DESIGN OF A SINGLE-CYCLE RISC-V PROCESSOR (RV32I)

Alaa Roshdy Sakran

12/2/2025

TABLE OF CONTENTS

INTRODUCTION

- This document provides a comprehensive guide to the design and implementation of a single-cycle RISC-V processor using the RV32I architecture. This project aims to demonstrate the fundamentals of processor design, including the data path, control path, and instruction set implementation. The RISC-V architecture was chosen due to its simplicity and extensibility.

- The single-cycle processor executes one instruction per clock cycle, ensuring straightforward control logic and minimal complexity. This design supports a subset of the RV32I instruction set, including all R-type and I-type instructions, as well as essential branch, load/store, and jump instructions.

FEATURES AND SUPPORTED INSTRUCTIONS

- Supports all R-type and I-type instructions
- Supports branch instructions (`beq`, `blt`)
- Supports jump instructions (`jal`, `jalr`)
- Supports load/store instructions (`lw`, `sw`)
- Includes `lui`, `slt`, and `slti`
- Any other instruction could be implemented using the above instructions so for simplicity in design purpose the instruction set was limited to the above

ARCHITECTURE OVERVIEW

Since the choice of design is a single-cycle processor the overview is pretty simple.

The design consists of four main blocks:

Instruction Memory – Register File – ALU (Arithmetic Logic Unit) – Data Memory

Design Flow:

1-**Instruction Fetch:** Instruction is fetched from the first block by the PC .

2-**Instruction Decode:** The fetched instruction is decoded into the Register File, accessing the registers specified by the instruction .

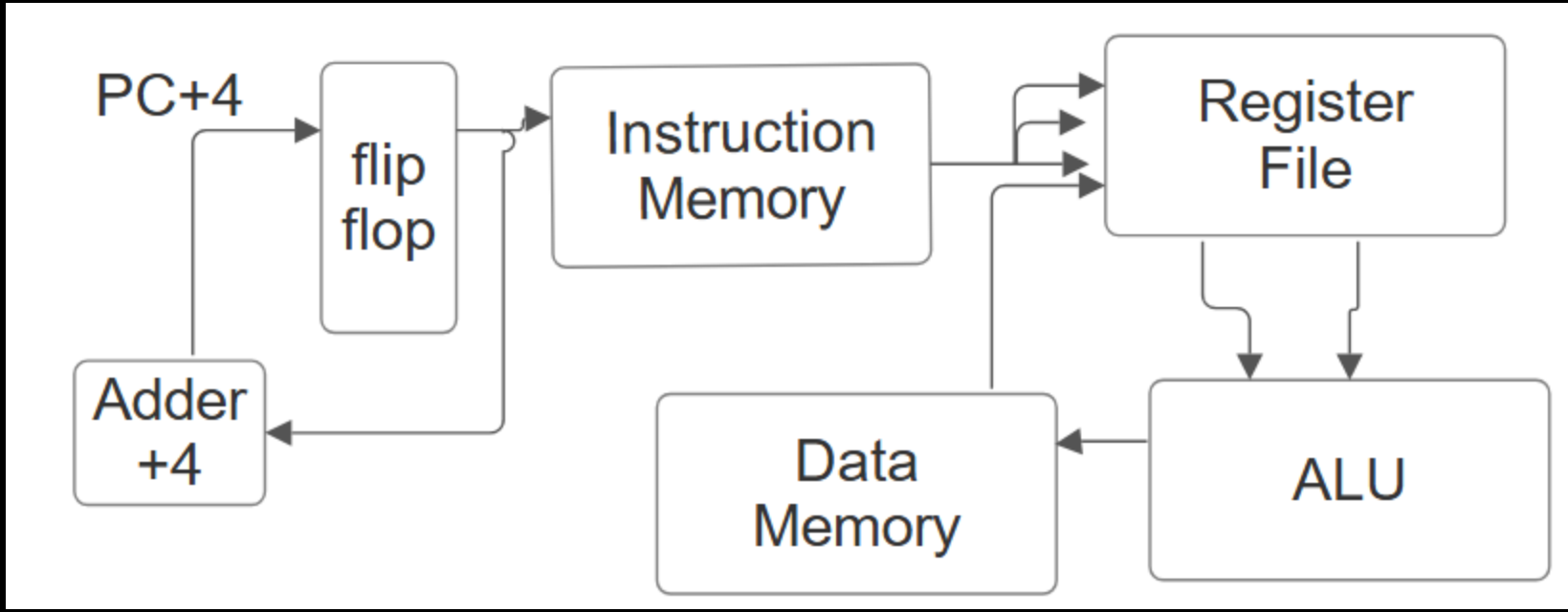
3-**ALU Operation:** The decoded instruction and its operands are passed to the ALU, which performs the specified operation (such as addition, subtraction, bitwise operations, etc.).

4-**Memory Access:** If the instruction involves memory access (e.g., load or store), the operation is carried out using the Data Memory.

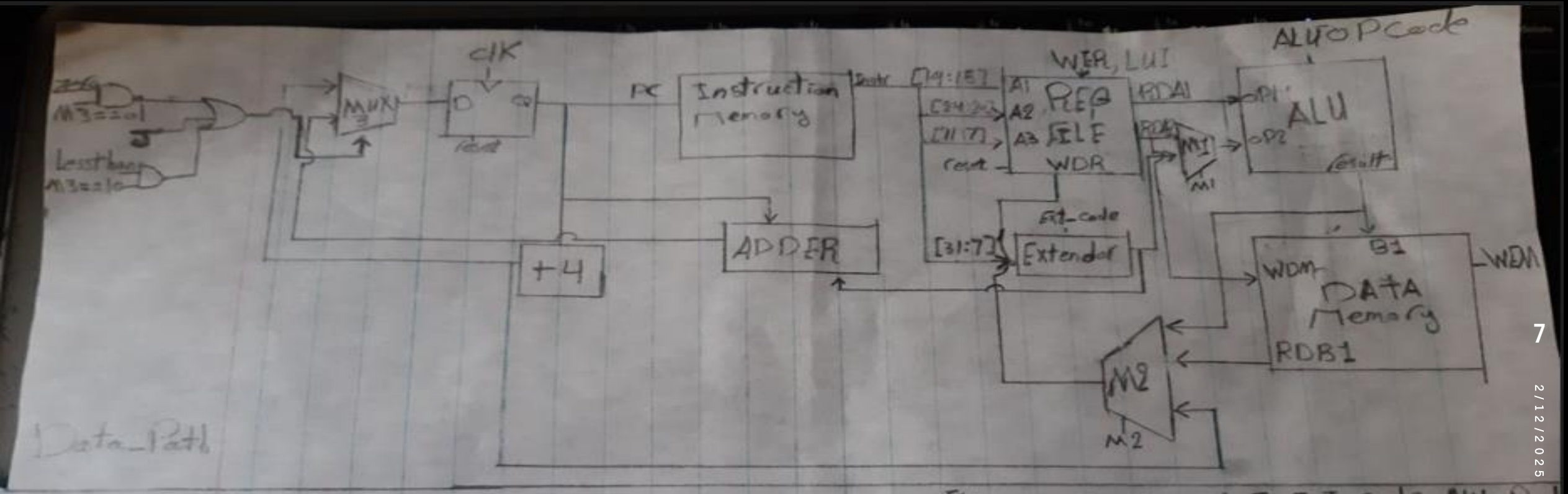
Key Points:

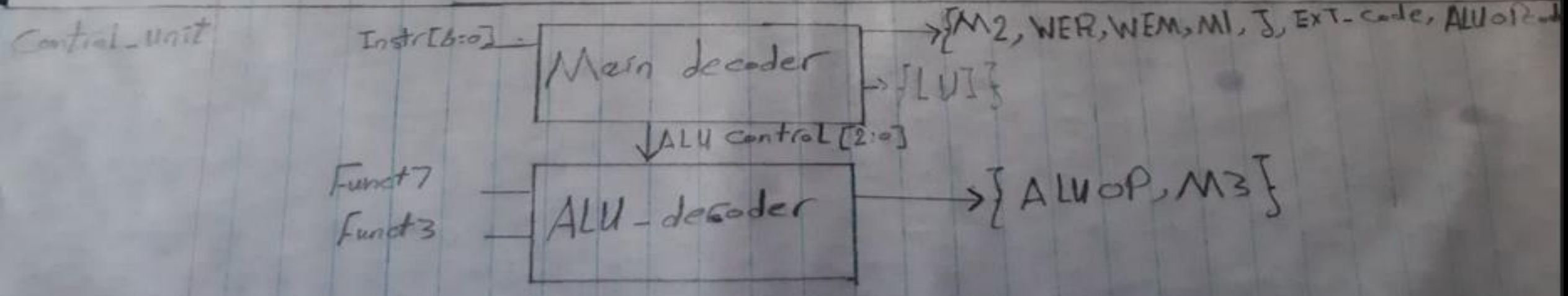
- Single-Cycle Design:
  - This is a single-cycle design, meaning each instruction is completed in one clock cycle.
  - As a result, the Instruction Memory is separate from the Data Memory and is treated as a separate ROM. While this separation might be impractical for real-world applications, it ensures that each operation completes within a single cycle.

ARCHITECTURE OVERVIEW



DETAILED DESIGN: HAND-DRAWN FULL DESIGN





DETAILED DESIGN

- I'll go through each block on paper and explain It before depicting the blocks generated by the simulator
- First we have the **Flip-Flop** to the **PC**:
- At each clock cycle the PC should be incremented by 4 as we all know to fetch the next instruction. But that's not always the case sometimes **Jump** or **Branch** causes the **PC** to not be incremented by 4 instead leaps to a new value therefore we have this **MUX3** which decides whether the **PC** should be incremented, jump to a new address or branch
- The output of the **MUX3** goes through **D** in **Flip-Flop**.

DETAILED DESIGN

EXPLAINING THE DATA PATH BLOCKS

- Next is the **Instruction Memory**:
- As mentioned earlier this is treated as a **ROM** and the program is loaded onto it before execution in my code I would load it in test bench using initial block with the program it has one input which is the address of instruction and one output which is the 32-bit **instruction**
- **REG File**: this block has 7 input and two outputs:
- **Reset**: resets all 32 regs to zero, **WER (Write Enable Reg), clk** (since writing occurs at positive edge), 5 bit **A1,A2,A3** these give the numerical value for the register to be accessed (eg. If **A1** is 5 then reg **x05** is read),32 bit outputs **RDA1,RDA2** which read the registers, input 32 bit long **WDR** (Write Data Register) which writes to **reg[A3],** and lastly an input **LUI** which is a flag to the **LUI** instruction(makes it write only the 20 most significant bits).

DETAILED DESIGN

EXPLAINING THE DATA PATH BLOCKS

- Extendor:
- This one has the input Instruction[31:7] and depending on the EXT_OPCODE it would decode the immediate value in the instruction and then sign extend it to make it 32 bits suitable for any later operation.
- ALU:
- This one is simple it's the ALU that does almost all the operations of the micro-processor
- Like addition, logical, shifting and more. It has two input it could operate on the outputs of the REG FILE or the immediate from the Extendor if it's I-type instruction for example therefore MUX1 comes into play here to decide which operands to be taken.

DETAILED DESIGN

EXPLAINING THE DATA PATH BLOCKS

- **Data Memory:**
- It's where all the data for variables are stored (not the program). It has a 5 bit input **B1** which is the output of the **ALU** and it has the input **WEM (Write Enable Memory)** and **WDM (Write Data Memory), clk,** and the output **RDB1**.
- It doesn't have a reset so reading a memory location without initialization would lead to undefined readings
- Finally in Data Path there is the block **Adder** in middle this one adds the immediate from **Extendor** with **PC** to prepare the new **PC** with instructions like **Jal, Jalr, beq, blt**

EXPLAINING THE CONTROL UNIT BLOCKS

- Next Is the control unit segment:
- This segment is responsible for decoding the **instruction** and giving out **control signals** for the data path such **ALU_OPCODE** or **MUX select** signals to route the signals or **EXT_OPCODE** to choose which immediate decoding method to be executed.
- **Main DEC:**
- This main decoder block takes the 7 bit **OPCODE** and determines the type of instruction and gives output control signals for most **muxes** and **Write Enables** and **ALUCODE** which is passed to the next Block (not the **ALU**).

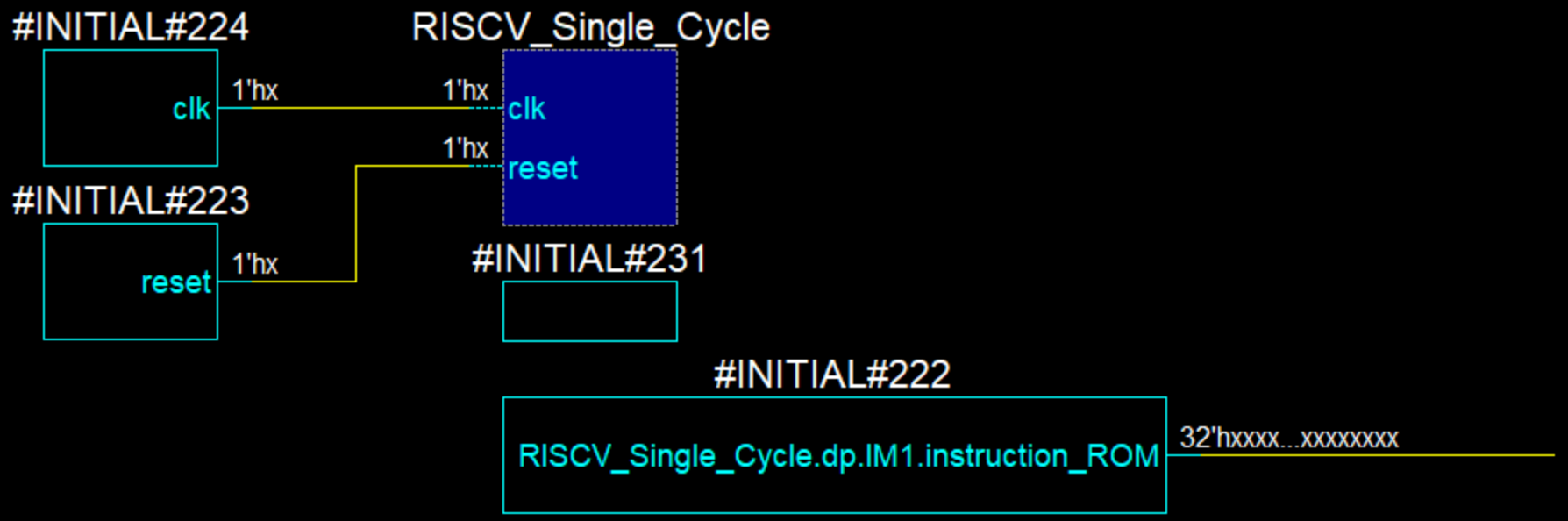
DETAILED DESIGN

EXPLAINING THE CONTROL UNIT BLOCKS

- ALU DECODER:
- This one takes the input ALU CODE from the main decoder and FUNCT7 and FUNCT3 from the instruction which will be mentioned later and gives the opcode for ALU and MUX3 select signal.
- Now I've explained the objective of each block and how the micro-processor works next I'll provide images for the blocks generated by the simulator and their connections.

DETAILED DESIGN

Next I'll show diagrams for the block of the processor (datapath and control unit) generated by the simulator:



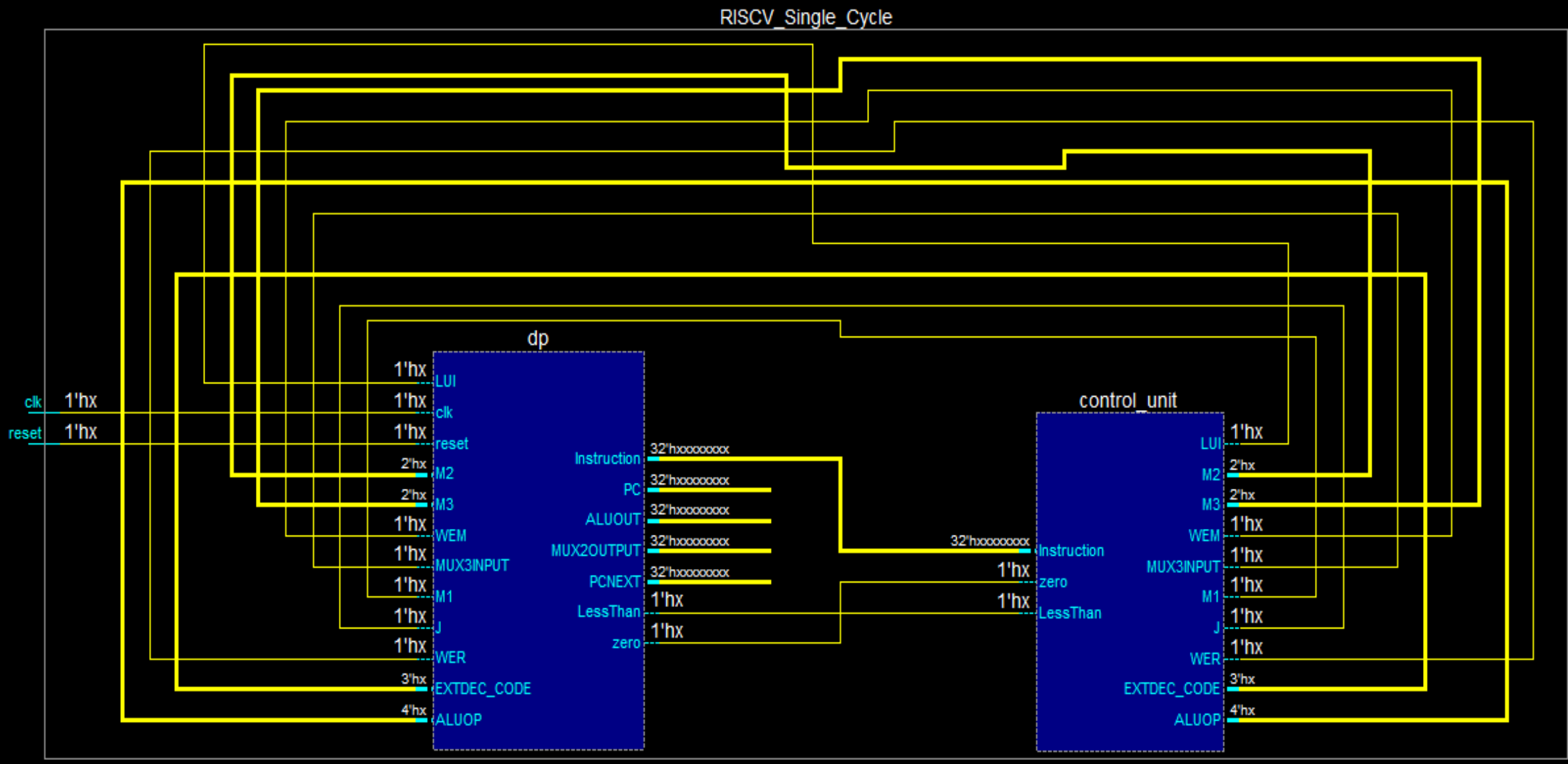
DETAILED DESIGN

From previous image we can tell that the processor main module is being called with two argument signals which are Reset, clk while the Instruction memory is loaded with the program externally.

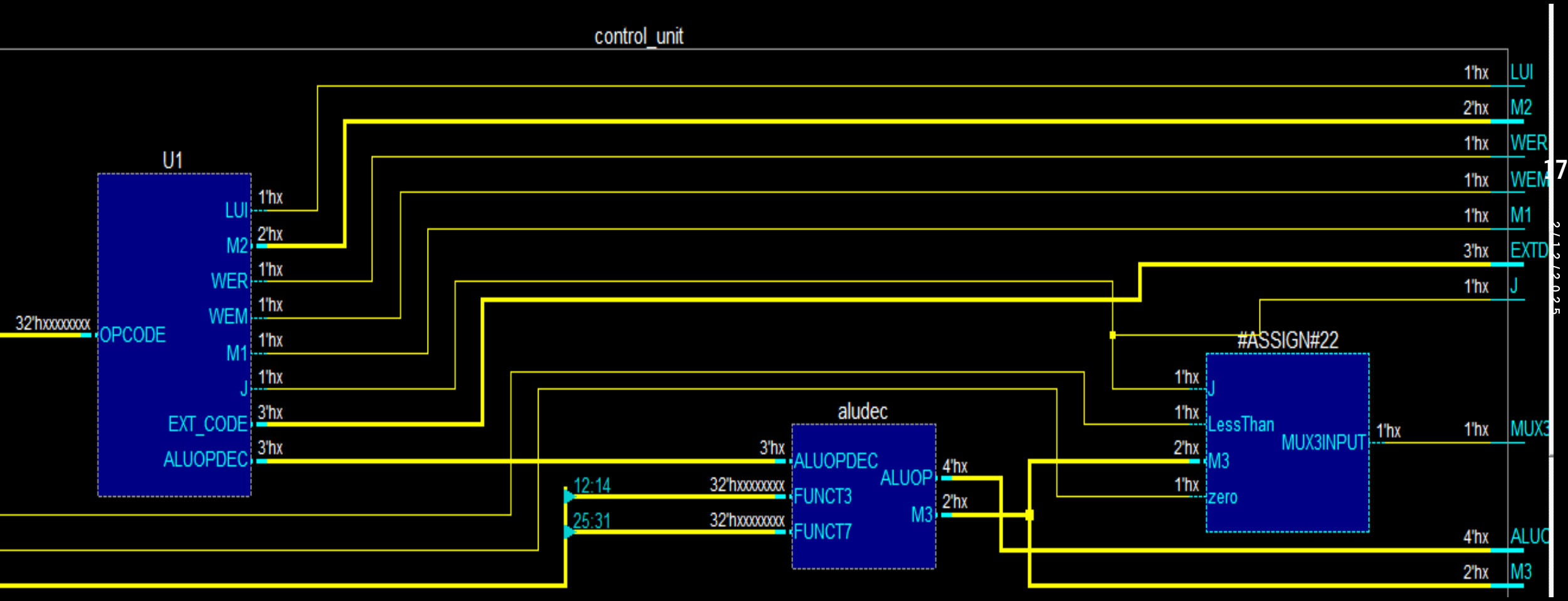
Taking the diagram one step closer we see in the next image that the micro-architecture consists of two main modules (datapath and control unit) with the control signal flowing from control unit to datapath to operate It and some outputs from DataPath to control unit as well.

DETAILED DESIGN

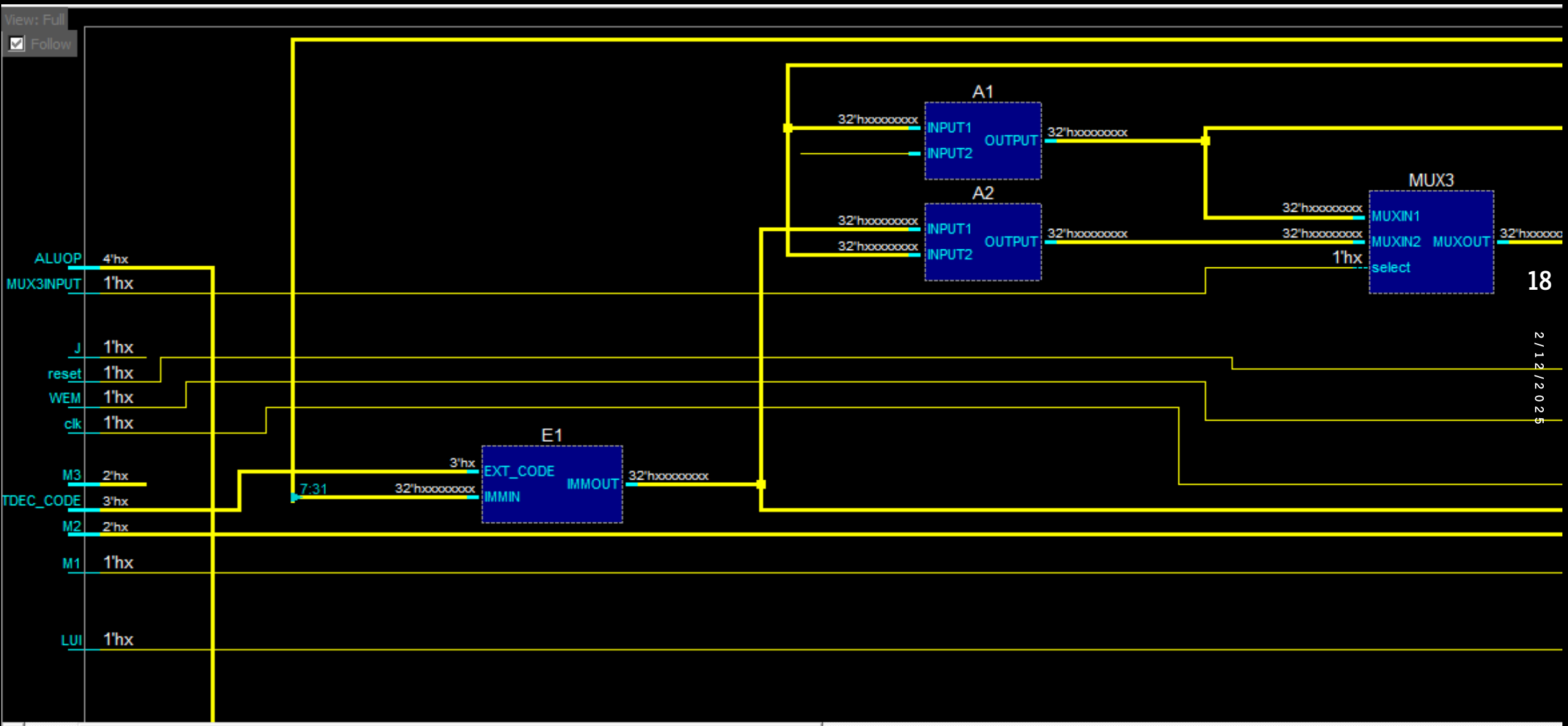
MAIN MODULE



DETAILED DESIGN
CONTORL UNIT DIAGRAM



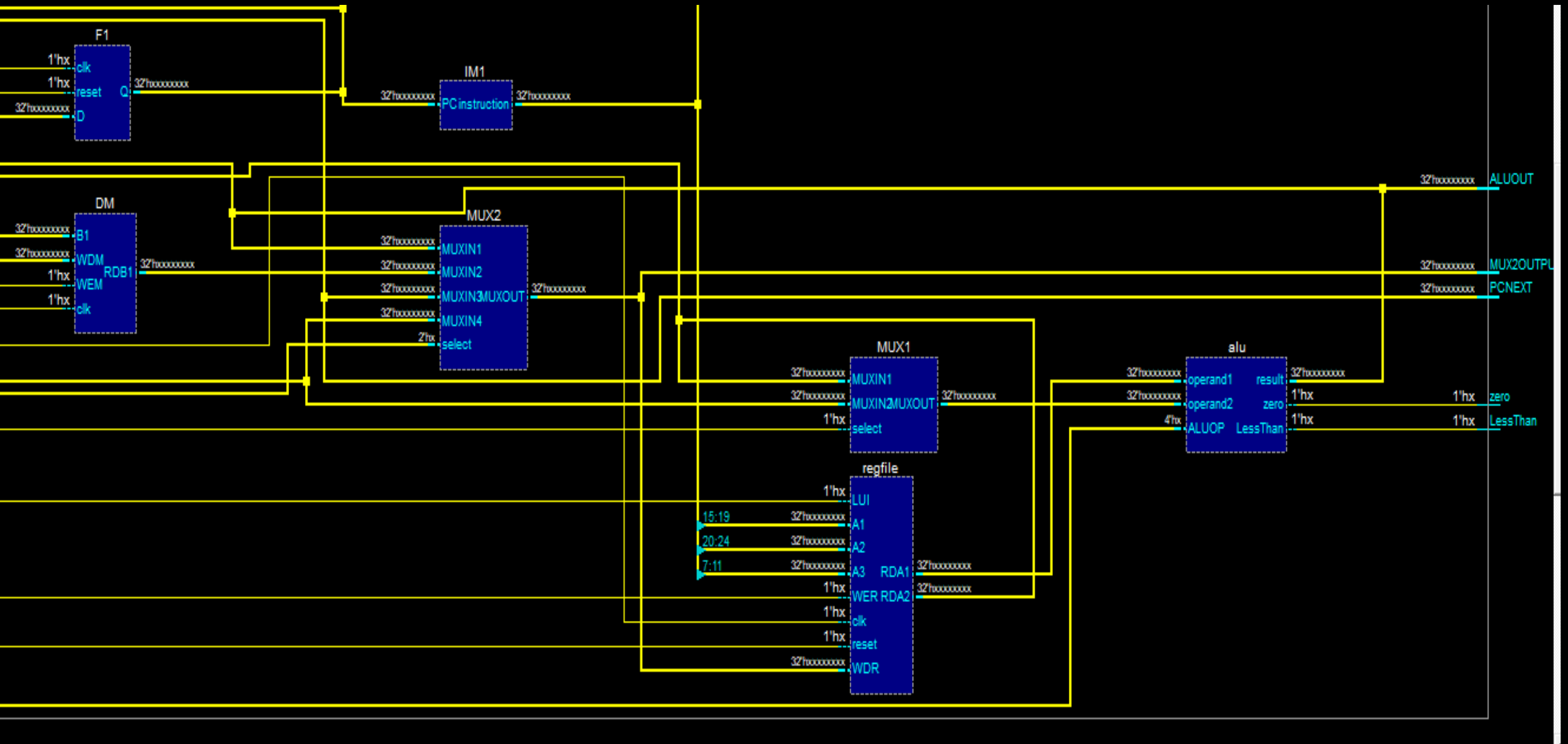
DATA PATH



2/12/2025

DETAILED DESIGN

DATA PATH

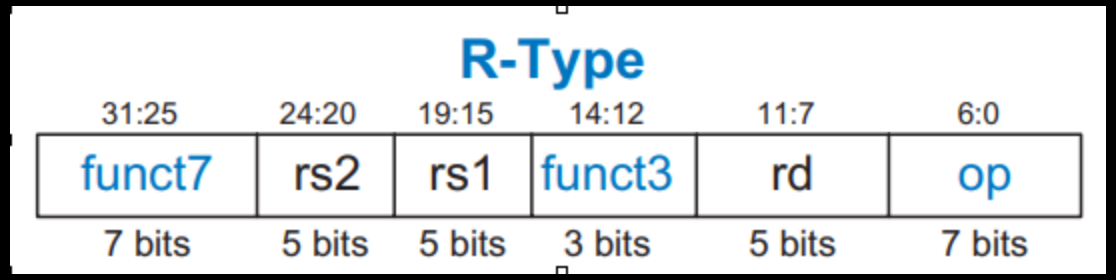


DETAILED DESIGN
DATA PATH

- We can see that the data path blocks are the same as on Paper. I will name the blocks by their instantiation names to clear the confusion:
- E1: Extendor
- F1: Flip-Flop
- DM: Data Memory
- IM1: Instruction Memory
- U1: Main decoder
- aludec: ALU decoder to generate ALUOP.
- A1,A2 are the AND Gates on paper.

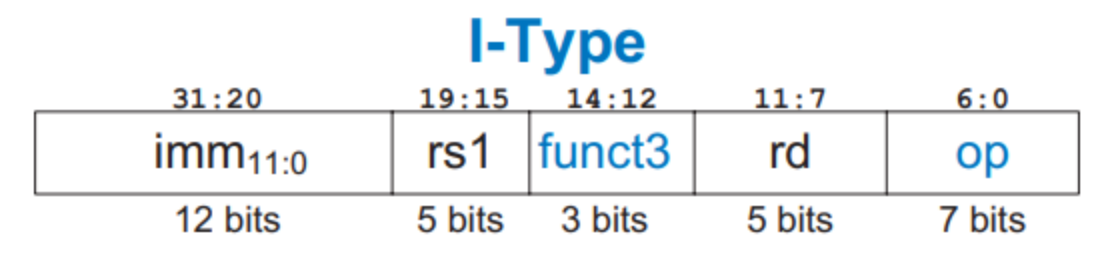
INSTRUCTION SET

- The Designed micro-processor supports a lot of instruction types including:



- This type operates on two source registers and loads the result into destination register
- Operation in **ALU** is defines by **funct3**, **funct7**
- Example: **add t4,s0,s1**
- All **R-type** operations are supported i.e. **add, sub, AND, OR ,XOR ,SLL, SRL, SRA, SLT**

INSTRUCTION SET



- This type operates on a source register and 12 bit Immediate data and places the result in destination register.
- Example: **andi t1,t2,0xFA0**
- Same operation for **R-type** are supported for **I-type** plus the **lw** instruction.

INSTRUCTION SET

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- For these types **S-type** is only used in this design with **sw**. Any other store instruction is not supported such as **sb**.
- **B-type** supports two branch instruction **beq, blt**. **Funct3** defines which one It Is.
- For **B-type, J-type** the **LSB** is always zero therefore **imm[0]** isn't in the instruction.

31:12	11:7	6:0	
$imm_{31:12}$	rd	op	U-Type
$imm_{20,10:1,11,19:12}$	rd	op	J-Type
20 bits	5 bits	7 bits	

- **U-type** is used with **lui** instruction, while **J-type** is used with **Jal**, **Jalr**

EXAMPLE PROGRAM

- I've coded multiple programs to confirm the correctness of the design and the executes of the correct operations, I'll be showcasing a simple program that checks for most instructions.
- The program was coded in assembly and then decoded the instructions to machine language and imported it in the instruction memory.
- The program will be in another file but here I'll display the output for both the design and the actual outputs from a RISC-V compiler.
- The program modifies the content of registers x01 to x18

EXAMPLE PROGRAMS

- Output of the RISC-V compiler:

Name	Number	Value
zero	0	0x00000000
ra	1	0x0000000a
sp	2	0x00000014
gp	3	0x0000001e
tp	4	0x00000028
t0	5	0x00000021
t1	6	0x00000014
t2	7	0x00a00000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x0000001e
a1	11	0x0000001e
a2	12	0x00000001
a3	13	0x0000000f
a4	14	0x00000050
a5	15	0x00000002
a6	16	0x000000ff
a7	17	0x00000005
s2	18	0x10010000

EXAMPLE PROGRAM

- Output of the designed micro-processor:

```
#  REGs from 0 to 18 are: x00 is 00000000 x01 is 0000000a x02 is 00000014 x03 is 0000001e
#  x04 is 00000028 x05 is 00000021 x06 is 00000014 x07 is 00a00000 x08 is 00000000
#  x09 is 00000000 x10 is 0000001e x11 is 0000001e x12 is 00000001
#  x13 is 0000000f x14 is 00000050 x15 is 00000002
#  x16 is 000000ff x17 is 00000005 x18 is 10010000
```

- As we can see results match. Of course I've used other programs as well and they worked well.
- This exact test bench will be attached to the project files along with the program in both Assembly and Machine language.

CONCLUSION

This project has successfully demonstrated the design and implementation of a single-cycle RISC-V processor using the RV32I architecture.

By focusing on the core set of instructions and ensuring the efficient execution of each, the processor is capable of handling a variety of computational tasks.

- Key achievements include:
- **Comprehensive Instruction Support:** The processor supports all **R-type** and **I-type** instructions, along with essential **branch**, **load/store**, and **jump** instructions. This ensures a wide range of functionalities, from arithmetic operations to memory access.
- **Single-Cycle Execution:** Each instruction completes in a single clock cycle, simplifying control logic and minimizing latency. This design choice, while making the processor simpler, also highlights the trade-offs between speed and complexity.
- **Detailed Data Path and Control Path Design:** The processor's architecture, including the data path and control path, has been meticulously designed and documented. This ensures clear understanding and easy maintenance or modification.
- **Functional Example Programs:** The processor has been tested with various example programs, verifying its correct operation and robustness. These examples serve as proof of the processor's capabilities and provide a foundation for further development or testing.
- **Educational Value:** This project serves as an excellent educational tool, providing insights into processor design and the RISC-V architecture. It can be used as a reference for learning about single-cycle processors and the implementation of different instruction types.

CONCLUSION

- Potential Improvements and future work:
- Firstly, the design doesn't support few instructions from the original RV32I ISA but that's not a big problem the reason I didn't include more instructions was for simplicity purpose.
- Adding more instructions is not difficult but since this micro-processor can do all essential instructions I didn't feel the urge to expand the ISA plus there is much more to improve rather than simply expanding the ISA.
- Multi-cycle and Pipelined designs are more popular and efficient nowadays therefore my next projects will most likely be about designing those versions.

REFERENCES

- I've only been relying on this book thus far which is excellent and beginner friendly in my opinion:
- **Digital Design and Computer Architecture RISC-V-Edition by Sarah Harris and David Harris.**
- Links for my LinkedIn, GitHub, social:
- LinkedIn: https://www.linkedin.com/in/alaa-sakran
- GitHub: https://github.com/Alaa-Sakran/SingleCycleRV32I_Processor
- Social: https://www.instagram.com/alaarushdie