Ingeniørhøjskolen Aarhus

Digital system design

Journal 1

Hold 50

Cecilie Moriat *201405949*
Alexander Bennedsen *201310498*
Lasse Stenhøj *201407500*

15. marts 2015

Indholdsfortegnelse

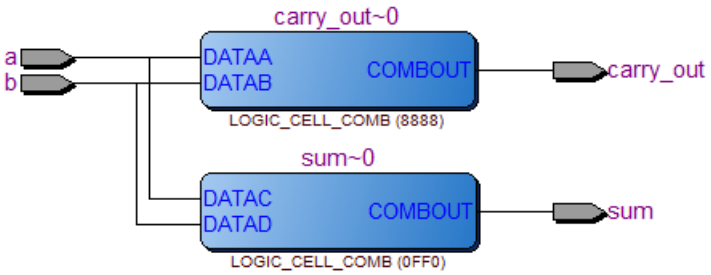
Øvelse 1

Øvelse 2

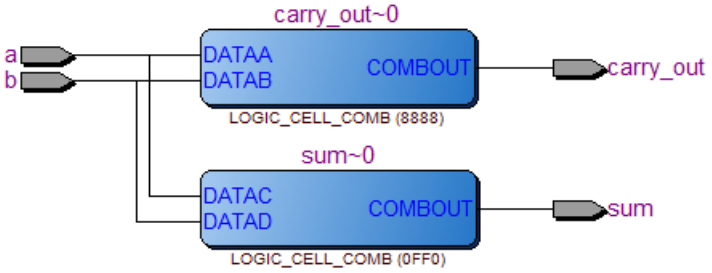
2.1 Opgave 1 - Half-adder

1) Architecture body i VHDL kan skrives på tre forskellige måder; dataflow, behavioral og structural.
   Half-adder beskrives i Dataflow style ved hjælp af direkte implementering af logiske gates. Skrivemåden gør det nemt at overføre programmet direkte til hardware og de logiske gates.
   I Behavioral style opskrives half-adderen ved brug af if/else statements. Dette giver en god forståelse af selve half-adderens funktion, men det er kompliceret at overføre det til logiske gates ud fra kodens syntax.
   I structural style laves et miks af de to ovenstående, da der først defineres en funktion for hver logisk gate som det ses i dataflow style, og derefter implementeres funktionerne i halfadder entity'en. Denne style er god at bruge, hvis man skal implementere en funktion flere gange.

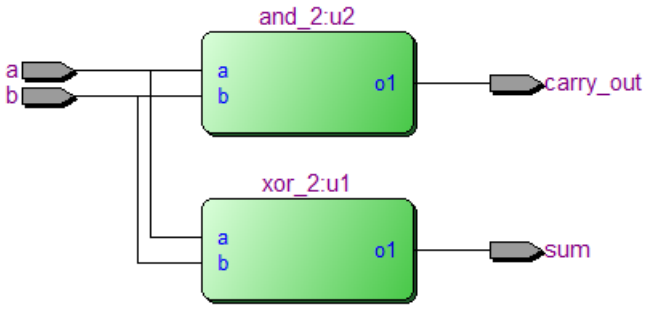
2) Et RTL view af de tre forskellige måder at skrive en half-adder på, giver følgende forskellige resultater:



Figur 2.1. Half-adder - Behavioral RTL view



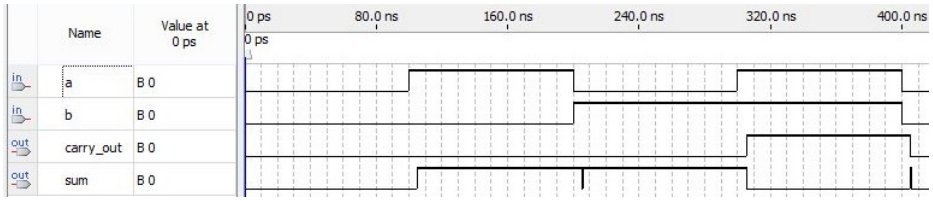
Figur 2.2. Half-adder - Dataflow RTL view



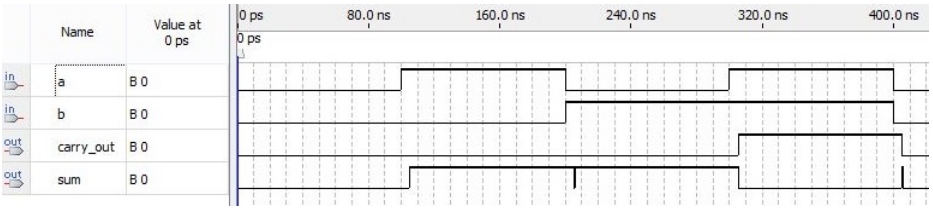
Figur 2.3. Half-adder - Structural RTL view

Ud fra ovenstående figurer ses det, at dataflow-style og behavioral-style "afkodes"på samme vis, hvor de blå bokse symboliserer en logisk funktion. I structural-style angiver de grønne bokse, også logiske funktioner, men disse er defineret af os, som hhv en AND-funktion og en XOR-funktion. Det er altså med structural-style at vi nemmest kan se, hvilke gates vi skal bruge i en fysisk opbygning af systemet.

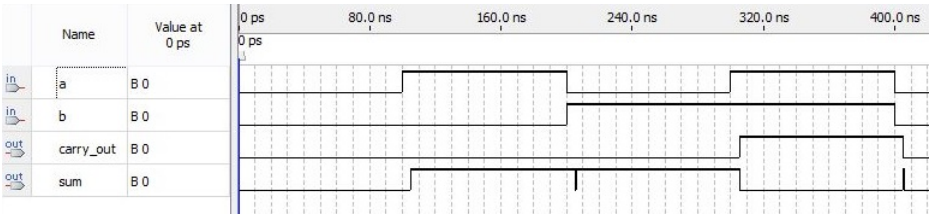
3) Følgende tre figurer viser en Timing simulation af hver style.



Figur 2.4. Half-adder - Behavioral Timing Simulation



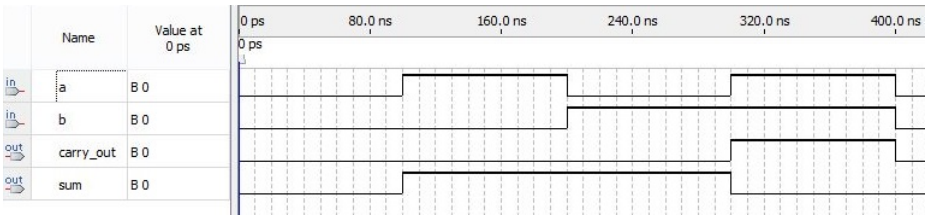
Figur 2.5. Half-adder - Dataflow Timing Simulation



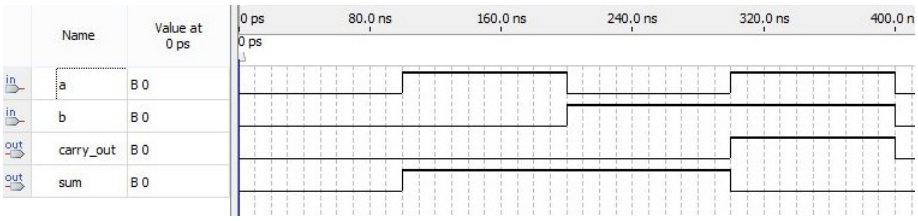
Figur 2.6. Half-adder - Structural Timing Simulation

Som det ses på figurerne, forekommer der nogle spikes på funktionerne. Dette skyldes static hazard, da de "gates"vi bruger i vores half-adder, vil have en lille tidsforskydning fra hinanden, og dermed kan give forkerte resultater i brøkdelen af et nanosekund, som det eksempelvis ses når både a-signalet og b-signalet ændrer status.

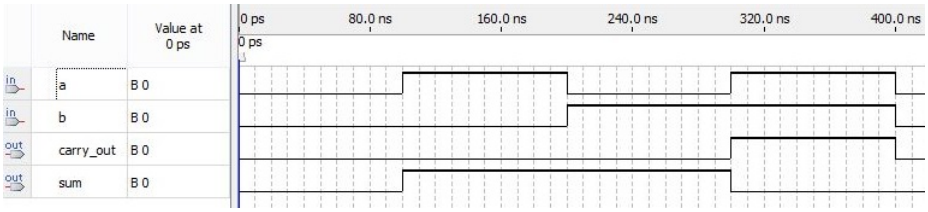
4) For at undgå disse spikes laver vi en functional simulation. Denne slags simulering tager højde for static hazard, og optimerer diagrammet til at vise et "perfekt"resultat. Figur 2.7, 2.8 og 2.9 viser denne type simulering.



Figur 2.7. Half-adder - Dataflow functional Simulation



Figur 2.8. Half-adder - Behavioral functional Simulation



Figur 2.9. Half-adder - Structural functional Simulation

2.2 Opgave 2 - Full-adder

1) Med to half-addere kan man lave en full-adder. Dette vil vi nu implementere i hhv. dataflow-style, behavioral-style og structural-style.

Kode 2.1. Full-adder Dataflow VHDL kode

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity full_adder_dataflow is
5  port (a, b, carry_in : in std_logic;
6  sum, carry_out : out std_logic);
7  end full_adder_dataflow;
8
9  architecture dataflow of full_adder_dataflow is
10
11 signal s1, s2, s3 : std_logic;
12 begin
13 s1 <= a xor b;
14 sum <= s1 xor carry_in;
15 s2 <= s1 and carry_in;
16 s3 <= a and b;
17 carry_out <= s2 or s3;
18
19 end dataflow;
```

Kode 2.2. Full-adder Behavioral VHDL kode

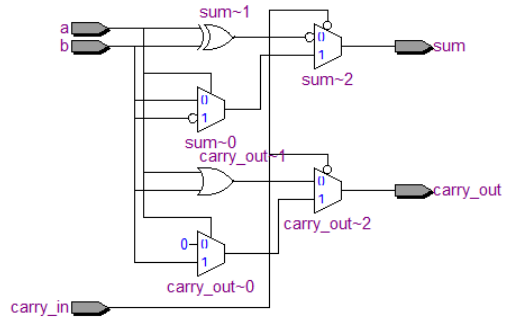
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity full_adder_behavioral is
5  port (a, b, carry_in : in std_logic;
6  sum, carry_out : out std_logic);
7  end full_adder_behavioral;
8
9  architecture behavioral of full_adder_behavioral is
10
11 signal s1, s2, s3 : std_logic;
12 begin
13 fa: process (carry_in, a, b)
14 begin
15 if carry_in = '0' then
16
17 if a = '1' then
18 sum <= not b;
19 carry_out <= b;
```

```
20    else
21    sum <= b;
22    carry_out <= '0';
23    end if;
24    else
25    sum <= a xnor b;
26    carry_out <= a or b;
27    end if;
28    end process fa;
29
30    end behavioral;
```

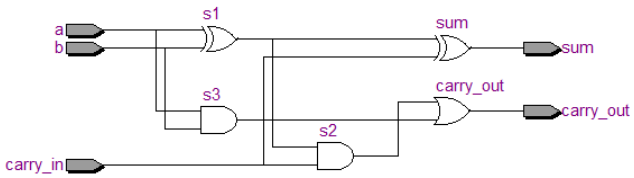
Kode 2.3. Full-adder Structural VHDL kode

```
1     library ieee;
2     use ieee.std_logic_1164.all;
3
4     entity full_adder_structural is
5     port (a, b, carry_in : in std_logic;
6     sum, carry_out : out std_logic);
7     end full_adder_structural;
8
9     architecture structural of full_adder_structural is
10
11    signal s1, s2, s3 : std_logic;
12    begin
13
14    ha1: entity work.half_adder_dataflow port map (a => a, b => b,
          sum => s1, carry_out => s3);
15    ha2: entity work.half_adder_dataflow port map (a => s1, b =>
          carry_in, sum => sum, carry_out => s2);
16    or1: entity work.or_2 port map (i1 => s2, i2 => s3, o1 =>
          carry_out);
17
18    end structural;
```

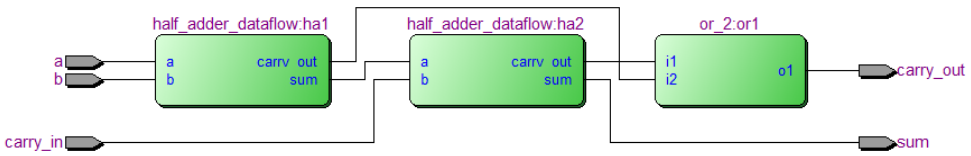
2) Med et RTL view kan vi se hvordan de tre forskellige koder vil blive omdannet til logiske gates.



Figur 2.10. Full-adder - Behavioral RTL view

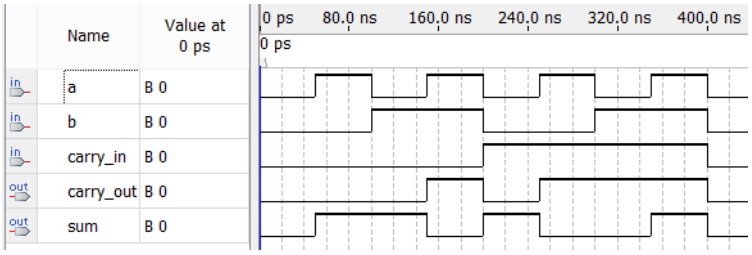


Figur 2.11. Full-adder - Dataflow RTL view

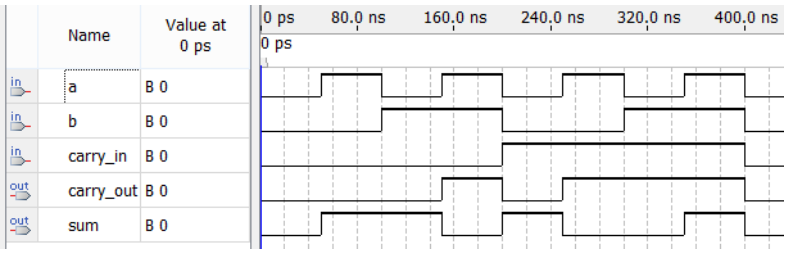


Figur 2.12. Full-adder - Structural RTL view

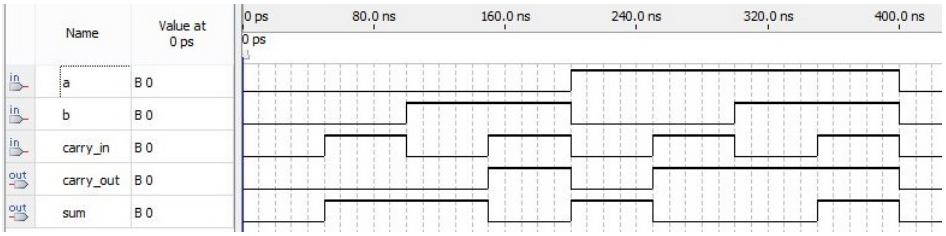
3) Til sidst laver vi en functional simulering for at se om vores tre full-adder koder opfører sig som vi ønsker.



Figur 2.13. Full-adder - Dataflow functional Simulation



Figur 2.14. Full-adder - Behavioral functional Simulation



Figur 2.15. Full-adder - Structural functional Simulation

Øvelse 3

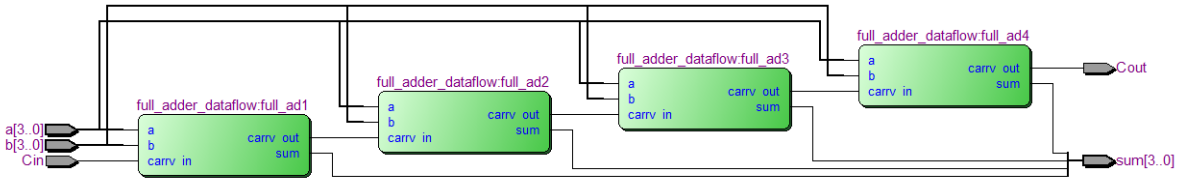
3.1 Opgave 1 - Four bit parallel adder

1) Vi designer en 4 bit parallel adder i structural style ved hjælp af den dataflow-style four bit full adder, vi har lavet i øvelse 2 (se Kode 2.1)

Kode 3.1. Four bit parallel adder Structural VHDL kode

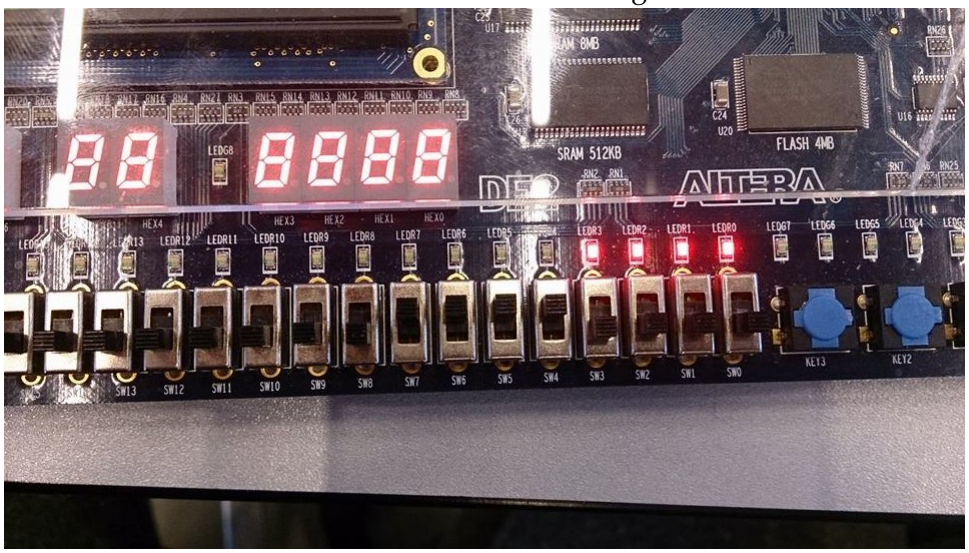
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity four_bit_full_adder is
5  port (a: in std_logic_vector (3 downto 0);
6  b: in std_logic_vector (3 downto 0);
7  Cin: in std_logic;
8  sum: out std_logic_vector (3 downto 0);
9  Cout: out std_logic);
10
11 end four_bit_full_adder;
12
13 architecture structural of four_bit_full_adder is
14
15 signal i1, i2, i3 : std_logic;
16 begin
17 full_ad1 :  entity work.full_adder_dataflow port map (a => a(0),
       b => b(0), carry_in => cin, sum => sum(0), carry_out => i1 );
18 full_ad2 :  entity work.full_adder_dataflow port map (a => a(1),
       b => b(1), carry_in => i1, sum => sum(1), carry_out => i2 );
19 full_ad3 :  entity work.full_adder_dataflow port map (a => a(2),
       b => b(2), carry_in => i2, sum => sum(2), carry_out => i3 );
20 full_ad4 :  entity work.full_adder_dataflow port map (a => a(3),
       b => b(3), carry_in => i3, sum => sum(3), carry_out => Cout );
21
22 end structural;
```

2) Vi kan ved hjælp af RTL-viewer se, om vores full adders er forbundet korrekt:



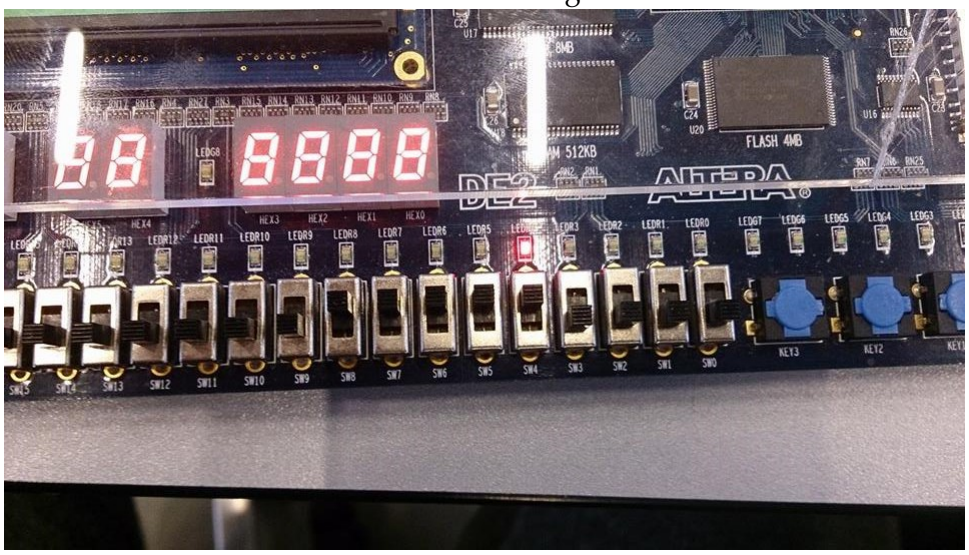
Figur 3.1. Four bit parallel adder - Structural RTL view

3) Vi starter med at sætte bits til 00001111 samt cin 0 og får det forventede resultat:



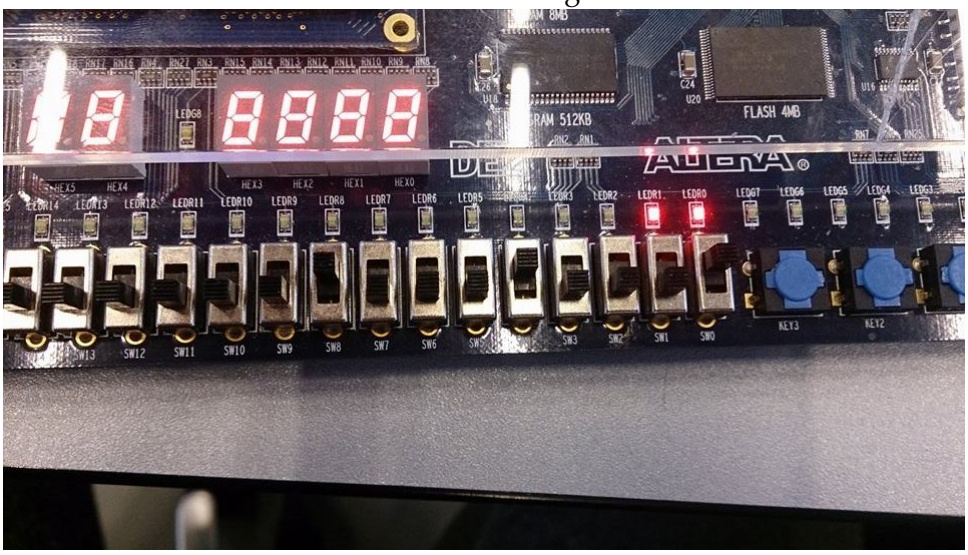
Figur 3.2. Four bit parallel adder - 00001111, cin=0

Vi sætter nu bits til 11110000 samt cin 1 og får det forventede resultat:



Figur 3.3. Four bit parallel adder - 11110000, cin=1

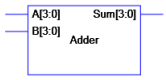
Vi sætter nu bits til 00010001 samt cin 1 og får det forventede resultat:



Figur 3.4. Four bit parallel adder - 00010001, cin=1

3.2 Opgave 2 - Four bit adder - using signed/unsigned logic

1) Vi laver en unsigned adder i dataflow style som vist på figur 3.5. Da input og output skal være af std logic vector typen, og vi skal bruge + operatoren, bliver vi nødt til at konvertere til unsigned først. Se Kode 3.2

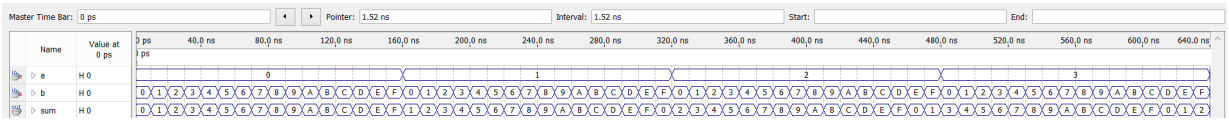


Figur 3.5. Four bit unsigned adder

Kode 3.2. Four bit unsigned adder Dataflow VHDL kode

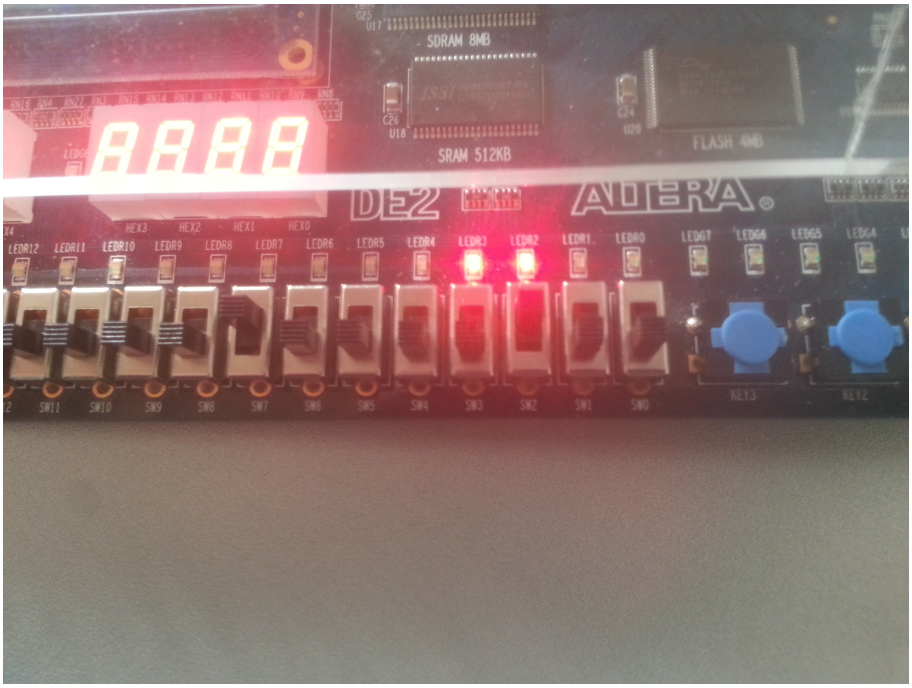
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity unsigned_adder is
6  port (a: in std_logic_vector (3 downto 0);
7  b: in std_logic_vector (3 downto 0);
8  sum: out std_logic_vector (3 downto 0));
9  end unsigned_adder;
10
11 architecture dataflow of unsigned_adder is
12 begin
13
14 sum <= std_logic_vector(unsigned(a) + unsigned(b));
15 end dataflow;
```

2) Vi tester nu vores kode med en functional simulation:



Figur 3.6. Four bit unsigned adder Functional simulation

3) Vi sætter nu bits til 1000 + 0100 og får det forventede resultat som det ses på Figur 3.7

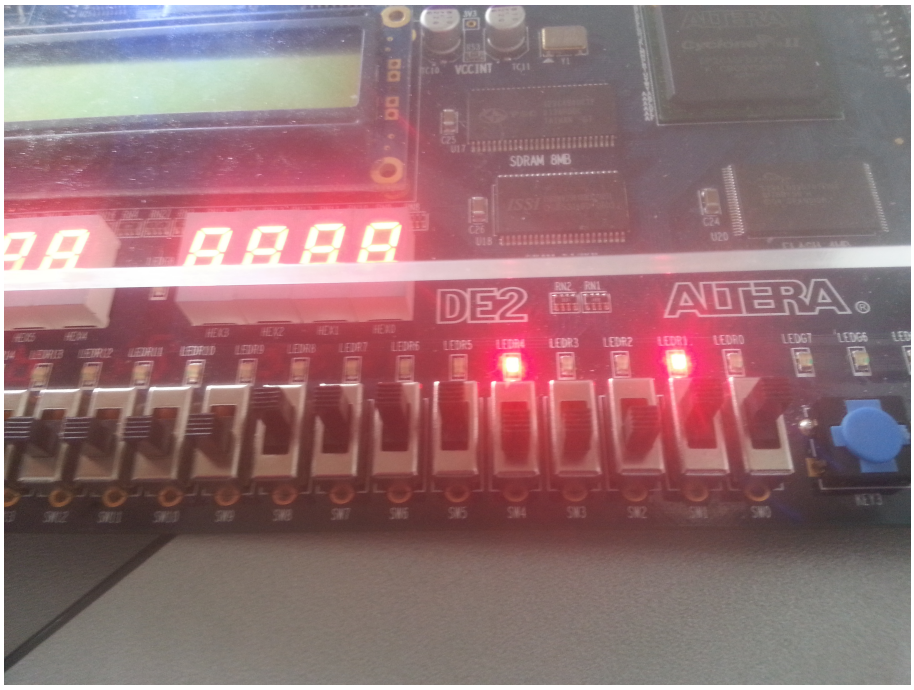


Figur 3.7. Four bit unsigned adder - 1000 + 0100

4) Vi ændrer nu koden så det bliver en signed adder som det ses i Kode 3.3. Vi tester det på vores DE2 board, og ser at der ingen forskel er på unsigned adderen og signed adderen som det ses på Figur 3.8. Dette skyldes at selve bit'sne ikke er anderledes, men det er kun måden de skal tolkes på.

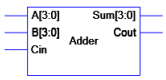
Kode 3.3. Four bit signed adder Dataflow VHDL kode

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity unsigned_adder is
6   port (a: in std_logic_vector (3 downto 0);
7   b: in std_logic_vector (3 downto 0);
8   sum: out std_logic_vector (3 downto 0));
9   end unsigned_adder;
10
11  architecture dataflow of unsigned_adder is
12  begin
13
14  sum <= std_logic_vector(unsigned(a) + unsigned(b));
15  end dataflow;
```



Figur 3.8. Four bit signed adder - 1000 + 0100

5) Vi laver nu vores unsigned adder om, så den også virker med et carry in, og leverer et carry out som vist på Figur 3.9. Vi benytter os af resize funktionen, samt laver nogle interne signaler, inden vi sender resultatet ud igen. Koden ses i Kode 3.4.



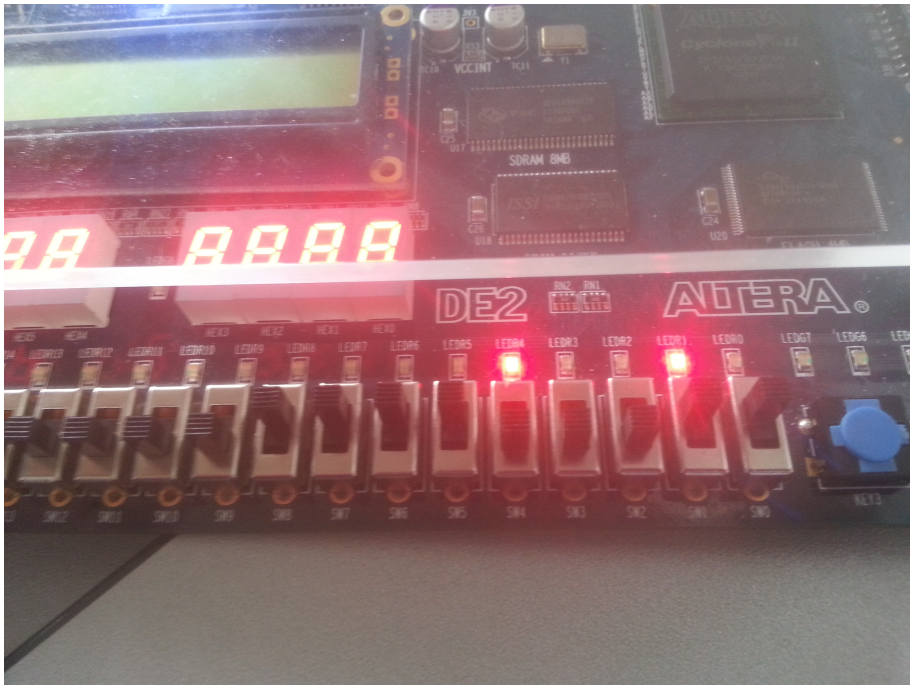
Figur 3.9. Four bit unsigned adder with carry

Kode 3.4. Four bit unsigned adder with carry Dataflow VHDL kode

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity unsigned_adder_carry is
6  port (a: in std_logic_vector (3 downto 0);
7  b: in std_logic_vector (3 downto 0);
8  carry_in: in std_logic;
9  carry_out : out std_logic_vector (0 downto 0);
10 sum: out std_logic_vector (3 downto 0));
11 end unsigned_adder_carry;
12
13 architecture dataflow of unsigned_adder_carry is
14 signal c : unsigned (3 downto 0);
15 signal s : unsigned (4 downto 0);
16 begin
17 c <= "000" & carry_in;
```

```
18    s <= resize(unsigned(a),5) + resize(unsigned(b),5) + resize(c,5)
          ;
19    sum <= std_logic_vector(s(3 downto 0));
20    carry_out <= std_logic_vector(s(4 downto 4));
21    end dataflow;
```

6) Vi overfører vores adder til DE2 boardet. Her adderer vi 1110 + 0011 samt carry in = 1, og får det forventede resultat som ses på figur 3.10.



Figur 3.10. Four bit unsigned adder with carry

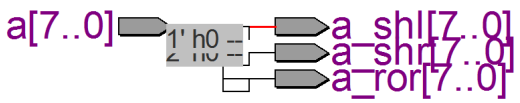
3.3 Opgave 3 - Concatenation

Kode 3.5. Concatenation kode

1)

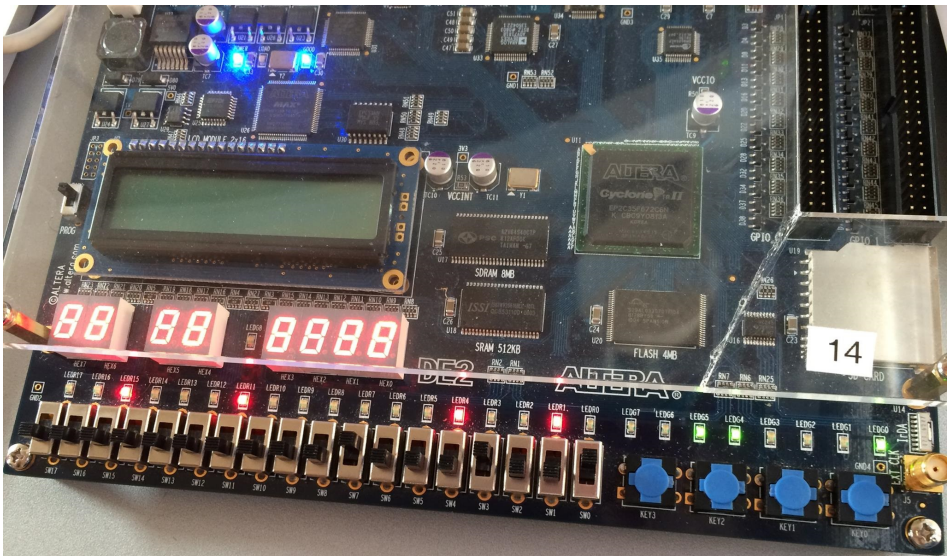
```
1  library ieee;
2  use ieee.Std_logic_1164.all;
3
4  entity shift_div is
5  port (a : in std_logic_vector(7 downto 0);
6  a_shl,a_shr,a_ror: out std_logic_vector(7 downto 0));
7  end shift_div;
8
9  architecture dataflow of shift_div is
10
11 begin
12 a_shl <= a(6 downto 0) & '0';
13
14 a_shr <= "00" & a(7 downto 2);
15
16 a_ror <= a(2 downto 0) & a(7 downto 3);
17
18 end dataflow ;
```

2) Vi kan se på figur 3.11 fra RTL-viewer, at der ikke er nogen logiske elementer i kodestykket:



Figur 3.11. Concatenation RTL

3) Vi overfører programmet til DE2-boardet. Inputtet a sættes som SW[7:0], output a-shl sættes til LEDR[7:0], a-shr sættes til LEDR[17:10] og a-ror sættes til LEDG[7:0]. Dette kan ses på figur 3.12.



Figur 3.12. Concatenation på DE2-board

3.4 Opgave 4 - Subtype

1) Vi opskriver koden for en 4-bit subtractor som ses i kode 3.6

Kode 3.6. Subtractor kode

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Subtypes is
5   port (a,b : in std_logic;
6   c  : out std_logic);
7   end Subtypes;
8
9   architecture dataflow of Subtypes is
10  subtype bool is std_logic range '1' to 'Z';
11  signal tmp : bool;
12  begin
13  tmp<= 'U';
14  c<= b and tmp;
15  end dataflow;
```

Grunden til at vi får fejlen at U er udenfor rækkevidde ses på figur 3.13 nedenfor:

Value	State	Strength
U	uninitialized	none
X	unknown	forcing
0	0	forcing
1	1	forcing
Z	none	high impedance
W	unknown	weak
L	0	weak
H	1	weak
-	don't care	none

Figur 3.13. Stater og styrker for std_ulogic værdier

Derfor retter vi til koden

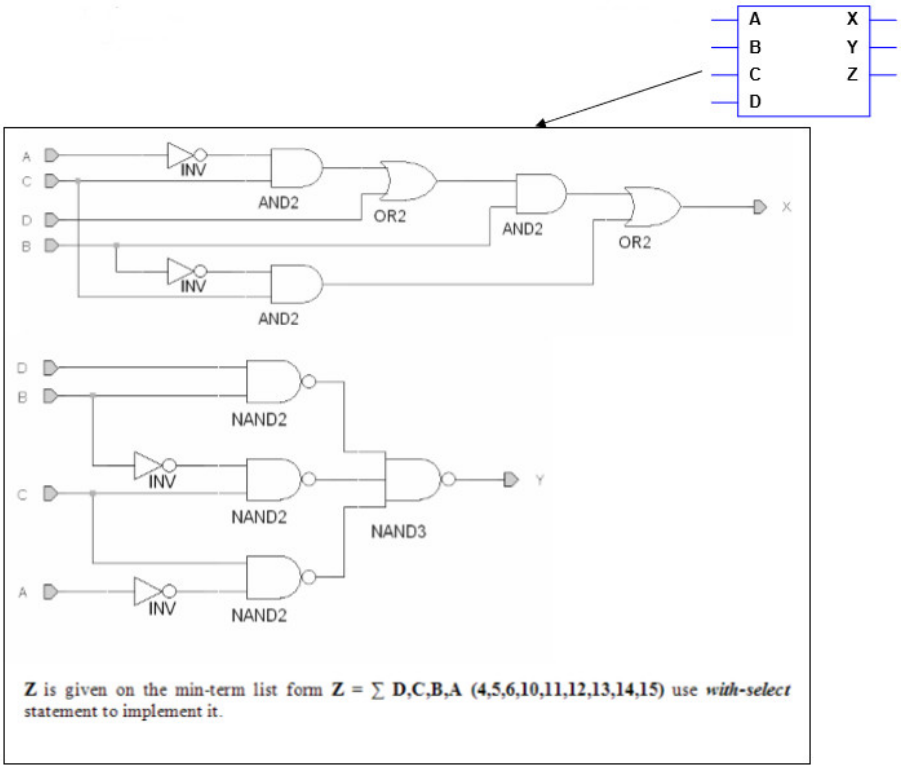
Kode 3.7. Rettet subtractor kode

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Subtypes is
5  port (a,b : in std_logic;
6  c  : out std_logic);
7  end Subtypes;
8
9  architecture dataflow of Subtypes is
10 subtype bool is std_logic range 'U' to 'Z';
11 signal tmp : bool;
12 begin
13 tmp<= 'U';
14 c<= b and tmp;
15 end dataflow;
```

Øvelse 4

4.1 Opgave 1: Combinational VHDL

2) Vi designer systemet som vist på figur 4.1, med koden 4.1.



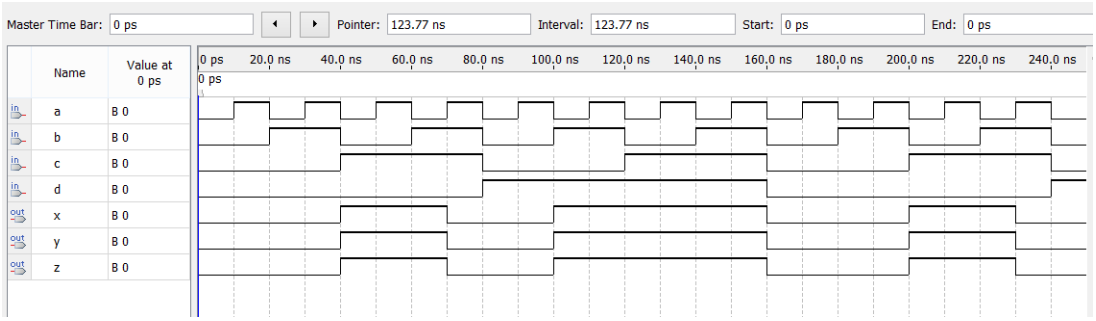
Figur 4.1.

Kode 4.1. Combinational VHDL kode

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4
5   entity combinational_VHDL is
6   port (a, b, c, d : in std_logic;
7   x, y, z : out std_logic);
8   end;
9
10  architecture dataflow of combinational_VHDL is
```

```
11      signal tmp : std_logic_vector(3 downto 0);
12      begin
13      tmp <= (d, c, b, a);
14      x <= (((((not a) and c) or d) and b) or ((not b) and c));
15      y <= not((not(d and b))and (not((not b)and c)) and (not((not
            a)and c)));
16      with tmp select
17      z <= '1' when "0100" |"0101" |"0110" |"1010" |"1011" |"1100"
            |"1101" |"1110" |"1111",
18      '0' when others;
19
20      end dataflow;
```

2) Vi laver nu en functional simulation af koden og får resultatet vist på figur 4.2.



Figur 4.2. Functional simulation

4.2 Opgave 2: Binary to Decimal Conversion

1) Vi har læst koden for BCD decoderen og er indforstået med hvad der fungerer hvordan, og hvorfor.

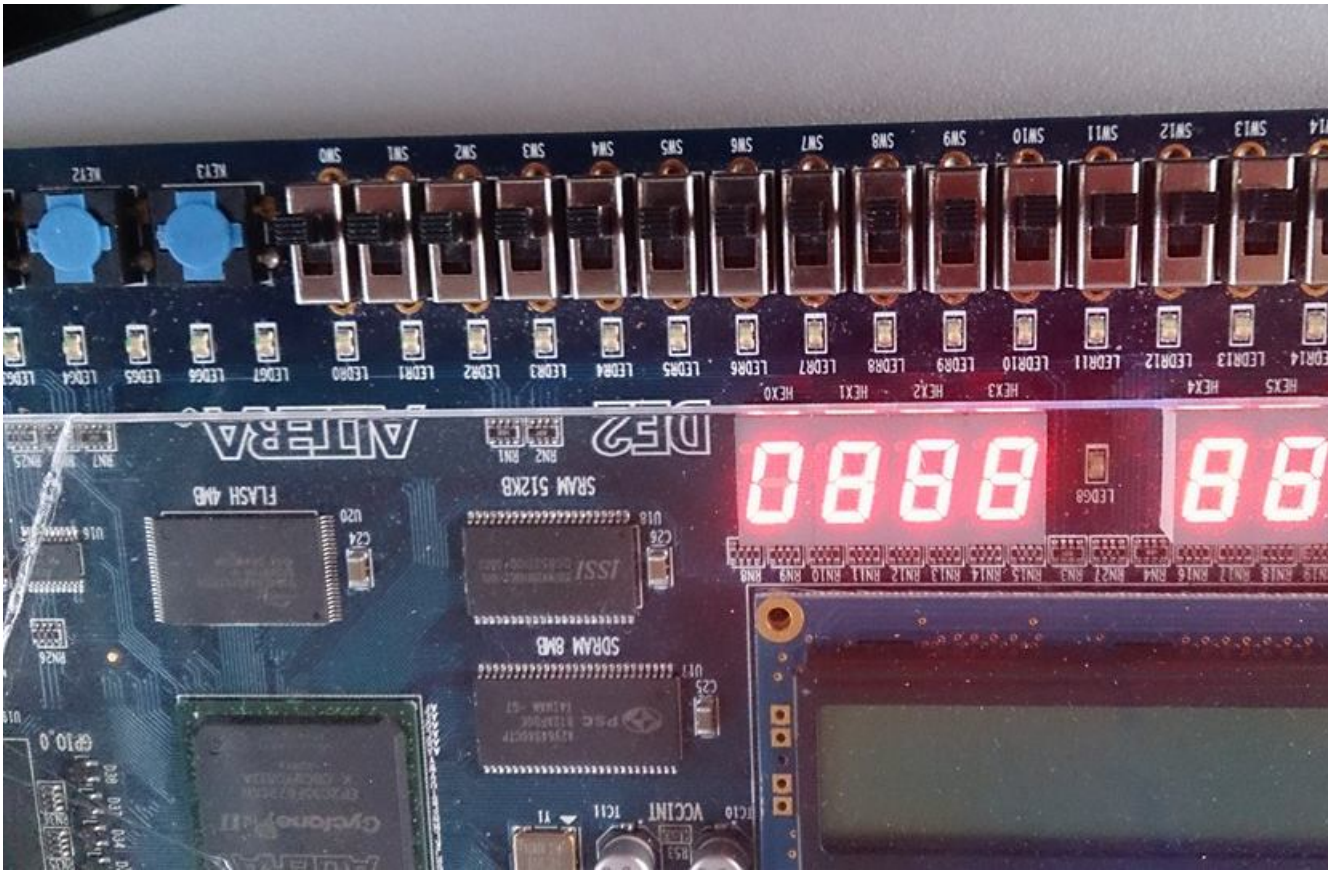
2) Vi skriver koden for en 7-segment decoder ved hjælp af with/select i VHDL:

Kode 4.2. BCD til 7 segment decoder

```
1       library ieee;
2       use ieee.std_logic_1164.all;
3
4       entity BCDdecoder is
5       port(dcba: in std_logic_vector(3 downto 0);
6       seg: out std_logic_vector(6 downto 0));
7       end BCDdecoder;
8
9       architecture selection of BCDdecoder is
10      begin
11      with dcba select
12      seg<="0000001" when "0000",
13      "1001111" when "0001",
14      "0010010" when "0010",
```

```
15          "0000110" when "0011",
16          "1001100" when "0100",
17          "0100100" when "0101",
18          "1100000" when "0110",
19          "0001111" when "0111",
20          "0000000" when "1000",
21          "0001100" when "1001",
22          "1111111" when others;
23          end selection;
```

3) Vi tester programmet på DE2-boardet. Vi bruger SW[3:0] som input og HEX0 som output. Vi tester først med 0 som input og får forventet output som ses på figur 4.3:



Figur 4.3. 7-segment decoder med 0 som input

Vi tester nu med 9 som input og får forventet output som ses på figur 4.4:

`pictures/Oevelse4/BCD_decoder/BCD_1seg_9.jpg`

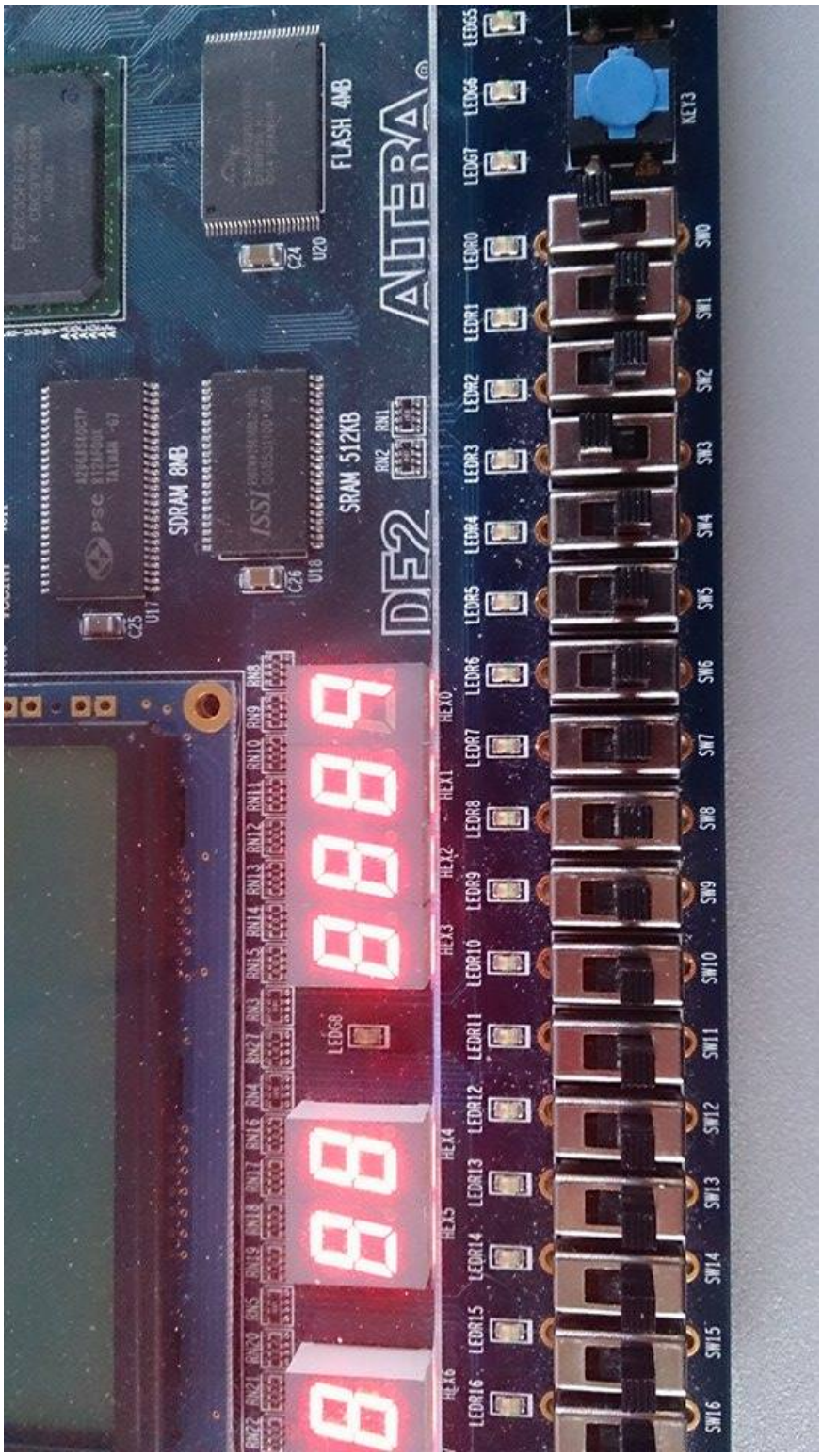
Figur 4.4. 7-segment decoder med 9 som input

Vi tester nu med ugyldigt input (over 9) og får som forventet blankt output som ses på figur 4.5:

4) Vi skriver koden for en to-segments decoder ved hjælp af when-statements, som ses i kode 4.3

Kode 4.3. BCD til to 7 segment decoder

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity TwoDigitDecoder is
6  port(V: in std_logic_vector(3 downto 0);
7  seg1, seg10: out std_logic_vector(6 downto 0));
8  end TwoDigitDecoder;
9
10 architecture selection of TwoDigitDecoder is
11 signal M, X: std_logic_vector(3 downto 0);
12 signal Z: std_logic ;
13
14 begin
15 Z <= '1' when(V > "1001") else '0';
16 M <= V when(Z = '0') else (std_logic_vector(unsigned(V) -
       10));
17 X <= "000" & Z;
18
19 seg1<="0000001" when (M = "0000") else
20 "1001111" when (M = "0001") else
21 "0010010" when (M = "0010") else
22 "0000110" when (M = "0011") else
23 "1001100" when (M = "0100") else
24 "0100100" when (M = "0101") else
25 "1100000" when (M = "0110") else
26 "0001111" when (M = "0111") else
27 "0000000" when (M = "1000") else
28 "0001100" when (M = "1001") else
29 "0000001" when (M = "1010") else
30 "1001111" when (M = "1011") else
31 "0010010" when (M = "1100") else
32 "0000110" when (M = "1101") else
33 "1001100" when (M = "1110") else
34 "0100100" when (M = "1111") else "1111111";
35
36 seg10<="0000001" when (X = "0000") else
37 "1001111" when (X = "0001") else "1111111";
38
39 end selection;
```



Figur 4.5. 7-segment decoder med ugyldigt input