ECE 545
Lecture 4
Dataflow Modeling in VHDL

Types of VHDL Description

Concurrent statements
- Dataflow
- Structural
- Behavioral (sequential)

Components and interconnects
- Testbenches
- Registers
- State machines
- Instruction decoders

Subset most suitable for synthesis

Synthesizable VHDL

Dataflow VHDL Description → VHDL code synthesizable
Dataflow VHDL Description → VHDL code synthesizable

Register-Transfer Level (RTL) Design Description

Today’s Topic
- Combinational Logic
- Storage elements

- Use of medium scale-components (adders, multipliers, MUXes, ROMs)
- The designer needs to specify what happens in the circuit in every clock cycle

Data-Flow VHDL

Concurrent Statements
- simple concurrent signal assignment (<=)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)

Required reading

• P. Chu, RTL Hardware Design using VHDL

  Chapter 4, Concurrent Signal Assignment
  Statements of VHDL
  Chapter 6.3, Realization of VHDL Data Types
Simple concurrent signal assignment

\[ \text{target_signal} \leftarrow \text{expression}; \]

Conditional concurrent signal assignment

\[ \text{target_signal} \leftarrow \text{value}_1 \text{ when } \text{condition}_1 \text{ else } \text{value}_2 \text{ when } \text{condition}_2 \text{ else } \ldots \text{value}_{N-1} \text{ when } \text{condition}_{N-1} \text{ else } \text{value}_N; \]

Selected concurrent signal assignment

\[
\begin{align*}
\text{with } \text{choice_expression} \text{ select } \\
\text{target_signal} & \leftarrow \text{expression}_1 \text{ when } \text{choices}_1, \\
& \text{expression}_2 \text{ when } \text{choices}_2, \\
& \ldots \\
& \text{expression}_{N} \text{ when } \text{choices}_N;
\end{align*}
\]

Data-Flow VHDL

Concurrent Statements

- simple concurrent signal assignment \((\leftarrow)\)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)

Signals

\[
\begin{align*}
\text{SIGNAL} \ a & : \text{STD_LOGIC}; \\
\text{SIGNAL} \ b & : \text{STD_LOGIC VECTOR}(7 \text{ DOWNTO} 0);
\end{align*}
\]
Merging wires and buses

\[ d = a \| b \| c \]

Splitting buses

\[ a = d \]

\[ b = d \]

\[ c = d \]

Data-flow VHDL: Example

```vhdl
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;
```
**Data-flow VHDL: Example (2)**

```
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**
  - **in VHDL-93 and later**
    - Highest: `not`
    - Lowest: `and`, `or`, `nand`, `nor`, `xor`, `xnor`

```
eqivalent to

ARCHITECTURE dataflow OF fulladd IS
BEGIN
    cout  <=  (x AND y) OR (cin AND x) OR (cin AND y);
    s       <=   x XOR y XOR cin;
END dataflow;
```

**No Implied Precedence**

Wanted: \( y = ab + cd \)

**Incorrect**

\( y <= a \text{ and } b \text{ or } c \text{ and } d \);

equivalent to

\( y <= ((a \text{ and } b) \text{ or } c) \text{ and } d \);

equivalent to

\( y = (ab + c)d \)

**Correct**

\( y <= (a \text{ and } b) \text{ or } (c \text{ and } d) \);

**Signal assignment statement with a closed feedback loop**

- a signal appears in both sides of a concurrent assignment statement
- E.g.,
  - \( q <= ((\text{not } q) \text{ and } (\text{not } en)) \text{ or } (d \text{ and } en) \);
- Syntactically correct
- Form a closed feedback loop
- Should be avoided

**Data-Flow VHDL**

**Concurrent Statements**

- **simple concurrent signal assignment**
  - `=`
- **conditional concurrent signal assignment**
  - `when-else`
- **selected concurrent signal assignment**
  - `with-select-when`
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;
```

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.

```
<table>
<thead>
<tr>
<th>i1</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>i0</td>
<td>n</td>
</tr>
<tr>
<td>sel</td>
<td></td>
</tr>
</tbody>
</table>
```

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;
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

```vhdl
USE ieee.numeric_std.all;
```

Operators

- Relational operators

  ```vhdl
  = /= < <= > >=
  ```

- Logic and relational operators precedence

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>not, ¬</td>
<td>and, ∧</td>
</tr>
<tr>
<td>or, ∨</td>
<td>nand, NAND</td>
</tr>
<tr>
<td>nor, NOR</td>
<td>xor, XOR</td>
</tr>
<tr>
<td>xnor, XNOR</td>
<td></td>
</tr>
</tbody>
</table>

Priority of logic and relational operators

- `compare a = bc`
- `Incorrect`

  ```vhdl
  ... when a = b and c else ...
  ```

  equivalent to

  ```vhdl
  ... when (a = b) and c else ...
  ```

- `Correct`

  ```vhdl
  ... when a = (b and c) else ...
  ```

Data-Flow VHDL

Concurrent Statements

- simple concurrent signal assignment (=>)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)

Selected concurrent signal assignment

```vhdl
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

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

Allowed formats of `choices_k`

- WHEN value
- WHEN value_1 | value_2 | ... | value N
- WHEN OTHERS

Allowed formats of `choice_k` - example

```vhdl
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**

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

• Can "11" be used to replace others?

```
architecture sel_arch of mux4 is begin
  with a select
  x <= a when "00",
    b when "01",
    c when "10",
    d when "11";
end sel_arch;
```

**E.g., 2-to-2^2 binary decoder**

```
architecture sel_arch of decoder4 is begin
  with sel select
  x <= "0001" when "00",
       "0010" when "01",
       "0100" when "10",
       "1000" when others;
  input  output
  s   x
  00  0001
  01  0010
  10  0100
  11  1000
end sel_arch;
```

**E.g., simple ALU**

```
arbitrary selection of inc
begin
  inc <= std_logic_vector(sign((src0) + 1));
  sum <= std_logic_vector(sign((src0) - (src1)));
  diff <= std_logic_vector(sign((src0) - (src1)));
  with ctrl select
  result <= inc when "000" |
            sum when "001" |
            diff when "010" |
            (src0 and src1) when "100" |
            src0 or src1 when others;
end sel_arch;
```

**E.g., Truth table**

```
library ieee;
use ieee.std_logic_1164.all;
entity truth_table is
  port(a, b : in std_logic;
       y : out std_logic);
end truth_table;
architecture a of truth_table is
  signal tmp : std_logic_vector(1 downto 0);
begin
  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 c: std_logic_vector(1 downto 0);

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

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;
```

From conditional assignment to selected assignment

```vhdl
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;
```

```vhdl
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';
```

```vhdl
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.,
    ```
    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 <= a when (a='00') else
                 b when (a='01') else
               c when (a='10') else
                       d;
    x <= c when (a='10') else
               a when (a='00') else
             b when (a='01') else
                   d;
    x <= c when (a='10') else
               b when (a='01') else
             a when (a='00') else
                   d;
    ```

when-else vs. with-select-when (1)

“when-else” should be used when:
1) there is only one condition (and thus, only one else), as in the 2-to-1 MUX
2) conditions are independent of each other (e.g., they test values of different signals)
3) conditions reflect priority (as in priority encoder); one with the highest priority need to be tested first.

when-else vs. with-select-when (2)

“with-select-when” should be used when there is
1) more than one condition
2) conditions are closely related to each other (e.g., represent different ranges of values of the same signal)
3) all conditions have the same priority (as in the 4-to-1 MUX).

Use of ‘-’

• In conventional logic design
  – ‘-’ as input value: shorthand to make table compact
  – E.g.,
    ```
    | input | output |
    |-------|--------|
    | req   | code  |
    |-------|--------|
    | 1 0 0 | 10     |
    | 1 0 1 | 10     |
    | 1 1 0 | 10     |
    | 1 1 1 | 10     |
    | 0 1 0 | 01     |
    | 0 1 1 | 01     |
    | 0 0 1 | 00     |
    | 0 0 0 | 00     |
    ```

    ```
    | input | output |
    |-------|--------|
    | req   | code  |
    |-------|--------|
    | 1 0 0 | 10     |
    | 1 0 1 | 01     |
    | 1 1 0 | 00     |
    | 1 1 1 | 00     |
    | 0 1 0 | 00     |
    | 0 1 1 | 00     |
    | 0 0 1 | 00     |
    | 0 0 0 | 00     |
    ```
Use '-' in VHDL

- As input value (against our intuition):
  
  ```vhdl
  y <= "10" when req="1--", "01" when req="01--", "00" when req="001", "00";
  ```

- Wrong:
  
  ```vhdl
  with req select
  y <= "10" when "1--", "01" when "01--", "00" when "001", "00" when others;
  ```

- Fix:
  
  ```vhdl
  with req select
  y <= "10" when "100"|"101"|"110"|"111", "00" when "010"|"011", "00" when others;
  ```

- Fix #1:
  
  ```vhdl
  y <= "10" when req(2)="1" else "01" when req(2 downto 1)="01" else "00" when req(2 downto 0)="001" else "00";
  ```

- Fix #2:
  
  ```vhdl
  use ieee.numeric_std.all;
  y <= "10" when std_match(req,"1--") else "01" when std_match(req,"01--") else "00" when std_match(req,"001") else "00";
  ```

- '-' as an output value in VHDL
  
  May work with some software
  
  ```vhdl
  sel <= a & b;
  with sel select
  y <= '0' when "00", '1' when "01", '1' when "10", '-' when others;
  ```

Use and synthesis of 'Z'

- Tri-state buffer:
  
  - Output with "high-impedance"
  - Not a value in Boolean algebra
  - Need special output circuitry (tri-state buffer)
  
  ```vhdl
  oe y
  a_in Z
  0 1 a_in
  ```
• Major application:
  – Bi-directional I/O pins
  – Tri-state bus
• VHDL description:
  \[ y \left\langle \begin{array}{l}
  'Z' \text{ when } oe='1' \\
  a\_in
\end{array} \right\rangle \\
  \]
  \[ \text{'}Z' \text{ cannot be used as input or manipulated} \\
  f \left\langle \begin{array}{l}
  'Z' \text{ and } a
\end{array} \right\rangle \\
  y \left\langle \begin{array}{l}
  \text{data\_a when in\_bus='Z' else } \\
  \text{data\_b}
\end{array} \right\rangle \\
  \]

• Separate tri-state buffer from regular code:
  – Less clear:
    \[
    \text{with sel select} \\
    y \left\langle \begin{array}{l}
    'Z' \text{ when } "00", \\
    '1' \text{ when } "01"|"11", \\
    '0' \text{ when others}; \\
    \end{array} \right\rangle \\
    \[
    \text{– better:} \\
    \text{with sel select} \\
    tmp \left\langle \begin{array}{l}
    '1' \text{ when } "01"|"11", \\
    '0' \text{ when others}; \\
    \end{array} \right\rangle \\
    y \left\langle \begin{array}{l}
    'Z' \text{ when sel='00' else } \\
    \text{tmp}
\end{array} \right\rangle \\
  \]

Bi-directional i/o pins

\[
\text{entity bi\_demo is} \\
\text{port}\text{(bi: inout std\_logic)}; \\
\begin{align*}
\text{begin} \\
\text{sig\_out} & \left\langle \text{output\_expression}; \\
\text{...} & \left\langle \text{expression\_with\_sig\_in}; \\
\text{...} & \left\langle \text{bi} & \left\langle \text{sig\_out when dir='1' else } 'Z'; \\
\text{sig\_in} & \left\langle \text{bi}; \\
\text{...} & \left\langle \text{end}\text{ begin}
\end{align*}
\]

Tri-state bus

sig\_in <= bi when dir='0' else 'Z';
- **Problem with tri-state bus**
  - Difficult to optimize, verify and test
  - Somewhat difficult to design: “parking”, “fighting”

- **Alternative to tri-state bus: mux**

```vhdl
-- binary decoder
with src_select select
  oe <= "0001" when "00",
       "0010" when "01",
       "0100" when "10",
       "1000" when others; — "I1"

-- tri-state buffers
y0 <= i0 when oe(0)="1" else 'Z';
y1 <= i1 when oe(1)="1" else 'Z';
y2 <= i2 when oe(2)="1" else 'Z';
y3 <= i3 when oe(3)="1" else 'Z';
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

```vhdl
with src_select select
data_bus <= i0 when "00",
         i1 when "01",
         i2 when "10",
         i3 when others; — "I1"
```