《计算机组成与设计》课程设计报告

实验课程 计算机组成原理课程设计

课设题目 模型计算机设计

学生班级 学堂计机 19

学生姓名 王泽宇

学生学号 201905130196

完成日期 2021 年 5 月 18 日

二〇二一年六月二日

目录

1 指令集架构

1.1 介绍

本次实验所涉及的模型机使用了一种采取精简指令集的架构，仿照 MIPS16 设计并进行简化。本实验使用的指令集架构包含 8 个通用寄存器，数据总线和地址总线宽度均为 16 位。支持乘法运算，实现了无符号数原码阵列乘法器。实现了常用基本功能，包括加减乘和逻辑运算、寄存器置数、多种寻址方式的 RAM 读写、无条件跳转、有条件跳转、栈、子程序等。同时，0 号寄存器常置为 0，可以辅助实现多种功能。采用 16 位固定字长指令简化实现。

1.2 实现指令组

指令采用 16 位固定字长。其中，前五位为操作数。所有计算操作的操作数全部来自寄存器，各种寻址方式的实现也大都依赖于寄存器。寄存器 0 为常 0 寄存器，里面存放的数永远是 0，这为用户实现立即数寻址、像寄存器送立即数等提供了便利，也便于用精简指令集来实现众多复杂的操作。

图1展现了实现指令组。其中 RS1、RS2、RD 分别代表不同的寄存器编号，# 开头代表立即数，# 后面指定了立即数位数。

2 总体结构

注意，后续所有代码均有 RTL Viewer 对应，请查看附件 6：原理图。

2.1 数据通路

图2展现了数据通路的主体结构。其中的控制信号连向控制单元，使用了微程序和硬布线两种不同的方式实现（后续介绍），本章介绍数据通路部分。

ALU 是算术逻辑运算单元，根据进位信号和控制信号，将 A 选择器和 B 选择器的输出作为操作数，进行算术运算、按位逻辑运算、比较运算等多种运算。为了支持有条件跳转，ALU 中还包含一个跳转标志寄存器，用于存储跳转标志。

A 选择器、B 选择器分别接受 8 个输入信号，根据控制单元发出的选择信号选择一个作为选择器的输出。

R0 R7 表示八个通用寄存器，每个通用寄存器有各自的脉冲信号用于接受来自 ALU 输出的赋值。八个寄存器总共有两个输出接口：寄存器选择器 A 和寄存器选择器 B，根据控制单元发出的控制信号选择以某个寄存器的输出作为选择器的输出。

PC 是程序计数寄存器，用于记录下一条指令的地址。IR 是指令寄存器，用于存储指令。指令可能包含 3 种立即数，这三种立即数对应的指令位回连向 AB 选择器。MAR

操作	指令格式														
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	操作码															
ADD RS1 RS2 RD 00	0	0	0	0	1	RS1			RS2			RD			00
将寄存器R1、R2相加，放到RD																
MOVIH RS1 #8-bit	0	0	0	1	0	RS1			#8-bit							
以高8位为#送到RS1																
MOVIL RS1 #8-bit	0	0	0	1	1	RS1			#8-bit							
以低8位为#送到RS1																
MOV RS1 RS2 000 00	0	0	1	0	0	RS1			RS2			00000				
将RS1赋值到RS2																
SUB RS1 RS2 RD 00	0	0	1	0	1	RS1			RS2			RD			00
寄存器RS1的值减去TS2的值，放到RD																
LDIDR RS1 RS2 #5bit	0	0	1	1	0	RS1			RS2			#5bit				
寄存器RS1+#5bit为地址，读取到RS2中																
STIDR RS1 RS2 #5bit	0	0	1	1	1	RS1			RS2			#5bit				
寄存器RS1+#5bit为地址，把RS2的内容写到RAM																
LDIDX RS1 RS2 RS3	0	1	0	0	0	RS1			RS2			RS3			00
寄存器RS1+寄存器RS2作为地址，读取到RS3中																
STIDX RS1 RS2 RS3	0	1	0	0	1	RS1			RS2			RS3			00
寄存器RS1+寄存器RS2作为地址，把RS3内容写到RAM																
AND RS1 RS2 RS3	0	1	0	1	0	RS1			RS2			RS3			00
寄存器RS1 AND 寄存器RS2送到寄存器RS3																
OR RS1 RS2 RS3	0	1	0	1	1	RS1			RS2			RS3			00
寄存器RS1 OR 寄存器RS2送到寄存器RS3																
XOR RS1 RS2 RS3	0	1	1	0	0	RS1			RS2			RS3			00
寄存器RS1 XOR 寄存器RS2送到寄存器RS3																
NOT RS1 RS2	0	1	1	0	1	RS1			RS2			00000				
寄存器NOT RS1送到寄存器RS2																
SHIFTL RS1 RS2	0	1	1	1	0	RS1			RS2			00000				
寄存器RS1的值左移1位送到寄存器RS2																
SHIFTR RS1 RS2	0	1	1	1	1	RS1			RS2			00000				
寄存器RS1的值右移1位送到寄存器RS2																
MULTI RS1 RS2 RS3	1	0	0	0	0	RS1			RS2			R3			00
寄存器RS1*寄存器RS2结果送到寄存器RS3																
JMP #11-bit	1	0	0	0	1	#11-bit										
跳转到当前PC+#11bit																
JMPI RS1 #8-bit	1	0	0	1	0	RS1			#8-bit							
跳转到RS1+#8bit																
JGEO RS1 RS2 #5-bit	1	0	0	1	1	RS1			RS2			#5bit				
如果RS1>=RS2，跳转到当前PC+#5bit																
JLEO RS1 RS2 #5-bit	1	0	1	0	0	RS1			RS2			#5bit				
如果RS1<=RS2，跳转到当前PC+#5bit																
JEO RS1 RS2 #5-bit	1	0	1	0	1	RS1			RS2			#5bit				
如果RS1==RS2，跳转到当前PC+#5-bit																
NOP	1	0	1	1	0	00000000000										
空一个机器周期																
ADDI RS1 RS2 #5-bit	1	0	1	1	1	RS1			RS2			#5bit				
RS1+#5-bit送到RS2																
SUBI RS1 RS2 #5-bit	1	1	0	0	0	RS1			RS2			#5bit				
RS1-#5-bit送到RS2																
PUSH RS1	1	1	0	0	1	RS1			00000000							
将RS1的值压入栈																
POP RS1	1	1	0	1	0	RS1			00000000							
将RS1的值弹出栈																
MOVESP RS1	1	1	0	1	1	RS1			00000000							
将栈帧寄存器置为RS1的值																
CALL RS1	1	1	1	0	0	RS1			00000000							
跳转到子程序RS1（地址），将返回地址压入栈																
JAL #11-bit	1	1	1	0	1	#11-bit										
跳转到子程序PC+#11-BIT（地址），将返回地址压入栈																
RET	1	1	1	1	0	00000000000										
返回主程序，栈顶为地址																
HLT	1	1	1	1	1	00000000000										
停机																

图 1: 实现指令

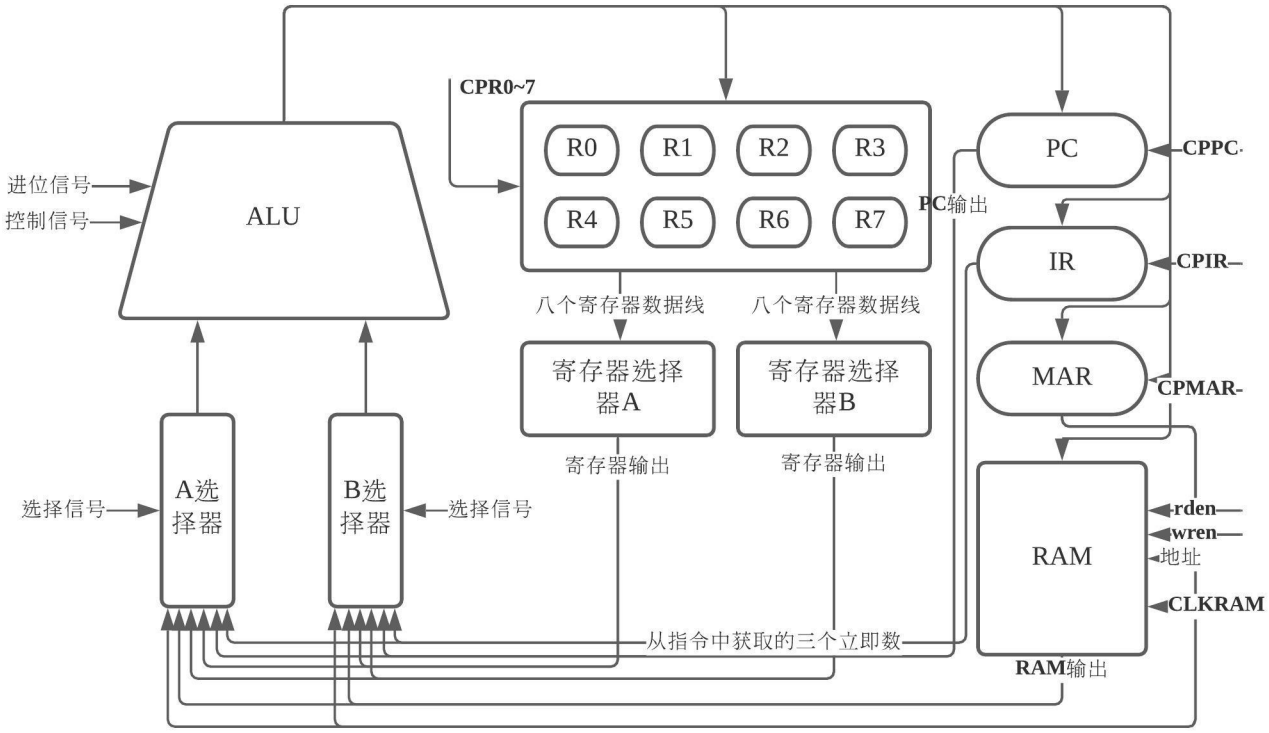

图 2: 数据通路

是存储器地址寄存器，存储将要访问的地址。

RAM 是随机访问存储器，存储程序和数据。地址来自 MAR 的输出，输入来自 ALU 的输出。

以上结构采用 verilog 实现。下面将给出每个部件的 verilog 相关代码及其 RTL Viewer。

2.2 算数逻辑运算单元 ALU

2.2.1 ALU 总体结构

```
1 module ALU(
2         input[4:0] S ,
3         input[15:0] A ,
4         input[15:0] B ,
5         input CN ,
6         input jumpTagCLK ,
7         output[15:0] ANS ,
```

```
8           output CNOUT ,
9           output reg jumpTag
10  );
11
12  reg[15:0] F;
13  reg CN4;
14  assign ANS = F;
15  assign CNOUT = CN4;
16
17
18  //A左移右移运算
19  reg shift_direction;
20  wire[15:0] shiftA_out;
21  shift16(
22          .A(A) ,
23          .P(shift_direction) ,
24          .B(shiftA_out)
25  );
26
27  //B左移右移运算
28  wire[15:0] shiftB_out;
29  shift16(
30          .A(B) ,
31          .P(shift_direction) ,
32          .B(shiftB_out)
33  );
34
35
36  //求补器
37  wire[15:0] A_complement, B_complement,
38                multi_out_complement, multi_out;
39  complement16 complementA(
40          .A(A) ,
41          .B(A_complement)
42  );
43  complement16 complementB(
44          .A(B) ,
```

```
45          .B(B_complement)
46  );
47  complement16 complement_multi_out(
48          .A(multi_out) ,
49          .B(multi_out_complement)
50  );
51
52  //阵列乘法器
53  multiplication16(
54          .A(A) ,
55          .B(B) ,
56          .C(multi_out)
57  );
58
59  always @(posedge jumpTagCLK) begin
60          case(S[4:0])
61                  5'b11001: begin
62                          if(A >= B) begin
63                                  jumpTag = 1;
64                          end
65                          else begin
66                                  jumpTag = 0;
67                          end
68                  end
69                  5'b11010: begin
70                          if(A <= B) begin
71                                  jumpTag = 1;
72                          end
73                          else begin
74                                  jumpTag = 0;
75                          end
76                  end
77                  5'b11011: begin
78                          if(A == B) begin
79                                  jumpTag = 1;
80                          end
81                          else begin
```

```
82                                      jumpTag = 0;
83                              end
84                      end
85
86              endcase
87  end
88
89
90  always @(A, B, S, CN, shiftA_out, shiftB_out, multi_out) begin
91              case(S[4:0])
92                      5'b00000: begin
93                              F = A;
94                              CN4 = 0;
95                      end
96                      5'b00001: begin
97                              F = B;
98                              CN4 = 0;
99                      end
100                     5'b00010: begin
101                             {CN4, F} = A + B;
102                     end
103                     5'b00011: begin
104                             {CN4, F} = A - B;
105                     end
106                     5'b00100: begin
107                             {CN4, F} = A + 1;
108                     end
109                     5'b00101: begin
110                             {CN4, F} = A - 1;
111                     end
112                     5'b00110: begin
113                             {CN4, F} = B + 1;
114                     end
115                     5'b00111: begin
116                             {CN4, F} = B - 1;
117                     end
118                     5'b01000: begin
```

```
119                         {CN4, F} = A + B + 1;
120                 end
121                 5'b01001: begin
122                         F = multi_out;
123                 end
124                 5'b01010: begin
125                         shift_direction = 0;
126                         F = shiftA_out;
127                 end
128                 5'b01011: begin
129                         shift_direction = 1;
130                         F = shiftA_out;
131                 end
132                 5'b01100: begin
133                         shift_direction = 0;
134                         F = shiftB_out;
135                 end
136                 5'b01101: begin
137                         shift_direction = 1;
138                         F = shiftB_out;
139                 end
140                 5'b01110: begin
141                         F = A & B;
142                 end
143                 5'b01111: begin
144                         F = A | B;
145                 end
146                 5'b10000: begin
147                         F = A ^ B;
148                 end
149                 5'b10001: begin
150                         F = ~A;
151                 end
152                 5'b10010: begin
153                         F = ~B;
154                 end
155                 5'b10011: begin
```

```
156                         F = ({1'b0, A} > {1'b0, B});
157                 end
158                 5'b10100: begin
159                         F = ({1'b0, A} < {1'b0, B});
160                 end
161                 5'b10101: begin
162                         F = ({1'b0, A} == {1'b0, B});
163                 end
164                 5'b10110: begin
165                         F = ({1'b0, A} != {1'b0, B}) ;
166                 end
167                 5'b10111: begin
168                         F[15:8] = A[7:0];
169                         F[7:0] = B[15:8];
170                 end
171                 5'b11000: begin
172                         F[15:8] = A[15:8];
173                         F[7:0] = B[7:0];
174                 end
175                 5'b11100: begin
176                         if(jumpTag == 1) begin
177                                 F = A + B;
178                         end
179                         else if(jumpTag == 0) begin
180                                 F = A;
181                         end
182                 end
183         endcase
184 end
185
186
187 endmodule
```

2.2.2 移位器

```
1 module shift16 (
2         input[15:0] A,
```

```
        input P ,                   //P=0 左移   P=1右移
        output reg[15:0] B
);
always @(*) begin
        if(P == 1) begin
                integer i;
                for(i = 0;i < 15;i = i + 1) begin
                        B[i] <= A[i+1];
                end
                B[15] <= 0;
        end
        else begin
                integer j;
                for(j = 1;j < 16;j = j + 1) begin
                                B[j] <= A[j - 1];
                end
                B[0] <= 0;
        end
end
endmodule
```

2.2.3 求补器

```
module complement16(
        input wire[15:0] A,
        output reg[15:0] B
);


reg C[16:0];
reg E ;

always @(*) begin
        integer i;
        B[15] = A[15];
        E = A[15];
        C[0] = 0 ;
```

```
        for(i = 0;i < 15;i = i + 1) begin
                B[i] = (E & C[i]) ^ A[i];
                C[i + 1] = C[i] | A[i];
        end
end

endmodule
```

2.2.4 阵列乘法器

2.2.4.1 基本加法单元

```
module add1(
        input A ,
        input B ,
        input C ,
        output S ,
        output C4
);
        assign S = A ^ B ^ C;
        assign C4 = (A & B) | ((A | B) & C);

endmodule
```

2.2.4.2 基本乘法单元

```
module multiunit(
        input A,
        input B,
        input C,                    //进位输入
        input M,                    //部分积输入

        output C4,
        output M4       //部分积输出
);

        add1 add(
```

```
                .A(M),
                .B(A&B),
                .C(C),
                .S(M4) ,
                .C4(C4)
        );


endmodule
```

2.2.4.3 行单元

```
module multi16row(
        input[15:0] A ,
        input B,
        input[15:0] M,          //部分积输入

        output[15:0] M4,        //部分积输出
        output P                        //结果输出
);

        wire[16:0] C;
        wire[15:0] MT;
        assign P = MT[0];
        assign M4 = {C[16], MT[15:1]};
        assign C[0] = 0;
        genvar i;
        generate
                for(i = 0;i < 16;i = i + 1) begin: union
                        multiunit unit(
                                .A(A[i]),
                                .B(B),
                                .C(C[i]),       //进位输入
                                .M(M[i]),       //部分积输入

                                .C4(C[i + 1]),
                                .M4(MT[i]),     //部分积输出
```

```
                        );
                end
        endgenerate


endmodule
```

2.2.4.4 整体结构

```
module multiplication16(
        input[15:0] A ,
        input[15:0] B,
        output[15:0] C
);
        wire[31:0] ans;
        wire[15:0] line[16:0];

        assign C = ans[15:0];
        assign line[0] = 16'b0;
        assign ans[31:16] = line[16];

        genvar i;
        generate
                for(i = 0;i < 16;i = i + 1) begin: union
                        multi16row row(
                                .A(A),
                                .B(B[i]),
                                .M(line[i]),     //部分积输入

                                .M4(line[i+1]), //部分积输出
                                .P(ans[i])       //结果输出
                        );
                end
        endgenerate

endmodule
```

2.3 A、B 选择器

A、B 选择器采用相同结构，均为 8 选 1 选择器，但控制信号有所不同。一种选择信号为 3 位数，内置 3-8 译码器译码；一种选择信号为 8 位数，无需译码。其中，寄存器 A、B 选择器采用了后者，作为 ALU 输入的 A、B 选择器使用了前者。

2.3.1 3 位选择信号的选择器

```
1
2  module selector8(
3          input [15:0] D0 ,
4          input [15:0] D1 ,
5          input [15:0] D2 ,
6          input [15:0] D3 ,
7          input [15:0] D4 ,
8          input [15:0] D5 ,
9          input [15:0] D6 ,
10         input [15:0] D7 ,
11         input [2:0] CPQ ,
12         output [15:0] Q ,
13         output[7:0] YY
14 );
15
16 wire[7:0] PY;
17 wire[7:0] Y;
18 assign YY = PY;
19
20 \74138 three_two_translator(
21         .G1(1) ,
22         .G2AN(0) ,
23         .G2BN(0) ,
24         .A(CPQ[0]) ,
25         .B(CPQ[1]) ,
26         .C(CPQ[2]) ,
27         .Y7N(Y[7]) ,
28         .Y6N(Y[6]) ,
29         .Y5N(Y[5]) ,
30         .Y4N(Y[4]) ,
```

```
        .Y3N(Y[3]) ,
        .Y2N(Y[2]) ,
        .Y1N(Y[1]) ,
        .Y0N(Y[0])
);


genvar i;
generate
        for(i = 0;i < 8;i = i + 1) begin:rank
                assign PY[i] = ~Y[i];
        end
        for(i = 0;i < 16;i = i + 1) begin:rank4
                assign Q[i] = (PY[0]&D0[i]) |
                              (PY[1]&D1[i]) |
                              (PY[2]&D2[i]) |
                              (PY[3]&D3[i]) |
                              (PY[4]&D4[i]) |
                              (PY[5]&D5[i]) |
                              (PY[6]&D6[i]) |
                              (PY[7]&D7[i]);
        end
endgenerate
endmodule
```

2.3.2 8 位选择信号的选择器

```
module selector8_2(
        input [15:0] D0 ,
        input [15:0] D1 ,
        input [15:0] D2 ,
        input [15:0] D3 ,
        input [15:0] D4 ,
        input [15:0] D5 ,
        input [15:0] D6 ,
        input [15:0] D7 ,
        input [7:0] Y ,
```

```
        output [15:0] Q
);

genvar i;
generate
        for(i = 0;i < 16;i = i + 1) begin:rank
                assign Q[i] = (Y[0]&D0[i]) |
                              (Y[1]&D1[i]) |
                              (Y[2]&D2[i]) |
                              (Y[3]&D3[i]) |
                              (Y[4]&D4[i]) |
                              (Y[5]&D5[i]) |
                              (Y[6]&D6[i]) |
                              (Y[7]&D7[i]);
        end
endgenerate



endmodule
```

2.4 通用寄存器、程序计数器、指令寄存器和存储器地址寄存器

以上各种寄存器均由 4 个 4 位寄存器 74173 组成。组成结构一致。

```
module register16(
        input[15:0] D ,
        input CLR ,
        input CPR ,
        output[15:0] Q
);

        wire CLRN = ~CLR;

        \74173 register8_1(
                .D1(D[0]) ,
                .D2(D[1]) ,
```

```
13                  .D3(D[2]) ,
14                  .D4(D[3]) ,
15                  .G1N(0) ,
16                  .G2N(0) ,
17                  .MN(0) ,
18                  .NN(0) ,
19                  .CLR(CLRN) ,
20                  .CLK(CPR) ,
21                  .Q1(Q[0]) ,
22                  .Q2(Q[1]) ,
23                  .Q3(Q[2]) ,
24                  .Q4(Q[3])
25          ) ;
26
27          \74173 register8_2(
28                  .D1(D[4]) ,
29                  .D2(D[5]) ,
30                  .D3(D[6]) ,
31                  .D4(D[7]) ,
32                  .G1N(0) ,
33                  .G2N(0) ,
34                  .MN(0) ,
35                  .NN(0) ,
36                  .CLR(CLRN) ,
37                  .CLK(CPR) ,
38                  .Q1(Q[4]) ,
39                  .Q2(Q[5]) ,
40                  .Q3(Q[6]) ,
41                  .Q4(Q[7])
42          ) ;
43          \74173 register8_3(
44                  .D1(D[8]) ,
45                  .D2(D[9]) ,
46                  .D3(D[10]) ,
47                  .D4(D[11]) ,
48                  .G1N(0) ,
49                  .G2N(0) ,
```

```
                .MN(0) ,
                .NN(0) ,
                .CLR(CLRN) ,
                .CLK(CPR) ,
                .Q1(Q[8]) ,
                .Q2(Q[9]) ,
                .Q3(Q[10]) ,
                .Q4(Q[11])
        ) ;
        \74173 register8_4(
                .D1(D[12]) ,
                .D2(D[13]) ,
                .D3(D[14]) ,
                .D4(D[15]) ,
                .G1N(0) ,
                .G2N(0) ,
                .MN(0) ,
                .NN(0) ,
                .CLR(CLRN) ,
                .CLK(CPR) ,
                .Q1(Q[12]) ,
                .Q2(Q[13]) ,
                .Q3(Q[14]) ,
                .Q4(Q[15])
        ) ;

endmodule
```

2.5 随机访问存储器

RAM 采用了 Quartus II 自带的 IP 核。

```
ram RAM(
                .address(address_RAM[9:0]) ,
                .clock(clk_RAM) ,
                .data(data_RAM) ,
                .q(q_RAM) ,
```

```
6                       .wren(wren_RAM)  ,
7                       .rden(rden_RAM)
8               );
```

3 微程序实现的控制部件 CU

3.1 整体框图

图3给出了微程序实现的控制单元 CU 的整体框图。

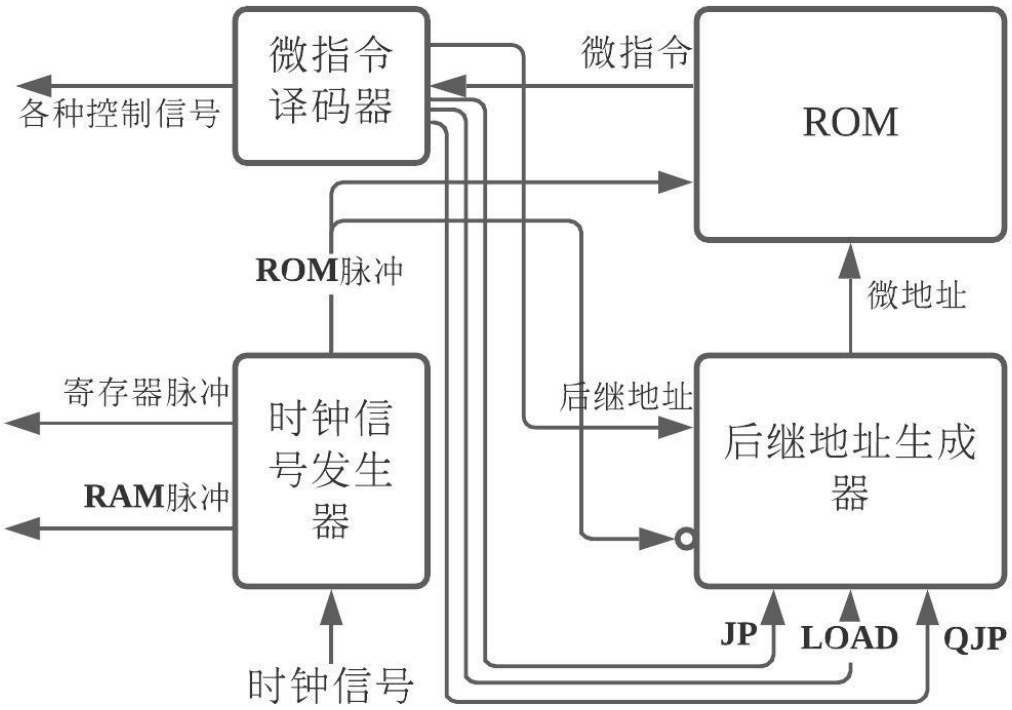

图 3: 微程序控制部件

时钟信号发生器将时钟信号按照合适的时序关系，处理为寄存器脉冲信号、RAM 脉冲信号和 ROM 脉冲信号。

后继地址生成器内含有微地址寄存器，根据上一次微指令给出的后继地址生成方式，生成下一个微地址并存储在微地址寄存器中并输出。

ROM 存储所有微程序。

微指令译码器将微指令翻译为各种控制信号。在实际实现中，该译码器并没有单独整合为一个器件，而是与数据通路一起放在 CPU 中。附件 6 中展示的该器件的原理图实际为 CPU 原理图。

3.2 时钟信号发生器

```
module clock3(
        input CLK ,
        input CLR ,
        output[2:0] q
);

        reg[2:0] i = 3'b111;
        assign q[0] = ~i[0];
        assign q[1] = ~i[1];
        assign q[2] = ~i[2];

        always @(posedge CLK or negedge CLR) begin
                if (CLR == 1'b0)
                        i <= 3'b111;
                else
                        i <= i + 3'b001;
        end

endmodule
```

3.3 后继地址生成器

```
module nextAddress(
        input LOAD ,
        input CLK ,
        input CLR ,
        input JP ,
        input QJP ,
        input[8:0] UIR ,
        input[4:0] OP ,
        output[8:0] address
);

reg[8:0] next_address = 9'b0;
```

```
13
14 assign address = next_address;
15
16 always @(posedge CLK or negedge CLR) begin
17         if(CLR == 0) begin
18                 next_address = 0;
19         end
20         else if(CLR && LOAD == 1) begin
21                 next_address = next_address + 1;
22         end
23         else if(CLR && LOAD == 0 && JP == 1) begin
24                 next_address = UIR;
25         end
26         else if(CLR && LOAD == 0 && QJP == 1) begin
27                 next_address = {OP, 4'b0000};
28         end
29 end
30
31 endmodule
```

3.4 微指令译码器

微指令译码功能与数据通路共同组成 CPU，并未以单独器件编写。因此，下面展示的代码为 CPU 代码中译码部分。

3.4.1 2-4 译码器

```
1
2 module twoToFour(
3         input[1:0] A,
4         output[3:0] B
5 );
6
7         assign B[0] = (!A[0]) && (!A[1]);
8         assign B[1] = (A[0]) && (!A[1]);
9         assign B[2] = (!A[0]) && (A[1]);
10        assign B[3] = (A[0]) && (A[1]);
```

```
endmodule
```

3.4.2 3-8 译码器

```
module twoToFour(
        input[1:0] A,
        output[3:0] B
);

        assign B[0] = (!A[0]) && (!A[1]);
        assign B[1] = (A[0]) && (!A[1]);
        assign B[2] = (!A[0]) && (A[1]);
        assign B[3] = (A[0]) && (A[1]);
endmodule
```

3.4.3 整体结构

```
        //微指令
        wire[63:0] microInstruction;

        //停机
        assign HaltTag = microInstruction[3];

        //8个寄存器的脉冲信号的选择控制
        wire[15:0] R_in[7:0], R_out[7:0];
        wire[15:0] RA, RB;

        wire CPR_P = microInstruction[38] & P_R;

        wire[7:0] CPR;
        wire[2:0] CPR3_from_microInstruction;
        wire[2:0] CPR3_from_instruction[2:0];
        wire[7:0] CPR_from_microInstruction;
        wire[7:0] CPR_from_instruction[2:0];
        wire[1:0] CPR_select_control2;
        wire[3:0] CPR_select_control4;
```

```
20
21          assign CPR3_from_microInstruction
22          = microInstruction[35:33];
23          assign CPR3_from_instruction[0]
24          = instruction[10:8];
25          assign CPR3_from_instruction[1]
26          = instruction[7:5];
27          assign CPR3_from_instruction[2]
28          = instruction[4:2];
29          assign CPR_select_control2
30          = microInstruction[37:36];
31          threeToEight cpr_three_eight1(
32                  .A(CPR3_from_microInstruction) ,
33                  .B(CPR_from_microInstruction)
34          );
35          threeToEight cpr_three_eight2(
36                  .A(CPR3_from_instruction[0]) ,
37                  .B(CPR_from_instruction[0])
38          );
39          threeToEight cpr_three_eight3(
40                  .A(CPR3_from_instruction[1]) ,
41                  .B(CPR_from_instruction[1])
42          );
43          threeToEight cpr_three_eight4(
44                  .A(CPR3_from_instruction[2]) ,
45                  .B(CPR_from_instruction[2])
46          );
47          twoToFour cpr_tow_four1(
48                  .A(CPR_select_control2) ,
49                  .B(CPR_select_control4)
50          );
51          genvar j;
52          generate
53                  for(j = 0;j < 8;j = j + 1) begin: cpr
54                          assign R_in[j] = ALU_F;
55                          assign CPR[j] = CPR_P &
56  ((CPR_from_microInstruction[j] & CPR_select_control4[3]) |
```

```
(CPR_from_instruction[0][j] & CPR_select_control4[0]) |
(CPR_from_instruction[1][j] & CPR_select_control4[1]) |
(CPR_from_instruction[2][j] & CPR_select_control4[2]));
                end
        endgenerate

        //8个寄存器的选择控制器
        wire[7:0] select_control_RA, select_control_RB;

        wire[2:0] select_control_RA_from_microInstruction3,
        select_control_RB_from_microInstruction3;
        wire[2:0] select_control_RA_from_instruction3[2:0],
        select_control_RB_from_instruction3[2:0];

        wire[7:0] select_control_RA_from_microInstruction8,
        select_control_RB_from_microInstruction8;
        wire[7:0] select_control_RA_from_instruction8[2:0],
        select_control_RB_from_instruction8[2:0];

        wire[1:0] control_select_control_RA2,
        control_select_control_RB2;
        wire[3:0] control_select_control_RA4,
        control_select_control_RB4;

        assign control_select_control_RA2 =
        microInstruction[7:6];
        assign control_select_control_RB2 =
        microInstruction[9:8];
        assign select_control_RA_from_microInstruction3 =
        microInstruction[12:10];
        assign select_control_RB_from_microInstruction3 =
        microInstruction[15:13];

        assign select_control_RA_from_instruction3[0] =
        instruction[10:8];
        assign select_control_RB_from_instruction3[0] =
        instruction[10:8];
```

```
94          assign select_control_RA_from_instruction3[1] =
95          instruction[7:5];
96          assign select_control_RB_from_instruction3[1] =
97          instruction[7:5];
98          assign select_control_RA_from_instruction3[2] =
99          instruction[4:2];
100         assign select_control_RB_from_instruction3[2] =
101         instruction[4:2];
102
103         threeToEight select_control_three_eight1(
104                 .A(select_control_RA_from_microInstruction3) ,
105                 .B(select_control_RA_from_microInstruction8)
106         );
107         threeToEight select_control_three_eight2(
108                 .A(select_control_RB_from_microInstruction3) ,
109                 .B(select_control_RB_from_microInstruction8)
110         );
111         threeToEight select_control_three_eight3(
112                 .A(select_control_RA_from_instruction3[0]) ,
113                 .B(select_control_RA_from_instruction8[0])
114         );
115         threeToEight select_control_three_eight4(
116                 .A(select_control_RB_from_instruction3[0]) ,
117                 .B(select_control_RB_from_instruction8[0])
118         );
119         threeToEight select_control_three_eight5(
120                 .A(select_control_RA_from_instruction3[1]) ,
121                 .B(select_control_RA_from_instruction8[1])
122         );
123         threeToEight select_control_three_eight6(
124                 .A(select_control_RB_from_instruction3[1]) ,
125                 .B(select_control_RB_from_instruction8[1])
126         );
127         threeToEight select_control_three_eight7(
128                 .A(select_control_RA_from_instruction3[2]) ,
129                 .B(select_control_RA_from_instruction8[2])
130         );
```

```
131         threeToEight select_control_three_eight8(
132                 .A(select_control_RB_from_instruction3[2]) ,
133                 .B(select_control_RB_from_instruction8[2])
134         );
135         twoToFour select_control_two_Four1(
136                 .A(control_select_control_RA2) ,
137                 .B(control_select_control_RA4)
138         );
139         twoToFour select_control_two_Four2(
140                 .A(control_select_control_RB2) ,
141                 .B(control_select_control_RB4)
142         );
143         genvar k;
144         generate
145                 for(k = 0;k < 8;k = k + 1) begin: select_control
146 assign select_control_RA[k] =
147 (control_select_control_RA4[3] &
148 select_control_RA_from_microInstruction8[k]) |
149 (control_select_control_RA4[0] &
150 select_control_RA_from_instruction8[0][k]) |
151 (control_select_control_RA4[1] &
152 select_control_RA_from_instruction8[1][k]) |
153 (control_select_control_RA4[2] &
154 select_control_RA_from_instruction8[2][k]);
155
156 assign select_control_RB[k] = (control_select_control_RB4[3]
157 & select_control_RB_from_microInstruction8[k]) |
158 (control_select_control_RB4[0] &
159 select_control_RB_from_instruction8[0][k]) |
160 (control_select_control_RB4[1] &
161 select_control_RB_from_instruction8[1][k]) |
162 (control_select_control_RB4[2] &
163 select_control_RB_from_instruction8[2][k]);
164                 end
165         endgenerate
166
167         assign CPIR = P_R & microInstruction[32];
```

```
168
169         assign rden_RAM = (microInstruction[5])
170                     && (!microInstruction[4]);
171         assign wren_RAM = (microInstruction[4])
172                     && (!microInstruction[5]);
173
174         assign CPMAR = P_R & microInstruction[28];
175         assign MAR_REST = microInstruction[29];
176
177         assign CPPC = P_R & microInstruction[30];
178         assign PC_REST = microInstruction[31];
179
180         assign select_control_A = microInstruction[18:16];
181         assign select_control_B = microInstruction[21:19];
182
183         assign ALU_S = microInstruction[26:22];
184         assign ALU_CN = microInstruction[27];
185
186         assign control_next_micro_address = microInstruction[2:0];
187         assign micro_address_from_UIR = microInstruction[47:39];
188         assign micro_address_from_OP = instruction[15:11];
189         assign LOAD = (control_next_micro_address[0]) &&
190 (!control_next_micro_address[1]) &&
191 (!control_next_micro_address[2]);
192         assign JP = (!control_next_micro_address[0]) &&
193 (control_next_micro_address[1]) &&
194 (!control_next_micro_address[2]);
195         assign QJP = (control_next_micro_address[0]) &&
196 control_next_micro_address[1]) &&
197 (!control_next_micro_address[2]);
```

3.5 ROM

ROM 采用了 Quartus II 提供的 IP 核实现。

```
1     rom ROM1(
2                 .clock(P_UPC) ,
```

```
3                   .address(next_micro_address) ,
4                   .q(microInstruction)
5           );
```

3.6 指令执行流程

图4给出了各个指令的执行流程图。其中，第一行为各条指令取值周期的执行流程，后面各行为各指令执行周期的执行流程。每一条微指令占用一个节拍。

操作	执行流程					
取值周期	RAM->IR	PC+1->PC	QJP			
ADD RS1 RS2 RD 00	RS1+RS2->RD	PC->MAR	JP			
MOVIH RS1 #8-bit	#H8bit+RS1#L8bit->RS1	PC->MAR	JP			
MOVIL RS1 #8-bit	RS1#H8bit+#L8bit->RS1	PC->MAR	JP			
MOV RS1 RS2 000 00	RS1->RS2	PC->MAR	JP			
SUB RS1 RS2 RD 00	RS1-RS2->RD	PC->MAR	JP			
LDIDR RS1 RS2 #5bit	RS1+#5bit->MAR	RAM->RS2	PC->MAR	JP		
STIDR RS1 RS2 #5bit	RS1+#5bit->MAR	RS2->RAM	PC->MAR	JP		
LDIDX RS1 RS2 RS3	RS1+RS2->MAR	RAM->RS3	PC->MAR	JP		
STIDX RS1 RS2 RS3	RS1+RS2->MAR	RS3->RAM	PC->MAR	JP		
AND RS1 RS2 RS3	RS1 AND RS2->RS3	PC->MAR	JP			
OR RS1 RS2 RS3	RS1 OR RS2->RS3	PC->MAR	JP			
XOR RS1 RS2 RS3	RS1 XOR RS2 -> RS3	PC->MAR	JP			
NOT RS1 RS2	NOT RS1->RS2	PC->MAR	JP			
SHIFTL RS1 RS2	SHIFTL RS1->RS2	PC->MAR	JP			
SHIFTR RS1 RS2	SHIFTR RS1->RS2	PC->MAR	JP			
MULTI RS1 RS2 RS3	RS1*RS2->RS3	PC->MAR	JP			
JMP #11-bit	PC+#11-bit->PC	PC->MAR	JP			
JMPI RS1 #8-bit	RS1->PC	PC+#5bit->PC	PC->MAR	JP		
JGEO RS1 RS2 #5-bit	RS1>=RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
JLEO RS1 RS2 #5-bit	RS1<=RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
JEO RS1 RS2 #5-bit	RS1==RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
NOP	PC->MAR	JP				
ADDI RS1 RS2 #5-bit	RS1+#5-bit->RS2	PC->MAR	JP			
SUBI RS1 RS2 #5-bit	RS1-#5-bit->RS2	PC->MAR	JP			
PUSH RS1	SP->MAR	SP-1->SP	RS1->RAM	PC->MAR	JP	
POP RS1	SP+1->SP	SP->MAR	RAM->RS1	PC->MAR	JP	
MOVESP RS1	RS1->SP	PC->MAR	JP			
CALL RS1	SP->MAR	SP-1->SP	PC->RAM	RS1->PC	PC->MAR	JP
JAL #11-bit	SP->MAR	SP-1->SP	PC->RAM	PC+#11-bit->PC	PC->MAR	JP
RET	SP+1->SP	SP->MAR	RAM->PC	PC->MAR	JP	
HLT	HALT					

图 4: 执行流程

下面给出图中简写的解释。

1. A->B 表示将 A 运算输出的内容输入到 B。
2. RS1、RS2、RD 表示指令中指定的不同位置的通用寄存器。
3. #xbit 或 #x-bit 表示从指令中获取的立即数，位数为 x。
4. A->jumpTAG 表示将表达式 A 的值存到 ALU 中的跳转标记寄存器。

5. jumpTag(x)->PC 表示，若跳转标记寄存器的值为 0，则 PC 值不变；若跳转标记寄存器的值为 1，则把表达式 x 的值赋值给 PC。

6. SP 表示栈帧寄存器，即通用寄存器 7（R7）。

7. HALT 表示停机操作。

8. JP 表示微地址跳转到取值周期对应的微程序。

9. QJP 表示微地址按操作码跳转到对应的微程序。

3.7 微指令

图5给出了微指令格式。

47-39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
9位后继地址	通用寄存器脉冲	寄存器脉冲选择		8个寄存器脉冲			IR脉冲	PC复位信号	PC脉冲	MAR复位信号	MAR脉冲	ALU进位	ALU控制（S5-S1）					B选择器的选择信号			A选择器的选择信号			B选择器中的寄存器选择信号			A选择器中的寄存器选择信号			B选择指令寄存器		A选择指令寄存器		10：读使能；01：		停机控制。0运行，1停机			后继地址形成方式

图 5: 微指令格式

图6给出了各个微指令的 16 位代码。

微指令		微指令	
RAM->IR	00000031A00203E1	SHIFTR RS1->RS2	00000050A2C80301
PC->PC+1	00000030E10703C1	RS1*RS2->RS3	00000060A2480101
QJP	00000030A00003C3	PC+#11-bit->PC	00000030E0AF03C1
PC->MAR	00000030B00703C1	RS1+#5-bit->RS2	00000050A0980301
RS1+RS2->RD	00000060A0880101	RS1-#5-bit->RS2	00000050A0D80301
JP0	00000030A00003C2	RS1->PC	00000030E0080301
#H8bit+RS1#L8bit->RS1	00000040A5CC03C1	PC+#8bit->PC	00000030E09F03C1
RS1#H8bit+#L8bit->RS1	00000040A6200301	RS1>=RS2->jumpTAG	00000030A6480101
RS1->RS2	00000050A0080101	jumpTag(PC+#5-bit)->PC	00000030E71F03C1
RS1-RS2->RD	00000060A0C80101	RS1<=RS2->jumpTAG	00000030A6880101
RS1+#5bit->MAR	00000030B0980301	RS1==RS2->jumpTAG	00000030A6C80101
RAM->RS2	00000050A00A03E1	SP->MAR	00000030B0081FC1
RS2->RAM	00000030A0080351	RS1->RAM	00000030A0080311
RS1+RS2->MAR	00000030B0880101	SP-1->SP	0000007EA1481FC1
RAM->RS3	00000060A0080BE1	SP+1->SP	0000007EA1081FC1
RS3->RAM	00000030A0080391	RAM->RS1	00000040A00A03E1
RS1 AND RS2->RS3	00000060A3880101	RS1->SP	0000007EA0080301
RS1 OR RS2->RS3	00000060A3C80101	PC->RAM	00000030A00F03D1
RS1 XOR RS2->RS3	00000060A4080101	RAM->PC	00000030E00A03E1
NOT RS1->RS2	00000050A4480301	PC+#11-bit->PC	00000030E0AF03C1
SHIFTL RS1->RS2	00000050A2880301	HALT	00000030A00003C9

图 6: 微指令代码

地址（二进制）	+000	+001	+010	+011	+100	+101
00000	00000031A00203E1	00000030E10703C1	00000030A00003C3			
	RAM->IR	PC+1->PC	QJP			
00001	00000060A0880101	00000030B00703C1	00000030A00003C2			
	RS1+RS2->RD	PC->MAR	JP			
00010	00000040A5CC03C1	00000030B00703C1	00000030A00003C2			
	#H8bit+RS1#L8bit->RS1	PC->MAR	JP			
00011	00000040A6200301	00000030B00703C1	00000030A00003C2			
	RS1#H8bit+#L8bit->RS1	PC->MAR	JP			
00100	00000050A0080101	00000030B00703C1	00000030A00003C2			
	RS1->RS2	PC->MAR	JP			
00101	00000060A0C80101	00000030B00703C1	00000030A00003C2			
	RS1-RS2->RD	PC->MAR	JP			
00110	00000030E0980301	00000050A00A03E1	00000030B00703C1	00000030A00003C2		
	RS1+#5bit->MAR	RAM->RS2	PC->MAR	JP		
00111	00000030E0980301	00000030A0080351	00000030B00703C1	00000030A00003C2		
	RS1+#5bit->MAR	RS2->RAM	PC->MAR	JP		
01000	00000030E0880101	00000060A0080EE1	00000030B00703C1	00000030A00003C2		
	RS1+RS2->MAR	RAM->RS3	PC->MAR	JP		
01001	00000030E0880101	00000030A0080391	00000030B00703C1	00000030A00003C2		
	RS1+RS2->MAR	RS3->RAM	PC->MAR	JP		
01010	00000060A3880101	00000030B00703C1	00000030A00003C2			
	RS1&RS2->RS3	PC->MAR	JP			
01011	00000060A3C80101	00000030B00703C1	00000030A00003C2			
	RS1 OR RS2->RS3	PC->MAR	JP			
01100	00000060A4080101	00000030B00703C1	00000030A00003C2			
	RS1 XOR RS2 -> RS3	PC->MAR	JP			
01101	00000050A4480301	00000030B00703C1	00000030A00003C2			
	NOT RS1->RS2	PC->MAR	JP			
01110	00000050A2880301	00000030B00703C1	00000030A00003C2			
	SHIFTL RS1->RS2	PC->MAR	JP			
01111	00000050A2C80301	00000030B00703C1	00000030A00003C2			
	SHIFTR RS1->RS2	PC->MAR	JP			
10000	00000060A2480101	00000030B00703C1	00000030A00003C2			
	RS1*RS2->RS3	PC->MAR	JP			
10001	00000030E0AF03C1	00000030B00703C1	00000030A00003C2			
	PC+#11-bit->PC	PC->MAR	JP			
10010	00000030E0080301	00000030E0A703C1	00000030B00703C1	00000030A00003C2		
	RS1->PC	PC+#8bit->PC	PC->MAR	JP		
10011	00000030A6480101	00000030E71F03C1	00000030B00703C1	00000030A00003C2		
	RS1>=RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
10100	00000030A6880101	00000030E71F03C1	00000030B00703C1	00000030A00003C2		
	RS1<=RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
10101	00000030A6C80101	00000030E71F03C1	00000030B00703C1	00000030A00003C2		
	RS1==RS2->jumpTAG	jumpTag(PC+#5-bit)->PC	PC->MAR	JP		
10110	00000030B00703C1	00000030A00003C2				
	PC->MAR	JP				
10111	00000050A0980301	00000030B00703C1	00000030A00003C2			
	RS1+#5-bit->RS2	PC->MAR	JP			
11000	00000050A0D80301	00000030B00703C1	00000030A00003C2			
	RS1-#5-bit->RS2	PC->MAR	JP			
11001	00000030B0081FC1	0000007EA1481FC1	00000030A0080311	00000030B00703C1	00000030A00003C2	
	SP->MAR	SP-1->SP	RS1->RAM	PC->MAR	JP	
11010	0000007EA1081FC1	00000030B0081FC1	00000040A00A03E1	00000030B00703C1	00000030A00003C2	
	SP+1->SP	SP->MAR	RAM->RS1	PC->MAR	JP	
11011	0000007EA0080301	00000030B00703C1	00000030A00003C2			
	RS1->SP	PC->MAR	JP			
11100	00000030B0081FC1	0000007EA1481FC1	00000030A00F03D1	00000030E0080301	00000030B00703C1	00000030A00003C2
	SP->MAR	SP-1->SP	PC->RAM	RS1->PC	PC->MAR	JP
11101	00000030B0081FC1	0000007EA1481FC1	00000030A00F03D1	00000030E0AF03C1	00000030B00703C1	00000030A00003C2
	SP->MAR	SP-1->SP	PC->RAM	PC+#11-bit->PC	PC->MAR	JP
11110	0000007EA1081FC1	00000030B0081FC1	00000030E00A03E1	00000030B00703C1	00000030A00003C2	
	SP+1->SP	SP->MAR	RAM->PC	PC->MAR	JP	
11111	00000030A00003C9					
	HALT					

图 7: ROM 中的微程序

3.8 微程序

图7给出了 ROM 中微程序的内容和存储地址。

3.9 RAM 中的应用程序

图8给出了 RAM 中的应用程序。左侧为机器码，右侧为汇编码。

地址	+00		+01		+10		+11	
000000000000	0001000111111111	MOVIH R1 #11111111	0000100000101000	ADD R0 R1 R2	0001101001011010	MOVIL R2 01011010	0010001001100000	MOV R2 R3
000000000100	0010101100110000	SUB R3 R1 R4	0011000010100010	LDIDR R0 R4 00010	0011100101000000	STIDR R1 R2 00000	0011000111000000	LDIDR R1 R6 00000
000000001000	0100000000111100	LDIDX R0 R1 R7	0100101100010100	STIDX R3 R0 R5	0101000110111000	AND R1 R5 R6	0011100111000001	STIDR R1 R6 00001
000000001100	0101100110011000	OR R1 R4 R6	0011100111000010	STIDR R1 R6 00010	0110000110111000	XOR R1 R5 R6	0011100111000011	STIDR R1 R6 00011
000000010000	0110110111000000	NOT R5 R6	0011100111000100	STIDR R1 R6 00100	0111010111000000	SHIFTL R5 R6	0011100111000101	STIDR R1 R6 00101
000000010100	0111110111000000	SHIFTR R5 R6	0011100111000110	STIDR R1 R6 00110	1000010010011000	MULTI R4 R4 R6	0011100111000111	STIDR R1 R6 00111
000000011000	1000010000000001	JMP #1	0011100100000000	STIDR R1 R0 00000	0011100100101000	STIDR R1 R1 01000	1011100001100010	ADDI R0 R3 00010
000000011100	1100001101100001	SUBI R3 R3 00001	0011100101101001	STIDR R1 R3 01001	1001000000100000	JMPI R0 00100000	0011100100000000	STIDR R1 R0 00000
000000100000	0011100100101010	STIDR R1 R1 #01010	1001100101000001	JGEO R1 R2 1	1001100100000001	JGEO R1 R0 1	0011100100000000	STIDR R1 R0 00000
000000100100	0011100100101011	STIDR R1 R1 01011	1010000100000001	JLEO R1 R0 1	1010000101000001	JLEO R1 R2 1	0011100100000000	STIDR R1 R0 00000
000000101000	0011100100101100	STIDR R1 R1 #01010	1010100101000001	JEO R1 R2 1	1010101001000001	JEO R2 R2 1	0011100100000000	STIDR R1 R0 00000
000000101100	0011100100101101	STIDR R1 R1 01101	1011000000000000	HLT	0010000111100000	MOV R1 R7	0001111111111111	MOVIL R7 11111111
000000110000	1100100000000000	PUSH R0	1100100100000000	PUSH R1	1100101000000000	PUSH R2	1100101100000000	PUSH R3
000000110100	1100110000000000	PUSH R4	1101011000000000	POP R6	1101011000000000	POP R6	1100100100000000	PUSH R1
000000111000	1100100100000000	PUSH R1	0001100111100000	MOVIL R1 11100000	1101100100000000	MOVESP R1	0001001011111111	MOVIH R2 11111111
000000111100	0001101000000000	MOVIL R2 00000000	0001010011111111	MOVIH R4 11111111	0001110010000000	MOVIL R4 10000000	1110010000000000	CALL R4
000001000000	0011001011001110	LDIDR R2 R6 #01110	1110101101000110	JAL 01101000110	0011001011001111	LDIDR R2 R6 01111	0011101000110000	STIDR R2 R1 10000
000001000100	0011101001010001	STIDR R2 R2 10001	0011101001110010	STIDR R2 R3 10010	0011101010010011	STIDR R2 R4 10011	0011101010110100	STIDR R2 R5 10100
000001001000	0011101011010101	STIDR R2 R6 10101	0011101011110110	STIDR R2 R7 10110	1111100000000000	HLT	0011101001010000	STIDR R2 R2 10000

01110000000	0011101000101110	STIDR R2 R1 01110	1111000000000000	RET
01110001000	0011101001101111	STIDR R2 R3 01111	1111000000000000	RET

图 8: RAM 中的应用程序

4 硬布线实现的控制部件 CU

4.1 整体框图

图9给出了硬布线实现的整体框图。

其中，节拍发生器根据时钟信号和指令操作码产生寄存器脉冲、RAM 脉冲和 W0、W1、T0、T1、T2、T3 六个节拍。W0、W1 控制取指周期和执行周期，T0、T1、T2、T3 控制执行周期的四个节拍。根据操作码判断指令需要执行几个节拍，从而使得在指令执行结束后立即跳过后续节拍，进入取指周期。

控制器生成信号根据指令和六个节拍，产生相应的各种控制信号。

4.2 节拍发生器

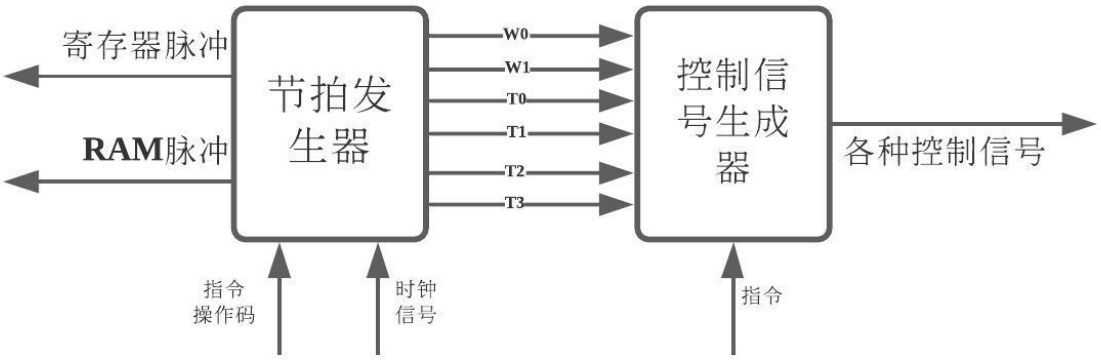

图 9: 硬布线控制部件

```
    module clock(
        input CLK ,
        input CLR ,
        input[4:0] OP ,

        output P_R ,
        output P_RAM ,
        output T0 ,
        output T1 ,
        output T2 ,
        output T3 ,
        output W0 ,
        output W1
);

        reg[3:0] num_P = 4'b1111;
        reg[1:0] num_P_down = 2'b11;



        assign P_R = num_P[1];
        //assign P_RAM = ~num_P[1];
        assign P_RAM = ~num_P_down[1];
```

```
24
25          reg[3:0] Tx = 4'b0000;
26          reg[1:0] Wx = 2'b01;
27
28          assign T0 = Tx[0];
29          assign T1 = Tx[1];
30          assign T2 = Tx[2];
31          assign T3 = Tx[3];
32
33          assign W0 = Wx[0];
34          assign W1 = Wx[1];
35
36          always @(negedge CLK) begin
37                  num_P_down = num_P_down + 2'b01;
38          end
39
40          always @(posedge CLK) begin
41                  if(num_P == 4'b1011 && Wx[0]) begin
42                          Wx = 2'b10;
43                          num_P = 4'b1111;
44                  end
45                  else if(num_P == 4'b1111 && Wx[1]) begin
46                          Wx = 2'b01;
47                  end
48                  if((num_P == 4'b0011) && Wx[1] &&
49                  (OP == 5'b00001 || OP == 5'b00010 ||
50                  OP == 5'b00011 || OP == 5'b00100 ||
51                  OP == 5'b00101 || OP == 5'b01010 ||
52                  OP == 5'b01011 || OP == 5'b01100 ||
53                  OP == 5'b01101 || OP == 5'b01110 ||
54                  OP == 5'b01111 || OP == 5'b10000 ||
55                  OP == 5'b10001 || OP == 5'b10111 ||
56                  OP == 5'b11000 || OP == 5'b11011 ||
57                  OP == 5'b11111)) begin
58                          num_P = 4'b1111;
59                          Wx = 2'b01;
60                  end
```

```
61                  if((num_P == 4'b0111) && Wx[1] &&
62                  (OP == 5'b00110 || OP == 5'b00111 ||
63                  OP == 5'b01000 || OP == 5'b01001 ||
64                  OP == 5'b10010 || OP == 5'b10011 ||
65                  OP == 5'b10100 || OP == 5'b10101)) begin
66                          num_P = 4'b1111;
67                          Wx = 2'b01;
68                  end
69                  if((num_P == 4'b1011) && Wx[1] &&
70                  (OP == 5'b11001 || OP == 5'b11010 ||
71                  OP == 5'b11110)) begin
72                          num_P = 4'b1111;
73                          Wx = 2'b01;
74                  end
75                  num_P = num_P + 4'b0001;
76
77
78          if(num_P[3:1] == 3'b000) begin
79                          Tx = 4'b0001;
80                  end
81                  else if(num_P[3:1] == 3'b010) begin
82                          Tx = 4'b0010;
83                  end
84                  else if(num_P[3:1] == 3'b100) begin
85                          Tx = 4'b0100;
86                  end
87                  else if(num_P[3:1] == 3'b110) begin
88                          Tx = 4'b1000;
89                  end
90          end
91
92  endmodule
```

4.3 控制信号发生器

控制信号产生逻辑与数据通路一起写在 CPU 中，因此附件 6 中对应的 RTL Viewer 原理图对应于 CPU_yingbuxian.pdf 中。

由于相关代码过于庞大,原理图请参考附件 6,代码请参考附录 5 中的 CPU_yingbuxian.v。

4.4 指令执行流程

与微程序的指令流程一致，请参考图4.

需要注意的是，取值周期对应 W0 高电平，执行周期对应 W1 低电平。执行流程中的微操作，分别对应节拍 T0、T1、T2 和 T3。

4.5 控制信号列表和逻辑表达式

图10展示了控制信号及其含义。

35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
通用寄存器脉冲	寄存器脉冲选择		8个寄存器脉冲			IR脉冲	PC复位信号	PC脉冲	MAR复位信号	MAR脉冲	ALU进位	ALU控制（S5-S1）					B选择器的选择信号			A选择器的选择信号			B选择器中的寄存器选择信号			A选择器中的寄存器选择信号			B选择指令寄存器		A选择指令寄存器		10：读使能；01：写使		停机控制。0运行，1停机

图 10: 控制信号列表

控制信号表达式过于繁杂，不在文中放出，请参考附录 5 中的 ControlSignal.txt。其中，ix 表示指令第 x 位，x 为十六进制数。

4.6 RAM 中的应用程序

与微程序中的应用程序一直，图8给出了 RAM 中的应用程序。左侧为机器码，右侧为汇编码。

5 课程设计总结

本次课程设计带领我复习了计算机组成原理理论课程的核心原理，也帮助我补习了电路相关内容。课程设计完成后，原本模糊而空洞的理论变得可以触碰，CPU、机器码这些以前总觉得遥不可及的东西变得不再神秘，能够十分清晰的在脑海中回溯出计算机最底层的运行原理。本次设计让我更加深刻了解了以时序为基础的 CPU 设计，初步了解了各器件时延和时序关系设计对计算机运行的深刻影响。同时，通过对 verilog 等硬件编程语言的学习，让我见到了另一种以时序和同步为核心的全新的程序设计思维，受到很大启发。

同样，在后续学习计算理论、操作系统等课程中，由于本次课程设计所带来的的对于计算机底层逻辑的深入了解，帮助我更好的理解两门课程中的核心内容，如操作系统中多道程序的实现、原子指令等等。由于底层设计的复杂和抽象，本次课程设计的过程中，我还编写了大量 Excel 文档作为辅助，增进了对这类软件的了解。同时结合 C++ 程序帮助处理文档、根据文档自动生成 verilog 代码，让我在本次课程设计中体会到了多课程交叉的乐趣。

本次课程让我对计算机硬件层面的实现和优化充满兴趣，极大的激发了我对组成原理、操作系统、汇编语言等领域的兴趣。

在完成课程设计过程中，遇到过很多问题。如，有条件跳转过程中，由于时序的问题，条件跳转标记始终不能处于正常值。通过在 ALU 中添加一个寄存器的方式解决。

最后，感谢张瑞华老师的教学和指导，感谢吴世广、于逸潇两位同学的帮助，他们的帮助使我少走了很多弯路，也为我提供了很多新的思路。