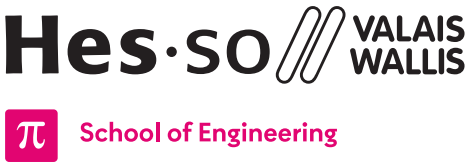

# HEIRV32 (HEIRV32)

## Lecture Computer Architecture (CAr)




**Orientation**: Information and Communication Technology (ISC)
**Specialisation**: Data Engineering (DE)
**Course**: Computer Architecture (CAr)
**Authors**: Silvan Zahno, Axel Amand
**Date**: 28.10.2025
**Version**: v4.0


# Contents

# 1 Introduction

The goal of the project is to directly apply the knowledge acquired at the end of the semester using a practical example. The aim is to create a reduced 5-stages **R**educed **I**nstruction **S**et **C**omputer architecture, open-sourced (RISCV) processor to run a small assembly program. The processor is first simulated and then deployed on a FPGA . The connection with the outside world can be done using buttons and leds. This processor system is represented in the Figure 1.

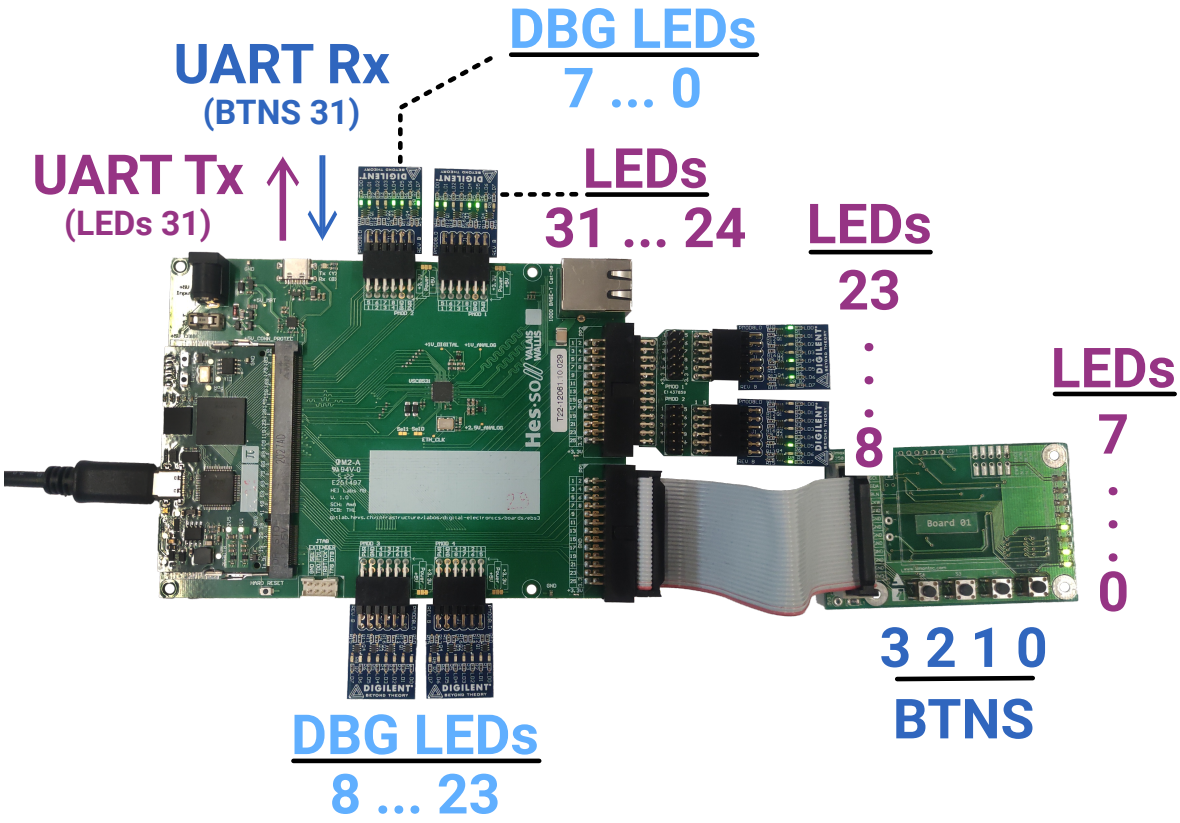



Figure 1 - System setup (EBS3)

The goal is to achieve the minimal Specification defined in Section 2.

## 1.1 Multi-cycle HEIRV32 microprocessor

The single-cycle architecture, completed in a previous laboratory, allows for a simple approach to the RISCV architecture. However, it has several problems:

- program and data memory are separated
  - requires to implement two chips for a single program
- memory is accessed asynchronously
  - maximum speeds are more limited than for synchronous memories in case of continuous access
  - control is more complex, requiring precise timings, and chips are subject to race conditions
- the speed of the processor is limited by the longest instruction
  - improvements are only possible through the evolution of transistor technologies and the reduction of routing lengths

Fundamentally, instructions consist of 5 steps:

- *Fetch*: the instruction is retrieved from memory
- *Decode*: the instruction is decoded: **`funct3`**, **`funct7`**, ALU and source settings ...
- *Execute*: the specified operation is executed
- *Memory*: accesses specific data from memory (optional)
- *Writeback*: stores data in memory (optional)

Since not all instructions require the execution of each step, these steps can be dissociated, separated by a clock cycle ⇨ multi-cycle. Thus, the clock frequency can be increased, the speed now being limited by the slowest step.

Shorter instructions requiring only 3 to 4 steps will benefit from faster processing.

The memories used are now synchronous.

This architecture also paves the way for the implementation of a pipeline system (i.e. loading an instruction at each clock cycle), for branch prediction, and any other technique that speeds up the general operation of the processor.

Only multi-cycle is addressed here.

### 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 Specification

## 2.1 Functions

The basic functions are defined as follows:

- The FPGA implements a 32-bit, multi-cycle RISCV microprocessor whose operation is confirmed by simulation and then deployed on a physical chip.
- The implemented microprocessor must support the following instructions:
  - R-Type instructions: **`add`**, **`sub`**, **`and`**, **`or`**, **`slt`**, **`xor`**, **`sll`**, **`srl`**
  - I-Type instructions: **`addi`**, **`andi`**, **`ori`**, **`slti`**, **`xori`**, **`slli`**, **`srli`**
  - Memory instructions: **`lw`**, **`sw`**
  - Jump instructions: **`beq`**, **`jal`**, **`jalr`**
- Using the supported commands, an assembler code is written and tested, capable of:
  - detecting the press of the buttons on the electronic board buttons-LEDs Figure 5
  - controlling the leds of the electronic board buttons-LEDs Figure 5
  - combining the two previous functionalities to demonstrate an interactive behavior
  - fulfilling the functionalities of a simple code (see Section 2.2.1)

To control the LED , simply write the register **`x30`**.

To read the buttons, simply read the register **`x31`**. A write to this register does not change its value.

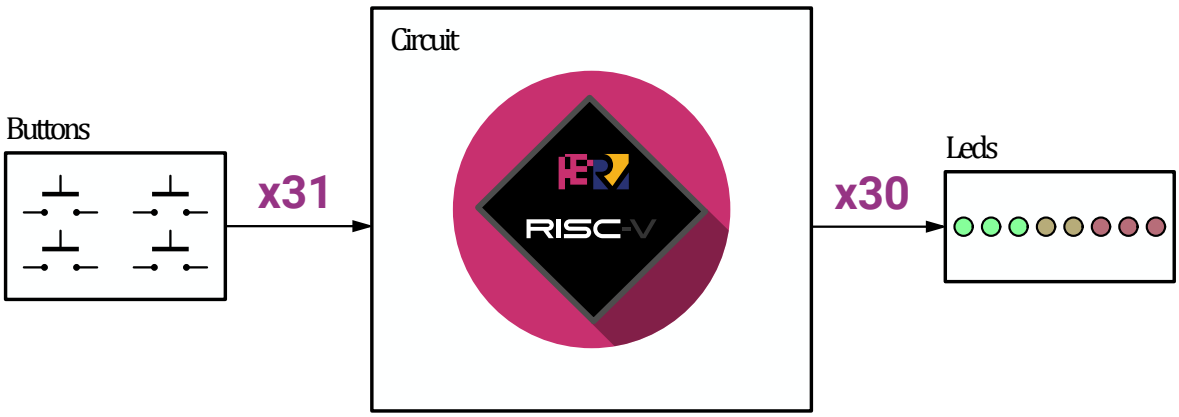



Figure 2 - RISCV Hardware circuit

A functional architecture, an “easy” code (see Section 2.2.1) and a perfect report are worth a maximum grade of

5

One or more advanced codes are required to obtain a higher grade.


## 2.2 Additional functions

Algorithms bring additional points depending on their difficulty.

Be careful to only use the implemented instructions.
*Using a RISC-V compiler will **NOT** provide you with compatible code (unsupported instructions, section directives ...).*

### 2.2.1 Easy

- **Button management**: display different patterns on the 32 LED according to the button press. The last pressed button is saved to leave the pattern displayed.
- **Timing management**: make the LED blink at a given frequency. Pressing the buttons allows to increase/decrease the frequency.

### 2.2.2 Medium

- **Chaser on the LED** : the LED light up one after the other at a moderate frequency; pressing a button stops the chaser, while another restarts it from 0. The buttons must be debounced in software and a press must be taken into account only once until it is released and pressed again.
- **Variable intensity LED** : the LED light up with a 50% set **P**ulse **W**idth **M**odulation (PWM) . Pressing a button increases the intensity by 10% with each press, while another decreases it by 10%. The buttons must be debounced in software and a press must be taken into account only once until it is released and pressed again.

### 2.2.3 Hard

- **Universal Asynchronous Receiver Transmitter (UART)** : when a button is pressed, a text transmission via UART is performed. The text is displayed on a PC.
- **Serial RGB LED s**: management of serial RGB LED s type WS2812/WS2813. Pressing a button turns on/off the LED s; two others allow to switch to the next/previous color.
- **Breathing LED s**: the LED s light up and then turn off gradually, giving a "breathing" effect. One button increases the breathing speed, another decreases it. Two others modify the intensity of the LED s.
- **Ultrasonic distance sensor**: management of a PMOD Maxsonar ultrasonic sensor - Table 3 - to display a distance level on the LED s.

**Of course, it is possible to propose your own ideas and to use additional material (see Section 3).**

It is highly recommended to develop and test your algorithm with the tools seen during the ISA labs (RISC-V online interpreter and Ripes)
It is possible to manually modify the registers on Ripes to simulate the buttons.

## 2.3 HDL Designer project

A predefined HDL-Designer project can be downloaded from Cyberlearn or cloned from Git. The file structure of the project is as follows:

```
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
```

The path to the project folder must not contain spaces.

The **doc/** folder contains many important information: datasheets, project evaluation and help documents for HDL-Designer, to name just a few.

The signals of the top-level are as follows:

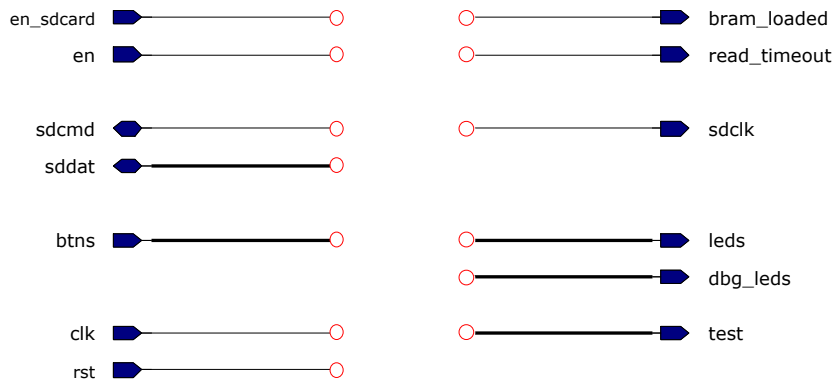



Figure 3 - Empty Toplevel signals

The available signals work according to the following groups:

- System
  - **`clk`** : system clock, clocked at **`25MHz`**
  - **`rst`** : asynchronous system reset
  - **`test[47:0]`** : vector grouping several system signals to automate tests in simulation
  - **`en_sdcard`** : signal activating the reading of the code from the SD card - **unused by default in the student version**
  - **`en`** : independent signal to enable/disable the processor - **unused by default in the student version**
- Processor
  - **`bram_loaded`** : indication that the program has been correctly loaded from the SD card, reflected on the green LED
  - **`read_timeout`** : indication that the program failed to be found/loaded from the SD card
- SD card
  - **`sdclk`**, **`sdcmd`**, **`sddat[3:0]`** : clock, command and data of the SD card
- Inputs/Outputs
  - **`btns[3:0]`** : buttons S4 to S1 of the buttons-LEDs electronic board Figure 4, directly connected to register **`x31`**
  - **`leds[31:0]`** : LED s or other outputs activated according to register **`x30`**
  - **`dbg_leds[31:0]`** : debug signals, allow to activate LED s according to internal signals. *Not connected to register **`x30`**.*
- Serial port (UART)
  - The USB-C port present on the main board can be used as a serial UART port to communicate with a PC as proposed for an advanced code Section 2.2.3.
  - **`uart_tx`** : UART transmission from the RISC-V processor, linked to register **`leds[31]`**
  - **`uart_rx`** : UART reception to the RISC-V processor, linked to register **`btns[31]`**

## 2.4 Provided blocks

All blocks necessary for the design of the microprocessor are provided in the block **heirv32_mc** and come from the following libraries:

- HEIRV32_MC
  - **`controlUnit`** : block for instruction decoding
  - **`heirv32_mc`** : top-level
  - **`instructionDataManagerSDCard`** : program memory grouping instructions and data, capable of reading and writing, reading the content of the SD card
- HEIRV32
  - **`ALU`** : a version of the ALU capable of addition, subtraction, AND, OR, and SLT
  - **`buffer*(Enable)`** : clocked buffer (flip-flops) with or without enable input
  - **`extend`** : instruction extension block for tests, supporting I, S, B and J instructions
  - **`mux3To1ULogVec`** : mux 3 to 1 of **`std_ulogic_vector`**
  - **`registerFile`** : block for managing the 32 registers, replacing **`x31`** with the vector **`btns`** - *button read register* - and **`x30`** with the vector **`leds`** - *led write register* -

The **`Board`** library contains the signal formatting logic, intended for deployment of the circuit on the FPGA .

Do not modify the names of the signals already present in the top-level. The vector **`test[47:0]`** is based on these names.

## 2.5 Indication on the daughter board

The indication LED on the daughter board is used to indicate the system status, in order:

- Blue LED : system heartbeat at 2 [Hz] (***solution version***) / 8 [Hz] (***student version***), showing that the **FPGA** design is active.
  *If the SD card is not inserted, the LED does not blink, the system being held in reset.*
- Green LED : indicates that the **program** has been correctly **loaded** from the SD card and the processor is active for code execution.
- Red LED : indicates that the **processor** is **activated**.
  *The S4 button enables/disables the processor. The S3 button enables the processor for a single clock cycle when it is disabled. These two buttons have no effect in the student version (processor always enabled).*

## 2.6 Simulator

The **`HEIRV32_test`** library contains the tester **`heirv32_mc_tb`** to simulate the processor execution.

The processor loads the file **`Simulation/code_sim_bram.txt`** into memory, which is the assembler code **`Simulation/code_sim.s`** compiled by **`heirv32-asm/HEIRV32-ASM_xxx`**.

# 3 Components

The system consists of three different hardware boards, visible in the figure Figure 1.

- An FPGA development board, see figure Figure 4.
- A control board with 4 buttons and 8 LEDs , see figure Figure 5.
- Two LEDs **P**eripheral **M**odule (PMod) boards, see figure Figure 6.

## 3.1 EBS3 FPGA board

The main board is the school's EBS 3 lab development board. It hosts a Lattice LFE5U-25F FPGA chip and has many different interfaces ( UART , PMod , PPT, Ethernet). The oscillator used produces a clock signal (**`clock`**) with a frequency of $f_{\text{clk}} = 100\text{MHz}$, internally reduced by PLL to $f_{\text{clk}} = 25\text{MHz}$.

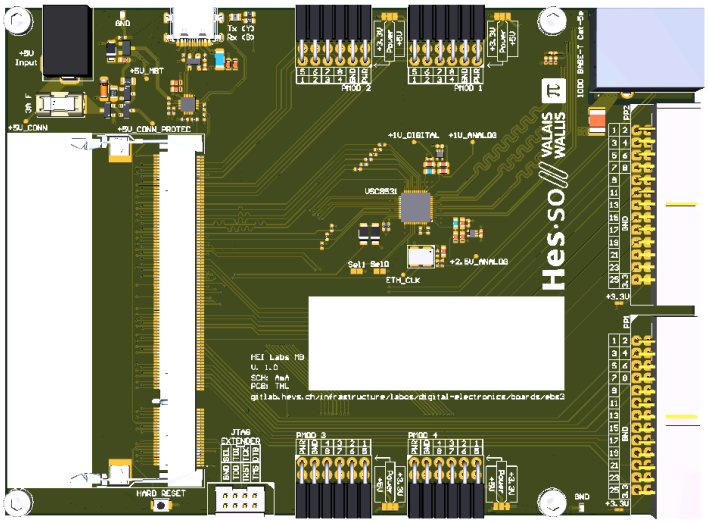



Figure 4 - FPGA board

## 3.2 Buttons and LEDs

The board with buttons and LEDs [8] is connected to the motherboard. It has 4 buttons and 8 LEDs that can be used in the design. If desired, this board can be equipped with an **L**quid **C**rystal **D**isplay (LCD) display [9], [30].

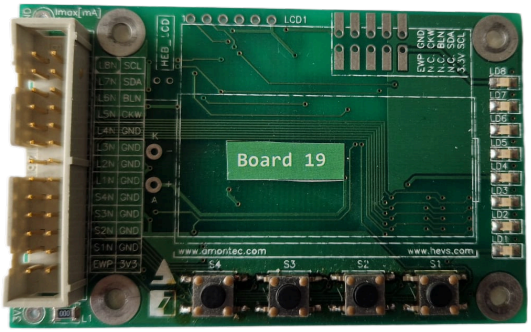



Figure 5 - Buttons and LED board

## 3.3 Additional LEDs

It is possible to extend the number of LED using dedicated extension boards. It is also possible to connect additional extension boards (distance sensor, open-drain outputs ...).

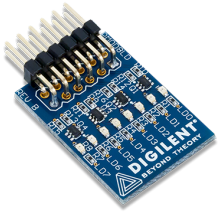


Figure 6 - LED PMod

## 3.4 Optional boards

To add functionality to your circuit, it is possible to use various extension boards. Their documentation is given under the folder **`doc/ext_boards`**.

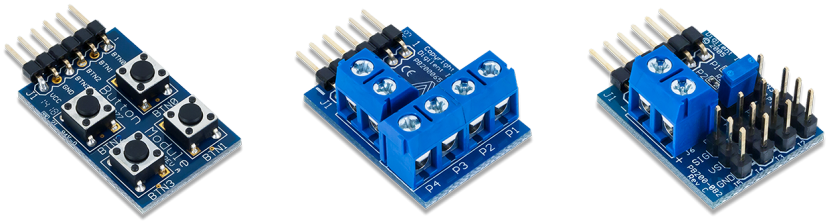


Table 1 - Inputs: PMod BTN [19], [20], PMod CON1 [21], [22], PMod CON3 [23], [24]

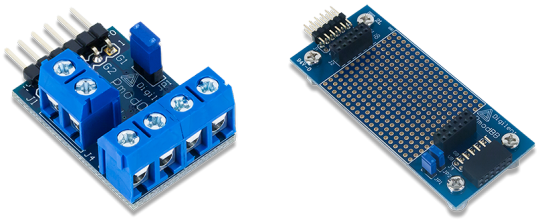


Table 2 - Outputs: PMod OD1 [26], [27], PMod BB [17], [18]

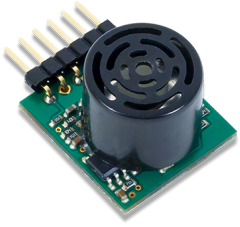


Table 3 - I/Os: PMod MAXSONAR [25], [29]

# 4 Evaluation

In the **doc/** folder, the file **evaluation-bewertung-riscv.pdf** shows the detailed evaluation scheme, table Table 4.

The final grade contains the report, the code as well as a presentation of your system.

| **Evaluated aspects** | Points |
|---|---|
| **Report** | **45** |
| Introduction | 3 |
| Specification | 5 |
| Project | 15 |
| Verification and validation | 5 |
| Integration | 9 |
| Conclusion | 3 |
| Formal aspects of the report | 5 |
| **Functionality of the circuit** | **60** |
| Minimal functions | 30 |
| Additional function 1 | 15 |
| Additional function 2 | 15 |
| **Quality of the solution** | **15** |
| **Presentation** | **30** |
| **Total** | **150** |

Table 4 - Bewertungsraster

The evaluation grid gives indications on the structure of the report. For a good report, consult the document “How to write a project report” [13].

# 5 Guide

To start the project, proceed as follows:

- Read the specifications and information carefully.
- Examine the hardware with the pre-installed program and the HDL-Designer project.
- Browse the documents in the **doc/** folder of your project.
- Analyze in detail the blocks that already exist.
- Develop a detailed functional diagram. You should be able to explain the signals and their functions.
- Implement and simulate the different blocks.
- Test the solution on the FPGA and find any errors
- Write and deploy your own code.

## 5.1 General architecture

First propose an architecture without implementing a block focusing on the multi-cycle principle:


Figure 7 - RISCV pipeline

### 5.1.1 Signal en

The **en** input allows to cut the operation of the processor.
It is held at '1' by default to be able to use the 4 buttons without stopping the system.
**Make sure to keep this signal as already wired in the given blocks!**

### 5.1.2 Instruction lw

The design is centered around the program memory by thinking about the **lw** instruction, an instruction requiring the 5 steps of the pipeline.

The **lw rd, imm(rs1)** instruction is summarized as follows:

- *Fetch*: the instruction is retrieved from memory
- *Decode*: the **rs1** register is extracted from the instruction, as well as the **imm** value which is extended to 32 bits
- *Execute*: the memory address is calculated by adding **rs1** and **imm**
- *Mem. Access*: the memory is pointed to the address calculated previously, giving the value to load into the **rd** register
- *Write Back*: the value is saved in the **rd** register

In terms of circuit, the **`lw rd, imm(rs1)`** instruction gives:

- **Fetch**: the program counter selects information from memory, generating a **`data (RD)`** and **`instruction (instr)`** bus. The buses are sequential: **`Data`** is read from memory at each clock tick, while **`Instr.`** is only updated if the **`IRWrite`** input is set to **`'1'`**.

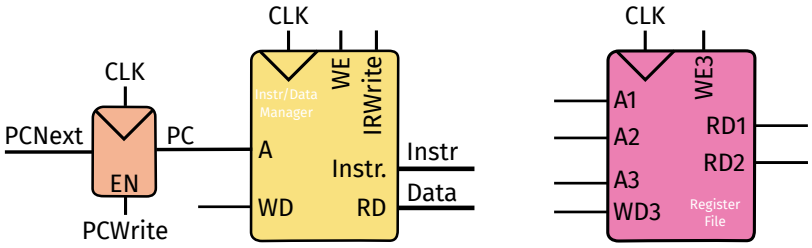



Figure 8 - Fetch

- **Decode**:
  - the base address is contained in the **`Instr.`** bus, bits 19 downto 15. They are used as an address to select the **`RD1`** register.
  - the immediate value is given in bits 31 downto 20, whose sign must be extended. To do this, the **`extend`** block allows, thanks to two control bits, to define whether the value is encoded on 12, 13 or 21 bits and thus to extend the sign of the value.
  - the flip-flops, separating this step, are hidden inside the **`register file`** block

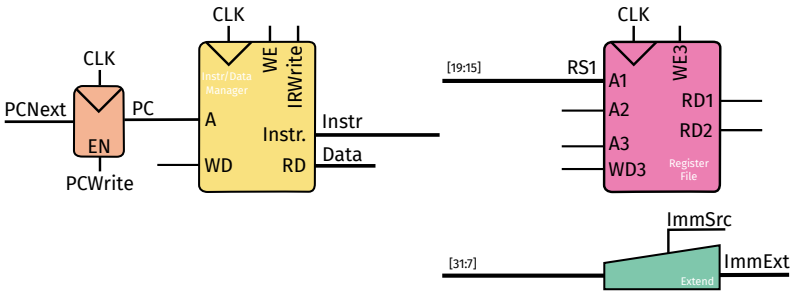



Figure 9 - Decode

- **Execute**: the immediate value is added to the register read through the ALU to determine the address to access in memory. The output flip-flop allows this step to be separated from the rest of the pipeline once again.

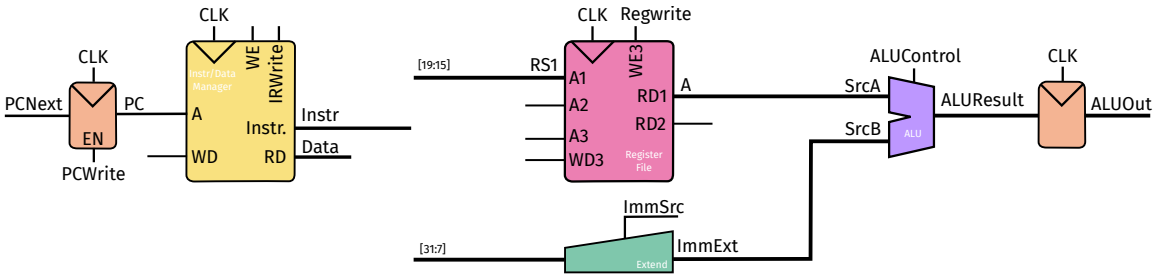



Figure 10 - Execute

- **Mem. Access**: the calculated value is used to read the memory again. For this purpose, a multiplexer is added to select between the PC or the ALU value, controlled by the **`AdrSrc`** signal.

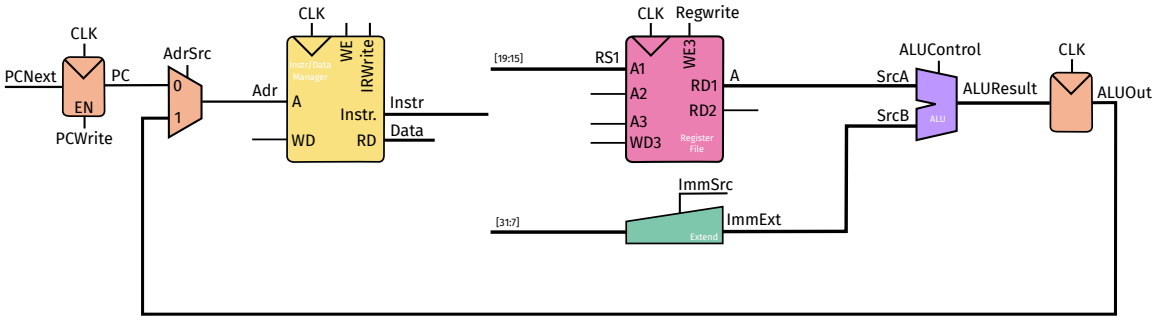



Figure 11 - Memory Access

- **Write Back**: the last step is to write the destination register specified by bits 11 downto 7 of the **`Instr.`** bus. The **`RegWrite`** signal loads the register pointed to by A3 at the next clock tick. This value can come from the ALU result as here or from the data itself. A multiplexer, controlled by the **`ResultSrc`** signal, is added:

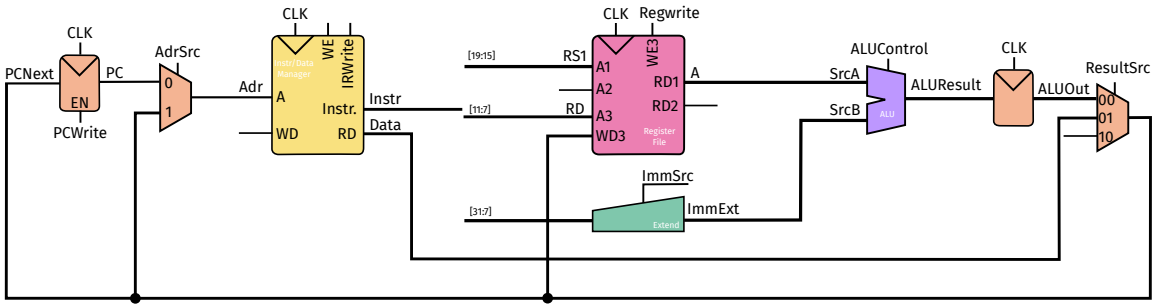



Figure 12 - Write Back

### PC + 4

In parallel to all this, the program counter must be incremented. This was done with a separate adder in the single-cycle architecture. However, here it is possible to use the ALU during the fetch step because no calculation is ongoing. To control the ALU sources, two multiplexers are added, controlled by the **`AdrSrcA`** and **`AdrSrcB`** signals. Here, the **`PC`** is loaded on A while B loads the value 4, the ALU being set to addition. Since the calculation must be used immediately (**`PC`** must be saved for later use), the ALU result flip-flop is bypassed and the signal added to the output multiplexer. The enable input controlled by **`PCWrite`** is set to **`'1'`**, allowing **`PCNext = PC + 4`** to be saved on **`PC`**:

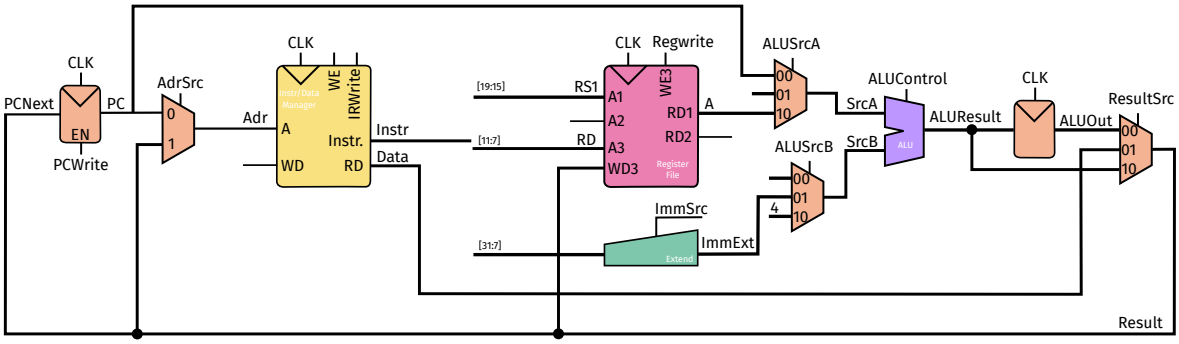



Figure 13 - PC + 4

*Control signals are generated by the block **`controlUnit`** addressed below.*

### 5.1.3 Instruction sw

In the same reasoning, extend the system to support **`sw`**:

- *Fetch*: reads the instruction from memory pointed to by **`PC`**.
- *Decode*: loads the register specifying the base address on **`RD1`**. Also, loads the register containing the information to be saved on **`RD2`**.
- *Execute*: adds the immediate value to the base address through the ALU to point to the memory address.
- *Write Back*: saves the value in memory using the **`MemWrite`** signal.

### 5.1.4 Instruction Type R - I

Then think about the paths needed for R and I type instructions:

- *Fetch*: reads the instruction from memory pointed to by **`PC`**.
- *Decode*: loads the necessary source registers.
- *Execute*: performs the operation requested by the instruction through the ALU.
- *Write Back*: saves the result in the destination register if necessary.

### 5.1.5 Instruction beq

Add the **`beq`** instruction that compares two registers and changes the **`PC`** if they are equal:

- *Fetch*: reads the instruction from memory pointed to by **`PC`**.
- *Decode*: loads the two registers to compare. Since the ALU is not used, the address in case of a jump can be calculated. However, the **`PC`** has already incremented. Therefore, it is necessary to save an old **`PC`** (**`oldPC`**) value in the previous step to load it onto the ALU source A. Source B is the immediate value.
- *Execute*: subtracts the two registers and sets the **`zero`** signal to **`'1'`** if the result is 0. In this case, the control block sets the **`PCWrite`** signal to **`'1'`**, a signal that allows the **`Result`** bus to be loaded onto **`PC`**. Thus, the jump value (ALU output from the previous step) is loaded: $a == b, \mathrm{PC} = \mathrm{PC} + \mathrm{imm}.$. Otherwise, **`PCWrite`** remains at **`'0'`** and **`PC`** is not modified: $a \neq b, \mathrm{PC} = \mathrm{PC} + 4$.

### 5.1.6 Instruction jal

Add the **`jal`** instruction that jumps after saving the return address:

- *Fetch*: reads the instruction from memory pointed to by **`PC`**.
- *Decode*: calculates the jump address **`oldPC`** + **`imm`**.
- *Execute*: saves the jump address as the new **`PC`** while calculating the return address **`oldPC + 4`** to be saved in the destination register.
- *Write Back*: saves the return address in the destination register.

### 5.1.7 Instruction jalr

Think about the necessary paths for the **`jalr`** instruction. It works similarly to **`jal`**, but instead of jumping to **`oldPC + imm`**, it jumps to the address **`rs1 + imm`**.

- *Fetch*: reads the instruction from memory pointed to by **`PC`**.
- *Decode*: gives time for **`rs1`** to be read from the register.
- *Execute1*: calculates the jump address **`rs1 + imm`**.
- *Execute2*: saves the jump address as the new **`PC`** while calculating the return address **`oldPC + 4`** to be saved in the destination register.
- *Write Back*: saves the return address in the destination register.

*It should be noted here that the execution step is divided into two sub-steps. In the case of **`JAL`**, it was possible to calculate the jump address during the decoding step because **`oldPC`** and **`imm`** are already available there on the ALU (the immediate value extension block does not require a clock tick).*
*In the case of **`JALR`**, the decoding step must be completed before the value of the **`rs1`** register can be used.*
*Several strategies are conceivable, including changes to the overall architecture for more complete processors. In our case, adding a second execution step is the simplest way to overcome this problem.*

## 5.2 Control Unit

A control block is still missing that tracks the current step according to the aforementioned pipeline and generates the various control signals. To simplify the work, separate the signal generation into three distinct blocks:

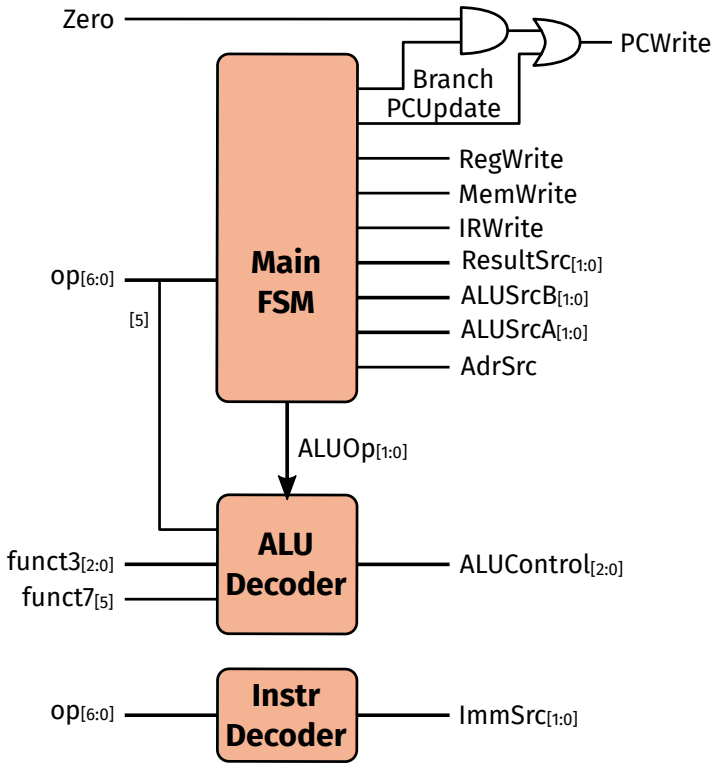



Figure 14 - Control unit

The instructions of the RV32I set are divided into 6 types: R, I, S, B, U and J. It is logical that two instructions of the same type follow the same processing. If this were not the case, each instruction would have to be treated differently by the controlUnit block. There are 40 of them just for the basic set, a set that does not even allow to run an OS!

### 5.2.1 Main FSM

The state machine manages the control signals of the processor.
It is responsible for tracking the different steps of the pipeline in order to configure each block accordingly.
Some steps of the pipeline are common to several instructions, others specific to a certain type.
*The **`ALUOp`** signal is explained under the Section 5.2.2.1 .*

### 5.2.2 ALU Decoder

The ALU performs arithmetic and logical functions according to the **`ALUControl`** signal as defined by the following table:

| **`ALU Control`** | **`Operation`** |
|---|---|
| **`000`** | *add* |
| **`001`** | *sub* |
| **`010`** | *AND* |
| **`011`** | *OR* |
| **`100`** | *XOR* |
| **`101`** | *Set Lesser Than* |
| **`110`** | *Shift Left Logical* |
| **`111`** | *Shift Right Logical* |
| Others | **`-`** |

Table 5 - Operation table for the ALU block

The **`ALU Decoder`** block must be able to control the ongoing mathematical/logical operation based on various signals.

#### 5.2.2.1 ALUOp Signal

There are pipeline steps for which the ALU is used by the FSM instead of the instruction. For example, during the **`Fetch`** step, the ALU is used to calculate the next address and must therefore be set to addition.

The **`Main FSM`** therefore generates the **`ALUOp`** signal, which is a 2-bit code that forces the **`ALU Decoder`** block to perform a specific operation:

| **operation** | $\text{ALUOp}_{[2:0]}$ |
|---|---|
| add | **`00`** |
| sub | **`01`** |
| instruction based | **`10`** |
| X | **`11`** |

Table 6 - Operation table for the ALUDecoder block

Some instructions are also related by their operation, even if they are of a different type. For example, `addi x2, x3, 30` and `add x2, x3, x4` both perform an addition operation. Instructions I and R are therefore grouped under the same code.

The following table presents this unification idea and lists the special cases for decoding:

| ALUOp | funct3 | $\text{Op}_5 \cdot \text{funct7}_5$ | **instr** | $\text{ALUControl}_{[2:0]}$ |
|---|---|---|---|---|
| **00** | **---** | **--** | *lw, sw* | **000** (add) |
| **01** | **---** | **--** | *beq* | **001** (sub) |
| **10** | **000** | **0·0**, **0·1**, **1·0** | *add / addi* | **000** (add) |
| **10** | **000** | **1·1** | *sub* | **001** (sub) |
| **10** | **001** | **--** | *sll / slli* | **110** (sll) |
| **10** | **010** | **--** | *slt / slti* | **101** (slt) |
| **10** | **100** | **--** | *xor / xori* | **100** (xor) |
| **10** | **101** | **-·0** | *srl / srli* | **111** (srl) |
| **10** | **110** | **--** | *or / ori* | **011** (or) |
| **10** | **111** | **--** | *and / andi* | **010** (and) |

Table 7 - Operation table for the ALUDecoder block

### 5.2.3 Instr. Decoder

The extend block is based on the instruction type given by **`immSrc`** to extract and extend the sign of the immediate value:

| **immSrc** | **Type** |
|---|---|
| **00** | *I* |
| **01** | *S* |
| **10** | *B* |
| **11** | *J* |
| **--** | *Others* |

Table 8 - Operation table for the Extend block

The **`Instr. Decoder`** block must therefore identify the instruction based on the operand **`op[6:0]`** and deduce the correct configuration for **`ImmSrc`**.

Complete the **`Main FSM`**, **`ALU Decoder`** and **`Instr. Decoder`** blocks to support the instructions listed under Section 2.

Be sure to check which type each instruction belongs to (R, I, S, B, U, J). The name of the instruction itself can be misleading!

# 6 Tests

## 6.1 Simulation

To simulate the complete circuit, a test bench is available under **`HEIRV32_test/heirv32_mc_tb`**. It runs the code given under **`Simulation/code_sim.s`**.

### 6.1.1 Understanding

Open and analyze the given code.

- What instructions are executed?
- Is it a good candidate to confirm the operation of your processor?

### 6.1.2 Test Automation

The tester **`HEIRV32_test/heirv32_mc_tester`** displays a **`testInfo`** signal on the simulation allowing to know theoretically which instruction should be currently executed. In case of non-functioning of the processor, this information is worthless.

Tests are automated through the procedure:

```
procedure checkProc(
  msg :           string;
  AdrArg :        unsigned(31 downto 0);
  ALUControlArg : std_ulogic_vector(2 downto 0);
  ALUSrcAArg :    std_ulogic_vector(1 downto 0);
  ALUSrcBArg :    std_ulogic_vector(1 downto 0);
  IRWriteArg :    std_ulogic;
  PCWriteArg :    std_ulogic;
  adrSrcArg :     std_ulogic;
  immSrcArg :     std_ulogic_vector(1 downto 0);
  memWriteArg :   std_ulogic;
  regwriteArg :   std_ulogic;
  resultSrcArg :  std_ulogic_vector(1 downto 0)) is
begin
  ...
end procedure checkProc;
```

It is used in the tester loop as follows:
`checkProc("Addi, addr. 0x00 - decode", x"00000004", "000", "01", "01", '0', '0', '0', "00", '0', '0', "00");`

It simply allows to check the current state of the processor against the expected state. In case of an error, the test stops.

Your implementation may differ slightly from the given solution depending on your implementation choices. It is up to you to manually judge and correct the test code if necessary.

Confirm that your circuit is working.

## 6.2 Code

Once the architecture has been validated by simulation, it is possible to flash the FPGA .

For this, the library **`Board`** contains the top-level. The flashed code is the one under **`Simulation/code.bin`**.

The given code **`Simulation/code.s`** is empty, unlike **`Simulation/code.bin`**. It is up to you to write a new code once the circuit operation is validated.

### 6.2.1 Equivalence Control

First, test your system through the given solution:

- Unplug the Developement board FPGA .
- Load the code onto the micro SD card by copying the **`Simulation/code.bin`** file while keeping the same name. A copy of the code is also available on the SD card - *do not delete it from the card.*
- Replug the board.
- Plug in the power supply via USB-C cable.
- Check the code loading, button operation, LED lighting, general system behavior.

Then flash your processor and repeat the tests. Make sure everything works correctly.

Refer to the document **`doc/Board_LFE5U-25F.pdf`** for flashing the FPGA .

First flash the FPGA via JTAG (temporarily). This allows you to return to the solution version by unplugging and replugging the power supply.
Once everything is validated, program the configuration into flash to keep your processor permanently.

Confirm that your processor is working on the FPGA.

### 6.2.2 Custom Code

When the circuit is functional, write your own code in a **`.s`** file, capable of reacting to button presses and lighting up LED :

- Writing a value to register **`x30`** allows you to light up the LED . LED 0 corresponds to bit 0, LED 1 to bit 1 ...
- Reading register **`x30`** gives the current state of the LED .
- Reading the buttons is done by reading register **`x31`**. Bit 0 corresponds to button **`S0`**, bit 1 to button **`S1`**.
- Writing to register **`x31`** does not modify it.

Test your code using the RISC-V online interpreter and Ripes before flashing it.

Write and test your code on a simulator.

Once the code is validated, compile it using the **`heirv32-asm/HEIRV32-ASM_xxx`** software to generate a new **`code.bin`** file.

To load the code, copy the generated **`xxxx.bin`** file to the micro SD card and rename it to **`code.bin`**.

Unplug the Developement board, insert the micro SD card, then plug the board back in.

Confirm that your code is working.


## 6.3 Tips

Here are some additional tips to avoid problems and time loss:

- Divide the problem into different blocks: use the empty Toplevel document for this. It is recommended to have a balanced mix between the number of components and the size/complexity of these components.
- Analyze the different input and output signals, their types, their sizes ... It is recommended to use the data sheets.
- Respect the DiD chapter "Methodology for the development of digital circuits (MET)" when creating the system. [10]
- Follow the incremental procedure proposed. Use tests as soon as possible.
- Save and document your intermediate steps. Architectures that have not worked, basic codes to test the architecture ... are all material that can be added to the report.

Don't forget to have fun.


# Glossary

***FPGA*** – **Field Programmable Gate Array** 2, 3, 5, 8, 10, 13, 22

***LCD*** – **Lquid Crystal Display** 10

***LED*** – **Light Emitting Diode** 2, 5, 6, 8, 9, 10, 11, 22, 23

***PMod*** – **Peripheral Module** 10, 11

***PWM*** – **Pulse Width Modulation** 6

***RISCV*** – **5-stages Reduced Instruction Set Computer architecture, open-sourced** 3, 4, 5, 13

***UART*** – **Universal Asynchronous Receiver Transmitter** 6, 10


# Bibliography


[1] A. Waterman, K. Asanovic, and F. Embeddev, "RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA." Accessed: Jun. 04, 2022. [Online]. Available: https://www.five-embeddev.com//riscv-isa-manual/latest/riscv-spec.html

[2] F. Embeddev, "RISC-V Quick Reference." Accessed: Jun. 04, 2022. [Online]. Available: https://www.five-embeddev.com//quickref/tools.html

[3] S. L. Harris and D. M. Harris, "Digital Design and Computer Architecture RISC-V Edition," *Digital Design and Computer Architecture*. Elsevier, pp. IBC1–IBC2, 2022. doi: 10.1016/B978-0-12-820064-3.00025-8.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - RISC-V Edition*, Second Edition. Elsevier, 2021.

[5] Xilinx, "Spartan-3 FPGA Family." Accessed: Nov. 20, 2021. [Online]. Available: 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, and Christophe Bianchi, "Methodologie Für Die Entwicklung von Digitalen Schaltungen." 2021.

[11] François Corthay, Silvan Zahno, and Christophe Bianchi, "Méthodologie de Conception de Cicuits Numériques." 2021.

[12] Christophe Bianchi, François Corthay, and Silvan Zahno, "Wie Verfasst Man Einen Projektbericht?." 2021.

[13] Christophe Bianchi, François Corthay, and 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.