E2DSD

Øvelse 2 - ARITHMETIC AND LOGICAL OPERATORS IN VHDL,

DATAFLOW-STYLE COMBINATORIAL DESIGNS IN VHDL

Gruppe 36

Hans Kobberø - 201803500

Rasmus M. Bertelsen - 201803610

Indholdsfortegnelse

Signed and Unsigned Arithmetic

Introduktion

I denne øvelse vil vi implementere signed og unsigned i en simpel 4-bit adder. Formålet er at se hvad forskellen er mellem hvordan den outputter i vores signed og unsiged adder. Derudover udvider vi med carry in og out, hvor vi bland andet bruger *resize* og *concatenation* til at adde og uddele resultatet til de rigtige LEDér.

Design og Implementering

```
1   Library ieee; -- library context clause
2   Use ieee.std_logic_1164.all; -- use clause
3   use ieee.numeric_std.all;
4   use work.all;
5
6   entity four_bit_adder is
7   port (
8   A : in std_logic_vector(3 downto 0);
9   B : in std_logic_vector(3 downto 0);
10  sum : out std_logic_vector(3 downto 0)
11  );
12  end four_bit_adder;
13
14  architecture unsigned_impl of four_bit_adder is
15  signal A_unsigned : unsigned(3 downto 0);
16  signal B_unsigned : unsigned(3 downto 0);
17  signal sum_unsigned : unsigned(3 downto 0);
18  begin
19
20  A_unsigned <= unsigned(A);
21  B_unsigned <= unsigned(B);
22
23  sum_unsigned <= A_unsigned + B_unsigned;
24
25  sum <= std_logic_vector(sum_unsigned);
26
27  end unsigned_impl;
28
29  -------------------------------------------------
30
31  architecture signed_impl of four_bit_adder is
32  signal A_signed : signed(3 downto 0);
33  signal B_signed : signed(3 downto 0);
34  signal sum_signed : signed(3 downto 0);
35  begin
36
37  A_signed <= signed(A);
38  B_signed <= signed(B);
39
40  sum_signed <= A_signed + B_signed;
41
42  sum <= std_logic_vector(sum_signed);
43
44  end signed_impl;
45
46
```

Figure 1 4-bit adder signed/unsigned

```
1   Library ieee; -- library context clause
2   Use ieee.std_logic_1164.all; -- use clause
3   use ieee.numeric_std.all;
4   use work.all;
5
6   entity four_bit_adder_with_carry is
7   port (
8       A : in std_logic_vector(3 downto 0);
9       B : in std_logic_vector(3 downto 0);
10      Cin : in std_logic;
11      sum : out std_logic_vector(3 downto 0);
12      Cout : out std_logic
13  );
14  end four_bit_adder_with_carry;
15
16  architecture unsigned_impl of four_bit_adder_with_carry is
17      signal A_unsigned : unsigned(3 downto 0);
18      signal B_unsigned : unsigned(3 downto 0);
19      signal carry_vector : unsigned(4 downto 0);
20      signal sum_unsigned : unsigned(4 downto 0);
21  begin
22
23      A_unsigned <= unsigned(A);
24      B_unsigned <= unsigned(B);
25      carry_vector <= ("0000" & Cin);
26
27      sum_unsigned <= resize(A_unsigned,5) + resize(B_unsigned,5) + carry_vector;
28
29      sum <= std_logic_vector(sum_unsigned(3 downto 0));
30      Cout <= std_logic(sum_unsigned(4));
31
32  end unsigned_impl;
33
```

Figure 2 4-bit unsigned adder with carry

```
1   Library ieee; -- library context clause
2   Use ieee.std_logic_1164.all; -- use clause
3   use ieee.numeric_std.all;
4   use work.all;
5
6   entity testbench is
7   port(
8       SW : in std_logic_vector(8 downto 0);
9       LEDR : out std_logic_vector(4 downto 0)
10  );
11  end testbench;
12
13  architecture test of testbench is
14  begin
15      uut: entity four_bit_adder_with_carry(unsigned_impl)
16      port map
17      (
18          A(3 downto 0) => SW(3 downto 0),
19          B(3 downto 0) => SW(7 downto 4),
20          Cin => SW(8),
21          sum(3 downto 0) => LEDR(3 downto 0),
22          Cout => LEDR(4)
23      );
24  end test;
25
```

Figure 3 Testbench 4-bit unsigned adder with carry

Resultater

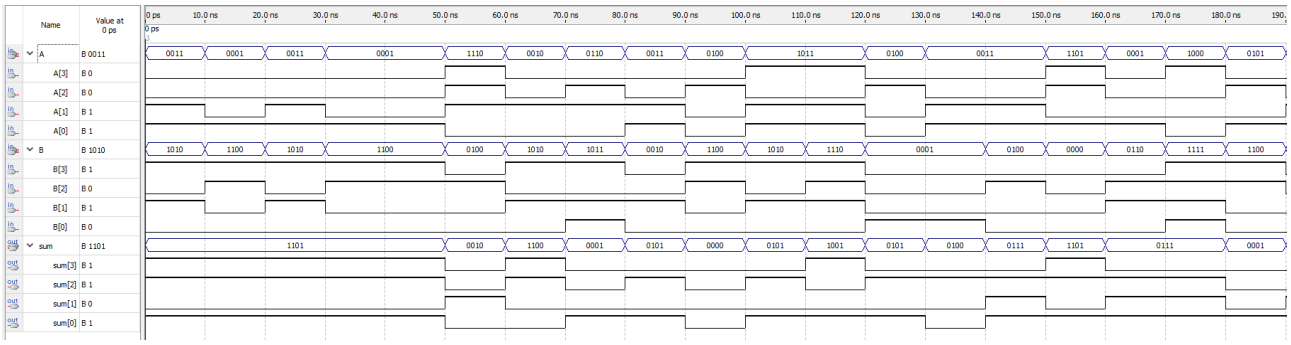


Figure 4 Functional analysis of Four bit adder

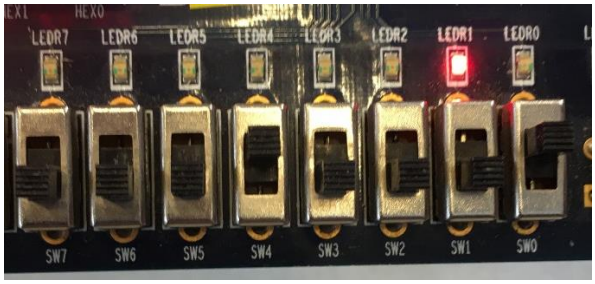


Figure 5 Four bit unsigned adder test: 1+1=2

Figure 6 Four bit unsigned adder test: 2+4=6

Figure 7 Four bit signed adder test: -7+7=0

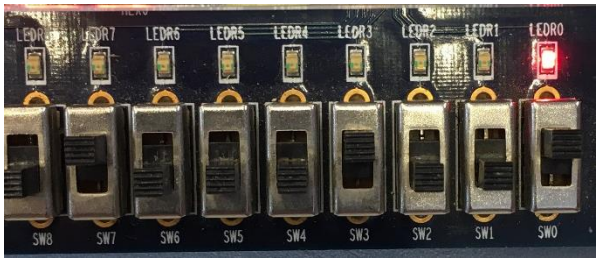


Figure 8 Four bit signed adder test: -7+8=1

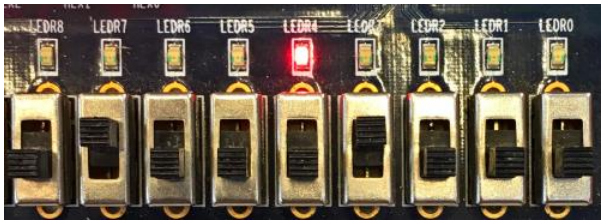


Figure 9 Four bit unsigned adder with carry test: 8+8 = Carry (16)

Diskussion

Når vi i øvelsen har implementeret four bit adderen som både signed og unsigned, har vi set at det reelt set ikke har gjort nogen forskel på vores output da vi testede det på DE2-boardet.

Konklusion

I forsøgende med 4 bit adderen hvor vi implementerede den både signed og unsigned så vi reelt ingen forskel på de tests vi lavede. Dette er fordi hardwaren lægger tallene sammen binært på samme måde uanset om de er signed eller unsigned. Når vi i koden fortæller den at den er signed eller unsigned er det egentlig mere til os selv så vi ved hvordan vi skal fortolke de resultater vi får ud. Der er funktioner i VHDL hvor det gør en forskel om det er signed eller unsigned, men når vi lægger to tal sammen binært, sker det på nøjagtig samme måde uanset om det er signed eller unsigned.

Concatenation

Introduktion

I denne øvelse skal vi bruge concatenation til at implementere en bit-shifter og bit-rotate. Formålet er at se hvordan man kan bruge & til at sætte datatyper sammen og flytte rundt på bit. Derudover bliver *resize* brugt til at tilpasse resultatet til outputtet.

Design og Implementering

```
1  Library ieee; -- library context clause
2  Use ieee.std_logic_1164.all; -- use clause
3  use ieee.numeric_std.all;
4
5  entity shift_div is
6     port (
7        a : in std_logic_vector (7 downto 0);
8        a_shl, a_shr, a_ror : out std_logic_vector (7 downto 0)
9     );
10 end shift_div;
11
12 architecture bit_operations of shift_div is
13       signal x_shr : std_logic_vector(9 downto 0); -- Help vector for shift right
14       signal x1_ror : std_logic_vector(2 downto 0); -- Help vector 1 for rotate right
15       signal x2_ror : std_logic_vector(10 downto 0); -- Help vector 2 for rotate right
16    begin
17       -- Shift one to left:
18       a_shl <= std_logic_vector(resize((unsigned(a & "0")), 8));
19
20       -- Shift two to right:
21       x_shr <= ("00" & a);
22       a_shr <= x_shr(9 downto 2);
23
24       -- Rotate three times to right:
25       x1_ror <= a(2 downto 0);
26       x2_ror <= (x1_ror & a);
27       a_ror <= x2_ror(10 downto 3);
28
29 end bit_operations;
30
```

Figur 1 Bit shifter implementering

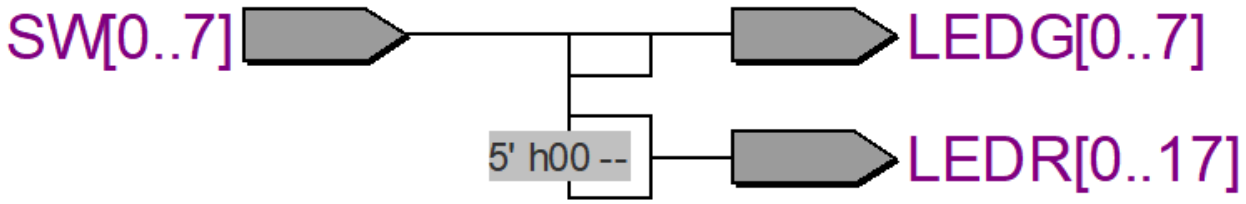
```
1  Library ieee; -- library context clause
2  use ieee.std_logic_1164.all; -- use clause
3  use ieee.numeric_std.all;
4  use work.all;
5
6  entity testbench is
7     port (
8        SW : in std_logic_vector(7 downto 0);
9        LEDR : out std_logic_vector(17 downto 0);
10       LEDG : out std_logic_vector(7 downto 0)
11    );
12 end testbench;
13
14 architecture test of testbench is
15    begin
16       uut: entity shift_div port map
17       (a => SW, a_shl => LEDR(17 downto 10), a_shr => LEDR(7 downto 0), a_ror => LEDG);
18 end test;
```

Figur 2 Bit shifter testbench

Resultater

Flow Summary	
Flow Status	Successful - Wed Feb 26 09:55:31 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	shift_div
Top-level Entity Name	testbench
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	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 3 Bit shifter compilation report



Figur 4 Bit shifter technology map viewer

Figur 5 Bit shifter 4 bit shiftes på DE2

Figur 6 Bit shifter 8 bit shiftes på DE2

LEDR 10 – 17: a_shl, Shift Left. LEDR 0 – 7: a_shr, Shift Right. LEDG 0 – 7: a_ror, Rotate Right.

Diskussion

Vi kan i vores compilation report se at der ikke er brugt nogen logiske elementer til at implementere denne arkitektur. Dette kan også ses i technology map vieweren hvor der ikke er nogen logiske gates.

Konklusion

Vi kan ud fra compilation reporten og technology map vieweren se at der ikke bliver brugt logiske elementer til at lave denne implementering. Det er ikke nødvendigt da der bare bliver lavet nogle forbindelser mellem indgangene og udgangene som gør at de forskellige bits bliver forskudt.

Multiplication

Introduktion

I denne øvelse kigger vi nærmere på hvordan synthesizeren tildeler logiske gates til vores gange funktion, når vi bruger DSP Block Balancing. Formålet er at se hvordan antallet af logiske gates stiger i forhold til antal bit vi ganger sammen. Derudover skal vi vise at der er nogle tal, der er nemmere at gange sammen end andre, for FPGA´eren.

Design og Implementering

```
1   Library ieee; -- library context clause
2   Use ieee.std_logic_1164.all; -- use clause
3   use ieee.numeric_std.all;
4   Use work.all; -- Der hvor vores egne entities ligger gemt
5
6   entity mult is
7      port (
8         A, B : in std_logic_vector (7 downto 0);
9         Prod : out std_logic_vector (15 downto 0)
10     );
11  end mult;
12
13  architecture eight_bit_mult of mult is
14     begin
15        Prod <= std_logic_vector(unsigned(A) * unsigned(B));
16  end eight_bit_mult;
17
18
```

Figur 7 Produkt af input A og B

```
1   Library ieee; -- library context clause
2   Use ieee.std_logic_1164.all; -- use clause
3   use ieee.numeric_std.all;
4   Use work.all; -- Der hvor vores egne entities ligger gemt
5
6   entity mult is
7      port (
8         A : in std_logic_vector (7 downto 0);
9         Prod : out std_logic_vector (15 downto 0)
10     );
11  end mult;
12
13  architecture one_bit_mult of mult is
14     begin
15        Prod <= std_logic_vector(unsigned(A) * 10);
16  end one_bit_mult;
17
```

Figur 8 Produkt af input A og tal

Resultater

Ud fra compilation rapporten kan vi se antallet af logiske elementer vi har brugt for at kunne gange f.eks. to 8-bit input med hinanden.

Flow Summary	
Flow Status	Successful - Wed Feb 26 10:27:42 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	mult
Top-level Entity Name	mult
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	103 / 33,216 (< 1 %)
Total combinational functions	103 / 33,216 (< 1 %)
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 %)

Figur 9 Compilation rapport med 8-bit

Flow Summary	
Flow Status	Successful - Wed Feb 26 11:21:56 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	mult
Top-level Entity Name	mult
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,430 / 33,216 (4 %)
Total combinational functions	1,430 / 33,216 (4 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	128 / 475 (27 %)
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 10 Complilation rapport med 32-bit

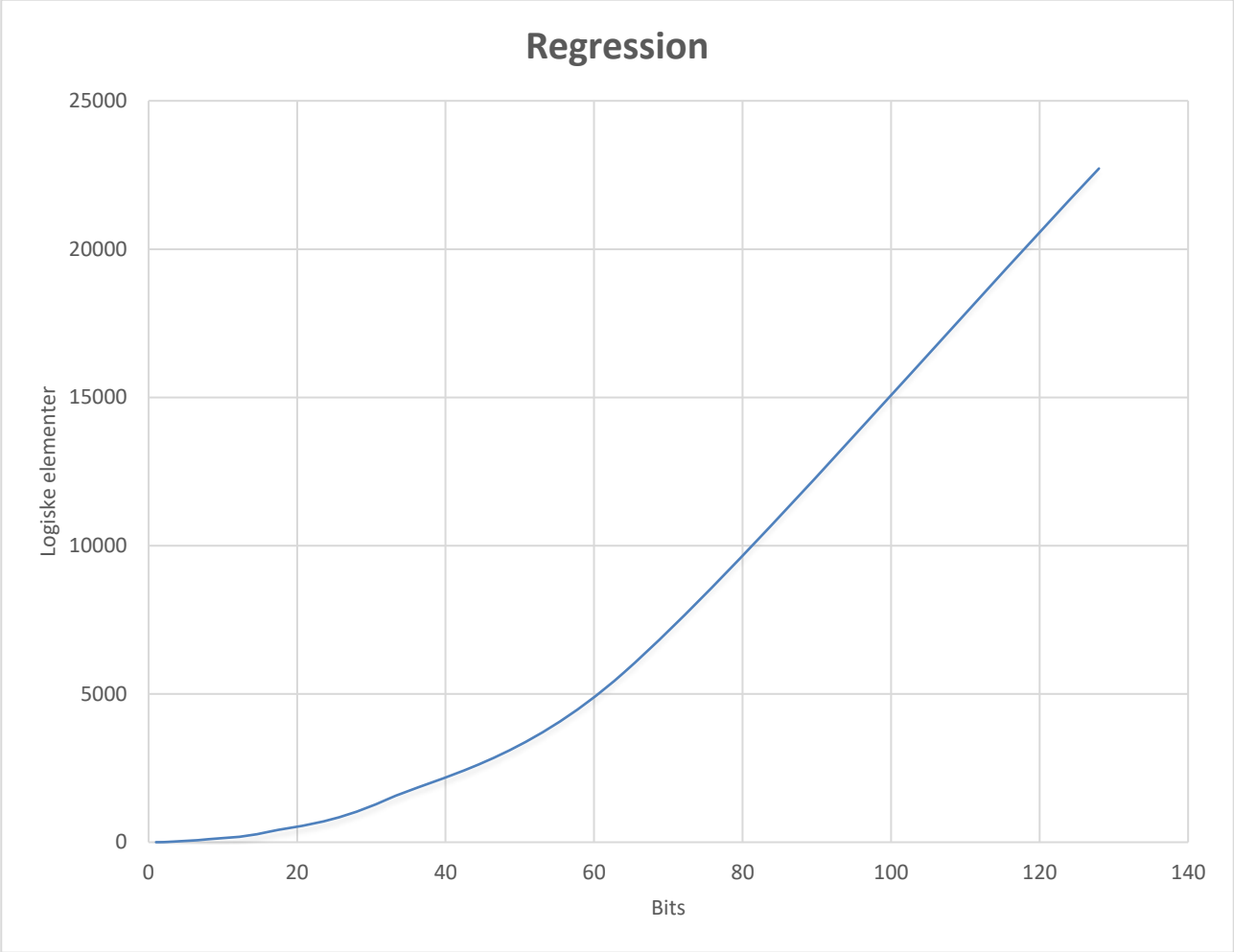
Regression

Alle tallene har vi samlet i en tabel og lavet regression på, for at se hvordan udviklingen skete.

X	Y
1	1
2	4
3	17
4	31
8	101
16	343
32	1430
64	5718
128	22718

Potens regression udført vha. CAS-værktøjet WordMat: $R^2 = 0{,}9931411$

$$y = 1{,}291392 \cdot x^{2{,}042669}$$



Vi kan se at en potens regression passer med 99,3%, når vi sammenligner vores forbrug af logiske elementer i forhold til hvor mange bits vi ganger med på inputtet.

Produkt med tal

Ved at gange med tallene 2, 3, 4, 7, 8 og 10, kunne vi se at antallet logiske gates varierede efter om tallet vi gangede med et tal, der gik op i 2-potens tallene (2, 4, 8, 16…) eller andre tal.

Flow Summary	
Flow Status	Successful - Wed Feb 26 11:10:35 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	mult
Top-level Entity Name	mult
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	24 / 475 (5 %)
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 11 Produkt af input A og tallet 4

Flow Summary	
Flow Status	Successful - Wed Feb 26 11:09:52 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	mult
Top-level Entity Name	mult
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	18 / 33,216 (< 1 %)
Total combinational functions	18 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	24 / 475 (5 %)
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 12 Produkt af input A og tallet 7

Diskussion

Vi kan ud fra øvelsens resultater se at FPGA´eren bruger virkelig mange logiske elementer til at udføre multiplikation. Når tallene bliver større stiger antallet af logiske elementer eksponentielt som vi har forsøgt at vise ved regression ovenfor.

Konklusion

Vi kan se ud fra vores resultater, at når vi bruger DSP Block Balancing, hvor vi tvinger synthesizeren til at bruge logiske gates, frem for den indbygget multiplier, så bruger vi en del logiske gates til at udføre regnestykket. Vi kan konkludere at antallet af logiske gates stiger potentielt med antallet af vores bit input. Vi kunne se at hvis vi gangede med 2-potens tallene, skulle vi ikke bruge nogen logiske gate, da den kunne lave regnestykket ved at bit-shift inputtet til outputtet.

Binary to 7-Segment Decoder Using Selected Signal Assignment ("WITH-SELECT")

Introduktion

I denne øvelse kigger vi nærmere på hvordan vi kan bruge "With-Select" statement til at udvikle en BCD til 7-segment driver. Formålet er at kunne bruge vores 4-bit input til at styre et 7-bit output.

Design og Implementering

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity bin2sevenseg is
5      port(
6         bin   : IN std_logic_vector(3 downto 0);
7         Sseg  : OUT std_logic_vector(6 downto 0)
8         );
9   end bin2sevenseg;
10
11  architecture logic_driver of bin2sevenseg is
12
13  begin
14     with bin select Sseg <=
15           "1000000" when "0000",  --0
16           "1111001" when "0001",  --1
17           "0100100" when "0010",  --2
18           "0110000" when "0011",  --3
19           "0011001" when "0100",  --4
20           "0010010" when "0101",  --5
21           "0000010" when "0110",  --6
22           "1111000" when "0111",  --7
23           "0000000" when "1000",  --8
24           "0010000" when "1001",  --9
25           "0001000" when "1010",  --a
26           "0000011" when "1011",  --b
27           "1000110" when "1100",  --c
28           "0100001" when "1101",  --d
29           "0000110" when "1110",  --e
30           "0001110" when "1111",  --f
31           "1111110" when others;  ---
32  end logic_driver;
```

Figur 13 BCD til 7-segment med With-Select

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity bin2sevenseg_tester is
6     port(
7        SW    : in std_logic_vector(3 downto 0);
8        HEX0  : out std_logic_vector(6 downto 0)
9     );
10 end bin2sevenseg_tester;
11
12 architecture logic_driver_tester of bin2sevenseg_tester is
13
14 begin
15    U1: entity bin2sevenseg port map(
16       bin => SW, Sseg => HEX0
17    );
18 end logic_driver_tester;
```

Figur 14 BCD til 7-segment testbench

Resultater

Switch 0 til 3 blev brugt til BCD input, for at teste om 7-segmentet viste de HEX-værdier vi ønskede.

Figur 15 DE2 test af BCD til 7-segment

BCD input = 0b0000 BCD input = 0b0111 BCD input = 0b1111

Diskussion

Vi kan se at med ”With-Select” er det nemt at vælge en værdi ud fra vores indgang, som kan blive tildele en udgang.

Konklusion

Vi kan konkludere at vi har lavet en BCD til 7-segment driver, som er nemt at bruge i fremtidige projekter.

The Conditional Signal Assignment ("WHEN-ELSE")

Introduktion

I denne øvelse kigger vi nærmere på brugen af "When-Else" statement, som en multiplexer. Formålet er at se hvordan en mux fungerer og hvordan latches kan påvirke systemet.

Design og Implementering

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use work.all;
4
5   entity hex_mux is
6      port(
7         tbin  : IN std_logic_vector(11 downto 0);
8         sel   : IN std_logic_vector(1 downto 0);
9         tsseg : OUT std_logic_vector(20 downto 0)
10     );
11  end hex_mux;
12
13  architecture logic_driver of hex_mux is
14     signal digit_seg  : std_logic_vector(20 downto 0);
15  begin
16     -- HEX display 0
17     U1: entity bin2sevenseg port map(
18        bin => tbin(3 downto 0), Sseg => digit_seg(6 downto 0)
19     );
20     -- HEX display 1
21     U2: entity bin2sevenseg port map(
22        bin => tbin(7 downto 4), Sseg => digit_seg(13 downto 7)
23     );
24     -- HEX display 2
25     U3: entity bin2sevenseg port map(
26        bin => tbin(11 downto 8), Sseg => digit_seg(20 downto 14)
27     );
28     -- Select with KEY 0 and 1
29     tsseg <= "11111110000000101011" when sel = "11"   else  -- On
30              digit_seg              when sel = "10"   else  -- Segment
31              "00001100101111101011111" when sel = "01";       -- Err
32
33  end logic_driver;
```

Figur 16 HEX MUX kode

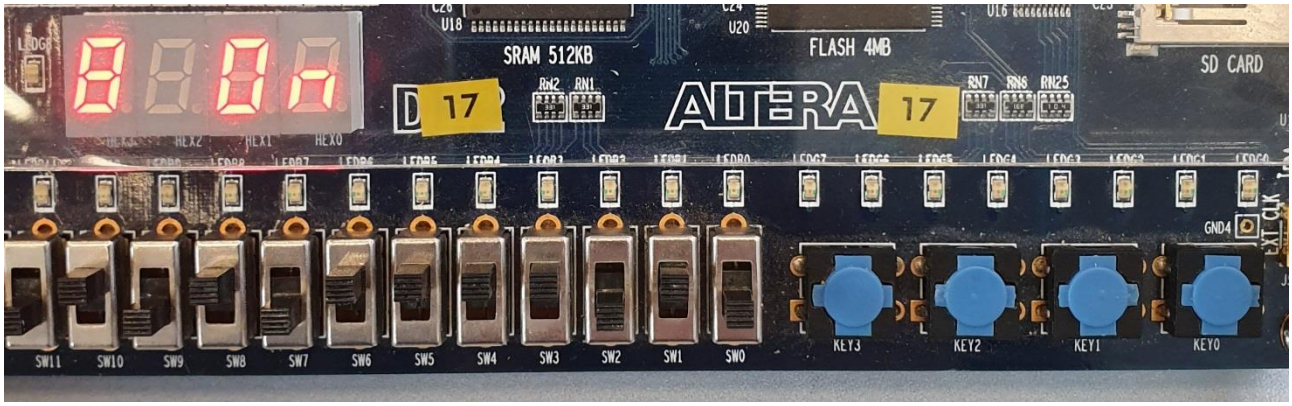
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity hex_mux is
6     port(
7        tbin  : IN std_logic_vector(11 downto 0);
8        sel   : IN std_logic_vector(1 downto 0);
9        tsseg : OUT std_logic_vector(20 downto 0)
10    );
11 end hex_mux;
12
13 architecture logic_driver of hex_mux is
14    signal digit_seg  : std_logic_vector(20 downto 0);
15 begin
16    -- HEX display 0
17    U1: entity bin2sevenseg port map(
18       bin => tbin(3 downto 0), Sseg => digit_seg(6 downto 0)
19    );
20    -- HEX display 1
21    U2: entity bin2sevenseg port map(
22       bin => tbin(7 downto 4), Sseg => digit_seg(13 downto 7)
23    );
24    -- HEX display 2
25    U3: entity bin2sevenseg port map(
26       bin => tbin(11 downto 8), Sseg => digit_seg(20 downto 14)
27    );
28    -- Select with KEY 0 and 1
29    tsseg <= "11111111000000101011" when sel = "11"   else  -- On
30             digit_seg              when sel = "10"   else  -- Segment
31             "00001100101111101011111" when sel = "01"   else  -- Err
32             "01111111011111101111111";                       -- --
33
34 end logic_driver;
```

Figur 17 HEX MUX kode med latch fixet

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity hex_mux_tester is
6     port(
7        SW    : in std_logic_vector(11 downto 0);
8        KEY   : in std_logic_vector(1 downto 0);
9        HEX0  : out std_logic_vector(6 downto 0);
10       HEX1  : out std_logic_vector(6 downto 0);
11       HEX2  : out std_logic_vector(6 downto 0)
12    );
13 end hex_mux_tester;
14
15 architecture logic_driver_tester of hex_mux_tester is
16
17 begin
18    U1: entity hex_mux port map(
19       tbin => SW, sel => KEY, tsseg(6 downto 0) => HEX0, tsseg(13 downto 7) => HEX1, tsseg(20 downto 14) => HEX2
20    );
21 end logic_driver_tester;
```

Figur 18 HEX MUX testbench

Resultater



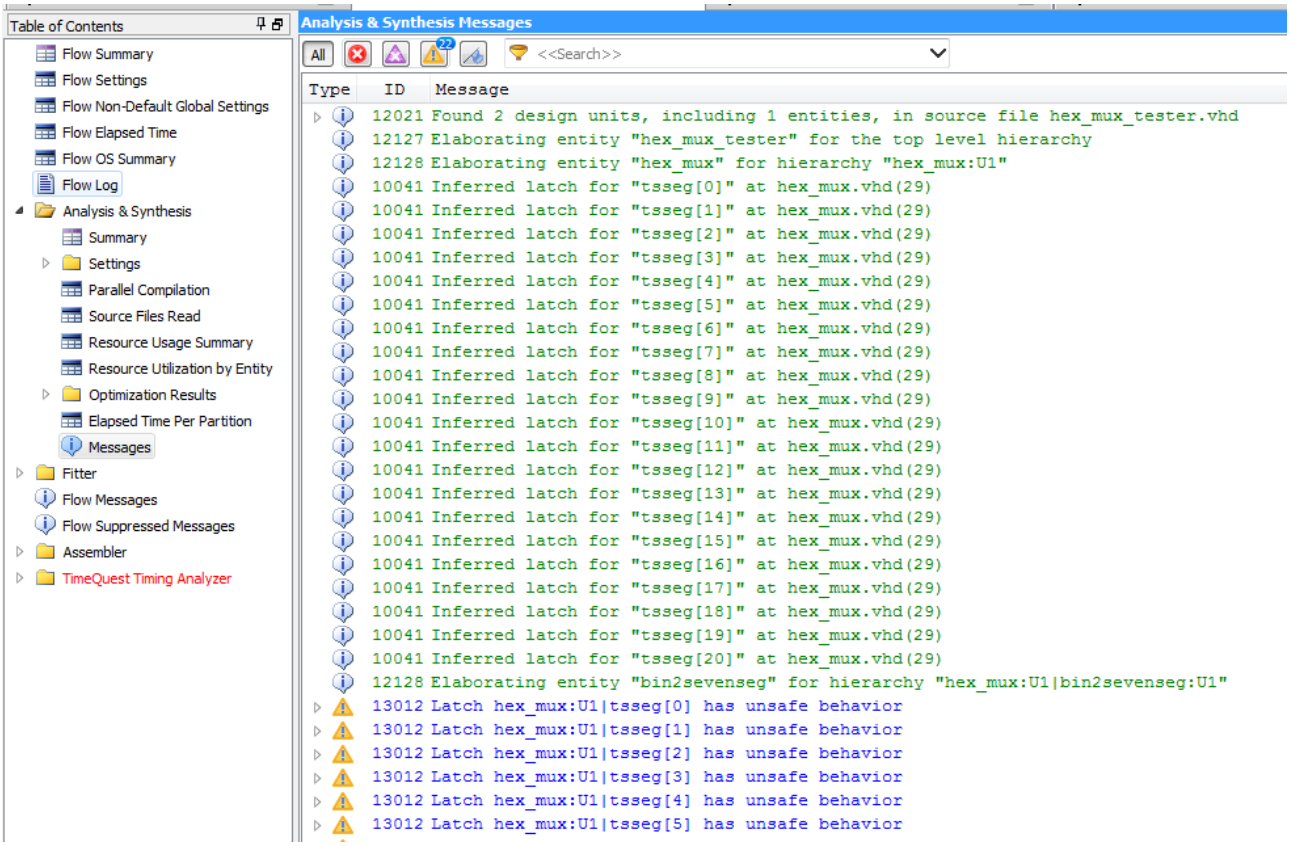
Figur 19 HEX MUX On

Figur 20 HEX MUX Err

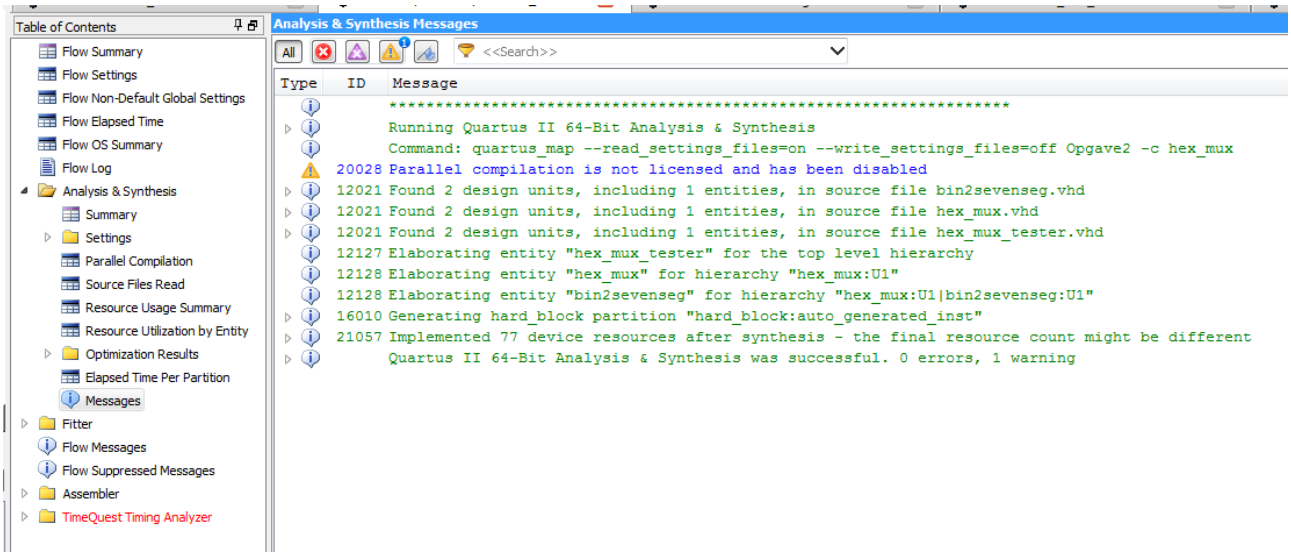
Figur 21 HEX MUX tal

Finding latch

Vi har søgt efter latches i vores complilation rapport og fandt at der var flere advarsler omkring inferred latch. Det fiksede vi ved at rette koden, som vist i Figur 17.



Figur 22 HEX MUX fundet inferred latch



Figur 23 HEX MUX fixed latch issues

Diskussion

Vores ”When-Else” statement kan skabe problemer med inferred latches, hvis den ikke er implementeret korrekt. Problemet i koden på Figur 16 er at der efter den sidste when statement ikke er en betingelse for default værdi, og det skaber en situation hvor den kan være i tvivl om hvad den skal sende ud. I det tilfælde vil den sende dens tidligere værdi ud, hvilket resulterer i en latch.

Konklusion

Vi kan konkludere at det er vigtigt at få en default værdi på vores statement, ellers vil vi få en masse inferred latches, som kan gemme værdier vi ikke ønsker. Ved at bruge koden på Figur 17, vil der være en betingelse for alle andre værdier, som ikke er defineret, hvor den outputer ”---”. Dette løser vores inferred latch problem.

Table Lookup

Introduktion

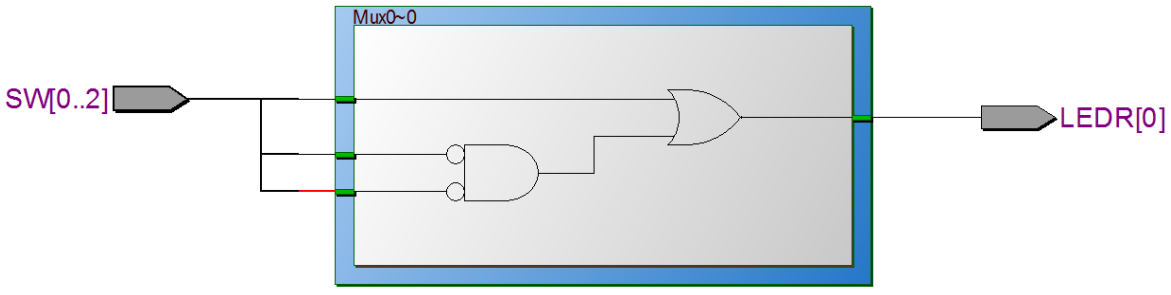
I denne øvelse vil vi implementere en look up tabel ved hjælp af et constant array hvor vi vælger output ved hjælp af index værdien, og har de forskellige outputs liggende i arrayet. I de tidligere øvelser har vi brugt ”with-select” og ”when-else” men for at lave en look up tabel som denne kan det være simplere at lave det med et constant array.

Design og Implementering

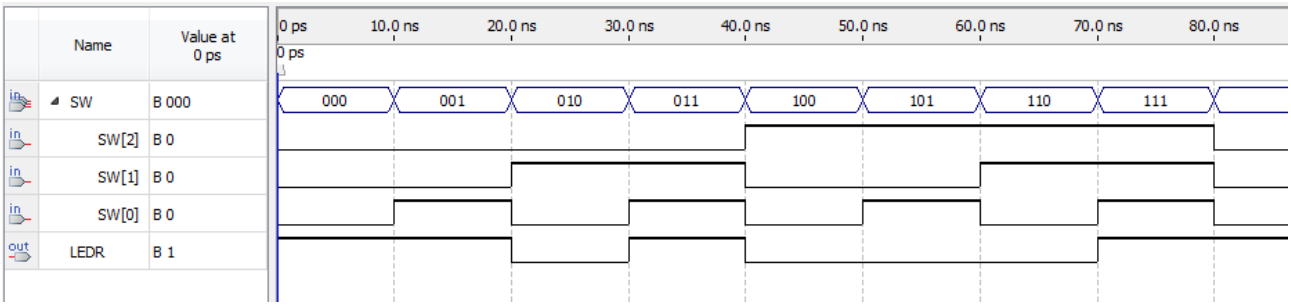
```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity lookUp is
6      port(
7         SW    : IN std_logic_vector(2 downto 0);
8         LEDR  : OUT std_logic_vector(0 downto 0)
9      );
10  end lookUp;
11
12  architecture logic_driver of lookUp is
13     type steps is array (0 to 7) of std_logic_vector(0 downto 0);
14     constant led_step : steps := ("1", "1", "0", "1", "0", "-", "-", "1");
15
16  begin
17     LEDR <= led_step(to_integer(unsigned(SW)));
18
19  end logic_driver;
```

Figur 24 Implementering af look up tabel

Resultater

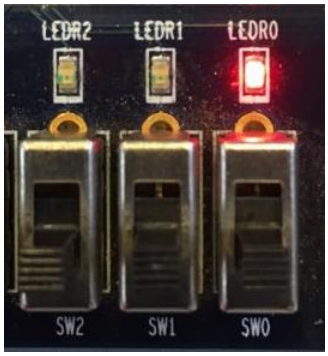


Figur 25 Look up tabel technology map viewer



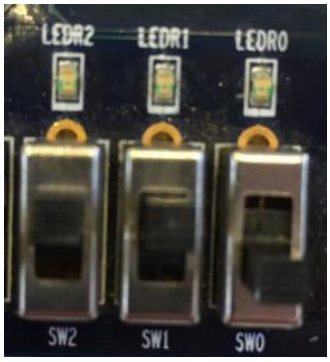
Figur 26 Look up tabel functional test

Figur 27 DE2 board 010 = 0



Figur 28 DE2 board 000 = 1

Figur 29 DE2 board 101 = 1 (Don´t care)



Figur 30 DE2 board 110 = 0 (Don´t care)

Diskussion

Når vi laver implementeringen, kan vi se i technology map at den bliver implementeret som en or-gate og en and-gate med inverterede indgange. Den bruger altså ét logisk element til at implementere look up tabellen. Vores to værdier som giver ”don´t care” output giver noget forskelligt når vi tester det på DE2 boardet i forhold til vores funktionelle analyse.

Konklusion

Vi kan ud fra øvelsen se at der er forskel på ”don´t care” outputs i den funktionelle analyse og den fysiske implementering på DE2 boardet. Dette er fordi den i analysen endnu ikke ved hvordan det bliver mappet på FPGA´eren, derfor antager den bare at de er ”0”. Når selve mappingen sker har vi nu givet programmet ”frie hænder” til at gøre med de to ”don´t care” kombinationer som den vil. Det vil sige at den kan vælge den implementering der er mest simpel uden at tage hensyn til de to kombinationer. I dette tilfælde vil det så sige at den ene bliver ”1” og den anden bliver ”0”.