



# HEIRV32 (HEIRV32)

Vorlesung Computerarchitektur (CAr)



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

**Spezialisierung:** Data Engineering (DE)

**Kurs:** Computerarchitektur (CAr)

**Authoren:** Silvan Zahno, Axel Amand

**Datum:** 28.10.2025

**Version:** v4.0



# 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 Einfach .....	6
2.2.2 Mittel .....	6
2.2.3 Schwer .....	6
2.3 HDL-Designer Projekt .....	8
2.4 Bereitgestellte Blöcke .....	9
2.5 Anzeige auf der Tochterplatine .....	10
2.6 Simulator .....	10
3 Komponenten .....	11
3.1 EBS3 Field Programmable Gate Array (FPGA) Platine .....	11
3.2 Knöpfe und Light Emitting Diodes (LEDs) .....	11
3.3 Zusätzliche LEDs .....	12
3.4 Optionale Karten .....	12
4 Bewertung .....	13
5 Leitfaden .....	14
5.1 Allgemeine Architektur .....	14
5.1.1 Signalen .....	14
5.1.2 Instruktion lw .....	14
5.1.3 Instruktion sw .....	17
5.1.4 Instruktion R- und I-Type .....	17
5.1.5 Instruktion beq .....	17
5.1.6 Instruktion jal .....	17
5.1.7 Instruktion jalr .....	18
5.2 Control Unit .....	19
5.2.1 Main FSM .....	20
5.2.2 ALU Decoder .....	20
5.2.3 Instr. Decoder .....	21
6 Tests .....	22
6.1 Simulation .....	22
6.1.1 Verständnis .....	22
6.1.2 Automatisierung von Tests .....	22
6.2 Code .....	23
6.2.1 Prüfung durch Äquivalenz .....	23
6.2.2 Benutzerdefinierter Code .....	24
6.3 Tips .....	25
Glossar .....	26

# 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 (RISC-V) Prozessor zu erstellen, um einen kleinen Assemblerprogramm auszuführen. Der Prozessor ist zuerst simuliert und dann auf einem FPGA implementiert. Die Verbindung mit der Außenwelt kann über Knöpfe und Leds erfolgen. Dieses Prozessorsystem ist in der Abbildung 1 dargestellt.

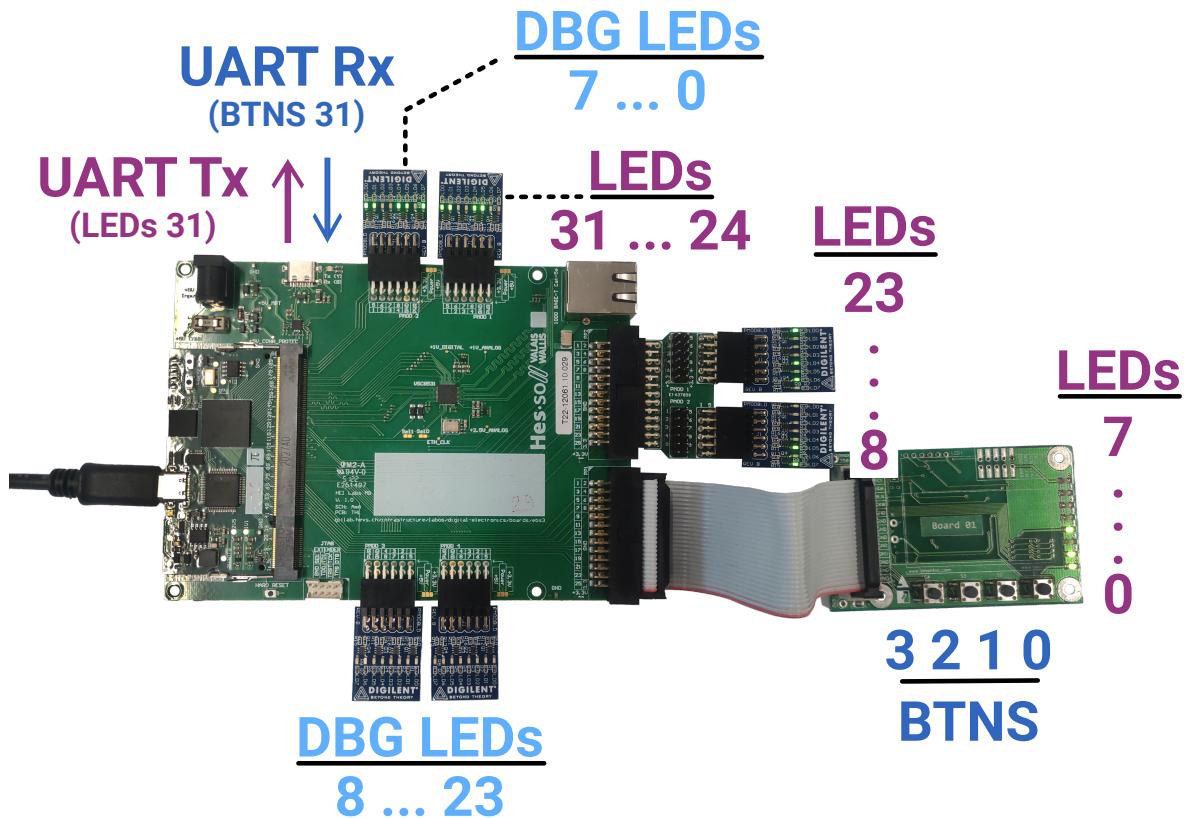


Abbildung 1 - Hardwareaufbau des Systeme (EBS3)

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 maximalen Geschwindigkeiten sind bei kontinuierlichem Zugriff begrenzter als bei synchronen Speichern
  - die Steuerung ist komplexer, erfordert präzise Timings und die Chips sind anfällig für sogenannte „Race Conditions“
- 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 ...
- *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 ⇒ 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 **RISCV** Mikroprozessor deren Funktionsweise durch Simulation und anschliessenden Einsatz auf einem physischen Chip bestätigt wird.
- Der implementierte Microprozessor muss die folgende Befehle unterstützen:
  - R-Type Befehle: **add, sub, and, or, slt, xor, sll, srl**
  - I-Type Befehle: **addi, andi, ori, slti, xori, slli, srli**
  - Speicher Befehle: **lw, sw**
  - Sprung Befehle: **beq, jal, jalr**
- Mit den unterstützten Befehlen wird ein Assembler-Code geschrieben, der in der Lage ist:
  - das Drücken der Tasten auf der elektronischen Platine Tasten-LEDs [Abbildung 5](#) zu erkennen
  - die LEDs der elektronischen Platine Tasten-LEDs [Abbildung 5](#) zu steuern
  - die beiden vorherigen Funktionen zu kombinieren, um ein interaktives Verhalten zu demonstrieren
  - die Funktionen eines einfachen Codes zu erfüllen (siehe [Abschnitt 2.2.1](#))

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 - **RISCV** Hardware Schaltung

Eine funktionale Architektur, ein „einfacher“ Code (siehe [Abschnitt 2.2.1](#)) und ein perfekter Bericht ergibt maximal eine Note von



5

Ein oder mehrere fortgeschrittene Codes sind erforderlich, um eine höhere Note zu erhalten.

## 2.2 Support für zusätzliche Befehlen

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



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 Einfach

- **Tastenverwaltung:** verschiedene Muster auf den 32 LED anzeigen, je nach Tastendruck. Die zuletzt gedrückte Taste wird gespeichert, um das Muster anzuzeigen.
- **Zeitmanagement:** die LED mit einer bestimmten Frequenz blinken lassen. Das Drücken der Tasten ermöglicht es, die Frequenz zu erhöhen/zu verringern.

### 2.2.2 Mittel

- **Lauflicht auf den LED :** die LED leuchten nacheinander mit einer moderaten Frequenz auf; das Drücken einer Taste stoppt das Lauflicht, während eine andere es von 0 neu startet. Die Tasten müssen in der Software **entprellt** werden, und ein Tastendruck darf nur einmal berücksichtigt werden, bis er losgelassen und erneut gedrückt wird.
- **Variable Intensität der LED :** die LED leuchten mit einer auf 50% eingestellten **Pulse Width Modulation (PWM)**. Das Drücken einer Taste erhöht die Intensität bei jedem Druck um 10%, während eine andere sie um 10% verringert. Die Tasten müssen in der Software **entprellt** werden, und ein Tastendruck darf nur einmal berücksichtigt werden, bis er losgelassen und erneut gedrückt wird.

### 2.2.3 Schwer

- **Universal Asynchronous Receiver Transmitter (UART) :** beim Drücken einer Taste wird eine Textübertragung über **UART** durchgeführt. Der Text wird auf einem PC angezeigt.
- **Serielle RGB LED :** Verwaltung von seriellen RGB LED vom Typ WS2812/WS2813. Das Drücken einer Taste schaltet die LED ein/aus; zwei weitere ermöglichen das Wechseln zur nächsten/vorherigen Farbe.
- **Atmende LED :** die LED leuchten allmählich auf und erlöschen dann, was einen „Atmungseffekt“ erzeugt. Eine Taste erhöht die Atemgeschwindigkeit, eine andere verringert sie. Zwei weitere ändern die Intensität der LED .
- **Ultraschall-Abstandssensor:** Verwaltung eines PMOD Maxsonar Ultraschallsensors - **Tabelle 3** - zur Anzeige eines Distanzniveaus auf den LED .

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 von [Git](#) geklont 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:

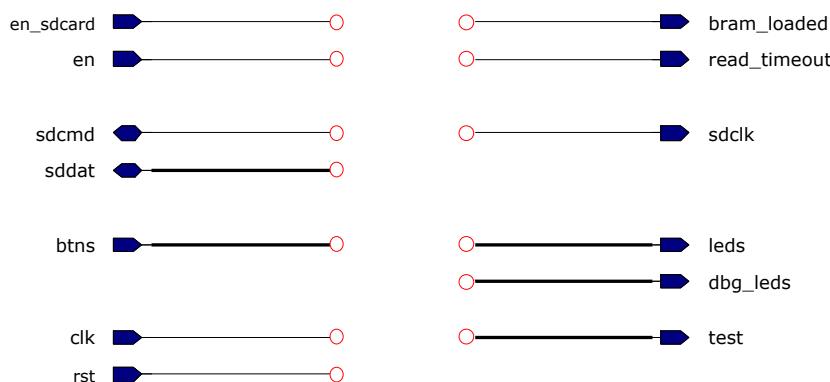


Abbildung 3 - Leere Toplevel Signale



Die verfügbaren Signale arbeiten in den folgenden Gruppen:

- System
  - **clk** : Systemtakt, getaktet auf **25MHz**
  - **rst** : asynchroner Systemreset
  - **test[47:0]** : Vektor, der mehrere Systemsignale gruppiert, um Tests in der Simulation zu automatisieren
  - **en\_sdcard** : Signal zum Aktivieren des Lesens des Codes von der SD-Karte - *standardmäßig in der Studentenlösung unbenutzt*
  - **en** : unabhängiges Signal zum Aktivieren/Deaktivieren des Prozessors - *standardmäßig in der Studentenlösung unbenutzt*
- Prozessor
  - **bram\_loaded** : Anzeige, dass das Programm korrekt von der SD-Karte geladen wurde, angezeigt mithilfe der **grünen LED**
  - **read\_timeout** : Anzeige, dass das Programm von der SD-Karte nicht gefunden/geladen werden konnte
- SD-Karte
  - **sdclk**, **sdcmd**, **sddat[3:0]** : Takt, Befehl und Daten der SD-Karte
- Ein-/Ausgänge
  - **btms[3:0]** : Tasten S4 bis S1 der elektronischen Platine Tasten-LEDs [Abbildung 4](#), direkt mit dem Register **x31** verbunden
  - **leds[31:0]** : **LED** oder andere Ausgänge, die entsprechend dem Register **x30** aktiviert werden
  - **dbg\_leds[31:0]** : Debug-Signale, die es ermöglichen, **LED** entsprechend internen Signalen zu aktivieren. *Sind nicht mit dem Register x30 verbunden.*
- Serieller Port (UART)
  - Der auf der Hauptplatine vorhandene USB-C-Port kann als serieller UART-Port verwendet werden, um mit einem PC zu kommunizieren, wie es für einen fortgeschrittenen Code vorgeschlagen wird [Abschnitt 2.2.3](#).
  - **uart\_tx** : UART-Übertragung vom RISC-V-Prozessor, verknüpft mit Register **leds[31]**
  - **uart\_rx** : UART-Empfang zum RISC-V-Prozessor, verknüpft mit Register **btms[31]**

## 2.4 Bereitgestellte Blöcke

Alle Blöcke, die für das Design des Mikroprozessors erforderlich sind, sind im Block **heirv32\_mc** enthalten und stammen aus den folgenden Bibliotheken:

- HEIRV32\_MC
  - **controlUnit** : Block für die Befehlsdekodierung
  - **heirv32\_mc** : Top-Level
  - **instructionDataManagerSDCard** : Programmspeicher, der Befehle und Daten gruppiert, lesend und schreibend, den Inhalt der SD-Karte liest
- HEIRV32
  - **ALU** : eine Version der ALU, die Addition, Subtraktion, AND, OR und SLT kann
  - **buffer\*(Enable)** : getakteter Puffer (Flip-Flops) mit oder ohne Enable-Eingang
  - **extend** : Befehlserweiterungsblock für Tests, unterstützt die Befehle I, S, B und J
  - **mux3To1ULogVec** : Mux 3 zu 1 von **std\_ulogic\_vector**
  - **registerFile** : Block zur Verwaltung der 32 Register, ersetzt **x31** durch den Vektor **btms** - *Tastenlese-Register* - und **x30** durch den Vektor **leds** - *LED-Schreibregister* -

Die **Board**-Bibliothek enthält die Signalaufbereitungslogik, die für die Bereitstellung des Schaltkreises auf der **FPGA** vorgesehen ist.



Ändern Sie nicht die Namen der bereits vorhandenen Signale im Top-Level. Der Vektor **test[47:0]** basiert auf diesen Namen.

## 2.5 Anzeige auf der Tochterplatine

Die Anzeige- **LED** auf der Tochterplatine wird verwendet, um den Systemstatus anzuzeigen, in der Reihenfolge:

- **Blau LED** : System-Heartbeat bei 2 [Hz] (*Lösungs-Version*) / 8 [Hz] (*Studenten-Version*), zeigt an, dass das **FPGA**-Design aktiv ist.

*Wenn die SD-Karte nicht eingelegt ist, blinkt die LED nicht, da das System im Reset gehalten wird.*

- **Grüne LED** : zeigt an, dass das **Programm** korrekt von der SD-Karte **geladen** wurde und der Prozessor zur Codeausführung aktiviert ist.

- **Rote LED** : zeigt an, dass der **Prozessor aktiviert** ist.

*Der Schalter S4 aktiviert/deaktiviert den Prozessor. Der Schalter S3 aktiviert den Prozessor für einen Taktzyklus, wenn er deaktiviert ist. Diese beiden Schalter haben in der Studenten-Version keine Wirkung (Prozessor immer aktiviert).*

## 2.6 Simulator

Die Bibliothek **HEIRV32\_test** enthält den Tester **heirv32\_mc\_tb** zum Simulieren der Prozessorausführung.

Der Prozessor lädt die Datei **Simulation/code\_sim\_bram.txt** in den Speicher, die der Assembler-Code **Simulation/code\_sim.s** ist, kompiliert von **heirv32-asm/HEIRV32-ASM\_xxx**.

# 3 | Komponenten

Das System besteht aus drei verschiedenen Hardwareplatinen, die in der Abbildung [Abbildung 1](#) zu sehen sind.

- eine [FPGA](#) Entwicklungskarte, siehe Abbildung [Abbildung 4](#).
- Eine Steuerplatine mit 4 Tasten und 8 [LEDs](#), siehe Abbildung [Abbildung 5](#).
- Zwei [Peripheral Module \(PMod\)](#) [LEDs](#) 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  $f_{\text{clk}} = 25\text{MHz}$  reduziert.

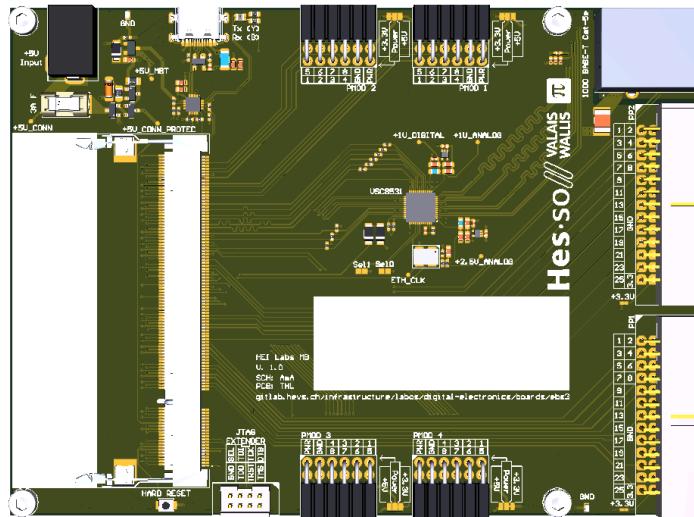


Abbildung 4 - [FPGA](#) Platine

## 3.2 Knöpfe und [LEDs](#)

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

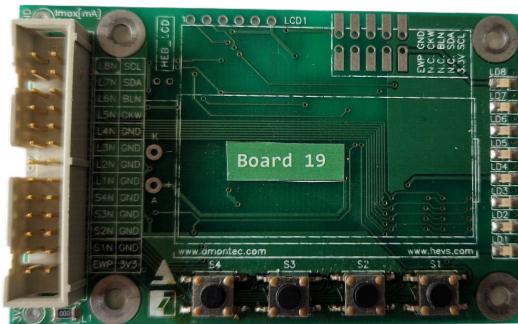


Abbildung 5 - Knopf LED Platine [8]

### 3.3 Zusätzliche LEDs

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](#).

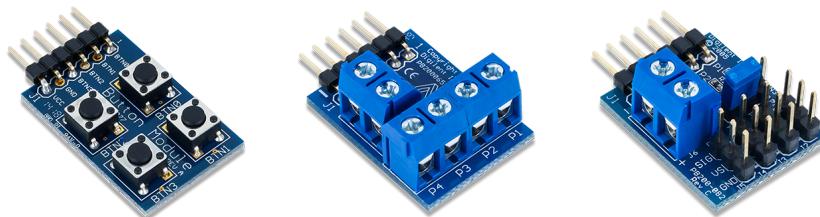


Tabelle 1 - Inputs: [PMOD BTN](#) [19], [20], [PMOD CON1](#) [21], [22], [PMOD CON3](#) [23], [24]

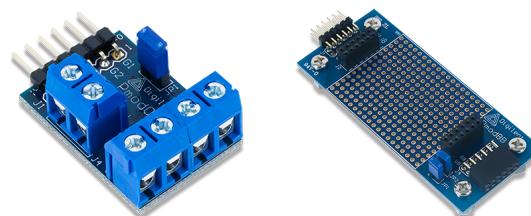


Tabelle 2 - Outputs: [PMOD OD1](#) [26], [27], [PMOD BB](#) [17], [18]



Tabelle 3 - I/Os: [PMOD MAXSONAR](#) [25], [29]

# 4 | Bewertung

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

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

Bewertete Aspekte	Punkte
<b>Bericht</b>	<b>45</b>
Einleitung	3
Spezifikation	5
Entwurf	15
Verifizierung und Validation	5
Integration	9
Schlussfolgerung	3
Formale Aspekte des Berichtes	5
<b>Funktionalität der Schaltung</b>	<b>60</b>
Minimale Funktionen	30
Zusatzfunktion 1	15
Zusatzfunktion 2	15
<b>Qualität der Lösung</b>	<b>15</b>
<b>Presentation</b>	<b>30</b>
<b>Total</b>	<b>150</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 setzen Sie Ihren eigenen Code ein

## 5.1 Allgemeine Architektur

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



Abbildung 7 - [RISCV Pipeline](#)

### 5.1.1 Signal en

Der Eingang **en** ermöglicht es, den Betrieb des Prozessors zu stoppen.

Er ist standardmäßig auf „1“ gesetzt, um die Verwendung der 4 Tasten zu ermöglichen, ohne dass das System stoppt.

**Achten Sie darauf, dieses Signal so zu belassen, wie es bereits in den gegebenen Blöcken verdrahtet ist!**

### 5.1.2 Instruktion lw

Das Design dreht sich um den Programmspeicher, indem man über die **lw**-Anweisung nachdenkt, eine Anweisung, die die fünf Schritte der Pipeline erfordert.

Die Anweisung **lw rd, imm(rs1)** lässt sich wie folgt zusammenfassen:

- *Fetch*: Die Anweisung wird aus dem Speicher geholt
- *Decode*: Das Register **rs1** wird aus der Anweisung extrahiert, ebenso wie der Wert **imm**, der auf 32 Bit erweitert wird
- *Execute*: Die Speicheradresse wird berechnet, indem **rs1** und **imm** addiert werden
- *Mem. Access*: Der Speicher wird an der zuvor berechneten Adresse angesprochen, wodurch der Wert in das Register **rd** geladen wird
- *Write Back*: Der Wert wird im Register **rd** gespeichert

In Bezug auf die Schaltung ergibt die Anweisung **lw rd, imm(rs1)**:

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

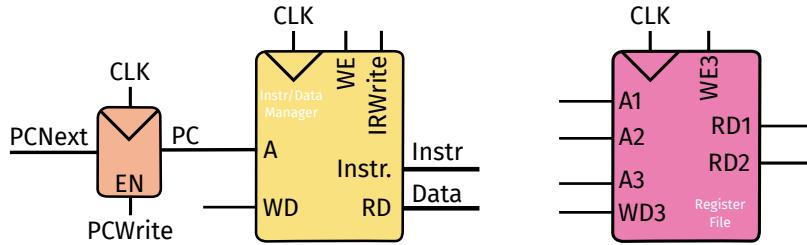


Abbildung 8 - Fetch

- **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 Flip-Flops, die diesen Schritt trennen, sind im **register file**-Block versteckt

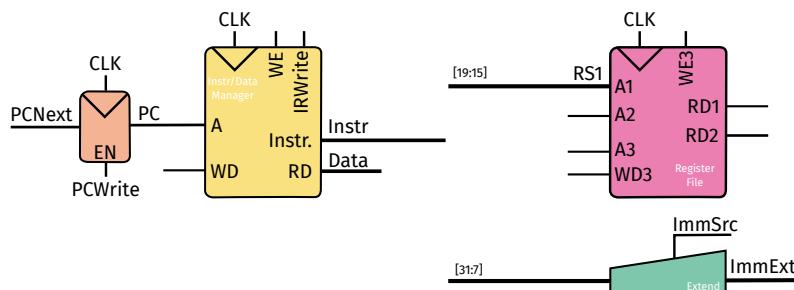


Abbildung 9 - Decode

- **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.

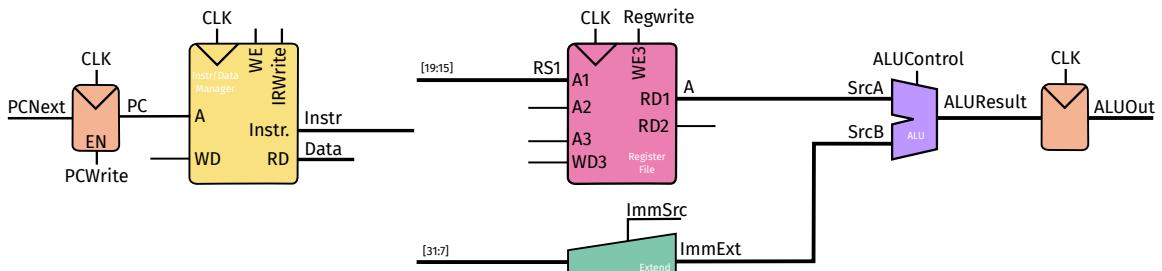


Abbildung 10 - Execute

- **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**.

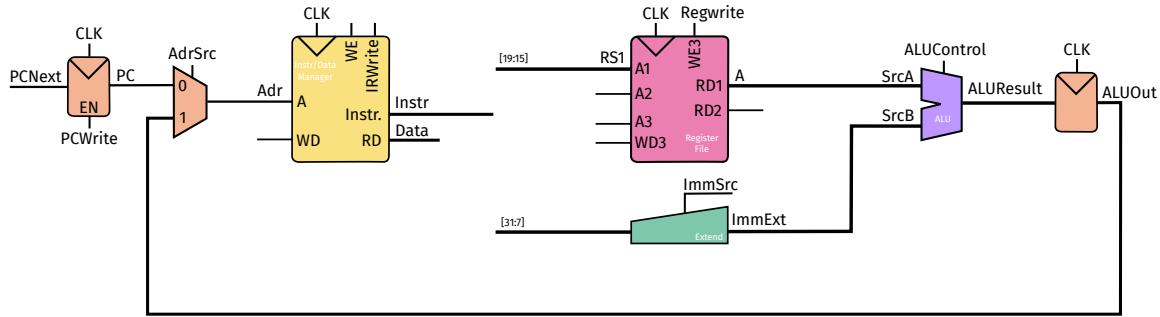


Abbildung 11 - Memory Access

- 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:

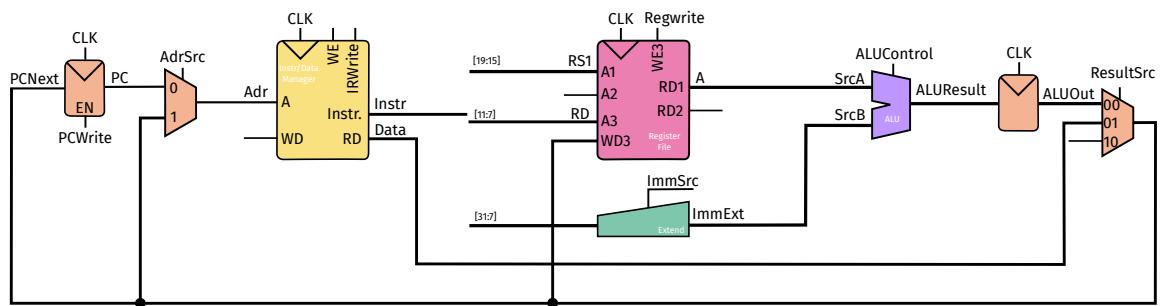


Abbildung 12 - Write Back

#### PC + 4

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:

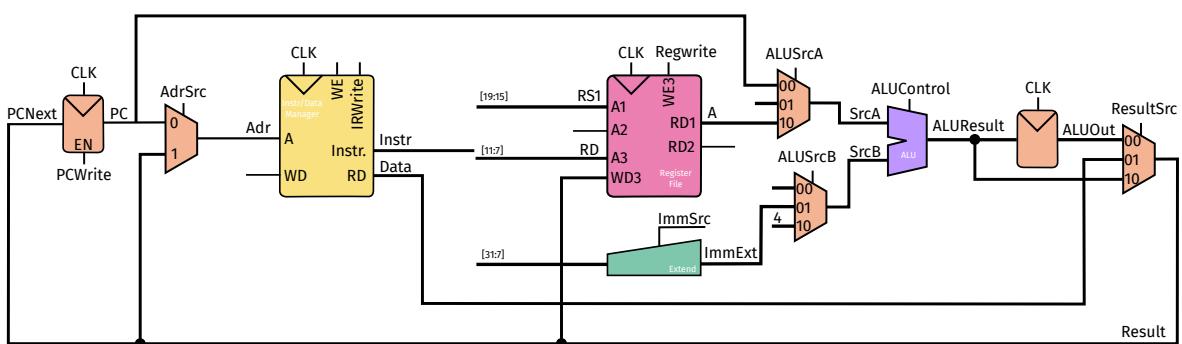


Abbildung 13 - PC + 4

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

### 5.1.3 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:

- *Fetch*: liest die Anweisung aus dem Speicher, auf den **PC** zeigt.
- *Decode*: lädt die notwendigen Quellregister.
- *Execute*: führt die von der Anweisung geforderte Operation über die ALU aus.
- *Write Back*: speichert das Ergebnis im Zielregister, falls notwendig.

### 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 (**oldPC**) 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:  $a == b$ ,  $PC = PC + imm..$  Andernfalls bleibt **PCWrite** auf '0' und **PC** wird nicht geändert:  $a \neq b$ ,  $PC = PC + 4$ .

### 5.1.6 Instruktion jal

Fügen Sie 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 **oldPC** + **imm**.
- *Execute*: speichert die Sprungadresse als neuen **PC**, während sie die im Zielregister zu speichernde Rücksprungadresse **oldPC** + 4 berechnet.
- *Write Back*: speichert die Rücksprungadresse im Zielregister.

### 5.1.7 Instruktion jalr

Überlegen Sie sich den notwendigen Pfade für die Anweisung **jalr**. Es funktioniert ähnlich wie **jal**, springt jedoch nicht zu **oldPC + imm**, sondern zu der Adresse **rs1 + imm**.

- *Fetch*: liest die Anweisung aus dem Speicher, auf den **PC** zeigt.
- *Decode*: gibt dem Lesen von **rs1** aus dem Register Zeit.
- *Execute1*: berechnet die Sprungadresse **rs1 + imm**.
- *Execute2*: speichert die Sprungadresse als neuen **PC**, während sie die im Zielregister zu speichern-de Rücksprungadresse **oldPC + 4** berechnet.
- *Write Back*: speichert die Rücksendeadresse im Zielregister.



*Es ist hier zu beachten, dass der Ausführungsschritt in zwei Unterschritte unterteilt ist. Im Fall von **JAL** war es möglich, die Sprungadresse während des Dekodierungsschritts zu berechnen, da **oldPC** und **imm** dort bereits auf der ALU verfügbar sind (der Block zur Erweiterung des unmittelbaren Wertes erfordert keinen Clock-Schlag).*

*Im Fall von **JALR** muss der Dekodierungsschritt abgeschlossen sein, bevor der Wert des Registers **rs1** verwendet werden kann.*

*Mehrere Strategien sind denkbar, darunter Änderungen der Gesamtarchitektur für vollständigere Prozessoren. In unserem Fall ist das Hinzufügen eines zweiten Ausführungsschritts die einfachste Lösung, um dieses Problem zu umgehen.*

## 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:

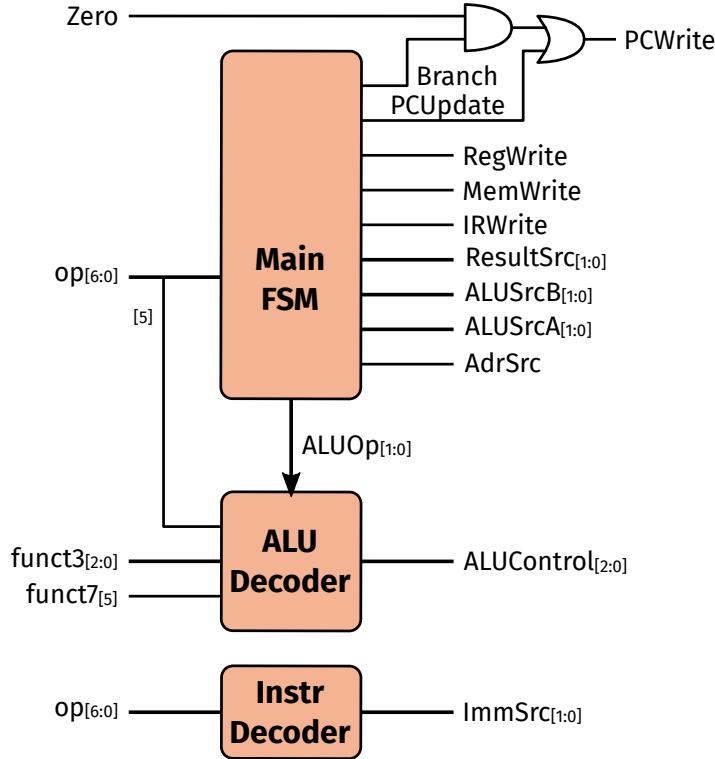


Abbildung 14 - Control unit



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 noch kein Betriebssystem betreiben kann!

### 5.2.1 Main FSM

Der Zustandsautomat steuert die Steuersignale des Prozessors.

Er ist dafür verantwortlich, die verschiedenen Schritte der Pipeline zu verfolgen, um jeden Block entsprechend zu konfigurieren.

Einige Schritte sind für mehrere Anweisungen gemeinsam, andere spezifisch für einen bestimmten Typ.

*Das Signal **ALUOp** wird unter den Abschnitt 5.2.2.1 erklärt.*

### 5.2.2 ALU Decoder

Die ALU führt arithmetische und logische Funktionen aus, die durch das Signal **ALUControl** gemäß der folgenden Tabelle festgelegt sind:

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>100</b>	<i>XOR</i>
<b>101</b>	<i>Set Lesser Than</i>
<b>110</b>	<i>Shift Left Logical</i>
<b>111</b>	<i>Shift Right Logical</i>
Others	-

Tabelle 5 - Operationstabelle für den ALU-Block

Auf der Grundlage verschiedener Signale sollte der Block **ALU Decoder** in der Lage sein, die laufende mathematische/logische Operation zu steuern.

#### 5.2.2.1 Signal ALUOp

Es gibt Pipeline-Schritte, bei denen die ALU von der FSM anstelle der Anweisung verwendet wird. Zum Beispiel wird die ALU während des **Fetch**-Schrittes verwendet, um die nächste Adresse zu berechnen, und muss daher auf Addition eingestellt werden.

Der **Main FSM** generiert also das Signal **ALUOp**, das ein 2-Bit-Code ist, der den Block **ALU Decoder** zwingt, eine bestimmte Operation auszuführen:

operation	ALUOp <sub>[2:0]</sub>
add	<b>00</b>
sub	<b>01</b>
instruction based	<b>10</b>
X	<b>11</b>

Tabelle 6 - Operationstabelle für den ALUDecoder-Block

Einige Anweisungen sind auch durch ihre Funktionsweise miteinander verbunden, auch wenn sie einen anderen Typ haben. 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	$Op_5 \cdot funct7_5$	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>0·0, 0·1, 1·0</b>	<i>add / addi</i>	<b>000</b> (add)
<b>10</b>	<b>000</b>	<b>1·1</b>	<i>sub</i>	<b>001</b> (sub)
<b>10</b>	<b>001</b>	--	<i>sll / slli</i>	<b>110</b> (sll)
<b>10</b>	<b>010</b>	--	<i>slt / slti</i>	<b>101</b> (slt)
<b>10</b>	<b>100</b>	--	<i>xor / xori</i>	<b>100</b> (xor)
<b>10</b>	<b>101</b>	<b>··0</b>	<i>srl / srli</i>	<b>111</b> (srl)
<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

### 5.2.3 Instr. Decoder

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>
--	<i>Others</i>

Tabelle 8 - Operationstabelle für den Extend-Block

Der Block **Instr. Decoder** muss also, basierend auf dem Operanden **op[6:0]**, die Anweisung identifizieren und die richtige Konfiguration für **ImmSrc** ableiten.



Vervollständigen Sie die Blöcke **Main FSM**, **ALU Decoder** und **Instr. Decoder**, um die unter **Abschnitt 2** aufgeführten Anweisungen zu unterstützen.



Denken Sie daran, zu überprüfen, zu welchem Typ jede Anweisung gehört (R, I, S, B, U, J). Der Name der Anweisung selbst kann irreführend sein!

# 6 | Tests

## 6.1 Simulation

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

### 6.1.1 Verständnis

Öffnen und analysieren Sie den gegebenen Code.



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

### 6.1.2 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");
```

Sie ermöglicht es einfach, den aktuellen Zustand des Prozessors mit dem erwarteten Zustand zu vergleichen. Im Fehlerfall stoppt der Test.

Ihre Implementierung kann leicht von der gegebenen Lösung abweichen, je nach Ihren Implementierungsentscheidungen. Es liegt an Ihnen, den Testcode bei Bedarf manuell zu beurteilen und zu korrigieren.



Verifizieren Sie, dass Ihre Schaltung funktioniert.

## 6.2 Code

Sobald die Architektur durch eine Simulation bestätigt wurde, kann man die **FPGA** flashen.

Hierfür enthält die Bibliothek **Board** das Top-Level. Der geflashte Code ist der unter **Simulation/code.bin**.



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

### 6.2.1 Prüfung durch Äquivalenz

Testen Sie Ihr System zunächst mit der gegebenen Lösung:

- Ziehen Sie die Entwicklungsplatine **FPGA** ab.
- Laden Sie den Code auf die micro SD-Karte, indem Sie die Datei **Simulation/code.bin** kopieren und den gleichen Namen beibehalten. Eine Kopie des Codes ist auch auf der SD-Karte verfügbar  
- *löschen Sie sie nicht von der Karte.*
- Stecken Sie die Platine wieder ein.
- Schließen Sie die Stromversorgung über ein USB-C-Kabel an.
- Überprüfen Sie das Laden des Codes, die Funktion der Tasten, das Aufleuchten der **LED** und das allgemeine Verhalten des Systems.

Flashen Sie dann Ihren Prozessor und wiederholen Sie die Tests. Stellen Sie sicher, dass alles korrekt funktioniert.

Beziehen Sie sich für das Flashen der **FPGA** auf das Dokument **doc/Board\_LFE5U-25F.pdf**.



Flashen Sie die FPGA zunächst über JTAG (vorübergehend). So können Sie durch Ab- und Wiederanschließen der Stromversorgung zur Lösungsvariante zurückkehren.

Sobald alles validiert ist, programmieren Sie die Konfiguration in den Flash, um Ihren Prozessor dauerhaft zu behalten.



Bestätigen Sie die Funktionsweise Ihres Prozessors auf dem FPGA.

### 6.2.2 Benutzerdefinierter Code

Wenn die Schaltung funktionsfähig ist, schreiben Sie Ihren eigenen Code in eine **.s**-Datei, die auf die Betätigung der beiden Tasten reagieren und die **LED** einschalten kann:

- Das Schreiben eines Wertes in das Register **x30** schaltet die **LED** ein. **LED** 0 entspricht Bit 0, **LED** 1 Bit 1 ...
- Das Lesen des Registers **x30** gibt den aktuellen Zustand der **LED** zurück.
- Das Lesen der Tasten erfolgt durch Lesen des Registers **x31**. Bit 0 entspricht der Taste **S0**, Bit 1 der Taste **S1**.
- Das Schreiben des Registers **x31** ändert es nicht.

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



Schreiben und testen Sie Ihren Code im Simulator.

Sobald der Code validiert ist, kompilieren Sie ihn mit der Software [heirv32-asm/HEIRV32-ASM\\_xxx](#), um eine neue Datei **code.bin** zu erstellen.

Um den Code zu laden, kopieren Sie die generierte Datei **xxxx.bin** auf die microSD-Karte und benennen Sie sie in **code.bin** um.

Entfernen Sie den Stecker der Entwicklungsplatine, stecken Sie die microSD-Karte ein und verbinden Sie diese schlussendlich wieder.



Bestätigen Sie die Funktionsweise Ihres Codes.



### 6.3 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](#), [5](#), [10](#), [11](#), [23](#)

**LCD** – Liquid Crystal Display [11](#)

**LED** – Light Emitting Diode [2](#), [5](#), [6](#), [9](#), [10](#), [11](#), [12](#), [23](#), [24](#)

**PMod** – Peripheral Module [11](#), [12](#)

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

**RISCV** – 5-stages Reduced Instruction Set Computer architecture, open-sourced [3](#), [4](#), [5](#), [14](#)

**UART** – Universal Asynchronous Receiver Transmitter [6](#), [11](#)



# 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.