the imperfect nature of the logic devices will have no effect. However, to have this advantage, we must choose the clock period such that it is longer than the worst multiple delay paths.

There are two main types of State machine, Mealy machine and Moore machine. For a Mealy machine, the outputs are function of both the inputs and the current states. For Moore machine, outputs are function of only the current states. Figure 3.17 show the structures of Mealy and Moore machines.



Mealy Machine

Next state and output logic blocks are combinational logic blocks. State memory is a sequential logic block.



Moore Machine

Figure 2.15 Structures of Mealy and Moore Machines

Clocked Synchronous State-Machine Design

The characteristic equations for various devices are listed below. These equations are needed when we analyse the state machine. Q^* denotes the next state.

S-R latch/flip flop	:	$Q^* = S + R'.Q$
D latch/flip flop	:	$Q^* = D$
J-K flip flop	:	$Q^* = J.Q' + K'.Q$
T flip flop	:	$Q^* = Q'$
T flip flop with enable:	:	$Q^* = EN.Q' + EN'.Q$

The synthesis (design) of the sequential circuits consists of obtaining a table of diagram for the time sequence of inputs, outputs and internal states. Detail steps are as follows:

- (1) Construct a state/output table corresponding to the word description or specification, using generic names for the states.
- (2) (Optional) Minimise the number of states in the state/output table.
- (3) Choose a set of state variables and assign state-variable combinations to the name states.
- (4) Substitute the state-variable combinations into the state/output table that shows the desired next state-variable combination and output for each state/input combination.
- (5) Choose a flip-flop type (e.g., D or J-K) for the state memory. In most cases, you'll already have choice in mind at the outset of the design, but this step is your last chance to change your mind.
- (6) Construct an excitation table that shows the excitation values, required to obtain the desired next state for each state/input combination.
- (7) Derive excitation equations from the excitation table.
- (8) Derive output equations from the state/output table.
- (9) Draw a logic diagram that shows the state-variable storage elements and realises the required excitation and output equations.

Design of a Moore machine, two-bit up/down counter which output, $Z= '1'$ when count = '11'

The outputs for the Moore-type circuits are independent of the inputs, i.e. the outputs are functions of the present state only. The Moore outputs change their values only when the state changes because of a change of the inputs. The figure below shows an example of a moore state diagram and state table. (Note synchronous counter is a special type of synchronous machine, the state variables themselves are the outputs of the state machines.)

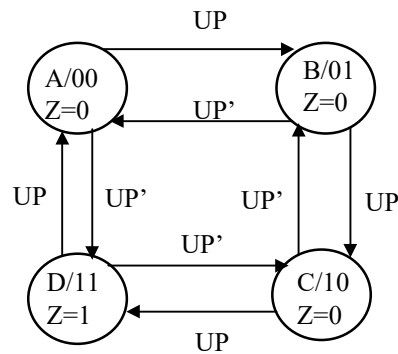


Figure 2.16 Two-bit up/down counter, Moore Machine

The state diagram above represents a synchronous circuit with four states, A, B, C and D, and an input variable, UP. In each state it is necessary for the circuit to be able to determine which state it is in and what the current value of UP is, and then to set up the FF inputs such that the correct state is entered when the clock input occurs. The arrows connecting the states represent the occurrence of a clock input and the variables alongside the arrows show the input condition that causes that path to be followed.

Present State Q1Q0	Next State Q1*Q0*		Output Z
	UP=0	UP=1	
A,00	D,11	B,01	0
B,01	A,00	C,10	0
C,10	B,01	D,11	0
D,11	C,10	A,00	1

Table 2.2 State/Transition/Output table

The implementation of a sequential circuit with n states will require m FFs where $2^m = n$. The outputs of these FFs are called the state variables and are used to identify which state the circuit is in.

The next step is to decide what FF to use. For this example, both approaches are used.

Using D FF

We can determine the next-state equations for each of the two state bits. $Q1^*$ and $Q0^*$ represent the values of the next state function. The present state values are represented by $Q1$ and $Q0$. The karnaugh maps can be generated easily from the state transition table. Each row corresponds to a state and each column corresponds to a combination of the inputs; the entries in the karnaugh maps correspond to the values of $Q1^*$ and $Q0^*$ in the transition table.

The characteristic equation for D flip-flop is $Q^* = D$. Therefore the karnaugh map can be used to find the minimal equations at the input of D flip-flops. The output equation can also be obtained from the state/transition/output table. In this case, there is no need to draw a karnaugh map for the output equation because only two variables are involved.

Output equation: $Z=Q1.Q0$

		UP	
		0	1
Q1Q0	00	1	0
	01	0	1
	11	1	0
	10	0	1

$$D1 = Q1'.Q0'.UP' + Q1'.Q0.UP + Q1.Q0'.UP + Q1.Q0.UP'$$

		UP	
		0	1
Q1Q0	00	1	1
	01	0	0
	11	0	0
	10	1	1

$$D0 = Q0'$$

Figure 2.17 Next state and output equations for the counter

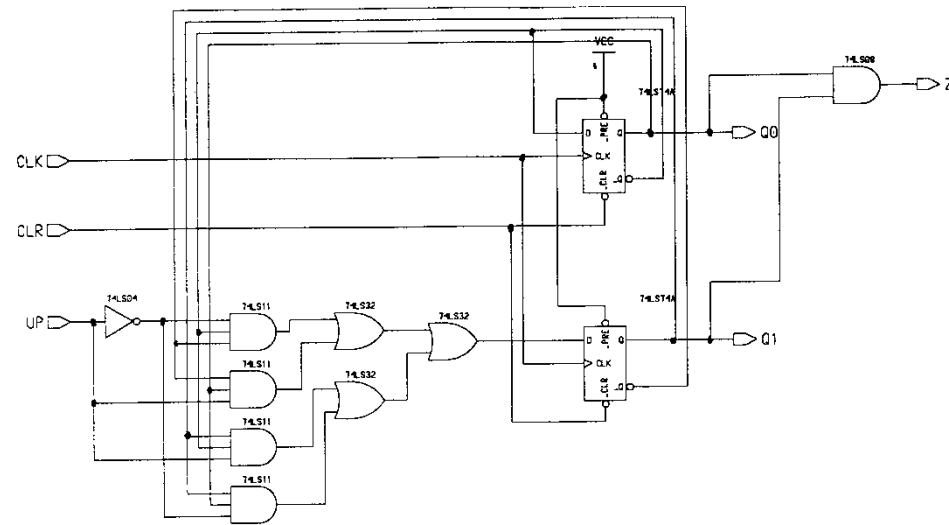


Figure 2.18 Circuit diagram using D FF

Design of a Mealy machine, two-bit up/down counter with UP pin and output, $Z = 1$ when count '11' and $UP = '1'$

We begin the design process by constructing a state diagram to meet these requirements. By assigning state A to '00', B to '01', C to '10' and D to '11', we arrive at the state diagram below:

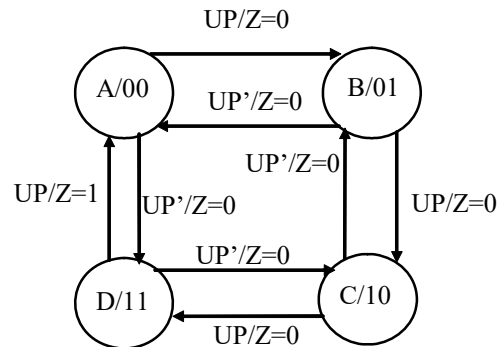


Figure 2.19 Two-bit up/down counter (Mealy Machine)

The state table for this sequence detector can easily be constructed from the state diagram.

Present State Q1Q0	Next State Q1*Q0*,Output Z	
	UP=0	UP=1
A,00	D,11,0	B,01,0
B,01	A,00,0	C,10,0
C,10	B,01,0	D,11,0
D,11	C,10,0	A,00,1

Table 2.3 State/Transition/Output table for two-bit up/down counter (Mealy Machine)

Before we can derive the flip-flop input equations, we must specify the type of flip flop to be used in the design. For this example, let us use D flip-flops. (Note: the K map is exactly the same shown in Figure 2.17. The only difference is the output equation which is now a function of both input and the states.)

		UP	
		0	1
Q1Q0	00	0	0
	01	0	0
	11	0	1
	10	0	0

$$Z = Q1.Q0.UP$$

Figure 2.20 Kmap for output equation

From equations derived from Figure 2.17 and Figure 2.20, we can draw the circuit for the state machine.

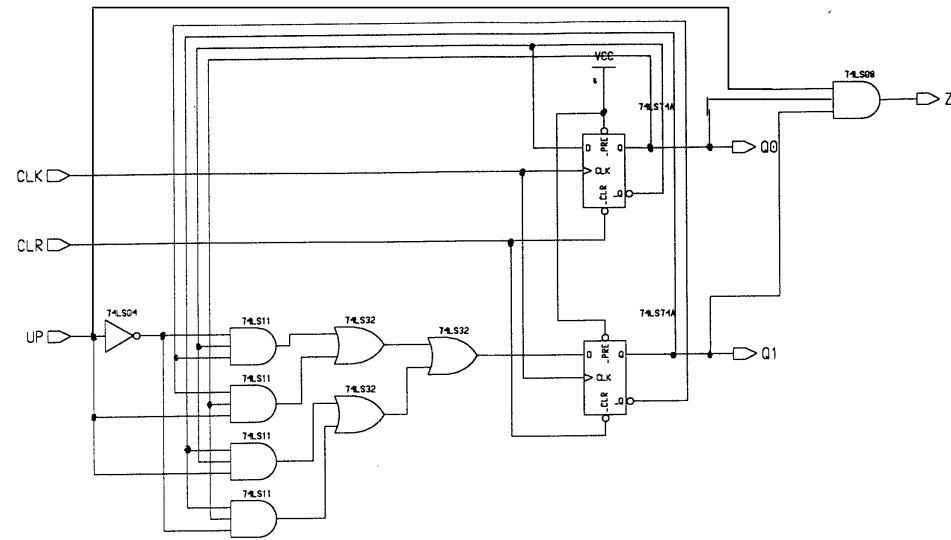


Figure 2.21 Circuit for the Mealy Machine

2.3 Case Study on Finite State Machine Design

0to9counter Design (See Lab 3)

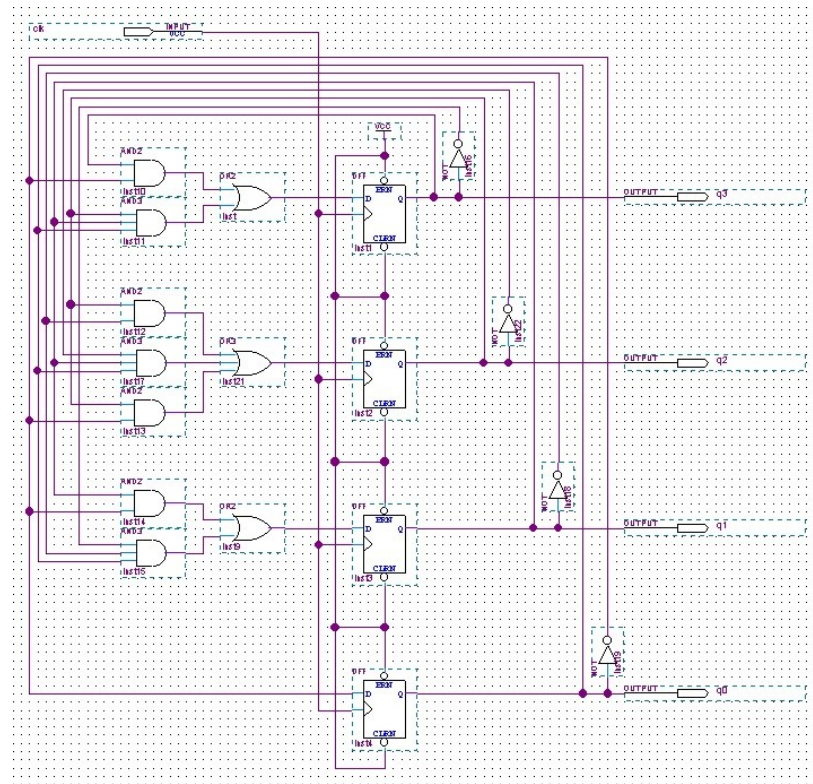


Figure 2.22 Sample counter (*0to9counter*) circuit

2.4 Digital Library Modeling and Development

Library Modeling

Level of Abstractions

In general, four different types of objects can be identified in the design process of electronics system: transistors, gates, registers and processor components. These abstractions are summarised in Table 2.4.

Level	Behavioural Forms	Library	Physical Objects
Transistor	Differential equations, Current-voltage diagrams	Transistors, resistors, capacitors	Analogue and digital Cells
Gate	Boolean equation, Finite-state machines (FSM)	Gates, Flip-flops	Modules, Units
Register	Algorithms, flowcharts, instruction sets, generalised FSM	Adders, comparators, registers, counters	Microchips
Processor	Executable specification, programs	Processors, controllers, memories, ASICs	Printed-circuit boards, multi-chips modules

Table 2.4 Level of Abstraction

Transistor Level - The main components in the Library for the transistor level are transistors, resistors and capacitors, which are combined to form analog and digital circuit that can satisfy a given functionality. This functionality is usually described by a set of differential equations or by some type of current-voltage relationships. The physical representation of these analogue and digital circuits, called cells, would consist of transistor-level components and the wires that connect them.

Gate Level - The main components on the gate level of abstraction are logic gates and flip-flops. Logic gates are special circuits that perform Boolean operations which are similar to conjunctions in the English language such as *or* and *and*. These cells can be grouped and placed on the silicon surface to form arithmetic and storage modules or units that are used as the basic components in the register level. These units are described behaviourally by means of Boolean equations and finite-state-machine (FSM) diagrams.

Register Level - The main components on the register level of abstraction are arithmetic and storage units, such as adders, comparators, multipliers, counters, registers, register files, queues and data paths. Each of these register components is a module or unit which has fixed dimensions, a fixed propagation time and a fixed position for the inputs and outputs located on its boundary.

These register components can be assembled and interconnected into microchips, which are used as the basic components on the next-higher level of abstraction. In general, these microchips are described by flowcharts, instruction sets, generalised FSM diagrams or state tables.

Processor Level - The highest level of abstraction presented in Table 2.4 is called the processor level, since the basic components on this level are processors, memories, controllers, and interfaces, in addition to the custom microchips called application-specific integrated circuits (ASIC). Generally, one or more of these components are placed on a printed-circuit board and connected with wires that are printed on the board.

In some cases we can reduce the dimensions of the board by using a silicon substrate instead of a printed-circuit board to connect the microchips, in which case the package is called a multi-chip module. The systems that are composed from these processor-level components are usually described behaviourally by either a natural language, e.g. a hardware description language, or an algorithm written in a programming language.

Design Process

During the design process of an electronic digital system, any one of these levels of abstraction may be used one or more times, depending on how many different goals, technologies, components, libraries and design alternatives we want to explore.

We must choose carefully an efficient design methodology that determines the proper subset of abstraction levels, synthesis tasks at the order in which they are executed, and the type of CAD tools available during the execution of each task in the design process.

The most popular methodology consists of building a library for a certain abstraction level, then using synthesis to convert a behavioural description into a structure that can be implemented with the components from this library.

In real practice, the design process is always heavily influenced by the following factors:

- the nature of the product being designed
- how soon the product must be brought to market
- the particular technology used for its manufacture
- the company's organisational structure
- the design team's experience
- the availability of CAD tools
- the amount of budget allocated.

In general, the design process can be divided into the following steps: Design Specification, Library Development and Design Synthesis, Design Analysis, Documentation and Manufacturing.

Library Development

Library development is one the most important step in any design process. It has great impact on the final design.

Once the high-level block diagram has been developed in the specification phase, it must be iteratively refined or decomposed into smaller components. The goal of this process is to ensure that the product contains nothing but the predefined components from a library of components that has been characterised for a particular manufacturing technology.

Different EDA tools require different information for the database it is working on. These information are usually represented in the form of library/model/ rules. Some of these library/model/rules are generic in nature (e.g. the behaviour of an NAND function, or a pentium), but many of them are specific to a particular foundry. (e.g. the propagation delay of a NAND gate from CSM most likely will be different from a NAND gate from TSMC).

The component in the library must be designed, tested, and fully documented so that designer can use them without having to analyse their structure. The content of EDA library typically include the followings:

- The component's functionality, the names of inputs and outputs and the typical application i.e. relation between the outputs and inputs.
- The component's physical dimensions, the position of inputs and outputs and the packaging information i.e. Physical device parameters: e.g. parameter of a transistor, resistor, area etc
- The electrical constraints, the power supply requirements, the current and voltage ranges that are allowed at the inputs and outputs and the heat dissipation.

- The voltage waveforms for inputs and outputs, the timing relationships between them, and the critical delays from inputs to outputs eg propagation delay, set-up/hold timing requirement etc.
- The components models to be used by CAD tools for simulation, synthesis, physical design and testing.
- The symbol and abstract of the components ie graphical representation of the model and the layout of the model.
- Wire load Model

Digital Library Modelling

Definition of timing parameters:

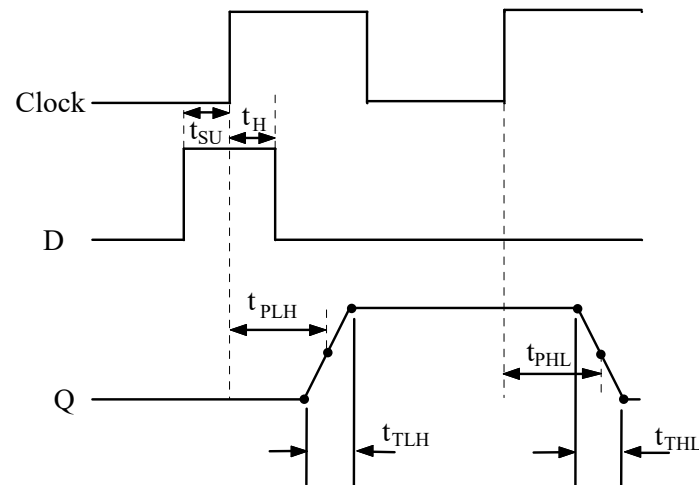


Figure 2.23 Flip-Flop timing parameters

Propagation Delay (t_{PHL} and t_{PLH})

The amount of time it takes for the output of the flip-flop to change its state from a clock trigger or asynchronous set or reset. It is defined from the 50% point of the input pulse to the 50% point of the output pulse.

t_{PHL} : The propagation time from HIGH to a LOW.

t_{PLH} : The propagation time from LOW to HIGH.

Output Transition Time

The output transition time is defined as the rise time or fall time of the output.

Rise Time (t_{TLH}) : is the 10% to 90% time, or LOW to HIGH transition time.

Fall Time (t_{THL}) : is the 90% to 10% time, or HIGH to LOW transition time.

Setup Time (t_{SU})

The time interval immediately preceding the active transition of the clock pulse during which the control or data inputs must be stable (at a valid logic level).

Hold Time (t_H)

The amount of time that the control or data inputs must be stable after the clock trigger occurs.

Removal Time (t_{rem})

The time between the end of an overriding asynchronous input, such as clear or reset and the earliest allowable beginning of a synchronous clock input.

Operating Frequency (f_{max})

The maximum input clock frequency of the clock input to the flip-flop. If f_{max} is not listed, an approximate maximum frequency of operation can be found by taking the reciprocal of the worst case average propagation delay time $(t_{PHL} + t_{PLH}) / 2$. If the calculated f_{max} is larger than the listed f_{max} , always used the lower value for f_{max} .

Content of Digital Library

The content of digital library is as follows:

- Function : Boolean equation (e.g. $!(A B)+C$)
State table (e.g. state table for Flip-Flop)
- Signal Flow Direction : Input, output or bi-directional
- Loading : Capacitive Load at input and output
- Unateness : Positive, negative or unknown
- Timing Arc : Logic 0 to logic 1 propagation delay
Logic 1 to logic 0 propagation delay
Logic 0 to hi-Z propagation delay
hi-Z to logic 1 propagation delay
Logic 1 to hi-Z propagation delay
hi-Z to hi-Z propagation delay
Logic 0 to Logic 1 transition time
Logic 1 to Logic 0 transition time
- Timing Requirement : Setup time, Hold time
Minimum pulse width

- Operating Condition : Derating in performance for Voltage variation;
Derating in performance for Temperature variation;
Derating in performance for foundry process variation
- Wire load : Wire length estimation

Digital Library development/Characterisation of timing model

Process of a library development for a timing model is as follow:

- Perform spice simulations using spice model from the foundry.
- Measure the performance from the simulation result.
- Design test structure for fabrication.
- Co-relate silicon performance with spice simulation.

Timing Library development

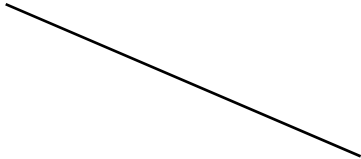
A more complete library must be characterised at different operating condition and different input condition. Most EDA tools adopted a table look-up approach for estimating delay. A 2-D table model is most commonly used for deep-sub micron technology. The delay is a function of input skew and the total load at output.

Example of a 2-input NAND Gate:

1. Measure delay and output transition from input pin A to output pin Y for different output load and input transition.
2. Repeat (1) for input pin B.
3. Repeat (1) and (2) for different operating condition.

Sample Results for a inverter output Pin Y w.r.t Pin A at typical corner (operation conditions):

Propagation delay for 0 to 1 transition



Output				
Load (pf)				
Input fall time (ns)	0.01	0.04	0.14	0.54
0.08	0.097	0.174	0.525	1.607
0.60	0.160	0.257	0.611	1.686
1.20	0.199	0.317	0.721	1.785
3.00	0.258	0.425	0.954	2.113

Propagation delay for 1 to 0 transition



Output Load (pf) Input rise time (ns)	0.01	0.04	0.14	0.54
0.08	0.081	0.136	0.391	1.178
0.60	0.162	0.242	0.510	1.293
1.20	0.216	0.322	0.650	1.430
3.00	0.321	0.476	0.942	1.860

Despite taken into consideration of many important factors in library characterisation, sometime we still are unable to get very good delay estimation in the EDA tool.

Possible causes:

1. For deep sub-micron technology, the delay due to wire load is significant (for 0.35um, 40% is not uncommon). Tools like logic synthesis, pre-layout simulator can only use estimated wire load.
2. Timing is characterised only at a few input conditions/operating conditions. Need to be interpolated or extrapolated.

2.5 Digital Simulation

Digital simulation allows you to model the behaviour of a circuit in terms of logical values of signals and the timing of events, before actually building. Digital Simulators are mainly events driven simulators, they calculate and store changes on signals (or nodes) in a circuit's network.

Digital simulators do not calculate the precise voltage or current values for nodes in a circuit, they only have to track the changes to the state of each node in the system. These states are typically a combination of logic levels (e.g. 0, 1 or unknown) and signal strength (strong, resistive, high impedance or indeterminate).

There are many factors that influence the accuracy of the digital simulators including:

- The number of logic states defined by the simulators
- The method used to model delays
- The method used (if any) to model the effect of fan in, fan out, loading, temperature and power

When using a digital simulator, you apply stimulus to a software model of a circuit and view the resulting circuit outputs. Stimulus can be provided to the simulator in many forms including waveforms, simulator commands, test vector etc. Similarly the output can be in many forms including text window, waveform etc. Simulating a digital circuit requires the stimulus and the circuit software model. To understand simulation, it is useful to think analogies to actual hardware, you need the circuit on the breadboard and the necessary inputs to test the circuit. Figure 2.24 shows how a conceptual design, consisting of a collection of component devices, is processed during simulation.

The simulator accepts test stimulus and applies this stimulus to the circuit. The simulator then model your design using the models or components that you have used in this circuit and reports how the circuit reacts to the stimulus over time. The users can choose either to look at the response at the final outputs or/and the response at the intermediate nodes within the circuit.

Advantages of doing a digital simulation

The reasons for doing digital simulation are as follows:

- Simulation allows you to experiment with components that you don't actually have available.
- Simulation saves time. Once you have become familiar with the simulation methods, you will find that setting up and running a simulation takes far less time than a hardware prototype.
- Simulation gives you greater control and you can observe the inputs and outputs of each component in the circuit.
- Simulation allows you to modify the design easily.

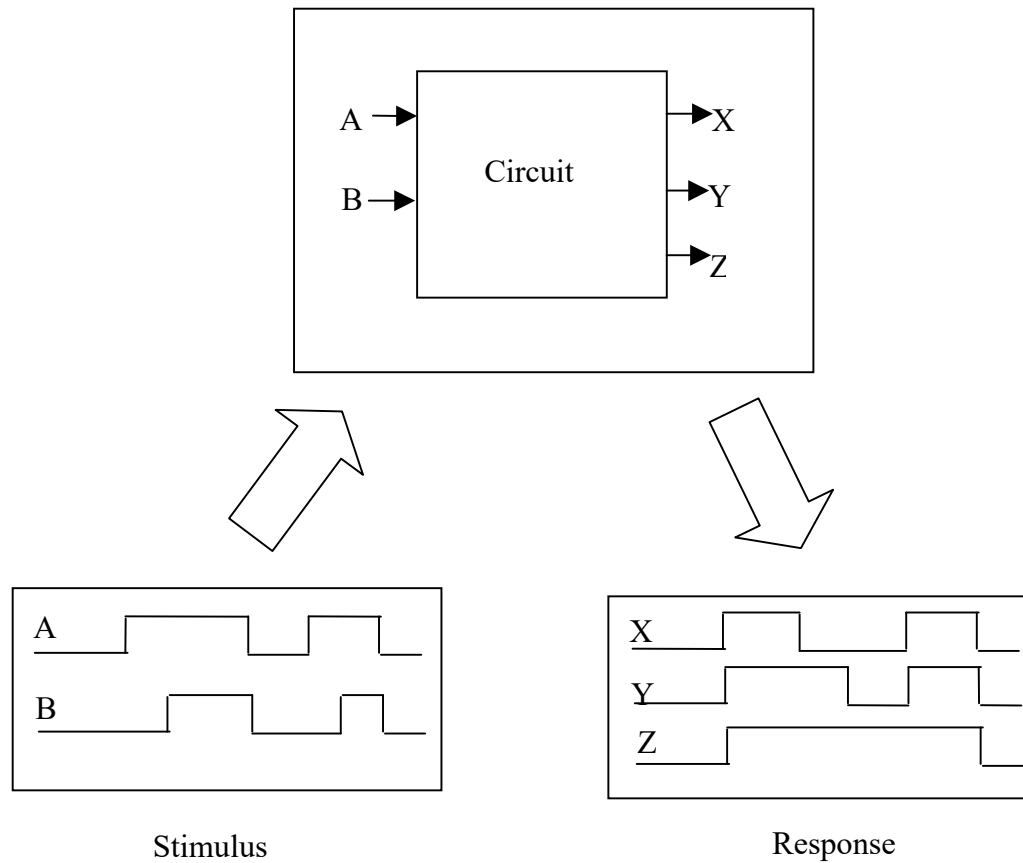


Figure 2.24 Basic flow during simulation

Timing Modes

Each mode consists of settings that make simulation speed and accuracy tradeoffs. You can set the mode for the design when you invoke, or you can use individual commands on individual instances. The timing modes are as follows:

Functional Simulation – Zero delay

In Functional simulation the simulator helps you to determine if the basic operation of the circuit is correct, without regard to the actual operating speeds and internal delays. Zero delays between events are assumed.

Functional Simulation - Unit delay

This mode provides high performance at the expense of accuracy. You can use this mode when debugging design functionality, find out the race condition and glitches and other circuit errors that results from such condition. With unit delay, a single delay time is assigned to every gate in the design i.e. all output and I/O pins have a rise and fall delay of one time step and all input pins have a rise and fall delay of zero time step. The simulator ignores all technology files.

Timing Simulation

In Timing simulation, the simulator actually calculates and models the delays between events, based on the knowledge that it has about the propagation delays on signals, gate switching delays, rise and fall times and other factors. It determines how your design will behave when it is operated at speed in the final implementation.

Timing Simulation - Linear timing

This mode provides straight-line approximations of the timing that is defined by the associated technology files. In this mode, you can debug the effects of timing on your design's functionality, but only if the components include linear technology files. In this mode, timing is computed significantly faster than when you use full technology files.

Timing Simulation - Linear timing with constraint checking

This mode provides straight-line timing approximations with full constraint checking. You can use this mode to produce timing violation messages. As with the linear timing mode, your components must provide linear technology files. The next step is to use full timing with constraint checking.

Timing Simulation - Full timing

This mode provides timing accuracy without timing constraint checking. This mode simulates the effects of timing on the design logic. Full timing consists of the min, typical, or max values from all technology file-specified delay equations, rise and fall pin delays, and BLM and VHDL delay instructions. Simulating a design with complete timing will result in a more accurate reflection of the actual design's performance. A simulator that models actual gates and signal delays has a lot more to keep track of. Full timing simulation is used only when you have decided on the final implementation technology.

Timing Simulation - Full timing with constraint checking

This mode provides complete timing accuracy with full constraint checking. You can use this timing mode to produce timing violation messages during full-circuit debugging operations. This mode uses the min, typical, or max values from all technology file-specified delay equations, rise and fall pin delays, and BLM and VHDL delay instructions; it also checks for constraints and spike, contention, and hazard violations.

Logic Values and Drive Strengths

There are several signal states. Each signal state is a combination of a logic value and a drive strength. Logic value and drive strength are defined as follows:

Logic value is a Boolean value that indicates the level of a signal. The logic values are:

- '0' A low signal level
- '1' A high signal level
- 'X' An unknown signal level

Drive strength is a value that allows the simulator to resolve signal contention and to simulate effects of different technologies. The drive strengths are:

- S A strong signal strength
- R A resistive signal strength
- Z A high impedance signal strength
- I An indeterminate signal strength

2.6 Hierarchical Design and System Partitioning

Hierarchy

The use of hierarchy or “divide and conquer”, involves dividing a system/module into sub-modules and then repeating this operation on the sub-modules until the complexity of the sub-modules is at an appropriately comprehensible level of detail. This approach is sometimes referred to as design partition and sub-partition of the modules. Partition of the design in general is based on functionality. Example of design partition may be based on digital and analog blocks or digital blocks with distinct functions.

Partitioning

Partitioning can also be done based on combinational logic and sequential logic blocks. Within the main blocks, sub-partition is done on the block and this approach is to descend into the hierarchy until the level where modules are defined in terms of standard functions like adder, multiplexer, comparators, registers etc.

Regularity

Hierarchy involves dividing a system into a set of sub-modules/sub-blocks. However, hierarchy alone does not necessarily solve the complexity problem. For instance, the repeatedly division of the hierarchy of a design into different sub-modules but will still end up with a large number of different sub-modules/sub-blocks.

With regularity as a guide, the designer attempts to divide hierarchy into a set of similar building blocks. The use of iteration to form arrays of identical cells/sub-blocks is an illustration of use of regularity in a system/IC design. This allows the reuse of sub-blocks. Regularity can exist at all levels of the design hierarchy.

Regularity allows an improvement in productivity by reusing specific blocks/designs in a number of places, thus reducing the number of different designs that need to be completed.

Modularity

The tenet of modularity adds to hierarchy and regularity the condition that sub-modules have well defined functions and interfaces (interconnections with other modules). The interconnections, along with functionality must also be defined in an unambiguous manner.

Modularity helps the designer to clarify and document an approach to a problem and also allows a design system to more easily check the attributes of a module as it is constructed. The ability to divide a task into a set of well-defined modules also helps in a huge design where different modules are design by different engineers.

The correct decision regarding modularity allows one to break up a system into blocks with confidence that when the blocks are re-combined, the system will function as specified.

Locality

Locality usually refers to “time locality”; that is to mean to pay attention to clock generation distribution network of the system. The approach is to ensure that timing critical sub-blocks (share the same clock for synchronisation) must be kept as close as possible.

If a system have several analogue and digital blocks, it would be necessary to physically group the analogue and digital blocks separately and thus ensuring system having better noise immunity as high speed digital circuit generate noise in the power lines.

2.7 Digital Design Implementation

Overview

Once a design has been created, simulated, and synthesized, the next step is implementation of the design into the particular complex electronic device. Figure 2.25 shows a sample implementation process for complex electronic device. Usually the implementation process uses the tools supplied by the device (e.g., FPGA) vendor. The functions that were defined in the design have to be matched to the available blocks, gates, and other logic elements on the chip. Some basic steps in implementing a design are:

- Floorplan
- Translate
- Map
- Place and Route

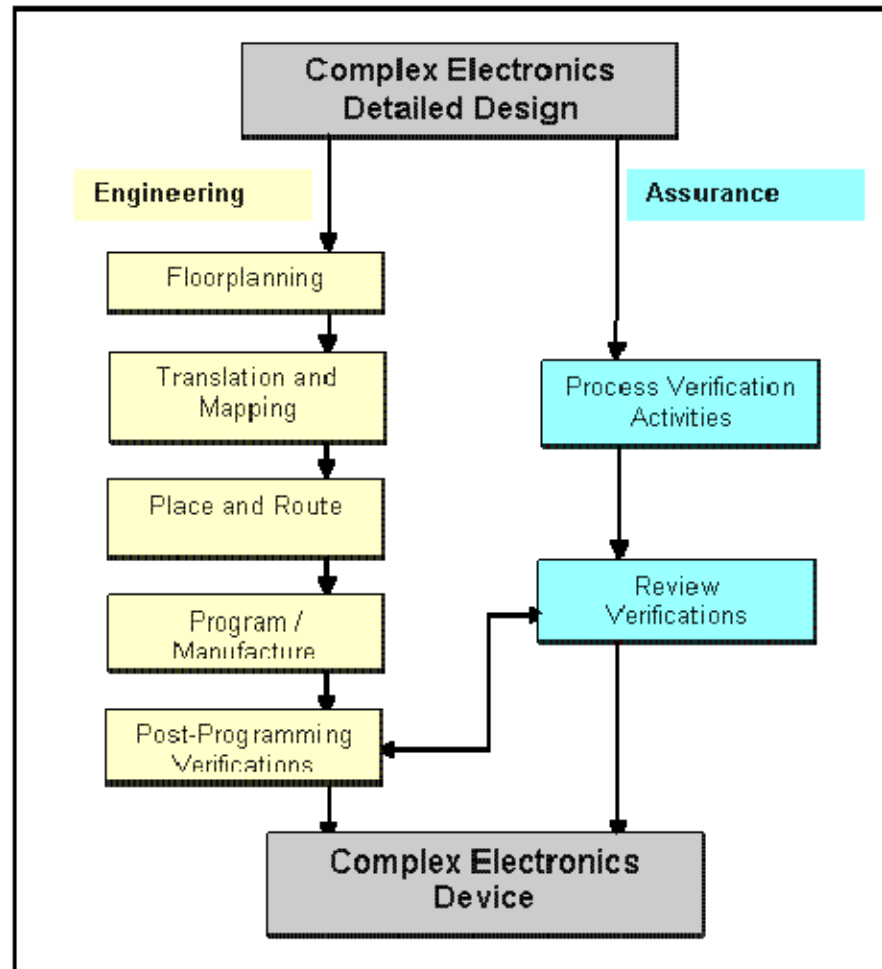


Figure 2.25 Implementation Process for Complex Electronics Device

Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them. In designing complex electronics, there are multiple goals that must be met, and the goals often conflict. Finding the best balance between the various goals and requirements is something of an art. Some goals are:

- Minimize space on the chip (allows choice of less costly chips)
- Meet or exceed required performance
- Place everything close to everything else to minimize transmission time in the signal paths

Translation involves converting the results of the synthesis process to the format supported internally by the vendor's place-and-route tools. The incoming netlist is checked for adherence to design rules and is then optimized for the chip.

Translation may also be referred to as compilation or compiling. This process is automatic, but it takes some wading through the reports produced by the tool to verify that the translation/compile was correct. An intelligent post-processor, rather than the designer (or worse, the quality assurance engineer), should be used to find syntax and binding errors - otherwise you will have to do this for each design modification!

Mapping takes the logic blocks and determines what logic gates and interconnections on the device should be used to implement those blocks. During the mapping step, the functions within the device (such as counters, registers, or adders) are aligned with the logic resources of the chip. The exact process is device dependent. For example, FPGAs have look-up tables that perform logic operations. The mapping tool (part of the vendor's tool suite) collects the gates defined by the netlist into groups that will fit within the look-up tables.

Place and Route is the process of placing the logic blocks in the best spots on the chip to achieve efficient routing. Items that the place and route tool will look at include routing length (how far does a signal have to travel), track congestion (how many signals are coming into or out of an area), and path delays. While the process is usually performed automatically by the vendor-supplied tools, the designer can specify some parameters and constraints that the final layout has to meet, including:

- the initial placement of the cells
- a position for each physical connector
- a form factor

Programming the device

Once the design is successfully verified and found to meet timing and performance requirements, the final step is to actually program the device. At the completion of placement and routing, a binary programming file is created. It's used to configure the device. The process of programming is usually dependent on the type of memory used to store the device configuration and on the device type (e.g., FPGA or ASIC). ASICs are manufactured, rather than programmed by the end-user, and verification of the design is critically important. Re-generating an ASIC is costly, both in dollars and in schedule time. FPGAs and other programmable devices are programmed by the end-user, either in-circuit or in a special programming device. Usually, a software tool running on a PC will interface with the programmable device and download the program using the appropriate format.

Entrance Criteria

The following criteria should be met prior to beginning the implementation process.

- The design is reviewed and approved.
- Design synthesis was successful.
- Design verification and simulation was successfully performed.

Exit Criteria

At the end of the implementation phase, the following criteria should be met:

The device is programmed with the design.

2.8 Case Study on Digital System Design Hierarchy, Partitioning and Implementation

Stop Watch Design (See Chapter 3)

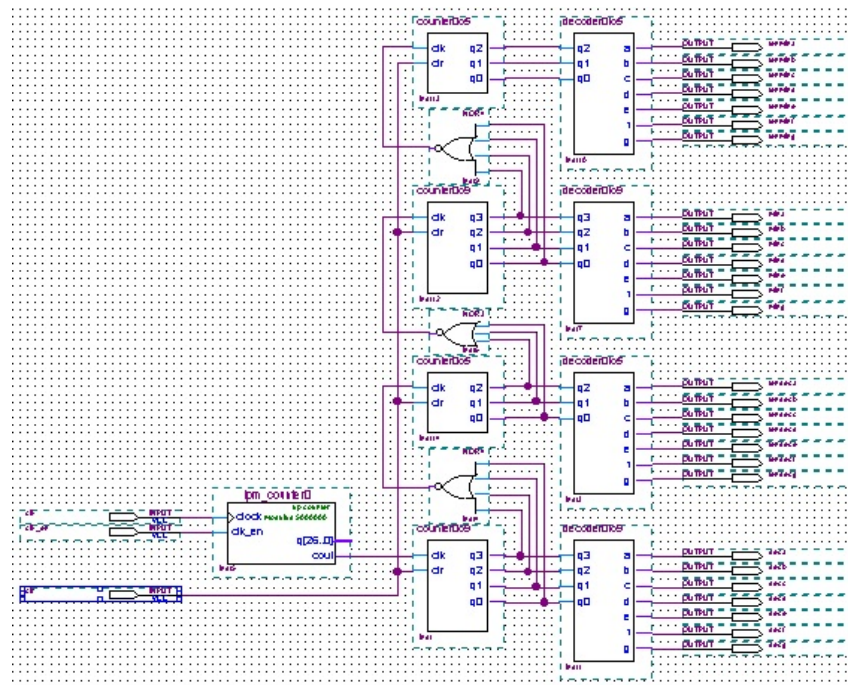


Figure 2.26 System Partitioning and Hierarchy Design of Stop Watch

Implementation of Stop Watch Design (See Lab 7)

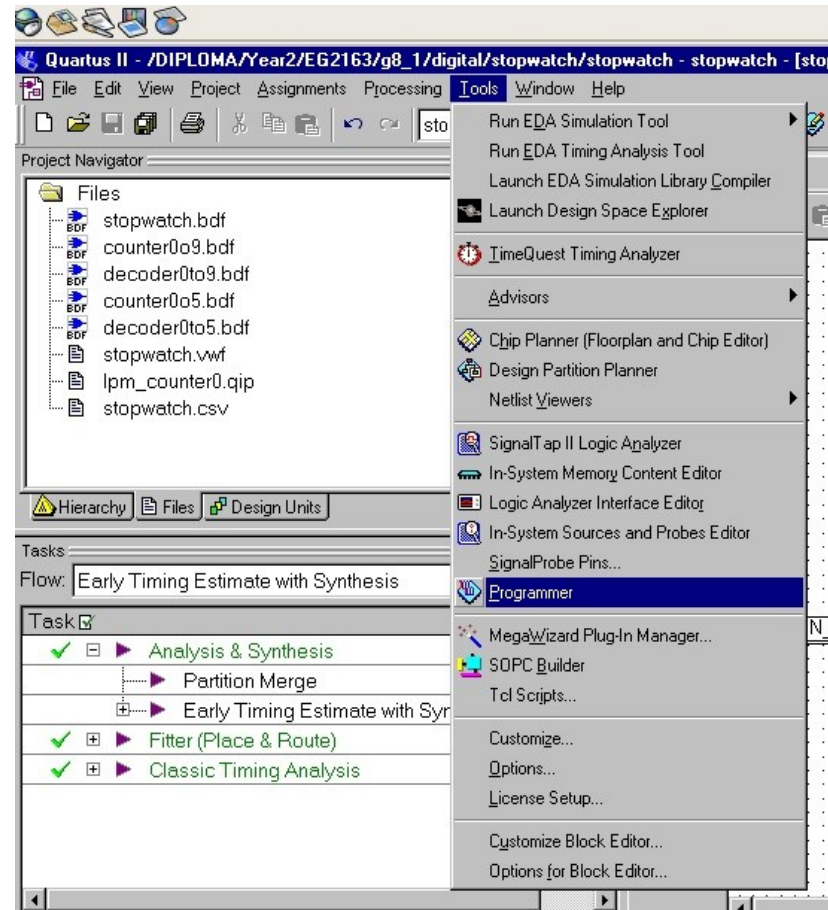


Figure 2.27 Altera's FPGA Programmer Tools