VHDL program to simulate 4 bit binary adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Adder is
  port
  (
    nibble1, nibble2 : in unsigned(3 downto 0);

    sum      : out unsigned(3 downto 0);
    carry_out : out std_logic
  );
end entity Adder;

architecture Behavioral of Adder is
  signal temp : unsigned(4 downto 0);
begin
  temp <= ("0" & nibble1) + nibble2;
  -- OR use the following syntax:
  -- temp <= ('0' & nibble1) + ('0' & nibble2);

  sum      <= temp(3 downto 0);
  carry_out <= temp(4);
end architecture Behavioral;
```

VHDL code to implement 8×1 mux

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY MUX8_1 IS
PORT(DIN:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
SEL:IN STD_LOGIC_VECTOR(2 DOWNTO 0);
DOUT:OUT STD_LOGIC);
END MUX8_1;
ARCHITECTURE BEH123 OF MUX8_1 IS
BEGIN
PROCESS(DIN,SEL)
BEGIN
CASE SEL IS
WHEN"000"=>DOUT<=DIN(0);
WHEN"001"=>DOUT<=DIN(1);
WHEN"010"=>DOUT<=DIN(2);
WHEN"011"=>DOUT<=DIN(3);
WHEN"100"=>DOUT<=DIN(4);
WHEN"101"=>DOUT<=DIN(5);
WHEN"110"=>DOUT<=DIN(6);
WHEN"111"=>DOUT<=DIN(7);
WHEN OTHERS=>
DOUT<='Z';
END CASE;
END PROCESS;
END BEH123;
```

VHDL code for Shift register (Parallel input parallel output)

```
#vhdl code for parallel input parallel output
library ieee;
use ieee.std_logic_1164.all;

entity pipo is
 port(
 clk : in std_logic;
 D: in std_logic_vector(3 downto 0);
 Q: out std_logic_vector(3 downto 0)
 );
end pipo;

architecture arch of pipo is
 begin
  process (clk)
 begin
 if (CLK'event and CLK='1') then
 Q <= D;
 end if;
 end process;
end arch;
```

VHDL code for octal to binary encoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity OCT2BIN is
   Port ( D : in std_logic_vector (7 downto 0);
          Y : out std_logic_vector (2 downto 0));
end OCT2BIN;
architecture Behavioral of OCT2BIN is
begin
          Y(0) <= D(1) OR D(3) OR D(5) OR D(7);
          Y(1) <= D(2) OR D(3) OR D(6) OR D(7);
          Y(2) <= D(4) OR D(5) OR D(6) OR D(7);
end Behavioral;
```

VHDL Modeling styles:

An architecture can be written in one of three basic coding styles:

1. Dataflow
2. Behavioral
3. Sturcture

The difference between these styles is based on the type of concurrent statements used:

- A dataflow architecture uses only concurrent signal assignment statements
- A behavioral architecture uses only process statements
- A structure architecture uses only component instantiation statements.

Instead of writing an architecture exclusively in one of these styles, we mix two or more, resulting in a mixed sytle.

Dataflow Syle of Modelling

1. Dataflow style describes a system in terms of how data flows through the system. data dependencies in the description match those in a typical hardware implementation
2. A dataflow description directly imples a corresponding gate-level implementation
3. Dataflow descriptions consists of one or more concurrent signal assignment statements.

Dataflow style half-adder description

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

entity half_adder is
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
  end half_adder;

architecture dataflow of half_adder is
  begin
  sum <= a xor b;
  carry_out <= a and b;
end dataflow;
```

Behavioral Syle of modeling

1. A behavioral description describes a system's behavior or function in an algorithmic fashion
2. Behavioral style is the most abstract style. The description is abstract in the sense that it does not directly imply a particular gate-level implementation
3. Behavioral style consists of one or more process statements. Each process statement is a single concurrent statement that itself contains one or more sequential statements.
4. Sequential statements are executed sequentially by a simulator, the same as the execution of sequential statements in a conventional programming language.

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

entity half_adder is
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
  end half_adder;

architecture behavior of half_adder is
  begin
    ha: process (a, b)
    begin
      if a = 1 then
        sum <= not b;
        carry_out <= b;
      else
        sum <= b;
        carry_out <= 0;
      end if;
    end process ha;
```

```
end behavior;
```

structural Style of Modelling

1. In structural sytle of modeling, an entity is described as a set of interconnected components.
2. The top-level design entity's architecture describes the interconnection of lower-level design entities. Each lower-level design entity can, in turn, be described as an interconnection of design entities at the next-lower level, and so on.
3. Sturecture style is most useful and efficient when a complex system is described as an interconnection of moderately complex design entities. This approach allows each edesign entity to be independently designed and verified before being used in the higher-level description.

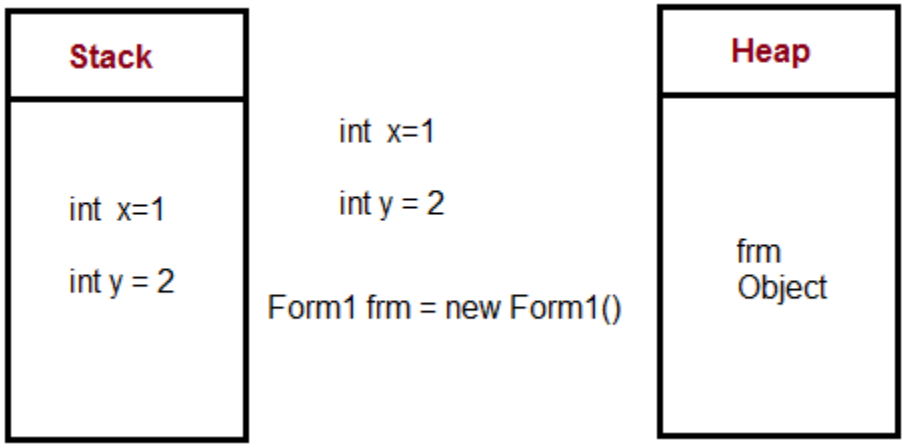
```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is                -- Entity declaration for half adder
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
end half_adder;

architecture structure of half_adder is     -- Architecture body for half adder

  component xor_gate                -- xor component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

  component and_gate                -- and component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

begin
   u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
   u2: and_gate port map (i1 => a, i2 => b, o1 => carry_out);
-- We can also use Positional Association
--  => u1: xor_gate port map (a, b, sum);
--  => u2: and_gate port map (a, b, carry_out);
end structure;
```

Difference between stack and heap

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM .

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled. When a function or a method calls another function which in turns calls another function etc., the execution of all those functions remains suspended until the very last function returns its value. The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory . Element of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.



You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
In a multi-threaded situation each thread will have its own completely independent stack but they will share the heap. Stack is thread specific and Heap is application specific. The stack is important to consider in exception handling and thread executions.

Programmer view's on microcontroller (what are his/her concern)

A programmer writes the program instructions that carry out the desired functionality on the general-purpose processor. The programmer may not actually need to know detailed information about the processor's architecture or operation, but instead may deal with an architecture abstraction, which hides much of the details.

The level of abstraction depends on the level of programming. We can distinguish between two levels of programming. The first is **assembly-language programming** and second is **structured language programming.** The structural programming language hides much of the hardware details and however to program a controller in assembly language the programmer need to know about architecture as well as instruction sets that are supported to a particular controller. Programmer view microcontroller on the basis of following criteria:

1. Instruction Sets

The assembly-language programmer must know the processor's instruction set. The instruction typically has two parts, an opcode and operand fields. The opcode specifies the operation to take place during the instruction. We can classify instructions into three catagories:

- Data transfer instruction
- Arithmetic instructions
- Branching instructions.

An operand field specifies the location of the actual data that takes part in an operation. Source operands serve as input to the location, while a destination operand stores the output.

The structured-language programmer would not need to know the instruction set of the processor. However, some portions of the code which deals with input/output operations need to be written in lower language. Such section of code which deals with input and output operation is code drivers.

2. Program and Data Memory Space

The embedded system programmer must be aware of the size of the available memory for program and for data. A particular processor may have 64K program and 64K data space. The programmer must not exceed these limits.

3. Registers

Assembly-language programmer must know how many registers are available for general purpose data storage. They must also be familiar with other registers that have special functions.

Special-function registers must be known by both the assembly-language and the structured-language programmer, since such registers are used for configuring built-in timers, counters serial communication and other peripherals.

4. I/O

The programmer should be aware of the processor's input and output facilities, with which the processor communicates with other device. One common I/O facility is parallel I/O, in which the programmer can read or write a port by reading or writing a special function register. Another common I/O facility is a system bus, consisting of address and data ports that are automatically activated by certain address or instructions.

5. Interrupts

An interrupt causes the processor to suspend execution of the main program and jumps to an interrupt service routine (ISR) that fulfils a special, short-term processing need. The processor stores the current PC (Program counter) and sets it to the address of the ISR. After ISR completes its execution processor resumes by restoring the PC. The programmer must be aware of the types of interrupts supported by the processor and must write ISR when necessary.