



Chapter 2.13-2.15

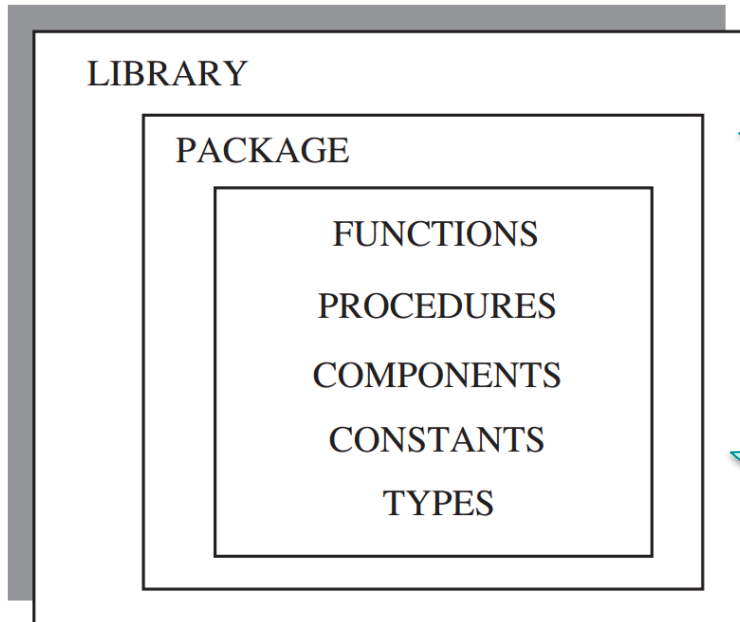
Introduction to VHDL

Version: 2023/11/28

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.13 VHDL Libraries



A **library** is a collection of commonly used pieces of code

Libraries extend the functionality of VHDL by defining types, functions, components, and **overloaded** operators

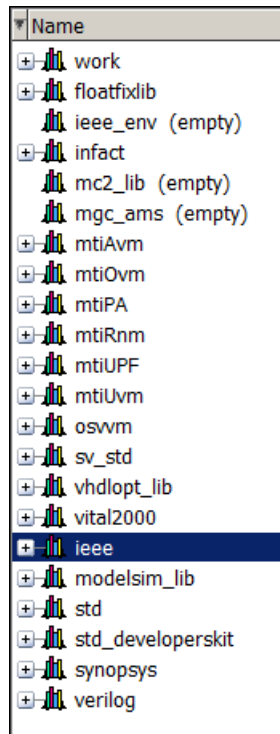
Overload (重载)

It means that two or more functions may have the **same name**, so long as the parameter types are sufficiently different enough to distinguish which function is actually intended

2.13 VHDL Libraries

```
LIBRARY ieee;  
LIBRARY std;  
LIBRARY work;
```

At least three libraries and their packages are usually needed in a design



work is where we save our design

- **std** and **work** are made visible by default
- There is no need to declare them

2.13 VHDL Libraries

Library **ieee;**

Name	Type	Path
work	Library	work
floatfixlib	Library	\$MODEL_TECH/./floatfixlib
ieee_env (empty)	Library	\$MODEL_TECH/./ieee_env
infact	Library	\$MODEL_TECH/./infact
mc2_lib (empty)	Library	\$MODEL_TECH/./mc2_lib
mgc_ams (empty)	Library	\$MODEL_TECH/./mgc_ams
mtiAvm	Library	\$MODEL_TECH/./avm
mtiOvm	Library	\$MODEL_TECH/./ovm-2.1.2
mtiPA	Library	\$MODEL_TECH/./pa_lib
mtiRnm	Library	\$MODEL_TECH/./rnm
mtiUPF	Library	\$MODEL_TECH/./upf_lib
mtiUvm	Library	\$MODEL_TECH/./uvm-1.1d
osvwm	Library	\$MODEL_TECH/./osvwm
sv_std	Library	\$MODEL_TECH/./sv_std
vhdlopt_lib	Library	\$MODEL_TECH/./vhdlopt_lib
vital2000	Library	\$MODEL_TECH/./vital2000
ieee	Library	\$MODEL_TECH/./ieee
modelsim_lib	Library	\$MODEL_TECH/./modelsim_lib
std	Library	\$MODEL_TECH/./std
std_developerskit	Library	\$MODEL_TECH/./std_developerskit
synopsys	Library	\$MODEL_TECH/./synopsys
verilog	Library	\$MODEL_TECH/./verilog

- In the initial days of CAD, every tool vendor used to create its own libraries and packages
- Porting designs from one environment to another became a problem

2.13 VHDL Libraries

Library **ieee;**

Name	Type	Path
vital2000	Library	\$MODEL_TECH/./vital2000
ieee	Library	\$MODEL_TECH/./ieee
fixed_float_types	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_float_types.vhdl
fixed_generic_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_generic_pkg.vhdl
fixed_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_pkg.vhdl
float_generic_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/float_generic_pkg.vhdl
float_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/float_pkg.vhdl
ieee_bit_context	VHDL Con...	\$MODEL_TECH/./vhdl_src/ieee/ieee_bit_context.vhd
ieee_std_context	VHDL Con...	\$MODEL_TECH/./vhdl_src/ieee/ieee_std_context.vhd
math_complex	Package	\$MODEL_TECH/./vhdl_src/ieee/1076-2code.vhd
MATH_COMPLEX_...	Package	D:/qa/buildsites/10.4a/builds/win32pe_edu/modeltech/vhdl_sr...
math_real	Package	\$MODEL_TECH/./vhdl_src/ieee/1076-2code.vhd
MATH_REAL_mti...	Package	D:/qa/buildsites/10.4a/builds/win32pe_edu/modeltech/vhdl_sr...
numeric_bit	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_bit.vhd
numeric_bit_unsig...	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_bit_unsigned.vhd
numeric_std	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_std.vhd
numeric_std_unsig...	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_std_unsigned.vhd
std_logic_1164	Package	\$MODEL_TECH/./vhdl_src/ieee/stdlogic.vhd
std_logic_arith	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_arith.vhd
std_logic_misc	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_misc.vhd
std_logic_signed	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_signed.vhd
std_logic_textio	Package	\$MODEL_TECH/./vhdl_src/synopsys/std_logic_textio.vhd
std_logic_unsigned...	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_unsigned.vhd
upf	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_upf.vhd
vital_memory	Package	\$MODEL_TECH/./vhdl_src/vital2000/memory_p_2000.vhd
vital_primitives	Package	\$MODEL_TECH/./vhdl_src/vital2000/prmtvs_p_2000.vhd
vital_timing	Package	\$MODEL_TECH/./vhdl_src/vital2000/timing_p_2000.vhd
modelsim_lib	Library	\$MODEL_TECH/./modelsim_lib
std	Library	\$MODEL_TECH/./std
std_deviancelib	Library	\$MODEL_TECH/./std_deviancelib

IEEE: Institute of Electrical and Electronic Engineers 电气电子工程师学会

IEEE has developed **standard libraries** and packages to make design protability easier

2.13 VHDL Libraries

Library **ieee;**

Name	Type	Path
vital2000	Library	\$MODEL_TECH/./vital2000
ieee	Library	\$MODEL_TECH/./ieee
fixed_float_types	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_float_types.vhdl
fixed_generic_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_generic_pkg.vhdl
fixed_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/fixed_pkg.vhdl
float_generic_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/float_generic_pkg.vhdl
float_pkg	Package	\$MODEL_TECH/./vhdl_src/ieee/float_pkg.vhdl
ieee_bit_context	VHDL Con...	\$MODEL_TECH/./vhdl_src/ieee/ieee_bit_context.vhd
ieee_std_context	VHDL Con...	\$MODEL_TECH/./vhdl_src/ieee/ieee_std_context.vhd
math_complex	Package	\$MODEL_TECH/./vhdl_src/ieee/1076-2code.vhd
MATH_COMPLEX_...	Package	D:/qa/buildsites/10.4a/builds/win32pe_edu/modeltech/vhdl_sr...
math_real	Package	\$MODEL_TECH/./vhdl_src/ieee/1076-2code.vhd
MATH_REAL_mti...	Package	D:/qa/buildsites/10.4a/builds/win32pe_edu/modeltech/vhdl_sr...
numeric_bit	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_bit.vhd
numeric_bit_unsig...	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_bit_unsigned.vhd
numeric_std	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_std.vhd
numeric_std_unsig...	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_numeric_std_unsigned.vhd
std_logic_1164	Package	\$MODEL_TECH/./vhdl_src/ieee/stdlogic.vhd
std_logic_arith	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_arith.vhd
std_logic_misc	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_misc.vhd
std_logic_signed	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_signed.vhd
std_logic_textio	Package	\$MODEL_TECH/./vhdl_src/synopsys/std_logic_textio.vhd
std_logic_unsigned...	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_unsigned.vhd
upf	Package	\$MODEL_TECH/./vhdl_src/ieee/mti_upf.vhd
vital_memory	Package	\$MODEL_TECH/./vhdl_src/vital2000/memory_p_2000.vhd
vital_primitives	Package	\$MODEL_TECH/./vhdl_src/vital2000/prmtvs_p_2000.vhd
vital_timing	Package	\$MODEL_TECH/./vhdl_src/vital2000/timing_p_2000.vhd
modelsim_lib	Library	\$MODEL_TECH/./modelsim_lib
std	Library	\$MODEL_TECH/./std
std_devianarbit	Library	\$MODEL_TECH/./std_devianarbit

std_logic_1164




➤ **1164** is the IEEE standard number

Original VHDL standard only defined **2-valued** logic (bits, bit-vectors)

IEEE.std_logic_1164 defines a **std_logic** type that has **multivalued** logic

2.13 VHDL Libraries

IEEE standard packages

 numeric_std_unsigned... Package		\$MODEL_TECH/./vhdl_src/ieee/numeric_std_unsigned.vhd
 std_logic_1164	Package	\$MODEL_TECH/./vhdl_src/ieee/stdlogic.vhd
 std_logic_arith	Package	\$MODEL_TECH/./vhdl_src/synopsys/mti_std_logic_arith.vhd

`IEEE.std_logic_1164`

`type std_logic`

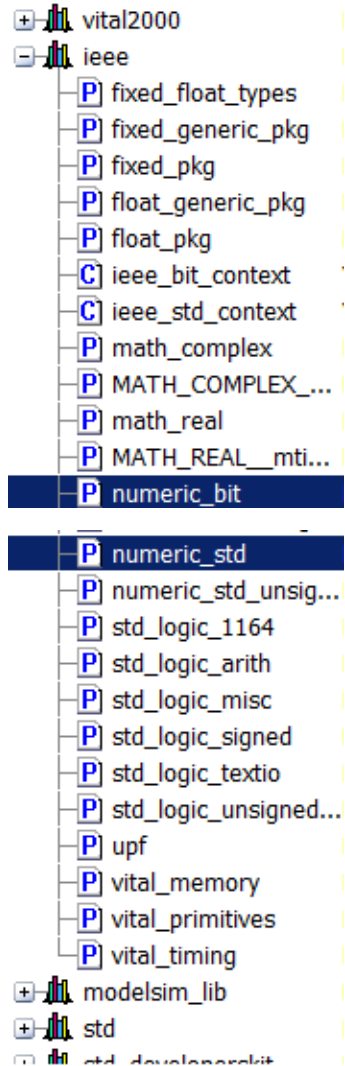
std_logic has nine values, including '0', '1', 'X'(unknown), and 'Z' (high impedance)

`type std_logic_vectors`

- The package also defines **logic** operations and other functions for working with **std_logic** and **std_logic_vectors**
- It does **not** provide for **arithmetic** operations

std_logic_1164 and its use for simulation and synthesis will be described in more detail

2.13 VHDL Libraries



Other packages are introduced to facilitate writing synthesizable code

Package **numeric_bit** uses **bit_vectors** to represent unsigned and signed binary numbers

Package **numeric_std** uses **std_logic_vectors** to represent unsigned and signed binary numbers

Use a **library statement** and a **use statement** to access functions and components from a library

DL Libraries

```
library ieee;  
use ieee.numeric_bit.all;
```

Whenever a package is used in a module, the library and use statements must be placed before the entity in the module period

```
entity Adder4 is  
    port(A, B: in unsigned(3 downto 0); Ci: in bit;      -- Inputs  
          S: out unsigned(3 downto 0); Co: out bit);    -- Outputs  
end Adder4;  
  
architecture overload of Adder4 is  
    signal Sum5: unsigned(4 downto 0);  
begin  
    Sum5 <= '0' & A + B + unsigned'(0=>Ci) ;           -- adder  
    S    <= Sum5(3 downto 0);  
    Co   <= Sum5(4);  
end overload;
```

- The scope of the library declaration statement extends from the declaration of an entity to the structure and configuration it belongs to, or from the declaration of a package to the end of its definition
- When there are multiple entities in a source code, or a package definition coexists with an entity, the library declaration statement needs to be **repeated** before each entity declaration and package declaration

2.13 VHDL Libraries

numeric_bit

```
type unsigned is array (natural range <>) of bit;  
type signed is array (natural range <>) of bit;
```

`numeric_bit` defines `unsigned` and `signed` types as unconstrained array of `bits`

Signed numbers are represented in `2's complement` form

It also contains `overloaded` for arithmetic, relational, logical, and shifting operations on unsigned and signed numbers

2.13 VHDL Libraries

```
library ieee;
use ieee.numeric_bit.all;

entity test is
    port( A, B : in  bit_vector(3 downto 0);
          C      : out bit_vector(3 downto 0));
end test;

architecture equ of test is
begin
    C <= A + B;
end;
```

compiler error if A, B, and C are **bit_vectors**

```
vcom -reportprogress 300 -work work E:/Kuaipan/Teaching/FPGA/Code/Chp2/chp2_13/test.vhd
# Model Technology ModelSim ALTERA vcom 6.5e Compiler 2010.02 Feb 27 2010
# -- Loading package standard
# -- Loading package numeric_bit
# -- Compiling entity test
# -- Compiling architecture equ of test
# ** Error: E:/Kuaipan/Teaching/FPGA/Code/Chp2/chp2_13/test.vhd(11): No feasible entries for infix operator "+".
# ** Error: E:/Kuaipan/Teaching/FPGA/Code/Chp2/chp2_13/test.vhd(11): Type error resolving infix expression "+" as type std.standard.bit_vector.
# ** Error: E:/Kuaipan/Teaching/FPGA/Code/Chp2/chp2_13/test.vhd(12): VHDL Compiler exiting
```

ModelSim>

error

2.13 VHDL Libraries

```
library ieee;
use ieee.numeric_bit.all;

entity test is
    port( A, B : in  unsigned(3 downto 0);
          C      : out unsigned(3 downto 0));
end test;

architecture equ of test is
begin
    C <= A + B;
end;
```

If A, B, C are of type **unsigned** or **signed**, the compiler will invoke the appropriate overloaded operator to carry out the addition

```
vcom -reportprogress 300 -work work E:/Kuaipan/Teaching/FPGA/Code/Chp2/chp2_13/test.vhd
# Model Technology ModelSim ALTERA vcom 6.5e Compiler 2010.02 Feb 27 2010
# -- Loading package standard
# -- Loading package numeric_bit
# -- Compiling entity test
# -- Compiling architecture equ of test

ModelSim>
```

2.13 VHDL Libraries

Overloaded operators in numeric_bit

	overloaded operators	left and right operand pairs
arithmetic	+, -, *, /, rem, mod	unsigned and unsigned unsigned and natural natural and unsigned signed and signed signed and integer integer and signed
relational	=, /=, >, <, >=, <=	
logical	not, and, or, nand, nor, xor, xnor	unsigned and unsigned signed and signed
shifting	shift_left, shift_right, rotate_left, rotate_right, sll, srl, rol, ror	

2.13 VHDL Libraries

+ and **-** operators with **unsigned** operands of different lengths

`"1011" + "110" = "1011" + "0110" = "0001"`

The carry is discarded

The shortest operand
will be extended by
filling in 0's on the left

`Sum <= A + B + 1;`

acceptable: unsigned and integer

`Sum <= A + B + carry;`

carry of type **bit** is **not** allowed

`unsigned'(0 => carry)`

type conversion

2.13 VHDL Libraries

*Qualified Expression

Qualified Expression

Used to define the type of an expression where otherwise the type would be ambiguous.

Syntax

```
{either}  
TypeName' (Expression)  
TypeName' Aggregate
```

Where

See Expression

Rules

The *Expression* or *Aggregate* must be compatible with the *TypeName*; a qualified expression is not a type conversion!

Tips

Use a qualified expression when the compiler gives an error indicating that the type of an expression is ambiguous. This can occur when calling overloaded functions and procedures (e.g. (1) and (2) below), and when constructing aggregates (e.g. (3) and (4) below).

Example

```
subtype T is STD_LOGIC_VECTOR(1 to 2);  
...  
if U > UNSIGNED("10000000") then      -- (1)  
    WRITE (L, STRING("Hello"));        -- (2)  
    V := (others => T'(others => '1')); -- (3)  
    case T'(A, B) is                   -- (4)
```

See Also

Aggregate, Expression, Type Conversion, TEXTIO

2.13 VHDL Libraries

*Aggregate

Aggregate

A way to write a value for any array or record.

Syntax

```
([Choices =>] Expression, ...)
```

```
Choices = Choice | ...
```

```
Choice = {either}
```

```
ConstantExpression
```

```
Range
```

```
others {the last choice}
```

Where

See Expression, Target

Rules

- An aggregate must give a value for every element of the array or record.
- The two forms of syntax (ordered list or explicitly named choices) can be mixed, but the ordered values must come before the named choices.

Gotchas!

Aggregates frequently need to be qualified to disambiguate their type (see example below).

Synthesis

Many synthesis tools do not allow aggregates as targets of assignments.

Tips

The aggregate (**others** => *Expression*) is a very useful way of setting all the elements of an array to the same value; you do not even have to know how big the array is! For example, to set the value of a parameter of an unconstrained array type.

Example

```
('0', '1', '0', '1')
(1, 2, 3, 4, 5)
(1 => A, 2 => B, 3 => C)
(1, 2, 3, others => 4)
(others => 'Z')
(A, B, C) := D;           -- Aggregate as the target of an assignment
T'(others => '0')          -- Qualified expression
(others => T'(others => '0'))
```

See Also

Array, Record, Expression, Range

4-bit adder in Chp2.4

```
entity FullAdder is
```

```
  port (X, Y, Cin: in bit; Cout, Sum: out bit);
```

```
end FullAdder;
```

```
architecture Equations of FullAdder is
```

```
begin
```

```
  Sum <= X xor Y xor Cin after 10 ns;
```

```
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
```

```
end Equations
```

```
entity Adder4 is
```

```
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;      -- Inputs  
        S: out bit_vector(3 downto 0); Co: out bit);      -- Outputs
```

```
end Adder4;
```

```
architecture Structure of Adder4 is
```

```
component FullAdder
```

```
  port (X, Y, Cin: in bit;      -- Inputs  
        Cout, Sum: out bit);    -- Outputs
```

```
end component;
```

```
signal C: bit_vector(3 downto 1);
```

```
begin    --instantiate four copies of the FullAdder
```

```
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
```

```
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
```

```
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
```

```
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
```

```
end Structure;
```

bit_vector is used

VHDL Code for 4-Bit Adder Using **Unsigned** Vectors

```
library IEEE;  
use IEEE.numeric_bit.all;  
  
entity Adder4_v2 is  
    port(A, B      : in unsigned(3 downto 0);  
         Ci       : in bit;  
         S        : out unsigned(3 downto 0);  
         Co       : out bit);  
end Adder4_v2;
```

unsigned is used

```
architecture overload of Adder4_v2 is  
    signal Sum5: unsigned(4 downto 0);  
begin  
    Sum5 <= '0' & A + B + unsigned'(0 => Ci); -- adder  
    S    <= Sum5(3 downto 0);  
    Co   <= Sum5(4);  
end overload;
```

Conversion functions in numeric_bit

TO_INTEGER(A)	converts an unsigned vector A to an integer
TO_UNSIGNED(B,N)	converts an integer to an unsigned vector of length N
UNSIGNED(A)	causes the compiler to treat a bit_vector A as an unsigned vector
BIT_VECTOR(B)	causes the compiler to treat a unsigned vector as a bit_vector

2.13 VHDL Libraries

```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.numeric_std.all;
```

If multivalued logic is desired, **numeric_std** can be used as **numeric_bit**

numeric_std defines unsigned and signed types as **std_logic_vectors** instead of **bit_vectors**

3 statements are required to use **numeric_std**

numeric_std defines the same set of overloaded operators and functions on unsigned and signed numbers as **number_bit**

VHDL Code for 4-Bit Adder Using **std_logic_unsigned**

```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;  
  
entity Adder4_v3 is  
  port(A, B      : in      std_logic_vector(3 downto 0);  
        Ci       : in      std_logic;  
        S        : out     std_logic_vector(3 downto 0);  
        Co       : out     std_logic);  
end Adder4_v3;  
  
architecture overload of Adder4_v3 is  
  signal Sum5: std_logic_vector(4 downto 0);  
  begin  
    Sum5 <= '0' & A + B + Ci; --adder  
    S    <= Sum5(3 downto 0);  
    Co   <= Sum5(4);  
  end overload;
```

std_logic_unsigned is used,
std_logic_vector is used


```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;
```

std_logic_unsigned does not define unsigned types, but instead it defines some overloaded **arithmetic** operators for **std_logic_vectors**

These operators treat std_logic_vectors as if they were unsigned numbers

- When used with std_logic_1164, both **arithmetic** and **logic** operations can be performed
- std_logic_1164 defines the **logic** operations

std_logic_unsigned is not an IEEE standard, but it is commonly placed in the IEEE library

VHDL Code for 4-Bit Adder Using the **std_logic_unsigned** Package

```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;  
  
entity Adder4_v3 is  
  port(A, B      : in      std_logic_vector(3 downto 0);  
        Ci       : in      std_logic;  
        S        : out     std_logic_vector(3 downto 0);  
        Co       : out     std_logic);  
end Adder4_v3;  
  
architecture overload of Adder4_v3 is  
  signal Sum5: std_logic_vector(4 downto 0);  
  begin  
    Sum5 <= '0' & A + B + Ci;  
    S    <= Sum5(3 downto 0);  
    Co   <= Sum5(4);  
  end overload;
```

Type conversion is not needed

std_logic_unsigned provides an overloaded operator to add a std_logic bit to a std_logic_vector

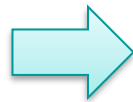
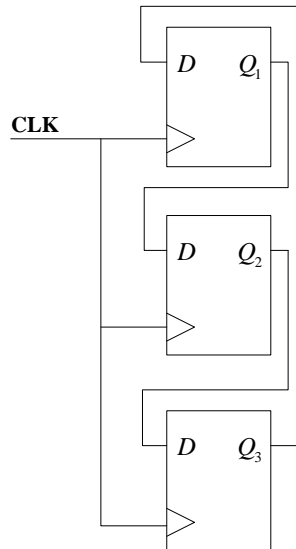
Packages	Type defined	Conversion functions	Logic operations	Arithmetic operations
std_logic_1161	std_logic std_logic_vector		✓	x
numeric_bit	(un)signed (using bit_vector)	to_integer(A) to_unsigned(B,N) unsigned(A) bit_vector(B)	✓	✓
numeric_std	(un)signed (using std_logic_vector)		✓	✓
std_logic_unsigned			x	✓

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.14 Modeling registers and counters using VHDL processes

Cyclic Shift Register(循环移位寄存器)

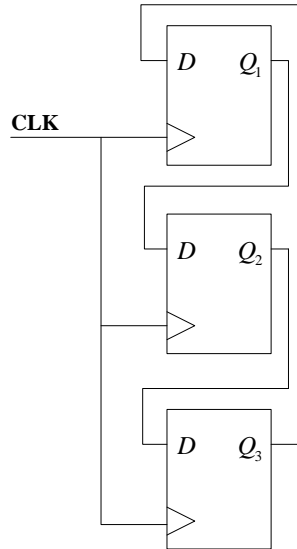


```
process (CLK)
begin
    if CLK'event and CLK = '1' then
        Q1 <= Q3 after 5 ns;
        Q2 <= Q1 after 5 ns;
        Q3 <= Q2 after 5 ns;
    end if ;
end process;
```

When several flip-flops change state on the same clock edge, statements representing these flip-flops can be placed in the same clocked process

2.14 Modeling registers and counters using VHDL processes

Cyclic Shift Register

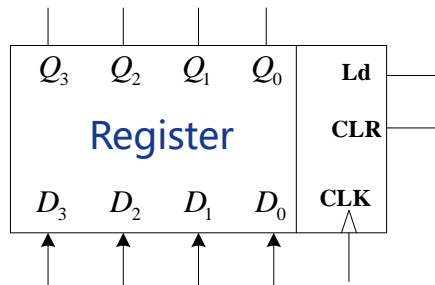


```
process (CLK)
begin
    if CLK'event and CLK = '1' then
        Q1 <= Q3 after 5 ns;
        Q2 <= Q1 after 5 ns;
        Q3 <= Q2 after 5 ns;
    end if ;
end process;
```

- ❑ The **old values** of Q1, Q2, and Q3 are used to compute the new values
- ❑ At CLK ↑, all of the D inputs are loaded into the flip-flops, but the state change does not occur until after a propagation delay

2.14 Modeling registers and counters using VHDL processes

Register with Synchronous Clear and Load

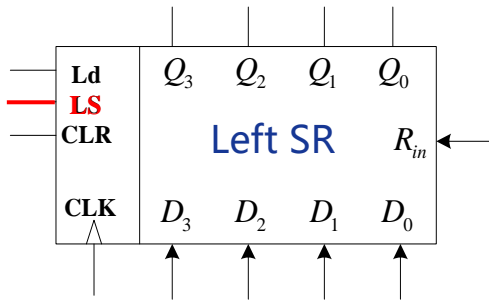


```
process (CLK)
begin
    if CLK'event and CLK = '1' then
        if CLR = '1' then Q <= "0000";
        elsif Ld = '1' then Q <= D;
        end if ;
    end if;
end process;
```

- ❑ The register can be loaded or cleared on CLK ↑
 - ❑ If CLR = '1', the register is cleared
 - ❑ If Ld = '1', D are loaded into the register

- ❑ Since the register outputs can only change on CLK ↑, CLR and Ld are not on the sensitivity list
- ❑ If CLR = Ld = '0', no change of Q occurs

Left Shift Register with Synchronous Clear and Load

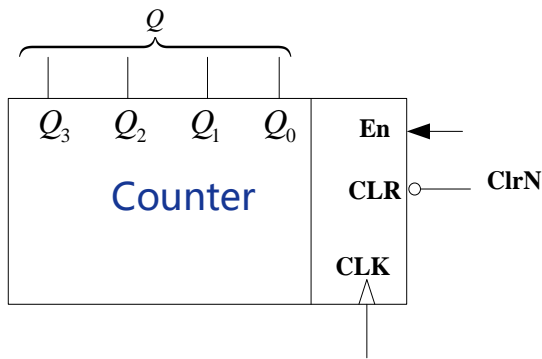


```
process(CLK)
begin
  if CLK'event and CLK = '1' then
    if CLR = '1' then Q <= "0000";
    elsif Ld = '1' then Q <= D;
    elsif LS = '1' then Q <= Q(2 downto 0) & Rin;
    end if ;
  end if;
end process;
```

When **LS**='1', the contents of the register are shifted left and the rightmost bit is set equal to R_{in}

If $CLR = Ld = LS = '0'$, Q remains unchanged

Synchronous counter (同步计数器)

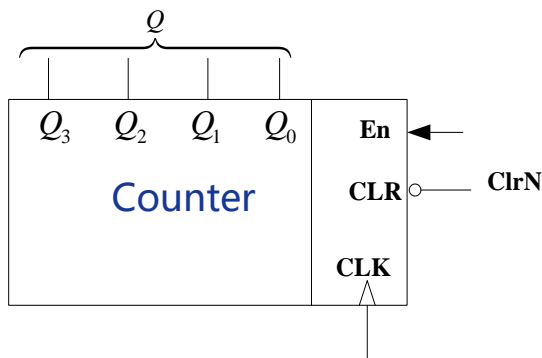


```
signal Q: unsigned (3 downto 0);  
process (CLK)  
begin  
    if CLK'event and CLK = '1' then  
        if ClrN = '0' then Q <= "0000";  
        elsif En = '1' then Q <= Q+1;  
        end if ;  
    end if;  
end process;
```

On $CLK \uparrow$,

- ❑ counter is cleared when $ClrN='0'$
- ❑ counter is incremented when $ClrN=En='1'$

Synchronous counter



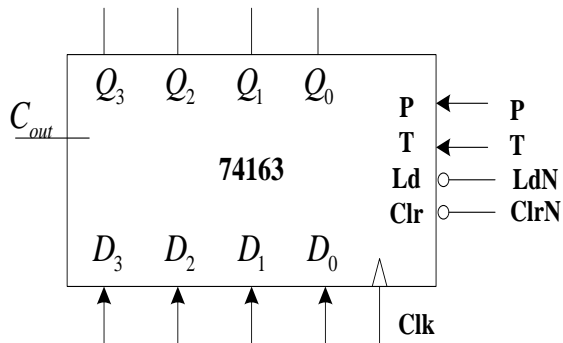
```
signal Q: unsigned (3 downto 0);  
process (CLK)  
begin  
    if CLK'event and CLK = '1' then  
        if ClrN = '0' then Q <= "0000";  
        elsif En = '1' then Q <= Q+1;  
        end if ;  
    end if;  
end process;
```

Since addition is not defined for bit-vectors, we have declared Q to be of type **unsigned**

Then we can increment the counter using the "+" operator that is overloaded in the **ieee.numeric_bit** package

Q+1: unsigned and integer, acceptable

74163 Counter Operation



Control Signals			Next State				
ClrN	LdN	PT	Q3+	Q2+	Q1+	Q0+	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	1	present state + 1				(increment count)
1	1	0	Q3	Q2	Q1	Q0	(no change)

Both count-enable inputs (P and T) must be high to count

```
library IEEE;
```

```
use IEEE.numeric_bit.all;
```

```
entity c74163 is
```

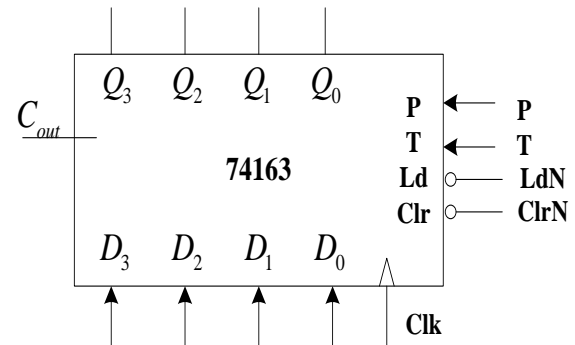
```
    port( LdN, ClrN, P, T, Clk    : in    bit;
```

```
          D                        : in    unsigned(3 downto 0);
```

```
          Cout                    : out    bit;
```

```
          Qout                    : out    unsigned(3 downto 0));
```

```
end c74163;
```

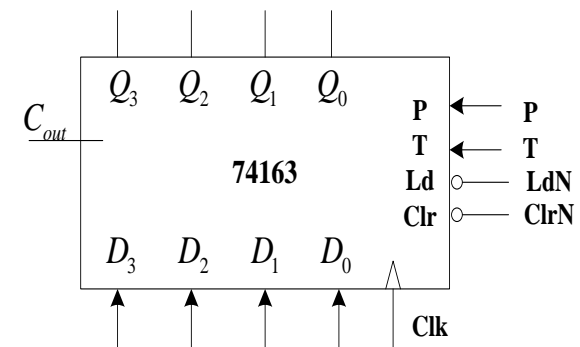


Control Signals			Next State				
ClrN	LdN	PT	Q3+	Q2+	Q1+	Q0+	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	1	present state +1				(increment count)
1	1	0	Q3	Q2	Q1	Q0	(no change)

```

architecture b74163 of c74163 is
signal Q: unsigned(3 downto 0);
begin
    Qout <= Q;
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
    process (Clk)
    begin
        if Clk'event and Clk = '1' then
            if ClrN = '0' then Q <= "0000";
            elsif LdN = '0' then Q <= D;
            elsif (P and T) = '1' then Q <= Q + 1;
            end if;
        end if;
    end process;
end b74163;

```



Control Signals			Next State				
ClrN	LdN	PT	Q3+	Q2+	Q1+	Q0+	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	1	present state +1				(increment count)
1	1	0	Q3	Q2	Q1	Q0	(no change)

```

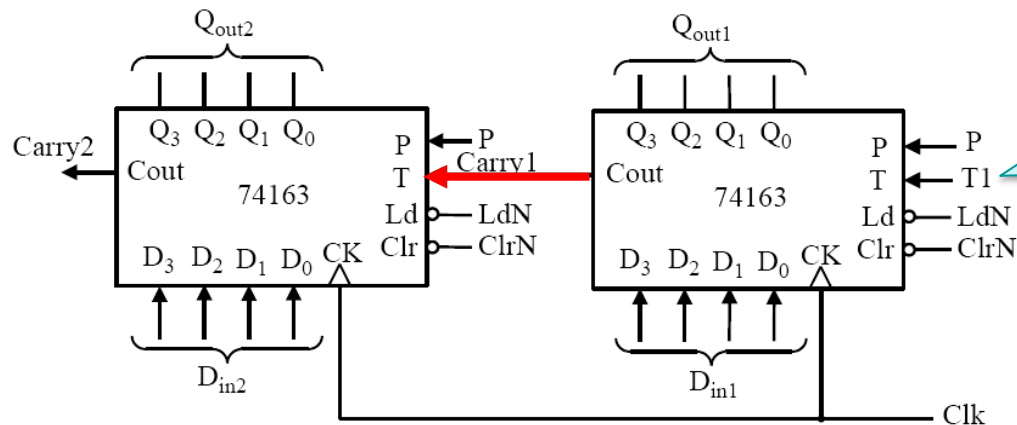
architecture b74163 of c74163 is
signal Q: unsigned(3 downto 0);
begin
    Qout <= Q;
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
    process (Clk)
    begin
        if Clk'event and Clk = '1' then
            if ClrN = '0' then Q <= "0000";
            elsif LdN = '0' then Q <= D;
            elsif (P and T) = '1' then Q <= Q + 1;
            end if;
        end if;
    end process;
end b74163;

```

Q is of type **unsigned**

2.14 Modeling registers and counters using VHDL processes

Two 74163 Counters Cascaded to Form an 8-bit Counter

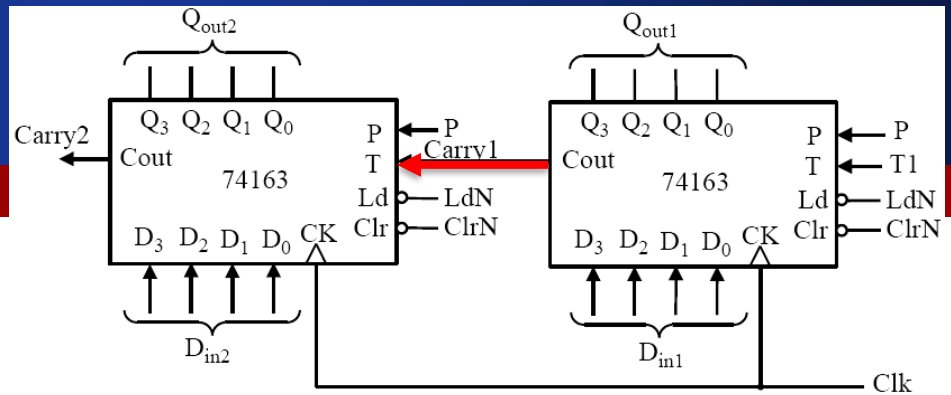


1. When right counter is in state 1111 and T1='1', Carry1 = '1'

2. For left counter, PT='1' if P='1'. If PT='1', on next Clk ↑, right counter +1 to 0000, at the same time left counter +1

```
library IEEE;
use IEEE.numeric_bit.all;

entity eight_bit_counter is
    port(ClrN, LdN, P, T1, Clk : in    bit;
         Din1, Din2           : in    unsigned(3 downto 0);
         Count                : out   integer range 0 to 255;
         Carry2               : out   bit);
end eight_bit_counter;
```



```

architecture cascaded_counter of eight_bit_counter is
  component c74163
    port(LdN, ClrN, P, T, Clk      : in    bit;
         D                        : in    unsigned(3 downto 0);
         Cout                    : out    bit;
         Qout                    : out    unsigned(3 downto 0));
  end component;

  signal Carry1: bit;
  signal Qout1, Qout2: unsigned(3 downto 0);
begin
  ct1: c74163 port map (LdN,ClrN,P,T1,Clk,Din1,Carry1,Qout1);
  ct2: c74163 port map (LdN,ClrN,P,Carry1,Clk,Din2,Carry2,Qout2);
  Count <= to_integer(Qout2 & Qout1);
end cascaded_counter;

```

convert unsigned vector to integer

Synthesis of VHDL Code

```
C <= A and B after 5 ns;  
E <= C or D after 5 ns;
```

- A VHDL synthesizer cannot synthesize delays
- Clauses of the form "after time expression" will be ignored by the synthesizers

```
port(A, B: in integer := 2; C, D: out bit);
```

- Although initial values for signals may be specified in port and signal declarations, these initial values are ignored by the synthesizer
- A reset signal should be provided if the hardware must be set to a specific initial state

Synthesis of VHDL Code

- When an **integer** signal is synthesized, the integer is represented in hardware by its binary equivalent
- If the range of an integer is not specified, the synthesizer will assume the maximum number of bits, usually 32

signal count: integer range 0 to 7;

This would result in a **3-bit** counter

signal count: integer;

It could result in a **32-bit** counter

Synthesis of VHDL Code

- VHDL signals retain their current values until they are changed
- This can result in creation of unwanted latches when the code is synthesized

if X = '1' then B <= 1; end if;

- For example, in a combinational process, the statement would create **latches** to hold the value of B when X changes to '0'

if X = '1' then B <= 1 else B <= 0; end if;

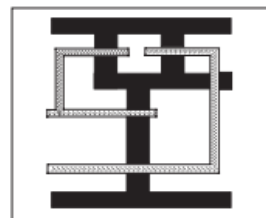
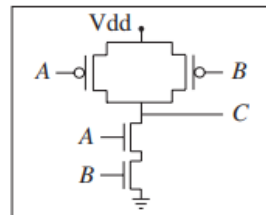
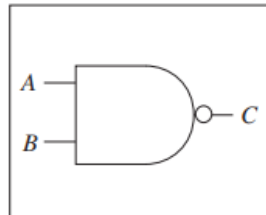
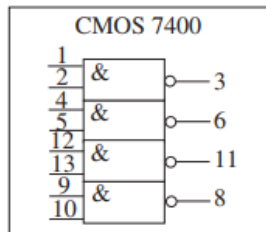
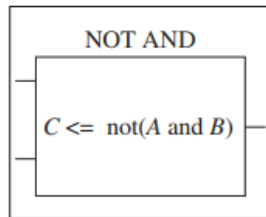
- To avoid creation of unwanted latches in a combinational process, always include an **else** clause in every **if** statement
- For example, this would create a MUX to switch the value of B from 1 to 0

Chapter 2 Introduction to VHDL

1	Computer-aided design	11	Simple synthesis examples
2	Hardware description language	12	VHDL models for multiplexers
3	VHDL description of combinational circuits	13	VHDL libraries
4	VHDL modules	14	Modeling registers and counters using VHDL processes
5	Sequential statements and VHDL processes	15	Behavioral and structural VHDL
6	Modeling flip-flop using VHDL processes	16	Variables, signals, and constants
7	Processes using wait statements	17	Arrays
8	Two types of VHDL delays: Transport and inertial delays	18	Loops in VHDL
9	Compilation, simulation, and synthesis of VHDL code	19	Assert and report statements
10	VHDL data types and operators		

2.15 Behavioral and Structural VHDL

Different Levels of Abstraction



Behavior

SSI Gates

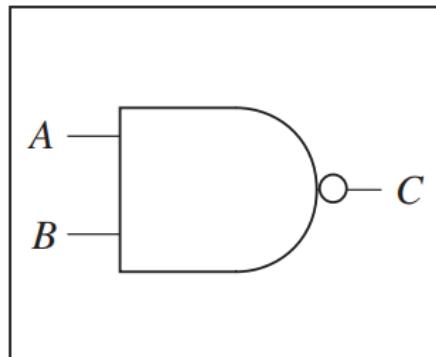
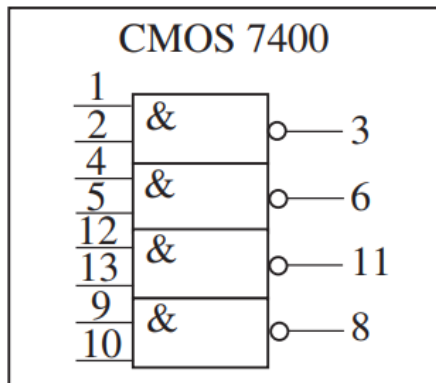
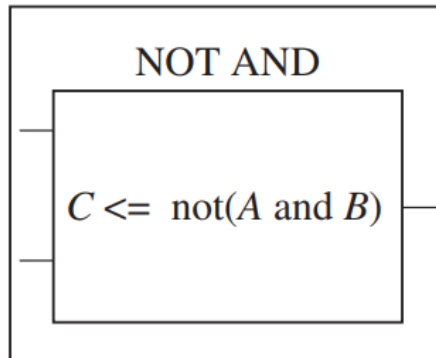
Logic

Transistor

Layout

When hearing the term **NAND**, different designers, depending on the domain of their design level, think of these different representations of the same NAND device

2.15 Behavioral and Structural VHDL



Behavior

Some would think of just a block representing the **behavior** of a NAND operator

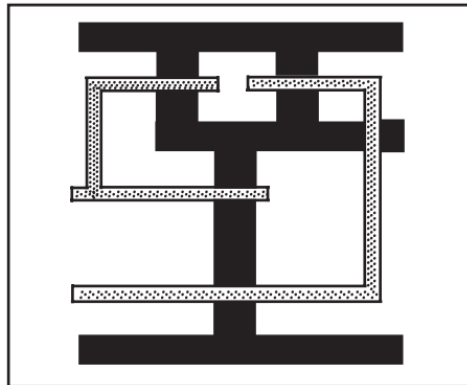
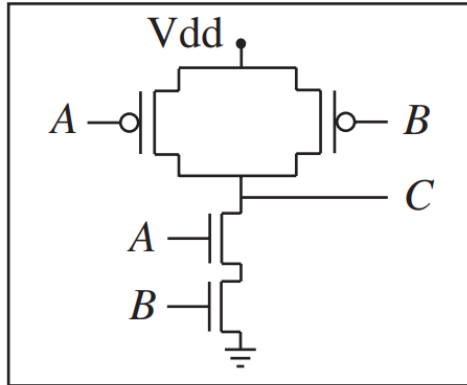
SSI Gates

Some others might think of the 4 **gates** in a CMOS 7400 chip

Logic

For designers who work at the logic level they think of the **logic** symbol for a NAND gate

2.15 Behavioral and Structural VHDL



↓
Transistor

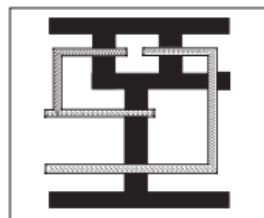
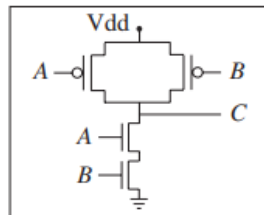
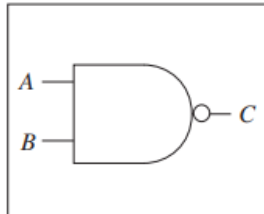
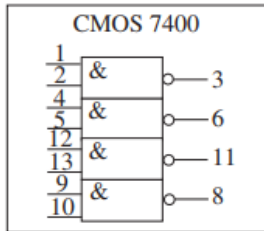
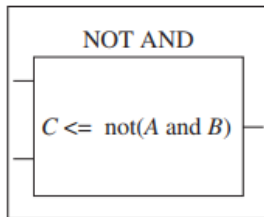
Transistor-level circuit designers think of the transistor-level circuit to achieve the NAND functionality

↓
Layout

What passes through the mind of a physical level designer is the layout of a NAND gate

2.15 Behavioral and Structural VHDL

Different Levels of Abstraction



Behavior

SSI Gates

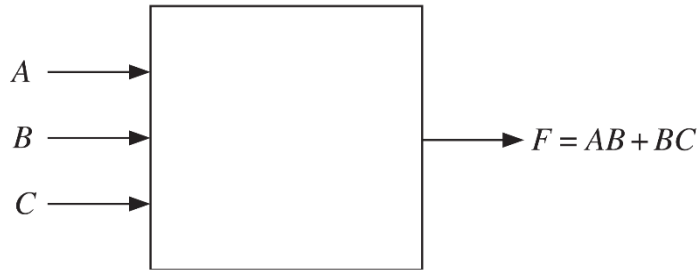
Logic

Transistor

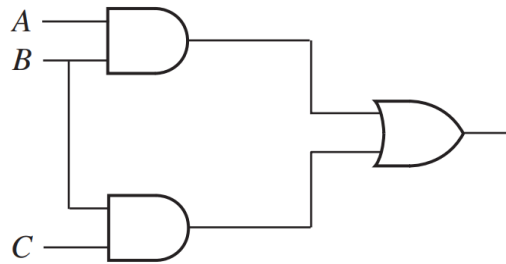
Layout

All of the figures represent the same device, but they are differ in the amout of detail provided in the description

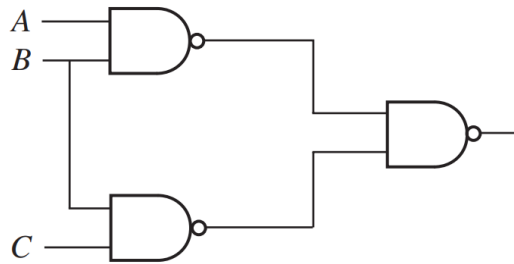
2.15 Behavioral and Structural VHDL



Two Implementations of $F = AB + BC$



(a) using AND-OR



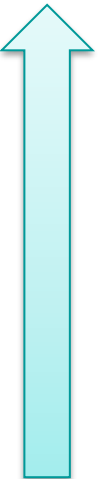
(b) using NAND

A **structural** description gives different descriptions, whereas the same behavioral description could result in either of these two representations

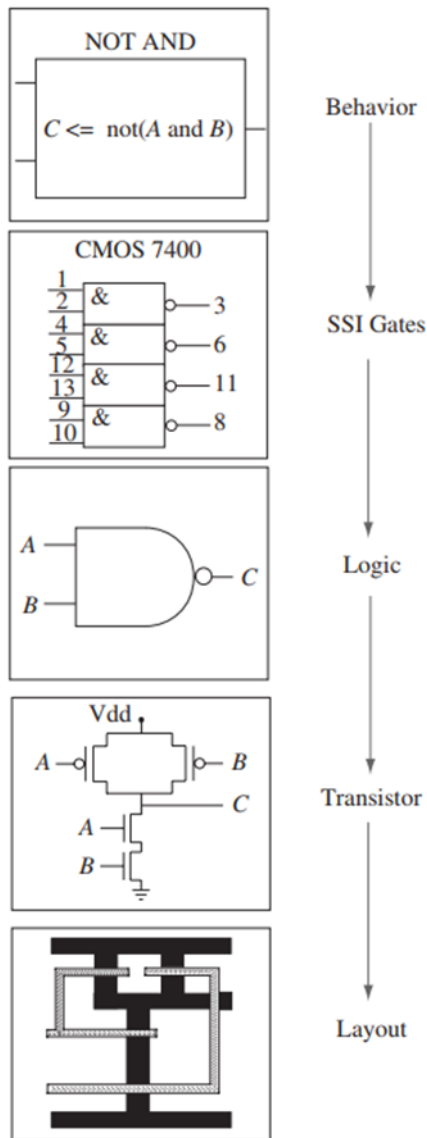
- A **structural** description specifies more details
- A **behavioral** level description only specifies the behavior at a higher level of abstraction

2.15 Behavioral and Structural VHDL

VHDL Models		Abstraction Level
Behavioral models	Only the overall behavior is specified	High
Dataflow models	Using logic equations directly (Data path and control signals are specified)	Intermediate
Structural models	The components used and the structure of the interconnection between the components are clearly specified	Low



2.15 Behavioral and Structural VHDL

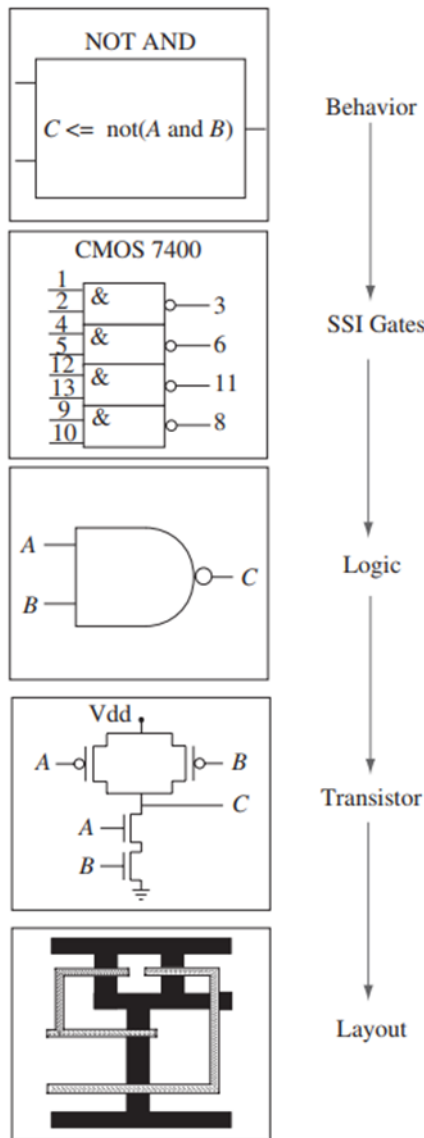


- ❑ If designs are specified at higher levels of abstraction, they need to get converted to the lower levels in order to get implemented

- ❑ In the early days of design automation, there were not enough automatic software tools to perform this conversion
- ❑ Hence, designs needed to be specified at the lower level of abstraction
- ❑ Designs were entered using schematic capture or abstraction

- ❑ Nowadays, synthesis tools perform very efficient conversion of behavioral level design into target technologies

2.15 Behavioral and Structural VHDL



- ❑ Behavior and structural design techniques are often combined
- ❑ Different parts of the design are often done with different techniques

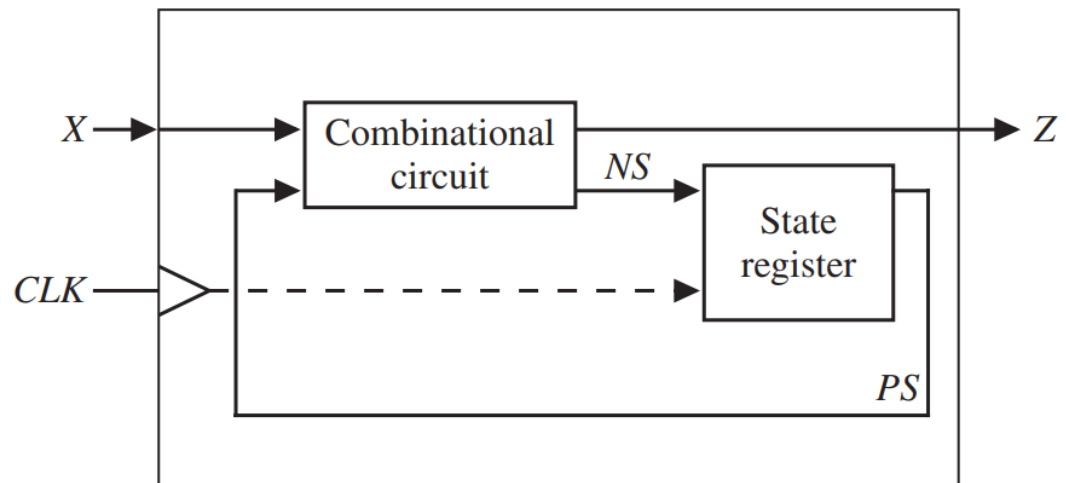
- ❑ State-of-the-art design automation tools generate efficient hardware for logic and arithmetic circuits
- ❑ Hence, a large part of those designs is done at the behavioral level

- ❑ However, memory structures often need **manual optimizations** and are done by custom design, as opposed to automatic synthesis

2.15.1 Modeling a Sequential Machine

In the following, we discuss several ways of writing VHDL descriptions for sequential machines

PS	NS		Z	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



We first write a **behavioral** model for a Mealy sequential circuit (BCD to excess-3 code converter)

PS and **NS** are not visible externally

2.15.1 Modeling a Sequential Machine

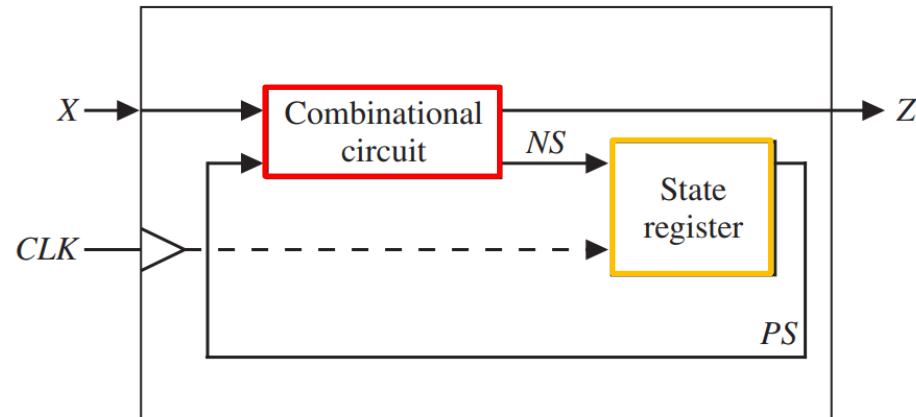
<i>PS</i>	<i>NS</i>		<i>Z</i>	
	<i>X</i> = 0	<i>X</i> = 1	<i>X</i> = 0	<i>X</i> = 1
<i>S</i> ₀	<i>S</i> ₁	<i>S</i> ₂	1	0
<i>S</i> ₁	<i>S</i> ₃	<i>S</i> ₄	1	0
<i>S</i> ₂	<i>S</i> ₄	<i>S</i> ₄	0	1
<i>S</i> ₃	<i>S</i> ₅	<i>S</i> ₅	0	1
<i>S</i> ₄	<i>S</i> ₅	<i>S</i> ₆	1	0
<i>S</i> ₅	<i>S</i> ₀	<i>S</i> ₀	0	1
<i>S</i> ₆	<i>S</i> ₀	—	1	—



```
entity Code_Converter1 is
    port(X, CLK : in bit;
          Z      : out bit);
end Code_Converter1;
```

Behavioral model for code converter using two processes

PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S ₀	S ₁	S ₂	1	0
S ₁	S ₃	S ₄	1	0
S ₂	S ₄	S ₄	0	1
S ₃	S ₅	S ₅	0	1
S ₄	S ₅	S ₆	1	0
S ₅	S ₀	S ₀	0	1
S ₆	S ₀	—	1	—



```

architecture Behavioral of Code_Converter1 is
signal State, Nextstate: integer range 0 to 6;
begin

```

Here we use **integer** to represent the states

```

process(State, X) -- Combinational Circuit
begin
    case State is
end process;

```

Combinational circuit

```

process(CLK) -- State Register
begin
    if CLK'EVENT and CLK = '1' then -- rising edge of clock
        State <= Nextstate;
    end if;
end process;

```

State register

```

end Behavioral;

```

2.15.1 Modeling a Sequential Machine

PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S ₀	S ₁	S ₂	1	0
S ₁	S ₃	S ₄	1	0
S ₂	S ₄	S ₄	0	1
S ₃	S ₅	S ₅	0	1
S ₄	S ₅	S ₆	1	0
S ₅	S ₀	S ₀	0	1
S ₆	S ₀	—	1	—

Combinational circuit:
Update Z and Nextstate

Others cases should
not occur

```

process(State, X)                                -- Combinational Circuit
begin
    case State is
        when 0 =>
            if X = '0' then Z <= '1'; Nextstate <= 1;
            else Z <= '0'; Nextstate <= 2; end if;
        when 1 =>
            if X = '0' then Z <= '1'; Nextstate <= 3;
            else Z <= '0'; Nextstate <= 4; end if;
        when 2 =>
            if X = '0' then Z <= '0'; Nextstate <= 4;
            else Z <= '1'; Nextstate <= 4; end if;
        when 3 =>
            if X = '0' then Z <= '0'; Nextstate <= 5;
            else Z <= '1'; Nextstate <= 5; end if;
        when 4 =>
            if X = '0' then Z <= '1'; Nextstate <= 5;
            else Z <= '0'; Nextstate <= 6; end if;
        when 5 =>
            if X = '0' then Z <= '0'; Nextstate <= 0;
            else Z <= '1'; Nextstate <= 0; end if;
        when 6 =>
            if X = '0' then Z <= '1'; Nextstate <= 0;
            else Z <= '0'; Nextstate <= 0; end if;
        when others => null;                        -- should not occur
    end case;
end process;
    
```

2.15.1 Modeling a Sequential Machine

```
process(State, X)                                -- Combinational Circuit
begin
  case State is
    when 0 =>
      if X = '0' then Z <= '1'; Nextstate <= 1;
      else Z <= '0'; Nextstate <= 2; end if;
    when 1 =>
      if X = '0' then Z <= '1'; Nextstate <= 3;
      else Z <= '0'; Nextstate <= 4; end if;
    when 2 =>
      if X = '0' then Z <= '0'; Nextstate <= 4;
      else Z <= '1'; Nextstate <= 4; end if;
    when 3 =>
      if X = '0' then Z <= '0'; Nextstate <= 5;
      else Z <= '1'; Nextstate <= 5; end if;
    when 4 =>
      if X = '0' then Z <= '1'; Nextstate <= 5;
      else Z <= '0'; Nextstate <= 6; end if;
    when 5 =>
      if X = '0' then Z <= '0'; Nextstate <= 0;
      else Z <= '1'; Nextstate <= 0; end if;
    when 6 =>
      if X = '0' then Z <= '1'; Nextstate <= 0;
      else Z <= '0'; Nextstate <= 0; end if;
    when others => null;                          -- should not occur
  end case;
end process;
```

- The statement **when others => null** is not actually needed here
- It is because the outputs and next states of all possible values of State are explicitly specified

However, it should be included:

- whenever the else clause of any if statement is omitted or
- when actions for all possible values of State are not specified

2.15.1 Modeling a Sequential Machine

```
when 6 =>  
    if X = '0' then Z <= '1'; Nextstate <= 0;  
    else Z <= '0'; Nextstate <= 0; end if;  
    when others => null;           -- should not occur  
end case;  
end process;
```

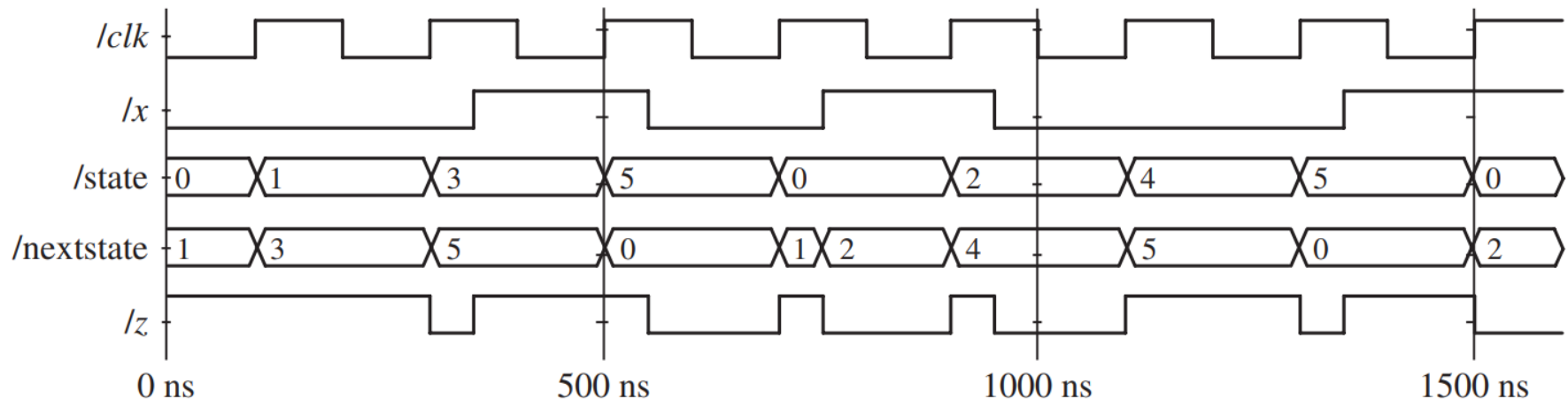
- The **null** implies no action, which is appropriate since the other values of State should never occur
- If **else** clauses are omitted or actions for any conditions are unspecified, synthesis typically results in creation of latches

2.15.1 Modeling a Sequential Machine

Simulator command

```
add wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

```
force signal_name v1 t1, v2 t2, ...
```



2.15.1 Modeling a Sequential Machine

<i>PS</i>	<i>NS</i>		<i>Z</i>	
	<i>X</i> = 0	<i>X</i> = 1	<i>X</i> = 0	<i>X</i> = 1
<i>S</i> ₀	<i>S</i> ₁	<i>S</i> ₂	1	0
<i>S</i> ₁	<i>S</i> ₃	<i>S</i> ₄	1	0
<i>S</i> ₂	<i>S</i> ₄	<i>S</i> ₄	0	1
<i>S</i> ₃	<i>S</i> ₅	<i>S</i> ₅	0	1
<i>S</i> ₄	<i>S</i> ₅	<i>S</i> ₆	1	0
<i>S</i> ₅	<i>S</i> ₀	<i>S</i> ₀	0	1
<i>S</i> ₆	<i>S</i> ₀	—	1	—



```
entity Code_Converter2 is
    port (X, CLK : in bit;
          Z      : out bit);
end Code_Converter2;
```

Behavioral model for code converter using a single process

```
architecture one_process of Code_Converter2 is
signal State: integer range 0 to 6 := 0;
begin
```

```
    process (CLK)
    begin
        if CLK'event and CLK = '1' then
            case State is
                when 0 =>
                    if X = '0' then State <= 1; else State <= 2; end if;
                when 1 =>
                    if X = '0' then State <= 3; else State <= 4; end if;
                when 2 =>
                    State <= 4;
                when 3 =>
                    State <= 5;
                when 4 =>
                    if X = '0' then State <= 5; else State <= 6; end if;
                when 5 =>
                    State <= 0;
                when 6 =>
                    State <= 0;
            end case;
        end if;
    end process;
```

The process updates the state

PS	NS	
	X = 0	X = 1
S ₀	S ₁	S ₂
S ₁	S ₃	S ₄
S ₂	S ₄	S ₄
S ₃	S ₅	S ₅
S ₄	S ₅	S ₆
S ₅	S ₀	S ₀
S ₆	S ₀	—

```
    Z <= '1' when (State = 0 and X = '0') or (State = 1 and X = '0')
                or (State = 2 and X = '1') or (State = 3 and X = '1')
                or (State = 4 and X = '0') or (State = 5 and X = '1')
                or State = 6
                else '0';
end one_process;
```


Behavioral model for code converter using a single process

```
architecture one_process of Code_Converter2 is
signal State: integer range 0 to 6 := 0;
begin
    process(CLK)
    begin
        if CLK'event and CLK = '1' then
            case State is
                when 0 =>
                    if X = '0' then State <= 1; else State <= 2; end if;
                when 1 =>
                    if X = '0' then State <= 3; else State <= 4; end if;
                when 2 =>
                    State <= 4;
                when 3 =>
                    State <= 5;
                when 4 =>
                    if X = '0' then State <= 5; else State <= 6; end if;
                when 5 =>
                    State <= 0;
                when 6 =>
                    State <= 0;
            end case;
        end if;
    end process;

    Z <= '1' when (State = 0 and X = '0') or (State = 1 and X = '0')
        or (State = 2 and X = '1') or (State = 3 and X = '1')
        or (State = 4 and X = '0') or (State = 5 and X = '1')
        or State = 6
    else '0';
end one_process;
```

update output Z

PS	Z	
	X = 0	X = 1
S ₀	1	0
S ₁	1	0
S ₂	0	1
S ₃	0	1
S ₄	1	0
S ₅	0	1
S ₆	1	—

Sequential machine model using equations (dataflow approach)

		XQ ₁			
		00	01	11	10
Q ₂ Q ₃	00	1	1	1	1
	01	X	1	1	X
	11	0	0	0	0
	10	0	0	0	X

$$D_1 = Q_1^+ = Q_2'$$

		XQ ₁			
		00	01	11	10
Q ₂ Q ₃	00	0	1	1	0
	01	X	1	1	X
	11	0	1	1	0
	10	0	1	1	X

$$D_2 = Q_2^+ = Q_1$$

		XQ ₁			
		00	01	11	10
Q ₂ Q ₃	00	0	1	0	1
	01	X	0	0	X
	11	0	1	1	0
	10	0	1	0	X

$$D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$$

		XQ ₁			
		00	01	11	10
Q ₂ Q ₃	00	1	1	0	0
	01	X	0	1	X
	11	0	0	1	1
	10	1	1	0	X

$$Z = X'Q_3' + XQ_3$$

The dataflow VHDL model is based on the next state and output equations

2.15.1 Modeling a Sequential Machine

$$D_1 = Q_1^+ = Q_2' \quad D_2 = Q_2^+ = Q_1 \quad D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$$
$$Z = X'Q_3' + XQ_3$$



```
entity Code_Converter3 is
    port(X, CLK : in bit;
          Z      : out bit);
end Code_Converter3;

architecture Equations of Code_Converter3 is
    signal Q1, Q2, Q3: bit;
begin
    process(CLK)
    begin
        if CLK = '1' and CLK'event then -- rising edge of clock
            Q1 <= not Q2 after 10 ns;
            Q2 <= Q1 after 10 ns;
            Q3 <= (Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
                  (X and not Q1 and not Q2) after 10 ns;
        end if;
    end process;
    Z <= (not X and not Q3) or (X and Q3) after 20 ns;
end Equations;
```

2.15.1 Modeling a Sequential Machine

```
entity Code_Converter3 is
    port(X, CLK : in bit;
          Z      : out bit);
end Code_Converter3;

architecture Equations of Code_Converter3 is
    signal Q1, Q2, Q3: bit;
begin
    process(CLK)
    begin
        if CLK = '1' and CLK'event then -- rising edge of clock
            Q1 <= not Q2 after 10 ns;
            Q2 <= Q1 after 10 ns;
            Q3 <= (Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
                  (X and not Q1 and not Q2) after 10 ns;
        end if;
    end process;
    Z <= (not X and not Q3) or (X and Q3) after 20 ns;
end Equations;
```

- ❑ Note that in order to do VHDL modeling **using equations**, we need to perform **state assignments**, derive **next state equations**, and so on
- ❑ In contrast, at the **behavioral level**, the **state table** was sufficient to create the VHDL model

Structural model of sequential machine

```
entity Code_Converter4 is
    port( X, CLK : in bit;
          Z      : out bit);
end Code_Converter4;
```

```
architecture Structure of Code_Converter4 is
```

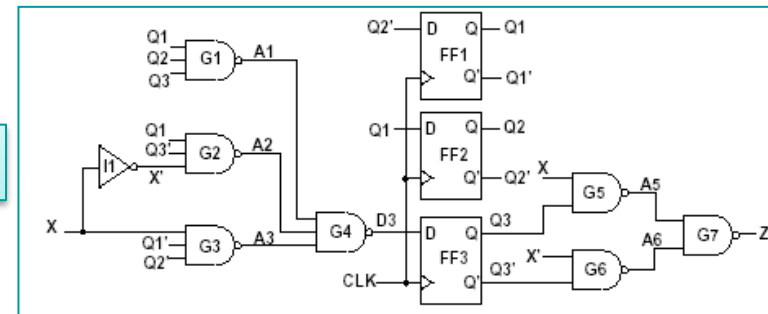
```
    component DFF
        port(D, CLK: in bit; Q: out bit; QN: out bit := '1');
    end component;
    component Nand2
        port(A1, A2: in bit; Z: out bit);
    end component;
    component Nand3
        port(A1, A2, A3: in bit; Z: out bit);
    end component;
    component Inverter
        port(A: in bit; Z: out bit);
    end component;
```

```
    signal A1, A2, A3, A5, A6, D3: bit;
    signal Q1, Q2, Q3: bit;
    signal Q1N, Q2N, Q3N, XN: bit;
begin
```

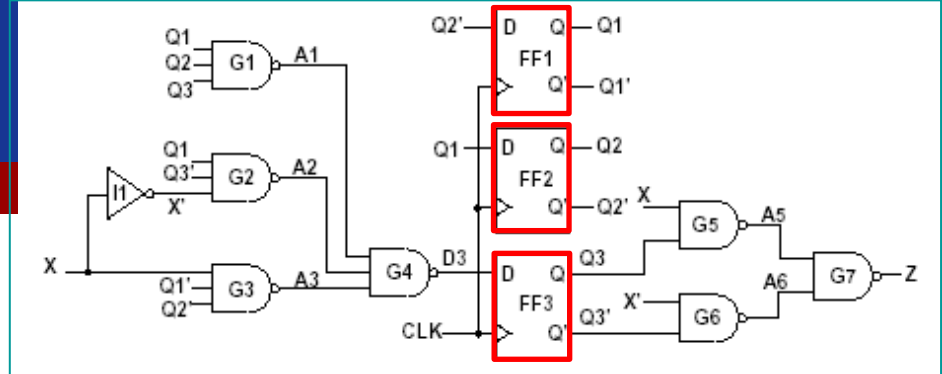
```
    I1 : Inverter port map (X, XN);
    G1 : Nand3     port map (Q1, Q2, Q3, A1);
    G2 : Nand3     port map (Q1, Q3N, XN, A2);
    G3 : Nand3     port map (X, Q1N, Q2N, A3);
    G4 : Nand3     port map (A1, A2, A3, D3);
    FF1: DFF       port map (Q2N, CLK, Q1, Q1N);
    FF2: DFF       port map (Q1, CLK, Q2, Q2N);
    FF3: DFF       port map (D3, CLK, Q3, Q3N);
    G5 : Nand2     port map (X, Q3, A5);
    G6 : Nand2     port map (XN, Q3N, A6);
    G7 : Nand2     port map (A5, A6, Z);
```

```
end Structure;
```

Components



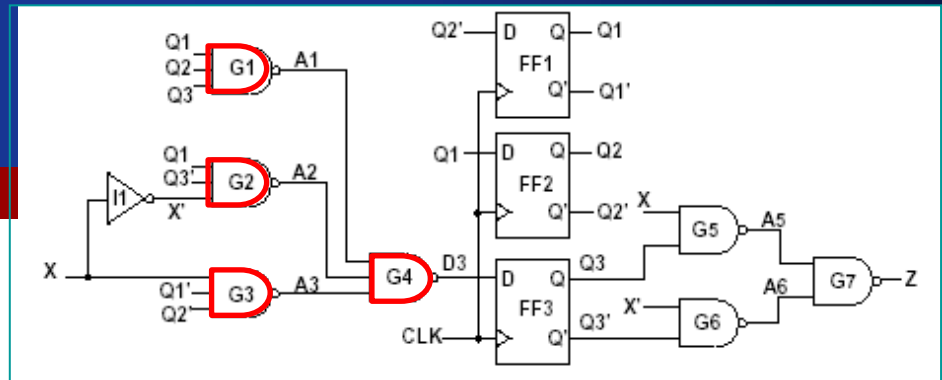
DFF module



```
--D Flip-Flop
entity DFF is
  port(D, CLK: in bit;
        Q: out bit;
        QN: out bit := '1');
end DFF;
```

initialize QN to '1' since bit signals are defaulted to '0'

```
architecture SIMPLE of DFF is
begin
  process(CLK) -- process is executed when CLK changes
  begin
    if CLK'event and CLK = '1' then --rising edge of clock
      Q <= D after 10 ns;
      QN <= not D after 10 ns;
    end if;
  end process;
end SIMPLE;
```



Nand3 module

--3 input NAND gate

entity Nand3 **is**

port(A1, A2, A3: **in bit**; Z: **out bit**);

end Nand3;

architecture concur **of** Nand3 **is**

begin

 Z <= **not** (A1 **and** A2 **and** A3) **after** 10 ns;

end concur;

Nand2 module

Inverter module

The Nand2 and Inverter modules are similar except for the number of inputs

Structural model of sequential machine

```
entity Code_Converter4 is
    port( X, CLK : in bit;
          Z       : out bit);
end Code_Converter4;
```

```
architecture Structure of Code_Converter4 is
```

```
    component DFF
```

```
        port(D, CLK: in bit; Q: out bit; QN: out bit := '1');
```

```
    end component;
```

```
    component Nand2
```

```
        port(A1, A2: in bit; Z: out bit);
```

```
    end component;
```

```
    component Nand3
```

```
        port(A1, A2, A3: in bit; Z: out bit);
```

```
    end component;
```

```
    component Inverter
```

```
        port(A: in bit; Z: out bit);
```

```
    end component;
```

```
    signal A1, A2, A3, A5, A6, D3: bit;
```

```
    signal Q1, Q2, Q3: bit;
```

```
    signal Q1N, Q2N, Q3N, XN: bit;
```

```
begin
```

```
    I1 : Inverter port map (X, XN);
```

```
    G1 : Nand3      port map (Q1, Q2, Q3, A1);
```

```
    G2 : Nand3      port map (Q1, Q3N, XN, A2);
```

```
    G3 : Nand3      port map (X, Q1N, Q2N, A3);
```

```
    G4 : Nand3      port map (A1, A2, A3, D3);
```

```
    FF1: DFF        port map (Q2N, CLK, Q1, Q1N);
```

```
    FF2: DFF        port map (Q1, CLK, Q2, Q2N);
```

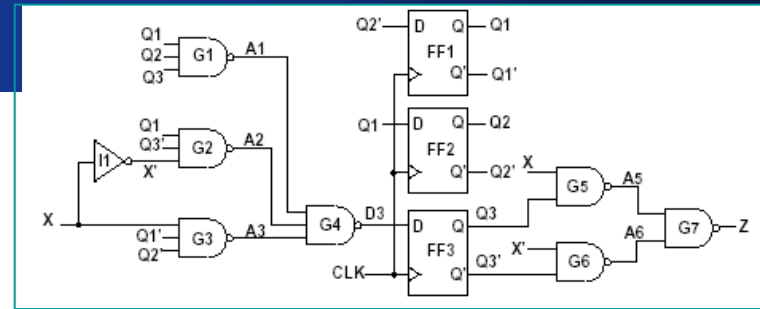
```
    FF3: DFF        port map (D3, CLK, Q3, Q3N);
```

```
    G5 : Nand2      port map (X, Q3, A5);
```

```
    G6 : Nand2      port map (XN, Q3N, A6);
```

```
    G7 : Nand2      port map (A5, A6, Z);
```

```
end Structure;
```



If we synthesized this structural description, we would get exactly the same circuit that we have in mind