ECE-GY 6913 Computing Systems Architecture
Performance Modelling Project - RISC-V processor

Md Raz
N17762874
mr4425@nyu.edu

Siddharth Kandpal
N10799721
sk8944@nyu.edu

Note: This project was compiled in VS Code using the g++ compiler.

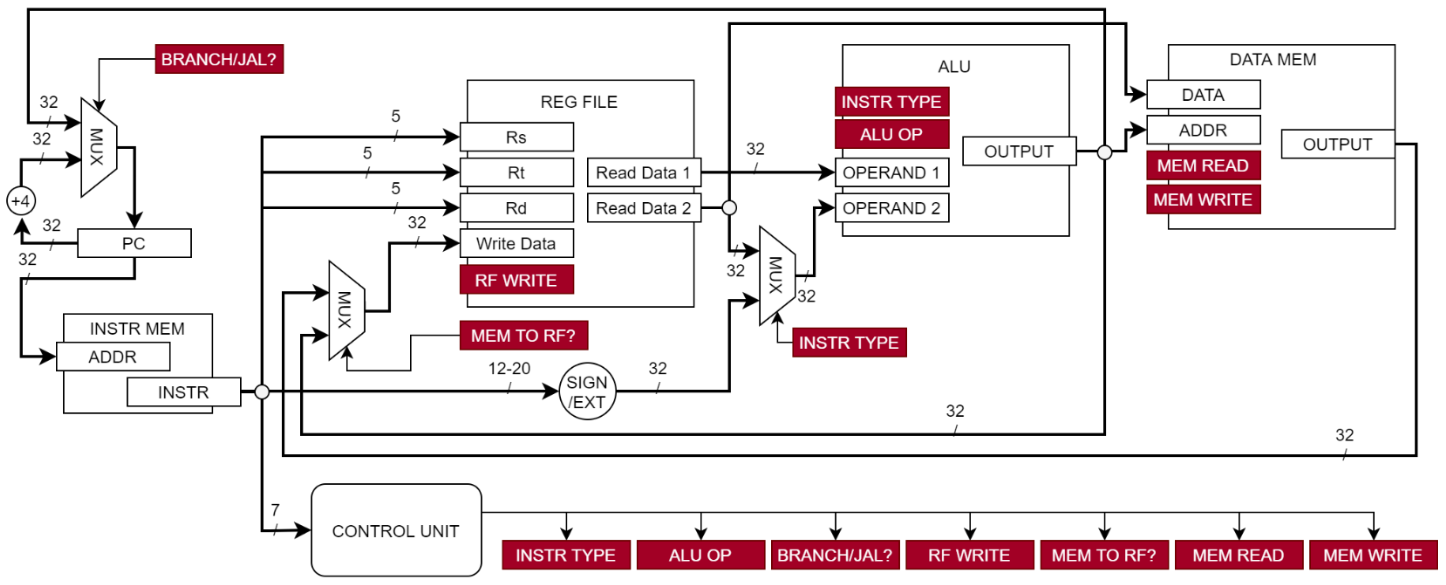
The file (`mr4425_sk8944-NYU_RV32I_6913.cpp`) is reproduced in the appendix of this report, and is given in the same directory as this report within the .zip file for verification.

Report Contents:

1. Schematic and Code

a. Single Sage

The schematic for the single cycle RISCV simulator is given below:



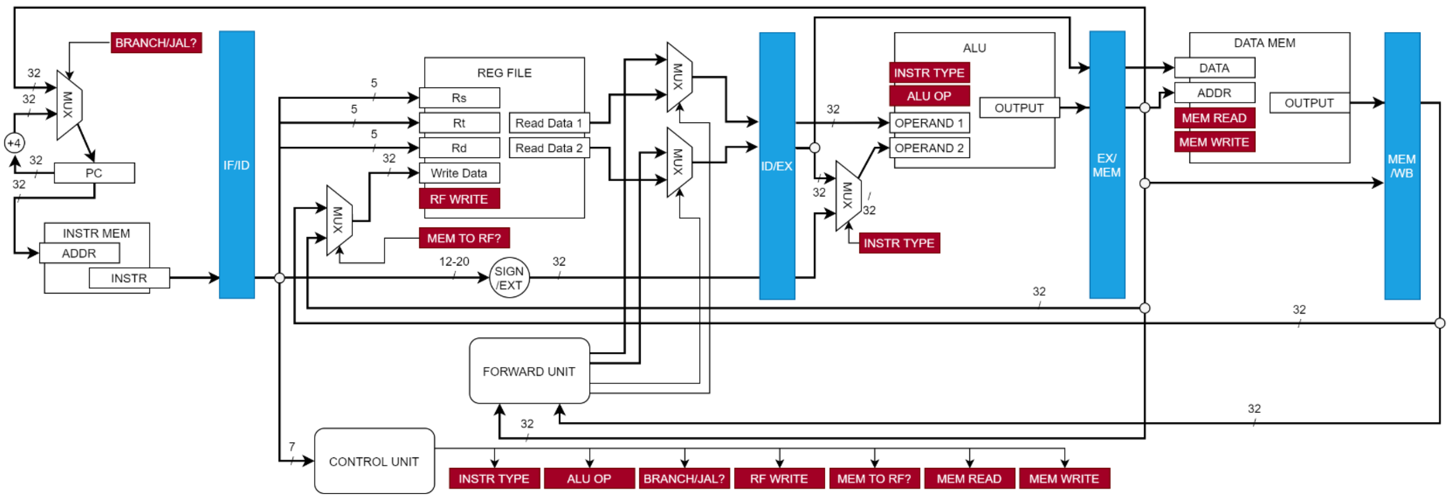
As shown above, the schematic contains a single register for the program counter, separate memories for the Instruction Memory and Data memory, a discrete register file and a discrete ALU. This implementation of the processor does not use discrete stages, as all five of the tradition processor stages are executed sequentially to complete one instruction and consume one CPU cycle. The control unit uses the fetched instruction to determine what instruction type it is, what ALU operation to execute, whether the PC should branch or Jump, and whether the memories should be read or written to. As per the schematic, all instructions which use an immediate value have the immediate value sign extended to 32 bits automatically in the equivalent instruction decode stage. Several MUXs are used, implemented as if-then statements, for switching inputs into modules as determined by the control unit. Specifically, the MUXs shown above are used to determine which value to be loaded into the PC (PC + IMM from ALU or PC + 4), whether values from the output of the Data Memory should be written to the RF file, and also whether Operand 2 for the ALU comes from the Register file or is loaded from the immediate value.

Below, the code for the single cycle implementation of the cycle accurate RISCV process is given. This is a code snippet for the `step()` function for each cycle of the single stage processor. The entirety of the code is given at the end of this report, in the appendix section, and as a separate CPP file along with this report.

```
void step() {
    /* Your implementation*/
    // We will use the given state structs above.

    if(is_first_cycle){
        state.IF.PC = (bitset<32>)0;
        state.IF.NOP =  false;
        is_first_cycle = false;
    }

    if (state.IF.NOP) {
        halted = true;
        cout << "Processor Halted\n";
    }

    if(!halted){
        // Retrieve instr
        if(debug_mode & (!halted)) cout << "\nPC =  "<<state.IF.PC.to_ulong()<<"  \t-->\t";
        state.ID.Instr = ext_imem.readInstr(state.IF.PC);

        // Decipher the instruction
        string opcode = state.ID.Instr.to_string().substr(25,7);
        string func7 = state.ID.Instr.to_string().substr(0,7);
        string func3 = state.ID.Instr.to_string().substr(17,3);

        // RS1 set
        state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
        state.EX.Read_data1 = myRF.readRF(state.EX.Rs);

        // Rs2 set
        state.EX.Rt = bitset<5>(state.ID.Instr.to_string().substr(7,5));
        state.EX.Read_data2 = myRF.readRF(state.EX.Rt);

        // Rd set
        state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));

        // Temporary imm ops
        bitset<12> temp_imm_s; bitset<12> temp_imm_b; bitset<20> temp_imm_j;

        // Opcode --> Instr type
        if      (opcode == "0110011") {state.EX.instr = r_type;}
        else if (opcode == "0010011") {state.EX.instr = i_type_imm;}
        else if (opcode == "0000011") {state.EX.instr = i_type_lw;}
        else if (opcode == "1101111") {state.EX.instr = j_type;}
        else if (opcode == "1100011") {state.EX.instr = b_type;}
        else if (opcode == "0100011") {state.EX.instr = s_type;}
        else if (opcode == "1111111") {state.EX.instr = halt;}

        if(state.EX.instr != halt){
            state.IF.PC = (bitset<32>(state.IF.PC.to_ulong() + 4)); // Premptivly Pc = PC + 4
            instr_count++;
        }

        switch(state.EX.instr){

            case r_type: if(debug_mode) cout << "Executing R type instr\n";
                // Control signal set
                state.EX.is_I_type = false;

                // Mem control set
                state.EX.rd_mem = false;
                state.EX.wrt_mem = false;

                //Set alu op
                if (func7 == "0100000") {state.EX.alu_op = SUB;
                } else {
                    if (func3 == "100") { state.EX.alu_op = XOR;
                    } else if (func3 == "110") { state.EX.alu_op = OR;
                    } else if (func3 == "111") { state.EX.alu_op = AND;
                    } else { state.EX.alu_op = ADD;
                    }
            }

            // Write to Rd?
            state.EX.wrt_enable = true;

            break;

        case i_type_imm: if(debug_mode) cout << "Executing I IMM type instr\n";
            // Control signal set
            state.EX.is_I_type = true;  state.EX.rd_mem = false; state.EX.wrt_mem = false;

            //Set alu op
            if (       func3 == "100") { state.EX.alu_op = XORI;
            } else if (func3 == "110") { state.EX.alu_op = ORI;
            } else if (func3 == "111") { state.EX.alu_op = ANDI;
            } else { state.EX.alu_op = ADDI;
            }

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                            : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

            // Rd set
            state.EX.wrt_enable = true;
            break;

        case i_type_lw: if(debug_mode) cout << "Executing I LW type instr\n";
            // Control signal set
            state.EX.is_I_type = true;
            state.EX.rd_mem = true; // We will write to mem
            state.EX.wrt_mem = false; // we will not read from mem
            state.EX.alu_op = ADDI;

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())?
                             ((bitset<32>)(string(20,'1') +  state.ID.Instr.to_string().substr(0,12)))
                            : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

            // Rd set
            state.EX.wrt_enable = true;
            break;

        case j_type: if(debug_mode) cout << "Executing J type instr\n";
            // Control signal set
            state.EX.is_I_type = false;
            state.EX.rd_mem    = false; // We will write to mem
            state.EX.wrt_mem   = false; // we will not read from mem

            // Set ALU OP
            state.EX.alu_op = ADDI;

            // Descramble the 20 bit IMM<20|10:1|11|19:12>
            temp_imm_j = (bitset<20>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 20
                              state.ID.Instr.to_string().substr(12, 8) + // Bit 19:12
                              state.ID.Instr.to_string().substr(11, 1) + // bit 11
                              state.ID.Instr.to_string().substr(1, 10)    ).to_ulong()); // bits 10:1

            temp_imm_j <<= 1;

            //Sign extend and load the immediate
            state.EX.Imm =    (((bitset<1>)(temp_imm_j.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(12,'1') + temp_imm_j.to_string().substr(0,20)))
                            : ((bitset<32>)(temp_imm_j.to_string().substr(0,20)));

            state.EX.Read_data1 = state.IF.PC;  //RS1 = Current PC + 4

            state.EX.wrt_enable = true;

            // Discard instruction fetched this cycle, update PC
            state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
            cout<<"Jump and Link Taken\n";
            break;

        case b_type: if(debug_mode) cout << "Executing B type instr\n";
            // Control signal set
            state.EX.is_I_type = false;
            state.EX.rd_mem    = false; // We will write to mem
            state.EX.wrt_mem   = false; // we will not read from mem
            state.EX.wrt_enable = false;

            // Descramble the 12 bit IMM<12|10:5>   <4:1|11>
            // The imm exists in instruction bits 31:25, 11:7,  --> 7 bits, 5 bits
            temp_imm_b = (bitset<12>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 12
                                    state.ID.Instr.to_string().substr(24, 1) + // Bit 11
                                    state.ID.Instr.to_string().substr(1, 6) + // Bits 10:5
                                    state.ID.Instr.to_string().substr(20, 4)    ).to_ulong()); // bits 4:1

            temp_imm_b <<= 1;

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(temp_imm_b.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(20,'1') + temp_imm_b.to_string().substr(0,12)))
                            : ((bitset<32>)(temp_imm_b.to_string().substr(0,12)));

            cout<<"RS1 = "<<(int)state.EX.Read_data1.to_ulong()<<", RS2 = “
                                  <<(int)state.EX.Read_data2.to_ulong()<<"\n";

            // We will resolve the branch here:
            if ((func3 == "000") & (state.EX.Read_data1 == state.EX.Read_data2)) { //BEQ
                // Discard instruction fetched this cycle, update PC
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                cout<<"Branch Taken: BEQ\n";

            } else if ((func3 == "001") & (state.EX.Read_data1 != state.EX.Read_data2)) { //BNE
                // Discard instruction fetched this cycle, update PC & instr from prev cycle
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                cout<<"Branch Taken: BNE\n";

            } else {
                cout << "No Branch Taken\n";
            }

            break;

        case s_type: if(debug_mode) cout << "Executing S type instr\n";
            // Control signal set
            state.EX.is_I_type = false;
            state.EX.rd_mem    = false;
            state.EX.wrt_mem   = true;
            state.EX.wrt_enable = false;

            // Set ALU OP
            state.EX.alu_op = ADDI;

            // Descramble the 12 bit IMM<11:5>  <4:0>
            temp_imm_s = (bitset<12> (state.ID.Instr.to_string().substr(0, 7) + // Bits 11:5
                             state.ID.Instr.to_string().substr(20, 5)).to_ulong()); // bits 4:0

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(temp_imm_s.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(20,'1') + temp_imm_s.to_string().substr(0,12)))
                            : ((bitset<32>)(temp_imm_s.to_string().substr(0,12)));
            break;

        default: if(debug_mode) cout << "Executing HALT instr\n";
            state.IF.NOP = true;
            state.EX.rd_mem    = false;
            state.EX.wrt_mem   = false;
            state.EX.wrt_enable = false;
            break;
        }

        /*----------------------------------- EXEC / MEM / WB ---------------------------------------*/
        if((state.EX.instr != j_type)&((state.EX.instr != b_type))){
            switch(state.EX.alu_op){
                case ADDI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() +
                                                   state.EX.Imm.to_ulong()); break;
                case XORI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^
                                                   state.EX.Imm.to_ulong()); break;
                case ORI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() |
                                                   state.EX.Imm.to_ulong()); break;
                case ANDI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() &
                                                   state.EX.Imm.to_ulong()); break;
                case ADD:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() +
                                                   state.EX.Read_data2.to_ulong()); break;
                case SUB:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() -
                                                   state.EX.Read_data2.to_ulong()); break;
                case XOR:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^
                                                   state.EX.Read_data2.to_ulong()); break;
                case OR:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() |
                                                   state.EX.Read_data2.to_ulong()); break;
                case AND:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() &
                                                   state.EX.Read_data2.to_ulong()); break;
            }
        } else if(state.EX.instr == j_type){
                    state.MEM.ALUresult = state.EX.Read_data1; // We use this for WB during JAL
        }

        // Set mem config for next stage
        state.MEM.Store_data    = state.EX.Read_data2;
        state.MEM.Rt            = state.EX.Rt;
        state.MEM.Rs            = state.EX.Rs;
        state.MEM.Wrt_reg_addr  = state.EX.Wrt_reg_addr;
        state.MEM.rd_mem        = state.EX.rd_mem;
        state.MEM.wrt_mem       = state.EX.wrt_mem;
        state.MEM.wrt_enable    = state.EX.wrt_enable;

        // If mem read/write == true, read/write it
        if(state.MEM.rd_mem){ // LW type
            state.WB.Wrt_data = ext_dmem.readDataMem(state.MEM.ALUresult);
        } else if(state.MEM.wrt_mem){ // SW Type
            ext_dmem.writeDataMem(state.MEM.ALUresult,state.MEM.Store_data);
            state.WB.Wrt_data=state.MEM.Store_data;

            cout<<"Write data: "<<state.MEM.Store_data<<"\n";
            cout<<"Write addr: "<<state.MEM.ALUresult<<"\n";
            cout<<"Written dat: "<<ext_dmem.readDataMem(state.MEM.ALUresult)<<"\n";

        } else  { // R TYPE
            state.WB.Wrt_data = state.MEM.ALUresult;
        }

        // Set WB Config for next cycle
        state.WB.Rs             = state.MEM.Rs;
        state.WB.Rt             = state.MEM.Rt;
        state.WB.Wrt_reg_addr   = state.MEM.Wrt_reg_addr;
        state.WB.wrt_enable     = state.MEM.wrt_enable;

        //writeback: if writeback == true, write to reg
        if(state.WB.wrt_enable)  myRF.writeRF(state.WB.Wrt_reg_addr, state.WB.Wrt_data);
    }
    myRF.outputRF(cycle); // dump RF per cycle
    if(halted) ext_dmem.outputDataMem();

    printState(state, cycle);
    cycle++;

}
```

b. 5-Stage Pipeline

The schematic for the 5-Stage Pipelined RISCV cycle accurate processor is given below:



As shown above, the main difference between the single stage and 5-Stage is that each of the stages are discrete, they load their cycle values from registers located between them (Shown in blue) and the processor has a module to detect stalls and forwards. Structs were used to hold intermediate values between each of the five stages, Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback.

During the IF stage, the PC is used to index the Instruction memory and load the instruction into the struct for the Decode stage. Here, the PC is also preemptively incremented by four, as it is assumed that branches will not execute. In the ID stage, the instruction is decoded, and arguments needed for execution are loaded, such as Rs1, Rs2, Rd, and all the control signals. If any branches are taken, the ID Stage NOP bit is set and the next branched instruction is loaded immediately. During the EX stage, the ALU operation is completed and the struct for writing to memory is loaded. During the Memory and WB stages, the structs are filled and control signals are used to properly read/write from the data memory, and/or write the data memory output or ALU result to the Register file.

Forwarding and the use of stalls are also determined in the ID stage:

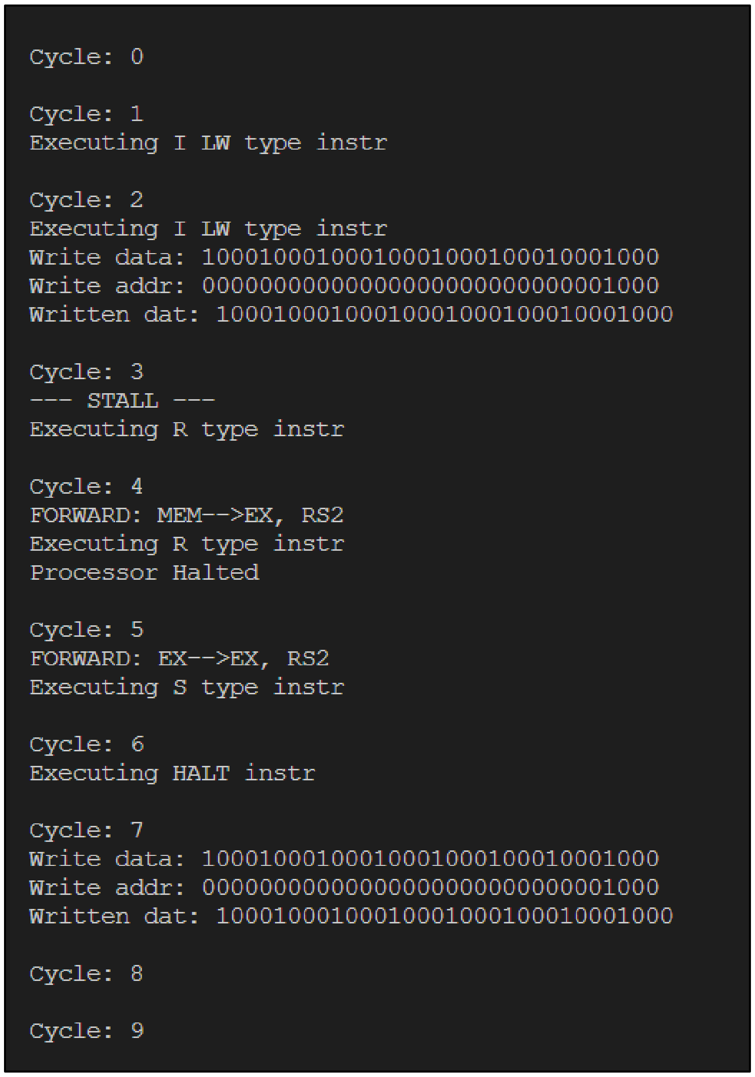
A stall is executed when a Load-Use Hazard is encountered through checking if the previous instruction was a Load instruction, and the previous Rd register matches a current Rs1 or Rs2 register. A stall must be executed since a forward alone will not be enough to mitigate a Load-Use Hazard.

The forwarding unit checks for RAW (Read-after-Write) hazards, and has the ability to forward both from previous EX and MEM stages into the current EX stage. This is used when avoiding RAW hazards and forwarding for the second cycle of a Load-Use Hazard (first cycle must stall).

Below, the code for the 5-Stage implementation of the cycle accurate RISCV process is given. This is a code snippet for the `step()` function for each cycle of the 5-Stage processor. The entirety of the code is given at the end of this report, in the appendix section, and as a separate CPP file along with this report

```
void step() {
    /* Your implementation */
    // We will use the given state structs above.

    if(is_first_cycle){
        // Reset PC on init, set intial states
        state.IF.PC = (bitset<32>) 0;
        state.IF.NOP =  false;
        state.ID.NOP =  true;
        state.EX.NOP =  true;
        state.MEM.NOP = true;
        state.WB.NOP =  true;

        is_first_cycle = false;
    }

    cout<<"\nCycle: "<<cycle<<"\n";

    /* --------------------- WB stage --------------------- */
    if(!state.WB.NOP){
        //writeback: if writeback == true, write to reg
        if(state.WB.wrt_enable)  {

            myRF.writeRF(state.WB.Wrt_reg_addr, state.WB.Wrt_data);
        }
    } state.WB.NOP = state.MEM.NOP;
    /* --------------------- MEM stage -------------------- */
    if(!state.MEM.NOP){
        // If mem read/write == true, read/write it
        if(state.MEM.rd_mem){ // LW type
            state.WB.Wrt_data = ext_dmem.readDataMem(state.MEM.ALUresult);
        } else if(state.MEM.wrt_mem){ // SW Type
            ext_dmem.writeDataMem(state.MEM.ALUresult,state.MEM.Store_data);
            state.WB.Wrt_data=state.MEM.Store_data;

            cout<<"Write data: "<<state.MEM.Store_data<<"\n";
            cout<<"Write addr: "<<state.MEM.ALUresult<<"\n";
            cout<<"Written dat: "<<ext_dmem.readDataMem(state.MEM.ALUresult)<<"\n";

        } else  { // R TYPE
            state.WB.Wrt_data = state.MEM.ALUresult;
        }
        // Set WB Config for next cycle
        state.WB.Rs              = state.MEM.Rs;
        state.WB.Rt              = state.MEM.Rt;
        state.WB.Wrt_reg_addr    = state.MEM.Wrt_reg_addr;
        state.WB.wrt_enable      = state.MEM.wrt_enable;


    } state.MEM.NOP = state.EX.NOP;
    /* --------------------- EX stage --------------------- */
    if(!state.EX.NOP){
        if((state.EX.instr != j_type)&((state.EX.instr != b_type))){
            switch(state.EX.alu_op){
                case ADDI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() +
                                               state.EX.Imm.to_ulong()); break;
                case XORI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^
                                               state.EX.Imm.to_ulong()); break;
                case ORI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() |
                                               state.EX.Imm.to_ulong()); break;
                case ANDI:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() &
                                               state.EX.Imm.to_ulong()); break;
                case ADD:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() +
                                               state.EX.Read_data2.to_ulong()); break;
                case SUB:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() -
                                               state.EX.Read_data2.to_ulong()); break;
                case XOR:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^
                                               state.EX.Read_data2.to_ulong()); break;
                case OR:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() |
                                               state.EX.Read_data2.to_ulong()); break;
                case AND:
                    state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() &
                                               state.EX.Read_data2.to_ulong()); break;
            }
        } else if(state.EX.instr == j_type){
                    state.MEM.ALUresult = state.EX.Read_data1;
        }

        // Set mem config for next stage
        state.MEM.Store_data     = state.EX.Read_data2;
        state.MEM.Rt             = state.EX.Rt;
        state.MEM.Rs             = state.EX.Rs;
        state.MEM.Wrt_reg_addr   = state.EX.Wrt_reg_addr;
        state.MEM.rd_mem         = state.EX.rd_mem;
        state.MEM.wrt_mem        = state.EX.wrt_mem;
        state.MEM.wrt_enable     = state.EX.wrt_enable;

        if(state.EX.instr == halt){
                state.IF.NOP = true;
                state.ID.NOP = true;
                state.EX.NOP = true;
        }

    } state.EX.NOP = state.ID.NOP;
    /* --------------------- ID stage --------------------- */
    if(!state.ID.NOP){
        // Decipher the instruction
        string opcode = state.ID.Instr.to_string().substr(25,7);
        string func7 = state.ID.Instr.to_string().substr(0,7);
        string func3 = state.ID.Instr.to_string().substr(17,3);

        // RS1 set
        state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
        state.EX.Read_data1 = myRF.readRF(state.EX.Rs);
        // Rs2 set
        state.EX.Rt = bitset<5>(state.ID.Instr.to_string().substr(7,5));
        state.EX.Read_data2 = myRF.readRF(state.EX.Rt);

        // Rd set
        state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));

        // Check if there will be a load use hazard:
        // If prev inst was LW and Prev RD is current RS1//RS2
        if((state.EX.instr == i_type_lw) &
        ((state.MEM.Wrt_reg_addr==state.EX.Rs)|(state.MEM.Wrt_reg_addr==state.EX.Rt))){
            stall = true;
            state.ID.NOP = true;
            cout<<"--- STALL ---\n";
        }


        forward = false; // Set forward = false before fwd check
        //-------------------- Forwarding Unit ------------------------------//

        if((!stall)&(cycle>2)){
            // Forward ops: PREV EX --> NEXT EX
            // Rd from MEM input == RS1 or RS2 : forward the Rd data here, overwrite prev val
            if (state.MEM.Wrt_reg_addr == state.EX.Rs){
                cout<<"FORWARD: EX-->EX, RS1\n";    state.EX.Read_data1 = state.MEM.ALUresult;
                forward = true;
            }
            if (state.MEM.Wrt_reg_addr == state.EX.Rt){
                cout<<"FORWARD: EX-->EX, RS2\n";    state.EX.Read_data2 = state.MEM.ALUresult;
                forward = true;
            }

            // Forward ops: PREV MEM --> NEXT EX
            // Rd from WB input == RS1 or RS2 : forward the Rd data here, overwrite prev val
            if ((state.WB.Wrt_reg_addr == state.EX.Rs) & (!forward)){
                cout<<"FORWARD: MEM-->EX, RS1\n";
                state.EX.Read_data1 = state.WB.Wrt_data;

            }
            if ((state.WB.Wrt_reg_addr == state.EX.Rt) & (!forward)){
                cout<<"FORWARD: MEM-->EX, RS2\n";
                state.EX.Read_data2 = state.WB.Wrt_data;

            }

        }

        // Temporary imm ops
        bitset<12> temp_imm_s; bitset<12> temp_imm_b; bitset<20> temp_imm_j;

        if      (opcode == "0110011") {state.EX.instr = r_type;}
        else if (opcode == "0010011") {state.EX.instr = i_type_imm;}
        else if (opcode == "0000011") {state.EX.instr = i_type_lw;}
        else if (opcode == "1101111") {state.EX.instr = j_type;}
        else if (opcode == "1100011") {state.EX.instr = b_type;}
        else if (opcode == "0100011") {state.EX.instr = s_type;}
        else if (opcode == "1111111") {state.EX.instr = halt;}

        switch(state.EX.instr){

            case r_type: cout << "Executing R type instr\n";
                // Control signal set
                state.EX.is_I_type = false;

                // Mem control set
                state.EX.rd_mem = false;
                state.EX.wrt_mem = false;

                //Set alu op
                if (func7 == "0100000") {state.EX.alu_op = SUB;
                } else {
                    if (func3 == "100") { state.EX.alu_op = XOR;
                    } else if (func3 == "110") { state.EX.alu_op = OR;
                    } else if (func3 == "111") { state.EX.alu_op = AND;
                    } else { state.EX.alu_op = ADD;
                    }
                }

                // Write to Rd?
                state.EX.wrt_enable = true;

                break;

            case i_type_imm: cout << "Executing I IMM type instr\n";
                // Control signal set
                state.EX.is_I_type = true;  state.EX.rd_mem = false; state.EX.wrt_mem = false;

                //Set alu op
                if (       func3 == "100") { state.EX.alu_op = XORI;
                } else if (func3 == "110") { state.EX.alu_op = ORI;
                } else if (func3 == "111") { state.EX.alu_op = ANDI;
                } else { state.EX.alu_op = ADDI;
                }

                //Sign extend and load the immediate
                state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                                : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

                // Rd set
                state.EX.wrt_enable = true;
                break;

            case i_type_lw: cout << "Executing I LW type instr\n";
                // Control signal set
                state.EX.is_I_type = true;
                state.EX.rd_mem = true; // We will write to mem
                state.EX.wrt_mem = false; // we will not read from mem
                state.EX.alu_op = ADDI;

                //Sign extend and load the immediate
                state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())?
                            ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                            : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

                // Rd set
                state.EX.wrt_enable = true;
                break;

            case j_type: cout << "Executing J type instr\n";
                // Control signal set
                state.EX.is_I_type = false;
                state.EX.rd_mem    = false; // We will write to mem
                state.EX.wrt_mem   = false; // we will not read from mem

                // Set ALU OP
                state.EX.alu_op = ADDI;

                // Descramble the 20 bit IMM<20|10:1|11|19:12>
                temp_imm_j = (bitset<20>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 20
                                state.ID.Instr.to_string().substr(12, 8) + // Bit 19:12
                                state.ID.Instr.to_string().substr(11, 1) + // bit 11
                                state.ID.Instr.to_string().substr(1, 10)    ).to_ulong()); // bits 10:1

                temp_imm_j <<= 1;

                //Sign extend and load the immediate
                state.EX.Imm =    (((bitset<1>)(temp_imm_j.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(12,'1') + temp_imm_j.to_string().substr(0,20)))
                                : ((bitset<32>)(temp_imm_j.to_string().substr(0,20)));
            state.EX.Read_data1 = state.IF.PC;  //RS1 = Current PC + 4

            state.EX.wrt_enable = true;

            // Discard instruction fetched this cycle, update PC
            state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
            state.ID.Instr = ext_imem.readInstr(state.IF.PC);
            cout<<"Jump and Link Taken\n";
            break;

        case b_type: cout << "Executing B type instr\n";
            // Control signal set
            state.EX.is_I_type = false;
            state.EX.rd_mem    = false; // We will not write to mem
            state.EX.wrt_mem   = false; // we will not read from mem
            state.EX.wrt_enable = false;

            // Descramble the 12 bit IMM<12|10:5>   <4:1|11>
            // The imm exists in instruction bits 31:25, 11:7,  --> 7 bits, 5 bits
            temp_imm_b = (bitset<12>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 12
                                  state.ID.Instr.to_string().substr(24, 1) + // Bit 11
                                  state.ID.Instr.to_string().substr(1, 6) + // Bits 10:5
                                  state.ID.Instr.to_string().substr(20, 4)    ).to_ulong()); // bits 4:1

            temp_imm_b <<= 1;

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(temp_imm_b.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(20,'1') + temp_imm_b.to_string().substr(0,12)))
                            : ((bitset<32>)(temp_imm_b.to_string().substr(0,12)));

            cout<<"RS1 = "<<(int)state.EX.Read_data1.to_ulong()<<", RS2 = "
                                     <<(int)state.EX.Read_data2.to_ulong()<<"\n";

            // We will resolve the branch here:
            if ((func3 == "000") & (state.EX.Read_data1 == state.EX.Read_data2)) { //BEQ
                // Discard instruction fetched this cycle, update PC
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                state.EX.NOP = true;
                cout<<"Branch Taken: BEQ\n";

            } else if ((func3 == "001") & (state.EX.Read_data1 != state.EX.Read_data2)) { //BNE
                // Discard instruction fetched this cycle, update PC & instr from prev cycle
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                state.EX.NOP = true;
                cout<<"Branch Taken: BNE\n";
            } else {
                cout << "No Branch Taken\n";
            }
            break;

        case s_type: cout << "Executing S type instr\n";
            // Control signal set
            state.EX.is_I_type = false;
            state.EX.rd_mem    = false;
            state.EX.wrt_mem   = true;
            state.EX.wrt_enable = false;

            // Set ALU OP
            state.EX.alu_op = ADDI;

            // Descramble the 12 bit IMM<11:5>  <4:0>
            temp_imm_s = (bitset<12> (state.ID.Instr.to_string().substr(0, 7) + // Bits 11:5
                             state.ID.Instr.to_string().substr(20, 5)).to_ulong()); //bits 4:0

            //Sign extend and load the immediate
            state.EX.Imm =   (((bitset<1>)(temp_imm_s.to_string().substr(0,1))).to_ulong())
                            ? ((bitset<32>)(string(20,'1') + temp_imm_s.to_string().substr(0,12)))
                            : ((bitset<32>)(temp_imm_s.to_string().substr(0,12)));
            break;
            default: cout << "Executing HALT instr\n";
                // Halt // Error
                state.EX.rd_mem    = false;
                state.EX.wrt_mem   = false;
                state.EX.wrt_enable = false;
                break;

        }
    } if(!stall) state.ID.NOP = state.IF.NOP;

    /* --------------------- IF stage --------------------- */
    if(!state.IF.NOP){
        if(stall){
            stall = false;
            // Execute the prev stalled instruction again
            state.ID.NOP = state.IF.NOP;
        } else {
            state.ID.Instr = ext_imem.readInstr(state.IF.PC);
            state.IF.PC = (bitset<32>(state.IF.PC.to_ulong() + 4));
            instr_count++;
        }
    }
    /* --------------------- 5 Stage End ------------------- */

    if (state.IF.NOP && state.ID.NOP && state.EX.NOP && state.MEM.NOP && state.WB.NOP){
        halted = true;
        ext_dmem.outputDataMem();
    }
    myRF.outputRF(cycle); // dump RF
    printState(state, cycle); //print states after executing cycle 0, cycle 1, cycle 2 ...

    cycle++;
}
```

Below, a sample output of the terminal is given during execution. It showcases both the stalling and forwarding capabilities of the processor, as a Load-Use hazard is detected, and a Stall and a Forward is used to mitigate it.



```
Cycle: 0

Cycle: 1
Executing I LW type instr

Cycle: 2
Executing I LW type instr
Write data: 10001000100010001000100010001000
Write addr: 00000000000000000000000000001000
Written dat: 10001000100010001000100010001000

Cycle: 3
--- STALL ---
Executing R type instr

Cycle: 4
FORWARD: MEM-->EX, RS2
Executing R type instr
Processor Halted

Cycle: 5
FORWARD: EX-->EX, RS2
Executing S type instr

Cycle: 6
Executing HALT instr

Cycle: 7
Write data: 10001000100010001000100010001000
Write addr: 00000000000000000000000000001000
Written dat: 10001000100010001000100010001000

Cycle: 8

Cycle: 9
```

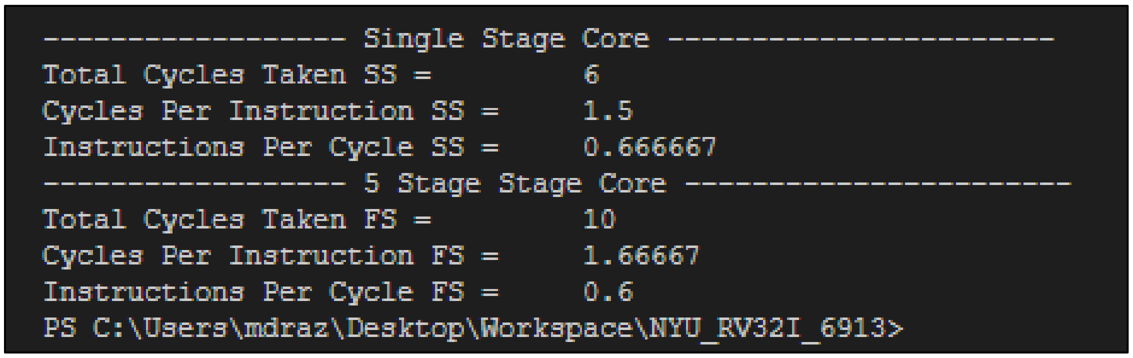
2. Measured Average CPI, Total Execution Cycles, Instructions Per Cycle

a. Performance Monitor Code

Both processors were given counters to count the number of instructions which were decoded and executed, and the number of cycles consumed at the end of the program. These two variables, the instruction count and cycle count, were then used to measure the performance metrics of the processors, which were the Cycles per instruction (CPI) and instructions per cycle (IPC), as shown below:

```
    // Here, we will output the Performance Metrics after both
    // Processors are run to completion

    float CPI_SS = ((float)(SSCore.cycle)) / ((float)(SSCore.instr_count));
    float IPC_SS = ((float)(SSCore.instr_count)) / ((float)(SSCore.cycle));


    cout<<"\n------------------ Single Stage Core -----------------------\n";
    cout<<"Total Cycles Taken SS = \t"<<SSCore.cycle<<"\n";
    cout<<"Cycles Per Instruction SS = \t"<<CPI_SS<<"\n";
    cout<<"Instructions Per Cycle SS = \t"<<IPC_SS<<"\n";

    float CPI_FS = ((float)(FSCore.cycle)) / ((float)(FSCore.instr_count));
    float IPC_FS = ((float)(FSCore.instr_count)) / ((float)(FSCore.cycle));

    cout<<"------------------ 5 Stage Stage Core -----------------------\n";
    cout<<"Total Cycles Taken FS = \t"<<FSCore.cycle<<"\n";
    cout<<"Cycles Per Instruction FS = \t"<<CPI_FS<<"\n";
    cout<<"Instructions Per Cycle FS = \t"<<IPC_FS<<"\n";

    ofstream metricsFile;
        metricsFile.open (ioDir + "\\PerfMetrics.txt");
        metricsFile<<"\n------------------ Single Stage Core -----------------------\n";
        metricsFile<<"Total Cycles Taken SS = \t"<<SSCore.cycle<<"\n";
        metricsFile<<"Cycles Per Instruction SS = \t"<<CPI_SS<<"\n";
        metricsFile<<"Instructions Per Cycle SS = \t"<<IPC_SS<<"\n";
        metricsFile<<"------------------ 5 Stage Stage Core -----------------------\n";
        metricsFile<<"Total Cycles Taken FS = \t"<<FSCore.cycle<<"\n";
        metricsFile<<"Cycles Per Instruction FS = \t"<<CPI_FS<<"\n";
        metricsFile<<"Instructions Per Cycle FS = \t"<<IPC_FS<<"\n";
        metricsFile.close();
```

A sample terminal output is given below for the above code:



```
------------------- Single Stage Core -----------------------
Total Cycles Taken SS =         6
Cycles Per Instruction SS =     1.5
Instructions Per Cycle SS =     0.666667
------------------- 5 Stage Stage Core -----------------------
Total Cycles Taken FS =         10
Cycles Per Instruction FS =     1.66667
Instructions Per Cycle FS =     0.6
PS C:\Users\mdraz\Desktop\Workspace\NYU_RV32I_6913>
```

b. Single Stage Metrics

Single stage metrics for the 5 released test cases are given below, along with an average cycle count, CPI, and IPC:

	Test Case 0	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Average
Total Cycles	6	40	7	8	28	17.8
Cycles Per Instruction	1.5	1.05	1.4	1.33	1.07	1.27
Instructions Per Cycle	0.67	0.95	0.71	0.75	0.92	0.8

c. 5-Stage Pipeline Metrics

5-Stage metrics for the 5 released test cases are given below, along with an average cycle count, CPI, and IPC:

	Test Case 0	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Average
Total Cycles	10	46	10	11	31	21.6
Cycles Per Instruction	1.67	1.15	1.43	1.37	1.11	1.346
Instructions Per Cycle	0.6	0.87	0.7	0.72	0.9	0.758

3. Single Stage and Pipeline Comparison

	Single Stage	Five Stage Pipeline
Average Total Cycles	17.8	21.6
Average Cycles Per Instruction	1.27	1.346
Average Instructions Per Cycle	0.8	0.758

From the metrics shown above, it may seem that the 5-Stage pipeline lags behind the Single stage processor in terms of performance—however this is not the case, as a single cycle is not technically equivalent between the two processors: the single-stage processor executes all stages of execution during a cycle, while the 5-stage processor executes a single stage of the execution pipeline during a cycle. This means that of the total 21.6 cycles consumed on average by the 5-stage processor, this would essentially be equivalent to 21.6 / 5 → 4.32 single stage cycles, making the 5-stage processor about 5 times higher performance than the single stage processor. Although it is higher performance, the 5-stage processor would be more costly to implement as the added registers between stages and added stall/forward modules would take up more space on an IC die.

4. Optimizations and Features to Improve Performance

There would be several features that could be implemented in order to improve performance:

One feature that would improve performance would be implementing a cache for the data memory. Whenever the current processor encounters a load-use hazard, the processor must stall for a single cycle as just forward alone cannot mitigate this hazard. The stall causes a loss in performance as the processor now must wait for the correct value from the data memory to be written to the register file. If a cache were to be implemented, for programs where the same data memory is accessed and loaded, such several of the test cases, the cache may be able to totally eliminate all stalls. For programs where a variety of addresses are accessed in the data memory, the cache may be able to provide data for a few of the addresses, lowering the overall number of stalls if not removing them completely. Both of these scenarios would yield and increase in the performance of as less cycles are wasted on processor stalls.

Another feature that would improve the performance of the processor would be implementing better branch prediction. The current processor does not have any branch prediction algorithm, and defaults to assuming that no branches will be taken. Whenever a branch is taken, the pipeline has to be flushed and stalled, consuming a cycle and wasting an instruction. Implementing better branch prediction would reduce the number of times the pipeline would be flushed or stalled, therefore increasing the performance through reduced cycle consumption.

5. Appendix

For completeness, and in case of the (`mr4425_sk8944-NYU_RV32I_6913.cpp`) file not being available, the entirety of the code for the RISCV cycle-accurate simulator is reproduced below.

```
//--------------------------------------------------------------------------------
//--      ECE-GY 6913 Computing Systems Architecture - RISC-V processor simulator
//--      Fall 2021
//--
//--      Md Raz           --  Siddharth Kandpal
//--      N17762874        --  N10799721
//--      mr4425@nyu.edu   --  sk8944@nyu.edu
//---------------------------------------------------------------------------------

#include <iostream>
#include <string>
#include <vector>
#include <bitset>
#include <fstream>

using namespace std;
#define MemSize 1000

struct IFStruct {
    bitset<32>  PC;
    bool        NOP;
};

struct IDStruct {
    bitset<32>  Instr;
    bool        NOP;
};

enum op_type {
    ADDI,
    XORI,
    ORI,
    ANDI,
    ADD,
    SUB,
    XOR,
    OR,
    AND
};

enum instr_type {
    r_type,
    i_type_imm,
    i_type_lw,
    j_type,
    b_type,
    s_type,
    halt
};

struct EXStruct {
    bitset<32>  Read_data1;
    bitset<32>  Read_data2;
    bitset<32>  Imm;
    bitset<5>   Rs;
    bitset<5>   Rt;
    bitset<5>   Wrt_reg_addr;
    op_type     alu_op;
    instr_type  instr;
    bool        is_I_type;
    bool        rd_mem;
    bool        wrt_mem;
    bool        wrt_enable;
    bool        NOP;
};

struct MEMStruct {
    bitset<32>  ALUresult;
    bitset<32>  Store_data;
    bitset<5>   Rs;
    bitset<5>   Rt;
    bitset<5>   Wrt_reg_addr;
    bool        rd_mem;
    bool        wrt_mem;
    bool        wrt_enable;
    bool        NOP;
};

struct WBStruct {
    bitset<32>  Wrt_data;
    bitset<5>   Rs;
    bitset<5>   Rt;
    bitset<5>   Wrt_reg_addr;
    bool        wrt_enable;
    bool        NOP;
};
```

```
struct stateStruct {
    IFStruct    IF;
    IDStruct    ID;
    EXStruct    EX;
    MEMStruct   MEM;
    WBStruct    WB;
};


class InsMem
{
    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
            // we will concatenate 4 lines of the inst mem to return 32 bits

            string instr_str;
            for (int i = 0; i < 4; i++) instr_str.append(IMem[ReadAddress.to_ulong() + i].to_string());
            return (bitset<32>(instr_str));
        }

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

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

            // we will concatenate 4 lines of the dat mem to return 32 bits
            string dat_str;
            for (int i = 0; i < 4; i++) dat_str.append(DMem[Address.to_ulong() + i].to_string());
            return ((bitset<32>)dat_str);
        }

        void writeDataMem(bitset<32> Address, bitset<32> WriteData) {
            //We will break down the incoming 32 bits and write 8 bits at a time
             for (int i = 0; i < 4; i++)     {
                DMem[Address.to_ulong() + i] = bitset<8>(WriteData.to_string().substr((8 * i), 8));
             }

        }

        void outputDataMem() {
            ofstream dmemout;
            dmemout.open(opFilePath, std::ios_base::trunc);
            if (dmemout.is_open()) {
                for (int j = 0; j< 1000; j++)
                {
                    dmemout << DMem[j]<<endl;
                }

            }
            else cout<<"Unable to open "<<id<<" DMEM result file." << endl;
            dmemout.close();
        }

    private:
        vector<bitset<8> > DMem;
};

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) {
            // We will return a 32-bit bitset, reading from the Reg_addr
            return Registers[Reg_addr.to_ulong()];
        }

        void writeRF(bitset<5> Reg_addr, bitset<32> Wrt_reg_data) {
            // We will write the 32 bit reg data to the Reg_Addr
            Registers[Reg_addr.to_ulong()] = Wrt_reg_data.to_ullong();
            Registers[0] = bitset<32> (0); // Ensure reg0 stays 0 val
        }

        void outputRF(int cycle) {
            ofstream rfout;
            if (cycle == 0)
                rfout.open(outputFile, std::ios_base::trunc);
            else
                rfout.open(outputFile, std::ios_base::app);
            if (rfout.is_open())
            {
                rfout<<"State of RF after executing cycle:\t"<<cycle<<endl;
                for (int j = 0; j<32; j++)
                {
                    rfout<<Registers[j]<<endl;
                }
            }
            else cout<<"Unable to open RF output file."<<endl;
            rfout.close();
        }

    private:
        vector<bitset<32> >Registers;
};

class Core {
    public:
        RegisterFile myRF;
        uint32_t cycle = 0;
        uint32_t instr_count = 0;
        bool halted = false;
        bool is_first_cycle = true;
        bool debug_mode = false;
        bool stall = false;
        bool forward = false;
        string ioDir;
        struct stateStruct state, nextState;
        InsMem ext_imem;
        DataMem ext_dmem;

        Core(string ioDir, InsMem &imem, DataMem &dmem): myRF(ioDir), ioDir{ioDir}, ext_imem {imem}, ext_dmem {dmem} {}

        virtual void step() {}

        virtual void printState() {}
};

class SingleStageCore : public Core {
    public:
        SingleStageCore(string ioDir, InsMem &imem, DataMem &dmem): Core(ioDir + "\\SS_", imem, dmem), opFilePath(ioDir +
"\\StateResult_SS.txt") {}

        void step() {
            /* Your implementation*/
            // We will use the given state structs above.

            if(is_first_cycle){
                state.IF.PC = (bitset<32>)0;
                state.IF.NOP =  false;
                is_first_cycle = false;
            }

            if (state.IF.NOP) {
                halted = true;
                cout << "Processor Halted\n";
            }
            if(!halted){
                // Retrieve instr
                if(debug_mode & (!halted)) cout << "\nPC =  "<<state.IF.PC.to_ulong()<<"  \t-->\t";
                state.ID.Instr = ext_imem.readInstr(state.IF.PC);

                // Decipher the instruction
                string opcode = state.ID.Instr.to_string().substr(25,7);
                string func7 = state.ID.Instr.to_string().substr(0,7);
                string func3 = state.ID.Instr.to_string().substr(17,3);

                // RS1 set
                state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
                state.EX.Read_data1 = myRF.readRF(state.EX.Rs);

                // Rs2 set
                state.EX.Rt = bitset<5>(state.ID.Instr.to_string().substr(7,5));
                state.EX.Read_data2 = myRF.readRF(state.EX.Rt);

                // Rd set
                state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));

                // Temporary imm ops
                bitset<12> temp_imm_s; bitset<12> temp_imm_b; bitset<20> temp_imm_j;

                // Opcode --> Instr type
                if      (opcode == "0110011") {state.EX.instr = r_type;}
                else if (opcode == "0010011") {state.EX.instr = i_type_imm;}
                else if (opcode == "0000011") {state.EX.instr = i_type_lw;}
                else if (opcode == "1101111") {state.EX.instr = j_type;}
                else if (opcode == "1100011") {state.EX.instr = b_type;}
                else if (opcode == "0100011") {state.EX.instr = s_type;}
                else if (opcode == "1111111") {state.EX.instr = halt;}

                if(state.EX.instr != halt){
                    state.IF.PC = (bitset<32>(state.IF.PC.to_ulong() + 4)); // Premptivly Pc = PC + 4
                    instr_count++;
                }

                switch(state.EX.instr){

                    case r_type: if(debug_mode) cout << "Executing R type instr\n";
                        // Control signal set
                        state.EX.is_I_type = false;

                        // Mem control set
                        state.EX.rd_mem = false;
                        state.EX.wrt_mem = false;

                        //Set alu op
                        if (func7 == "0100000") {state.EX.alu_op = SUB;
                        } else {
                            if (func3 == "100") { state.EX.alu_op = XOR;
                            } else if (func3 == "110") { state.EX.alu_op = OR;
                            } else if (func3 == "111") { state.EX.alu_op = AND;
                            } else { state.EX.alu_op = ADD;
                            }
                        }

                        // Write to Rd?
                        state.EX.wrt_enable = true;

                        break;

                    case i_type_imm: if(debug_mode) cout << "Executing I IMM type instr\n";
                        // Control signal set
                        state.EX.is_I_type = true;  state.EX.rd_mem = false; state.EX.wrt_mem = false;

                        //Set alu op
                        if (        func3 == "100") { state.EX.alu_op = XORI;
                        } else if (func3 == "110") { state.EX.alu_op = ORI;
                        } else if (func3 == "111") { state.EX.alu_op = ANDI;
                        } else { state.EX.alu_op = ADDI;
                        }

                        //Sign extend and load the immediate
                        state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())
                                        ? ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                                        : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

                        // Rd set
                        state.EX.wrt_enable = true;
                        break;

                    case i_type_lw: if(debug_mode) cout << "Executing I LW type instr\n";
                        // Control signal set
                        state.EX.is_I_type = true;
                        state.EX.rd_mem = true; // We will write to mem
                        state.EX.wrt_mem = false; // we will not read from mem
                        state.EX.alu_op = ADDI;

                        //Sign extend and load the immediate
                        state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())?
                                           ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                                        : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));
                // Rd set
                state.EX.wrt_enable = true;
                break;

            case j_type: if(debug_mode) cout << "Executing J type instr\n";
                // Control signal set
                state.EX.is_I_type = false;
                state.EX.rd_mem    = false; // We will write to mem
                state.EX.wrt_mem   = false; // we will not read from mem

                // Set ALU OP
                state.EX.alu_op = ADDI;

                // Descramble the 20 bit IMM<20|10:1|11|19:12>
                temp_imm_j = (bitset<20>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 20
                                            state.ID.Instr.to_string().substr(12, 8) + // Bit 19:12
                                            state.ID.Instr.to_string().substr(11, 1) + // bit 11
                                            state.ID.Instr.to_string().substr(1, 10)    ).to_ulong()); // bits 10:1

                temp_imm_j <<= 1;

                //Sign extend and load the immediate
                state.EX.Imm =    (((bitset<1>)(temp_imm_j.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(12,'1') + temp_imm_j.to_string().substr(0,20)))
                                : ((bitset<32>)(temp_imm_j.to_string().substr(0,20)));

                state.EX.Read_data1 = state.IF.PC;  //RS1 = Current PC + 4

                state.EX.wrt_enable = true;

                // Discard instruction fetched this cycle, update PC
                state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                cout<<"Jump and Link Taken\n";
                break;

            case b_type: if(debug_mode) cout << "Executing B type instr\n";
                // Control signal set
                state.EX.is_I_type = false;
                state.EX.rd_mem    = false; // We will write to mem
                state.EX.wrt_mem   = false; // we will not read from mem
                state.EX.wrt_enable = false;

                // Descramble the 12 bit IMM<12|10:5>   <4:1|11>
                // The imm exists in instruction bits 31:25, 11:7,  --> 7 bits, 5 bits
                temp_imm_b = (bitset<12>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 12
                                            state.ID.Instr.to_string().substr(24, 1) + // Bit 11
                                            state.ID.Instr.to_string().substr(1, 6) + // Bits 10:5
                                            state.ID.Instr.to_string().substr(20, 4)    ).to_ulong()); // bits 4:1

                temp_imm_b <<= 1;

                //Sign extend and load the immediate
                state.EX.Imm =   (((bitset<1>)(temp_imm_b.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(20,'1') + temp_imm_b.to_string().substr(0,12)))
                                : ((bitset<32>)(temp_imm_b.to_string().substr(0,12)));

                cout<<"RS1 = "<<(int)state.EX.Read_data1.to_ulong()<<", RS2 = "<<(int)state.EX.Read_data2.to_ulong()<<"\n";

                // We will resolve the branch here:
                if ((func3 == "000") & (state.EX.Read_data1 == state.EX.Read_data2)) { //BEQ
                    // Discard instruction fetched this cycle, update PC
                    state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                    cout<<"Branch Taken: BEQ\n";

                } else if ((func3 == "001") & (state.EX.Read_data1 != state.EX.Read_data2)) { //BNE
                    // Discard instruction fetched this cycle, update PC & instr from prev cycle
                    state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                    cout<<"Branch Taken: BNE\n";

                } else {
                    cout << "No Branch Taken\n";
                }

                break;

            case s_type: if(debug_mode) cout << "Executing S type instr\n";
                // Control signal set
                state.EX.is_I_type = false;
                state.EX.rd_mem    = false;
                state.EX.wrt_mem   = true;
                state.EX.wrt_enable = false;

                // Set ALU OP
                state.EX.alu_op = ADDI;

                // Descramble the 12 bit IMM<11:5>  <4:0>
                temp_imm_s = (bitset<12> (state.ID.Instr.to_string().substr(0, 7) + // Bits 11:5
                                        state.ID.Instr.to_string().substr(20, 5)).to_ulong()); //bits 4:0

                //Sign extend and load the immediate
                state.EX.Imm =   (((bitset<1>)(temp_imm_s.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(20,'1') + temp_imm_s.to_string().substr(0,12)))
                                : ((bitset<32>)(temp_imm_s.to_string().substr(0,12)));
                break;
                default: if(debug_mode) cout << "Executing HALT instr\n";
                    state.IF.NOP = true;

                    state.EX.rd_mem    = false;
                    state.EX.wrt_mem   = false;
                    state.EX.wrt_enable = false;
                    break;

            }


            /*---------------------------------- EXEC / MEM / WB ---------------------------------------*/
            if((state.EX.instr != j_type)&((state.EX.instr != b_type))){
                switch(state.EX.alu_op){
                    case ADDI:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() + state.EX.Imm.to_ulong()); break;
                    case XORI:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^ state.EX.Imm.to_ulong()); break;
                    case ORI:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() | state.EX.Imm.to_ulong()); break;
                    case ANDI:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() & state.EX.Imm.to_ulong()); break;
                    case ADD:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() + state.EX.Read_data2.to_ulong()); break;
                    case SUB:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() - state.EX.Read_data2.to_ulong()); break;
                    case XOR:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^ state.EX.Read_data2.to_ulong()); break;
                    case OR:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() | state.EX.Read_data2.to_ulong()); break;
                    case AND:
                        state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() & state.EX.Read_data2.to_ulong()); break;
                }
            } else if(state.EX.instr == j_type){
                        state.MEM.ALUresult = state.EX.Read_data1; // We use this for WB during JAL
            }

            // Set mem config for next stage
            state.MEM.Store_data    = state.EX.Read_data2;
            state.MEM.Rt            = state.EX.Rt;
            state.MEM.Rs            = state.EX.Rs;
            state.MEM.Wrt_reg_addr  = state.EX.Wrt_reg_addr;
            state.MEM.rd_mem        = state.EX.rd_mem;
            state.MEM.wrt_mem       = state.EX.wrt_mem;
            state.MEM.wrt_enable    = state.EX.wrt_enable;

            // If mem read/write == true, read/write it
            if(state.MEM.rd_mem){ // LW type
                state.WB.Wrt_data = ext_dmem.readDataMem(state.MEM.ALUresult);
            } else if(state.MEM.wrt_mem){ // SW Type
                ext_dmem.writeDataMem(state.MEM.ALUresult,state.MEM.Store_data);
                state.WB.Wrt_data=state.MEM.Store_data;

                cout<<"Write data: "<<state.MEM.Store_data<<"\n";
                cout<<"Write addr: "<<state.MEM.ALUresult<<"\n";
                cout<<"Written dat: "<<ext_dmem.readDataMem(state.MEM.ALUresult)<<"\n";

            } else  { // R TYPE
                state.WB.Wrt_data = state.MEM.ALUresult;
            }

            // Set WB Config for next cycle
            state.WB.Rs             = state.MEM.Rs;
            state.WB.Rt             = state.MEM.Rt;
            state.WB.Wrt_reg_addr   = state.MEM.Wrt_reg_addr;
            state.WB.wrt_enable     = state.MEM.wrt_enable;

            //writeback: if writeback == true, write to reg
            if(state.WB.wrt_enable)  myRF.writeRF(state.WB.Wrt_reg_addr, state.WB.Wrt_data);
        }

        myRF.outputRF(cycle); // dump RF per cycle
        if(halted) ext_dmem.outputDataMem();

        printState(state, cycle);
        cycle++;

    }

    void printState(stateStruct state, int cycle) {
        ofstream printstate;
        if (cycle == 0)
            printstate.open(opFilePath, std::ios_base::trunc);
        else
            printstate.open(opFilePath, std::ios_base::app);
        if (printstate.is_open()) {
            printstate<<"State after executing cycle:\t"<<cycle<<endl;

            printstate<<"IF.PC:\t"<<state.IF.PC.to_ulong()<<endl;
            printstate<<"IF.NOP:\t"<<state.IF.NOP<<endl;
        }
        else cout<<"Unable to open SS StateResult output file." << endl;
        printstate.close();
    }
private:
        string opFilePath;
};

class FiveStageCore : public Core{
    public:

        FiveStageCore(string ioDir, InsMem &imem, DataMem &dmem): Core(ioDir + "\\FS_", imem, dmem), opFilePath(ioDir + "\\StateResult_FS.txt")
{}

        void step() {
            /* Your implementation */
            // We will use the given state structs above.

            if(is_first_cycle){
                // Reset PC on init, set intial states
                state.IF.PC = (bitset<32>) 0;
                state.IF.NOP =  false;
                state.ID.NOP =  true;
                state.EX.NOP =  true;
                state.MEM.NOP = true;
                state.WB.NOP =  true;

                is_first_cycle = false;
            }

            cout<<"\nCycle: "<<cycle<<"\n";

            /* --------------------- WB stage --------------------- */
            if(!state.WB.NOP){
                //writeback: if writeback == true, write to reg
                if(state.WB.wrt_enable)  {

                    myRF.writeRF(state.WB.Wrt_reg_addr, state.WB.Wrt_data);
                }
            } state.WB.NOP = state.MEM.NOP;
            /* --------------------- MEM stage -------------------- */
            if(!state.MEM.NOP){
                // If mem read/write == true, read/write it
                if(state.MEM.rd_mem){ // LW type
                    state.WB.Wrt_data = ext_dmem.readDataMem(state.MEM.ALUresult);
                } else if(state.MEM.wrt_mem){ // SW Type
                    ext_dmem.writeDataMem(state.MEM.ALUresult,state.MEM.Store_data);
                    state.WB.Wrt_data=state.MEM.Store_data;

                    cout<<"Write data: "<<state.MEM.Store_data<<"\n";
                    cout<<"Write addr: "<<state.MEM.ALUresult<<"\n";
                    cout<<"Written dat: "<<ext_dmem.readDataMem(state.MEM.ALUresult)<<"\n";

                } else  { // R TYPE
                    state.WB.Wrt_data = state.MEM.ALUresult;
                }

                // Set WB Config for next cycle
                state.WB.Rs             = state.MEM.Rs;
                state.WB.Rt             = state.MEM.Rt;
                state.WB.Wrt_reg_addr   = state.MEM.Wrt_reg_addr;
                state.WB.wrt_enable     = state.MEM.wrt_enable;


            } state.MEM.NOP = state.EX.NOP;
            /* --------------------- EX stage --------------------- */
            if(!state.EX.NOP){
                if((state.EX.instr != j_type)&((state.EX.instr != b_type))){
                    switch(state.EX.alu_op){
                        case ADDI:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() + state.EX.Imm.to_ulong()); break;
                        case XORI:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^ state.EX.Imm.to_ulong()); break;
                        case ORI:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() | state.EX.Imm.to_ulong()); break;
                        case ANDI:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() & state.EX.Imm.to_ulong()); break;
                        case ADD:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() + state.EX.Read_data2.to_ulong()); break;
                        case SUB:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() - state.EX.Read_data2.to_ulong()); break;
                        case XOR:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() ^ state.EX.Read_data2.to_ulong()); break;
                        case OR:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() | state.EX.Read_data2.to_ulong()); break;
                        case AND:
                            state.MEM.ALUresult = bitset<32>(state.EX.Read_data1.to_ulong() & state.EX.Read_data2.to_ulong()); break;
                    }
                } else if(state.EX.instr == j_type){
                            state.MEM.ALUresult = state.EX.Read_data1;
                }

                // Set mem config for next stage
                state.MEM.Store_data    = state.EX.Read_data2;
                state.MEM.Rt            = state.EX.Rt;
                state.MEM.Rs            = state.EX.Rs;
                state.MEM.Wrt_reg_addr  = state.EX.Wrt_reg_addr;
                state.MEM.rd_mem        = state.EX.rd_mem;
                state.MEM.wrt_mem       = state.EX.wrt_mem;
                state.MEM.wrt_enable    = state.EX.wrt_enable;
            if(state.EX.instr == halt){
                    state.IF.NOP = true;
                    state.ID.NOP = true;
                    state.EX.NOP = true;
            }

        } state.EX.NOP = state.ID.NOP;
        /* --------------------- ID stage --------------------- */
        if(!state.ID.NOP){
            // Decipher the instruction
            string opcode = state.ID.Instr.to_string().substr(25,7);
            string func7 = state.ID.Instr.to_string().substr(0,7);
            string func3 = state.ID.Instr.to_string().substr(17,3);

            // RS1 set
            state.EX.Rs = bitset<5>(state.ID.Instr.to_string().substr(12,5));
            state.EX.Read_data1 = myRF.readRF(state.EX.Rs);

            // Rs2 set
            state.EX.Rt = bitset<5>(state.ID.Instr.to_string().substr(7,5));
            state.EX.Read_data2 = myRF.readRF(state.EX.Rt);

            // Rd set
            state.EX.Wrt_reg_addr = bitset<5>(state.ID.Instr.to_string().substr(20,5));

            // Check if there will be a load use hazard:
            // If prev inst was LW and Prev RD is current RS1//RS2
            if((state.EX.instr == i_type_lw) & ((state.MEM.Wrt_reg_addr==state.EX.Rs)|(state.MEM.Wrt_reg_addr==state.EX.Rt))){
                stall = true;
                state.ID.NOP = true;
                cout<<"--- STALL ---\n";
            }


            forward = false; // Set forward = false before fwd check

            if((!stall)&(cycle>2)){
                // Forward ops: PREV EX --> NEXT EX
                // Rd from MEM input == RS1 or RS2 : forward the Rd data here, overwrite prev val
                if (state.MEM.Wrt_reg_addr == state.EX.Rs){
                    cout<<"FORWARD: EX-->EX, RS1\n";    state.EX.Read_data1 = state.MEM.ALUresult;
                    forward = true;
                }
                if (state.MEM.Wrt_reg_addr == state.EX.Rt){
                    cout<<"FORWARD: EX-->EX, RS2\n";    state.EX.Read_data2 = state.MEM.ALUresult;
                    forward = true;
                }

                // Forward ops: PREV MEM --> NEXT EX
                // Rd from WB input == RS1 or RS2 : forward the Rd data here, overwrite prev val
                if ((state.WB.Wrt_reg_addr == state.EX.Rs) & (!forward)){
                    cout<<"FORWARD: MEM-->EX, RS1\n";
                    state.EX.Read_data1 = state.WB.Wrt_data;

                }
                if ((state.WB.Wrt_reg_addr == state.EX.Rt) & (!forward)){
                    cout<<"FORWARD: MEM-->EX, RS2\n";
                    state.EX.Read_data2 = state.WB.Wrt_data;

                }

            }

            // Temporary imm ops
            bitset<12> temp_imm_s; bitset<12> temp_imm_b; bitset<20> temp_imm_j;

            if      (opcode == "0110011") {state.EX.instr = r_type;}
            else if (opcode == "0010011") {state.EX.instr = i_type_imm;}
            else if (opcode == "0000011") {state.EX.instr = i_type_lw;}
            else if (opcode == "1101111") {state.EX.instr = j_type;}
            else if (opcode == "1100011") {state.EX.instr = b_type;}
            else if (opcode == "0100011") {state.EX.instr = s_type;}
            else if (opcode == "1111111") {state.EX.instr = halt;}

            switch(state.EX.instr){

                case r_type: cout << "Executing R type instr\n";
                    // Control signal set
                    state.EX.is_I_type = false;

                    // Mem control set
                    state.EX.rd_mem = false;
                    state.EX.wrt_mem = false;

                    //Set alu op
                    if (func7 == "0100000") {state.EX.alu_op = SUB;
                    } else {
                        if (func3 == "100") { state.EX.alu_op = XOR;
                        } else if (func3 == "110") { state.EX.alu_op = OR;
                        } else if (func3 == "111") { state.EX.alu_op = AND;
                        } else { state.EX.alu_op = ADD;
                        }
                    }
                    // Write to Rd?
                    state.EX.wrt_enable = true;

                    break;

                case i_type_imm: cout << "Executing I IMM type instr\n";
                    // Control signal set
                    state.EX.is_I_type = true;  state.EX.rd_mem = false; state.EX.wrt_mem = false;

                    //Set alu op
                    if (        func3 == "100") { state.EX.alu_op = XORI;
                    } else if (func3 == "110") { state.EX.alu_op = ORI;
                    } else if (func3 == "111") { state.EX.alu_op = ANDI;
                    } else { state.EX.alu_op = ADDI;
                    }

                    //Sign extend and load the immediate
                    state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())
                                    ? ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                                    : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

                    // Rd set
                    state.EX.wrt_enable = true;
                    break;

                case i_type_lw: cout << "Executing I LW type instr\n";
                    // Control signal set
                    state.EX.is_I_type = true;
                    state.EX.rd_mem = true; // We will write to mem
                    state.EX.wrt_mem = false; // we will not read from mem
                    state.EX.alu_op = ADDI;

                    //Sign extend and load the immediate
                    state.EX.Imm =   (((bitset<1>)(state.ID.Instr.to_string().substr(0,1))).to_ulong())?
                                       ((bitset<32>)(string(20,'1') + state.ID.Instr.to_string().substr(0,12)))
                                    : ((bitset<32>)(state.ID.Instr.to_string().substr(0,12)));

                    // Rd set
                    state.EX.wrt_enable = true;
                    break;

                case j_type: cout << "Executing J type instr\n";
                    // Control signal set
                    state.EX.is_I_type = false;
                    state.EX.rd_mem    = false; // We will write to mem
                    state.EX.wrt_mem   = false; // we will not read from mem

                    // Set ALU OP
                    state.EX.alu_op = ADDI;

                    // Descramble the 20 bit IMM<20|10:1|11|19:12>
                    temp_imm_j = (bitset<20>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 20
                                                state.ID.Instr.to_string().substr(12, 8) + // Bit 19:12
                                                state.ID.Instr.to_string().substr(11, 1) + // bit 11
                                                state.ID.Instr.to_string().substr(1, 10)    ).to_ulong()); // bits 10:1

                    temp_imm_j <<= 1;

                    //Sign extend and load the immediate
                    state.EX.Imm =    (((bitset<1>)(temp_imm_j.to_string().substr(0,1))).to_ulong())
                                    ? ((bitset<32>)(string(12,'1') + temp_imm_j.to_string().substr(0,20)))
                                    : ((bitset<32>)(temp_imm_j.to_string().substr(0,20)));

                    state.EX.Read_data1 = state.IF.PC;  //RS1 = Current PC + 4

                    state.EX.wrt_enable = true;

                    // Discard instruction fetched this cycle, update PC
                    state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                    state.ID.Instr = ext_imem.readInstr(state.IF.PC);
                    cout<<"Jump and Link Taken\n";
                    break;

                case b_type: cout << "Executing B type instr\n";
                    // Control signal set
                    state.EX.is_I_type = false;
                    state.EX.rd_mem    = false; // We will write to mem
                    state.EX.wrt_mem   = false; // we will not read from mem
                    state.EX.wrt_enable = false;

                    // Descramble the 12 bit IMM<12|10:5>   <4:1|11>
                    // The imm exists in instruction bits 31:25, 11:7,  --> 7 bits, 5 bits
                    temp_imm_b = (bitset<12>  ( state.ID.Instr.to_string().substr(0, 1) + // Bit 12
                                                state.ID.Instr.to_string().substr(24, 1) + // Bit 11
                                                state.ID.Instr.to_string().substr(1, 6) + // Bits 10:5
                                                state.ID.Instr.to_string().substr(20, 4)    ).to_ulong()); // bits 4:1

                    temp_imm_b <<= 1;

                    //Sign extend and load the immediate
                    state.EX.Imm =   (((bitset<1>)(temp_imm_b.to_string().substr(0,1))).to_ulong())
                                    ? ((bitset<32>)(string(20,'1') + temp_imm_b.to_string().substr(0,12)))
                                    : ((bitset<32>)(temp_imm_b.to_string().substr(0,12)));

                    cout<<"RS1 = "<<(int)state.EX.Read_data1.to_ulong()<<", RS2 = "<<(int)state.EX.Read_data2.to_ulong()<<"\n";
                // We will resolve the branch here:
                if ((func3 == "000") & (state.EX.Read_data1 == state.EX.Read_data2)) { //BEQ
                    // Discard instruction fetched this cycle, update PC
                    state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                    //state.ID.Instr = ext_imem.readInstr(state.IF.PC);
                    //stall = true;
                    state.EX.NOP = true;
                    cout<<"Branch Taken: BEQ\n";

                } else if ((func3 == "001") & (state.EX.Read_data1 != state.EX.Read_data2)) { //BNE
                    // Discard instruction fetched this cycle, update PC & instr from prev cycle
                    state.IF.PC = bitset<32>(state.IF.PC.to_ulong() + state.EX.Imm.to_ulong() - 4);
                    state.EX.NOP = true;
                    cout<<"Branch Taken: BNE\n";
                } else {
                    cout << "No Branch Taken\n";
                }
                break;

            case s_type: cout << "Executing S type instr\n";
                // Control signal set
                state.EX.is_I_type = false;
                state.EX.rd_mem    = false;
                state.EX.wrt_mem   = true;
                state.EX.wrt_enable = false;

                // Set ALU OP
                state.EX.alu_op = ADDI;

                // Descramble the 12 bit IMM<11:5>  <4:0>
                temp_imm_s = (bitset<12> (state.ID.Instr.to_string().substr(0, 7) + // Bits 11:5
                                        state.ID.Instr.to_string().substr(20, 5)).to_ulong()); //bits 4:0

                //Sign extend and load the immediate
                state.EX.Imm =   (((bitset<1>)(temp_imm_s.to_string().substr(0,1))).to_ulong())
                                ? ((bitset<32>)(string(20,'1') + temp_imm_s.to_string().substr(0,12)))
                                : ((bitset<32>)(temp_imm_s.to_string().substr(0,12)));
                break;

            default: cout << "Executing HALT instr\n";
                // Halt // Error

                state.EX.rd_mem    = false;
                state.EX.wrt_mem   = false;
                state.EX.wrt_enable = false;
                break;

        }
    } if(!stall) state.ID.NOP = state.IF.NOP;


    /* --------------------- IF stage --------------------- */
    if(!state.IF.NOP){
        if(stall){
            stall = false;
            // Execute the prev stalled instruction again
            state.ID.NOP = state.IF.NOP;
        } else {
            state.ID.Instr = ext_imem.readInstr(state.IF.PC);
            state.IF.PC = (bitset<32>(state.IF.PC.to_ulong() + 4));
            instr_count++;
        }
    }
    /* --------------------- 5 Stage End ------------------- */



    if (state.IF.NOP && state.ID.NOP && state.EX.NOP && state.MEM.NOP && state.WB.NOP){
        halted = true;
        ext_dmem.outputDataMem();
    }
    myRF.outputRF(cycle); // dump RF
    printState(state, cycle); //print states after executing cycle 0, cycle 1, cycle 2 ...

    cycle++;
}

void printState(stateStruct state, int cycle) {
    ofstream printstate;
    if (cycle == 0)
        printstate.open(opFilePath, std::ios_base::trunc);
    else
        printstate.open(opFilePath, std::ios_base::app);
    if (printstate.is_open()) {
        printstate<<"\nState after executing cycle:\t"<<cycle<<endl;

        printstate<<"IF.PC:\t"<<state.IF.PC.to_ulong()<<endl;
        printstate<<"IF.NOP:\t"<<state.IF.NOP<<endl;

        printstate<<"ID.Instr:\t"<<state.ID.Instr<<endl;
        printstate<<"ID.NOP:\t"<<state.ID.NOP<<endl;

        printstate<<"EX.Read_data1:\t"<<state.EX.Read_data1<<endl;
        printstate<<"EX.Read_data2:\t"<<state.EX.Read_data2<<endl;
```

```
                printstate<<"EX.Imm:\t"<<state.EX.Imm<<endl;
                printstate<<"EX.Rs:\t"<<state.EX.Rs<<endl;
                printstate<<"EX.Rt:\t"<<state.EX.Rt<<endl;
                printstate<<"EX.Wrt_reg_addr:\t"<<state.EX.Wrt_reg_addr<<endl;
                printstate<<"EX.is_I_type:\t"<<state.EX.is_I_type<<endl;
                printstate<<"EX.rd_mem:\t"<<state.EX.rd_mem<<endl;
                printstate<<"EX.wrt_mem:\t"<<state.EX.wrt_mem<<endl;
                printstate<<"EX.alu_op:\t"<<state.EX.alu_op<<endl;
                printstate<<"EX.wrt_enable:\t"<<state.EX.wrt_enable<<endl;
                printstate<<"EX.NOP:\t"<<state.EX.NOP<<endl;

                printstate<<"MEM.ALUresult:\t"<<state.MEM.ALUresult<<endl;
                printstate<<"MEM.Store_data:\t"<<state.MEM.Store_data<<endl;
                printstate<<"MEM.Rs:\t"<<state.MEM.Rs<<endl;
                printstate<<"MEM.Rt:\t"<<state.MEM.Rt<<endl;
                printstate<<"MEM.Wrt_reg_addr:\t"<<state.MEM.Wrt_reg_addr<<endl;
                printstate<<"MEM.rd_mem:\t"<<state.MEM.rd_mem<<endl;
                printstate<<"MEM.wrt_mem:\t"<<state.MEM.wrt_mem<<endl;
                printstate<<"MEM.wrt_enable:\t"<<state.MEM.wrt_enable<<endl;
                printstate<<"MEM.NOP:\t"<<state.MEM.NOP<<endl;

                printstate<<"WB.Wrt_data:\t"<<state.WB.Wrt_data<<endl;
                printstate<<"WB.Rs:\t"<<state.WB.Rs<<endl;
                printstate<<"WB.Rt:\t"<<state.WB.Rt<<endl;
                printstate<<"WB.Wrt_reg_addr:\t"<<state.WB.Wrt_reg_addr<<endl;
                printstate<<"WB.wrt_enable:\t"<<state.WB.wrt_enable<<endl;
                printstate<<"WB.NOP:\t"<<state.WB.NOP<<endl;
            }
            else cout<<"Unable to open FS StateResult output file." << endl;
            printstate.close();
        }
    private:
        string opFilePath;
};

int main(int argc, char* argv[]) {


    string ioDir = "";

    if (argc == 1) {
        cout << "Enter path containing the memory files: ";
        cin >> ioDir;
    }
    else if (argc > 2) {
        cout << "Invalid number of arguments. Machine stopped." << endl;
        return -1;
    }
    else {
        ioDir = argv[1];
        cout << "IO Directory: " << ioDir << endl;
    }

    InsMem imem = InsMem("Imem", ioDir);
    DataMem dmem_ss = DataMem("SS", ioDir);
    DataMem dmem_fs = DataMem("FS", ioDir);

    SingleStageCore SSCore(ioDir, imem, dmem_ss);
    FiveStageCore FSCore(ioDir, imem, dmem_fs);

    while (1) {
        if (!SSCore.halted)
            SSCore.step();

        if (!FSCore.halted)
            FSCore.step();

        if (SSCore.halted && FSCore.halted)
            break;
    }

    // Here, we will output the Performance Metrics after both
    // Processors are run to completion

    float CPI_SS = ((float)(SSCore.cycle)) / ((float)(SSCore.instr_count));
    float IPC_SS = ((float)(SSCore.instr_count)) / ((float)(SSCore.cycle));


    cout<<"\n------------------ Single Stage Core -----------------------\n";
    cout<<"Total Cycles Taken SS = \t"<<SSCore.cycle<<"\n";
    cout<<"Cycles Per Instruction SS = \t"<<CPI_SS<<"\n";
    cout<<"Instructions Per Cycle SS = \t"<<IPC_SS<<"\n";

    float CPI_FS = ((float)(FSCore.cycle)) / ((float)(FSCore.instr_count));
    float IPC_FS = ((float)(FSCore.instr_count)) / ((float)(FSCore.cycle));

    cout<<"------------------ 5 Stage Stage Core -----------------------\n";
    cout<<"Total Cycles Taken FS = \t"<<FSCore.cycle<<"\n";
    cout<<"Cycles Per Instruction FS = \t"<<CPI_FS<<"\n";
    cout<<"Instructions Per Cycle FS = \t"<<IPC_FS<<"\n";

    ofstream metricsFile;
        metricsFile.open (ioDir + "\\PerfMetrics.txt");
        metricsFile<<"\n------------------ Single Stage Core -----------------------\n";
        metricsFile<<"Total Cycles Taken SS = \t"<<SSCore.cycle<<"\n";
        metricsFile<<"Cycles Per Instruction SS = \t"<<CPI_SS<<"\n";
        metricsFile<<"Instructions Per Cycle SS = \t"<<IPC_SS<<"\n";
        metricsFile<<"------------------ 5 Stage Stage Core -----------------------\n";
        metricsFile<<"Total Cycles Taken FS = \t"<<FSCore.cycle<<"\n";
        metricsFile<<"Cycles Per Instruction FS = \t"<<CPI_FS<<"\n";
        metricsFile<<"Instructions Per Cycle FS = \t"<<IPC_FS<<"\n";
        metricsFile.close();

}
```