

Laborbericht zum Digitallabor an der Hochschule Karlsruhe

Michael Nestor und Robin Fritz

23. September 2018

Zusammenfassung

Die uns aus der Vorlesung *Technische Informatik* bekannten Techniken werden im Digitallabor begleitend an der konkreten Hardware eingeübt. Dies dient zur Konkretisierung und Vertiefung des Stoffes sowie zur persönlichen Erfolgskontrolle. Ziel ist es dabei die verschiedenen logischen Grundsaltungen, sowie die Zahlendarstellung in verschiedenen Zahlensystemen zu verstehen und anzuwenden. Das Erlernen des Umgangs mit einem Mikrocontroller-Entwicklungssystem und verstehen des Aufbaus sowie die Bedienung typischer Peripherieschaltungen sind weitere Lernziele des Labors. Die Versuche enthalten Übungen zur Zahlendarstellung, zu Mikrocontrollern und zur Verwendung von parallelen Peripherieschaltkreisen sowie Zähler/Zeitgebern. Dieses Dokument gibt einen ausführlichen Überblick von den von uns ausgeführten Versuchen.

Inhaltsverzeichnis

Zusammenfassung	i
I Dokumentation der Laborversuche	1
1 Versuch I	2
1.1 Aufgabe 1	2
1.1.1 Quellcode für ein Or mit zwei Eingängen	2
1.2 Aufgabe 2	3
1.2.1 Quellcode für einen Halbaddierer	3
1.3 Aufgabe 3	4
1.3.1 Quellcode für einen Volladdierer	4
1.4 Aufgabe 4	4
1.4.1 Quellcode für einen Serienaddierer	5
1.4.2 Testbench für den Serienaddierer	5
2 Versuch II	8
2.1 Aufgabe 1	8
2.1.1 Quellcode und Simulation des 4-Bit Binärzähler	9
2.2 Aufgabe 2	10
2.2.1 Anmerkungen zu Aufgabe 2	10
2.3 Aufgabe 3	10
2.3.1 Quellcode, Testbench und Simulation zu Aufgabe 3	10
2.4 Aufgabe 4	11
2.4.1 Quellcode, Testbench und Simulation zu Aufgabe 4	11
2.5 Aufgabe 5	11
2.5.1 Quellcode, Testbench und Simulation zu Aufgabe 5	11
2.6 Aufgabe 6	11
2.6.1 Quellcode, Testbench und Simulation zu Aufgabe 6	12
2.7 Aufgabe 7	12
2.7.1 Quellcode, Testbench und Simulation zu Aufgabe 7	12
3 Versuch III	13
3.1 Aufgabe 1	13
3.1.1 Quellcode, Listing und Screenshots der Aufgabe 1	13
3.2 Aufgabe 2	17
3.2.1 Quellcode, Listing und Screenshots der Aufgabe 2	17
3.3 Aufgabe 3	21
3.3.1 Anmerkungen zu Aufgabe 3	22

3.3.2	Quellcode und Screenshots der Aufgabe 3	22
3.4	Aufgabe 4	24
3.4.1	Quellcode und Screenshot zu Aufgabe 4	24
3.5	Aufgabe 5	26
3.5.1	Quellcode und Screenshot zu Aufgabe 5	27
4	Versuch IV	28
4.1	Aufgabe 1	28
4.1.1	Anmerkungen zu Aufgabe 1	28
4.2	Aufgabe 2	28
4.2.1	Anmerkungen zu Aufgabe 2	28
4.3	Aufgabe 3	28
4.3.1	Anmerkungen zu Aufgabe 3	28
4.4	Aufgabe 4	28
4.4.1	Anmerkungen zu Aufgabe 4	28
5	Versuch V	29
5.1	Aufgabe 1	29
5.1.1	Anmerkungen zu Aufgabe 1	29
5.2	Aufgabe 2	29
5.2.1	Anmerkungen zu Aufgabe 2	29
5.3	Aufgabe 3	29
5.3.1	Anmerkungen zu Aufgabe 3	29
II	Anhang	30
A	Aufgabenblatt 1	31
B	Aufgabenblatt 2	34
C	Aufgabenblatt 3	43
D	Aufgabenblatt 4	46
E	Aufgabenblatt 5	49

Quellcodeverzeichnis

1.1	or mit zwei Eingängen	2
1.2	nebenläufiges VHDL für einen Halbaddierer	3
1.3	Volladdierer in ein strukturellem VHDL	4
1.4	Serienaddierer in VHDL	5
1.5	Testbench des Serienaddierers	5
2.1	4-Bit Binärzähler	9
3.1	Definitio der Konstanten in Vision Assembler source	13
3.2	Listning von work1.a66	15
3.3	Definition der 16bit Variablen	17
3.4	Listning von work2.a66	19
3.5	Quellcode für Rechenoperationen	22
3.6	Unterprogramm der Aufgabe 4	24

Abbildungsverzeichnis

1.1	Funktionstabelle für einen Halbaddierer	3
2.1	Ampelsteuerung	8
2.2	Simulation des 4-Bit Binärzähler	10
2.3	Funktionstabelle der Ampelphasen	11

Teil I

Dokumentation der Laborversuche

Labortermin 1

Versuch I

Im ersten Versuch lag der Schwerpunkt auf kombinatorischem und strukturellem VHDL im GAL Baustein. Zur Vorbereitung machten wir uns mit der Oberfläche des ISP-Levler-Programm vertraut, übersetzten eine Funktionstabelle in VHDL und beschäftigten uns mit dem Aufbau von Halb-, Voll- und Serienaddierer.

Ziel des Versuch sollten erste Erfahrungen mit der Sprache VHDL und dem hierarchischem Design sein. Hierbei wurde ein kleiner programmierbaren Baustein, das GAL eingesetzt. Mit Hilfe des Versuches erfuhren wir, dass die Kombination aus HDL und programmierbarer Hardware schnell zu funktionierenden Schaltungen führt, und auch ziemlich flexibel bei Änderungen ist.

Für die Designerstellung wurde das Programm *Classic* und zur Simulation der *VHDL Functional Simulator* verwendet.

1.1 Aufgabe 1

Die Aufgabe beinhaltete das Schreiben eines nebenläufigen VHDL Modells für ein ODER Gatter mit zwei Eingängen welches wir mit einer Gleichung zur Beschreibung der Funktionalität umsetzten. Simuliert wurde das Design durch die direkte Eingabe der Testvektoren. Nach der erfolgreiche Simulation programierten wir den GAL und testeten ihn mit zwei Schaltern.

1.1.1 Quellcode für ein Or mit zwei Eingängen

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity ODER2_ent is
7     PORT (a, b : IN std_logic;
8           y  : OUT std_logic);
9
10
11 end;
12
13 architecture ODER2_arch of ODER2_ent is
```



```

14 begin
15 y <= (a or b);
16
17 end ODER2_arch;

```

Listing 1.1: or mit zwei Eingängen

1.2 Aufgabe 2

Im folgenden sollte eine Funktionstabelle als nebenläufiges VHDL Modell für einen Halbaddierer umgesetzt werden. Die Korrektheit des Designs wurde mittel einer Simulation überprüft.

D	C	B	A	Y
0	0	0	0	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	0	1
1	1	1	1	1
Alle anderen Kombinationen von D,C,B,A				0

Abbildung 1.1: Funktionstabelle für einen Halbaddierer

1.2.1 Quellcode für einen Halbaddierer

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity halbadd_ent is
7     port ( B, A : IN std_logic;
8           S, C : OUT std_logic);
9
10 end;
11
12 architecture halbadd_arch of halbadd_ent is
13 begin
14     with std_logic_vector'(B, A) select
15         S <= '1' when "01",
16         '1' when "10",
17         '0' when others;
18         C <= (B and A);
19 end halbadd_arch;

```

Listing 1.2: nebenläufiges VHDL für einen Halbaddierer

1.3 Aufgabe 3

Aus zwei Instanzen des Halbaddierers und einem ODER Gatter erstellen wir einen Volladdierer. Gefordert war dabei rein strukturelles VHDL, das nur die Verknüpfung der Komponenten beschreibt.

1.3.1 Quellcode für einen Volladdierer

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity VA_ent is
7     port ( ai, bi, ci : in std_logic;
8           sumi, cout : out std_logic );
9
10 end;
11
12 architecture VA_arch of VA_ent is
13     signal N1, N2, N3 : std_logic;
14
15
16     component halbadd_ent
17     port ( B, A : in std_logic;
18           S, C : out std_logic );
19 end component;
20
21 component ODER2_ent
22     port ( a, b : in std_logic;
23           y : out std_logic );
24 end component;
25
26
27 begin
28     I1 : ODER2_ent
29     Port Map ( a=>N2, b=>N3, y=>cout );
30     I2 : halbadd_ent
31     Port Map ( B => bi, A=>ai, S=>N1, C=>N2 );
32     I3 : halbadd_ent
33     Port Map ( B => ci, A=>N1, S=> sumi, C=>N3 );
34 end;
```

Listing 1.3: Volladierer in ein strukturellem VHDL

1.4 Aufgabe 4

Wir verwendeten in dieser Aufgabe den darvor erstellten Volladierer um mit zwei Instanzen einen Serienaddierer zu erstellen welcher zwei Zahlen zu je zwei Bit zusammenzufügt. Die beiden Eingangszahlen a und b sind je ein 2 Bit breiter Vektor, für die Summe wurde ein 3 Bit breiter Vektor verwendet.// Für den Serienaddierer haben wir des weiteren eine Testbench erstellt und ihn mit dieser Simuliert. Nach der Simulation nutzten wir den GAL Baustein und verbanden die Eingänge mit den Schaltern, die Ausgänge mit der BCD -> 7-Segment - Anzeige. Anschlussbuchse „C/C – D2“ an der 7-Segment-Anzeige mussten an GND angeschlossen werden.

1.4.1 Quellcode für einen Serienaddierer

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity Va2_ent is
7     port ( b, a : in std_logic_vector (1 downto 0);
8           sum      : out std_logic_vector (2 downto 0));
9
10 end;
11
12 architecture Va2_arch of Va2_ent is
13     signal N1: std_logic;
14
15     component VA_ent
16     port ( ai, bi, ci : in std_logic;
17           sumi, cout : out std_logic);
18 end component;
19
20 begin
21     VA0 : VA_ent
22     Port Map ( bi=>b(0), ai=>a(0), ci=>'0', sumi=>sum(0), cout=>
23               N1 );
24     VA1 : VA_ent
25     Port Map ( bi=>b(1), ai=>a(1), ci=>N1, sumi=>sum(1), cout=>sum
26               (2) );
27 end;
```

Listing 1.4: Serienaddierer in VHDL

1.4.2 Testbench für den Serienaddierer

```
1
2 -- VHDL Test Bench Created from source file VA_ent.vhd -- 04/16/18
3 -- 17:32:01
4 --
5 -- Notes:
6 -- 1) This testbench template has been automatically generated
7 -- using types
8 -- std_logic and std_logic_vector for the ports of the unit under
9 -- test.
10 -- Lattice recommends that these types always be used for the top-
11 -- level
12 -- I/O of a design in order to guarantee that the testbench will
13 -- bind
14 -- correctly to the timing (post-route) simulation model.
15 -- 2) To use this template as your testbench, change the filename
16 -- to any
17 -- name of your choice with the extension .vhd, and use the "source
18 -- ->import"
19 -- menu in the ispLEVER Project Navigator to import the testbench.
20 -- Then edit the user defined section below, adding code to
21 -- generate the
22 -- stimulus for your design.
23 --
24 LIBRARY ieee;
25 LIBRARY generics;
26 USE ieee.std_logic_1164.ALL;
27 USE ieee.numeric_std.ALL;
28 USE generics.components.ALL;
```

```

21
22 ENTITY testbench IS
23 END testbench;
24
25 ARCHITECTURE behavior OF testbench IS
26
27     COMPONENT VA_ent
28     PORT(
29         ai : IN std_logic;
30         bi : IN std_logic;
31         ci : IN std_logic;
32         sumi : OUT std_logic;
33         cout : OUT std_logic
34     );
35     END COMPONENT;
36
37     SIGNAL ai : std_logic;
38     SIGNAL bi : std_logic;
39     SIGNAL ci : std_logic;
40     SIGNAL sumi : std_logic;
41     SIGNAL cout : std_logic;
42
43 BEGIN
44
45     uut: VA_ent PORT MAP(
46         ai => ai,
47         bi => bi,
48         ci => ci,
49         sumi => sumi,
50         cout => cout
51     );
52
53
54 -- *** Test Bench - User Defined Section ***
55     tb : PROCESS
56         x_tb : PROCESS
57     BEGIN
58
59         x <= "00";
60
61         if x = "10" then
62             x <= "11";
63         elsif x = "11" then
64             x <= "00";
65         elsif x = "00" then
66             x <= "01";
67         elsif x = "01" then
68             x <= "10";
69         end if;
70         WAIT FOR 100 ns;
71     end PROCESS;
72
73
74
75     y_tb : PROCESS
76     BEGIN
77
78         y <= "00";
79
80         if y = "00" then
81             y <= "01";
82         elsif y = "01" then

```

```

83 y <= "10";
84 elsif y = "10" then
85 y <= "11";
86 elsif y = "11" then
87 y <= "00";
88 end if;
89 WAIT FOR 100 ns;
90
91 end PROCESS;
92
93
94
95     tb : PROCESS
96     BEGIN
97         wait; -- will wait forever
98     END PROCESS;
99 -- *** End Test Bench - User Defined Section ***
100
101 END;

```

Listing 1.5: Testbench des Serienaddierers

Labortermin 2

Versuch II

In diesem Versuch nutzten wir die nächste Technologiestufe, die CPLDs. Durch Verwendung eines MACH Bausteines konnten wir gleichzeitig auf die In-System-Programmierung zurückgreifen. Der Baustein konnte d.h. über den JTAG Anschluss direkt auf der Platine programmiert werden. Dabei war es zielführend die Beschreibung von sequentiellen Schaltungen mit VHDL und die Simulation dieser Schaltungen kennen lernen. Natürlich sollten wir auch unsere Kenntnisse über das Erstellen hierarchischer Designs und den Umgang mit einer Testbench vertiefen.

Für die Durchführung des Versuchs benötigten wir ein ispMach-Board und ein I/O-Board mit welchen wir eine Ampelsteuerung, die aus vier Komponenten gemäß Abbildung 2.1 aufgebaut wurde. Als Datentyp wurde durchgehend die *std_logic* verwendet.

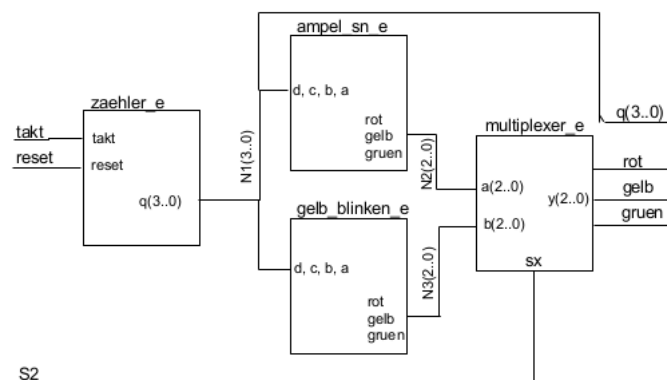


Abbildung 2.1: Ampelsteuerung

2.1 Aufgabe 1

Wir erstellten einen 4-Bit Binärzähler mit asynchronem Reset in VHDL den Vorgaben entsprechend. Für die sequenzielle Schaltung verwendeten wir einen

Prozess welcher entsprechend getriggert wurde. Im Prozess wird die asynchrone Bedingung $intReset = \hat{1}$ abgefragt, falls diese wahr ist wird der Vektor *temp* auf 0 gesetzt, andernfalls wird dieser um 1 hochgezählt. Außerhalb des Prozesses wird nebenläufig *temp* dem Ausgang zugewiesen. Im folgenden wurde der Zähler mittels Simulation getestet.

Wenn der Reset zubeginn auf 0 steht, besitzt der Ausgang einen unbestimmten Zustand und kann d.h. nicht zählen.

2.1.1 Quellcode und Simulation des 4-Bit Binärzähler

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity zaehler_e is
7
8     port(intTakt, intReset : in std_logic;
9          q : out std_logic_vector(3 downto 0));
10
11 end zaehler_e;
12
13 architecture zaehler_a of zaehler_e is
14
15     signal tmp: std_logic_vector(3 downto 0);
16
17     begin
18     process (intTakt, intReset)
19
20     begin
21         if (intReset = '1') then
22             tmp <= "0000";
23         elsif (intTakt'event and intTakt = '1') then
24             tmp <= tmp + 1;
25         end if;
26     end process;
27
28     q <= tmp;
29
30 end zaehler_a;
```

Listing 2.1: 4-Bit Binärzähler

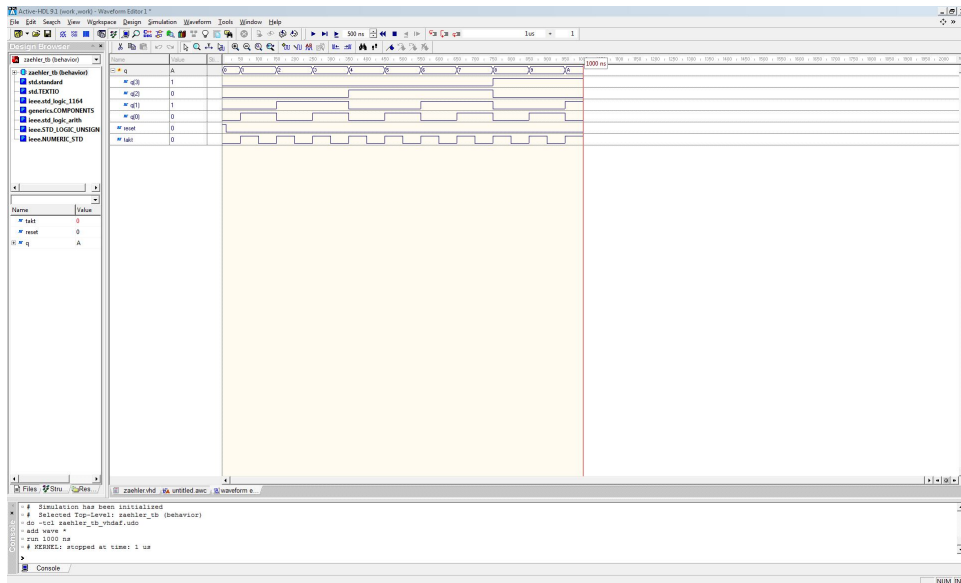


Abbildung 2.2: Simulation des 4-Bit Binärzähler

2.2 Aufgabe 2

Für den 4-Bit Binärzähler haben wir eine Testbench erstellt. Die Testbench enthält für die Signale jeweils einen Prozess in welchem es Initialisiert wird. Die Dauer des Signals wird dabei durch *wait for* festgelegt danach wird das Signal gekippt usw. durch einen zweiten Prozess für wiederholende Signale verwenden wir einen *loop* wieder unter zuhelfenahme des *wait for* Befehls. Zum programmieren des CLPD Bauteins mussten noch diverse Einstellungen vorgenommen werden unter anderem die Zuweisung der Pins mittels des *Constraint Editor*.

2.2.1 Anmerkungen zu Aufgabe 2

!!!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!!!

2.3 Aufgabe 3

In dieser Aufgabe sollten wir ein VHDL Modell für einen rein kombinatorischen Ampel Steuerungsblock entwerfen. Die in Funktionstabelle in Abbildung 2.3 gab uns die 16 Ampelphasen vor welche implementiert wurden.

Weiter sollten wir diese mit einer Testbench simulieren wobei die 16 Eingangswerte mit Hilfe einer FOR-Schleife erzeugt wurden.

2.3.1 Quelcode, Testbensh und Simulation zu Aufgabe 3

!!!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!!!

Eingänge des Schaltnetzes				Ausgänge des Schaltnetzes			
D	C	B	A	GRÜN	GELB	ROT	
0	0	0	0	1	0	0	GRÜN-Phase
0	0	0	1	1	0	0	
0	0	1	0	1	0	0	
0	0	1	1	1	0	0	
0	1	0	0	0	1	0	GELB-Phase
0	1	0	1	0	1	0	
0	1	1	0	0	0	1	ROT-Phase
0	1	1	1	0	0	1	
1	0	0	0	0	0	1	
1	0	0	1	0	0	1	
1	0	1	0	0	0	1	
1	0	1	1	0	0	1	
1	1	0	0	0	0	1	
1	1	0	1	0	0	1	
1	1	1	0	0	1	1	ROT-GELB-Phase
1	1	1	1	0	1	1	

Abbildung 2.3: Funktionstabelle der Ampelphasen

2.4 Aufgabe 4

Den AmpelSteuerungsblock aus der vorigen Aufgabe fügten wir in ein Strukturmodell ein, das den Zähler und das Ampel Schaltnetz instanziert. Neben den Ports aus Aufgabe 1 kamen folgende Ports hinzu: *rot*, *gelb*, *gruen* : *out std_logic*

Simuliert wurde der Code mit der Testbench aus Aufgabe 2 im Anschluss programmierte wir die Hardware wofür wir alle Signals aus der Liste einem entsprechenden den Pins zuwiesen und testeten die Schaltung.

2.4.1 Quelcode, Testbench und Simulation zu Aufgabe 4

!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!

2.5 Aufgabe 5

Wir erstellten einen zusätzlichen Ampel Steuerungsblock als getrenntes VHDL-Modul. Es liefert ebenfalls die Signale *rot* / *gelb* / *gruen*. Die Ampelphasen wurden dabei so angepasst ,dass die gelbe LED zwei Takte leuchtet und zwei Takte aus ist, die anderen blieben aus. Simmuliert wurde diese Schaltung mit der Testbench aus Aufgabe 3.

2.5.1 Quelcode, Testbench und Simulation zu Aufgabe 5

!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!

2.6 Aufgabe 6

Ziel dieser Aufgabe war es ein sequentielles VHDL Modell eines 2:1 Multiplexers zu entwerfen für zwei je drei Bit breite *std_logic* Vektoren A und B, die

abhängig von einem Signal Select auf den drei Bit breiten Ausgangsvektor Y durchgeschaltet werden.

Wir testeten das Modell via Simulation.

2.6.1 Quellcode, Testbench und Simulation zu Aufgabe 6

!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!

2.7 Aufgabe 7

Als letztes fügten wir alle Komponenten zusammen. Neben den Ports aus Aufgabe 4 fügten wir noch den Port: *S2 : in std_logic* hinzu. Zur Simulation nutzten wir die Testbench aus Aufgabe 2 welche wir um *S2* erweiterten. Im Anschluss wurde die Hardware programmiert.

2.7.1 Quellcode, Testbench und Simulation zu Aufgabe 7

!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlen!!!!!!!!!!!!!!!!!!!!

Labortermin 3

Versuch III

Der dritte Labortermin hatte den Schwerpunkt erste Schritte mit maschinen-naher C166 Programmierung zumachen. Ziel des Versuchs sollte es sein die internen Abläufe in einem typischen Prozessor zu verstehen. Wir erarbeiten, was die Unterschiede zwischen Konstanten und Variablen sind, und wie der Prozessor mit den internen Registern arbeitet. Weiterhin beschäftigten wir uns mit dem Übergabemechanismus von Parameteradressen an Unterprogramme und der Verarbeitung der Parameter per indirekter Adressierung.

Verwendet wurde die Keil Software mit der vorgegebenen Vorlage. Zum Debuggen benutzten wir den Simulator.

3.1 Aufgabe 1

Wir erstellten Konstanten mit folgenden Namen und Werten:

op1 = 30000, op2 = 5000, op3 = 40000, op4 = 4999, op5 = -30000

Im Hauptprogramm wurden die Konstanten den Registern R1 bis R5 zugewiesen. Im Anschluss assemblierten und banden wir das Programm um es zu debuggen. Mit Hilfe des Register-Window konnte nun überprüft werden, ob die in den Konstanten gespeicherten Werte in die Register geladen wurden, was hier der Fall war. Die Konstanten sind direkt im kompilierten Programmcode enthalten wie man im Disassembly-Fenster sehen konnte.

3.1.1 Quellcode, Listing und Screenshots der Aufgabe 1

```
1 ;$NONSEGMENTED                ; Nur Tiny Model möglich
2 $MODV2                        ; Code fuer 161/163/164/165/167
3 $NOLIST
4 $INCLUDE (REG167.INC)        ; include CPU definition file
5 $LIST
6 ;
7 ; TITLE <Mein Text>
8 ;
9 ; Prozessor Definitionen
10 ;
```

```

11 ;
12 ; Hier folgen Ihre Konstanten Definitionen
13 ;
14 Ora      EQU 0 ;Portbit f r Ora
15 op1      EQU 30000;
16 op2      EQU 5000;
17 op3      EQU 40000;
18 op4      EQU 4999;
19 op5      EQU -30000;
20
21 Datenspeicher SECTION Data
22 ;
23 ; Hier folgen Ihre Speicher Vereinbarungen
24 ;
25 Mein_Speicher      DB      12      ;Byte Initialisiert
26 Mein_Speicher1     DW      65      ;Word Initialisiert
27 Mein_Speicher2     DD      4711
28 Mein_Text          DB      'ASCII Text '
29
30 Datenspeicher ENDS
31 ;
32 ;
33 Programme Section Code
34 ASSUME  DPP3:SYSTEM
35 ASSUME  DPP0:Datenspeicher
36 ;
37 ; Hier stehen Ihre Unterprogramme
38 ;
39 ;
40 ;PInit  Initialisiert den Port Px
41 ;=====
42 ; I: nix
43 ; O: nix
44 ; Z: Register , die zerstr t werden.
45 ;
46 MyUp1 PROC NEAR
47
48     Ret
49 MyUp1 EndP;
50
51 ;
52 ; Hier startet das Hauptprogramm
53 ;
54 A_RESET PROC TASK STARTUP INTNO RESET = 0
55     mov r0 ,BUSCON0
56     and r0,#0xFF00
57     mov BUSCON0,r0
58     mov dpp0,#PAG Datenspeicher      ;Page von Datenspeicher => DPP0
59     mov  KSCCFG,#0x0003;              ;Module Enable
60     mov  R0,KSCCFG      ;wieder ruecklesen
61     JMP Main
62 A_RESET EndP
63
64 MAIN  Proc
65 ;
66 ;Hier geht Ihr Hauptprogramm los
67 ;
68
69     MOV R1, #op1
70     MOV R2, #op2
71     MOV R3, #op3
72     MOV R4, #op4

```

```

73 MOV R5, #op5
74
75 Call MyUp1
76
77 Forever:
78
79
80 jmp Forever
81 ;
82 ; Notstop
83 ;
84 StopJetzt: JMP StopJetzt
85
86 Main ENDP
87
88 Programme ENDS
89 ;
90
91 END

```

Listing 3.1: Definitio der Konstanten in Vision Assembler source

```

1 A166 MACRO ASSEMBLER WORK1
2
3
4 MACRO ASSEMBLER A166 V5.30
5 OBJECT MODULE PLACED IN work1.OBJ
6 ASSEMBLER INVOKED BY: C:\Keil\C166\BIN\A166.EXE work1.a66 MODV2
7 SEGMENTED MODV2 SET(SMALL) DEBUG NOSYMBOLS EP
8
9 LOC OBJ LINE SOURCE
10
11 Tiny Model mglich 1 ;$NONSEGMENTED ; Nur
12 fuer 161/163/164/165/167 2 $MODV2 ; Code
13 598 $LIST
14 599 ;
15 600 ; TITLE <Mein Text>
16 601 ;
17 602 ; Prozessor Definitionen
18 603 ;
19 604 ;
20 ;
21 Definitionen 605 ; Hier folgen Ihre Konstanten
22
23 0000 606 ;
24 Portbit f r Ora 607 Ora EQU 0 ;
25 7530 608 op1 EQU 30000;
26 1388 609 op2 EQU 5000;
27 9C40 610 op3 EQU 40000;
28 1387 611 op4 EQU 4999;
29 FFFFFFFF8AD0 612 op5 EQU -30000;
30 613
31 614 Datenspeicher SECTION Data
32 615
33 ;
34 Vereinbarung 616 ; Hier folgen Ihre Speicher

```

```

31                                     617      ;
32 00000000 0C                                     618      Mein_Speicher      DB
      12      ;Byte Initialisiert
33 00000002 4100                                     619      Mein_Speicher1      DW
      65      ;Word Initialisiert
34 00000004 67120000                                     620      Mein_Speicher2      DD
      4711
35 00000008 41534349                                     621      Mein_Text      DB      '
      ASCII Text '
36 0000000C 49205465
37 00000010 7874                                     621
38                                     622
39 -----                                     623      Datenspeicher      ENDS
40                                     624      ;
41                                     625      ;
42 -----                                     626      Programme      Section Code
43                                     627      ASSUME DPP3:SYSTEM
44                                     628      ASSUME DPP0:Datenspeicher
45                                     629
      ;
46                                     630      ; Hier stehen Ihre Unterprogramme
47                                     631      ;
48                                     632      ;
49                                     633      ; PInit Initialisiert den Port Px
50                                     634      ;=====
51                                     635      ;      I: nix
52                                     636      ;      O: nix
53                                     637      ;      Z: Register , die zerstört
      werden .
54                                     638      ;
55                                     639      MyUp1      PROC NEAR
56                                     640
57 00000000 CB00                                     641      Ret
58                                     642      MyUp1      EndP;
59                                     643
60                                     644
      ;
61                                     645      ; Hier startet das Hauptprogramm
62                                     646      ;
63                                     647      A_RESET PROC TASK STARTUP INTNO
      RESET = 0
64 00000002 F2F00CFF                                     648      mov      r0 ,BUSCON0
65 00000006 66F000FF                                     649      and      r0 ,#0xFF00
66 0000000A F6F00CFF                                     650      mov      BUSCON0,r0
67 0000000E E600???? R                                     651      mov      dpp0,#PAG
      Datenspeicher ;Page von Datenspeicher => DPP0

68 A166 MACRO ASSEMBLER WORK1
      05/25/2018 16:26:17 PAGE      2
69
70 00000012 E60E0300                                     652      mov      KSCCFG,#0x0003;
      ;Module Enable
71 00000016 F2F01CFE                                     653      mov      R0,KSCCFG
      ;wieder ruecklesen
72 0000001A 0D00                                     654      JMP      Main
73                                     655      A_RESET EndP
74                                     656
75                                     657      MAIN      Proc
76                                     658
      ;
77                                     659      ;Hier geht Ihr Hauptprogramm los
78                                     660      ;
79                                     661

```

```

80 0000001C E6F13075      662      MOV R1, #op1
81 00000020 E6F28813      663      MOV R2, #op2
82 00000024 E6F3409C      664      MOV R3, #op3
83 00000028 E6F48713      665      MOV R4, #op4
84 0000002C E6F5D08A      666      MOV R5, #op5
85                                     667
86 00000030 BBE7          668      Call    MyUp1
87                                     669
88                                     670      Forever:
89                                     671
90                                     672
91 00000032 0DFF          673      jmp      Forever
92                                     674      ;
93                                     675      ; Notstop
94                                     676      ;
95 00000034 0DFF          677      StopJetzt: JMP      StopJetzt
96                                     678
97                                     679      Main      ENDP
98                                     680
99 _____ 681      Programme ENDS
100                                     682      ;
101                                     683
102                                     684      END
103
104
105
106 ASSEMBLY COMPLETE. 0 WARNING(S), 0 ERROR(S)

```

Listing 3.2: Listing von work1.a66

!!!!!!!!!!!!!!!!!!!!Screenshot fehlt!!!!!!!!!!!!!!!!!!!!

3.2 Aufgabe 2

Die Werte wurden im Anschluss in 16bit Variablen abgelegt. Dies geschah im Variablenbereich des Programms. Durch eine direkte Adressierung wird der Wert der Variable in das zugewiesene Register geladen, dabei kann man in der Listing-Datei erkennen, dass für den Speicherort ein Offset-Parameter angelegt wurde, dieser wurde auf die entsprechende Startadresse addiert wodurch der Speicherort definiert ist. Im Memory-Fenster konnte man an denn berechneten Adressen die Werte der Variablen ablesen.

3.2.1 Quellcode, Listing und Screenshots der Aufgabe 2

```

1 ;$NONSEGMENTED          ; Nur Tiny Model möglich
2 $MODV2                  ; Code fuer 161/163/164/165/167
3 $NOLIST
4 $INCLUDE (REG167.INC)   ; include CPU definition file
5 $LIST
6 ;
7 ; TITLE <Mein Text>
8 ;
9 ; Prozessor Definitionen
10 ;
11 ;
12 ; Hier folgen Ihre Konstanten Definitionen
13 ;
14 Ora      EQU 0 ;Portbit f r Ora
15
16 Datenspeicher SECTION Data

```

```

17 ;
18 ; Hier folgen Ihre Speicher Vereinbarungen
19 ;
20 Mein_Speicher      DB      12      ;Byte Initialisiert
21 Mein_Speicher1     DW      65      ;Word Initialisiert
22 Mein_Speicher2     DD      4711
23 Mein_Text          DB      'ASCII Text'
24 op1                DW      30000 ;
25 op2                DW      5000  ;
26 op3                DW      40000 ;
27 op4                DW      4999  ;
28 op5                DW      -30000 ;
29
30 Datenspeicher ENDS
31 ;
32 ;
33 Programme Section Code
34 ASSUME  DPP3:SYSTEM
35 ASSUME  DPP0:Datenspeicher
36 ;
37 ; Hier stehen Ihre Unterprogramme
38 ;
39 ;
40 ;PInit  Initialisiert den Port Px
41 ;=====
42 ; I: nix
43 ; O: nix
44 ; Z: Register , die zerstört werden.
45 ;
46 MyUp1 PROC NEAR
47
48     Ret
49 MyUp1 EndP;
50
51 ;
52 ; Hier startet das Hauptprogramm
53 ;
54 A_RESET PROC TASK STARTUP INTNO RESET = 0
55     mov r0 ,BUSCON0
56     and r0 ,#0xFF00
57     mov BUSCON0,r0
58     mov dpp0,#PAG Datenspeicher    ;Page von Datenspeicher => DPP0
59     mov  KSCCFG,#0x0003;           ;Module Enable
60     mov  R0,KSCCFG    ;wieder ruecklesen
61     JMP Main
62 A_RESET EndP
63
64 MAIN  Proc
65 ;
66 ;Hier geht Ihr Hauptprogramm los
67 ;
68
69     MOV R1, op1
70     MOV R2, op2
71     MOV R3, op3
72     MOV R4, op4
73     MOV R5, op5
74     MOV R10, R1
75     ADD R10, R2
76     MOV R11, R1
77     ADD R11, R3
78     MOV R12, R4

```



```

79 SUB R12, R2
80 MOV R13, R1
81 ADD R13, R5
82 MOV R14, R3
83 ADD R14, R5
84
85
86 Call MyUp1
87
88 Forever:
89
90
91 jmp Forever
92 ;
93 ; Notstop
94 ;
95 StopJetzt: JMP StopJetzt
96
97 Main ENDP
98
99 Programme ENDS
100 ;
101
102 END

```

Listing 3.3: Definition der 16bit Variablen

```

1 A166 MACRO ASSEMBLER WORK2
2 05/26/2018 06:40:30 PAGE 1
3
4 MACRO ASSEMBLER A166 V5.30
5 OBJECT MODULE PLACED IN work2.OBJ
6 ASSEMBLER INVOKED BY: C:\Keil\C166\BIN\A166.EXE work2.a66 MODV2
7 SEGMENTED MODV2 SET(SMALL) DEBUG NOSYMBOLS EP
8
9 LOC OBJ LINE SOURCE
10 Tiny Model mglich 1 ;$NONSEGMENTED ; Nur
11 fuer 161/163/164/165/167 2 $MODV2 ; Code
12 598 $LIST
13 599 ;
14 600 ; TITLE <Mein Text>
15 601 ;
16 602 ; Prozessor Definitionen
17 603 ;
18 604 ;
19 ; Hier folgen Ihre Konstanten
20 Definitionen 605
21 0000 606 ;
22 Portbit f r Ora 607 Ora EQU 0 ;
23 608
24 609 Datenspeicher SECTION Data
25 610
26 ; Hier folgen Ihre Speicher
27 Vereinbarungen 611

```

```

26                                     612      ;
27 00000000 0C                                     613      Mein_Speicher      DB
      12      ;Byte Initialisiert
28 00000002 4100                                     614      Mein_Speicher1      DW
      65      ;Word Initialisiert
29 00000004 67120000                                     615      Mein_Speicher2      DD
      4711
30 00000008 41534349                                     616      Mein_Text      DB      '
      ASCII Text '
31 0000000C 49205465
32 00000010 7874                                     616
33 00000012 3075                                     617      op1
      DW      30000      ;
34 00000014 8813                                     618      op2
      DW      5000      ;
35 00000016 409C                                     619      op3
      DW      40000      ;
36 00000018 8713                                     620      op4
      DW      4999      ;
37 0000001A D08A                                     621      op5
      DW      -30000      ;
38                                     622
39 -----                                     623      Datenspeicher      ENDS
40                                     624      ;
41                                     625      ;
42 -----                                     626      Programme      Section Code
43                                     627      ASSUME DPP3:SYSTEM
44                                     628      ASSUME DPP0:Datenspeicher
45                                     629
      ; -----
46                                     630      ; Hier stehen Ihre Unterprogramme
47                                     631      ;
48                                     632      ;
49                                     633      ; PInit Initialisiert den Port Px
50                                     634      ;=====
51                                     635      ;      I: nix
52                                     636      ;      O: nix
53                                     637      ;      Z: Register , die zerstört
      werden .
54                                     638      ;
55                                     639      MyUp1      PROC NEAR
56                                     640
57 00000000 CB00                                     641      Ret
58                                     642      MyUp1      EndP ;
59                                     643
60                                     644
      ; -----
61                                     645      ; Hier startet das Hauptprogramm
62                                     646      ;
63                                     647      A_RESET PROC TASK STARTUP INTNO
      RESET = 0
64 00000002 F2F00CFF                                     648      mov      r0 ,BUSCON0
65 00000006 66F000FF                                     649      and      r0 ,#0xFF00
66 0000000A F6F00CFF                                     650      mov      BUSCON0,r0
67 0000000E E600???? R                                     651      mov      dpp0,#PAG
      Datenspeicher      ;Page von Datenspeicher => DPP0

68 A166 MACRO ASSEMBLER WORK2
      05/26/2018 06:40:30 PAGE      2
69
70 00000012 E60E0300                                     652      mov      KSCCFG,#0x0003 ;
      ;Module Enable

```

```

71 00000016 F2F01CFE          653          mov      R0,KSCCFG
                                ;wieder ruecklesen
72 0000001A 0D00          654          JMP      Main
73                                655          A_RESET EndP
74                                656
75                                657          MAIN      Proc
76                                658
                                ;
77                                659          ;Hier geht Ihr Hauptprogramm los
78                                660          ;
79                                661
80 0000001C F2F1???? R      662          MOV R1, op1
81 00000020 F2F2???? R      663          MOV R2, op2
82 00000024 F2F3???? R      664          MOV R3, op3
83 00000028 F2F4???? R      665          MOV R4, op4
84 0000002C F2F5???? R      666          MOV R5, op5
85 00000030 F0A1          667          MOV R10, R1
86 00000032 00A2          668          ADD      R10, R2
87 00000034 F0B1          669          MOV R11, R1
88 00000036 00B3          670          ADD      R11, R3
89 00000038 F0C4          671          MOV R12, R4
90 0000003A 20C2          672          SUB      R12, R2
91 0000003C F0D1          673          MOV R13, R1
92 0000003E 00D5          674          ADD R13, R5
93 00000040 F0E3          675          MOV R14, R3
94 00000042 00E5          676          ADD R14, R5
95                                677
96                                678
97 00000044 BBDD          679          Call      MyUp1
98                                680
99                                681          Forever :
100                               682
101                               683
102 00000046 0DFF          684          jmp      Forever
103                               685          ;
104                               686          ; Notstop
105                               687          ;
106 00000048 0DFF          688          StopJetzt : JMP      StopJetzt
107                               689
108                               690          Main          ENDP
109                               691
110                               692          Programme      ENDS
111                               693          ;
112                               694
113                               695          END
114
115
116
117 ASSEMBLY COMPLETE.  0 WARNING(S) , 0 ERROR(S)

```

Listing 3.4: Listing von work2.a66

!!!!!!!!!!!!!!!!!!!!Screenshot fehlt!!!!!!!!!!!!!!!!!!!!

3.3 Aufgabe 3

Wir erweitern das Programm aus Aufgabe 2, um folgende Rechenoperationen mit den Ziel-Registern R10 bis R14:

$R10 = op1 + op2$; $R11 = op1 + op3$; $R12 = op4 - op2$; $R13 = op1 + op5$,
 $R14 = op3 + op5$;

Danach debuggen wir im Single Step unser Programm.

3.3.1 Anmerkungen zu Aufgabe 3

Bei R10 $op1 + op2$ entstand bei der Rechnung mit signed ein Overflow, da der zulässige Zahlenbereich überschritten wurde, das MSB wird bei der signed Darstellung als Vorzeichenbit behandelt. Bei $op1 + op2$ wurde dieses Bit auf 1 gesetzt.

Bei R11 $op1 + op3$ wurde der Zahlenbereich von 16bit unsigned überschritten, daher wurde das Carry-Flag gesetzt.

Bei R12 $op4 - op2$ wurde in der signed Darstellung das Ergebnis negativ (Flag N). Da der Subtrahend größer als der Minuend ist, wurde ein Carry-Flag gesetzt.

Bei R13 $op1 + op5$ wurde das Zero-Flag gesetzt, da das Ergebnis 0 ist.

Bei R14 $op3 + op5$ handelte es sich um eine signed Operation. $op3$ war jedoch nicht im definierten signed Bereich, Daher wurde die Overflow-Flag gesetzt. Bei der Addition entstand ein Übertrag, d.h. wurde das Carry-Flag gesetzt.

3.3.2 Quellcode und Screenshots der Aufgabe 3

```
1 ;$NONSEGMENTED ; Nur Tiny Model möglich
2 $MODV2 ; Code fuer 161/163/164/165/167
3 $NOLIST
4 $INCLUDE (REG167.INC) ; include CPU definition file
5 $LIST
6 ;
7 ; TITLE <Mein Text>
8 ;
9 ; Prozessor Definitionen
10 ;
11 ;-----
12 ; Hier folgen Ihre Konstanten Definitionen
13 ;
14 Ora EQU 0 ;Portbit f r Ora
15
16
17 Datenspeicher SECTION Data
18 ;-----
19 ; Hier folgen Ihre Speicher Vereinbarungen
20 ;
21 Mein_Speicher DB 12 ;Byte Initialisiert
22 Mein_Speicher1 DW 65 ;Word Initialisiert
23 Mein_Speicher2 DD 4711
24 Mein_Text DB 'ASCII Text'
25
26 MyVar32W1 DW (120000 AND 0xFFFF) ; unteren 16bit
27 DW (120000 SHR 16) ; obere 16bit
28
29 MyVar32W2 DW (75000 AND 0xFFFF) ; unteren 16bit
30 DW (75000 SHR 16) ; obere 16bit
31
32 Datenspeicher ENDS
33 ;
34 ;
35 Programme Section Code
36 ASSUME DPP3:SYSTEM
37 ASSUME DPP0:Datenspeicher
```

```

38 ;
39 ; Hier stehen Ihre Unterprogramme
40 ;
41 ;
42 ; PInit Initialisiert den Port Px
43 ;=====
44 ; I: nix
45 ; O: nix
46 ; Z: Register, die zerstört werden.
47 ;
48
49 Add32 Proc NEAR
50     PUSH R0
51     PUSH R1 ; Inhalt von R1 und R2 auf den Stack kopieren um die
52               Werte zusichern
53     MOV R2, [R0+] ; R0 nach R2 kopieren R0 zu R1 inkrementieren
54     ADD R2, [R1+] ; R1 auf R2 addieren R1 zu R2 inkrementieren
55     MOV R3, [R0] ; R0(R1) nach R3 kopieren
56     ADDC R3, [R1] ; R1(R2) mit Carrybit auf R3 addieren
57
58     POP R1 ; Originale vom Stack hohlen
59     POP R0
60
61     Ret
62 Add32 EndP;
63
64 Sub32 Proc NEAR
65     PUSH R0
66     PUSH R1 ; Inhalt von R1 und R2 auf den Stack kopieren um die
67               Werte zusichern
68     MOV R2, [R0+] ; R0 nach R2 kopieren R0 zu R1 inkrementieren
69     SUB R2, [R1+] ; R1 auf R2 addieren R1 zu R2 inkrementieren
70     MOV R3, [R0] ; R0(R1) nach R3 kopieren
71     SUBC R3, [R1] ; R1(R2) mit Carrybit auf R3 addieren
72
73     POP R1 ; Originale vom Stack hohlen
74     POP R0
75
76     Ret
77 Sub32 EndP;
78
79 MyUp1 PROC NEAR
80
81     Ret
82 MyUp1 EndP;
83
84 Addition PROC NEAR
85     MOV R2, [R0]
86     ADD R2, [R1]
87     ret
88 Addition EndP;
89
90 ;
91 ; Hier startet das Hauptprogramm
92 ;
93 A_RESET PROC TASK STARTUP INTNO RESET = 0
94     mov r0, BUSCON0
95     and r0, #0xFF00
96     mov BUSCON0, r0
97     mov dpp0, #PAG Datenspeicher ; Page von Datenspeicher => DPP0

```

```

98     mov     KSCCFG,#0x0003;           ;Module Enable
99     mov     R0,KSCCFG      ;wieder ruecklesen
100    JMP Main
101 A_RESET EndP
102
103 MAIN  Proc
104 ;-----
105 ;Hier geht Ihr Hauptprogramm los
106 ;
107
108     MOV R0, #MyVar32W1
109     MOV R1, #MyVar32W2
110     CALL Add32
111
112     MOV R0, #MyVar32W1
113     MOV R1, #MyVar32W2
114     CALL Sub32
115
116     MOV R0, #MyVar32W2
117     MOV R1, #MyVar32W1
118     CALL Sub32
119
120     Call    MyUp1
121
122 Forever:
123
124     jmp     Forever
125 ;
126 ;   Notstop
127 ;
128 ;
129 StopJetzt: JMP StopJetzt
130
131 Main    ENDP
132
133 Programme ENDS
134 ;
135
136     END

```

Listing 3.5: Quellcode für Rechenoperationen

!!!!!!!!!!!!!!!!!!!!Screenshot fehlt!!!!!!!!!!!!!!!!!!!!

3.4 Aufgabe 4

Wir erstellen ein Unterprogramm, das zwei 16 Bit Zahlen addiert. Die Zahlen standen direkt im Speicher. Das Unterprogramm sollte beim Aufruf in R0 einen Pointer auf den ersten Operanden, in R1 und einen Pointer auf den zweiten Operanden enthalten. Das Resultat wurde als Wert in R2 zurückgeliefert. Wir nutzten das Programm 4 mal, um die Werte in R10, R11, R13 und R14 zu berechnen.

3.4.1 Quellcode und Screenshot zu Aufgabe 4

```

1  ;$NONSEGMENTED           ; Nur Tiny Model möglich
2  $MODV2                   ; Code fuer 161/163/164/165/167
3  $NOLIST
4  $INCLUDE (REG167.INC)    ; include CPU definition file
5  $LIST

```

```

6 ;
7 ; TITLE <Mein Text>
8 ;
9 ; Prozessor Definitionen
10 ;
11 ;-----
12 ; Hier folgen Ihre Konstanten Definitionen
13 ;
14 Ora          EQU 0 ;Portbit f r Ora
15
16
17 Datenspeicher SECTION Data
18 ;-----
19 ; Hier folgen Ihre Speicher Vereinbarungen
20 ;
21 ;Mein_Speicher          DB          12          ;Byte Initialisiert
22
23 op1          DW          30000  ;
24 op2          DW          5000   ;
25 op3          DW          40000  ;
26 op4          DW          4999   ;
27 op5          DW          -30000  ;
28
29 Datenspeicher ENDS
30 ;
31 ;
32 Programme Section Code
33 ASSUME  DPP3:SYSTEM
34 ASSUME  DPP0:Datenspeicher
35 ;-----
36 ; Hier stehen Ihre Unterprogramme
37 ;
38 ;
39 ;PInit  Initialisiert den Port Px
40 ;=====
41 ; I: nix
42 ; O: nix
43 ; Z: Register , die zerst rt werden.
44 ;
45 Addi  PROC NEAR
46     MOV R2, [R0]
47     ADD R2, [R1]
48
49     Ret
50 Addi  EndP;
51
52 Subbi PROC NEAR
53     MOV R2, [R0]
54     SUB R2, [R1]
55
56     Ret
57 Subbi EndP;
58
59 ;-----
60 ; Hier startet das Hauptprogramm
61 ;
62 A_RESET PROC TASK STARTUP INTNO RESET = 0
63     mov r0,BUSCON0
64     and r0,#0xFF00
65     mov BUSCON0,r0
66     mov dpp0,#PAG Datenspeicher    ;Page von Datenspeicher => DPP0
67     mov  KSCCFG,#0x0003;          ;Module Enable

```

```

68     mov    R0,KSCCFG      ;wieder ruecklesen
69     JMP    Main
70 A_RESET EndP
71
72 MAIN   Proc
73 ;-----
74 ;Hier geht Ihr Hauptprogramm los
75 ;
76
77
78     MOV R0, #op1
79     MOV R1, #op2
80     CALL Addi
81     MOV R10, R2
82
83     MOV R0, #op1
84     MOV R1, #op3
85     CALL Addi
86     MOV R11, R2
87
88     MOV R0, #op4
89     MOV R1, #op2
90     CALL Subbi
91     MOV R12, R2
92
93     MOV R0, #op1
94     MOV R1, #op5
95     CALL Addi
96     MOV R13, R2
97
98     MOV R0, #op3
99     MOV R1, #op5
100    CALL Addi
101    MOV R14, R2
102
103
104
105 Forever:
106
107
108     jmp     Forever
109 ;
110 ;   Notstop
111 ;
112 StopJetzt: JMP StopJetzt
113
114 Main     ENDP
115
116 Programme ENDS
117 ;
118
119     END

```

Listing 3.6: Unterprogramm der Aufgabe 4

!!!!!!!!!!!!!!!!Screenshot fehlt!!!!!!!!!!!!!!!!

3.5 Aufgabe 5

Das Ziel der letzten Aufgabe war es mit 32bit Variablen zurechnen. Hoerfür wurde eine Variable angelegt, die aus 2 Words bestand welche im Speicher direkt hintereinander lagen. Mittels Bitshifting wurden die Wertebereiche der 32bit

Variable umgesetzt.

Im folgenden wurden nun die unteren 16bit der beiden zu addierenden Variablen in das Ergebnisregister geladen und addiert. Damit der Zeiger auf der nächsten Speicherstelle, und somit auf die oberen 16bit zeigt, musste in diesem Schritt die Startadressen inkrementiert werden. Im nächsten Schritt konnten die oberen 16bit mit Hilfe des *ADD C Befehls* addiert werden, hierdurch kann gesetztes Carry-bit der Addition der unteren 16bit berücksichtigt werden. Um r0 und r1 in die Ursprungsform zu bringen mussten auf dem Stack abgelegten Werte wieder in das entsprechende Register geladen werden.

3.5.1 Quellcode und Screenshot zu Aufgabe 5

!!!!!!!!!!!!!!!!!!!!Quellcode und Screenshot fehlt!!!!!!!!!!!!!!!!!!!!

Labortermin 4

Versuch IV

4.1 Aufgabe 1

4.1.1 Anmerkungen zu Aufgabe 1

4.2 Aufgabe 2

4.2.1 Anmerkungen zu Aufgabe 2

4.3 Aufgabe 3

4.3.1 Anmerkungen zu Aufgabe 3

4.4 Aufgabe 4

4.4.1 Anmerkungen zu Aufgabe 4

Labortermin 5

Versuch V

5.1 Aufgabe 1

5.1.1 Anmerkungen zu Aufgabe 1

5.2 Aufgabe 2

5.2.1 Anmerkungen zu Aufgabe 2

5.3 Aufgabe 3

5.3.1 Anmerkungen zu Aufgabe 3

Teil II

Anhang

Anhang A

Aufgabenblatt 1

Digitallabor

Versuch: Kombinatorisches und strukturelles VHDL im GAL Baustein

Ziel: Im heutigen Versuch sollen Sie erste Erfahrungen mit der Sprache VHDL sammeln und dabei auch etwas mit hierarchischem Design experimentieren. Hierbei werden wir einen kleinen programmierbaren Baustein, das GAL einsetzen.

Mit Hilfe des heutigen Versuches sollen Sie erfahren, dass die Kombination aus HDL und programmierbarer Hardware schnell zu funktionierenden Schaltungen führt, und auch ziemlich flexibel bei Änderungen ist.

Benutzen Sie zur Designerstellung das Programm „ispLever Classic“ und zur Simulation den „Aldec VHDL Functional Simulator“. Verwenden Sie durchgängig std_logic.

Aufgabe 1

Schreiben Sie ein nebenläufiges VHDL Modell für ein ODER Gatter mit zwei Eingängen. Benutzen Sie eine Gleichung zur Beschreibung der Funktionalität. Sie können analog zum in der Bedienungsanleitung beschriebenen UND3 Gatter vorgehen. Simulieren Sie Ihr Design durch direkte Eingabe der Testvektoren, programmieren Sie ein GAL und testen Sie es mit zwei Schaltern und einer LED.

Führen Sie die Funktion vor und besprechen Sie Ihre Vorgehensweise mit den Betreuern.

Aufgabe 2

Schreiben Sie ein nebenläufiges VHDL Modell für einen Halbaddierer. Benutzen Sie eine Funktionstabelle zur Beschreibung des Halbaddierers.

Verifizieren Sie die Korrektheit des Designs mit Hilfe der Simulation.

Aufgabe 3

Stellen Sie dann aus zwei Instanzen des Halbaddierers und einem ODER Gatter einen Volladdierer zusammen. Gefordert ist rein strukturelles VHDL, das nur die Verknüpfung der Komponenten beschreibt.

Tipp: Es ist sicher hilfreich, wenn Sie auf die Skizze, die Sie zur Vorbereitung gemacht haben, zurückgreifen. Beschriften Sie alle Ein- und Ausgänge, die internen Namen der Komponenten und geben Sie den Verbindungen Namen.

Simulieren Sie den Volladdierer.

Aufgabe 4

Verwenden Sie zwei Volladdierer um ein rein strukturelles Modell eines Serienaddierers für zwei Zahlen zu je zwei Bit zusammenzufügen. Verwenden Sie für die beiden Eingangszahlen a und b je einen 2 Bit breiten Vektor, für die Summe einen 3 Bit breiten Vektor. Auch hier wäre sicher eine Skizze hilfreich.

Schreiben Sie eine Testbench für das Modell. Simulieren Sie den Addierer mit Hilfe der Testbench und programmieren Sie nach erfolgreicher Simulation das GAL.

Verbinden Sie die Eingänge mit den Schaltern, die Ausgänge mit der BCD -> 7-Segment-Anzeige. Anschlussbuchse „C/C – D2“ an der 7-Segment-Anzeige muss an GND angeschlossen werden.

Na, was gibt $2 + 3$???

Anhang B

Aufgabenblatt 2

D i g i t a l l a b o r

Versuch: MACH Programmierung

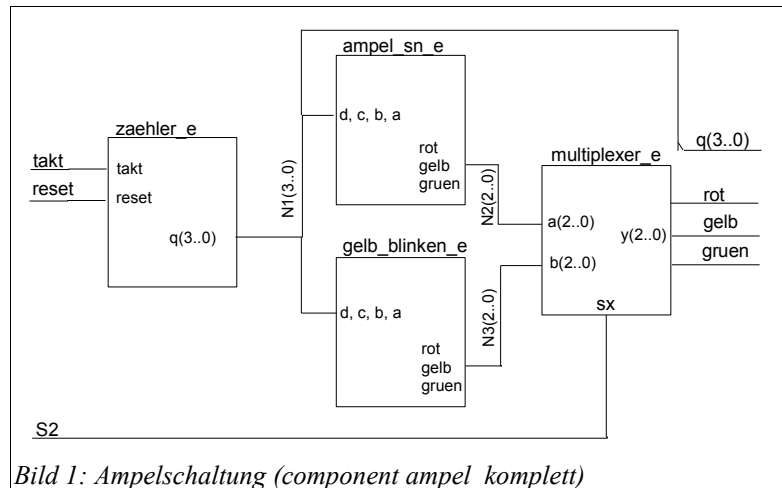
Ziel: Im heutigen Versuch gehen wir zur nächsten Technologiestufe, den CPLDs weiter. Durch Verwendung eines MACH Bausteines können wir gleichzeitig auf die In-System-Programmierung zurückgreifen. Sie brauchen den Baustein also nicht mehr in einem speziellen Gerät zu programmieren, sondern über den JTAG Anschluss direkt auf der Platine.

Sie sollten heute die Beschreibung von sequentiellen Schaltungen mit VHDL und die Simulation dieser Schaltungen kennen lernen. Natürlich sollen Sie auch Ihre Kenntnisse über das Erstellen hierarchischer Designs und den Umgang mit einer Testbench vertiefen.

Zu jeder Aufgabe ist in der Dokumentation ein Screen Shot des Simulationsergebnisses gefordert, den Sie mit dem „Snipping Tool“ einfach aufzeichnen können.

Das Ziel des heutigen Versuchs ist eine Ampelsteuerung, die aus vier Komponenten gemäß Bild 1 aufgebaut ist. Die einzelnen Komponenten werden nun Schritt für Schritt entworfen. Bleiben Sie für alle Aufgaben im gleichen Projekt und verwenden Sie durchgängig den Datentyp „std_logic“.

Für die Durchführung des Versuchs brauchen Sie ein ispMach-Board und ein I/O-Board.



Aufgabe 1

Erstellen Sie einen 4-Bit Binärzähler mit asynchronem Reset in VHDL. Die Entity muss den Namen "zaehler_e" haben. Benutzen Sie folgende Port Namen:

```
takt, reset : in std_logic;
q           : out std_logic_vector(3 downto 0)
```

Simulieren Sie das Modell durch Vorgabe der Stimuli per Simulator. Was passiert, wenn Sie Reset einfach auf '0' setzen und den Takt loslaufen lassen? Ist das ein Fehler? Wie lässt sich diese Situation vermeiden?

Aufgabe 2

Zur Simulation der Aufgabe 1 erstellen Sie sich eine Testbench mit zwei Prozessen:

```
tb_res: process -- Prozess für Reset und ggf. weitere Signale
begin
    reset <= .... ; -- ab hier folgen Ihre Zuweisungen
    wait for 10 ns; -- mit wait for ... getrennt.
    reset <= .... ; usw.
    wait ; -- Schluss
end process;

tb_takt: process -- zur Takterzeugung, Periode 100ns
begin
    takt <= '0';          -- initialisiere
    loop
        wait for 50 ns;    -- einen halben Takt warten
        takt <= not takt;  -- takt kippen
    end loop;
end process;
```

Simulieren Sie Ihr Design mit dieser Testbench.

Programmieren des CPLD-Bausteins:

Der CPLD-Baustein auf dem ispMach-Board hat einen eingebauten Oszillator, der noch konfiguriert werden muss. Dazu dient die Datei "HARD_A2.vhd" (siehe Anhang A). Sie finden die Datei im LAT unter W:\IWI-I\DTL_Vorlage. Kopieren Sie sich die Datei in ihr Projektverzeichnis.

Importieren sie die Datei in ihr Projekt. Wenn Sie sich an die Namens-Vorgaben gehalten haben, müsste ihr Projekt wie in Bild 2 dargestellt aussehen. Wichtig ist, dass HARD_A2e das Top Modul ist.

Weisen Sie mit dem Constraint Editor die Pins zu. Zählerausgang => LEDs D4 .. D1, reset => Schalter S1, Test-Eingang T1 => Taster 1. (Hinweis: Taster und Schalter sind als Pull UP zu konfigurieren).

Die zugehörigen Pins können der Bedienungsanleitung „Arbeiten mit dem ispMACH 4000ZE Breakout Board“ entnommen werden.

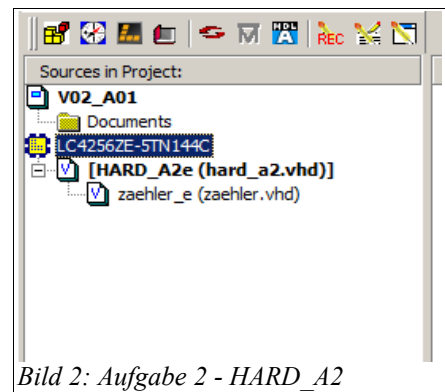


Bild 2: Aufgabe 2 - HARD_A2

Nach dem erfolgreichen Compilieren programmieren Sie nun die Hardware.

Nach dem erfolgreichen Test löschen Sie HARD_A2e wieder aus dem Projekt.

Aufgabe 3

Wie Sie in TI 1 gesehen haben, kann es ganz schön mühsam sein, eine rein kombinatorische Schaltung mit KV-Diagrammen zu entwerfen. Hier machen wir es besser: Entwerfen Sie ein VHDL Modell für einen rein kombinatorischen Ampel Steuerungsblock, der die in Tabelle 1 gezeigte Funktionstabelle mit 16 Ampelphasen implementiert.

Simulieren Sie den Block mit einer Testbench, die die 16 Eingangswerte mit Hilfe einer FOR-Schleife erzeugt.

Eingänge des Schaltnetzes				Ausgänge des Schaltnetzes		
D	C	B	A	GRÜN	GELB	ROT
0	0	0	0	1	0	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	1	1
1	1	1	1	0	1	1

Tabelle 1: Funktionstabelle des Schaltnetzes

Aufgabe 4

Fügen Sie den Ampel Steuerungsblock aus der vorigen Aufgabe in ein Strukturmodell ein, das den Zähler und das Ampel Schaltnetz instanziiert. Der Name der Entity sollte "zaehler_ampel_e" heißen (siehe Bild 3). Neben den Ports aus Aufgabe 1 kommen folgende Ports hinzu:

rot, gelb, gruen : out std_logic

Simulation mit der Testbench aus Aufgabe 2.

Zum Programmieren der Hardware kopieren Sie die Datei "HARD_A4.vhd" (siehe Anhang B) in ihr Projektverzeichnis und importieren sie in ihr Projekt.

Wenn Sie sich an obige Vorgaben gehalten haben, müsste ihr Projekt wie in Bild 4 dargestellt aussehen.

Weisen Sie die benötigten Pins zu. Es müssen alle Signals aus der Liste einem entsprechenden Pin zugewiesen werden.

Testen Sie Ihre Schaltung.

Nach dem erfolgreichen Test löschen Sie „HARD_A4e“ wieder aus ihrem Projekt.

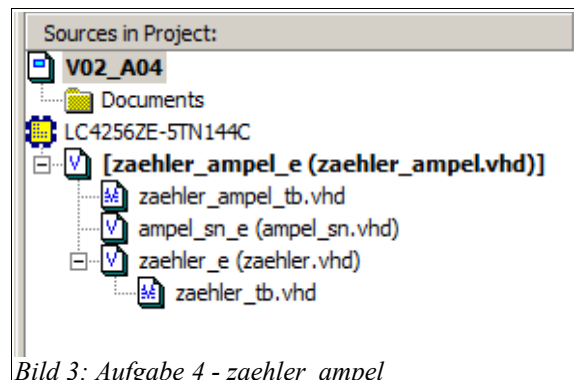


Bild 3: Aufgabe 4 - zaehler_ampel

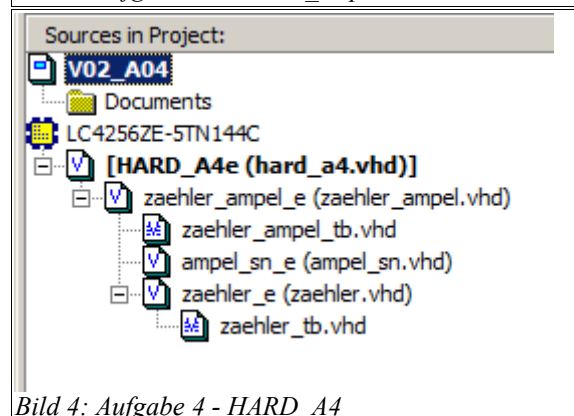


Bild 4: Aufgabe 4 - HARD_A4

Aufgabe 5

Entwerfen Sie einen zusätzlichen Ampel Steuerungsblock als getrenntes VHDL-Modul. Er liefert ebenfalls die Signale "rot / gelb / gruen" und zwar so, dass die gelbe LED zwei Takte leuchtet und zwei Takte aus ist. Die anderen sind immer aus.

Simulation mit der Testbench aus Aufgabe 3.

Aufgabe 6

Entwerfen Sie ein sequentielles VHDL Modell eines 2:1 Multiplexers für zwei je drei Bit breite std_logic Vektoren A und B, die abhängig von einem Signal Select auf den drei Bit breiten Ausgangsvektor Y durchgeschaltet werden.

Test per Simulation.

Aufgabe 7

Als letztes fügen Sie alle Komponenten so wie in Bild 1 gezeigt zusammen. Die Entity muss ampel_komplett_e heißen. Neben den Ports aus Aufgabe 4 kommt noch der Port:

S2 : in std_logic

zur Umschaltung der Betriebsarten hinzu. Simulieren Sie mit Hilfe der um S2 erweiterten Testbench aus Aufgabe 2.

Zum Programmieren der Hardware kopieren Sie sich die Datei "HARD_A7.vhd" (siehe Anhang C) in ihr Projektverzeichnis und importieren sie in ihr Projekt.

Wenn Sie sich an obige Vorgaben gehalten haben, müsste ihr Projekt wie in Bild 5 dargestellt aussehen.

Weisen Sie die benötigten Pins zu. Verwenden Sie Schalter S2 als Betriebsarten Umschalter.

Testen Sie Ihre Schaltung.

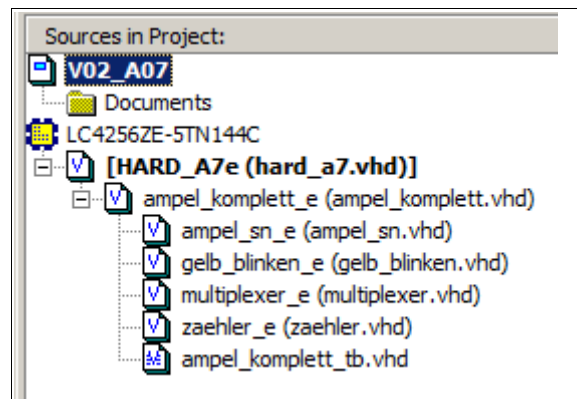


Bild 5: Aufgabe 7 - HARD_A7

Anhang A: Datei "HARD_A2.vhd"

```
library ieee;
library MACH;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use MACH.components.all;

entity HARD_A2e is
    port (S1      : in  std_logic; -- Schalter S1 => reset
          T1      : in  std_logic; -- Taster T1 => Test-Eingang
          q       : out std_logic_vector(3 downto 0)); -- Zaehlerausgang
end;

architecture HARD_A2a of HARD_A2e is
    signal takt    : std_logic;
    signal q_out   : std_logic_vector(3 downto 0);

    component OSCTIMER
        generic (TIMER_DIV : string);
        port (DYNOSCDIS    : in  std_logic;
              TIMERRES     : in  std_logic;
              OSCOUT       : out std_logic;
              TIMEROUT     : out std_logic);
    end component;

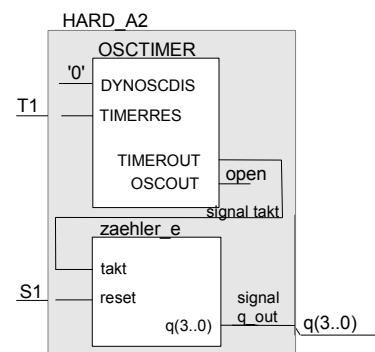
    component zaehler_e is
        port (takt, reset : in  std_logic;
              q           : out std_logic_vector(3 downto 0));
    end component;

begin
    i1: OSCTIMER
        generic map (TIMER_DIV => "1048576") -- Teilungsfaktor - es sind nur 3 Werte
                                                -- zulässig: 128, 1024 und 1048576
        port map (DYNOSCDIS => '0',
                  TIMERRES  => not T1,      -- Taster T1 zum Test
                  OSCOUT    => open,
                  TIMEROUT  => takt);      -- auf signal takt

    i2 : zaehler_e port map (takt=>takt, reset=>S1, q=>q_out);

    q <= not q_out; -- aktuellen Zaehlerstand den LEDs invertiert zuweisen
                    -- da LEDs leuchten, wenn eine 0 anliegt

end HARD_A2a;
```



Anhang B: Datei "HARD_A4.vhd"

```
library ieee;
library MACH;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use MACH.components.all;

entity HARD_A4e is
    port (S1          : in  std_logic; -- Schalter S1 => reset
          T1          : in  std_logic; -- Taster T1 => Test-Eingang
          rot, gelb, gruen : out std_logic;
          q            : out std_logic_vector(3 downto 0)); -- Zaehlerausgang
end;

architecture HARD_A4a of HARD_A4e is
    signal takt      : std_logic;
    signal q_out     : std_logic_vector(3 downto 0);

    component OSCTIMER
        generic (TIMER_DIV : string);
        port     (DYNOSCDIS : in  std_logic;
                  TIMERRES  : in  std_logic;
                  OSCOUT    : out std_logic;
                  TIMEROUT  : out std_logic);
    end component;

    component zaehler_ampel_e is
        port (takt      : in  std_logic;
              reset     : in  std_logic;
              q          : out std_logic_vector(3 downto 0);
              rot, gelb, gruen : out std_logic);
    end component;

begin
    i1: OSCTIMER
        generic map (TIMER_DIV => "1048576") -- Teilungsfaktor
        port map (DYNOSCDIS => '0',
                  TIMERRES  => not T1,      -- Taster T1 zum Test
                  OSCOUT    => open,
                  TIMEROUT  => takt);      -- auf signal takt

    i2: zaehler_ampel_e port map (takt => takt, reset => S1,
                                  rot => rot, gelb => gelb, gruen => gruen,
                                  q => q_out);

    q <= not q_out; -- aktuellen Zaehlerstand den LEDs invertiert zuweisen
                  -- da LEDs leuchten, wenn eine 0 anliegt

end HARD_A4a;
```

Anhang C: Datei "HARD_A7.vhd"

```
library ieee;
library MACH;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use MACH.components.all;

entity HARD_A7e is
  port (S1          : in  std_logic; -- S1 => reset
        S2          : in  std_logic; -- S2 => Sx
        T1          : in  std_logic; -- Test-Eingang
        rot, gelb, gruen : out std_logic;
        q           : out std_logic_vector(3 downto 0)); -- Zaehlerausgang
end;

architecture HARD_A7a of HARD_A7e is
  signal takt      : std_logic;
  signal q_out     : std_logic_vector(3 downto 0);

  component OSTIMER
    generic (TIMER_DIV : string);
    port (DYNOSCDIS    : in  std_logic;
          TIMERRES     : in  std_logic;
          OSCOUT       : out std_logic;
          TIMEROUT     : out std_logic);
  end component;

  component ampel_komplett_e is
    port (takt      : in  std_logic;
          reset     : in  std_logic;
          S2        : in  std_logic; -- Umschalter, Ampel oder gelb blinken
          q         : out std_logic_vector(3 downto 0);
          rot, gelb, gruen : out std_logic);
  end component;

begin
  i1: OSTIMER
    generic map (TIMER_DIV => "1048576")
    port map (DYNOSCDIS => '0',
              TIMERRES  => not T1,
              OSCOUT    => open,
              TIMEROUT  => takt);

  i2: ampel_komplett_e port map (takt=>takt, reset=>S1, S2=>S2,
                                q=>q_out,
                                rot=>rot, gelb=>gelb, gruen=>gruen);
  q <= not q_out; -- aktuellen Zaehlerstand den LEDs invertiert zuweisen
                 -- da LEDs leuchten, wenn eine 0 anliegt

end HARD_A7a;
```


Anhang C

Aufgabenblatt 3

Digitallabor

Versuch: Erste Schritte mit maschinennaher C166 Programmierung

Ziel: Im heutigen Versuch sollen Sie die internen Abläufe in einem typischen Prozessor verstehen. Sie sollten sich erarbeiten, was die Unterschiede zwischen Konstanten und Variablen sind, und wie der Prozessor mit den internen Registern arbeitet. Weiterhin sollten Sie den Übergabemechanismus von Parameteradressen an Unterprogramme (call by reference) und die Verarbeitung der Parameter per indirekter Adressierung verstehen.

Verwenden Sie die Keil Software mit der vorgegebenen Vorlage. Zum Debuggen brauchen wir heute keine Hardware, der Simulator genügt. Verständnisfragen, die bei den Aufgaben gestellt werden, beantworten Sie in der Ausarbeitung.

Aufgabe 1

Vereinbaren Sie Konstanten mit folgenden Namen und Werten:

op1 = 30000, op2 = 5000, op3 = 40000, op4 = 4999, op5 = -30000

Im Hauptprogramm laden Sie dann einfach die Register R1 bis R5 mit den Konstanten op1 bis op5. Assemblieren und binden Sie Ihr Programm und debuggen Sie im Einzelschritt. Öffnen Sie ggf. das "Register-Window", falls es nicht offen ist. Werden die richtigen Werte in die Register geladen? Welche Adressierungsart mussten Sie verwenden, und an welcher Stelle im übersetzten Programmcode tauchen die Konstanten auf? Schauen Sie zur Beantwortung dieser Frage im Assembler-Listing und mit Hilfe des Debuggers direkt im Programmspeicher nach.

Aufgabe 2

Kopieren Sie Ihr Programm aus Aufgabe 1 in eine neue Datei und übernehmen Sie diese in Ihr Projekt. Löschen Sie die Konstanten aus Ihrem Programm und vereinbaren Sie stattdessen gleichnamige 16 Bit-Variablen, die mit den Werten aus Aufgabe 1 initialisiert sind.

Im Hauptprogramm laden Sie dann wiederum die Register R1 bis R5 mit den Werten der Variablen op1 bis op5. Debuggen Sie erneut und verifizieren Sie, dass die richtigen Werte geladen werden. Welche Adressierungsart wurde diesmal verwendet, und was taucht nun im übersetzten Programmcode an Stelle der Konstanten auf? Ermitteln Sie mit Hilfe des Assembler- und des Linker-Listings die Adresse der Variablen und zeigen Sie den entsprechenden Speicher im "Memory-Window" an. Was fällt auf.

Aufgabe 3

Erweitern Sie das Programm aus Aufgabe 2, indem Sie nach dem Laden der Register R1 bis R5 folgende Rechenoperationen mit den Register-Operanden und den Ziel-Registern R10 bis R14 durchführen:

$R10 = op1 + op2$; $R11 = op1 + op3$; $R12 = op4 - op2$; $R13 = op1 + op5$, $R14 = op3 + op5$;

Debuggen Sie Ihr Programm im Single Step. Stimmen die Rechenergebnisse? Notieren Sie nach jeder Rechenoperation die Werte der Flags C, Z, V und N. Wie kommen diese Werte zustande und was bedeuten sie?

Aufgabe 4

Schreiben Sie ein Unterprogramm, das zwei 16 Bit Zahlen addiert. Die Zahlen stehen direkt im Speicher. Das Unterprogramm soll beim Aufruf in R0 einen Pointer auf den ersten Operanden, in R1 einen Pointer auf den zweiten Operanden erhalten. Das Resultat wird als Wert in R2 zurückgeliefert. Ausser R2 und den Flags soll das UP keine Register zerstören.

Nutzen Sie Ihr Programm 4 mal, um die Werte in R10, R11, R13 und R14 zu berechnen.

Aufgabe 5

Na ja, die 16 bittige Rechnerei ist ja bereichsmäßig wohl doch etwas eingeschränkt und es war zugegeben keine gute Idee, ein Unterprogramm zu schreiben, dessen Aufruf-Overhead größer ist, als der Nutzen. Beides wollen wir jetzt ändern und Unterprogramme für 32 Bit Arithmetik erstellen.

Dazu legen Sie zunächst zwei 32 Bit Variablen an, die Sie auf die Werte 120000 und 75000 initialisieren.

Moment mal, 32 Bit Variablen anlegen, das haben wir doch gar nicht besprochen?! Stimmt, denn der Prozessor unterstützt dieses Datenformat nicht. Und wie immer, wenn entweder der Prozessor, oder die Programmiersprache ein Datenformat nicht unterstützt, dann muss man es eben selbst programmieren. Hier legen Sie eben einfach zwei 16 Bit Werte hintereinander in den Speicher. Ganz little Endian mäßig kommen zuerst die unteren 16 Bit, dann die oberen. Das sieht dann so aus:

```
MyVar32    DW    (120000 AND 0xFFFF)    ;untere 16 Bit
            DW    (120000 SHR 16)        ;obere 16 Bit
```

So, nachdem wir nun die beiden 32 Bit Variablen angelegt haben, wollen wir auch mit ihnen rechnen:

Schreiben Sie ein Unterprogramm "add32", das zwei 32 Bit Zahlen addiert. Es erhält die Pointer wie in Aufgabe 4 und liefert das Resultat in R2 (untere 16 Bit) und R3 (obere 16 Bit) zurück. Ausser R2, R3 und den Flags soll es ebenfalls keine Register zerstören.

Rufen Sie das UP mit den beiden Variablen auf und berechnen Sie $120000 + 75000$.

Abschliessend kopieren Sie Ihr UP und modifizieren die Kopie zu "sub32", das zwei 32 Bit Zahlen, bei gleicher Aufruf-Struktur subtrahiert.

Berechnen Sie zusätzlich $120000 - 75000$ und $75000 - 120000$ und geben Sie die Hex-Resultate der 3 Rechnungen in der Ausarbeitung an.

Anhang D

Aufgabenblatt 4

Digitallabor

Versuch: Nutzung des UConnect XE166 Real Time Signal Controllers

Ziel: Im heutigen Versuch sollen Sie sich mit den Bitbefehlen des C166 – speziellen und wortweise arbeitenden – vertraut machen und diese auf die Parallelports des XE166 anwenden. Als weiteres Ziel des heutigen Versuchs sollen Sie das Debugging eines Embedded System kennen lernen.

Verwenden Sie die Keil 3 Software mit der vorgegebenen Assembler-Vorlagen Datei. Zum Debuggen verwenden wir heute den **UConnect XE166 Real Time Signal Controller** (siehe Bild 1), der an den USB-Bus des PCs angeschlossen wird, sowie eine kleine Platine mit 4 LEDs, 2 Schaltern und 2 Tastern (siehe Bild 2), die am Port P0 des Prozessors angeschlossen ist. Die Zuordnung der Portbits zu den LEDs, Schaltern und Tastern finden Sie in Tabelle 1.

Tip: Übertragen Sie die Werte aus der Tabelle gleich in EQUs. Die LEDs leuchten, wenn eine '0' am Port liegt. Die Tasten liefern den Wert '0', wenn sie gedrückt sind. Die Schalterstellung "oben" liefert den Wert '1'.

Legen Sie für jede Aufgabe ein neues Projekt an.



Bild 1: UConnect XE166 Real Time Signal Controller

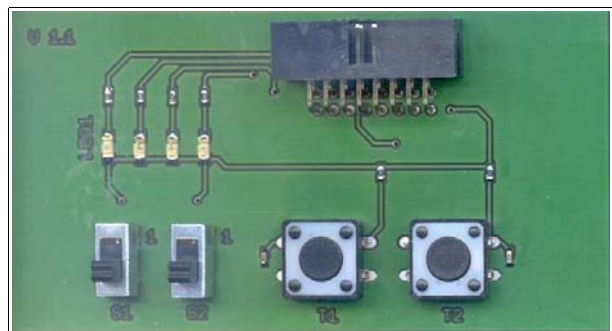


Bild 2: Platine mit 4 LEDs, 2 Schaltern und 2 Tastern

Anschlußbelegung der Platine mit 4 LEDs, 2 Schaltern und 2 Tastern.

<i>X</i>	<i>Steckerbelegung</i>	<i>Port</i>	<i>Zugriff über</i>	<i>Richtungsregister</i>
S1: Schalter links	Pin 10	P0.0	P0_IN.0	P0_IOCR_0
S2: Schalter rechts	Pin 8	P0.1	P0_IN.1	P0_IOCR_1
T1: Taster links	Pin 7	P0.2	P0_IN.2	P0_IOCR_2
T2: Taster rechts	Pin 9	P0.3	P0_IN.3	P0_IOCR_3
LED1: rot	Pin 13	P0.4	P0_OUT.4	P0_IOCR_4
LED2: gelb	Pin 11	P0.5	P0_OUT.5	P0_IOCR_5
LED3: grün	Pin 12	P0.6	P0_OUT.6	P0_IOCR_6
LED4: rot	Pin 14	P0.7	P0_OUT.7	P0_IOCR_7

Tabelle 1: Zuordnung der Portbits zu den LEDs, Schaltern und Tastern der Platine

Aufgabe 1

Schreiben Sie ein Assembler Programm, das LED1 leuchten lässt, wenn die Taste T1 gedrückt wird und LED2 leuchten lässt, wenn die Taste T2 gedrückt wird.

Die Initialisierung des Ports P0 führen Sie in einem Unterprogramm "PortInit" aus, das die Pins für Tasten und Schalter auf Eingang, die für die LEDs auf Ausgang stellt. Die Richtungsregister heißen P0_IOCR_0 bis P0_IOCR_7 für Portpin 0 bis 7. Ihr Programm PortInit hat damit etwa folgenden Aufbau:

```
PortInit PROC
; fuer die Ausgaenge
    mov     R0,# ...           ;Wert fuer Ausgang
    mov     P0_IOCR_?,R0      ;fuer alle Ausgaenge
    mov     P0_IOCR_?,R0      ;einzeln zuweisen
    ...
; jetzt die Eingaenge
    mov     R0,# ...           ;Wert fuer Eingang
    mov     P0_IOCR_?,R0      ;fuer alle Eingaenge
    mov     P0_IOCR_?,R0      ;einzeln zuweisen
    ...
    ret
PortInit    EndP
```

Das Hauptprogramm geht nach dem Aufruf von PortInit in eine Endlosschleife. Wie viele Befehle brauchen Sie in dieser Schleife?

Aufgabe 2

Erweitern Sie die Lösung aus Aufgabe 1 so, dass die Tasten nur dann eine Wirkung haben, wenn der entsprechende Schalter "oben" steht. Tip: Als Zwischenspeicher für einzelne Bits können Sie die Bits der GPRs (R0 bis R15) verwenden, da diese ja ebenfalls bit-adressierbar sind

Aufgabe 3

In dieser Aufgabe wollen wir uns auf das Niveau eines einfacheren Desktop Prozessors hinabgeben, der keine Bitbefehle kennt. Lösen Sie die Aufgabe 1 unter Verzicht auf die Einzelbit-Befehle des C166.

So richtig unübersichtlich würde die Sache dann für Aufgabe 2. Deshalb wollen wir es mit Aufgabe 1 bewenden lassen. Verstehen Sie jetzt, warum ein Desktop Prozessor, vom Preis einmal abgesehen, gar keine so gute Lösung für ein embedded System wäre?

Aufgabe 4

Schreiben Sie ein Unterprogramm "Delay", das nichts anderes macht, als etwa eine halbe Sekunde zu warten. Nutzen Sie das UP um LED3 im Sekundentakt blinken zu lassen. Wenn das klappt, fügen Sie die beiden Befehle aus Aufgabe 1, die die LEDs bedienen, zu Ihrem Hauptprogramm dazu. Das UP "Delay" lassen Sie selbstverständlich unverändert. Gehen die LEDs an, wenn Sie auf die Tasten drücken?

Anhang E

Aufgabenblatt 5

Digitallabor

Versuch: Anwendung von Hochsprachen für hardwarenahe Programmierung

Ziel: Im heutigen Versuch sollen Sie die hardwarenahe Programmierung mit Hilfe der Sprache „C“ kennen lernen. Dazu benutzen Sie wieder mit Hilfe des Uconnect USB die typischen Peripherie-Komponenten Parallelports und Timer des XE164.

Für die erfolgreiche Versuchsdurchführung müssen nachfolgende Einstellungen im Keil-Programm µVision3 gemacht werden:

Einstellungen zu den Aufgaben 1, 2 und 3

- legen Sie ein neues Projekt an und wählen Sie den Microcontroller **XE164F-96F** aus.
- Startup Code für die Simulation kopieren – unter Source Group 1 muss die Datei START_V3.A66 stehen.
Nachfolgende Parameter müssen in der Datei START_V3.A66 geändert werden:
=> Zeile 292: \$SET (INIT_HPOSCON = 0)
=> Zeile 349: \$SET (INIT_PLLCON = 0)
- Nachfolgende Projekteinstellungen müssen unter „Menüleiste: Project /Options for Target 'Target 1' “ gemacht werden:
- Register Listing: Haken bei Assembly Code
- Register C166: Haken bei Double-precision Floating-point

Einstellungen nur für die Aufgabe 1 (Simulation)

- Register L166 Misc: Im Feld für Interrupt Vector Table Address muss die Adresse **0x000000** stehen
- Register Debug: Use Simulator auswählen, Haken bei „Load Application at Startup“ und bei „Run to main()“

Aufgabe 1

Passen Sie das allererste Übungsbeispiel aus Kernighan/Ritchie, die Umwandlung Fahrenheit in Celsius auf den XE164 an. Die Ausgabe über die Konsole ersetzen Sie durch Anschauen der Werte für Celsius mit dem Debugger im Simulationsmodus. Den Quellcode finden Sie unten (siehe Text 1).

Fordern Sie im Listing die Ausgabe des erzeugten Assemblercodes an (siehe Einstellungen zu den Aufgaben). Fertigen Sie eine Tabelle an, die für die Teilaufgaben b) bis e) nachfolgend Werte enthält:

- **Codegröße** die nach der Übersetzung unten im Log-Fenster angezeigt wird (siehe Bild 1 auf Seite 4)
- **Laufzeit** bis zum Ende des Programms (Run bis Breakpoint auf schließende Klammer!) die im Debugger im Registerfenster ganz unten vor dem PSW oder in der Statusleiste des Debuggers als t1: angezeigt wird (siehe Bild 2 auf Seite 4).


```

/* Umwandlung von Fahrenheit in Celsius fuer fahr = 0, 20, ..., 300 */

void main(void) {
    int celsius, fahr;
    int lower, upper, step;

    lower = 0;
    upper = 300;
    step = 20;

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9; //Diese Zeile kommt in c) in Function
        fahr = fahr + step;
    }
}

```

Text 1: Quellcode: Umwandlung Fahrenheit in Celsius

- Übersetzen und binden Sie das Programm. Schauen Sie sich den erzeugten Code im Programmlisting an. Was fällt auf?
- Ok, da müssen wir dem Compiler wohl etwas auf die Sprünge helfen. Verlegen Sie die Deklaration von celsius vor „main“, alles andere bleibt wie es ist. Schauen Sie den erzeugten Code im Listing an und achten Sie auch mal auf die Multiplikation mit 5.
- Im nächsten Schritt lagern Sie die Zeile zur Umrechnung in eine Funktion fahr2cels aus. Achten Sie im erzeugten Code auf die Parameterübergabe in und aus der Funktion.
- Jetzt wollen wir die Rechenkünste des Prozessors mal austesten. Ändern Sie die Deklaration von fahr und celsius und natürlich auch der Funktion fahr2cels nacheinander in long, float und double (dafür müssen Sie auch die Option im C166 Reiter ändern). Auswirkungen auf Code und Rechenzeit?
- Wie müsste man das Programm umgestalten, damit es bei gleicher Rechengenauigkeit schneller wird?

Einstellungen nur für Aufgabe 2 und 3 mit der Hardware UConnect XE166

- Register L166 Misc: Im Feld für Interrupt Vector Table Address muss die Adresse **0xC00000** stehen.
- Register Debug: **Infineon DAS Client for XC16x** auswählen, Haken bei „Load Application at Startup“ und bei „Run to main()“
=> Button Settings: DAS Server: **JTAG over USB Chip** auswählen. Als Device wird bei funktionierender Hardware „XE166/XC2000-Family“ angezeigt.
=> Register Flash Download Options auswählen. Nachfolgenden Programming-Algorithm hinzufügen: XE16x-96F On-chip Flash.
- Register Utilities: Nachfolgenden Target Driver for Flash Programming auswählen: Infineon **DAS Client for XC16x**
=> Zur Überprüfung des Programming-Algorithm => Button Settings drücken. Überprüfen Sie die Einstellungen.
- Die Datei t3power.c befindet sich im Verzeichnis W:\IWI-I\mc_C167*. Die Include Datei XE164F_HS.h wurde schon in das entsprechende Verzeichnis kopiert (C:\Keil3\C166\inc*).

Aufgabe 2

Schreiben Sie ein eigenes Header File, das die Deklarationen (ohne Verwendung der Keil Erweiterungen) der 8 Richtungssteuerregister P0_IOCRxx sowie P0_OUT und P0_IN enthält. Lösen Sie Aufgabe 1 und 2 des vorigen Aufgabenblattes (Taster und LEDs) mit Hilfe Ihrer Definitionen und von Maskierungsoperationen. Achten Sie wieder auf den erzeugten Code.

Aufgabe 3

Für diese Aufgabe greifen Sie auf die Definitionen der Keil Entwicklungsumgebung zurück. Die Definitionen aus Aufgabe 2 brauchen Sie hier nicht mehr.

Verwenden Sie den Timer T3, um die grüne LED mit 1 Hz blinken zu lassen und die beiden LEDs über die Tasten diesmal ohne Verzögerung zu bedienen.

Damit das klappt, müssen Sie zwei Voraussetzungen schaffen:

1. Fügen Sie über #include "XE164F_HS.h" die Register und Bitdefinitionen in Ihren Code ein. Nun haben Sie die Bezeichnungen aus Tabelle 1 auch für einzelne Bits zur Verfügung. Ihr Header File aus Aufgabe 2 brauchen Sie nun nicht mehr.
2. Fügen Sie die Datei t3power.c in Ihr Projekt ein und starten Sie ganz zu Beginn Ihrer Initialisierungen die Funktion void t3power(void); um den T3 einzuschalten.

Initialisieren und starten Sie den T3 in einer eigenen Methode T3Init. In der Hauptschleife fügen Sie neben den Zeilen, die die Tasten in die LEDs kopieren, einfach eine Zeile ein, die T3OTL in die LED kopiert. Das ist zwar nicht ganz optimal, denn normalerweise würde man diesen Job eher per Interrupt erledigen, hier aber durchaus ok, da der Prozessor ja ohnehin in einer Schleife läuft. Eine (am besten zusätzliche) Lösung per Interrupt ist aber nicht verboten, wenn die Vorlesung schon so weit vorangekommen ist.

	Port	Zugriff über	Richtungsregister
S_1: Schalter links	P0.0	P0_IN_P0	P0_IOCR00
S_2: Schalter rechts	P0.1	P0_IN_P1	P0_IOCR01
T_1: Taster links	P0.2	P0_IN_P2	P0_IOCR02
T_2: Taster rechts	P0.3	P0_IN_P3	P0_IOCR03
LED1: rot	P0.4	P0_OUT_P4	P0_IOCR04
LED2: gelb	P0.5	P0_OUT_P5	P0_IOCR05
LED3: grün	P0.6	P0_OUT_P6	P0_IOCR06
LED4: rot	P0.7	P0_OUT_P7	P0_IOCR07

Tabelle 1: Zuordnung der Portbits zu den LEDs, Schaltern und Tastern der Platine

f _{CPU} = 10MHz BPS1 = 00 _B	Timer Input Selection T2I / T3I / T4I							
	000 _B	001 _B	010 _B	011 _B	100 _B	101 _B	110 _B	111 _B
Prescale factor	8	16	32	64	128	256	512	1024
Input Frequency	1,25MHz	625,0kHz	312,5kHz	156,25kHz	78,125kHz	39,06kHz	19,53kHz	9,77kHz
Resolution	800ns	1,6µs	3,2µs	6,4µs	12,8µs	25,6µs	51,2µs	102,4µs
Period	52,43ms	104,9ms	209,7ms	419,4ms	838,9ms	1,678s	3,355s	6,711s

Tabelle 2: T3 Vorteiler

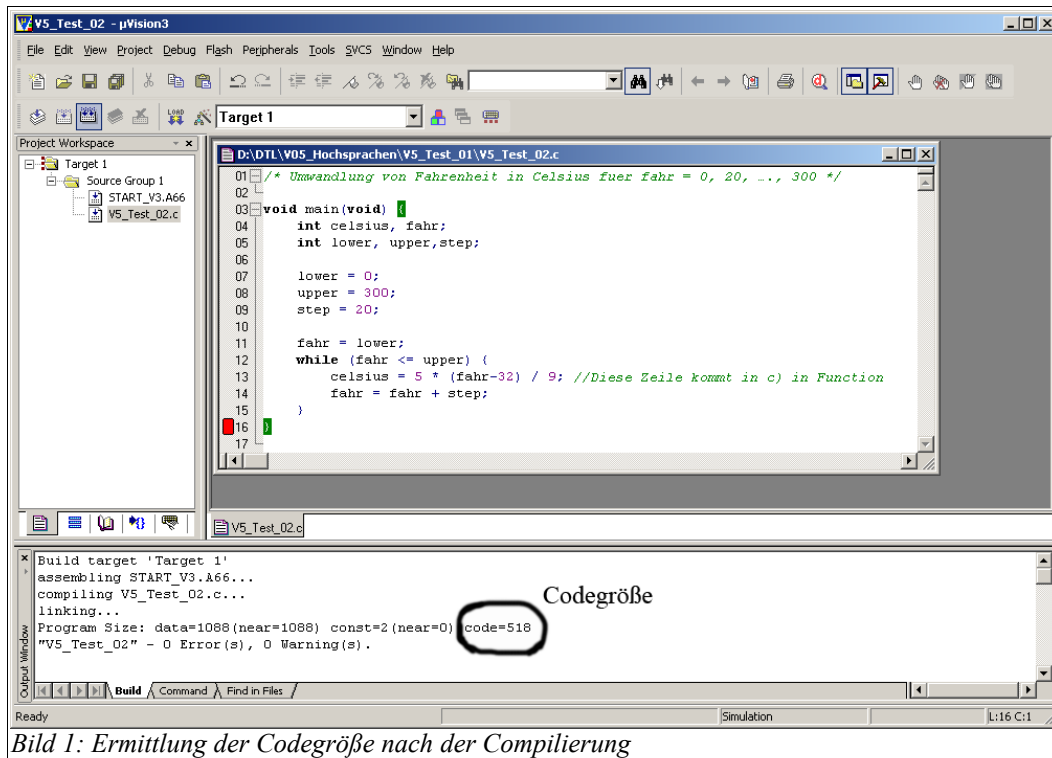


Bild 1: Ermittlung der Codegröße nach der Compilierung

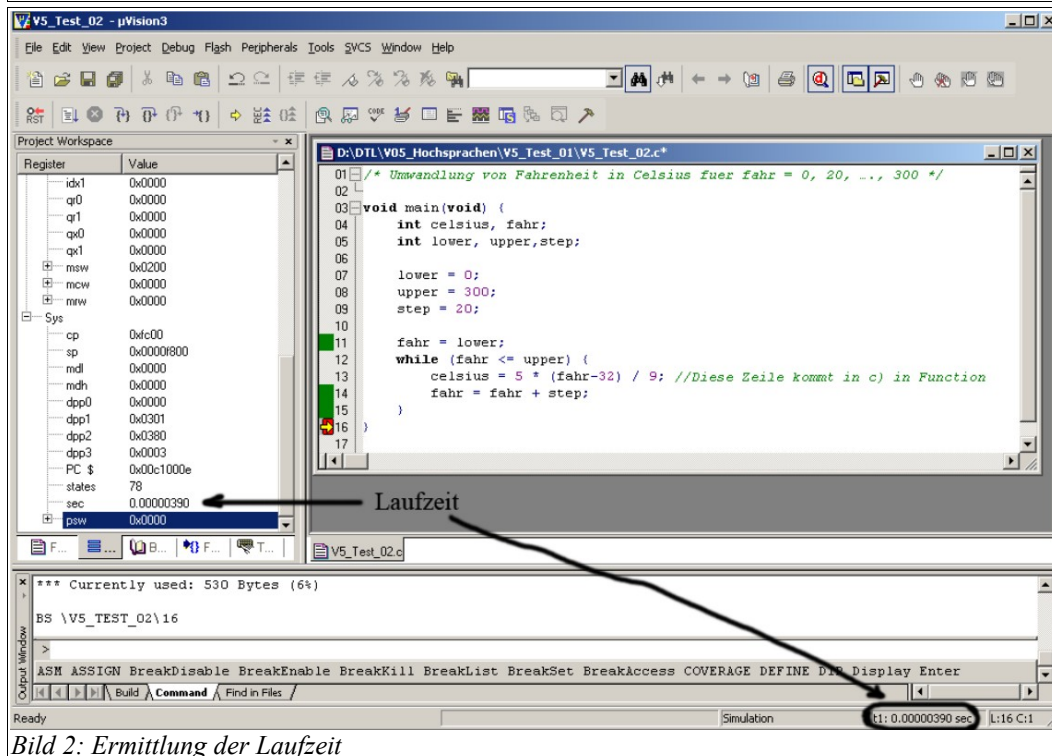


Bild 2: Ermittlung der Laufzeit