# RISC_V_32I Lab2 Report

> 实验目标：使用verilog HDL实现RISC_V_32I流水线CPU
>
> 实验环境与工具：
>
> 操作系统：Windows10(MSYS_NT-10.0 DESKTOP-E4RKA7V 2.11.2(0.329/5/3) 2018-11-10 14:38 x86_64 Msys)
>
> 综合工具：Vivado 2018.3
>
> 姓名：张劲暾
>
> 学号：PB16111485

## 核心代码段设计

### 控制单元：ControlUnit

设计思路：

1. 解析指令确定指令信号
2. 根据指令信号为每个输出信号设计产生逻辑

具体设计如下：

```verilog
`include "Parameters.v"
module ControlUnit(
    input wire [6:0] Op,
    input wire [2:0] Fn3,
    input wire [6:0] Fn7,
    output wire JalD,
    output wire JalrD,
    output reg [2:0] RegWriteD,
    output wire MemToRegD,
    output reg [3:0] MemWriteD,
    output wire LoadNpcD,
    output reg [1:0] RegReadD,
    output reg [2:0] BranchTypeD,
    output reg [3:0] AluContrlD,
    output wire [1:0] AluSrc2D,
    output wire AluSrc1D,
    output reg [2:0] ImmType
    );
//==========================================================================================
// 基本思路：Op + Fn3 + Fn7 确定一条指令
//==========================================================================================
// OpCode
// load upper immediate
localparam LUI_OP   = 7'b011_0111;
// add upper immediate to pc
localparam AUIPC_OP = 7'b001_0111;
```

```verilog
// 立即数算术逻辑运算的操作码
localparam ALUI_OP  = 7'b001_0011;
// 寄存器算术逻辑运算的操作码
localparam ALUR_OP  = 7'b011_0011;
// 跳转并连接操作码
localparam JAL_OP   = 7'b110_1111;
// 寄存器跳转并连接操作码
localparam JALR_OP  = 7'b110_0111;
// 分支指令操作码
localparam BR_OP    = 7'b110_0011;
// Load指令操作码
localparam LOAD_OP  = 7'b000_0011;
// Store指令操作码
localparam STORE_OP = 7'b010_0011;
//===========================================================================================
// 具体指令信号
// 移位指令
wire SLLI, SRLI, SRAI, SLL, SRL, SRA;
assign SLLI = (Op == ALUI_OP ) && (Fn3 == 3'b001);
assign SRLI = (Op == ALUI_OP ) && (Fn3 == 3'b101) && (Fn7 == 7'b000_0000);
assign SRAI = (Op == ALUI_OP ) && (Fn3 == 3'b101) && (Fn7 == 7'b010_0000);
assign SLL  = (Op == ALUR_OP ) && (Fn3 == 3'b001);
assign SRL  = (Op == ALUR_OP ) && (Fn3 == 3'b101) && (Fn7 == 7'b000_0000);
assign SRA  = (Op == ALUR_OP ) && (Fn3 == 3'b101) && (Fn7 == 7'b010_0000);
// 计算指令
wire ADD, SUB, ADDI;
assign ADD  = (Op == ALUR_OP ) && (Fn3 == 3'b000) && (Fn7 == 7'b000_0000);
assign SUB  = (Op == ALUR_OP ) && (Fn3 == 3'b000) && (Fn7 == 7'b010_0000);
assign ADDI = (Op == ALUI_OP ) && (Fn3 == 3'b000);
// 比较指令
wire SLT, SLTU, SLTI, SLTIU;
assign SLT  = (Op == ALUR_OP ) && (Fn3 == 3'b010);
assign SLTU = (Op == ALUR_OP ) && (Fn3 == 3'b011);
assign SLTI = (Op == ALUI_OP ) && (Fn3 == 3'b010);
assign SLTIU= (Op == ALUI_OP ) && (Fn3 == 3'b011);
// 逻辑指令
wire XOR, OR, AND, XORI, ORI, ANDI;
assign XOR  = (Op == ALUR_OP ) && (Fn3 == 3'b100);
assign OR   = (Op == ALUR_OP ) && (Fn3 == 3'b110);
assign AND  = (Op == ALUR_OP ) && (Fn3 == 3'b111);
assign XORI = (Op == ALUI_OP ) && (Fn3 == 3'b100);
assign ORI  = (Op == ALUI_OP ) && (Fn3 == 3'b110);
assign ANDI = (Op == ALUI_OP ) && (Fn3 == 3'b111);
// 立即数指令
wire LUI, AUIPC;
assign LUI  = (Op == LUI_OP  );
assign AUIPC= (Op == AUIPC_OP);
// 跳转与分支指令
wire JAL, JALR, BEQ, BNE, BLT, BLTU, BGE, BGEU;
assign JAL  = (Op == JAL_OP  );
assign JALR = (Op == JALR_OP );
assign BEQ  = (Op == BR_OP   ) && (Fn3 == 3'b000);
assign BNE  = (Op == BR_OP   ) && (Fn3 == 3'b001);
assign BLT  = (Op == BR_OP   ) && (Fn3 == 3'b100);
assign BGE  = (Op == BR_OP   ) && (Fn3 == 3'b101);
assign BLTU = (Op == BR_OP   ) && (Fn3 == 3'b110);
assign BGEU = (Op == BR_OP   ) && (Fn3 == 3'b111);
// Load指令
wire LB, LH, LW, LBU, LHU;
```

```
assign LB   = (Op == LOAD_OP ) && (Fn3 == 3'b000);
assign LH   = (Op == LOAD_OP ) && (Fn3 == 3'b001);
assign LW   = (Op == LOAD_OP ) && (Fn3 == 3'b010);
assign LBU  = (Op == LOAD_OP ) && (Fn3 == 3'b100);
assign LHU  = (Op == LOAD_OP ) && (Fn3 == 3'b101);
// Store指令
wire SB, SH, SW;
assign SB   = (Op == STORE_OP) && (Fn3 == 3'b000);
assign SH   = (Op == STORE_OP) && (Fn3 == 3'b001);
assign SW   = (Op == STORE_OP) && (Fn3 == 3'b010);
//==========================================================================================
// 辅助信号
// 在译码阶段由非load指令产生的寄存器写入标志
wire RegWD_NL = LUI || AUIPC || (Op == ALUR_OP) || (Op == ALUI_OP) || JAL || JALR;
//==========================================================================================
// 各输出信号处理
//------------------- JalD ------------------------
assign JalD = JAL;
//------------------- JalrD ------------------------
assign JalrD= JALR;
//------------------- RegWriteD --------------------
always @ (*)
    begin
      if(RegWD_NL)  RegWriteD <= `LW;
      else if(LB)   RegWriteD <= `LB;
      else if(LH)   RegWriteD <= `LH;
      else if(LW)   RegWriteD <= `LW;
      else if(LBU)  RegWriteD <= `LBU;
      else if(LHU)  RegWriteD <= `LHU;
      else          RegWriteD <= `NOREGWRITE;
    end
//------------------- MemToRegD --------------------
assign  MemToRegD = (Op == LOAD_OP);
//------------------- MemWriteD --------------------
always @ (*)
    begin
        if(SB)      MemWriteD <= 4'b0001;
        else if(SH) MemWriteD <= 4'b0011;
        else if(SW) MemWriteD <= 4'b1111;
        else        MemWriteD <= 4'b0000;
    end
//------------------- LoadNpcD ---------------------
assign LoadNpcD = JAL || JALR;
//------------------- RegReadD ---------------------
always @ (*)
    begin
        RegReadD[0] <= (Op == ALUR_OP) || (Op == BR_OP  ) || (Op == STORE_OP);
        RegReadD[1] <= (Op == ALUI_OP)
                    || (Op == ALUR_OP)
                    || (Op == LOAD_OP )
                    || (Op == STORE_OP)
                    || (Op == BR_OP)
                    || JALR;
    end
//------------------- BranchTypeD ------------------
always @ (*)
    begin
        if(BEQ)         BranchTypeD <= `BEQ;
        else if(BNE)    BranchTypeD <= `BNE;
        else if(BLT)    BranchTypeD <= `BLT;
        else if(BLTU)   BranchTypeD <= `BLTU;
        else if(BGE)    BranchTypeD <= `BGE;
        else if(BGEU)   BranchTypeD <= `BGEU;
        else            BranchTypeD <= `NOBRANCH;
    end
//------------------- AluContrlD -------------------
always @ (*)
    begin
        if      (SLL || SLLI)           AluContrlD <= `SLL;
        else if (SRA || SRAI)           AluContrlD <= `SRA;
        else if (SRL || SRLI)           AluContrlD <= `SRL;
        else if (ADD || ADDI || AUIPC || JALR || Op == LOAD_OP || Op == STORE_OP)
                                        AluContrlD <= `ADD;
        else if (SUB)                   AluContrlD <= `SUB;
        else if (SLT || SLTI)           AluContrlD <= `SLT;
        else if (SLTU||SLTIU)           AluContrlD <= `SLTU;
        else if (XOR || XORI)           AluContrlD <= `XOR;
        else if (OR  || ORI )           AluContrlD <= `OR;
        else if (AND || ANDI)           AluContrlD <= `AND;
        else if (LUI)                   AluContrlD <= `LUI;
        //else                            AluContrlD <= 4'dx;
        else                            AluContrlD <= 4'b1111;
    end
//------------------- AluSrc2D --------------------
// 00 -- 寄存器; 01 -- rs2的5位（移位操作那5位）; 10 -- 立即数
assign AluSrc2D =
    (SLLI || SRAI || SRLI) ? 2'b01 : ( (Op == ALUR_OP || Op == BR_OP) ? 2'b00 : 2'b10 );
//------------------- AluSrc1D --------------------
// 0 -- 寄存器; 1 -- PC值
assign AluSrc1D = AUIPC;
//------------------- ImmType --------------------
always @ (*)
    begin
        if      (Op == ALUR_OP)                             ImmType <= `RTYPE;
        else if (Op == ALUI_OP || Op == LOAD_OP || JALR)    ImmType <= `ITYPE;
        else if (LUI || AUIPC )                             ImmType <= `UTYPE;
        else if (JAL)                                       ImmType <= `JTYPE;
        else if (Op == BR_OP)                               ImmType <= `BTYPE;
        else if (Op == STORE_OP)                            ImmType <= `STYPE;
        else                                                ImmType <= 3'b111;
    end
//-----------------------------------------------
endmodule
```

### 计算单元：ALU

设计思想：根据控制指令操作操作数

具体设计如下：

```verilog
module ALU(
    input wire [31:0] Operand1,
    input wire [31:0] Operand2,
    input wire [3:0] AluContrl,
    output reg [31:0] AluOut
    );
always@(*)
    begin
        case(AluContrl)
            //=====================================================================
            // 逻辑左移
            `SLL:     AluOut <= Operand1 << Operand2[4:0];
            // 逻辑右移
            `SRL:     AluOut <= Operand1 >> Operand2[4:0];
            // 算术右移
            `SRA:     AluOut <= $signed(Operand1) >>> Operand2[4:0];
            //=====================================================================
            // 无符号加法
            `ADD:     AluOut <= Operand1 + Operand2;
            // 无符号减法
            `SUB:     AluOut <= Operand1 - Operand2;
            //=====================================================================
            // 异或
            `XOR:     AluOut <= Operand1 ^ Operand2;
            // 或
            `OR:      AluOut <= Operand1 | Operand2;
            // 与
            `AND:     AluOut <= Operand1 & Operand2;
            //=====================================================================
            // 有符号数比较
            `SLT:     AluOut <= $signed(Operand1) < $signed(Operand2) ? 32'd1 : 32'd0;
            // 无符号数比较
            `SLTU:    AluOut <= $unsigned(Operand1) < $unsigned(Operand2) ? 32'd1 : 32'd0;
            //=====================================================================
            // 立即数加载 Load Upper Immediate，使用U类格式
            `LUI:     AluOut <= {Operand2[31:12],12'd0};
            //=====================================================================
            default:    AluOut <= 32'b0;
        endcase
    end
endmodule
```

### 冲突处理单元：HarzardUnit

设计思想：

1. 有RAW数据相关或写存储器时stall取址和解码阶段
2. 跳转take时冲刷not take的段寄存器

具体设计如下：

```verilog
module HarzardUnit(
    input wire CpuRst, ICacheMiss, DCacheMiss,
    input wire BranchE, JalrE, JalD,
    input wire [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    input wire [1:0] RegReadE,
    input wire MemToRegE,
    input wire [2:0] RegWriteM, RegWriteW,
    //-----------------------------------------------------------------------------------
    output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM, StallW, FlushW,
    output reg [1:0] Forward1E, Forward2E
    );
    //Stall and Flush signals generate
always @ (*)
    begin
        if(CpuRst)                                                              // CPU初始化
            begin
                StallF <= 1'b0; FlushF <= 1'b1;
                StallD <= 1'b0; FlushD <= 1'b1;
                StallE <= 1'b0; FlushE <= 1'b1;
                StallM <= 1'b0; FlushM <= 1'b1;
                StallW <= 1'b0; FlushW <= 1'b1;
            end
        else if (MemToRegE && ((RdE == Rs1D) || (RdE == Rs2D)) && RdE != 5'b0)  // 读写等待
            begin
                StallF <= 1'b1; FlushF <= 1'b0;
                StallD <= 1'b1; FlushD <= 1'b0;
                StallE <= 1'b0; FlushE <= 1'b0;
                StallM <= 1'b0; FlushM <= 1'b0;
                StallW <= 1'b0; FlushW <= 1'b0;
            end
        else if (BranchE || JalrE)                                              // Ex阶段冲刷
            begin
                StallF <= 1'b0; FlushF <= 1'b0;
                StallD <= 1'b0; FlushD <= 1'b1;
                StallE <= 1'b0; FlushE <= 1'b1;
                StallM <= 1'b0; FlushM <= 1'b0;
                StallW <= 1'b0; FlushW <= 1'b0;
            end
        else if (JalD)                                                          // ID阶段冲刷
            begin
                StallF <= 1'b0; FlushF <= 1'b0;
                StallD <= 1'b0; FlushD <= 1'b1;
                StallE <= 1'b0; FlushE <= 1'b0;
                StallM <= 1'b0; FlushM <= 1'b0;
                StallW <= 1'b0; FlushW <= 1'b0;
            end
        else                                                   // 没有冲突，正常执行
            begin
                StallF <= 1'b0; FlushF <= 1'b0;
                StallD <= 1'b0; FlushD <= 1'b0;
                StallE <= 1'b0; FlushE <= 1'b0;
                StallM <= 1'b0; FlushM <= 1'b0;
                StallW <= 1'b0; FlushW <= 1'b0;
            end
    end
//============================================================================================
// 2'b10 是刚从ALU出来的结果，2'b01 是写回的结果，2'b00是直接寄存器出来的结果
//============================================================================================
//Forward Register Source 1
always @ (*)
    begin
```

```
        if      (   (RegReadE[1] == 1'b1)
                 && (RegWriteM != 3'b0)
                 && (RdM != 5'b0)
                 && (Rs1E == RdM)
                )
            Forward1E <= 2'b10;
        else if (   (RegReadE[1] == 1'b1)
                 && (RegWriteW != 3'b0)
                 && (RdW != 5'b0)
                 && (Rs1E == RdW)
                )
            Forward1E <= 2'b01;
        else
            Forward1E <= 2'b00;
    end
//===========================================================================================
//Forward Register Source 2
always @ (*)
    begin
        if      (   (RegReadE[0] == 1'b1)
                 && (RegWriteM != 3'b0)
                 && (RdM != 5'b0)
                 && (Rs2E == RdM)
                )
            Forward2E <= 2'b10;
        else if (   (RegReadE[0] == 1'b1)
                 && (RegWriteW != 3'b0)
                 && (RdW != 5'b0)
                 && (Rs2E == RdW)
                )
            Forward2E <= 2'b01;
        else
            Forward2E <= 2'b00;
    end
//===========================================================================================
endmodule
```

### 分支决策单元：BranchDecisionMaking

设计思想：根据判断类型和操作数判断跳转是否take

具体设计如下：

```verilog
`include "Parameters.v"
module BranchDecisionMaking(
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1,Operand2,
    output reg BranchE
    );
always @ (*)
    begin
        case(BranchTypeE)
            `NOBRANCH:  BranchE <= 1'b0;
            `BEQ:       BranchE <= (Operand1 == Operand2);
            `BNE:       BranchE <= (Operand1 != Operand2);
            `BLT:       BranchE <= ($signed(Operand1) < $signed(Operand2));
            `BLTU:      BranchE <= ($unsigned(Operand1) < $unsigned(Operand2));
            `BGE:       BranchE <= ($signed(Operand1) >= $signed(Operand2));
            `BGEU:      BranchE <= ($unsigned(Operand1) >= $unsigned(Operand2));
            default:    BranchE <= 1'b0;
        endcase
    end
endmodule
```

### 取址决策单元：NPC_Generator

设计思想：EX阶段产生的Branch和Jalr有效信号优先于ID阶段产生的Jal有效信号

具体设计如下：

``` verilog
module NPC_Generator(
    input wire [31:0] PCF,JalrTarget, BranchTarget, JalTarget,
    input wire BranchE,JalD,JalrE,
    output reg [31:0] PC_In
    );
always@(*)
    begin
        if(JalrE)               // 间接跳转指令
            PC_In <= JalrTarget;
        else if(BranchE)        // 分支指令
            PC_In <= BranchTarget;
        else if(JalD)           // 跳转并连接指令
            PC_In <= JalTarget;
        else
            PC_In <= PCF + 4;
    end
endmodule
```

### 数据加载单元：DataExt

设计思想：根据load指令类型和load字节选取产生32位输出结果

具体设计如下：

``` verilog
`include "Parameters.v"
module DataExt(
    input wire [31:0] IN,
    input wire [1:0] LoadedBytesSelect,
    input wire [2:0] RegWriteW,
    output reg [31:0] OUT
    );
always @ (*)
    begin
        case (RegWriteW)
            `NOREGWRITE:    OUT <= 32'b0;
            `LB:
                begin
```

```verilog
                    case(LoadedBytesSelect)
                        2'b00:  OUT <= { {24{IN[ 7]}}, IN[ 7: 0] };
                        2'b01:  OUT <= { {24{IN[15]}}, IN[15: 8] };
                        2'b10:  OUT <= { {24{IN[23]}}, IN[23:16] };
                        2'b11:  OUT <= { {24{IN[31]}}, IN[31:24] };
                        default:OUT <= 32'bx;
                    endcase
                end
            `LH:
                begin
                    casex(LoadedBytesSelect)
                        2'b00:  OUT <= { {16{IN[15]}}, IN[15: 0] };
                        2'b01:  OUT <= { {16{IN[23]}}, IN[23: 8] };
                        2'b10:  OUT <= { {16{IN[31]}}, IN[31:16] };
                        default:OUT <= 32'bx;
                    endcase
                end
            `LW:            OUT <= IN;
            `LBU:
                begin
                    case(LoadedBytesSelect)
                        2'b00:  OUT <= { 24'b0, IN[ 7: 0] };
                        2'b01:  OUT <= { 24'b0, IN[15: 8] };
                        2'b10:  OUT <= { 24'b0, IN[23:16] };
                        2'b11:  OUT <= { 24'b0, IN[31:24] };
                        default:OUT <= 32'bx;
                    endcase
                end
            `LHU:
                begin
                    casex(LoadedBytesSelect)
                        2'b00:  OUT <= { 16'b0, IN[15: 0] };
                        2'b01:  OUT <= { 16'b0, IN[23: 8] };
                        2'b10:  OUT <= { 16'b0, IN[31:16] };
                        default:OUT <= 32'bx;
                    endcase
                end
            default:        OUT <= 32'bx;
        endcase
    end
endmodule
```

### 立即数解析单元：ImmOperandUnit

设计思想：根据立即数类型解析立即数

具体设计如下：

```verilog
 `include "Parameters.v"
module ImmOperandUnit(
    input wire [31:7] In,
    input wire [2:0] Type,
    output reg [31:0] Out
    );
    //
    always@(*)
    begin
        case(Type)
```

~~~
                `ITYPE:     Out <= { {21{In[31]}}, In[30:20] };
                `RTYPE:     Out <= 32'd0;
                `UTYPE:     Out <= { In[31:12], 12'b0 };
                `BTYPE:     Out <= { {20{In[31]}}, In[7], In[30:25], In[11:8], 1'b0 };
                `STYPE:     Out <= { {21{In[31]}}, In[30:25], In[11:7] };
                `JTYPE:     Out <= { {12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0};
                default:    Out <= 32'hxxxxxxxx;
            endcase
        end

    endmodule
```
~~~

### 写回控制单元：WBSegReg

设计思想：默认store类型的指令先将要保存的数据和有效位放在低位，在这里根据低两位地址做一个位移再交给memory

具体设计如下：

~~~
``` verilog
        wire [31:0] RD_raw;
        reg [ 3:0] WE_SHIFT;
        reg [31:0] WD_SHIFT;
        always @ (*)
            begin
                case(WE)
                    4'b0001: WE_SHIFT <= WE << A[1:0];
                    4'b0011: WE_SHIFT <= WE << A[1:0];
                    4'b1111: WE_SHIFT <= WE;
                    default: WE_SHIFT <= 4'b0000;
                endcase
                WD_SHIFT <= WD << (A[1:0] * 8);
            end
        DataRam DataRamInst (
            .clk    ( clk              ),                        //请补全
            .wea    ( WE_SHIFT         ),                        //请补全
            .addra  ( A[31:2]          ),                        //请补全
            .dina   ( WD_SHIFT         ),                        //请补全
            .douta  ( RD_raw           ),
            .web    ( WE2              ),
            .addrb  ( A2[31:2]         ),
            .dinb   ( WD2              ),
            .doutb  ( RD2              )
        );
```
~~~

## 实验结果

标准测试testAll838个测试样例模拟测试通过，现场检查已验收。