E2DSD — Journal #2 "Arithmetic and Logical Operators" og "Dataflow-style Combinatorial Designs"

Gruppe #37
Adam Ryager Høj — 201803767
Rasmus Kahr — 201803491
Hoang Phuoc Pho Tran — 201507142

2020-03-17

Indhold

1 Øvelse 3 — Arithmetic and Logical Operators

I denne øvelse er en testbench brugt, hvorpå alle switches og LED'er er mappet til, se source code table 1.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE work.ALL;
4
5  ENTITY UnitUnderTest IS
6      PORT (
7          SW : IN std_logic_vector(17 DOWNTO 0);
8          LEDR : OUT std_logic_vector(17 DOWNTO 0);
9          LEDG : OUT std_logic_vector(7 DOWNTO 0));
10 END;
11
12 ARCHITECTURE structural OF UnitUnderTest IS
13 BEGIN
14 --     UUT0 : ENTITY fourbitaddersimple(CarryImpl)
15 --         PORT MAP
16 --         (
17 --             -- INPUTS
```

Tab. 1: Testbench setup

1.1 Signed and Unsigned Arithmetic

1.1.1 Introduktion

Ved at udnytte VHDLs indbyggede arithmetiske funktioner, har vi mulighed for at simplificere four-bit-adderen fra øvelse 2 (Ø2). Dette vil ske vha. *casting* fra `std_logic_vector` til `unsigned` / `signed`.
Vi har i denne opgave, skrevet og testet 3 four-bit-adders: signed, unsigned og unsigned w. carry,

1.1.2 Design og implementering

Vi brugte i Ø2 `std_logic_vector` som input og output - dette vil ske igen.
Dog bliver både signalet der behandles, midlertidigt lavet om til enten `signed` eller `unsigned`.
Dette giver mulighed for at bruge regneoperatorer på en ny måde - der kan adderes, trækkes fra, multipliceres og divideres. Den fulde entity, med de tilhørende architectures ses i source code table 2, 3 og 4, det hele er samlet i én fil.

```vhdl
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY fourbitaddersimple IS
    PORT (
        A, B : IN std_logic_vector(3 DOWNTO 0);
        Cin : IN std_logic := '0';
        A_out : OUT std_logic_vector(3 DOWNTO 0);
        B_out : OUT std_logic_vector(3 DOWNTO 0);
        SUM : OUT std_logic_vector(3 DOWNTO 0);
        Cout : OUT std_logic := '0');
END fourbitaddersimple;

-- Unsigned with Carry
ARCHITECTURE CarryImpl OF fourbitaddersimple IS

    SIGNAL ua : unsigned(4 DOWNTO 0);
    SIGNAL ub : unsigned(4 DOWNTO 0);
    SIGNAL uc : unsigned(4 DOWNTO 0);
    SIGNAL usum : unsigned(4 DOWNTO 0);

BEGIN
    A_out <= a;
    B_out <= b;
    ua <= resize(unsigned(a), 5);
    ub <= resize(unsigned(b), 5);
    uc <= (0 => Cin, OTHERS => '0');
    usum <= ua + ub + uc;
    Cout <= std_logic(usum(4));
    sum <= std_logic_vector(resize(usum, 4));
    -- sum <= std_logic_vector(usum);
END;
```

Tab. 2: *Entity dekleration, der går igen i de 3 adders og Four-bit-adder: architecture: unsigned with carry*

```
1  -- Unsigned
2  ARCHITECTURE unsigned_impl OF fourbitaddersimple IS
3
4      SIGNAL ua : unsigned(3 DOWNTO 0);
5      SIGNAL ub : unsigned(3 DOWNTO 0);
6      SIGNAL usum : unsigned(3 DOWNTO 0);
7      --
8  BEGIN
9      ua <= unsigned(a);
10     ub <= unsigned(b);
11     usum <= ua + ub;
12     sum <= std_logic_vector(usum);
13
14 END;
```

Tab. 3: *Four-bit-adder: architecture: unsigned_impl*

```
1  -- Signed
2  ARCHITECTURE signed_impl OF fourbitaddersimple IS
3
4      SIGNAL sa : signed(3 DOWNTO 0);
5      SIGNAL sb : signed(3 DOWNTO 0);
6      SIGNAL ssum : signed(3 DOWNTO 0);
7
8  BEGIN
9      sa <= signed(a);
10     sb <= signed(b);
11     ssum <= sa + sb;
12     sum <= std_logic_vector(ssum);
13 END;
```

Tab. 4: *Four-bit-adder: architecture: signed_impl*

1.1.3 Resultater

Efter fuldført syntese kan vi se på RTL diagrammerne, der viser den funktionelle opbygning af koden, disse ses i

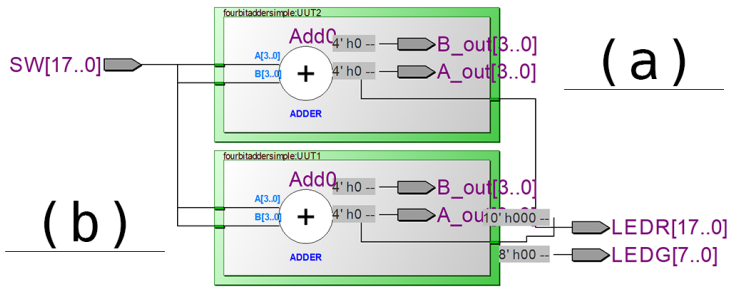

Fig. 1: *(a) signed full adder, (b) unsigned full adder*

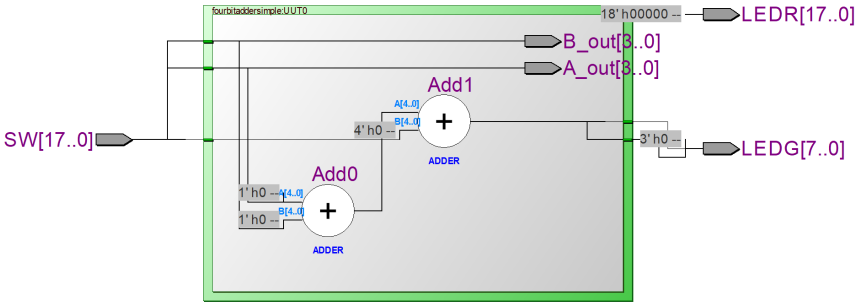

Fig. 2: *RTL view af en unsigned full adder med carry*

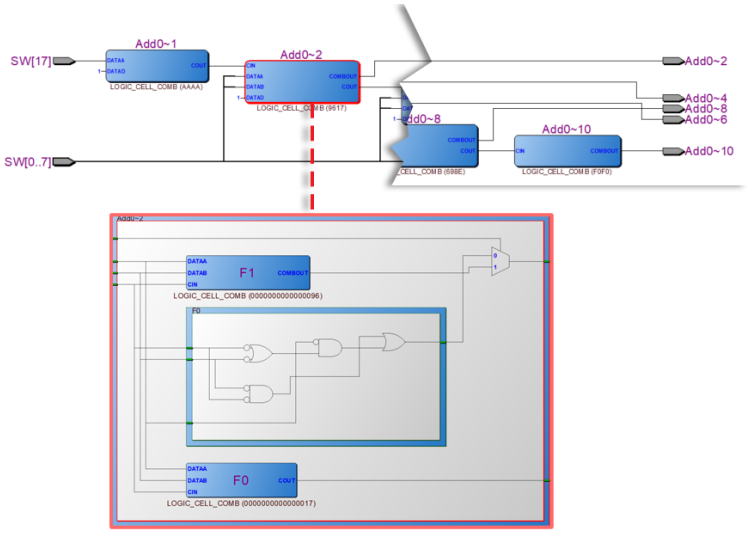

Fig. 3: *Technology map view af unsigned full adder med carry - inkl. udsnit af de interne blokkes arbejde.*

Desuden kan vi se på technology map viewet, der viser hvordan FPGA boardet mapper dets interne forbindelser, dette ses i fig. 3 og fig. 4.

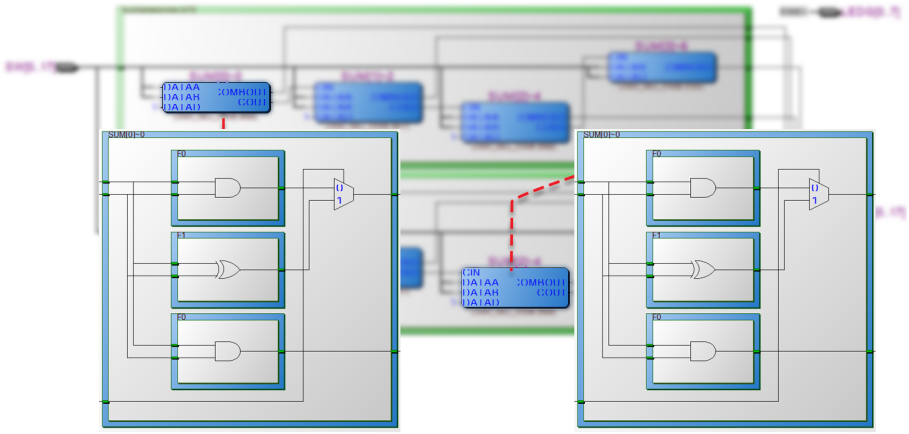

Fig. 4: *Udsnit af technology map view af en signed og unsigned adder. Der er ingen forskel i praksis.*

Ydermere kan der sættes en timing simulation op der viser den system under påvirkning, dette ses i fig. 5.

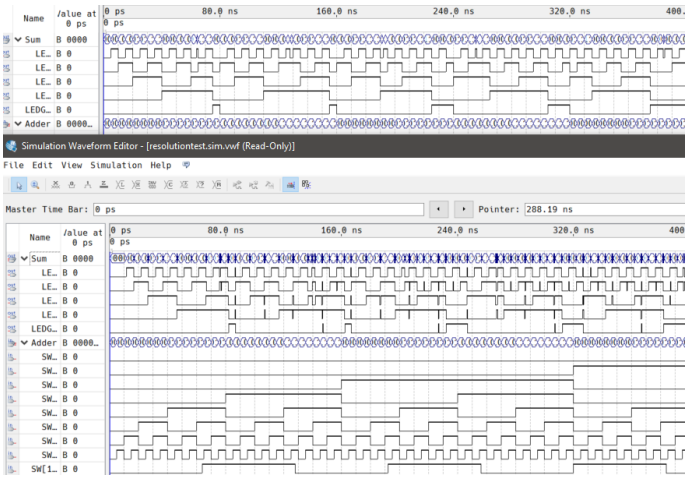

Fig. 5: *Udsnit af timingsimuleringeringen der viser funktionaliten for adderen med carry.*

For at teste de to adders funktionalitere satte vi nogle testcases op, der blot skulle addere to binære input, den ene test er vist i tab. 5.

Input	Forventet output	Reelt output
A: 0b0110+	S: 0b1000	S: 0b1000
B: 0b0010	U: 0b1000	U: 0b1000

Tab. 5: *Testsetup til signed og unsigned four-bit-adders.*

1.1.4 Diskussion

Sammenholder man de to RTL-views ses det funktionaliteten i de adders er ens, dette vises på fig. 1 og ses på technology map viewet i fig. 4.
Ved den funktionelle timingsimulering fremkom en række uforklarlige outputs, se fig. 6 .

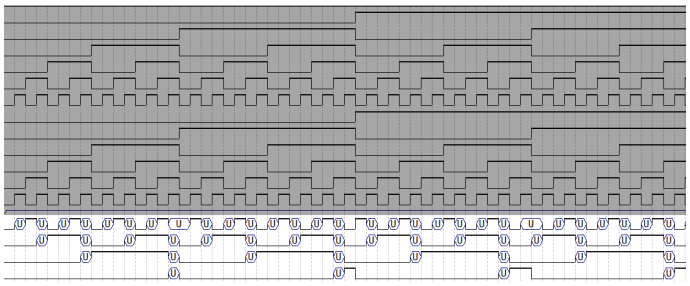

Fig. 6: *Timing issues fra signed og unsigned Adder.*

1.1.5 Konklusion

Ved at bruge `ieee.numeric_std`, kan man vha. meget få operationer, skrive relativt komplekse funktioner via dataflowstilarten.
Desuden pakkes flere implementationsmuligheder nemt ind i en enkelt entity.

1.2 Concatenation

1.2.1 Introduktion

Vi ønsker i denne opgave at benytte et af VHDL's biblioteker til simplificere nogle processerser. Dette gøre ved at bruge *concatenation*. Ved at udnytte funktioner fra `ieee.numeric_std.ALL` - specielt dens multiplikations og divisions operatøre, vil vi kunne omarrangere bits på en nem måde.

1.2.2 Design og implementering

Med en viden om at `0b0001 * 2 = 0b0010 & 0b0010 * 4 = 0b1000`, og `0b1000 / 2 = 0b0100 & 0b0100 / 4 = 0b0001`, skrev vi en kode der på den måde implementerede en bitrearranger, denne ses i source code table 6. Udover bitshifterne, er der også en bitrotater der tager to *slices* af vores `std_logic_vector` og omarrangere dem i en ny `std_logic_vector`.

```
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 shift_div;
11
12 ARCHITECTURE logicshift OF shift_div IS
13 BEGIN
14     a_shl <= a(6 downto 0) & '0';
15     -- shiftes til venstre og begraenses til 8 pladser
16     a_shr <= '0' & '0' & a(5 downto 0);
17     -- shiftes til hoejre med 2 og begraenses til 8 pladser
18     a_ror <= a(2 DOWNTO 0) & a(7 DOWNTO 3);
19     -- Rotere 3 til hoejre - plads er stadig 8.
20 END logicshift; -- logicshift
```

Tab. 6: *Entitien der viser de 3 funktioner: `a_shl`, `a_shr`, `a_ror`*

```
--              A_shl(7 DOWNTO 0) => LEDR(17 DOWNTO 10),
--              A_shr(7 DOWNTO 0) => LEDR(7 DOWNTO 0),
--              A_ror(7 DOWNTO 0) => LEDG(7 DOWNTO 0)
--
--          );

    UUT4 : ENTITY mult
        PORT MAP
        (
            -- INPUTS
            A(7 DOWNTO 0) => SW(7 DOWNTO 0),
            B(7 DOWNTO 0) => SW(15 DOWNTO 8),
```

Tab. 7: *Testbench port mappingen*

Ved at sende et output til LED'erne har vi mulighed for at få et visuelt feedback, og dermed udføre en række test, som ses i tab. 8.

Input	Forventet output	Reelt output
`input: 0b00011110`	`a_shl: 0b00111100`	`a_shl: 0b00111100`
	`a_shr: 0b00000111`	`a_shr: 0b00000111`
	`a_ror: 0b11000011`	`a_ror: 0b11000011`

Tab. 8: *Testcase til `shift_div` funktionaliteten*

1.2.3 Resultater

Testen i tab. 8 viste tilfredsstillende resultater, der blev 1 shiftet til venstre, 2 til højre og en rotation på 3 mod højre.

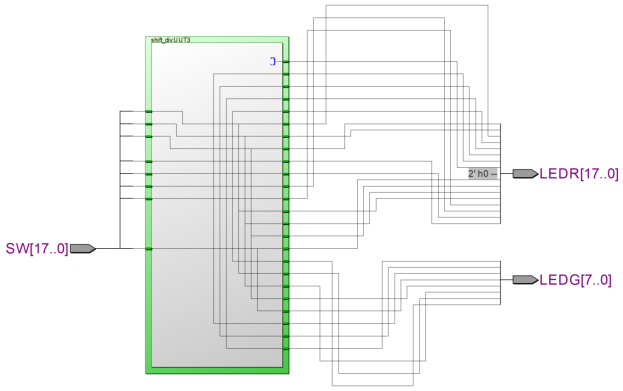

Fig. 7: *RTL view af hele entitien i source code table 6 — det ses at der ingen logiske elementer bruges — der flyttes blot rundt på interne forbindelser.*

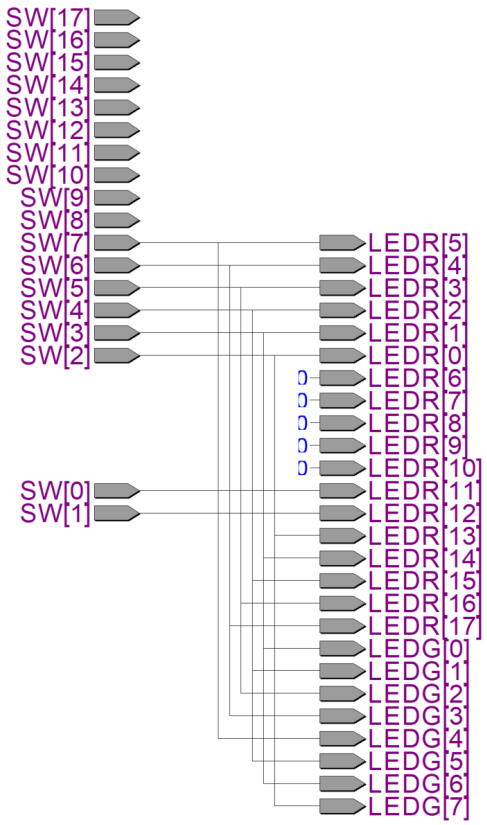

Fig. 8: *Technology mapping view af hele `shift_div` funktionen — igen ses det at der flyttes blot rundt på interne forbindelser.*

1.2.4 Diskussion

Efter syntesen gennemgik vi RTL mappingen og technology mapping viewet - de viser begge at der ikke bliver brugt et eneste logisk element i denne proces. Der kan altså shiftes og roteres *gratis*[1] i VHDL.

1.2.5 Konklusion

Med få linjer kode kan man skabe en *gratis* funktionalitet, forstået på den måde at *ingen* logiske elementer brugt.

1.3 Multiplication

1.3.1 Introduktion

FPGA-boardet vi arbejder med har en række indbyggede 9-bits multipiere - hvad sker der hvis man slår dem fra?

[1] Dog gælder: $a \cdot 2^n \quad a, n \in \mathbb{Z}$

1.3.2 Design og implementering

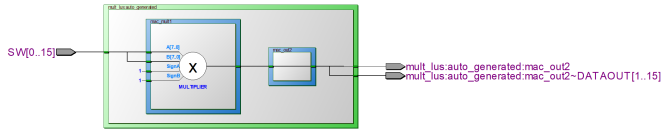

Fig. 9: Technology map view der viser `mult` funktionen.

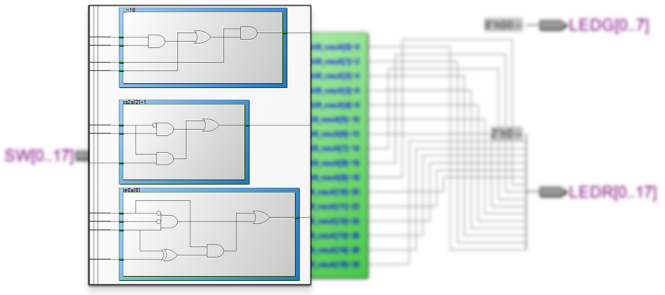

Fig. 10: Ved at aktivere DSP Block Balancing, tvinges FGPA'en til at bruge hardware logikelementer — Her er vist et udsnit af mappingen.

```
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 : IN std_logic_vector(31 DOWNTO 0);
8          B : IN std_logic_vector(31 DOWNTO 0);
9          Prod : OUT std_logic_vector(63 DOWNTO 0)
10     );
11 END mult;
12
13 ARCHITECTURE multiplication OF mult IS
14
15 BEGIN
16     Prod <= std_logic_vector(
17         signed(A) * signed(B)
18         );
19 END;
```

Tab. 9: Entitiy der viser de multiplier funktionen: `mult`

```
1     UUT4 : ENTITY mult
2         PORT MAP
3         (
4             -- INPUTS
5             A(7 DOWNTO 0) => SW(7 DOWNTO 0),
6             B(7 DOWNTO 0) => SW(15 DOWNTO 8),
7             -- OUTPUTS
8             Prod(15 DOWNTO 0) => LEDR(15 DOWNTO 0)
9         );
```

Tab. 10: *Testbench for* `mult`

1.3.3 Resultater

Input	Forventet output	Reelt output
A: 0b00010100 × B: 0b00111000	0b0000010001100000	0b0000010001100000

Tab. 11: *Testcase til* `mult`

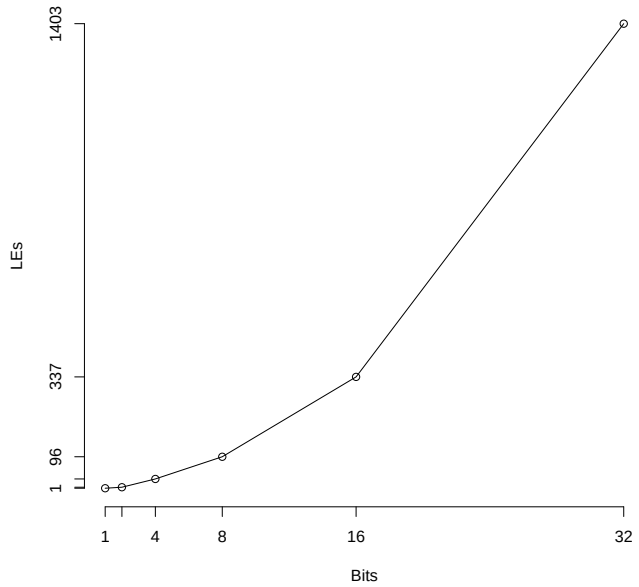

Fig. 11: *Plot der viser forholdet mellem bits og logikelementer - dette er en potensieludvikling.*

2

[2]Dog gælder: $a \cdot b \quad a, b \in 2^n \quad n \in \mathbb{Z}$

1.3.4 Diskussion

Bits	LEs
1	1
2	4
4	29
8	96
16	337
32	1403

Tab. 12: *Forholdet m. bits og logiskeelementer*

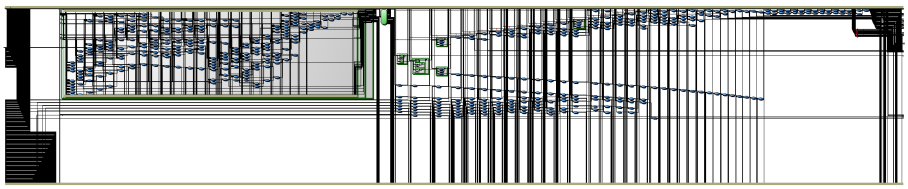

Fig. 12: *Et kaotisk indblik i en 32 bits multiplier — ét system med 1403 logiskeelementer, for mange til at have på én side.*

Det ses ud fra compilation reporten at der ved 2 x 32 bit input skal bruges 1403 logiskeelementer, se Technology map viewet fig. **??**
Ved test med DSP block slået fra skulle der først bruges logiskeelementer v. 32 bits multiplikation, alt derunder blev klaret af den embedded multiplier.

1.3.5 Konklusion

Denne øvelse viste ikke meget om multiplikation, udover at det fungere fremragende på FGPA'en, men viste tilgengæld hvor effektivt FPGA'en kan syntetiseres.

2 Øvelse 4 — Dataflow-style Combinatorial Designs

I denne øvelse er en testbench brugt som mapper switches, keys, hex displays og LED'er på DE2-boardet. I source code table 13 ses testbenchens interface.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE work.ALL;
4
5  ENTITY UnitUnderTest IS
6      PORT (
7          SW : IN std_logic_vector(17 DOWNTO 0);
8          KEY : IN std_logic_vector(3 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         LEDR : OUT std_logic_vector(0 DOWNTO 0)
13     );
14 END;
```

Tab. 13: *Testbench interface brugt i øvelse 4*

2.1 Binary to 7-Segment Decoder Using Selected Signal Assignment

2.1.1 Introduktion

I denne opgave skal der implementeres en BCD til 7-segment decoder ved brug af "Selected Signal Assignment", altså en såkaldt `WITH-SELECT` statement i VHDL.

2.1.2 Design og implementering

Source koden for vores decoder ses herunder i source code table 14. Som givet i opgavens IBD har vi et 4-bit input signal `bin` som mappes til et 7-bit output signal `Sseg`, således at displayet viser den hexadecimale værdi angivet på inputtet. Desuden har vi inkluderet en `WHEN OTHERS` situation til sidst for at undgå latches i vores design.

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

ENTITY bin2sevenseg IS
    PORT (
        bin : IN std_logic_vector(3 DOWNTO 0);
        Sseg : OUT std_logic_vector(6 DOWNTO 0)
    );
END bin2sevenseg;

ARCHITECTURE sevenseg OF bin2sevenseg IS

BEGIN
    WITH bin SELECT
        Sseg <= "1000000" WHEN "0000", -- 0
        "1111001" WHEN "0001", -- 1
        "0100100" WHEN "0010", -- 2
        "0110000" WHEN "0011", -- 3
        "0011001" WHEN "0100", -- 4
        "0010010" WHEN "0101", -- 5
        "0000010" WHEN "0110", -- 6
        "1111000" WHEN "0111", -- 7
        "0000000" WHEN "1000", -- 8
        "0011000" WHEN "1001", -- 9
        "0001000" WHEN "1010", -- A
        "0000011" WHEN "1011", -- B
        "0100111" WHEN "1100", -- C
        "0100001" WHEN "1101", -- D
        "0000110" WHEN "1110", -- E
        "0001110" WHEN "1111", -- F
        "-------" WHEN OTHERS;
    END sevenseg;
```

Tab. 14: *Implementering af BCD til 7-segment i VHDL*

Vores decoder entity `bin2sevenseg` testes med testbenchen `UnitUnderTest` som set nedenfor i source code table 15.

```
ARCHITECTURE structural OF UnitUnderTest IS
BEGIN
    UUT0 : ENTITY bin2sevenseg
        PORT MAP
        (
            -- INPUTS
            bin(3 DOWNTO 0) => SW(3 DOWNTO 0),
            -- OUTPUTS
            Sseg(6 DOWNTO 0) => HEX0
        );
END;
```

Tab. 15: Test af entity bin2sevenseg

2.1.3 Resultater

I opgaven undersøges en multiplexer (mux) lavet vha. WITH-SELECT statement i VHDL, til at styre et 7-segment display. Når koden syntetiseres ser vi på det resulterende RTL-view fig. 13 at der ikke bare er én, men hele 7 multiplexere.

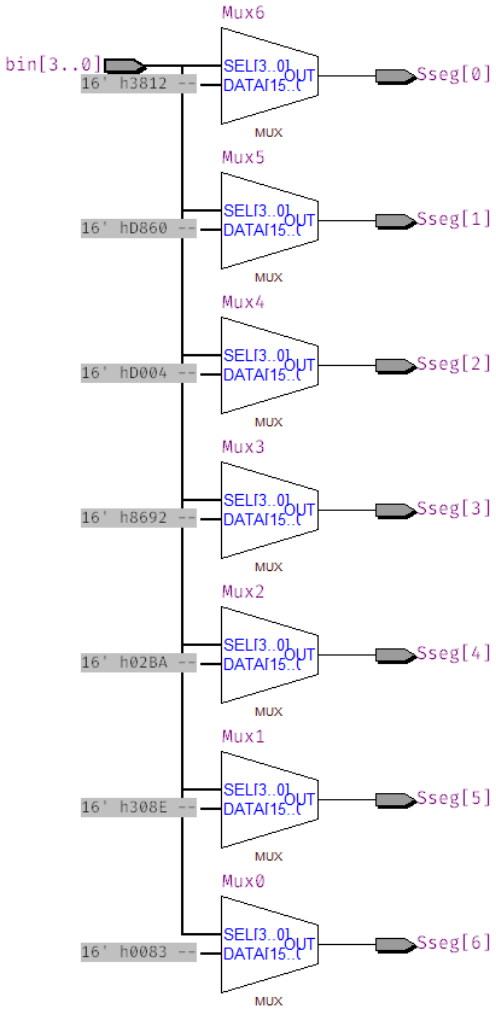

Fig. 13: RTL-view af den syntetiserede decoder

2.1.4 Diskussion

Vi ser ud fra vores RTL-view på fig. 13 at vores ene selected signal assignment er blevet syntetiseres til 7 multiplexere som hver driver et enkelt segment på 7-segment displayet med deres select baseret på `bin` inputtet. Dette giver mening da hver segment så kan drives af en meget simpel multiplexer.

2.1.5 Konklusion

Vi har i denne opgave erfaret at man med en meget simpel `WITH-SELECT` statement nemt kan lave omfattende mapninger mellem signaler af forskellige størrelser. Dette står i kontrast med hvor tidskrævende og fejlbarligt det ville være at lave denne mapning med basale logiske operatorer.

2.2 The Conditional Signal Assignment

2.2.1 Introduktion

Vi ønsker i denne opgave at implementere systemet der er vist i opgavetekstens figur 2. Dette skal gøres med en "Conditional Signal Assignment", dvs. en `WHEN-ELSE` statement i VHDL. Med denne implementering viser vi en anden måde at multiplex/demultiplex i VHDL.

2.2.2 Design og implementering

Vores implementering af HEX multiplexeren ses i source code table 16. Vi bruger vores `bin2sevenseg` entity fra forrige opgave, der mapper det binære input `bin` til tre interne signaler. Output signalet `tsseg` gives så en værdi baseret på inputtet `sel`, hvoraf to output er hardcoded til at vise "On"og "Err", det tredje er en konkatinering af de interne signaler.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  USE work.ALL;
5
6  ENTITY hex_mux IS
7      PORT (
8          bin : IN std_logic_vector(11 DOWNTO 0);
9          sel : IN std_logic_vector(1 DOWNTO 0);
10         tsseg : OUT std_logic_vector(20 DOWNTO 0)
11     );
12 END hex_mux;
13
14 ARCHITECTURE mux OF hex_mux IS
15     SIGNAL seg1 : std_logic_vector(6 DOWNTO 0);
16     SIGNAL seg2 : std_logic_vector(13 DOWNTO 7);
17     SIGNAL seg3 : std_logic_vector(20 DOWNTO 14);
18 BEGIN
19     b2sseg1 : ENTITY bin2sevenseg
20         PORT MAP(
21             bin => bin(3 DOWNTO 0),
22             Sseg => seg1
23         );
24
25     b2sseg2 : ENTITY bin2sevenseg
26         PORT MAP(
27             bin => bin(7 DOWNTO 4),
28             Sseg => seg2
29         );
30
31     b2sseg3 : ENTITY bin2sevenseg
32         PORT MAP(
33             bin => bin(11 DOWNTO 8),
34             Sseg => seg3
35         );
36
37     tsseg <= "10000000101011111111" WHEN sel = "11" ELSE -- "On
       "
38         seg3 & seg2 & seg1 WHEN sel = "10" ELSE
39         "00001100101111101011111" WHEN sel = "01" ELSE -- "Err"
40         (OTHERS => '-');
41 END mux; -- mux
```

Tab. 16: *Implementering af HEX multiplexer med Conditional Signal Assignment*

Som i forrige opgave er vores HEX multiplexer testet med testbench entity `UnitUnderTest`, dette ses i source code table 17

```
ARCHITECTURE structural OF UnitUnderTest IS
BEGIN
    UUT1 : ENTITY hex_mux
        PORT MAP
        (
            -- INPUTS
            bin => SW(11 DOWNTO 0),
            sel => KEY(1 DOWNTO 0),
            -- OUTPUTS
            tsseg(6 DOWNTO 0) => HEX0,
            tsseg(13 DOWNTO 7) => HEX1,
            tsseg(20 DOWNTO 14) => HEX2
        );
END;
```

Tab. 17: Test af entity hex_mux

2.2.3 Resultater

Vi har brugt conditional signal assignment til at implementere en multiplexer der vælger mellem tre forskellige output. Syntesen af vores kode giver det resulterende RTL-view som set på fig. 14.
Vi så ingen inferred latches i vores design, da vi som set på source code table 16 linje 40 har husket en assignment uden condition, som bliver brugt i alle de tilfælde der ikke eksplicit er taget højde for.

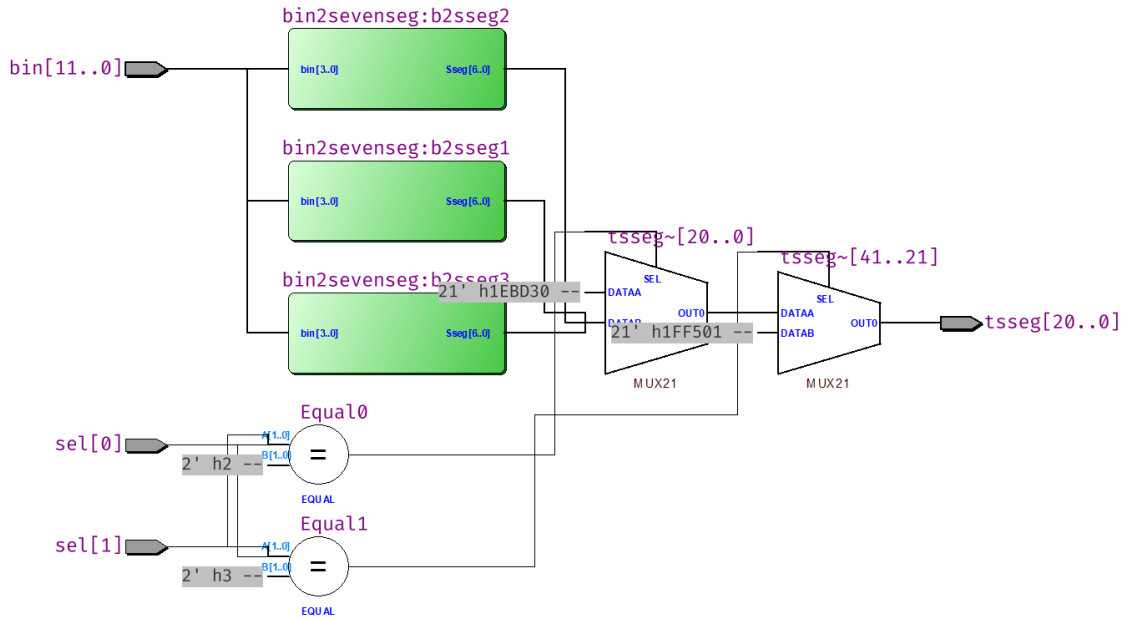

Fig. 14: RTL-view af multiplexer implementeret med conditional signal assignment

2.2.4 Diskussion

Ud fra det resulterende RTL-view kan vi se at *synthesizeren* har lavet vores HEX multiplexer med to multiplex komponenter i serie. Vores `sel` input sammenlig-

nes med hardcodede værdier og bruges som input select til de to multiplexere. Den første vælger så mellem en hardcoded værdi og det sammensatte output fra vores decodere, den anden vælger mellem output fra forrige mux eller en hardcoded værdi. Dermed har vi opnået en 3-input mux som ønsket.

2.2.5 Konklusion

Som i forrige opgave har vi her set at man med en meget simpel `WHEN-ELSE` statement kan vælge mellem forskellige input. En af styrkerne ved en `WHEN-ELSE` er at der kan bruges forskellige signaler til hver condition, designet i denne opgave var dog så simpelt at det ikke var nødvendigt.

2.3 Table Lookup

2.3.1 Introduktion

I denne opgave vil vi bruge en tredje metode til implementering af multiplexing, nemlig et "lookup table". Vi vil med denne metode implementere sandhedstabellen som vist i opgavens tabel 1. Lookup tabellen defineres som en hardcoded konstant array med output værdier, hvorefter den numeriske værdi af inputsne bruges som indeks i arrayet.

2.3.2 Design og implementering

Vi har ud fra tabel 1 i opgaven implementeret entity `table_lookup` med 3 inputs a, b og c, og outputtet x. Source code table 18 viser vores implementering, med den konstante array `table` der har output værdierne som defineret i sandhedstabelle og type casting af de tre input, først til typen `unsigned` og derefter til `integer`, således at værdien kan bruges som indeks.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY table_lookup IS
6      PORT (
7          a, b, c : IN std_logic;
8          x : OUT std_logic
9      );
10 END table_lookup;
11
12 ARCHITECTURE lookup OF table_lookup IS
13     CONSTANT table : std_logic_vector(0 TO 7) := "11010--1";
14 BEGIN
15     x <= table(to_integer(unsigned'(a, b, c)));
16 END lookup;
```

Tab. 18: *Implementering af table lookup i VHDL*

Vi har igen brugt vores testbench entity `UnitUnderTest` til at teste vores design. Set i source code table 19 er vores architecture for test af table lookup entity.

```
ARCHITECTURE structural OF UnitUnderTest IS
BEGIN
    UUT2 : ENTITY table_lookup
        PORT MAP
        (
            -- INPUTS
            a => SW(2),
            b => SW(1),
            c => SW(0),
            -- OUTPUTS
            x => LEDR(0)
        );
END;
```

Tab. 19: Test af entity `table_lookup`

2.3.3 Resultater

Vores design resulterede efter syntetisering i det RTL-view som er set på fig. 15. Desuden opstillede vi en funktionel simuleringstest som set på fig. 16 hvor alle de 8 mulige kombinationer af inputs og resulterende outputs gennemgås.

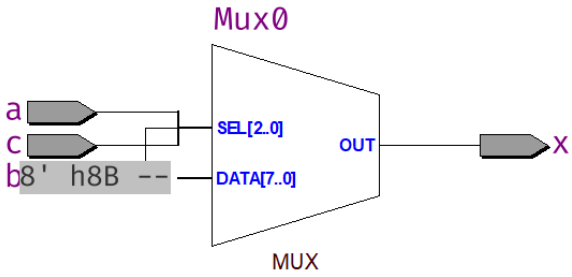

Fig. 15: RTL-view af den syntetiserede table lookup

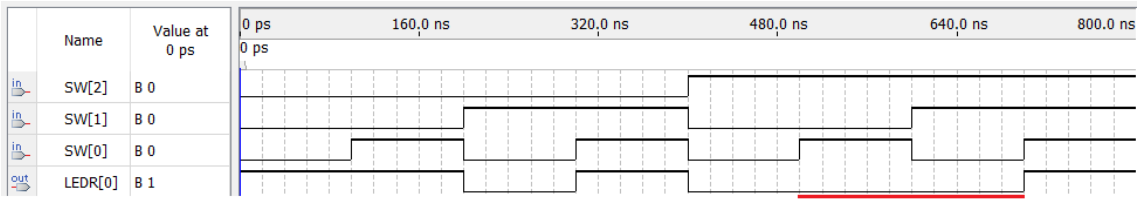

Fig. 16: Funktionel simulering af table lookup

2.3.4 Diskussion

Vi ser på fig. 15 at vores table lookup er blevet syntetiseret til et multiplexer med vores tre input som select og vores hardcodede værdi som data, dette stemmer overens med vores forventning.
Desuden kan vi ud fra vores funktionelle simulation som set på fig. 16 se, at sammenhængen mellem input og output stemmer overens med sandhedstabellen. Vi

ser desuden at de to outputs som er sat til "don't care" (markeret med rødt på billedet) bliver sat til 0 i simulationen.

2.3.5 Konklusion

Vi har i denne opgave erfaret at man som alternativ til `WITH-SELECT` og `WHEN-ELSE` også kan bruge en array af konstante værdier og omdanne input til et indeks i arrayet. Dette bliver korrekt omdannet til en multiplexer ligesom de andre selection metoder. Vi erfarede også at i den funktionelle simulation bliver "don't care" holdt til 0.