



2018

## 计算机组成原理

## 课程设计报告

题 目： 5 段流水 CPU 设计

专 业： 计算机科学与技术

班 级： ACM1501

学 号： U201514545

姓 名： 胡学仕

电 话： 13006369675

邮 件： hubachelor@gmail.com

完成日期： 2012-04-08 周日下午



计算机科学与技术学院

# 华中科技大学课程设计报告

---

## 目 录

<b>1 课程设计概述.....</b>	<b>3</b>
1.1 课设目的.....	3
1.2 设计任务.....	3
1.3 设计要求.....	3
1.4 技术指标.....	4
<b>2 总体方案设计.....</b>	<b>6</b>
2.1 理想流水 CPU 设计.....	6
2.2 中断机制设计.....	6
2.3 流水 CPU 设计.....	7
2.4 气泡式流水线设计.....	7
2.5 数据转发流水线设计.....	8
2.6 动态分支预测机制.....	8
<b>3 详细设计与实现.....</b>	<b>9</b>
3.1 理想流水 CPU 实现.....	9
3.2 中断机制实现.....	15
3.3 流水 CPU 实现.....	17
3.4 数据转发流水线实现.....	19
3.5 动态分支预测机制实现.....	22
<b>4 实验过程与调试.....</b>	<b>33</b>
4.1 测试用例和功能测试.....	33
4.2 测试技巧总结.....	36
4.3 主要故障与调试.....	36
4.4 实验进度.....	43

# 华 中 科 技 大 学 课 程 设 计 报 告

---

---

5 设计总结与心得.....	45
5.1 课设总结.....	45
5.2 课设心得.....	45
参考文献.....	47

# 华中科技大学课程设计报告

---

## 1 课程设计概述

### 1.1 课设目的

计算机组成原理是计算机专业的核心基础课。该课程力图以“培养学生现代计算机系统设计能力”为目标，贯彻“强调软/硬件关联与协同、以 CPU 设计为核心/层次化系统设计的组织思路，有效地增强对学生的计算机系统设计与实现能力的培养”。课程设计是完成该课程并进行了多个单元实验后，综合利用所学的理论知识，并结合在单元实验中所积累的计算机部件设计和调试方法，设计出一台具有一定规模的指令系统的简单计算机系统。所设计的系统能在 LOGISIM 仿真平台和 FPGA 实验平台上正确运行，通过检查程序结果的正确性来判断所设计计算机系统正确性。

课程设计属于设计型实验，不仅锻炼学生简单计算机系统的设计能力，而且通过进行中央处理器底层电路的实现、故障分析与定位、系统调试等环节的综合锻炼，进一步提高学生分析和解决问题的能力。

### 1.2 设计任务

本课程设计的总体目标是利用 FPGA 以及相关外围器件，设计五段流水 CPU，要求所设计的流水 CPU 系统能支持自动和单步运行方式，能正确地执行存放在主存中的程序的功能，对主要的数据流和控制流通过 LED、数码管等适时的进行显示，方便监控和调试。尽可能利用 EDA 软件或仿真软件对模型机系统中各部件进行仿真分析和功能验证。在学有余力的前提下，可进一步扩展相关功能。

### 1.3 设计要求

- (1) 根据课程设计指导书的要求，制定出设计方案；
- (2) 分析指令系统格式，指令系统功能。
- (3) 根据指令系统构建基本功能部件，主要数据通路。
- (4) 根据功能部件及数据通路连接，分析所需要的控制信号以及这些控制信

# 华中科技大学课程设计报告

- 号的有效形式；
- (5) 设计出实现指令功能的硬布线控制器；
- (6) 调试、数据分析、验收检查；
- (7) 课程设计报告和总结。
- ## 1.4 技术指标
- (8) 支持表 1.1 前 27 条基本 32 位 MIPS 指令；
- (9) 支持教师指定的 4 条扩展指令；
- (10) 支持多级嵌套中断，利用中断触发扩展指令集测试程序；
- (11) 支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；
- (12) 能运行由自己所设计的指令系统构成的一段测试程序，测试程序应能涵盖所有指令，程序执行功能正确。
- (13) 能运行教师提供的标准测试程序，并自动统计执行周期数
- (14) 能自动统计各类分支指令数目，如不同种类指令的条数、冒险冲突次数、插入气泡数目、load-use 冲突次数、动态分支预测流水线能自动统计预测成功与失败次数。

表 1.1 指令集

#	指令助记符	简单功能描述	备注
1	ADD	加法	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
2	ADDI	立即数加	
3	ADDIU	无符号立即数加	
4	ADDU	无符号数加	
5	AND	与	
6	ANDI	立即数与	
7	SLL	逻辑左移	
8	SRA	算数右移	
9	SRL	逻辑右移	
10	SUB	减	
11	OR	或	

# 华 中 科 技 大 学 课 程 设 计 报 告

#	指令助记符	简单功能描述	备注
12	ORI	立即数或	
13	NOR	或非	
14	LW	加载字	
15	SW	存字	
16	BEQ	相等跳转	
17	BNE	不相等跳转	
18	SLT	小于置数	
19	STI	小于立即数置数	
20	SLTU	小于无符号数置数	
21	J	无条件转移	
22	JAL	转移并链接	
23	JR	转移到指定寄存器	If \$v0==10 halt(停机指令) else 数码管显示\$a0 值
24	SYSCALL	系统调用	
25	MFC0	访问 CP0	中断相关, 可简化, 选做
26	MTC0	访问 CP0	中断相关, 可简化, 选做
27	ERET	中断返回	异常返回, 选做
28	SRAV	算术移位	
29	BLEZ	分支指令	
30	LH	加载半字	
31	SRLV	逻辑移位	

## 2 总体方案设计

### 2.1 理想流水 CPU 设计

#### 2.1.1 主要设计思想

理想流水的设计是课设开始的基础的，只有可以将原来的暴露在数据通路上面的各个控件重新封装成为的单独的模块，然后清晰区分每一个的阶段应该持有什么的数据的时候，之后的气泡和重定向等等才可以清晰的展开。

将 CPU 的架构的划分的五个区域，IF IM EXE MEM 和 WB 阶段，其中 EXE 阶段处理的事情计算出来 NPC。

#### 2.1.2 流水接口部件设计

流水接口部件需要实现缓存的数据，同时支持清空 和 暂停。

### 2.2 中断机制设计

#### 2.2.1 总体设计

中断的设计需要软硬件结合的方法，硬件识别中断保存中断信息，软件记录发生中断的地址，寄存器文件的现场的保护。

#### 2.2.2 硬件设计

使用优先编码器实现对于最高级中断的响应，对于每一个中断使用一个寄存器保存的该中断是从未执行的，还是执行过。

#### 2.2.3 软件设计

所有中断程序首先需要利用堆栈保存当前地址 和 即将使用过的寄存器数值

# 华中科技大学课程设计报告

---

## 2.3 流水 CPU 设计

### 2.3.1 总体设计

流水的设计首先将数据划分为五个阶段，然后处理数据冲突和控制冲突，控制的冲突需要处理的位置是气泡流水的设计和动态分支预测，虽然重定向也是处理数据冲突，但是由于和气泡几乎没有任何的区别。

控制冲突出现的原因是，默认的时候总是采用  $PC + 4$  的数值，但是只有到达的 EXE 或者 IM 段的时候才可以检查的出来当前指令是否是一个跳转的指令，这个时候，预先加入的指令将会是一条的错误的指令，如果计算 NPC 的位置是在 IM 段，那么需要清除一个预取的指令，但是如果在 EXE 阶段计算 NPC 的数值，那么需要清除两个数值。为了设计上面的方便，最终将 NPC 的计算位置放置到的 EXE 段，由于计算 NPC 的时候，对于 B 指令，需要分析从寄存器获取两个数值，重定向的时候，这两个数值的也是需要重新处理的，所以导致设计上复杂化，虽然让动态分支预测的失败成本增加。

数据冲突的原因为 RAW，也就是前面指令尚且没有更新的寄存器中间的数值，但是指令已经开始需要使用该寄存器的中间的数值，气泡处理策略是，绝对不会把错误的寄存器文件的输出数值流到下一个阶段，一旦发现冲突，暂停当前的流水线直到需要读入的数据被刷新了。对于重定向的思路是，既然只要防止错误的数据流入到下一个周期里面，那么在前面阶段计算出来了，那么使用多路选择器直接获取正确的数据即可，当然 load-use 还是需要导致的插入一个气泡，更加简单的一个解释是，IM 发现和 EXE 段数据发生冲突，但是 EXE 段数据需要在 MEM 段才可以计算出来，所以需要插入气泡，延迟一个周期。

## 2.4 气泡式流水线设计

对于控制冲突，采用清空流水线的操作，具体需要清空的那些位置数据取决于的计算 NPC 的位置，对于的数据冲突，采用插入气泡的，直到没有出现任何数据冲突，在数据冲突的分析上面和重定向非常的类似。

# 华中科技大学课程设计报告

---

## 2.5 数据转发流水线设计

添加两个控制单元，一个是产生重定向信号的 controller 和一个接受的 controller 信号实现选择正确数据的 handler。Controller 的接受参数为 IM 段的读入寄存器编号，和 EXE MEM 阶段写入寄存器文件，输出信号为对于数据的选择。Handler 除了接受控制信号，而且需要接受 EXE MEM 的输出的数据。

## 2.6 动态分支预测机制

在 IM 段对于 BHT 进行查询，在 EXE 段对于 BHT 表进行更新，BHT 表一共含有 8 个表项，每一个表项含有的内容为 pc npc LRU 计数 valid 位。LRU 算法实现的方法类似华莱士树计算最大值，当一个新数值需要使用的时候，最大值对应的项目将会被删除掉。插叙的方法的使用为全相连的方式，任何查询的数值都是需要和所有项目中 valid 的地址进行比对，如果查询到，那么根据统计信息来决定是否跳转。

## 3 详细设计与实现

### 3.1 理想流水 CPU 实现

#### 3.1.1 主要功能部件实现

##### 1) 程序计数器 (PC)

###### ① Logism 实现：

使用一个 10 寄存器实现程序计数器 PC，触发方式为上升沿触发，输入为下一条将要执行的指令的地址，输出为当前执行指令的地址。Halt 为停机信号，当出现 halt 信号的时候，由于控制对应的 enable 端，所以让指令暂停向下推进，如图 3.1 所示。

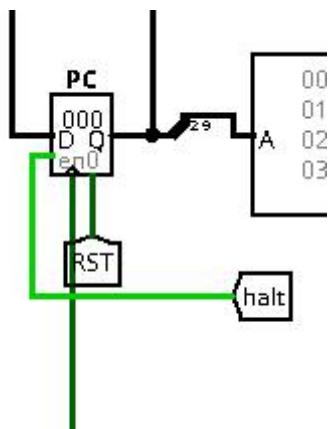


图 3.1 PC

###### ② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
'timescale 1ns / 1ps
module program_counter(
    input [11:0] pc_in,
    input clk,
    input rst,
    input enable,
    output reg [11:0] pc
);

initial begin
    pc = 0;
```

# 华中科技大学课程设计报告

```
end
always @(posedge clk) begin
    if(rst) begin
        pc <= 0;
    end else if(enable) begin
        pc <= pc_in;
    end
end
endmodule
```

## 2) 指令存储器 (IM)

- ① Logism 实现：由于 IM 的存取的时候一定会使用按照字节访问，所以需要对于 pc 的数值，将其中的低两位忽略掉来作为访问地址的方法，如图 3.2 所示。

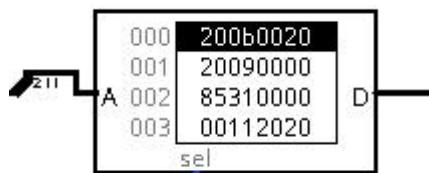


图 3.2 指令存储器 (IM)

## ② FPGA 实现：

首先的将编译二进制代码存储在一个特定的文件夹中间，然后在使用该文件的路径作为参数写入，那么可以立刻自行含有该数据

指令存储器 IM 的 Verilog 代码如下：

```
`timescale 1ns / 1ps
module IM(
    A,
    D_out
);
    input [9:0] A;
    output [31:0] D_out;
    reg [31:0] data [1023:0]; // 256x31 data
    integer i;
    initial begin
        for (i=0;i<1024;i=i+1) data[i] = 'h00000000;
    end
    initial
$readmemh("/home/martin/X-Brain/sys_design/documents/cc/verilog/test/benchmark_ccmb.hex", data);
    assign D_out = data[A];
endmodule
```

直接调用之前设置的 ROM 作为指令存储器，输入为指令地址的 2-11 位，输出为该指令。

### 3) 数据存储器的实现 (DM)

① Logism 实现：由于 IM 的存取的时候一定会使用按照字节访问，所以需要对于 pc 的数值，将其中的低两位忽略掉来作为访问地址的方法，但是有些指令会进行特殊存储操作，需要使用 sel 实现处理，此处使用模块为第三方提供的文件。如图 3.3 所示

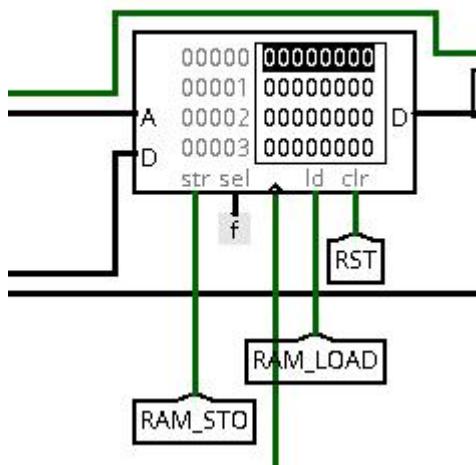


图 3.3 数据存储区(DM)

③ FPGA 实现：需要使用申明数组方法，对于访问的时候，只要提供的一个地址即可，由于含有展示存储器数值的要求，所以需要添加一个参数用于的获取任意的参数位置，由于使用更加大数组会导致出现综合时间非常长，而且也没有必要使用过大的数组，所以数组的大小仅仅是为  $2^{**} 6$  的大小

```
'timescale 1ns / 1ps
module DM(
    _rA, _rB, D, WE, mode, clr, clk,
    A_out, B_out);
    input [5:0] _rA, _rB; // rA is read-write but rB is read-only (for
display)
    input [31:0] D;           // D will be wrote at address 'rA'
    input [1:0] mode;
    input WE, clr, clk;
```

# 华中科技大学课程设计报告

```
output reg [31:0] A_out, B_out;  
  
reg [31:0] data [0:63]; // 64x31 data  
  
integer i;  
  
wire [5:0] rA, rB;  
  
assign rA = _rA;  
assign rB = _rB;  
  
initial begin  
    for (i=0; i<64; i=i+1) data[i] <= 'h00000001;  
    A_out <= 0;  
    B_out <= 0;  
end  
always @(posedge clk or posedge clr) begin  
    if (clr) begin  
        for (i=0; i<64; i=i+1) data[i] <= 'h00000000;  
    end  
    else if (WE) begin  
        case (mode) // write  
            2'b00: data[rA] <= D;  
            2'b01: data[rA][15:0] <= D[15:0];  
            2'b10: data[rA][7:0] <= D[7:0];  
            default: data[rA] <= D;  
        endcase  
    end  
end  
always @(*) begin  
    A_out = data[rA];  
    B_out = data[rB];  
end  
  
endmodule
```

# 华中科技大学课程设计报告

## 3.1.2 数据通路 和 控制器的实现

数据通路和控制器的两者是出现是部分前后，一次将所有的数据通路完成，使用的一个表格来实现对于所有指令的设计到数据通路实现分析，如图为对应的表格项目的部分数据如图 3.4 和图 3.5 所示。

OP	funct	rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MUX-3	
	R型指令	0	0	0	1	0	0	&&&&&	1	0	
100000	add							0101			
100001	addu							0101			
100100	and							0111			
100010	sub							0110			
100101	or							1000			
100111	nor							1010			
101010	slt(set less than)							1011			
101011	sltu(set less than unsigned)							1011			
		rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MUX-3	
000110	srl(可变)	0	3	3	1	0	0	00010	1	0	
000111	sra(可变算数右移)							0001			
000000	sll	0	3	*	1	0	1	0000	1	0	
000011	sra							0001			
000010	srl							0010			
I型号指令(rs rt 计算, 然后使用16bit)											
		rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MI	blez beq bne
000110	blez(小于*)	0	0	0	*	0	*	0	1	0	1
000100	beq										1
000101	bne										1
	rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MUX-3		
001000	addi	3	0	*	1	0	3	0101	1	0	
001001	addiu							0101			
001100	andi							0111			
001101	ori							1000			
001010	slti							1011			
	Memory	rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MI	RAM RAM Half_y

图 3.4 数据通路表第一部分

# 华中科技大学课程设计报告

100001	Ih(存储)	3	0*		1	3	30101	1	0	0	1	1
100011	lw	3	0*		1	3	30101	1	0	0	1	0
101011	sw	*	0*		0*		30101	1	0	1	0*	
	J 型号指	rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MUX-3		
000010	j label	*	*	*		0*	*	*	0	0		
000011	jal label		1*	*		1	1*	*	0	0		
001000	jr	*		0*		0*	*	*	*	1		
	Special											
	周期数											
	有条件跳转											
	无条件跳转											
	成功有条件跳转											
	rW	Ra	Rb	WE	w	Y	ALU-S	PC-N	PC-MI	syscall		
001100	syscall	*		1	1	0*	*	*	1	0	1	

图 3.5 数据通路表第一部分

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理图如图 3.6 所示，其中的鸭绿色长条的为缓冲的区间，在 IM 段的包含的部分为计算 NPC 和显示的数据的位置。

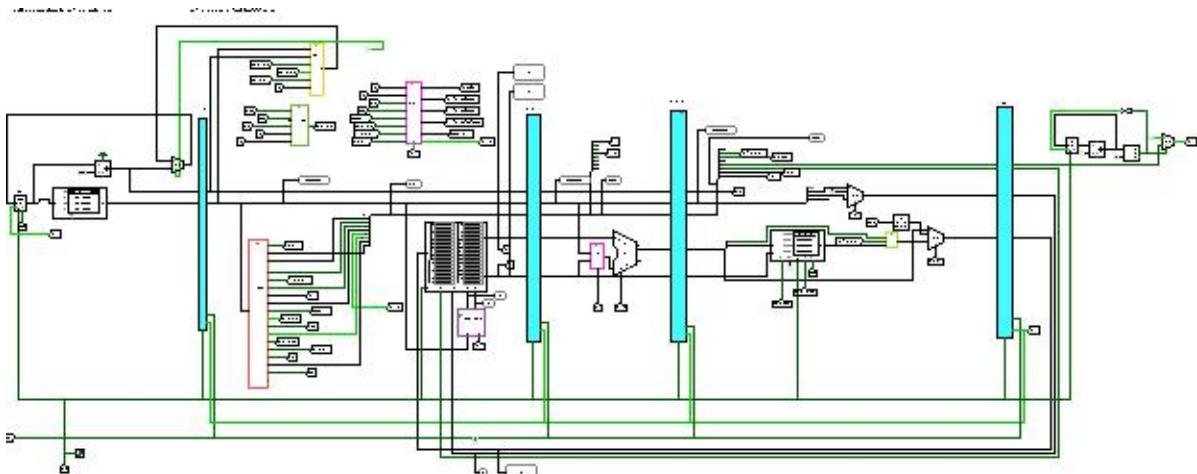


图 3.6 数据通路 logisim 整体的架构图

构建数据通路是 FPGA 中间的最复杂地方，第一是工作量巨大，为转化成为一个 FPGA 代码，首先需要为所有线全部的命名，导致申明出来的变量非常之多，而且

出可以出现任何错误，第二个问题在于需要出现问题没有办法确定是当前模块的问题还是引用的子模块的问题，前仿真调试难以进行。

## 3.2 中断机制实现

### 3.2.1 高优先级的打断低优先级实现

使用优先编码器和一个比较器可以实现的当高优先级的中断的出现的时候，那么当前中断会停止的执行，原理是，如果出现高优先级的中断，那么的优先编码器的输出数值必定比当前的数值更加大，比较器显示原来数值提升，然后保存 pc，开始出现中断，对应的 logisim 的实现如图 3.7 所示

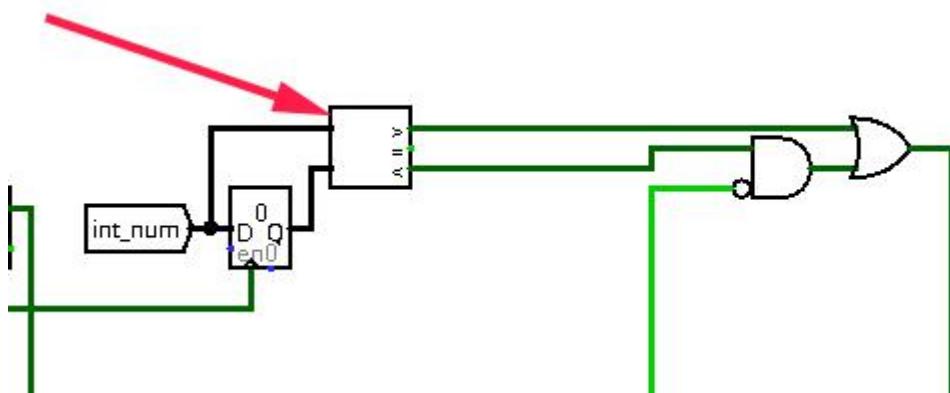


图 3.7 高级别中断信号检查结构

### 3.2.2 中断的返回的实现

对于的多级中断中间，如果中断的顺序依次按下是 2 3 1，那么当 3 执行的结束之后，对于的 2 是返回，而 2 结束之后，不是返回，而是采用中断，所以的对于什么时候采用的中断，什么时候为返回需要使用一个寄存器进行存取，当前中断返回之后，所执行中断是否已经执行过，对应 FPGA 的实现如图 3.6，其中 one\_exe 的型号表示中断 1 是否执行过，如果没有执行过，那么当高级中断开始结束之后，该中断开始进行，如果执行过，那么该中断继续进行。

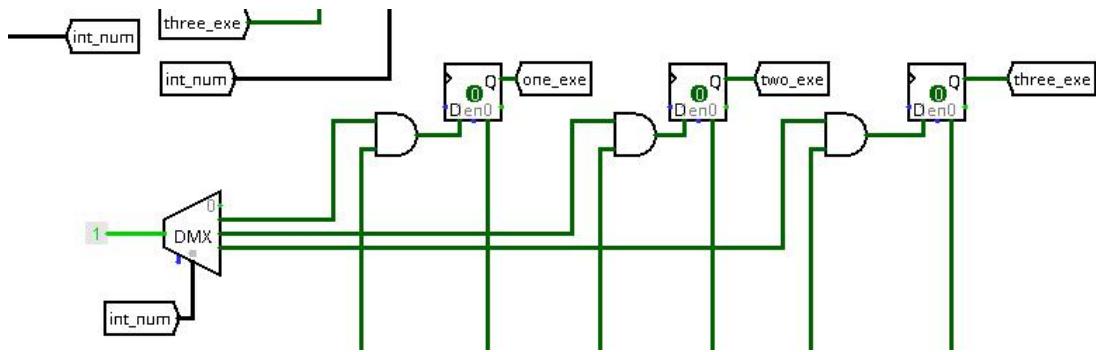


图 3.8 中断是否执行的记录结构

### 3.2.3 中断和流水的整合

将多级中断和流水线进行整合的时候，需要思考的部分在于，在哪一个的位置对于的中断检查以及何处清空流水线。

本次试验采用在写会的阶段检查是否出现的中断，如果出现中断，那么就立刻清空前面的所有流水线中间的内容，然后开始中断，但是开始中断的前提是清楚的知道当前的指令的 npc，当时如果当前指令是一个的气泡的时候，那么保存的指令就是一个错误的指令，所以中断信号需要持续到一个非气泡的指令到达 WB 阶段才可以的，使用寄存器可以实现这一个要求，如图 3.10。

如何的判断的当前指令是一个气泡还是一个的全部都是的 0 的指令 sll,\$0, \$s0, \$0，实现的方法的是，添加一个新变量，在开始的时候，也就是 IF 段，将该变量设置为 1，随着流水进行，如果当前指令被清空，那么该变量必定会被清零，如图 3.11 所示。

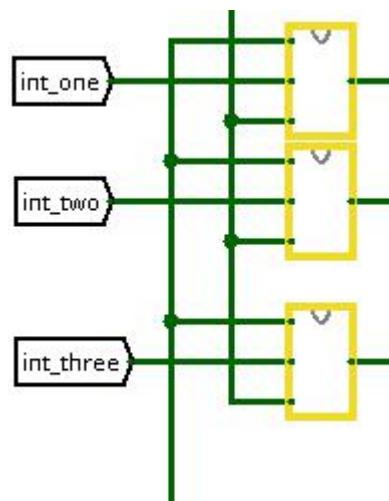


图 3.9 中断信号缓存器

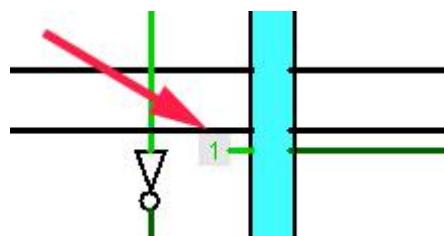


图 3.10 气泡判断信号

## 3.3 流水 CPU 实现

### 3.3.1 流水接口部件实现

- ① Logism 实现：所有流水接口需要提供的接口为清空流水线和是否让流水线的继续进行，根据具体要求添加的对应数目寄存器，清空的原理是，如果的含有清空信号，那么输入信号采用 0，否则采用模块输入，流通信号的产生的为，如果当前信号表示的为不通过，那么将寄存器的 enable 端口关闭，而是让寄存器的数据拒绝更新，如图 3.11

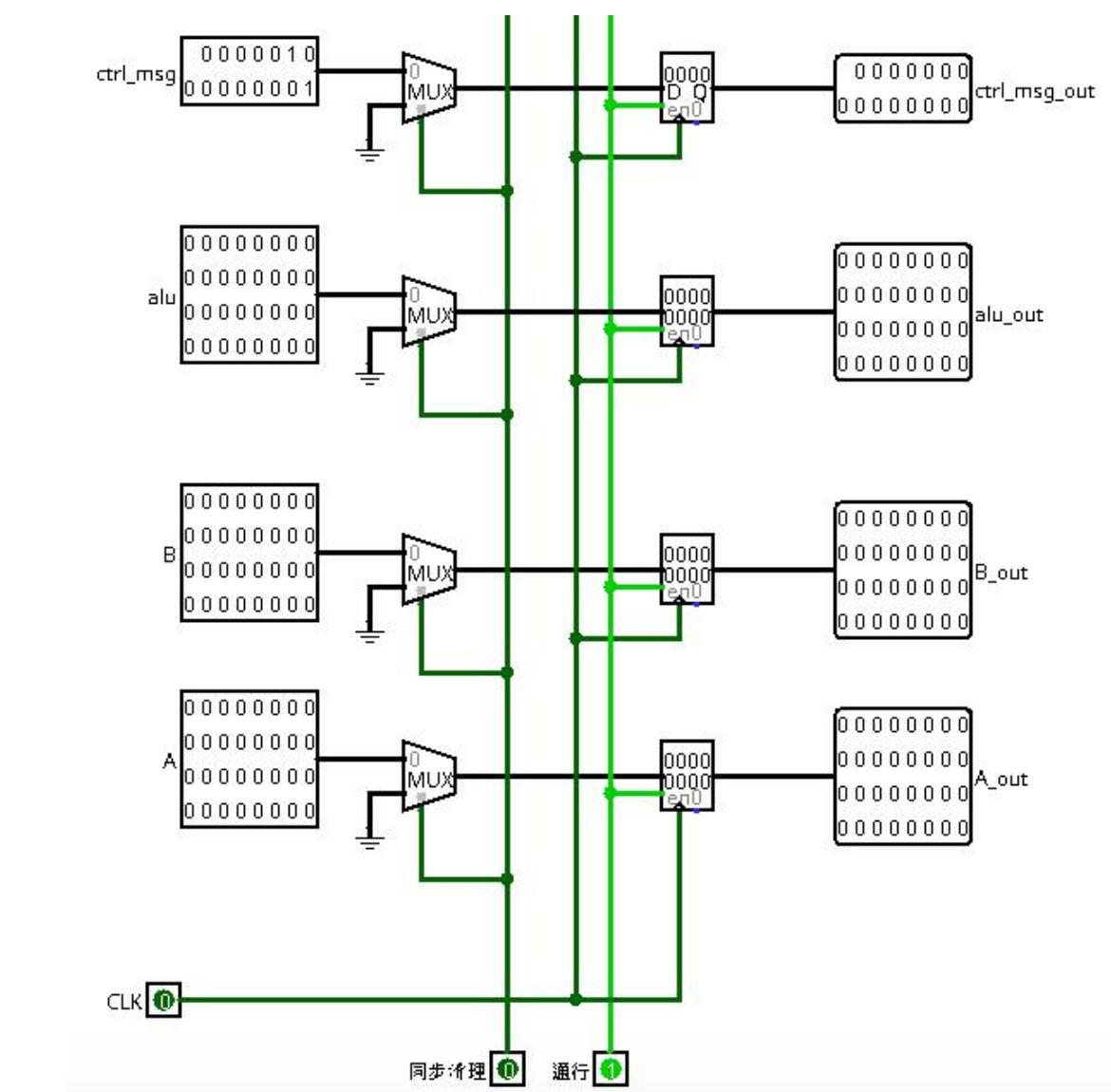


图 3.11 流水接口部分实现

- ② FPGA 实现：FPGA 实现的时候，需要利用的一个第三方的模块为多路选择器

```

`timescale 1ns / 1ps
module IF_ID(
    input [11:0] pc_4,
    input [31:0] instruction,
    input [11:0] addr,
    input p,
    input clear,
    input go_one,
    input go_two,

```

# 华中科技大学课程设计报告

```
input clk,  
  
output reg [11:0] pc_4_out,  
output reg [31:0] instruction_out,  
output reg [11:0] addr_out,  
output reg p_out  
);  
  
wire go;  
assign go = go_one & go_two;  
initial begin  
    pc_4_out = 0;  
    instruction_out = 0;  
    addr_out = 0;  
    p_out = 0;  
end  
wire [11:0] pc_4_out_t;  
wire [31:0] instruction_out_t;  
wire [11:0] addr_out_t;  
wire p_out_t;  
  
MUX_2 #12 mux_1(clear, pc_4, 12'h000, pc_4_out_t, 1'b0);  
MUX_2 #32 mux_2(clear, instruction, 32'h0000_0000,  
instruction_out_t, 1'b0);  
MUX_2 #12 mux_100(clear, addr, 12'h000, addr_out_t, 1'b0);  
MUX_2 #1 mux_101(clear, p, 1'b0, p_out_t, 1'b0);  
always @(posedge clk) begin  
    if(go)  
        begin  
            pc_4_out = pc_4_out_t;  
            instruction_out = instruction_out_t;  
            addr_out = addr_out_t;  
            p_out = p_out_t;  
        end  
    end  
end
```

## 3.4 数据转发流水线实现

### 3.4.1 重定向的控制器的实现

#### ① Logism 实现：

重定向原理在于，需要保证从寄存器的读出来两个数值的都是正确的，但是由于 RAW 的存在，在 EXE 和 MEM 阶段的中间数据没有刷新到寄存器中间，使用的重定向的方法，使用多路选择器来将正确的数据选择出来，检查的原理的在于查看当前的读入数据的是否和 EXE 和 MEM 阶段的中间的数据是否

# 华中科技大学课程设计报告

含有的冲突，如果含有 load-use 冲突，那么暂停的 IM 阶段的一个周期，如果不是该类型冲突，那么的选择正确的数据，而且 EXE 的优先级更加高。如图为 logisim 的实现：

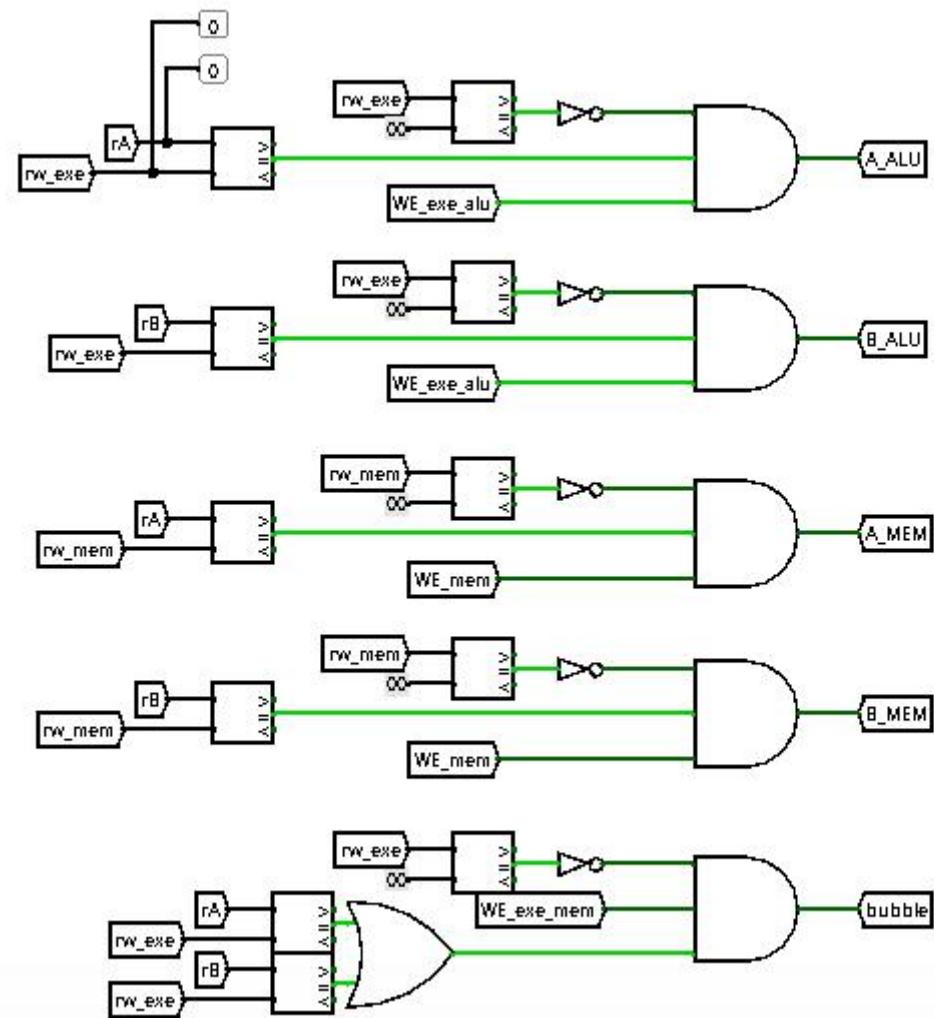


图 3.13 重定向控制器

② FPGA 实现：模块的输入为 IM 读取寄存器编号 EXE MEM 的写入寄存器，输出的为对于寄存器 A B 两个输出采用哪一个阶段的数据以及当前是否发生 load use 冲突信号。

```
'timescale 1ns / 1ps
module redirection(
    input [4:0] rA,
    input [4:0] rB,
    input [4:0] rw_exe,
    input WE_exe_alu,
    input WE_exe_mem,
```

```

input [4:0] rw_mem,
input WE_mem,

output A_ALU,
output B_ALU,
output A_MEM,
output B_MEM,
output bubble
);
assign A_ALU = (rA == rw_exe) && (rw_exe != 4'b0000) &&
WE_exe_alu;
assign B_ALU = (rB == rw_exe) && (rw_exe != 4'b0000) &&
WE_exe_alu;
assign A_MEM = (rA == rw_mem) && (rw_mem != 4'b0000) &&
WE_mem;
assign B_MEM = (rB == rw_mem) && (rw_mem != 4'b0000) &&
WE_mem;

assign bubble = ((rA == rw_exe) || (rB == rw_exe)) &&
WE_exe_mem && (rw_exe != 4'b0000);
endmodule

```

### 3.4.2 重定向的数据选择器的实现

① Logism 实现：重定向的数据选择器的实现在于获取的 EXE 阶段和 MEM 阶段的数据，从流水控件传递过来的重定向的控制信号和寄存器输出数据，然后确定到底的是选择一个数据，一共需要四个两输入多路选择的选择器来实现，在 EXE 阶段的数据具有更加高优先级。具体的实现如图所示。

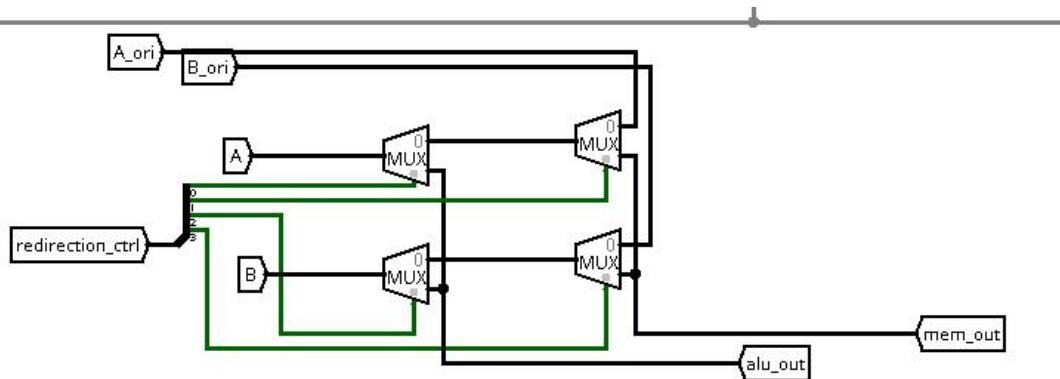


图 3.14 重定向控制信号处理器

# 华中科技大学课程设计报告

② FPGA 实现：调用第三发的模块多路选择器，代码如下

```
'timescale 1ns / 1ps
module redirection_handler(
    input [31:0] A_ori,
    input [31:0] B_ori,
    input [31:0] alu_out,
    input [31:0] mem_out,
    input [3:0] redirection_ctrl,

    output [31:0] A,
    output [31:0] B
);
wire [31:0] A_mem;
wire [31:0] B_mem;
MUX_2 #32 mux_0(redirection_ctrl[1], A_ori, mem_out, A_mem,
1'b0);
MUX_2 #32 mux_1(redirection_ctrl[0], A_mem, alu_out, A, 1'b0);

MUX_2 #32 mux_2(redirection_ctrl[3], B_ori, mem_out, B_mem,
1'b0);
MUX_2 #32 mux_3(redirection_ctrl[2], B_mem, alu_out, B, 1'b0);
endmodule
```

## 3.5 动态分支预测机制实现

### 3.5.1 分支预测的 LRU 算法的实现

LRU 算法只有 FPGA 的版本，计算 LRU 算法类似于华莱士树，对于含有 8 个缓冲器使用一个 3 层的树结构可以该当前的哪一个位置地址是一个最久为被使用的。对应的部分 verilog 代码如下。

```
wire [2:0] i_12;
wire [2:0] i_34;
wire [2:0] i_56;
wire [2:0] i_78;

wire [2:0] i_1234;
wire [2:0] i_5678;

wire [2:0] max_index;

assign i_12 = LRU[0] >= LRU[1] ? 3'b000 : 3'b001;
assign i_34 = LRU[2] >= LRU[3] ? 3'b010 : 3'b011;
assign i_56 = LRU[4] >= LRU[5] ? 3'b100 : 3'b101;
assign i_78 = LRU[6] >= LRU[7] ? 3'b110 : 3'b111;
```

# 华中科技大学课程设计报告

```
assign i_1234 = LRU[i_12] >= LRU[i_34] ? i_12 : i_34;
assign i_5678 = LRU[i_56] >= LRU[i_78] ? i_56 : i_78;

assign max_index = LRU[i_1234] >= LRU[i_5678] ?
i_1234 : i_5678;
```

## 3.5.2 分支预测数据通路的建立

需要的修改为 IF 和 MEM 段，其中在 IF 段使用 BHT 的数据，在 MEM 阶段的来实现更新 BHT, IF 执行的操作是对于的所有 PC 数据都会进行查询，如果查询到，那么使用多路选择器进行选择一个预测跳转的位置。在 MEM 阶段的如果识别到当前指令是一个跳转指令，那么对于 BHT 表进行的更新。

```
'timescale 1ns / 1ps
module data_route(
    input clk1,
    input [5:0]ram_addr_display,
    input rst,
    input frequency,
    input[2:0] display,
    input continue,
    input is_benchmark,

    output [7:0] AN,
    output [7:0] SEG
);

////////////////////////////Common Operation/////////////////////
// frequency exchange !
wire clk;
frequency_switch frequency_switch_0(clk1, clk, frequency);
// 用于展示 ram
wire [31:0] ram_display;
////////////////////////////Declaration/////////////////////////
wire [11:0] npc;
wire ctrl_clash;
wire [31:0] A_wb;
wire [4:0] rw_wb;
```

# 华中科技大学课程设计报告

```
wire WE_wb;

wire [31:0] A_exe;
wire [31:0] B_exe;
wire [31:0] alu_out;
wire WE_exe_alu;
wire WE_exe_mem;
wire WE_mem;
wire [4:0] rw_exe;
wire [4:0] rw_mem;

wire bubble;
wire [31:0] A_mem;
wire [31:0] B_mem;

wire [31:0] word_wb;

// when stop or halt, freeze all the buffer
wire stop_g;
wire go;
wire stop;
wire halt;
assign go = (stop && halt) || rst;
stop_ctrl stop_ctrl_0(clk, continue, stop_g, stop);

// dynamic branch predict
/////////////////////////////动态分支预测/////////////////////////////
wire branch;
wire unbranch;
wire condi_suc;
wire [11:0] pc_4;
wire [11:0] pc_4_exe;
wire [11:0] predict_addr;
wire [11:0] instruction_addr_if_fake;
wire [11:0] instruction_addr_if;
wire [11:0] instruction_addr_id;
wire [11:0] instruction_addr_exe;
wire [11:0] query_ins_addr;

assign query_ins_addr = pc_4;

wire predict_jump_if;
wire predict_jump_id;
```

# 华中科技大学课程设计报告

```
wire predict_jump_exe;
MUX_2 #12 mux_2_1(predict_jump_if, pc_4, predict_addr,
instruction_addr_if_fake, 1'b0);
MUX_2 #12 mux_2_2(is_benchmark, instruction_addr_if_fake,
pc_4, instruction_addr_if, 1'b0);
wire is_branch;
assign is_branch = branch || unbranch;

wire [11:0] insert_ins_addr;
wire [11:0] insert_ins_next_addr;
wire is_suc;
wire is_jump;
assign is_jump = unbranch || condi_suc;
assign is_suc = is_jump;
reg [31:0] suc_counter;

initial begin
    suc_counter = 0;
end
always @(posedge clk) begin
    if(is_jump == predict_jump_exe && is_jump) begin
        suc_counter = suc_counter + 1;
    end
end
end

assign insert_ins_addr = pc_4_exe;
assign insert_ins_next_addr = npc;

BHT bht(clk,
    insert_ins_addr, insert_ins_next_addr, is_branch, is_suc,
    query_ins_addr,
    predict_addr, predict_jump_if);
//////////////////////////////.////////////////////////////

//////////////////////////////IF Area////////////////////////////
// pc
wire [11:0]pc;

wire [11:0] pc_in;
wire pc_enable = halt && stop && (!bubble);
program_counter p_c_0(pc_in, clk, rst, pc_enable, pc);
wire [9:0]addr;
assign addr = pc[11:2];
assign pc_4 = pc + 4;
// IM
wire [31:0] instruction;
IM im(addr, instruction);
// ctrl
```

# 华中科技大学课程设计报告

```
wire n_ctrl_clash;
MUX_2 #12 mux_2_123(n_ctrl_clash, npc, instruction_addr_if,
pc_in, 1'b0);
/////////////////////////////id Area/////////////////////////////
wire [11:0] pc_4_id;
wire [31:0] instruction_id;
wire clear_if_id;
assign clear_if_id = (!bubble) || rst;
IF_ID if_id(pc_4, instruction, instruction_addr_if, predict_jump_if,
ctrl_clash, go, clear_if_id, clk,
pc_4_id, instruction_id, instruction_addr_id, predict_jump_id);

wire [1:0]rA_t;
RA_ctrl r_c_0_0(instruction_id, rA_t);

wire[4:0] rA;
wire[4:0] rB;
read_reg_ctrl read_reg_ctrl_0(instruction_id, rA_t, rA, rB);
// Registers registers(rA, rB, rW, WE, w, clk, A, B);
wire[31:0] A_id;
wire[31:0] B_id;
Registers registers(rA, rB, rw_wb, WE_wb, word_wb, clk, A_id,
B_id);

wire A_alu_red;
wire B_alu_red;
wire A_mem_red;
wire B_mem_red;
redirection redirection_0(rA, rB, rw_exe, WE_exe_alu,
WE_exe_mem, rw_mem, WE_mem,
A_alu_red, B_alu_red, A_mem_red, B_mem_red, bubble); // 通道
名

wire [3:0]redirection_ctrl_id;
assign redirection_ctrl_id = {{B_mem_red},{B_alu_red},
{A_mem_red}, {A_alu_red}};
/////////////////////////////exe Area/////////////////////////////
wire [31:0] A_exe_ori;
wire [31:0] B_exe_ori;
wire [31:0] instruction_exe;
```

# 华中科技大学课程设计报告

```
wire [3:0] redirection_ctrl_exe;
wire clear_id_exe = bubble | ctrl_clash | rst;
ID_EXE      id_ex(pc_4_id,      instruction_id,      A_id,      B_id,
redirection_ctrl_id, instruction_addr_id, predict_jump_id,
go, clear_id_exe, clk,
pc_4_exe,      instruction_exe,      A_exe_ori,      B_exe_ori,
redirection_ctrl_exe, instruction_addr_exe, predict_jump_exe);
// go clear_one clear_two

// should we change the instruction in the verilog
wire [1:0]rW_t_exe;
wire WE_exe;
wire [1:0]wc_exe; // choose which word to wirte
wire [1:0]Y_t;
wire [3:0]alu_s;
wire PC_MUX_2;
wire PC_MUX_3;
wire blez;
wire beq;
wire bne;
wire RAM_STO_exe;
wire RAM_LOAD_exe;
wire half_word_exe;
wire syscall_t_exe;
controller controller_0(instruction_exe,
rW_t_exe, WE_exe, wc_exe, Y_t, alu_s, PC_MUX_2, PC_MUX_3,
blez, beq, bne, RAM_STO_exe, RAM_LOAD_exe, half_word_exe, branch,
unbranch, syscall_t_exe);

wire [14:0] ctrl_msg;
assign ctrl_msg = {{RAM_LOAD_exe}, {rW_t_exe}, {wc_exe},
{WE_exe}, {syscall_t_exe}, {4'b0000}, {RAM_STO_exe},
{half_word_exe}, {2'b00}};

wire [31:0] Y;
Y_ctrl y_ctrl(instruction_exe, B_exe, Y_t, Y);

redirection_handler r_h_0(A_exe_ori,      B_exe_ori,      alu_out,
word_wb, redirection_ctrl_exe,
A_exe, B_exe);

wire [31:0] alu_exe;
wire [31:0] useless_0;
wire useless_1;
wire useless_2;
wire useless_3;
ALU      alu_0(A_exe,      Y,      alu_s,      alu_exe      ,useless_0,
useless_1,useless_2, useless_3);
```

# 华中科技大学课程设计报告

```
wire write_alu;
MUX_4 #1 mux_4_0(wc_exe, 1'b1, 1'b1, 1'b0, 1'b0, write_alu,
1'b0);
assign WE_exe_alu = (WE_exe & write_alu);
assign WE_exe_mem = (WE_exe & !write_alu);

write_reg_ctrl      write_reg_ctrl_0(rW_t_exe,      instruction_exe,
rw_exe);

wire [31:0] pc_4_exe_32;
wire [31:0] merge_alu;
assign pc_4_exe_32 = {{20{1'b0}}, pc_4_exe};
MUX_4 #32 mux_4_1(wc_exe, alu_exe, pc_4_exe_32,
32'h0000_0000, alu_exe, merge_alu, 1'b0);

condi_jump c_j_0(A_exe, B_exe, blez, beq, bne, condi_suc);

wire strong_halt;
assign strong_halt = stop & halt;
wire [31:0]total_cycles;
wire [31:0]uncondi_num;
wire [31:0]condi_num;
wire [31:0]condi_suc_num;
wire [31:0]SyscallOut;

statistic statistic_0(A_exe, B_exe, clk, rst, syscall_t_exe,
condi_suc, unbranch, branch, strong_halt,
total_cycles, uncondi_num, condi_num, condi_suc_num,
SyscallOut);

npc_generator np_0(instruction_exe, A_exe, pc_4_exe,
condi_suc, PC_MUX_2, PC_MUX_3, npc);
assign n_ctrl_clash = (npc == instruction_addr_exe);
assign ctrl_clash = !n_ctrl_clash;
//////////////////////////////MEM Area/////////////////////////////
wire [31:0] instruction_mem;
wire [14:0] ctrl_msg_mem;
EXE_MEM exe_mem_0(instruction_exe, ctrl_msg, merge_alu,
A_exe, B_exe,
```

# 华中科技大学课程设计报告

```
go, rst, clk,
instruction_mem, ctrl_msg_mem, alu_out, A_mem, B_mem);

wire half_word;
wire RAM_STO;
wire syscall_mem;
wire[1:0] wc_mem;
wire[1:0] rW_t;
wire RAM_LOAD;
assign half_word = ctrl_msg_mem[2];
assign RAM_STO = ctrl_msg_mem[3];
assign syscall_mem = ctrl_msg_mem[8];
assign WE_mem = ctrl_msg_mem[9];
assign wc_mem = ctrl_msg_mem[11:10];
assign rW_t = ctrl_msg_mem[13:12];
assign RAM_LOAD = ctrl_msg_mem[14];

write_reg_ctrl w_r_c_0(rW_t, instruction_mem, rw_mem);

wire byte_choose;
wire [5:0] ram_addr;
assign byte_choose = alu_out[1];
assign ram_addr = alu_out[7:2];
wire [31:0] ram_word;
DM dm_0(ram_addr, ram_addr_display, B_mem, RAM_STO,
2'b11, rst, clk, ram_word, ram_display);

wire [31:0] ram_word_se;
word_ctrl w_c_0(byte_choose, half_word, ram_word,
ram_word_se);

wire [31:0] word_mem;
MUX_4 #32 mux_1_1(wc_mem, alu_out, alu_out,
32'h0000_0000, ram_word_se, word_mem, 1'b0);

//////////////////////////////WB Area/////////////////////////////
wire syscall_wb;
MEM_WB mem_wb(syscall_mem, WE_mem, rw_mem, A_mem,
word_mem,
go, rst, clk,
syscall_wb, WE_wb, rw_wb, A_wb, word_wb);

assign halt = !((A_wb == 32'ha) && syscall_wb);
assign stop_g = !((A_wb == 32'h32) && syscall_wb);
```

# 华中科技大学课程设计报告

```
///////////Show Area///////////
wire [31:0] show_pc;
assign show_pc = {{20'h00000}, {pc}};
Data_Choose show_data( display, ram_display, total_cycles,
condi_num, uncondi_num,
    condi_suc_num, SyscallOut, show_pc, clk1,
    AN, SEG);
///////////
endmodule
```

### 3.5.3 全连接的实现

通过的全连接的实现来对于输入地址和所有 valid 地址进行比较，然后将比较得到的数据取或，得到是否命中信号，根据是否命中的信号和比对信号来确定的选择的哪一个阶段的数据。对应代码如下。

```
///////////0/////////
wire insert_hit_0;
wire insert_suc_0;
wire insert_fail_0;
assign insert_hit_0 = is_branch && (insert_ins_addr ==
addr[0]) && valid[0];
assign insert_suc_0 = insert_hit_0 && is_suc;
assign insert_fail_0 = insert_hit_0 && !is_suc;
///////////

///////////1/////////
wire insert_hit_1;
wire insert_suc_1;
wire insert_fail_1;
assign insert_hit_1 = is_branch && (insert_ins_addr ==
addr[1]) && valid[1];
assign insert_suc_1 = insert_hit_1 && is_suc;
assign insert_fail_1 = insert_hit_1 && !is_suc;
///////////

///////////2/////////
wire insert_hit_2;
wire insert_suc_2;
wire insert_fail_2;
assign insert_hit_2 = is_branch && (insert_ins_addr ==
addr[2]) && valid[2];
```

# 华中科技大学课程设计报告

```
assign insert_suc_2 = insert_hit_2 && is_suc;
assign insert_fail_2 = insert_hit_2 && !is_suc;
/////////////////////////////3/////////////////////////////
wire insert_hit_3;
wire insert_suc_3;
wire insert_fail_3;
assign insert_hit_3 = is_branch && (insert_ins_addr ==
addr[3]) && valid[3];
assign insert_suc_3 = insert_hit_3 && is_suc;
assign insert_fail_3 = insert_hit_3 && !is_suc;
/////////////////////////////4/////////////////////////////
wire insert_hit_4;
wire insert_suc_4;
wire insert_fail_4;
assign insert_hit_4 = is_branch && (insert_ins_addr ==
addr[4]) && valid[4];
assign insert_suc_4 = insert_hit_4 && is_suc;
assign insert_fail_4 = insert_hit_4 && !is_suc;
/////////////////////////////5/////////////////////////////
wire insert_hit_5;
wire insert_suc_5;
wire insert_fail_5;
assign insert_hit_5 = is_branch && (insert_ins_addr ==
addr[5]) && valid[5];
assign insert_suc_5 = insert_hit_5 && is_suc;
assign insert_fail_5 = insert_hit_5 && !is_suc;
/////////////////////////////6/////////////////////////////
wire insert_hit_6;
wire insert_suc_6;
wire insert_fail_6;
assign insert_hit_6 = is_branch && (insert_ins_addr ==
addr[6]) && valid[6];
assign insert_suc_6 = insert_hit_6 && is_suc;
assign insert_fail_6 = insert_hit_6 && !is_suc;
/////////////////////////////7/////////////////////////////
wire insert_hit_7;
wire insert_suc_7;
wire insert_fail_7;
```

# 华中科技大学课程设计报告

```
assign insert_hit_7 = is_branch && (insert_ins_addr ==  
addr[7]) && valid[7];  
    assign insert_suc_7 = insert_hit_7 && is_suc;  
    assign insert_fail_7 = insert_hit_7 && !is_suc;  
    /////////////////////////////////  
  
wire insert_no_hit; // 没有命中  
assign insert_no_hit = is_branch && !(insert_hit_0 ||  
insert_hit_1 || insert_hit_2 || insert_hit_3 || insert_hit_4 || insert_hit_5 ||  
insert_hit_6 || insert_hit_7);  
    /////////////////////////////////
```

## 4 实验过程与调试

#### 4.1 测试用例和功能测试

### 4.1.1 测试用例 1

使用的 benchmark + ccmb 进行测试， 测试结果如下所示

### 1. B 指令测试结果如图 4.1

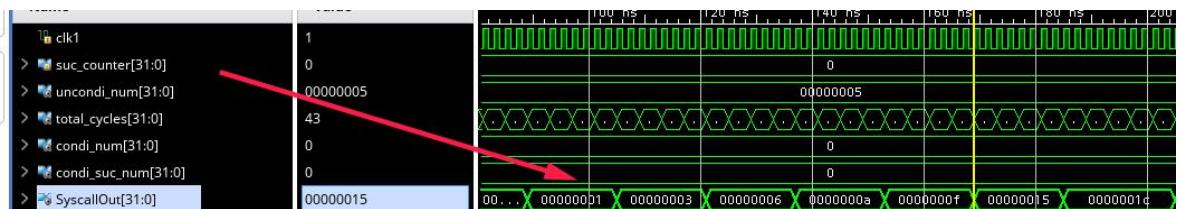


图 4.1 B 指令测试结果

2. 移位指令测试结果如图 4.2 图 4.3 和图 4.4 所示

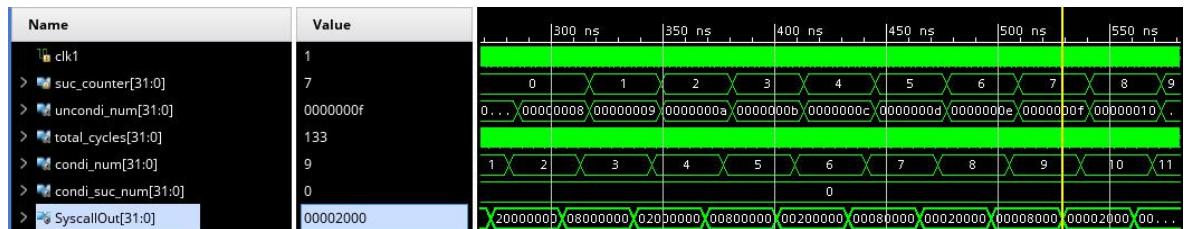


图 4.2 B 指令测试结果

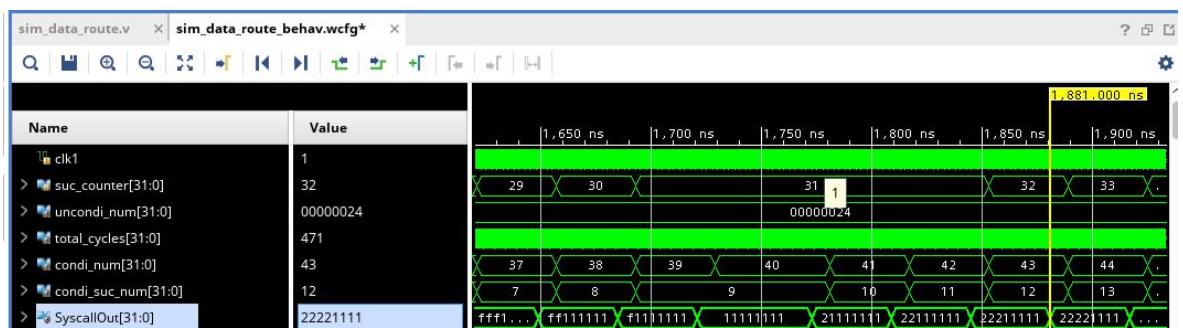


图 4.3 B 指令测试结果

# 华 中 科 技 大 学 课 程 设 计 报 告

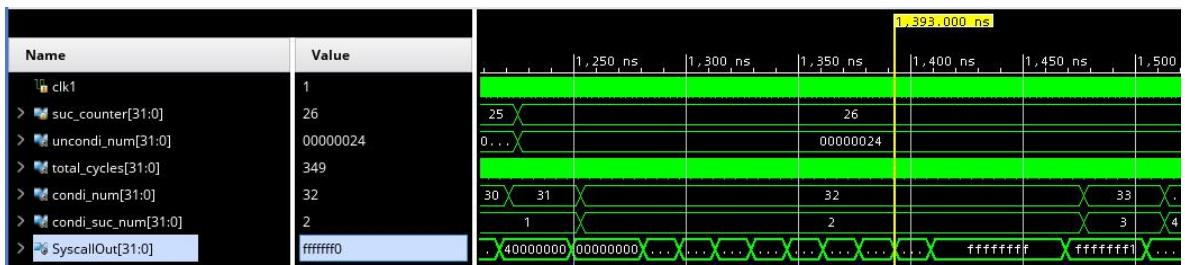


图 4.4 B 指令测试结果

### 3. 存取指令的测试结果如图 4.5

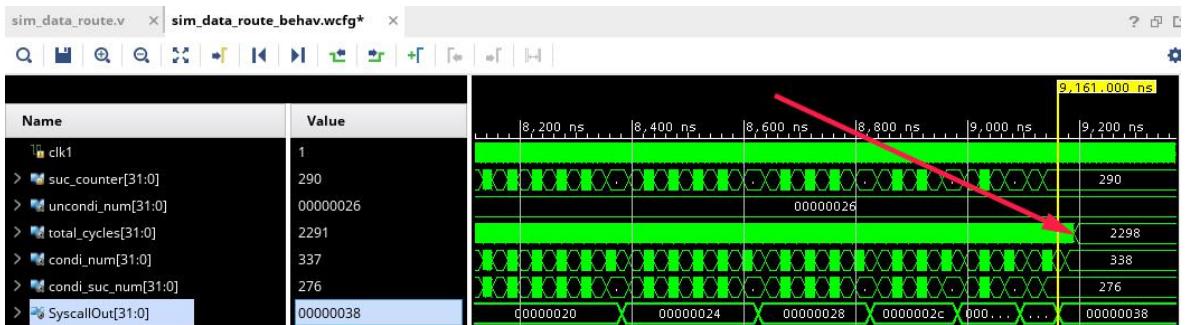


图 4.5 存取指令测试结果

4. SRLV 指令测试结果，SRLV 指令移位的时候，高位 0 补全，从图中间的  
可以看见，数值从 87600000 变化为 08760000，最后变化为 0

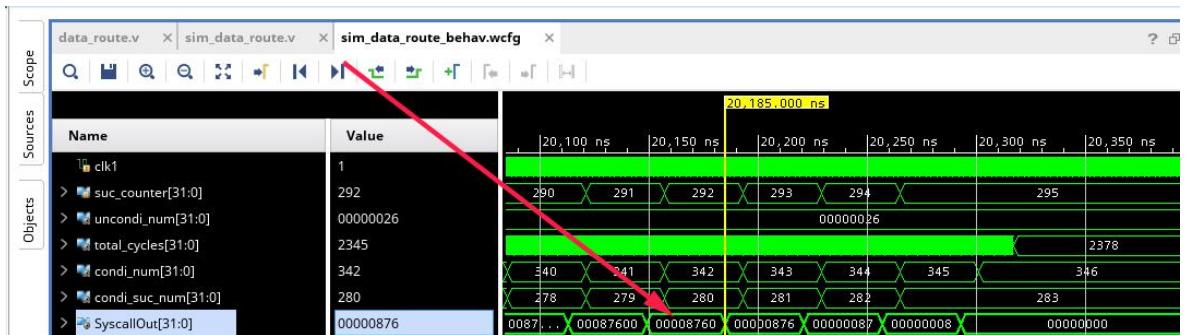


图 4.6 SRLV 指令测试结果

5. SRAV 指令测试结果如图所示， SLAV 指令移动位置的根据原来最高位的变量是什么确定，从图 4.7 中间可以看出来， 数值变化为 87600000 到 f8760000，最后变化为 ffffffff。

# 华中科技大学课程设计报告

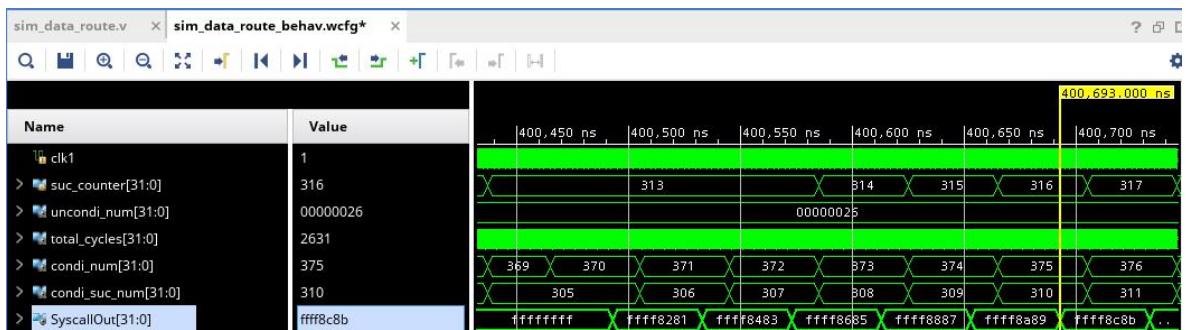


图 4.7 SRAV 指令测试结果

6. SH 指令测试结果如图 4.8 所示，SH 指令将一个字的高位获取，然后进行符号扩展，使用标准的测试程序来测试的时候，对应内存的数据如图所示。

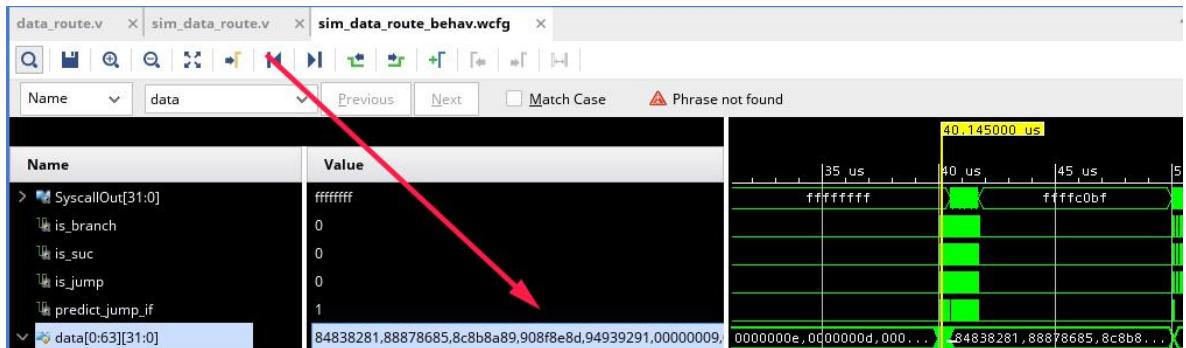


图 4.8 SH 指令测试，数据存储的实时数据

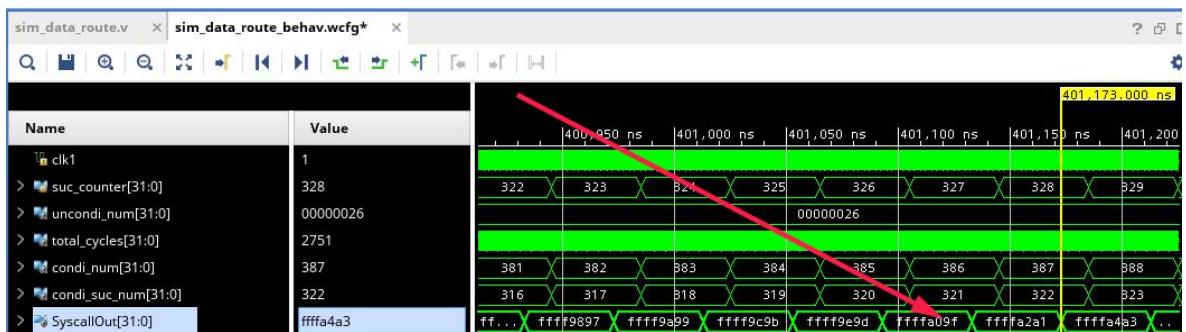


图 4.8 SH 指令测试结果

# 华中科技大学课程设计报告

7. BLEZ 指令测试结果如图 4.9 所示。

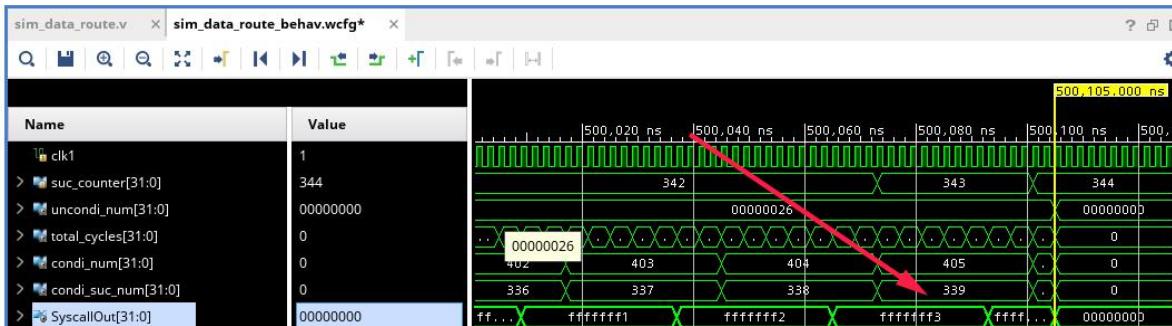


图 4.9 B 指令测试结果

## 4.2 测试技巧总结

如果不死于测试，首先不要使用危险的操作，在 logisim 中间，不要试图使用和上下跳沿的方法，不然会出现非常微妙的错误。

在 verilog 中间，复制代码似乎比 for 循环更加的安全。

书写 verilog 的时候，如果不想在波形图中间检查错误，那么首先需要保证的在 compile 和 elaboration 中间的任何警告都被处理掉，需要可以清楚区分这两个的阶段的 warning 和 综合阶段的 warning 的严重程度不同，对于位宽不一致的操作，显然会导致未定义的错误，但是 vivado 确将其规定的为 warning，显然是不应该的。

最终方法，使用 logisim 的 probe 和 vivado 的型号追条比对，但是其中有现实的地方在于，并没有良好处理的方法大量数据比对，首先测试模块尽可能小，然后使用二分比对，比对的项目首先是指令是否对应，如果发现指令不对应，那么的显然是数据通路的错误跳转的错误，如果所有指令都是对应，一般对应为存取等比较简单的错误。

## 4.3 主要故障与调试

### 4.3.1 Verilog 带来 wire 线理解的出现错误

重定向上板：在 MEM 段出现的一个的错误，出现了名称冲突 A\_mem 和 redirection 的 A\_mem 出现冲突。

# 华中科技大学课程设计报告

**故障现象：**在测试的重定向的时候发现重定向的数据不是正确的。

**原因分析：**通过检查文件的名称时候，首先发现使用的 mem 的寄存器的两个输出的名称只有的对应的一个名称，但是用于的重定向的 reg 的两个的输出需要单独的分析的，如图 4.9 所示，所以的认定的应该缺少的一个的正确的命名方法，导致的出现的冲突。

**解决方案：**给重新使用两个的命名的方案的，对于用于重定向的数据使用 A\_mem\_red 的命名方案，对于五段流水的数值，采用的 A\_mem 的命名的方案的，寄存器的输出 B，采用的相同的命名的方案的，在的对应的前仿真的中间的可以得到的结果的如下图 4.10 所示。

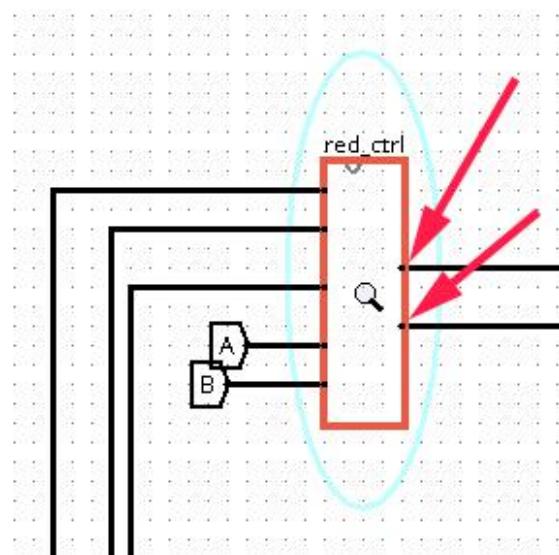


图 4.9 重定向控制信号处理器件

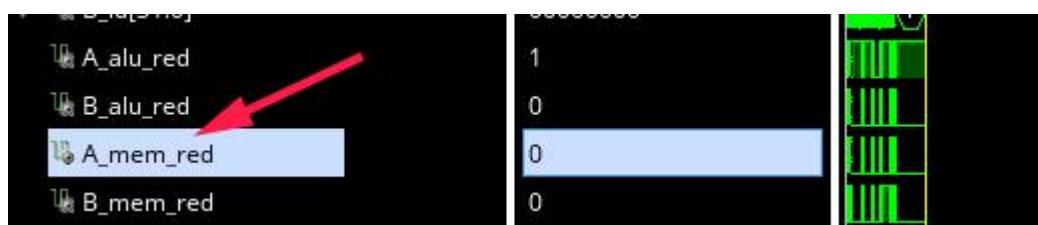


图 4.10 使用不同命名方案实现两者

## 4.3.2 Halt 暂停指令处理故障

重定向的上板：halt 和 stop 信号出现的错误，导致的没有正常的暂停的功能的

**故障现象：**如图 4.11 所示，对应的图形结果没有出现的演示的 CCMB 的过程的，导致之后的波形图中间的只会出现 CPU 停机的结果

**原因分析：**添 在使用 CPU 停机的设计的时候，由于的采用的策略是的 halt 信号的是没有的办法屏蔽的，但是的使用 continue 按钮可以让的 stop 信号的继续，也就是说，当检查的 continue 之后，应该出现的继续的结果。

**解决方案：**修添加的一个模块的来处理 continue 按钮事件的监听，实现的功能是，只要出现了 sotp 信号，产生停机信号，如果出现了 continue 的信号，那么的立刻屏蔽的该 stop，并且在 continue 信号结束之后，依旧持续的一个周期，从而当出现 halt 信号或者 stop 信号的时候，都会停机，但是 continue 信号可以消除 sotp 信号的影响。

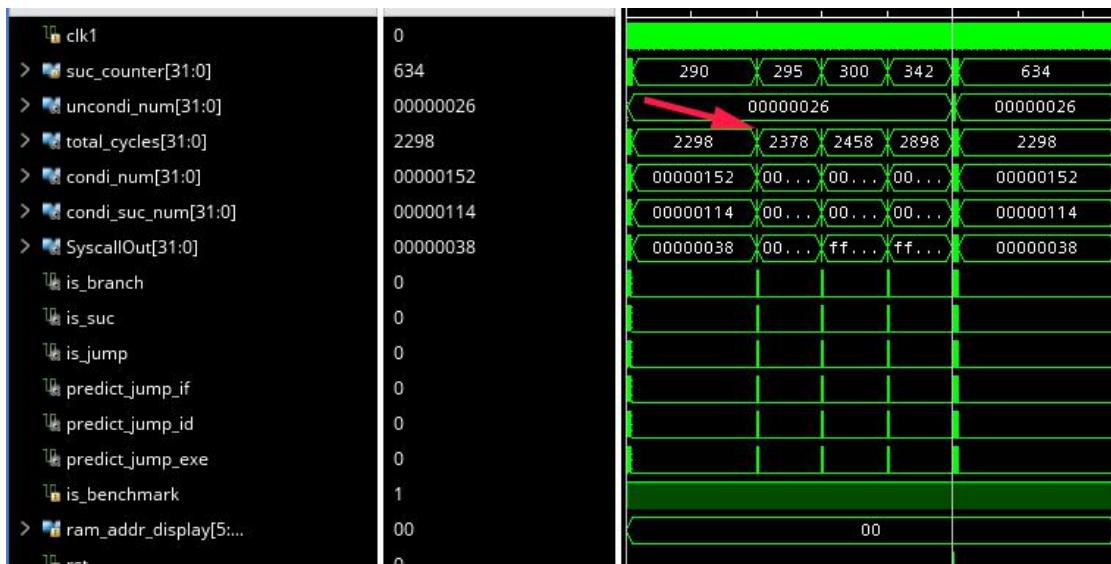


图 4.11 halt 指令可以办法正确执行

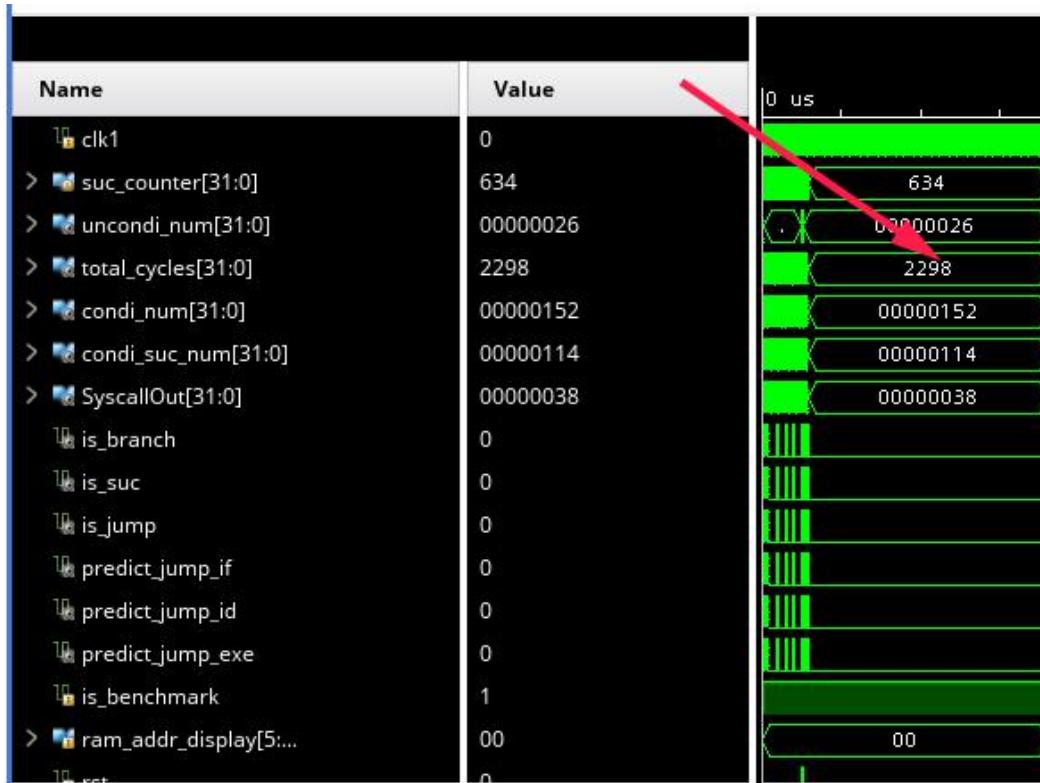


图 4.1.2 halt 指令无法办法正确执行

### 4.3.3 重定向信号故障

Verilog 实现流水重定向：重定向信号数据通路的没有构建正确。

**故障现象：**如使用 benchmark 测试的时候，显示的结果和预期不一致，而且在非常的开始的位置就是的出现错误，使用 B 指令进行测试，依旧出现的错误，可以断定，错误的是一个非常严重的数据通路的错误。

**原因分析：**分码 在实现重定向的时候，首先完成了 redirection\_controller 的模块，该模块计算的位置在于的 IM 段，最开始的时候使用该数据的时候发现虽然的计算的出来的控制信号是正确的，但是该数据发挥作用的时候却是在下一个的周期的位置，由此可以初步的确定，对于的重定向控制单元数据应该传递到下一个周期的时候使用。

**解决方案：**在流水 IM/EXE 阶段添加一个新的缓冲的单元，来实现对于的控制信号的延迟的释放的，如图 4.1.2

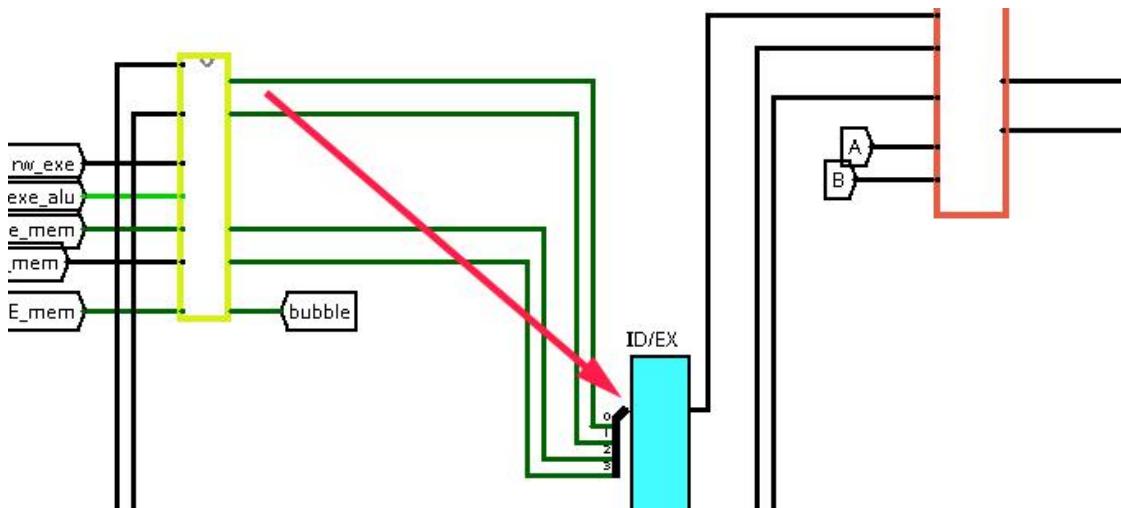


图 4.12 将控制信号延迟一个周期

#### 4.3.4 LRU 算法实现错误

**动态分支预测：**错误的实现 LRU 算法。

**故障现象：**动态分支预测优化数量没有达到要求。

**原因分析：**分码原来设计的方案是只要出现新的项目的时候，一个计数指针加 1，然后将项目插入到该位置，使用这一个想法的原因是，只要可以只要出现新的数据，然后就将该数据替换掉栈最底部数据，但是如果一个数据本来就是在 BHT 中间，那么没有办法将该项目移动最上面，所以该算法是错误的。

**解决方案：**使用树状结构计算出来的最大的数值，然后将其他的所有的项目数值加 1，但是该数值设置为 0 .

#### 4.3.5 流水接口实现错误

**重定向上板：**流水接口 FPGA 实现错误

**故障现象：**如使用 benchmark 测试的时候，数据通路从来没有暂停过，即时 load-use 信号被检测出来，重定向控制器输出没有任何问题，但是 IM 段流水接口依旧没有办法停止。

**原因分析：**如果输入到流水的接口的信号是暂停，但是流水接口拒绝停止，那

# 华中科技大学课程设计报告

么原因只有两个，数据通路书写错误，命名错误导致正确的数据其实并没有的写入对应的模块中间，还有的可能就是，该模块没有正确处理好该数据。

**解决方案：**在 verilog 代码中间从原来的基本 and 逻辑修改为或逻辑，表示为只要出现任何暂停请求都是会导致该流水接口的发生暂停

## 4.3.6 ORI ANDI 指令理解错误

**数据重定向：**对于 ori 和 andi 指令没有使用算术扩展

**故障现象：**正在构建的重定向的时候，本来应该显示的一个 FFFF4321 但是实际上显示的是 00004321

**原因分析：**面对这一种情况，很有可能错误的原因是本来是算术符号扩展，但是的实际上被处理为逻辑扩展，但是的非常奇怪的是既然含有此种错误，那么应该在实现单周期的 CPU 的时候就是应该出现错误，分析之后发现原来是原来的测试的数据强度不够，ORI 和 ANDI 一直都是含有错误，如图所示。

**解决方案：**对于 CPU 的输入端口 Y 的控制器添加一个判断信号，如果发现当前的指令为 ORI 或者 ANDI 指令，那么将 instruction 的立即数进行符号扩展，否则进行逻辑扩展。

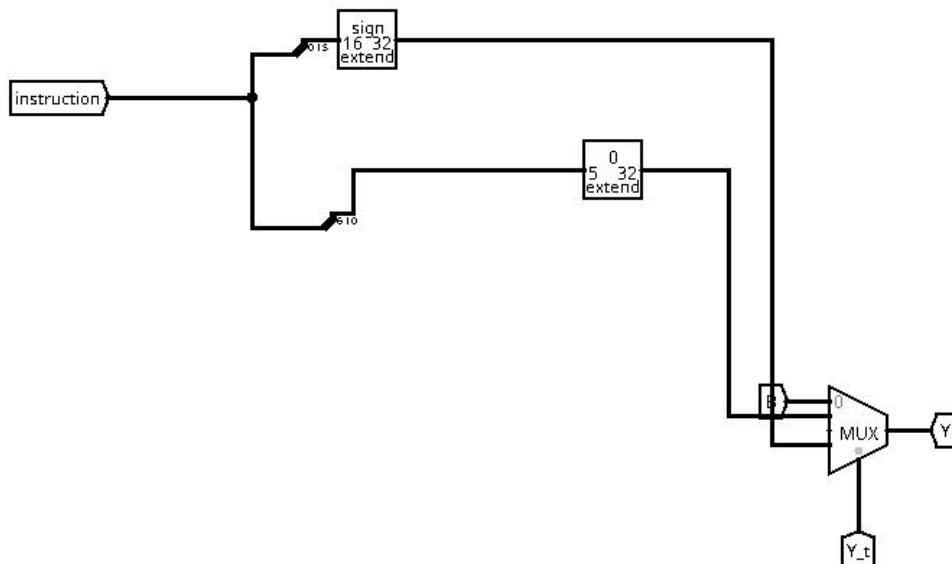


图 4.13 ori andi 指令没有被单独处理

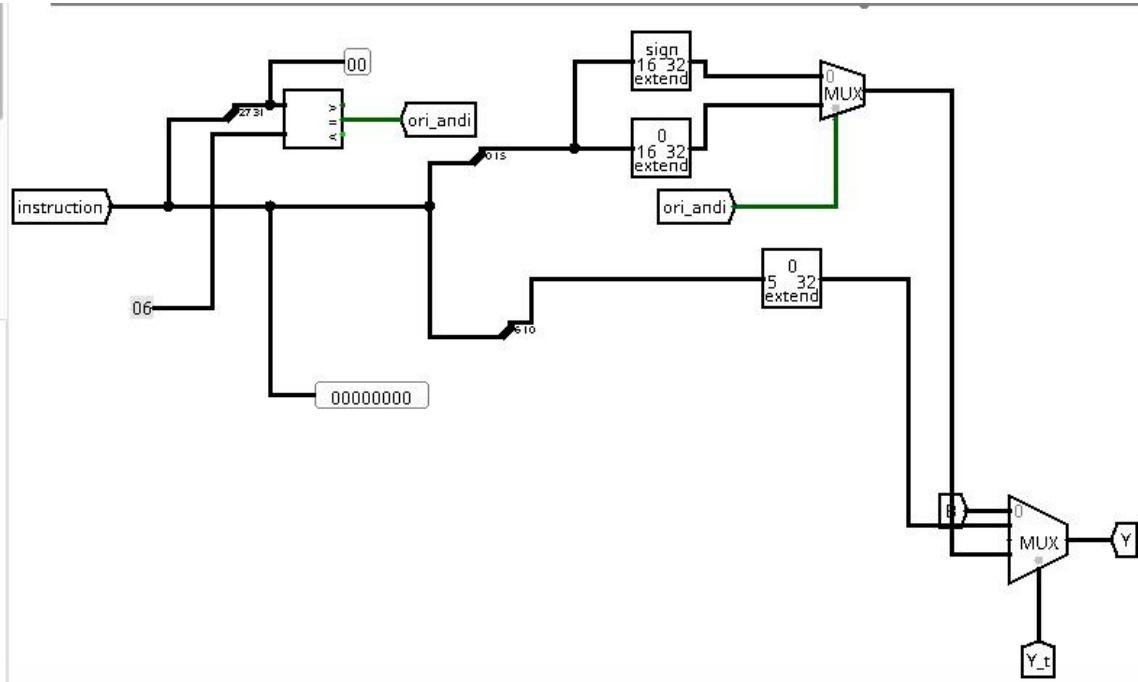


图 4.1.4 ori andi 指令单独处理

### 4.3.7 多级中断返回错误

Verilog 实现流水重定向：重定向信号数据通路的没有构建正确。

**故障现象：**如使用 benchmark 测试的时候，显示的结果和预期不一致，而且在非常的开始的位置就是的出现错误，使用 B 指令进行测试，依旧出现的错误，可以断定，错误的是一个非常严重的数据通路的错误。

**原因分析：**分码 在实现重定向的时候，首先完成了 `redirection_controller` 的模块，该模块计算的位置在于的 IM 段，最开始的时候使用该数据的时候发现虽然的计算的出来的控制信号是正确的，但是该数据发挥作用的时候却是在下一个的周期的位置，由此可以初步的确定，对于的重定向控制单元数据应该传递到下一个周期的时候使用。

**解决方案：**在流水 IM/EXE 阶段添加一个新的缓冲的单元，来实现对于的控

# 华中科技大学课程设计报告

制信号的延迟的释放的，如图 4.5

## 4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	分配的任务，得到的任务的是处理数据通路和一个小型的模块的 完成的数据通路的构建，检查的出来部分 bug，这里面含有的 bug 有： alu 对于 <<< 运算理解错误，在 X = 0x8000000 y = 0x0000001f op = 1 的实现的为逻辑移动，但是应该是算术移动
第二天	alu_cont 出现了错误 变量赋值反了 <pre>assign y_in2 = imm_16_32_signed; assign y_in3 = imm_16_32_unsigned;</pre> DM 读数据错误 DM 读数据应该是组合逻辑，写数据为时序逻辑，实现时读写均为时序逻辑。 但是距离上板遥遥无期。
第三天	完成的所有 bug 的查询的，开始着手的重构的原来的单周期的图形，变化成为一个经典的五段的流水的结构。
第四天	完成的经典流水，由于在构建单周期的时候没有对于模块实现较好封装，所以需要添加非常多的模块，从 CPU 的输入控制器，到 NPC 模块，从条件跳转成功指示器到统计信息。
第五天	完成气泡流水，毫无难度可言。
第六天	完成的重定向的流水
第七天	CCF 考试，备考，和考试，晚上需要的上课，所以没有任何的进度。
第八天	阅读中断的文档，为的重定向的流水添加一个暂停的功能，发现了从单周期到重定向流水一直存在 bug，那就是在 ori 和 andi 指令理解出现了失误。
第九天	完成单周期的多级中断的构建，直接跳过了单级中断。

# 华中科技大学课程设计报告

时间	进度
第十天	完成多级中断在单周期上面的测试，并且将多级中断移植到重定向的流水上面，发现的含有某些问题的，中断有的时候并不可以进入，退出的时候也是没有办法正常退出到一个正确的位置。
第十一天	检查的重定向多级中间的 bug，并且初步完成流水的上板代码。
第十二天	<p>每一个的模块检查 vivado 的波形图，从分别检查出来的 bug 有，没有初始化的寄存器，变量的名称不对应。</p> <p>对于的缓存器的理解出现问题，内部设置通行的逻辑错误，当出现通行的时候，应该是首先使用 and 逻辑。</p> <p>IM 处理错误，错误的使用了分线器，应该直接删除的第一个位置，然后使用第二个位置判断应该是哪一个的位置，同时，需要修改的存储器，删除内部移动位置。</p> <p>对于 syscall 的理解错误，原来是只要不为 10 就是可以显示了。</p> <p>控制器书写错误，当使用默认数值的时候会出现相互冲突的情况，使用枚举方法的时候应该让在三元运算符号的时候，都是保持结尾位置是一个 z 结束的</p> <p>申明的变量的位置出现错误，最后发现只有的排序指令和 SH 指令的测试含有问题</p>
第十三天	修复了的存取指令的 bug，重定向的前仿真完成，书写了动态分支预测的代码，但是发现的优化的数量达不到的要求的。
第十四天	分析动态分支预测的错误，发现淘汰策略书写错误，重写使用的 LRU 算法构建的动态分支预测的淘汰算法。发现实现的暂停的控制模块含有问题，在的后仿真的时候的会失败的，经过一位大佬指点，发现是由于少写了一个 else 语句
第十五天	支持同时检查动态分支预测和重定向的上板成功，可以检查了。

## 5 设计总结与心得

### 5.1 课设总结

在上一个的学期的单周期的 CPU 的设计的基础上面，从 FPGA 上板到逐渐实现了气泡，重定向，中断和分支预测的功能。其中加深了对于团队的协作的理解，强化了的自己对于硬件的底层工作的理解。作了如下几点工作：

- 1) 完成了单周期的 CPU 的 FPGA 上板，理想的流水对于原有图形的重新的设计，单周期流水气泡的实现，重定向的流水的 logisim 的实现，重定向 FPGA 的实现，重定向和多级中断的综合，重定向和动态分支预测的整合，总而言之，终于通关了。
- 2) 在 logisim 上实现了支持多级中断的重定向的 CPU 的设计，在 FPGA 上面支持的动态分支预测的重定向的流水 CPU。
- 3) 重构了 CPU 的架构，将原来的单周期的 CPU 强化为一个严格的五段流水的架构。

### 5.2 课设心得

本次课设的收获是巨大的，中间的过程中一直觉得有一个的 98 分或者 105 就是的值的，但是面对一个每年文档都在更新，充满挑战，思路清晰的试验，我觉得的不通关的是对于自己的认知能力的侮辱。反观所谓的操作系统试验，完全过期的文档，不清晰的描述，检查老师的恶意刁难，真的让人的觉得恶心。

本次课设的是计算机组成原理，并行计算理论于时间，计算机体系结构三个的课程的实践的收官之战，从运算器的设计，存储的设计，CPU 的设计到指令级的并行执行，到动态分支预测，虽然使用了两个星期的时间，但是收货也是显然的。

本次试验不仅仅是知识上面的收获，更加是团队协作上面的收获，单周期上板的过程中间，团队话费的时间巨大，直到第三天的时候彻底完成，虽然在技术，团队中间的每一个成员的都是毫无疑问的，从对于 CPU 设计理解，到对于的

# 华 中 科 技 大 学 课 程 设 计 报 告

---

verilog 的语言理解， 到版本控制， 多人任务协作， 不需要含有任何质疑， 但是关键的问题在于群龙无首， 开始的时候的任务的分配的没有处理好， 导致有些人任务非常的重， 有的人几乎没有的什么任务， 而团队的进度取决于最慢的一个的， 其中最搞笑的在于， debug 的任务开始被分配给一个人， 结果显然的导致的最慢的成员压力巨大， 而其他的人只有干着急或者做私活， 后来重洗分配的任务之后， 才让进度得到快速的推进。

当然， 本次试验的也是含有一些问题， 比如需求的变化的，在上板的时候， 是否支持的 CCMB 的问题耽误了团队一个下午， 单周期上板开始的说不用支持 CCMB， 后来在可以开始检查之后， 有开始调试 CCMB 指令，在版本更新了多个之后， 最后放弃的了， 回退到的不支持的版本， 试验中间的还有一个的非常让人的头痛的问题在于的如何让的 CPU 支持停止的指令， 这一个添加的要求的问题在于的如何将一个脉冲信号转化成为一个电频信号， 而且的持续时间不可以超过一个的周期， 我采用的方案非常的不自然， 也花费不少的时间的理解。

本次试验的的文档的上面的没有提供的部分是如何快速的在 vivado 上面的 debug， 这一个的问题的在单周期上板的时候困扰了很多人，在重定向上板的时候依旧是一个让人的恐惧的问题， 我在后来的重定向的上板的时候的逐渐体会到的， 只要所有小的测试可以通过， 那么综合部分的通过是没有任何的问题的， 问题的关键的在部分的小模块不够小， 尤其的是用于测试的存取指令的排序， 一共含有的 1000 多个周期， 在这里面的发现是从哪一个周期出现的问题的， 使用二分的查找的方法， 也是需要检查的 10 多次， 而且到底是比对的那些数据项目是一个具有挑战性的问题。

最后在这里也感谢老师门， 其中某些简单的 verilog 问题一再打扰狐狸老师， 老师反而每次的都是的热情的给出的答案， 对于设计上面的问题询问老虎老师， 也是总是可以的得到的清晰明了的回答。

# 华 中 科 技 大 学 课 程 设 计 报 告

---

---

## 参考文献

- [1]DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第4版).北京：  
机械工业出版社.
- [2]David Money Harris(美).数字设计和计算机体系结构(第二版). 机械工业出版  
社
- [3]秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011年.
- [4]袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011年.
- [5]张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008年.

**·指导教师评定意见·**

---

---

## 一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：

