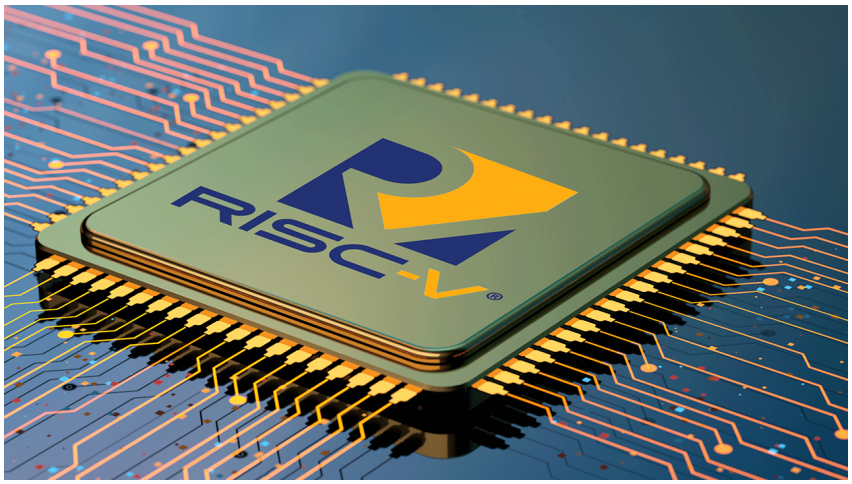
METU EE446
Computer Architecture
Laboratory



Laboratory Project - Single Cycle RISC-V Processor

Objectives

This project aims to explore a novel, license-free, open-source instruction set architecture that is becoming increasingly popular in the industry. You will construct the datapath and the control unit of a single-cycle 32-bit RISC-V processor. For this project, the instruction set has been extended by one more instruction. The designed processor will be able to execute all instructions in the **extended** instruction set. Finally, you will embed your design into the FPGA of the Nexys A7 board and demonstrate your design.

This project will be done in groups of 2 students unless you choose to do the project by yourself. You can choose your partner. Each partner is expected to contribute in equal amounts. If the work is divided too unevenly, the student who did more will be generously graded, while the student who did less will be penalized. The most uneven work division acceptable is 60-40. If you wish to do the project in a group but cannot find a partner, we will match you with another student (if possible).

The project needs to be done by groups individually. **In other words, inter-group cooperation will be considered as cheating and further action will be taken as explained in the EE446 Laboratory Manual.**

Contents

1 Introduction

RISC-V is an innovative instruction-set architecture (ISA) that was initially developed to support research and education in computer architecture at UC Berkeley. Its design aspirations have since expanded, aiming to become a standard, free, and open architecture for industry implementations. Characterized by its flexibility and generality, RISC-V is not tailored to specific microarchitectural styles or technologies, making it suitable for a wide range of hardware implementations, from custom chips to multicore processors. The architecture includes a modular structure with a base integer ISA and optional extensions supporting 32-bit and 64-bit address spaces. RISC-V is designed to facilitate efficient implementations and is fully virtualizable, simplifying hypervisor development. This open and versatile ISA holds significant potential to influence academic research and industrial applications broadly.

1.1 Reading Assignment

In this project, a part of RISC-V ISA will be implemented. To get familiar with it, read the RISC-V specification given in the link below. Chapters 2, 24 and 25 are of most interest. Note that this project manual doesn't contain the low-level details needed to implement the processor fully. Therefore, you'll need to seek additional information from relevant parts of the RISC-V specification.

RISC-V Specification 20240411.pdf

Additionally, Chapter 7.3 of Harris & Harris Risc-v book contains an implementation of part of the ISA, which you can consider:

Sarah Harris, David Harris - Digital Design and Computer Architecture RISC-V Edition (2021)

Since RISC-V is an open standard, many materials are freely available online.

As usual, taking any code without citing its origin or taking large sections of code from any source is Plagiarism and will result in a `0x0` (zero) grade, whereas disciplinary action may be taken.

1.2 Comparison with ARM

ARM 32-bit had 16 architectural registers, with PC=R15. On the other hand, RISC-V has 32 registers (x0 to x31), and PC is not one of these registers. Also, register0 is hardwired to zero value.

ARM had conditional instructions, but RISC-V only has conditional branches. Therefore, ALU flags don't need to be saved to a register. RISC-V does not have shifted operands, but it has shift instructions.

RISC-V has six types of instructions as seen in Figure 1. Instructions can be read from the rs1/rs2 registers and written to the rd register. Instruction encodings may look complicated, especially the immediate encodings. However, these encodings reduce the number of multiplexers and are efficient to implement in hardware.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Figure 1: RISC-V instruction types

2 Project Preliminary Work (20%)

2.1 ISA to be implemented

RISC-V consists of base ISA and extensions. In this project, the Unprivileged RV32I Base Integer Instruction Set will be implemented. Additionally, one new instruction will be added as an extension.

FENCE, ECALL, EBREAK, and HINT instructions will not be implemented, as they are irrelevant.

The list of instructions to be implemented is given in Table 1. The parts written in square brackets represent distinct instructions, all of which are required. For example, SLT[I][U] expands to SLT, SLTI, SLTU, and SLTIU. For more information about the instructions, refer to the RISC-V specification.

Arithmetic instructions:	ADD[I], SUB
Logic instructions:	AND[I], OR[I], XOR[I]
Shift instructions:	SLL[I], SRL[I], SRA[I]
Set if less than:	SLT[I][U]
Conditional branch:	BEQ, BNE, BLT[U], BGE[U]
Unconditional jump:	JAL, JALR (Return-address stack push/pop functionality will not be implemented)
Load:	LW, LH[U], LB[U]
Store:	SW, SH, SB
Others:	LUI, AUIPC

Table 1: List of Instructions

2.2 Datapath (25%)

2.2.1 Datapath Design

Design a datapath that will support all of the instructions listed in Table 1. In your report, explain your design with appropriate visuals. It can be based on the datapath shown in Figure 3.

2.2.2 Datapath Implementation

Implement your design in Verilog HDL. Datapath must consist of submodules but should not contain "always blocks" or direct logic. You can use the modules from laboratories or create your own.
Datapath needs to have a synchronous reset.
Show the synthesized RTL view of your datapath in your report. Explain how each instruction type is executed in your datapath.

2.3 Controller (25%)

2.3.1 Controller Design

Design a controller that matches your datapath. In your report, explain your controller with appropriate visuals. Explain any submodules the controller has.

2.3.2 Controller Implementation

Implement your design in Verilog HDL. The controller can be written in a single module, or it can have submodules.
Show the synthesized RTL view of your controller in your report. Explain how each instruction type is executed in your controller. Explain the control signals related with the UART peripheral.

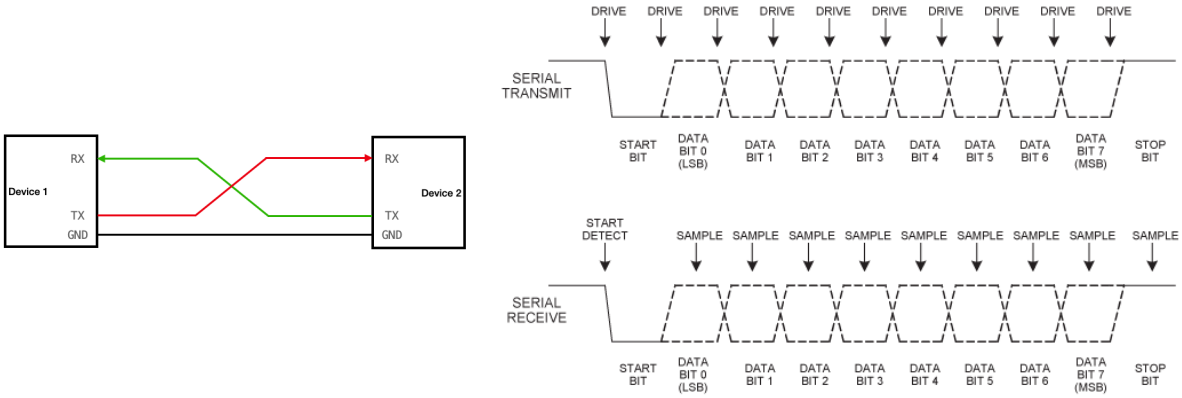
2.4 UART Peripheral (10%)

Many microcontrollers have various peripherals that expand the CPU's functionalities. In this project, you will also design a Universal Asynchronous Receive Transmit (UART) Peripheral that transmits and

receives bytes. You will use 9600 as your constant baudrate, and 8-N-1 as your protocol setting. You will use memory store/load instructions to interact with the peripheral, for sending and receiving a byte, respectively.

2.4.1 UART Protocol

UART is a serial communication protocol that's commonly used in microcontrollers. It is a full duplex protocol, meaning that both devices can send and receive data at the same time. It is also an asynchronous protocol, meaning that there are no clock signals, and both transceivers should agree on a baudrate (bit/s) at which the data is transferred, beforehand. Ignoring the hardware flow control signals, which are optional and will not be used in this project, the communication is established with only two signal lines called Rx and Tx. Tx is used for transmitting data, and Rx is used for receiving data. By that logic, two devices communicate by connecting one's Tx signal to the other's Rx pin, and vice versa. An example connection and signaling diagram can be seen in Figure 2. You can find the necessary information for implementing the UART protocol on the Internet.



(a) UART Connection

(b) UART Signaling

Figure 2: UART Protocol

2.4.2 Memory Mapped Interface

The processor we are implementing has 2 separate 32-bit address spaces for instructions and data. In practice, the data address space is rarely used at its full range by the data memory. For example, a 64-bit processor can address 16 EB (18,446,744,073,709,551,616 bytes) of memory, but actual usable memory is much smaller. A memory-mapped interface leverages this fact and assigns the unused memory locations to interact with different peripherals. When you read or write to a memory address that belongs to a peripheral, you transfer data to/from that peripheral.

The UART peripheral should use 2 memory addresses for the write and read operations. The specifications for the peripheral are provided below.

Write Operation A write access to address `0x00000400` is interpreted as a transmit request. The written byte is placed into the UART transmitter for serial transmission. Software should use the `SB` (Store Byte) instruction to write to this address. A write must only be performed when the transmitter is not busy; otherwise, writing a new byte during an ongoing transmission results in undefined behavior.

Read Operation The UART receiver continuously listens for incoming bytes. A read access to address `0x00000404` returns the newly received byte from the UART receiver. If no new data is available at the time of the read, the peripheral returns `0xFFFFFFFF`. The `LW` (Load Word) instruction can be used to perform the read. If a new byte is received when there is no available space in the internal buffer, the resulting behavior is undefined.

2.4.3 UART Peripheral Design

Design a UART peripheral that will be placed inside your datapath. In your report, explain your peripheral with appropriate visuals. Explain any submodules the peripheral has.

2.4.4 UART Peripheral Implementation

Implement your design in Verilog HDL. The UART peripheral can be written in a single module, or it can have submodules.
Show the synthesized RTL view of your peripheral in your report. Explain the state machines of both transmitter and receiver.

2.4.5 FIFO Receive Buffer

You need to implement a 16-byte first-in, first-out (FIFO) buffer for the received bytes. This ensures that consecutively received bytes are not lost even if the program does not read the first received byte in time. Partial credit may still be awarded for the receiver during the demonstration without FIFO buffer.

2.5 Top Level

In this part, the controller and datapath are assembled in a top module. The top module needs to have a synchronous reset.

For debugging purposes, connect five switches to the register files debug port select. Connect debug register output and PC register output to 7-segment displays as in previous labs, using the provided MSSD module. Also, connect your UART peripheral's Rx and Tx signals to the board's USB UART pins UART_TXD_IN and UART_RXD_OUT, respectively. (Available in the Master XDC file.)

2.6 Testbench (40%)

It is required to verify your Computer's operation by writing a testbench.

- The testbench should read the instructions from a hex file like the HDL computer.
- It should execute all the instructions by itself and then compare its register file and PC values to the HDL design.
- The testbench should be able to execute arbitrary RISC-V code consisting of the given RV32I instructions.
- A proper testbench is automated, so it should indicate when something fails in the design without needing any manual work.

Your testbench should be very similar in form to the supplied testbenches on ODTUClass. You can use the single-cycle ARM computer testbench that was provided and modify it appropriately. For debugging purposes, you can use prints/logs in your testbench.

Explain how you implemented your testbench and its details in your report. You'll need to write RISC-V program(s) that cover all instructions with their special cases. Explain your programs and how they cover all instructions and special cases.

You don't need to test UART operation in testbench. But you need to check if you read the value `0xFFFFFFFF` when you load from the address `0x00000404`, since no byte would be received.

2.7 Important Considerations

Your codes should be clean, easily readable, and understandable. Check your indentations so that they are obvious where a block ends. Put comments when and where needed.

Since you have almost graduated, we expect a more professional report for the project compared to some of your laboratory reports. Your report should be proofread and structured well, and the texts in it should not be pictures of handwriting.

If you used some online/offline tools to compile or assemble RISC-V codes, specify them in your report.

You can lose grades if your code/report is unprofessional, hard to read, etc.

Deliverables:

- PDF Report
- Python testbench codes
- HDL codes

3 Project Demonstration (80%)

In the demonstration, you will be given a code segment for your computer; you will first use the code in your testbench and demonstrate it to your teachers. Then, load your processor designed in Part 2 to the Nexys A7 board with the given instructions.

Additionally, the teachers will ask questions about the parts each student has contributed. You should be very familiar with your work to be able to answer any questions. If a student cannot answer the questions to the satisfaction of the teachers, you will lose grades.

You should also bring your own test codes in case the computer cannot execute the given instructions. Any part that cannot be demonstrated with either Cocotb or FPGA will not receive any points.

3.1 Testing of UART at Project Demonstration

The lab computers have a program called Tera Term installed for communicating with serial ports. This program will be used for testing the UART peripheral of your design.

A Useful Materials

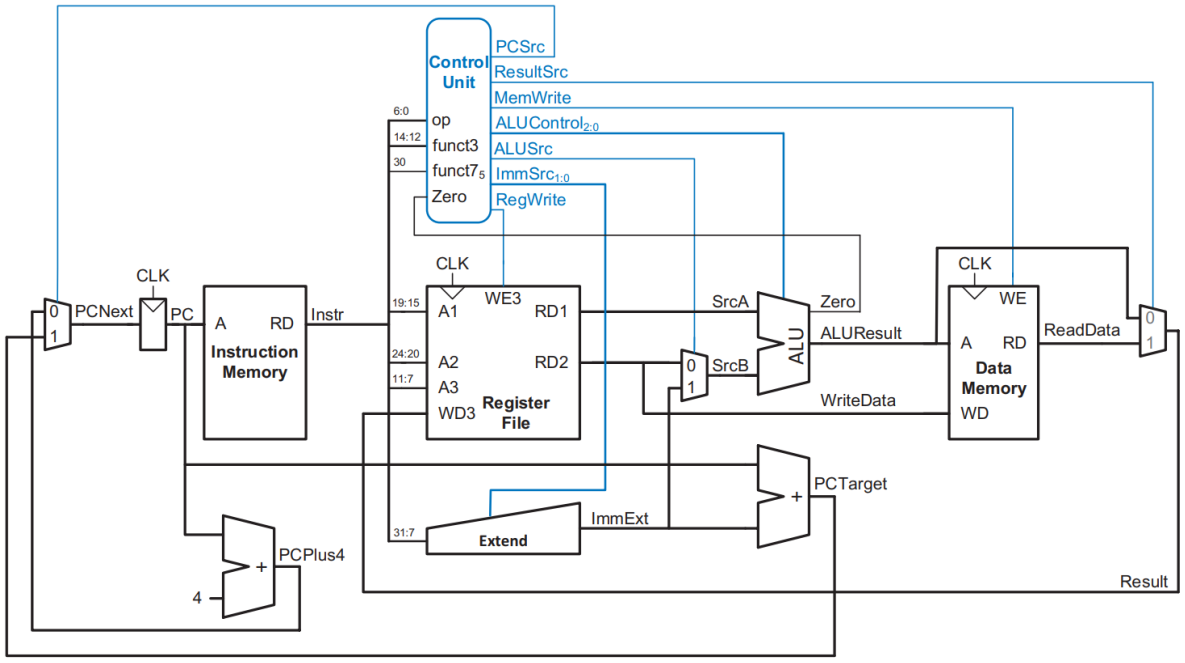


Figure 3: RISC-V Single Cycle Computer as Described in Harris&Harris