Milestone 3: Risc-V Processor

Computer Architecture

Prof. Cherif Salama

Hussein Elazhary 900200733

Eslam Tawfik 900215295

Milestone Breakdown

In Milestone 3 of the project, we worked on pipelining our 40 supported risc v instruction datapath architecture and ensuring proper handling of hazards. This includes data, structure, and control hazards. We did so by implementing a hazard detection unit and forwarding unit in order to mitigate some of these hazards, we also worked on implementing the datapath using a single memory architecture that we used a better hazard handling paradigm for which counts as one of our 2 bonus features. The second bonus feature implemented is an automatic test generator. We also re-tested our full instructions to ensure proper functionality of the CPU.

Implemented and Tested Instructions

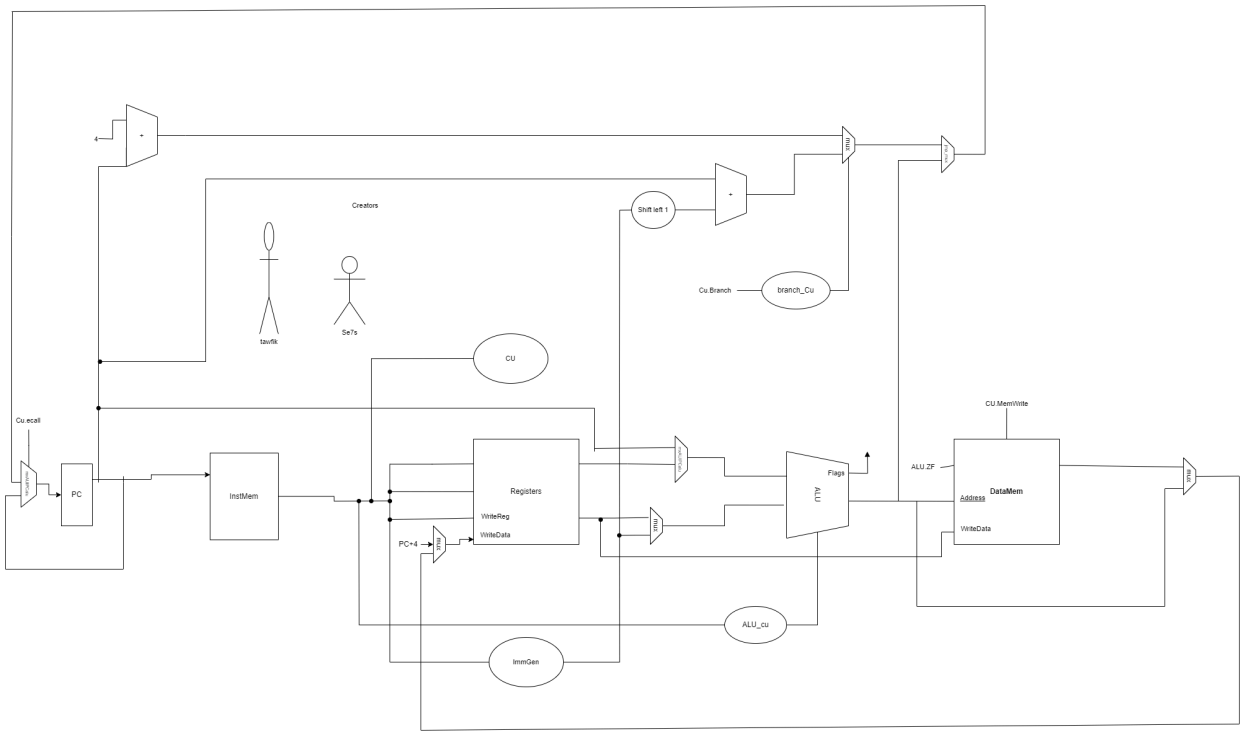
All 40 instructions have been implemented:

RV32I Base Instruction Set

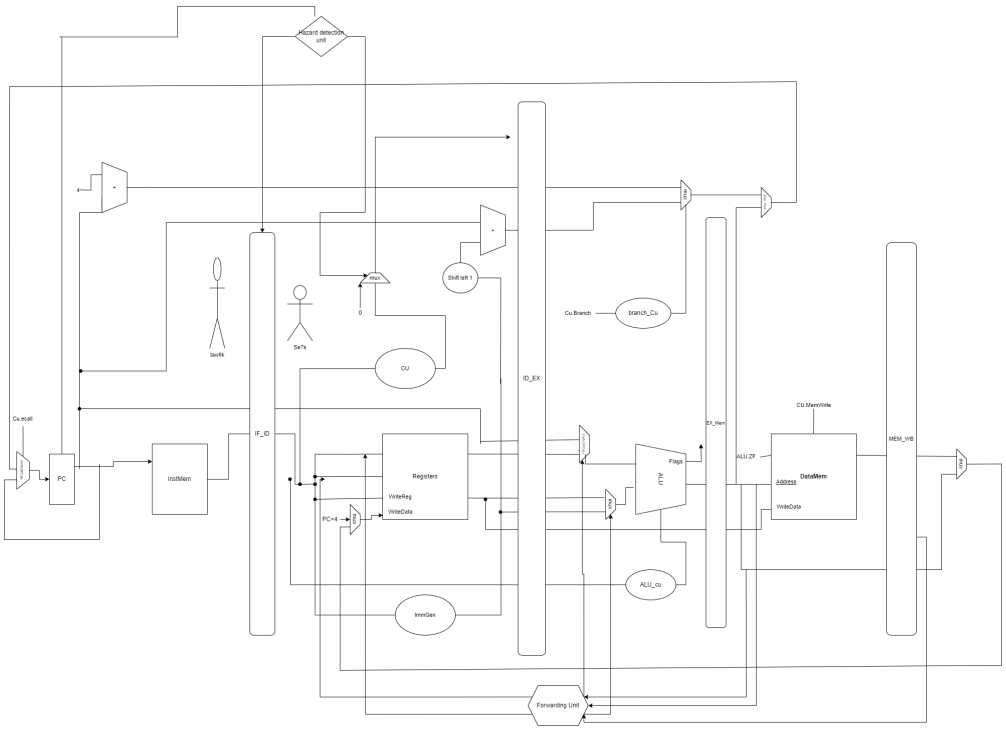
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

According to: https://riscv.org/technical/specifications/

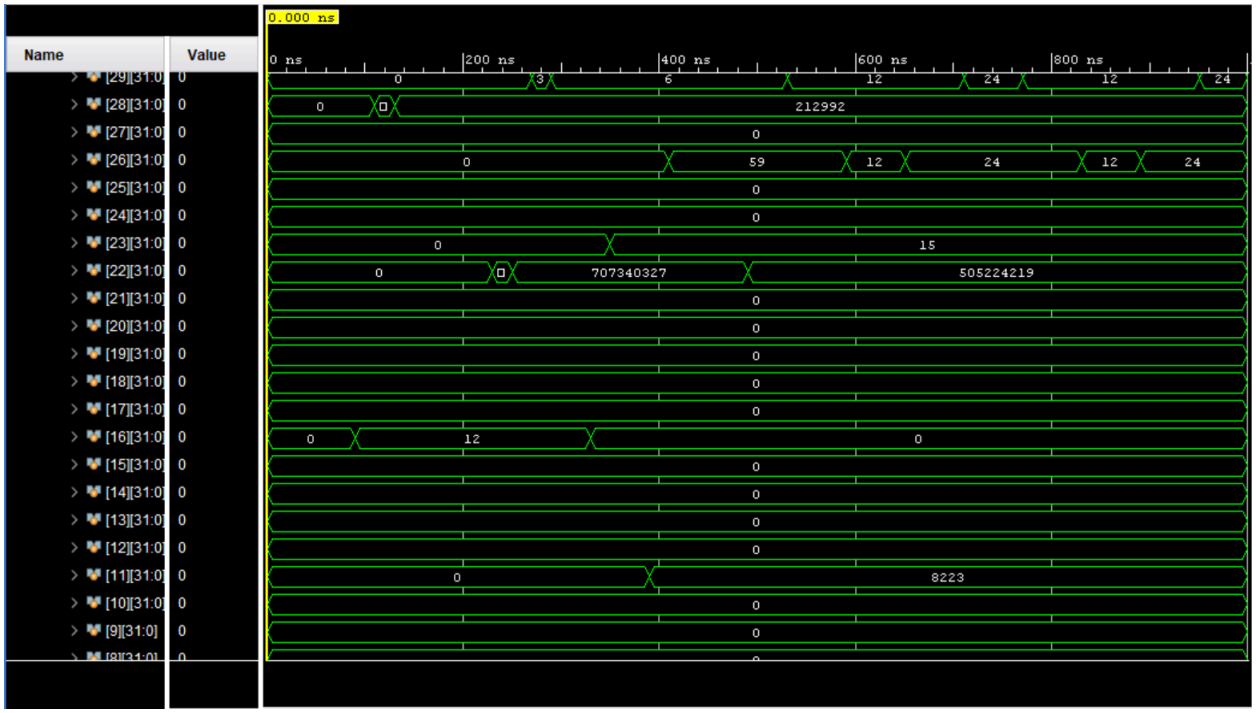
Single Cycle Datapath Block Diagram



Complete Pipelined Datapath Block Diagram with Hazard handling and Forwarding



Random Test Gen Sim



The above simulation screenshot shows the result of the random test generated. The test gen output looks like the following in the file it saves to:

File Edit View

```
mem[0] = 32'b00000000010111001110100000110011;
mem[1] = 32'b00000000000000011010011000110111;
mem[2] = 32'b00000010110000000000011010101111;
mem[3] = 32'b00000001100011010110001011100011;
mem[4] = 32'b00000001101110000010101100000011;
mem[5] = 32'b00000000000111110100011101100111;
mem[6] = 32'b00000000000000010101101100110111;
mem[7] = 32'b00000001100101100010101110000011;
mem[8] = 32'b00000000000000011100100100110111;
mem[9] = 32'b00000001011111010111001100010011;
mem[10] = 32'b00000010110100111100101110000011;
mem[11] = 32'b00000001111100101001010110000011;
mem[12] = 32'b00000010110110101100001110000011;
mem[13] = 32'b00000011000011001000111111100111;
mem[14] = 32'b00000000000000011011111100010111;
mem[15] = 32'b00000010110101100000000011100111;
mem[16] = 32'b00000001011000111011010111100011;
mem[17] = 32'b00000000000000001111011000010111;
mem[18] = 32'b00000000000000011111010011000010111;
mem[19] = 32'b00000011011101111010100010100011;
mem[20] = 32'b00000000000000001011101110010111;
mem[21] = 32'b00000010100111000100010010101100111;
mem[22] = 32'b00000000000000000000101001001011;
mem[23] = 32'b00000010110010101101001110000011;
mem[24] = 32'b00000100000011100110010110010011;
mem[25] = 32'b00000000000000011011000010001011;
mem[26] = 32'b00001000000000000000111011101111;
mem[27] = 32'b00000001000111000000111100000011;
mem[28] = 32'b00000001100111010000110010110011;
mem[29] = 32'b00000010100001010100100010000011;
mem[30] = 32'b00000011101100001000001000010011;
mem[31] = 32'b00000010001101001010010010100011;
mem[32] = 32'b00000100001000000000001101101111;
mem[33] = 32'b00000000111101100000000110010011;
mem[34] = 32'b00000001010011111011011000010011;
mem[35] = 32'b00000011000000000000010101101111;
```

The following screenshot shows the python code that generated this output:

```python
def to_binary(n, length):
    # Handle negative numbers with two's complement
    if n < 0:
        return format((1 << length) + n, '0{}b'.format(length))
    else:
        return format(n, '0{}b'.format(length))

def format_jal_immediate(binary_str):
    # Ensure the binary string is 20 bits long
    if len(binary_str) != 20:
        raise ValueError("Input must be a 20-bit binary string.")
    reordered = binary_str[0] + binary_str[10:20] + binary_str[9] + binary_str[1:9]

    return reordered

random_opcode = ["0110111", "0010111", "1101111", "1100111", "1100011", "0000011", "0100011", "0010011", "0110011", "0001111", "1110011"]
random_funct3b = ["000", "001", "011", "100", "101", "110", "111"]  # Branch
random_funct3l = ["000", "001", "010", "100", "101"]  # Load
random_funct3s = ["000", "001", "010"]  # Store
random_funct3i = ["000", "010", "011", "100", "110", "111"]  # Immediate arithmetic/logical
random_funct3r = ["000", "001", "010", "011", "100", "101", "110", "111"]  # Register arithmetic/logical


readyinputs = []
i=0
#generate up to 20 instruction program automatically
for _ in range(50):
    opcode = random.choice(random_opcode)
    # Immediate values for different instruction formats limited to 64 for simplicity
    u_imm = to_binary(random.randint(0, 64), 7)
    i_imm = to_binary(random.randint(0, 64), 12)
    b_imm = to_binary(random.randint(0, 64), 5)
    s_imm = to_binary(random.randint(0, 64), 12)
    si_imm = to_binary(random.randint(0, 64), 20)


    # Register identifiers as 5-bit binary values
    rd = to_binary(random.randint(1, 31), 5)
    rs1 = to_binary(random.randint(1, 31), 5)
    rs2 = to_binary(random.randint(1, 31), 5)
    shamt = to_binary(random.randint(1, 31), 5)  # Shift amount for shift instructions

    # Generate final instruction
    if opcode in ["0110111", "0010111"]:  # LUI, AUIPC
        final_output = "00000000000000" + u_imm + rd + opcode  # Concatenate with 13 zeros
    elif opcode == "1100111":  # JALR
        final_output = i_imm + rs1 + "000" + rd + opcode
    elif opcode == "1101111":  # JAL fixed
        final_output = str(format_jal_immediate(si_imm)) + rd + opcode
    elif opcode == "1100011":  # Branch
        funct3 = random.choice(random_funct3b)
        final_output = "0000000" + rs2 + rs1 + funct3 + b_imm + opcode
    elif opcode == "0000011":  # Load
        funct3 = random.choice(random_funct3l)
        final_output = i_imm + rs1 + funct3 + rd + opcode
    elif opcode == "0100011":  # Store
        funct3 = random.choice(random_funct3s)
        final_output = s_imm[:7] + rs2 + rs1 + funct3 + s_imm[7:] + opcode
    elif opcode == "0010011":  # Immediate arithmetic/logical
        funct3 = random.choice(random_funct3i)
        if funct3 in ["001", "101"]:  # Shift left/right
            funct7 = "0000000" if random.randint(0, 1) == 0 else "0100000"
            final_output = funct7 + shamt + rs1 + funct3 + rd + opcode
        else:
            final_output = i_imm + rs1 + funct3 + rd + opcode
    elif opcode == "0110011":  # Register arithmetic/logical
        funct3 = random.choice(random_funct3r)
        funct7 = "0000000"
        final_output = funct7 + rs2 + rs1 + funct3 + rd + opcode
    elif opcode in ["0001111", "1110011"]:  # FENCE and SYSTEM instructions
        final_output = ("00000000000000000000" if random.randint(0, 1) == 0 else "00000000000100000000") + opcode
    else:
        print(f"{opcode} Error")
        continue

    if len(final_output) == 32:
        readyinput = ("mem[" +str(i)+ "] = 32'b" + final_output+";")
        print(final_output)
        readyinputs.append(readyinput)
        i = i+1

directory_path = r"C:\\Users\\saso-\\Desktop\\uniWork\\ArchProject\\Milestone2ArchProject\\TestPrograms\\AutoTestGen"  # Use raw string to avoid escaping issues
file_name = "InstMemAuto.txt"
full_path = directory_path + "\\" + file_name



with open(full_path, 'w') as file:
    for instruction in readyinputs:
        file.write(instruction + "\n")

print("Instructions saved to "+ file_name)
```

Code breakdown:

The provided Python script is designed to automatically generate a sequence of assembly instructions for a processor simulation. It works by randomly selecting different operation codes (opcodes) and corresponding parameters such as register identifiers and immediate values. The script includes specific functions to ensure that numbers are correctly formatted in binary. Once the instruction parameters are selected and formatted as well, they are assembled into a complete binary string that represents a full machine instruction. The script repeats this process multiple times, accumulating a list of instructions, which it then formats and writes to a file such that it can be imported immediately into Vivado for testing.