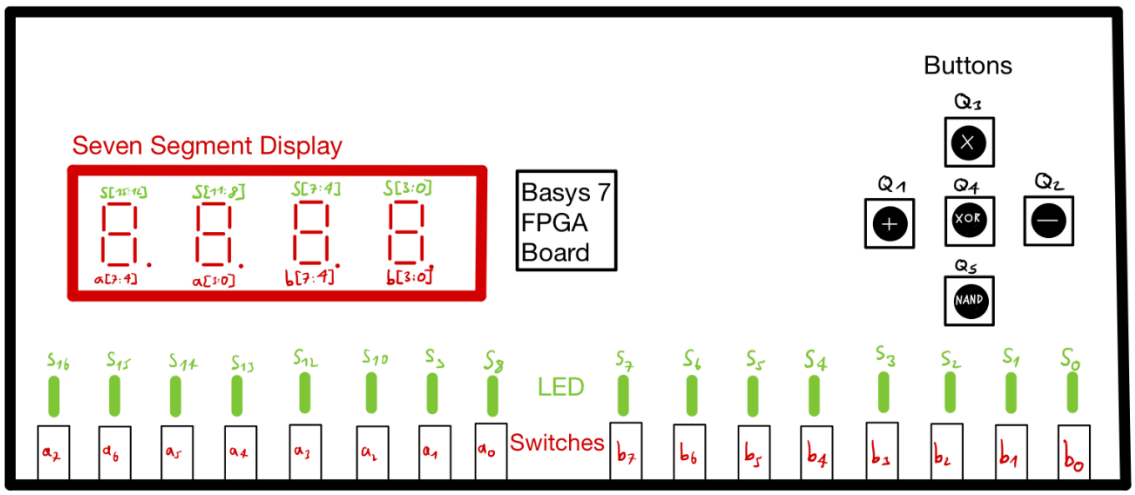
Taschenrechner FPGA Report

By Yash Trivedi & Severin Memmishofer

Inhaltsverzeichnis

1. Bedienungsanleitung / Features

In diesem Abschnitt werden wir eine kurze Einführung in die Bedienung unseres Taschenrechners geben und dessen Funktionen erklären. Die Bedienung ist relativ einfach, wenn man entsprechende Grundkenntnisse im binären Zahlensystem und der Umwandlung zum Hexadezimalsystem hat. Für die Subtraktion wird ausserdem Erfahrung im Umgang mit dem 2er-Komplement vorausgesetzt.



a. Default State

Als erstes braucht man eine Stromversorgung. Da das Programm auf dem Board intern gespeichert ist, muss man nicht zwingend einen PC haben, auf dem Vivado läuft, sondern es ist jede angemessene Stromversorgung ausreichend. Wir haben es sogar mit einer Powerbank getestet und es hat einwandfrei funktioniert, was in der Benutzung eine grosse Flexibilität bietet. Die Stromversorgung wird über ein USB-Kabel gewährleistet (entweder über den Micro-USB Anschluss oder den USB-A Anschluss).

Nachdem das Board mit Strom versorgt ist, kann man es mit dem «Power» - Schalter oben links anschalten.

Standardmässig sollten nun die eingestellten Zahlen «A» und «B» auf dem Display zu sehen sein. Dabei stehen die linken 8 Switches für die Zahl A und die rechten Switches für die Zahl B. Die Zahl muss man nun in Binärform eingeben. (Schalter in unterer Stellung entspricht einer 0 und Schalter in oberer Stellung einer binären 1). Wobei das niedrigste Bit jeweils ganz rechts steht und das höchstwertige ganz links.

Wenn man nun etwas mit den Switches herumspielt, kann man sehen, dass im 7 Segment Display die entsprechenden Zahlen in Hexadezimal-Form zu sehen sind. Die beiden linken Digits stehen dabei für die linken 8 Switches (Zahl A) und die beiden rechten Digits entsprechend für die Zahl B.

b. Addition

Wenn man das Resultat der Addition sehen möchte, kann man den linken Button drücken. Dann wird das Resultat der Addition der zwei Zahlen «A» und «B», die auf den Switches eingestellt sind, angezeigt. Das Resultat erscheint dabei einerseits auf dem 7-segment-display als Hexadezimalzahl und gleichzeitig als Binärzahl auf den LEDs, falls man mit dem Binärsystem besser vertraut ist. Bei den LEDs steht die LED ganz rechts (LD0) für das niederwertigste Bit und die Zahl wird von insgesamt 9 LEDs angezeigt. Wenn das 9te LED (LD8) leuchtet, symbolisiert das also einen Übertrag. Dies ist deswegen nötig, weil die Addition von zwei 8-Bit-Zahlen ja maximal 255 + 255 = 510 ergibt und man für das Darstellen dieser Zahl 9 Bits benötigt.

c. Subtraktion

Beim Drücken des rechten Buttons wird analog zur Addition das Resultat der Subtraktion der Zahl «A» minus der Zahl «B» angezeigt. Auf dem 7-segment-display als Hexadezimalzahl und auf den LEDs im binären Zahlensystem.

Wichtig: Das Resultat ist im 2er-Komplement. Für korrekte Ergebnisse muss man die Zahl also umwandeln. Da das Resultat im 2er-Komplement angezeigt wird, kommt der 9ten LED (LD8) eine besondere Funktion zuteil: Wenn sie leuchtet, dann ist die Zahl negativ; falls nicht, handelt es sich beim Resultat um eine positive Zahl.

d. Multiplikation

Wenn man den oberen Button drückt, wird das Resultat der Multiplikation A * B angezeigt. Wiederum ähnlich, wie bei Addition und Subtraktion in Form einer Hexadezimalzahl auf dem 7-segment-display oder in binärer Form auf den LEDs. Hierbei sind jedoch nicht nur 9 LEDs aktiviert, sondern alle 16. Diese werden benötigt, da das grösstmögliche Ergebnis (255 * 255 = 65'025 = ‘1111111000000001’) eine 16 Bit Zahl ist.

e. XOR

Beim Drücken des mittleren Buttons wird das Resultat der Operation (A XOR B) angezeigt. Wiederum kann man es entweder als Hexadezimalzahl oder auf den LEDs als Binärzahl ansehen. Es empfiehlt sich jedoch, auf die LEDs zu schauen, da das Resultat so intuitiver abzulesen ist.

f. NAND

Wenn man den unteren Button drückt, wird analog wie bei XOR, das Resultat der Operation (A NAND B) angezeigt. Wiederum sowohl als Hexadezimalzahl auf dem 7-seg-display, oder binär auf den LEDs.

2. Bauanleitung

In diesem Abschnitt werden wir einen Überblick geben, wie man unser Projekt nachbauen könnte, und wir werden auch einige Tipps geben und von unseren Erfahrungen während der Programmierung berichten.

a. Besorgung des Boards

Der erste Schritt wäre die Auswahl und Bestellung eines passenden FPGA-Boards. Wir haben das «Xilinx Basys 3» für ungefähr 150 CHF gewählt. Unseren Taschenrechner kann man aber mit beliebigen vergleichbaren Boards auch bauen. Je nachdem läuft es auf einer anderen Programmiersprache, aber die Grundfunktionalität ist die gleiche. Unser Board läuft auf «Verilog», aber es wäre problemlos auch mit anderen Boards, die auf der Sprache «VHDL» laufen, machbar. Zur Programmierung mittels Verilog haben wir die Entwicklungsumgebung Vivado benutzt. Wichtig ist nur, dass das Board ein 7-segment-Display besitzt und mindestens 16 Switches und 4 Pusbuttons. (Falls das Board weniger Switches besitzt, müsste man sich einfach auf die Addition von kleineren Zahlen beschränken).

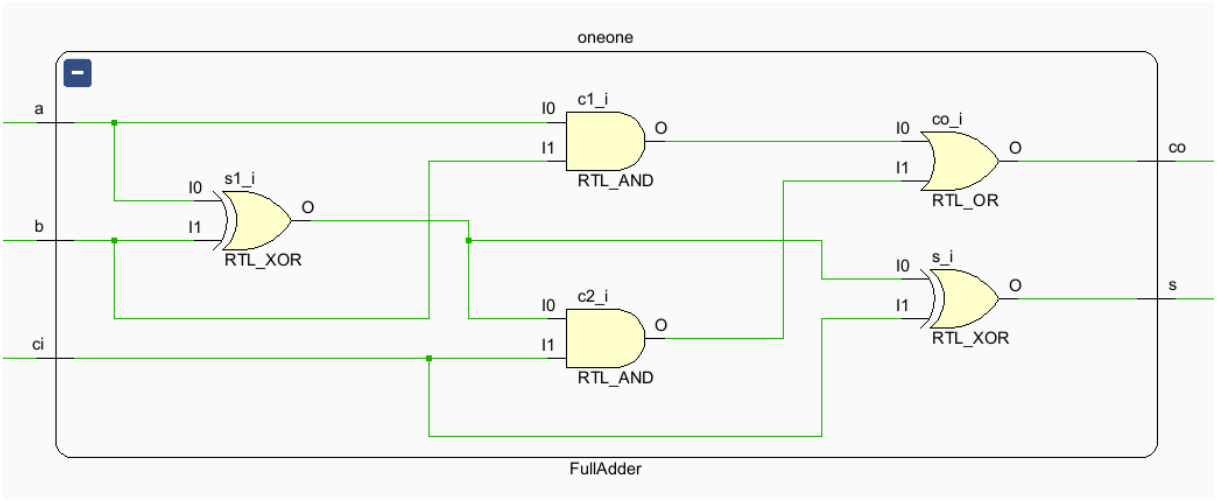
Wichtig: Man braucht noch ein Kabel für die Programmierung am PC und es muss unbedingt ein Micro-USB zu USB-A Kabel sein, da die Programmierung beim Basys3-Board nur über den Micro-USB Anschluss funktioniert; nicht aber über den USB-A Anschluss!

b. Addierer

Wir haben mit dem einfachsten Baustein eines Taschenrechners begonnen, und zwar mit einem 1-Bit-Addierer. Wenn man diesen einmal implementiert hat, kann man sich schrittweise an einen 8-Bit-Addierer und an andere Funktionen wagen. Zuerst müssen wir wissen, wie die Wahrheitstabelle eines 1-Bit-Addierers aussieht und dementsprechend, welche Gates wir für die Implementierung brauchen.

A	B	carry in	Sum	carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Nachdem wir die Wahrheitstabelle des Full-Adders kennen, können wir diesen mit den Gates implementieren. Nachfolgend eine Übersicht der Implementierung auf Gate-Ebene.

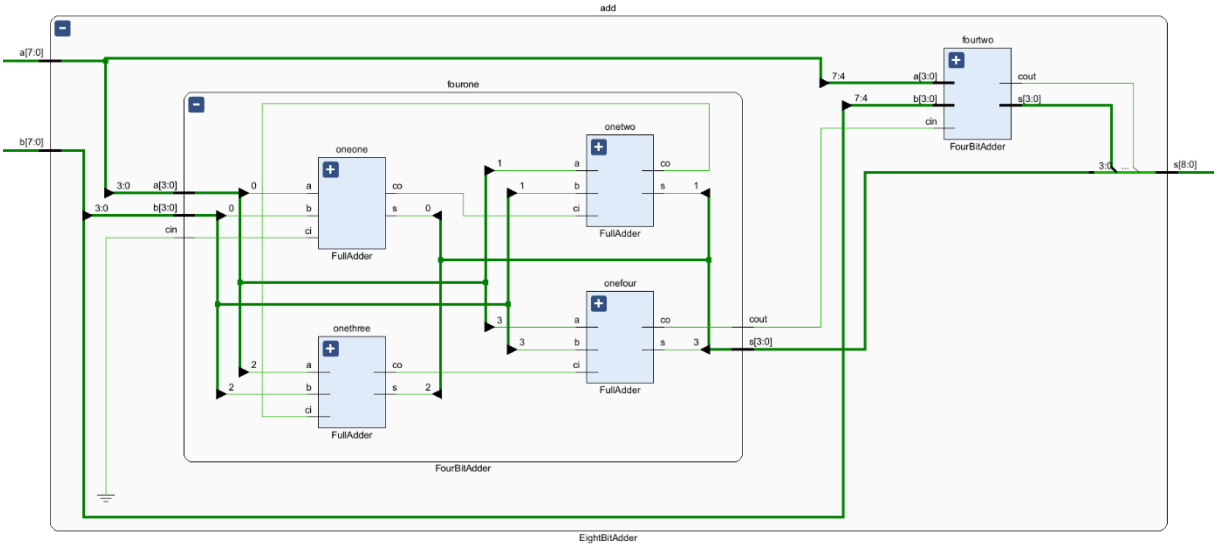


Und hier noch die entsprechende Implementierung im Code:

```
`timescale 1ns / 1ps
// Create Date: 20.11.2022 12:52:44

// Klassischer Full-Adder
module FullAdder(input a, input b, input ci, output s, output co);

// implementiert, wie in der Übungsstunde (Übung 1)

xor(s1,a,b);
xor(s,s1,ci);
and(c1,a,b);
and(c2,s1,ci);
or(co,c1,c2);

endmodule
```

Hinweis: Die Gates sind jeweils so implementiert, dass als erstes der Output genannt wird und anschliessend die zwei Inputs. xor(s1,a,b) heisst also, dass der Output s1 dem Resultat von xor(a,b) entspricht.

Nachfolgend können wir die einzelnen Full-Adder zusammensetzen zu einem 8-Bit-Adder. Dies kann man auf verschiedene Arten tun. Wir haben uns dazu entschieden, zuerst einmal einen 4-Bit-Adder zu machen und danach aus zwei 4-Bit-Addern einen 8-Bit-Adder zusammenzusetzen. Man hätte stattdessen auch direkt aus 8 1-Bit-Addern einen 8-Bit-Adder machen können. Nachfolgend unsere Gate-Implementierung und der entsprechende Code:




```
`timescale 1ns / 1ps
// Create Date: 20.11.2022 12:53:20

// Zusammensetzen von 4 One-Bit-Full-Adders
module FourBitAdder(input cin, input [3:0] a, input [3:0] b, output [3:0] s,
output cout);

FullAdder oneone(a[0],b[0],cin,s[0],c1);
FullAdder onetwo(a[1],b[1],c1,s[1],c2);
FullAdder onethree(a[2],b[2],c2,s[2],c3);
FullAdder onefour(a[3],b[3],c3,s[3],cout);

Endmodule
```

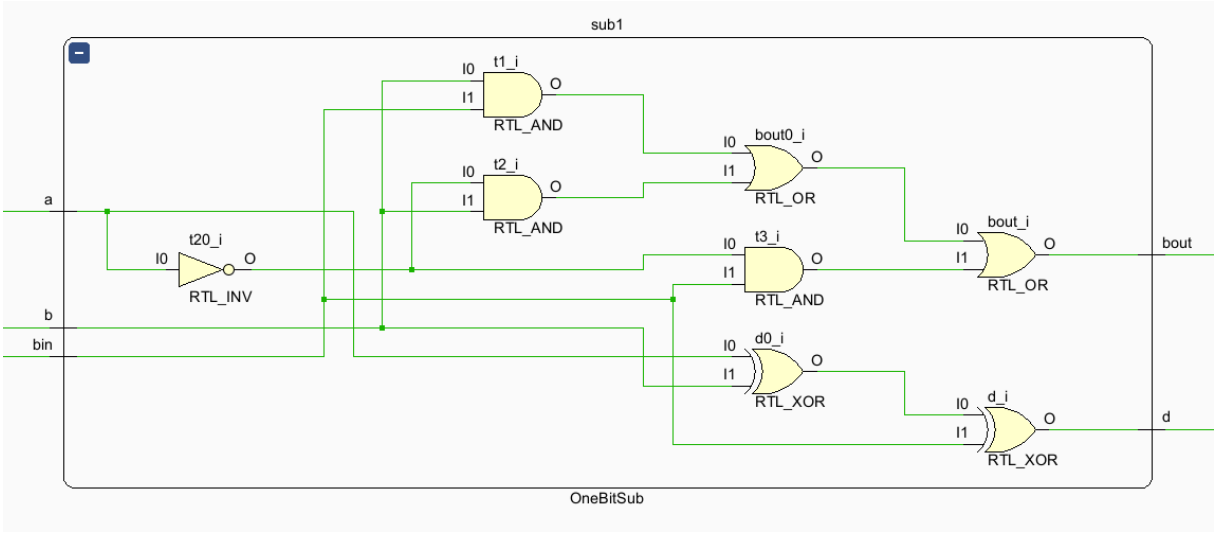
```
`timescale 1ns / 1ps
// Create Date: 22.11.2022 17:11:56

// Zusammensetzen von 2 4-bit-Addern
module EightBitAdder (input [7:0] a, input [7:0] b, output [8:0] s);

// temp ist ein Wire, welches den Carry temporär "speichert"
wire temp;
FourBitAdder fourone(0, a[3:0], b[3:0], s[3:0], temp);
FourBitAdder fourtwo(temp, a[7:4], b[7:4], s[7:4], s[8]);

endmodule
```

Hinweis: Das wire «temp» agiert hier sozusagen als Zwischenspeicher. Wir deklarieren also das wire und der CarryOut aus dem FourBitAdder wird danach als CarryIn im zweiten FourBitAdder verwendet.

c. Subtrahierer

Nun, da wir den Addierer haben, können wir die Module für den 8-Bit-Subtrahierer sehr ähnlich aufbauen. Dazu müssen wir wieder beim kleinsten Modul, dem 1-Bit-Subtractor beginnen. Hierfür müssen wir wieder zuerst die Wahrheitstabelle kennen:

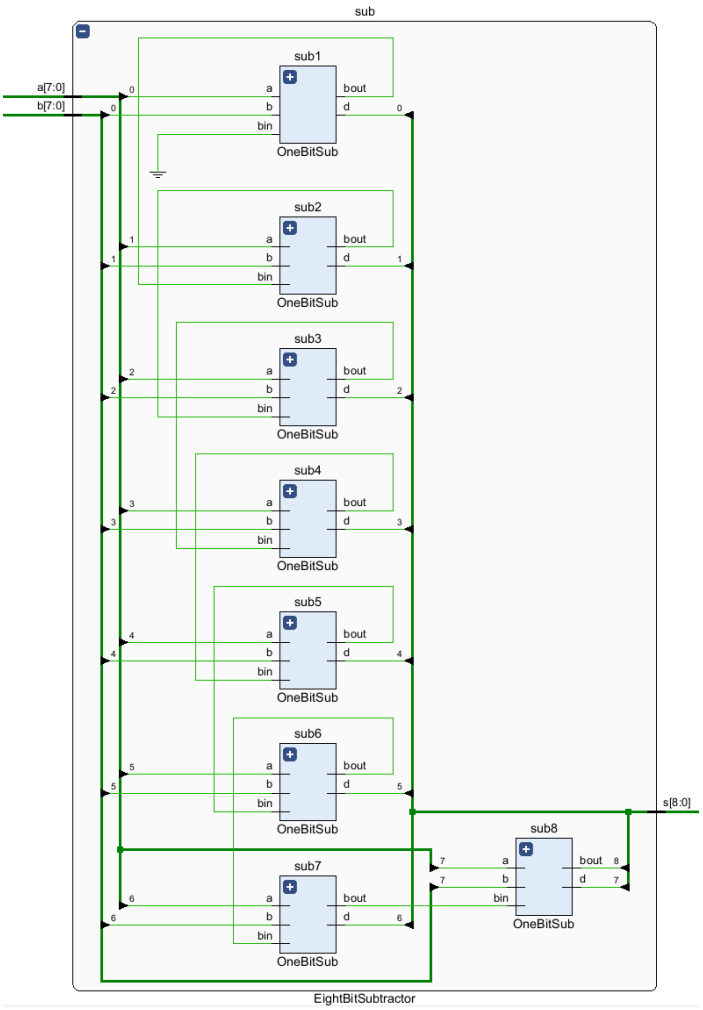
A	B	borrow input	D	borrow output
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1




```
`timescale 1ns / 1ps
// Create Date: 29.12.2022 15:16:33


module OneBitSub(
    input a, input b, input bin,
    output d, output bout
    );
    // Implementation Siehe Truth Table und Skizze im Report
    assign d = a ^ b ^ bin;
    assign t1 = b & bin;
    assign t2 = ~a & b;
    assign t3 = ~a & bin;
    assign bout = t1 | t2 | t3;
endmodule
```

Hier sieht man noch eine weitere Option: als Alternative zu den Gates kann man auch mit «assign» Statements arbeiten. Die Wellenlinie '~' steht für «NOT», '&' steht für «AND», '|' für «OR» und '^' für «XOR».

Nachdem wir das kleinste Modul haben, können wir analog die grösseren Module aufbauen. Wir haben uns hier für die Variante des 8-Bit-Subtractors aus 8 1-Bit-Subtractors entschieden.




```
`timescale 1ns / 1ps
// Create Date: 24.12.2022 16:21:25


module EightBitSubtractor(
input [7:0] a, input [7:0] b, output [8:0] s
    );
    OneBitSub sub1(a[0],b[0],0,s[0],borrow1);
    OneBitSub sub2(a[1],b[1],borrow1,s[1],borrow2);
    OneBitSub sub3(a[2],b[2],borrow2,s[2],borrow3);
    OneBitSub sub4(a[3],b[3],borrow3,s[3],borrow4);
    OneBitSub sub5(a[4],b[4],borrow4,s[4],borrow5);
    OneBitSub sub6(a[5],b[5],borrow5,s[5],borrow6);
    OneBitSub sub7(a[6],b[6],borrow6,s[6],borrow7);
    OneBitSub sub8(a[7],b[7],borrow7,s[7],s[8]);
endmodule
```

d. Pushbuttons

Für die Funktionalität der Pushbuttons sind einige Zwischenschritte notwendig. Da wir für unseren Taschenrechner die Funktion haben möchten, dass man je nachdem, welcher Pushbutton gedrückt ist, ein anderes Resultat sehen kann, müssen wir die Pushbuttons implementieren und ausserdem ihren Zustand speichern.

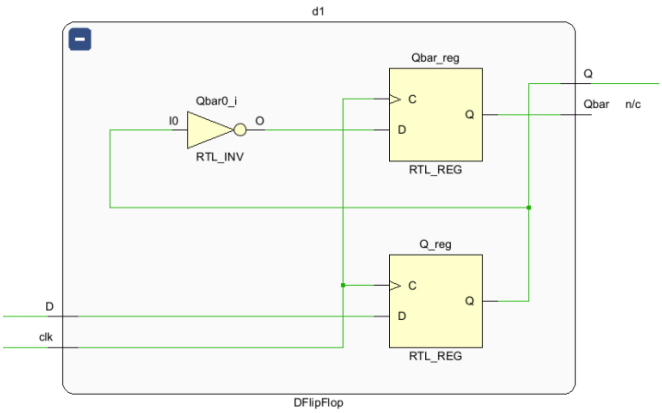
Da die Pushbuttons analog sind, ist ihr Signal jedoch am Anfang noch etwas instabil und «flackert» zwischen 0 und 1, bis es sich stabilisiert hat. Da wir nicht möchten, dass später unsere LEDs auch flackern, müssen wir die Buttons «debouncen». Dafür implementieren wir einen langsamen Clock (4Hz = 1s/4), sodass das Signal des Buttons Zeit hat, sich zu stabilisieren.

```
module SlowClk4Hz(
    input clk_in, // input 100 Mhz of the FPGA Board Clock
    output clk_out // 4Hz Slow Clock
    );
    reg [25:0] count = 0;
    reg clk_out;

    always @ (posedge clk_in) // Macht etwas jedes mal, wenn der clock-edge
positiv wird (rising-edge-triggered)
    begin // loop
    count <= count+1; // Zählt die Anzahl an "rising edges"
    if(count == 12_500_000) // Wenn die Anzahl an "rising edges = 12'500'000
ist; müssen wir unseren 4HZ-Clock ändern"
    begin
    count <= 0; // Danach müssen wir wieder von vorne mit Zählen beginnen, bis
zum nächsten Wechsel unseres 4Hz Clocks
    clk_out =~ clk_out; // Der 4Hz Clock wird invertiert
    end
    end
endmodule
```

Dazu müssen wir wissen, wie schnell der Clock des FPGA-Boards ist, was bei unserem Modell 100Mhz entspricht. Um den Clock zu verlangsamen, zählen wir die Anzahl positiven Taktflanken des eingehenden Clocks und erhöhen einen Counter. Um herauszufinden, wie hoch wir zählen müssen, damit der Output Clock 4Hz entspricht, geht man folgendermassen vor: Man teilt die eingehenden 100Mhz / 4Hz = 25'000'000. Und danach muss man die Hälfte davon nehmen, da wir ja nur die positive Taktflanke brauchen. Nun wechseln wir unseren Output 4Hz Clock jedes Mal, wenn der Counter 12'500'000 erreicht hat.

e. D – Flip – Flop

Da wir den Zustand unseres Buttons irgendwie speichern müssen, um danach, je nachdem, ob der Button gedrückt ist, oder nicht, eine entsprechende Aktion einleiten zu können, brauchen wir noch ein Speichermedium. Zum Beispiel ein klassisches taktgesteuertes D – Flip – Flop.



```
`timescale 1ns / 1ps
// Create Date: 04.12.2022 11:17:14

// Zum Speichern des Button-States
module DFlipFlop( // klassischer D-Flip-Flop
    input clk, // input clock; in unserem Fall slow Clock 4Hz
    input D, // input D; in unserem Fall der Push Button
    output reg Q,
    output reg Qbar // Damit wir nicht 2 verschiedene FlipFlops machen müssen;
Qbar ist das Gegenteil von Q
    );

    always @ (posedge clk) // Anpassung wird nur jedes Mal beim Rising Edge
gemacht
    begin
    Q <= D; // D wird auf Q zugewiesen
    Qbar <= !Q;
    end
endmodule
```

Auf die Implementierung im Main-Module kommen wir später zurück.

f. 7 – Segment – Display

Für die Implementierung des 7-seg-displays sind einige weiteren Überlegungen und Zwischenschritte nötig. Zuerst müssen wir wissen, wie das Display intern angesteuert wird, um es zu implementieren. Und zwar besteht das Display aus, wie der Name schon sagt, sieben Segmenten, die entweder leuchten können oder nicht. Um diese anzusteuern, gibt es sieben Datenleitungen (eine achte, wenn man die Datenleitung für den «decimal point» mitzählt, welchen wir hier aber nicht verwenden). Nun kann man aber nicht jeden der 4 Digits einzeln ansteuern, sondern man muss sich an einem kleinen Trick bedienen. Und zwar gibt es ja nur 7 Datenleitungen, aber wir haben 4 Digits. Zusätzlich gibt es noch 4 Leitungen, die die einzelnen Anoden der 4 Digits ansteuern. Diese bestimmen, ob das jeweilige Digit leuchtet oder nicht.

Um nun auf den 4 Digits *unterschiedliche* Zahlen anzeigen zu lassen, müssen wir also in einer hohen Frequenz jeweils zwischen den Digits wechseln. Und zwar «schickt» man auf den Datenleitungen die Daten für die Segmente, die man beleuchtet haben möchte, auf dem 1. Digit. (Für eine Zahl 1 wären das zum Beispiel die 2 «rechten» Striche). Gleichzeitig aktiviert man das 1. Digit mit der entsprechenden Anode. Danach wechselt man und schickt die Daten für Digit 2. Gleichzeitig aktiviert man mit den Anoden das Digit 2 und lässt das Digit 1 wieder ablöschen und so weiter. Dieser Wechsel der Datenleitungen und der Anoden muss jedoch synchron geschehen.

Wenn man nun genügend schnell zwischen den 4 Digits hin- und her wechselt, kann man also 4 unterschiedliche Zahlen anzeigen lassen. Da dieser Wechsel sehr schnell geschieht und die LEDs, sobald sie abdunkeln, gleich wieder aktiviert werden, ist der Vorgang für das menschliche Auge nicht wahrnehmbar und die Digits werden als «dauerhaft» beleuchtet wahrgenommen.

Beginnen werden wir mit einem schnellen Clock; der die Geschwindigkeit des Wechsels zwischen den einzelnen Ziffern vorgibt.

i. 7-seg-clock

```
`timescale 1ns / 1ps
// Create Date: 26.12.2022 17:14:35

// Analog implementiert, wie der Slow-Clock; einfach schneller
module SevenSegClock(
    input clk_in, // input 100 Mhz of the FPGA Board Clock
    output clk_out
    );
    // Da wir nur bis 5'000 zählen, brauchen wir nur 14 Register, um die Zahl
zu speichern.
    reg [13:0] count = 0; // vorher 13 bei 5'000
    reg clk_out;

    always @ (posedge clk_in) // Macht etwas jedes mal, wenn der clock-edge
positiv wird (rising-edge-triggered)
    begin // loop
    count <= count+1; // Zählt die Anzahl an "rising edges"
    if(count == 5_000) // Wenn die Anzahl an "rising edges = 5'000 ist; müssen
wir unseren 4HZ-Clock ändern"
    begin
    count <= 0; // Danach müssen wir wieder von vorne mit Zählen beginnen, bis
zum nächsten Wechsel unseres 4Hz Clocks
    clk_out =~ clk_out; // Der Clock wird invertiert
    end
    end
endmodule
```

Dieses Modul ist prinzipiell sehr ähnlich, wie beim vorher beschriebenen «SlowClock», nur dass wir hier weniger hoch zählen müssen, da wir ja einen schnelleren Clock haben möchten.

ii. Refresh Counter

```
`timescale 1ns / 1ps
// Create Date: 26.12.2022 17:21:36

// Für die Steuerung des 7SegDisplays: Zählt bei jedem "FastClk" einmal hoch;
// Sagt, welches Digit gerade angezeigt werden soll und wechselt die
zugehörigen Anoden
module refreshcounter(
    input refresh_clock,
    output reg[1:0] refreshcounter
    );
    // Da das Register nur 2 Bit gross ist; wird von 0 bis 3 gezählt; und
danach fängt es wieder von vorne an.
    always @(posedge refresh_clock) refreshcounter <= refreshcounter +1;
endmodule
```

Dieses Modul ist dafür gedacht, dass wir den Wechsel der Anoden und den der Datenleitungen aufeinander abstimmen können. Die Taktfrequenz entspricht dem obigen «7-segment-clock». Und zwar zählt dieser Counter von 0 – 3 und fängt danach wieder von vorne an, da das 2-Bit-Register

«voll» ist. Somit bestimmt es, welches der Digits gerade an der Reihe ist, und stimmt den Wechsel der Datenleitungen und der Anoden aufeinander ab.

iii. Anode_Control

```
`timescale 1ns / 1ps
// Create Date: 26.12.2022 17:27:51

// Sagt; abhängig vom refreshcounter; welche Anode gerade an der Reihe ist;
für das 4-Digit-Display
module anode_control(
    input[1:0] refreshcounter,
    output reg[3:0] anode = 0
    );

    always@(refreshcounter)
    begin
        case(refreshcounter)
        2'b00:
        begin
            anode = 4'b0111; // digit 1 on (1=OFF, 0=ON)
        end
        2'b01:
        begin
            anode = 4'b1011; // digit 2 on (1=OFF, 0=ON)
        end
        2'b10:
        begin
            anode = 4'b1101; // digit 3 on (1=OFF, 0=ON)
        end
        2'b11:
        begin
            anode = 4'b1110; // digit 4 on (1=OFF, 0=ON)
        end
        endcase
    end
endmodule
```

Dieses Modul nimmt als Input den vorher gemachten «refreshcounter» und aktiviert die entsprechende Anode, die gerade an der Reihe ist. Wenn der Counter 0 ist, ist das erste Digit an der Reihe. Dementsprechend muss die Anode für das erste Digit auf 0 stehen und die Anode für die anderen Digits auf 1.

Achtung: Wenn die Anode auf 0 gesetzt wird, bedeutet das, dass das entsprechende Digit leuchtet und eine Anode auf 1 bedeutet, dass es nicht leuchtet. Entgegen der Intuition!

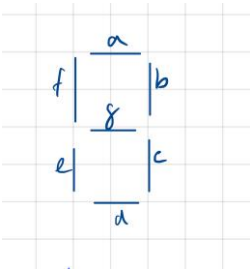
iv. BCD_control

```
`timescale 1ns / 1ps
// Create Date: 26.12.2022 17:41:53
```

```
// Sagt abhängig vom Refreshcounter; welches Digit gerade displayed wird; das
heisst welche Daten an die Kontrollleitungen
// des 7SegDisplays geschickt werden sollen.
// WICHTIG: Dies muss in Sync mit dem anode_control geschehen; diese zwei
Module müssen aufeinander abgestimmt sein.
module BCD_control(
input [3:0] digit1,
input [3:0] digit2,
input [3:0] digit3,
input [3:0] digit4,
input [1:0] refreshcounter,
output reg [3:0] displayed_digit = 0
    );

    always@(refreshcounter)
        begin
            case(refreshcounter)
                2'b00:
                    displayed_digit = digit1; //digit1 value (right digit)
                2'b01:
                    displayed_digit = digit2; //digit2 value
                2'b10:
                    displayed_digit = digit3; //digit3 value
                2'b11:
                    displayed_digit = digit4; //digit4 value (left digit)
            endcase
        end
endmodule
```

Dieses Modul funktioniert sehr ähnlich, wie das Modul «anode_control». Es nimmt DENSELBEN (Wichtig, damit der Wechsel aufeinander abgestimmt ist) Counter als Input und bestimmt, welches Digit gerade angezeigt werden soll. Wenn der Counter auf 0 steht, soll das erste Digit angezeigt werden und so weiter. Die Digits sind dabei 4 Bit gross (0-15 in Dezimal), da wir ja eine Hexadezimalzahl auf dem Display anzeigen möchten.

v. 7-segment-Decoder



Nun fehlt nur noch ein Decoder, der entsprechend des Input-Digits (0-15) bestimmt, welche Segmente des 7-seg-Displays leuchten sollen und einen entsprechenden Output gibt. Dazu müssen wir uns kurz den Aufbau eines einzelnen Digits und seiner Datenleitungen anschauen. In der Darstellung rechts sieht man, welche Datenleitungen mit welchem Segment korrespondieren.

x	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0

4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
A	0	0	0	1	0	0	0
b	1	1	0	0	0	0	0
c	0	1	1	0	0	0	1
d	1	0	0	0	0	1	0
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	0

Wichtig: Ähnlich wie bei den Anoden entspricht hier eine 0 einem leuchtenden Segment, und eine 1 steht für ein «abgelöschtes» Segment. Deswegen besteht die «8» zum Beispiel aus lauter 0-en, da für eine 8 alle Segmente leuchten müssen.

```
`timescale 1ns / 1ps
// Create Date: 26.12.2022 15:22:09

// Decoder; welcher die Digits umwandelt zum 7SegmentDisplay
// Sagt, welche Segments bleuchtet sein müssen; um die entsprechende Zahl
darzustellen
module hex7seg (
    input wire [3:0]x,
    output reg [6:0]a_to_g
    );

    always@(*)
    begin
        case(x)
        // Note: 0 equals illuminated part of the 7-seg-display; 1 equals LED
OFF!!
        0:  a_to_g = 7'b0000001;
        1:  a_to_g = 7'b1001111;
        2:  a_to_g = 7'b0010010;
        3:  a_to_g = 7'b0000110;
        4:  a_to_g = 7'b1001100;
        5:  a_to_g = 7'b0100100;
        6:  a_to_g = 7'b0100000;
        7:  a_to_g = 7'b0001111;
        8:  a_to_g = 7'b0000000;
        9:  a_to_g = 7'b0000100;
        'hA: a_to_g= 7'b0001000;
        'hB: a_to_g= 7'b1100000;
        'hC: a_to_g= 7'b1110010;
```

```
        'hD: a_to_g= 7'b1000010;
        'hE: a_to_g= 7'b0110000;
        'hF: a_to_g= 7'b0111000;
        default: a_to_g = 7'b0000001; // default case: display "0"
        endcase
    end
endmodule
```

Nun haben wir einen 7-Bit-Output, der kontrolliert, welche LEDs des Displays leuchten sollen, je nachdem, was der Wert des Input-Digits ist.

Jetzt haben wir im Wesentlichen alle benötigten Module zusammen und können die Module zu einem «Ganzen» zusammenfügen.

g. Main Module

In einem ersten Schritt müssen wir zuerst alle Inputs -und Outputs definieren:

```
`timescale 1ns / 1ps
// Create Date: 04.12.2022 11:40:23

// main-module; regelt die Steuerung mit den push-Buttons, etc...
module main_module(
    input [7:0] a, input [7:0] b, input pb, input pb2, input pb3, input pb4,
input pb5, input clk_in,
    output [15:0] s, output wire [6:0] a_to_g, output wire [3:0] anodes,
output wire dp
    );
```

Danach definieren wir alle «Variablen», die wir für die «Zwischenspeicherung» von Resultaten benötigen:

```
    wire clk_out;
    wire [8:0] outAdd; // für Zwischenspeicherung des Resultats des 8BitAdders
    wire [8:0] outSub; // für Zwischenspeicherung des Resultats des
8BitSubtrahierers
    wire [3:0] temp; //Zwischenspeichern der Ausgabe des Digits, welches
gerade an der Reihe ist.
    wire [1:0] refreshCount;

    reg [15:0] stemp; // Zwischenspeichern des Resultats für die LED's
    // Digits speichern die Ausgabe für die einzelnen Digits des 7SegDisplays
    reg [3:0] digit1;
    reg [3:0] digit2;
    reg [3:0] digit3;
    reg [3:0] digit4;
```

Danach muss man die Flip-Flops initialisieren, welche den Status der Buttons speichern (ob der Pushbutton gedrückt ist oder nicht). Da wir insgesamt 5 Buttons haben, brauchen wir auch 5 Flip-Flops. Ausserdem lassen wir unseren Adder und unseren Subtractor, die wir auf Gate-Level implementiert haben, das Resultat der Switches ausrechnen.

```
    // Für Speichern der Button-States:
    SlowClk4Hz u1(clk_in, clk_slow);
    DFlipFlop d1(clk_slow, pb, Q1);
```

```
    DFlipFlop d2(clk_slow, pb2, Q2);
    DFlipFlop d3(clk_slow, pb3, Q3);
    DFlipFlop d4(clk_slow, pb4, Q4);
    DFlipFlop d5(clk_slow, pb5, Q5);

    EightBitAdder add(a, b, outAdd);
    EightBitSubtractor sub(a,b,outSub);
```

Im folgenden Block wird entschieden, abhängig davon, welcher Button gerade gedrückt ist; welche Operation durchgeführt wird und dementsprechend, was der Wert der 4 Digits des 7-seg-Displays sein soll.

```
    always @ (Q1 or Q2 or Q3 or Q4 or Q5) begin
        case ({Q1, Q2, Q3, Q4, Q5})
            5'b10000:
                // Addition; Beim Drücken des linken Buttons
                begin
                    stemp[15:0] = 0; // Zur Sicherheit; damit keine Werte z.B.
noch von der Multiplikation übrig bleiben; könnte man auch weglassen
                    stemp = outAdd;
                    digit1 = 4'b0000;
                    digit2 = stemp[8];
                    digit3 = stemp[7:4];
                    digit4 = stemp[3:0];
                end
            5'b01000:
                // Subtraktion; beim Drücken vom rechten Button
                begin
                    stemp[15:0] = 0;
                    stemp = outSub;
                    digit1 = 4'b0000;
                    digit2 = stemp[8];
                    digit3 = stemp[7:4];
                    digit4 = stemp[3:0];
                end
            5'b00100:
                // Multiplikation; Beim Drücken des oberen Buttons
                begin
                    stemp = a*b;
                    digit1 = stemp[15:12];
                    digit2 = stemp[11:8];
                    digit3 = stemp[7:4];
                    digit4 = stemp[3:0];
                end
            5'b00010:
                // XOR; Beim Drücken des mittleren Buttons
                begin
                    stemp = a^b; // Das Resultat ist  hier wieder 16 Bit gross
(a und b werden als 16 Bit Zahl interpretiert)
                    // Aber da beim Lesen von a und b als 16 Bit Zahl die
vorderen 16 Bit als 0 gelesen werden und
                    // 0 XOR 0 wieder 0 gibt, ist das Resultat trotzdem auch
für 8 Bit korrekt
                    digit1 = stemp[15:12];
                    digit2 = stemp[11:8];
                    digit3 = stemp[7:4];
                    digit4 = stemp[3:0];
                end
```

```
            5'b00001:
                // NAND; Beim Drücken des unteren Buttons
                begin
                    stemp = ~(a&b);
                    stemp[15:8] = 0;
                    digit1 = stemp[15:12];
                    digit2 = stemp[11:8];
                    digit3 = stemp[7:4];
                    digit4 = stemp[3:0];
                end
            default:
                // "Default case"; Falls kein Button gedrückt wird
                begin
                    stemp[15:0] = 0;
                    digit1 = a[7:4];
                    digit2 = a[3:0];
                    digit3 = b[7:4];
                    digit4 = b[3:0];
                end
        endcase
    end

    assign s = stemp;
```

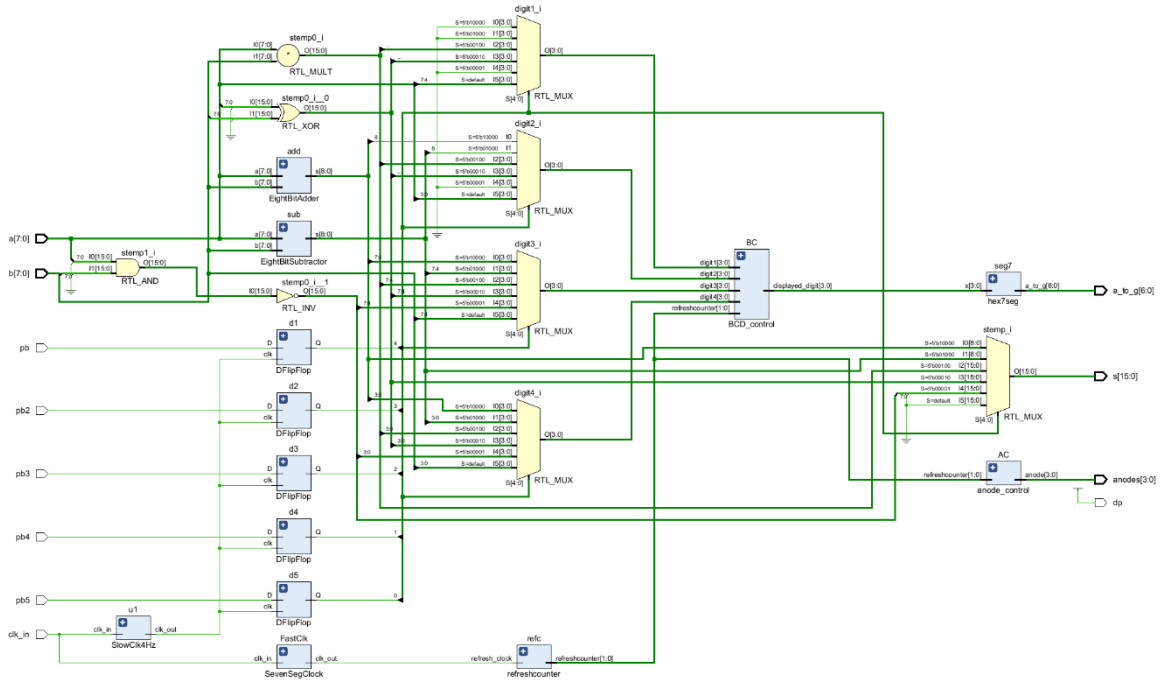
Die Operationen MULT, XOR, NAND wurden hier nicht explizit von uns selbst berechnet, sondern das FPGA übernimmt implizit die Berechnung dieser Werte. Die Implementation dieser zusätzlichen Funktionen auf Gate-Level wäre auch möglich, hätte aber den Rahmen unserer Arbeit gesprengt. Da wir diese Funktionen aber trotzdem bieten möchten, haben wir das FPGA die «Arbeit» übernehmen lassen.

Nun folgt noch die «Verdrahtung» unserer restlichen Module, welche für das 7-Seg-Display benötigt werden:

```
    // Für 7-segment Display:
    assign dp = 1; // decimal point off
    SevenSegClock FastClk(clk_in, refresh_clock); // Generiert einen Clock;
schnell genug um das menschliche Auge zu täuschen
    // aber langsam genug; dass die Ausgabe nicht fehlerhaft ist
    refreshcounter refc(refresh_clock, refreshCount); // Generiert
RefreshCounter für BCD & Anode Control
    anode_control AC(refreshCount, anodes);
    BCD_control BC(digit1, digit2, digit3, digit4, refreshCount, temp);
    hex7seg seg7(temp, a_to_g); // Decoded die Ausgabe fürs 7SegDisplay

endmodule
```

Der komplette Schaltplan für das Projekt sieht jetzt folgendermassen aus:

Hinweis: Die Schaltpläne kann man sich automatisch von Vivado generieren lassen in der «schematic» Ansicht.



h. Pin-Mapping / Constraint-File

Nun sind wir fast am Schluss angelangt. Jetzt fehlt nur noch das Pin-Mapping. Wir müssen jetzt, entsprechend dem Schaltplan unseres Boards, (dieser kann je nach Board variieren), die Pins zu unseren In- und Outputs mappen. Das Pin-Layout kann man dem User-manual des jeweiligen Boards entnehmen.

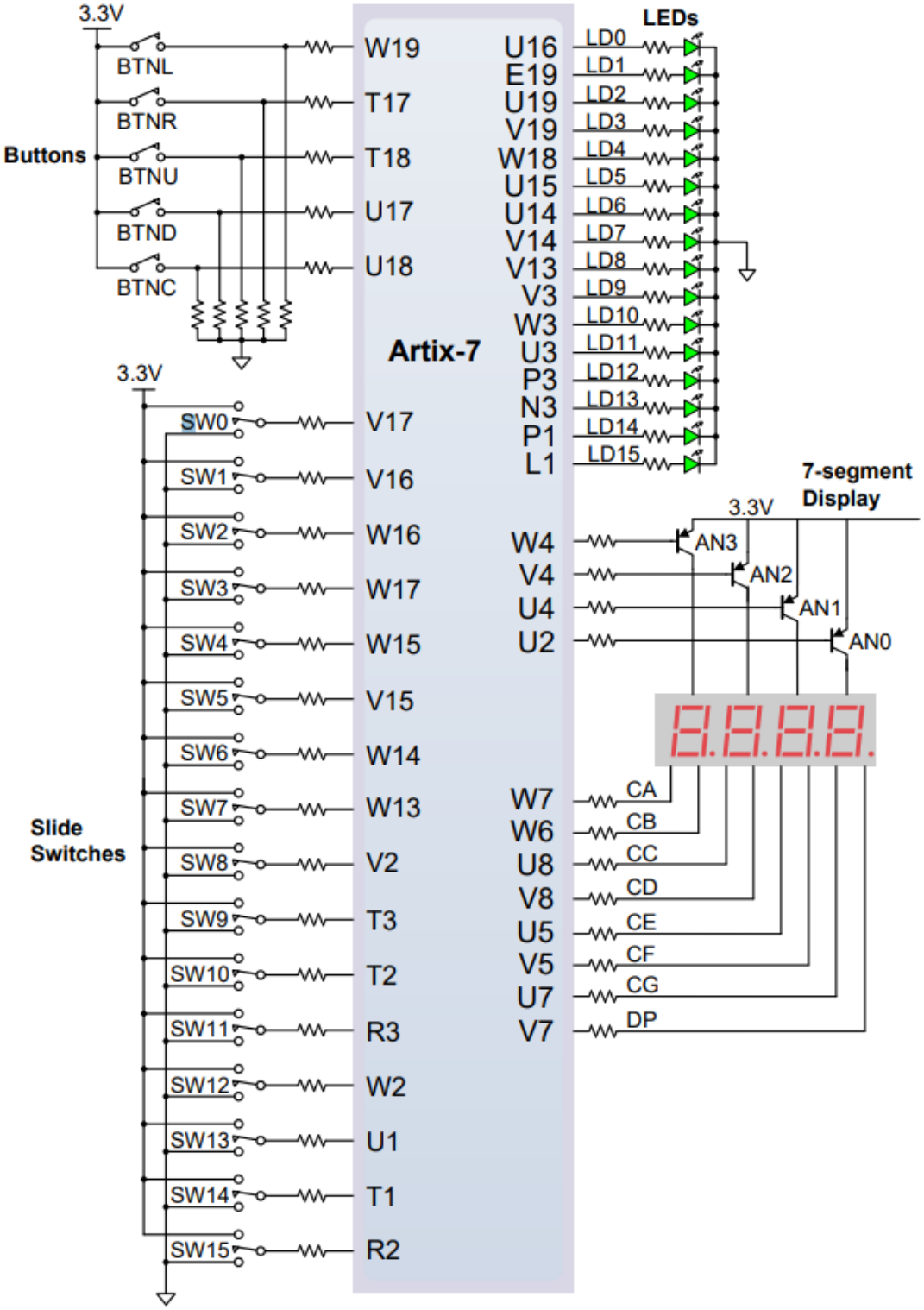


Figure 16. General purpose I/O devices on the Basys 3.

Hinweis: Das Pin-Mapping wäre in der Vivado-Software über ein entsprechendes Interface möglich; da wir aber die Erfahrung gemacht haben, dass die automatische Generierung des Constraint-Files nicht immer reibungslos funktioniert, empfehlen wir das Schreiben des Constraint-Files von Hand.

```
# Clock Signal
set_property PACKAGE_PIN W5 [get_ports clk_in]
set_property IOSTANDARD LVCMOS33 [get_ports clk_in]

# Push Button für Addition (links)
set_property PACKAGE_PIN W19 [get_ports pb]
set_property IOSTANDARD LVCMOS33 [get_ports pb]

# Push Button 2 für Subtraktion (rechts)
set_property PACKAGE_PIN T17 [get_ports pb2]
set_property IOSTANDARD LVCMOS33 [get_ports pb2]

# Push Button 3 für Multiplikation (oben)
set_property PACKAGE_PIN T18 [get_ports pb3]
set_property IOSTANDARD LVCMOS33 [get_ports pb3]

# Push Button 4 für XOR (Mitte)
set_property PACKAGE_PIN U18 [get_ports pb4]
set_property IOSTANDARD LVCMOS33 [get_ports pb4]

# Push Button 5 für NAND (unten)
set_property PACKAGE_PIN U17 [get_ports pb5]
set_property IOSTANDARD LVCMOS33 [get_ports pb5]

# Switches
set_property PACKAGE_PIN V2 [get_ports {a[0]}]
set_property PACKAGE_PIN T3 [get_ports {a[1]}]
set_property PACKAGE_PIN T2 [get_ports {a[2]}]
set_property PACKAGE_PIN R3 [get_ports {a[3]}]
set_property PACKAGE_PIN W2 [get_ports {a[4]}]
set_property PACKAGE_PIN U1 [get_ports {a[5]}]
set_property PACKAGE_PIN T1 [get_ports {a[6]}]
set_property PACKAGE_PIN R2 [get_ports {a[7]}]
set_property PACKAGE_PIN V17 [get_ports {b[0]}]
set_property PACKAGE_PIN V16 [get_ports {b[1]}]
set_property PACKAGE_PIN W16 [get_ports {b[2]}]
set_property PACKAGE_PIN W17 [get_ports {b[3]}]
set_property PACKAGE_PIN W15 [get_ports {b[4]}]
set_property PACKAGE_PIN V15 [get_ports {b[5]}]
set_property PACKAGE_PIN W14 [get_ports {b[6]}]
set_property PACKAGE_PIN W13 [get_ports {b[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
```

```
#LED's
set_property PACKAGE_PIN U16 [get_ports {s[0]}]
set_property PACKAGE_PIN E19 [get_ports {s[1]}]
set_property PACKAGE_PIN U19 [get_ports {s[2]}]
set_property PACKAGE_PIN V19 [get_ports {s[3]}]
set_property PACKAGE_PIN W18 [get_ports {s[4]}]
set_property PACKAGE_PIN U15 [get_ports {s[5]}]
set_property PACKAGE_PIN U14 [get_ports {s[6]}]
set_property PACKAGE_PIN V14 [get_ports {s[7]}]
set_property PACKAGE_PIN V13 [get_ports {s[8]}]
set_property PACKAGE_PIN V3 [get_ports {s[9]}]
set_property PACKAGE_PIN W3 [get_ports {s[10]}]
set_property PACKAGE_PIN U3 [get_ports {s[11]}]
set_property PACKAGE_PIN P3 [get_ports {s[12]}]
set_property PACKAGE_PIN N3 [get_ports {s[13]}]
set_property PACKAGE_PIN P1 [get_ports {s[14]}]
set_property PACKAGE_PIN L1 [get_ports {s[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[0]}]

#7-segment-display cathodes a_to_g und dp
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a_to_g[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports dp]
set_property PACKAGE_PIN V7 [get_ports dp]
set_property PACKAGE_PIN W7 [get_ports {a_to_g[6]}]
set_property PACKAGE_PIN W6 [get_ports {a_to_g[5]}]
set_property PACKAGE_PIN U8 [get_ports {a_to_g[4]}]
set_property PACKAGE_PIN V8 [get_ports {a_to_g[3]}]
set_property PACKAGE_PIN U5 [get_ports {a_to_g[2]}]
set_property PACKAGE_PIN V5 [get_ports {a_to_g[1]}]
set_property PACKAGE_PIN U7 [get_ports {a_to_g[0]}]

#7-segment-display anodes
set_property PACKAGE_PIN U2 [get_ports {anodes[0]}]
set_property PACKAGE_PIN U4 [get_ports {anodes[1]}]
set_property PACKAGE_PIN V4 [get_ports {anodes[2]}]
set_property PACKAGE_PIN W4 [get_ports {anodes[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anodes[3]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {anodes[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anodes[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anodes[0]}]
```