DSD Journal 2

Gruppe 39
AU - SWT - Digital Systemdesign

Filip Rolighed Christensen	201907625
Sigurd Skov Jensen	201804402

18. marts 2020

Exercise 3

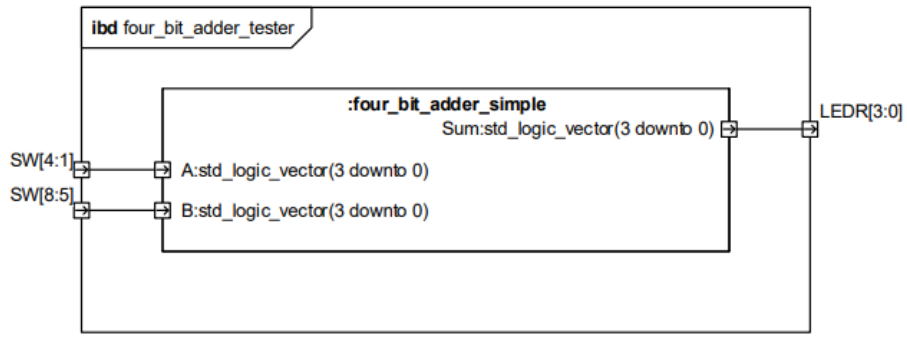
3.1 Signed and Unsigned Arithmetic

Introduktion

I forrige journal blev der opbygget en 4-bit-adder ud fra halfaddere. I denne øvelse udnyttes aritmetik til at opbygge den samme funktion meget simplere.

Design og Implementering

Four_bit_adder_simple
Firebit-adderen er bygget op ud fra dette blokdiagram, med 8 input og 4 output.



Figur 1: ibd af four bit adder testeren, taget fra opgavebeskrivelsen

Inputs er inddelt i to dele af 4, og formået med adderen er at lægge de to værdier til hinanden.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity four_bit_adder_simple is
6  Port (
7      A : in std_logic_vector(3 downto 0);
8      B : in std_logic_vector(3 downto 0);
9      sum : out std_logic_Vector(3 downto 0)
10     );
11 end four_bit_adder_simple;
12
13 architecture unsigned_impl of four_bit_adder_simple is
14 begin
15     sum <= std_logic_vector(unsigned(A) + unsigned(B));
16 end unsigned_impl;
17
18 architecture signed_impl of four_bit_adder_simple is
19 begin
20     sum <= std_logic_Vector(signed(A) + signed (B));
21 end signed_impl;
22
```

Figur 2: Kode til den simple four-bit adder

Vi laver en fire bit adder ved først at instantiere en entity.
Denne entity har to inputs i 4 bit og et output i 4 bit.

Unsigned Architecture

Arkitekturen for unsigned lægger unsigned A og unsigned B sammen som en std_logic_vector, ved konvertering og gemmer det i sum. Det lægges ind i en std_logic_vector, fordi, som man kan se i entitien, er sum defineret som en std_logic_vector og de er nødt til at være den samme type.
En std_logic_vector kan imidlertid ikke lægges sammen, da det er et array. Så vi er nødt til at kalde hver individuel værdi, og gemme dem i de tilsvarende pladser, hvilket compileren kan hjælpe os med, igennem metoden vist ovenover.
At den er unsigned betyder blot at den lægger sammen (binært i dette tilfælde). Så vi kan nu tælle fra 0 til 15

Signed Architecture

Arkitekturen for signed har samme generelle logik bag sig som unsigned, beskrevet ovenover, forskellen er blot i at værdien er signed, hvilket betyder at når den yderste bit er sat, vil talrækken tælle nedad, men i negativ. Således at 0001 betyder 1, 0111 betyder 7, 1000 betyder -7 og 1111 betyder -1.
Så vi kan tælle fra -7 til 7.

Tester

Selve testeren anvender først og fremmest vores tidligere entity og architecture igennem linje 2 og 3.

Derefter definerer vi en test-entity, hvor vi definerer nogle porte, således sættes 8 porte af til inputs i form af switches og 4 sættes af til outputs i form af LED'er

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity four_bit_adder_simple_tester is
6  Port (
7      SW : in std_logic_vector(7 downto 0);
8      LEDR : out std_logic_Vector(3 downto 0)
9      );
10 end four_bit_adder_simple_tester;
11
12 architecture tester of four_bit_adder_simple_tester is
13 begin
14     a : entity four_bit_adder_simple(signed_impl) port map (
15     A => sw(3 downto 0),
16     B => sw(7 downto 4),
17     sum => LEDR(3 downto 0));
18 end tester;
19
```

Figur 3: En tester til koden i figur 2

Arkitekturen af testeren definerer først hvilken entity, og i parentes hvilken arkitektur, vi anvender, herefter hvor input og outputs skal være, ved A som de første 4 switches, B som de næste 4 switches og sum som de 4 første LED'er.

Hvis vi er interesseret i at test begge arkitekturer, skal vi blot ændrer i arkitekturen, under a's kald af entiteten, inde i parentesen er nemlig defineret hvilken arkitektur vi anvender.

Udvidet four bit adder

Dog, med den tidligere forklaret logik, vil vi ikke kunne håndtere simpel overflow, så vi udvider vores entitet lidt, med formålet at til sidst lave en full-adder.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity four_bit_adder_unsimple is
6  Port (
7      cin : in std_logic;
8
9      A : in std_logic_vector(3 downto 0);
10
11     B : in std_logic_vector(3 downto 0);
12
13     sum : out std_logic_Vector(3 downto 0);
14
15     cout : out std_logic
16     );
17 end four_bit_adder_unsimple;
```

Figur 4: En fulladder

Dette gør vi ved at lave endnu et input (cin) og endnu et output (cout).
cin repræsenterer et input denne full-adder kan have fået fra et andet system, eks. en anden adder der havde overflow.
cout repræsenterer vores systems overflow.

Unsigned

Vi er nødt til at ændre vores arkitektur, til at kunne håndtere flere output og input

```
18
19 architecture unsigned_impl of four_bit_adder_unsimple is
20 signal ud : std_logic_Vector(4 downto 0);
21 begin
22     ud <= std_logic_vector(resize(unsigned(A), 5) + resize(unsigned(B), 5) + ("0000" & cin));
23
24     sum <= ud(3 downto 0);
25
26     cout <= ud(4);
27 end unsigned_impl;
28
```

Figur 5: Arkitektur til unsigned four bit adder

Først skaber vi et internt signal inde i arkitekturen, som vi kalder ud.
Ud skal bestå af summen af værdien A, hvis std_logic_vector skal udvides til at kunne indeholde 5 værdier, B, hvis std_logic_vector skal udvides til at kunne indeholde 5 værdier, og cin, som kun er et enkelt bit, så vi er nødt til at putte 4 0'ere foran, således den logisk minder om de andre værdier. Det er nemlig ikke muligt at lægge værdier sammen, hvis de ikke har samme bitcount.
Alle disse værdier lægges sammen og gemmes i ud.
Sum skal så defineres som de første 4 værdier af ud. Altså de værdier vores adder kan finde frem til (0 til 15) og cout, repræsenterer et overflow, en carry som kunne sendes videre til den næste adder.

Signed

Hvis vi dog gerne vil kunne lave en adder der kan håndtere negative tal, er arkitekturen, som beskrevet i den tidligere del, nødt til at blive ændret en smule.
Så vi skriver blot signed i stedet for unsigned.

```
29  architecture signed_impl of four_bit_adder_unsimple is
30  signal ud : std_logic_Vector(4 downto 0);
31  begin
32      ud <= std_logic_vector(resize(signed(A), 5) + resize(signed(B), 5) + ("0000" & cin));
33
34      sum <= ud(3 downto 0);
35
36      cout <= ud(4);
37  end signed_impl;
```

Figur 6: Arkitektur til signed four bit adder

Samme logik som tidligere. Nu fungerer de 4 bit som de gjorde tidligere, men hvis addition forårsager et overflow, så har vi cout til at identificere dette.

Tester

Vores tester minder meget om den tidligere, med undtagelse af at vi sætter 9 switches af, da den første skal sættes som cin og vi sætter 5 LED'er af, hvor den yderste er cout.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use work.all;
4
5   entity four_bit_adder_unsimple_tester is
6   Port (
7       SW : in std_logic_vector(8 downto 0);
8       LEDR : out std_logic_Vector(4 downto 0)
9       );
10  end four_bit_adder_unsimple_tester;
11
12  architecture tester of four_bit_adder_unsimple_tester is
13  begin
14      a : entity four_bit_adder_unsimple(unsigned_impl) port map (
15      cin => sw(0),
16      A => sw(4 downto 1),
17      B => sw(8 downto 5),
18      sum => LEDR(3 downto 0),
19      cout => LEDR(4)
20      );
21  end tester;
22
```

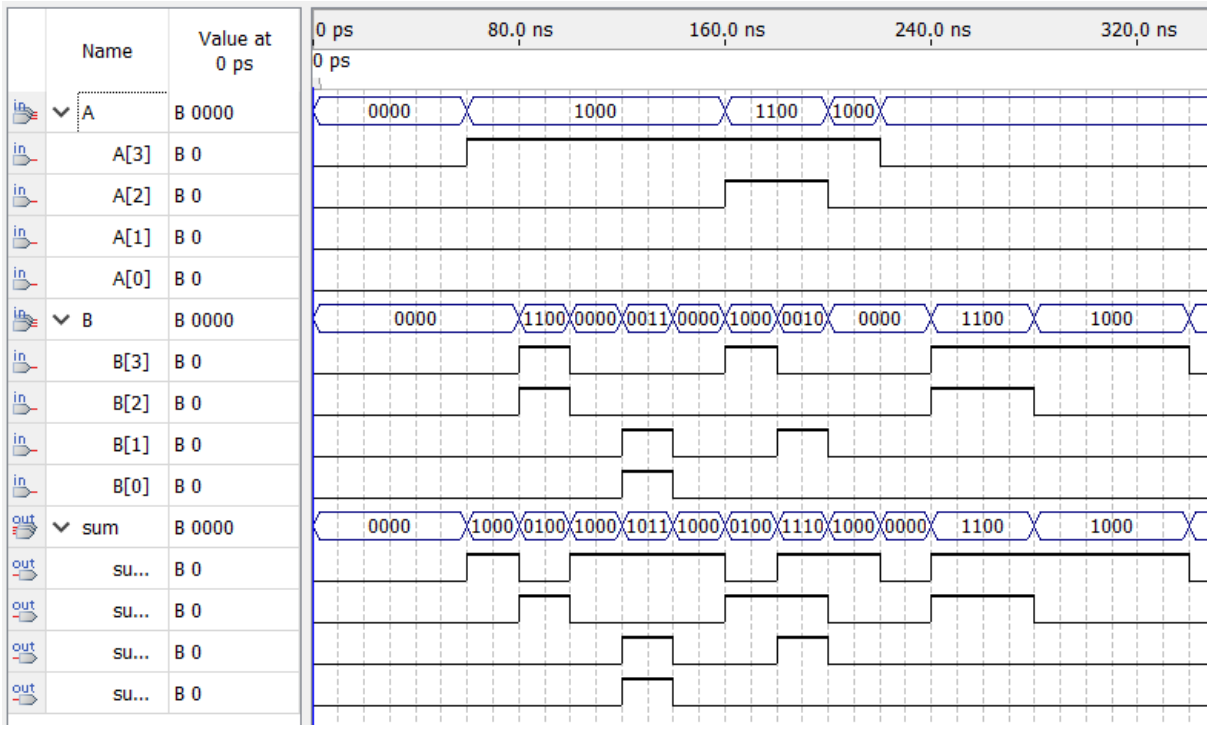
Figur 7: Testeren for begge (i arkitekturen kan begge arkitekturer af four_bit_adder_unsimple vælges

Arkitekturen af testeren har samme definitionslogik som beskrevet tidligere, så vi kan vælge den arkitektur vi ønsker at teste relativt ved blot at rette i a : entitity (<arkitektur>) - definitionen.
Dog sætter vi specifikt switch 0 til cin og LED 4 til cout.

Resultater og Diskussion

Unsigned

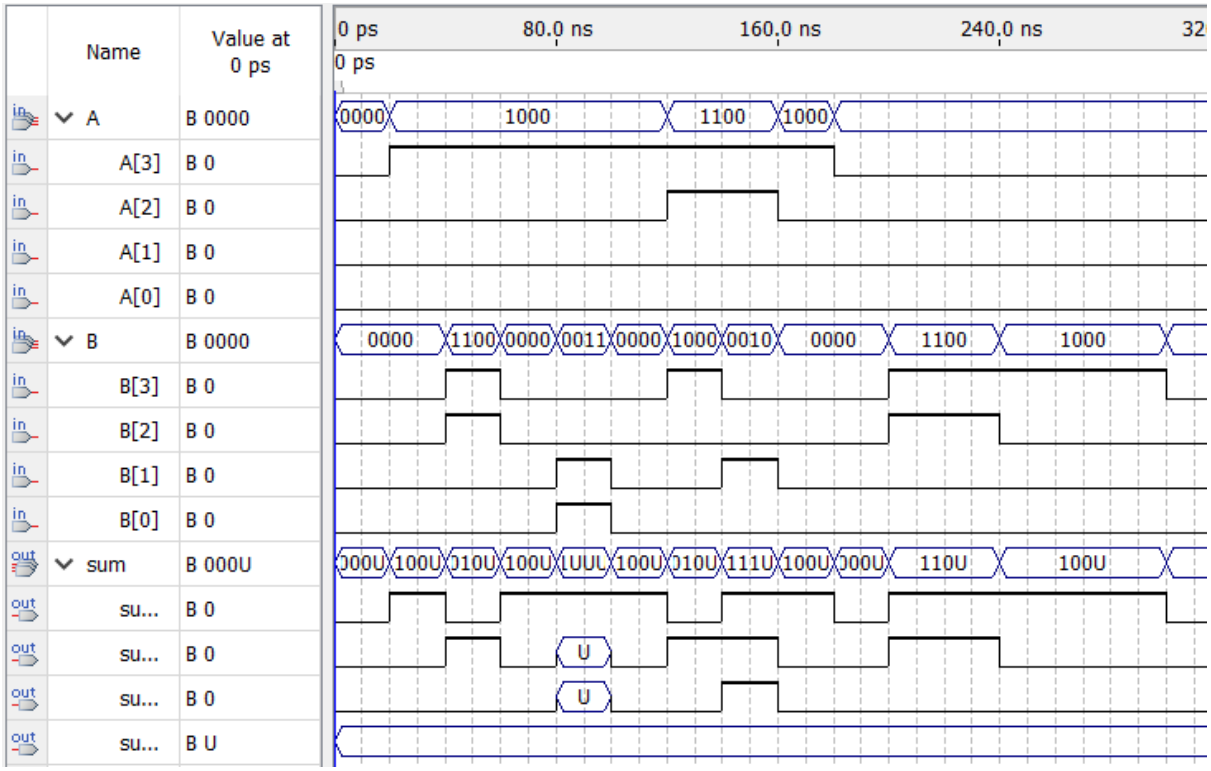
Herunder ses resultatet af en simulering af en simulering af projektet. Her ses det at sumvektoren påvirkes som forventet af input-vektorerne. Det bemærkes også at idet både A[3] og B[3] er høje, overflower summen, da den ikke kan gengive værdier højere end 2^4-1=15, hvilket er maksimalværdien i 4 bit.



Figur 8: Funktionel test af simpel unsigned four bit adder

Signed

I signed derimod kan vi få negative resultater.

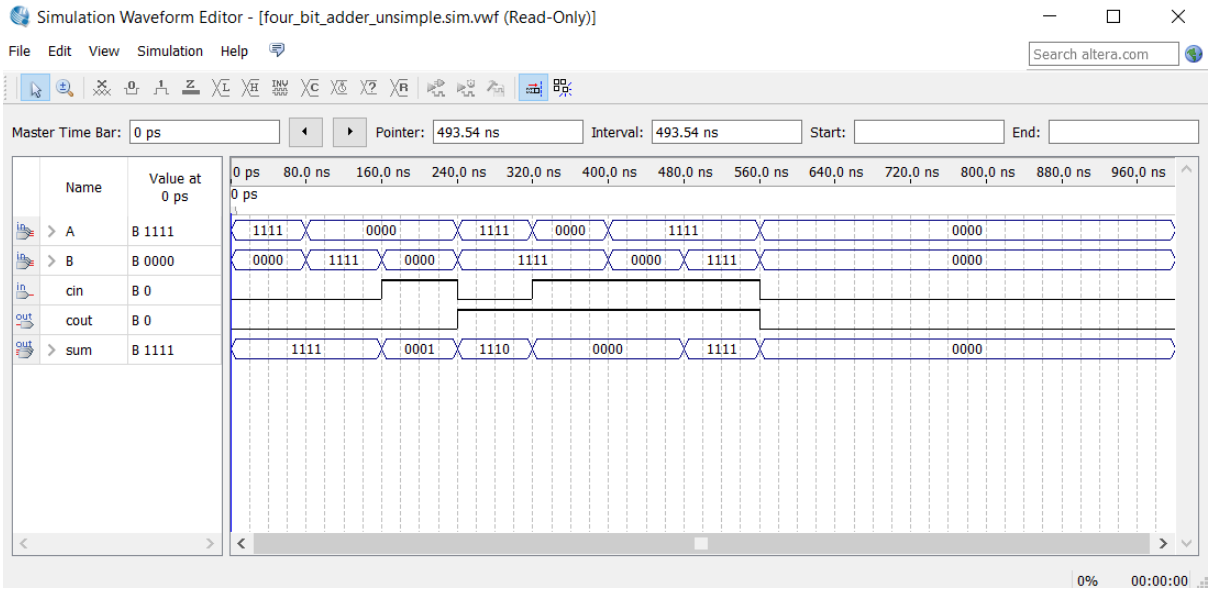


Figur 9: Funktionel test af simpel signed four bit adder

Full adder

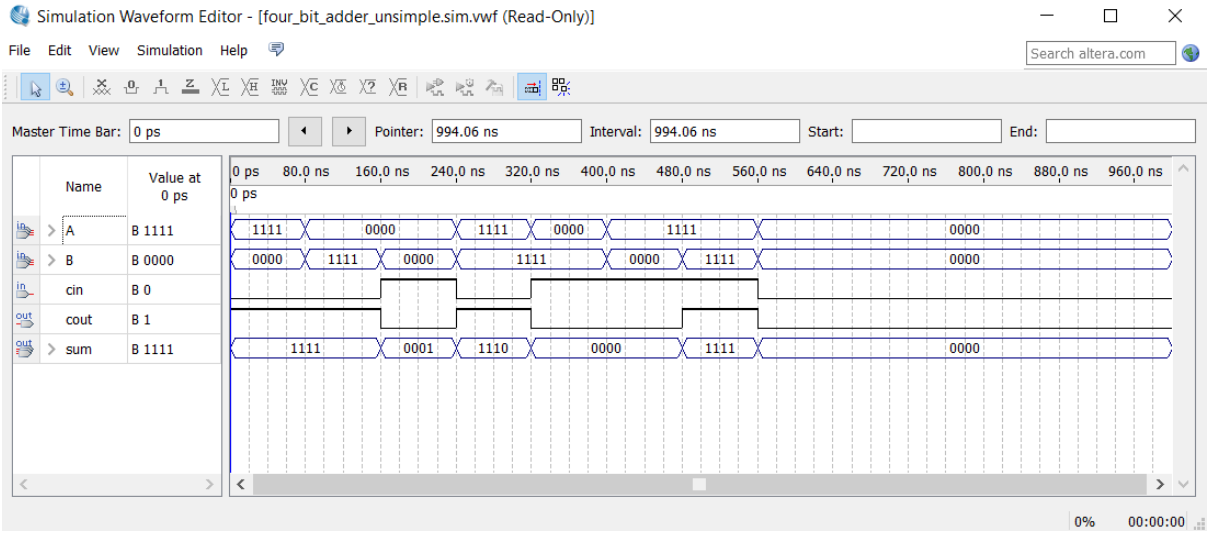
Simulation af unsigned

Som man kan se i vores fulladder, så tager vi os af vores overflow med vores cout, således er der en lineær progression og fulladderen kan ikke vise et forkert eller ufuldstændigt resultat.



Figur 10: En funktionel simulering af en unsigned full adder

Simulation af signed
Igen ser vi at vores system håndterer overflows, men fordi talrækken er lavere, så kan man se at den både overflower og går i negativ, hvis en af de 4 bit inputs alle bliver sat høje.



Figur 11: En funktionel simulation af en signed full adder

Modify the signed version and compare the results. How do they compare with the previous results? Do you see any effect of using signed/unsigned? Why?
Der forekommer en ændring, i form af når input-værdien er 1111+0000+1 overflowes ikke i signed, da 1-1 er 0.
I bund og grund, fordi nogle tal kan være negative i signed, kommer de endelige resultater i additionstesten til at være forskellige.

When do you think it can be beneficial to have multiple architectures?
Her skal vi blot ændrer på navnet i parentesen for at ændre arkitekturen, hvilket kan gøre prototyper og tests markant lettere.

Konklusion

For at anvende negative tal, skal vi opgive vores most important bit, dermed halverer vi den maksimalværdi vi kan arbejde med, til gengæld er den aktuelle mængde af værdier den samme, da disse blot er negative. Foruden øger signed kompleksiteten af programmet meget, da det kan tage længere tid at forstår for en programmør der ikke er garvet i signed og complements.
Samt kan det ses at man skal være meget forsigtig med sine typer i VHDL, da sproget ikke holder en i hånden, eller order type-discrepancies for programmøren.

3.2 Concatenation

Introduktion

I denne del udnyttes "&" operatoren til at sammensætte (concatinate) arrays.

Design og Implementeringen

```
1   library ieee;
2   use ieee.std_logic_1164.all;
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;
11
12  architecture code of shift_div is
13  begin
14  a_shl <= (a(6 downto 0)& "0");
15  a_shr <= ("00" & a(7 downto 2));
16  a_ror <= (a(2 downto 0) & a(7 downto 3));
17  end code;
```

Figur 12: Kode i VHDL, der benytter concatination til at rotere og skubbe bits.

Herover ses koden til opgaven, hvor entiteten er kopieret fra opgavesættet (linje 1-10). Der er skrevet en arkitektur, der på forskellig vis sammensætter arrayet:

- a_shl er de sidste 7 bit i a, med et 0 tilføjet til sidst.
- a_shr fungerer modsat, 00 med de første 6 bits i a tilføjet til sidst.
- a_ror er det de sidste 3 bit i a der står først, med de første 5 tilføjet til sidst.

Herunder ses test-koden, der bruger shift-div entiteten, med LED'er som output og Switches som input på DE-II

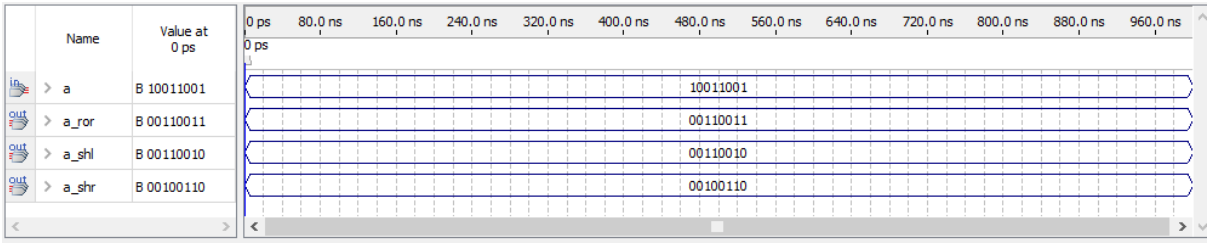
```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use work.all;
4
5   entity shift_div_tester is
6    port(
7       SW : in std_logic_vector(7 downto 0);
8       LEDR : out std_logic_vector(17 downto 0);
9       LEDG : out std_logic_vector(7 downto 0)
10      );
11  end shift_div_tester;
12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

Figur 13: Testprogram til entiteten shift_div

Her ses det at a_ror er tildelt de grønne LEDer, mens a_shr og a_shl er tildelt rød LED 17 til 10 og 7 til 0.

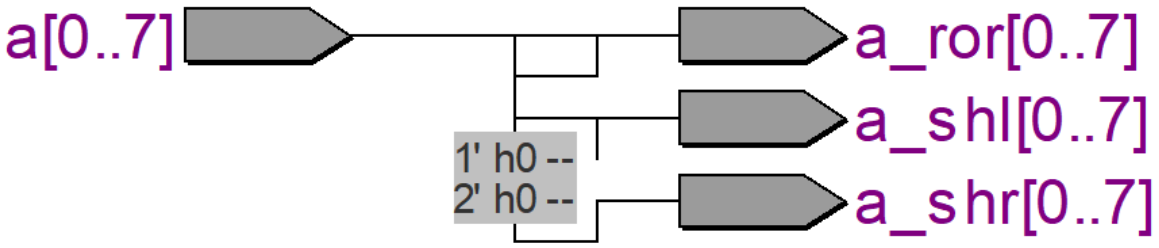
Resultater og Diskussion

For at teste funktionen af koden er der simuleret forskellige inputs til a. Herunder ses et af dem:

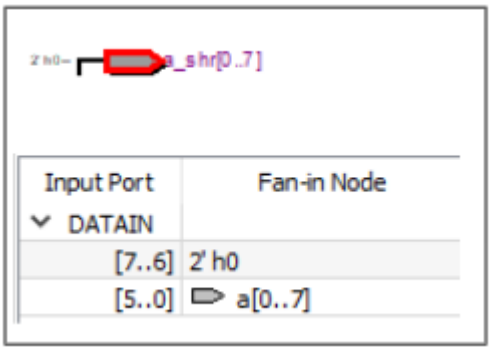


Figur 14: Simulering af programmet

Det ses i simuleringen at i a_ror variablen er tallene roteret, således at fx det yderste 1-tal til højre, nu står som det 3. tal fra venstre.
I a_shl er alle tallene rykket 1 plads til venstre, og 1-tallet yderst til venstre er "forsvundet".
I a_shr er alle tallene rykket 2 pladser til højre, og 1-tallet yderst til højre er "forsvundet".



Figur 15:
Herover er der inkluderet et technology map view, der i sig selv ikke siger særligt meget, dog opnås der flere detaljer i vinduet til venstre, der forklarer hver enkelt variabel fx:



Figur 16: Detalje omkring et markeret element, i dette tilfælde a_shr

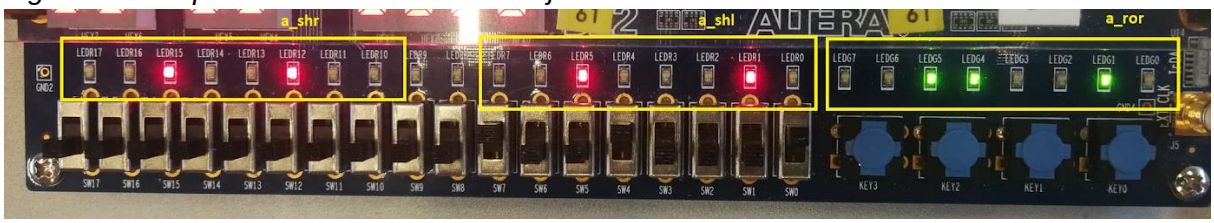
Flow Status	Successful - Wed Feb 26 09:15:08 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	shift_div
Top-level Entity Name	shift_div
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	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 %)

Figur 17: Compilation report for programmet

Som det ses herover, benyttes der ikke nogen logiske elementer i programmet. Dette kan virke overraskende, men der udføres ikke logiske operationer i programmet, der flyttes kun rundt på bits.

Slutteligt i denne resultatsektion ses fotografier af test på DE2 board. I det første billede er alle 8 kontakter sat til høj, og det ses at alle de grønne LEDer lyser, da alle bit er roteret 3 gange, og ingen input “går til spilde” ved a_shr og a_shl ses det at nogle af de 8 bit er skubbet “ud”

Figur 18: Test på DE-II: Alle 8 bit sat til høj



Figur 19: Test på DE-II: De yderste bit sat til høj

Figur 20: Test på DE-II: De to yderste bit sat til høj

Det bemærkes så også at LEDR 17, 16 og 0 (samt 8 og 9 der ikke udnyttes) forbliver

slukkede uanset input, hvilket er meget tydeligt i koden, hvor der trodsalt er indsat “0” på disse pladser.

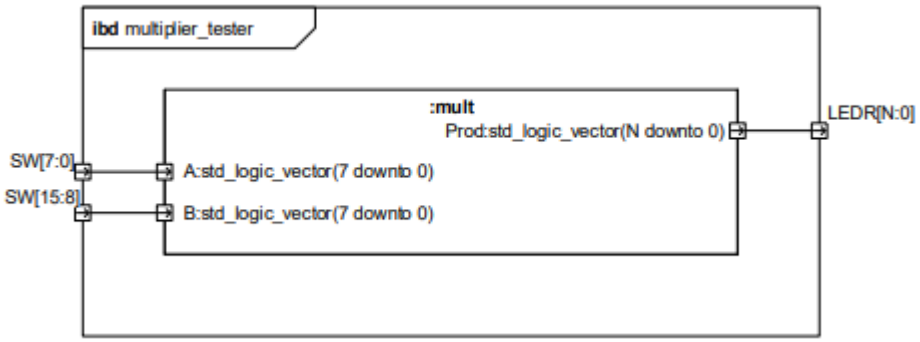
Konklusion

Det er lykkedes at rotere og skubbe bits, ved at bruge “Concationation” (&). med 1 input A dannes 3 output, der henholdsvis skubber bits til højre, venstre og roterer dem.

3.3 Multiplikation

Introduktion

Det er muligt at udføre alle normale aritmetiske funktioner i VHD. Her udforskes multiplikation, med operatoren *, ud fra følgende ibd:



Figur 21: ibd for multiplier tester

Design og Implementeringen

Herunder ses koden til multiplikatoren “mult”:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity mult is
6   port (
7       A,B : in std_logic_vector(7 downto 0);
8       Prod: out std_logic_vector(15 downto 0)
9   );
10  end;
11
12  architecture code of mult is
13  begin
14      Prod <= std_logic_vector(unsigned(A)*unsigned(B));
15  end code;
```

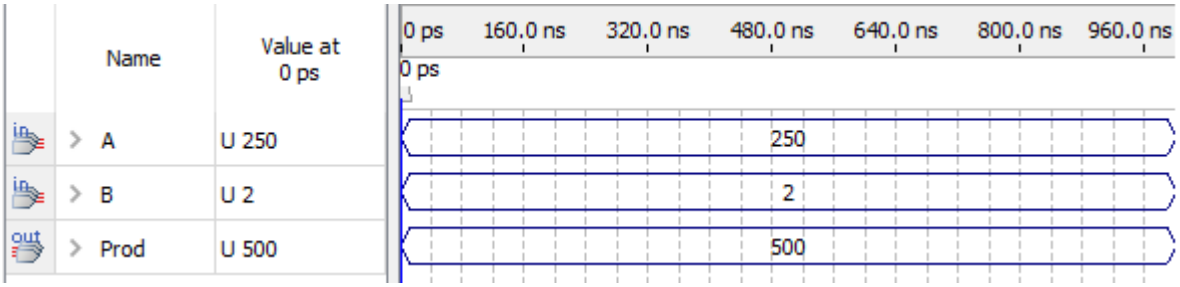
Figur 22: Kode for multiplikatoren “mult”

Det ses at output vektoren er dobbelt så stor som input vektorerne (16 bit i stedet for 8). Dette er nødvendigt, for at kunne have plads til det størst mulige resultat. (255*255)

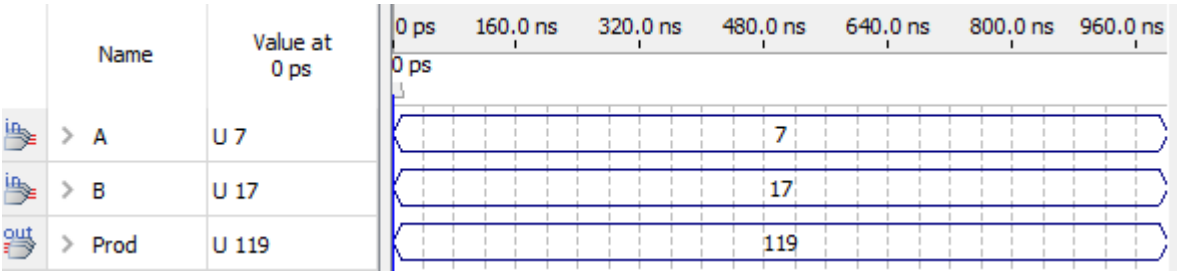
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity multiplier_tester is
6   port(
7      SW : in std_logic_vector(15 downto 0);
8      LEDR : out std_logic_vector(15 downto 0)
9      );
10 end multiplier_tester;
11
12 architecture tester of multiplier_tester is
13 begin
14    aa: entity mult port map(
15        A => SW(7 downto 0),
16        B => SW(15 downto 8),
17        Prod => LEDR(15 downto 0)
18    );
19 end tester;
```

Figur 23: Testprogram for "mult"

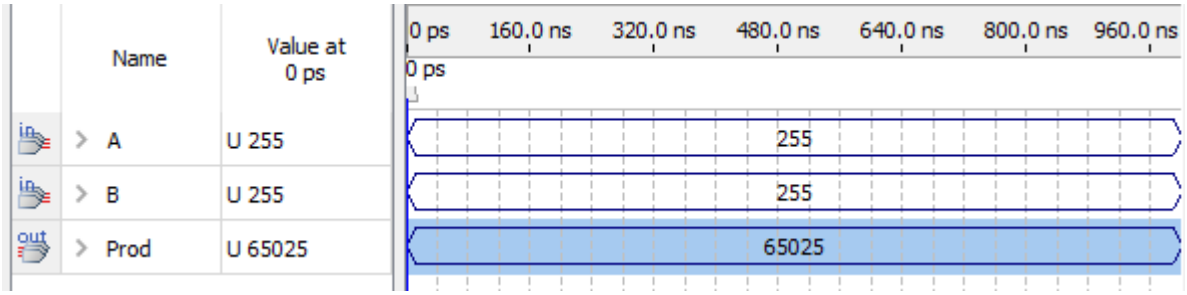
Resultater og Diskussion



*Figur 24: Simulering af multiplikatoren (250*2=500)*

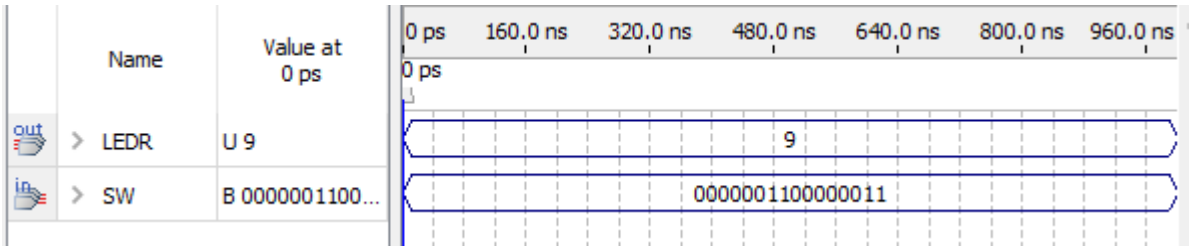


*Figur 25: Simulering af multiplikatoren (7*17=119)*



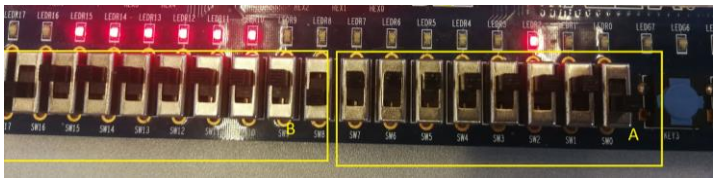
*Figur 26: Simulering af multiplikatoren (255*255=65025)*

Herover ses tre eksempler på multiplikation simuleret i Quartus. For nemhedens skyld står værdierne i decimal, så svarende nemmere kan verificeres. I alle tre tilfælde svarer den rigtigt.
Der er også lavet en simulering af multiplier_tester programmet, inden det er programmeret net til DE-II boardet:

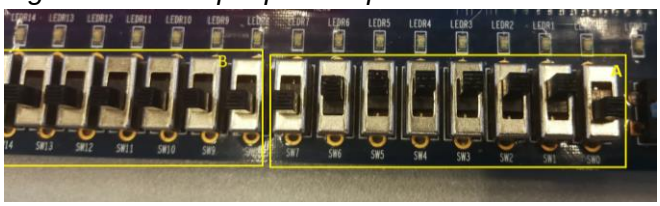


*Figur 27: Eksempel på simulering af multiplikatoren med 3*3 (00000110*00000110)som input. Output er i simuleringen konverteret til decimal.*

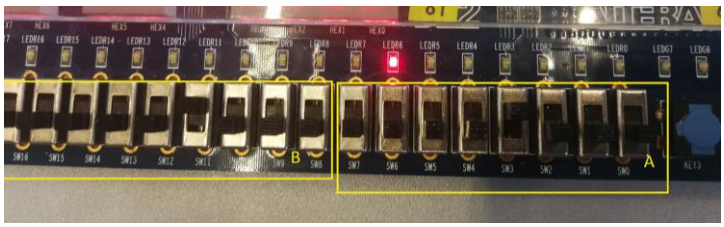
Programmet benytter SW7-SW0 til at definere A og SW17-SW8 til at definere B. Produktet vises binært på de røde LED'er.



Figur 28: Eksempel på multiplikationstest



*Figur 29: 0*A=0*



*Figur 30: 8*8=64*

Som det ses i compilation report herunder, benyttes der 103 logiske enheder til at udføre operationen. Dette skyldes at nogle af optimeringerne, der kunne have flyttet beregningerne hen til DSP. Havde dette været tilfældet, ville tallet have været langt mindre.

Flow Summary	
Flow Status	Successful - Sun Mar 01 18:50:04 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	mult
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	103
Total combinational functions	103
Dedicated logic registers	0
Total registers	0
Total pins	32
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Figur 31: Compilation Report for mult

Herunder er multiplikatoren ændret til at kunne regne på to tal á 32 bit, og dermed få et produkt på 64 bit. Dermed burde forbruget af logiske enheder stige.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity mult is
6   port (
7       A, B : in std_logic_vector(31 downto 0);
8       Prod: out std_logic_vector(63 downto 0)
9   );
10  end;
11
12  architecture code of mult is
13  begin
14      Prod <= std_logic_vector(unsigned(A)*unsigned(B));
15  end code;
```

Figur 32: Kode, med ændrede størrelser på A, B og Prod.

Og det ses også at forbruget at Logiske Enheder stiger til 1430:

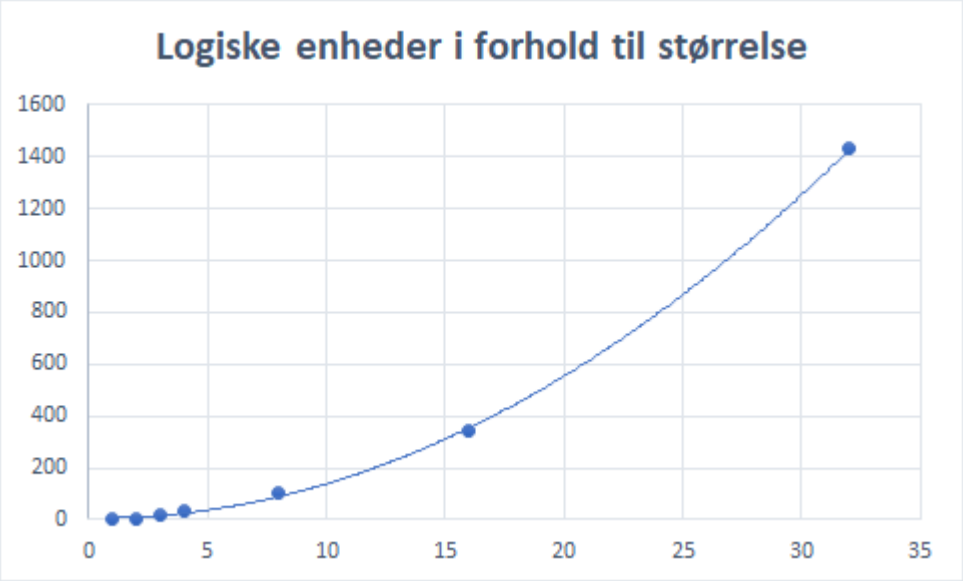
Flow Summary	
Flow Status	Successful - Sun Mar 01 18:49:05 2020
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	shift_div
Top-level Entity Name	mult
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,430
Total combinational functions	1,430
Dedicated logic registers	0
Total registers	0
Total pins	128
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Figur 33: Compilation report for multiplier med 32 bit

Her ses det at der benyttes 80 logiske elementer til at regne på 32 bit tal. I skemaet herunder ses lignende resultater, på forskellige størrelser beregninger:

Bit-størrelse:	32	16	8	4	3	2	1
Logiske Enheder	1430	343	103	31	17	4	1

Figur 34: Logiske enheder i forhold til størrelsen på gangestykket



Figur 35: Antal logiske enheder (y aksen) i forhold til størrelse på gangestykket (x aksen)

Det ses her at mængden af logiske enheder stiger eksponentielt i forbindelse med bit-størrelsen. Dette skyldes algoritmen bag binær multiplikation, hvor selve handlingen at

sammenligne to bits (gange, eller “and’e”) er meget simpel. Der opstår dog hurtigt en meget stor mængde data, der skal summeres til sidst.

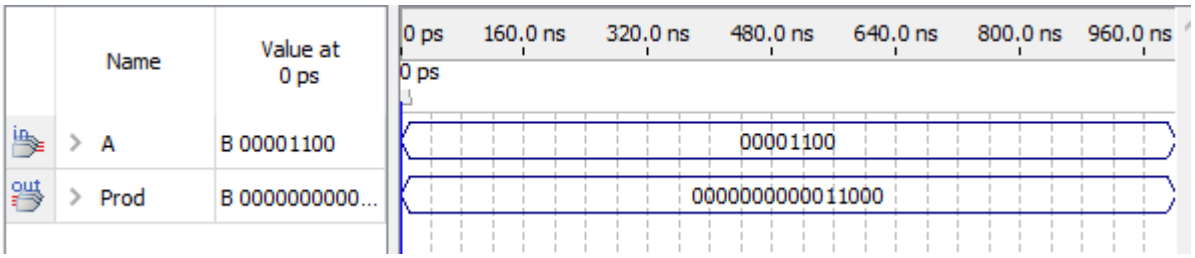
Multiplikation med konstant

Koden kan ændres. så den ganger A med en konstant, i stedet for B. Her er A ændret tilbage på en størrelse på 8 bit.

```
13  begin
14      Prod <= std_logic_vector(unsigned(A)*2);
15  end code;
```

Figur 36: Linie 14 i koden er ændret

Her ses et eksempel på en simulering med den nye kode. Det ses at “1100” (12) bliver til “11000” (24). Det bemærkes her at tallet blot er “bit-shiftet” til venstre.



Figur 37: Simulering af A ganget med en konstant (2)

Forbruget af logiske elementer varierer afhængig af konstanten der ganges med. Herunder ses sammenhængen for nogle udvalgte konstanter:

Konstant (B)	2	3	4	7	8	10
Logiske elementer	0	9	0	18	0	9

Figur 38: Forbrug af logiske elementer i forhold til konstant ganget på A

Her ses det at der ikke benyttes logiske elementer, i de tilfælde hvor der ganges med 2, 4 8 (2^x), der man her kan nøjes med at bitshifteinputtet. I tilfælde 3 og 10 er der heller ikke et stort forbrug af logiske elementer, sandsynligvis fordi den først bitshifter, og derefter adderer med A.

Konklusion

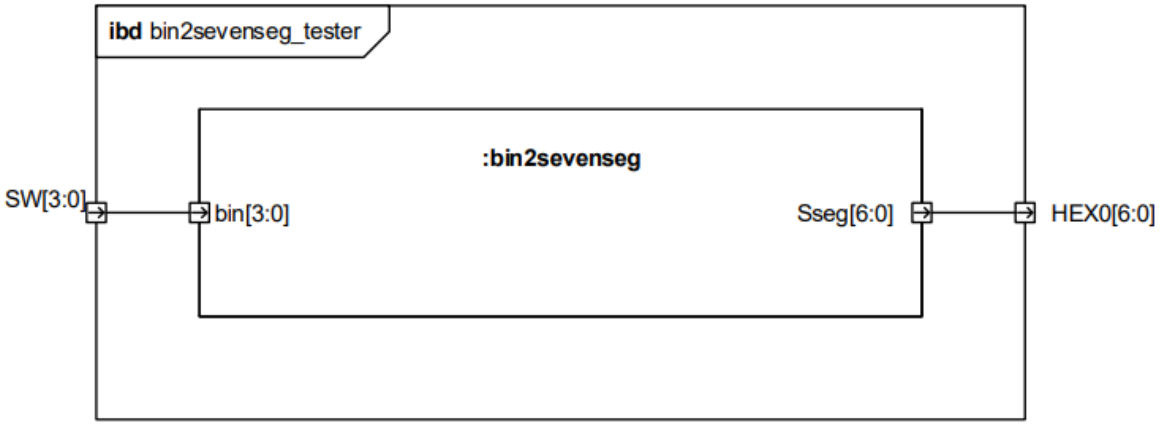
Der er gennem arbejde opnået større forståelse for multiplikation i VHDL, og forbrug af Logiske enheder. Med Multiplikationsprogrammer er det muligt at udføre simple gangestykker på DE-II board. Samtidigt er der givet indblik i aritmetikken der ligger bag koden.

Exercise 4

4.1 Binary to 7-Segment Decoder Using Selected Signal Assignment

Introduktion

I det følgende afsnit udforskes muligheden for styre 7-segment enhederne på DE2-boardet. Dette gøres ved at programmere en binær til 7-segment converter ud fra følgende diagram:



Figur 39: ibd for 7 for entity og tester, med 4 binære input, og 7 output til 7-segment. Taget fra opgavebeskrivelsen.

Design og implementering

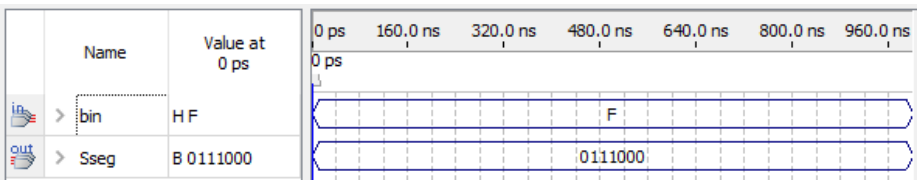
I dette afsnit beskrives koden for vores 7-segment display

```
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;
10
11  architecture code of bin2sevenseg is
12  begin
13      with bin select
14      Sseg<="0000001" when "0000", -- 0
15            "1001111" when "0001", -- 1
16            "0010010" when "0010", -- 2
17            "0000110" when "0011", -- 3
18            "1001100" when "0100", -- 4
19            "0100100" when "0101", -- 5
20            "0100000" when "0110", -- 6
21            "0001111" when "0111", -- 7
22            "0000000" when "1000", -- 8
23            "0000100" when "1001", -- 9
24            "0000010" when "1010", -- a
25            "1100000" when "1011", -- b
26            "0110001" when "1100", -- c
27            "1000010" when "1101", -- d
28            "0110000" when "1110", -- e
29            "0111000" when "1111", -- f
30            "0111100" when others;
31  end code;
```

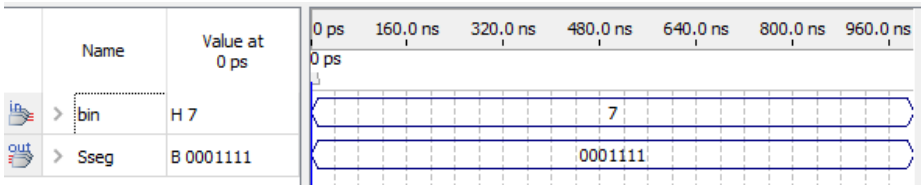
Figur 40: Kode til vores 7-segment display

I koden ovenover kaldes de nødvendige libraries.
defineres først en entity ved navn “bin2sevenseg” med portene “bin” som er vores input array med 4 pladser, og “Sseg” som er vores output array med 7 pladser.
Herefter beskrives vores arkitektur som “code” af vores tidligere beskrevet entity “bin2sevenseg”.
Herinde vælges et output til Sseg, når bin er en bestemt bitværdi. Det kan læses som sætningen: “with ‘bin’, select ‘output’ for ‘Sseg’ when ‘bin’ is ‘input’.

Koden er først simuleret for at teste hvorvidt det overhovedet fungerer. Herunder ses to eksempler med hhv. “F” og “7” som input (Her vist som Hex-decima for overblik, men selvfølgelig som binær i virkeligheden).

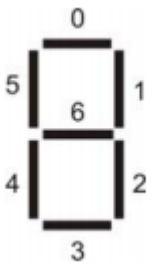


Figur 41: Simulation af 7-segment display ‘F’



Figur 42: Simulation af 7-segment display '7'

Resultatet virker her fornuftigt, men er desværre spejlvendt, idet rækkefølgen for segmenterne i 7-segment-displayet, er programmeret modsat illustrationen i manualen.



Figur 43: 7-segment-display i DE2 manual.

En hurtig løsning på vores "forkerte" rækkefølge af segmenterne i koden, er simpelthen at vende rækkefølgen om i tester programmet. Her ses det at det sædvanlige "6 downto 0" er erstattet med "0 to 6" på linje 8 i figur 44.

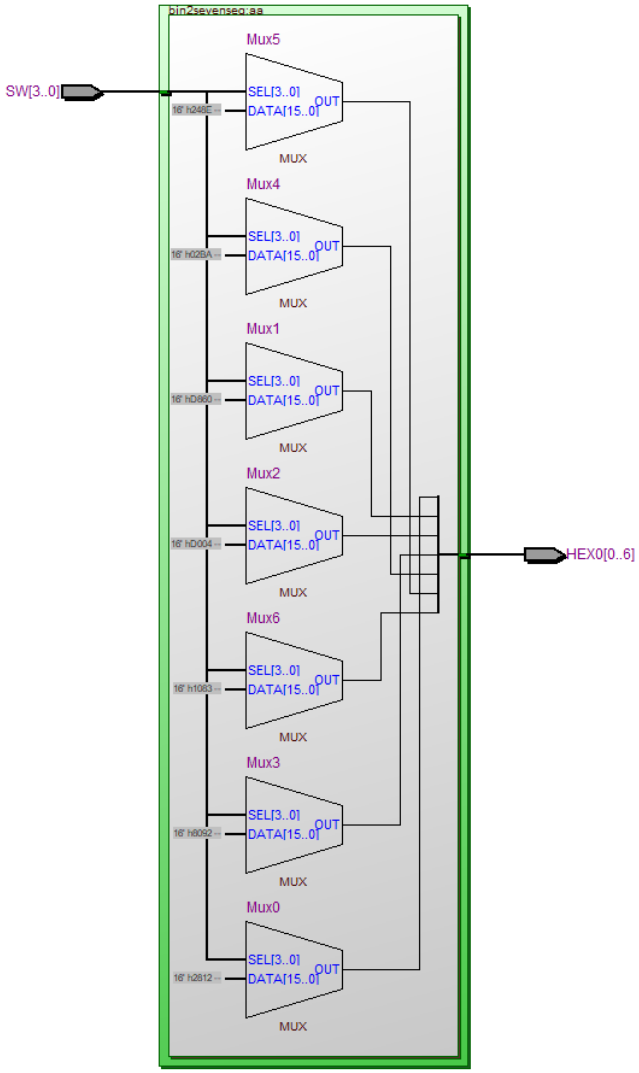
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity bin2sevenseg_tester is
6   port(
7      SW : in std_logic_vector(3 downto 0);
8      HEX0 : out std_logic_vector(0 to 6)
9      );
10 end bin2sevenseg_tester;
11
12 architecture tester of bin2sevenseg_tester is
13 begin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

Figur 44: Kode til 7-segment display tester

I testprogrammet er der allokeret 4 switches via et array med 4 pladser og en 7-segment display port, via et array med 7 pladser
I arkitekturen af testeren er aa, som kalder vores entity 'bin2sevenseg' fortæller vi hvordan 'bin's porte skal mappes i forhold til de switches og 7-segment display, som vi har allokeret.

Resultater og Diskussion

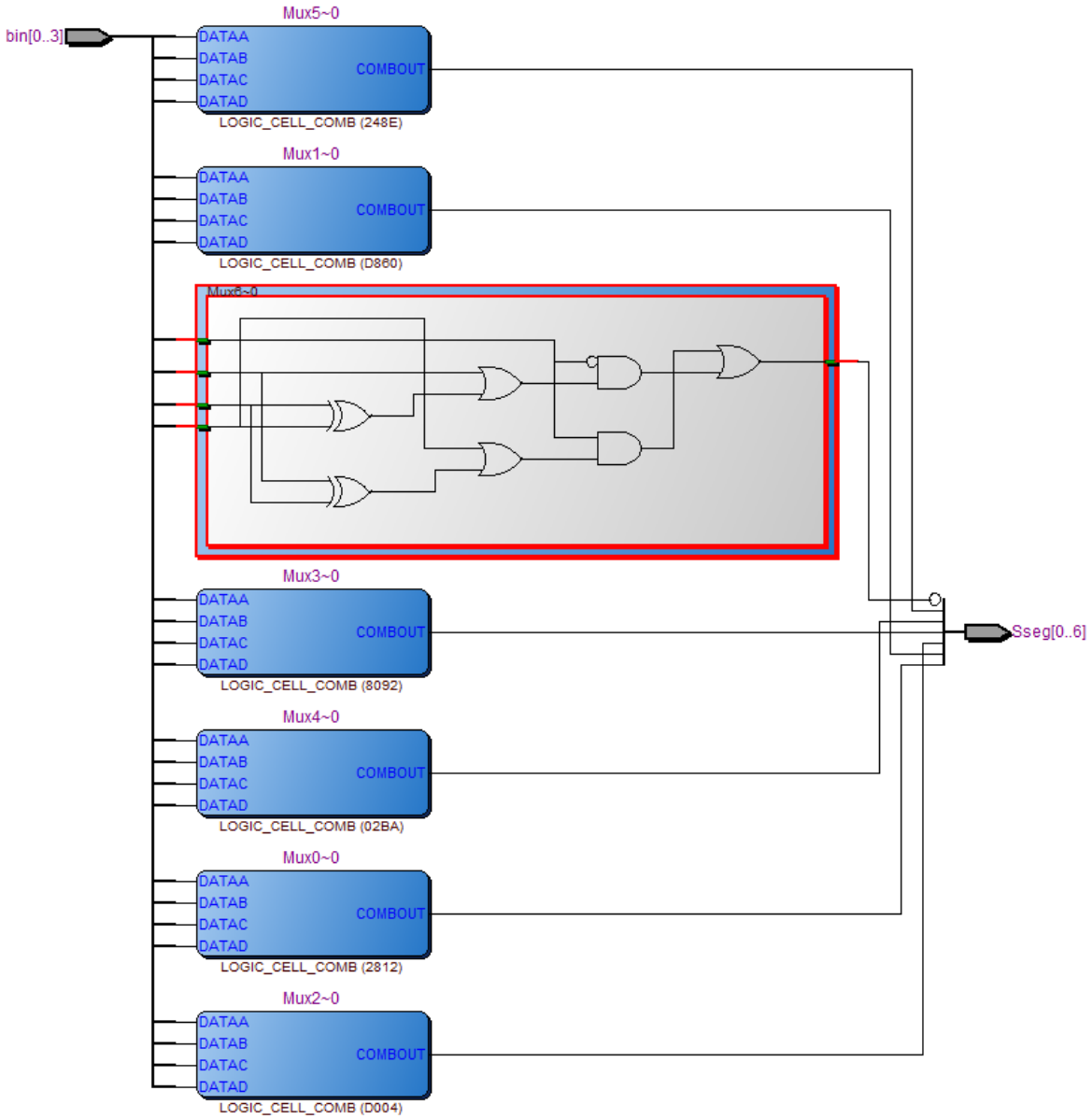
Efter kompilering består koden af en række multiplexere, som det ses i RTL-View herunder. I den efterfølgende netliste, bliver det muligt at se de anvendte gates i projektet.



Figur 45: RTL-view for :bin2sevenseg

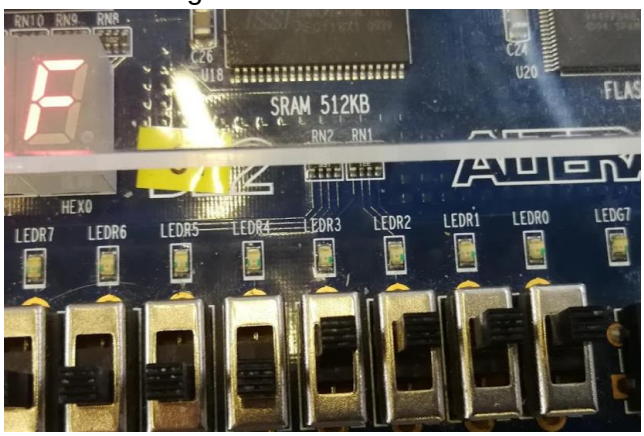
Herover ses det at programmet er bygget op af en række multiplexere, med input fra Switches, og output til 7-segment-display

Herunder ses det at det at konverteringen fra input til output udføres med or- and- not- og xor-gates.



Figur 46: Netlist: Technologymap for bin2sevenseg

Programmet er også testet på DE2-board, hvor det ses at, at 7-segment-display'et fungerer som forventeligt.



Figur 47: Alle fire kontakter høj (1111)=F

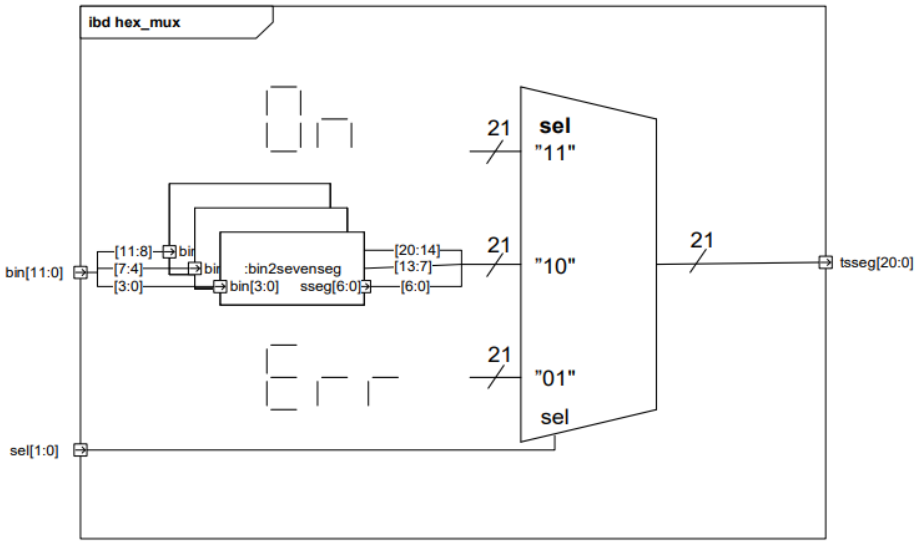
Figur 48: To kontakter til højre høj (0011)=3

Konklusion

Der er skrevet kode til bin2sevenseg, med tilhørende testprogram, der kunne konvertere input fra kontakter, gennem multiplexere, til 7-segement-display. Efter vi tog højde for nummereringen af segmenterne, fungerer programmet tilfredsstillende.

4.2 The Conditional Signal Assignment (“WHEN-ELSE”)

Introduktion



Figur 49: ibd for vores mux, fra opgavebeskrivelsen

I denne del af rapporten har vi benyttet when else, til at skifte imellem forskellige tilstande, ved brug af knapperne, og det tidligere anvendte system fra opgave 4.1.
Der er i alt 3 tilstande: En afventende tilstand hvori der kan læses “on” på vores 7-segment display, en fejltilstand, når der trykkes på den forkerte knap, hvor 7-segment displayet viser “Err”, og en operationstilstand, hvori vores system opererer, som det gjorde i opgave 4.1.

Design og implementering

Herunder ses koden til hex_mux, hvor linje 16 til 24 bliver brugt til at konvertere de binære input fra switches til 7-segment. Dette resultat gemmes i vektoren “temp”.
I linje 27 til 29 besluttes der hvad der skal sendes videre, ud fra værdien af sel.
Er værdien “10” vises det konverterede switch-input “temp”
Herudover er der 3 andre muligheder, hhv. teksten “On” “Err” eller “8888”. Den sidste fungerer som en *else* mulighed, vi ikke forventer vil ske, men er alligevel medtaget, for at undgå latches.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.all;
4
5  entity hex_mux is
6   port(
7     bin : 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 code of hex_mux is
14 signal temp : std_logic_vector(20 downto 0);
15 begin
16    aa: entity bin2sevenseg port map(
17       bin => bin(11 downto 8),
18       Sseg => temp(20 downto 14));
19    ab: entity bin2sevenseg port map(
20       bin => bin(7 downto 4),
21       Sseg => temp(13 downto 7));
22    ac: entity bin2sevenseg port map(
23       bin => bin(3 downto 0),
24       Sseg => temp(6 downto 0))
25    ;
26
27    tsseg <= temp when sel = "10" else
28             "10000000101011111111" when sel = "11" else--On
29             "00001100101111101011" when sel = "01" else -- Err
30             "00000000000000000000"; --8888
31
32
33 end code;
```

Figur 50: Kode til hex_mux

bin2sevensegment fra forrige opgave er her opdateret for at passe til det nye program. Dette ses i linje 31, hvor rækkefølgen på bits er vendt rundt, for at passe med rækkefølgen på segmenterne:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   entity bin2sevenseg is
4   port (
5       bin : in std_logic_vector(3 downto 0);
6       Sseg: out std_logic_vector(6 downto 0)
7   );
8   end;
9
10  architecture code of bin2sevenseg is
11  signal temp : std_logic_vector(0 to 6);
12  begin
13      with bin select
14      temp<="0000001" when "0000", -- 0
15            "1001111" when "0001", -- 1
16            "0010010" when "0010", -- 2
17            "0000110" when "0011", -- 3
18            "1001100" when "0100", -- 4
19            "0100100" when "0101", -- 5
20            "0100000" when "0110", -- 6
21            "0001111" when "0111", -- 7
22            "0000000" when "1000", -- 8
23            "0000100" when "1001", -- 9
24            "0000010" when "1010", -- a
25            "1100000" when "1011", -- b
26            "0110001" when "1100", -- c
27            "1000010" when "1101", -- d
28            "0110000" when "1110", -- e
29            "0111000" when "1111", -- f
30            "0111100" when others;
31        Sseg <= temp(6)&temp(5)&temp(4)&temp(3)&temp(2)&temp(1)&temp(0);
32  end code;
```

Figur 51: Opdateret bin2sevenseg

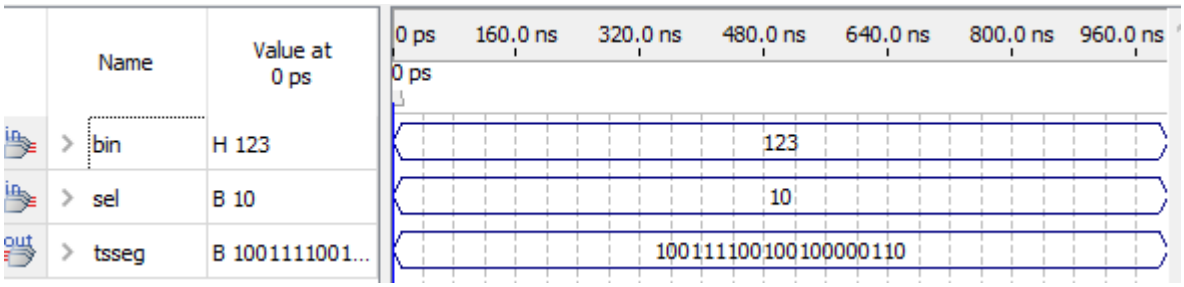
Herunder ses koden til testprogrammet, hvor 21 output fra hex_mux (gennem temp) er tildelt de tre 7-segement-display.

```
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       HEX0, HEX1, HEX2 : out std_logic_vector(6 downto 0);
9       KEY : in std_logic_vector(1 downto 0)
10      );
11  end hex_mux_tester;
12
13  architecture tester of hex_mux_tester is
14  signal temp : std_logic_vector(20 downto 0);
15  begin
16      aa: entity hex_mux port map(
17          bin => SW(11 downto 0),
18          tsseg => temp(20 downto 0),
19          sel => KEY(1 downto 0)
20      );
21
22      HEX2 <= temp(20 downto 14);
23      HEX1 <= temp(13 downto 7);
24      HEX0 <= temp(6 downto 0);
25  end tester;
```

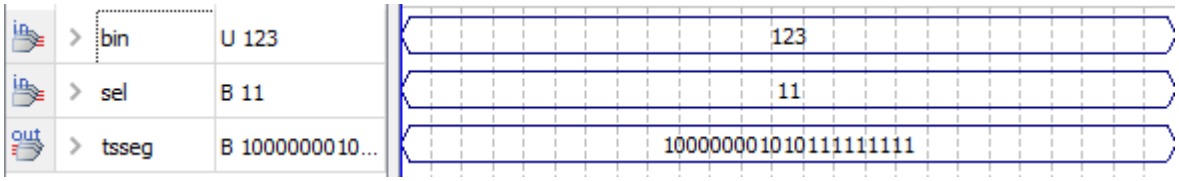
Figur 52: hex_mux_tester

Inferred latches forekommer når man ikke har en if uden en matchende else. Så hvis vi havde skrevet 'when', og ikke afsluttet med en else, uden conditions, så havde vi skabt en latch. Dette har ikke været tilfældet i vores program, da der er inkluderet else-statement

Resultater og Diskussion

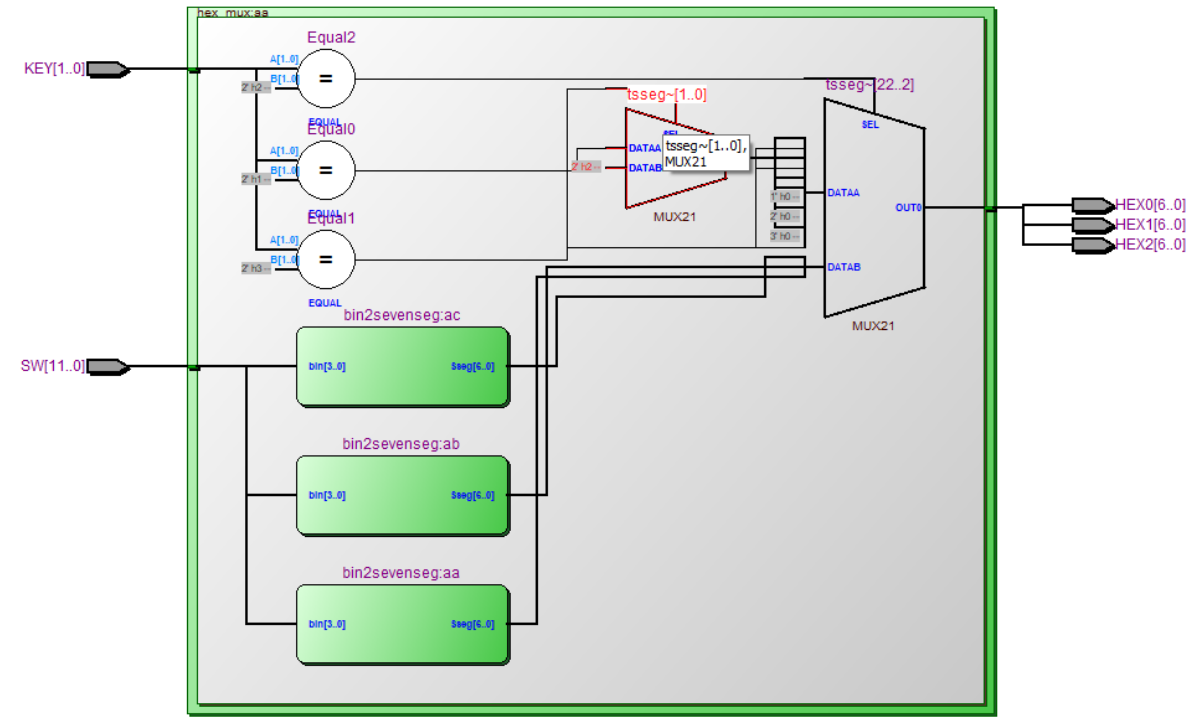


Figur 53: sel "10" tsseg: 7-segment bin



Figur 54: sel "11" tsseg: "On"

Ovenover er 'bin' beskrevet, værende switch-inputtet, select som mode, og tsseg er outputtet på 7-segment displayet.
Så det første billede er en hex-kode baseret på inputtet, og det andet billede viser "On" som output.



Figur 55: RTL view for hex_mux

På RTL-view viser et sæt af keys der afgør valg i to multiplexere. I multiplexeren til venstre vælges mellem On og Err, i muliplexeren til højre, vælges der mellem resultatet af de forrige to, og så bin2sevenseg input.

Figur 56: Display viser Err, ved tryk på KEY1

Figur 57: Display viser On, uden tryk

Figur 58: Display viser Switchværdi, ved tryk på KEY0.

Konklusion

Vi kan konkludere at mux er en enhed der kan bruges til at bestemme hvilket stadie et system skal befinde sig i, baseret på muxens input.
I muxen er der to slags input, som vi kan kalde A og B. Den modtager flere slags B input, men kun 1 A input. Dens A input bestemmer hvilket B input der skal sendes videre.

4.3 Table Lookup

Introduktion

Her benyttes “with select” til at gennemsøge en sandhedstabel.

a	b	c	x
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	-
1	1	0	-
1	1	1	1

Figur 59: Sandhedstabel for denne opgave, taget fra opgavebeskrivelsen

Design og implementering

Herunder ses koden for lookuptable, med et array defineret i linje 15, med indhold fra sandhedstabellen herover.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity lookuptable is
6   port (
7       Ind : in std_logic_vector(2 downto 0);
8       Ud: out std_logic
9   );
10  end;
11
12
13  architecture code of lookuptable is
14
15      type truths is array (0 to 7) of std_logic;
16      constant truth : truths := (
17         '1','1','0','1','0','-','-','1');
18
19  begin
20
21      Ud <=truth(to_integer(unsigned(Ind)));
22
23  end code;
24
```

Figur 60: Kode for look-up table

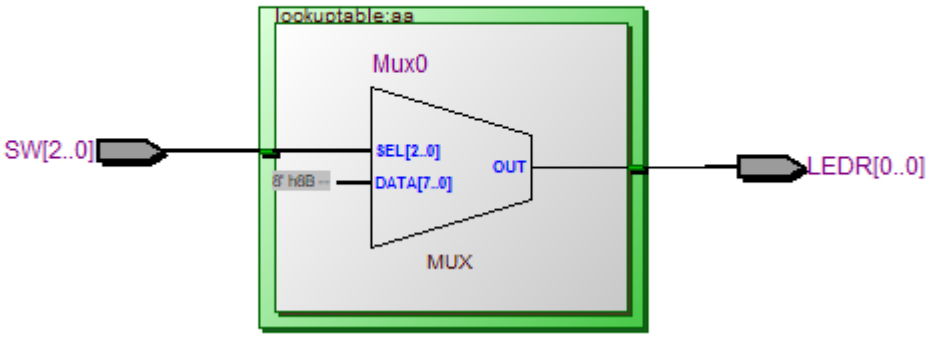
Herefter laver vi en tester, hvor vi reserverer 3 switches til input, og 1 LED til output. Vi bliver nødt til at kalde denne som et array, idet LEDR er et array af LED'er.
Derefter kalder vi look-up table, som defineret ovenover, og sætter vores Ind til de to switches, som vi lige har defineret, og ud værende enten tændt eller slukket, baseret på værdien af Ind.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use work.all;
4
5   entity lookuptable_tester is
6    port(
7       SW : in std_logic_vector(2 downto 0);
8       LEDR : out std_logic_vector(0 downto 0)
9       );
10  end lookuptable_tester;
11
12  architecture tester of lookuptable_tester is
13  begin
14     aa: entity lookuptable port map(
15        Ind => SW(2 downto 0),
16        Ud => LEDR(0)
17     );
18  end tester;
```

Figur 61: Kode til look-up table tester

Resultater og Diskussion

I RTL-view herunder er det muligt at se hvordan lookuptable fungerer som en multiplexer, med output til LEDR, baseret på SW og sandhedstabellen.



Figur 62: RTL-view for testprogrammet
Herunder ses simuleringerne for vores program. Her ses det at outputtet (Ud) stemmer overens med sandhedstabellen. Det bemærkes at resultatet ved de to "dont care" er 0.

in	Ind	B 000
out	Ud	B 1

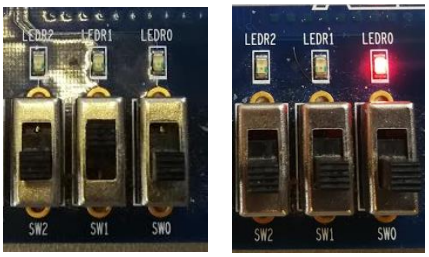
Figur 63: Input: 0 Output 1

in	Ind	B 010
out	Ud	B 0

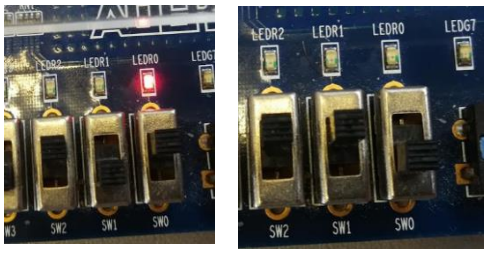
Figur 64: Input: 2: Output: 0

Figur 65: Input: 5 (don't care) Output: 0

Programmet er også testet på DE2-board, hvor programmet også fungerer tilfredsstillende. Her ses bemærkes det dog at de to dont care tilfælde giver to forskellige resultater, i modsætning til simuleringen.



Figur 66: Test på DE2-board, hhv. "010" = 0 og "000" =1



Figur 67: De to dont care, med to forskellige resultater.

Konklusion

Hvor en with-select mulighed også kunne have virket, er et array med konstante pladser hurtigere at skrive, når vi blot skal lave en sandhedstabel. Dermed minimerer vi mængden af linjer vi skal definere, men i stedet skal vi lave noget typeskift.