

Labor Projekt

RISC-V RV32I Softcore Dokumentation

Guillaume Fournier-Mayer
Tinf-101922
Fachsemester 7
Verwaltungssemester 12

Wedel, den 10. Mai 2021

Inhaltsverzeichnis

I Benutzerhandbuch	1
1 Ablaufbedingungen	2
1.1 Software	2
1.2 Hardware	2
2 Bedienungsanleitung	3
2.1 Öffnen des Quartus Prime Projektes	3
2.2 Bauen des Projektes	4
2.3 Flashen des FPGAs	5
2.4 Kompilieren des Programmcodes	7
2.4.1 Plattform.h	8
2.5 Flashen des Softcores	10
2.6 Zurücksetzen	11
II Entwicklerhandbuch	12
3 Motivation	13
3.1 RISC-V	13
3.2 FPGA (Field Programmable Gate Array)	13
4 Entwicklerkonfiguration	14
4.1 Software	14
4.2 Hardware	14
5 Probelmanalyse	15
5.1 Software	15
5.2 Hardware	15
5.3 RV32I Befehlssatz	17
5.4 Execution environment interface (EEI)	18
5.5 Softcore Design	18
5.5.1 Mikroarchitektur	18
5.5.2 Befehlsverarbeitung	20
5.5.3 Speicher	22

6 Realisation in VHDL	24
6.1 Steuerwerk	25
6.1.1 Dekodiereinheit	25
6.1.2 Programm Counter (PC)	25
6.1.3 Branch	26
6.2 Register	27
6.2.1 Registereinheit	27
6.2.2 Register Multiplexer	27
6.3 Arithmetic logic unit (ALU)	28
6.3.1 Rechenwerk	28
6.3.2 Rechenwerk Multiplexer	29
6.4 Memory	30
6.4.1 Byte-Adressierung	31
6.4.2 Initialisierung	31
6.4.3 Sign-Extender	32
6.5 IO	33
6.5.1 LED	34
6.5.2 UART	34
7 Toolchain	35
7.1 Compiler	35
7.1.1 Executable and Linkable Format (ELF)	35
7.1.2 Linker	36
7.2 Bootloader	37
7.3 MIF Generierung	37
8 Tests	39
8.1 Unit Tests	39
8.1.1 Rechenwerk (test_alu.vhd)	39
8.1.2 Vergleichseinheit (test_comparator.vhd)	39
8.1.3 Multiplexer (test_mux_*.vhd)	40
8.1.4 Vergleichseinheit (test_pc.vhd)	40
8.1.5 Vergleichseinheit (test_sign_extender_mem.vhd)	40
8.2 End-to-End Tests	40
8.2.1 Kontrollstrukturen (counter.c)	40
8.2.2 Shiftoperationen (lightshift.c)	41
8.2.3 Multiplikation und Division (mul_div.c)	41
8.2.4 Unterprogrammsprünge (subroutines.c)	41
8.2.5 Rekursion (recursion.c)	41
8.2.6 Linker (tobig.c)	42
9 Ausblick	43

Teil I

Benutzerhandbuch

Kapitel 1

Ablaufbedingungen

Um den Softcore in Betrieb zu nehmen wird folgende Soft- und Hardware benötigt.

1.1 Software

Software	Version	Quelle
Quartus Prime Lite	20.1.1	Intel
Arrow USB Programmer	2.4.1-1	Trenz Electronics
RISC-V GCC Toolchain	9.2.0	RISC-V Organisation
Golang Toolchain	1.16.2	RISC-V Organisation

Tabelle 1.1: Benötigte Software für Inbetriebnahme

1.2 Hardware

Hardware	Version	Quelle
TEI0003 - CYC1000	0.2	Trenz Electorincs
Micro-USB-Kabel	2.0	

Tabelle 1.2: Benötigte Hardware für Inbetriebnahme

Kapitel 2

Bedienungsanleitung

2.1 Öffnen des Quartus Prime Projektes

Zunächst wird *Quartus Prime Lite* gestartet. Sobald das Hauptfenster geöffnet wurde kann über den Menüpunkt *File* der Untermenüpunkt *Open Project* (Abbildung 2.1) geklickt werden. Daraufhin öffnet sich ein Dateibrowser (Abbildung 2.2) durch den das Projekt ausgewählt werden kann. Der relative Pfad des Projektes im Repository ist dabei *RiscV-i32/CPU/Design/RiscV.qpf*

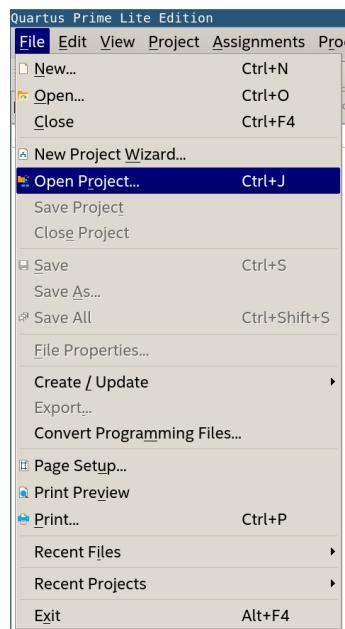


Abbildung 2.1: Öffnen des Projektes in Quartus



Abbildung 2.2: Öffnen des Projektes in Quartus

2.2 Bauen des Projektes

Sobald sich das Projekt erfolgreich geöffnet hat, kann das Bauen über ein Klick auf das *Play*-Symbol (Abbildung 2.3) gestartet werden. Das eigentliche Bauen kann, je nach Leistung des Rechners mehrere Minuten dauern. Dabei wird der Status im unteren Fensterdrittel angezeigt. War das Bauen erfolgreich, wird die Meldung *Quartus Prime Full Compilation was successful* (Abbildung 2.4) angezeigt.

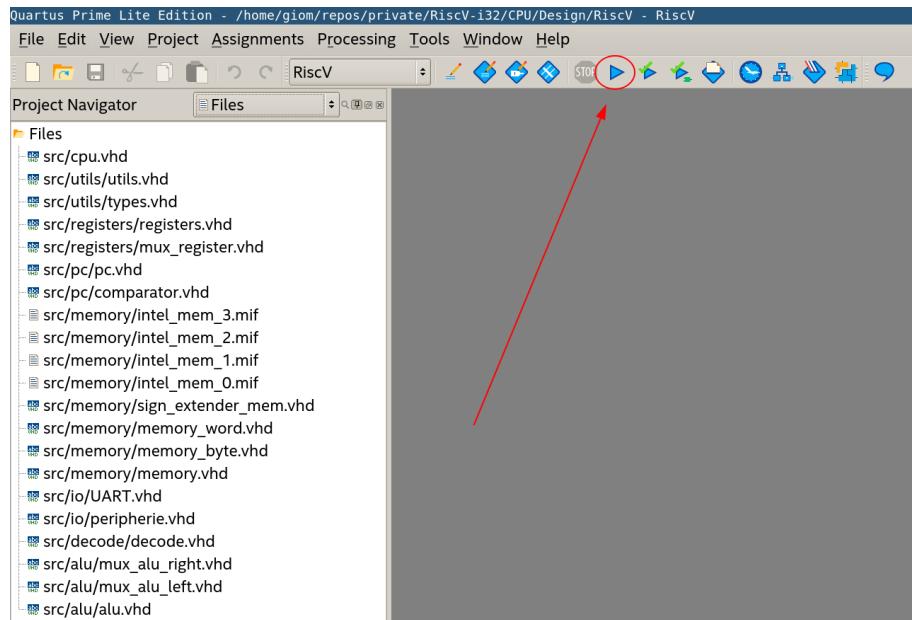


Abbildung 2.3: Bauen des Projektes in Quartus

293000 Quartus Prime Full Compilation was successful. 0 errors, 0 warnings

Abbildung 2.4: Erfolgreiches Bauen des Projektes in Quartus

2.3 Flashen des FPGAs

Zunächst muss das FPGA-Board über USB angeschlossen werden. Zusätzlich muss der Treiber installiert sein. Ist dies der Fall kann über den Menüpunkt *Tools* der Unter- menüpunkt *Programmer* ausgewählt werden (Abbildung 2.5). Dadurch öffnet sich der Programmer (Abbildung 2.6) und die Hardware kann durch einen Klick auf *Hardware Setup* eingerichtet werden. Ein weiteres Fenster öffnet sich (Abbildung 2.7). Ein Doppelklick auf *Arrow-USB-Blaster (1)* wählt das FPGA-Board als Ziel für den Programmer. Dies kann durch *Currently selected Hardware (2)* bestätigt werden. Ein weiterer Klick auf *close* schließt das Fenster wieder. Nun kann über *Start* (Abbildung 2.8) das eigentliche Flashen beginnen. Eine visuelle Rückmeldung bietet hierbei der Ladebalken *Progress*.

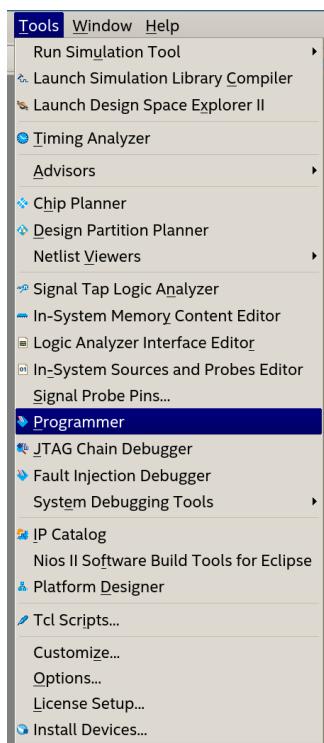


Abbildung 2.5: Öffnen des Programmers in Quartus

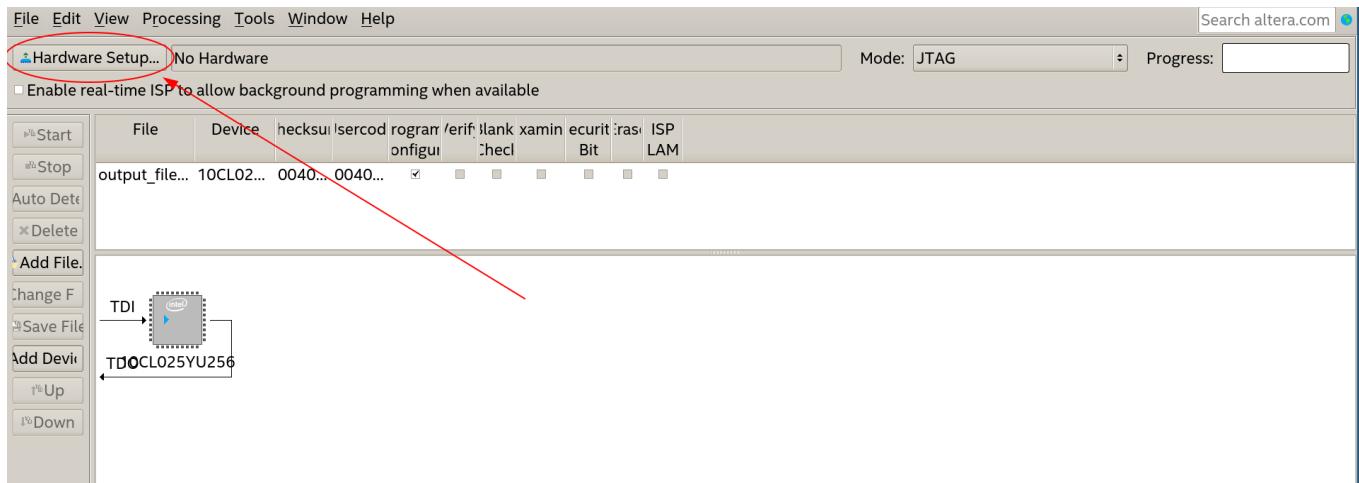


Abbildung 2.6: Einrichten der Hardware in Quartus

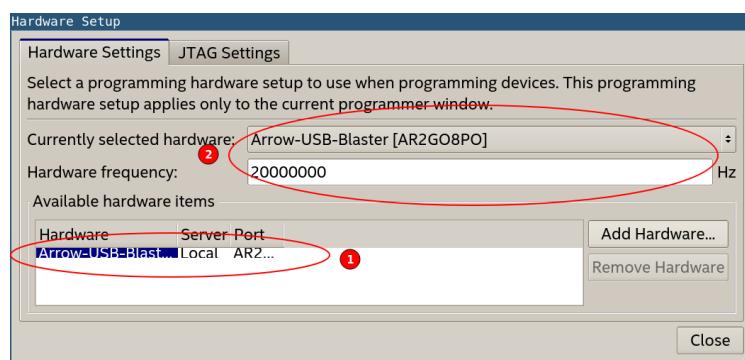


Abbildung 2.7: Einrichten der Hardware in Quartus

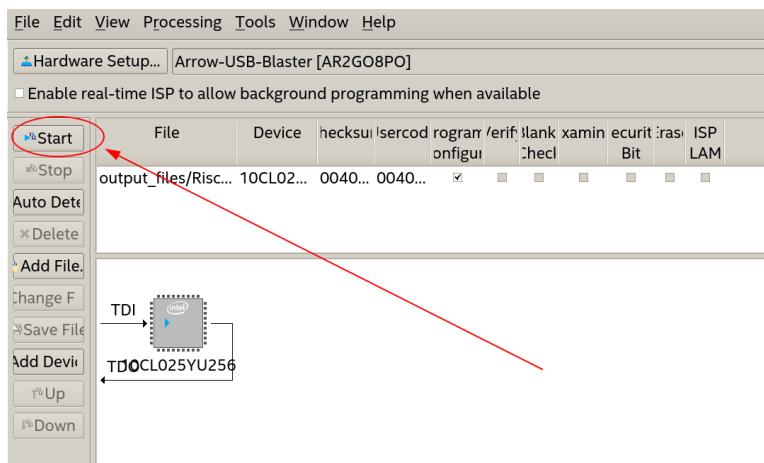


Abbildung 2.8: Starten des Flashens des FPGAs

2.4 Kompilieren des Programmcodes

Um Programmcode zu Compilieren muss zunächst die Toolchain, dessen Quellcode sich im *Firmware* Verzeichnis befindet, gebaut werden. Nach wechseln in das Verzeichnis kann über den Befehl `go build *.go` die Toolchain gebaut werden. War das bauen erfolgreich, erscheint eine ausführbare Datei namens *build* im Verzeichnis. Durch `./build -help` wird die Parameterübersicht des Programmes ausgegeben (Abbildung 2.9).

```
[0] giom@glap ~/r/p/R/Firmware (main) ~ ./build --help
Usage of ./build:
  -bootloader string
    Path to Bootloader code (default "crt0.s")
  -bytesize int
    Size of byte in bit (default 8)
  -linker string
    Path to linker Script File (default "linker.lds")
  -memorySize int
    Memory Size in Words (default 16384)
  -print
    Print ELF infos (default true)
  -source string
    Path to source code
  -wordsize int
    Size of word in bit (default 32)
```

Abbildung 2.9: Parameterübersicht von Build

Die Standardbelegung der Parameter ist auf die Hardware Eigenschaften des Softcores ausgelegt, sodass nur `./build -source=PFAD` benötigt wird um das jeweilige Programm zu Compilieren.

Zum testen des Bauprozesses befindet sich Beispielquellcode im *src* Verzeichnis. Beispielsweise kann über `./build -source=src/lightshift.c` ein einfaches Programm Compiliert werden. Da der *Print*-Parameter standardmäßig auf *true* gesetzt ist, werden nach Erfolgreichem compilieren die Partitionstabellen sowie der assemblierte Programmcode ausgegeben (Abbildung 2.10).

```
[0] giom@glap ~/r/p/R/Firmware (main) → ./build --source=src/counter.c
Partition: 1: 93 93 93 13 ef 6f 13 23 23 23 13 93 93 6f 93 b7 93 e3 93 63 93 6f 93 6f
Partition: 2: 05 95 e5 81 00 00 01 26 24 22 04 00 04 00 84 47 87 da 07 96 00 f0 80 f0
Partition: 3: f0 85 f5 c5 80 00 01 11 81 91 01 00 00 80 14 0f f7 97 f0 f0 00 df 10 5f
Partition: 4: 0f 00 0f ff 00 00 ff 00 00 00 01 00 00 00 00 23 fe 0f 00 00 fd 00 fd

Hex dump of section '.text':
0x00000000 9305f00f 93958500 93e5f50f 1381c5ff ..... .
0x00000010 ef008000 6f000000 130101ff 23261100 ....o.....#&..
0x00000020 23248100 23229100 13040101 93000000 #$..#".....
0x00000030 93040000 6f008000 93841400 b7470f00 ....o.....G..
0x00000040 9387f723 e3da97fe 9307f00f 6396f000 ...#.....c...
0x00000050 93000000 6ff0dff0 93801000 6ff05ffd ....o.....o._.

out/counter.elf:      file format elf32-littleriscv

Disassembly of section .text:

00000000 <_start>:
 0: 0ff00593          addi   a1,zero,255
 4: 00859593          slli   a1,a1,0x8
 8: 0ff5e593          ori    a1,a1,255
 c: ffc58113          addi   sp,a1,-4
10: 008000ef          jal    ra,18 <main>

00000014 <_end>:
14: 0000006f          jal    zero,14 <_end>

00000018 <main>:
18: ff010113          addi   sp,sp,-16
1c: 00112623          sw    ra,12(sp)
20: 00812423          sw    s0,8(sp)
24: 00912223          sw    s1,4(sp)
28: 01010413          addi   s0,sp,16
2c: 00000093          addi   ra,zero,0
30: 00000493          addi   s1,zero,0
34: 0080006f          jal    zero,3c <main+0x24>
38: 00148493          addi   s1,s1,1
3c: 000f47b7          lui    a5,0xf4
40: 23f78793          addi   a5,a5,575 # f423f <main+0xf4227>
44: fe97dae3          bge   a5,s1,38 <main+0x20>
48: 0ff00793          addi   a5,zero,255
4c: 00f09663          bne   ra,a5,58 <main+0x40>
50: 00000093          addi   ra,zero,0
54: fddff06f          jal    zero,30 <main+0x18>
58: 00108093          addi   ra,ra,1
5c: fd5ff06f          jal    zero,30 <main+0x18>
```

Abbildung 2.10: Beispielquellcode compiliert durch build

2.4.1 Plattform.h

Der Zugriff auf die Peripherie wird über *Memory-Mapping* realisiert. Die Adressen sind hierfür in der *Plattform.h* im *src*-Verzeichnis hinterlegt (Listing 2.1).

Listing 2.1: Plattform.h

```

1 #ifndef PLATTFORM_H
2 #define PLATTFORM_H
3
4 #include <stdint.h>
5
6 #define LED_ADDRESS 0x00010000
7 #define LED (*((volatile uint8_t *)(LED_ADDRESS)))
8
9 #define UART_RX_ADDRESS 0x00010001
10 #define UART_RX (*((volatile uint8_t *)(UART_RX_ADDRESS)))
11
12 #define UART_TX_ADDRESS 0x00010002
13 #define UART_TX (*((volatile uint8_t *)(UART_TX_ADDRESS)))
14
15 #define UART_STATUS_ADDRESS 0x00010003
16 #define UART_STATUS (*((volatile uint8_t *)(UART_STATUS_ADDRESS)))
17
18 #endif /* PLATTFORM_H */

```

Die Bedienung der Peripherie erfolgt aus dem Programmcode wie im Beispiel von *lightsshift.c* (Listing 2.2).

Listing 2.2: Ligthshift.c

```

1 #include <stdbool.h>
2 #include "plattform.h"
3
4 const int SLEEP_CYCLES = 1000000;
5
6 int main() {
7     // Set LED to startvalue and set start direction
8     LED = 1;
9     bool direction_up = true;
10    // infinti loop
11    while (1) {
12        // Sleep
13        for (int i = 0; i < SLEEP_CYCLES; i++)
14            ;
15
16        if (direction_up) {
17            // Shift to left
18            LED <= 1;
19            // If maxvalue change direction
20            if (LED == 128) {
21                direction_up = false;
22            }

```

```

23     } else {
24         // Shift to right
25         LED >>= 1;
26         // If minvalue change direction
27         if (LED == 1) {
28             direction_up = true;
29         }
30     }
31 }
32 return 0;
33 }
```

2.5 Flashen des Softcores

Zunächst müssen die generierten *MIFs* aus dem *out* Verzeichnis in *RISC-V/CPU/Design/src/memory* kopiert werden. Als nächstes wird über den Menüpunkt *Processing, Update Memory Initialization File* angeklickt (Abbildung 2.11). Daraufhin muss im selben Menüpunkt das Untermenü *Start* geöffnet werden um *Start Assembler* anzuklicken (Abbildung 2.12). Als letztes muss der FPGA erneut geflashed werden (Siehe 2.3). Dabei wird jedoch nur der letzte Schritt benötigt (Abbildung 2.8).

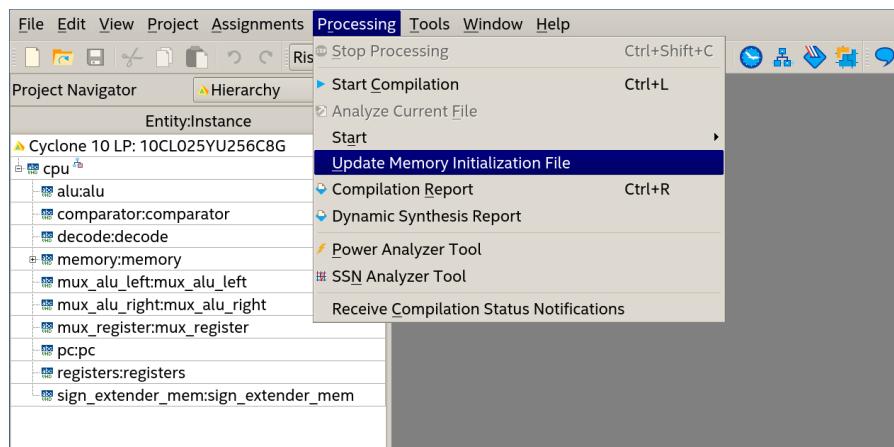


Abbildung 2.11: MIF Aktualisieren

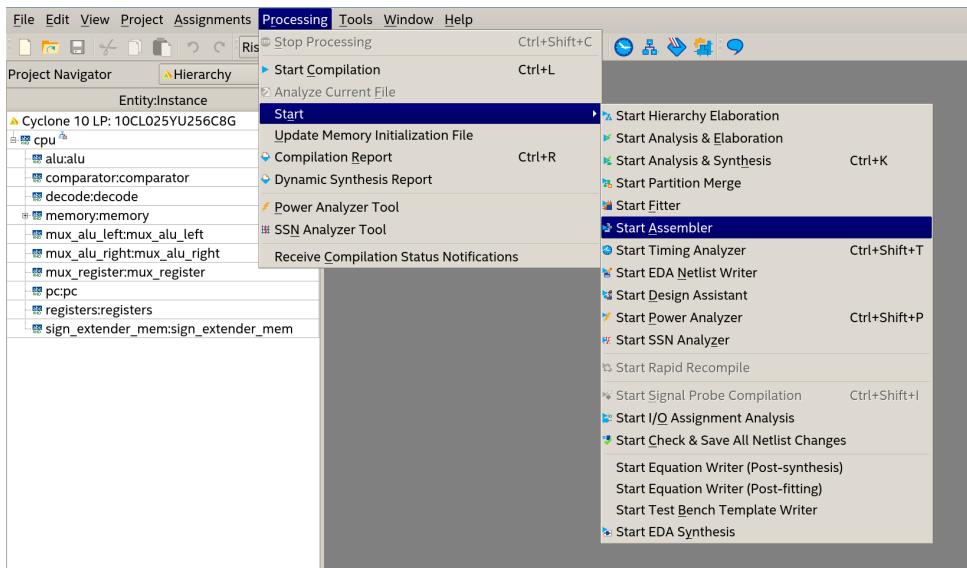


Abbildung 2.12: Assemblierung

2.6 Zurücksetzen

Der Programmcode kann durch einen drücken des Tasters neben den LEDs (Abbildung 2.13 Nr. 11) gestoppt werden. Durch ein loslassen dieses Taster beginnt der Programmcode wieder von vorne zu laufen.

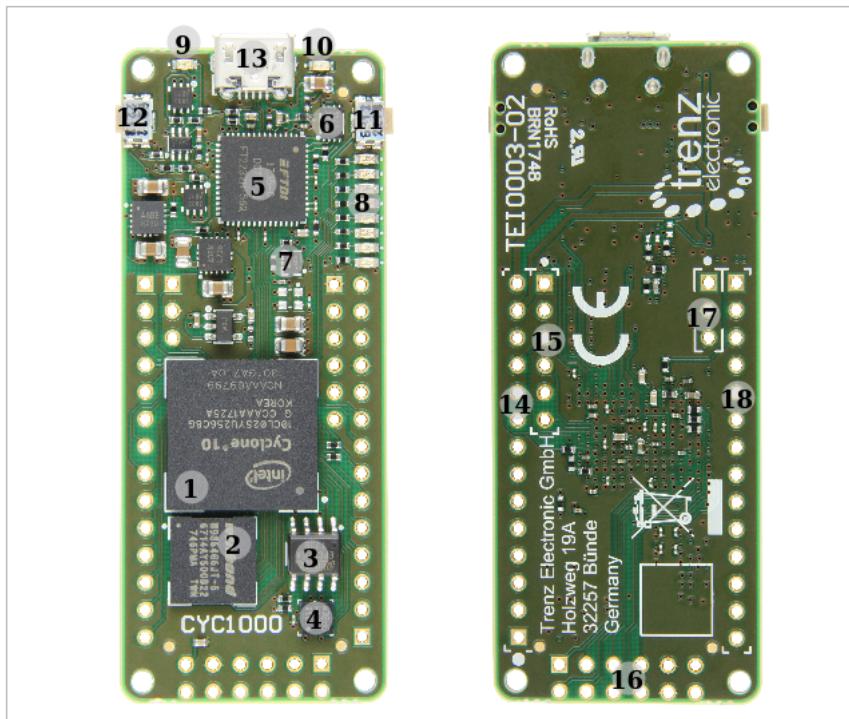


Abbildung 2.13: Platine des TEI0003 TRM FPGA-Board [Ele]

Teil II

Entwicklerhandbuch

Kapitel 3

Motivation

3.1 RISC-V

RISC-V ist eine offene und erweiterbare Befehlssatzarchitektur (ISA) die sich an dem *RISC (Reduced Instruction Set Computer)* Designprinzip orientiert. Dank der freizügigen *BSD-Lizenz* ist es, im Gegensatz zu bspw. der *x86 ISA* von *Intel*, jedem erlaubt *RISC-V* Mikroprozessoren zu entwerfen, herzustellen und zu verkaufen. Die Lizenz erlaubt es zusätzlich den Befehlssatz nach belieben zu erweitern und somit optimal an eine Hardwarearchitektur anzupassen. In Tabelle 3.1 werden die Grundbefehlssätze von RISC-V dargestellt. Darüber hinaus besitzt RISC-V noch zusätzliche Befehlssätze die z.B. Hardwaremultiplikation oder Fließkomma-Arithmetik erlauben. [ris]

Name	Beschreibung	Version
RV32I	32Bit Integer Basisbefehlssatz mit 32 Registern	2.1 (Ratifiziert)
RV32E	32Bit Integer Basisbefehlssatz mit 16 Registern	1.9 (Offen)
RV64I	64Bit Integer Basisbefehlssatz	2.1 (Ratifiziert)
RV128I	128Bit Integer Basisbefehlssatz	1.7 (Offen)

Tabelle 3.1: RISC-V Grundbefehlssätze

3.2 FPGA (Field Programmable Gate Array)

Ein FPGA, (Field Programmable Gate Array) ist ein integrierter Schaltkreis (IC) der Digitaltechnik, in welchen eine logische Schaltung geladen werden kann. Im Vergleich zu der Programmierung von Computern oder Mikrocontrollern wird die Schaltungsstruktur eines FPGAs durch eine Hardwarebeschreibungssprache (z.B. VHDL) beschrieben. Man spricht daher auch von der Konfiguration des FPGA. Ohne diese hat der Baustein keine Funktion. [wika]

Durch einen FPGA ist es somit möglich einen *RISC-V* Befehlssatz in VHDL zu formulieren und auf Hardware zu testen, ohne Kosten für eine Fertigung des Chips aufzubringen.

Kapitel 4

Entwicklerkonfiguration

Um mit dem Softcore zu Entwickeln wird folgende Soft- und Hardware benötigt.

4.1 Software

Software	Version	Quelle
Quartus Prime Lite	20.1.1	Intel
Modelsim Intel FPGA	20.1	Intel
Arrow USB Programmer	2.4.1-1	Trenz Electronics
RISC-V GCC Toolchain	9.2.0	RISC-V Organisation
Golang Toolchain	1.16.2	RISC-V Organisation

Tabelle 4.1: Benötigte Software für Inbetriebnahme

4.2 Hardware

Hardware	Version	Quelle
TEI0003 - CYC1000	0.2	Trenz Electorincs
Micro-USB-Kabel	2.0	

Tabelle 4.2: Benötigte Hardware für Inbetriebnahme

Kapitel 5

Probelmanalyse

In dem Laborprojekt soll ein Softcore entwickelt werden der den *RV32I* Befehlssatz implementiert. Dieser soll auf einem *FPGA*-Board hochgeladen werden und Programmcode ausführen können. Zusätzlich soll durch Simulationstests sowie durch Tests durch Programmcode die Korrektheit bewiesen werden. Die Aufgabenstellung kann in Soft- und Hardware unterteilt werden.

5.1 Software

Aus Softwaresicht wird eine Toolchain benötigt die Programmcode compiliert und in ein Dateiformat bringt, welches vom Softcore interpretiert werden kann.
Dies wird im folgendem Kapitel 7 behandelt.

5.2 Hardware

Aus Hardwaresicht wird ein *FPGA*-Board benötigt welches eine Möglichkeit bietet den in *VHDL* modellierten Softcore auf das *FPGA*-Board sowie Programmcode in den Softcore zu laden.

Die Wahl fällt auf das *FPGA*-Board *TEI0003 TRM* von *Trenz Electronic*. Herz des Boards ist der *Cyclone 10LP 10CL025 FPGA SoC* von Intel der an einem *12Mhz* Oszillator angeschlossen ist. Als Speicher stehen 66 *M9K*-Blöcke [Intb][2.1 Tabelle 2] mit jeweils 8.192 Bit [Intb][2.2] ($66 * 8192 / 8 = 67584B$) des *Cyclone 10LP* sowie 8MB des *TEI0003-02* zu Verfügung. Abbildung 5.1 zeigt den schematischen Aufbau des *TEI0003 TRM*.

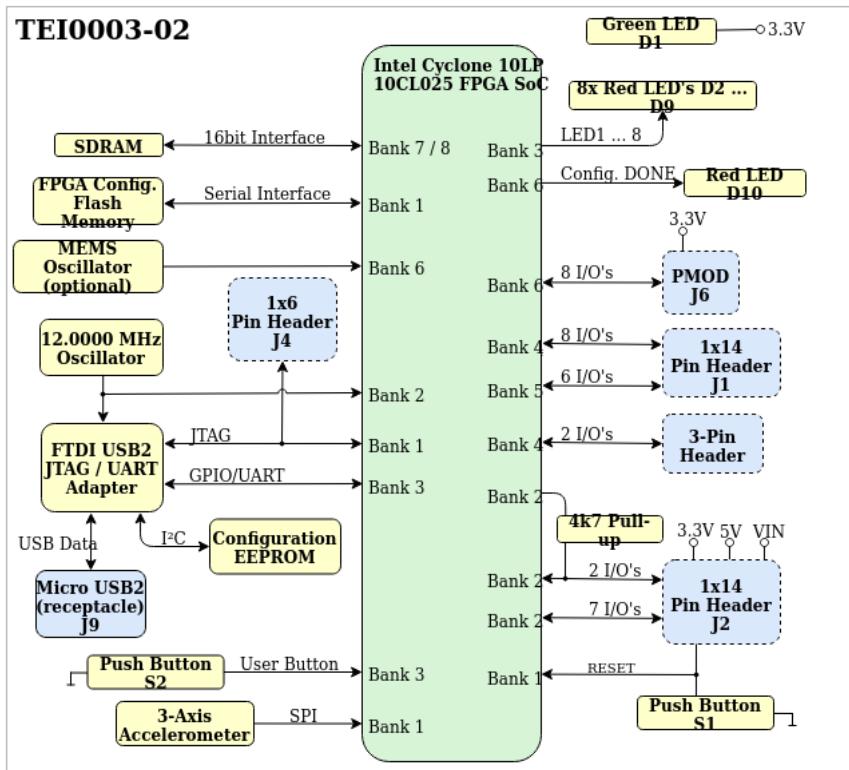


Abbildung 5.1: Schaubild des TEI0003 TRM FPGA-Board [Ele]

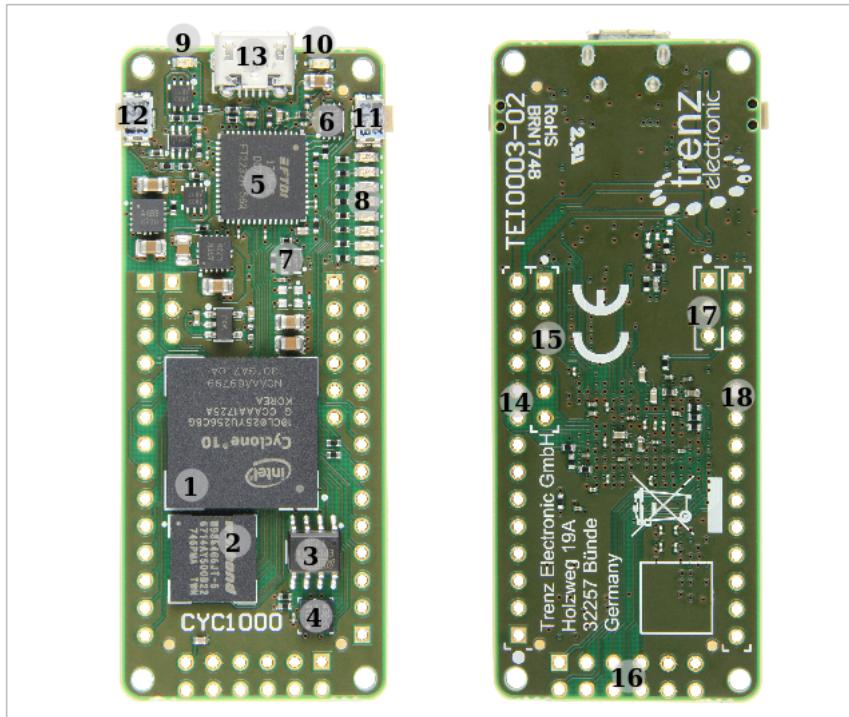


Abbildung 5.2: Platine des TEI0003 TRM FPGA-Board [Ele]

1. Intel Cyclone 10LP 10CL025 FPGA SoC, U1
2. 8 Mbyte SDRAM 166MHz, U2
3. 2 MByte serial configuration memory, U5
4. ST Microelectronics LIS3DH 3-axis accelerometer, U4
5. FTDI USB2 to JTAG/UART adapter, U3
6. Configuration EEPROM for FTDI chip, U9
7. 12.0000 MHz oscillator, U7
8. 8x red user LEDs, D2 ... D9
9. Red LED (Conf. DONE), D10
10. Green LED (indicating supply voltage), D1
11. Push button (user), S2
12. Push button (reset), S1
13. Micro USB2 B socket (receptacle), J9
14. 1x14 pin header (2.54mm pitch), J2
15. 1x6 pin header (2.54mm pitch), J4
16. 2x6 Pmod connector, J6
17. 3-pin header (2.54mm pitch), J3
18. 1x14 pin header (2.54mm pitch), J1

5.3 RV32I Befehlssatz

Der *RV32I* ist eine *Load and Store* Architektur und kann somit nur mit Lade- bzw. Speicherbefehlen auf den Speicher zugreifen. Der Prozessor arbeitet nur auf den 32 Registern die zuvor mit Daten aus dem Speicher geladen werden müssen. Dabei bietet *RV32I*, 32 Bit weite Register und kann nur Integerarithmetik in Hardware ausführen.

RV32I bietet die Basis für alle *RISC-V* Befehlssätze, da jede Erweiterung zumindest diesen Befehlssatz implementieren muss. Für den zu entwickelnden Softcore wurde somit der unprivilegierte *RV32I* Befehlssatz gewählt. Außer Acht gelassen werden jedoch die *FENCE* und *EBREAK* Befehle, da diese für den aktuellen Gebrauch des Softcores nicht relevant sind.

Tabelle 5.1 zeigt die verschiedenen Typen des *RV32I* Befehlssatzes.

Typ	31-25	24-20	19-15	14-12	11-7	6-0
R-Type	funct7	rs2	rs1	funct3	rd	opcode
I-Type	imm[11:0]		rs1	funct3	rd	opcode
S-Type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B-Type	imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode
U-Type		imm[31:12]			rd	opcode
J-Type	imm[20 10:1 11 19:12]				rd	opcode

Tabelle 5.1: RV32I Befehlssatztypen [ris]

R-Type sind arithmetische und logische Befehle

I-Type sind Immediate-, Lade- sowie relative Sprungbefehle

S-Type sind Speicherbefehle

B-Type sind Branchbefehle

J-Type sind absolute Sprungbefehle

5.4 Execution environment interface (EEI)

Die *Execution environment interface (EEI)* stellt eine Schnittstelle zur Laufzeitumgebung dar und definiert Initialzustände aber auch Attribute die z.B. den Speicher betreffen [ris][1.2]. Der Befehlssatz lässt viel Spielraum zu, sodass viele Werte frei gewählt werden können. Da der Softcore von Grund auf entwickelt wird, können viele Parameter so gewählt werden, dass sie zur Architektur passen.

5.5 Softcore Design

5.5.1 Mikroarchitektur

Von-Neumann-Architektur (VNA)

Die nach *John von Neumann* benannte Mikroarchitektur *Von-Neumann-Architektur (VNA)* bietet eine Grundlage für die Arbeitsweise der meisten heute bekannten Computer. Dabei ist charakteristisch, dass die Daten sowie das Programm im selben Speicher abgelegt sind und der Zugriff auf diese nur über den selben Bus stattfindet (Abbildung 5.3). Dies hat den Vorteil, dass *Race Conditions* sowie Daten-Inkohärenzen ausgeschlossen werden können. Ein wesentlicher Nachteil dieses Ansatzes ist der sogenannte *Von-Neumann-Flaschenhals*. Dieser entsteht dadurch, dass die Instruktionen nicht zur gleichen Zeit gelesen, wie Daten geschrieben bzw. gelesen werden können. Wenn beispielsweise ein Ladebefehl aus dem Programmspeicher geladen wird, beinhaltet dieser die Adresse aus der die eigentlichen

Daten ausgelesen werden sollen. Nun kann der Adressbus nicht zur gleichen Zeit die Instruktion sowie die Daten ansprechen. Das Auslesen der Daten erfolgt somit erst im nächsten Taktzyklus. Dadurch werden für die Lade- und Speicheroperationen immer zwei Zyklen verwendet, welches sich negativ auf die Performance auswirkt. Da es aus Sicht der Speichers keinen Unterschied zwischen Instruktion und Daten gibt, könnten theoretisch Daten als Instruktionen ausgelesen werden und somit von Schadcode ausgenutzt werden. Ein *Von-Neumann-Rechner* kann in folgenden Komponenten unterteilt werden.

ALU (Arithmetic Logic Unit) Das Rechenwerk führt arithmetische Operationen sowie logische Verknüpfungen durch.

Control Unit Das Steuerwerk decodiert die Befehle des Programmes, setzt die entsprechenden Steuerleitungen und regelt die Befehlsabfolge.

Bus Das Bussystem (Steuerbus, Adressbus und Datenbus) ist für die Kommunikation zwischen den einzelnen Komponenten verantwortlich.

Memory Der Speicher, speichert das eigentliche Programm sowie die Daten.

IO Das Ein- bzw Ausgabewerk steuert die Ein bzw. Ausgabedaten die zum Anwender sowie zu anderen Systemen führen.

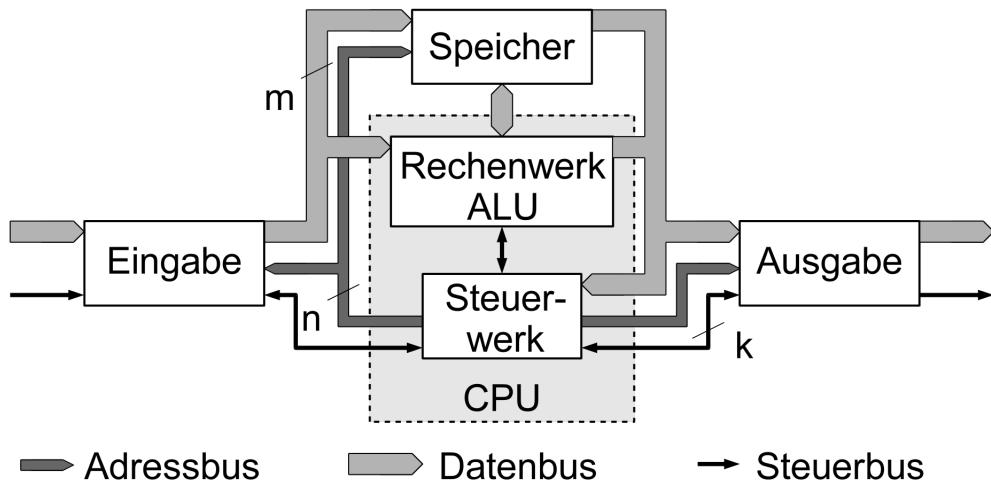


Abbildung 5.3: Von-Neumann-Architektur [wikd]

Harvard-Architektur

Eine weitere verbreitete Architektur ist die *Harvard-Architektur*. Grundlegend unterscheidet sich diese zur *Von-Neumann-Architektur* nur in ihrem Speicher und dem Bus. Die *Harvard-Architektur* verfolgt den Ansatz, Instruktionen und Daten physikalisch strikt voneinander zu trennen. Dabei werden zwei separate Speicher mit eigenem Adress- sowie Datenbus verwendet. Dies hat, im Gegensatz zur *Von-Neumann-Architektur*, den Vorteil,

dass Instruktionen und Daten in einem Taktzyklus gelesen bzw. geschrieben werden können. Hauptnachteil ist jedoch, dass es zu Speicherfragmentierung kommt, da weder nicht genutzter Programmspeicher als Datenspeicher noch umgekehrt, genutzt werden kann.

Für den zu entwickelnden Softcore wird eine *modifizierte Harvard-Architektur* verwendet, die die Vorteile der beiden oben genannten Architekturen vereint. Um die Speicherfragmentierung zu eliminieren, wird wie bei der *Von-Neumann-Architektur* nur ein Speicher verwendet. Jedoch werden wie bei der *Harvard-Architektur* zwei Bussysteme (Instruktionsbus und Datenbus) implementiert.

5.5.2 Befehlsverarbeitung

Von-Neumann-Zyklus

Die Phasen der Befehlsverarbeitung werden als *Von-Neumann-Zyklus* bezeichnet und bestehen aus folgenden fünf Teilschritten. Jede Phase benötigt unterschiedlich viel Zeit um sie zu durchlaufen. Dabei ist zu beachten, dass nicht jede Instruktion alle Phasen durchlaufen muss. Zum Beispiel ist ein absoluter Sprung schneller abgearbeitet als eine arithmetische Operation, die zunächst die Operanden aus den Registern laden, verarbeiten und anschließend zurück in die Register schreiben muss.

Fetch Der Befehl wird aus dem Speicher geladen.

Decode Der Befehl wird dekodiert und die Steuerleitungen werden gesetzt.

Fetch Operands Die Operanden für die ALU werden geladen

Execute Das Rechenwerk verarbeitet die Operanden

Writeback Das Ergebnis wird in den Speicher zurück geschrieben

Cycles per Instruction (CPI)

Cycles per Instruction (CPI) ist ein Maß zur Beurteilung der Performanz eines Prozessors und sagt aus wie viele Taktzyklen benötigt werden um eine Instruktion abzuarbeiten. Je kleiner der CPI Wert ist desto performanter kann ein Prozessor eingeschätzt werden.

$$CPI = \frac{\text{Taktzyklen}}{\text{Instruktion}} \quad (5.1)$$

Dabei wird ein Prozessor mit einem CPI Wert größer als eins ($CPI > 1$) als subskalar, mit einem Wert gleich eins ($CPI = 1$) als skalar und mit einem Wert kleiner als eins ($CPI < 1$) als superskalarer Prozessor bezeichnet.

Ein-Zyklus-Prozessor

Als Ein-Zyklus-Prozessor wird ein Prozessor verstanden, der alle Phasen des *Von-Neumann-Zyklus* in einem Taktzyklus abarbeitet. Somit ergibt sich ein CPI-Wert von eins ($CPI = 1$)

und der Prozessor kann als *Skalar* bezeichnet werden. Die maximal Taktfrequenz bzw. die minimale Taktzeit ist dabei direkt abhängig von der Signallaufzeit der längsten Instruktion.

$$Taktzeit > Signallaufzeit_{Gesamtbefehl} \quad (5.2)$$

Wesentlicher Vorteil dieser Architektur ist die einfache Implementierung, da keine zusätzliche Logik benötigt wird.

Pipelining-Prozessor

Im Gegensatz zum Ein-Zyklus-Prozessor steht der Pipelining-Prozessor. Statt eines gesamten Befehls wird während eines Taktzyklus nur jeweils eine Teilaufgabe abgearbeitet. Dabei können Teilaufgaben mehrerer Befehle gleichzeitig abgearbeitet werden. Eine Teilaufgabe kann dabei z.B. eine *Von-Neumann-Phase* sein oder noch feiner granuliert werden. Da eine Teilaufgaben eine kürzere Signallaufzeit aufweist als der Gesamtbefehl, kann die Taktzeit kürzer sein als die Signallaufzeit des Gesamtbefehls.

$$\begin{aligned} Taktzeit &< Signallaufzeit_{Gesamtbefehl} \\ Taktzeit &> Signallaufzeit_{Teilaufgabe} \end{aligned} \quad (5.3)$$

Die Taktzeit ist somit nur noch abhängig von der Signallaufzeit der Teilaufgabe die am längsten dauert. Die Teilaufgaben eines Befehls werden auch *Pipeline-Stages* genannt. Abbildung 5.4 zeigt eine vierstufige Befehlspipeline und der daraus resultierende erhöhte Gesamtdurchsatz. Wesentlicher Nachteil von Pipelining sind jedoch die Konflikte (*Pipeline-Hazards*) die dabei auftreten können. Dabei können folgende drei Konfliktarten auftreten.

Ressourcenkonflikte wenn eine Stufe der Pipeline, Zugriff auf eine Ressource benötigt, die bereits von einer anderen Stufe belegt ist

Datenkonflikte wenn ein Befehl, der sich in der Pipeline befindet, abhängig von einem Befehl ist, der in sich in einer der nächsten Pipelinestufen befindet

Kontrollflusskonflikte wenn die Pipeline abwarten muss, ob ein bedingter Sprung ausgeführt wird oder nicht

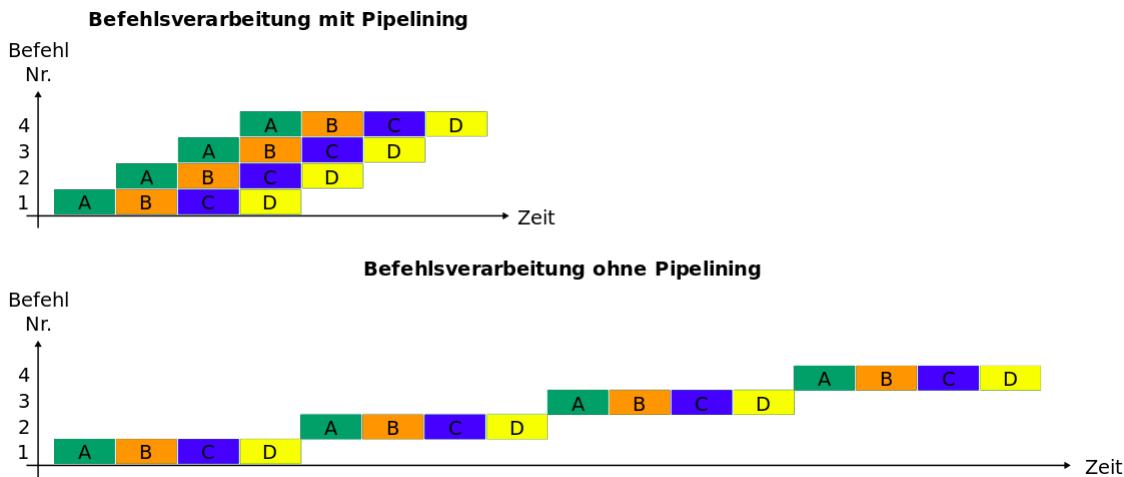


Abbildung 5.4: Befehlsverarbeitung mit und ohne Pipelining [wikic]

A: Befehlscode laden (IF, Instruction Fetch)

B: Instruktion dekodieren und laden der Daten (ID, Instruction Decoding)

C: Befehl ausführen (EX, Execution)

D: Ergebnisse zurückgeben (WB, Write Back)

Es ist jedoch zu beachten, dass Pipelining nur von Vorteil ist, wenn die Taktfrequenz, aufgrund der höheren Signallaufzeiten in einem Ein-Zyklus-Prozessor, nicht weiter erhöht werden kann. Da das gewählte *FPGA*-Board nur mit 12MHz taktet, ist eine Pipelining-Architektur nicht von Vorteil für den Softcore, sodass eine Ein-Zyklus-Architektur gewählt werden kann um die Softcore zu implementieren. Dies hat zusätzlich den Vorteil, dass weniger Logikblöcke des *FPGA*'s benutzt werden.

5.5.3 Speicher

Memory Wall

Als *Memory Wall* bezeichnet man das wachsende Ungleichgewicht zwischen der Prozessor- und der Speichergeschwindigkeit. Bei frühen Computern war der Prozessor die langsamste Einheit des Rechners [McK95]. Die Datenbereitstellungszeit stellte somit nur einen geringen Anteil an der Verarbeitungszeit dar. Seit den 1980 Jahren begann jedoch der Prozessor exponentiell schneller zu werden als der Speicher [Car02]. Es kann davon ausgegangen werden, dass jede fünfte Instruktion den Speicher beansprucht [McK95]. Somit würde der Prozessor 20% der Zeit auf Daten warten. Ein idealer Speicher würde so schnell wie der Prozessor sein und die Daten in einem Taktzyklus liefern. Wie Abbildung 5.5 zeigt, wäre solch ein Speicher sehr teuer. Eine Maßnahme um der *Memory Wall* entgegen zu wirken ist der Einsatz von Cachespeicher direkt im Prozessor.

Auf dem *FPGA*-Board stehen die *M9K*-Blöcke sowie *SDRAM* als Speicher zu Verfügung (Siehe 5.2). Dabei wird der *SDRAM* über eine 16 Bit Interface angesprochen. Der *RV32I* Befehlssatz ist jedoch 32 Bit breit. Dadurch werden zwei Taktzyklen benötigt um eine Instruktion aus dem Speicher zu laden und würde dadurch den Softcore ausbremsen. Um das Problem der *Memory Wall* zu lösen werden die *M9K*-Blöcke des *FPGA*'s als Speicher benutzt. Diese können in der Breite frei variiert werden und ermöglichen den Zugriff auf die Daten in einem Taktzyklus. Wesentlicher Nachteil dieser Lösung ist, dass nur *67584B* Speicher für Daten und Instruktionen zu Verfügung stehen.

Computer Memory Hierarchy

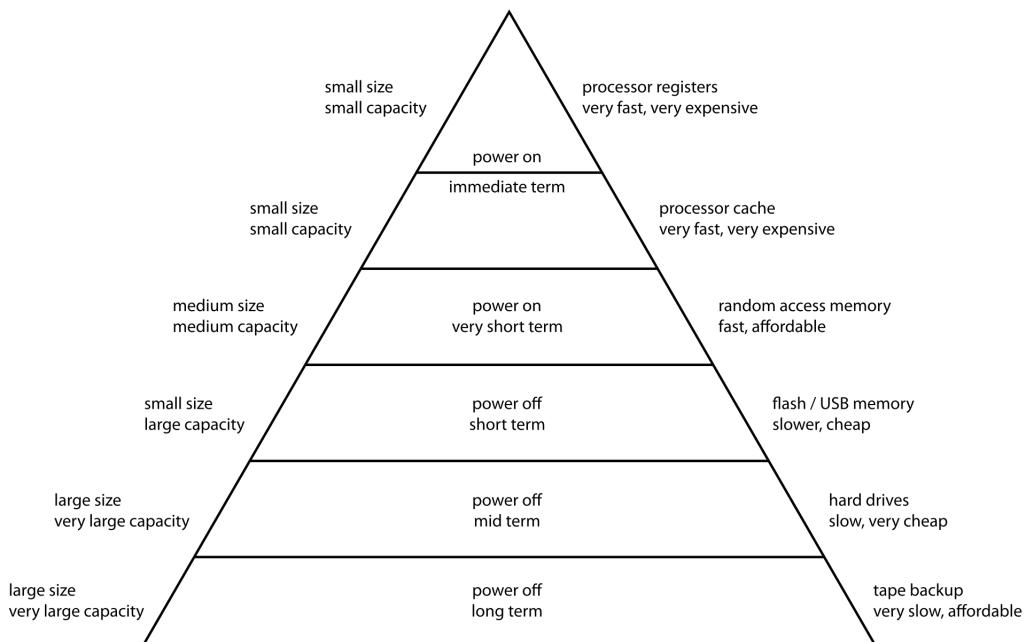


Abbildung 5.5: Speicherpyramide [wikib]

Speicherausrichtung

Da eine *modifizierte Harvard-Architektur* gewählt wurde (Siehe 5.5.1), liegen im Speicher sowohl Daten als auch Instruktionen. Die Instruktionen sind hierbei in vier Byte Grenzen zu halten (natürliche Ausrichtung) [ris][2.2]. Der Befehlssatz lässt es jedoch der EEI (Siehe 5.4) überlassen, ob die Daten auch natürlich ausgerichtet liegen sollen [ris][2.6]. Der Vorteil von natürlich ausgerichteten Daten liegt darin, dass keine zusätzliche Logik benötigt wird um Daten im Speicher auszulesen bzw. zu schreiben. Wesentlicher Nachteil ist die Speicherfragmentierung, da im Worst-case 32 Bit für einen 8 Bit Wert verbraucht werden. Die ungenutzten 24 Bit ($32\text{Bit} - 8\text{Bit} = 24\text{Bit}$) werden mit Nullen aufgefüllt. Der Einfachheit halber wird der Ansatz des natürlich ausgerichteten Speichers gewählt.

Kapitel 6

Realisation in VHDL

Folgendes Kapitel gibt eine Übersicht über die implementierten Einheiten (Entitys) in VHDL.

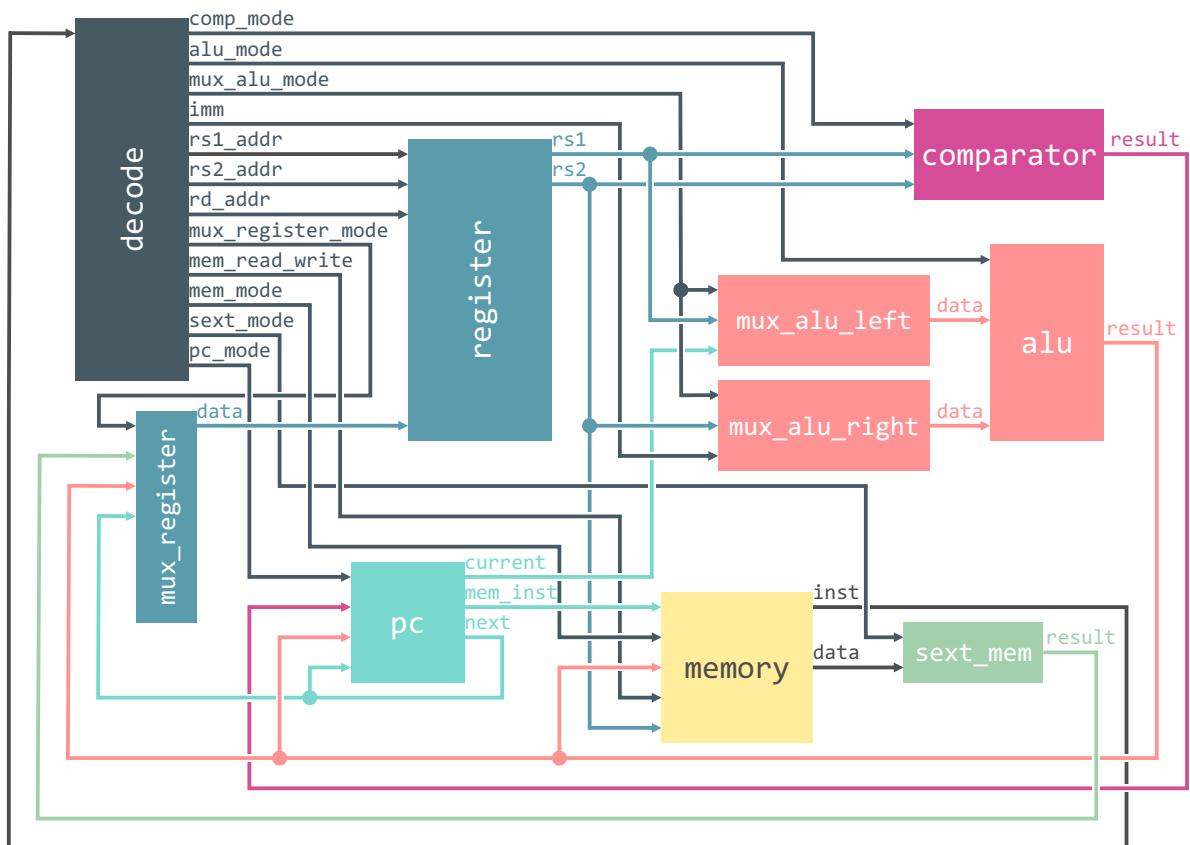


Abbildung 6.1: Gesamtübersicht des Softcores

6.1 Steuerwerk

Die Dekodiereinheit wird im Verbund mit dem *Programm Counter (PC)* (Siehe 6.1.2) auch Steuerwerk genannt.

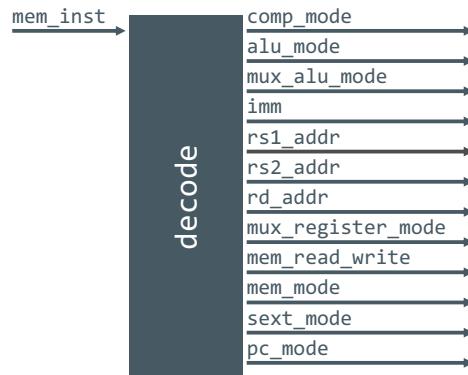


Abbildung 6.2: Dekodiereinheit

6.1.1 Dekodiereinheit

Die Dekodiereinheit ist dafür zuständig die Instruktionen aus dem Speicher zu interpretieren und die notwendigen Steuerleitungen zu setzen. Hierfür ist die Einheit direkt über den 32 Bit Instruktionbus mit dem Speicher verbunden.

6.1.2 Programm Counter (PC)

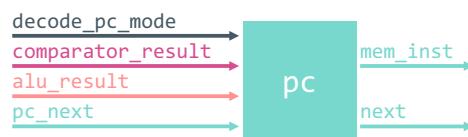


Abbildung 6.3: Befehlszählers

Der Befehlszähler steuert den Programmablauf und speichert in einem 32 Bit Register die aktuelle Adresse der Instruktion im Programspeicher. In jedem Taktzyklus wird die Adresse um ein Wort (32 Bit) erhöht und zeigt somit auf den nächsten Befehl. Dieser kann wieder von der Dekodiereinheit dekodiert werden und es können die Steuerleitungen gesetzt werden. Der Befehlszähler kann jedoch nicht nur inkrementiert sondern auch überschrieben werden.

Listing 6.1: Prozess des Befehlszählers

```

1  process (i_clock, i_reset, i_mode) begin
2      if i_reset = '0' then
3          pc_register <= "11111111111111111111111111111100";
4      elsif rising_edge(i_clock) then
5          if i_mode = PC_SRC_ALU or (i_mode = PC_SRC_COMP_ALU and
6              i_comp = '1') then
7              pc_register <= i_src_alu;
8          else
9              pc_register <= i_src_next;
10         end if;
11     end process;

```

6.1.3 Branch



Abbildung 6.4: Vergleichswerk

Wenn eine Anweisung zur Ablaufsteuerung auf Programmebene ausgeführt werden soll, muss zunächst die Bedingung geprüft werden und dann ggf. eine Instruktion ausgeführt werden, die nicht sequentiell hinter der aktuellen im Speicher liegt. Dies nennt man einen Zweig (englisch: Branch). Ein Zweig erfordert zusätzliche Logik in Steuerwerk und wird durch eine Vergleichseinheit realisiert. Diese Einheit vergleicht zwei 32 Bit Daten (Links und Rechts) und liefert ein boolesches Ergebnis (ein Bit). Tabelle 6.1 zeigt welche Vergleichsoperatoren zu Verfügung stehen.

Zustand	Operation
COMP_EQUAL	signed(links) == signed(rechts)
COMP_NOT_EQUAL	signed(links) != signed(rechts)
COMP_LESS_THEN	signed(links) < signed(rechts)
COMP_GREATER_EQUAL	signed(links) >= signed(rechts)
COMP_LESS_THEN_U	unsigned(links) < unsigned(rechts)
COMP_GREATER_EQUAL_U	unsigned(links) >= unsigned(rechts)

Tabelle 6.1: Zustandstabelle Vergleichswerk

6.2 Register



Abbildung 6.5: Registereinheit

6.2.1 Registereinheit

Die Registereinheit bildet 32 Register mit einer Breite von 32 Bit ab [ris][2.1]. Dabei kann in einem Taktzyklus auf zwei Register (rs1 und rs2) lesend und auf einem Register (rd) schreibend zugegriffen werden. Die *ISA* besagt zudem, dass Register x0 immer Nullen liefern soll und nicht überschrieben werden darf [ris][2.1]. Dies wird durch eine Abfrage der Adresse im Prozess gesteuert. Falls das Zielregister, das Nullregister sein sollte werden die Werte nicht in das Zielregister übernommen.

Listing 6.2: Prozess der Registereinheit

```

1  process (i_clock, i_reset) begin
2      if i_reset = '0' then
3          register_table <= (others => (others => '0'));
4      elsif rising_edge(i_clock) then
5          if i_write_register_address /= "00000" then
6              register_table(to_integer(unsigned(i_write_register_address))) <=
7                  i_write_register_data;
8          end if;
9      end if;
10 end process;

```

6.2.2 Register Multiplexer



Abbildung 6.6: Register Multiplexer

Die Eingangsdaten werden instruktionsabhängig ausgewählt und in das Zielregister (rd) geschrieben. Dies geschieht in einem vorgesetztem Multiplexer. Die Dekodiereinheit

setzt dabei die notwendigen Steuerleitungen. Als Auswahlmöglichkeiten der Daten stehen die *ALU*, der Speicher, der nächste Befehlszähler (*PC + 4*). Als gesonderte Absicherung dient der *MUX_REG_ZERO* Zustand. Dieser wird standardmäßig beim initialisieren gesetzt und soll ein Überschreiben von Daten verhindern wenn z.B. eine unbekannte Instruktion ausgeführt werden soll.

Listing 6.3: Prozess des register Multiplexers

```

1 process (i_mode, i_alu, i_mem, i_pc) begin
2   case i_mode is
3     -- Write ALU to register
4     when MUX_REG_ALU => o_data <= i_alu;
5     -- Write memory content to register
6     when MUX_REG_MEM => o_data <= i_mem;
7     -- Write programm counter +4 to register
8     when MUX_REG_PC => o_data <= i_pc;
9     -- Do nothing
10    when MUX_REG_ZERO => o_data <= (others => '0');
11  end case;
12 end process;

```

6.3 Arithmetic logic unit (ALU)

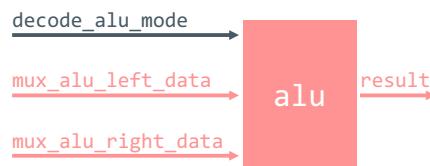


Abbildung 6.7: ALU

6.3.1 Rechenwerk

Die *Arithmetic logic unit (ALU)* ist das Rechenwerk und berechnet arithmetisch sowie logische Operationen. Dabei werden nicht nur Register-Register Berechnungen sondern auch Immediate- sowie Sprungberechnungen basierend auf dem aktuellen Befehlszähler durchgeführt. Da der Befehlssatz nur Integer Operationen erlaubt können auch nur diese in Hardware umgesetzt werden [ris][2.4]. Ebenfalls fehlen der *ISA* Multiplikation sowie Division. Diese können über Softwarebibliotheken mit Addition und Subtraktion durchgeführt werden, benötigen jedoch mehr Instruktionen, verbrauchen dadurch mehr Programmspeicher und schlagen sich somit negativ auf die Performanz aus.

6.3.2 Rechenwerk Multiplexer

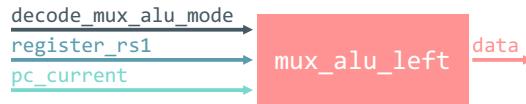


Abbildung 6.8: Linker ALU Multiplexer

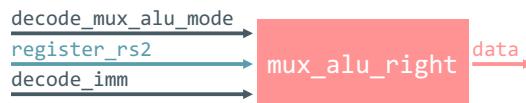


Abbildung 6.9: Rechter ALU Multiplexer

Die beiden Rechenwerk Multiplexer (Links und Rechts) steuern die Eingangsdaten des Rechenwerks und bestimmen somit welche Daten berechnet werden. Die Steuerleitungen werden dabei instruktionsabhängig in der Dekodiereinheit gesetzt. Beide Multiplexer teilen die selben Zustände, reagieren jedoch anders. Tabelle 6.2 zeigt die verschiedenen Zustände und die damit verbundenen Eingangsdaten für die *ALU*.

Zustand	Linker Operand	Rechter Operand	Berechnung
MUX_ALU_RS1_RS2	rs1	rs2	Register-Register
MUX_ALU_RS1_IMM	rs1	immediate	Register-Immediate
MUX_ALU_PC_IMM	pc	immediate	Sprung
MUX_ALU_PC_RS2	pc	rs2	Sprung

Tabelle 6.2: Zustandstabelle ALU Multiplexer

Listing 6.4: Prozess des linken Rechenwerk Multiplexers

```

1 process (i_mode, i_rs1, i_pc) begin
2     case i_mode is
3         -- RS1 and RS2 are required -> rs1 is tunneled through
4         when MUX_ALU_RS1_RS2 => o_data <= i_rs1;
5             -- RS1 and immediate are required -> rs1 is tunneled through
6             when MUX_ALU_RS1_IMM => o_data <= i_rs1;
7                 -- Programm counter and Immediate are required -> PC is tunneled
                    through

```

```

8      when MUX_ALU_PC_IMM => o_data <= i_pc;
9      -- Programm counter and RS2 are required -> PC is tunneled through
10     when MUX_ALU_PC_RS2 => o_data <= i_pc;
11   end case;
12 end process;

```

Listing 6.5: Prozess des rechten Rechenwerk Multiplexers

```

1 process (i_mode, i_rs2, i_imm) begin
2   case i_mode is
3     -- RS1 and RS2 are required -> RS2 is tunneled through
4     when MUX_ALU_RS1_RS2 => o_data <= i_rs2;
5     -- Programm counter and RS2 are required -> RS2 is tunneled
       through
6     when MUX_ALU_PC_RS2 => o_data <= i_rs2;
7     -- RS1 and immediate are required -> Immediate is tunneled through
8     when MUX_ALU_RS1_IMM => o_data <= i_imm;
9     -- Programm counter and immediate are required -> Immediate is
       tunneled through
10    when MUX_ALU_PC_IMM => o_data <= i_imm;
11  end case;
12 end process;

```

6.4 Memory

Durch die gewählte *modifizierte Havard-Architektur* und den *Ein-Zyklus-Ansatz* muss sichergestellt werden, dass in einem Taktzyklus sowohl Instruktion gelesen als auch die damit verbundenen Daten gelesen bzw. geschrieben werden können. Dies wird dadurch ermöglicht, dass die Instruktionen zur steigenden Taktflanke gelesen und die Daten zur fallenden Taktflanke gelesen bzw. geschrieben werden. Dies ist nur möglich, da die Taktzeit mehr als doppelt so lang ist, wie die Zeit die benötigt wird um eine Speicheroperation durchzuführen [Intc][Tabelle 23].

$$\frac{1}{12Mhz} > 2 \cdot \frac{1}{238Mhz} \quad (6.1)$$

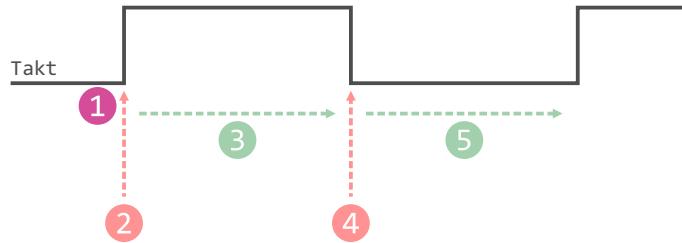


Abbildung 6.10: Taktflanken am Speicher

1. Neue Instruktionsadresse muss bereits anliegen
2. Instruktion wird aus Speicher gelesen / Steuerwerk setzt Kontrollsignale
3. Instruktion wird ausgeführt / Adresse für Daten wird berechnet.
4. Daten werden an berechneter Adresse geschrieben bzw. gelesen
5. Gelesene Daten werden durch ALU verrechnet und in Zielregister geschrieben. / Daten werden aus Register in Speicher geschrieben

6.4.1 Byte-Adressierung

Auch wenn die Speicherausrichtung natürlich ist (Siehe 5.5.3), fordert der Befehlssatz eine Byte-Adressierung, sodass auf einzelne Bytes in einem Wort zugegriffen werden kann. Leider erlauben das die *M9K* Speicherblöcke nicht. Aus diesem Grund wurden vier Speicherpartitionen erstellt die jeweils ein Viertel des gesamten Speichers ausmachen. Soll nur ein einzelnes Byte gelesen bzw. geschrieben werden, wird nur eine Speicherpartition adressiert. Bei einem *half-Word* werden hingegen zwei Partitionen ausgelesen. Dabei wird sichergestellt, dass nicht über die natürliche Ausrichtung hinweg adressiert werden kann. Eine Speicherpartition wird durch *memory_byte.vhd* repräsentiert und wird durch die *Megafuction* erstellt. Somit wird gewährleistet, dass die *M9K*-Blöcke des *FPGA*'s benutzt werden. In *memory_word.vhd* wird die Logik definiert, die zuständig ist um bei jeweiliger Adresse die richtige Speicherpartition zu adressieren. Für Instruktionen und Daten stehen insgesamt 65535 Bytes zu Verfügung.

6.4.2 Initialisierung

Die Initialisierung erfolgt über ein *Memory initialization files (MIF)* [Inta]. Die einfache *Key-Value*-Syntax (Tabelle 6.3) erlaubt eine Abbildung der Speicheradressen sowie deren korrespondierenden Daten. Die eigentliche Initialisierung erfolgt entweder zur Synthesesezeit oder zur Laufzeit über den Quartus Assembler (Siehe 2.5). Listing 6.6 zeigt ein Beispiel MIF.

Schlüsselwort	Wert
---------------	------

DEPTH	Speichergröße in Words
WIDTH	Wordgröße
ADDRESS_RADIX	Repräsentation der Adressen
DATA_RADIX	Repräsentation der Daten
CONTENT	Adress-Daten Paare

Tabelle 6.3: MIF Syntax

Listing 6.6: Beispiel MIF

```

1 DEPTH = 16384;
2 WIDTH = 8;
3 ADDRESS_RADIX = HEX;
4 DATA_RADIX = HEX;
5 CONTENT
6 BEGIN
7
8 0: 93;
9 1: 93;
10 2: 93;
11 3: 13;
12 [4..3fff]: 00;
13 END;

```

6.4.3 Sign-Extender



Abbildung 6.11: Vorzeichenerweiterung

Die Vorzeichenerweiterung wird verwendet um ausgelesene Werte aus dem Speicher auf Wordgröße zu erweitern. Grund dafür ist, dass Werte kleinerer Breite ausgelesen werden können wie z.B. ein Byte oder Halfword. Da die Register jedoch alle mit einer fixen Wordbreite von 32 Bit arbeiten muss ggfs. aufgefüllt werden. Ob mit Nullen oder Einsen aufgefüllt wird, ist abhängig von der Operation und wird über den *sext_mode* durch den Dekoder eingestellt.

6.5 IO

Die Anbindung von Peripherie erfolgt über *Memory-Mapping*. Dabei wird ein zusätzlicher Bereich im Speicher nur für Peripherie reserviert (Abbildung 6.12). Je nachdem welcher Bereich adressiert wird, werden Daten geschrieben bzw. gelesen oder die Peripherie angesteuert. Darüber entscheidet der Prozess *ram_or_ext* in der *memory.vhd* (Listing 6.7).

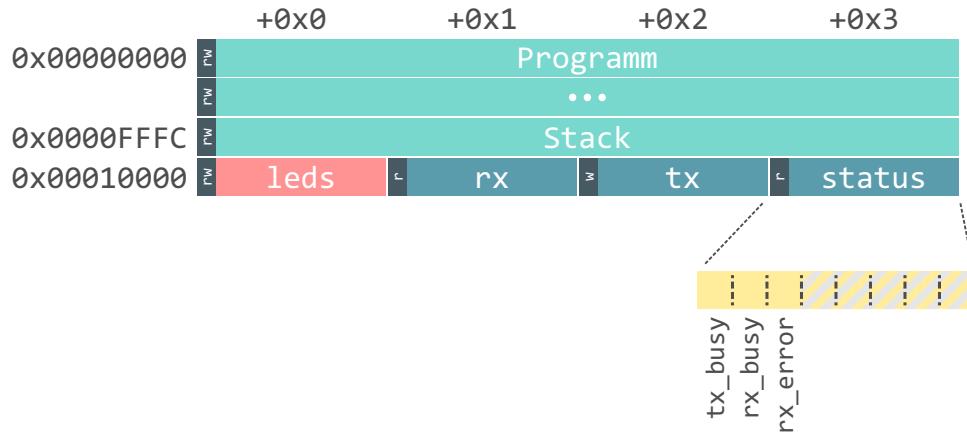


Abbildung 6.12: Speicher mit IO Bereich

Listing 6.7: Adressaufteilung im Speicher

```

1 -- RAM or in the extended area and sets the corresponding data.
2 ram_or_ext : process (i_data_address, i_data_read_write, ram_data,
3   ext_data) begin
4   ram_read_write <= MEM_DIR_READ;
5   ext_read_write <= MEM_DIR_READ;
6   o_data <= (others => '0');
7
8   -- Address is in RAM section
9   if i_data_address <= x"0000FFFF" then
10    o_data <= ram_data;
11    ram_read_write <= i_data_read_write;
12    -- Address is in extended Section
13  elsif i_data_address <= x"00010003" then
14    o_data <= ext_data;
15    ext_read_write <= i_data_read_write;
  end if;

```

Listing 6.8: Leseprozess für die Peripherie

```

1  read: process (address, io_register, uart_rx_data, uart_status) begin
2    case(address) is
3      when C_LEDS => o_data <= extend(io_register, 32);
4      when C_UART_RX => o_data <= extend(uart_rx_data, 32);
5      when C_UART_STATUS => o_data <= extend(uart_status, 32);
6      when others => o_data <= (others => '0');
7    end case;
8  end process;

```

Listing 6.9: Schreibprozess für die Peripherie

```

1  write: process (i_clock, i_data_store_mode, i_data, i_reset) begin
2    if i_reset = '0' then
3      io_register <= (others => '0');
4    elsif rising_edge(i_clock) then
5      if i_data_read_write = MEM_DIR_WRITE and address = C_LEDS then
6        io_register <= i_data(7 downto 0);
7      end if;
8    end if;
9  end process;

```

6.5.1 LED

Es stehen acht LED's auf dem *FPGA*-Board zur Verfügung die über den erweiterten Speicherbereich angesteuert werden können. Der Status der LED's kann hierbei gesetzt sowie abgefragt werden. Dazu dienen die beiden Prozesse in *peripherie.vhd* (Siehe Listing 6.8 und 6.9).

6.5.2 UART

Zusätzlich steht eine *UART*-Schnittstelle zu Verfügung die es erlaubt Byteweise Daten zu senden oder empfangen. Dabei wurde das Modul im Rahmen dieser Arbeit nicht entwickelt und ist somit ein Fremdmodul [Lar]. Es funktioniert jedoch nach dem gleichen Prinzip des *Memory-Mappings*. Wie Listing 6.8 und 6.9 zeigen, besteht es aus drei Adressbereichen (Status, RX und TX) aus denen gelesen bzw. in die geschrieben werden kann.

Kapitel 7

Toolchain

Das folgenden Kapitel beschreibt das Zusammenspiel verschiedener Softwarschichten um eine *Toolchain* abzubilden die in der Lage ist, für den Softcore passenden Compilate zu erstellen, Speicherpartitionen zu generieren sowie den Bootloader zu erstellen und zu linken. Dafür wurde im Rahmen dieses Projektes ein, in *GO* geschriebenes, Programm entworfen welches folgende Punkte in einem Programm vereint. Die Benutzung des Programmes wird in Kapitel 2.4 beschrieben.

7.1 Compiler

Der *Compiler* wird benötigt um Programmcode in *RISC-V* Maschinenbefehle zu übersetzen. Zusätzlich formt dieser die Maschineninstruktionen in ein *ELF*-Dateiformat (Siehe 7.1.1).

Dazu wird die offene und freie *GCC (GNU Compiler Collection)* verwendet. Das *RISC-V* Team hat hierfür schon vorarbeitet geleistet und bietet den kompletten Toolchain Quellcode an. Dies ermöglicht das Bauen des *Cross-Compilers* sowie von hilfreichen Zusatzprogrammen [Ris].

7.1.1 Executable and Linkable Format (ELF)

Das *Executable and Linkable Format (ELF)* definiert ein standard Binärformat für ausführbare Programme, Bibliotheken sowie von Speicherauszügen. Jede *ELF* besteht aus Kopfinformationen (Header) gefolgt von Daten. Die Kopfinformationen beinhalten Informationen über die eigentlichen Daten. Dazu zählen bspw. Wordbreite, Byte-Reihenfolge, Elf-Typ oder Maschinentyp.

Zu den Daten gehören unter anderem *Sections Headers*. Diese beinhalten dabei Informationen über mehrere Code-Sektionen und deren Zugriffsrechte. Die eigentlichen Daten liegen dabei unter *Data*. Die vier wichtigsten Sektionen für ausführbare Programme sind dabei folgende.

.text ausführbarer Programmcode (Instruktionen)

.data Initialisierte Daten mit Schreib- sowie Lesezugriff

.rodata Initialisierte Daten mit Lesezugriff (Konstanten)

.bss Uninitialisierte Daten (mit nullen initialisiert)

7.1.2 Linker

Der Linker ist, im Kontext dieser Arbeit, vor allem dafür zuständig die Speicheranordnung zu definieren. Dafür werden die *Sections* (Siehe 7.1.1) in eine Reihenfolge gebracht die für die Hardware passt. Listing 7.1 zeigt das Linkerskript welches die Speicheranordnung definiert. **ENTRY(_start)** definiert hierbei den Einstiegsplatz (Siehe 7.2). **MEMORY** beschreibt die Speichergröße, sodass der Compiler weiß wie viel Speicher ihm zu Verfügung steht. Aufgrund der gewählten *modifizierten Havard-Architektur* (Siehe 5.5.1) gibt es nur einen Speicherpool, sodass Instruktionen sowie Daten zusammen liegen. In **SECTIONS** wird die Reihenfolge der Instruktionen und der Daten innerhalb des Speicher definiert. Im ersten Bereich stehen die Instruktionen (**.text**) gefolgt von den Konstanten (**.rodata**). Als nächstes folgen die Variablen (**.data**) sowie die Null-initialisierten Variablen (**.bss**).

Listing 7.1: Linkerskript

```

1 ENTRY(_start)
2
3 MEMORY
4 {
5     MEM (wrx) : ORIGIN = 0x0, LENGTH = 65536
6 }
7
8 SECTIONS
9 {
10     .text :
11     {
12         *(.text*);
13     } > MEM
14
15     .rodata :
16     {
17         *(.rodata*);
18     } > MEM
19
20     .data :
21     {
22         *(.data*);
23     } > MEM
24
25     .bss :
26     {
27         *(.bss*);
28         *(COMMON);

```

```

29     } > MEM
30
31 }
```

7.2 Bootloader

Der Bootloader ist in Assembler geschrieben und ist dazu da den Softcore zu initialisieren, sodass dieser bereit ist, Programmcode auszuführen. Dabei wird hauptsächlich die Stackadresse gesetzt und die *Main* Funktion des C-Codes aufgerufen (Listing 7.2). Hierbei ist zu beachten, dass der Stack von hinten nach vorne wächst. Aus diesem Grund wird als Stackadresse das letzte adressierbare Byte (0xFFFF) benutzt.

Listing 7.2: Bootloader crt0.s

```

1 .text
2 .align 2
3 .globl _start
4 .type _start, @function
5 .org 0
6
7 _start:
8     # Create stack address 0xFFFF
9     addi a1,zero,0xFF
10    slli a1,a1,8
11    ori a1,a1,0xFF
12    # Set the stack address
13    addi sp, a1, -4
14    # Call the main function
15    jal ra, main
16 _end:
17    j _end
```

7.3 MIF Generierung

Die zugrunde liegende Speicherarchitektur (Siehe 6.4) und deren Initialisierung (Siehe 7.1.1) benötigt vier *MIF*'s die durch die Toolchain generiert werden. Hierfür wird, vergleichbar mit dem Round-Robin-Verfahren, jedes Byte eines Words in einer unterschiedliche MIF geschrieben.

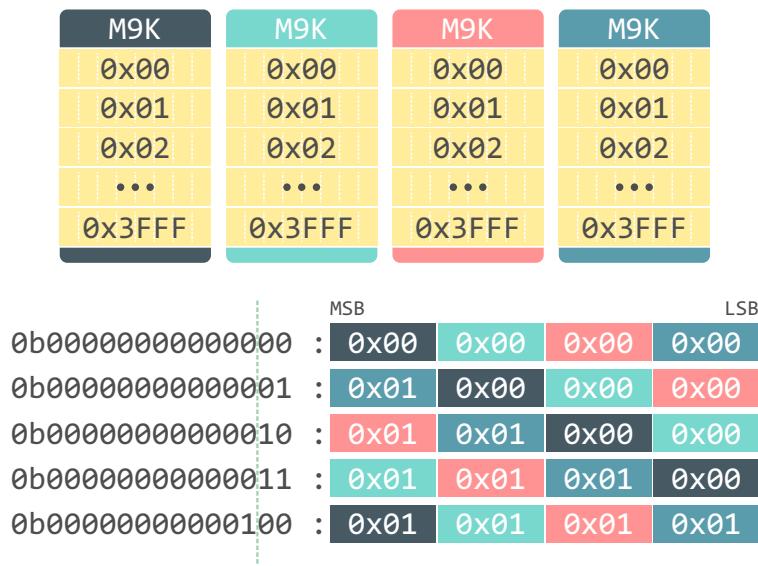


Abbildung 7.1: Speicherpartitionen

Kapitel 8

Tests

8.1 Unit Tests

Als Unitests werden *Modelsim* Simulationen verwendet. Die *Testbench* befindet sich unter *RiscV-i32/CPU/Design-Test/testbench*. Dabei wird jede einzelne Einheit separat von einander getestet. Das Ergebnis ist vordefiniert und wird mit dem Testergebnis verglichen. Als Hilfe dienen Prozeduren aus dem *test_util.vhd* Modul im selben Verzeichnis. Zu diesen Hilfprozeduren zählt *compare_assert* welches das Testergebnis und das zu erwartende Ergebnis vergleicht und eine Rückmeldung über den Test ausgibt. Dabei wird **PASSED** ausgegeben wenn der Test Erfolgreich war und **ASSERT <NACHRICHT> -> expect: [<ERWARTET>], actual: [<ERHALTEN>]** falls ein Fehler auftrat. Listing 8.1 zeigt die Anwendung von *compare_assert*.

Listing 8.1: ALU Testbench

```
1  -- Test: ADD
2  alu_mode <= ALU_ADD;
3  left <= "00000000000000000000000000000001";
4  right <= "00000000000000000000000000000001";
5  compare_assert(result, "00000000000000000000000000000010", "
    ALU_ADD", C_DELAY);
```

8.1.1 Rechenwerk (test_alu.vhd)

Die Rechenwerktests testen die arithmetischen sowie die logischen Operationen.

Erwartetes Ergebnis: Die ALU verrechnet alle Operanden korrekt miteinander.

Tatsächliches Ergebnis: Die Testfälle verhalten sich wie erwartet.

8.1.2 Vergleichseinheit (test_comparator.vhd)

Die Vergleichstests testen die vergleiche von zwei Zahlenwerte wie sie in Kontrollstrukturen verwendet werden.

Erwartetes Ergebnis: Der Comparator vergleicht die Werte korrekt miteinander.

Tatsächliches Ergebnis: Die Testfälle verhalten sich wie erwartet.

8.1.3 Multiplexer (test_mux_*.vhdl)

Die Multiplexertests testen das korrekte Fürcrreichen der Werte an die entsprechenden Einheiten.

Erwartetes Ergebnis: Die Multiplexer reichen, abhängig von den Steuerleitungen, die korrekten Werte weiter.

Tatsächliches Ergebnis: Die Testfälle verhalten sich wie erwartet.

8.1.4 Vergleichseinheit (test_pc.vhd)

Die Befehlszählertests testen das Hochzählen bzw. das Setzen des Befehlszählers abhängig von den Steuerleitungen.

Erwartetes Ergebnis: Der Befehlszähler zählt bzw. setzt den Befehlszähler korrekt.

Tatsächliches Ergebnis: Die Testfälle verhalten sich wie erwartet.

8.1.5 Vergleichseinheit (test_sign_extender_mem.vhd)

Die Signextendertests testen das Erweitern der ausgelesenen Daten aus dem Speicher

Erwartetes Ergebnis: Der Signextender erweitert je nach Steuersignal die ausgelesenen Daten auf Wordgröße.

Tatsächliches Ergebnis: Die Testfälle verhalten sich wie erwartet.

8.2 End-to-End Tests

Als End-to-End Tests wird C-Code verwendet der über die LEDs das Testergebnis ausgibt. Der Quellcode der Tests befindet sich unter *Firmware/src/*.

8.2.1 Kontrollstrukturen (counter.c)

Der Zählertest soll zunächst einfache Kontrollstrukturen testen. Dafür wird die LED als Acht-Bit-Zähler verwendet und kontinuierlich hochgezählt. Wenn das Maximum ($2^8 = 255$) erreicht ist wird der Zähler wieder auf null zurück gesetzt.

Erwartetes Ergebnis: Die LEDs werden Binär hochgezählt bis bis das Maximum erreicht wurde und werden dann zurück gesetzt.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

8.2.2 Shiftoperationen (lightshift.c)

Der Shifttest testet die Shiftoperationen der ALU. Dafür wird die LED auf eins gesetzt und nach links geshiftet. Ist der Maximalwert erreicht, wird nach rechts zurück geshiftet. Bei erreichen des Minimalwertes wird wieder von vorne begonnen.

Erwartetes Ergebnis: Es entsteht ein Lauflicht welches von rechts nach links und zurück läuft.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

8.2.3 Multiplikation und Division (mul_div.c)

Der Multiplikations- und Divisionstest testet beide Rechenarten, die Aufgrund der fehlenden Hardware Unterstützung, durch den Compiler, in Software umgesetzt werden. Hierfür wird eine Abwandlung des Shifttests (Siehe 8.2.2) verwendet der das Shiften durch eine Multiplikation bzw. Division mit zwei ersetzt.

Erwartetes Ergebnis: Es entsteht ein Lauflicht welches von rechts nach links und zurück läuft.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

8.2.4 Unterprogrammsprünge (subroutines.c)

Um Unterprogrammsprünge zu testen wird eine Abwandlung des Kontrollstrukturtests (Siehe 8.2.1) verwendet. Hinzu kommt eine Routine die die Sleep Funktionalität in ein Unterprogramm kapselt. Um zusätzlich Rückgabewerte zu testen wird das Erhöhen des Wertes in einer Funktion abgehandelt.

Erwartetes Ergebnis: Die LEDs werden Binär hochgezählt bis bis das Maximum erreicht wurde und werden dann zurück gesetzt.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

8.2.5 Rekursion (recursion.c)

Der Rekursionstest inkrementiert den Wert in einer Routine die zusätzlich prüft, ob der Wert unter dem maximalwert liegt. Ist dies der Fall, wird die selbe Routine erneut aufgerufen. Dies geschieht solange bis der Wert den Maximalwert erreicht. Die Endlosschleife setzt den Wert anschließend wieder auf null und ruft die Rekursionsroutine erneut auf.

Erwartetes Ergebnis: Die LEDs werden Binär hochgezählt bis bis das Maximum erreicht wurde und werden dann zurück gesetzt.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

8.2.6 Linker (tobig.c)

Der Linkertest testet die Speicherpartitionierung in dem ein zu großes Array an Daten angelegt wird.

Erwartetes Ergebnis: Das Compilieren ist erfolgreich. Das Linken zeigt jedoch einen Fehler an, dass nicht genügend Speicher zur Verfügung steht um das Array zu Speichern.

Tatsächliches Ergebnis: Der Testfall verhält sich wie erwartet.

Kapitel 9

Ausblick

Diese Arbeit zeigt, dass es mit dem *RISC-V* Befehlssatz möglich ist einen einfachen Mikrocontroller zu entwerfen und Programmcode auf diesem auszuführen. Durch die freizügige Lizenz des Befehlssatzes, die damit verbundene Erweiterbarkeit und die freie Wahl eines *FPGAs*, kann der entwickelte Softcore optimal an die Anforderung angepasst werden. Nichtdestotrotz ist der Softcore, der in dieser Arbeit entwickelt wurde, aus Komplexitätsgründen und den damit verbunden Zeitaufwand eher simpel gehalten, sodass ein Anwendungsfall schwer zu bestimmen ist.

Wird der Softcore mit einem Mikrocontroller wie dem *Atmega128* verglichen fällt auf, dass es einen großen Unterschied bei der Peripherie gibt. Hier fehlen gängige Datenbusse wie *SPI* oder *I2C*. Zusätzlich fehlen Analog-Digital-Wandler bzw. Digital-Analog-Wandler, Hardwaredtimer sowie externe Interrupts. All dies macht das Zusammenspiel des Softcores mit Sensoren oder Aktoren schwierig. Ein nachrüsten dieser Funktionalitäten ist jedoch in *VHDL* vergleichsweise einfach umzusetzen und könnte im Rahmen einer zukünftigen Arbeit geschehen.

Von der Peripherie abgesehen gibt es auch Potential bei der Implementierung der Befehlsätze für Multiplikation und fließkommazahl Arithmetik. Dies würde den Mehraufwand durch Softwareimplementierungen verringern und somit die Performanz des Softcores steigern. Eine Anbindung von externem Flash für den Programmspeicher und RAM würde für manche Aufgaben nützlich sein, da somit mehr Speicher zu Verfügung stehen würde. Die daraus zu Verfügung stehenden internen Speicherblöcke könnten daraufhin auch als Cache fungieren und die langsamen Speicherzugriffe beschleunigen. Sicherlich wäre damit auch ein Wechsel auf eine Pipelinearchitektur verbunden.

Abbildungsverzeichnis

2.1	Öffnen des Projektes in Quartus	3
2.2	Öffnen des Projektes in Quartus	4
2.3	Bauen des Projektes in Quartus	4
2.4	Erfolgreiches Bauen des Projektes in Quartus	4
2.5	Öffnen des Programmers in Quartus	5
2.6	Einrichten der Hardware in Quartus	6
2.7	Einrichten der Hardware in Quartus	6
2.8	Starten des Flashens des FPGAs	6
2.9	Parameterübersicht von Build	7
2.10	Beispielquellcode compiliert durch build	8
2.11	MIF Aktualisieren	10
2.12	Assemblierung	11
2.13	Platine des TEI0003 TRM FPGA-Board	11
5.1	Schaubild des TEI0003 TRM FPGA-Board	16
5.2	Platine des TEI0003 TRM FPGA-Board	16
5.3	Von-Neumann-Architektur	19
5.4	Befehlsverarbeitung mit und ohne Pipelining	22
5.5	Speicherpyramide	23
6.1	Gesamtübersicht des Softcores	24
6.2	Dekodiereinheit	25
6.3	Befehlszählers	25
6.4	Vergleichswerk	26
6.5	Registereinheit	27
6.6	Register Multiplexer	27
6.7	ALU	28
6.8	Linker ALU Multiplexer	29
6.9	Rechter ALU Multiplexer	29
6.10	Taktflanken am Speicher	31
6.11	Vorzeichenerweiterung	32
6.12	Speicher mit IO Bereich	33
7.1	Speicherpartitionen	38

Listings

2.1	Plattform.h	8
2.2	Lighthshift.c	9
6.1	Prozess des Befehlszählers	26
6.2	Prozess der Registereinheit	27
6.3	Prozess des register Multiplexers	28
6.4	Prozess des linken Rechenwerk Multiplexers	29
6.5	Prozess des rechten Rechenwerk Multiplexers	30
6.6	Beispiel MIF	32
6.7	Adressaufteilung im Speicher	33
6.8	Leseprozess für die Peripherie	34
6.9	Schreibeprozess für die Peripherie	34
7.1	Linkerskript	36
7.2	Bootloader crt0.s	37
8.1	ALU Testbench	39

Literatur

- [Car02] Carlos Carvalho. *The Gap between Processor and Memory Speeds*. http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf. 2002.
- [Ele] Trenz Electronic. *TEI0003 TRM*. <https://wiki.trenz-electronic.de/display/PD/TEI0003+TRM>. [Zugegriffen 03.2021].
- [Inta] Intel. *Intel MIF Doc*. https://www.intel.com/content/www/us/en/programmable/quartushelp/current/index.htm#reference/glossary/def_mif.htm. [Zugegriffen 03.2021].
- [Intb] Intel. *Intel® Cyclone® 10 LP Core Fabric and General Purpose I/Os Handbook*. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-10/c10lp-51003.pdf>. Version 2020.12.03.
- [Intc] Intel. *Intel® Cyclone® 10 LP Device Datasheet*. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-10/c10lp-51002.pdf>. Version 2020.05.21.
- [Lar] Scott Larson. *UART Vhdl*. <https://www.digikey.com/eewiki/pages/viewpage.action?pageId=59507062>. [Zugegriffen 03.2021].
- [McK95] Win. A. Wulf & Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.5726&rep=rep1&type=pdf>. März 1995.
- [Ris] Riscv.org. *Risc-V Toolchain*. <https://github.com/riscv/riscv-gnu-toolchain>. [Zugegriffen 03.2021].
- [ris] riscv.org. *RISC-V Spec*. <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>. [Zugegriffen 03.2021].
- [wika] wikipedia.org. *FPGA*. https://de.wikipedia.org/wiki/Field_Programmable_Gate_Array. [Zugegriffen 03.2021].
- [wikb] wikipedia.org. *Memory hierarchy*. <https://de.wikipedia.org/wiki/Speicherhierarchie>. [Zugegriffen 03.2021].
- [wikc] wikipedia.org. *Pipelining*. [https://de.wikipedia.org/wiki/Pipeline_\(Prozessor\)](https://de.wikipedia.org/wiki/Pipeline_(Prozessor)). [Zugegriffen 03.2021].
- [wikd] wikipedia.org. *Von-Neumann-Architektur*. <https://de.wikipedia.org/wiki/Von-Neumann-Architektur>. [Zugegriffen 03.2021].