Lösningsförslag övningsuppgifter 2025-04-04

1. Du ska konstruera en heladderare innehållande tre insignaler A, B och C_{IN} *(carry in)* samt två utsignaler S *(sum)* och C_{OUT} *(carry out).* Summan av insignalerna ska tilldelas till utsignal S. Eftersom S bara utgörs av en bit kan det hända att vi får en carry-bit, som ska tilldelas till utsignal C_{OUT}.

Man kan också tänka att utsignalerna tillsammans utgör ett 2-bitars tal, där C_{OUT} utgör MSB (mest signifikant bit), vilket kan uttryckas i en ekvation såsom visas nedan:

$$\{C_{OUT}, S\} = A + B + C_{IN}$$

Därmed gäller att

$$\{C_{OUT}, S\} = \begin{cases} \{0,0\}\ om\ A + B + C_{IN} = 0 \\ \{0,1\}\ om\ A + B + C_{IN} = 1 \\ \{1,0\}\ om\ A + B + C_{IN} = 2 \\ \{1,1\}\ om\ A + B + C_{IN} = 3 \end{cases}$$

a) Rita upp en sanningstabell för heladderaren. Härled sedan logiska ekvationer för heladderarens utsignaler S och C_{OUT}.

b) Realisera grindnätet i CircuitVerse. Kontrollera att det fungerar korrekt.

a) Implementera konstruktionen i VHDL via en modul döpt *full_adder.* Välj FPGA-kort Terasic DE0 (enhet 5CEBA4F23C7). Toppmodulen ska ha samma namn som projektet.

b) Skapa en testbänk döpt *full_adder_tb* och simulera samtliga åtta kombinationer 000 – 111 av insignalerna under 10 ns var. Den totala simuleringstiden ska alltså uppgå till 80 ns. Validera konstruktionen i ModelSim.

c) Verifiera konstruktionen på ett FPGA-kort. Anslut insignaler A, B samt C_{IN} till var sin slide-switch samt utsignaler S samt C_{OUT} till var sin lysdiod, se databladet för PIN-nummer (finns här).

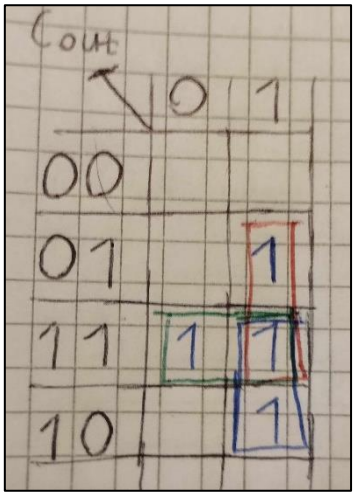
Lösning

Vi börjar med att rita upp heladderarens sanningstabell utefter beskrivningen:

ABC_{in}	$C_{out}Sum$
000	00
001	01
010	01
011	10
100	01
101	10
110	10
111	11

Tabell 1: Sanningstabell för heladderaren.

Vi ritar om sanningstabellen ovan till Karnaugh-diagram för utsignaler C_{out} och Sum.
Vi börjar med att rita Karnaugh-diagram för utsignal C_{out} såsom visas till höger.



Figur 1: Karnaugh-diagram för heladderarens utsignal C_{out}.

Vi placerar insignaler AB i y-led samt insignaler C_{in} i x-led. Vi placerar AB i 2-bitars Grey-kod, alltså i ordningsföljden 00, 01, 11, 10, så att samtliga celler har en bit gemensam med samtliga intilliggande celler, inklusive ytterkanterna.

Vi lägger till ettor i de celler där C_{out} = 1. I sanningstabellen ser vi att C_{out} = 1 för kombinationer ABC_{in} = 011, 101, 110 samt 111. Vi struntar i att skriva ut nollor i övriga celler, då vi enbart är intresserade av ettorna.

Vi noterar i Karnaugh-diagrammet ovan att vi får tre block innehållande ettor. Vi ringar in dessa block med röd, grön respektive blå färg. De ettor som är inringade i rött har gemensamt att BC_{in} = 11. De ettor som är inringade i grönt har gemensamt att AB = 11. De ettor som är inringade i blått har gemensamt att AC_{in} = 11.

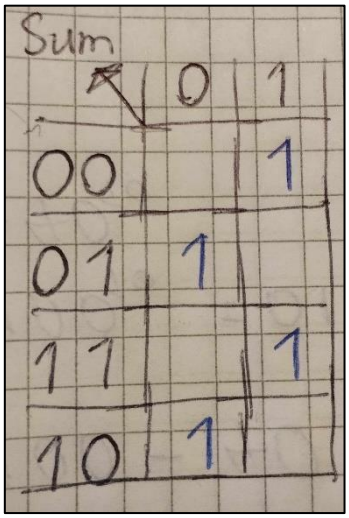
Därmed gället att C_{out} = 1 om AB = 11, AC_{in} = 11 eller BC_{in} =11, vilket på boolesk algebra skrivs enligt nedan:

$$C_{out} = AB + AC_{in} + BC_{in}$$

Vi noterar att de två sista termerna har C_{in} gemensamt. Ekvationen ovan kan därmed omvandlas till

$$C_{out} = AB + (A + B)C_{in}$$

Vi ritar sedan Karnaugh-diagram för utsignal Sum såsom visas till höger.



Figur 2: Karnaugh-diagram för heladderarens utsignal Sum.

Vi placerar insignaler AB i y-led samt insignaler C_{in} i x-led. Vi placerar AB i 2-bitars Grey-kod, alltså i ordningsföljden 00, 01, 11, 10, så att samtliga celler har en bit gemensam med samtliga intilliggande celler, inklusive ytterkanterna.

Vi lägger till ettor i de celler där Sum = 1. I sanningstabellen ser vi att S = 1 för kombinationer ABC_{in} = 001, 010, 100 samt 111. Återigen struntar vi i att skriva ut nollor i övriga celler, då vi enbart är intresserade av ettorna.

Vi noterar i Karnaugh-diagrammet ovan att vi inte får några vertikala eller horisontella block. Att ettorna är placerade sicksack indikerar dock ett XOR- eller XNOR-mönster. Vi får lösa ekvationen algebraiskt i stället.

Vi börjar med att skriva ut att Sum = 1 då ABC_{in} = 001, 010, 100 samt 1111 algebraiskt:

$$Sum = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$$

Vi noterar att de två första termerna har A' gemensamt, medan de två efterföljande termerna har A gemensamt. Genom att bryta ut A' respektive A kan ekvationen ovan transformeras till

$$Sum = A'(B'C_{in} + BC_{in}') + A(B'C_{in}' + BC_{in}),$$

där B och C_{in} i den första termen bildar ett XOR-mönster:

$$B'C_{in} + BC_{in}' = B\hat{}C_{in},$$

samtidigt som B och C_{in} i den andra termen bildar ett XNOR-mönster:

$$B'C_{in}' + BC_{in} = (B\hat{}C_{in})'$$

Därmed gäller att

$$Sum = A'(B\hat{}C_{in}) + A(B\hat{}C_{in})'$$

I detta fall bildar A samt (B ^ C_{in}) bildar ett XOR-mönster. Därmed gäller att

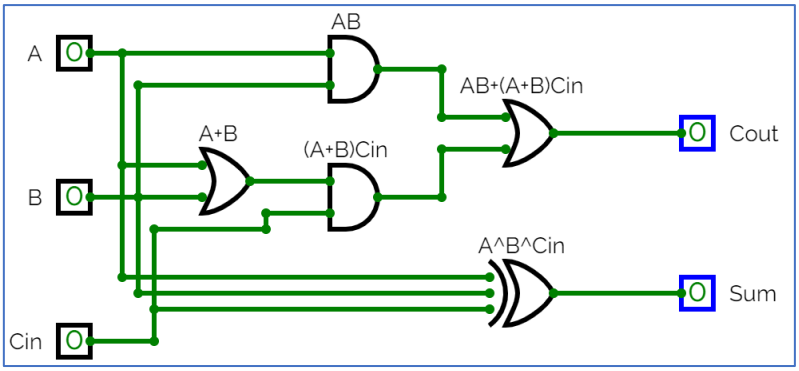
$$Sum = A\hat{}(B\hat{}C_{in})$$

Parentesen har i detta fall ingen påverkan, då vi endast har XOR-operatorer, som har samma prioritet, i ekvationen:

$$Sum = A\hat{}B\hat{}C_{in}$$

Sum kan alltså realiseras via en 3-ingångars XOR-grind med A, B samt C_{in} som insignaler. Utsignalen ur en 3-ingångars XOR-grind är hög vid udda antal höga insignaler, i detta fall om vi har en eller tre höga insignaler. Vi noterar i sanningstabellen att Sum blir hög just då vi har en eller tre höga insignaler.

Grindnätet för heladderaren kan därmed realiseras såsom visas nedan:

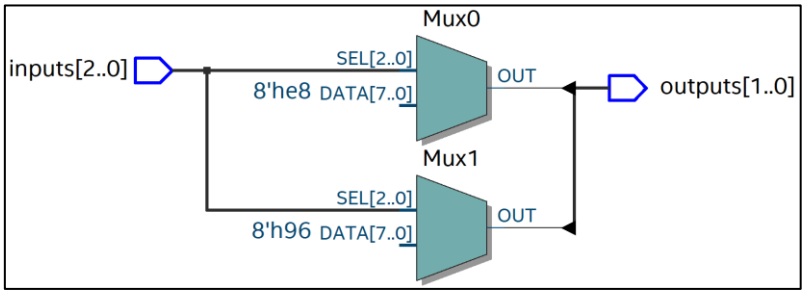


Figur 3: Grindnät för realisering av heladderaren för hand.

Heladderaren kan realiseras i VHDL via följande modul döpt *full_adder:*

```
--------------------------------------------------------------------------------
-- @brief Implementation of a full adder consisting of inputs {a, b, cin} and
--        outputs {cout, sum}. The truth table is shown below:
--
--        {a, b, cin} {cout, sum}
--            000          00
--            001          01
--            010          01
--            011          10
--            100          01
--            101          10
--            110          10
--            111          11
--
-- @param inputs[in]   Inputs and carry in.
-- @param outputs[out] Carry out and sum of inputs.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
    port(inputs : in std_logic_vector(2 downto 0);
         outputs: out std_logic_vector(1 downto 0));
end entity;

architecture behaviour of full_adder is
begin
    ------------------------------------------------------------------------------
    -- @brief Updates the output whenever the input changes.
    ------------------------------------------------------------------------------
    output_process: process (inputs) is
    begin
        case (inputs) is
            when "000"  => outputs <= "00";
            when "001"  => outputs <= "01";
            when "010"  => outputs <= "01";
            when "011"  => outputs <= "10";
            when "100"  => outputs <= "01";
            when "101"  => outputs <= "10";
            when "110"  => outputs <= "10";
            when "111"  => outputs <= "11";
            when others => outputs <= "00";
        end case;
    end process;

end architecture;
```

Efter kompilering av ovanstående VHDL-kod har följande krets syntetiserats. Notera att användning av en case-sats för realisering av heladderaren medförde av kompilatorn fick mer frihet att realisera grindnätet godtyckligt och valde då att använda multiplexers.

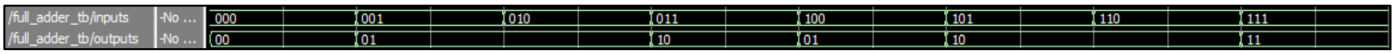


Figur 4: Syntetiserad krets för heladderaren.

Konstruktionen kan valideras via simulering i ModelSim. För att konfigurera simuleringen behöver vi en testbänk, såsom *full_adder_tb* nedan. I denna testbänk skapas en instans av modulen *full_adder* och vi ansluter signaler *inputs* samt *outputs* till portarna med samma namn. Varje kombination 000 – 111 av insignaler {A, B, C_{IN}}, här i form av vektorn *inputs,* testas under 10 ns var. Simuleringstiden uppgår därmed till 80 ns:

```
--------------------------------------------------------------------------------
-- @brief Test bench for the full_adder module. Each combination 000 - 111 of
--        inputs {a, b, cin} is tested during 10 ns, hence the simulation time
--        is 160 ns.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity full_adder_tb is
end entity;


architecture behaviour of full_adder_tb is

--------------------------------------------------------------------------------
-- @brief Signals used for simulating the full adder.
--------------------------------------------------------------------------------
signal inputs : std_logic_vector(2 downto 0) := "000";
signal outputs: std_logic_vector(1 downto 0) := "00";

begin

    --------------------------------------------------------------------------------
    -- @brief Creates a full adder and connects signals for simulation.
    --------------------------------------------------------------------------------
    sim_instance: entity work.full_adder
    port map(inputs, outputs);

    --------------------------------------------------------------------------------
    -- @brief Tests all combinations 000 - 111 of inputs during 10 ns each.
    --        After the simulation is finished, the process is halted, else the
    --        simulation would start over.
    --------------------------------------------------------------------------------
    sim_process: process is
    begin
        for i in 0 to 7 loop
            inputs <= std_logic_vector(to_unsigned(i, 3));
            wait for 10 ns;
        end loop;
        wait;
    end process;

end architecture;
```

Genom att jämföra resultatet i ModelSim med sanningstabellen för heladderaren ser vi att konstruktionen fungerar som tänkt:



Figur 5: Resultat efter simulering i ModelSim.