计算机组成原理 实验报告

> 姓名：张艺耀
>
> 学号：PB20111630
>
> 实验日期：2022- -

实验题目

单周期CPU设计

实验目的

理解CPU的结构和工作原理

掌握单周期CPU的设计和调试方法

熟练掌握数据通路和控制器的设计和描述方法

实验平台

FPGAOL

Vivado

Mac + VSCode-remote + SSH + VLab

实验过程

Step 1 : 实现单周期RISC-V CPU 可执行以下十条命令(add addi sub auipc lw sw beq blt jal jalr)

本CPU采用与Ripes基本相同的数据通路 添加了多个mux和io_bus debug_bus。

top.v

```verilog
module top(
    input clk,
    input rst,

    //选择CPU工作方式
    input run,
    input step,

    //输入switch的端口
    input valid,
    input [4:0] in,

    //输出led和seg的端口
    output [1:0] check,  //led6-5:查看类型
    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;

wire [7:0] io_addr;
wire [31:0] io_dout;
wire io_we;
wire [31:0] io_din;

  wire [7:0] m_rf_addr;
wire [31:0] rf_data;
wire [31:0] m_data;
  wire [31:0] pc_out;

pdu pdu(clk, rst, run, step, clk_cpu, valid, in, check, out0, an, seg, ready, io_addr, io_dout, io_we, io_din, m_rf_addr, rf_data, m_data, pc_out);
cpu cpu(clk_cpu, rst, io_addr, io_dout, io_we, io_din, m_rf_addr, rf_data, m_data, pc_out);

endmodule
```

pdu.v

采用提供的源码。

```verilog
module  pdu(
  input clk,
  input rst,

  //选择CPU工作方式
  input run,
  input step,
  output clk_cpu,

  //输入switch的端口
  input valid,
  input [4:0] in,

  //输出led和seg的端口
  output [1:0] check,  //led6-5:查看类型
  output [4:0] out0,    //led4-0
  output [2:0] an,     //8个数码管
  output [3:0] seg,
  output ready,          //led7

  //IO_BUS
  input [7:0] io_addr,
  input [31:0] io_dout,
  input io_we,
  output [31:0] io_din,

  //Debug_BUS
  output [7:0] m_rf_addr,
  input [31:0] rf_data,
  input [31:0] m_data,
  input [31:0] pc
);

reg [4:0] in_r;    //同步外部输入用
reg run_r, step_r, step_2r, valid_r, valid_2r;
wire step_p, valid_pn;  //取边沿信号

reg clk_cpu_r;      //寄存器输出CPU时钟
reg [4:0] out0_r;   //输出外设端口
reg [31:0] out1_r;
reg ready_r;
reg [19:0] cnt;     //刷新计数器，刷新频率约为95Hz
reg [1:0] check_r;  //查看信息类型, 00-运行结果，01-寄存器堆，10-存储器，11-PC

reg [7:0] io_din_a; //_a表示为满足组合always描述要求定义的，下同
reg ready_a;
reg [4:0] out0_a;
reg [31:0] out1_a;
reg [3:0] seg_a;

assign clk_cpu = clk_cpu_r;
assign io_din = io_din_a;
assign check = check_r;
assign out0 = out0_a;
assign ready = ready_a;
assign seg = seg_a;
assign an = cnt[19:17];
assign step_p = step_r & ~step_2r;     //取上升沿
assign valid_pn = valid_r ^ valid_2r;  //取上升沿或下降沿
assign m_rf_addr = {{3{1'b0}}, in_r};

//同步输入信号
always @(posedge clk) begin
  run_r <= run;
  step_r <= step;
  step_2r <= step_r;
  valid_r <= valid;
  valid_2r <= valid_r;
  in_r <= in;
end

//CPU工作方式
always @(posedge clk, posedge rst) begin
  if(rst)
    clk_cpu_r <= 0;
  else if (run_r)
    clk_cpu_r <= ~clk_cpu_r;
  else
    clk_cpu_r <= step_p;
end

//读外设端口
always @* begin
  case (io_addr)
    8'h0c: io_din_a = {{27{1'b0}}, in_r};
    8'h10: io_din_a = {{31{1'b0}}, valid_r};
    default: io_din_a = 32'h0000_0000;
  endcase
end

//写外设端口
always @(posedge clk, posedge rst) begin
if (rst) begin
  out0_r <= 5'h1f;
  out1_r <= 32'h1234_5678;
  ready_r <= 1'b1;
end
else if (io_we)
  case (io_addr)
    8'h00: out0_r <= io_dout[4:0];
    8'h04: ready_r <= io_dout[0];
    8'h08: out1_r <= io_dout;
    default: ;
  endcase
end

//LED和数码管查看类型
always @(posedge clk, posedge rst) begin
if(rst)
    check_r <= 2'b00;
```

```verilog
  else if(run_r)
    check_r <= 2'b00;
  else if (step_p)
    check_r <= 2'b00;
  else if (valid_pn)
    check_r <= check - 2'b01;
end

//LED和数码管显示内容
always @* begin
  ready_a = 1'b0;
  case (check_r)
    2'b00: begin
      out0_a = out0_r;
      out1_a = out1_r;
      ready_a = ready_r;
    end
    2'b01: begin
      out0_a = in_r;
      out1_a = rf_data;
    end
    2'b10: begin
      out0_a = in_r;
      out1_a = m_data;
    end
    2'b11: begin
      out0_a = 5'b00000;
      out1_a = pc;
    end
  endcase
end

//扫描数码管
always @(posedge clk, posedge rst) begin
  if (rst) cnt <= 20'h0_0000;
  else cnt <= cnt + 20'h0_0001;
end

always @* begin
  case (an)
    3'd0: seg_a = out1_a[3:0];
    3'd1: seg_a = out1_a[7:4];
    3'd2: seg_a = out1_a[11:8];
    3'd3: seg_a = out1_a[15:12];
    3'd4: seg_a = out1_a[19:16];
    3'd5: seg_a = out1_a[23:20];
    3'd6: seg_a = out1_a[27:24];
    3'd7: seg_a = out1_a[31:28];
    default: ;
  endcase
end

```

```
endmodule
```

cpu.v

CPU主模块 为整个数据通路整体的框架。

```
module cpu(
    input clk,
    input rst,

    //io_bus
    output [31:0] io_addr,    //led和seg的地址
    output [31:0] io_dout,  //输出led和seg的数据
    output io_we,    //输出led和seg数据时的使能信号
    input [31:0] io_din,    //输出led和seg数据时的使能信号

    //debug_bus
    input [7:0] m_rf_addr,  //存储器（mem）或寄存器堆（rf）的调试读口地址
    output [31:0] rf_data,  //从rf读取的数据
    output [31:0] m_data,   //从mem读取的数据
    output [31:0] pc_out    //pc的内容
);

wire branch;
wire [31:0] pc;
wire [31:0] pc_in;
wire [31:0] pc_plus4;
wire [31:0] inst;
wire [31:0] imm_out;
wire rf_wr_en;
wire alu_a_sel;
wire alu_b_sel;
wire [1:0] alu_ctrl;
wire dm_rd_ctrl;
wire dm_wr_ctrl;
wire dm_wr_ctrl_aft;
wire [1:0] rf_wr_sel;
wire [2:0] comp_ctrl;
wire do_branch;
wire do_jump;

reg [31:0] rf_wd;
wire [31:0] rf_rd0, rf_rd1;
wire [31:0] alu_a, alu_b, alu_out;
wire [31:0] dm_dout;
wire [31:0] dm_dout_aft;

assign alu_a = alu_a_sel ? rf_rd0 : pc;
assign alu_b = alu_b_sel ? imm_out : rf_rd1;
assign pc_in = (pc >= 32'h3000 && pc <= 32'h33ff) ? pc : 32'h0;
assign pc_out = pc_in;
```

```verilog

//io_bus
assign dm_wr_ctrl_aft = (~(io_addr[10])) & dm_wr_ctrl;  //AND gate
assign dm_dout_aft = io_addr[10] ? io_din : dm_dout;    //Mux
assign io_dout = rf_rd1;
assign io_we = io_addr[10] & dm_wr_ctrl;    //Mux
assign io_addr = alu_out;

program_counter program_counter(clk, rst, branch, alu_out, pc, pc_plus4);
alu alu(alu_a, alu_b, alu_ctrl, alu_out);
register_file register_file(.clk (clk), .rst (rst), .ra0 (inst[19:15]), .ra1 (inst[24:20]), .ra2 (m_rf_addr[4:0]), .wa (inst[11:7]), .wd (rf_wd), .we (rf_wr_en), .rd0 (rf_rd0), .rd1 (rf_rd1), .rd2 (rf_data));
imm imm(.inst (inst), .imm_out (imm_out));
br br(.a (rf_rd0), .b (rf_rd1), .comp_ctrl (comp_ctrl), .do_branch (do_branch), .do_jump (do_jump), .branch (branch));
controller controller(inst, rf_wr_en, alu_a_sel, alu_b_sel, alu_ctrl, dm_rd_ctrl, dm_wr_ctrl, rf_wr_sel, comp_ctrl, do_branch, do_jump);

ins_mem ins_mem(.a (pc_in[9:2]), .spo (inst));
data_mem data_mem(.a (alu_out[9:2]), .d (rf_rd1), .dpra (m_rf_addr), .clk (clk), .we (dm_wr_ctrl_aft), .spo (dm_dout), .dpo (m_data));

//Mux
always@(*) begin
    case (rf_wr_sel)
        2'b00: rf_wd = 32'h0;
        2'b01: rf_wd = pc_plus4;
        2'b10: rf_wd = alu_out;
        2'b11: rf_wd = dm_dout_aft;
        default: rf_wd = 32'h0;
    endcase
end

endmodule
```

program_counter.v

PC模块 rst为1时重置初始地址为 `0x3000`。

```verilog
module program_counter(
    input clk,
    input rst,
    input br,
    input [31:0] alu_out,
    output reg [31:0] pc,
    output [31:0] pc_plus4
);

always@(posedge clk or posedge rst) begin
    if(rst) pc <= 32'h3000;
    else if(br) pc <= alu_out;
    else pc <= pc_plus4;
end

assign pc_plus4 = pc + 32'h4;

endmodule
```

alu.v

ALU（算术逻辑运算单元） 由于本次实验的运算操作仅涉及到加减，故省略了逻辑操作和移位操作等。

```
module alu(
    input [31:0] a,b,
    input [1:0] alu_ctrl,
    output reg [31:0] alu_out
);

always@(*) begin
    case(alu_ctrl)
    4'h0: alu_out = a + b;
    4'h1: alu_out = a - b;
    default: alu_out = 32'h0;
    endcase
end

endmodule
```

register.v

三端口寄存器（端口3用于连接io总线访问外设）。

```
module register_file (
    input clk, rst,
    input [4:0] ra0,    //读端口0地址
    output [31:0] rd0,   //读端口0数据

    input [4:0] ra1,    //读端口1地址
    output [31:0] rd1,   //读端口1数据

    input [4:0] ra2,    //读端口2地址
    output [31:0] rd2,  //读端口2数据

    input [4:0] wa, //写端口地址
    input we,   //写使能 高电平有效
    input [31:0] wd //写端口数据
);

reg [31:0] regfile [0:31];
assign rd0 = ra0 ? regfile[ra0] : 32'h0;
assign rd1 = ra1 ? regfile[ra1] : 32'h0;
```

```verilog
assign rd2 = ra2 ? regfile[ra2] : 32'h0;
integer i;

always @(posedge clk) begin
    if(rst) begin
        for(i = 0; i <= 31; i = i + 1) begin
            regfile[i] <= 32'b0;
        end
    end
    else if(we) begin
        if(wa == 0) regfile[wa] <= 0;   //r0内容恒为0
        else regfile[wa] <= wd;
    end
end

endmodule
```

imm.v

立即数扩展模块。

```verilog
module imm(
    input [31:0] inst,
    output reg [31:0] imm_out
);
wire [6:0] inst_type;

assign inst_type = inst[6:0];

always@(*) begin
    case(inst_type)
    7'h03: imm_out = {{21{inst[31]}}, inst[30:20]}; //i_type:lw
    7'h13: imm_out = {{21{inst[31]}}, inst[30:20]}; //i_type_logic
    7'h17: imm_out = {inst[31:12], 12'h0};  //aupic
    7'h37: imm_out = {inst[31:12], 12'h0};  //lui
    7'h6f: imm_out = {{13{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};    //jal
    7'h63: imm_out = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};  //b_type
    7'h23: imm_out = {{21{inst[31]}}, inst[30:25], inst[11:7]}; //s_type
    7'h67: imm_out = {{21{inst[31]}}, inst[30:20]}; //jalr
    default: imm_out = 32'h0;
    endcase
end

endmodule
```

br.v

branch模块，根据控制模块输出comp_ctrl决定是否跳转。

```
module br(
    input [31:0] a,
    input [31:0] b,
    input [2:0] comp_ctrl,
    input do_branch,
    input do_jump,
    output branch
);
reg taken;

always@(*) begin
    case(comp_ctrl)
    3'h0: taken = (a == b);
    3'h4: taken = (a < b);
    default: taken = 0;
    endcase
end

assign branch = (taken && do_branch) || do_jump;

endmodule
```

controller.v

控制模块，解析指令类型并产生相应控制信号。

```
module controller(
    input [31:0] inst,
    output rf_wr_en, alu_a_sel, alu_b_sel,
    output reg [1:0] alu_ctrl,
    output reg dm_rd_ctrl,
    output reg dm_wr_ctrl,
    output reg [1:0] rf_wr_sel,
    output [2:0] comp_ctrl,
    output do_branch, do_jump
);

wire [6:0] opcode;
wire [2:0] funct3;
wire [6:0] funct7;

wire is_add;
wire is_addi;
wire is_sub;
wire is_auipc;
wire is_lw;
wire is_sw;
```

```verilog
wire is_beq;
wire is_blt;
wire is_jal;
wire is_jalr;

wire is_add_type;
wire is_u_type;
wire is_jump_type;
wire is_b_type;
wire is_r_type;
wire is_i_type;
wire is_s_type;

assign opcode = inst[6:0];
assign funct3 = inst[14:12];
assign funct7 = inst[31:25];

assign is_add = (opcode == 7'h33) && (funct3 == 3'h0) && (funct7 == 7'h0);
assign is_addi = (opcode == 7'h13) && (funct3 == 3'h0);
assign is_sub = (opcode == 7'h33) && (funct3 == 3'h0) && (funct7 == 7'h20);
assign is_auipc = (opcode == 7'h17);
assign is_lw = (opcode == 7'h03) && (funct3 == 3'h2);
assign is_sw = (opcode == 7'h23) && (funct3 == 3'h2);
assign is_beq = (opcode == 7'h63) && (funct3 == 3'h0);
assign is_blt = (opcode == 7'h63) && (funct3 == 3'h4);
assign is_jal = (opcode == 7'h6f);
assign is_jalr = (opcode == 7'h67) && (funct3 == 3'h0);

assign is_add_type = is_auipc | is_jal | is_jalr | is_b_type | is_s_type | is_lw | is_add
| is_addi;
assign is_u_type = is_auipc;
assign is_jump_type = is_jal;
assign is_b_type = is_beq | is_blt;
assign is_r_type = is_add | is_sub;
assign is_i_type = is_jalr | is_lw | is_addi;
assign is_s_type = is_sw;

always@(*) begin
    if(is_add_type) alu_ctrl = 2'h0;
    else if(is_sub) alu_ctrl = 2'h1;
end

assign rf_wr_en = is_u_type | is_jump_type | is_r_type | is_i_type;
assign alu_a_sel = is_r_type | is_i_type | is_s_type;
assign alu_b_sel = ~ is_r_type;

always@(*) begin
    if(is_lw) dm_rd_ctrl = 1'b1;
    else dm_rd_ctrl = 1'b0;
end

always@(*) begin
```

```
    if(is_sw) dm_wr_ctrl = 1'b1;
    else dm_wr_ctrl = 1'b0;
end

always@(*) begin
    if(opcode == 7'h3) rf_wr_sel = 2'h3;
    else if(((~ is_jalr) & is_i_type) | is_u_type | is_r_type) rf_wr_sel = 2'h2;
    else if(is_jal | is_jalr) rf_wr_sel = 2'h1;
    else rf_wr_sel = 2'h0;
end

assign comp_ctrl = funct3;
assign do_branch = is_b_type;
assign do_jump = is_jal | is_jalr;

endmodule
```

ins_mem.v 和 data_mem.v

采用ip核例化，ins_mem选用256*32ROM、data_mem选用256*32的双端口RAM。

> 文章 https://cloud.tencent.com/developer/article/1814213 介绍了不同端口的分布式和块式RAM的区别。

Step 2 : 对10条指令进行仿真测试

在RARS中编写汇编测试文件（根据上一实验改写而成 新增了四条指令）

test.asm

```
.data
out: .word 0xff
in: .word 0xff

.text
la a0, out  #auipc
sw x0, 0(a0)  #sw 0 in address out
addi x5, x0, 0xf0 #addi
lw x5, 4(a0)  #lw

li x6, 0xf
li x7, 0xf0
add x5, x6, x7  #add
sub x5, x6, x7  #sub

li x7, 0xf
beq x6, x7, equal  #beq
jal exit  #jal

```

```
equal:
li x7, 0xf0
blt x6, x7, lessthan #blt
jal exit

lessthan:
la a0, exit
jalr a0 #jalr

exit:
li a7, 10
ecall
```

生成 `ins.coe` 和 `data.coe`

ins.coe

```
memory_initialization_radix = 16;
memory_initialization_vector =
ffffd517
00050513
00052023
0f000293
00452283
00f00313
0f000393
007302b3
407302b3
00f00393
00730463
01c000ef
0f000393
00734463
010000ef
00000517
00c50513
000500e7
00a00893
00000073
```

data.coe

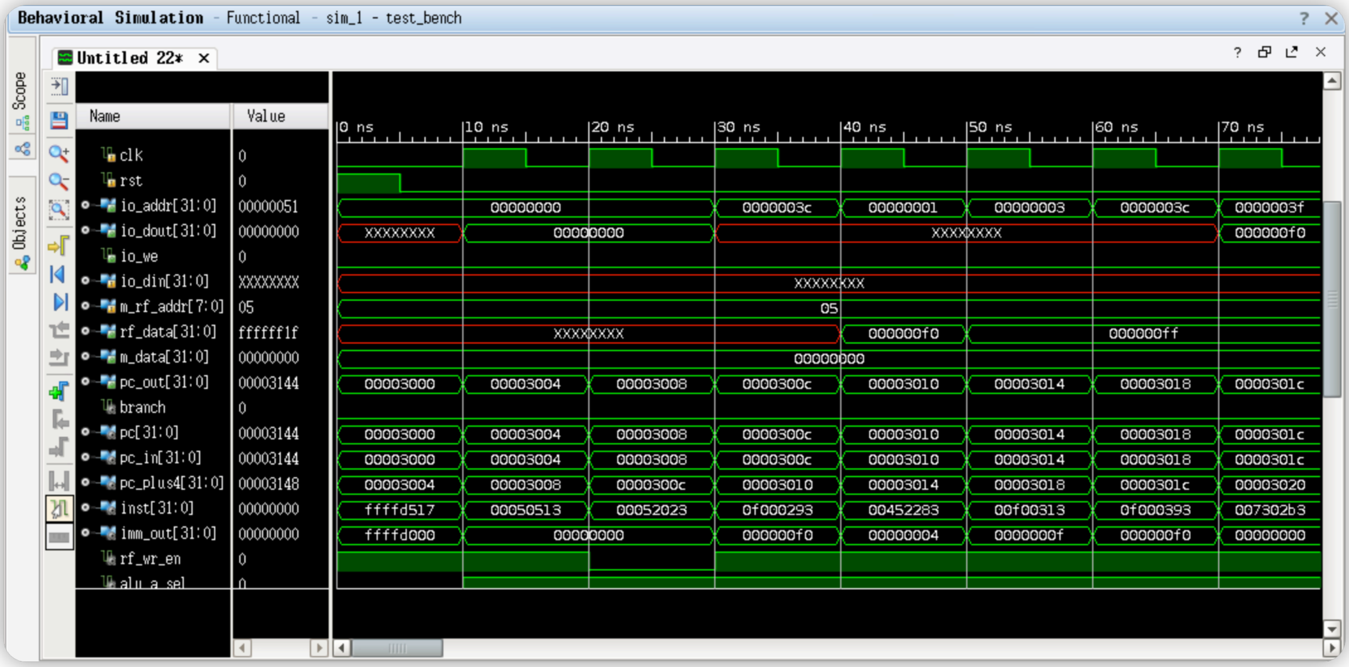
```
memory_initialization_radix = 16;
memory_initialization_vector =
000000ff
000000ff
```

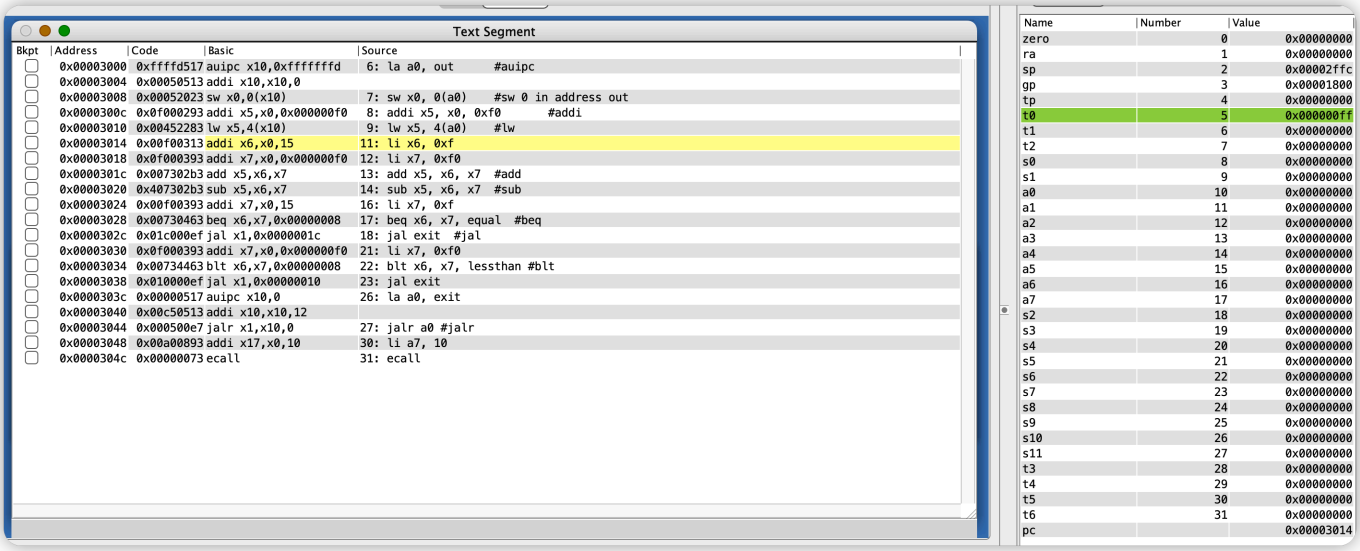
仿真

test_bench.v

首先查看 `x5` 寄存器值。

```verilog
module test_bench();
reg clk;
reg rst;

wire [31:0] io_addr;
wire [31:0] io_dout;
wire io_we;
reg [31:0] io_din;

reg [7:0] m_rf_addr;
wire [31:0] rf_data;
wire [31:0] m_data;
wire [31:0] pc_out;

cpu cpu(clk, rst, io_addr, io_dout, io_we, io_din, m_rf_addr, rf_data, m_data, pc_out);

initial begin
    rst = 1; m_rf_addr = 5;
    clk = 0;#5
    rst = 0;

    forever #5 clk = ~clk;
end

initial begin
    #800 $finish;
end

endmodule
```




Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00003000	0xffffd517	auipc x10,0xfffffffd	6: la a0, out #auipc
☐	0x00003004	0x00050513	addi x10,x10,0	
☐	0x00003008	0x00052023	sw x0,0(x10)	7: sw x0, 0(a0) #sw 0 in address out
☐	0x0000300c	0x0f000293	addi x5,x0,0x000000f0	8: addi x5, x0, 0xf0 #addi
☐	0x00003010	0x00452283	lw x5,4(x10)	9: lw x5, 4(a0) #lw
☐	0x00003014	0x00f00313	addi x6,x0,15	11: li x6, 0xf
☐	0x00003018	0x0f000393	addi x7,x0,0x000000f0	12: li x7, 0xf0
☐	0x0000301c	0x007302b3	add x5,x6,x7	13: add x5, x6, x7 #add
☐	0x00003020	0x407302b3	sub x5,x6,x7	14: sub x5, x6, x7 #sub
☐	0x00003024	0x00f00393	addi x7,x0,15	16: li x7, 0xf
☐	0x00003028	0x00730463	beq x6,x7,0x00000008	17: beq x6, x7, equal #beq
☐	0x0000302c	0x01c000ef	jal x1,0x0000001c	18: jal exit #jal
☐	0x00003030	0x0f000393	addi x7,x0,0x000000f0	21: li x7, 0xf0
☐	0x00003034	0x00734463	blt x6,x7,0x00000008	22: blt x6, x7, lessthan #blt
☐	0x00003038	0x010000ef	jal x1,0x00000010	23: jal exit
☐	0x0000303c	0x00000517	auipc x10,0	26: la a0, exit
☐	0x00003040	0x00c50513	addi x10,x10,12	
☐	0x00003044	0x000500e7	jalr x1,x10,0	27: jalr a0 #jalr
☐	0x00003048	0x00a00893	addi x17,x0,10	30: li a7, 10
☐	0x0000304c	0x00000073	ecall	31: ecall

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00002ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x000000ff
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00003014

运行结果一致

Step 3 : 对fib进行仿真测试

将 `test_bench.v` 中的 `m_rf_addr` 改为0（fib输出地址）。

`m_rf_addr + 4` 的地址为输入数据地址 这里在汇编文件中初始化为5，即计算斐波那契数列第五项的值存到地址0。

将RARS生成的 `fib_ins.coe` 和 `fib_data.coe` 文件分别导入 `ins_mem` 和 `data_mem` 中

fib.asm

```asm
.data
out: .word 0
in: .word 2

.text
li t1, 1
li t2, 2
la a0, out
lw t3, 4(a0)
beq t1, t3, first
beq t2, t3, second
addi t3, t3, -2
loop:
add t4, t1, t2
addi t1, t2, 0
addi t2, t4, 0
addi t3, t3, -1
bgtz t3, loop
sw t4, 0(a0)
jal exit

first:
sw t1, 0(a0)
jal exit

second:
sw t2, 0(a0)
jal exit

exit:
li a7, 10
ecall
```

fib_data.coe

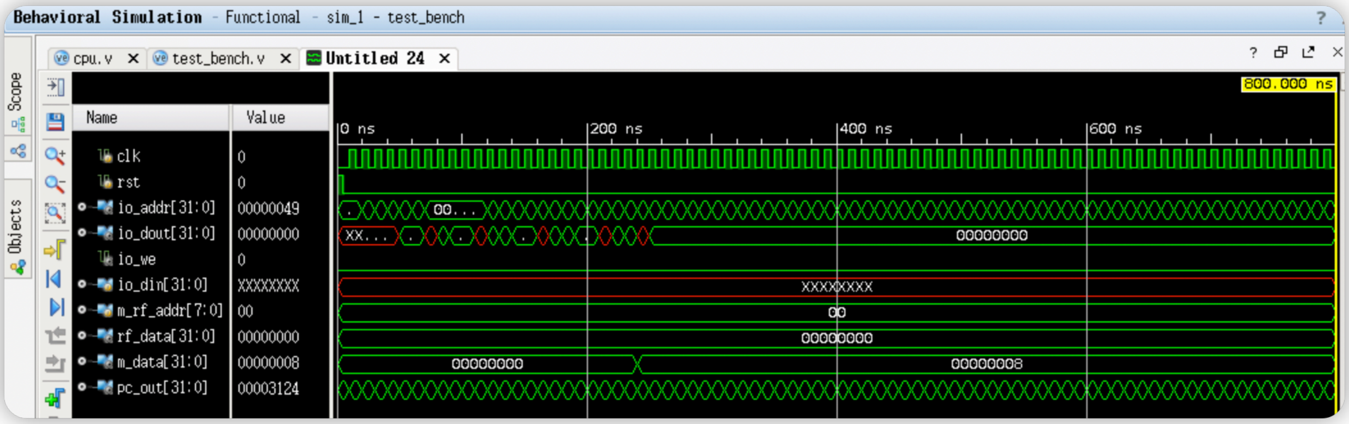
```
memory_initialization_radix = 16;
memory_initialization_vector =
00000000
00000005
```

fib_ins.coe

```
memory_initialization_radix = 16;
memory_initialization_vector =
00100313
00200393
ffffd517
ff850513
```

```
00452e03
03c30463
03c38663
ffee0e13
00730eb3
00038313
000e8393
fffe0e13
ffc048e3
01d52023
014000ef
00652023
00c000ef
00752023
004000ef
00a00893
00000073
```

仿真波形如下：



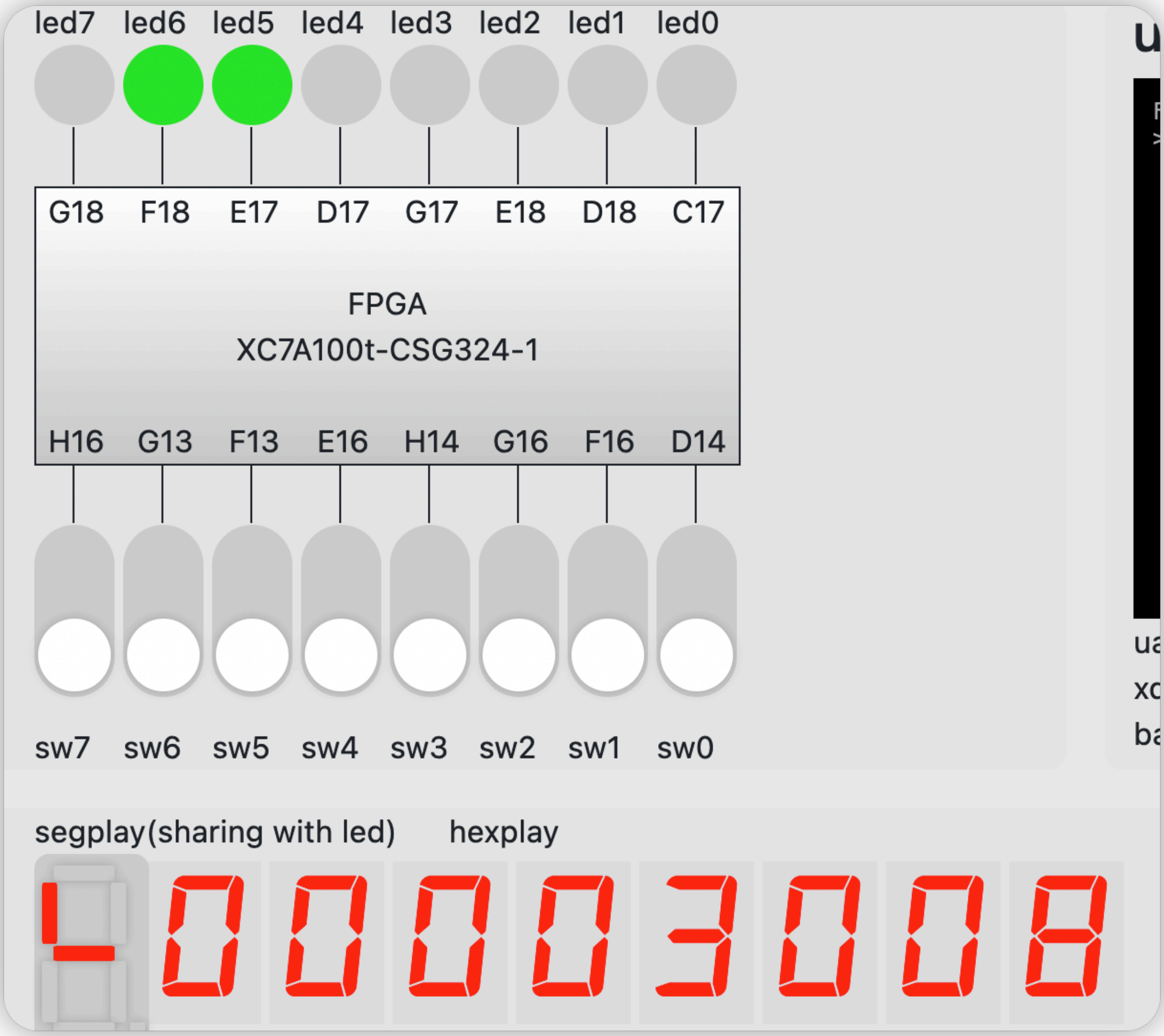
Name	Value
clk	0
rst	0
io_addr[31:0]	00000049
io_dout[31:0]	00000000
io_we	0
io_din[31:0]	XXXXXXXX
m_rf_addr[7:0]	00
rf_data[31:0]	00000000
m_data[31:0]	00000008
pc_out[31:0]	00003124

`rf_data` 中读出的结果为8 正确。

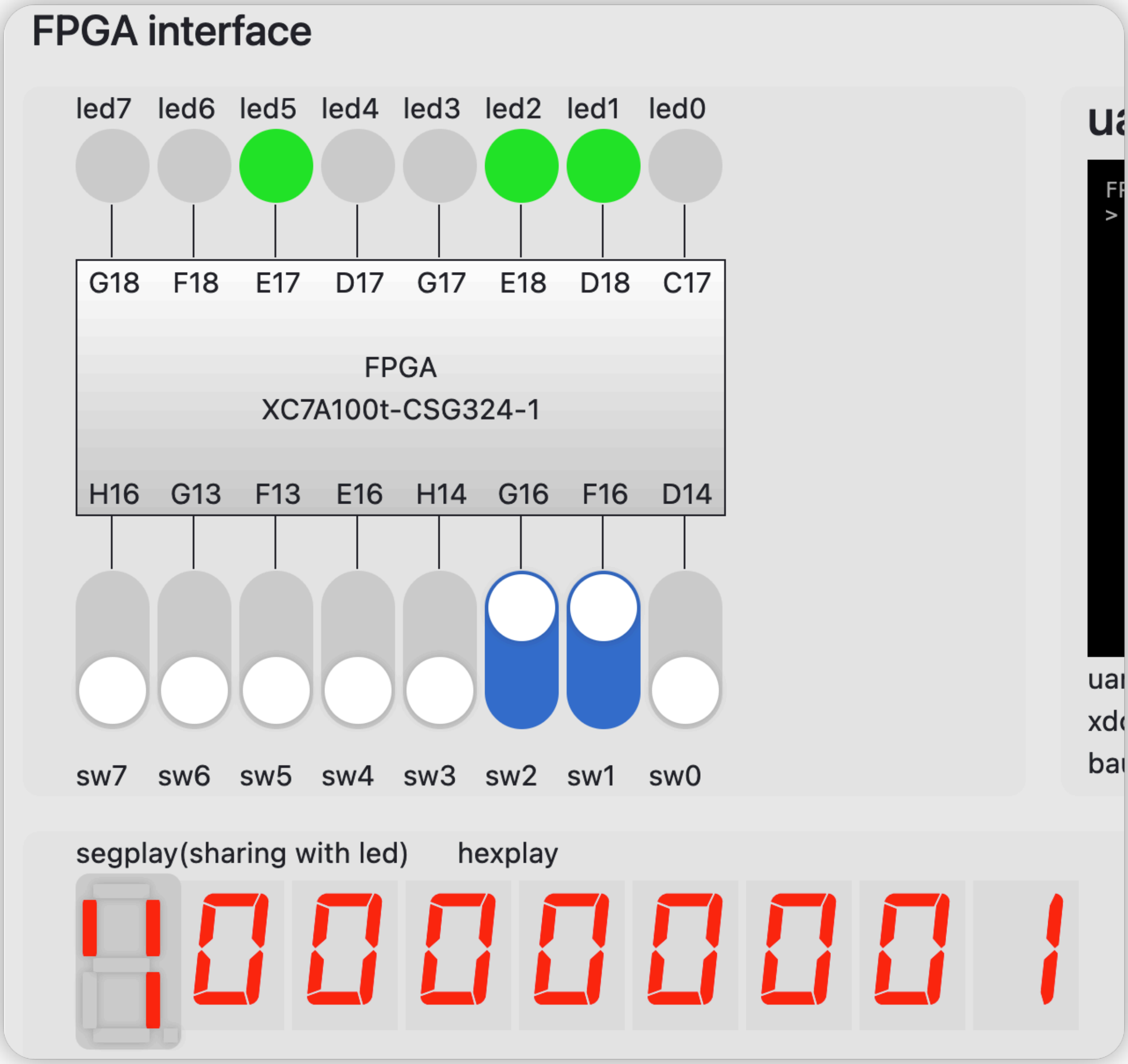
实验结果

测试Lab3 fib.s

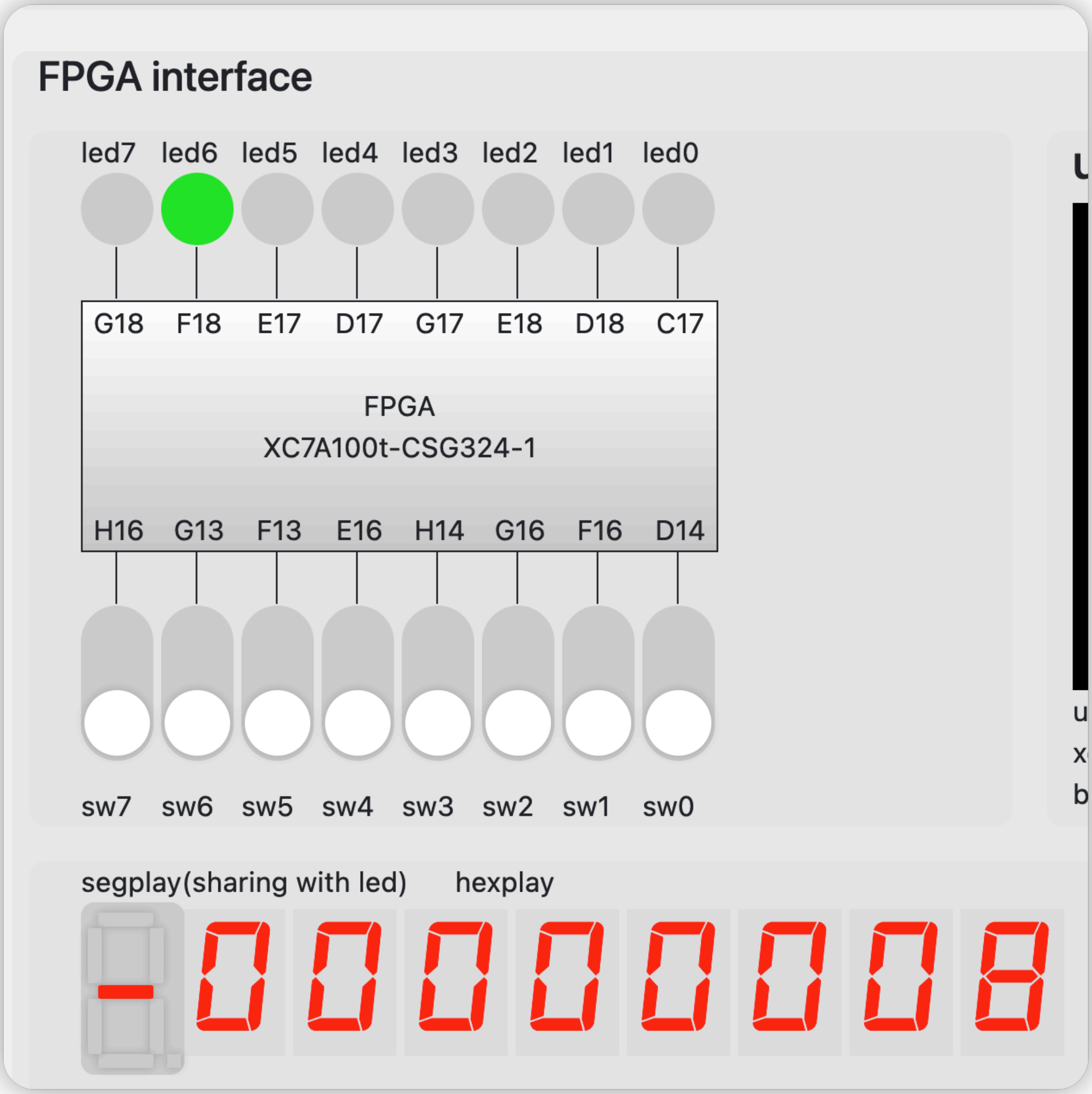
烧写到FPGAOL上 mem reg显示结果均与仿真结果一致。


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
segplay(sharing with led)
hexplay
00003008

pc为0x3008时 寄存器x6的值为1



计算fib第五项结果存到data_mem[0]处 结果为8


FPGA interface
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
segplay(sharing with led)
hexplay
00000008

测试fib_test.s（外设输入）

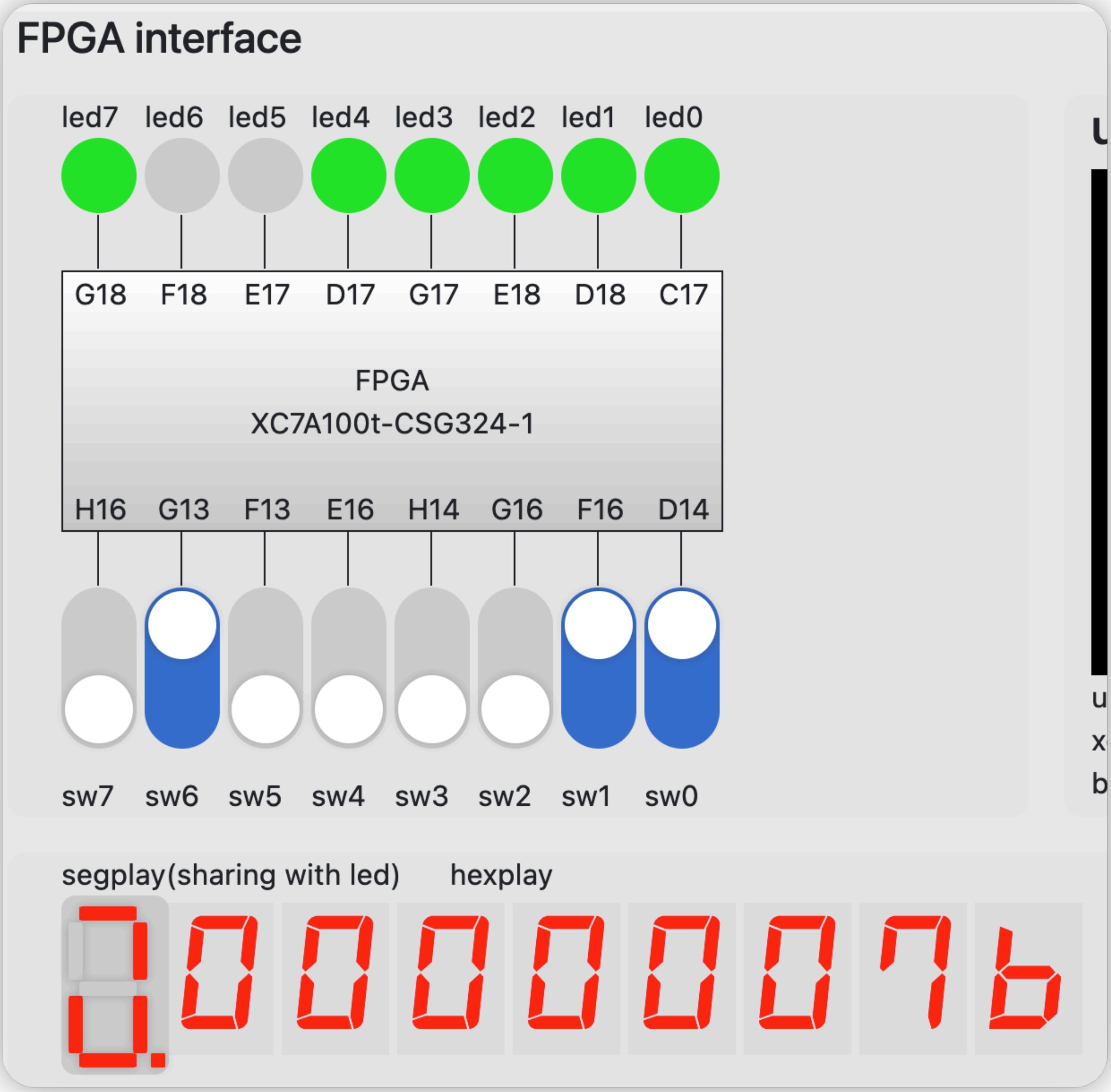
fib_test.asm

```
.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
```

fib_test.coe

```
memory_initialization_radix  = 16;
memory_initialization_vector =
00100093
00000333
40102223
41002283
fe028ee3
40c02403
40802423
00832023
00430313
40002223
41002283
fe128ee3
40102223
41002283
fe028ee3
40c02483
40902423
00932023
00430313
40002223
41002283
fe128ee3
009402b3
```

```
40502423
00532023
00430313
00900433
005004b3
40102223
41002283
fe028ee3
40002223
41002283
fe128ee3
fd1ff06f
```



结果正确。

心得体会

本实验由于工程量较大，在cpu数据通路编写和debug方面花费了较多时间。中间由于部分端口打错，仿真与烧写结果不一致，对比二者后发现在top模块端口名打错。但是vivado在generatebitstream过程中并未弹出错误提示，因此这点应该多加小心。

建议是实验文档可以详细说一下pdu的功能。