RT-Level Combinational Logic

This slide set describes Register Transfer Level (RTL) components, including adders, comparators and multiplexers.

Plus other VHDL operators and constructs.

We’ve seen that logical operators, e.g., and and or, are synthesizable.

Relational operators and several arithmetic operators are also synthesizable.

These synthesize into larger, module-level components such as comparators and adders.

VHDL defines six relational operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Data type of operands</th>
<th>Data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>equal to</td>
<td>any</td>
<td>boolean</td>
</tr>
<tr>
<td>a /= b</td>
<td>not equal to</td>
<td>any</td>
<td>boolean</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>less than</td>
<td>any</td>
<td>boolean</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>less than or equal to</td>
<td>any</td>
<td>boolean</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>greater than</td>
<td>any</td>
<td>boolean</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>greater than or equal</td>
<td>any</td>
<td>boolean</td>
</tr>
</tbody>
</table>
Relational and Arithmetic Operators

During synthesis, comparators are *inferred* for the relational operators.

Other operators supported in the `std_logic_1164` package:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Data type of operands</th>
<th>Data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ** b</td>
<td>exponentiation</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>a * b</td>
<td>multiplication</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>a / b</td>
<td>division</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>a + b</td>
<td>addition</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>a - b</td>
<td>subtraction</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>concatenation</td>
<td>1-D array, element</td>
<td>1-D array</td>
</tr>
<tr>
<td><strong>not</strong> a</td>
<td>negation</td>
<td>boolean, std_logic,</td>
<td>same as operand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>std_logic_vector</td>
<td></td>
</tr>
<tr>
<td>a and b</td>
<td>and</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>a or b</td>
<td>or</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>a xor b</td>
<td>xor</td>
<td>same</td>
<td>same</td>
</tr>
</tbody>
</table>
Relational and Arithmetic Operators

The VHDL standard supports arithmetic operations on integer and natural (integers \( \geq 0 \)) data types.

Many times you want precise control over the exact number of bits and format, i.e., signed vs unsigned.

The IEEE numeric_std package supports this by adding signed and unsigned data types and defines the relational and arithmetic operators for them through operator overloading.

Both of these data types are defined as an array of elements of std_logic data type.

Use the following to invoke the package:

```vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```
# Relational and Arithmetic Operators

The following operators are overloaded in the `numeric_std` package.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Data type of operands</th>
<th>Data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a * b</td>
<td>multiply</td>
<td>unsigned, natural</td>
<td>unsigned, signed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed, integer</td>
<td></td>
</tr>
<tr>
<td>a + b</td>
<td>add</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>a - b</td>
<td>subtract</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>a = b, ...</td>
<td>relational ops</td>
<td>same</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Note that synthesis of the multiplication operator depends on the synthesis software and target device technology.

Many FPGAs have embedded multipliers that the synthesis software will use instead of building the multiplier from CLBs.

However, each FPGA has only a limited number of embedded multipliers and each has a limited input width that the user needs to be aware of.
Type Conversion

Since VHDL is strongly typed, std_logic_vector, unsigned and signed are treated as different data types (even though all are defined as an array of std_logic elements).

Therefore, conversion functions and type casting are required to convert between types.

<table>
<thead>
<tr>
<th>Data type of $a$</th>
<th>To data type</th>
<th>Conversion function/ type casting</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned, signed</td>
<td>std_logic_vector</td>
<td>std_logic_vector($a$)</td>
</tr>
<tr>
<td>signed, std_logic_vector</td>
<td>unsigned</td>
<td>unsigned($a$)</td>
</tr>
<tr>
<td>unsigned, std_logic_vector</td>
<td>signed</td>
<td>signed($a$)</td>
</tr>
<tr>
<td>unsigned, signed</td>
<td>integer</td>
<td>to_integer($a$)</td>
</tr>
<tr>
<td>natural</td>
<td>unsigned</td>
<td>to_unsigned($a$, size)</td>
</tr>
<tr>
<td>integer</td>
<td>signed</td>
<td>to_signed($a$, size)</td>
</tr>
</tbody>
</table>

Note: no direct conversion is possible between std_logic_vector and integer, because std_logic_vector is NOT interpreted as a number.

Some examples follow.
Type Conversion

Given the following:

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

signal s1, s2, s3, s4, s5: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);

u1 <= s1; -- ERROR
u2 <= 5;  -- ERROR
s2 <= u3; -- ERROR
s3 <= 5;  -- ERROR
```

The correct way to do it:

```vhdl
u1 <= unsigned(s1);
u2 <= to_unsigned(5, 4);
s2 <= std_logic_vector(u3);
```

Text gives other examples...
**Concatenation Operator**

Combines elements and small arrays to form larger arrays.

```vhdl
signal a1: std_logic;
signal a4: std_logic_vector(3 downto 0);
signal b8, c8, d8: std_logic_vector(7 downto 0);

b8 <= a4 & a4;
c8 <= a1 & a1 & a4 & "00";
d8 <= b8(3 downto 0) & c8(3 downto 0);
```

Concatenation is a *wiring-only* operation (no operators required).

The & operator can be used to implement *shifting*

```vhdl
signal a: std_logic_vector(7 downto 0);
signal rot: std_logic_vector(7 downto 0);

-- rotate a to the right by 3 bits
rot <= a(2 downto 0) & a(8 downto 3);
```
High Impedance and Conditional/Selected Signal Assignment

The *std_logic* data type includes ’Z’ (high impedance).

It can only be synthesized by a tri-state buffer

\[
\begin{array}{c|c|c}
\text{oe} & y & \\
0 & Z & \\
1 & a & \\
\end{array}
\]

\[
y <= a \text{ when oe = '1' else 'Z'};
\]

Commonly used to implement a *bidirectional port* (see text for example).

Conditional and Selected Signal Assignment

These are similar to *if* and *case* in C, except they are concurrent. During synthesis, they are mapped to a *routing* network.

Conditional Signal Assignment

\[
\text{signal_name <= val_expr_1 when bool_1 else val_expr_2 when bool_2 else ... val_expr_n;}
\]
**Conditional Signal Assignment**

The *Boolean* expressions are evaluated successively until one is found true. The \( val_x \) is then assigned to the *signal_name* if a \( bool_x \) returns true else \( val_n \) is assigned.

Note that this evaluation process implies a *priority routing* scheme, e.g.,

\[
\begin{align*}
  r &\leq a + b + c \quad \text{when } m = n \quad \text{else} \\
  &\quad a - b \quad \text{when } m > n \quad \text{else} \\
  &\quad c + 1;
\end{align*}
\]

Muxiplexers are used to implement:

Note that the *Boolean* expressions and *value* expressions are all evaluated *concurrently*.

Also note that there may be glitches at the output \( r \) due to delays along the various paths.
Conditional Signal Assignment
Example: Priority Encoder

<table>
<thead>
<tr>
<th>input $r$</th>
<th>output $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1---</td>
<td>100</td>
</tr>
<tr>
<td>01--</td>
<td>011</td>
</tr>
<tr>
<td>001-</td>
<td>010</td>
</tr>
<tr>
<td>0001</td>
<td>001</td>
</tr>
<tr>
<td>0000</td>
<td>000</td>
</tr>
</tbody>
</table>

library ieee;
use ieee.std_logic_1164.all;

entity priority_encoder is
  port(
    r: in std_logic_vector(4 downto 1);
    p: out std_logic_vector(2 downto 0);
  );
end priority encoder;
Conditional Signal Assignment

Example: **Priority Encoder**

```vhdl
architecture cond_arch of priority_encoder is
begin
  p <= "100" when (r(4) = '1') else
       "011" when (r(3) = '1') else
       "010" when (r(2) = '1') else
       "001" when (r(1) = '1') else
       "000";
end cond_arch;
```

Example: **Binary Decoder**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity decoder_2_4 is
port(
  a: in std_logic_vector(1 downto 0);
  en: in std_logic;
  y: out std_logic_vector(3 downto 0);
);
Conditional/Selected Signal Assignment

Example: **Binary Decoder**

```vhdl
);
end decoder_2_4;

architecture cond_arch of decoder_2_4 is
begin
    y <= "0000" when (en = '0') else "0001" when (a = "00") else "0010" when (a = "01") else "0100" when (a = "10") else "1000";
end cond_arch;
```

**Selected Signal Assignment**

```vhdl
with sel select
    sig <= val_expr_1 when choice_a;
    val_expr_2 when choice_b; ...
    val_expr_n when others;
```
Selected Signal Assignment

The selected signal assignment stmt is similar to a case stmt.

The choice_x must be valid values of sel, they must be mutually exclusive (non-priority) and all inclusive (all possible values of sel must be given).

sel is usually of type std_logic_vector and therefore the others clause ensures all other values of sel are covered, including unsynthesizable values (’X’, ’U’, etc.)

The selected signal assignment stmt implies a non-priority based MUX scheme.

```vhdl
signal sel: std_logic_vector(1 downto 0);
with sel select

  r <= a + b + c when "00";
  a - b       when "10";
  c + 1       when others;
```

![MUX circuit diagram]

sel

(9/3/08)
Selected Signal Assignment

Example: **Priority Encoder** *(entity declaration identical to that shown earlier)*.

```vhdl
architecture sel_arch of priority_encoder is
begin
  with r select
  p <= "100" when "1000" | "1001" | "1010" | "1011" |
    "1100" | "1101" | "1110" | "1111",
  "011" when "0100" | "0101" | "0110" | "0111",
  "010" when "0010" | "0011"
  "001" when "0001",
  "000" when others; -- r = "0000"
end cond_arch;
```

Example: **Binary Decoder**

```vhdl
architecture sel_arch of decoder_2_4 is
signal s: std_logic_vector(2 downto 0);
begin
  s <= en & a; -- concatenate signals
```
Selected Signal Assignment and Process Construct

Example: Binary Decoder

```vhdl
with s select
  y <= "0000" when "000" | "001" | "010" | "011",
    "0001" when "100",
    "0010" when "101",
    "0100" when "110",
    "1000" when others; -- s = "111"
end cond_arch;
```

Process Construct

The *process* construct allows stmts to be executed *sequentially* (just as in a traditional programming language).

This helps with system modeling by providing a recipe type approach, e.g., do this, followed by that, etc.

Note that the *process* itself is a *concurrent* stmt, that executes in parallel with other concurrent stmts, such as those we just covered.
Caveats Concerning Processes

There are a variety of *sequential* stmts that can be included in a process, many of which don’t have a clear hardware implementation.

In general, processes provide the user with the ability to describe the system *behaviorally*, which, as you will see, is a significant enhancement over *structural*.

But consider the following caveats:

Perhaps the most difficult part for students who begin writing behavioral code is to stop thinking about programming and start thinking about hardware.

In other words, for every piece of behavioral code that you write, you should have a combinational or sequential circuit in mind that it should synthesize to.

Do NOT think of a *process* as an escape mechanism that allows you to just write code as you would in C to implement an algorithm.

Start by drawing the hardware system that you want to realize FIRST, and then start the coding process.
Caveats Concerning Processes

Caveats:

This may require some experimentation with smaller circuits.

I know what you are thinking! What’s the point -- why not just write structural code directly.

You can, of course, but you will be limited in what you can do.

The behavioral synthesis tools will enable you to synthesize MUCH more complex systems, but you’ll need to learn what to expect from them.

Once you’ve practiced with smaller circuits and understand what to expect, you will gain confidence and learn to appreciate the power of behavioral synthesis.

If you follow the coding style you learned C programming 101, you’ll certainly become frustrated or end up coding a system that doesn’t synthesize.

The most important exercise that you can do as a novice is write code fragments, synthesize and look at the structural schematic that is generated.
In this course, we will restrict the use of sequential stmts within processes to if and case stmts and templates for memory elements.

The simplified syntax of a process is

```
process (sensitivity_list)
begin
    sequential stmt;
    sequential stmt;
end process;
```

The sensitivity_list is a set of signals to which the process responds.

For processes designed to describe combinational logic, ALL input signals (those used on the right side of the sequential stmts) must be included in the list.

**Sequential Signal Assignment**

```
sig <= value_expression;
```
Process: Sequential Signal Assignment

Note that the concurrent signal assignment is very similar.

The behavior or semantics are different however.

In a process, it is possible to assign multiple times to the same signal.

In such cases, only the last assignment takes effect.

\[
\text{process } (a, \ b) \\
\text{begin} \\
\ c <= a \ \text{and} \ b; \\
\ c <= a \ \text{or} \ b; \quad -- \ c \text{ is determined by this stmt} \\
\text{end process;}
\]

If these statements appear outside a process, this implies that the wire represented by \( c \) is driven by BOTH an \text{and} and an \text{or} gate.

This obviously implies a design error.

The most common use of multiple assignments within a process is to avoid unintended memory, as we shall see.
Process: if Stmt
if and case are commonly used within processes to describe routing structures.

if stmt:
    if boolean_expr_1 then
        sequential_stmts;
    elsif boolean_expr_2 then
        sequential_stmts;
    ...
    else
        sequential_stmts;
    end if;

The if stmt is similar to the concurrent conditional signal assignment stmt.
They are equivalent when there is only one sequential_stmt inside the if.

For example, the following can be re-written using if stmts.

\[
\begin{align*}
  r &\leq a + b + c \text{ when } m = n \text{ else } \\
  &a - b \text{ when } m > n \text{ else } \\
  &c + 1;
\end{align*}
\]
**Process: if Stmt**

As an if stmt within a process:

```vhdl
process (a, b, c, m, n)
begin
    if m = n then
        r <= a + b + c;
    elsif m > 0 then
        r <= a - b;
    else
        r <= c + 1;
    end if;
end;
```

Example: **Priority Encoder**

```vhdl
architecture if_arch of priority_encoder is
begin
    process (r)
    begin
        if (r(4) = '1') then
```
Process: if and caseStmts

    p <= "100";
    elsif (r(3) = '1') then
        p <= "011";
    elsif (r(2) = '1') then
        p <= "010";
    elsif (r(1) = '1') then
        p <= "001";
    else
        p <= "000";
    end if;

end process;

end;

case stmt:
    case sel is
        when choice_1 =>
            sequential_stmts;
        when others => ...
end case;
Process: case Stmt

The same rules apply here as those described for selected signal assignment stmts. choice_x must be valid for sel, choices must be mutually exclusive and all inclusive.

The two are equivalent when the branches of the case contain only a single stmt.

```
with sel select
  r <= a + b + c when "00";
  a - b    when "10";
  c + 1    when others;
```

Equivalent to:
```
process(a, b, c, sel)
begin
  case sel is
    when "00" => r <= a + b + c;
    when "10" => r <= a - b;
    when others => r <= c + 1;
  end case;
end;
```
**Process: if and caseStmts**

The *if* and *case* stmts allow *any number* and *any type* sequential stmts.

This enables a more concise way to describe a circuit (over *conditional* and *selected* signal assignment stmts) and sometimes more efficient.

For example, consider the circuit that sorts two input signals and routes them to the *large* and *small* outputs.

\[
\begin{align*}
\text{large} & \leq a \quad \text{when} \quad a > b \quad \text{else} \quad b; \\
\text{small} & \leq b \quad \text{when} \quad a > b \quad \text{else} \quad a;
\end{align*}
\]

With the two ‘>’ operators, the synthesis software may infer two comparators. Alternatively:

\[
\begin{align*}
\text{process}(a, b) \; & \; \\
\text{begin} \; & \; \\
\text{if} \; a > b \; \text{then} \; & \; \\
\text{large} \; & \leq a;
\end{align*}
\]
Process: if and case Stmts

```vhdl
if and case Stmts
    small <= b;
    else
        large <= b;
        small <= a;
    end if;
end;
```

Here only one ‘>’ comparator is explicitly given.

Text gives another example where the if stmt is better because it allows a nested structure that is more intuitive.

Unintended Memory

When processes are used to describe combinational logic, it is easy to make a mistake that introduces a memory element.

The VHDL standard specifies that a signal will keep its previous value if it is NOT assigned in a process.
Unintended Memory in a Process

The following rules MUST be adhered to in order to avoid unintended memory.

- Include ALL input signals in the sensitivity list.
- Include the else branch with EVERY if stmt.
- Assign a value to EVERY signal in EVERY branch.

Here’s a code snippet that violates all three rules

```vhdl
process(a)
begin
  if ( a > b ) then
    gt <= '1';
  elsif ( a = b ) then
    eq <= '1';
  end if;
end process;
```

How does `gt` get assigned 0?
Unintended Memory in a Process

The correct way to specify it.

```vhdl
process (a, b)
begin
  if (a > b) then
    gt <= '1';
    eq <= '0';
  elsif (a = b) then
    gt <= '0';
    eq <= '1';
  else
    gt <= '0';
    eq <= '0';
  end if;
end process;
```

Is there a more efficient way that violates the ’assign all signals in all branches’ rule?

Assigning a signal *multiple* times can be done in this context but beware in others.
**Constants**

It is good practice to replace *literals* is symbolic constants.

General form:

```vhdl
constant const_name: data_type := value_expr;
```

For example:

```vhdl
constant DATA_BIT: integer := 8;
customar DATA_RANGE: integer := 2**DATA_BIT - 1;
```

The constant expression is evaluated during pre-processing and therefore has no effect on the synthesized circuit.

Example: **Adder**

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

Example: Adder (without constants)

```vhdl
entity add_w_carry is
  port (
    a, b: in std_logic_vector(3 downto 0);
    cout: out std_logic;
    sum: out std_logic_vector(3 downto 0)
  );
end add_w_carry;

architecture hard_arch of add_w_carry is
  signal a_ext, b_ext, sum_ext: unsigned(4 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(3 downto 0));
  cout <= sum_ext(4);
end hard_arch;
```
Constants and Generics

Example: Adder (WITH constants)

```vhdl
architecture const_arch of add_w_carry is
  constant N: integer := 4;
  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 const_arch;
```

Generics

A generic is a construct that can also be used to pass ‘constant’ information into an entity.

It cannot be modified inside the architecture.
Generics

entity entity_name is
  generic(
    generic_name: data_type := default_values;
    generic_name: data_type := default_values;
    ...
  )
  port(
    port_name: mode data_type;
    ...
  )
end entity_name;

For the Adder example:

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

entity gen_add_w_carry is
  generic(N: integer := 4);
end entity gen_add_w_carry;
Generics

    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;

N is also defined within the architecture body:

    architecture gen_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 gen_arch;
Generics

Another advantage is that the *generic* can be assigned in the component instantiation (*generic mapping*), so a different value can be used when the component is re-used elsewhere.

```vhdl
signal a4, b4, sum4: unsigned(3 downto 0);
signal a8, b8, sum8: unsigned(7 downto 0);
signal a16, b16, sum16: unsigned(15 downto 0);
signal c4, c8, c16: std_logic;

-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(gen_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(gen_arch)
generic map(N=>16)
port map(a=>a16, b=>b16, cout=>c16, sum=>sum16);
```

The 4-bit version can be instantiated WITHOUT the *generic map* (4 is default value)