IE1204 Digital Design

L7: Combinational circuits, Introduction to VHDL

Elena Dubrova
KTH / ICT / ES
dubrova@kth.se
This lecture

The multiplexer (MUX)

• The multiplexer can select which input you are going to connect to the output
• "If S then X, else Y"

\[ Z = SX + \overline{S}Y \]
How can the following functions be implemented with a 2:1 multiplexer?

- \( Z = B' \) (INV)
- \( Z = AB \) (AND)
- \( Z = A + B \) (OR)
- \( Z = A \oplus B \) (XOR)

![Multiplexer Diagram]

\[ Z = SX + \overline{S}Y \]
Inverter implemented with a MUX

• Specification:
  – If input $x_0 = "1"$ then result = "0"
  – If input = "0" then result = "1"

![Inverter Circuit Diagram]
AND implemented with a MUX

• Specification:

<table>
<thead>
<tr>
<th>x₀</th>
<th>x₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ Z = \begin{cases} 1 & \text{if } x_0 = 1 \\
0 & \text{otherwise} \end{cases} \]

\[ Z = \left( \overline{x_0} \cdot \overline{x_1} \right) + \left( x_0 \cdot \overline{x_1} \right) + \left( \overline{x_0} \cdot x_1 \right) + \left( x_0 \cdot x_1 \right) \]
OR implemented with a Mux

- Specification:

\[
\begin{array}{c|cc}
\text{x}_1 & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

\[\begin{array}{c}
X_1 \\
X_0 \\
1 \\
1 \\
\end{array} \rightarrow Z\]
XOR implemented with a Mux

- Specification:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_0$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \overline{X_0} \]

\[ X_0 \]

\[ X_1 \]

\[ Z \]
Hierarchy of MUXes

IE1204 Digital Design, Autumn 2014
Implementation of larger functions with MUXes

An \((n + 1)\)-input function can be implemented with a MUX that has \(n\) select inputs!

Choose any of the inputs as address inputs ...

... And minimize/implement function for each input. Draw new Karnaugh diagrams if necessary.

\[
f = z'x' + x'y + zy
\]
Any Boolean function $f(x_n, ..., x_1, x_0)$ can be partitioned as

$$f(x_n, ..., x_1, x_0) = x_0 \cdot f(x_n, ..., x_1,1) + \overline{x}_0 \cdot f(x_n, ..., x_1,0)$$

$$= x_0 \cdot f_1(x_n, ..., x_1) + \overline{x}_0 \cdot f_0(x_n, ..., x_1)$$

The function can then be implemented with a multiplexer.
Mapping to MUXes: Shannon decomposition

- Any Boolean function \( f(x_n, ..., x_1, x_0) \) can be decomposed (recursively) as

\[
f(x_n, ..., x_1, x_0) = x_0 f_1(x_n, ..., x_1) + \overline{x_0} f_0(x_n, ..., x_1)
\]

\[
= x_1 x_0 f_{11}(x_n, ..., x_2) + x_1 \overline{x_0} f_{10}(x_n, ..., x_2)
\]

\[
+ \overline{x_1} x_0 f_{01}(x_n, ..., x_2) + \overline{x_1} \overline{x_0} f_{00}(x_n, ..., x_2)
\]
Proof

• Right-hand side:
  – If $x_0 = 1$ then the right term is zero. Then $f$ is equal to the left term.
  – If $x_0 = 0$, the left term is zero. Then $f$ is equal to the right term.

• Left-hand side:
  – if $x_0 = 1$, then $f$ is equal to $f(x_n, \ldots, x_1, 1)$ (= left term on the right-hand side)
  – if $x_0 = 0$ then $f$ is equal to $f(x_n, \ldots, x_1, 0)$ (= right term in the right-hand side)

• Left-hand side = Right-hand side
Mux circuits

But this is a memory
(ROM, RAM ...)

IE1204 Digital Design, Autumn 2014
Look-up tables (LUT)

Programmable cell

Two-input LUT

A LUT with $n$ inputs can realize all combinational functions with up to $n$ inputs
Example: XOR gate

Two-input LUT

Programmed Values

X₂
X₁

f
The simplest FPGA cell consists of a single table (e.g. Look-Up-Table - LUT), a D flip-flop and a bypass MUX.
One way to identify functions...

\[ f(x_3, x_2, x_1, x_0) = 0110100110010110 = f_{6996} \]

The functions that are stored in a LUT are usually numbered after the number that is made up of the 1's in the truth table / Karnaugh map.

\[ n \text{ inputs} \Rightarrow 2^{(2^n)} \text{ possible different Boolean functions} \]
\[ f(x_3, x_2, x_1, x_0) = "0110100110010110" = f_{6996} \]

With a LUT, all functions are realized in the same way, so all of them have the same cost.
**Decoder**

- Mostly used as address decoders
- Only one output is active when the 'enable' (En) signal is active
- The active output is selected by $a_1a_0$

<table>
<thead>
<tr>
<th>$En$</th>
<th>$a_1$</th>
<th>$a_0$</th>
<th>$y_0$</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2-to-4 decoder
The demultiplexer

- The demultiplexer has basically the same function as the decoder, but it is drawn differently.
- The input f is connected to a selected output.

<table>
<thead>
<tr>
<th>f</th>
<th>a₁</th>
<th>a₀</th>
<th>y₀</th>
<th>y₁</th>
<th>y₂</th>
<th>y₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Read-Only Memory

Decoder

Sel_0
Sel_1
\( a_0 \) .... \( a_{m-1} \)
Sel_{2m-1}

<table>
<thead>
<tr>
<th>Sel_0</th>
<th>0/1</th>
<th>0/1</th>
<th>...</th>
<th>0/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sel_1</td>
<td>0/1</td>
<td>0/1</td>
<td>...</td>
<td>0/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sel_{2m-1}</td>
<td>0/1</td>
<td>0/1</td>
<td>...</td>
<td>0/1</td>
</tr>
</tbody>
</table>

Programable bits

Three-state buffers

\( d_{n-1} \) \( d_{n-2} \) .... \( d_0 \)

IE1204 Digital Design, Autumn 2014
Encoders

- Encoder has the opposite function to a decoder, i.e. it translates $2^N$ bit input into an $N$-bit code.
  - The information is greatly reduced

Keyboard with $16 = 2^4$ keys
Priority Encoder

- A Priority Encoder gives back the address of the input with the lowest (or highest) indices that are set to 1 (or 0 depending on what you are looking for)
- If all inputs are 0, the output $z = 0$, else $z = 1$

<table>
<thead>
<tr>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$z$</th>
<th>$y_1$</th>
<th>$y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The output is well-defined even if several inputs are active at the same time.
Code converters

- A code converter translates from one code to another. Typical examples are:
  - Binary to BCD (Binary-Coded Decimal)
  - Binary to Gray code
  - 7-4-2-1 code
  - BCD to seven-segment decoder

A variant of the 7-4-2-1 code is used today to store the bar code.
BCD-to-seven segment decoder

- BCD-to-7 segment decoder consists of 7 different combinatorial circuits, one for each segment
- To get optimal circuits, all 7 functions have to be minimized simultaneously so that common logic is shared
Gray code to binary code converter

Wind direction indicators usually use Gray code to provide safe data registration.

<table>
<thead>
<tr>
<th>Binär-kod</th>
<th>Gray-kod</th>
<th>Binär-kod</th>
<th>Gray-kod</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>1000</td>
</tr>
</tbody>
</table>
Disadvantage of binary codes

Binary code, adjacent code words:
1-2 double change
3-4 triple change
5-6 double change
7-8 quadruple change!
9-A double change
B-C quadruple change!
D-E double change
F-0 quadruple change!

Can two bits change at exactly the same time?
- For safe data registration use Gray code
- For data processing use binary code
Gray code

- By changing the order of the codewords in a binary code, one construct codes in which no more than one bit is changing at a time
- Such codes are called Gray codes

0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000
Can solve "Towers of Hanoi"

Gray code can be helpful for those who want to solve the game "Towers of Hanoi", constructed by the mathematician Edouard Lucas in 1883.

According to the legend, three monks move 64 golden discs, of different sizes, from one bar to another. The discs should be moved one by one, and they must always be placed so that a smaller disc ports on a larger disc.

When the monks are done with their work the earth will go under - it will take $2^{64}$ moves, so the whole thing will probably take a while ...

According to the legend, three monks move 64 golden discs, of different sizes, from one bar to another. The discs should be moved one by one, and they must always be placed so that a smaller disc ports on a larger disc.
### Conversion between binary and Gray

**Binary → Gray:**
If Binary bit $b_n$ and bit $b_{n-1}$ are *different*, the Gray code bit $g_{n-1}$ is "1", else "0".

**Gray → Binary** (most common transformation direction):
If Binary bit $b_n$ and Gray code bit $g_{n-1}$ are *different* the Binary bit $b_{n-1}$ is "1", else "0".
Logic for Gray to Binary conversion

XOR-gate is "1" if its inputs are different!

<table>
<thead>
<tr>
<th>Binär-kod</th>
<th>Gray-kod</th>
<th>Binär-kod</th>
<th>Gray-kod</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

4 bit code converter
Gray code to Binary code

Tabell med Binärkod och Graykod.
Introduction to VHDL

• VHDL is a language used to specify the hardware
  – HDL - VHSIC Hardware Description Language
  – VHSIC - Very High Speed Integrated Circuit
  – Used mostly in Europe

• Verilog is another language used to specify the hardware
  – Used mostly in the United States
Why VHDL?

• VHDL is used to
  – verify that you have connected right by simulating the circuit
  – describe the large structures in a simple way and then generate the circuit by synthesis
  – allows for structured descriptions of a circuit

VHDL increases the level of abstraction!
There are two types of VHDL code

- VHDL for synthesis: The code is used as an input to a synthesis tool which converts it into an implementation (for example FPGA or ASIC)
- VHDL modeling and simulation code is used to describe a system at an early stage. Since the code can be simulated so you can check whether the intended functionality is correct
Entity

• The primary abstraction level in VHDL is called *Entity*

• In a behavioral description, the entity defines responses to signals and inputs

• A behavioral model is the same thing as a "black box"
  – The inside is not visible from the outside
  – Entity's behavior is defined by the black box’s functionality.
Entity (cont'd.)

- An entity describes a component's *interface* with the outside world.
- PORT-declaration indicates if it is an input or an output.
- An *Entity* is a symbol of a component.

```vhdl
ENTITY xor_gate IS
  PORT (x, y: IN BIT;
        q: OUT BIT);
END xor_gate;
```

Use English names for variable names in the code!
**VHDL Basics**

- PORT declaration establishes *interface* between the component and the outside world.
- A port declaration contains three things:
  - The name of the port
  - The direction of the port
  - Port's datatype

**Example:**

```vhdl
ENTITY test IS
  PORT ( name : direction data_typ);
END test;
```
The most common data types

• Scalars (single-variable signals)
  – Bit ("0", "1")
  – Std_logic ('U', '0', '1', 'X', 'Z', 'L', 'H', 'W', '-')
  – Integer
  – Real
  – Time

• Vectors (many-variable signals)
  – Bit_vector - vector of bits
  – STD_LOGIC_VECTOR - vector of std_logic
Architecture

- A *architecture* describes the operation of a component
- An entity can have many architectures, but only one can be active at a time
- An architecture corresponds to the component diagram or behavior

```
ARCHITECTURE behavior OF xor_gate IS
BEGIN
  q <= a xor b after 5 ns;
End behavior;
```

Code for Simulation
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Multiplexer_41 IS
  PORT(ce_n: IN std_logic; -- Chip Enable (active low)
        data_in : IN std_logic_vector(3 DOWNTO 0);
        sel: IN std_logic_vector(1 DOWNTO 0);
        data_out : OUT std_logic); -- TriState Output
END ENTITY Multiplexer_41;
ARCHITECTURE RTL OF Multiplexer_41 IS
BEGIN
    PROCESS (ce_n, data_in, sel)
    BEGIN
        IF ce_n = '1' THEN
            data_out <= 'Z';
        ELSE
            CASE sel is
                WHEN "00" => data_out <= data_in(0);
                WHEN "01" => data_out <= data_in(1);
                WHEN "10" => data_out <= data_in(2);
                WHEN "11" => data_out <= data_in(3);
                WHEN OTHERS => null;
            END CASE;
        END IF;
    END PROCESS;
END ARCHITECTURE RTL;
More on VHDL

- The study material on synthesis shows a number VHDL constructions and the resulting hardware.
- The following slides contain extra material.
  The book gives many examples and detailed explanations of VHDL.
Synthesis tool Quartus

- The course textbook contains a CD with the synthesis tool, Quartus
- You will use Quartus in Lab 3
## Summary

- Implementation of functions with MUXes
  - Shannon decomposition
- Look-up tables, ROM
- Decoder, encoder, code converters
- Introduction to VHDL
- Next lecture: BV pp. 383-418, 469-471
VHDL (Not part of the exam)
Signal declaration

Signal *declaration* is used inside architectures to declare internal (local) signals:

```plaintext
signal a, b, c, d: bit;
signal a, b, sum: bit_vector (31 downto 0);
```

Signal *assignment* is used to describe the behavior:

```plaintext
sum <= a + b; signal assignment without delay
```
VHDL description styles

• Structural
  – similar to how to connect components

• Sequential
  – similar to how to write desktop applications

• Data Flow
  – Concurrent assignments
Sequential vs Parallel Code

• There are two types of code execution in VHDL: sequential and parallel

• Hardware can then be modeled in two different ways
  – VHDL supports different levels of abstraction.

• Sequential code describes the hardware from a "programmer's" point of view and it is executed in the order which is defined

• The parallel code is executed regardless of the order. It is \textit{asynchronous}. 
Sequential style

XOR gate

Process \((x, y)\)
begin
  if \((x \neq y)\) then
    \(q \leftarrow '1';\)
  else
    \(q \leftarrow '0';\)
end if;
end process;
Data flow style

XOR gate

\[ q \leq a \text{ xor } b; \]

- Or in behavioral dataflow style

\[ q \leq '1' \text{ When } a \neq b \text{ else } "0"; \]
Structural style

XOR gate

u1: not_gate port map (x, xi);
u2: not_gate port map (y, y);
u3: and_gate port map (x, y, t3);
u4: and_gate port map (y, x, t4);
u5: or_gate port map (t3, t4, q);
Structural Code

- A component must be declared before it can be used

```
ARCHITECTURE Test OF test_entity 
    COMPONENT and_gate 
        Port (in1, in2: IN BIT; 
              out1: BIT OUT); 
    END COMPONENT; 
    ... more statements...
```

- It is necessary, unless it is not in a library somewhere
• Component *instantiation ring* connects the component interface with the signals in the architecture

```vhdl
ARCHITECTURE Test OF test_entity
    COMPONENT and_gate
        Port (in1, in2: IN BIT;
               out1: BIT OUT);
    END COMPONENT;
    SIGNAL S1, S2, S3: BIT;
BEGIN
    Gate1: and_gate PORT MAP (S1, S2, S3);
END test;
```
Structural Code (cont'd.)

- Generate-statement couples many similar elements

ENTITY adder IS
  GENERIC (N: integer)
  PORT (a, b: IN bit_vector (N-1 downto 0);
        sum: OUT bit_vector (N-1 downto 0));
END adder;
ARCHITECTURE OF structural adder IS
  COMPONENT full_adder
    PORT (a, b, cin: IN bit; cout, s: OUT bit);
  END COMPONENT;
signal c: bit_vector (N-2 downto 0);
BEGIN
  U0: full_adder PORT MAP (a(0), b(0), '0', c(0), p(0));
  UN: full_adder PORT MAP (a(n-1), b(n-1), c(n-2), OPEN, s(n-1);
END structural;
Test benches

• To test if your design works, you have to create a test bench.
• It has three functions:
  – Generating stimuli for simulation
  – Applying these stimuli to an entity to be tested
  – Comparing the output values with expected values
The test bench stimuli 1

ENTITY testbench IS END testbench;

ARCHITECTURE xor_stimuli_1 of testbench IS
  COMPONENT xor_gate
    PORT(x,y:IN bit;q:OUT bit);
  END COMPONENT;
signal x,y,u1,ut2,ut3:bit;
BEGIN
  x <= not(x) after 10 ns;
y <= not(y) after 20 ns;
  U1:xor_gate PORT MAP (x,y,ut1);
  U2:xor_gate PORT MAP (x,y,ut2);
  U3:xor_gate PORT MAP (x,y,ut3);
END example;
The test bench stimuli 2

ENTITY testbench IS END testbench;

ARCHITECTURE xor_stimuli_2 of testbench IS
  COMPONENT xor_gate
    PORT(x,y:IN bit;q:OUT bit);
  END COMPONENT;
  signal x,u1,u2,u3:bit; -- Endast en in-signal
  for U1:xor_gate use entity work.xor_gate(behave);
  for U2:xor_gate use entity work.xor_gate(data_flow);
  for U3:xor_gate use entity work.xor_gate(structural);
BEGIN
  x <= not(x) after 10 ns;
  U1:xor_gate PORT MAP (x,x,ut1);
  U2:xor_gate PORT MAP (x,x,ut2);
  U3:xor_gate PORT MAP (x,x,ut3);
END example;