CDA 4253/CIS 6930 FPGA System Design
Modeling of Combinational Circuits

Hao Zheng
Dept of Comp Sci & Eng
USF
Reading

→ P. Chu, FPGA Prototyping by VHDL Examples
  → Chapter 3, RT-level combinational circuit
  → Sections 3.1 - 3.2, 3.5 - 3.7.

→ XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices
  → Chapter 3 and 7
library ieee;
use ieee.std_logic_1164.all;

entity entity_name is
  port declarations
end [entity] entity_name;

ARCHITECTURE architecture_name OF entity_name IS
  Signal & component declarations
BEGIN
  Concurrent statements
END [ARCHITECTURE] architecture_name;
Concurrent Statements

→ Simple concurrent signal assignment
  →  \( z \leq a \text{ xor } b \)

→ Conditional signal assignment (when-else)

→ selected concurrent signal assignment (with-select-when)

→ Process statements
  → To be covered later
VHDL Modeling Styles

VHDL Descriptions

- dataflow
  - Concurrent statements

- structural
  - Components and interconnects

- behavioral
  - Sequential statements
    - Registers
    - State machines
    - Instruction decoders

• Testbenches

Subset most suitable for synthesis
Combinational Circuit Building Blocks
Fixed Shifters & Rotators
**Fixed Logical Shift Right in VHDL**

```vhdl
SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);

C <= A srl 1;
C <= '0' & A(3 downto 1);
```

**Note:**
- `srl` stands for **s**hift **r**ight.
- The diagram illustrates the shift operation on the signal `A` to obtain `C`.
**Fixed Arithmetic Shift Right in VHDL**

```
SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);

sra: arithmetic shift left

C <= A sra 1;
c <= A(3) & A(3 downto 1);
```
Fixed Rotation in VHDL

SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);

rol: rotation to left

C <= A rol 1
Logic Gates
Basic Gates – AND, OR, NOT

(a) AND gates

(b) OR gates

(c) NOT gate
Basic Gates – NAND, NOR

(a) NAND gates

(b) NOR gates
Basic Gates – XOR

(a) Truth table

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f = x_1 \oplus x_2$</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>

(b) Graphical symbol

(c) Sum-of-products implementation
Basic Gates – XNOR

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f = x_1 \oplus x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Truth table

(b) Graphical symbol

(c) Sum-of-products implementation
1-Bit Full Adder
1-Bit Full Adder

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fa1b IS
  PORT( x : IN STD_LOGIC ;
        y : IN STD_LOGIC ;
        cin : IN STD_LOGIC ;
        s : OUT STD_LOGIC ;
        cout : OUT STD_LOGIC ) ;
END fa1b;
1-Bit Full Adder

ARCHITECTURE dataflow OF fa1b IS
BEGIN
    s <= x XOR y XOR cin;
    cout <= (x AND y) OR (cin AND x) OR (cin AND y);
END dataflow;
Logic Operators

- Logic operators:
  - and
  - or
  - nand
  - nor
  - xor
  - not
  - xnor

- Logic operators precedence:
  - Highest:
    - not
  - and
  - or
  - nand
  - nor
  - xor
  - xnor

  Lowest
No Implied Precedence

Wanted: \( y = ab + cd \)

Incorrect
\[
y \leq a \text{ and } b \text{ or } c \text{ and } d;
\]
equivalent to
\[
y \leq ((a \text{ and } b) \text{ or } c) \text{ and } d;
\]
equivalent to
\[
y = (ab + c)d
\]

Correct
\[
y \leq (a \text{ and } b) \text{ or } (c \text{ and } d);
\]
Modeling Routing Structures with Conditional Concurrent Signal Assignment (when-else)
2-to-1 Multiplexer

(a) Graphical symbol

(b) Truth table
2-to-1 Multiplexer

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux2to1 IS
  PORT( w0, w1, sel : IN STD_LOGIC;
       f : OUT STD_LOGIC);
END mux2to1;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
  f <= w0 WHEN sel = '0' ELSE w1;
END dataflow;
Conditional Concurrent Signal Assignment

target_signal <= value1 when condition1 else value2 when condition2 else 

. . .

value_{N+1} when condition_{N+1} else value_N;

→ Branches are evaluated one by one from top to bottom.
→ Induces priority among branches
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux_cascade IS
  PORT (w1, w2, w3 : IN STD_LOGIC;
        s1, s2        : IN  STD_LOGIC;
        f             : OUT STD_LOGIC ) ;
END mux_cascade ;

ARCHITECTURE dataflow OF mux_cascade IS
BEGIN
  f  <= w1 WHEN s1 = '1' ELSE w2 WHEN s2 = '1' ELSE w3;
END dataflow ;
Cascade of Multiplexers

Notice the priority of selection.
Conditional Concurrent Signal Assignment

target_signal <= value1 \text{ when } condition1 \text{ else } \\
value2 \text{ when } condition2 \text{ else } \\
\ldots \\
value_{N+1} \text{ when } condition_{N+1} \text{ else } \\
value_{N};
More Operators

- Relational operators:
  - equality: =
  - non-equality: /=
  - less than: <
  - less than or equal to: <=
  - greater than: >
  - greater than or equal to: >=

- Logic and relational operators precedence:
  - Highest: not
  - Next:
    - equality: =
    - non-equality: /=
    - less than: <
    - less than or equal to: <=
    - greater than: >
    - greater than or equal to: >=
  - Lowest:
    - logical AND: and
    - logical OR: or
    - logical NAND: nand
    - logical NOR: nor
    - logical XOR: xor
    - logical XNOR: xnor
Precedence of Logic and Relational Operators

Comparison \[ a = bc \]

Incorrect

... when \( a = b \) and \( c \) else ...

equivalent to

... when \((a = b) \ and \ c\) else ...

Correct

... when \( a = (b \ and \ c) \) else ...
Modeling Routing Structures
with
Selected Concurrent Signal Assignment (with-select-when)
4-to-1 Multiplexer

(a) Graphic symbol

(b) Truth table

No priority, and choices are disjoint.
A 4-to-1 Multiplexer

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux4to1 IS
    PORT( w0, w1, w2, w3 : IN STD_LOGIC;
          s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
          f : OUT STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE dataflow OF mux4to1 IS
BEGIN
    WITH s SELECT
    f <= w0 WHEN "00",
         w1 WHEN "01",
         w2 WHEN "10",
         w3 WHEN OTHERS;
END dataflow;

default condition
Selected Concurrent Signal Assignment

```vhdl
with choice_expression select
  target <= expression1 when choices_1,
               expression2 when choices_2,
               . . .
               expressionN when choices_N;
```

All choices are mutually exclusive and cover all values of `choice_expression`. 
with choice_expression select
    target <= expression1 when choices_1,
    expression2 when choices_2,
    . . .
    expressionN when choices_N;
Formats of Choices

- **when** $Expr$

- **when** $Expr_1 \mid \ldots \mid Expr_N$
  - this branch is taken if any of $Expr_x$ matches *choice_expression*

- **when** others
Formats of Choices - Example

```with sel select
    y <= a when "000",
    c when "001" | "111",
    d when others;
```
Decoders
## 2-to-4 Decoder

(a) Truth table

<table>
<thead>
<tr>
<th>$En$</th>
<th>$w_1$</th>
<th>$w_0$</th>
<th>$y_3$</th>
<th>$y_2$</th>
<th>$y_1$</th>
<th>$y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Graphical symbol
-- LIBRARY not shown
ENTITY dec2to4 IS
PORT ( w : IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
      En : IN STD_LOGIC ;
      y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
  SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
  Enw <= En & w ;
  WITH Enw SELECT
  y <= "0001" WHEN "100",
       "0010" WHEN "101",
       "0100" WHEN "110",
       "1000" WHEN "111",
       "0000" WHEN OTHERS ;
END dataflow ;
Encoders
Priority Encoder

\[
\begin{array}{cccc}
\[w_0\] & \[w_1\] & \[w_2\] & \[w_3\] \\
\[y_0\] & \[y_1\] & \[z\] \\
\end{array}
\]

<table>
<thead>
<tr>
<th>[w_3]</th>
<th>[w_2]</th>
<th>[w_1]</th>
<th>[w_0]</th>
<th>[y_1]</th>
<th>[y_0]</th>
<th>[z]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

VHDL code for a Priority Encoder

-- library not shown
ENTITY priority IS
  PORT ( w : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
         y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         z : OUT STD_LOGIC ) ;
END priority ;

ARCHITECTURE dataflow OF priority IS
BEGIN
  y <= "11" when w(3) = '1' else
       "10" when w(2) = '1' else
       "01" when w(1) = '1' else
       "00" when others;

  z <= '0' when w = "0000" else
       '1' when others;
END dataflow ;
Adders
16-bit Unsigned Adder

\[ S = X + Y \]
Operations on Unsigned Numbers

For operations on \texttt{unsigned} numbers

\texttt{USE}

\texttt{ieee.numeric_std.all}

and

signals of the type \texttt{UNSIGNED}

and

conversion functions \texttt{std\_logic\_vector()}, \texttt{unsigned()}

\texttt{OR USE}

\texttt{ieee.std\_logic\_unsigned.all}

and

signals of the type \texttt{STD\_LOGIC\_VECTOR}
16-bit Unsigned Adder

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;--non-IEEE standard

ENTITY adder16 IS
  PORT( Cin  : IN  STD_LOGIC ;
       X    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
       Y    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
       S    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0) ;
       Cout : OUT STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE dataflow OF adder16 IS
  SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
  Sum  <= ('0' & X) + Y + Cin ;
  S    <= Sum(15 DOWNTO 0) ;
  Cout <= Sum(16) ;
END dataflow ;
Addition of Unsigned Numbers (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all; -- IEEE standard

ENTITY adder16 IS
  PORT( Cin : IN STD_LOGIC;
       X, Y : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
       S, Cout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
  )
END adder16;
ARCHITECTURE dataflow OF adder16 IS
  SIGNAL Xu, Yu : UNSIGNED(15 DOWNTO 0);
  SIGNAL Su : UNSIGNED(16 DOWNTO 0);
BEGIN
  Xu <= unsigned(X);
  Yu <= unsigned(Y);
  Su <= ('0' & Xu) + Yu + unsigned('0' & Cin);
  S  <= std_logic_vector(Su(15 DOWNTO 0));
  Cout <= Su(16);
END dataflow;

Signed and unsigned are arrays of std_logic.
Operations on Signed Numbers

For operations on signed numbers

• Either use
  
  \texttt{ieee.numeric\_std.all},
  
signals of the type \texttt{SIGNED}, and
conversion \texttt{std\_logic\_vector()}, \texttt{signed()}

• Or use
  
  \texttt{ieee.std\_logic\_signed.all}, and
signal type \texttt{STD\_LOGIC\_VECTOR}
Signed/Unsigned Types in numeric_std

→ Behave exactly like

    std_logic_vector

→ They determine whether a given vector should be treated as a signed or unsigned number.

→ Prefer to use

    ieee.numeric_std.all;

→ Use either numeric_std or std_logic_unsigned (or signed).
   → Do NOT mix them together.
Multipliers
Unsigned vs. Signed Multiplication

In Xilinx, a multiplier can be implemented either in a DSP or CLB
8x8-bit Unsigned Multiplier

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

entity mult8b is
  port(...);
end mult8b;

architecture arch of mult8b is
begin
  c <= a * b;
end arch;
8x8-bit Signed Multiplier

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

entity mult8b is
    port(...);
end mult8b;

architecture arch of mult8b is
begin
    c <= a * b;
end arch;
```
Signed/Unsigned Multiplication

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multiply is
  port( a : in STD_LOGIC_VECTOR(7 downto 0);
        b : in STD_LOGIC_VECTOR(7 downto 0);
        cu : out STD_LOGIC_VECTOR(15 downto 0);
        cs : out STD_LOGIC_VECTOR(15 downto 0));
end multiply;

architecture dataflow of multiply is begin

  -- signed multiplication
  cs <= std_logic_vector(signed(a)*signed(b));

  -- unsigned multiplication
  cu <= std_logic_vector(unsigned(a)*unsigned(b));
end dataflow;
Multiplication with Constants

If either $A$ or $B$ in $A \times B$ is a constant, more efficient implementation with shifts and additions.

$$A \times 9$$

can be implemented as

$$A \ll 3 + A$$
## Operators in `numeric_std` Package

<table>
<thead>
<tr>
<th>overloaded operator</th>
<th>description</th>
<th>data type of operand a</th>
<th>data type of operand b</th>
<th>data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs a - a</td>
<td>absolute value negation</td>
<td>signed</td>
<td></td>
<td>signed</td>
</tr>
<tr>
<td>a * b</td>
<td>arithmetic</td>
<td>unsigned</td>
<td>unsigned, natural</td>
<td>unsigned</td>
</tr>
<tr>
<td>a / b</td>
<td>arithmetic</td>
<td>unsigned</td>
<td>unsigned, natural</td>
<td>unsigned</td>
</tr>
<tr>
<td>a mod b</td>
<td>arithmetic</td>
<td>unsigned, natural</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
<tr>
<td>a rem b</td>
<td>arithmetic</td>
<td>signed</td>
<td>signed, integer</td>
<td>signed</td>
</tr>
<tr>
<td>a + b</td>
<td></td>
<td>signed, integer</td>
<td></td>
<td>signed</td>
</tr>
<tr>
<td>a - b</td>
<td></td>
<td></td>
<td></td>
<td>signed</td>
</tr>
<tr>
<td>a = b</td>
<td>relational</td>
<td>unsigned</td>
<td>unsigned, natural</td>
<td>boolean</td>
</tr>
<tr>
<td>a /= b</td>
<td>relational</td>
<td>unsigned</td>
<td>unsigned, natural</td>
<td>boolean</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>relational</td>
<td>unsigned, natural</td>
<td>unsigned</td>
<td>boolean</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>relational</td>
<td>signed</td>
<td>signed, integer</td>
<td>boolean</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>relational</td>
<td>signed, integer</td>
<td>signed</td>
<td>boolean</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>relational</td>
<td>signed, integer</td>
<td>signed</td>
<td>boolean</td>
</tr>
</tbody>
</table>
Parameterized Models
Design Reuse

→ How to design for the 32-bit problem below?

\[ O = A + B + C \]

→ Create a new 32-bit adder
   ➔ waste of effort

→ Reuse previously designed adder
   ➔ but it is 16-bit
3.5.2 Generics

VHDL provides a construct, known as a **generic**, to pass information into an entity and component. Since a generic cannot be modified inside the architecture, it functions somewhat like a constant. A generic is declared inside an entity declaration, just before the port declaration:

```
type generic_name is array of data_type;
```

```
generic (generic_name: data_type := default_value;
          generic_name: data_type := default_value;
          ...
          generic_name: data_type := default_value)
```

```
port (port_name: mode data_type;
      ...
      );
```

For example, the previous adder code can be modified to use the adder width as a generic, as shown in Listing 3.11.

**Listing 3.11** Adder using a generic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
  generic(N : integer := 4);
  port(
    a, b : in std_logic_vector(N - 1 downto 0);
    cout : out std_logic;
    sum : out std_logic_vector(N - 1 downto 0)
  );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
  signal a_ext, b_ext, sum_ext : unsigned(N downto 0);
begin
  a_ext <= unsigned('0' & a);
  b_ext <= unsigned('0' & b);
  sum_ext <= a_ext + b_ext;
  sum <= std_logic_vector(sum_ext(N - 1 downto 0));
  cout <= sum_ext(N);
end arch
```
Instances of Generic Models

-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(arch)
  generic map(N=>8)
  port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));

-- instantiate 16-bit adder
adder_16_unit: work.gen_add_w_carry(arch)
  generic map(N=>16)
  port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));

-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
  port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
A Word on Generics

- Generics are typically **integer** values
  - In this class, the entity inputs and outputs should be `std_logic` or `std_logic_vector`.
  - But the generics should be **integer**.
- Generics are given a default value
  - `GENERIC ( N : INTEGER := 16 ) ;`
  - This value can be overwritten when entity is instantiated as a component
- Generics are very useful when instantiating an often-used component
  - Need a 32-bit register in one place, and 16-bit register in another
  - Can use the same generic code, just configure them differently
Constants – Make Code More Readable

Syntax:

```
class constant name : type := value;
```

Examples:

```
constant init_val : STD_LOGIC_VECTOR(3 downto 0) := "0100";
class constant ANDA_EXT : STD_LOGIC_VECTOR(7 downto 0) := X"B4";
class constant counter_width : INTEGER := 16;
class constant buffer_address : INTEGER := x"FFFE";
class constant clk_period : TIME := 20 ns;
class constant strobe_period : TIME := 333.333 ms;
```
Constants vs Generics

→ Constants:
  → Create symbolic names
  → Make code more readable
  → Declared in packages, entity, or architecture.
  → Cannot create generic designs: still need two design entities for Adder_8b and Adder_32b.

→ Generics:
  → Can be passed through design hierarchy through component instantiation
  → Used for creating generic designs: a single design entity Adder for Adder_8b and Adder_32b.
Binary to BCD Conversion
Shift and Add-3 (Double-Dabble)

```c
for(i=0; i<8; i++) {
    // add 3 to a column if it is >= 5
    for each column
        if (column >= 5)
            column += 3;
    // shift binary digits left 1
    Hundred << 1;
    Hundreds[0] = Tens[3];
    Tens << 1;
    Tens[0] = Ones[3];
    Ones << 1;
    Ones[0] = Binary[7];
    Binary << 1;
}
```
Shift and Add-3 (Double-Dabble)

1. If the binary value in any of the BCD columns is 5 or greater, add 3 to that value in that BCD column.
2. Shift the binary number left one bit.
3. If 8 shifts have taken place, the BCD number is in the Hundreds, Tens, and Ones column. Terminate
4. Otherwise, go to 1.

Example:

<table>
<thead>
<tr>
<th>Hundreds</th>
<th>Tens</th>
<th>Ones</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>11110011</td>
</tr>
</tbody>
</table>
Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
</tbody>
</table>

...
### Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100010</td>
<td>0&lt;&lt;1</td>
</tr>
</tbody>
</table>
### Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>100010</td>
<td>&lt;&lt; 1</td>
</tr>
</tbody>
</table>
# Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0101</td>
<td>00010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>1000</td>
<td>00010</td>
<td>+3</td>
</tr>
</tbody>
</table>
## Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0101</td>
<td>00010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>1000</td>
<td>00010</td>
<td>+3</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>10</td>
<td>&lt;&lt; 1</td>
</tr>
</tbody>
</table>
# Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100’s</th>
<th>10’s</th>
<th>1’s</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>10100010</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>100010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0101</td>
<td>00010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>1000</td>
<td>00010</td>
<td>+3</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>10</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>1000</td>
<td>0001</td>
<td>0</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>0000</td>
<td>1011</td>
<td>0001</td>
<td>0</td>
<td>+3</td>
</tr>
</tbody>
</table>
### Shift and Add-3 (Double-Dabble)

<table>
<thead>
<tr>
<th>100's</th>
<th>10's</th>
<th>1's</th>
<th>Binary</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>00100010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>0100010</td>
<td>&lt;&lt; #1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>100010</td>
<td>&lt;&lt; #2</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td></td>
<td>00010</td>
<td>&lt;&lt; #3</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td></td>
<td>add 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>00000010</td>
<td>&lt;&lt; #4</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>0000010</td>
<td>&lt;&lt; #5</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>000010</td>
<td>&lt;&lt; #6</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td>00010</td>
<td>&lt;&lt; #7</td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td>add 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td></td>
<td>0010</td>
<td>&lt;&lt; #8</td>
</tr>
</tbody>
</table>

Goto wiki for more information and VHDL implementation
Summary

→ More concurrent statements for DF modeling
  → describing routing structures
→ Modeling of basic combinational circuit blocks
  → Adders, multipliers, muxes, encoder/decoder
→ Generic design modeling
  → Using VHDL generics
Backup
Comparators
2-bit Number Comparator

\[ A = B \quad \text{AeqB} \]

A

B

2

2
4-bit Unsigned Number Comparator

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity compare is
  port( A, B : in STD_LOGIC_VECTOR(1 downto 0);
        AeqB : out STD_LOGIC);
end compare;

architecture dataflow of compare is
begin
  AeqB <= '1' when A = B else '0';
end dataflow;
4-bit Unsigned Number Comparator

library ieee;
use ieee.std_logic_1164.all;

entity compare is
  port( A, B : in STD_LOGIC_VECTOR(1 downto 0);
       AeqB : out STD_LOGIC );
end compare ;

-- Create a different model?
architecture dataflow of compare is
begin

end dataflow ;
4-bit Signed Number Comparator

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity compare is
  port( A, B : in STD_LOGIC_VECTOR(1 downto 0);
        AeqB : out STD_LOGIC);
end compare ;

architecture dataflow of compare is begin
  AeqB <= '1' when A = B else '0';
end dataflow ;
Hexadecimal to 7-Segment Display
7-Segment Display

To illuminate a segment, the corresponding control signal should be driven low.
7-Segment Display

To illuminate a segment, the corresponding control signal should be driven low – A = ‘0,’ ..., F = ‘0,’ G = ‘1’
To illuminate a segment, the corresponding control signal should be driven low – A = ‘0,’ ..., E = ‘1’, F = ‘0’, G = ‘0’
# Hex to 7-Segment

<table>
<thead>
<tr>
<th>Hex Input</th>
<th>7-Segment Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 (0)</td>
<td>10000000</td>
</tr>
<tr>
<td>0001 (1)</td>
<td>11110001</td>
</tr>
<tr>
<td>0010 (2)</td>
<td>01001000</td>
</tr>
<tr>
<td>0011 (3)</td>
<td>01100000</td>
</tr>
<tr>
<td>0100 (4)</td>
<td>00110001</td>
</tr>
<tr>
<td>0101 (5)</td>
<td>00100100</td>
</tr>
<tr>
<td>0110 (6)</td>
<td>00000100</td>
</tr>
<tr>
<td>0111 (7)</td>
<td>11110000</td>
</tr>
<tr>
<td>1000 (8)</td>
<td>00000000</td>
</tr>
<tr>
<td>1001 (9)</td>
<td>00100000</td>
</tr>
<tr>
<td>1010 (A)</td>
<td>00010000</td>
</tr>
</tbody>
</table>
7-Segment Display

All four displays share common segment control signals.
Only one display can be illuminated at a time when signal ANx is driven high.
7-Segment Display Controller

- hex2sseg
- hex2sseg
- hex2sseg
- hex2sseg
- disp_mux
- clk
- in0
- in2
- in2
- in3
- sseg
- an
1. Right shift bcd1, with the LSB shifting to the MSB of bcd0.
2. Right shift bcd0, with the LSB shifting to the MSB of bin.
3. If bcd0 is now > 4, subtract 3
4. repeat steps 1-3, 7 times.
### BCD to Binary Conversion

<table>
<thead>
<tr>
<th>Iteration:</th>
<th>BCD</th>
<th>bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01000001</td>
<td>0000000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>0100001</td>
<td>0000000</td>
</tr>
<tr>
<td>1 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010000</td>
<td>1000000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>0 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00101</td>
<td>0100000</td>
</tr>
<tr>
<td>8 &gt; 4, -3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1010000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>001</td>
<td>0101000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>000</td>
<td>1010100</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0101010</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ✖️ 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ✖️ 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

- **BCD**: 0100, 0010
- **Binary**: 0101010

The conversion process is shown step-by-step, illustrating the transformation of BCD to binary.
8-bit Variable Rotator Left

A

A <<< B

B

C

3

8

8

8
Tri-State Buffers
Tri-State Buffers

(a) A tri-state buffer

(b) Equivalent circuit

(c) Truth table
Four types of Tri-state Buffers

(a) 

(b) 

(c) 

(d)
Tri-state Buffer – Example (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity tri_state is
  port( ena, input : IN STD_LOGIC;
        output : OUT STD_LOGIC);
end tri_state;

architecture dataflow of tri_state is
begin
  output <= input when (ena = '1') else 'Z';
end dataflow;
ROM
ROM 8x16 example (1)
ROM 8x16 example (2)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity rom is
  port (Addr : in STD_LOGIC_VECTOR(2 downto 0);
       Dout : out STD_LOGIC_VECTOR(15 downto 0));
end rom;

-- architecture body is defined on the next slide
architecture dataflow of rom is
  signal temp: integer range 0 to 7;
  type vector_array is array(0 to 7) of
    std_logic_vector(15 downto 0);
  constant memory : vector_array := ( X"800A",
                                      X"D459",
                                      X"A870",
                                      X"7853",
                                      X"650D",
                                      X"642F",
                                      X"F742",
                                      X"F548");
begin
  temp <= to_integer(unsigned(Addr));
  Dout <= memory(temp);
end dataflow;