

it

华中科技大学

2017

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:	CS1409
学 号:	U201414800
姓 名:	刘一龙
电 话:	15927004132
邮 件:	sabertazimi@gmail.com
完成日期:	2016-02-24 周五下午



计算机科学与技术学院

华中科技大学课程设计报告

目 录

1	课程设计概述.....	3
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
2	总体方案设计.....	6
2.1	单周期 CPU 设计	6
2.2	中断机制设计.....	18
2.3	流水 CPU 设计	23
2.4	数据转发流水线设计	25
2.5	气泡式流水线设计	26
2.6	动态分支预测机制	27
3	详细设计与实现.....	29
3.1	单周期 CPU 实现	29
3.2	中断机制实现.....	42
3.3	流水 CPU 实现	45
3.4	数据转发流水线实现	47
3.5	气泡式流水线实现	49
3.6	动态分支预测机制实现	50
4	实验过程与调试.....	54
4.1	测试用例和功能测试	54
4.2	性能分析	56
4.3	主要故障与调试.....	57
4.4	实验进度	60

华中科技大学课程设计报告

5 设计总结与心得	61
5.1 课设总结	61
5.2 课设心得	61
参考文献.....	64

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及实现能力的培养”。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号的有效形式；
- (5) 设计出实现指令功能的硬布线控制器；

华中科技大学课程设计报告

- (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	SU b	减	
11	OR	或	
12	ORI	立即数或	
13	NOR	或非	

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
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)
24	SYSCALL	系统调用	else 数码管显示\$a0 值
25	MFC0	访问 CP0	Move from CP0
26	MTC0	访问 CP0	Move to CP0
27	ERET	中断返回	EPC -> PC
28	DIVU	无符号除	\$Rs/\$Rt -> LO
29	MFLO	读 LO 寄存器	LO -> \$Rd
30	LB	读字	MEM[\$Rs+Offset][Addr1..0] -> Rt
31	BGTZ	大于 0 跳转	\$Rs > 0 Jump

2 总体方案设计

2.1 单周期 CPU 设计

本次我们采用的方案硬布线控制，且采用哈佛结构进行内存管理（指令存储与数据存储分离）的方案，即利用控制单元生成控制信号，控制整个 CPU 随时钟变化进行工作。

总体结构图如图 2.1 所示。

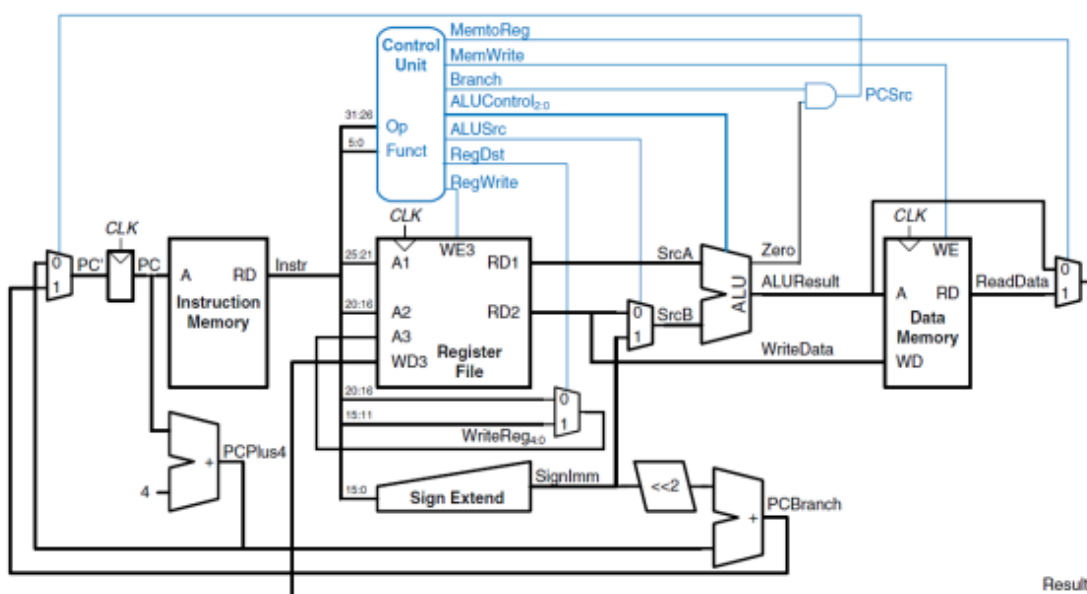


图 2.1 总体结构图

2.1.1 主要功能部件

1. 指令译码器

根据 MIPS 3 种类型的指令，将指令的各个位段提取出来，即可完成指令译码器，输出的位段有 op（6 位）、rs（5 位）、rt（5 位）、rd（5 位）、sham（5 位）、funct（6 位）、imm16（16 位）、imm26（26 位）。

利用 op 位(6bit) 与 funct 位(6bit)，可以为所有指令生成一个多元与式，以确定当前指令类型。其关系表如表 2.1 指令类型与关系表所示：

表 2.1 指令类型与关系表

华中科技大学课程设计报告

指令	op5	op4	op3	op2	op1	op0	fn5	fn4	fn3	fn2	fn1	fn0
add	0	0	0	0	0	0	1	0	0	0	0	0
addi	0	0	1	0	0	0	x	x	x	x	x	x
addiu	0	0	1	0	0	1	x	x	x	x	x	x
addu	0	0	0	0	0	0	1	0	0	0	0	1
and	0	0	0	0	0	0	1	0	0	1	0	0
andi	0	0	1	1	0	0	x	x	x	x	x	x
sll	0	0	0	0	0	0	0	0	0	0	0	0
sra	0	0	0	0	0	0	0	0	0	0	1	1
srl	0	0	0	0	0	0	0	0	0	0	1	0
sub	0	0	0	0	0	0	1	0	0	0	1	0
or	0	0	0	0	0	0	1	0	0	1	0	1
ori	0	0	1	1	0	1	x	x	x	x	x	x
nor	0	0	0	0	0	0	1	0	0	1	1	1
lw	1	0	0	0	1	1	x	x	x	x	x	x
sw	1	0	1	0	1	1	x	x	x	x	x	x
beq	0	0	0	1	0	0	x	x	x	x	x	x
bne	0	0	0	1	0	1	x	x	x	x	x	x
slt	0	0	0	0	0	0	1	0	1	0	1	0
slti	0	0	1	0	1	0	x	x	x	x	x	x
sltu	0	0	0	0	0	0	1	0	1	0	1	1
j	0	0	0	0	1	0	x	x	x	x	x	x
jal	0	0	0	0	1	1	x	x	x	x	x	x
jr	0	0	0	0	0	0	0	0	1	0	0	0
sys	0	0	0	0	0	0	0	0	1	1	0	0

利用上述关系，可以构造任意指令的布尔信号表达式。以 add（R 型） 为例，其布尔信号为

$$\text{add} = \sim\text{op5} \ \& \ \sim\text{op4} \ \& \ \sim\text{op3} \ \& \ \sim\text{op2} \ \& \ \sim\text{op1} \ \& \ \sim\text{op0} \ \& \ \text{fn5} \ \& \ \sim\text{fn4} \ \& \ \sim\text{fn3} \ \&$$

华中科技大学课程设计报告

$\sim fn2 \& \sim fn1 \& \sim fn0$;

以 j（非 R 型）为例，其布尔信号为

$$j = \sim op5 \& \sim op4 \& \sim op3 \& \sim op2 \& op1 \& \sim op0。$$

同理可得到所有指令的布尔信号表达式，利用 logisim 的分析电路功能，利用这些表达式自动生成布尔信号电路。由于 logisim 分析电路将输入输出门个数限制在 12 个，故需生成 2 次电路（1 次 12 条指令），再将其封装成一个完整的指令布尔信号电路。

2. PC 更新单元

将整个 CPU 分成 2 个部分。首先观察 PC 更新部分，即取指单元（IFU）。分析所要实现的 28 条指令可以发现，除 beq、bne、bgtz、j、jal、jr 这 6 条指令外，大部分指令对于 PC 的操作都是简单的 $PC += 4 \text{ bytes}$ 。利用第 2 个设计中的指令布尔信号，可以得到当前是否为跳转指令。可以把 24 条指令大致分为 4 组，第 1 组为顺序取指（大部分指令），第 2 组为相对 PC 跳转取指（beq/bne），第 3 组为绝对跳转取指（j/jal），第 4 组为寄存器间接跳转取指（jr）。于是可以得到，此时需要 3 个 1 选 2 选择器，以根据不同指令选择不同的 PC 更新方式，由此产生了 3 个控制信号。假设 $PC + 4$ 为默认更新方式，则第 2 组选择控制信号应为 $Bne \& \sim Eq \mid Beq \& Eq \mid Bgtz \& \sim Less \& \sim Eq$ ，第 3 组选择控制信号为 $J \mid Jal$ ，第 4 组选择控制信号为 Jr ，将上述逻辑综合成一张关系表，如表 2.2 PC 更新逻辑表所示。

表 2.2 PC 更新逻辑表

Beq	Bne	Bgtz	Eq	Less	J	Jal	Jr	PC
1	0	0	1	x	0	0	0	PC 相对
0	1	0	0	x	0	0	0	PC 相对
0	0	1	0	0	0	0	0	PC 相对
x	x	x	x	x	1	x	0	伪绝对
x	x	x	x	x	x	1	0	伪绝对
x	x	x	x	x	x	x	1	PC+OF

华中科技大学课程设计报告

3. 运算器

根据 ALU_OP，利用多选器将不同部件（如移位器、加法器、乘法器等）的输出选择输出即可。

表 2.3 算术逻辑运算单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
X	输入	32	操作数 X
Y	输入	32	操作数 Y
ALU_OP	输入	4	运算器功能码，具体功能见下表
Result	输出	32	ALU 运算结果
Result2	输出	32	ALU 结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零
OF	输出	1	有符号加减溢出标记，其他操作为零
UOF	输出	1	无符号加减溢出标记，其他操作为零
Equal	输出	1	$Equal=(x==y)?1:0$ ，对所有操作有效

ALU 唯一需要注意的便是溢出检测单元。

无符号溢出检测：无符号加时，若产生溢出，则加法器最高位进位为 1；无符号减时，若产生溢出，则加法器（Y 进行处理后直接输入 32 位全并行加法器）最高位进位为 0。利用 4 选 16 选择电路将无符号加/减的溢出输出至 UOF。即选择器输入 5 的逻辑式为： $in5 = adder_cout$ ，选择器输入 6 的逻辑式为： $in6 = \sim adder_cout$ 。

有符号溢出检测：有符号加时，产生溢出的情况只可能为 ++->- 与 -->+，故可得溢出码逻辑式为： $addOF = (X_f \& Y_f \& !S_f) | (!X_f \& !Y_f \& S_f)$ ；同理，有符号减时，产生溢出的情况只可能为 +->- 与 -+>+，故可得溢出码逻辑式为： $subOF = (X_f \& !Y_f \& !S_f) | (!X_f \& Y_f \& S_f)$ 。同无符号溢出检测，利用 4 选 16 选择电路将有符号加/减的溢出输出至 OF。

4. 寄存器堆 RF

利用 logisim 平台构建一个 MIPS 寄存器组，内部包含 32 个 32 位寄存器，其具体功能如下，具体封装文件为 regfile.circ，如表 2.4 芯片引脚与功能描述所示。

华中科技大学课程设计报告

表 2.4 芯片引脚与功能描述

引脚	输入/输出	位宽	功能描述
R1#	输入	5	读寄存器 1 编号
R2#	输入	5	读寄存器 2 编号
W#	输入	5	写入寄存器编号
Din	输入	32	写入数据
WE	输入	1	写入使能信号，为 1 时，CLK 上跳沿将 Din 数据写入 W#寄存器
CLK	输入	1	时钟信号，上跳沿有效
R1	输出	32	R1#寄存器的值
R2	输出	32	R2#寄存器的值
\$s0	输出	32	编号为 16 的寄存器的值
\$s1	输出	32	编号为 17 的寄存器的值
\$s2	输出	32	编号为 18 的寄存器的值
\$ra	输出	32	编号为 31 的寄存器的值

在设计寄存器组的实现方式时，唯一需要注意的地方便是寄存器组的使能端。只要利用合理的组合逻辑，使得同一时刻只有 1 个寄存器（写目标寄存器）的使能端为 1（有效），便可实现寄存器组的输入功能。在此种情况下，时钟源与输入数据只需同时连接 31 个寄存器即可（除开 0 号寄存器），无需而外逻辑。下面利用一个单独的模块，实现上述使能端信号选择功能。

给定一个 W#（5 bits）信号，表示当前写入目标寄存器的编号，则只需利用一个解复用器，将 W# 作为选择端，常量 1 作为输入端，即可实现同时只有目标寄存器的写使能信号为 1 的设计。输出为 8 组使能信号，每组 4 bits 使能信号，同时只有 1 bit 使能信号为 0。（分组是为了使能信号接入寄存器组时的连线美观）

只需利用多路选择器，将 R1#/R2#（读出寄存器编号）作为选择信号，32 寄存器输出端最为选择输入端即可实现此功能。

将上述 2 个设计综合在一起，即可得到完整的寄存器组。唯一值得注意的地方，需要使 0 号寄存器（\$zero）保持常 0。

2.1.2 数据通路的设计

数据通路的实现并非采用工程化的方式，而是利用逐步扩展的方式完成整个数据通路的实现。下面是各个阶段的设计思路：

常规 R 型指令数据通路：只需寄存器组与 ALU 即可实现常规 R 型指令的数据通路，此时只需 2 个控制信号，RegWe 控制寄存器组的写使能，ALUOp 控制 ALU 运算逻辑，此类指令包括 add、addu、and、sub、or、nor、slt、sltu。

常规 I 型指令数据通路：只需在 R 型指令通路的基础上，利用新的控制信号将 ALU_Y 改为立即数输入，同时将写入寄存器编号 RW# 改为 rt 即可，此时需要 3 个新的控制信号，并加入 3 个多选器，RegDst 选择写入寄存器编号（rt 或 rd），ALUSrc 选择 Y 端输入（rt 或 imm16），ExtOp 选择立即数扩展方式（无符号或有符号扩展），此类指令包括 addi、addiu、andi、ori、slti。值得一提的是，addiu 指令对 imm16 进行的也是符号扩展。

移位指令数据通路：移位指令的源操作数不再为（rs），而是（rt），且移位字段不是由寄存器/imm16 给出，故需要 1 个新的信号 ALUSham，并加入 2 个多选器，以改变 ALU 2 个输入端的值。

访存指令数据通路：从寄存器组到 ALU 中间的数据通路，lw/sw 指令与常规 I 型指令间没有差异，唯一不同的是，lw/sw 指令 ALU 的输出端不再是普通的值，而是计算好的地址值。由于访存操作的存在，需要引入新的主存组件至数据通路中，并增加 2 个新的控制信号，以控制主存的读写。对于 lw 指令，需要从主存读取数据至寄存器，增加新信号 RAMtoReg，并增加一个多选器，以选择寄存器组的输入数据（ALU_Result 或 RAMDataOut）；对于 sw 指令，需要增加 1 个新信号 RAMWe，以控制对主存的写访问，并将（rt）接至主存加载数据端（RAMDataIn）。条件跳转指令数据通路：beq/bne 指令无需修改数据通路，只需给出合适的控制信号，使得 $ALU_X = (rs)$ ， $ALU_Y = (rt)$ ，并从 ALU 得到 Eq 标志量，从而在 IFU 中完成指令跳转功能。

直接跳转指令数据通路：j 与 jr 指令无需修改数据通路，直接通过前述 IFU 设计即可实现指令跳转功能。jal 指令还需借助原数据通路将 rs 值取出加 4 后写入 \$a0 寄存器，此时 $ALU_X = (rs)$ ， $ALU_Y = (4)$ ， $ALU_OP = ADD$ ， $RW\# = 0x1f$ ，故需修改数据通路。

华中科技大学课程设计报告

系统调用指令数据通路：关于 `syscall` 的实现，无需修改数据通路，只需将 `$v0` 的值从寄存器组引出，并将其与 `0xah` 进行比较，再结合 `Syscall` 指令布尔信号，即可得到 `Halt` 停机信号与输出七段管信号。利用 `Halt` 信号，对时钟源进行简单处理，即可实现 `syscall 0xah` 功能。

扩展指令数据通路：`divu` 需要增加一个 `LO` 寄存器与一个 `WriteToLO` 信号（使能控制信号，将 `ALU` 除的结果在合适的时机写入 `LO` 寄存器），需将 `ALUOp` 译码成除法操作；`mflo` 需要修改写回寄存器的数据通路，额外增加一个控制信号，将写回寄存器的数据修改为 `LO` 寄存器的值；`bgtz` 只需在原来 `branch` 分支跳转成功逻辑电路的基础上增加判断 `bgtz` 跳转成功的逻辑即可，即 $\text{branch} = (\text{Beq} \ \& \ \text{Eq}) \mid (\text{Bne} \ \& \ \sim\text{Eq}) \mid (\text{Bgtz} \ \& \ \sim\text{Eq} \ \& \ \sim\text{Less})$ ；`lb` 指令只需将 `RAM` 读出的数据进行字节选择并进行符号扩展，得到新的一个 32 bit 的数据，再利用控制信号 `RAMByte` 在原来 `RAM` 数据输出和新数据间进行多选即可。

2.1.3 控制器的设计

首先对于控制信号进行统计，包括各个主要部件所需要输入的控制信号，以及数据通路合并表中所示的具有多输入的主要部件需要进行输入选择的控制信号，并且对各个统计信号的各种取值情况进行定义，统计得到的控制信号以及说明如表 2.5 主控制器控制信号的作用说明所示。

表 2.5 主控制器控制信号的作用说明

控制信号	取值	说明
<code>ALUOp</code>	0-15	<code>ALU</code> 不同计算
<code>RegWe</code>	0	寄存器组写使能无效
	1	寄存器组写使能有效
<code>ALUSrc</code>	0	<code>ALU_Y</code> 的输入来自 <code>\$rt</code>
	1	<code>ALU_Y</code> 的输入来自 <code>imm16</code>
<code>ALUSham</code>	0	<code>ALU_Y</code> 的输入来自 <code>\$rt/imm16</code>
	1	<code>ALU_Y</code> 的输入来自 <code>sham</code>
<code>RegDst</code>	0	寄存器组写寄存器号为 <code>rt</code>
	1	寄存器组写寄存器号为 <code>rd</code>

华中科技大学课程设计报告

控制信号	取值	说明
ExtOp	0	imm16 进行无符号扩展
	1	imm16 进行有符号扩展
RAMWe	0	RAM 写使能无效
	1	RAM 写使能有效
RAMtoReg	0	寄存器组写寄存器数据来自 ALU
	1	寄存器组写寄存器数据来自 RAM
Beq/Bne/B	0	当前不是对应的指令（beq/bne/bgtz/j/jal/jr）
gtz/J/Jal/Jr	1	当前是对应的指令（beq/bne/bgtz/j/jal/jr）
Syscall	0	当前指令不是系统调用
	1	当前指令是系统调用
WritetoLO	0	LO 寄存器写使能无效
	1	LO 寄存器写使能有效
LOtoReg	0	寄存器组写寄存器数据来自 ALU/RAM
	1	寄存器组写寄存器数据来自 LO 寄存器
RAMByte	0	RAM 的输出为字数据
	1	RAM 的输出为字节扩展数据

对照所有控制信号，依次分析各条指令，分析该指令执行过程中需要哪些控制信号，对于与本条指令无关的控制信号，控制信号的取值一律为 0，以简化控制器电路的设计。设计好数据通路，即可按照指令特性与数据通路走向，确定对于每一个指令的控制信号集。控制信号结果如表 2.6 控制信号表的框架（1）、表 2.7 控制信号表的框架（2）、表 2.8 控制信号表的框架（3）、表 2.9 控制信号表的框架（4）。利用多路或门，即可实现所有控制信号的生成，从而实现完整的控制单元。

表 2.6 控制信号表的框架（1）

华中科技大学课程设计报告

控制信号	add	addi	addiu	addu	and	andi	sll	sra
ALUOp0	1	1	1	1	1	1	0	1
ALUOp1	0	0	0	0	1	1	0	0
ALUOp2	1	1	1	1	1	1	0	0
ALUOp3	0	0	0	0	0	0	0	0
RegWe	1	1	1	1	1	1	1	1
1enable								
ALUSrc	0	1	1	0	0	1	x	x
0rt								
1imm16								
ALUSham	0	0	0	0	0	0	1	1
1=sham								
RegDst	1	0	0	1	1	0	1	1
0=rt								
1=rd								
ExtOp	x	1	1	x	x	0	x	x
1=s								
RAMWe	0	0	0	0	0	0	0	0
RAMtoReg	0	0	0	0	0	0	0	0
Beq	0	0	0	0	0	0	0	0
Bne	0	0	0	0	0	0	0	0
Jmp	0	0	0	0	0	0	0	0
Jal	0	0	0	0	0	0	0	0
Jr	0	0	0	0	0	0	0	0
Syscall	0	0	0	0	0	0	0	0

表 2.7 控制信号表的框架（2）

华中科技大学课程设计报告

控制信号	srl	sub	or	ori	nor	lw	sw	beq
ALUOp0	0	0	0	0	0	1	1	x
ALUOp1	1	1	0	0	1	0	0	x
ALUOp2	0	1	0	0	0	1	1	x
ALUOp3	0	0	1	1	1	0	0	x
RegWe	1	1	1	1	1	1	0	0
1enable								
ALUSrc	x	0	0	1	0	1	1	0
0rt								
1imm16								
ALUSham	1	0	0	0	0	0	0	0
1=sham								
RegDst	1	1	1	0	1	0	x	x
0=rt								
1=rd								
ExtOp	x	x	x	0	x	1	1	x
1=s								
RAMWe	0	0	0	0	0	0	1	0
RAMtoReg	0	0	0	0	0	1	x	0
Beq	0	0	0	0	0	0	0	1
Bne	0	0	0	0	0	0	0	0
Jmp	0	0	0	0	0	0	0	0
Jal	0	0	0	0	0	0	0	0
Jr	0	0	0	0	0	0	0	0
Syscall	0	0	0	0	0	0	0	0

表 2.8 控制信号表的框架（3）

华中科技大学课程设计报告

控制信号	bne	slt	slti	sltu	j	jal	jr	syscall
ALUOp0	x	1	1	0	x	1	x	x
ALUOp1	x	1	1	0	x	0	x	x
ALUOp2	x	0	0	1	x	1	x	x
ALUOp3	x	1	1	1	x	0	x	x
RegWe	0	1	1	1	0	1	0	0
1enable								
ALUSrc	0	0	1	0	x	x	x	0
0rt								
1imm16								
ALUSham	0	0	0	0	x	x	x	0
1=sham								
RegDst	x	1	0	1	x	x	x	x
0=rt								
1=rd								
ExtOp	x	x	1	x	x	x	x	x
1=s								
RAMWe	0	0	0	0	0	0	0	0
RAMtoReg	0	0	0	0	x	0	x	x
Beq	0	0	0	0	0	0	0	0
Bne	1	0	0	0	0	0	0	0
Jmp	0	0	0	0	1	0	x	0
Jal	0	0	0	0	0	1	x	0
Jr	0	0	0	0	0	0	1	0
Syscall	0	0	0	0	0	0	0	1

表 2.9 控制信号表的框架（4）

华中科技大学课程设计报告

控制信号	divu	mflo	lb	bgtz
ALUOp0	0	x	1	1
ALUOp1	0	x	0	1
ALUOp2	1	x	1	0
ALUOp3	0	x	0	1
RegWe	0	1	1	0
1enable				
ALUSrc	0	x	1	0
0rt				
1imm16				
ALUSham	0	x	0	0
1=sham				
RegDst	x	1	0	x
0=rt				
1=rd				
ExtOp	x	x	1	x
1=s				
RAMWe	0	0	0	0
RAMtoReg	0	0	1	0
Beq	0	0	0	0
Bne	0	0	0	0
Jmp	0	0	0	0
Jal	0	0	0	0
Jr	0	0	0	0
Syscall	0	0	0	0
WriteToLO	1	0	0	0
LOToReg	0	1	0	x
RAMByte	0	0	1	x

2.2 中断机制设计

2.2.1 总体设计

中断在本质上，可以等同于 J/Jal/Jr 等指令，但由于其具有一定的特殊性，在实现上比 J/Jal/Jr 等指令复杂得多。多级嵌套中断的整体过程如图 2.2 多级嵌套中断流程图所示。可以看到一次中断执行大致分为 5 个部分：中断识别与保护断点，保护现场与屏蔽字，中断处理服务例程，恢复现场与屏蔽字，中断返回。需要实现以下电路：中断识别电路、断点保存电路以及 3 条特殊的中断相关指令。

利用中断识别电路，利用硬件识别当前中断源与中断号，利用硬件将 PC 切换至中断处理程序地址，可以完成中断识别的任务，而中断号保存于 Cause 寄存器。

利用断点保存电路，简单地将旧的 PC_Next 值存于一个新的寄存器，以起到断点保存的作用，将这个新寄存器称为 EPC 寄存器。

除此之外，开/关中断、保护现场与屏蔽字、恢复现场与屏蔽字以及中断返回，皆可利用 mfc0、mtc0、eret 以及一些通用指令（如 lw/sw 实现栈帧）完成，利用栈帧可实现多级中断的现场维护，使得保护现场与恢复现场都在栈帧上进行操作，使得中断得以嵌套，不会影响上一级中断现场的保护与恢复。其中，屏蔽字与中断使能位保存于 Status 寄存器。将上述提到的 3 个特殊寄存器 Status（中断使能位与屏蔽字）、EPC（断点）、Cause（中断号）封装成一个组件，实现一个简易的 CP0 协处理器（See MIPS Run 第二版）。

综上所述，所需实现的任务有中断识别电路、简易 CP0 协处理器、PC 保护与恢复电路逻辑。

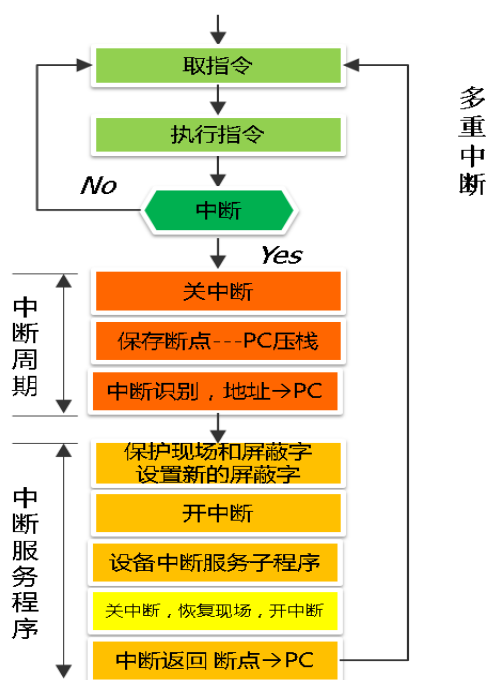


图 2.2 多级嵌套中断流程图

2.2.2 硬件设计

中断识别电路如所示。当有中断来时，结合中断屏蔽字，判定当前是否产生中断，将中断的发生信号接入优先编码器，即可利用优先编码器的特性，识别具有最高响应优先级的中断，并生成其中断号。同时优先编码器的有效位输出与中断使能信号相与，可得到最终的中断发生信号，其表示当前系统是否有中断发生。当中断得到响应与处理后，将相应中断寄存器中锁存的中断信号利用清零端清零即可。其中，中断屏蔽字由 Status 寄存器 8~14 位提供（此次实验只需 8-10 位），中断使能信号由 Status 寄存器 0 位提供（此次实验利用 1 个逻辑非门将其功能反置，0 表示开中断，1 表示关中断，使得整个 CPU 默认处于开中断状态），中断号输出接至 Cause 寄存器。

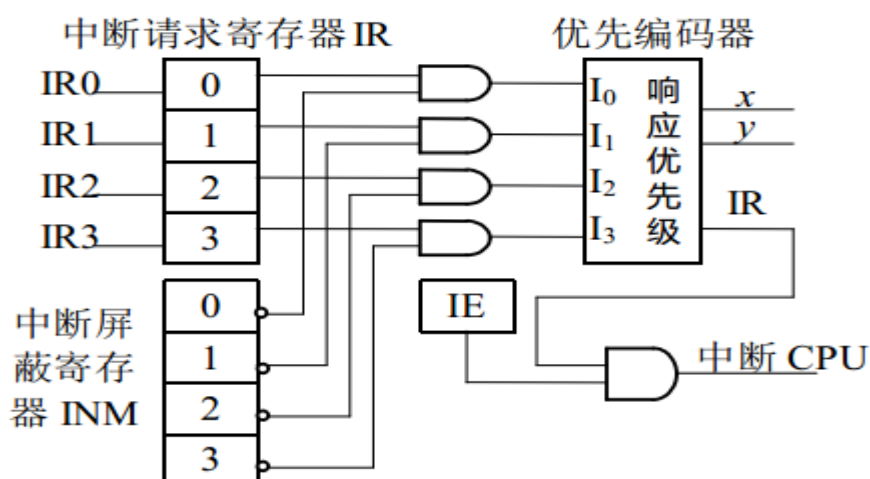


图 2.3 中断识别与响应电路

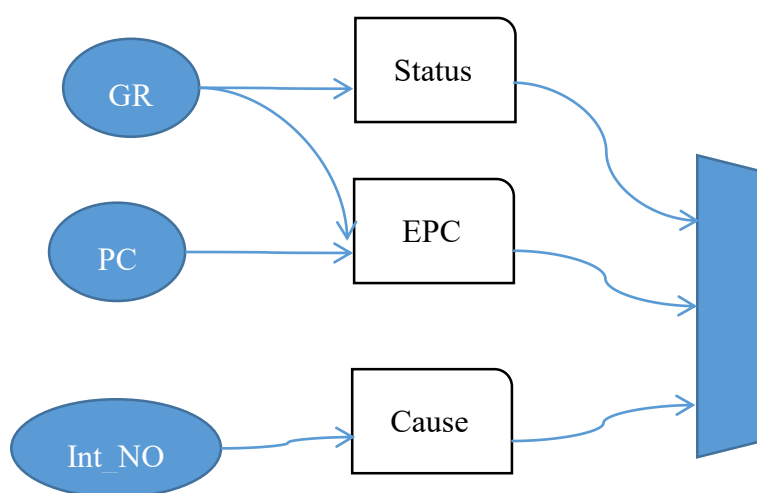


图 2.4 简易 CP0 示意图

CP0 协处理器：在此次实验中，其本质是一个简易的寄存器组，拥有 3 个寄存器，分别为 Status、EPC、Cause，用于保存屏蔽字、中断使能位、断点（旧 PC_Next）、中断号。其功能与寄存器组一致，包括输入与输出 2 部分，其原理图如图 2.4 简易 CP0 示意图所示。输出部分逻辑简单，根据 MFC0 指令中 rd 部分对 3 个寄存器输出进行多选即可。输入部分逻辑更复杂：Status 只有 1 个输入，为 MTC0 中 rt 对应的通用寄存器组中寄存器的值；Cause 只有 1 个输入，本次实验无需利用软件对其进行修改，故其输入仅为中断识别电路中的优先编码器的中断号输出；EPC 有 2 个输入，一个为 MTC0 指定的通用寄存器数据，另一个即为用于保护断点的旧 PC 值，利用中断信号的有无对其进行多选即可。

PC 更新：通过分析中断过程可以发现，只有在 2 个时机需要改变 PC 值，即中

断进入与中断返回（eret）时。只需在原来 IFU 模块中的 PC 更新处增加 2 个多路器，分别输入 0x800（中断程序地址）与 EPC 即可完成中断进入与中断返回的硬件支持。

2.2.3 软件设计

要完成多级嵌套中断的实现，需要 3 条指令的支持，分别为 mfc0、mtc0 和 eret，其作用分别为从 CP0 取值至通用寄存器组、将通用寄存器组的值送 CP0、以及中断返回。其硬件设计已在 2.2.3 节讲述，即增加 CP0RegWe 控制信号以完成对 CP0 中寄存器的写、CP0toReg 控制信号以完成对通用寄存器的写（将数据输入端由 ALU/RAM/LO 切换为 CP0）、Eret 控制信号以完成对 PC 的恢复（EPC 送 PC）。这 3 条指令的控制信号如表 2.10 中断相关指令控制信号表所示。

表 2.10 中断相关指令控制信号表

华中科技大学课程设计报告

控制信号	mfc0	mtc0	eret
ALUOp0	x	x	x
ALUOp1	x	x	x
ALUOp2	x	x	x
ALUOp3	x	x	x
RegWe	1	0	0
ALUSrc	x	x	x
ALUSham	x	x	x
RegDst	0	x	x
ExtOp	x	x	x
RAMWe	0	0	0
RAMtoReg	0	x	x
Beq	0	0	0
Bne	0	0	0
Jmp	0	0	0
Jal	0	0	0
Jr	0	0	0
Syscall	0	0	0
WriteToLO	0	0	0
LOToReg	0	x	0
RAMByte	0	0	0
Bgtz	0	0	0
CP0RegWe	0	1	0
CP0ToReg	1	x	0
Eret	0	0	1

2.3 流水 CPU 设计

2.3.1 总体设计

流水线是一种实现多条指令重叠执行的技术。一个 MIPS 指令通过包含以下 5 个处理步骤：从指令存储器中读指令，称为取指阶段（IF）；指令译码的同时读取寄存器，称为译码/取数阶段（ID）；执行操作，进行逻辑运算，称为执行阶段（EX）；从数据存储器中读取操作数，或将数据写入数据存储器，称为访存阶段（MEM）；将结果写回寄存器，称为写回阶段（WB）。

可以看出，单周期 CPU 的实现与器件排布可以十分简单地扩展、拉伸成 5 段流水，唯二需要注意的 2 点：写寄存器相关的 3 个输入即写寄存器号、写寄存器使能、写寄存器数据，需要从寄存器组周围大迁徙至 WB 阶段；需要仔细推敲地址计算与分支跳转逻辑的放置阶段，本次实验决定按照《数字设计与计算机体系结构（Digital Design and Computer Architecture）》一书中所提供的思路，将其置于 ID 阶段，与译码、取数同时进行，降低误取深度，提升流水线性能。

除此之外，还需解决流水线 3 大经典问题，结构冒险、数据冒险与控制冒险。由于本次实验采用哈佛结构，故取指与访问不会存在结构冒险，同时由于 MIPS 规定只有 L/S 型指令可以访存，基于以上 2 点可以得出当前流水线不存在结构冒险；通过重定向，将 MEM 或 WB 阶段的数据通过旁路接至 EX 端供计算使用，可以解决大部分数据冒险，再利用插气泡的方式解决 Load-Use 数据冒险即可；当误取指令后，需要清空前段流水线，将误取指令清除，由于将地址计算放置于 ID 段，故只需清除 ID/IF 段流水寄存器。需要注意的是，由于将地址计算置于 ID 段，需要在此段解决 Beq/Bne/Bgtz 存在的数据冒险，同样地可以利用重定向与插气泡的方式解决。

经过上述分析，可以得到流水线基本原理图如图 2.5 流水线原理图（Digital Design and Computer Architecture）所示。实际实现与其存在一些差异，如 ALU 输入端的多选、地址计算与跳转逻辑以及 PC 更新逻辑，具体差异参见实现一节。

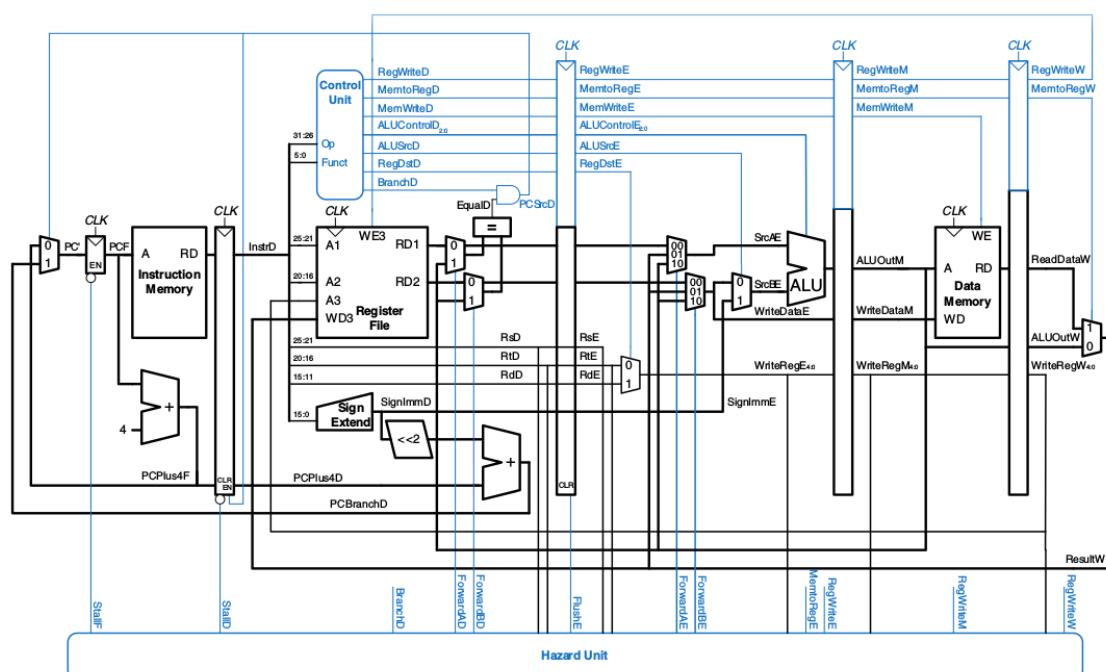


图 2.5 流水线原理图 (Digital Design and Computer Architecture)

2.3.2 流水接口部件设计

通过观察流水线各个阶段，只需将后面阶段所需信号与数据通过流水线向后传递即可。各个流水寄存器输入输出设计如下所示：

IF/ID: PC (ID/EX/MEM/WB)、IR (ID/EX/MEM/WB)

ID/EX : PC (EX/MEM/WB) 、 IR (EX/MEM/WB) 、
WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall (WB)、RAMWe/RAMByte (MEM)、
RegDst/ALUOp/ALUSrc/ExtOp/ALUSham/Jal (EX)、rs/rt/rd/sham/imm16/R1 (EX)、
R2 (EX/MEM)

EX/MEM : PC (MEM/WB) 、 IR (MEM/WB) 、
WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall (WB)、RAMWe/RAMByte (MEM)、
rt/R2 (MEM)、Result/RW# (MEM/WB)

MEM/WB : PC 、 IR 、 WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall 、
RAMData/Result/RW#。

利用寄存器将上述信号与数据在流水段之间随时钟进行锁存即可。

其中，将 PC 与 IR 充满整个流水线，方便调试。

华中科技大学课程设计报告

2.3.3 理想流水线设计

理想流水线无需考虑数据冒险与控制冒险，只需利用 4 个流水寄存器将 CPU 分为 5 段流水即可，其原理图如图 2.6 理想流水线原理图所示。

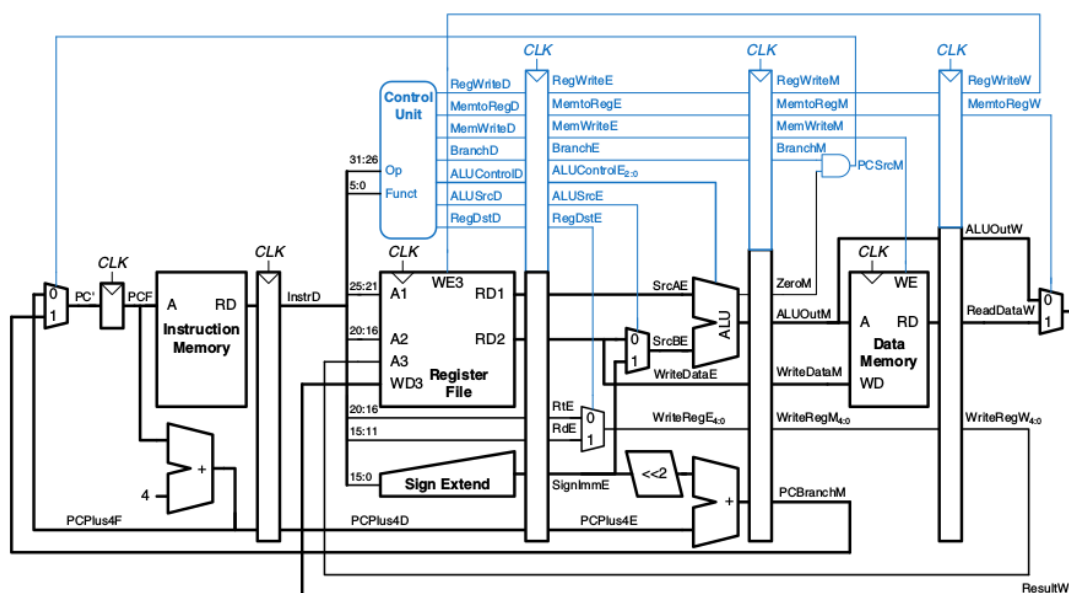


图 2.6 理想流水线原理图

2.4 数据转发流水线设计

在总体设计中提到过，大部分数据冒险都可以用转发解决，故在此次实验中先完成数据转发模块，再进行利用气泡方式清除剩余数据冒险。

在 ID 阶段，由于将地址计算与分支逻辑置于此阶段，故需要将此进行数据转发。其转发源头为 MEM ALU 计算结果 Result 以及 WB 写回寄存器的值 RegData。以 R1 为例，其转发逻辑如下所示（同理，R2 的转发只需将 rs 换成 rt 即可）：

```
if ((IF/ID.rs != 0) && (IF/ID.rs == EX/MEM.RW#) && EX/MEM.RegWe) begin
    IF/ID.ForwardA = 10 (MEM)
end else if ((IF/ID.rs != 0) && (IF/ID.rs == MEM/WB.RW#) && MEM/WB.RegWe)
begin
    IF/ID.ForwardA = 01 (WB)
end else begin
    IF/ID.ForwardA = 00 (no forwarding)
```

end

在 EX 阶段，若此时此阶段为任何使用寄存器的指令，且 MEM 与 WB 阶段对其用到的某个寄存器进行写，则会发生数据冒险，需要进行转发。以 ALU_X 端输入为例，其转发逻辑如下所示（ALU_Y 端转发同理，将 rs 替换成 rt 即可）：

```
if ((ID/EX.rs != 0) && (ID/EX.rs == EX/MEM.RW#) && EX/MEM.RegWe) begin
    ID/EX.ForwardA = 10 (MEM)
end else if ((ID/EX.rs != 0) && (ID/EX.rs == MEM/WB.RW#) && MEM/WB.RegWe)
begin
    ID/EX.ForwardA = 01 (WB)
end else begin
    ID/EX.ForwardA = 00 (no forwarding)
end
```

最后，在 MEM 阶段，若此时此阶段为 sw 指令，WB 阶段为 lw 指令，则利用转发可以解决数据冒险，此种不插入气泡即可解决此种看上去为 Load-Use 的数据冒险，其转发逻辑如下所示：

```
MEMForward = (MEM/WB.RegWe && EX/MEM.RAMWe && EX/MEM.rt != 0 &&
EX/MEM.rt == MEM/WB.RW#)
```

综上，利用上述 3 处的转发，即可解决大部分数据冒险。

2.5 气泡式流水线设计

解决完大部分数据冒险后，仍有一种数据冒险无法靠转发解决，即 Load-Use 冒险。当 EX 端为寄存器读指令，而 MEM 阶段为 lw 指令时，由于写入寄存器的值不在 EX/MEM 与 MEM/WB 流水寄存器，由于数据从存储器读出具有一定的延迟，故用转发不适合解决此种数据冒险。最有效地办法为插入气泡，使 Use 指令停一个周期，等 lw 指令来到 WB 阶段后，再利用原先的转发逻辑解决数据冒险。而 Load-Use 阶段应置于 ID - EX 段，其检测逻辑如下所示：

```
lwstall = ID/EX.rt != 0 && ID/EX.RAMtoReg && (ID/EX.rt == IF/ID.rs || ID/EX.rt
== IF/ID.rt)
```

当检测到 Load-Use 冒险后，在下一个上升沿，关闭 PC 寄存器与 IF/ID 流水寄存器的写使能，打开 ID/EX 流水寄存器的同步清零信号，即可成功地在 EX 段插

华中科技大学课程设计报告

入一个气泡，使得 ID 段的 Use 指令延后一个周期，达到清除冒险的任务。

除此之外，还存在一个数据冒险。由于将地址计算放置于 ID 段，当 EX 段指令为写寄存器指令或 MEM 段为 lw 指令时，无法利用 ID 段的转发逻辑解决此冒险（理由与 Load-Use 冒险一样，ALU 运算器与 RAM 读取皆存在一定的时延），故需向流水线 EX 插入一个气泡，其检测逻辑如下所示：

$$\text{branchstall} = (\text{IF/ID.JmpNeedReg} \ \&\& \ \text{ID/EX.RegWe} \ \&\& \ \text{ID/EX.RW\#} \neq 0 \ \&\& \ (\text{ID/EX.RW\#} == \text{IF/ID.rs} \parallel \text{ID/EX.RW\#} == \text{IF/ID.rt}))$$
$$\parallel (\text{IF/ID.JmpNeedReg} \ \&\& \ \text{EX/MEM.RAMtoReg} \ \&\& \ \text{EX/MEM.RW\#} \neq 0 \ \&\& \ (\text{EX/MEM.RW\#} == \text{IF/ID.rs} \parallel \text{EX/MEM.RW\#} == \text{IF/ID.rt}))$$

当检测到分支指令数据冒险时，插入气泡即可解决。

最后，利用气泡解决控制冒险。当指令为 J/Jal/Jr 或者成功跳转的 Beq/Bne/Bgtz 时，流数线会误取 1 条指令至 IF/ID 流水寄存器，所以需要及时将误取的指令清零。只需将清零信号接至 IF/ID 流水寄存器的同步清零端，在合适的时机清空流水寄存器即可插入气泡（此时无需暂停流水线），其检测逻辑即为单周期 CPU 中 PC 更新逻辑中的信号，即为

$$\text{flush} = (\text{J} \parallel \text{Jal} \parallel \text{Jr}) \parallel (\text{Beq} \ \&\& \ \text{Eq}) \parallel (\text{Bne} \ \&\& \ \sim \text{Eq}) \parallel (\text{Bgtz} \ \&\& \ \sim \text{Eq} \ \&\& \ \sim \text{Less})$$

至此，所有冒险全部解决。

2.6 动态分支预测机制

通过《计算机组成与设计-硬件/软件接口》一书以及网络上 Princeton、MIT、CMU 有关于动态分支预测的课件的学习与研究，得到了一个较为简易的带 BTB 表的动态分支预测机制。

采用插入气泡以清除误取指令的方式来处理控制冒险会使得 CPU 的效率很低，所以采取分支预测的方式，提前预测分支指令可能的跳转地址。若分支预测的准确性较高（甚至达到 80% 以上），则可以大幅度降低因误取指令而带来的流水线损耗，提升 CPU 处理效率。

动态分支预测策略是通过观察上一次执行分支指令时分支是否发生，来决定此次预测是否执行分支。实现此策略的方法为分支预测缓存（Branch Target Buffer），将之前分支预测的历史保存在一块较小的缓存（全相联存储器）中，当在 IF 阶段遇到分支指令时，先尝试从缓存中读取历史，若读取成功，则根据历史记录进行分

华中科技大学课程设计报告

支预测；若读取失败，则重新计算地址，采取初始策略（跳转或不跳转），进行分支预测，同时更新 BTB 对应表项。

当预测失败时，只需像之前的设计那样，清除在流水寄存器的误取指令即可，并将 PC 修正为正确的跳转地址。

换言之，带 BTB 的动态分支预期器由 2 大部分组成，预测策略状态机与 BTB 跳转历史缓存。

此次实验采用 2 位的预测策略状态机，其状态变迁如图 2.7 2 位预测状态机变迁图所示。其状态分为 4 个，分别为强跳转（Strongly Taken）、弱跳转（Weakly Taken）、弱不跳转（Weakly Not Taken）、强不跳转（Strongly Not Taken）。根据实际跳转方向，不断修改当前状态。而预测的方向，只于当前状态有关。所以这是一个 Moore 型的状态机。

BTB 表设计为一个全相联映射的缓存，其地址划分如表 2.11 BTB 表项地址划分所示。有效位表示当前表项是否被占据，预测位表示对应分支的预测策略（1 位表示是否跳转，0 位表示策略强度），分支标记保存分支指令的 PC 值，用于识别存储于此行的历史属于哪个分支指令，预测地址为对应的跳转地址缓存。

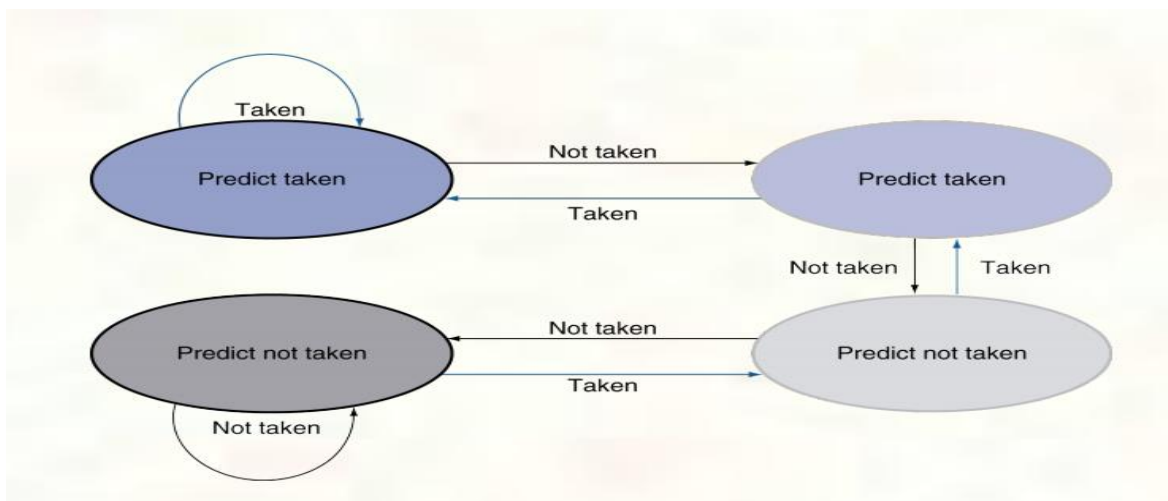


图 2.7 2 位预测状态机变迁图

表 2.11 BTB 表项地址划分

1 bit	2 bit	32 bit	32 bit
有效位	预测位	分支标记	预测地址

3 详细设计与实现

3.1 单周期 CPU 实现

3.1.1 主要功能部件实现

1) ALU 实现

Logism 实现:

根据前述设计，利用多选器对不同运算进行多选，然后将结果输出即可完成一个简易的 ALU，如图 3.1 ALU 结构图（1）、图 3.2 ALU 结构图（2）所示。

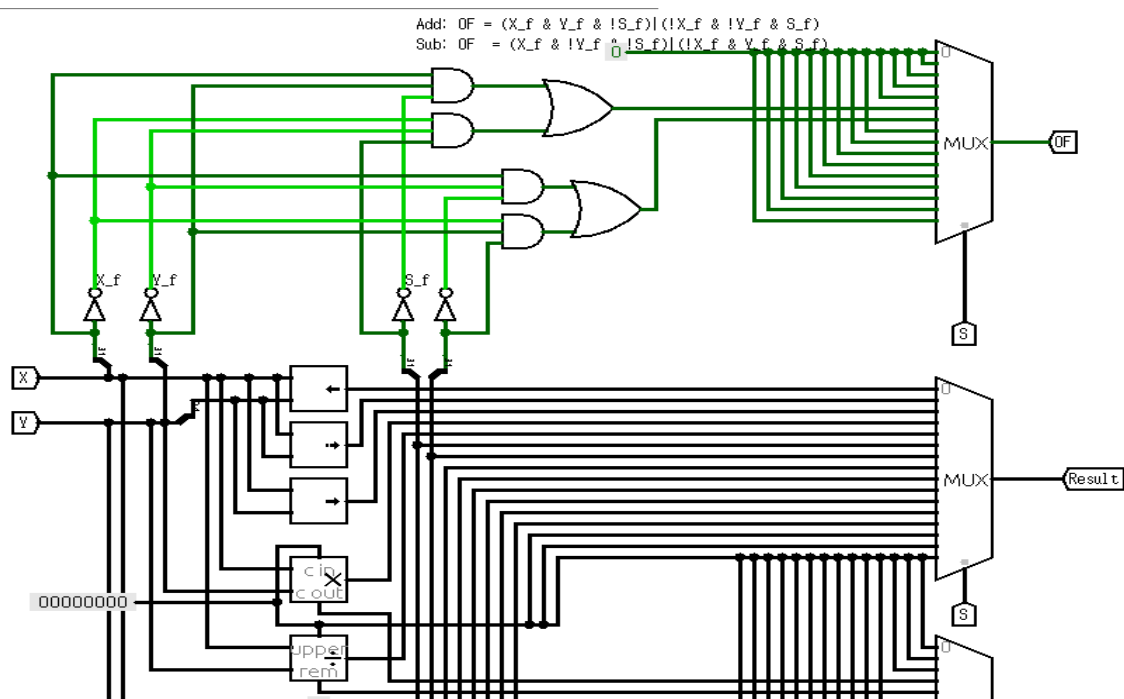


图 3.1 ALU 结构图（1）

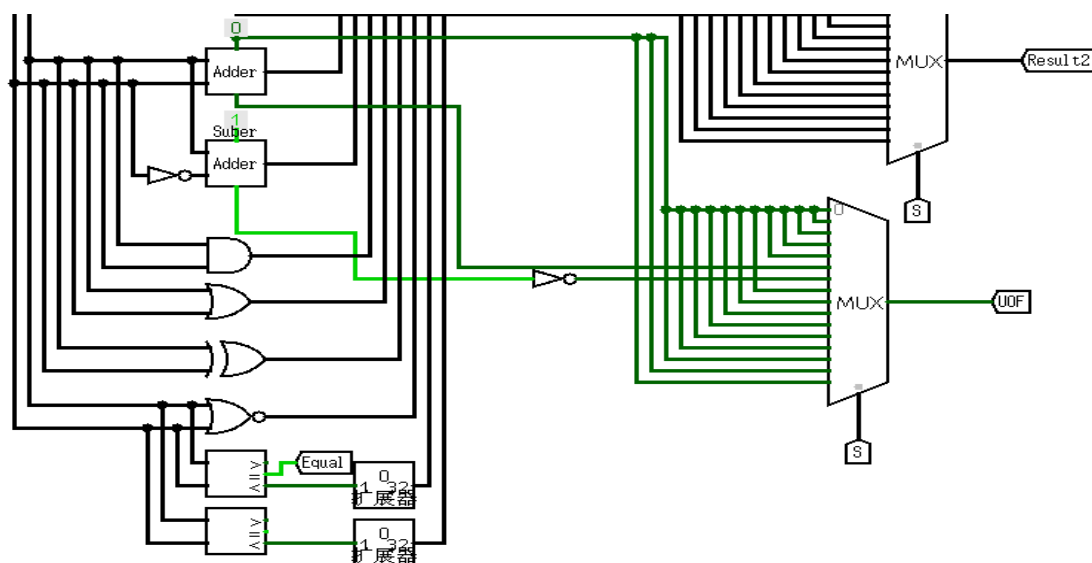


图 3.2 ALU 结构图 (2)

FPGA 实现:

在 Verilog 中, 所有变量默认都是 unsigned 型的, 但 >>> (算数右移)、< (有符号比较) 需要 signed 型的数, 故需利用 \$signed 任务将无符号变量转化为有符号变量再进行运算。

ALU 的 Verilog 代码如下:

```
assign signed_srcA = $signed(srcA);
assign signed_srcB = $signed(srcB);

always @ ( * ) begin
    case (aluop)
        4'd0: aluout <= srcA << srcB;
        4'd1: aluout <= signed_srcA >>> srcB;
        4'd2: aluout <= srcA >> srcB;
        4'd3: aluout <= srcA * srcB;
        4'd4: aluout <= srcA / srcB;
        4'd5: aluout <= srcA + srcB; // awesome tip
        4'd6: aluout <= srcA - srcB;
        4'd7: aluout <= srcA & srcB;
        4'd8: aluout <= srcA | srcB;
```

华中科技大学课程设计报告

```
4'd9: aluout <= srcA ^ srcB;
4'd10: aluout <= ~(srcA | srcB);
4'd11: aluout <= (signed_srcA < signed_srcB) ? 1 : 0;
4'd12: aluout <= (srcA < srcB) ? 1 : 0;
default: aluout <= 0;

endcase

end
```

2) Regfile 实现

Logism 实现:

根据前述设计, 将输入部分与输出部分利用复用器与解复用器进行分离, 实现对不同寄存器的写与读, 再结合寄存器阵列, 即可实现一个包含 32 个寄存器的通用寄存器组。写入使能选择电路如图 3.3 使能信号选择电路所示, 用于控制寄存器写; 输出部分电路如图 3.4 寄存器组输出部分所示, 用于控制寄存器读; 寄存器组完整电路如图 3.5 寄存器组完整电路所示。

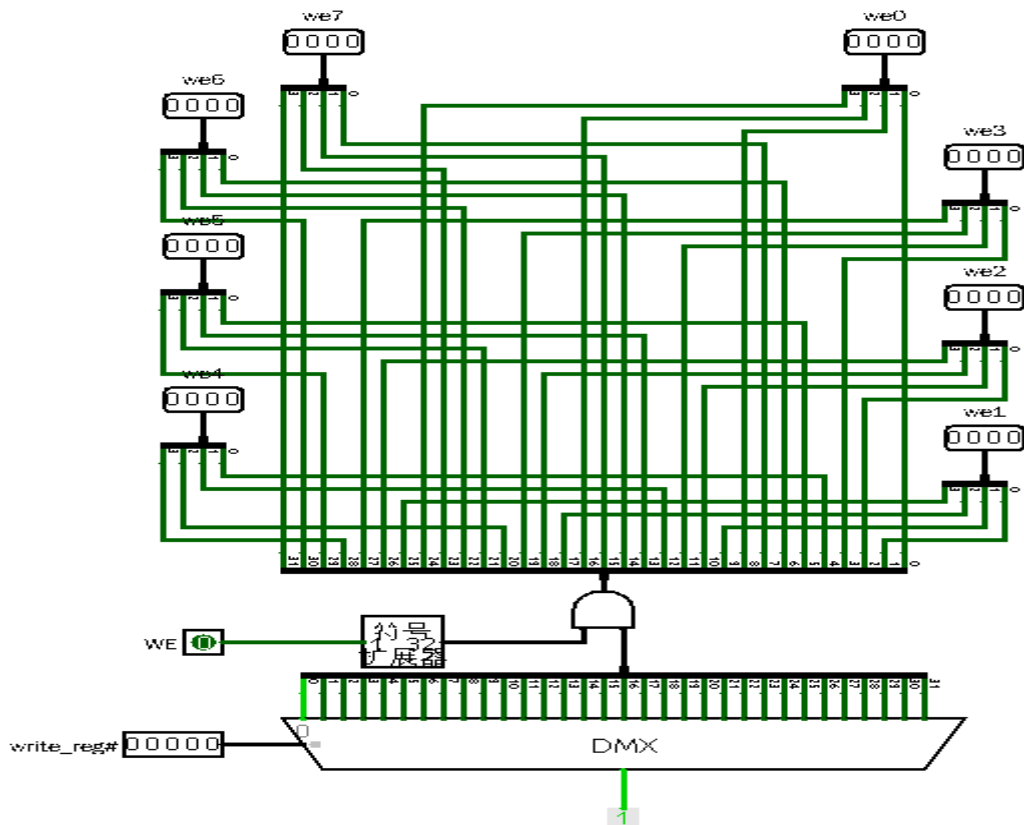


图 3.3 使能信号选择电路

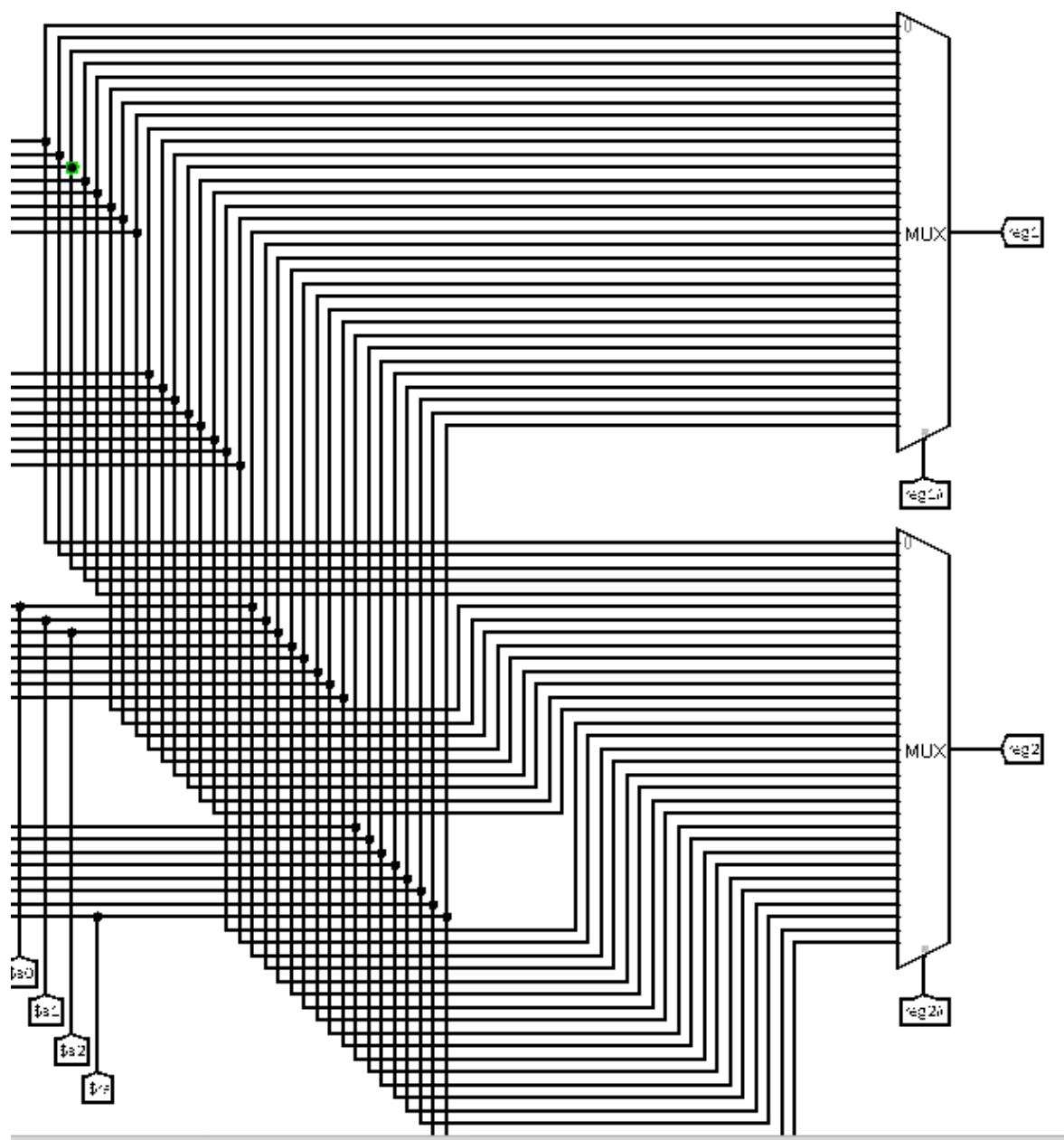


图 3.4 寄存器组输出部分

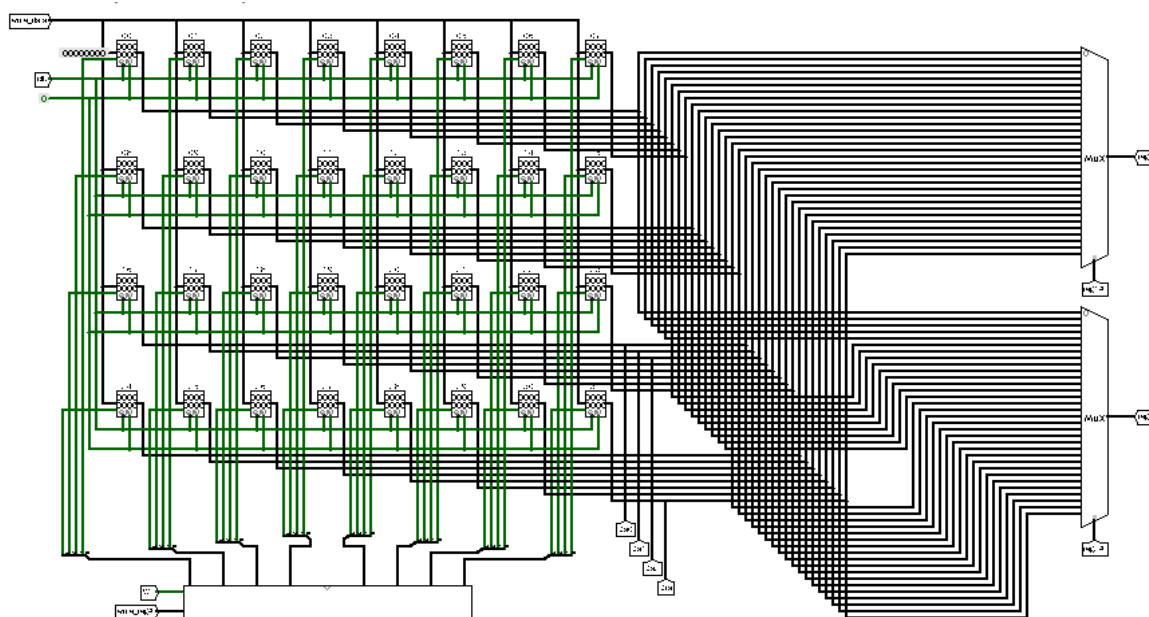


图 3.5 寄存器组完整电路

FPGA 实现：

Verilog 实现较为简单，直接利用 Reg 型的数组即可实现寄存器组。

其 Verilog 代码如下：

```

reg [DATA_WIDTH-1:0] regfile [0:31];    ///< three ported regfile contains 32
registers

always @(posedge clk) begin
    if (rst) begin
        for (i = 0; i < 31; i = i + 1)
            begin
                regfile[i] <= 0;
            end
    end else if (we && waddr != 0) begin
        regfile[waddr] <= wdata;
    end
end

assign regA = (raddrA != 0) ? regfile[raddrA] : 0;
assign regB = (raddrB != 0) ? regfile[raddrB] : 0;
    
```

3) 指令译码器

华中科技大学课程设计报告

Logism 实现:

根据 MIPS 3 种类型的指令,将指令的各个位段提取出来,即可完成指令译码器,输出的位段有 op (6 位)、rs (5 位)、rt (5 位)、rd (5 位)、sham (5 位)、funct (6 位)、imm16 (16 位)、imm26 (26 位)。如图 3.6 指令译码电路图所示

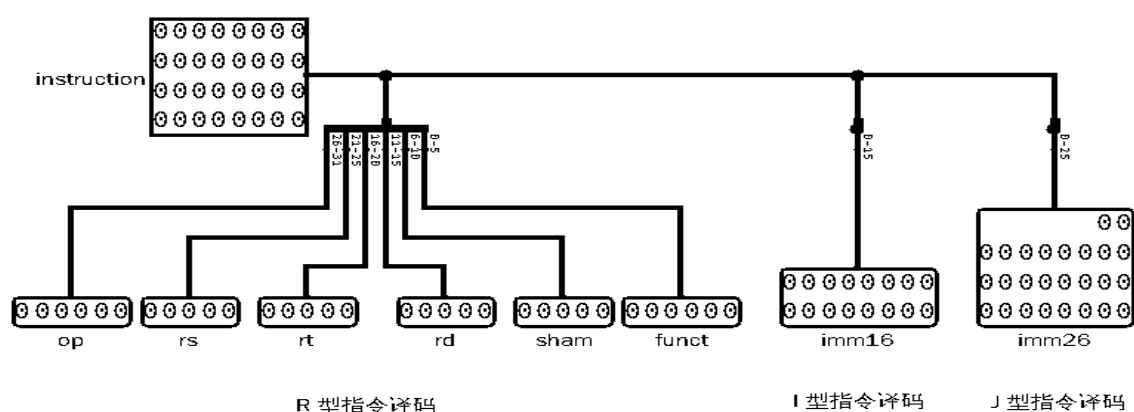


图 3.6 指令译码电路图

FPGA 实现:

利用位操作符即可分离信号, 其 Verilog 代码如下:

```
assign op      = instruction[`OP];
assign rs      = instruction[`RS];
assign rt      = instruction[`RT];
assign rd      = instruction[`RD];
assign sham    = instruction[`SHAM];
assign funct   = instruction[`FUNCT];
assign imm16   = instruction[`IMM16];
assign imm26   = instruction[`IMM26];
```

4) PC 更新逻辑

Logism 实现:

按照前述设计, 利用多个多选器, 根据当前译码信号 J/Jal/Jr/Beq/Bne/Bgtz 以及地址计算结果 Eq/Less, 生成当前跳转方式的表达式, 如表 2.2 PC 更新逻辑表所示。以相对跳转方式为例, 需要将 PC 按此种方式更新的逻辑表达式为

$$\text{Branch} = (\text{Beq} \ \&\& \ \text{Eq}) \parallel (\text{Bne} \ \&\& \ \sim\text{Eq}) \parallel (\text{Bgtz} \ \&\& \ \sim\text{Eq} \ \&\& \ \sim\text{Less}).$$

其具体实现如图 3.7 PC 更新逻辑单元所示。

FPGA 实现:

其 Verilog 代码如下:

5) ROM 和 RAM

华中科技大学课程设计报告

Logism 实现:

直接利用 Logisim 内建组件即可。

FPGA 实现:

无论是 ROM 还是 RAM, 实现思路都与 Regfile 类似, 利用 Reg 型的数组来模拟连续内存空间即可。

ROM 的 Verilog 代码如下:

```
reg [DATA_WIDTH-1:0] ROM [0:(2**BUS_WIDTH)-1];  
assign rdata = ROM[addr];
```

RAM 的 Verilog 代码如下:

```
reg [DATA_WIDTH-1:0] RAM [0:(2**BUS_WIDTH)-1];  
always @ (posedge clk) begin  
    if (we) begin  
        RAM[addr] <= wdata;  
    end  
end  
assign rdata = re ? RAM[addr] : {(DATA_WIDTH-1){1'b0}};
```

3.1.2 数据通路的实现

本次课程设计为采用工程化的方式实现数据通路, 而是逐步在原有数据通路的基础上整合新的数据通路。其实现过程如下所示:

常规 R 型指令数据通路: 按照前述设计, 此时只需 2 个控制信号, RegWe 控制寄存器组的写使能, ALUOp 控制 ALU 运算逻辑, 此类指令包括 add、addu、and、sub、or、nor、slt、sltu, 其数据通路如图 3.8 常规 R 型指令数据通路图所示。

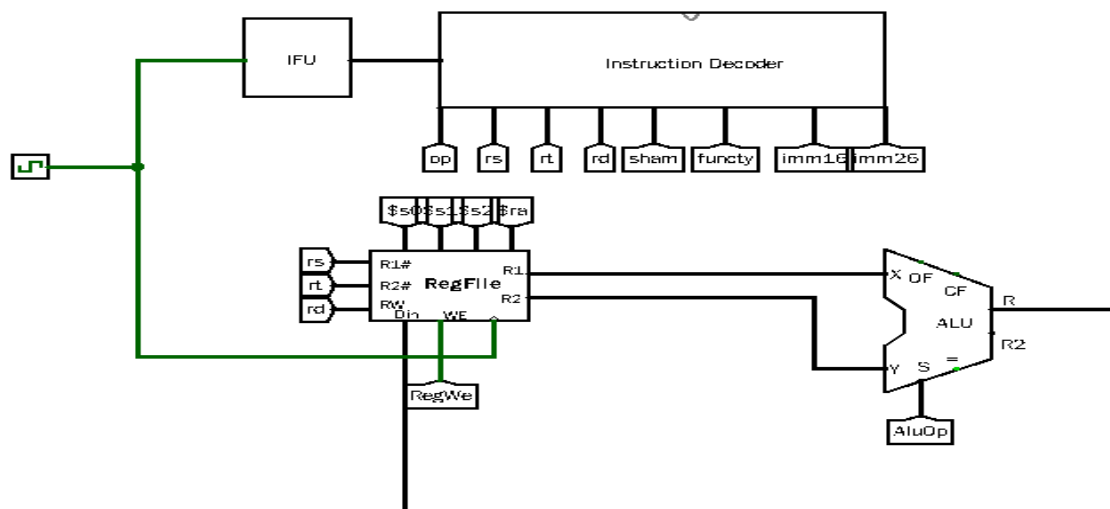


图 3.8 常规 R 型指令数据通路图

常规 I 型数据通路：按照前述设计，利用 3 个新的控制信号，RegDst 选择写入寄存器编号（rt 或 rd），ALUSrc 选择 Y 端输入（rt 或 imm16），ExtOp 选择立即数扩展方式（无符号或有符号扩展），此类指令包括 addi、addiu、andi、ori、slti。其具体通路如图 3.9 常规 I 型指令数据通路图所示。

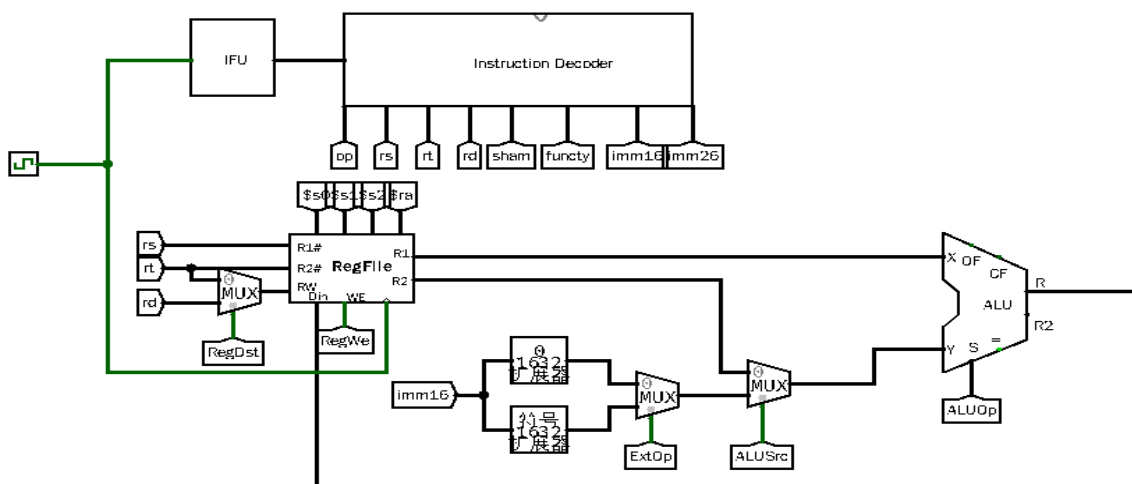


图 3.9 常规 I 型指令数据通路图

移位指令数据通路：按照前述设计，1 个新的信号 ALUSham，以改变 ALU 2 个输入端的值，其通路如图 3.10 移位指令数据通路图所示。

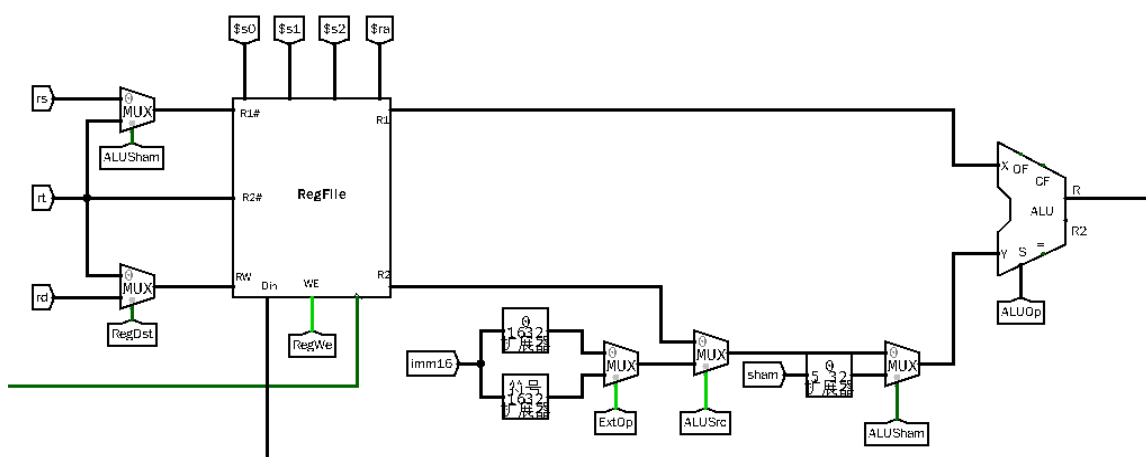


图 3.10 移位指令数据通路图

访存指令数据通路：按照前述设计，2 个新的控制信号，以控制主存的读写。对于 lw 指令，需要从主存读取数据至寄存器，增加新信号 RAMtoReg，以选择寄存器组的输入数据（ALU_Result 或 RAMDataOut）；对于 sw 指令，需要增加 1 个新信号 RAMWe，以控制对主存的写访问，并将（rt）接至主存加载数据端（RAMDataIn）。其具体通路如图 3.11 访存指令数据通路图所示。

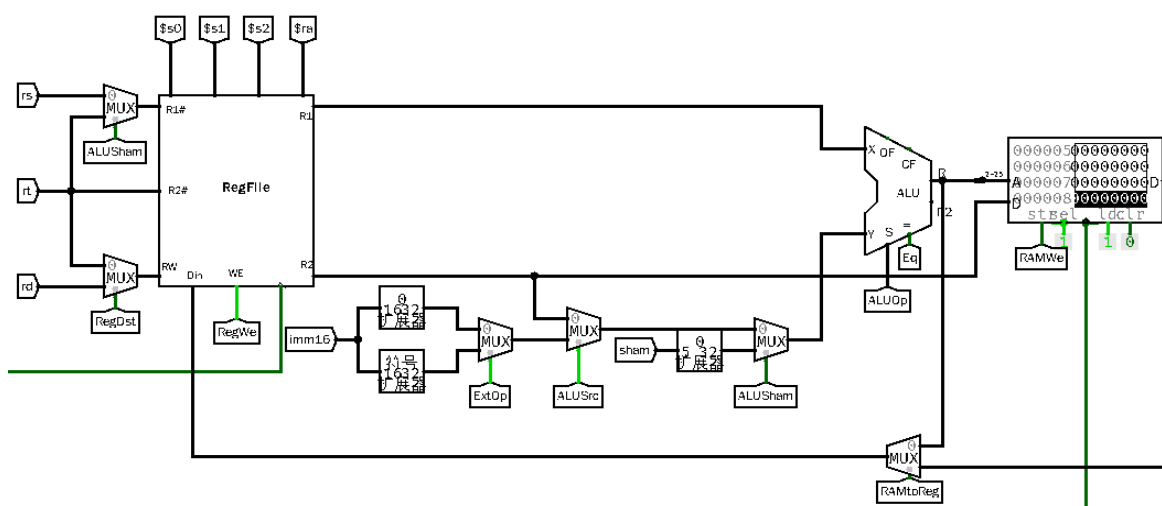


图 3.11 访存指令数据通路图

跳转指令：j/jal/jr/beq/bne/bgtz 指令需要按照图 3.7 PC 更新逻辑单元所示，修改 PC 更新逻辑，在合适的是否选择合适的新 PC 值更新 PC 寄存器。除此之外，jal 指令还需借助原数据通路将 rs 值取出加 4 后写入 \$a0 寄存器，此时 $ALU_X = (rs)$ ， $ALU_Y = (4)$ ， $ALU_OP = ADD$ ， $RW\# = 0x1f$ ，故需修改数据通路，如图 3.12 jal

华中科技大学课程设计报告

指令数据通路图所示。

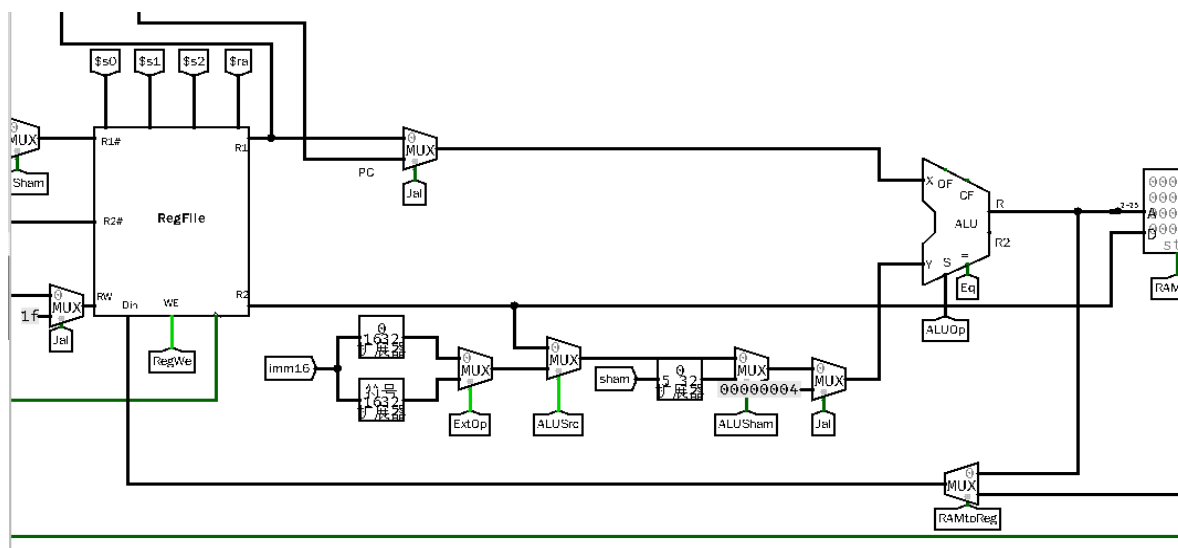


图 3.12 jal 指令数据通路图

关于 syscall 的实现，无需修改数据通路，只需将 \$v0 的值从寄存器组引出，并将其与 0xah 进行比较，再结合 Syscall 指令布尔信号，即可得到 Halt 停机信号与输出七段管信号。其具体实现如图 3.13 syscall 实现电路图所示。

利用 Halt 信号，对时钟源进行简单处理，即可实现 syscall 0xah 功能，停机功能具体实现如图 3.14 利用 syscall 模块生成的 halt 信号处理时钟源所示。

两张图中的计数器均为达到最大值则停止计数，实现锁存。

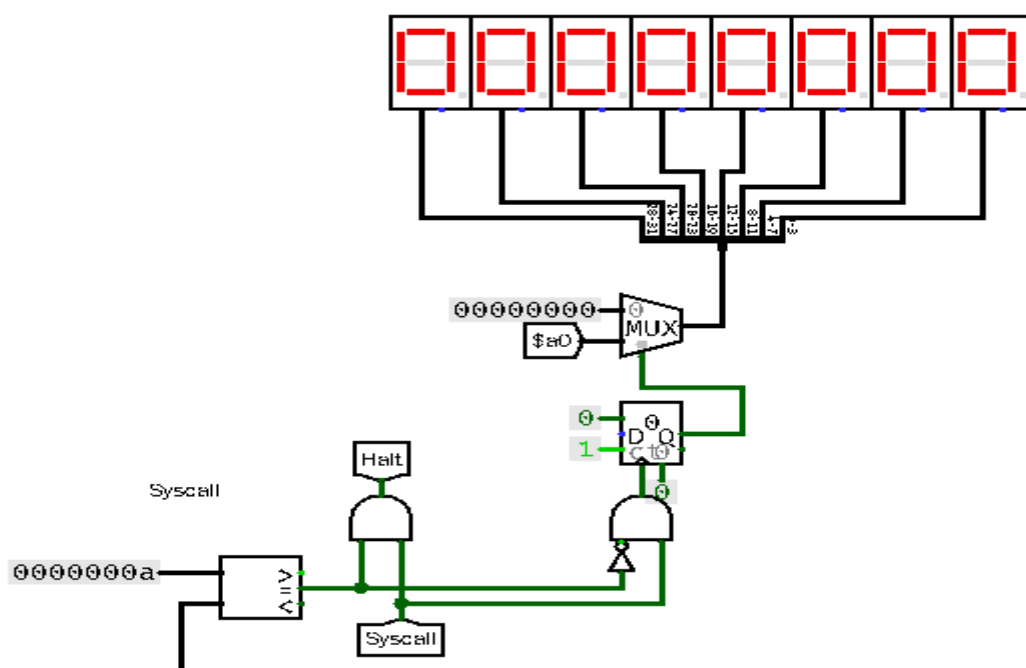


图 3.13 syscall 实现电路图

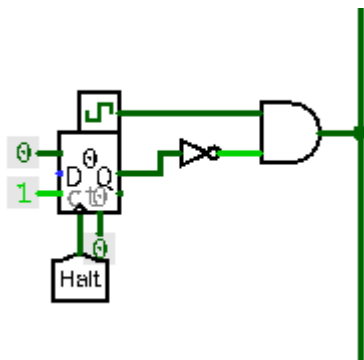


图 3.14 利用 syscall 模块生成的 halt 信号处理时钟源

扩展指令数据通路：divu 需要增加一个 LO 寄存器与一个 WriteToLO 信号（使能控制信号，将 ALU 除的结果在合适的时机写入 LO 寄存器），需将 ALUOp 译码成除法操作；mflo 需要修改写回寄存器的数据通路，额外增加一个控制信号，将写回寄存器的数据修改为 LO 寄存器的值；lb 指令只需将 RAM 读出的数据进行字节选择并进行符号扩展，得到新的一个 32 bit 的数据，再利用控制信号 RAMByte 在原来 RAM 数据输出和新数据间进行多选即可。divu 与 mflo 数据通路如图 3.15 divu 与 mflo 数据通路图所示，lb 数据通路如图 3.16 lb 数据通路图所示。

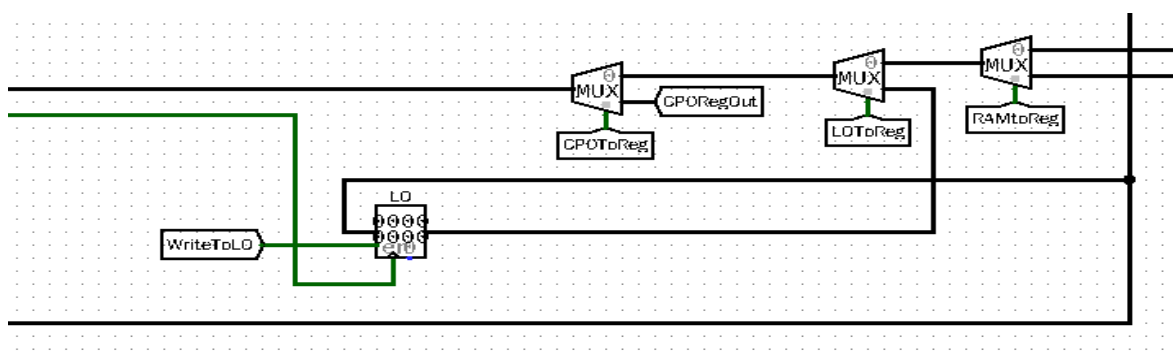


图 3.15 divu 与 mflo 数据通路图

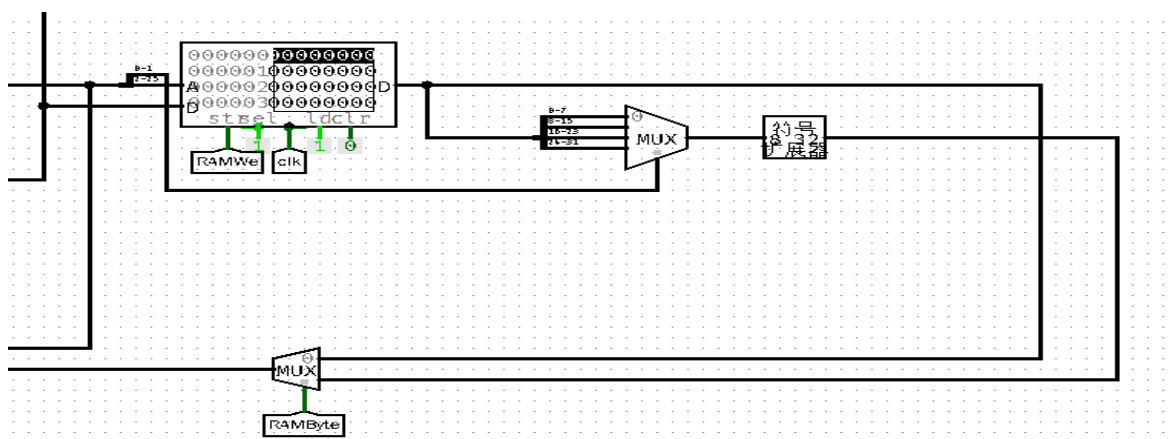


图 3.16 lb 数据通路图

3.1.3 控制器的实现

设计好数据通路，即可按照指令特性与数据通路走向，确定对于每一个指令的控制信号集。控制信号结果如表 2.6 控制信号表的框架（1）、表 2.7 控制信号表的框架（2）、表 2.8 控制信号表的框架（3）、表 2.9 控制信号表的框架（4）所示。利用多路或门，即可实现所有控制信号的生成，从而实现完整的控制单元，其电路如图 3.17 控制单元电路图（1）、图 3.18 控制单元电路图（2）所示。

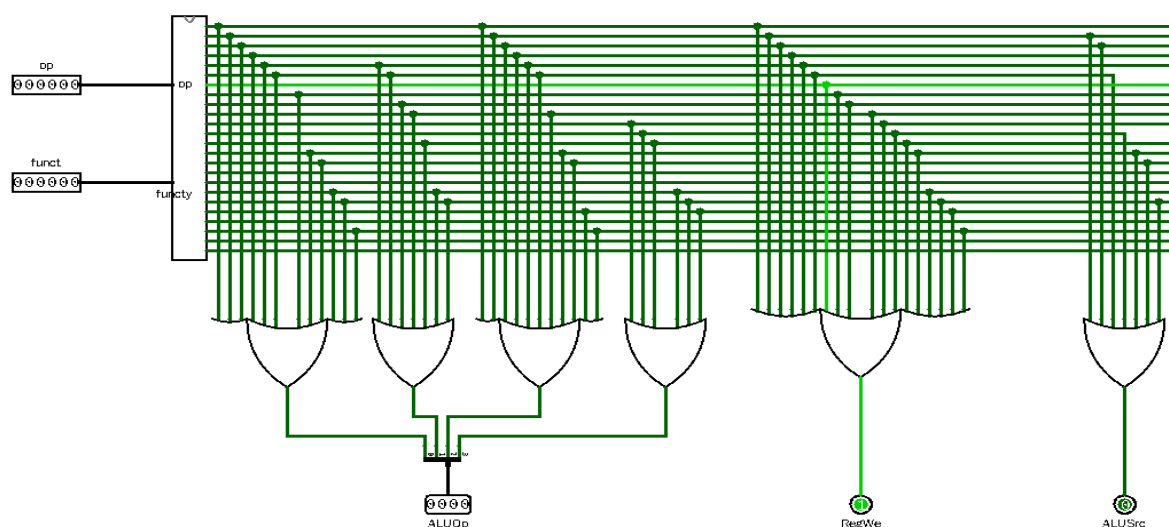


图 3.17 控制单元电路图（1）

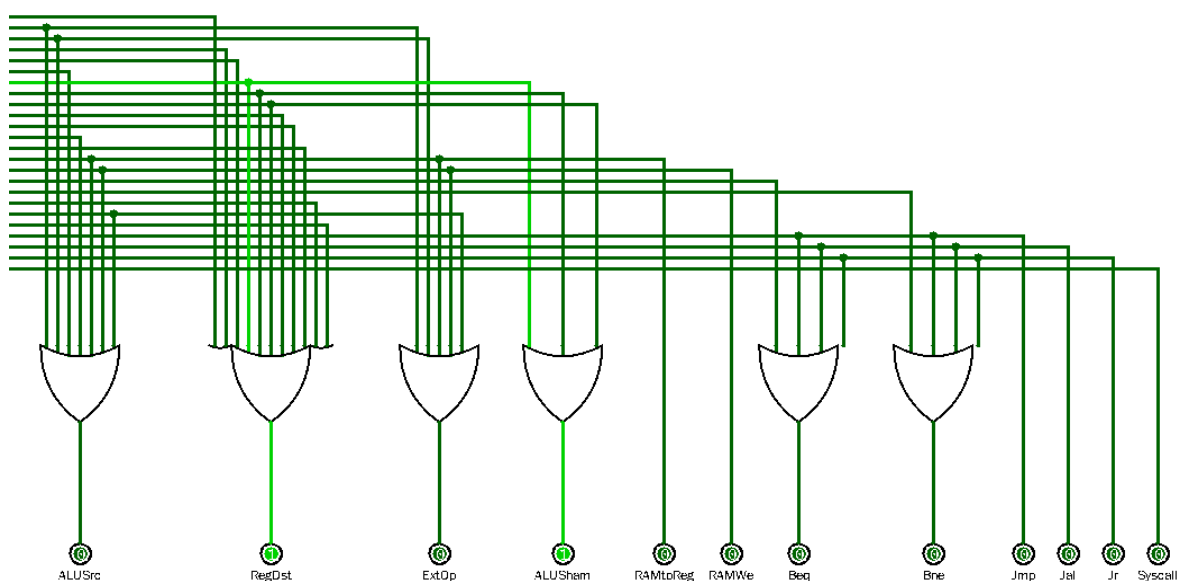


图 3.18 控制单元电路图（2）

根据在 Logism 实现中得到的各个一位控制信号的表达式，直接使用数据流建模，

华中科技大学课程设计报告

使用 assign 分的 Verilog 代码过于冗长，故只取对于控制信号 RegWe 的生成代码举例如下：

```
assign regwe = add || addi || addiu || addu || s_and || andi  
              || sll || sra || srl || sub || s_or || ori || s_nor || lw  
              || slt || slti || sltu || jal || mflo || lb;
```

以此类推，最终便可以实现整个主控制器中所有控制信号的生成。在 Vivado 中使用 Verilog 语言构成的主控制器原理图如图 3.19 主控制器原理图所示。

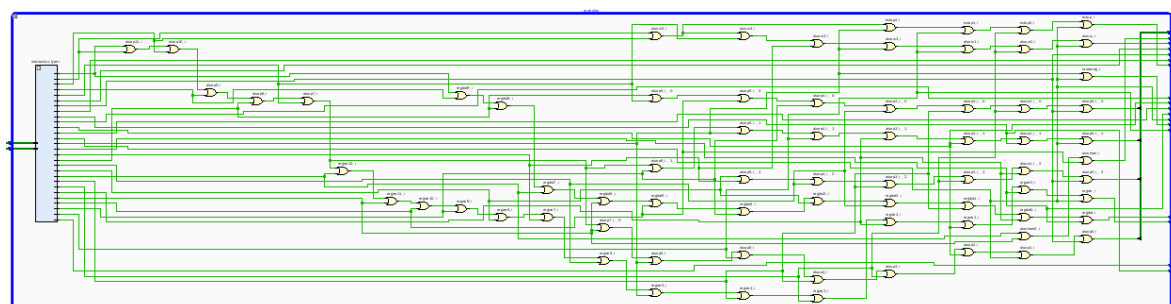


图 3.19 主控制器原理图

3.2 中断机制实现

3.2.1 中断识别与响应电路

按照如图 2.3 中断识别与响应电路所示原理图，在 Logisim 上实现等价电路，并在此基础上增加中断清零逻辑，即可完成中断识别与响应电路，如图 3.20 中断识别与响应电路所示。当有中断请求后，通过优先编码器的作用可以得到中断请求原始信号与中断号，中断号会送 Cause 寄存器。而作为输入的中断屏蔽字与中断使能位由 Status 寄存器确定。图中的复用器与解复用器实现对指定中断进行清除操作，当有中段被响应后，复用器与解复用器的循环会根据当前中断号，同步清零中断请求寄存器，其输出反过来清零右下角的寄存器，使得清零信号也被清零，达到稳态。

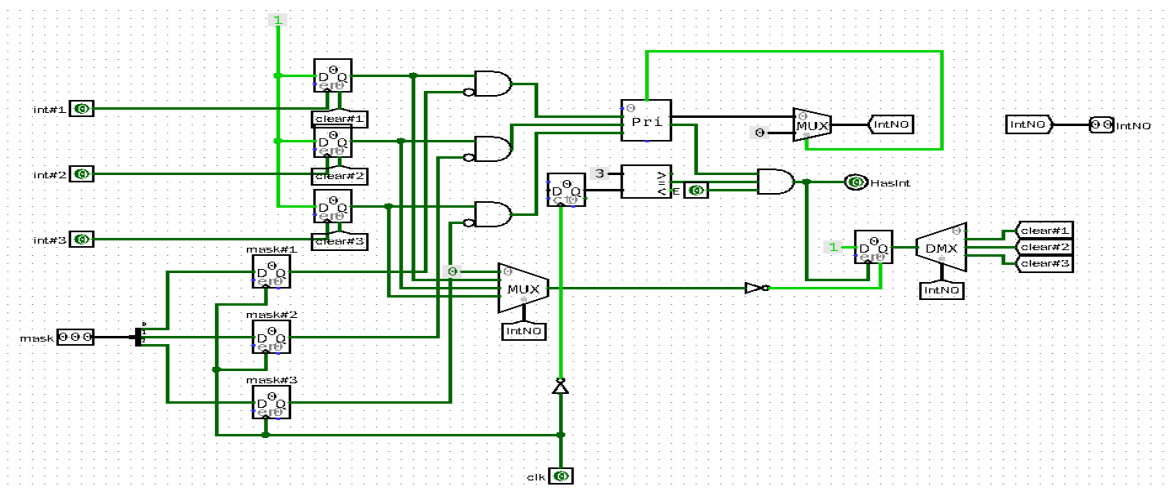


图 3.20 中断识别与响应电路

3.2.2 CP0 协处理器

按照前述设计,如图 2.4 简易 CP0 示意图所示,利用实现通用寄存器组的原理,实现一个简易的只含 3 个寄存器 (Status、EPC、Cause) 的 CP0 协处理器,如图 3.21 简易 CP0 协处理器实现所示。三个数据输入,分别为通用寄存器的值 Datain、旧的 PC 以及中断号,三个控制输入,HasInt 表示当前有中断发生,此时 EPC 的输入改为 PC 寄存器,用于保存断点,rd 用于控制输入与输出的目标寄存器,以实现 mfc0 与 mtc0 指令,clk 为时钟源。四个输出,其中 Status 的 0 位与 8-10 位分别作为中断使能位与中断屏蔽字接入中断识别与响应电路,而 OldPC 用于接入 PC 更新单元用于实现 Eret, CP0RegOut 接入经多选后通用寄存器组的数据输入端用于实现 mfc0。利用以上逻辑,便可实现 mfc0、mtc0 与 eret 三条指令,为中断实现软件支撑。

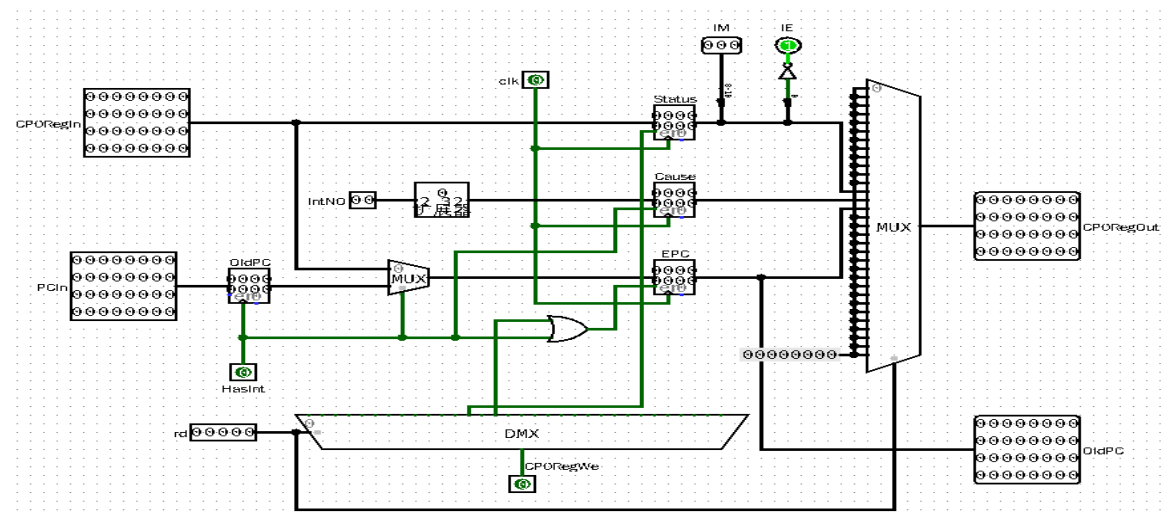


图 3.21 简易 CP0 协处理器实现

3.2.3 PC 更新

按照前述设计，修改 PC 更新单元，额外利用多选器实现中断跳转与中断返回，如图 3.22 修改 PC 更新逻辑所示。HasInt 表示当前有中断发生，PC 跳至 0x800，开始处理中断；Eret 信号表示当前有中断返回，PC 被修改为 EPC 中保存的断点，返回原用户程序继续执行。

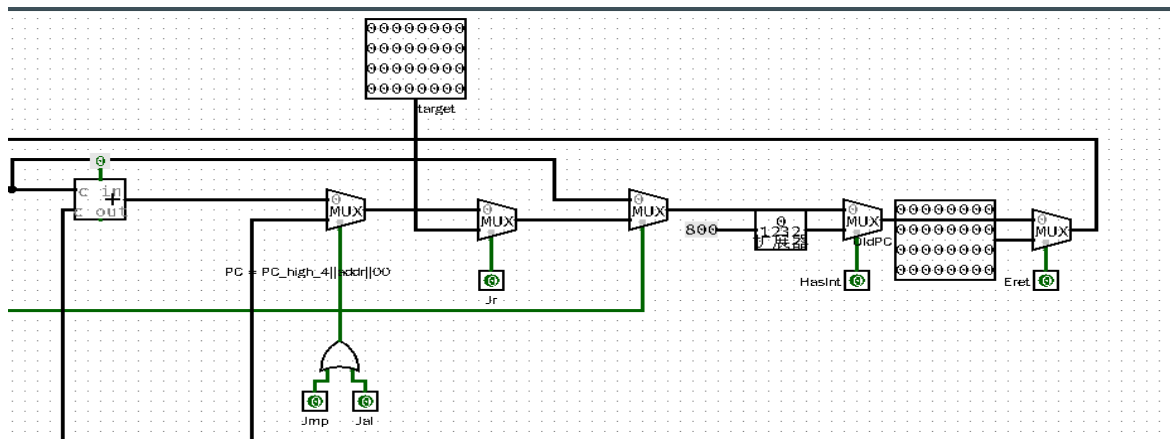


图 3.22 修改 PC 更新逻辑

完成以上 3 大单元，即可实现多级嵌套中断的硬件布局，其中中断识别与响应电路和 CP0 电路之间的耦合如图 3.23 完整中断电路所示。

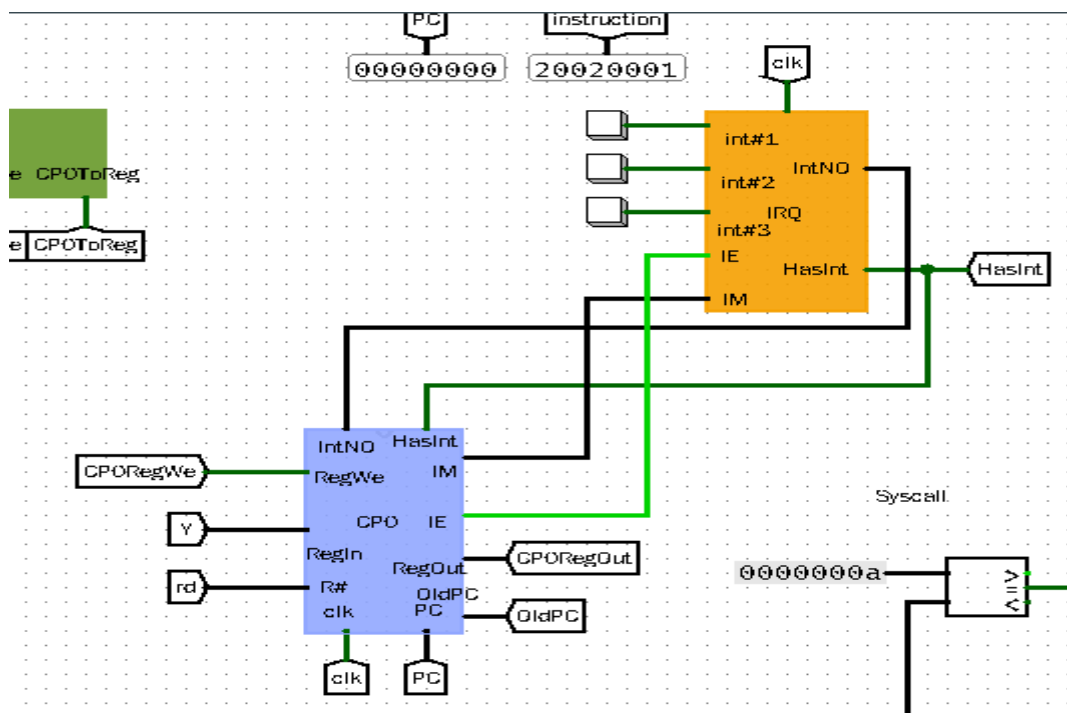


图 3.23 完整中断电路


```
(  
    input clk,input rst,input en,input [DATA_WIDTH-1:0] din,  
    output reg [DATA_WIDTH-1:0] dout  
);  
always @(posedge clk) begin  
    if (rst) begin  
        dout <= 0;        // reset  
    end else if (en) begin  
        dout <= din;      // update  
    end else begin  
        dout <= dout;     // hold  
    end  
end  
endmodule // register
```

3.3.2 理想流水线实现

实现完所有 4 个流水寄存器后，按照如图 2.6 理想流水线原理图所示，将整个数据通路串联起来，如图 3.25 理想流水线实现所示。

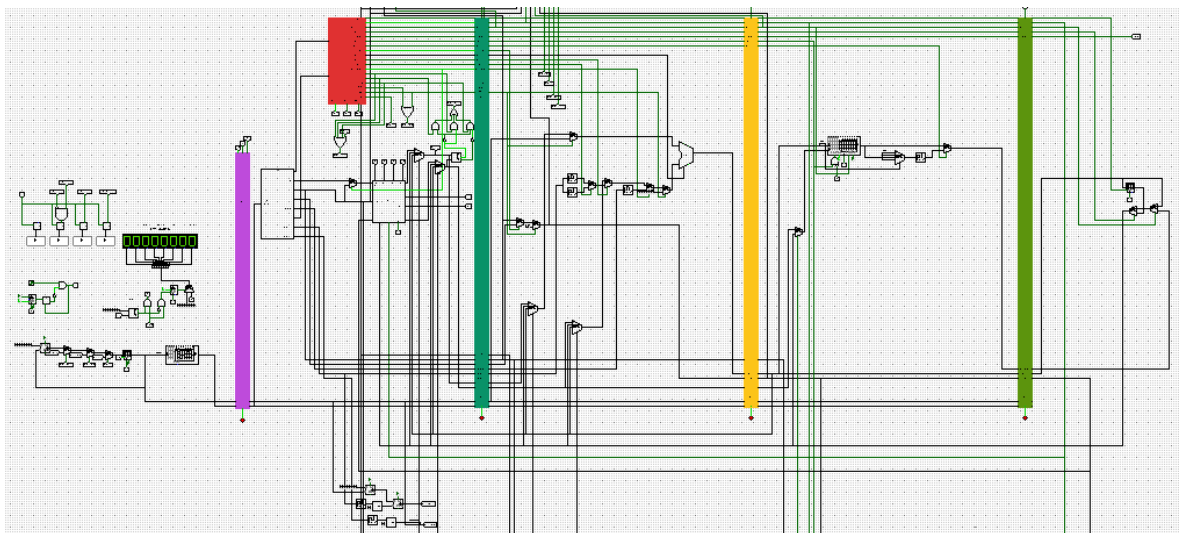


图 3.25 理想流水线实现

3.4 数据转发流水线实现

根据前述 2.4 设计，将 3 个阶段的转发信号逻辑表达式 (if else 分支判断) 转化成电路图，其中 ID 段转发信号如图 3.26 ID 转发信号图所示，EX 段转发信号如图 3.27 EX 段转发信号图所示，MEM 段转发信号如图 3.28 MEM 段转发信号图所示。修改原有数据通路，如图 3.29 ID 段转发通路、图 3.30 EX 段转发通路、图 3.31 MEM 段转发通路所示。

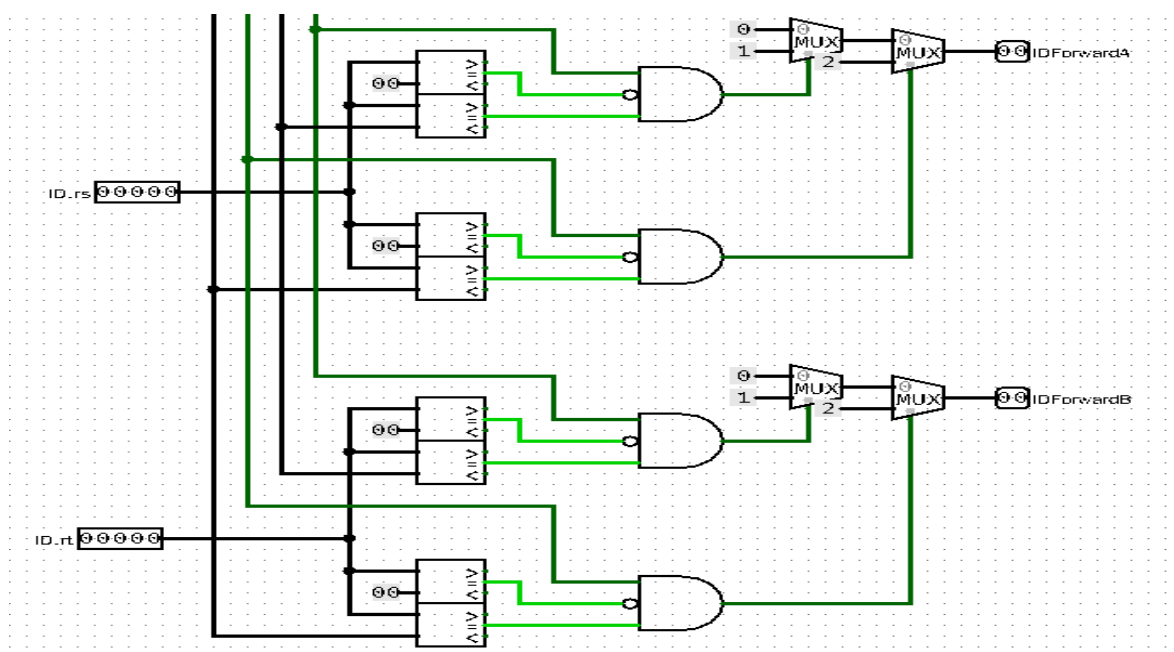


图 3.26 ID 转发信号图

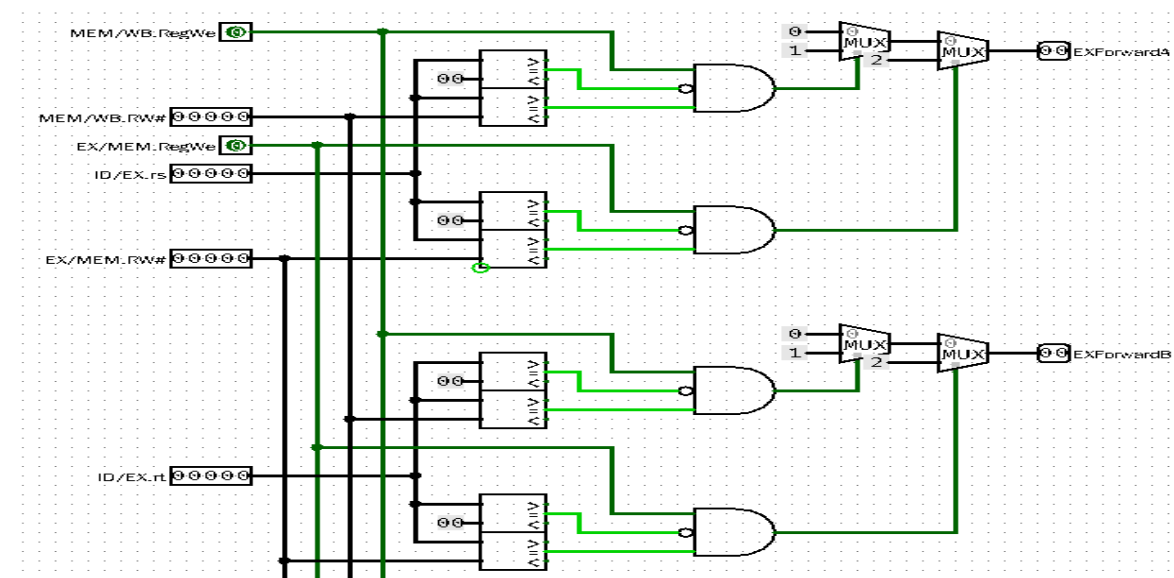


图 3.27 EX 段转发信号图

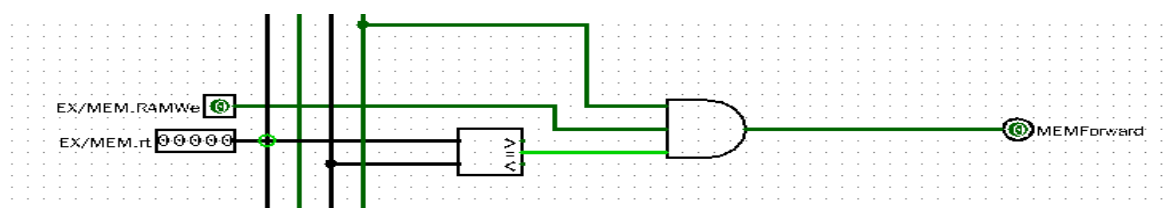


图 3.28 MEM 段转发信号图

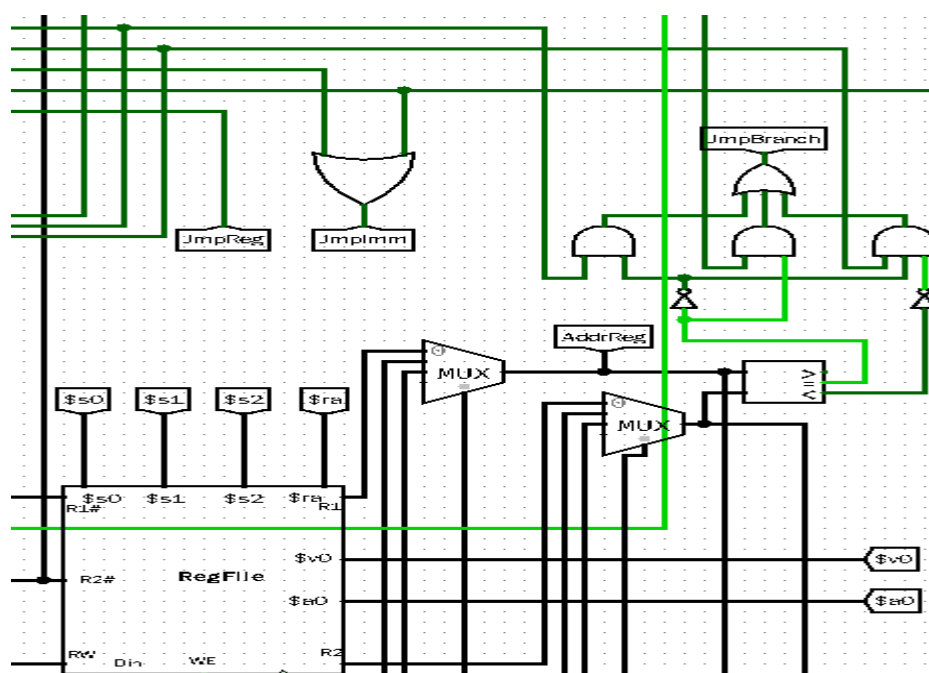


图 3.29 ID 段转发通路

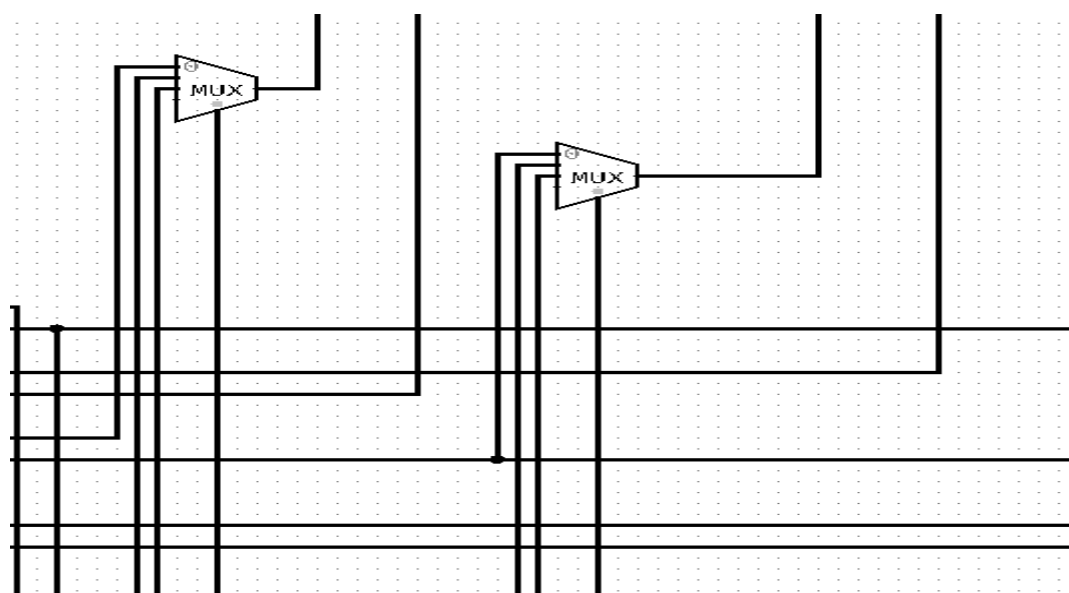


图 3.30 EX 段转发通路

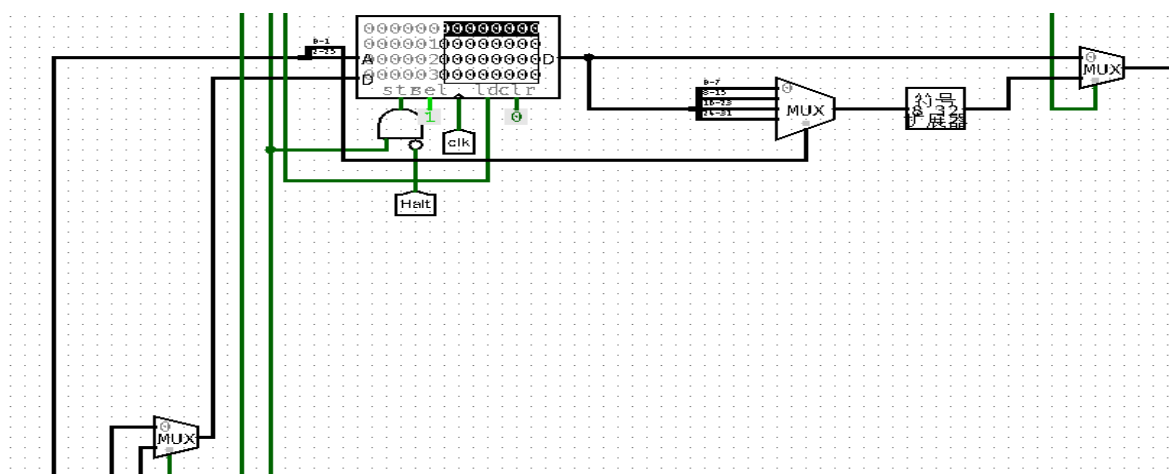


图 3.31 MEM 段转发通路

以 ID R1 的转发信号为例，其转化为 Verilog 代码如下所示：

```
always @( * ) begin
    if ((ID_rs != 0) && (ID_rs == MEM_RW) && MEM_regwe) begin
        ID_forwardA <= 2'b10;    // from MEM stage
    end else if ((ID_rs != 0) && (ID_rs == WB_RW) && WB_regwe) begin
        ID_forwardA <= 2'b01;    // from WB stage
    end else begin
        ID_forwardA <= 2'b00;    // no forwarding
    end
end
end
```

3.5 气泡式流水线实现

按照前述 2.5 设计，实现 load-use 冒险检测模块，产生 lwstall 信号，向流水线插气泡，如图 3.32 load-use 冒险检测模块所示；实现 branch 冒险检测模块，产生 branchstall 与 flushD 信号，向流水线插气泡与清空误取指令，如图 3.33 分支数据冒险检测模块、图 3.34 分支控制冒险检测模块所示。

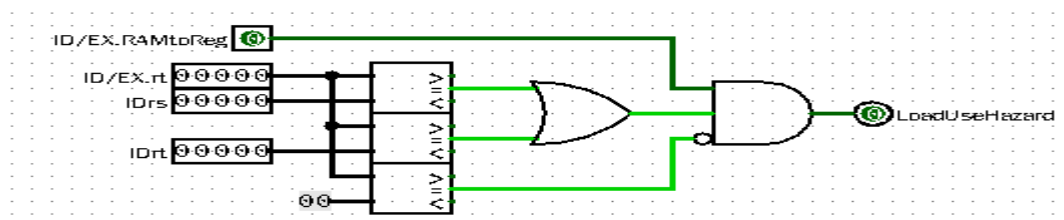


图 3.32 load-use 冒险检测模块

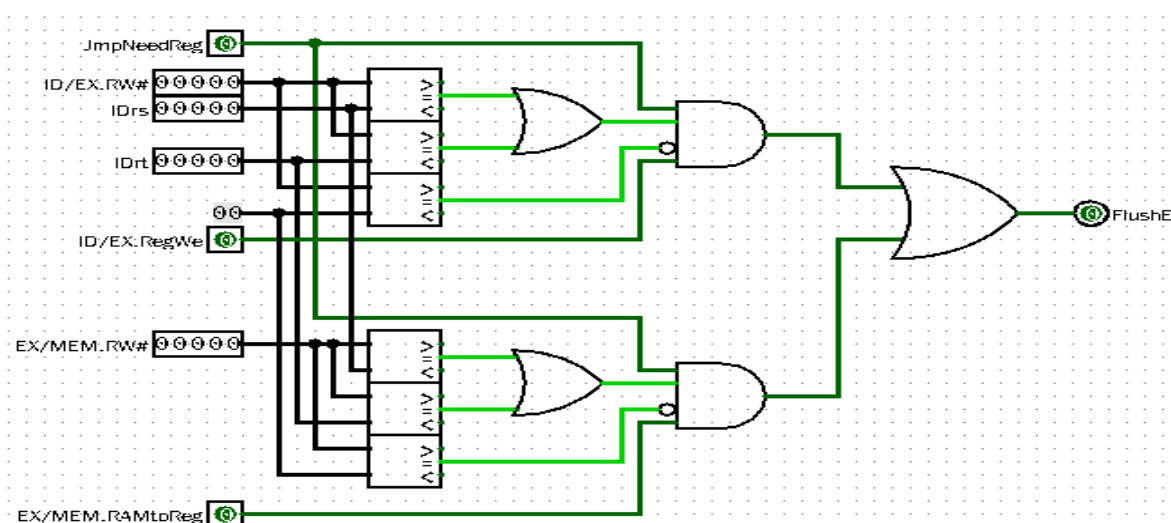


图 3.33 分支数据冒险检测模块

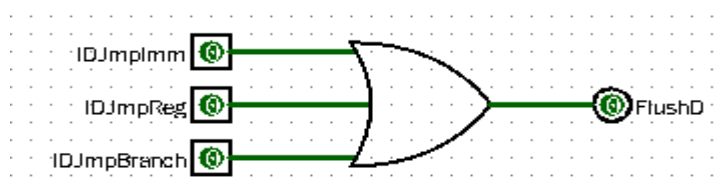


图 3.34 分支控制冒险检测模块

其转化为 Verilog 代码如下：

```
assign load_use_hazard = EX_rt != 0 && EX_ramtoReg && (EX_rt == ID_rs ||
EX_rt == ID_rt);

assign branch_flushD = ID_jmp_reg || ID_jmp_imm || ID_jmp_branch;

assign branch_flushE = (ID_jmp_need_reg && EX_regwe && EX_RW != 0
&& (EX_RW == ID_rs || EX_RW == ID_rt))
|| (ID_jmp_need_reg && MEM_ramtoReg && MEM_RW != 0 &&
(MEM_RW == ID_rs || MEM_RW == ID_rt));
```

3.6 动态分支预测机制实现

按照前述设计，此部分实现分为 2 大部分，一个为二位预测器的 Moore 型状态机实现，一个为全相联 BTB 表的实现（实际上为 Cache 的实现）。

按照如图 2.7.2 位预测状态机变迁图所示，实现一个 Moore 型状态机，其 Verilog 代码如下所示：

```
case (predict_bits[ID_access_line])
```

华中科技大学课程设计报告

```
`STRONGLY_TAKEN:
    case (misprediction)
        0: predict_bits[ID_access_line] <= `STRONGLY_TAKEN;
        1: predict_bits[ID_access_line] <= `WEAKLY_TAKEN;
    endcase
`WEAKLY_TAKEN:
    case (misprediction)
        0: predict_bits[ID_access_line] <= `STRONGLY_TAKEN;
        1: predict_bits[ID_access_line] <= `WEAKLY_NOT_TAKEN;
    endcase
`WEAKLY_NOT_TAKEN:
    case (misprediction)
        0: predict_bits[ID_access_line] <= `STRONGLY_NOT_TAKEN;
        1: predict_bits[ID_access_line] <= `WEAKLY_TAKEN;
    endcase
`STRONGLY_NOT_TAKEN:
    case (misprediction)
        0: predict_bits[ID_access_line] <= `STRONGLY_NOT_TAKEN;
        1: predict_bits[ID_access_line] <= `WEAKLY_NOT_TAKEN;
    endcase
endcase
```

第二步，完成相联比较器，利用多个比较电路，再利用一个 `always` 块进行编码，即可完成相联比较并得到目标行号，此模块的输出为 `hit` 与 `hit_line`，分别表示是否命中，以及命中时的行号其关键 Verilog 代码如下：

```
assign hit0 = valid0 && (src_tag == tag0);
assign hit1 = valid1 && (src_tag == tag1);
assign hit2 = valid2 && (src_tag == tag2);
assign hit3 = valid3 && (src_tag == tag3);
assign hit4 = valid4 && (src_tag == tag4);
```

华中科技大学课程设计报告

```
assign hit5 = valid5 && (src_tag == tag5);  
assign hit6 = valid6 && (src_tag == tag6);  
assign hit7 = valid7 && (src_tag == tag7);
```

利用相联比较器的比较结果，可以得到此次访问的实际 BTB 表行号， $\text{access_line} = \text{hit} ? \text{hit_line} : \text{lru_line}$ 。若此次访问命中，则从 BTB 中取出跳转地址；若不命中，则更新表项。除了在 IF 段访问时有可能修改表项，当 ID 段检测到预测错误时，也会更新表项，其关键 Verilog 代码如下：

```
if (~IF_hit && IF_branch) begin  
    valid[IF_access_line] <= 1;  
    predict_bits[IF_access_line] <= `WEAKLY_TAKEN;  
    branch_tags[IF_access_line] <= IF_branch_pc;  
    target_PCs[IF_access_line] <= IF_predict_addr;  
end  
if (ID_branch) begin  
    if (~ID_hit) begin  
        valid[ID_access_line] <= 1;  
        predict_bits[ID_access_line] <= `WEAKLY_TAKEN;  
        branch_tags[ID_access_line] <= ID_branch_pc;  
        target_PCs[ID_access_line] <= ID_branch_addr;  
    end  
end
```

得到最终的预测输出如下所示，即实现了 BTB 表的基本逻辑：

```
assign taken = IF_hit ? predict_bits[IF_access_line] : `WEAKLY_TAKEN;  
assign btb_branch_addr = IF_hit ? target_PCs[IF_access_line] : IF_predict_addr;
```

除此之外，还需实现预测错误修正逻辑，当预测出错时，需要进行 2 项工作，其一为清除误取指令，修改 flushD 的逻辑表达式即可，其二为修正跳转地址，修改 PC 更新逻辑即可。其关键 Verilog 代码如下所示：

```
assign branch_flushD = ID_jmp_reg || ID_misprediction;  
assign IF_mispredict_fix_addr = ID_jmp_branch ? ID_addr_branch : (ID_pc + 4);  
assign IF_pc_next = ID_jmp_reg ? ID_addr_reg
```

华中科技大学课程设计报告

```
: ID_misprediction ? IF_mispredict_fix_addr
```

```
: IF_predict_addr;
```

最后是关于 LRU (Least Recently Used) 算法的实现, 出现了一定问题, 基本思路为给每一行额外添加 3 bit 位, 用于表示当前行的引用计数, 每当 BTB 表被访问后, 被访问行计数清零, 其余行计数+1, 计数到 8 时便停止计数。下一次再次访问 BTB 表时, 若需要淘汰某一行时, 选择引用计数最大的一行进行替换 (表示其最久未被访问) 即可。遗憾地是, 进行硬件实现的时候, 出现了一些未知故障, 在验收期前并未将其调通。于是导致当前的 BTB 表只能有效地利用 1 行进行跳转历史的保存, 真是尴尬。LRU 算法的实现详见 lru_counter.v 模块。

4 实验过程与调试

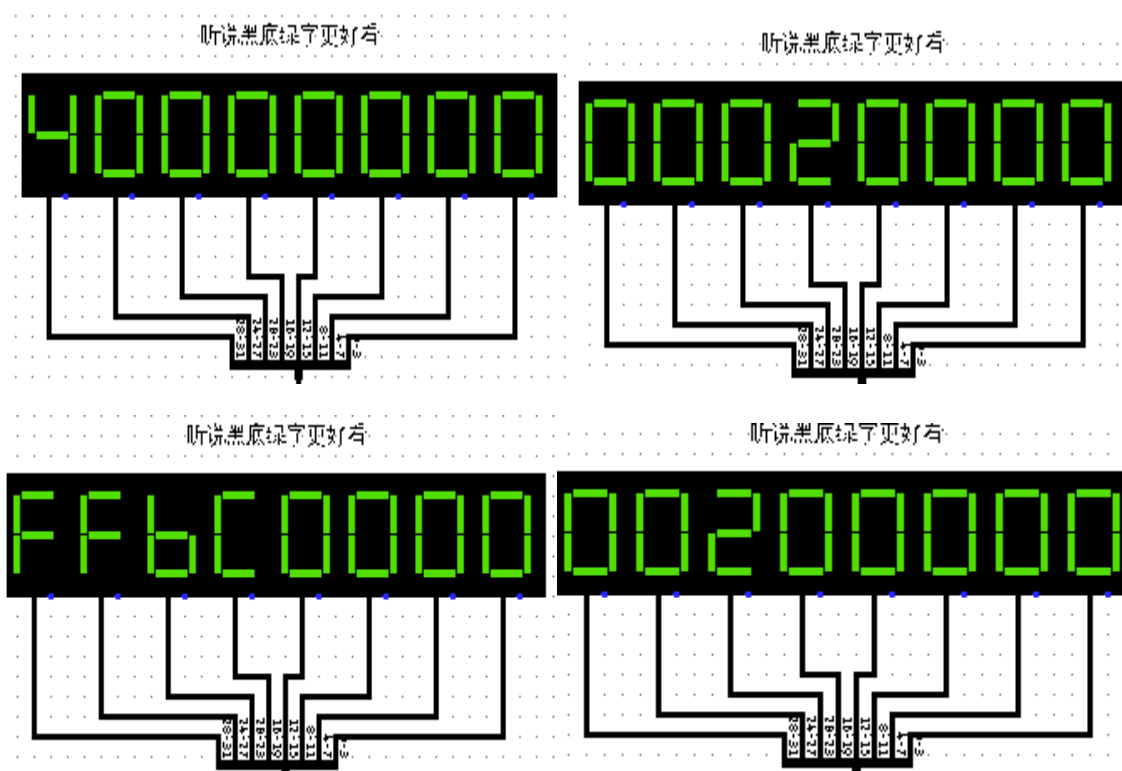
4.1 测试用例和功能测试

4.1.1 测试用例 1：扩展指令测试

自行编写了简短的测试代码，测试了 4 条扩展指令。通过比较 Mars 与 Logisim 的运行过程（PC 与 IR 变化）以及最终结果（Regfile 与 RAM 变化），发现扩展指令实现正确无误。受报告篇幅限制，具体测试代码见 `extend.asm`。

4.1.2 测试用例 2：中断测试

分别按下 2、1、3 号中断按钮，可以观察到程序执行过程为 用户程序->2 号跑马灯->3 号跑马灯->2 号跑马灯->用户程序->1 号跑马灯，其过程如图 4.1 中断执行过程图所示。



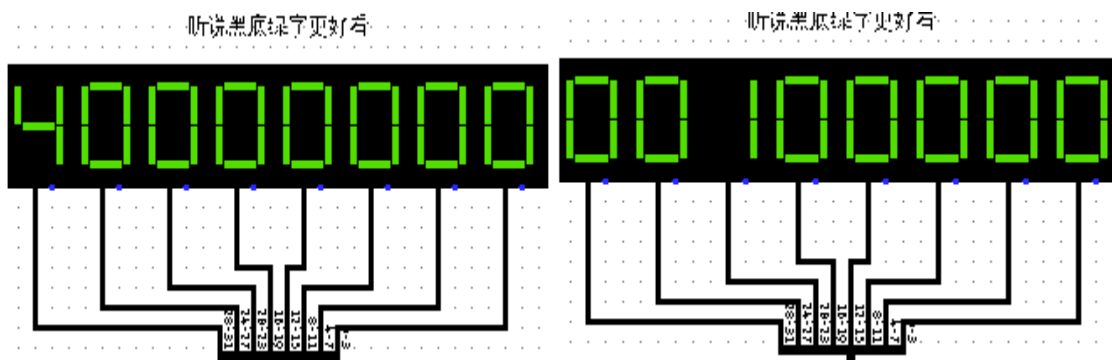


图 4.1 中断执行过程图

4.1.3 测试用例 3：5 段流水线测试

将 benchmark.hex 加载至指令 ROM，观察周期数，利用计算验证可以得到

$$2306 = 1546 \text{ (单周期)} + 4 + 38 \text{ (j/jal/jr)} + 276 \text{ (成功分支)} + 120 \text{ (load use 气泡)} + 322 \text{ (分支数据气泡)},$$

如图 4.2 Benchmark 流水线周期数所示。

利用 iverilog 在 Linux 下进行仿真，可以得到周期数亦为 2306 (0x902)，FPGA 实现也正确无误，如图 4.3 FPGA 仿真结果所示。

关于流水线的更多测试用例，可参见周期统计的 Excel 表格。

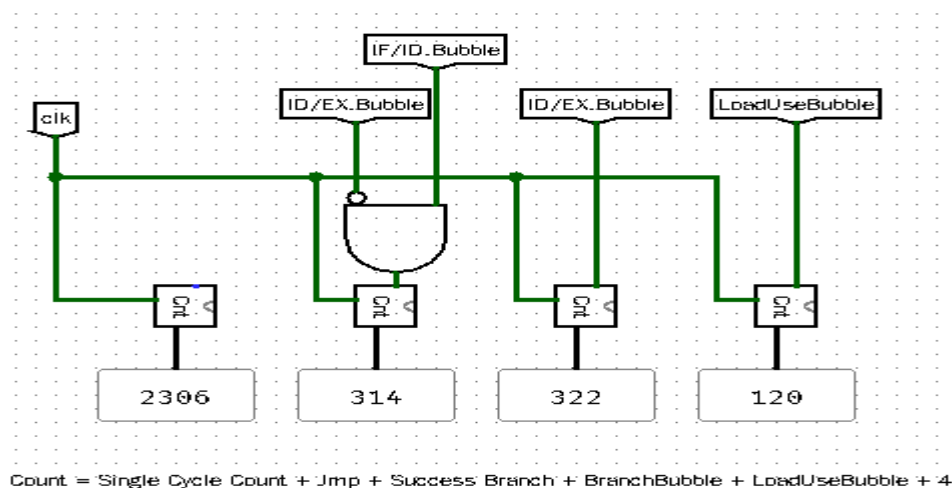


图 4.2 Benchmark 流水线周期数

```
0, 1, 00000038, 00000900
0, 1, 00000038, 00000901
0, 1, 00000038, 00000902
# vvp out/mips_tb -lxt2
sabertazimi@avalon: verilog 02/26/17-18:01:32
```

图 4.3 FPGA 仿真结果

华中科技大学课程设计报告

4.1.4 测试用例 4：分支预测测试用例

将自己利用扩展指令编写的跑马灯加入 benchmark.asm，进行仿真，得到周期数统计如图 4.5 benchmarkpp.asm 周期仿真图所示，从左至右的数值分别为 led 数据、总周期数、预测成功周期数、预测失败周期数、流水线清空次数（预测失败周期数+jr 命令条数）、load-use 冒险数、branch 数据冒险数。通过 Mars 可以得到单周期数为,如图 4.4 benchmarkpp.asm 单周期数所示，可以计算得出：

$2943 = 2418$ （单周期数）+4+79（流水线清零数）+120（load-use 冒险数）+322（branch 数据冒险数）。可得分支预测实现正确无误，预测成功率在 85%以上。

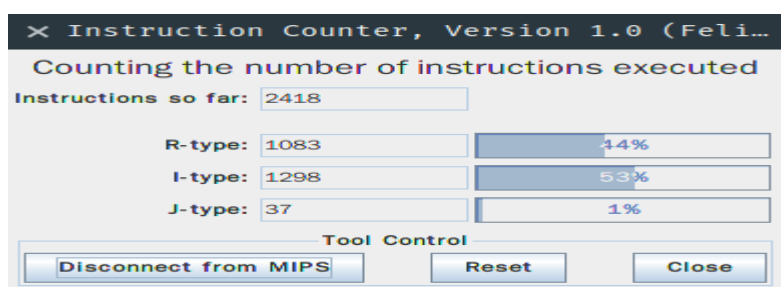


图 4.4 benchmarkpp.asm 单周期数

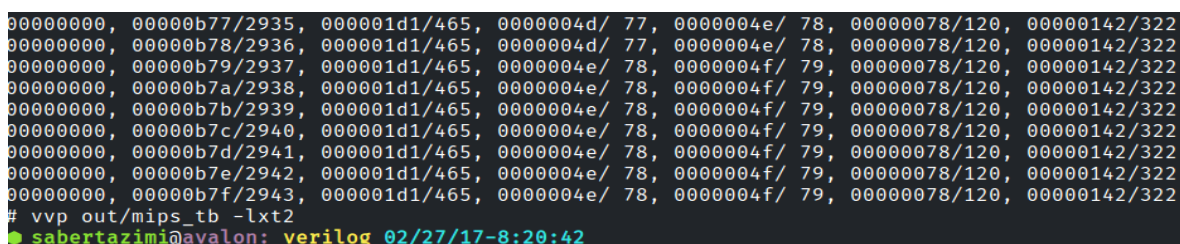


图 4.5 benchmarkpp.asm 周期仿真图

4.2 性能分析

此次实验将分支指令的大部分逻辑放在 ID 段，理论上每次跳转指令的误取深度只有 1。在理想情况下，运行 benchmark.asm 的代码只需 1984 个时钟周期。但由于分支指令（beq/bne/bgtz）产生的数据冒险，无法利用转发逻辑解决（由于 ALU 与 RAM 的自身延迟，利用转发十分危险），故需要使用插入气泡的方式解决数据冒险。当代码中的每一条 beq/bne/bgtz 都存在数据冒险时，这将会对流水线的效率造成重大影响。而 benchmark.asm 的每一条分支指令都存在此种冒险（322 个气泡），最后导致运行完 benchmark.asm 需要 2306 个时钟周期。

仅考虑 benchmark.asm 的情况下，若将跳转逻辑置于 EX 段，则其数据冒险可利

华中科技大学课程设计报告

用已有的 EX 段的转发逻辑解决，无需插入气泡。尽管其误取深度为 2，但最后运行 benchmark.asm 仅虚 2298 个时钟周期。可以看到，对于不同的代码，理论上效率更高的架构有可能效率更低。

但总体来说，跳转逻辑置于 ID 段的流水线的效率显然是要高于跳转逻辑较深的流水线。最终，实现后的电路图总体性能如图 4.6 FPGA 实现性能图所示。

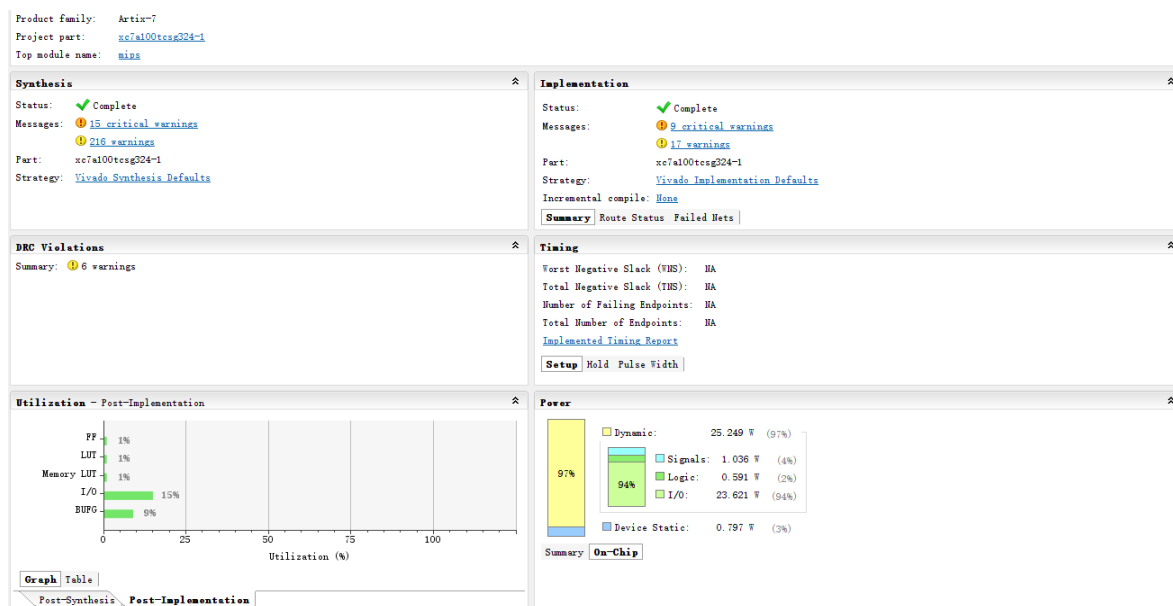


图 4.6 FPGA 实现性能图

4.3 主要故障与调试

4.3.1 中断请求响应电路设计缺陷

Logisim 单周期 CPU：中断请求响应电路存在设计缺陷。

故障现象：通过学习后发现，原有的响应电路（如图 4.7 简易中断请求响应电路所示）并未将中断请求进行锁存这一设计，是不符合实际要求的。若时钟足够慢，则输入的中断源必须保持足够长的时间，才可以成功地引起中断，从而使 CPU 转入中断处理程序，这一设计显然是不合理的。

原因分析：由于 3 个中断请求寄存器的时钟源为 clk 使得其不能及时锁存中断源，故应修改中断请求寄存器的时钟源。

解决方案：如图 4.8 同步清 0 锁存中断源的中断请求响应电路所示，将中断请求寄存器的时钟源改为中断源，即可实现中断请求的立即锁存，再利用 clk 进行组合逻辑，得到 HasInt 信号。这样一来，既可以不漏漏任何中断源，又可以实现中断请

求信号的产生同步于时钟。

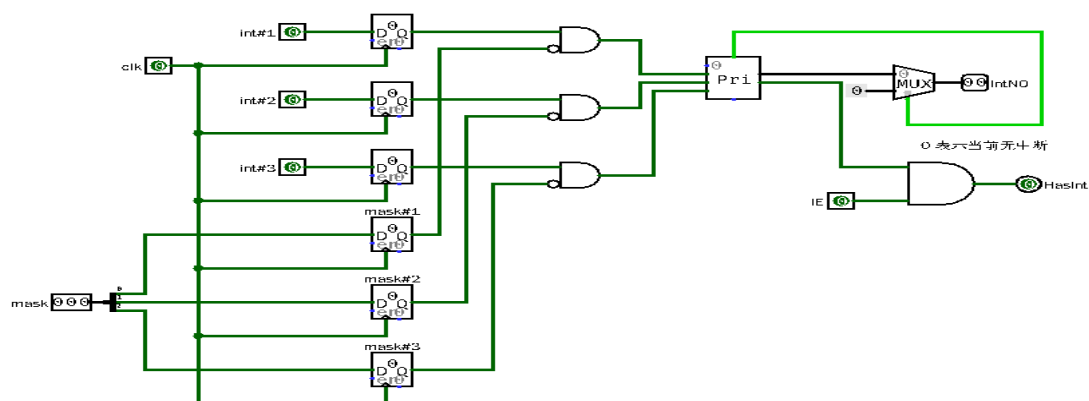


图 4.7 简易中断请求响应电路

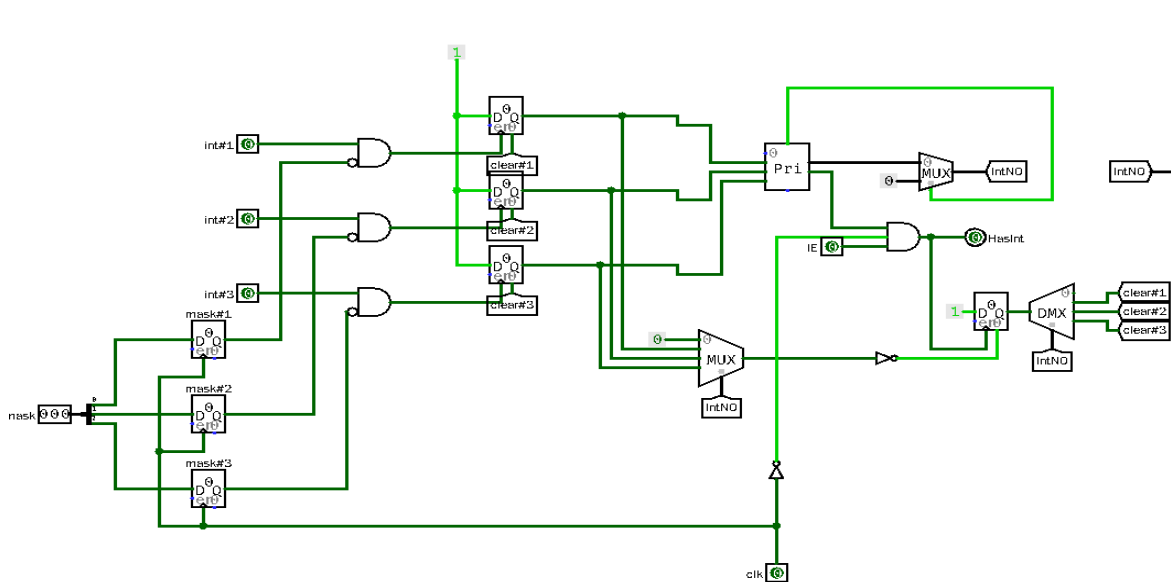


图 4.8 同步清 0 锁存中断源的中断请求响应电路

4.3.2 WB-ID 冒险

流水线 CPU：数据冒险未解决。

故障现象：执行

```
addu $s0,$zero,$zero
```

```
addi $s2,$zero,12
```

```
addiu $s6,$0,3
```

addiu \$s0, \$s0, 1 时最后一条指令会产生数据冒险。

原因分析：根据《计算机组成与设计》一书，寄存器组实现一般为前半周期写，

华中科技大学课程设计报告

后半周期读，如此一来上述指令会读取新的值，不会产生数据冒险。但由于此次寄存器的实现并未严格按照此标准，导致最终上述指令读取的 `s0` 为旧值，产生数据冒险。

解决方案：在 ID 段也进行数据转发即可。

4.3.3 其余故障

受篇幅限制，仅列举以上 2 个在设计上所遇到的错误，更多错误参见图 4.9 Github issues 列表、图 4.10 issues 相关提交一览。

<input type="checkbox"/>		Bug: waterfall_lamp test fail for btb module bug	
		#22 by sabertazimi was closed 3 days ago	
<input type="checkbox"/>		Bug: invalid instructions when syscall (halt) got invoked bug	
		#21 by sabertazimi was closed 7 days ago	
<input type="checkbox"/>		Feat: implement pipeline MIPS CPU in verilog on FPGA platform feature	
		#20 by sabertazimi was closed 5 days ago	
<input type="checkbox"/>		Bug: clear when stall bug	
		#19 by sabertazimi was closed 8 days ago	
<input type="checkbox"/>		Bug: ID WB data hazard bug	
		#18 by sabertazimi was closed 8 days ago	
<input type="checkbox"/>		Bug: syscall data hazard bug	
		#17 by sabertazimi was closed 5 days ago	
<input type="checkbox"/>		Bug: PC+4 load error bug	
		#16 by sabertazimi was closed 9 days ago	
<input type="checkbox"/>		Bug: EPC got dropped bug	
		#15 by sabertazimi was closed 9 days ago	
<input type="checkbox"/>		Bug: 未将中断源进行锁存 bug	
		#14 by sabertazimi was closed 11 days ago	
<input type="checkbox"/>		Bug: 发生中断时，未将 PC 进行暂停 bug	
		#13 by sabertazimi was closed 11 days ago	
<input type="checkbox"/>		Bug: change jr to eret bug	
		#12 by sabertazimi was closed 11 days ago	
<input type="checkbox"/>		Bug: int\$1 and int\$2 get wrong, int\$3 get right bug	
		#11 by sabertazimi was closed 12 days ago	
<input type="checkbox"/>		Bug: can't get right PC to EPC bug	
		#10 by sabertazimi was closed 12 days ago	
<input type="checkbox"/>		Bug: lb - error btye load bug	
		#9 by sabertazimi was closed 12 days ago	

图 4.9 Github issues 列表

```
● sabertazimi@avalon: verilog 02/26/17-18:06:40
→ git log --oneline | grep fix
b2f9642 fix(arch-branch prediction-IF_taken): error IF_taken
ca4d4b9 fix(arch-branch prediction-error ID new addr): send correct ID addr
cec440d fix(arch-branch prediction-enable): add global enable switch
9253ce3 fix(arch-branch prediction-FPGA): rectify error in addr_calculation
a96a71c fix(arch-pipeline(FPGA)-halt): rectify halt diable error
af25d61 fix(arch-pipeline(FPGA)-WB stage): rectify WB_result error
8fbef47 fix(arch-pipeline(FPGA)-syntax): rectify all syntax error
1befd4f fix(arch-pipeline-syscall): rectify excution of redundant instructions
b05fa4c fix(arch-pipeline hazard-IF/ID): rectify stall and clear conflict
3333036 fix(arch-pipeline hazard-WB/ID hazard): rectify a subtle dat hazard
c9ec1be fix(arch-ideal pipeline-pc): rectify pc update error
cc4c4ee fix(arch-ideal pipeline-branch): change jmp implementation
d570209 fix(arch-interrupt-irq): store interrupt request with registers
8debb77 update(arch-interrupt-clock): start to fix clock error
3d53c65 fix(arch-interrupt): rectify error on int$1 and int$2
29eeaa1 fix(arch-interrupt-pc): add a register to store oldpc in CP0
c4af5d2 fix(arch-lb): rectify lb logic error
1f071f1 fix(arch-cpu-counter): rectify counter error
3c72503 fix(arch-cpu-syscall): restore led output
fd5b61b fix(arch-lab3 cpu-instruction): adapt comparator to or logical gate
06a6a4f fix(arch-lab3 cpu-ExtOp): rectify ExtOp control error
56c1c55 fix(arch-lab3 cpu-shift): rectify error format for instructions
fb5c0e4 fix(arch-lab3 cpu-control unit): rectify control unit bugs
b7caa70 fix(arch-ALU): rectify NAND to NOR
```

图 4.10 issues 相关提交一览

华中科技大学课程设计报告

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	阅读任务书，查阅相关资料，扩展 4 条指令，完成了测试，仔细研究中断机制的实现。
第二天	实现了中断响应处理电路，完成了中断指令的扩充，完成了简易 CP0 协处理器（3 寄存器），利用偏软件方式实现了多级嵌套的优先级中断机制。
第三天	重新设计中断系统，修正原有设计中存在的种种重大缺陷；撰写问题日志，完成第一阶段的报告。
第四天	单周期多级嵌套中断验收通过；研究流水线机制；着手实现理想流水线，已完成 IF/ID 与 ID/EX 流水寄存器的实现，将 CPU 变为 3 段流水。
第五天	完成理想 5 段流水线，通过验收检测；完成简单分支跳转（flush），进行了简单的跑马灯测试（不完全测试）。
第六天	完成转发单元的设计与实现；在此基础上，完成剩余的冒险检测单元，进行气泡的插入；完成计数模块；全冒险流水线检查验收通过；完成 FPGA 模块上基础模块的实现。
第七天	完成 FPGA 流水线所有模块的实现；连接所有模块，实现了一个完整的 Pipeline MIPS CPU，检查验收通过。
第八天	完成不带 BTB 表的双预测位动态分支预测电路；研究 BTB 表的设计，研究 LRU 算法的实现，研究 BTB 模块的整体设计。
第九天	完成 BTB 模块的实现，将 2 位预测器整合至 BTB 模块；尝试硬件实现 LRU 算法。
第十天	最终验收完成；着手完成课设报告。

5 设计总结与心得

5.1 课设总结

此次实验难点在于中断系统的设计、流水线的建立以及 BTB 表的实现。作了如下几点工作：

- 1) 基于上学期的实验基础，扩展完成 4 条简单指令；
- 2) 实现中断识别与响应电路；
- 3) 实现简易 CP0 协处理器，完成 3 个特殊寄存器（Status、EPC、Cause）的相关逻辑；
- 4) 完成中断系统的耦合，编写中断处理程序（软件分支方式）；
- 5) 完成理想流水线，简单地将 CPU 分为 5 段流水；
- 6) 完成转发单元，处理 ID、EX、MEM 阶段发生的大部分数据冒险；
- 7) 完成冒险检测单元，处理 load-use 冒险、branch 数据冒险、branch 控制冒险，完成全冒险 5 段流水 MIPS CPU；
- 8) 完成所有基础组件与控制组件的 Verilog 实现，成功地将 Logisim 的电路原理图实现成 Verilog 代码，并下载到 FPGA 板上成功运行；
- 9) 完成 2 位动态分支预测状态机，完成 BTB 模块，成功地将分支预测成功率提升至 85%以上，减少数百个执行周期。

5.2 课设心得

本次课程设计可以说是迄今为止除了编译原理实验中难度最大的一门。两个星期从早到晚的不懈努力才终于完成了大部分课程设计的设计任务。现在再来回顾整个课程设计的整个过程，可以看到许多不足之处。

当第 1 天成功地扩展了 4 个指令后，便天真地以为中断也将是非常简单的事情。但结果却出乎我的意料。尽管第 2 天就把中断系统的原型实现了，可以进行 1-2-3 跳转，但发现自己的设计却存在重大问题--无法锁存中断请求，这将导致被屏蔽的中断请求丢失，完全不符合多级嵌套中断的实际情况。

作为结果，第 3 天重新设计了中断系统（尤其是中断响应电路），再结合老师的

华中科技大学课程设计报告

中断 28 问，成功地发现了中断实现中一个又一个坑，诸如屏蔽字的保存与恢复、中断号的利用、屏蔽中断与恢复中断的时机、同步清零中断请求的机制等等，最终磕磕绊绊地终于在第 4 天完成了中断系统。这一趟下来，终于使我意识到了研究一个一知半解问题的不易。如果没有老师的 28 问，不去进行任何思考，势必会陷入补坑的循环，一个接一个的坑，直到最后成一个大窟窿。

吸取了中断的教训，在进行后续流水线的设计与实现前，都进行了大范围的资料研究，尽可能地构建一个较为完整的流水线设计。有了精妙设计的加成（实际上部分设计都来源于下文参考文献中提及的几本书），实现流水线的过程没有遇到太大困难，仅仅是在同步流水寄存器清零信号时（由于 Logisim 内建的寄存器为异步清零，所以必须利用触发器与时钟掌握好清零时机）遇到一些挑战。

而将 Logisim 的电路图移植到 FPGA 平台上就更加顺风顺水。对比之前完成数字逻辑课设时的无助与挣扎，此次由于有了 Logisim 的可运行电路图（特别地，所有信号都已经通过一定的逻辑转为了时钟同步），再结合之前完成数字逻辑课设的 Verilog 编码经验，移植后仅仅遇到了 2 个接线错误，便成功地在 FPGA 上运行起了 benchmark 跑马灯。

动态分支预测的实现与中断一样，令我挣扎了 2 天。若单单地实现一个 2 位的动态分支预测器，那是十分简单的，它从原理上看仅仅是一个 Moore 型状态机，而且是一个简单的可逆计数器状态机。在第二周的第 3 天，仅仅花了半天的时间，就成功地在 FPGA 上实现了 2 位动态分支预测。真正令人头疼的是 BTB 表的实现。BTB 表的实现实际上是全相联 Cache 的实现加上状态机的整合。将状态机整合进 BTB 表难度不大，但用硬件实现全相联存储器以及 LRU 淘汰算法，着实废了一些功夫。尽管实现了一个全相联 Cache，但利用计数器实现的 LRU 淘汰算法并未成功地整合进 BTB 表，以至于最后的 BTB 表只有一行可用，这是此次课设的一个小小的遗憾。

总而言之，通过此次课设，学到了许多东西，重新认识与掌握了包括之前在计算机系统基础课上学习到的流水线，以及在组成原理课上接触的中断系统。希望老师们能够不断地改进这个课设（尽管现在已经足够有趣）

对于本次课程设计，我有一点小小的建议。按照本次课程设计的宗旨，每个人都要完成自己的一份 CPU，原则上说是没有分工可言的，分组仅仅是加强组内的交流，而不是进行分工协作。若以后的课设也采取不分工形式的话，个人建议分组的规模不应过大，太大的话容易造成贫富差距过大，组内进度最后的成员容易消极懈怠，一味

华中科技大学课程设计报告

地想要依赖所谓的大腿，而不进行独立思考，这有违分组的初衷。所以，建议一组的人数控制在 4 人以内。

最后，十分感谢老师们无论是在实验室还是即时通讯工具上（甚至深夜）对于我在本次课程设计中无数问题的耐心解答。

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, 2015
- [3] John Hennessy, David Patterson. 计算机系统结构-量化计算方法: 机械工业出版社, 2012
- [4] Dominic Sweetman. See MIPS Run(2nd Edition). San Francisco: Morgan Kaufmann, 2007

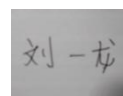
• 指导教师评定意见 •

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：

A rectangular box containing a handwritten signature in black ink, which appears to read "刘一峰" (Liu Yifeng).