华中科技大学

2022

硬件综合训练　　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：	CS2004
学　　号：	U202012046
姓　　名：	范彦廷
电　　话：	17762584717
邮　　件：	2390335608@qq.com

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 11
2.3 流水 CPU 设计 12
2.4 气泡式流水线设计 13
2.5 数据转发流水线设计 13
2.6 动态分支预测机制 13

3 详细设计与实现 15

3.1 单周期 CPU 实现 15
3.2 中断机制实现 20
3.3 流水 CPU 实现 23
3.4 理想流水线实现 25
3.5 气泡式流水线实现 25
3.6 重定向流水线实现 27
3.7 动态分支预测机制实现 28
3.8 流水线 CPU 上板实现 31

4 实验过程与调试 32

4.1 测试用例和功能测试 32
4.2 性能分析 33

4.3 主要故障与调试.....33
4.4 实验进度.....35
5 团队任务.....36
5.1 选题与设计.....36
5.2 团队任务负责部分.....36
5.3 实现效果.....37
6 设计总结与心得.....38
6.1 课设总结.....38
6.2 课设心得.....38
参考文献.....40

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 24 条基本 32 位 RISC-V 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 RISC-V32 指令集，最终功能以 RARS 模拟器为准。
2	ADDI	立即数加	
3	AND	与	
4	ANDI	立即数与	
5	SLLI	逻辑左移	
6	SRAI	算数右移	
7	SRLI	逻辑右移	
8	SUB	减	
9	OR	或	
10	ORI	立即数或	
11	XORI	或非/立即数异或	
12	LW	加载字	
13	SW	存字	

#	指令助记符	简单功能描述	备注
14	BEQ	相等跳转	
15	BNE	不相等跳转	
16	SLT	小于置数	
17	SLTI	小于立即数置数	
18	SLTU	小于无符号数置数	
19	JAL	转移并链接	
20	JALR	转移到指定寄存器	If $v0==10 halt(停机指令)
21	ECALL	系统调用	else 数码管显示$a0 值
22	CSRRSI	访问 CSR 寄存器	中断相关，可简化
23	CSRRCI	访问 CSR 寄存器	中断相关，可简化
24	ERET	中断返回	异常返回
25	SRA	逻辑右移	
26	XOR	异或	
27	LBU	取单个字，进行零扩展	
28	BLT	小于跳转	

2 总体方案设计

2.1 单周期 CPU 设计

单周期 CPU 本次采用的方案是硬布线控制方案，将指令与数据分开存储，使用硬布线将二者进行联系。为便于开发，本次采用同步时序，CPU 内全部元件听从同一时钟周期。在一个周期内，首先控制器读取指令并转化得到 ALU 所需指令，然后从指令的其余部分获得目的寄存器或者立即数，随后将转化得到的数字经过 ALU 计算后写入内存或寄存器。

总体结构图如图 2.1 所示。

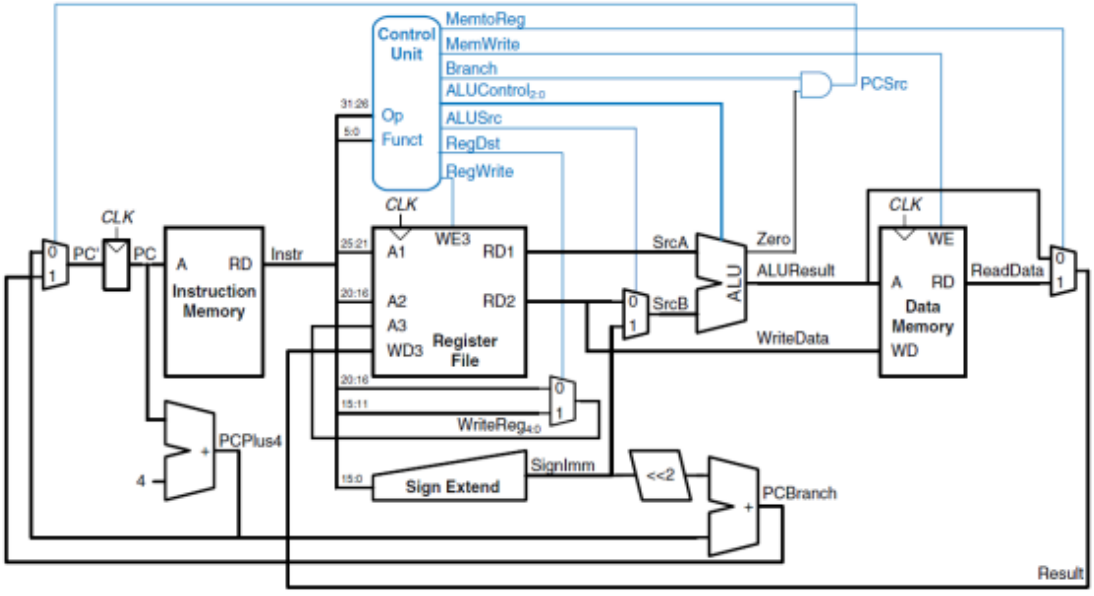

图 2.1 总体结构图

2.1.1 主要功能部件

以下为单周期 CPU 的几个核心部件的设计原理。

1. 程序计数器 PC

程序计数器为一个寄存器，用于存储下一条将要运行的指令地址。其输入从多种可能的下条地址中选取，经过时钟周期的控制输出即将执行的地址，传输给 IM。

2. 指令存储器 IM

指令存储器存储了程序所有可能运行的指令，具体执行指令地址由 PC 的输出地址决定。由于 RISC-V 为定长指令，每条指令均为 4 字节，因而 IM 中地址仅有 PC 输出地址中的 2-11 位决定。

3. 运算器

运算器的主要工作为，根据控制器传来的 ALUOP 指令，将两个操作数进行运算。其输入和输出接口如下。

表 2.1 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
Less	输出	1	小于标记
Equal	输出	1	等于标记
≥	输出	1	大于等于标记

每个 ALUOP 对应的 ALU 功能如下表：

表 2.2 算术逻辑运算单元 ALUOP 与对应功能描述

输入 OP	功能描述
0	逻辑左移
1	算术右移
2	逻辑右移
3	无符号乘法，高位使用 result2 存储
4	无符号除法，result2 存储余数
5	加法

输入 OP	功能描述
6	减法
7	按位与
8	按位或
9	按位异或
10	按位或非
11	有符号比较，小于输出 1
12	无符号比较，小于输出 1

4. 寄存器堆 RF

寄存器堆由 32 个 32 位寄存器构成，其输入引脚与输出引脚的名称与功能如下：

表 2. 3 寄存器堆输入输出引脚名称与对应功能描述

引脚	输入/输出	位宽	功能描述
R1#	输入	5	源操作寄存器 1 编号
R2#	输入	5	源操作寄存器 2 编号
W#	输入	5	写入寄存器编号
WE	输入	1	现在要写入寄存器
CLK	输入	1	时钟使能信号
RDin	输入	32	写入寄存器内容
R1	输出	32	寄存器 R1#存储内容
R2	输出	32	寄存器 R2#存储内容

5. 内存 MEM

内存和指令存储器结构类似，也是由 10 条地址线（2-11 位）进行寻址，要求访问按字对齐。寻址的地址来源于寄存器中的地址或者 ALU 的计算结果。如果需要访问某个特定的字，或者写入到某个非对齐的地址，需要先将对应的字取出，然后使用 0-1 位来从中选择取出的字节，写入同理。

2.1.2 控制器的设计

控制器需要从指令中得到源操作数的具体数值（如 I 型指令）或地址，以及对取出的数字进行的操作种类，和最后写入的寄存器编号或内存地址。其中最核心的就是控制信号，通过控制信号可以对整个电路的功能进行控制。控制器的控制信号定义与作用如表 2.3：

表 2. 2 主控制器控制信号的作用说明

控制信号	取值	说明
R1_used	0、1	是否需要使用操作数 R1
R2_used	0、1	是否需要使用操作数 R2
AluOP	0-12	使用 ALU 的指令编号
BEQ	0、1	是否为 BEQ 指令
BNE	0、1	是否为 BNE 指令
MemToReg	0、1	是否需要从内存写入寄存器
MemWrite	0、1	是否需要写入内存
AluSrcB	0、1	第二操作数是为寄存器 2 中值（0）还是立即数（1）
RegWrite	0、1	是否需要写入寄存器
LBU	0、1	是否为 LBU 指令
JALR	0、1	是否为 JALR 指令
JAL	0、1	是否为 JAL 指令
BLT	0、1	是否为 BLT 指令
URET	0、1	是否需要中断返回
S_Type	0、1	是否为 S 型指令
Ecall	0、1	是否为 ecall 指令

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。该控制信号表的框架如表 2.3 所示。为简化表格，因而 BEQ 类指示是否为该条指令的输出被省略了。

表 2.3 主控制器控制信号框架

指令	ALU_OP	MemToReg	ALU_Srcb	RegWrite	Ecall	S_Type	RS1_used	RS2_used
ADD	5		1	1			1	1
SUB	6			1			1	1
AND	7			1			1	1
OR	8			1			1	1
SLT	11			1			1	1
SLTU	12			1			1	1
ADDI	5		1	1			1	
ANDI	7		1	1			1	
ORI	8		1	1			1	
XORI	9		1	1			1	
SLTI	11		1	1			1	
SLLI	0		1	1			1	
SRLI	2		1	1			1	
SRAI	1		1	1			1	
LW		1	1	1			1	
SW			1			1	1	1
ECALL					1			
BEQ	6						1	1
BNE	6						1	1
JAL								
JALR			1				1	
URET					1			
SRA	1						1	1
XOR	9						1	1
LBU	5	1					1	
BLT	11		1				1	1

2.2 中断机制设计

2.2.1 总体设计

中断类型大体可以分为内部中断和外部中断，本次实验设计的中断为可屏蔽的外部中断。本次实验中支持单周期单级中断、单周期多级中断和流水单级中断。

对于所有的外部中断，跳转到中断指令的方法一般为：保存下一条指令地址，保护现场的寄存器，然后跳到中断源执行中断。从中断返回的方法一般为：根据 ecall 指令恢复中断现场的寄存器，指令跳转到存储的下一条指令的地址。中断的执行基本逻辑如图 2.2 所示：

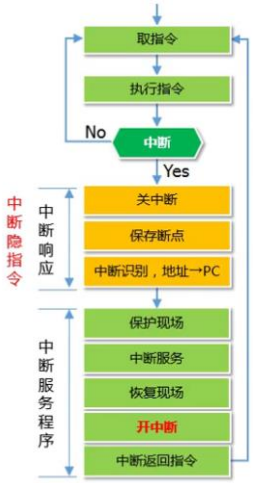

图 2.2 中断执行逻辑

对于单级中断，因为在执行中断服务程序的时候不会再接受新的中断请求，因而可以使用一个寄存器来保存当前是否处于执行中断的状态，以及中断返回地址。对于多级中断，需要考虑不同等级的中断，以及执行中断服务程序时不能被低级和同级中断打断，但是允许被更高级的中断所中断，因而需要根据执行中断服务程序的进度（即是否正在保存和恢复现场）来灵活调整中断指示标志，具体可以使用 CSSRCI 和 CSSRSI 两条指令来控制。

2.2.2 硬件设计

本次实验设计的中断方法为硬件保存，即对于每一种类、每一等级的中断，都单独使用一套完整的寄存器进行现场的保存。具体而言，一组中断寄存器由 EPC 和 IRID 分别保存中断返回地址和当前执行的中断号。此外还用 IE 寄存器保存当前是否允许执行更高等级的中断的 IE 信号，高电平有效。

对于单级中断，由于在执行中断程序时不允许被中断去执行其他中断程序，因而实现较为简单，仅需使用一组中断寄存器保存当前中断的信息即可。对于多级中断，由于支持了中断程序继续被中断，因而需要三套中断寄存器保存中断信息。使用优先编码器来对当前中断信号进行选择，若正在执行中断程序且此时允许被打断则利用中断屏蔽字和中断号的比较器判断当前中断是否允许打断当前终端，这样可以动态的改变中断优先级。

对于两种中断，均需要支持一个中断请求队列，即在中断执行过程中接受到的中断在当前中断周期结束后转而去执行队列中的请求。这里只需要对每个中断请求使用一个寄存器保存是否需要执行该中断的信号，若执行完整则激活对应清除请求的 reset 信号，即可将执行完整的中断从队列中弹出。

2.3 流水 CPU 设计

2.3.1 总体设计

本次实验实现的是五段流水线 CPU，即将一条指令分成取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB 五个阶段，通过四组寄存器保存每一段所需要的信息来实现每条指令五个阶段执行的相对独立性，这也是理想流水线的基本设计原理。但是在实际执行过程中，由于存在无条件和有条件跳转指令，因而有些指令的具体效果需要等待上条指令的结果，破坏了指令之间的隔离关系。因而在理想流水线上根据消除影响逻辑和方式不同，分成了气泡流水线和重定向流水线这两大类。此外，利用动态分支预测可以进一步加速了重定向流水线。本次课程设计中将流水线CPU进行了上板。

2.3.2 流水接口部件设计

根据五个阶段所需的信息不同，设计了如下的流水线接口：

表 2.5 不同阶段所需流水线接口

阶段	所需流水线接口
IF/ID	IR、PC、PC+4、PredictJump
ID/EX	BEQ、BNE、MemToReg、MemWrite、AluSrcB、RegWrite、S_Type、Ecall、JAL、JALR、LBU、BLT、URET、ALUOP、RD、R1、R2、IMM_1、IMM_2、PC、PC+4、IR、RS1_Forward、RS2_Forward、PredictJump
EX/MEM	JAL、JALR、MemToReg、MemWrite、LBU、RegWrite、Halt、RD、Result、WriteData、IR、PC、PC+4

阶段	所需流水线接口
MEM/WB	JAL、JALR、MemToReg、RegWrite、Halt、RD、ALUResult、ReadData、IR、PC、PC+4

2.3.3 理想流水线设计

理想流水线由于不涉及同时进入流水线的指令之间的相互作用，因而只需要将单周期的 CPU 按照上述五个步骤直接进行拆分，使用四组流水线接口即可。对于停机指令，在 EX 段处理完停机逻辑后，需要经过两个周期传送到 WB 段再最终执行停机操作，以保证全部指令都执行完毕再停机。

2.4 气泡式流水线设计

为了支持同在流水线的指令的相互作用关系，在理想流水线的基础之上需要增加由当前的指令操作信号控制的 DataHazzard 组合逻辑以判定是否需要插入气泡，插入气泡等效于清空流水线接口内寄存器的值。DataHazzard 的主要判定逻辑是根据源寄存器的使用状态和数据相关检测来判定是否存在冲突。具体而言，若判定需要插入气泡，则 ID/EX 接口和 IF/ID 接口均需要清空，以跳转到正确的指令处重新执行。由于指令的相互作用，气泡被插入了流水线，导致流水线的效率下降。

2.5 数据转发流水线设计

数据转发流水线（下称重定向流水线）以气泡流水线的基础之上进行了一定的优化，结构类似，但是减少了气泡的插入。重定向将数据可能的全部路径进行汇总，然后根据指令的执行情况获取数据，选择一条分支。但是若相邻两条指令存在数据相关，且前一条是访存指令，此时就会出现冲突（称为 Load_Use），此时仍需要使用气泡来避免冲突。

2.6 动态分支预测机制

动态分支预测是在重定向流水线上进一步的优化。重定向只是在气泡流水线上减少了气泡的产生，但是并未减少分支误取得代价，导致了周期的浪费。动态分支预测使用一个 BHT 表来记忆过去的跳转指令，使用 LRU 算法来动态更新 Cache 槽的存储内容，同时使用双预测位的四状态的状态机来判定是否需要跳转（PredictJump），不

断根据当前实际跳转情况来进行状态机的更新。具体而言，电路在 IF 段通过状态判断出预测结果，在 EX 段根据实际结果通过硬件来更新状态机。通过这两个方法来实现预取得正确的指令，减少分治误取的代价。

2.6.1 BHT 表的设计

BHT 表全程分支预测分支历史表，存放了每次分支指令的地址、分治目标地址、历史跳转位、valid 位，硬件设计为 8 位的 Cache 槽。在具体插入地址时，优先插入空行，其次是最古老的跳转地址（即 LRU 算法）。

2.6.2 动态分支预测的流水线设计

动态分治预测在重定向流水线的基础上，增加了利用 BHT 进行预测功能。有以下三种情况：

1. BHT 未命中，则与正常重定向流水线行为一致。

2. BHT 命中但预测失败。此时需要插入气泡，将预取指令取出，并且更新 BHT 表。此时与正常重定向流水线行为大致一致。

3. BHT 命中且预测成功。此时减少了气泡的插入，且 IF 指令预取正确，电路继续指令。

整体而言，预测成功可以减少平均误取深度和气泡插入数目，因而效率相对于重定向流水线又有了进一步的提升。

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) 程序计数器（PC）

① Logism 实现：

使用一个 32 位寄存器实现程序计数器 PC，触发方式为下降沿触发，输入为下一条将要执行的指令的地址（用多路选择器实现），输出为当前执行指令的地址。Halt 为停机信号，将此控制信号通过非门取反之后和时钟相与，当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，与 Go 信号或之后再与与时钟信号相与，屏蔽时钟信号，使整个电路停机，同时支持 Go 信号恢复电路功能，如图 3.1 所示。

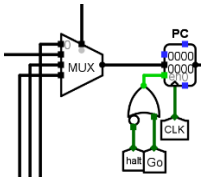

图 3.1 程序计数器（PC）的 Logisim 实现

② FPGA 实现：

在程序中，程序计数器 PC 的 Verilog 代码如下：

```
always @* begin
        if (sig.branch[0] && (sig.branch[1] ^ !aluRes)) nextPC <= PC + (imm << 1);
        else if (sig.irOp == IR_JAL) nextPC <= PC + (imm << 1);
        else if (sig.irOp == IR_JALR) nextPC <= R1 + (imm);
        else nextPC <= PC + 4;
    end
```

2) 指令存储器（IM）

① Logism 实现：

使用一个只读存储器 ROM 实现指令存储器（IM）。设置该只读存储器的地址位宽为 10 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 10 位，且严格按照字对齐，故将 32 位指令地址高位部分（12-31 位）

和字节偏移部分（0-1 位）直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为指令存储器的输入地址。如图 3.2 所示。

图 3.2 指令存储器（IM）的 Logisim 实现

② FPGA 实现：

选择 ROM 的数据位宽为 32 位，通过屏蔽指令地址高位部分和字节偏移部分，该 RAM 的地址位宽为 10 位，选择 RAM 的大小选择为 1024。去掉 addr 的高位，指令存储器 IM 的 Verilog 代码如下：

```
int data [1 << SIZE];
assign dout = data[addr >> 2];
```

3) 内存（MEM）

① Logisim 实现

电路中的内存和指令存储器一样，是由十位地址线（2-11 位）寻字，低两位寻字节的存储器，如下图 3.3，包含了 LBU 指令通路。

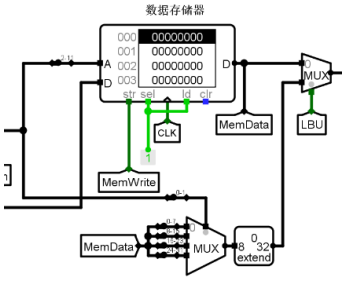

图 3.3 内存（MEM）的 Logisim 实现

② FGPA 实现

FGPA 基本根据电路图进行实现，增补了另一种取双字节的方式。

```
always @(posedge clk) begin
    if (rst) begin
        for (int i = 0; i < (1 << SIZE); ++i) data[i] <= 0;
    end
    else if (we) case(mode[1:0])
```

```
            0: case (addr[1:0])
                3: data[addr>>2][31:24] <= din[7:0];
                2: data[addr>>2][23:16] <= din[7:0];
                1: data[addr>>2][15:8] <= din[7:0];
                default: data[addr>>2][7:0] <= din[7:0];
            endcase
            1: case (addr[1])
                1: data[addr>>2][31:16] <= din[15:0];
                default: data[addr>>2][15:0] <= din[15:0];
            endcase
            default: data[addr>>2] <= din[31:0];
        endcase
    end
```

4) 寄存器（RegFile）

① Logisim 实现

RegFile 由 32 个 32 位的寄存器构成，其主要接口和功能都已经在总体设计方案中阐明。在设计包中 Logisim 的 RegFile 电路已经封装完成，因而只需要直接调用即可。对于停机指令，可以在外侧使用简单逻辑电路进行判定具体输入。Losigim 实现如图 3.3 所示：

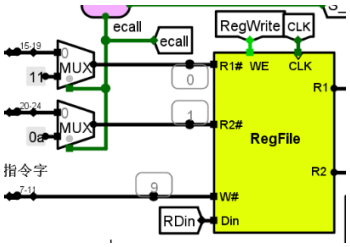

图 3.3 寄存器文件（RegFile）的 Logisim 实现

② FGPA 实现

Verilog 中将 Logisim 中进行了直接的翻译，使用 32 个 int 作为寄存器文件。

```
localparam REG_NUM = 32;
    int regs [REG_NUM];
    assign A = rA? regs[rA] : 0;
    assign B = rB? regs[rB] : 0;
```

```
always @(negedge clk) begin
    if (rst) begin
        regs[1] <= 0;
        regs[2] <= (1 << (7 + 2)) - 4;
        for (int i = 3; i < REG_NUM; ++i) regs[i] <= 0;
    end else if (we) regs[rW] <= W;
end
```

5) ALU

① Logisim 实现

和 RegFile 相同，ALU 也是已经提供好的子电路，接口也相对简单。

② FPGA 实现

Verilog 中仅需要将每个 ALUOP 对应的计算指令进行转化即可，代码相对简单，因而不再报告中赘述。

3.1.2 数据通路的实现

数据通路基本由 PC->ROM->寄存器和控制器->ALU->MEM 的流程构成，根据每条指令的不同，控制信号会生成不同的数据选择通路，来灵活选择每条指令的具体数据通路。下图为 Logisim 中产生的数据通路。

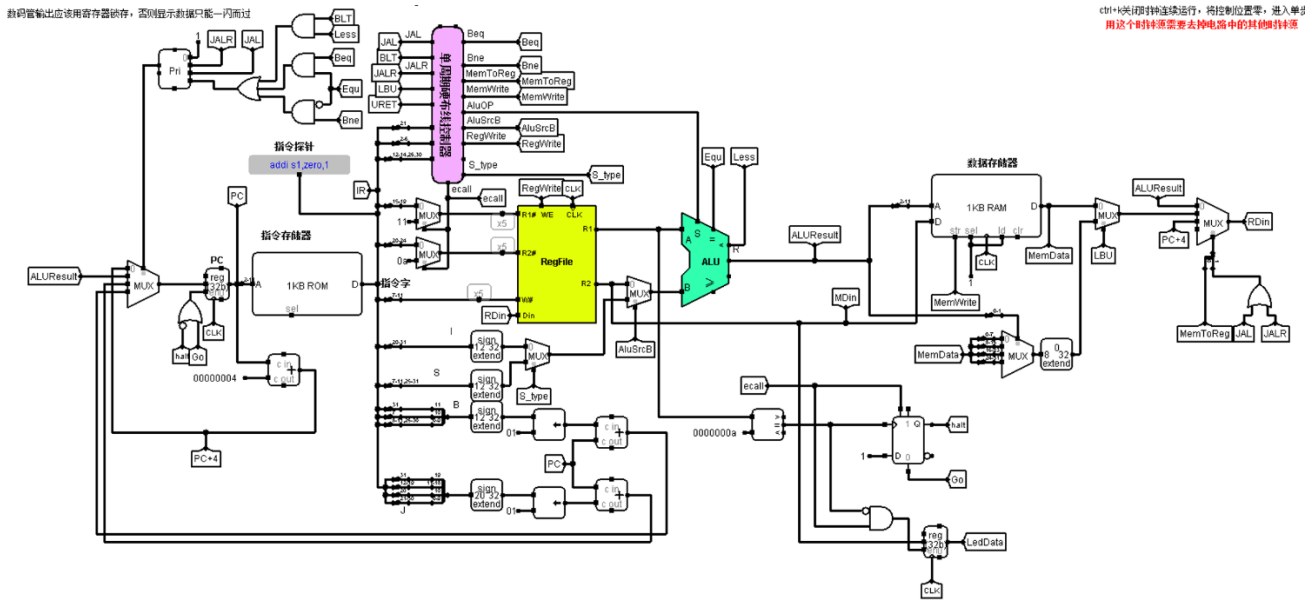

图 3.4 单周期 CPU 数据通路（Logisim）

本次实验使用 Verilog 实现了流水线 CPU，因而此处省略 Vivado 中单周期 CPU 的

原理图。

3.1.3 控制器的实现

根据总体方案设计中控制器的设计那一小节的相关内容，分别在 Logism 和 Vivado 上进行主控制器、Branch 控制器、ECALL 控制器的具体实现。

① Logisim 实现

主控制器对照表位于表 2.3。具体每个信号的电路通过填表，由 Logisim 自行生成。主控制器又由两个子电路构成：ALUOP 产生和其他控制信号产生器。最终总体结构图 3.4 所示：

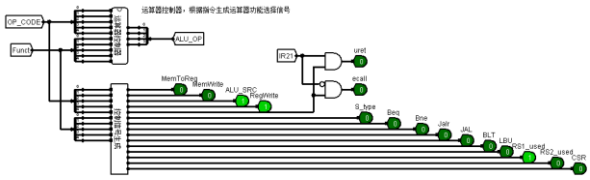

图 3.4 主控制器电路（Logisim）

Branch 控制器利用 ALU 计算得到的控制信号和主控制器信号来联合控制跳转指令，原理图见图 3.5：

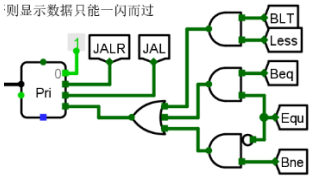

图 3.5 Branch 控制器电路（Logisim）

ECALL 控制器主要实现停机和 LedData 的输出，根据寄存器中的值和 Ecall 信号来进行控制，原理图见图 3.6：

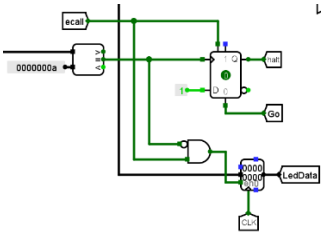

图 3.6 ECALL 控制器电路（Logisim）

② FPGA 实现

Logisim 上是根据每一位的电路直接进行的控制，但是由于输入信号过多，因而逐位去比对并发生信号是非常容易出错的。一个简易的方法是对于每条指令单独做一个控制信号，形成相对独立的数据通路，以减少大量的逻辑运算的输入工作。

对于立即数类的处理可以在控制器这里同步完成：

```
assign imm =
        sig.irType == IR_TYPE_I ? signed'(ir[31:20]) :
        sig.irType == IR_TYPE_S ? signed'({ir[31:25], ir[11:7]}) :
        sig.irType == IR_TYPE_B ? signed'({ir[31], ir[7], ir[30:25], ir[11:8]}) :
        sig.irType == IR_TYPE_U ? signed'(ir[31:12]) :
        sig.irType == IR_TYPE_J ? signed'({ir[31], ir[19:12], ir[20], ir[30:21]}) : 0;
```

以此类推，最终便可以实现整个主控制器中所有控制信号和立即数的生成。由于原理图过大，因而在此省略 Vivado 仿真图。

3.2 中断机制实现

对于各中断信号的产生，沿用提供的电路进行，如图 3.7：

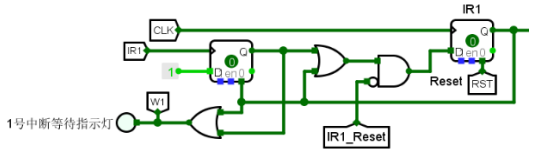

图 3.7 中断信号产生器

本次实验将多级中断进行了 Vivado 仿真上板。

3.2.1 单级中断

单级中断和多级中断都需要涉及下一条指令的跳转地址。在单周期 CPU 的三种选择之上，使用优先编码器和多个选择器通过控制信号从 PC+4、中断源、EPC 中地址选择下一条地址，中断源地址可以通过 RARS 仿真器看到具体的地址。Logisim 原理图如图 3.8 所示：

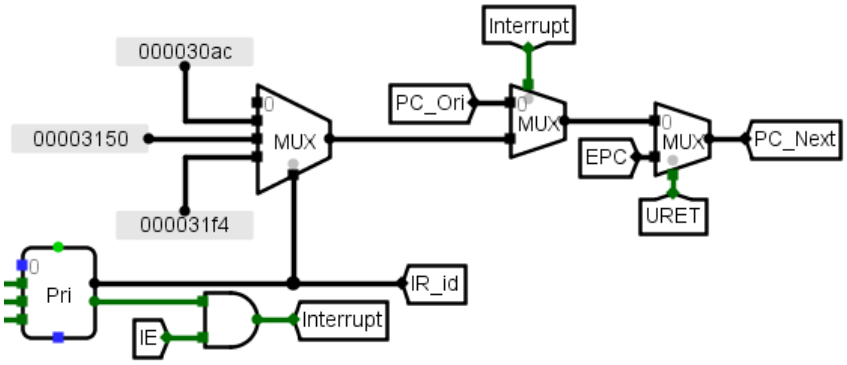

图 3.8 跳转地址选择电路（Logisim）

硬件实现上通过使用 EPC 和 IE 寄存器来保存当前中断的状态和返回地址，以及清空中断请求队列。电路如图 3.9 所示：

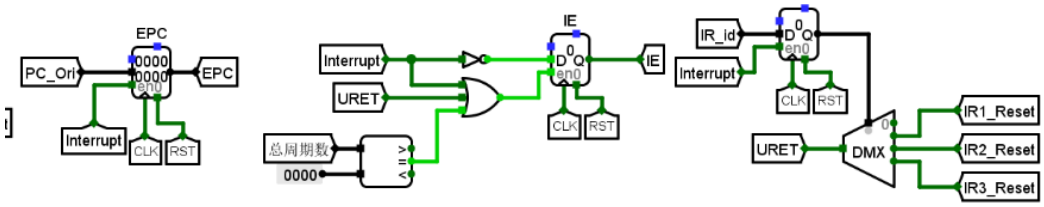

图 3.9 硬件保存中断电路（Logisim）

对于流水线 CPU，相较于单周期 CPU 的主要不同之处在于，BranchTaken 信号和 stall 信号也是需要进行现场保护的，后续中断程序结束后返回的地址由寄存器中的信号决定。因而只需增补图 3.10 所示电路：

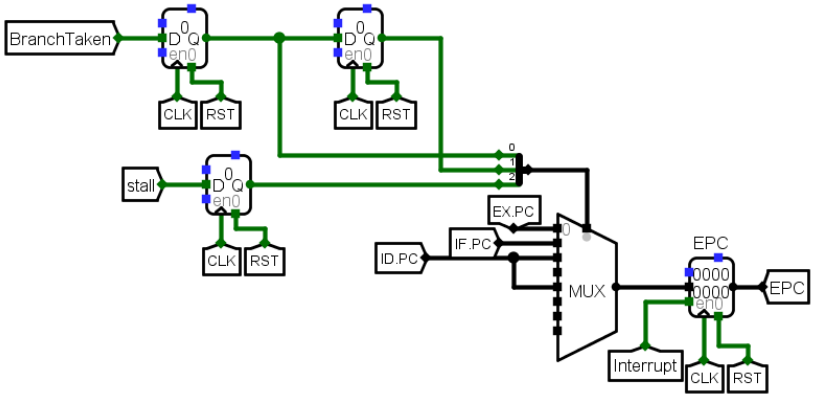

图 3.10 中断返回地址筛选电路（Logisim）

3.2.2 多级中断

① Logisim 实现

多级中断中利用硬件保存的实现方法使用三套 EPC 和 IR 寄存器来保存中断号和返回地址，每一套的处理逻辑同图 3.9 所示。并同时采用 CSRRCI 和 CSRRSI 来控制开关中断，具体而言是利用这两条指令来控制 IE 寄存器的信号。IE 寄存器初始为 1，高电平表示开中断，IE 处理逻辑位于图 3.11 左上部分。具体 Logisim 电路如图 3.11 所示：

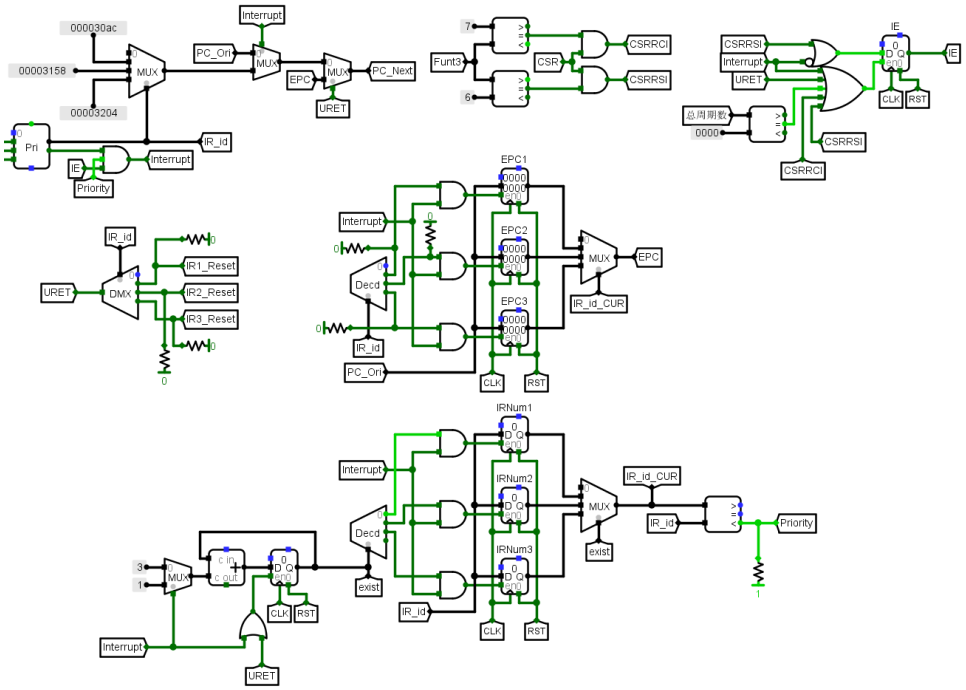

图 3.11 件实现多级中断电路（Logisim）

② FPGA 实现

Verilog 的实现较为容易：首先判定是否处于开中断阶段，否则比较当前中断号和正在执行中断号的大小关系，若允许打断则跳转执行更高级中断。

下面为维护 IR 寄存器的 Verilog 代码：

```
always @(posedge clk) begin
    if (rst) {reqStatus, irWait, ir} <= 0;
    else for (int i = 0; i < 32; ++i) begin
        if (clr[i]) ir[i] <= 0;
        else if (irWait[i]) {irWait[i], ir[i]} <= 'b01;
        if (req[i] && !reqStatus[i]) begin
            irWait[i] <= 1;
            reqStatus[i] <= 1;
        end
        else if (!req[i]) reqStatus[i] <= 0;
    end
end
```

下面为是否允许中断的 Verilog 代码：

```
always @* begin
        if (EX.sig.irOp == IR_NOP) begin
            intID = 0;
            intAddr = 0;
        end
        else begin
            intID = intAcc & -intAcc;
            intAddr = 0;
            for (int i = 0; i < 32; ++i) begin
                if (intID >> i & 1) intAddr |= (i + 1) << 2;
            end
        end
    end
```

下面为 EPC Verilog 代码：

```
    always @(posedge clk) begin
        if (rst || intRet) IE <= 0;
        else if (intLoad) IE <= intID;

        if (intLoad) begin
            if (jump) EPC <= realPC;
            else EPC <= EX.PC + 4;
        end
    end
```

3.3 流水 CPU 实现

该部分主要阐述了流水线接口的子电路设计。

① Logisim 实现

各流水线接口为若干个寄存器，对每条输入和输出使用一个寄存器进行存储，并使用 reset 和 flush 等复位信号进行清空（插入气泡）。大致电路如图 3.12 所示：

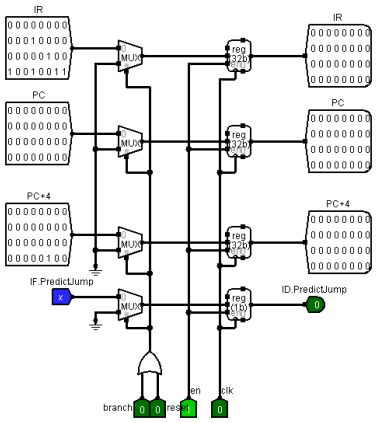

图 3.12 流水接口电路（Logisim），以 IF/ID 为例

② FPGA 实现

为简化 Verilog 代码编写难度，因而泛化一个流水线接口结构 CPUPipline，对于四个接口仅修改其对外接口函数。下面为 PipelineReg 模块，用以进行实例化：

```
module PipelineReg #(
    parameter WDATA
) (
    input clk, input rst, input clr, input en,
    input [WDATA-1:0] din,
    output bit [WDATA-1:0] dout
);
    always @(posedge clk) begin
        if (clr || rst) dout <= 0;
        else if (en) dout <= din;
    end
endmodule
```

而每个流水线接口则单独设计接口，其中 ID/EX 接口如下所示，其余类似：

```
typedef struct packed {
    int PC, IR;
    Signal sig;
    RegAddr rd;
    int R1, R2, imm;
    logic [1:0] fwdR1, fwdR2;
```

```
    logic predJump;
} PiplineRegEX;
```

由于寄存器数目较多，因而 Vivado 仿真电路图过大，不便在报告中呈现。

3.4 理想流水线实现

理想流水线是在单周期CPU 的基础上，将各部件根据五段的分法分成五个部分，每个部分使用流水线接口进行拼接。Logisim 电路如图 3.13 所示：

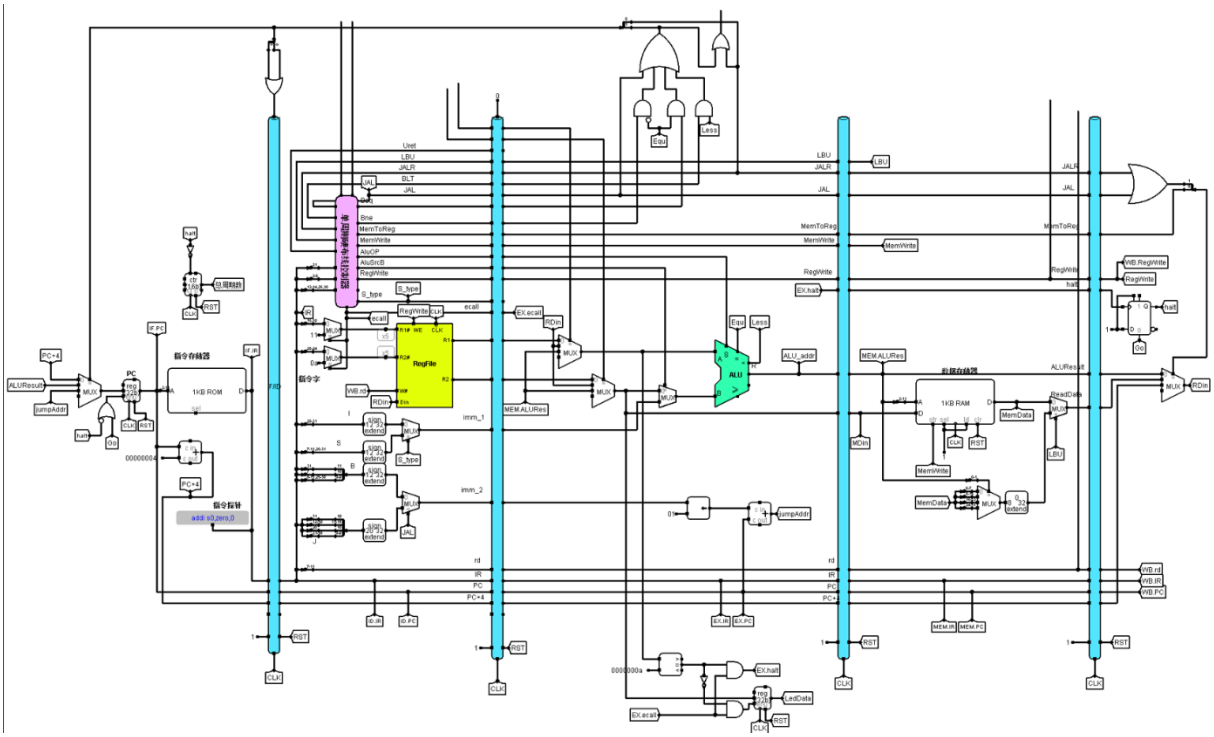

图 3.13 理想流水线电路图（Logisim）

3.5 气泡式流水线实现

气泡流水线就是在理想流水线的基础之上，根据当前电路的状态（RS1 和 RS2 是否被使用，且是否需要写入内存、写入寄存器）增加 DataHazzard 逻辑判断，输出 stall=DataHazzard 信号和 flush=DataHazzard|BranchTaken 信号，其具体组合电路如图 3.14 所示：

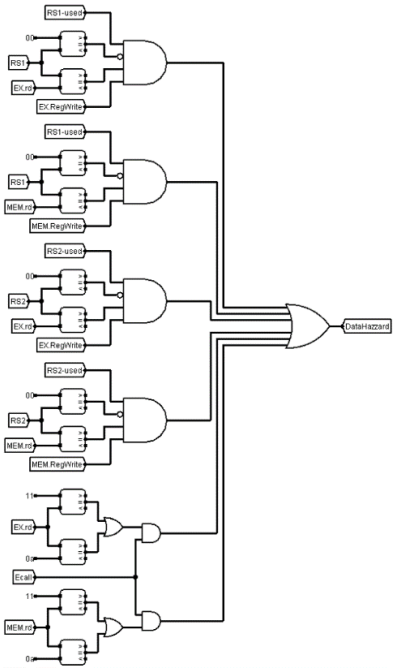

图 3.14 气泡流水线 DataHazzard 逻辑判断（Logisim）

当气泡被插入之后，flush 信号的激活 IF/ID 和 ID/EX 流水线接口都会清空，并且取指也会因为 stall 信号的激活，暂停一个时钟周期以完整消除分支误取对 CPU 的影响，以保证从正确的指令处重新执行。完整气泡流水线 Logisim 电路如图 3.15 所示：

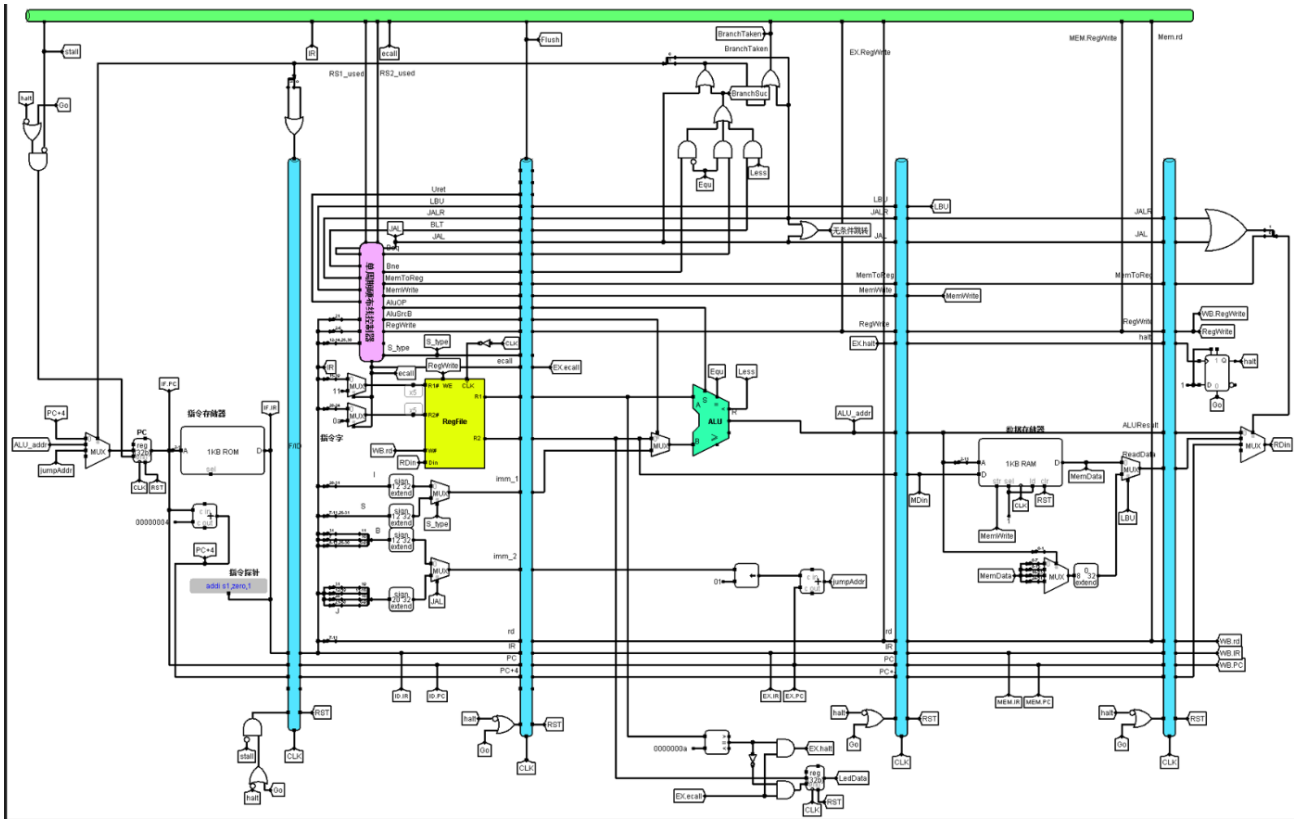

图 3.15 气泡流水线完整电路（Logisim）

3.6 重定向流水线实现

① Logisim 实现

重定向流水线在气泡流水线的基础上，首先增加了 Load_Use 逻辑判断，其基本原理见 2.5 节，Losigim 电路如图 3.16 所示：

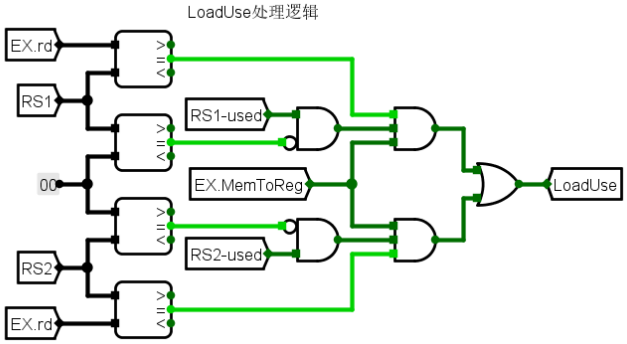

图 3.16 Load_Use 处理逻辑（Logisim）

此外依照数据转发的思想，设计了 RS1 的转发和 RS2 的转发逻辑如图 3.17 所示：

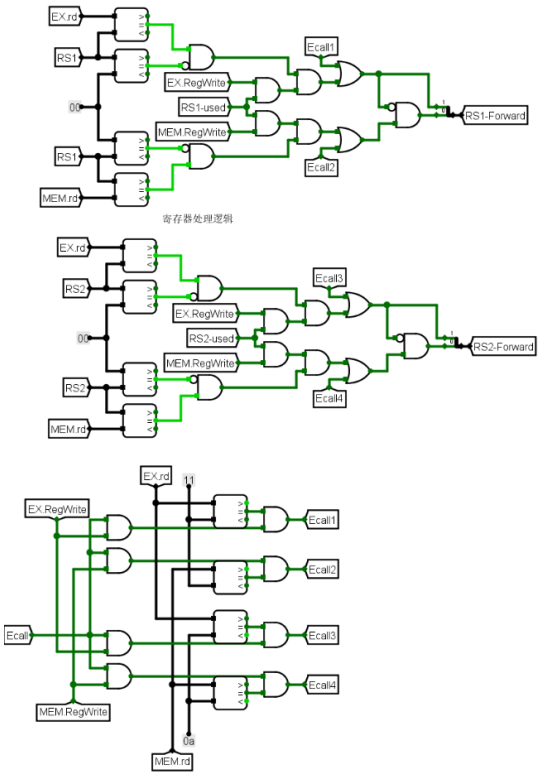

图 3.17 RS1 和 RS2 数据转发逻辑（Logisim）

根据 Flush=BranchTaken|Load_Use 和 stall=Load_Use 逻辑，可以在气泡流水线的基础上，得到重定向流水线的完整电路，如图 3.18 所示：

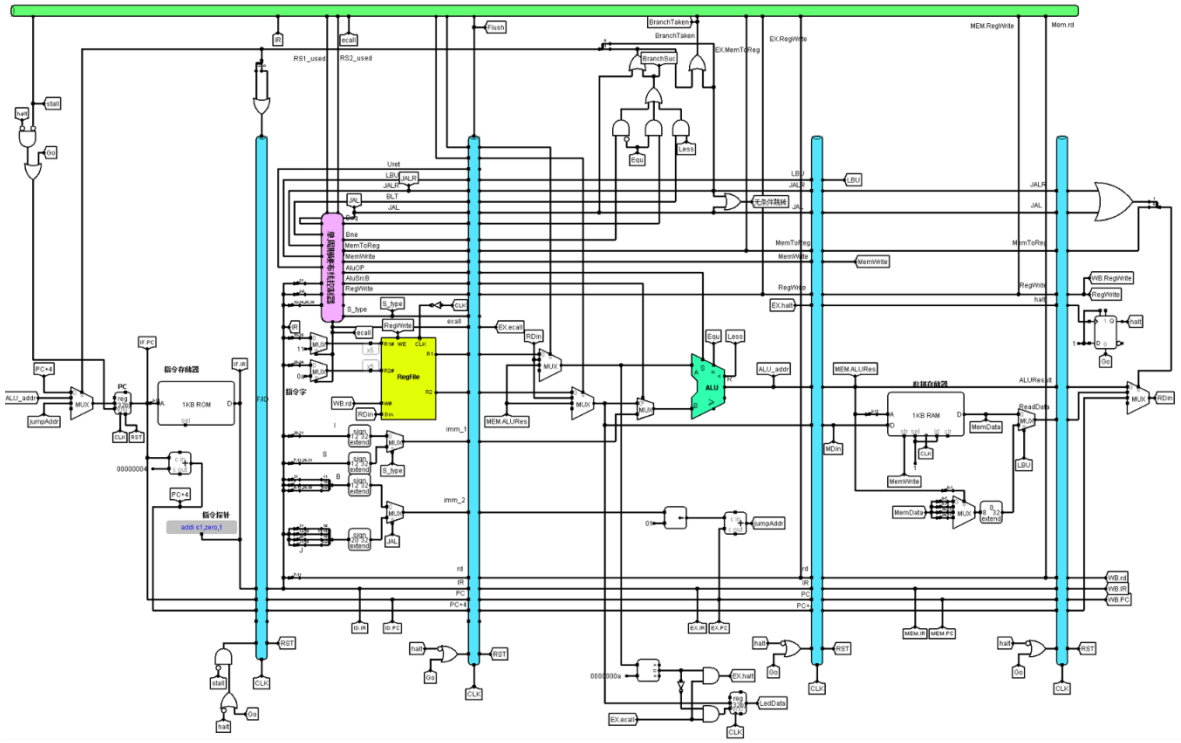

图 3.18 重定向流水线电路（Logisim）

② FPGA 实现

Verilog 实现仅需将 Logisim 电路直接转化即可生成对应的重定向逻辑。

```
logic EXconf1, EXconf2, MEMconf1, MEMconf2;
assign EXconf1 = rs1 && rs1 == EXrd;
assign EXconf2 = rs2 && rs2 == EXrd;
assign MEMconf1 = rs1 && rs1 == MEMrd;
assign MEMconf2 = rs2 && rs2 == MEMrd;
assign loadUse = (EXconf1 || EXconf2) && EXload;
assign fwdR1 = EXconf1? 2: MEMconf1? 1: 0;
assign fwdR2 = EXconf2? 2: MEMconf2? 1: 0;
```

3.7 动态分支预测机制实现

① Logisim 实现

动态分支预测在重定向上增补了 BTB 逻辑。利用寄存器设计 8 位 Cache 槽，每一位 Cache 槽如图 3.19 所示：

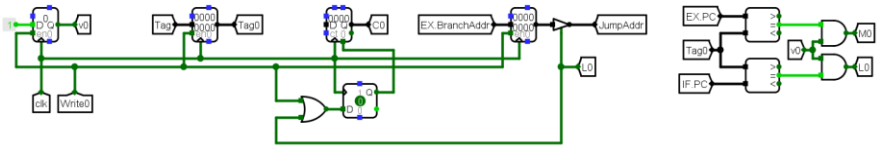

图 3.19 Cache 槽（Logisim）

此外根据优先写入空 Cache 槽，优先覆盖最古老的跳转地址的原则，有以下的 LRU 硬件实现电路如图 3.20 所示：

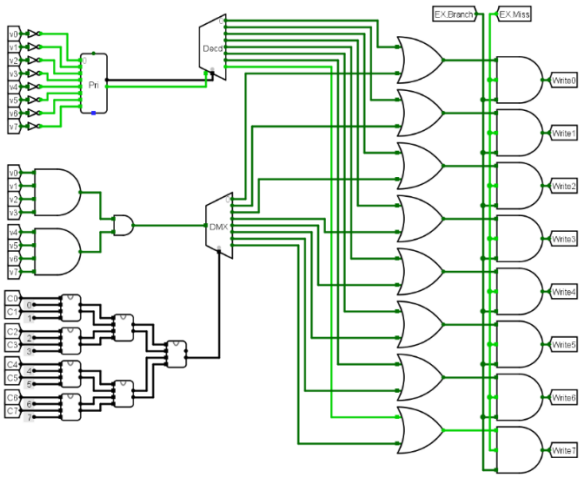

图 3.20 LRU 电路实现（Logisim）

并且根据双位预测，使用一个两位的寄存器以保存当前跳转的状态机状态，每次根据实际跳转情况（BranchTaken 为高电平表示当前跳转）和预测情况来更新状态机状态，共计 8 个电路。电路实现如图 3.21 所示：

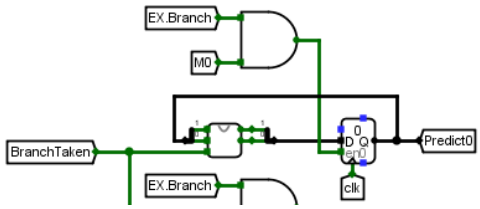

图 3.21 双位预测电路实现（Logisim）

最后将各部分电路进行综合即可得到最终的 BTB 电路。最终的动态分支预测电路如图 3.22 所示：

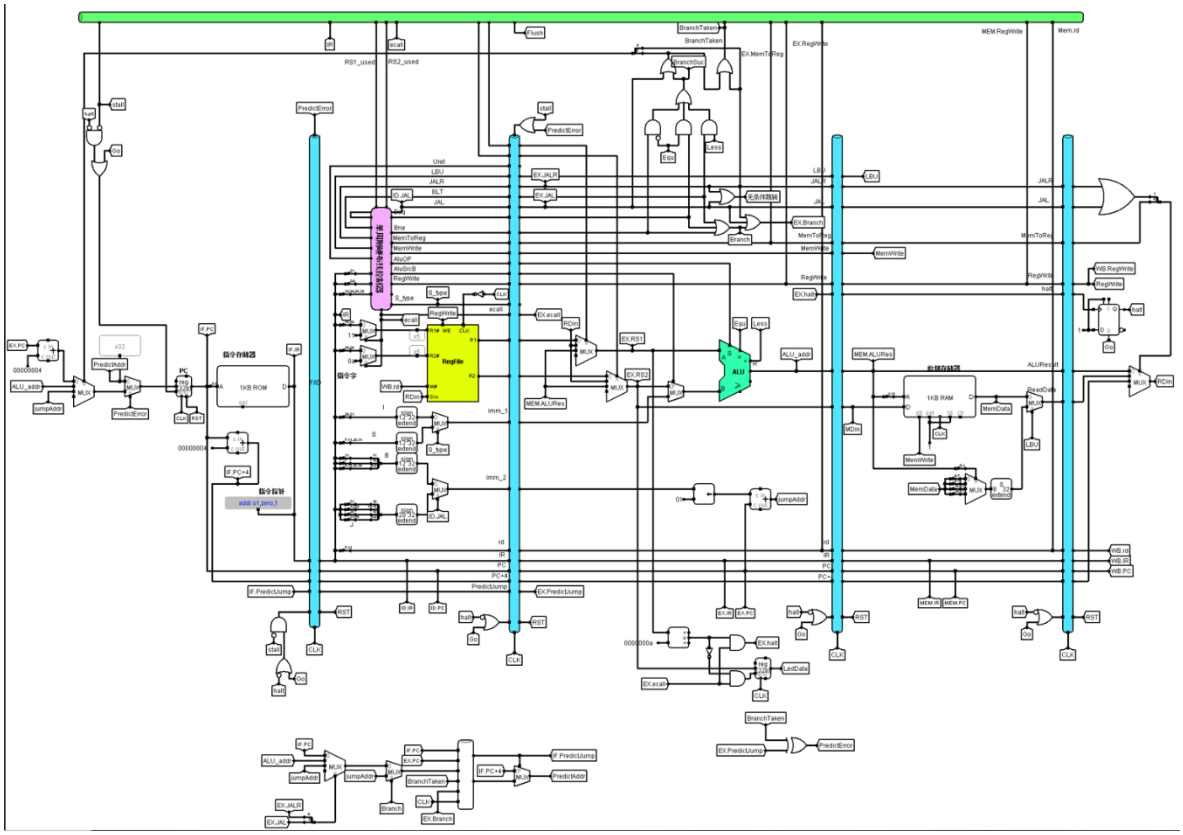

图 3.22 动态分支预测电路实现（Logisim）

② FPGA 实现

BTB 的主体逻辑已经在 2.6.2 节得到阐述，由于 Verilog 代码较长，且为主体逻辑的直接复现，因而不便于直接呈现在本文中。

将 BTB 与重定向逻辑相结合，可以得到以下的代码：

```
LUController luCtrl (
    rs1, rs2, EX.rd, MEM.rd,
    EX.sig.load || EX.sig.irOp == IR_ECALL,
    fwdR1, fwdR2, loadUse
);
BTB #(.SIZE(SIZE_BTB)) btb (
    clk, rst,
    isBranch, EX.PC, realPC, jump,
    PC, branchPC, predJump
);
assign predPC = predJump ? branchPC : PC + 4;
```

3.8 流水线 CPU 上板实现

根据前数节的内容，利用以上 Verilog 代码可以得到 Vivado 仿真的流水线电路如图所示图 3.23：

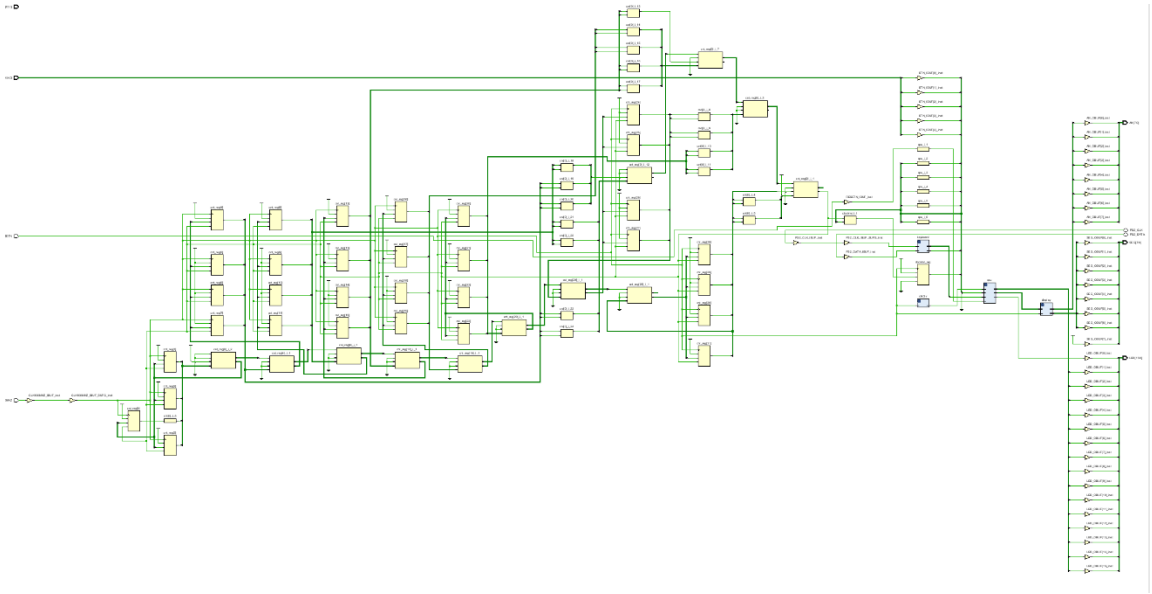

图 3.23 Vivado 仿真的 Verilog 电路整体布线图

4 实验过程与调试

4.1 测试用例和功能测试

本次实验中，单周期 CPU、气泡流水线 CPU、重定向流水线 CPU、单级和多级中断均在头歌平台通过仿真测试。

具体测试用例有 Benchmark.asm 和中断测试.asm 两个测试文件。

4.1.1 Benchmark.asm 测试

以下为 Benchmark.asm 在单周期、气泡流水线、重定向流水线、动态分支预测电路的运行结果：

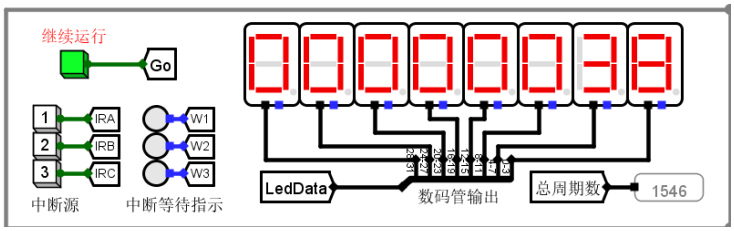

图 4.1 单周期 CPU 运行最终结果

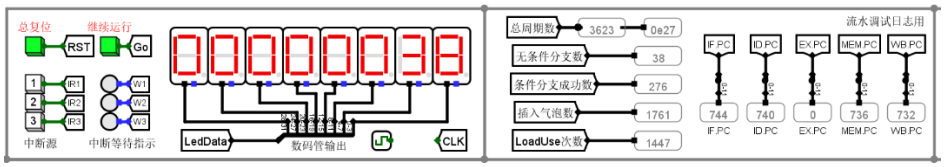

图 4.2 气泡流水线 CPU 运行最终结果

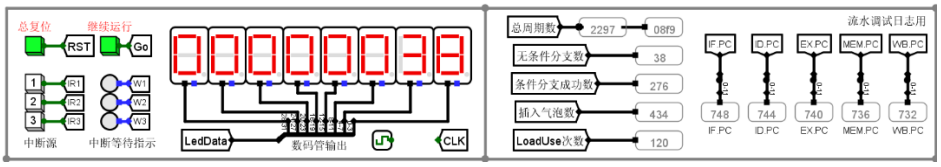

图 4.3 重定向流水线 CPU 运行最终结果

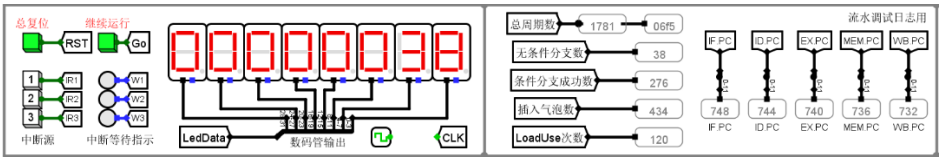

图 4.4 动态分支预测重定向流水线 CPU 运行最终结果

以上各运行结果均与实际结果一致。

4.1.2 中断测试.asm 测试

对于三种中断，对于中断按键操控的中断源均能在开中断时及时响应，并且在当前中断操作结束完成后及时响应中断请求队列中级别最高的中断请求，因而可以认为实验成功。

4.2 性能分析

在单周期流水线 CPU 中，执行指令条数为 1546，但是每周期长度较长。

对于多周期的流水线，仅估算起见，每个周期长度可认为约为单周期 CPU 的周期长度的 1/5。对于气泡流水线，需要执行 3624 周期，相较于单周期流水线，时间缩短一半。对于重定向流水线，所需执行周期数为 2298，比气泡流水线又减少 1/3。对于动态分支预测，其周期数仅 1782，在重定向流水线的基础之上又减少了 1/4。

动态分支预测执行相同的代码所需的时间周期仅为单周期 CPU 的 1/5，CPU 执行效率得到了空前的提升。

4.3 主要故障与调试

4.3.1 单周期数据通路故障

单周期 CPU：调试 benchmark 指令。

故障现象：在执行排序操作时，无法正常向内存写入数据。

原因分析：在数据通路中，内存写入数据直接由寄存器中值决定，不允许直接将立即数写入内存。而在错误电路中（图 4.1），当 ALU_SRCB 为高电平时，内存将直接接受到立即数，这是不允许的。

解决方案：修改数据通路，将内存写入线直接与寄存器输出接口相连，如图 4.2 所示。

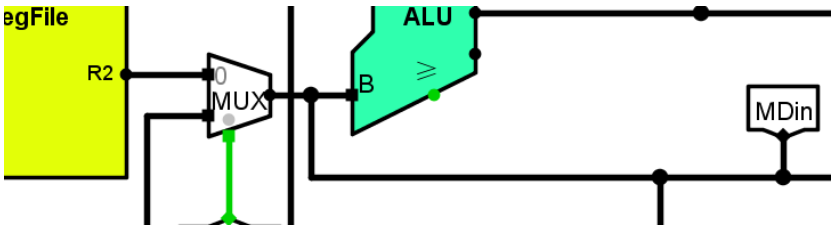

图 4.1 错误电路数据通路

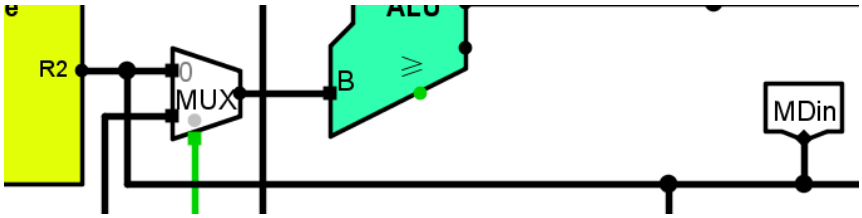

图 4.2 修改后的电路数据通路

4.3.2 LBU 执行错误

单周期 CPU 增加支持 CCAB 指令：实现 LBU 指令。

故障现象：LED 灯输出为 81、85、89，每次直接加 4 而不是加 1。

原因分析：由于内存访问要求按字对齐，寻址线为 2-11 位，而 LBU 指令需要截取其中的某一个字节，因而导致内存数据线引出的永远都是该字的信息，接入 LBU 处理模块的时候处理的都是该字的最低字节而非指定的字节。

解决方案：使用 2 选 4 的多路选择器，将寻址线的 0-1 位用于选取 LBU 处理字节，如图 4.2 所示。

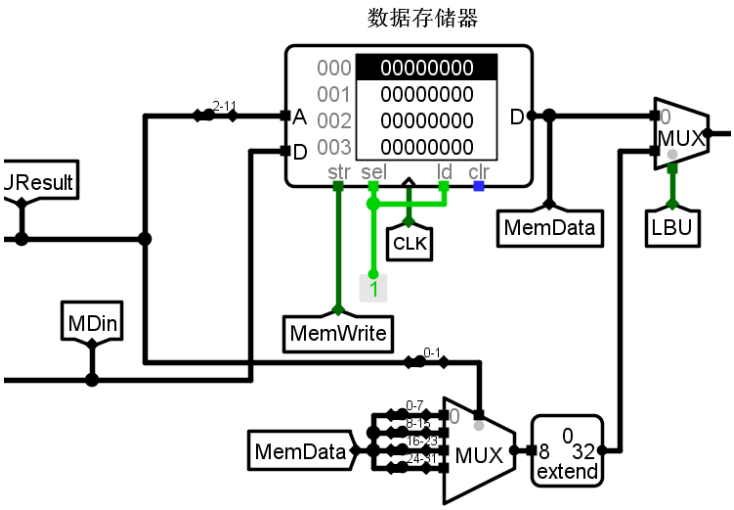

图 4.2 LBU 指令修改方案

4.3.3 动态分支预测故障

实现动态分支预测：预测正确后后续指令未执行。

故障现象：在执行 benchmark 时，执行到右移指令的第三个周期（显示 02000000）时会一直卡顿到下一个测试程序的开始，期间 LED 灯一直未更新。从第三个周期开始，每次都会提示 PredictJump，且预测地址正确。程序一直在被执行。

原因分析：BHT 表给出的预测地址正确且有 PredictJump 说明 BHT 逻辑正确。这里涉及到重定向流水线和动态分支预测在预测情况时的不同——当 BHT 命中且预测正确时，程序应当继续执行，而非清空。对于重定向流水线，遇到分支指令均需要将前两个流水线接口清空。在实际的运行过程中，被清除掉的指令恰好是更新 LED 灯内容的指令，因而 LED 灯会一直卡顿。

解决方案：修改前两个流水线接口清空的条件，当成功预测时不进行清空操作，使其正常执行。

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	学习组成原理 CPU 相关理论知识，阅读课设任务书，阅读 RISC-V 指令手册，并列出 CPU 各部件的数据通路表。
第二天	完成单周期 CPU 的控制信号表，使用 Logisim 搭建控制器，调试单周期 CPU。
第三天	调试成功 Logism 单周期 CPU，并通过头歌测试。
第四天	学习流水线 CPU 结构，尝试使用 Logisim 搭建重定向流水线 CPU。
第五天	调试重定向流水线 CPU 成功并通过头歌测试。学习 Verilog 语言。
第六天	学习气泡流水线 CPU，修改重定向流水线 CPU 得到理想流水线和气泡流水线并均通过头歌测试。完成流水线 CPU 部分原件的 Verilog 代码编写。
第七天	学习 CPU 的中断机制，并尝试使用 Logisim 搭建单级中断。完成流水线 CPU 的 Verilog 代码编写并开始调试。
第八天	学习多级中断。调试成功单级中断和多级中断。继续调试 Verilog 代码。
第九天	学习动态分支预测，并尝试搭建动态分支预测流水线。
第十天	流水线 CPU 通过 Vivado 的模拟仿真，并将 bit 流烧入板中。

5 团队任务

5.1 选题与设计

团队任务选定为贪吃蛇的改编版本：在 32*32 的网格中，没有墙体的限制，遇到边界和蛇体自己会死亡。该团队任务使用支持单级中断的单周期 CPU 在 Logisim 平台上进行实现，用户通过按键和 LED 屏幕进行简单交互，并统计用户操控蛇吃到食物个数进行计分。

5.2 团队任务负责部分

我主要负责汇编代码实现，主要实现方式为通过编写 C 语言源程序然后使用 RISC-V 编译器进行源代码的编译，再使用 rars 工具，转化成可以由 CPU 直接执行的机器代码。

在设计过程中，主要涉及以下几个难点：

1. 如何快速保存地图。在实现上因为 LED 屏幕仅能接受 01 两个状态，因而对于一行 32 个格点的显示可以采用一个 32 位的 word 进行压缩存储，因而整个地图仅需内存中 32 个 word 值即可存储。这样操作也使得每一帧的更新、修改以位运算、访字存字等简单操作实现，每帧更新速度加快。
2. 对外接口显示。在个人任务中的汇编代码通过 a7=34 和 ecall 进行 LED 管的显示，在进行展示的时候我使用类似的思想，使用 a7=34 进行屏幕的展示，a7=98 进行分数（LED 管）的展示。
3. 用户操作。用户操作可以抽象为中断程序，通过中断号来控制蛇头的运行方向。

通过编译器编译出来的代码也存在指令集不兼容的问题，我通过实现简单汇编代码转换器解决了指令集的兼容问题。

5.3 实现效果

实现效果（游戏中途界面）如图 5-1 和图 5-2 所示：

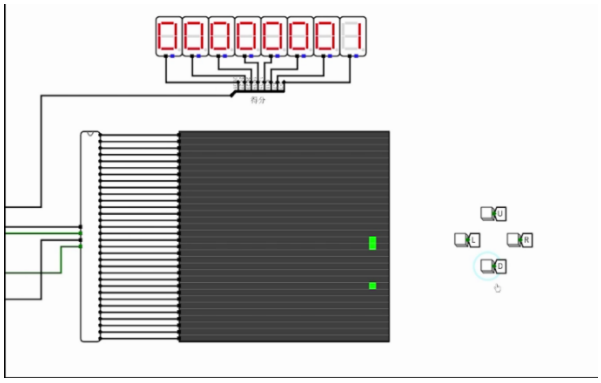

图 5-1 游戏中途界面

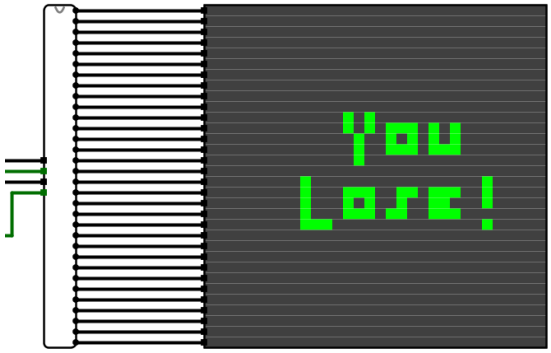

图 5-2 游戏结束界面

6 设计总结与心得

6.1 课设总结

在本次课程设计中，我一共作了如下几点工作：

1） 基础任务：设计了单周期 CPU、理想流水线 CPU、气泡流水线 CPU、重定向流水线 CPU、支持单级中断和多级中断的单周期 CPU，支持单级中断的流水线 CPU，以及动态分支预测的重定向流水线 CPU。

2） 功能总结：所有的 CPU 除理想流水线外，均支持 24 条基本指令和 4 条扩展 CCAB 指令。重定向流水线 CPU 减少了普通流水线 CPU 的气泡插入，加速了 CPU 的执行速度。动态分支预测技术利用 BHT 表进一步加速了重定向流水线 CPU，减少了分支误取的代价。对于多级中断，支持 3 条不同等级中断源的嵌套中断。

3） 附加任务：对于重定向流水线 CPU 进行了上板，并支持多级中断和 BTB 动态分支预测技术。

6.2 课设心得

本次课程设计是我转专业到计科来的第一门课程，除开课程本身的难度，更大的难度在于我还没有先修计算机组成原理和 Verilog 语言的情况下需要独立的完成本次课程设计，因而为了完成本门课程之前还自学了部分组原和 Verilog 知识以完成本次实验。

在实现单周期 CPU 的时候我就遇到了很大的困难——没有一个总体的概念。面对 24 条指令所构成的 24 种通路，最开始的时候很难想到一个很周全的涵盖支持这全部数据通路的布线方法，以及判定走哪一条具体数据通路的方法。在经过了初步的学习之后我大致有了一个基本的框架，通过长时间不断的摸索尝试出了不同数据通路的组合和数据分支方式。其间也因为头歌测试结果看不懂，调试了很长时间。

通过了单周期 CPU 之后，对于流水线式的结构，我又学习和钻研了好久，但是有了单周期的基础这部分掌握的很快。数据重定向的判定利用书上的条件也很快得到了实现。流水线 CPU 的调试工作异常的艰难，因为总线的数目增多，且加入了流水线接口的子电路后，不应交叉的线路之间的交叉会变得非常难找。且气泡的插入错误

也很难直接通过汇编代码进行检查，通常一条气泡的插入错误导致的错误会引发很后面周期的错误，所以流水线 CPU 的调试也是持续了非常长的时间。

对于中断部分，我也是经过了很长时间的学习，大致了解了基本原理后，就直接开始在 logisim 上进行尝试。单级中断由于原理较为简单，因而很快就通过了；多级中断中对于中断是否允许打断这个逻辑就卡住了我，在后面需要支持队列式的中断请求存储的这个操作又琢磨了很长时间。在动态分支预测这里，我去学习了很多的文档以学习 BTH 的实现。在经过不懈的努力后最终实现了动态分支预测。

对于 Verilog 语言，刚刚开始学的时候感觉非常的陌生，但是尝试和上手之后发现和 C 语言非常类似，因而编写程序起来效率很快。在使用轻量级的 Verilog 仿真器的检测之后使用 Vivado 成功上板。

在主体部分完成后，一些周边的电路逻辑例如 Go 和各项指标的计数上也是根据自己理解程度的不断加深，而在不断修正中逐渐明晰。

在整个课程设计的过程中，我对于电路的理解和操控能力相对于刚学完数电时有了极大的提升，并且通过不断的实践也学习到了组原的部分知识，也初步了了解了汇编代码到机器电路级执行这最底层的一步，通过实际的动手收获匪浅。

本课程的时间分配上可以适当的延长与调整：对于刚入手的单周期和流水线可以适当加长时间，而对于中断和动态分支预测等进阶电路，因为有单周期和流水线 CPU 的电路基础，因而速度可以加快。并且对于上板的部分，可以增补一些关于 Verilog 语言的资料和轻量化编译器的提供。Vivado 由于其功能齐全而体量巨大，每次综合模拟所需用时平均在 1 小时以上，非常不利于新手的上手。可以使用轻量化的编译器先进行初步的模拟后，最后使用 Vivado 进行正式上板，以节约时间。

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]谭志虎，秦磊华，吴非，肖亮.计算机组成原理. 北京：人民邮电出版社，2021 年.

[4]谭志虎，秦磊华，胡迪青.计算机组成原理实践教程.北京：清华大学出版社，2018.

[5]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.

[6]张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008 年.

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: 范彦廷