4X1 MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY Mux IS
Generic (n: integer := 3);
PORT(
        I3: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
        I2: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
        I1: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
        I0: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
        S: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        O: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0)
);
END Mux;
ARCHITECTURE MuxArch OF Mux IS
BEGIN
         O <= I0 WHEN S=“00” ELSE
        I1 WHEN S=“01” ELSE
        I2 WHEN S=“10” ELSE
        I3 WHEN S=“11” ELSE "ZZZ";
END MuxArch;
```

2x1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY Mux2 IS
Generic (n: integer := 8);
PORT(
        I0,I1: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
        S: IN STD_LOGIC ; y: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
);
END Mux;
ARCHITECTURE MuxArch OF Mux2 IS
        BEGIN
        y <= I1 WHEN S ELSE I0;
END MuxArch;
```

Sign extend

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity signext is
port (
        a : in STD_LOGIC_VECTOR(15 downto 0);
        y : out STD_LOGIC_VECTOR(31 downto 0)
);
end signext;
architecture beave of signext is
begin
        y <= X"FFFF" & a when a(15) = '1' else X"0000" & a;
end beave;
```

4-bit parallel adder

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

entity adder is
   port (
      A : in STD_LOGIC_VECTOR(3 downto 0);
      B : in STD_LOGIC_VECTOR(3 downto 0);
      Cin : in STD_LOGIC;
      S : out STD_LOGIC_VECTOR(3 downto 0);
      Carry : out STD_LOGIC
   );
end adder;

architecture Behavioral of adder is
   signal Tmp : STD_LOGIC_VECTOR(4 downto 0);
begin
   Tmp <= ('0' & A) + ('0' & B) + Cin;
   S <= Tmp(3 downto 0);
   Carry <= Tmp(4);
end Behavioral;
```

32-bit alu

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

entity alu32 is
   port (
      A, B : in STD_LOGIC_VECTOR(31 downto 0);
      F : in STD_LOGIC_VECTOR(2 downto 0);
      Y : out STD_LOGIC_VECTOR(31 downto 0)
   );
end alu32;

architecture synth of alu32 is
   signal S, BB : STD_LOGIC_VECTOR(31 downto 0);
   signal Bout : STD_LOGIC_VECTOR(31 downto 0);
begin
   Bout <= (not B) when F(2) = '1' else B;
   S <= A + Bout + F(2);

   Y <= A and Bout when F(1 downto 0) = "00" else
        A or Bout when F(1 downto 0) = "01" else
        S when F(1 downto 0) = "10" else
        ("00000000000000000000000000000000" & S(31)) when F(1 downto 0) = "11" else
        X"00000000";
end synth;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ALU is
   Port ( A      : in  STD_LOGIC_VECTOR(7 downto 0);  -- First Operand
          B      : in  STD_LOGIC_VECTOR(7 downto 0);  -- Second Operand
          opcode  : in  STD_LOGIC_VECTOR(3 downto 0);  -- Operation Select
          result  : out STD_LOGIC_VECTOR(7 downto 0);  -- ALU Result
          zero    : out STD_LOGIC                      -- Zero Flag
          );
end ALU;

architecture Behavioral of ALU is
begin
   process(A, B, opcode)
   begin
      case opcode is
         -- Arithmetic Operations
         when "0000" =>  -- ADD
            result <= A + B;
         when "0001" =>  -- SUB
            result <= A - B;

         -- Logical Operations
         when "0010" =>  -- AND
            result <= A and B;
         when "0011" =>  -- OR
            result <= A or B;
         when "0100" =>  -- XOR
            result <= A xor B;

         -- Shift Operations
         when "0101" =>  -- Logical Shift Left
            result <= A sll 1;
         when "0110" =>  -- Logical Shift Right
            result <= A srl 1;

         -- Default case
         when others =>
            result <= (others => '0');
      end case;

      -- Zero Flag: If result is zero, set zero flag
      if result = "00000000" then
         zero <= '1';
      else
         zero <= '0';
      end if;
   end process;
end Behavioral;
```

DFF asynchronous reset

```
ENTITY DFF IS
        PORT (
                clk: IN STD_LOGIC;
                D: IN STD_LOGIC;
                rst: IN STD_LOGIC;
                Q: OUT STD_LOGIC
        );
END DFF;
ARCHITECTURE Behavioral OF DFF IS
BEGIN
        PROCESS (clk, rst)
                BEGIN
                        IF (rst = '1') THEN Q <= '0' ;
                        ELSIF (clk'EVENT and clk = '1') THEN Q <= D ;
                        END IF;
        END PROCESS;
END Behavioral;
```

Synchronous reset:

```
        IF (clk'EVENT and clk = '1') THEN
                IF (rst = '1') THEN
                        Q <= '0' ;
                ELSE
                        Q <= D ;
                END IF;
        END IF;
```

Flopr (entity)

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

entity FlopRegister is
   generic (n : NATURAL := 32);
   port (
      clk    : in  STD_LOGIC;
      reset  : in  STD_LOGIC;
      data_in : in  STD_LOGIC_VECTOR(n-1 downto 0);
      data_out : out STD_LOGIC_VECTOR(n-1 downto 0)
   );
end FlopRegister;

architecture Behavioral of FlopRegister is
begin
   process(clk, reset)
   begin
      if reset = '1' then
         data_out <= (others => '0'); -- Reset to all zeros
      elsif rising_edge(clk) then
         data_out <= data_in; -- Latch input data on rising clock edge
      end if;
   end process;
end Behavioral;
```

flopr(component)

```
PACKAGE MyPackage is
Component flopr is
           generic (n : NATURAL := 32);
           port(clk, reset: in STD_LOGIC;
           d: in STD_LOGIC_VECTOR(n-1 downto 0);
            q: out STD_LOGIC_VECTOR(n-1 downto 0));
end Component;
end Behavioral;
```

Main Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.MyPackage.all;

entity FloprMainModule is
   generic (n : NATURAL := 32); -- Width of the registers
   port (
      clk : in STD_LOGIC; -- Clock signal
      C   : in STD_LOGIC; -- Clear control for R1
      L   : in STD_LOGIC; -- Load control for R2
      R1  : out STD_LOGIC_VECTOR(n-1 downto 0); -- Output of R1
      R2  : out STD_LOGIC_VECTOR(n-1 downto 0)  -- Output of R2
```

```
    );
end FloprMainModule;

architecture Behavioral of FloprMainModule is
    -- Signal declarations
    signal Tmp1 : STD_LOGIC_VECTOR(n-1 downto 0); -- Input for R1
    signal Tmp2 : STD_LOGIC_VECTOR(n-1 downto 0); -- Output of R1 and input for R2

    -- Component declaration for flopr
    component flopr is
        generic (n : NATURAL := 32); -- Width of the register
        port (
            clk      : in  STD_LOGIC;
            reset    : in  STD_LOGIC;
            data_in  : in  STD_LOGIC_VECTOR(n-1 downto 0);
            data_out : out STD_LOGIC_VECTOR(n-1 downto 0)
        );
    end component;
begin
    -- Logic for R1: Clear R1 when C is active
    Tmp1 <= (others => '0') when C = '1' else Tmp2;

    -- Instantiate flopr for R1
    R1Map: flopr
        generic map (n => n)
        port map (
            clk      => clk,
            reset    => C,
            data_in  => Tmp1,
            data_out => Tmp2
        );

    -- Instantiate flopr for R2: Load R1 value into R2 when L is active
    R2Map: flopr
        generic map (n => n)
        port map (
            clk      => clk,
            reset    => '0', -- No reset signal for R2
            data_in  => Tmp2 when L = '1' else R2,
            data_out => R2
        );

    -- Output R1
    R1 <= Tmp2;
end Behavioral;
```

JK ff

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

entity JK_FF is
   port (
      clk   : in  STD_LOGIC;            -- Clock signal
      rst   : in  STD_LOGIC;            -- Reset signal
      J     : in  STD_LOGIC;            -- J input
      K     : in  STD_LOGIC;            -- K input
      Q     : out STD_LOGIC;            -- Output Q
      Qbar  : out STD_LOGIC             -- Complement of Q
   );
end JK_FF;

architecture Behavioral of JK_FF is
   signal current_state : STD_LOGIC;      -- Internal state signal
   signal jk_input      : STD_LOGIC_VECTOR(1 downto 0); -- Combined J and K input
begin
   -- Combine J and K inputs into a vector
   jk_input <= J & K;

   -- Process to implement JK Flip-Flop behavior
   process (clk, rst)
   begin
      if rst = '1' then
         current_state <= '0'; -- Reset state to 0
      elsif rising_edge(clk) then
         case jk_input is
            when "11" =>
               current_state <= not current_state; -- Toggle state
            when "10" =>
               current_state <= '1'; -- Set state to 1
            when "01" =>
               current_state <= '0'; -- Reset state to 0
            when others =>
               null; -- No change
         end case;
```

```
        end if;
    end process;

    -- Assign outputs
    Q <= current_state;
    Qbar <= not current_state;
end Behavioral;
```