ECE 368: CAD Based Logic Design

Lecture Notes # 1

Introduction: Combinational Logic and Its Description Using Hardware Description Languages (HDLs)

Shantanu Dutt

Department of Electrical and Computer Engineering
University of Illinois, Chicago
Phone: (312) 355-1314; e-mail: dutt@ece.uic.edu
URL: http://www.ece.uic.edu/~dutt
Besides, computers, logic circuits find the bulk of their application as the embedded systems or brains of an enormous variety of products. In such systems they are generally used as controllers of a larger (non-logic—e.g., analog, electro-mechanical) system.
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
Application of Logic Circuits (contd.)

- 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 loopings in these constructs. E.g.,

  Repeat
  If input condition $C_1$ holds then do output action $A_1$
  else if input condition $C_2$ holds then do output action $A_2$
  ...
  Until condition ($C_0$)

- 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)
• Conditions $C_i$ 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 the furnace on
  – Tape the program on channel 7
  – Turn on the light
The Role of Logic–An Introduction

• Digital system design is based on the principle of logic and the manipulation of logic symbols

• Digital system design is thus also termed logic design

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

• Example: Statement A: Today is a nice day \( \equiv F \)
  Statement B: This is a class of bright students \( \equiv T \)

• 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: \( A \ AND \ B \equiv (\text{Today is a nice day}) \ AND \ (\text{This is a class of bright students}) \equiv F \)

• Similarly, \( A \ OR \ B \equiv T; \ NOT(B) \equiv F; \ A \ OR \ (NOT(B)) \equiv F \)

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

- 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 \( f \).
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A</th>
<th>B</th>
<th>A OR B</th>
<th>A</th>
<th>NOT(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- More Examples: A generic 2-input function

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>f(A₁, A₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- There are a total of \(2^2\) combinations of 2 inputs (i/p)–so the TT has 4 rows
- Each output (o/p) of a row can have 2 possible values (1/0)–so there are \(2^4 = 16\) possible combinations of values for a 4-row TT
- Each of these 16 combinations is a distinct TT–thus there are 16 different 2-i/p functions
A generic 3-i/p function:

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$f(A_1, A_2, A_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- There are $2^3 = 8$ input combinations
  - thus 8 rows
- Thus a 3-i/p TT can have $2^8 = 256$
  possible output combinations in its
  8 rows—thus 256 different 3-i/p func-
  tions
- In general an $n$-i/p function/TT has
  $2^n$ input combinations
- It thus has $2^{2n}$ output combination,
  one for each distinct function—thus
  there are $2^{2n}$ different $n$-i/p functions
A Simple Design Problem

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 Design Problem (Contd.)

Design Steps:

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

**Input Encoding:**

Switch $s_1$ is pushed $\equiv$ logic input $s_1 = 1$
Switch $s_1$ is not pushed $\equiv$ logic input $s_1 = 0$
Similarly for switch $s_2$ and logic input variable $s_2$
(In this case we are using the same name for both the input sensor and the corresponding single input variable–this may not always be the case)

**Imp. Note:** The term $s_1 = 1$ can be more simply represented as the logic expression $s_1$; the term $s_1 = 0$ can be more simply represented as $NOT(s_1)$ or $\bar{s}_1$

**Output Encoding:**

Turn the (electro-mechanical) switch SW on $\equiv$ logic output $Z = 1$
Turn the (electro-mechanical) switch SW off $\equiv$ logic output $Z = 0$
2. Get TTs for each output variable (i.e., for each logic function required) after analyzing the requirement and from the TT obtain its logic expression

NOTE: This TT-based approach should really should be and can be done 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, and/or high-level and/or algorithmic manner. Will later see these latter approaches for the design of arithmetic circuits.

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; both lead to correct designs in terms of the given specification).
Each row in a TT corresponds to an *AND term* or *product term* or *minterm* of the input variables.

- In the product term an i/p variable $X$ occurs as $\bar{X}$ if it is 0 in that row otherwise as $X$.

- The output column is the OR of all the product terms (rows) for which it is a 1—thus $Z = (\bar{s}_1 \text{ AND } s_2) \text{ OR } (s_1 \text{ AND } \bar{s}_2)$.

- More compactly, $Z = (s_1s_2) + (s_1\bar{s}_2)$, product form replaces AND and plus (+) form replaces OR. This is called a *canonical* sum-of-product (SOP) expression.
3. Minimize the logic expression(s) using various methods like algebraic manipulation, K-Maps, Quine-McCluskey and Petrick’s algorithm. The above expression 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)
Purpose of Hardware Description Language (HDL)

• Q. What is the purpose of describing a digital circuit or system in this manner?

   Ans.  
   – 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 to hardware executes that algorithm by mimicking the computer execution (as a byproduct of the mapping, for e.g., id there is an ADD operation to be performed the computer will process it via a general purpose ADD instruction while the specialised hardware will process it by sourcing the relevant operands to the inputs of an adder via control signals from a controller).
Hardware Synthesis – Example 1

Programming language statement

\[ a := b + c; \]

Computer Code Generation

Hardware Synthesis

Execution in a CPU

Programming language statement

\[ a := b + c; \]

Computer Code Generation

Hardware Synthesis

Execution in a CPU

Programming language statement

\[ a := b + c; \]

Computer Code Generation

Hardware Synthesis

Execution in a CPU
Hardware Synthesis – Example 2

Programming language construct

if (a <= b) then begin
    Block A of code;
end
else begin
    Block B of code;
end

Computer Code Generation

Load r2 a
Load r3 b
SUB r2 r2 r3
BZ A
BNEG A
B: Block B
    assmb code
    JMP C
A: Block A
    assmb code
    JMP C
C: Block C
    assmb code

Execution in a computer

Computer

O/Ps of code block

Hardware Synthesis

I/Ps to code block

Hardware for Block B

Hardware for Block A

Recursively Synthesized

Control Unit (FSM)

Control signals

Demux

Demux1_select

0 1

Mux

Mux1_select

0 1

O/Ps of code block

ldblda

Mux1_select

2's compl

sign

overflow

zero

add

Sub
Purpose of HDL (contd.)

– Simulating and testing designs for correctness before actual implementation/fabrication.

– Automatic synthesis of large designs specified behaviorally or algorithmically using CAD tools:
  • can also verify each transformation (algorithm → circuit (netlist) → chip layout) using verification tools
  • reduces human error
  • increases design productivity
  • can lead to more innovative designs as it frees the designer from well-known nitty-gritty details of circuit design (e.g., design of a fast adder) allowing the designer to explore alternative designs/algorithms at a higher level
Q. How can this combinational circuit be described using programming language type constructs?

Ans. Three ways ranging from the abstract level which has no implementation descriptions to the implementation level that specifies the interconnections (wiring) 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
First the **entity** description specifying input/output signals:

```
entity ckt1 is
  port(s1, s2: in bit; Z: out bit);
  -- i/o signals of type **bit** can only take on ‘0’ and ‘1’ values
end entity ckt1;
```

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

```
architecture behav1 of ckt1 is begin
  sw-controller: process is
    begin
      if s1 = 0 and s2 = 1 then Z <= 1; ✋
      elsif s1 = 1 and s2 = 0 then Z <= 1;
      else Z <= 0;
    end process;
end
```
end if;
end process sw-controller;
end architecture behav1;

OR specify its expression directly if known:
architecture behav2 of ckt1 is begin
Z <= (not(s1) and s2) or (s1 and not(s2));
end architecture behav2

What is missing from this description with respect to an actual circuit? Ans. **Delay**; so this attribute needs to be present in a *hardware description language (HDL).*

E.g., $Z <= (\text{not}(s_1) \text{ and } s_2) \text{ or } (s_1 \text{ and } \text{not}(s_2))$ after 5ns;

Disadvantage: Have to calculate the gate delays of an actual implementation to specify the delay in a behavioral description of a non-trivial SOP/POS expression. 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 is specified.
(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 where the basic operations are arithmetic operations like $+,-,\times,\div$, etc.). For the above e.g.:

```vhdl
architecture dataflow of ckt1 is
  (signal, etc. declarations)
begin
  sbar_1 <= not(s_1); (or sbar_1 <= not(s_1) after 1ns);
  sbar_2 <= not(s_2); (or sbar_2 <= not(s_2) after 1ns);
  x <= s_1 and sbar_2; (or x <= s_1 and sbar_2 after 2ns);
  y <= s_2 and sbar_1; (or y <= s_2 and sbar_1 after 2ns);
  Z <= x or y; (or Z <= x or y after 2ns;)
end architecture dataflow
```
where \( sbar_1, sbar_2, x, y \) are intermediate signals. Notice how the processing of the function \( Z \) is described in detail via all intermediate signals.

- The above statements are called *concurrent statements* are to be executed *concurrently* (i.e., simulated concurrency) by a simulator and not sequentially as in a regular programming language.

- Concurrent statements represent hardware perfectly since the hardware is always “executing” each statement concurrently!

- A statement is executed only when one of the signals, say, \( x \) on its left hand side (LHS) changes—this is called an *event* on \( x \) and such a simulation of a circuit is called a *discrete event simulation*.

- There are also *sequential statements* in VHDL which are included in a *process* definition as in the behavioral description of the circuit given earlier.

- Statements in a process are executed in order of their appearance sequentially just like statements in a regular programming language.
Digression—Discrete Event Simulation (DES) – w/o Delays

- An event is said to occur on a signal whenever its value changes.
- All discussion of DES (w/ and w/o delays) apply only to concurrent statements (henceforth referred to only as “statement” in disc. of DES)
- A statement is simulated only when an event occurs on at least one of its inputs. If the output signal $X$ of this statement changes after evaluation, then $X$ is put in in event queue.
- 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 an input are evaluated.
- For initialization, each signal is just initialized to its "leftmost" value (e.g., for "bit" type this is '0').
- Evaluation once to determine the initial value of each signal and variable.
- When there are no delays specified with statements or modules, DES proceeds in rounds. In round $i + 1$ evaluation takes place of all statements and modules whose one or more input signals have been put in the event queue in round $i$.
- Also, the new values on signals that change after the evaluation of their assignment statements or driving modules in round $i$, are “displayed” in round $i+1$.
Discrete Event Simulation – E.g. w/o Delays

\[ \text{sbar}_1 \leq \text{not}(s_1); \quad \text{sbar}_2 \leq \text{not}(s_2); \]
\[ x \leq s_1 \text{ and } \text{sbar}_2; \quad y \leq s_2 \text{ and } \text{sbar}_1; \]
\[ Z \leq x \text{ or } y; \]

- **Start:** \( s_1 = 0, s_2 = 0, \text{sbar}_1 = 1, \text{sbar}_2 = 1, x = 0, y = 0, Z = 0 \)
- **Rnd 1 (I/P signal } s_2 \text{ changes):** \( s_1 = 0, s_2 = 1, \text{sbar}_1 = 1, \text{sbar}_2 = 1, x = 0, y = 0, Z = 0 \)
  - Eval \( \text{sbar}_2 \leq \text{not}(s_2); \) The value ‘0’ is scheduled on \( \text{sbar}_2. \)
  - Eval \( y \leq s_2 \text{ and } \text{sbar}_1; \) The value ‘1’ is scheduled on \( y. \)
- **Rnd 2:** \( s_1 = 0, s_2 = 1, \text{sbar}_1 = 1, \text{sbar}_2 = 0, x = 0, y = 1, Z = 0 \)
  - Eval \( x \leq s_1 \text{ and } \text{sbar}_2; \) No change in \( x, \) so no event scheduled on \( x \)
  - Eval \( Z \leq x \text{ or } y; \) The value ‘1’ is scheduled on \( Z. \)
- **Rnd 3:** \( s_1 = 0, s_2 = 1, \text{sbar}_1 = 1, \text{sbar}_2 = 0, x = 0, y = 1, Z = 1 \)
  - No new events are scheduled.
  - Simulation suspends until the value of some signal (generally, input) changes.
Discrete Event Simulation – With Delays

– The first three definitions and issues (about event, when simulation is performed of statements and modules, and the initial evaluations) also apply in DES with delays. In addition we have the following concepts.

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

– If a delay of, say, 10 ns, is associated with an output signal $X$ and 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 event queue is kept sorted by increasing time tags and the DES picks signals to be displayed from the beginning of the event queue.

– Statement/module evaluation takes place at a certain time $t$ after all signals scheduled for display at time $t$ are displayed and their new values asserted. Only those statements/modules are evaluated which have at least one input from these displayed signals.

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

– Instead of rounds, we now have the concept of simulation time.
Discrete Event Simulation — Event Queue for Simulation w/ Delays

Statements:  
\[ sbar_1 <= not(s_1) \text{ after } 1 \text{ ns}; \]  
\[ sbar_2 <= not(s_2) \text{ after } 1 \text{ ns}; \]  
\[ x <= s_1 \text{ and } sbar_2 \text{ after } 2 \text{ ns}; \]  
\[ y <= s_2 \text{ and } sbar_1 \text{ after } 2 \text{ ns}; \]  
\[ Z <= x \text{ or } y \text{ after } 2 \text{ ns}; \]

DES with delays

At time t:  
\[ S1: 0 \rightarrow 1, \text{ S2}=0, \text{ Sbar1}=1, \text{ Sbar2}=1, \text{ x}=0, \text{ y}=0, \text{ Z}=0 \]

At time t: display \[ S1 = 1 \] & execute \[ sbar1 <= not(s1) \text{ after } 1\text{ ns} \];  
\[ x <= s1 \text{ and } sbar2 \text{ after } 2\text{ ns} \];  
Result $\rightarrow$ 2 events: \[ sbar1: 1 \rightarrow 0 \text{ @ t+1 ns}, \text{ x}: 0 \rightarrow 1 \text{ @ t+2 ns} \]

At time t+1: disp \[ sbar1=0 \] & execute \[ y <= s2 \text{ and } sbar1 \text{ after } 2\text{ ns} \];  
Result $\rightarrow$ 0 events: \[ y: 0 \rightarrow 0 \];

At time t+2: disp \[ x=1 \] & execute \[ Z <= x \text{ or } y \text{ after } 2\text{ ns} \];  
Result $\rightarrow$ 1 event: \[ Z: 0 \rightarrow 1 \text{ @ t+4 ns} \]

At time t+4: disp \[ Z=1 \], no statement execution
Discrete Event Simulation – E.g. With Delays

\( sbar_1 \leq \) not\((s_1)\) after 1ns; \( sbar_2 \leq \) not\((s_2)\) after 1ns;
\( x \leq s_1 \) and \( sbar_2 \) after 2ns; \( y \leq s_2 \) and \( sbar_1 \) after 2ns;
\( Z \leq x \) or \( y \) after 2ns;

\[
\begin{align*}
\text{Eval } Sbar2 & \leq \text{not}(S2) \text{ at } t=1ns \\
\text{Eval } x & \leq S1 \text{ and } Sbar2 \text{ at } t=2ns \text{ (no change in } x) \\
\text{Eval } y & \leq S2 \text{ and } Sbar1 \text{ at } t=1ns \\
\text{Eval } Z & \leq x \text{ or } y \text{ at } t=3ns
\end{align*}
\]
Discrete Event Simulation — Event Queue for Simulation w/o Delays

- Statements: \( sbar_1 \leq \text{not}(s_1); sbar_2 \leq \text{not}(s_2); \)
  \( x \leq s_1 \text{ and } sbar_2; y \leq s_2 \text{ and } sbar_1; Z \leq x \text{ or } y; \)

- In actuality, statements with delays are simulated in a similar way as statements with delays by assigning a “fictitious” \( \Delta \) delay to a statement w/o delays.

- So a statement \( x \leq s_1 \text{ and } sbar_2; \) becomes equivalent to \( x \leq s_1 \text{ and } sbar_2 \text{ after } \Delta; \)

- The following rules apply to such statements: (a) \( i\Delta > j\Delta \) if \( i > j. \) (b) \( i\Delta < \) any specified “real” delay however small (e.g., 1 fs).

- These rules and the concept of fictitious \( \Delta \) delays allows treatment of statements with and without delay specifications to be treated in an uniform manner, as illustrated below:

DES without delays

At time \( t: S1: 0 \rightarrow 1, S2=0, Sbar1=1, Sbar2=1, x=0, y=0, Z=0 \)

Event New value Event Q

Available @ time

At time \( t: \text{display } S1 = 1 \) & execute \( sbar1 \leq \text{not}(s1); x \leq s_1 \text{ and } sbar2; \) ~ Round 1
Result -> 2 events: \( sbar1: 1 \rightarrow 0 @ t+ \text{del}, \ x: 0 \rightarrow 1 @ t+\text{del}. \)

At time \( t+\text{del}: \text{disp } sbar1=0, \ x=1, \ & \text{execute: } y \leq s2 \text{ and } sbar1; \ Z \leq x \text{ or } y \text{ after } 2\text{ns}; \) ~ Round 2
Result -> 1 event: \( Z: 0 \rightarrow 1 @ t+ 2*\text{del}; \)
(3) Structural:

- Specify the interconnection of the modules/gates. The modules/gates themselves need to be described as separate entities in either a behavioral, data-flow or structural manner and so forth.
- A structural (or mixed) description also allows us to specify the design in a top-down hierarchical manner.
- However, the algorithm, data flow, etc. of the processing is not clearly evident in such a description.
- Such a description is mainly used for simulation and testing to verify the correctness of a circuit level design.

For the above e.g.:

- First declare the input/output ports or signals of each module (enclosed
in the **entity** definition) and the **architecture** of the module:

- **entity** notgate is
  
  ```vhdl
  port (a: in bit; b: out bit);
  end entity notgate;
  ```

- **entity** andgate is
  
  ```vhdl
  port (a, b: in bit; c: out bit);
  end entity andgate;
  ```

- **entity** orgate is
  
  ```vhdl
  port (a, b: in bit; c: out bit);
  end entity orgate;
  ```

- E.g. architecture definition of orgate:

  ```vhdl
  architecture behav of orgate is
  begin
  c <= a or b after 2 ns;
  -- “or” is an inbuild operator of VHDL
  end architecture behav;
  ```
Then specify the interconnections of these modules to represent the circuit:

```
architecture structural of ckt1 is

signal sbar_1, sbar_2, x, y: bit;

begin
    not1: entity work.notgate (behav)
    port map (s1, sbar_1);
    not2: entity work.notgate (behav)
    port map (s2, sbar_2);
    and1: entity work.andgate (behav)
    port map (s1, sbar_2, x);
```
and2: entity work.andgate(behav)
port map (s2, sbar1, y);
or1: entity work.orgate(behav)
port map (x, y, z);
end architecture structural;

• 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.
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

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

architecture structure of ckt1 is
component and_gate is
  port(x,y:in bit;
       z:out bit);
end component and_gate;

component not_gate is
  port(x:in bit;
       y:out bit);
end component not_gate;

component or_gate is
  port(x,y:in bit;
       z:out bit);
end component or_gate;
signal sbar1,sbar2,X,Y:bit;
beg

N1:not_gate port map(s1,sbar1);
N2:not_gate port map(s2,sbar2);
and1:and_gate port map(s1,sbar2,X);
and2:and_gate port map(s2,sbar1,Y);
or1:or_gate port map(X,Y,Z);
end architecture structure;
Summary

Discussed the following:

- Uses of HDL

- *Entity* (generic parameters, i/o signals), *architecture* (functional and/or mechanism descriptions) and *binding* (which entity-architecture combinations to use for the instantiated components (if any)—configuration description, direct instantiation, etc.) descriptions of a VHDL module

- Three types of architecture descriptions: *behavioral, dataflow, structural*. Behavioral descriptions can be done at many levels. A *mixed* description incorporates elements of 2 or more of the above types

- *Concurrent* statements, *sequential* statements & *processes*—basic concepts

- *Discrete event simulation* of statements w/ and w/o delays
  - Concepts of *events, event Q, simulation time*
  - Concept of $\Delta$-delays to tackle statements w/o delays