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：	CS2001
学　　号：	U202011641
姓　　名：	刘景宇
电　　话：	15671569229
邮　　件：	2537738252@qq.com

目 录

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

(1) 支持规定的 32 位 RISC-V 指令集（指令集任选），具体见**表 1.1 指令集**；

(2) 在 CCAB 扩展指令集中支持 2 条 C 类运算指令，1 条 M 类存储指令，1 条 B 类分支指令，具体任务每位同学不一样，指令编号详见公文包中的任务分配；

(3) 支持多级嵌套中断，利用中断触发扩展指令集测试程序；

(4) 支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

(5) 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；

(6) 能运行教师提供的标准测试程序，并自动统计执行周期数；

(7) 能自动统计各类无条件分支指令数目，条件分支成功次数、插入气泡 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	指令类型	简单功能描述	备注
1	ADD	R	加法	指令格式与功能请参考 RISC-V32 指令集英文手册，或参考 RARS 模拟器
2	ADDI	I	立即数加	
3	AND	R	与	
4	ANDI	I	立即数与	
5	SLLI	I	逻辑左移	
6	SRAI	I	算术右移	
7	SRLI	I	逻辑右移	
8	SUB	R	减法	
9	OR	R	或	
10	ORI	I	立即数或	

#	指令助记符	指令类型	简单功能描述	备注
11	XORI	I	或非/立即数异或	
12	LW	I	加载字	
13	SW	S	存字	
14	BEQ	B	相等跳转	
15	BNE	B	不相等跳转	
16	SLT	R	小于置数	
17	SLTI	I	小于立即数置数	
18	SLTU	R	小于无符号数置数	
19	JAL	J	转到并链接	
20	JALR	J	转移到指定寄存器	
21	ECALL	I	系统调用	If ($a7==34) LED 输出$a0 的值 else 等待 Go 按键暂停
22	CSRRSI	I	访问 CSR 寄存器	中断相关，可简化为开中断
23	CSRRCI	I	访问 CSR 寄存器	中断相关，可简化为关中断
24	URET	I	中断返回	清中断，mEPC 送 PC，开中断
25	SLL	R	逻辑左移	指令格式与功能请参考 RISC-V32 指令集英文手册，或 参考 RARS 模拟器
26	XOR	R	异或	
27	SH	S	存半字	
28	BLT	B	小于跳转	

2 总体方案设计

2.1 单周期 CPU 设计

单周期处理器是指所有指令均在一个时钟周期内完成的处理器。尽管不同指令执行时间不同，但对单周期处理器而言，时钟周期必须设计成对所有指令都等长。一条指令执行过程中数据通路的任何资源都不能被重复使用，因此需要被多次使用的资源（如加法器）都需要设置多个，否则就会发生资源冲突。取指令和执行指令的阶段均需要使用存储器，所以单周期处理器只能采用指令存储器和数据存储器相分离的结构（哈佛结构）。在实现过程中，采用 Logisim 与 FPGA 两种方式。

总体结构图如**图 2.1** 所示。

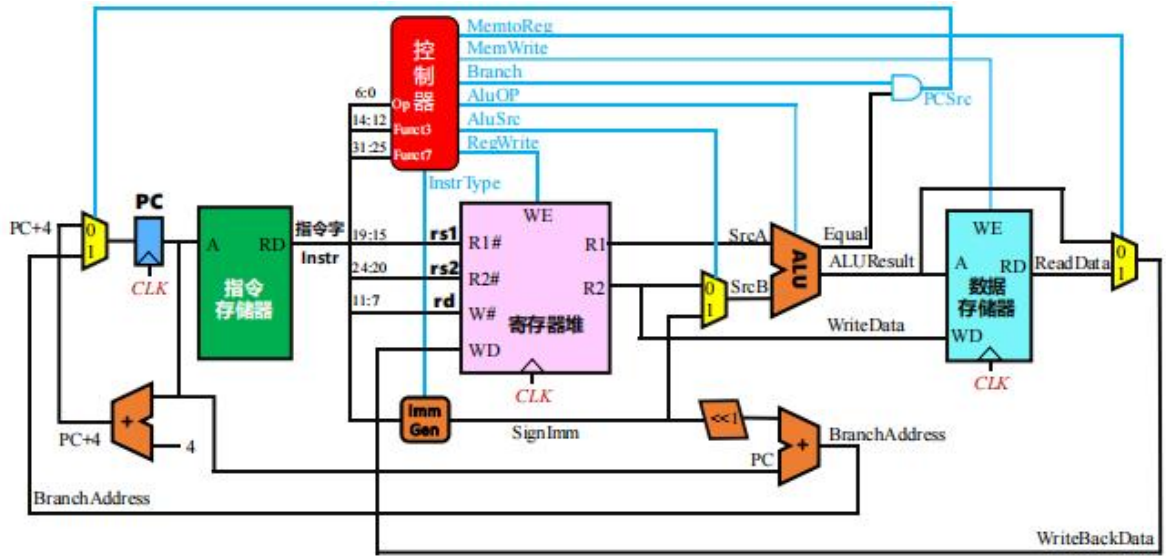


图 2.1 总体结构图

2.1.1 主要功能部件

1. 程序计数器 PC

程序计数器锁存地址对应的机器码，并能在时钟上跳沿来到时，根据跳转信号等决定是实现 PC+4 还是跳转，程序计数器 PC 可以采用寄存器实现，其输出作为指令存储器 IM 的地址输入。

2. 指令存储器 IM

指令存储器的输入是 PC 锁存的地址，为 32 位的字节地址，指令存储器存储一段 RISC-V 汇编代码对应的机器码，在程序执行中不可更改，所以采用 ROM 进行实现。其输出是一条指令的 32 位机器码，所以数据位宽是 32 位。

3. 运算器 ALU

运算器主要用来完成不同的运算，比如算术运算、移位运算、比较等。输入两个 32 位数据，由 ALU_OP 信号来决定是运算的类型，输出运算结果。算术逻辑运算单元引脚与功能描述如**表 2.1** 所示。

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
A	输入	32	操作数 A
B	输入	32	操作数 B
ALU_OP	输入	4	运算器功能码，具体功能见下**表 2.2**
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
Equal	输出	1	Equal=(x==y)?1:0，对所有操作有效
<	输出	1	比较运算，对 SLT 类似的指令有效
⩾	输出	1	比较运算，对 SLT 类型的指令有效

表 2.2 运算器规格

ALU_OP	十进制	运算功能
0000	0	Result = X << Y 逻辑左移 (Y 取低五位) Result2=0
0001	1	Result = X >>>Y 算术右移 (Y 取低五位) Result2=0
0010	2	Result = X >> Y 逻辑右移 (Y 取低五位) Result2=0
0011	3	Result = (X * Y)[31:0]; Result2 = (X * Y)[63:32] 无符号乘法
0100	4	Result = X/Y; Result2 = X%Y 无符号除法
0101	5	Result = X + Y (Set OF/UOF)

ALU_OP	十进制	运算功能
0110	6	Result = X - Y (Set OF/UOF)
0111	7	Result = X & Y 按位与
1000	8	Result = X | Y 按位或
1001	9	Result = X⊕Y 按位异或
1010	10	Result = ~(X |Y) 按位或非
1011	11	Result = (X < Y) ? 1 : 0 符号比较
1100	12	Result = (X < Y) ? 1 : 0 无符号比较

4. 寄存器堆 RF

RISC-V 中大多数指令都和寄存器有联系，包括 32 个寄存器，其中 0 号寄存器始终为 0。从指令的机器码中解析出 rs1，rs2，rd，从寄存器堆中读到 R[rs1]，R[rs2]，由 RegWrite 信号控制是否将运算结果写回，当时钟上升沿到达时，将数据写回。

5. 数据存储器 DM

处理器寄存器的个数是有限的，需要一个比较大的空间存储数据，这就是数据存储器。在本次课程设计中，和数据存储器相关的指令有 SW，LW，SH。输入要读取或者要写入的地址，由 MemToReg 信号控制是 ALU 结果还是数据存储器中读到的数据写回，输出写回的数据。

2.1.2 数据通路的设计

对于每一条指令，将其改写成 RTL（Register Transfer Level），忽略控制类信号，仅保留数据类信号，根据 RTL 功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源，指令系统数据通路框架如表 2.3。

表 2.2 指令系统数据通路框架

指令	PC	IM	RF				ALU			DM		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	

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 2.3**。

表 2.3 主控制器控制信号的作用说明

控制信号	取值	说明
RegWrite	1	寄存器写回信号
MemWrite	1	写入内存控制信号，SW、SH 指令，未单独设置 MemRead 信号
AluOP	见**表 2.2**	运算器操作控制符
MemToReg	1	寄存器写入数据来自存储器
S_Type	1	S 型指令译码信号
Alu_SrcB	0	运算器 B 输入选择寄存器
	1	运算器 B 输入选择立即数
JALR	1	JALR 指令译码信号
JAL	1	JAL 指令译码信号
BEQ	1	BEQ 指令译码信号
BNE	1	BNE 指令译码信号
ECALL	1	ECALL 指令译码信号
SH	1	SH 指令译码信号
BLT	1	BLT 指令译码信号

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 2.4 所示。

表 2.4 主控制器控制信号框架

指令	funct7	funct3	OpCode	ALUop	MemtoReg	MemWrite	ALU_Src	RegWrite

ecall	S_Type	BEQ	BNE	JAL	JALR

2.2 中断机制设计

2.2.1 总体设计

本次课程设计中需要为已实现的 24 条指令的单周期 CPU 增加单级中断处理机制，需要支持三个外部按键中断。该 CPU 支持 3 个 Logisim 按钮触发的中断源，分别对应编号为 1、2、3 的三个按钮，3 个 LED 指示灯 W1、W2、W3 分别表示对应中断源的存在中断请求，中断处理完成时 LED 熄灭，中断优先级 1＜2＜3，CPU 执行中断服务程序时不能被其他中断请求中断。

实现的多重中断处理机制，要求支持 3 个外部按键中断源。CPU 支持 3 个 Logisim 按钮触发的中断源，分别对应编号为 1、2、3 的三个按钮，3 个 LED 指示灯 W1、W2、W3 分别表示对应中断源的存在中断请求，中断处理完成时 LED 熄灭，中断优先级 1＜2＜3，高优先级中断应该正确中断低优先级中断服务子程序。

2.2.2 硬件设计

1）增加中断按键信号采样电路，输出与中断屏蔽位进行逻辑与后送中断优先编码器，同步清零信号用于清除中断请求信号，中断请求信号必须等待终端服务注意中断请求信号必须等待中断服务程序执行到中断返回时才能清除，中断等待指示 LED 用于指示当前中断请求，中断服务程序返回时应熄灭。

2) 实现与中断相关的寄存器，包括中断使能寄存器 IE、异常程序计数器 mEPC。IE 用于开关中断，1 表示开中断，0 表是关中断，开关中断采用同步置位和复位方式。mEPC 用于存放中断程序返回地址，在中断响应阶段硬件会自动将主程序 PC 值送 mEPC 保存。

3)设计中断识别逻辑，能实现实验要求的中断响应优先级，能正确识别 1~3 号中断源，并设计向量中断机制，可由中断号寻找中断程序入口地址，为简化实现，中断向量表可以直接用硬逻辑实现（中断入口地址固定）。中断识别部分设计可以采用优先编码器实现，由于不需要动态调整中断处理优先级，所以中断屏蔽寄存器

部分电路可以省略。

4)增加中断隐指令数据通路，中断响应周期需要实现硬件关中断、将主程序断点保存至 mEPC 寄存器、将中断识别逻辑产生中断服务程序入口地址送 PC。

5) 增加 uret 指令数据通路，本实验只需增加 uret 的数据通路即可。这里 uret 指令主要功能是将 mEPC 寄存器送 PC，开中断。

6) 增加 csrrsi、csrrci、csrrw 指令数据通路，这 3 条指令主要用于访问 CSR 寄存器，在本实验中用于实现开中断，关中断，访问 mEPC 寄存器

7) 在本实验中采用硬件堆栈保护 mEPC 寄存器，只需要将前 2 条指令分别对应开关中断指令。

2.2.3 软件设计

在本实验中，当产生按键中断时，硬件部分实现了程序的中断，并将地址保存在 mEPC 中，并且由硬件控制中断使能，在硬件部分中实现了 uret 数据通路。完成的中断还需要实现保护现场、恢复现场、中断返回等，这些功能需要结合软件指令来完成。其中保护现场、恢复现场可以配合$sp 寄存器，借助 lw，sw 两条指令来实现。中断返回可以由 uret 指令完成，其中 uret 将 mEPC 寄存器的值送到 PC，开中断，在硬件部分已实现数据通路。

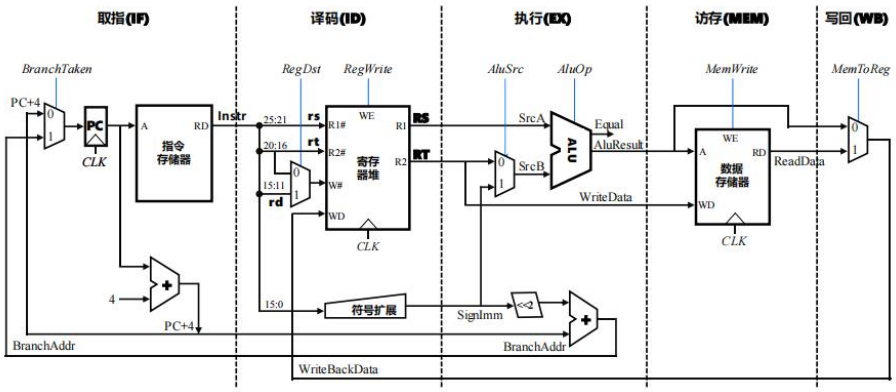
在多重中断中，增加了 csrrsi、csrrci 两个指令，分别对应开关中断。在一个中断处理程序保护现场后开中断，允许高优先级中断，在恢复现场前关中断，不允许被打断。

2.3 流水 CPU 设计

2.3.1 总体设计

将单周期 CPU 修改为流水线 CPU。将指令过程分成 5 个阶段 IF、ID、EX、MEM、WB，分支指令在 EX 段完成，不同阶段之间设置缓冲接口部件，构建各阶段之间的接口部件，接口定义尽可能简化，流水线应向后续段传递数据信息，控制信息，向前段传递反馈信息，后续部件对数据的加工处理依赖于前阶段传递过来的信息。ID 段译码生成该指令的所有控制信号，控制信号将逐段向后传递，后续部件控制信号不再单独生成。IF 段包括程序计数器 PC、指令存储器以及计算下条指令地址逻辑；

ID 段包括操作控制器、取操作数逻辑、立即数符号扩展模块；EX 段主要包括算术逻辑运算单元 ALU、分支地址计算模块；MEM 段主要包括数据存储器读写模块；WB 段主要包括寄存器写入控制模块。

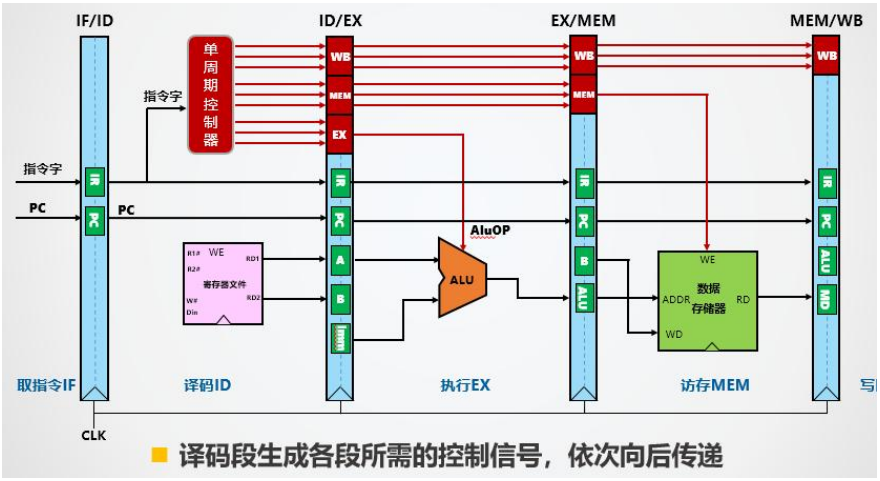


2.3.2 流水接口部件设计

流水线接口部件要能够锁存数据和控制信号，具有使能端，并且需要能够实现同步清零，能够对流水线进行控制。分析每段需要的数据和控制信号，确定流水线接口部件需要锁存哪些数据和控制信号，数据主要包括寄存器的值，立即数的值，ALU 运算结果等，按照不同的需要，可以将控制信号分为 WB、MEM、EX 段。

2.3.3 理想流水线设计

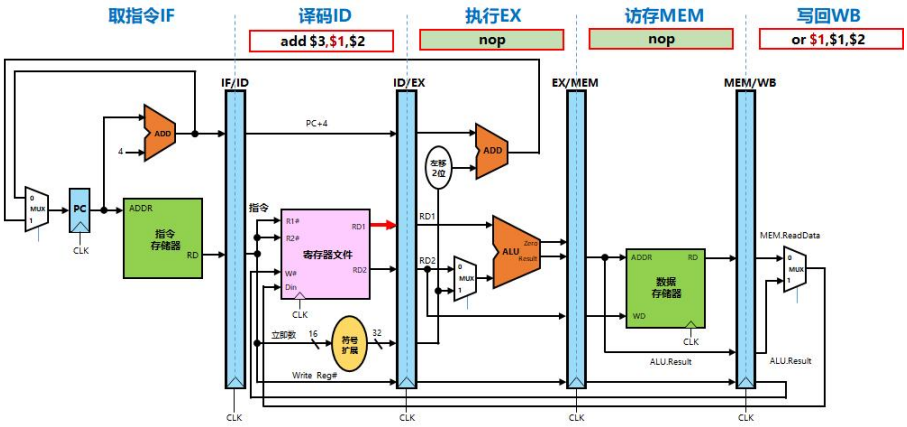
理想流水线不考虑分支跳转、冲突检测，所以理想流水线支持的指令有限，并且对汇编指令间有一定的约束，不能产生数据冲突，在硬件实现方面，理想流水线实现简单，容易由单周期 CPU 修改，IF 段完成取指操作，ID 段完成译码操作，EX 段完成运算操作，MEM 段完成存储操作，WB 完成写回操作，理想流水线总体结构图如图所示。

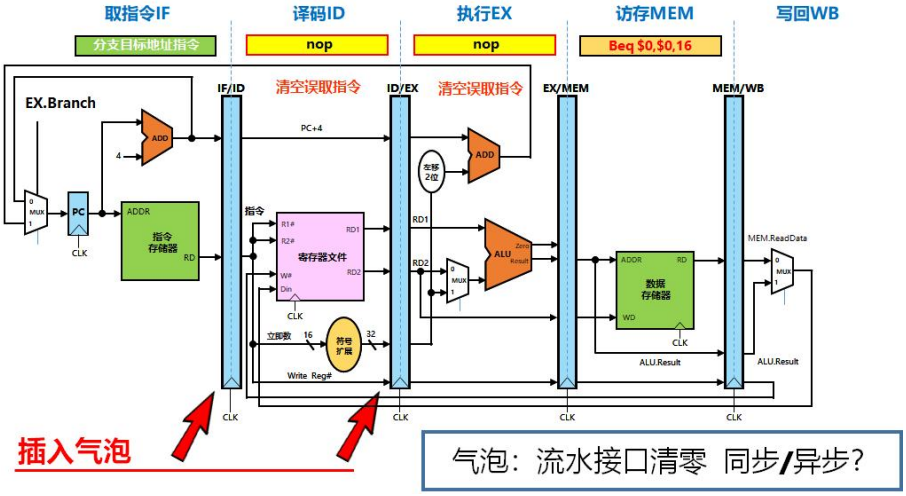


2.4 气泡式流水线设计

理想流水线存在数据冲突以及不支持指令跳转的问题，对其进行改进，增加实现气泡机制，增加流水冲突检测器。

为理想流水线增加冲突处理机制，通过先写后读的方式解决寄存器资源冲突，通过插入气泡方式解决数据相关冲突，通过清空误取指令方式解决分支冲突。当产生数据冲突时，通过流水线接口部件的使能信号暂停，当时钟来临时产生，产生气泡，待数据冲突解决后恢复使能信号，继续执行，从而解决了数据冲突。当 EX 段产生跳转时，利用流水接口部件的清零信号，将前面的接口部件清零，按照新的地址取值，产生了相应的气泡。对于解决数据冲突的总体结构图如图，对于解决分支跳转的总体结构如图。





为了实现正确的数据冲突检测，产生合适的气泡，需要添加数据相关检测与处理逻辑，流水线中的数据相关检测逻辑如下：

```
DataHazzard = RsUsed & (rs≠0) & EX.RegWrite & (rs==EX.WriteReg#)
+ RtUsed & (rt≠0) & EX.RegWrite & (rt==EX.WriteReg#)
+ RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#)
+ RtUsed & (rt≠0) & MEM.RegWrite & (rt==MEM.WriteReg#)
```

2.5 数据转发流水线设计

气泡流水线通过延缓 ID 段取操作数动作的方式解决数据冲突问题，但大量气泡的插入会严重影响指令流水线性能。通过修改气泡流水线，将数据相关处理方式修改成数据重定向。可以直接将正确的操作数从其所在位置重定向（Forwarding）到 EX 段合适的位置，如果相邻两条指令存在数据相关，且前一条指令是访存指令时（称为 Load-Use 相关），这种数据相关并不能采用重定向方式进行处理，LoadUse 相关逻辑如下：

```
LoadUse = RsUsed & (rs≠0) & EX.MemRead & (rs==EX.WriteReg#)
+ RtUsed & (rt≠0) & EX.MemRead & (rt==EX.WriteReg#)
```

其他数据相关都可以采用重定向的方式以无阻塞的方式解决，在 ID 段生成两个重定向选择信号 RsFoward，RtFoward 传递给 ID/EX 流水寄存器，相关逻辑如下：

```
IF (RsUsed & (rs≠0) & EX.RegWrite & (rs==EX.WriteReg#))
RsFoward = 2   # ID 段与 EX 段数据相关
else IF (RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#))
```

RsFoward = 1 # ID 段与 MEM 段数据相关

else RsFoward = 0 # 无数据相关

当发生 Load-Used 相关时，需要暂停 IF、ID 段指令执行、并在 EX 段插入气泡，需要控制 PC 使能端 EN、IF/ID 使能端 EN、ID/EX 清零端 CLR，而 EX 段执行分支指令时会清空 ID 段、EX 段中的误取指令，会使用 IF/ID 清零端 CLR、ID/EX 清零端 CLR。综合两部分逻辑，可以得到相关处理逻辑阻塞信号 Stall、清空信号 Flush，各控制端口的逻辑：

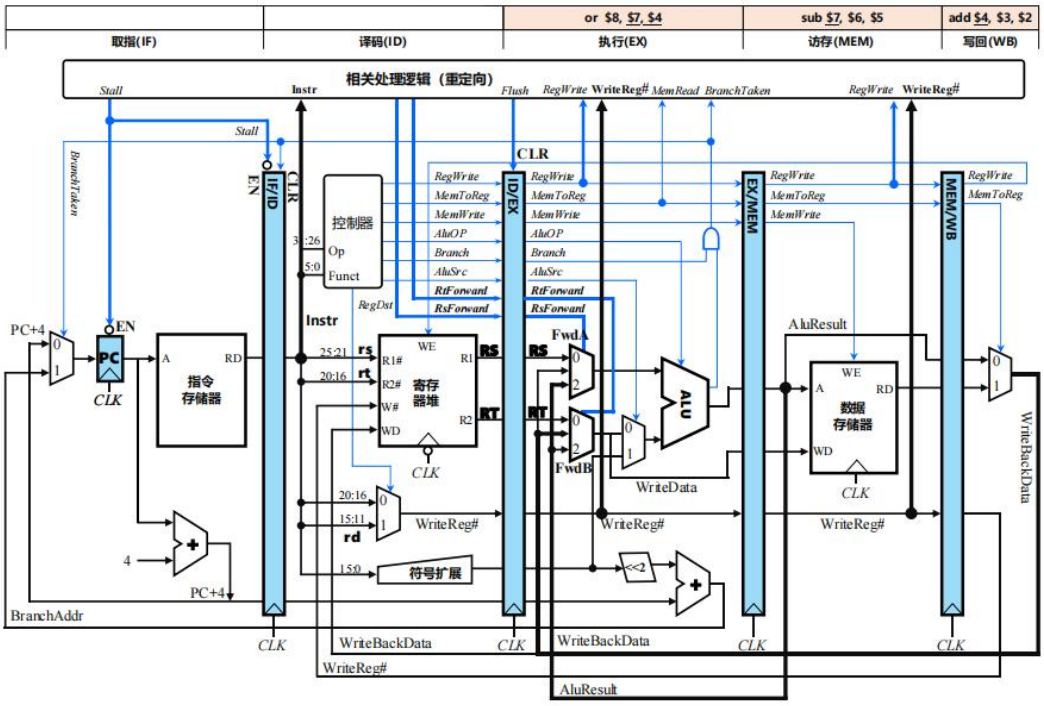
Stall = LoadUse # Load-Use 相关时要暂停 IF、ID 段指令执行

IF/ID.CLR = BranchTaken # 出现分支跳转时要清空 IF/ID

ID/EX.CLR = Flush = BranchTaken + LoadUse # 分支跳转或 Load-Use 相关时要清空 ID/EX

PC.EN = ~Stall # 程序计数器 PC 使能端输入

采用插入重定向方式处理数据相关的处理，总体结构图如下图所示。



2.6 动态分支预测机制

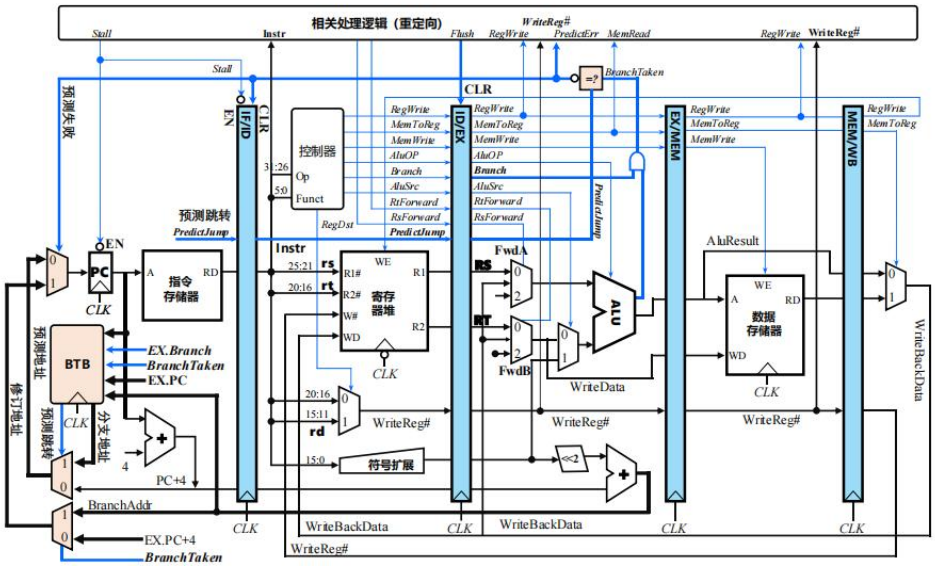
采用重定向机制后，指令流水线中数据相关基本不需要插入气泡就可解决，只有少数 Load-Use 相关还需要插入一个气泡，流水线性能得到极大的提升。此时流

水线中的控制冲突对流水线性能影响最大，基于加快经常性事件的原理，应优先考虑如何减少分支指令引起的分支延迟损失。为减少分支延迟损失，应尽可能提前执行分支指令，比如将分支指令放在 ID 段完成。

分支预测历史位本质上是当前分支指令历史跳转情况的统计信息，是进行分支预测的依据，本次课程设计采用双位预测。当状态位为 00、01 时预测不跳转，为 10、11 时预测跳转，当前分支指令是否发生跳转将会决定状态的变迁。

动态分支预测逻辑必须用硬件实现，内部是一个全相联的 cache 结构，其主要输入输出引脚和功能如表所示。

valid	I/O 类型	位宽	功能说明
CLK	输入	1	时钟控制信号，BTB 表载入新表项或更新预测位需要时钟配合
PC	输入	32	程序计数器地址，是在 BTB 表中全相联查找的关键字
EX.Branch	输入	1	EX 段分支指令译码信号，1 为分支指令，分支指令才会更新 BTB
EX.BranchTaken	输入	1	EX 段分支指令是否跳转，1 为跳转，0 为不跳转
EX.PC	输入	32	EX 段指令对应的 PC 地址
EX.BranchAddr	输入	32	EX 段分支指令的分支目标地址
PredictJump	输出	1	预测跳转信息位，向右依次传输给 EX 段，为 1 表示预测跳转
JumpAddr	输出	32	预测跳转位为 1 时输出 ID 段跳转指令的分支目标地址



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) 程序计数器（PC）

① Logisim 实现：

使用一个 32 位寄存器实现程序计数器 PC，触发方式为上升沿触发，锁存当前执行的指令地址，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，连接在寄存器的使能端，当需要进行停机时，halt 控制信号为 1，经过非门之后为 0，PC 寄存器的使能端为 0，从而使整个电路停机。如**图 3.1** 所示。

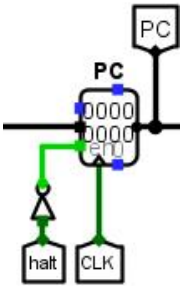


图 3.1 程序计数器（PC）

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
module pc_reg(clk,en,rst_n,pc_new,pc_out);
    input clk;
    input en;                          //停机信号
    input rst_n;                       //置零信号
    input [31:0] pc_new;
    output reg [31:0] pc_out;
    initial pc_out = 0;
    always@(posedge clk or posedge rst_n)
    begin
```

```
            if(rst_n)        //rst 信号优先级比较高
                pc_out<=`zero_word;
            else if(en)      //有使能信号
                pc_out <= pc_new;
        end
    endmodule
```

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，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，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）

② FPGA 实现：

在 Vivado 中使用寄存器组来实现指令存储器，因为 RISC-V 中 1 条指令为 32 位，所以选择 ROM 的数据位宽为 32 位，因为该 ROM 的地址位宽为 10 位，所以选择 ROM 的大小选择为 1024。

指令存储器 IM 的 Verilog 代码如下：

```
module instr_memory (addr, instr);
    input [9:0] addr;
    output [31:0] instr;
    reg [31:0] rom[1023:0];        //指令 ROM
    initial begin
        $readmemh("D:/cpu_verilog/riscv.txt",rom); //从本地读取机器码
    end
```

```
    assign instr = rom[addr];
Endmodule
```

3) 寄存器堆（RF）

① Logism 实现：

CS3410.jar 中提供了一个标准 RISC-V 寄存器文件的库，由美国香奈儿大学开发, 在 Logisim 平台中通过加载 JAR 库的方式加载第三方 JAVA 库。如**图 3.2** 所示。

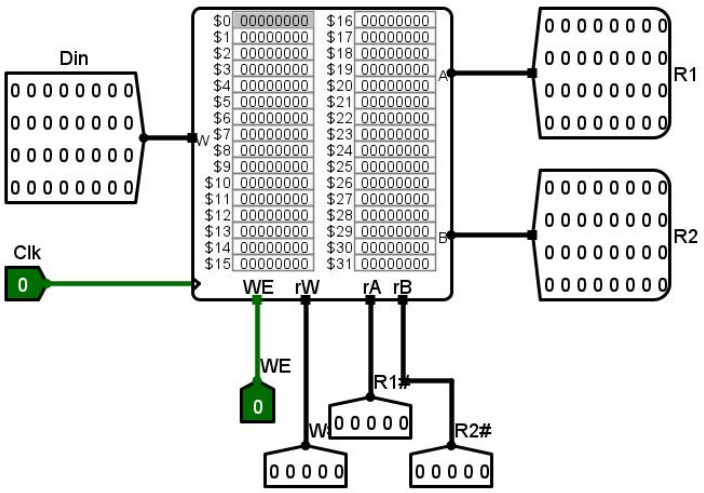


图 3.3 指令存储器（IM）

② FPGA 实现：

在 Vivado 中使用寄存器组来实现寄存器堆，由于 RISC-V 提供了 32 个寄存器，指令解析出来的 rs1，rs2，rd 均为 5 位数据，输出端口会将寄存器堆中对应 rs1，rs2 的数据输出，WE 信号控制是否写回，当 WE 为 1 时并且时钟上升沿到来，会将 Din 的数据写入到寄存器堆中 rd 对应的寄存器中。对于 0 号寄存器单独考虑，保持 0 号寄存器的值始终为 0。

指令存储器 IM 的 Verilog 代码如下：

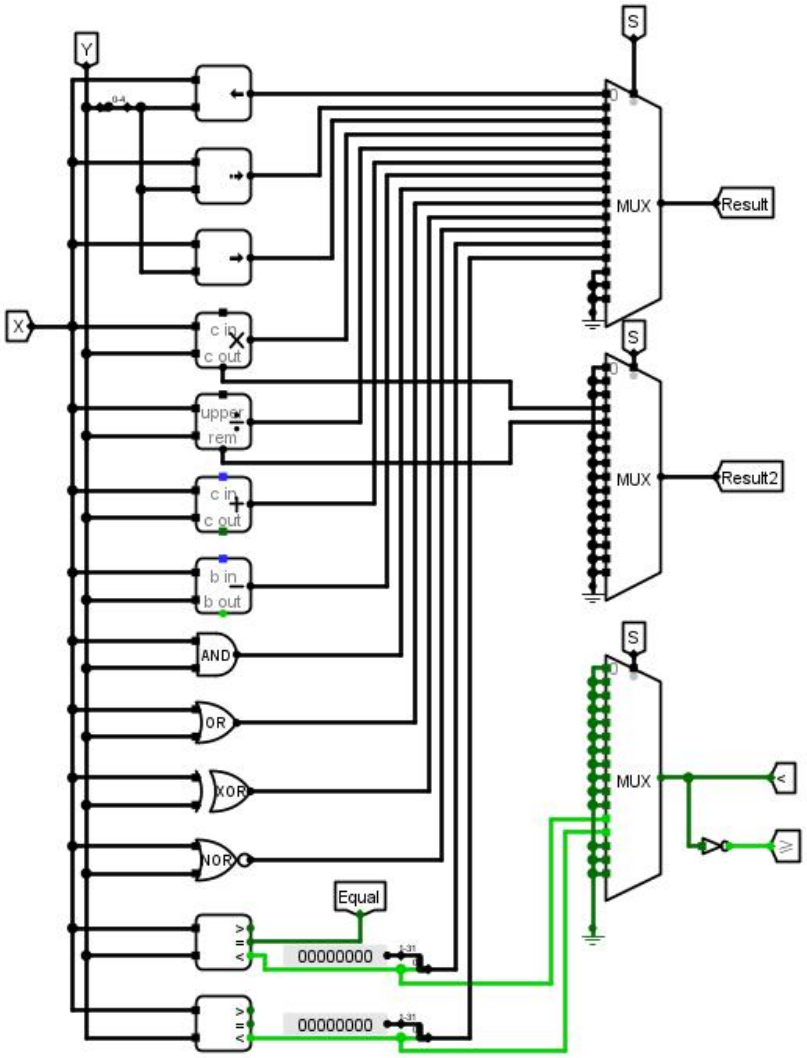
```
module registers(clk, W_en, Rs1, Rs2, Rd, Wr_data, Rd_data1, Rd_data2);
    input clk;
    input W_en;
    input [4:0]Rs1;
    input [4:0]Rs2;
    input [4:0]Rd;
    input [31:0]Wr_data;
```

```
    output [31:0]Rd_data1;
    output [31:0]Rd_data2;
    reg [31:0] regs [31:0];
    always@(posedge clk )//写入数据，0 号寄存器始终为 0
        begin
            if(W_en & (Rd!=0))
                regs[Rd]<=Wr_data;
        end
    assign Rd_data1=(Rs1==5'd0)?`zero_word: regs[Rs1]; //读取寄存器
    assign Rd_data2=(Rs2==5'd0)?`zero_word: regs[Rs2];
endmodule
```

4) 运算器（ALU）

① Logism 实现：

ALU 对输入的两个数据进行运算，根据 ALU_OP 控制信号决定进行运算的类型，并将结果输出，在计算的过程中，如果两个操作数相等，则 Equal 信号输出 1，当指令为 SLT 相关的指令时，根据运算结果给 <，⩾两个信号赋值。对于 ALU_OP 控制信号，使用多路选择进行控制。如图 3.3 所示。



② FPGA 实现：

运算器 ALU 的 Verilog 代码如下：

```
module alu(ALU_DA, ALU_DB, ALU_CTL, ALU_ZERO, ALU_OverFlow, ALU_DC);
        input [31:0]      ALU_DA;
        input [31:0]      ALU_DB;
        input [3:0]       ALU_CTL;
        output            ALU_ZERO;
        output            ALU_OverFlow;
        output reg [31:0]    ALU_DC;       //运算结果
         wire SUBctr;
         wire SIGctr;
```

```
        wire Ovctr;
        wire [1:0] Opctr;
        wire [1:0] Logicctr;
        wire [1:0] Shiftctr;
        assign SUBctr = (~ ALU_CTL[3]   & ~ALU_CTL[2]   & ALU_CTL[1]) |
( ALU_CTL[3]   & ~ALU_CTL[2]);
        assign Opctr = ALU_CTL[3:2];//运算种类选择信号（加法/逻辑/比较/移位）
        assign Ovctr = ALU_CTL[0] & ~ ALU_CTL[3]   & ~ALU_CTL[2] ;//（减法）
        assign SIGctr = ALU_CTL[0]; //小于置一选择（有符号比较还是无符号比较）
        assign Logicctr = ALU_CTL[1:0];     //逻辑运算选择
        assign Shiftctr = ALU_CTL[1:0];     //移位运算选择
        reg [31:0] logic_result;//逻辑运算
        always@(*) begin
            case(Logicctr)
               2'b00:logic_result = ALU_DA & ALU_DB;
               2'b01:logic_result = ALU_DA | ALU_DB;
               2'b10:logic_result = ALU_DA ^ ALU_DB;
               2'b11:logic_result = ~(ALU_DA | ALU_DB);
           endcase
        end
        wire [4:0]   ALU_SHIFT;//移位运算
        wire [31:0] shift_result;
        assign ALU_SHIFT=ALU_DB[4:0];
        Shifter Shifter(.ALU_DA(ALU_DA),
                        .ALU_SHIFT(ALU_SHIFT),
                        .Shiftctr(Shiftctr),
                        .shift_result(shift_result));
        wire [31:0] BIT_M,XOR_M;//加减法运算
        wire ADD_carry,ADD_OverFlow;
        wire [31:0] ADD_result;
```

```
    assign BIT_M={32{SUBctr}};
    assign XOR_M=BIT_M^ALU_DB;
    Adder Adder(.A(ALU_DA),
                .B(XOR_M),
                .Cin(SUBctr),
                .ALU_CTL(ALU_CTL),
                .ADD_carry(ADD_carry),
                .ADD_OverFlow(ADD_OverFlow),
                .ADD_zero(ALU_ZERO),
                .ADD_result(ADD_result));
    assign ALU_OverFlow = ADD_OverFlow & Ovctr;//
    wire [31:0] SLT_result;//小于置一运算
    wire LESS_M1,LESS_M2,LESS_S,SLT_M;
    assign LESS_M1 = ADD_carry ^ SUBctr;
    assign LESS_M2 = ADD_OverFlow ^ ADD_result[31];
    assign LESS_S = (SIGctr==1'b0)?LESS_M1:LESS_M2;
    assign SLT_result = (LESS_S)?32'h00000001:32'h00000000;
    always @(*)   //选择运算结果
    begin
        case(Opctr)
            2'b00:ALU_DC<=ADD_result;
            2'b01:ALU_DC<=logic_result;
            2'b10:ALU_DC<=SLT_result;
            2'b11:ALU_DC<=shift_result;
        endcase
    end
endmodule
```

5) 数据存储器（DM）

① Logism 实现：

数据存储器不同于指令存储器，数据存储器需要具备可读可写的需求，RAM

可以实现数据的读写，对于 RAM 来说，当 str 信号为 1 且时钟上升沿到达时，将数据写入到数据存储器中，当 str 信号不为 0 是，将地址对应的数据输出。MemWrite 信号用来控制是否将数据写入，连接到 RAM 的 str 端，SH 信号用来选片，将输入的 32 位的低 16 位写入到对应的位置，为了方便 SH 信号选片，使用多个 RAM。如**图 3.2** 所示。

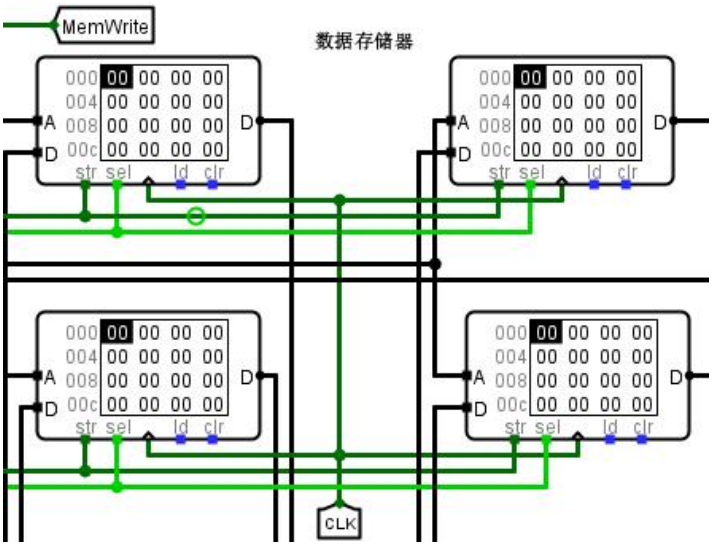


图 3.4 指令存储器（IM）

② FPGA 实现：

数据存储器 DM 的 Verilog 代码如下：

```
module data_memory(clk, W_en, addr, SH, din, dout);
    input clk,W_en;
    input [11:0]addr;                    //输入地址
    input SH;          //写类型，半字还是字，这里主要区分是 sw 还是 sh
    input [31:0]din;                     //写入数据
    output [31:0]dout;                   //读出数据
    reg [31:0]ram[1023:0];               //RAM;
    wire [31:0]Rd_data;
    wire [31:0]Wr_data_H;                //半字拼接
    wire [31:0]Wr_data;
    integer i;
    initial begin for (i=0; i<1024; i=i+1) ram[i] = 0; end
    assign Rd_data=ram[addr[11:2]];
```

```
        assign Wr_data_H=(addr[1]) ?{din[15:0],Rd_data[15:0]} : {Rd_data[31:16],
din[15:0]};
       //根据写类型，选择写入的数据，sw 和 sh，sw 写入一字节，sh 写入半个字
        assign Wr_data=(SH==2'b0) ? din : Wr_data_H;
         always@(posedge clk) //上升沿写入数据
         begin
            if(W_en)
                ram[addr[11:2]]<=Wr_data;
         end
         assign dout=Rd_data;
    endmodule
```

3.1.2 数据通路的实现

本次课程设计采用的工程化的设计模式，一次性构建所有的数据通路。主要实现方法为，对于每一条指令，将其改写成 RTL（Register Transfer Level），忽略控制类信号，仅保留数据类信号，根据 RTL 功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源。

根据总体方案设计中数据通路设计那一小节的详细内容，具体分析每一条指令在执行过程中各个主要部件的输入和输出端口的连接，完成指令系统数据通路表的填写，如表 3.1 所示。

表 3.1 指令系统数据通路表

指令	PC	IM	RF				ALU			DM		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	
ADD	PC+4	PC	rs1	rs2	rd	alu	r1	r2	5			
SUB	PC+4	PC	rs1	rs2	rd	alu	r1	r2	6			
AND	PC+4	PC	rs1	rs2	rd	alu	r1	r2	7			
OR	PC+4	PC	rs1	rs2	rd	alu	r1	r2	8			
SLT	PC+4	PC	rs1	rs2	rd	alu	r1	r2	11			

指令	PC	IM	RF				ALU			DM		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	
SLTU	PC+4	PC	rs1	rs2	rd	alu	r1	r2	12			
ADDI	PC+4	PC	rs		rd	alu	r1	I	5			
ANDI	PC+4	PC	rs		rd	alu	r1	I	7			
ORI	PC+4	PC	rs		rd	alu	r1	I	8			
XORI	PC+4	PC	rs		rd	alu	r1	I	9			
SLTI	PC+4	PC	rs		rd	alu	r1	I	11			
SLLI	PC+4	PC	rs		rt	alu	r1	I	0			
SRLI	PC+4	PC	rs		rd	alu	r1	I	12			
SRAI	PC+4	PC	rs		rd	alu	r1	I	1			
LW	PC+4	PC	rs1		rd	DM	r1	I	5	r1+I		
SW	PC+4	PC	rs1	rs2			r1	I	5	r1+I	r2	
ECALL	PC+4	PC	rs1	rs2								
BEQ	if (r1 == r2) pc += sext(offset)	PC	rs1	rs2			r1	r2				
BNE	if (r1 ≠ r2) pc += sext(offset)	PC	rs1	rs2			r1	r2				
JAL	pc += sext(offset)	PC			rd							
JALR	pc=(x[rs1]+sext(offset))&~1	PC	rs1		rd							
SLL	PC+4	PC	rs1	rs2	rd	alu	r1	r2	0			
XOR	PC+4	PC	rs1	rs2	rd	alu	r1	r2	9			
SH	PC+4	PC	rs		rd	alu	r1	r2				
BLT	if (r1 < r2) pc += sext(offset)	PC	rs1	rs2			r1	r2	11			

在完成指令系统数据通路表的填写之后，根据列出的数据通路表，进行多指令

数据通路的合并输入数，表，将各个主要功能部件进行连接，根据数据通路合并表的最终结果，对于所有的多输入部件使用多路选择器进行输入选择。最终便可以完成数据通路的搭建。

同时，本次课程设计采用简单迭代法实现单周期处理器。主要实现方法为，先完成一条固定地 R 型指令（如加法指令）的基本数据通路，测试运行通过后再在这个基础上不断增加新的数据通路，支持新的指令，直至所有指令都能正常运行。要支持一条指令的运行必须有取指令逻辑和执行指令逻辑。按照设计取指令数据通路，构建 R 型指令数据通路，构建 I 型运算指令数据通路，构建 S 型指令数据通路，构建 B 型和 J 型跳转指令的数据通路，实现硬布线控制器的顺序逐步实现单周期处理器。

在逐步构建的过程中，在控制器处根据指令功能需要添加控制信号，然后根据指令的功能进行数据通路的连接。实现的数据通路如**图 3.3 单周期 CPU 数据通路（Logism）**所示。

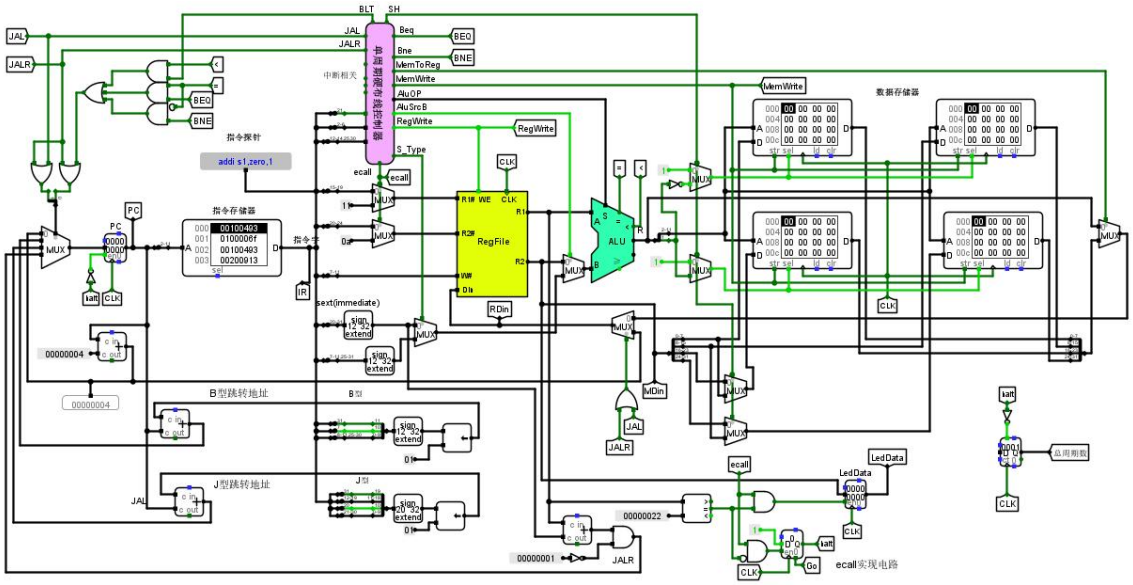


图 3.5 单周期 CPU 数据通路（Logism）

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理，如**图 3.4 单周期 CPU 数据通路（FPGA）**所示。

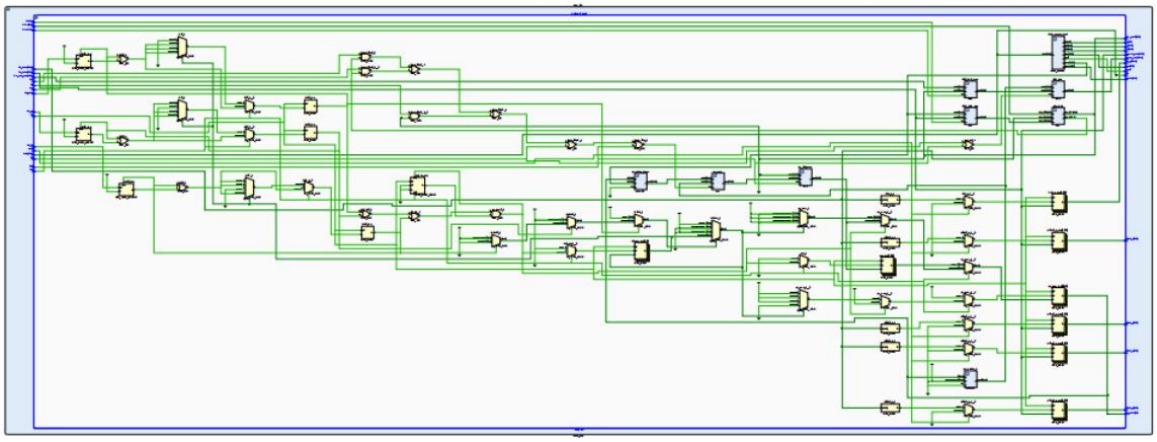


图 3.6 单周期 CPU 数据通路（FPGA）

3.1.3 控制器的实现

① Logisim 实现

对于单周期处理器而言，硬布线控制器就是纯组合逻辑，控制器的功能就是根据指令字中操作码 OP 和 Funct 字段的值直接输出所有操作的控制信号，根据指令功能以及完成的数据通路填写主控制器控制信号，如**表 3.1 主控制器控制信号**所示。

由于硬布线控制器采用组合逻辑实现，可以利用 logisim 的分析电路的功能，根据逻辑表达式自动生成电路，利用工具包里的 EXCEL 表生成的逻辑表达式，生成电路。

表 3.1 主控制器控制信号

指令	funct7	funct3	OpCode	ALUop	MemtoReg	MemWrite	ALU_Src	RegWrite	ecall	S_Type	BEQ	BNE	JAL	JALR
ADD	0	0	c	5				1						
SUB	32	0	c	6				1						
AND	0	7	c	7				1						
OR	0	6	c	8				1						
SLT	0	2	c	11				1						
SLTU	0	3	c	12				1						
ADDI		0	4	5			1	1						
ADDI		7	4	7			1	1						

ORI		6	4	8			1	1						
XORI		4	4	9			1	1						
SLTI		2	4	11			1	1						
SLLI	0	1	4	0			1	1						
SRLI	0	5	4	2			1	1						
SRAI	32	5	4	1			1	1						
LW		2	0		1		1	1						
SW		2	8			1	1			1				
ECALL	0	0	1c						1					
BEQ		0	18								1			
BNE		1	18									1		
JAL			1b					1					1	
JALR		0	1b				1	1						1
CSRRCI		6	1c											
CSRRSI		7	1c											
URET	0	0	1c											
CSRRW		1	1c											
SLL	0	1	c	0			1							
XOR	0	4	c	9			1							
SH		1	8		1	1			1					
BLT		4	18	11										

② FPGA 实现

将各个指令的 opcode、func3、func7 存放在一个文件中，命名为 define.v，

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

```
`include "define.v"
module main_control(opcode, func3, i21, uret, MemtoReg, ALUop, MemWrite, ALUSrc,
```

```
    RegWrite, jal, jalr, beq, bne, blt, SH, ecall);
    input [6:0]opcode;
    input [2:0]func3;
    output [1:0]ALUop;          //确定运算的类型
    output i21, uret, MemtoReg, MemWrite, ALUSrc, RegWrite, jal, jalr, beq, bne, blt,
  SH, ecall;
    wire branch, R_type, I_type, load, store;
    assign uret = ecall & i21;              //符合 ecall 并且指令第 21 位是 1
    assign branch=(opcode==`B_type)?1'b1:1'b0;
    assign R_type=(opcode==`R_type)?1'b1:1'b0;
    assign I_type=(opcode==`I_type)?1'b1:1'b0;
    assign load=(opcode==`load)?1'b1:1'b0;
    assign store=(opcode==`store)?1'b1:1'b0;
    assign ecall=(opcode==`ecall)?1'b1:1'b0;
    assign jal=(opcode==`jal)?1'b1:1'b0;
    assign jalr=(opcode==`jalr)?1'b1:1'b0;
    assign beq= branch & (func3==3'b000);
    assign bne= branch & (func3==3'b001);
    assign blt= branch & (func3==3'b100);
    assign SH=func3[0] & store;              //这里只是用来区分 SW 和 SH
    assign MemWrite= store;
    assign RegWrite= jal| jalr | load | I_type |R_type;
    assign ALUSrc= load | store |I_type | jalr;   //选择立即数传入 ALU
    assign MemtoReg= load;                   //选择 mem 读入到寄存器堆中
    assign ALUop[1]= R_type|branch;          //R 10 I 01 B 11 add 00
    assign ALUop[0]= I_type|branch;
endmodule
```

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 3.5 所示。

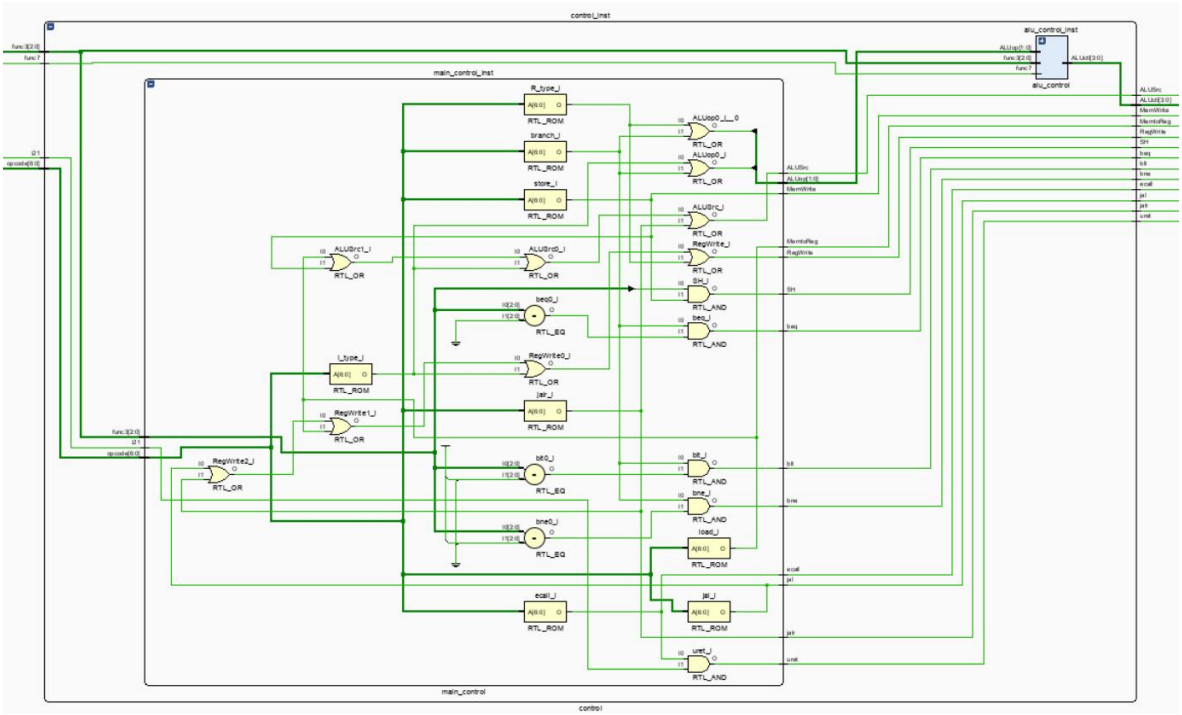
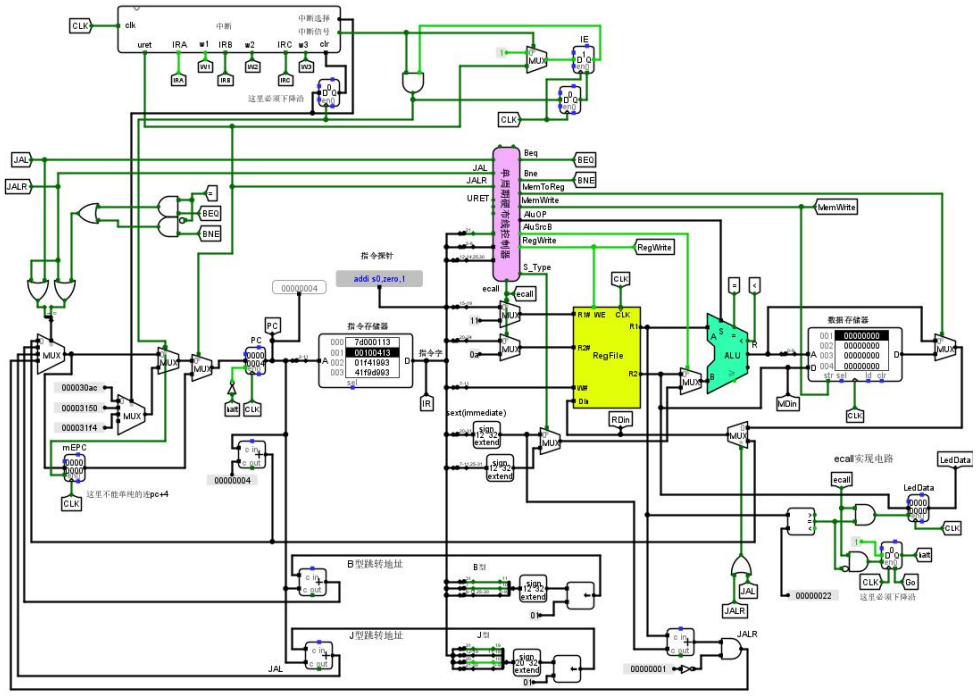


图 3.7 主控制器原理图

3.2 中断机制实现

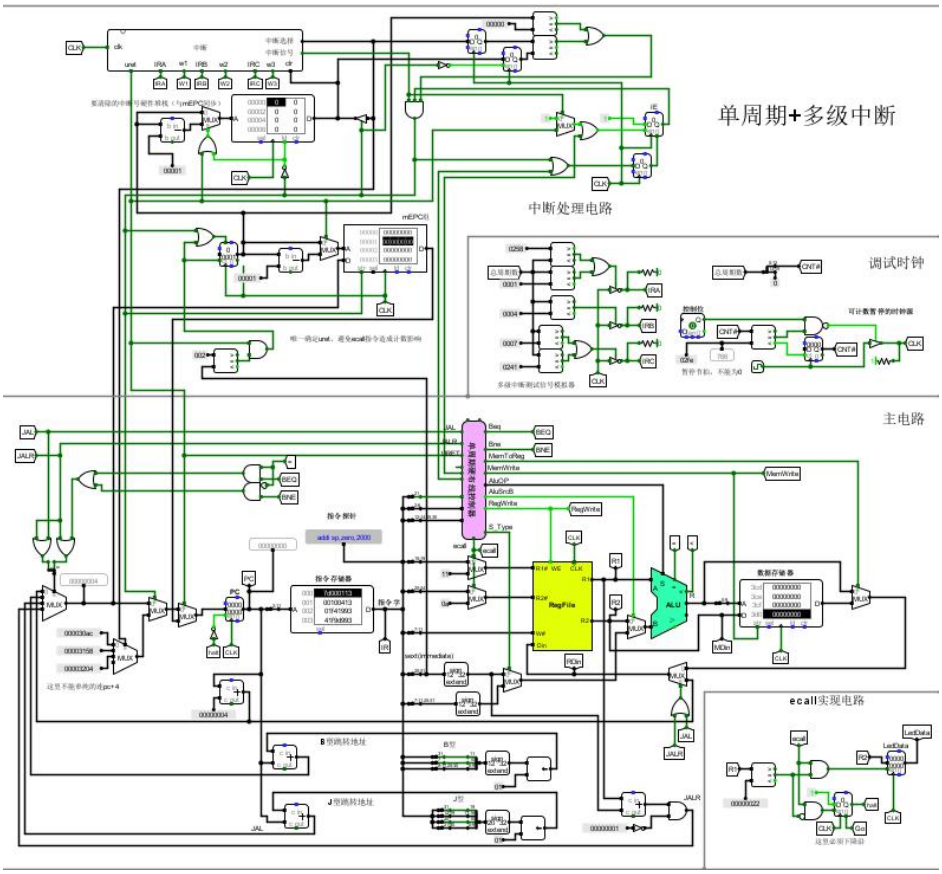
3.2.1 单级中断机制实现

XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)



3.2.2 多级中断机制实现

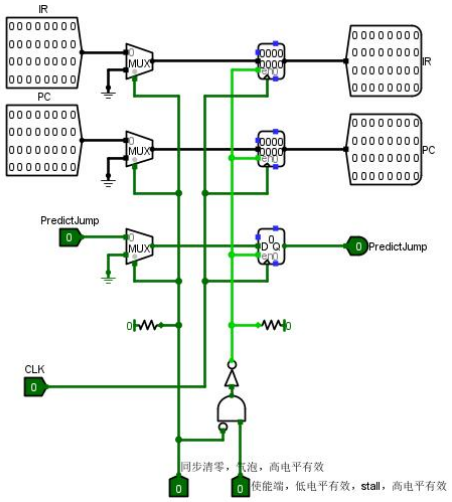
XXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)



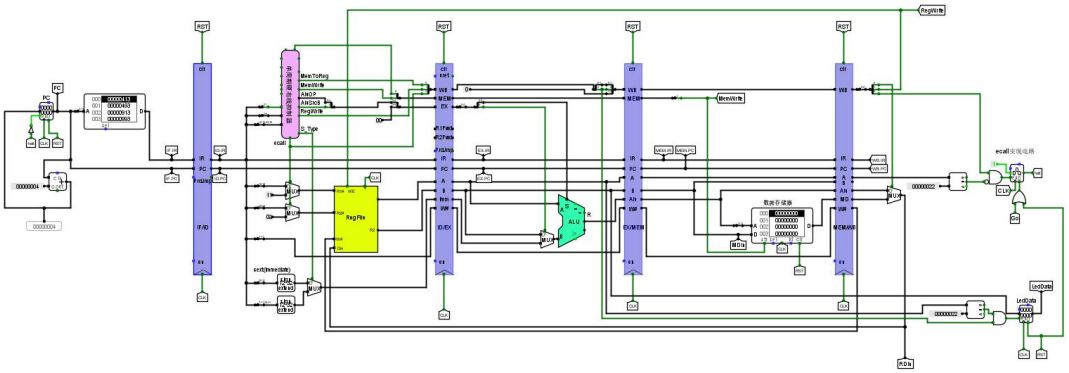
3.3 流水 CPU 实现

3.3.1 流水接口部件实现

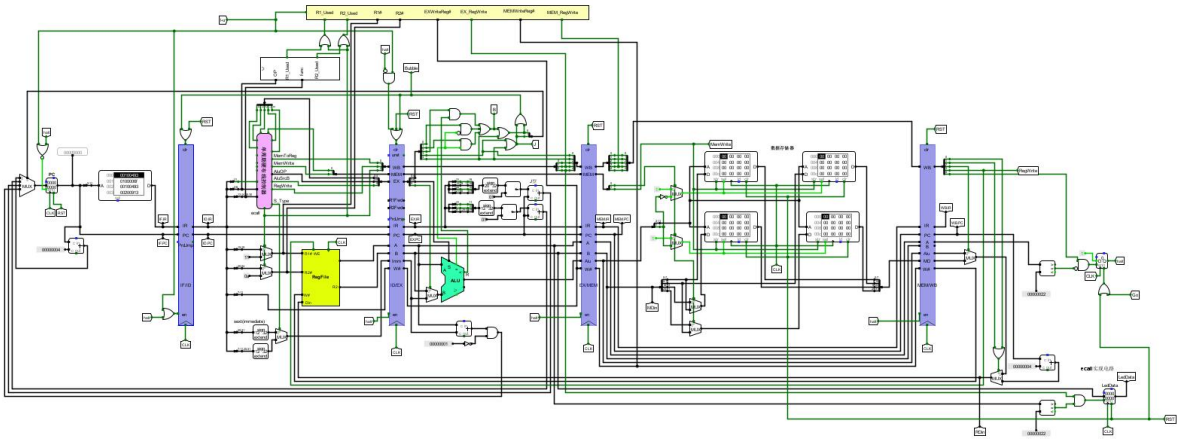
流水线接口部件采用多个寄存器来锁存数据和控制信号，具有使能端，并且需要能够实现同步清零，能够对流水线进行控制。IF/ID 段流水接口如图所示，其余流水接口部件类似。



3.3.2 理想流水线实现

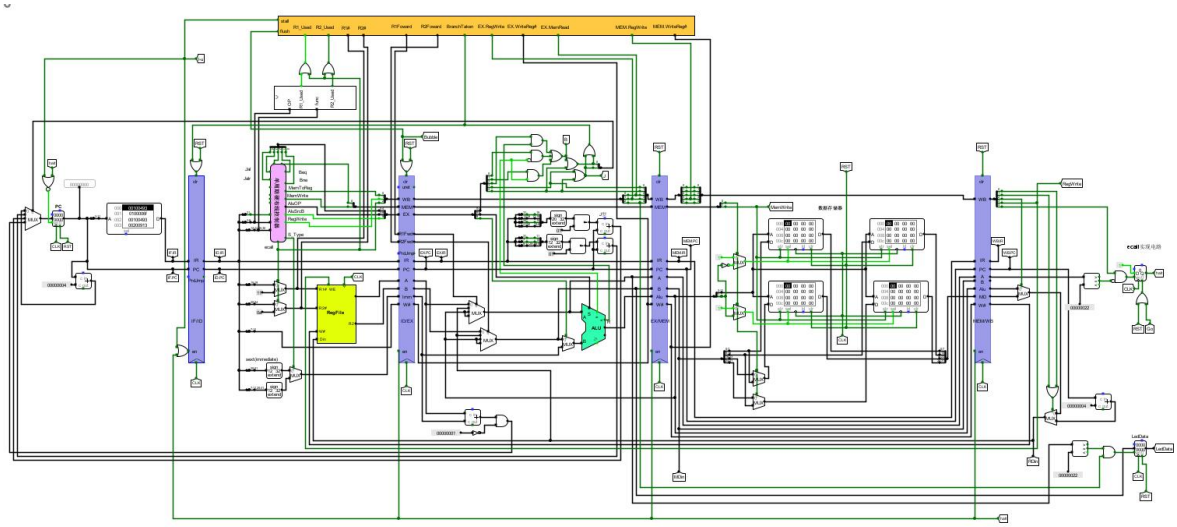


3.4 气泡式流水线实现



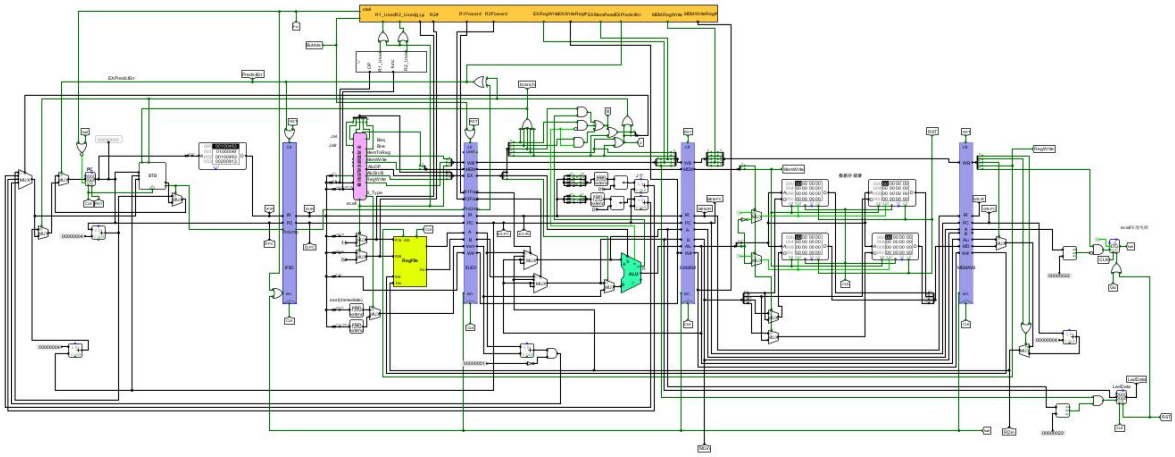
XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)

3.5 数据转发流水线实现



XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)

3.6 动态分支预测机制实现



4 实验过程与调试

4.1 测试用例和功能测试

完成数据通路和控制器后，在指令存储器中载入 risc-v-benchmark.hex 程序，对单周期 CPU 电路进行测试，执行完毕后宗周期数计数为 1546，且 LED 显示应该是 00000038。对于个人的 4 条指令，在 benchmark 中添加 ccab，利用 rars 生成 benchmark+ccab.hex，将指令导入指令存储器，对差异化指令测试，将输出结果与输出汇总表进行比对。对于理想流水线，需要载入理想流水线对应 hex 测试。对于单级中断，多级中断，分别载入实验包中对应的中断测试程序的 hex 文件进行测试。

4.1.1 测试用例 1

在单周期 CPU 中运行 benchmark 程序测试，结果如**图 4.1 单周期 CPU 测试结果**所示。

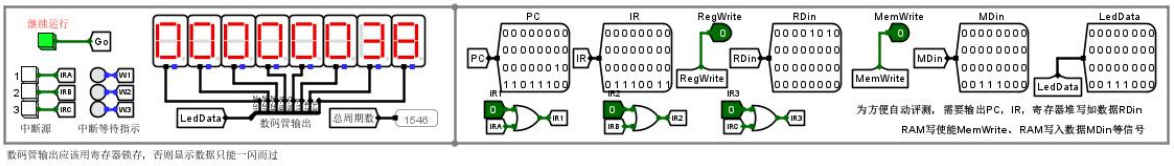


图 4.1 单周期 CPU 测试结果

在气泡流水线中运行 banechmark 测试结果，如**图 4.2 气泡流水线测试结果**所示。

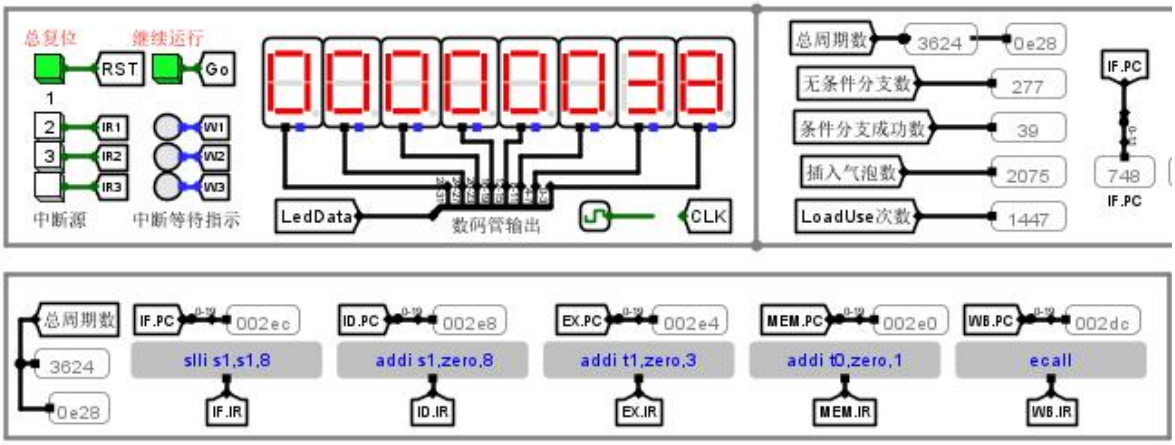


图 4.2 气泡流水线测试结果

在重定向流水线中运行 banechmark 测试结果，如**图 4.3 重定向流水线测试结果**所示。

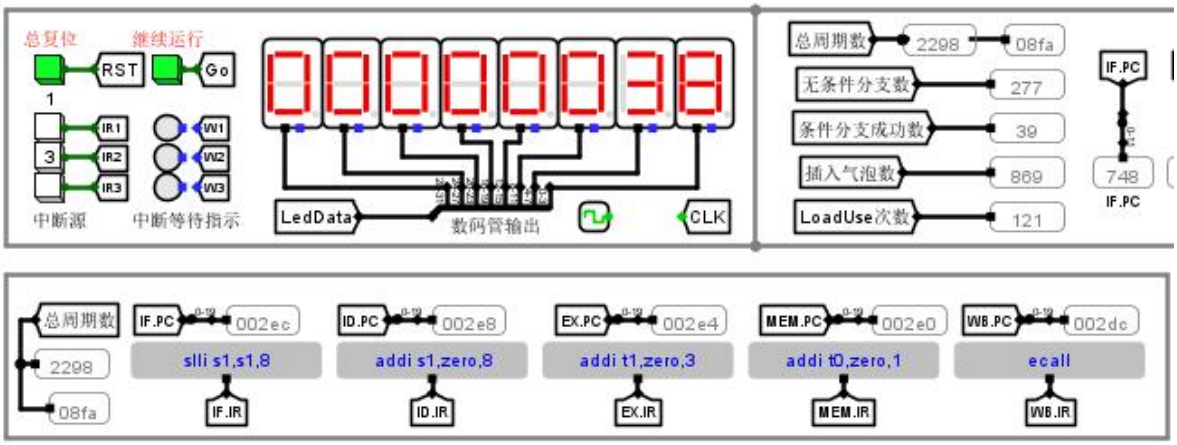


图 4.3 重定向流水线测试结果

在动态分支预测流水线中运行 banechmark 测试结果，如**图 4.4 动态分支预测测试结果**所示。

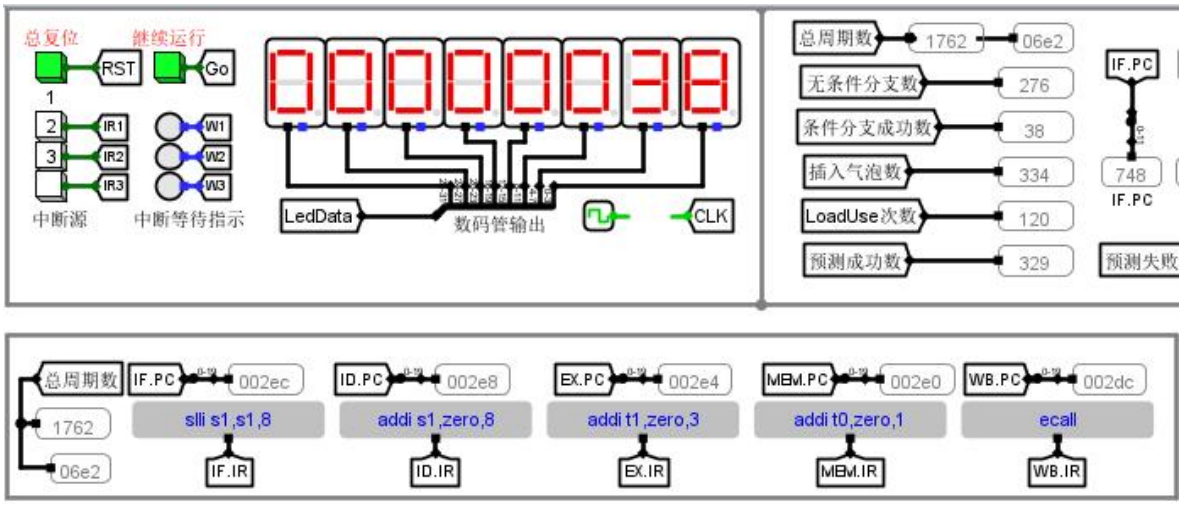


图 4.4 动态分支预测测试结果

流水线中断同样采用 benchmark 测试，观察 LED 灯的变化，与预期结果比对。

4.1.2 测试用例 2

利用头歌平台测试理想流水线，如**图 4.5 理想流水线测试结果**所示。

图 4.5 理想流水线测试结果

4.1.3 测试用例 3

利用头歌平台测试中断电路，单级中断和多级中断的测试结果如**图 4.6 中断电路测试**所示，可以看出完成了中断电路的测试。

图 4.6 中断电路测试

4.2 性能分析

在 4.1 节中可以看出，气泡流水线执行 benchmark 需要 3624 个时钟周期，重定向流水线需要 2298 个时钟周期，而动态分支预测流水线只需要 1762 个时钟周期即可执行完毕，以上结果符合预期。

重定向流水线将气泡流水线中的数据冲突部分进行优化，可以在不阻塞的情况下继续执行指令，降低了因为数据冲突造成的性能损失。

动态分支预测在重定向的基础上，降低了分支跳转造成的性能损失，它根据历史跳转位推算指令地址，降低了同一代码段多次循环造成气泡增多，从而减少了流水线暂停的时钟周期数，增强了流水线的性能。

4.3 主要故障与调试

4.3.1 指令跳转故障

单周期 CPU： 指令跳转错误，跳转位置与预期略有偏差。

故障现象：执行跳转指令出错，跳转到错误的位置。

原因分析：通过单步调试，和头歌预测跳转结果对比，发现指令计算结果错误，重新查阅指令手册中的跳转指令，pc+offset 为跳转指令，在连接时，将 pc+4+offset 传给 pc 寄存器，导致跳转错误。

解决方案：修改计算跳转地址的电路，重新连接，解决跳转出错的问题。

4.3.2 气泡流水线指令执行错误

气泡流水线：添加数据冲突后，部分该产生气泡的位置没有产生气泡。

故障现象：运行中的流水线，执行 ecall 指令后，会直接停机，和预期的 LED 显示不同。

原因分析：通过单步调试，在 ecall 指令执行前后观察寄存器的值以及 ecall 相关的控制信号，发现没有正确的产生气泡，分析是由于未产生数据冲突信号导致，ecall 指令产生了错误的信号，发现没有将 ecall 指令的 r1_used 信号置 1。

解决方案：修改控制信号 Excel 表，重新生成控制器电路，重新测试，可以正确的产生气泡。

4.3.3 动态分支预测周期数大于预期值

动态分支预测流水线：总时钟周期数大于预期值。

故障现象：运行 benchmark 程序，停机时，时钟周期数较大，超过预期值。

原因分析：通过单步调试，观察 BTB 表中各个 cache 行数据的变化。将 BTB 表中的结构与教材中的结构对比，发现实现中对整个的 cache 共用了一个双位跳转预测，而应该对每一个 cache 行都采用一个双位跳转预测。

解决方案：修改 BTB 实现的逻辑，为每一个 cache 行都添加一个双位预测跳转，并重新测试，逐步调试后，动态分支预测周期数与预期值相等。

4.3.4 中断程序处理完之后跳转到错误地址

单级中断：部分中断返回时，会跳转到错误地址。

故障现象：运行 benchmark 程序，发现 LED 灯显示与预期有偏差。

原因分析：通过单步调试，发现跳转的地址有偏差，并且部分中断跳转 EPC 保存的值错误，存入 EPC 的值出现了偏差，观察 EPC 保存的电路逻辑。发现 EPC 存储地址时，单纯的连接了 PC+4，没有考虑到跳转的情况，导致返回到错误的地址。

解决方案：修改 EPC 保存地址的逻辑，修改电路连接，使正确的地址保存到 EPC 中，修改后，中断处理程序能够返回到正确的地址，并通过了测试。

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	复习组成原理 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，并完成数据通路的基本构建。
第二天	完成单周期 CPU 的控制信号表, 使用 Logisim 搭建控制器, 实现了单周期 CPU 并且通过了测试。完成部分 Logism 单周期 CPU 故障报告。
第三天	完成 Logism 单周期 CPU 的故障报告，并且通过了 Logism 单周期 CPU 的检查，并完成流水接口部件的连线，完成 Logisim 平台上的理想流水线，并通过测试。
第四天	在理想流水线的基础上完成气泡流水线的设计，连接，通过头歌平台上的气泡流水线测试。
第五天	在气泡流水线基础上实现重定向流水线，并通过测试，复习动态分支预测相关的理论，初步实现动态分支预测流水线。
第六天	完成动态分支预测的实现与调试，并完成流水线部分的故障报告。
第七天	在 Logisim 中完成单级中断的电路的连接，并通过头歌平台的测试。
第八天	在单级中断的基础上，添加硬件堆栈，完成多级中断的设计与实现，并通过头歌平台的测试。
第九天	使用 Verilog 实现了部分单周期 CPU 的重要部件，并通过仿真检查。
第十天	继续使用 Verilog 进行实现单周期 CPU 的工作，进行数据通路的拼接，生成比特流，上板验证。

5 设计总结与心得

5.1 课设总结

硬件综合训练课程是完成该计算机组成原理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，并利用设计的 CPU 实现一个可演示的团队任务。

作了如下几点工作：

1) 设计了单周期 CPU，实现了支持 24+ccab 指令的单周期 CPU。
2) 设计了理想流水线、气泡流水线、重定向流水线、动态分支预测流水线，并完成了电路的连接，通过了测试。
3) 设计了单级中断和多级中断的电路,实现了程序的单级中断与多级中断。
4) 完成了流水中断的电路的连接。
5) 实现了单周期 CPU 的 FPGA 电路，完成了上板验证
6) 设计了团队任务的顶层视图，FPGA 的 CPU 模块添加了中断处理逻辑，完成了汇编代码的编写，FPGA 电路的连接以及 VGA 显示

5.2 课设心得

本次课程设计不仅工作量大，工作难度也比较大，经历了无数多个 bug，并逐渐调试。虽然说上个学期已经做过 CPU 的实验，但是刚开始拿到一个电路的时候毫无头绪，因为组原实验中只是连接 CPU 中的某一个模块，而课程设计需要连接整个 CPU。为了做成任务，还是硬着头皮对着任务书逐字逐句的读，翻一翻 MOOC 上的课设内容的讲解，一步步连。根据对 CPU 执行指令的理解，填写控制信号的表，开了头之后，连接起来确实快了一点，但是最困难的时候是遇到的小 bug，一点点改，改了一个又冒出来一个。不过慢慢地，等到 CPU 逐渐成形的时候，是满满的成就感，“我能做一个简单 CPU”，这可能是属于计科人特有的成就感吧。

遇到 bug，并修复的过程，对于加深对于 CPU 的理解的有帮助，在单步调试的

过程中，一步步理清数据通路中数据的传输，通过多次的调试，整个电路的功能，各个部件之间的连接，控制信号的作用等几乎了然于胸。

当用 logisim 连接过电路之后，再编写数据通路的 FPGA 代码就会容易很多，对于各主要功能部件的熟悉。尽管上学期选修了 Verilog, 但是只了解了简单的语法，并没有对 vivado 的使用很熟悉，通过本次课程设计，进一步熟悉了 verilog 的语法，尤其是做团队任务中遇到 bug 调试好时。

团队任务中选取了 Flybird，并根据需要设计了整个顶层视图。为了完成功能，需要修改 ecall 指令（系统调用）功能，使得其适配我们的团队任务，为了展示更丰富，学习了 VGA 显示的原理，本来打算要加入键盘中断的，但是由于时间的原因，并且键盘的优先级没有 VGA 优先级高，就把键盘搁置了，采用了开发板自带的按键。

在完成团队任务时，对于 FPGA 实现中断记忆深刻，尤其是中断采样电路，调试这一块调试了太多次了，不是说因为电路逻辑，而是因为 verilog 语法，尽管感觉是对的，但是一综合就错了，但最终还是调好了。

对于本次课程设计，我还有一些建议和改进。感觉使用 logisim 转 FPGA 的文档不详细，转 Verilog 或者写 Verilog 代码比较困难。为了让没选 Verilog 的同学也接触 FPGA，建议在组成原理课程实验中加入一些 FPGA 实现的实验，或者将几个实验改为 FPGA 实现，可以部分参考南京大学组原实验做一些扩展（https://nju-projectn.github.io/dlco-lecture-note/index.html）。这样的话大家可以尽力采用 FPGA 设计 CPU，而不是只依赖 logisim 转 FPGA 完成上板任务，并且大家都能在采用 FPGA 实现的团队任务中都能尽一份力，而不是说不会 Verilog 的同学就只能等着会 Verilog 同学做。

最后在这里也感谢谭老师为我们设计了这样的一个课程设计任务以及详细的文档，将组原实验和课程设计变成闯关游戏。我相信组成原理课程设计必将成为我整个大学生涯中一段无比难忘的回忆。

6. 小兔子乖乖队：使用 FPGA 设计与实现 Flybird

6.1 课设心得

经过个人任务的实现，本队拥有 4 个 logisim 实现的 CPU 以及 1 个 FPGA 实现的 CPU，logisim 工具中可展示的部分有限，而 FPGA 可以利用开发板上的各个接口实现更丰富的功能，可以更好地展示。本队经过讨论，选择做 FlyBird，因为它不需要编写复杂的指令，对于课程设计中的支持 24 条指令的 CPU 友好，并且需要外部按键中断，能够充分展示 CPU 的功能。

nexys4ddr 开发板支持 VGA 显示，板支持的 RGB 格式为 RGB444，对于不同的显示分辨率以及屏幕刷新频率，需要采用不同的时钟频率。由于开发板上的存储空间有限，显示的图片需要占用比较大的空间，所以选择了显示分辨率为 640x480，刷新频率为 60hz，需要给 VGA 提供的时钟频率为 25.17mhz。

由于 flybird 不需要很大的显示空间，按照图 1，将屏幕分为 5 部分，其中中间的 300x300 区域为游戏区域，其余部分为静态展示。

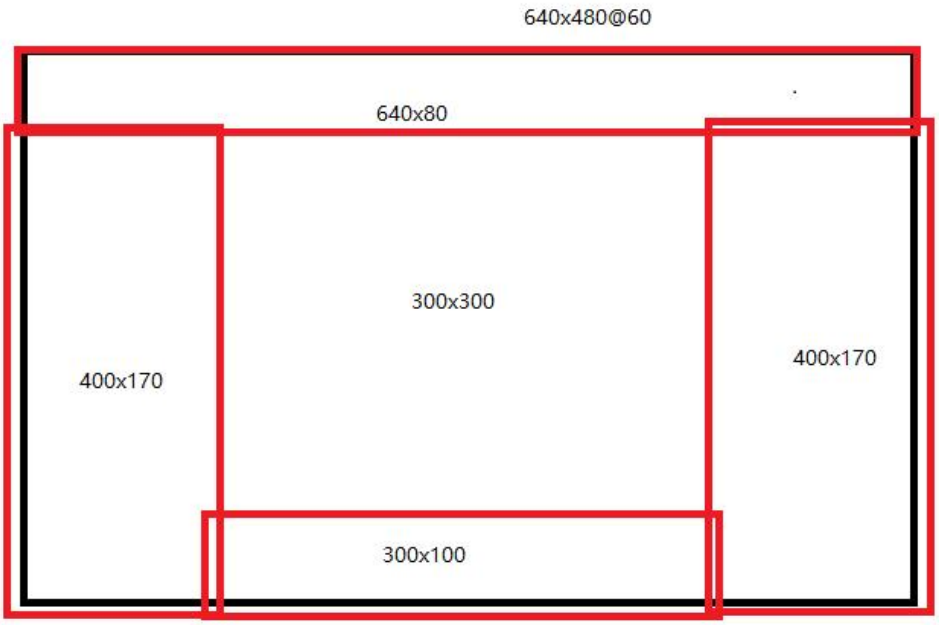


图 1 显示设计图

制作背景图片以及 bird 的图片，并利用 matlab 将图片转换 coe 格式，利可以用 vivado 提供的 IP 核的功能，将 coe 文件存入到 ROM 中。

为了能够实现 VGA 的显示，需要将鸟的纵坐标，第一个柱子的坐标，第二个柱子的坐标存储下来，鸟的横向坐标固定，所以需要 5 个寄存器维护以上信息。由于当前的 CPU 不能直接维护以上 5 个寄存器，要在 VGA 显示和 CPU 之间加一个寄存器组，用来维护信息。VGA 显示直接从寄存器组中并行的读取信息进行显示。而 CPU 则需要传送信息给寄存器组。如图 2 所示。

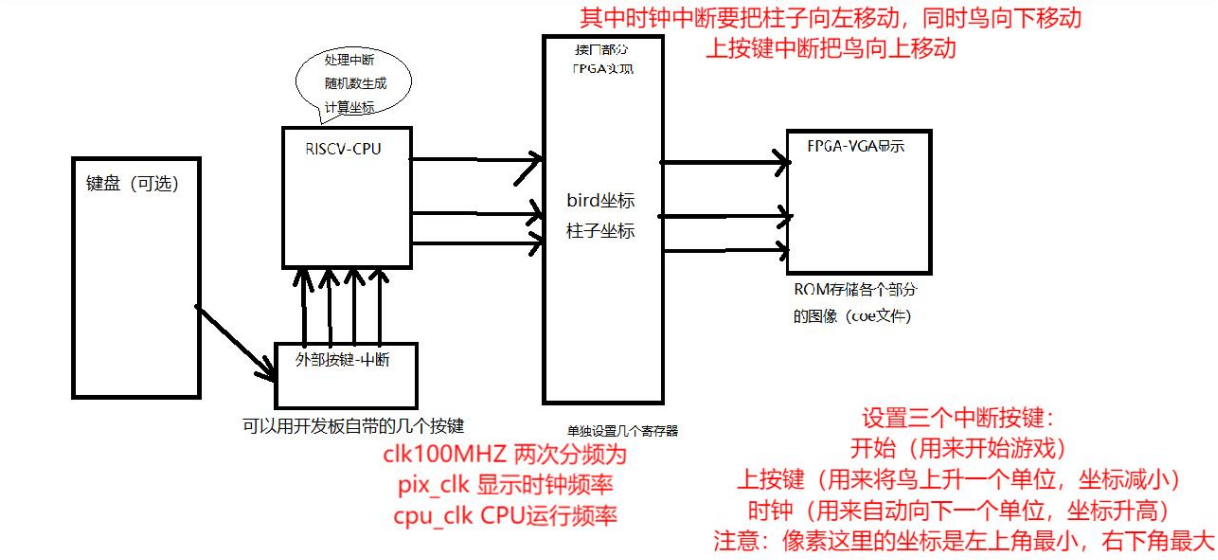


图 2 顶层结构设计图

由于 CPU 无法直接和寄存器组交互，将 CPU 中的 ECALL 指令功能进行改写，根据$a7 的值，将$a0 的值送到寄存器组的某个位置，串行地将数据维护。规定$a7 寄存器的值为 1,2,3,4,5 时，分别将数据送往寄存器组中的对应的五个寄存器。

为了实现交互，还需要实现中断，需要两个按键中断，以及一个时钟中断，包括开始游戏按键，上升按键，以及时钟中断（控制鸟自动下降和柱子自动左移）。将个人任务中 FPGA 实现的 CPU 进行更改，添加中断 uret 指令，添加 EPC、IE 寄存器、中断向量表以及中断采样电路，为了简化电路，采用单级中断实现。

编写汇编代码，主要分为三个中断函数，并且在汇编中采用伪随机数的方式实现柱子的高度变化，为了简化汇编的编写，尽量直接使用寄存器进行运算，为了减少读写数据存储器的次数等进行保护现场、恢复现场的操作，将 x1-x5 寄存器和 VGA 与 CPU 交换数据的寄存器组中的 5 个寄存器对应，其余寄存器可以自由使用。

使用 RARS 将汇编代码转换成机器码，在 vivado 中读取，将以上部分连接起来，

完成整个项目，上板检验成果。

6.2 队员分工

① 刘景宇

实现 VGA 显示模块，添加中断处理逻辑，实现 CPU 与 VGA 数据交换，协助刘从政编写汇编。

②刘从政

编写 flybird 包含中断的汇编代码。

③张传飞

寻找图片，制作要显示的图片，协助张锦程制作展示 ppt。

④张锦程

将图片转换成 coe 格式，负责制作团队任务展示 ppt。

6.3 任务实现与调试

① 设计制作显示的背景图片，得到的图片如下：

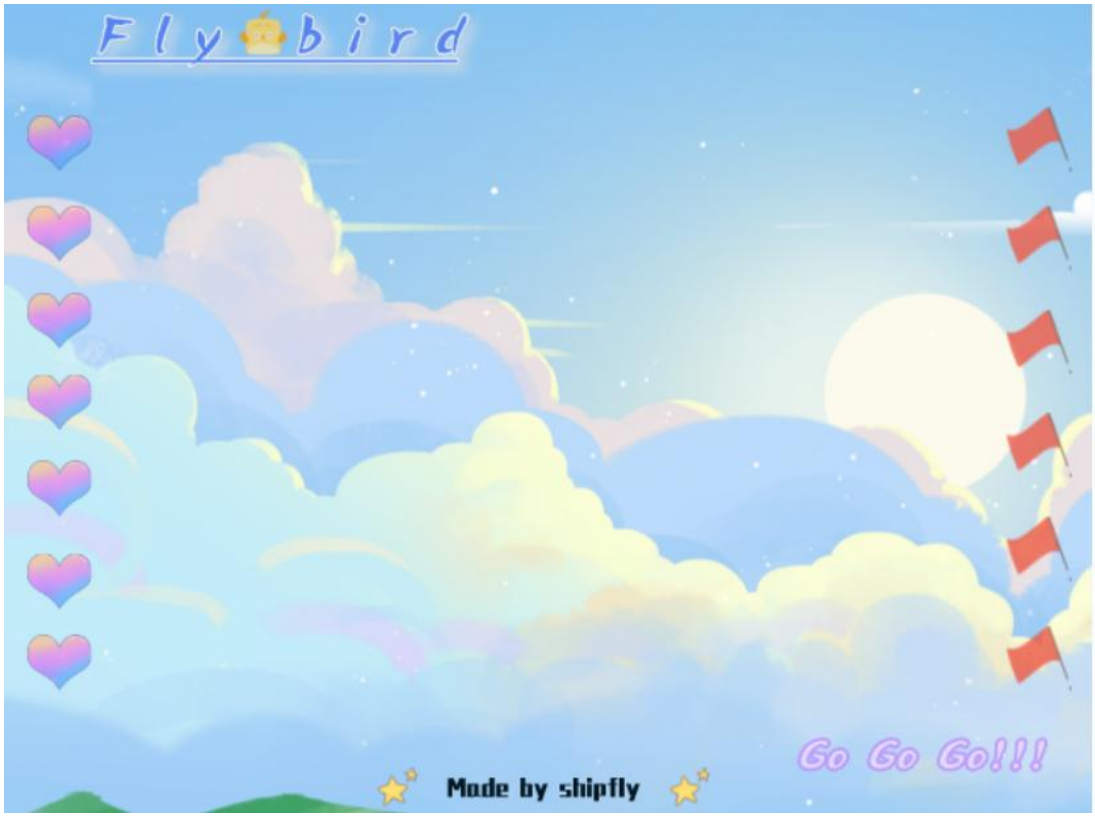


图 3 背景图

② 将转换成 coe 文件，利用 matlab 转换代码，将 png 图片转换成 coe 格式

背景图和小鸟图制作好后，先将背景图和小鸟图分别像素设置为 640x480 和 25x25；再将.png 文件修改文件名转化为.bmp 文件，最后通过 Matlab 代码将.bmp 文件转为 rgb444 彩色图像并制作 coe 文件。

307200 由 640x480 计算得出，此为转化背景图；若要转化小鸟图，则修改为 625 即可。

③ 编写维护小鸟和柱子坐标的汇编代码

主函数初始化各个坐标，然后为一个死循环，直到遇到中断时跳转到相应的中断处理函数，在其他函数中使用 uret 指令完成中断返回。对于柱子的纵坐标设计随机数函数使得坐标在正确范围内伪随机，主程序的汇编代码如下，其余代码放在附件中，其中对 ecall 指令按照以上的规定使用。

```
init:
    addi x1,zero,0      # 初始化，不是中断：开始游戏
    addi x2,zero,150    # bird 的纵坐标
    addi x3,zero,200    # 第一个柱子的横坐标
    addi x4,zero,110    # 第一个柱子显示的位置
    addi x5,zero,380    # 第二个柱子不显示出来（两个柱子始终横向始终差 180）
    addi x6,zero,140
    addi a0,x2,0        # 系统调用,方便 VGA 显示,ecall 根据 a7 的内容，将对应的数值送到对应的位置（系统调用）
    addi a7,zero,1      # bird 的 y
    ecall
    addi a0,x3,0        # 第一个柱子的 x
    addi a7,zero,2
    ecall
    addi a0,x4,0        # 第一个柱子的 y
    addi a7,zero,3
    ecall
    addi a0,x5,0        # 第二个柱子的 x
    addi a7,zero,4
```

```
    ecall
    addi a0,x6,0          # 第二个柱子的 y
    addi a7,zero,5
    ecall
main:
    nop              # 缓冲一下
    j main
```

④ 完成 VGA 显示和 CPU 更改的硬件设计。

新添加的单级中断处理部分的代码如下，其余部分代码放在附件中，其中要特别注意中断采样电路的实现，在调试中经常出现一个错误，在两个 always 中对同一个 wire 进行赋值会报错，尽管他们的触发条件不同（仍可能冲突），就必须进行更改。实现逻辑和个人任务中的 logisim 实现逻辑类似。

```
    reg [31:0] pc_int1;
    reg [31:0] pc_int2;
   reg [31:0] pc_int3;
    reg int1,int2,int3,intA,intB,intC;          //保留当前的中断中断信号
    reg [1:0]intnum;                            //当前中断处于处理的部分
    wire [1:0]intsel;                           //产生的中断中最大的中断号
    reg [31:0]epc ;
    reg int_en ;                               //中断使能信号
    wire int_sig;                                    //表明有中断
     assign     int_sig = int1 | int2 | int3;          //表明当前是否有中断信号
     initial begin
          pc_int1 <= 32'h0000005c;
        pc_int2 <= 32'h00000078;
        pc_int3 <= 32'h00000084;
        epc <= 32'h0;
         int_en <= 1'b1;int1 <= 0;int2<=0;int3<=0; intA<=0;intB<=0;intC<=0;
    end
    always@(posedge btn_start or posedge int1) begin if(int1)intA=0;else intA=1;end
```

```
always@(posedge btn_up or posedge int2) begin if(int2)intB=0;else intB=1;end
always@(posedge clk_int or posedge int3) begin if(int3)intC=0;else intC=1;end
assign intsel=(int1==1) ? 2'b11:((int2==1) ? 2'b10 :(int3==1 ? 2'b01:2'b00 ));
always@(posedge clk)begin              //中断模块，同时替代了 PC 寄存器
    int1=int1|intA;                    //主要起一个同步的作用
     int2=int2|intB;
     int3=int3|intC;
    if(uret)      begin                      //uret 指令,中断返回指令
          pc_new = epc;
          int_en = 1;                  //开中断
          case(intnum)                 //根据产生中断时的内容进行处理
               2'b11: int1<=0;         //同步清零
               2'b10: int2<=0;
               2'b01: int3<=0;
           endcase
          end
    else if(int_sig & int_en)begin     //有中断信号并且处于开中断
           int_en = 0;                 //关中断
           epc = pc_normal;
           case(intsel)                //根据当前的中断选择
               2'b11: begin intnum=2'b11; end      //记录下是谁产生的中断
               2'b10: begin intnum=2'b10; end
               2'b01: begin intnum=2'b01; end
          endcase
          case(intnum)                       //根据当前的中断选择，intnum 中
才是存放的当前是哪个中断
               2'b11: begin pc_new=pc_int1; end
               2'b10: begin pc_new=pc_int2; end
               2'b01: begin pc_new=pc_int3; end
          endcase
```

```
            end
        else pc_new=pc_normal;
    end
    always@(posedge clk)
    begin
        if(ecall)
        begin
            case(Rd_data1)
                32'd1:bird_y <= Rd_data2;
                32'd2:pillar1_x <= Rd_data2;
                32'd3:pillar1_y <= Rd_data2;
                32'd4:pillar2_x <= Rd_data2;
                32'd5:pillar2_y <= Rd_data2;
            endcase
        end
    End
```

对于 VGA 显示模块，参考 VGA 实现原理，将显示模块和自己的功能适配，为了简化实现，将整个图片 coe 使用 IP 核 ROM 存储下来，同时为了避免显示时冲突，设置控制信号，确认当前显示像素点的坐标，从而确定当前显示鸟的 rgb，柱子的 rgb 还是背景的 rgb，按照以上逻辑编写代码，代码放于附件中。

将整个项目整合，文件结构如下图，其中 top 为整个项目，riscv_top 为 cpu 部分，vga_display 为 vga 显示部分，顶部视图如图 4 所示。

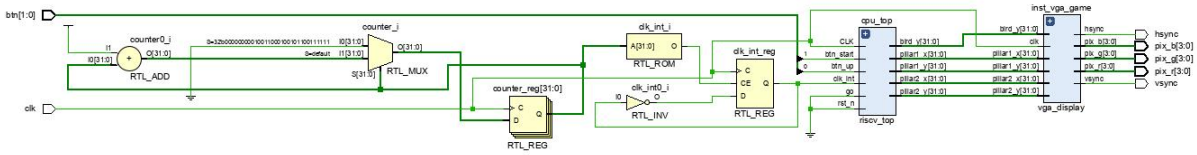


图 4 顶部视图

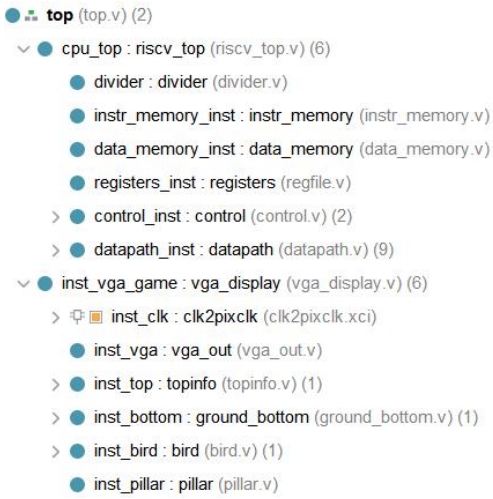


图 5 顶层文件

⑤ 效果图展示。

图 6 效果图展示

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：刘景宇