Middle East Technical University
Electrical & Electronics Engineering Department

EE446 Computer Architecture II - RISC-V Processor with UART Peripheral

Term Project Report

Necati Teoman Bahar 2515583
Uğur Eroğlu 2516151

May 25, 2025

Necati Teoman Bahar 2515583
Uğur Eroğlu 2515161

Introduction

In this report, the development and verification process of a Single-Cycle RISC-V Processor is documented. The processor is capable of handling R, I, B and J-Type instructions as well as 2 U-Type instructions. Final project is equipped with a UART Peripheral, which is also designed and tested within this project. A fully automated and user-friendly testbench is deployed to assert the functionality of overall system at the end of the project.

Datapath Design

The datapath design is started by implementing the basic datapath given in our lecture book [1]. Proper functionalities of each module is determined and the architectural modules and registers are implemented one-by-one. Paths for R-type instructions are already present as well as I-type and B-type. However, some paths needed to be added to the submodules to achieve R, I and B type instructions completely. Finalized datapath can be seen in Fig. 1 below. In the upcoming sections, the additions made to submodules and new paths will be described.

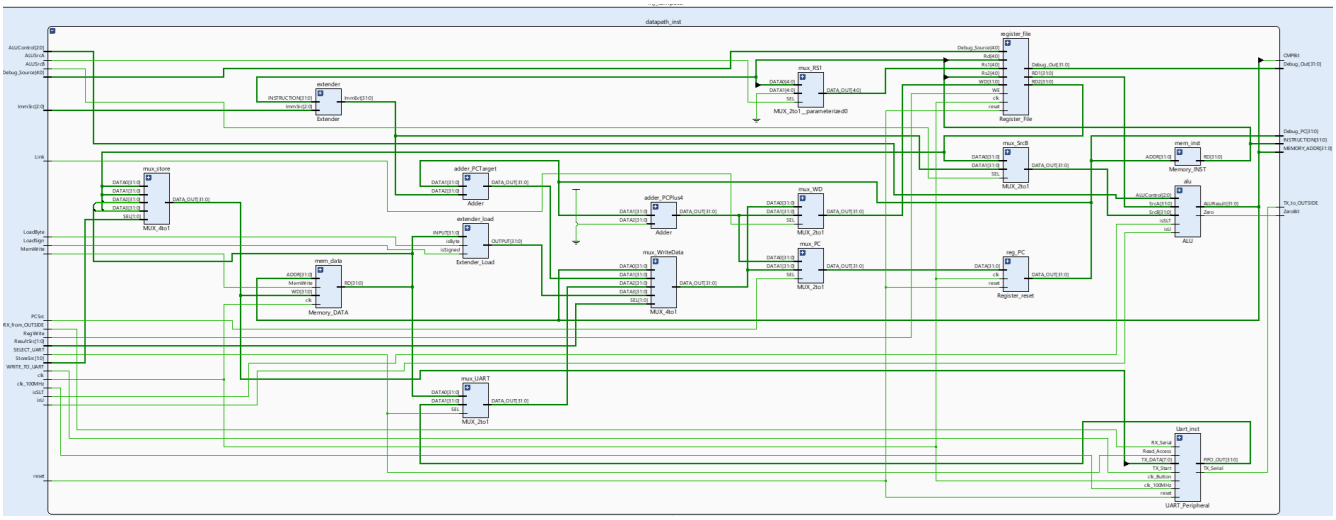

Figure 1: RTL view of the Finalized Datapath

SHIFT Instructions

Apart from the ARM Architecture we have been implementing throughout the semester, RISC-V architecture doesn't need a barrel shifter since it doesn't include shifted immediate. However, SHIFT Instructions are still an essential part of the architecture. To achieve the shifting operation, a shifter unit is implemented within the ALU. ALU is composed of 2 submodules which can be combined and controlled by 4-bit control signal however our approach is to separate the SLT[I][U] Instructions from the Arithmetic, Logic and Shift Instructions and select the proper output from the Instructions. This approach allowed us to nearly directly embed the funct3 part of the instruction into our ALU for ASLU(Arithmetic-Shift-Logic-Unit) submodule.

STORE Instructions

A directly connected data cable is not sufficient to fully cover all store instructions within our instruction set. Bit and Halfword selections needed still a 32-bit input. To not disturb the unnecessary bits, a 4-to-1 MUX placed. 3 of its inputs are configured to concate the needed bits with the data written in the selected address. With this the remaining bits stay same while only a byte or a halfword part of the address is changed.

LOAD Instructions

Load Instructions have their own unique extension types which need to be accommodated within the datapath. An additional extender module is implemented to achieve the operation. The extender is placed between the MUX responsible for the written data selection of register file and the output of the data memory. The extended data from the memory can be selected before it is written in the register with the help of this configuration.

JUMP Instructions

Even though JUMO instructions are similar to the B-type instructions, they have an extra function which is writing data to the register file. The data written in the registers is same, *PC + 4*, so a single 2-to-1 MUX at the input of the register file to select between the *PC + 4* and the other result from the ALU or Memory is sufficient. But for JALR instruction, operation of the ALU is needed. JAL instruction is easily implemented with already present extender and PCTarget signal, but the address of the JALR is obtained from the ALU and then given to the MUXs present at the input of the PC register. Proper selection of all data is done with the help of the control signals generated within the controller.

With the additions, the system is now able to execute every instruction. R-Type instructions are executed by selecting the operands of the ALU to be read data of the register file and selecting the data that will be written to the register file as ALUResult. I-Type's are similar, however second operand is not the register value but the immediate encoded in the instruction after properly extended. Load instructions, which are also I-Type, are also follows the same pattern, but instead of ALUResult as the final data, ALUResult is used as an address for the data memory and the accessed data is extended according to the instruction then loaded to the register. JALR is the exception in this case. To execute JALR, *PC + 4* is given to the register file input data and ALUResult is given to the PC Register. S-Type instructions are executed like load. An address is created with *rs1* and encoded immediate, then *rs2* is placed to the address. B-Type instructions are executed with pseudo-flags which will be decribed in the later sections, and the calculated branch address placed into the PC register. And lastly, U-Type instructions use the PCTarget adder with a MUX to select whether PC will be added. Then the result is placed in the PC register and similar to B-Type.

Controller Design

The controller decodes the 32-bit RISC-V instruction and generates the correct control signals so that the decoded instruction can be properly executed. Supported RISC-V instructions are laid out in table **??** below. Related instruction formats are also provided in the figure 2 below. In the following sections, we will detail the control signals one by one and discuss when they are set/reset and why.
The RTL view of the finalized controller design can be seen in Figure 3.

Controller Inputs

- **INSTRUCTION:** 32-bit instruction.
- **ZeroBit:** 1-bit Zero flag output of the ALU. Note that it is note staged, and all flags are combinational.
- **CMPBit:** Least Significant bit of the ALU Results. It is used for compare operations. If the ALU result is used to represent comparisons of 2 values, ALU outputs 0x00000001 or 0x00000000. The CMPBit is used to capture the result of the comparison. Note that, CMPBit is set if Less Than condition holds. This will be used in the control signal generation.
- **MEMORY_ADDR:** 32-bit Memory Address. It is used in the generation of UART control signals.

Controller Outputs

We give the detailed explanations of control signals below. Please note that, in the controller module, we decided to decode the instruction in the following order:

1. Decode the Opcode
2. Decode funct3
3. Decode funct7

so that the Controller module is reader-friendly and easy to debug. Furthermore, the Controller module is well commented and each control signal assignment contains comments regarding the effect of the control signal assignment. Thus, we will spare the reader and will not detail bit-wise assignment of controls signals, but provide explanations for the effect of control signals and how and when are they generated.

Please note that all control signals are decoded in the given order, even if it is not needed. For example, Since only store type instructions Set the MemWrite control signal, StoreSrc could be directly decoded from funct3, even if the instruction is not store type because it won't have an effect on the other instruction executions. While in the Controller module, for other types, StoreSrc is chosen as a default value and only explicitly decoded under S-Type case, we will explain the intended decoding od the StoreSrc, and similar instructions, in the following pages; since, it would be hard to follow and unnecessary to follow the exact decoding of the HDL codes, as they are well commented but 500 lines long!

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 2: Supported RISC-V instruction formats

Table 1: Supported Instructions

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

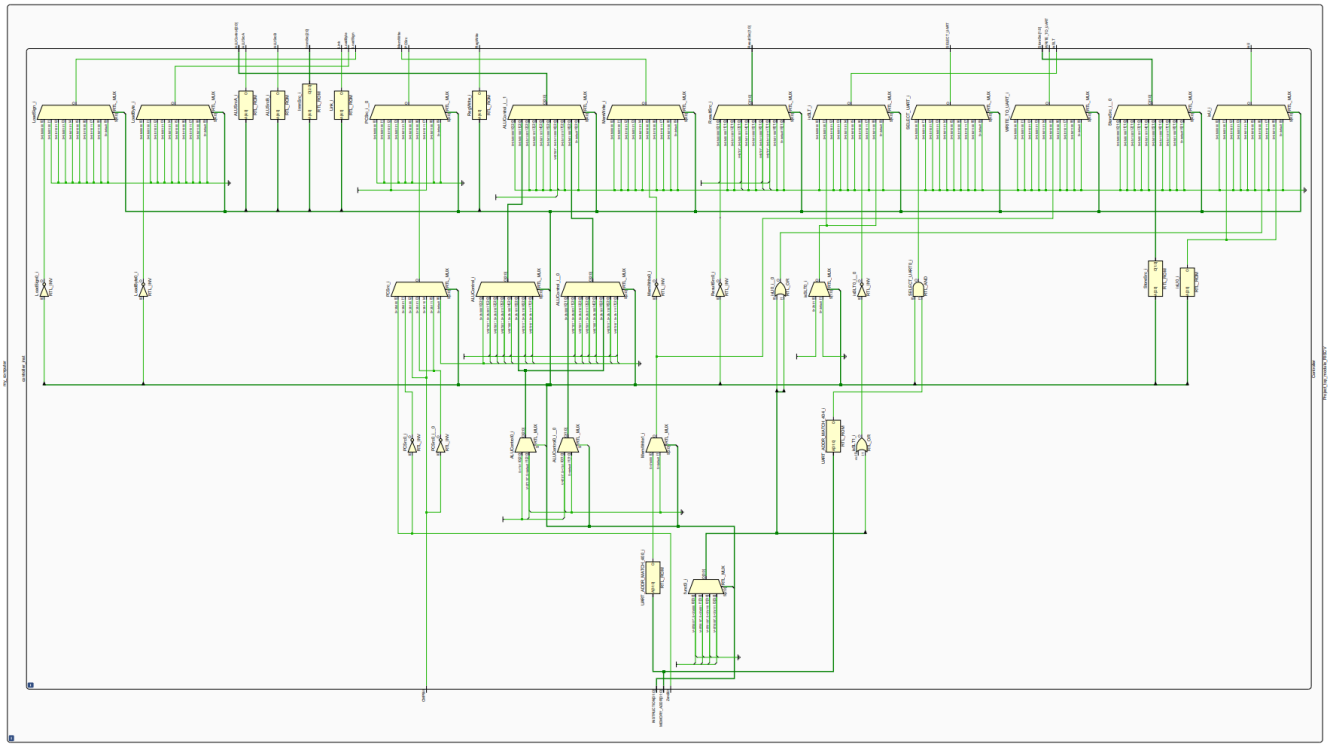

Figure 3: RTL view of the Controller

- **ImmSrc:**
  - **Width:** 3-bit.
  - **Explanation:** It is the control signal of the immediate extender module. Selects how the immediate value should be formed and extended. It has different values of I,S,B,J and U type instructions. R-Type instructions do not use the immediate values, thus ImmSrc is not important for R-Type.
  - **Generation:** Decoded from Opcode. Since each instruction has its own immediate type, decoding the Instruction Type is enough the decide the ImmSrc.
- **ALUSrcA:**
  - **Width:** 1-bit.
  - **Explanation:** Selects the input of the First Read Address of the Register file. When SET, selects zero (for Register Zero) as address. Otherwise, it chooses rs1 address of instruction. It's only use is to move the immediate value of **lui** instruction through the ALU, using addition with zero; hence, the register zero.
  - **Generation:** Decoded from Opcode. SET only for U-Type instructions. Note that auipc does not use ALU, so it is fine to SET ALUSrcA for both U-type instructions.
- **ALUSrcB:**
  - **Width:** 1-bit.
  - **Explanation:** Selects the SrcB of ALU. When SET, selects extended immediate. Otherwise, it chooses rs2 value.

  - **Generation:** Decoded from Opcode. chosen as LOW only for B-Type and R-Type instructions since they use both register values as inputs to ALU.

- **ALUControl:**
  - **Width:** 3-bit.
  - **Explanation:** Controls the ALU operation. Note that there all three shift operations are embedded in the ALU.
  - **Generation:** Decoded from Opcode and funct3 and possibly funct7. For B-Type, select operations is subtraction (to perform comparisons by setting setting ZeroBit and CMPBit inputs). For R/I Types with direct use for ALU(such as Add, Sll, ori, slt), it is further decoded from funct3 and funct7. For Load instructions, and all other instructions, it is operation is chosen as addition. Note that for all SLT[I][U] instructions, subtraction operation is chosen to perform comparison, i.e. less than.

- **StoreSrc:**
  - **Width:** 2-bit.
  - **Explanation:** Selects the memory write data input. It choses the lower byte, 2 bytes or entire eord of data that is to be loaded to memory. It fills the rest of the word with memory data if needed, as explained in the Store Instruction subsection.
  - **Generation:** Decoded from funct3.

- **LoadByte & LoadSign:**
  - **Width:** 1-bit each.
  - **Explanation:** Selects the masking of the memory read data output in Load instructions. LoadByte choses lower 8-bits when High, 16-bits when Low. LoadSign masks the remaining upper bits of the word with the sign-bit (most significant bit of the chosen lower part) when High, or with all zeros when Low. Note that it produces 4 different results in the forms of Sign/Zero-Extended Byte/Half Word. Loading the entire word is select in the ResultSrc.
  - **Generation:** Decoded from funct3.

- **ResultSrc:**
  - **Width:** 2-bit.
  - **Explanation:** Selects data forwared to Result line. For Load Word instruction, memory read data is selected. For other loads, masked memory read data selected. For B-Type, J-Type and AUIPC, Branch Target Address selected. Otherwise, ALU result selected.
  - **Generation:** Decoded from opcode and funct3.

- **RegWrite:**
  - **Width:** 1-bit.
  - **Explanation:** Enables Register Write. Low when S-Type or B-Type, High otherwise.
  - **Generation:** Decoded from opcode.

- **MemWrite:**
  - **Width:** 1-bit.
  - **Explanation:** Enables Memory Write. High for S-Type, unless instruction is store byte with store address 0x400 (UART Transmit address).

  - **Generation:** Decoded from opcode, funct3 and MEMORY_ADDR. For Store-Byte with MEMORY_ADDR == 0x400, it is forced to LOW.

- **WRITE_TO_UART:**
  - **Width:** 1-bit.
  - **Explanation:** Enables UART transmit. High for Store Byte with store address 0x400 (UART Transmit address). Low otherwise.
  - **Generation:** Decoded from opcode, funct3 and MEMORY_ADDR. High only for Store-Byte with MEMORY_ADDR == 0x400.

- **SELECT_UART:**
  - **Width:** 1-bit.
  - **Explanation:** Enables loading from UART FIFO. High for Load Word with Read address 0x404 (UART FIFO address). Low otherwise.
  - **Generation:** Decoded from opcode, funct3 and MEMORY_ADDR. High only for Load Word with MEMORY_ADDR == 0x404.

- **PCSrc:**
  - **Width:** 1-bit.
  - **Explanation:** Selects next PC source. If low, PC+4 is chosen. Otherwise, Result line is directed to PC register input.
  - **Generation:** Decoded from opcode and potentially funct3. High for jal and jalr instructions, low for non-branch instructions. For Branch instructions, the CMPBit and ZeroBit inputs are used to decide the PCSrc. Exact decoding is provided in the figure 4.

- **Link:**
  - **Width:** 1-bit.
  - **Explanation:** Selects write data of Register File. If low, Result line is chosen. If High, PC+4 is chosen as write data of Register File. It is used to store the PC+4 address in link register.
  - **Generation:** Decoded from opcode. As the name suggests, it is High only for jal and jalr instructions.

- **isSLT:**
  - **Width:** 1-bit.
  - **Explanation:** Control input for ALU. When High, ALU results is chosen as the result of less than comparison. Used for SLT[I][U] instructions and branch-type instructions.
  - **Generation:** Decoded from opcode and funct3. High for SLT[I][U] and Btype instructions.

- **isU:**
  - **Width:** 1-bit.
  - **Explanation:** Control input for ALU. When High, ALU treats its input as unsigned numbers. Note that ALU uses isU to decide to use LT flag calculation or LTU flag calculation.
  - **Generation:** Decoded from opcode and funct3. High for SLT[I]u and Btype instructions with unsigned comparisons.

```
case (funct3)
    BEQ : PCSrc =  ZeroBit; // rs1 == rs2
    BNE : PCSrc = ~ZeroBit; // rs1 != rs2
    BLT : PCSrc =   CMPBit; // rs1 <  rs2
    BGE : PCSrc =  ~CMPBit; // rs1 >= rs2
    BLTU: PCSrc =   CMPBit; // rs1 <  rs2
    BGEU: PCSrc =  ~CMPBit; // rs1 >= rs2
    default: PCSrc = LOW;   // UNDEFINED
endcase
```

Figure 4: Decoding of PCSrc for Branch Type instructions.

UART

UART is a commonly used communication protocol. This protocol consists of a start bit, a stop bit and predetermined number of bits as the data. The receiving end, RX, listens constantly to the transmitter of another module which constantly transmits HIGH bit as long as it is IDLE. When RX receives a LOW bit, it interprets it as start bit and starts loading the data as it comes. At the end when it verifies the data ends with a stop bit, outputs a 8-bit data. In this project we have implemented a basic UART Peripheral that is integrated to the data memory within the datapath with SB and LW instructions.

RX and TX both works as a FSM to achieve the mentioned functionality. They both have 4 states that is represented with 2-bit; IDLE, START_BIT, DATA, STOP_BIT. IDLE state is the state in which RX is waiting for a *start bit* to come and TX is constantly transmitting HIGH bit.

After TX is given the start signal by another module, TX enters the START_BIT stage and creates a 10-bit data by concatenating the 8-bit data with proper start and stop bit. After creation, TX transmits a single LOW bit and enters the DATA state. A counter counts the amount of state clocks passed in the DATA state. At state clock, LSB of the data is transmitted and the data register is shifted to right 1-bit. When the counter hits 7, which means 7 cycle have been passed and TX will transmit the last DATA bit, FSM enters STOP_BIT state. In this state another single HIGH bit is sent and state returns back to the IDLE. During all of these operations a ***TX_BUSY*** signal is created and assigned HIGH. As long as the busy signal is high, the system cannot be interrupted.

On the other hand, RX enters START_BIT state after it receives a LOW bit at any time. In this state, the counter that is responsible for baud rate constraint is assigned half of the normal value. This allows sampling the start bit one more time to verify its correctness as well as allowing a wider range for transmitter signal to be stabilize. After the verification, system enters the DATA state. During this state RX samples data from mid periods then loads them into a register. When 8 bits are collected, state changes to STOP_BIT. As another measurement of verification, stop bit is checked and if it is valid the received and constructed data is outputted as a 8-bit data.

UART RTL View can be found in the figure 5.

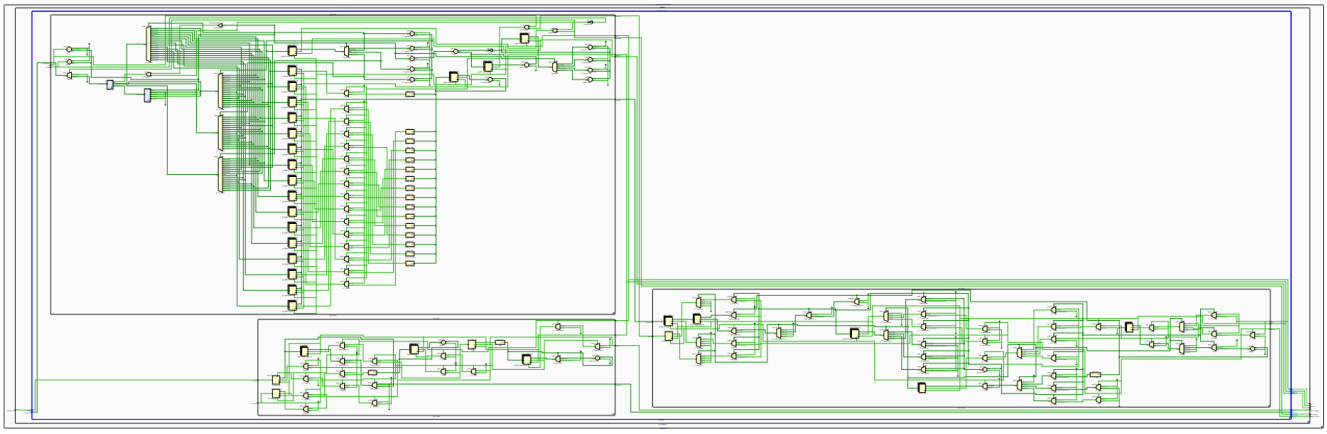

Figure 5: RTL View of UART module.

Testbench

To asses the functionality of the processor, we have realized a fully automatic **cocotb** testbench. The testbench is written in python and works by comparing our project with a performance model clock-by-clock for assertion. If a mismatch occurs between the performance model's and the DUT's register file or PC value, then testbench stops and showcases the error. The performance model works by collecting the instruction from an instruction list that corresponds to the Assembly code that we have written, and then extracting the operation from that instruction. To correctly define the result for each case in performance model, the register values are stored as binary values; however, binary values are logged as signed decimal numbers for better readability.

The Assembly code is written to cover all of the assigned instructions of this project. The PC value only increments within the program, if all the instructions are executed correctly. Some branches are should not be taken for a successful run, but if taken, the PC goes to another instruction, which results in a change that can be easily detectable, or the PC decrements, which is not the characteristic of the program. Each instruction is executed within the code at least 1 time, while some instructions are used repeatedly to showcase the other instructions or UART Peripheral, such as SB instruction.

The testbench also includes a helper lib for extensive logging capabilities for debugging purposes.

One can find the instructions set used to test functionality.. We prepared the instruction set in mnemonic form. Then, we write a python code to automatically encode the mnemonic instructions to RISC-V instruction hexadecimal code.

Table 2: Instructions used in testing the implementation

00	addi r1, r0, 5	# r1 = 5
04	slli r2, r1, 2	# r2 = 20
08	xor r3, r1, r2	# r3 = 17
0C	sub r4, r3, r2	# r4 = -3
10	sll r5, r4, r1	# r5 = -96
14	auipc r6, 0	# r6 = 20
18	srai r7, r2, 1	# r7 = 10
1C	or r8, r7, r5	# r8 = -86
20	andi r9, r6, 4	# r9 = 4
24	slt r10, r5, r6	# r10 = 1
28	sb r1, 0xA0(r0)	# [160] = 5
2C	bne r2, r6, 16	# NOT TAKEN, if taken PC= 0x38
30	beq r2, r6, 16	# Taken 0x40
38	add r2, r3, r4	# NOT EXECUTED, else r2 = 14
40	add r2, r6, r8	# r2 = -66
44	sb r2, 0(r7)	# [10] = 0x0000_00BE, 190
48	lb r12, 0(r7)	# r12 = -66
4C	lbu r13, 0(r7)	# r13 = 190
50	bltu r12, r13, 16	# NOT TAKEN, if taken PC= 0x60
54	blt r12, r13, 16	# Taken 0x64
60	jal r30, -96	# NOT EXECUTED, r30 = 84, PC=0x0 ERROR
64	jal r30, 12	# r30 = 88, PC=0x70
70	sra r14, r5, r1	# r14 = -3
74	srl r15, r5, r1	# r15 = 134217725
78	bge r14, r15, 12	# NOT TAKEN, if taken PC= 0x84
7C	bgeu r14, r15, 12	# Taken 0x88
84	jal r30, -132	# NOT EXECUTED, r30 = 0x88, PC= 0x0 ERROR
88	sh r15, 0(r6)	# [20] = 0x0000_FFFD, 65533

8C	sw r15, 0(r9)	# [4] = 0x07FF_FFFD, 134217725
90	lhu r16, 0(r6)	# r16 = 655533
94	lh r17, 0(r9)	# r17 = -3
98	lui r18, 0xFFFF0000	# r18 = -65536
9C	and r19, r15, r18	# r19 = 134152192
A0	slti r20, r19, 0x800	# r20 = 0
A4	sltiu r21, r19, 0x800	# r21 = 1
A8	ori r22, r21, 0x0FE	# r22 = 255
AC	xori r23, r22, 0x0AA	# r23 = 85
B0	jalr r31, r13, 0x002	# r31 = 0xB4 = 180
C0	srli r24, r23, 0x002	# r24 = 21
C4	sltu r11, r5, r6	# r11 = 0
C8	addi r1, r0, 65	# r1 = 65, ASCII(”A”)
CC	addi r2, r0, 0x82	# r2 = 0x82, gibberish ascii
D4	addi r3, r0, 66	# r3 = 66, ASCII(”B”)
D8	addi r4, r0, 0x42	# r4 = 0x42, ASCII(”B”)
DC	addi r5, r0, 5	# r5 = 5, not used
E0	sb r1, 0x400(r0)	# UART TRANSMIT ASCII(”A”)
E4	sb r2, 0x400(r0)	# UART TRANSMIT gibberish
E8	sb r3, 0x400(r0)	# UART TRANSMIT ASCII(”B”)
EC	sb r4, 0x400(r0)	# UART TRANSMIT ASCII(”B”)
F0	lw r26, 0x404(r0)	# UART RECIEVE: in tb 0xFFFF_FFFF, in fpga whatever in FIFO
F4	addi r5, r0, 38	# r5 = 38, 0x26 in hex, our group id

Conclusion

In this report, the design and verification process of a Single-Cycle RISC-V Processor is examined. A datapath and controller design approach similar to the **Single Cycle Processor** we have tackled in the first experiment was sufficient to some degree but some additions and removals were made to properly configure the modules' connections.Most significant change from the ARM counterpart of this processor is the lack of a shifter in the overall design. The resulting system was satisfactory and was verified with a testbench that works flawless. The project was a significant step to integrating the designs we did throughout the semester by adding interfaces as well as learning to adapt changes like the transition between the ARM processor and RISC-V processor.

References

[1] S. Harris and D. Harris, *Digital design and computer architecture, RISC-V edition.* Oxford, England: Morgan Kaufmann, Nov. 2021.