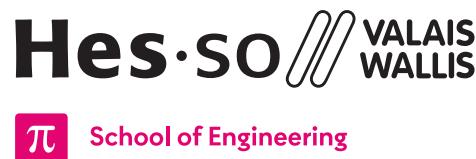




# HEIRV32 (HEIRV32)

Vorlesung Computerarchitektur (CAr)



**Orientierung:** Informatik und Kommunikationswissenschaften (ISC)

**Spezialisierung:** Data Engineering (DE)

**Kurs:** Computerarchitektur (CAr)

**Authoren:** Silvan Zahno, Axel Amand

**Datum:** 31.01.2025

**Version:** v2.2



# Inhalt

1 Einführung .....	3
1.1 Multi-cycle HEIRV32 Mikroprozessor .....	4
1.1.1 Reference Documents .....	4
2 Spezifikation .....	5
2.1 Funktionen .....	5
2.2 Support für zusätzliche Befehlen .....	6
2.2.1 Easy .....	6
2.2.2 Medium .....	6
2.2.3 Hard .....	6
2.3 HDL-Designer Projekt .....	7
3 Komponenten .....	9
3.1 EBS3 <b>Field Programmable Gate Array (FPGA)</b> Platine .....	9
3.2 Knöpfe und <b>Light Emitting Diode (LED)</b> .....	9
3.3 Zusätzliche LED .....	10
3.4 Optionale Karten .....	10
4 Bewertung .....	11
5 Leitfaden .....	12
5.1 Allgemeine Architektur .....	12
5.1.1 Instruktion lw .....	12
5.1.2 Signal en .....	14
5.1.3 Instruktion sw .....	14
5.1.4 Instruktion R- und I-Type .....	15
5.1.5 Instruktion beq .....	15
5.1.6 Instruktion jal .....	15
5.2 Control Unit .....	16
5.2.1 ALU Block .....	17
5.2.2 Extend Block .....	17
5.2.3 ALUOp Signal .....	17
5.3 Simulation .....	18
5.3.1 Verständnis .....	18
5.3.2 Automatisierung von Tests .....	18
5.4 Code .....	19
5.4.1 Prüfung durch Äquivalenz .....	19
5.4.2 Benutzerdefinierter Code .....	19
5.5 Tips .....	20
Glossar .....	21

# 1 | Einführung

Ziel des Projekts ist es, das erworbene Wissen am Ende des Semesters direkt mit Hilfe eines praktisches Beispiele anzuwenden. Es geht darum, einen reduzierten **5-stages Reduced Instruction Set Computer architecture, open-sourced (RISCV)** Prozessor zu erstellen, um einen kleinen Assemblerprogramm auszuführen. Der Prozessor kann simuliert und auf einer **FPGA** -Hardware implementiert werden. Die Verbindung mit der Außenwelt kann über Knöpfe und Leds erfolgen. Dieses Prozessorsystem ist in der Abbildung 1 dargestellt.

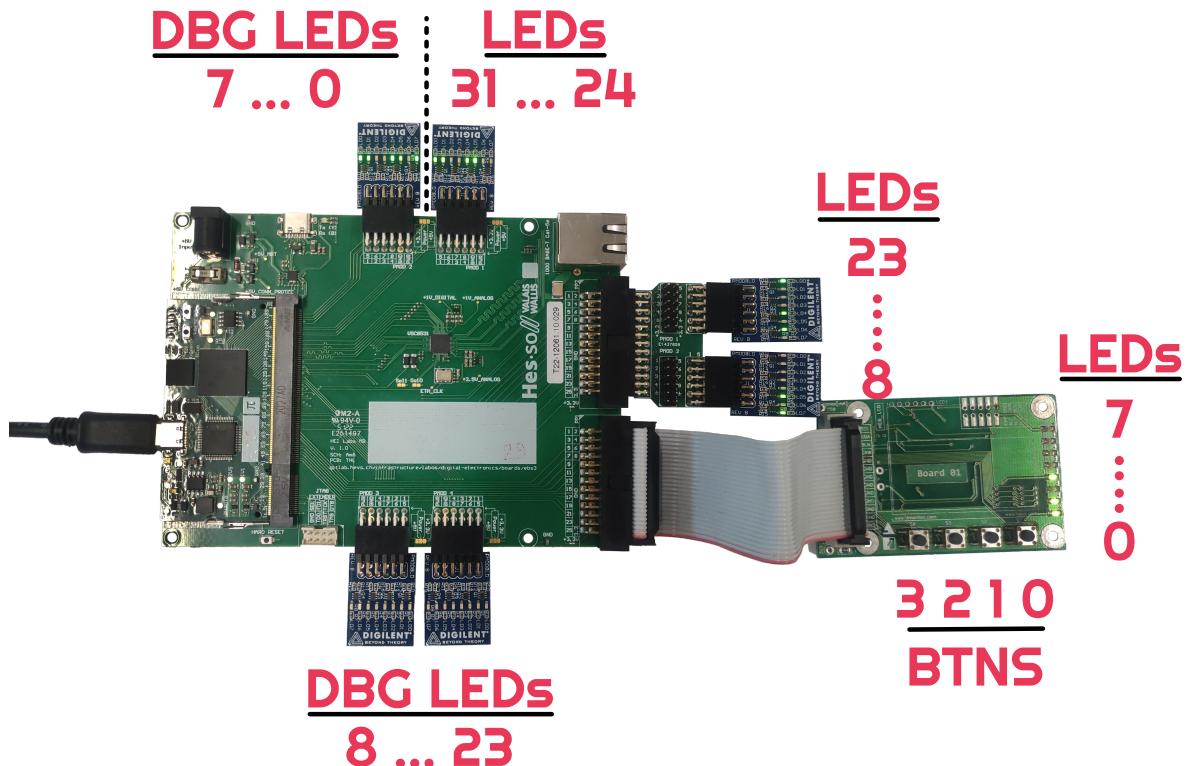


Abbildung 1 - Hardwareaufbau des Systeme (EBS3)

Die Aufgabe besteht aus einer klar definierten minimalen Spezifikation Abschnitt 2.

## 1.1 Multi-cycle HEIRV32 Mikroprozessor

Die Ein-Zyklus-Architektur, die in einem früheren Labor vervollständigt wurde, ermöglicht eine einfache Annäherung an die RISC-V Architektur. Es bringt jedoch mehrere Probleme mit sich:

- Programm- und Datenspeicher sind getrennt
  - die Implementierung erfordert zwei Chips für ein einzelnes Programm
- auf den Speicher wird asynchron zugegriffen
  - die auf dem Markt verkauften Speicher unterstützen nur den synchronen Zugriff auf Daten
  - der Platz, den die Decodierlogik einnimmt, wird beträchtlich : HEIRV32 single-cycle unterstützte nur  $2^5$  Befehle und füllte somit den **FPGA** aus
  - die Kapazitäten dieser Speicher würden weit unter den heute verfügbaren liegen
- die Geschwindigkeit des Prozessors wird durch den längsten Befehl begrenzt
  - Verbesserungen sind nur noch durch die Weiterentwicklung der Transistortechnologie und die Verkürzung der Verkabelungs-Längen möglich

Grundsätzlich bestehen die Anweisungen aus fünf Schritten:

- Fetch: die Anweisung wird aus dem Speicher abgerufen
- Decode: die Anweisung wird dekodiert : funct3, funct7, ALU- und Quelleneinstellungen dots
- Execute: die angegebene Operation wird ausgeführt
- Memory: greift auf bestimmte Daten im Speicher zu (optional)
- Writeback: speichert Daten im Speicher (optional)

Da nicht alle Anweisungen die Ausführung jedes einzelnen Schritts erfordern, können diese Schritte durch einen Taktzyklus getrennt werden  $\Rightarrow$  Multi-Cycle. Auf diese Weise kann die Taktfrequenz erhöht werden, die Geschwindigkeit wird nun durch den langsamsten Schritt begrenzt.

Kürzere Befehle die nur 3-4 Schritte erfordern profitieren von einer schnelleren Verarbeitung:

Die verwendeten Speicher sind nun synchron.



Diese Architektur ebnet auch den Weg für die Implementierung eines „[Pipeline-Systems](#)“ (d. h. bei jedem Taktzyklus wird ein Befehl geladen), für „[die Vorhersage von Verzweigungen](#)“ und für jede andere Technik, die den allgemeinen Betrieb des Prozessors beschleunigt.

Nur Multi-Cycle wird hier behandelt.

### 1.1.1 Reference Documents

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29]



## 2 | Spezifikation

### 2.1 Funktionen

Die Basisfunktionen sind wie folgt definiert:

- Die **FPGA** beinhaltet einen 32-Bit, multi-cycle **RISC-V** Mikroprozessor deren Funktionsweise durch Simulation und anschliessenden Einsatz auf einem physischen Chip bestätigt wird.
- Der implementierte Microprozessor muss folgende Befehle unterstützen:
  - R-Type Befehle: **add, sub, and, or, slt**
  - I-Type Befehle: **addi, andi, ori, slli**
  - Speicher Befehle: **lw, sw**
  - Sprung Befehle: **beq, jal**
- Mithilfe der unterstützten Befehlen schreibe Sie einen Assemblercode, der in der Lage ist:
  - erkennt, wenn die Tasten auf der Karte Knöpfe-LEDs-LCD Platine Abbildung 5 gedrückt werden und handelt entsprechend.
  - die **LED** auf der Karte Knöpfe-LEDs-LCD Platine Abbildung 5 kontrollieren

Um die **LED** zu steuern, schreiben Sie einfach das Register **x30**.

Um die Tasten zu lesen, lesen Sie einfach das Register **x31**. Ein Schreibvorgang in dieses Register ändert seinen Wert nicht.

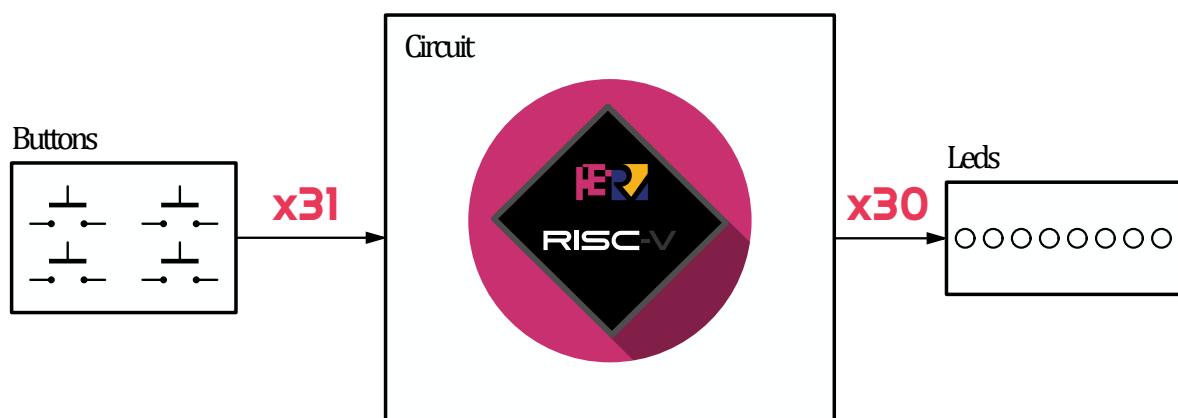


Abbildung 2 - **RISC-V** Hardware Schaltung



Die obligatorische Funktionalität besteht darin, dass die LEDs einfach aufleuchten, wenn eine Taste gedrückt wird. Siehe Abschnitt 2.2 für optionale Funktionen, die zusätzliche Punkte bringen.

## 2.2 Support für zusätzliche Befehlen

Die Algorithmen bringen je nach Schwierigkeitsgrad zusätzliche Extra-Punkte ein.



Bevor Sie starten, überprüfen mit Ihren Betreuern dass die benötigten Materialien vorhanden und mit Ihrer Idee kompatibel sind.

Achten Sie darauf, dass Sie nur die implementierten Anweisungen verwenden.

*Die Verwendung eines RISC-V-Compilers wird Ihnen **NICHT** kompatiblen Code liefern (nicht unterstützte Anweisungen, Abschnittsdirektiven ...).*

### 2.2.1 Easy

- **Verwaltung von Schaltflächen:** auf den 32 LED je nach Tastendruck unterschiedliche Patterns anzuzeigen.
- **Verwaltung von Timings:** die LED in einer bestimmten Frequenz blinken lassen.

### 2.2.2 Medium

- **LED Lauflichter:** Die LED leuchten nacheinander in mässiger Frequenz auf; ein Tastendruck stoppt die Fahrt des Lauflichts, solange er gedrückt wird, während ein weiterer Knopfdruck das Lauflicht von 0 wieder anlaufen lässt.
- **LED mit variabler Intensität:** die LED leuchten mit einem auf 50% eingestellten **Pulse Width Modulation (PWM)**. Ein Tastendruck erhöht die Intensität bei jedem Drücken um 10%, ein weiterer senkt sie um 10%.

### 2.2.3 Hard

- **Universal Asynchronous Receiver Transmitter (UART):** beim Drücken einer Taste wird eine Textübertragung durch **UART** durchgeführt. Der Text wird auf einem PC angezeigt.
- **Serielle LED RGB:** Verwaltung von drei seriellen RGB LED vom Typ WS2812/WS2813. Ein Tastendruck schaltet die LED ein/aus; zwei weitere schalten zur nächsten/vorherigen Farbe um.
- **Atmende LED:** die LED leuchten auf und erlöschen dann allmählich, wodurch ein „Atmungseffekt“ entsteht. Mit einem Knopf kann die Atemgeschwindigkeit erhöht, mit einem anderen verringert werden.

Natürlich ist es auch möglich, eigene Ideen einzubringen und Zusatzmaterial zu verwenden (siehe Abschnitt 3).



Es ist sehr ratsam, Ihren Algorithmus mit den Werkzeugen zu entwickeln und zu testen, die Sie in den ISA-Laboren gesehen haben (**RISC-V online interpreter** und **Ripes**)

Es ist möglich, die Register auf Ripes manuell zu bearbeiten, um die Knöpfe zu simulieren.

## 2.3 HDL-Designer Projekt

Ein vordefiniertes HDL-Designer Projekt kann im [Cyberlearn](#) heruntergeladen oder geklont von [Git](#) werden. Die Dateistruktur des Projektes sieht folgendermassen aus:

```

car_heirv
+--Board/                      # Project and files for programming the fpga
|   +-concat/                  # Complete VHDL file including PIN-UCF file
|   +-hds/                     # Board-related VHDL files
|   +-ise/                     # Xilinx ISE project
|   +-diamond/                 # Lattice Diamond project
+--HEIRV32/                     # Library for the components of the student solution
+--HEIRV32\_test/                # Library for the simulation testbenches
+--Libs/                        # External libraries which can be used e.g. gates, io, sequential
+--Prefs/                       # HDL-Designer settings
+--Scripts/                     # HDL-Designer scripts
+--Simulation/                  # Modelsim simulation files
+--doc/                          # Folder with additional documents relevant to the project
|   +-Board/                   # All schematics of the hardware boards
|   +-Components/              # All data sheets of hardware components
|   +-HEIRV32\_MC\-\-x          # This doc
+--heirv32\-asm/                 # Dedicated assembler for HEIRV32 (doc and execs)
+--img/                         # Pictures

```



Der Pfad des Projektordners darf keine Leerzeichen enthalten.



Im Projektordner **doc/** können viele wichtige Informationen gefunden werden. Datenblätter, Projektbewertung sowie Hilfsdokumente für HDL-Designer um nur einige zu nennen.

Die Signale des Top-Levels welche zu implementieren sind, sehen wie folgt aus:

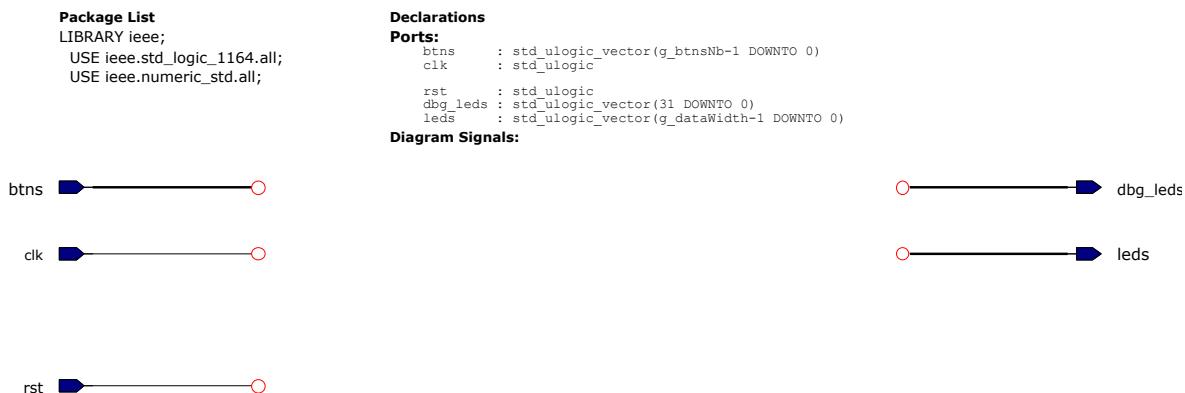


Abbildung 3 - Leere Toplevel Schaltung

Die verfügbaren Signale sind wie folgt:

- **btns**: Werte der Tasten S0 bis S4 auf der Knöpfe-LEDs-LCD Platine Abbildung 5. Dies werden mit dem Register **x31** verlinkt.
- **leds**: LED die mit dem Register **x30** verlinkt sind.
- **clk**: Systemclock, getaktet mit **50MHz** für die EBS3 boards und **66 MHz** für die EBS2 boards.
- **rst**: System-Reset, asynchron.
- **dbg\_leds**: Debug-Signal, ermöglicht die Aktivierung von **LED** nach internen Signalen. Sind nicht an das Register x30 gebunden.

Alle Blöcke, die für den Entwurf des Mikroprozessors benötigt werden, werden im Block **heirv32\_mc** bereitgestellt und stammen aus den folgenden Bibliotheken:

- HEIRV32\_MC
  - **controlUnit**: Block für die Dekodierung von Anweisungen.
  - **heirv32\_mc**: Top-level
  - **instructionDataManager**: Programmspeicher, der Anweisungen und Daten vereint und gelesen und geschrieben werden kann.
- HEIRV32
  - **ALU**: eine Version des ALU, welcher die Funktionen: Addition, Subtraktion, AND, OR und SLT implementiert.
  - **buffer\*(Enable)**: getakteter Buffer (Flip-Flops) mit oder ohne Enable-Eingang.
  - **extend**: Anweisungserweiterungsblock für Tests, der die Anweisungen I, S, B und J unterstützt.
  - **mux3To1ULogVec**: mux 3-to-1 of **std\_ulogic\_vector**.
  - **registerFile**: Block zur Verwaltung der 32 Register, wobei x31 durch den Vektor **btns** - *Register zum Lesen der Tasten* - und **x30** durch den Vektor **leds** - *Register zum Schreiben der LED* - ersetzt wird.

Die **Board**-Bibliothek enthält die Signalformungslogik, die für den Einsatz der Schaltung auf einer **FPGA** vorgesehen ist.

Die Bibliothek **HEIRV32\_test** enthält einen Tester, der mit dem Code **Simulation/code\_sim.s** verknüpft ist.



Der Simulator ist standardmäßig auf EBS3 eingestellt. Um dies zu ändern, öffnen Sie den Block **heirv32\_mc\_tb** und ändern Sie die Konstante  
`constant c_clockFrequency : real := 50.0E6;`  
 in  
`constant c_clockFrequency : real := 66.0E6;`

# 3 | Komponenten

Das System besteht aus drei verschiedenen Hardwareplatten, die in der Abbildung Abbildung 1 zu sehen sind.

- eine **FPGA** Entwicklungskarte, siehe Abbildung Abbildung 4.
- Eine Steuerplatine mit 4 Tasten und 8 **LED**, siehe Abbildung Abbildung 5.
- Zwei **Peripheral Module (PMod)** LED Karten, siehe Abbildung Abbildung 6.

## 3.1 EBS3 FPGA Platine

Die Hauptplatine ist die **FPGA** EBS 3 Laborentwicklungsplatine der Schule. Diese beherbergt eine **Lattice LFE5U-25F FPGA** und verfügt über viele verschiedene Schnittstellen (**UART**, **PMod**, PPT, Ethernet). Der benutzte Oszillator erstellt ein Taktsignal (**clock**) mit einer Frequenz von  $f_{\text{clk}} = 100\text{MHz}$ , intern durch PLL auf  $f_{\text{clk}} = 50\text{MHz}$  reduziert.

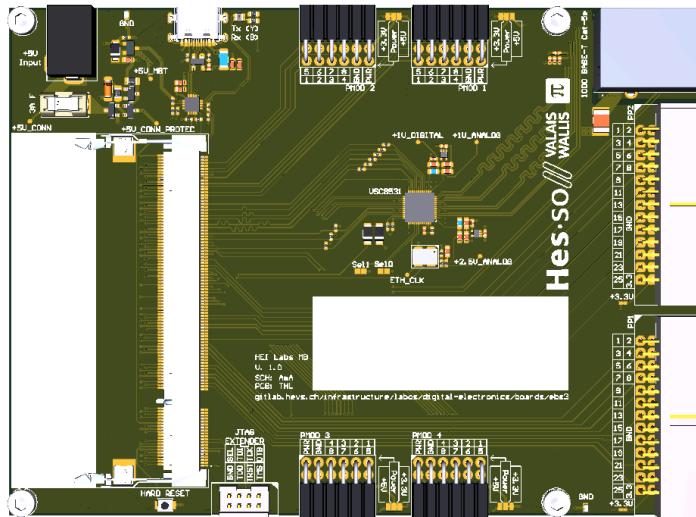


Abbildung 4 - **FPGA** Platine

## 3.2 Knöpfe und LED

Die Platine mit den Knöpfen und **LED** [8] wird an die motherboard angeschlossen. Sie hat 4 Tasten und 8 **LED**, die im Design verwendet werden können. Falls gewünscht kann diese Platine mit einer **Liquid Crystal Display (LCD)** Anzeige ausgestattet werden [9], [30].



Abbildung 5 - Knöpfe **LED LCD** Platine [8]

### 3.3 Zusätzliche LED

Es ist möglich, die Anzahl der **LED** mithilfe von dedizierten Erweiterungsboards zu erweitern.  
Es ist auch möglich, weitere Erweiterungsplatten anzuschliessen (Abstandssensor, Open-Drain-Ausgänge ...).



Abbildung 6 - **LED PMod**

### 3.4 Optionale Karten

Um die Funktionalität Ihrer Schaltung zu erweitern, können Sie Erweiterungsplatten verwenden.  
Ihre Dokumentation finden Sie unter dem Ordner **doc/ext\_boards**.

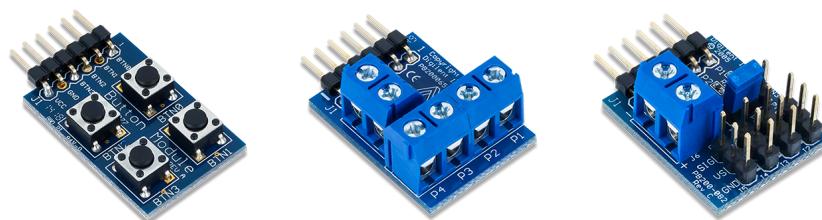


Tabelle 1 - Inputs: **PMOD** BTN [19], [20], **PMOD** CON1 [21], [22], **PMOD** CON3 [23], [24]

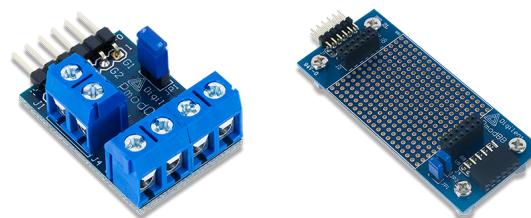


Tabelle 2 - Outputs: **PMOD** OD1 [26], [27], **PMOD** BB [17], [18]

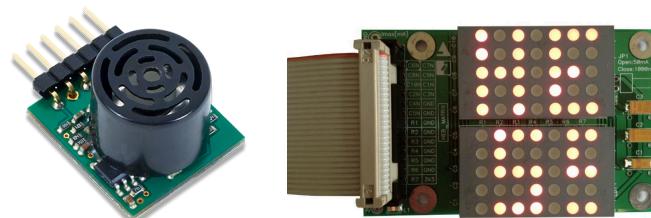


Tabelle 3 - I/Os: **PMOD** MAXSONAR [25], [29], PP-MATRIX [28]

## 4 | Bewertung

Im Ordner **doc/** zeigt die Datei **evaluation-bewertung-riscv.pdf** das detaillierte Bewertungsschema, Tabelle Tabelle 4.

Die Schlussnote beinhaltet den Bericht, den Code sowie eine Präsentation eurerseits des Systems.

Evaluierte Aspekte	Punkte
<b>Bericht</b>	<b>50</b>
Einleitung	3
Spezifikation	5
Entwurf	15
Verifizierung und Validation	10
Integration	9
Schlussfolgerung	3
Formale Aspekte des Berichtes	5
<b>Funktionalität der Schaltung</b>	<b>30</b>
<b>Qualität der Lösung</b>	<b>10</b>
<b>Präsentation</b>	<b>10</b>
<b>Total</b>	<b>100</b>

Tabelle 4 - Bewertungsraster



Das Bewertungsraster gibt bereits Hinweise über die Struktur des Berichtes. Für einen guten Bericht konsultieren Sie das Dokument „Wie verfasst man einen Projektbericht?“ [12].

# 5 | Leitfaden

Um mit dem Projekt zu beginnen, kann folgendermassen vorgehen werden:

- Lest die obigen Spezifikationen und Informationen genau durch.
- Schaut euch die Hardware mit dem vorinstallierten Programm und das gegeben HDL-Designer Projekt.
- Stöbert durch die Dokumente im Ordner **doc/** eures Projektes.
- Analysiert im Detail die Blöcke welche bereits vorhanden bzw. vorgegeben sind.
- Entwickelt ein detailliertes Blockdiagramm. Die Signale und deren Funktionen solltet Ihr erklären können.
- Implementierung und Simulation der verschiedenen Blöcken.
- Testen der Lösung in der Simulation und finden etwaiger Fehler **🐞**.
- Schreiben und verteilen Sie Ihren eigenen Code

## 5.1 Allgemeine Architektur

Schlagen Sie zunächst eine Architektur vor, die keinen Block implementiert und sich um das Multi-Cycle-Prinzip dreht:



Abbildung 7 - RISC-V Pipeline

### 5.1.1 Instruktion Iw

Das Design dreht sich um den Programmspeicher, indem es über die Anweisung **Iw** nachdenkt, eine Anweisung, die die fünf Schritte der Pipeline erfordert:

- Fetch: der Programm-Counter ordnet Informationen aus dem Speicher und erzeugt so einen **data(RD)**- und **instruction (instr)**-Bus. Die Busse sind sequentiell: RD wird bei jedem Clockschlag aus dem Speicher gelesen, während Instr. nur dann aktualisiert wird, wenn der IRWrite-Eingang auf „1“ gesetzt ist.

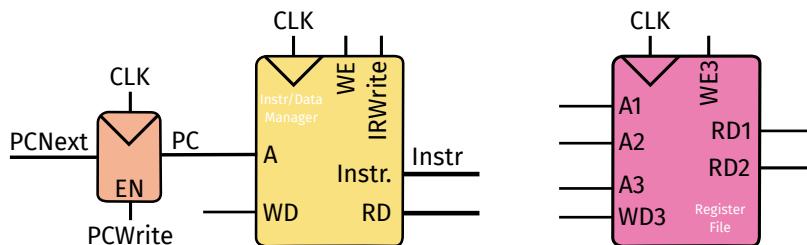


Abbildung 8 - Fetch

- Decode:
  - die Basisadresse ist im Instr-Bus enthalten, Bits 19 downto 15. Sie werden als Adresse verwendet, um das Register **RD1** auszuwählen.
  - der unmittelbare Wert ist in den Bits 31 downto 20 gegeben, dessen Vorzeichen erweitert werden muss. Dazu kann man mit dem Block **extend** über zwei Kontrollbits festlegen, ob der Wert mit 12, 13 oder 21 Bits codiert wird und somit das Vorzeichen des Wertes erweitern.
  - die Register (hier RD1) werden bei jedem Clockschlag aktualisiert.

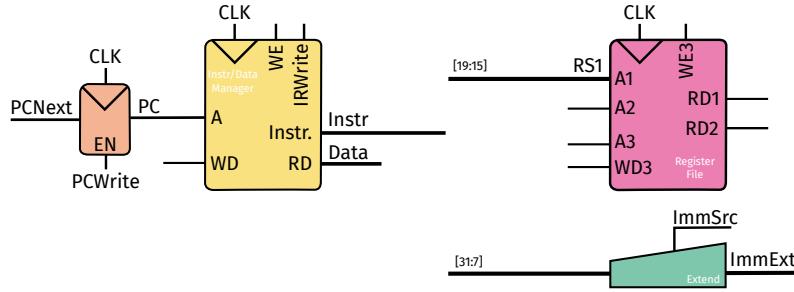


Abbildung 9 - Decode

- Execute: der unmittelbare Wert wird dem über die ALU ausgelesenen Register hinzugefügt, um die Adresse zu bestimmen, auf die im Speicher zugegriffen werden soll. Mit dem Output-Flip-Flop wird dieser Schritt noch einmal vom Rest der Pipeline getrennt.

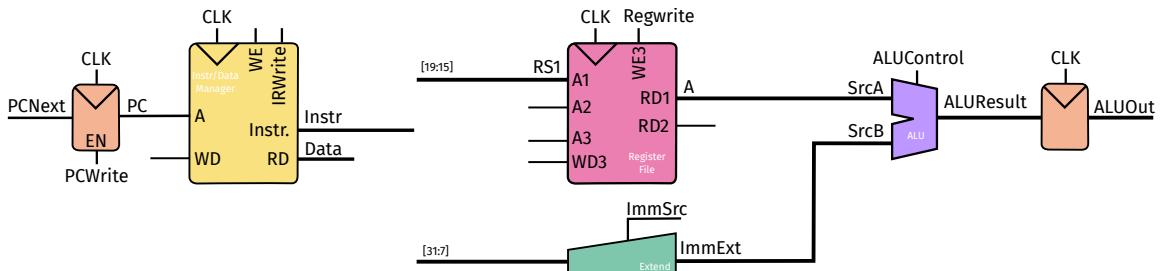


Abbildung 10 - Execute

- Mem. Access: der berechnete Wert wird verwendet, um den Speicher neu zu lesen. Zu diesem Zweck wird ein Multiplexer hinzugefügt, der zwischen dem PC oder dem ALU-Wert wählt, gesteuert durch das Signal AdrSrc.

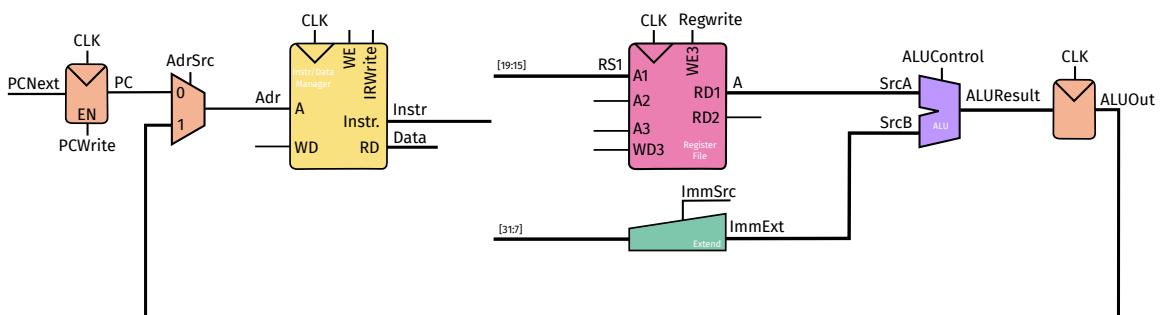


Abbildung 11 - Memory Access

- Write Back: der letzte Schritt ist um das Zielregister zu schreiben, das durch die Bits 11 downto 7 des Instr-Busses angegeben wird. Das Signal **RegWrite** lädt beim nächsten Clockschlag das von A3 angezeigte Register. Dieser Wert kann wie hier aus dem Ergebnis der ALU oder aus den Daten selbst stammen. Ein Multiplexer, ist hinzugefügt, der durch das Signal **ResultSrc** gesteuert wird:

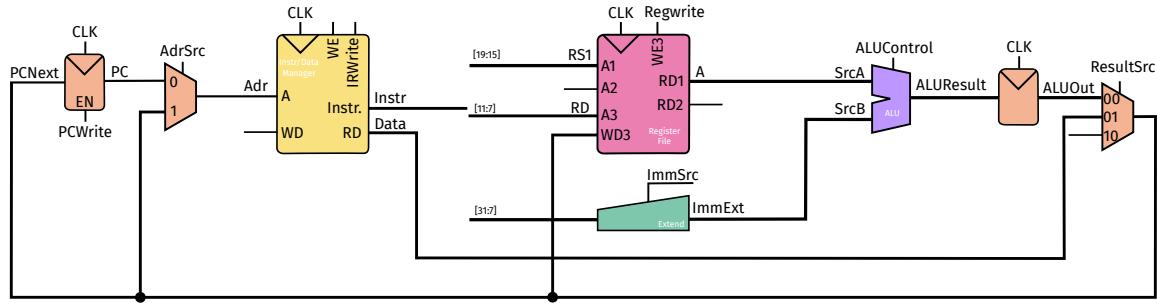


Abbildung 12 - Write Back

Parallel zu all dem muss der Programmzähler inkrementiert werden. Dies wurde in der Single-Cycle-Architektur mit einem separaten Addierer gemacht. Hier ist es jedoch möglich, die ALU während des Fetch-Schritts zu verwenden, da keine Berechnung erforderlich ist. Zur Steuerung der ALU-Quellen werden zwei Multiplexer hinzugefügt, die von den Signalen **AdrSrcA** und **AdrSrcB** gesteuert werden. Hier wird der PC auf A geladen, während B den Wert 4 lädt, wobei der ALU auf Addition eingestellt ist. Da die Berechnung sofort verwendet werden soll (PC muss zur späteren Verwendung gespeichert werden), wird das Flip-Flop des ALU-Ergebnisses umgangen und das Signal zum Ausgangsmultiplexer addiert. Der von PCWrite gesteuerte Enable-Eingang ist auf '1' gesetzt, wodurch PCNext (= PC + 4) auf dem PC gespeichert werden kann:

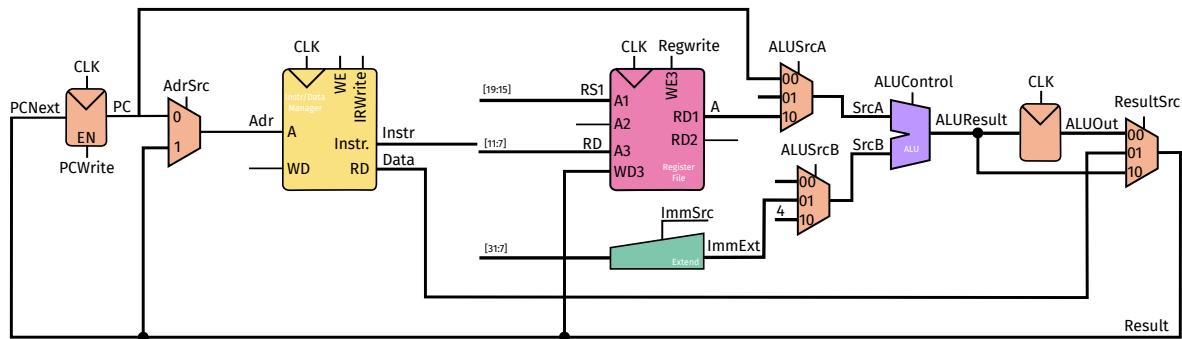


Abbildung 13 - PC + 4

*Steuersignale werden von dem weiter unten angesprochenen Block **controlUnit** erzeugt.*

### 5.1.2 Signal en

Das Signal **en** ermöglicht es, die Arbeit des Prozessors zu unterbrechen. ALLE Schritte der Pipeline müssen blockiert werden, wenn dieser Eingang auf '0' gesetzt ist. Ändern Sie Ihre Architektur, um dieses Signal dort zu berücksichtigen, wo es benötigt wird.

### 5.1.3 Instruktion sw

In der gleichen Argumentation erweitern Sie das System, um **sw** zu unterstützen:

- Fetch: liest die Anweisung aus dem Speicher, auf den PC zeigt.
- Decode: lädt das Register, das die Basisadresse angibt, auf RD1. Lädt ausserdem das Register, das die zu sichernde Information enthält, auf RD2.
- Execute: fügt den unmittelbaren Wert über die ALU zur Basisadresse hinzu, um auf die Speicheradresse zu verweisen.

- Write Back: speichert den Wert mithilfe des Signals **MemWrite** im Speicher.

#### 5.1.4 Instruktion R- und I-Type

Überlegen Sie dann, welche Pfade für Anweisungen vom Typen R und I benötigt werden.

#### 5.1.5 Instruktion beq

Fügen Sie die Anweisung **beq** hinzu, die zwei Register vergleicht und den PC ändert, wenn diese gleich sind:

- Fetch: liest die Anweisung aus dem Speicher, auf den PC zeigt.
- Decode: lädt die beiden zu vergleichenden Register. Da die ALU nicht verwendet wird, kann die Adresse bei einem Sprung berechnet werden. Allerdings hat sich der PC schon inkrementiert. Daher muss im vorherigen Schritt ein alter PC-Wert gespeichert werden, um ihn in die Quelle A des ALUs zu laden. Quelle B ist der Sofortwert.
- Execute: subtrahiert die beiden Register und setzt das Signal **zero** auf '1', wenn das Ergebnis 0 ist. In diesem Fall setzt der Kontrollblock das Signal **PCWrite** auf '1', ein Signal mit dem der Result-Bus auf den PC geladen wird. Damit wird der Sprungwert (ALU-Ausgang aus dem vorherigen Schritt) geladen ( $a == b$ ,  $PC = PC + \text{imm.}$ ). Andernfalls bleibt PCWrite auf '0' und PC wird nicht geändert ( $a \neq b$ ,  $PC = PC + 4$ ).

#### 5.1.6 Instruktion jal

Fügen Sie schliesslich die Anweisung **jal** hinzu, die nach dem Speichern der Rücksprungadresse springt:

- Fetch : liest die Anweisung aus dem Speicher, auf den PC zeigt.
- Decode : berechnet die Sprungadresse.
- Execute : speichert die Sprungadresse als neuen PC, während sie die im Zielregister zu speichern-de Rücksprungadresse berechnet.
- Write Back : speichert die Rücksendeadresse im Zielregister.

## 5.2 Control Unit

Es fehlt noch ein Kontrollblock, der den aktuellen Schritt gemäss der oben genannten Pipeline verfolgt und die verschiedenen Kontrollsingale erzeugt. Um die Arbeit zu vereinfachen, trennen Sie die Signalerzeugung in drei verschiedene Blöcke:

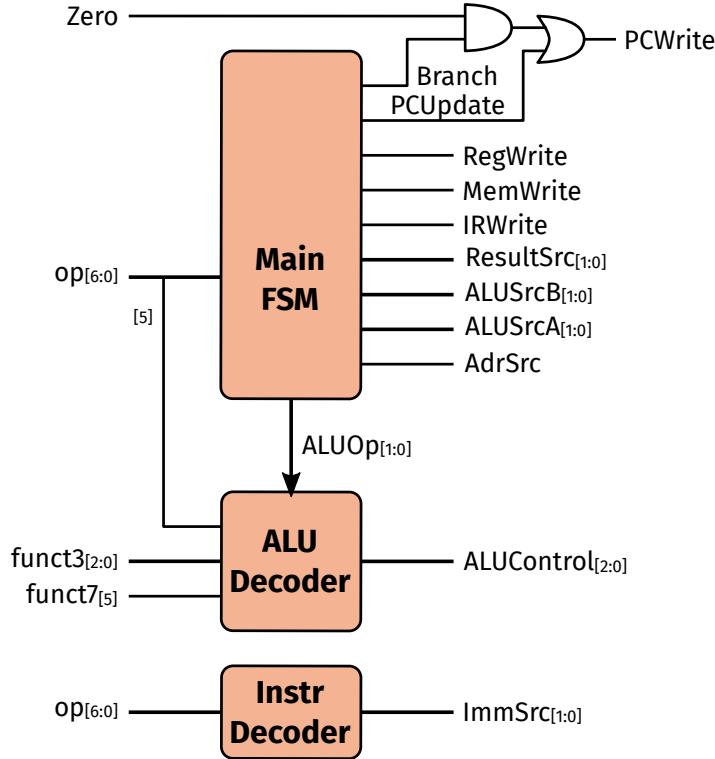


Abbildung 14 - Control unit



Die Anweisungen des RV32I-Sets sind in 6 Typen unterteilt: R, I, S, B, U und J. Es ist logisch, dass zwei Anweisungen desselben Typs die gleiche Verarbeitung erfahren. Wenn dies nicht der Fall wäre, müsste jede Anweisung vom control-Unit-Block unterschiedlich behandelt werden. Allein für das Basic-Set gibt es 40 davon, Set mit dem man kein Betriebssystem betreiben kann!

### 5.2.1 ALU Block

Die ALU realisiert die arithmetischen und logischen Funktionen gemäss der folgenden Tabelle:

ALU Control	Operation
<b>000</b>	<i>add</i>
<b>001</b>	<i>sub</i>
<b>010</b>	<i>AND</i>
<b>011</b>	<i>OR</i>
<b>101</b>	<i>Set Lesser Than</i>
Others	-

Tabelle 5 - Operationstabelle für den ALU-Block

### 5.2.2 Extend Block

Der Block extend stützt sich auf den von **immSrc** vorgegebenen Anweisungstyp, um das Vorzeichen des unmittelbaren Wertes zu extrahieren und zu erweitern:

immSrc	Type
<b>00</b>	<i>I</i>
<b>01</b>	<i>S</i>
<b>10</b>	<i>B</i>
<b>11</b>	<i>J</i>

Tabelle 6 - Operationstabelle für den Extend-Block

### 5.2.3 ALUOp Signal

Um Doppelungen zu vermeiden, wird **op<sub>[6:0]</sub>** nicht in den Block **aluDecoder** übernommen. Stattdessen erzeugt **mainDecoder** das kleinere Signal **ALUOp**, das mehrere Arten von Anweisungen zusammenfasst, die nach einem willkürlichen Code auf die gleiche Weise arbeiten. Beispielsweise führen **addi x2, x3, 30** und **add x2, x3, x4** beide eine Additionsoperation aus. Die Anweisungen I und R sind daher unter demselben Code zusammengefasst.

Die folgende Tabelle zeigt diese Idee der Vereinheitlichung und listet die speziellen Kästchen für die Dekodierung auf:

ALUOp	funct3	Op <sub>5</sub> /funct7 <sub>5</sub>	instr	ALUControl <sub>[2:0]</sub>
<b>00</b>	---	--	<i>lw, sw</i>	<b>000</b> (add)
<b>01</b>	---	--	<i>beq</i>	<b>001</b> (sub)
<b>10</b>	<b>000</b>	<b>00, 01, 10</b>	<i>add / addi</i>	<b>000</b> (add)
<b>10</b>	<b>000</b>	<b>11</b>	<i>sub</i>	<b>001</b> (sub)
<b>10</b>	<b>010</b>	--	<i>slt / slti</i>	<b>101</b> (slt)
<b>10</b>	<b>110</b>	--	<i>or / ori</i>	<b>011</b> (or)
<b>10</b>	<b>111</b>	--	<i>and / andi</i>	<b>010</b> (and)

Tabelle 7 - Operationstabelle für den ALUDecoder-Block

## 5.3 Simulation

Um die gesamte Schaltung zu simulieren, steht Ihnen unter **HEIRV32\_test/heirv32\_mc\_tb** ein Prüfstand zur Verfügung. Sie führt den Code aus, der unter **Simulation/code\_sim.s** angegeben ist.

### 5.3.1 Verständnis

Öffnen und analysieren Sie den gegebenen Code.

- Welche Anweisungen werden ausgeführt?
- Ist er ein guter Kandidat, um die Funktionsweise Ihres Prozessors zu bestätigen?

### 5.3.2 Automatisierung von Tests

Der Tester **HEIRV32\_test/heirv32\_mc\_tester** zeigt in der Simulation ein Signal **testInfo** an, das theoretisch Auskunft darüber gibt, welcher Befehl gerade ausgeführt werden sollte. Im Falle eines nicht funktionierenden Prozessors ist diese Information wertlos.

Die Tests werden automatisiert durch das Verfahren:

```

1  procedure checkProc(
2    msg :          string;
3    AdrArg :       unsigned(31 downto 0);
4    ALUControlArg : std_ulogic_vector(2 downto 0);
5    ALUSrcAArg :   std_ulogic_vector(1 downto 0);
6    ALUSrcBArg :   std_ulogic_vector(1 downto 0);
7    IRWriteArg :   std_ulogic;
8    PCWriteArg :   std_ulogic;
9    adrSrcArg :   std_ulogic;
10   immSrcArg :  std_ulogic_vector(1 downto 0);
11   memWriteArg : std_ulogic;
12   regwriteArg : std_ulogic;
13   resultSrcArg : std_ulogic_vector(1 downto 0)) is
14 begin
15 ...
16 end procedure checkProc;
```

Sie wird in der Testerschleife wie folgt verwendet:

```
checkProc("Addi, addr. 0x00 - decode", x"00000004", "000", "01", "01", '0', '0', '0',
"00", '0', '0', "00");
```



Verifizieren Sie, dass Ihre Schaltung funktioniert.

## 5.4 Code

Sobald die Architektur durch eine Simulation bestätigt wurde, kann man die [FPGA](#) flashen.

Hierfür enthält die Bibliothek **Board** den Top-Level. Der geflashte Code ist der unter **Simulation/code\_mc\_ebs3\_bram.txt**.



Der angegebene Code **Simulation/code\_mc\_ebs3.s** ist leer, **Simulation/code\_mc\_ebs3\_bram.txt** dagegen nicht. Es liegt an Ihnen, einen neuen Code zu schreiben, sobald die Funktion der Schaltung bestätigt wurde.

### 5.4.1 Prüfung durch Äquivalenz

Flashen Sie zunächst den vorgegebenen Testcode und vergewissern Sie sich, dass er auf dieselbe Weise funktioniert wie der bereits auf dem Board verfügbare.

Lesen Sie dazu das Dokument [doc/Board\\_LFE5U-25F.pdf](#).



Bestätigen Sie die Funktionsweise Ihres Prozessors auf dem FPGA.

### 5.4.2 Benutzerdefinierter Code

Wenn die Schaltung funktioniert, schreiben Sie in **Simulation/code\_mc\_ebs3.s** einen eigenen Code, der auf das Drücken der beiden Knöpfe reagiert und die [LED](#) zum Leuchten bringt:

- Wenn Sie einen Wert in das Register **x30** schreiben, werden die [LED](#) eingeschaltet. Die [LED](#) 0 entspricht Bit 0, [LED](#) 1 entspricht Bit 1 ...
- Das Register **x30** zu lesen, gibt den aktuellen Status der [LED](#) an.
- Das Lesen der Schaltflächen erfolgt durch das Lesen des Registers **x31**. Bit 0 entspricht der Schaltfläche **S0**, Bit 1 der Schaltfläche **S1**.
- Das Schreiben des Registers **x31** verändert es nicht.

Testen Sie Ihren Code mit dem [RISC-V Online Interpreter](#) und [Ripes](#) bevor Sie ihn flashen.



Schreiben und testen Sie Ihren Code auf dem Simulator.

Sobald der Code bestätigt wurde, kompilieren Sie ihn mit der Software **HEIRV32-ASM**, die im gleichnamigen Ordner bereitgestellt wird, um eine neue Datei **code\_mc\_ebs3\_bram.txt** zu generieren.

Schliesslich müssen Sie den Top-Level neu kompilieren und die [FPGA](#) mit Ihrem neuen Code flashen.



## 5.5 Tips

Anbei noch einige zusätzlichen Tips um Probleme und Zeitverlust zu vermeiden:

- Teilen Sie das Problem in verschiedene Blöcke auf, benutzt hierzu das leere Toplevel Dokument. Es ist ein ausgeglichener Mix zwischen Anzahl Komponenten und Komponentengröße empfohlen.
- Analysieren Sie die verschiedenen Ein- und Ausgangssignale, ihre Arten, ihre Größen ... Sollte man sich teilweise auf die Datenblätter und die Dokumentation beziehen.
- Beachten Sie bei der Erstellung des Systems das DiD Kapitel „Methodologie für die Entwicklung von digitalen Schaltungen (MET)“ [10]
- Halten Sie sich an die vorgeschlagene inkrementelle Vorgehensweise. Benutzen Sie die Tests so früh wie möglich.
- Speichern und dokumentieren Sie Ihre Zwischenschritte. Nicht funktionierende Architekturen, grundlegende Codes zum Testen der Architektur ... sind allesamt Material, das dem Bericht hinzugefügt werden kann.



Vergessen Sie nicht Spass zu haben.





# Glossar

**FPGA** – Field Programmable Gate Array [2](#), [3](#), [4](#), [5](#), [8](#), [9](#), [19](#)

**LCD** – Liquid Crystal Display [9](#)

**LED** – Light Emitting Diode [2](#), [5](#), [6](#), [8](#), [9](#), [10](#), [19](#)

**PMod** – Peripheral Module [9](#), [10](#)

**PWM** – Pulse Width Modulation [6](#)

**RISCV** – 5-stages Reduced Instruction Set Computer architecture, open-sourced [3](#), [4](#), [5](#), [12](#)

**UART** – Universal Asynchronous Receiver Transmitter [6](#), [9](#)



# Literatur

- [1] A. Waterman, K. Asanovic, und F. Embeddev, „RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA“. Zugegriffen: 4. Juni 2022. [Online]. Verfügbar unter: <https://www.five-embeddev.com/riscv-isa-manual/latest/riscv-spec.html>
- [2] F. Embeddev, „RISC-V Quick Reference“. Zugegriffen: 4. Juni 2022. [Online]. Verfügbar unter: <https://www.five-embeddev.com/quickref/tools.html>
- [3] S. L. Harris und D. M. Harris, „Digital Design and Computer Architecture RISC-V Edition“, *Digital Design and Computer Architecture*. Elsevier, S. IBC1–IBC2, 2022. doi: [10.1016/B978-0-12-820064-3.00025-8](https://doi.org/10.1016/B978-0-12-820064-3.00025-8).
- [4] D. A. Patterson und J. L. Hennessy, *Computer Organization and Design - RISC-VEdition*, Second Edition. Elsevier, 2021.
- [5] Xilinx, „Spartan-3 FPGA Family“. Zugegriffen: 20. November 2021. [Online]. Verfügbar unter: <https://www.xilinx.com/products/silicon-devices/fpga/spartan-3.html>
- [6] Xilinx, „Datasheet Spartan-3E FPGA Family“. 2008.
- [7] Silvan Zahno, „Schematic: FPGA-EBS v2.2“. 2014.
- [8] Silvan Zahno, „Schematic: Parallelport HEB LCD V2“. 2014.
- [9] Electronic Assembly, „Datasheet: DOGM Graphics Series 132x32 Dots“. 2005.
- [10] François Corthay, Silvan Zahno, und Christophe Bianchi, „Methodologie Für Die Entwicklung von Digitalen Schaltungen“. 2021.
- [11] François Corthay, Silvan Zahno, und Christophe Bianchi, „Méthodologie de Conception de Circuits Numériques“. 2021.
- [12] Christophe Bianchi, François Corthay, und Silvan Zahno, „Wie Verfasst Man Einen Projektbericht?“. 2021.
- [13] Christophe Bianchi, François Corthay, und Silvan Zahno, „Comment Rédiger Un Rapport de Projet?“. 2021.
- [14] S. Zahno, „CAr RISC-V Summary (RISC-V)“, 2022.
- [15] D. Inc, „Pmod 8LD Reference Manual“. 2015.
- [16] D. Inc, „Pmod 8LD Schematics“. 2008.
- [17] D. Inc, „Pmod BB Reference Manual“. 2016.
- [18] D. Inc, „Pmod BB Schematics“. 2007.
- [19] D. Inc, „Pmod BTN Reference Manual“. 2016.
- [20] D. Inc, „Pmod BTN Schematics“. 2005.
- [21] D. Inc, „Pmod CON1 Reference Manual“. 2015.
- [22] D. Inc, „Pmod CON1 Schematics“. 2015.
- [23] D. Inc, „Pmod CON3 Reference Manual“. 2016.



- [24] D. Inc, „Pmod CON3 Schematics“. 2005.
- [25] D. Inc, „Pmod MAXSONAR Reference Manual“. 2015.
- [26] D. Inc, „Pmod OD1 Reference Manual“. 2016.
- [27] D. Inc, „Pmod OD1 Schematics“. 2006.
- [28] Laurent Gauch, „Schematic: HEB Dot Matrix v1.0“. 2003.
- [29] Maxbotix, „LV-MAXSONAR-EZ Datasheet“. 2021.
- [30] Sitronix, „Datasheet Sitronix ST7565R 65x1232 Dot Matrix LCD Controller/Driver“. 2006.