DI - Hausarbeit

Fehlersichere Übertragung und Speicherung

Eric Antosch

2020

Inhaltsverzeichnis

Abbildungsverzeichnis

List of Listings

Kapitel A

Plausibilitätsprüfung

A.1 Aufgabe

Aufgabe 1.0

In dem ersten Teilbereich der Hausarbeit soll das von einem externen Sensor erfasste 10 Bit breite Datenpacket auf hinreichende Abtastung mittels einer Plausibilitätsprüfung überprüft werden. Dabei soll das Signal G_X überprüft und dann mit einem Signal G_X_OK dargestellt werden, dass das Signal zur Weiterverarbeitung übertragen werden kann.

A.2 Blockschaltbild

Wir wollen nun zunächst das Blockschaltbild für unser Vorhaben erstellen, sodass wir bei Beschreibung des Systems in VHDL einen besseren Überblick über alle Signale und Komponenten haben.

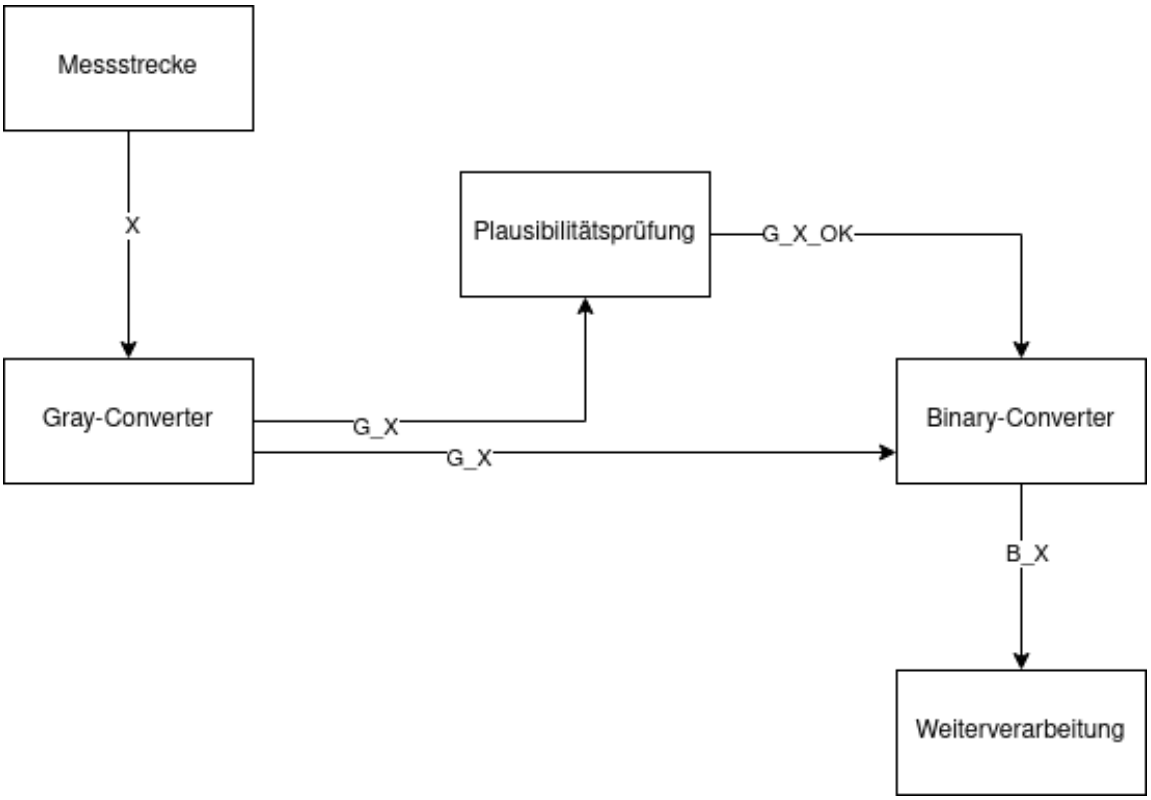

Abbildung A.1: Blockschaltbild der Aufgabe A zur Plausibilitätsprüfung

A.3 Lösungsidee

Wir wollen uns zuerst einmal anschauen, was der Gray-Code überhaupt ist, um eine Idee dafür zu bekommen, mit welchen Methoden die Aufgabe bewältigen können.

Hamming-Distanz Während bei dem in der Digitaltechnik sehr verbreitetem Binärcode die sogenannte Hamming-Distanz unterschiedlich groß sein kann, ist der Gray-Code so konzipiert, dass diese immer nur genau 1 beträgt. Die Hamming-Distanz beschreibt dabei die Distanz zwischen zwei aufeinanderfolgenden Zahlenwerten charakterisiert als die Differenz in den Ziffern der beiden

Zahlen. Die Zahl $n = 3$ und zu der Zahl $n = 4$ haben die binären Darstellungen $n_B = 11_{(2)}$ und $n_B + 1 = 100_{(2)}$. Die Hamming-Distanz ist hier also 3, da sich drei Stellen der Zahl zu der nächsten ändern. Für die Messung von Messstrecken findet sich so keine einfache Methode, die Richtigkeit bzw. Plausibilität der Werte zu testen. Im Gegensatz dazu findet sich $n_G = 010_{(2G)}$ und $n_G + 1 = 110_{(2G)}$ mit einer Hamming-Distanz von 1. Hier können wir einfach überprüfen, ob der Wert, der als nächstes eingelesen wird, sich in der Hamming-Distanz um 1 von dem vorherigen verändert hat. Wenn nicht, so stimmt die Messung nicht vollständig oder die Auflösung ist nicht hoch genug. Besonders bei Messungen von Werten, die keine allzu starken Schwankungen erlauben, ist diese Art der Plausibilitätsprüfung sehr sinnvoll.

Konkrete Idee Wie oben erwähnt, bietet es sich bei der Aufgabe an, mithilfe der Eigenschaften des Gray-Codes eine Überprüfung der Plausibilität durchzuführen. Dafür wollen nehmen wir unser Eingangssignal G_X und speichern dies zunächst auf ein Signal buf, um es später mit dem im nächsten Zyklus eingelesenem Eingangssignal zu vergleichen. Basierend auf dem Ergebnis der Hamming-Distanz-Berechnung setzen wir G_X_OK also entweder auf True oder False. Das eingelesene Signal G_X wird das neue buf und der Prozess wiederholt sich.

A.4 Beschreibung in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity plausible is
  port (
    G_X    : in bit_vector(9 downto 0);
    G_X_OK : out bit
  );
end plausible;

architecture arch of plausible is

  signal buf, plausible : bit_vector(9 downto 0);

begin

  -- Evaluates the current signal with the buffered previous data
  eval : process begin
    wait on G_X;
    plausible <= buf xor G_X;
    buf       <= G_X;
  end process eval;

  -- Counts the differences between the evaluated signal and the buffered one
  count : process
    variable sum : integer := 0;
  begin
    wait on plausible;
    sum := 0;
    for i in 0 to 9 loop
      if plausible(i) = '1' then
        sum := sum + 1;
      end if;
    end loop;
    if sum <= 1 then
      G_X_OK <= '1';
    else
      G_X_OK <= '0';
    end if;
  end process;

end architecture;
```

Listing 1: Der Code für die Plausibilitätsprüfung

Wir überprüfen zunächst in *eval* die Hammingdistanz zwischen *buf* und *G_X* und speichern dann in unserem Buffer den derzeitigen Wert von *G_X*. In *count* benutzen wir eine Integervariable, um die 1 in dem Signal *plausible* zu zählen, welche das Ergebnis der Hammingdistanz enthält. Wenn dann sum über den Wert 1 geht, dann ist das Signal nach unserem Verständnis nicht mehr plausibel.

A.5 Simulation der Ergebnisse

Da wir unsere Schaltung in ModelSim simulieren, verwenden wir eine .do-Datei, um den Verlauf der Schaltung zu simulieren. Wir wollen dabei sowohl den Fall erproben, dass die Daten richtig sind, als auch, dass die Daten zuweit voneinander entfernt sind, um plausibel zu sein.

Abbildung A.2: Die Simulation der Ergebnisse

```
vsim work.plausible
restart
view wave
radix bin
add wave *


force G_X "1010101010"
run 20ns

force G_X "1111111111"
run 20ns

force G_X "1111111110"
run 20ns
```

Listing 2: .do-File (Testbench) zur Simulation der Ergebnisse in A.2

A.6 Fazit

Wir können anhand unserer Ergebnisse einige Dinge feststellen. Zum einen ist es wirklich erstaunlich, wie die Eigenschaften von speziellen Codes die Verarbeitung und Implementierung von bestimmten Lösungen vereinfacht. Die Logik, um eine solche Plausibilitätsprüfung durchzuführen, wenn die Eingangsdaten zum Beispiel weiterhin als Binärcode kodiert sind, wäre um einiges komplizierter und außerdem nicht wirklich effizient oder wirtschaftlich. Es bietet sich also stark an, die Möglichkeit von den verschiedenen Codes aus der Digitaltechnik einzustudieren. Es ist außerdem ersichtlich, dass sich eine Darstellung von Problemstellungen nach dem gelernten Schema anbietet. Es lässt einen stark über die einzelnen Komponenten und die Zusammenarbeit auf einem tatsächlichen FPGA-Board nachdenken.

Kapitel B

Berechnung der Regelabweichung

B.1 Aufgabe

Aufgabe 1.0

In der nächsten Aufgabe soll es um die Berechnung der Regelabweichung von zwei Signalen mithilfe eines Carry-Lookahead-Addierers gehen. Dabei ist unser auf Plausibilität geprüfte Signal B_X mit einem Sollwert B_W zu vergleichen und die Differenz auf einer bidirektionalen 10-bit breiten Busleitung auszugeben. Ebenfalls sollen die dedizierten Signale EXAKT und ZU_KLEIN seperat je nach Ergebnis ausgegeben werden.

B.2 Blockschaltbild

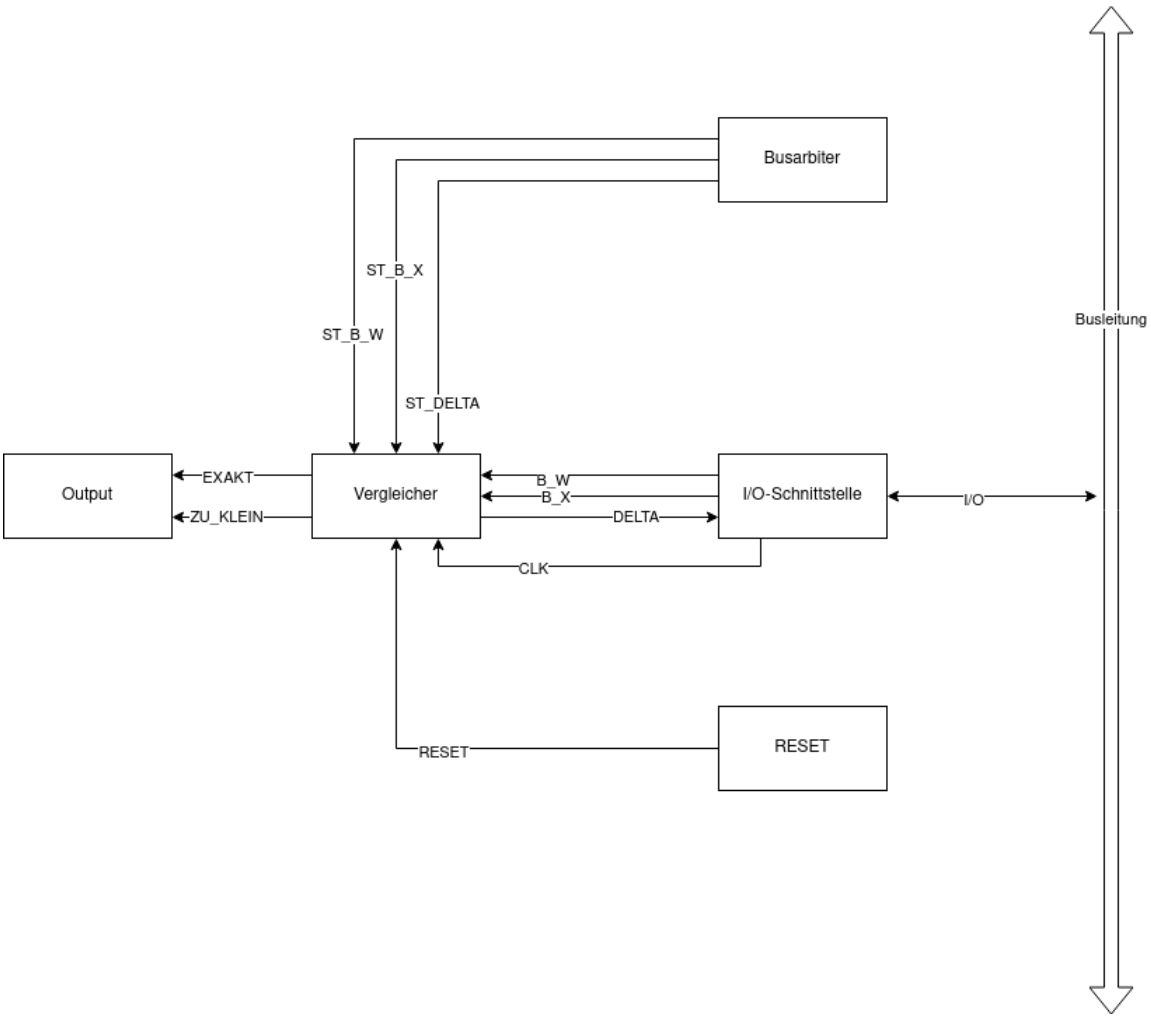

Abbildung B.1: Blockschaltbild für die Regelabweichung

B.3 Automatengraph

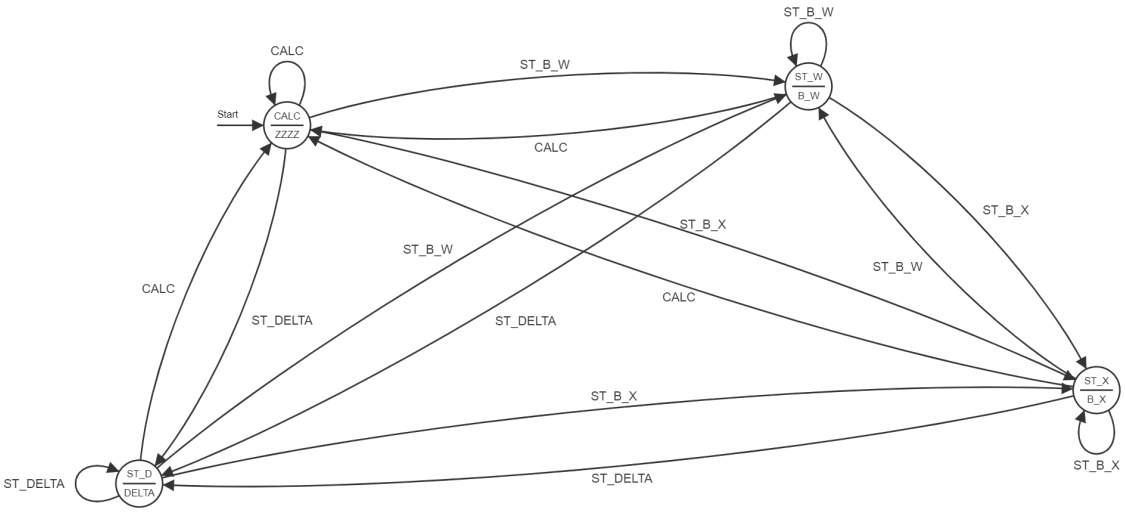

Abbildung B.2: Der Automatengraph für die Berechnung der Regelabweichung

Zustand/Signal	Erklärung
CALC	Der CALC-Zustand stellt die durchgehende Berechnung des Ergebnisses dar. Der zugehörige Übergang $CALC \in \Sigma$ stellt die jeweilige Beendigung des anderes Zustands dar. Im Falle von ST_DELTA z.B. das Ende von zwei Systemtakten.
ST_B_X	In diesem Zustand wird der neue Ist-Wert von Bus eingelesen. Dieses wird von dem Steuersignal ST_B_X eingeleitet.
ST_B_W	In diesem Zustand wird der neue Soll-Wert von Bus eingelesen. Dieses wird von dem Steuersignal ST_B_W eingeleitet.
ST_DELTA	In diesem Zustand wird das Ergebnis auf dem Bus ausgegeben. Dabei gibt das Signal ST_DELTA das Signal, den Bus zwei Systemtakte lang mit dem Ergebnis zu beschreiben.

Tabelle B.1: Erklärung der einzelnen Zustände und Übergänge

B.4 Lösungsidee

Wir wollen uns nun einmal anschauen, mit welchen Mitteln wir unser Ziel erreichen können.

B.4.1 Carry-Lookahead-Addierer

Bei dem Carry-Lookahead-Addierer, den wir für unsere Berechnung der Regelabweichung verwenden wollen, handelt sich um eine Weiterentwicklung des Carry-Ripple-Addierers. Hier werden also nicht einfach nur mehrere Volladdierer miteinander so verschaltet, dass mehrere Bit breite Signale miteinander addiert werden können, sondern es gibt ein komplizierteres System, um die Übertrage der Volladdierer zu bestimmen. Fundamental für diese Überlegung sind die Konzepte von Generate, Propagate und Absorb. [Fit21]

Generate Bei Generate handelt sich um ein logisches Prädikat, welches wir im Folgenden durch $G(A, B)$ bezeichnen möchten mit $G(A, B) = A \cdot B$, wobei $\cdot$ das logische Und darstellt. Wir sagen also, dass zwei Ziffern genau dann generieren, wenn die beiden Ziffern immer einen Übertrag bilden und dieser nicht abhängig ist, ob ein weiterer Übertrag auf die Ziffern hinzuaddiert werden.

Propagate Bei Propagate handelt es sich ebenfalls um ein logisches Prädikat, welches wir im Folgenden mit $P(A, B) = A + B$ bezeichnen, wobei $+$ das logische Oder darstellt. Wir sagen also, dass zwei Ziffern genau dann propagieren, wenn die beiden Ziffern nur dann einen Übertrag bilden, wenn ein weiterer Übertrag von der Berechnung vorher kommt.

Absorb Wir wollen im folgenden Absorb als denjenigen Zustand definieren, der einen Übertrag von der vorherigen Berechnung absorbieren würde, dabei allerdings keinen Übertrag generiert. Mit $A(A, B) = \not(A + B)$ definieren wir einen Zustand, der das Gegenteil von Propagate darstellt und nur dann wahr ist, wenn beide Ziffern (im Falle einer binären Addition) 0 sind.

B.4.2 Zweier Komplement

Wenn wir die Differenz von zwei binären Zahlen berechnen wollen, müssen wir eine der beiden, da unser Carry-Lookahead-Addierer eigentlich nur Addition beherrscht, in ein anderes Format umwandeln, um eine Differenz berechnen zu können. Wir entscheiden uns hier für das Zweier Komplement, um aus einer der beiden Signalen, die eine binäre Zahl darstellen, eine negative Zahl zu machen, und diese dann zu addieren.

Bei der Umwandlung gehen wir dabei nach folgendem Schema vor:

1. Wir invertieren alle weiteren Bits, alle 1 werden zu 0 und umgekehrt.
2. Das Most-Significant-Bit stellt im zweier Komplement das Vorzeichen dar.
3. Zum Schluss addieren wir auf die Zahl eine 1, unsere Zahl ist damit umgewandelt.

Umgekehrt setzen wir die Schritte einfach erneut ein, wir invertieren also alle Stellen und addieren dann eine 1 auf die derzeitige Zahl. Wir erhalten genau die Zahl, mit der wir begonnen haben.

B.4.3 Taktsynchronität

Wir müssen uns nun noch mit der Anforderung an unsere Lösung beschäftigen, taktsynchron und in Harmonie mit dem Busarbiter zu funktionieren. Dafür wollen wir in unserer Beschreibung mit VHDL eine Top-Level-Komponente einrichten, die sich um genau diese Anforderungen kümmert. Wir nennen diese comparator. Sie wird sich mithilfe des Moore-Automatengraphs durch die einzelnen Zustände des Automaten arbeiten und die entsprechenden Daten aus dem bidirektionalen Bus einlesen, weiterverarbeiten und dann auf dem gleichen Bus ausgeben.

B.5 Beschreibung in VHDL

B.5.1 Volladdierer

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5
6
7  entity fulladder is
8    port (
9      a    : in bit;
10     b    : in bit;
11     cin  : in bit;
12     S    : out std_logic;
13     cout : out bit
14   );
15 end entity fulladder;
16
17 architecture behav of fulladder is
18
19 begin
20   S    <= To_STDULOGIC(a xor b xor cin);
21   cout <= (a and b) or (cin and a) or (cin and b);
22 end architecture;
```

Listing 3: Der Volladdierer als Baustein des Carry-Lookahead-Addierers

Der Volladdierer ist schon aus der Vorlesung bekannt und wurde in vergangenen Praktika bereits verwendet. Dabei ist allerdings zu bemerken, dass hier bereits Verzögerungen und eine Umwandlung von Bit zu STD_LOGIC vorgenommen wurde. Der Hintergrund ist die anschließende Umwandlung des Signals mittels der signed und abs Funktionen der Standardbibliothek IEEE.

B.5.2 Carry-Lookahead-Addierer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity carrylookahead is
  port (
    add1   : in bit_vector(9 downto 0);
    add2   : in bit_vector(9 downto 0);
    sign   : out std_logic;
    result : out std_logic_vector(8 downto 0)
  );
end entity;

architecture behav of carrylookahead is
  component fulladder
    port (
      a    : in bit;
      b    : in bit;
      cin  : in bit;
      S    : out std_logic;
      cout : out bit
    );
  end component;
  signal carry : bit_vector(10 downto 0);
  signal sum   : std_logic_vector(9 downto 0);
  signal gen   : bit_vector(10 downto 0);
  signal prop  : bit_vector(10 downto 0);
begin
  clahgen : for i in 0 to 9 generate
    fulladder_inst : fulladder
    port map(
      a    => add1(i),
      b    => add2(i),
      cin  => carry(i),
      S    => sum(i),
      cout => open
    );
  end generate;

  genprop : for j in 0 to 9 generate
    gen(j)       <= add1(j) and add2(j);
    prop(j)      <= add1(j) or add2(j);
    carry(j + 1) <= gen(j) or (prop(j) and carry(j));
  end generate;

  carry(0) <= '1';
  sign     <= sum(9);
  result   <= std_logic_vector(abs(signed(sum(result'range))));
end architecture;
```

Listing 4: Der Carry-Lookahead-Addierer, dabei wird das Ergebnis automatisch in das gewünschte Format geschrieben

Die Logik des Carry-Lookahead-Addierers, die wir in B.4.1 bereits beschrieben haben, wird nun mittels einiger For-Generate-Statements verwirklicht. Es werden also entsprechend viele Volladdierer, sowie die Logik für Generate, Propagate und Carry-Over generiert und diese dann zum Schluss mittels $std\_logic\_vector(abs(signed(sum(result'range))))$ in den entsprechenden Betrag umgewandelt wird. Das Vorzeichen wird in *sign* gespeichert. [vhd] [nan]

B.5.3 Substractor

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
entity subtractor is
  port (
    B_X   : in std_logic_vector(9 downto 0);
    B_W   : in std_logic_vector(9 downto 0);
    DELTA : out std_logic_vector(9 downto 0)
  );
end entity subtractor;

architecture behav of subtractor is
  component carrylookahead is
    port (
      add1   : in bit_vector(9 downto 0);
      add2   : in bit_vector(9 downto 0);
      sign   : out std_logic;
      result : out std_logic_vector(8 downto 0)
    );
  end component;

  signal TC_B_W   : bit_vector(9 downto 0);
  signal sA       : bit_vector(9 downto 0);
  signal sR       : std_logic_vector(8 downto 0);
  signal signedBX : std_logic;

begin

  cla : carrylookahead
  port map(
    add1   => sA,
    add2   => TC_B_W,
    sign   => signedBX,
    result => sR
  );
  sA     <= to_bitvector(B_X);
  TC_B_W <= not(to_bitvector(B_W));
  DELTA  <= signedBX & sR;
end architecture;
```

Listing 5: Der Subtractor, der die Verarbeitung der Werte aus dem Vergleicher mit dem Carry-Lookahead-Addierer übernimmt

Der Substractor behandelt nun implizit die Umrechnung des Sollwerts in das Zweierkomplement, um eine Subtraktion der beide Signale durchführen zu können. Dabei wird das Signal TC_B_W zunächst einfach konvertiert. Durch das Cin_0 des Carry-Lookahead-Addierers, welches wir in der Datei auf 1 gesetzt haben, ist das Addieren der 1 zur Umwandlung in das Zweierkomplement auch gegeben. Zum Schluss wird dann *DELTA* durch eine Konkatenation, die in VHDL über & passiert, des Vorzeichens und des Betrags aus dem Ergebnis des Carry-Lookahead-Addierers gebildet. [Ger16]

B.5.4 Vergleicher

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5
6  entity comparator is
7    port (
8      clk      : in std_logic;
9      reset    : in std_logic;
10     io       : inout std_logic_vector(9 downto 0) := "ZZZZZZZZZZ";
11     ST_B_X   : in bit;
12     ST_B_W   : in bit;
13     ST_DELTA : in bit;
14     EXAKT    : out bit;
15     ZU_KLEIN : out bit
16   );
17 end entity comparator;
18
19 architecture rtl of comparator is
20   component subtractor is
21     port (
22       B_X   : in std_logic_vector(9 downto 0);
23       B_W   : in std_logic_vector(9 downto 0);
24       DELTA : out std_logic_vector(9 downto 0)
25     );
26   end component;
27
28   signal result : std_logic_vector(9 downto 0);
29   type statetype is (reading_BX, reading_BW, calculating, output);
30   signal X, W       : std_logic_vector(9 downto 0) := "0000000000";
31   signal state      : statetype                    := calculating;
32   signal next_state : statetype                    := calculating;
33 begin
34
35   sub : subtractor
36   port map(
37     B_X   => X,
38     B_W   => W,
39     DELTA => result
40   );
41
42   WR : process (state, clk, ST_DELTA)
43   begin
44     if state = output and clk'event and clk = '1' then
45       io <= result;
46     elsif state = calculating and clk'event and clk = '1' then
47       io <= (others => 'Z');
48     end if;
49     if result(9) = '1' then
50       ZU_KLEIN <= '1';
51     elsif X = W then
52       EXAKT <= '1';
53     else
```

```
54        ZU_KLEIN <= '0';
55        EXAKT    <= '0';
56      end if;
57    end process;
58
59    RD : process (io, ST_B_X, clk)
60    begin
61      if state = reading_BX and clk'event and clk = '1' then
62        X <= io;
63      elsif state = reading_BW and clk'event and clk = '1' then
64        W <= io;
65      end if;
66    end process;
67
68    SW : process (reset, clk)
69    begin
70      if clk'event and clk = '1' and reset = '1' then
71        state <= calculating;
72      elsif clk'event then
73        state <= next_state;
74      end if;
75    end process;
76
77    SN : process (ST_B_W, ST_B_X, ST_DELTA, clk)
78      variable counter : integer := 0;
79    begin
80      case state is
81        when calculating =>
82          io <= "ZZZZZZZZZZ";
83          if ST_B_X = '1' then
84            next_state <= reading_BX;
85          elsif ST_B_W = '1' then
86            next_state <= reading_BW;
87          elsif ST_DELTA = '1' then
88            next_state <= output;
89          end if;
90        when reading_BX =>
91          if ST_B_X = '0' then
92            next_state <= calculating;
93          elsif ST_B_W = '1' then
94            next_state <= reading_BW;
95          elsif ST_DELTA = '1' then
96            next_state <= output;
97          end if;
98        when reading_BW =>
99          if ST_B_W = '0' then
100           next_state <= calculating;
101         elsif ST_B_X = '1' then
102           next_state <= reading_BX;
103         elsif ST_DELTA = '1' then
104           next_state <= output;
105         end if;
106       when output =>
107         if clk'event and clk = '1' then
```

```
108             counter := counter + 1;
109             if clk'event and clk = '1' and counter = 2 then
110               next_state <= calculating;
111               counter := 0;
112             end if;
113           end if;
114       end case;
115     end process;
116 end architecture;
```

Listing 6: Der Hauptprozess, der den Zugriff auf den Bus (io) übernimmt.

In diesem Prozess behandeln wir, wie wir die Taktsynchronität, wir in B.4.3 schon angesprochen haben, umsetzen wollen. Wir verwenden den in B.3 erstellten Graphen, um im Prozess *SN* die Bedingungen für das Ändern des Zustandes wählen. Über das Signal *io*, was den Bus darstellt, werden in den richtigen Momenten die Werte eingelesen. Der Prozess *SW* wechselt vom *state* in den *next_ state*, und zwar immer auf dem Ändern des Clocksignals. Hier wird außerdem noch der Reset eingelesen und verarbeitet. Der Prozess *WR* schreibt, sollte der Zustand auf *output* sein, das Ergebnis des *subtractors* auf den Bus, ansonsten wird dieser auf hochohmig gelegt. Der letzte Prozess *RD* liest bei gegebenen Zustand den des Busses aus.

B.6 Simulation der Ergebnisse

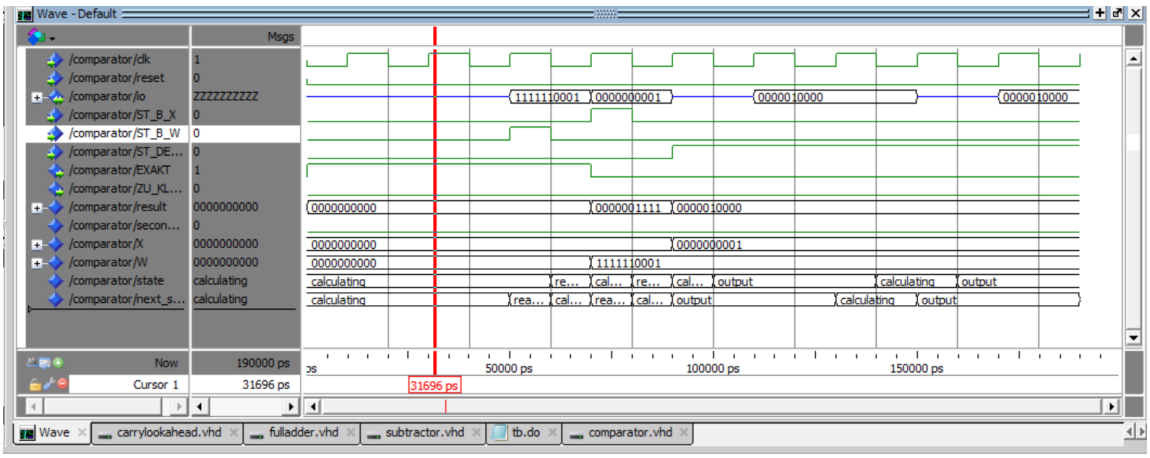

Abbildung B.3: Simulation der Beschreibung in VHDL

Wir erkennen auf der Abbildung, dass wir eine relativ gute Lösung finden konnten, die mit unseren Wünschen und Anforderungen an unser System übereinstimmen. Wir lesen zunächst den Sollwert B_W nach dem Steuersignal ST_B_W ein, gefolgt von dem Ist-Wert B_X über das ST_B_X-Signal. Es wird über unser Substractor-Component, der den Carry-Lookahead-Addierer enthält, ein Ergebnis in unseren result-Buffer geschrieben, der dann zwei Systemtakte lang auf der bidirektionalen Busleitung io ausgegeben wird. Diese wird zwischen den Operationen, und so auch nach der Ausgabe des Ergebnisses, wieder hochohmig (dargestellt durch die blaue Linie). Auch unser Ergebnis von $16_{(10)} = 0000010000_{(2)}$, welches Sie aus der Distanz des Sollwerts $B\_W = -15_{(10)} = 1111110001_2$ zu dem Ist-Wert $B\_X = 1_{(10)} = 0000000001_{(2)}$ ergibt.

```
vsim work.comparator
restart
view wave
radix bin
add wave *

force clk 0,1 10ns -r 20ns

force reset 0
force ST_B_W 0
force ST_B_X 0
force ST_DELTA 0
run 50ns

force ST_B_W 1
force io "1111110001"
run 10ns

force ST_B_W 0
run 10ns


force ST_B_X 1
force io "0000000001"
run 10ns

force ST_B_X 0
run 10ns

noforce io
force ST_B_X 0
force ST_DELTA 1
run 100ns
```

Listing 7: .do-File (Testbench) für die Simulation der Ergebnisse in B.3

B.6.1 Kritischer Pfad

Der kritische Pfad eines Systems beschreibt denjenigen Pfad eines Signals, welcher die längste Verzögerung von Eingang zu Ausgang bietet. So können beispielsweise große Zahlen in einem Carry-Ripple-Addierer zu größerer Berechnungszeit führen. Der kritische Pfad führt also durch alle Überträge hindurch. Wir haben bei uns, durch den Carry-Lookahead-Addierer nur noch die Verzögerung der Berechnung der Summe im Volladdierer selbst. Wir simulieren hierbei Verzögerung über *after x ns*. Die Umwandlung im CLA und die Berechnung des Carryoverbits durch Generate und Propagate werden auch mit entsprechenden Werten versehen. Allerdings ist der kritische Pfad, den es zu bestimmen gilt, lediglich von der Berechnung selbst abhängig. Wir bekommen mit den Verzögerungen bei unserer Testberechnung eine Verzögerung von $12ns$, den kritischen Pfad konnten wir mit einer pauschalen Verarbeitungszeit der übrigen Schaltungselemente mit $52ns$ mit $\Delta t_g = \Delta t_b + \Delta t_v = 12ns + 40ns = 52ns$ bestimmen. Eine maximale Taktfrequenz, ergibt sich dann mit $f = \frac{1}{\Delta t_g} = \frac{1}{52ns} = 19,23MHz$ bestimmen.

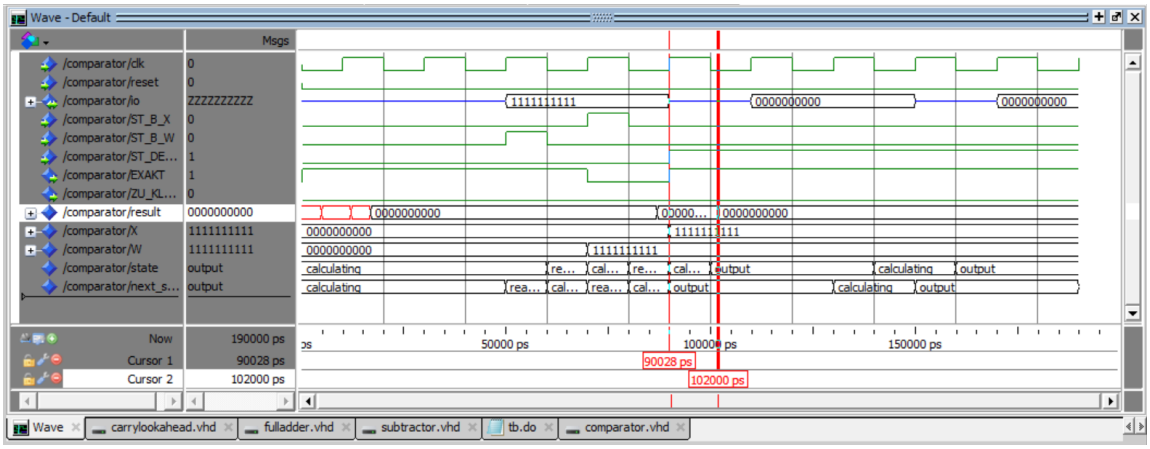

Abbildung B.4: Die Berechnung der Berechnungsverzögerung mithilfe von ModelSim

B.7 Fazit

Was sich bei dieser Aufgabe als ein extrem prägnantes und sehr hilfreiches Analysewerkzeug entpuppt hat, ist die Darstellung der Zustände innerhalb eines Moore-Automaten, da sich darüber analog die Verarbeitung der Steuersignale, der Inputs und der Outputs ergeben hat. Eine Simulation eines solchen Automaten, was wir bereits im dritten Praktikum erfolgreich getan haben und uns auch in der Vorlesung angeschaut haben, hat hier die Vorgehensweise extrem linear und einfach gestaltet. Eine Umwandlung der Werte in das Zweierkomplement, eine Erweiterung des bereits verwendeten Carry-Ripple-Addierers zum Carry-Lookahead-Addierers und eine sinnvolle Verwendung bereits gelernter Konzepte hat den Rest der Aufgaben ausgemacht. Ich konnte hier allerdings Lösungen aus der Literatur, der Vorlesung und des Internets verwenden, um für alle weiteren Probleme eine gute Herangehensweise zu finden.

Kapitel C

Literaturverzeichnis

[Fit21] Prof. Dr. Robert Fitz. *Digitaltechnik Skript*. HAW, 2021.

[Ger16] Winfried Gerke. *Digitaltechnik: Grundlagen*. Springer, 2016.

[nan] nandland. `nandland.com`. Zugriff (15.07.2021).

[vhd] vhdlwhiz. `https://vhdlwhiz.com/`. Zugriff (15.07.2021).