1-运算器及其应用 实验报告

PB20000296 郑滕飞

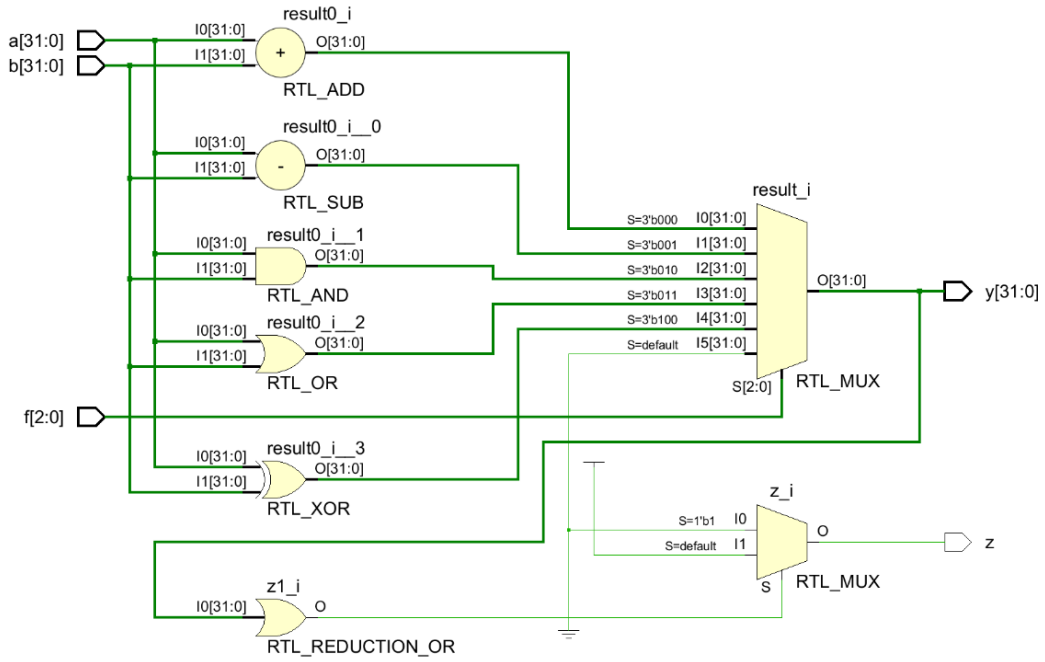
ALU 模块：

1、32 位 ALU

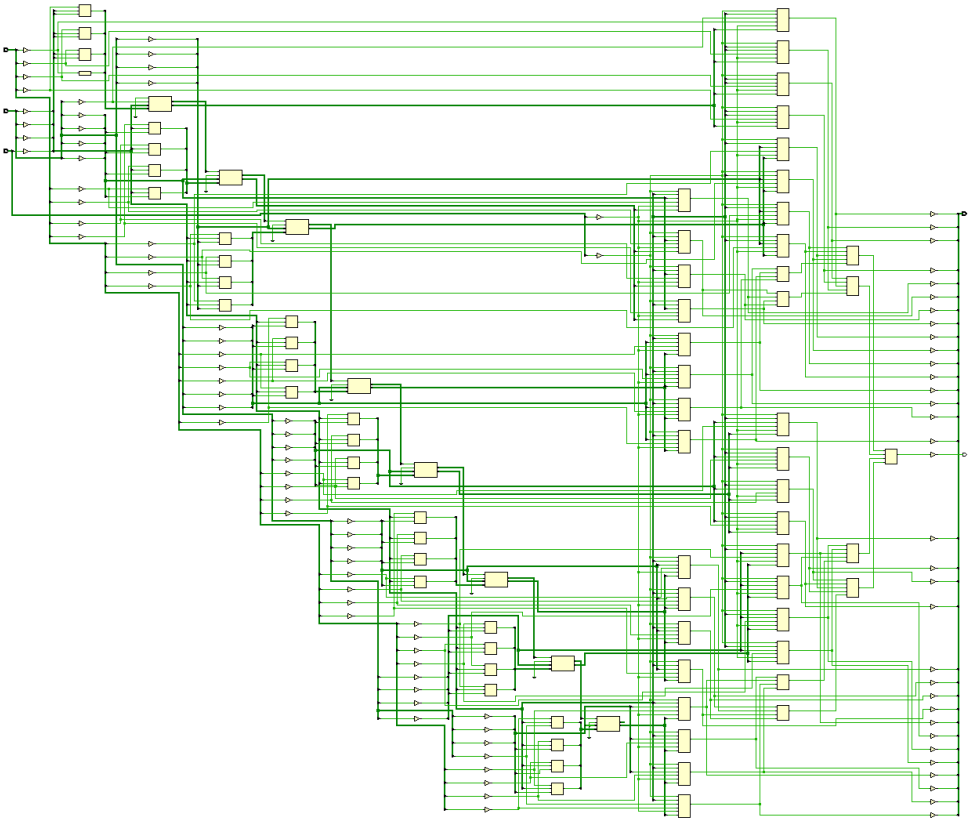
由于 ALU 的输入中并没有时钟信号，其构成组合逻辑电路。利用 result 寄存器存储计算的结果，根据 f 的取值决定选择哪个计算结果作为最终值，并检测结果是否为 0，从而调整 z 的取值：

```
always @(*) begin
    case(f)
        0: result = a + b;
        1: result = a - b;
        2: result = a & b;
        3: result = a | b;
        4: result = a ^ b;
        default: result = 0;
    endcase
end
assign y = result;
assign z = y?0:1;
```

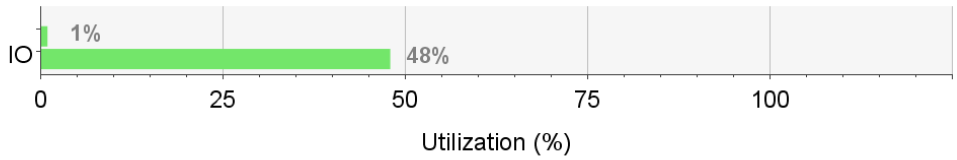
生成的 RTL 电路图如下：



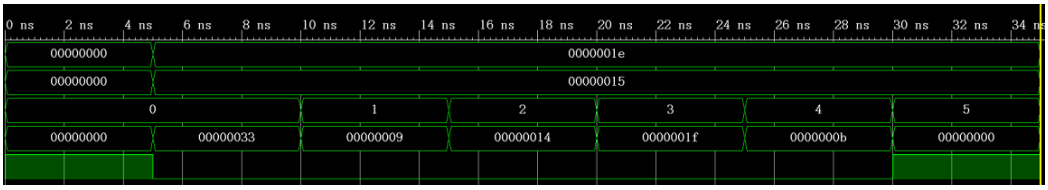
以下为综合电路图与资源利用报告(由于此电路为组合逻辑电路，不存在时钟信号 clk，时间性能报告中并没有内容)：



Resource	Utilization	Available	Utilization %
LUT	73	63400	0.12
IO	100	210	47.62



模拟测试结果（从上至下分别为 a,b,d,y,z）：



模拟测试结果通过，由此进入下一步，6 位 ALU 的设计。

2、分时复用

为完成 6 位 ALU，需先利用 en 与 sel 达成分时复用。构造 f,a,b 三个寄存器后，通过 sel 确定 x 存入哪个寄存器：

```
always @(posedge clk) begin
    if (en) begin
        case (sel)
            0: f <= x[2:0];
            1: a <= x;
            2: b <= x;
        endcase
    end
end
```

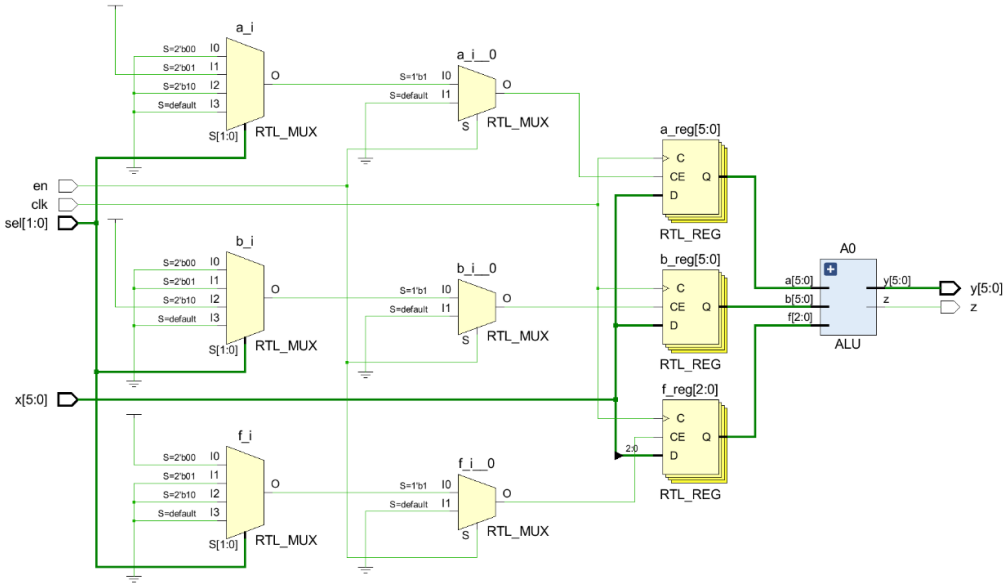
此处的 sel 与 en 是手动操作的，若 sel 利用 en 与时钟信号形成计数器，就构成了分时复用。

分时复用部分后，只需例化上方的 ALU 模块，并将参数设置为 6，即可完成设计：

```
ALU #(6) A0 (.a(a), .b(b), .f(f), .y(y), .z(z));
```

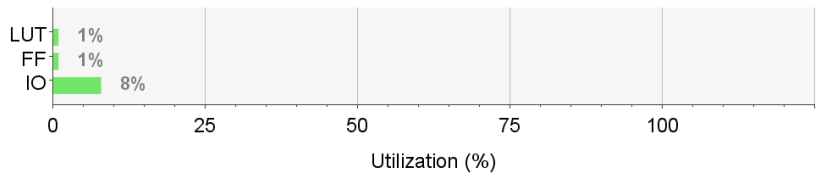
3、最终效果

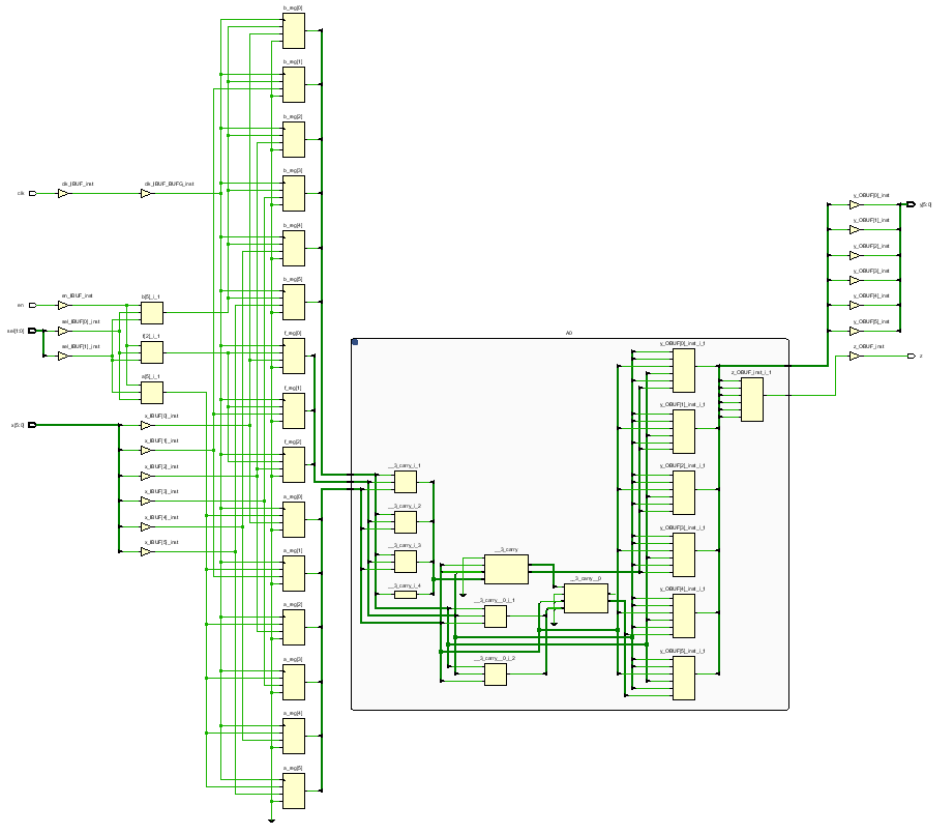
RTL 电路图：



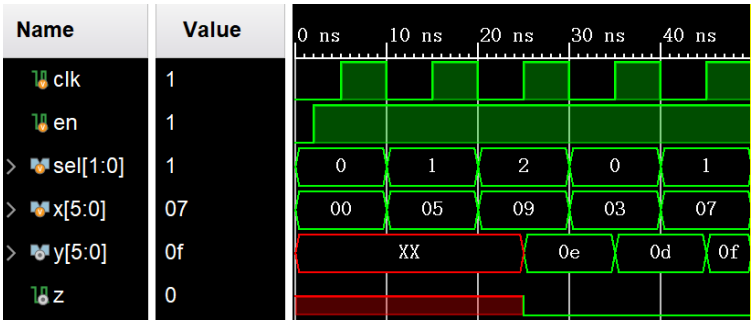
资源利用率、综合电路：

Resource	Utilization	Available	Utilization %
LUT	16	63400	0.03
FF	15	126800	0.01
IO	17	210	8.10

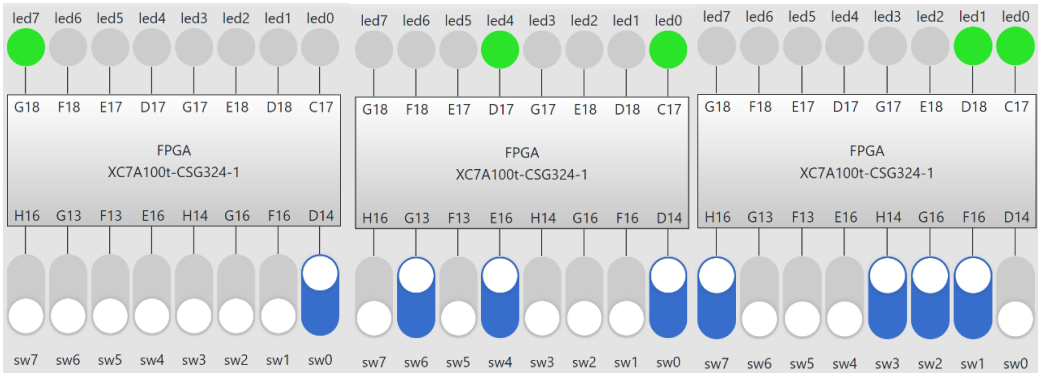




模拟测试结果(演示为 5+9=14,5|9=13,7|9=15):



FPGA 效果(截图中为演示 17-14=3):



三张图分别为：将 f 设置为 1，即减法，由于 a 与 b 默认为 0 显示结果为 0；将 a 设置为 17，此时由于 b 为 0 显示结果为 17；将 b 设置为 14，显示结果为 3。

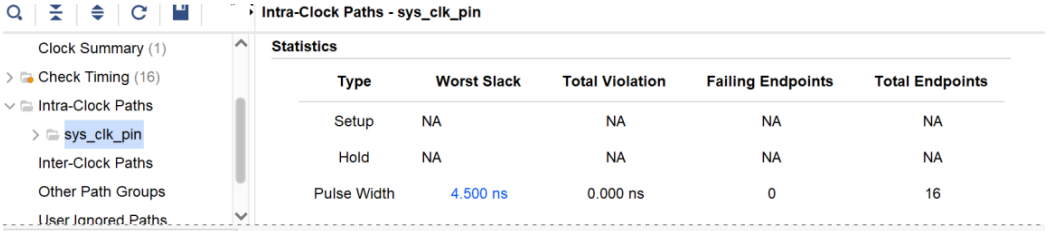
4、时间性能测试相关问题与解决

虽然已经通过了模拟测试与下载测试，却发现无法看到时间性能报告，没有检测到 Intra-Clock Paths。

经过助教提醒，在 xdc 文件中加入了一句：

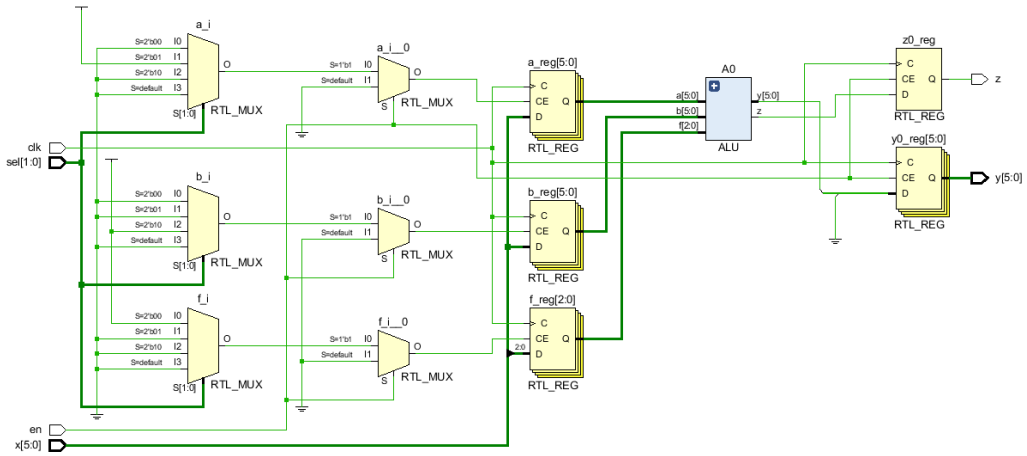
```
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk }];
```

在加入这句后，可以显示 Intra-Clock Paths 了，但仍然没有出现具体路径：



再次对比 ppt 中的设计图，发现虽然在输入处利用 reg 储存了 f,a,b，在输出处并没有利用 reg，而是 y,z 直接与之的组合逻辑电路 ALU 连接。虽然结果并没有问题，但输出处没有连接时钟导致没有出现时钟路径。

在出口处连接寄存器后，RTL 电路如下：



综合后再次查看时间性能分析，可以正常查看路径与延迟：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.063 ns	Worst Hold Slack (WHS):	0.213 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2	Total Number of Endpoints:	2	Total Number of Endpoints:	18

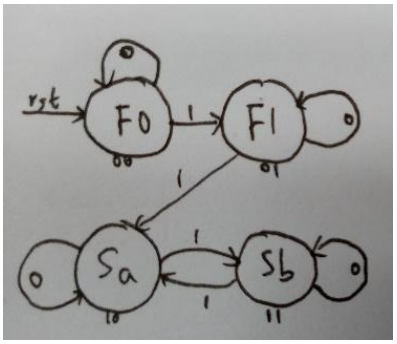
Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	6.063	4	5	2	b_reg[1]/C	z0_reg/D	3.801	1.847	1.954
Path 2	7.744	2	3	2	a_reg[0]/C	y0_reg[0]/D	2.120	1.464	0.656

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 3	0.213	1	2	6	f_reg[1]/C	y0_reg[0]/D	0.457	0.245	0.212
Path 4	0.434	2	3	6	f_reg[1]/C	z0_reg/D	0.678	0.290	0.388

FLS：

1、逻辑设计

为完成数列的表示，采用 a,b 两个寄存器记录项，状态机共有四个状态：F0 表示输入第一个数(并存入寄存器 a)，F1 表示输入第二个数(并存入寄存器 b)，Sa 表示计算当前 a,b 中两数之和并存入 a，Sb 表示计算当前 a,b 两数之和并存入 b。实际工作中，在 rst 后，先输入数列第一项，再输入第二项，然后在 Sa,Sb 两状态中循环。



这个设计的目的在于，通过 cs[1]可以确定结果是两数之和还是直接从输入中获取，而通过 cs[0]可以确定结果存入寄存器 a 还是 b。有了逻辑设计后，即可以构建状态机对应的电路。

2、电路构建

利用 F0,F1,Sa,Sb 分别表示 0,1,2,3 后，状态机的电路可以写成：

```
always @(posedge clk) begin
    if (rst) cs <= F0;
    else cs <= ns;
end

always @(*) begin
    if (en)
       case (cs)
            F0: ns = F1;
            F1: ns = Sa;
            Sa: ns = Sb;
            Sb: ns = Sa;
        endcase
    else ns = cs;
end
```

由于直接使用了 cs[0]与 cs[1]的信号，状态机的第三部分直接与此后的电路合并，成为（代码中 a,b,result 均为寄存器）：

```
assign re = cs[1] ? a+b : d;
assign f = result;

always @(posedge clk)
    if(rst) result <= 0;
```

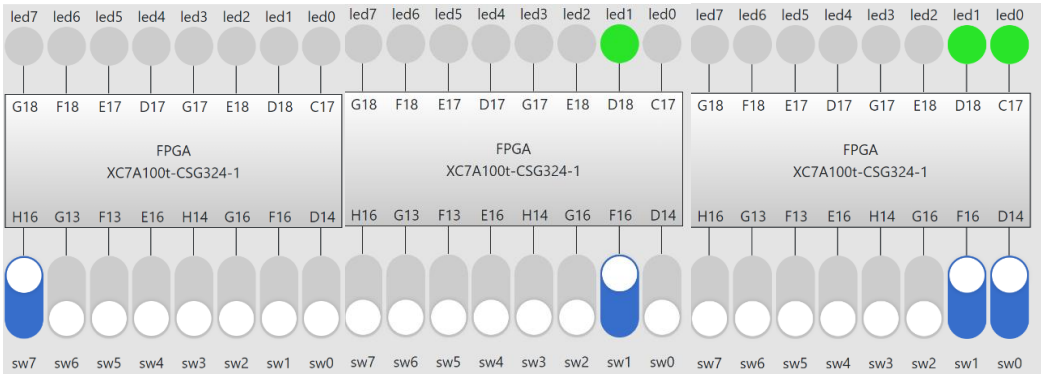
```
    else if(en) begin
            result <= re;
            if (cs[0]) b <= re;
            else a <= re;
    end
```

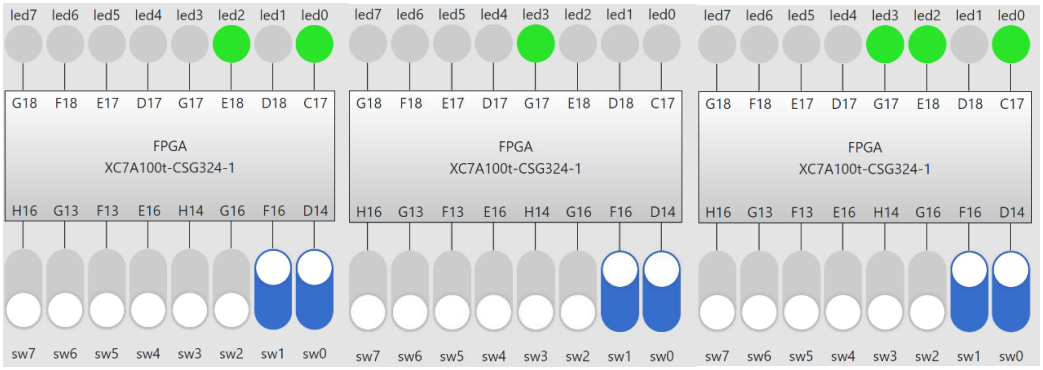
此处吸取了之前无法显示时间性能报告的教训，将结果先用寄存器存放再赋值给 f，从而可以查看数据通路的时间性能。

这个电路通过了仿真测试，但在烧写到 FPGA 时，总是无法得到正确的结果。经过对电路的检查，发现是由于实际运行时，按动一次按钮经历了很多周期，导致结果无法控制，由此，上方的 en 应替换为 edg，一个用来记录 en 上升沿的信号，具体代码为：

```
module en_edge(
    input clk,
    input en,
    output edg
);
    reg en_1 = 0,en_2 = 0;
    always @(posedge clk) begin
        en_1 <= en;
        en_2 <= en_1;
    end
    assign edg = en_1 & ~en_2;
endmodule
```

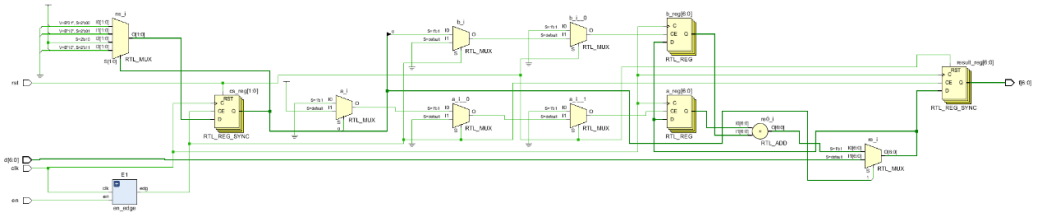
利用 edg 来决定何时进入下一状态后(注：这里赋寄存器初值 0 是为了方便模拟)，烧写到 FPGA 即可获得正常结果(下图为 reset 后输入 2,3, 电路继续计算出 5,8,13 的过程)：



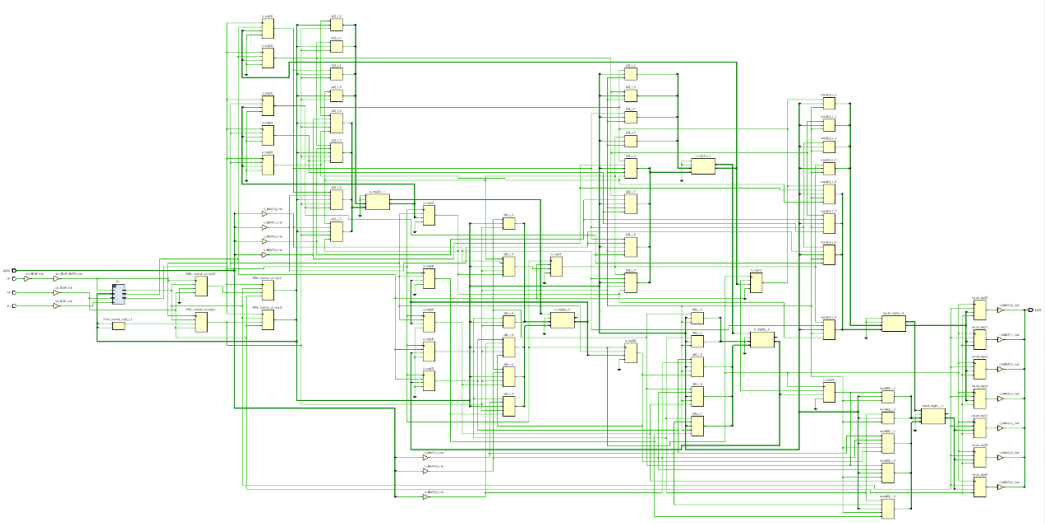


3、最终结果

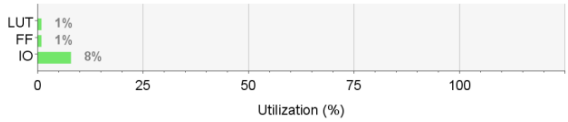
生成的 RTL 电路如下：



综合电路、资源利用、时间性能报告(setup 与 hold)如下：



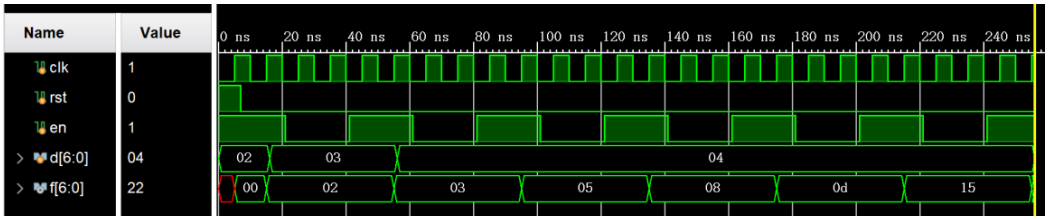
Resource	Utilization	Available	Utilization %
LUT	25	63400	0.04
FF	27	126800	0.02
IO	17	210	8.10



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 3	7.388	3	4	7	b_reg[0]/C	result_reg[5]/D	2.508	1.710	0.798
Path 4	7.469	3	4	7	b_reg[0]/C	a_reg[6]/D	2.427	1.629	0.798
Path 5	7.469	3	4	7	b_reg[0]/C	b_reg[6]/D	2.427	1.629	0.798
Path 6	7.469	3	4	7	b_reg[0]/C	result_reg[6]/D	2.427	1.629	0.798
Path 7	7.487	1	2	7	FSM_o...[1]/C	b_reg[0]/CE	2.131	0.799	1.332

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 11	0.142	1	2	4	a_reg[0]/C	b_reg[0]/D	0.386	0.245	0.141
Path 12	0.152	0	1	1	FSM_o...[0]/C	FSM_o...[1]/D	0.288	0.147	0.141
Path 13	0.155	2	3	3	a_reg[2]/C	a_reg[2]/D	0.413	0.257	0.156
Path 14	0.155	2	3	3	a_reg[6]/C	a_reg[6]/D	0.413	0.257	0.156
Path 15	0.157	2	3	3	a_reg[1]/C	a_reg[1]/D	0.415	0.258	0.157

模拟效果如下：



（FPGA 的结果已在上方给出）

总结：

由于没有选过数字电路与数字电路实验，第一次实验进行地并不算顺利。不论在烧写 FPGA、时间性能报告还是电路设计上都因为经验不足遇到了一些障碍。比如，检测上升沿其实是在 verilogoj 上做过的题目，但真的遇到问题时也并没有想到解决方案，是在助教提醒下查看了数字电路实验文档 08 才知道应该如何解决。

因此，一方面是不要忘记自己做过的题目，另一方面则是学会查看数字电路时的实验文档，二者结合应能解决不少问题。

2-寄存器堆与储存器及其应用 实验报告

PB20000296 郑滕飞

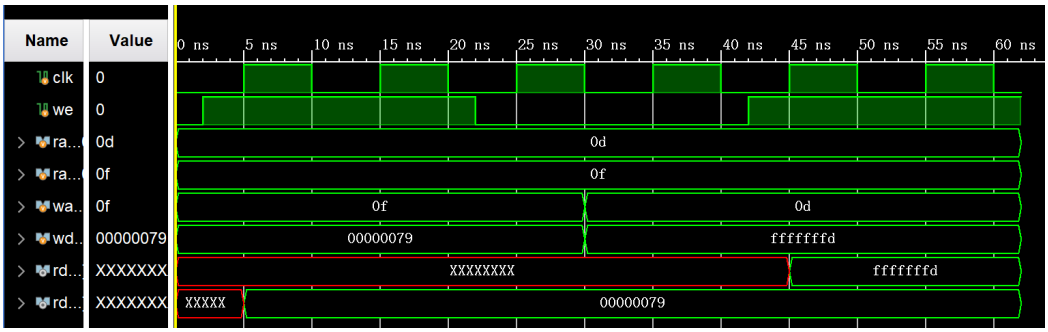
FIFO:

1、寄存器堆

根据题目所给要求，可直接写出核心代码（注：32×WIDTH 寄存器堆与 8×4 寄存器堆代码无区别，此段中仿真波形展示的是 32×32，之后应用的都是 8×4）：

```
assign rd0 = regfile[ra0];
assign rd1 = regfile[ra1];
always @(posedge clk)
    if (we) regfile[wa] <= wd;
```

通过仿真可以得到波形：



2、SDK 模块

此模块是作为刷新显示使用的，寄存器 xr 从 0 到 7 不断遍历，如果遇到了输入的 valid 为真，an 就调整到对应位置，并且 sel 存入对应位置的值。

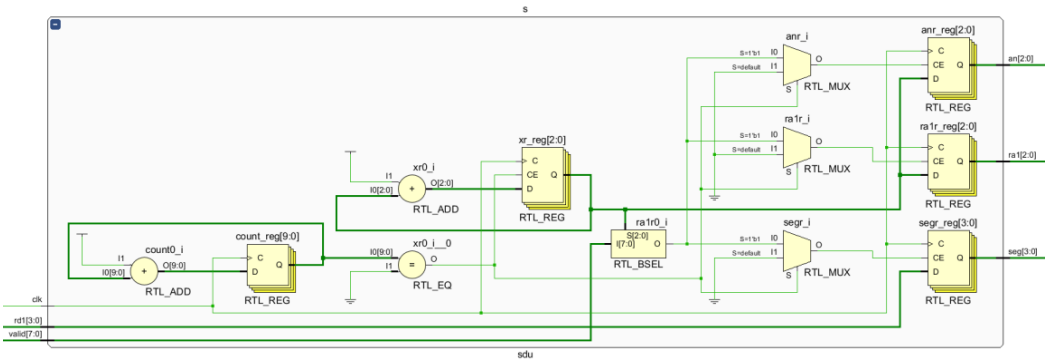
起初的代码如下：

```
always @(posedge clk) begin
    xr <= xr + 1;
    if (valid[xr]) begin
        ra1r <= xr;
        anr <= xr;
        segr <= rd1;
    end
end
```

其中 ra1r,anr,selr，分别代表对应信号连接的寄存器，而 ra1 的数据读入到 rd1 中，从而传给 seg。

然而，这样的设计虽然通过了仿真，但在下载测试时出现了问题，无法正常显示队列，经过分析发现主要原因在于刷新的频率过高，超过了灯的反应速度。由此，在其中又附加了 count 寄存器，在 0 到 1023 循环，每次循环到 0 再让 xr 前进一位，由此即可正常显示(此时实际刷新一遍所需的时间为$1024 \times 8 \times 10ns$，即约为$0.08ms$)。

此模块的 RTL 电路图如下：



3、LCU 模块-状态机的利弊

在标准的实验要求上，需要用状态机实现 LCU 模块，但在实际书写代码时，发现此块利用状态机事实上只是增加了复杂度，无论在可读性还是实现容易程度上都不如不使用状态机的版本，下面分析理由：

首先，状态机最方便之处是状态间切换的复杂性，例如，现在有 A,B,C 三个状态，在 A 状态时，经历 b1 到 B，经历 c1 到 C，否则不变；在 B 状态时，经历 a2 到 A，经历 c2 到 C，否则不变；在 C 状态时，经历 a3 到 A，经历 b3 到 B，否则不变。这种情况下，如果使用 case 语句，首先需要一个指标变量 flag 决定所在的状态，再根据接收的不同数据决定 flag 的切换，这时如果不将状态与内容分开，就容易出现错误。然而，如果不管是什么状态都在条件 a 切换为 A，条件 b 切换为 B，条件 c 切换为 C，那么

```
if (a) begin
    function at A;
end
else if (b) begin
    function at B;
end
else if (c) begin
    function at C;
end
```

这样的实现方式既清晰又能减少电路的复杂度。回到这次实验的情况，如果有 rst 就进入初始状态，切换到 ENQU 状态的方式恒为 enq 上升沿且不满，切换到 DEQU 的状态的方式恒为 deq 上升沿且不空，否则就是 IDLE，什么都不做的静态，符合所说的后一种情况，因此不用状态机更为简便

其次，这次实验中的状态机有一个很明显的长期存在状态，也就是 IDLE 静态。在极端的情况下，其他所有的状态都可以先回到静态后再切换，并不需要构建几种状态之间的切换。在这样一个稳态和其他非稳态的情况里，用 if-else 能更好表现离开稳态进行操作的状态，使代码更易读懂。

由此，代码的核心结构是：

```
always @(posedge clk)
    if (rst) begin
        head <= 0;
        tail <= 0;
        validr <= 0;
```

```
        empr <= 1;
        fullr <= 0;
        wer <= 0;
    end
    else if (enq_edg & !full) begin
        wer <= 1;
        war <= tail;
        validr[tail] <= 1;
        wdr <= in;
        tail <= tail1;
        fullr <= tail1 == head;
        empr <= 0;
    end
    else if (deq_edg & !emp) begin
        wer <= 0;
        validr[head] <= 0;
        ra0r <= head;
        outr <= rd0;
        head <= head1;
        fullr <= 0;
        empr <= head1 == tail;
    end
    else wer <= 0;
```

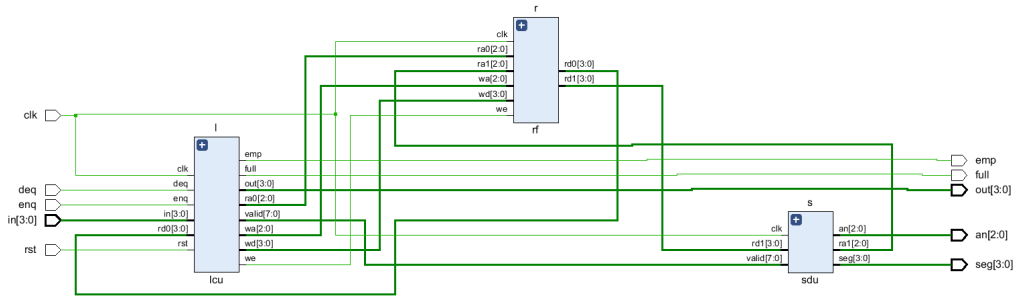
其中后面带 r 的为寄存器，去掉 r 即为其所连接的线路的名字，而 head1 与 tail1 都表示值为 head+1/tail+1 的 3 位 wire 型变量。由这部分结构即产生了一个循环队列。

值得注意的是，由于非阻塞赋值，不同寄存器里的值同时变化，因此虽然入队后 head 与 tail 相等表示队满，事实上需要比较 head+1 与 tail 来确定队满(因为入队后 head 成为 head+1)。

(此部分的 RTL 电路将在下一部分展示)

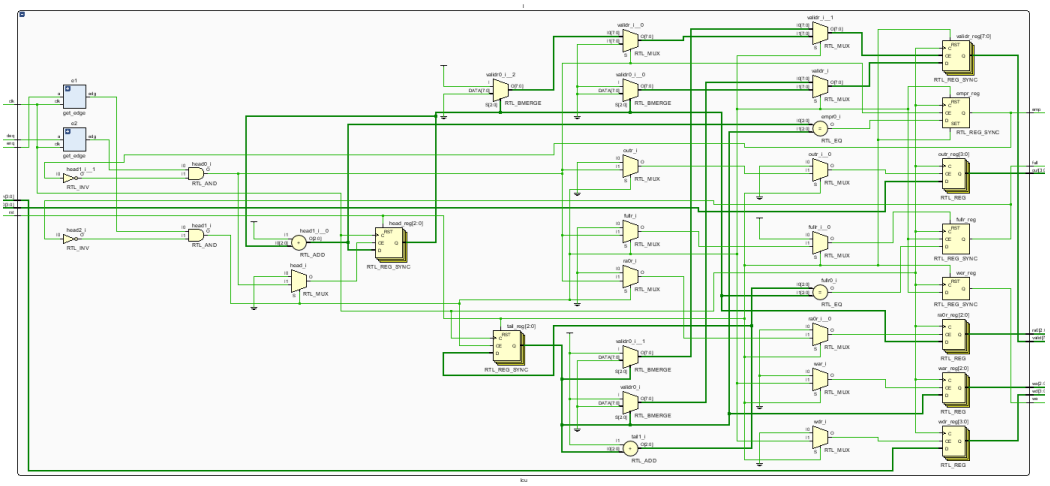
4、总体设计-模块化与合并模块

按照 ppt 中的设计，可以得到如下的模块化的 RTL 电路图:



全部展开后可以得到它使用了 97 Cells, 21 I/O Ports, 264 Nets。

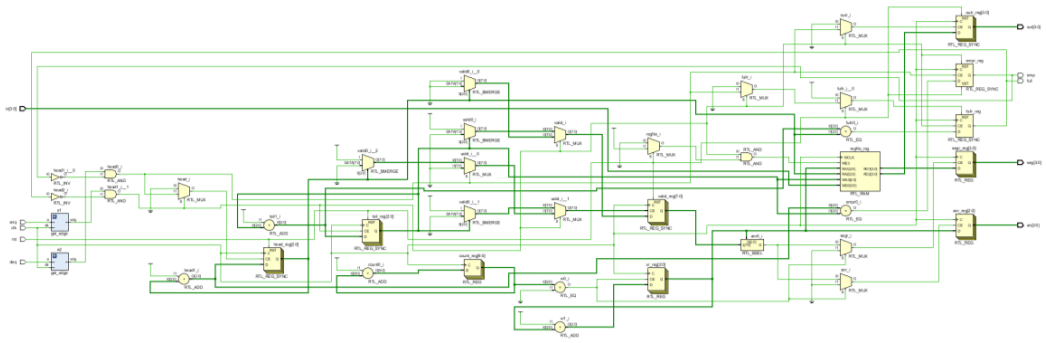
其中 LCU 模块的 RTL 电路图如下:



在写这些代码时，有一个很麻烦的情况，也就是确定 I/O 的接口。尤其是中间的寄存器堆，其读与写的代码并不复杂，对应接口却要消耗很久。由此，我尝试了将三个模块合为一体的写法。这时，代码的部分确实可以简化不少，例如 LCU 的核心代码：

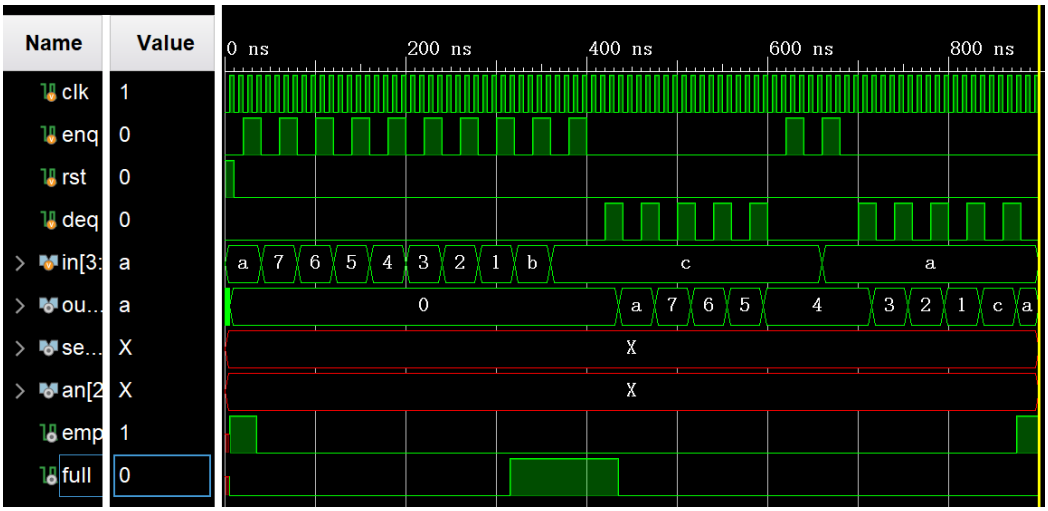
```
always @(posedge clk)
    if (rst) begin
        head <= 0;
        tail <= 0;
        valid <= 0;
        empr <= 1;
        fullr <= 0;
        outr <= 0;
    end
    else if (enq_edg & !full) begin
        valid[tail] <= 1;
        regfile[tail] <= in;
        tail <= tail1;
        fullr <= tail1 == head;
        empr <= 0;
    end
    else if (deq_edg & !emp) begin
        valid[head] <= 0;
        outr <= regfile[head];
        head <= head1;
        fullr <= 0;
        empr <= head1 == tail;
    end
```

可以发现形成的 RTL 电路如下，展开后可知，I/O Ports 个数仍为 21，不过所需的 Cells 降到了 70，减少了近 30%，而 Nets 更是降到了 151，减少了 40%多。由此可见，不使用模块的情况下，电路的复杂度大大降低了。模块的主要优势在于可复用与方便 debug，但电路复杂度过高时，适当合并模块或许是更好的选择。

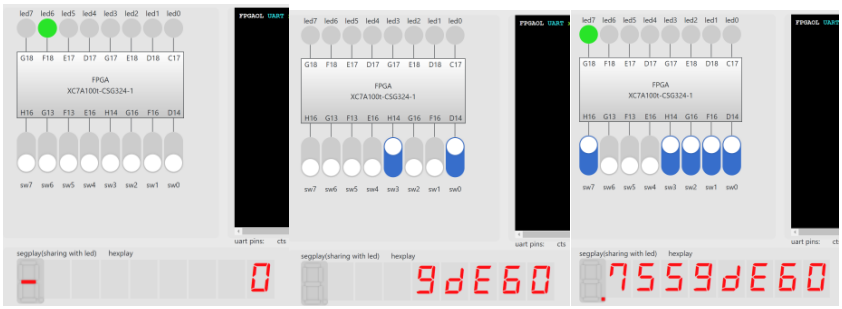


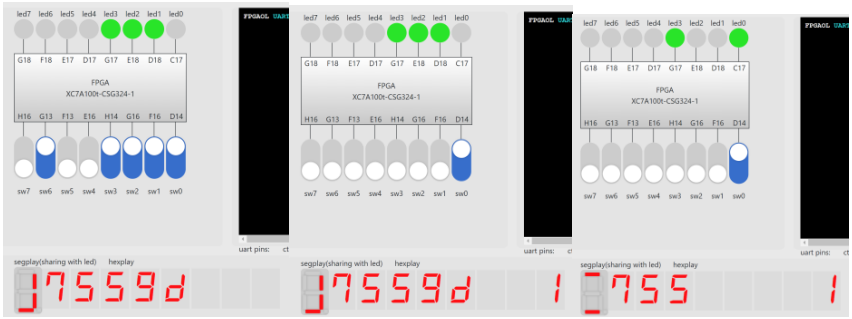
5、效果展示

仿真波形如下(由于 SDU 模块引入 count 计数器作延时，在仿真波形里无法看出功能，但根据入队出队情况与空满标志可知正确)：



烧写结果如下：



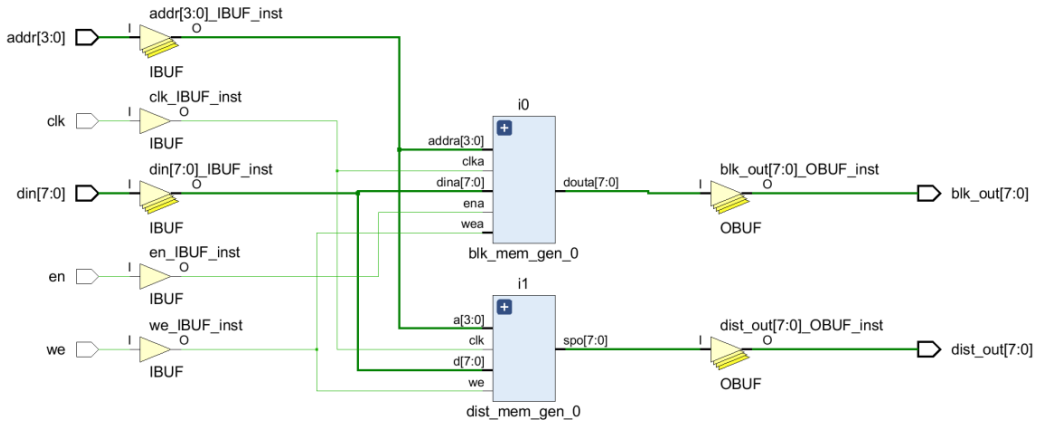


（展示的具体内容为：初始，入五个，再入三个满后不能入，出三个，入一个，出两个）

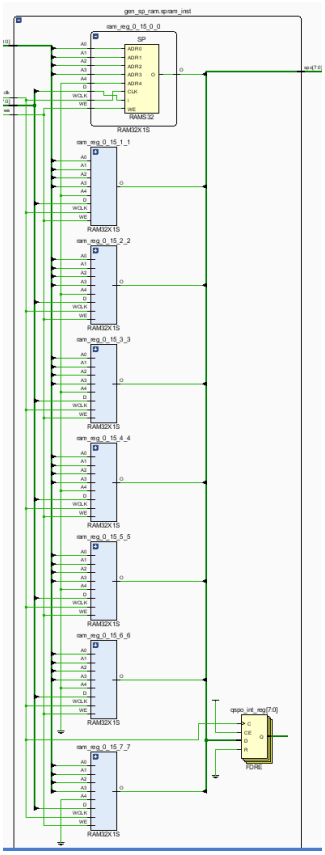
RAM：

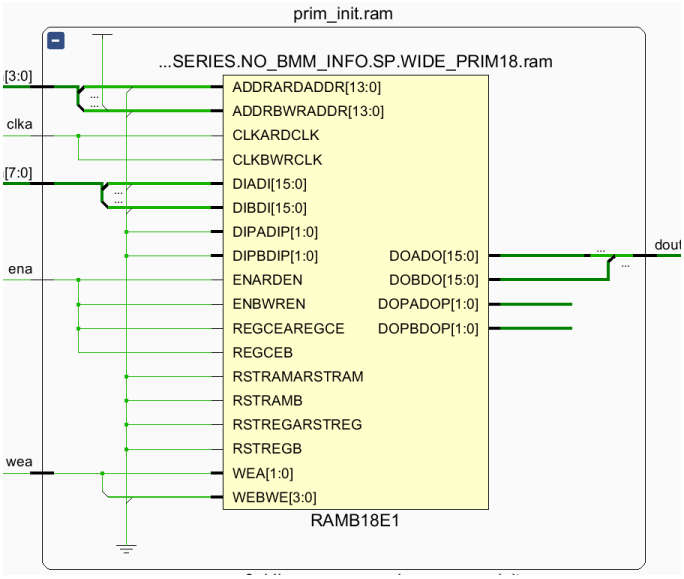
1、例化与结构层面对比

例化时，直接给两种不同的 RAM 连入相同的输入(块状的选择了写优先的模式)，RTL 电路如下：



将两个 RAM 展开后，可以发现结构上的明显差异：





左侧为分布式 RAM 展开后的内部结构，可以发现将每一位分别储存，因此共有 8 个不同的小 RAM 分别负责存储，再通过出口处的寄存器组合。

右侧为块式 RAM 展开后的内部结构，可发现正如其名字一样成为了一整块，对信号进行了整体的处理与操作。(块状 RAM 的内部结构特点与影响将在下一部分中进行阐述)

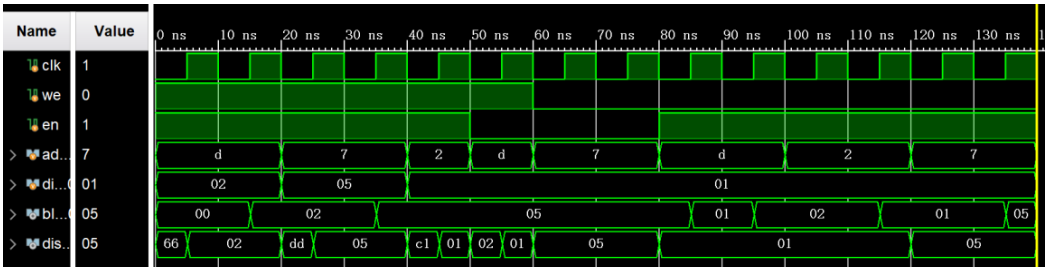
在端口上，分布式 RAM 没有将总使能与写使能区分，只有一个写使能，一直进行读的操作，这也导致了之后波形的一些差异。

2、仿真与波形层面对比

仿真前所写的 COE 文件如下：

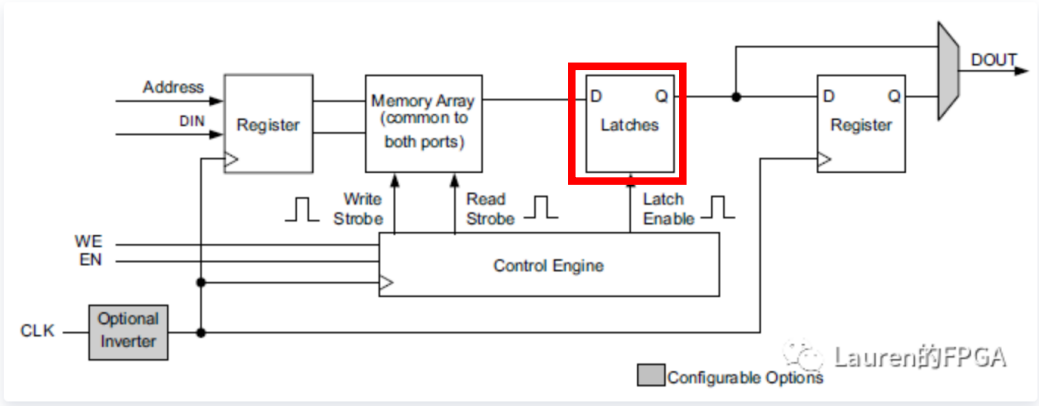
```
memory_initialization_radix = 16;
memory_initialization_vector =
a1 b2 c1 d2 aa bb cc dd 11 22 33 44 55 66 77 88;
```

仿真波形如下：



从波形中可看出若干区别：由于分布式 RAM 没有总使能信号，读取是异步进行的，而写才是同步；块式 RAM 则读写都是同步进行。更为奇怪的一点是，块式 RAM 的数据显示总有着一个时钟周期的延迟，虽然读入的时间正常，但每次都是延迟一个时钟周期后才反映到输出中。

根据助教提供的链接，其中由这么一张图：



图中可以看出，数据在输出之前需要先经历锁存器(红圈)，再进入输出的寄存器。正是这个锁存器导致了一个时钟周期的延迟。

总结：

这次实验主要的收获在于讨论了状态机使用与模块化设计的优缺点。状态机提供了很好的分离形式意义上的状态与每个状态的内涵、操作的方式，但在一些情况下直接使用 `if-else` 代码块更加简单便捷；模块化设计方便了 debug、复用，增加了可读性，但也导致了接线更为复杂。在之后的设计中，需要自己在这些优缺点间作出权衡。

3-汇编程序设计 实验报告

PB20000296 郑滕飞

示例程序：

1、代码含义

示例输出程序分为几个部分：

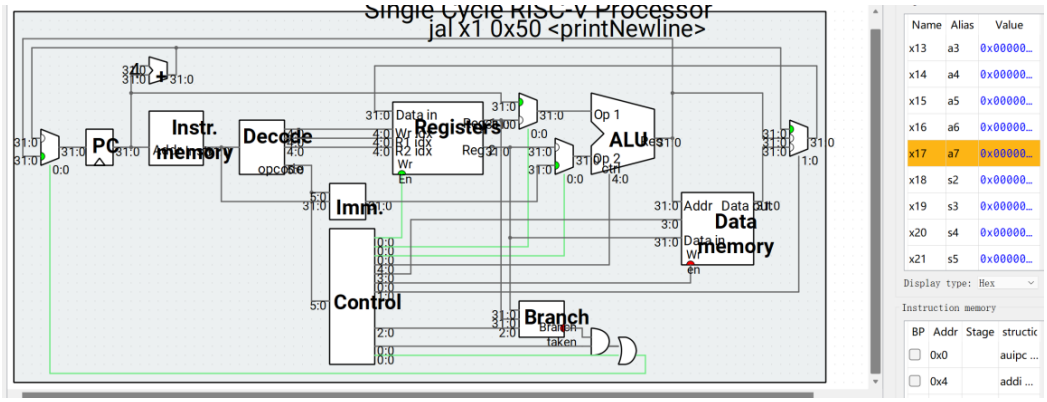
第一部分为打印一个字符串 `A string`，将首地址放入 `a0`，`a7` 置为 `4`(代表以 `a0` 为首地址以字符串形式打印)后调用 `ecall` 打印。之后打印换行与逗号分隔都是用相同的方式。

第三部分与第一部分结构相同，只是 `a7` 置为 `2`(代表以浮点数方式打印 `a0` 中存储的数)后直接打印。

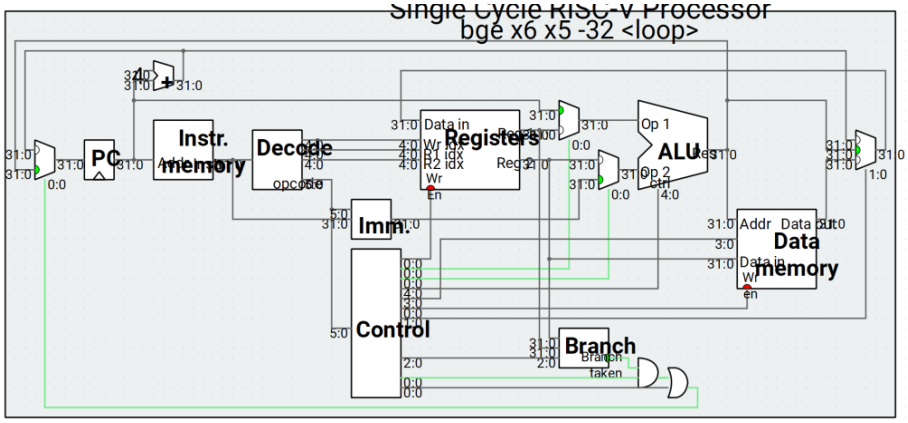
第二部分与第四部分都调用了 `loopPrint` 函数。函数将 `a0,a1` 传递给 `t0,t1`，`t0` 每次增加 `1`，小于等于 `t1` 则持续循环，实质上遍历了`[t0,t1]`间的所有整数。每次遍历时，都调用 `ecall` 以 `a2` 的模式打印(`1` 与 `11` 分别代表以带符号整数/ASCII 码方式打印 `a0` 中存储的数)。

全部打印结束后，将 `a7` 置为 `10` 调用 `ecall` 以结束程序。

2、仿真



从单步仿真中，可以看到寄存器值的变化与运用的数据通路。例如，上方图片中的指令为 `jal`，打开了写入寄存器的使能并将当前 PC 保存到了寄存器，同时计算新的 PC 地址，利用 MUX 让 PC 存入新的地址；而下方的 `bge` 指令没有保存寄存器的环节，但是需要通过 Branch 确定 PC 是否跳转到新的地址。



仿真的最终结果如下，控制台输出与预期一致：

```
Console

A string

-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
3.14159
!, ", #, $, %, &, ', (, ), *, +, ,, -, ., /, 0, 1, 2, 3, 4, 5,
Program exited with code: 0
```

测试程序：

1、lw 与 sw

读取 in 中的数据，并且在不同情况下存入 out 中以检测存取指令，代码为：

```
la a0, out
sw x0, 0(a0)
addi t0, x0, 0x55
sw t0, 0(a0)
lw t0, 4(a0)
sw t0, 0(a0)
```

2、add 与 addi

通过直接进行加法操作以检测加法指令，代码为：

```
addi t1, x0, 0x01
sw t1, 0(a0)
addi t2, x0, 0x07
sw t2, 0(a0)
add t3, t1, t2
sw t3, 0(a0)
```

3、jal

将加法的代码放入 jal 中以检测调用，代码为：

```
    jal begin
```

```
begin:
    addi t1, x0, 0x01
    sw t1, 0(a0)
    addi t2, x0, 0x07
    sw t2, 0(a0)
    add t3, t1, t2
    sw t3, 0(a0)
    jr x1
```

4、beq

在 jal 代码中已将 t1,t2 的值置为 1 与 7，由此设计循环可检测 beq：

```
loop:   beq t1, t2, exit
    addi t2, t2, -1
    j loop
```

最后添加结束部分即可结束代码运行。

FLS:

以下假设 in 中存储需要的项数，在 out 中输出项。代码如下:

```
        la a0, out
        lw t0, 4(a0) #将输入读入 t0
        addi t1, x0, 0x01
        addi t2, x0, 0x01 #将 t1,t2 初始值设为 1
loop:   beq t0, x0, output #如果 t0 为 0 则退出
        add t2, t2, t1 #将 t2 设置为和
        sub t1, t2, t1 #将 t1 设置为原来的 t2
        addi t0, t0, -1 #将 t0 减少 1 以计数
        j loop

output: sw t1, 0(a0) #输出 t1
        li a7, 10
        ecall #退出
```

先将 t1,t2 设置成第零项与第一项，在每次循环中均前进一项，这样输入 t0 与循环次数相等，也与 t1 所保存的项数相等，输出 t1 即满足要求。

Address	Value (+0)	Value (+4)
0x10010000	0x00000000	0x0000000a

Address	Value (+0)	Value (+4)
0x10010000	0x00000059	0x0000000a

测试数据输入为 10，斐波那契数列第 10 项为 89，16 进制表示为 0x59，结果正确。

4-单周期 CPU 设计 实验报告

PB20000296 郑滕飞

CPU：

1、指令储存器

在 CPU 设计中，指令储存器利用 coe 文件初始化后即不会再被改变，因此直接采用了 ROM 的 IP 核，也即只有读地址与读数据端口。实际设计时，用 PC 当前值(储存在寄存器 pc_reg)中的[9:2]位(由于涉及地址总是 4 的倍数且四位地址对应一个 ROM 寄存器，实际需要的是这八位来控制)接入读地址端口，读数据端口输出接到 ir，也即代表当前指令。之后的操作都是利用 ir 进行。

2、寄存器堆

此部分直接利用了之前写好的 32 个 32 位寄存器，增加了连接 m_rf_addr[4:0]的读端口与 rf_data 的出口。由于 rs1,rs2,rd 在指令中的位置是固定的，直接连入 ir[19:15], ir[24:20]与 ir[11:7]。两个读端口的出口记为 rs1d 与 rs2d，决定是否写入的 RegWrite 则由控制单元产生。

3、控制单元

此部分传入 opcode，也就是 ir[6:0]，来调整一些控制信号。具体的实现方式是先判断 opcode 属于哪个指令(AS 代表 ADD/SUB)：

```
assign JAL = (opcode == 7'b1101111);
assign JALR = (opcode == 7'b1100111);
wire ADDI = (opcode == 7'b0010011);
wire AS = (opcode == 7'b0110011);
assign BRANCH = (opcode == 7'b1100011);
wire AUIPC = (opcode == 7'b0010111);
wire LW = (opcode == 7'b0000011);
wire SW = (opcode == 7'b0100011);
```

再根据不同的需要决定控制信号(此处不考虑指令无法识别的情况，因此有的直接用了非门)：

```
assign MemWrite = SW;
assign ALU1Scr = AUIPC;
assign ALU2Scr = ~(AS|BRANCH);
assign RegWrite = ~(BRANCH|SW);

assign RegScr[0] = LW;
assign RegScr[1] = JALR|JAL;

assign ALUop[0] = AS;
assign ALUop[1] = BRANCH;

assign Imm_gen[0] = JAL|BRANCH|AUIPC;
assign Imm_gen[1] = BRANCH|SW;
```

```
assign Imm_gen[2] = ADDI|LW|AUIPC|JALR;
//1-JAL 2-SW 3-BRANCH 4-ADDI/LW/JALR 5-AUIPC
```

一位信号中，ALU1Scr 为真代表 ALU 第一个操作数选择 PC，否则为 rs1d，ALU2Scr 为真代表 ALU 第一个操作数选择立即数，否则为 rs2d。两位信号 RegScr 用于选择存入的数据，如果为 0 则代表存入 ALU 的运算结果，为 1 代表存入数据储存器中读出的值，为 2 代表直接存入 PC+4；ALUop 则是用于将 AS 和 BRANCH 与其他指令区分开，然后在 ALU 控制器中结合 func3 与 func7 生成真正的 ALU 控制信号。Imm_gen 根据不同立即数的类型分类，在接下来的立即数生成模块后生成 32 位的立即数。

4、立即数生成器

这一部分的主要代码如下：

```
always @(*) case(Imm_gen)
    1: Imm = {{12{ir[31]}},ir[19:12],ir[20],ir[30:21],{1'b0}};
    2: Imm = {{21{ir[31]}},ir[30:25],ir[11:7]};
    3: Imm = {{20{ir[31]}},ir[7],ir[30:25],ir[11:8],{1'b0}};
    4: Imm = {{21{ir[31]}},ir[30:20]};
    5: Imm = {ir[31:12], {12'h000}};
    default: Imm = 0;
endcase
```

根据立即数信号的不同类型，生成各种不同的立即数。值得注意的是，由于这里已经将 BRANCH,JAL,JALR 这些指令单独区分，shiftleft 操作直接以最后一位添加 1’b0 完成，不再需要单独的单元。

5、ALU 控制器

此部分用来生成 ALU 的控制信号：

```
assign ALUc[0] = (ALUop[0]&add_sub)|(ALUop[1]&~beq_blt);
assign ALUc[1] = ALUop[1]&beq_blt;
//0-add 1-sub 2-lt
```

当接收到 SUB 或者 BEQ 时，需要调用减法，当接收到 BLT 时，需要额外考虑，因此将这三者作了区分。

6、ALU

注意到，是否小于可利用减法结果的首位来判断，由此写出如下代码：

```
assign result = ALUc ? op1 - op2 : op1 + op2;
assign zero = ALUc[1] ? result[31] : ~|result;
```

SUB,BEQ,BLT 时均计算减法，否则计算加法。当指令为 BLT 时，将 BRANCH 的标记设置为减法结果的第一位，否则判断是否为 0。

7、PC 变化

此部分中，用 PC4 存入 PC+4 的结果，PCImm 存入 PC+Imm 的结果，利用信号 jump 进行选择。jump 的生成为：

```
assign jump[0] = JAL | (BRANCH&zero);
assign jump[1] = JALR;
```

当 jump 为 0 时，PC 正常前进。当 jump 为 1 时，代表触发 PC+立即数的跳转，jump 为 2 时则为 ALU 计算结果(此时即 rs1d+立即数)并将最后一位改为 0。

8、数据储存器

利用分布式的双端口 RAM，将 DEBUG_BUS 中的 m_rf_addr 接入只读端口，并将 m_data 接到只读端的出口。读写的地址均为 ALU 的运算结果(此时为 rs1d+立即数)，而写入的数据来自 SW，因此为 rs2d，we 则接入控制单元产生的 MemWrite。

9、寄存器堆写入数据

与 PC 类似，也是利用一个三选一的 mux，选择分别为 ALU 计算结果(一般情况)、数据储存器读取(LW 时)、PC4(JAL 与 JALR 时)。

10、IO_BUS

当 ALU 计算出的地址为 0x00004xxx 时，需要利用到 io，由此前述的控制信号需要根据 alu_result[10]作出一定的修改。当发现其为 1 时，应阻断正常的读写，接入 io 的读写，例如：

```
assign MW_real = MemWrite&~alu_result[10];
assign memd_real = alu_result[10] ? io_din : memd;
```

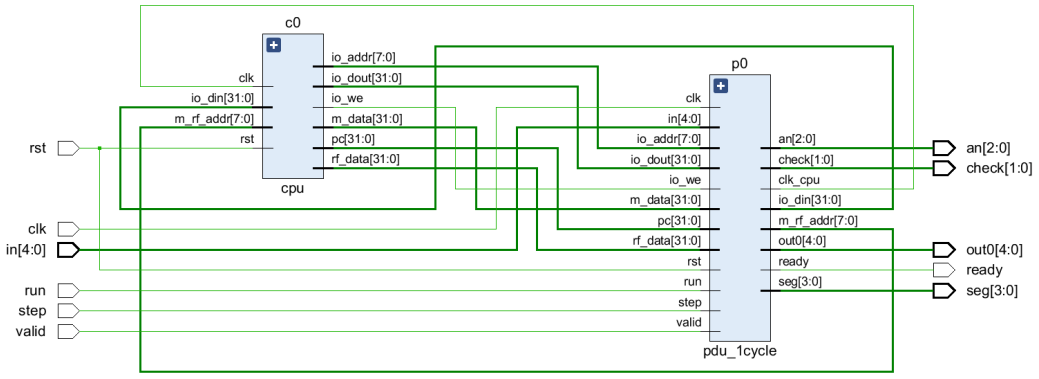
上面的两句分别为 MemWrite 信号与读出数据的调整，而 io_we 也需要凭此输出：

```
assign io_we = MemWrite & alu_result[10];
```

利用 alu_result[10]作总控制后，即可以处理输入输出。

11、连接 PDU

编写 top 模块将 CPU 与 PDU 连接，最终的 RTL 电路图如下：



结果展示：

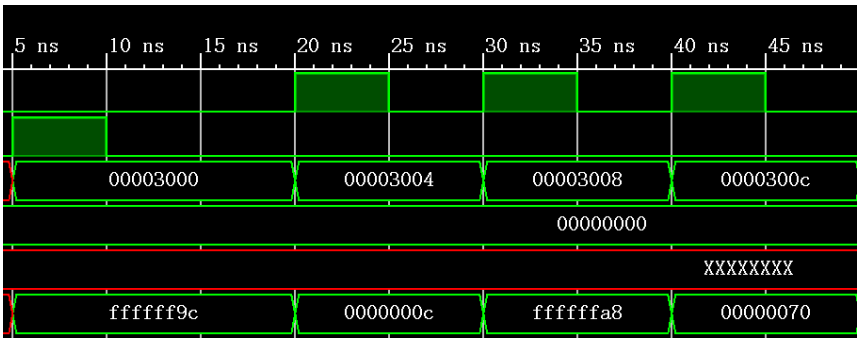
首先编写汇编文件测试十条指令能否正确实行：

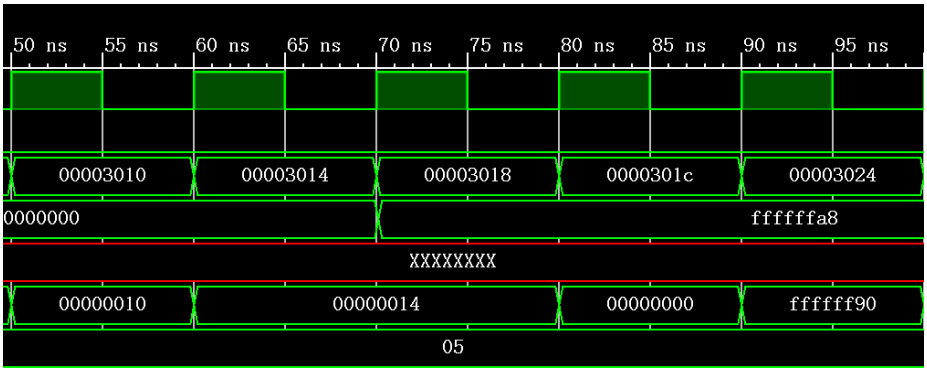
```
addi t1, x0, -100
addi t2, x0, 12
add t3, t1, t2
sub t4, t2, t1
auipc t5, -3
```

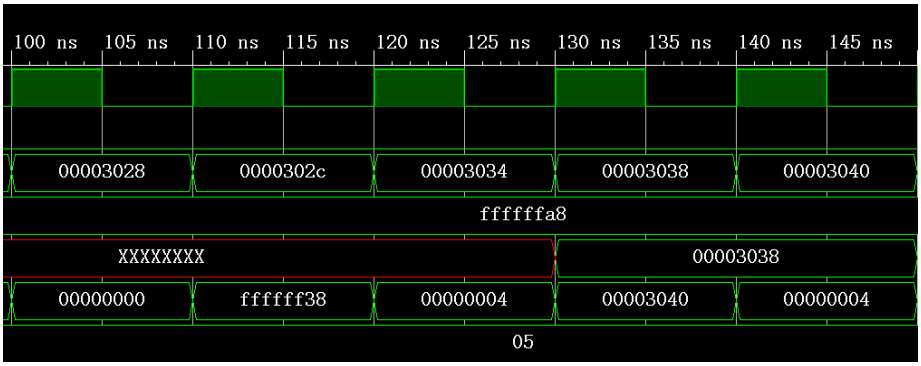
```
    sw t3, 4(t5)
    lw t6, 4(t5)
    beq t3, t6, lb1
    addi t0, x0, 0
lb1:beq t1, t2, lb1
    blt t3, t3, lb1
    blt t3, t4, lb2
    addi t0, x0, 0
lb2:jal t0, lb4
lb4:jalr t6, t0, 8
    addi t0, x0, 0
    addi t0, x0, 4
```

为了测试是否正确，在单周期 CPU 的 DEBUG_BUS 上增加了一条 alur，用来监测 ALU 的运算结果。

编写仿真文件查看(由于 ALU 运算结果已经被监测，m_rf_addr 不用精细调整)：

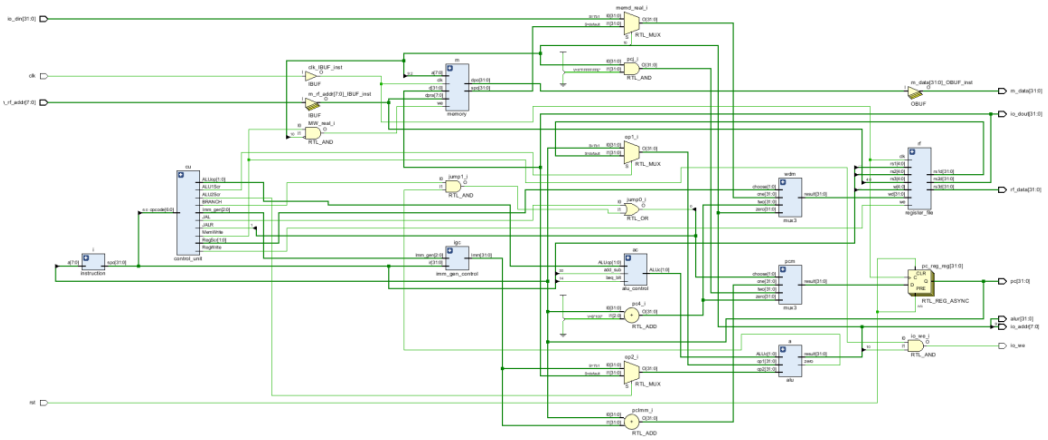






从上到下依次为 clk,rst,pc,m_addr,rf_addr,alur,m_rf_addr 根据 PC 的跳转情况与 ALU 的输出结果可知程序正确。

CPU 的 RTL 电路图如下：



利用老师所给的 fls 文件初始化指令储存器后，以下为下载测试结果：

(重置后输入前两项 2,1，计算出接下来的 3,4,7,11)

总结：

这次实验的教训堪称惨痛。由于没有养成看数据通路的习惯，花了九个多小时的额外时间才做完。当设计的结构比较复杂时，一定要根据数据通路进行模块化设计，否则即使理论上可以完成也容易出现各种布线问题，这是硬件所具有的特殊性。

5-流水线 CPU 设计 实验报告

PB20000296 郑滕飞

基础 CPU：

1、减回 6 条指令

在单周期 CPU 的基础上，由于 10 条指令中存在不少 6 条指令不需要的处理，先将指令减回了 6 条。此时，ALU 控制单元可以直接省略，因为只有 beq 指令会让 alu 执行减法，通过 opcode 即可判断。此外，为 auipc 与 jalr 添加的部件也可以删除，因此最终的控制单元如下：

```
module control_unit(
    input [6:0] opcode,
    output JAL, BRANCH, MemWrite, ALU2Scr, RegWrite,
    output [1:0] RegScr, Imm_gen
);
assign JAL = (opcode == 7'b1101111);
wire ADDI = (opcode == 7'b0010011);
wire ADD = (opcode == 7'b0110011);
assign BRANCH = (opcode == 7'b1100011);
wire LW = (opcode == 7'b0000011);
wire SW = (opcode == 7'b0100011);
assign MemWrite = SW;
assign ALU2Scr = JAL|ADDI|LW|SW;
assign RegWrite = JAL|ADDI|ADD|LW;
assign RegScr[0] = LW;
assign RegScr[1] = JAL;
assign Imm_gen[0] = SW|JAL;
assign Imm_gen[1] = BRANCH|JAL;
//0-ADDI/LW 1-SW 2-BRANCH 3-JAL
endmodule
```

2、寄存器堆改写

在原本的寄存器堆中，读写的实现方式是(以第一个读端口为例，writing 信号表示写入且写入目标不为 x0)：

```
assign writing  = we && w;
assign rs1d = regfile[rs1];
always @(posedge clk) if (writing) regfile[w] <= wd;
```

但是，此时如果在同一个时钟周期读写，写入 wd 的同时读出了旧数据，对以下指令的测试即会发生冒险：

```
addi x1, x0, 1
NOP
NOP
add x2, x1, x1
```

因此，当写入目标与读目标相同时，必须直接读出写入的值，也就是读优先，具体实现

为：

```
wire writing = we && w;
assign rs1d = (writing && w == rs1) ? wd : regfile[rs1];
always @(posedge clk) begin
    if (writing) regfile[w] <= wd;
end
```

3、流水段寄存器

为了实现流水线，需要在 IF_ID、ID_EX、EX_MEM 与 MEM_WB 四个段落设置寄存器，每个时钟周期前进。例如，ID_EX 段落的实现如下：

```
always @(posedge clk) begin
    ctrl[6:0] <= {RegScr, RegWrite, ALU2Scr, MemWrite, BRANCH, JAL};
    a <= rs1d;
    b <= rs2d;
    imm <= Imm0;
    rd <= rd0;
    pce <= pcd;
end
```

其中，ctrl 部分是利用不同的位来接收控制单元生成的控制信号，其他则是单纯信号的传递。流水线的实现特点是，每个流水段的操作都是运用上个流水段传入的信号，例如 EX 段：

```
assign op2 = ctrl[3] ? imm : b;
alu alu0(.op1(a), .op2(op2), .ALUc(ctrl[1]),
    .result(alu_result), .zero(zero));
assign pcImm = pce + imm;
assign jump = ctrl[0] | (ctrl[1] & zero);
assign pcin = jump ? pcImm : pc4;
```

这里的 pc 控制与 alu 都是利用刚刚传来的信号，以保证实现的恰好是到达这一流水段的代码。

其他流水段寄存器的设计也与这段类似，不过，在外设中提供的端口没有考虑到 jal 指令需要将 pc+4 写回寄存器，在实际实现的时候，不能仅仅到 pce 为止，还需要传到 pcm 与 pcw，最后的 MEM_WB 实现如下：

```
always @(posedge clk) begin
    ctrlw <= ctrlm;
    mdr <= memd_real;
    yw <= y;
    rdw <= rdm;
    pcw <= pcm;
end
```

4、初值设定

由于在第一条指令执行时(流水线执行的前四个周期)，流水线中后方部分如果不在运行，就不能实现 pc 的改变，也会导致其他部分的结果不确定，因此实质上需要通过给控制

信号赋予初值在其中填充 NOP 指令。

值得注意的是，控制信号并不只有 ctrl、ctrlm 与 ctrlw，还有 IF_ID 中的的 ir。为了保证结果正确，需要给这些部分赋予初值 0，这样可以保证除了 pc 正常前进之外不进行任何其他操作，也就保证了代码前几步的运行正确。

5、外设接口

从上个实验的通路中可以看出，外设实现作用实际上是利用特殊的地址，阻断了正常的对储存器读写，而是从外设去读写。因此，外设应该放在 MEM 部分，具体的实现只需要把上个实验的连线改为 MEM 对应信号的连线即可：

```
assign io_we = ctrlm[2] & y[10];
assign MW_real = ctrlm[2] & ~y[10];
memory m(.a(y[9:2]), .d(bm), .dpra(m_rf_addr),
    .clk(clk), .we(MW_real), .spo(memd),
    .dpo(m_data));
assign memd_real = y[10] ? io_din : memd;
assign io_addr = y[7:0];
assign io_dout = bm;
```

至此，不发生冒险时的情况已经解决，接下来就需要处理

冒险处理：

1、跳转指令处理

事实上，在上一部分中已经给出了插入 NOP 指令的方式，也就是将对应的控制信号设置为 0。发现分支被采用或者跳转指令是在 EX 阶段，此时 pc 跳转到对应位置执行，应该将之前的 IF_ID 与 ID_EX 输出的控制信号设置为 0。

以 ID_EX 为例，需要将 EX 段得出的是否采用的 jump 信号输入进模块，当发现其为 1 时设置为 0，也就是：

```
initial ctrl = 0;
always @(posedge clk) begin
    ctrl[6:0] <= jump ? 7'b0 : {RegScr, RegWrite, ALU2Scr, MemWrite,
BRANCH, JAL};
end
```

对 IF_ID 中的控制则是：

```
initial ir = 0;
always @(posedge clk) begin
    ir <= jump ? 32'b0 : ir0;
end
```

由此，只要发生跳转，就清楚已经进入流水线的指令，而如果不发生，则正常进行，这就完成了分支与跳转。

2、加载-使用冒险

与跳转处理的想法类似，当发生加载-使用冒险之后，需要让使用及之前的指令停顿一级，并且清除 ID_EX 输出的控制信号。

首先，利用 luhazard 单元判断是否发生了加载-使用冒险(此处的 rs1,rs2 连入的实

际上是 ir[19:15],ir[24:20]):

```
assign hazard = ctrl[5] & (rs1 == rd | rs2 == rd);
```

接着,将 hazard 传入 PC、IF_ID、ID_EX 三级,并且作对应处理。PC 部分只有 luhazard 不发生时才前进:

```
always @(posedge clk, posedge rst)
    if(rst) pc_reg <= 32'h00003000;
    else if(~hazard) pc_reg <= pcin;
```

IF_ID 部分也是同理。由于 luhazard 只在 load 与 use 连续发生时才会出现，事实上不可能与跳转同时发生，因此先后均可。此处设定为跳转优先:

```
always @(posedge clk) begin
    ir <= jump ? 32'b0 : (hazard ? ir : ir0);
end
```

ID_EX 部分则与之前一致，直接清除 ctrl 即可:

```
always @(posedge clk) begin
    ctrl[6:0] <= (jump | hazard) ? 7'b0 : {RegScr, RegWrite, ALU2Scr,
MemWrite, BRANCH, JAL};
end
```

通过这样的控制，即可保证加载-使用冒险被正确解决。

3、前递

最后，需要完成其他数据冒险的解决方法，也就是前递模块:

```
forwarding F(.ctrlm(ctrlm), .ctrlw(ctrlw), .rdm(rdm),
    .rdw(rdw), .rs1(rs1), .rs2(rs2),
    .afwd(afwd), .bfwd(bfwd));

module forwarding(
    input [31:0] ctrlm, ctrlw,
    input [4:0] rdm, rdw, rs1, rs2,
    output [1:0] afwd, bfwd
);
assign afwd[0] = ctrlm[4] & (rs1 == rdm);
assign bfwd[0] = ctrlm[4] & (rs2 == rdm);
assign afwd[1] = ctrlw[4] & (rs1 == rdw);
assign bfwd[1] = ctrlw[4] & (rs2 == rdw);
endmodule
```

这个模块通过对比不同阶段的 rd 与 rs，来输出选择的控制信号。在 MEM 与 WB 处同时检测到需要前递时，实际上应该连续两次发生 MEM 处的前递，因此 fwd[0]应优先采用。为此，三选一 mux 对应设定为，1 与 3 时都输出 1 时的结果:

```
assign result = choose[0] ? one : (choose[1] ? two : zero);
```

再将 a 与 b 连入三选一 mux 即可:

```
mux3 amux(.zero(a), .one(y), .two(wd),
    .choose(afwd), .result(areal));
mux3 bmux(.zero(b), .one(y), .two(wd),
```

```
    .choose(bfwd), .result(breal));
```

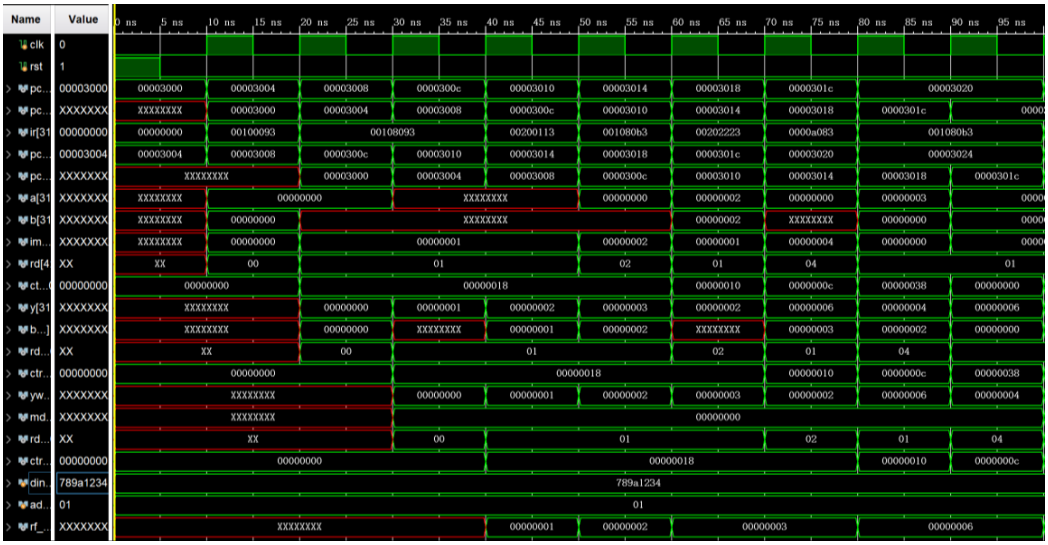
4、仿真测试

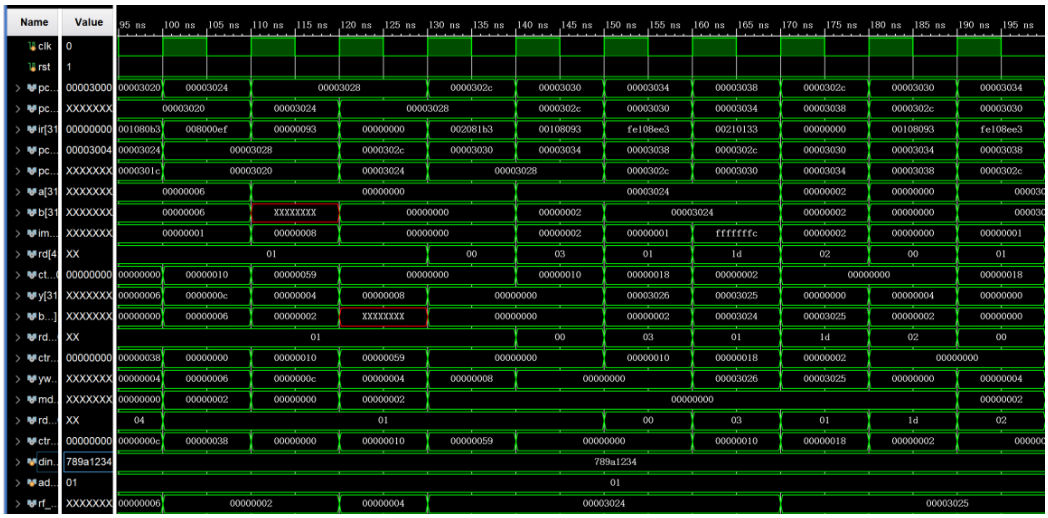
为仿真测试冒险的解决，写了如下的汇编文件：

```
.text
addi x1, x0, 1
addi x1, x1, 1
addi x1, x1, 1
addi x2, x0, 2
add x1, x1, x1
sw x2, 4(x0)
lw x1, 0(x1)
add x1, x1, x1
jal x1, l1
addi x1, x0, 0
l1: add x3, x1, x2
l2: addi x1, x1, 1
beq x1, x1, l2
add x2, x2, x2
```

前三行用来测试连续进行 MEM 处的前递，第 5 行为 WB 处的前递，7,8 行为 load-use hazard，9,13 行为 jal 和 beq 的采用，且测试了所需的六条指令。最终的仿真结果如下，由于 m_rf_addr 设置为了 1，观察 x1 的值的变化即可发现仿真结果正确，同时可根据过程中不同流水段寄存器的变化来确认结果的正确性(例如在 jal 指令后 ir 与 ctrl 被清零，从过程上看 pc 前进两周期后进行跳转，而 load-use hazard 发生时 pc 延迟了一个周期不变)。

此外，烧写测试的过程仍然为编写 top 模块连接 CPU 与 PDU，并且仍然使用老师提供的 fib_test 文件，于是看到的烧写结果实际上与上一个实验中完全相同，由此不再展示，仅展示仿真的结果。





总结：

这次实验实际上比预想中进行地顺利。实际上，最大的难点在于 `jal` 和 `io` 的数据通路需要自己完成，而前递、停顿与清除的过程由于已经有现成的数据通路设计，只需要按照通路完成即可。不过，事实上也有不少细节会造成影响，例如第一部分说到的写优先、赋初值等操作，都可能导致执行结果出现问题。好在中途的寄存器都已经接入了 `DEBUG_BUS`，通过对数据的观察可以及时找到问题所在，

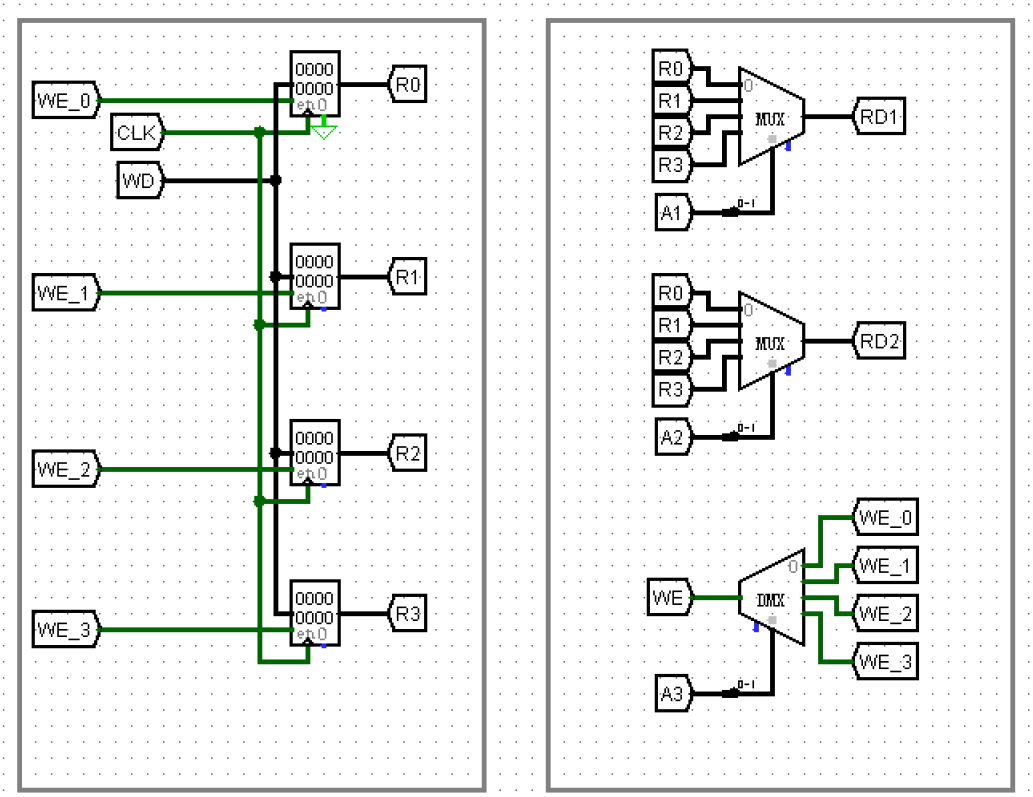
6-综合设计 实验报告

PB20000296 郑滕飞

单周期 CPU：

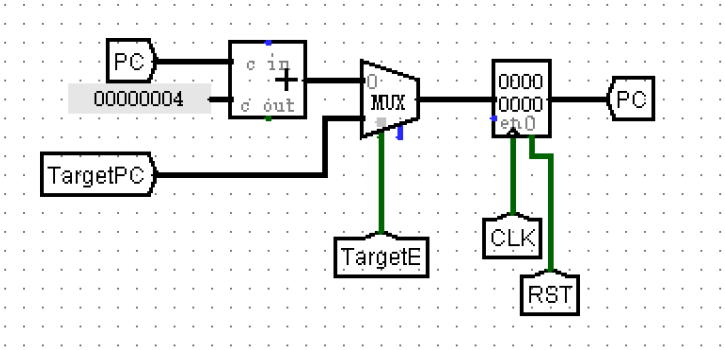
1、寄存器堆(Exp1)

将左侧的寄存器堆部分如图所示补全即可，根据测试样例的要求，需要保证 R0 恒为 0，因此 R0 的异步复位需要接入 1，其余直接对应接口即可：



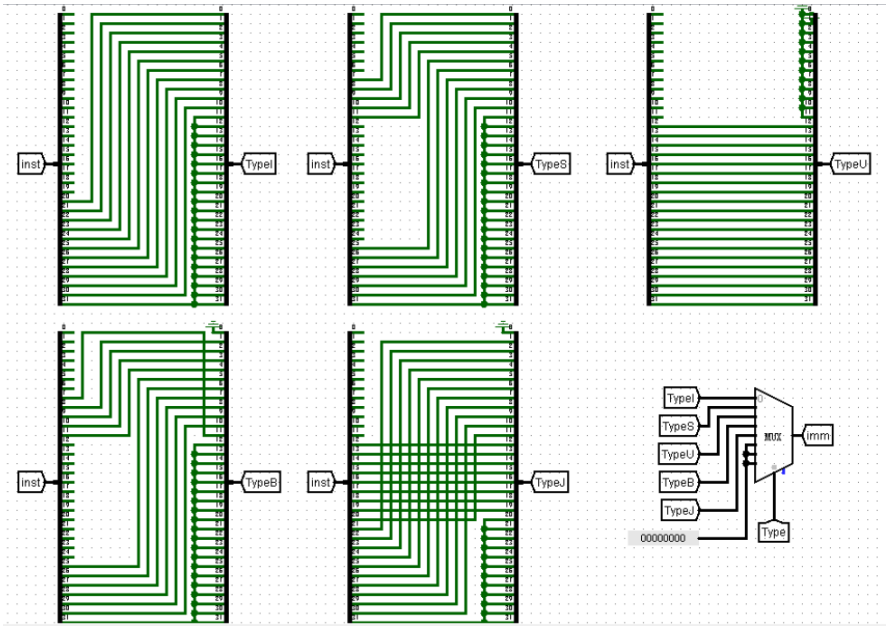
2、PC(Exp2)

根据提示，补充 MUX 选择 PC+4 与 TargetPC 信号，再增添寄存器存储 PC 值即可：



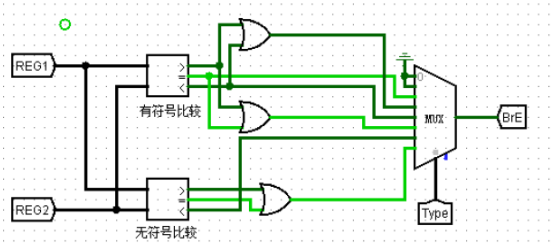
3、立即数生成器(Exp3)

根据提示，按照不同立即数类型生成立即数即可。立即数为 0 部分直接连接地线：



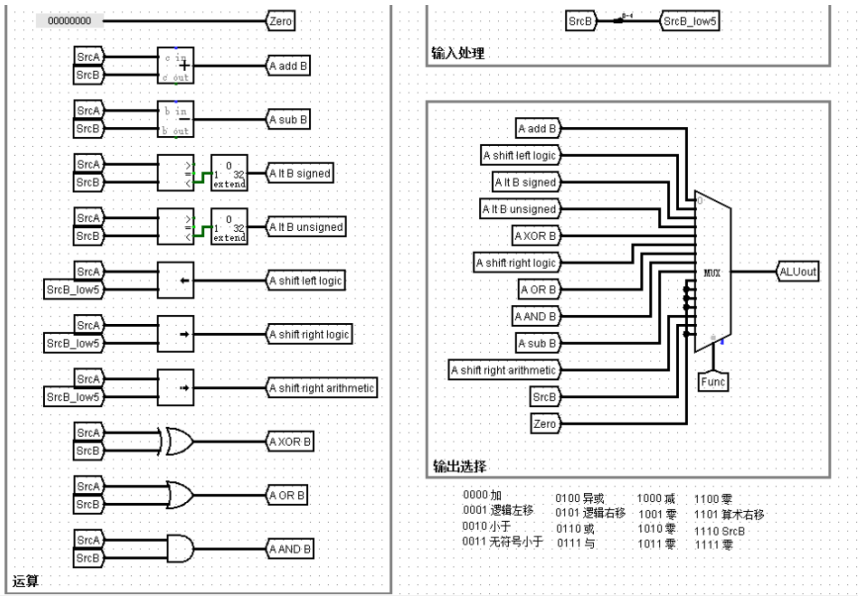
4、分支判断器(Exp4)

根据提示，按照分支指令不同类型对应连接即可，通过或门来处理两种情况的情形：



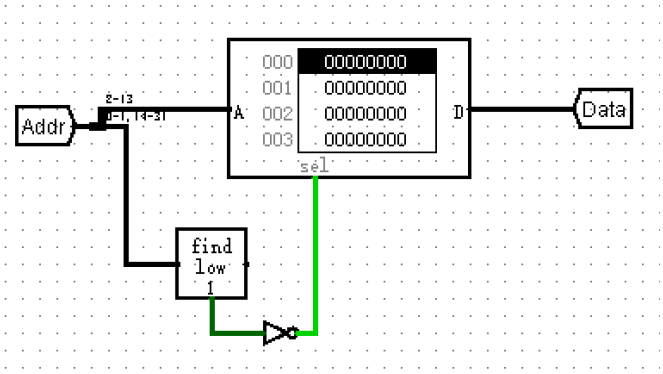
5、ALU(Exp5)

左侧根据运算的要求选择合适的算术单元进行操作，右侧根据选择的结果连线即可：

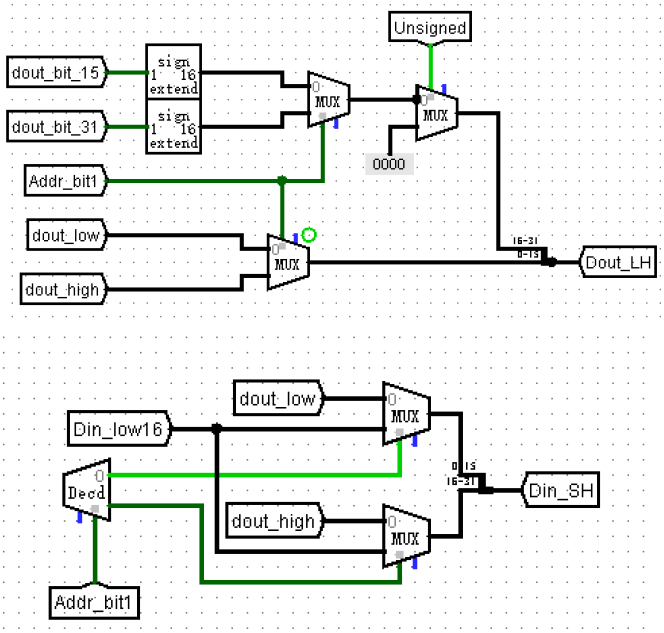


6、储存器(Exp6)

对指令储存器，验证有效后将需求地址接入 ROM 即可：

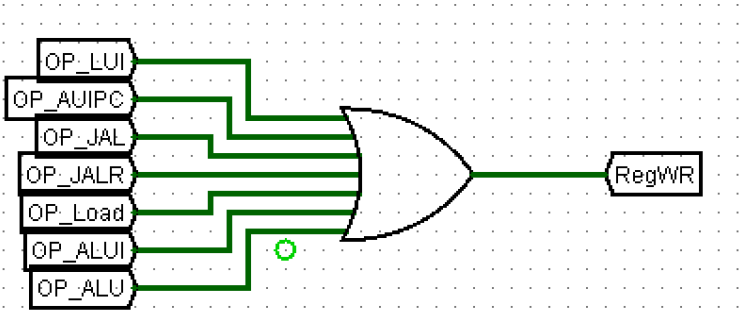


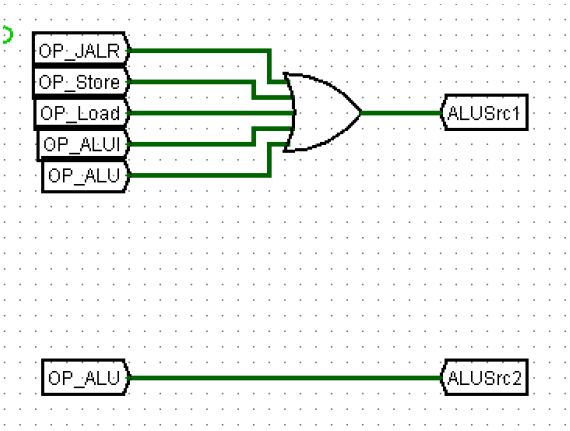
对数据储存器，根据提示补全 LH 与 SH 的处理即可：



7、控制单元(Exp7)

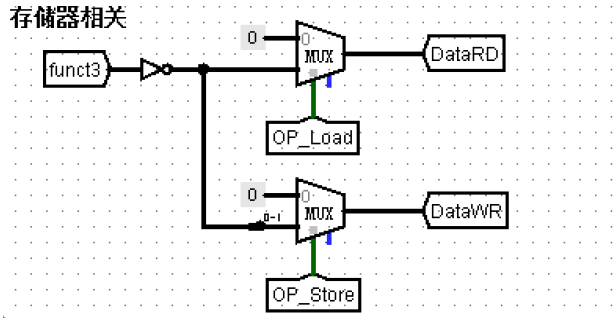
需要按照要求完成各控制信号的处理。寄存器写使能、ALU 两个操作数的选择均直接对需求取或即可：





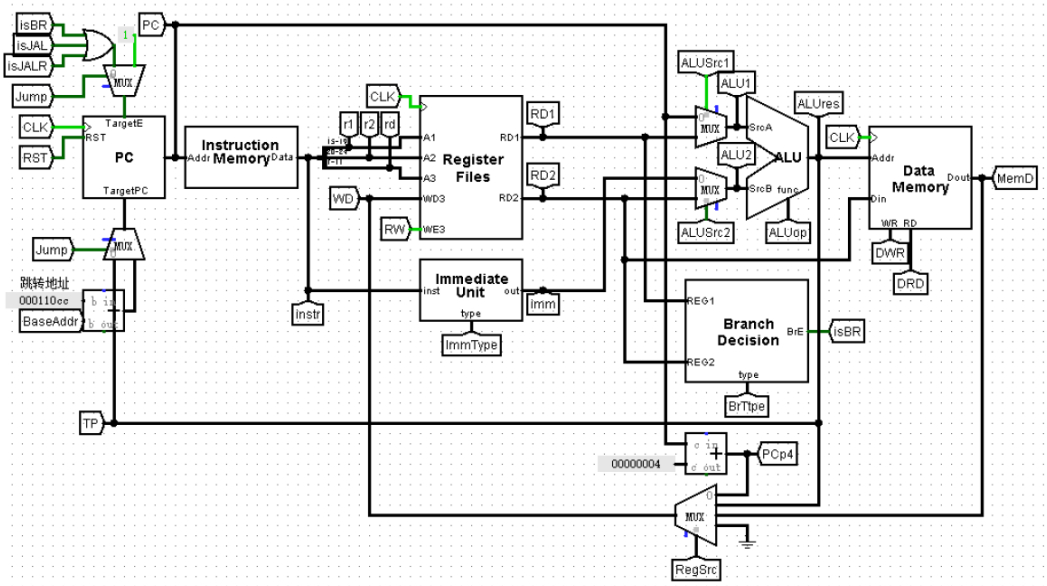
跳转相关直接对应连接 OP_JAL 与 OP_JALR 即可。

存储器相关按照要求进行按位取反，在读取时只保留低两位：



8、完整通路(Exp8)

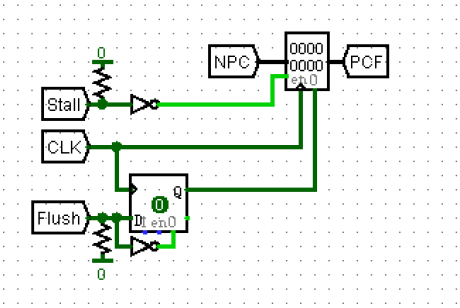
需要按照要求完成通路即可，根据控制单元中对 ALU1Src、ALU2Src 与 RegSrc 信号的介绍可知 CPU 中的对应连线方式：



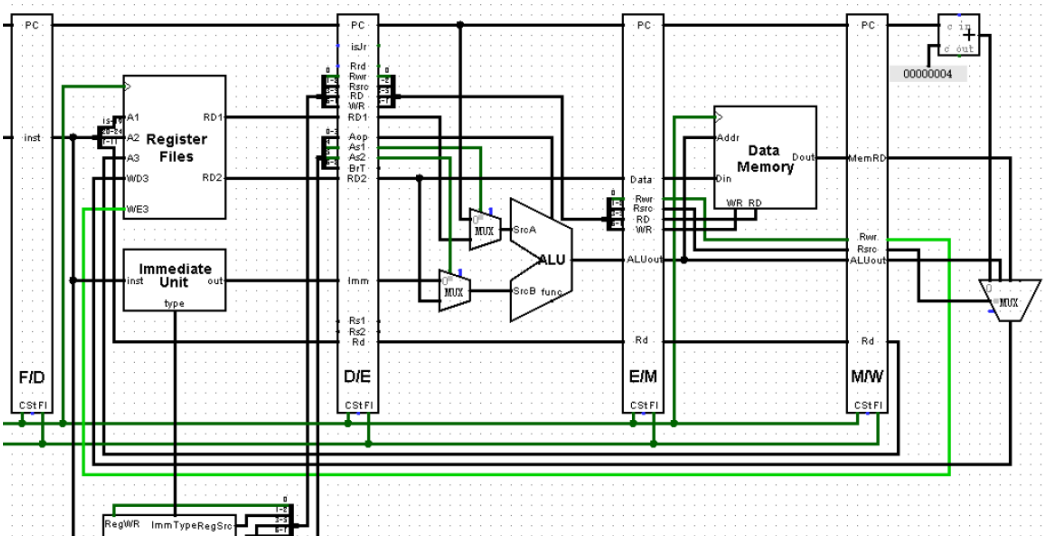
流水线 CPU：

1、理想流水线(Exp9)

PC 部分类似段间寄存器构造即可：

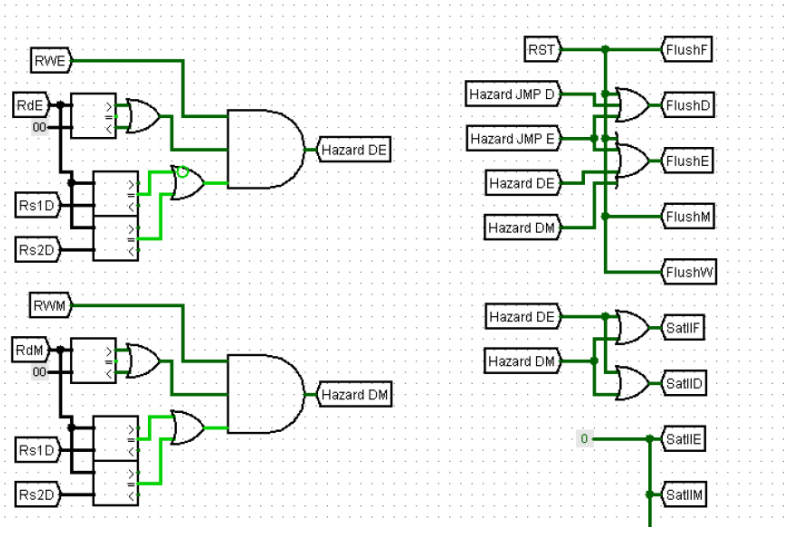


流水线部分根据数据通路连接即可，由于为理想流水线，不需要处理分支与跳转信号，也不需要将 Rs1 与 Rs2 存储 D/E(此为冒险处理单元使用)：

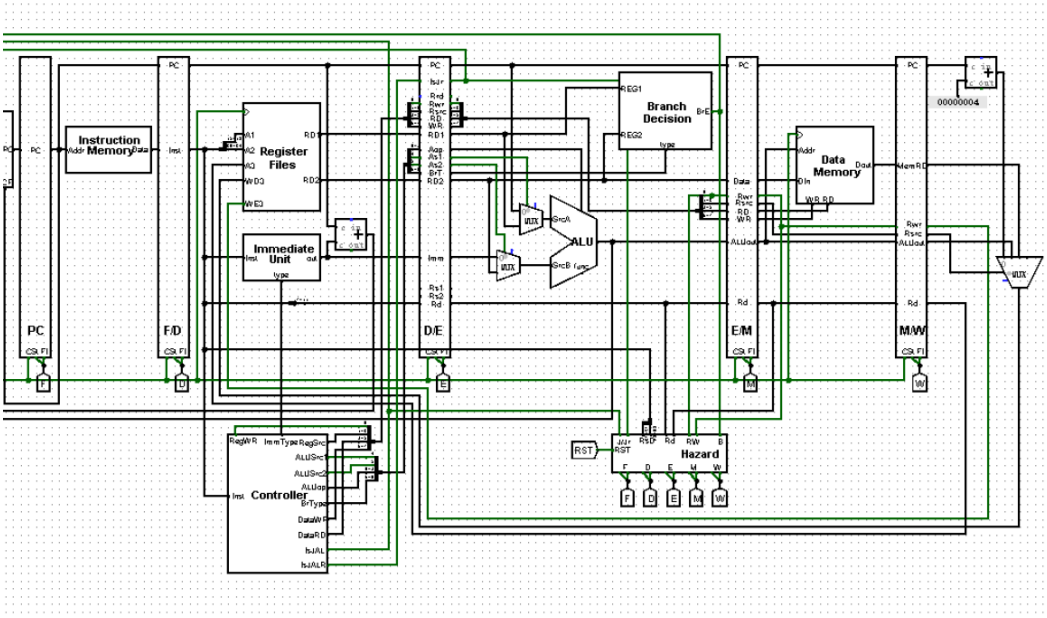


2、气泡流水线(Exp10)

根据提示可连接冒险检测模块的电路：

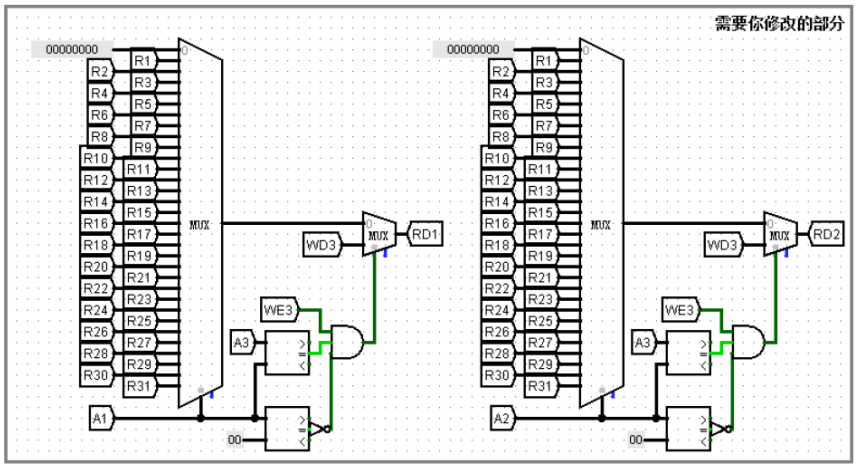


再连接整体电路即可：

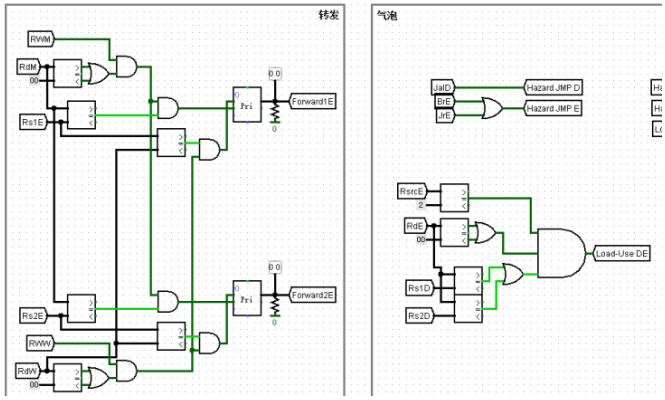


3、转发流水线(Exp11)

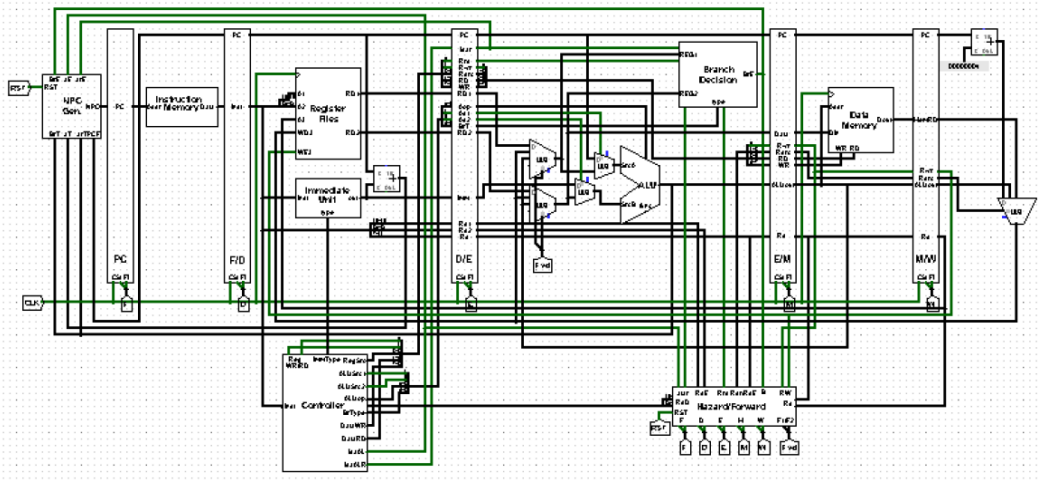
寄存器文件需要改为写优先的形式：



冒险与前递单元按要求生成信号：



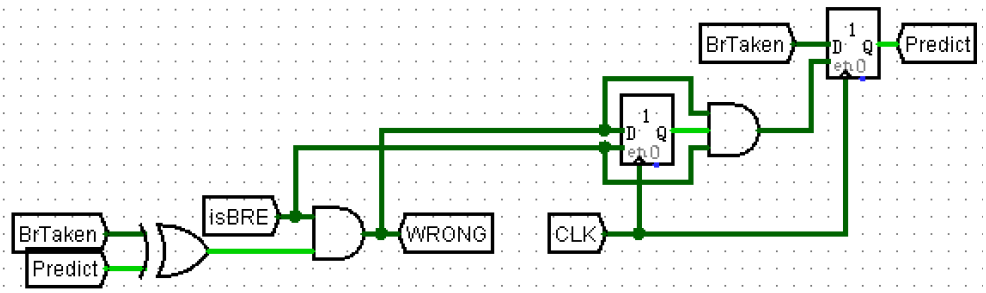
复制控制信号后，按要求连接整体电路即得最终结果：



分支预测：

1、状态机设计

我们希望实现的分支预测是：每当连续两次预测错误，将预测修改为当前的结果，否则维持原有预测。为此设计了如下电路：

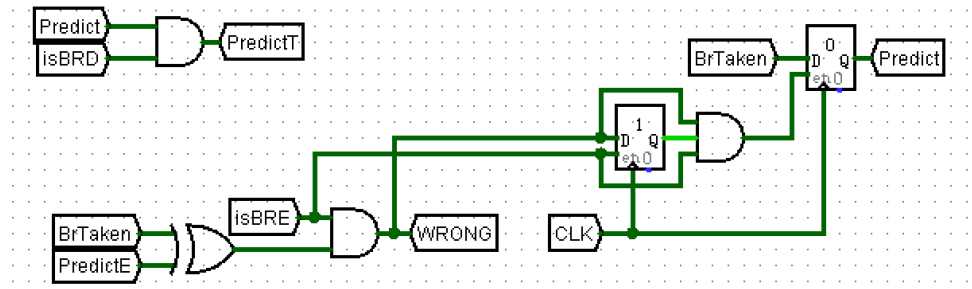


isBRE 代表对指令是否是 Branch 的判断，Predict 代表预测的结果，也是右侧寄存器中存储的值，BrTaken 代表实际的结果，CLK 则为时钟信号。

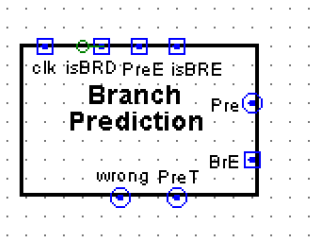
单次预测的正误当 isBRE 有效时，通过 BrTaken 与 Predict 是否相同进行判断，若相同则代表正确，否则为错误，每当 CLK 上升沿进行检测，isBRE 有效时，中间的寄存器即储存上次预测的结果。当且仅当上次预测的结果、这次预测的结果均错误时，下次将预测此次跳转的结果。

2、分支预测模块

实际的分支预测模块实现如下：



由于实际预测是在 ID 阶段，而判断预测是否正确是在 EX 阶段，需要利用段寄存器将输出的 Predict 带到 EX 阶段后再输入为 PredictE 以判断正误。此外，Predict 只表示分支时预测是否跳转，而真正跳转需要结合 ID 阶段确定是否为分支指令的 isBRD 信号，再输出真正控制跳转的 PredictT 信号。

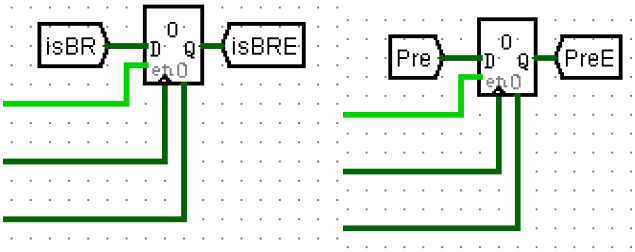


模块外观如上，时钟信号，D、E 阶段的是否为 BR、E 阶段的是否 Taken(BrE)作为输入，输出预测结果 Pre，跳转信号 PreT 与预测错误的信号 wrong。

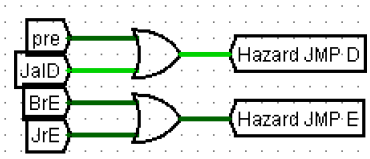
3、控制信号相关调整

首先，Controller 中需要添加 isBR 的输出，作为判断是否有分支的依据：

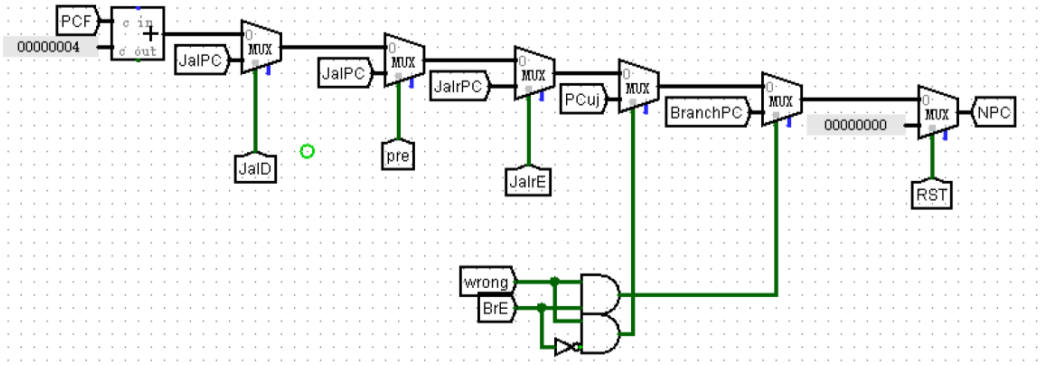
在 ID 阶段与 EX 阶段均需要判断，因此 ID/EX 段寄存器也需要增添 isBR 与 isBRE，而由于需要判断结果正误，预测结果也要通过此段寄存器：



对于 Hazard/Forward 单元，预测跳转事实上产生的效果等同于 jar，而预测错误产生的影响与 BrE(事实上可以看作预测不跳转但跳转)完全相同，因此如此修改即可。



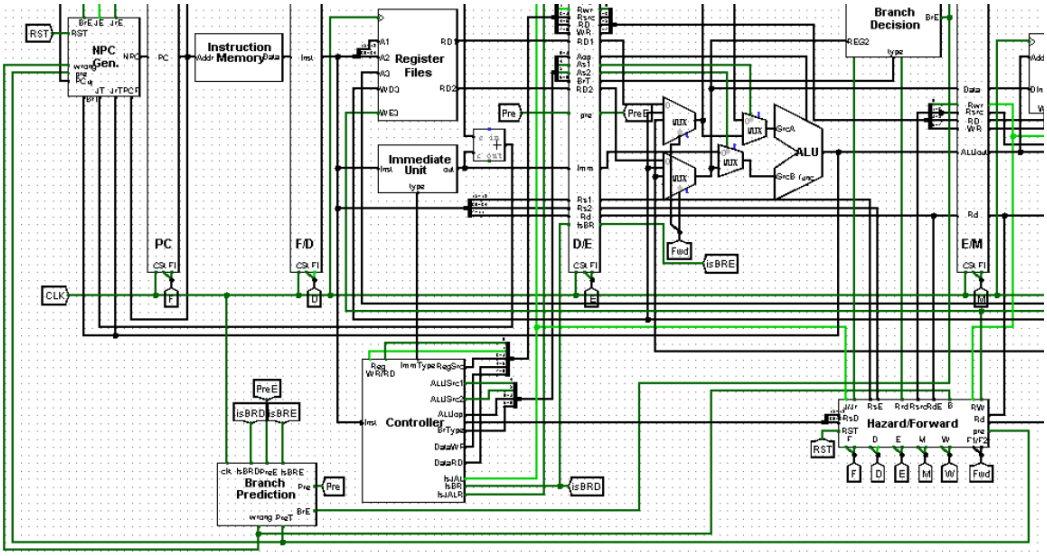
较为复杂的是 NPC 生成器相关的调整：



当发现预测错误时，根据实际情况是否跳转选择 PC 跳转的结果，若无预测错误，先考虑 EX 阶段 Jalr 形成的跳转，若无则代表 EX 阶段不存在跳转，考虑 ID 阶段，JalD 与预测结果 pre 均可能引起跳转，跳转结果实质上都是 PC+Imm 计算的结果，即 JalPC。

4、整体电路

如图所示连接整体电路：



将 Controller、Hazard/Forward、ID/EX、NPC Gen.模块增加端口后，与 Branch Prediction 对应端口连接，即可得到最终电路。

思考题：

1、怎样减少分支延迟槽，减少后 Hazard Detection 和 Forwarding 有何变化？

提前分支的判断时间，如将 Branch Decision 从 EX 提前到 ID，此时采用时发生的 Hazard 与 Jal 指令相同，而如果 Branch 与 Load 发生 Hazard 则比 LU Hazard 需要多暂停流水线一次，且前递单元需要检测 A1,A2 与 EX/Mem/WB 阶段是否相同，并前递到 ID 处以供判断。

2、在原本流水线结构中，Branch 的优先级比 Jal 高。现若在流水线中加入分支预测，Branch 指令在 IF 阶段就可以跳转（假设预测跳转）。假设现在有一条 Jal 指令在 ID 阶段，有一条 Branch 指令在 IF 阶段，此时会导致执行顺序位于前面的 Jal 指令跳转被忽略，如何解决这个问题？

既然已经可以在 IF 阶段实现跳转，可以直接将 Jal 的跳转也提前到 IF 阶段，如果不允许提前 Jal 指令的跳转，则跳转的优先级必须设置成越往后流水线越优先，即 EX 跳转优先于 ID 跳转优先于 IF 跳转。

3、我们在 ID 阶段处理 Jal 跳转，在 EX 阶段处理 Jalr 和 Br 跳转以及 Load-use 冒险为什么他们有处理阶段的差别？是否可以将某些处理放在更前的流水段进行？如果可以，应该怎么做，Forwarding 模块应该如何修改，效率会不会提升？

Jalr 与 Br 需要寄存器堆中读出的值，Jalr 需要 ALU 而 Br 需要 Branch Decision 模

块，且三者都需要立即数生成的模块的结果，因此 Jal 在 ID，Jalr 与 Br 在 EX。最早可把 Jal 提前到 IF(利用单独的立即数生成器)，Br 提前到 ID(如思考题 1 答案所示调整前递单元)，Jalr 提前到 ID(利用单独的 rs1 与立即数加法器)，由于减少了停顿，可以提升效率。

4、在单周期 CPU 中，如果将指令存储器的容量增大到 64KB，有哪些地方需要改动？尽量给出具体的值。

(参考 Exp6 实验文档)64KiB 即 16 位，从 0x00000000 到 0x0000fffc，因此指令储存器中的 find low 应为 31-16,0-1，同时 ROM 的地址位宽应改为 14，连接到 Addr 的 2-15 位。

5、在流水线 CPU 中，当 ID 和 WB 阶段需要读写同一个寄存器时会存在数据冒险，给出两种解决该数据冒险的方法。

(参考 Exp10、Exp11 实验文档)让寄存器堆在时钟下降沿有效，从而前半周期写入、后半周期读出，或在寄存器堆文件中添加转发(即将寄存器堆改为写优先。)

总结：

开学初速成 Verilog，实验结束前速成 Logisim，表面只选了一门计组，事实上连带着学完了数电的一学期终于要结束了。硬件的东西确实和软件是完全不同的体验，debug 流程更加麻烦、需要注意的东西更多，不过真的搭出个东西后也有种奇妙的成就感——虽然如此，以后肯定是不会再选体系结构了，实在是做不动。

Logisim 的优点是更加直观形象，但是具体电路需要自己搭建(尤其是布线)也增加了不少复杂度。因此，二选一的话，在复杂任务上还是硬件描述语言更合适。不过，就像实现分支预测的时候用 Logisim 发现的各种问题，如果当时我直接选择用 Verilog 去写，很多段寄存器上的隐秘问题大概更难发现。

最后，感谢助教这学期的付出与对我的帮助^__^

COD LAB 问题汇总

LAB 之外:

*尽早提醒前置需要学习 verilog、安装 vivado，给退课/速成提供机会
*提供更好的自学方式(verilogoj、教程、数电实验文档、一些特殊情况解决方案的链接等)
*与 Logisim 实验的搭配(如果能将 LAB1-3 替换或部分替换为 Logisim 实验，之后可以减少大量负担)
*提醒有时问题是来自 vivado，如果实在找不到错但跑不通可以尝试重新烧写/让别人跑
*减轻报告负担

LAB 1:

建议顺序：verilog 基础与注意 -> 32 位 ALU 任务 -> 查看 RTL、综合电路、资源利用 -> 6 位 ALU 任务 -> **xdc 文件与烧写** -> 查看时间报告 -> **Moore 型状态机** -> FLS 的设计
ERR: 32 位 ALU 组合逻辑，不应有查看时间报告的任务
ERR: 6 位 ALU 实际上并没有分时复用，用词导致误解
*提醒选用的芯片型号
*说明查看 RTL、综合电路、资源利用的含义与**动机**
*六位 ALU 任务解释清楚 当前应做 与 接入时钟后可进行分时复用
*解释 xdc 文件与烧写，并且提醒时间报告需要调整的 xdc
*时间报告的形成条件：使用 clk 且入口出口都有寄存器形成**通路**
*对状态机的基本讲解
*Moore 型状态机可提供 verilogoj 上的题目解释
EXTRA: FLS 只利用加法，实际上不适合 ALU 的测试，是否有更好的选择？

LAB 2:

建议顺序：例化 IP 核与 coe -> 不同运作方式的对比 -> 寄存器堆编写 -> FIFO 总任务 -> **an,seg 工作原理** -> **队列基本原理** -> 三模块的结构与通路 -> 主控模块出入队的提示
ERR: 例化 IP 核的操作流程截图中有的选项并不是真实应选的，产生迷惑
*例化 IP 核的部分强调**不需要自己实现，只需要编写 top 例化**
*运作方式对比中提供结构图，需要解释为何 block 的读会延迟一个周期(锁存器)
*寄存器堆编写中出现的 RF 种类太多，应明确说明哪个是需要实现的
*寄存器堆中编写的 RF 和 FIFO 中运用的不同，建议直接双参数
*FIFO 应解释清楚 an 与 seg 分时复用的工作方式
*适当补充队列与循环队列基本原理
*主控模块实现提示可以更加清楚，尤其是循环队列的头尾
*提醒非阻塞赋值的特点，操作后头尾相等代表操作前相差 1
EXTRA: FIFO 中用状态机有害无利，是否不应增加此限制？

LAB 3:

建议顺序：复习汇编 -> RIPES 使用与示例代码 -> **RARS 使用** -> 测试代码 -> FLS 代码
ERR: 此处编写的测试代码测试的指令与下个实验不对接
ERR: Dump to File 前应该先调整结构设置，保证出来的 coe 是下个实验直接可用的
*RIPES 部分强调对数据通路的观察
*添加 java 环境安装与 RARS 使用部分
*提醒自己编写代码时限制指令的使用，不要用 ecall 等

*不要使用 in、out、led 等说法产生迷惑，此实验只编写汇编(尤其是此实验的 IO 与下个实验不同)
*FLS 说清楚需要的输入方式与输出方式
EXTRA: 适当与下个实验对接，如提示考虑使用数据外的地址作为 IO 等操作

LAB 4:
建议顺序：复习汇编机器码 -> CPU 模块概述 -> **IO_BUS 解释** -> 数据通路与提示 -> PDU 模块概述 -> **PDU 具体代码** -> 不涉及 IO 的测试 -> 利用 IO 的测试
ERR: 提供的数据通路无法完成十条指令，如果不想提供完整的数据通路，可以适当挖空
ERR: ppt 中外设部分的地址对应 io_addr 的数值与 pdu 中不一致，按 ppt 说法 io_addr 应连接 alu_out[9:2]，但实际为 alu_out[7:0]
*地址的 8bits 与寄存器的 32bits 对应方式应讲解清楚
*对 IO 的讲解和实现应更加清楚
*数据通路可考虑分块解释，而不是直接一张图
*数据通路最好合并在 ppt 内
*建议先给出不含 IO 部分的完整通路再解释 IO 连线
*如果需要写 PDU 模块，可考虑延长时间以代码填空形式
*如果不需要写 PDU，建议在 ppt 补充结合代码的更详细讲解
*测试时最好给出一些 debug 方面的建议(例如将 ALU 的结果接入 DEBUG_BUS、拖动查看仿真)
EXTRA: 更好的数据通路与模块设计(可参考 Logisim 实验的数据通路)

LAB 5:
建议顺序：复习流水线知识 -> 段寄存器-> 理想流水线 -> 停顿与清除(跳转/Load use) -> 数据前递 -> PDU 简介 -> 外设使用方式
ERR: 提供的数据通路和端口无法完成 jal，需要改进
ERR: RIPES 的数据通路中 Flush 信号应接到 clear 而非 enable
*由于之前实现为读优先，寄存器堆改为写优先应在数据通路前强调
*提醒中间模块的控制信号应设初始值为 0，保证前几步的正确运行
*段寄存器的复杂而重复的实现可以考虑给出完整接口(或代码)以方便操作
*提醒在段寄存器中预留 stall 与 flush 的接口
*提示考虑 IO 部分的阶段(MEM)从而确定连线
*提示利用查看段寄存器的方式 debug
EXTRA: 一些改进操作(如在 ID 阶段执行 jal 等)
EXTRA: 更好的数据通路与模块设计(可参考 Logisim 实验的数据通路)

LAB 6:
*可以考虑提供更详细的实现方式与参考(如 CPU 改进可根据体系结构实验文档改编、应用程序可根据数字电路实验文档改编等)
*在多种选择时给出相对公平合理的评分标准，方便统一