Lab 6 Report

PB22020514 郭东昊 2024.1.4

一、 实验目的与内容

实验内容

1. 设计单周期和多周期龙芯 LA32 CPU，执行以下指令（“龙芯杯”C1 级）：
   add.w, lu12i.w, addi.w, ld.w, st.w, bne
2. 完成电路的仿真、分析、下载测试及 SDU 调试

实验目的

1. 理解单周期和多周期 CPU 的结构和工作原理
2. 掌握单周期和多周期 CPU 的设计和调试方法
3. 熟练掌握数据通路和控制器的设计和 Verilog 描述方法
4. 熟练掌握查看生成电路及其性能和资源使用情况

二、 逻辑设计

单周期 CPU

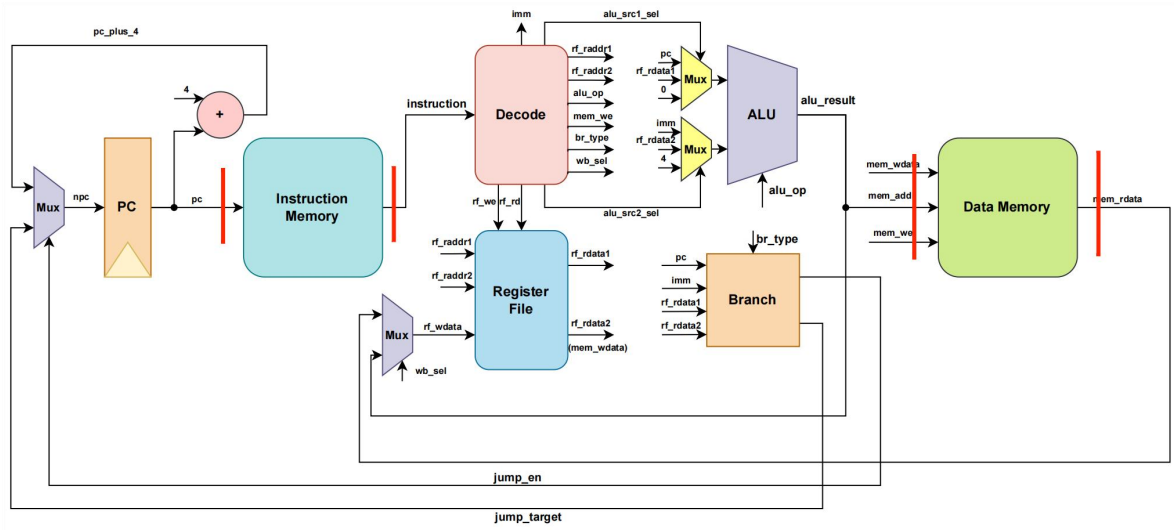

图 1 单周期 CPU 数据通路 by 马老师和我的红线

设计的单周期 CPU 数据通路基本遵照给出的示意图。但和该图有几处不同：

1. ALU 两个操作数的两个三选一选择器变成二选一选择器。由于只要完成 C1 指令集，两个选择器无需做到三选一。具体而言，略去了 alu_src1 的选择器前的“pc”及 alu_src2 的选择器前的“4”。这二者应该都用于某些跳转指令。

2. alu_op 悬置。由于待实现的指令只要实现加法（bne 指令涉及的判断不等运算在 branch 模块中实现），ALU 暂置为加法器即可。

3. br_type 悬置。因为待实现的分支指令只有一种。

4. 如图中红线所示，根据冯诺依曼自动机理论以及 CPU 在计算机中的实际结构，将指令存储器、数据存储器与 CPU 主要部分划分开。这样做并没有什么卵用，还给我后期带来了不少麻烦，但代码是这样写的，图 4 体现了这样的差别。

5. 代码中部分变量名与图中不同，如下表：

图中	代码
jump_en	br_en
jump_target	br_target
instruction	inst

表 1

多周期 CPU

多周期 CPU，不是我设想的有一个控制单元监督每一个过程的执行，而是这个控制单元（即有限状态机）提前根据指令的不同，给每个指令的执行预设了相应的不同的周期数，从而给执行过程数不一的不同指令框定了不同的执行时间，仅此而已。写得不好的多周期 CPU 的执行速度未必比单周期 CPU 快，最多只是提高主频而已。

多周期 CPU 相对单周期 CPU，要注意的唯有对 PC 自增的处理。其余增加的寄存器，真正重要的只有 IR，其余寄存器的作用只是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟，但这个说法也有问题，其实，单纯实现多周期 CPU 本身根本就没有中间寄存器的添加必要，只要 IR 寄存器内的值不变，数据通路上的中间值在计算稳定后就不会波动，并不需要加入时序寄存器来保存。

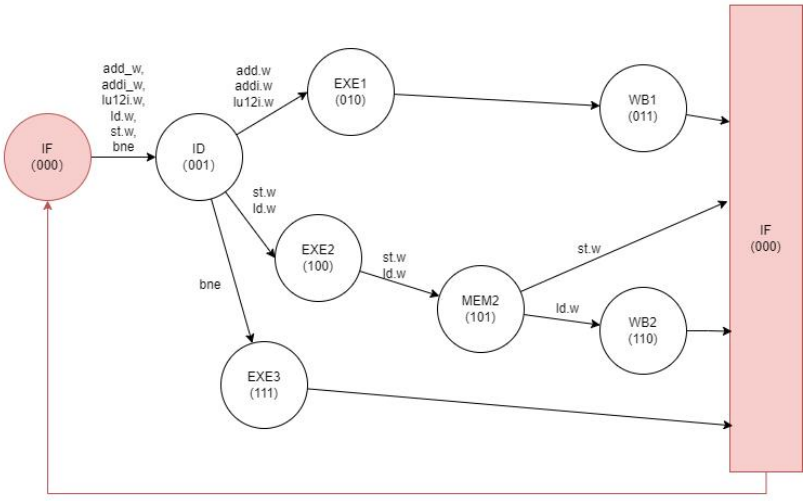

图 2 多周期 CPU 状态图

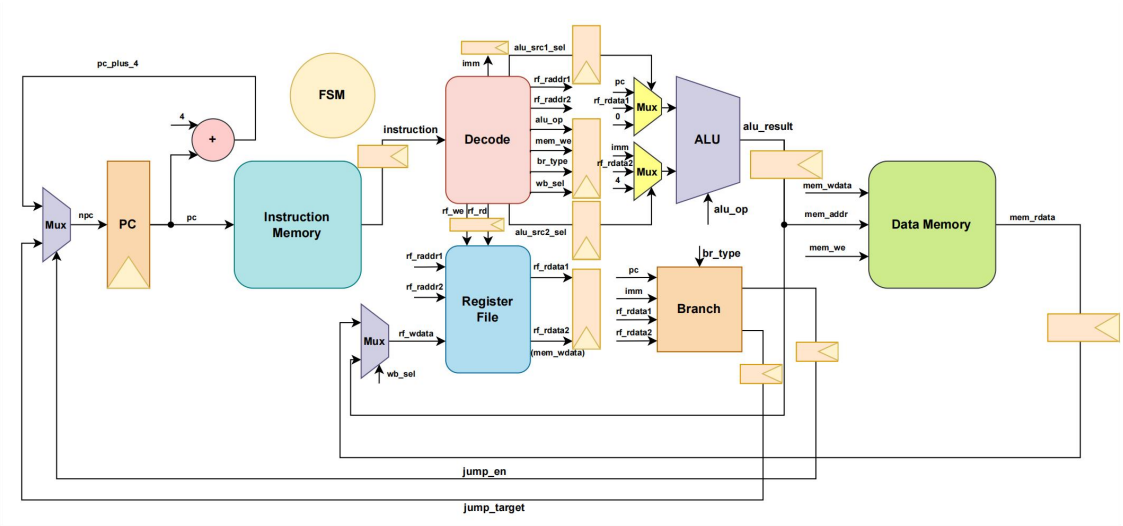

图 3 多周期 CPU 数据通路 by 马老师

CPU 的指令存储器及数据存储器使用 IP 核实现，这样做的坏处是不便将读端口扩展至 3 个用于 debug，所以寄存器堆使用手动实现，以便增加读端口至 3 个。

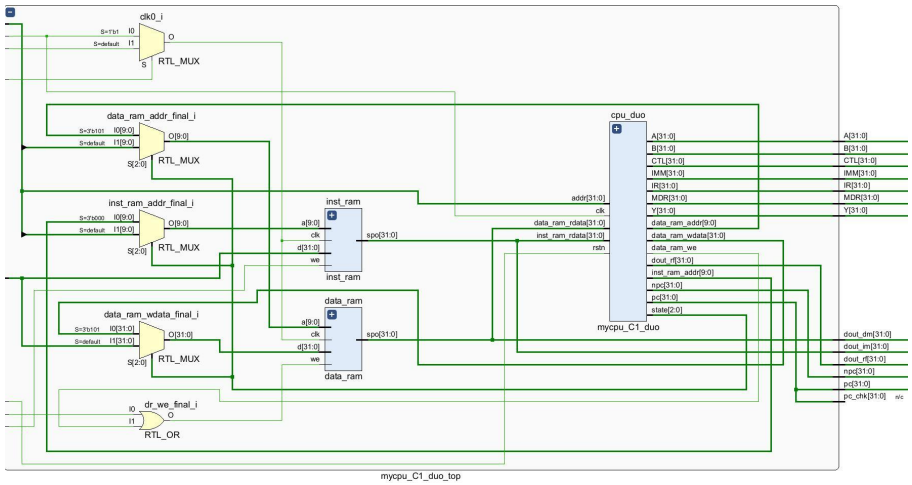

图 4 多周期 CPU 内部线路

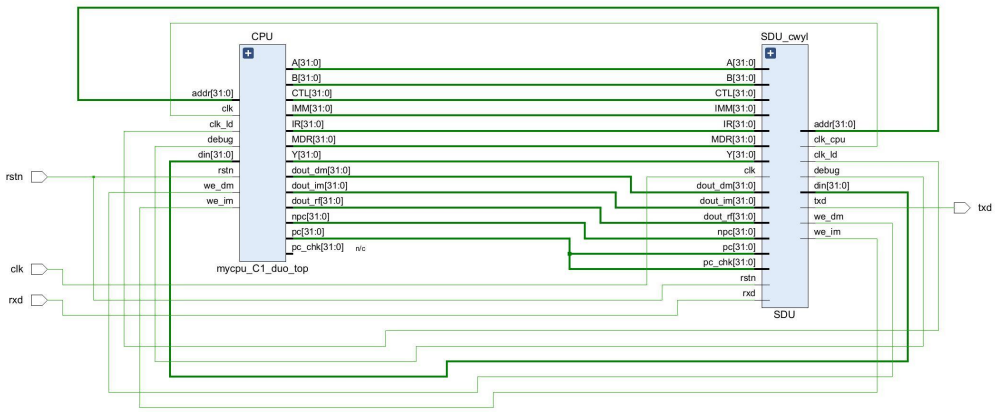

图 5 多周期 CPU 与 SDU 的接线

三、 核心代码

单周期 CPU

```
module mycpu_C1 (
    input           clk,
    input           rstn,

    output          inst_ram_we,
    output  [9:0]   inst_ram_addr,
    output  [31:0]  inst_ram_wdata,
    input   [31:0]  inst_ram_rdata,

    output          data_ram_we,
    output  [9:0]   data_ram_addr,
    output  [31:0]  data_ram_wdata,
    input   [31:0]  data_ram_rdata
```

图 6 单周期 CPU 接口

取指和译码采用的思路如下，分为四步。我认为是可读性非常高的，书写也非常简单：

```
//指令的截取
assign op_31_26 = inst[31:26];
assign op_25_22 = inst[25:22];
assign op_21_20 = inst[21:20];
assign op_19_15 = inst[19:15];
assign rk       = inst[14:10];
assign rj       = inst[9:5];
assign rd       = inst[4:0];
assign si_12    = inst[21:10];
assign si_20    = inst[24:5];
assign offs_16  = inst[25:10];
```

图 7 Step1 指令截取

```
//指令分段译码
decoder_6_64 u_dec0(.in(op_31_26),  .co(op_31_26_d));
decoder_4_16 u_dec1(.in(op_25_22),  .co(op_25_22_d));
decoder_2_4  u_dec2(.in(op_21_20),  .co(op_21_20_d));
decoder_5_32 u_dec3(.in(op_19_15),  .co(op_19_15_d));
```

图 8 Step2 指令分段译码

```
//指令信号的生成
assign inst_add_w   = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h00];
assign inst_addi_w  = op_31_26_d[6'h00] & op_25_22_d[4'ha];
assign inst_ld_w    = op_31_26_d[6'h0a] & op_25_22_d[4'h2];
assign inst_st_w    = op_31_26_d[6'h0a] & op_25_22_d[4'h6];
assign inst_bne     = op_31_26_d[6'h17];
assign inst_lu12i_w = op_31_26_d[6'h5];
```

图 9 Step3 指令信号的生成

```
//控制信号的生成
assign alu_src1     = inst_lu12i_w ? 32'h00000000 : rf_rdata1;
assign alu_src2     = (inst_lu12i_w | inst_addi_w | inst_st_w | inst_ld_w) ? imm : rf_rdata2;
assign alu_op       = inst_add_w;//ALU操作数的生成逻辑，先暂时设置为加法，暂时用不上
assign data_ram_we = inst_st_w;
assign rf_we        = inst_add_w | inst_addi_w | inst_ld_w;
assign rf_raddr2_sel= inst_st_w | inst_bne;//值为1时，另一个寄存器为rd，否则为rk
assign imm_sel      = inst_lu12i_w;//值为1时，选择{si_20,12{0}}
assign wb_sel       = inst_ld_w;//值为1时，写回DR中的值；值为0时，写回ALU_result
assign br_type      = inst_bne;//br_type生成逻辑，暂时等于inst_bne，暂时用不上
```

图 10 Step4 控制信号的生成

根据龙芯架构 32 位精简版手册的 55 页 6.3 节有关复位的要求，PC 复位值是 0x1c000000，虽然我们的实验中取指地址用不到这么多的地址位，但是有些计算（例如 pcaddu12i 指令）需要使用完整的 pc 值，所以一定把 pc 要复位到这个值。

```
always@(posedge clk) begin
    if(!rstn)begin
        pc <= 32'h1c000000;
    end
    else begin
        pc <= nextpc;
    end
end
```

图 11 PC

以下是 br 和 imm 的相关逻辑：

```
assign br_offs      = {14'b0, offs_16, 2'b0};
assign br_target    = pc + br_offs;
assign rj_eq_rd     = (rf_rdata1 == rf_rdata2);
assign br_en        = inst_bne && !rj_eq_rd;
assign nextpc       = br_en ? br_target : (pc + 4);
```

```
assign si_12_to_32  = {{20{si_12[11]}},si_12[11:0]};
assign si_20_to_32  = {si_20[19:0],12'b0};
assign imm          = imm_sel ? si_20_to_32 : si_12_to_32;
```

图 12 br 和 imm 的逻辑

其他按需连线即可，总共一百多行代码即可实现。

多周期 CPU

根据前文列出的状态图，相对于单周期 CPU，主要增加了状态机和指令寄存器 IR。增加的代码如下：

```
/*******多周期CPU状态机的状态定义*******/
//reg     [2:0]   state;
reg     [2:0]   next_state;
parameter IF    = 3'b000;
parameter ID    = 3'b001;
parameter EXE1  = 3'b010;
parameter WB1   = 3'b011;
parameter EXE2  = 3'b100;
parameter MEM2  = 3'b101;
parameter WB2   = 3'b110;
parameter EXE3  = 3'b111;
```

图 13 状态机的状态定义

状态机的状态转移逻辑略，根据状态图填 case 和 if 即可。
添加大量寄存器，并在相应位置用对应变量的寄存器值替换原变量

```
/*************多周期CPU添加的各寄存器************/
//IF后的寄存器
//reg     [31:0]      IR;//inst_reg
//ID后的寄存器
reg     [31:0]      rf_rdata1_reg;
reg     [31:0]      rf_rdata2_reg;
reg     [31:0]      imm_reg;
reg                 alu_src1_sel_reg;
reg                 alu_src2_sel_reg;
reg                 alu_op_reg;
reg                 data_ram_we_reg;
reg                 rf_we_reg;
reg     [31:0]      rd_reg;
reg                 wb_sel_reg;
reg                 br_type_reg;
//EXE后的寄存器
reg     [31:0]      alu_result_reg;
reg                 br_en_reg;
reg                 br_target_reg;
//MEM后的寄存器
reg     [31:0]      data_ram_rdata_reg;
//新增PC_we和IR_we
reg                 pc_we;
reg                 IR_we;
```

```
//指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好
//也就是说，除PC外，各寄存器都在时钟下降沿写入，这样能保证下一周期有写入操作时，能够读到上一周期的结果（可能是数据，也可能是控制信号）
always@(negedge clk)begin
    rf_rdata1_reg       <= rf_rdata1;
    rf_rdata2_reg       <= rf_rdata2;
    imm_reg             <= imm;
    alu_src1_sel_reg    <= alu_src1_sel;
    alu_src2_sel_reg    <= alu_src2_sel;
    alu_op_reg          <= alu_op;
```

图 14 添加的各寄存器

为保证 PC 每个指令只更新一次，增加控制信号 pc_we。为保证 IR 指令稳定，添加控制信号 IR_we（其实并没有那么必要，PC 稳定 IR 自然稳定）

```
//PC_we的生成逻辑
always @(negedge clk) begin
    if(next_state == IF)begin
        pc_we <= 1'b1;
    end
    else begin
        pc_we <= 1'b0;
    end
end
```

```
//IR_we的生成逻辑
always@(*)begin
    case(state)
        IF: begin
            IR_we = 1'b1;
        end
        default: begin
            IR_we = 1'b0;
        end
    endcase
end
```

图 15 PC_we 和 IR_we 的生成逻辑

相应改变 PC 和 IR 的更新逻辑，不再列出代码。

根据 SDU 对多周期 CPU 进行改写

SDU 需要访问数据存储器 DM，指令存储器 IM，寄存器堆 RF，CPU 也要读 RF, IM, DM；CPU 要写 RF, DM, SDU 也要写 IM, DM。这是需要解决的冲突，最简单的方法是在三个存储器上多加第三个 debug 读端口，再加一个 debug 输出端口。

然而，DM 和 IM 均采用 IP 核例化，无法在已有两个读端口的基础上再加第三个 debug 读端口，所以采取了如下策略：对于读操作，如果当前状态（由多周期 CPU 添加的状态机给出）下 CPU 不需访问 DM/IM，将传入 DM/IM 的读地址 data_ram_addr_final/inst_ram_addr_final 取 SDU 传入的 addr，否则采用原值；对于写操作，CPU 对 IM 无写权限，IM 的写使能和写数据由 SDU 全权控制，而由于 CPU 对 DM 有写操作，对于 DM 的写使能，也采用上述思路作选择。代码实现如下：

```
assign inst_ram_we = we_im;         //IM只由SDU写
assign inst_ram_addr_final = (state == 3'b000) ? inst_ram_addr : addr[9:0];//只有在状态为IF（000）时，才会从CPU读取地址
assign inst_ram_wdata = din;
assign dout_im = inst_ram_rdata;

assign dr_we_final = we_dm | data_ram_we;   //数据存储器的写使能由CPU和SDU共同决定
assign data_ram_addr_final = (state == 3'b101) ? data_ram_addr : addr[9:0]; //只有在状态为MEM（101）时，才会从CPU读取地址
assign data_ram_wdata_final = (state == 3'b101) ? data_ram_wdata : din;     //只有在状态为MEM（101）时，才会从CPU读取写数据
assign dout_dm = data_ram_rdata;
```

图 16 CPU 连接 SDU 时对 DM 和 IM 的冲突处理

对于 RF，寄存器堆使用了手动实现，以便增加读端口至 3 个。

其他改写根据 SDU 的引脚要求作相应的代码改动即可。

四、 电路设计与分析

RTL

CPU 综合电路的 RTL 电路图如下，与 SDU 文档默认的通路相一致：

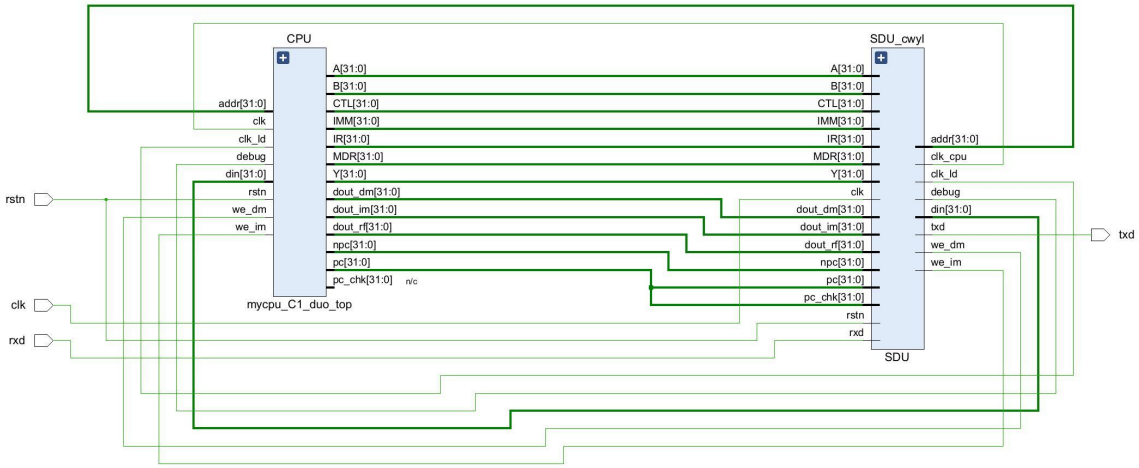

图 17 CPU 综合电路的 RTL 电路图

CPU 模块的 RTL 电路图如下，内含复杂的数据通路、控制通路与各模块

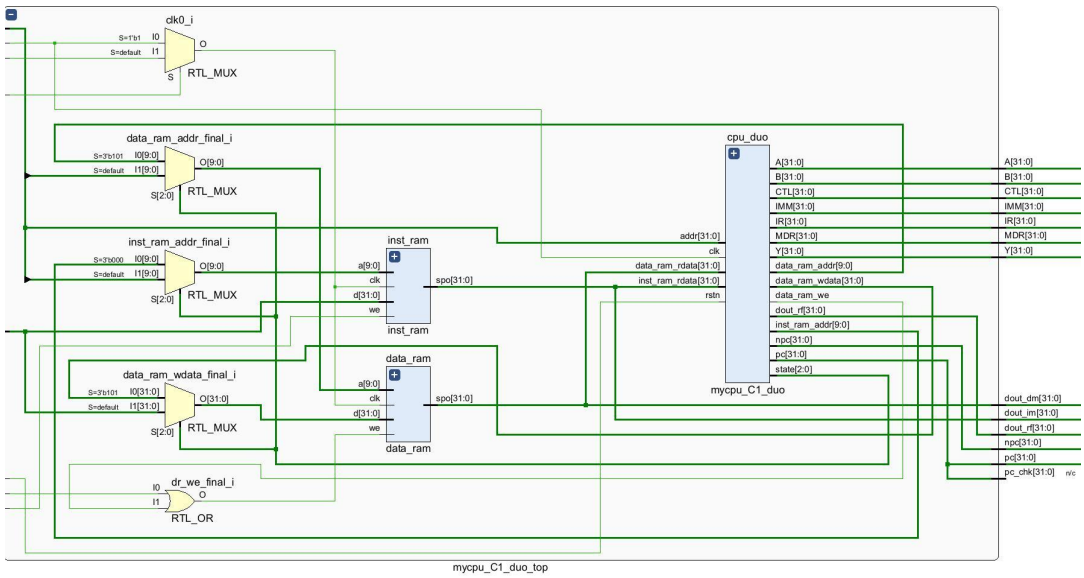

图 18 CPU 模块的 RTL 电路图

资源使用情况

电路的资源使用情况及 WNS 如下图：

WNS	TNS	WHS	THS
7.592	0.000	0.219	0.000

图 19 WNS

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
SDU_top	3518	1784	705	294	4	3
CPU (mycpu_C1_duo_top)	2220	1200	681	288	0	0
SDU_cwyl (SDU)	1298	584	24	6	0	0

图 20 资源使用情况表

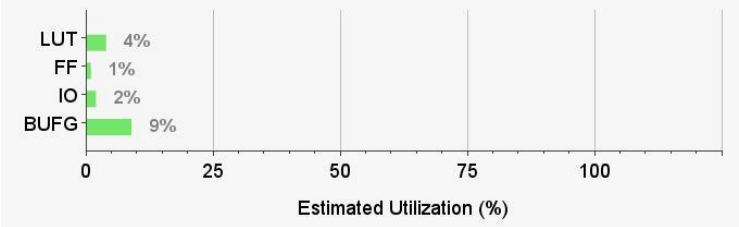

图 21 资源使用情况图表

WNS 非负，电路可以正常运行。

五、 上板结果与分析

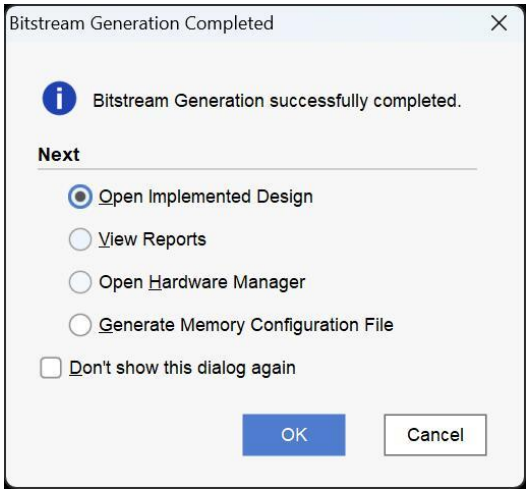

图 22 烧录成功

比特流文件烧录成功，由于时间原因，很遗憾，未上板调试。

六、 总结

本次实验，我设计了 C1 级的单周期和多周期龙芯 LA32 CPU，并根据 SDU 将多周期 CPU 进行改写，解决了一些冲突，最后将改写后的多周期 CPU 和 SDU 进行了连接，完成了上板前的全部流程。我并未完成上板 debug 的工作，希望学期内未完成的工作能在寒假完成。

通过本次实验，我理解了单周期和多周期 CPU 的结构和工作原理；掌握了单周期和多周期 CPU 的设计方法，但是没有掌握调试方法；掌握了数据通路和控制器的设计和 Verilog 描述方法。