50条指令流水线CPU设计

实验目的

本次实验利用verilog语言在Vivado上实现了五级流水线CPU设计，支持MIPS汇编50条指令。

模块设计

我根据五级流水线将CPU分成五个阶段实现，每个阶段用流水线寄存器存储相应控制信号。

IF阶段

顶层模块实现

```
// 通过上一条指令传出的PCSrc确定当前PC的选择
MUX4X32_addr mux4x32(IF_PCAdd4, ID_B, ID_J, ID_rs, ID_PCSrc, IF_Choose);
// 根据stall与block确定流水线是否继续或是将流水线冻结在IF阶段
PC PC(IF_Choose, clock, reset, IF_PC, stall, stall2, block);
// 通过PC求出下一条指令地址
PCAdd4 PCAdd4(IF_PC, IF_PCAdd4);
// 通过PC得到当前指令机器代码
InstMemory inst_memory(IF_PC, IF_Inst);
// 将IF阶段控制信号传到ID阶段
REG_IF_ID REG_IF_ID(IF_PCAdd4, IF_Inst, IF_PC, clock, reset, ID_PCAdd4, ID_Inst,
ID_PC, stall, stall2, block);
```

PC

PC模块需要对stall，stall2和block三个信号值进行判断，若其中一个信号为1，则代表流水线需要在IF阶段冻结。

```
module PC(NextPC, clock, reset, address, stall, stall2, block);
    input clock;
    input reset;
    input[31:0] NextPC;
    input stall, stall2, block;
    output reg[31:0] address;

    wire enable;

    assign enable = ~stall & ~stall2 & ~block;

    initial begin
        address <= 32'h00003000;
    end

    always @(posedge clock) begin
        if(enable == 1) begin
            address <= NextPC;
        end
    end

    always @(posedge reset) begin
        address <= 32'h00003000;
    end
endmodule
```

PCAdd4

传入PC，调用32位同步进位加法器获得下一条指令地址

```
module PCAdd4(PC, PCAdd4);
    input [31:0] PC;
    output [31:0] PCAdd4;
    Adder adder(PC, 32'b100, PCAdd4);
endmodule
```

InstMemory

在初始化的时候读入代码文件，之后在PC对应位置读出指令

```
module InstMemory(Addr, Inst);
    input [31:0] Addr;
    reg [31:0] Rom[2047:0];

    output [31:0] Inst;

    integer i;
    initial begin
        $display("start simulation");
        for (i = 0; i < 2048; i = i + 1)
            Rom[i] = 32'b0;
        $readmemh("C:\\Users\\HBW\\Desktop\\pipeline-tester-py\\code.txt", Rom);
    end

    assign Inst = Rom[Addr[12:2] - 11'b10000000000];

endmodule
```

REG_IF_ID

该模块将IF阶段的控制信号转移到ID阶段，当enable为1时传递，否则冻结。

```
module REG_IF_ID (IF_PCAdd4, IF_Inst, IF_PC, clock, reset, ID_PCAdd4, ID_Inst,
ID_PC, stall, stall2, block);
    input [31:0] IF_PCAdd4, IF_Inst, IF_PC;
    input clock, reset, stall, stall2, block;
    output reg [31:0] ID_PCAdd4, ID_Inst, ID_PC;

    wire enable;
    assign enable = ~stall & ~stall2 & ~block;
    initial begin
        ID_Inst = 0;
        ID_PCAdd4 = 0 ;
        ID_PC = 0;
    end

    always @(posedge clock) begin
        if(enable == 1)begin
            ID_PCAdd4 = IF_PCAdd4;
            ID_Inst = IF_Inst;
            ID_PC = IF_PC;
        end
    end
```

```
    always @(posedge reset) begin
        ID_Inst = 0;
        ID_PCAdd4 = 0 ;
        ID_PC = 0;
    end
endmodule
```

ID阶段

顶层模块

```
// 通过指令机器码获取控制信号
CU CU(ID_PC, ID_Inst, ID_Inst[5:0], busy, EX_md, ID_B_code, ID_RegDst, Se,
ID_RegWrite, ID_ALUXSrc, ID_ALUYSrc, ID_ALUControl, ID_MemWrite, ID_PCSrc,
ID_MemtoReg, ID_load_option, ID_save_option, ID_usigned, c_adventure,
ID_md_control, ID_start, ID_md, EX_md_control[2], block);

MUX2X5 mux2x5(ID_Inst[15:11], ID_Inst[20:16], ID_RegDst, ID_WriteReg);

// 访问寄存器读写，写所用控制型号为WB阶段，同时将相应寄存器读入Qa，Qb
RegisterFile register_file(ID_Inst[25:21], ID_Inst[20:16], ID_Inst[15:11],
WriteInput, WB_WriteReg, WB_RegWrite, clock, reset, ID_Qa, ID_Qb, WB_PCSrc,
WB_PC, WB_PC);

// 判断是否发生控制冒险
if_c_adventure if_c_adventure(ID_rs, ID_rt, ID_ALUControl, ID_usigned,
c_adventure);

// 旁路转发控制模块
FU FU(EX_RegWrite, EX_WriteReg, EX_MemtoReg, ME_RegWrite, ME_WriteReg,
ME_MemtoReg, EX_PCSrc, ME_PCSrc, ID_Inst[25:21], ID_Inst[20:16], ID_FA, ID_FB,
ID_Inst[31:26], ID_Inst[5:0], c_adventure, stall, stall2);

// 通过转发信号对Qa，Qb值进行选择，用于控制冒险提前
MUX3X32 mux3x32_ID_X(ID_Qa, EX_Alu_Result, ME_Alu_Result, ID_FA, ID_rs);
MUX3X32 mux3x32_ID_Y(ID_Qb, EX_Alu_Result, ME_Alu_Result, ID_FB, ID_rt);

// 数据扩展模块
EXT16T32 ext16t32(ID_Inst[15:0], Se, ID_Ext32);
EXT5T32 ext5t32(ID_Inst[10:6], ID_sa);
Shifter32L2 shifter(ID_Ext32, ID_Ext32_L2);

// 计算指令跳转地址
Adder adder(ID_PCAdd4, ID_Ext32_L2, ID_B);
ShifterCombination get_j_address(ID_Inst[25:0], ID_PCAdd4, ID_J);

// 将ID阶段控制信号传递给EX阶段
REG_ID_EX REG_ID_EX(ID_B_code, ID_sa, ID_RegDst, ID_RegWrite, ID_ALUXSrc,
ID_ALUYSrc, ID_ALUControl, ID_MemWrite, ID_MemtoReg, ID_WriteReg, ID_usigned,
ID_Qa, ID_Qb, ID_Ext32, ID_FA, ID_FB, ID_load_option, ID_save_option, ID_PC,
ID_PCSrc, ID_md_control, ID_start, ID_md, clock, reset,
```

```
            EX_B_code, EX_sa, EX_RegDst, EX_RegWrite, EX_ALUXSrc, EX_ALUYSrc,
EX_ALUControl, EX_MemWrite, EX_MemtoReg, EX_WriteReg, EX_usigned, EX_Qa, EX_Qb,
EX_Ext32, EX_FA, EX_FB, EX_load_option, EX_save_option, EX_PC, EX_PCSrc,
EX_md_control, EX_start, EX_md, stall, stall2, block);
```

CU

我首先根据指令机器码将各指令单独求出，之后对控制信号再对各个指令进行组合。

控制信号

- ID_Inst：ID阶段指令
- RegDst：寄存器写回地址
- Se：判断扩展单元是零扩展还是符号扩展
- RegWrite：判断是否需要写回寄存器堆
- ALUXSrc：控制ALU的第一个操作数是来自寄存器堆还是扩展单元
- ALUOp：控制ALU进行运算的类型
- MemWrite：判断是否需要对数据内存进行写操作
- PCSrc：判断下一条指令地址来源
- MemtoReg：判断写回寄存器堆的数据来源
- B_code：用来区分bltz指令和bgez指令
- ALUXSrc：ALU的第一个输入的来源
- load_option：用来判断load指令的类型
- save_option：用来判断save指令的类型
- usigned：判断指令是否是unsigned指令
- md_control：标记当前是乘除指令的哪一条
- start：当前指令进入EX阶段后需要访问MDU
- block：当前阻塞乘除指令

```
module CU(PC, Inst, Func, busy, EX_md, B_code, RegDst, Se, RegWrite, ALUXSrc,
ALUYSrc, ALUControl, MemWrite, PCSrc, MemtoReg, load_option, save_option,
usigned, c_adventure, md_control, md_start, is_md, next_inst, block);
    input [5:0] Func;
    input [31:0] PC, Inst;
    input c_adventure, busy, EX_md, next_inst;
    output RegDst, Se, RegWrite, ALUXSrc, ALUYSrc, MemWrite, MemtoReg, usigned,
md_start, B_code, block, is_md;
    output [2:0] PCSrc, load_option, md_control;
    output [1:0] save_option;
    output [3:0] ALUControl;

    wire R_type=~Inst[31] & ~Inst[30] & ~Inst[29] & ~Inst[28] & ~Inst[27] &
~Inst[26];
    wire ID_lb = Inst[31] & ~Inst[30] & ~Inst[29] & ~Inst[28] & ~Inst[27] &
~Inst[26];
    // 其他控制信号
    ...
```

```
    assign RegDst = ID_lb | ID_lbu | ID_lh | ID_lhu | ID_lw | ID_addi | ID_addiu
| ID_andi | ID_ori | ID_xori | ID_lui | ID_slti | ID_sltiu;
    assign Se = ID_lb | ID_lbu | ID_lh | ID_lhu | ID_lw | ID_sb | ID_sh | ID_sw |
ID_addi | ID_addiu | ID_slti | ID_sltiu | ID_beq | ID_bne | ID_blez | ID_bgtz |
ID_bltz | ID_bgez;
    assign RegWrite = ID_lb | ID_lbu | ID_lh | ID_lhu | ID_lw | ID_add | ID_addu
| ID_sub | ID_subu | ID_sll | ID_srl | ID_sra | ID_sllv | ID_srlv | ID_srav |
ID_and | ID_or | ID_xor | ID_nor | ID_addi | ID_addiu | ID_andi | ID_ori |
ID_xori | ID_lui | ID_slt | ID_slti | ID_sltiu | ID_sltu | ID_mfhi | ID_mflo |
ID_jalr;
    assign ALUXSrc = ID_sll | ID_srl | ID_sra;
    assign ALUYSrc = ID_add | ID_addu | ID_sub | ID_subu | ID_and | ID_or |
ID_xor | ID_nor | ID_slt | ID_sltu | ID_beq | ID_bne | ID_j | ID_jal | ID_jalr |
ID_jr | ID_sll | ID_srl | ID_sra | ID_sllv | ID_srlv | ID_srav;
    assign ALUControl[0] =ID_sub | ID_subu | ID_sll | ID_sllv | ID_or | ID_nor |
ID_ori | ID_slt | ID_slti | ID_sltiu | ID_sltu | ID_beq | ID_bne | ID_bgtz |
ID_bgez;
    assign ALUControl[1] =ID_sll | ID_sllv | ID_and | ID_or | ID_andi | ID_ori |
ID_lui | ID_blez | ID_bgtz | ID_sra | ID_srav;
    assign ALUControl[2] =ID_sll | ID_sllv | ID_xor | ID_nor | ID_xori | ID_lui |
ID_bltz | ID_bgez | ID_sra | ID_srav;
    assign ALUControl[3] =ID_srl | ID_sra | ID_srlv | ID_srav | ID_slt | ID_slti
| ID_sltiu | ID_sltu | ID_blez | ID_bgtz | ID_bltz | ID_bgez | ID_sll | ID_sllv;
    assign MemWrite = ID_sb | ID_sh | ID_sw;
    assign PCSrc[0] = (ID_blez & c_adventure) | (ID_bgtz & c_adventure)| (ID_bltz
& c_adventure)| (ID_bgez & c_adventure)| ID_jal | ID_jalr | (ID_beq &
c_adventure) | (ID_bne & ~c_adventure);
    assign PCSrc[1] = ID_j | ID_jal;
    assign PCSrc[2] = ID_jalr | ID_jr;
    assign MemtoReg = ID_lb | ID_lbu | ID_lh | ID_lhu | ID_lw;
    assign load_option[0] = ID_lb | ID_lbu | ID_lh | ID_lhu;
    assign load_option[1] = ID_lh | ID_lhu;
    assign load_option[2] = ID_lh | ID_lb;
    assign save_option[0] = ID_sb;
    assign save_option[1] = ID_sh;
    assign usigned = ID_lbu | ID_lhu | ID_addu | ID_subu | ID_addiu | ID_sltiu |
ID_sltu;
    assign B_code = Inst[16];

    assign md_control[0] = ID_multu | ID_divu | ID_mtlo | ID_mflo;
    assign md_control[1] = ID_div | ID_divu | ID_mtlo | ID_mthi;
    assign md_control[2] = ID_multu | ID_mult | ID_div | ID_divu;

    assign md_start = ID_multu | ID_mult | ID_div | ID_divu | ID_mthi | ID_mtlo |
ID_mfhi | ID_mflo;
    assign is_md = ID_multu | ID_divu | ID_div | ID_divu;
    assign block = (busy | EX_md | next_inst) & md_start;
endmodule
```

RegisterFile

寄存器堆包含组合逻辑的读寄存器和时序逻辑的写寄存器。为实现在同一周期中先写后读的逻辑，我将写寄存器的信号设为时钟下降沿，对于不同PCSrc和RegWrite信号进行不同处理。

```
```

```
module RegisterFile(ReadReg1, ReadReg2, rd, WriteData, WriteReg, RegWrite, clock,
reset, ReadData1, ReadData2, PCSrc, WB_PC, PC);
    input [4:0] ReadReg1, rd;
    input [4:0] ReadReg2;
    input [31:0] WriteData, PC;
    input [4:0] WriteReg;
    input RegWrite;
    input clock;
    input reset;
    input [2:0] PCSrc;
    input [31:0] WB_PC;
    output [31:0] ReadData1;
    output [31:0] ReadData2;


    reg [31:0] regFile[0:31];
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            regFile[i] <= 0;
    end

    assign ReadData1 = regFile[ReadReg1];
    assign ReadData2 = regFile[ReadReg2];

    always @(negedge clock) begin
        if (PCSrc == 3'b101 && RegWrite) begin
            $display("@%h: $%d <= %h", PC, WriteReg, WB_PC + 8);
            if (WriteReg) begin
                regFile[WriteReg] = WB_PC + 8;
            end
        end else if (PCSrc == 3'b101) begin
            $display("@%h: $%d <= %h", PC, rd, WB_PC + 8);
            regFile[rd] = WB_PC + 8;
        end else if (PCSrc == 3'b011) begin
            $display("@%h: $31 <= %h", PC, WB_PC + 8);
            regFile[31] = WB_PC + 8;
        end else if (RegWrite) begin
            $display("@%h: $%d <= %h", PC, WriteReg, WriteData);
            if (WriteReg) begin
                regFile[WriteReg] = WriteData;
            end
        end
    end
endmodule
```

if_c_adventure

此模块判断是否会出现控制冒险，根据ALUOperation中的值确定跳转指令类型，对于两个操作数进行比较后将是否跳转的结果存入c_adventure中。

```
module if_c_adventure(
    A, B, Op, usigned, c_adventure
);
    input [31:0] A, B;
```

```
    input [3:0] Op;
    input usigned;
    output c_adventure;

    wire less_res, less_v_res;
    wire unsigned_less_res, unsigned_less_v_res;
    wire less_equal_res;
    wire greater_equal_res;
    wire greater_res;
    wire eq_res;

    assign less_v_res = $signed(A) < $signed(B);
    assign unsigned_less_v_res = A < B ;
    assign less_res = $signed(A) < 0 ;
    assign less_equal_res = $signed(A) <= 0 ;
    assign greater_equal_res = $signed(A) >= 0 ;
    assign greater_res = $signed(A) > 0 ;
    assign eq_res = A == B;

    assign c_adventure = (~Op[3] & ~Op[2] & ~Op[1]) ? eq_res : (Op[2] == 0 &
Op[1] == 0 & Op[0] == 1) ? (usigned == 1 ? unsigned_less_v_res : less_v_res) :
(Op[2] == 1 ? (Op[0] == 1 ? greater_equal_res : less_res) : (Op[0] == 1 ?
greater_res : less_equal_res));

endmodule
```

FU

此模块用于处理旁路转发信号，流水线竞争处理设计如下：

旁路设计

我一共设计了两个旁路机制，一个位于译码（ID）阶段，另一个位于执行（EX）阶段。

在执行阶段（EX），我设置了一个旁路选择器，它可以提前从EX/MEM或MEM/WB阶段获取数据。这样可以确保在执行阶段所需的数据提前得到，并能被使用，避免了大部分数据冒险问题。

而在译码阶段（ID），我设置了另一个旁路选择器。这个选择器的存在是因为对于跳转类指令例如beq在译码阶段就需要决定下一条进入流水线的指令。在这个阶段，同样需要考虑尚未写回带来的数据冒险问题。不过，不同于在执行（EX）阶段，这里仅需从MEM/WB寄存器获取数据。

nop指令插入位置

在插入nop指令时，我们需要冻结PC，冻结IF/ID并清除ID/EX。同时在以下四种情况需要插入nop指令：

1. lw指令的数据冒险。lw写入的寄存器在下一条指令中被当做rs或rt使用了，因为lw在访存（MEM）阶段才能返回结果，所以旁路无法解决该数据冒险，只能插nop指令。
2. 跳转指令数据冒险：在跳转指令中，我们需要在译码（ID）阶段提前计算跳转条件，因此若发生数据冒险无法从EX/MEM获得数据，因此需要插入nop指令。
3. lw+跳转指令数据冒险：因为lw在访存（MEM）阶段才能返回结果而跳转指令，而跳转指令在译码（ID）阶段提前计算跳转条件，因此需要插入两条nop指令。
4. lw与跳转指令相隔一条指令：与上述情况类似，需要插入一条nop指令。

控制冒险

我将跳转语句提前到ID阶段计算跳转条件是否成立，以减少nop数，使得分支预测失败代价降低。

```
module FU(
    EX_RegWrite, EX_WriteReg, EX_MemtoReg, ME_RegWrite, ME_WriteReg, ME_MemtoReg,
EX_PCSrc, ME_PCSrc,
    ID_rs, ID_rt, ID_FA, ID_FB, ID_Op, ID_func, c_adventure, stall, stall2
);
    input [4:0] EX_WriteReg, ME_WriteReg, ID_rs, ID_rt;
    input EX_RegWrite, ME_RegWrite, EX_MemtoReg, ME_MemtoReg, c_adventure;
    input [5:0] ID_Op, ID_func;
    input [2:0] EX_PCSrc, ME_PCSrc;
    output reg [1:0] ID_FA, ID_FB;
    output stall, stall2;

    wire R_type = ~ID_Op[5] & ~ID_Op[4] & ~ID_Op[3] & ~ID_Op[2] & ~ID_Op[1] &
~ID_Op[0];
    wire ID_beq = ~ID_Op[5] & ~ID_Op[4] & ~ID_Op[3] & ID_Op[2] & ~ID_Op[1] &
~ID_Op[0];
    wire ID_bne = ~ID_Op[5] & ~ID_Op[4] & ~ID_Op[3] & ID_Op[2] & ~ID_Op[1] &
ID_Op[0];
    wire ID_jalr = R_type & ~ID_func[5] & ~ID_func[4] & ID_func[3] & ~ID_func[2]
& ~ID_func[1] & ID_func[0];
    wire ID_lui = ~ID_Op[5] & ~ID_Op[4] & ID_Op[3] & ID_Op[2] & ID_Op[1] &
ID_Op[0];
    wire ID_jr = R_type & ~ID_func[5] & ~ID_func[4] & ID_func[3] & ~ID_func[2] &
~ID_func[1] & ~ID_func[0];

    always @ (EX_WriteReg, ME_WriteReg, EX_RegWrite, ME_RegWrite, ID_rs, ID_rt)
begin
        ID_FA = 2'b00;
        if ((((ID_rs == EX_WriteReg) & (EX_WriteReg != 0) & (EX_RegWrite == 1)) |
((ID_rs == 5'b11111) & (EX_PCSrc == 3'b011))) & ~(ID_rs == 5'b11111 & ID_jr))
begin
            ID_FA = 2'b01;
        end else if ((((ID_rs == ME_WriteReg) & (ME_WriteReg != 0) & (ME_RegWrite
== 1)) | ((ID_rs == 5'b11111) & (ME_PCSrc == 3'b011))) & ~(ID_rs == 5'b11111 &
ID_jr)) begin
            ID_FA = 2'b10;
        end
    end

    always @ (EX_WriteReg, ME_WriteReg, EX_RegWrite, ME_RegWrite, ID_rs, ID_rt)
begin
        ID_FB = 2'b00;
        if (((ID_rt == EX_WriteReg) & (EX_WriteReg != 0) & (EX_RegWrite == 1)) |
((ID_rt == 5'b11111) & (EX_PCSrc == 3'b011))) begin
            ID_FB = 2'b01;
        end else if (((ID_rt == ME_WriteReg) & (ME_WriteReg != 0) & (ME_RegWrite
== 1)) | ((ID_rt == 5'b11111) & (ME_PCSrc == 3'b011))) begin
            ID_FB = 2'b10;
        end
    end
```

```
    assign stall = stall2 | (((ID_rs == EX_WriteReg) | (ID_rt == EX_WriteReg)) &
(EX_MemtoReg == 1) & (EX_WriteReg != 0) & (EX_RegWrite == 1) & ~ID_lui) |
((ID_beq | ID_bne | ID_jalr) & ((ID_rs == EX_WriteReg) | (ID_rt == EX_WriteReg))
& (EX_WriteReg != 0) & (EX_RegWrite == 1)) | (((ID_rs == ME_WriteReg) | (ID_rt ==
ME_WriteReg)) & (ME_MemtoReg == 1) & (ME_WriteReg != 0) & (ME_RegWrite == 1) &
ID_beq);
    assign stall2 = ((ID_rs == EX_WriteReg) | (ID_rt == EX_WriteReg)) &
(EX_MemtoReg == 1) & (EX_WriteReg != 0) & (EX_RegWrite == 1) & ID_beq;

endmodule
```

REG_ID_EX

该模块处理控制信号从ID到EX的传递，当处于stall或block状态时，流水线阻塞，需要清空EX阶段所有控制信号。

```
module REG_ID_EX(ID_B_code, ID_sa, ID_RegDst, ID_RegWrite, ID_ALUXSrc,
ID_ALUYSrc, ID_ALUControl, ID_MemWrite, ID_MemtoReg, ID_WriteReg, ID_usigned,
ID_Qa, ID_Qb, ID_Ext32, ID_FA, ID_FB, ID_load_option, ID_save_option, ID_PC,
ID_PCSrc, ID_md_control, ID_start, ID_md, clock, reset,
                EX_B_code, EX_sa, EX_RegDst, EX_RegWrite, EX_ALUXSrc,
EX_ALUYSrc, EX_ALUControl, EX_MemWrite, EX_MemtoReg, EX_WriteReg, EX_usigned,
EX_Qa, EX_Qb, EX_Ext32, EX_FA, EX_FB, EX_load_option, EX_save_option, EX_PC,
EX_PCSrc, EX_md_control, EX_start, EX_md, stall, stall2, block);

    input [31:0] ID_Qa, ID_Qb, ID_Ext32, ID_sa, ID_PC;
    input [4:0] ID_WriteReg;
    input [1:0] ID_FA, ID_FB;
    input [3:0] ID_ALUControl;
    input [2:0] ID_load_option, ID_PCSrc, ID_md_control;
    input [1:0] ID_save_option;
    input clock, reset, stall, stall2, block;
    input ID_RegDst, ID_RegWrite, ID_ALUYSrc, ID_MemWrite, ID_MemtoReg,
ID_ALUXSrc, ID_usigned, ID_B_code, ID_start, ID_md;

    wire enable;
    assign enable = ~stall & ~stall2 & ~block;

    output reg [31:0] EX_Qa, EX_Qb, EX_Ext32, EX_sa, EX_PC;
    output reg [1:0] EX_FA, EX_FB;
    output reg [3:0] EX_ALUControl;
    output reg [4:0] EX_WriteReg;
    output reg [2:0] EX_load_option, EX_PCSrc, EX_md_control;
    output reg [1:0] EX_save_option;
    output reg EX_RegDst, EX_RegWrite, EX_ALUYSrc, EX_MemWrite, EX_MemtoReg,
EX_ALUXSrc, EX_usigned, EX_B_code, EX_start, EX_md;

    initial begin
        EX_sa = 0;
        EX_ALUControl = 0;
        EX_ALUXSrc = 0;
        EX_ALUYSrc = 0;
        EX_Ext32 = 0;
        EX_FA = 0;
        EX_FB = 0;
```

```
        EX_MemtoReg = 0;
        EX_MemWrite = 0;
        EX_Qa = 0;
        EX_Qb = 0;
        EX_RegDst = 0;
        EX_RegWrite = 0;
        EX_WriteReg = 0;
        EX_usigned = 0;
        EX_B_code = 0;
        EX_load_option = 0;
        EX_save_option = 0;
        EX_PC = 0;
        EX_PCSrc = 0;
        EX_start = 0;
        EX_md_control = 0;
        EX_md = 0;
    end

    always @(posedge clock) begin
        if (enable == 0) begin
            EX_sa = 0;
            EX_ALUControl = 0;
            EX_ALUXSrc = 0;
            EX_ALUYSrc = 0;
            EX_Ext32 = 0;
            EX_FA = 0;
            EX_FB = 0;
            EX_MemtoReg = 0;
            EX_MemWrite = 0;
            EX_Qa = 0;
            EX_Qb = 0;
            EX_RegDst = 0;
            EX_RegWrite = 0;
            EX_WriteReg = 0;
            EX_usigned = 0;
            EX_B_code = 0;
            EX_load_option = 0;
            EX_save_option = 0;
            EX_PC = 0;
            EX_PCSrc = 0;
            EX_start = 0;
            EX_md_control = 0;
            EX_md = 0;
        end else begin
            EX_sa = ID_sa;
            EX_ALUControl = ID_ALUControl;
            EX_ALUXSrc = ID_ALUXSrc;
            EX_ALUYSrc = ID_ALUYSrc;
            EX_Ext32 = ID_Ext32;
            EX_FA = ID_FA;
            EX_FB = ID_FB;
            EX_MemtoReg = ID_MemtoReg;
            EX_MemWrite = ID_MemWrite;
            EX_Qa = ID_Qa;
            EX_Qb = ID_Qb;
            EX_RegDst = ID_RegDst;
```

```
            EX_RegWrite = ID_RegWrite;
            EX_WriteReg = ID_WriteReg;
            EX_usigned = ID_usigned;
            EX_B_code = ID_B_code;
            EX_load_option = ID_load_option;
            EX_save_option = ID_save_option;
            EX_PC = ID_PC;
            EX_PCSrc = ID_PCSrc;
            EX_start = ID_start;
            EX_md_control = ID_md_control;
            EX_md = ID_md;
        end
    end

    always @(posedge reset) begin
        EX_sa = 0;
        EX_ALUControl = 0;
        EX_ALUXSrc = 0;
        EX_ALUYSrc = 0;
        EX_Ext32 = 0;
        EX_FA = 0;
        EX_FB = 0;
        EX_MemtoReg = 0;
        EX_MemWrite = 0;
        EX_Qa = 0;
        EX_Qb = 0;
        EX_RegDst = 0;
        EX_RegWrite = 0;
        EX_WriteReg = 0;
        EX_usigned = 0;
        EX_B_code = 0;
        EX_load_option = 0;
        EX_save_option = 0;
        EX_PC = 0;
        EX_PCSrc = 0;
        EX_start = 0;
        EX_md_control = 0;
        EX_md = 0;
    end
endmodule
```

EX阶段

顶层模块

```
// 将md_control转化为md_operation_t类型的变量
assign operation = (EX_md_control == 3'b000) ? MDU_READ_HI :
                   (EX_md_control == 3'b001) ? MDU_READ_LO :
                   (EX_md_control == 3'b010) ? MDU_WRITE_HI :
                   (EX_md_control == 3'b011) ? MDU_WRITE_LO :
                   (EX_md_control == 3'b100) ? MDU_START_SIGNED_MUL :
                   (EX_md_control == 3'b101) ? MDU_START_UNSIGNED_MUL :
                   (EX_md_control == 3'b110) ? MDU_START_SIGNED_DIV :
MDU_START_UNSIGNED_DIV; // change to mdu_operation_t
```

```
// 转发与选择
MUX3X32 mux3x32_ex_X(EX_Qa, ME_Alu_Result, WriteInput, EX_FA, EX_NUM_X);
MUX2X32 choose_alu_x(EX_NUM_X, EX_sa, EX_ALUXSrc, Alu_X);

MUX3X32 mux3x32_ex_Y(EX_Qb, ME_Alu_Result, WriteInput, EX_FB, EX_NUM_Y);
MUX2X32 choose_alu_y(EX_Ext32, EX_NUM_Y, EX_ALUYSrc, Alu_Y);

// 调用MDU与ALU
MultiplicationDivisionUnit MDU(reset, clock, EX_NUM_X, EX_NUM_Y, operation,
EX_start, busy, EX_md_Result);
ALU ALU(EX_PC, EX_PCSrc, Alu_X, Alu_Y, EX_ALUControl, EX_usigned, EX_Alu_Result,
zero, over);

// 将控制信号传递给ME阶段
REG_EX_MEM REG_EX_MEM(EX_RegWrite, EX_RegDst, EX_MemWrite, EX_MemtoReg,
EX_WriteReg, EX_NUM_Y, EX_Alu_Result, EX_load_option, EX_save_option, EX_PC,
EX_PCSrc, EX_md_Result, EX_start, clock, reset, EX_md_control[2],
                      ME_RegWrite, ME_RegDst, ME_MemWrite, ME_MemtoReg,
ME_WriteReg, ME_NUM_Y, ME_Alu_Result, ME_load_option, ME_save_option, ME_PC,
ME_PCSrc);
```

ALU

我利用组合逻辑设计ALU模块，设计四个子模块Adder，Shifter，BitOp，Leg对应四类运算加减法，移位操作，位运算，比较操作。之后再利用ALUOperation和usigned信号来选择四种计算的结果。

```
module Adder(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
    );
    // 同步进位加法器
     ....
endmodule

module Shift(input [31:0] B, input[3:0] Op, input [31:0] A, input usigned,output
[31:0] res);
    wire [31:0] left_shift;
    wire [31:0] right_shift;
    wire [31:0] aright_shift;

    assign left_shift = B << A[4:0];
    assign right_shift = B >> A[4:0];
    assign aright_shift = $signed(B) >>> A[4:0];

    assign res = (Op[0] == 1)? left_shift : (Op[1] == 1? aright_shift :
right_shift);
endmodule

module BitOp(input[31:0] A, input[31:0] B, input[3:0] Op, output [31:0] res);
    wire [31:0] and_res;
    wire [31:0] or_res;
    wire [31:0] xor_res;
    wire [31:0] nor_res;
```

```
    assign and_res = A & B;
    assign or_res = A | B;
    assign xor_res = A ^ B;
    assign nor_res = ~or_res;

    assign res = Op[2] == 0? (Op[0] == 0? and_res : or_res) : (Op[0] == 0 ?
xor_res : nor_res);
endmodule

module Leg(input[31:0] A, input[31:0] B, input[3:0] Op ,input usigned,output
[31:0] res);
    wire less_res, less_equal_res;
    wire greater_res, greater_equal_res;
    wire less_v_res, unsigned_less_v_res, unsigned_less_res;
    wire eq_res;

    assign less_v_res = $signed(A) < $signed(B);
    assign unsigned_less_v_res = A < B;
    assign less_res = $signed(A) < 0;
    assign less_equal_res = $signed(A) <= 0;
    assign greater_equal_res = $signed(A) >= 0;
    assign greater_res = $signed(A) > 0;
    assign eq_res = A == B;

    assign res = (~Op[3]&&~Op[2]&&~Op[1])?eq_res:(Op[2]==0 && Op[1]==0 &&
Op[0]==1) ? (usigned==1 ? unsigned_less_v_res: less_v_res):(Op[2]==1 ? (Op[0]==1
? greater_equal_res : less_res):(Op[0]==1 ? greater_res : less_equal_res));
endmodule

 module ALU(PC, PCSrc, A, B, Op, usigned, C, zero, over);
    input [31:0] PC;
    input [31:0] A;
    input [31:0] B;
    input [3:0] Op;
    input [2:0] PCSrc;
    input usigned;
    output [31:0] C;
    output zero;
    output over;

    wire [31:0] shift_res, aoxn_res, sum_res, leg_res, PC_res;
    wire [31:0] neg_B, b_input;
    wire cout;
    Adder neg(~B, 32'b1, neg_B);
    assign b_input = Op[0]? neg_B : B;

    Shift shift(B, Op, A, usigned, shift_res);
    BitOp bitOp(A, B, Op, aoxn_res);
    Adder adder(A, b_input, sum_res);
    Adder adder_pc(PC, 32'b1000, PC_res);
    Leg leg(A, B, Op, usigned, leg_res);

    assign over = (~Op[3] & ~Op[2] & ~Op[1]) ? ((usigned) & ((A[31] ==
b_input[31]) & (A[31] != sum_res[31]))) : 0;
```

```
    assign C = (PCSrc == 3'b011)? PC_res : (Op[3]? ((~Op[2] & ~Op[1] & Op[0])?
leg_res : shift_res) : ((~Op[2] & ~Op[1])? sum_res : (Op[2] & Op[1])? (B << 16) :
aoxn_res));

    assign zero = (C == 0) ? 1 : 0;
endmodule
```

REG_EX_ME

此模块的传递不需要冻结或清空，直接传递即可。

```
module REG_EX_MEM(
    EX_RegWrite, EX_RegDst, EX_MemWrite, EX_MemtoReg, EX_WriteReg, EX_Qb,
EX_Alu_Result, EX_load_option, EX_save_option, EX_PC, EX_PCSrc, EX_md_Result,
EX_start, clock, reset, busy,
    ME_RegWrite, ME_RegDst, ME_MemWrite, ME_MemtoReg, ME_WriteReg, ME_Qb,
ME_Alu_Result, ME_load_option, ME_save_option, ME_PC, ME_PCSrc
);

    input [31:0] EX_Alu_Result, EX_md_Result, EX_Qb, EX_PC;
    input [4:0] EX_WriteReg;
    input [2:0] EX_load_option, EX_PCSrc;
    input [1:0] EX_save_option;
    input clock, reset;
    input EX_RegWrite, EX_RegDst, EX_MemWrite, EX_MemtoReg, EX_start, busy;

    output reg [31:0] ME_Alu_Result, ME_Qb, ME_PC;
    output reg [4:0] ME_WriteReg;
    output reg [2:0] ME_load_option, ME_PCSrc;
    output reg [1:0] ME_save_option;
    output reg ME_RegWrite, ME_RegDst, ME_MemWrite, ME_MemtoReg;

    initial begin
        ME_Alu_Result = 0;
        ME_MemtoReg = 0;
        ME_MemWrite = 0;
        ME_Qb = 0;
        ME_RegDst = 0;
        ME_RegWrite = 0;
        ME_WriteReg = 0;
        ME_load_option = 0;
        ME_save_option = 0;
        ME_PC = 0;
        ME_PCSrc = 0;
    end

    always @(posedge clock) begin
        ME_Alu_Result = (EX_start & ~busy)? EX_md_Result : EX_Alu_Result;
        ME_MemtoReg = EX_MemtoReg;
        ME_MemWrite = EX_MemWrite;
        ME_Qb = EX_Qb;
        ME_RegDst = EX_RegDst;
        ME_RegWrite = EX_RegWrite;
        ME_WriteReg = EX_WriteReg;
        ME_load_option = EX_load_option;
```

```
        ME_save_option = EX_save_option;
        ME_PC = EX_PC;
        ME_PCSrc = EX_PCSrc;
    end

    always @(posedge reset) begin
        ME_Alu_Result = 0;
        ME_MemtoReg = 0;
        ME_MemWrite = 0;
        ME_Qb = 0;
        ME_RegDst = 0;
        ME_RegWrite = 0;
        ME_WriteReg = 0;
        ME_load_option = 0;
        ME_save_option = 0;
        ME_PC = 0;
        ME_PCSrc = 0;
    end

endmodule
```

ME阶段

顶层模块

```
// 利用save_option中的信号对sw操作分类
Save2BE save_to_BE(ME_save_option, BE);

// 从上层传下的结果和PC+8进行选择
MUX2X32 mux2x32_2(ME_Result, ME_PC + 8, ME_PCSrc[0], ME_Alu_Result);

// 访存模块，组合逻辑在中读出内存到ME_Dout中同时写入内存
DataMemory data_memory(ME_Alu_Result, BE, ME_NUM_Y, ME_Dout, ME_MemWrite, clock,
ME_PC);

// 将控制信号传入WB阶段
REG_MEM_WB REG_MEM_WB(ME_RegWrite, ME_MemtoReg, ME_Alu_Result, ME_Dout,
ME_WriteReg, ME_load_option, ME_PC, ME_PCSrc, clock, reset,
                      WB_RegWrite, WB_MemtoReg, WB_Alu_Result, WB_Dout,
WB_WriteReg, WB_load_option, WB_PC, WB_PCSrc);
```

DataMemory

在此模块中通过不同的访存方式来写入内存不同的位置。

```
module DataMemory(
    Addr, BE, Din, Dout, MemWrite, clock, PC
);
    input [31:0] Din, PC;
    input [31:0] Addr;
    input [3:0] BE;
    input clock, MemWrite;
    output [31:0] Dout;
    reg [31:0] Ram[2047:0];
```

```
    assign Dout = Ram[Addr[12:2]];
    wire [7:0] Din_b;
    wire [15:0] Din_h;
    assign Din_b = Din[7:0];
    assign Din_h = Din[15:0];

    always @(posedge clock) begin
        if (MemWrite && BE[3] == 1) begin
            $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc, Din);
            Ram[Addr[12:2]] <= Din;
        end else if (MemWrite && BE[1] == 1) begin
            if (Addr[1] == 1) begin
                Ram[Addr[12:2]][31:16] = Din_h;
                $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
            end else begin
                Ram[Addr[12:2]][15:0] = Din_h;
                $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
            end
        end else if (MemWrite && BE[0] == 1) begin
            if (Addr[1] == 1) begin
                if (Addr[0] == 1) begin
                    Ram[Addr[12:2]][31:24] = Din_b;
                    $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
                end else begin
                    Ram[Addr[12:2]][23:16] = Din_b;
                    $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
                end
            end else begin
                if (Addr[0] == 1) begin
                    Ram[Addr[12:2]][15:8] = Din_b;
                    $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
                end else begin
                    Ram[Addr[12:2]][7:0] = Din_b;
                    $display("@%h: *%h <= %h", PC, Addr & 32'hfffffffc,
Ram[Addr[12:2]]);
                end
            end
        end
    end

    integer i;
    initial begin
        for (i = 0; i < 2048; i = i + 1)
            Ram[i] = 0;
    end
endmodule
```

REG_ME_WB

同REG_EX_ME，将ME控制信号传递到WB阶段，不需要阻塞。

```
module REG_MEM_WB(
    ME_RegWrite, ME_MemtoReg, ME_Alu_Result, ME_Dout, ME_WriteReg,
ME_load_option, ME_PC, ME_PCSrc, clock, reset,
    WB_RegWrite, WB_MemtoReg, WB_Alu_Result, WB_Dout, WB_WriteReg,
WB_load_option, WB_PC, WB_PCSrc
);

    input [31:0] ME_Alu_Result, ME_Dout, ME_PC;
    input [4:0] ME_WriteReg;
    input [2:0] ME_load_option, ME_PCSrc;
    input clock, reset, ME_RegWrite, ME_MemtoReg;

    output reg [31:0] WB_Alu_Result, WB_Dout, WB_PC;
    output reg [4:0] WB_WriteReg;
    output reg [2:0] WB_load_option, WB_PCSrc;
    output reg WB_RegWrite, WB_MemtoReg;

    initial begin
        WB_RegWrite = 0;
        WB_MemtoReg = 0;
        WB_Alu_Result = 0;
        WB_Dout = 0;
        WB_WriteReg = 0;
        WB_load_option = 0;
        WB_PC = 0;
        WB_PCSrc = 0;
    end

    always @(posedge clock) begin
        WB_RegWrite = ME_RegWrite;
        WB_MemtoReg = ME_MemtoReg;
        WB_Alu_Result = ME_Alu_Result;
        WB_Dout = ME_Dout;
        WB_WriteReg = ME_WriteReg;
        WB_load_option = ME_load_option;
        WB_PC = ME_PC;
        WB_PCSrc = ME_PCSrc;
    end

    always @(posedge reset) begin
        WB_RegWrite = 0;
        WB_MemtoReg = 0;
        WB_Alu_Result = 0;
        WB_Dout = 0;
        WB_WriteReg = 0;
        WB_load_option = 0;
        WB_PC = 0;
        WB_PCSrc = 0;
    end
endmodule
```

WB阶段

顶层模块

```
// 通过读入信号来处理写入寄存器值
data_ext_load data_ext_load(WB_Dout, WB_Alu_Result, WB_load_option, WB_ext_Dout);
// 选择写入寄存器的值
MUX2X32 mux2x322(WB_Alu_Result, WB_ext_Dout, WB_MemtoReg, WriteInput);
```

data_ext_load

我利用load_option控制信号来区分读入字节类型，同时区分符号扩展和零扩展。

```
module data_ext_load (
    Dout, Addr, load_option, ext_Dout
);
    input [31:0] Dout;
    input [31:0] Addr;
    input [2:0] load_option;
    output [31:0] ext_Dout;

    wire [31:0] lb, lbu, lh, lhu, lw;
    wire [23:0] e1 = {24{Dout[7]}};
    wire [23:0] e2 = {24{Dout[15]}};
    wire [23:0] e3 = {24{Dout[23]}};
    wire [23:0] e4 = {24{Dout[31]}};
    wire [15:0] e5 = {16{Dout[15]}};
    wire [15:0] e6 = {16{Dout[31]}};
    parameter z1 = 24'b0;
    parameter z2 = 16'b0;
    assign lb = (Addr[1] == 1)? (Addr[0] == 1 ? {e4, Dout[31:24]} : {e3,
Dout[23:16]}) : (Addr[0] == 1 ? {e2, Dout[15:8]} : {e1, Dout[7:0]});
    assign lh = (Addr[1] == 1)? {e6, Dout[31:16]} : {e5, Dout[15:0]};
    assign lbu = (Addr[1] == 1)? (Addr[0] == 1 ? {z1, Dout[31:24]} : {z1,
Dout[23:16]}) : (Addr[0] == 1 ? {z1, Dout[15:8]} : {z1, Dout[7:0]});
    assign lhu = (Addr[1] == 1)? {z2, Dout[31:16]} : {z2, Dout[15:0]};
    assign lw = Dout;

    MUX5X32 mux5x32(lb, lbu, lh, lhu, lw, load_option, ext_Dout);
endmodule
```

正确性测试

以上流水线CPU通过了以下随机性指令测试与对应C代码测试。

测试仓库：

https://github.com/ceerRep/mips-asm-test

https://github.com/ceerRep/pipeline-tester-py

测试截图

```
user:
@000030e4: $ 2 <= 0000000a
mars:
1040
C:\Users\HBW\Desktop\mips-asm-test\pointer-chasing.asm PASSED
------------------------------------------------
```

```
------------------------------------------------
Running C:\Users\HBW\Desktop\mips-asm-test\random-asms\test_99.asm
Mars result: 70 lines
User result: 71698 lines
C:\Users\HBW\Desktop\mips-asm-test\random-asms\test_99.asm PASSED
------------------------------------------------
```

```
round 198/200
Mars result: 86 lines
User result: 577 lines
round 199/200
Mars result: 445 lines
User result: 680 lines
```