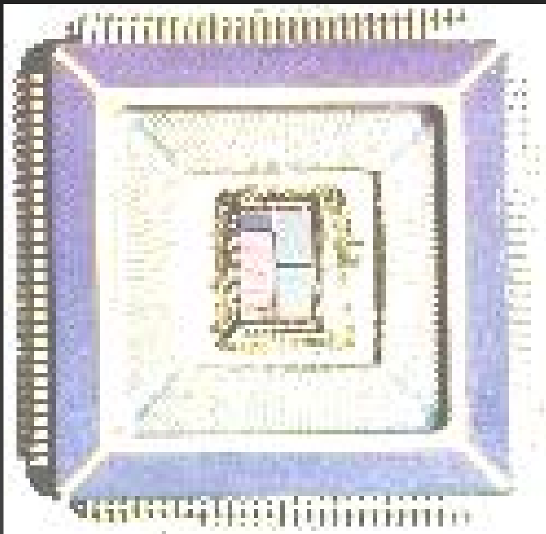


Hardware Modeling

VHDL – Syntax

Vienna University of Technology
Department of Computer Engineering
ECS Group

- Identifiers
- Types & Attributes
- Operators
- Sequential Statements
- Subroutines

- Consists of characters, digits and underscores
- Case insensitive
- First letter must be a character!
- Underscores not allowed at the beginning/end
- No two underscores in a row

Identifiers - Examples

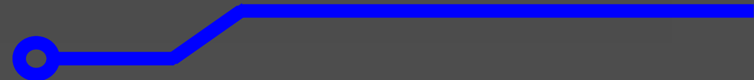
- Valid identifiers
  - state, name, stage1, cpu2, memory_3
- Invalid identifiers:
  - _state, _memory
  - 2stage
  - cpu_
  - my__name
  - $name

Contents

- Identifiers
- Types & Attributes
- Operators
- Sequential Statements
- Subroutines

VHDL Types

- bit [0,1], bit_vector
- std_logic, std_ulogic
- std_logic_vector, std_ulogic_vector
- integer, real
- Enumerations
- boolean
- character, string

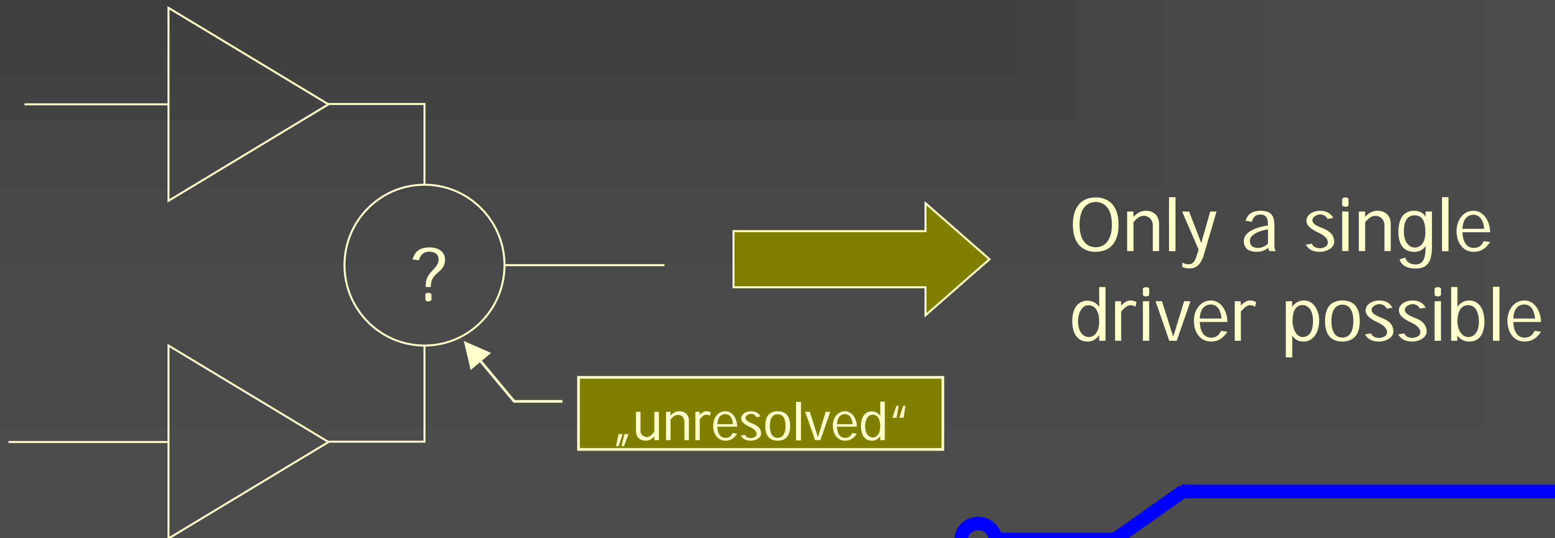


IEEE 1164 Logic System

- std_logic: can have one of the following nine values:

```
U, X, 0, 1, Z, W, L, H, -
```

- Difference std_ulogic and std_logic



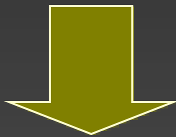
Vectors

- Collection of multiple elements of the same type
- Increasing/decreasing index range possible (to/downto)
- Positional binding on assignments
- Concatenation possible
- Bit-vectors have no numerical meaning ("1001" smaller than "111")

Vector Examples

```
signal a_bus: bit_vector(3 downto 0);
signal b_bus: bit_vector(0 to 3);
signal c_bus: bit_vector(15 downto 0);
signal A,B,C: bit;
```

```
a_bus <= b_bus;
```



```
a_bus(3) <= b_bus(0);
a_bus(2) <= b_bus(1);
a_bus(1) <= b_bus(2);
a_bus(0) <= b_bus(3);
```

```
a_bus(2) <= '1';
a_bus(1 downto 0) <= "10";
b_bus(0 to 1) <= "01";
a_bus(2 downto 0) <= b_bus(1 to 3);
b_bus <= (A,'0',C,'0');
```

```
c_bus <= a_bus & b_bus & "00000000";
c_bus <= (7 => B,15 downto 8 =>'1',others => '0');
```

Integers

- Integers are defined in the range of $-2^{31}+1$ to $+2^{31}-1$ (32 bit)
- May be restricted using a range specifier
- Predefined integer arithmetic
- Examples:

```
signal i : integer;
signal counter : integer range 0 to 255;
signal a, b, c : integer range 10 to 20;
```

- Package: numeric_std
- signed / unsigned are special versions of std_logic_vectors
- Explicit casting required
  - to_signed, to_unsigned, to_integer
  - unsigned, signed, std_logic_vector
- Arithmetics only defined for signed and unsigned but not for std_logic_vector!

Enumerations

```
type TYPE NAME is (NAME1, NAME2, ...);
```

- Examples:

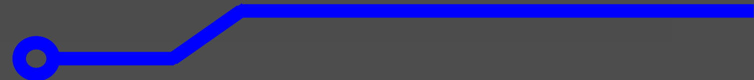
```
type MEM_OP_TYPE is (NONE, READ, WRITE);
type STATE_TYPE is (STATE_IDLE, STATE_RUNNING, STATE_SLEEP);
type CACHE_ROW_TYPE is (MODIFIED, EXCLUSIVE, SHARED, INVALID);
```

- Encoding examples:
  - NONE="00", READ="01", WRITE="10 "
  - STATE_IDLE="001", STATE_RUNNING="010", STATE_SLEEP="100"
  - MODIFIED ="0001", EXCLUSIVE="0010", SHARED= ="0100", INVALID="1000"
  - MODIFIED ="00", EXCLUSIVE="01", SHARED= ="10", INVALID="11"

- Enumeration items
  - STATE_IDLE, EXCLUSIVE
- Integers and Reals
  - 2, 250_000, 3E6, 2.0, 2.6E-5, 3_123.56
- Characters
  - `x` , `1`
- Strings
  - "a string" & "another one"
- Bits and Bit-Strings
  - `0` , `1` ,
  - "11001", b"11001", x"1A", o"31"

Attributes

- Range based attributes (examples)
  - a'left, a'right
  - a'low, a'high
  - a'range
  - a'length
- Signal bases attributes (example)
  - s'event
  - s'stable



Attributes Examples

```
signal a : std_logic_vector(3 downto 0);
signal b : std_logic_vector(0 to 3);
```

a‘left → 3 a‘right → 0 a‘high → 3 a‘low → 0

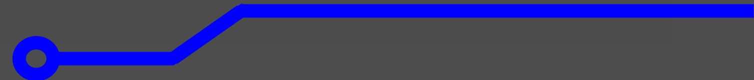
b‘left → 0 b‘right → 3 b‘high → 3 b‘low → 0

a‘range → 3 downto 0 b‘range → 3 to 0

a‘length → 4 b‘length → 4

Contents

- Identifiers
- Types & Attributes
- Operators
- Sequential Statements
- Subroutines



Operators (1)

- Logical operators (and, or, xor, nand, nor, xnor)
- Comparison operators (=, /=, <,<=,>,>=)
- Shift operators (sll, srl, sla, sra, rol, ror)
- Addition operators (+, -, &)
- Signs (+, -)
- Multiplication operators (*, /, mod, rem)
- Other operators (**, abs, not)

Operators given in ascending priority!

Operators (2)

- Keep implementation complexity in mind!
  - Especially for mult, div, mod, rem

- Operators may be overloaded
- Comparison of std_logic_vector with different length:
  - "110" > "1010"

Contents

- Identifiers
- Types & Attributes
- Operators
- Sequential Statements
- Subroutines

Sequential Statements

- Signal assignments
- Variable assignments
- Wait statements
- If-Then-Else statements
- Case statements
- Loop statements
- Assert statements

- Comparable to the variables of a C program
- Declared within processes
- Immediate assignment of values (in difference to signal assignments)
- Variables keep the value between different iterations of a process

Variables - Example

```
process (A,B,C)
  variable X,Y : bit;
begin
  X := A;
  Y := B;
  Z <= X and Y;
  Y := C;
  R <= X and Y:
end process;
```

Declared within the process

Immediate assignments

Constants

- Comparable to constants in C
- Value defined at declaration
- Value not changeable
- Declaration possible in packages, architectures or processes
- Examples:

```
architecture beh of xyz is
   constant CNT_MAX : integer := 1000;
   constant INST_ADD : std_logic_vector(3 downto 0) := x"A";
begin
   .....
```

Difference Signal - Variable

	Signal	Constant	Variable
Changeable	Yes	No	Yes
Assignment operator	<=	------	:=
Assignment executed	At next wait statement	------	Immedeatly
Visibility	Whole architecture	Depending on definition	Process locally

Variable-/Signal-Assignment

```
architecture beh of abc is

begin
  process(A, B, C)
    variable X, Y : bit;
  begin
      X := A;
      Y := B;
      Z <= X and Y;
      Y := C;
      R <= X and Y;
  end process;
end architecture beh;
```

```
architecture beh of abc is
  signal X, Y : bit;
begin
  process(A, B, C, X, Y)

  begin
      X <= A;
      Y <= B;
      Z <= X and Y;
      Y <= C;
      R <= X and Y;
  end process;
end architecture beh;
```

Variable-/Signal-Assignment

```
architecture beh of abc is

begin
  process(A, B, C)
    variable X, Y : bit;
  begin
      X := A;
      Y := B;
      Z <= X and Y;
      Y := C;
      R <= X and Y;
  end process;
```

```
architecture beh of abc is
   signal X, Y : bit;
begin
  process(A, B, C, X, Y)

  begin
      X <= A;
      Y <= B;
      Z <= X and Y;
      Y <= C;
      R <= X and Y;
  end process;
```

Iteration	Left Process	Right Process
First	Z = A and B R = A and C	Z = X_{old} AND Y_{old} R = X_{old} AND Y_{old}
Second	No second iteration	Z = A AND C R = A AND C

Shared Variables

- Introduced in VHDL'93
- Visible in multiple processes
- Immediate assignment
- Attention: may lead to indeterministic behavior
- Only synthesizable in special cases (e.g. memory with multiple write ports in different clock domains)

Wait Statements

Four variants:

- wait on signal list;
- wait until condition;
- wait for time;
- wait;

Examples:

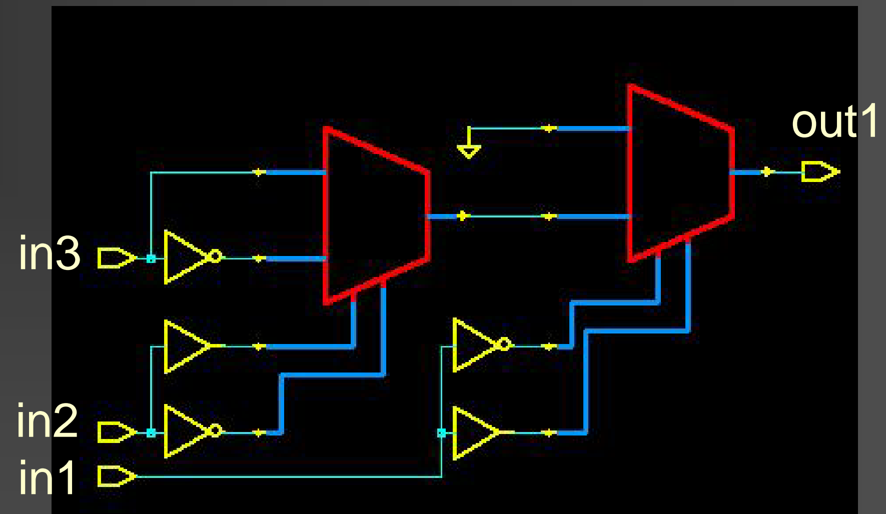
```
wait on clk;
wait until clk = '1' and clk'event;
wait for 10 ns;
```

Syntax:

```
[label :] if boolean expression then
  {sequential statement}
[{elsif boolean expression then
      {sequential statement}}]
[else
      {sequential statement}]
end if [label];
```

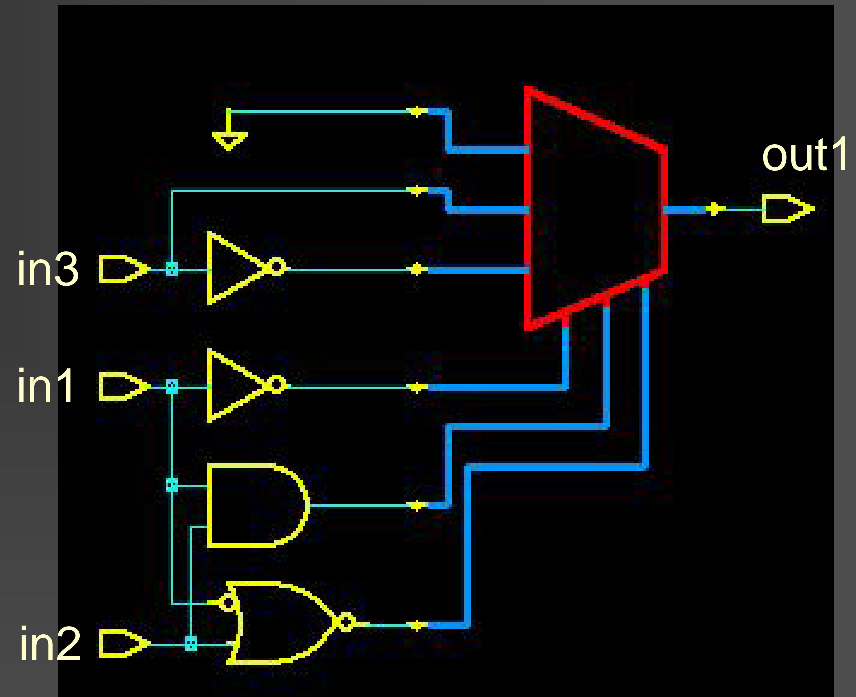
If-Then-Else Example 1

```
process (in1, in2, in3)
begin
  if in1 = '0' then
    out1 <= '0';
  else
    if in2 = '1' then
      out1 <= in3;
    else
      out1 <= not in3;
    end if;
  end if;
end process;
```



If-Then-Else Example 2

```
process (in1, in2, in3)
begin
   if in1 = '0' then
     out1 <= '0';
   elsif in2 = '1' then
     out1 <= in3;
   else
     out1 <= not in3;
   end if;
end process;
```



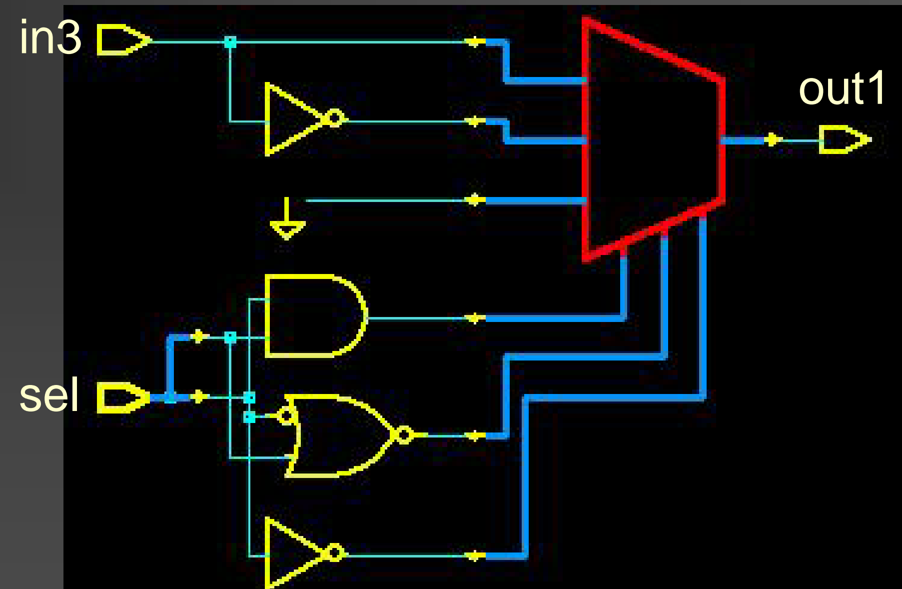
Case Statement

Syntax:

```
[case_label :] case expression is
  {when choices =>
    {sequential statement}}
  [when others =>
    {sequential statement}]
end case [case_label];
```

Case Example

```
process (sel, in3)
Begin
  case sel is
    when "11" =>
      out1 <= in3;
    when "10" =>
      out1 <= not in3;
    when others =>
      out1 <= '0';
  end case;
end process;
```



```
[label :] [iteration scheme] loop
          {sequential statement}
end loop [label];
```

Iteration scheme:

```
while condition
for range
```

Example: Parity Check

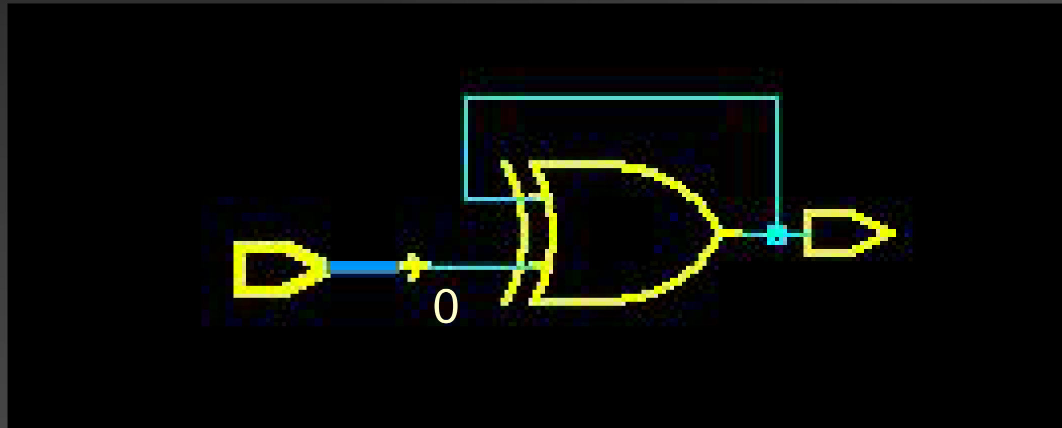
- Input: 16 bit data word
  - as std_logic_vector, most significant bit on highest index (➔ 15 downto 0)
- Output: Single parity bit (std_logic)
  - 1 if number of 1's in data word is odd
  - 0 otherwise
- Implementation as a process
- Using different forms of loops

Example: Parity Check (Signal)

```
architecture beh_sig of parity is
  signal parity : std_logic;
begin
  process (in_vec)
  begin
    parity <= '0';
    for i in 15 downto 0 loop
      parity <= parity xor in_vec(i);
    end loop;
    out1 <= parity;
  end process;
end beh_sig;
```

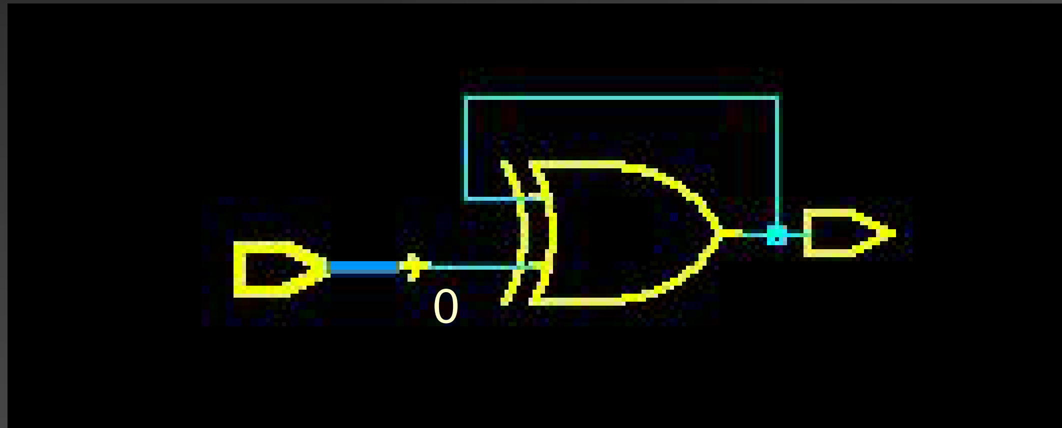
Example: Parity Check (Signal)

- Synthesis result:



Example: Parity Check (Signal)

- Synthesis result:



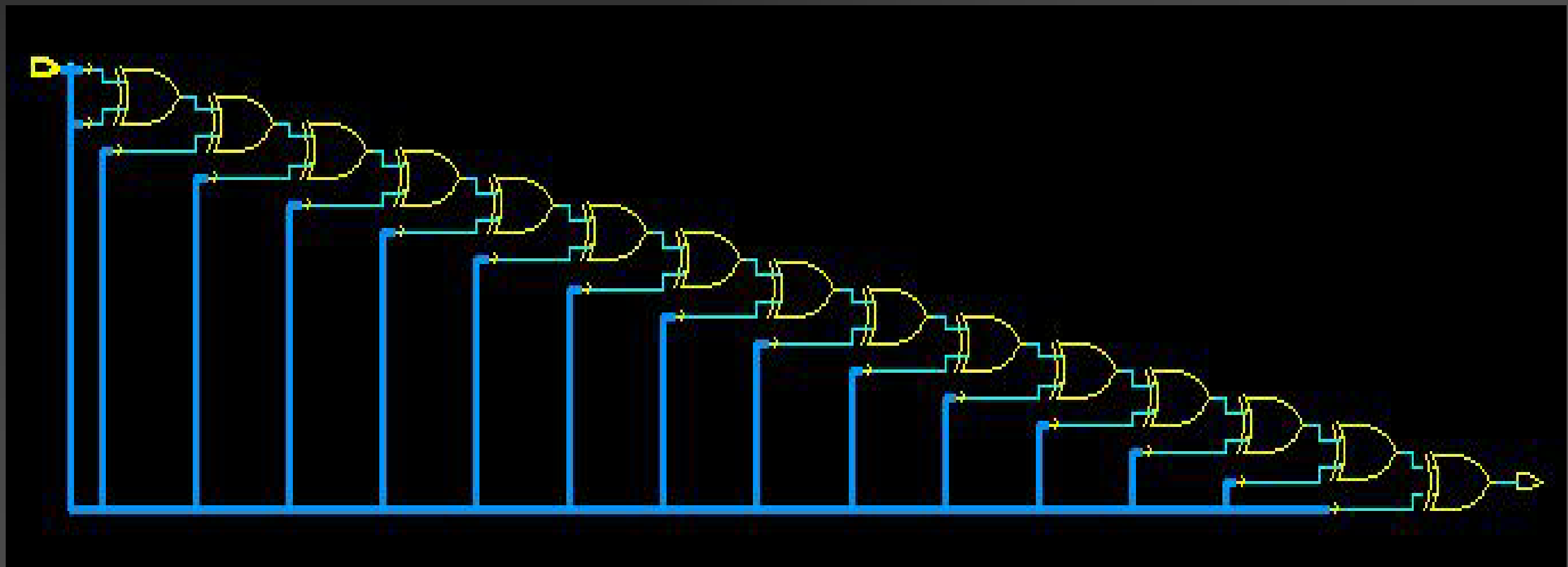
Not quite what we expected, isn't it?

Example: Parity Check (Variable)

```
architecture beh_var of parity is
begin
  process (in_vec)
    variable parity : std_logic;
  begin
    parity := '0';
    for i in 15 downto 0 loop
      parity := parity xor in_vec(i);
    end loop;
    out1 <= parity;
  end process;
end beh_var;
```

Example: Parity Check (Variable)

- Synthesis result:



Example: Parity Check (while)

```
architecture beh_while of parity is
begin
  process (in_vec)
    variable parity : std_logic;
    variable i : integer range 0 to 16;
  begin
    parity := '0';
    i := 0;
    while i < 16 loop
      parity := parity xor in_vec(i);
      i := i + 1;
    end loop;
    out1 <= parity;
  end process;
end beh_while;
```

```
assert condition [report string] [severity level];
```

- Activated, if the condition evaluates to false
- string is printed to simulation console
- severity level: note, warning, error or failure
- Abort: level dependents on used simulator
- Is widely used as abort condition for testbenches
- Simplifies circuit validation

Contents

- Identifiers
- Types & Attributes
- Operators
- Sequential Statements
- Subroutines

Subroutines (1)

- Functions
  - No, one or multiple arguments
  - Returns exactly one value
  - Can be used in places where the return type may be used
  - Explicit return statement necessary
- Procedures
  - List of bidirectional arguments
  - Can be used in places where a statement may be used
  - Terminates either at a return statement or at the end of the function

Subroutines (2)

	Function	Procedure
Argument direction	INPUT	INPUT, OUTPUT or BIDIRECTIONAL
Argument types	Constants, signals	Constants, signals, variables
Return value	Exactly one	Abitrary count
Usable in	Places the return type may be used	Places a statement may be used
Return statement	Necassary	Optional

```
function name
[ ( {[class] name {, name} :
            [in] type [:=default value] ;}
      [class] name {, name} :
            [in] type [:=default value] ) ]
return type;
```

Class:

```
constant
signal
```

Function Definition

```
function name
[ ( {[class] name {, name} :
             [in] type [:=default value] ;}
      [class] name {, name} :
             [in] type [:=default value] ) ]
return type is
   Declarations
begin
   Sequential statements
   return statement necessary
   No WAIT-Statements allowed!
end [function] [name];
```

Function - Example

```
function count_ones(a : bit_vector)
  return integer is
  variable count : integer := 0;
begin
  for c in a'range loop
    if a(c) = '1' then
      count := count+1;
    end if;
  end loop;
  return count;
end function count_ones;
```

- Note: The bit_vector has a variable width
  - Actual width is set at compile time

Function Call (1)

- Positional argument binding

```
name [(argument {,argument})]
```

- Named argument binding

```
name [(name=>argument  {, name=>argument})]
```

If a parameter is omitted, its default value is assumed!

Function Call (2)

- Each function call is replaced by a hardware instance.
- Resource sharing possible
- Examples:

```
signal my_vector : bit_vector(7 downto 0);
signal second_vector : bit_vector(31 downto 0);

num_ones <= count_ones(my_vector);

if count_ones(a=>second_vector) > 5 then
   ...
end if;
```

Procedure Declaration

```
procedure name
  [ ( {[class] name  {,name}  :
             [mode] type [:=default value] ;}
         [class] name {,name} :
             [mode] type [:=default value]
  ) ];
```

Class:

```
constant | signal | variable
```

Mode:

```
in | out | inout
```

```
procedure name
  [ ( {[class] name  {,name}  :
             [mode] type [:=default value] ;}
         [class] name {,name} :
             [mode] type [:=default value]
  ) ]
is
  Declarations
begin
  sequential statements
  optional return statement
end [procedure] [name];
```

Procedure - Example

```
procedure d_ff (
  constant delay : in time:=2ns;
  signal d, clk : in bit;
  signal q, q_bar: out bit) is
begin
  if clk = '1' and clk'event then
    q <= d after delay;
    q_bar <= not d after delay;
  end if;
end procedure d_ff;
```

- Delays not synthesizable!

Procedure Call (1)

- Positional argument binding

```
name [(argument {,argument})]
```

- Named argument binding

```
name [(name=>argument  {, name=>argument})]
```

If a parameter is omitted, its default value is assumed!

Procedure Call (2)

- Each procedure call is replaced by a hardware instance.
- Resource sharing possible
- Examples:

```
signal c : bit;
signal d1, q1, q1_n : bit;
signal d2, q2, q2_n : bit;

d_ff(delay=>10 ps, d=>d, clk=>c, q=>q, q_bar=>q_n);
d_ff(5 ps, d2, c, q2, q2_n);
```

- Identifiers
- VHDL type system
  - Scalar types
    - 9-valued logic (std_logic)
    - Arithmetic on std_logic
  - Vectors

Summary

- Operators
  - Keep hardware costs in mind (e.g., /, mod)
- Sequential statements
  - Effects of „else“ and „others“ clause
- Subroutines
  - Procedure
  - Function