MP4 Design Checkpoint

November 2nd, 2020

Alex Vetsavong
Mohan Li
Peter Kircher

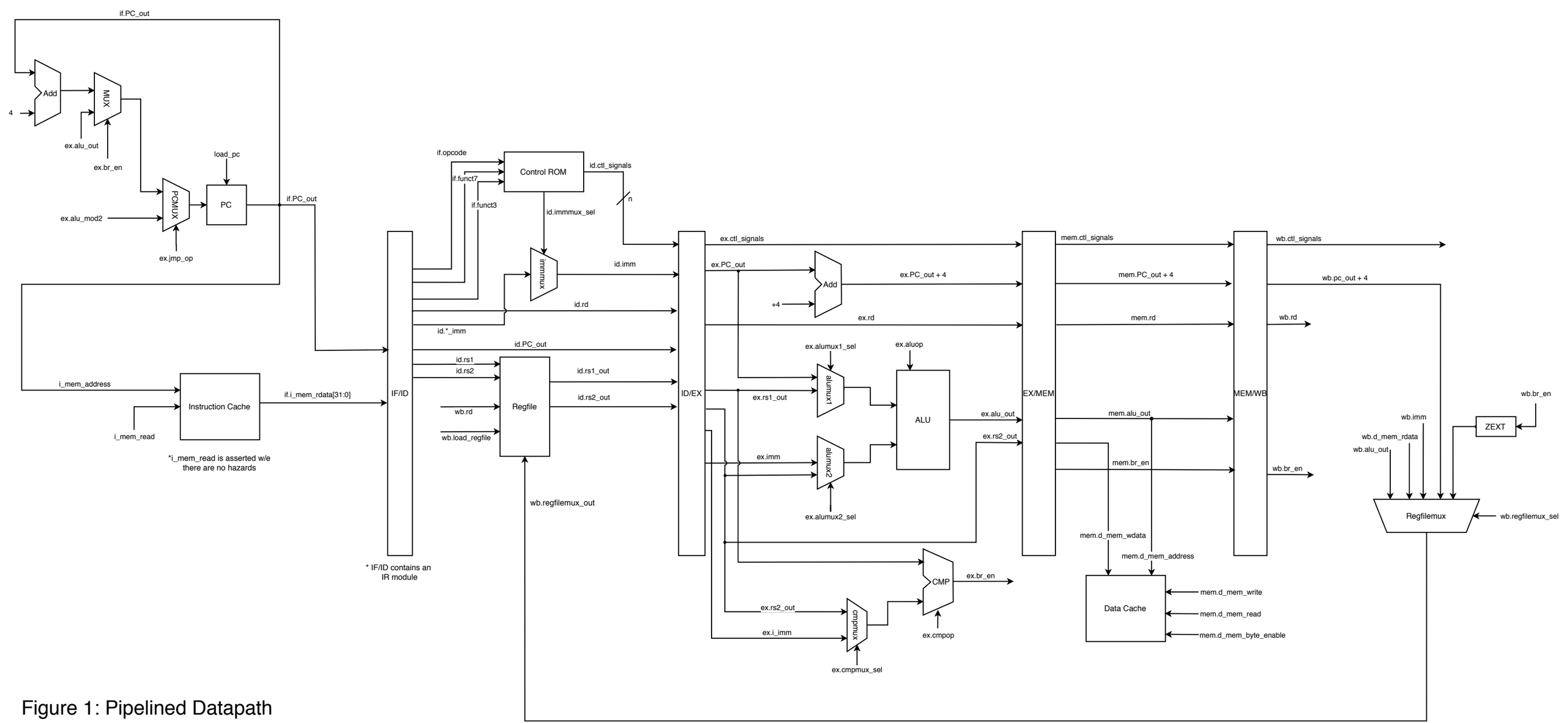

Figure 1: Pipelined Datapath

Control ROM

The prefixes for the control signals only indicate where they will be used, to be consistent with the datapath diagram. In implementation, all the signals will be set at the ID stage, where the Control ROM will reside, and passed down the pipeline.

```
module control_rom(
input rv32i_opcode opcode,
input logic [2:0] funct3,
input logic [6:0] funct7,

output logic mem_read, mem_write,
output logic [3:0] mem_byte_enable,
output immmux:immmux_sel_t immmux_sel,
output alumux::alumux1_sel_t alumux1_sel,
output alumux::alumux2_sel_t alumux2_sel,
output regfilemux::regfilemux_sel_t regfilemux_sel,
output cmpmux::cmpmux_sel_t cmpmux_sel,
output alu_ops aluop,
output branch_funct3_t cmpop,
output logic load_regfile
);
// default values
// id.jmp_op = 0
// id.immmux_sel = i_imm
// ex.aluop = alu_add
// ex.alumux1_sel = rs1_out
// ex.alumux2_sel = rs2_out
// ex.cmpop = beq
// mem.d_mem_read = 0
// mem.d_mem_write = 0
// mem.d_mem_byte_enable = 0000
// wb.regfilemux_sel = alu_out
// wb.load_regfile = 0
case(opcode)
    // values for each opcode
    lui:
         // id.immmux_sel = u_imm
         // wb.regfilemux_sel = imm
         // wb.load_regfile = 1
    auipc:
         // id.immmux_sel = u_imm
         // ex.aluop = alu_add
```

```
        // ex.alumux1_sel = pc_out
        // ex.alumux2_sel = imm
        // wb.regfilemux_sel = alu_out
        // wb.load_regfile = 1
    jal:
        // id.immmux_sel = j_imm
        // ex.jmp_op = 1
        // ex.aluop = alu_add
        // ex.alumux1_sel = pc_out
        // ex.alumux2_sel = imm
        // wb.regfilemux_sel = wb.pc_out + 4
        // wb.load_regfile = 1

    jalr:    (uses the I-type encoding)
        // id.immmux_sel = i_imm
        // ex.jmp_op = 1
        // ex.aluop = alu_add
        // ex.alumux1_sel = rs1_out
        // ex.alumux2_sel = imm
        // wb.regfilemux_sel = wb.pc_out + 4
        // wb.load_regfile = 1
    br:
        // id.immmux_sel = b_imm
        // ex.cmpop = (correct instruction)
        // ex.cmpmux_sel = i_imm
        // ex.aluop = alu_add
        // ex.alumux1_sel = pc_out
        // ex.alumux2_sel = imm
    load:
        // id.immmux_sel = i_imm
        // ex.alumux1_sel = rs1_out
        // ex.alumux2_sel = imm
        // mem.d_mem_address = alu_out
        // mem.d_mem_read = 1
        // mem.d_mem_byte_enable = rmask
        // wb.regfilemux_sel = wb.d_mem_rdata
        // wb.load_regfile = 1
    store:
        // id.immmux_sel = s_imm
        // ex.alumux1_sel = rs1_out
        // ex.alumux2_sel = imm
        // mem.d_mem_address = alu_out
        // mem.d_mem_write = 1
```

```
        // mem.d_mem_wdata = rs2_out
        // mem.d_mem_byte_enable = wmask
        // wb.load_regfile = 0
    imm:
        case(funct3)
            slt:
                // id.immmux_sel = i_imm
                // ex.cmpop = blt
                // ex.alumux1_sel = rs1_out
                // ex.alumux2_sel = imm
                // wb.load_regfile = 1
            sltu:
                // id.immmux_sel = i_imm
                // ex.cmpop = bltu
                // ex.alumux1 = rs1_out
                // ex.alumux2 = imm
                // wb.load_regfile = 1

            sr:
                // id.immmux_sel = i_imm
                // ex.alumux1 = rs1_out
                // ex.alumux2 = imm
                // ex.aluop = alu_sra
                // wb.load_regfile = 1
            default:
                // id.immmux_sel = i_imm
                // ex.alumux1 = rs1_out
                // ex.alumux2 = imm
                // ex.aluop = funct3
                // wb.load_regfile = 1
        endcase
    regs:
        case(funct3)
            slt:
                // ex.cmpop = blt
                // ex.alumux1 = rs1_out
                // ex.alumux2 = rs2_out
                // wb.load_regfile = 1
            sltu:
                // ex.cmpop = bltu
                // ex.alumux1 = rs1_out
                // ex.alumux2 = rs2_out
                // wb.load_regfile = 1
```

```
            sr:
                // ex.alumux1 = rs1_out
                // ex.alumux2 = rs2_out
                // ex.aluop = alu_sra
                // wb.load_regfile = 1
            add:
                // ex.alumux1 = rs1_out
                // ex.alumux2 = rs2_out
                // ex.alu_op = alu_add or alu_sub
                // wb.load_regfile = 1
            default:
                // ex.alumux1 = rs1_out
                // ex.alumux2 = rs2_out
                // ex.alu_op = funct3
                // wb.load_regfile = 1
        endcase
endcase
endmodule;
```

When implemented, ideally instead of individual signals, a typedef struct will be defined for readability in the port instantiations.

IF/ID Register : Example for Pipeline Registers

```
module IF.ID(
    input clk, rst, load,
    input logic [31:0] IF.mem_rdata,
    input logic [31:0] IF.PC,
    output logic [31:0] ID.mem_rdata,
    output logic [31:0] ID.PC,
    output logic [2:0] funct3,
    output logic [6:0] funct7,
    output rv32i_opcode opcode,
    output logic [31:0] i_imm,
    output logic [31:0] s_imm,
    output logic [31:0] b_imm,
    output logic [31:0] u_imm,
    output logic [31:0] j_imm,
    output logic [4:0] rs1,
    output logic [4:0] rs2,
    output logic [4:0] rd
);
logic [31:0] pc, mem_rdata;

assign ID.PC = pc;
assign ID.mem_rdata = mem_rdata;

//IR instantiation as a part of the pipeline register
IR IF.ID_IR(.*); // will explicitly instantiate ports in actual code

always_ff(@posedge clk) begin
    mem_rdata <= IF.mem_rdata;
    pc <= IF.PC;
end

endmodule
```

A basic look at what could be passed into the IF/ID pipelining register. Here, we take advantage of the given IR code to break it up into several registers and make the code more readable.

A similar approach will be taken with the rest of the pipeline registers, based on our datapath design.