RISC-V Single Cycle Implementation (Enhanced)

Created By:

Mohamed Ahmed Mohamed Hussein   *15/8/2024*

Introduction to RISC-V:

RISC-V is an open-source instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). RISC-V is designed to be simple, efficient, and highly extensible, making it suitable for a wide range of applications, from small embedded systems to high-performance computing. The RISC-V architecture emphasizes a streamlined instruction set, a load/store architecture, and the use of a fixed instruction length, all of which contribute to faster execution and easier implementation.

RISC-V Instruction Set Architecture (ISA):

The RISC-V ISA defines the set of instructions that a RISC-V processor can execute. These instructions are categorized into several types based on their format and functionality. The most important types in the RISC-V ISA are I-type, B-type, R-type, J-type, and S-type.

- *I-Type (Immediate Type):*

    I-type instructions are used for operations that involve an immediate value, such as arithmetic operations with a constant or load instructions. The immediate value is encoded within the instruction itself, allowing for quick access and execution.

I-Type Layout

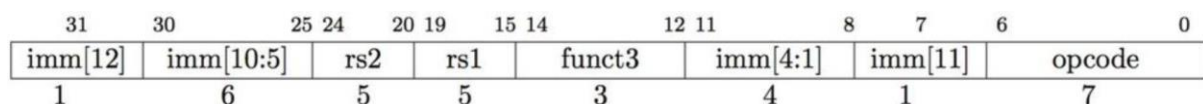| 31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
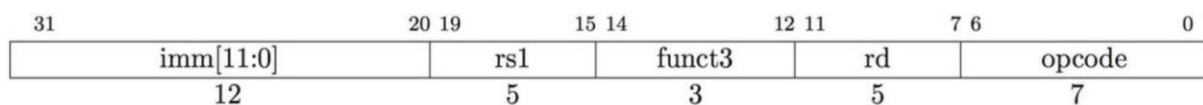
- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, `imm[11:0]`

- *B-Type (Branch Type):*

    B-type instructions are used for conditional branching. They allow the program to change the flow of execution based on the comparison of register values. The target address for the branch is calculated using an immediate value encoded within the instruction.
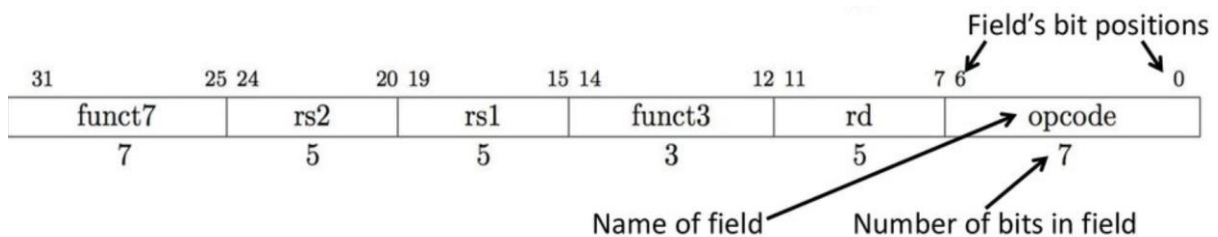
B-Type Layout

31	30	25 24	20 19	15 14	12 11	8 7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	
1	6	5	5	3	4	1	7	

- *R-Type (Register Type):*

  R-type instructions are used for arithmetic and logical operations that involve two source registers and a destination register. These instructions do not use immediate values and are typically used for operations like addition, subtraction, and logical AND/OR.
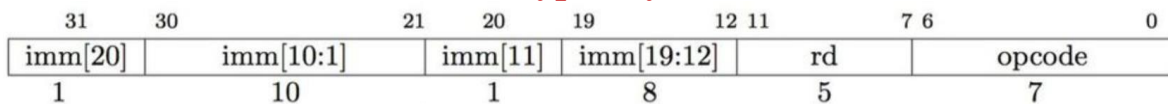
R-Type Layout

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

Field's bit positions

Name of field

Number of bits in field

- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32

- *J-Type (Jump Type):*

  J-type instructions are used for jump operations, where the program counter is updated to a new address. This type is typically used for unconditional jumps and function calls, with the target address being calculated using an immediate value.

J-Type Layout

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7

- *S-Type (Store Type):*

  S-type instructions are used for store operations, where data from a register is stored into memory. The memory address is calculated using a base address from a register and an immediate value.

S-Type Layout

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE
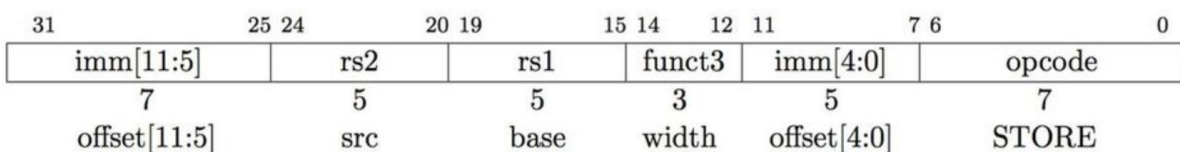
Summary of RISC-V Instructions

31	30	25	24	21	20	19	15	14	12	11	8	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]	imm[10:5]		rs2			rs1		funct3		imm[4:1]		imm[11]	opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]			imm[11]		imm[19:12]				rd			opcode		J-type

---

RISC-V Single Cycle Processor:

A single-cycle processor executes each instruction in one clock cycle. In a RISC-V single-cycle processor, all stages of instruction execution—fetch, decode, execute, memory access, and write-back—occur within a single cycle. This design is straightforward and easy to implement but can be inefficient because the clock cycle must accommodate the longest instruction.

- *Schematic:*
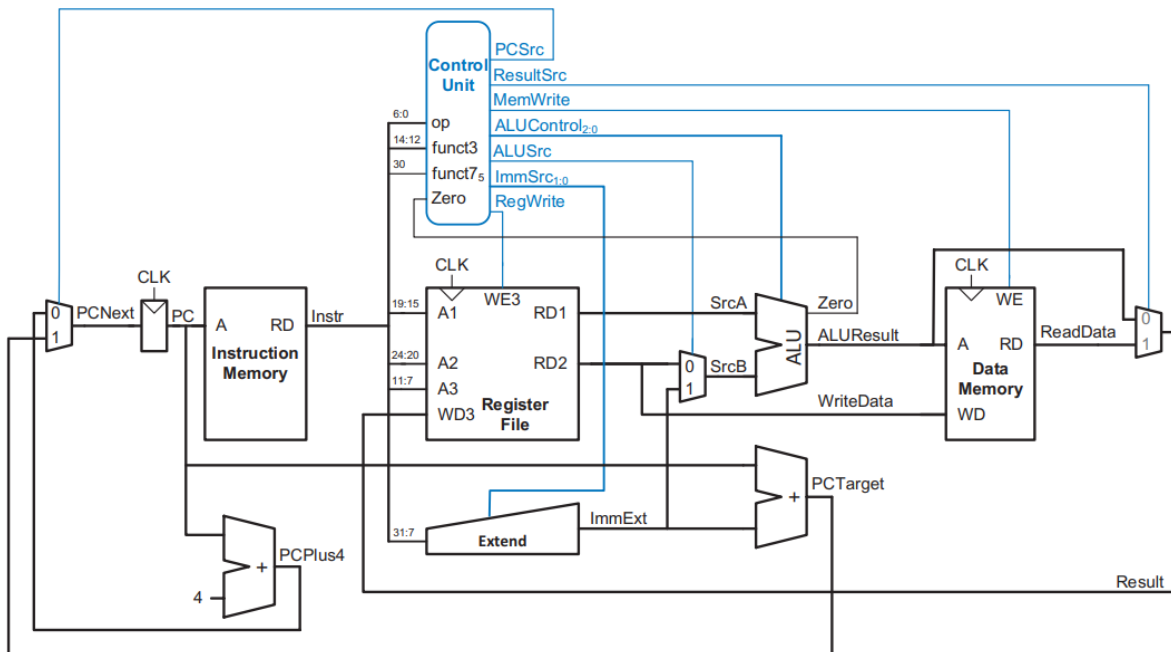
I even enhanced the processor to include **J-Type** instructions!

- *Enhanced Schematic:*
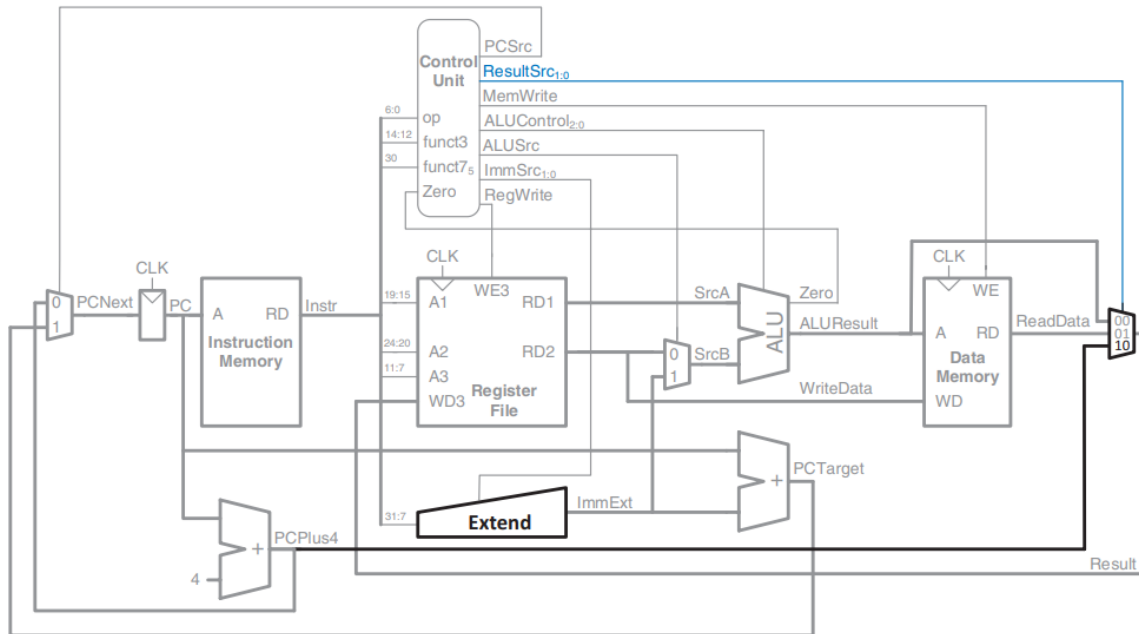
More Details:

- **Control Unit from inside:**
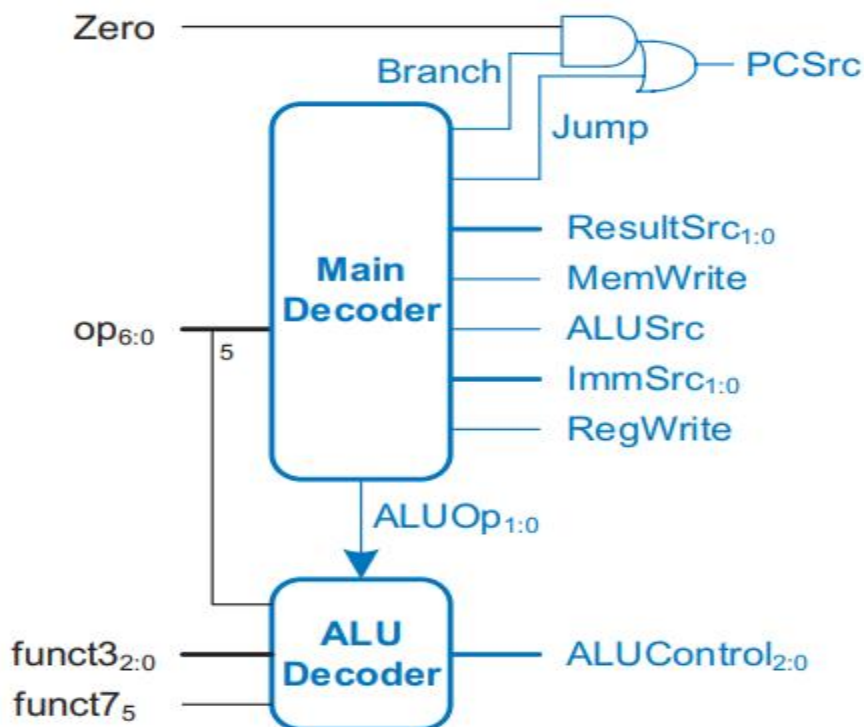
o Main Decoder Truth Table:

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

o ALU Decoder Truth Table:

ALUOp	funct3	{op$_5$, funct7$_5$}	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

• Immediate layout for each type:

ImmSrc	ImmExt	Type	Description
00	{{20{$Instr$[31]}}, $Instr$[31:20]}	I	12-bit signed immediate
01	{{20{$Instr$[31]}}, $Instr$[31:25], $Instr$[11:7]}	S	12-bit signed immediate
10	{{20{$Instr$[31]}}, $Instr$[7], $Instr$[30:25], $Instr$[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{$Instr$[31]}}, $Instr$[19:12], $Instr$[20], $Instr$[30:21], 1'b0}	J	21-bit signed immediate

Verifying Functionality:

We will write an assembly code with a specific scenario to check our design.

• Assembly instructions:

```
1    //assembly code to test functionality
2
3    //1: I-type
4    //2: S-type
5    //3: R-type
6    //4: B-type
7    //5: J-type
8    //6: I-type
9
10   lw x6, -4(x9)   //Address 0x0000  (will assign the value 9 from data memory in x6)
11   sw x6, 8(x9)    //Address 0x0004  (will print the value 9 at address 3 in the Data Memory)
12   or x4, x5, x6   //Address 0x0008
13   beq x4, x4, 32  //Address 0x000C  (this will increment the pc by 16 which increments the address of instruction memory by 4)
14   jal x10 , 128   //Address 0x001C  (this will increment the PC by 64 which will incremnt the instruction memory address by 16)
15   addi x1, x0, 8  //Address 0x005C
```

In **assembly.asm** file you can checkout the code

• Scenario:

Initially register **x9** holds the value **4**, **x5** holds the value **6** and memory location **0x0000** in **Data Memory** has the value **9**, so according to the assembly instructions we are expecting the following:

- Value 9 loaded to x6
- Value 9 loaded at data memory location 3 as we will add as we add the base register (x9) value (4) to the offset (8) and then divide it by 4 we will get the memory address 3
- X4 will have the value 0xF as (9 OR 6)

  (1001 OR 0110) will be 15

- Beq instruction will increment the PC by 16 which increments the address of instruction memory by 4
- Jal instruction will increment the PC by 64 which will increment the instruction memory address by 16, and it will assign (PC + 4) to the destination register which is x10, so the register x10 will have the value 0x20
- Finally I add another I-type instruction to test the branching of jal instruction, so the register x1 will have the value 8 at the end

Note: we will see each one of the above observations in order in the **snippets** section

• **Snippets:**

o Snapshot of whole wave form:
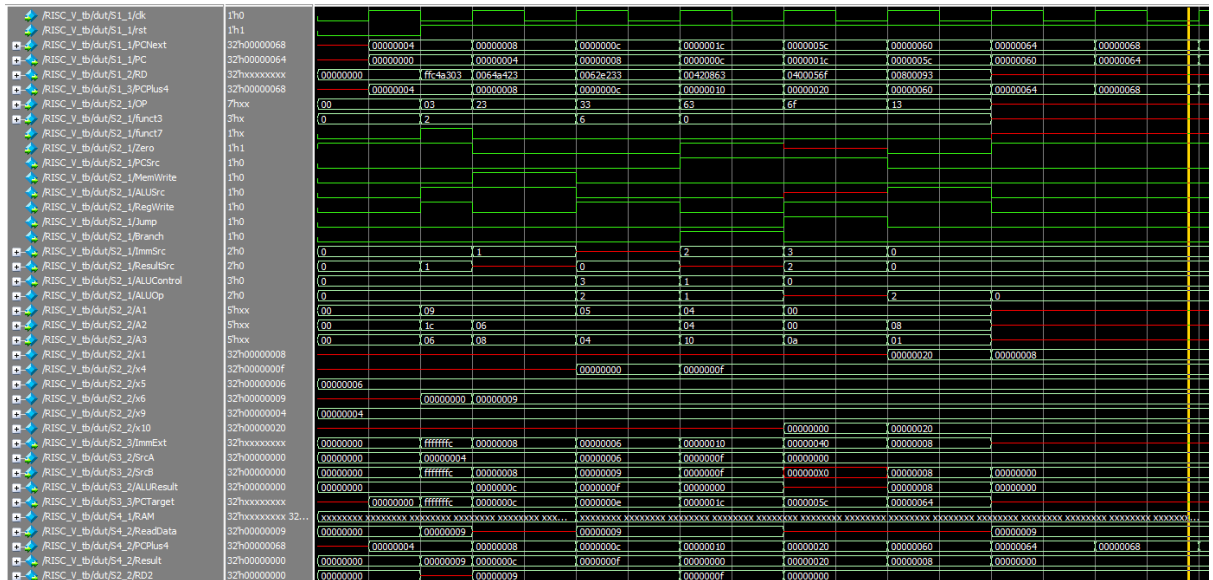
o Snapshot of x6 having the value 9:
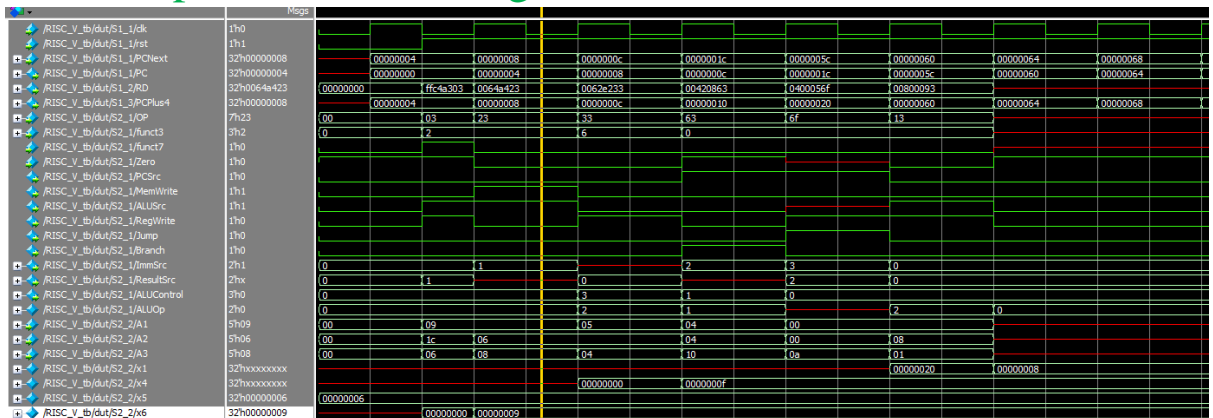
o Snapshot of data memory address 3 having the value 9:
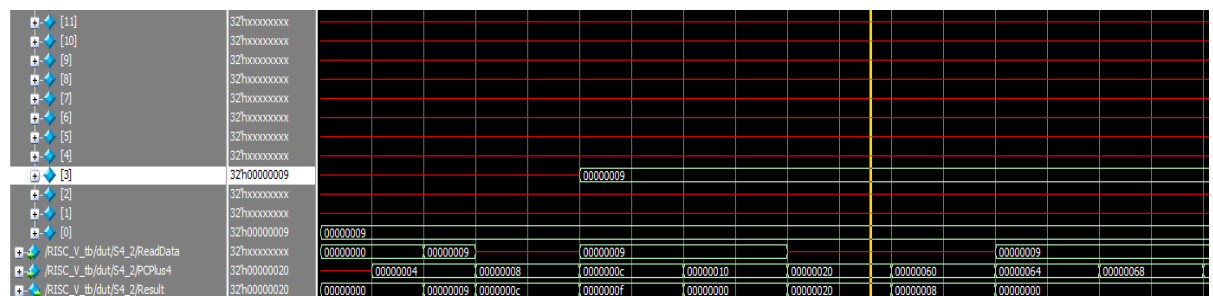
o Snapshot of x4 having the value 15 (0xF):

o Snapshot of the PC incremented by 16 (0x10):

o Snapshot of the PC incremented by 64 (0x40):

o Snapshot of x10 having the value 0x20 (PC + 4):
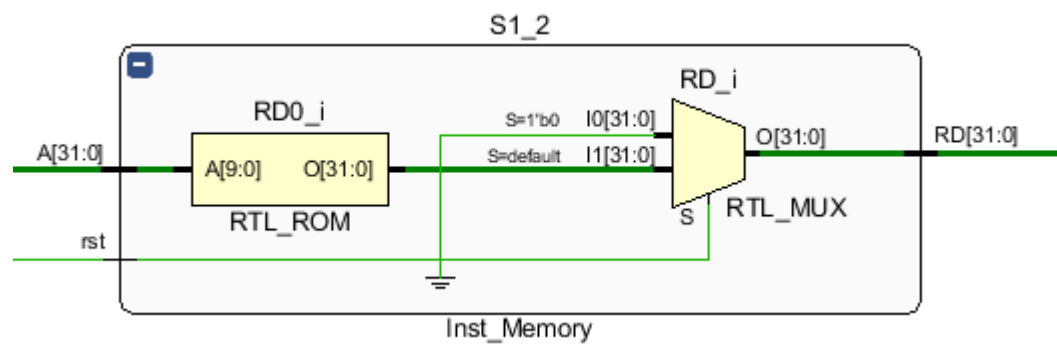

o Snapshot of x1 having the value 8:

Vivado:
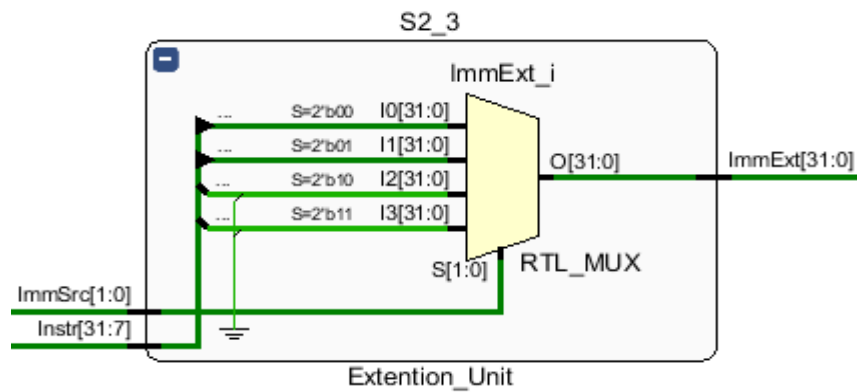
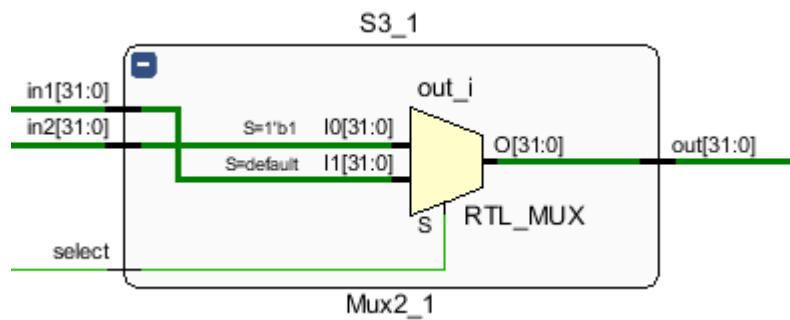- Elaboration:
  - System:
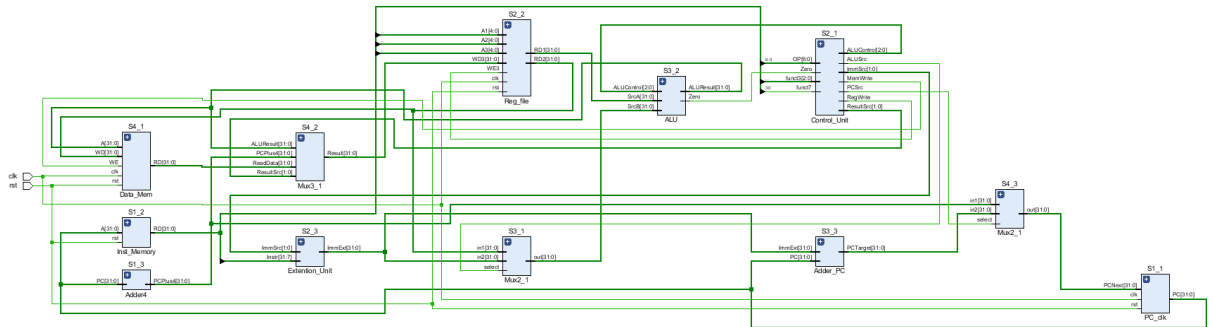
  - PC_clk:
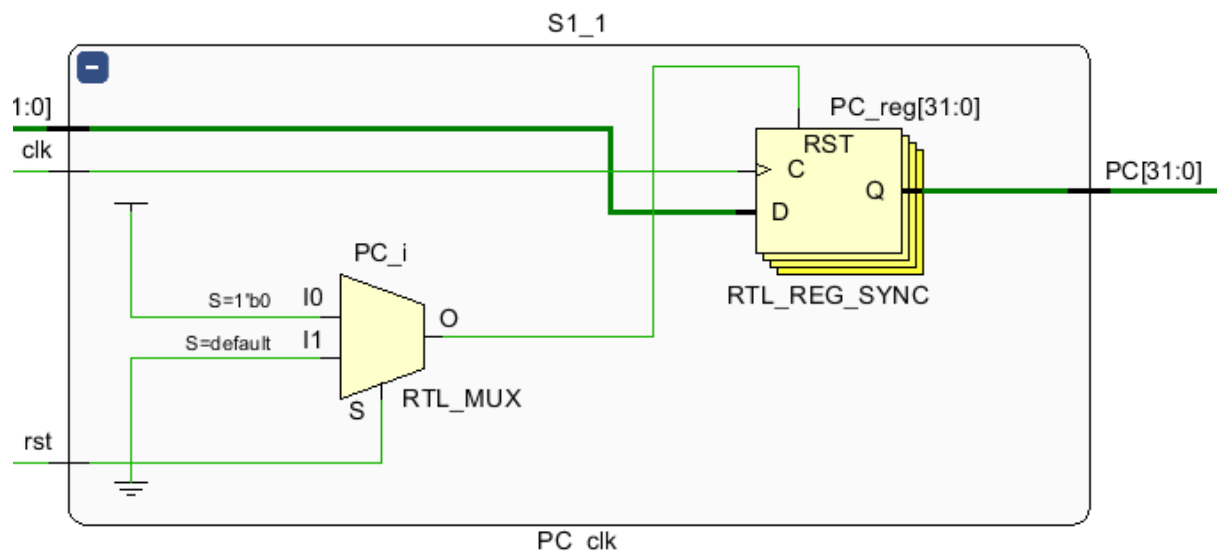
  - Inst_Memory:

- Adder4:
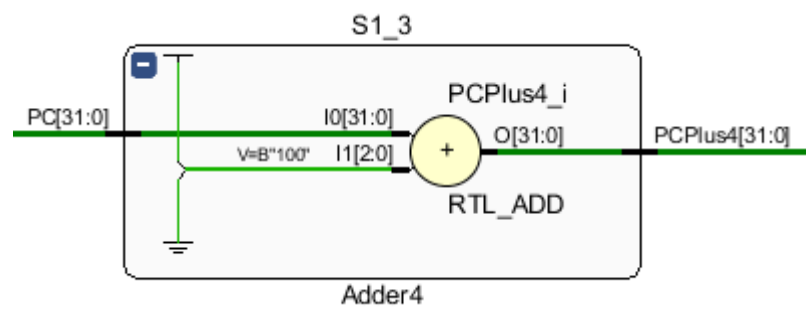
- Reg_File:
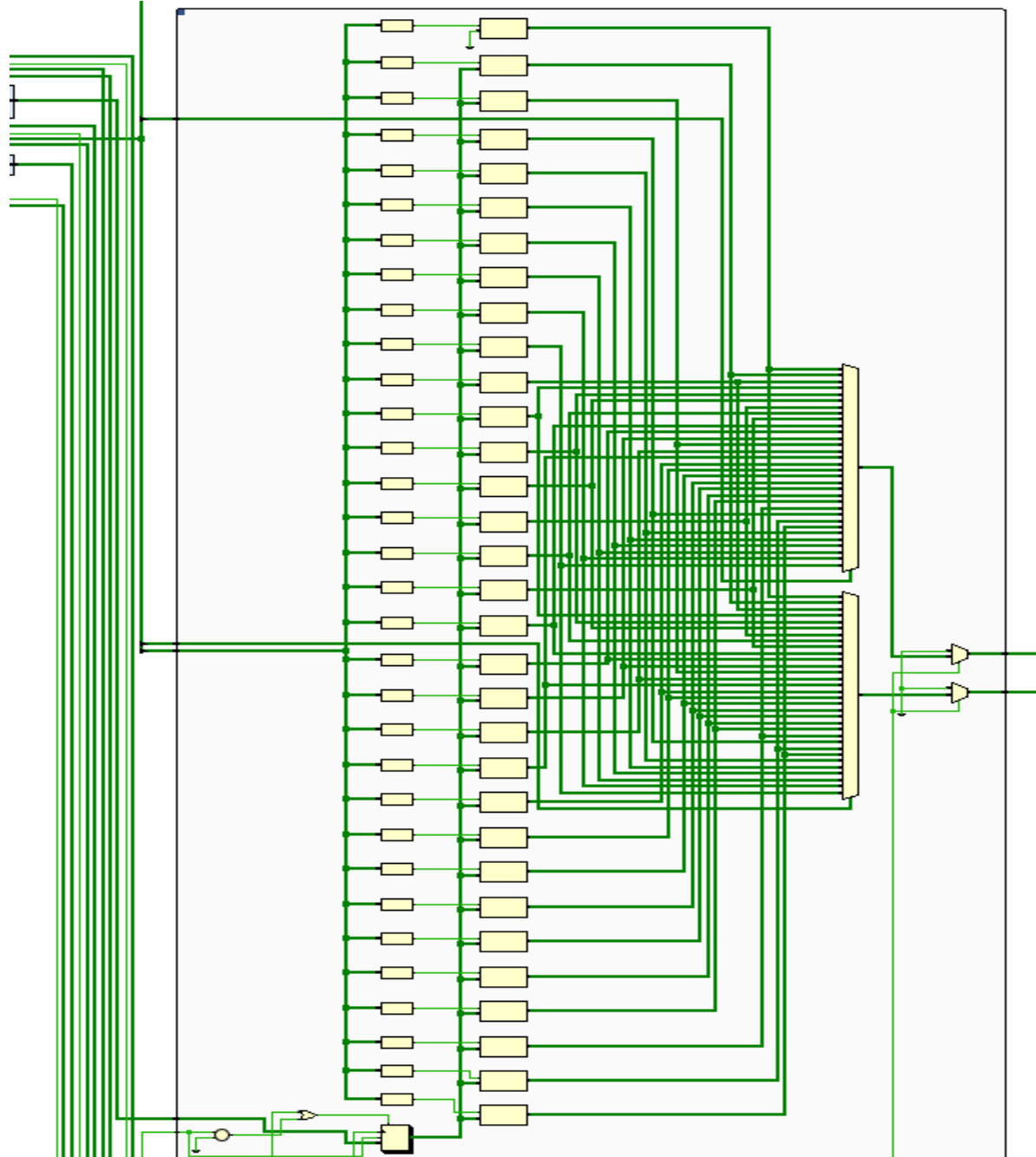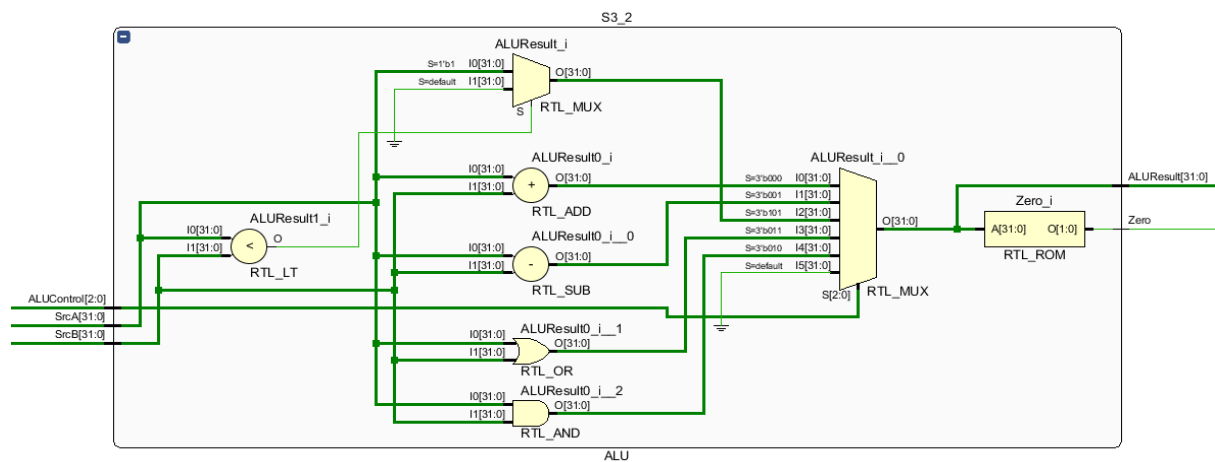
- **Extention_Unit:**
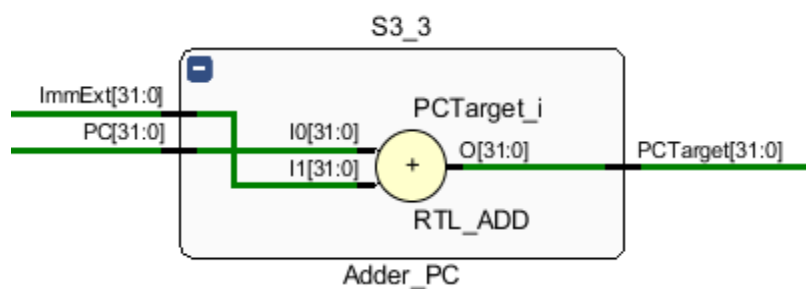

- **Mux2_1(SrcB):**

- **ALU:**

- **Adder_PC:**

- **Data_Mem:**

Data_Mem

- **Mux3_1:**

Mux3_1
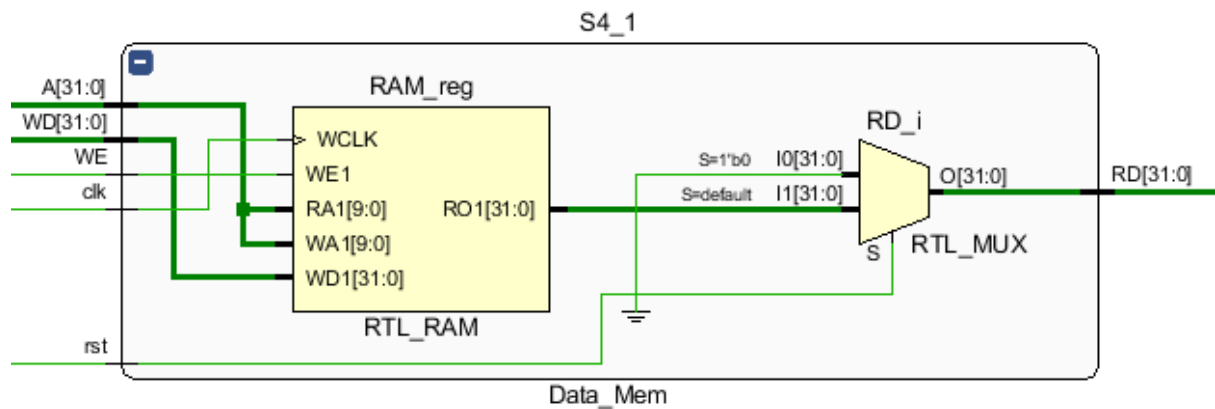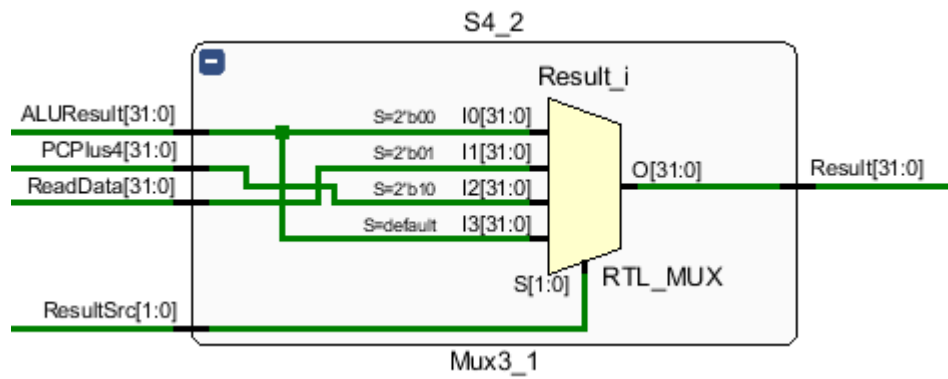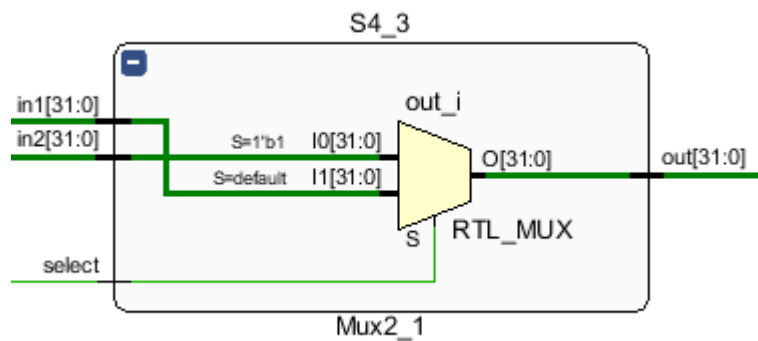
- **Mux2_1(PC):**

Mux2_1

References:

Digital Design and Computer Architecture RISC-V Edition, Sarah L. Harris David Harris