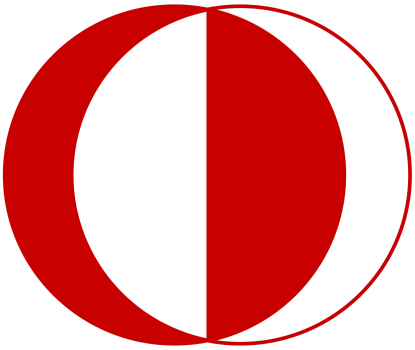


MIDDLE EAST TECHNICAL UNIVERSITY

ELECTRICAL AND ELECTRONICS
ENGINEERING DEPARTMENT

EE446 Computer Architecture II Laboratory Project

“Single Cycle RISC-V Processor”

Project Report

Group: 28

Ezgi Sena Karabacak - 2375178

Merve Nur Zembil - 2376242

Table of Contents

Introduction

This report presents the design and implementation of a Single-Cycle 32-bit RISC-V Processor, undertaken as a lab project for our Computer Architecture II course at METU. Building on our foundational knowledge from previous labs, which focused on ARM architecture, this project allowed us to delve into the innovative, open-source RISC-V instruction set architecture. The RISC-V ISA's modularity and flexibility provided a comprehensive learning experience as we developed the datapath and control unit, incorporating an additional custom instruction to extend the base ISA. The culmination of our efforts involved embedding our design into the FPGA of the DE1-SoC board, demonstrating its functionality and performance.

Datapath

In our design, we improved the datapath given in the project description pdf. Our improvements are listed as follows:

- We have added a shifter that can perform the SLL, SRL, and SRA instructions.
- We have added a mux to the Shamt of the shifter to choose between ImmExt and RD2 values.
- We have added another much to the PCTarget adder to choose between ImmExt and 12-bit left-shifted ImmExt.
- Finally, we extended the Result selection mux, so the following is also connected to it as inputs:
    - Added shifter’s output
    - ALU’s negative flag extended to have 31-zeroes before it
    - PCPlus4
    - PCTarget

In order to implement functionalities for load/store instructions with different sizes, we made some modifications. A 3-bit MemControl signal is added to change the type of memory operation, including word (32-bit), halfword (16-bit), and byte (8-bit) loads and stores. The operations can be signed or unsigned as required.

Read Operations

- Word (32-bit): Reads four consecutive bytes.
- Halfword Unsigned (16-bit): Reads two bytes with zero extension.
- Halfword (16-bit): Reads two bytes with sign extension.
- Byte Unsigned (8-bit): Reads a single byte with zero extension.
- Byte (8-bit): Reads a single byte with sign extension.

Write Operations

On the rising edge of the clock, if the write enable (WE) signal is asserted, data is written to memory based on the MemControl signal:

- Word (32-bit): Writes four consecutive bytes.
- Halfword (16-bit): Writes two bytes.
- Byte (8-bit): Writes a single byte.

This improved datapath design can be found in Figure 1.

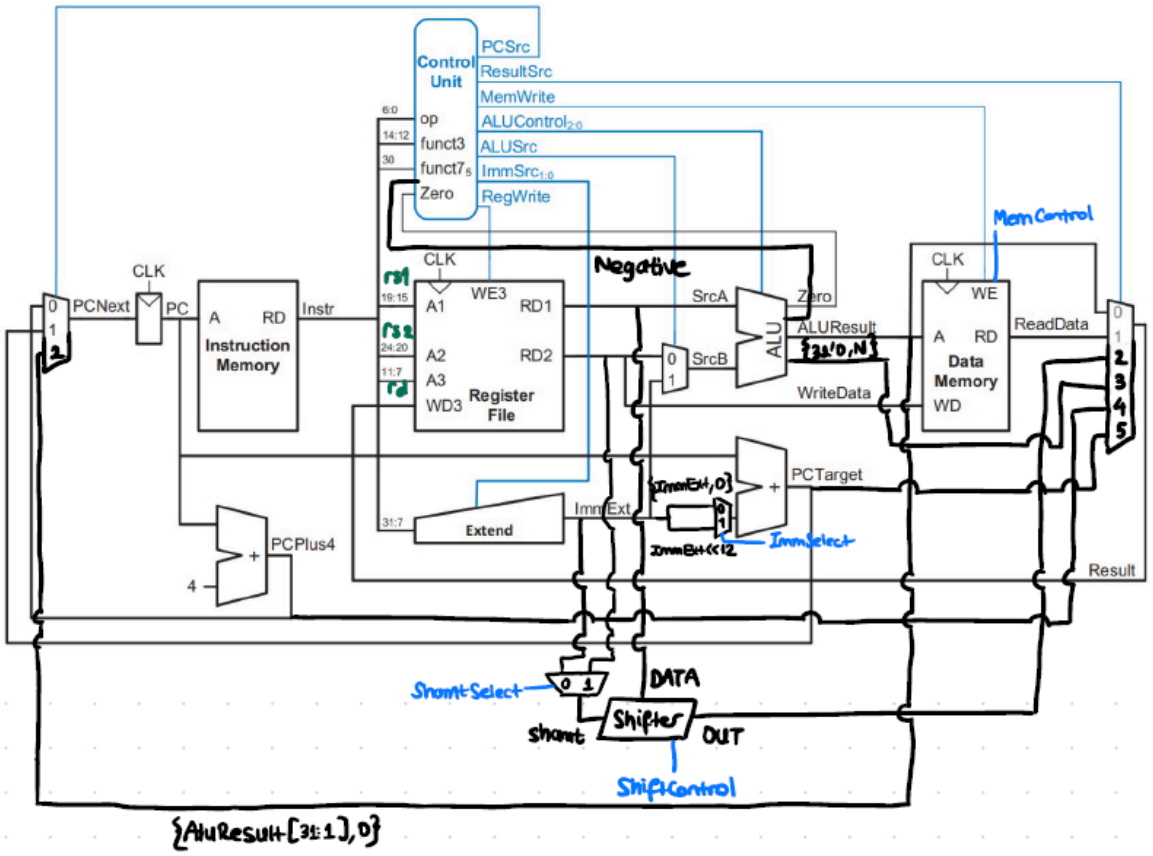


Figure 1. Our final RISC-V datapath design.

The explained datapath's RTL view composed in FPGA Design Software Quartus can be found in Figure 2.

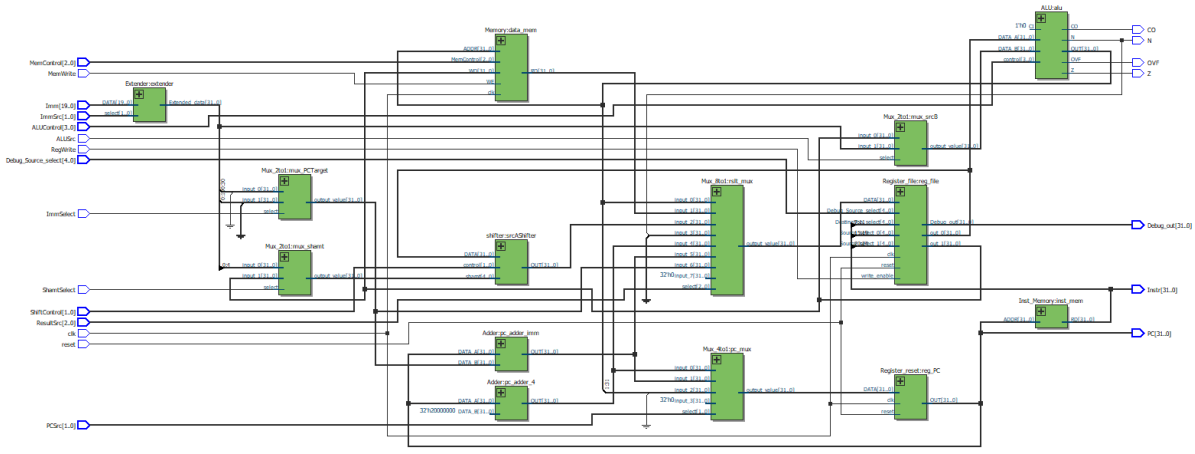


Figure 2. RTL view of our RISC-V datapath design.

Controller

Our control signals, which differ from those in the pdf file, are ShamtControl, ShiftControl, and ExtendedMuxControl. The control signals needed for all instructions can be found in Appendix 2, where each instruction (or instruction group) is explained individually.

R-Type Instructions (Opcode: 7'b0110011)

ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU:

- These instructions perform arithmetic or logical operations between two register operands.
- The ALU performs the specified operation based on funct7 and funct3 fields.
- The result is written back to the destination register.

I-Type Instructions (Opcode: 7'b0010011)

ADDI, ANDI, ORI, XORI, SLTI, SLTIU:

- These instructions perform arithmetic or logical operations between a register and an immediate value.
- The immediate value is extracted from the instruction.
- The ALU performs the operation using the register and immediate value.
- The result is written back to the destination register.

SLLI, SRLI, SRAI:

- These are immediate shift instructions.
- The shift amount is specified in the instruction.
- The ALU performs the shift operation.

- The result is written back to the destination register.

JALR:

- This is a jump-and-link register instruction.
- The target address is computed by adding an immediate value to a base register.
- The program counter (PC) is updated to this target address.
- The return address (PC + 4) is stored in the link register.

JAL (Opcode: 7'b1101111)

- This is a jump-and-link instruction.
- The target address is computed using an immediate value within the instruction.
- The PC is updated to this target address.
- The return address (PC + 4) is stored in the link register.

B-Type Instructions (Opcode: 7'b1100011)

BEQ, BNE, BLT, BGE, BLTU, BGEU:

- These are conditional branch instructions.
- The ALU compares two register values.
- Based on the comparison result and the branch condition, the PC may be updated to a target address computed using an immediate value.

L-Type Instructions (Opcode: 7'b0000011)

LB, LBU, LH, LHU, LW:

- These are load instructions.
- The effective address is computed by adding a base register and an immediate value.
- Data is read from memory at this address.
- The data is loaded into the destination register.

S-Type Instructions (Opcode: 7'b0100011)

SB, SH, SW:

- These are store instructions.
- The effective address is computed by adding a base register and an immediate value.
- Data from a source register is written to memory at this address.

AUIPC (Opcode: 7'b0010111)

- This instruction computes a target address by adding an immediate value to the current PC.

- The result is stored in the destination register.

LUI (Opcode: 7'b0110111)

- This instruction loads an immediate value into the upper 20 bits of the destination register.
- The lower 12 bits of the register are set to zero.

Our controller's final RTL view consists of 12 pages, the first page can be found in Figure 3. The rest can be found in the Appendix 1.

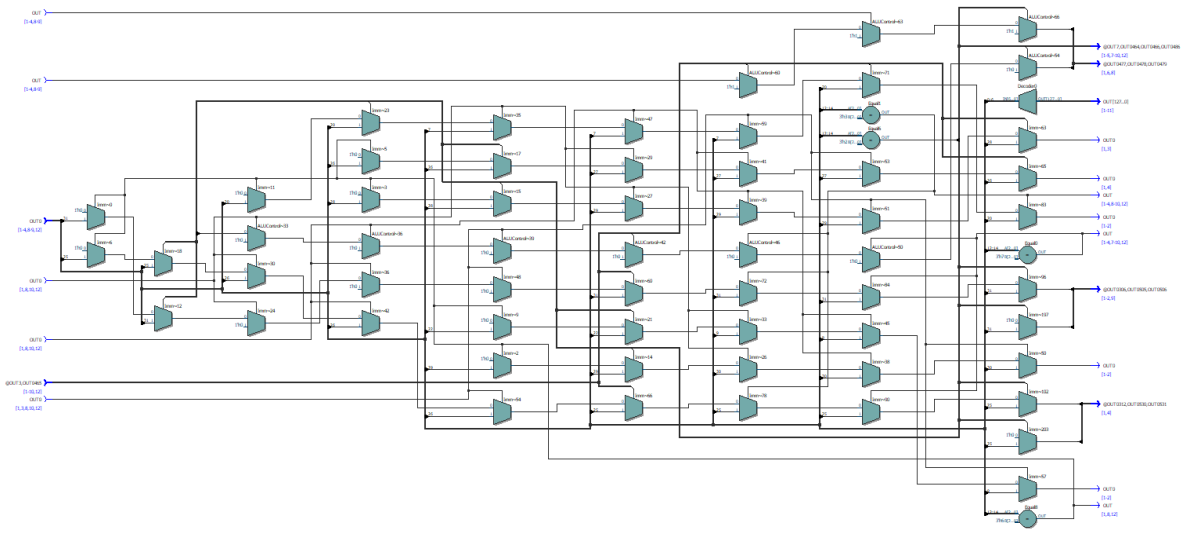


Figure 3. RTL view of the first page of our RISC-V controller design.

Testbench

Our testbench for the single-cycle RISC-V processor using Cocotb, along with the helper library for instruction handling, aims to simulate and validate the processor's behavior against a reference model. Here's an overview of the key components and functionality:

Single_Cycle_Test.py

TB Class

- **Initialization:**
  - Initializes internal state, including the program counter (PC), zero flag, register file, and memory.
  - Configures logging to track the performance model and hardware design under test (DUT).
- **log_dut:**
  - Logs the state of the DUT by calling user-defined logging functions for the datapath and controller.

- **compare_result:**
    - Compares the internal PC and register file state of the performance model against the DUT.
    - Asserts equality to ensure the DUT matches the expected behavior.
- **performance_model:**
    - Simulates the execution of instructions in the RISC-V ISA.
    - Handles different instruction types (R, I, S, B, U, J) by decoding the binary representation and executing the corresponding operation.
    - Updates the PC and register file according to the instruction semantics.
- **run_test:**
    - Executes the performance model and hardware simulation in lockstep.
    - Logs state before and after each clock cycle.
    - Compares the internal model against the DUT after each cycle.

Single_cycle_test

- **Cocotb Test:**
    - Starts a clock for the DUT.
    - Resets the DUT and reads instructions from an input file.
    - Initializes the testbench with the DUT and instruction list.
    - Runs the testbench to simulate and verify the processor's execution.

Helper_lib.py

read_file_to_list

- Reads a text file line by line into a list, useful for loading instructions from a file.

Instruction Class

- **Initialization:**
    - Parses a 32-bit RISC-V instruction from its hexadecimal representation into binary.
    - Extracts opcode, registers, funct3, funct7, and immediate values.
- **log:**
    - Logs the decoded instruction fields for debugging.

ByteAddressableMemory Class

- Simulates a simple byte-addressable memory for storing and retrieving data during simulation.

Utility Functions

- **reverse_hex_string_endiannes:**
    - Reverses the byte order of a hexadecimal string to match little-endian representation.
- **rotate_right:**
    - Rotates an integer value to the right by a specified number of bits.

- **shift_helper:**
    - Helper function for performing different types of shifts (logical, arithmetic, rotate).

Notes

- The performance model includes a comprehensive implementation of various RISC-V instructions, handling edge cases like sign extension and immediate value extraction.
- Logging and comparison functions ensure that any discrepancy between the expected and actual behavior of the DUT is promptly identified, making debugging easier.

This testbench setup provides a robust framework for validating a single-cycle RISC-V processor design, ensuring correctness through detailed logging and comparison with a performance model.

RISC-V Program

The RISC-V program we used to test our processor design can be found in Figure 4.

```
1   0x00000000:  93 00 C0 00   addi r1, r0, #12
2   0x00000004:  37 10 00 00   lui  r0, r1
3   0x00000008:  33 81 00 40   sub  r1, r1, r0
4   0x0000000c:  8B C1 00 00   xorid r3, r1
5   0x00000010:  33 22 31 00   slt  r4, r2, r3
6   0x00000014:  23 A0 30 00   sw r3, r1, #0
7   0x00000018:  83 A1 00 00   lw  r5, r1, #0
8   0x0000001c:  33 D3 12 40   sra r6, r5, r1
9   0x00000020:  B3 B3 20 00   sltu r7, r1, r2
10  0x00000024:  63 81 51 00   beq r3, r5, 2
11  0x00000028:  00 00 00 00   nop
12  0x0000002c:  63 72 41 00   bgeu r0, r2, 4
13  0x00000030:  00 00 00 00   nop
14  0x00000034:  00 00 00 00   nop
15  0x00000038:  00 00 00 00   nop
16  0x0000003c:  6F 04 20 00   jal r8, 2
17  0x00000040:  00 00 00 00   nop
18  0x00000044:  A3 92 50 00   sh r5, r1, #5
19  0x00000048:  83 84 50 00   lb  r9, r1, #5
20  0x00000052:  17 15 00 00   auipc  r10, r4, #10
```

Figure 4. RISC-V test program code.

In this program, we first begin by checking if the basic arithmetic and logical instructions work. After that, we check to see if the custom instruction we added also works properly.

Following this, we perform a series of comparisons to check the correct functionality of both signed and unsigned set-less-than operations. The program then tests memory access by storing a value in memory and subsequently loading it back to confirm the data is correctly written and read.

Shift operations are tested to ensure that the processor handles these properly. Conditional branches are included to validate the processor's ability to correctly execute branch instructions based on equality and unsigned comparisons. Jump operation is included to test the processor's handling of control flow changes.

Additional memory operations are performed to test different types of data handling, including halfword stores and byte loads. Finally, an instruction that adds an immediate value to the program counter is executed to ensure the processor correctly updates the program counter with immediate values.

This comprehensive test program is designed to cover a wide range of instructions and edge cases, ensuring that our single-cycle RISC-V processor functions correctly across different types of operations. By running this program through our testbench, we validate the correctness and robustness of our processor design.

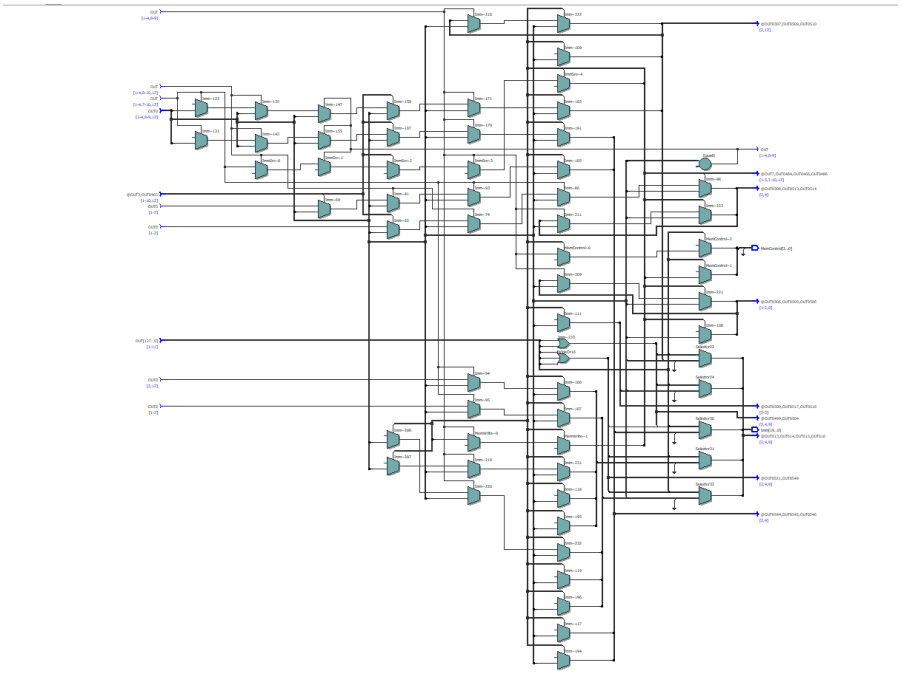
Conclusion

In conclusion, our project successfully demonstrates the design and implementation of a Single-Cycle 32-bit RISC-V Processor, showcasing the capabilities of the RISC-V ISA and its modular nature. Through the development of both the datapath and control unit, we were able to integrate an additional custom instruction, extending the base ISA. This project not only reinforced our understanding of processor architecture but also provided hands-on experience in hardware design and FPGA implementation.

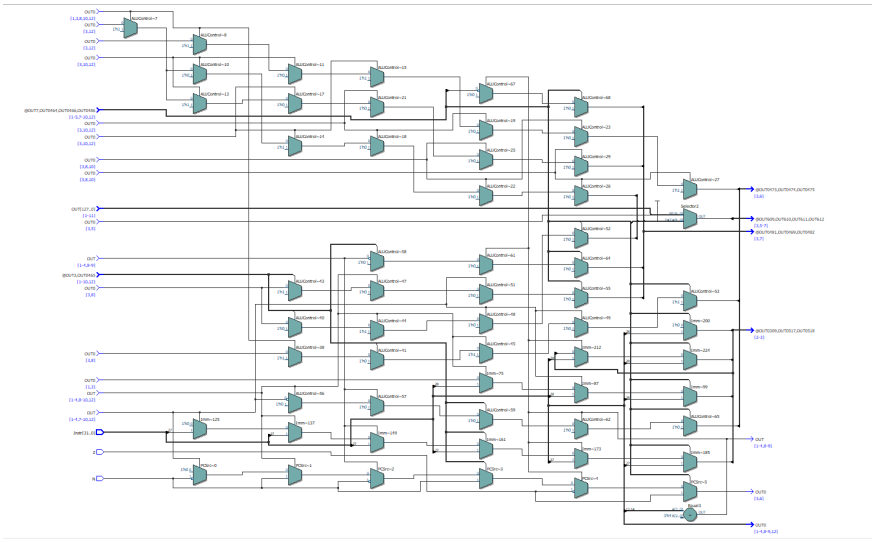
The use of Cocotb for testbench development enabled thorough validation and simulation of our processor, ensuring accurate functionality and performance. By comparing the DUT against a performance model, we ensured our design adhered to the expected RISC-V instruction set semantics. Overall, this project provided a comprehensive learning experience, equipping us with valuable skills in processor design, hardware simulation, and debugging.

Appendix 1

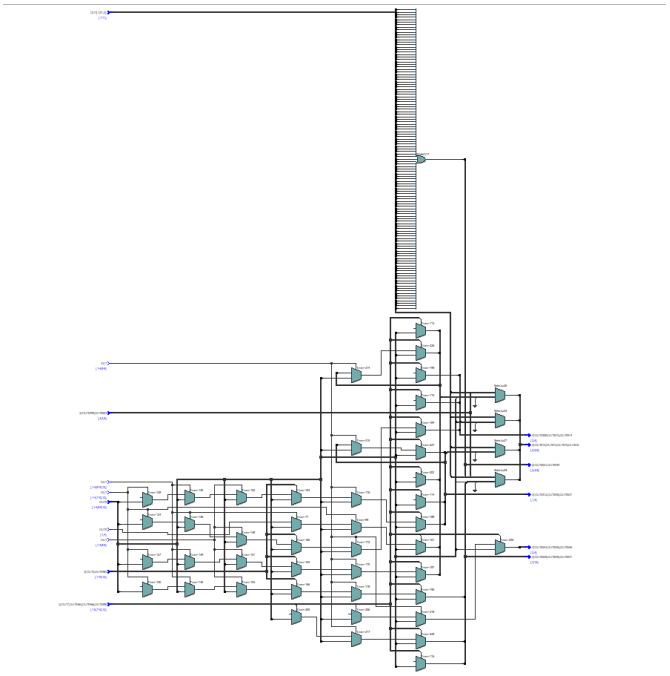
Page 2:

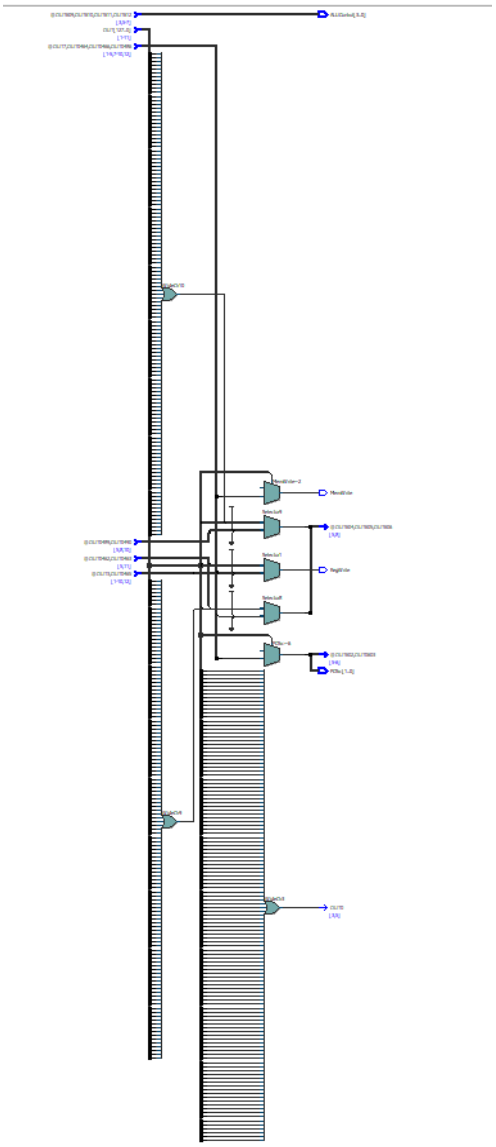


Page 3:

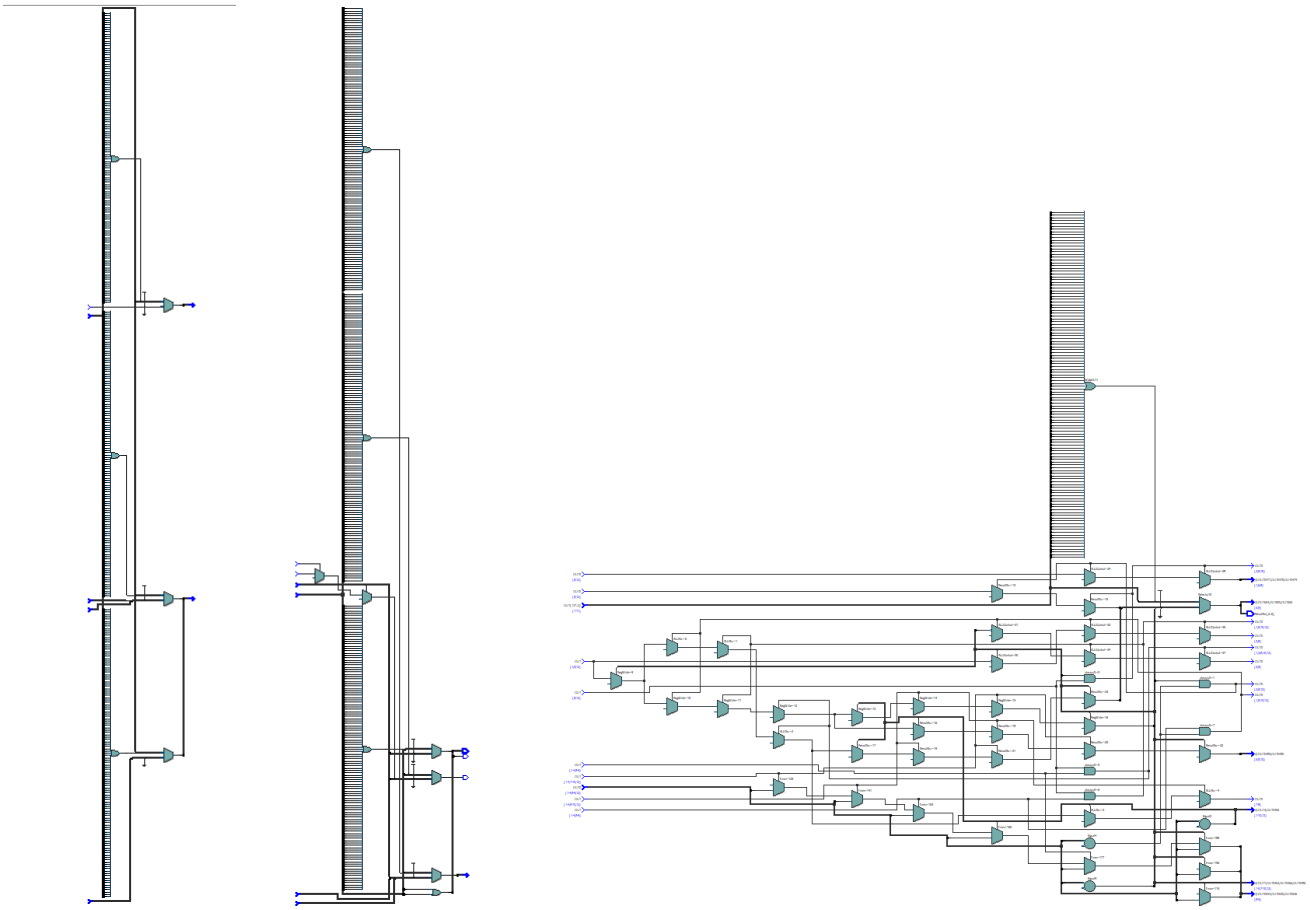


Pages 4 and 5:

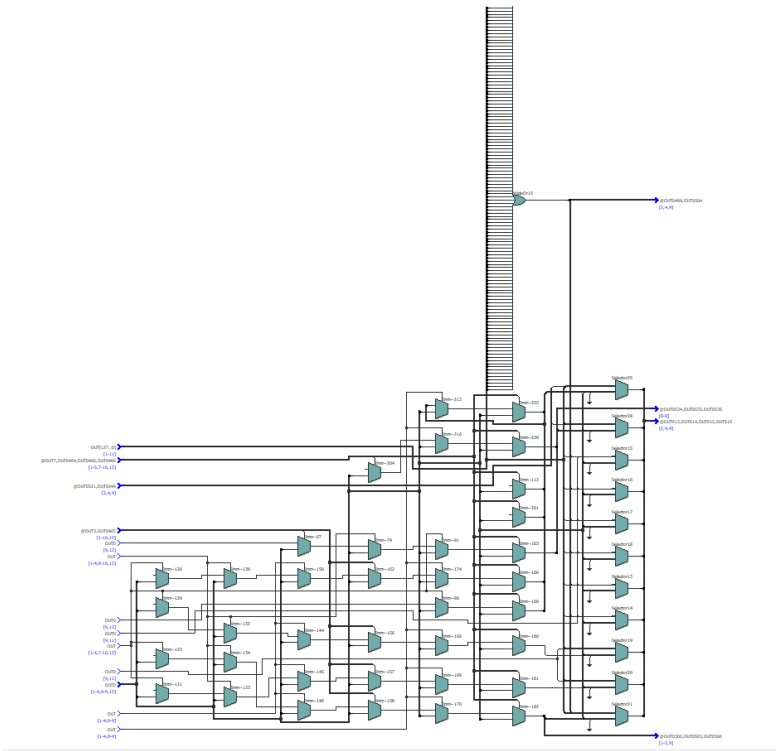




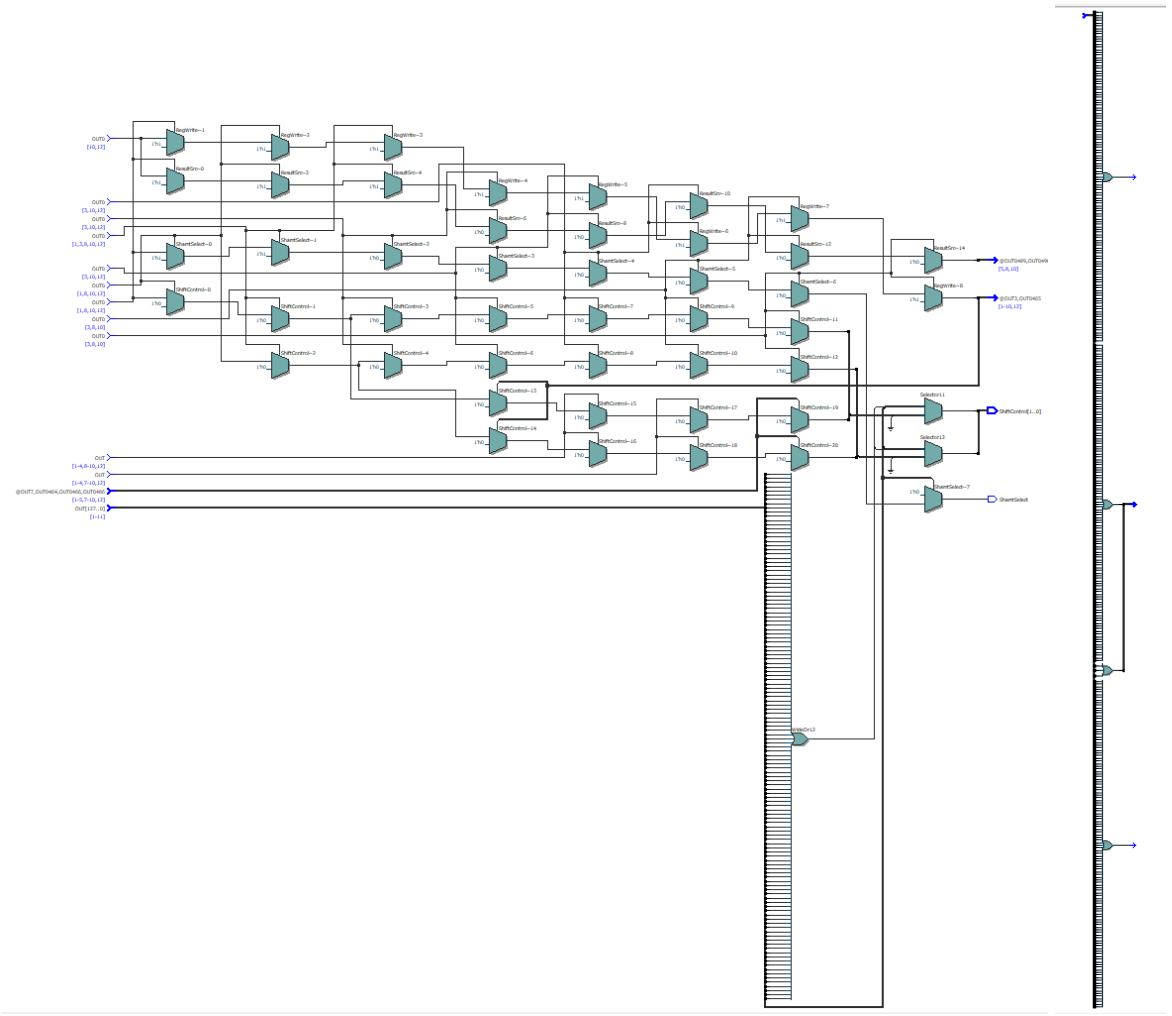
Pages 6, 7, and 8:



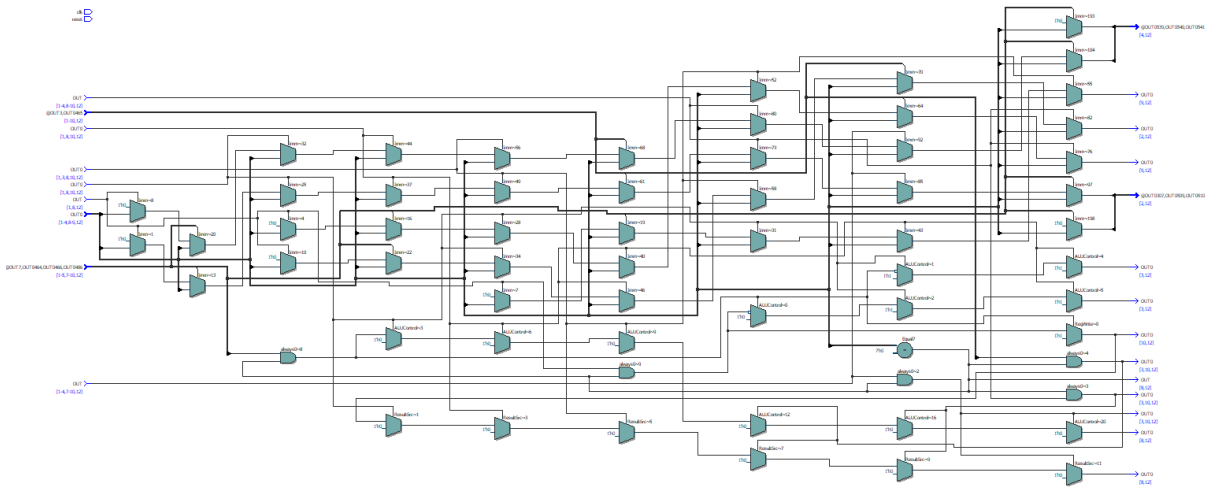
Page 9:



Pages 10 and 11:



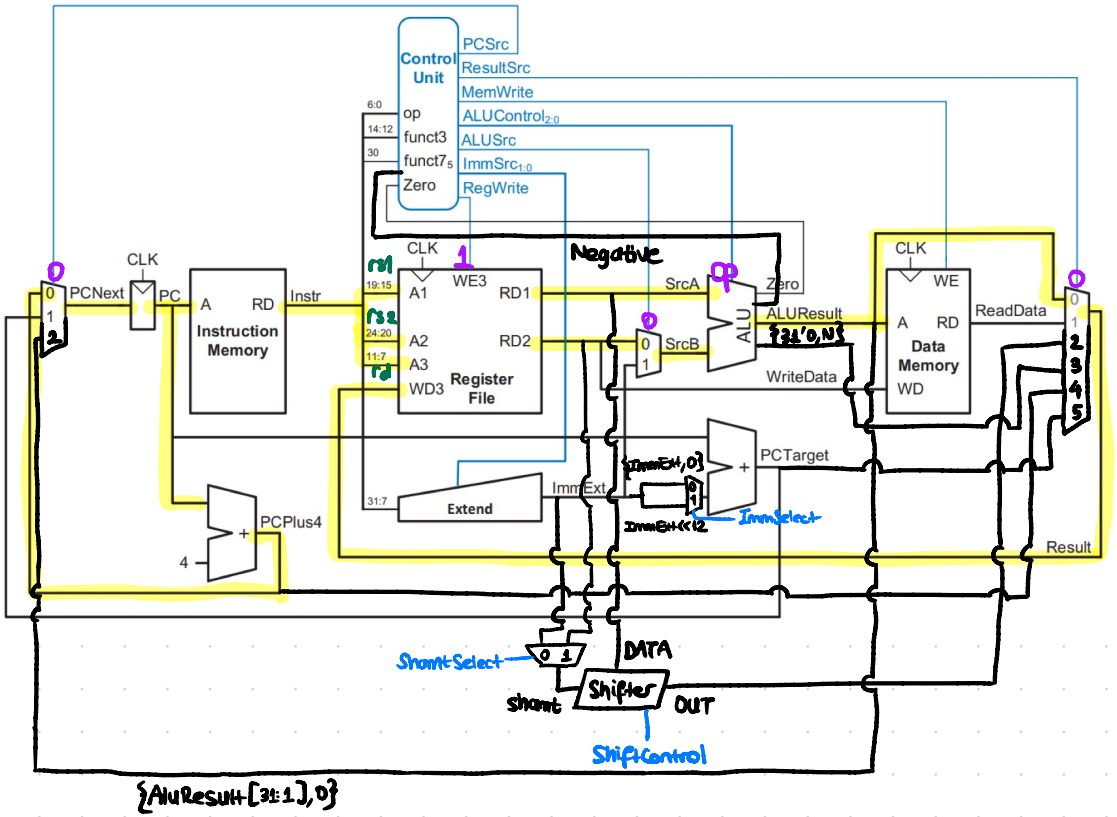
Page 12:



Appendix 2

ADD / SUB / AND / OR / XOR

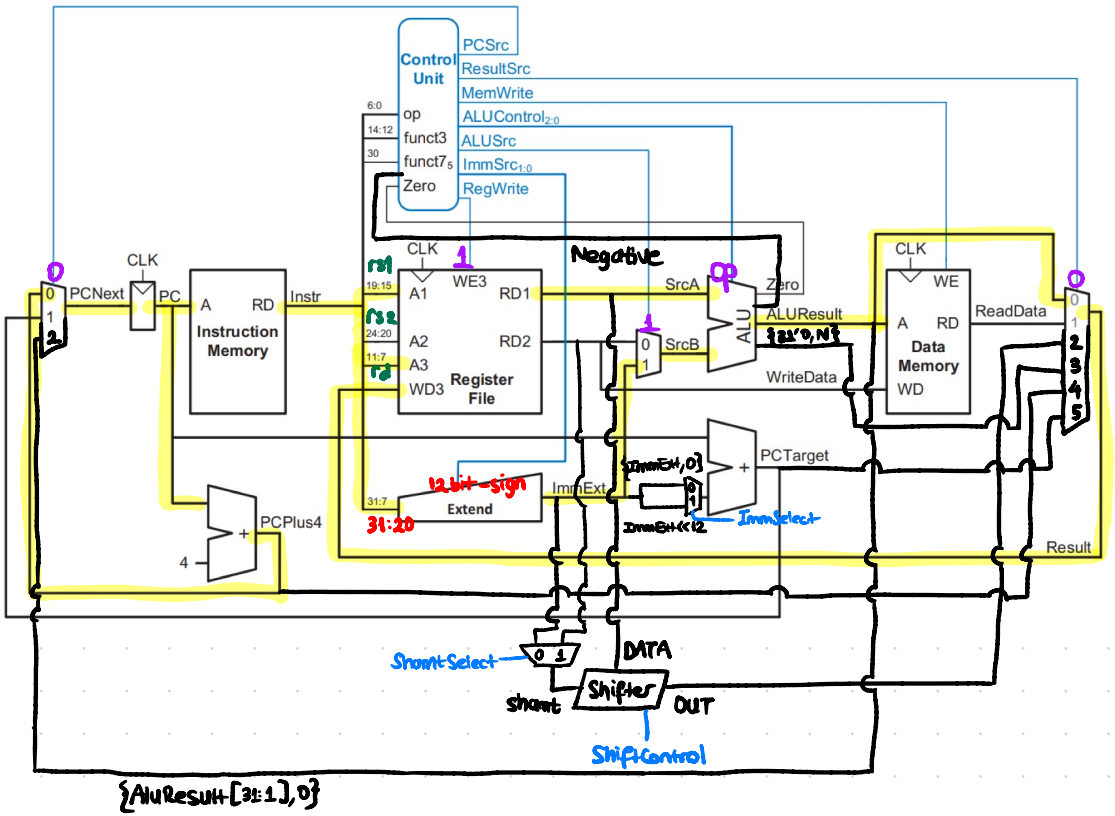
rd ← rs1 op rs2



MemControl

- 000: word
- 001: half-word, unsigned
- 010: half-word, signed
- 011: byte, unsigned
- 100: byte, signed

ADD(I) / AND(I) / OR(I) / XOR(I) rd ← rs1 op imm



MemControl

000: word

001: half-word, unsigned

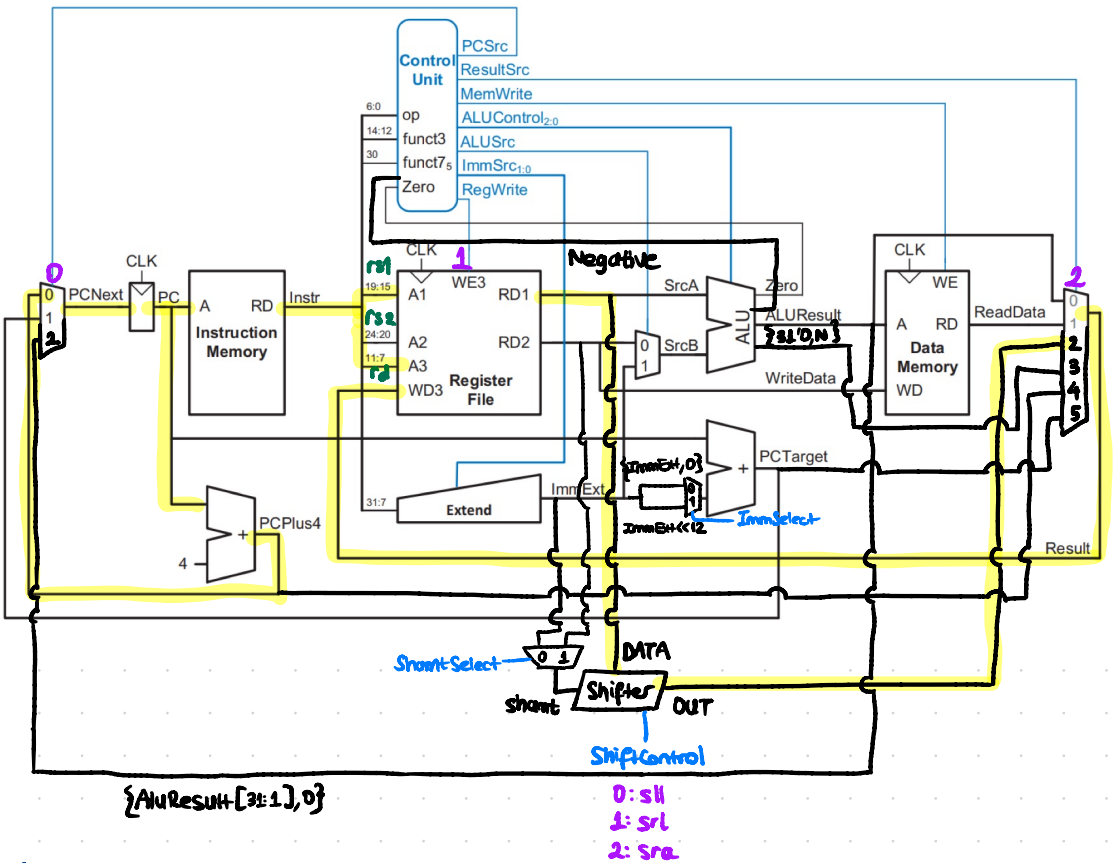
010: half-word, signed

011: byte, unsigned

100: byte, signed

SLL / SRL / SRA

rd = rs1 op rs2



MemControl

000: word

001: half-word, unsigned

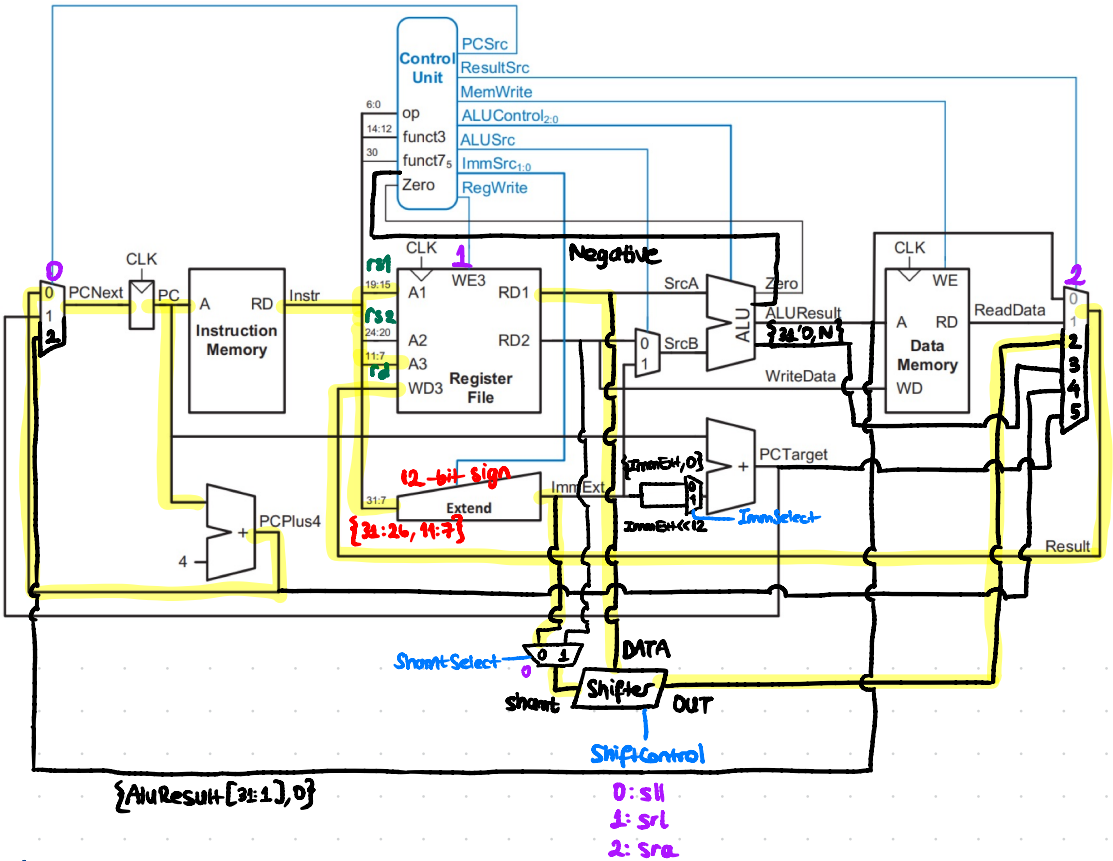
010: half-word, signed

011: byte, unsigned

100: byte, signed

SLL(I) / SRL(I) / SRA(I)

rd ← rs1 op imm

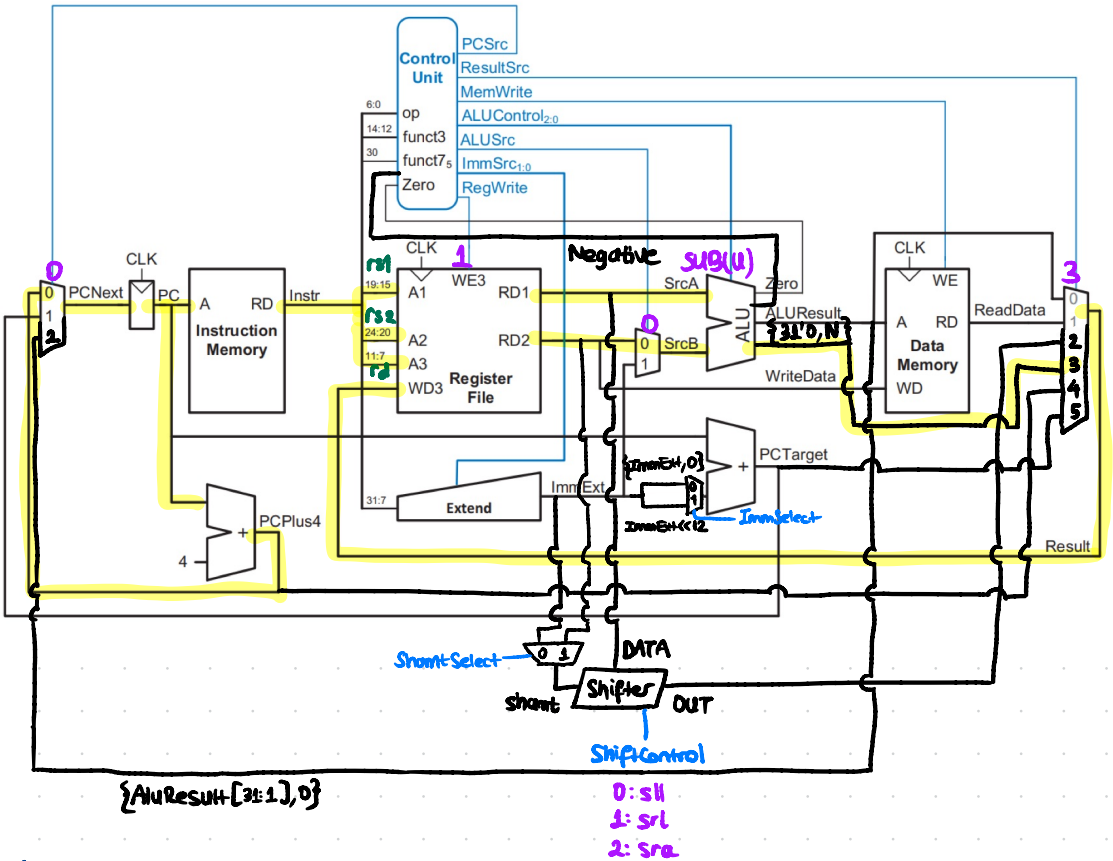


MemControl

- 000: word
- 001: half-word, unsigned
- 010: half-word, signed
- 011: byte, unsigned
- 100: byte, signed

SLT(U)

rd ← 1 if rs1 < rs2



MemControl

000: word

001: half-word, unsigned

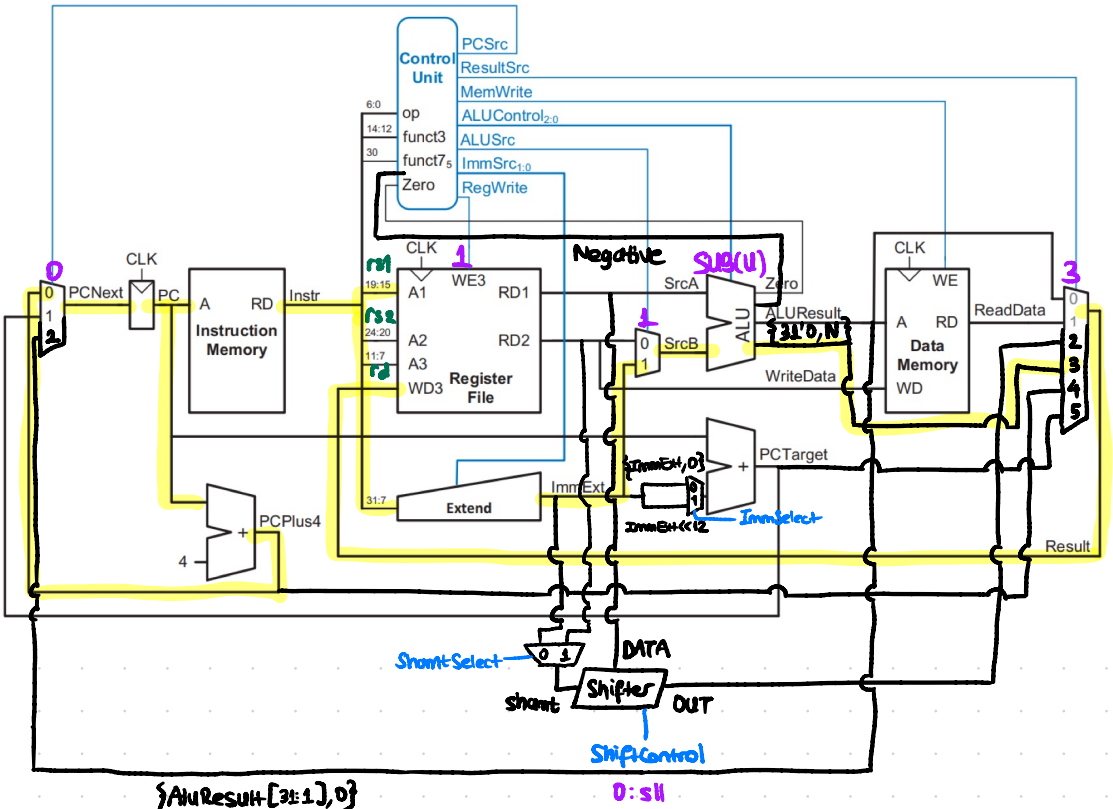
010: half-word, signed

011: byte, unsigned

100: byte, signed

SLT(I)(U)

rd ← 1 if rs1 < imm



MemControl

000: word

001: half-word, unsigned

010: half-word, signed

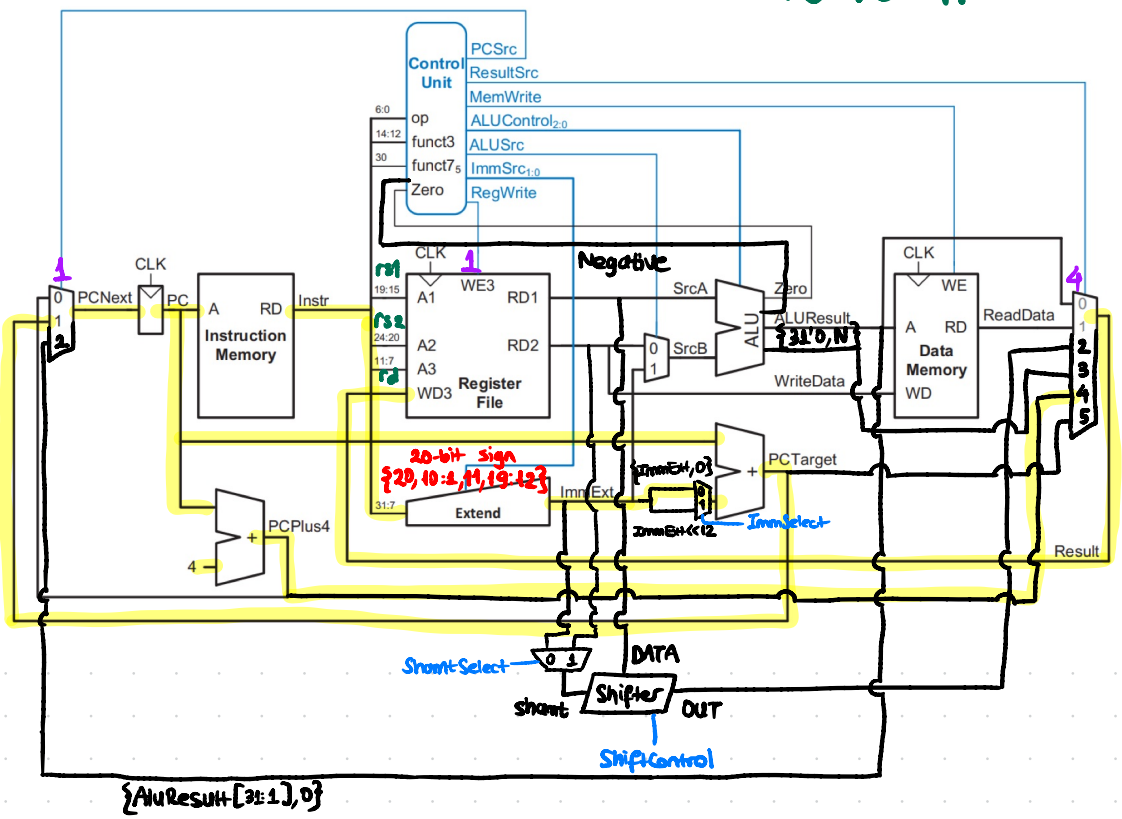
011: byte, unsigned

100: byte, signed

JAL

rd = PC+4

PC = PC+offset



MemControl

000: word

001: half-word, unsigned

010: half-word, signed

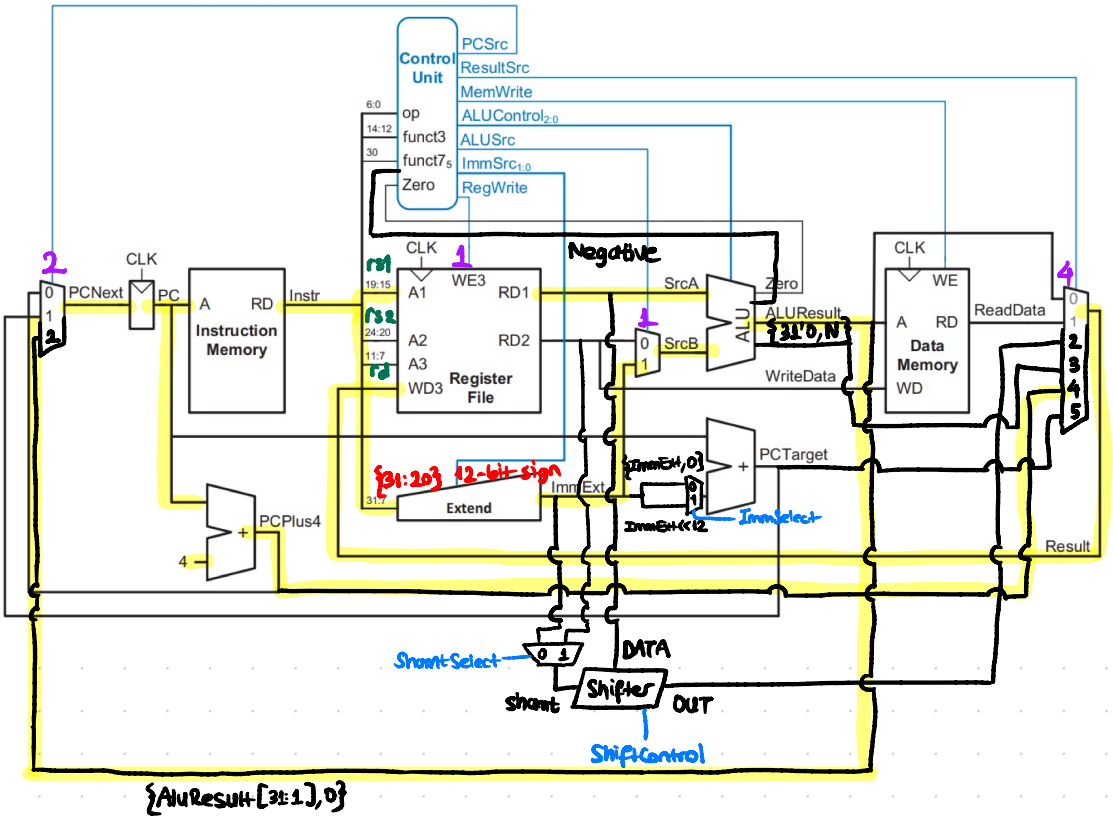
011: byte, unsigned

100: byte, signed

JALR

rd = PC+4

PC = rs1 + imm



MemControl

000: word

001: half-word, unsigned

010: half-word, signed

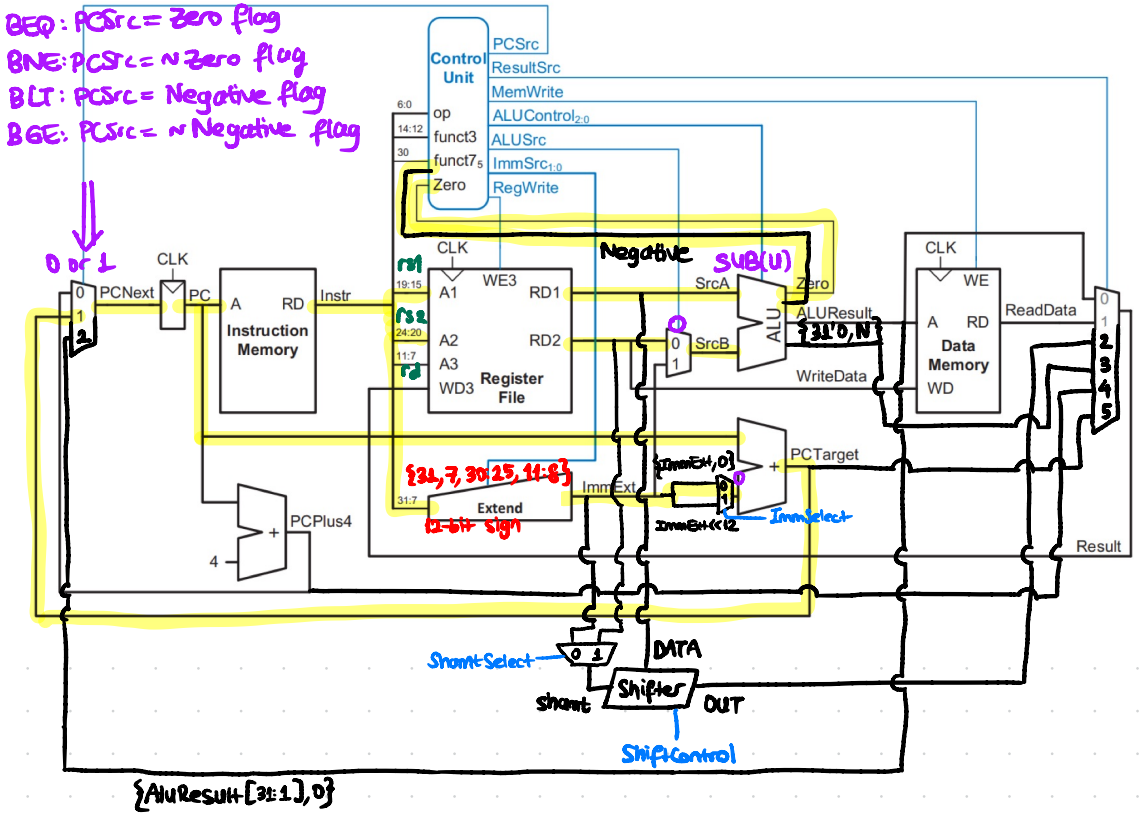
011: byte, unsigned

100: byte, signed

BEQ / BNE / BLT(U) / BGE(U)

if (rs1 == rs2)
PC = PC + offset

BEQ: PCSrc = Zero flag
BNE: PCSrc = ~Zero flag
BLT: PCSrc = Negative flag
BGE: PCSrc = ~Negative flag



MemControl

000: word

001: half-word, unsigned

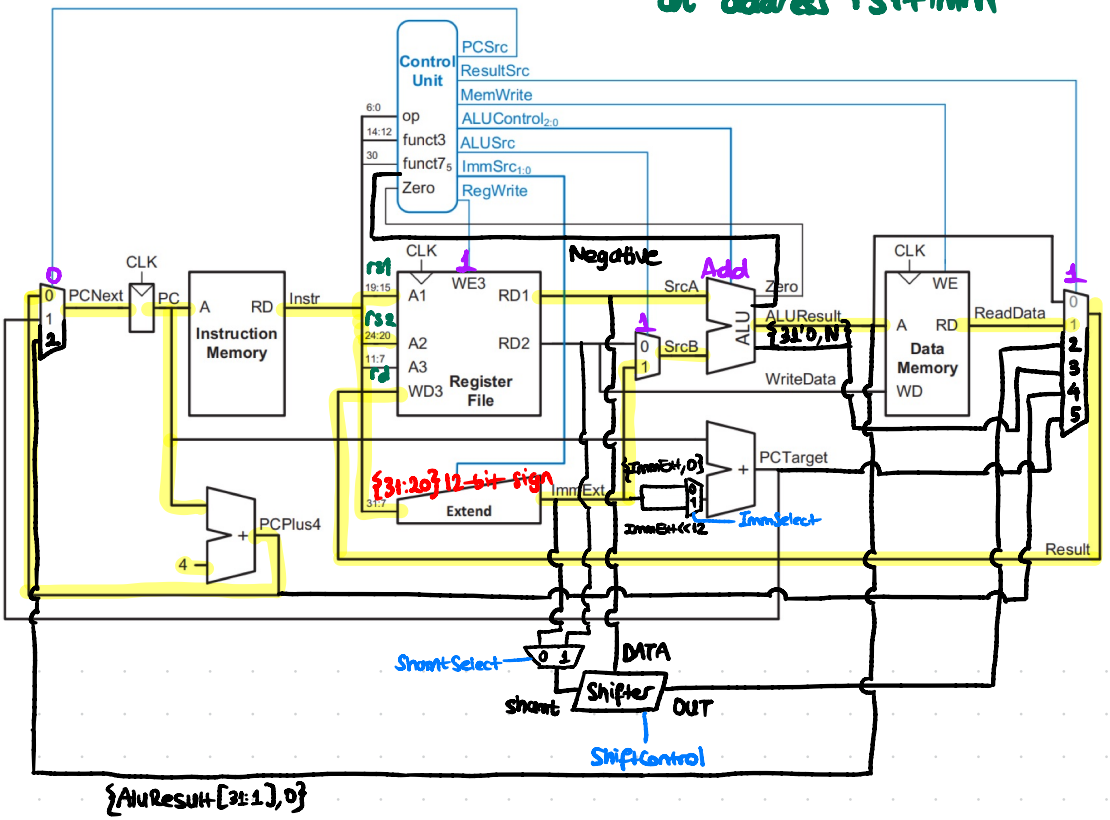
010: half-word, signed

011: byte, unsigned

100: byte, signed

LW / LH(U) / LB(U)

rd = 4 bytes of memory starting at address rs1+imm

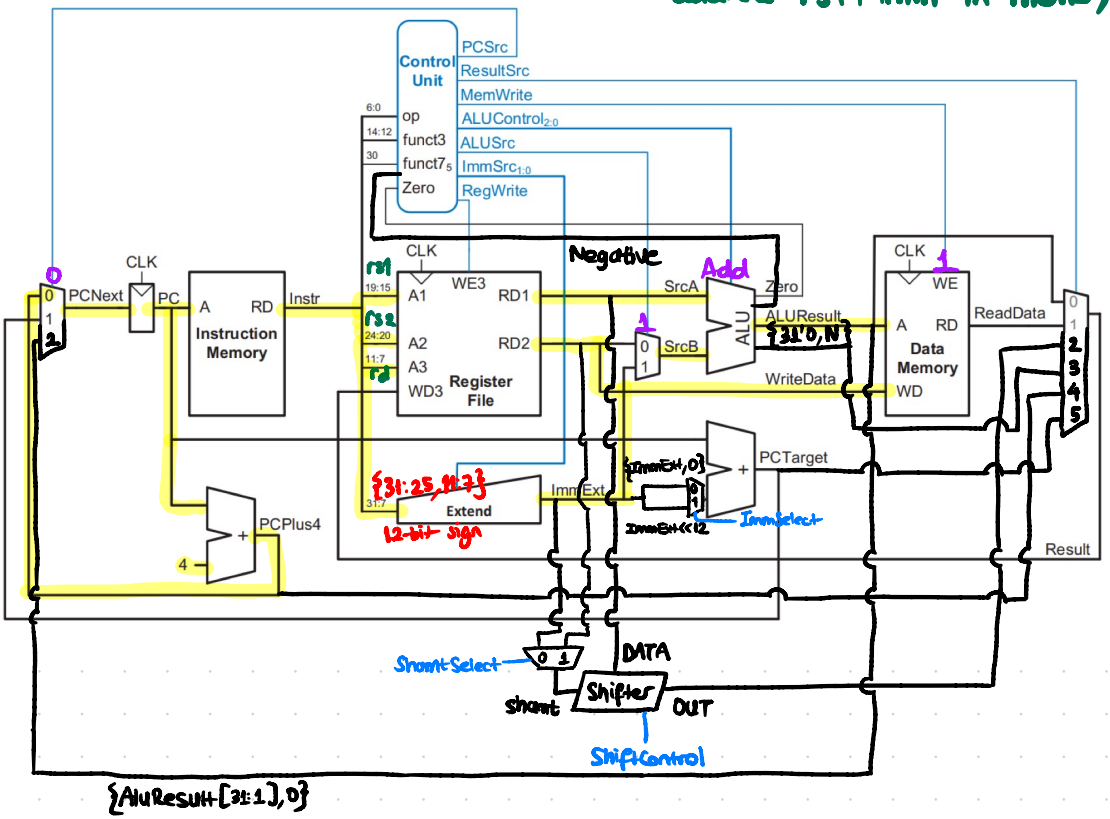


MemControl

- 000: word
- 001: half-word, unsigned
- 010: half-word, signed
- 011: byte, unsigned
- 100: byte, signed

SW / SH / SB

Stores rs2 starting at the address rs1 + imm in memory

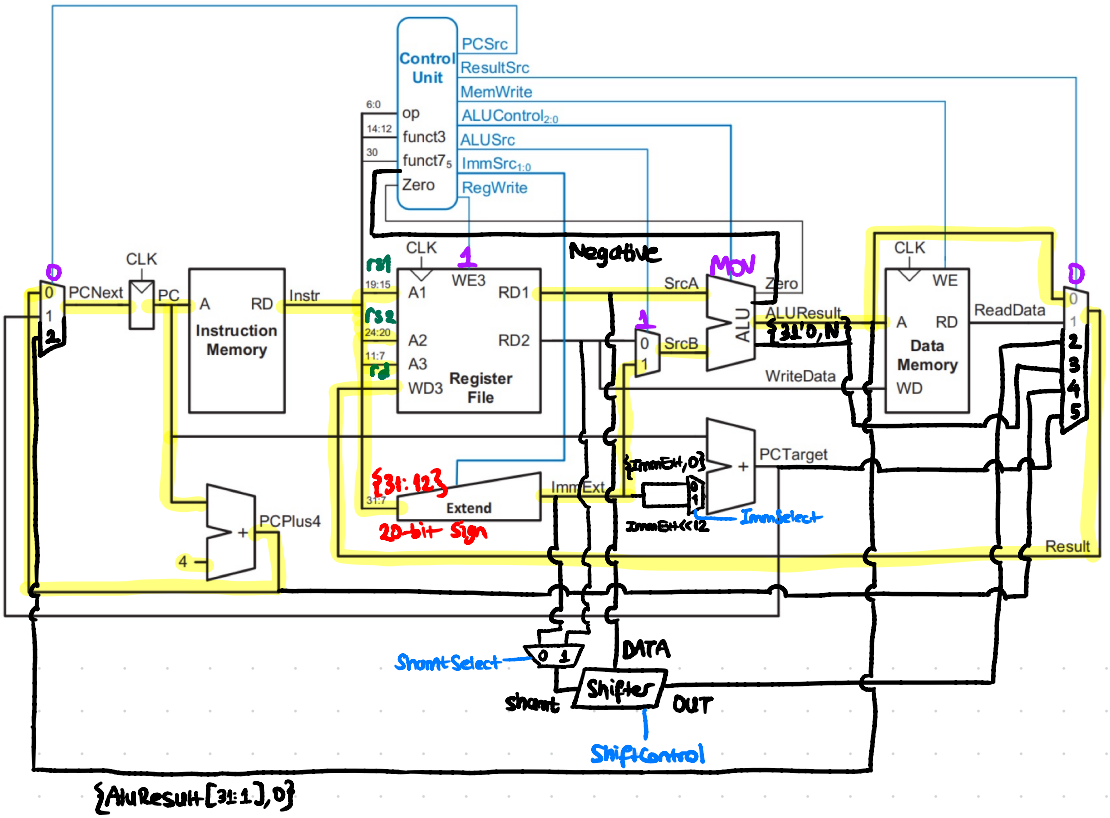


MemControl

- 000: word
- 001: half-word, unsigned
- 010: half-word, signed
- 011: byte, unsigned
- 100: byte, signed

LUI

rd = imm << 12



MemControl

000: word

001: half-word, unsigned

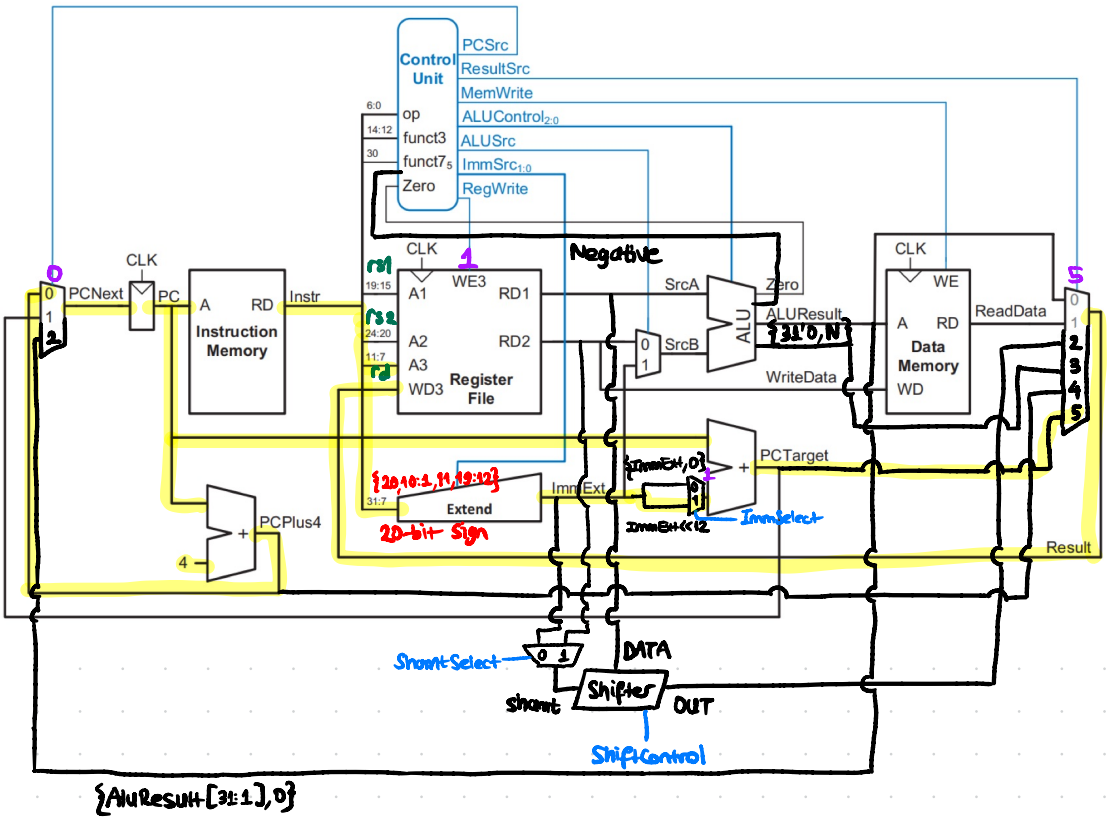
010: half-word, signed

011: byte, unsigned

100: byte, signed

AUIPC

rd = PC + (imm << 12)

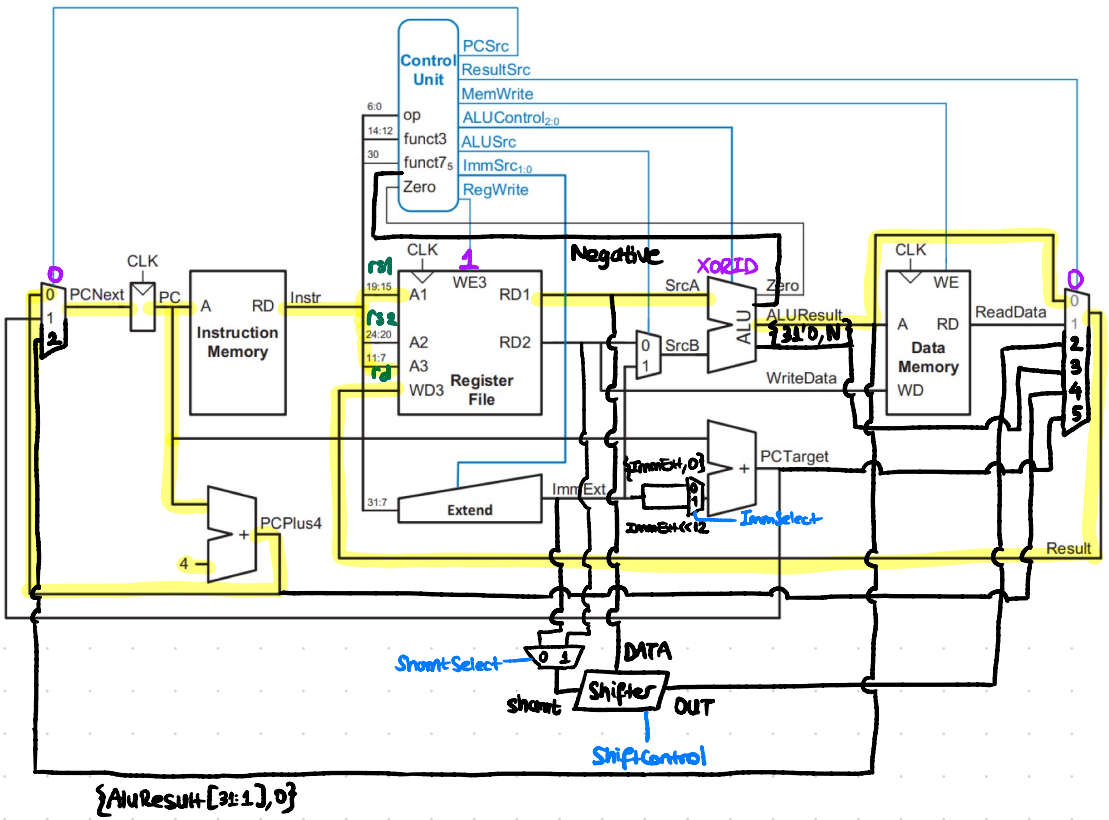


MemControl

- 000: word
- 001: half-word, unsigned
- 010: half-word, signed
- 011: byte, unsigned
- 100: byte, signed

XORID

rd←rs1 ⊕ (studentId1 ⊕ studentId2)



MemControl

000: word

001: half-word, unsigned

010: half-word, signed

011: byte, unsigned

100: byte, signed