22/09/2022 Enes Erçin

* Project 8
* Single Cycle RISC-V CPU

The eighth task which was given during internship at Lab is to build a single cycle CPU with RISC-V architecture. To successfully achieve the given task, it was necessary to understand basics of computer architecture as well as instruction set fundamentals. RISC-V is an open source instruction set which revolutionized way we interpret computers. The qualities that make RISC-V especially important are the reliability, easy to understand features, highly adaptable nature and a strong community. It is often compared with Linux as one being first majority used operating system and the other being first open source instruction set. RISC-V is thought to be more adaptable because base instruction set will remain unchanged yet it is possible to add custom instruction on top of it. By this methodology the devices which use RISC-V instruction sets will be still relevant although there are millions of different widely used custom RISC-V instructions. The fact that RISC-V is open source makes it possible to find a solution to the problem much more easily on top of that there are much more developer tools available designed by the open source community.

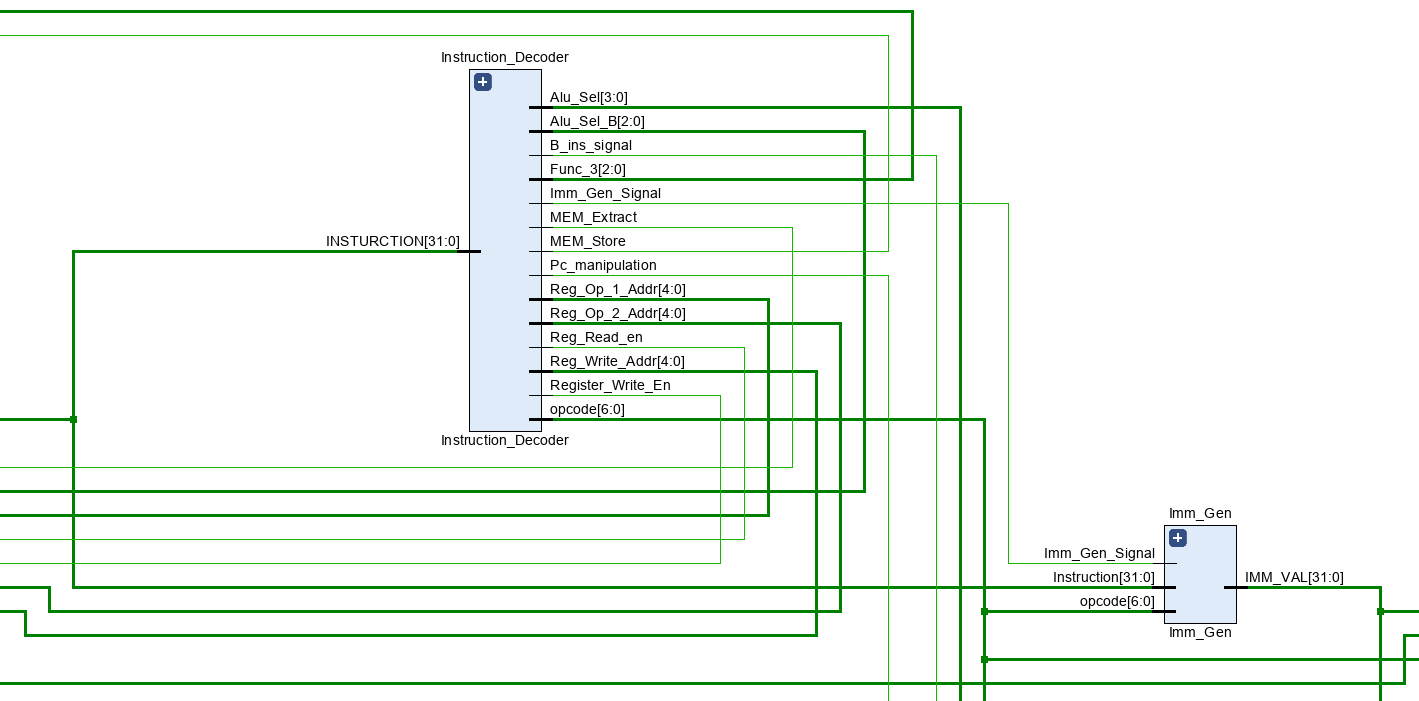
For the eighth assignment I designed a basic single cycle central processor unit based off of RISC-V. The design consists of multiple modules mainly arithmetic logic unit, program counter, instruction decoder, program counter arithmetic unit, register and memory modules, immediate value generator and arithmetic logic unit op selector.

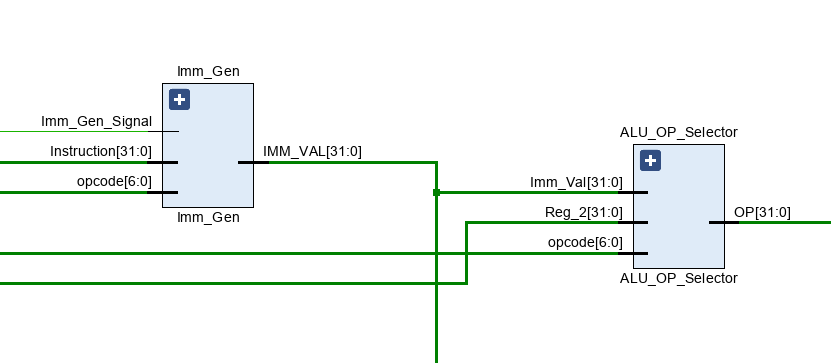
The ***ALU\_OP\_SEL***is connected to two wires one being connected to ***IMM\_GEN*** result and other is to ***Register\_Mod*** second operand. Depending on instruction **opcode** ***ALU\_OP\_SEL*** unit picks the right input for ALU’s second operand as first operand is always comes from register.

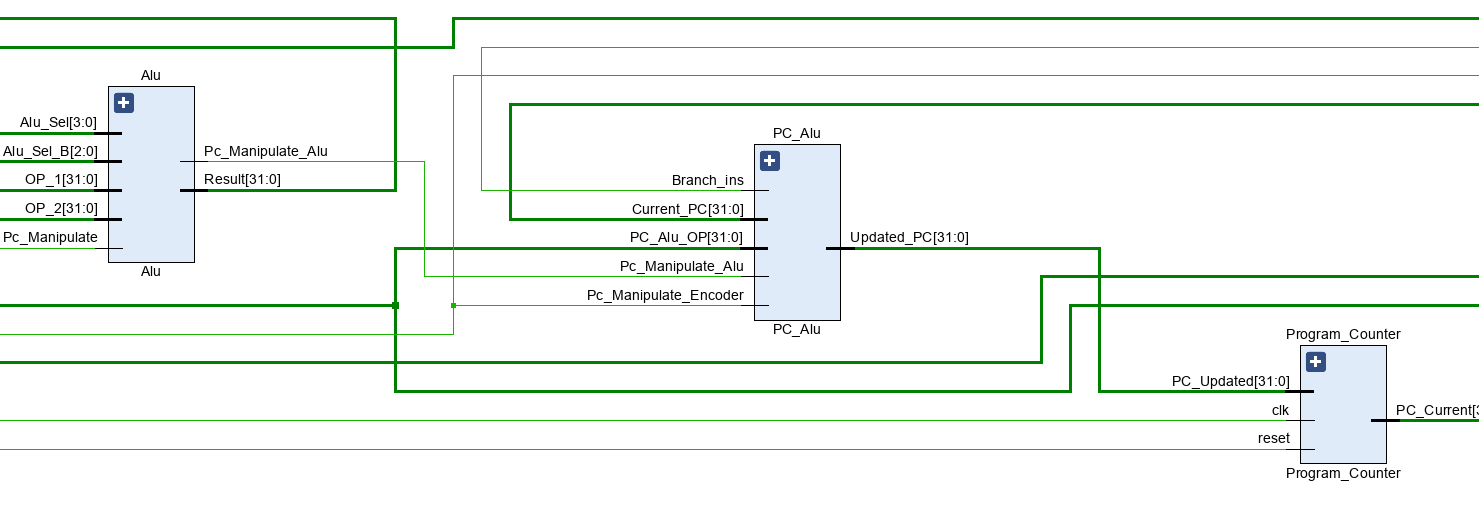
***IMM\_GEN*** module generates the 32-bit output from 32 bit instructions. As different instructions types decode the immediate value in a different way, this module has to take the whole instruction as input.

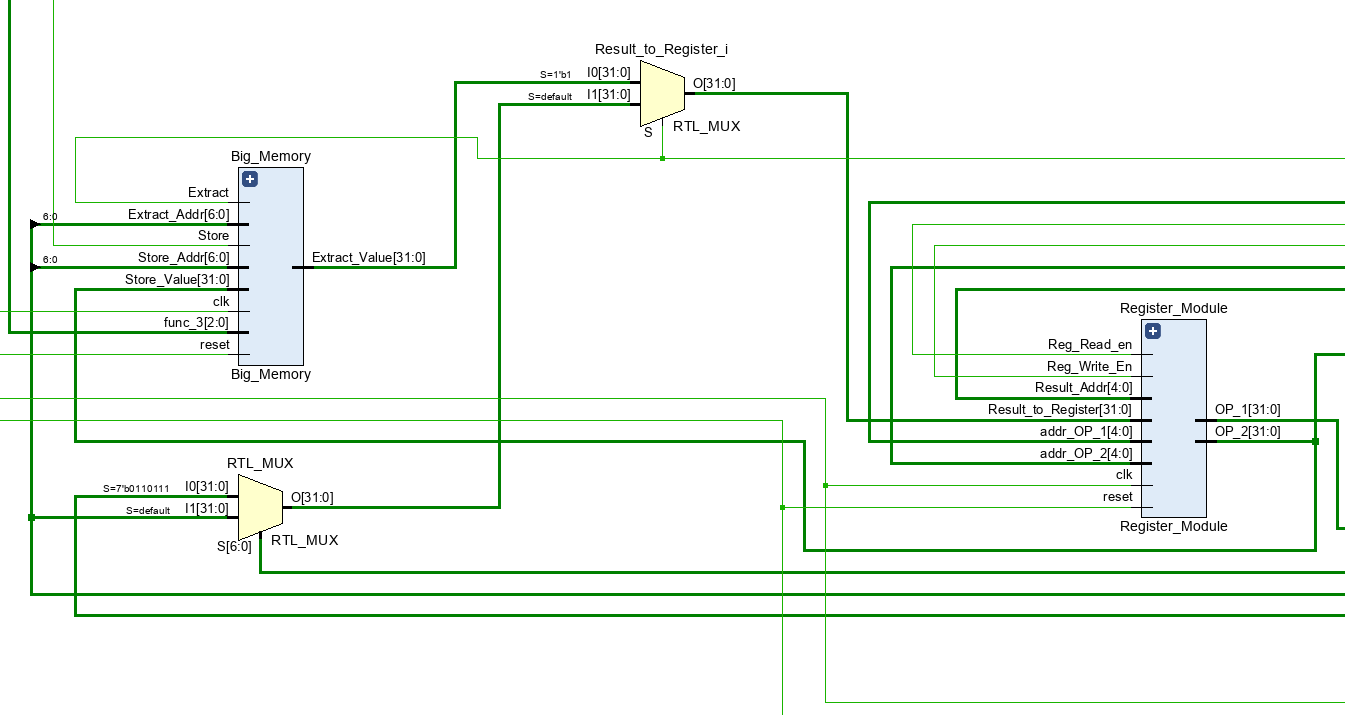
***ALU*** module is the core of this design as it does most of the heavy lifting of the computation work. ALU unit takes two operands and two ALU selection inputs one for branch instruction (**ALU\_SEL\_B**) and all the other types of instruction which needs binary arithmetic (**ALU\_SEL**). The reason behind this separation is because branch instructions allows the opportunity to create one-bit output while other instructions which need arithmetic operations need 32-bit output result. I am aware that I need to pass extra **ALU\_SEL\_B** signal which contains multiple bits as input. The reason behind my preference is totally intuitional as I did not make any calculation to deduce which design methodology is more efficient in terms or speed and area.

The signals such as **ALU\_SEL, ALU\_SEL\_B, opcode, extract\_enable, read\_enable, PC\_maniuplate** are derived from ***Instruction\_Decoder***. Apart from **ALU** it is the most fundamentalmodule in this design. Depending on the specific instruction this module separates different encoded logic arrays such as opcode, func3 and func7. Separation of immediate value could have been adjusted in this module as the purpose of that operation is suitable but my preference was to do that in ***IMM\_GEN*** module. On the next page you can observe these separate modules inputs and outputs visually via RTL schematic produced by **Vivado software**.

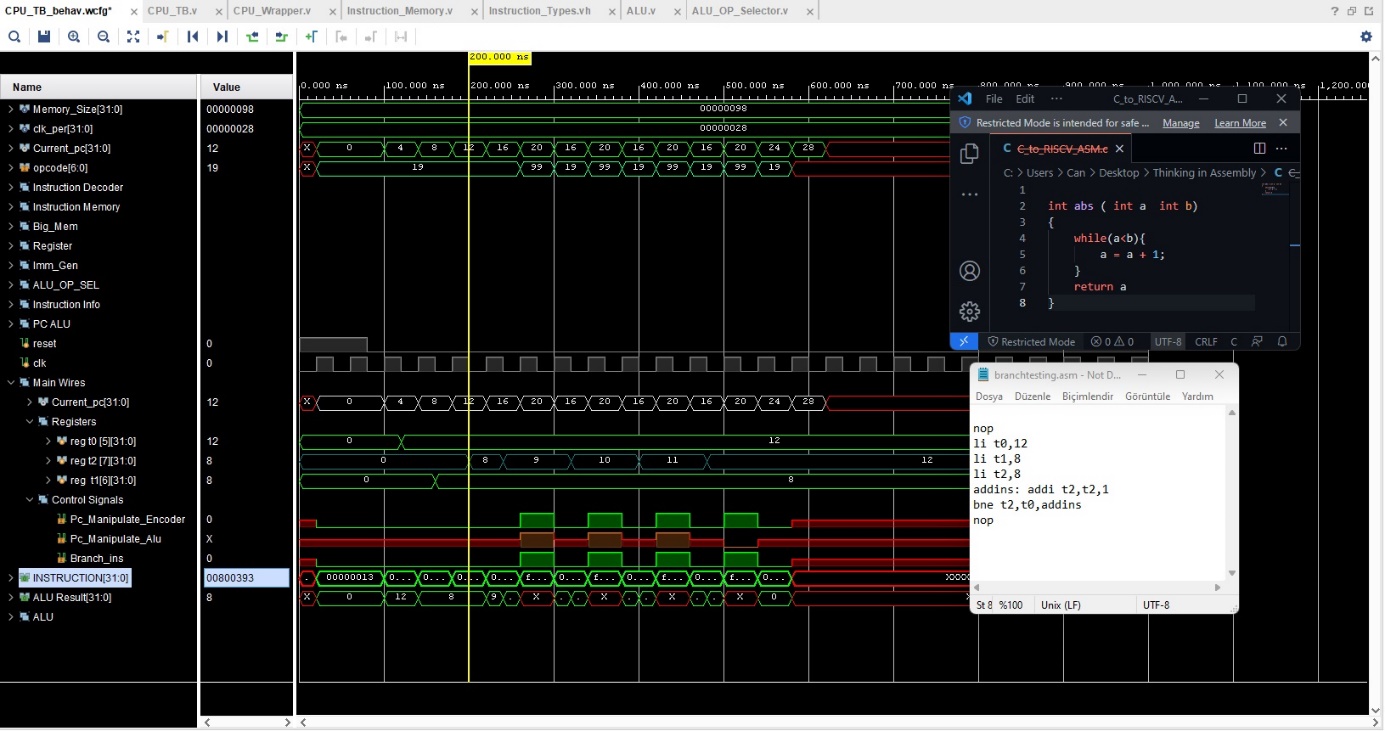




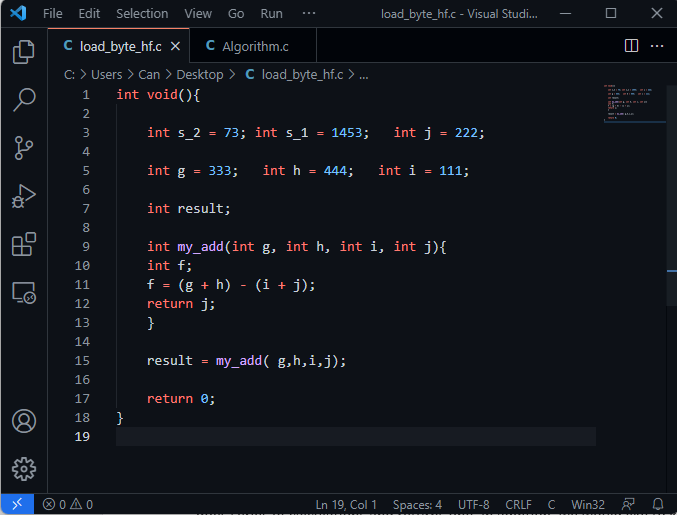




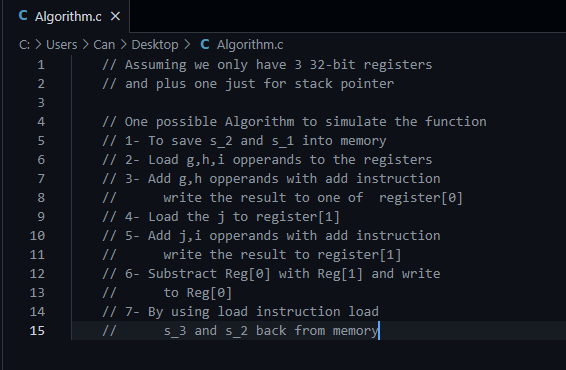
Although to verify and debug such a large project is difficult single cycle RISC-V CPU has some standard functionalities. First of all, we must test with all of the instructions. Attempting to fill **Instruction\_Memory** with all instructions by hand is not efficient way to test the design it is fortunate that my peers avoided me to verify with that methodology. The right way to test is writing a machine code with meaningful intention. Although examples of previous test codes have been shared with me during my internship I found particularly helpful to test with smaller assembly codes as it allowed me to debug my hardware design. First of all, I tested with simple conditional loop which adds one to a register value until it became equal to the target registers value. By doing that test I became sure instructions **li**, **addi** and **bneq** works as they were the instructions to implement addition loop function. RARS open source compiler tool which I have used to transform assembly code to hexadecimal RISC-V instructions.

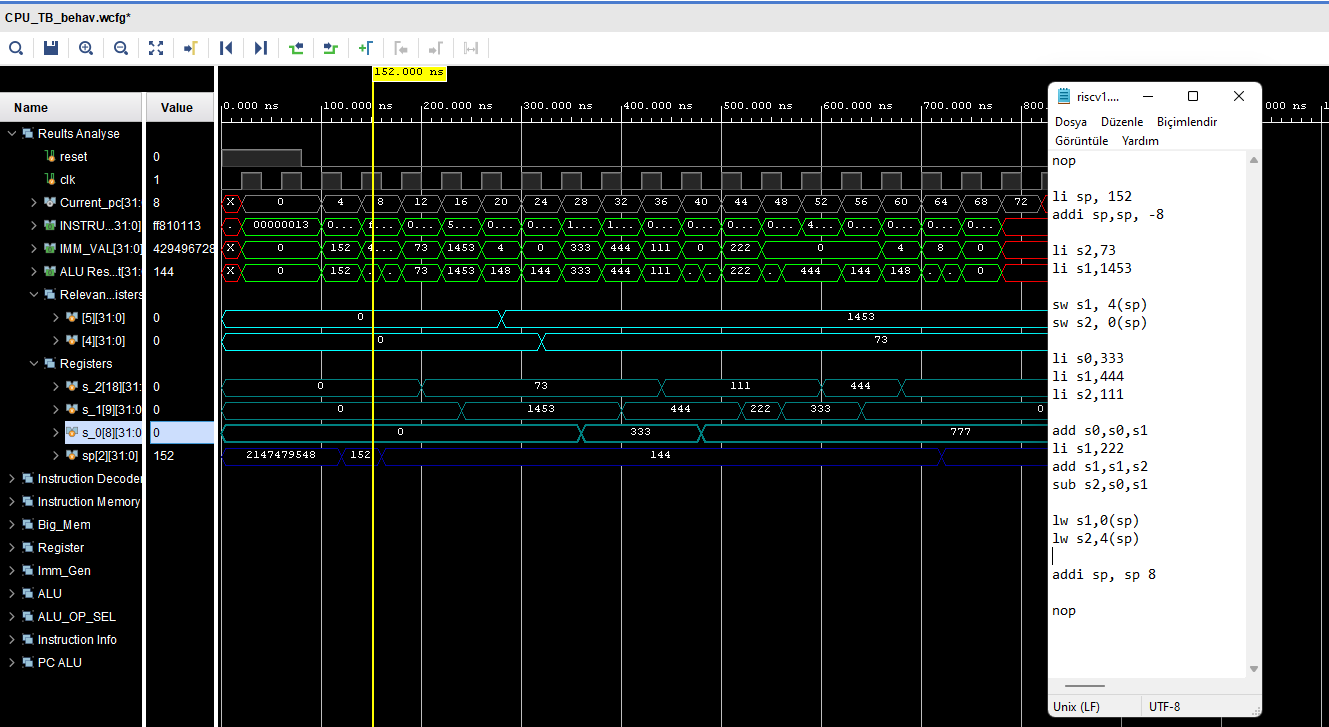


The initial verification test of my CPU focuses on branches and simple R-Type instructions. For the second verification test of the design the focus is shifted towards Store and Load instructions. In order to make sense of the specific verification test, it may be helpful to think about a theoretical scenario. Let’s imagine if we have only three 32- bit register and one reserved for stack pointer. The function we have to implement takes four input parameters, does series of calculations and returns one. In addition, we would like to keep the already existing values inside three registers. The described function is given below.



Although there are many different possible ways to achieve the task given, one possible solution for algorithm is given below.





* **MENTIONED TERMS**

---------------------------------------------------------------------------------------------------------------------------

* **RTL- Modules**

***ALU*:** Arithmetic Logic Unit.

***IMM\_GEN***: Immediate Value Generator.

***Register\_Mod:*** Register Module

***ALU\_OP\_SEL***: Arithmetic logic unit op selector.

***Instruction\_Decoder***: The module which controls other sub modules. Main outputs of this modules are **PC\_maniuplate, ALU\_SEL, ALU\_SEL\_B, opcode and func\_3.**

* **RTL-Signals**

**ALU\_SEL\_B:** Arithmetic logic unit equality check selector for branch instruction.

**ALU\_SEL:** Arithmetic logic unit arithmetic operation selector for branch instruction.

**extract\_enable:** Enable signal to read from memory module.

**read\_enable:** Enable signal to read from register module for second operand. Not essential design preference.

**PC\_maniuplate:** One of the outputs of the instruction decoder which is passed to PC\_Alu. If the instruction allows a possibility to manipulate ALU in a non-generic fashion meaning not increment with four but do a different calculation than it gives a high signal.

**opcode:** The first 7 bits of the 32-bit instruction which describes the type of the instruction.

**Note: Same instruction types may have different opcodes the reasoning behind that is the operations are significantly different example Immediate type load instruction versus Immediate register types.**

* **Tools**

**Vivado software:** The software used to simulate the Verilog code for my design. This software is used to implanted the design on fpga. It can be seen as sophisticated hardware description language compiler also.

**RARS:** Assembly complier which compiles assembly to RISC-V instructions in a hexadecimal or binary format.

* **RISC-V Instructions**

**li**: Load immediate to specified register.

**addi**: add immediate value to specified register.

**bneq**: adjust program counter so that it points to the specified instruction address.