ECE 545
Lecture 5

Data Flow Modeling of Combinational Logic
Required reading

- P. Chu, *RTL Hardware Design using VHDL*

*Chapter 4, Concurrent Signal Assignment Statements of VHDL*
Dataflow VHDL Design Style
VHDL Design Styles

- **dataflow**
  - Concurrent statements

- **structural**
  - Components and interconnects

- **behavioral (sequential)**
  - Sequential statements
    - Registers
    - State machines
    - Instruction decoders

- Subset most suitable for synthesis

- • Testbenches
Synthesizable VHDL

Dataflow VHDL Design Style

\[\rightarrow\] VHDL code synthesizable

Dataflow VHDL Design Style

\[\times\] VHDL code synthesizable
Register Transfer Level (RTL) Design Description

Today’s Topic

Combinational Logic

Registers

Combinational Logic
Data-Flow VHDL

Concurrent Statements

• concurrent signal assignment
  \( \leftarrow \rightarrow \)

• conditional concurrent signal assignment
  (when-else)

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

• generate scheme for equations
  (for-generate)
Data-flow VHDL

**Major instructions**

**Concurrent statements**

- concurrent signal assignment \( \Leftarrow \)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)
Data-flow VHDL: Example
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( x : IN STD_LOGIC ;
          y : IN STD_LOGIC ;
          cin : IN STD_LOGIC ;
          s : OUT STD_LOGIC ;
          cout : OUT STD_LOGIC ) ;
END fulladd ;
ARCHITECTURE dataflow OF fulladd 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

only in VHDL-93 or later
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**

\( y \leq (a \text{ and } b) \text{ or } (c \text{ and } d) \);
• E.g.,
  status <= '1';
  even <= (p1 and p2) or (p3 and p4);
  arith_out <= a + b + c - 1;

• Implementation of last statement
Signal assignment statement with a closed feedback loop

- a signal appears in both sides of a concurrent assignment statement
- E.g.,
  \[ q \leq ((\textbf{not} \ q) \ \textbf{and} \ (\textbf{not} \ en)) \ \textbf{or} \ (d \ \textbf{and} \ en); \]
- Syntactically correct
- Form a closed feedback loop
- Should be avoided
Data-flow VHDL

**Major instructions**

**Concurrent statements**

- concurrent signal assignment  \( (\Leftarrow) \)
- **conditional concurrent signal assignment**  \( \text{(when-else)} \)
- selected concurrent signal assignment  \( \text{(with-select-when)} \)
- generate scheme for equations  \( \text{(for-generate)} \)
Conditional concurrent signal assignment

When - Else

target_signal <= value1 when condition1 else value2 when condition2 else . . . valueN-1 when conditionN-1 else valueN;
Most often implied structure

When - Else

target_signal <= value1 when condition1 else value2 when condition2 else 
  . . .
valueN-1 when conditionN-1 else valueN;

<table>
<thead>
<tr>
<th>Value N</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value N-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition N-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value 2</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value N</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target Signal
2-to-1 “abstract” mux

- sel has a data type of boolean
- If sel is true, the input from “T” port is connected to output.
- If sel is false, the input from “F” port is connected to output.
signal_name <= value_expr_1 when boolean_expr_1 else value_expr_2;
signal_name <= value_expr_1 when boolean_expr_1 else value_expr_2 when boolean_expr_2 else value_expr_3 when boolean_expr_3 else value_expr_4;

RTL Hardware Design  Chapter 4  21
• E.g.,

```vhdl
signal a, b, y: std_logic;

y <= '0' when a = b else '1';
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b</td>
<td>a=b</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
</tr>
</tbody>
</table>
• E.g.,

```vhdl
signal a, b, c, x, y, r: std_logic;

r <= a when x = y else
    b when x > y else
    c;
```

![Diagram](image)
E.g.,

```vhdl
signal a, b, r: unsigned(7 downto 0);
signal x, y: unsigned(3 downto 0);

r <= a + b when x + y > 1 else
    a - b - 1 when x > y and y != 0 else
    a + 1;
```

```vhdl```

![ VHDL code for a hardware design example ]

---

RTL Hardware Design

Chapter 4
Signed and Unsigned Types

Behave exactly like

STD_LOGIC_VECTOR

plus, they determine whether a given vector should be treated as a signed or unsigned number.

Require

USE ieee.numeric_std.all;
Operators

- Relational operators

\[ = \quad /= \quad < \quad <= \quad > \quad >= \]

- Logic and relational operators precedence

<table>
<thead>
<tr>
<th>Highest</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
</tr>
<tr>
<td>=</td>
</tr>
<tr>
<td>/=</td>
</tr>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>nand</td>
</tr>
<tr>
<td>nor</td>
</tr>
<tr>
<td>xor</td>
</tr>
<tr>
<td>xnor</td>
</tr>
</tbody>
</table>

Lowest
Priority of logic and relational operators

compare \ a = bc

**Incorrect**

\[ \text{... when } a = b \text{ and } c \text{ else ...} \]

equivalent to

\[ \text{... when } (a = b) \text{ and } c \text{ else ...} \]

**Correct**

\[ \text{... when } a = (b \text{ and } c) \text{ else ...} \]
## VHDL operators

<table>
<thead>
<tr>
<th>Highest precedence</th>
<th>Operator Class</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Miscellaneous</td>
<td>**, ABS, NOT</td>
</tr>
<tr>
<td></td>
<td>Multiplying</td>
<td>*, /, MOD, REM</td>
</tr>
<tr>
<td></td>
<td>Sign</td>
<td>+, −</td>
</tr>
<tr>
<td></td>
<td>Adding</td>
<td>+, −, &amp;</td>
</tr>
<tr>
<td></td>
<td>Shift</td>
<td>SLL, SRL, SLA, SRA, ROL, ROR</td>
</tr>
<tr>
<td>Lowest precedence</td>
<td>Relational</td>
<td>=, /=, &lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td></td>
<td>Logical</td>
<td>AND, OR, NAND, NOR, XOR, XNOR</td>
</tr>
</tbody>
</table>
Data-flow VHDL

Major instructions

Concurrent statements

- concurrent signal assignment  (\(\leftarrow\))
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)
Selected concurrent signal assignment

With –Select-When

with choice_expression select
    target_signal <= expression1 when choices_1,
    expression2 when choices_2,
    . . .
    expressionN when choices_N;
Most Often Implied Structure

*With –Select-When*

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

Diagram:
- `choice_expression` arrow pointing to `target_signal`
- `choices_1`, `choices_2`, ..., `choices_N` as intermediate nodes
- `expression1`, `expression2`, ..., `expressionN` as nodes connected to the `choices` nodes

Allowed formats of \textit{choices}_k

\begin{itemize}
  \item WHEN value
  \item WHEN value\_1 \textbar value\_2 \textbar \ldots \textbar value \_N
  \item WHEN OTHERS
\end{itemize}
Allowed formats of $choice_k$ - example

WITH sel SELECT

    y <= a WHEN "000",
    c WHEN "001" | "111",
    d WHEN OTHERS;
Syntax

• Simplified syntax:

```vhdl
with select_expression select
signal_name <=
  value_expr_1 when choice_1,
  value_expr_2 when choice_2,
  value_expr_3 when choice_3,
  ...
  value_expr_n when choice_n;
```
• **select_expression**
  – Discrete type or 1-D array
  – With finite possible values

• **choice_i**
  – A value of the data type

• **Choices must be**
  – mutually exclusive
  – all inclusive
  – **others** can be used as last choice_i
E.g., 4-to-1 mux

```vhdl
architecture sel_arch of mux4 is
begin
    with s select
        x <= a when "00",
            b when "01",
            c when "10",
            d when others;
end sel_arch;
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>x</td>
</tr>
<tr>
<td>00</td>
<td>a</td>
</tr>
<tr>
<td>01</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>c</td>
</tr>
<tr>
<td>11</td>
<td>d</td>
</tr>
</tbody>
</table>
• Can “11” be used to replace others?

```vhd
with s select
  x <= a when "00",
       b when "01",
       c when "10",
       d when "11";
```
E.g., 2-to-$2^2$ binary decoder

```vhdl
architecture sel_arch of decoder4 is
begin
    with sel select
    begin
        x <= "0001" when "00",
             "0010" when "01",
             "0100" when "10",
             "1000" when others;
    end sel_arch;
end
```

<table>
<thead>
<tr>
<th>input s</th>
<th>output x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0001</td>
</tr>
<tr>
<td>0 1</td>
<td>0010</td>
</tr>
<tr>
<td>1 0</td>
<td>0100</td>
</tr>
<tr>
<td>1 1</td>
<td>1000</td>
</tr>
</tbody>
</table>
E.g., 4-to-2 priority encoder

```vhdl
architecture sel_arch of prio_encoder42 is
begin
    with r select
        code <= "11" when "1000"|"1001"|"1010"|"1011" |
               "1100"|"1101"|"1110"|"1111",
        "10" when "0100"|"0101"|"0110"|"0111",
        "01" when "0010"|"0011",
        "00" when others;
    active <= r(3) or r(2) or r(1) or r(0);
end sel_arch;
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td>r</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>0 1</td>
<td>10</td>
</tr>
<tr>
<td>0 0 1</td>
<td>01</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>00</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>00</td>
</tr>
</tbody>
</table>
• Can we use ‘-’?

```vhdl
with a select
  x <= "11" when "1---",
      "10" when "01--",
      "01" when "001-",
      "00" when others;
```
E.g., simple ALU

```vhdl
architecture sel_arch of simple_alu is
  signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
  inc <= std_logic_vector(signed(src0)+1);
  sum <= std_logic_vector(signed(src0)+signed(src1));
  diff <= std_logic_vector(signed(src0)-signed(src1));
  with ctrl select
  begin
    result <= inc when "000"|"001"|"010"|"011",
              sum when "100",
              diff when "101",
              src0 and src1 when "110",
              src0 or src1 when others;
  end;
end sel_arch;
```

<table>
<thead>
<tr>
<th>ctrl</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>src0 + 1</td>
</tr>
<tr>
<td>010</td>
<td>src0 + src1</td>
</tr>
<tr>
<td>011</td>
<td>src0 - src1</td>
</tr>
<tr>
<td>100</td>
<td>src0 and src1</td>
</tr>
<tr>
<td>111</td>
<td>src0 or src1</td>
</tr>
</tbody>
</table>
libraryieee;
useieee.std_logic_1164.all;
entity truth_table is
  port(
    a,b: in std_logic;
    y: out std_logic
  );
end truth_table;
architecturea of truth_table is
  signal tmp: std_logic_vector(1 downto 0);
begina
  tmp <= a & b;
  with tmp select
    y <= '0' when "00",
       '1' when "01",
       '1' when "10",
       '1' when others; -- "11"
end a;
Conceptual implementation

• Achieved by a multiplexing circuit
• Abstract (k+1)-to-1 multiplexer
  – sel is with a data type of (k+1) values: c0, c1, c2, ..., ck
- `select_expression` is with a data type of 5 values: `c0, c1, c2, c3, c4`

```vhdl
with select_expression select
    sig <= value_expr_0 when c0,
         value_expr_1 when c1,
         value_expr_n when others;
```
• E.g.,

```vhdl
signal a, b, r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);

with s select
  r <= a+1 when "11",
       a-b-1 when "10",
       a+b  when others;
```

![Diagram of a circuit with signals a, b, and s connected to a multiplexer and output r.]
3. Conditional vs. selected signal assignment

- Conversion between conditional vs. selected signal assignment
- Comparison
From selected assignment to conditional assignment

```vhdl
with sel select
  sig <= value_expr_0 when c0,
       value_expr_1 when c1|c3|c5,
       value_expr_2 when c2|c4,
       value_expr_n when others;
```

```vhdl
sig <=
  value_expr_0 when (sel=c0) else
  value_expr_1 when (sel=c1) or (sel=c3) or (sel=c5) else
  value_expr_2 when (sel=c2) or (sel=c4) else
  value_expr_n;
```
From conditional assignment to selected assignment

```
sig <= value_expr_0 when bool_exp_0 else value_expr_1 when bool_exp_1 else value_expr_2 when bool_exp_2 else value_expr_n;

sel(2) <= '1' when bool_exp_0 else '0';
sel(1) <= '1' when bool_exp_1 else '0';
sel(0) <= '1' when bool_exp_2 else '0';
with sel select
    sig <= value_expr_0 when "100" | "101" | "110" | "111",
          value_expr_1 when "010" | "011",
          value_expr_2 when "001",
          value_expr_n when others;
```
Comparison

• Selected signal assignment:
  – good match for a circuit described by a functional table
  – E.g., binary decoder, multiplexer
  – Less effective when an input pattern is given a preferential treatment
• Conditional signal assignment:
  – good match for a circuit that needs to give preferential treatment for certain conditions or to prioritize the operations
  – E.g., priority encoder
  – Can handle complicated conditions. e.g.,

```python
pc_next <=
    pc_reg + offset when (state=jump and a=b) else
    pc_reg + 1 when (state=skip and flag='1') else

...```

– May “over-specify” for a functional table based circuit.

– E.g., mux

\[
x \leftarrow \begin{align*}
a & \text{ when } (s=\text{"00"}) \text{ else } \\
b & \text{ when } (s=\text{"01"}) \text{ else } \\
c & \text{ when } (s=\text{"10"}) \text{ else } \\
d & \text{ else } \\
x \leftarrow c & \text{ when } (s=\text{"10"}) \text{ else } \\
a & \text{ when } (s=\text{"00"}) \text{ else } \\
b & \text{ when } (s=\text{"01"}) \text{ else } \\
d & \text{ else } \\
x \leftarrow c & \text{ when } (s=\text{"10"}) \text{ else } \\
b & \text{ when } (s=\text{"01"}) \text{ else } \\
a & \text{ when } (s=\text{"00"}) \text{ else } \\
d & \text{ else }
\end{align*}
\]
MLU Example
MLU Block Diagram
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mlu IS
  PORT(
    NEG_A : IN STD_LOGIC;
    NEG_B : IN STD_LOGIC;
    NEG_Y : IN STD_LOGIC;
    A : IN STD_LOGIC;
    B : IN STD_LOGIC;
    L1 : IN STD_LOGIC;
    L0 : IN STD_LOGIC;
    Y : OUT STD_LOGIC
  );
END mlu;
MLU: Architecture Declarative Section

ARCHITECTURE mlu_dataflow OF mlu IS

SIGNAL  A1 :  STD_LOGIC;
SIGNAL  B1 :  STD_LOGIC;
SIGNAL  Y1 : STD_LOGIC;
SIGNAL  MUX_0 : STD_LOGIC;
SIGNAL  MUX_1 : STD_LOGIC;
SIGNAL  MUX_2 : STD_LOGIC;
SIGNAL  MUX_3 : STD_LOGIC;
SIGNAL  L: STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
  A1<= NOT A WHEN (NEG_A='1') ELSE A;
  B1<= NOT B WHEN (NEG_B='1') ELSE B;
  Y <= NOT Y1 WHEN (NEG_Y='1') ELSE Y1;

  MUX_0 <= A1 AND B1;
  MUX_1 <= A1 OR B1;
  MUX_2 <= A1 XOR B1;
  MUX_3 <= A1 XNOR B1;

  L <= L1 & L0;

  with (L) select
    Y1 <= MUX_0 WHEN "00",
         MUX_1 WHEN "01",
         MUX_2 WHEN "10",
         MUX_3 WHEN OTHERS;

END mlu_dataflow;
Modeling Common Combinational Logic Components Using Dataflow VHDL
Wires and Buses
Signals

SIGNAL a : STD_LOGIC;

SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);
Merging wires and buses

SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(9 DOWNTO 0);
d <= a & b & c;
Splitting buses

SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(9 DOWNTO 0);

a <= d(9 downto 6);
b <= d(5 downto 1);
c <= d(0);
Fixed Shifters & Rotators
Fixed Shift in VHDL

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

A >>= 1

AshiftR <=
Fixed Rotation in VHDL

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

A<<<1

A(3) A(2) A(1) A(0)

ArotL <=
Tri-state Buffer

(a) A tri-state buffer

\[
x \quad \rightarrow \quad f
\]

(b) Equivalent circuit

\[
x \quad \rightarrow \quad f \quad \text{\(e = 0\)}
\]

\[
x \quad \rightarrow \quad f \quad \text{\(e = 1\)}
\]

(c) Truth table

<table>
<thead>
<tr>
<th>(e)</th>
<th>(x)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</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>
Four types of Tri-state Buffers

(a) \[ x \rightarrow e \rightarrow f \]
(b) \[ x \rightarrow e \rightarrow f \]
(c) \[ x \rightarrow f \]
(d) \[ x \rightarrow f \]
Tri-state Buffer – example (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tri_state IS
    PORT ( ena: IN STD_LOGIC;
           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;
Multiplexers
2-to-1 Multiplexer

(a) Graphical symbol

(b) Truth table
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END dataflow ;
Cascade of two multiplexers
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 mux2to1 IS
BEGIN
    f <= w1 WHEN s1 = '1' ELSE
        w2  WHEN s2 = '1' ELSE
        w3 ;
END dataflow ;
4-to-1 Multiplexer

(a) Graphic symbol

(b) Truth table
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;
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 ieee ;
USE ieee.std_logic_1164.all ;

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

\[
\begin{array}{cccc|ccc}
  w_3 & w_2 & w_1 & w_0 & y_1 & y_0 & z \\
  0 & 0 & 0 & 0 & d & d & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 1 \\
  0 & 0 & 1 & x & 0 & 1 & 1 \\
  0 & 1 & x & x & 1 & 0 & 1 \\
  1 & x & x & x & 1 & 1 & 1 \\
\end{array}
\]
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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";
  z <= '0' WHEN w = "0000" ELSE '1';
END dataflow;