IE1204 Digital Design

L7: Combinational circuits, Introduction to VHDL

Masoumeh (Azin) Ebrahimi (masebr@kth.se)

Elena Dubrova (dubrova@kth.se)

KTH / ICT / ES
This lecture

• BV 318-339, 60-65, 280-291, 341-365
PLD (eg. PAL)

Fan-in issue
FPGA (eg. Cyclone II)

Typically 50000 logic elements

Technology :  MUX tree
Multiplexer (MUX)

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

\[ Z = SX + S\bar{Y} \]
**Multiplexer (MUX)**

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

\[ Z = SX + \bar{S}Y \]
Multiplexer (MUX)

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

\[ Z = SX + \overline{S}Y \]
Implementation of functions using MUXes

How can the following functions be implemented with a 2:1 multiplexer?

- $Z = \overline{B}$ (INV)
- $Z = AB$ (AND)
- $Z = A + B$ (OR)
- $Z = A \oplus B$ (XOR)

\[ Z = SX + \overline{S}Y \]
Quickie Question ...

- How to connect the inputs of the MUX in order to implement an inverter?

Desired function: \( z = \overline{x} \)
Inverter implemented with a MUX

Specification:

if input = '1' then result <= '0';
if input = '0' then result <= '1';

\[
\begin{array}{c|c}
0 & 1 \\
\hline
1 & 0 \\
\end{array}
\]

\[Z = S \cdot X + \overline{S} \cdot Y = x_0 \cdot 0 + \overline{x}_0 \cdot 1 = \overline{x}_0\]
Quickie Question …

• *How to connect the inputs of the MUX in order to implement an AND gate?*

  Desired function: $z = xy$

![Diagram of MUX configurations](image)
AND implemented with a MUX

- Specification:

\[ Z = \overline{S\bar{X}} + SY = x_1 \cdot \overline{x_0} + \overline{x_1} \cdot \overline{0} = x_1 \cdot x_0 \]
AND implemented with a MUX

• Specification:

\[
Z = SX + \overline{S}Y = x_1 \cdot x_0 + \overline{x_1} \cdot 0 = x_1 \cdot x_0
\]
AND implemented with a MUX

- Specification:

\[ Z = SX + \overline{SY} = x_1 \cdot x_0 + \overline{x_1} \cdot 0 = x_1 \cdot x_0 \]
OR implemented with a Mux

• Specification:

\[ Z = x_1 x_0 + x_1 x_0 + x_1 x_0 = \]

\[ = \{ S X + \bar{S} Y \} = x_1 (x_0 + \bar{x}_0) + \bar{x}_1 \cdot x_0 = \]

\[ = x_1 \cdot 1 + x_1 \cdot x_0 \]
OR implemented with a Mux

- Specification:

\[
Z = x_1 x_0 + \overline{x_1} x_0 + \overline{x_1} x_0 =
\]

\[
= \{SX + \overline{SY}\} = x_1 \overline{x_0} + x_0 \overline{x_1} \cdot x_0 =
\]

\[
= x_1 \overline{1} + x_1 \cdot x_0
\]
OR implemented with a Mux

- Specification:

\[ Z = x_1 x_0 + \overline{x_1} x_0 + x_1 \overline{x_0} = \]
\[ = \{ Sx + \overline{Sy} \} = x_1 (x_0 + \overline{x_0}) + \overline{x_1} \cdot x_0 = \]
\[ = x_1 \cdot 1 + x_1 \cdot \overline{x_0} \]
XOR implemented with a Mux

- Specification:

\[
\begin{array}{c|c|c}
X & 0 & 1 \\
\hline
x_0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

\[
Z = SX + \bar{S}Y = x_1 \cdot \bar{x}_0 + x_1 \cdot x_0 = x_1 \oplus x_0
\]
XOR implemented with a Mux

• Specification:

\[
Z = SX + \overline{S}Y = \\
= x_1 \cdot \overline{x_0} + x_1 \cdot x_0 = x_1 \oplus x_0
\]
XOR implemented with a Mux

• Specification:

\[
Z = SX + \overline{SY} = \\
= x_1 \cdot \overline{x_0} + x_1 \cdot \overline{x_0} = x_1 \oplus x_0
\]
Hierarchy of MUXes

\[ X_{11} \quad X_{10} \quad X_{01} \quad X_{00} \]
\[ S_0 \quad 0 \quad S_1 \quad 0 \]
\[ Z \]

\[ X_{11} \quad X_{10} \quad X_{01} \quad X_{00} \]
\[ S_1S_0 \quad 00 \]
\[ Z \]
Implementation of larger functions with MUXes

Choose any of the inputs as address inputs ...

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

An \((n + 1)\)-input function can be implemented with a MUX that has \(n\) select inputs!
Mapping into MUXes: Shannon decomposition (BV 6.1.2)

- Any Boolean function $f(x_n, \ldots, x_1, x_0)$ can be partitioned as
  
  $f(x_n, \ldots, x_1, x_0) = x_0 f_1(x_n, \ldots, x_1, 1) + \overline{x}_0 f_0(x_n, \ldots, x_1, 0)$

- The function can then be implemented with a multiplexer
Any Boolean function \( f(x_n, ..., x_1, x_0) \) can be partitioned as

\[
f(x_n, ..., x_1, x_0) = x_0 f(x_n, ..., x_1, 1) + \overline{x_0} f(x_n, ..., x_1, 0)
\]

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

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

\[
f(x_n, \ldots, x_1, x_0) = x_0 f_1(x_n, \ldots, x_1, 1) + \overline{x}_0 f_0(x_n, \ldots, x_1, 0) \\
= x_1 x_0 f_{11}(x_n, \ldots, x_2, 1, 1) + x_1 \overline{x}_0 f_{10}(x_n, \ldots, x_2, 1, 0) \\
+ \overline{x}_1 x_0 f_{01}(x_n, \ldots, x_2, 0, 1) + \overline{x}_1 \overline{x}_0 f_{00}(x_n, \ldots, x_2, 0, 0)
\]
Proof

Left-hand side

\[ f(x_n, \ldots, x_1, x_0) = x_0 \cdot f(x_n, \ldots, x_1, 1) + x_0 \cdot f(x_n, \ldots, x_1, 0) \]

Right-hand side

- **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 2015 27
Look-up tables (LUT)

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

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Two-input LUT
The simplest FPGA cell consists of a single table (e.g. Look-Up-Table - LUT), a D flip-flop and a bypass MUX.

D-flipflop will be explained soon in this course.
One way to identify functions...

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

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 inputs => \(2^{2^n}\) possible different Boolean functions
LUT function number

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

<table>
<thead>
<tr>
<th>( x_3 x_2 x_1 x_0 )</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Odd parity

\[ f_{6996} = x_3 \oplus x_2 \oplus x_1 \oplus x_0 \]

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
Demultiplexer

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

<table>
<thead>
<tr>
<th>$I$</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>
Read-Only Memory

Decoder

Sel<sub>0</sub>

Sel<sub>1</sub>

a<sub>0</sub>

a<sub>1</sub>

...

a<sub>m-1</sub>

Sel<sub>2m-1</sub>

0/1

0/1

0/1

0/1

0/1

Programable bits

Three-state buffers

En

d<sub>n-1</sub>

d<sub>n-2</sub>

...

d<sub>0</sub>
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

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

$2^n$ inputs

$w_0 \quad \vdots \quad w_{2n-1}$

$n$ outputs

$y_0 \quad \vdots \quad y_{n-1}$
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.
Overview

Multiplexer

Demultiplexer

Encoder

Decoder

- 2^n inputs
- n outputs
- Used in priority encoders

Equivalent

IE1204 Digital Design, Autumn 2015
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

<table>
<thead>
<tr>
<th>$w_3$</th>
<th>$w_2$</th>
<th>$w_1$</th>
<th>$w_0$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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
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!

4 bit code converter
Gray code to Binary code

<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>

Tabell med Binärkod och Graykod.
Logic for Gray to Binary conversion

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

### 4 bit code converter
Gray code to Binary code

<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>

Tabell med Binärikod 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
The entity describes the ports to the outside of the circuit. The circuit as a block.
Architecture describes the function inside the circuit.
Entity

- 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.

```
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:

  ```vhd
  ENTITY test IS
      PORT ( name : direction data_type);
  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
An 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 of VHDL constructs 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:

    signal a, b, c, d: bit;

    signal a, b, sum: bit_vector (31 downto 0);

Signal-assignment is used to describe the behavior:

    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 *asynchronous*. 

IE1204 Digital Design, Autumn 2015 63
Sequential style

XOR gate

```
Process (x, y)
begin
  if (x /= y) then
    q <= '1';
  else
    q <= '0';
  end if;
end process;
```
Data flow style

XOR gate

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

- Or in behavioral dataflow style

\[
q \leftarrow '1' \text{ When } a \neq b \text{ else } "0";
\]
u1: not_gate port map (x,xi);
u2: not_gate port map (y,yi);
u3: and_gate port map (xi,y,t3);
u4: and_gate port map (yi,x,t4);
u5: or_gate port map (t3,t4,q);
A component must be declared before it can be used.

```vhdl
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

---

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;
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
    G0: for i in 1 to N-2 Generate
        U0: full_adder PORT MAP (a (i), b (i), c (i-1), c (i), p (i));
    end Generate; - G0
    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;
Generate n-bit adder

Five lines of code generates the ripple-carry n-bit adder from F5!
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:bit;
BEGIN
  x <= not(x) after 10 ns;
  y <= not(y) after 20 ns;
  U1:xor_gate PORT MAP (x,y,U1);
END example;

The ENTITY is empty!
The circuit under test is used as a component of the test bench program
Here are the test signals generated
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;
A test bench can mark when the desired events occur during the execution.

Or mark when unwanted events occur

The result of a run with a test bench can be saved in a file, as proof that everything is ok - or as a troubleshooting aid if it did not go well.