流水线 MIPS 处理器设计

张智帅
（无 78 2016010483）

目录

0 课程任务和实验要求

0.1 课程任务

将春季学期实验四设计的单周期 MIPS 处理器改进为流水线结构，并利用此处理器完成数据排序。

0.2 实验要求

1. 设计一个 5 级流水线的 MIPS 处理器，采用如下方法解决竞争问题：

i. 采用完全的 forwarding 电路解决数据关联问题。

ii. 对于 Load-use 类竞争采取阻塞一个周期+Forwarding 的方法解决

iii. 对于分支指令在 EX 阶段判断（提前判断也可以），在分支发生时刻取消 ID 和 IF 阶段的两条指令。

iv. 对于 J 类指令在 ID 阶段判断，并取消 IF 阶段指令。

2. 分支和跳转指令做如下扩充：分支指令（beq、bne、blez、bgtz、bltz)和跳转指令(j、jal、jr、jalr)；

3. 该处理器支持异常（为简单起见，可以只支持未定义指令异常）和中断（定时器中断）的处理。PC 的最高位 PC[31]为监督位。当该位为‘1’时，处理器处于内核态，此时异常和中断被禁止；当该位为‘0’时，处理器处于普通态，此时允许发生中断和异常。注意 PC[31]不能作为地址最高位去索引指令存储器，取指令时应当固定将地址最高位置零。只有 RESET、异常、中断等有可能将 PC[31]设置为‘1’，其他指令不能设置该位为‘1’，JR 和 JALR 指令可以使监督位清零。

- 在处理器复位后，PC 中的值应该为 0x80000000（处于内核态）；
- 发生中断时，PC 中的值应该为 0x80000004（处于内核态）；
- 在发生异常时，PC 中的值应该为 0x80000008（处于内核态）；
- PC+4 逻辑电路实现时应该保证 PC[31]不变；
- 分支语句和 J、JAL 语句不应该改变 PC[31]；
- 当执行 JR、JALR 指令时，PC[31]的值由跳转地址（$Ra）中的第 31 位（最高位）决定。

4. 数据存储的地址空间被划分为 2 部分：0x00000000～0x3FFFFFFF（字节地址）为数据 RAM，可以提供数据存储功能；0x40000000～0x7FFFFFFF（字节地址）为外设地 址空间，对其地址的读写对应到相应的外设资源（LEDs、SWITCH…）。

1 设计方案

本节描述了本流水线设计的结构，给出了文字描述、示意图、表格或伪代码。对应 Verilog 代码见第 2 节“关键代码及文件清单”。

1.1 整体结构设计

本实验设计实现一个符合哈佛架构的处理器，命名为 MonkeyMIPS。它遵循 MIPS32 Release1 指令集架构。

本处理器 SOPC（可编程片上系统）的顶层结构如图 1 所示，为核心处理器外接指令存储器、数据存储器和外设。核心处理器是一个具有五级流水线结构的处理器，指令存储器是一个 ROM 只读存储器，数据存储器是一个可读写的 RAM 随机存储器，外设包括定时器、数码管等外部设备及其控制结构。

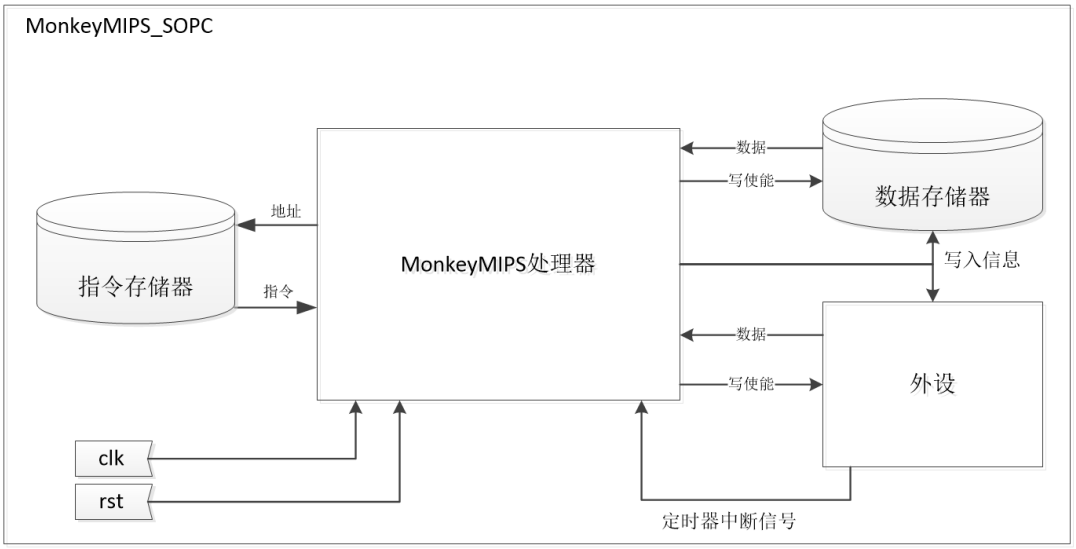


图 1 处理器 SOPC 结构设计（示意图）

设计核心处理器的五级流水线结构如图 2 所示。在“数字逻辑与处理器”课程所介绍的流水线结构基础之上，增添了 0 号协处理器 CP0 和控制模块 ctrl。由于信号传递关系众多，不便一一绘制，因此此处仅给出主要信号通路设计的示意图。其中黑实线是正常数据通路，点划线是转发通路，疏虚线是分支跳转地址，密虚线是异常处理和阻塞相关的路径。

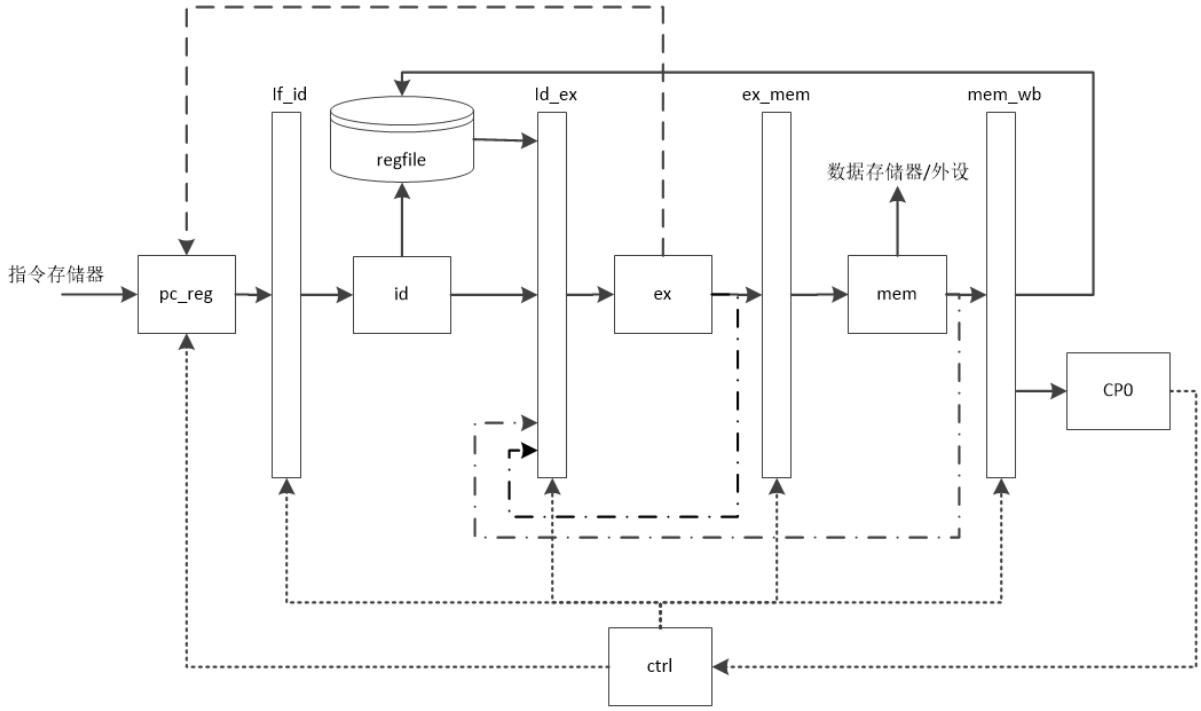


图 2 流水线主要信号通路设计（示意图）

图 3 是由 Vivado 生成的 RTL 原理图，可见实际实现的处理器 SOPC 结构与设计蓝图图 1 完全一致。其中处理器 MonkwyMIPS 的实现也满足了设计要求，由于生成的原理图尺寸太大、布线过于密集，此处不便展示。

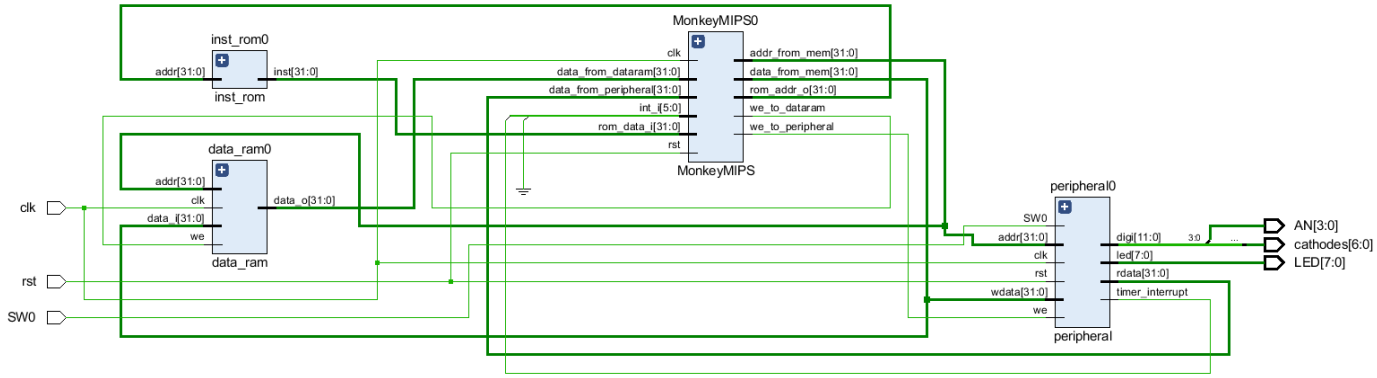


图 3 实际实现的处理器 SOPC 结构（RTL 原理图）

1.2 指令实现

1.2.1 取指令

取指令在 if 阶段中实现。需要通过一串条件判断来确定 PC 的来源，来源包括复位、中断、异常、分支、跳转、阻塞正常多种选项，如图 4 所示。其优先级如表 1 所示。

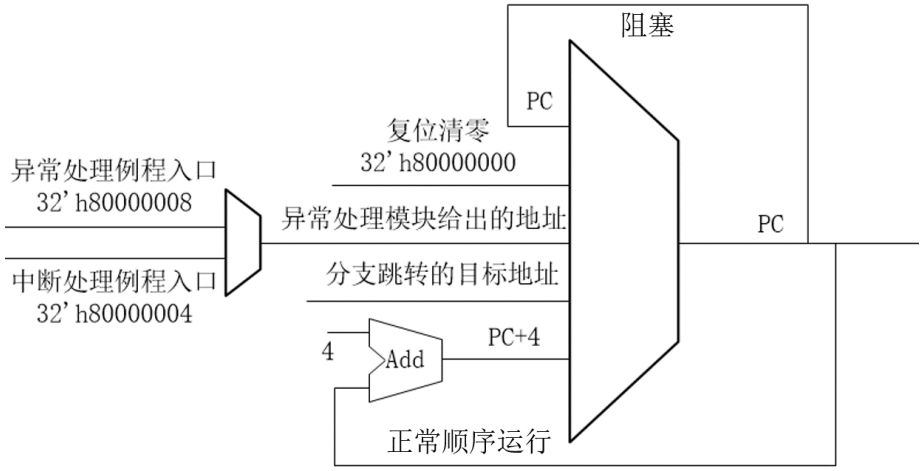


图 4 PC 来源示意图

优先级	条件	PC输出
	复位	80000000
高	flush信号有效	异常处理例程入口
↓	发生分支跳转	分支跳转目标地址
低	不发生阻塞	PC+4
		PC

表 1 PC 来源优先级表

1.2.2 译码

译码在 id 阶段实现。译码的码字完全按照 MIPS32 Release1 指令集架构。设计译码步骤如下：

步骤一，对 32 位指令分段，根据 opcode 区分指令类型，再根据 func 定位具体指令。

此处，指令类型包括 R 型指令、Regimm 型指令、Cop0 型指令和 I 型指令。具体指令涵盖了要求的 29 条指令，还在此之外增加了 mtc0，mfc0，eret 等指令，共计实现了 39 条指令。译码表见表 2、表 3。

R型指令	000000	100100 and 100101 or 100110 xor 100111 nor 000100 sllv 000110 srlv 000111 srav 100000 add 100001 addu 110100 teq	100010 sub 100011 subu 101010 slt 101011 sltu 001000 jr 001001 jalr 000000 sll 000010 srl 000011 sra
REGIMM型指令*	000001	00001 bgez	00000 bltz
COP0型指令**	010000	00100 mtc0 10000 eret	00000 mfc0

表 2 译码表

（*注：Regimm 型通过判断 Rt 确定指令；**注：COP0 型通过 Rs 确定指令）

指令类型（续）	opcode 具体指令	opcode 具体指令
I型指令	100011 lw 101011 sw 000010 j 000011 jal 000100 beq 000101 bne 000110 blez 000111 bgtz	001101 ori 001100 andi 001110 xori 001111 lui 001000 addi 001001 addiu 001010 slti 001011 sltiu

表 3 译码表（续）

步骤二，译码到具体指令之后，确定该指令的“将要用到的数据”和“将要进行的操作”，并向后传递。

此处，“将要用到的数据”包括送入寄存器堆的数据、写寄存器与否和立即数。“将要进行的操作”包括“运算类型码”和“具体执行码”。总结为下表。

信息	命名	数据宽度	去向
寄存器1的地址	reg1_addr_o	5	寄存器堆
寄存器2的地址	reg2_addr_o	5	寄存器堆
是否读寄存器1	reg1_read_o	1	id_ex段间寄存器
是否读寄存器2	reg2_read_o	1	id_ex段间寄存器
是否写寄存器	w_reg_o	1	id_ex段间寄存器
要写入的寄存器地址	w_dest_o	5	id_ex段间寄存器
32位立即数	imm	32	id_ex段间寄存器
运算类型码	alusel_o	3	id_ex段间寄存器
具体执行码	aluop_o	8	id_ex段间寄存器

表 4 译码阶段需确定的主要信息

其中，“将要进行的操作”的划分意味着把每个指令的执行过程拆分为两级，1）运算类型，2）具体执行。具体执行码与指令是一一对应的关系。而指令类型包括 7 种，如下表所示。

运算类型	编码	宏定义	内容
逻辑	001	EXE_RES_LOGIC	and,or,xor,nor,andi,ori,xori,lui
移位	010	EXE_RES_SHIFT	sll,srl,sra,sllv,srlv,srav
存储加载	011	EXE_RES_LOAD_STORE	lw,sw
移动	100	EXE_RES_MOVE	mfco,mtco
算术	101	EXE_RES_ARITHMETRIC	add,addu,addi,addiu,sub,subu,slt,sltu,slti,sltiu
转移	111	EXE_RES_JUMP_BRANCH	j,jal,beq,bgtz,blez,bne
空	000	EXE_RES_NOP	eret,teq

表 5 运算类型表

译码过程的实例见第 2 节“关键代码及文件清单”，展示了 and 指令的从译码至执行的全过程。

1.2.3 确定数据来源

id_ex 段间寄存器判断转发与否、读寄存器与否，从而确定 ex 级的数据来源。判断条件及其优先级见下表。

端口	优先级	条件	数据来源
reg1	1	EX/MEM hazard	ex输出的将要写入寄存器堆的运算结果
	2	MEM/WB hazard	mem输出的将要写入寄存器堆的运算结果
	3	需读取寄存器1	寄存器1的输出
	4	不需读取寄存器1	id传来的立即数
reg2	1	EX/MEM hazard	ex输出的将要写入寄存器堆的运算结果
	2	MEM/WB hazard	mem输出的将要写入寄存器堆的运算结果
	3	需读取寄存器2	寄存器2的输出
	4	不需读取寄存器2	id传来的立即数

表 6 执行阶段数据来源表

1.2.4 执行

ex 级获得 id_ex 段间寄存器传来的信息，包括数据来源、“运算类型码”、“具体执行码”以及“是否写”、“写哪里”。主要输出包括“是否写”、“写哪里”、“写什么”，以及“是否发生分支跳转”、“跳转到哪里”。表 7 总结了 ex 级的主要输出信息。

信息	命名	数据宽度	去向
是否写寄存器	w_reg_o	1	ex_mem段间寄存器
要写入的目的寄存器地址	w_dest_o	5	ex_mem段间寄存器
要写入目的寄存器的数据	w_data_o	32	ex_mem段间寄存器
是否发生转移	branch_flag_o	1	pc寄存器
分支跳转的目标地址	branch_target_addr_o	32	pc寄存器

表 7 执行阶段需要确定的主要信息

有 5 个类型（空类型无运算结果，存储加载型不在 ex 级运算）的运算在 ex 级中同时进行，得到 5 个运算结果。如果指令会产生有效运算结果的话，其中 4 个是 32’h000000 空结果，一个是正确结果；否则都是空结果。根据“运算类型码”选择该所需结果输出。工作原理如同一个两级多路选择器，如图 5 所示。

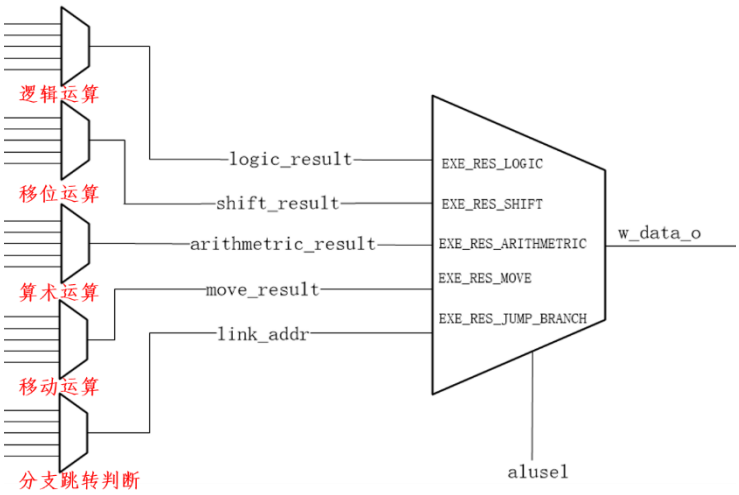


图 5 执行阶段两级多路选择示意图

其中，分支跳转判断的输出是 jal、jalr 指令要链接的地址。移动运算用于 mtco、mtco 指令。

存储加载型指令在 mem 级中执行，用单级选择即可。

1.2.5 存取

存取操作与 ex 阶段的运算操作类似，都是根据指令种类，利用获取的数据进行相应操作。

可存储地址区间被划分为两部分， 0x00000000～0x3FFFFFFF 为数据存储器，可以提供数据存储功能；0x40000000～0x7FFFFFFF 为外设地址空间，对其地址的读写对应到相应的外设资源。两类可存储地址仅有高四位有区别，因此只需对高四位进行判断。若为 0000~0011 则对数据存储器进行操作，若为 0100~0111 则对外设进行操作。

表 8 总结了 mem 模块与数据存储器及外设的信号交互。读与写操作的机制略有不同：在读取时，同时读得来自数据存储器和外设的数据，根据指令中的地址取用其中一个；在写入时，“要写入的地址”和“待写入的数据”同时传给数据存储器和外设，写使能信号仅使其中一个生效。

信息	命名	数据宽度	类型	去向
连接存储器/外设的地址	mem_addr_o	32	ouput	数据存储器和外设
待写入的数据	mem_data_o	32	ouput	数据存储器和外设
是否写入数据存储器	dataram_we_o	1	ouput	数据存储器
是否写入外设	peripheral_we_o	1	ouput	外设
来自数据存储器的数据	dataram_data_i	32	input	数据存储器
来自外设的数据	peripheral_data_i	32	input	外设

表 8 与数据存储器、外设交互的信息

1.2.6 回写

回写即写入寄存器堆。根据 mem 阶段输出的写寄存器信息，判断是否需要在上升沿写入寄存器堆，注意需要解决 RAW（先写后读）冒险。

这一阶段任务较少，因此为了流水线的平衡性，把异常与中断处理模块置于 wb 阶段中。见 1.4 小节“异常及中断处理”。

1.3 竞争问题解决

1.3.1 采用完全的 forwarding 电路解决数据关联问题

解决数据关联问题，即根据一系列条件判断是否发生数据冒险，以此确定执行阶段的数据来源。以下给

出两个数据端口的共四个数据关联判别式：

（1）数据端口 1 发生 EX/MEM hazard 的判别式为：

```
if(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs):
ID/EX.RegisterRs = EX/MEM.RegisterRd
```

（2）数据端口 1 发生 MEM/WB hazard 的判别式为：

```
if(MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRs)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs || ~ EX/MEM.RegWrite)):
ID/EX.RegisterRs = MEM/WB.RegisterRd
```

（3）数据端口 2 发生 EX/MEM hazard 的判别式为：

```
if(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRt):
ID/EX.RegisterRt = EX/MEM.RegisterRd
```

（4）数据端口 2 发生 MEM/WB hazard 的判别式为：

```
if(MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRs)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs || ~ EX/MEM.RegWrite)):
ID/EX.RegisterRt = MEM/WB.RegisterRd
```

该转发判定模块仿在 id_ex 段间寄存器中，与确定使用寄存器还是使用立即数一起比较，见 1.2.3 小节“确定数据来源”。

1.3.2 对于 load-use 类竞争采取阻塞一个周期+Forwarding 的方法解决

在 id 译码阶段，判定发生 load-use 类竞争的判别式为：

```
if ((EX.Opcode == lw) and (EX.RegisterRd == ID.RegisterRs)): stallRequest1 <= Stall
if ((EX.Opcode == lw) and (EX.RegisterRd == ID.RegisterRt)): stallRequest2 <= Stall
stallRequest = stallRequest1 | stallRequest2
```

即分别判断两个数据端口是否与 ex 阶段输出的目的寄存器重合，以此判定是否需要请求流水线阻塞。整体的流水线阻塞信号是两个子请求信号的或。

1.3.3 分支指令在 EX 阶段判断，在分支发生时刻取消 ID 和 IF 阶段的两条指令

分支指令在译码到具体指令后，产生一个信号，标志着此时发生分支。分支信号经过处理成为 flush 信号，传入 if_id 段间寄存器和 id_ex 段间寄存器。若 flush 信号有效，则在时钟上升沿来临时，段间寄存器的所有输出置零。id 和 if 阶段的所有输出置零，即取消 id 和 if 阶段的两条指令。

1.4 异常及中断处理

本处理器设计通过实现**协处理器**，支持对“**精确异常**”的处理。中断被视为异常的一种。

1.4.1 协处理器

协处理器是 MIPS 架构中的一个可选部件，具有与处理器核独立的寄存器，负责处理指令集的某个扩展。MIPS32 架构提供了最多 4 个协处理器，分别是 Coproc0~Coproc3。其中 0 号协处理器用作系统控制，例如：异常和中断处理、更改处理器配置、提供定时器。而 1 号和 3 号是负责浮点数运算的 FPU，2 号被保留用于特定实现。除了 0 号协处理器，都是可选的。

协处理器 Coprocessor 0 内部共有 32 个寄存器，支持配置工作状态、缓存控制、异常控制、TLB 控制等广泛的功能。部分寄存器是可写、可读的，通过指令 mtc0 (Move To Coprocessor 0)、指令 mfc0 (Move From Coprocessor 0)实现读写操作。

本流水线仅实现 32 个寄存器之中的 4 个寄存器，以完成异常处理功能，其标号、名称和功能见表 9。顺带实现了两个读写指令 mtc0 与 mfc0。

寄存器标号	名称	宽度	功能
9	Count	32	记录处理器运行周期数
11	Compare	32	定时器中断控制
13	Cause	32	保存异常原因
14	EPC	32	保存上一次异常时的PC地址

表 9 本流水线实现的协处理器 CP0 中的寄存器

（1）Count 寄存器是一个不停计数的 32 位寄存器，计数频率与 CPU 时钟相同，当计数达到 32 位上限时，从零重新开始计数。

（2）Compare 寄存器也是一个 32 位寄存器，与 Count 一起完成定时中断功能。当 Count 寄存器中的计数数值与 Compare 寄存器中的值一样时，会产生定时中断。该中断信号会一直保持，直至有新数据被写入 Compare 寄存器。因此，如果在中断处理例程中有写入 Compare 寄存器的指令，就可以实现按照特定周期不停地产生定时中断信号。

（3）Cause 寄存器也是一个 32 位寄存器，主要记录最近一次异常发生的原因以及各种相关信息，其各字段如表 10 所示。

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
内容	BD	R	CE		DC	PCI	0		IV	WP	0					
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
内容	IP								0	EXECODE					0	

表 10 Cause 寄存器字段表

其中，本处理器仅实现灰色的两个字段，IP (Interruption Pending)指中断挂起，用于指明对应中断是否发生。该字段一共有 8 位，最多可以支持 8 个外部中断信号的识别。EXECODE 是一个 5 位编码，用于记录发生了哪种异常。

这 5 位数字可以支持最多 32 个异常信息，实际上 MIPS 架构设定了中断、TLB 各种异常、地址或总线错误、系统调用、断点、未定义指令、协处理器不可用、整数溢出、自陷指令、机器检测错误等 16 个异常，其余编码被保留。本流水线中实现了其中 5 个异常信息。

EXECODE编码	hex	异常信息
00001	1	中断
01000	8	系统调用异常
01010	a	未定义指令异常
01101	d	自陷指令异常
01100	c	整数溢出异常
01110	e	异常返回

表 11 本流水线实现的异常信息

（4）EPC (Exception Program Counter)寄存器也是一个 32 位寄存器，专门用于存储异常指令的地址。异常处理例程结束之后，可以直接使用 eret (Exception Return)指令返回到异常发生的地址。这替代了实验要求中把异常中断地址存到某个寄存器中，再通过指令 jr 跳转返回的过程。

1.4.2 精确异常

什么是**精确异常**？它指的是，**异常处理的顺序**与**指令的顺序**一致，而不是与**异常发生的顺序**一致。

首先，通过一个例子阐述这两种顺序的区别，并以此说明精确异常的意义。

设有相邻的两条指令，lw $1,3($0)和 di $1,$5,$1，如图 6 所示。它们都会触发异常：指令 1 偏移量为 3，不是 4 的倍数，无法正确读取存储器地址，会导致地址未对齐异常，该异常会在第 4 个时钟周期发生；而指令 2 中的 di 并不是一个有效指令，会引起未定义指令异常，该异常会在第 3 个时钟周期发生。即，后一条指令先发生异常。此时，**异常处理的顺序**与**异常发生的顺序**一致，而不是与**指令的顺序**一致。

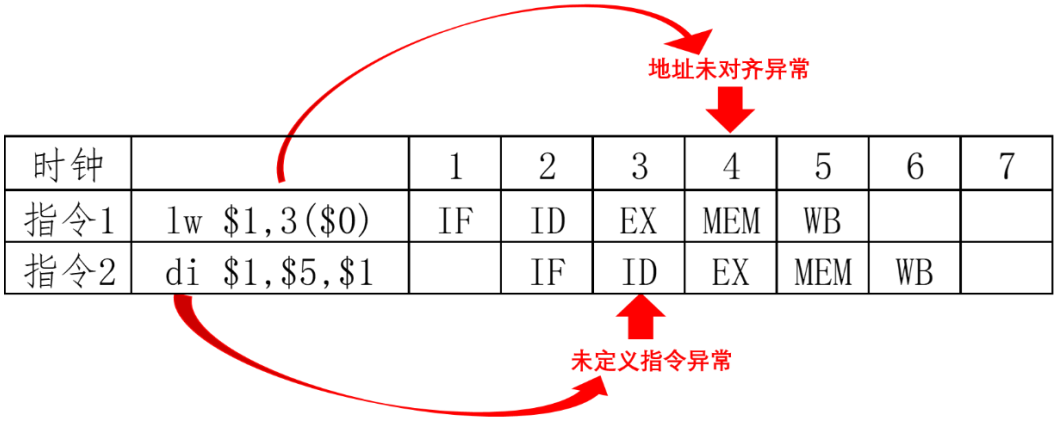


图 6 在流水线中，异常发生的顺序与指令的顺序未必一致

这是不希望发生的情况。因为当一个异常发生后，系统的顺序执行会被中断。此时有若干条指令处于流水线上的不同阶段，处理器会转移到异常处理例程，异常处理结束后返回原程序继续执行。**如果一旦异常发生就立即处理，则进入异常处理例程之后，返回的地址是指令 2 的下一个地址，则指令 1 就相当于是被遗弃了。**这破坏了原程序的正常顺序执行。

精确异常可以解决这个问题。在一个实现精确异常的处理器中，先发生的异常并不立即处理。异常只是被标记，并继续运行流水线，最终在某个阶段统一判定异常的发生与否。这样，异常的处理就会按照指令的顺序，而不是异常发生的顺序。

仍以上文的两个指令为例。本流水线在 mem 阶段中统一处理异常，因此对 di 指令在第 3 个周期发生的未定义指令异常不做处理，只是保存一个异常标记；它将会在第 5 周期被处理。而在第 4 周期，lw 指令的未对齐异常已经在 mem 阶段被检查到，处理器将进入未对齐异常的异常处理例程。异常处理例程结束之后，返回的地址是指令 1 的地址，所以处理器将顺序执行指令 2，再进行指令 2 的异常处理。由此可见，精确异常保持了原程序的顺序执行。

1.4.3 异常处理过程

在本流水线设计中，精确异常的“异常标记”机制使用异常向量实现。设计一个 32 位的向量，使用其中的部分字段用以记录异常信息，其余保留，如表 12 所示。

Bit	31:13	12	11	10	9	8	7:0
excepttype	0	异常返回	整数溢出	自陷指令	未定义指令	系统调用	中断

表 12 异常向量 excepttype 的字段表

当 mem 级检测到异常向量 excepttype 的某一位有效时，即异常发生时，处理器会执行一系列动作以处理异常。步骤如下：

（1） 检测监督位，如果处理器已经处于内核态，则不进行异常处理；

（2） 根据异常向量所表明的异常类型，转移到事先定义好的一个地址。该地址称为异常处理例程入

口，它指向异常处理例程，可进行异常处理；

（3） 在异常处理例程中进行具体的异常处理。处理结束后，使用 eret 指令返回到异常发生前的状态。

本流水线所设计的完整的异常处理过程流程图如图 7 所示。

其中，异常处理例程入口的选择和 flush 信号的生成，是在 ctrl 控制模块中完成的。它是为了控制 flush 信号、stall 信号和提供异常处理例程入口而专门设计的模块。

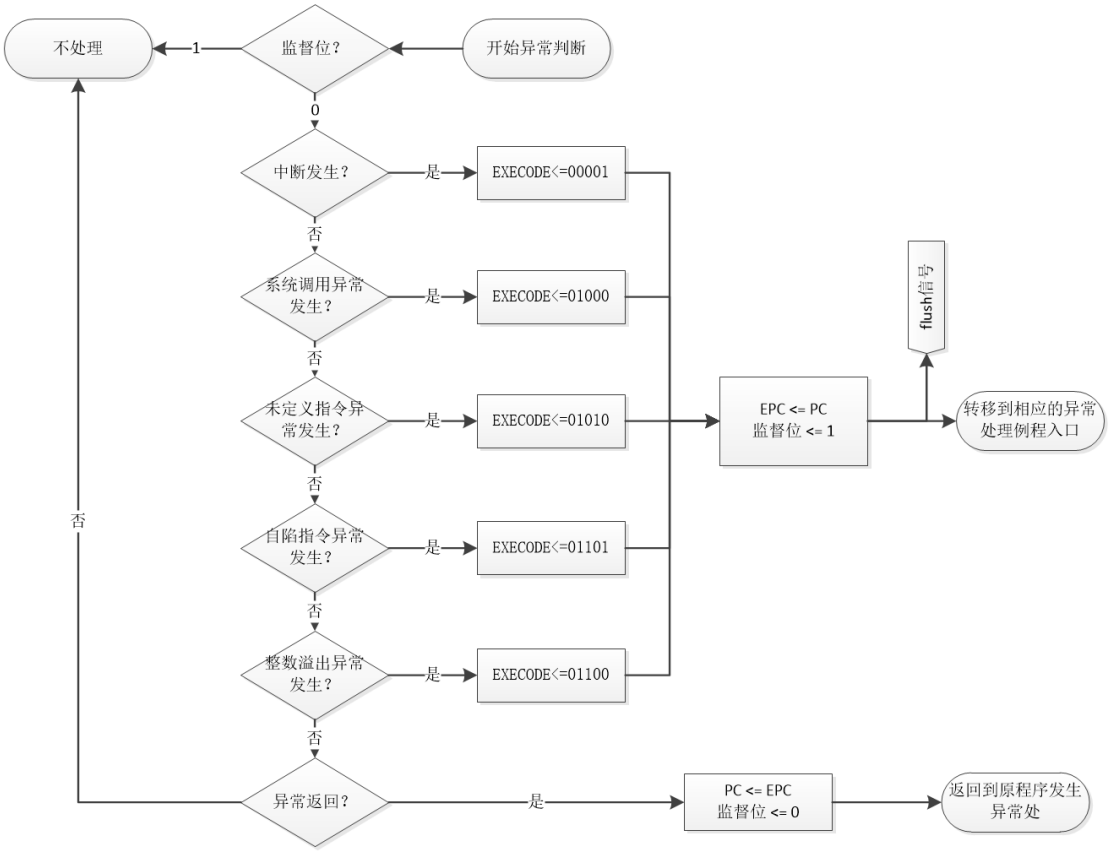


图 7 异常处理流程图

1.4.4 可扩展性

做出以上较为复杂的设计，最主要的考虑是可扩展性。

——如果流水线变得更复杂，例如级数增加、异常种类增加，怎么解决可能的出现异常处理顺序与指令顺序不一致的情况？

——如何规范地统一异常处理接口，使得可以方便地增加可处理的异常种类？

本流水线通过**精确异常**和**异常向量**的设计解决了这两个问题。

1.5 外设实现

可存储地址区间的第二部分，0x40000000～0x7FFFFFFF，为外设地址空间，对其地址的读写对应到相应的外设资源，包括 LED、数码管等。

实际上，协处理器提供了定时中断功能，可以通过以下步骤实现：

（1） 赋值一个通用寄存器，比如令$t1=100，使用汇编指令 mtf0 $t1,$11，把 100 存入协处理器 CP0 的 Compare 寄存器，此时 100 就是预置的周期数；

（2） 写入 Compare 寄存器的时候，运行计数寄存器 Count 归零，之后又开始每周期自增。增加到 Compare 预置值的时候，中断信号变为有效；

（3） 把该中断信号传给异常中断处理程序，即实现了定时中断。

以上做法已验证可行。但是为了满足本实验的要求，仍严格按照实验指导书上的做法实现：把预值周期数存入定时器 TH 中，等待定时器 TL 递增到32'hffffffff。

该定时器产生的周期信号作为数码管的时钟信号，驱动数码管的扫频显示。令数字的显示周期为 0.5 秒，则需要设置 TH 为$\text{ffffffff}_{\text{hex}} - 5000000_{\text{dec}} = \text{fd050f7f}_{\text{hex}}$，即把$\text{fd050f7f}_{\text{hex}}$设置为 TL 的初值。

系统时钟计数器 SysTick 的实现与协处理器中的 Count 时钟计数器实现非常类似，在复位信号使能时清零，之后在每个时钟周期的上升沿自增即可。

2 关键代码及文件清单

2.1 关键代码

2.1.1 指令实现

2.1.1.1 取指令

指令存放在指令 ROM 中，指令 ROM 的关键代码只有一句：

```
reg [`InstBus] inst_mem[0:`InstMemNum-1];
```

其中，宏定义 InstBus=32，是指令的宽度；宏定义 InstMemNum=512，是指令存储器的容量，即 ROM 的实际大小为$512 \times 4 \div 8 = 256$字节。指令 ROM 使用 initial 语句在初始时刻批量写入，之后不再更改，如图 8 所示。

本实验使用的排序算法，含写入数据存储器和数码管显示，共计 268 行，若手动翻译并写入是繁琐的。因此使用了 mips-linux-gnu 交叉编译工具链、链接描述脚本和 python 代码，实现了由汇编代码到 inst_mem[0] ~ inst_mem[267]这 268 行赋值指令的自动生成。这部分比较繁琐且与流水线关系不大，详见所附文件。

```
31      initial begin
32          inst_mem[0] <= 32'h08000019;
33          inst_mem[1] <= 32'h08000003;
34          inst_mem[2] <= 32'h08000018;
35          inst_mem[3] <= 32'h3c094000;
36          inst_mem[4] <= 32'h35290008;
37          inst_mem[5] <= 32'had200000;
```

图 8 写入指令 ROM 示例 (inst_rom.v)

取指令在 pc_reg 中实现。关键是通过一系列 if-else 语句判定 pc 的来源，代码如下图所示。

```
38      wire [4:0] PC_Source;
39      assign PC_Source = {rst, flush, jump_flag_i, branch_flag_i, stall};
40
41      //判断PC的来源
42      always @ (posedge clk) begin
43          case(PC_Source)
44              5'b10X00: pc <=  32'h00000000;
45              5'b01000: pc <=  new_pc;
46              5'b00100: pc <=  jump_target_addr_i;
47              5'b00010: pc <=  branch_target_addr_i;
48              5'b00001: begin end
49              default: pc[31:0] <=  pc[31:0] + 4;
50          endcase
51      end
```

图 9 取指令关键代码 (pc_reg.v)

2.1.1.2 译码

首先，在宏文件中定义运算类型编码和指令编码。格式如图 10、图 11 所示。

```
199     //AluSel
200     `define EXE_RES_LOGIC       3'b001
201     `define EXE_RES_SHIFT       3'b010
202     `define EXE_RES_MOVE        3'b100
203     `define EXE_RES_ARITHMETRIC 3'b101
204     `define EXE_RES_JUMP_BRANCH 3'b111
205     `define EXE_RES_LOAD_STORE  3'b011
206     `define EXE_RES_NOP         3'b000
```

图 10 运算类型编码宏定义 (defines.v)

```
137     //AluOp
138     //分支跳转指令
139     `define EXE_BEQ_OP          8'b11000100
140     `define EXE_BNE_OP          8'b11000101
141     `define EXE_BLEZ_OP         8'b11000110
142     `define EXE_BGTZ_OP         8'b11000111
143     `define EXE_BLTZ_OP         8'b11100000
144     `define EXE_BGEZ_OP         8'b11100001
145
146     `define EXE_JR_OP           8'b11001000
147     `define EXE_JALR_OP         8'b11001001
```

图 11 部分指令编码宏定义 (defines.v)

译码在 id 级实现。如图 12 所示，先定义主要信息输出端口。在模块内部，使用 case 结构，根据 opcode 定位指令，包括 R 型指令、Regimm 型指令、Cop0 型指令和 I 型指令，形式见图 12。

相比手动列出真值表并译码，这种结构有三大好处。(1) 具有极强的可扩展性，可以任意增删指令；(2) 利用了 case 结构的并行性，缩短最长路径；(3) 便于 Vivado 进行编译优化。

```
54      //输出到EX阶段的信息
55      output reg [`AluSelBus] alusel_o,    //ALU操作的类型（逻辑运算 等）
56      output reg [`AluOpBus] aluop_o,      //ALU操作的子类型（具体的 或 运算等）
57      output reg [`RegBus] reg1_o,         //运算的源操作数1
58      output reg [`RegBus] reg2_o,         //运算的源操作数2
59      output reg w_reg_o,                  //是否要写入目的寄存器
60      output reg [`RegAddrBus] w_dest_o    //要写入的目的寄存器地址
```

图 12 定义 id 译码模块的主要信息输出 (id.v)

```
99          case (opcode)
100             //R型指令
101             `EXE_R_TYPE:begin...
231             //end the case of "`EXE_R_TYPE"
232
233             `EXE_REGIMM_TYPE: begin...
256             //end the case of "`EXE_REGIMM_TYPE"
257
258             `EXE_LW: begin...
267             `EXE_SW: begin...
275             `EXE_J: begin...
283             `EXE_JAL: begin...
294             `EXE_BEQ: begin...
302             `EXE_BGTZ: begin...
310             `EXE_BLEZ: begin...
318             `EXE_BNE: begin...
326             `EXE_ORI:begin...
336             `EXE_ANDI:begin...
346             `EXE_XORI:begin...
356             `EXE_LUI :begin...
366             `EXE_ADDI :begin...
```

图 13 根据 opcode 定位指令 (id.v)

此处以指令 and 的例子说明译码过程，分支跳转指令的译码和执行过程将在第 2.1.2.2 小节说明。如图 14 所示，定位到 and 指令之后，对待输出信息进行赋值。指令 and 需要写寄存器，故写使能信号 w_reg_o 赋值为使能。指令 and 的“具体执行码”为 EXE_AND_OP，是一个预先定义好的 8 位码。指令 and 是逻辑运算指令，所以其“运算类型码”赋值为 EXE_RES_LOGIC。指令 and 需要读两个寄存器，所以两个数据端口 reg1 与 reg2，即 Rs 与 Rt，都赋值为读使能。变量 instvalid 的赋值是为了未定义异常的异常处理，默认该变量为 InstInvalid，只有定位到了有效指令中，才会将其赋值为有效。这样就可以实现对未定义异常的标记。

```
case (opcode)
    //R型指令
    `EXE_R_TYPE:begin
        case(func)
            `EXE_AND:begin
                w_reg_o <= `WriteEnable;
                aluop_o <= `EXE_AND_OP;
                alusel_o <= `EXE_RES_LOGIC;
                reg1_read_o <= `ReadEnable;
                reg2_read_o <= `ReadEnable;
                instvalid <= `InstValid;
            end
```

图 14 指令 and 的译码赋值（id.v）

```
//转发判断之后
else begin
    if(id_reg1_read_o == `ReadEnable)
        ex_reg1 <= RF_reg1_data_i;//取 读寄存器端口1 输出的值
    else
        ex_reg1 <= id_imm;//取 立即数
end
```

图 15 确定数据端口 1 的数据来源（不包括转发，id_ex.v）

2.1.1.3 确定数据来源

在 id_ex 段间寄存器确定数据来源。首先进行转发判断，转发相关代码见 2.1.2.1 小节“采用完全的 forwarding 电路解决数据关联问题”。在转发判断之后，根据是否需要读寄存器，决定输出是取寄存器堆读

取到的值还是取立即数，这部分代码见图 15。

2.1.1.4 寄存器堆

寄存器堆的关键代码也只有一句：

```
reg [`RegBus] regs[0:`RegNum-1];     //定义 32 个寄存器（二维向量）
```

其中，宏定义 RegBus=32，是寄存器堆数据的宽度；宏定义 RegNum=32，是寄存器堆的寄存器数量。寄存器堆采用了逻辑读取、时序写入的方案。既写又读的 RAW (Read After Write)数据冒险的解决见第 2.1.2.2 小节“RAW 数据冒险”。

2.1.1.4 执行

ex 级取得 id_ex 段间寄存器传来的信息进行运算等操作。

继续上文中指令 and 的例子。如图 16 所示，根据“具体执行码”定位到对应操作，运算结果 logic_result 被赋值为两个源数据的与。其余的 shift_result、arithmetic_result 等结果都被置零。最终，根据“运算类型码”选择一个最终结果作为 ex 级的输出，如图 17 所示。

```
//根据aluop_i指示的运算子类型进行运算：逻辑运算
always @ (*) begin
    logic_result <= `ZeroWord;
    case (aluop_i)
        `EXE_OR_OP: logic_result <= reg1_i | reg2_i;
        `EXE_AND_OP: logic_result <= reg1_i & reg2_i;
        `EXE_XOR_OP: logic_result <= reg1_i ^ reg2_i;
        `EXE_NOR_OP: logic_result <= ~(reg1_i | reg2_i);
        default: begin
        end
    endcase
end
```

图 16 根据“具体执行码”执行运算 (ex.v)

```
//根据alusel_i指示的运算类型进行运算，给出EX阶段的运算结果
always @(*) begin
    case (alusel_i)
        `EXE_RES_LOGIC: w_data_o <= logic_result;
        `EXE_RES_SHIFT: w_data_o <= shift_result;
        `EXE_RES_ARITHMETRIC: w_data_o <= arithmetric_result;
        `EXE_RES_MOVE: w_data_o <= move_result;
        `EXE_RES_JUMP_BRANCH: w_data_o <= link_addr;
    endcase
end
```

图 17 根据“运算类型码”选择一个最终结果(ex.v)

2.1.1.5 存取

数据存储器与使用二维数组实现，实现与指令存储器类似：

```
reg [`DataBus] data_mem[0:`DataMemNum-1];
```

其中，宏定义 DataBus=32，是数据的宽度；因为本实验要排序 100 个数字，所以取数据存储器的容量 DataMemNum 为 110。当然，这是非常容易调节的。

与指令存储器不同，它是随时可写的。在时钟的上升沿，若写使能信号有效，则可以写入数据存储器。如下图所示。

```
//写操作：上升沿写
always @ (posedge clk) begin
    if(we ==`WriteEnable) begin
        data_mem[addr[`DataMemNumLog2+1:2]] <=data_i;
    end
end
```

图 18 时序写数据存储器 (data_ram.v)

2.1.2 竞争问题解决

2.1.2.1 Forwarding

严格按照 1.3.1 小节“采用完全的 forwarding 电路解决数据关联问题”中列出的转发判别伪代码，写出从 ex 级或 mem 级的输出转发至数据端口 1 和数据端口 2 的条件判断代码。图 19 展示了对数据端口 1 的转

发判断。

```
111            if((ex_w_dest_o == id_reg1_addr_o)
112                &&(ex_w_dest_o != `RegNumLog2'h0)
113                &&(ex_w_reg_o == `WriteEnable)
114                &&(id_reg1_read_o == `ReadEnable)
115                )begin
116                ex_reg1 <= ex_w_data_o;
117            end else if((mem_w_dest_o == id_reg1_addr_o)
118                &&(mem_w_dest_o != `RegNumLog2'h0)
119                &&(mem_w_reg_o == `WriteEnable)
120                &&(id_reg1_read_o == `ReadEnable)
121                &&((ex_w_dest_o != id_reg1_addr_o)||(ex_w_reg_o == `WriteDisable))
122                ) begin
123                ex_reg1 <= mem_w_data_o;
124            end
```

图 19 对数据端口 1 的转发判断 (id_ex.v)

注 1：注意到，代码中判别式里的 4 个条件的顺序与此前判别式伪代码中的顺序不完全一致。这是经过多次试验发现的，若首先判断读/写寄存器的地址，主频会稍微高一些。原因可能是由于串联与表达式的短路特性，地址相同的情况较少。

注 2：把转发判断放在了 id_ex 阶段，在上升沿时刻判断。这也是经过试验确定的方案。**相比于转发到 id 阶段，本方案极大地缩短了最长路径，一次性把主频从 81MHz 提高到了 120MHz 左右。**原因可能有三个：

（1）转发判断是一套级联的包含很多串联逻辑表达式的 if-else 语句，耗时很长，放在本就相对冗长的 id 阶段，使得 id 阶段更加冗长。搬到 id_ex 阶段后，流水线相对平衡；

（2）时序逻辑仅在时钟上升沿生效，相对组合逻辑判断次数较少；

（3）消除了环路，这从改进前和改进后的最长路径 path 图中可以看出。

2.1.2.2 RAW 数据冒险

对于 RAW 数据冒险，也是用类似转发的方式解决。在读取寄存器堆时判断是否既写又读，若是，则直接把要写入的数据转发给要读取的数据，如下图所示。

```
//读寄存器端口1
always @ (*) begin
    if (raddr1 == `RegNumLog2'h0) begin        //所读寄存器为$zero的情况
        rdata1 <= `ZeroWord;
    end else
        //既写又读，此处解决了相邻2条指令时的RAW数据冒险
        if((raddr1 == waddr)&&(we == `WriteEnable)&&(re1 == `ReadEnable)) begin
            rdata1 <= wdata;
        end else begin
            rdata1 <= regs[raddr1];
        end
end
```

图 20 解决 RAW 数据冒险 (regfile.v)

2.1.2.3 Load-use 冒险

对于 load-use 类竞争采取阻塞一个周期+Forwarding 的方法解决。在 id 级对“上一条指令是否为 lw 指令”、“要写入的寄存器是否是当前在读的寄存器”和“当前是否要读寄存器”三个条件进行判断。若三个条

件同时成立，判定发生 load-use 冒险，发出阻塞请求信号。

```
539     assign pre_inst_is_load = (ex_aluop_i == `EXE_LW)?`TRUE:`FALSE;
540     assign stall_request = stall_request_load_use_1 | stall_request_load_use_2;
```

图 21 判断上一条指令是否为 lw 指令与请求阻塞 (id.v)

```
542     always @ (*) begin
543         stall_request_load_use_1 <= `NoStall;
544         stall_request_load_use_2 <= `NoStall;
545         //Load-use Hazard Stall
546         if ((pre_inst_is_load==`TRUE)&&(ex_wd_i==reg1_addr_o)&&(reg1_read_o==`TRUE)) begin
547                 stall_request_load_use_1 <= `Stall;
548         end
549         if ((pre_inst_is_load==`TRUE)&&(ex_wd_i==reg2_addr_o)&&(reg2_read_o==`TRUE)) begin
550                 stall_request_load_use_2 <= `Stall;
551         end
552     end
```

图 22 分别判断 load-use 冒险、请求阻塞的语句 (id.v)

2.1.2.4 控制冒险

本小节说明分支和跳转指令的实现机制，以及其对控制冒险的解决。

在 id 阶段中定位指令后，得到其写寄存器信息、“具体执行码”、“运算类型码”等信息。图 23 给出了两个例子，分别是 jalr 指令和 beq 指令。

```
`EXE_JALR:begin
    w_reg_o <= `WriteEnable;
    w_dest_o <= reg_Rd;
    aluop_o <= `EXE_JALR_OP;
    alusel_o <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= `ReadEnable;
    reg2_read_o <= `ReadDisable;
    instvalid <= `InstValid;
end
```

```
`EXE_BEQ: begin
    w_reg_o <= `WriteEnable;
    aluop_o <= `EXE_BEQ_OP;
    alusel_o <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= `ReadEnable;
    reg2_read_o <= `ReadEnable;
    instvalid <= `InstValid;
end
```

图 23 在 id 阶段定位指令 (id.v)

在 ex 阶段后，需要根据“具体执行码”确定 3 个变量：链接地址 link_addr、是否转移 branch_flag_o 和转移目的地址 branch_target_addr_o。它们都先默认置为无效或零，定位到“具体执行码”后进行赋值。

试以两个指令 bltz 和 beq 说明分支指令的实现。由于不要求 bal、bgezal 等指令，分支指令不需要对 link_addr 赋值。所以只需要在判断之后决定是否转移，转移地址为 PC+4 再加上偏移量的符号扩展。执行过程如图 24 所示。

```
`EXE_BLTZ_OP: begin
    if (reg1_i[31] == 1'b1) begin
        branch_flag_o <= `Branch;
        branch_target_addr_o <= pc_plus_4 + imm_sll2_signedext;
    end
end

`EXE_BEQ_OP: begin
    if (reg1_i == reg2_i) begin
        branch_flag_o <= `Branch;
        branch_target_addr_o <= pc_plus_4 + imm_sll2_signedext;
    end
end
```

图 24 分支指令的执行 (ex.v)

试以指令 j、jal、jr 说明跳转指令的实现。指令 j 和 jal 跳转过程是在 id 阶段完成的，在 id 段给出了跳转信号 jump_flag=`True 和跳转目标地址 jump_target_addr_o={PC+4 的高四位, 16 位立即数, 2'b00}。指令 j 不需要其他的执行内容，而指令 jal 需要保存当前地址的下一条地址，所以其执行过程就只剩下把链接地址赋值为 PC+8。**jr 则需要读取寄存器，这可能存在转发的问题，所以把跳转判断放在 ex 阶段中，视为分支类型的指令进行处理。**如图 25 所示。

```
case (aluop_i)
    `EXE_J_OP: begin
    end

    `EXE_JAL_OP: begin
        link_addr <= pc_plus_8;
    end

    `EXE_JR_OP: begin
        link_addr <= `ZeroWord;
        branch_target_addr_o <= reg1_i;
        branch_flag_o <= `Branch;
    end
```

图 25 跳转指令 jal 的执行 (ex.v)

```
always @ (posedge clk) begin
    if (flush == `FLUSH) begin
        id_pc <= `ZeroWord;
        id_inst <= `ZeroWord;
    end else if (stall != `Stall) begin
        id_pc <= if_pc;
        id_inst <= if_inst;
    end
end
```

图 26 取消 if 阶段的信号 (if_id.v)

分支和跳转发生时，转移标志信号 branch_flag_o 生效，使 pc_reg 的取指令地址改为转移目的地址 branch_target_addr_o。同时，该标志信号与异常处理的 flush 信号进行或运算之后，输入 if_id 和 id_ex 两个段间寄存器；若其有效，则在时钟上升沿将段间寄存器的输出清零，即实现了 if 和 id 两个阶段信号的取消。取消 if 阶段信号的代码见图 26，取消 id 阶段信号的代码与此类似。

是以解决了控制冒险。

2.1.3 异常及中断处理

2.1.3.2 协处理器

首先实现协处理器 CP0，定义协处理器的各个寄存器，如下图所示。

```
37      output reg [`RegBus] data_o,          //读出的CP0中某个寄存器的值
38      output reg [`RegBus] count_o,         //Count寄存器
39      output reg [`RegBus] compare_o,       //Compare寄存器
40      output reg [`RegBus] status_o,        //Status寄存器
41      output reg [`RegBus] cause_o,         //Cause寄存器
42      output reg [`RegBus] epc_o,           //EPC寄存器
43      output reg [`RegBus] prid_o,          //PRId寄存器
44      output reg [`RegBus] config_o,        //Config寄存器
45      output reg timer_int_o                //是否有定时中断发生
```

图 26 定义协处理器的寄存器 (CP0.v)

对这些寄存器均实现了与读写 regfile 寄存器堆相似的读写操作，此处不表。图 27 展示了协处理器的 Count 计数器，以及它与 Compare 寄存器比较生成中断信号的机制。

```
57              count_o <= count_o + 1;       //Count计数器，递增
58
59              if (compare_o != `ZeroWord && count_o ==compare_o) begin
60                  timer_int_o <= `Interrupt;  //定时器中断信号
61              end
```

图 27 协处理器定时中断 (CP0.v)

协处理器最主要的功能是根据异常向量进行异常处理。样例如图 28 所示。

```
case (execode_i)
        32'h1: begin                            //中断
            if (status_o[1] == 1'b0) begin
                epc_o <= current_pc_i;
                status_o[1] <= 1'b1;
            end
            cause_o[6:2] <= 5'b00000;
        end

        32'h8: begin                            //未定义指令
            if (status_o[1] == 1'b0) begin
                epc_o <= current_pc_i;
                status_o[1] <= 1'b1;
            end
            cause_o[6:2] <= 5'b01000;
        end
```

图 28 协处理器异常处理示例 (CP0.v)

2.1.3.2 精确异常

设置一个异常向量，随着正常的数据通路从 id 级向 mem 级传递。在每一级都可能出现异常，若出现异常，则将对应的字段置为有效。见图和图。最后在 mem 级中如果检查到整个异常向量并非全零，则开始异常处理。

异常向量是一个 32 位的 wire 型变量，支持 5 种异常的异常向量设计如下所示：

```
output [`RegBus] excepttype_o;
```

在 id 级，可能发生异常返回、未定义指令和系统调用指令三种异常：

```
assign excepttype_o = {19'b0,excepttype_is_eret,2'b0,instvalid,excepttype_is_syscall,8'b0};
```

在 ex 级，承接 id 级传来的异常，并且可能新增溢出和自陷指令两种异常：

```
assign excepttype_o = {excepttype_i[31:12],Overflow,Trap,excepttype_i[9:8],8'b0};
```

该异常向量一路传递到 mem 级，在此进行判断。图 29 展示了 mem 级对异常向量的判断。

```
114     //给出异常类型
115     always @ (*) begin
116         Execode_o <= `ZeroWord;
117         if(current_pc_i != `ZeroWord) begin
118             if ((cp0_cause_i[15:8] != 8'h00))   Execode_o <= 1'h1;      //中断
119             if ((excepttype_i[9] == `TRUE))     Execode_o <= 4'ha;      //未定义指令
120             if (excepttype_i[10] == `TRUE)      Execode_o <= 4'hd;      //自陷指令
121             if (excepttype_i[11] == `TRUE)      Execode_o <= 4'hc;      //溢出
122             if (excepttype_i[12] == `TRUE)      Execode_o <= 4'he;      //异常返回
123         end
```

图 29 在 mem 级统一处理异常信息 (mem.v)

从 mem 模块传出的异常信息 excepttype_o 传入控制模块 ctrl，在此给出 flush 信号和异常处理例程入口。

```
37        case (execode_i)
38            4'h0: begin new_pc <= `ZeroWord;     flush <= `NOFLUSH; end
39            4'h1: begin new_pc <= 32'h80000004; flush <= `FLUSH; end     //中断
40            4'ha: begin new_pc <= 32'h80000008; flush <= `FLUSH; end     //未定义指令
41            4'hd: begin new_pc <= 32'h80000004; flush <= `FLUSH; end     //自陷指令
42            4'hc: begin new_pc <= 32'h80000004; flush <= `FLUSH; end     //溢出
43            4'he: begin new_pc <= cp0_epc_i;    flush <= `FLUSH; end     //异常返回
44            default: begin end
45        endcase
```

图 30 控制模块给出 flush 信号和异常处理例程入口 (ctrl.v)

可以看到，中断的处理例程入口是 32'h80000004，异常的处理例程入口是 32'h80000008，而异常返回会回到 EPC 寄存器保存的地址。

2.1.4 外设实现

外设的读写严格按照“小学期实验 2019”PPT 上的要求完成，此处不表。数码管的控制照搬了实验二频率计实验中的数码管控制，在其实验报告中已有详述，此处不再赘述。

2.2 文件清单

本实验的文件一共分为两类，一类是 Verilog 代码，另一类是汇编代码及辅助工具。表 13、14 对此做了分类与说明。

分类	子分类	文件名	作用
Verilog	仿真专用	Monkey_playground.v	仿真testbench
	流水线	Monkey_MIPS_sopc.v	处理器系统的顶层文件
		MonkeyMIPS.v	流水线的顶层文件
		ctl.v	控制模块
		pc_reg.v	给出指令地址
		if_id.v	if与id之间的段间寄存器
		id.v	译码模块
		regfile.v	32位通用整数寄存器
		id_ex.v	id与ex之间的段间寄存器（含转发）
		ex.v	执行模块
		ex_mem.v	ex与mem之间的段间寄存器
		mem.v	访存模块（含精确异常判断）
		mem_wb.v	mem与wb之间的段间寄存器
		CP0.v	0号协处理器CP0
	存储器	data_ram.v	数据存储器RAM
		inst_rom.v	指令存储器ROM
	外设	peripheral.v	外设模块
		display.v	数码管控制模块
	宏定义文件	defines.v	宏定义文件

表 13 文件清单 1（Verilog 代码）

分类	子分类	文件名	作用
汇编代码及辅助工具	汇编代码	mysort20190824.S	冒泡排序汇编代码
		mysort20190824.txt	汇编代码翻译成的指令存储器赋值语句
	辅助工具	requirements.txt	环境依赖
		makefile	自动翻译程序，可直接在命令行中使用
		ram.ld	链接描述脚本，把交叉编译得到的elf可重定位文件转化为可执行文件
		organize.py	Python脚本，把二进制可执行文件翻译成指令存储器赋值语句
		generate_random.py	Python脚本，生成把100个随机数写入数据存储器的汇编代码
		random_for_mysort.txt	生成的把100个随机数写入数据存储器的汇编代码

表 14 文件清单 2（汇编代码及辅助工具）

3 仿真结果及分析

本节给出排序算法设计及其仿真结果。

3.1 输入设计

直接把待排序的 100 个随机数数组写在汇编代码中。一共产生了 100 个数字，范围在 1~3000 之间。把它们写入数据存储器的汇编代码段是用 python 程序生成的，下图展示了前四个数的写入。

```
55  #存入100个数
56  addi $t1,$zero,0x00000000  #数组开始的地址
57
58  addi $t0,$zero,0x1734
59  sw $t0,0($t1)
60  addi $t0,$zero,0x145
61  sw $t0,4($t1)
62  addi $t0,$zero,0x56
63  sw $t0,8($t1)
64  addi $t0,$zero,0x1109
65  sw $t0,12($t1)
```

图 31 把待排序的 100 个随机数写入存储器的汇编代码段 (mysort20190824.S)

3.2 排序算法

排序算法使用冒泡排序。其核心排序代码如下：

```
269 LOOP:                              # Start bubblesort
270     addi $s0,$t1,0
271     addi $s3,$zero,0               # set a flag
272 loop:
273     lw $s1,0($s0)
274     lw $s2,4($s0)          Load-use Hazard
275     beq $s2,$zero,exit             # if reaches the end of the array, go to "exit"
276     sub $s4,$s1,$s2        Ex-mem Hazard
277     blez $s4,NOswap        Ex-mem Hazard
278 swap:
279     addi $s3,$s3,1
280     sw $s2,0($s0)                  # swap
281     sw $s1,4($s0)
282 NOswap:
283     addi $s0,$s0,4
284     j loop
285 exit:
286     beq $s3,$zero,EXIT
287     j LOOP
288 EXIT:                              # End bubblesort
```

图 32 冒泡排序代码 (mysort20190824.S)

注意到，这段代码中有三处数据冒险，包括一处 load-use 冒险，两处 ex-mem 冒险。

验证过该算法的正确性之后，使用 Mars 的 Instruction Counter 对本排序算法进行计数，得共需要执行 64576 条指令。这个计数截至排序完成，不包括排序后调用外设的指令。

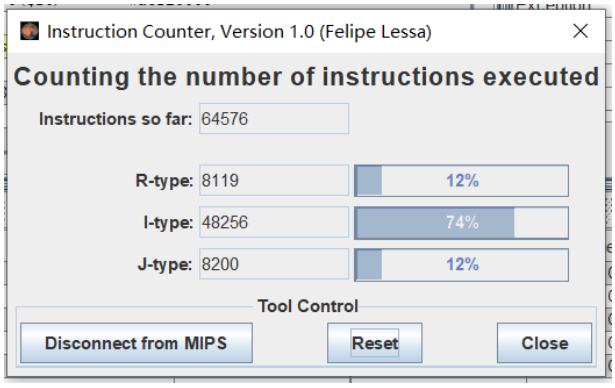


图 33 指令计数器结果

3.3 仿真结果

通过仿真，验证了设计的正确性，并且得到了本程序运行所需的周期数，并以此计算 CPI。表 15 总结了“有指令调度”以及“无指令调度”两种情况下的周期数、指令数以及 CPI。之后将进行具体分析。

指令调度	周期数(十进制)	指令数（十进制）	CPI
否	92736	64576	1.436
是	84536	64576	1.309

表 15 仿真结果汇总

仿真波形显示，冒泡排序的确得到了执行，交换过程如图 34 所示。图中共有 3 个数字“冒泡上浮”，1 个数字位置不变。

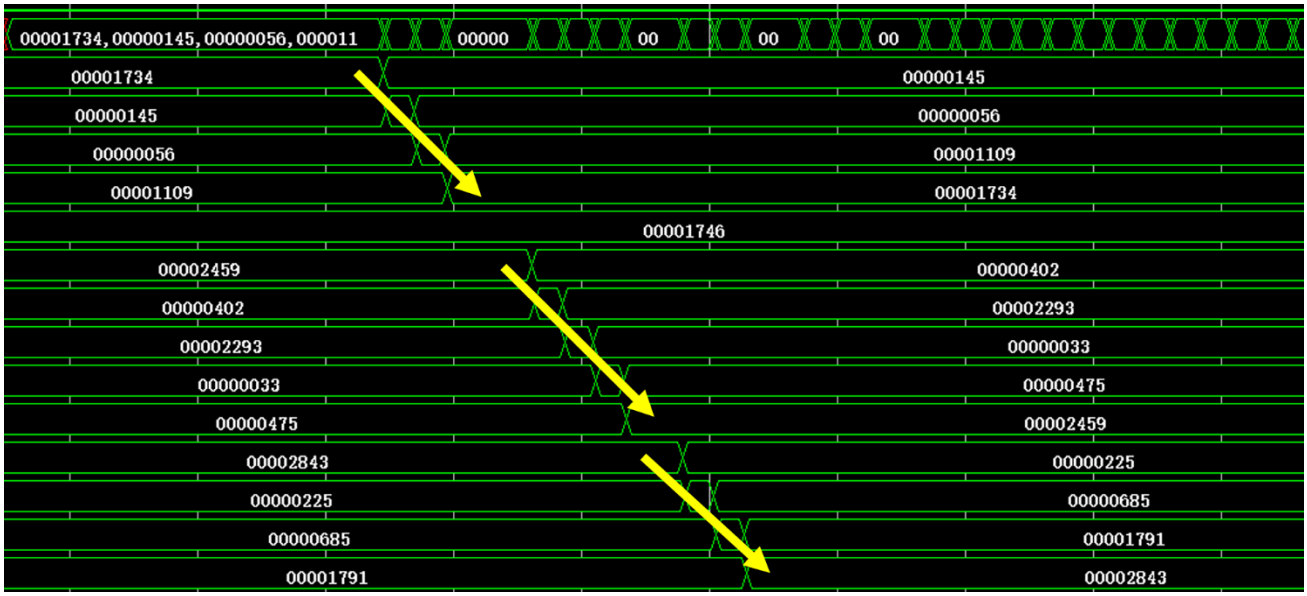


图 34 冒泡排序过程示意

经过数万次比较与交换，100 个随机数排序完毕，从小到大依次为 4，25，33，34……如图 35 所示。

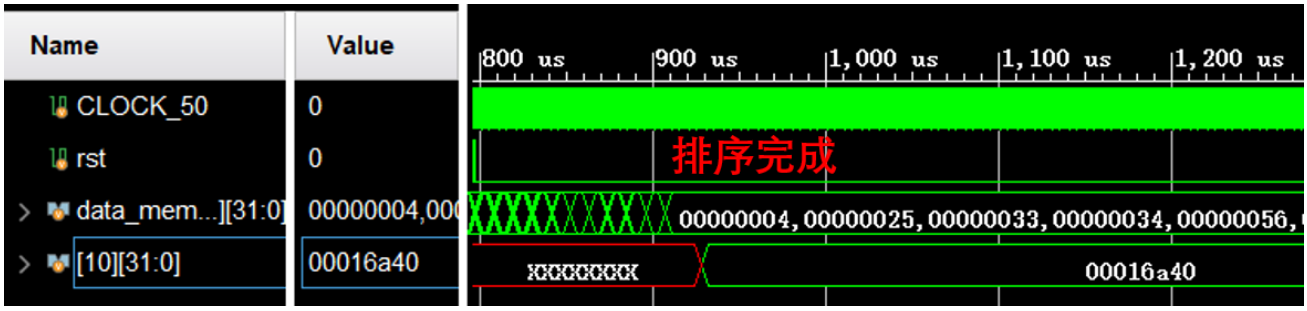


图 35 排序完成（指令调度前）

排序完成后，把此时外设记录的运行周期数存入寄存器$t2($10)，从上图读得周期数为$16a40_{hex} = 92736_{dec}$，即运行本程序共需要100935个时钟周期。由前所述，指令数为64576，算得$CPI = \frac{92736}{64576} = 1.436.$

注意到，此时运行的程序是存在 load-use 冒险的。如图 36、图 37 所示。

```
lw $s1,0($s0)
lw $s2,4($s0)
beq $s2,$zero,exit
```

图 36 指令调度前

```
lw $s2,4($s0)
lw $s1,0($s0)
beq $s2,$zero,exit
```

图 37 指令调度后

该冒险的解决方案是“阻塞一个时钟周期+转发”，所以空耗了很多时钟周期。若通过指令调度，把两句 lw 指令的顺序调换一下，消除此处的 load-use 冒险，则需要$14a38_{hex} = 84536_{dec}$个时钟周期，算得$CPI = \frac{84536}{64576} = 1.309.$

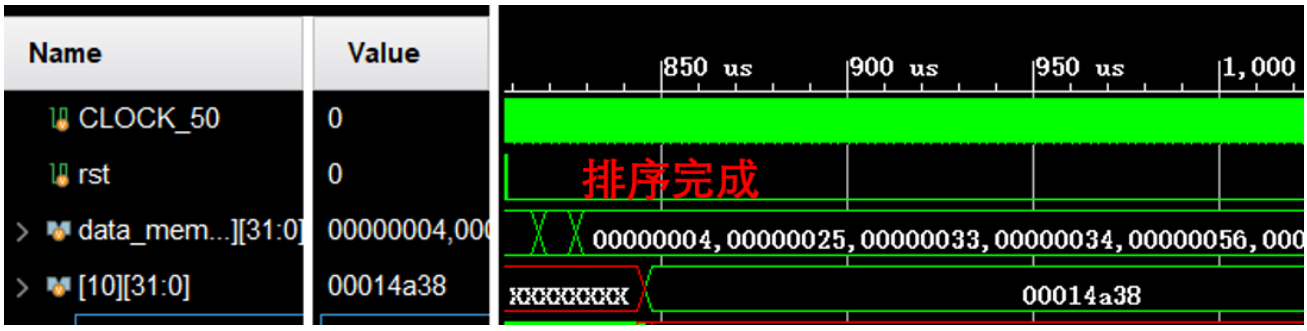


图 38 排序完成（指令调度后）

4 综合情况

4.1 整体综合情况

在综合与实现时，尝试了数十种不同的约束和策略的组合设置。表 16 展示了两组典型结果：

优化策略	时序性能					资源利用		功耗
	时钟约束/ns	WNS/ns	WHS/ns	最长路径/ns	主频/MHz	LUT	FF	总功耗/W
优选策略	6.200	0.007	0.052	5.786	172.8	1715(8.25%)	900(2.16%)	0.096
默认策略	10.000	1.818	0.059	7.611	131.4	1526(7.34%)	891(2.14%)	0.100

表 16 综合实现主要指标汇总

第一组是经过数十组尝试筛选出来的、可使本设计的时序性能达到最优的结果。该“优选策略”的策略

组合为：

（1）约束时钟周期为 6.20ns；

（2）取综合策略“Flow_PerfOptimized_high (Vivado Synthesis 2018)”；

（3）取实现策略“Performance_ExplorePostRoutePhysOpt (Vivado Implementation 2018)”。

第二组未利用任何额外的编译优化。即综合实现策略都取默认选项，并且时钟约束与 FPGA 保持一致。该“默认策略”的策略组合为：

（1）约束时钟周期为 10.00ns；

（2）取综合策略“Vivado Synthesis Defaults (Vivado Synthesis 2018)”；

（3）取实现策略“Vivado Implementation Defaults (Vivado Implementation 2018)”。

以下给出具体的分析与时序报告。

4. 2 策略一

4.2.1 时序性能

时序报告指出，$WNS = 0.007ns, WHS = 0.052ns$，均为正值。因此它是满足所有时序要求的。

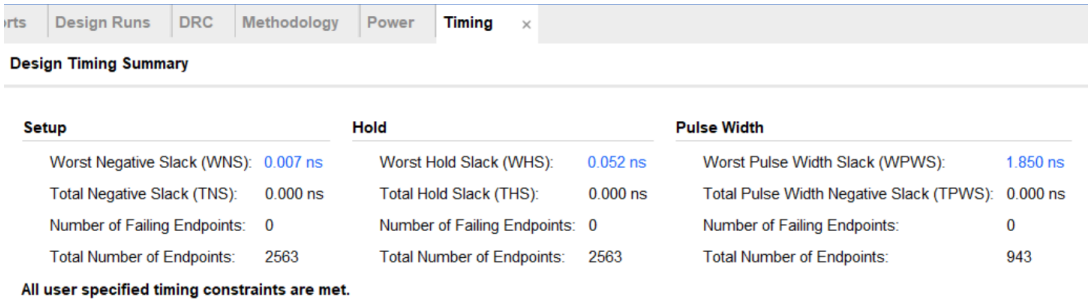
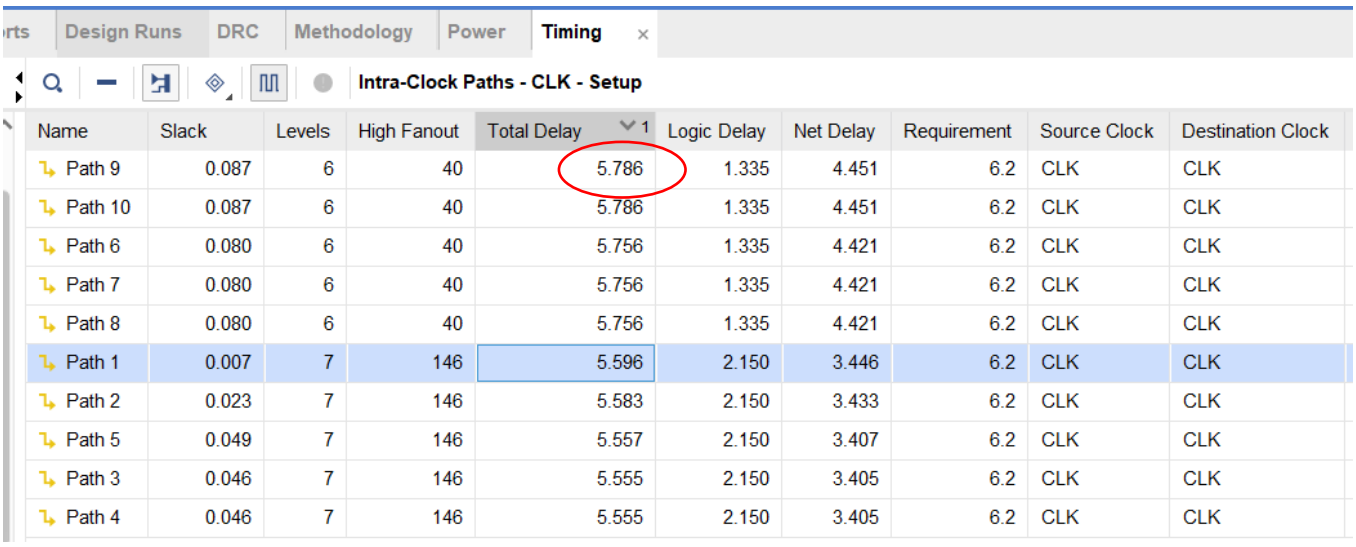


图 39 满足时序要求

时序报告表明，最长路径为$5.786ns$，算得主频为$\frac{1}{5.786ns} = 172.8MHz$。



Name	Slack	Levels	High Fanout	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 9	0.087	6	40	5.786	1.335	4.451	6.2	CLK	CLK
Path 10	0.087	6	40	5.786	1.335	4.451	6.2	CLK	CLK
Path 6	0.080	6	40	5.756	1.335	4.421	6.2	CLK	CLK
Path 7	0.080	6	40	5.756	1.335	4.421	6.2	CLK	CLK
Path 8	0.080	6	40	5.756	1.335	4.421	6.2	CLK	CLK
Path 1	0.007	7	146	5.596	2.150	3.446	6.2	CLK	CLK
Path 2	0.023	7	146	5.583	2.150	3.433	6.2	CLK	CLK
Path 5	0.049	7	146	5.557	2.150	3.407	6.2	CLK	CLK
Path 3	0.046	7	146	5.555	2.150	3.405	6.2	CLK	CLK
Path 4	0.046	7	146	5.555	2.150	3.405	6.2	CLK	CLK

图 40 最长路径为$5.786ns$

注意到，**裕量最少的路径**未必是**总延迟最长的路径**。需要通过对总延迟 Total Delay 进行降序排序，以取得最长路径延迟来计算主频。

4.2.2 资源利用

资源利用报告表明，使用 LUT 共 1715 个，占总可用数量的 8.25%；使用 LUTRAM 共 144 个，占总可用数量的 1.50%；使用 Slice Registers 共 900 个，占总可用数量的 2.16%；使用 IO 端口 21 个，占总可用数量的 19.81%。

Hierarchy

Name	Slice LUTs (20800)	Bonded IOB (106)	BUFGCTRL (32)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)
Monkey_MIPS_sopc	1715	21	7	900	36	558	1571	144
data_ram0 (data_ram)	162	0	0		0	65	66	96
MonkeyMIPS0 (MonkeyMIPS)	1411	0	0		36	470	1363	48
peripheral0 (peripheral)	129	0	0		0	71	129	0

图 41 资源利用报告

Resource	Utilization	Available	Utilization %
LUT	1715	20800	8.25
LUTRAM	144	9600	1.50
FF	900	41600	2.16
IO	21	106	19.81

图 42 资源利用报告（续）

4.2.3 功耗情况

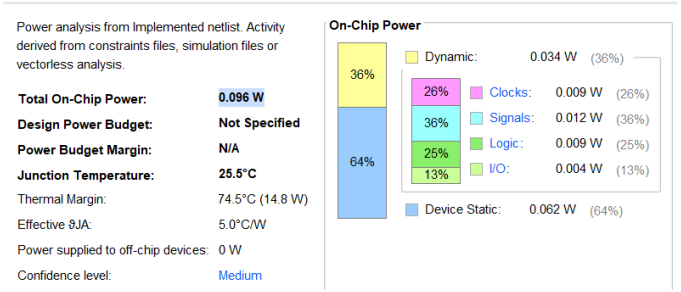


图 43 功耗情况

总功耗为$0.096W$，其中动态功耗$0.34W$，占总功耗的36%；静态功耗$0.62W$，占总功耗的64%。

4. 3 策略二

4.3.1 时序性能

时序报告指出，$WNS = 1.818ns, WHS = 0.059ns$，均为正值。因此它是满足所有时序要求的。

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.818 ns	Worst Hold Slack (WHS):	0.059 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2548	Total Number of Endpoints:	2548	Total Number of Endpoints:	934

All user specified timing constraints are met.

图 44 满足时序要求

时序报告表明，最长路径为$7.611ns$，换算得主频为$\frac{1}{7.611ns}=131.4MHz$。

Design Runs | DRC | Methodology | Power | Timing

Intra-Clock Paths - CLK - Setup

Name	Slack	Levels	High Fanout	Total...	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	1.818	6	140	7.611	1.393	6.218	10.000	CLK	CLK
Path 2	1.818	6	140	7.611	1.393	6.218	10.000	CLK	CLK
Path 3	1.864	6	140	7.564	1.393	6.171	10.000	CLK	CLK

图 45 最长路径为$7.611ns$

4.3.2 资源利用

资源利用报告表明，使用 LUT 共 1526 个，占总可用数量的 7.34%；使用 LUTRAM 共 144 个，占总可用数量的 1.50%；使用 Slice Registers 共 891 个，占总可用数量的 2.14%；使用 IO 端口 21 个，占总可用数量的 19.81%。

Hierarchy

Name	Slice LUTs (20800)	Bonded IOB (106)	BUFGCTRL (32)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)
Monkey_MIPS_sopc	1526	21	6	891	40	533	1382	144
data_ram0 (data_ram)	146	0	0		0	46	50	96
MonkeyMIPS0 (MonkeyMIPS)	1252	0	0		40	441	1204	48
peripheral0 (peripheral)	115	0	0		0	72	115	0

图 46 资源利用报告

Resource	Utilization	Available	Utilization %
LUT	1526	20800	7.34
LUTRAM	144	9600	1.50
FF	891	41600	2.14
IO	21	106	19.81

图 47 资源利用报告（续）

4.3.3 功耗情况

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.1 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	74.5°C (14.8 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W

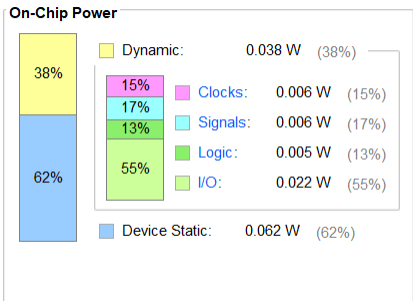


图 48 功耗情况

片上总功耗为$0.1W$，其中动态功耗$0.38W$，占总功耗的38%；静态功耗$0.62W$，占总功耗的62%。

4. 4 小结

4.4.1 资源与功耗比较

优化策略	主频/MHz	资源占用				功耗
		LUT	LUTRAM	FF	IO	总功耗/W
优选策略	172.8	1715(8.25%)	144(1.50%)	900(2.16%)	21(19.81%)	0.096
默认策略	131.4	1526(7.34%)	144(1.50%)	891(2.14%)	21(19.81%)	0.100

表 17 资源与功耗比较

比较两种优化策略的资源占用和功耗结果，发现：主频更高的情况下，资源占用略多，但是功耗略有降低。我的理解是，综合策略 Flow_PerfOptimized_high 和实现策略 Performance_ExplorePostRoutePhysOpt 起到了“以空间换时间”的作用，多占用了资源，以进一步优化时序路径。功耗的差别可能是由布线结束之后的物理优化引起的。

4.4.2 主频较高的原因分析

与许多同学的处理器相比，本处理器的主频相对较高，达到了 172.8MHz(131.4MHz)。试分析，除了优化策略的设置，主要原因有三个：

（1） 流水线结构较均衡。

流水线的频率是由最长路径最长的一级限制的，如果各级工作量失衡，延时最长的一级会严重拖累整个流水线的主频。本设计中，把 EX/MEM、MEM/EB 的转发以及 ex 阶段的数据来源判断放在了 id_ex 段间寄存器中，促进了 ID 级和 EX 级的平衡，实现了主频的大幅提升。更多分析见第 2.1.2.1 小节“Forwarding”的注 2。另外，把部分异常处理放在了相对清闲的 WB 阶段，也提高了各级的平衡性。

（2） 译码模块没有使用优先级译码电路。

优先级译码电路主要是由 if-else 结构语句生成的。级联的 if-else 语句耗时非常多，需要尽量避免。各项平级的 if 语句或者 case 并行逻辑不带有级联结构，可以避免这样的优先级译码设计。而且，case 结构更能利用 FPGA 的查找表结构，便于 Vivado 进行布局布线优化。

（3） 留意了一些优化细节。

例如：利用逻辑判别式短路特性，把不容易成立的条件放在串联与表达式的开头，见第 2.1.2.1 小节“Forwarding”的注 1；缩短从指令存储器取指令和从数据存储器取数据时的比较位宽（使用实际使用的地址线宽度 DataMemNumLog2 和 InstMemNumLog2 寻址，而不是使用 32 位寻址，以限制比较位宽）。

5 硬件调试情况

在 FPGA 上运行与仿真完全一致，时钟周期和 100 个数字依次显示都正常，没有出现意料之外的情况。

一共验收了两次，第一次是 8 月 30 日，完成了外设显示之后就验收。验收顺利通过，记录的主频为 81MHz 左右。第二次是 9 月 2 日，经过调整流水线结构，主频大幅提升，这次验收记录的主频为 129MHz。

6 思想体会

6.1 改进说明

本处理器增添了协处理器 CP0，实现了精确异常的控制与处理。并且直接用 eret 指令来实现异常返回。

必要性：我认为实验指导书所要求的是一个简化版的 MIPS。在查阅资料、了解完整版的 MIP32 架构之后，**我认为有必要遵循了较完整的 MIPS32 Release1 指令集架构来进行本实验的设计。**

合理性：改进后能够完成排序和扫描显示。

带来的好处：集中处理异常和中断，使流水线更为均衡，提高了主频速度；可扩展性极强，为更广泛的指令和操作留足了空间，见第 1.4.4 小节“可扩展性”。

6.2 亮点

（1）**主频较高，达到 172.8MHz (131.4MHz)。**

（2）大量使用了宏定义，使得代码有一定的可读性、可修改性。

（3）在译码阶段使用了一套“具体执行码”和“运算类型码”的译码逻辑，层次分明，实现指令的增删非常方便。

（4）用代码严格地、直观地描述了图 1、图 2 中的“处理器 SOPC 结构设计”与“流水线主要信号通路设计”示意图，较好地体现了“硬件描述语言”中的“描述”两个字。

（5）**我参与了国庆专项活动，并且是独立完成本实验。**

6.3 实验历程

本实验的持续时间比较长。原因是：我原计划暑期到海外实践，但是目的国家发生大罢工，大使馆停止了签证服务，所以我原计划未能成行；最终，暑假安排变成了在家写一个月大作业。具体历程大致如下：

- ~7 月 5 日：四处查阅资料，学习了 MIPS32 的完整指令集架构。
- 7 月 15 日~7 月 27 日：在未发实验指导书的情况下，完成了处理器初稿。这一阶段比较盲目，理念仅仅是想要尽可能地复现一个 MIPS32 处理器。在这一阶段，实现了接近 70 个指令，包括乘法、除法、乘累加、自陷指令等；实现了 5 种异常（见图 7）；实现了协处理器的定时中断功能。另外，分支跳转使用了延迟槽技术实现。
- 8 月 1 日：发现主频仅有 49MHz。
- 8 月 1 日-8 月 17 日：国庆专项活动训练。
- 8 月 18 日-8 月 28 日：为了满足实验指导书的要求，以及提高主频，做了大幅的删减，主要是删去了用不到的指令和多余的异常。因此命名此版本为 MonkeyMIPS，意为“猴版 MIPS”。同时，增加了外设。
- 8 月 30 日：第一次验收，顺利通过。主频 81MHz。
- 9 月 1 日-9 月 5 日：排查出耗时最多的因素是转发和延迟槽带来的大量判断。之后调整转发位置，弃用延迟槽，并且做了大量的优化，最终使主频提升到了 172.8MHz (131.4MHz)。
- 9 月 5 日-9 月 8 日：撰写实验报告。

6.4 附注

以上主频截图是 9 月 5 日的效果，仅含有实验指导书上的的异常处理。

之后补全了 5 种异常处理逻辑，并作了少量调整，也能达到 120MHz 以上。

7 参考文献

[1] Dominic Sweetman. 李鹏, 鲍峥, 石洋译. MIPS 体系结构透视[M].北京：机械工业出版社，2008.

[2] David A. Patterson, John L.Hennessy. 王党辉, 康继昌, 安建峰译. 计算机组成与设计 硬件/软件接口[M].北京：机械工业出版社，2015

[3] 高亚军.Vivado 从此开始[M].北京：电子工业出版社，2017.

[4] 何宾.Xilinx FPGA 权威设计指南 [M].北京：电子工业出版社，2018.

[5] 雷思磊.自己动手写 CPU[M].北京：电子工业出版社，2014.