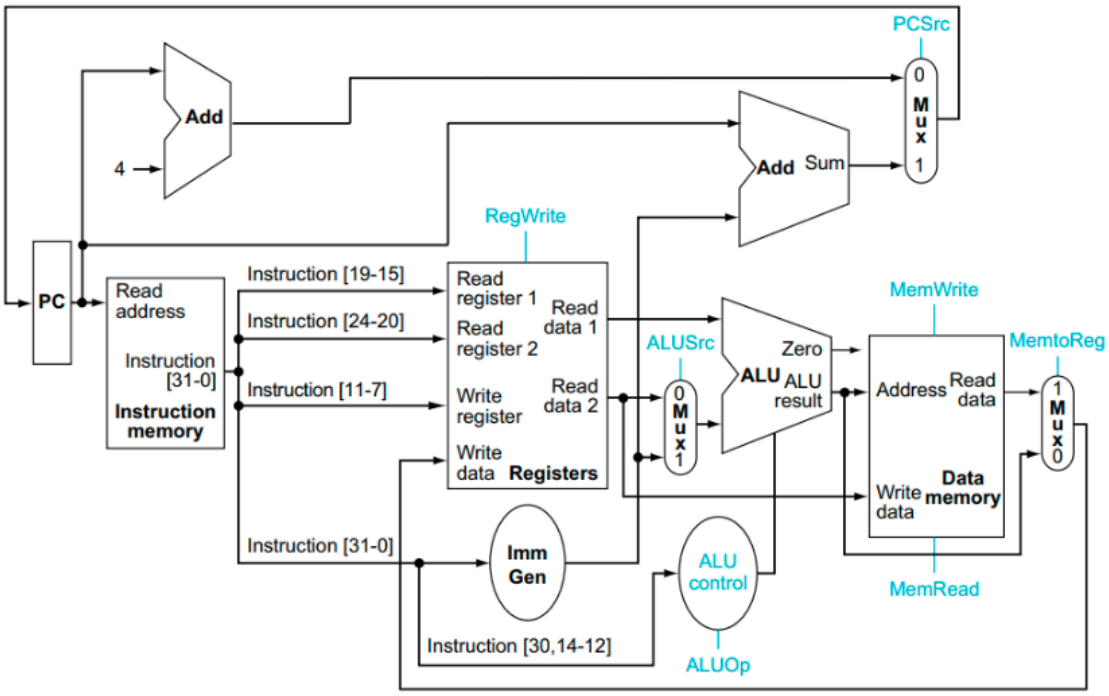
NYU Tandon School of Engineering
Computer System Architecture ECE-6913
Spring 25

Manas Nagaraj Krishna
Mk10024

1. Draw the schematic for a single stage processor and fill in your code to run the simulator.



Components that are implemented in code

- **PC: Program counter**

```
SingleStageCore(string ioDir, InsMem &imem, DataMem &dmem): Core(ioDir + "\\SS_", imem, dmem), opFilePath(ioDir + "\\StateResult_SS.txt") {
    // Initialize PC to 0
    state.IF.PC = bitset<32>(0);
    state.IF.nop = false;
    nextState = state;
}
```

Updating the PC to branch or increasing by to next instruction

```
    // Update PC
    if (take_branch) {
        nextState.IF.PC = bitset<32>(state.IF.PC.to_ulong() + imm);
    }
    else {
        nextState.IF.PC = bitset<32>(state.IF.PC.to_ulong() + 4);
    }
```

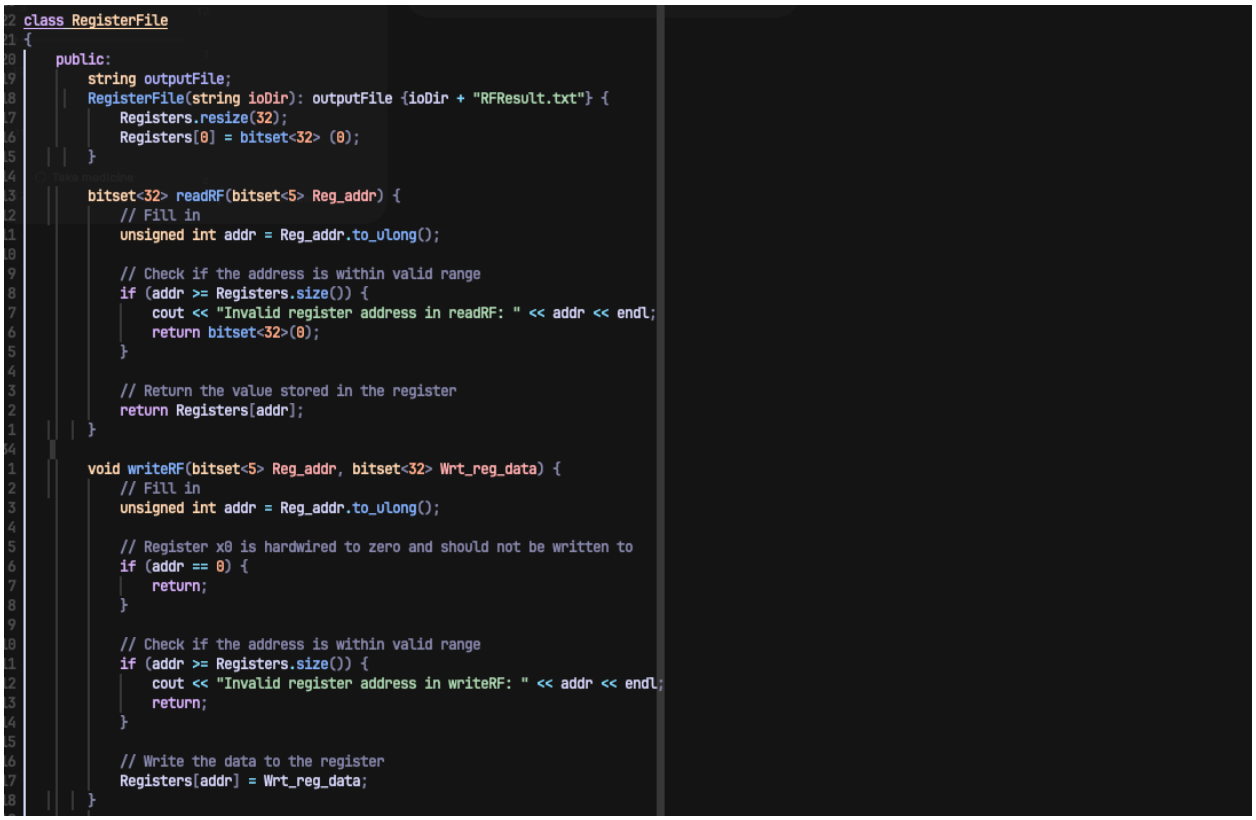
Updating PC by 4 to next instruction

```
        // Update PC
        nextState.IF.PC = bitset<32>(state.IF.PC.to_ulong() + 4);
```

Instruction Memory(Imem in code)

```
class InsMem
Alacritty
    public:
        string id, ioDir;
        InsMem(string name, string ioDir) {
            id = name;
            IMem.resize(MemSize);
            ifstream imem;
            string line;
            int i=0;
            imem.open(ioDir + "\\imem.txt");
            if (imem.is_open())
            {
                while (getline(imem,line))
                {
                    IMem[i] = bitset<8>(line);
                    i++;
                }
            }
            else cout<<"Unable to open IMEM input file.";
            imem.close();
        }

        bitset<32> readInstr(bitset<32> ReadAddress) {
            // read instruction memory
            // return bitset<32> val
            // Convert the bitset address to an unsigned integer
            unsigned int addr = (unsigned int)(ReadAddress.to_ulong());

            // Check for address overflow
            if (addr + 3 >= MemSize) {
                cout << "Address out of bounds in readInstr" << endl;
                return bitset<32>(0);
            }

            // Read 4 bytes from IMem in Big-Endian order
            bitset<8> byte0 = IMem[addr];
            bitset<8> byte1 = IMem[addr + 1];
            bitset<8> byte2 = IMem[addr + 2];
            bitset<8> byte3 = IMem[addr + 3];

            // Assemble the bytes into a 32-bit instruction
            bitset<32> instruction(0);
            instruction = (bitset<32>(byte0.to_ulong()) << 24) |
                (bitset<32>(byte1.to_ulong()) << 16) |
                (bitset<32>(byte2.to_ulong()) << 8) |
                bitset<32>(byte3.to_ulong());

            return instruction;
        }

    private:
        vector<bitset<8> > IMem;
};
```

- **Register File**: containing the 32 general purpose registers; x0 to x31



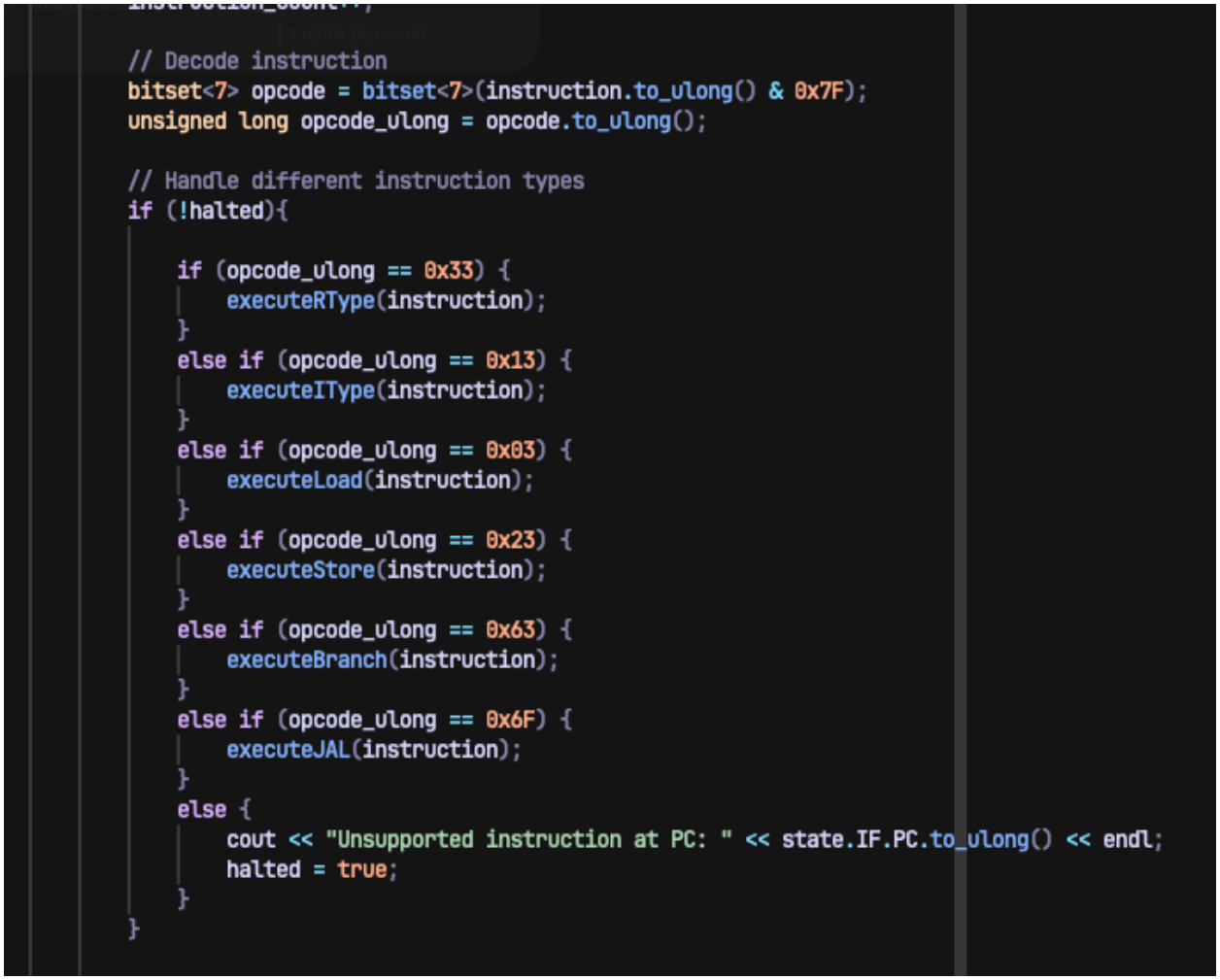
```
class RegisterFile
{
    public:
        string outputFile;
        RegisterFile(string ioDir): outputFile {ioDir + "RFResult.txt"} {
            Registers.resize(32);
            Registers[0] = bitset<32> (0);
        }

        bitset<32> readRF(bitset<5> Reg_addr) {
            // Fill in
            unsigned int addr = Reg_addr.to_ulong();

            // Check if the address is within valid range
            if (addr >= Registers.size()) {
                cout << "Invalid register address in readRF: " << addr << endl;
                return bitset<32>(0);
            }

            // Return the value stored in the register
            return Registers[addr];
        }

        void writeRF(bitset<5> Reg_addr, bitset<32> Wrt_reg_data) {
            // Fill in
            unsigned int addr = Reg_addr.to_ulong();

            // Register x0 is hardwired to zero and should not be written to
            if (addr == 0) {
                return;
            }

            // Check if the address is within valid range
            if (addr >= Registers.size()) {
                cout << "Invalid register address in writeRF: " << addr << endl;
                return;
            }

            // Write the data to the register
            Registers[addr] = Wrt_reg_data;
        }
```

Immediate generator(Imm Gen): generates the immediate value for instruction that ad hoc.

Immediate generator for load instruction

```
// Extract immediate and sign-extend
int32_t imm = (instruction.to_ulong() >> 20) & 0xFFF;
if (imm & 0x800) {
    imm |= 0xFFFFF000;
}
```

ALU: Arithmetic and logical unit processes arithmetic functions like multiplication, subtraction, division and logical operations like AND, XOR etc which are essential for the CPU



```
// Decode instruction
bitset<7> opcode = bitset<7>(instruction.to_ulong() & 0x7F);
unsigned long opcode_ulong = opcode.to_ulong();

// Handle different instruction types
if (!halted){

    if (opcode_ulong == 0x33) {
        executeRType(instruction);
    }
    else if (opcode_ulong == 0x13) {
        executeIType(instruction);
    }
    else if (opcode_ulong == 0x03) {
        executeLoad(instruction);
    }
    else if (opcode_ulong == 0x23) {
        executeStore(instruction);
    }
    else if (opcode_ulong == 0x63) {
        executeBranch(instruction);
    }
    else if (opcode_ulong == 0x6F) {
        executeJAL(instruction);
    }
    else {
        cout << "Unsupported instruction at PC: " << state.IF.PC.to_ulong() << endl;
        halted = true;
    }
}
```

Handling various functions as follow

Execute I-Type opcodes

```
void executeIType(bitset<32> instruction) {
    // Extract fields
    bitset<3> funct3 = bitset<3>((instruction.to_ulong() >> 12) & 0x7);
    bitset<5> rs1 = bitset<5>((instruction.to_ulong() >> 15) & 0x1F);
    bitset<5> rd = bitset<5>((instruction.to_ulong() >> 7) & 0x1F);

    // Extract immediate and sign-extend
    int32_t imm = (instruction.to_ulong() >> 20) & 0xFFF;
    if (imm & 0x800) {
        imm |= 0xFFFFF000;
    }

    // Read operand
    bitset<32> operand1 = myRF.readRF(rs1);

    bitset<32> result;

    // Execute operation based on funct3
    if (funct3.to_ulong() == 0x0) { // ADDI
        result = bitset<32>(operand1.to_ulong() + imm);
    }
    else if (funct3.to_ulong() == 0x4) { // XORI
        result = bitset<32>(operand1.to_ulong() ^ imm);
    }
    else if (funct3.to_ulong() == 0x6) { // ORI
        result = bitset<32>(operand1.to_ulong() | imm);
    }
    else if (funct3.to_ulong() == 0x7) { // ANDI
        result = bitset<32>(operand1.to_ulong() & imm);
    }
    else {
        cout << "Unsupported funct3 in I-type at PC: " << state.IF.PC.to_ulong() << endl;
        halted = true;
        return;
    }

    // Write result
    myRF.writeRF(rd, result);

    // Update PC
    nextState.IF.PC = bitset<32>(state.IF.PC.to_ulong() + 4);
}
```

Data memory: Used when accessing or writing data outside the general-purpose registers in the register file. It is managed in a manner similar to the InsMem class described earlier.

```
class DataMem
{
    public:
        string id, opFilePath, ioDir;
        DataMem(string name, string ioDir) : id{name}, ioDir{ioDir} {
            DMem.resize(MemSize);
            opFilePath = ioDir + "\\" + name + "_DMEMResult.txt";
            ifstream dmem;
            string line;
            int i=0;
            dmem.open(ioDir + "\\dmem.txt");
            if (dmem.is_open())
            {
                while (getline(dmem,line))
                {
                    DMem[i] = bitset<8>(line);
                    i++;
                }
            }
            else cout<<"Unable to open DMEM input file.";
                dmem.close();
        }

        bitset<32> readDataMem(bitset<32> Address) {
            // read data memory
            // return bitset<32> val
            unsigned int addr = (unsigned int)(Address.to_ulong());

            // Check for address overflow
            if (addr + 3 >= MemSize) {
                cout << "Address out of bounds in readDataMem" << endl;
                return bitset<32>(0);
            }

            // Read 4 bytes from DMem in Big-Endian order
            bitset<8> byte0 = DMem[addr];
            bitset<8> byte1 = DMem[addr + 1];
            bitset<8> byte2 = DMem[addr + 2];
            bitset<8> byte3 = DMem[addr + 3];

            // Assemble the bytes into a 32-bit data word
            bitset<32> data(0);
            data = (bitset<32>(byte0.to_ulong()) << 24) |
                (bitset<32>(byte1.to_ulong()) << 16) |
                (bitset<32>(byte2.to_ulong()) << 8) |
                bitset<32>(byte3.to_ulong());
```

2. Measure and report average CPI, Total execution cycles, and Instructions per cycle for both these cores by adding performance monitors to your code. (Submit code and print results to console or a file.) (5 points)

The code that calculates the performance metric in simulator

```
// Compute performance metrics
uint32_t total_cycles = SSCore.cycle;
uint32_t total_instructions = SSCore.instruction_count;
double average_CPI = static_cast<double>(total_cycles) / total_instructions;
double IPC = static_cast<double>(total_instructions) / total_cycles;

// Write performance metrics
ofstream perf_metrics_file;
perf_metrics_file.open(ioDir + "\\result.txt", std::ios_base::trunc);
if (perf_metrics_file.is_open()) {
    perf_metrics_file << "----------------------------Single Stage Core Performance Metrics-----------------------------" << endl;
    perf_metrics_file << "Number of cycles taken: " << total_cycles << endl;
    perf_metrics_file << "Total Number of Instructions: " << total_instructions << endl;
    perf_metrics_file << "Cycles per instruction: " << average_CPI << endl;
    perf_metrics_file << "Instructions per cycle: " << IPC << endl;
}
else {
    cout << "Unable to open result.txt for writing." << endl;
}
perf_metrics_file.close();

return 0;
```

Testcase	Performance Metrics Output

LW R1, R0, #0
LW R2, R0, #4
ADD R3, R1, R2
SW R3, R0, #8
HALT
	Number of cycles taken: 6
	Total Number of Instructions: 5
	Cycles per instruction: 1.2
	Instructions per cycle: 0.833333

LW R1, R0, #0
LW R2, R0, #4
ADD R3, R1, R2
SUB R4, R1, R2
SW R3, R0, #8
SW R4, R0, #12
AND R5, R1, R2
OR R6, R1, R2
XOR R7, R1, R2
LW R2, R0, #16
ADD R8, R1, R2
SUB R9, R1, R2
AND R10, R1, R2
OR R11, R1, R2
XOR R12, R1, R2
	Number of cycles taken: 40
	Total Number of Instructions: 39
	Cycles per instruction: 1.02564
	Instructions per cycle: 0.975001

LW R1, R0, #20
LW R2, R0, #24
ADD R13, R1, R2
SUB R14, R1, R2
AND R15, R1, R2
OR R16, R1, R2
XOR R17, R1, R2
ADDI R18, R2, #2047
ANDI R19, R2, #2047
ORI R20, R2, #2047
XORI R21, R2, #2047
ADDI R22, R1, #2047
ANDI R23, R1, #2047
ORI R24, R1, #2047
XORI R25, R1, #2047
ADDI R26, R2, #-1
ANDI R27, R2, #-1
ORI R28, R2, #-1
XORI R29, R2, #-1
ADDI R30, R1, #-1
ANDI R31, R1, #-1
ORI R31, R1, #-1
XORI R0, R1, #-1
HALT	

LW R2, R0, #4
LW R3, R0, #8
B1: ADDI R4, R4, #1
ADD R5, R4, R5
BNE R5, R3, #8
JAL R10, #12
B2: BNE R4, R2, #-16
HALT
FN: SW R4, R0, #16
SW R10, R0, #20
BEQ R0, R0, #-16
	Number of cycles taken: 28
	Total Number of Instructions: 27
	Cycles per instruction: 1.03704
	Instructions per cycle: 0.964283

3. What optimizations or features can be added to improve performance? (Extra credit 1 point)

Instruction Prefetching

a. Fetch next instruction(s) early
b. Reduces fetch delay and hides memory latency

Enhanced ALU Capabilities

a. Add support for complex operations (e.g., multiplication, division)
b. Reduces number of required instructions

Operand Forwarding (Simulated)

a. Forward result directly to dependent instruction without writing back
b. Reduces pipeline stalls caused by data hazards

Branch Prediction (Static)

a. Use simple strategies like always-taken or always-not-taken
b. Helps reduce stalls due to branches

Cache Memory Simulation (Static/Timing-Approximate)

a. Simulate simple instruction/data caches with basic hit/miss stats
b. Provides more realistic memory modeling

Branch Target Buffer (BTB)

a. Caches destination addresses of branches
b. Speeds up instruction fetch for frequent branches

Return Address Stack

a. Optimizes prediction for function calls and returns
b. Reduces misprediction penalty in call-heavy code

Pipeline Hazard Detection Unit

a. Detects and handles structural, data, and control hazards
b. Ensures safe and efficient pipelining

Basic Superscalar Execution

a. Allows multiple instructions to issue per clock cycle
b. Increases throughput

Vector/SIMD Support (RVV-lite)

a. Adds support for vector operations
b. Enables data-parallel computation like matrix operations