CODH - 综合实验　实验报告

院系:　　姓名:　　学号:

2023 年 7 月 13 日

1　实验目的

- 掌握 Cache 基本原理、结构、设计和调试方法
- 掌握 CPU 输入/输出的编址和控制方式
- 熟练掌握数据通路和控制器的设计和描述方法

2　实验环境

- macOS 13.0
- Rars1_5.jar (Riscv Assembler and Runtime Simulator)
- Vivado 2019.3
- Nexys 4 DDR 开发板

3　实验内容

3.1　Dcache 与 Dmem

- 熟练掌握数据通路和控制器的设计和描述方法
- 将原来的数据存储器改为 Dmem，假定按块访问，首字读取延迟 16 个时钟，随后每字 1 个时钟
- 添加缓存 Dcache，要求使用直接/二路组相连实现，采用写回写分配策略
- Dcache 与 CPU，Dcache 与 Dmem 之间都使用 Ready/Valid 实现

3.2　IOU

- CPU 使用存储器映射的方式输入输出以对外设进行访问
- 主要的 IO 端口有：led，switch，btn，seg 等
- 采用查询式输出过程和查询式输入过程
- 通过开关输入数据，LED 及数码管输出显示

3.3 测试程序

- 使用 IOU 输入数组大小以及数组首元素
- 通过 LFSR 算法生成出数组其他元素
- 利用之前的排序代码排序，并检查排序结果是否正确。
- 排序前后计算所用时钟周期数

4 逻辑设计 / 核心代码

4.1 Dcache 与 Dmem 的实现

首先计算地址格式，要求 Dmem 总容量 4KB，设置块大小为 4 字，因此 1 块 =4 字 =16byte=128 位，总共分为 256 块，块地址应该有 8 位，字地址应该有 8+2=10 位。

使用 IP 核例化相应存储器后，在其上添加一层以满足实验需求。

在这里 Dmem 使用了 FSM 来实现，分为七个状态：IDLE，DELAY，WRITE，WT_RDY，READ，RD_RDY，FINISH。

初始 Dmem 处于 IDLE 状态，在读/写的 valid 信号有效后，进入 DELAY 状态模拟延时 16 个时钟周期，然后进入相应的读写状态，读写完成后置相应的 ready 位有效，再次经由 FINISH 状态回到 IDLE 状态。

```
    if(~rstn) begin
        state <= IDLE;
        r_ready <= 0;
        w_ready <= 0;
        delay_cnt <= 0;
        offset_cnt <= 0;
        wt_en <= 0;
    end
    else begin
        case(state)
            IDLE: begin
                if(w_valid || r_valid) begin
                    r_ready <= 0;
                    w_ready <= 0;
                    state <= DELAY;
                end
                else begin
                    r_ready <= 0;
                    w_ready <= 0;
                    state <= IDLE;
                end
            end
            DELAY: begin
                if(delay_cnt < CYCLE) begin //wait 16 cycles
                    delay_cnt = delay_cnt + 1;
                    offset_cnt <= 0;
                end
                else begin
                    if(w_valid) begin
                        delay_cnt <= 0;
                        offset_cnt <= 0;
                        wt_en <= 1;
```

```
                    state <= WRITE;
                end
                else if(r_valid) begin
                    delay_cnt <= 0;
                    offset_cnt <= 1;
                    rd_addr <= {r_addr, offset_cnt[1:0]};
                    state <= READ;
                end
            end
        end
        WRITE: begin
            if(offset_cnt < WORD_SIZE - 1)begin
                //wt_data = ...
                wt_en <= 1;
                offset_cnt <= offset_cnt + 1;
            end
            else begin
                offset_cnt <= 0;
                wt_en <= 0;
                state <= WT_RDY;
            end
        end
        WT_RDY: begin
            w_ready <= 1;
            state <= FINISH;
        end
        READ: begin
            wt_en <= 0;
            if(offset_cnt < WORD_SIZE) begin
                rd_addr <= {r_addr, offset_cnt[1:0]};
                r_data <= {dout, r_data[127:32]}; // 0,
                offset_cnt <= offset_cnt + 1;
            end
            else begin
                r_data <= {dout, r_data[127:32]};
                offset_cnt <= 0;
                state <= RD_RDY;
            end
        end
        RD_RDY: begin
            r_ready <= 1;
            state <= FINISH;
        end
        FINISH: begin
            r_ready <= 0;
            w_ready <= 0;
            wt_en <= 0;
            state <= IDLE;
        end

    endcase
end
```

Dmem.v

接下来是 Dcache，对地址进行分析，应该有 `addr[9:0] = tag[9:7], index[6:2], offset[1:0]` 我们首先实现的是二路组相连 Dcache，后期实现直接映射 Cache 时只需要修改地址格式并且将附加装置去掉即可。

Dcache 的整体设计，参考了普通班的文档，但因为实际要求不同做了些许修改，主要核心是两个寄存器作为 cache，两个寄存器作为 tag，以及两个记录寄存器分别记录 dirty 以及 valid。除此之外控制信号的产生还需要有相应的 FSM，写回时需要选路的基于 LRU 的选择器，以及写入 cache 时的数据插入器（cache 是 4 字，但是 Dcache 每次写入是 1 字，需要插入相应位置）。

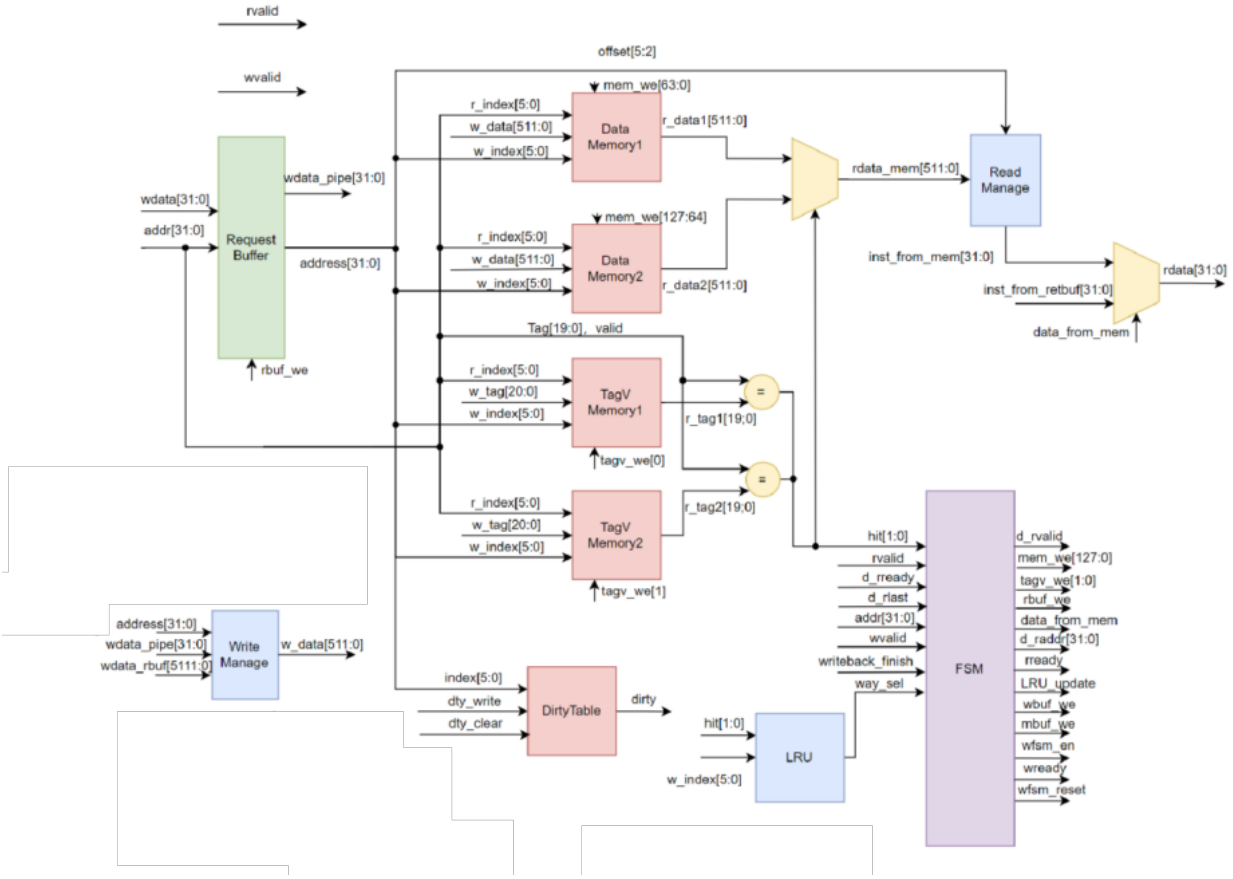

图 1: CPU-fig

Dcache 还有一个重点是 FSM 的设计，这里参照了书本上的简化 FSM 实现。主要分为 IDLE，WRITE_BACK，FETCH，FINISH 几个阶段。

初始 Dcache 处于 IDLE 状态，当 CPU 发出读/写请求时，根据地址计算出 tag 以及 index，然后根据 index 去 cache 中查找，如果命中则进入下一周期就置 ready，仍然停留在 IDLE 状态。否则需要替换 cache，先检查 LRU 选择的 cache 块是否 dirty，若是则进入 WRITE_BACK 将块写回 Dmem，若不是则进入 FETCH 阶段直接 fetch 所需要读写的块，然后在 FINISH 阶段对 cache 块中的数据进行读写，置 ready 位有效并跳回 IDLE 状态。

具体代码如下

```
case (state)
    IDLE: begin
        if (w_valid || r_valid) begin
            if (hit) begin //stay IDLE
                //更新 LRU
                LRU_change <= 0;
                LRU_update <= 1;
                if(w_valid) begin
                    w_data_sel <= 0; // w_data_write = (w_word + r_data)
```

```
                    dty_write <= 1;
                    w_ready <= 1;
                    r_ready <= 0;
                    if(hit1) begin
                        mem_we1 <= 1;
                        tag_we1 <= 0;
                    end
                    else if (hit2) begin
                        mem_we2 <= 1;
                        tag_we2 <= 0;
                    end
                    state <= FINISH;
                end
                else if(r_valid) begin
                    data_from_mem <= 0;
                    dty_write <= 0;
                    r_ready <= 1;
                    w_ready <= 0;
                    valid_write <= 0;
                    mem_we1 <= 0;
                    mem_we2 <= 0;
                    tag_we1 <= 0;
                    tag_we2 <= 0;
                    state <= IDLE;
                end
            end
            else begin //miss
                LRU_change <= 0;
                LRU_update <= 0;
                w_ready <= 0;
                r_ready <= 0;
                dty_clear <= 0;
                dty_write <= 0;
                valid_write <= 0;
                mem_we1 <= 0;
                mem_we2 <= 0;
                tag_we1 <= 0;
                tag_we2 <= 0;
                if(dirty) begin
                    dw_valid <= 1;
                    dr_valid <= 0;
                    state <= WRITE_BACK;
                end
                else begin
                    dr_valid <= 1;
                    dw_valid <= 0;
                    state <= FETCH;
                end
            end
        end
        else begin //stay IDLE
            LRU_change <= 0;
            LRU_update <= 0;
            w_ready <= 0;
            r_ready <= 0;
            state <= IDLE;
        end
    end
    WRITE_BACK: begin
```

```
69              LRU_change <= 0;
70              LRU_update <= 0;
71              if(~dw_ready) begin
72                  dw_valid <= 1;
73                  dr_valid <= 0;
74                  state <= WRITE_BACK;
75              end
76              else begin
77                  dw_valid <= 0;
78                  dr_valid <= 1;
79                  state <= FETCH;
80              end
81          end
82          FETCH: begin
83              if(~dr_ready) begin
84                  dr_valid <= 1;
85                  dw_valid <= 0;
86                  state <= FETCH;
87              end
88              else begin//写入新的tag, data, dirty, LRU, 读取完成
89                  if(valid[way_sel]) begin
90                      LRU_change <= 1;
91                      LRU_update <= 0;
92                  end
93                  else begin
94                      LRU_change <= 0;
95                      LRU_update <= 0;
96                  end
97                  dr_valid <= 0;
98                  dw_valid <= 0;
99
100                 w_data_sel <= 1;
101
102                 mem_we1 <= way_sel ? 0 : 1;
103                 mem_we2 <= way_sel ? 1 : 0;
104                 tag_we1 <= way_sel ? 0 : 1;
105                 tag_we2 <= way_sel ? 1 : 0;
106
107                 w_ready <= 1;
108                 r_ready <= 1;
109                 if (~w_valid) begin
110                     dty_clear <= 1;
111                     dty_write <= 0;
112                 end
113                 else  begin
114                     dty_clear <= 2;
115                     dty_write <= 0;
116                 end
117                 valid_write <= 1;
118                 data_from_mem <= 1;
119                 state <= FINISH;
120             end
121         end
122         FINISH: begin
123             data_from_mem <= 0;
124         w_data_sel <= 0;
125         LRU_update <= 0;
126         LRU_change <= 0;
127         dty_clear <= 0;
```

```
128             dty_write <= 0;
129             valid_write <= 0;
130             mem_we1 <= 0;
131             mem_we2 <= 0;
132             tag_we1 <= 0;
133             tag_we2 <= 0;
134             w_ready <= 0;
135             r_ready <= 0;
136                 state <= IDLE;
137             end
138             endcase
```

fsm.v

Dcache 的正确性通过专门编写的汇编程序进行了测试，这个汇编程序尝试写入数据到 index 相同地址不同的 3 块，并且再读出数据，以此来检查 Dcache 能否正确写回以及 LRU 选择是否正确。汇编程序如下：

```
1  .text
2  main:
3      addi t1, zero, 1
4      slli a0, t1, 13
5      sw t1, 0(a0)
6      addi t1, t1, 1
7      addi a0, a0, 4
8      sw t1, 0(a0)
9      addi t1, t1, 1
10     addi a0, a0, 4
11     sw t1, 0(a0)
12     addi t1, t1, 1
13     addi a0, a0, 4
14     sw t1, 0(a0)
15
16     addi a0, zero,1
17     slli a0, a0, 13
18     addi a0, a0, 0x100
19     addi a0, a0, 0x100
20     addi t1, t1, 1
21     sw t1, 0(a0)
22     addi a0, a0, 4
23
24     addi t1, t1, 1
25     sw t1, 0(a0)
26     addi a0, a0, 4
27
28     addi t1, t1, 1
29     sw t1, 0(a0)
30     addi a0, a0, 4
31
32     addi t1, t1, 1
33     sw t1, 0(a0)
34
35     addi t1, t1, 1
36     addi a0, zero, 1
37     slli a0, a0, 13
38     addi a0, a0, 0x100
39     addi a0, a0, 0x100
40     addi a0, a0, 0x100
41     addi a0, a0, 0x100
```

```
42      addi t1, t1, 1
43      sw t1, 0(a0)
44      addi a0, a0, 4
45
46      addi t1, t1, 1
47      sw t1, 0(a0)
48      addi a0, a0, 4
49
50      addi t1, t1, 1
51      sw t1, 0(a0)
52      addi a0, a0, 4
53
54      addi t1, t1, 1
55      sw t1, 0(a0)
56
57      addi a0, zero,1
58      slli a0, a0, 13
59      lw t1, 0(a0)
60      lw t1, 4(a0)
61      lw t1, 8(a0)
62      lw t1, 12(a0)
63
64
65
66
67
68  end: jal end
```

test.asm

4.2 IOU

这一部分相对 Dcache 比较简单，主要工作量在于需要实现比较繁琐的各个相应外设部件的代码。

首先对于输入，要将按钮信号去抖动；对于开关要实现数码管实时显示编辑数据，每次拨动开关输入一个 16 进制数字，按下 del 按钮删去一个数字，按下 data 按钮完成数据编辑。这一部分主要是编码的工作，开关编辑数据的核心代码如下

```
1  always@(posedge clk)
2  begin
3      if(~rstn || btnc)
4      begin
5          tmp <= 0;
6      end
7      else if(p && !btnr)
8      begin
9          tmp <= (tmp << 4) + h;
10     end
11     else if(!p && btnr)
12     begin
13         tmp <= tmp >> 4;
14     end
15 end
```

switch_data.v

每当收到开关信号产生变化时，在别的模块中会开始去边沿，并将处理完成的数据以及一个时钟周期的有效位给到 switch_data 模块中，在这里将其加入数据的最后一位。

对于 IOU 的主体部分，简单的数据将地址对应的外设寄存器读出/写入即可。

比较复杂的部分即是开关数据以及七段数码管的查询时输入输出，具体逻辑如下：当数码管准备好时，rdy 置 1；可以输出到数码管显示数据，此时 rdy 置 0（数码管忙）；用户查看完显示数据后按下按钮，rdy 再次置 1。开关数据编辑好前，swx_rdy 置 0，按下按钮后置 1，此时 CPU 可以读入开关数据，读入完成后将 swx_rdy 置 0，数据清零，等待下一次编辑。核心代码如下：

```
    always @(posedge clk) begin     //CPU输出
    if (~rstn) begin
        led_data <= 16'hFFFF;
        seg_data <= 32'h12345678;
    end
    else if (io_we) begin
        case (io_addr)
            8'h00:
                led_data <= io_dout;
            8'h0C:
                seg_data <= io_dout;
            default: ;
        endcase
    end
end


always @(posedge clk) begin
    if (~rstn)
        seg_rdy <= 1;
    else if (io_we & (io_addr == 8'h0C))
        seg_rdy <= 0;
    else if (BTNU_P || BTNL_P || BTND_P)
        seg_rdy <= 1;
end


always @(posedge clk) begin
    if (~rstn)
        swx_vld <= 0;
    else if (BTNC_P & ~swx_vld) begin
        swx_data <= tmp;
        swx_vld <= 1;
    end
    else if (io_rd & (io_addr == 8'h14))
        swx_vld <= 0;
end

always@(posedge clk) begin
    if(~rstn)
        cnt_data <= 32'h0;
    else
        cnt_data <= cnt_data + 32'h1;
end

always @(*) begin
    case (io_addr)
        8'h04:
            io_din = {{11{1'b0}}, BTNC_P, BTNU_P, BTNL_P, BTNR_P, BTND_P, sw};
        8'h08:
            io_din = {{31{1'b0}}, seg_rdy};
        8'h10:
```

```
            io_din = {{31{1'b0}}, swx_vld};
        8'h14:
            io_din = swx_data;
        8'h18:
            io_din = cnt_data;
        default:
            io_din = 32'h0;
    endcase
end
```

IOU.v

4.3 汇编测试代码

相较之前的代码只添加了 IOU 输入数据以及使用 LFSR 生成的部分，代码如下，其中 LFSR 采用 [9 5 0] 的本原多项式。

```
    loop:
    beq t0, zero, sort_wait
    sw t1, 0(t5)

    addi t2, zero, 1 #t2 -> 0x1
    slli t2, t2, 9
    and t3, t1, t2    #t3 = t1[9]
    srli t3, t3, 9    #-> [0]

    # addi t2, zero, 0x1 //t2 -> 0x1
    srli t2, t2, 3
    and t4, t1, t2    # t4 = t1[5]
    srli t4, t4, 6

    addi t2, zero, 1
    sub t0, t0, t2
    xor t3, t3, t4
    xor t3, t3, t2   # t3 = t1[9] ^ t1[5] ^ 1
    slli t3, t3, 11

    srli t1, t1, 1
    add t1, t1, t3  # t1 = {t3, t1 >> 1}

    addi t5, t5, 4
    jal loop
```

test.asm

最后的测试部分通过简单的遍历所有数据并且判断是否前一项大于后一项，如果不是则说明排序失败，否则说明排序成功。

若排序成功，将会在 LED 输出 0x40，否则输出 0x20，通过查看 LED 即可判断排序是否成功。

4.4 CPU 的修改

CPU 的部分不用修改太多，只需要将之前的 data_mem 替换为 IOU 和 Dcache 的结合体即可，根据地址选择 IOU 或是 Dcache 进行响应。

除此之外，还修改了 stall 部分，当数据读写 valid 时，若 ready 信号没有产生，则需要 stall 整个 CPU。

CPU 的修改详见 cpu.v

烧录后测试程序以及排序程序的正确性已经由助教检查完成，综合设计任务至此完成。

4.4.1 结果分析

RTL 电路：

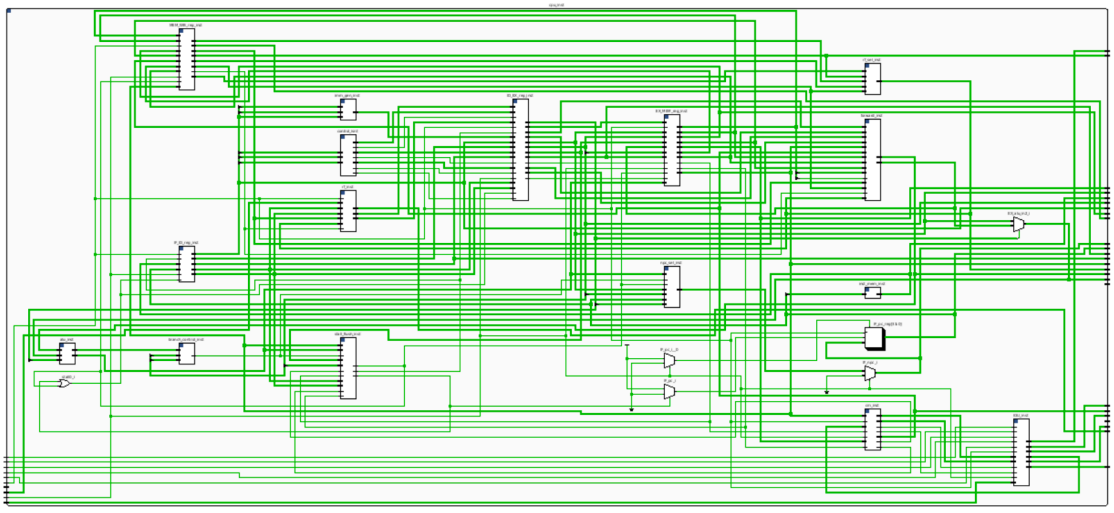

图 2: RTL

电路资源使用情况：

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
MAIN	3652	1692	359	139	56	8
cpu_inst (cpu)	3156	1430	314	132	0	0
cm_inst (cach	1060	418	260	128	0	0
Dcache_ins	1060	418	260	128	0	0
cache_ir	128	128	0	0	0	0
Dmem_ir	769	245	256	128	0	0
fsm_inst	151	11	0	0	0	0
tag_inst1	2	2	0	0	0	0
valid_ins	10	32	4	0	0	0
control_isnt (	0	7	0	0	0	0
EX_MEM_reg_i	337	168	23	4	0	0
forward_inst (	356	64	0	0	0	0
ID_EX_reg_ins	377	172	30	0	0	0
IF_ID_reg_inst	101	64	0	0	0	0
inst_mem_inst	113	0	0	0	0	0
IOU_inst (IOU)	486	277	1	0	0	0
disp_inst (c	360	45	0	0	0	0
edge_inst1	8	8	0	0	0	0
edge_inst2	6	8	0	0	0	0
edge_inst3	8	8	0	0	0	0
edge_inst4	10	8	0	0	0	0
edge_inst5	6	8	0	0	0	0
sw_inst1 (s	48	45	0	0	0	0
swd_inst (s	37	33	0	0	0	0
MEM_WB_reg_	123	164	0	0	0	0
npc_sel_inst (	33	32	0	0	0	0
rf_inst (reg fil	82	0	0	0	0	0

图 3: UTL1

电路性能:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination
Path 1	9.613	10	11	5	fd/counter_reg[2]/C	fd/counter_reg[0]/D	5.251	2.606	2.645	15.0	sys_clk_pin	sys_clk_pin
Path 2	9.613	10	11	5	fd/counter_reg[2]/C	fd/counter_reg[1]/D	5.251	2.606	2.645	15.0	sys_clk_pin	sys_clk_pin
Path 3	9.613	10	11	5	fd/counter_reg[2]/C	fd/counter_reg[3]/D	5.251	2.606	2.645	15.0	sys_clk_pin	sys_clk_pin
Path 4	9.613	10	11	5	fd/counter_reg[2]/C	fd/counter_reg[7]/D	5.251	2.606	2.645	15.0	sys_clk_pin	sys_clk_pin
Path 5	9.613	10	11	5	fd/counter_reg[2]/C	fd/counter_reg[9]/D	5.251	2.606	2.645	15.0	sys_clk_pin	sys_clk_pin
Path 6	10.807	4	5	33	fd/counter_reg[23]/C	fd/counter_reg[0]/CE	3.811	1.145	2.666	15.0	sys_clk_pin	sys_clk_pin
Path 7	10.807	4	5	33	fd/counter_reg[23]/C	fd/counter_reg[10]/CE	3.811	1.145	2.666	15.0	sys_clk_pin	sys_clk_pin
Path 8	10.807	4	5	33	fd/counter_reg[23]/C	fd/counter_reg[11]/CE	3.811	1.145	2.666	15.0	sys_clk_pin	sys_clk_pin
Path 9	10.807	4	5	33	fd/counter_reg[23]/C	fd/counter_reg[12]/CE	3.811	1.145	2.666	15.0	sys_clk_pin	sys_clk_pin
Path 10	10.807	4	5	33	fd/counter_reg[23]/C	fd/counter_reg[13]/CE	3.811	1.145	2.666	15.0	sys_clk_pin	sys_clk_pin

图 4: TIMING

5 实验总结

1. 通过本次实验，我学习到了如何依据需求修改流水线 CPU 的数据通路来增加功能，比如 cache 和 IOU
2. 通过本次实验，我理解了如何通过地址映射以及 IOU 组件使 CPU 与外设进行交互，还有查询式输入输出的实现方法

3. 通过本次实验，我学习了 Dcache 的各个模块如何具体设计，以及 FSM 的编写方法。学习到了先设计再编写的重要性

6 意见/建议

实验过于复杂，文档要求含糊不清，而且教学部分几乎为 0。希望能够改进。