Introduction to Programmable Logic Devices(PLD's)
&
VHDL

UNIT-6

Courtesy: J. S. Katre & R.P. Jain

Introduction to PLD's & VHDL

- **Introduction to PLD's:** - PAL, PLA, Applications of PLAs to implement combinational and sequential logic circuits, FPGA and CPLD.
- **VHDL:** Introduction to HDL, Data Objects & Data Types, Attributes., VHDL- Library, Design Entity, Architecture, Modeling Styles, Concurrent and Sequential Statements.
- **Design Examples:** VHDL for Combinational Circuits-Adder, MUX, VHDL for Sequential Circuits, Synchronous and Asynchronous Counter.

Programmable Logic Device

- A Combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.
- We have studied different IC's such as multiplexers, demultiplexers, adders, code converters which are called as fixed function ICs.
- The advantages of designing logic circuits using the fixed function ICs are low development cost & easy testing, & disadvantages is requirement of large board space, large power requirements, no security, additional costs needed for the modification of existing circuit.
- The ICs that are used for designing a specific application is called the application specific integrated circuits (ASICs). They are too complex to design.
- The third approach to the problem of digital circuit design is to use the programmable logic devices (PLDs). It has the advantages of both the approaches discussed earlier.
- PLD are special type of ICs which can be programmed by the users as per their requirements.
- Thus it is possible to implement a combinational or sequential circuit using the PLD ICs.

Programmable Logic Device

Advantages of PLDs:

- Reduced space requirement.
- Reduction in power requirement.
- Increased speed of switching.
- Better security, Higher density, low production cost as compared to ASICS.
- Low development cost, short design cycle i.e. time required to implement the design is short.
- More flexibility to the designer.
- Reprogramming is possible to be done within few seconds.
- Modifications can be carried out within a short span of time.
- An important application of PLDs is that we can prepare prototype ASIC design, & reduces the time required to design ASIC

Programmable Logic Device

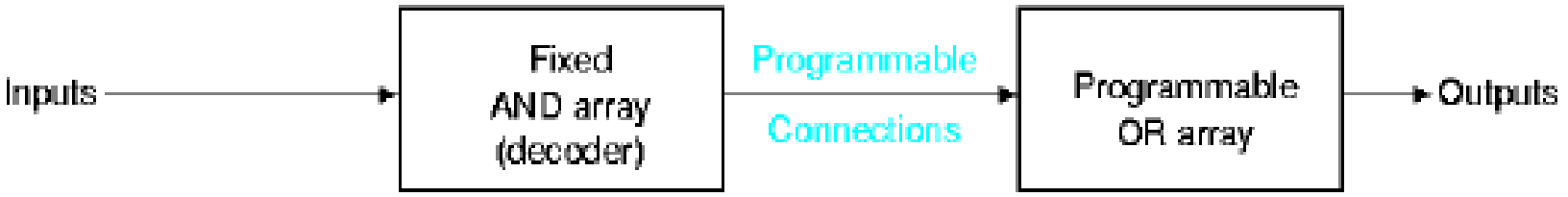
Types of PLDs:

- Programmable Read only memories (PROMs).
- Programmable Logic Array (PLAs).
- Programmable Array Logic (PALs).
- Simple Programmable Logic Devices (SPLDs).
- Complex Programmable Logic Devices (CPLDs).
- Field Programmable Gate Arrays (FPGAs).

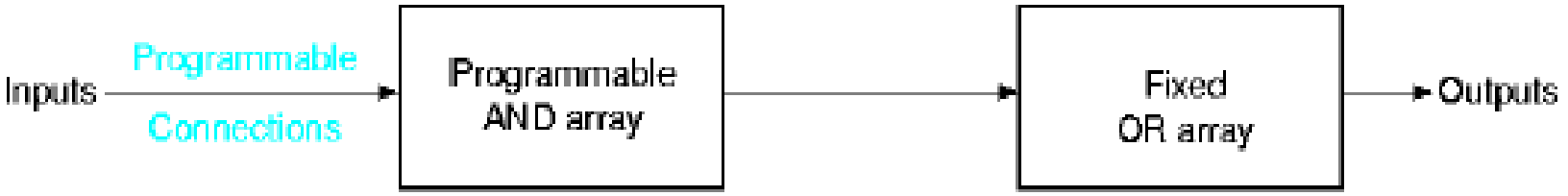
Note:

- Stores permanent binary information (nonvolatile). Can be read only (cannot be altered). Information is specified by designer and physically inserted (embedded) into the PLD.
- Programmable connections are formed by fuses, masks, or antifuses depending on the technology.

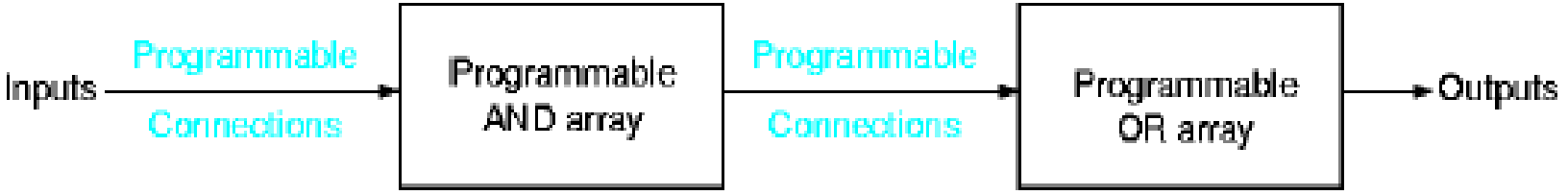
PROGRAMMABLE LOGIC DEVICES



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL) device

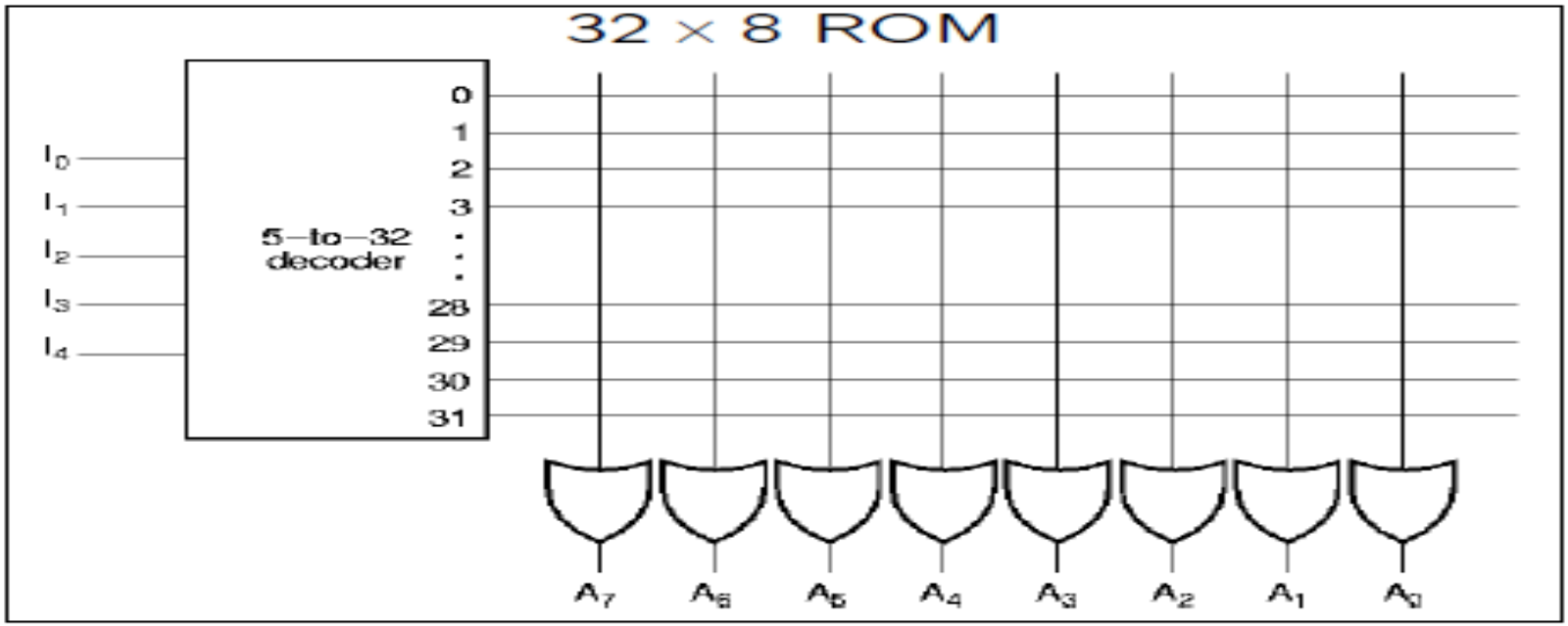


(c) Programmable logic array (PLA) device

ROM as PLD

Read-Only Memory

k inputs (address) $\Rightarrow$ $2^k \times n$ ROM $\Rightarrow$ n outputs (data)



ROM as PLD

- k *x* 2^k decoder to decode input address.
- n OR gates with 2^k input each.
- Decoder output is connected to all n OR gates through fuses.
- ROM -> 2^k *x* n programmable connections.
- EX: 4 x 2 ROM.
- Truth table -> address and content of ROM.
- Programming -> stores truth table in ROM.
- 0 = Open connection = Fuse blown.
- 1 = Closed connection = Fuse intact.

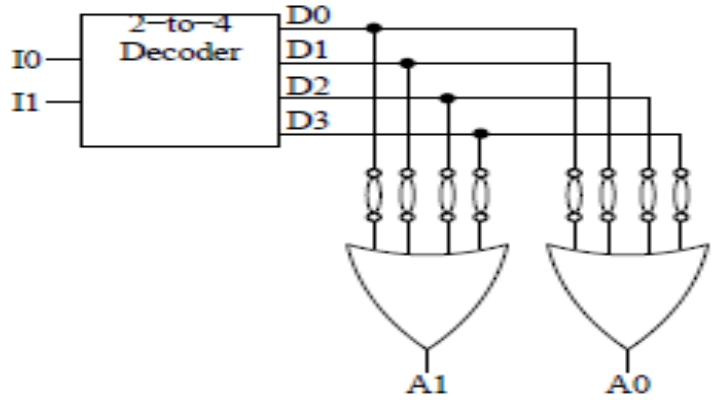
ROM as PLD

Example of 4 × 2 ROM

Truth table

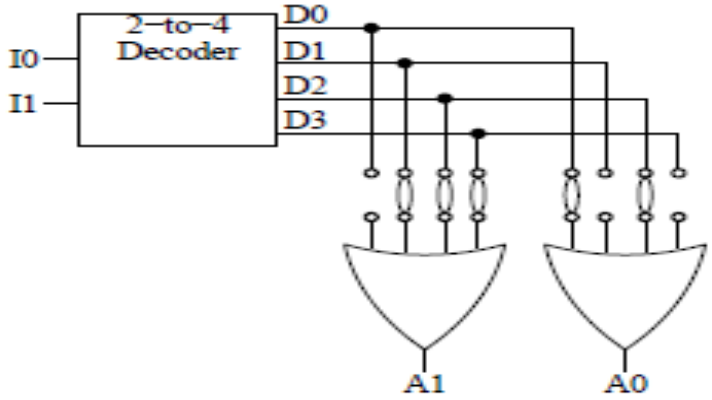
Address		Content	
I_1	I_0	A_1	A_0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

NonProgrammed ROM

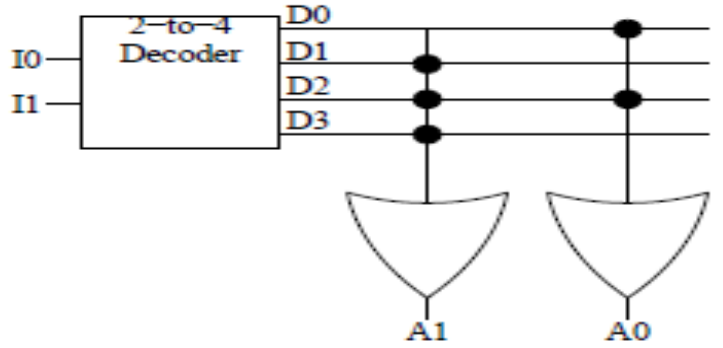


⇓

Programmed ROM



Compact ROM



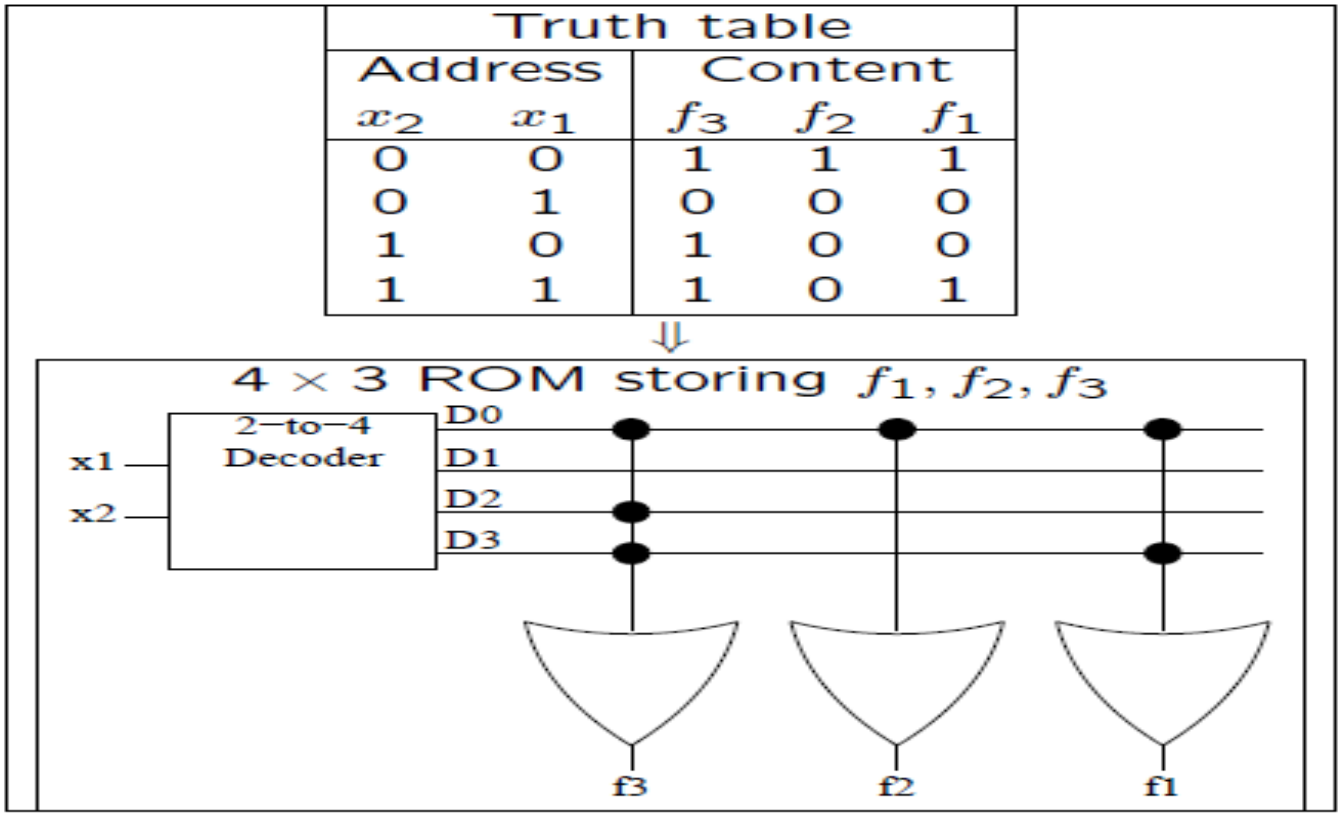
Function Synthesis with ROM

Any set of functions $f_1(x_k, \ldots, x_1)$, ..., $f_n(x_k, \ldots, x_1)$ can be realized with a $2^k \times n$ ROM

Example: Implement $f_1(x_2, x_1) = \sum m(0, 3)$, $f_2(x_2, x_1) = \overline{x_2 + x_1}$, and $f_3(x_2, x_1) = \prod M(1)$ with a 4×3 ROM

Truth table

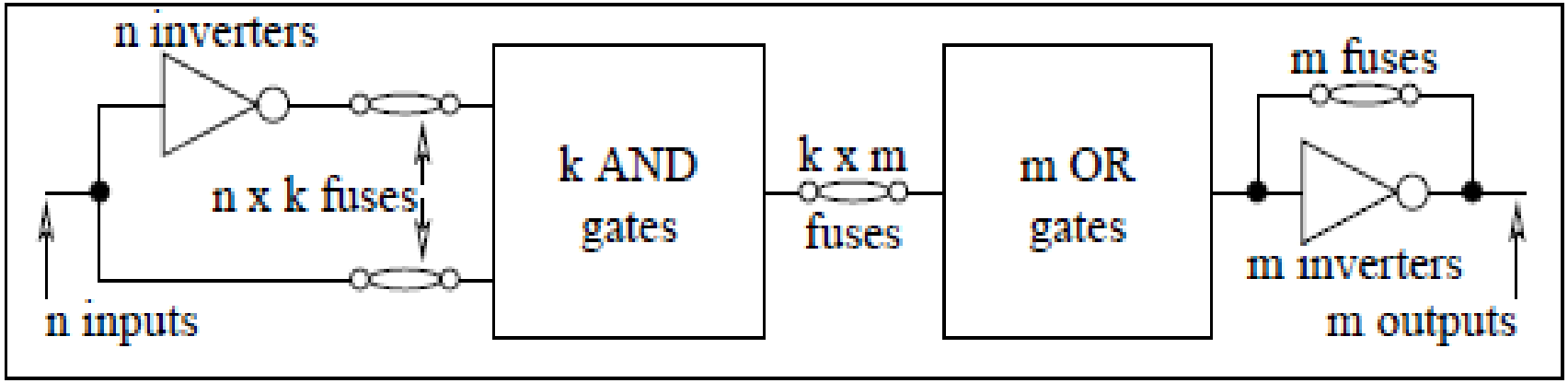
Address		Content		
x_2	x_1	f_3	f_2	f_1
0	0	1	1	1
0	1	0	0	0
1	0	1	0	0
1	1	1	0	1



Programmable Logic Array

- A PLD generally consists of programmable array of logic gates. Interconnections are made with the array inputs.
- The outputs are connected to the device pins through inverting or noninverting buffers and flip flops.
- The logic gates used can be two level AND-OR, NAND-NAND or NOR-NOR configuration. Sometimes AND-OR-EXOR configuration is also used.
- There are two types of PLDs namely the PLA i.e. programmable logic arrays and PAL i.e. the programmable array logic. Due to the used of AND matrix followed by the OR matrix, we can use them for the implementation of logic functions in the SOP form.
- A PLA consists of two level of AND-OR circuit on the same chip.
- The AND matrix can be used to implement the product terms in the SOP form and the OR matrix is used for implementing the sum of product term.

Programmable Logic Array

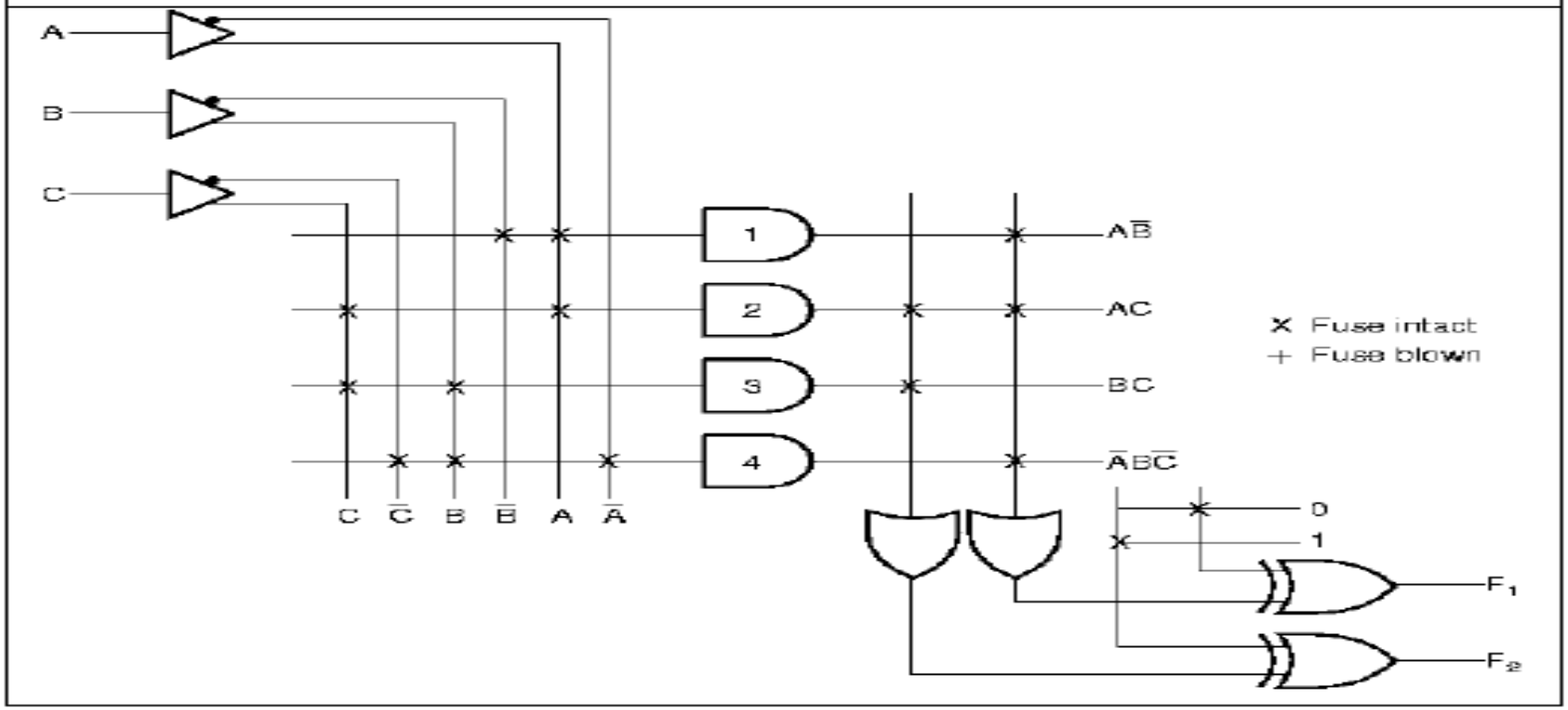


- Behave like a ROM but has different structure:
- Uses ANDs array instead of decoder to produce product terms of inputs
- Has programmable connections before ANDs, between ANDs and ORs, after ORs. That is $2nk + km + m$ fuses
- More flexible than ROM but more difficult to program.
- Logic expressions for content information to be stored in PLA must be obtained first, then minimized, and finally programmed into the PLA using a PLA program table.
- PLA program table specifies product terms and sum terms of information that will be stored in PLA.

Programming a PLA

PLA Program Table						
		Inputs			Outputs	
Term	Term #	A	B	C	F_1	F_2
$A\bar{B}$	1	1	0	_	1	_
AC	2	1	_	1	1	1
BC	3	_	1	1	_	1
$\bar{A}B\bar{C}$	4	0	1	0	1	_
					T	C

Corresponding PLA Implementation



Function Synthesis with PLA

Any set of functions $f_1(x_1, \ldots, x_n)$, ..., $f_m(x_1, \ldots, x_n)$ can be realized with a PLA

Example Implement $f_1(a, b, c) = \sum m(3, 5, 6, 7)$ and $f_2(a, b, c) = \sum m(0, 2, 4)$ with a PLA

First Simplify $f_1, \bar{f}_1, f_2, \bar{f}_2$, that is

$$f_1(a, b, c) = ab + ac + bc \qquad f_1, f_2 \rightarrow 5 \text{ terms}$$
$$\bar{f}_1(a, b, c) = \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{c} \qquad f_1, \bar{f}_2 \rightarrow 4 \text{ terms}$$
$$f_2(a, b, c) = \bar{a}\bar{c} + \bar{b}\bar{c} \qquad \bar{f}_1, f_2 \rightarrow 3 \text{ terms}$$
$$\bar{f}_2(a, b, c) = ab + c \qquad \bar{f}_1, \bar{f}_2 \rightarrow 5 \text{ terms}$$

Second Select combination of functions that has less terms, that is

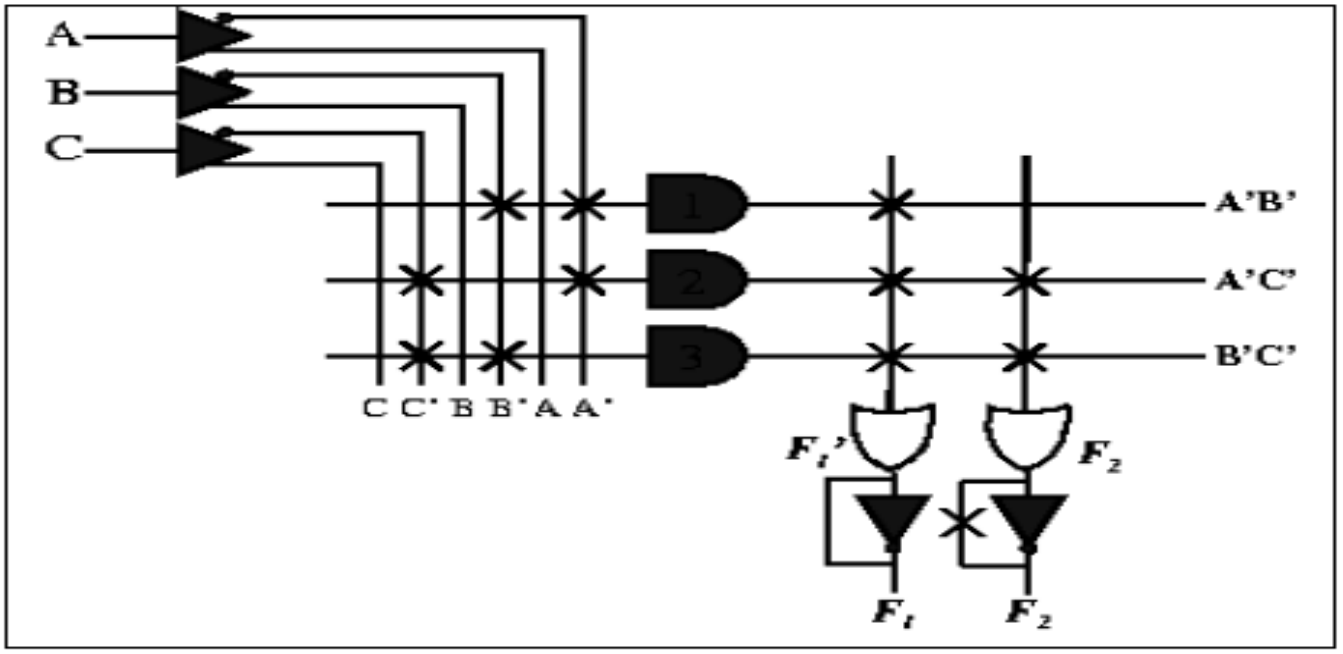
$$f_1 = \overline{\bar{f}_1} = \overline{\bar{a}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{c}}$$
$$f_2(a, b, c) = \bar{a}\bar{c} + \bar{b}\bar{c}$$

Function Synthesis with PLA

Third Construct a PLA program table from selected functions

Term	Term#	Inputs a	b	c	Outputs f_1	f_2
$\bar{a}\bar{b}$	1	0	0	_	1	_
$\bar{a}\bar{c}$	2	0	_	0	1	1
$\bar{b}\bar{c}$	3	_	0	0	1	1
					C	$\overline{\text{T}}$

Fourth Construct PLA circuit from PLA program table



Programmable Logic Array

Input Buffers: (See Pg. No. 14-12 J.S. Katre)

- It is used for avoiding the loading of sources connected at the inputs.
- Buffers are of two types namely, inverted buffers and non-inverted buffers.
- One such buffer is used for each of the M input lines.

AND Matrix:

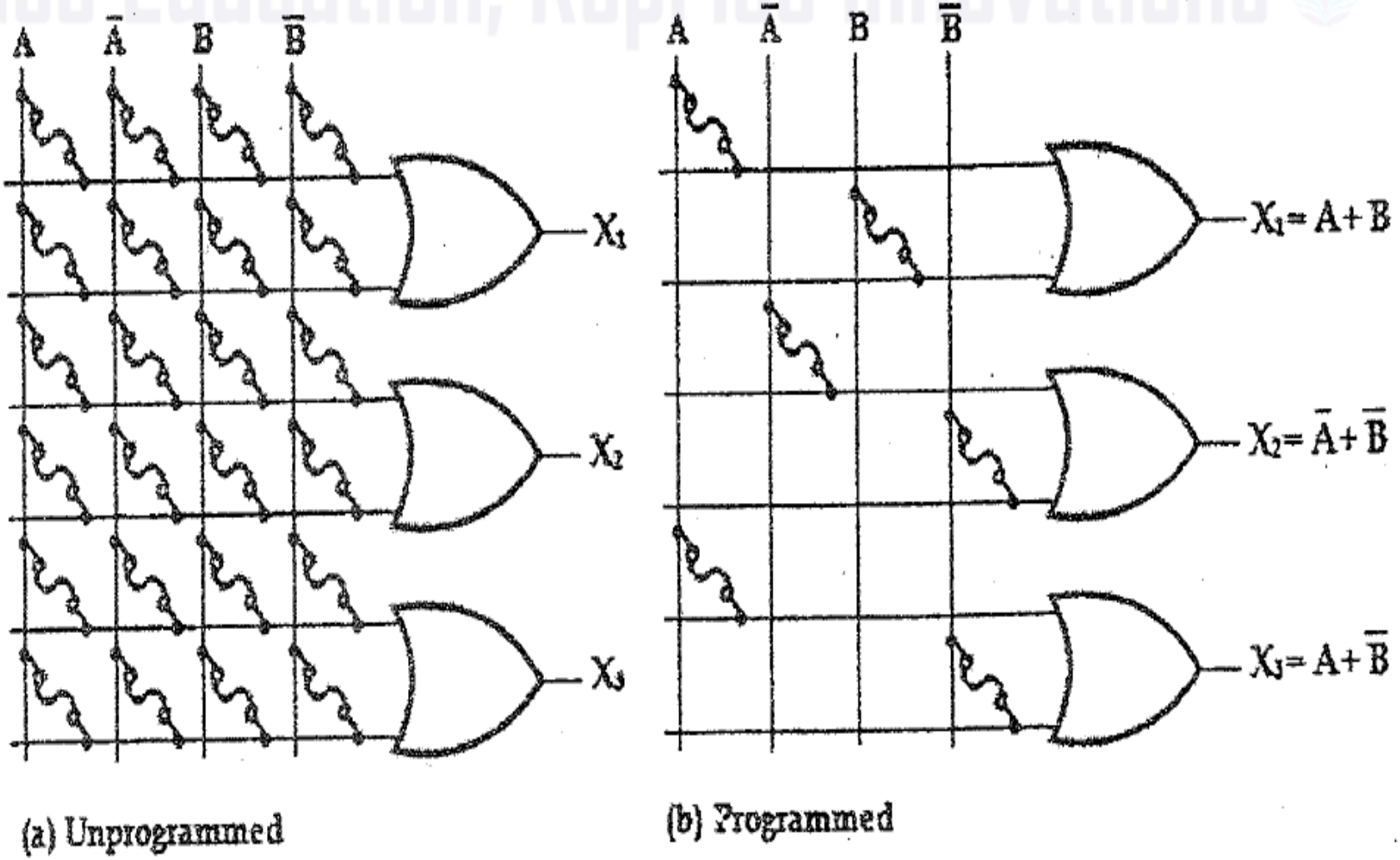
- There are 2M inputs for each AND gate, Po is the output of one of such an AND gate.
- There is a nichrome fuse link connected in series with each input. Thus Po=0 if all the fuse links are intact.
- In an unprogrammed PLA device all the fuse links are intact.
- Thus it consists of ‘n’ such AND gates formed with the help of diodes. Each AND gate has 2M inputs. The output is thus a product term. So the required product term can be generated by opening the unwanted nichrome fuse links.
- The (x) marks indicate that a connection is present. Each AND gate has 2M inputs which are shown only by a single line.
- When a logic function is to be implemented, we have to program the array. In programming the desired connections are left with the (x) marks and such mark is not used when connection is not required.

Programmable Logic Array(PLA)

OR Matrix:

- The output of AND matrix is used as the inputs to the OR matrix. These are applied to the bases of transistors.
- An OR gate is formed by parallel connected transistors and the load is common and connected in the emitters.
- The required sum terms are obtained by opening the unwanted fuse links.
- It is possible to programme the OR matrix, by open circuiting the unwanted fuse links. The open fuse links are equivalent to a “0” at the input of corresponding OR gate.

The OR Array

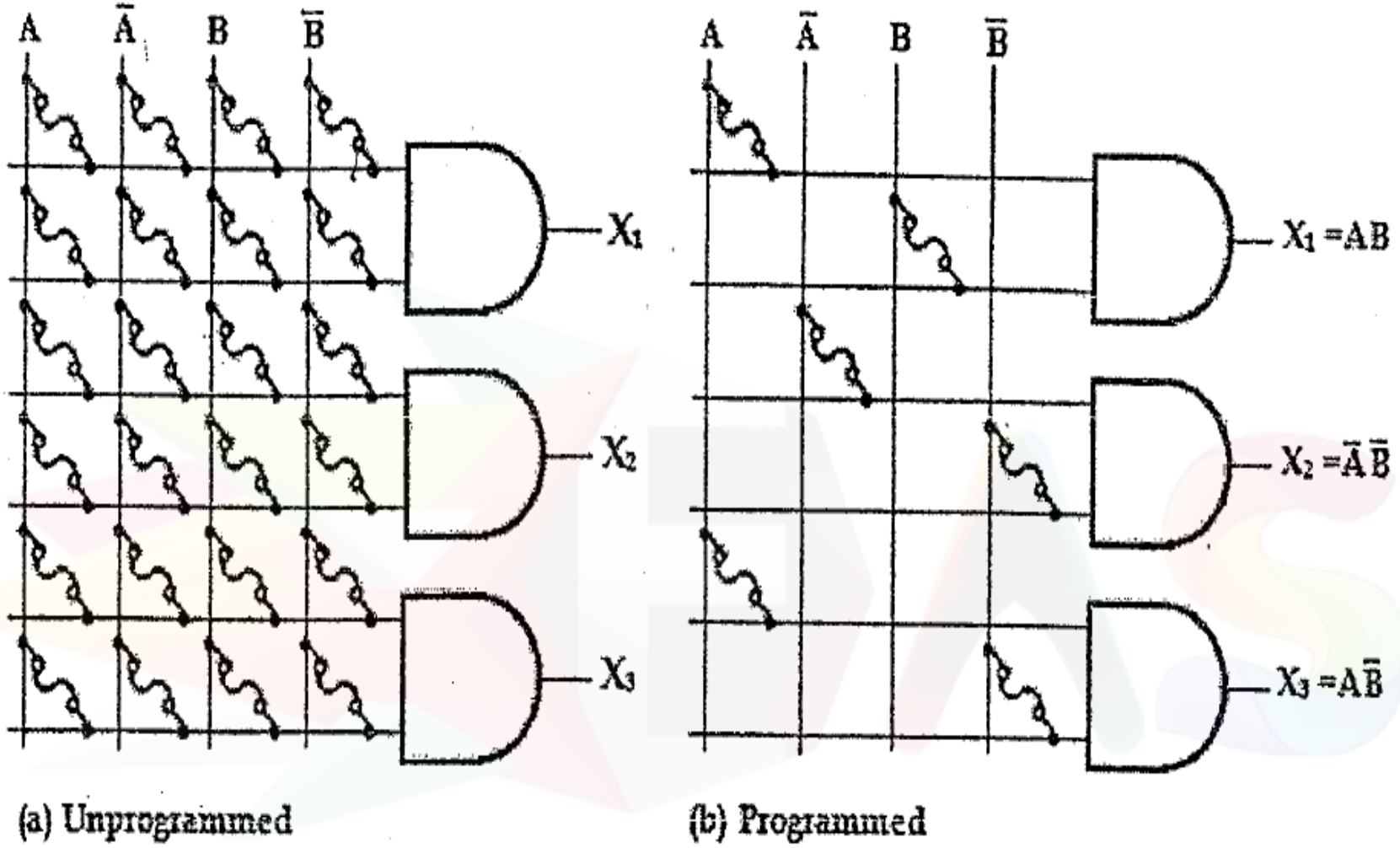


(a) Unprogrammed

(b) Programmed

An example of a basic programmable OR array

The AND Array



An example of a basic programmable AND array

Programmable Logic Array

Invert/Non Invert Matrix:

- The block following the OR matrix is the inverting/non inverting matrix which is basically a programmable buffer.
- It can invert its input if active low output is required. Its input is passed without any inversion, if active high output is required.
- Thus the sum of product terms obtained at the outputs of the OR matrix can be either inverted or passed through as it is.
- For the EX-OR gate, if the fuse link is closed, then one of inputs will be 0. Hence $0 \oplus S = S$. Whereas if the fuse link is opened then that input will be treated as 1. So $1 \oplus S = \overline{S}$. i.e. inversion will take place.

Output Buffers:

- The current sourcing capability of the PLA can be increased by using the output buffer. The PLA outputs are generally TTL compatible.
- In tristate output buffer, the chip enable $\overline{CE}$ input is applied to the output buffers. When $\overline{CE}$ =0, the outputs will be made available.

Programmable Logic Array

Output through Buffers and Flip Flops:

- If the PLA devices are to be used for state machine applications, then the output circuit consists of buffers and flip flops.
- The OR gate outputs are connected to the inputs of the positive edge triggered SR flip flops. The flip flop outputs are then applied to the tristate buffers and the device outputs are obtained as the buffer outputs.

Programming Procedure for PLA:

- PLA programming is similar to the ROM programming.
- Like ROM, it is possible to mask programme the PLA devices.
- According to the requirement specified by the user, the manufacturer prepares a mask and the data patterns are stored on the PLA.
- In field PLA (FPLA), all the nichrome fuse elements are intact at the time of manufacturing. At the time of programming, all the unwanted links are open circuited electrically.
- It is not possible to reprogramme an FPLA.

Programmable Logic Array

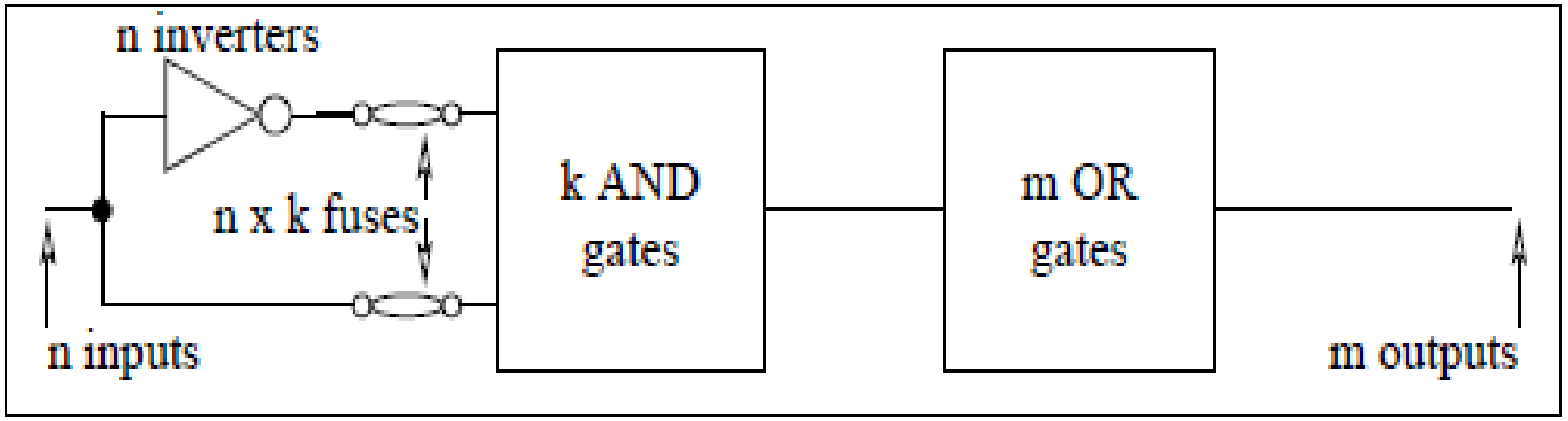
Applications Areas of PLA:

- We can implement the combinational as well as the sequential circuits using PLA.
- For implementing the combinational circuits, the PLA devices with only output buffers are used.
- For implementing the sequential circuits, the PLA devices with flip flop and buffers are included in the output stage.

Programmable Array Logic(PAL)

- Similar to PLA
- Only the connection inputs to ANDs are programmable
- Easier to program than but not as flexible as PLA
- There are feedback connections
- Logic expressions for content information to be stored in PAL must be obtained first, then minimized, and finally programmed into the PAL using a PAL program table

Programmable Array Logic



Field Programmable Logic Array

- FPLA can be programmed by the user.
- Commercial hardware programmer units are available for use in conjunction with certain FPLAs.
- Here we have fused programmable AND array and fused programmable OR array.

Field Programmable Gate Array (FPGA)

- It is an integrated circuit designed to be configured by the customer or designer after manufacturing “field-programmable”.
- It consists of an array of logic blocks with programmable row and column and interconnecting channels surrounded by programmable input output blocks.
- It is generally specified using a hardware description language(HDL).

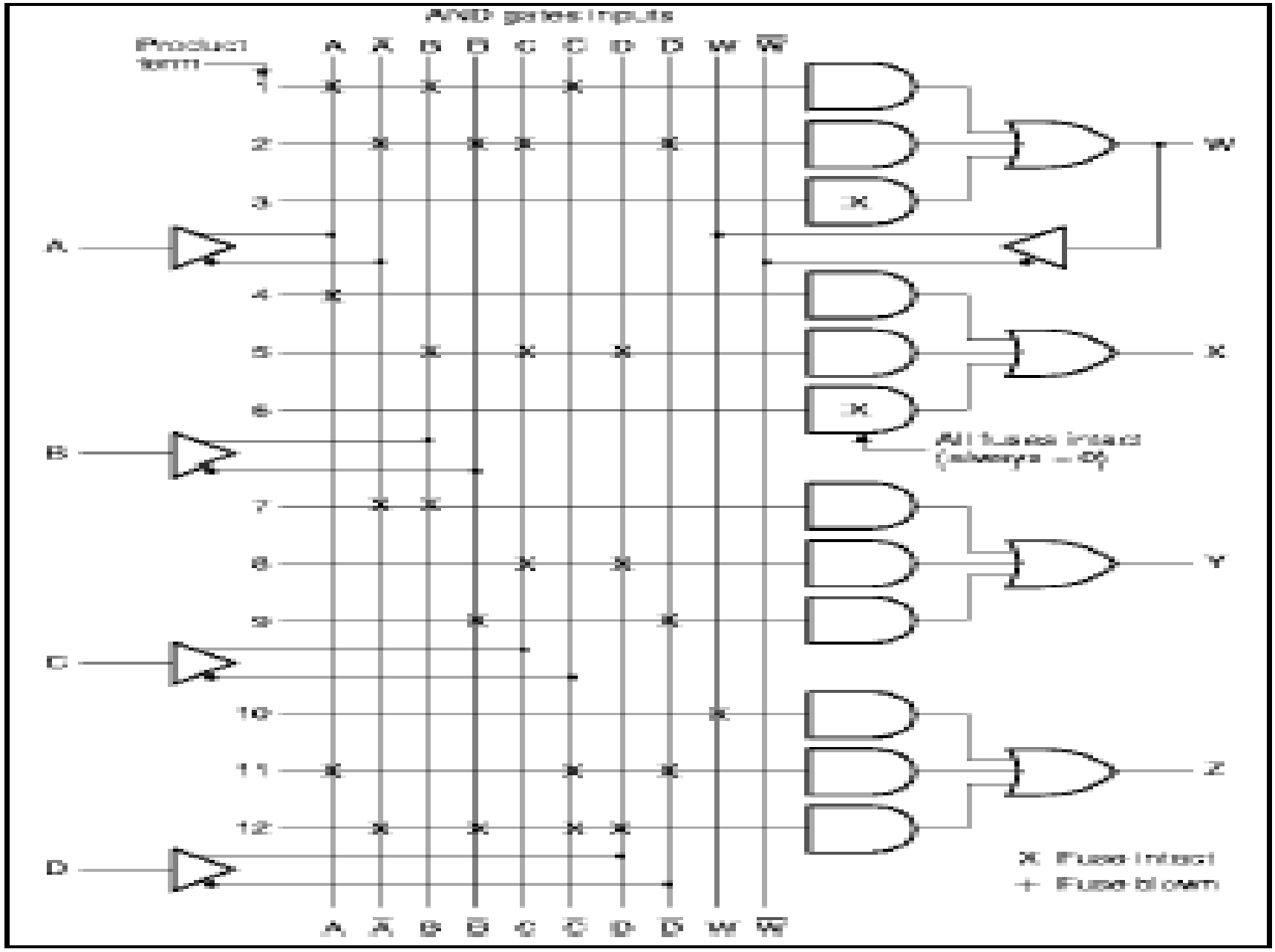
Complex Programmable Logic Devices(CPLD):

- It is a collection of individual PLDs on a single chip, with a programmable interconnection structure that allows the PLDs to get connected as the user wants.

Comparison between PROM, PAL, PLA.

S.No	PROM	PLA	PAL
1	AND array is fixed and OR array is programmable.	Both AND & OR arrays are programmable.	OR array is fixed and AND array is programmable.
2.	Cheaper and simple to use.	Costliest & complex than PAL & PROMs.	Cheaper and simpler.
3.	All minterms are decoded.	AND array can be programmed to get desired minterms.	AND array can be programmed to get desired minterms.
4.	Only Boolean functions in standard SOP form can be implemented using PROM.	Any Boolean functions in standard SOP form can be implemented using PROM.	Any Boolean functions in standard SOP form can be implemented using PROM.

Programming a PAL



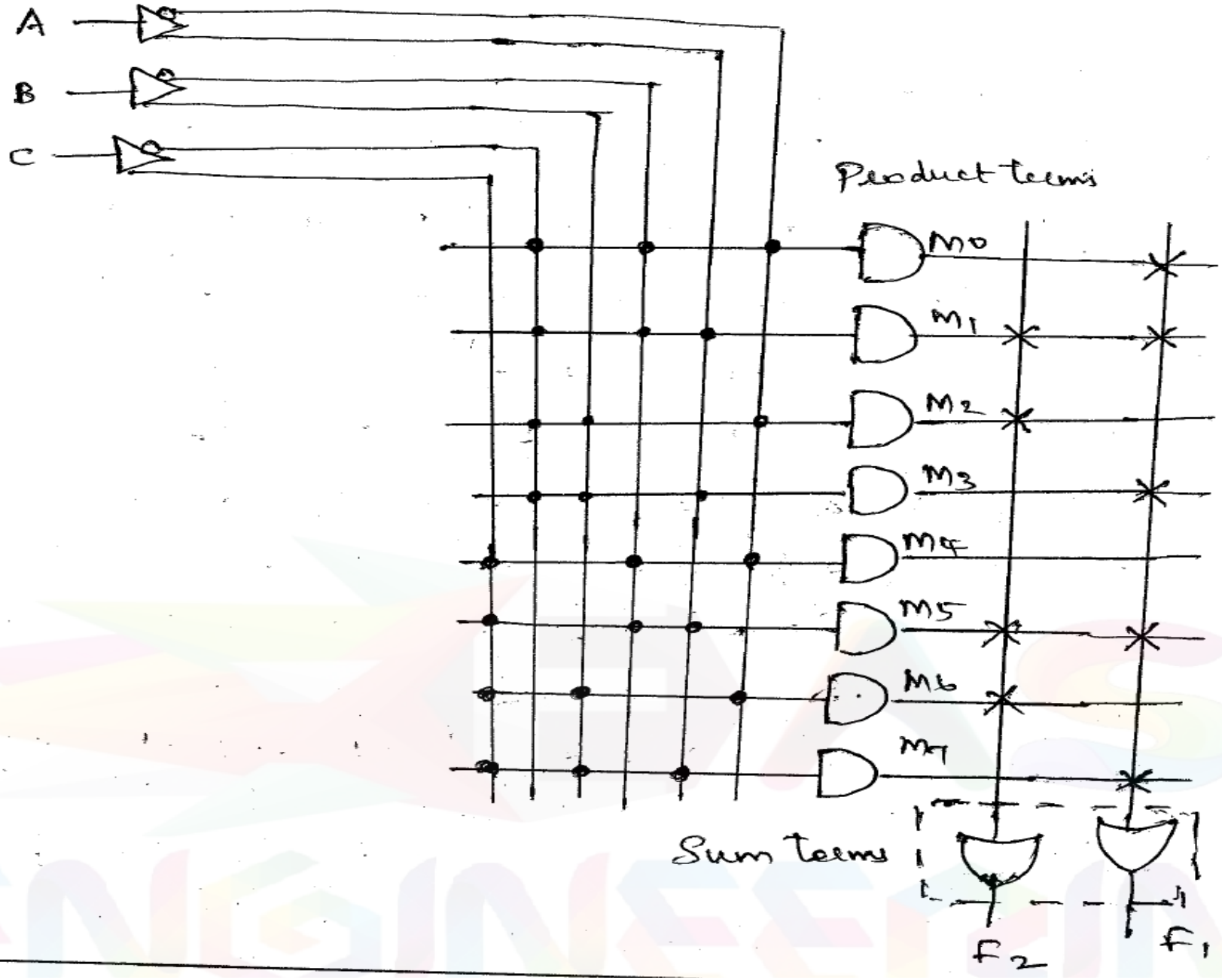
Using PROM realize the following expression.

$f_1(A,B,C) = \Sigma_m(0,1,3,5,7)$

$f_2(A,B,C) = \Sigma_m(1,2,5,6)$

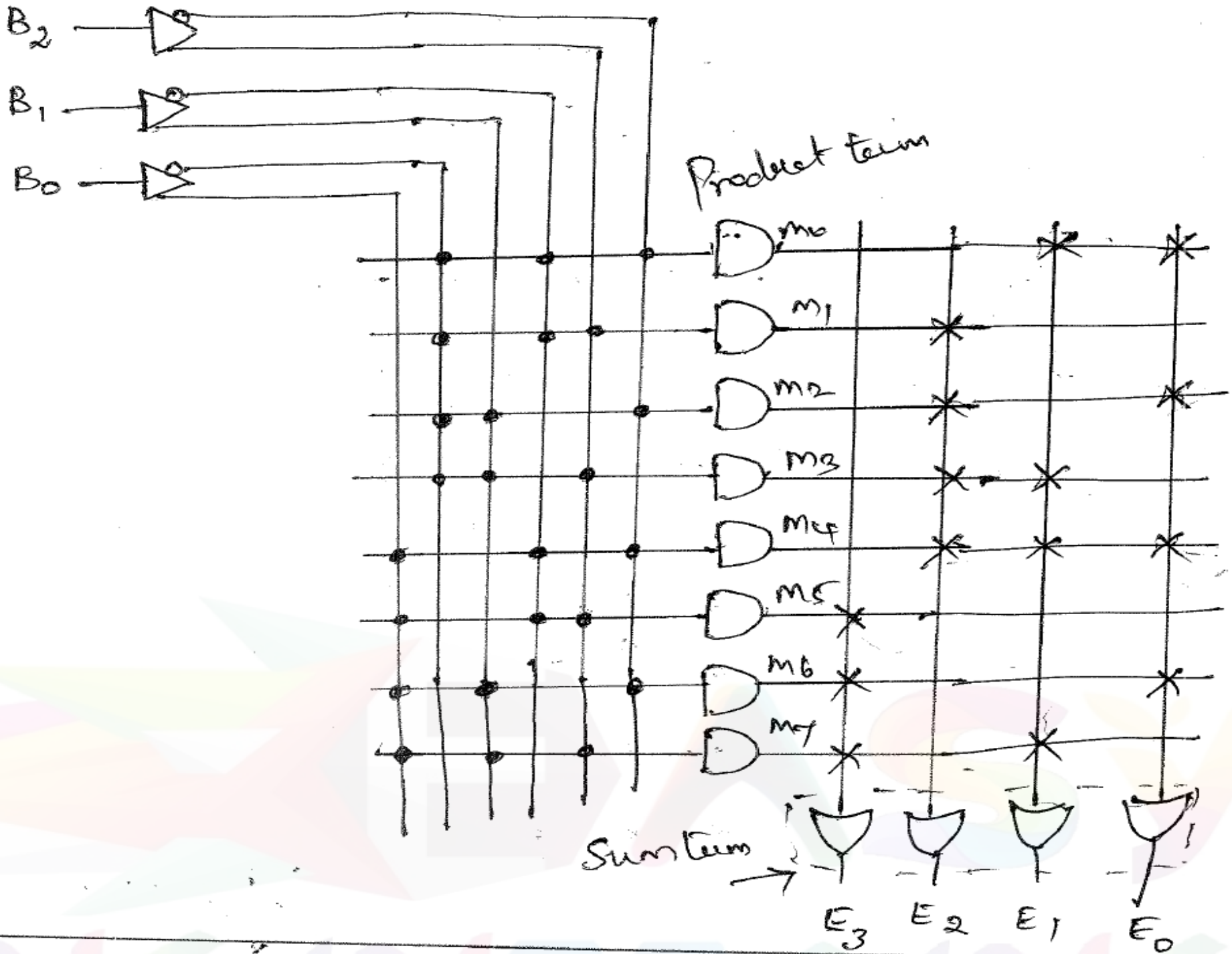
Truth table.

A	B	C	f_1	F_2
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0


A
B
C
Product terms
M0
M1
M2
M3
M4
M5
M6
M7
Sum terms
F2
F1

Design a combinational circuit using PROM. The circuit accepts 3-bit binary and generates its equivalent excess-3 code.

B_2 B_1 B_0	E_3 E_2 E_1 E_0
0 0 0	0 0 1 1
0 0 1	0 1 0 0
0 1 0	0 1 0 1
0 1 1	0 1 1 0
1 0 0	0 1 1 1
1 0 1	1 0 0 0
1 1 0	1 0 0 1
1 1 1	1 0 1 0


B_2
B_1
B_0
Product term
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
Sum term
E_3
E_2
E_1
E_0

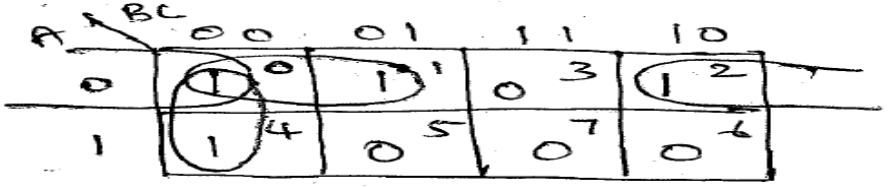
1. Implement the Combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions. $f_1(A,B,C) = \sum m(0,1,2,4)$

$f_2(A,B,C) = \sum m(0,5,6,7)$.

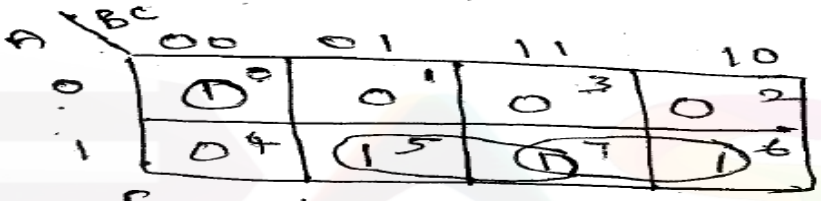
Truth table

A	B	C	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

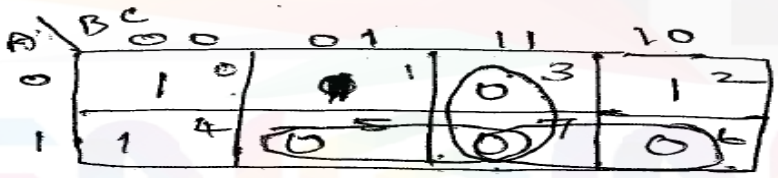
K-map simplification



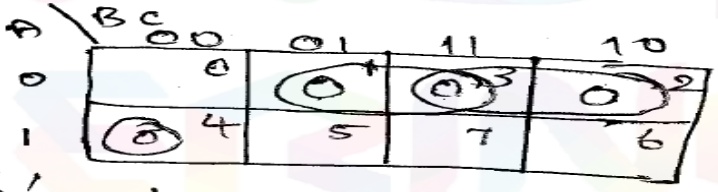
$F_1 = A'B' + A'C' + B'C'$.



$f_2 = AC + AB + A'B'C'$.



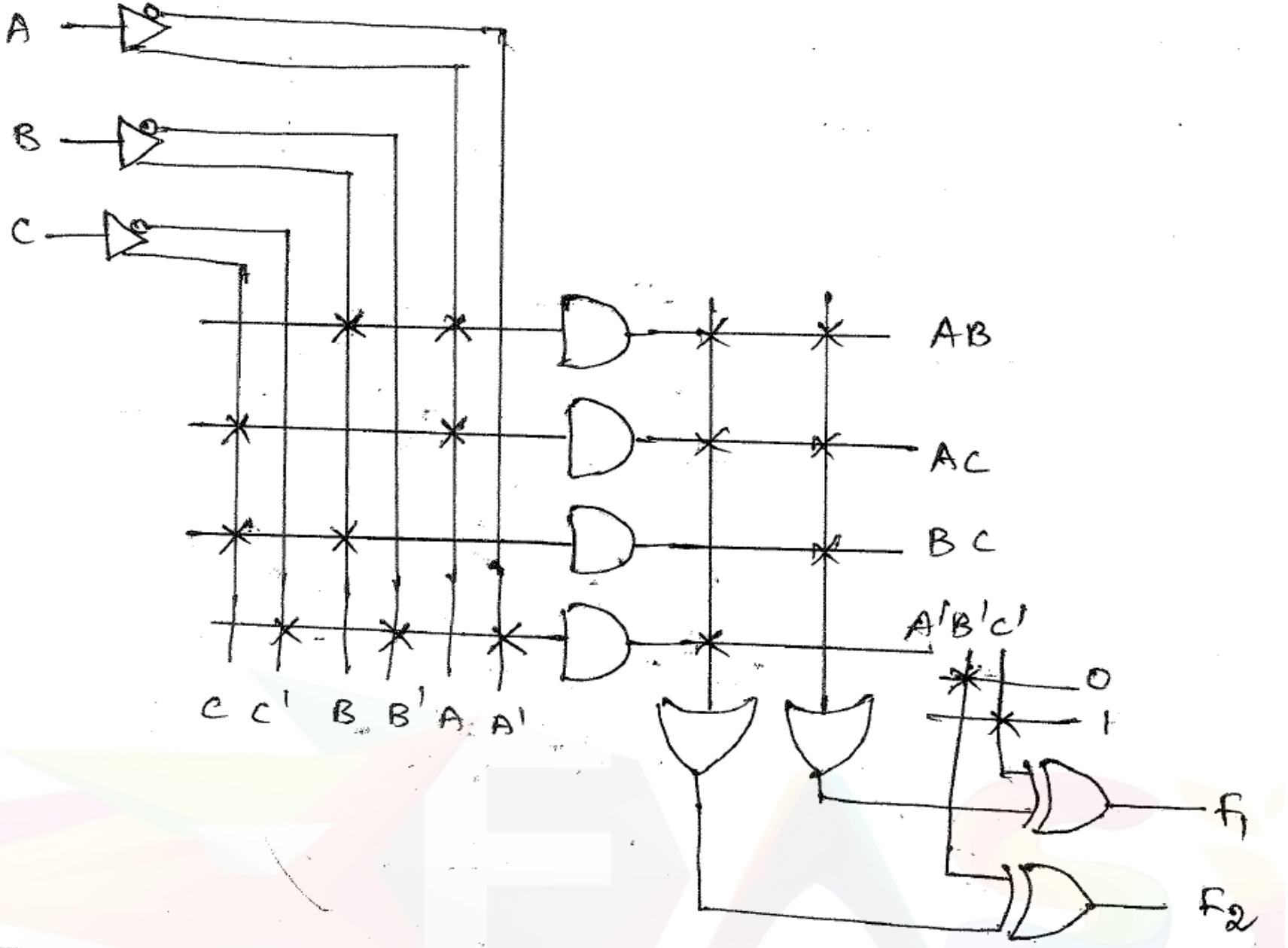
$F_1' = AC + BC + AB$.



$f_2' = A'C + A'B + A'B'C'$.

Program table.

	Product term	Inputs A	Inputs B	Inputs C	Output $F_1(C)$	Output $F_2(T)$
AB	1	1	1	—	1	1
AC	2	1	—	1	1	1
BC	3	—	1	1	1	—
$A'B'C'$	4	0	0	0	—	1


A
B
C
AB
AC
BC
A'B'C'
0
1
C C' B B' A A'
f1
F2

Implement the following function using PLA

$f_1(A,B,C) = \sum m(1,2,4,6)$

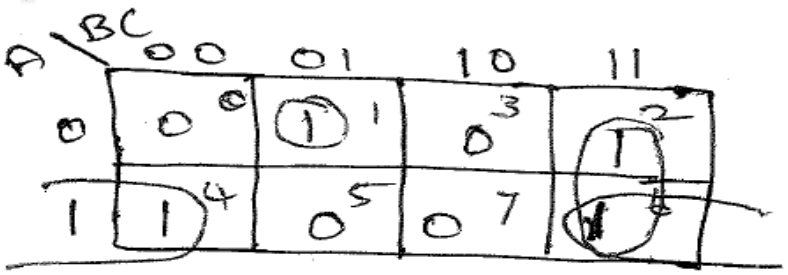
$F_2(A,B,C) = \sum m(0,1,6,7)$

$f_3(A,B,C) = \sum m(2,6)$

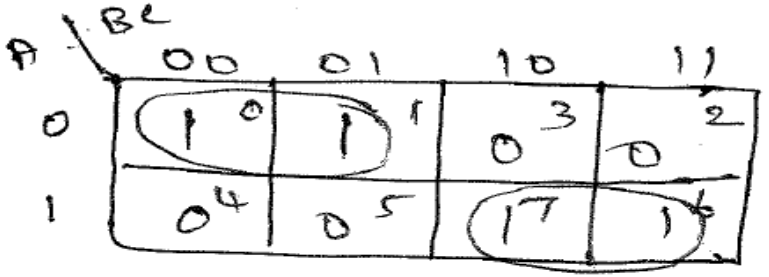
Truth table

A	B	C	f_1	f_2	f_3
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0

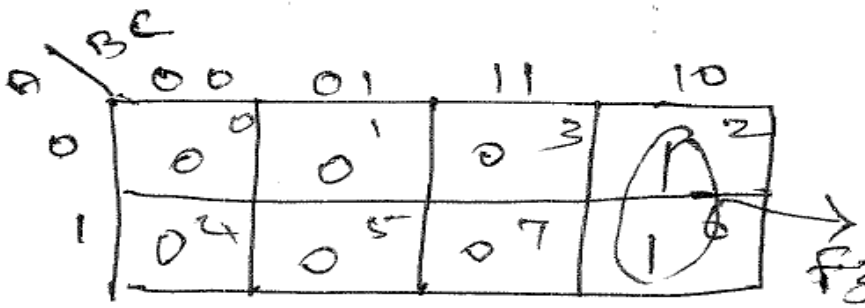
K map simplification



$f_1 = A'B'C + AC' + BC'$



$f_2 = A'B' + AB.$



$f_3 = BC'$

Implement the following function using PLA

$f_1(A,B,C) = \sum m(1,2,4,6)$

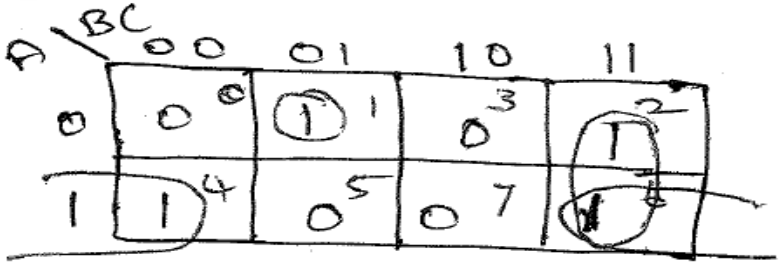
$F_2(A,B,C) = \sum m(0,1,6,7)$

$F_3(A,B,C) = \sum m(2,6)$

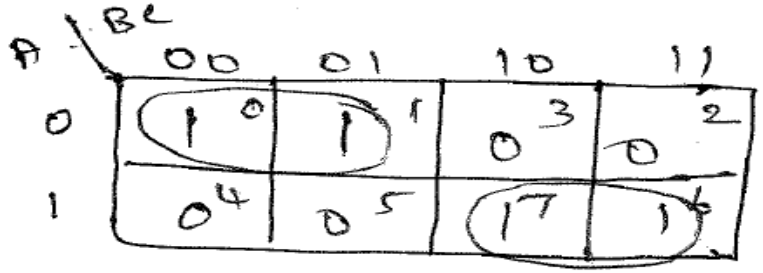
Truth table

A	B	C	f_1	f_2	f_3
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0

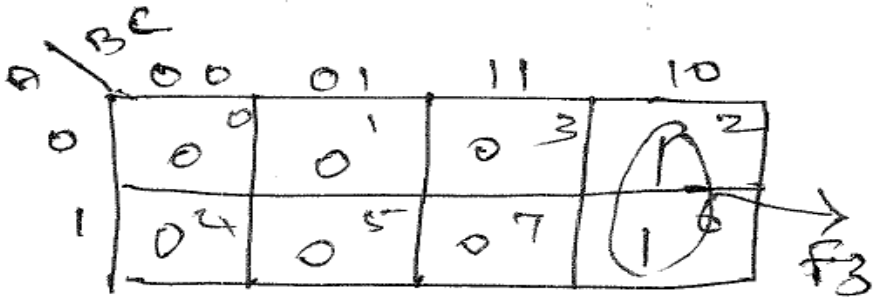
K map simplification



$f_1 = A'B'C + AC' + BC'$



$F_2 = A'B' + AB$

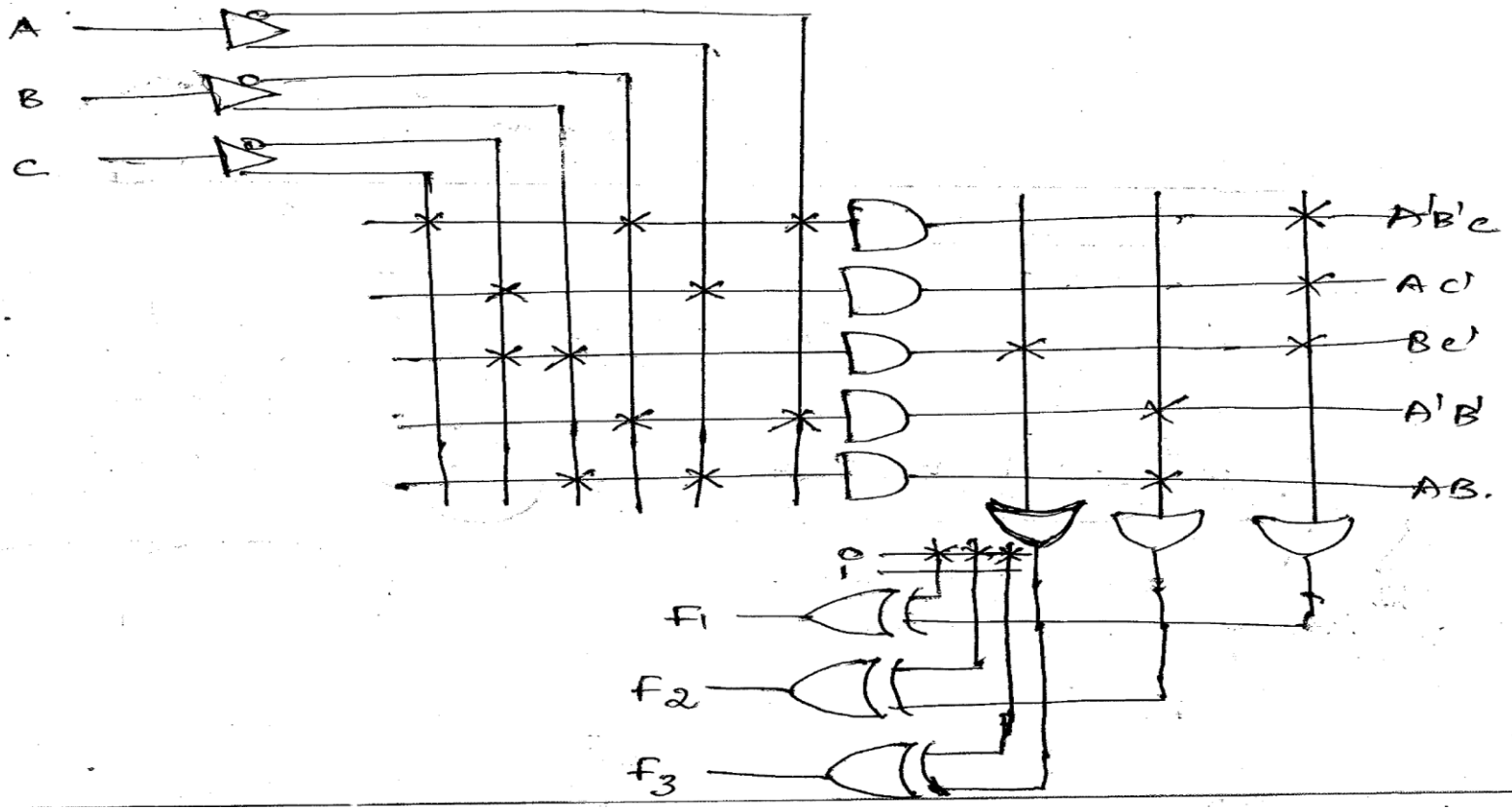


$f_3 = BC'$

Program table

	Product term	Inputs			outputs		
		A	B	C	$F_1(T)$	$F_2(T)$	$F_3(T)$
$A'B'C$	1	0	0	1	1	—	—
AC'	2	1	—	0	1	—	—
BC'	3	—	1	0	1	—	1
$A'B'$	4	0	0	—	—	1	—
AB	5	1	1	—	—	1	—

PLA diagram



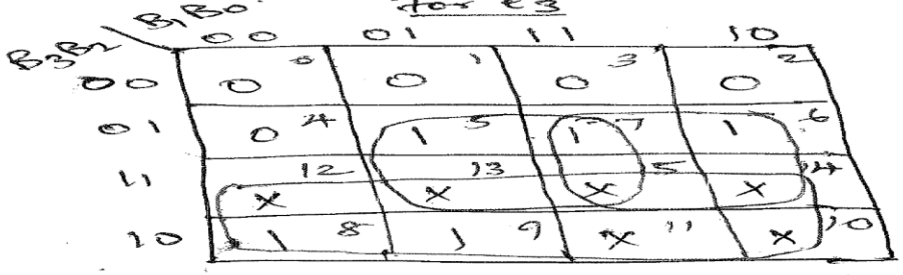
Design a BCD to Excess-3 code convertor and implement using suitable PLA.

Truth table.

BCD code.				Excess-3 code.			
B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

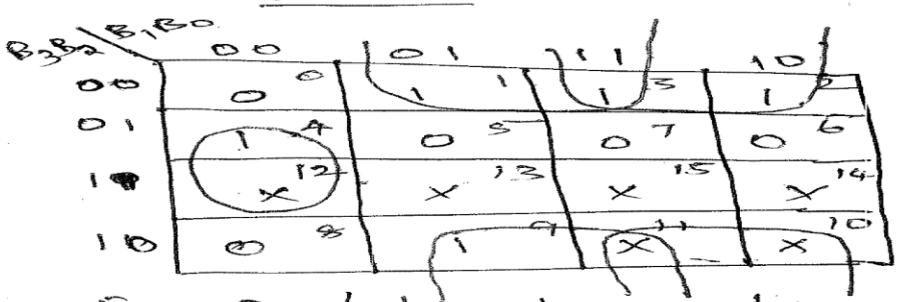
K-map simplification

For E_3



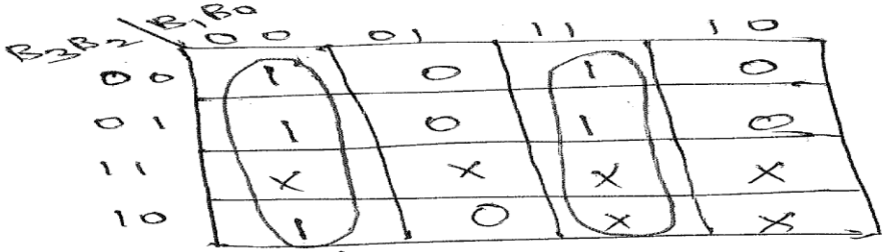
$E_3 = B_3 + B_2B_0 + B_2B_1$

For E_2



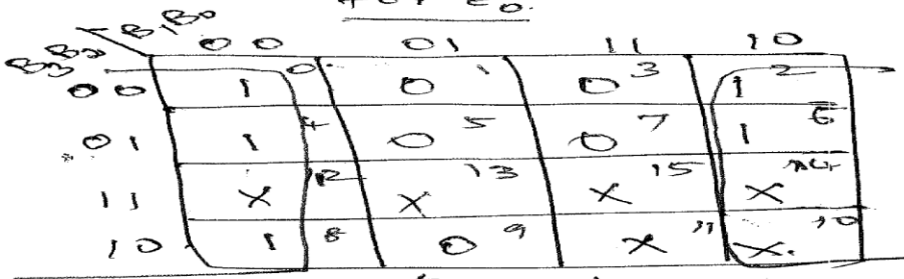
$E_2 = B_2B_1'B_0' + B_2'B_0 + B_2'B_1$

For E_1



$E_1 = B_1'B_0' + B_1B_0$

For E_0

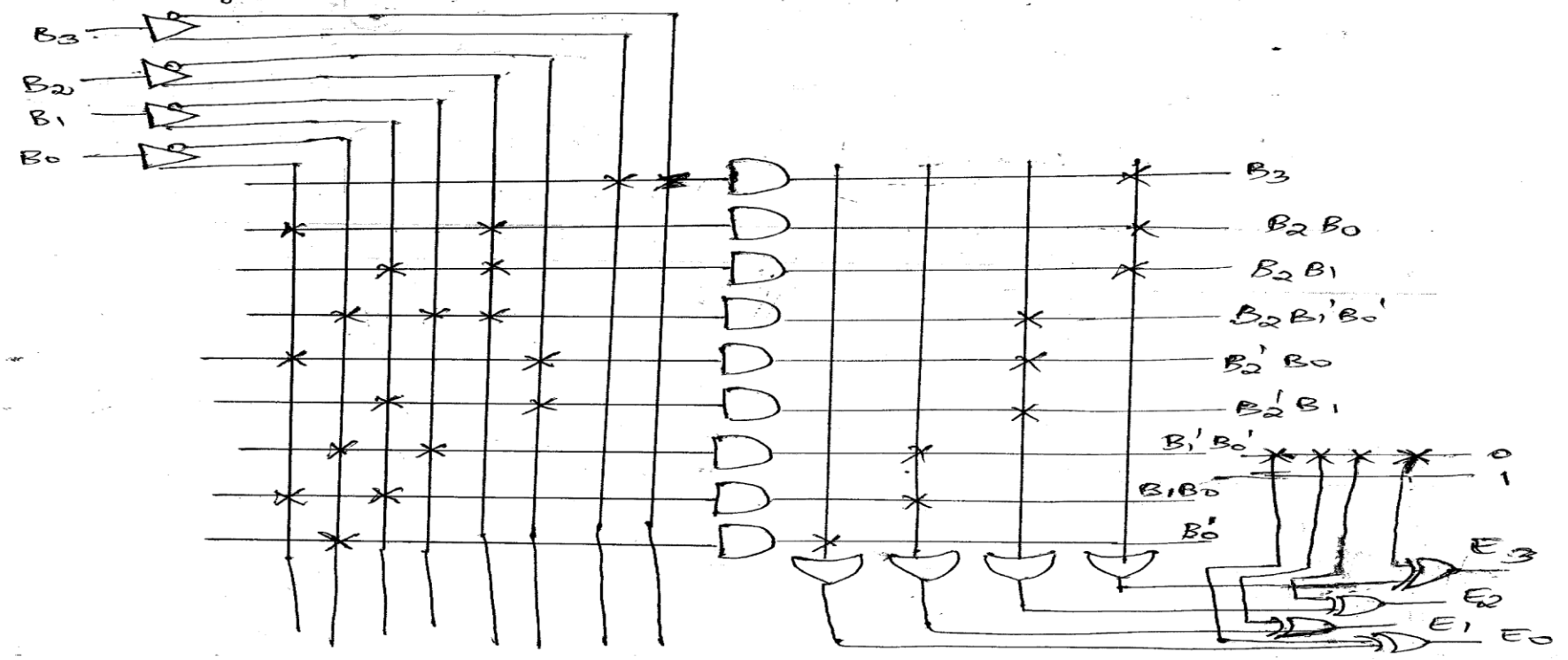


$E_0 = B_0'$

Program table.

	Product term	Inputs				Outputs			
		B_3	B_2	B_1	B_0	$E_3(T)$	$E_2(T)$	$E_1(T)$	$E_0(T)$
B_3	1	1	–	–	–	1	–	–	–
$B_2 B_0$	2	–	1	–	1	1	–	–	–
$B_2 B_1$	3	–	1	1	–	1	–	–	–
$B_2 B_1' B_0'$	4	–	1	0	0	–	1	–	–
$B_2' B_0$	5	–	0	–	1	–	1	–	–
$B_2' B_1$	6	–	0	1	–	–	1	–	–
$B_1' B_0'$	7	–	–	0	0	–	–	1	–
$B_1 B_0$	8	–	–	1	1	–	–	1	–
B_0'	9	–	–	–	0	–	–	–	1

PLA diagram:



Programmable array logic (PAL).

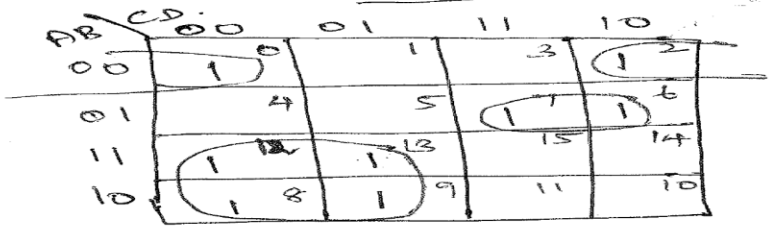
$W(A,B,C,D) = \Sigma(0,2,6,7,8,9,12,13)$

$X(A,B,C,D) = \Sigma(0,2,6,7,8,9,12,13,14)$

$Y(A,B,C,D) = \Sigma(2,3,8,9,10,12,13)$

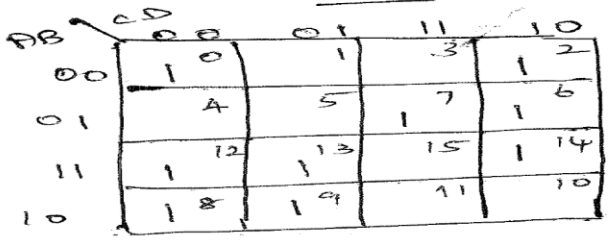
$Z(A,B,C,D) = \Sigma(1,3,4,6,9,12,14)$

For W



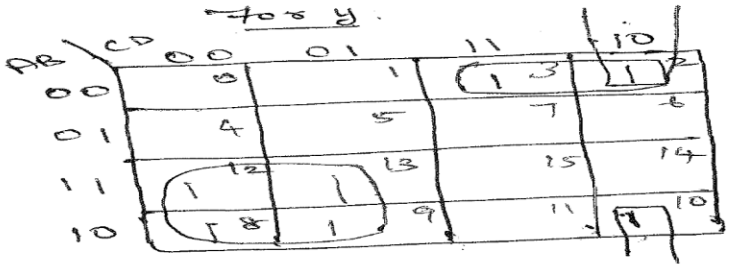
$W = \bar{A}\bar{B}\bar{D} + \bar{A}BC + A\bar{C}$

For X



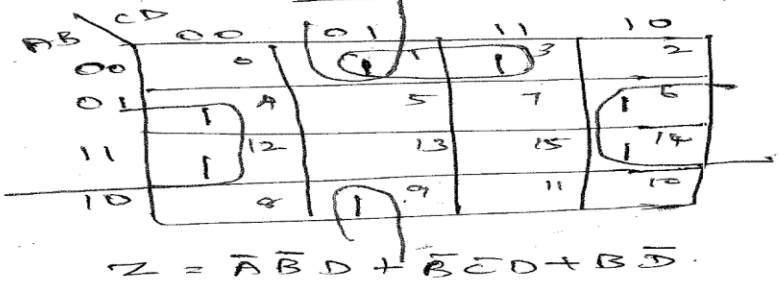
$X = \bar{A}\bar{B}\bar{D} + \bar{A}BC + A\bar{C} + BC\bar{D}$

For Y



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

For Z



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

Product term	AND inputs				outputs
	A	B	C	D	W
1. $\bar{A}\bar{B}\bar{D}$	0	0	–	0	–
2. $\bar{A}BC$	0	1	1	–	–
3. $A\bar{C}$	1	–	0	–	–
4. W	–	–	–	–	1
5. $BC\bar{D}$	–	1	1	0	–
6. –	–	–	–	–	–
7. $\bar{A}\bar{B}C$	0	0	1	–	–
8. $\bar{B}C\bar{D}$	–	0	1	0	–
9. $A\bar{C}$	1	–	0	–	–
10. $\bar{B}\bar{D}$	0	0	–	1	–
11. $\bar{B}\bar{C}D$	–	0	0	1	–
12. $B\bar{D}$	–	1	–	0	–

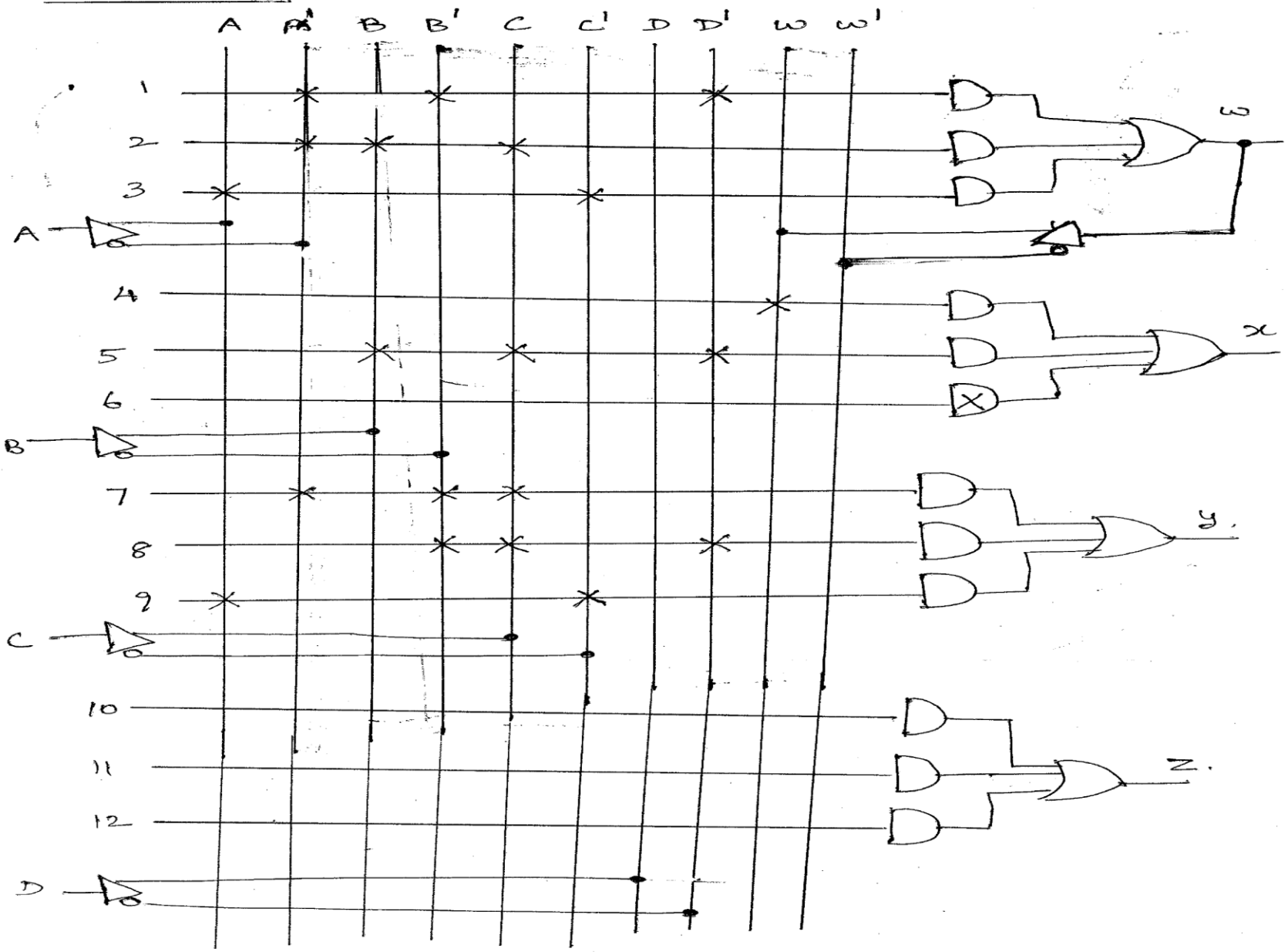
$W = \bar{A}\bar{B}\bar{D} + \bar{A}BC + A\bar{C}$

$X = W + BC\bar{D}$

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

$Z = \bar{A}\bar{B}D + \bar{B}\bar{C}D + B\bar{D}$.

PAL diagram.



VHDL

- VHDL: VHSIC Hardware Description Language

- VHSIC --
- Very High Speed Integrated Circuits

VHDL

- It is a programming language that allows one to model and develop complex digital systems in a dynamic environment.
- Object Oriented methodology for people using C can be observed -- modules can be used and reused.
- Allows to designate in/out ports (bits) and specify behavior or response of the system.

VHDL

- **VHDL** is an acronym for **V**ery high speed integrated circuit **(VHSIC) H**ardware **D**escription **L**anguage.
- It is a programming language that describes a logic circuit by function, data flow behavior, and/or structure.
- Hardware description is used to configure a programmable logic device (**PLD**), such as a field programmable gate array (**FPGA**), with a custom logic design.
- The general format of a VHDL program is built around the concept of **BLOCKS** which are the basic building units of a VHDL design.
- A VHDL design begins with an **ENTITY** block that describes the interface for the design.
- The interface defines the input and output 1ogic signals of the circuit being designed.

VHDL

- The **ARCHITECTURE** block describes the internal operation of the design.
- Design created can be simulated and synthesized to check its logical operation.
- **SIMULATION** is a bare bones type of test to see if the basic logic works according to design and concept.
- **SYNTHESIS** allows timing factors and other influences of actual field programmable gate array (**FPGA**) devices to effect the simulation thereby doing a more thorough type of check before the design is committed to the **FPGA** or similar device.

Modeling Digital Systems

VHDL is for coding models of a digital system...

Reasons for modeling

- requirements specification
- documentation
- testing using simulation
- formal verification
- synthesis
- class assignments

Goal

- most ‘reliable’ design process, with minimum cost and time.
- avoid design errors!

Data Types

- VHDL is a very **strongly** typed language.
- It does not allow a lot of intermixing of data types.
- It describes the different types of data that can be used with VHDL which include bits, buses, Boolean, strings, real and integer number types, physical, and user defined enumerated types.

VHDL Data Types

- VHDL has a set of standard data types (predefined / built-in).
- It can also have user defined data types and subtypes.
- Some of the predefined data types in VHDL are: BIT, BOOLEAN and INTEGER.
- The STD_LOGIC and STD_LOGIC_VECTOR data types are not built-in
- VHDL data types, but are defined in the standard logic 1164 package of the IEEE library.
- Thus we need to include this library in our VHDL code and specify that the STD_LOGIC_1164 package must be used in order to use the STD_LOGIC data type:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

VHDL Data Types

BIT

- The BIT data type can only have the value 0 or 1. When assigning a value of 0 or 1 to a BIT in VHDL code, the 0 or 1 must be enclosed in single quotes: '0' or '1'.

BIT_VECTOR

- The BIT_VECTOR data type is the vector version of the BIT type consisting of two or more bits. Each bit in a BIT_VECTOR can only have the value 0 or 1.
- When assigning a value to a BIT_VECTOR, the value must be enclosed in double quotes, e.g. "1011" and the number of bits in the value must match the size of the BIT_VECTOR.

VHDL Data Types

STD_LOGIC

- The STD_LOGIC data type can have the value X, 0, 1 or Z. There are other values that this data type can have, but the other values are not synthesizable – i.e. they can not be used in VHDL code that will be implemented on a CPLD or FPGA.
- These values have the following meanings:
- X – unknown
- 0 – logic 0
- 1 – logic 1
- Z – high impedance (open circuit) / tristate buffer
- When assigning a value to a STD_LOGIC data type, the value must be enclosed in single quotes: 'X', '0', '1' or 'Z'.

STD_LOGIC_VECTOR

- The vector version of the STD_LOGIC data type. Each bit in the set of bits that make up the vector can have the value X, 0, 1 or Z.
- When assigning a value to a STD_LOGIC_VECTOR type, the value must be enclosed in double quotes, e.g. "1010", "ZZZZ" or "ZZ001". The number of bits in the value must match the size of the STD_LOGIC_VECTOR.

Basic VHDL Concepts

- Interfaces -- i.e. ports
- Behavior
- Structure
- Test Benches
- Analysis, simulation
- Synthesis

3 ways to DO IT -- the VHDL way

- Dataflow
- Behavioral
- Structural

VHDL Modelling Styles

VHDL Modelling Styles:

- **Behavioral,**
- **Dataflow,**
- **Structural**

Body is used to describe the type of modelling used:

- Behavior: As a set of sequential assignment statement.
- Dataflow: As set of concurrent assignment statement.
- Structure: As set of interconnected compare.
- Mixed: As set of combination of above three.

- A behavioral architecture uses only process statements.
- A dataflow architecture uses only concurrent signal assignment statements.
- A structural architecture uses only component instantiation statements.

Concurrent vs Sequential Statements

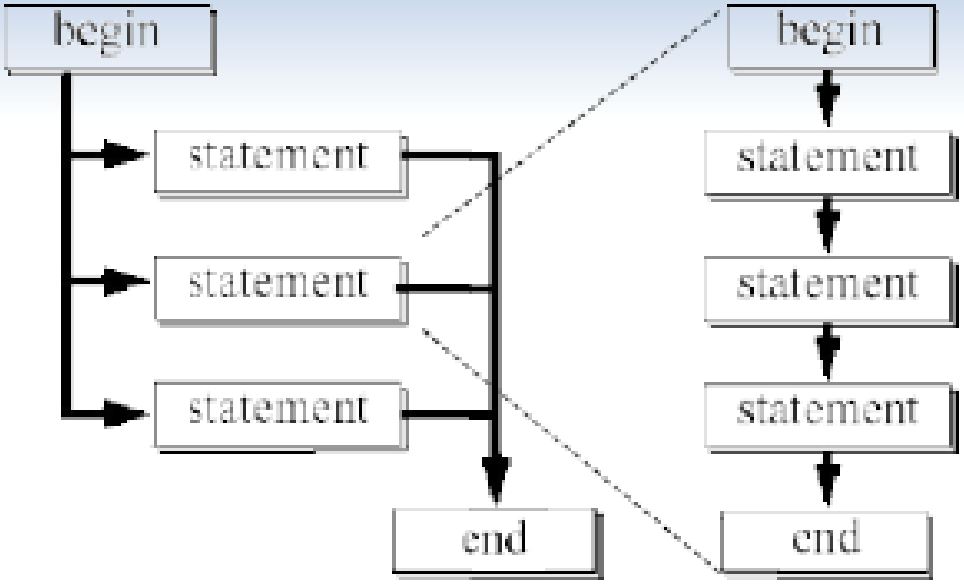
Concurrent statements

- Simple signal assignment statement
- Conditional signal assignment statement
- Selected signal assignment statement

Sequential Statements

- VHDL process
- Sequential signal assignment statement
- Variable assignment statement
- If statement
- Case statement
- Simple for loop statement

Concurrent vs Sequential VHDL



Modeling Style	Concurrent	Sequential
Location	inside architecture	inside process
Example statements	process, component instance, concurrent signal assingment	if, for, switch-case, signal assignment

Sequential Statements

The sequential statements that may appear in a process or subprogram are presented:

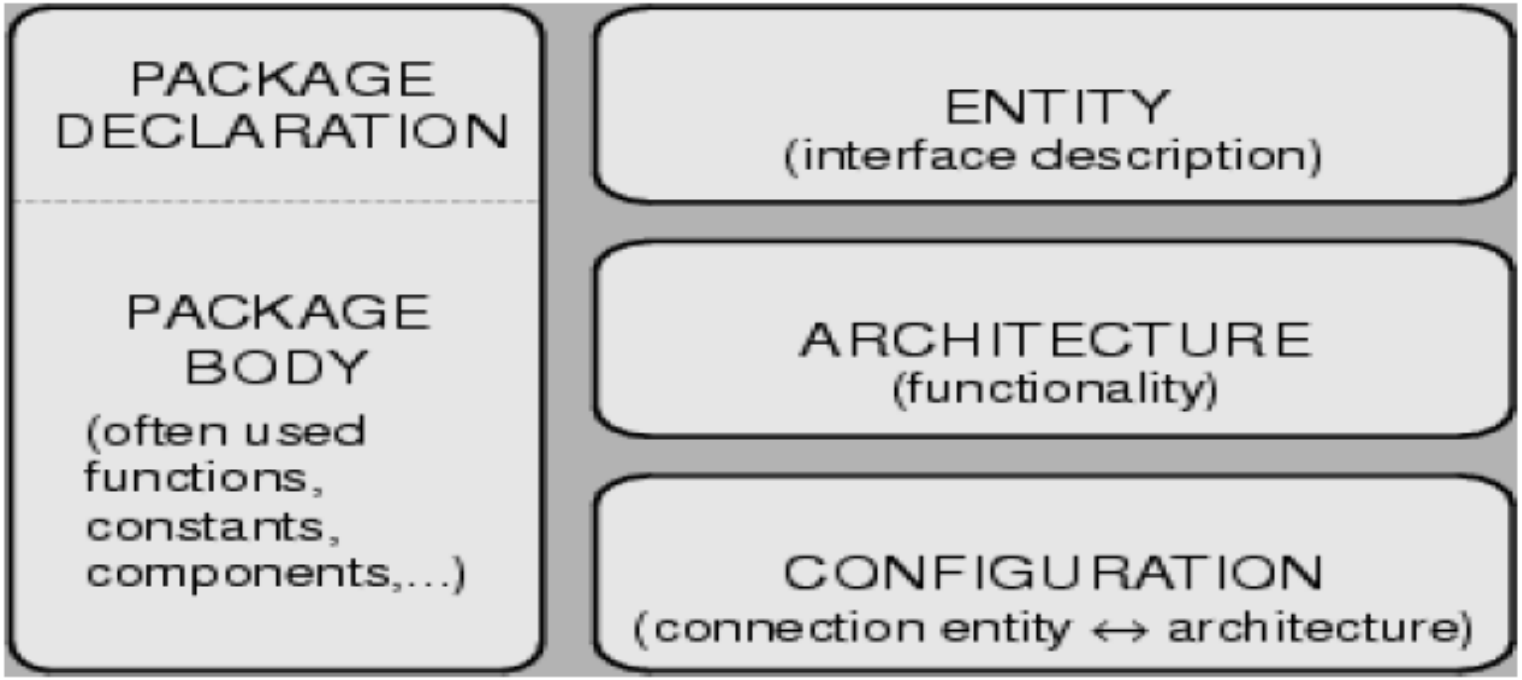
- sequential signal assignment,
- variable assignment,
- if statement,
- case statement,
- loop statements (loop, while loop, for loop, next, exit), and
- sequential assert statement.
- procedure call statement and
- the return statement from a procedure or function.

Concurrent Statements

The most important concurrent statement is

- the process declaration.
- the con-current signal assignment statement,
- the block statement,
- the concurrent assert statement,
- the concurrent procedure call statement,
- the component instantiation statement, and
- the generate statement.

VHDL Program Structure



entity entity-name **is**
[port(interface-signal-declaration);]
end [entity] [entity-name];

architecture architecture-name **of** entity-name **is**
[declarations]
begin
architecture body
end [architecture] [architecture-name];

Modeling the Dataflow way

- uses statements that defines the actual flow of data.....
  - such as,
    - x <= y -- this is NOT less than equal to as in C.
    -
- This assigns the boolean signal x to the value of boolean signal y... i.e. x = y
- This will occur whenever y changes....

Jumping right in to a Model -- e.g. 1

- lets look at a d - flip-flop model -- doing it the dataflow way..... ignore the extra junk for now --

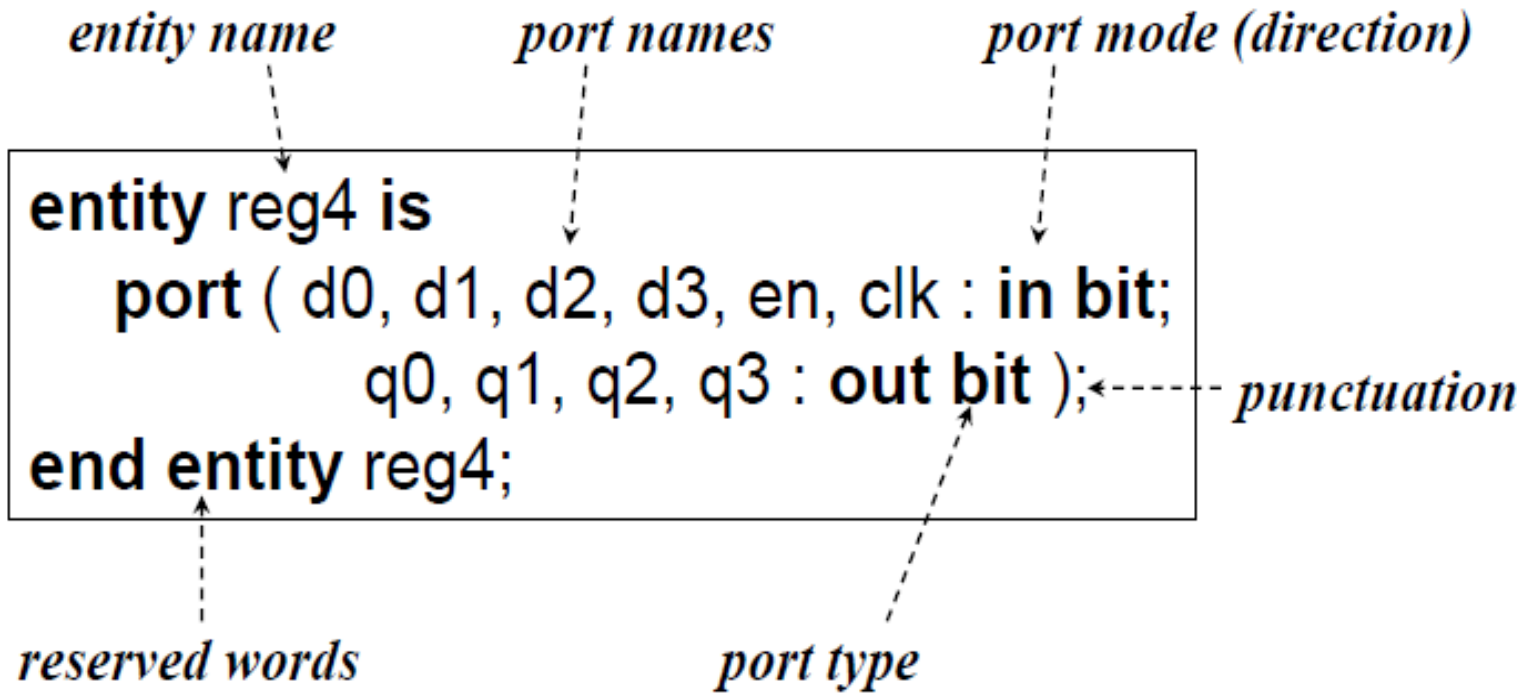
```
entity dff_flow is
   port (          d          :in     bit;
                   prn        :in     bit;
                   clrn       :in     bit;
                   q          :out    bit;
                   qbar       :out    bit;
         );
end dff_flow;
architecture  arch1 of dff_flow  is
begin
   q  <=  not prn    Or   (clrn  And  d);     % this is  the DATAFLOW %
   qbar <= prn And  (not clrn  Or not d);    %  STUFF    %
end arch1;
```

Jumping right in to a Model -- e.g. 1

```
library ieee; use ieee.std_logic_1164.all;

entity fulladd is
        port(A1,A2,Cin: IN std_logic;
                Sum, Cout: OUT std_logic);
end fulladd;

Architecture a of fulladd is
Begin
        process(A1,A2,Cin)
        Begin
           Sum <= Cin XOR A1 XOR A2;
            Cout <= (A1 AND A2) OR (Cin AND (A1 XOR A2));
        end process;
end a;
```

Modeling Interfaces

- *Entity* declaration
  - describes the input/output *ports* of a module



Modeling the Behavior way

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral* architecture
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
      - *sequential statements*, including
        - » *signal assignment statements* and
        - » *wait statements*

The Behavior way -- eg 2

```
architecture behav of reg4 is
begin
    process (d0, d1, d2, d3, en, clk)
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
        if en = '1' and clk = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;
        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
    end process;
end behav;
```

sensitivity list

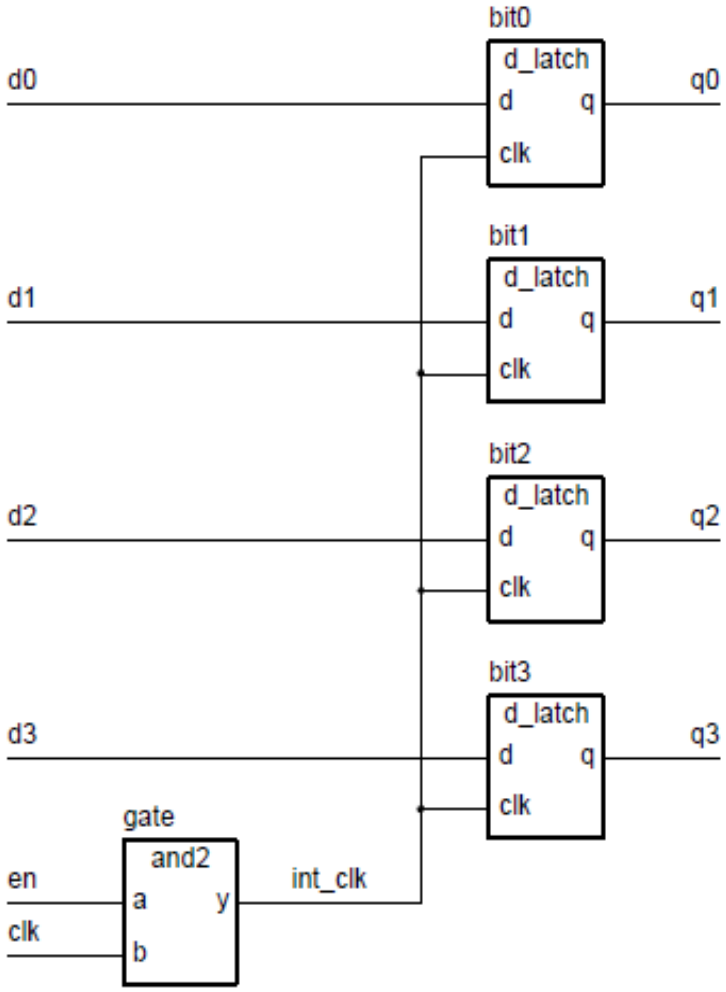
notice := syntax
used for equating values
from signals...

simulates real-world
propagation delays.

Modeling the Structurural way

- *Structural* architecture
  - implements the module as a composition of subsystems
  - contains
    - *signal declarations*, for internal interconnections
      - the entity ports are also treated as signals
    - *component instances*
      - instances of previously declared entity/architecture pairs
    - *port maps* in component instances
      - connect signals to component ports

Structural way -- e.g. 3



Structural way cont..

- First declare D-latch and and-gate entities and architectures

notice semicolon placements -- odd as it is, omit from last statement

```
entity d_latch is
    port ( d, clk : in bit;  q : out bit );
end entity d_latch;

architecture basic of d_latch is
begin
    process (clk, d)
    begin
        if clk = '1' then
            q <= d after 2 ns;
        end if;
    end process;
end basic;
```

```
entity and2 is
    port ( a, b : in bit;  y : out bit );
end entity and2;

architecture basic of and2 is
begin
    process (a, b)
    begin
        y <= a and b after 2 ns;
    end process ;
end basic;
```

Structural way...

- Declare corresponding components in register architecture body

```
architecture struct of reg4 is
    component d_latch
        port ( d, clk : in bit;  q : out bit );
    end component;
    component and2
        port ( a, b : in bit;  y : out bit );
    end component;
    signal int_clk : bit;
...
```

Structural way..

- Now use them to implement the register

```
...
begin
    bit0 : d_latch
        port map ( d0, int_clk, q0 );
    bit1 : d_latch
        port map ( d1, int_clk, q1 );
    bit2 : d_latch
        port map ( d2, int_clk, q2 );
    bit3 : d_latch
        port map ( d3, int_clk, q3 );
    gate : and2
        port map ( en, clk, int_clk );
end struct;
```

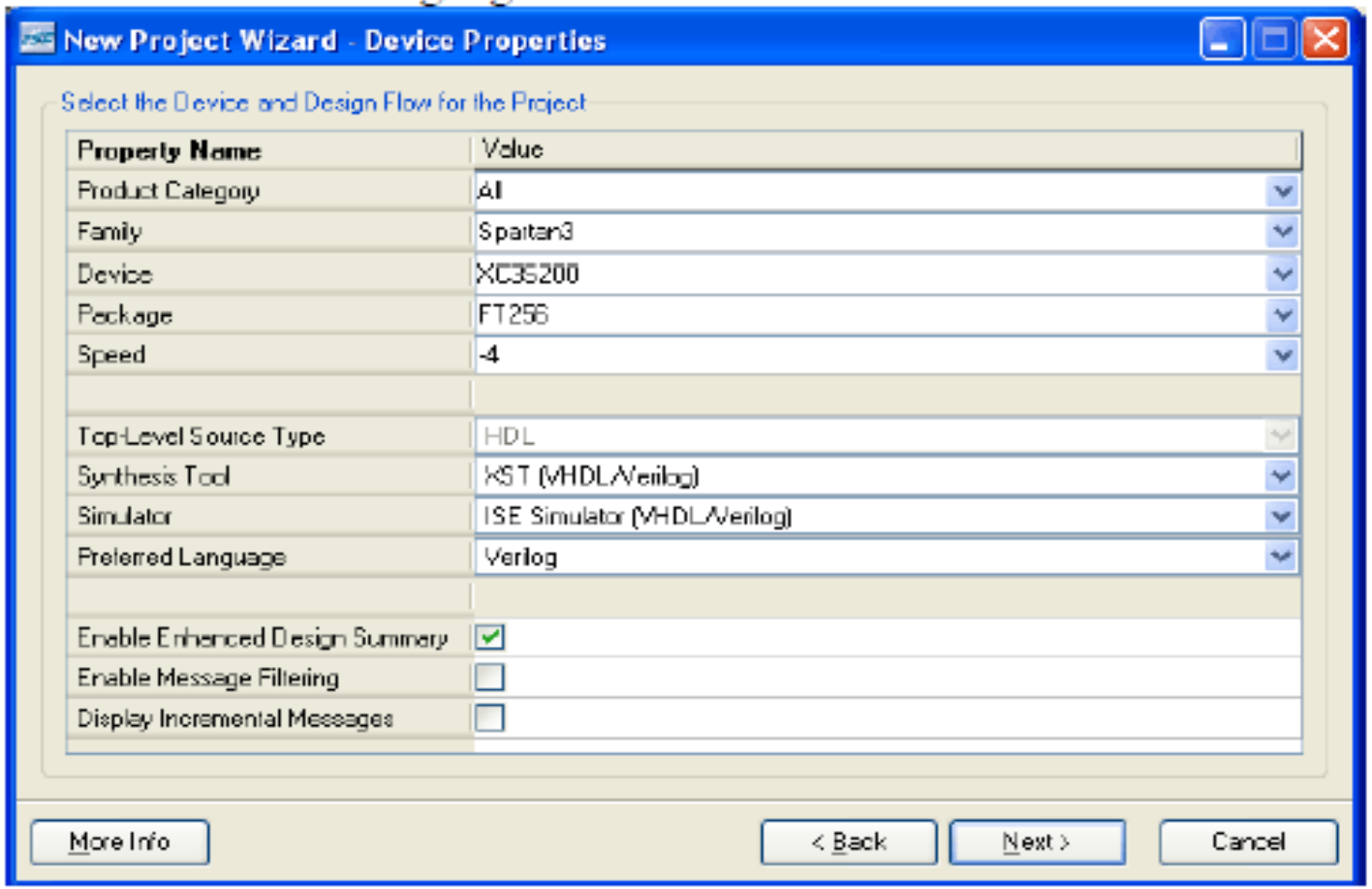
VHDL Execution Process

VHDL Examples: (Language Templates → Coding Examples

Process Steps:

1. File → New Project → Project Name: Gate1
2. Device Properties:

   Top-Level Source Type = HDL
   Preferred Language = VHDL



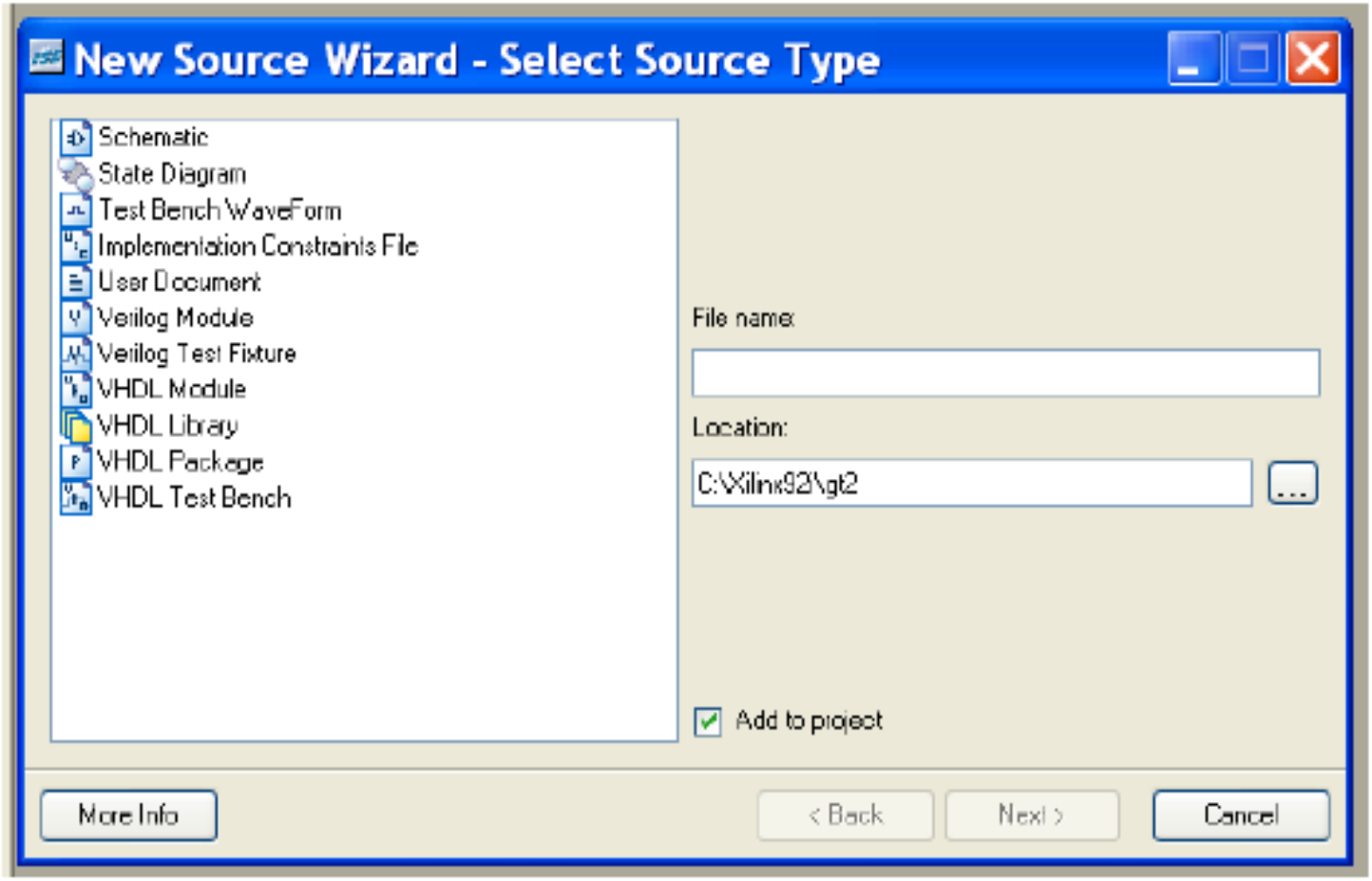
3. Press Next Key.
4. Create New Source Window Displayed : Click New Source
5. Select VHDL module & give any filename: for e.g. Gate2 & press Next.
6. Define Module:

A	in	
B	in	
Y	out	& press Next

7. Summary Window is displayed, select Finish & save file by selecting Yes.
8. Press Next & further again Next.
9. Displays Project Summary, Press Finish.
10. In Code area: Add the code for 'AND' gate as given below:

```
architecture behavioural of Gate2 is
begin
        Y<= A AND B;
end behavioural;
```

11. From above left hand side, select **synthesis/ Implementation** from **Sources For Menu** .
12. Select Processes bar, and from this select **Implement Design** and further select Synthesize, from this double click on **check syntax**, to check for the syntactical errors in your code. Save the file & double click Synthesize-XST to generate HDL synthesize report for the gate structure.
13. From above left hand side, select **Behavioral Simulation** from **Sources For Menu**.
14. Select Filename Gate2 then right click & select new source



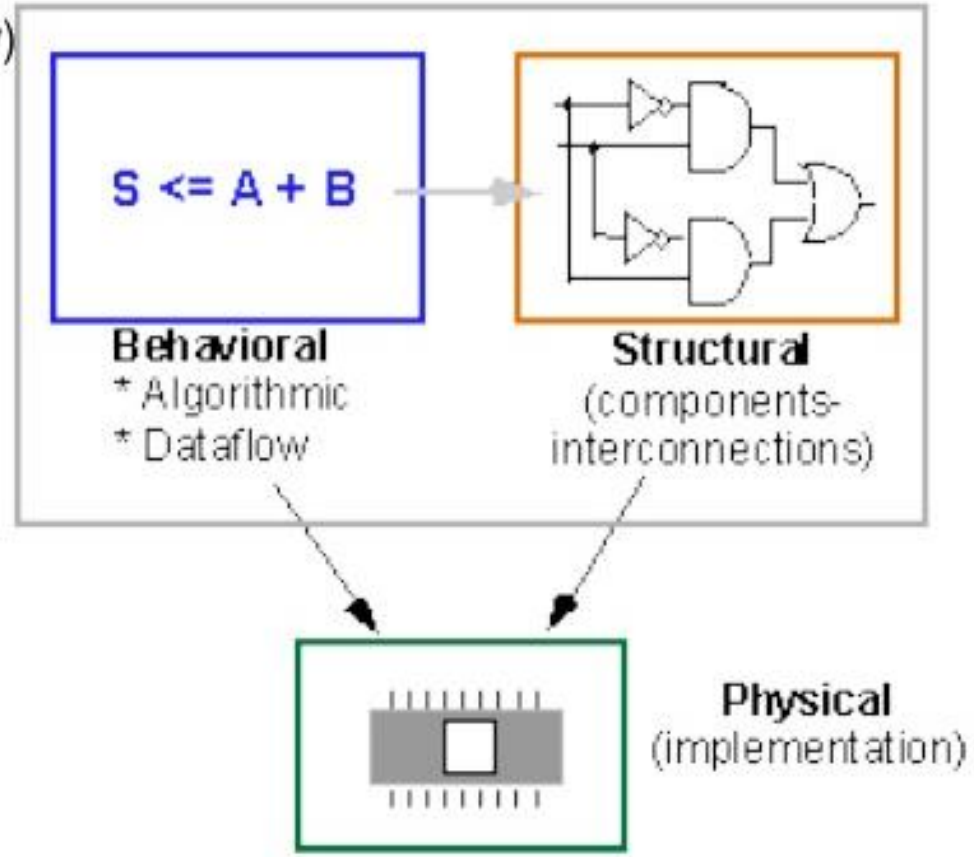
15. From the above menu select Test Bench Waveform & assign a name other than the entity name as Gateinput & Press Next: 2 times & see the summary of the code and click finish.
16. From Initialize Timing Menu select Combinational (or internal Clock) & click on finish.
17. A Wave form is displayed & file Gateinput.tbw is created.
18. Save the file & from processes bar select Simulate Behavioral Mode & double click it & save the file.

VHDL Coding Basics

TYPES OF REPRESENTATION

There are two types of representation

1.Behavorial(Dataflow)
2.Structural



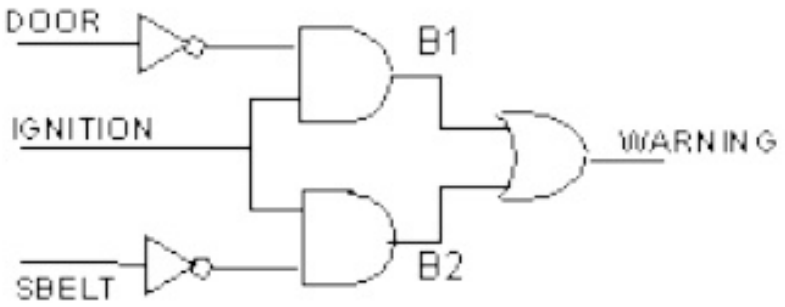
EXPLANATION OF TYPES

- Behavorial(Dataflow)

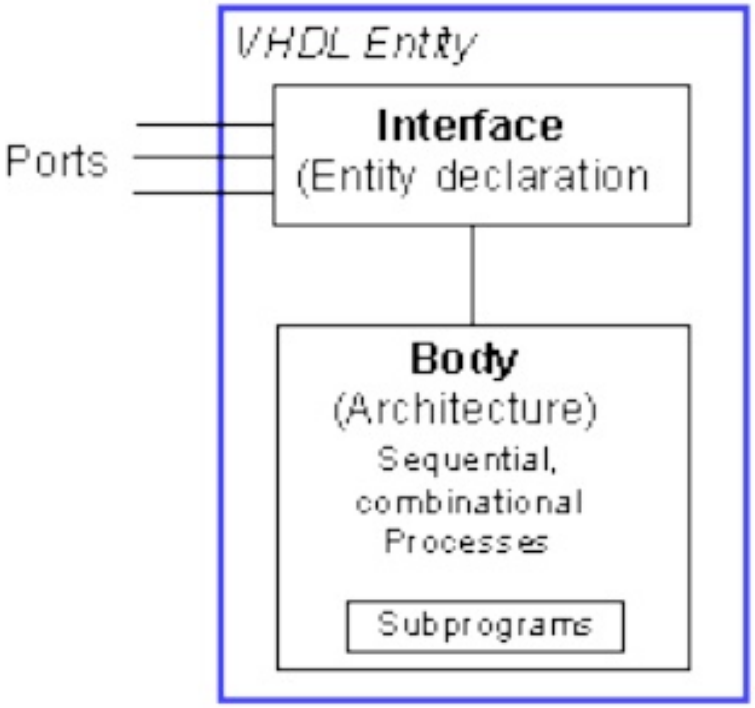
Warning = Ignition_on **AND** (Door_open **OR** Seatbelt_off)

- Structural

We should have the knowledge of the circuit which drives the logic.



BASIC STRUCTURE OF A PROGRAM



ENTITY DECLARATION

The entity declaration defines the NAME of the entity and lists the input and output ports.

```
entity BUZZER is
        port (DOOR, IGNITION, SBELT: in std_logic;
           WARNING: out std_logic);
    end BUZZER;
```

ARCHITECTURE BODY

- The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

- **architecture** dataflow **of** BUZZER **is**

begin

WARNING <= (**not** DOOR **and** IGNITION) **or** (**not** SBELT **and** IGNITION);

end behavioral;

STRUCTURAL TYPE…..

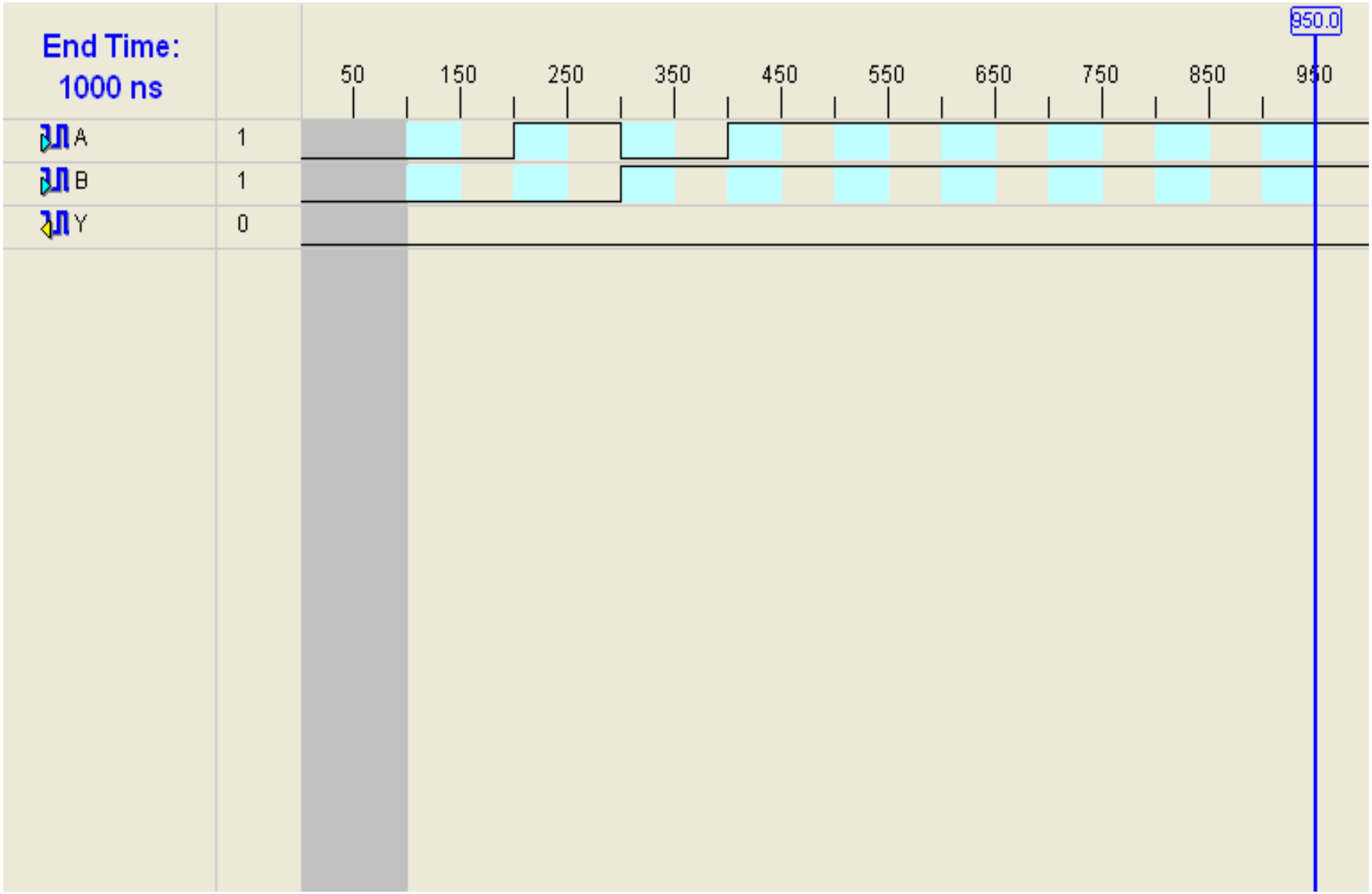
```
architecture structural of BUZZER is
        component AND2
            port (in1, in2: in std_logic;
                  out1: out std_logic);
        end component;
        component OR2
            port (in1, in2: in std_logic;
                  out1: out std_logic);
        end component;
        component NOT1
            port (in1: in std_logic;
                  out1: out std_logic);
        end component;
```

```
signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;

begin
        U0: NOT1 port map (DOOR, DOOR_NOT);
        U1: NOT1 port map (SBELT, SBELT_NOT);
        U2: AND2 port map (IGNITION, DOOR_NOT, B1);
        U3: AND2 port map (IGNITION, SBELT_NOT, B2);
        U4: OR2  port map (B1, B2, WARNING);
  end structural;
```

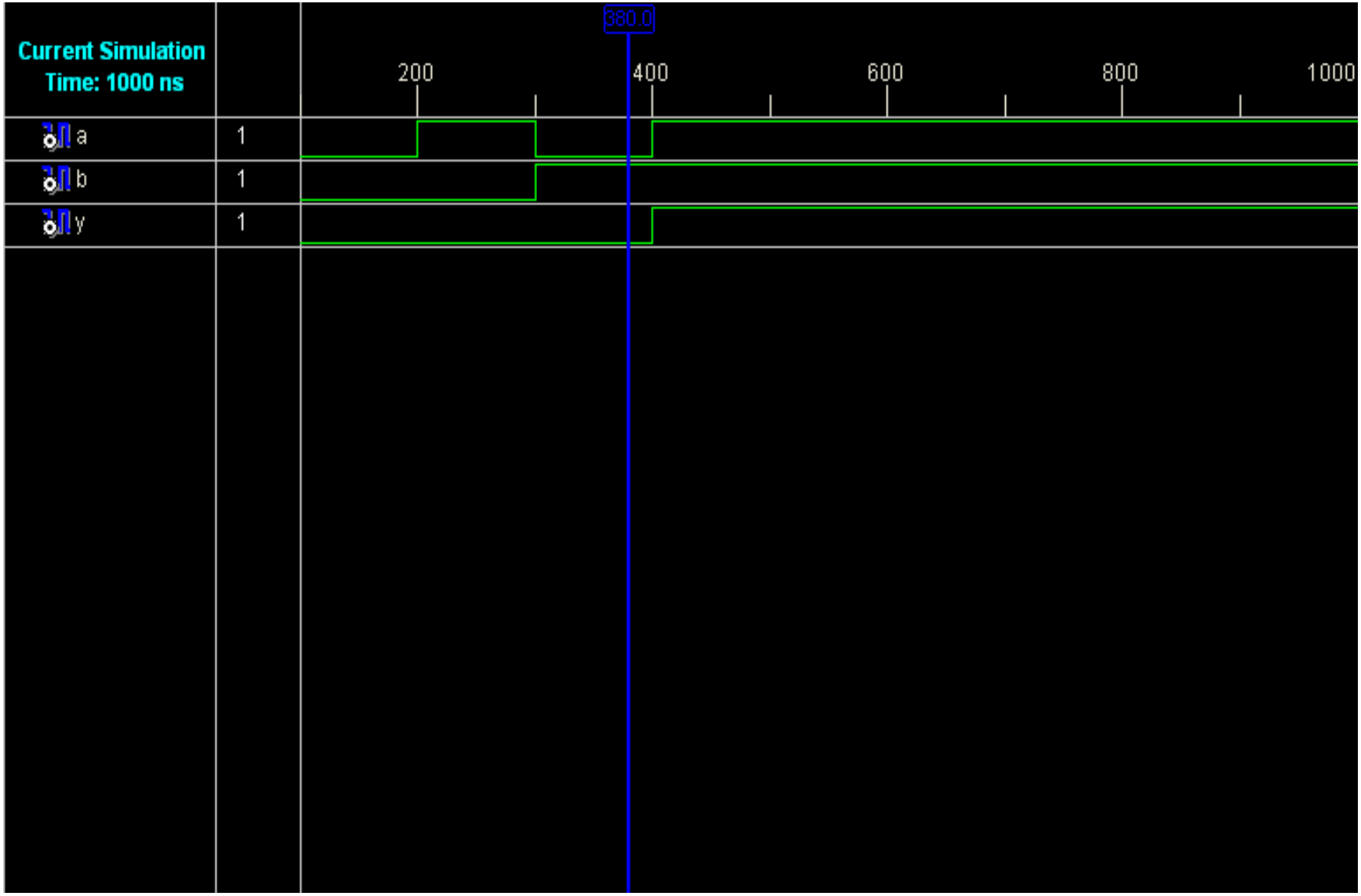
VHDL Code for AND Gate

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity ANDG is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Y : out  STD_LOGIC);
end ANDG;

architecture Behavioral of ANDG is

begin

Y <= A AND B;

end Behavioral;
```

Input to AND Gate



Output of AND Gate



Full Adder VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

 entity FA4B is
    Port ( A0 : in  STD_LOGIC;
           A1 : in  STD_LOGIC;
           A2 : in  STD_LOGIC;
           A3 : in  STD_LOGIC;
           B0 : in  STD_LOGIC;
           B1 : in  STD_LOGIC;
           B2 : in  STD_LOGIC;
           B3 : in  STD_LOGIC;
           Y0 : out  STD_LOGIC;
           Y1 : out  STD_LOGIC;
           Y2 : out  STD_LOGIC;
           Y3 : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
 end FA4B;
```

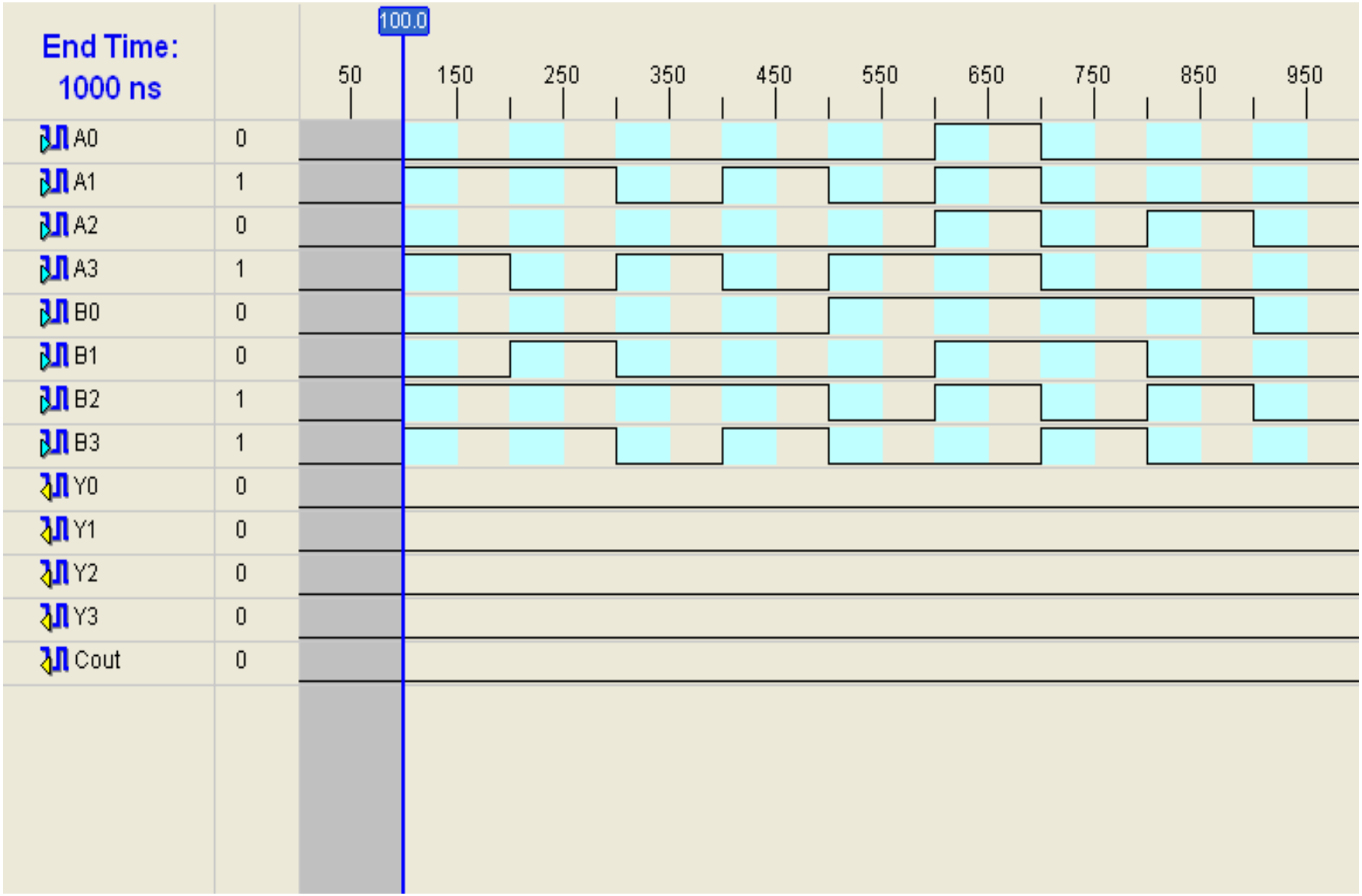
Full Adder VHDL Code

```
architecture Behavioral of FA4B is
 component FA is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           Sum : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
end component;
 signal  C1, C2, C3 : std_logic;

begin
 FA1 : FA port map ( A => A0, B => B0, Cin => '0', Sum => Y0, Cout => C1 );
FA2 : FA port map ( A => A1, B => B1, Cin => C1, Sum => Y1, Cout => C2 );
FA3 : FA port map ( A => A2, B => B2, Cin => C2, Sum => Y2, Cout => C3 );
FA4 : FA port map ( A => A3, B => B3, Cin => C3, Sum => Y3, Cout => Cout );
 end Behavioral;
```

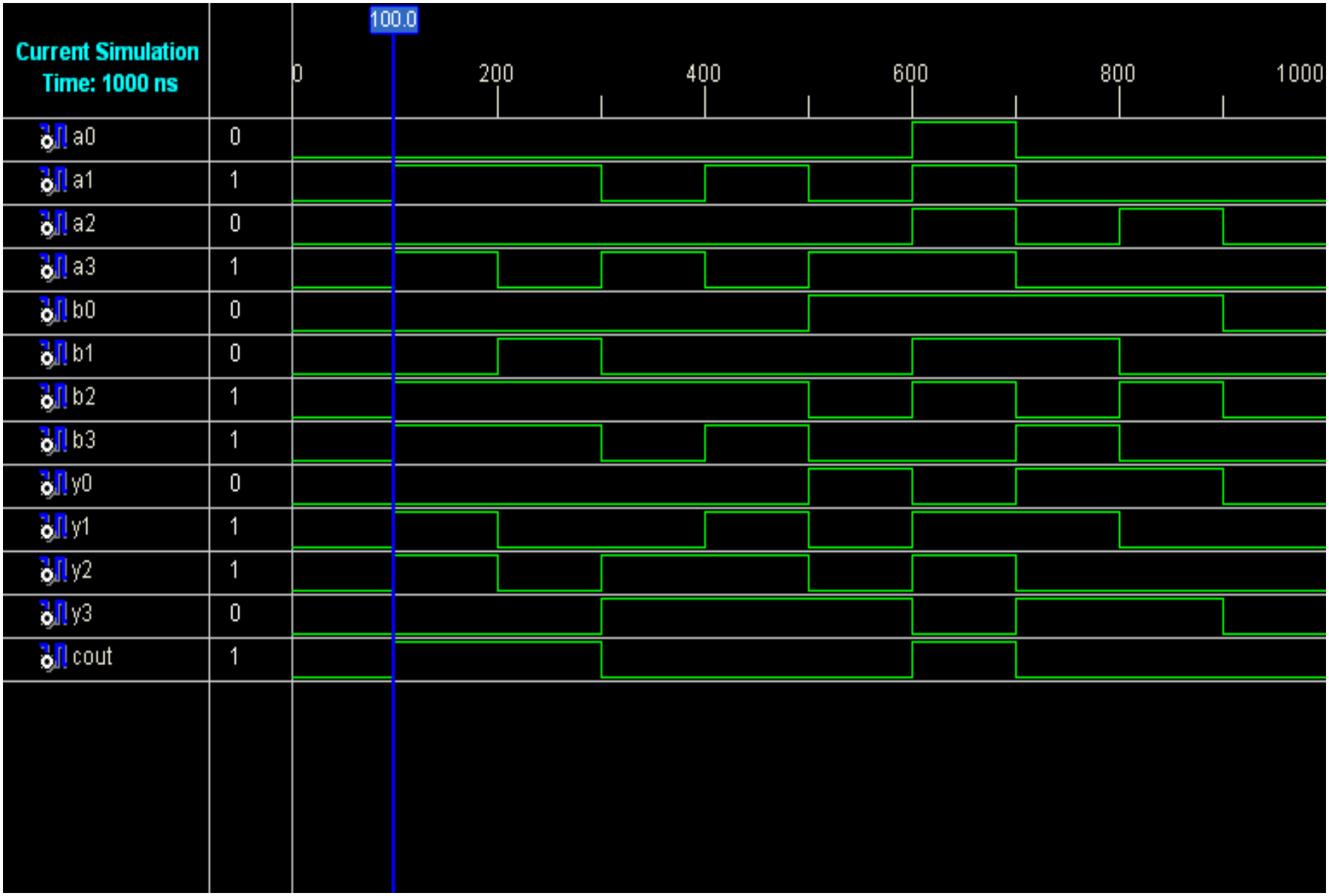
Full Adder VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FA is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           Sum : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
end FA;

architecture Behavioral of FA is

begin
Sum <= ( ( A XOR B ) XOR Cin );
Cout <= ( ( ( A AND B ) OR ( B AND Cin ) ) OR ( Cin AND A ) );
end Behavioral;
```

Full Adder Input



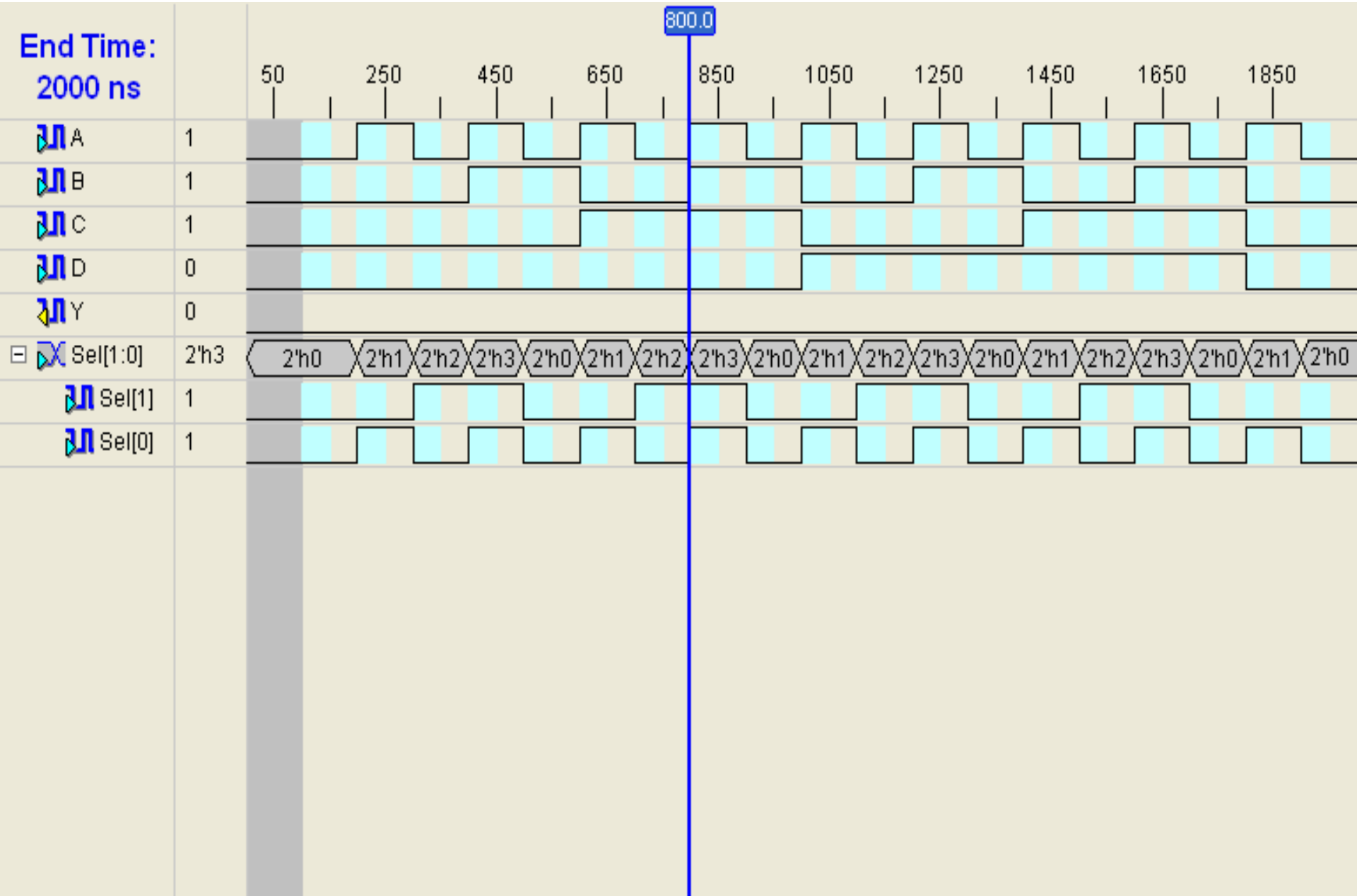
Full Adder Output



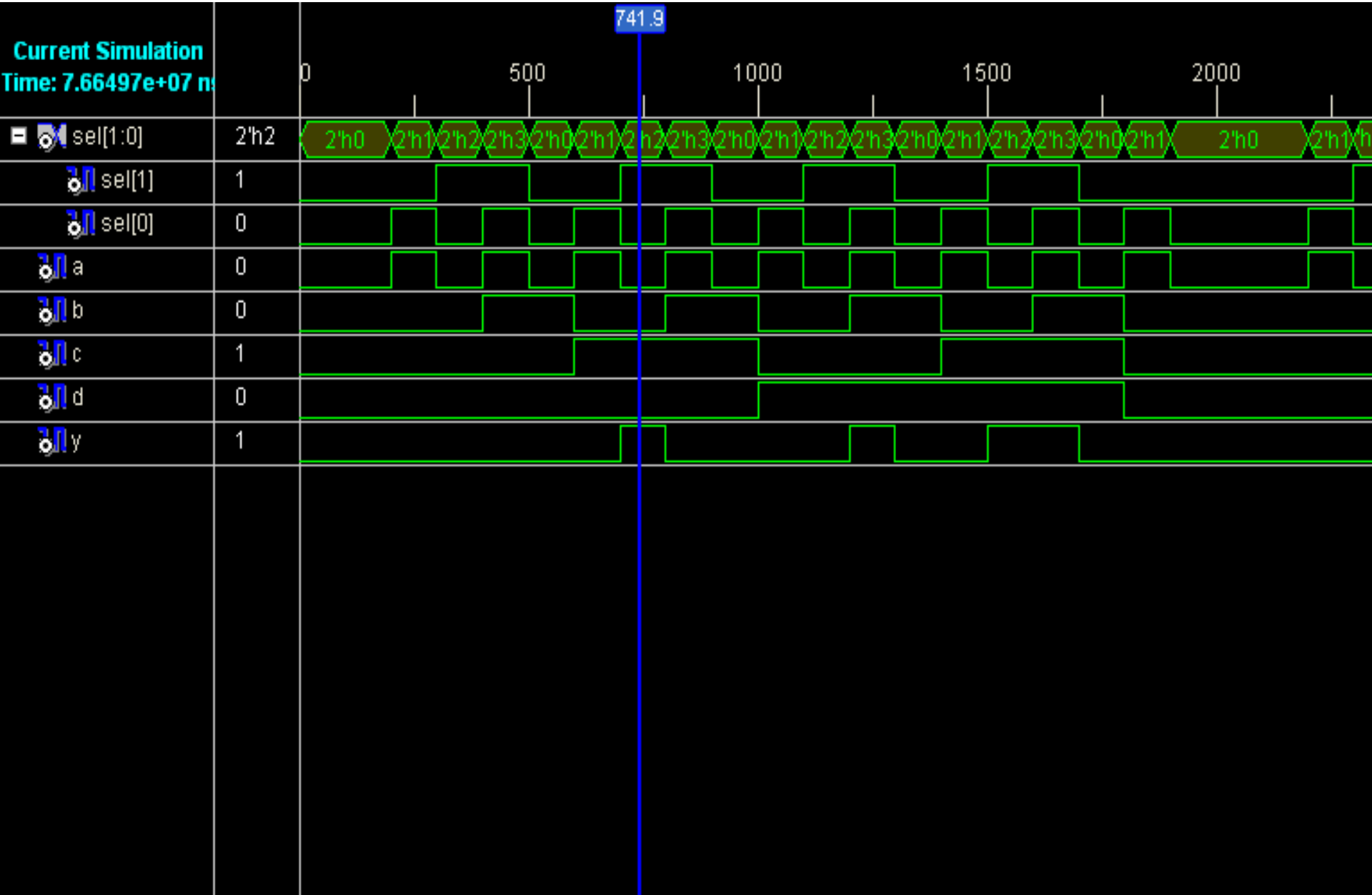
4:1 Mux VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity MUX4_1 is
    Port ( Sel :  in  STD_LOGIC_VECTOR(1 downto 0);
                      A, B, C, D : in  STD_LOGIC;
           Y : out  STD_LOGIC);
end MUX4_1;


architecture Behavioral of MUX4_1 is
begin

    process (Sel, A, B, C, D )
       begin
                                  if(Sel = "00") then y <= A;
                                  elsif (Sel = "01") then y <= B;
                                  elsif (Sel = "10") then y <= C;
                                  else  y <= D;
                                  end if;
                                  end if;
                                  end if;
    end process;
end Behavioral;
```

Input to 4:1 Mux



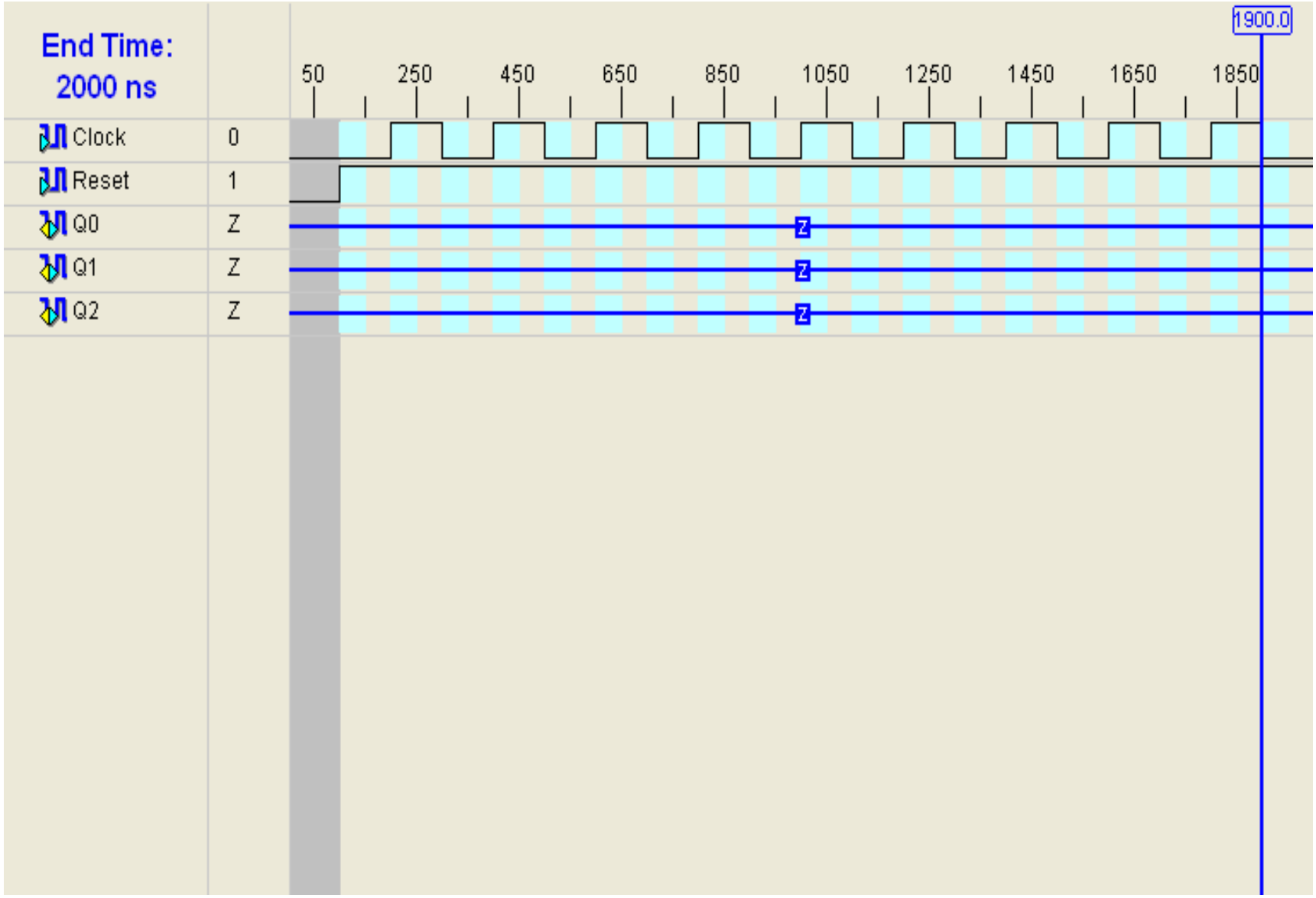
Output to 4:1 Mux



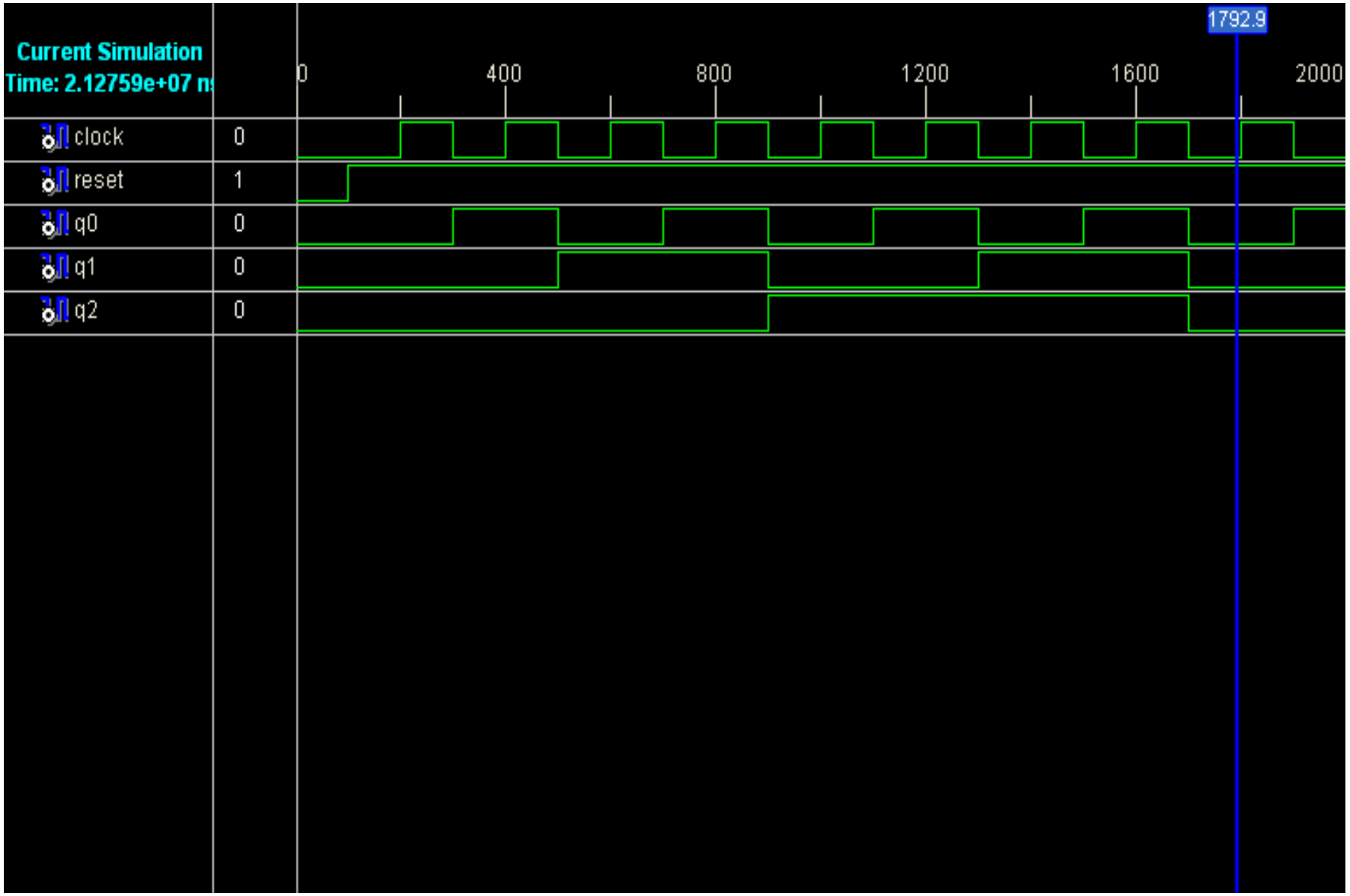
Asynchronous UP Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

 entity UP is
    Port ( Clock : in  STD_LOGIC;
           Reset : in  STD_LOGIC;
           Q0 : inout  STD_LOGIC;
           Q1 : inout  STD_LOGIC;
           Q2 : inout  STD_LOGIC);
end UP;
 architecture Behavioral of UP is

component JKFF is
    Port ( J : in  STD_LOGIC;
           K : in  STD_LOGIC;
           Clock : in  STD_LOGIC;
           Reset : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end component;

begin

JK1 : JKFF port map
( J => '1', K => '1', Clock => Clock, Reset => Reset, Q => Q0);

JK2 : JKFF port map
( J => '1', K => '1', Clock => Q0, Reset => Reset, Q => Q1);

JK3 : JKFF port map
( J => '1', K => '1', Clock => Q1, Reset => Reset, Q => Q2);

end Behavioral;
```

Asynchronous UP Counter



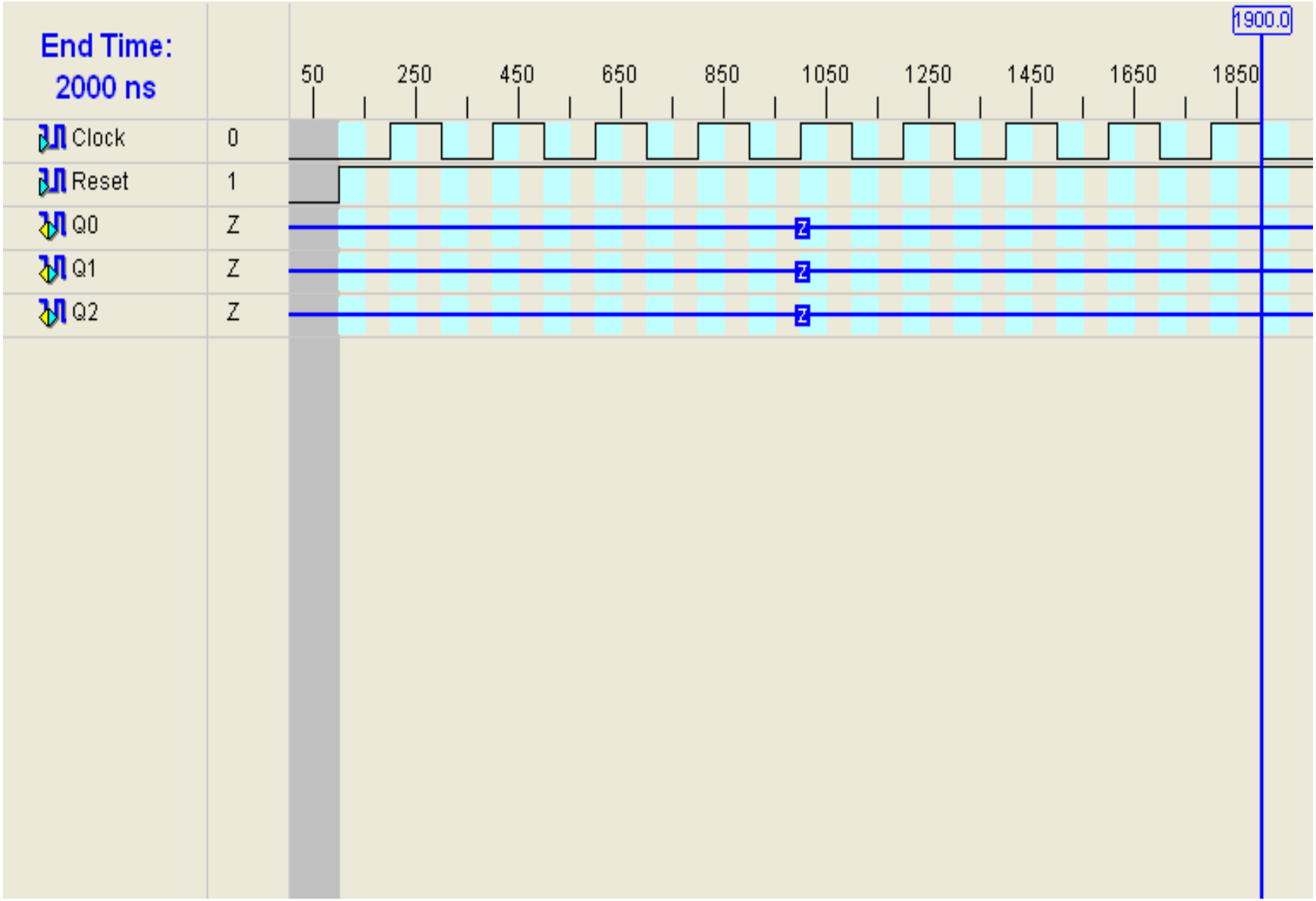
Asynchronous UP Counter



Synchronous UP Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

 entity UP is
    Port ( Clock : in  STD_LOGIC;
           Reset : in  STD_LOGIC;
           Q0 : inout  STD_LOGIC;
           Q1 : inout  STD_LOGIC;
           Q2 : inout  STD_LOGIC);
end UP;

architecture Behavioral of UP is
 component JKFF is
                        Port ( J : in  STD_LOGIC;
           K : in  STD_LOGIC;
           Clock : in  STD_LOGIC;
           Reset : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end component;

signal A : STD_ULOGIC;
 begin
 JK1 : JKFF port map
( J => '1', K => '1', Clock => Clock, Reset => Reset, Q => Q0);

JK2 : JKFF port map
( J => Q0, K => Q0, Clock => Clock, Reset => Reset, Q => Q1);

A <= Q0 AND Q1;

JK3 : JKFF port map
( J => A, K => A, Clock => Clock, Reset => Reset, Q => Q2);
 end Behavioral;
```

Synchronous UP Counter



Synchronous UP Counter