计算机组成原理实验 实验报告

实验题目：Lab6 综合设计

学生姓名：王章瀚

学生学号：PB18111697

完成日期：2020 年 6 月 25 日

计算机实验教学中心制

2019年09月

1 实验题目

Lab6 综合设计

2 实验目的

1. 理解计算机系统的组成结构和工作原理;
2. 理解计算机总线和接口的结构和功能;
3. 掌握软硬件综合系统的设计和调试方法.QQQ

3 实验平台

Vivado

4 实验设计内容说明

本次实验要求实现一个简单的计算机应用系统. 但其实要做好这么一个简易系统, 主要是要实现总线. **只要实现了总线, 其他更为复杂的操作, 就变成了搭积木一样简单了**. 因此认为, 讲义给出的斐波那契数列应用, 足以反映我对于总线的认识与实现它的能力. 因此就按照老师给出的应用来完成本次实验.

本次设计中, 采用了Lab5中完成的流水线CPU来做. 除了添加了总线设计以外, 还添加了分支预测, 便于日后实验其他功能的时候使用. 总体而言, 是设计了一个**带分支预测和总线的流水线CPU**.

5 实验过程

5.1 基本过程

5.1.1 总线的设计

总线的设计中, 主要需要有一个总线控制器来控制数据传输. 而相关的总线大致需要有:

1. din: CPU传到总线上的数据线
2. dout: 总线传给CPU的数据线
3. done: 总线上din放好数据的信号
4. we: 设备写使能
5. re: 设备读使能

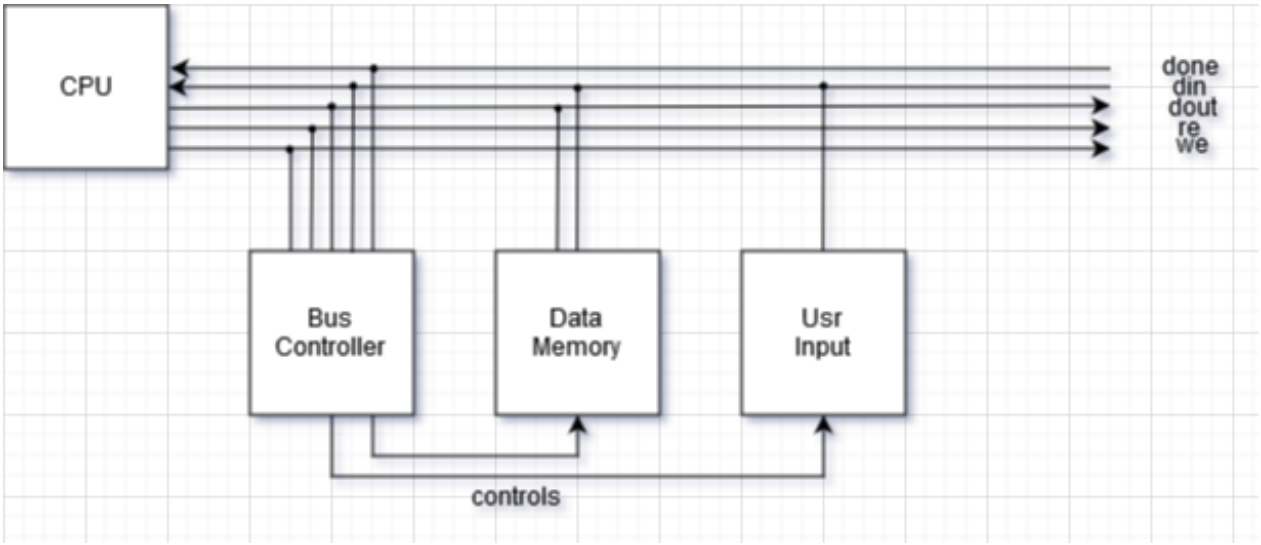
5.1.2 分支预测的设计

这里分支预测采用的是2位动态分支预测, 这种预测方法足以适配大多数情况的分支预测, 实现CPU的高效性.

5.2 数据通路

分支预测的数据通路很简单, 就是添加了一个分支预测模块及预测是否分支的控制线等, 故这里就不展示了.

至于总线的数据通路和老师的没有太大区别, 但我用了自己的变量名称, 方便助教查看程序的时候能够理解.

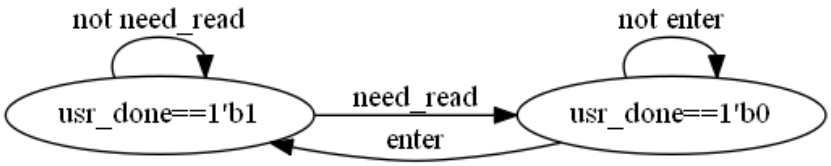


5.3 状态转换

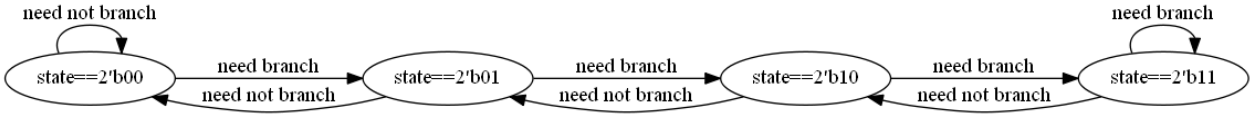
5.3.1 用户输入器的状态转换

总线控制这块没有什么太多的状态可言. 所以就简单描述一下用户Input模块的状态.

下面这个图描述的是需要lw到用户输入的时候, 就会阻塞等待用户输入, 知道用户按了enter(某个按钮), 才会继续.



5.3.2 分支预测器的状态转换



5.4 代码讲解

5.4.1 总线的代码讲解

这里面用了两个外设, 一个是数据存储器, 另一个是自己实现的用户输入器(获取由switches表示的数据, 按某个按钮确认输入), 分别作为外设0和外设1. 这里主要需要控制的是,

1. 当对应外设准备好数据的时候, done就会被置为1'b1
2. 当对应外设未被选中, 应当作为32'hzzzz_zzzz
3. 当对应外设被选中, 将其输出作为总线输出(即din)

```
module bus_unit
(
input clk,
input rst,
input [31:0] addr,
input [31:0] dout,        // CPU输出到总线的数据
input we,
input re,                 // 读使能
// input of usr_input
input enter,
input [15:0] sw,
// ouput
output [31:0] din,        // 总线要给CPU的数据
output reg done,              // done标志
input [31:0] DBU_mem_rf_addr,
output [31:0] DBU_mem_data
);

// 片选使能 addr[31:10] == en[21:0]
// 片内地址 addr[9:2]

wire [31:0] data_0;
wire [31:10] en;

assign en = addr[31:10];

// 外设0: 数据存储器
data_mem_256x32 data_mem(.a(addr >> 2),
                         .d(dout),
                         .dpra(DBU_mem_rf_addr),
                         .clk(clk),
                         .we(we),
                         .spo(data_0),
                         .dpo(DBU_mem_data));
assign din = en == 22'h0 ? data_0 : 32'hzzzz_zzzz;

// 外设1: 用户输入读取
wire [31:0] data_1;
wire usr_done;
reg need_usr_input;
wire flag;
assign flag =usr_done && need_usr_input;

usr_input usr_input(clk, rst,
                    addr[3:2],
                    re,
                    enter,
                    sw,
```

```
                    data_1,
                    usr_done);
assign din = en == 22'h1 ? data_1 : 32'hzzzz_zzzz;


always @(*) begin
    case(en)
        22'h0: begin
            done = 1'b1;
        end
        22'h1: begin
            done = usr_done;
        end
        default: done = 1'b1;
    endcase
end

endmodule
```

5.4.2 用户输入器的代码讲解

这里输入信号需要有sw和enter(作为确认输入)
主要实现的内容有:

1. 对enter取边沿, 以防连续输入.
2. 在需要用户输入的时候(need_usr_input==1'b1), 若未摁enter, 则置done为1'b0
3. 若用户摁了enter, 则置done为1'b1, 表示完成用户输入的读入.

```
module usr_input
(
input clk,
input rst,
input [1:0] addr,
input need_usr_input,
input enter,
input [15:0] sw,
output reg [31:0] data,
output reg done
);

wire enter_edge;

edge_taker #(.N(1))
    input_edge_taker(.clk(clk),
                     .rst(rst),
                     .in(enter),
                     .out(enter_edge));

always @(*) begin
    case(addr)
        2'b00: data = {{24{1'b0}}, sw[7:0]};
        2'b01: data = {{24{1'b0}}, sw[15:8]};
        2'b10: data = 32'hffff_ffff;
        default: data = 32'h0000_0000;
    endcase
```

```
end

always @(*) begin
    if(rst) begin
        done = 1'b1;
    end
    else begin
        if(need_usr_input) done = 1'b0;
        else done = done;
        if(enter_edge) begin
            done = 1'b1;
        end
    end
end

endmodule
```

5.4.3 分支预测器的代码讲解

这一部分的代码实现了一个简单的2位动态分支预测器. 代码内容主要是对分支成功与失败的时候, 作相应位的增减. 并且给出shall_branch来告诉CPU是否预测为要分支.

```
module branch_predictor
#(
parameter HIGH = 5      // 分支预测器cache地址最高位
)
(
input clk,
input rst,
input [5:0] im_instr_opcode,
input [HIGH:0] im_instr_lowpc,
input [HIGH:0] IF_ID_lowpc,
input IF_ID_is_branch,
input equal,
output shall_branch
);

localparam BEQ_op = 6'b000100;
localparam CACHE_SIZE = {2'b10, {HIGH{1'b0}}}; // 根据 HIGH 判断缓存大小
reg [1:0] cache[CACHE_SIZE-1:0];
integer i;

assign shall_branch = cache[im_instr_lowpc] >= 2'b10 && im_instr_opcode == BEQ_op ? 1'b1 : 1'b0;

always @(posedge clk, posedge rst) begin
    if(rst) begin
        for(i = 0; i < CACHE_SIZE; i = i + 1) begin
            cache[i] <= 2'b00;
        end
    end
    else if(IF_ID_is_branch) begin
        case(equal)
            1'b0: cache[IF_ID_lowpc] <= cache[IF_ID_lowpc] == 2'b00 ? 2'b00 : cache[IF_ID_lowpc] -
     2'b01;
            1'b1: cache[IF_ID_lowpc] <= cache[IF_ID_lowpc] == 2'b11 ? 2'b11 : cache[IF_ID_lowpc] +
     2'b01;
            default: cache[IF_ID_lowpc] <= 2'b01;
        endcase
    end
```

```
36 end
37
38 endmodule
```

6 实验结果

实验结果分两部分说明, 一部分是总线这一块, 另一部分是分支预测器这一块.

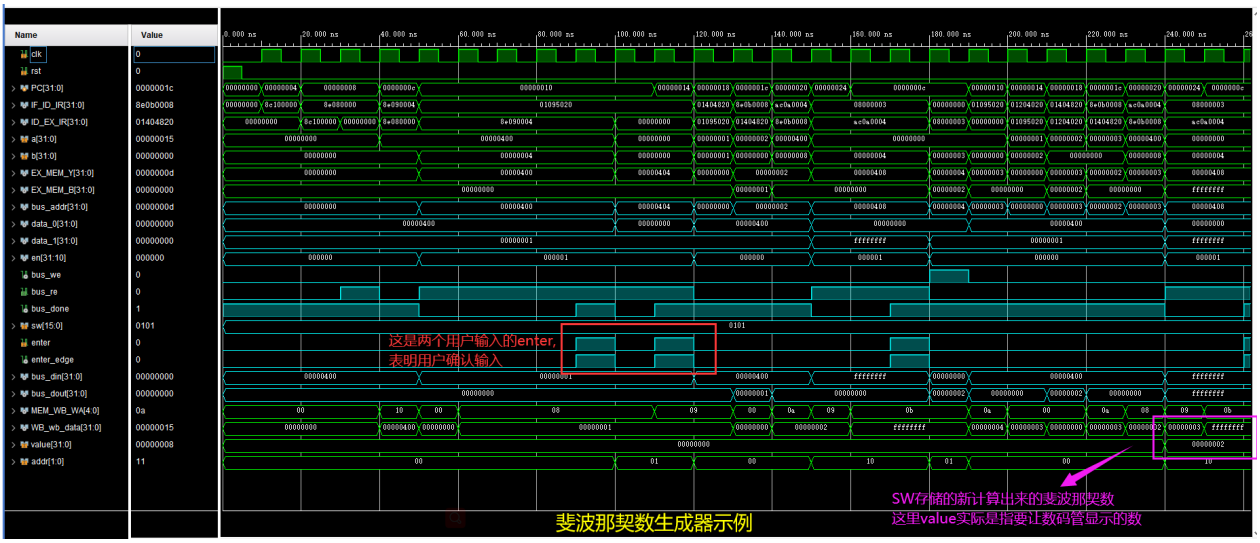
6.1 总线的实验结果

这一部分的实验验证, 采用了一个可以获取用户输入初始两个数的斐波那契数生成器, 并且在用户摁下按钮的时候, 才显示下一个数字(按照代码, 应该会显示在数码管上, 然而没有板子去验证)

6.1.1 起始输入及第一个数计算

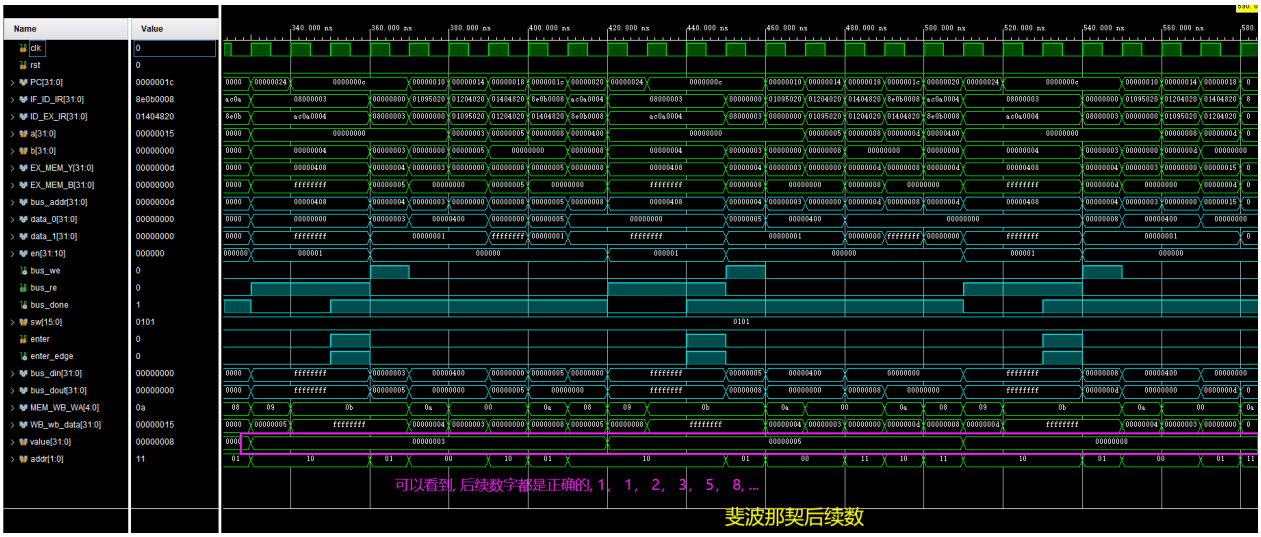
可以看到下图,

1. 仿真代码中将SW设置成了输入是两个1, 因此斐波那契初始值为1, 1
2. 蓝色波形都是关于总线的波形, 标识出来方便助教查看
3. 红色框是两个用户的enter, 标识用户确认输入的两个数字
4. 红色框同一行后面还有一个enter, 标识用户确认显示下一个斐波那契数
5. 紫色框是斐波那契数的sw以及在数码管将要显示的数字, 可以看到取得了1+1=2



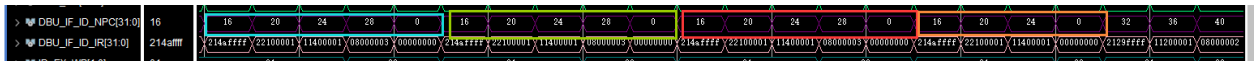
6.1.2 后续计算及显示

再去看后面的数字, 也都在相应的enter后, 显示正确数字



6.2 分支预测器的实验结果

之前的代码都表现不出来分支预测有无的区别, 下面用一个能表现出区别的示例代码(见附录9.2[1])来展现.



可以看到图中原来一个循环中IF.ID_NPC是要经历16,20,24,28,0的循环, 过了几次后, 变成16,20,24,0了.

反过来也是可以的, 但是限于篇幅就不展示了, 助教如果感兴趣可以自己运行一下我的代码(附件里都有)

7 心得体会

有了总线的设计了, 这个系列实验下来才算比较完整. 这套实验让我学会了如何从0去搭建一个CPU, 及其配套总线等等, 收获非常大.

8 意见建议

没有什么意见建议.

[1] 特别感谢黄致远同学提供分支预测的测试代码

9 附录

9.1 附录1, 总线测试代码

```
_start:
    lw  $s0, 0($zero)          # 8c100000
    lw  $t0, 0($s0)            # 8e080000  Read the usr's first input
    lw  $t1, 4($s0)            # 8e090004  Read the usr's second input
_loop:
    add $t2, $t0, $t1          # 01095020  Compute the next fibonacci number
    add $t0, $t1, $zero        # 01204020  $t0 <- $t1
    add $t1, $t2, $zero        # 01404820  $t1 <- $t2
    lw  $t3, 8($s0)            # 8e0b0008  Get usr's confirmation to continue
    sw  $t2, 4($zero)          # ac0a0004  Store the fibonacci number into memory at 0x00000004
    j   _loop                  # 08000003  Jump back to _loop

```

9.2 附录2, 分支预测器测试代码

```
#calculate 4*5
#save to &=$s1
_start:
        addi $t0,$0,4           # t0=4   0                  20080004
        addi $t1,$0,5           # t1=5  4                   20090005
_loop2:
        add $t2,$t0,$0          # mov $t0 to $t2    8       01005020
_loop1:
        addi $t2,$t2,-1         # t2 = t2 - 1    12         214affff
        addi $s0,$s0,1          # s0 = s0 + 1   16          22100001
        beq $t2,$zero,_loop1out #       20                11400001
        j _loop1                #       24                08000003
_loop1out:
        addi $t1,$t1,-1         # t1 = t1 - 1   28          2129ffff
        beq  $t1,$zero,_finish  #       32                11200001

        j _loop2                #       36                08000002
_finish:
        sw   $s0,8($0)          # sw to 0x8     40          ac0a0008
        j   _finish             #       44                0800000a

```