中国科学技术大学计算机学院
《计算机组成原理实验报告》

实验题目：综合设计
学生姓名：林宸昊
学生学号：PB20000034
完成日期：2022.5.24

【实验题目】综合设计

【实验目的】

- 进一步理解系统的组成结构和工作原理
- 掌握软硬件综合系统的设计和调试方法

【实验环境】VIVADO FPGAOL

【实验步骤】

【CPU内部优化——添加指令存储器】

【背景】

- 在第设计五级流水线CPU实验中，处理beq及同类指令时需要两个周期的气泡阻塞，优化后也仍需要一个周期，在需要多次执行的程序中这样的气泡带来的影响将被放大（如斐波那契实验），使用instruction cache可以使流水线在执行同样指令时无需气泡直接执行，从而提高执行效率。

【cache总体结构】

- 采用课件所给出的基础cache结构

Index:PC[4:2]	Valid	Tag:PC[31:5]	Data:PC JUMP 2
000	N/Y		
001	N/Y		

Index:PC[4:2]	Valid	Tag:PC[31:5]	Data:PC JUMP 2
010	N/Y		
011	N/Y		
100	N/Y		
101	N/Y		
110	N/Y		
111	N/Y		

- 代码实现

```
reg [7:0] valid;
reg [31:0] data [0:7];
reg [26:0] tag [0:7];
```

【实现情况】

- 其一，非b类型的条件跳转，即直接跳转，则默认未命中，并将pc_in自动设置为需要跳转的pc；
- 其二，确认发生跳转，但未命中缓存，则正常执行，生成气泡；
- 其三，确认为条件跳转指令，且命中缓存，但实际并未发生跳转，则改变valid，pc_in正常递增；
- 若非以上三种情况，考察当前pc，若命中，直接利用data重置pc_in，否则递增。
- 代码实现

```
assign index = pc[4:2];
assign index2 = pce[4:2];           //from EX stage
assign hit = valid[index2] & (pce[31:5] == tag[index2]);
//需要注意，利用EX阶段pce来判断是否命中，这样可以利用此阶段产生的其他值来对pc做对应修改
//如果使用最初的pc，则会导致若失败则无法接续下去。事实上这么做没有意义，只有当执行完EX后才能通过初始pc来判断是否命中缓存并直接替换

always @(*) begin
    if(jal)                                                 //情况1
    begin
        miss = 1;
        pc_in = PCjmp2;
    end
    else if(isPCjmp && !hit)                                //情况2
    begin
        miss = 1;
        pc_in = PCjmp2;
    end
```

```
    else if(br && !isPCjmp && hit)                          //情况3
    begin
        miss =1;
        pc_in = pce + 4;
    end
    else if(valid[index] && (tag[index] == pc[31:5]))       //情况4
    begin
        pc_in = data[index];
        miss = 0;
    end
    else begin                                              //默认
        pc_in = pc + 4;
        miss = 0;
    end
end
```

【cache的写入】

- 其一，发生跳转且未命中，写入；
- 其二，读入跳转指令，命中，但并不发生跳转，更改valid。
- 代码实现

```
always @ (posedge clk, posedge rst)
    begin
        if(rst)
            valid <= 0;
        else begin
            if(isPCjmp && !hit)             //情况1
            begin
                valid[index2] <= 1;
                tag[index2] <= pce[31:5];
                data[index2] <= PCjmp2;
            end
            else if(br && !isPCjmp && hit)  //情况2
            begin
                valid[index2] <= 0;         //认为不可用
            end
        end
    end
```

【与外部交互】

- 模块例化

```
cache ins_cache(
    .clk(clk),
    .rst(rst),
```

```
    .pc(pc),
    .br(ctrl[8]),
    .isPCjmp(branch),
    .jal(ctrl[9]),
    .pce(pce),
    .PCjmp2(PCjmp2),
    .pc_in(pc_in),
    .miss(fail)
);
```

其中信号与第五次实验一致。

- 交互

```
assign branch = br & ctrl[8];   //代表实际发生了跳转
assign PCjmp2 = (imm << 1) + pce;   //此为EX stage信号
assign rs1_clr_new = rs1_clr || fail;
assign rs2_clr_new = rs2_clr || fail;   //如果未命中，则如正常情况清空IFID
段寄存器
```

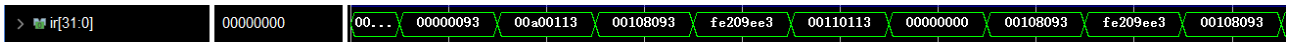
至此，一个指令cache基本完成。

【仿真测试】

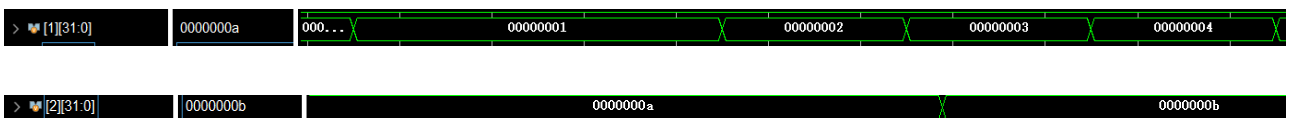
- 汇编文件

```
addi x1, x0, 0
addi x2, x0, 10
t:
addi x1, x1, 1
bne x1, x2, t
addi x2, x2, 1
```

- 仿真结果



可以看到在第一次之后ir直接做出了更改，不再产生气泡。



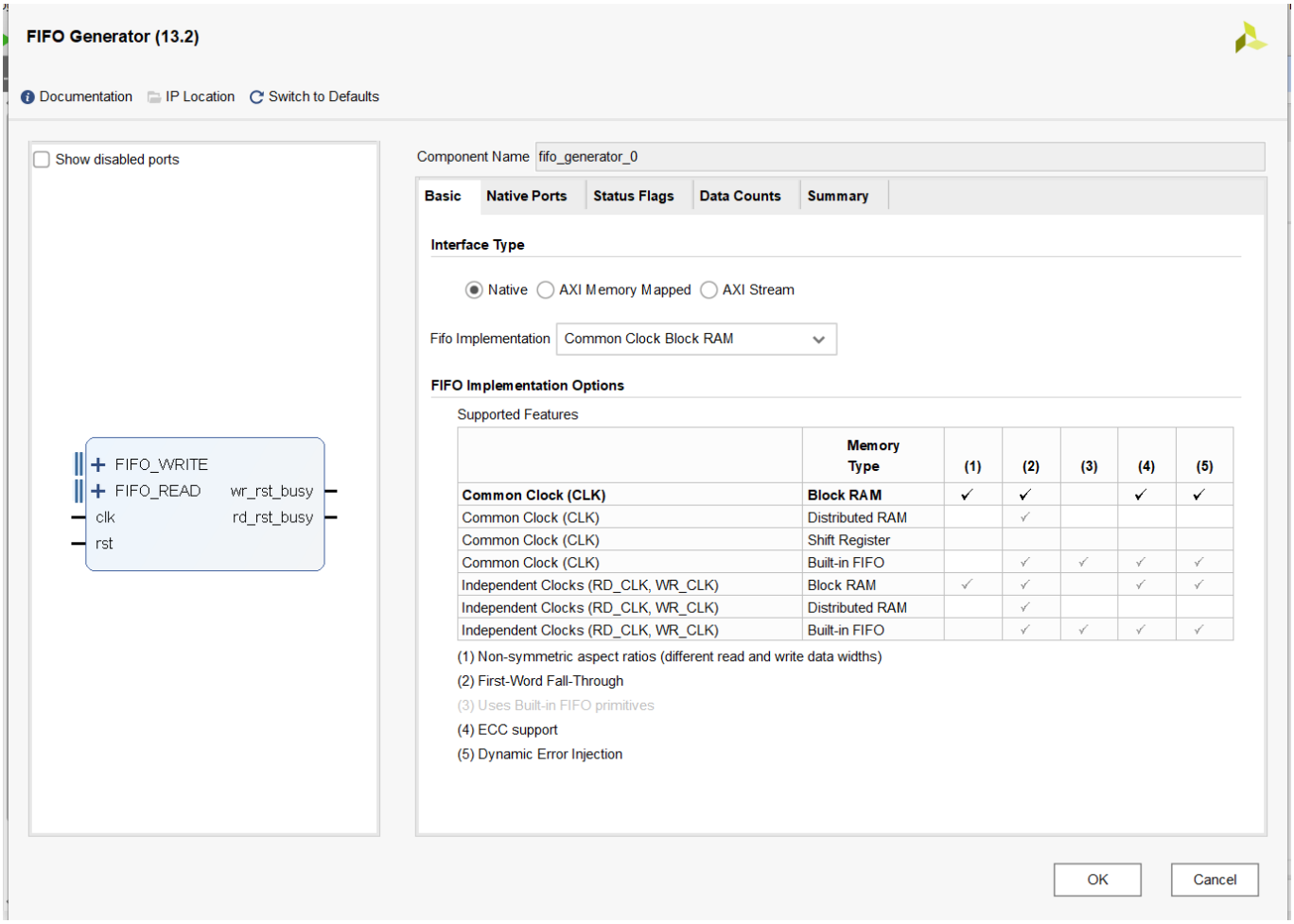
与汇编预期结果一致，cache生效。

【CPU外部优化——串口通信】

【背景】

- 藉由上学期所使用的的基本串口通信思路，将按动开关查看段寄存器改为通过串口输入序号查看，更为便捷；
- 同时增加实时更改内部寄存器值功能，实现异步写入。

【串口输入输出】



- 使用IP核FIFO，将串口的rx、tx放入先进先出队列然后依序接收或者输出：

```
fifo_generator_0      rx_fifo(
.clk                 (clk),
.rst                 (rst),
.din                 (rx_data),
.wr_en               (rx_vld),
.rd_en               (rx_fifo_en),
.dout                (rx_fifo_data),
.full                (),
.empty               (rx_fifo_empty)
);

fifo_generator_0      tx_fifo(
.clk                 (clk),
.rst                 (rst),
.din                 (tx_fifo_din),
.wr_en               (tx_fifo_wr_en),
.rd_en               (tx_rd),
```

```
.dout                (tx_data),
.full                (tx_fifo_full),
.empty               (tx_fifo_empty)
);
```

- 使用计数器与多个寄存器来逐个获取FIFO内的内容

```
always@(posedge clk or posedge rst)
begin
    if(rst)
        rx_byte_cnt <= 4'h0;
    else if(curr_state==C_CMD_DC)
    begin
        if((rx_fifo_en)&&(rx_fifo_empty==1'b0)&&(rx_byte_cnt<4'hf))
            rx_byte_cnt <= rx_byte_cnt + 4'b1;
    end
    else
        rx_byte_cnt <= 4'h0;
end
……

case(rx_byte_cnt)        //tx输出同理
            4'h0:   rx_byte_buff_0 <= rx_fifo_data;
            4'h1:   rx_byte_buff_1 <= rx_fifo_data;
            4'h2:   rx_byte_buff_2 <= rx_fifo_data;
            4'h3:   rx_byte_buff_3 <= rx_fifo_data;
            4'h4:   rx_byte_buff_4 <= rx_fifo_data;
            4'h5:   rx_byte_buff_5 <= rx_fifo_data;
            4'h6:   rx_byte_buff_6 <= rx_fifo_data;
            4'h7:   rx_byte_buff_7 <= rx_fifo_data;
endcase
```

由于总体代码实现较复杂，此处仅简述思路。

【段寄存器查看】

- 实现思路

  将串口输入赋值给一个地址寄存器，根据这个地址寄存器赋予回显所用的数据寄存器相应的段寄存器的值。

```
case(rd_addr[7:0])
    8'h0:   rd_data = pc;
    8'h01:  rd_data = pcd;
    8'h02:  rd_data = ir;
    8'h03:  rd_data = pcin;
    8'h08:  rd_data = pce;
    8'h09:  rd_data = a;
```

```
    ......
endcase
```

【registers file查看】

- 实现思路

  以五位输入作为地址，前面加上“r”，即以“r0000"格式作为输入，通过辨认串口所接收到的第一个字符来判别查看段寄存器还是数据寄存器。

```
if(rx_byte_buff_1[1])                          //check registers file
        begin
        rd_addr[5] <= rx_byte_buff_1[1];
        rd_addr[4] <= rx_byte_buff_2[0];
        rd_addr[3] <= rx_byte_buff_3[0];
        rd_addr[2] <= rx_byte_buff_4[0];
        rd_addr[1] <= rx_byte_buff_5[0];
        rd_addr[0] <= rx_byte_buff_6[0];
        end
        else begin                             //check stage registers
        rd_addr[4] <= rx_byte_buff_1[0];
        rd_addr[3] <= rx_byte_buff_2[0];
        rd_addr[2] <= rx_byte_buff_3[0];
        rd_addr[1] <= rx_byte_buff_4[0];
        rd_addr[0] <= rx_byte_buff_5[0];
        flag <= 0;
        end
```

- 为能实时查看寄存器内部的值，将所产生的地址作为输出连接至寄存器堆，以获取对应地址的输出。

```
 if(rd_addr[5])
        begin
        rd_data = rdd;
        end
assign rdd = registers[rd_data];
```

【实时写入】

- 同样，增加一个写入状态来处理写入命令。

```
assign  is_wb_cmd = (curr_state==C_CMD_DC)
                    &&(rx_byte_buff_1=="w")
                    &&(rx_byte_buff_2==" ")&&(rx_byte_buff_5==" ")
//输入格式为"w" + " " + 二位十进制地址 + " " + 二位十六进制数据
......
else if(curr_state == C_CMD_WB)
```

```
    begin
            wr_en   <= 1'b1;
            wr_addr[7:4] <= rx_byte_buff_3[3:0];
            wr_addr[3:0] <= rx_byte_buff_4[3:0];
            wr_data[7:4] <= rx_byte_buff_6[3:0];
            wr_data[3:0] <= rx_byte_buff_7[3:0];
    end
```

- 为实现写入，同样需要将地址与寄存器堆连接

```
if(wr_en && wr_addr != 0)
        begin
        regfile[wr_addr[7:4] * 10 + wr_addr[3:0]] <= wr_data;
        end
```

需要注意的是，在实际例化中，需要传入独立的时钟信号而非PDU所产生的时钟信号，以做到异步输入。

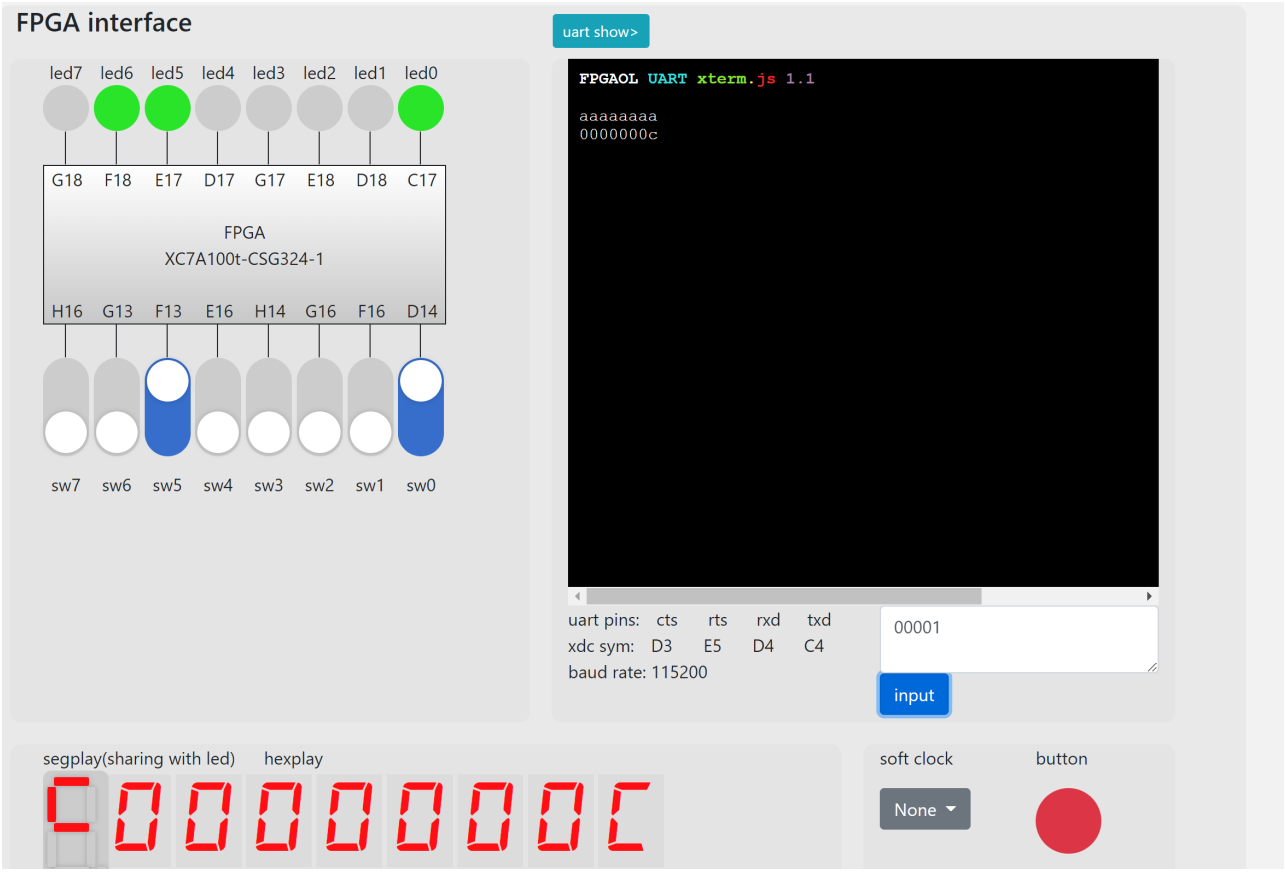
```
always @ (posedge clk_0) //直接与FPGA时钟信号相连
    if(wr_en && wr_addr != 0)
        begin
        regfile[wr_addr[7:4] * 10 + wr_addr[3:0]] <= wr_data;
        end
```

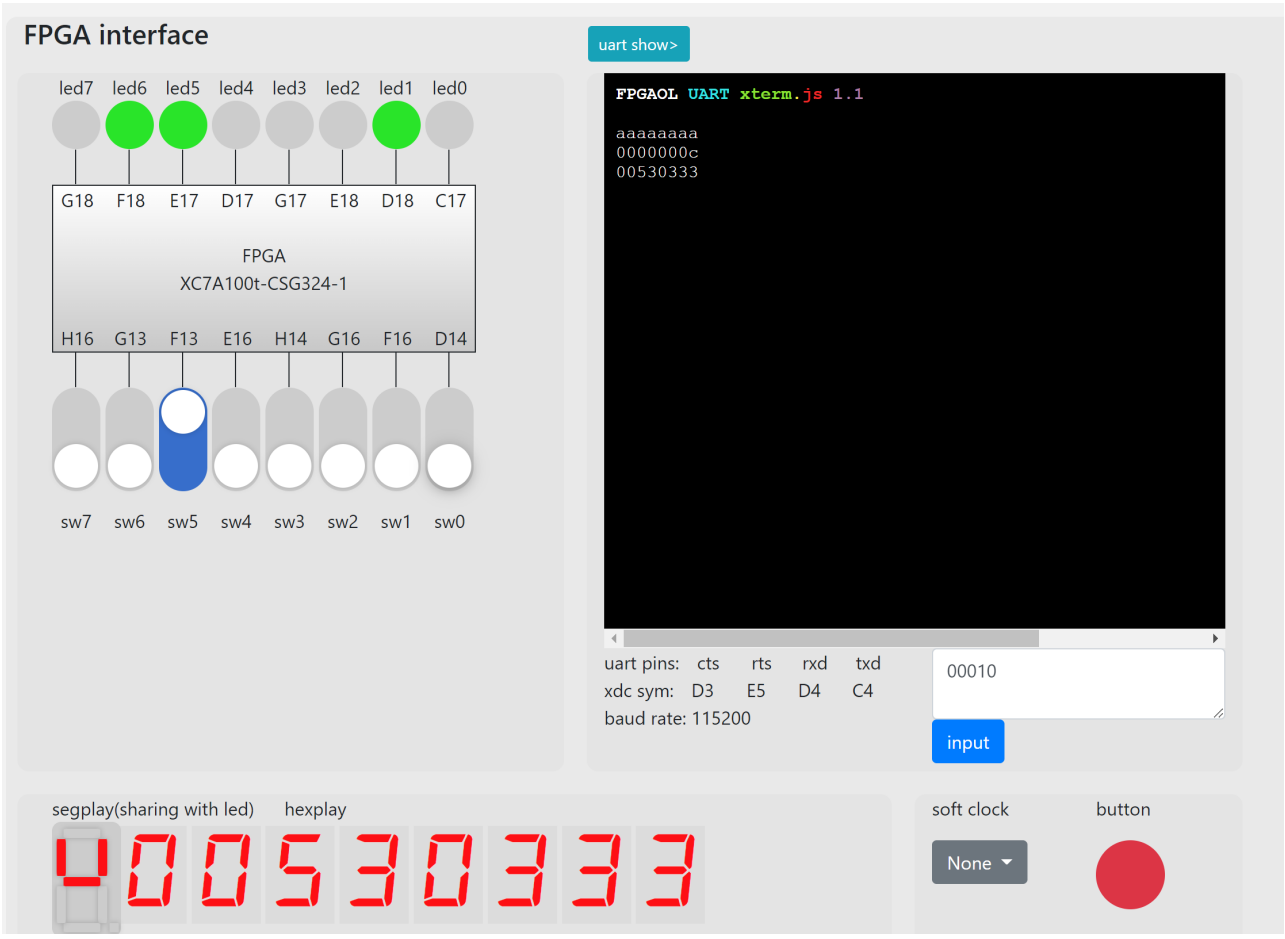
至此，实现了串口的通信。

【写入至FPGA测试】

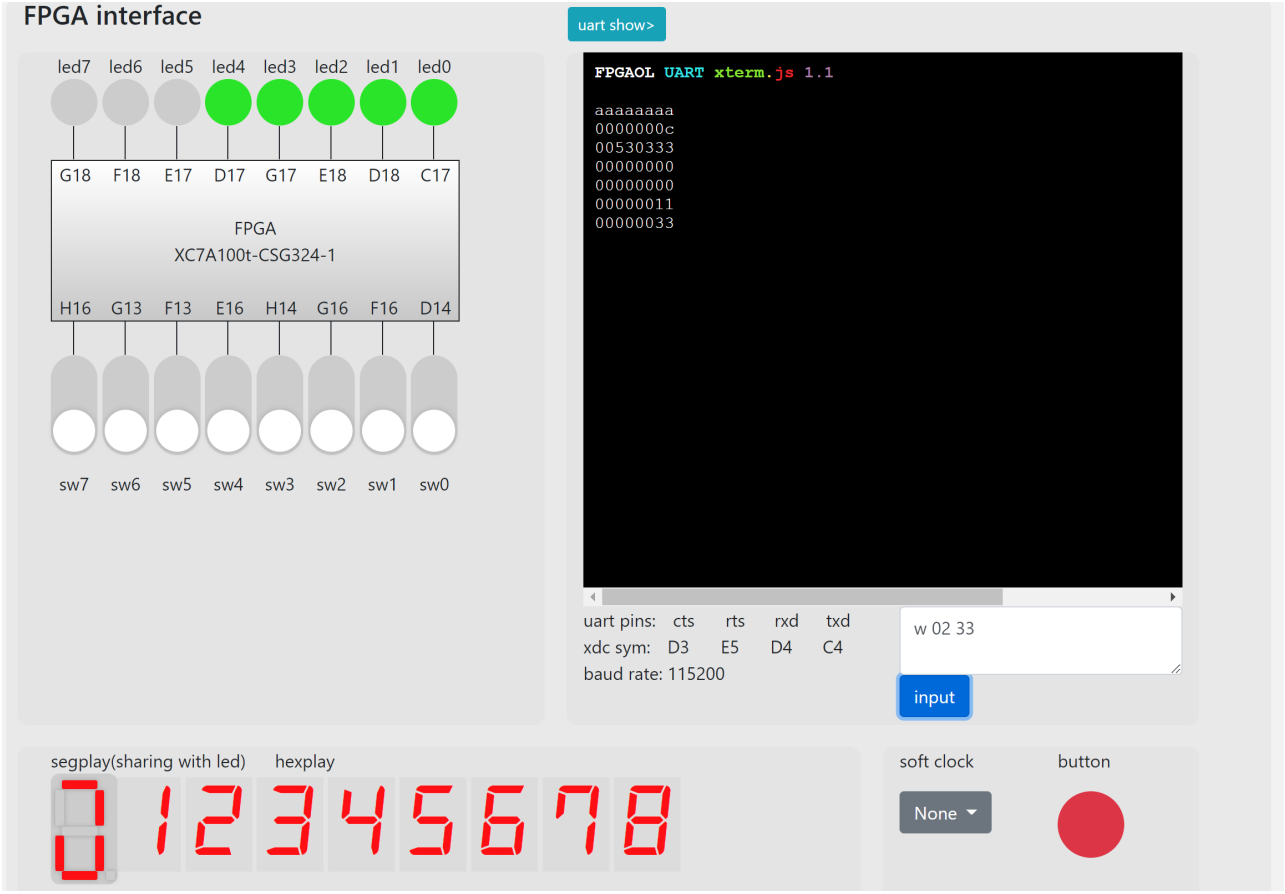
使用仿真时所用的数据文件。

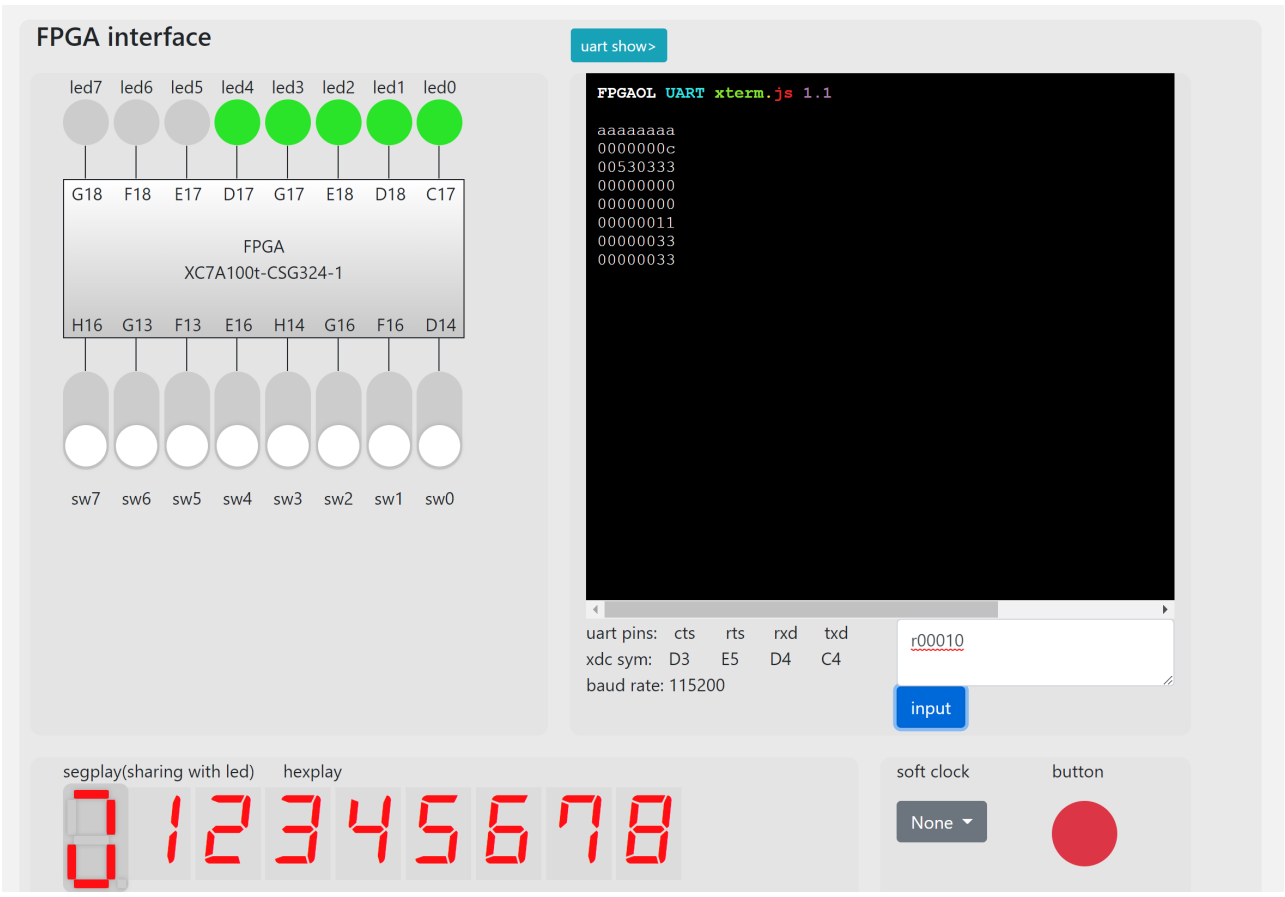
- 查看段寄存器


FPGA interface
uart show>
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
FPGAOL UART xterm.js 1.1
aaaaaaaa
0000000c
uart pins: cts rts rxd txd
xdc sym: D3 E5 D4 C4
baud rate: 115200
00001
input
segplay(sharing with led)
hexplay
soft clock
button
None


FPGA interface
uart show>
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
FPGAOL UART xterm.js 1.1
aaaaaaaa
0000000c
00530333
uart pins: cts rts rxd txd
xdc sym: D3 E5 D4 C4
baud rate: 115200
00010
input
segplay(sharing with led)
hexplay
soft clock
button
None

- 写入寄存器并查看





与预期结果一致，串口通信实现。

【总结与思考】

- 实验总结

- cache的实现总体难点在于选择哪一个阶段的信号判断是否命中，以及缓存读取情况的分类和处理；
- 串口的实现总体难点在于输入的读取与回显的输出，以及对应输入的处理，早期文件不适配版本之后重新用fifo重新写让人找回上学期的恐惧，何况上学期还只做了读取没做回显，很可怕；
- 其实单独的一个指令cache实现并不难，但是事实上2bit分支预测能做到的cache未必不能做到，但可能实现上更简单一些；因此除此之外还增添了串口通信来做进一步完善；
- 最后，恭喜助教解脱下班，即刻卸载vivado（