Universidade Federal de Santa Catarina

EEL 510389 - Digital Systems and Reconfigurable Logic Devices

Design and implementation in VHDL for FPGAs of a single cycle RISC-V based architecture

Author: Carlos Raúl Morales Hernández

UFSC/CTC/PPGEEL, 2019/1

Florianopolis, Junio 2019

Abstract

The work presents the design of a single cycle reduced version of the RISC-V architecture described by David A. Patterson and John L. Hennessy in the book, “Computer Organization and Design RISC-V Edition: The Hardware Software Interface”, Morgan Kaufmann, 2017. The design is described in VHDL (Very High-Speed Integrated Circuit Hardware Description Language) to be implemented in an FPGA (Field-programmable gate array) and includes all the necessary hardware to implement some of the most common instructions in the RISC-V architecture. The functioning is tested by making use of three programs in assembly code, stored in the instruction memory, which use all the instructions and the input/output interface of the platform. Testbenches for the functionality of the design are provided.

Summary

List of Figures

Introduction

RISC-V is a new general-purpose, open-source ISA, usable in any hardware or software without royalties. It was developed at UC Berkeley starting in 2010. Although x86 and ARM are widely available and supported, they are complex, and the licensing model is difficult for experimental and academic use, also, developing a microprocessor is a very hard and multidisciplinary task. Normally licenses can cost around $1M to $10M and the negotiation time can vary from 6 to 24 months and doesn't even let you design your own core. RISC-V was created as the solution for this problem as one free and open ISA everyone can use.

The addition of a new open-source ISA like RISC-V to the market could lead to greater innovation via free-market competition from many more designers. Shared open core designs, which would mean shorter time to market, lower cost from reuse and fewer error. Processors becoming affordable for more devices, which helps expand the Internet of Things (IoTs).

RISC-V aims to serve all markets, including microcontrollers as well as image, graphics, and server processors. Therefor it must be consistent in across architectures, from an in-order scalar design to a heavily out-of-order design. To address this issue, RISC-V defines a guaranteed base integer instruction set of 32, 64, or 128 bits, a family of optional and predefined extensions, and a mechanism for creating variable-length extensions. The RISC-V ISA offers all the basic RISC features with a few twists that simplify the implementation, thereby reducing die area and potentially power consumption. Compared with today's two most popular ISAs, RISC-V offers considerable area savings, particularly for low-end designs, and the ability to add custom extensions.

Also, the RISC-V foundation is expanding really fast. In November 2018, the RISC-V Foundation announced a joint collaboration with the Linux Foundation. As part of this collaboration, the Linux Foundation will also provide an influx of resources for the RISC-V ecosystem, such as training programs, infrastructure tools, as well as community outreach, marketing and legal expertise. The foundation currently has board directors from companies like Google, Western Digital, NVIDIA, NXP and others. In 2017 Western

Digital was shipping one billion cores annually to fuel RISC-V and NVIDIA is updating some it’s cores for RISC-V implementations.

Objectives

Most popular and successful instruction set architectures (ISAs) are patented from companies like ARM, IBM and Intel, this prevent others from using them without license. The negotiations can take a lot of time and costs a lot of money, this make the use of this ISAs impossible for entrepreneurs and others with small resources. Also, a license doesn’t normally allow you to create your own core but to use their design. RISC-V architecture solves this problem by providing a free and open ISA that everyone can use for anything they want. This is why the goal of this project is to design a simple, open source core, based on the RISC-V architecture (RV32I), that can be implemented in any platform and capable of executing some of the most common instructions in a program.

Design Criteria

The ISA is cleanly architected to simplify implementation. The instruction encoding is highly regular and lacks complicated memory instructions. The benefit is that minimal RISC-V cores are much smaller than similar ARM or x86 cores, although the difference is not noticeable for more powerful cores.

Registers

RISC-V has 32 registers (x0 – x31) of 32 bits (RV32I) plus the program counter (PC) and any arithmetic operation must be performed from one of this 32 registers. These registers have functions by default that the compiler must follow, for example, the first register is wired to ground, that means is always zero, this was made to simplify the comparison with zero and some jumps.

Memory Management

The RISC-V address space is byte addressed and little-endian, and do not have alignment restrictions. The data transfer instructions are called load doubleword (LD) and store doubleword (SD). Loads and stores operate also on byte, 16 bits and 32 bits. Load operations make sign extension when loading but RISC-V also provides instructions to deal with unsigned numbers and to make unsigned loads like load byte unsigned (LBU).

Instructions

To simplify decoding, the instruction encoding is highly regular. Basic instructions are 32 bits long and are naturally aligned. The designers where very careful to ensure consistency between formats. For example, the opcode and operands are in fixed locations, simplifying the decode process. The source and destination registers fields are of 5 bits for decoding one of the 32 registers. There are basically six types of instructions as is shown in the following table.

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Table 1: RISC-V instructions format (Source: Computer Organization and Design RISC-V edition)

- rd (destiny register): The register destination operand. It gets the result of the operation.
- funct3: An additional opcode field.
- opcode: Basic operation of the instruction, and this abbreviation is its traditional name.
- rs1 (source register 1): The first register source operand.

- rs2 (source register 2): The second register source operand.
- funct7: An additional opcode field.

R-type instructions include all the operations between registers, this includes all the arithmetic and logic operations. The operands are located in the source registers and the result is stored in the destination register.

I-type instructions is used by arithmetic operands with one constant operand, this includes load instructions (being the offset the immediate operand). The 12-bit immediate is interpreted as a two's complement value so it can represent integers from $-2^{11}\ to\ 2^{11}$.

S-type is for store instruction. The immediate is broken in two fields but the source registers are kept in the same place.

SB-type is for branch instructions. This format can represent branch addresses from −4096 to 4094, in multiples of 2. It is only possible to branch to even addresses.

UJ-type is reserved exclusively for *jump-and-link instruction (jal)*, which is an unconditional jump. This instruction includes an immediate of 20 bits, but it can jump only to pair address. RISC-V uses PC-relative addressing for both conditional branches and unconditional jumps, because the destination of these instructions is likely to be close to the branch.

U-type is for *load upper immediate (lui)* instruction. This instruction loads a 20-bit constant into bits 12 through 31 of a register. The leftmost 32 bits are filled with copies of bit 31, and the rightmost 12 bits are filled with zeros. This instruction allows a 32-bit constant to be created with two instructions. RISC-V also allow very long jumps to any 32 bits address with two instructions, *lui* and *jarl.*

Addressing modes

RISC-V includes the following addressing modes:

1. Immediate addressing, where the operand is a constant within the instruction itself.
2. Register addressing, where the operand is a register.
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.

4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction.

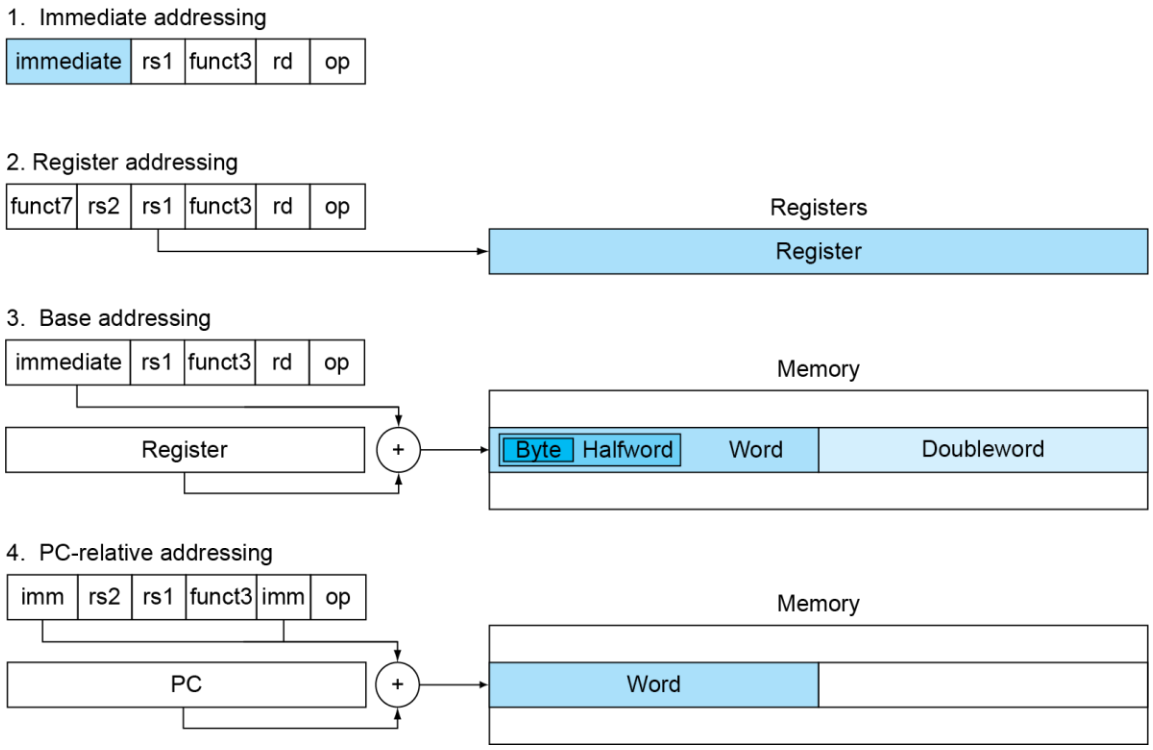

Figure 1: RISC-V addressing modes. Source: Computer Organization and Design RISC-V edition.

Methods

The design is based in the RISC-V architecture (RV64I) described by David A. Patterson and John L with some modifications. The components are described in VHDL and the behavior of them is represented using truth tables in the appendices. The VHDL code was written using Sublime Text version 3.2.1, this is a text editor free that allows VHDL code by installing a plugin. The architecture is single cycle, therefor, almost every component, except the memories are combinational logics to increase speed. Every component is described first independently and tested in a testbench, then are combined and also tested.

To verify the behavior of the design are used three programs stored in the instruction memory, the programs are written in assembly code and translated to RISC-V instructions

using a RIPES simulator, available in GitHub. The first program is the instructions set proposed, one after another to simply test the correct execution of every instruction. The second program uses the multiplication and the third program is for testing the jumps and branch instructions. The architecture designed allows also that any other program that uses this instructions repertory can be executed.

The implementation and simulation are made in the Intel SoC Cyclone® V FPGA using the software Quartus II ver. 15.0. An input/output strategy is defined for testing the physical behavior. The addresses from 0H to 7H in the data memory are reserved for output operations and the addresses from 8H to FH are used for inputs. This means that the results from operations is stored the output addresses and the data to manage in the programs in the inputs addresses.

Design Description

The design has to allow the execution of several instructions (see Table 2 in appendices). To execute any instruction, the architecture starts by fetching the instruction from memory. The memory containing the instructions is a ROM memory, since only read operations can be performed from it. The program counter is incremented by 4 to point to the next instruction. The Program Counter is a register that is updated with a new value in every clock cycle.

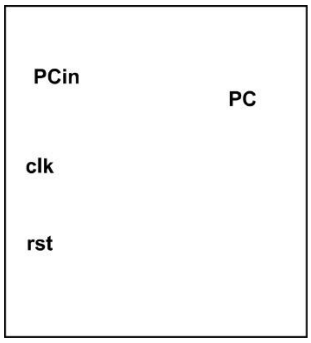

Figure 2: Program Counter Register. Source: Author.

The PC receives a reset signal to restart the counter to zero, a clock signal and the address for the next instruction. The CLK and reset signals comes from a block called

Edge Detector. This block is the interface with the buttons and switches that control the flow of the program. The block receives an input called CLK/Manual for indicate if the program runs by the CLK signal or manually, if it is manually, the PC will change the value with the signal Step, this is for step by step execution. The Edge Detector also detects when a button is pressed and generate a pulse that corresponds with the clock period, so it will behave as the new clock of the system until a reset is given.

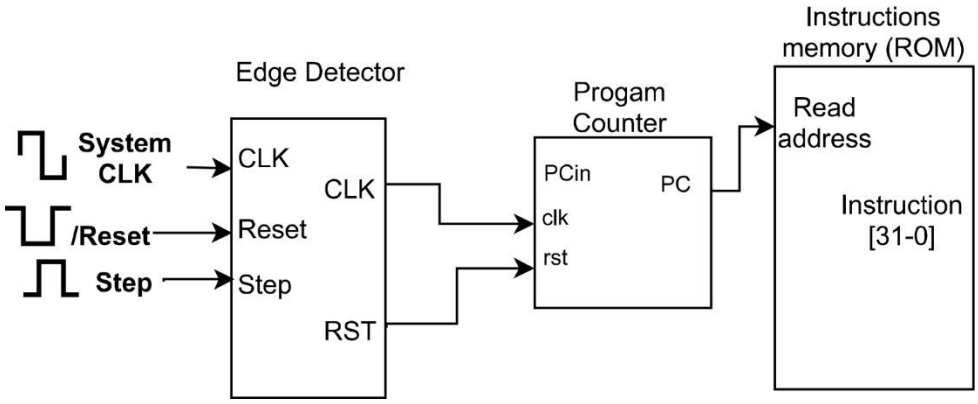

Figure 3: Edge detector and Program Counter. Source: Author.

R-type, I-type and S-type Instructions Datapath

Since the opcode field (bits 6:0), the source registers (19:15 and 24:20) and the destination register (11:7) are in the same position for many instructions the design can take advantage of this, the bits in those positions are used to decode the register in the register file that will be used.

R-Type instructions all reads 2 registers and perform an ALU operation on the content of the registers, and write the result in a register. The processor's 32 general-purpose registers are stored in a structure called a Register File. The Register File always outputs the contents of whatever register numbers are on the Source Registers. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

The ALU performs all the arithmetic and logic operations with the exception of multiplication, which is made in a separate unit, see Table 6 in the Appendices for ALU

operations. Another instruction is SLT or Set Less Than. This instruction will set all but the least significant bit to 0, with the least significant bit set according to the comparison. This comparison is done by subtracting the two registers in the ALU and checking the sign bit, if the sign bit is one then writes one the LSB if not writes zero. This instruction is R-type. The operation is done by the ALU and the output goes in the result.

For loads and stores is created a RAM memory. The Data Memory must be written on store instructions; hence, data memory has Write Control signals, an address input, and an input for the data to be written into memory. The operands of the ALU can be a register or an immediate, therefor, is placed a multiplexor at the input of the ALU. The output from the data memory goes to a combinational logic that sign-extend the byte in the 32-bits register loaded for *LB* instructions. The position in memory is selected by calculating the offset in the ALU and the data to be written comes from the register file (immediate addressing).

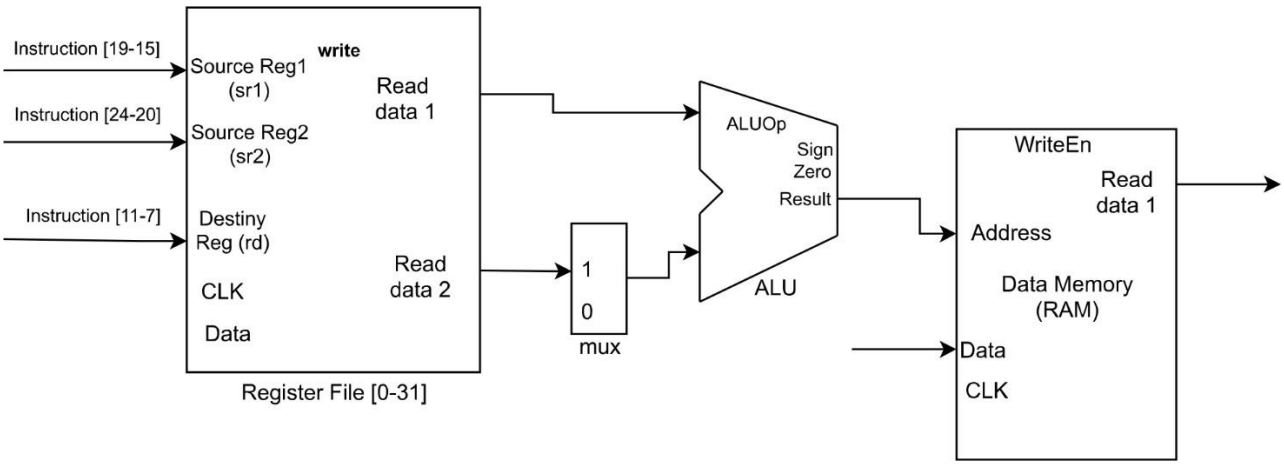

Figure 4: Register File, ALU and Memory connected. Source: Author.

The immediate generation unit takes the instruction as input and selects the immediate field to sign-extend it into a 32 bits result appearing on the output. The unit uses the opcode field to differentiate between instructions types. For loads and immediate arithmetic must select the bits must selects the bits [31:20], for S-type the bits [31:25] and [11:7], for branches [31,7,30:25,11:8] and for Jumps bits [31,19:12,20,30:21].

To allow the *SB (store byte)* instruction is placed a multiplexor at the input of the data to be written in memory. One input of the multiplexor is an and mask with zero of the higher part of the word, leaving only the byte (zeros from bit 31 to bit 8 and one from bit 7 to bit 0), the other input is the word. The data to be written in the register file can come from the ALU (R-types) or from the memory (load operations), so in order to select which one, a multiplexor is placed, the output from this multiplexor goes to the register file. The complete Datapath for these types of instructions is shown in Figure 5.

Another R-Type instruction is multiplication or MUL. This instruction takes the lower part (16 bits) of two register, makes the multiplication and stores the result in a third register. In order to do that, is placed a multiplication block. The multiplication could have been included in the ALU but it was decided to do it separately to show better that the design includes multiplication. The multiplication is performed in a single cycle, this was achieved designing a generic n x n array multiplier based in full adders (n = 32 in this case). The data to the Register File can come from many different places depending on the instruction (Memory, ALU, Multiplier, etc.), so a multiplexor is placed at the data input of the Register File to select the correct data,

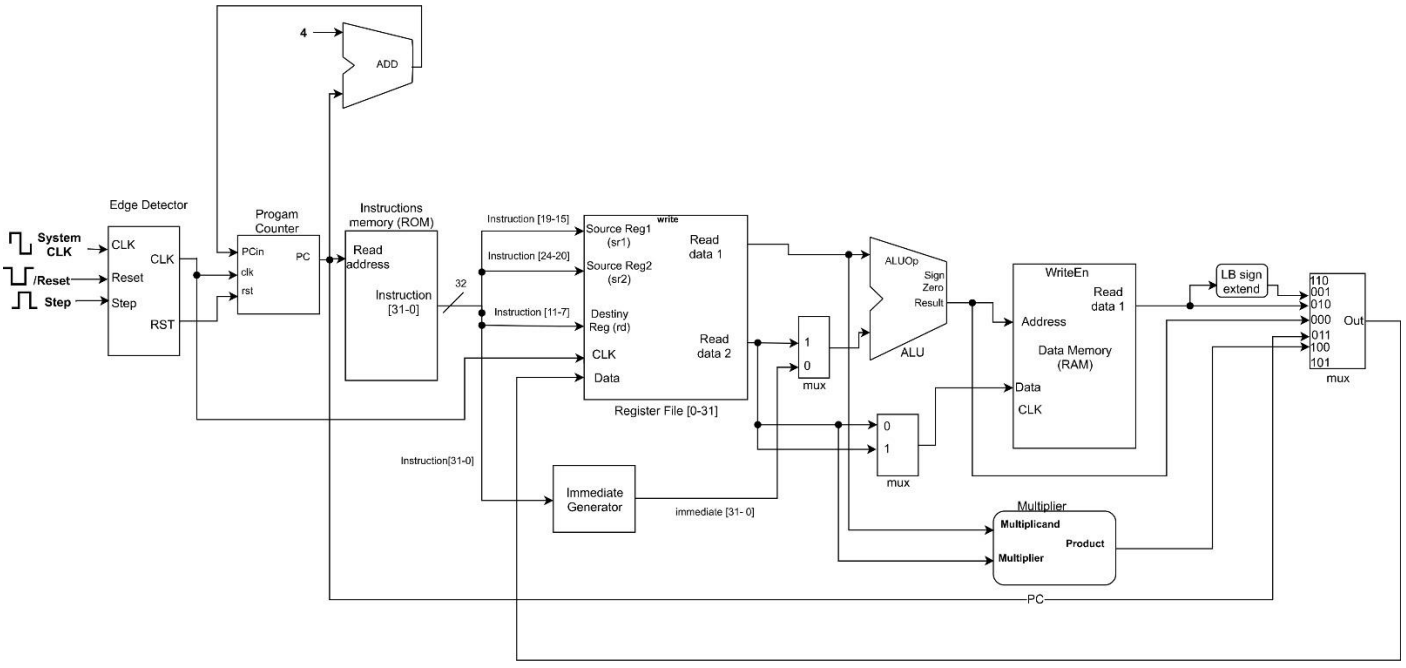

Figure 5: Datapath for R-type, I-type and S-type instructions. Source: Author.

Branches and Jumps

To implement branch instructions, is computed the branch target address by adding the sign extended offset field of the instruction to the PC. The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2. To determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address is used the Zero and Sign signals from the ALU.

To be able of handling more branch instructions and jumps, a combinational logic decides when there was a branch, a jump, or not. The unit receives the branch signal from the control unit, indicating which branch instruction is, then make a comparison either with the Sign bit or the Zero bit of the ALU, the output of this block is the signal PCSrc that controls the value with the one PC is going to be updated. See Table 5 in appendices for the behavior of the Branch controller.

The instruction JAL is of U-type. This instruction is an unconditional jump, it changes the value of PC with the value of the jump, contained in the immediate, and saves the following instruction in a register. This instruction writes in the register file the value of PC+4, therefor, the PC+4 output is connected also to the multiplexor that decides the value that is going to be written in the register file.

The *JALR* instruction is an unconditional jump to an address contained in a register. The ALU makes an addition to calculate the offset value, taking the immediate and the value in the register. The result of this operation is shifted once and added to PC. To select between which data is added to PC (immediate or result from operation) is placed a multiplexor controlled by the signal JUMP in the control unit, this signal is 0 if is a branch instruction and 1 if there is a *JALR* instruction. The Datapath for these instructions is shown in Figure 6.

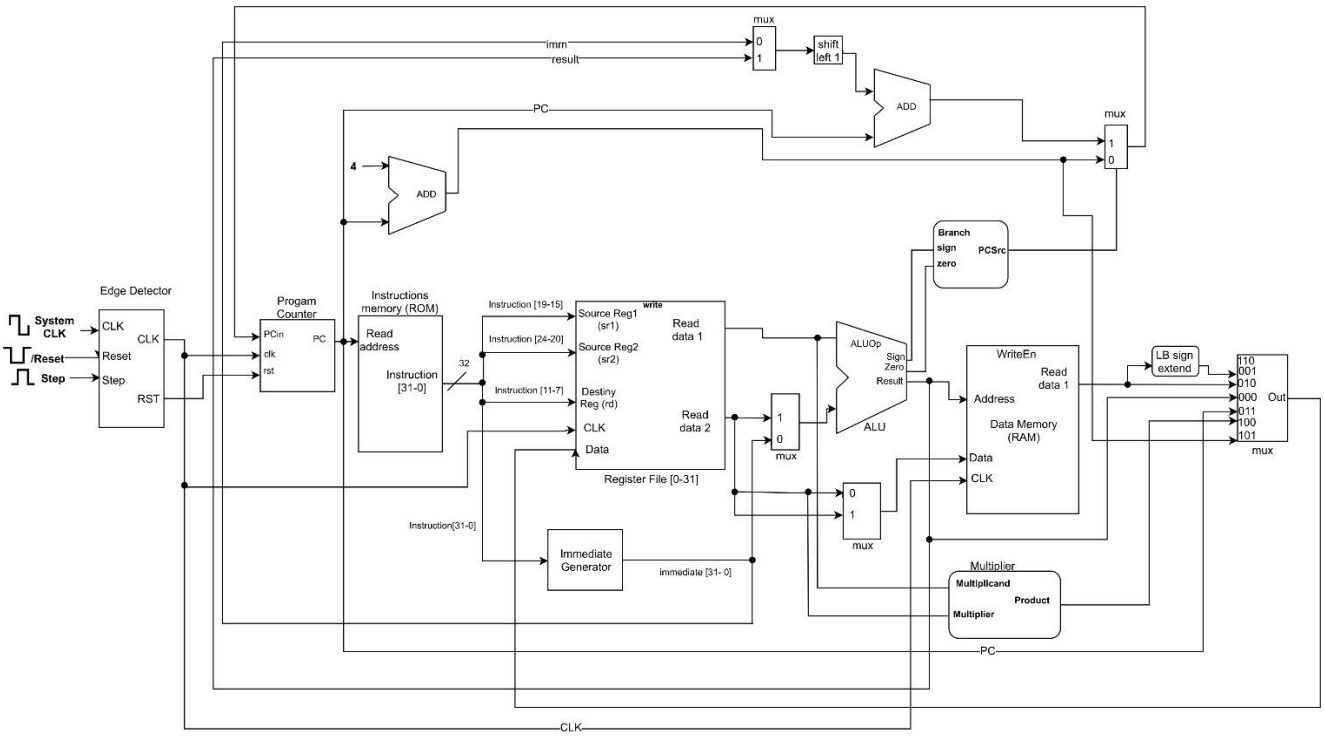

Figure 6: Datapath including Branches and Jumps. Source: Author.

Control Unit

To manage all the control signals there is a control unit. The control unit gets the opcode, the funct3 and the funct7, so it can differentiate between different instructions. Depending on the type of instruction, sets or clear the control signals for the different units, it is basically a truth table (see Truth Table in Appendices).

The Control Unit have 9 output signals:

- ToRegister: Selects the output for the Register File.
- Jump: Controls the multiplexor that allows the jump instruction.
- MemWrite: Allows the write in the Data Memory.
- Branch: Determine which type of Branch or Jump is.
- ALUOp: Determine the operation that must be performed by the ALU.
- StoreSel: Selects between the SB and SW data.

- ALUSrc: Selects the second operator for the ALU (register or immediate).
- WriteReg: Allows writes in the Register File.
- OutputSel: Controls the output logic to the leds and Seven Segments lamps in the board.

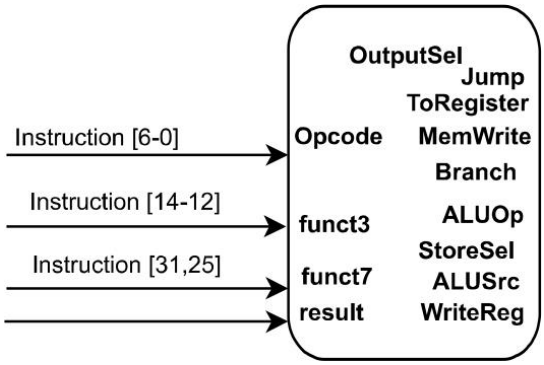

Figure 7: Control Unit. Source: Author

The unit receives the Opcode, Funct3 and Funct7, these are used to differentiate between instructions. The unit also receives the result from the ALU, this data is used to check the address obtained when using loads and stores instructions and compare it with the values used for the output logic. If there was a load or a store instruction the result from the ALU will indicate if the value to store or load comes or go to the input/output logic described next.

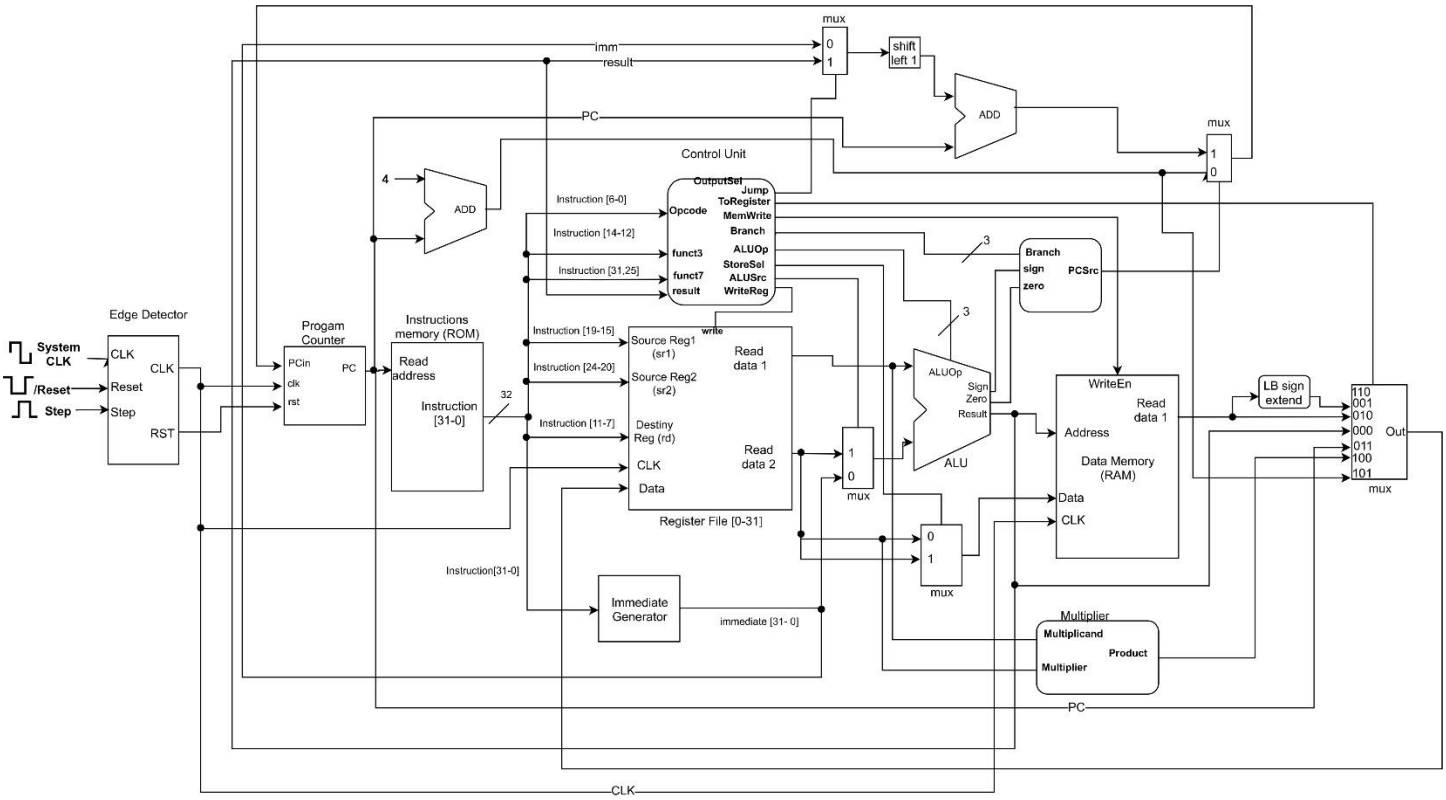

Figure 8: Datapath with Control Unit Integration. Source: Author.

Adopted memory mapped I/O procedure

For store operations, if the address is between 0H to 3H, then:

- *Store reg, 0(0):*
  - *LEDR(9 downto 0) <= Reg(9 downto 0)*
- *Store reg, 0(1):*
  - *HEX0 <= Reg(3 downto 0)*
  - *HEX1 <= Reg(7 downto 4)*
- *Store reg, 0(2)*
  - *HEX2 <= Reg(3 downto 0)*
  - *HEX3 <= Reg(7 downto 0)*
- *Store reg, 0(3)*
  - *HEX4 <= Reg(3 downto 0)*
  - *HEX5 <= Reg(7 downto 4)*

For input operations is only register in address 4H. Key0 is the global reset, every time is pressed, PC goes to address 0. KEY1 is the debug key. The first time KEY1 is pressed, the architecture enters in the debug mode. The same KEY1 button can be used for a step-by-step execution if needed.

- *Load Reg, 0(4)*
    - *Reg(9 downto 0) <= (SW9 downto 0)*
    - *Reg(10) <= KEY0*
    - *Reg(11) <= KEY1*
    - *Reg(12) <= KEY2*
    - *Reg(13) <= KEY3*

In order to identify the address in store and load instructions, the result from the ALU is added to the Control Unit as another input, since the result is the address for this type of instructions. The Control Unit determine if there is a Load or a Store instruction, if there is one, checks the address and resolve the path.

To show the result of a store operation is added a block called Output Logic. This block receives the register, a clock signal, a reset and a selector. The unit outputs the seven-segments code of the first byte in the register to the lamps, according to the address selected in the store instruction, as explained above.

For inputs operations, the register with the value of the switches and buttons is connected to multiplexor that controls the data that goes to the register file.

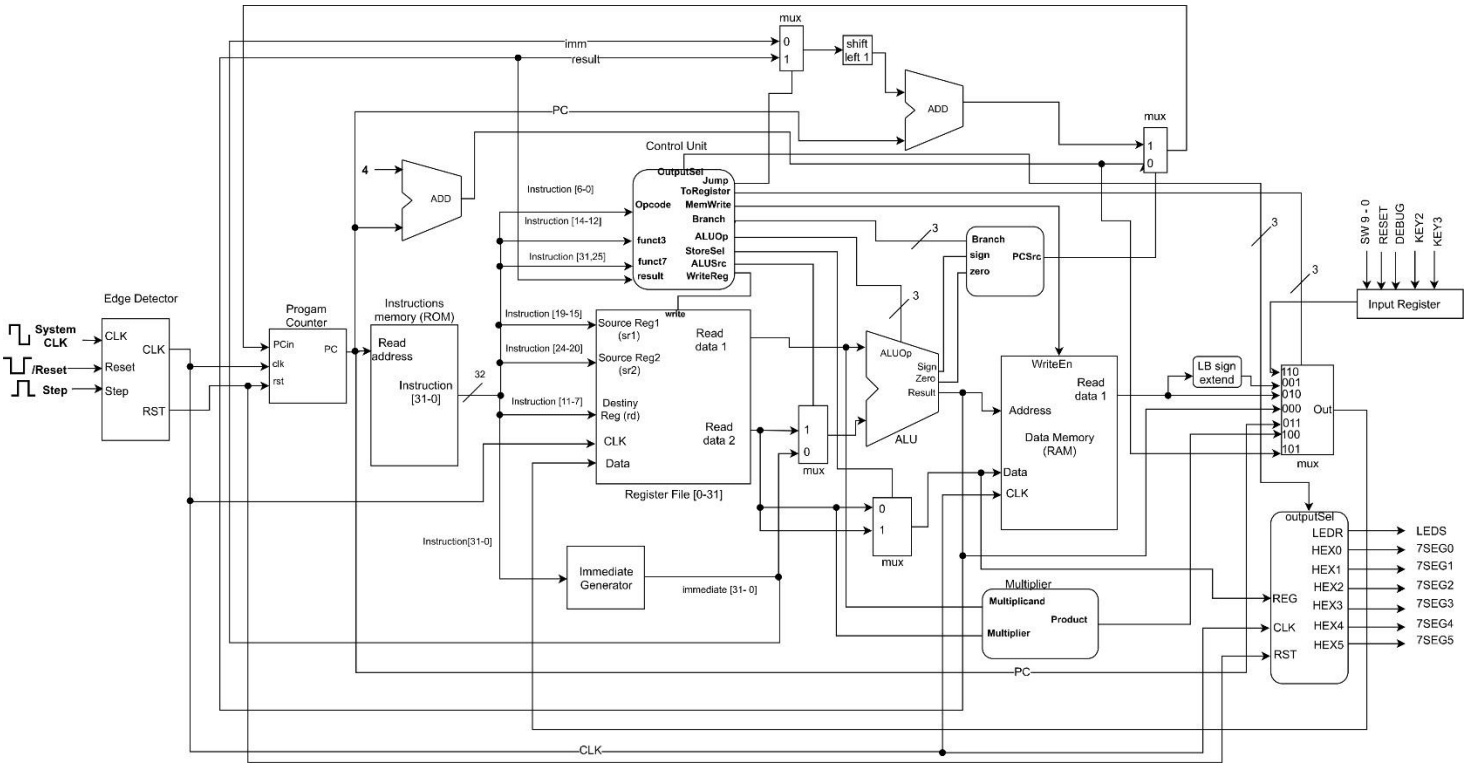

Figure 9: Datapath with Input/ Output Logic integration. Source: Author.

Results

All the components where developed in VHDL and tested using testbenches. Then where putted together and also tested. The CLK, Reset and Step signals are provided by the testbench for the simulation, when implementing, Reset and Step will be buttons. For the simulation all the instruction where placed in the Instructions Memory. Figures 10, 11 and 12 shows the testbench for the ALU, the Branch control and the control unit, the rest is verified in the programs.

Figure 13 the behavior for load and store instructions and Figure 14 for Branch instructions, in this case only BEQ. The program being executed is stored in the instruction memory and contain all the instructions in the repertoire being treated. Figure 13 the execution of *lb x2, 5(x3)*, *lw x4, 5(x3), sb x3, 5(x2), sw x5, 3(x2).*

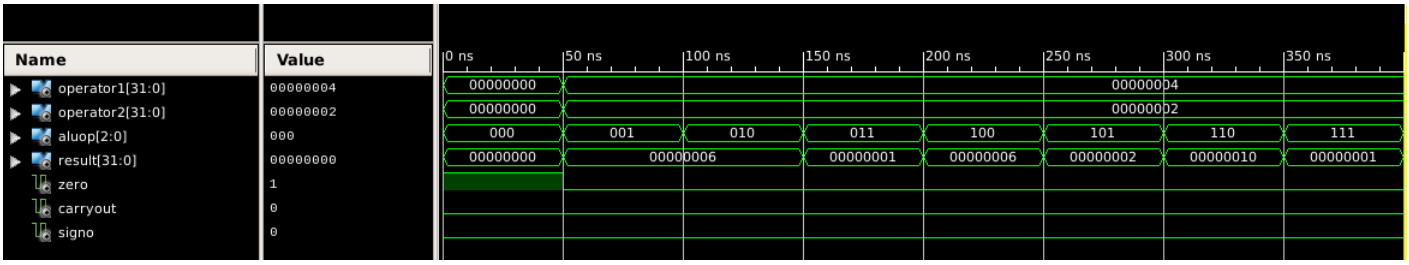

Figure 10: ALU's testbench.

The ALU's testbench performs all the instructions in the ALU by incrementing in one the ALUOp signal every period. The Testbench shows the result from all the arithmetic operations that the ALU can perform. To see the meaning of the instructions, see ALU's truth table in Appendices.

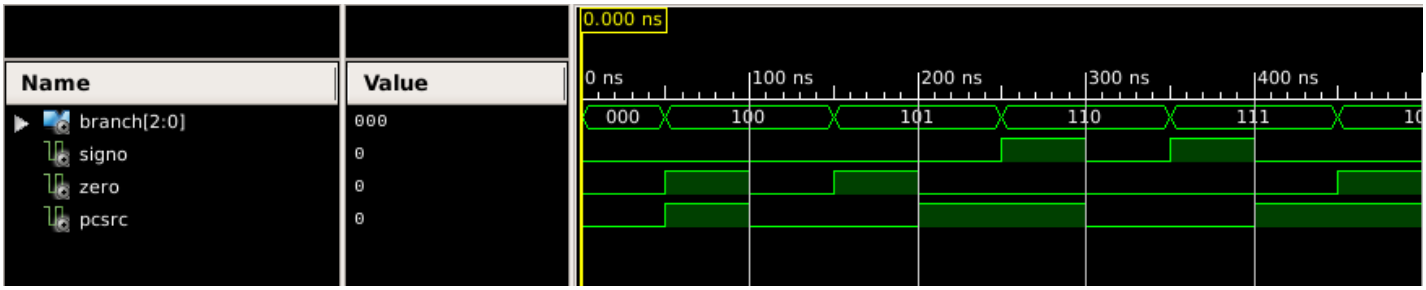

Figure 11: Branch Control Unit Testbench. Source: Author.

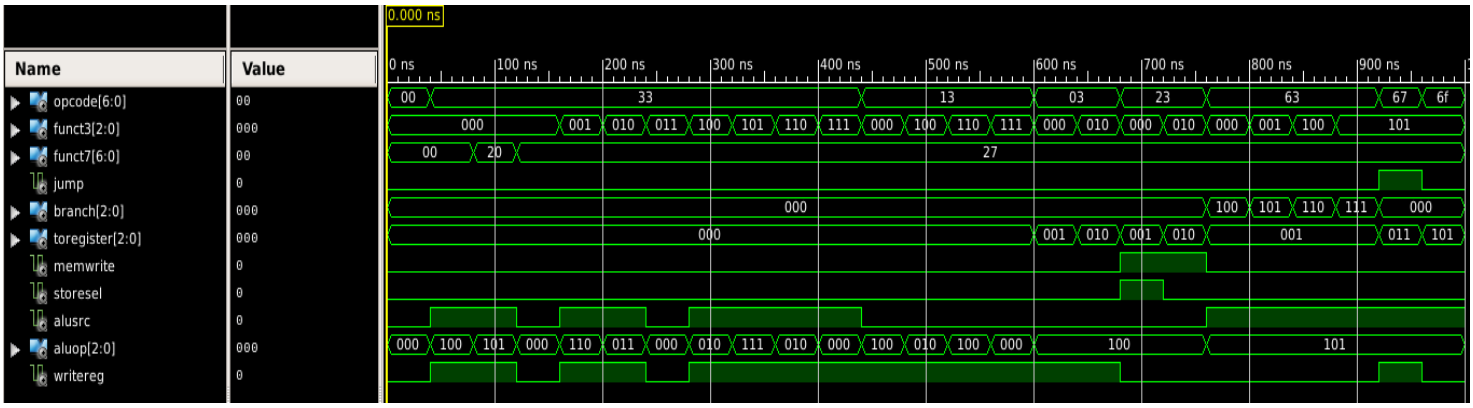

Figure 12: Control Unit Testbench. Source: Author.

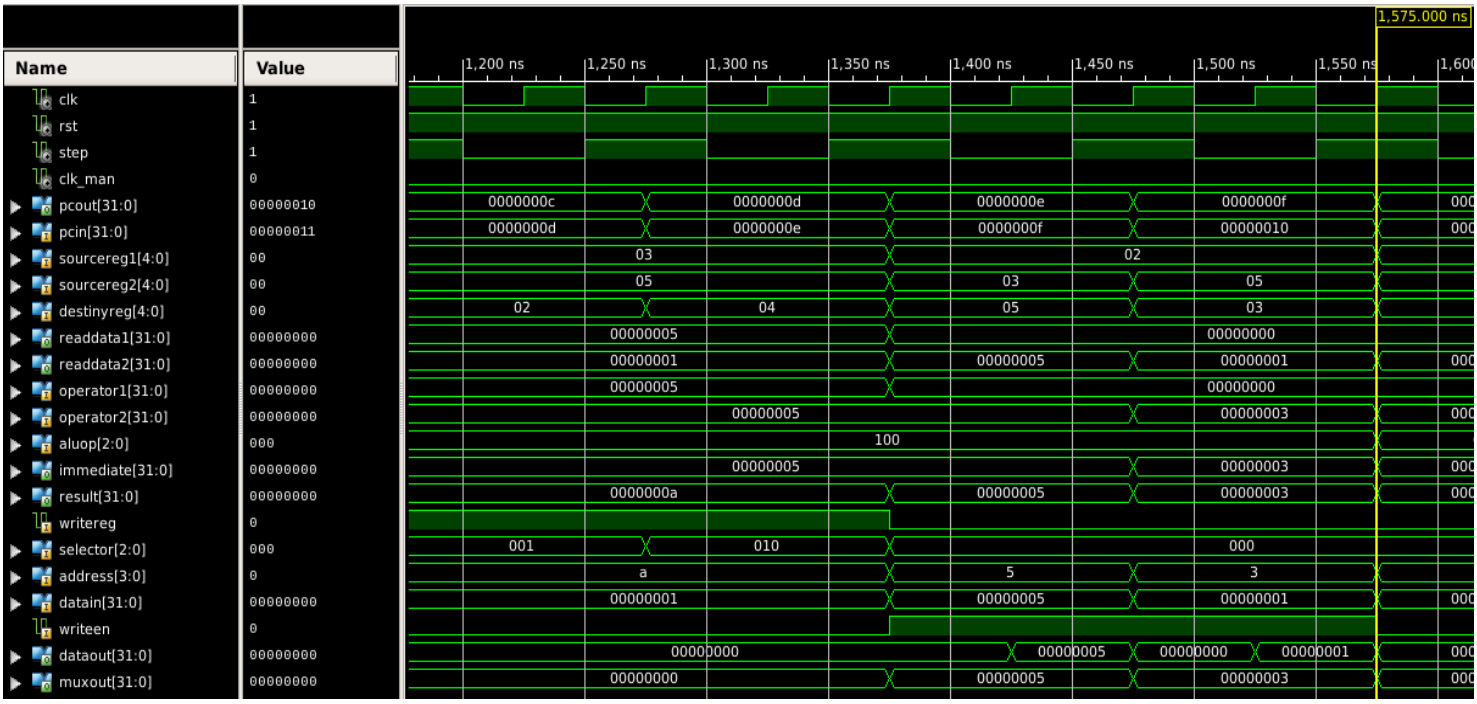

Figure 13: Testbench for Loads and Stores. Source: Author.

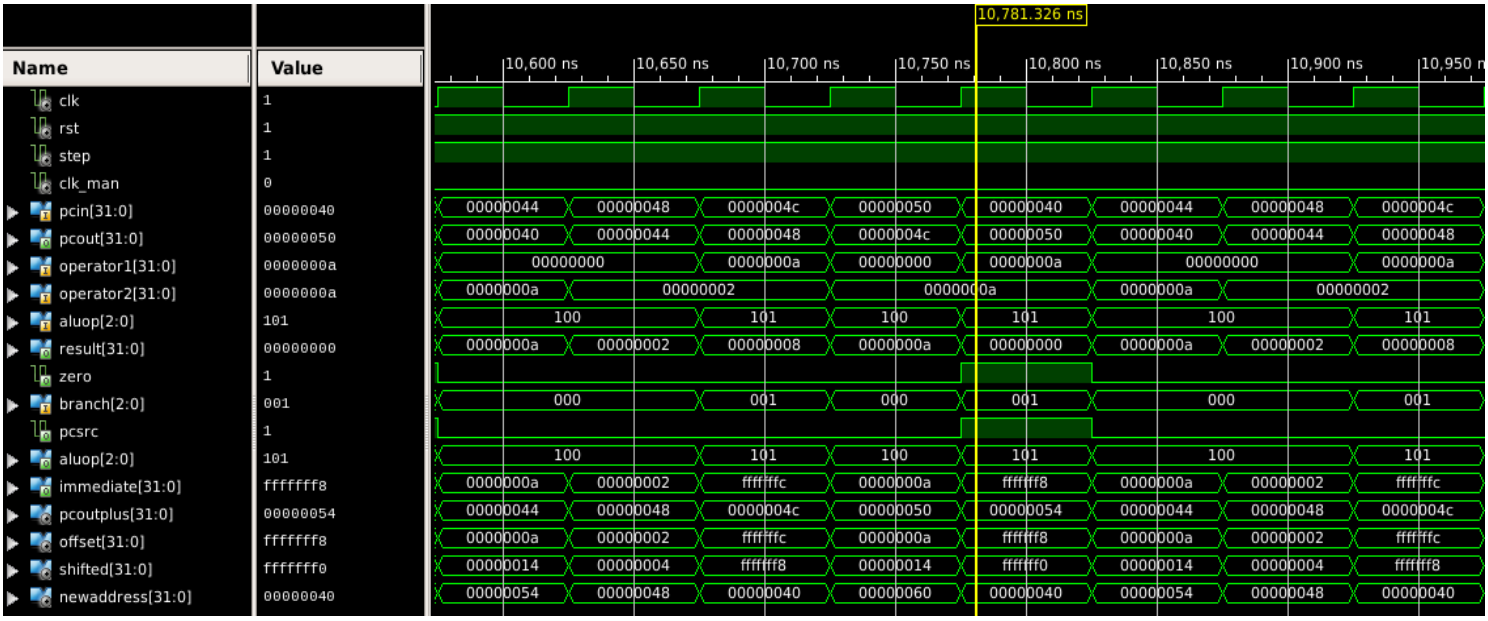

Figure 14: Testbench for Branch instructions. Source: Author.

Figure 15, 16 and 17 shows the Testbench for the Factorial Program to test multiplication. The number is read from a register simulating the input value that comes from the switches, in this case the number is three (3), therefor the result must be 6 and showed in the lamps and leds. The code for the factorial program is in the Appendices. The simulation is done using the Step signal, so the transitions occurs when the button is pressed.

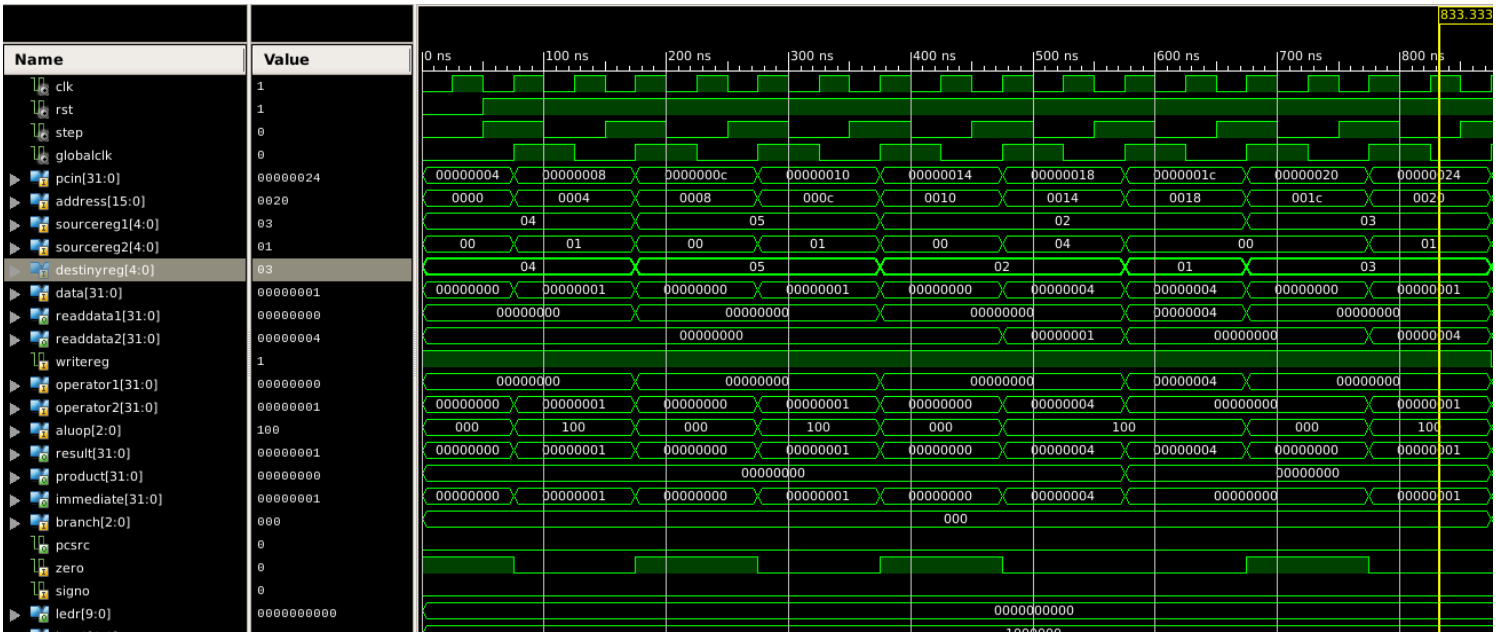

Figure 15: Factorial Program.

In the program, the clock is the signal Global CLK, which is the clock that receives the system, this signal will be equal to clock or the clock pulse detected when a button is pressed. The transitions occur with the rising edge. The program counter is incremented by four in every instruction.

This first instructions are arithmetic's instructions to clean the registers and initialize values. The sourcereg signals indicates the registers been addressed, the data signal the value written in the register. The readdata signals are the values read from the register file and the writereg flag if there is a write in the register. The operator signals are the inputs to the ALU and result is the output. Product is the result from the multiplication between readdata1 and readdata2. The immediate signal is the immediate generated.

Branch indicates if there is a branch. The LEDs starts with zero, which means off and the lamps displays zero.

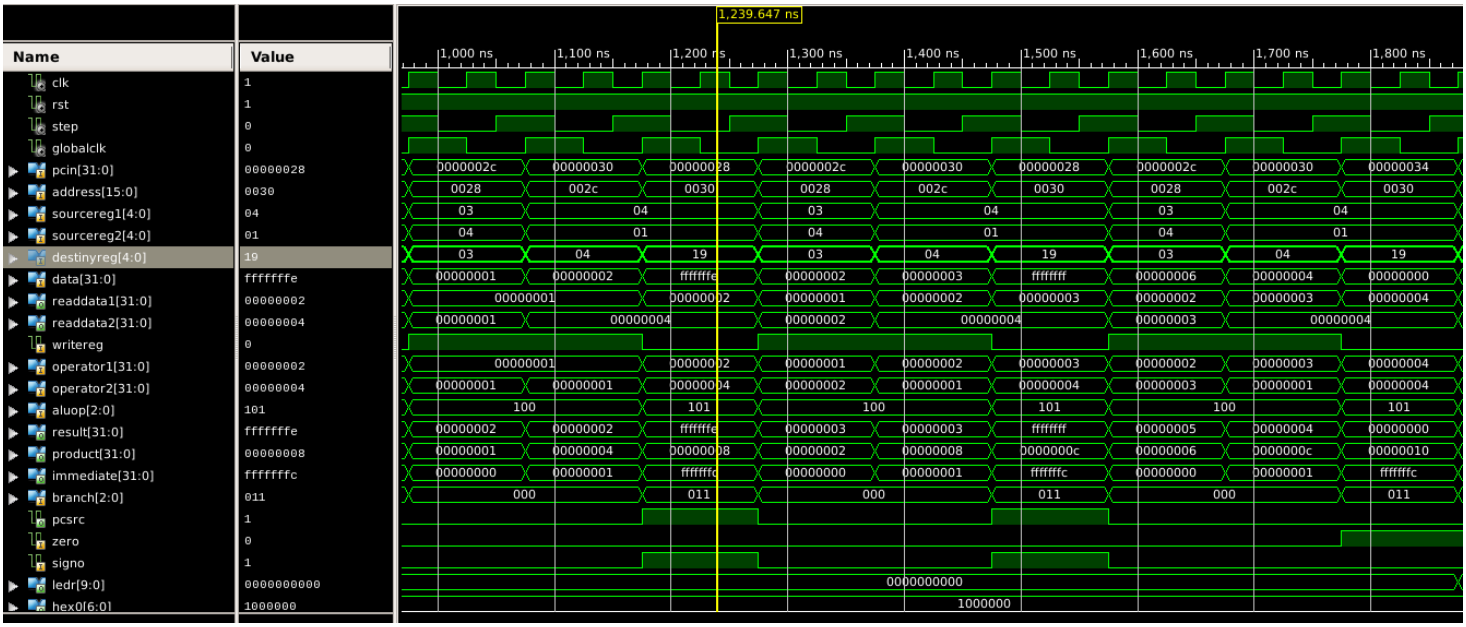

Figure 16: Factorial Program part two.

This part of the program shows the correct functioning of the branches in the program. PC changes from address 30H to address 28H and until the comparison is not true. Finally, the result is shown in the Leds and the Lamps (see figure 17).

Figure 18 shows the testbench for the arithmetic instructions. The program starts by making immediate additions, registers additions, subtractions and logic operations. Then some results are showed in the lamps. The program also tests the subtraction between for negative numbers and if the signo flag function correctly.

Figure 19 shows the Flow summary for the design implemented in a Cyclone V FPGA and Figure 20 the resource used by every component. The next figures show the RTL schematics generated by Quartus when compiling the design.

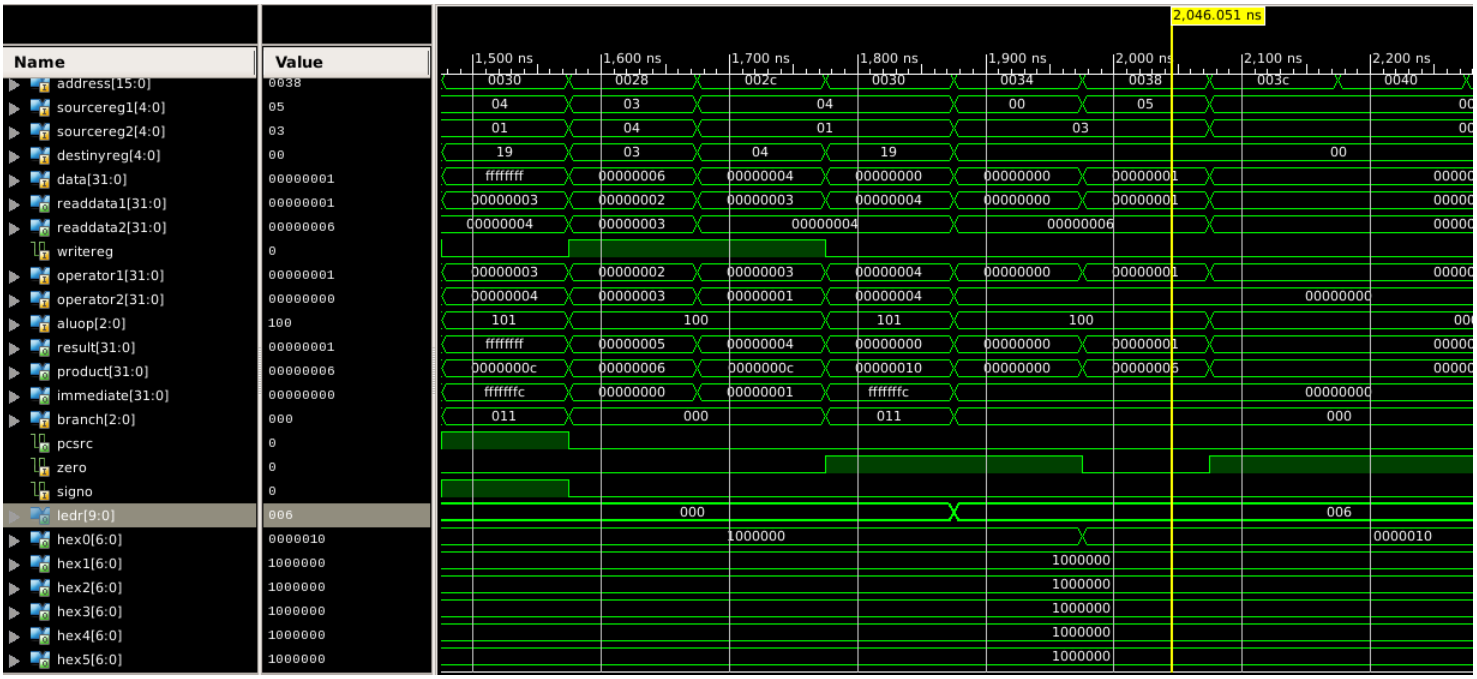

Figure 17: Factorial Program part three.

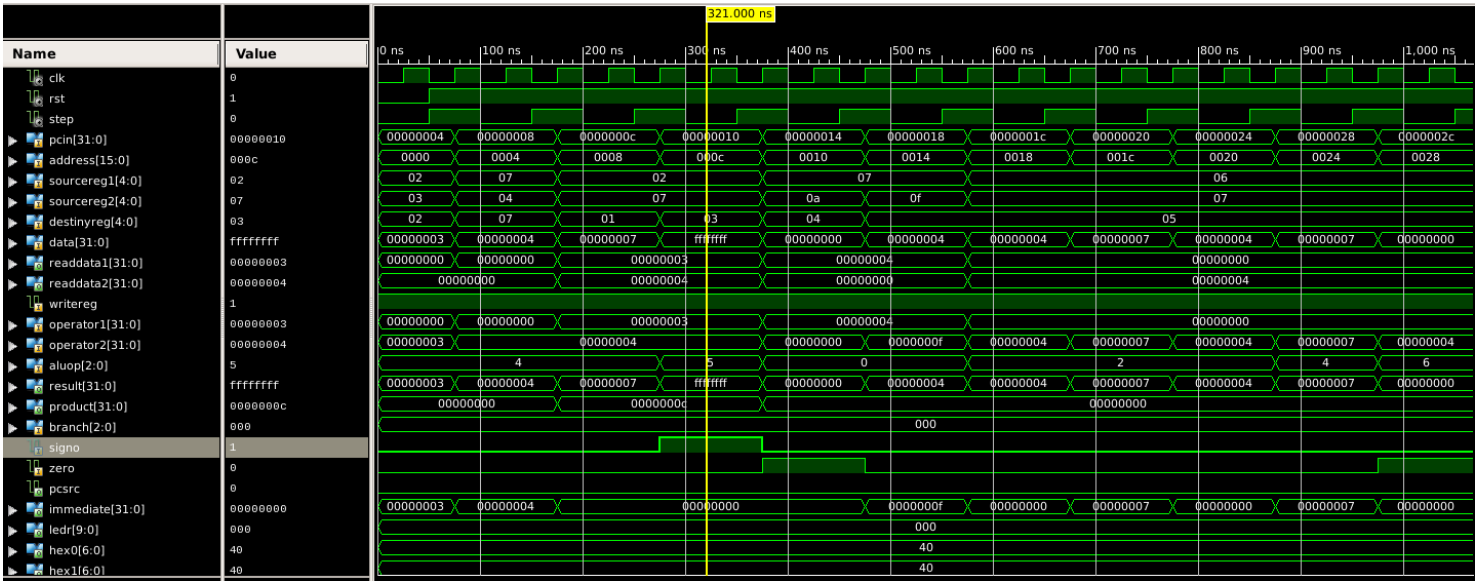

Figure 18: Arithmetic instructions program.

Flow Summary

<<Filter>>

Flow Status	Successful - Thu Jul 04 18:03:30 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	RISC_V_Micro
Top-level Entity Name	DataPath
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,051 / 32,070 (6 %)
Total registers	1154
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 19: Flow Summary.

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name
1	|DataPath	2638 (63)	1154 (1)	0	0	67	0	|DataPath
1	|ALU_RV32:ALU|	288 (288)	0 (0)	0	0	0	0	|DataPath|ALU_RV32:ALU
2	|Branch_Control:BRControl|	1 (1)	0 (0)	0	0	0	0	|DataPath|Branch_Control:BRControl
3	|Control:Ctrl|	54 (54)	0 (0)	0	0	0	0	|DataPath|Control:Ctrl
4	|Data_Mem:RAM|	299 (299)	128 (128)	0	0	0	0	|DataPath|Data_Mem:RAM
5	|Immediate_Generator:Imm|	56 (56)	0 (0)	0	0	0	0	|DataPath|Immediate_Generator:Imm
6	|Instruction_Mem:ROM|	189 (189)	0 (0)	0	0	0	0	|DataPath|Instruction_Mem:ROM
7	|Mux:Mux0|	14 (14)	0 (0)	0	0	0	0	|DataPath|Mux:Mux0
8	|Mux:Mux1|	24 (24)	0 (0)	0	0	0	0	|DataPath|Mux:Mux1
9	|Mux_ToRegFile:MuxReg|	159 (159)	0 (0)	0	0	0	0	|DataPath|Mux_ToRegFile:MuxReg
10	|OutputLogic:outInterface|	92 (50)	0 (0)	0	0	0	0	|DataPath|OutputLogic:outInterface
11	|PC:PCount|	1 (1)	31 (31)	0	0	0	0	|DataPath|PC:PCount
12	|Reg_File:RFILE|	759 (759)	992 (992)	0	0	0	0	|DataPath|Reg_File:RFILE
13	|multiplier2:Mult|	637 (206)	0 (0)	0	0	0	0	|DataPath|multiplier2:Mult
14	|rising_edge_detector:EdgDet|	2 (2)	2 (2)	0	0	0	0	|DataPath|rising_edge_detector:EdgDet

Figure 20: Resources Utilization by Entities.

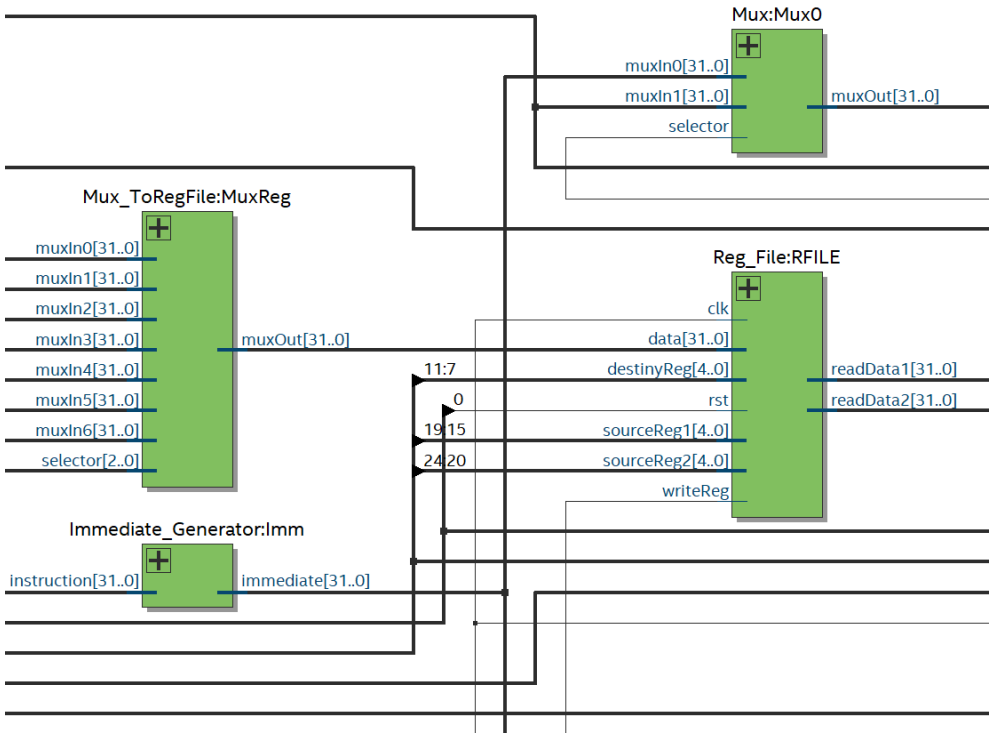

Figure 21: RTL view of Register File, immediate generator and the multiplexor for the register file.

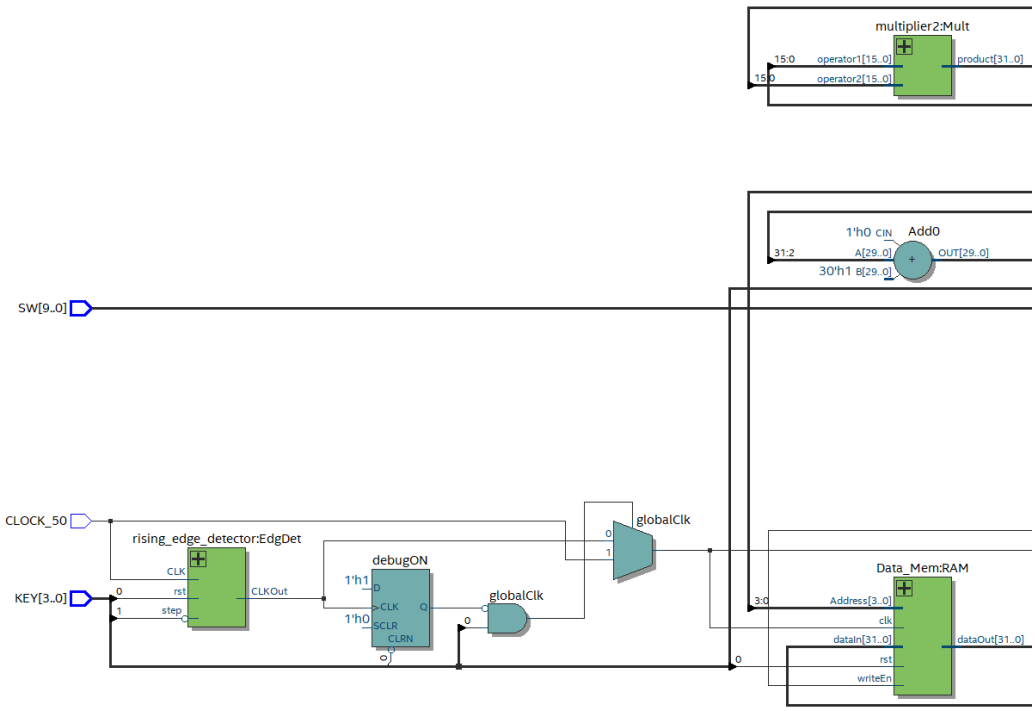

Figure 22: RTL view of the Data Mem, the Multiplier and the input configuration.

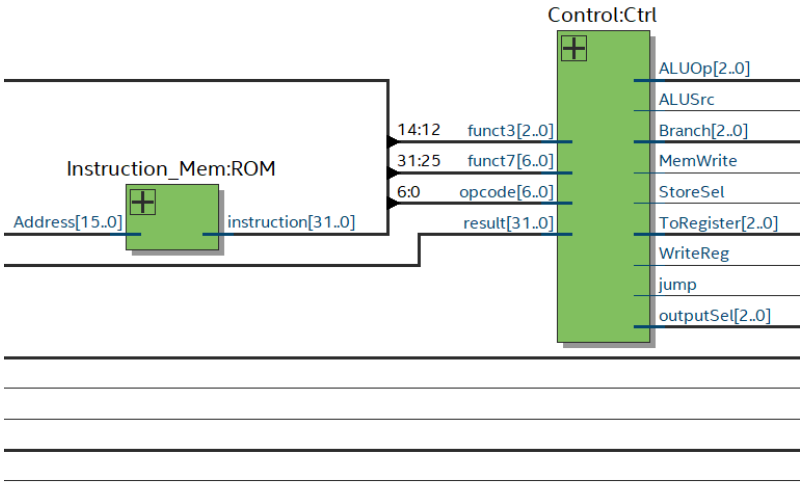

Figure 23: RTL view of the connections between the ROM and the Control Unit.

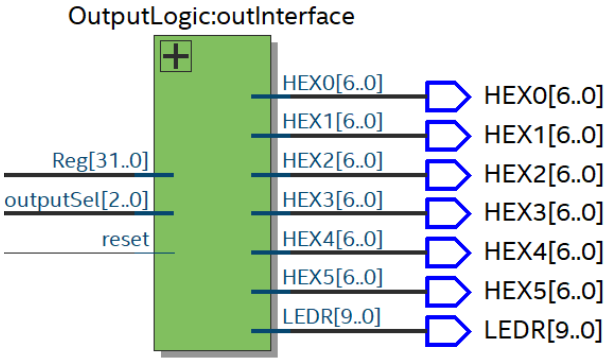

Figure 24: RTL view of the Output Logic to the Lamps.

Next Steps

Dates	Delivery 1	Delivery 2	Delivery 3	Delivery 4	Delivery 5	Delivery 6
Thursday, 23 May	Preliminary Report	x	x	x	x	x
Thursday, 30 May	x	VHDL + Testbench of Datapath. Revised Document	x	x	x	x
Thursday, 6 June	x	x	VHDL + Testbench of the Control Unit. Revised Document.	x	x	x
Thursday, 13 June	x	x	x	VHDL + Testbench of integration of units Revised Document.	x	x
Thursday, 20 June	x	x	x	X	VHDL + Testbench + implementation. Revised Document.	x
Jul 4	x	X	x	x	x	Final Doc

References

1) David A. Patterson, John L. Hennessy, “Computer Organization and Design RISC-V Edition: The Hardware Software Interface”, Morgan Kaufmann, 2017.
2) https://riscv.org/risc-v-foundation/
3) O’Connor, Rick. RISC-V ISA & Foundation Overview. https://content.riscv.org/wp.../1-RISC-V-ISA-Foundation-Overview-DAC2018-1.pdf
4) Kanter D. RISC-V offers simple, modular ISA. The Linley Group MICROPROCESSOR Report (March 2016). 2016 Mar.
5) https://github.com/rv8-io/rv8/blob/master/doc/pdf/riscv-instructions.pdf
6) https://rv8.io/

Appendices

Comment	Type	Instruction example	opcode	Funct3	Func7
Addition (with overflow)	R-type	ADD RD, RS1, RS2	0110011	000	0000000
Addition immediate (with overflow)	I-type	ADDI RD, RS1, IMM	0010011	000	n.a.
Subtract (with overflow)	R-type	SUB RD, RS1, RS2	0110011	000	0100000
AND	R-type	AND RD, RS1, RS2	0110011	111	0000000
AND immediate	I-type	ANDI RD, RS1, IMM	0010011	111	n.a.
XOR	R-type	XOR RD, RS1, RS2	0110011	100	0000000
XOR immediate	I-type	XORI RD, RS1, IMM	0010011	100	n.a.
OR	R-type	OR RD, RS1, RS2	0110011	110	0000000
OR immediate	I-type	ORI RD, RS1, IMM	0010011	110	n.a.
Shift Left logical	R-type	SLL RD, RS1, RS2	0110011	001	0000000
Shift Right logical	R-type	SRL RD, RS1, RS2	0110011	101	0000000
Set less than	R-type	SLT RD, RS1, RS2	0110011	010	0000000
Branch on equal	SB-type	BEQ RS1, RS2, OFFSET	1100011	000	n.a.
Branch on not equal	SB-type	BNE RS1, RS2, OFFSET	1100011	001	n.a.
Branch less than	SB-type	BLT RS1, RS2, OFFSET	1100011	100	n.a.
Branch greater than equal	SB-type	BGE RS1, RS2, OFFSET	1100011	101	n.a.
Jump and Link	UJ-type	JAL RD, OFFSET	1101111	n.a.	n.a.
Jump and Link register	I-type	JALR RD, OFFSET (RS1)	1100111	000	n.a.
Load Byte	I-type	LB RD, OFFSET (RS1)	0000011	000	n.a.

Load Word	I-type	LW RD, OFFSET (RS1)	0000011	010	n.a.
Store Byte	S-type	SB RS2, OFFSET (RS1)	0100011	000	n.a.
Store Word	S-type	SW RS2, OFFSET (RS1)	0100011	010	n.a.
Multiply	R-type	MUL RD, RS1, RS2	0110011	000	0000001

Table 2: Instructions treated.

Opcode	Funct3 /bit30/ bit25	Type	Instruction	Jump	Branch	ToRegister	MemWrite/MemRead	StoreSel	ALUSrc	ALUOp	WriteReg
0110011	000	R-type	ADD	0	000	00	0/0	0	1	100	1
0110011	000/1	R-type	SUB	0	000	00	0/0	0	1	101	1
0110011	111	R-type	AND	0	000	00	0/0	0	1	000	1
0110011	110	R-type	OR	0	000	00	0/0	0	1	001	1
0110011	100	R-type	XOR	0	000	00	0/0	0	1	010	1
0110011	010	R-type	SLT	0	000	00	0/0	0	1	011	1
0110011	001	R-type	SLL	0	000	00	0/0	0	1	110	1
0110011	101	R-type	SRL	0	000	00	0/0	0	1	111	1
0010011	000	I-type	ADDI	0	000	00	0/0	0	0	100	1
0010011	111	I-type	ANDI	0	000	00	0/0	0	0	000	1
0010011	100	I-type	XORI	0	000	00	0/0	0	0	010	1

0010011	110	I-type	ORI	0	000	00	0/0	0	0	001	1
1100111	000	I-type	JALR	1	101	11	0/0	0	0	000	1
0000011	000	I-type	LB	0	000	01	0/1	0	0	100	1
0000011	010	I-type	LW	0	000	10	0/1	0	0	100	1
0100011	000	S-type	SB	0	000	00	1/0	1	0	100	0
0100011	010	S-type	SW	0	000	00	1/0	0	0	100	0
1100011	000	SB-type	BEQ	0	001	00	0/0	0	1	101	0
1100011	001	SB-type	BNQ	0	010	00	0/0	0	1	101	0
1100011	100	SB-type	BLT	0	011	00	0/0	0	1	101	0
1100011	101	SB-type	BGT	0	100	00	0/0	0	1	101	0
1101111	Don't care	UJ-Type	JAL	0	110	00	0/0	0	1	000	0

Table 3: Truth table for the Control Unit.

ToRegister	Multiplexor Out
000	Result
001	Load Byte
010	Load Word
011	PC
100	Multiplication

101	PC+1

Table 4: Behavior of multiplexor to data.

Branch	Instruction	Comparison	PCSrc
000	No branch	No compare	0
001	BEQ	Zero /! Zero	1/0
010	BNQ	Zero/! Zero	0/1
011	BLT	Sign/! Sign	1/0
100	BGT	Sign/! Sign	0/1
101	JALR	X	1
110	JAL	X	1

Table 5: Truth table of Branch Controller.

ALUOp	Operation
000	AND
001	OR
010	XOR
011	SLT
100	ADD
101	SUB
110	SLL
111	SRL

Table 6: ALU truth table.

ADD X1, X2, X7	"00000000","00010010","10000010","10010011"
ADDI X5, X6, 5	"00000000","00010010","10000010","10010011”
SUB X3, X5, X7	"01000000","01110010","10000001","10110011"
AND X5, X6, X7	"00000000","01110011","01110010","10110011"
ANDI X5, X6, 0	"00000000","00000011","01110010","10010011"
XOR X5, X6, X7	"00000000","01110011","01000010","10110011"
XORI X5, X6, 7	"00000000","01110011","01000010","10010011"
OR X5, X6, X7	"00000000","01110011","01100010","10110011"
ORI X5, X6, 7	"00000000","01110011","01100010","10010011"
SLL X5, X6, X7	"00000000","01110011","00010010","10110011"
SRL X5, X6, X7	"00000000","01110011","01010010","10110011"
SLT X5, X6, X7	"00000000","01110011","00100010","10110011"
SB X3, 5(X2)	"00000000","00110001","00000010","10100011"
SW X5, 3(X3)	"00000000","01010001","00100001","10100011"
LB X2, 5(X2)	"00000000","01010001","00000001","00000011"
LW X6, 3(X3)	"00000000","00110001","10100011","00000011"

Table 7: Code of First Program

ANDI X1, X1, 0X0	00000000000000001111000010010011
ANDI X2, X2, 0X0	00000000000000010111000100010011
ANDI X3, X3, 0X0	00000000000000011111000110010011
ANDI X4, X4, 0X0	00000000000000100111001000010011
ANDI X5, X5, 0X0	00000000000000101111001010010011

ANDI X7, X7, 0X0	00000000000000111111001110010011
ADDI X7, X7, 0X1	00000000000100111000001110010011
ADDI X3, X3, 0X1	00000000000100011000000110010011
ADDI X1, X1, 0X5	00000000010100001000000010010011
ANDI X8, X8, 0X0	00000000000001000111010000010011
ADDI X8, X8, 0X0	00000000000001000000010000010011
ADDI X9, X9, 0X7	00000000011101001000010010010011
BLT X5, X7, NEXTC	00000000011100101100011001100011
ADDI X4, X5,0X0	00000000000000101000001000010011
JAL X0, STORE	00000001000000000000000001101111
ADD X4, X2, X3	00000000001100010000001000110011
ADDI X2, X3,0	00000000000000001100000100010011
ADDI X3, X4,0	00000000000000010000000110010011
SB X4,0(X8)	00000000010001000000000000100011
ADDI X8, X8,1	00000000000010100000010000010011
ADDI X5, X5,1	00000000000100101000001010010011
BEQ X8, X9, BEGIN	11111010100101000000011011100011
BGT X1, X5, FIBONACCI	11111100000100101100110011100011

Table 8: Code for Fibonacci Program

ANDI X4, X4, 0	**00000000000000100111001000010011**
ADDI X4,X4, 1	00000000000100100000001000010011
ANDI X5,X5,0	00000000000000101111001010010011
ADDI X5,X5,1	00000000000100101000001010010011

ANDI X2,X2, 0	00000000000000010111000100010011
ADDI X2,X2, 4	00000000010000010000000100010011
LW X1, 0(X2)	00000000000000010010000010000011
ANDI X3,X3 0	00000000000000011111000110010011
ADDI X3,X3, 1	00000000000100011000000110010011
BEQ X1,X0, BEGIN_FACT	11111100000000001000111011100011
MUL X3,X3,X4	00000010010000011000000110110011
ADDI X4, X4, 1	00000000000100100000001000010011
BGT X1,X4, FOR	11111110000100100100110011100011
SW X3,0(X0)	00000000001100000010000000100011
SW X3,0(X5)	00000000001100101010000000100011

Table 9: Code for Factorial Program.