

Lab6 - MIPS-CPU综合实验

实验内容

- 利用MIPS-CPU、存储器和HC-42蓝牙芯片完成一个计算两个数的最大公因数的应用。
- MIPS-CPU：Lab5设计的，实现了中断，增加了srlv、sllv、eret指令。
- 外设：拨动/按钮开关、指示灯、数码管、UART串口通信、HC-06蓝牙、安卓手机。
- 应用：计算两个数的最大公因数，以及在手机上调试mips-cpu。

应用详述

- 应用是：**手机通过蓝牙发送两个32位整数给mips-cpu，cpu 64位buffer满，产生中断信号，执行中断服务程序，在程序中计算两个数的最大公因数，然后再通过蓝牙返回给手机。**
- 安卓手机通过蓝牙与装有HC-06蓝牙外设的fpga相连，实现手机与mips-cpu蓝牙通信。
- HC-06蓝牙芯片通过串口与mips-cpu相连，串口通信使用UART协议。
- cpu使用中断的方式接收蓝牙的数据。即，当蓝牙数据缓冲区满时，产生中断信号，使cpu跳转执行蓝牙中断服务例程。
- 蓝牙通信数据通过IO内存映射的方式提供给cpu，cpu可以通过lw和sw指令直接访问蓝牙地址空间。
- 在蓝牙中断服务例程中完成取数据、计算最大公因数、存结果、返回给手机的操作。

逻辑设计

1.MIPS-CPU

cpu主要使用的是lab5设计的cpu。这里不再赘述。

内存大小为1024*32 bits。这样IO地址的起始为0x1000。

2.中断的实现

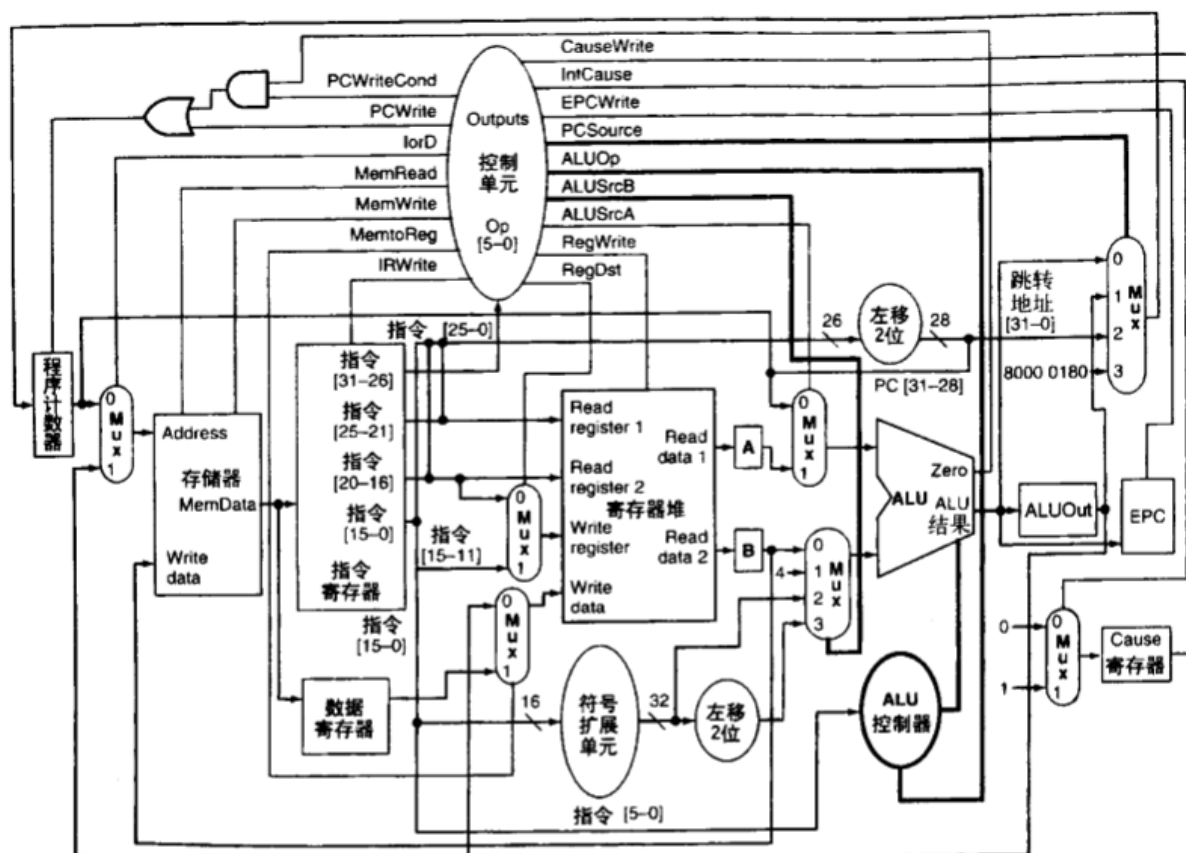


图 5-39 加入了异常处理的多周期数据通路

数据通路用的是COD3的上图所展示的。在原来的cpu上加了EPC错误指令寄存器和Cause寄存器以及相应的控制信号。书上的EPC是用来保存当前错误指令的，但是本实验中的中断要保存的是中断前要执行的下一条指令。Cause寄存器在这里并无实际用处，因为本实验中只有IO中断。

中断原理：

在每一条指令执行完后，进行中断检测，如果此时有中断信号，则把当前pc里的值保存到EPC里去，再把中断服务例程的首地址装载到pc里，保存相应的cpu状态寄存器，并清除掉中断信号，这样就完成了中断跳转。中断服务程序执行完后，执行eret指令，该指令的作用的是把EPC里的值装入PC，使cpu重新回到当初执行的程序。

在eret返回时，需要把IO的ready位清0，使之能重新接收数据。

```

1  always @(posedge clk or posedge rst)
2      begin
3          if(rst) PC <= 32'h00000000;
4          else
5              begin
6                  if(real_PCwrite) PC <= Mux_PC;
7              end
8          end
9
10 assign Mux_INT_PC = INTPCSource ? INTTable : EPC; //中断开始和中断结束的PC输入
11
12 always @(PCSource or ALUResult or ALUOut or JumpAddr or Mux_INT_PC)
13     begin

```

```

14     case(PCSource)
15         2'b00: Mux_PC = ALUResult;
16         2'b01: Mux_PC = ALUOut;
17         2'b10: Mux_PC = JumpAddr;
18         2'b11: Mux_PC = Mux_INT_PC; //中断处理
19         default: Mux_PC = ALUResult;
20     endcase
21 end
22
23 //控制单元 FSM
24
25 assign real_int = int & enable_int; //中断开关
26 assign init = cont ? CONT : STEP; //单步或连续执行
27 assign end_next = real_int ? INT : init; //指令结束判断是否进入中断处理
28
29 5'b01101: //中断
30     begin
31         ALUSrcA <= 1'b0;
32         ALUSrcB <= 2'b01;
33         ALUOp <= 2'b00; //add
34         INTPCSource <= 1'b1;
35         PCSource <= 2'b11;
36         IRWrite <= 1'b0;
37         MemWrite <= 1'b0;
38         RegWrite <= 1'b0;
39         PCWrite <= 1'b1;
40         PCWriteCond <= 1'b0;
41         PCWriteCond_bne <= 1'b0;
42         EPCWrite <= 1'b1;
43         CauseWrite <= 1'b1;
44         rst_int <= 1'b1; //中断信号清除
45     end
46 5'b01110: //eret
47     begin
48         INTPCSource <= 1'b0;
49         PCSource <= 2'b11;
50         IRWrite <= 1'b0;
51         MemWrite <= 1'b0;
52         RegWrite <= 1'b0;
53         PCWrite <= 1'b1;
54         PCWriteCond <= 1'b0;
55         PCWriteCond_bne <= 1'b0;
56         EPCWrite <= 1'b0;
57         CauseWrite <= 1'b0;
58         rst_ready <= 1'b1; //数据ready状态清0
59     end

```

3.指令增加

```

1 //ALU控制单元, ALUOp
2 2'b10: ALUControlCode = funct[5:0]; //ALU控制码, 逻辑算术运算的控制码由fucnt字段决定
3
4 //ALU
5 localparam SRLV = 6'h6; //逻辑可变右移
6 localparam SLLV = 6'h4; //逻辑可变左移
7 SRLV : result = b >> a[4:0];
8 SLLV : result = b << a[4:0];

```

事实上SRLV和SLLV指令均是RR指令，所以只要在ALU控制码译码时，添加上对应的运算即可。

4.内存映射IO

```

1 MEM mo_mem(clk_cpu, rst, real_we, wd, a_cpu, addr_ddu, mem_data_cpu, mem_data_ddu);
2 CPU mo_cpu(clk_cpu, cont, run, int, rst, data_cpu, addr_ddu[6:2], a_cpu, we, wd,
  reg_data, pc, rst_int, rst_ready, en_tx, start); //
3
4
5 assign overflow = (a_cpu >= 32'h1000) ? 1:0; //cpu访问地址不在Mem内, 则在IO映射区内
6 assign real_we = we & !overflow;
7
8 assign addr_io_byte = a_cpu - 32'h1000; //IO区实际字节地址
9 assign addr_io_word = addr_io_byte >> 2;
10
11
12 assign io_data_cpu = R[addr_io_word];
13 assign data_cpu = overflow ? io_data_cpu : mem_data_cpu; //cpu读取的是IO还是Mem数据
14
15 //IO数据Buffer与IO数据内存相连
16 assign buffer_TX[31:0] = R_TX[0];
17 assign buffer_TX[63:32] = R_TX[1];
18 assign R_RX[0] = buffer_RX[31:0];
19 assign R_RX[1] = buffer_RX[63:32];
20
21 assign R[0] = R_RX[0];
22 assign R[1] = R_RX[1];
23 assign R[2] = R_TX[0];
24 assign R[3] = R_TX[1];
25 assign io_data_ddu = R[addr_ddu[31:2]];
26
27 //写入IO数据
28 always @(posedge clk_cpu)
29 begin
30     if(we && overflow) R_TX[addr_io_word - 2] = wd;
31 end

```

本实验中，我把IO的起始地址设为0x1000，这样刚好与主存连接上了。只要cpu访问的地址超过0x1000，就会访问到IO数据区。这样可以用lw和sw指令直接访问IO数据。

5.HC-06收发数据

- 数据接收

```
1 //蓝牙芯片波特率9600 8N1, 数据接收, UART协议, 8位数据帧, 无奇偶校验, 1数据终止位。16倍频采样
2 always@(posedge clk_153600hz)
3     begin
4         if(RX == 1'b0 && continue == 1'b0 && !receiving)
5             begin
6                 continue = 1;
7                 c = 1;
8             end
9         else if(continue)
10            begin
11                if(RX == 0) c = c + 1;
12                else continue = 0;
13                if(c == 0)
14                    begin
15                        receiving = 1;
16                        continue = 0;
17                        cyc = 1;
18                        b = 0;
19                    end
20            end
21        else if(receiving == 1 && cyc==0 )
22            begin
23                if(b==4'b1000) receiving = 0;
24                cyc = cyc + 1;
25                n[b] = RX;
26                b = b + 1;
27                if(b == 4'b1000)
28                    begin
29                        flag = 1;
30                        frame = n;
31                    end
32            end
33        else
34            begin
35                flag = 0;
36                cyc = cyc + 1;
37            end
38    end
39
40 //中断信号产生与清除
41 always@(posedge valid_frame or posedge rst_int or posedge rst_ready or posedge rst)
42     begin
43         if(rst_int || rst)
44             begin
45                 cnt = 0;
46                 int = 0;
47                 ready = 0;
48             end
49         else if(rst_ready)
```

```

50     begin
51         ready = 0;
52         cnt = 0;
53     end
54     else if(!ready && valid_frame)
55     begin
56         //buffer_RX = buffer_RX >> 8;
57         //buffer_RX[63:56] = frame;
58         buffer_RX = buffer_RX << 8;
59         buffer_RX[7:0] = frame;
60         cnt = cnt + 1;
61         if(cnt == 0)
62         begin
63             ready = 1;
64             int = 1;
65         end
66     end
67 end

```

这里值得说明的一点是，开始我直接用9600的频率去采样，结果误码率十分高。后来我使用16倍频时钟采样，大大降低了误码率。

在缓冲区填满后，会发出中断信号、数据ready信号，在中断处理完后，中断信号与ready信号都被清0。

- 数据发送

```

1  always@(posedge clk_9600hz)
2      begin
3          if(en_tx && !traning)
4          begin
5              low = 6'd0;
6              cyc_tx = 0;
7              traning = 1;
8              cnt = 0;
9              gap = 1'b1;
10         end
11         else if(traning && gap)
12         begin
13             if(cnt == 2'b1)
14             begin
15                 TX = 0;
16                 gap = 0;
17                 low = low - 6'd8;
18             end
19             else TX = 1;
20             cnt = cnt + 1'b1;
21         end
22         else if(traning)
23         begin
24             TX = buffer_TX[low + cyc_tx];
25             cyc_tx = cyc_tx + 1;
26             if(cyc_tx == 3'b0) gap = 1'b1;

```

```

27         if(low == 6'b0 && cyc_tx == 3'b0)
28             begin
29                 traning = 0;
30             end
31         end
32     else
33         begin
34             TX = 1;
35         end
36     end

```

这里一次一共发送8个数据帧，即64位数据，两个整数。

cpu在开始会产生发送使能信号，该IO模块检测到en_tx信号后，即把en_tx信号清除，然后再进行发送。

应用算法

本实验主要是求两个数的最大公因数。

算法是辗转相减法。

该算法基于以下事实：

GCD(x,y)=x (x==y) GCD(x,y)=2*(GCD(x/2,y/2)) (!!(x&1) and !(y&1)) GCD(x,y)=GCD(x/2,y) (!(x&1) and (y&1))(因为 2 显然不是公因数，所以我们可以果断地筛掉它) GCD(x,y)=GCD(x,y/2) ((x&1) and !(y&1))理由同上
GCD(x,y)=GCD(x-y,y) (辗转相减)

通过有限次的循环就可以求出两个数的最大公因数了。

代码会在附录里给出。

仿真/下载

```

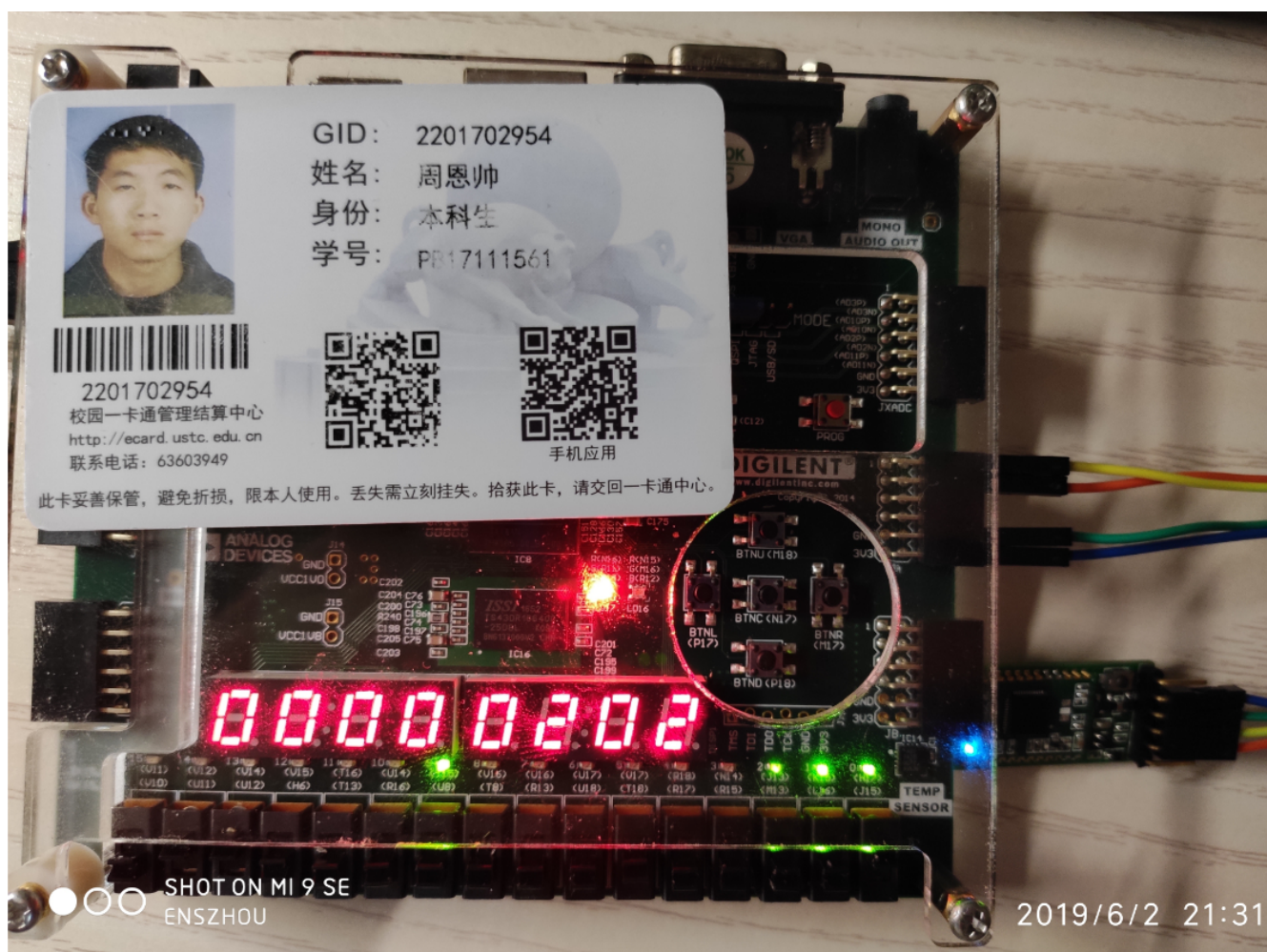
1  module CPU_MEM_tb(
2      );
3
4      reg clk, rst, mode_cpu, cont, run, v;
5      wire [31:0] mem_data, reg_data, pc;
6      reg [31:0] addr;
7      reg [7:0] frame;
8
9      CPU_MEM mo_cpu_mem_tb(clk, cont, run, rst, addr, pc, mem_data, reg_data, v, frame);
10
11
12      initial
13      begin
14          v <= 0;
15          frame <= 8'b01010101;
16          clk <= 0;
17          run <= 1;

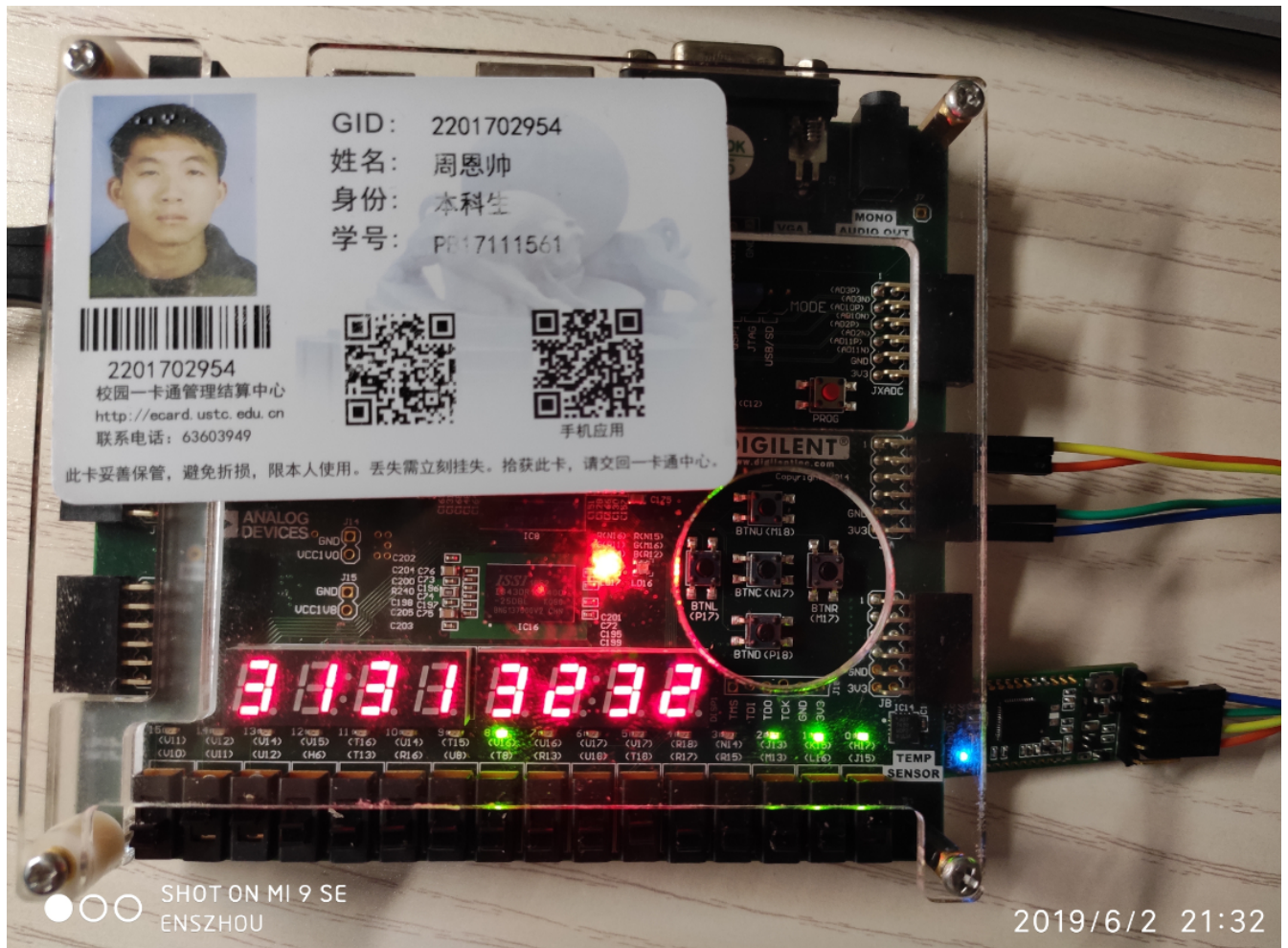
```

```

18         rst <= 0;
19         cont <= 1;
20         addr <= 32'h8;
21         #10 rst <= 1;
22         #4 rst <= 0;
23         #500 mode_cpu <= 0;
24         #400 addr <= 32'h4;
25     end
26
27     always
28     forever #1 clk = ~clk;
29
30     always
31     forever #25 v <= ~v;
32
33
34 endmodule

```





上述求0x11223344和0x55667788的最大公因数为0x44

根据DDU以及手机调试，以及下载结果，可以看出结果是对的。

实验总结

本次实验我学会UART通信协议，学会了多周期cpu的中断设计，学会了IO内存映射。

附录

- 最大公因数汇编代码

```
1 .text
2 _start:
3     lw $t1 0x1000($0)
4     lw $t2 0x1004($0)
5
6     beq $t1 $0 _check_mem
7
8     andi $s0 $s0 0
9     addi $s1 $s0 1
```

```

10
11 _loop:
12     beq $t2 $0 _end
13     beq $t1 $0 _t1_equal_0
14     beq $t1 $t2 _end
15
16     andi $t3 $t1 1
17     andi $t4 $t2 1
18     or $t5 $t3 $t4
19     beq $t5 $0 _t1_t2_even
20
21     beq $t3 $0 _t1_nt2_even
22     beq $t4 $0 _nt1_t2_even
23
24     slt $t6 $t1 $t2
25     beq $t6 $0 _t1_large_t2
26
27     sub $t2 $t2 $t1                # t1 < t2
28     beq $0 $0 _loop
29
30     _t1_large_t2:
31         sub $t1 $t1 $t2 # t1 > t2
32         beq $0 $0 _loop
33
34
35
36 _t1_nt2_even:
37     srlv $t1 $t1 $s1
38     beq $0 $0 _loop
39
40 _nt1_t2_even:
41     srlv $t2 $t2 $s1
42     beq $0 $0 _loop
43
44 _t1_t2_even:
45     addi $s0 $s0 1
46     srlv $t1 $t1 $s1
47     srlv $t2 $t2 $s1
48     beq $0 $0 _loop
49
50
51 _t1_equal_0:
52     add $t1 $t2 $0
53
54 _end:
55     sllv $t0 $t1 $s0
56     sw $t0 0x1008($0)
57     eret
58
59 _check_mem:
60     lw $t0 0($t2)
61     sw $t0 0x1008($0)
62     sw $t2 0x100c($0)

```

- 硬件设计代码

```

1  module Top(
2      input cont,
3      input step,
4      input [2:0] mode_seg,
5      input inc,
6      input dec,
7      input rst,
8      input CLK100MHZ,
9      input RX,
10     output TX,
11     output mode_kernel,
12     output mode_user,
13     output tag_frame,
14     output [15:0] led,
15     output [6:0] seg,
16     output [7:0] an,
17     output dp
18 );
19 wire clk_8mhz, clk_cpu, clk_2mhz, clk_9600hz, clk_9600khz, clk_153600hz, clk_19_2mhz;
20 wire run;
21 wire [31:0] pc, mem_data, reg_data, addr, io_data;
22 wire [15:0] display;
23 wire [7:0] frame;
24 wire [1:0] mode;
25 wire valid_frame, ready, en_tx, start;
26 wire [63:0] buffer;
27
28 assign dp = display[15];
29 assign seg = display[14:8];
30 assign an = display[7:0];
31 assign {mode_kernel, mode_user} = mode;
32 assign tag_frame = valid_frame | ready;
33
34 //assign clk_cpu = clk_2mhz & run;
35
36 clk_wiz_0 mo_clk_wiz_0(clk_8mhz, clk_19_2mhz, CLK100MHZ);
37 CLK #(4) mo_clk_2m(clk_8mhz, 0, 1, clk_2mhz);
38 CLK #(2000) mo_clk_9600(clk_19_2mhz, 0, 1, clk_9600hz);
39 CLK #(125) mo_clk_153600(clk_19_2mhz, 0, 1, clk_153600hz);
40
41 DDU mo_ddu(cont, step, mode_seg, inc, dec, pc, mem_data, reg_data, frame, valid_frame,
io_data, rst, clk_8mhz, clk_2mhz, run, addr, led, display, mode);
42 CPU_MEM mo_cpu_mem(clk_2mhz, cont, run, rst, addr, pc, mem_data, reg_data,
io_data, valid_frame, frame, ready, en_tx, buffer, start);
43 HC_42
mo_hc_42(clk_9600hz, clk_153600hz, RX, en_tx, buffer, TX, start, valid_frame, frame);
44

```

```

45 endmodule
46
47 module CLK #(parameter N = 5000000)(
48     input clk_xhz,
49     input rst,
50     input enable,
51     output reg Q
52 );
53     reg [29:0] cnt;
54     wire [29:0] n;
55
56     assign n = N >> 1;
57
58     always @ (posedge clk_xhz or posedge rst)
59     begin
60         if(rst)
61         begin
62             Q <= 0;
63             cnt <= 30'b0;
64         end
65         else
66         begin
67             if(enable)
68             begin
69                 if(cnt >= n)
70                 begin
71                     Q <= ~Q;
72                     cnt <= 30'b0;
73                 end
74                 else cnt <= cnt + 30'b1;
75             end
76         end
77     end
78 endmodule
79
80 //DDU在上次报告里有，且本次实验未作任何改动，故这里不再附上
81 module DDU(
82     input cont,
83     input step,
84     input [2:0] mode_seg,
85     input inc,
86     input dec,
87     input [31:0] pc,
88     input [31:0] mem_data,
89     input [31:0] reg_data,
90     input [7:0] frame,
91     input valid_frame,
92     input [31:0] io_data,
93     input rst,
94     input clk_8mhz,
95     input clk_cpu,
96     output reg run,
97     output reg [31:0] addr,

```

```

98     output [15:0] led,
99     output [15:0] display,
100    output [1:0] mode
101 );
102
103 wire [6:0] seg;
104 wire [7:0] AN;
105 wire DP;
106 wire clk_4khz, inc_sin, dec_sin, step_sin;
107 reg [3:0] x0, x1, x2, x3, x4, x5, x6, x7;
108
109
110 assign display = {DP, seg, AN};
111 assign led[7:0] = pc[10:2];
112 assign led[15:8] = addr[10:2];
113 assign mode = (pc >= 32'h00000040) ? 2'b01 : 2'b10;
114
115 CLK #(2000) mo_clk_4k(clk_8mhz, 0, 1, clk_4khz);
116 SEG mo_seg(clk_4khz, 8'b11111111, 8'b00000000, x0, x1, x2, x3, x4, x5, x6, x7, AN, DP, seg);
117 MUL_SIN mo_mul_sin_inc(clk_cpu, inc, rst, inc_sin);
118 MUL_SIN mo_mul_sin_dec(clk_cpu, dec, rst, dec_sin);
119 MUL_SIN mo_mul_sin_step(clk_cpu, step, rst, step_sin);
120
121 always@(mode_seg or mem_data or reg_data or frame)
122 begin
123     if(mode_seg == 3'b1)
124     begin
125         x0 = mem_data[3:0];
126         x1 = mem_data[7:4];
127         x2 = mem_data[11:8];
128         x3 = mem_data[15:12];
129         x4 = mem_data[19:16];
130         x5 = mem_data[23:20];
131         x6 = mem_data[27:24];
132         x7 = mem_data[31:28];
133     end
134     else if(mode_seg == 3'b0)
135     begin
136         x0 = reg_data[3:0];
137         x1 = reg_data[7:4];
138         x2 = reg_data[11:8];
139         x3 = reg_data[15:12];
140         x4 = reg_data[19:16];
141         x5 = reg_data[23:20];
142         x6 = reg_data[27:24];
143         x7 = reg_data[31:28];
144     end
145     else if(mode_seg == 3'b10)
146     begin
147         x0 = frame[0];
148         x1 = frame[1];
149         x2 = frame[2];
150         x3 = frame[3];

```

```

151         x4 = frame[4];
152         x5 = frame[5];
153         x6 = frame[6];
154         x7 = frame[7];
155     end
156     else
157     begin
158         x0 = io_data[3:0];
159         x1 = io_data[7:4];
160         x2 = io_data[11:8];
161         x3 = io_data[15:12];
162         x4 = io_data[19:16];
163         x5 = io_data[23:20];
164         x6 = io_data[27:24];
165         x7 = io_data[31:28];
166     end
167 end
168
169 always@(posedge clk_cpu or posedge rst)
170 begin
171     if(rst)
172     begin
173         addr <= 32'b0;
174     end
175     else
176     begin
177         if(inc_sin) addr <= addr + 4;
178         else if(dec_sin) addr <= addr - 4;
179     end
180 end
181
182 always@(cont or clk_cpu or step_sin)
183 begin
184     if(cont) run = 1'b1;
185     else run = step_sin;
186 end
187
188 endmodule
189
190 module SEG(
191     input clk_4khz,
192     input [7:0] en, //pos
193     input [7:0] dp_x, //pos
194     input [3:0] x0,x1,x2,x3,x4,x5,x6,x7,
195     output [7:0] AN,
196     output DP,
197     output reg [6:0] seg
198 );
199
200
201 reg [2:0] state,next_state;
202 reg [7:0] an;
203 reg dp;

```

```

204     wire [6:0] seg0, seg1, seg2, seg3, seg4, seg5, seg6, seg7;
205
206     bcd_to_seg b0(x0, seg0);
207     bcd_to_seg b1(x1, seg1);
208     bcd_to_seg b2(x2, seg2);
209     bcd_to_seg b3(x3, seg3);
210     bcd_to_seg b4(x4, seg4);
211     bcd_to_seg b5(x5, seg5);
212     bcd_to_seg b6(x6, seg6);
213     bcd_to_seg b7(x7, seg7);
214
215     assign AN = an | ~en;
216     assign DP = ~dp;
217
218     always @ (posedge clk_4khz)
219         begin
220             state <= next_state;
221         end
222
223     always @(state)
224     begin
225         case(state)
226             3'b000: next_state = 3'b001;
227             3'b001: next_state = 3'b010;
228             3'b010: next_state = 3'b011;
229             3'b011: next_state = 3'b100;
230             3'b100: next_state = 3'b101;
231             3'b101: next_state = 3'b110;
232             3'b110: next_state = 3'b111;
233             3'b111: next_state = 3'b000;
234             default: next_state = 3'b000;
235         endcase
236     end
237
238
239     always @ (posedge clk_4khz)
240         begin
241             case(state)
242                 3'b000:
243                     begin
244                         an = 8'b11111110;
245                         dp = dp_x[0];
246                         seg = seg0;
247                     end
248                 3'b001:
249                     begin
250                         an = 8'b11111101;
251                         dp = dp_x[1];
252                         seg = seg1;
253                     end
254                 3'b010:
255                     begin
256                         an = 8'b11111011;

```

```

257         dp = dp_x[2];
258         seg = seg2;
259     end
260     3'b011:
261     begin
262         an = 8'b11110111;
263         dp = dp_x[3];
264         seg = seg3;
265     end
266     3'b100:
267     begin
268         an = 8'b11101111;
269         dp = dp_x[4];
270         seg = seg4;
271     end
272     3'b101:
273     begin
274         an = 8'b11011111;
275         dp = dp_x[5];
276         seg = seg5;
277     end
278     3'b110:
279     begin
280         an = 8'b10111111;
281         dp = dp_x[6];
282         seg = seg6;
283     end
284     3'b111:
285     begin
286         an = 8'b01111111;
287         dp = dp_x[7];
288         seg = seg7;
289     end
290     default:
291     begin
292         an = 8'b11111110;
293         seg = seg0;
294         dp = dp_x[0];
295     end
296     endcase
297 end
298 endmodule
299
300 module bcd_to_seg(
301     input [3:0] x,
302     output reg [6:0] seg);
303     always @ (x)
304     begin
305         case(x)
306             4'b0000: seg = 7'b1000000;
307             4'b0001: seg = 7'b1111001;
308             4'b0010: seg = 7'b0100100;
309             4'b0011: seg = 7'b0110000;

```



```

310         4'b0100:seg = 7'b0011001;
311         4'b0101:seg = 7'b0010010;
312         4'b0110:seg = 7'b0000010;
313         4'b0111:seg = 7'b1111000;
314         4'b1000:seg = 7'b0000000;
315         4'b1001:seg = 7'b0010000;
316         4'b1010:seg = 7'b0001000;
317         4'b1011:seg = 7'b0000011;
318         4'b1100:seg = 7'b0100111;
319         4'b1101:seg = 7'b0100001;
320         4'b1110:seg = 7'b0000110;
321         4'b1111:seg = 7'b0001110;
322         default:seg = 7'b1000000;
323     endcase
324 end
325 endmodule
326
327 module MUL_SIN(
328     input clk,
329     input s,
330     input rst,
331     output reg q
332 );
333     reg last;
334
335     always @(negedge clk or posedge rst)
336     begin
337         if(rst)
338             begin
339                 last <= 0;
340             end
341         else
342             begin
343                 if(!last && s)
344                     begin
345                         q <= 1'b1;
346                         last <= 1'b1;
347                     end
348                 else if(last && s) q <= 1'b0;
349                 else if(last && !s)
350                     begin
351                         q <= 1'b0;
352                         last <= 1'b0;
353                     end
354                 else q <= 1'b0;
355             end
356         end
357     endmodule
358
359 module HC_42(
360     input clk_9600hz,
361     input clk_153600hz,
362     input RX,

```

```

363     input en_tx,
364     input [63:0] buffer_TX,
365     output reg TX,
366     output start,
367     output reg flag,
368     output reg [7:0] frame
369 );
370 reg receiving, traning, gap;
371 reg [3:0] b;
372 reg [7:0] n;
373 reg [2:0] cyc_tx;
374 reg [5:0] low;
375 reg cnt;
376 wire [31:0] data;
377 reg [2:0] c;
378 reg continue;
379 reg[3:0] cyc;
380
381 assign start = traning;
382
383 always@(posedge clk_153600hz)
384 begin
385     if(RX == 1'b0 && continue == 1'b0 && !receiving)
386     begin
387         continue = 1;
388         c = 1;
389     end
390     else if(continue)
391     begin
392         if(RX == 0) c = c + 1;
393         else continue = 0;
394         if(c == 0)
395         begin
396             receiving = 1;
397             continue = 0;
398             cyc = 1;
399             b = 0;
400         end
401     end
402     else if(receiving == 1 && cyc==0 )
403     begin
404         if(b==4'b1000) receiving = 0;
405         cyc = cyc + 1;
406         n[b] = RX;
407         b = b + 1;
408         if(b == 4'b1000)
409         begin
410             flag = 1;
411             frame = n;
412         end
413     end
414     else
415     begin

```

```

416         flag = 0;
417         cyc = cyc + 1;
418     end
419 end
420
421 always@(posedge clk_9600hz)
422 begin
423     if(en_tx && !traning)
424     begin
425         low = 6'd0;
426         cyc_tx = 0;
427         traning = 1;
428         cnt = 0;
429         gap = 1'b1;
430     end
431     else if(traning && gap)
432     begin
433         if(cnt == 2'b1)
434         begin
435             TX = 0;
436             gap = 0;
437             low = low - 6'd8;
438         end
439         else TX = 1;
440         cnt = cnt + 1'b1;
441     end
442     else if(traning)
443     begin
444         TX = buffer_TX[low + cyc_tx];
445         cyc_tx = cyc_tx + 1;
446         if(cyc_tx == 3'b0) gap = 1'b1;
447         if(low == 6'b0 && cyc_tx == 3'b0)
448         begin
449             traning = 0;
450         end
451     end
452     else
453     begin
454         TX = 1;
455     end
456 end
457 endmodule
458
459 module CPU_MEM(
460     input clk_2mhz,
461     input cont,
462     input run,
463     input rst,
464     input [31:0] addr_ddu,
465     output [31:0] pc,mem_data_ddu,reg_data,io_data_ddu,
466     input valid_frame,
467     input [7:0] frame,
468     output reg ready,

```

```

469     output en_tx,
470     output [63:0]buffer_TX,
471     input start
472 );
473
474     wire clk_cpu, we,rst_int, rst_ready;
475     wire [31:0] a_cpu,wd,mem_data_cpu,data_cpu;
476     wire [31:0] io_data_cpu;
477     reg [2:0] cnt;
478     reg int;
479     reg [63:0] buffer_RX;
480     wire [31:0] R[3:0];
481     wire [31:0] R_RX[1:0];
482     reg [31:0] R_TX[1:0];
483
484     wire overflow,real_we;
485     wire [31:0] addr_io_byte,addr_io_word;
486
487     MEM mo_mem(clk_cpu, rst, real_we, wd, a_cpu, addr_ddu, mem_data_cpu,
mem_data_ddu);
488     CPU mo_cpu(clk_cpu, cont, run, int, rst, data_cpu, addr_ddu[6:2], a_cpu, we, wd,
reg_data, pc, rst_int, rst_ready,en_tx,start);//
489
490     assign clk_cpu = clk_2mhz;
491
492     assign overflow = (a_cpu >= 32'h1000) ? 1:0;
493     assign real_we = we & !overflow;
494
495     assign addr_io_byte = a_cpu - 32'h1000;
496     assign addr_io_word = addr_io_byte >> 2;
497
498
499     assign io_data_cpu = R[addr_io_word];
500     assign data_cpu = overflow ? io_data_cpu : mem_data_cpu;
501
502     assign buffer_TX[31:0] = R_TX[0];
503     assign buffer_TX[63:32] = R_TX[1];
504     assign R_RX[0] = buffer_RX[31:0];
505     assign R_RX[1] = buffer_RX[63:32];
506
507     assign R[0] = R_RX[0];
508     assign R[1] = R_RX[1];
509     assign R[2] = R_TX[0];
510     assign R[3] = R_TX[1];
511     assign io_data_ddu = R[addr_ddu[31:2]];
512
513     always @(posedge clk_cpu)
514     begin
515         if(we && overflow) R_TX[addr_io_word - 2] = wd;
516     end
517
518

```

```

519     always@(posedge valid_frame or posedge rst_int or posedge rst_ready or posedge
rst)
520     begin
521         if(rst_int || rst)
522             begin
523                 cnt = 0;
524                 int = 0;
525                 ready = 0;
526             end
527         else if(rst_ready)
528             begin
529                 ready = 0;
530                 cnt = 0;
531             end
532         else if(!ready && valid_frame)
533             begin
534                 buffer_RX = buffer_RX << 8;
535                 buffer_RX[7:0] = frame;
536                 cnt = cnt + 1;
537                 if(cnt == 0)
538                     begin
539                         ready = 1;
540                         int = 1;
541                     end
542             end
543         end
544     end
545 endmodule
546
547 module MEM(
548     input clk,
549     input rst,
550     input we, //cpu使能
551     input [31:0]wd, //cpu写数据
552     input [31:0]a_cpu, //cpu 地址
553     input [31:0]ra_ddu, //ddu读地址
554     output [31:0]rd_cpu,
555     output [31:0]rd_ddu
556 );
557
558 dist_mem_gen_0    mo_dist_mem ( //256*32
559     .a(a_cpu[31:2]),          // input wire [9 : 0] a
560     .d(wd),                   // input wire [31 : 0] d
561     .dpra(ra_ddu[31:2]),      // input wire [9 : 0] dpra
562     .clk(clk),                // input wire clk
563     .we(we),                  // input wire we
564     .spo(rd_cpu),
565     .dpo(rd_ddu)              // output wire [31 : 0] dpo
566 );
567
568 endmodule
569
570 module CPU(

```

```

571     input clk,
572     input cont,
573     input run,
574     input int,
575     input rst,
576     input [31:0] MemData,
577     input [4:0] RegNum,
578     output [31:0] MemAddr,
579     output MemWrite_out,
580     output [31:0] MemWriteData,
581     output [31:0] RegData,
582     output [31:0] PC_out,
583     output rst_int, rst_ready,en_tx,
584     input start
585 );
586
587     wire [5:0] opcode;
588     wire [1:0] ALUSrcB,ALUOp,PCSource;
589     wire
INTPCSource,ALUSrcA,IorD,IRwrite,PCwrite,PCwriteCond,MemWrite,RegDst,RegWrite,MemtoR
eg,PCwriteCond_bne,EPCwrite,Causewrite;
590     wire enable_int;
591     wire real_PCwrite;
592
593
594     //some registers
595     reg [31:0] PC, EPC;
596     reg [31:0] ALUOut;
597     reg [31:0] A,B;
598     reg [31:0] MDR;
599     reg [31:0] IR;
600
601     //ALU port
602     wire zero;
603     wire [31:0] ALUResult;
604     wire [5:0] ALUControlCode;
605
606     //Register port
607     //wire [4:0] RegReadNum1,RegReadNum2;
608     wire [31:0] RegReadData1, RegReadData2;
609
610     //some muxes
611     wire [31:0] Mux_MemAddr, Mux_RegWriteData, Mux_ALU_A,Mux_INT_PC;
612     reg [31:0] Mux_ALU_B, Mux_PC;
613     wire [4:0] Mux_RegWriteNum;
614
615     //Instruction extend
616     wire [31:0] Ins_16Signed32, Ins_16Signed32_noSL, Ins_16Signed32_SL;
617     wire [15:0] sign;
618     wire [27:0] Ins_26to28;
619     wire [31:0] JumpAddr;
620
621     localparam INTTable = 32'h00000040;

```

```

622
623     Reg_File mo_reg_file(IR[25:21], IR[20:16], RegNum, Mux_RegWriteNum,
Mux_RegWriteData, RegWrite, rst, clk, RegReadData1, RegReadData2, RegData);
624     ALUControlUnit mo_alucu(IR[5:0], ALUOp , opcode, ALUControlCode);
625     ALU mo_alu(Mux_ALU_A, Mux_ALU_B, ALUControlCode, ALUResult, Zero);
626     ControlUnit
mo_control_unit(clk, cont, run, int, enable_int, rst, opcode, rst_int, rst_ready, en_tx, start
, ALUSrcB, ALUOp, PCSource, INTPCSource, ALUSrcA, IorD, IRWrite, PCWrite, PCWriteCond, MemWrite
e, RegDst, RegWrite, MemtoReg, PCWriteCond_bne, EPCWrite, CauseWrite);
627     assign opcode = IR[31:26];
628     //output
629     assign MemAddr = Mux_MemAddr;
630     assign MemWriteData = B;
631     assign MemWrite_out = MemWrite;
632     assign PC_out = PC;
633
634     //singal
635     assign real_PCWrite = PCWrite | (PCWriteCond & Zero) | (PCWriteCond_bne &
~Zero);
636     assign enable_int = 1'b1;
637
638     //Instruction extend
639     assign sign = IR[15] ? 16'hffff : 16'h0000;
640     assign Ins_16Signed32 = {sign, IR[15:0]};
641     assign Ins_16Signed32_noSL = Ins_16Signed32;
642     assign Ins_16Signed32_SL = Ins_16Signed32 << 2;
643     assign Ins_26to28 = {IR[25:0], 2'b00};
644     assign JumpAddr = {PC[31:28], Ins_26to28};
645
646     //MUX
647     assign Mux_MemAddr = IorD ? ALUOut : PC;
648     assign Mux_RegWriteData = MemtoReg ? MDR : ALUOut;
649     assign Mux_RegWriteNum = RegDst ? IR[15:11] : IR[20:16];
650     assign Mux_ALU_A = ALUSrcA ? A : PC;
651     assign Mux_INT_PC = INTPCSource ? INTTable : EPC;
652
653     //PC
654     always @(posedge clk or posedge rst)
655     begin
656         if(rst) PC <= 32'h00000000;
657         else
658         begin
659             if(real_PCWrite) PC <= Mux_PC;
660         end
661     end
662
663     //EPC
664     always @(posedge clk)
665     begin
666         //if(EPCWrite) EPC <= ALUResult;
667         if(EPCWrite) EPC <= PC;          //
668     end
669

```

```

670 //MDR
671 always @(posedge clk)
672 begin
673     MDR <= MemData;
674     A <= RegReadData1;
675     B <= RegReadData2;
676     ALUOut <= ALUResult;
677 end
678
679 //IR
680 always @(posedge clk)
681 begin
682     if(IRWrite) IR <= MemData;
683 end
684
685 //Mux_ALU_B 4-way
686 always @(ALUSrcB or B or Ins_16Signed32_noSL or Ins_16Signed32_SL)
687 begin
688     case(ALUSrcB)
689         2'b00: Mux_ALU_B = B;
690         2'b01: Mux_ALU_B = 32'd4;
691         2'b10: Mux_ALU_B = Ins_16Signed32_noSL;
692         2'b11: Mux_ALU_B = Ins_16Signed32_SL;
693         default: Mux_ALU_B = B;
694     endcase
695 end
696
697 //Mux_PC 3-way
698 always @(PCSource or ALUResult or ALUOut or JumpAddr or Mux_INT_PC)
699 begin
700     case(PCSource)
701         2'b00: Mux_PC = ALUResult;
702         2'b01: Mux_PC = ALUOut;
703         2'b10: Mux_PC = JumpAddr;
704         2'b11: Mux_PC = Mux_INT_PC; //中断处理
705         default: Mux_PC = ALUResult;
706     endcase
707 end
708
709 endmodule
710
711 module ControlUnit(
712     input clk,
713     input cont,
714     input run,
715     input int,
716     input enable_int,
717     input rst,
718     input [5:0] opcode,
719     output reg rst_int, rst_ready, en_tx,
720     input start,
721     output reg [1:0] ALUSrcB, ALUOp, PCSource,

```



```

722     output reg
INTPCSource,ALUSrcA,IorD,IRWrite,PCWrite,PCWriteCond,MemWrite,RegDst,RegWrite,MemtoR
eg,PCWriteCond_bne,EPCWrite,CauseWrite
723 );
724
725     localparam CONT = 5'b00000;
726     localparam STEP = 5'b01111;
727     localparam INT = 5'b01101;
728
729     wire [3:0] init,end_next;
730     wire real_int;
731
732     assign real_int = int & enable_int;
733     assign init = cont ? CONT : STEP;
734     assign end_next = real_int ? INT : init;
735
736     reg [3:0] state, nextstate;
737     reg flag;
738     //reg [1:0] ALUSrcB,ALUOp,PCSource;
739     //reg
ALUSrcA,IorD,IRWrite,PCWrite,PCWriteCond,MemWrite,RegDst,RegWrite,MemtoReg;
740
741     always@(posedge clk or posedge rst)
742     begin
743         if(rst)
744             begin
745                 state <= init;
746             end
747         else
748             begin
749                 state <= nextstate;
750             end
751     end
752
753     always@(state or opcode or run or end_next or init)
754     begin
755         case(state)
756             5'b00000:nextstate = 5'b00001;
757             5'b00001:
758                 begin
759                     if(opcode == 6'h23 || opcode == 6'h2b) nextstate = 5'b00010;
760                     else if(opcode == 6'h4)nextstate = 5'b01000;
761                     else if(opcode == 6'h2)nextstate = 5'b01001;
762                     else if(opcode == 6'h5)nextstate = 5'b01010;
763                     else if(opcode == 6'h0)nextstate = 5'b00110; //RR
764                     else if(opcode == 6'h10)nextstate = 5'b01110;
765                     else nextstate = 5'b01011; //RI
766                 end
767             5'b00010:
768                 begin
769                     if(opcode == 6'h23) nextstate = 5'b00011;
770                     else nextstate = 5'b00101;
771                 end

```

```

772         5'b00011: nextstate = 5'b00100;
773         5'b00100: nextstate = end_next;
774         5'b00101: nextstate = end_next;
775         5'b00110: nextstate = 5'b00111;
776         5'b00111: nextstate = end_next;
777         5'b01000: nextstate = end_next;
778         5'b01001: nextstate = end_next;
779         5'b01010: nextstate = end_next;
780         5'b01011: nextstate = 5'b01100; //compute
781         5'b01100: nextstate = end_next; //reg write
782         5'b01101: nextstate = init; //中断
783         5'b01110: nextstate = init; //eret
784         5'b01111: if(run) nextstate = 5'b00000;
785                 else nextstate = 5'b01111;
786     default: nextstate = init;
787 endcase
788
789 end
790
791 always @(posedge clk or posedge rst)
792 begin
793     //if(clk)begin
794         if(rst)
795         begin
796             ALUSrcA <= 1'b0;
797             ALUSrcB <= 2'b01;
798             ALUOp <= 2'b00;
799             IorD <= 1'b0;
800             PCSrc <= 2'b00;
801             IRWrite <= cont;
802             MemWrite <= 1'b0;
803             RegWrite <= 1'b0;
804             PCWrite <= cont;
805             PCWriteCond <= 1'b0;
806             PCWriteCond_bne <= 1'b0;
807             EPCWrite <= 1'b0;
808             CauseWrite <= 1'b0;
809             rst_int <= 1'b0;
810             rst_ready <= 1'b0;
811         end
812     else
813     begin
814         case(nextstate)
815             5'b00000:
816                 begin
817                     ALUSrcA <= 1'b0;
818                     ALUSrcB <= 2'b01;
819                     ALUOp <= 2'b00;
820                     IorD <= 1'b0;
821                     PCSrc <= 2'b00;
822                     IRWrite <= 1'b1;
823                     MemWrite <= 1'b0;
824                     RegWrite <= 1'b0;

```

```

825         PCWrite <= 1'b1;
826         PCWriteCond <= 1'b0;
827         PCWriteCond_bne <= 1'b0;
828         EPCWrite <= 1'b0;
829         CauseWrite <= 1'b0;
830         rst_int <= 1'b0;
831         rst_ready <= 1'b0;
832     end
833 5'b00001:
834     begin
835         ALUSrcA <= 1'b0;
836         ALUSrcB <= 2'b11;
837         ALUOp <= 2'b00;
838         IRWrite <= 1'b0;
839         MemWrite <= 1'b0;
840         RegWrite <= 1'b0;
841         PCWrite <= 1'b0;
842         PCWriteCond <= 1'b0;
843         PCWriteCond_bne <= 1'b0;
844         EPCWrite <= 1'b0;
845         CauseWrite <= 1'b0;
846     end
847 5'b00010:
848     begin
849         ALUSrcA <= 1'b1;
850         ALUSrcB <= 2'b10;
851         ALUOp <= 2'b00;
852         IRWrite <= 1'b0;
853         MemWrite <= 1'b0;
854         RegWrite <= 1'b0;
855         PCWrite <= 1'b0;
856         PCWriteCond <= 1'b0;
857         PCWriteCond_bne <= 1'b0;
858         EPCWrite <= 1'b0;
859         CauseWrite <= 1'b0;
860     end
861 5'b00011:
862     begin
863         IorD <= 1'b1;
864         IRWrite <= 1'b0;
865         MemWrite <= 1'b0;
866         RegWrite <= 1'b0;
867         PCWrite <= 1'b0;
868         PCWriteCond <= 1'b0;
869         PCWriteCond_bne <= 1'b0;
870         EPCWrite <= 1'b0;
871         CauseWrite <= 1'b0;
872     end
873 5'b00100:
874     begin
875         RegDst <= 1'b0;
876         MemtoReg <= 1'b1;
877         IRWrite <= 1'b0;

```

```

878         MemWrite <= 1'b0;
879         RegWrite <= 1'b1;
880         PCWrite  <= 1'b0;
881         PCWriteCond <= 1'b0;
882         PCWriteCond_bne <= 1'b0;
883         EPCWrite <= 1'b0;
884         CauseWrite <= 1'b0;
885     end
886 5'b00101:
887     begin
888         IorD <= 1'b1;
889         IRWrite <= 1'b0;
890         MemWrite <= 1'b1;
891         RegWrite <= 1'b0;
892         PCWrite  <= 1'b0;
893         PCWriteCond <= 1'b0;
894         PCWriteCond_bne <= 1'b0;
895         EPCWrite <= 1'b0;
896         CauseWrite <= 1'b0;
897     end
898 5'b00110:
899     begin
900         ALUSrcA <= 1'b1;
901         ALUSrcB <= 2'b00;
902         ALUOp  <= 2'b10;
903         IRWrite <= 1'b0;
904         MemWrite <= 1'b0;
905         RegWrite <= 1'b0;
906         PCWrite  <= 1'b0;
907         PCWriteCond <= 1'b0;
908         PCWriteCond_bne <= 1'b0;
909         EPCWrite <= 1'b0;
910         CauseWrite <= 1'b0;
911     end
912 5'b00111:
913     begin
914         RegDst <= 1'b1;
915         MemtoReg <= 1'b0;
916         IRWrite <= 1'b0;
917         MemWrite <= 1'b0;
918         RegWrite <= 1'b1;
919         PCWrite  <= 1'b0;
920         PCWriteCond <= 1'b0;
921         PCWriteCond_bne <= 1'b0;
922         EPCWrite <= 1'b0;
923         CauseWrite <= 1'b0;
924     end
925 5'b01000:
926     begin
927         ALUSrcA <= 1'b1;
928         ALUSrcB <= 2'b00;
929         ALUOp  <= 2'b01;
930         PCSrc <= 2'b01;

```

```

931         IRWrite <= 1'b0;
932         MemWrite <= 1'b0;
933         RegWrite <= 1'b0;
934         PCWrite <= 1'b0;
935         PCWriteCond <= 1'b1;
936         PCWriteCond_bne <= 1'b0;
937         EPCWrite <= 1'b0;
938         CauseWrite <= 1'b0;
939     end
940 5'b01001:
941     begin
942         PCSource <= 2'b10;
943         IRWrite <= 1'b0;
944         MemWrite <= 1'b0;
945         RegWrite <= 1'b0;
946         PCWrite <= 1'b1;
947         PCWriteCond <= 1'b0;
948         PCWriteCond_bne <= 1'b0;
949         EPCWrite <= 1'b0;
950         CauseWrite <= 1'b0;
951     end
952 5'b01010:
953     begin
954         ALUSrcA <= 1'b1;
955         ALUSrcB <= 2'b00;
956         ALUOp <= 2'b01;
957         PCSource <= 2'b01;
958         IRWrite <= 1'b0;
959         MemWrite <= 1'b0;
960         RegWrite <= 1'b0;
961         PCWrite <= 1'b0;
962         PCWriteCond <= 1'b0;
963         PCWriteCond_bne <= 1'b1;
964         EPCWrite <= 1'b0;
965         CauseWrite <= 1'b0;
966     end
967 5'b01011:
968     begin
969         ALUSrcA <= 1'b1;
970         ALUSrcB <= 2'b10;
971         ALUOp <= 2'b11;
972         IRWrite <= 1'b0;
973         MemWrite <= 1'b0;
974         RegWrite <= 1'b0;
975         PCWrite <= 1'b0;
976         PCWriteCond <= 1'b0;
977         PCWriteCond_bne <= 1'b0;
978         EPCWrite <= 1'b0;
979         CauseWrite <= 1'b0;
980     end
981 5'b01100:
982     begin
983         RegDst <= 1'b0;

```

```

984         MentoReg <= 1'b0;
985         IRWrite  <= 1'b0;
986         MemWrite <= 1'b0;
987         RegWrite <= 1'b1;
988         PCWrite  <= 1'b0;
989         PCWriteCond <= 1'b0;
990         PCWriteCond_bne <= 1'b0;
991         EPCWrite <= 1'b0;
992         CauseWrite <= 1'b0;
993     end
994     5'b01101:          //中断
995     begin
996         ALUSrcA <= 1'b0;
997         ALUSrcB <= 2'b01;
998         ALUOp <= 2'b00; //add
999         INTPCSource <= 1'b1;
1000        PCSource <= 2'b11;
1001        IRWrite  <= 1'b0;
1002        MemWrite <= 1'b0;
1003        RegWrite <= 1'b0;
1004        PCWrite  <= 1'b1;
1005        PCWriteCond <= 1'b0;
1006        PCWriteCond_bne <= 1'b0;
1007        EPCWrite <= 1'b1;
1008        CauseWrite <= 1'b1;
1009        rst_int <= 1'b1;
1010    end
1011    5'b01110: //eret
1012    begin
1013        INTPCSource <= 1'b0;
1014        PCSource <= 2'b11;
1015        IRWrite  <= 1'b0;
1016        MemWrite <= 1'b0;
1017        RegWrite <= 1'b0;
1018        PCWrite  <= 1'b1;
1019        PCWriteCond <= 1'b0;
1020        PCWriteCond_bne <= 1'b0;
1021        EPCWrite <= 1'b0;
1022        CauseWrite <= 1'b0;
1023        rst_ready <= 1'b1;
1024    end
1025    5'b01111:
1026    begin
1027        IRWrite  <= 1'b0;
1028        MemWrite <= 1'b0;
1029        RegWrite <= 1'b0;
1030        PCWrite  <= 1'b0;
1031        PCWriteCond <= 1'b0;
1032        PCWriteCond_bne <= 1'b0;
1033        EPCWrite <= 1'b0;
1034        CauseWrite <= 1'b0;
1035        rst_int <= 1'b0;
1036        rst_ready <= 1'b0;

```

```

1037         end

1038     default:
1039         begin
1040             IRWrite <= 1'b0;
1041             MemWrite <= 1'b0;
1042             RegWrite <= 1'b0;
1043             PCWrite <= 1'b0;
1044             PCWriteCond <= 1'b0;
1045             PCWriteCond_bne <= 1'b0;
1046             EPCWrite <= 1'b0;
1047             CauseWrite <= 1'b0;
1048             rst_int <= 1'b0;
1049             rst_ready <= 1'b0;
1050         end
1051     endcase
1052 end
1053
1054 end
1055
1056 always @(posedge clk or posedge rst or posedge start)
1057 begin
1058     if(rst || start) en_tx = 0;
1059     else if(nextstate == 5'b01110)
1060     begin
1061         en_tx = 1;
1062     end
1063 end
1064
1065 endmodule
1066
1067 module Reg_File (
1068     input [4:0]ra0,
1069     input [4:0]ra1,
1070     input [4:0]ra2,
1071     input [4:0]wa,
1072     input [31:0]wd,
1073     input we,
1074     input rst,
1075     input clk,
1076     output [31:0]rd0,
1077     output [31:0]rd1,
1078     output [31:0]rd2
1079 );
1080
1081     reg [31:0]R[31:0];
1082     reg [4:0] addr;
1083
1084     assign rd0 = R[ra0];
1085     assign rd1 = R[ra1];
1086     assign rd2 = R[ra2];
1087

```

```

1088     always @(posedge clk or posedge rst)
1089     begin
1090         if(rst) R[0] = 32'b0;
1091         else if(we)
1092         begin
1093             addr = wa;
1094             R[addr] = wd;
1095             if(addr==0) R[0] = 32'b0;
1096         end
1097     end
1098
1099 endmodule
1100
1101 module ALUControlUnit(
1102     input [5:0] funct,
1103     input [1:0] ALUOp,
1104     input [5:0] opcode,
1105     output reg [5:0] ALUControlCode
1106 );
1107
1108     localparam ADD = 6'h20;
1109     localparam SUB = 6'h22;
1110     localparam SLT = 6'h2a;
1111
1112     always @(ALUOp or funct or opcode)
1113     begin
1114         case(ALUOp)
1115             2'b00: ALUControlCode = ADD;
1116             2'b01: ALUControlCode = SUB;
1117             2'b10: ALUControlCode = funct[5:0];
1118             2'b11:
1119                 begin
1120                     if(opcode == 6'ha) ALUControlCode = 6'h2a;
1121                     else ALUControlCode = opcode[5:0] + 6'h18;
1122                 end
1123             default: ALUControlCode = ADD;
1124         endcase
1125     end
1126
1127 endmodule
1128
1129 module ALU(
1130     input [31:0] a,
1131     input [31:0] b,
1132     input [5:0] ALUControlCode,
1133     output reg [31:0] result,
1134     output wire Zero
1135 );
1136
1137     localparam ADD = 6'h20;
1138     localparam SUB = 6'h22;
1139     localparam AND = 6'h24;
1140     localparam OR = 6'h25;

```



```
1141     localparam NOR = 6'h27;
1142     localparam XOR = 6'h26;
1143     localparam SLT = 6'h2a;
1144     localparam SRLV = 6'h6;
1145     localparam SLLV = 6'h4;
1146
1147     wire [31:0] slt, c;
1148     assign Zero = ~(|result);
1149     assign c = a + ~b + 1;
1150     assign slt = c[31] ? 32'b1 : 32'b0;
1151
1152     always @(*)
1153     begin
1154         case(ALUControlCode)
1155             ADD: result = a + b;
1156             SUB: result = a - b;
1157             AND: result = a & b;
1158             OR : result = a | b;
1159             NOR: result = ~(a | b);
1160             XOR: result = a ^ b;
1161             SLT: result = slt;
1162             SRLV : result = b >> a[4:0];
1163             SLLV : result = b << a[4:0];
1164             default: result = 32'b1;
1165         endcase
1166     end
1167 endmodule
```