*Lab session #3*

**Activity #1: Hamming distance**

A **Hamming** distance of two words is the number of bit positions in which the two words differ. For example, the Hamming distance of two 8-bit words "00010011" and "10010010" is 2 since the bits at position 0 (LSB) and position 7 (MSB) are different. The Hamming distance is used in some error correction and data compression applications. Design a combinational circuit that calculates the Hamming distance of two 8-bit inputs (VHDL std_logic_vectors). The output should be a VHDL std_logic_vector as well. Design a suitable testbench.

Suggestion: proceed in 2 steps:

1. determine which pairs of bits differ and mark each of them as 1
2. count the number of 1s in the result of the previous step by stages. In the first stage of the circuit, divide the 8 bits into 4 pairs and add the 1s in each pair using 4 adders. In the second stage add 2 by 2 the 4 results of the previous step using 2 adders. In the third stage add the 2 results of the previous step to obtain the final result.

Provide 2 versions of the design resorting:

1. either to a behavioral or dataflow VHDL description style
2. or to a structural VHDL description style.

Create a testbench to simulate the circuit on relevant inputs.

**Activity #2: Multi-function right barrel shifter**

A barrel shifter is a circuit that can shift input data by any number of positions compatible with its size. Shifting operations can be done in either the left or right direction. This exercise is restricted to right shifts. Right shifts are divided into:

1. ***rotate-right n positions:*** the n least significant bits are shifted into the n most significant bits
   example: ror(1011, 2) returns 1110
2. ***logic shift-right n positions:*** n 0's are shifted into the n most significant bits
   example: srl(1011, 3) returns 0001
3. ***arithmetic shift-right n positions:*** the sign bit is shifted into the n most significant bits
   example: sra(1011, 1) returns 1101

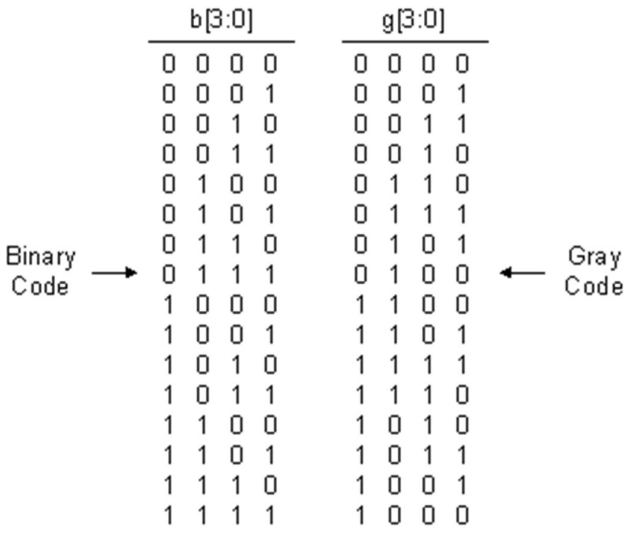
Design an 8-bit shifting circuit that can perform rotate right, logic shift right or arithmetic shift right. A control signal *lar* (for logic shift lar = "00", for arithmetic shift lar = "01" and for rotate lar = "10") specifies the operation to be performed. Another control signal *amt* (for amount) specifies the number of positions to be rotated or shifted (from 0 to 7). Resort to a behavioural or dataflow description style.

NOTE: do not use VHDL shift operators (e.g. SRL, ROL, ROR, etc.).

Create a testbench to simulate the circuit on relevant inputs.

**Activity #3: Gray code incrementer**

The ***Gray code*** is a special kind of code in that **only** a single bit changes between any 2 successive code words. It minimizes the number of transitions when a signal switches between successive words. A 4-bit Gray code and its corresponding binary code are shown below:




A Gray code incrementer is a circuit that generates the next word in Gray code. If the number of bits in the code is fixed, a straightforward solution is to draw the truth table and to translate it into a selected signal assignment statement. However, this approach is not scalable and there is no easy algorithm to derive the next Gray code word directly. Since an algorithm exists for conversion from Gray to binary code, one possible approach is to derive it indirectly by using a binary incrementer. Design a combinational circuit that increments by 1 the N-bit Gray code word on its inputs. Resort to a dataflow description style. Use the generic feature of VHDL.

Suggestion: work in 3 steps:
1. convert a Gray code word to the corresponding binary word
2. increment the binary word
3. convert the result back to the Gray code word.

The binary-to-Gray conversion algorithm is based on the following observation: the i-th bit $g_i$ of the Gray code word is '1' if the i-th $b_i$ bit and the (i+l)-th bit $b_{i+1}$ in the corresponding binary word are different:

$$g_i = b_i \oplus b_{i+1}$$

Resorting to your knowledge of Boolean algebra, express $b_i$ as a function of $g_i$ and of $b_{i+1}$ to formalize the Gray-to-binary conversion algorithm.

Create a testbench to simulate the circuit on relevant inputs.

**Activity #4: ALU**

An **ALU** performs a set of arithmetic and logical operations. The function table of a simple ALU is shown below. The inputs include two N-bit std_logic_vector signals , *src0* and *src1*, and a 3-bit control signal, *ctrl*, which specifies the function to be performed. The output is the N-bit *result* std_logic_vector signal. There are five functions, including three arithmetic operations (increment by 1, addition and subtraction) and two logical operations (bitwise AND and OR operations). Furthermore, we assume that the input and output are interpreted as signed integers when an arithmetic function is selected.

| ctrl | | | result |
|---|---|---|---|
| 0 | - | - | src0 + 1 |
| 1 | 0 | 0 | src0 + src1 |
| 1 | 0 | 1 | src0 - src1 |
| 1 | 1 | 0 | src0 AND src1 |
| 1 | 1 | 1 | src0 OR src1 |

Describe the ALU resorting to the dataflow VHDL description style and using the generic feature of the language. Create a suitable testbench with assertions to test the ALU on relevant values.