



计算机组成原理课程设计

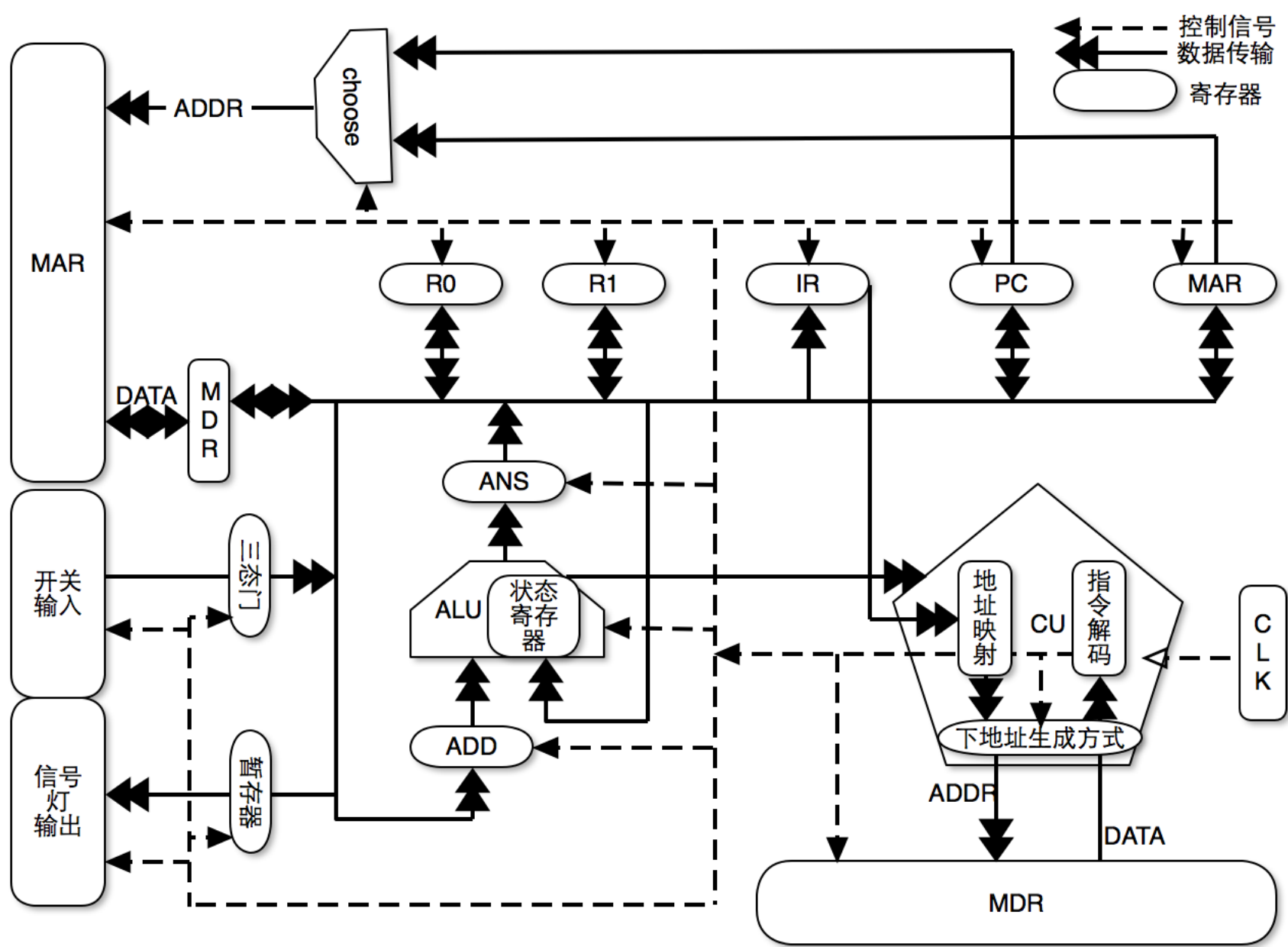
# 实验报告

计算机 15 级四班张景涛

实验三 .....	3
模式图 .....	3
数据通路说明 .....	3
物理连线图 .....	4
组成详解 .....	5
寄存器 .....	5
计算与逻辑处理单元 .....	7
CU .....	11
微指令详解 .....	14
微指令设计举例 .....	15
指令(8 位)设计 .....	17
辅助程序说明 .....	19
助记符转换成 Verilog(decode_7 模块)映射关系代码、微程序(ROM)内存内容与指令和指令名称的映射文件 .....	19
类汇编语言转换成机器码(存放于 RAM 的指令)简易编译器 .....	19
测试 .....	21
测试一 .....	21
测试二 .....	21
实验三 .....	22
测试四 .....	23
实验感悟 .....	24
拓展空间 .....	25
实验四 .....	25
模式图 .....	25
物理连接图 .....	26
组件详解 .....	26
节拍发生器 .....	26
ALU .....	27
CU .....	28
对 RAM 的读写 .....	29
控制指令的设计 .....	29
JAVA 程序 .....	30
测试 .....	30
测试运算计算器 .....	30
遇到的问题 .....	30
拓展空间 .....	30

实验三

模式图

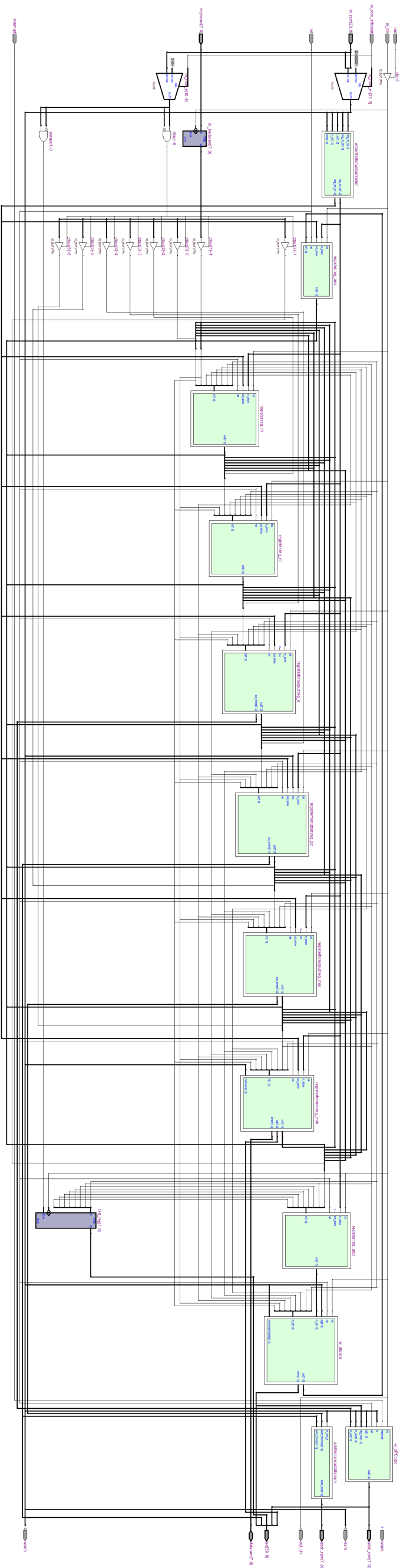


模型机的整体组成方式如上图所示。分为两大通路：数据通路和控制通路。分为四大组成部分：寄存器组，CU，计算与逻辑处理单元(choose, ALU)和外设(MAR, MDR, 输入，输出)；

数据通路说明

在数据通路中，每一个时刻最多只有一个寄存器可以读(将自己锁存的数据输出到总线上)，最多只有一个寄存器可以写(刷新自己锁存的数据)，寄存器的写读由 CU 控制指令生成模块给出。外设输出到灯，可以独立工作，暂存器在接收锁存控制信号时锁存总线上的数据。

物理连线图



组成详解

寄存器

寄存器分为三种，普通寄存器，带直输出的寄存器，MDR 专用寄存器

其中

R0, R1, ADD, ANS 寄存器使用普通寄存器；

IR, PC, MAR 寄存器使用带直输出的寄存器；

MDR 使用 MDR 专用寄存器；

普通寄存器

输入输出：

时钟脉冲输入clk，数据输入 in，数据输出 out，允许输入in\_allow，允许输出out\_allow，置 0 rst

逻辑思想：

- 1、 在时钟的下降沿触发操作
- 2、 当 rst == 0 mem 置 mem 初值 0
- 3、 当 in\_allow == 1 将寄存器的值刷成输入的值
- 4、 输出由 out\_allow 来选择是输出 mem，还是高阻态(相当于三态门 断开输出)

代码实现：

```
/**
 * 普通寄存器设计模型
 */
module register(
    input clk,
    input [7:0] in,
    output wire [7:0] out,
    input out_allow,
    input in_allow,
    input rst);
    /**
     * rst = 0 清零
     * in_allow 锁存数据
     * out_allow 输出数据
     */
    reg [7:0] mem;
    always @(negedge clk) begin
        if (!rst)
```

```

        mem <= 8'b00000000;
    else if (in_allow)
        mem <= in;
    end
    assign out = out_allow ? mem : 8'bz;
endmodule

```

## 带直输出的寄存器

带直输出的寄存器相比于普通寄存器，多了一个直输出的数据通路，该数据通路直接连接到了 mem 上，多了 mem 自增的信号(自增信号只用于 PC 寄存器)。

```

/**
 * for MAR IR PC 专用寄存器
 */
module registerforoutput(
    input clk,
    input [7:0] in,
    input wire inc,
    output wire [7:0] out,
    input out_allow,
    input in_allow,
    input rst,
    output wire [7:0] out_direct
);
    /**
     * rst = 0  清零
     * in_allow 锁存数据
     * out_allow 输出数据
     */
    reg [7:0] mem;
    always @(negedge clk) begin
        if (!rst)
            mem <= 8'b00000000;
        else if (in_allow)
            mem <= in;
        if(inc)
            mem <= mem+1;
    end
    assign out_direct = mem;
    assign out = out_allow ? mem : 8'bz;
endmodule

```

## MDR 专用寄存器

MDR 专用寄存器是最复杂的寄存器，MDR 寄存器是在普通寄存器的基础上添加了与 MAR 之间的数据连接。在普通的寄存器的基础上添加读 MAR 和写 MAR 的两个指令。

```

/**
 * MDR 专用寄存器 http://blog.sina.com.cn/s/blog\_7bf0c30f0100tedd.html

```

```

*/
module registerformdr(
    input clk,
    input [7:0] in,
    output wire [7:0] out,
    input out_allow,
    input in_allow,
    input rst,
    inout wire [7:0] ioram/*连接 ram*/,
    input wire [1:0] rwforram /*对 ram 输出还是接收数据*/
);

    /**
    *   rst = 0    清零
    *   in_allow   锁存数据
    *   out_allow  输出数据
    *   rwforram[1] ioram 对外输出
    *   rwforram[0] ioram 对内输入
    */
    reg [7:0] mem;
    always @(negedge clk) begin
        if (!rst)
            mem <= 8'b0;
        else if (in_allow)
            mem <= in;
        else if (rwforram[0]) //read ram
            mem <= ioram;
    end
    assign out = out_allow ? mem : 8'bz;
    //写 ram
    assign ioram = rwforram[1] ? mem : 8'bz;
endmodule

```

## 计算与逻辑处理单元

### ALU

ALU 可以实现的功能： 加法、减法、乘法、in\_b 取非、异或、in\_b 自增、高阻态(不输出)

其中加法分为双符号位的补码运算、记录进位的加法，使用进位的加法、记录并使用进位的加法

减法为双符号位的补码运算

计算过程中若结果为全 0 会自动将 2 号状态位自动置 1

```

/**
* 组合逻辑的 ALU 设计电路
* clk,op 选择要执行的操作,
* in_a 连接累加器, in_b 连接 BUS,cin 进位标志位, out 连接到总线,cout 连接状态寄存器
*/
module w_alu (

```

```

input clk, input rst, input [2:0] op/*选择要执行的操作*/, input [7:0] in_a/*连接累加器*/, input [7:0] in_b/*连接 BUS*/,
input wire [1:0] controllerforstate,/*状态寄存器控制器*/
output reg [7:0] out/*连接到总线*/, output reg [7:0] mem/*状态寄存器的输出*/);
/**
 * op 000 加法
 * op 001 减法
 * op 010 乘法
 * op 011 对 in_b 取非
 * op 100 in_a, in_b 异或
 * op 101 in_b, 自增 1      为 PC 服务
 * op 110 除
 * op 111 不输出
 *
 * 状态寄存器 若结果全为 0 则判断最终的结果的为 1      即相等
 */
wire [7:0] in_allow;
reg cout_add; reg cout_sub;
wire equal; /*判断输出是否为全 0*/
assign equal = out[0] | out[1] | out[2] | out[3] | out[4] | out[5] |
out[6] | out[7];
/** 状态位含义
 * 0:      进位标志符
 * 1:      借位标志符
 * 2:      结果为 0 为 1
 * 3:      补码运算正溢出
 * 4:      补码运算负溢出
 */
always @(*) begin
    if (!rst)
        mem <= 8'b0;
    case(op) //使用双符号位的补码进行运算
        3'b000: begin //加法
            case(controllerforstate)
                2'b00: begin
                    {cout_add, out[7:0]} <= {in_a[7], in_a} + {in_b[7], in_b};
                    if({cout_add, out[7]} == 2'b01) // 正溢
                        mem[3] <= 1'b1;
                    if({cout_add, out[7]} == 2'b10) // 负溢
                        mem[4] <= 1'b1;
                    end
                    2'b01: {mem[0], out[7:0]} <= {in_a} + {in_b}; //记录进位
                    2'b10: begin out[7:0] <= {in_a} + {in_b}; //使用进位
                                out[7:0] <= out[7:0] + mem[0];
                            end
                    2'b11: {mem[0], out[7:0]} <=
{in_a} + {in_b} + {7'b0000000, mem[0]}; //记录并使用进位
                    endcase
                    mem[2] <= ~equal;
                    end
                3'b001: begin //减法

```



```

        {cout_sub,out[7:0]} <=
{in_a[7],in_a}+{~in_b[7],~in_b}+9'd1;
        if({cout_add,out[7]} == 2'b01) // 正溢
            mem[3] <= 1'b1;
        if({cout_add,out[7]} == 2'b10) //负溢
            mem[4] <= 1'b1;
        mem[2] <= ~equal;
    end
    3'b010: begin out[7:0] <= in_a*in_b;mem[2] <= ~equal;end
    3'b011: begin out[7:0] <= ~in_b;mem[2] <= ~equal;end
    3'b100: begin out[7:0] <= in_a^in_b;mem[2] <= ~equal;end
    3'b101: begin out[7:0] <= in_b+1;mem[2] <= ~equal;end
    3'b110: begin out[7:0] <= in_a / in_b;mem[2] <= ~equal;end
    3'b111: out[7:0] <= 8'bz;
endcase

end
endmodule

```

choose

二选一选择器

```

/**
 * RAM 地址输出
 */
module addrforram(
    input [7:0] addr_frommar,
    input [7:0] addr_frompc,    //ram 地址来源
    input in_rom_e,            //选择信号
    output wire [7:0] addr_ram //地址输出
);
    assign addr_ram = in_rom_e ? addr_frommar : addr_frompc;
endmodule

```

decode

译码器

```

/**
 * 译码器 三 - 八 当且仅当 open 为 1 的时候
 */
module decode(
    input [2:0] in,
    input open,
    output reg [7:0] out);
    always @(*)
        if (open)
            case(in)
                3'b000: out <= 8'b00000001;
                3'b001: out <= 8'b00000010;

```

```

        3'b010: out <= 8'b00000100;
        3'b011: out <= 8'b00001000;
        3'b100: out <= 8'b00010000;
        3'b101: out <= 8'b00100000;
        3'b110: out <= 8'b01000000;
        3'b111: out <= 8'b10000000;
    endcase
else
    out <= 8'b00000000;
endmodule

```

```

/**
 * 译码器 三 - 八 当且仅当 open 为 1 的时候
 */
module decode(
    input [2:0] in,
    input open,
    output reg [7:0] out);
always @(*)
    if (open)
        case(in)
            3'b000: out <= 8'b00000001;
            3'b001: out <= 8'b00000010;
            3'b010: out <= 8'b00000100;
            3'b011: out <= 8'b00001000;
            3'b100: out <= 8'b00010000;
            3'b101: out <= 8'b00100000;
            3'b110: out <= 8'b01000000;
            3'b111: out <= 8'b10000000;
        endcase
    else
        out <= 8'b00000000;
endmodule

```

```

/**
 * 8 — 8 译码器 将指令映射到微程序的起始地址
 */
module decode_7(
    input [7:0] in,
    output reg [7:0] out);
always @(*)
    case(in)
        8'b00110111: out <= 8'b10001010; // 00110111 T0 138 ALU_反_R1
        8'b00010111: out <= 8'b01010001; // 00010111 T0 81 STORE_PC_ANS
        8'b00110110: out <= 8'b01111100; // 00110110 T0 124 ALU_反_R0
        8'b00010000: out <= 8'b00110110; // 00010000 T0 54 STORE_MAR_R0
        8'b00010010: out <= 8'b00111100; // 00010010 T0 60 STORE_MAR_MAR
        8'b00010001: out <= 8'b00111001; // 00010001 T0 57 STORE_MAR_R1
        8'b00100111: out <= 8'b01101100; // 00100111 T0 108 R1-PC
    endcase
endmodule

```

```
        endcase
    endmodule
```

## CU

### 下地址生成方式

微程序的地址形成方式分为：

使用 op[20:13]的地址 jump、使用 MAR 的地址 jump、自增、使用 IR 经过 8-8 译码器([decode\\_7](#))译码产生的地址、

根据 ALU 运算结果是否为 0 jump(运算结果为 0，顺序执行，否则根据 MAR 地址跳转)、

根据 interrupt 跳转(若 interrupt==1，始终停留在 interrupt 的判断语句上，若==0，根据 PC 继续向下执行)

```
/**
 * 微程序设计指令的地址生成器
 */
module w_uPC(input clk,input rst,input ld,input [1:0] op,//下地址生成方式
             input [7:0] reg_sta,                //状态寄存器
             input interrupt,                    //中断
             input [7:0] in_upc,input [7:0] in_pc,output wire [7:0] out);

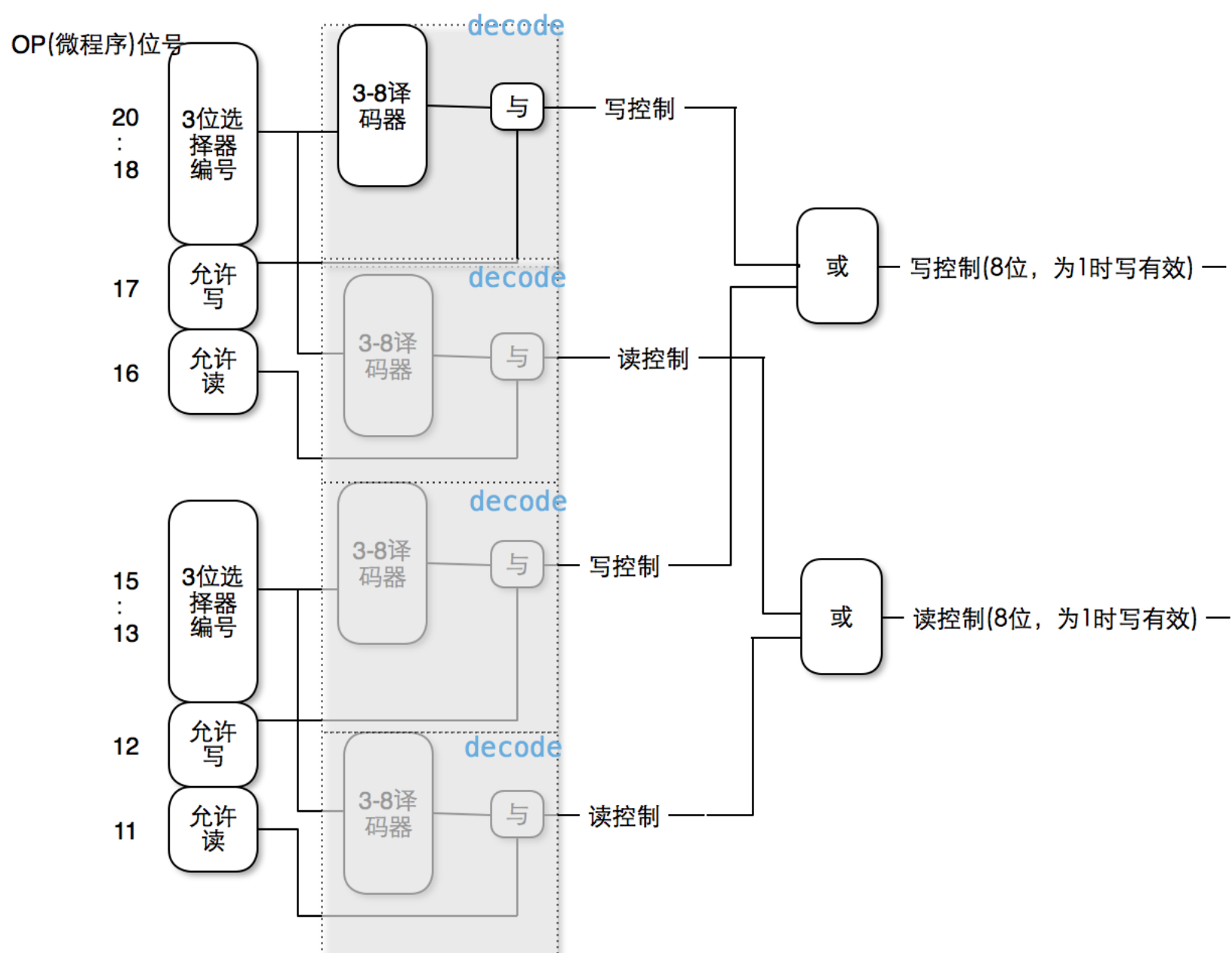
    /**
    *   rst = 0 清零
    *   rst = 1, ld = 0; 直传 out = in
    *       op[1:0] == 01 in_pc;
    *       op[1:0] == 10 in_upc;
    *   rst = 1, ld = 1,op[1:0] == 00; 计数
    *   可能的更改:reg 直接输出
    */
    wire [7:0] pctoupc;
    decode_7 decode(
        .in(in_pc),
        .out(pctoupc)
    );
    reg [7:0] value;
    always @(posedge clk) begin
        if (!rst)
            value <= 8'b0;
        else begin
            case({ld,op[1:0]})
                3'b100: value <= value + 1;
                3'b001: value <= pctoupc;//译码
                3'b010: value <= in_upc; //jump
                3'b000:begin//无条件跳转
                    value <= 8'b0000_0011;
                end
            endcase
        end
    end
end
```

```
3'b011: begin//==
        if(reg_sta[2]==1)//两个数字相等    继续向下运行
            value <= 8'b0;
        else
            value <= 8'b0000_0011;           //不相等跳转到 MAR 指定
        end
3'b101: begin
        if(interrupt)
            value <= 8'b0000_0111;
        else
            value <= 8'b0000_0000;
        end
    endcase
end
end
end
assign out = value;
endmodule
```

### 控制指令(寄存器的读写信号)生成

在指令组成中分为两个寄存器选择器，每一个均有读写控制信号(控制指定的寄存器的工作状态)

每个寄存器都有一个编号 0~7，可以通过 3 位二进制，经过 3-8 译码器得到。在分别和读写信号相与，最后两个写，读分别相或。



该控制器输出为两组 8 位的选择线分为写选中，读选中，每一组上最多有一个 1.

因为微程序的 20 ~ 11 与 jump 指令的跳转存在复用，所以读写控制信号只有 在下地址生成方式为自增，使用 IR 经过 8- - 8 译码器(decode\_7)译码产生下地址以及停机时 有效

```
/**
 * 寄存器读写控制器      纯组合逻辑
 */
module wrcontroller(
    input [2:0] reg_ch_l,      /*低位指定的寄存器选择线      (24 位微程序控制指令)*/
    input [2:0] reg_ch_h,      /*高位指定的寄存器选择线*/
    input [1:0] l_wr,          /*低位指定的寄存器  读写控制器*/
    input [1:0] h_wr,          /*高位指定的寄存器  读写控制器*/
    input [2:0] next,          /*根据下地址生成方式决定是否有效*/
    output wire [7:0] reg_ch_w, /*写寄存器选择线*/
    output wire [7:0] reg_ch_r /*读寄存器选择线*/);

    wire [7:0] reg_ch_l_w; /*低位指定的写寄存器选择线  (24 位微程序控制指令)*/
    wire [7:0] reg_ch_l_r; /*低位指定的读寄存器选择线*/
    wire [7:0] reg_ch_h_w; /*高位指定的写寄存器选择线*/
    wire [7:0] reg_ch_h_r; /*高位指定的读寄存器选择线*/

    decode decode_l_r(          /*低位指定的读寄存器*/
        .in      (reg_ch_l),
        .open    (l_wr[0]),
        .out      (reg_ch_l_r)
    );
    decode decode_l_w(          /*低位指定的写寄存器*/
        .in      (reg_ch_l),
        .open    (l_wr[1]),
        .out      (reg_ch_l_w)
    );
    decode decode_2_r(          /*高位指定的读寄存器*/
        .in      (reg_ch_h),
        .open    (h_wr[0]),
        .out      (reg_ch_h_r)
    );
    decode decode_2_w(          /*高位指定的写寄存器*/
        .in      (reg_ch_h),
        .open    (h_wr[1]),
        .out      (reg_ch_h_w)
    );
    // 寄存器状态控制器
    //          计数          通过 PC 转换          停机
    assign reg_ch_w = (next == 3'b100 || next == 3'b001 || 3'b111) ?
(reg_ch_l_w | reg_ch_h_w) : 8'b0; /*写线汇总*/
    assign reg_ch_r = (next == 3'b100 || next == 3'b001 || 3'b111) ?
(reg_ch_l_r | reg_ch_h_r) : 8'b0; /*读线汇总*/
endmodule
```

控制指令(IO 输入输出)

keyboard 连接开关输入

in\_keyboard 锁存开关的输入

led\_reg 输出锁存

```
// IO 控制区
reg [7:0] in_keyboard;
always@(negedge clk)
begin
    in_keyboard <= keyboard;
end
assign dbus = (in_rom_e[10]==1'b0 && in_rom_e[9] == 1'b1) ? in_keyboard :
8'bz; // IN

reg [7:0] led_reg;
assign led[23:16] = led_reg; // (1,1)
always@(negedge clk)
begin
    if(in_rom_e[10]==1'b1 && in_rom_e[9] == 1'b0)
        led_reg <= dbus;
end
```

其他控制信号

其他控制信号(MAR，MDR 的读写信号，外设的输入输出信号)由微指令直接给出，具体请参见指令设计部分。

微指令详解

微指令组成

ALU功能	寄存器选择	写读	寄存器选择	写读	IO	PC++	state	RAM地址输入	RAM读写控制	下地址生成方式
[23:21]	[20:18]	[17:16]	[15:13]	[12:11]	[10:9]	[8]	[7:6]	[5]	[4:3]	[2:0]

微指令具体内容

Data	ALU	寄存器选择	下地址生成方式
000	加法	PC	无条件跳转到 MAR
001	减法	IR	根据 IR 译码
010	乘法	MAR	跳转到指令的[20:13]

011	In_b 取反	MDR	条件跳转 ALU 结果为 0, 计数；非 0 跳转到 MAR
100	异或	ADD	计数
101	自增	ANS	没有中断, 计数；有中断轮询
110	除	R1	保留
111	高阻态	R0	停机

写读为两位为 1 时有效；举例 01 表示寄存器读

IO 两位 01 接收输入 10 输出

RAM 地址输入 0 PC 1MAR

RAM 写读控制 1 有效 0 无效

PC++ 1 有效 0 无效

State 加法工作模式 00 补码加法 01 记录进位 10 使用进位 11 记录和使用进位

微指令设计举例

取指周期

M(PC) → MDR  
MDR → IR ; PC + 1 → PC

译码跳转

op	指令	PC二进制	助记符											跳转地址
			ALU	寄存器	WR	寄存器	WR	Dispayl get N	PC++	state	RAM地址输入	RAMWR	下地址声称方式	
op	GETPC	10000000	H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	IR	W	N	1	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	译码	

跳转到 MAR 指定的指令上

MAR → PC  
M(PC) → MDR  
MDR → IR ; PC + 1 → PC

译码跳转

op	GETMAR	10000001	H	MAR	R	PC	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	IR	W	N	1	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	译码	

根据 MAR 寻址将数据存储到 R0

M(MAR) → MDR



MDR → R0

跳转取指周期

op	LOAD_MAR_R0	00000000	H	PC	N	PC	N	N	0	N	M	R	计数	
			H	MDR	R	R0	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

根据 PC 寻址将数据存储到 R0

M(PC) → MDR

MDR → MAR; PC + 1 → PC

M(MAR) → MDR

MDR → R0

跳转取指周期

op	LOAD_PC_R0	00000100	H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	MAR	W	N	1	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	M	R	计数	
			H	MDR	R	R0	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

载入立即数到 R0

M(PC) → MDR

MDR → R0; PC + 1 → PC

跳转取指周期

op	LOAD_NUM_R0	00001000	H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	R0	W	N	1	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

将 R0 存储到 MAR 指向的地址

R0 → MDR

MDR → M(MAR)

跳转到取指周期

op	STORE_MAR_R0	00010000	H	R0	R	MDR	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	M	W	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

将 R0 存储到 PC 指向的地址

M(PC) → MDR

MDR → MAR; PC + 1 → PC

R0 → MDR

MDR → M(MAR)

跳转到取指周期

op	STORE_PC_R0	00010100	H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	MAR	W	N	1	N	P	N	计数	
			H	R0	R	MDR	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	M	W	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

将 M(PC+1)存储到 M(PC)的地址

M(PC) → MDR



$MDR \rightarrow MAR; PC + 1 \rightarrow PC$   
 $M(PC) \rightarrow MDR$   
 $MDR \rightarrow M(MAR); PC + 1 \rightarrow PC$

跳转到取指周期

op	STORE_NUM_PC	00011001	H	PC	N	PC	N	N	0	N	P	R	计数	
			H	MDR	R	MAR	W	N	1	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	R	计数	
			H	PC	N	PC	N	N	1	N	M	W	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

寄存器间的数据转移

$R0 \rightarrow MAR$

跳转到取指周期

op	RO--MAR	00100000	H	R0	R	MAR	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

ADD\*R0 结果存到 ANS 中

$ADD * R0 \rightarrow ANS$

跳转到取指周期

op	ALU_乘_R0	00110100	乘法	R0	R	ANS	W	N	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

IO

设定 IO 状态     在设置相关寄存器的读写状态即可

op	IN_MAR	01110110	H	PC	N	MAR	W	IN	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC
op	OUT_R0	01111000	H	R0	R	PC	N	OUT	0	N	P	N	计数	
			H	PC	N	PC	N	N	0	N	P	N	jump	GETPC

HALT

op	停机	11110000	H	PC	N	PC	N	N	0	N	P	N	停机	
----	----	----------	---	----	---	----	---	---	---	---	---	---	----	--

指令(8 位)设计

OP	寻址方式	目的寄存器
0000(LOAD)	00(根据 MAR 寻址)	00(R0) 01(R1) 10(ADD)  11(MAR)
	01(根据 PC 寻址)	
	10(立即数)	
0001(STORE)	00(根据 MAR 寻址)	
	01(根据 PC 寻址)	

	10(立即数)	00(R0) 01(R1) 10(MAR)  11(ANS)	
	源寄存器	目的寄存器	
0010(寄存器间数据转移)	00(R0)	00(MAR) 01(ADD) 10(R1) 11(PC)	
	01(R1)	00(MAR) 01(ADD) 10(R0) 11(PC)	
	10(ANS)	00(MAR) 01(R0) 10(R1) 11(PC)	
	运算功能		数据源
0011(ALU 运算)	000(加)001(减)010(乘)011(反)100(异  或)101(自增)110(除)		0(R0) 1(R1)
0100(记录进位)0101(使用进位)0110(记录并使 用进位)	000(加法)		0(R0) 1(R1)
	IO 方式	相关寄存器	
0111(IO)	01(IN) 10(OUT)	00(R0)01(R1)10(MAR)11(ADD)	
		00(R0)01(R1)10(ANS)11(MAR)	
HALT			
11110000			

助记符转换成 Verilog(decode\_7 模块)映射关系代码、微程序(ROM)内存内容与指令和指令名称的映射文件



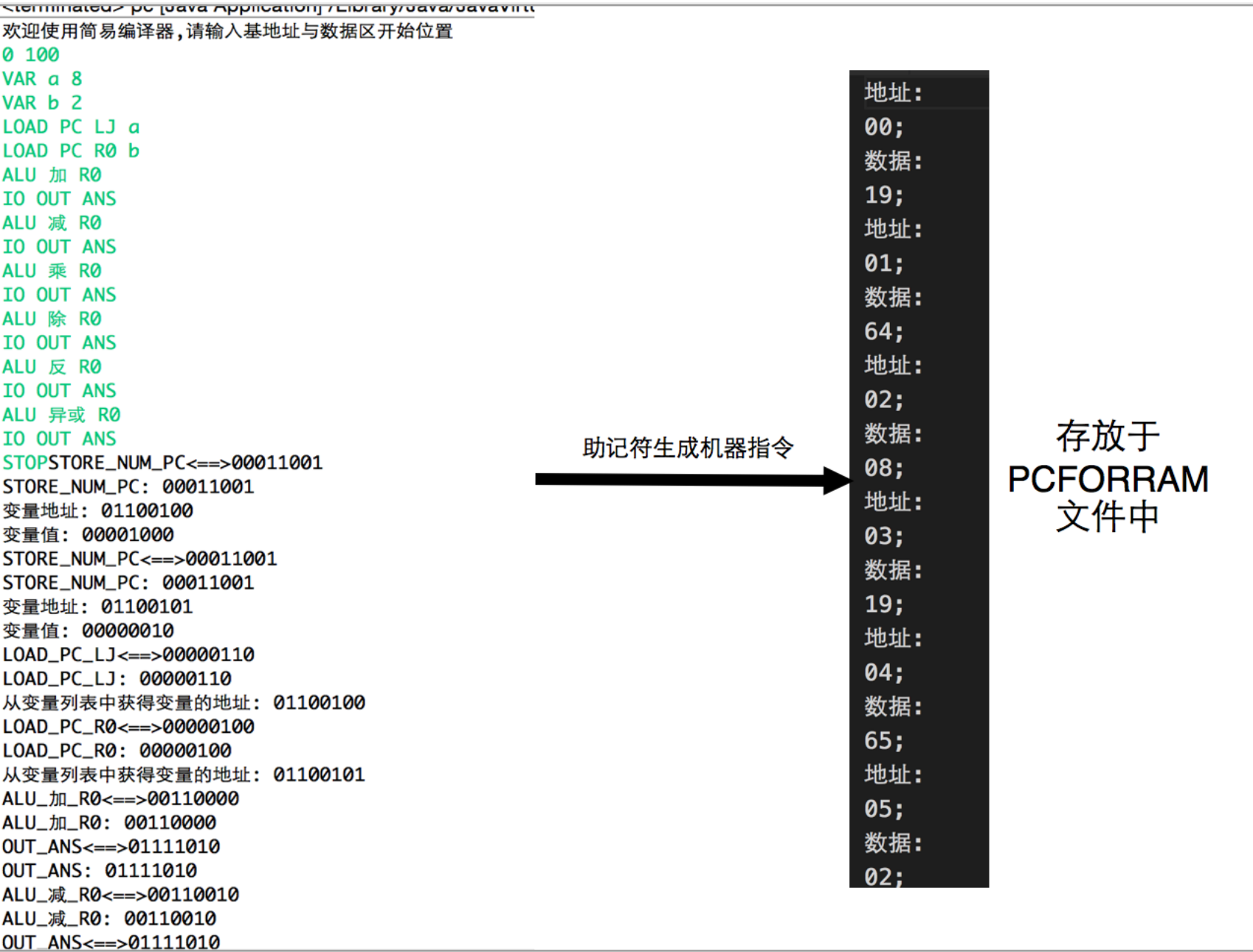
类汇编语言转换成机器码(存放于 RAM 的指令)简易编译器

简介

简易编译器首先需要知道指令区域与数据区域的其实地址

然后根据输入进来的助记符进行转换，生成的机器指令存储在 PCFORRAM 中

实例：














指令详解

LOAD	<寻址方式> <目标寄存器> [数值/地址]	根据 MAR 寻址不需要最后的[数值/地址]
STORE	<寻址方式> <源寄存器> [数值/地址]	根据 MAR 寻址不需要最后的[数值/地址]
TR	<源寄存器> <目标寄存器>	寄存器间的数据转移
ALU	<功能选择> <源寄存器>	ADD 与源寄存器或者源寄存器单独运算，结果存放在 ANS 中
IO	<IN/OUT> <相关寄存器>	IO 操作中的 IN，在读入开关输入之前会加 BP 操作
JUMP E		这里比较别扭 若 ALU 计算结果==0 则顺序执行，否则跳转到 MAR 给定的地址
JUMP N		无条件跳转
PUSH	<源寄存器>	将源寄存器的值存储到栈中,栈从 RAM 的最高地址向下存储
PULL	<目的寄存器>	将栈顶的值存储到目的寄存器中
VAR	<变量名称> <变量值>	将变量值按照顺序存放在数据区。同时 java 程序会记录该位置的地址。
PARA	<变量名称> <变量值>	不产生机器指令，只是让 java 程序记录名称与值的对应关系，方便后面的书写
BP		设置断点，若 interrupt==0 会顺序向下执行，中断不为 0 会一直停留在此处

PROGRAMMER <程序名>      不产生机器指令，记录下一条指令的地址

目录结构：

名称	^	修改日期	大小	种类
 微指令		2017年10月31日 下午4:56	47 KB	Office 电子表
 微指令ForMDR	实验三微指令	17年10月31日 下午3:52	5 KB	文本编辑 文
 助记符--PC	实验三指令名称与指令二进制	17年10月31日 下午3:52	1 KB	文本编辑 文
▶  bin		2017年11月20日 上午8:24	--	文件夹
 decodeForVerilog	实验三verilog中的指令与微指令地之间的映射关系	:52	4 KB	文本编辑 文
 excel	实验三微指令助记符	2017年10月31日 下午3:52	10 KB	Visual...de
 hard_verilog	实验四的指令与时钟节拍共同译码关系图	17年11月20日 上午8:24	5 KB	文本编辑 文
 hardexec	实验四指令助记符	2017年11月17日 下午8:16	2 KB	文本编辑 文
 PCFORRAM	实验三生成的机器指令	2017年10月31日 下午4:51	624 字节	文本编辑 文
▶  src		今天 下午3:45	--	文件夹
 upc助记符	实验三助记符符号映射到二进制编码	2017年10月24日 下午5:37	299 字节	文本编辑 文

测试

测试一

测试 IO，中断以及停机指令的工作是否正常

助记符

0 100

IO IN R0

IO OUT R0

STOP

生成的指令

地址:	01;	数据:	f0;
00;	数据:	78;	地址:
数据:	74;	地址:	04;
82;	地址:	03;	数据:
地址:	02;	数据:	f0;

测试二

接收用户输入的两个数字（补码的形式）， 若两个数字的和为 0，则退出程序；否则一致循环接收用户的两次输入

助记符

0 100

PROGRAMMER p1

IO IN LJ

IO IN R0

ALU 加 R0

LOAD NUM MAR p1

JUMP E

LOAD NUM R1 170  
IO OUT R1  
STOP

生成的指令

地址:	02;	数据:	00;	地址:	0b;
00;	数据:	30;	地址:	09;	数据:
数据:	82;	地址:	07;	数据:	f0;
82;	地址:	05;	数据:	aa;	地址:
地址:	03;	数据:	83;	地址:	0c;
01;	数据:	0b;	地址:	0a;	数据:
数据:	74;	地址:	08;	数据:	f0;
77;	地址:	06;	数据:	79;	
地址:	04;	数据:	09;	地址:	

实验三

运算器测试

助记符  
0 100  
VAR a 8  
VAR b 2  
LOAD PC LJ a  
LOAD PC R0 b  
ALU 加 R0  
IO OUT ANS  
ALU 减 R0  
IO OUT ANS  
ALU 乘 R0  
IO OUT ANS  
ALU 除 R0  
IO OUT ANS  
ALU 反 R0  
IO OUT ANS  
ALU 异或 R0  
IO OUT ANS  
STOP

生成的指令

地址:	数据:	地址:	数据:	地址:	数据:
00;	08;	05;	64;	0a;	32;
数据:	地址:	数据:	地址:	数据:	地址:
19;	03;	02;	08;	30;	0d;
地址:	数据:	地址:	数据:	地址:	数据:
01;	19;	06;	04;	0b;	7a;
数据:	地址:	数据:	地址:	数据:	地址:
64;	04;	06;	09;	7a;	0e;
地址:	数据:	地址:	数据:	地址:	数据:
02;	65;	07;	65;	0c;	34;

地址:	数据:	地址:	数据:	地址:	数据:
0f;	3c;	12;	7a;	15;	f0;
数据:	地址:	数据:	地址:	数据:	地址:
7a;	11;	36;	14;	7a;	17;
地址:	数据:	地址:	数据:	地址:	数据:
10;	7a;	13;	38;	16;	f0;

测试四

计算 1 到 4 的和

```
助记符
0 100
VAR i 4
VAR sum 0
LOAD PC R1 i
LOAD PC R0 sum
PROGRAMMER p
TR R0 LJ
ALU 加 R1
TR ANS R0
LOAD NUM LJ -1
ALU 加 R1
TR ANS R1
LOAD NUM MAR p
IO OUT R0
JUMP E
IO OUT R0
STORE PC R0 102
STOP
```

生成的指令

地址:	数据:	地址:	数据:	地址:	数据:
00;	65;	09;	0a;	12;	14;
数据:	地址:	数据:	地址:	数据:	地址:
19;	05;	65;	0e;	0a;	17;
地址:	数据:	地址:	数据:	地址:	数据:
01;	00;	0a;	ff;	13;	66;
数据:	地址:	数据:	地址:	数据:	地址:
64;	06;	21;	0f;	78;	18;
地址:	数据:	地址:	数据:	地址:	数据:
02;	05;	0b;	31;	14;	f0;
数据:	地址:	数据:	地址:	数据:	地址:
04;	07;	31;	10;	83;	19;
地址:	数据:	地址:	数据:	地址:	数据:
03;	64;	0c;	29;	15;	f0;
数据:	地址:	数据:	地址:	数据:	
19;	08;	2a;	11;	78;	
地址:	数据:	地址:	数据:	地址:	
04;	04;	0d;	0b;	16;	

实验感悟

- 1、实验中存在一个复用的管脚需要特别注意。
- 2、实验过程中， 写入 ROM 或者 RAM 的数据一次不能太多， 数据太多会导致后面的数据写不进去， 推荐一次对多写 260 个数据单元。
- 3、在微指令设计过程中,注意复用位的使用， 避免产生错误的控制信号。例如在进行跳转时， 跳转的地址占用了原来的寄存器的选择与读写控制信号位， 所以在控制信号(寄存器读写控制信号)生成时， 需要在 jump 时， 让控制信号无效。这同时意味着寄存间没有数据转移,不能进行 ALU 的运算， 因而此时 PC 的加 1 不能由 ALU 来实现。
- 4、在写入数据到 RAM 中， 会存在数据写进去和读出来的不一致的现象， 此种问题可能的原因是： 写入 PFGA 的电路图让 RAM 处于写操作， 因而 RAM 中的某位一致处于写状态， 导致读出来的与我们写进去的不一致的假象。处理方法是， 设置一个开关控制微指令是否有效， 若开关==0 时， 则让 RAM 处于一种安全的状态(读写均无效)。
- 5、本实验中的所有寄存器的赋值采用的都是阻塞赋值的方法， 该赋值的特点是， 寄存器的结果先产生影响， 然后寄存器的值被刷新掉， 这个对时序逻辑的理解十分重要。



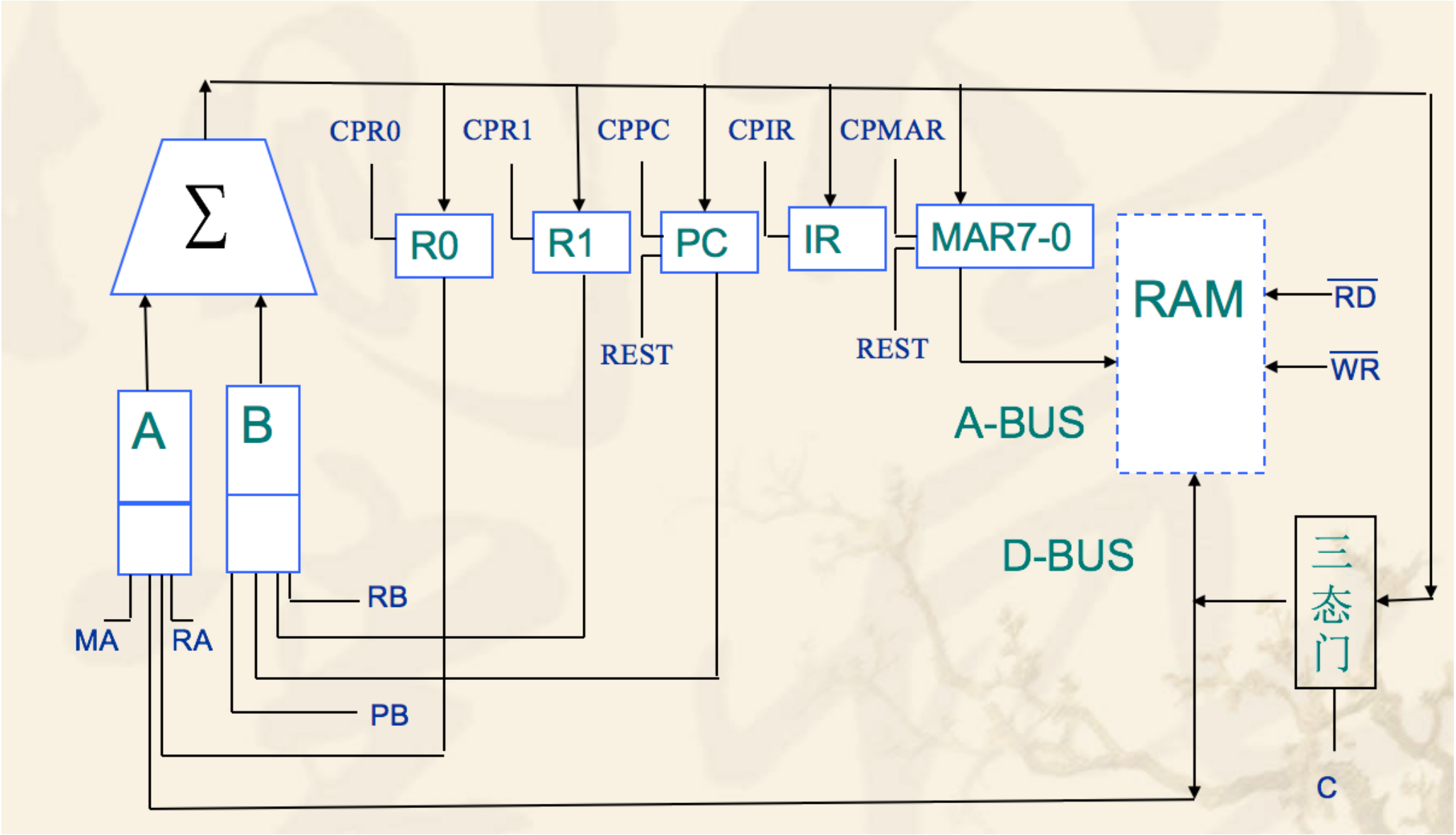
6、出现莫名其妙的错误时，可以尝试重启实验箱，或者更换。

拓展空间

- 1、在寄存器的控制指令生成过程中，可以减少 3-8 译码器的使用，减少资源消耗
- 2、在 IR 中的数据向微指令地址译码的过程中，可以通过层级译码的方式(先使用 OP 区的指令进行译码，再结合寻址方式等进行两层译码方案)来减少译码电路的复杂度。由于时间原因，没有进行尝试。
- 3、在 ALU 内部的状态寄存器中，可以添加更为完善的溢出处理子程序
- 4、在两个 Java 程序中都存在健壮性的问题，对于非开发者而言，不是很友好，有很大的优化空间。

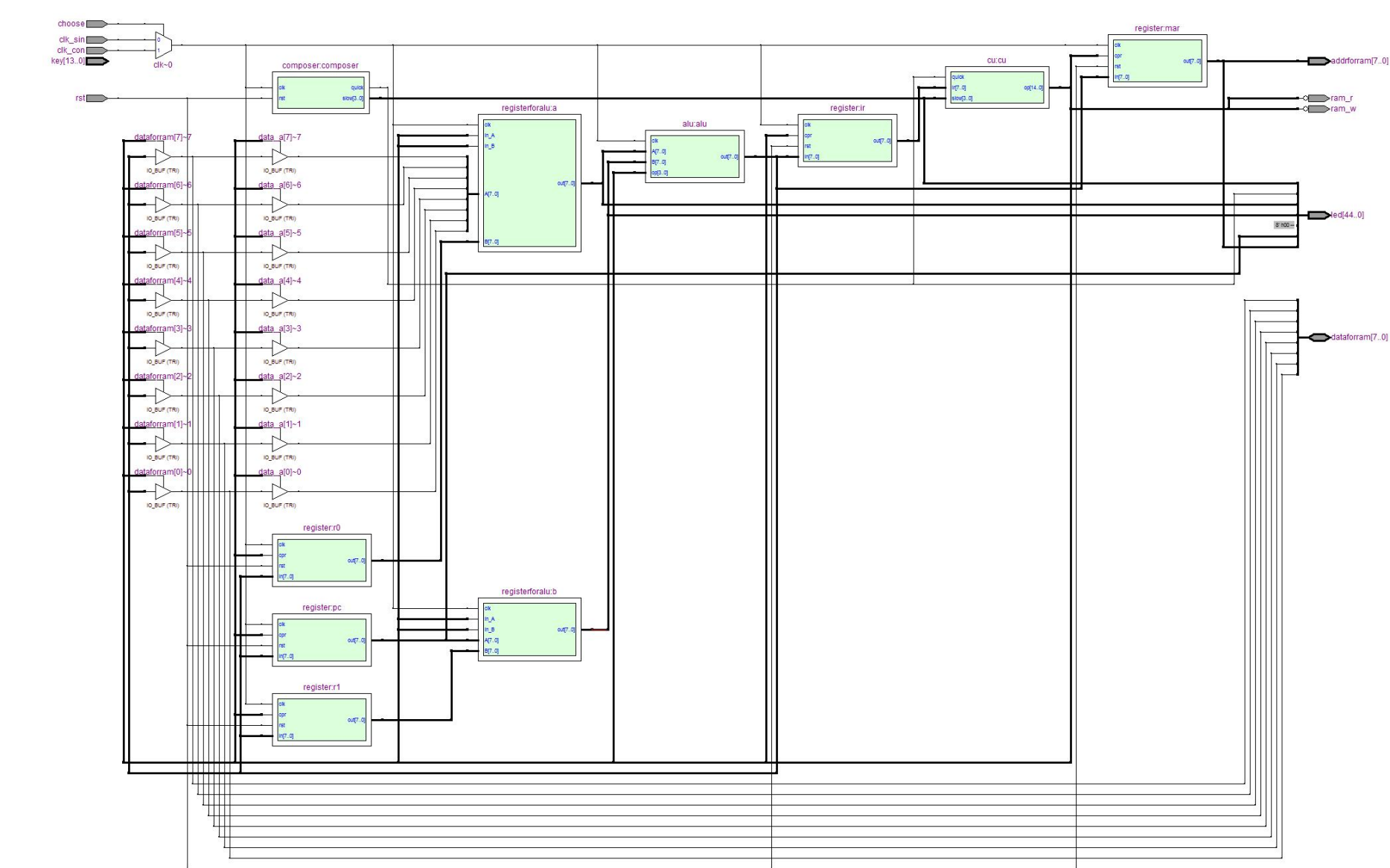
实验四

模式图



在本实验中使用的是老师提供的模型机。其中 A、B 是二选一选择器。寄存器在接受到 CP 命令时锁存数据，存储的数据一直输出到对外连接线上。

物理连接图



组件详解

节拍发生器

其主要思路是一个一个三位的二进制数字 count\_t ， 在源时钟源的每个上升沿， count\_t 加 1

正好形成 000 001 010 011 100 101 110 111 循环的变化

其中最高位代表大的时钟周期 quick(w 为 1 时为高电平， 为 0 时为低电平)

低两位控制 slow 脉冲的变化 slow 脉冲有 4 个， 低两位的循环周期也是 4， 每当 count\_t == 脉冲的编号， 该脉冲为高电平， 否则为低电平。

```
module composer(  
    input clk,  
    input rst,  
    output wire [3:0] slow,  
    output wire quick  
);  
    wire t1,t2,t3,t4,w1;  
    reg[2:0] count_t;  
    always@(posedge clk)  
    begin
```

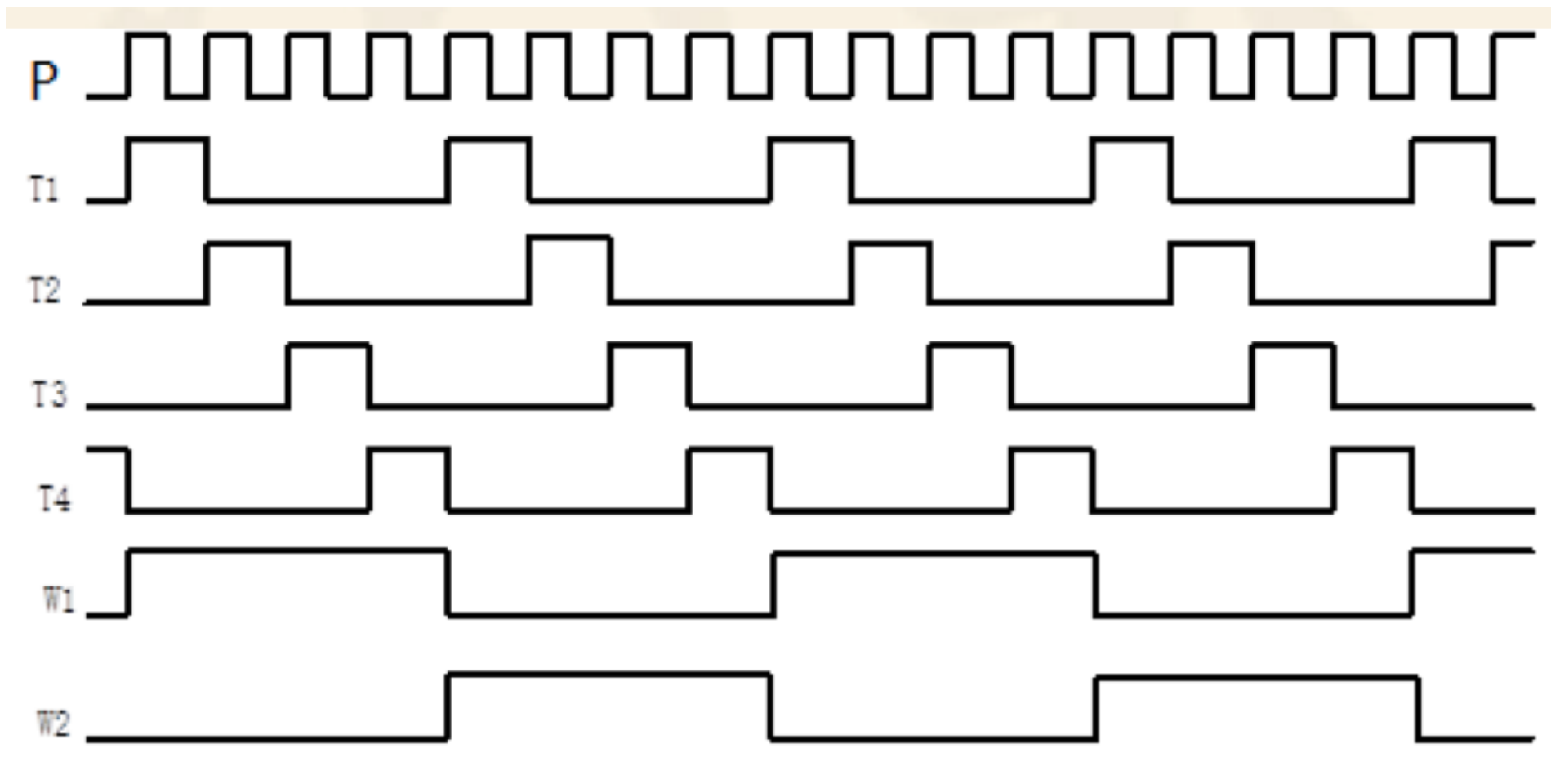
```

    if(!rst)begin
        count_t <= 3'b000;
    end
    else
        count_t <= count_t + 1;
    end
    assign t1 = (count_t[1:0]==2'b00) ? 1'b1 : 1'b0;
    assign t2 = (count_t[1:0]==2'b01) ? 1'b1 : 1'b0;
    assign t3 = (count_t[1:0]==2'b10) ? 1'b1 : 1'b0;
    assign t4 = (count_t[1:0]==2'b11) ? 1'b1 : 1'b0;
    assign w1 = (count_t[2]==1'b0) ? 1'b1 : 1'b0;
    assign slow = {t1,t2,t3,t4};
    assign quick = w1;

```

endmodule

正好形成如下的时序电路图



## ALU

此处的 ALU 的设计与实验三的基本一致，在此不再赘述

```

module alu(
    input clk,
    input [7:0] A,
    input [7:0] B,
    input [3:0] op,
    output reg [7:0] out
);
    always@(*)
    begin
        case(op)
            4'b0000: out <= A;
            4'b0001: out <= B;
            4'b0010: out <= B + 1;

```

```

        4'b0011: out <= A+B;
        4'b0100: out <= A-B;
        4'b0101: out <= A^B;
        4'b0110: out <= A + 1;
        4'b0111: out <= A - 1;
        4'b1000: out <= A&B;
        4'b1001: out <= A|B;
        4'b1010: out <= ~A;
        4'b1011: out <= A*B;
        4'b1100: out <= A/B;
        4'b1101: out <= B - 1;
        4'b1111: out <= 8'bz;
    endcase
end
endmodule

```

## CU

由于 verilog 语言存在 case 语句，可以实现自动的译码处理，并且存在优化，所以这里的 CU 的设计在这里比较暴力，直接将 8 位指令加上 5 个节拍脉冲映射成控制信号。

```

module cu( input [7:0] ir,
           input [3:0] slow,
           input quick,
           output reg[14:0] op);

/**
 * 14:11 ALU
 * 10:7 A B
 * 6:2 register
 * 1:0 W R
 */
always@(slow,quick)begin
    if(quick == 1'b1)begin
        case(slow)
            4'b1000: op <= 15'b000100100000100;
            4'b0100: op <= 15'b001000100010000;
            4'b0010: op <= 15'b000010000001001;
            4'b0001: op <= 15'b111100000000000;

            endcase
        end else begin
            case(ir)
                8'b00000110: begin
                    case(slow)
                        4'b1000: op <= 15'b000100100000100;
                        4'b0100: op <= 15'b000010000100001;
                        4'b0010: op <= 15'b001000100010000;
                        4'b0001: op <= 15'b111100000000000;

                    endcase
                end
                8'b00000010: begin
                    case(slow)

```

```

                4'b1000: op <= 15'b000100100000100;
                4'b0100: op <= 15'b000010001000001;
                4'b0010: op <= 15'b001000100010000;
                4'b0001: op <= 15'b111100000000000;
            endcase
        end
        `
        `
        `
        `

    end
8'b11111111: begin
    case(slow)
        4'b1000: op <= 15'b111100000000000;
        4'b0100: op <= 15'b110100100010000;
        4'b0010: op <= 15'b111100000000000;
        4'b0001: op <= 15'b111100000000000;
    endcase
end
endcase
end
end
endmodule

```

## 对 RAM 的读写

data\_a 连接到了 A 选择器，data\_forram 连接到 RAM 的数据线上 inregister 连接到了 ALU 的输出

op[0]==1 时为 RAM 的读信号

op[1]==1 时为 RAM 的写信号

当 RAM 为写状态，相当于 RAM 的数据线只连接到了 ALU 的输出上

当 RAM 为读状态，相当于 RAM 的数据线只连接到了 A 选择器的输入上

```

assign data_a = op[0] ? dataforram : 8'bz;

assign dataforram = op[1] ? inregister : 8'bz;

```

## 控制指令的设计

ALU 控制位为 4 位

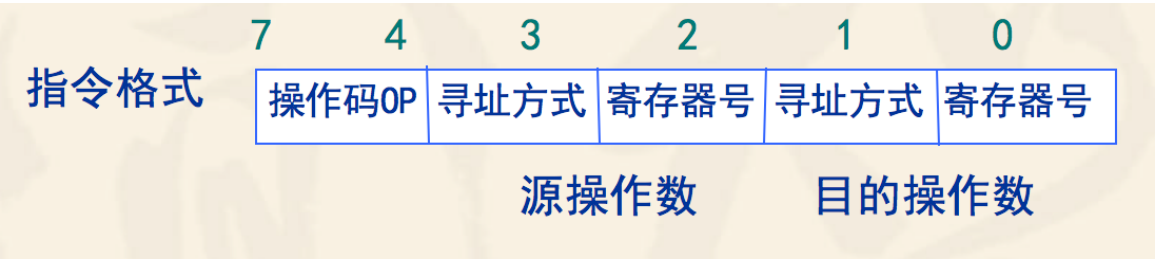
A、 B 选择器有 4 位控制位

寄存器 5 位控制位

RAM 读写控制位 2 位

					14:11	10	9	8	7	6	5	4	3	2	1	0
op		时钟周期W1		时钟周期T	ALU	MA	R0A	PB	R1B	R0	R1	PC	IR	MAR	W	R
取指	F0	1	11110000	1000	B	0	0	1	0	0	0	0	0	1	0	0
				0100	B++	0	0	1	0	0	0	1	0	0	0	0
				0010	A	1	0	0	0	0	0	0	1	0	0	1
				0001	N	0	0	0	0	0	0	0	0	0	0	0
M(pc)->R1&&PC++	06	0	00000110	1000	B	0	0	1	0	0	0	0	0	1	0	0
				0100	A	1	0	0	0	0	1	0	0	0	0	1
				0010	B++	0	0	1	0	0	0	1	0	0	0	0
				0001	N	0	0	0	0	0	0	0	0	0	0	0

指令的设计符合下列规则



指令的设计比较直观， 再次便不再解释 请参见 **硬布线**文件夹下的**指令.xlsx**

java 程序

hardupc 将上述的文件转换成 CU 的 case 语句， 和实验三的类似， 在此不再赘述

测试

测试运算计算器

指令顺序为

02	02	06	04	14	0D	20	02	02	06	04	14	94	21	.....	FF	FF
M(pc)->R0	立即数	M(pc)->R1	立即数	ALU 功能选择	R1->M(M(PC))	地址										停机

遇到的问题

- 1、逻辑混乱， 将选择器的控制位和数据位搞混了， 导致查了很长的错误。

拓展空间

- 1 添加优化器， 当当前的周期的有效指令执行结束， 马上将节拍发生器重置， 立即进入取指周期
- 2 在 CU 处添加层级译码， 降低译码的代价