Programmable Logic Design
Grzegorz Budzyń
Lecture 6: Combinational & sequential circuits

Choose yourself and new technologies
Plan

• Complex types and types conversion
• Resolution functions
• Combinational circuits
• Statements examples
• Simple sequential circuits
• Regular sequential circuits
Complex types and types conversion
Types and subtypes

• Number of types available in VHDL is very limited
• It is possible and common to construct usable data subtypes
• There are a few methods of doing so:
  – 1) limiting the range of the basic data type

```vhdl
signal s: std_logic_vector(7 downto 0);
s <= (others => 'Z');
s <= "110XX011";
s(3 downto 0) <= "ZZ11";
s(6) <= '1'
```
Types and subtypes

– 2) making a new data type with limited range of the basic data type

```vhdl
  type test_type is std_logic_vector(7 downto 0);
  signal s: test_type;

  s <= (others => 'Z');
  s <= "110XX011";
  s(3 downto 0) <= "ZZ11";
  s(6) <= '1'
```
Types and subtypes

– 3a) construct a composit types: **arrays** or records

```pascal
type byte is array (7 downto 0) of bit;

type memory is array (0 to 2**16-1) of byte;

signal s_byte: Byte;
signal s_memory: Memory;

s_byte(0) -- refers to element 0
s_byte(3 DOWNTO 1) -- slice of three elements
s_memory(2**15-1 TO 2**16-1) -- slice of 2**15 elements
s_byte -- refers to the entire array

FOR i IN 1 TO 2**16 LOOP
  s_memory(i):=x"aa"; --array is initialised with $aa values
END LOOP;
```
Types and subtypes

- 3a) construct a composite types: **arrays** or records – array

---

-- init of single dimmensional array

    SIGNAL sq4: bit_nibble := ('1', '0', '1', '1');

-- init of multidimmensional array

    TYPE memory IS ARRAY(0 TO 11) OF std_logic_vector(0 TO 7);
    SIGNAL M: memory :=
        ("00000001", "00000010",
        "11001100", "00000011",
        "00001011", "00000101",
        "00001010", "00000101",
        "00001011", "11011110",
        "00000000", "00000010");

    TYPE bit_4by8 IS ARRAY(3 DOWNTO 0, 0 TO 7) OF BIT;
    SIGNAL sq_4_8: bit_4by8 :=
        (
        ('0', '0', '0', '0', '1', '1', '1', '1'),
        ('0', '0', '0', '0', '1', '1', '1', '1'),
        ('0', '0', '1', '1', '1', '1', '1', '1'),
        ('0', '1', '1', '1', '1', '1', '1', '1'));
Types and subtypes

– 3b) construct a composit types: arrays or **records**

```plaintext
TYPE Opcode IS (Add, Add_with_carry, Sub, Sub_with_carry, Complement);

TYPE Address IS RANGE 16#0000# TO 16#FFFF#;

TYPE Instruction IS RECORD
  Op_field : Opcode;
  Operand_1 : Address;
  Operand_2 : Address;
END RECORD;

signal test_sig : Instruction;

test_sig.Operand_1 <= 1234;
test_sig.OP_field <= Add;
```
Type conversion

- VHDL is a strongly typed language
- `std-logic-vector`, `unsigned`, and `signed` are treated as different data types even when all of them are defined as an array with elements of the `std-logic` data type
- A *conversion function* or *type casting* is needed to convert signals of different data types
# Data conversion table

<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>
Data conversion - example

```verbatim
library ieee;
use ieee.std_logic-1164.all;
use ieee.numeric_std.all;

signal s1,s2,s3: std_logic_vector(3 downto 0);
signal u1,u2,u3: unsigned(3 downto 0);

u1 <= s1;  -- not ok, type mismatch
u2 <= 5;   -- not ok, type mismatch
s1 <= u1;  -- not ok, type mismatch
s2 <= 5;   -- not ok, type mismatch

u1 <= unsigned(s1);  -- ok, type casting
u2 <= to_unsigned(5,4); -- ok, conversion function
s1 <= std_logic_vector(u1); -- ok, type casting
s2 <= std_logic_vector(to_unsigned (5 , 4 ));
  -- ok

u4 <= u2 + u1;  -- ok, both operands unsigned
u5 <= u2 + 1 ;  -- ok, operands unsigned and natural

s5 <= s2 + s1;  -- not ok, + undefined over the types
s6 <= s2 + 1 ;  -- not ok, + undefined over the types

s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));  -- ok
s6 <= std_logic_vector(unsigned(s2) + 1 );  -- ok
```
Data conversion – user function

```vhdl
signal X_BOOL: boolean;
signal X_STD:    std_ulogic;

function BOOL_TO_SL(X : boolean)
    return std_ulogic is
begin
    if X then
        return ('1');
    else
        return ('0');
    end if;
end BOOL_TO_SL;
...

X_STD <= X_BOOL;             --illegal
X_STD <= BOOL_TO_SL(X_BOOL);
```
Resolution functions
Resolution function

- VHDL does not allow multiple concurrent signal assignments to an *unresolved* signal
- Resolution functions are used to determine the assigned value when there are multiple signal drivers to the same signal
- Resolution functions can be user defined or called from a package
Resolution function

• The specification of a resolution function is the same as for ordinary functions with one requirement: the resolution function must be pure
  – The value returned by an **impure** function can depend on items other than just its input parameters (e.g. shared variables)
Resolution function

• Syntax:

```
function function_name (parameters) return type is

  declarations

  begin

  sequential statements

  end function function_name;
```
Resolution function

• Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of signals or in the declaration of the signal subtype.

• Standard types (BIT and BIT_VECTOR) are not resolved and it is not possible to specify multiple-source buses with these types. This is quite restrictive for typical applications, which use buses.

• Because Std_Logic and Std_Logic_Vector are resolved and can handle buses, they became the de facto industrial standard types.
Resolution function

\[ Y \leq A; \quad \text{-- in process1} \]
\[ Y \leq B; \quad \text{-- in process2} \]

**Example:**

```vhdl
package RESOLVED is
    function wired_and (V:bit_vector) return bit;
    subtype rbit is wired_and bit;
end RESOLVED;
package body RESOLVED is
    function wired_and(V:bit_vector) return bit is
    begin
        for I in V'range loop
            if V(I)='0' then return '0'; end if;
        end loop;
        return '1';
    end wired_and;
end RESOLVED;

signal y : rbit;
```
Combinational circuits
Combinational circuits

- Combinational Circuits: Circuits whose outputs depend only on the current inputs; hence they appear to combine the inputs in some way to produce the outputs.
- Combinational circuits are composed of intermediate-sized components, such as adders, comparators, and multiplexers.
Combinational circuits

- A combinational circuit can be thought of as an implementation of a Boolean function.
- It takes some inputs A, B, C, etc., and produces a unique output f(A,B,C, ... ).
- The circuit itself is just a network of the basic gates, where the information flows in one direction only (generally from left to right).
Combinational circuits
Building blocks
Building blocks

- Sequential circuits are composed of simple basic blocks like simple gates:

![Diagrams of NOT, AND, and OR gates]

Source: [1]
Building blocks

- Multiple input gates:
  - \( Q \leq x_1 \text{ AND } x_2 \text{ AND } x_3 \text{ AND } x_4 \ldots \)
Building blocks

• Multiple input gates:
  – $Q \leq x_1 \text{ OR } x_2 \text{ OR } x_3 \text{ OR } x_4...$
Building blocks

• Multiplexers:

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

entity mux is
  port (a, b, c, d: in std_logic_vector (3 downto 0);

      s: in std_logic_vector (1 downto 0);
      x: out std_logic_vector (3 downto 0));
end mux;

architecture arch_mux of mux is
begin
  with s select
  x <= a when "00",
    b when "01",
    c when "10",
    d when "11",
    others;
end arch_mux;
```

Source: [1]
Building blocks

• Decoders:

```vhdl
process(clk)
begin
  case w is
    when "000" => y <= "00000001";
    when "001" => y <= "00000010";
    when "010" => y <= "00001000";
    when "011" => y <= "00010000";
    when "100" => y <= "00100000";
    when "101" => y <= "01000000";
    when "110" => y <= "10000000";
    when "111" => y <= "11000000";
    when others => y <= "00000000";
  end case;
end process;
```
Building blocks

• Arithmetic comparators:

Process(input1 , input2 )
begin
  if ( input1 >= input2 ) then
    output <= '1';
  else
    output <= '0';
  end if;
end process;

Source: [1]
Statement examples
when...else example

```
signal_name <= value_expr_1 when boolean_expr_1 else value_expr_2 when boolean_expr_2 else ...
value_expr_n;

r <= a + b + c when m = n else a - b when m > n else c + 1;
```
when...else example
with...select example

```vhdl
with sel select
  sig <= value_expr_1 when choice_1,
       value_expr_2 when choice_2,
       value_expr_3 when choice_3,
       ... value_expr_n when others;

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;
```
with...select example
if...else example

```plaintext
if boolean_expr_1 then
    sequential_statements;
elsif boolean_expr_2 then
    sequential_statements;
elsif boolean_expr_3 then
    sequential_statements;
else
    sequential_statements;
end if;
```

```plaintext
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;
```
generic example

• VHDL provides a construct, known as a generic, to pass information into an entity and component.
• Since a generic cannot be modified inside the architecture, it functions somewhat like a constant.

```vhdl
entity entity_name is
  generic(
    generic_name: data_type := default_values;
    generic_name: data_type := default_values;
    ...
    generic_name: data_type := default_values
  )
  port(
    port_name: mode data_type;
    ...
  );
end entity_name;
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
  generic(N: integer:=4);
  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;

architecture 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 arch;
\textit{generic example}

\begin{verbatim}
signal a4, b4, sum4: unsigned(3 \texttt{downto} 0);
signal a8, b8, sum8: unsigned(7 \texttt{downto} 0);
signal a16, b16, sum16: unsigned(15 \texttt{downto} 0);
signal c4, c8, c16: std\_logic;

\begin{verbatim}
-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(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(arch)
  generic map(N=>16)
  port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));

-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
  port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
\end{verbatim}
\end{verbatim}
Barrel shifter example

library ieee;
use ieee.std_logic_1164.all;

entity barrel_shifter is
  port(
    a: in std_logic_vector(7 downto 0);
    amt: in std_logic_vector(2 downto 0);
    y: out std_logic_vector(7 downto 0)
  );
end barrel_shifter;

architecture sel_arch of barrel_shifter is
begin
  with amt select
    y <= a
    when "000",
    a(0) & a(7 downto 1)
    when "001",
    a(1 downto 0) & a(7 downto 2)
    when "010",
    a(2 downto 0) & a(7 downto 3)
    when "011",
    a(3 downto 0) & a(7 downto 4)
    when "100",
    a(4 downto 0) & a(7 downto 5)
    when "101",
    a(5 downto 0) & a(7 downto 6)
    when "110",
    a(6 downto 0) & a(7) when others; -- ///
end sel_arch;
Simple sequential circuits
Sequential circuits

- A sequential circuit is a circuit with memory, which forms the internal state of the circuit.

- Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state.
Sequential circuits

• The present state of a sequential circuit depends on a previous state and on the values of input signals.
• In the case of synchronous sequential circuits, the change of state is controlled by a clock signal.
• With asynchronous circuits, the change of state may be caused by the random change in time of an input signal.
Sequential circuits

- Sequential circuits are circuits that perform a computation in multiple steps (clock cycles)
- They compose of:
  - Memory registers
  - Combinational circuits
- Intermediate results are held in registers and transferred from register-to-register using combinational circuits
Synchronous System - Blocks

- **State register**: a collection of D FFs controlled by the same clock signal
- **Next-state logic**: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register
- **Output logic**: combinational logic that generates the output signal
Building blocks
Building blocks

• The most basic storage component in a sequential circuit is a D-type flip-flop (D FF):
  – Stores n-bits of data
  – Rising (or falling) clock latches the input
  – Output always maintains the value of stored bit
  – May have asynchronous reset

• A collection of D FFs can be grouped together to store multiple bits and is known as a register.
library ieee;
use ieee.std_logic_1164.all;

entity dff is
  port (d: in std_logic;
        clk: in std_logic;
        q: out std_logic);
end dff;

architecture example of dff is
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end example;
library ieee;
use ieee.std_logic_1164.all;

entity dff is
  port (d: in std_logic;
        rst: in std_logic;
        clk: in std_logic;
        q: out std_logic);
end dff;

architecture example of dff is
begin
  process (rst,clk)
  begin
    if rst = '0' then
      q <= '0';
    else if (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end example;

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(b) DFF with asynchronous reset
library ieee;
use ieee.std_logic_1164.all;

entity dff is
  port (d: in std_logic;
       rst: in std_logic;
       en: in std_logic;
       clk: in std_logic;
       q: out std_logic);
end dff;

architecture example of dff is
begin
  process (rst,clk,en)
  begin
    if rst = '0' then
      q <= '0';
    else if (clk'event and clk = '1') then
      if en = '1' then
        q <= d;
      else
        q <= q;
      end if;
    end if;
  end process;
end example;

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>en</th>
<th>q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>#f</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>#f</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(c) DFF with synchronous enable
library ieee;
use ieee.std_logic_1164.all;

entity d_latch is
  port (d: in std_logic;
       clk: in std_logic;
       q: out std_logic);
end dff;

architecture example of d_latch is
begin
  process (clk, d)
  begin
    if (clk = '1') then
      q <= d;
    end if;
  end process;
end example;
library ieee;
use ieee.std_logic_1164.all;

entity register8 is
  port (d: in std_logic_vector(7 downto 0);
        clk: in std_logic;
        q: out std_logic_vector(7 downto 0));
end register8;

architecture ex_reg of register8 is
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end ex_reg;
library ieee;
use ieee.std_logic_1164.all;

entity count3 is
  port (clk: in std_logic;
        reset : in std_logic;
        count: buffer Integer range 0 to 7);
end count3;

architecture count3_integer of count3 is
begin
  cnt: process (reset,clk)
  begin
    if reset = '0' then
      count <= (others => '0');
    elsif (clk'event and clk = '1') then
      count <= count + 1;
    end if;
  end process cnt;
end count3_integer;
Synchronous vs asynchronous reset

**Synchronous**

```vhdl
gate diagram
```

```vhdl
architecture example_r of dff is
begin
    process (clk, reset)
    begin
        if (reset = '1') then
            q <= '0';
        elsif rising_edge (clk) then
            q <= d;
        end if;
    end process;
end example_r;
```

**Asynchronous**

```vhdl
architecture example_r_sync of dff is
begin
    process (clk)
    begin
        if rising_edge (clk) then
            if (reset = '1') then
                q <= '0';
            else
                q <= d;
            end if;
        end if;
    end process;
end example_r_sync;
```

```vhdl
gate diagram
```
DFF - parameters

• The three main timing parameters of a D FF are:
  – $T_{cq}$ (clock-to-q delay),
  – $T_{setup}$ (setup time),
  – $T_{hold}$ (hold time).
• $T_{cq}$ is the time required to propagate the value of $d$ to $q$ at the rising edge of the clock signal. The $d$ signal must be stable around the sampling edge to prevent the FF from entering the metastable state.
• $T_{setup}$ and $T_{hold}$ specify the time intervals before or after the sampling edge.
DFF - parameters

\[ T_{\text{setup}} \quad T_{\text{hold}} \]

CLK

D can change \quad Stable \quad D can change
Maximal operating frequency

- It is very important and difficult to estimate the maximal operating frequency of the sequential circuit.
- As all storage components are grouped together, so they can be treated as single register – this simplifies the analysis.
Maximal operating frequency

\[ f_{\text{max}} = \frac{1}{T_{\text{clock}}} = \frac{1}{T_{\text{cq}} + T_{\text{comb}} + T_{\text{setup}}} \]

Where:
- \( T_{\text{cq}} \) - clock-to-q delay,
- \( T_{\text{setup}} \) - setup time,
- \( T_{\text{comb}} \) - the maximal propagation delay of next-state logic
Sequential circuits - categories

- Based on the characteristics of the next-state logic, sequential circuits can be divided into three categories:
  - Regular sequential circuit
  - FSM (finite state machine)
  - FSMD (FSM with data path).
Sequential circuits - categories

• **Regular sequential circuit.**
  – The state transitions in the circuit exhibit a “regular” pattern, as in a counter or shift register. The next-state logic is constructed primarily by a predesigned, “regular” component, such as an incrementor or shifter.
Sequential circuits - categories

• **FSM (finite state machine).**
  – The state transitions in the circuit do not exhibit a simple, repetitive pattern. The next-state logic is constructed by “random logic” and synthesized from scratch.
Sequential circuits - categories

• **FSMD (FSM with data path).**
  – The circuit consists of a regular sequential circuit and an FSM.
  – The two parts are known as a data path and a control path, and the complete circuit is known as an FSMD.
  – This type of circuit is used to implement an algorithm represented by register-transfer (RT) methodology, which describes system operation by a sequence of data transfers and manipulations among registers.
Sequential circuits - categories

• **FSMD (FSM with data path).**
  – The job of the finite state machine is to sequence operations on a datapath
Thank you for your attention
References