Using Hardware Description Languages (HDLs)

Introduction: Combinational Logic and Its Description

Lecture Notes #1

ECE 368: CAD Based Logic Design
Besides, computers, logic circuits find the bulk of their application as the 'brain' or 'controllers' of a larger (non-logic—e.g., analog, electro-mechanical) system. These 'controllers' or 'brains' of an enormous variety of products. In such systems, embedded systems provide an important variety of products. In such systems, embedded systems or brains of an enormous variety of products, the bulk of their application as the

**Application of Logic Circuits—The Big Picture**
Application of Logic Circuits

- Computers: The brain, body and limbs of computer systems—everything in it except peripherals
- Embedded Systems: The brains that control the system (e.g., avionics, auto electronics, VCRs, microwaves, etc.)
- Digital Signal Processing (DSP): E.g., in digital cellular phones, digital TV
Logic Circuits are used to realize functions of the type:

- If input condition $C_1$ holds then do output action $A_1$
- Else if input condition $C_2$ holds then do output action $A_2$
- ...

Or have looping in these constructs. E.g.,

- Repeat if input condition $C_0$ holds then do output action $A_1$
- Else if input condition $C_2$ holds then do output action $A_2$
- Else if input condition $C_1$ holds then do output action $A_1$
- Repeat

In short, any function that can be specified by a "program" can be implemented by a combination of combinational circuits, sequential circuits and memory (besides the memory inherent in sequential circuits).

In conclusion, sequential circuits are used to realize functions of the type:
Conditions can be of the type:

- The temp. 70 degrees F
- The time 12:30 pm
- Value of variables $A_1A_2 = 10$

Actions $A_i$ can be of the type:

- Turn on the light
- Tape the program on channel 7
- Turn the furnace on

- Value of variables $A_1A_2 = 10$
- The time 12:30 pm
- The temp. 70 degrees F
The Role of Logic—An Introduction

Boolean Logic: There are only 2 values associated with a statement or computation—True (T) or False (F).

Switching Logic: T is represented by 1 and F is represented by 0.

Example: \( A \text{ AND } B \equiv (T) \text{ OR } (\text{NOT}(B)) \). Similarly, \( A \text{ OR } B \equiv (T) \text{ OR } (\text{NOT}(B)) \).

Example: \( A \text{ OR } (\text{NOT}(B)) \). This is a class of bright students.

Compound Statement: Formed of atomic or simple statements that are joined together by the 3 basic operators: AND, OR, NOT. It is also called a logic expression.

Example: \( A \text{ AND } B \equiv (T) \text{ OR } (\text{NOT}(B)) \). Today is a nice day (T) OR (This is a class of bright students). Today is a nice day (T).

Statement B: This is a class of bright students \( T \). Statement A: Today is a nice day \( T \).

Digital system design is thus also termed logic design.
Introduction: The Role of Logic (contd.)

- **Defn.** A logic function \( f(A_1, A_2, \ldots, A_n) \), where \( A_1, \ldots, A_n \) are logic variables that can take 1/0 values, is one which has either 1 or 0 values as its output depending on the input combination of its 1/0 values.

- **Fact:** Any logic function can be represented by a combination of AND, OR, NOT operators on the input variables.

- **Fact:** Any logic function can also be represented by a Truth Table (TT).

- **Fact:** A Truth Table is a tabular representation of a logic function (or operation) where the output value (1/0) is indicated for each input 1/0 combination.

- **Examples for the AND, OR, NOT functions:**

- **Defn.** A logic circuit is a circuit that realizes a logic function \( f(A_1, \ldots, A_n) \) "electrically". It does so by electrically representing 1's and 0's as voltage levels, and manipulating them to produce 1 or 0 at the output as required by function \( f \).
**More Examples: A Generic 2-input Function**

<table>
<thead>
<tr>
<th>(\overline{A} \overline{B} )</th>
<th>(A )</th>
<th>(B)</th>
<th>(A \vee B)</th>
<th>(A \cdot B)</th>
<th>(\overline{A})</th>
<th>(\overline{B})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>I</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\overline{A} \overline{B} )</th>
<th>(A)</th>
<th>(B)</th>
<th>(A \vee B)</th>
<th>(A \cdot B)</th>
<th>(\overline{A})</th>
<th>(\overline{B})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>1</td>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\(f(A, B) = \overline{A} \overline{B} + A \cdot B\)
there are different \( n \)-i/p functions one for each distinct function—thus if this has \( 2^n \) input combinations—this has \( 2^n \) input combinations.

In general, an \( n \)-i/p function \( TT \) has

- thus \( 3 \)-i/p \( TT \) can have \( 2^8 = 256 \) possible output combinations in its
- thus \( 8 \) rows
- thus \( 2^3 = 8 \) input combinations
- A generic \( 3 \)-i/p function:

\[
\begin{array}{c|cccc}
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Problem: Design a logic circuit that allows a light bulb to be controlled by toggling any one of the two provided switches $S_1$, $S_2$. 

A Simple Design Problem
ADesignProblem (Contd.)

DesignSteps:

1. Encodetheinput(s) and output(s) of the logic circuit as logic variables so that C and A are logic expressions

   Input Encoding:

   Switch S1 is pushed: Logic input s1 = 1
   Switch S1 is not pushed: Logic input s1 = 0

   Similarly for switch S2 and Logic input variables s2:

   Switch S1 is pushed: Logic input s2 = 1
   Switch S1 is not pushed: Logic input s2 = 0

   Output Encoding:

   0 = Z
   1 = Z

   Turn the (electro-mechanical) switch SW off ≡ Logic output Z

   Turn the (electro-mechanical) switch SW on ≡ Logic output Z

   Output Encoding:

   \( \text{NOT}(s_1) \) or \( s_1 \)

   Imp. Note: The term \( s_1 \) can be more simply represented as the corresponding single input variable—this may not always be the case (in this case we are using the same name for both the input sensor and the corresponding single input variable).

   Similarly, for switch S2 and Logic input variable s2:

   \( s_2 \) is not pushed ≡ Logic input s2
   \( s_2 \) is pushed ≡ Logic input s2

   Input Encoding:

   1. Encode the input(s) and output(s) of the logic circuit as Logic variables so that C and A are logic expressions

A Design Problem (Contd.)
terms of the given specification.

The $TT$ obtained assuming the light should be OFF when both switches are 0 (another $TT$ will be obtained if the assumption is that the LIGHT should be ON when both switches are 0) should be used when both switches are 0; both lead to correct designs in

NOTE: This $TT$-based approach should really be used only for small design problems (up to, say, 6 input variables). For larger problems, one needs to think in a more divide-and-conquer/hierarchical manner. Will later see these latter approaches for the design of arithmetic circuits.

1. After analyzing the requirements and from the $TT$ obtain its logic expressions.
2. Get $TT$s for each output variable (i.e., for each logic function required).
called a canonical sum-of-product (SOP) expression. Each row in a TT corresponds to an AND term or prod-

More compactly, \( Z = \overline{s_1} s_2 + (\overline{s_1} s_2) \), product form re-
ic-

The output column is the OR of all the prod-
ic-

\[ Z = \overline{s_1} s_2 + (\overline{s_1} s_2) \]

The output column is the OR of all the prod-
ic-

Each row in a TT corresponds to an AND term or prod-
ic-

\begin{tabular}{c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 \\
I & 0 & I & 0 & 0 & 0 \\
I & I & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
Z & s_1 & s_2 & s_1 & s_2 & s_1 \\
\end{tabular}
3. Minimize the logic expression(s) using various methods like algebraic manipulation, K-Maps, Quine-McCluskey, and Petrick's algorithm. The minimized form for Z is already in minimized form.

4. Implement the logic expression(s) using switches or gates (or some other technologies like PLA, multiplexers, ROMs—will do later).
Describing Combinational Ckts and Data Paths Using Programming-Language Constructs - Concepts

How can this combinational circuit be described using programming language type constructs?

Ans. Three ways ranging from the abstract level which has no implementation details to the implementation level that specifies the interconnections between modules/gates but does not explicitly reveal the processing of the logic signals or the flow of the computation:

1. Behavioral
2. Data Flow
3. Structural

Q. How can this combinational circuit be described using programming language type constructs?
What is the purpose of describing a digital circuit or system in this manner?

Ans.

Purpose of Hardware Description Language (HDL)

- Ease of specifying the design of large circuits/systems using well-understood programming language and algorithmic constructs.

Note: Did you know that any algorithm that can be mapped to a program (in say C++) can also be mapped to hardware that then directly implements the algorithm? This is because a program is ultimately executed by a hardware—a computer, and a direct mapping of the algorithm executed by a hardware—a computer, and a direct mapping of the algorithm by mimicking the computer with signals from a controller.

By sourcing the relevant operands to the inputs of an adder via control signals, the process of ADD instruction within the specialized hardware will process a general operation to be performed by the computer will process it via a general execution (as a byproduct of the mapping, often called the area of the algorithm). If there is an ADD execution (as a byproduct of the mapping, for e.g., if there is an ADD execution by a hardware—a computer, and a direct mapping of the algorithm by mimicking the computer), then the mapping is ultimately implemented by the algorithm. This is because a program is ultimately executed by a hardware—a computer.
Hardware Synthesis – Example 1

Hardware Synthesis

Generation

Computer Code

Programming Language Statement

a := b + c;

ALU

Mux Mux Mux2 select Mux1 select

Mux Select

Control Signals

A D D E R

Reg r/w

Reg addr

Alu OPER select

Control Unit (FSM)

Control Signals

Register File

Read Bus A Read Bus B

Mux

Write Bus

Store R2 a

Load R2 b

Load R7 c

ADD R2 R5 R7

32 32

32

ADD ALU

r2

r7

r5

32

Execution in a CPU

Load R17 c

Load R15 b

rdw

rdw

rdw
Hardware Synthesis – Example 2

O/Ps of code block

I/Ps to code block

Synthesized

Recursively

Computer

Execution in

1 0

1 0

ldb lda

Mux1 select

zero

ovfl

sign

Demux1 select

Demux

Block B

for

Hardware

Mux

Block A

for

Hardware

Conrtol

Control signals

Unit (FSM)

.Then (P5M)

(end)

Adder

Conrtol signals

(Adder may be

used for other

operations)

2's compl

b a

Adder

Block C

Load r3 b

BZ ABNEG A

SUB r2 r2 r3

Assmb code

end

Block B of code;

else begin

end

Block A of code;

if (a <= b) then begin

Assmb code

else begin

End

Block, C

Assmb code

A: Block A

JMP C

Assmb code

B: Block B

BNEG A

BZ

SUB r2 r2 r3

Load r3 b

Load r2 a

Hardware Synthesis – Example 2
- Simulating and testing designs for correctness before actual implementation.
- Automatic synthesis of large designs specified behaviorally or algorithmically reduces human error.
- Increases design productivity.
- Chip layout using verification tools.
- Can also verify each transformation (circuit, netlist).
- Automatically synthesizes of large designs specified behaviorally or algorithmically.
- Fabrication.
- Testing and testing designs for correctness before actual implementation.

| Purpose of HDL (contd.) |
First the entity declarations specifying input/output signals:

```
entity ckt1 is
  port(
    s1 : in bit;
    s2 : in bit;
    Z : out bit);
end entity ckt1;
```

(1) Behavioral: Just specify the o/p function's conditions for being 1 and 0.

```
architecture behav1 of ckt1 is begin
  if s1 = '1' and s2 = '0' then
    Z = '1';
  elsif s1 = '1' and s2 = '0' then
    Z = '1';
  else
    Z = '0';
  end if;
end architecture behav1;
```

Describing Combinational Ckts and Data Paths (Contd.)
functions or modules/gates is specified. These functions are specified by the compiler/simulator when the delays of the individual atomic SOP/POS expressions. This is not the case with data flow and structural descriptions, where the delay of the overall function is automatically computed by the compiler/simulator when the delays of the individual atomic functions or modules/gates are known.

**Disadvantage:** Have to calculate the gate delays of an actual implementation.

**Ans:**

```vhdl
Z \rightarrow (\neg s_1 \land s_2) \lor (s_1 \land \neg s_2)
```

```vhdl
end architecture behav2
end architecture behav2
```

**OR:**

```vhdl
end process sw-controller
```

end if.
(2) Data Flow: As the name implies, the processing flow of the signals (i.e., data) is specified at some level (low level consisting of basic Boolean operations like AND, OR, NOT, NAND, NOR, etc.) or at a higher level (signal, etc., declarations). For the above e.g.:

```vhdl
architecture dataflow of ckt1 is
begin
  (signal, etc. declarations)
  not sbar1 after 1 ns;
  not sbar2 after 1 ns;
  sbar1 after 1 ns;
  sbar2 after 1 ns;
  x s1 and not sbar2 after 2 ns;
  y s2 and not sbar1 after 2 ns;
  Z x or y after 2 ns;
end architecture
```

NOT gate
AND gate
OR gate
S1
S2
Sbar2
Sbar1
Generally just like statements in a regular programming language, statements in a process are executed in order of their appearance as earlier. Concurrently, statements are also sequential statements in VHDL which are included in a process definition as in the behavioral description of the circuit given earlier. The above statements are called concurrent statements and are executed concurrently (i.e., simultaneously) by a simulator and not sequentially as in a regular programming language. The above statements are called concurrent statements where \( sbar_1, sbar_2, x, y \) are intermediate signals. Notice how the process-
Digression—Discrete Event Simulation (DES)– w/o Delays

An event is said to occur on a signal whenever its value changes. Also, the new values on signals that change after the evaluation of their

gue in Round 1.

and modules whose one or more input signals have been put in the event queue in Round 1. In Round 1 + 1 evaluation takes place of all statements.

When there are no delays specified with statements or modules, DES proceeds in rounds. In Round 1 evaluation takes place of all statements.

However, at the beginning each statement (in case of behavioral or data flow descriptions) and module (in case of structural descriptions) is evaluated. X is an input are evaluated.

When X is picked from the event queue (in the next round or at the right time in case of specified delays), then all statements or modules for which X is a signal change after evaluation, their input. If the output signal X of this statement changes after evaluation, a statement is simulated only when an event occurs on at least one of its inputs (henceforth referred to only as „statement“ in disc. of DES). All discussion of DES (w/ and w/o delays) apply only to concurrent state-

An event is said to occur on a signal whenever its value changes.
Simulations suspend until the values of some signals (generally, input)
change.
No new events are scheduled.

\[ I = Z, I = \lambda, 0 = x, 0 = z \]
\[ S = I, S = I, S = I, S = I \]

**Rand 3:**

\[ s = 0, s = 0, s = 0, s = 1 \]

The value of \( z \) is scheduled on \( Z \).

The value of \( I \) is scheduled on \( I \).

**Eval:** \( x \) is scheduled.

**Eval:** \( x \) is not scheduled.

No change in \( x \). So no event scheduled on \( x \).

\[ 0 = Z, I = \lambda, 0 = x, 0 = z \]

The value of \( s \) is scheduled.

The value of \( s \) is not scheduled.

\[ 0 = Z, 0 = \lambda, 0 = x \]

**Rand 1:**

Nonew events are scheduled.

Simulations suspend until the value of some signal (generally, input)
changes.

**Start:**

\[ s = 1, s = 1, s = 1, s = 1 \]

Events w/o Delays
Instead of rounds, we now have the concept of simulation time.

- The current simulation time is the time tag of the statement or module picked from the event queue to be executed.

- Statement/module evaluation takes place at a certain time \( t \) after all signals to be displayed from the beginning of the event queue.

- Statement/module evaluation takes place after all signals are displayed and their new values are

- If a delay of, say, 10 ns is associated with an output signal \( X \), when its corresponding statement is executed, its value changes, then \( X \) is put in the event queue with a time tag of \( t + 10 \) ns, where \( t \) is the current simulation time.

- The current simulation time is the time which the DES has performed simulations of the various statements or modules of the circuit.

- Discrete Event Simulation - With Delays

- In addition we have the following concepts:

- The initial simulation time is the circuit time \( t \) which the DES has

- The first three definitions and issues (about event, when simulation is

---
Discrete Event Simulation - E.G. With Delays

Event: Event

\[ x \Rightarrow z \]

\[ s1 \Rightarrow s1 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]

\[ s2 \Rightarrow s2 \]
(3) Structural:

Back to Description of Combinational Ckts and Data Paths
end architecture behav;
    \textit{or} is an inbuilt operator of VHDL.
    \[ c \text{ or } q \]
begin
architecture behav of ckt1 is
\textit{E.g.} architecture definition of orgate:
end entity orgate;
port (a, q: in bit; c: out bit),
    \textit{entity orgate is}
end entity orgate;
port (a, q: in bit; c: out bit),
    \textit{entity andgate is}
end entity andgate;
port (a, q: in bit; c: out bit),
    \textit{entity notgate is}
end entity notgate;
port (a: in bit; q: out bit),
    \textit{entity notgate is}
end entity notgate;

\hspace{1cm} \text{in the \textit{entity definition} and the \textit{architecture} of the module:}
Then specify the interconnections of these modules to represent the circuit.

(b) Module interconnection from VHDL structural description:

(circuit:

\[
\begin{align*}
& \text{NOT gate} \\
& \text{OR gate} \\
& \text{AND gate}
\end{align*}
\]

(a) Circuit details:

\[
\begin{align*}
& \text{AND gate} \\
& \text{NOT gate} \\
& \text{AND gate}
\end{align*}
\]

Then specify the interconnections of these modules to represent the circuit.

(b) Module interconnection from VHDL structural description:

(circuit:

\[
\begin{align*}
& \text{NOT gate} \\
& \text{OR gate} \\
& \text{AND gate}
\end{align*}
\]

(a) Circuit details:

\[
\begin{align*}
& \text{AND gate} \\
& \text{NOT gate} \\
& \text{AND gate}
\end{align*}
\]
The above type of component instantiation is available only in VHDL-90 and is called direct instantiation. This will not work in VHDL-87. Will show instantiation mechanism later for VHDL-87.

Note that multiple architecture definitions are possible in VHDL for the same circuit/system.

begin
architecture structural
port map (x, y, z);
end architecture structural;

port map (s2, sbar);
and2: entity work.andgate(behav)
and1: entity work.orgate(behav);
port map (x, y, z);
end architecture structural;
Switch-Based Logic Circuits

Switch is a "mechanism" (mechanical, electrical, electro-mechanical) that, given some external stimulus will either make or break an electrical connection.

**NORMALLY OPEN (NOP) SWITCH:**

\[
\begin{array}{c|c|c}
A=0 & Y & X \\
A=1 & X & Y \\
\end{array}
\]

**NORMALLY CLOSED (NOC) SWITCH:**

\[
\begin{array}{c|c|c}
A=0 & Y & X \\
A=1 & X & Y \\
\end{array}
\]
Implementing NOT, AND, OR functions:

Implementing OR Terms (Parallel Connections):

Implementing Product Terms (Series Connections):

Implementing A:

Implementing NOT, AND, OR functions:
The correct implementation is on the right and is called a *complement* switching network.

\[ Z = S_1S_2 + S_1S_2 \]

Implementing \( Z \)
**Transistor Implementation of Switches**

**NORMALLY OPEN (NOP) SWITCH:**
- Conduction when \( A = 1 \)
- Open when \( A = 0 \)

**NORMALLY CLOSED (NOC) SWITCH:**
- Conduction when \( A = 0 \)
- Open when \( A = 1 \)
Problem with Large Switching Networks

nMOS transistors do not conduct a "good" 1; pMOS transistors do not conduct a "good" 0. Thus in long series paths with many pMOS transistors, a "good" 1 conduct a "good" 0.

Thus need to use blocks of small switching networks called gates to implement logic circuits. Examples of typical gates are AND, OR, NOT, NAND, NOR, EXOR and XNOR.

Similarly long series paths with many nMOS transistors, a "good" 0 can appear at the output. Thus in long series paths with many nMOS transistors, a "good" 1; pMOS transistors do not conduct a "good" 0.
Transistor Implementation of Gates

Due to the above problems of good 1 and 0 conduction in CMOS technology, NAND and NOR gates are preferred building blocks instead of AND and OR gates. This implementation will not conduct good 1s and 0s.

The NOP and NOC switches can also be described behaviorally in an HDL (more "values" besides '1' and '0' are now needed for signals, e.g., 'Z' in VHDL for type std_logic, "holding/high-impedance" is needed – 'Z' in VHDL for type std_logic)

Gates can then be described structurally using such switch modules ac-

(standard logic)

Illustration for preference of NAND over AND (NOR over OR) stems from a similar reason):