verilog单周期cpu设计文档

注意事项

1. 仔细阅读**RTL语言**
2. 搭电路时不要影响到其他指令的实现
3. 仔细读报错信息
4. **`default_nettype none**的使用
5. 数组从低到高定义
6. for(i = 0;i<32;i=i+1)括号内使用**阻塞赋值**

核心思想

本部分内容主要来自于阅读肖利民老师的PPT以及阅读学长往年的博客

核心思想：CPU的功能是控制指令执行，过程是**取指、取数、执行。**

注意一定要仔细读RTL语言！！！

MIPS模型机指令集

模型机指令编码

R 类型格式	OP (31 ~ 26)	Rs (25 ~ 21)	Rt (20 ~ 16)	Rd (15 ~ 11)	Shamt (10 ~ 6)	Funct (5 ~ 0)
add rd, rs, rt	000000	rs	rt	rd	XXXXX	100000
sub rd, rs, rt	000000	rs	rt	rd	XXXXX	100010
and rd, rs, rt	000000	rs	rt	rd	XXXXX	100100
or rd, rs, rt	000000	rs	rt	rd	XXXXX	100101

I 类型格式	OP (31 ~ 26)	Rs (25 ~ 21)	Rt (20 ~ 16)	16 bits immediate or address (15 ~ 0)
lw rt, rs, imm16	100011	rs	rt	imm16
sw rt, rs, imm16	101011	rs	rt	imm16
beq rs, rt, imm16	000100	rs	rt	imm16

J 类型格式	OP (31 ~ 26)	26 bits address
j target	000010	target

- 对于R型指令，处理流程为：

  从IM中读指令——送Controller中译码——从GRF中取数——送ALU运算——送GRF存结果——PC加4

- 对于I型指令，处理流程为：
  从IM中读指令——送Controller中译码——从GRF中取数&在EXT中扩展立即数——（送ALU运算）（从DM中读数）——送GRF存结果|送DM存结果|计算nPC——PC+4|PC变到指定值

- 对于J型指令，处理流程为：
  从IM中读指令——送Controller中译码——从GRF中取数|将立即数扩展到32位——改变NextPC的值

子电路具体实现方法

Controller(控制器)

- 端口：

信号名	方向	描述
op[5:0]	input	对应Instr[31:26]
fc[5:0]	input	对应Instr[5:0]
ALUSrc	output	1时ALU的第二个操作数为立即数，否则为寄存器
ALUOp[3:0]	output	ALU操作选择信号
ExtOp[2:0]	output	Ext操作选择信号
nPC_sel[2:0]	output	nPC操作选择信号
MemWrite	output	1是可以向DM写入数据
RegDst[1:0]	output	GRF的A3数据来源
RegWrite	output	1时可以向Reg写入数据
RegType[1:0]	output	GRF的WD数据来源

- 代码：

```
`timescale 1ns / 1ps
`default_nettype none
`define TRUE 1'b1
//R类型
`define R 6'b000000
`define ADD 6'b100000
`define SUB 6'b100010
`define JR 6'b001000    //!!!
//I类型
`define ORI 6'b001101
`define LUI 6'b001111
`define BEQ 6'b000100
`define LW 6'b100011
`define SW 6'b101011
//J类型
`define JAL 6'b000011
//////////////////////////////////////////////////////////////////////////////////
module Controller(
    input wire [5:0] op,          //opCode
    input wire [5:0] fc,            //func
    output wire ALUSrc,               //ALUSrc==1'b1时ALU的第二个操作数为立即数，否则为
寄存器
    output wire [3:0] ALUOp,          //ALUOp:加、减、或、比
    output wire [2:0] ExtOp,          //ExtOp：零扩展、符号扩展、移到高位并后补零
    output wire [2:0] nPCSel,         //nPCSel：正常状况、beq状况、jal状况、jr状况
    output wire MemWrite,             //MemWrite为1'b1时向DM写入数据
    output wire [2:0] DMOp,           //DMOp:word,half word,byte
```

```
    output wire [1:0] RegDst,         //RegDst: 2'b00:rt, 2'b01:rd, 2'b10:ra
    output wire RegWrite,             //RegWrite为1'b1时向GRF写入数据
    output wire [1:0]RegType          //RegType: 2'b00:from ALU, 2'b01:from DM,
2'b10:from PC+4
    );
    /* 与逻辑 */
    //R类型
    wire add = (op == `R)&&(fc == `ADD);
    wire sub = (op == `R)&&(fc == `SUB);
    wire jr = (op == `R)&&(fc == `JR);
    //I类型
    wire ori = (op == `ORI);
    wire lui = (op == `LUI);
    wire beq = (op == `BEQ);
    wire lw = (op == `LW);
    wire sw = (op == `SW);
    //J类型
    wire jal = (op == `JAL);

    /* 或逻辑 */
    assign ALUSrc = ((`TRUE == ori) || (`TRUE == lui) || (`TRUE == lw) || (`TRUE
== sw)) ? 1'b1:1'b0;
    assign ALUOp = (`TRUE == sub) ? 4'b0001 :
                   (`TRUE == ori) ? 4'b0010 :
                   (`TRUE == beq) ? 4'b0011 :
                   4'b0000;
    assign ExtOp = (`TRUE == lw || `TRUE == sw) ? 3'b001 :
                   (`TRUE == lui) ? 3'b010 :
                   3'b000;
    assign nPCSel = (`TRUE == beq) ? 3'b001 :
                    (`TRUE == jal) ? 3'b010 :
                    (`TRUE == jr) ? 3'b011 :
                    3'b000;
    assign MemWrite = (`TRUE == sw) ? 1'b1 : 1'b0;
    assign DMOp = 3'b000; //待扩展
    assign RegDst = (`TRUE == add || `TRUE == sub) ? 2'b01 :
                    (`TRUE == jal) ? 2'b10 :
                    2'b00;
    assign RegWrite = (`TRUE == add || `TRUE == sub || `TRUE == ori || `TRUE ==
lui || `TRUE == lw || `TRUE == jal) ? 1'b1 : 1'b0;
    assign RegType  = (`TRUE == lw) ? 2'b01 :
                      (`TRUE == jal) ? 2'b10 :
                      2'b00;
  endmodule
```

IFU(取指令单元)

- 端口:

端口名称	方向	描述
clk	input	时钟信号
reset	input	同步复位信号，1时将PC复位到0x0000_30000

端口名称	方向	描述
nextPC[31:0]	input	下一周期的PC值
nowPC[31:0]	output	本周期执行指令的PC值
inStr[31:0]	output	本周期执行指令的二进制编码

- 代码:

```
    `timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
module IFU(
    input wire clk,
    input wire reset,
    input wire [31:0] nextPC,
    output wire [31:0] nowPC,
    output wire [31:0] inStr
    );
    //计算nowPC
    reg [31:0]nowPCReg;
    initial begin
        nowPCReg = 32'h0000_3000;
    end
    assign nowPC = nowPCReg;
    always @(posedge clk) begin
        if (1'b1 == reset) begin
            nowPCReg <= 32'h0000_3000;
        end
        else begin
            nowPCReg <= nextPC;
        end
    end

    //得到inStr
    reg [31:0] IM [0:4095];  //IM
    initial begin
       $readmemh("code.txt",IM);
    end
    wire [31:0]addr;
    assign addr = nowPC - 32'h0000_3000;
    assign inStr = IM[addr[13:2]];
endmodule
```

EXT(扩展器)

- 端口:

信号名	方向	描述
op[2:0]	input	操作选择信号
input[15:0]	input	16位操作数

信号名	方向	描述
output[31:0]	output	32为结果数

- 代码：

```
    `timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
module EXT(
    input wire [2:0] op,
    input wire [15:0] in,
    output wire [31:0] out
    );

    reg [31:0]outReg;

    assign out = outReg;

    always @(*) begin
        case (op)
            //3'b000  0扩展
            3'b000: outReg = {{16'h0000},in};
            //3'b001  符号扩展
            3'b001: outReg = {{16{in[15]}},in};
            //3'b010  移到高位，后补0
            3'b010: outReg = {in,{16'h0000}};
            default: outReg = 32'd0;
        endcase
    end
endmodule
```

ALU(算数逻辑单元)

- 端口：

信号名	方向	描述
op[3:0]	input	操作选择信息
in1[31:0]	input	操作数1
in2[31:0]	input	操作数2
in3[31:0]	input	操作数2
result[31:0]	output	结果数
equalZero	output	判断结果是否为零，1时为0

- 代码：

```
    `timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
```

```
module ALU(
    input wire [3:0] op,
    input wire [31:0] in1,
    input wire [31:0] in2,
    input wire [31:0] in3,
    output wire [31:0] result,
    output wire equalZero            //判断结果是否为零，为零输出1'b1
    );

    reg [31:0]resultReg;

    assign equalZero = (result == 32'b0) ? 1'b1 : 1'b0;
    assign result = resultReg;

    always @(*) begin
        case (op)
            //4'b0000  加法
            4'b0000: resultReg = in1 + in2;
            //4'b0001  减法
            4'b0001: resultReg = in1 - in2;
            //4'b0010  逻辑或
            4'b0010: resultReg = in1 | in2;
            //4'b0011  比较大小  看作有符号数！！！
            //等于输出0，大于输出1，小于输出-1
            4'b0011: begin
                if ($signed(in1) == $signed(in2)) resultReg = 32'd0;
                else if ($signed(in1) > $signed(in2)) resultReg = 32'd1;
                else resultReg = 32'hffff_ffff;
            end
            default: resultReg = 32'd0;
        endcase
    end
endmodule
```

GRF(通用寄存器组)

- 端口：

信号名	方向	描述
clk	input	时钟信号
reset	input	同步复位信号
nowPC[31:0]	input	PC地址
A1[4:0]	input	地址输入信号，对应R1
A2[4:0]	input	地址输入信号，对应R2
A3[4:0]	input	地址输入信号，将对应的寄存器写入的目标
WE	input	写使能信号，1时可以向GRF中写入数据，0时不可
WD[31:0]	input	需要写入的数据

信号名	方向	描述
RD1[31:0]	output	输出A1指定的寄存器中的值
RD2[31:0]	output	输出A2指定的寄存器中的值

- 代码：

```
    `timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
module GRF(
    input wire clk,
    input wire reset,
    input wire [31:0]nowPC,
    input wire [4:0] A1,
    input wire [4:0] A2,
    input wire[4:0] A3,
    input wire WE,
    input wire [31:0] WD,
    output wire [31:0] RD1,
    output wire [31:0] RD2
    );
    //定义
    reg [31:0] grf [0:31];  //registers
    integer i = 0;

    //初始化
    initial begin
        for (i = 0;i < 32;i = i + 1) begin
            grf[i] = 32'd0;
        end
    end
    //assign & always
    assign RD1 = grf[A1];
    assign RD2 = grf[A2];
    always @(posedge clk) begin
        if (1'b1 == reset) begin
            for (i = 0;i < 32;i = i + 1) begin
                grf[i] <= 32'd0;
            end
        end
        else begin
            if ((1'b1 == WE)&&(5'b0 != A3)) begin
                $display("@%h: $%d <= %h",nowPC,A3,WD);
                grf[A3] <= WD;
            end
            else grf[A3] <= grf[A3];
        end
    end
endmodule
```

DM(数据存储器)

- 端口：

信号名	方向	描述
clk	input	时钟信号
reset	input	同步复位信号
nowPC[31:0]	input	指令地址
MemWrite	input	写入控制信号
addr[31:0]	input	读取/写入地址
inputDate[31:0]	input	写入数据
op[2:0]	input	数据位宽控制信号
outputDate[31:0]	output	读取数据

- 代码：

```
    `timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
//
module DM(
    input wire clk,
    input wire reset,
    input wire [31:0] nowPC,
    input wire MemWrite,
    input wire [31:0] addr,
    input wire [31:0] inputDate,
    input wire [2:0] op,  //(000:lw/sw)(001:lh/sh)(010:lb/sb)
    output wire [31:0] outputDate
    );
    reg [31:0] dm [0:3071];
    reg [31:0] outputReg;
    integer i = 0;

    initial begin
        for (i = 0;i < 3072;i = i + 1) begin
            dm[i] = 0;
        end
    end

    assign outputDate = outputReg;

    //output
    always @(*) begin
        case (op)
            //word
            3'b000: outputReg = dm[addr[13:2]];
            //half word
```

```
            3'b001: begin
                if (addr[1] == 1'b0) outputReg = {{16'd0},{dm[addr[13:2]]
[15:0]}};
                else outputReg = {{16'd0},dm[addr[13:2]][31:16]};
            end
            //byte
            3'b010: begin
                if (addr[1:0] == 2'b00) outputReg = {{24'd0},dm[addr[13:2]]
[7:0]};
                else if (addr[1:0] == 2'b01) outputReg = {{24'd0},dm[addr[13:2]]
[15:8]};
                else if (addr[1:0] == 2'b10) outputReg = {{24'd0},dm[addr[13:2]]
[23:16]};
                else outputReg = {{24'd0},dm[addr[13:2]][31:24]};
            end
            default: outputReg = 32'd0;
        endcase
    end
    always @(posedge clk) begin
        if (1'b1 == reset) begin
            for (i = 0;i < 3072;i = i + 1) begin
                dm[i] = 0;
            end
            outputReg <= 32'd0;
        end
        else if (1'b1 == MemWrite) begin
            $display("@%h: *%h <= %h",nowPC,addr,inputDate);
            case (op)
                //word
                3'b000: dm[addr[13:2]] <= inputDate;
                //half word
                3'b001: begin
                    if (addr[1] == 1'b0) dm[addr[13:2]][15:0] <=
inputDate[15:0];
                    else dm[addr[13:2]][31:16] <= inputDate[15:0];
                end
                //byte
                3'b010: begin
                    if (addr[1:0] == 2'b00) dm[addr[13:2]][7:0] <=
inputDate[7:0];
                    else if (addr[1:0] == 2'b01) dm[addr[13:2]][15:8] <=
inputDate[7:0];
                    else if (addr[1:0] == 2'b10) dm[addr[13:2]][23:16] <=
inputDate[7:0];
                    else dm[addr[13:2]][31:24] <= inputDate[7:0];
                end
                default: dm[addr[13:2]] <= dm[addr[13:2]];
            endcase
        end
        else dm[addr[13:2]] <= dm[addr[13:2]];
    end
endmodule
```

nPC(指令地址计算单元)

- 端口:

信号名	方向	描述
op[2:0]	input	PC选择控制信号
equalZero	input	PC跳转判断信号
nowPC[31:0]	input	目前正在执行的指令的地址
immNum16[15:0]	input	beq类型
immNum26[25:0]	input	jal类型
regDate[31:0]	input	jr类型
pcAddr4[31:0]	input	nowPC加4后的值
nextPC[31:0]	output	下一个要执行的指令的地址

- 代码:

```
    `timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
module nPC(
    input wire [2:0] op,
    input wire equalZero,
    input wire [31:0]nowPC,
    input wire [15:0] immNum16,
    input wire [25:0] immNum26,
    input wire [31:0] regDate,
    output wire [31:0] pcAdd4,
    output wire [31:0] nextPC
    );
    reg [31:0] npc;

    assign pcAdd4 = nowPC + 32'd4;
    assign nextPC = npc;

    always@(*) begin
        case (op)
            // 正常状况:  pc <- pc + 4
            3'b000: npc =  nowPC + 32'd4;
            // beq状况
            3'b001: begin
                if (1'b1 == equalZero) npc = nowPC + 32'd4 +
{{14{immNum16[15]}},immNum16,{2'b00}};
                else npc = nowPC + 32'd4;
            end
            // jal状况
            3'b010: npc = {{nowPC[31:28]},{immNum26},{2'b00}};
            // jr状况
            3'b011: npc = regDate;
```

```
            default: npc = npc;
        endcase
    end
endmodule
```

MUX(多路选择器)

~~略过不谈~~

顶层设计电路

- 设计思路：分为三个部分：变量声明，端口连接，实例化子电路
- 代码：

```
    /*author:zyt 22373337 15269899725*/
`timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
//
module mips(
    input wire clk,
    input wire reset
    );
//////////////////////////////////////////////////////////////////////////////////
    /* Controller */
    wire [5:0] opCode;
    wire [5:0] func;
    wire ALUSrc;
    wire [3:0] ALUOp;
    wire [2:0] ExtOp;
    wire [2:0] nPCSel;
    wire MemWrite;
    wire [2:0] DMOp;
    wire [1:0] RegDst;
    wire RegWrite;
    wire [1:0] RegType;
    /* IFU */
    //nextPC将在nextPC模块中定义
    wire [31:0] nowPC;
    wire [31:0] inStr;
    /* EXT */
    //op已经在Controller模块中定义
    wire [15:0] ExtIn;
    wire [31:0] ExtOut;
    /* ALU */
    //ALUOp已经在Controller模块中定义
    wire [31:0] ALUIn1;
    wire [31:0] ALUIn2;
    wire [31:0] ALUOut;
    wire equalZero;
    /* GRF */
    //WE已经在Controller模块中定义
    //nowPC已经在IFU模块中定义
    wire [4:0] gA1;
    wire [4:0] gA2;
```

```
    wire [4:0] gA3;
    wire [31:0] gWD;
    wire [31:0] gRD1;
    wire [31:0] gRD2;
    /* DM */
    //MemWrite,op已经在Controller模块中定义
    //nowPC已经在IFU模块中定义
    wire [31:0] DMAddr;
    wire [31:0] DMIn;
    wire [31:0] DMOut;
    /* nextPC */
    //op已经在Controller模块中定义
    //equalZero已经在ALU模块中定义
    //nowPC已经在IFU模块中定义
    wire [15:0] nPCNum16;
    wire [25:0] nPCNum26;
    wire [31:0] nPCReg;
    wire [31:0] pcAdd4;
    wire [31:0] nextPC;
////////////////////////////////////////////////////////////////////////////////
    //Controller
    assign opCode = inStr[31:26];
    assign func = inStr[5:0];
    //Ext
    assign ExtIn = inStr[15:0];
    //ALU
    assign ALUIn1 = gRD1;
    MUX_32_2 muxALUSrc (
        .in0(gRD2),
        .in1(ExtOut),
        .op(ALUSrc),
        .out(ALUIn2)
    );
    //GRF
    assign gA1 = inStr[25:21];
    assign gA2 = inStr[20:16];
    reg [4:0] ra = 5'd31;
    MUX_32_4 muxRegAddr (
        .in0(inStr[20:16]),
        .in1(inStr[15:11]),
        .in2(ra),
        .op(RegDst),
        .out(gA3)
    );
    MUX_32_4 muxRegWD (
        .in0(ALUOut),
        .in1(DMOut),
        .in2(pcAdd4),
        .op(RegType),
        .out(gWD)
    );
    //DM
    assign DMAddr = ALUOut;
    assign DMIn = gRD2;
    //nextPC
```

```
    assign nPCNum16 = inStr[15:0];
    assign nPCNum26 = inStr[25:0];
    assign nPCReg = gRD1;
////////////////////////////////////////////////////////////////////////////////
    Controller controller (
        .op(opCode),
        .fc(func),
        .ALUSrc(ALUSrc),
        .ALUOp(ALUOp),
        .ExtOp(ExtOp),
        .nPCSel(nPCSel),
        .MemWrite(MemWrite),
        .DMOp(DMOp),
        .RegDst(RegDst),
        .RegWrite(RegWrite),
        .RegType(RegType)
    );
    IFU ifu (
        .clk(clk),
        .reset(reset),
        .nextPC(nextPC),
        .nowPC(nowPC),
        .inStr(inStr)
    );
    EXT ext (
        .op(ExtOp),
        .in(ExtIn),
        .out(ExtOut)
    );
    ALU alu (
        .op(ALUOp),
        .in1(ALUIn1),
        .in2(ALUIn2),
        .result(ALUOut),
        .equalZero(equalZero)
    );
    GRF grf (
        .clk(clk),
        .reset(reset),
        .nowPC(nowPC),
        .A1(gA1),
        .A2(gA2),
        .A3(gA3),
        .WE(RegWrite),
        .WD(gWD),
        .RD1(gRD1),
        .RD2(gRD2)
    );
    DM dm (
        .clk(clk),
        .reset(reset),
        .nowPC(nowPC),
        .MemWrite(MemWrite),
        .addr(ALUOut),
        .inputDate(DMIn),
```

```
        .op(DMOp),
        .outputDate(DMOut)
    );
    nPC npc (
        .op(nPCSel),
        .equalZero(equalZero),
        .nowPC(nowPC),
        .immNum16(nPCNum16),
        .immNum26(nPCNum26),
        .regDate(nPCReg),
        .pcAdd4(pcAdd4),
        .nextPC(nextPC)
    );
endmodule
```

思考题

第一题 根据你的理解，在下面给出的DM的输入示例中，地址新号addr位数为什么是[11:2]而不是[9:0]?

答：地址的单位是byte，DM中存储的单元是reg[31:0],即四个byte，那么addr的位数应该从2开始；另一方面，DM的容量为4KB，即32bit*1024字，那么应该需要十位地址，即addr的位数应该到11。

第二题 思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

答：代码示例：

```
//第一种译码方式
always @(*) begin
    //仅举一例
    if (`TRUE == sub) begin
        ALUOp = 4'b0001;
        RegDst = 2'b01;
        RegWrite = 1'b1;
    end
    //.....//
end
```

```
//第二种译码方式
    assign ALUSrc = ((`TRUE == ori) || (`TRUE == lui) || (`TRUE == lw) || (`TRUE
== sw)) ? 1'b1:1'b0;
    assign ALUOp = (`TRUE == sub) ? 4'b0001 :
                   (`TRUE == ori) ? 4'b0010 :
                   (`TRUE == beq) ? 4'b0011 :
                   4'b0000;
    assign ExtOp = (`TRUE == lw || `TRUE == sw) ? 3'b001 :
                   (`TRUE == lui) ? 3'b010 :
                   3'b000;
    assign nPCSel = (`TRUE == beq) ? 3'b001 :
                    (`TRUE == jal) ? 3'b010 :
                    (`TRUE == jr) ? 3'b011 :
                    3'b000;
```

```
    assign MemWrite = (`TRUE == sw) ? 1'b1 : 1'b0;
    assign DMOp = 3'b000; //待扩展
    assign RegDst = (`TRUE == add || `TRUE == sub) ? 2'b01 :
                    (`TRUE == jal) ? 2'b10 :
                    2'b00;
    assign RegWrite = (`TRUE == add || `TRUE == sub || `TRUE == ori || `TRUE ==
lui || `TRUE == lw || `TRUE == jal) ? 1'b1 : 1'b0;
    assign RegType  = (`TRUE == lw) ? 2'b01 :
                      (`TRUE == jal) ? 2'b10 :
                      2'b00;
```

与第一种译码方式，第二种译码方式类似p3中的或逻辑，关注的重点放在了Controller的输出上，符合思考逻辑，所以我认为第二种译码方式优于第一种译码方式。

第三题 在相应的部件中，复位信号的设计都是同步复位，这与p3中的设计要求不同。请对比同步复位与异步复位这两种方式的reset信号与clk信号优先级的关系。

答：在异步复位中，reset的优先级高于clk；在同步复位中。reset的优先级低于clk。

第四题 C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分（详见文档 page 34、page 35）。

答:

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp
endif
```

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

第一张图是addi的Operation部分，第二张图是addiu的Operation部分，从上面可以看到，在忽略溢出的情况下，那么addi只会执行else部分，而两个指令的temp是相同的，那么addi与addiu是等价的。同理可得add与addu是等价的。

测试方案

1. 写了一个自动生成300行mips代码的python代码
   测试的重点放在了0，-1,1，65535等边界值
   还可以测试beq指令，避开了死循环
2. 同时还用了p3的测试代码

```
.text
#lui
lui $0,0x1234
ori $t0,0xabcd
lui $t0,0xffff
#ori
ori $t1,$t0,0x1234
ori $0,$t1,0xffff
#add
ori $t2,0x1234
add $t2,$t2,$t2
add $0,$t2,$t2
ori $t3,0x0001
lui $t4,0xffff
ori $t4,$t4,0xfffe
add $t4,$t3,$t4
nop
#sub
lui $t5,0xffff
ori $t5,0xffff
ori $t6,0xabcd
sub $t6,$t5,$t6
sub $t5,$t5,$t5
nop
#sw
ori $t7,4
sw $t7,-4($t7)
lw $s4,-4($t7)
sw $t0,4($0)
sw $t1,8($0)
sw $t2,12($0)
sw $t3,12($0)
sw $0,16($0)
nop
#jal
jal func
ori $v0,1
#beq
ori $s0,0x1
ori $s1,0xffff
beq $s0,$s1,b1
b2:
sub $s1,$s1,$s1
beq $0,$s1,b3
b1:
beq $0,$0,b2
b3:
ori $s2,11451
add $s3,$s2,$0
lw $s7,0($0)
lw $s6,4($0)
lw $s5,8($0)
b4:
beq $s3,$s2,b4
```

```
func:
ori $a0,0x1234
add $a1,$a0,$a0
ori $a2,4
add $a2,$ra,$a2
jr $ra
```

debug过程中用到的一些脚本

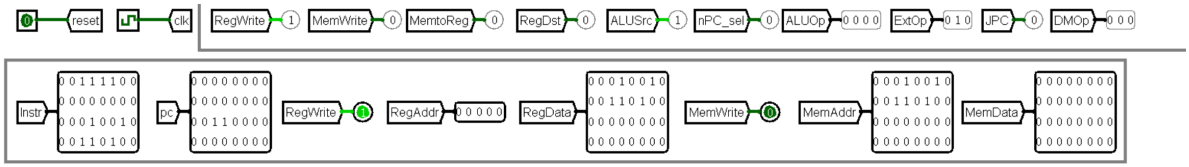
十六进制转二进制

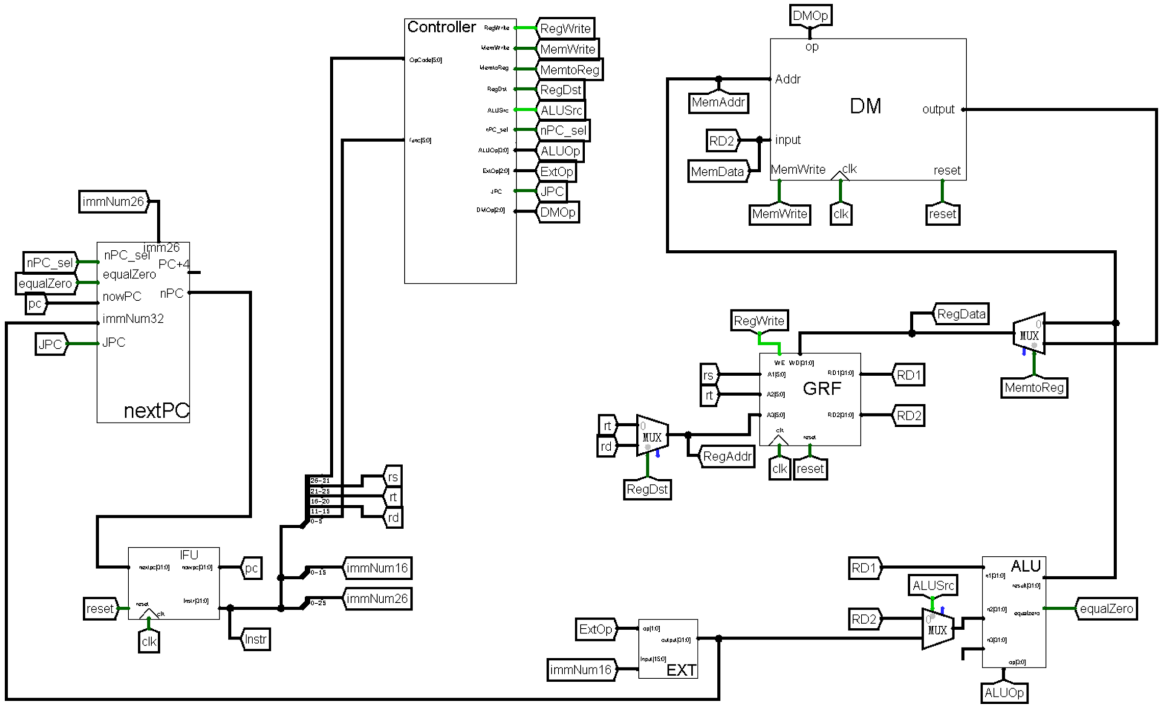
```python
fp = open("testcode.txt", "r+")
out = open("out.txt", "w")
fp.readline()
for line in fp:
    num = bin(int(line, 16))
    num = num[2:]
    while len(num) < 32:
        num = "0" + num
    out.write("{}\n".format(num))
```

Logisim单周期cpu各模块图片

本部分为p3课下部分所搭的cpu的图片，与p4并不完全相同，也存在一些错误

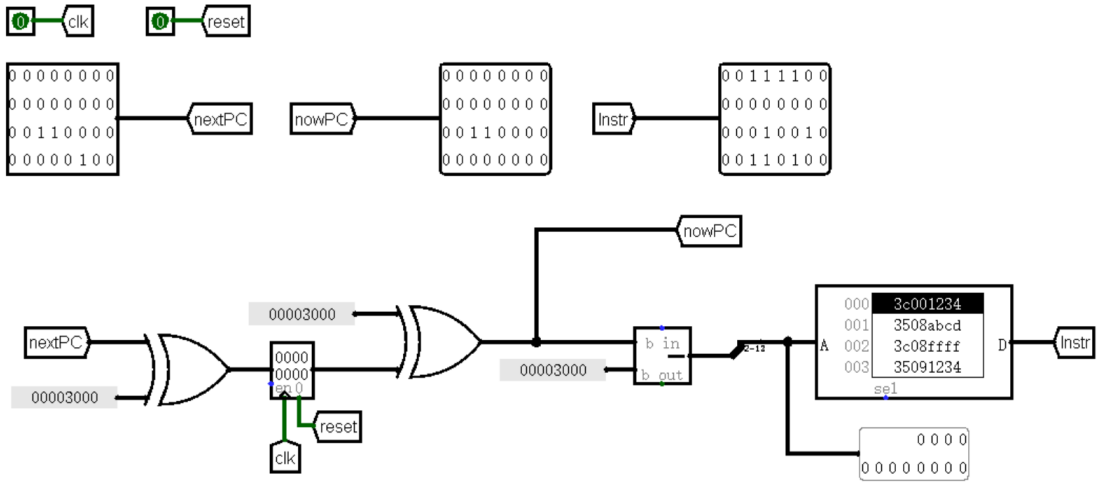
顶层电路





子电路

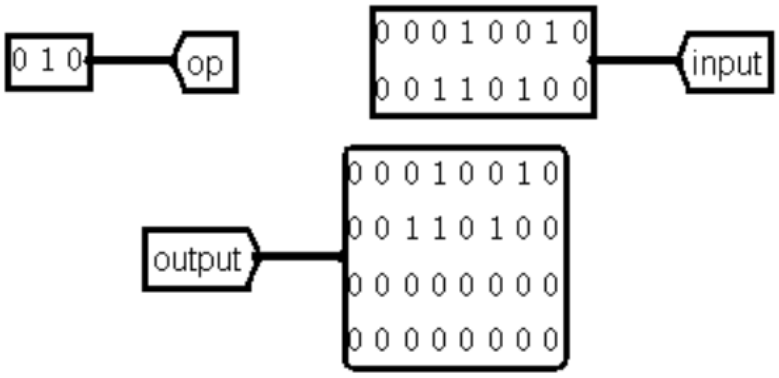
1. IFU

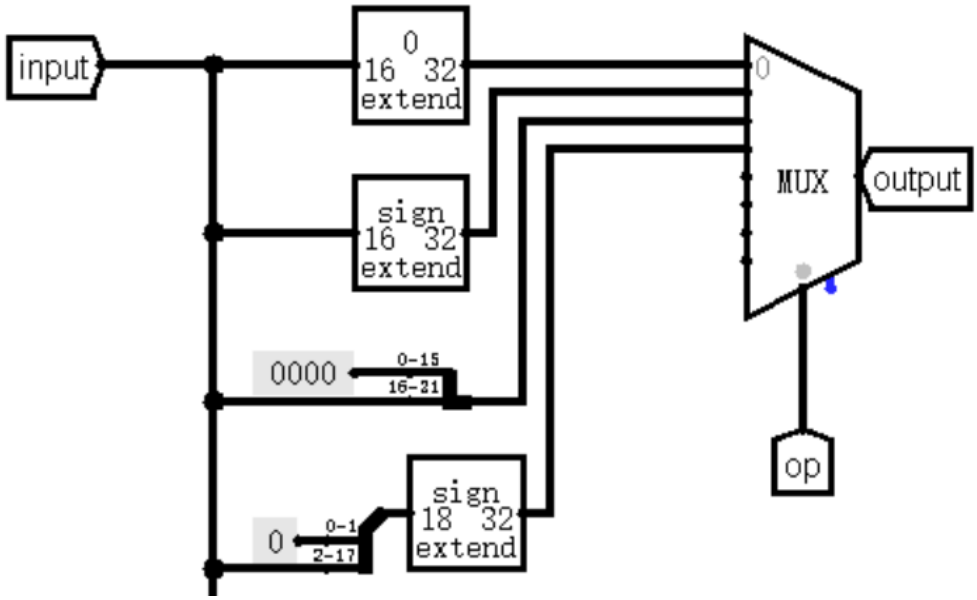


初始化3000

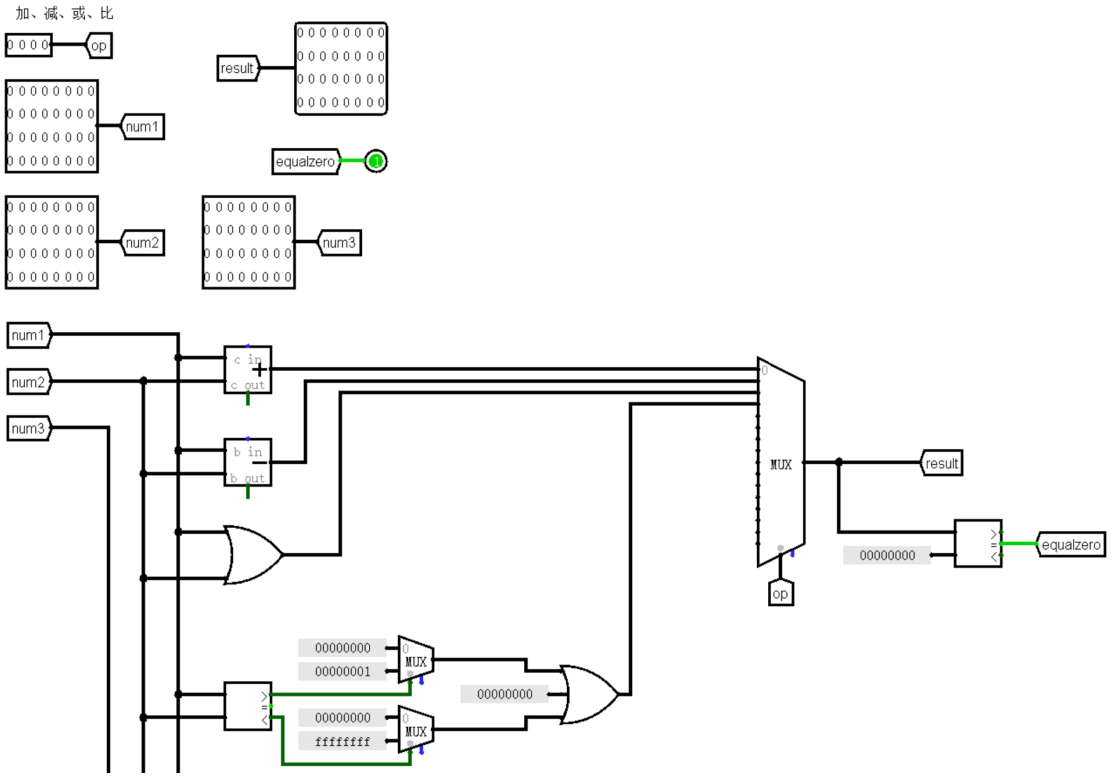
2. EXT

00:0扩展　01:符号扩展　10：移到高位　11：后补两个0，然后符号扩展

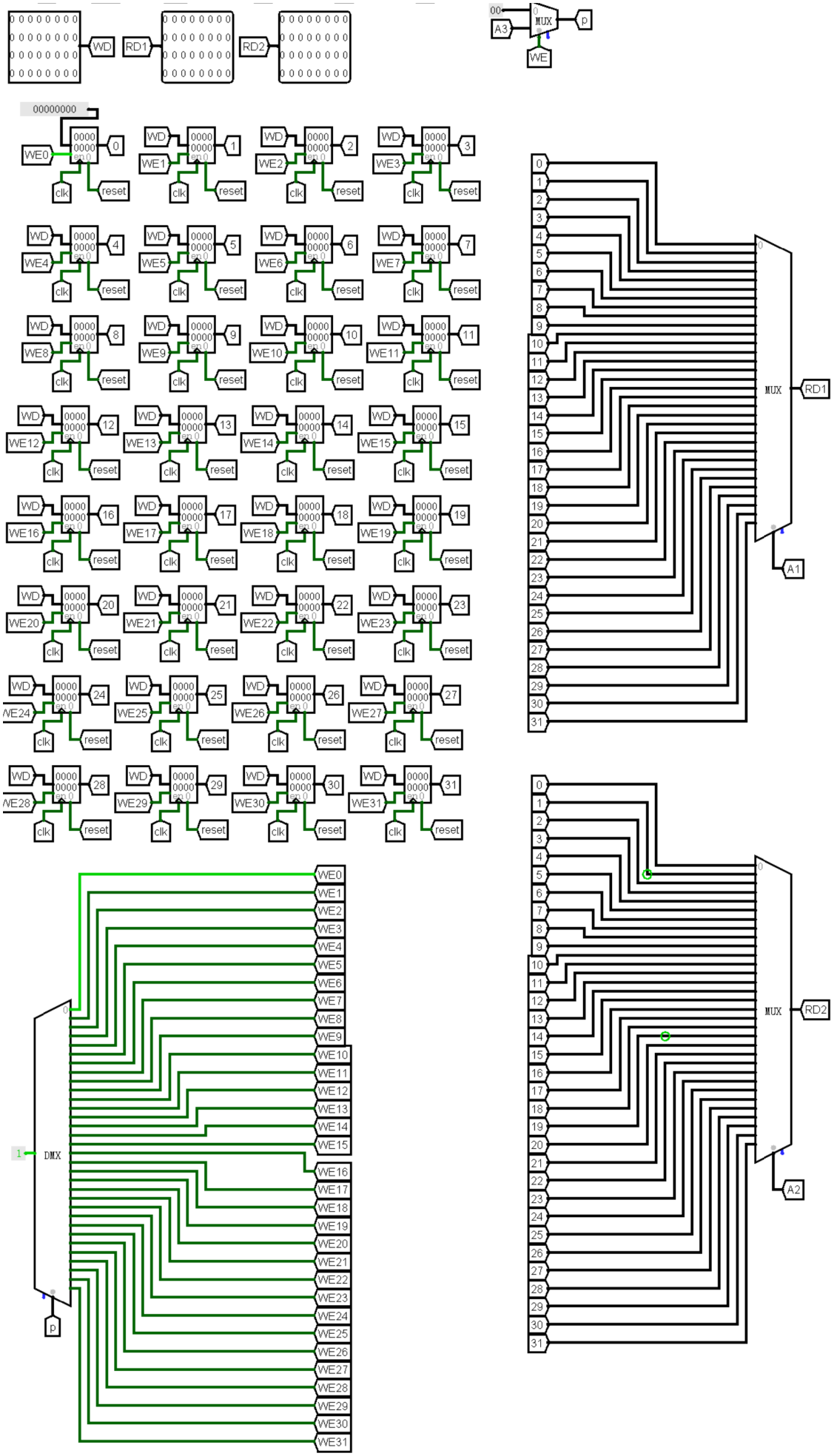




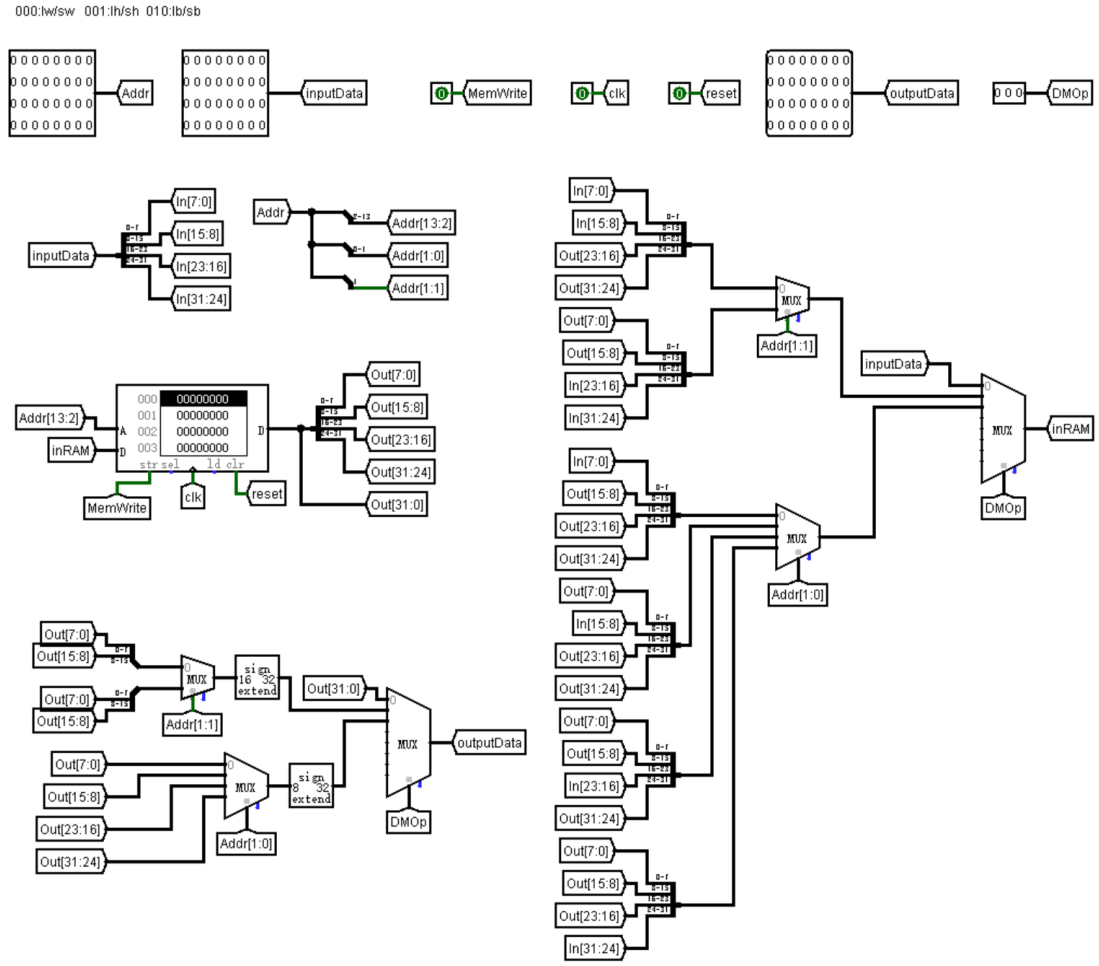
3. ALU



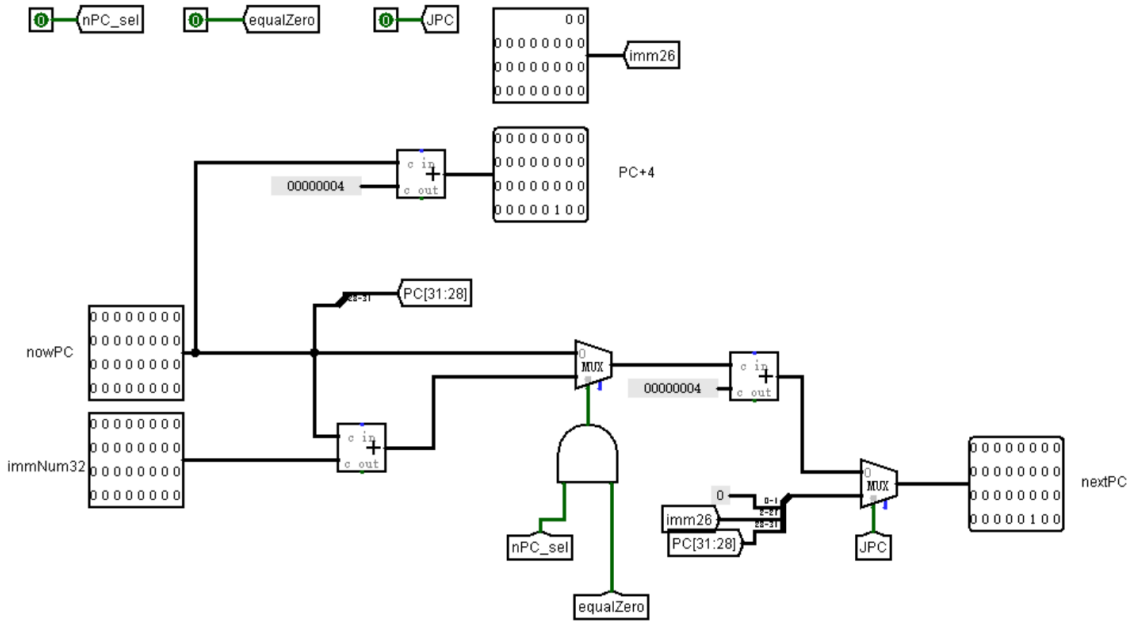
4. GRF


WD
RD1
RD2
00
A3
MUX
WE
p
00000000
WE0
WE1
WE2
WE3
WE4
WE5
WE6
WE7
WE8
WE9
WE10
WE11
WE12
WE13
WE14
WE15
WE16
WE17
WE18
WE19
WE20
WE21
WE22
WE23
WE24
WE25
WE26
WE27
WE28
WE29
WE30
WE31
clk
reset
A1
A2
DMX

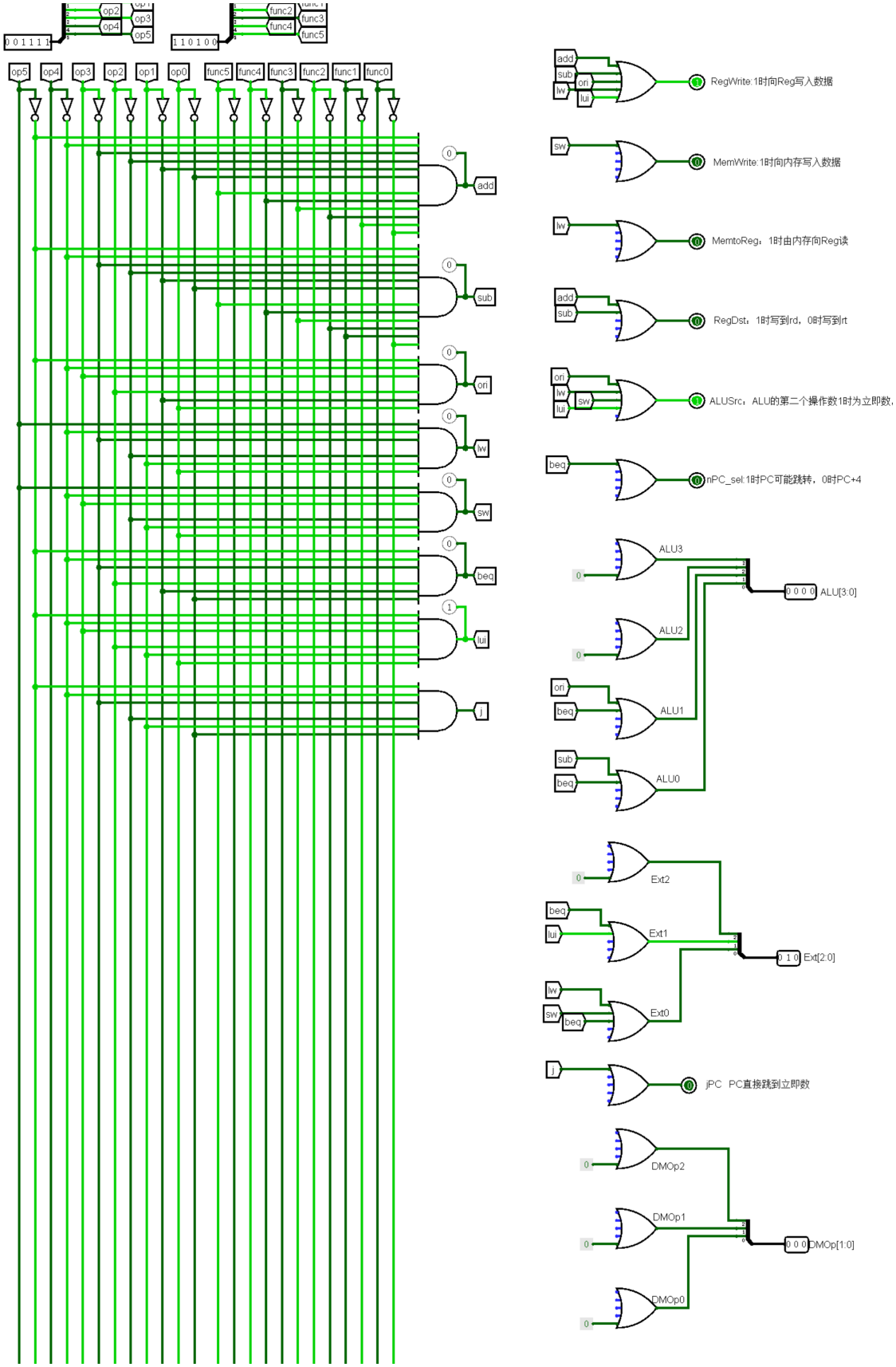
5. DM



6. nPC



7. Controller


op5
op4
op3
op2
op1
op0
func5
func4
func3
func2
func1
func0
add
sub
ori
lw
sw
beq
lui
j
RegWrite:1时向Reg写入数据
MemWrite:1时向内存写入数据
MemtoReg，1时由内存向Reg读
RegDst，1时写到rd，0时写到rt
ALUSrc，ALU的第二个操作数1时为立即数，
nPC_sel:1时PC可能跳转，0时PC+4
ALU3
ALU2
ALU1
ALU0
ALU[3:0]
Ext2
Ext1
Ext0
Ext[2:0]
jPC PC直接跳到立即数
DMOp2
DMOp1
DMOp0
DMOp[1:0]