Mini Report: Design of a 32-bit Single-Cycle CPU

Weiyuan Du

1 Introduction

In this project, I implemented a simplified 32-bit single-cycle CPU that supports a subset of RISC-V instructions. The primary focus was on the design of essential components such as the controller, immediate extension unit, ALU interface, and next-PC logic. Since the scope of the assignment does not require future extensibility, the design concentrates strictly on the required instruction types. All implemented instructions have been tested and shown to operate correctly.

2 System Design

2.1 NPC (Next Program Counter) Module

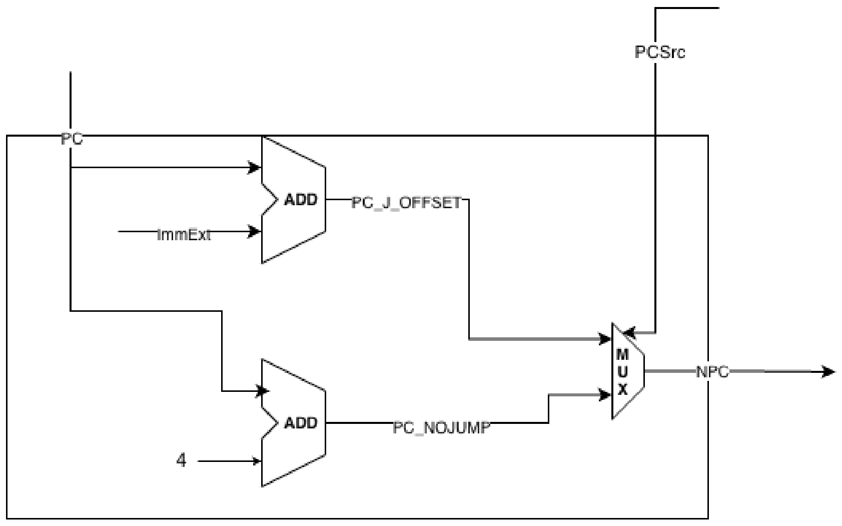


Figure 1: NPC Module

A dedicated NPC module is designed to compute the next program counter value. It supports two update modes:

- **PC_NOJUMP**: increments the program counter by 4 to fetch the next sequential instruction.
- **PC_J_OFFSET**: computes the next PC as the current address plus a sign-extended immediate, enabling branch and jump instructions.

A multiplexer controlled by the `PCSrc` signal selects between these two values before forwarding the result to the PC register.

2.2 Controller Module

Following the recommendation in the assignment, the controller is separated into two submodules to improve clarity and simplify decoding logic:

- **Op_Decoder**: performs first-level decoding and determines the instruction category as well as a coarse ALU operation class.
- **ALU_Decoder**: receives a 3-bit `ALUOp` signal from the first level and resolves it into the exact ALU control signal.

A zero flag is generated to support branch decisions. Since the RISC-V architecture does not require hardware support for overflow or carry flags, these conditions must be handled in software. In addition, a 3-bit `ImmSrc` signal is used to distinguish the five types of immediate extensions.

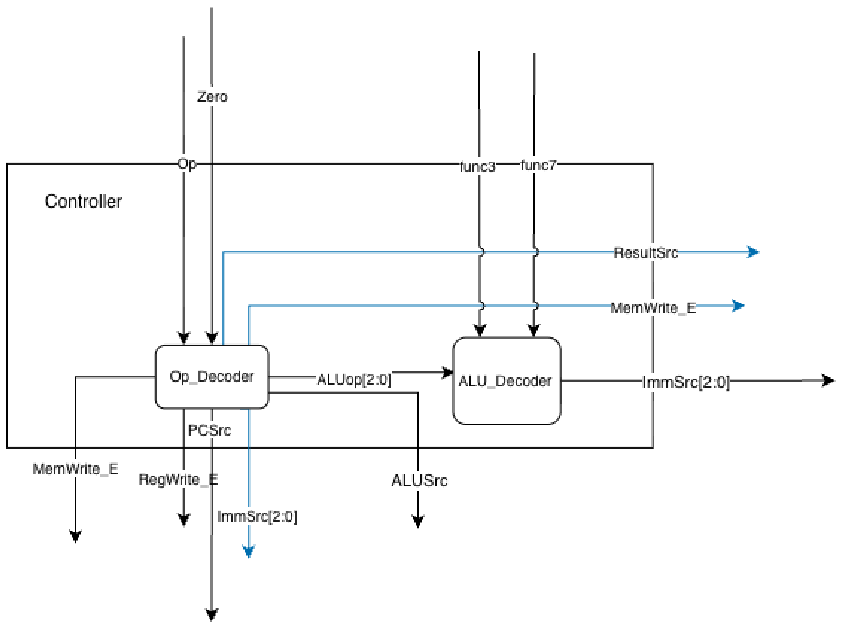


Figure 2: Controller Module

2.3 Overall Design

Green-highlighted components in the diagram show my modifications to the baseline CPU architecture. Some blocks are not highlighted because they existed in the original design and were only renamed for clarity.

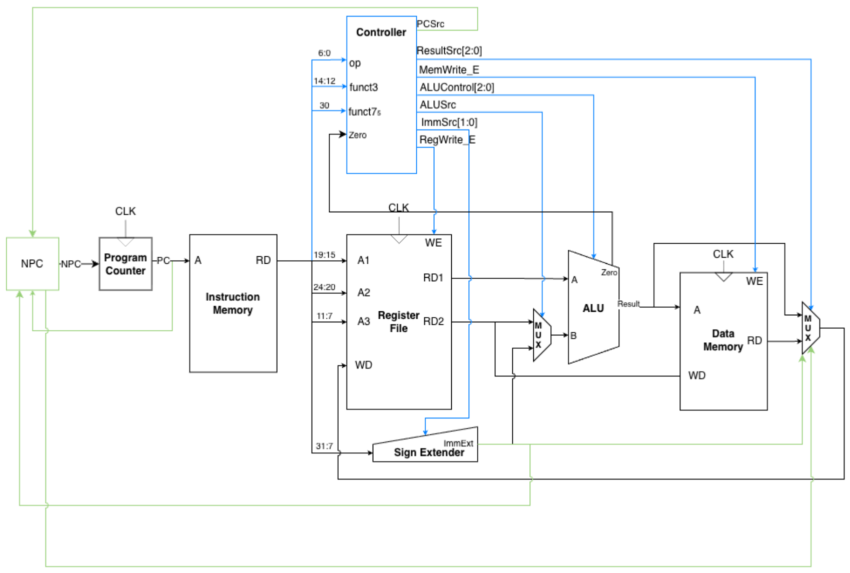
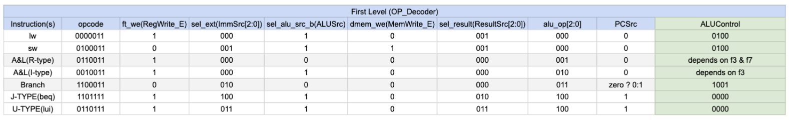


Figure 3: CPU Top-Level Architecture

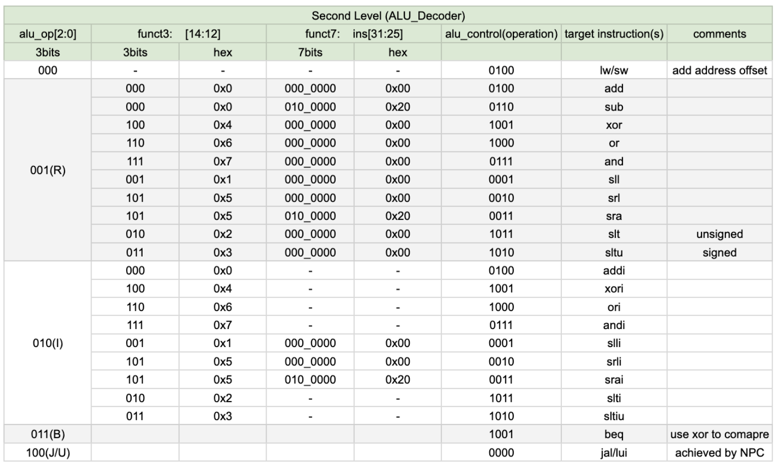
3 Two-Level Decoding Strategy

The two-level decoding strategy significantly simplifies controller logic and improves readability. The first-level decoder classifies instructions into I-type, S-type, B-type, U-type, and J-type. The second level receives a 3-bit `ALUOp` and resolves it into a precise ALU control signal.



First Level (OP_Decoder)									
Instruction(s)	opcode	ft_we(RegWrite_E)	sel_ext(ImmSrc[2:0])	sel_alu_src_b(ALUSrc)	dmem_we(MemWrite_E)	sel_result(ResultSrc[2:0])	alu_op[2:0]	PCSrc	ALUControl
lw	0000011	1	000	1	0	001	000	0	0100
sw	0100011	0	001	1	1	001	000	0	0100
A&L(R-type)	0110011	1	000	0	0	000	001	0	depends on f3 & f7
A&L(I-type)	0010011	1	000	1	0	000	010	0	depends on f3
Branch	1100011	0	010	0	0	000	011	zero ? 0:1	1001
J-TYPE(beq)	1101111	1	100	1	0	010	100	1	0000
U-TYPE(lui)	0110111	1	011	1	0	011	100	1	0000

Figure 4: First-Level Decoder (Op_Decoder)



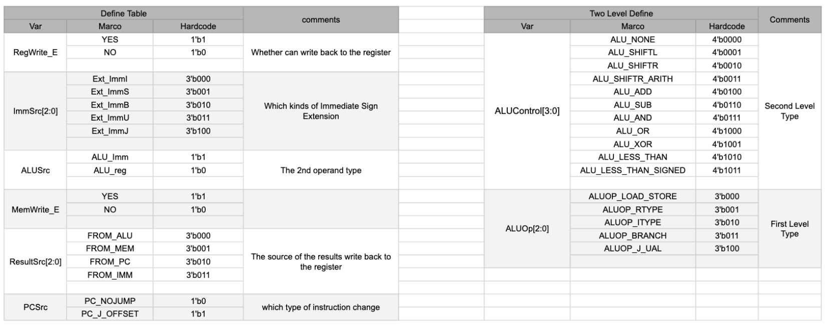
Second Level (ALU_Decoder)							
alu_op[2:0]	funct3: [14:12]		funct7: ins[31:25]		alu_control(operation)	target instruction(s)	comments
3bits	3bits	hex	7bits	hex			
000	-	-	-	-	0100	lw/sw	add address offset
001(R)	000	0x0	000_0000	0x00	0100	add	
	000	0x0	010_0000	0x20	0110	sub	
	100	0x4	000_0000	0x00	1001	xor	
	110	0x6	000_0000	0x00	1000	or	
	111	0x7	000_0000	0x00	0111	and	
	001	0x1	000_0000	0x00	0001	sll	
	101	0x5	000_0000	0x00	0010	srl	
	101	0x5	010_0000	0x20	0011	sra	
	010	0x2	000_0000	0x00	1011	slt	unsigned
	011	0x3	000_0000	0x00	1010	sltu	signed
010(I)	000	0x0	-	-	0100	addi	
	100	0x4	-	-	1001	xori	
	110	0x6	-	-	1000	ori	
	111	0x7	-	-	0111	andi	
	001	0x1	000_0000	0x00	0001	slli	
	101	0x5	000_0000	0x00	0010	srli	
	101	0x5	010_0000	0x20	0011	srai	
	010	0x2	-	-	1011	slti	
	011	0x3	-	-	1010	sltiu	
011(B)					1001	beq	use xor to comapre
100(J/U)					0000	jal/lui	achieved by NPC

Figure 5: Second-Level Decoder (ALU_Decoder)

The check of branch is achieved by **ALU_XOR**. As if two numbers are identicial, the xor of them will be zero. This is done by the ALU module. But **jal** is directly achieved by the NPC module.

3.1 Definition Tables

To streamline the implementation, macros are heavily used in the Verilog code. The following illustration contains the decoding tables and macro definitions referenced by the controller:



Define Table			comments
Var	Marco	Hardcode	
RegWrite_E	YES	1'b1	Whether can write back to the register
	NO	1'b0	
ImmSrc[2:0]	Ext_ImmI	3'b000	Which kinds of Immediate Sign Extension
	Ext_ImmS	3'b001	
	Ext_ImmB	3'b010	
	Ext_ImmU	3'b011	
	Ext_ImmJ	3'b100	
ALUSrc	ALU_Imm	1'b1	The 2nd operand type
	ALU_reg	1'b0	
MemWrite_E	YES	1'b1	
	NO	1'b0	
ResultSrc[2:0]	FROM_ALU	3'b000	The source of the results write back to the register
	FROM_MEM	3'b001	
	FROM_PC	3'b010	
	FROM_IMM	3'b011	
PCSrc	PC_NOJUMP	1'b0	which type of instruction change
	PC_J_OFFSET	1'b1	

Two Level Define			Comments
Var	Marco	Hardcode	
ALUControl[3:0]	ALU_NONE	4'b0000	Second Level Type
	ALU_SHIFTL	4'b0001	
	ALU_SHIFTR	4'b0010	
	ALU_SHIFTR_ARITH	4'b0011	
	ALU_ADD	4'b0100	
	ALU_SUB	4'b0110	
	ALU_AND	4'b0111	
	ALU_OR	4'b1000	
	ALU_XOR	4'b1001	
	ALU_LESS_THAN	4'b1010	
	ALU_LESS_THAN_SIGNED	4'b1011	
ALUOp[2:0]	ALUOP_LOAD_STORE	3'b000	First Level Type
	ALUOP_RTYPE	3'b001	
	ALUOP_ITYPE	3'b010	
	ALUOP_BRANCH	3'b011	
	ALUOP_J_UAL	3'b100	

Figure 6: Macro Definition Tables

4 Assembly Test Program

A complete RISC-V assembly test program was written to validate the correctness of the CPU. The test covers all implemented instruction types. Each instruction is annotated with comments explaining its expected behavior. A corresponding Verilog testbench is used to verify execution results.

Listing 1: RISC-V Test Assembly

```
.text
.global _start

_start:
    # ===== I-TYPE IMMEDIATE OPERATIONS =====
    addi x1, x0, 10            # x1 = 10
    addi x2, x0, 5             # x2 = 5
    addi x3, x0, 15            # x3 = 15

    # ===== R-TYPE ARITHMETIC OPERATIONS =====
    add  x4, x1, x2            # x4 = 10 + 5 = 15
    sub  x5, x1, x2            # x5 = 10 - 5 = 5

    # ===== R-TYPE LOGICAL OPERATIONS =====
    and  x6, x1, x2            # x6 = 10 & 5 = 0
    or   x7, x1, x2            # x7 = 10 | 5 = 15
    xor  x8, x1, x2            # x8 = 10 ^ 5 = 15

    # ===== I-TYPE LOGICAL OPERATIONS =====
    andi x9, x3, 7             # x9 = 15 & 7 = 7
    ori  x10, x1, 8            # x10 = 10 | 8 = 10
    xori x11, x3, 7            # x11 = 15 ^ 7 = 8

    # ===== SHIFT OPERATIONS =====
    slli x12, x1, 1            # x12 = 10 << 1 = 20
    srli x13, x3, 1            # x13 = 15 >> 1 = 7
    srai x14, x1, 1           # x14 = 10 >> 1 = 5

    # ===== COMPARISON OPERATIONS =====
    slt  x15, x2, x1          # x15 = 1 if 5 < 10 = 1
    sltu x16, x1, x2          # x16 = 1 if 10 < 5 (unsigned) = 0
    slti x17, x1, 15          # x17 = 1 if 10 < 15 = 1
    sltiu x18, x2, 3          # x18 = 1 if 5 < 3 (unsigned) = 0

    # ===== U-TYPE OPERATIONS =====
    lui  x19, 0x1             # x19 = 0x1000

    # ===== MEMORY OPERATIONS =====
    sw   x4, 100(x1)          # Store x4 (15) at mem[100+10] = mem[110]
    sw   x5, 104(x2)          # Store x5 (5) at mem[104+5] = mem[109]
    lw   x20, 100(x1)         # Load x20 from mem[110] (should be 15)
    lw   x21, 104(x2)         # Load x21 from mem[109] (should be 5)

    # ===== BRANCH OPERATION (BEQ only) =====
    beq  x20, x4, success     # Should branch (15 == 15)
    addi x22, x0, 999         # Should be skipped
    addi x21, x0, 888         # Should be skipped

success:
    addi x22, x0, 42          # x22 = 42

    # ===== JUMP OPERATION =====
    jal  x23, function
    addi x24, x0, 100         # x24 = 100

    sw   x22, 8(x19)         # x22 =  42
    sw   x23, 12(x19)        # x23 = 116
    sw   x24, 16(x19)        # x24 = 100

end_loop:
    beq  x0, x0, end_loop     # Loop forever (BEQ x0,x0 always true)

function:
    addi x25, x0, 200         # x25 = 200 (function executed flag)
```

5 Limitation

The current implementation requires the machine code loaded into the system to begin with a line of zeros. This is due to the PC module updating on the positive clock edge. Because I am still refining my understanding of combinational versus sequential logic, I have not yet implemented a fully robust mechanism to control the initial PC state.

Conceptually, inserting a “null” instruction to mark the start (and optionally the end) of the program is simple and resembles conventions found in hardware and signal-processing systems. However, this behavior is not part of the official RISC-V ISA. I plan to improve this aspect in future iterations.

6 Code Availability

All source code and design files are available in my GitHub repository: You can find more test bench in my repository, and also automativie tools help to reduce your work. :) `https:`

`//github.com/SinYita/riscv-single`