Lab 4 单周期 CPU 设计

姓名:傅申 学号: PB20000051 实验日期: 2022-4-20

1 实验题目

单周期 CPU 设计

2 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

3 实验平台

- Xilinx Vivado v2019.1
- Microsoft Visual Studio Code
- FPGAOL

4 实验过程

本次实验实现了一个单周期 RISC-V CPU, 并且将其与提供的 PDU 连接以实现 I/O 操作. CPU 可执行的指令有: `add`, `addi`, `sub`, `auipc`, `lw`, `sw`, `beq`, `blt`, `jal`, `jalr`.

整个项目层次结构如下:

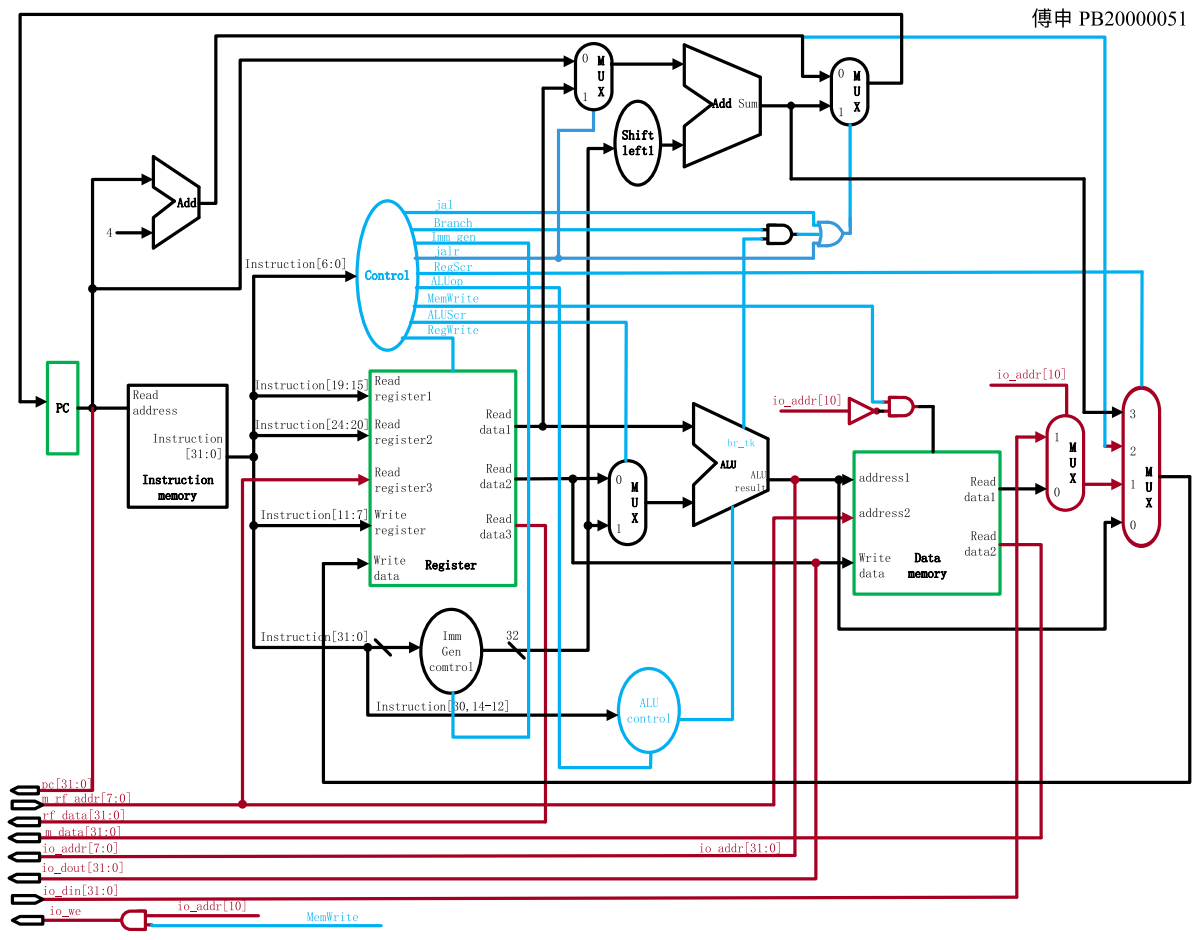
```
main (Main.v)
├─── PDU: pdu (PDU.v)
└─── CPU: cpu (CPU.v)
     ├─── Inst_Mem:         inst_mem         (inst_mem.xci)
     ├─── Data_Mem:         data_mem         (data_mem.xci)
     ├─── Control:          control          (Control.v)
     ├─── RegFile:          regfile          (Reg_File.v)
     ├─── Imm_Gen_Control:  imm_gen_control  (Imm_Gen_Control.v)
     ├─── ALU_Control:      alu_control      (ALU_Control.v)
     └─── ALU:              alu              (ALU.v)
```

4.1 CPU 数据通路

因为提供的数据通路只实现了 6 条指令, 所以要对其进行补充, 具体修改如下:

- Control 模块多了一个输出信号 `jalr`, 用于指示执行 `jalr` 指令.
- ALU 可以进行比较操作, `zero` 输出信号换为 `br_tk` 输出信号, 用于检查比较操作是否成立 (是否进行条件跳转). ALU 的输入信号 `funct` 用于选择相应操作.
- Add Sum 的第一个操作数 (上面的端口) 前添加一个 MUX, 选择信号为 `jalr` 信号, 由其决定选择 `rs1` 还是 `pc`.
- PC 前的 MUX 的选择信号逻辑变为 `PCSrc = jal | jalr | (Branch & BR_Taken)`, `BR_Taken` 为 ALU 输出的 `br_tk` 信号.
- 寄存器堆的写数据端口 Write Data 前的 MUX 增加一个输入信号接 Add Sum 模块的输出信号, 变为 4 选 1 数据选择器, 以实现 `auipc` 指令.

具体修改后的数据通路如下:



4.2 Control 模块

Control 模块输入指令的低 7 位 (opcode 段), 输出一系列控制信号如下:

输出信号	位数	含义
jal	1	当前指令是否为 jal
jalr	1	当前指令是否为 jalr
Branch	1	当前指令是否为条件跳转指令
Imm_Gen	3	立即数生成器的控制信号
RegSrc	2	寄存器堆写数据来源选择信号
ALUop	2	指示 ALU 要进行的操作
MemWrite	1	数据存储器写使能
ALUSrc	1	ALU 第二操作数来源选择信号
RegWrite	1	寄存器堆写使能

具体的 Verilog 代码如下

```
module control (
    input       [6:0] inst,
    output            jal,
    output            jalr,
    output            Branch,
    output reg  [2:0] Imm_Gen,
    output reg  [1:0] RegSrc,
    output      [1:0] ALUop,
    output            MemWrite,
    output            ALUSrc,
    output            RegWrite
);
    reg [5:0] type;
    always @(*) begin
        case (inst)
        7'b0110011: type = 6'b000001; // R-type
        7'b1100111,
        7'b0000011,
        7'b0010011: type = 6'b000010; // I-type
        7'b1100011: type = 6'b000100; // B-type
        7'b0100011: type = 6'b001000; // S-type
        7'b1101111: type = 6'b010000; // J-type
        7'b0010111: type = 6'b100000; // U-type
        default:    type = 6'b000000;
        endcase
    end

    assign jal      = type[4]; // J-type
    assign jalr     = (inst == 7'b1100111);
    assign Branch   = type[2]; // B-type
    assign MemWrite = type[3]; // S-type
    assign ALUSrc   = type[1] | type[3]; // I-type, S-type
    // R-type, I-type, J-type, U-type
    assign RegWrite = type[0] | type[1] | type[4] | type[5];
    assign ALUop    = {type[2] , (type[0] | inst == 7'b0010011)};

    always @(*) begin
        case (type)
        6'b000010: Imm_Gen = 3'b001; // I-type
        6'b000100: Imm_Gen = 3'b010; // B-type
```

```
        6'b001000: Imm_Gen = 3'b011; // S-type
        6'b010000: Imm_Gen = 3'b100; // J-type
        6'b100000: Imm_Gen = 3'b101; // U-type
        default:   Imm_Gen = 3'b000;
        endcase
    end

    always @(*) begin
        if (type[5])                // auipc
            RegSrc = 2'b11;
        else if (jalr || type[4])   // jal(r)
            RegSrc = 2'b10;
        else if (inst == 7'b0000011) // load
            RegSrc = 2'b01;
        else
            RegSrc = 2'b00;
    end
endmodule
```

4.3 ALU Control 和 ALU 模块

ALU Control 输入指令的 funct3, funct7 段和 Control 的 `ALUop` 信号, 生成并输出一个两位的 `funct` 信号以控制 ALU 模块的操作, 对应关系如下:

funct	ALU 操作
00	加法
01	减法
10	比较是否相等
11	比较是否小于

可以看出 `funct` 信号的高位用于区分比较操作和算术操作, 低位用于区分具体操作.

两个模块的 Verilog 代码如下

```
module alu_control(
    input               funct7,
    input       [2:0]   funct3,
    input       [1:0]   opcode,
    output reg  [1:0]   funct
);
    always @(*) begin
        if (opcode[1]) begin
            funct = {1'b1, funct3[2]};
        end else if (opcode[0]) begin
            funct = {1'b0, funct7};
        end else
            funct = 2'b00;
    end
endmodule
```

```
module alu(
    input       [31:0] op1,
    input       [31:0] op2,
    input       [1:0]  funct,
    output  reg [31:0] result,
    output  reg        br_tk
);
    always @(*) begin
        if (funct[1]) begin
            case (funct[0])
            1'b0: br_tk = (op1 == op2);
            1'b1: br_tk = (op1 <  op2);
            endcase
            result = 32'h0;
        end else begin
            case (funct[0])
            1'b0: result = op1 + op2;
            1'b1: result = op1 - op2;
            endcase
            br_tk = 1'b0;
        end
    end
endmodule
```

4.4 寄存器堆模块

寄存器堆为 32×32 位的寄存器堆, 且 0 号寄存器始终为 0. 具体的 Verilog 代码如下

```
module regfile #(
    parameter WIDTH = 32
) (
    input               clk,
    input               we,
    input   [4:0]       ra1,
    input   [4:0]       ra2,
    input   [4:0]       ra3,
    input   [4:0]       wa,
    input   [WIDTH-1:0] wd,
    output  [WIDTH-1:0] rd1,
    output  [WIDTH-1:0] rd2,
    output  [WIDTH-1:0] rd3
);
    reg [WIDTH-1:0] rf [31:0];
    assign rd1 = (ra1 == 5'h0) ? 0 : rf[ra1];
    assign rd2 = (ra2 == 5'h0) ? 0 : rf[ra2];
    assign rd3 = (ra3 == 5'h0) ? 0 : rf[ra3];

    always @(posedge clk) begin
        if (we) rf[wa] <= wd;
    end
endmodule
```

4.5 立即数生成模块

立即数生成模块接收 `Imm_Gen` 信号, 取指令相应的字段生成对应立即数, 具体的 Verilog 代码如下

```
module imm_gen_control(
    input       [31:0]  inst,
    input       [2:0]   imm_gen,
    output reg  [31:0]  imm
);
    always @(*) begin
        case (imm_gen)
        3'b001:  // I-type
            imm = {{21{inst[31]}}, inst[30:20]};
        3'b010:  // B-type
            imm = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};
        3'b011:  // S-type
            imm = {{21{inst[31]}}, inst[30:25], inst[11:7]};
        3'b100:  // J-type
            imm = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};
        3'b101:  // U-type
            imm = {inst[31:12], 12'h0000};
        default: imm = 32'h0;
        endcase
    end
endmodule
```

4.6 存储器

- 指令存储器为 256 × 32 位的 ROM (分布式), 因为存储器一个地址存储一个字, 所以输入 `a` 端口接 `pc[9:2]` 而不是 `pc[7:0]`.
- 数据存储器为 256 × 32 位的双端口 RAM (分布式), 输入 `a` 端口接 ALU 输出的 `[9:2]` 位, 原因同上.

4.7 CPU 连线

按照数据通路给 CPU 连线, 具体的 Verilog 代码如下

```
module cpu(
    input               clk,
    input               rst,
    // IO_BUS
    input       [31:0]  io_din,  //来自 sw 的输入数据
    output      [7:0]   io_addr,
    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

    //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 reg  [31:0]  pc         //PC的内容
);
    wire [31:0] new_pc;
```

```
wire [31:0] next_pc;
wire [31:0] added_pc;
wire [1:0]  PCSrc;
wire [31:0] inst;
wire [2:0]  Imm_Gen;
wire [1:0]  RegSrc;
wire [1:0]  ALUop;
wire        jal;
wire        jalr;
wire        Branch;
wire        MemWrite;
wire        ALUSrc;
wire        RegWrite;
wire [31:0] reg1;
wire [31:0] reg2;
reg  [31:0] regw;
wire [31:0] imm;
wire [31:0] op1;
wire [31:0] op2;
wire [1:0]  ALU_funct;
wire [31:0] ALU_out;
wire        BR_Taken;
wire        mem_we;
wire [31:0] mem_out;
wire [31:0] load_data;

assign new_pc    = (PCSrc) ? added_pc : next_pc;
assign next_pc   = pc + 32'h4;
assign added_pc  = ((jalr) ? reg1 : pc) + (imm & 32'hFFFFFFFE);
assign PCSrc     = jal | jalr | (Branch & BR_Taken);
assign op1       = reg1;
assign op2       = (ALUSrc) ? imm : reg2;
assign io_dout   = reg2;
assign io_addr   = ALU_out[7:0];
assign io_we     = ALU_out[10] & MemWrite;
assign mem_we    = (~ALU_out[10]) & MemWrite;
assign load_data = (ALU_out[10]) ? io_din : mem_out;

always @(posedge clk or posedge rst) begin
    if (rst) pc <= 32'h0000_3000;
    else pc <= new_pc;
end

inst_mem Inst_Mem (
    .a   (pc[9:2]),
    .spo (inst)
);

control Control (
    .inst     (inst[6:0]),
    .Imm_Gen  (Imm_Gen),
    .RegSrc   (RegSrc),
    .ALUop    (ALUop),
    .jal      (jal),
    .jalr     (jalr),
    .Branch   (Branch),
    .MemWrite (MemWrite),
    .ALUSrc   (ALUSrc),
    .RegWrite (RegWrite)
);
```

```
    regfile #(.WIDTH(32)) RegFile (
        .clk (clk),
        .we  (RegWrite),
        .ra1 (inst[19:15]),
        .ra2 (inst[24:20]),
        .ra3 (m_rf_addr[4:0]),
        .wa  (inst[11:7]),
        .wd  (regw),
        .rd1 (reg1),
        .rd2 (reg2),
        .rd3 (rf_data)
    );

    imm_gen_control Imm_Gen_Control (
        .inst    (inst),
        .imm_gen (Imm_Gen),
        .imm     (imm)
    );

    alu_control ALU_Control (
        .funct7 (inst[30]),
        .funct3 (inst[14:12]),
        .opcode (ALUop),
        .funct  (ALU_funct)
    );

    alu ALU (
        .op1    (op1),
        .op2    (op2),
        .funct  (ALU_funct),
        .result (ALU_out),
        .br_tk  (BR_Taken)
    );

    data_mem Data_Mem (
        .clk  (clk),
        .we   (mem_we),
        .a    (ALU_out[9:2]),
        .dpra (m_rf_addr),
        .d    (reg2),
        .spo  (mem_out),
        .dpo  (m_data)
    );

    always @(*) begin
        case (RegSrc)
        2'b00: regw = ALU_out;
        2'b01: regw = load_data;
        2'b10: regw = next_pc;
        2'b11: regw = added_pc;
        endcase
    end

endmodule
```

4.8 顶层模块 Main

顶层模块将 CPU 与 PDU 相连, 具体的 Verilog 代码如下

```
module main(
    input           clk,
    input           btn,
    input   [7:0]   sw,
    output  [7:0]   led,
    output  [2:0]   an,
    output  [3:0]   d
);
    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;

    pdu PDU(
        .clk        (clk),
        .rst        (sw[7]),
        .run        (sw[6]),
        .step       (btn),
        .clk_cpu    (clk_cpu),
        .valid      (sw[5]),
        .in         (sw[4:0]),
        .check      (led[6:5]),
        .out0       (led[4:0]),
        .an         (an),
        .seg        (d),
        .ready      (led[7]),
        .io_addr    (io_addr),
        .io_dout    (io_dout),
        .io_we      (io_we),
        .io_din     (io_din),
        .m_rf_addr  (m_rf_addr),
        .rf_data    (rf_data),
        .m_data     (m_data),
        .pc         (pc)
    );

    cpu CPU(
        .clk        (clk_cpu),
        .rst        (sw[7]),
        .io_din     (io_din),
        .io_addr    (io_addr),
        .io_dout    (io_dout),
        .io_we      (io_we),
        .m_rf_addr  (m_rf_addr),
        .rf_data    (rf_data),
        .m_data     (m_data),
        .pc         (pc)
    );
endmodule
```

5 实验结果

5.1 指令测试

使用如下的 RISC-V 汇编进行测试

```
.text
start:
    # test for auipc, addi, jal
    auipc   t0, 0x12342     # t0 <- 0x12345000
    addi    x1, x0, 1       # x1 <- 1
    sw      t0, 0x408(x0)   # echo t0
    jal     x2, next_step   # wait for io

    # test for add, sub
    addi    t1, x0, 0x24    # t1 <- 0x24
    addi    t2, x0, 0x13    # t2 <- 0x13
    add     t3, t1, t2      # t3 <- 0x37
    sw      t3, 0x408(x0)   # echo t3
    jal     x2, next_step   # wait for io
    sub     t3, t1, t2      # t3 <- 0x11
    sw      t3, 0x408(x0)   # echo t3
    jal     x2, next_step   # wait for io

    # test for lw, sw
    sw      t0, 0x000(x0)   # mem[0] <- 0x12345000
    jal     x2, next_step   # wait for io
    lw      t3, 0x000(x0)   # t3 <- mem[0]
    sw      t3, 0x408(x0)   # echo t3
    jal     x2, next_step   # wait for io
    jal     x0, start

# test for blt, beq, jalr
next_step:
    sw      x0, 0x404(x0)   # set ready bit
poll_up:
    lw      t3, 0x410(x0)
    blt     t3, x1, poll_up
poll_down:
    lw      t3, 0x410(x0)
    beq     t3, x1, poll_down
    sw      x1, 0x404(x0)   # reset ready bit
    jalr    x0, 0(x2)
```

它采用等待 I/O 的方式, 通过按动 valid 按钮一步步运行, 对下面的仿真文件

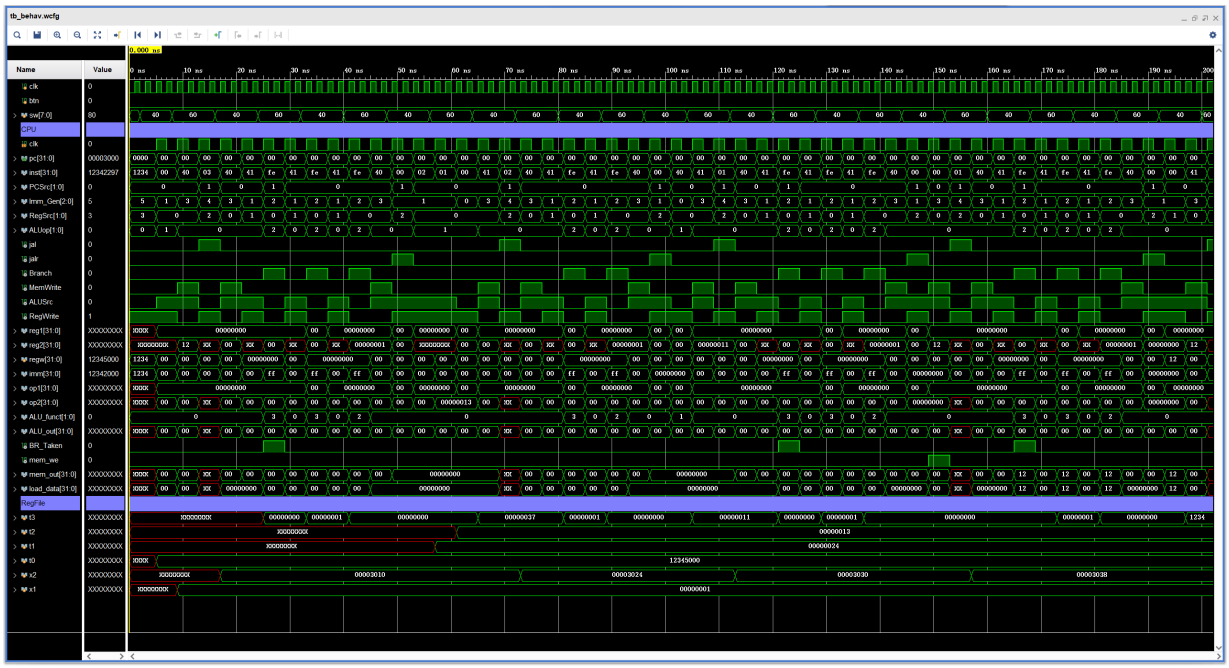
```
module tb();
    reg         clk;
    reg         btn;
    reg   [7:0] sw;
    wire  [7:0] led;
    wire  [2:0] an;
    wire  [3:0] d;

    main test(
```

```
        .clk(clk),
        .btn(btn),
        .sw(sw),
        .led(led),
        .an(an),
        .d(d)
    );

    initial begin
        clk = 0;
        btn = 0;
        sw  = 8'b10000000;
        #2 sw[7:6] = 2'b01;
        #200 $finish;
    end
    always #1 clk = ~clk;
    always #8 sw[5] = ~sw[5];
endmodule
```

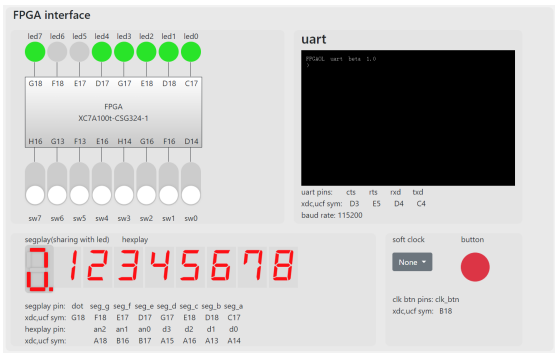
仿真结果如下

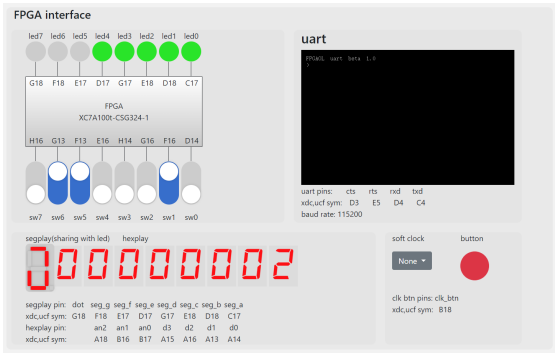


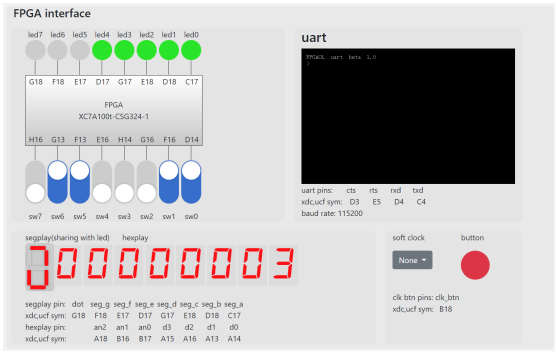
观察各个寄存器的值可知, 几个部分的指令都正确运行了. 生成比特流后上板, 可以看到逐步运行的结果如下, 其中测试完 `lw` 指令后数码管显示没有变化, 故没有展示:

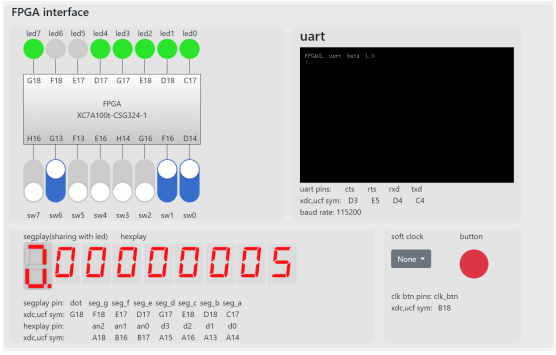
5.2 应用程序测试

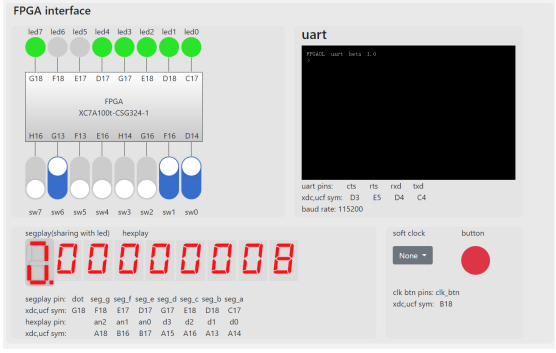
使用 `fib_from_io.s` 生成的 coe 文件例化指令存储器, 它也使用了等待 I/O 的方式, 生成比特流后上板, 可以看到逐步运行的结果如下:

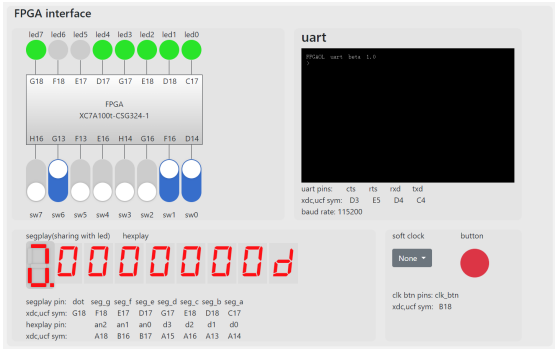












可以看出程序正常执行

6 心得体会

本次实验难度中等, 对于理解单周期 CPU 有很大的帮助.