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

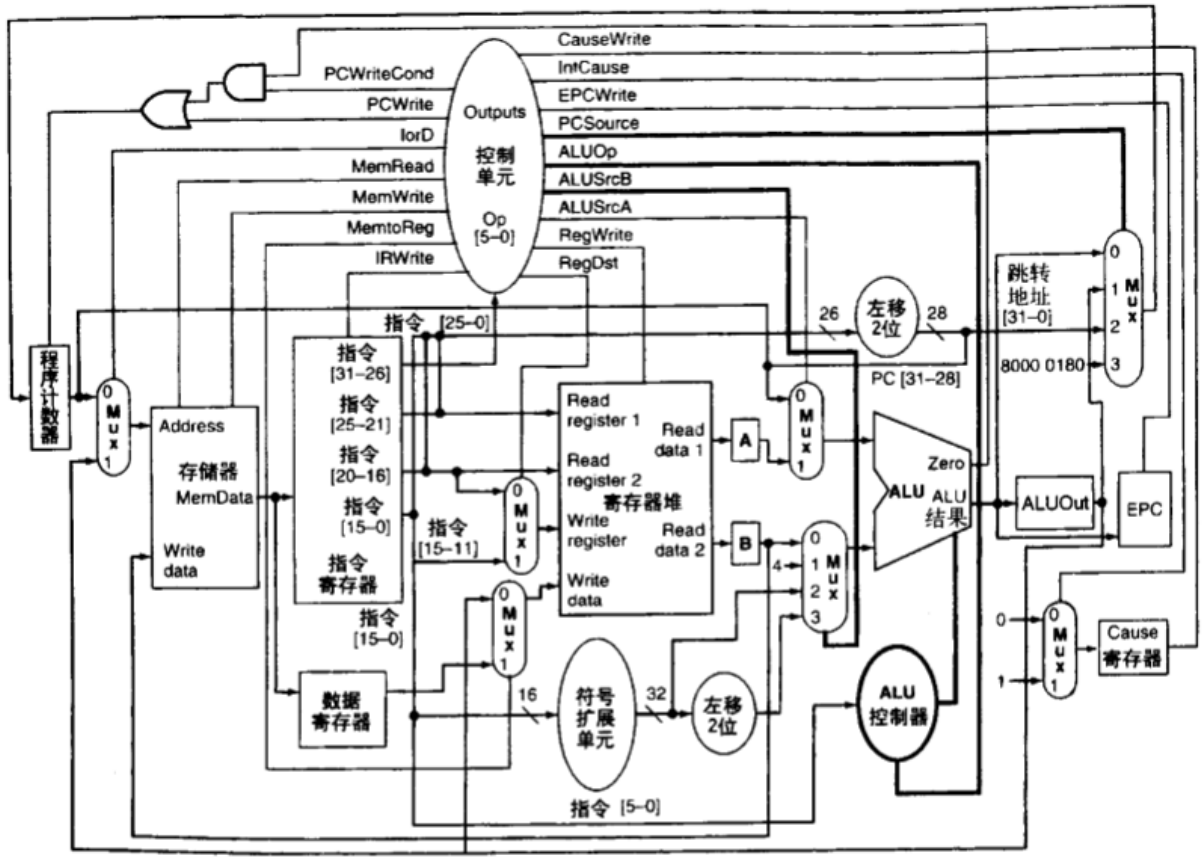


图 5-39　加入了异常处理的多周期数据通路

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。Casue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。

```
 always @(posedge clk or posedge rst)
    begin
        if(rst) PC <= 32'h00000000;
        else
        begin
            if(real_PCWrite) PC <= Mux_PC;
        end
    end

assign Mux_INT_PC = INTPCSource ? INTTable : EPC; //中断开始和中断结束的PC输入

always @(PCSource or ALUResult or ALUOut or JumpAddr or Mux_INT_PC)
   begin
```

```
        case(PCSource)
            2'b00: Mux_PC = ALUResult;
            2'b01: Mux_PC = ALUOut;
            2'b10: Mux_PC = JumpAddr;
            2'b11: Mux_PC = Mux_INT_PC; //中断处理
            default: Mux_PC = ALUResult;
        endcase
    end

//控制单元 FSM

assign real_int = int & enable_int; //中断开关
assign init = cont ? CONT : STEP; //单步或连续执行
assign end_next = real_int ? INT : init; //指令结束判断是否进入中断处理

5'b01101:    //中断
    begin
        ALUSrcA <= 1'b0;
        ALUSrcB <= 2'b01;
        ALUOp <= 2'b00; //add
        INTPCSource <= 1'b1;
        PCSource <= 2'b11;
        IRWrite  <= 1'b0;
        MemWrite <= 1'b0;
        RegWrite <= 1'b0;
        PCWrite  <= 1'b1;
        PCWriteCond <= 1'b0;
        PCWriteCond_bne <= 1'b0;
        EPCWrite <= 1'b1;
        CauseWrite <= 1'b1;
        rst_int <= 1'b1; //中断信号清除
    end
5'b01110: //eret
    begin
        INTPCSource <= 1'b0;
        PCSource <= 2'b11;
        IRWrite  <= 1'b0;
        MemWrite <= 1'b0;
        RegWrite <= 1'b0;
        PCWrite  <= 1'b1;
        PCWriteCond <= 1'b0;
        PCWriteCond_bne <= 1'b0;
        EPCWrite <= 1'b0;
        CauseWrite <= 1'b0;
        rst_ready <= 1'b1;   //数据ready状态清0
    end
```

3.指令增加

```
//ALU控制单元，ALUOp
2'b10: ALUControlCode = funct[5:0]; //ALU控制码，逻辑算数运算的控制码由fucnt字段决定

//ALU
localparam SRLV = 6'h6; //逻辑可变右移
localparam SLLV = 6'h4; //逻辑可变左移
SRLV : result = b >> a[4:0];
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

```
MEM mo_mem(clk_cpu, rst, real_we, wd, a_cpu, addr_ddu, mem_data_cpu, mem_data_ddu);
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);//


assign overflow = (a_cpu >= 32'h1000) ? 1:0; //cpu访问地址不在Mem内，则在IO映射区内
assign real_we = we & !overflow;

assign addr_io_byte = a_cpu - 32'h1000; //IO区实际字节地址
assign addr_io_word = addr_io_byte >> 2;


assign io_data_cpu = R[addr_io_word];
assign data_cpu = overflow ? io_data_cpu : mem_data_cpu; //cpu读取的是IO还是Mem数据

//IO数据Buffer与IO数据内存相连
assign buffer_TX[31:0] = R_TX[0];
assign buffer_TX[63:32] = R_TX[1];
assign R_RX[0] = buffer_RX[31:0];
assign R_RX[1] = buffer_RX[63:32];

assign R[0] = R_RX[0];
assign R[1] = R_RX[1];
assign R[2] = R_TX[0];
assign R[3] = R_TX[1];
assign io_data_ddu = R[addr_ddu[31:2]];

//写入IO数据
always @(posedge clk_cpu)
begin
    if(we && overflow) R_TX[addr_io_word - 2] = wd;
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

```
//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
always@(posedge clk_153600hz)
     begin
        if(RX == 1'b0 && continue == 1'b0 && !receiving)
        begin
            continue = 1;
            c = 1;
        end
        else if(continue)
        begin
            if(RX == 0) c = c + 1;
            else continue = 0;
            if(c == 0)
            begin
                receiving = 1;
                continue = 0;
                cyc = 1;
                b = 0;
            end
        end
        else if(receiving == 1 && cyc==0 )
        begin
          if(b==4'b1000) receiving = 0;
          cyc = cyc + 1;
          n[b] = RX;
          b = b + 1;
          if(b == 4'b1000)
          begin
              flag = 1;
              frame = n;
          end
        end
        else
        begin
            flag = 0;
            cyc = cyc + 1;
        end
     end

//中断信号产生与清除
always@(posedge valid_frame or posedge rst_int or posedge rst_ready or posedge rst)
    begin
        if(rst_int || rst)
        begin
            cnt = 0;
            int = 0;
            ready = 0;
        end
        else if(rst_ready)
```

```
        begin
            ready = 0;
            cnt = 0;
        end
        else if(!ready && valid_frame)
        begin
            //buffer_RX = buffer_RX >> 8;
            //buffer_RX[63:56] = frame;
            buffer_RX = buffer_RX << 8;
            buffer_RX[7:0] = frame;
            cnt = cnt + 1;
            if(cnt == 0)
            begin
                ready = 1;
                int = 1;
            end
        end
    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

```
always@(posedge clk_9600hz)
     begin
        if(en_tx && !traning)
        begin
            low = 6'd0;
            cyc_tx = 0;
            traning = 1;
            cnt = 0;
            gap = 1'b1;
        end
        else if(traning && gap)
        begin
            if(cnt == 2'b1)
            begin
                TX = 0;
                gap = 0;
                low = low - 6'd8;
            end
            else TX = 1;
            cnt = cnt + 1'b1;
        end
        else if(traning)
        begin
            TX = buffer_TX[low + cyc_tx];
            cyc_tx = cyc_tx + 1;
            if(cyc_tx == 3'b0) gap = 1'b1;
```

```
            if(low == 6'b0 && cyc_tx == 3'b0)
            begin
                traning = 0;
            end
        end
        else
        begin
            TX = 1;
        end
     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)

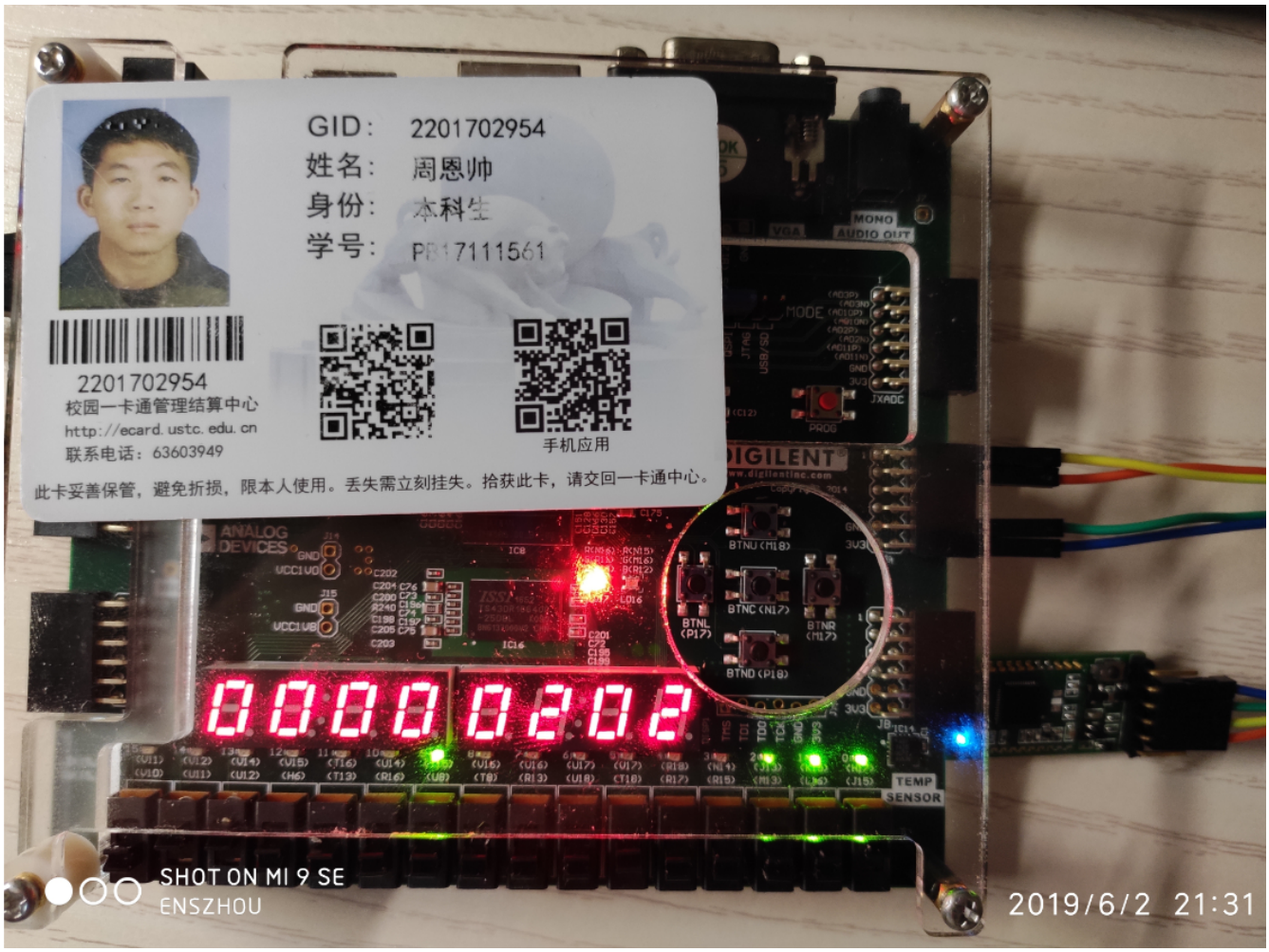
通过有限次的循环就可以求出两个数的最大公因数了。

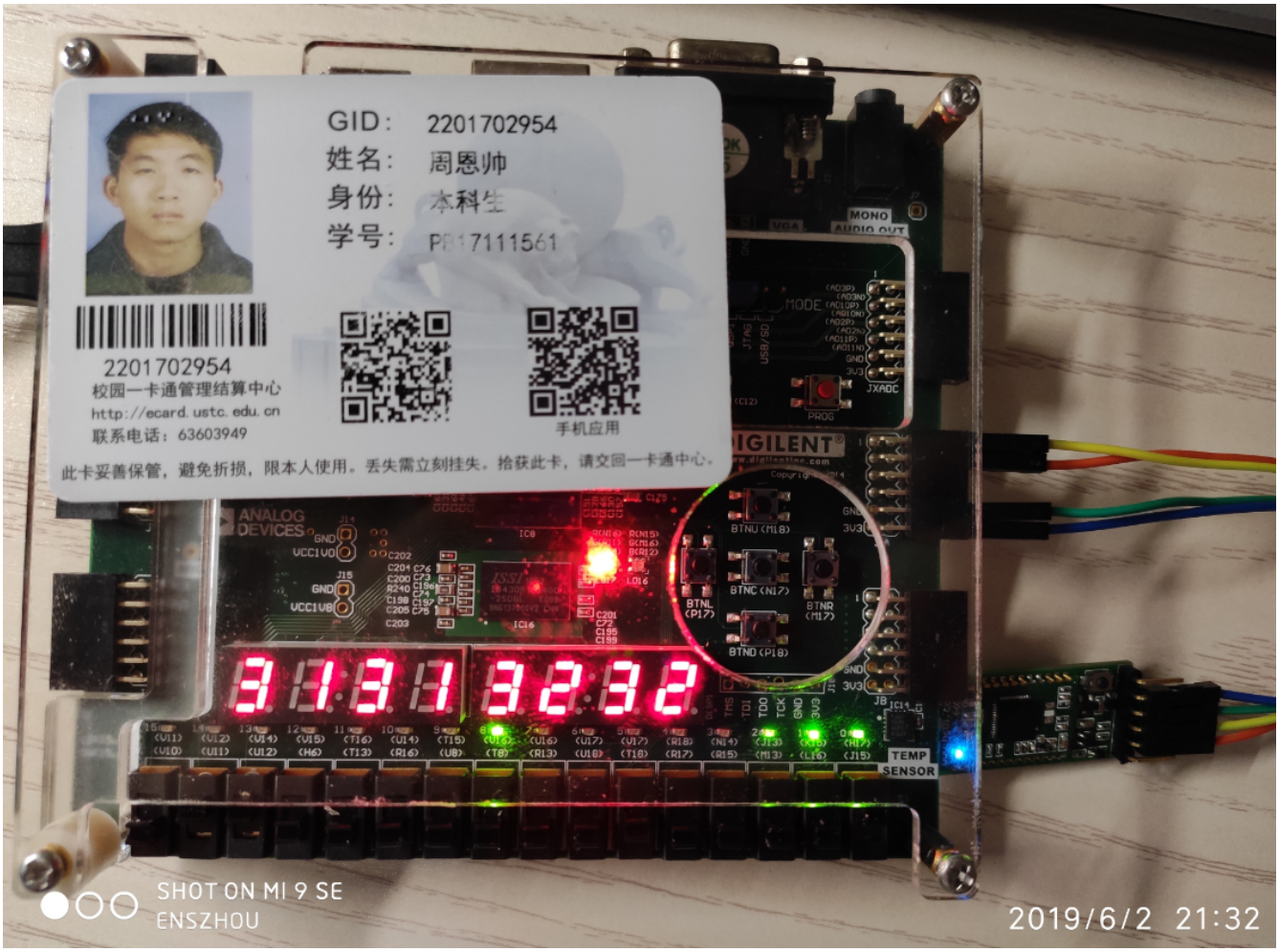
代码会在附录里给出。

仿真/下载

```
module CPU_MEM_tb(
    );

    reg clk, rst, mode_cpu,cont,run,v;
    wire [31:0] mem_data, reg_data, pc;
    reg [31:0]addr;
    reg [7:0] frame;

    CPU_MEM mo_cpu_mem_tb(clk,cont, run, rst, addr, pc, mem_data,reg_data,v,frame);


    initial
    begin
        v <= 0;
        frame <=8'b01010101;
        clk <= 0;
        run <= 1;
```

```
        rst <= 0;
        cont <= 1;
        addr <= 32'h8;
        #10 rst <= 1;
        #4 rst <= 0;
        #500 mode_cpu <= 0;
        #400 addr <= 32'h4;
    end


    always
    forever #1 clk = ~clk;

    always
    forever #25 v <= ~v;

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

```
.text
_start:
    lw $t1 0x1000($0)
    lw $t2 0x1004($0)

    beq $t1 $0 _check_mem

    andi $s0 $s0 0
    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)
```

```
63      eret
```

- 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
```

```
endmodule

module CLK #(parameter N = 5000000)(
    input clk_Xhz,
    input rst,
    input enable,
    output reg Q
    );
    reg [29:0]cnt;
    wire [29:0]n;

    assign n = N >> 1;

    always @ (posedge clk_Xhz or posedge rst)
    begin
        if(rst)
        begin
            Q <= 0;
            cnt <= 30'b0;
        end
        else
        begin
            if(enable)
            begin
                if(cnt >= n)
                begin
                    Q <= ~Q;
                    cnt <= 30'b0;
                end
                else cnt <= cnt + 30'b1;
            end
        end
    end
endmodule

//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
module DDU(
    input cont,
    input step,
    input [2:0] mode_seg,
    input inc,
    input dec,
    input [31:0] pc,
    input [31:0] mem_data,
    input [31:0] reg_data,
    input [7:0]frame,
    input valid_frame,
    input [31:0] io_data,
    input rst,
    input clk_8mhz,
    input clk_cpu,
    output reg run,
    output reg [31:0] addr,
```

```
    output [15:0] led,
    output [15:0] display,
    output [1:0] mode
    );

    wire [6:0]seg;
    wire [7:0]AN;
    wire DP;
    wire clk_4khz,inc_sin,dec_sin,step_sin;
    reg [3:0] x0,x1,x2,x3,x4,x5,x6,x7;


    assign display = {DP,seg,AN};
    assign led[7:0] = pc[10:2];
    assign led[15:8] = addr[10:2];
    assign mode = (pc >= 32'h00000040) ? 2'b01 : 2'b10;

    CLK #(2000) mo_clk_4k(clk_8mhz,0,1,clk_4khz);
    SEG mo_seg(clk_4khz,8'b11111111,8'b00000000,x0,x1,x2,x3,x4,x5,x6,x7,AN,DP,seg);
    MUL_SIN mo_mul_sin_inc(clk_cpu,inc,rst,inc_sin);
    MUL_SIN mo_mul_sin_dec(clk_cpu,dec,rst,dec_sin);
    MUL_SIN mo_mul_sin_step(clk_cpu,step,rst,step_sin);

    always@(mode_seg or mem_data or reg_data or frame)
    begin
        if(mode_seg == 3'b1)
        begin
            x0 = mem_data[3:0];
            x1 = mem_data[7:4];
            x2 = mem_data[11:8];
            x3 = mem_data[15:12];
            x4 = mem_data[19:16];
            x5 = mem_data[23:20];
            x6 = mem_data[27:24];
            x7 = mem_data[31:28];
        end
        else if(mode_seg == 3'b0)
        begin
            x0 = reg_data[3:0];
            x1 = reg_data[7:4];
            x2 = reg_data[11:8];
            x3 = reg_data[15:12];
            x4 = reg_data[19:16];
            x5 = reg_data[23:20];
            x6 = reg_data[27:24];
            x7 = reg_data[31:28];
        end
        else if(mode_seg == 3'b10)
        begin
            x0 = frame[0];
            x1 = frame[1];
            x2 = frame[2];
            x3 = frame[3];
```

```
            x4 = frame[4];
            x5 = frame[5];
            x6 = frame[6];
            x7 = frame[7];
        end
        else
        begin
           x0 = io_data[3:0];
           x1 = io_data[7:4];
           x2 = io_data[11:8];
           x3 = io_data[15:12];
           x4 = io_data[19:16];
           x5 = io_data[23:20];
           x6 = io_data[27:24];
           x7 = io_data[31:28];
        end
    end

    always@(posedge clk_cpu or posedge rst)
    begin
        if(rst)
        begin
            addr <= 32'b0;
        end
        else
        begin
            if(inc_sin) addr <= addr + 4;
            else if(dec_sin) addr <= addr - 4;
        end
    end

    always@(cont or clk_cpu or step_sin)
    begin
        if(cont) run = 1'b1;
        else run = step_sin;
    end

endmodule

module SEG(
    input clk_4khz,
    input [7:0] en, //pos
    input [7:0] dp_x, //pos
    input [3:0] x0,x1,x2,x3,x4,x5,x6,x7,
    output [7:0]AN,
    output DP,
    output reg [6:0]seg
    );


    reg [2:0] state,next_state;
    reg [7:0] an;
    reg dp;
```

```
    wire [6:0]seg0,seg1,seg2,seg3,seg4,seg5,seg6,seg7;

    bcd_to_seg b0(x0,seg0);
    bcd_to_seg b1(x1,seg1);
    bcd_to_seg b2(x2,seg2);
    bcd_to_seg b3(x3,seg3);
    bcd_to_seg b4(x4,seg4);
    bcd_to_seg b5(x5,seg5);
    bcd_to_seg b6(x6,seg6);
    bcd_to_seg b7(x7,seg7);

    assign AN = an | ~en;
    assign DP = ~dp;

    always @ (posedge clk_4khz)
        begin
            state <= next_state;
        end

    always @(state)
    begin
        case(state)
            3'b000: next_state = 3'b001;
            3'b001: next_state = 3'b010;
            3'b010: next_state = 3'b011;
            3'b011: next_state = 3'b100;
            3'b100: next_state = 3'b101;
            3'b101: next_state = 3'b110;
            3'b110: next_state = 3'b111;
            3'b111: next_state = 3'b000;
            default: next_state = 3'b000;
        endcase
    end


    always @ (posedge clk_4khz)
        begin
            case(state)
                3'b000:
                begin
                an = 8'b11111110;
                dp = dp_x[0];
                seg = seg0;
                end
                3'b001:
                begin
                an = 8'b11111101;
                dp = dp_x[1];
                seg = seg1;
                end
                3'b010:
                begin
                an = 8'b11111011;
```

```
                dp = dp_x[2];
                seg = seg2;
                end
                3'b011:
                begin
                an = 8'b11110111;
                dp = dp_x[3];
                seg = seg3;
                end
                3'b100:
                begin
                an = 8'b11101111;
                dp = dp_x[4];
                seg = seg4;
                end
                3'b101:
                begin
                an = 8'b11011111;
                dp = dp_x[5];
                seg = seg5;
                end
                3'b110:
                begin
                an = 8'b10111111;
                dp = dp_x[6];
                seg = seg6;
                end
                3'b111:
                begin
                an = 8'b01111111;
                dp = dp_x[7];
                seg = seg7;
                end
                default:
                begin
                an = 8'b11111110;
                seg = seg0;
                dp = dp_x[0];
                end
            endcase
        end
endmodule

module bcd_to_seg(
    input [3:0] x,
    output reg [6:0]seg);
    always @ (x)
        begin
            case(x)
                4'b0000:seg = 7'b1000000;
                4'b0001:seg = 7'b1111001;
                4'b0010:seg = 7'b0100100;
                4'b0011:seg = 7'b0110000;
```

```
                4'b0100:seg = 7'b0011001;
                4'b0101:seg = 7'b0010010;
                4'b0110:seg = 7'b0000010;
                4'b0111:seg = 7'b1111000;
                4'b1000:seg = 7'b0000000;
                4'b1001:seg = 7'b0010000;
                4'b1010:seg = 7'b0001000;
                4'b1011:seg = 7'b0000011;
                4'b1100:seg = 7'b0100111;
                4'b1101:seg = 7'b0100001;
                4'b1110:seg = 7'b0000110;
                4'b1111:seg = 7'b0001110;
                default:seg = 7'b1000000;
            endcase
        end
endmodule

module MUL_SIN(
    input clk,
    input s,
    input rst,
    output reg q
    );
    reg last;

    always @(negedge clk or posedge rst)
    begin
        if(rst)
        begin
            last <= 0;
        end
        else
        begin
            if(!last && s)
            begin
                q <= 1'b1;
                last <= 1'b1;
            end
            else if(last && s) q <= 1'b0;
            else if(last && !s)
            begin
                q <= 1'b0;
                last <= 1'b0;
            end
            else q <= 1'b0;
        end
    end
endmodule

module HC_42(
    input clk_9600hz,
    input clk_153600hz,
    input RX,
```

```
    input en_tx,
    input [63:0] buffer_TX,
    output reg TX,
    output start,
    output reg flag,
    output reg [7:0] frame
    );
    reg receiving,traning,gap;
    reg [3:0] b;
    reg [7:0] n;
    reg [2:0] cyc_tx;
    reg [5:0] low;
    reg cnt;
    wire [31:0] data;
    reg [2:0] c;
    reg continue;
    reg[3:0] cyc;

     assign start = traning;

     always@(posedge clk_153600hz)
     begin
        if(RX == 1'b0 && continue == 1'b0 && !receiving)
        begin
            continue = 1;
            c = 1;
        end
        else if(continue)
        begin
            if(RX == 0) c = c + 1;
            else continue = 0;
            if(c == 0)
            begin
                receiving = 1;
                continue = 0;
                cyc = 1;
                b = 0;
            end
        end
        else if(receiving == 1 && cyc==0 )
        begin
          if(b==4'b1000) receiving = 0;
          cyc = cyc + 1;
          n[b] = RX;
          b = b + 1;
          if(b == 4'b1000)
          begin
              flag = 1;
              frame = n;
          end
        end
        else
        begin
```

```
            flag = 0;
            cyc = cyc + 1;
        end
     end

     always@(posedge clk_9600hz)
     begin
        if(en_tx && !traning)
        begin
            low = 6'd0;
            cyc_tx = 0;
            traning = 1;
            cnt = 0;
            gap = 1'b1;
        end
        else if(traning && gap)
        begin
            if(cnt == 2'b1)
            begin
                TX = 0;
                gap = 0;
                low = low - 6'd8;
            end
            else TX = 1;
            cnt = cnt + 1'b1;
        end
        else if(traning)
        begin
            TX = buffer_TX[low + cyc_tx];
            cyc_tx = cyc_tx + 1;
            if(cyc_tx == 3'b0) gap = 1'b1;
            if(low == 6'b0 && cyc_tx == 3'b0)
            begin
                traning = 0;
            end
        end
        else
        begin
            TX = 1;
        end
     end
endmodule

module CPU_MEM(
    input clk_2mhz,
    input cont,
    input run,
    input rst,
    input [31:0] addr_ddu,
    output [31:0] pc,mem_data_ddu,reg_data,io_data_ddu,
    input valid_frame,
    input [7:0] frame,
    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 
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(
```

```
    input clk,
    input cont,
    input run,
    input int,
    input rst,
    input [31:0] MemData,
    input [4:0] RegNum,
    output [31:0] MemAddr,
    output MemWrite_out,
    output [31:0] MemWriteData,
    output [31:0] RegData,
    output [31:0] PC_out,
    output rst_int, rst_ready,en_tx,
    input start
    );

    wire [5:0] opcode;
    wire [1:0] ALUSrcB,ALUOp,PCSource;
    wire
INTPCSource,ALUSrcA,IorD,IRWrite,PCWrite,PCWriteCond,MemWrite,RegDst,RegWrite,MemtoR
eg,PCWriteCond_bne,EPCWrite,CauseWrite;
    wire enable_int;
    wire real_PCWrite;


    //some registers
    reg [31:0] PC, EPC;
    reg [31:0] ALUOut;
    reg [31:0] A,B;
    reg [31:0] MDR;
    reg [31:0] IR;

    //ALU port
    wire Zero;
    wire [31:0] ALUResult;
    wire [5:0] ALUControlCode;

    //Register port
    //wire [4:0] RegReadNum1,RegReadNum2;
    wire [31:0] RegReadData1, RegReadData2;

    //some muxes
    wire [31:0] Mux_MemAddr, Mux_RegWriteData, Mux_ALU_A,Mux_INT_PC;
    reg [31:0] Mux_ALU_B, Mux_PC;
    wire [4:0]  Mux_RegWriteNum;

    //Instruction extend
    wire [31:0] Ins_16Signed32, Ins_16Signed32_noSL, Ins_16Signed32_SL;
    wire [15:0] sign;
    wire [27:0] Ins_26to28;
    wire [31:0] JumpAddr;

    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,MemWrit
    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
```

```
    //MDR
    always @(posedge clk)
    begin
        MDR <= MemData;
        A <= RegReadData1;
        B <= RegReadData2;
        ALUOut <= ALUResult;
    end

    //IR
    always @(posedge clk)
    begin
        if(IRWrite) IR <= MemData;
    end

    //Mux_ALU_B 4-way
    always @(ALUSrcB or B or Ins_16Signed32_noSL or Ins_16Signed32_SL)
    begin
        case(ALUSrcB)
            2'b00: Mux_ALU_B = B;
            2'b01: Mux_ALU_B = 32'd4;
            2'b10: Mux_ALU_B = Ins_16Signed32_noSL;
            2'b11: Mux_ALU_B = Ins_16Signed32_SL;
            default: Mux_ALU_B = B;
        endcase
    end

    //Mux_PC 3-way
    always @(PCSource or ALUResult or ALUOut or JumpAddr or Mux_INT_PC)
    begin
        case(PCSource)
            2'b00: Mux_PC = ALUResult;
            2'b01: Mux_PC = ALUOut;
            2'b10: Mux_PC = JumpAddr;
            2'b11: Mux_PC = Mux_INT_PC; //中断处理
            default: Mux_PC = ALUResult;
        endcase
    end

endmodule

module ControlUnit(
    input clk,
    input cont,
    input run,
    input int,
    input enable_int,
    input rst,
    input [5:0] opcode,
    output reg rst_int,rst_ready,en_tx,
    input start,
    output reg [1:0] ALUSrcB,ALUOp,PCSource,
```

```
    output reg
INTPCSource,ALUSrcA,IorD,IRWrite,PCWrite,PCWriteCond,MemWrite,RegDst,RegWrite,MemtoR
eg,PCWriteCond_bne,EPCWrite,CauseWrite
    );

    localparam CONT = 5'b00000;
    localparam STEP = 5'b01111;
    localparam INT = 5'b01101;

    wire [3:0] init,end_next;
    wire real_int;

    assign real_int = int & enable_int;
    assign init = cont ? CONT : STEP;
    assign end_next = real_int ? INT : init;

    reg [3:0] state, nextstate;
    reg flag;
    //reg [1:0] ALUSrcB,ALUOp,PCSource;
    //reg
ALUSrcA,IorD,IRWrite,PCWrite,PCWriteCond,MemWrite,RegDst,RegWrite,MemtoReg;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
        begin
            state <= init;
        end
        else
        begin
            state <= nextstate;
        end
    end

    always@(state or opcode or run or end_next or init)
    begin
        case(state)
            5'b00000:nextstate = 5'b00001;
            5'b00001:
                begin
                    if(opcode == 6'h23 || opcode == 6'h2b) nextstate = 5'b00010;
                    else if(opcode == 6'h4)nextstate = 5'b01000;
                    else if(opcode == 6'h2)nextstate = 5'b01001;
                    else if(opcode == 6'h5)nextstate = 5'b01010;
                    else if(opcode == 6'h0)nextstate = 5'b00110; //RR
                    else if(opcode == 6'h10)nextstate = 5'b01110;
                    else nextstate = 5'b01011; //RI
                end
            5'b00010:
                begin
                    if(opcode == 6'h23) nextstate = 5'b00011;
                    else nextstate = 5'b00101;
                end
```

```
            5'b00011: nextstate = 5'b00100;
            5'b00100: nextstate = end_next;
            5'b00101: nextstate = end_next;
            5'b00110: nextstate = 5'b00111;
            5'b00111: nextstate = end_next;
            5'b01000: nextstate = end_next;
            5'b01001: nextstate = end_next;
            5'b01010: nextstate = end_next;
            5'b01011: nextstate = 5'b01100; //compute
            5'b01100: nextstate = end_next; //reg write
            5'b01101: nextstate = init; //中断
            5'b01110: nextstate = init; //eret
            5'b01111: if(run) nextstate = 5'b00000;
                      else nextstate = 5'b01111;
            default: nextstate = init;
        endcase

    end

    always @(posedge clk or posedge rst)
    begin
    //if(clk)begin
        if(rst)
        begin
            ALUSrcA  <= 1'b0;
            ALUSrcB  <= 2'b01;
            ALUOp    <= 2'b00;
            IorD     <= 1'b0;
            PCSource <= 2'b00;
            IRWrite  <= cont;
            MemWrite <= 1'b0;
            RegWrite <= 1'b0;
            PCWrite  <= cont;
            PCWriteCond <= 1'b0;
            PCWriteCond_bne <= 1'b0;
            EPCWrite <= 1'b0;
            CauseWrite <= 1'b0;
            rst_int <= 1'b0;
            rst_ready <= 1'b0;
        end
        else
        begin
        case(nextstate)
            5'b00000:
                begin
                    ALUSrcA  <= 1'b0;
                    ALUSrcB  <= 2'b01;
                    ALUOp    <= 2'b00;
                    IorD     <= 1'b0;
                    PCSource <= 2'b00;
                    IRWrite  <= 1'b1;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
```

```
                    PCWrite  <= 1'b1;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                    rst_int <= 1'b0;
                    rst_ready <= 1'b0;
                end
            5'b00001:
                begin
                    ALUSrcA  <= 1'b0;
                    ALUSrcB  <= 2'b11;
                    ALUOp    <= 2'b00;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b00010:
              begin
                  ALUSrcA  <= 1'b1;
                  ALUSrcB  <= 2'b10;
                  ALUOp    <= 2'b00;
                  IRWrite  <= 1'b0;
                  MemWrite <= 1'b0;
                  RegWrite <= 1'b0;
                  PCWrite  <= 1'b0;
                  PCWriteCond <= 1'b0;
                  PCWriteCond_bne <= 1'b0;
                  EPCWrite <= 1'b0;
                  CauseWrite <= 1'b0;
              end
            5'b00011:
              begin
                  IorD <= 1'b1;
                  IRWrite  <= 1'b0;
                  MemWrite <= 1'b0;
                  RegWrite <= 1'b0;
                  PCWrite  <= 1'b0;
                  PCWriteCond <= 1'b0;
                  PCWriteCond_bne <= 1'b0;
                  EPCWrite <= 1'b0;
                  CauseWrite <= 1'b0;
              end
            5'b00100:
              begin
                  RegDst <= 1'b0;
                  MemtoReg <= 1'b1;
                  IRWrite  <= 1'b0;
```

```
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b1;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b00101:
                begin
                    IorD <= 1'b1;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b1;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b00110:
                begin
                    ALUSrcA  <= 1'b1;
                    ALUSrcB  <= 2'b00;
                    ALUOp    <= 2'b10;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b00111:
                begin
                    RegDst <= 1'b1;
                    MemtoReg <= 1'b0;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b1;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b01000:
                begin
                    ALUSrcA  <= 1'b1;
                    ALUSrcB  <= 2'b00;
                    ALUOp    <= 2'b01;
                    PCSource <= 2'b01;
```

```
                IRWrite  <= 1'b0;
                MemWrite <= 1'b0;
                RegWrite <= 1'b0;
                PCWrite  <= 1'b0;
                PCWriteCond <= 1'b1;
                PCWriteCond_bne <= 1'b0;
                EPCWrite <= 1'b0;
                CauseWrite <= 1'b0;
            end
          5'b01001:
            begin
                PCSource <= 2'b10;
                IRWrite  <= 1'b0;
                MemWrite <= 1'b0;
                RegWrite <= 1'b0;
                PCWrite  <= 1'b1;
                PCWriteCond <= 1'b0;
                PCWriteCond_bne <= 1'b0;
                EPCWrite <= 1'b0;
                CauseWrite <= 1'b0;
            end
          5'b01010:
            begin
                ALUSrcA  <= 1'b1;
                ALUSrcB  <= 2'b00;
                ALUOp    <= 2'b01;
                PCSource <= 2'b01;
                IRWrite  <= 1'b0;
                MemWrite <= 1'b0;
                RegWrite <= 1'b0;
                PCWrite  <= 1'b0;
                PCWriteCond <= 1'b0;
                PCWriteCond_bne <= 1'b1;
                EPCWrite <= 1'b0;
                CauseWrite <= 1'b0;
            end
          5'b01011:
            begin
                ALUSrcA  <= 1'b1;
                ALUSrcB  <= 2'b10;
                ALUOp    <= 2'b11;
                IRWrite  <= 1'b0;
                MemWrite <= 1'b0;
                RegWrite <= 1'b0;
                PCWrite  <= 1'b0;
                PCWriteCond <= 1'b0;
                PCWriteCond_bne <= 1'b0;
                EPCWrite <= 1'b0;
                CauseWrite <= 1'b0;
            end
          5'b01100:
            begin
                RegDst <= 1'b0;
```

```
                    MemtoReg <= 1'b0;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b1;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                end
            5'b01101:           //中断
                begin
                    ALUSrcA <= 1'b0;
                    ALUSrcB <= 2'b01;
                    ALUOp <= 2'b00; //add
                    INTPCSource <= 1'b1;
                    PCSource <= 2'b11;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b1;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b1;
                    CauseWrite <= 1'b1;
                    rst_int <= 1'b1;
                end
            5'b01110: //eret
                begin
                    INTPCSource <= 1'b0;
                    PCSource <= 2'b11;
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b1;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                    rst_ready <= 1'b1;
                end
            5'b01111:
                begin
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                    rst_int <= 1'b0;
                    rst_ready <= 1'b0;
```

```
                end


            default:
                begin
                    IRWrite  <= 1'b0;
                    MemWrite <= 1'b0;
                    RegWrite <= 1'b0;
                    PCWrite  <= 1'b0;
                    PCWriteCond <= 1'b0;
                    PCWriteCond_bne <= 1'b0;
                    EPCWrite <= 1'b0;
                    CauseWrite <= 1'b0;
                    rst_int <= 1'b0;
                    rst_ready <= 1'b0;
                end
        endcase
        end

    end

    always @(posedge clk or posedge rst or posedge start)
    begin
        if(rst || start) en_tx = 0;
        else if(nextstate == 5'b01110)
        begin
            en_tx = 1;
        end
    end

endmodule

module Reg_File (
    input [4:0]ra0,
    input [4:0]ra1,
    input [4:0]ra2,
    input [4:0]wa,
    input [31:0]wd,
    input we,
    input rst,
    input clk,
    output [31:0]rd0,
    output [31:0]rd1,
    output [31:0]rd2
    );

    reg [31:0]R[31:0];
    reg [4:0] addr;

    assign rd0 = R[ra0];
    assign rd1 = R[ra1];
    assign rd2 = R[ra2];

```

```
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
```