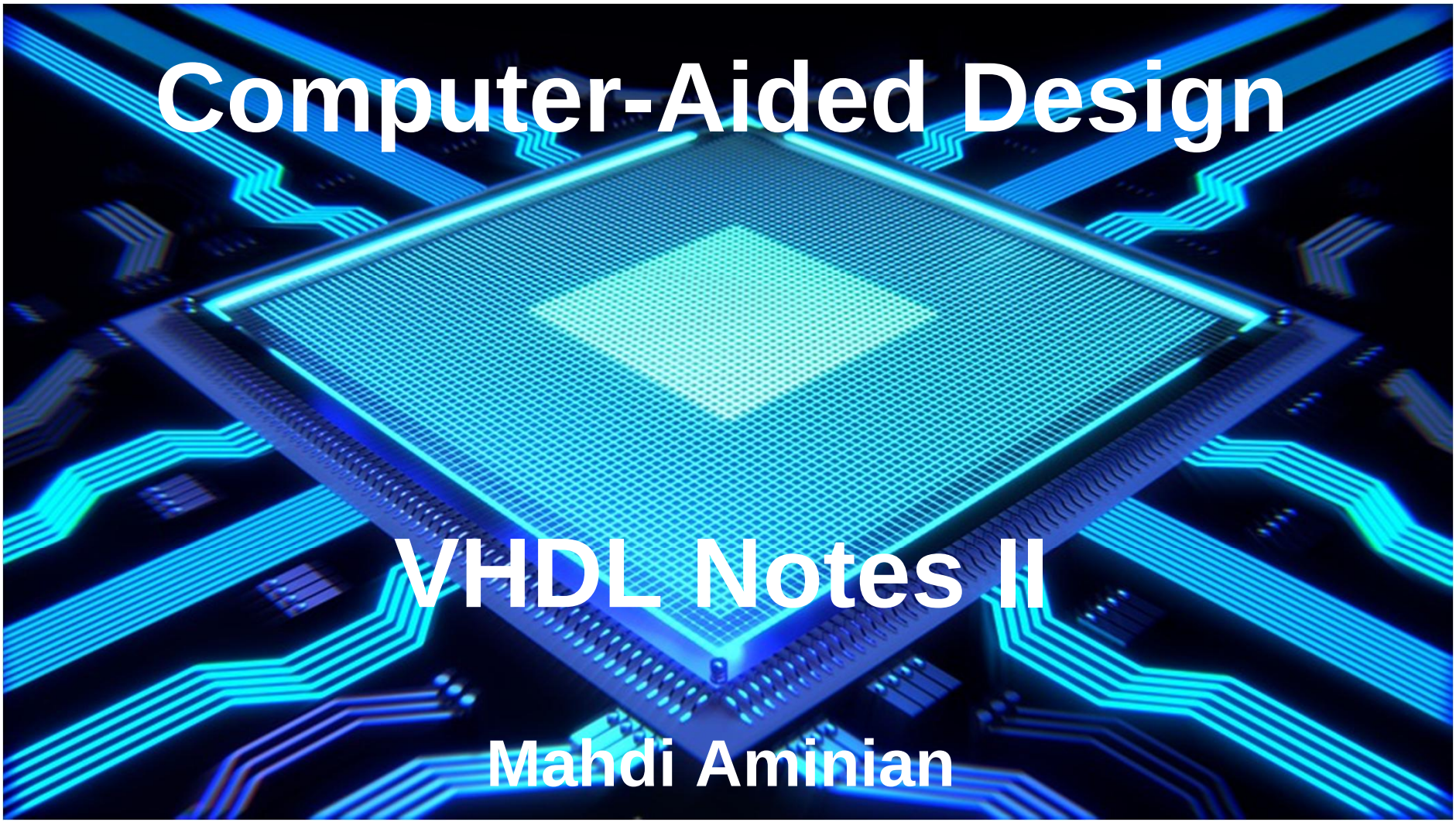


Computer-Aided Design

VHDL Notes II

Mahdi Aminian

The University Of Guilan

Rasht - Iran

Some slides courtesy of:

- “Hardware Systems Modeling”, A. Vachoux, EPFL
- CAD slides from Dr. Saheb Zamani

Subprograms

Are used as expression in other VHDL statements (concurrent or sequential)

- **Procedures**
  - May have **in**, **out**, or **inout** arguments
  - Sequential and concurrent procedures
- **Functions**
  - May only have mode **in** arguments
  - Represent terms in expressions
- **Predefined functions**
  - Arithmetic operators "+", "-", etc.
  - Logical operators "**and**", "**or**", etc.
  - Function **now** returning the current simulation time as a value of type time

Subprograms

- **Sequential Execution**
- **Functions**
  - function name can be an operator
  - arbitrary number of input parameters
  - exactly one return value
  - no WAIT statement allowed
  - function call <=> VHDL expression
- **Procedures**
  - arbitrary number of parameters of any possible direction (input/output/inout)
  - RETURN statement optional (no return value!)
  - procedure call <=> VHDL statement
  - WAIT is allowed (if its parent is not a function)
- **Subprograms can be overloaded**
- **Parameters can be constants, signals, variables or files**

Example of Procedure

```
architecture EXAMPLE of PROCEDURES is
  procedure COUNT_ZEROS (A: in std_logic_vector; signal Q: out
                integer) is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS +1;
      end if;
    end loop;
    Q <= ZEROS;
  end COUNT_ZEROS;
  signal COUNT: integer;
  signal IS_0:     boolean;
begin
  process
  begin
    IS_0 <= true;
    COUNT_ZEROS("01101001", COUNT);
    wait for 0 ns;
    if COUNT > 0 then   IS_0 <= false;  end if;
    wait;
  end process;
end EXAMPLE;
```

Are used as VHDL statements (concurrent or sequential)

No return value

Parameter values may be updated (mode out/inout)

Body split into declarative and definition part

Unconstrained parameters possible
(array size remains unspecified)

For out parameters, default: variable

Example of Function

```
architecture EXAMPLE of FUNCTIONS is
  [(im)pure] function COUNT_ZEROS (A: bit_vector)  return integer is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS +1;
      end if;
    end loop;
    return ZEROS;
  end COUNT_ZEROS;
  signal WORD:     bit_vector(15  downto 0);
  signal WORD_0: integer;    signal IS_0:        boolean;
begin
  WORD_0 <= COUNT_ZEROS(WORD);
  process
  begin
      IS_0 <= true;
      if COUNT_ZEROS("01101001") > 0 then
          IS_0 <= false;
      end if;
      wait;
  end process;
end EXAMPLE;
```

(Im)pure declaration optional (default: 'pure')

Body split into declarative and definition part

Unconstrained parameters possible (array size remains unspecified)

Are used as expression in other VHDL statements (concurrent or sequential)

Subprogram Declaration and Overloading

- **Subprograms may be declared/defined in any declaration part**
  - package
  - entity
  - architecture
  - process
  - subprogram
- **Overloading of subprograms possible**
  - identical name
  - different parameters
- **During compilation/runtime that subprogram is called whose formal parameters match the provided actuals**
  - **Cannot overload with different parameter classes.**

Subprogram Declaration and Overloading

- **توابع Impure به objectهای بیرون از scope خود دسترسی دارند.**
- **زیربرنامه ای که در package اعلان شده باشد در همهٔ واحدهایی که به این package ارجاع می دهند قابل دسترسی است.**
- **زیربرنامه ای که در زیربرنامهٔ دیگری اعلان شده باشد فقط در بدنهٔ زیربرنامهٔ parent آن قابل دسترسی است.**
- **با overloadکردن علائم می توان مثلاً + را برای bit_vectorها هم استفاده کرد.**
  - **مانند پکیج‌های std_logic_unsigned و std_logic_signed**

Package Declaration & Body

- **Design units**
- **Package declaration**
  - Declarations of constants, types, signals, subprograms, files, components
- **Package body**
  - Definition of deferred constants
  - Subprogram bodies
- **Used with a context clause**

```
use work.example_pkg.all;
-- assuming that the package units
-- have been analyzed in the
-- logical library WORK
```

```
package example_pkg is
   constant MAX : integer := 10;
   constant MAX_SIZE : natural;  -- deferred constant
   subtype bv10 is bit_vector(MAX-1 downto 0);
   procedure proc (A : in bv10; B : out bv10);
   function func (A, B : in bv10) return bv10;
end package example_pkg;
```

```
package body example_pkg is
   constant MAX_SIZE : natural := 200;

   procedure proc (A : in bv10; B : out bv10) is
   begin
      B := abs(A);
   end procedure proc;

   function func (A, B : in bv10) return bv10 is
      variable V : bv10;
   begin
      V := A and B;
      return (not(V));
   end function func;
end package body example_pkg;
```

Configuration Declaration

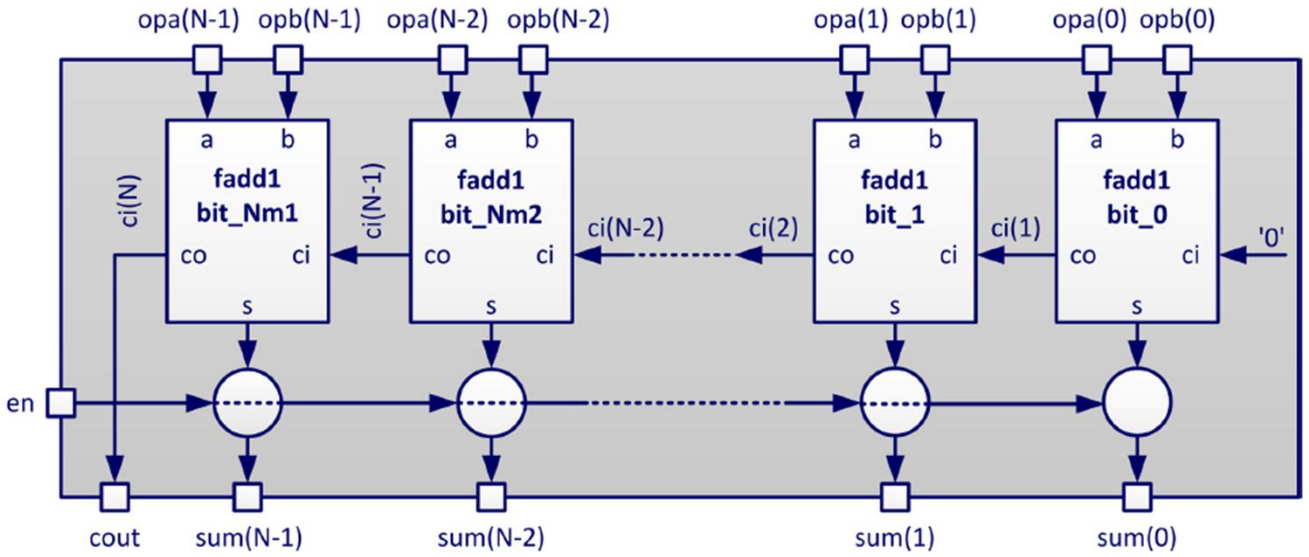
- **Binds component instances to analyzed design entities**
- **General form**

```
context-clause
configuration config-name of entity-name is
   for architecture-name
      for component-specification
         binding-indication;
      end for;
   end for;
end configuration config-name;
```

Default Configuration

- **Most recently analyzed architecture is considered**

Generic N-bit Adder Using Components




```
entity addne is
   generic (NBITS : positive := 8);  -- bus size
   port (
      signal en       : in  std_logic;
      signal opa, opb : in  std_logic_vector(NBITS-1 downto 0);
      signal sum      : out std_logic_vector(NBITS-1 downto 0);
      signal cout     : out std_logic);
end entity addne;
```

Generic N-bit Adder Using Components

```
architecture str of addne is
begin  -- architecture str
   STAGES : for k in NBITS-1 downto 0 generate
      signal s_unbuffered : std_logic;
   begin
      LSB : if k = 0 generate  --  least significant bit
         BIT : component c_fadd1
            port map (a => opa(0), b => opb(0), ci => '0',
                      s => s_unbuffered, co => ci(1));
      end generate LSB;

      OTHERB : if k /= 0 generate  -- all other bits
         BIT : component c_fadd1
            port map (a => opa(k), b => opb(k), ci => ci(k),
                      s => s_unbuffered, co => ci(k+1));
      end generate OTHERB;

      OUT_STAGE : process (en, s_unbuffered)  -- latch stage
      begin
         if en = '1' then
            sum(k) <= s_unbuffered;
         end if;
      end process OUT_STAGE;
   end generate STAGES;
   cout <= ci(NBITS);
end architecture str;
```

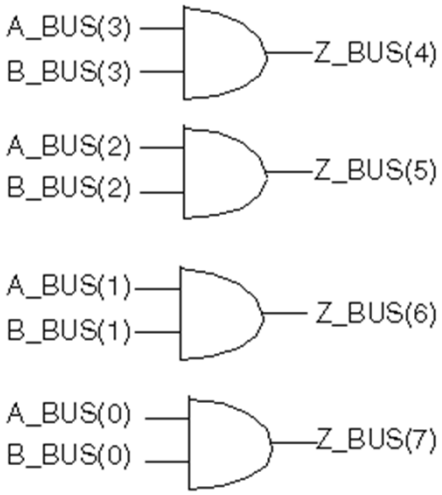
Types of Assignment for 'bit' Data Types

```
architecture EXAMPLE of ASSIGNMENT is

  signal Z_BUS :     bit_vector (3 downto 0);
  signal BIG_BUS : bit_vector (15 downto 0);

begin

  -- legal assignments:
  Z_BUS(3) <= '1';
  Z_BUS     <= "1100";

  Z_BUS     <=  b"1100";
  Z_BUS     <=  x"c";
  Z_BUS     <=  X"C";
  BIG_BUS  <=  B"0000_0001_0010_0011";

end EXAMPLE;
```

- **Single bit values are enclosed in '.'**
- **Vector values are enclosed in "..."**
  - optional base specification (default: binary)
  - values may be separated by underscores to improve readability

Logical Operations with Arrays

```
architecture EXAMPLE of LOGICAL_OP is
  signal A_BUS, B_BUS : bit_vector (3 downto 0);
  signal Z_BUS :        bit_vector (4 to 7);
begin
  Z_BUS <= A_BUS and B_BUS;
end EXAMPLE;
```



- **Operands of the same length and type**
- **Assignment via the position of the elements (according to range definition)**

Comparison Operations with Arrays

```
architecture EXAMPLE of COMPARISON is
  signal PART: bit_vector(3 downto 0);
  signal BYTE:    bit_vector(0 to 7);
begin
  PART <= "1001";
  BYTE    <= "00001111";

  COMPARE: process (PART, BYTE)
  begin
    if (PART < BYTE) then
      -- evaluated as:
      -- if (PART(3) < BYTE(0)) or
      --    ((PART(3) = BYTE(0)) and
      --     (PART(2) < BYTE(1))) or
      --    ((PART(3) = BYTE(0)) and
      --     (PART(2) = BYTE(1)) and
      --     (PART(1) < BYTE(2))) or
      ...
    -- better:
    if (( "0000"& PART) <= BYTE) then
      ...

end EXAMPLE;
```

- **Arrays:**
  - may differ in length
  - left-alignment prior to comparison
  - are compared element after element
- **No numerical interpretation (unsigned, 2-complement, etc.)**

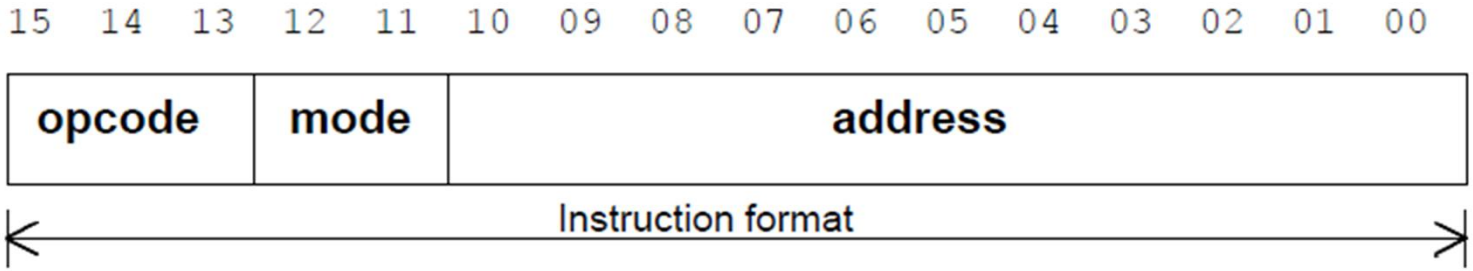
Adjust the length of arrays prior to comparison

Records

```
architecture EXAMPLE of AGGREGATE is
   type MONTH_NAME is (JAN, FEB, MAR, APR,
                       MAY, JUN, JUL, AUG,
                       SEP, OCT, NOV, DEC);
   type DATE is
        record
            DAY:       integer range 1 to 31;
            MONTH: MONTH_NAME;
            YEAR:     integer range 0 to 4000;
        end record;
   type PERSON is
        record
          NAME:        string (0 to 8);
          BIRTHDAY: DATE;
        end record;
   signal TODAY:      DATE;
   signal STUDENT_1: PERSON;
   signal STUDENT_2: PERSON;
begin
  TODAY      <= (26, JUL, 2010);
  STUDENT_1 <= (“Ali", TODAY);
  STUDENT_2 <= STUDENT_1;
  STUDENT_2.BIRTHDAY.YEAR <= 1990;
end EXAMPLE;
```

- **Elements of different type**
- **Possible assignments**
  - record <= record
  - record <= aggregate
  - record.element <= value

Records




```
TYPE opcode IS (sta, lda, add, sub, and, nop, jmp, jsr);
TYPE mode IS integer RANGE 0 TO 3;
TYPE address IS BIT_VECTOR (10 DOWNTO 0);

 TYPE instruction_format IS RECORD
    opc : opcode;
    mde : mode;
    adr : address;
 END RECORD;
```

Aliases

```
architecture EXAMPLE of ALS is
    signal DATA is bit_vector(9 downto 0);

   alias STARTBIT :   bit is DATA(9) ;
    alias MESSAGE:  bit_vector(6 downto 0) is
                DATA (8 downto 2);
    alias PARITY:       bit is DATA(1);
    alias STOPBIT:    bit is DATA(0);
    alias REVERSE:  bit_vector(1 to 10) is DATA;

    function calc_parity(data: bit_vector) return bit is
    . . .

begin
    STARTBIT       <= '0';
    MESSAGE      <= "1100011";
    PARITY          <= calc_parity(MESSAGE);
    REVERSE(10) <= '1';

end EXAMPLE;
```

- **Give new names to already existing objects**
- **Make it easier to handle complex data structures**

Aliases are not always supported by synthesis tools

Don’t Care

- در سنتز کننده ها مقایسه با ’-‘ در شرطها عموماً نتیجهٔ **FALSE** می دهد (هیچگاه مقدار سیگنال = ’-‘ نمی شود):

```
when (a = “1---”)
   ….
```

- اگر مثلاً **a = “1000”** شود شرط **TRUE** نمی شود.

- برای این منظور می توان از **stdmatch(s1, s2)** (در پکیج **numeric_std**) استفاده کرد:

```
when (std_match(a,  “1---”))
   ….
```

- همهٔ حالات ’-‘ را آزمایش می کند.
- راه بهتر (**portable**) : توصیف دقیق:

```
when (a(3) = ‘1’)
   ….
```