中国科学技术大学计算机学院

《计算机组成原理实验报告》

学生姓名：林宸昊

学生学号：PB20000034

完成日期：2022.4.15

【实验题目】单周期CPU设计

【实验目的】

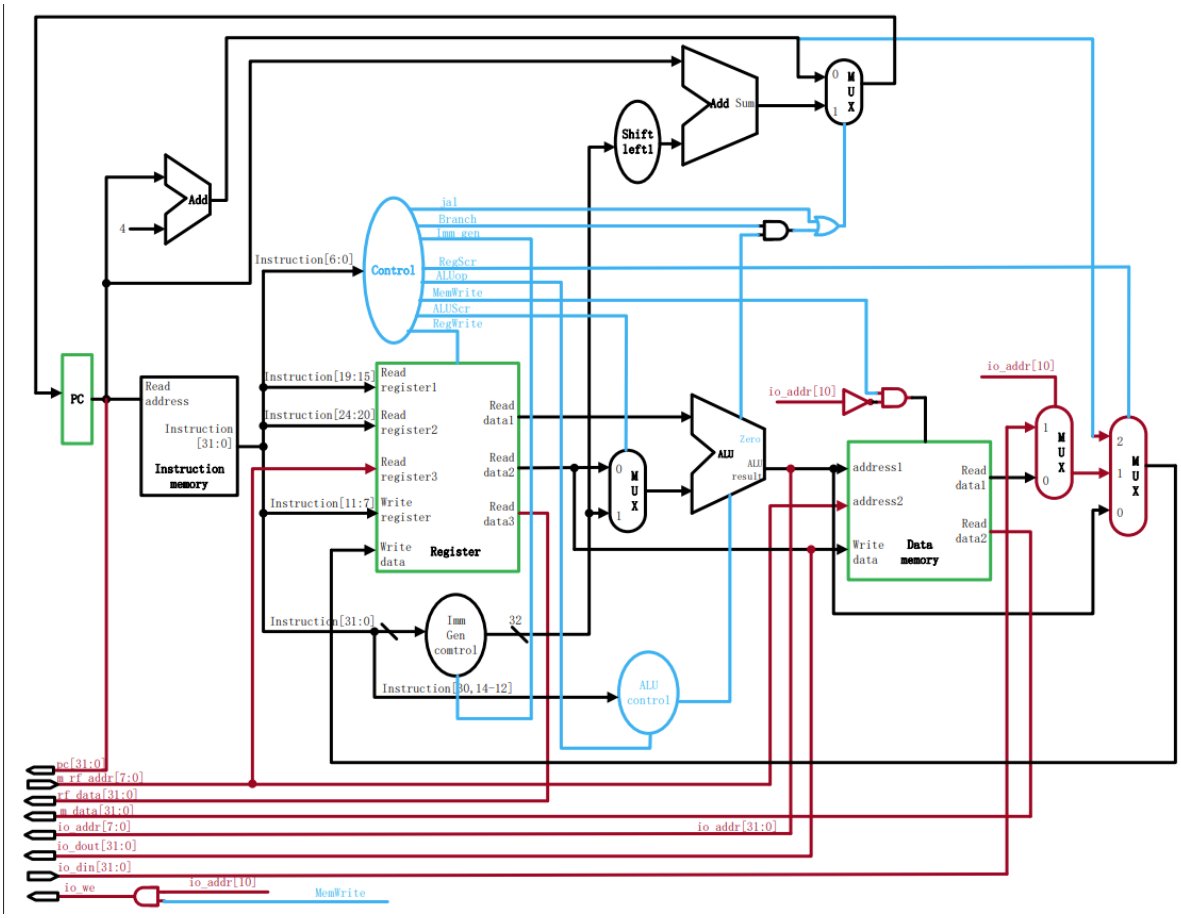
- 理解CPU结构及其工作原理；
- 掌握单周期CPU的设计和调试方式；
- 熟练掌握数据通路和控制器设计及描述方法。

【实验环境】Vivado FPGAOL

【实验内容】

【一、CPU设计】

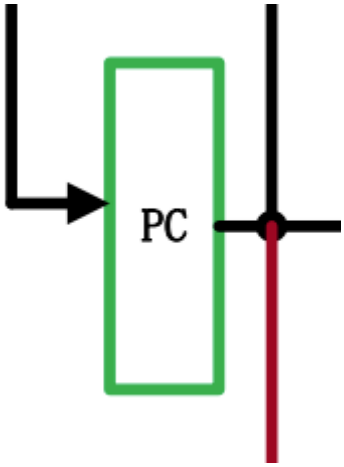
【数据通路】



- 在有完整数据通路情况下，可以依据各部件直接进行模块设计。

【PC】

- 数据通路中的模块



- 设计文件

```
module PC(
    input [31:0] in,    //左侧输入
    input rst,
    input clk,
    output reg [31:0] out   //上方黑色输出
    );
    always @ (posedge clk or posedge rst)
    begin
        if(rst)
            out <= 32'h0000_3000;   //初始化
        else
```

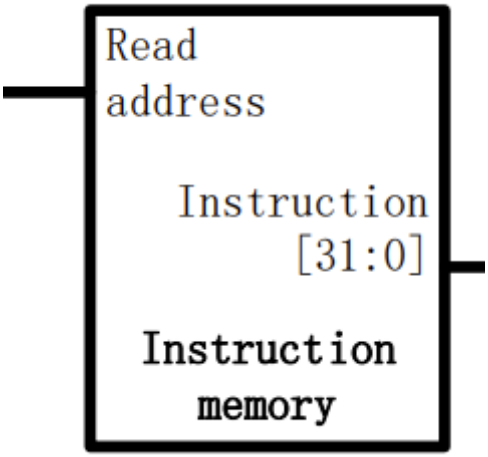
```
            out <= in;
    end
endmodule
```

- 模块例化

```
PC PC(
    .in(pc_in),
    .rst(rst),
    .clk(clk),
    .out(pc) //红线及黑线输出
);
```

【指令存储器】

- 数据通路中的模块

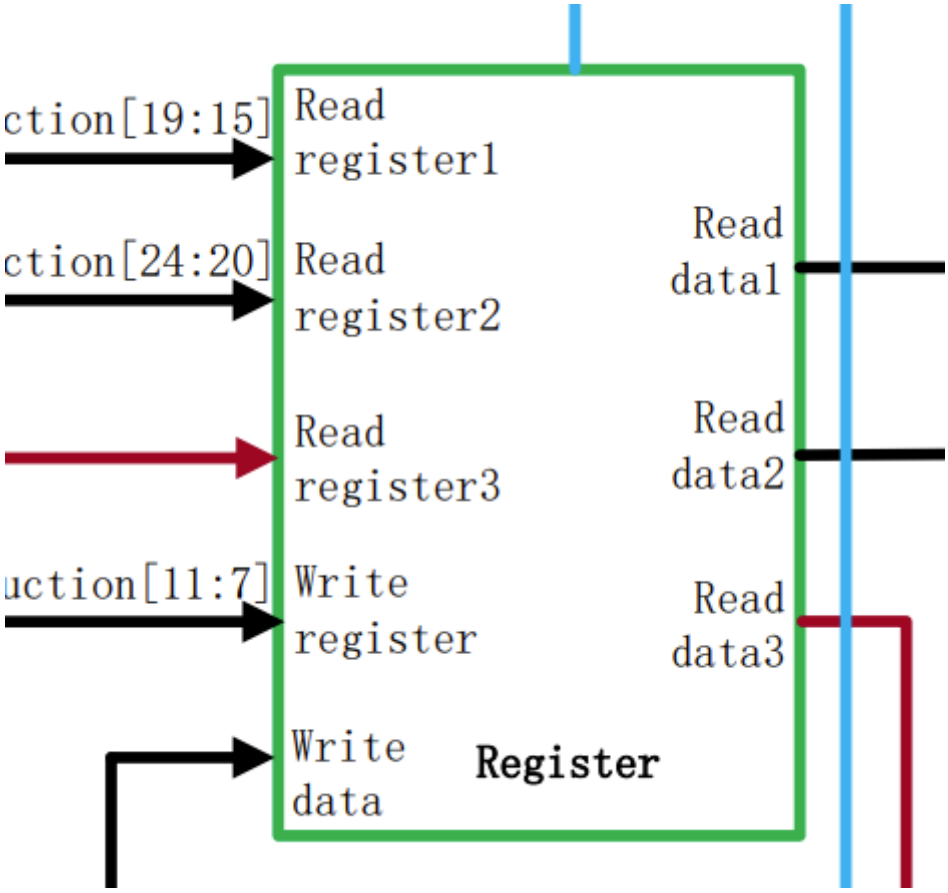


- 直接使用只读式分布式存储器IP核，故可直接例化模块

```
dist_mem_gen_0 ins_mem(
    .a(pc[9:2]),     //pc从x0003000开始且每次+4，即0100，故使用[9:2]
    .spo(INS)        //指令输出
);
```

【寄存器堆】

- 数据通路中的模块



- 设计文件

```
module register_file # (parameter WIDTH = 32) (
    input clk,
    input [4:0] ra0,            //读端口1
    output [WIDTH - 1:0] rd0,
    input [4:0] ra1,            //读端口2
    output [WIDTH - 1:0] rd1,
    input [4:0] wa,
    input we,
    input [WIDTH - 1:0] wd,
    input [7:0] ra2,            //调试用读端口
    output [WIDTH - 1:0] rd_debug
);

    reg [WIDTH - 1:0] regfile [31:0];
    initial
    begin
    regfile[0] = 0;
    //……此处略去初始化
    end

    assign rd0 = regfile[ra0];
    assign rd1 = regfile[ra1];
    assign rd_debug = regfile[ra2];


    always  @ (posedge  clk)
        if (we)
            if(wa == 0)
                regfile[wa] <= 0;       //使r0内容恒定为0
            else
```

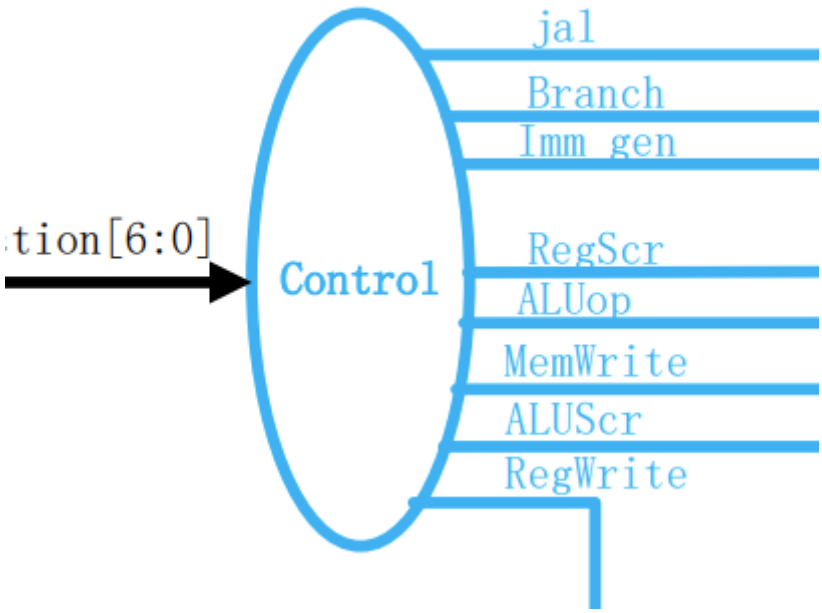
```
                regfile[wa]  <=  wd;

endmodule
```

- 模块例化

```
register_file Registers(
    .clk(clk),
    .ra0(INS[19:15]),           //rs1
    .rd0(ReadData1_reg),        //
    .ra1(INS[24:20]),           //rs2
    .rd1(ReadData2_reg),        //
    .wa(INS[11:7]),             //dest
    .we(RegWrite),              //
    .wd(MUX_3_1_out_data),      //what to write
    .ra2(m_rf_addr[7:0]),       //used for debug
    .rd_debug(rf_data[31:0])    //used for debug
);
```

【控制单元】

- 数据通路中的模块



事实上由于blt以及sub的加入，只传入[6:0]是不够的，我们传入完整指令;

Immgen可以直接略去。

- 设计文件

```
module control_unit(
    input [31:0] ins,
    output  reg blt,
            reg sp_jump, //used for jalr
            reg auipc,
            reg Jump,    //used for jal & jalr
            reg Branch,
            reg [1:0] RegSrc,
            reg MemWrite,
            reg ALUSrc,
```

```
            reg RegWrite,
            reg [1:0] ALUop
);
    always @ (*)
    begin
        blt = 0;    //special judgement
    case (ins[6:0])
        7'b0110011://add
        if(ins[30])
            begin
                Jump = 0;       //not involved
                Branch = 0;     //not involved
                RegSrc = 2'b00; //choose ALU_result
                ALUop = 2'b01;  //add
                MemWrite = 0;   //not involved in writing memory
                ALUSrc = 0;     //choose reg
                RegWrite = 1;   //involved in writing reg
                auipc = 0;      //not involved
                sp_jump = 0;    //notinvolved
            end
        else
            begin
                Jump = 0;
                Branch = 0;
                RegSrc = 2'b00;
                ALUop = 2'b00; //sub
                MemWrite = 0;
                MemRead = 0;
                ALUSrc = 0;
                RegWrite = 1;
                auipc = 0;
                sp_jump = 0;
            end
        7'b0010011://addi
        begin
            Jump = 0;
            Branch = 0;
            RegSrc = 2'b00;
            ALUop = 2'b00;
            MemWrite = 0;
            MemRead = 0;
            ALUSrc = 1;     //choose imm
            RegWrite = 1;
            auipc = 0;
            sp_jump = 0;
        end
        7'b0010111://auipc
        begin
            Jump = 0;
            Branch = 0;
            RegSrc = 2'b10;     //choose pc_result
            ALUop = 2'b00;
            MemWrite = 0;
            MemRead = 0;
            ALUSrc = 1;
            RegWrite = 1;
            auipc = 1;      //involved
            sp_jump = 0;
```

```
        end
        7'b1101111://jal
        begin
            Jump = 1;       //involved
            Branch = 0;
            RegSrc = 2'b10;
            ALUop = 2'b00;
            MemWrite = 0;
            MemRead = 0;
            ALUSrc = 0;
            RegWrite = 1;
            auipc = 0;
            sp_jump = 0;
        end
        7'b1100111://jalr
        begin
            Jump = 1;
            Branch = 0;
            RegSrc = 2'b10;
            ALUop = 2'b00;
            MemWrite = 0;
            MemRead = 0;
            ALUSrc = 0;
            RegWrite = 1;
            sp_jump = 1;    //special jump
            auipc = 0;
        end
        7'b1100011://beq or blt
        if(ins[14])
        begin
            Jump = 0;
            Branch = 1;
            RegSrc = 2'b00;
            ALUop = 2'b01;
            MemWrite = 0;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
            auipc = 0;
            sp_jump = 0;
            blt = 1;
        end
        else
        begin
            Jump = 0;
            Branch = 1;
            RegSrc = 2'b00;
            ALUop = 2'b01;
            MemWrite = 0;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
            auipc = 0;
            sp_jump = 0;
            blt = 0;
        end
        7'b0000011://lw
        begin
```

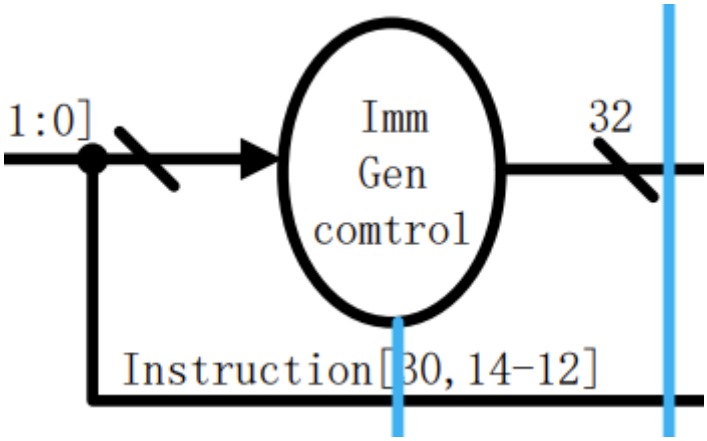
```
            Jump = 0;
            Branch = 0;
            RegSrc = 2'b01;
            ALUop = 2'b00;
            MemWrite = 0;
            ALUSrc = 1;
            RegWrite = 1;
            auipc = 0;
            sp_jump = 0;
        end
        7'b0100011://sw
        begin
            Jump = 0;
            Branch = 0;
            RegSrc = 2'b00;
            ALUop = 2'b00;
            MemWrite = 1;
            ALUSrc = 1;
            RegWrite = 0;
            auipc = 0;
            sp_jump = 0;
        end
        default: ;
    endcase
    end
endmodule
```

- 模块例化

```
control_unit control(
    .ins(INS),
    .Jump(jal),
    .Branch(Branch),
    .RegSrc(RegSrc),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .blt(blt),
    .RegWrite(RegWrite),
    .ALUop(ALUop),
    .auipc(auipc),
    .sp_jump(sp_jump)
);
```

【立即数产生模块】

- 数据通路中的模块



- 设计文件

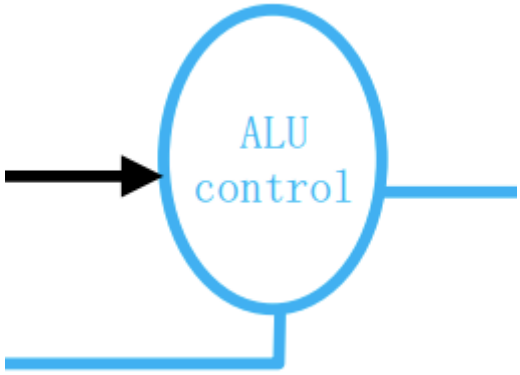
```
module ImmGen(
    input [31:0] ins,
    output reg [31:0] imm
    );
    always @ (*)
    begin
        if(ins[6:0] == 7'b0010011 )    //addi
            imm = {{20{ins[31]}},ins[31:20]};
        if(ins[6:0] == 7'b0000011)    //lw
            imm = {{20{ins[31]}},ins[31:20]};
        if(ins[6:0] == 7'b0100011)    //sw
            imm = {{20{ins[31]}},ins[31:25],ins[11:7]};
        if(ins[6:0] == 7'b1100011)    //beq
            imm = {{20{ins[31]}},ins[31],ins[7],ins[30:25],ins[11:8]};
        if(ins[6:0] == 7'b1101111)    //jal
            imm = {{12{ins[31]}},ins[31],ins[19:12],ins[30],ins[30:21]};
        if(ins[6:0] == 7'b0010111)    //auipc
            imm = (ins[31:12] << 12);
        if(ins[6:0] == 7'b1100111)      //jalr
            imm = {{20{ins[31]}}, ins[31:20]};
    end
endmodule
```

- 模块例化

```
ImmGen ImmGen(
    .ins(INS),
    .imm(Immgen)
);
```

【ALU控制器】

- 数据通路中的模块



- 设计文件

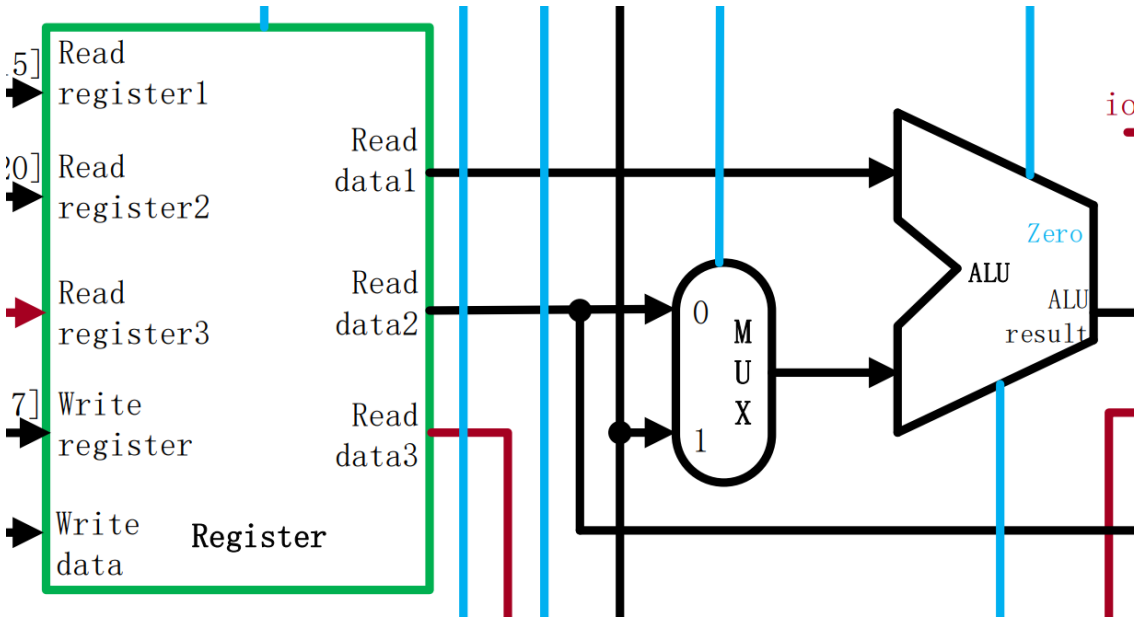
```
module ALUcontrol(
    input [1:0] ALUop,
    output reg [2:0] ALUout
);
    parameter ADD = 3'b000;
    parameter SUB = 3'b001;
    always @ (*)
    begin
        case (ALUop)
            2'b00:
                ALUout = ADD;
            2'b01:
                ALUout = SUB;
            default: ;
        endcase
    end
endmodule
```

- 模块例化

```
ALUcontrol ALUcontrol(
    .ALUop(ALUop),
    .ALUout(ALUfunc)
);
```

【立即数or寄存器2选1选择器】

- 数据通路中的模块



- 选择器设计文件（后同，说明时则仅做例化说明）

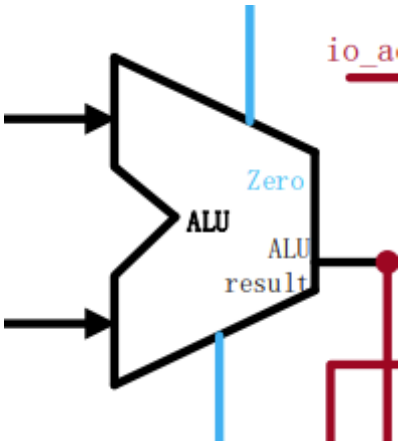
```
module mux2_1 # (parameter WIDTH = 32) (
    input [WIDTH - 1 : 0] a, b,
    input  sel,
    output [WIDTH - 1 : 0] o
);
    assign o  = sel ? b : a;
endmodule
```

- 模块例化

```
mux2_1 MUX_reg(
    .a(ReadData2_reg),
    .b(Immgen),
    .sel(ALUSrc),
    .o(MUX_out_reg)
);
```

【alu】

- 数据通路中的模块



- 设计文件

```
module alu (
    input  [31:0] a, b,     //num to operate
```

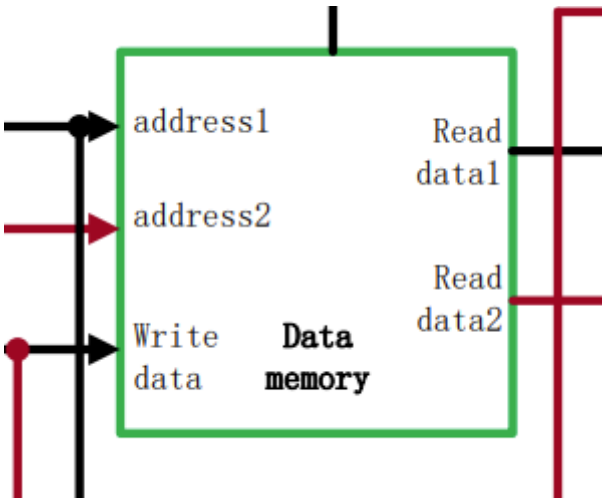
```
    input  [2:0] f,          //function
    input blt,               //blt signal
    output  reg [31:0] y,    //result
    output  reg z,           //beq
    output  reg less         //blt
);
    always @ (*)
    begin
        z = 0;
        less = 0;
        if(f == 3'b000)
        begin
             y = a + b;
             if(y == 0)
                 z = 1;
        end
        else if(f == 3'b001)
        begin
             y = a - b;
             if(y == 0)
                z = 1;
             if (y[31] && blt)
                less = 1;
        end
        else if(f == 3'b010)
        begin
            y = a & b;
             if(y == 0)
                 z = 1;
        end
        else if(f == 3'b011)
        begin
            y = a | b;
             if(y == 0)
                 z = 1;
        end
        else if(f == 3'b100)
        begin
             y = a ^ b;
             if(y == 0)
                 z = 1;
        end
        else
        begin
            y = 0;
            z = 1;
            less = 0;
        end
    end
endmodule
```

- 模块例化

```
alu alu(
    .a(ReadData1_reg),
    .b(MUX_out_reg),
    .f(ALUfunc),
    .y(ALU_result),
    .blt(blt),
    .z(ALU_zero),
    .less(ALU_less)
);
```

【存储器模块】

- 数据通路中的模块

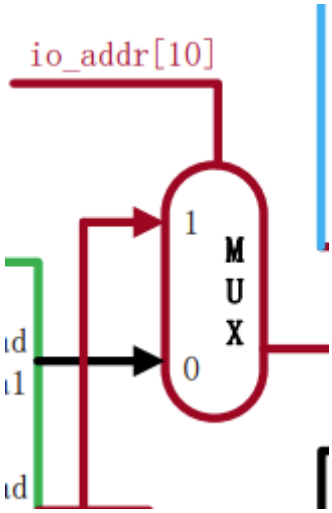


- 采用例化IP核的方式，使用256x32双端口的分布式存储器；
- 例化模块

```
dist_mem_gen_1 data_mem(
    .a(ALU_result[9:2]),     //address1
    .d(ReadData2_reg),       //Writedata
    .dpra(m_rf_addr[7:0]),   //address2
    .dpo(m_data[31:0]),      //ReadData2
    .clk(clk),
    .we(MemWrite && (~ALU_result[10])),
    .spo(ReadData1_data)     //ReadData1
);
```

【存储器orIO选择器】

- 数据通路中的模块

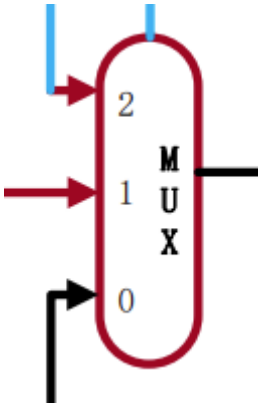


- 模块例化

```
mux2_1 MUX_2_1_data_mem(
    .a(ReadData1_data),
    .b(io_din[31:0]),
    .sel(ALU_result[10]),   //io_addr[10]
    .o(MUX_2_1_out_data)
);
```

【寄存器写入三选一选择器】

- 数据通路中的模块



- 设计文件

```
module mux3_1 #(parameter WIDTH = 32) (
    input [WIDTH - 1 : 0] a, b, c,
    input [1:0]sel,
    output reg [WIDTH - 1 : 0] o
);

    always @(*) begin
        case (sel)
            2'b00: o = a;
            2'b01: o = b;
            2'b10: o = c;
        endcase
    end
endmodule
```

- 模块例化

```
mux3_1 MUX_3_1_DataMemory(
    .c(t),
    .b(MUX_2_1_out_data),
    .a(ALU_result),
    .sel(RegSrc),
    .o(MUX_3_1_out_data)
);
```

- 其中t为auipc指令所产生的特殊pc值

```
wire [31:0] con = 4;
wire [31:0] MUX_2_1_out_pc;

mux2_1 MUX_2_1_pc(
    .a(con),
    .b(Immgen),
    .sel(auipc),
    .o(MUX_2_1_out_pc)
);

wire [31:0] t;
assign t = pc + MUX_2_1_out_pc;
```

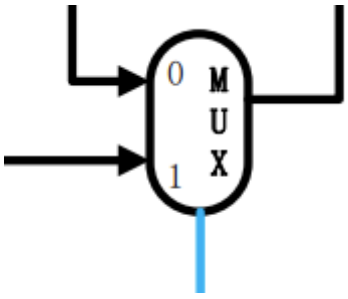
【jalr的特殊处理——利用2选1选择器】

- 模块例化

```
wire [31:0] shiftleft_pc;
assign shiftleft_pc = (Immgen << 1) + pc;
wire [31:0] jalr_pc;
wire [31:0] a = 1;
assign jalr_pc = (ReadData1_reg + Immgen) & ~a;
wire [31:0] changed_pc;

mux2_1 MUX_2_1_jalr(
    .a(shiftleft_pc),
    .b(jalr_pc),
    .sel(sp_jump),      //jalr happend
    .o(changed_pc)
);
```

【pc输入】

- 数据通路中的模块



- 模块例化

```
wire signal;
assign signal = jal || (Branch && ALU_zero) || (Branch && ALU_less);

mux2_1 MUX_2_1_changed(
    .a(pc + 4),
    .b(changed_pc),
    .sel(signal),
    .o(pc_in)
);
```

【IO及DEBUG模块】

- 设计文件

```
  assign io_dout = ReadData2_reg;
  assign io_addr[7:0] = ALU_result[7:0];
  assign io_we = MemWrite && ALU_result[10];
```

【cpu整体例化】

- 设计文件

```
module cpu_one_cycle(
    input run,      //sw6
    input step,     //button
    input valid,    //sw5
    input [4:0] in, //sw4-0
    input rst,      //sw7
    input clk,
    output [1:0] check,   //led6-5
    output [4:0] out0,    //led4-0
    output [2:0] an,      //8个数码管
    output [3:0] seg,
    output ready          //led7
    );
    wire clk_cpu, io_we;
    wire [7:0] io_addr, m_rf_addr;
    wire [31:0] io_dout, io_din, rf_data, m_data, pc;

pdu_1cycle(
  .clk(clk),
  .rst(rst),

  //choose how to work
  .run(run),
  .step(step),
  .clk_cpu(clk_cpu),

  //switch input
  .valid(valid),
  .in(in),

  //led and seg output
  .check(check),  //led 5-6
  .out0(out0),    //led 4-0
  .an(an),
```

```
    .seg(seg),
    .ready(ready),  //led7

    //IO_BUS
    .io_addr(io_addr),
    .io_dout(io_dout),
    .io_we(io_we),
    .io_din(io_din),

    //Debug_BUS
    .m_rf_addr(m_rf_addr),
    .rf_data(rf_data),
    .m_data(m_data),
    .pc(pc)
);

cpu cpu (
    .clk(clk_cpu),
    .rst(rst),

    //IO_BUS
    .io_addr(io_addr),      //address of led and seg
    .io_dout(io_dout),      //led and seg output
    .io_we(io_we),          //led seg output enable
    .io_din(io_din),        //sw input

    //Debug_BUS
    .m_rf_addr(m_rf_addr),    //mem or rf address
    .rf_data(rf_data),        //rf output
    .m_data(m_data),          //mem output?
    .pc(pc)
);
endmodule
```

【约束文件】

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];


# FPGAOL LED (signle-digit-SEGPLAY)

set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports { out0[0] }];
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { out0[1] }];
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports { out0[2] }];
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports { out0[3] }];
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { out0[4] }];
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports { check[0] }];
set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports { check[1] }];
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports { ready }];


## FPGAOL SWITCH

set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports { in[0] }];
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { in[1] }];
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports { in[2] }];
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports { in[3] }];
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports {  in[4]}];
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports { valid }];
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports { run }];
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { rst }];
```

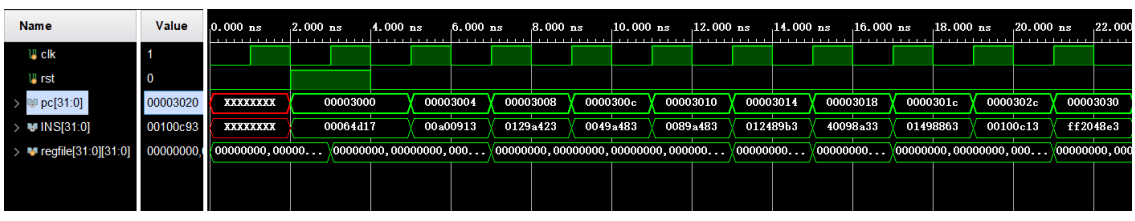
```
## FPGA0L HEXPLAY

set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports { seg[0] }];
set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVCMOS33 } [get_ports { seg[1] }];
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports { seg[2] }];
set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports { seg[3] }];
set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33 } [get_ports { an[0] }];
set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 } [get_ports { an[1] }];
set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVCMOS33 } [get_ports { an[2] }];

## FPGA0L BUTTON & SOFT_CLOCK

set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports { step }];
```

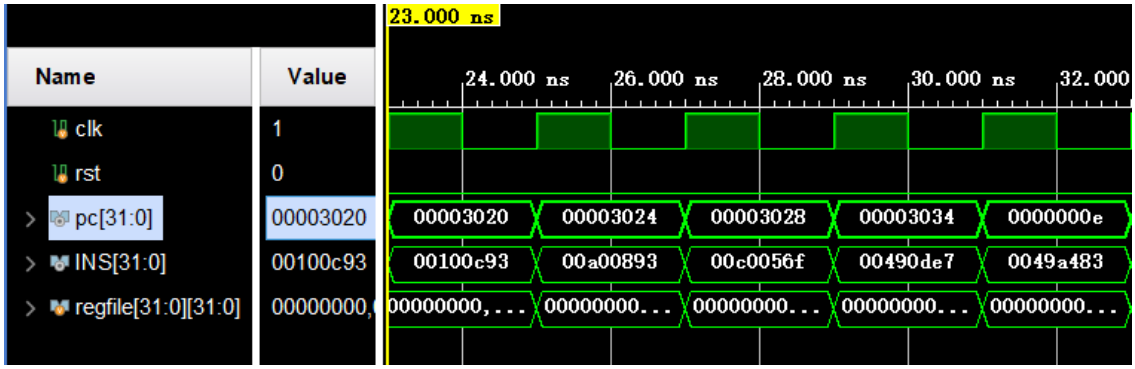
【二、10条指令测试仿真】

- asm汇编文件

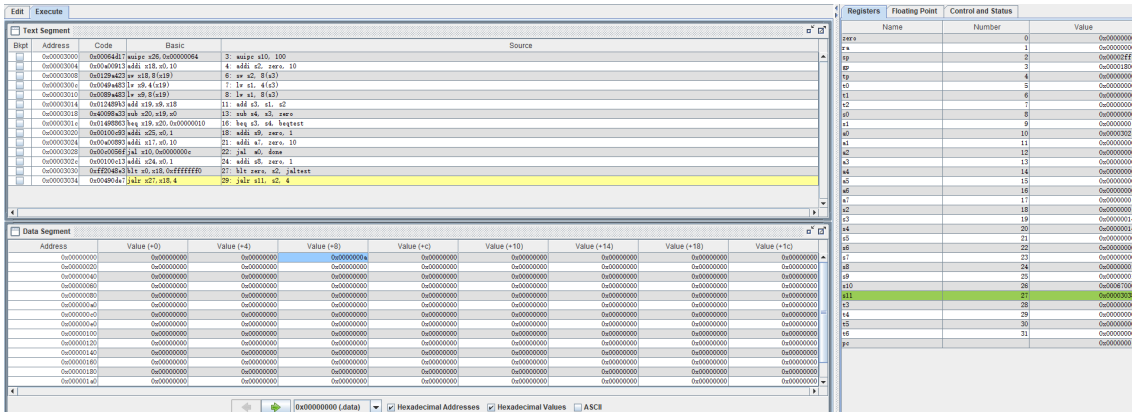
```
.text
main:
auipc s10, 100
addi s2, zero, 10
# sw and lw test
sw s2, 8(s3)
lw s1, 4(s3)
lw s1, 8(s3)
# s1 should be 10 now
# add and addi test
add s3, s1, s2
# s3 should be 20 now
sub s4, s3, zero
# s3 should be 5 now
# beq test
beq s3, s4, beqtest
jaltest:
addi s9, zero, 1
# s5 should be 1 now
exit:
addi a7, zero, 10
jal  a0, done
beqtest:
addi s8, zero, 1
# s8 should be 1 now
# jump to jaltest
blt zero, s2, jaltest
done:
jalr s11, s2, 4
```

- 生成coe文件并初始化指令存储器
- 仿真结果





- rars汇编结果



- 可以注意到每条指令都时对应的，并且取最终pc为例，仿真得到的结果与rars汇编结果一致，均为0000000e，即仿真结果正确。

【三、FPGA测试斐波那契数列】

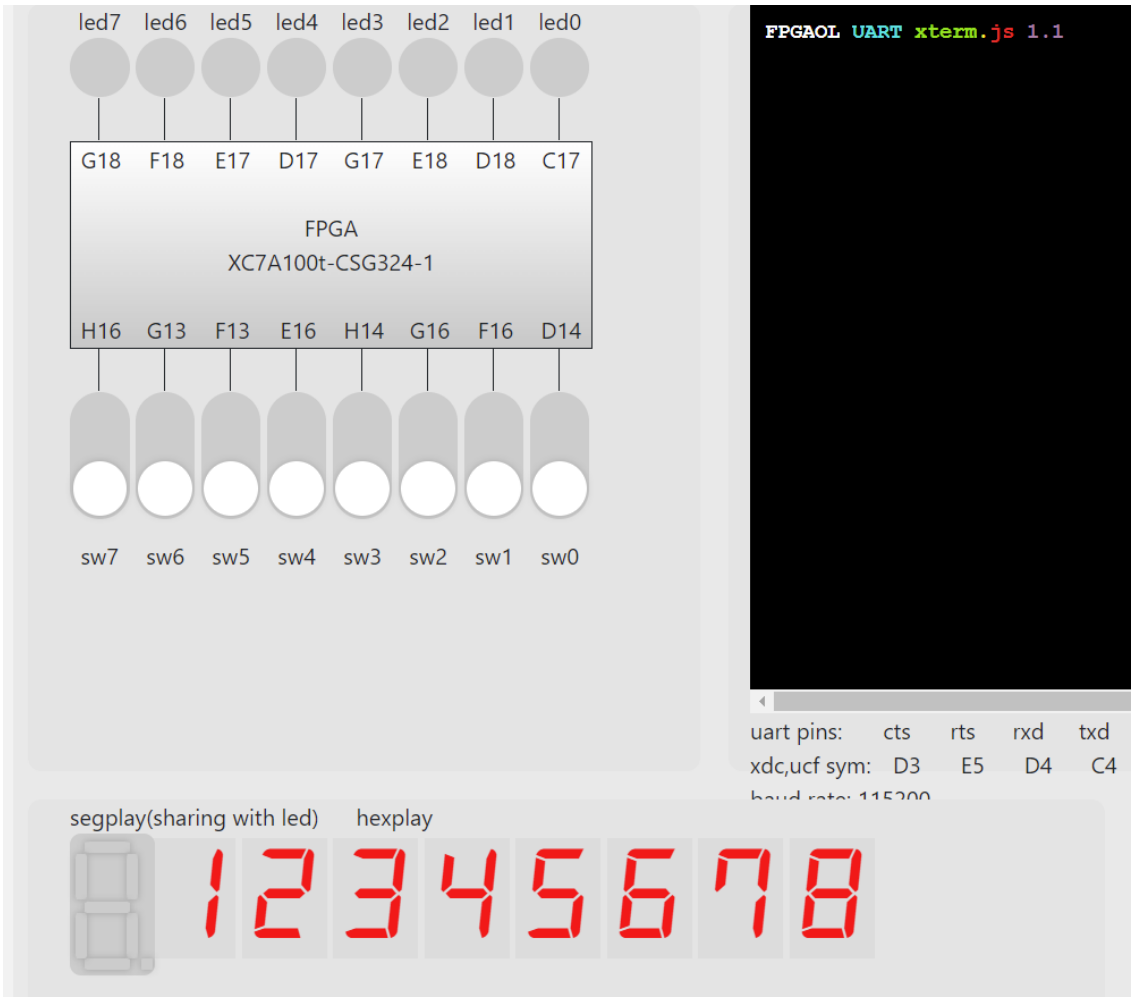
- 汇编代码

```
.text
    addi x1, x0, 1    #x1=1
    add t1, x0, x0    #store fib series @t1

#### input f0
    sw x1, 0x404(x0)  #rdy=1
l1:
    lw t0, 0x410(x0)  #wait vld=1
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv t0, a0         #for debug end

    beq t0, x0, l1
    lw s0, 0x40c(x0)  #s0=vin
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv s0, a0         #for debug end

    sw s0, 0x408(x0)  #out1=f0
    sw s0, 0(t1)      #store f0
    addi t1, t1, 4

    sw x0, 0x404(x0)  #rdy=0
l2:
    lw t0, 0x410(x0)  #wait vld=0
#    addi a7, x0, 5    #for debug begin
#    ecall
```

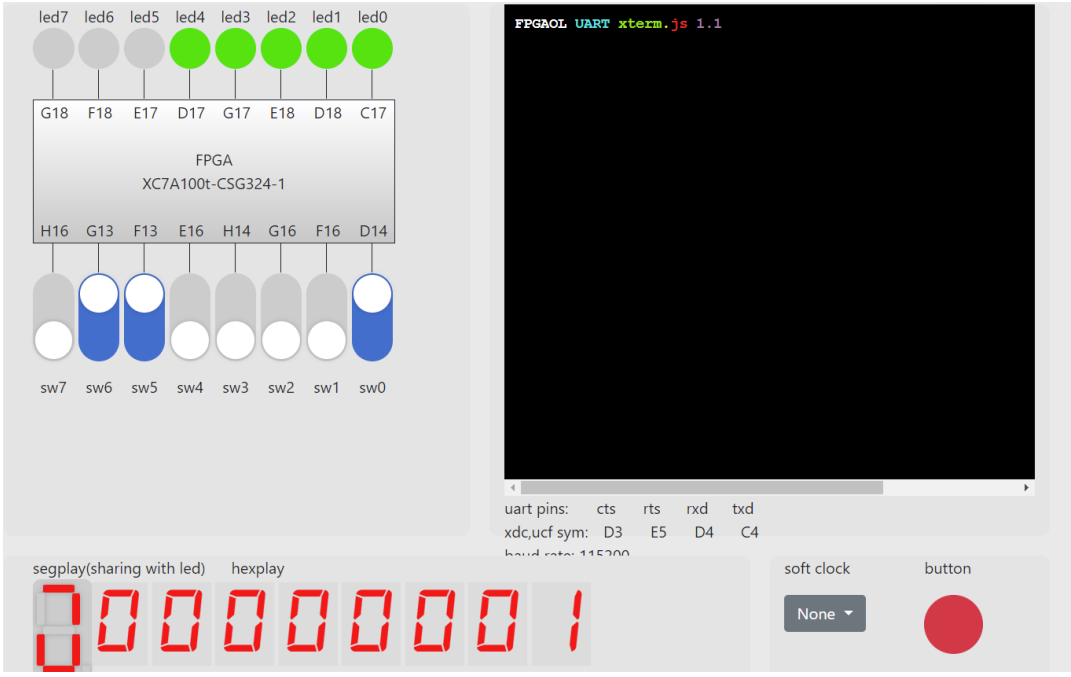
```
#    mv t0, a0          #for debug end

    beq t0, x1, l2

#### input f1
    sw x1, 0x404(x0)  #rdy=1
l3:
    lw t0, 0x410(x0)  #wait vld=1
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv t0, a0          #for debug end

    beq t0, x0, l3
    lw s1, 0x40c(x0)  #s1=vin
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv s1, a0          #for debug end

    sw s1, 0x408(x0)  #out1=f1
    sw s1, 0(t1)      #store f1
    addi t1, t1, 4

    sw x0, 0x404(x0)  #rdy=0
l4:
    lw t0, 0x410(x0)  #wait vld=0
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv t0, a0          #for debug end

    beq t0, x1, l4

#### comput fi = fi-2 + fi-1
next:
    add t0, s0, s1    #fi
    sw t0, 0x408(x0)  #out1=fi
    sw t0, 0(t1)      #store fi
    addi t1, t1, 4

    add s0, x0, s1
    add s1, x0, t0

    sw x1, 0x404(x0)  #rdy=1
l5:
    lw t0, 0x410(x0)  #wait vld=1
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv t0, a0          #for debug end

    beq t0, x0, l5
    sw x0, 0x404(x0)  #rdy=0
l6:
    lw t0, 0x410(x0)  #wait vld=0
#    addi a7, x0, 5    #for debug begin
#    ecall
#    mv t0, a0          #for debug end

    beq t0, x1, l6
    jal x0, next
```

事实上，每一个beq在没有输入的情况下都是一个无限循环，只有在对应输入发生变化，改变了vld或rdy的值时，才会进入到下一阶段，即实现了全互锁方式——只有双方都有应答才能继续执行。vld通过valid按钮输入。

- 烧写结果

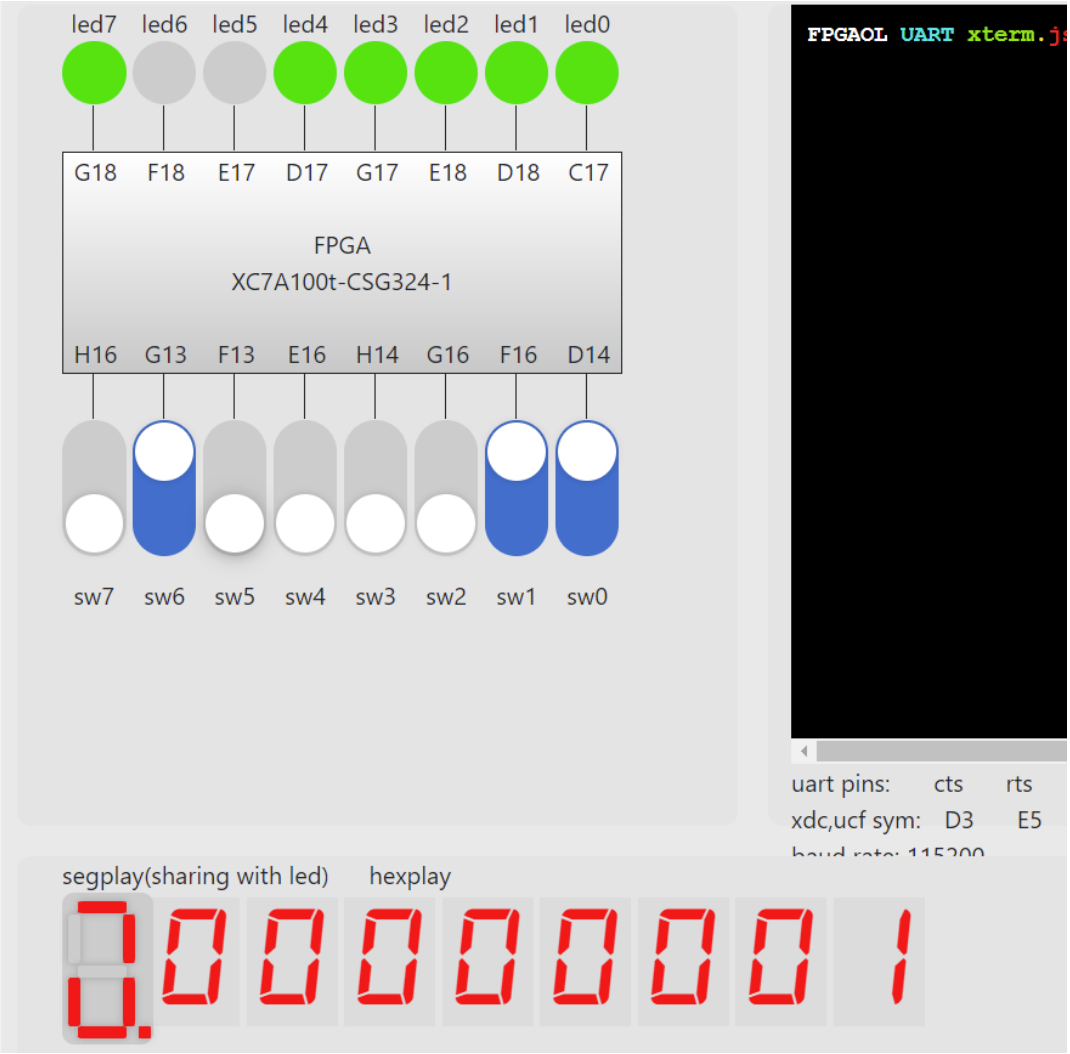


  - 输入第一项，按动valid以结束循环



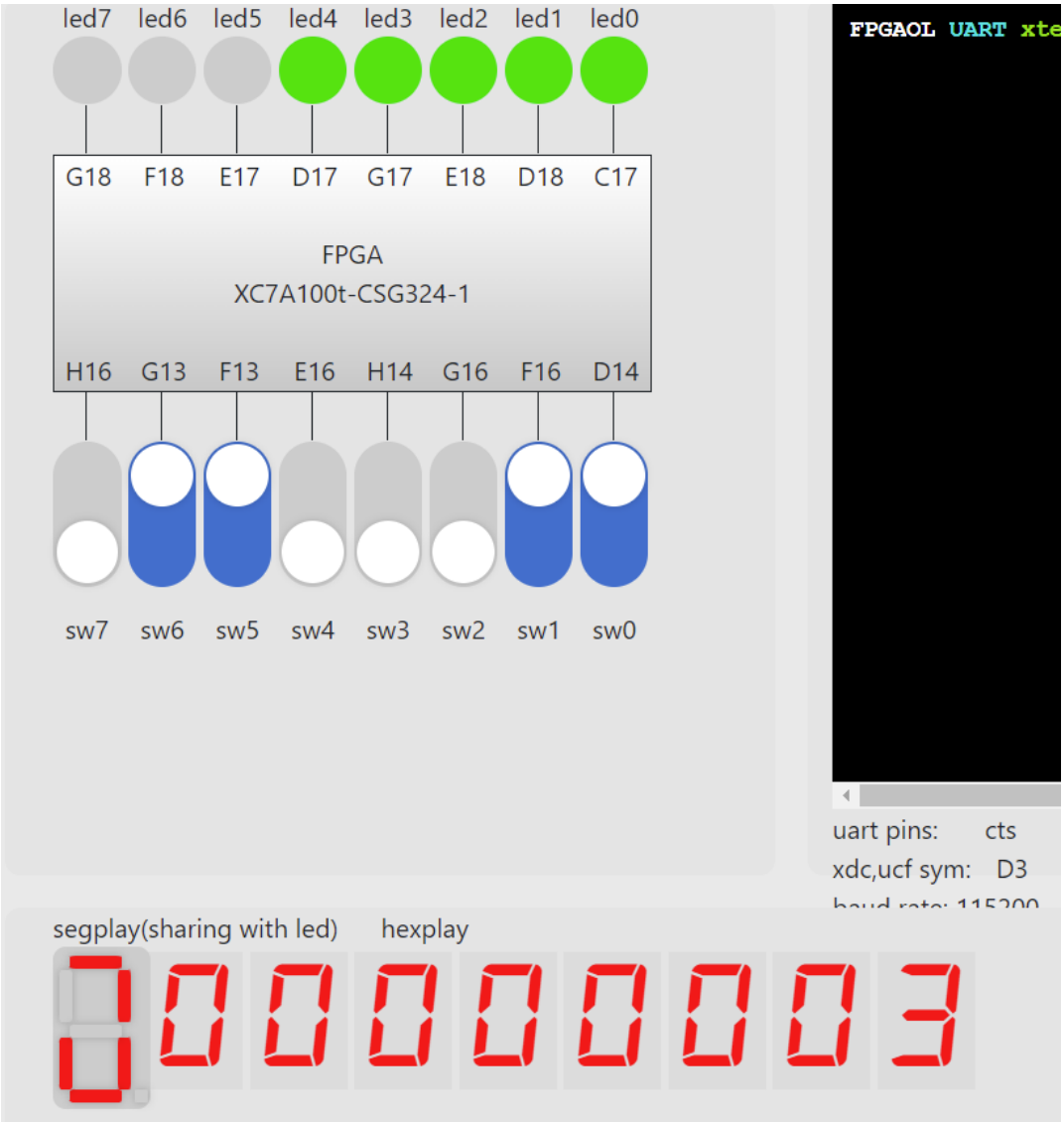
    此时同样进入下一个循环，需要使vld = 0才能接受下一次输入

  -

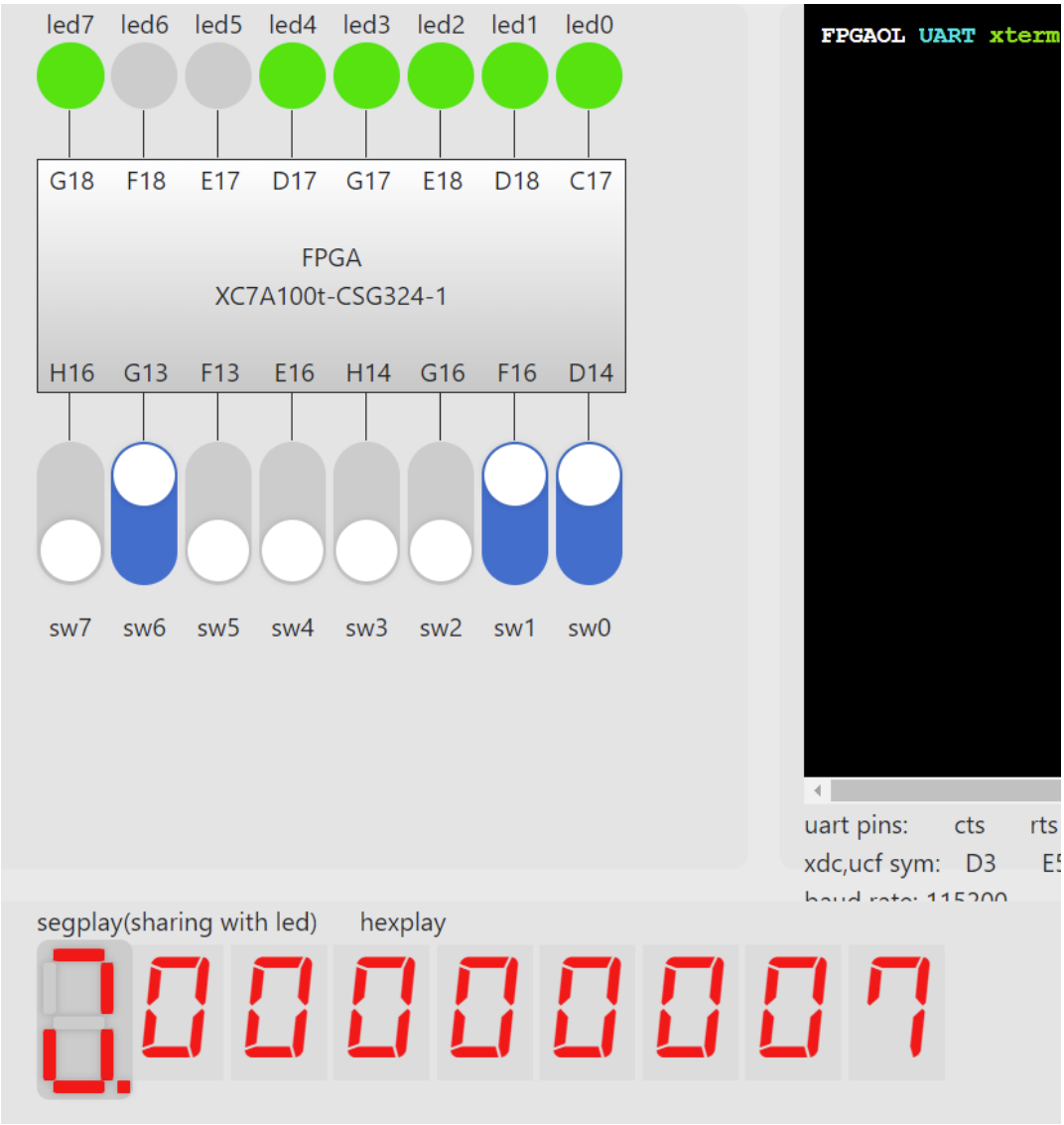


此时rdy信号亮，可以进行第二项的输入。

- 输入第二项


led7 led6 led5 led4 led3 led2 led1 led0
G18 F18 E17 D17 G17 E18 D18 C17
FPGA
XC7A100t-CSG324-1
H16 G13 F13 E16 H14 G16 F16 D14
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
FPGAOL UART xte
uart pins: cts
xdc,ucf sym: D3
segplay(sharing with led) hexplay
0 0 0 0 0 0 0 3

- 逐步运行



- 即最终烧写结果正确。

【总结与思考】

- 实验总结
  - 老师后续给出了数据通路，所以事实上是依葫芦画瓢，将对应的数据通路用verilog语言进行描述即可，难点在于理解各通路信号的含义，以及额外增加的四条指令需要怎样改变数据通路使得能够正确运行；
  - 事实上在烧写中第一次无法成功就是因为blt的误判使得虽然不满足beq的条件但是满足了blt的条件所以自动跳转了，所以多加了一个控制信号用于判断blt；
  - 学习了全互锁方式的实现与处理。
  - 其实相比之下，只要有完整数据通路就不难完成，想要完成选做只需要深入理解一下pdu和老师所给的测试文件的代码，总体来说难度适中。