ALU and MIPS Processor

Student: Xiangyi Li

ID: 119010153

ALU

The goal of this part of the assignment is to implement an arithmetic-logic unit, which is an essential component in modern computer CPU. In the process I have also learned a lot about verilog and bit computation and manipulation when I am tweaking with testbenches.

Execution

To run the `ALU.v` program, run these commands as stated in the `alu/makefile`:

```
test:compile; vvp test_alu;
compile: alu.v test_alu.v;
    iverilog -g2012 -o test_alu test_alu.v
```

which is:

```
make test
```

This command will run my custom testbench `test_alu.v`, which utilizes the `$write` and `$display` functions to print out a CSV-compatible logging, for example:

```
➜  alu git:(master) ✗ make test
iverilog -g2012 -o test_alu test_alu.v
test_alu.v:30: warning: Static variable initialization requires explicit lifetime in this context.
vvp test_alu;
add, 00000000001000100001100000100000, 00000003, 00000002, 00, 20, 00000005, 000, PASS
addi, 00100000001000111111111111111110, 00000003, 00000000, 08, 3e, 00000001, 000, PASS
addu, 00000000001000100001100000100001, 00000003, 00000002, 00, 21, 00000005, 000, PASS
addiu, 00100100001000111111111111111110, 00000003, 00000000, 09, 3e, 00000001, 000, PASS
sub, 00000000001000100001100000100010, 00000003, 00000002, 00, 22, 00000001, 000, PASS
subu, 00000000001000100001100000100011, 00000003, 00000002, 00, 22, 00000001, 000, PASS
and, 00000000001000100001100000100100, 00000005, 00000003, 00, 24, 00000001, 000, PASS
andi, 00110000001000110000000000000011, 00000005, 00000000, 0c, 03, 00000001, 000, PASS
nor, 00000000001000100001100000100111, 00000005, 00000003, 00, 27, fffffff8, 000, PASS
ori, 00110100001000110000000000000011, 00000005, 00000000, 0d, 03, 00000007, 000, PASS
xor, 00000000001000100001100000100110, 00000005, 00000003, 00, 26, 00000006, 000, PASS
xori, 00111000001000110000000000000011, 00000005, 00000000, 0e, 03, 00000006, 000, PASS
beq, 00010000001000001000000000100000, 00000002, 00000002, 04, 20, 00000000, 100, PASS
bne, 00010100001000001000000000100000, 00000002, 00000002, 04, 20, 00000000, 100, PASS
slt, 00101000000000011000000000000001, 00000002, 00000002, 04, 20, 00000000, 100, PASS
slti, 00101000000000011000000000000001, 00000002, 00000002, 04, 20, 00000000, 100, PASS
sltiu, 00101100000000011111111111111111, 00000003, 00000003, 0b, 3f, 00000001, 000, PASS
sltu, 00000000001000000000000000101011, ffffffff, fffffffd, 00, 2b, 00000000, 100, PASS
```

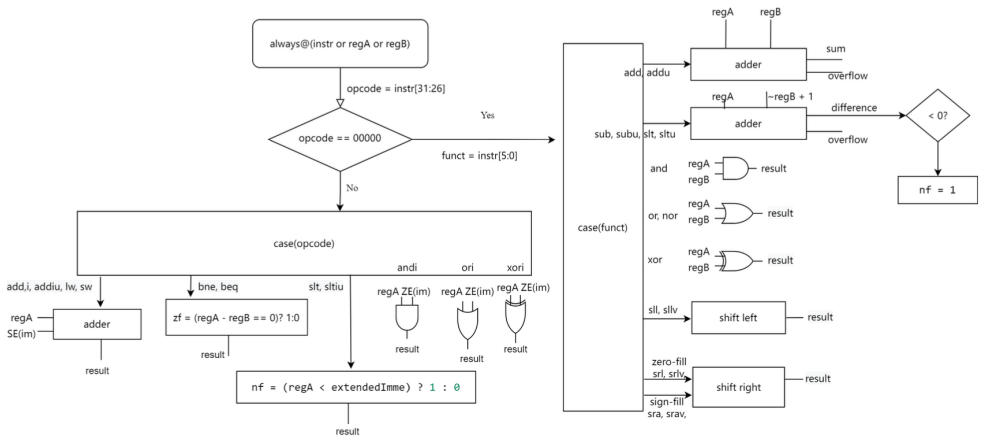
```
lw, 10001100001000001111111111111111, xxxxxxxx, fffffffd, 23, 3f, fffffffc, 000, PASS
sw, 10101100001000001111111111111111, xxxxxxxx, fffffffd, 23, 3f, fffffffc, 000, PASS
sll, 00000000001000000000000010000000, 00000001, 00000001, 00, 00, 00000004, 000, PASS
sllv, 00000000000000001000000000000100, 00000004, 00000001, 00, 04, 00000010, 000, PASS
srl, 00000000000000001000000010000010, xxxxxxxx, 00000008, 00, 02, 00000002, 000, PASS
srlv, 00000000000000001000000000000110, 00000002, 00000008, 00, 06, 00000002, 000, PASS
sra, 00000000000000001000000010000011, xxxxxxxx, 00000008, 00, 03, 00000002, 000, PASS
srav, 00000000000000001000000000000111, 00000004, 0000000f, 00, 07, 00000000, 100, PASS
```

The testing statements could be easily modified. It's also possible to massively generate testing statements using python and templating text files.

Design

The high level design of the program follows the switch-case statments data flow as specified in the tutorial. With three inputs including 32-bit long instruction, regA, and regB, a 32-bit long output result, and a 3-bit long flag output.

Data flow



Implementation

Since in verilog, `assign` is not available in an `always` statements, I used two `reg` variables to store values of computation and corresponding flags to assign to the two output wires after the sequential logic:

```
reg[31:0] alu_result;
reg[2:0] flags_reg;
```

Here's parsing command:

```
reg[5:0] opcode;
reg[5:0] funct;
reg[31:0] shamt;
reg[15:0] imm;
reg[31:0] imm_ext;
reg[31:0] reg0;

flags_reg = 3'b000;
opcode = instruction[31:26];
funct = instruction[5:0];
imm = instruction[15:0];
imm_ext = { {16{imm[15]}} , imm };
shamt = instruction[10:6];
```

For instructions that only uses one register, I also mimiced the tutorial to detect which register is being deployed, for example:

```
6'b101011: begin // alu_result = regA + imm_ext; // sw
    reg0 = $signed(instruction[15:0]);
    if(instruction[25:21] == 5'b00000)
        alu_result = $signed(regA) + $signed(reg0);
    if(instruction[25:21] == 5'b00001)
        alu_result = $signed(regB) + $signed(reg0);
end
```

5-Stage CPU

The author attempted to solve R-Type hazards with forwarding by implementing a `ForwardingUnit` module. The other types of hazards are not looked at due to limited amount of time.

Execution

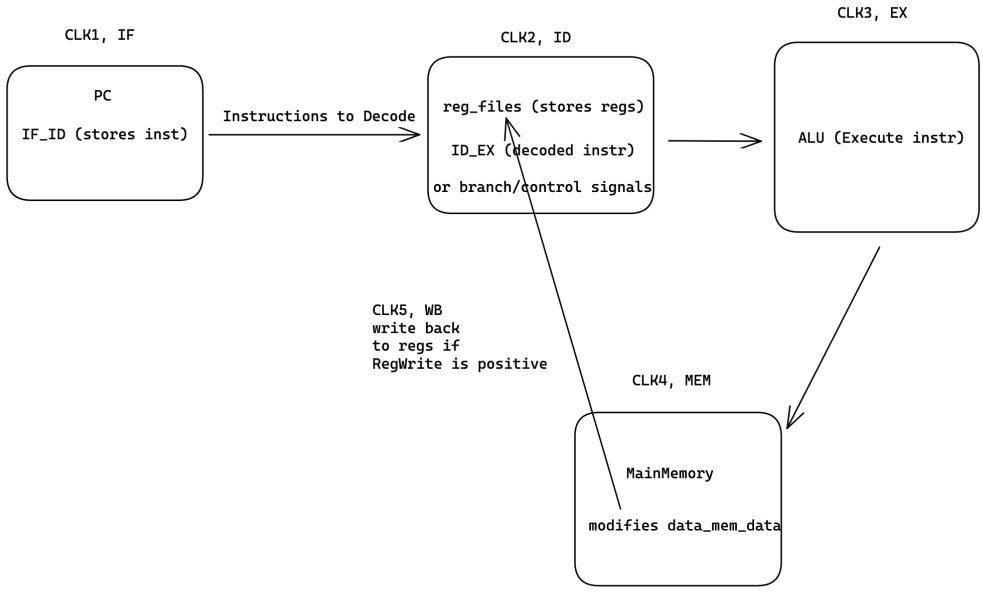
The execution is similar to the alu program, utilizing the SystemVerilog flag as shown in `cpu/makefile`.

```
test:compile; vvp test_cpu.vvp;
compile: cpu.v test_cpu.v;
    iverilog -g2012 -o test_cpu.vvp test_cpu.v
```

The program can be tested manually by copying `machine_code${number}.txt` to `CPU_instruction.bin` as the instructions, and compare the dumped result at `data.bin` with `DATA_RAM${number}.txt`. Currently the program is failing all testcases, and the author was encountering massive trouble trying to produce waveforms on a macOS system with constant failing of dumping the `dump.vvd` file in the custom `test_cpu.v` testbench.

Design

Here’s a simplified version of the design spec of the assignment requirements.



Implementation

The author tried to use a top-down approach to design the CPU, since it could offer modularity through the Object-Oriented paradigm:

```
module CPU (input CLK);
    // Instruction Memory
    wire [31:0] instruction;
    reg [31:0] pc = 0;
    wire [31:0] fetch_address = pc >> 2;
    reg [31:0] register_file [0:31];
    InstructionRAM instruction_ram (
        .CLOCK(CLK),
        .RESET(1'b0),
        .ENABLE(1'b1),
        .FETCH_ADDRESS(fetch_address),
        .DATA(instruction));

    // Data Memory
    wire [31:0] data_mem_data;
    reg [31:0] data_mem_address;
    reg [64:0] data_mem_edit_serial;
    // $display(data_mem_data);
    MainMemory data_memory (
        .CLOCK(CLK),
        .RESET(1'b0),
        .ENABLE(1'b1),
        .FETCH_ADDRESS(data_mem_address),
        .EDIT_SERIAL(data_mem_edit_serial),
        .DATA(data_mem_data));
```

```
    // Pipeline Stages
    reg [31:0] IF_ID;
    reg [95:0] ID_EX;
    reg [63:0] EX_MEM;
    reg [95:0] MEM_WB;
    wire [31:0] write_data;
    reg [4:0] write_reg;

    // Stage 1: Instruction Fetch (IF)
    always @(posedge CLK) begin
        IF_ID <= instruction;
        pc <= pc + 4;
    end

    // Stage 2: Instruction Decode (ID)
    ID_stage ID_stage_inst (
        .CLOCK(CLK),
        .instruction(IF_ID),
        .write_data(write_data),
        .write_reg(write_reg),
        .ID_EX(ID_EX));

    // Stage 3: Execute (EX)
    EX_stage EX_stage_inst (
        .CLOCK(CLK),
        .ID_EX(ID_EX),
        .EX_MEM(EX_MEM));

    // Stage 4: Memory Access (MEM)
    MEM_stage MEM_stage_inst (
        .CLOCK(CLK),
        .EX_MEM(EX_MEM),
        .data_mem_data(data_mem_data),
        .MEM_WB(MEM_WB));

    // Stage 5: Write Back (WB)
    reg RegWrite;
    WB_stage WB_stage_inst (
        .CLOCK(CLK),
        .MEM_WB(MEM_WB),
        .write_data(write_data),
        .write_register(write_reg),
        .RegWrite(RegWrite));

    // Connect Data Memory to the MEM stage
    always @(posedge CLK) begin
        data_mem_address <= EX_MEM[31:0];
        data_mem_edit_serial <= {EX_MEM[63], EX_MEM[62:31]};
    end

    // Write to register file
    always @(posedge CLK) begin
        if (RegWrite) begin
            register_file[write_reg] <= write_data;
        end
    end
endmodule
```

And in connecting the necessary modules, the author pipes the datapath in a `{PREVIOUS_STAGE}_{NEXT_STAGE}` variable that varies in length to mimic the piping operations

commonly found in functional programming. This is inconvenient sometimes since the length of the bit number could be as long as 96 bits.

To deal with hazards, the author implemented a forwarding unit in the ID stage, though commonly it should be implemented in EX or MEM, but I didn’t have time to try out different implementations.

The author also re-impelemented the ALU using combinatorial logic since I have come to realize it should be a more natural way to program hardwares.