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

Weiyuan Du

1 Introduction

In this project, I implemented a 32-bit multi-cycle RISC-V CPU that supports a functional subset of instructions including R-type, I-type (ALU and Load), Store, Branch, and Jump operations. Unlike a single-cycle design, this architecture optimizes hardware usage by utilizing a shared ALU and shared memory for both instructions and data, managed through a central Finite State Machine (FSM).

2 Questions

1. **In Figure 1, some registers have write enable (en) while some do not. Explain why en is necessary or not.**
   Enable signals are necessary for registers that must hold their data across multiple clock cycles, such as the `PC`, `Instruction Register`, and `Register File`. Without an enable, these would be overwritten every cycle. Temporary registers like `ALUOut` do not need enables because they only cached content until next stage.

2. **Why is the output of the sign extender not registered?**
   The sign extender is a combinational circuit that takes its input from the `Instruction Register` (IR). Since the IR is already a register, the output of the sign extender remains stable without needing its own register.

3. **Where is the slowest datapath (critical path) between two registers, given the delays of the components are constants?**
   The critical path is typically the memory access path:

   $$\text{PC} \rightarrow \text{Address Mux} \rightarrow \text{Memory} \rightarrow \text{Instruction Register}$$

4. **How does a multicycle architecture allow the clock period to be shorter than in a single-cycle one? What is the benefit?**
   A multicycle design breaks instructions into smaller steps. The clock period only needs to be long enough for the longest *individual* rather than the entire instruction. This save times by preventing simple instructions waiting all the time.

5. **Why might a multicycle design be easier (or harder) to extend with new instructions compared to a single-cycle design?**
   It is easier because you can reuse existing hardware (like the ALU) by simply adding new states to the FSM. However, it is harder because the FSM logic becomes significantly more complex to design and verify as the number of instructions and states increases.

6. **Briefly describe the bug you have encountered.**
   I encountered a significant bug where the control logic for `Branch` and `R-type` instructions was incorrectly encoded together. This overlap made it impossible for the ALU to stably identify the required operation, leading to frequent execution failures. I resolved this

by re-encoding the `ALUOp` signals to clearly separate branch instructions from R-type operations, ensuring the `ALUControl` logic could reliably generate the correct signals for each instruction type.

2.0.1 Performance Analysis

Instruction Type	Cycles per Instr.	# in Program	Total Cycles
R-Type (add, sub, or, etc.)	4	7	28
I-Type (addi)	4	4	16
U-Type (lui)	4	1	4
Load (lw)	5	1	5
Store (sw)	4	2	8
Branch (beq)	3	3	9
Jump (jal)	4	1	4
FSM Latency / Startup	-	-	2
Total	-	**19**	**76**

Table 1: Performance Analysis: Instruction Breakdown and Cycle Counts

Performance Analysis: The manual calculation for the 19 executed instructions sums to 74 cycles. The simulation reports 76 cycles ($\approx 2.7\%$ difference). This discrepancy is due to the hardware testbench capturing the initial clock cycle during the reset phase and the final cycle required for the Program Counter (PC) to update to the terminal branch address before the simulation *stop* condition is met.

Average CPI:

$$\text{CPI} = \frac{\text{Cycles}}{\text{Instructions}} = \frac{74}{19} = 3.89$$

3 System Design

3.1 FSM

Please find attached pdf page at the end of this report.

3.2 Overall Design

Please find attached pdf page at the end of this report.

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:
lui  x6, 0x00010          # x6 = 0x00010000 (65536)
addi x2, x0, 5            # x2 = 5
addi x3, x0, 12           # x3 = 12
addi x7, x3, -9           # x7 = (12 - 9) = 3
```

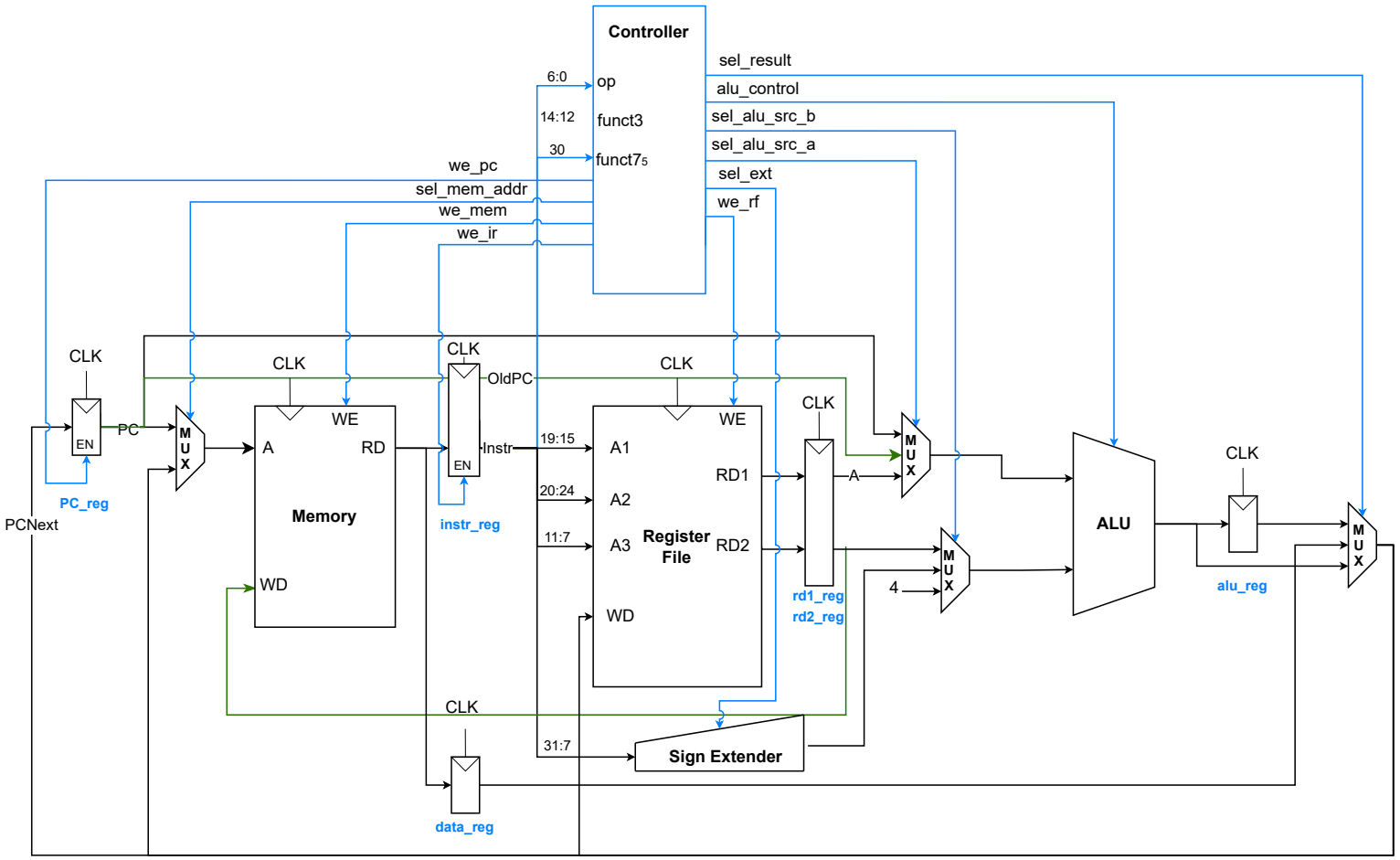
```
or   x4, x7, x2          # x4 = (3 OR 5) = 7
and  x5, x3, x4          # x5 = (12 AND 7) = 4
add  x5, x5, x4          # x5 = (4 + 7) = 11
beq  x5, x7, end         # shouldn't be taken
slt  x4, x3, x4          # x4 = (12 < 7) = 0
beq  x4, x0, around      # should be taken
addi x5, x0, 0           # shouldn't happen
around:
slt  x4, x7, x2          # x4 = (3 < 5) = 1
add  x7, x4, x5          # x7 = (1 + 11) = 12
sub  x7, x7, x2          # x7 = (12 - 5) = 7
sw   x7, 84(x3)          # [96] = 7
lw   x2, 96(x0)          # x2 = [96] = 7
add  x9, x2, x5          # x9 = (7 + 11) = 18
jal  x3, end             # jump to end, x3 = return address
addi x2, x0, 1           # shouldn't happen
end:
add  x2, x2, x9          # x2 = (7 + 18) = 25
sw   x2, 0x20(x3)        # mem[104] = 25
done:
beq  x2, x2, done        # infinite loop
```

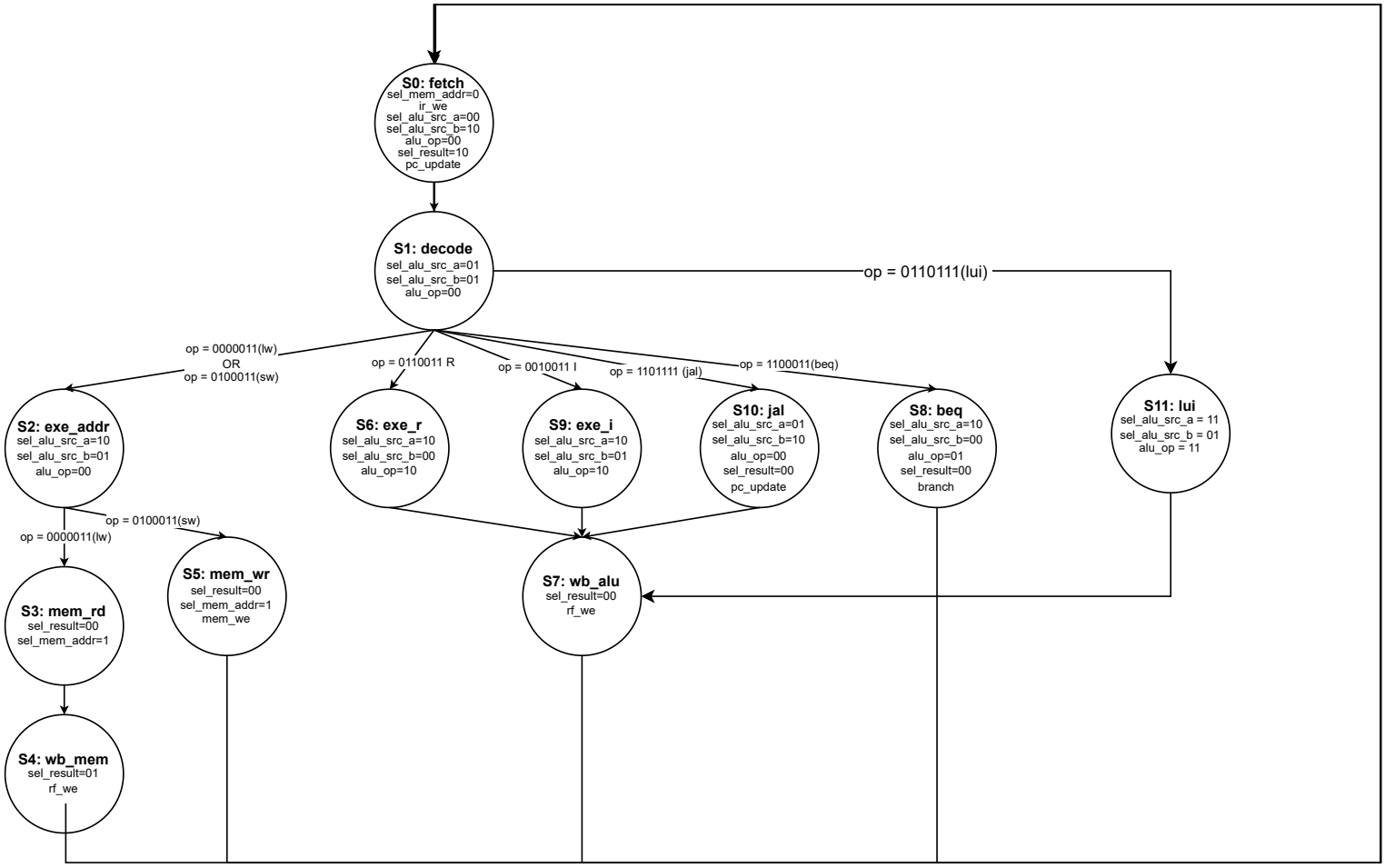
5 Limitation

I only checked with the assembly code inside my project. To make it more clear and readable, I didn't do a lot test bench works, except the top level one. There might be some other bugs hidden in my project.

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.git`


Controller
op
funct3
funct7₅
6:0
14:12
30
sel_result
alu_control
sel_alu_src_b
sel_alu_src_a
sel_ext
we_rf
we_pc
sel_mem_addr
we_mem
we_ir
CLK
EN
PC
PCNext
PC_reg
MUX
A
WE
RD
WD
Memory
OldPC
Instr
instr_reg
19:15
20:24
11:7
A1
A2
A3
Register File
RD1
RD2
WE
WD
rd1_reg
rd2_reg
A
4
ALU
alu_reg
31:7
Sign Extender
data_reg


S0: fetch
sel_mem_addr=0
ir_we
sel_alu_src_a=00
sel_alu_src_b=10
alu_op=00
sel_result=10
pc_update
S1: decode
sel_alu_src_a=01
sel_alu_src_b=01
alu_op=00
op = 0110111(lui)
op = 0000011(lw)
OR
op = 0100011(sw)
op = 0110011 R
op = 0010011 I
op = 1101111 (jal)
op = 1100011(beq)
S2: exe_addr
sel_alu_src_a=10
sel_alu_src_b=01
alu_op=00
S6: exe_r
sel_alu_src_a=10
sel_alu_src_b=00
alu_op=10
S9: exe_i
sel_alu_src_a=10
sel_alu_src_b=01
alu_op=10
S10: jal
sel_alu_src_a=01
sel_alu_src_b=10
alu_op=00
sel_result=00
pc_update
S8: beq
sel_alu_src_a=10
sel_alu_src_b=00
alu_op=01
sel_result=00
branch
S11: lui
sel_alu_src_a = 11
sel_alu_src_b = 01
alu_op = 11
op = 0100011(sw)
op = 0000011(lw)
S3: mem_rd
sel_result=00
sel_mem_addr=1
S5: mem_wr
sel_result=00
sel_mem_addr=1
mem_we
S7: wb_alu
sel_result=00
rf_we
S4: wb_mem
sel_result=01
rf_we