E2DSD

Øvelse 3 - ARITHMETIC AND LOGICAL OPERATORS IN VHDL

Gruppe 17

Robert Gudbjerg – 202010534

Nicolaj Kold Zefting- 201204718

Opgave 1: Signed and Unsigned Arithmetic 3
1.1 Introduktion 3
1.2 Design og Implementering 3
1.3 Resultater 6
1.4 Test på DE2-Board 8
1.5 Diskussion 11
1.6 Konklusion 11
Opgave 2: Concatenation 12
2.1 Introduktion 12
2.2 Design og Implementeringen 12
2.3 Resultater 14
2.4 Test på DE2-Board 14
2.5 Konklusion 15
3 Multiplication 16
3.1 Introduktion 16
3.2 Design og Implementeringen 16
3.3 Resultater 18
3.5 Diskussion 21
3.6 Konklusion 21
Opgave 4: Binary to 7-Segment Decoder Using “WITH-SELECT” 22
4.1 Introduktion 22
4.2 Design og Implementering 22
4.3 Resultater 25
4.4 Test på DE2-Board 25
4.6 Konklusion 26
Opgave 5: Multiplexing Using “WHEN” 27
5.1 Introduktion 27
5.2 Design og Implementeringen 27
5.4 Test på DE2-Board 29
5.6 Konklusion 31

Opgave 1: Signed and Unsigned Arithmetic

1.1 Introduktion

Denne øvelse undersøges implementeringen af aritmetiske og logiske operationer. Formålet er at lære de forskellige kodningsformer i VHDL, samt at arbejde med aritmetik i VHDL.

1.2 Design og Implementering

I øvelse 1 implementeres en 4-bit adder, som vist på Figur 1. Den kan regne med signed og unsigned operationer.

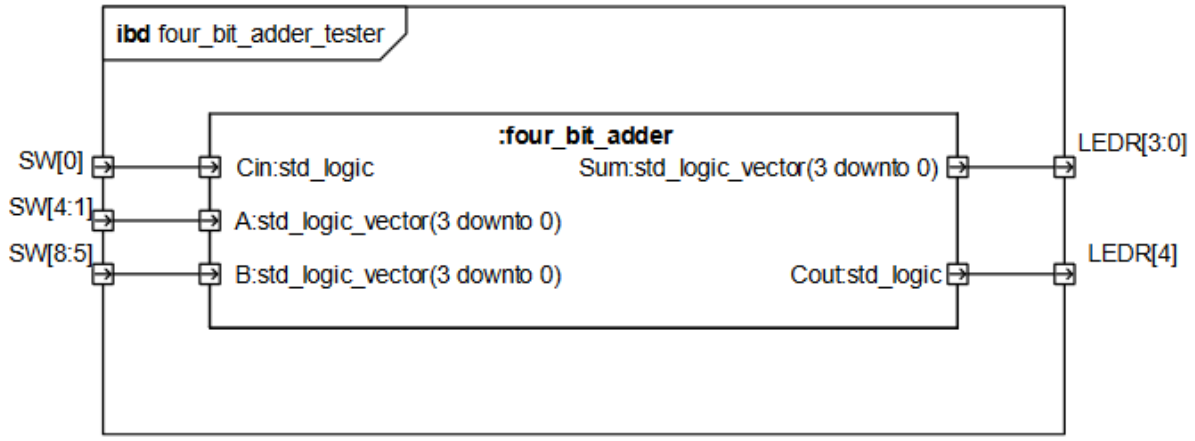


***Fig. 2:** 4-bit adder with carry.*

Figur 1 IDB for four bit adder. Fra opgavevejledningen

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.ALL;

ENTITY four_bit_adder_simple IS

   GENERIC -- set array size to 4
   (
      DATA_WIDTH : NATURAL := 4
   );

   PORT
   (
      a   : IN STD_LOGIC_VECTOR ((DATA_WIDTH - 1) DOWNTO 0);
      b   : IN STD_LOGIC_VECTOR ((DATA_WIDTH - 1) DOWNTO 0);
      Cin : IN STD_LOGIC;
      sum : OUT STD_LOGIC_VECTOR ((DATA_WIDTH - 1) DOWNTO 0);
      Cout: OUT STD_LOGIC
   );

END ENTITY;

ARCHITECTURE unsigned_impl OF four_bit_adder_simple IS
   signal sum_temp : std_LOGIC_VECTOR(4 downto 0);
   constant zeroVevtor : std_LOGIC_VECTOR (3 downto 0) := "0000";
BEGIN
-- save the resized unsigned a and b + the and product of cin and zeroVevtor
   sum_temp <= STD_LOGIC_VECTOR(resize(unsigned(a),5) + resize(unsigned(b),5) + unsigned(zeroVevtor & cin));
-- output bit 3 to 0 in sum_temp to sum
   sum <= sum_temp(3 downto 0);
-- output bit 4 in cout
   Cout <= sum_temp(4);
END unsigned_impl;
```

Snippet 1 Unsigned implementation af four bit adder

```vhdl
ARCHITECTURE signed_impl OF four_bit_adder_simple IS
    signal sum_temp : std_LOGIC_VECTOR(4 downto 0);
    constant zeroVevtor : std_LOGIC_VECTOR (3 downto 0) := "0000";
BEGIN
    sum_temp <= STD_LOGIC_VECTOR(resize(signed(a),5) + resize(signed(b),5) + signed(zeroVevtor & cin));
    sum <= sum_temp(3 downto 0);
    Cout <= sum_temp(4);
END signed_impl;
```

Snippet 2 Signed implementation of four bit adder. Se forklarende kommentarer ovenfor.

Koden for signed og unsigned er tæt på at være ens, men med forskellen at vi typecaster til enten signed eller unsigned.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY four_bit_adder_tester IS
   PORT
   (
      -- Input ports
      SW     : IN STD_LOGIC_VECTOR(8 DOWNTO 0);

      -- Output ports
      LEDR   : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)

   );
END four_bit_adder_tester;
ARCHITECTURE four_bit_adder_tester_impl OF four_bit_adder_tester IS
BEGIN

   uut : ENTITY work.four_bit_adder_simple(unsigned_impl) PORT MAP
      (
         -- set ports according to the IBD
         A => SW(4 downto 1),
         B => SW(8 downto 5),
         CIN => SW(0),
         Sum => LEDR(3 downto 0),
         COUT => LEDR(4)
      );

END four_bit_adder_tester_impl;
```

Snippet 3 Tester program. Her testes den unsigned implantation

Vi har lavet et testprogram som tester, at vores kode gør som beskrevet i opgaven. Vi har set ports i forhold til det i opgaven angivet IDB (se Figur 1).

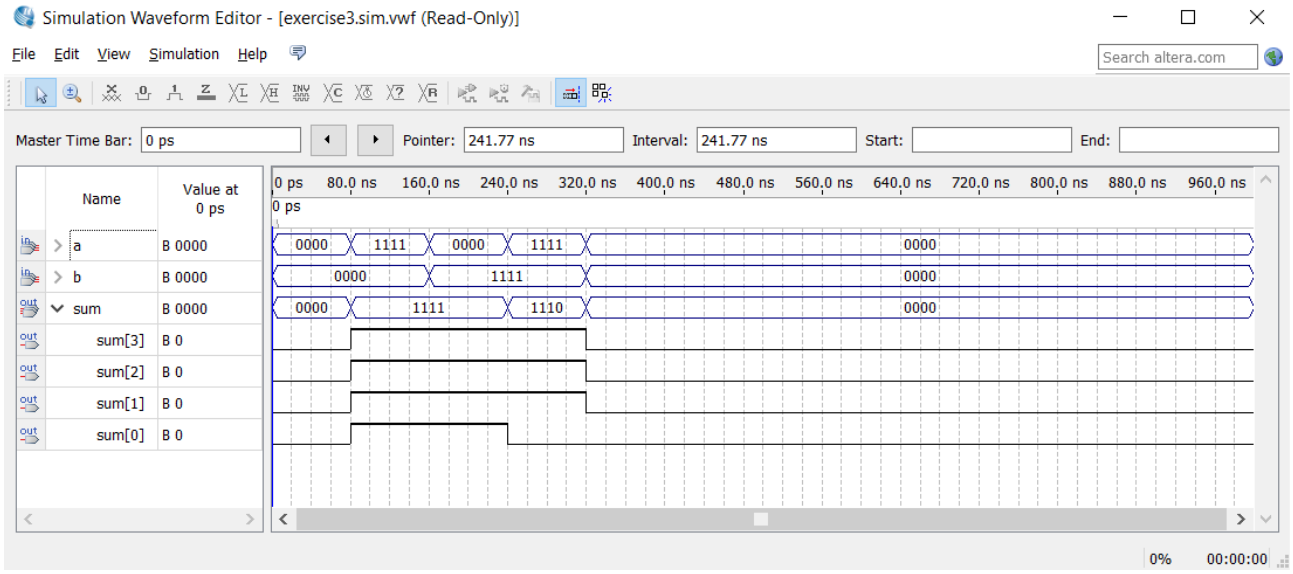
Vi testede også vores signed implementering

```
uut2 : ENTITY work.four_bit_adder_simple(signed_impl) PORT MAP
      (
         A => SW(4 downto 1),
         B => SW(8 downto 5),
         Sum => LEDR(3 downto 0)
      );

END four_bit_adder_tester_impl;
```

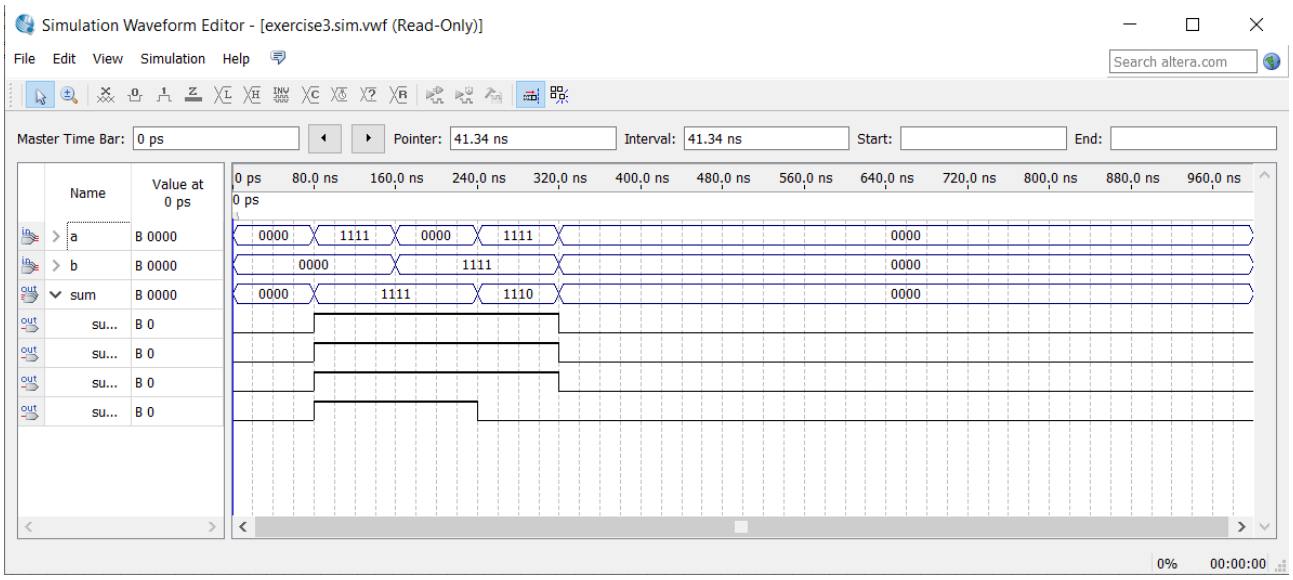
Snippet 4 Tester program. Her testes den signed implementation

1.3 Resultater

Vi testede vores implementation med funktionelle simuleringer.



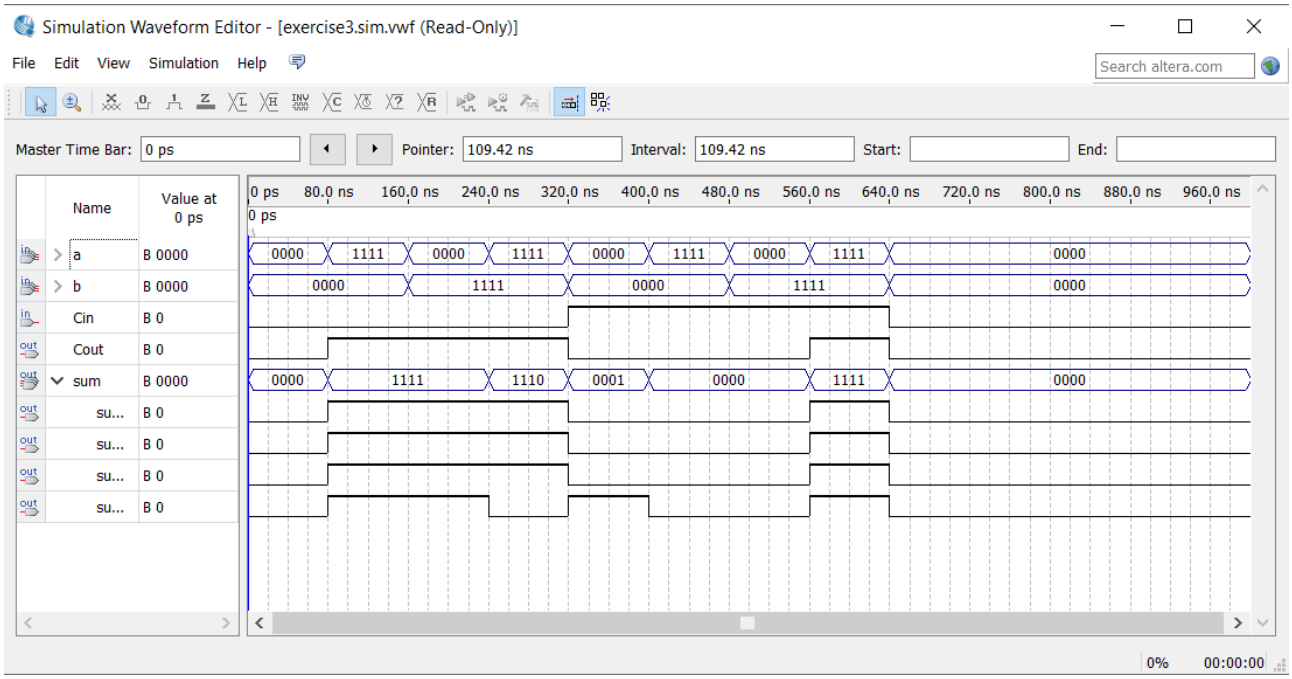
Figur 2 unsigned



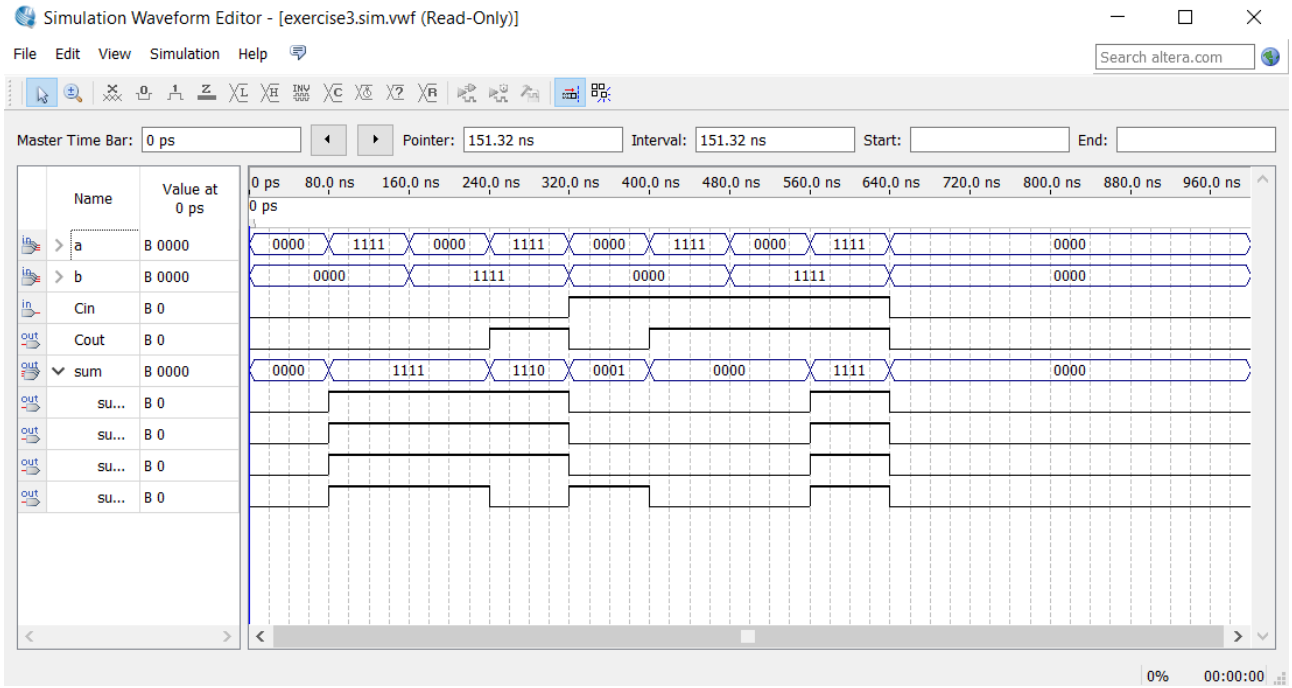
Figur 3 signed

Baseret på vores test, kan vi udlede at der rent simuleringsmæssigt ikke er nogen forskel mellem de to implementationer.

Herefter implementerede vi en carry og simulerede henholdsvis unsigned og signed.



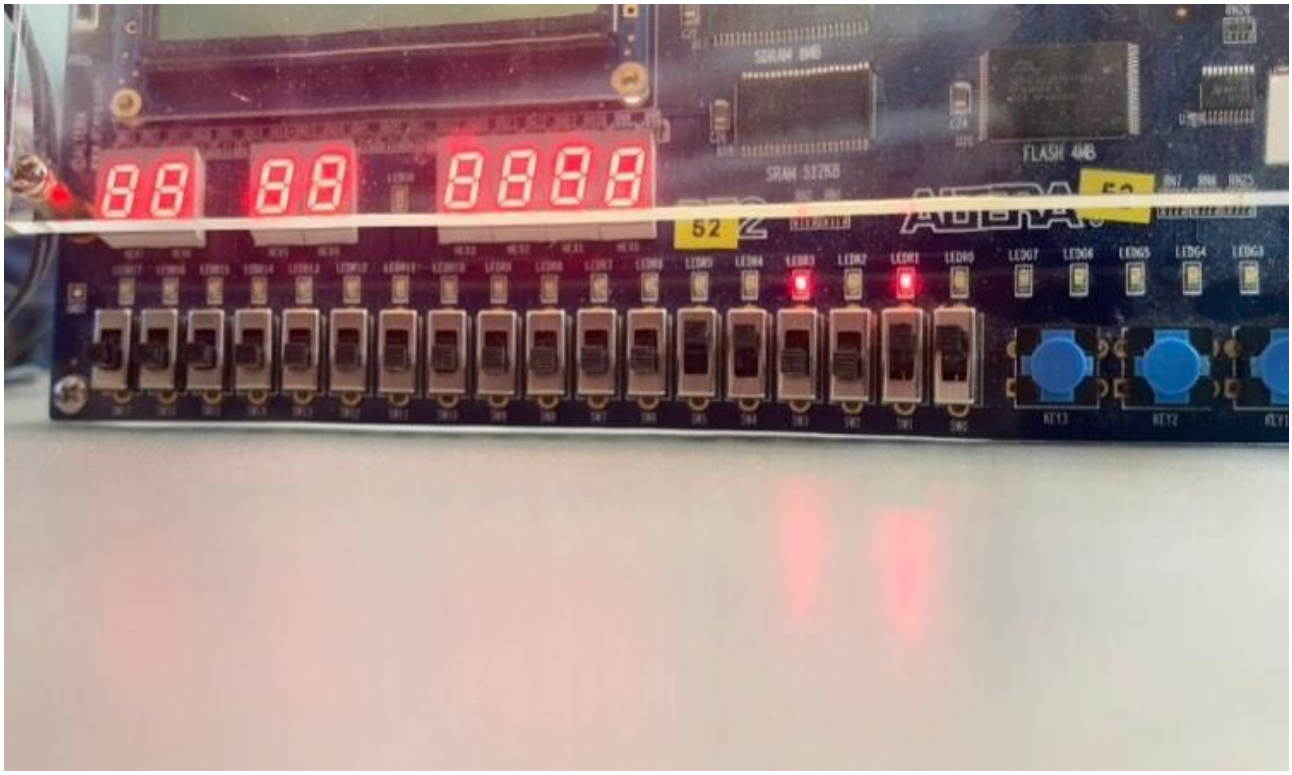
Figur 4 unsigned



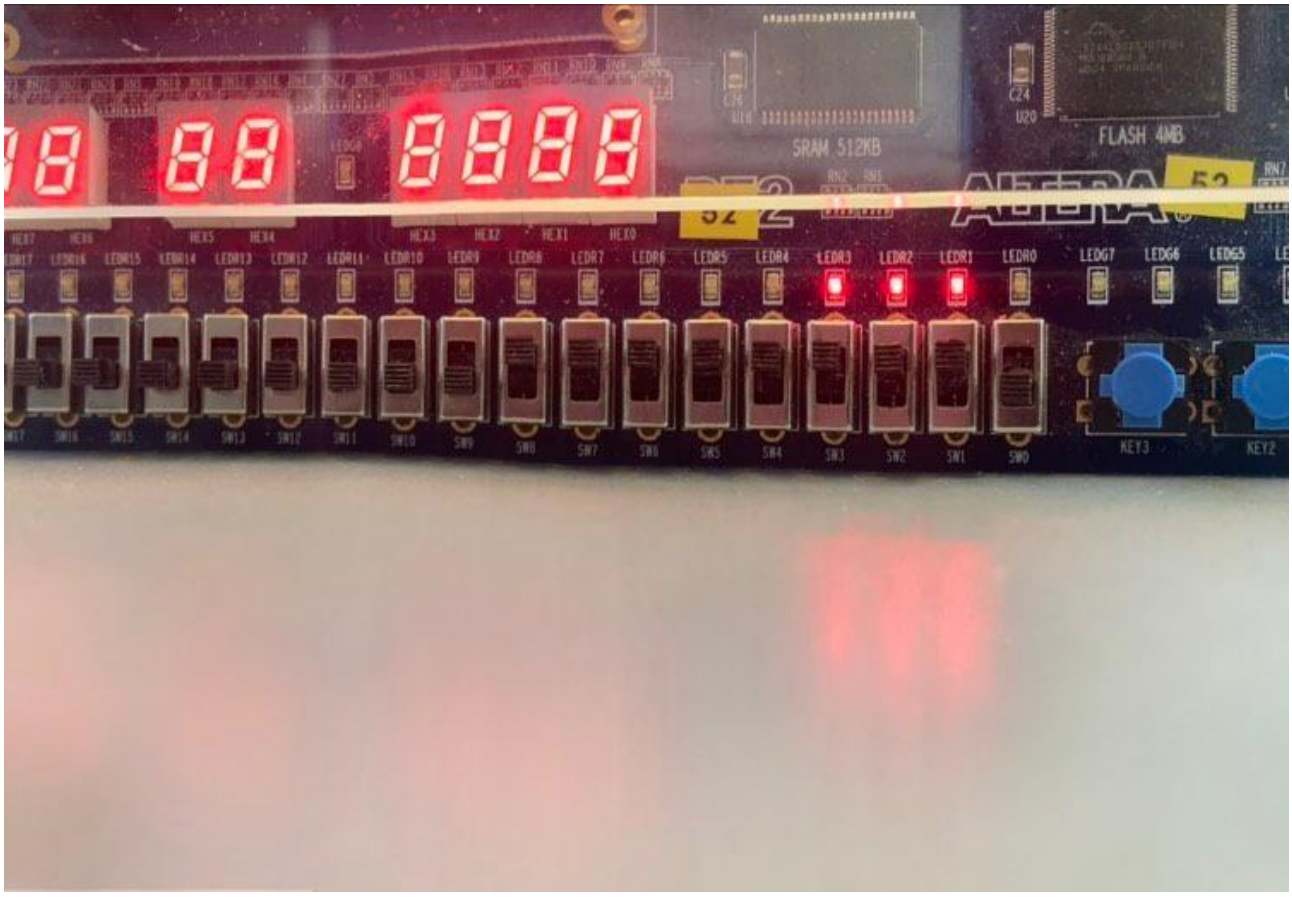
Figur 5 signed

Her observerer vi en forskel ved Cout, hvor outputtet er spejlvendt mellem signed og unsigned. Dette kan skyldes overflow.

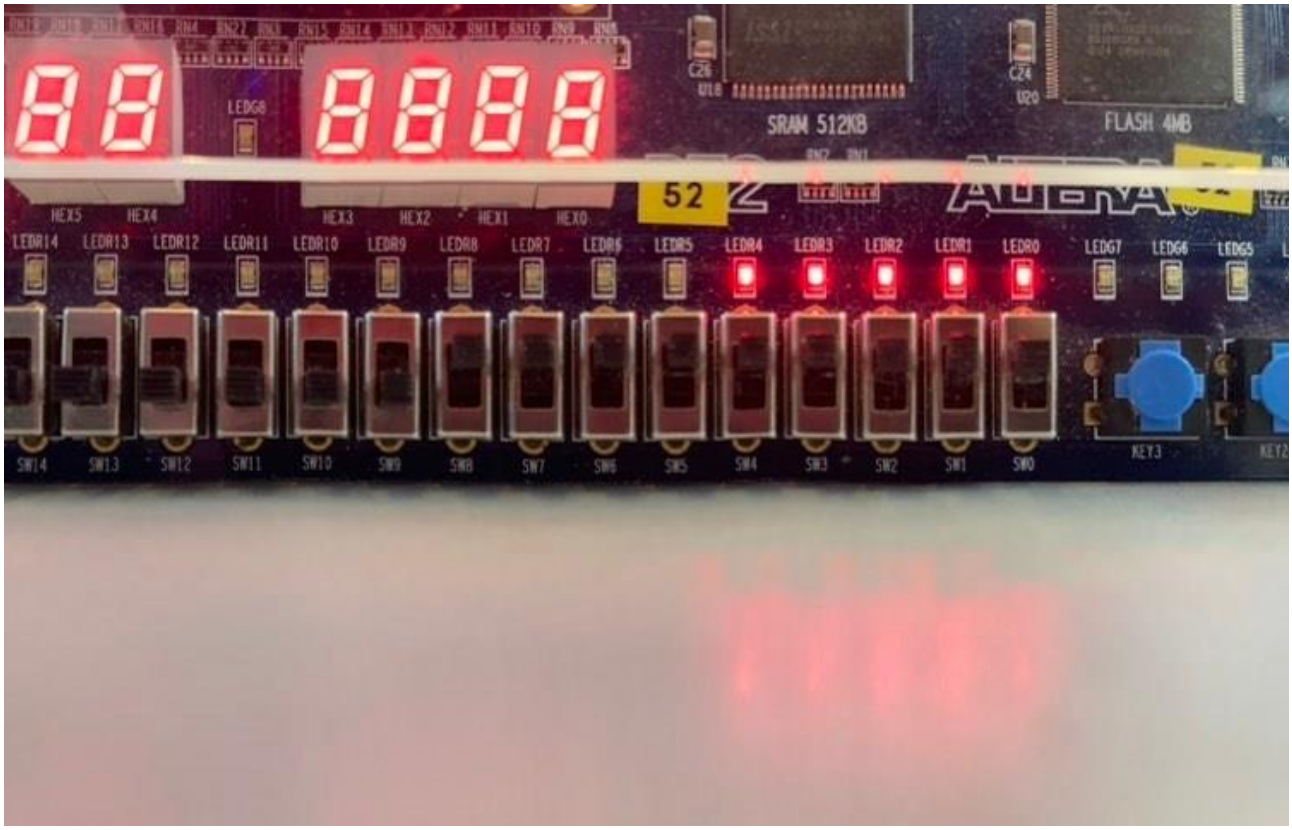
1.4 Test på DE2-Board



Figur 6 A01-b01 unsigned

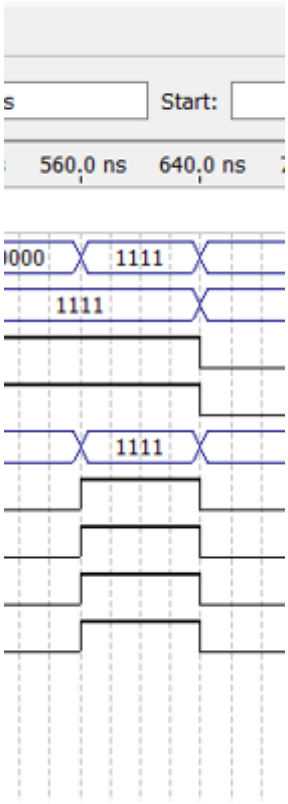


Figur 7 A1111B1111 Sw1-8



Figur 8 SW8 Carry

Sammenligner vi dette resultat med vores funktionelle analyse observerer vi, at Figur 8 har det forventede in og output i forhold til simuleringen (Figur 9).



Figur 9 uddrag af figur 5

1.5 Diskussion

Vi koder VHDL-kode til at tage to grupper af 4 bit og lægger sammen. Vi forsøger at typecaste værdierne til signed og derefter unsigned. Vi undersøger herefter, hvordan den signed version og den unsigned version opfører sig.

1.6 Konklusion

Vi har fået erfaring med brug af signed og unsigned logiktyperne og deres forskelle. Baseret på vores forsøg kan vi konkludere, at Cout (outputtet) er spejlvendt, når vi simulerer koden. Ved at have two implementeringer kan vi sikre os at compileren anvender den implementation vi ønsker til den givne opgave. Hvis vi ikke specificerede dette ville compileren implicit vælge, og derved kan uforudsete resultater opstå.

Opgave 2: Concatenation

2.1 Introduktion

I øvelse 2 skal vi udføre shift operationer på bits. Det inkluderer en left shift, hvor vi skal flytte en bit serie til venstre, så der kommer '0' på de pladser som skubbes ind fra højre. En right shift, hvor vi skal flytte en bit serie til højre, så der kommer '0' på de pladser som skubbes ind fra venstre. Og endelig en rotate right som flytter en bit serie til højre og flytter de bits som går udover i højre side til den venstre side. Formålet med denne øvelse er at få en forståelse for concatination af bits serier via '&' operatoren.

2.2 Design og Implementeringen

```
entity shift_div is
port (
a : in std_logic_vector(7 downto 0);
a_shl, a_shr, a_ror: out std_logic_vector(7 downto 0)
);
end;

architecture shift_div_impl OF shift_div is
BEGIN
a_shl <= a(6 DOWNTO 0) & '0';
a_shr <= "00" & a(7 DOWNTO 2);
a_ror <= a(2 downto 0) & a(7 downto 3);

END shift_div_impl;
```

Tabel 1 implimenting af bit shift

Vi har implementeret de tre bit shifts ved at contatenere på følgende måde:

SHL: 7 første bit concatineres med 0. Det indsætter et nul på plads 0. fx 11111110

SHR: Der indsættes nuller på plads 7 og 6 ved at 00 & med de 5 første bit i a.

ROR: vi flytter de tre nederste bits til de tre øverste pladser.

Vi har endvidere implementeret en tester:

```
ENTITY shift_div_tester IS
    PORT
    (
        -- Input ports
        SW      : IN STD_LOGIC_VECTOR(7 DOWNTO 0);

        -- Output ports
        LEDR    : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
        LEDG    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)

    );
END shift_div_tester;
ARCHITECTURE shift_div_tester_impl OF shift_div_tester IS
BEGIN

    uut : ENTITY work.shift_div PORT MAP
        (
            A => SW(7 downto 0),
            a_shl => LEDG(7 downto 0),
            a_shr => LEDR(7 downto 0),
            a_ror => LEDR(17 downto 10)

        );

END shift_div_tester_impl;
```

Figur 10 implimentation af tester

Under testen på DE2 boardet skal både de grønne og røde LED'er anvendes. Vi sætter `a_shl` til at benytte `LEDG` og vi nyttter alle 18 røde led'er. `a_ror` benytter led'er 17 til 10, men `a_shr` benytter 7 til 0.

2.3 Resultater

Der er ikke brugt nogle logiske elementer, da alle bitoperationer kan klares med ledninger.

Flow Summary	
Flow Status	Successful - Thu Sep 29 09:18:49 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	exercise3
Top-level Entity Name	shift_div
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	32 / 475 (7 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Tabel 2 Summary - Concatenation

2.4 Test på DE2-Board

Figur 11 Concat sw 4 og 5 slukket

Figur 12 Concat sw 7-0

2.5 Konklusion

Vi har implementeret kode, som kan foretage bitshiftoperationerne ” Shift a one-time to the left”, ” Shift a two times right”, “Rotate a three times right”.

Vi ser, at der ved bitshifts ikke benyttes nogle logiske operationer, da dette kan klares med ledninger.

Vi har også implementeret kode, som baseret på input kan tænde de røde og grønne LED’er.

3 Multiplication

3.1 Introduktion

I denne øvelse skrives VHDL kode til multiplikation af to værdier samt tester. Herefter ændres koden først til at håndtere input i forskellige bit størrelser. Derefter ganges med en konstant. Formålet med denne øvelse er at lære at arbejde med aritmetiske operationer.

3.2 Design og Implementeringen

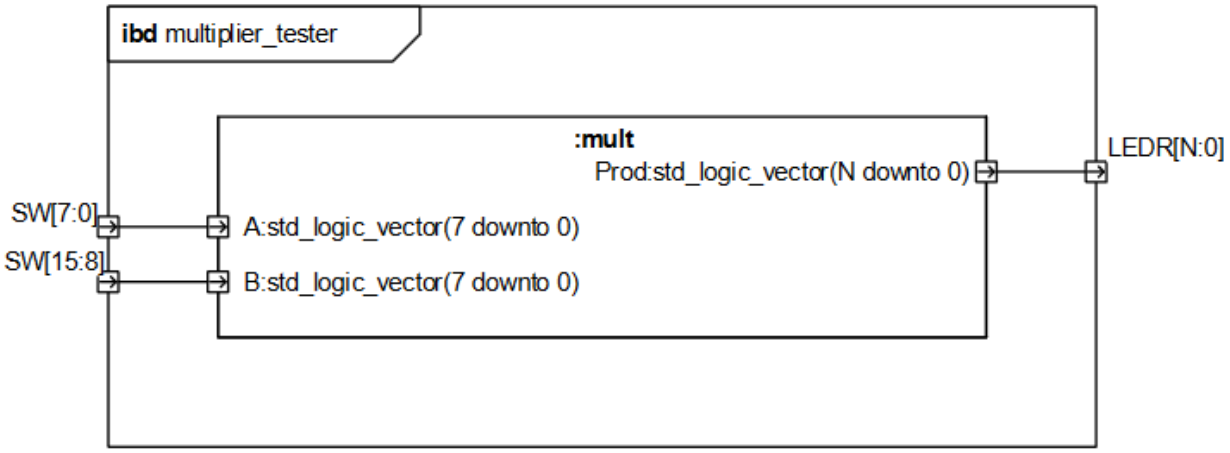


***Fig. 4:** Multiplier.*

Figur 13 IDB for multiplier_tester. fra opgavevejledningen

Vi har implementeret kode, som kan gange to inputs med hinanden.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity multiplier is
port (
    a : in std_logic_vector(0 downto 0);
    b : in std_logic_vector(0 downto 0);
    Prod: out std_logic_vector(1 downto 0)
);
end;


architecture multiplier_impl OF multiplier is
BEGIN
    Prod <= a * b;

END multiplier_impl;
```

Figur 14 Implementation af multiplier med multiplication af to inputs. Her multiplikation af to 1 bit. Størrelsen varieres under forsøget.

Vi har desuden implementeret kode, som kan foretage multiplikation mellem et input og en konstant. Vi fik under opgaveregningen at vide af Jesper, at inputtet skulle castes til unsigned. Vi har her kun inkluderet vores architecture. Den resterende kode er identisk med Figur 14.

Under udførelsen af opgave 3C varierede vi SW og LEDR til størrelserne (x downto 0)

SW: 31,15,8,3,2,1,0.

LEDR: 63, 29, 15, 5, 3, 2, 1.

```
architecture multiplier_impl OF multiplier is
BEGIN

   Prod <= std_logic_vector(unsigned(a) * 10);

END multiplier_impl;
```

Figur 15 Implementation af input og en konstant.

Vi har implimenteret en tester (multiplikation af 8 bits).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.ALL;

ENTITY multiplier_tester IS
    PORT
    (
        -- Input ports
        SW      : IN STD_LOGIC_VECTOR(15 DOWNTO 0);

        -- Output ports
        LEDR    : OUT STD_LOGIC_VECTOR(17 DOWNTO 0)

    );
END multiplier_tester;

ARCHITECTURE multiplier_tester_impl OF multiplier_tester IS
BEGIN

    uut : ENTITY work.multiplier
    PORT MAP
        (
            A => SW(7 downto 0),
            B => SW(15 downto 8),
            Prod => LEDR(15 downto 0)

        );


END multiplier_tester_impl;
```

Resultaterne af denne øvelse vises i Tabel 3.

3.3 Resultater

Vi har testet implementationen på DE2 boardet.

Figur 16 Testudførelse

Flow Summary	
Flow Status	Successful - Thu Sep 29 11:01:26 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	exercise3
Top-level Entity Name	multiplier_tester
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	101 / 33,216 (< 1 %)
Total combinational functions	101 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	34 / 475 (7 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

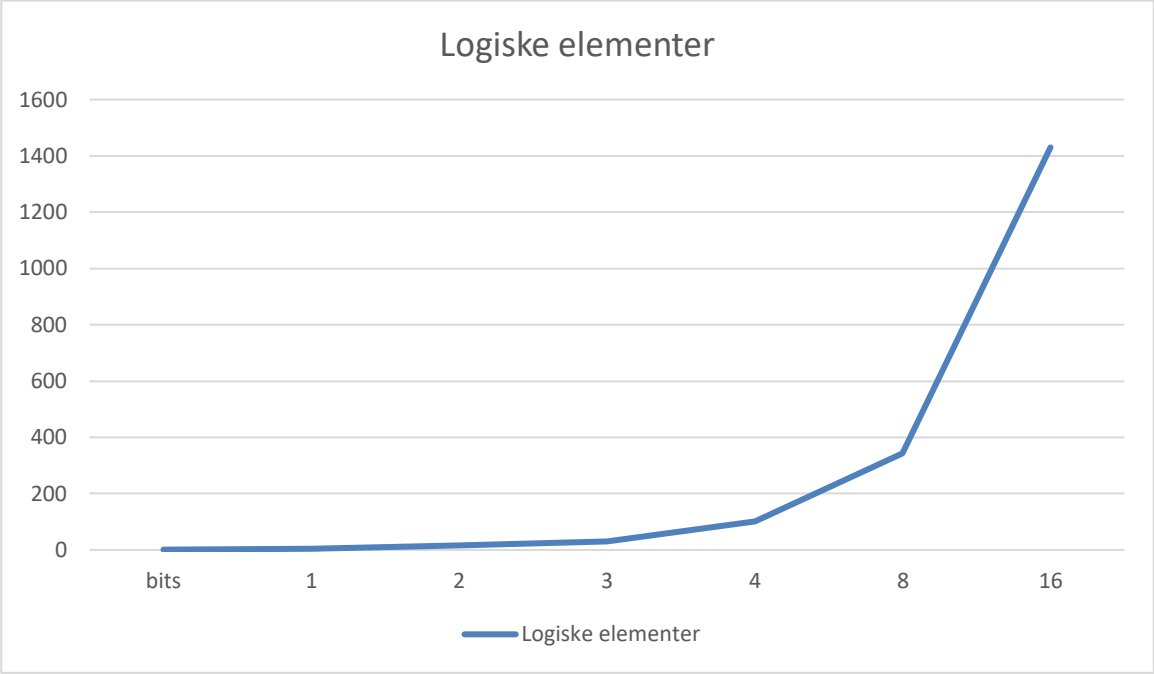
Figur 17 Compilation report for vores implimentation af multiplier

Her observeres det at der anvendes 101 logiske elementer.

Antal bits	Logiske elementer
1	1
2	4
3	17
4	31
8	101
16	343
32	1430

Tabel 4 Oversigt over sammenhængen mellem bits og antal logiske elementer

Vi har plottet dette og observerer, at der ikke er tale om en eksponentiel vækst.



Tabel 5 Plot af sammenhæng mellem bits og logiske elementer

Endvidere har vi analyseret sammenhængen ved multiplikation af konstanter.

Konstant	Antal logiske elementer
2	0
3	9
4	0
7	18
8	0
10	9

3.5 Diskussion

Vi ganger input 'A' med konstanterne 2, 3, 4, 7, 8 og 10. Vi kan se på ressourceforbruget at når vi ganger de lige konstanter bruger de ingen logiske elementer før vi når op på konstanten 10 der bruger 9 logiske elementer. De ulige konstanter bruger til gengæld elementer et stigende antal elementer

3.6 Konklusion

Baseret på vores forsøg kan vi konkludere, at antallet af logiske elementer stiger eksponentielt jo flere bits der skal ganges sammen. Vi kan desuden konkludere, at det ikke kræver nogle logiske elementer at gange med et lige ental. Derimod kræver det dobbelt så mange logiske elementer at gange med 7, som at gange med 3. Endvidere viser vores resultater, at det kræve logiske elementer at gange med konstanten 10.

E2DSD

Øvelse 4 - DATAFLOW-STYLE COMBINATORIAL DESIGNS IN VHDL

Gruppe 17

Robert Gudbjerg – 202010534

Nicolaj Kold Zefting- 201204718

Opgave 4: Binary to 7-Segment Decoder Using "WITH-SELECT"

4.1 Introduktion

I denne øvelse vil vi producere VHDL kode til at udskrive et binært tal til en 7-segment display og teste det på De-II board. Vi gør det for at lære at arbejde med "with-select" VHDL-statements.

4.2 Design og Implementering

Vi har benyttet en "with" statement til at løse den stillede opgave. Baseret på hvilke SW-kontakter, der er aktive, skal displayet den tilsvarende værdi som hex. bitværdien til venstre for "when" er de LED'er i displayet som skal tænde for at vise input-værdien.

```
entity bin2sevenseg is
port (
    bin : in std_logic_vector(3 downto 0);
    Sseg: out std_logic_vector(6 downto 0)
);
end;

architecture bin2sevenseg_impl OF bin2sevenseg is
BEGIN

with bin select Sseg <=
          "1000000" when "0000",--0
          "1111001" when "0001",--1
          "0100100" when "0010",--2
          "0110000" when "0011",--3
          "0011001" when "0100",--4
          "0010010" when "0101",--5
          "0000010" when "0110",--6
          "1111000" when "0111",--7
          "0000000" when "1000",--8
          "0011000" when "1001",--9
          "0001000" when "1010",--A
          "0000011" when "1011",--b
          "1000110" when "1100",--C
          "0100001" when "1101",--d
          "0000110" when "1110",--E
          "0001110" when "1111",--F
          "0111110" when others;--U


END bin2sevenseg_impl;
```

Figur 18 Implementation af et display. Kan vise 0-F.

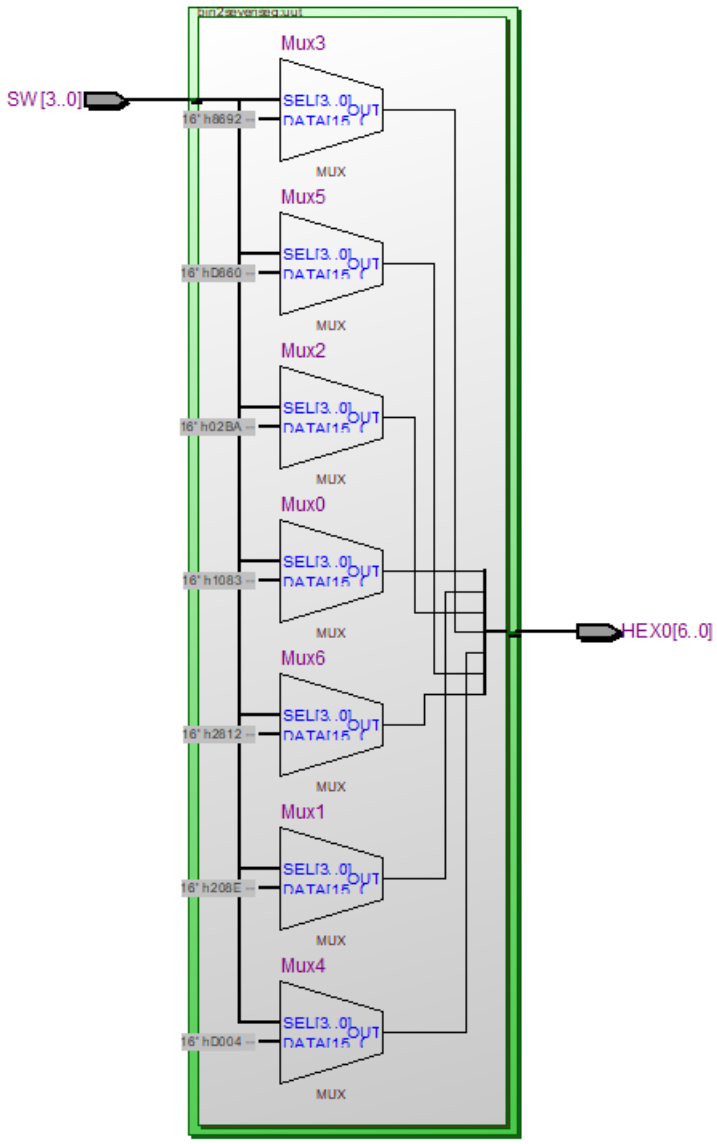
Vi har implementeret en tester.

```
ENTITY bin2sevenseg_tester IS
    PORT
    (
        -- Input ports
        SW      : IN STD_LOGIC_VECTOR(3     DOWNTO 0);

        -- Output ports
        HEX0    : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)

    );
END bin2sevenseg_tester;

ARCHITECTURE bin2sevenseg_tester_impl OF bin2sevenseg_tester IS
BEGIN

    uut : ENTITY work.bin2sevenseg
    PORT MAP
        (
            bin => SW(3 downto 0),
            Sseg => HEX0(6 downto 0)

        );


END bin2sevenseg_tester_impl;
```

Figur 19 Implementation af tester.

4.3 Resultater

Som en del af opgaven skulle vi lave et RTL-view af vores kode.



Figur 20 RTL-view

RTL-viewet viser, at der er blevet genereret et antal ”mux” som svare til antallet af inputs.

4.4 Test på DE2-Board

I dette afsnit vil vi dokumentere vores hardware test.

Vi testede ved at få systemet til at udskrive bitsekvensen ”1100”, hvilket svare til hex værdien ”C”

Figur 21 Implementation af tester - Viser hex C

4.6 Konklusion

Vi har implementeret et system som anvender with statements til at udskrive hex værdierne fra 0-F. Vi har endvidere implementeret en ”others” case, som udskriver et U. Others håndtere alle andre input kombinationer end de specificerede. Vi har fået erfaring med “with-select" statements i VHDL.

Opgave 5: Multiplexing Using "WHEN"

5.1 Introduktion

Formålet med denne opgave er at arbejde med "when" statements. Desuden er en del af formålet også at lære at identificere latches og håndtere dem.

5.2 Design og Implementeringen

Vi opretter arkitektur med et signal til at opbevare værdien af pins mens vi arbejder med dem. Eftersom signalet er en ledning som holder værdien uden at gemmer den, kan designet siges at være Combinatorial.

```
architecture hex_mux_impl OF hex_mux is
signal Number: std_logic_vector (20 downto 0);
BEGIN
```

Figur 22 Signal Number. Bruges til at gemme SW-værdier

Her slicer vi vores input og udskriver på vores 7-segments display. Det vil sige at bin som oprindeligt er 11 bits bliver til tre grupper af 4 bits. Sseg forventer 7 bits, så vores 21 bit Number deler vi ned til tre grupper af 7 bits.

```
h1: ENTITY work.bin2sevenseg
PORT MAP
(
   bin => bin(3 downto 0),
   Sseg => Number(6 downto 0)
);
h2: ENTITY work.bin2sevenseg
PORT MAP
(
   bin => bin(7 downto 4),
   Sseg => Number(13 downto 7)
);
h3: ENTITY work.bin2sevenseg
PORT MAP
(
   bin => bin(11 downto 8),
   Sseg => Number(20 downto 14)
);
```

Figur 23 port map til bin2sevenseg fra opg. 1.

Udskrift af “On” eller “Err”. Den udskriver “On” når ingen knapper er trykket og "Off” når KEY1 trykkes på.

```
tsseg <=
      "0000110010111101011111" when sel = "01" else-- E = 0000110  -- r = 0101111
      --"1111001" when "0001",
      "10000000101011111111" when sel = "11" else--On
      number when sel = "10" else -- switch output
      "01010010100101001010" ; -- latch avoidance
```

Figur 24 Kode for at tænde de relevante LED'er på boardet. Desuden håndtering af latches.

Vores input og output fra vores tester. SW er vores switches som vi bruger til at vælge værdi, KEY er trykknapper og HEX er vores 7-segmentdisplays.

```
ENTITY hex_mux_tester IS
   PORT
   (
      -- Input ports
      SW      : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
      KEY     : in STD_LOGIC_VECTOR(1 DOWNTO 0);

      -- Output ports
      HEX0    : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX1    : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX2    : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)

   );
END hex_mux_tester;
```

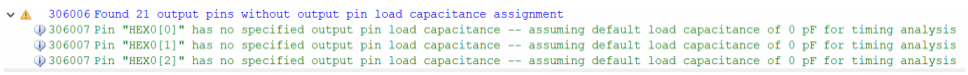
Figur 25 Implementation af tester

Arkitektur af vores tester. Her sender vi data igennem hex_mux og dirigere dataen til det korrekte output.

```
ARCHITECTURE hex_mux_tester_impl OF hex_mux_tester IS
BEGIN

   uut : ENTITY work.hex_mux
   PORT MAP
      (
         bin => SW(11 downto 0),
         sel => KEY(1 downto 0),

         tsseg(6 downto 0) => HEX0,
         tsseg(13 downto 7) => HEX1,
         tsseg(20 downto 14) => HEX2
      );

END hex_mux_tester_impl;
```

Figur 26 implementation af tester

Da vi kompilerede koden, fik vi en warrning om inferred latches (se Figur 27).



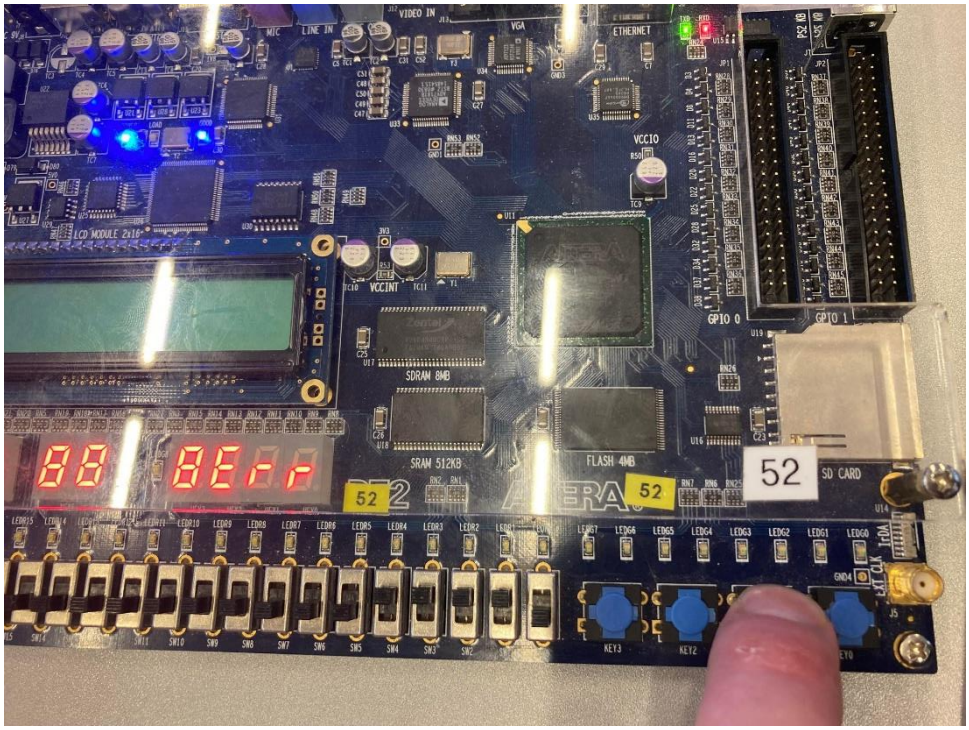
Figur 27 warning om inferred latches

Vi løste problemet med inferred latches ved at inkludere en else statement, hvor vi angav en bitsekvens. Den sidste else statement anvendes, hvor de tre første cases ikke dækker. Derved har vi fjernet de inferred latches.

5.4 Test på DE2-Board

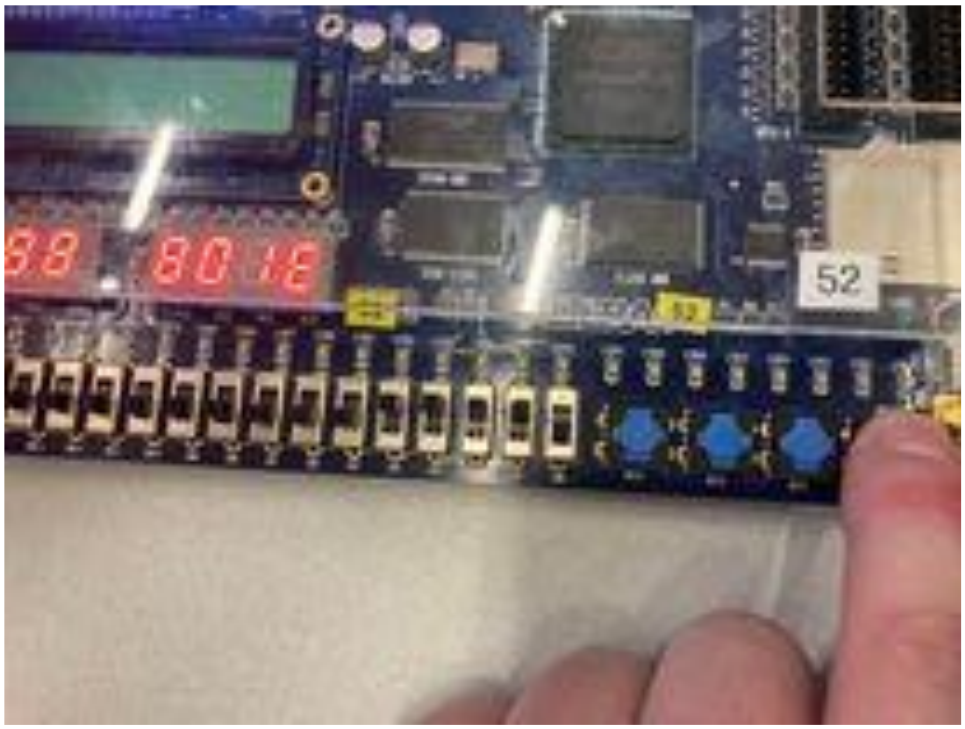
I dette afsnit vil vi dokumentere vores hardvare test.

Når ”KEY1” nedtrykkes, viser displayet ”Err”.



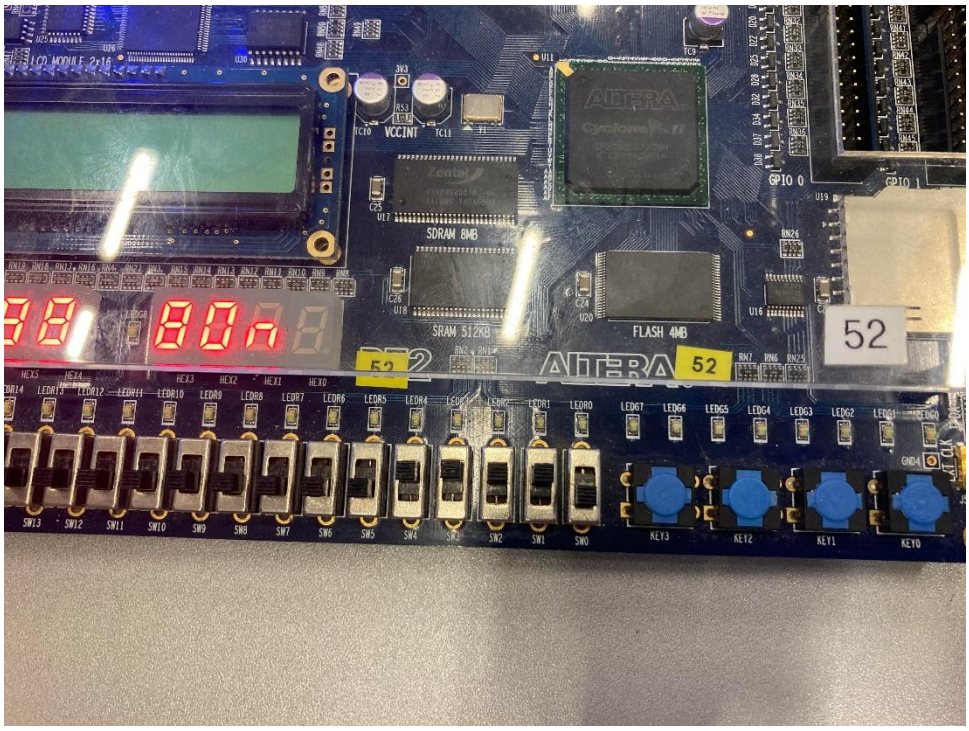
Figur 28 Viser Err

Når ”KEY0” nedtrykkes udskrives værdien fra SW-kontakterne.



Figur 29 viser bitværdien baseret på SW-kontakterne

Når ingen af KEY-knapperne nedtrykkes, viser displayet ”On”.



Figur 30 viser On, når ingen knapper nedtrykkes

5.6 Konklusion

Vi har anvendt et when statement til at lave et system som kan anvende tre knapper, en række switches og tre displays. Vi har håndteret latches ved at introducere en fjerde case, som håndterer de cases, som ikke er eksplicit defineret. Herved har vi kontrol over hvad der sker i de øvrige cases. Vi har lært at arbejde med "when” VHDL statemens og har fået mere erfaring med bit ordering via to og downto vektorer. Vi har også lært at identificere latches og hvordan vi håndtere dem.