

RV32IC

Verilog Implementation of the RISC-V 32 Integer and Compressed User-level and some Privileged-level instructions.

Mazen Amr Eid 900161021

# Overall Architecture

The processor is composed of the following 4 stages:

## Fetch

This stage is pretty straight forward and simple. The instruction to be executed is fetched via the address provided by the PC.

## Decode

**Decompression** (input[15:0] IF\_Instr\_16, output[31:0] IF\_Dec\_32)

This stage begins with first of all a decompression unit which checks if the fetched instruction is a compressed one or a normal 32 bit instruction. If it is compressed then this unit decompress it to its equivalent 32 bit instruction.

**RegFile** (inputs: clk, clk\_slow, rst, WriteEn, rs1\_rd, rs2, outputs: write\_data, read\_data1, read\_data2)

The instruction then gives its register 1, register 2 (if any) addresses to the register file so their value can be read and moved to the ID\_EX pipeline register. It is important to note that the register file’s rs1 and rd are both in the same port because writing in and reading from the register file in my design can never happen at the same time. Therefore, according to the stage I am in, I have a MUX deciding of choosing either the RD or RS1.

**ImmGen** (input: IF\_ID\_Instruction, output: ImmGen\_out)

This module generates the appropriate immediate from the instruction and passes it to the ID\_EX pipeline register.

I finally have the **ControlUnit**(inputs: instruction, opcode; outputs: Branch, Jump, MemRead, MemToReg, ALUOp, MemWrite, ALUSrc, RegWrite, Mode, Lui, Auipc, ECall, Ebreak, MRET, CSRWrite, CSRSet, CSRClear, CSRI)

***Jump***: Selection line of a MUX to the data port of the Register file that chooses the next instruction’s address so it can be strored in the RegFile (if it is 1).

***MemToReg***: Selection line of the Write Back Mux choosing between the ALU result or the value coming from the memory.

***ALUOp***: 2 bits going to the ALUControl unit to select the operation to be executed in the ALU.

***MemWrite***: If high, the memory writes its coming data in its selected address.

***ALUSrc***: Selection line choosing between the value of Register 1 coming out of the register file and the immediate generated.

***RegWrite***: If high, the register file writes its coming data in its selected address.

***Mode***: Selects how many bits to be read or written in the Memory (one word, half word or one byte).

***Lui***: Selection line to a MUX that chooses either the PC/jal value or the LUI’s Immediate to be written in the register file.

***Auipc:*** Selection line for a MUX choosing between the ID\_EX\_PC or the AUIPC immediate entering the ALU.

***ECall***: If high, this means that the current instruction is an ECall. This is used in the Interrupt\_Detector.

***Ebreak***: Same idea than ECall.

***MRET***: Selection line of a Mux, if high, then the Mux chooses the address coming from the CSR MEPC + 4 (assuming that there will not be any compressed instruction causing an interrupt which is false of course).

***CSRWrite***: If high, then the CSR Register File will write its data in its selected address.

***CSRSet***: A control signal to the CSR Register File so we write in the selected address the old value of the CSR register ORed with the entering data.

***CSRClear***: A control signal to the CSR Register File so we write in the selected address the old value of the CSR register ANDed with NOT the entering data.

***CSRI***: Selection line of a MUX choosing either RS1 from the CSR instruction or its UEIMM (which is the same field but 0 extended).

## Execute

**ForwardUnit**(inputs: ID\_EX\_Rs1, ID\_EX\_Rs2, EX\_MEM\_RegWrite, EX\_MEM\_Rd, MEM\_WB \_Rd, outputs : ForwardA, ForwardB)

Forwards the EX\_MEM\_Rd or the MEM\_WB \_Rd to ID\_EX\_RS1 or ID\_EX\_RS2 if the forwarding conditions are met.

**ALU**(inputs: selection, a, b; outputs: result, zeroFlag, SignFlag, CarryFlag, OverflowFlag)

This is the stage’s heart. It executes the Arithmetic or Logic Operation using the operands and the operation selection.

**BranchUnit**(inputs: branch, zeroFlag, SignFlag, CarryFlag, OverflowFlag, func3; outputs: branchTaken)

Detects which branch instruction is currently executed (beq, bne, bgt, blt, bgtu, or bltu) and outputs a high branchTaken flag if the conditions of the branching are met.

**ALUControl**(inputs: ALUOp, func3, func7; output: select)

Selects the appropriate operation to be executed by the ALU according to the func3 and func7 of the instruction.

This stage is also where the Branch and Jump addresses to be jumped to are computed and forwarded to the PC.

**CSRRegFile** (inputs: clk, clk\_slow, rst, pc, nmi, ecall, ebreak, int, en\_inter, en\_ecall, en\_int, en\_tmr, addr, CSR\_Write, CSR\_Write\_Data, Set, Clear; outputs: CSR\_Read\_Data, interF, interAddr)

A decision I took, is to put everything related to interrupts in this module. Either this decision is a good or bad design, but it helped to concatenate everything related to interrupts and CSR instructions in one module. This module updates the CSR files when needed. Furthermore, it has the following modules:

**Interrupt\_Detector(**inputs: nmi, ecall, ebreak, int, tmr, en\_inter, en\_ecall, en\_int, en\_tmr; outputs: interFlag, interSel)

This only sets interFlag to high when there is an interrupt and outputs which interrupt is this using the interSel which enters in the next module:

**InterruptAddressGenerator** (inputs: interrupt, interSel, intNum; output: addr)

Generates the address of the interrupt handler according to the inputted interrupt.

## Memory Access

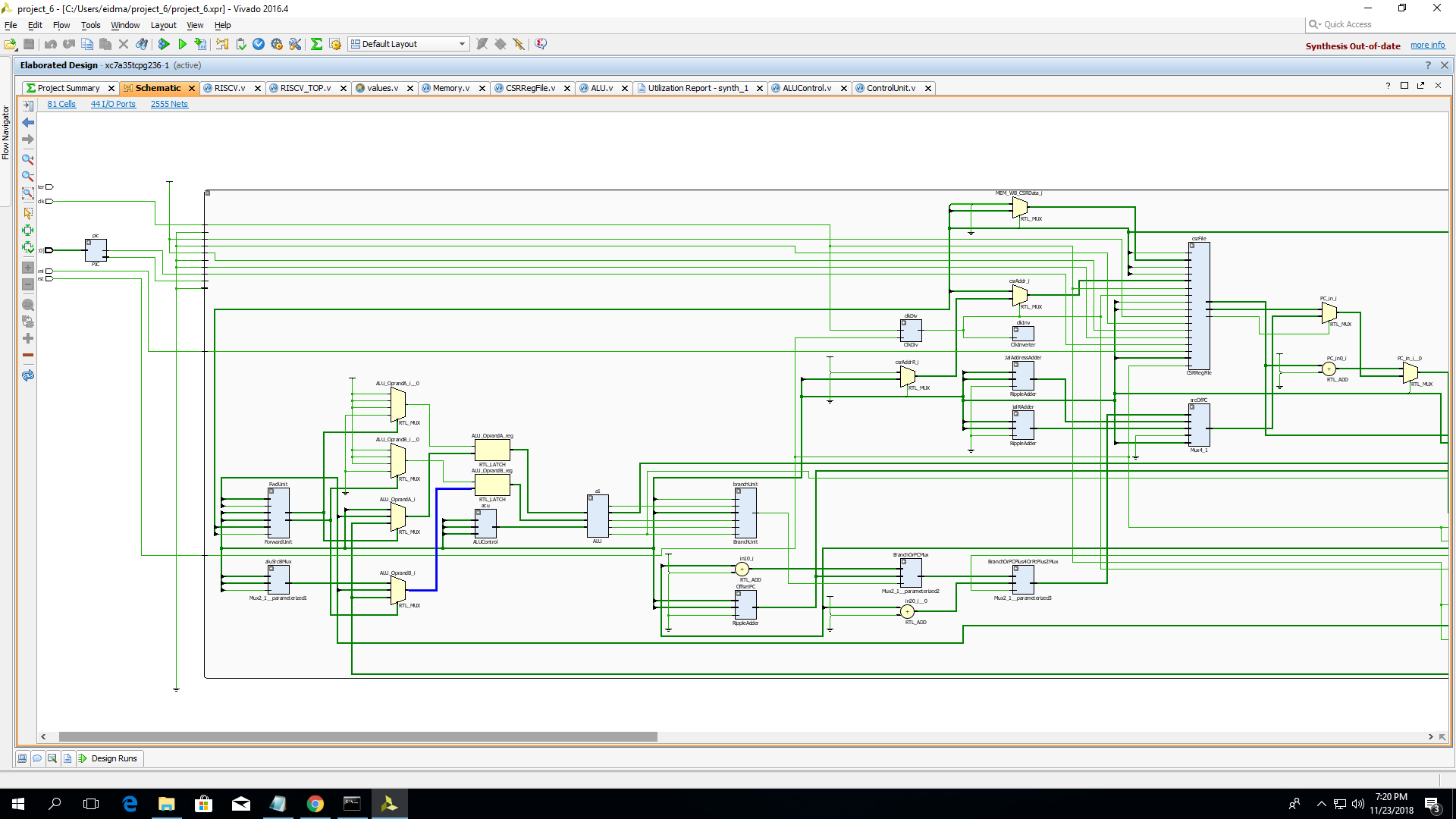
This stage is where data is read or written into the memory. The number of bits to be written or read changes according to the mode signal (which is generated by the control unit).

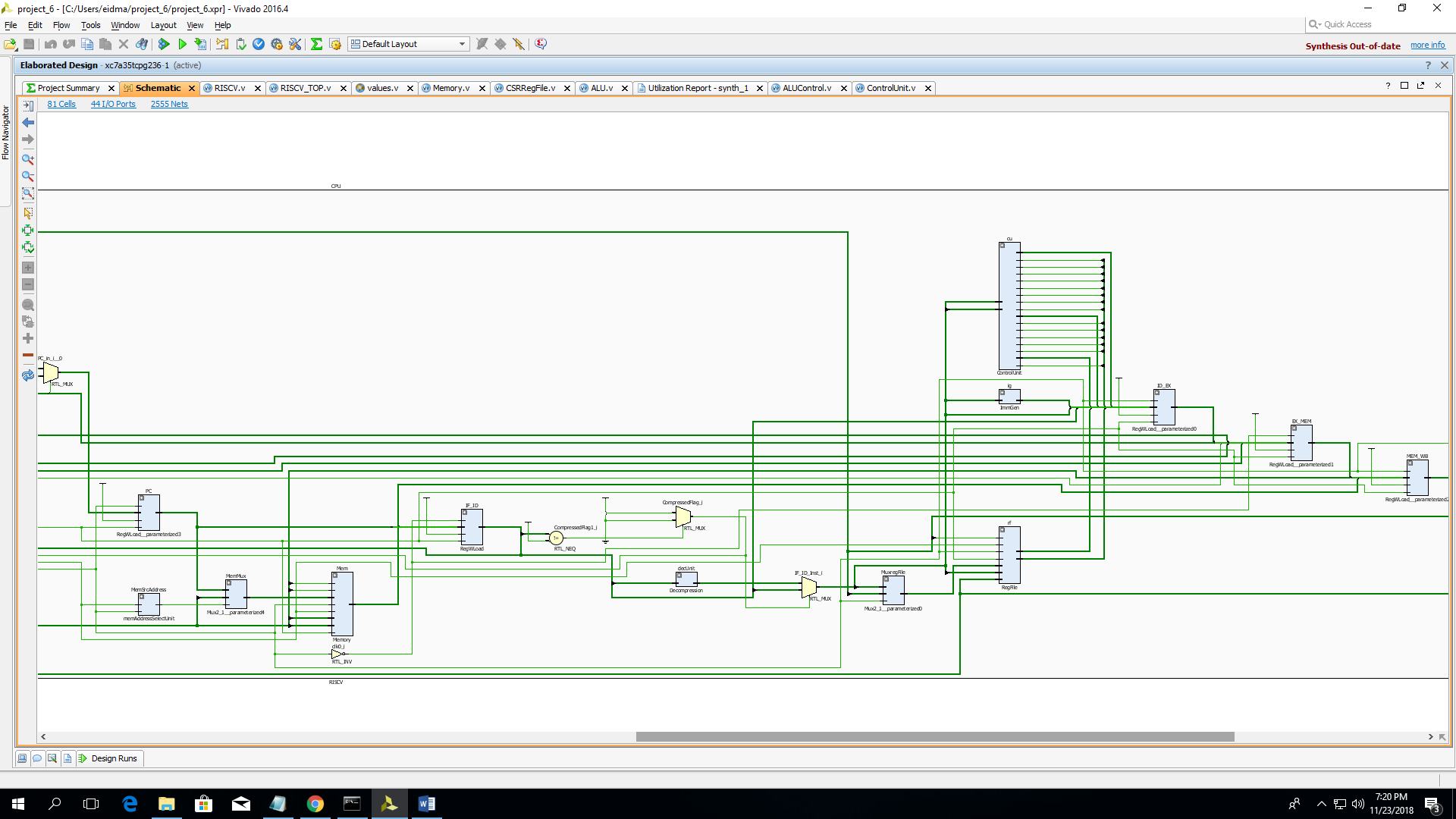
## Write Back

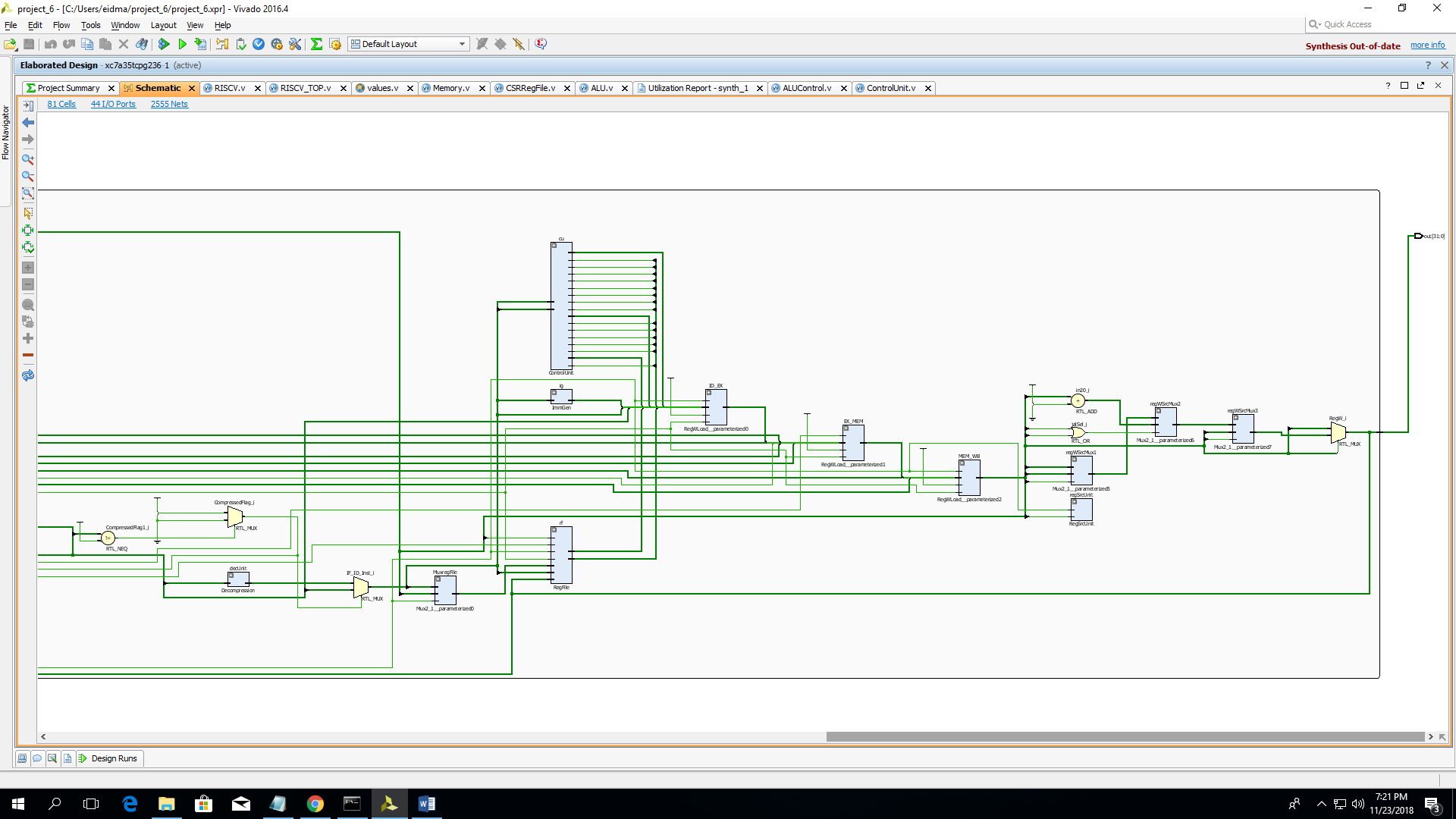
This stage consists only of a couple of MUXes to choose what data to write in the Register File.

# CPU Diagram

This is a detailed diagram to my processor:







# Tests

I tested every instruction separately to be sure that it works as predicted. In addition to that, I was using the 2 assembly tests provided for us on BlackBoard throughout the whole development process to ensure that everything in the cpu works fine and I was adding to these BB tests the new instructions that need to be tested.

## Run your own tests

If you want to test your own code: convert your code to hex, then go to the file “TestBench/testCase.txt” in the project directory and paste your code into it. **Please make sure to change the path of the testCase.txt file in the $readmemh() function in the Memory module according to your own path.**

# Area and propagation delay

|  |  |  |
| --- | --- | --- |
| Module | LUTs | Setup Time (ns) |
| ALU | 568 | 6.182 |
| Decompression Unit | 100 | 3.984 |
| Register File | 613 | 3.256 |
| Control Unit | 20 | 3.756 |
| CSRRegFile | 9 | 3.756 |
| PIC | 5 | 3.756 |

Memory’s delay is not included because it takes too much time to synthesize.

# Challenges and Limitations

* The large scale of the project. In fact, this project needs a lot of time and dedication to be done.
* The CPU needs further testing as there has not been enough time for lots of testing. Therefore, if given more time, we would test more the processor and check for corner cases.
* The CPU takes too much time synthesizing, this is why I was not able to do so.

# Cloudv bugs

* wire reg is accepted as declaration in cloud v’s testbenches.
* I didn’t use Cloud V throughout the whole project because of several reasons: One of them is that it doesn’t accept me to include my values.v file which contains all the defined constants and I have tried all different
* possible syntaxes (`include “values.v”, `include “src/values.v” or even `include “../src/values.v”) but unfortunately all of them didn’t work for me.

For log file:

https://github.com/djzenma/RV32IC-CPU