Concurrent Signal Assignment Statements

This slide set covers the *concurrent signal assignment* statements, which include the *conditional signal assignment* and *selected signal assignment* stmts

Topics include

- Simple signal assignment statement (conditional assign. without a condition)
- Conditional signal assignment statement
- Selected signal assignment statement
- Conditional vs. selected signal assignment

**Simple signal assignment statement**

```vhdl
signal_name <= projected_waveform;
```

For example

```vhdl
-- y changes after a or b changes + 10 ns
y <= a + b + 1 after 10 ns;
```

Timing info ignored in synthesis and \( \delta \)-delay is used for

```vhdl
signal_name <= value_expression;
```
Simple Signal Assignment Statements

Other examples:

status <= '1';

even <= (p1 and p2) or (p3 and p4);

arith_out <= a + b + c - 1;

Implementation of last statement

Note that this may be simplified during synthesis and that the size of the synthesized circuit can vary significantly for different stmts

Also note that it is syntactically correct for a signal to appear on both sides of a concurrent signal assignment
Simple Signal Assignment Statements

For example:

\[ q \leftarrow ((\text{not } q) \text{ and } (\text{not } en)) \text{ or } (d \text{ and } en); \]

Here, the \( q \) signal takes the value of \( d \) when \( en \) is ’1’, otherwise it takes the inverse of itself.

Although this is syntactically correct, the statement forms a *closed feedback loop* and should be avoided.

It may synthesize to an internal state (memory) in cases where the next value of \( q \) depends on the previous value, e.g.,

\[ q \leftarrow (q \text{ and } (\text{not } en)) \text{ or } (d \text{ and } en); \]

Or it may oscillate (as is true of the statement above)

This is REALLY BAD PRACTICE because the circuit becomes sensitive to internal propagation delay of its elements.

It also confuses the synthesis tools and complicates the testing process.
Conditional Signal Assignment Statements

Simplified syntax:

```vhdl
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_n
```

The `boolean_expr_i` return `true` or `false` and are each evaluated from top-to-bottom until one is found to be `true`.

When this occurs, the `value_expr_i` is assigned to the `signal_name` signal.

This type of statement can be represented by a `multiplexer` circuit.

<table>
<thead>
<tr>
<th>input s</th>
<th>output x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>a</td>
</tr>
<tr>
<td>0 1</td>
<td>b</td>
</tr>
<tr>
<td>1 0</td>
<td>c</td>
</tr>
<tr>
<td>1 1</td>
<td>d</td>
</tr>
</tbody>
</table>

This is the truth table for an 8-bit, **4-to-1 multiplexer**

Here, `a`, `b`, `c`, and `d` are input signals.

`s` is also an input, i.e., a 2-bit signal the input data to route to the output.
Conditional Signal Assignment Statements

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity mux4 is
    port(
        a, b, c, d: in std_logic_vector(7 downto 0);
        s: in std_logic_vector(1 downto 0);
        x: out std_logic_vector(7 downto 0)
    );

end mux4;

architecture cond_arch of mux4 is
begin
    x <= a when (s="00") else b when (s="01") else c when (s="10") else d;
end cond_arch;
```
Conditional Signal Assignment Statements

Note that the use of std_logic data type, which has 9 possible values, makes the last statement assign \( d \) to \( x \) under more conditions than the expected \( s = "11" \) case.

In fact, since each bit of \( s \) can assume 9 values, there are actually \( 9 \times 9 = 81 \) conditions for the two bit sequence including "0Z", "UX", "0-", etc.

Therefore, the last statement assigns \( d \) to \( x \) under 77 (81-4) additional conditions, but these conditions are ONLY possible in simulations.

Except for the limited use of 'Z', the metavales are ignored by synthesis software.

Some synthesis software allows the following alternative expression:

\[
x <= a \text{ when } (s="00") \text{ else b when } (s="01") \text{ else c when } (s="10") \text{ else d when } (s="11") \text{ else 'X'};
\]
### Conditional Signal Assignment Statements

**Binary decoder**: An $n$-to-$2^n$ decoder has an $n$-bit input and a $2^n$-bit output, where each bit of the *output* represents an *input* combination.

<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>

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity decoder4 is
  port(
    s: in std_logic_vector(1 downto 0);
    x: out std_logic_vector(3 downto 0)
  );
end decoder4;
```
Conditional Signal Assignment Statements

architecture cond_arch of decoder4 is
begin
  x <= "0001" when (s="00") else
       "0010" when (s="01") else
       "0100" when (s="10") else
       "1000";
end cond_arch;

Both the MUX and decoder are a better match to selected signal assignment (later)

Priority encoder: Checks the input requests and generates the code of the request with highest priority

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>code</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>code</td>
<td></td>
<td>active</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

There are four input requests, \( r(3), ..., r(0) \)

The outputs include a 2-bit signal (code), which is the binary code of the highest priority request and a 1-bit signal active that indicates if there is an active request
Conditional Signal Assignment Statements

The $r(3)$ has the highest priority, i.e., when asserted, the other three requests are ignored and the `code` signal becomes "11"

When $r(3)$ is **not** asserted, the second highest request, $r(2)$ is examined

The `active` signal is to distinguish the last case, when $r(0)$ is asserted and the case in which NO request is asserted

```
library ieee;
use ieee.std_logic_1164.all;

entity prio_encoder42 is
  port(
    r: in std_logic_vector(3 downto 0);
    code: out std_logic_vector(1 downto 0);
    active: out std_logic
  );
end prio_encoder42;
```
Conditional Signal Assignment Statements

\begin{verbatim}
architecture cond_arch of prio_encoder42 is
begin
  code <= "11" when (r(3)=’1’) else "10" when (r(2)=’1’) else "01" when (r(1)=’1’) else "00";
  active <= r(3) or r(2) or r(1) or r(0);
end cond_arch;
\end{verbatim}

The priority structure of the conditional signal assignment matches well this functionality

A simple ALU

\begin{center}
\begin{tabular}{l|l}
\hline
input & output \\
ctrl & result \\
\hline
0 & src0 + 1 \\
1 0 0 & src0 + src1 \\
1 0 1 & src0 - src1 \\
1 1 0 & src0 and src1 \\
1 1 1 & src0 or src1 \\
\hline
\end{tabular}
\end{center}

Input signals include \textit{ctrl, src0} and \textit{src1}

Output signal is \textit{result}

ALU performs 5 functions, 3 arithmetic and 2 Boolean

The input and output are interpreted as \textit{signed} integers when arithmetic ops are selected
Conditional Signal Assignment Statements

We will use \texttt{std_logic} data type for portability reasons and convert it to the desired data type, e.g., \texttt{signed}, in the architecture body.

Once the arithmetic operation is performed, the result is converted back to \texttt{std_logic}.

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

entity simple_alu is
  port(
    ctrl: in std_logic_vector(2 downto 0);
    src0, src1: in std_logic_vector(7 downto 0);
    result: out std_logic_vector(7 downto 0)
  );
end simple_alu;
```
Conditional Signal Assignment Statements

```vhdl
architecture cond_arch of simple_alu is
  signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
  -- note conversion to signed and back to std_logic
  inc <= std_logic_vector(signed(src0)+1);
  sum <= std_logic_vector(signed(src0)+signed(src1));
  diff <= std_logic_vector(signed(src0)-signed(src1));
  result <= inc when ctrl(2)='0' else
    sum when ctrl(1 downto 0)="00" else
    diff when ctrl(1 downto 0)="01" else
    src0 and src1 when ctrl(1 downto 0)="10" else
    src0 or src1;
end cond_arch;
```

The conditional signal assignment statement implements a priority structure
This type of structure is naturally realized easily by the sequential execution of a
CPU when expressed in a traditional programming language (temporal)
Conditional Signal Assignment Statements

In hardware, a priority-routing structure implements the priority structure (spatial)

\[
\text{signal\_name} <= \\
\quad \text{value\_expr\_1 \ when \ boolean\_expr\_1 \ else} \\
\quad \text{value\_expr\_2 \ when \ boolean\_expr\_2 \ else} \\
\quad \text{value\_expr\_3 \ when \ boolean\_expr\_3 \ else} \\
\quad \ldots \\
\quad \text{value\_expr\_n}
\]

In order to construct a conditional signal assignment statement, three groups of hardware are needed:

- Value expression circuits
- Boolean expression circuits
- Priority routing network

The boolean expression circuits are used to control the priority routing network which determines which of the value expression circuits are connected to the output.

The priority routing network is implemented by a series of 2-to-1 multiplexers.
Conditional Signal Assignment Statements

A 2-to-1 abstract MUX

With \( sel = \text{true} \), the \( n \)-bit signal \( i1 \) is routed to the output, otherwise \( i0 \) is routed

Consider the statement

\[
\text{signal\_name} \leftarrow \text{value\_expr\_1 \ when \ boolean\_expr\_1 \ else \ value\_expr\_2};
\]
Conditional Signal Assignment Statements

\[
\text{signal\_name} <= \text{value\_expr\_1 \ when \ boolean\_expr\_1 \ else \ value\_expr\_2 \ when \ boolean\_expr\_2 \ else \ value\_expr\_3 \ when \ boolean\_expr\_3 \ else \ value\_expr\_4};
\]

Basically, for each statement, another level is added

Bear in mind adding too many creates a long combinational delay
Conditional Signal Assignment Statements
Actual implementations

2-to-1 MUX

1-bit version

3-bit version

Consider

```vhdl
signal a, b, y: std_logic;
...
y <= '0' when a=b else '1';
```

How is the condition part actually implemented?
Conditional Signal Assignment Statements

The input type of \( a=b \) is \textit{std\_logic} and the output type is \textit{Boolean}.

Although signals of type \textit{std\_logic} has nine values, during synthesis only ’0’ and ’1’ matter.

Logic ’0’ is used for \texttt{false} and logic ’1’ is used for \texttt{true}.

\[
y = (a \oplus b) \cdot 1 + (a \oplus b) \cdot 0
\]

simplifies to

\[
y = (a \oplus b)
\]

Consider

\begin{verbatim}
signal a, b, c, x, y, r: std_logic;
...

r <= a when x=y else
    b when x>y else c;
\end{verbatim}
Conditional Signal Assignment Statements

Conceptually

Consider

```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;

... 
```
Conditional Signal Assignment Statements

Value and Boolean expressions are more involved

We can continue to refine the blocks on the left to gate-level components

Good coding practice can improve the implementation carried out by the synthesis tool dramatically (as we will see later)
Selected Signal Assignment Statements

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;
```

Similar to a case stmt in a traditional programming language

The `select_expression` must produce a value of a discrete type or 1-D array
It can only have a finite number of possibilities

`choice_i` must be value of the data type, e.g., if `bit_vector(1 downto 0)` used in
`select_expression`, then choices must be "00", "01", "10" and "11"

Choices must be mutually exclusive and all inclusive
`others` can be used as last `choice_n`
Selected Signal Assignment Statements

We saw this earlier in the conditional signal assignment material -- entity is the same

```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;
```

Remember std_logic has 9 possible values so it’s not possible to list the last entry as

```vhdl
d when "11";
```

Alternatively (last line ignored during synthesis)

```vhdl
x <= a when "00",
    b when "01",
    c when "10",
    d when "11",
    'X' when others; -- can also use '--'
```
Selected Signal Assignment Statements

For the **binary decoder** (2-to2²) we saw earlier

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

For the **priority encoder** (4-to-2)

```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",
end sel_arch;
```
Selected Signal Assignment Statements

"00" when others;
active <= r(3) or r(2) or r(1) or r(0);
end sel_arch;

Recall that "11" is assigned to code if r(3) is '1'

The shortcut taken in the conditional assignment stmt, i.e.,
code <= "11" when (r(3)=’1’) else

can NOT be taken here and all 8 values that have a ’1’ for r(3) must be listed, e.g.,
"1000", "1001", "1010", "1011", ... "1111"

You might be tempted to make this more compact by using the ’-’ (don’t-care) as

with r select
code <= "11" when "1---",
"10" when "01--",
"01" when "001-",
"00" when others;
Selected Signal Assignment Statements

But this doesn’t work since the ’-’ value never occurs in a real circuit

Because of this, the first three stmts are never executed and the whole clause becomes equivalent to

code <= "00";

The simple ALU:

```
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
        result <=
            inc when "000"|"001"|"010"|"011",
            sum when "100",
            diff when "101",
```
Selected Signal Assignment Statements

```
src0 and src1 when "110",
src0 or src1 when others; -- "111"
```

end sel_arch;

Truth Table: A new application that selected signal assignment can implement

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>y</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</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;```
Selected Signal Assignment Statements

architecture a of truth_table is

    signal tmp: std_logic_vector(1 downto 0);

begin

    tmp <= a & b; -- concatenate a and b

    with tmp select

        y <= '0' when "00", -- rows of the truth table
             '1' when "01",
             '1' when "10",
             '1' when others; -- "11"

end a;

The conceptual implementation can be realized by a multiplexing circuit

Abstract (k+1)-to-1 MUX

Instead of true and false as sel signal choices, sel is a data type of (k+1) values, $c_0$ through $c_k$
Selected Signal Assignment Statements

Consider a `select_expression` with a data type of 5 values: c0, c1, c2, c3, c4

```vhdl
with select_expression select
    signal_name <= value_expr_0 when c0,
                   value_expr_1 when c1,
                   value_expr_n when others;
```

All selected signal assignment statements have similar conceptual diagrams

You just need to be careful because the number of select items may become large and there are limitations here
Selected Signal Assignment Statements
Examples, 4-to-1 MUX

Here, the port names for the \textit{sel} signal are assigned "00", "01", "10" and "11"
Selected Signal Assignment Statements

```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;
```

Let’s compare and look at how to convert between **conditional** and **selected** signal assignment stmts
Conditional and Selected Signal Assignment Statements

Although the two statements imply a different routing structure (priority vs. non-priority), it is always possible to convert between the two.

Consider

\[
\text{with sel select}\nonumber
\]
\[
\text{sig} \leq \text{value}\_\text{expr}\_0 \text{ when c0,}\nonumber
\]
\[
\text{value}\_\text{expr}\_1 \text{ when } c1 \mid c3 \mid c5,\nonumber
\]
\[
\text{value}\_\text{expr}\_2 \text{ when } c2 \mid c4,\nonumber
\]
\[
\text{value}\_\text{expr}\_n \text{ when others}; \nonumber
\]

The choices can be described using Boolean expression

\[
\text{sig} \leq \text{value}\_\text{expr}\_0 \text{ when (sel=c0) else } \text{value}\_\text{expr}\_1 \text{ when (sel=c1) or (sel=c3) or (sel=c5) else } \text{value}\_\text{expr}\_2 \text{ when (sel=c2) or (sel=c4) else } \text{value}\_\text{expr}\_n; \nonumber
\]
Conditional and Selected Signal Assignment Statements

To convert in the other direction requires a little more work

```vhdl
signal_name <=
    value_expr_0 when boolean_expr_0 else
    value_expr_1 when boolean_expr_1 else
    value_expr_2 when boolean_expr_2 else
    value_expr_n;
```

We create a 3-bit auxiliary selection signal where each bit represents a Boolean expression as a means of preserving the desired priority

```vhdl
sel(2) <= '1' when bool_expr_0 else 0;
sel(1) <= '1' when bool_expr_1 else 0;
sel(0) <= '1' when bool_expr_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;
```

This same structure works for any 3-when clause conversion
Conditional and Selected Signal Assignment Statements

Use **selected signal assignment** statement for circuits described by truth tables or truth-table-like functions, such as decoders and multiplexers.

Using it for priority structures can be inefficient

```vhdl
with r select
    code <= "11" when "1000"|"1001"|"1010"|"1011"|
          "1100"|"1101"|"1110"|"1111",
  ...
```

Here, 8 of the 16 ports of the multiplexer are connected to an identical expression.

Use **conditional signal assignment** for circuits that need to give preferential treatment for certain conditions or to prioritize operations.

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

As you can see, it can handle complicated conditions.
Conditional and Selected Signal Assignment Statements

Using conditional signal assignment for truth tables if not efficient because it over-
specifies the function

```vhdl
x <= a when (s="00") else
    b when (s="01") else
    c when (s="10") else
    d;
```

But this can be written as

```vhdl
x <= c when (s="10") else
    b when (s="01") else
    a when (s="00") else
    d;
```

Or in any other combination

This is overspecified because the conditional signal assignment gives priority to the
first when clause but it is not needed

This may cause additional circuitry to be added during synthesis (BAD PRACTICE)