实验 4：完整单周期 CPU

张子康 PB22020660

2024 年 04 月 22 日

1 实验目的与内容

1.1 实验目的

在本次实验中，我们将进一步完善上一次实验设计的 CPU，为其增加分支指令和访存指令的相关功能。

1.2 实验内容

1.2.1 任务 1：访存控制单元设计

设计访存控制单元 SL_UNIT，以正确处理访存指令。

1.2.2 任务 2：搭建 CPU

正确实现 CPU 的各个功能模块，并根据数据通路将其正确连接。理论上，你只需要完成 CPU 模块及其子模块的设计，而无需修改其他模块的内容。最终，你需要在 FPGAOL 上上板运行，并通过我们给出的测试程序。

1.2.3 任务 3：斐波那契数列

将 Lab1 中编写的斐波那契数列程序（普通版本、大整数版本均可）导出为 COE 文件，在自己设计的 CPU 上运行。相关数据的输入、输出方式不限。

2 逻辑设计

2.1 任务 1：访存控制单元设计

访存单元代码如下：

```
1 //定义访存类型
2 `define LW 4'b0000
3 `define LH 4'b0001
4 `define LB 4'b0010
5 `define LHU 4'b0011
```

```
`define LBU 4'b0100
`define SW 4'b0101
`define SH 4'b0110
`define SB 4'b0111
module SLU (input [31 : 0] addr,
            input [3 : 0] dmem_access,
            input [31 : 0] rd_in,
            input [31 : 0] wd_in,
            output reg [31 : 0] rd_out,
            output reg [31 : 0] wd_out);
// addr_用于存储访存的地址
wire [31:0] addr_;

// 初始化
initial begin
    rd_out=0;
    wd_out=0;
end

// 将计算出的访存地址存储在addr_中
assign addr_=addr-32'h10010000;

// 根据输入的访存类型(dmem_access)来对DATA_MEM进行访问
always @(*) begin
    case(dmem_access)
        // 读字
        `LW:
        begin
            rd_out = rd_in;
            wd_out =0;
        end
        // 读半字，不考虑跨字读取，进行符号扩展
        `LH:begin
```

```
39              wd_out =0;
40              // 根据输入的addr_判断要读取的半字的位置
41              // 并进行符号扩展
42              if(addr_[1:0] == 0)
43                  rd_out = {{16{rd_in[15]}},rd_in[15:0]};
44              else
45                  rd_out = {{16{rd_in[31]}},rd_in[31:16]};
46          end
47          // 读字节，进行符号扩展
48          `LB:begin
49              // 根据输入的addr_判断要读取的字节的位置
50              // 并进行符号扩展
51              case(addr_[1:0])
52                  0:rd_out = {{24{rd_in[7]}},rd_in[7:0]};
53                  1:rd_out = {{24{rd_in[15]}},
54                              rd_in[15:8]};
55                  2:rd_out = {{24{rd_in[23]}},
56                              rd_in[23:16]};
57                  3:rd_out = {{24{rd_in[31]}},rd_in[31:24]};
58              endcase
59              wd_out =0;
60          end
61          // 读半字，不考虑跨字读取，进行无符号扩展
62          `LHU:begin
63              // 根据输入的addr_判断要读取的半字的位置
64              // 并进行无符号扩展
65              if(addr_[1:0] == 0)
66                  rd_out = {16'b0,rd_in[15:0]};
67              else
68                  rd_out = {16'b0,rd_in[31:16]};
69              wd_out =0;
70          end
71          // 读字节，进行无符号扩展
```

```
72          `LBU:begin
73              // 根据输入的addr_判断要读取的字节的位置
74              // 并进行无符号扩展
75              case(addr_[1:0])
76                  0:rd_out = {24'b0,rd_in[7:0]};
77                  1:rd_out = {24'b0,rd_in[15:8]};
78                  2:rd_out = {24'b0,rd_in[23:16]};
79                  3:rd_out = {24'b0,rd_in[31:24]};
80              endcase
81              wd_out =0;
82          end
83          // 写字
84          `SW:begin
85              wd_out=wd_in;
86              rd_out = 0;
87          end
88          // 写半字
89          `SH:begin
90              // 将要写入的部分与读取出的字的部分进行拼接
91              if(addr_[1:0] == 0)
92                  wd_out = {rd_in[31:16],wd_in[15:0]};
93              else
94                  wd_out = {wd_in[15:0],rd_in[15:0]};
95              rd_out = 0;
96          end
97          // 写字节
98          `SB:begin
99              // 将要写入的部分与读取出的字的部分进行拼接
100             case(addr_[1:0])
101                 0:wd_out = {rd_in[31:8],wd_in[7:0]};
102                 1:wd_out = {rd_in[31:16],wd_in[7:0],
103                             rd_in[7:0]};
104                 2:wd_out = {rd_in[31:24],wd_in[7:0],
```

```
                                rd_in[15:0]};
                    3:wd_out = {wd_in[7:0],rd_in[23:0]};
                endcase
                rd_out = 0;
            end
            // 其他情况
            default:
                begin
                    rd_out = 0;
                    wd_out =0;
                end
        endcase
end
endmodule
```

2.2 任务 2：搭建 CPU

Branch 模块代码如下：

```
// 定义分支跳转指令的代码
`define BEQ 4'b0000
`define BNE 4'b0001
`define BLT 4'b0010
`define BGE 4'b0011
`define BLTU 4'b0100
`define BGEU 4'b0101
`define JALR 4'b0110
`define JAL 4'b0111
`define ADD4 4'b1111
module BRANCH(input [3 : 0] br_type,
              input [31 : 0] br_src0,
              input [31 : 0] br_src1,
              output reg [1 : 0] npc_sel);
wire signed [31:0] src0;
```

```
wire signed [31:0] src1;
assign src0=br_src0;
assign src1=br_src1;
// 根据输入的代码，选择next_pc
always @(*) begin
    case (br_type)
        `BEQ:npc_sel=br_src0==br_src1;
        `BNE:npc_sel=br_src0!=br_src1;
        `BLT:npc_sel=src0<src1;
        `BGE:npc_sel=src0>=src1;
        `BLTU:npc_sel=br_src0<br_src1;
        `BGEU:npc_sel=br_src0>=br_src1;
        `JAL:npc_sel=2'b10;
        `JALR:npc_sel=2'b10;
        default:npc_sel=2'b00;
    endcase
end
endmodule
```

NPC 模块代码如下：

```
module NPC(input [31:0] pc_add4,
           input [31:0] pc_offset,
           input [31:0] pc_j,
           input [1:0] npc_sel,
           output reg [31:0] npc);
// 根据输入的npc_sel选择next_pc
always @(*) begin
    case (npc_sel)
        // pc+4
        2'b00:npc=pc_add4;
        // B-type
        2'b01:npc=pc_offset;
        // J-type
        2'b10:npc=pc_j;
```

```
        default: npc=pc_add4;
    endcase
end
endmodule
```

修改后 Decoder 部分如下:

```
`define ADD                 5'B00000
`define SUB                 5'B00010
`define SLT                 5'B00100
`define SLTU                5'B00101
`define AND                 5'B01001
`define OR                  5'B01010
`define XOR                 5'B01011
`define SLL                 5'B01110
`define SRL                 5'B01111
`define SRA                 5'B10000
`define SRC0                5'B10001
`define SRC1                5'B10010
/*---------------新增部分_begin------------------*/
`define LW 4'b0000
`define LH 4'b0001
`define LB 4'b0010
`define LHU 4'b0011
`define LBU 4'b0100
`define SW 4'b0101
`define SH 4'b0110
`define SB 4'b0111
`define BEQ 4'b0000
`define BNE 4'b0001
`define BLT 4'b0010
`define BGE 4'b0011
`define BLTU 4'b0100
`define BGEU 4'b0101
`define JALR 4'b0110
```

```
29 `define JAL 4'b0111
30 `define OTHERS 32'hFFFFFFFF
31 `define ADD4 4'b1111
32 `define PC_ADD4 2'b00
33 `define ALU_RES 2'b01
34 `define DMEM_RDATA 2'b10
35 `define ZERO 2'b11
36 /*---------------新增部分_end------------------*/
37 module DECODER (input [31 : 0] inst,
38                 output reg [4 : 0] alu_op,
39                 output [3 : 0] dmem_access,
40                 output reg [31 : 0] imm,
41                 output [4 : 0] rf_ra0,
42                 output [4 : 0] rf_ra1,
43                 output [4 : 0] rf_wa,
44                 output [0 : 0] rf_we,
45                 output reg [1 : 0] rf_wd_sel,
46                 output [0 : 0] alu_src0_sel,
47                 output [0 : 0] alu_src1_sel,
48                 output [3 : 0] br_type,
49                 output dmem_we);
50
51     reg [0:0] we;
52     reg [4:0] ra0;
53     reg [4:0] ra1;
54     reg [4:0] wa;
55     reg [0:0] src0_sel;
56     reg [0:0] src1_sel;
57 /*---------------新增部分_begin------------------*/
58     reg [31:0] d_a;
59     reg [31:0] b_t;
60     reg [0:0] d_we;
61 /*---------------新增部分_end------------------*/
```

```
62      initial begin
63          alu_op       = 0;
64          we       = 0;
65          ra0      = 0;
66          ra1      = 0;
67          wa       = 0;
68          src0_sel = 0;
69          src1_sel = 0;
70          imm      = 0;
71          d_a=`OTHERS;
72          b_t=`ADD4;
73          rf_wd_sel=`ALU_RES;
74          d_we=0;
75      end
76
77      assign rf_ra0       = ra0;
78      assign rf_ra1       = ra1;
79      assign rf_wa        = wa;
80      assign rf_we        = we;
81      assign alu_src0_sel = src0_sel;
82      assign alu_src1_sel = src1_sel;
83  /*---------------新增部分_begin------------------*/
84      assign dmem_access = d_a;
85      assign br_type=b_t;
86      assign dmem_we=d_we;
87  /*---------------新增部分_end------------------*/
88      always @(*) begin
89          if (inst[6:0] == 7'b0110011) begin
90              we       = 1;
91              ra0      = inst[19:15];
92              ra1      = inst[24:20];
93              wa       = inst[11:7];
94              src0_sel = 1'b0;
```

```
95              src1_sel = 1'b0;
96              imm      = 32'd00000000;
97              d_a=`OTHERS;
98              b_t=`ADD4;
99              rf_wd_sel=`ALU_RES;
100             d_we=0;
101             case ({inst[31:25], inst[14:12]})
102                 {7'b0000000, 3'b000}: alu_op = `ADD;
103                 {7'b0100000, 3'b000}: alu_op = `SUB;
104                 {7'b0000000, 3'b001}: alu_op = `SLL;
105                 {7'b0000000, 3'b010}: alu_op = `SLT;
106                 {7'b0000000, 3'b011}: alu_op = `SLTU;
107                 {7'b0000000, 3'b100}: alu_op = `XOR;
108                 {7'b0000000, 3'b101}: alu_op = `SRL;
109                 {7'b0100000, 3'b101}: alu_op = `SRA;
110                 {7'b0000000, 3'b110}: alu_op = `OR;
111                 {7'b0000000, 3'b111}: alu_op = `AND;
112                 default:              alu_op = 5'b11111;
113             endcase
114         end
115         else if (inst[6:0] == 7'b0010011) begin
116             we       = 1;
117             ra0      = inst[19:15];
118             ra1      = 5'b00000;
119             wa       = inst[11:7];
120             src0_sel = 1'b0;
121             src1_sel = 1'b1;
122             imm      = {{20{inst[31]}},inst[31:20]};
123             d_a=`OTHERS;
124             b_t=`ADD4;
125             rf_wd_sel=`ALU_RES;
126             d_we=0;
127             case (inst[14:12])
```

```
128                 3'b000: alu_op = `ADD;
129                 3'b010: alu_op = `SLT;
130                 3'b011: alu_op = `SLTU;
131                 3'b100: alu_op = `XOR;
132                 3'b110: alu_op = `OR;
133                 3'b111: alu_op = `AND;
134                 3'b001: begin
135                     alu_op   = `SLL;
136                     imm = {{27{inst[24]}},inst[24:20]};
137                 end
138                 3'b101:
139                 case (inst[31:25])
140                     7'b0000000:begin
141                         alu_op   = `SRL;
142                         imm = {{27{inst[24]}},inst[24:20]};
143                     end
144                     7'b0100000:begin
145                         alu_op   = `SRA;
146                         imm = {{27{inst[24]}},inst[24:20]};
147                     end
148                     default: alu_op = 5'b11111;
149                 endcase
150                 default:alu_op = 5'b11111;
151             endcase
152         end
153         else if (inst[6:0] == 7'b0110111) begin
154             we       = 1;
155             ra0      = 5'b00000;
156             ra1      = 5'b00000;
157             wa       = inst[11:7];
158             src0_sel = 1'b0;
159             src1_sel = 1'b1;
160             imm      = {inst[31:12],12'b0};
```

```
161             alu_op       = `SRC1;
162             d_a=`OTHERS;
163             b_t=`ADD4;
164             rf_wd_sel=`ALU_RES;
165             d_we=0;
166         end
167         else if (inst[6:0] == 7'b0010111) begin
168             we       = 1;
169             ra0      = 5'b00000;
170             ra1      = 5'b00000;
171             wa       = inst[11:7];
172             src0_sel = 1'b1;
173             src1_sel = 1'b1;
174             imm     = {inst[31:12],12'b0};
175             alu_op       = `ADD;
176             d_a=`OTHERS;
177             b_t=`ADD4;
178             rf_wd_sel=`ALU_RES;
179             d_we=0;
180         end
181 /*---------------新增部分_begin------------------*/
182         else if(inst[6:0] == 7'b0000011) begin
183             we       = 1;
184             ra0      = inst[19:15];
185             ra1      = 5'b00000;
186             wa       = inst[11:7];
187             src0_sel = 1'b0;
188             src1_sel = 1'b1;
189             imm      = {{20{inst[31]}},inst[31:20]};
190             alu_op   = `ADD;
191             b_t=`ADD4;
192             rf_wd_sel=`DMEM_RDATA;
193             d_we=0;
```

```
194             case (inst[14:12])
195                 3'b000:d_a=`LB;
196                 3'b001:d_a=`LH;
197                 3'b010:d_a=`LW;
198                 3'b100:d_a=`LBU;
199                 3'b101:d_a=`LHU;
200                 default: d_a=`OTHERS;
201             endcase
202         end else if(inst[6:0]==7'b0100011)begin
203             we=0;
204             ra0=inst[19:15];
205             ra1=inst[24:20];
206             wa=0;
207             src0_sel=1'b0;
208             src1_sel=1'b1;
209             imm={{20{inst[31]}},inst[31:25],inst[11:7]};
210             alu_op=`ADD;
211             b_t=`ADD4;
212             rf_wd_sel=`ZERO;
213             d_we=1;
214             case (inst[14:12])
215                 3'b000:d_a=`SB;
216                 3'b001:d_a=`SH;
217                 3'b010:d_a=`SW;
218                 default:d_a=`OTHERS;
219             endcase
220         end else if (inst[6:0]==7'b1101111)begin
221             we=1;
222             ra0=0;
223             ra1=0;
224             wa=inst[11:7];
225             src0_sel=1'b1;
226             src1_sel=1'b1;
```

```
227             imm={{11{inst[31]}},inst[31],inst[19:12],inst
                    [20],inst[30:21],1'b0};
228             alu_op=`ADD;
229             d_a=`OTHERS;
230             b_t=`JAL;
231             rf_wd_sel=`PC_ADD4;
232             d_we=0;
233         end else if (inst[6:0]==7'b1100111)begin
234             we=1;
235             ra0=inst[19:15];
236             ra1=0;
237             wa=inst[11:7];
238             src0_sel=1'b0;
239             src1_sel=1'b1;
240             imm={{20{inst[31]}},inst[31:20]};
241             alu_op=`ADD;
242             d_a=`OTHERS;
243             b_t=`JALR;
244             rf_wd_sel=`PC_ADD4;
245             d_we=0;
246         end else if (inst[6:0]==7'b1100011)begin
247             we=0;
248             ra0=inst[19:15];
249             ra1=inst[24:20];
250             wa=0;
251             src0_sel=1'b1;
252             src1_sel=1'b1;
253             imm={{19{inst[31]}},inst[31],inst[7],inst
                    [30:25],inst[11:8],1'b0};
254             alu_op=`ADD;
255             d_a=`OTHERS;
256             rf_wd_sel=`ZERO;
257             d_we=0;
```

```
258             case (inst[14:12])
259                 3'b000:b_t=`BEQ;
260                 3'b001:b_t=`BNE;
261                 3'b100:b_t=`BLT;
262                 3'b101:b_t=`BGE;
263                 3'b110:b_t=`BLTU;
264                 3'b111:b_t=`BGEU;
265                 default: b_t=`OTHERS;
266             endcase
267         end
268 /*---------------新增部分_end------------------*/
269         else begin
270             we        = 0;
271             ra0       = 5'b00000;
272             ra1       = 5'b00000;
273             wa        = 5'b00000;
274             src0_sel  = 1'b0;
275             src1_sel  = 1'b0;
276             imm      = 32'b0;
277             alu_op        = 5'b11111;
278             d_a=`OTHERS;
279             b_t=`ADD4;
280             rf_wd_sel=`ZERO;
281             d_we=0;
282         end
283     end
284 endmodule
```

在 Decoder 中新增了对 J-type, B-type 和 I-type(读写数据存储器部分)的支持，以上代码中标注出了主要的新增部分。CPU 模块核心代码如下:

```
1     wire [31:0] cur_npc; // 当前的next_pc
2     wire [31:0] cur_pc; // 当前的pc
3     wire [31:0] cur_inst; //当前的指令
```

```
4     wire [31:0] pc_add4; //pc+4
5     wire [31:0] pc_offset; //pc+offset
6     wire [4:0] rf_ra0; // 寄存器读地址
7     wire [4:0] rf_ra1; // 寄存器读地址
8     wire [4:0] rf_wa; // 寄存器写地址
9     wire [31:0] rf_wd; // 写入寄存器的值
10    wire [4:0] alu_op; // alu的opcode
11    wire alu_src0_sel; // 选择输入alu的数据
12    wire alu_src1_sel; // 选择输入alu的数据
13    wire [31:0] imm; // 立即数
14    wire rf_we; //寄存器写使能信号
15    wire [31:0] rf_rd0; // 寄存器读取出的值
16    wire [31:0] rf_rd1; // 寄存器读取出的值
17    wire [31:0] alu_src0; // 输入alu的操作数
18    wire [31:0] alu_src1; // 输入alu的操作数
19    wire [31:0] alu_res; // alu的计算结果
20    wire [31:0] pc_j; // pc跳转的地址(J-type)
21    wire [1:0] npc_sel; // 选择next_pc
22    wire [3:0] dmem_access; // 选择对DATA_MEM操作类型
23    wire [1:0] rf_wd_sel; // 选择写回寄存器的数据
24    wire [3:0] br_type; // 分支跳转类型
25    wire [31:0] dmem_wd_in; // 写入DATA_MEM的数据
26    wire [31:0] dmem_wd_out; // 写入DATA_MEM的数据
27    wire [31:0] dmem_rd_in; // 从DATA_MEM读取的数据
28    wire [31:0] dmem_rd_out; // 从DATA_MEM读取的数据
29    wire dmem_we_; // DATA_MEM的写使能信号
30
31    assign global_en  = !(cur_inst == 32'H00000013);
32    assign imem_raddr = {{cur_pc-32'h00400000}/'d4};
33    assign cur_inst   = imem_rdata;
34    assign pc_offset  = alu_res;
35    assign pc_j       = alu_res&~1;
36    assign dmem_wd_in = rf_rd1;
```

```
37      assign dmem_we    = dmem_we_;
38      assign dmem_addr  = {(alu_res-32'h10010000)/'d4};
39      assign dmem_wdata = dmem_wd_out;
40      assign dmem_rd_in = dmem_rdata;
41      PC_PLUS4 pc_plus(
42      .pc(cur_pc),
43      .pc_plus4(pc_add4)
44      );
45
46      NPC npc(
47      .pc_offset(pc_offset),
48      .pc_add4(pc_add4),
49      .pc_j(pc_j),
50      .npc_sel(npc_sel),
51      .npc(cur_npc)
52      );
53
54      PC pc(
55      .clk    (clk),
56      .rst    (rst),
57      // 当 global_en 为高电平时, PC 才会更新,
58      // CPU 才会执行指令。
59      .en     (global_en),
60      .npc    (cur_npc),
61      .pc     (cur_pc)
62      );
63
64      DECODER decoder(
65      .inst(cur_inst),
66      .alu_op(alu_op),
67      .imm(imm),
68      .rf_ra0(rf_ra0),
69      .rf_ra1(rf_ra1),
```

```
70      .rf_wa(rf_wa),
71      .rf_we(rf_we),
72      .alu_src0_sel(alu_src0_sel),
73      .alu_src1_sel(alu_src1_sel),
74      .dmem_access(dmem_access),
75      .rf_wd_sel(rf_wd_sel),
76      .br_type(br_type),
77      .dmem_we(dmem_we_)
78      );
79
80      REG_FILE reg_file(
81      .clk(clk),
82      .rf_ra0(rf_ra0),
83      .rf_ra1(rf_ra1),
84      .rf_wa(rf_wa),
85      .rf_we(rf_we),
86      .rf_wd(rf_wd),
87      .rf_rd0(rf_rd0),
88      .rf_rd1(rf_rd1),
89      .debug_reg_rd(debug_reg_rd),
90      .debug_reg_ra(debug_reg_ra)
91      );
92
93
94      MUX1 mux1(
95      .src0(rf_rd0),
96      .src1(cur_pc),
97      .sel(alu_src0_sel),
98      .res(alu_src0)
99      );
100     MUX1 mux2(
101     .src0(rf_rd1),
102     .src1(imm),
```

```
103     .sel(alu_src1_sel),
104     .res(alu_src1)
105     );
106 
107     ALU alu(
108     .alu_src0(alu_src0),
109     .alu_src1(alu_src1),
110     .alu_op(alu_op),
111     .alu_res(alu_res)
112     );
113 
114     SLU slu(
115     .addr(alu_res),
116     .dmem_access(dmem_access),
117     .rd_in(dmem_rd_in),
118     .wd_in(dmem_wd_in),
119     .rd_out(dmem_rd_out),
120     .wd_out(dmem_wd_out)
121     );
122 
123     MUX2 RF_MUX(
124     .src0(pc_add4),
125     .src1(alu_res),
126     .src2(dmem_rd_out),
127     .src3(0),
128     .sel(rf_wd_sel),
129     .res(rf_wd)
130     );
131 
132     BRANCH branch(
133     .br_type(br_type),
134     .br_src0(rf_rd0),
135     .br_src1(rf_rd1),
```

```
136     .npc_sel(npc_sel)
137     );
```

在 CPU 模块中加入了 BRANCH 模块，NPC 模块，SLU 模块以及 MUX2 模块（用于控制）写回寄存器的数据。

3 仿真结果与分析

3.1 test1（普通测试）

图 1: test1（普通测试）仿真结果

zero	0	0x00000000
ra	1	0x163b9620
sp	2	0xdcf52630
gp	3	0x163b9620
tp	4	0x00000000
t0	5	0x7a3ec640
t1	6	0x00000000
t2	7	0x7a3ec640
s0	8	0x00000001
s1	9	0x00000000
a0	10	0x000005b8
a1	11	0x000005b8
a2	12	0xfffffade
a3	13	0x1d5ea000
a4	14	0xdcf52630
a5	15	0x00000000
a6	16	0x00000000
a7	17	0xebb3a000
s2	18	0x00000001
s3	19	0x00000000
s4	20	0x4776d650
s5	21	0x0000007c
s6	22	0x4776d650
s7	23	0x00000001
s8	24	0xebb3a000
s9	25	0x00000000
s10	26	0x0000056d
s11	27	0xc8573610
t3	28	0xfffffbef
t4	29	0x00000000
t5	30	0x00000001
t6	31	0x000004a2
pc		0x00400680

图 2: 真实结果

仿真结果与真实结果相对应。

3.2 test2（分支与访存测试）

图 3: test2（分支与访存测试）

zero	0	0x00000000
ra	1	0x100193bf
sp	2	0x00000000
gp	3	0x10029317
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x10029ec8
t2	7	0x00000000
s0	8	0x1001f720
s1	9	0x1003e27a
a0	10	0x1001f860
a1	11	0x00000000
a2	12	0x10019699
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x10029afa
s3	19	0x00000000
s4	20	0x1001969a
s5	21	0x00000000
s6	22	0x10015f7a
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x1003947e
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x1002a163
t5	30	0x1001959c
t6	31	0x00000000
pc		0x00400724

图 4: 真实结果

仿真结果与真实结果相对应。

3.3 斐波那契数列

图 5: 大整数版本的斐波那契数列，输入 N=80，x3 和 x4 寄存器分别存储结果的高位与低位

仿真的出的结果为 533163_ef0321e5('H23416728348467685)，与正确结果相一致。

4 电路设计与分析

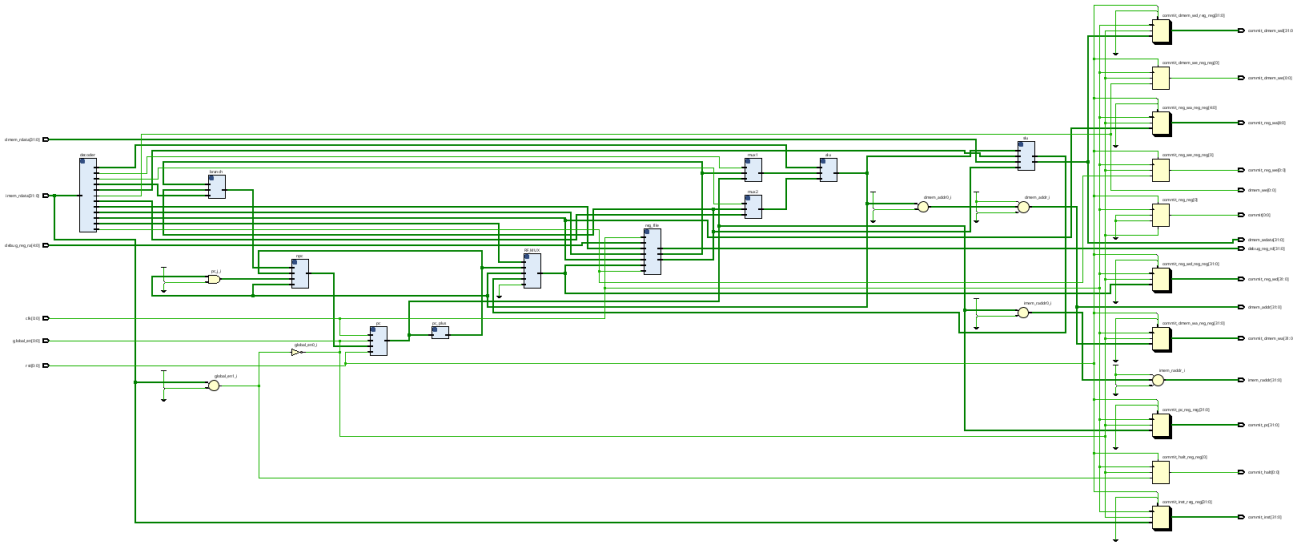

图 6: CPU

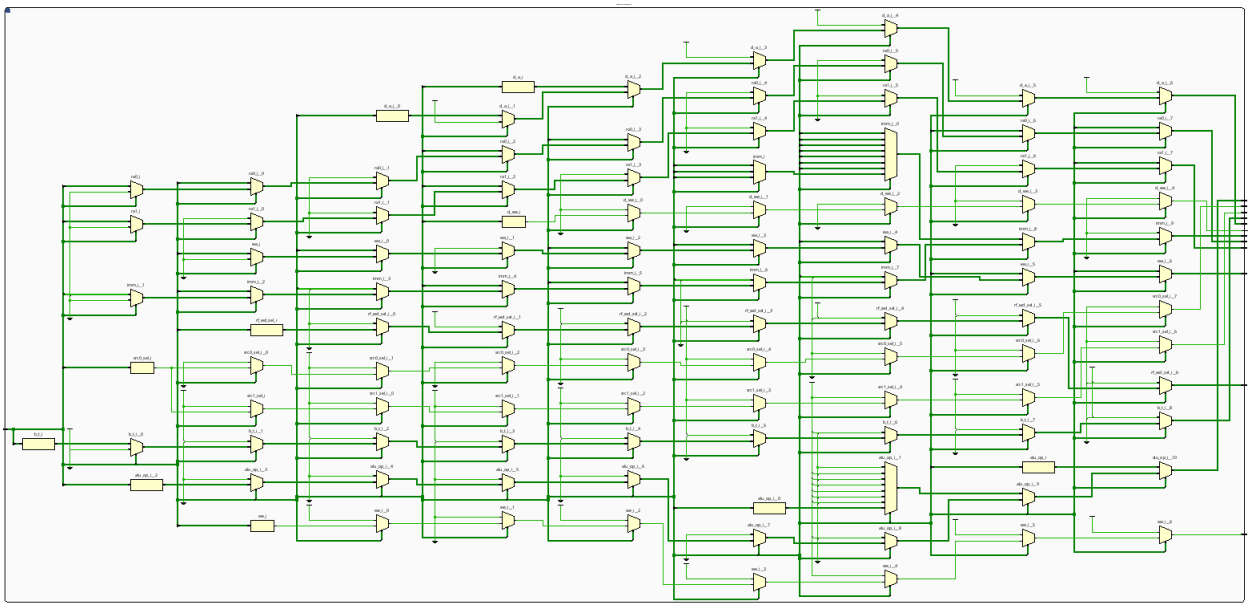

图 7: DECODER

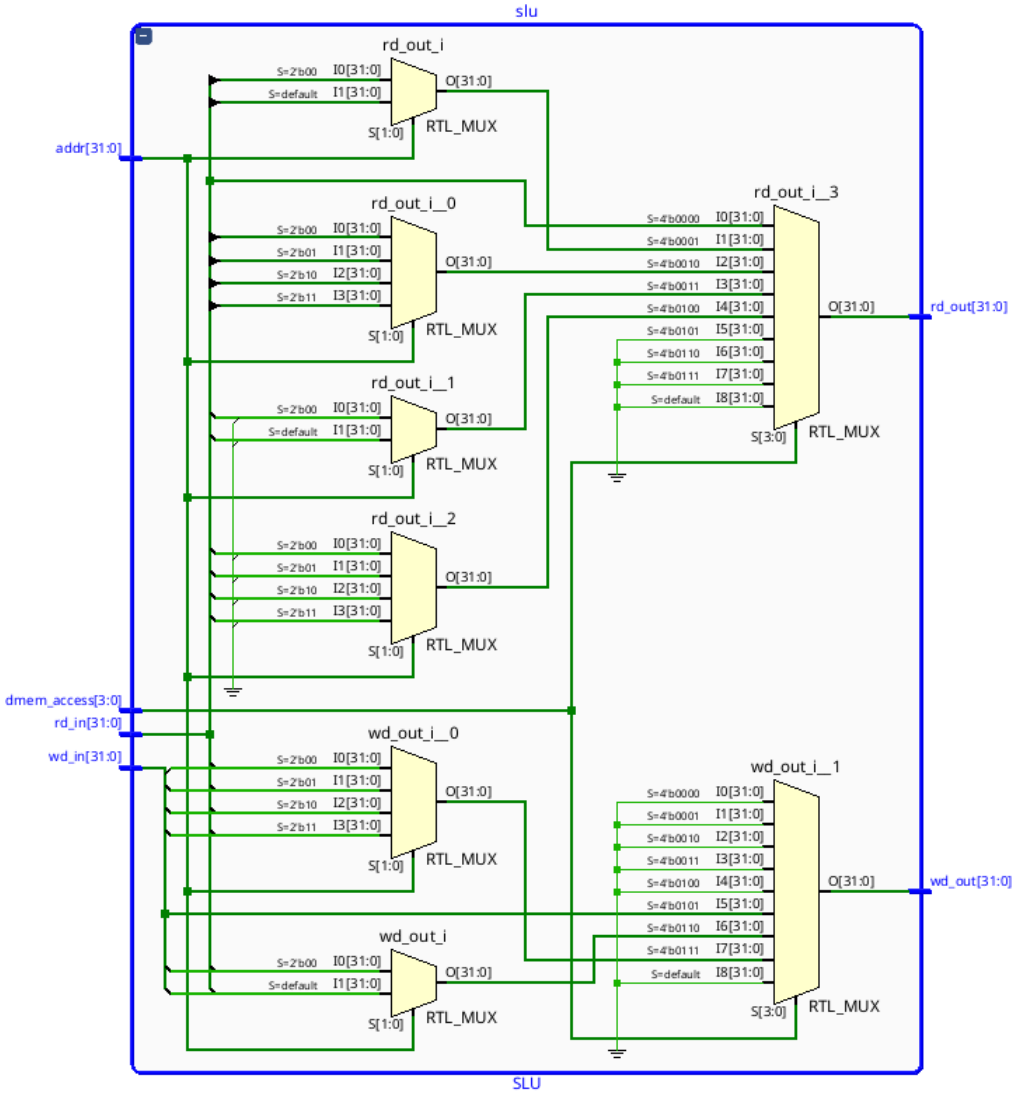
slu
rd_out_i
S=2'b00 I0[31:0]
S=default I1[31:0]
O[31:0]
S[1:0]
RTL_MUX
addr[31:0]
rd_out_i__0
S=2'b00 I0[31:0]
S=2'b01 I1[31:0]
S=2'b10 I2[31:0]
S=2'b11 I3[31:0]
O[31:0]
S[1:0]
RTL_MUX
rd_out_i__1
S=2'b00 I0[31:0]
S=default I1[31:0]
O[31:0]
S[1:0]
RTL_MUX
rd_out_i__2
S=2'b00 I0[31:0]
S=2'b01 I1[31:0]
S=2'b10 I2[31:0]
S=2'b11 I3[31:0]
O[31:0]
S[1:0]
RTL_MUX
rd_out_i__3
S=4'b0000 I0[31:0]
S=4'b0001 I1[31:0]
S=4'b0010 I2[31:0]
S=4'b0011 I3[31:0]
S=4'b0100 I4[31:0]
S=4'b0101 I5[31:0]
S=4'b0110 I6[31:0]
S=4'b0111 I7[31:0]
S=default I8[31:0]
O[31:0]
S[3:0]
RTL_MUX
rd_out[31:0]
dmem_access[3:0]
rd_in[31:0]
wd_in[31:0]
wd_out_i__0
S=2'b00 I0[31:0]
S=2'b01 I1[31:0]
S=2'b10 I2[31:0]
S=2'b11 I3[31:0]
O[31:0]
S[1:0]
RTL_MUX
wd_out_i
S=2'b00 I0[31:0]
S=default I1[31:0]
O[31:0]
S[1:0]
RTL_MUX
wd_out_i__1
S=4'b0000 I0[31:0]
S=4'b0001 I1[31:0]
S=4'b0010 I2[31:0]
S=4'b0011 I3[31:0]
S=4'b0100 I4[31:0]
S=4'b0101 I5[31:0]
S=4'b0110 I6[31:0]
S=4'b0111 I7[31:0]
S=default I8[31:0]
O[31:0]
S[3:0]
RTL_MUX
wd_out[31:0]
SLU

图 8: SLU

5 测试结果与分析

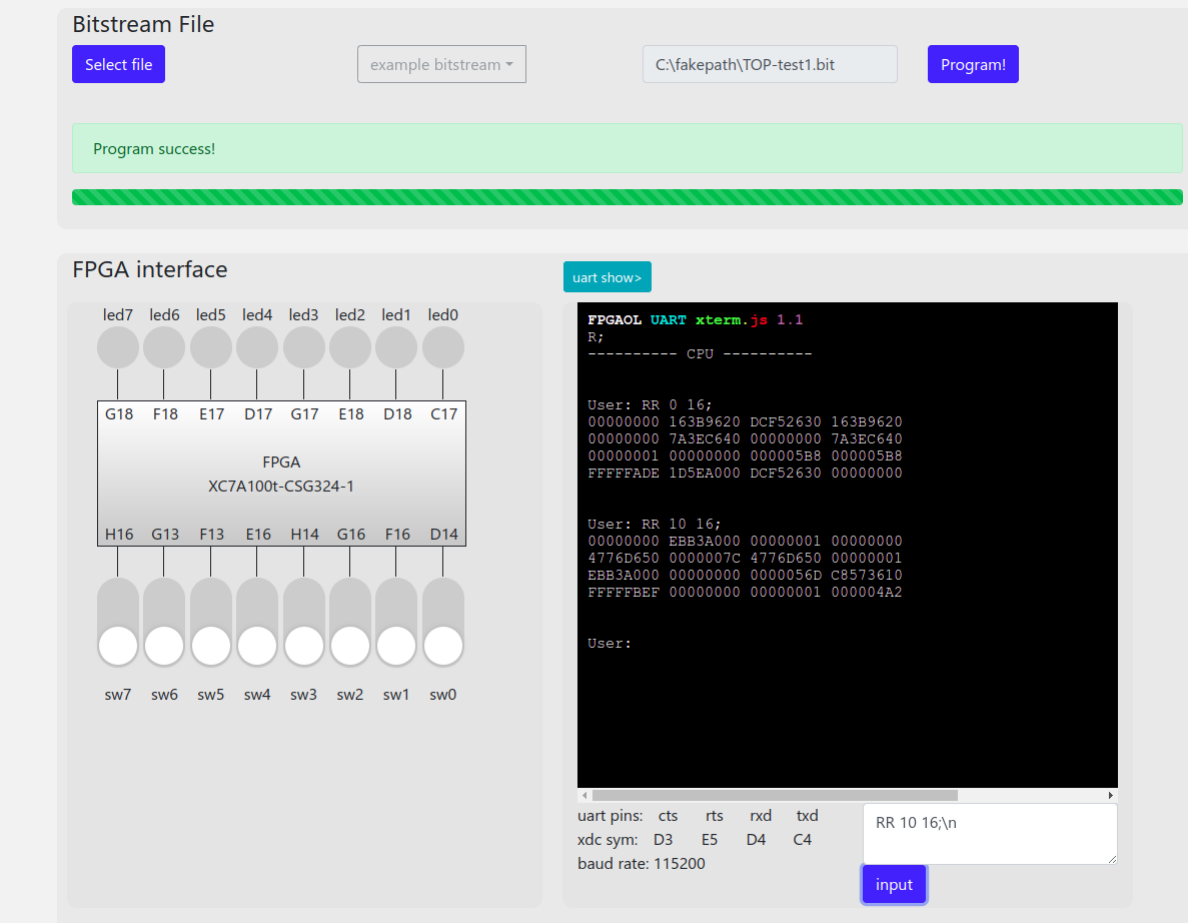

图 9: test1（普通测试）仿真结果

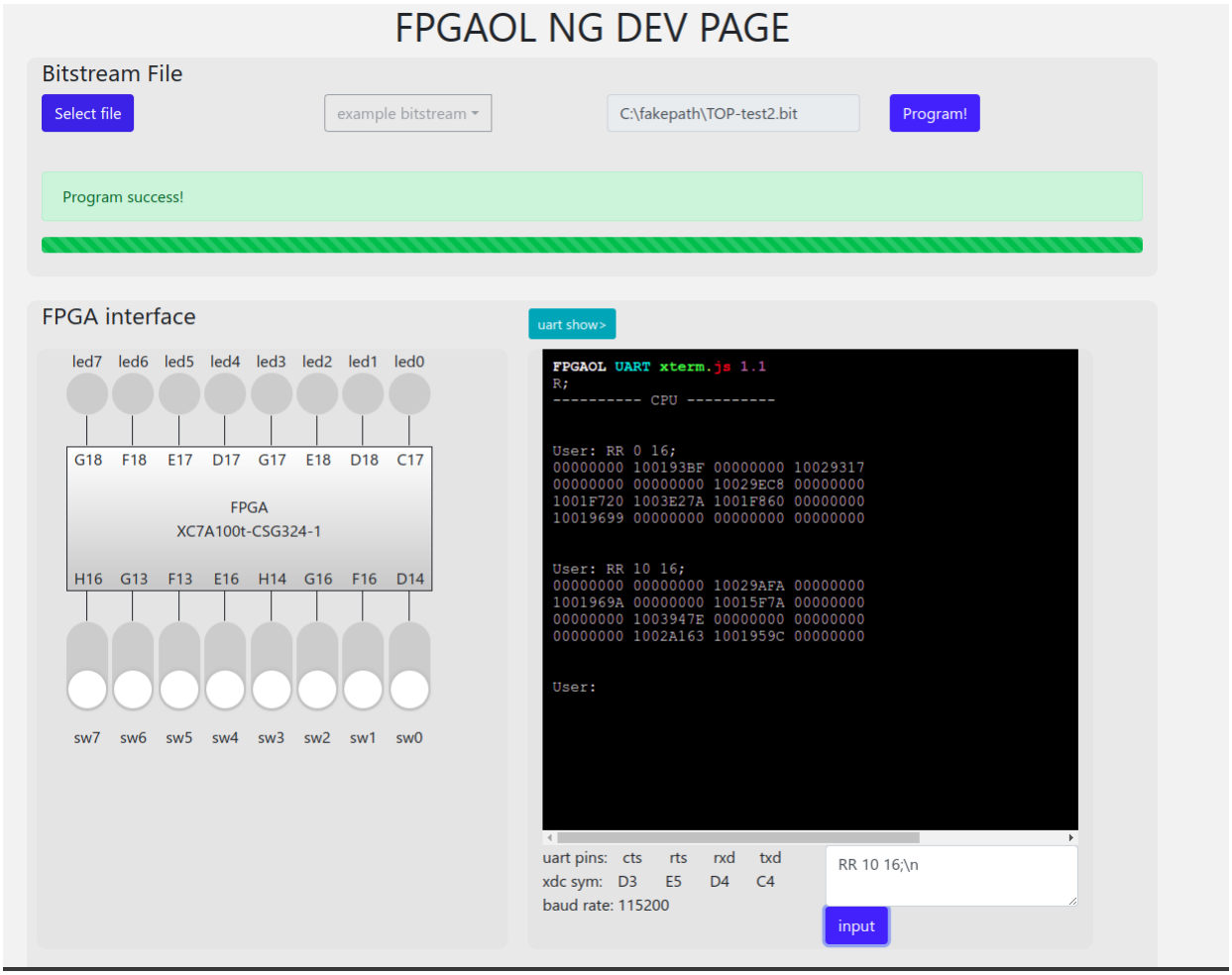

图 10: test2（分支与访存测试）

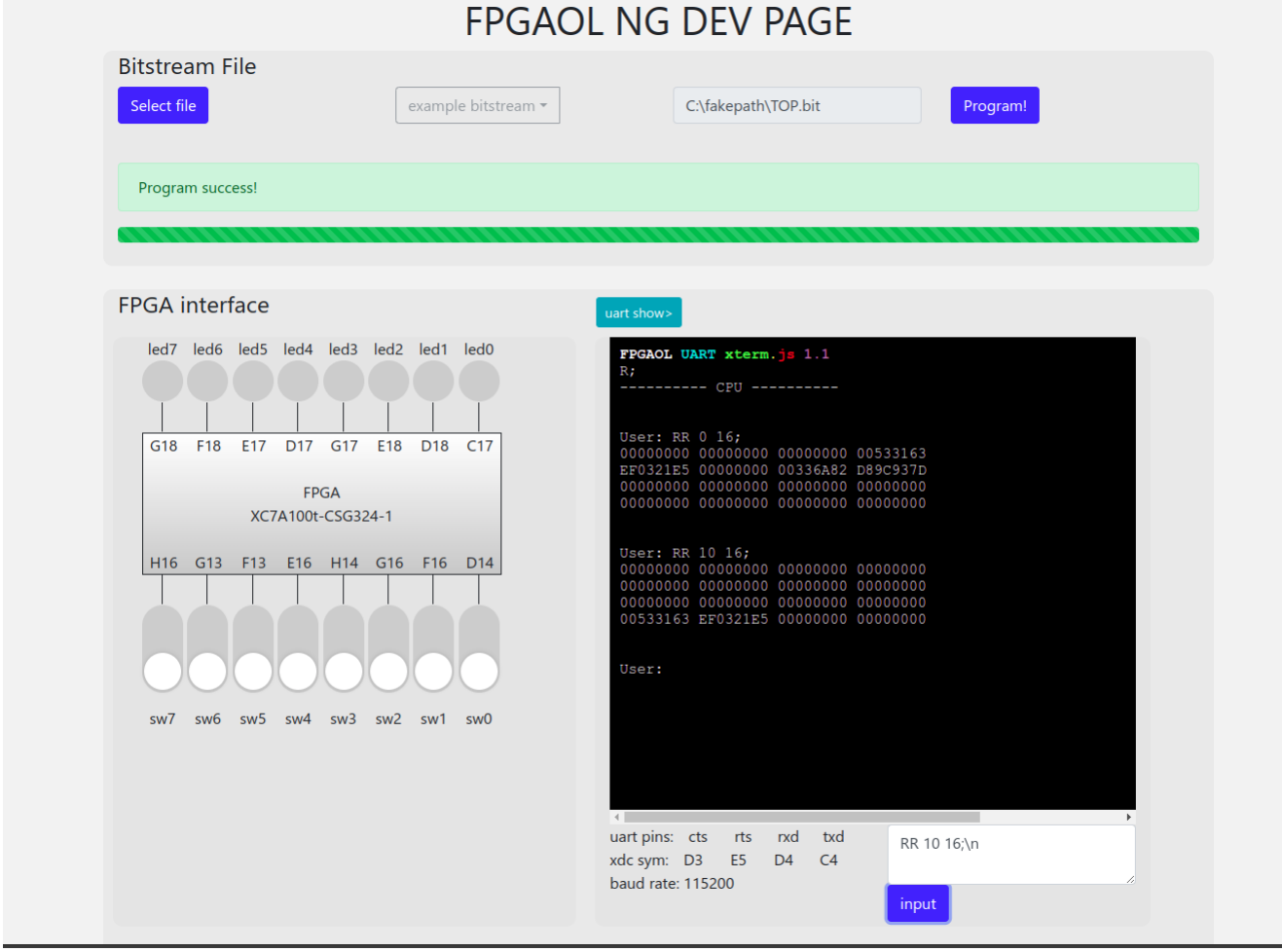

图 11: 大整数版本的斐波那契数列，输入 N=80，x3 和 x4 寄存器分别存储结果的高位与低位

6 思考与总结

6.1 掩码访问

SLU 代码如下：

```
`define LW 4'b0000
`define LH 4'b0001
`define LB 4'b0010
`define LHU 4'b0011
```

```
`define LBU 4'b0100
`define SW 4'b0101
`define SH 4'b0110
`define SB 4'b0111
module SL_UNIT  (input [31 : 0] addr,
            input [3 : 0] dmem_access,
            input [31 : 0] rd_in,
            input [31 : 0] wd_in,
            output reg [31 : 0] rd_out,
            output reg [31 : 0] wd_out,
            output reg [0:3] mask);
wire [31:0] addr_;
initial begin
    rd_out=0;
    wd_out=0;
    mask=0;
end
assign addr_=addr-32'h10010000;
always @(*) begin
    case(dmem_access)
        `LW:
        begin
            mask=4'b1111;
            rd_out = rd_in;
            wd_out =0;
        end
        `LH:begin
            mask=4'b1111;
            wd_out =0;
            if(addr_%4 == 0)
                rd_out = {{16{rd_in[15]}},rd_in[15:0]};
            else
                rd_out = {{16{rd_in[31]}},rd_in[31:16]};
```

```
38          end
39          `LB:begin
40              mask=4'b1111;
41              case(addr_%4)
42                  0:rd_out = {{24{rd_in[7]}},rd_in[7:0]};
43                  1:rd_out = {{24{rd_in[15]}},rd_in[15:8]};
44                  2:rd_out = {{24{rd_in[23]}},rd_in[23:16]};
45                  3:rd_out = {{24{rd_in[31]}},rd_in[31:24]};
46              endcase
47              wd_out =0;
48          end
49
50          `LHU:begin
51              mask=4'b1111;
52              if(addr_%4 == 0)
53                  rd_out = {16'b0,rd_in[15:0]};
54              else
55                  rd_out = {16'b0,rd_in[31:16]};
56              wd_out =0;
57          end
58
59          `LBU:begin
60              mask=4'b1111;
61              case(addr_%4)
62                  0:rd_out = {24'b0,rd_in[7:0]};
63                  1:rd_out = {24'b0,rd_in[15:8]};
64                  2:rd_out = {24'b0,rd_in[23:16]};
65                  3:rd_out = {24'b0,rd_in[31:24]};
66              endcase
67              wd_out =0;
68          end
69          `SW:begin
70              mask=4'b1111;
```

```
71              wd_out=wd_in;
72              rd_out = 0;
73          end
74          `SH:begin
75              if(addr_%4 == 0)
76                  mask=4'b0011;
77              else
78                  mask=4'b1100;
79              wd_out=wd_in;
80              rd_out = 0;
81          end
82
83          `SB:begin
84              case(addr_%4)
85                  0:mask=4'b0001;
86                  1:mask=4'b0010;
87                  2:mask=4'b0100;
88                  3:mask=4'b1000;
89              endcase
90              wd_out=wd_in;
91              rd_out = 0;
92          end
93          default:
94              begin
95                  mask=4'b0000;
96                  rd_out = 0;
97                  wd_out =0;
98              end
99      endcase
100 end
101 endmodule
```

在读取阶段，掩码 mask 统一为 4'b1111，这是因为即使利用掩码只读取所需的部分，实际返回值仍然是一个 32bit 的数字，需要从中提取出有效

部分，因此在我看来，在此阶段无需做额外的处理。

但是在写入阶段，通过掩码可以直接将一整个 wd_in 写入，而无需担心影响其他位，可以使代码更加简化。

6.2 可能的关键路径

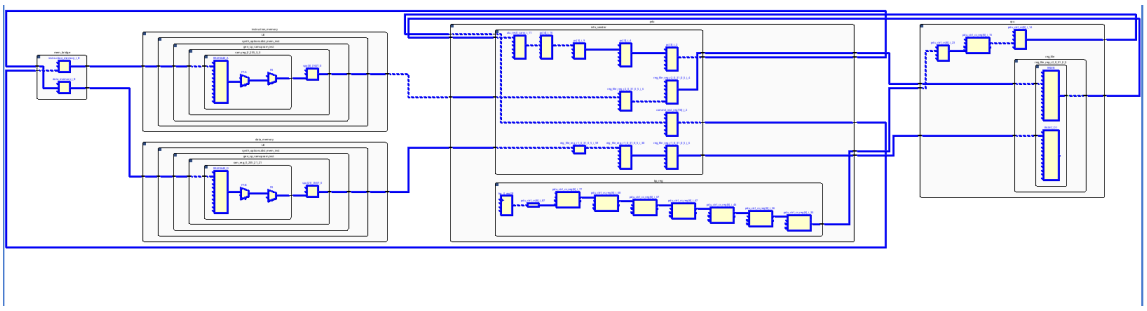

图 12: 可能的关键路径

以下是对各部分放大（从左往右）：

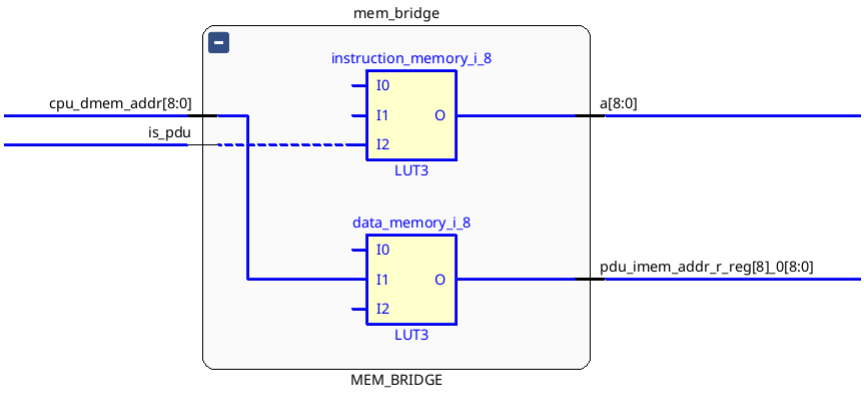

图 13: mem_bridge

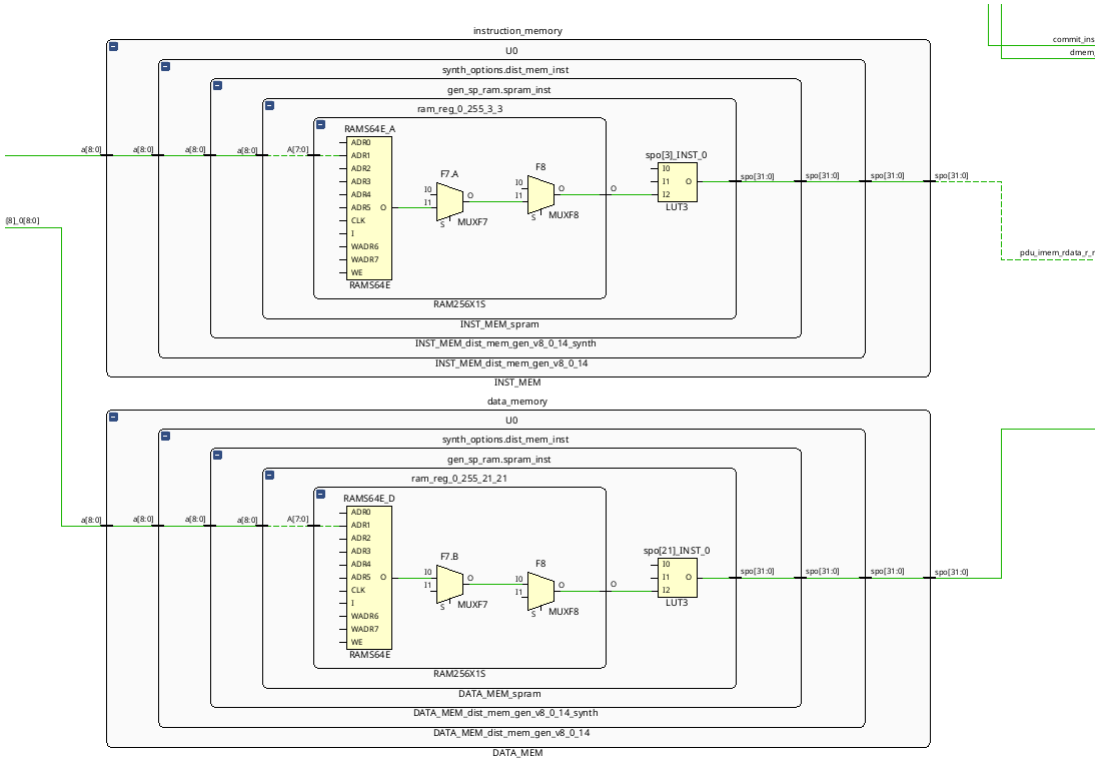

图 14: INST_MEM 和 DATA_MEM

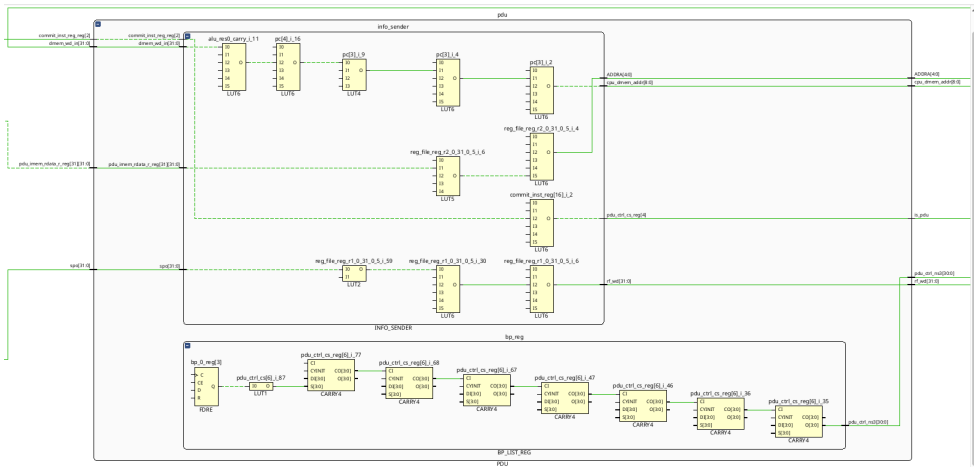

图 15: PDU

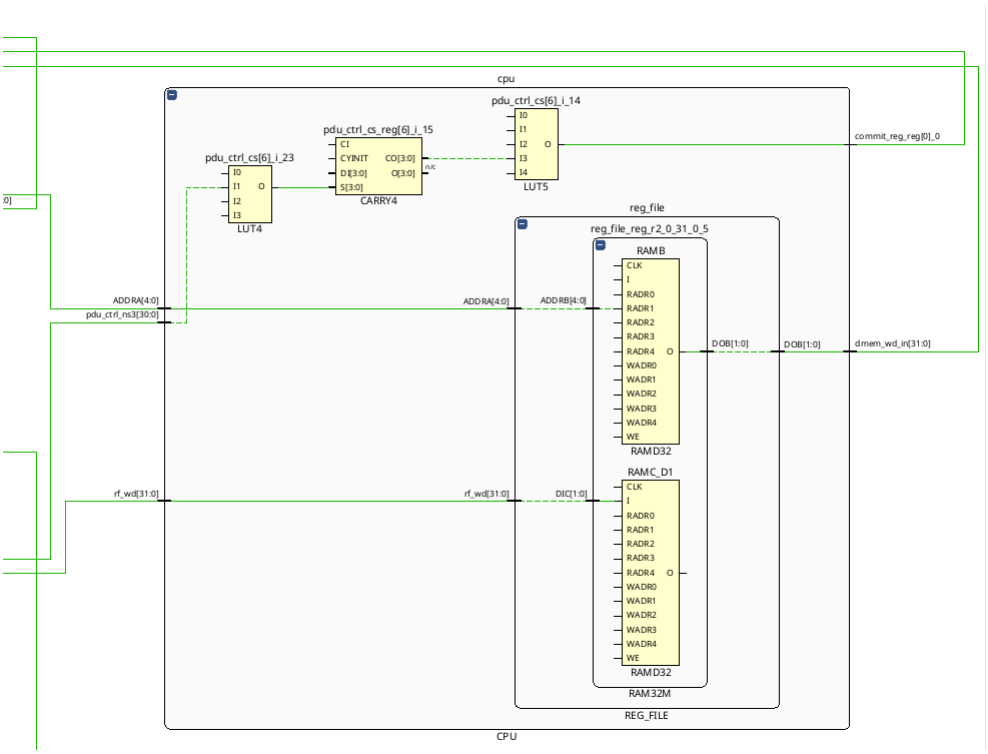
cpu
pdu_ctrl_cs[6]_i_14
LUT5
pdu_ctrl_cs_reg[6]_i_15
CARRY4
pdu_ctrl_cs[6]_i_23
LUT4
commit_reg_reg[0]_0
reg_file
reg_file_reg_r2_0_31_0_5
RAMB
RAMD32
RAMC_D1
RAMD32
RAM32M
REG_FILE
ADDRA[4:0]
pdu_ctrl_ns3[30:0]
ADDRA[4:0]
ADDRB[4:0]
DOB[1:0]
DOB[1:0]
dmem_wd_in[31:0]
rf_wd[31:0]
rf_wd[31:0]
DIC[1:0]
CPU

图 16: CPU