Entity

entity name **is**
  [generic(generic list**);]**
  port(port definition list **);**
end entity name;
Port declaration: name: mode data_type;
Mode is one of: **in**, **out**, **buffer**, or **inout**.
Generics allow static information of a design unit:
generic (size: **integer**[:= 5]);

Architecture

architecture arch_name **of** entity_name **is**
    ... declarations ...
begin
    ... concurrent statements ...
end architecture arch_name;

Declarations include data types, constants, signals, components, attributes, subprograms.
Concurrent statements describe a design unit by dataflow, structure, and/or behavior modelling abstraction
Behavioral Model: Written in sequential, procedural style.
Dataflow Model: Data path and control signals.
Structural Model: Components connection.

IEEE Standard Logic 1164 Package

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

Identifiers

Identifiers in VHDL must begin with a letter, and may comprise any combination of letters, digits, and underscores.

Integer Numeric Constants

Format: **base#digits#** -- base is decimal
  16#9FBA# (hexadecimal)
  2#1111_1101_1011# (binary)

Bit String Literals

  X"FFE" (12-bit hexadecimal)
  O"777" (9-bit octal)
  B"111111011101" (12-bit binary)

Arithmetic and Logical Expressions

Logical: **and**, **or**, **nand**, **nor**, **xor**, **xnor, not** (for boolean or bit/bit_vector types)
Relational: =, /=, <, <=, >, >=
Arithmetic: +, -, *, /, mod, rem, **, abs

Aggregate and concatenation

signal b,c : **bit**;
signal a : **bit_vector**(0 **to** 7);
signal d : **bit_vector**(15 **downto** 0);

--Aggregate (, , ...,)
a<=(1=>'0',0|2=>b, 5 to 7=>c, others=>'0');

--concatenation **&**
d<=a & b & "101010" & c;

Data Types

- **bit**: '0', '1'
- **bit_vector** (N to M), where N, M is integer constants and M>=N
- **bit_vector** (M downto N), where N, M is integer constants and M>=N
- **boolean**: TRUE, FALSE
- **integer**: -(2**31) to +(2**31 - 1)
- **natural**: 0 to integer'high
- **positive**: 1 to integer'high
- **character**: ASCII characters
- **string** array (natural range <>) of char

Relations operators (returns Boolean)

> great than
< less than
>= great/equal than
<= less/equal than
= equality
/= non equality

Shift operations (bit_vector)

sll/srl- shift left/right logical
sla/sra- shift left/right arithmetic
rol/ror- rotate left/right

IEEE Standard 1164

std_ulogic/std_logic values:
'U','X','1','0','Z','W','H','L','-'
std_ulogic_vector array (natural range <>) of std_ulogic
std_logic_vector array (natural range <>) of std_logic

User-Defined Enumeration Types

type opcode **is** (add, sub, jump, call);
signal instruc: opcode;

User Defined Arrays

Constrained array: Indexes are specified.
type word **is array** (0 **to** 15) **of bit;**
Unconstrained array
type memory **is array** (**integer range** <>) **of bit_vector**(0 **to** 7);
signal memory256: memory(0 **to** 255);

Aliases

An alias defines an alternate name for a signal or part of a signal.
signal instruction:**bit_vector**(31 **downto** 0);
alias opcode: **bit_vector**(6 **downto** 0) **is** instruction(31 **downto** 25);

Constants
constant symbol: type:= value;

Variables
For process and subprograms
variable symbol: **type** [:=initial_value];

Signals
A signal is an object with a history of values (related to "event" times).
signal sig_name: data_type [:=initial_value];

Concurrent Signal Assignment
A <= B; A <= B when condition1 else C **when** condition2 **else** D **when** condition3 **else** E;

with expression **select**
A **<=** B **when** choice1,
C **when** choice2,
D **when** choice3,
E **when others**;

Process Statement
[comb:] **process** (sensitivity list)
... local declarations ...
begin
... sequential statements ...
end process [comb];

[sync:] **process** (clock [, asynchronous signals])
... local declarations ...
begin
if (asynchronous signals) **then**
... sequential statements ...
elsif (clock edge detecting) **then**
... sequential statements ...
end if;
end process [sync];

clock edge detecting:
clock'event and clock='1'–-clock rise
clock'**event and** clock='0'–-clock fall
by std_logic_1164 functions:
rising_edge(clock_name)
falling_edge(clock_name)

Component instantiation
instance_name: component_name
[**generic map** (generic_constant=>actual_value)]
port map (port list);

-- component declaration
component adder
port(a,b : **in bit_vector**(7 **downto** 0);
s : **out bit_vector**(7 **downto** 0);
cin : **in bit**;
cout: **out bit**);
end component;

-- component instantiation

-- Positional association
A1: adder **port map** (v,w,x,y,z)

-- Named association
A1: adder **port map**
(a=>v, b=>w, s=>y, cin=>x, cout=>z);

Generate statement
label: **for** i **in** range **generate**
--concurrent statement
-- processes
--component instantiation
end generate label;
label: **if** (condition) **generate**
--concurrent statement
--processes
--component instantiation
end generate label;

Conditional Statements
For processes and subprograms

if condition **then**
... sequence of statements...
[elsif condition **then**
... sequence of statements...]
[else
... sequence of statements...]
end if;

case expression **is**
when choices **=>** statements
when choices **=>** statements
...
when others => statements
end case;

Loop statements
For processes and in subprograms

label: **while** condition **loop**
... sequence of statements ...
end loop label;
label: **for** loop_variable **in** range **loop**
... sequence of statements...
end loop label;

next [loop_label] **[when** condition**];** -- end current iteration

exit [loop_label] **[when** condition**];** -- exit from loop

Procedures
procedure procedure_name
[([signal/variable/constant] arg_name :
in/out/inout type)] **is**
[local variable declarations]
begin
... sequence of statements ...
end procedure procedure_name;

Functions
function name (arg:type) **return** type **is**
variable v_name: type;
begin
... sequence of statements ...
return v_name;
end function name;

Type Conversion
std_logic_1164:
to_bit(std_ulogic/logic[,xmap])
to_bitvector(std_ulogic/logic_vector[, xmap])
to_stdulogic(bit)
to_stdlogicvector(bit/std_ulogic_vector)
to_stdulogicvector(bit/std_logic_vector)

numeruc_std:
signed/unsigned(from)
std_logic_vector(signed/unsigned)
to_integer(from)
to_unsigned(from, size)
to_signed(from, size)

std_logic_arith:
signed(std_logic/ulogic_vector/)
unsigned(std_logic/ulogic_vector)
std_logic_vector(unsigned/signed)
conv_integer(std_logic_vector)
conv_unsigned(integer, size)
conv_signed(integer, size)
conv_std_logic_vector(integer, size)

std_logic_unsigned/signed:
conv_integer(std_logic_vector)

"IEEE" library types operations
between all signal types and vectors of signal types
in the "ieee" library.

Logical operations:
and, or, nand, nor, xor, xnor, not

Shift operations:
shift/rotate left/right logical/arithmetic:
sll, srl, sla, sra, rol, ror
Ex. a := x sll 2; -- shift left logical of x by 2 bit

Relational operations:
=,/=,<,>,<=,>=

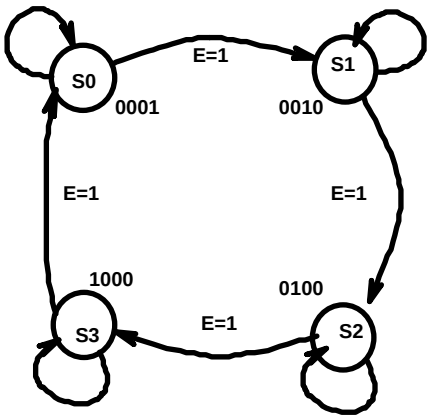
Arithmetic: +,-,*,/,rem,mod
SHIFT_LEFT(un, na) un
SHIFT_RIGHT(un, na) un
SHIFT_LEFT(sg, na) sg
SHIFT_RIGHT(sg, na) sg
ROTATE_LEFT(un, na) un
ROTATE_RIGHT(un, na) un
ROTATE_LEFT(sg, na) sg
ROTATE_RIGHT(sg, na) sg

Numeric_std/Numeric_bit:
RESIZE(sg, na) sg
RESIZE(un, na) un
STD_MATCH(u/l, u/l) bool
STD_MATCH(uv, uv) bool
STD_MATCH(lv, lv) bool
STD_MATCH(un, un) bool
STD_MATCH(sg, sg) bool

Array Attributes
A'LEFT(N) - left bound of index
A'RIGHT(N) - right bound of index
A'HIGH(N) - upper bound of index
A'LOW(N) - lower bound of index
A'LENGTH(N) - number of values in range of index
A'RANGE(N) - range: A'LEFT to A'RIGHT
A'REVERSE_RANGE(N) - range A'LEFT downto A'RIGHT

Type Attributes
T'LEFT(N) - left val of type
T'RIGHT(N) - right val of type
T'HIGH(N) - upper val of type
T'LOW(N) - lower val of type
T'POS(V) – position of val
T'VAL(P) – val of position

Example:




```
1  -- Writren by Flaxer Eli
2  -- One Process FSM
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity FSM is port (
7      Clk         :   in std_logic;
8      Reset       :   in std_logic;
9      Enable      :   in std_logic;
10     FsmOut      :   out std_logic_vector(3 downto 0));
11 end FSM;
12
13 architecture Flaxer of FSM is
14 TYPE    StatusType  IS (S0, S1, S2, S3);
15 -------------------------------------------------------------------
16 signal  CurStat :StatusType;
17 begin
18 -------------------------------------------------------------------
19 NextState: process(Clk, Reset)
20   begin
21     IF Reset = '1' THEN
22         CurStat <= S0;
23     ELSIF Rising_Edge(Clk) THEN
24       CASE CurStat IS
25         WHEN S0 =>  IF Enable = '1' THEN
26                         CurStat <= S1;
27                     ELSE
28                         CurStat <= S0;
29                     END IF;
30                     FsmOut <= "0001";
31         WHEN S1 =>  IF Enable = '1' THEN
32                         CurStat <= S2;
33                     ELSE
34                         CurStat <= S1;
35                     END IF;
36                     FsmOut <= "0010";
37         WHEN S2 =>  IF Enable = '1' THEN
38                         CurStat <= S3;
39                     ELSE
40                         CurStat <= S2;
41                     END IF;
42                     FsmOut <= "0100";
43         WHEN S3 =>  IF Enable = '1' THEN
44                         CurStat <= S0;
45                     ELSE
46                         CurStat <= S3;
47                     END IF;
48                     FsmOut <= "1000";
49       END CASE;
50     END IF;
51 end process;
52 -------------------------------------------------------------------
53 end Flaxer;
```