计算机系统结构实验 Lab06

简单的类 MIPS 多周期流水化处理器实现

郝馳宇 521030910229

摘要

在本次实验中，我实现了一个支持 31 条指令的简单的类 MIPS 多周期流水化处理器。该处理器基于对 lab3、lab4 已经实现的器件和 lab5 实现的完整简单的类 MIPS 单周期处理器进行一些修改，又新实现了高速缓存模块(Cache)。此外，为了实现流水线，我们新添加了流水线段寄存器和部分信号，并增加了转发/前向通路（forwarding）和停顿（stall）来解决流水线中会遇到的冒险问题；与此同时，我们也实现了预测不转移（predict-not-taken）来提高条件跳转时的流水线运行效率。并通过了软件仿真和上板验证。

目录

1. 实验目的

1. 理解 CPU Pipeline、流水线冒险(hazard)及相关性，在 lab5 基础上设计简单流水线 CPU
2. 在 1.的基础上设计支持 Stall 的流水线 CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险/竞争、控制冒险和结构冒险。
3. 在 2.的基础上，增加 Forwarding 机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能。
4. 在 3.的基础上，通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能。
5. 在 4.的基础上，将 CPU 支持的指令数量扩充为 31 条，使处理器功能更加丰富。
6. 应用 Cache 原理，设计 Cache Line 并进行仿真验证。
7. 软件仿真和上板验证。

2. 模块原理分析

2.1. 主控制器（Ctr）原理分析

MIPS 架构所属的 RISC 指令集为定长指令集，这意味着我们可以方便地对指令进行分段处理，这样做的好处在于其很大程度上降低了译码和流水线执行的复杂程度，MIPS 指令的分段与解释如图 2.1、表 2.1 所示。

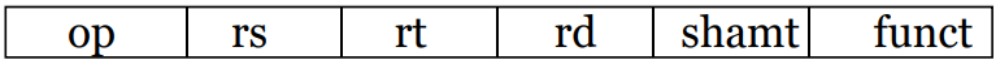

图 2.1 MIPS 指令分段示意图

表 2.1 MIPS 指令分段解释

指令段	位长	解释
op	6 bits	指示指令操作
rs	5 bits	第一个操作数的寄存器地址
rt	5 bits	第二个操作数的寄存器地址
rd	5 bits	写寄存器的地址
shamt	5 bits	位移量(只对 shift 指令有效)
funct	6 bits	opCode 的参数

MIPS 架构将指令分为 R 型，I 型与 J 型，其中 lw，sw，beq 指令属于 I 型，jump 指令属于 J 型，如图 2.2 所示。

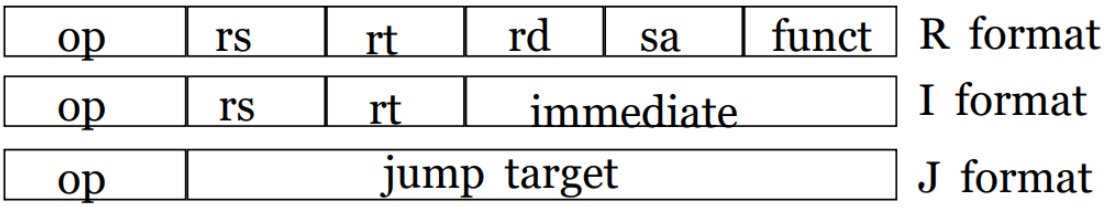

图 2.2 MIPS 各指令类型分段示意图

主控制单元（Ctr）的输入为指令的 opCode 字段[26:31]，操作码经过 Ctr 的译码，给 ALUCtr，Data Memory，Registers，Muxs 等功能单元输出正确的控制信号。其各输出信号与 opCode 输入的真值表如表 2.2 所示。与 lab5 不同的是，为了提高执行 jr 指令的效率，我们将 jrSign 的确定放在主控制器（Ctr）译码的阶段，即 ID 阶段完成，而 lab5 中的确定在 ALU 控制器（ALUCtr）的运行过程中进行

表 2.2 主控制器（Ctr）输入输出真值表

opCode	指令	regDst	aluSrc	memToReg	regWrite	memRead	memWrite	aluOp	jumpSign	extSign	jalSign	luiSign	jrSign	beqSign	bneSign
000000	R型指令	1	0	0	1	0	0	101	0	0	0	0	0	0	0
100011	lw	0	1	1	1	1	0	000	0	1	0	0	0	0	0
101011	sw	0	1	0	0	0	1	000	0	1	0	0	0	0	0
000100	beq	0	0	0	0	0	0	001	0	1	0	0	0	1	0
000100	bne	0	0	0	0	0	0	001	0	1	0	0	0	0	1
000010	j	0	0	0	0	0	0	000	1	0	0	0	0	0	0
000011	jal	0	0	0	1	0	0	000	1	0	1	0	0	0	0
000000	jr	1	0	0	0	0	0	101	0	0	0	0	1	0	0
001000	addi	0	1	0	1	0	0	000	0	1	0	0	0	0	0
001001	addiu	0	1	0	1	0	0	000	0	0	0	0	0	0	0
001100	andi	0	1	0	1	0	0	011	0	0	0	0	0	0	0
001101	ori	0	1	0	1	0	0	100	0	0	0	0	0	0	0
001110	xori	0	1	0	1	0	0	111	0	0	0	0	0	0	0
001111	lui	0	0	0	1	0	0	000	0	0	0	1	0	0	0
001010	slti	0	1	0	1	0	0	010	0	1	0	0	0	0	0
001010	sltiu	0	1	0	1	0	0	110	0	0	0	0	0	0	0
default	nop	0	0	0	0	0	0	00	0	0	0	0	0	0	0

其中各个信号含义如表 2.3 所示。

表 2.3 主控制器（Ctr）输出信号解释

信号	低电平时解释(0)	高电平时解释(1)
regDst	写寄存器的目标寄存器号来自 rt 字段	写寄存器的目标寄存器号来自 rd 字段
aluSrc	第二个 ALU 操作数来自寄存器第二个输出	第二个 ALU 操作数为指令低 16 位符号扩展
memToReg	写入寄存器的数据来自 ALU	写入寄存器的数据来自数据存储器
regWrite	无	寄存器堆写使能有效
memRead	数据存储器读使能有效	数据存储器读使能有效
memWrite	数据存储器写使能无效	数据存储器写使能有效
jumpSign	-	当前指令为无条件跳转指令
aluOp	<ALUCtr 输入信号>	<ALUCtr 输入信号>
extSign	符号扩展单元进行无符号扩展	符号扩展单元进行有符号扩展
jalSign	-	当前指令为 jal 指令
luiSign	-	当前指令为 lui 指令
jrSign	-	当前指令为 jr 指令
beqSign	-	当前指令为 beq 指令
bneSign	-	当前指令为 bne 指令

需要注意的是，表 2.3 中的 ALCOp 信号在单独设置时并没有实际的含义，需要将三位

组合起来解释，如表 2.4 所示。可见 opCode 的使用简化了部分非 R 型指令的译码过程，使其无需进入 ALUCtr 进一步设置信号即可得到 ALU 运算方式。

表 2.4 ALUOp 信号解释

ALUOp	指令	解释
000	lw, sw, addi, addiu	ALU 执行加法运算
001	beq	ALU 执行减法运算
010	slti	ALU 执行有符号数小于时置位运算
011	andi	ALU 执行按位与运算
100	ori	ALU 执行按位或运算
101	R-format	ALU 运算由[0:5]位的 funct 段决定
110	sltiu	ALU 执行无符号数小于时置位运算
111	xor	ALU 执行按位异或运算

可以看出，由于 lw，sw 需要计算地址偏移，因此 ALU 需要将两个输入进行加法运算，而 beq 指令实现跳转需要将 branch 信号与 ALU 的零输出信号取与，故需要将两个输入相减来设置零输出信号。对于 R 型指令，其 ALCOp 被设置为 010，而 ALU 具体的运算还需将 funct 字段输入 ALUCtr 中才能得到，详情见表 2.5。

另外注意 jr 和 R 型指令的 opCode 完全相同，均为 000000，为了方便设置 jrSign 信号，我们提前将 funct 段输入主控制器，在发下 opCode 为 000000 时，进一步判断 funct 段的类型，从而分辨出 jr 和 R 型指令，设置 jrSign 和 regWrite 信号。

由于输出信号组合和 opCode 输入一一对应，所以在实现主控制器的器件功能时，我们只需要使用 case 语句将 opcode 字段与输出信号一一对应，即可实现主控制器的功能。需要注意的时，对于 Ctr 不能识别的 opCode 取值，我们可以将该指令看作空指令（nop），将输出信号全部取 0 即可。

2.2. ALU 控制器部件（ALUCtr）原理分析

ALU 控制器部件（ALUCtr）接收指令 funct 字段[0:5]与主控制器（Ctr）输出的三位 ALUOp 信号，得出四位 ALUCtrOut 输出信号输出至算术逻辑运算单元（ALU），即可使得 ALU 根据不同的指令对两个输入值进行不同的运算。除此之外， shamtSign 信号的设置和输出也在此完成。

ALUCtr 根据不同输入得出的输出如表 2.5 所示。

表 2.5 ALUCtr 输出解释

aluOp	funct	指令	ALUCtr	ALU 运算
101	100000	add	0010	加法运算
101	100001	addu	0010	加法运算
101	100010	sub	0110	减法运算
101	100011	subu	0110	减法运算
101	100100	and	0000	逻辑与运算
101	100101	or	0001	逻辑或运算
101	100110	xor	1011	逻辑异或运算
101	100111	nor	1100	逻辑或非运算
101	101010	slt	0111	有符号数小于时置位运算
101	101011	sltu	1000	无符号数小于时置位运算
101	000000	sll	0011	逻辑左移运算

101	000010	srl	0100	逻辑右移运算
101	000011	sra	0100	算数右移运算
101	000100	sllv	0011	逻辑左移运算
101	000110	srlv	0100	逻辑右移运算
101	000111	srav	0100	算数右移运算
000	XXXXXX	lw	0010	加法运算
000	XXXXXX	sw	0010	加法运算
000	XXXXXX	addi	0010	加法运算
000	XXXXXX	addiu	0010	加法运算
001	XXXXXX	beq	0110	减法运算
001	XXXXXX	bne	0110	减法运算
011	XXXXXX	andi	0000	逻辑与运算
100	XXXXXX	ori	0001	逻辑或运算
111	XXXXXX	xori	1011	逻辑异或运算
010	XXXXXX	slti	0111	有符号数小于时置位运算
110	XXXXXX	sltiu	1000	无符号数小于时置位运算

其中，aluOp 为 101，funct 分别为 000000 或 000010 时，为 sll 或 srl 指令，此时 shamtSign 输出高电平指明该指令为位移指令，ALU 需要指令的 shamt 段作为输入。

另外注意这里 j、jr 和 jal 指令在运行时，ALU 实际上不用进行任何运算，这与 lab3 中的情形不同，这是因为 lab3 中我们只有几个简单的器件，而 lab6 中我们拥有一个完整的系统，这几个跳转指令跳转地址的计算可以交给另外的小器件完成，不需要 ALU 的工作。

2.3. 算术逻辑运算单元（ALU）原理分析

算术逻辑运算单元(ALU)是处理器的核心，其可以完成如加、减、逻辑与、逻辑或、逻辑或非和移位等操作，处理器各种指令的执行都离不开 ALU 的支持。

ALU 有三个输入，两个 32 位的输入作为操作数，一个 4 位的 ALUCtr 输入指定 ALU 对两个操作数执行的算术逻辑运算。并给出两个输出，一个 32 位的运算结果输出与零结果信号输出，其中零结果信号（Zero）输出在 ALU 运算结果为 0 时置 1，即输出高电平。Zero 信号用于与 branch 信号相与，如果结果为 1，则处理器将进行指令跳转，如果任意一位为 0，处理器都不会执行有条件跳转。

ALU 根据 ALUCtr 信号输入执行的不同操作如表 2.6 所示。

表 2.6 ALUCtr 与 ALU 对应运算

ALUCtr	解释
0000	逻辑与运算
0001	逻辑或运算
0010	加法运算
0011	逻辑左移
0100	逻辑右移
0101	无运算
0110	减法运算
0111	有符号数小于时置位运算
1000	无符号数小于时置位运算
1100	逻辑异或运算

1101	逻辑或非运算
1110	算数右移
default	无运算

2.4.寄存器模块（Registers）原理分析

寄存器模块（Register）是 CPU 内部用来存放数据的一些小型存储区域，用来暂时存放参与运算的数据和运算结果。寄存器完全由时序逻辑电路组成，由于其体积小，结构较简单，离处理器近，其数据传送速度非常快。

寄存器模块（Registers）将寄存器整合在一起，由于其内部有 32（2^5）个通用寄存器，故我们选择寄存器时需要向输入端口输入 5 位寄存器编号来选择目标寄存器。我们的寄存器位宽为 32 位，故寄存器写入端口和寄存器读取端口均为 32 位宽。另外考虑到快速读取数据和防止流水线结构冒险（Structure hazard）的发生，寄存器模块共有两个读端口和一个写端口。最后，我们还需要一个寄存器写使能信号（RegWrite）来防止数据竞争的发生。则寄存器模块的端口如图 2.2 所示。输入输出端口功能解释如表 2.7、2.8 所示。

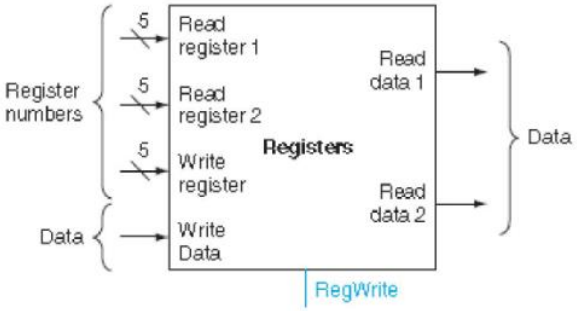

图 2.2 寄存器模块（Registers）端口图

表 2.7 寄存器模块（Registers）输入端口解释

输入端口	端口宽度	解释
readReg1	5	读取的寄存器 1 编号
readReg2	5	读取的寄存器 2 编号
writeReg	5	写入的寄存器编号
writeData	32	想要写入 writeReg 指示的寄存器的数据
regWrite	1	写使能信号，为 1 时允许写入
clk	1	时钟信号

表 2.8 寄存器模块（Registers）输出端口解释

输出端口	端口宽度	解释
readReg1	32	从 readReg1 指示的寄存器中读出的数据
readReg2	32	从 readReg2 指示的寄存器中读出的数据

由于读寄存器操作并不会改变寄存器内部的值，因此读寄存器可以在任意时间进行。而由于不确定 WriteReg、WriteData 和 RegWrite 信号的先后次序，我们采用时钟信号的下降沿作为写操作的同步信号，防止发生错误。

2.5.高速缓存模块（Cache）原理分析

高速缓存模块（Cache）是可以进行高速数据交换的存储器，它先于内存与 CPU 交换数

据，因此速率很快。当处理器访存时，它将先从 Cache 中寻找数据，如果想要的数据已经存在于 Cache 中，则处理器可以直接从 Cache 中获取想要的数据，从而缩短该访存操作所需要的时间；而若 Cache 中不存在想要的数据，则处理器才会访问内存，获取一个 Cache Line 并将其存在 Cache 中，以便下次使用。

根据 Cache Line 在 Cache 中存储的方式，Cache 可以分为直接映射、全相联映射和组相联映射三种。在直接映射中，Cache Line 只能放在 Cache 中的特定一行；在全相联映射中，Cache Line 可以放在 Cache 中的任意行；而在组相联映射中，Cache Line 可以放在 Cache 中的特定几行。在本实验中，我们使用全相联映射 Cache。

在本实验使用的 Cahce 一个 Cache Line 长度为 128 位，即 4 个字，一共 16 个 Cache Line。对于该 Cache，由于有 16 个 Cache Line，所以我们用地址中的 4 位确认映射位置再通过 25 位的 tag 和 valid 位来确认该位置的 Cache Line 是否为想要的数据。最后由于我们一次读取一个字，而一行 Cache Line 有 128 位，故我们使用最后 2 位地址选取想要的字。

最后，考虑到数据存储器实例化位于 Cache 的实例化内，因此我们需要保证接口的一致性，则 Cache 的输入输出接口和数据存储器完全相同。如表 2.9 和 2.10 所示。

表 2.9 高速缓存模块（Cache）输入端口解释

输入端口	端口宽度	解释
clk	1	时钟信号
address	32	访存地址
writeData	32	想要写入 address 指示的地址的数据
memWrite	1	写使能信号，为 1 时允许写入
memRead	1	读使能信号，为 1 时允许读取

表 2.10 高速缓存模块（Cache）输入端口解释

输出端口	端口宽度	解释
readData	32	从 address 指示的地址中读出的数据

2.6.数据存储器（Data Memory）原理分析

数据存储器（Data Memory）的功能是存储大量的数据，实际上同样是一种时序逻辑电路。按存储器的使用类型可分为只读存储器(ROM)和随机存取存储器(RAM)，而两者的结构和功能各不相同。在本次实验中，我们实际上需要设计实现的是可读可写的随机存取存储器（RAM），其需要支持数据读取、数据写入的功能。在实际计算机中，数据存储器的大小往往很大，但是在本次实验中，我们将其设置为保存 64 个 32 位数据，因此在理想状态下，该数据存储器的输入端口只需接收 6 位地址，但是在 32 位计算机中，地址的宽度可以达到 32 位，可以访问 2^{32}（4G）项存储单元，为了让我们的器件尽可能贴近真实计算机，我们还是将输入端口的宽度设置为 32 位。像寄存器模块一样，数据存储器也需要写使能信号，则其端口如图 2.3 所示。输入输出端口解释如表 2.11、2.12 所示。

在本次实验中，由于我们引入了 Cache，为了使其发挥作用，数据存储器一次访存应该能输出一行 Cache Line 的数据，即 128 字节，需要进行一些修改。

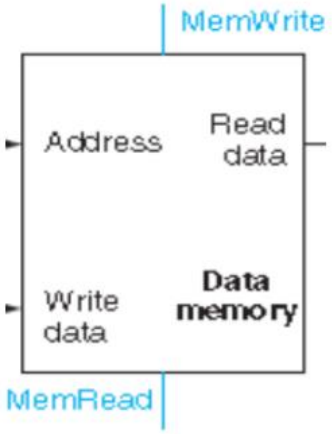

图 2.3 数据存储器（Data Memory）端口图

表 2.11 数据存储器（Data Memory）输入端口解释

输入端口	端口宽度	解释
clk	1	时钟信号
address	32	内存地址
writeData	32	想要写入 address 指示的地址的数据
memWrite	1	写使能信号,为 1 时允许写入
memRead	1	读使能信号,为 1 时允许读取

表 2.12 数据存储器（Data Memory）输入端口解释

输出端口	端口宽度	解释
readData	128	从 address 指示的地址中读出的数据

由于数据存储器和寄存器模块原理基本相同,故数据存储器的写操作同样需要设置在时钟信号的下降沿进行，防止发生错误。

另外，考虑到我们的数据存储器实际只有 64 个存储单元，但是地址有 32 位宽，为了防止地址越界导致出错,我们需要在器件实现中加入地址判断机制,如果输入的地址超过 1024,则将地址设为 0。

2.7.有符号扩展单元原理分析

有符号扩展单元的功能是将 16 位立即数符号扩展到 32 位，与 lab4 中的有符号扩展单元有所不同。该器件的接收 extSign 信号与一个 16 位的立即数作为输入，当 extSign 为 1 时，该模块进行有符号扩展，只需按照补码的规则，将 16 位有符号数最高位的值填充到 16-31 位即可；当 extSign 为 0 时该模块进行无符号扩展，直接将该立即数与 16 位的 0 进行拼接即可。对于该器件的有符号扩展有多种实现方法，一种简单的实现方法是将 16 位有符号数的最高位复制 16 次再与原数拼接在一起。其输入输出端口的解释如表 2.13、2.14 所示。

表 2.13 有符号扩展单元输入端口解释

输入端口	端口宽度	解释
in	16	进行有符号扩展的立即数
extSign	1	1：有符号扩展；0：无符号扩展

表 2.14 有符号扩展单元输出端口解释

输出端口	端口宽度	解释
out	32	有符号扩展结果

2.8.指令存储模块（Instruction Memory）原理分析

在该处理器中，指令存储模块与数据存储器（Data Memory）不同，为只读存储（ROM），存储处理器运行的各条指令。该模块接收一个 32 位的指令地址，并输出地址指示处的 32 位指令。

在本处理器中，我们的指令存储模块可以存储 1024 条 32 位指令，因此在理想状态下，该数据存储器的输入端口只需接收 6 位地址，但是在 32 位计算机中，地址的宽度可以达到 32 位，可以访问 2^{32}（4G）项存储单元，为了让我们的器件尽可能贴近真实计算机，我们还是将输入端口的宽度设置为 32 位。

2.9.多路选择器（Mux）原理分析

多路选择器（数据选择器）是一个多输入、单输出的组合逻辑电路，一个 n 输入的多路选择器就是一个 n 路的数字开关，可以根据通道选择控制信号的不同，从 n 个输入中选取一个输出到公共的输出端。在本处理器中，我们所使用的多路选择器均为 2 输入。其输入、输出端口如表 2.15、2.16 所示。

表 2.15 多路选择器（Mux）输入端口解释

输入端口	端口宽度	解释
sel	1	选择信号：1-input1　　2-input2
input1	5/32	输入数据 1
input2	5/32	输出数据 2

表 2.16 多路选择器（Mux）输出端口解释

输出端口	端口宽度	解释
out	5/32	输出数据

在本次实验中，我们实现了 5 位和 32 位两种宽度的多路选择器，以便处理 5 位寄存器地址和 32 位存储器地址。

2.10.程序计数器（Program Counter）原理分析

程序计数器模块用于管理当前指令地址。这个模块接受 32 位指令地址输入 address，在时钟上升沿将 address 作为结果 PCRes 输出。而当重置信号 reset 为高电平时，PCRes 输出 0，从而让整个系统从第一条指令开始重新运行，起到重置系统的作用。而在本次实验中，我们不专门设置一个程序计数器模块，而是在顶层模块中用几个小模块直接实现功能。

3. 流水线原理分析

3.1.流水线取指阶段（IF）原理分析

取指阶段较为简单，需要指令存储模块和程序计数器，然后按照程序计数器输出的指令地址读取指令存储器，并更新当前指令即可。

3.2.流水线译码阶段（ID）原理分析

译码阶段需要主控制器（Ctr）、寄存器模块（Registers）和有符号扩展单元，主控制器接收 IF 阶段传来的指令并将其分段，根据 2.1 所述产生必要的控制信号，而寄存器单元则负责根据指令里指定的寄存器读取所需要的数据，有符号扩展单元则根据主控制产生的 extSign 对指令的[15:0]部分进行有符号扩展或无符号扩展。

除此之外，本阶段还会设置一个多路选择器（MUX）根据主控制器产生的 regDst 信号在 rt 和 rd 中选出想要写入的寄存器，以备在写回阶段（WB）写入。

最后，无条件跳转指令（j、jr 和 jal）都会在本阶段完成。尽早结束无条件跳转可以减少流水线停顿（stall），提高流水线运行效率。

3.3.流水线执行阶段（EX）原理分析

执行阶段需要 ALU 控制器部件（ALUCtr）和算术逻辑运算部件（ALU）。ALU 控制器部件接收主控制器产生的 aluOp 信号和指令的 funct 段，产生控制 ALU 运算的信号和 shamtSign。而 ALU 则根据 ALUCtr 产生的信号对输入数据进行运算。

除了主要模块 ALUCtr 和 ALU 以外，执行阶段还有一些多路选择器（MUX），以便给模块提供正确的输入，例如根据 shamtSign 信号在 rs 和 shamt 段中选择一个输入 ALU 的操作数 1 的选择器和根据 aluSrc 信号在 rt 和 ID 阶段有符号扩展单元扩展结果中选择一个输入 ALU 的操作数 2 的选择器等；对于 lui 指令，lui 选择器会选择指令中的[15:0]部分作为结果的高 16 位。

3.4.流水线访存阶段（MEM）原理分析

访存阶段需要高速缓存模块(Cache)以加速访存操作。我们将数据存储器(Data Memory)的实例化放在 Cache 内部。对于需要进行访问内存的指令，将在本阶段完成。本阶段还有一个多路选择器，该选择器根据 memToReg 信号，在访存结果和 ALU 运算结果中选择一个数据作为访存阶段的结果。

3.5.流水线写回阶段（WB）原理分析

写回阶段需要使用到寄存器模块（Registers）。对于需要进行写入寄存器的指令，将在本阶段完成。写入寄存器在 ID 阶段已经确定，而写入的数据在 MEM 阶段也被确定。

4. 顶层模块（Top）原理分析

4.1.流水线段寄存器原理分析

流水线段寄存器存在于两个阶段之间，通过流水线段寄存器存储每个阶段产生的结果，可以实现数据和信号在流水线阶段之间流动。下面我们简要叙述本次实现实现的处理器中各组流水线寄存器的组成。亦如图 4.1 所示。

* IF/ID 流水线段寄存器：包含指令及其指令地址（PC）。

* ID/EX 流水线段寄存器：主要包含 EX 阶段运行所需要的信号（memWrite、memRead、memToReg、regWrite 、aluOp、aluSrc、luiSign、beqSign、bneSign）、指令各段如 rs、rt、funct、shamt 的值、写入的寄存器、有符号扩展单元的扩展结果、寄存器读取的输出、指令和地址（PC）。

* EX/MEM 流水线段寄存器：主要包含 MEM 阶段运行所需要的信号（memWrite、memRead、memToReg、regWrite）、ALU 运算输出的结果、rt 寄存器的值和写入的寄存器。

* MEM/WB 流水线段寄存器：主要包含 regWrite 信号、写入寄存器的数据以及写入的寄存器。

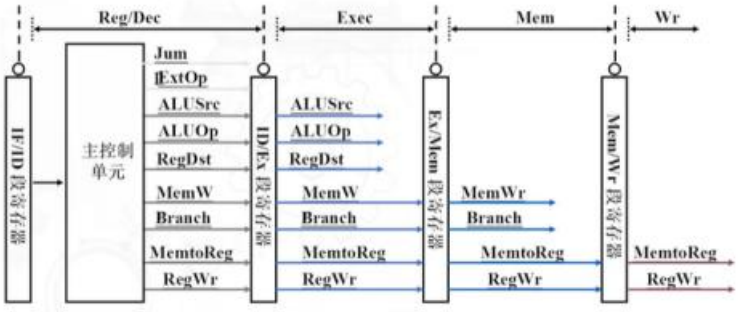

图 4.1 流水线段寄存器示意图

4.2. 流水线整体原理分析

顶层模块的功能是将之前所叙述的所有器件进行实例化后有逻辑地进行连线，连线中包含数据总线和控制总线，数据总线负责传输各器件的数据输入和输出，控制总线则负责传输系统运行过程中的各个控制信号。

顶层模块的整体结构如图 4.1 所示，该结构图来自计算机系统结构（A 类）（CS2305）沈艳艳老师的课件，在此向沈艳艳老师致以诚挚的谢意。

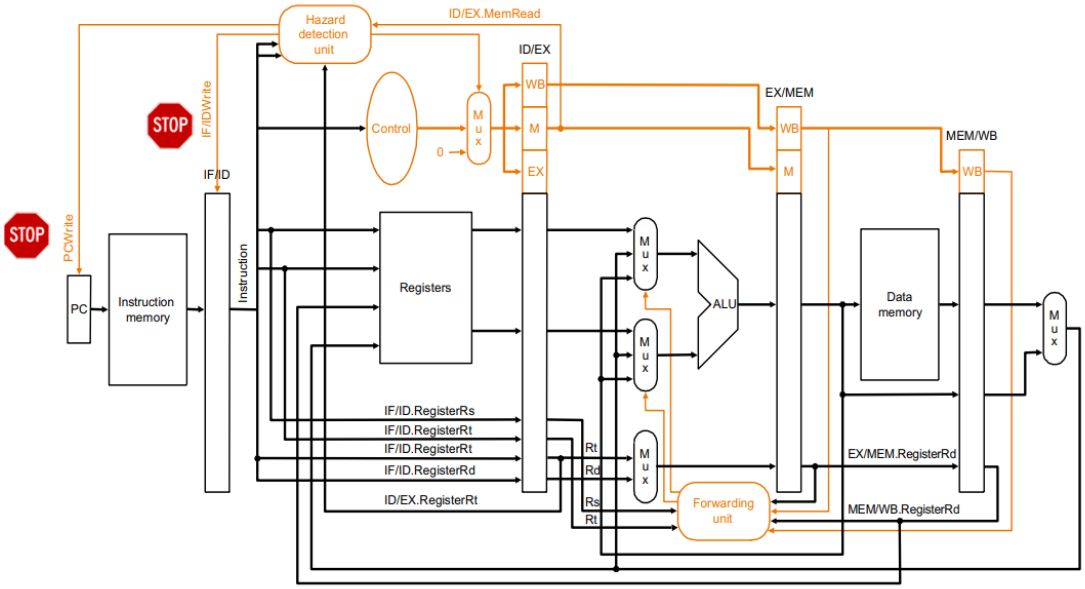

图 4.1 顶层模块（Top）整体结构图

4.3.数据转发（Forwarding）原理分析

在本实验实现的处理器中，流水线中会发生两种数据冒险（Data Hazard），一种是后面的运算依赖于前面的运算结果，另一种是后面的运算依赖于前面读取的数据。如果不加以特殊处理，这两种数据冒险均会引发流水线停顿（stall），影响流水线运行的效率。对于第一种数据冒险，我们使用数据转发（Forwarding）解决。通过在顶层模块中添加转发数据通路，运算可以提前从 EX\MEM、MEM\WB 流水线段寄存器中获取操作数，而不需要等待流水线运行至下一阶段，进而避免流水线停顿，提高流水线运行效率。

4.4.流水线停顿（stall）原理分析

按照 4.3 所述的数据转发只能结果第一类数据冒险，而对于第二类数据冒险，因为流水线必须等到前一指令 MEM 阶段结束才能从内存中取到想要的数据，而后一条指令必须在 EX 阶段前获取操作数，因此在后一条指令进入 EX 阶段前，流水线必须停顿一周期等到前一条指令离开 MEM 阶段，随后读取的数据将会通过数据转发交给后一条指令。

在每条指令的 ID 阶段，我们会检测当前是否存在第二类数据冒险，如果存在冒险，则发出流水线停顿信号，使得当前流水线 IF、ID 阶段的指令停顿一个周期。

另外，对于 jal 指令，其需要写入%31 寄存器，可能会与当前处于 WB 阶段的指令竞争写端口，为了避免这种情况发生，我们在执行 jal 指令时让 EX、MEM、WB 的指令停顿一个周期以让出寄存器模块写端口。

4.5.流水线分支预测（Predict-not-taken）原理分析

对于条件跳转指令，这些指令将会在 EX 阶段确定跳转或不跳转，通过每次都预测不转移（predict-not-taken），如果 EX 阶段确认确实不转移，则流水线继续运行，没有停顿。如果实际上发生了跳转，则对流水线进行 flush 操作，清空 ID，IF 阶段的段寄存器重新运行。

对于无条件跳转指令（j、jr 和 jal），我们将其提前至 ID 阶段处理，以此避免流水线停顿。

5. 功能实现

5.1. 主控制器（Ctr）功能实现

根据 2.1 部分的分析，我们建立 Ctr.v 文件，采用 case 语句对各种 opCode 输入对应的信号输出进行穷举。其主要逻辑如图 5.1 所示。限于篇幅，在这里我们只展示部分主要代码，其余部分代码可以根据 2.1 节的表 2.2 推出。相比 lab5，我们将 branch 信号放在其他阶段进行，并新加了四个新信号。

需要注意的是，另外注意 jr 和 R 型指令的 opCode 完全相同，均为 000000，为了方便设置 jrSign 信号，我们提前将 funct 段输入主控制器，在发下 opCode 为 000000 时，进一步判断 funct 段的类型，从而分辨出 jr 和 R 型指令，设置 jrSign 和 regWrite 信号。

```
always @(opCode)
begin
    case(opCode)
    //R-format, jr
    6'b000000:
    begin
        RegDst = 1;
        ALUSrc = 0;
        MemToReg = 0;
        MemRead = 0;
        MemWrite = 0;
        JumpSign = 0;
        ALUOp = 3'b101;
        ExtSign = 0;
        JalSign = 0;
        if (funct == 6'b001000) //jr
        begin
            RegWrite = 0;
            JrSign = 1;
        end
        else
        begin
            RegWrite = 1;
            JrSign = 0;
        end
        LuiSign = 0;
        BeqSign = 0;
        BneSign = 0;
    end

        //lw
        6'b100011:
        begin
            RegDst = 0;
            ALUSrc = 1;
            MemToReg = 1;
            RegWrite = 1;
            MemRead = 1;
            MemWrite = 0;
            ALUOp = 3'b000;
            JumpSign = 0;
            ExtSign = 1;
            JalSign = 0;
            LuiSign = 0;
            JrSign = 0;
            BeqSign = 0;
            BneSign = 0;
        end
```

图 5.1 Ctr.v 主要逻辑

注意这里我们将穷举范围外的 opCode 视为空指令（nop）并设置将所有信号设置为 0。

5.2. ALU 控制器部件（ALUCtr）功能实现

根据 2.2 部分的分析，我们建立 ALUCtr.v 文件，采用 case 语句对各种 ALUOp 和 funct 输入对应的信号输出进行穷举。其主要逻辑如图 5.2 所示。限于篇幅，在这里我们只展示部分主要代码，其余部分代码可以根据 2.2 节的表 2.3 推出。

```
always @ (aluOp or funct)
begin
    ShamtSign = 0;

    casex ({aluOp, funct})
    //R-format--------------------------------------------------------------------
        //add
        9'b101100000:
            ALUCtrOut = 4'b0010;

        //addu
        9'b101100001:
            ALUCtrOut = 4'b0010;

        //sub
        9'b101100010:
            ALUCtrOut = 4'b0110;

        //sub
        9'b101100011:
            ALUCtrOut = 4'b0110;


      //srl
      9'b101000010:
      begin
          ALUCtrOut = 4'b0100;
          ShamtSign = 1;
      end

      //sra
      9'b101000011:
      begin
          ALUCtrOut = 4'b1110;
          ShamtSign = 1;
      end

      //sllv
      9'b101000100:
          ALUCtrOut = 4'b0011;

      //srlv
      9'b101000110:
          ALUCtrOut = 4'b0100;
```

```
//non R-format------------------------------------------------------------

    //lw, sw, addi, addiu
    9'b000xxxxxx:
        ALUCtrOut = 4'b0010;

    //beq, bne
    9'b001xxxxxx:
        ALUCtrOut = 4'b0110;

    //andi
    9'b011xxxxxx:
        ALUCtrOut = 4'b0000;

    //ori
    9'b100xxxxxx:
        ALUCtrOut = 4'b0001;
```

图 5.2 ALUCtr.v 主要逻辑

注意这里我们使用了 casex 语句而非 case 语句以覆盖到输入中 X 的所有取值。但是我们仍然为器件的输出取了确定的值，这么做的意图在于在仿真时显示绿色波形，保证美观性，否则在波形图中，带 X 的输入输出会被认为是无确定值而显示红色

此外，同样值得注意的是我们将 shmatSign 的设置放在了 ALUCtr 中完成而不是 Ctr。

5.3. 算术逻辑运算单元（ALU）功能实现

根据 2.3 部分的分析，我们建立 ALU.v 文件，采用 case 语句对各种 ALUOp 输入对应的信号输出进行穷举，并执行对应的算术逻辑运算。其主要逻辑如图 5.3 所示。

```
always @ (input1 or input2 or aluCtr)
begin
    case(aluCtr)
    //and
    4'b0000:
        ALURes = input1 & input2;

    //or
    4'b0001:
        ALURes = input1 | input2;

    //add
    4'b0010:
        ALURes = input1 + input2;

    //left shift
    4'b0011:
        ALURes = input2 << input1;

    //right shift
    4'b0100:
        ALURes = input2 >> input1;
```

```
//nothing happened
4'b0101:
    ALURes = input1;

//sub
4'b0110:
    ALURes = input1 - input2;

//slt
4'b0111:
    ALURes = ($signed(input1) < $signed(input2));

//nor
4'b1100:
    ALURes = ~(input1 | input2);

//xor
4'b1101:
    ALURes = input1 ^ input2;

//arithmetic shift right
4'b1110:
    ALURes = ($signed(input2) >> input1);

default:
    ALURes = input1;

endcase
```

图 5.3 ALU.v 主要逻辑单元

```
if(ALURes == 0)
    Zero = 1;
else
    Zero = 0;
```

图 5.4 零输出信号（zero）设置

注意在这部分中 slt 指令为针对有符号数进行计算的指令，故在比较之前我们需要先将输入类型转换为有符号数，这里我们使用了$signed()函数进行类型转换；此外，在 ALU.v 文件中我们需要实现或非(nor)逻辑运算,这个指令实验指导书没有在 ALUCtr 部分中给出；最后，根据输出结果，我们需要对零结果信号（zero）进行设置，见图 5.4 。

5.4. 寄存器模块（Registers）功能实现

根据 2.4 部分的分析，我们建立 Registers.v 文件并设置两个 always 语句分别负责读取和写入操作，正如之前所述，由于读寄存器操作并不会改变寄存器内部的值，因此读寄存器可以在任意时间进行。因此我们选取 lab4 中介绍的实现方式 2，直接将读输出端口与寄存器模块内部绑定。

而由于不确定 WriteReg、WriteData 和 RegWrite 信号的先后次序，我们采用时钟信号的下降沿作为写操作的同步信号，防止发生错误。代码实现见图 5.5。

```
module Registers(
    input [4:0] readReg1,
    input [4:0] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    input regWrite,
    input clk,
    input reset,
    output [31:0] readData1,
    output [31:0] readData2
    );

    reg [31:0] RegFile[31:0];
    reg [31:0] ReadData1;
    reg [31:0] ReadData2;
    integer i;

    always @ (reset)
    begin

    -
      for (i = 0; i < 32; i = i + 1)
          RegFile[i] = 0;
  end

  always @ (negedge clk)
  begin
      if(regWrite)
          RegFile[writeReg] = writeData;
  end

  assign readData1 = RegFile[readReg1];
  assign readData2 = RegFile[readReg2];

endmodule
```

图 5.5 寄存器模块（Registers）实现代码

值得注意的是，我们需要对寄存器模块进行初始化，设置循环将每一个存储单元均设置为 0，防止仿真波形图由于寄存器未初始化而出现红色波形。

5.5. 数据存储器（Data Memory）功能实现

根据 2.6 部分的分析，我们建立 dataMemory.v 文件并设置两个 always 语句分别负责读取和写入操作，正如之前所述， 由于读取数据存储器操作并不会改变数据存储器内部的

值，因此读取数据存储器可以在任意时间进行。而由于不确定 Address、WriteData 和 MemWrite 信号的先后次序，我们采用时钟信号的下降沿作为写操作的同步信号，防止发生错误。代码实现见图 5.6。注意这里输出端口我们修改为了 128 位以满足高速缓存模块的需要

```
reg [31:0] memFile [0:63];
reg [127:0] ReadData;

always @(memRead or address or memWrite)
begin
    if(memRead)
        if(address < 64)
            ReadData = memFile[address];
        else
            ReadData = 0;
end

always @(negedge clk)
begin
    if(memWrite)
        if(address < 64)
            memFile[address] = writeData;
end

assign readData = ReadData;
```

图 5.6 数据存储器（Data Memory）实现代码

值得注意的是，我们需要对数据存储器进行初始化，设置循环将每一个存储单元均设置为 0，防止仿真波形图由于数据存储器未初始化而出现红色波形。

此外，为了防止 32 位地址访问到空内存，当地址大于等于 64 时，读操作将输出 0，而写操作将拒绝写入。

5.6. 有符号扩展单元功能实现

根据 2.7 部分的分析，我们建立 singext.v 文件，正如之前所述，这里对于有符号扩展我们采用将 16 位有符号数的最高位复制 16 次再与原数拼接在一起的方法实现器件，如图 5.7 所示。

```
module signext(
    input [15:0] in,
    input extSign,
    output [31:0] out
    );

    assign out = extSign ? {{16{in[15]}}, in} : { 16'h0000, in};

endmodule
```

图 5.7 有符号扩展单元实现代码

5.7.指令内存模块（Instruction Memory）功能实现

指令内存只需要根据输入 address 指示的地址取出对应的指令并输出即可，唯一需要注意的是该处理器中我们按字节寻址，而一条指令长 4 个字节，因此指令的地址均按 4 位对齐。器件主要逻辑见图 5.8

```
module instMemory(
    input [31 : 0] address,
    output [31 : 0] inst
    );

    reg [31 : 0] instFile [0 : 1023];

    assign inst = instFile[address / 4];
endmodule
```

图 5.8 指令内存模块（Instruction Memory）实现代码

5.8.多路选择器（MUX）功能实现

多路选择器的实现也较为简单，只需要根据 sel 信号的值输出对应的输入即可。我们可以使用三目运算符来实现这一功能。5 位和 32 位的多路选择器的实现代码如图 5.9 所示。

```
module Mux_32bits(
    input sel,
    input [31:0] input1,
    input [31:0] input2,
    output [31:0] out
    );

    assign out = sel ? input1 : input2;
endmodule

module Mux_5bits(
    input sel,
    input [4:0] input1,
    input [4:0] input2,
    output [4:0] out
    );

    assign out = sel ? input1 : input2;
endmodule
```

图 5.9 多路选择器（MUX）实现代码

5.9.程序计数器（Program Counter）功能实现

程序计数器的实现按照 2.10 所述，已经修改为了顶层模块中的小模块，故不在此单独展示。

5.10.高速缓存模块（Cache）功能实现

如 2.6 所述，Cache 内置数据存储器（Data Memory）。当处理器访存时，它将先从 Cache 中寻找数据，如果想要的数据已经存在于 Cache 中，则处理器可以直接从 Cache 中获取想要的数据，从而缩短该访存操作所需要的时间；而若 Cache 中不存在想要的数据，则处理器才会访问内存，获取一个 Cache Line 并将其存在 Cache 中，以便下次使用。

该 Cache 使用全相联映射，在一开始时需要对 Cache 中的各 Cache Line 和 valid 寄存器进行初始化以保证其正常功能。如图 5.10 所示。

```
reg[31:0]cacheFile[0:63];
reg Valid[0:15];
reg[25:0]tag[0:15];

reg[31:0] ReadData;
wire[127:0] MemFileData;
wire[3:0] cacheAddr = address[5:2];

integer i;

initial
begin
    for(i=0;i<64;i=i+1)
        Valid[i] = 1'b0;
        tag[i] = 16'b0;
end

always @(memRead or address or memWrite)
begin
    if(memRead)
    begin
        //cache hit
        if(Valid[cacheAddr] & tag[cacheAddr] == address[31:6])
            ReadData = cacheFile[address[5:0]];


       //cache miss
       else begin
           tag[cacheAddr] = address[31:6];
           Valid[cacheAddr] = 1'b1;
           cacheFile[{cacheAddr,2'b11}] = MemFileData[31:0];
           cacheFile[{cacheAddr,2'b10}] = MemFileData[63:32];
           cacheFile[{cacheAddr,2'b01}] = MemFileData[95:64];
           cacheFile[{cacheAddr,2'b00}] = MemFileData[127:96];
           ReadData = cacheFile[address[5:0]];
       end
   end
end


        always @(negedge clk)
        begin
            if(memWrite)
                Valid[cacheAddr] = 1'b0;
        end

        assign readData = ReadData;
    endmodule
```

图 5.10 高速缓存模块（Cache）实现代码

另外注意 Cache 的外部接口于 lab4，lab5 中数据存储器（Data Memory）的外部接口完全相同，以便 Cache 能无缝应用于其他器件。

5.11. 顶层模块（Top）功能实现

首先，我们需要声明所有的流水线段寄存器，如图 5.11 所示。

```
//pipeline registers
//IF to ID
reg [31:0] IF2ID_Inst;
reg [31:0] IF2ID_PC;


 //pipeline registers
 //ID to EX
 reg [2:0] ID2EX_ALUOp;
 reg [7:0] ID2EX_Ctr_Signal_Bus;
 reg [31:0] ID2EX_Ext_Res;
 reg [4:0] ID2EX_Reg_Rs;
 reg [4:0] ID2EX_Reg_Rt;
 reg [31:0] ID2EX_Reg_Read_Data1;
 reg [31:0] ID2EX_Reg_Read_Data2;
 reg [5:0] ID2EX_Inst_Funct;
 reg [4:0] ID2EX_Inst_Shamt;
 reg [4:0] ID2EX_Reg_Dest;
 reg [31:0] ID2EX_PC;


//pipeline registers
//EX to MA
reg [3:0] EX2MEM_Ctr_Signal_Bus;
reg [31:0] EX2MEM_ALU_Res;
reg [31:0] EX2MEM_Reg_Read_Data_2;
reg [4:0] EX2MEM_Reg_Dest;


 //MA to WB
 reg MEM2WB_Ctr_Signal_Bus;
 reg [31:0] MEM2WB_Final_Data;
 reg [4:0] MEM2WB_Reg_Dest;
```

图 5.11 顶层模块流水线段寄存器声明

在流水线每一阶段，我们需要实例化当前阶段所需要的器件并将其互相连接。这里以较简单的 MEM 阶段为例，如图 5.12 所示。除了内部的连线以外，我们还需要将外部的 clk 与 reset 信号引入系统。

```
//MA stage----------------------------------------------
//fetch signals
wire MEM_Mem_Write = EX2MEM_Ctr_Signal_Bus[3];
wire MEM_Mem_Read = EX2MEM_Ctr_Signal_Bus[2];
wire MEM_Mem_To_Reg = EX2MEM_Ctr_Signal_Bus[1];
wire MEM_Reg_Write = EX2MEM_Ctr_Signal_Bus[0];

wire [31:0] MEM_Mem_Read_Data;
Cache DataMemory(
    .clk(clk),
    .address(EX2MEM_ALU_Res),
    .writeData(EX2MEM_Reg_Read_Data_2),
    .memWrite(MEM_Mem_Write),
    .memRead(MEM_Mem_Read),
    .readData(MEM_Mem_Read_Data)
);

 wire [31:0] MA_Final_Data;
 Mux_32bits MemToRegMux(
     .sel(MEM_Mem_To_Reg),
     .input1(MEM_Mem_Read_Data),
     .input2(EX2MEM_ALU_Res),
     .out(MA_Final_Data)
 );

 //MA to WB
 reg MEM2WB_Ctr_Signal_Bus;
 reg [31:0] MEM2WB_Final_Data;
 reg [4:0] MEM2WB_Reg_Dest;
```

图 5.12 流水线访存（MEM）阶段结构

其余阶段等限于篇幅不再展示。值得一提的是关于地址的选择部分，如图 5.13 所示。其通过简单的逻辑判断根据各信号自动生成下一条指令的地址。

```
//produce address
wire EX_Beq_Branch = EX_Beq_Signal & EX_ALU_Zero;
wire EX_Bne_Branch = EX_Bne_Signal & (~ EX_ALU_Zero);

wire[31:0] NEXT_PC = EX_Bne_Branch ? Branch_Dest :
(EX_Beq_Branch ? Branch_Dest :
(ID_Jr_Signal ? ID_Reg_Read_Data1 :
(ID_Jump_Signal ? ((IF2ID_PC + 4) & 32'hf0000000) + (IF2ID_Inst [25 : 0] << 2) :
IF_PC + 4)));

wire BRANCH = EX_Beq_Branch | EX_Bne_Branch;  //decide at EX stage
```

图 5.13 顶层模块地址选择部分

关于数据转发（Forwarding）部分，其结构如图 5.14 所示，对于 ALU 的第一个操作数，我们拥有两个多路选择器，分别对应来自 MEM\WB 和 MEM\WB 流水线段寄存器转发的数据。如果依赖上一条指令，则通过 forward_A_mux2 转发，如果依赖上上条指令，则通过 forward_A_mux1 转发，如果依赖于在之前的指令，我们认为不存在数据冒险，则无需转

发，直接使用 ID2EX_Reg_Read_Data 即可。

```
wire[31:0] EX_Forwarding_1_Temp;
wire[31:0] EX_Forwarding_2_Temp;
//forward to next next stage
Mux_32bits Forward1Mux1(
    .sel(WB_Reg_Write & (MEM2WB_Reg_Dest == ID2EX_Reg_Rs)),
    .input1(MEM2WB_Final_Data),
    .input2(ID2EX_Reg_Read_Data1),
    .out(EX_Forwarding_1_Temp)
);
//forward to next stage
Mux_32bits Forward1Mux2(
    .sel(MEM_Reg_Write & (EX2MEM_Reg_Dest == ID2EX_Reg_Rs)),
    .input1(EX2MEM_ALU_Res),
    .input2(EX_Forwarding_1_Temp),
    .out(Forwarding_Res_A)
);

Mux_32bits Forward2Mux1(
    .sel(WB_Reg_Write & (MEM2WB_Reg_Dest == ID2EX_Reg_Rt)),
    .input1(MEM2WB_Final_Data),
    .input2(ID2EX_Reg_Read_Data2),
    .out(EX_Forwarding_2_Temp)
);
Mux_32bits Forward2Mux2(
    .sel(MEM_Reg_Write & (EX2MEM_Reg_Dest == ID2EX_Reg_Rt)),
    .input1(EX2MEM_ALU_Res),
    .input2(EX_Forwarding_2_Temp),
    .out(Forwarding_Res_B)
);
```

图 5.14 顶层模块数据转发（Forwarding）部分

最后是预测不转移（predict-not-taken）和停顿的实现，详见图 5.15 注释。

```
always @(posedge clk)
begin
    NOP = ID_Jump_Signal | ID_Jr_Signal | BRANCH;
   //ID_MEM_READ_SIG, 这条指令是lw; 后面的指令使用的是lw的结果, 必须要等待lw完成, 停顿
    STALL = ID2EX_Ctr_Signal_Bus[2] & ((ID2EX_Reg_Rt == ID_Inst_Rs) | (ID2EX_Reg_Rt == ID_Inst_Rt));
    if(!reset)
        // IF/ID
        if(!STALL)
        begin
            if(NOP)
            begin
                if(IF_PC == NEXT_PC)
                begin
                    //从NOP中退出
                    IF_PC <= IF_PC + 4;
                    IF2ID_PC <= IF_PC;
                    IF2ID_Inst <= IF_Inst;
                end
```

```
        else begin
            //进入NOP
            IF_PC <= NEXT_PC;
            IF2ID_PC <= 0;
            IF2ID_Inst <= 0;
        end
    end
    else begin
        //无事发生
        IF_PC <= NEXT_PC;
        IF2ID_PC <= IF_PC;
        IF2ID_Inst <= IF_Inst;
    end
end

// ID/EX
//预测的原理是，我们总是认为下一条指令是顺序执行，则跳转指令后的指令正常取指，如果发现发生
//了跳转，则将NOP置1，清空pipeline register，从而截断命令流，从跳转处指令开始执行
if (!ID_Jal_Signal)
begin
    if (STALL | NOP)
    begin

         ID2EX_PC <= IF2ID_PC;
         ID2EX_Ctr_Signal_Bus <= 0;
         ID2EX_ALUOp <= 3'b000;
         ID2EX_Ext_Res <= 0;
         ID2EX_Reg_Rs <= 0;
         ID2EX_Reg_Rt <= 0;
         ID2EX_Inst_Funct <= 0;
         ID2EX_Inst_Shamt <= 0;
         ID2EX_Reg_Read_Data1 <= 0;
         ID2EX_Reg_Read_Data2 <= 0;
         ID2EX_Reg_Dest <= 0;
     end
     else
     begin
         ID2EX_PC <= IF2ID_PC;
         ID2EX_Ctr_Signal_Bus <= ID_Ctr_Signal_Bus[7:0];
         ID2EX_ALUOp <= ID_Ctr_Signal_ALUOp;
         ID2EX_Ext_Res <= ID_Ext_Res;
         ID2EX_Reg_Rs <= ID_Inst_Rs;
         ID2EX_Reg_Rt <= ID_Inst_Rt;
         ID2EX_Inst_Funct <= ID_Inst_Funct;
         ID2EX_Inst_Shamt <= ID_Inst_Shamt;
         ID2EX_Reg_Read_Data1 <= ID_Reg_Read_Data1;
         ID2EX_Reg_Read_Data2 <= ID_Reg_Read_Data2;
         ID2EX_Reg_Dest <= ID_Reg_Dest;
     end
end

// EX/MA
if (!ID_Jal_Signal)
begin
```

```
            EX2MEM_Ctr_Signal_Bus <= ID2EX_Ctr_Signal_Bus[3:0];
            EX2MEM_Reg_Read_Data_2 <= Forwarding_Res_B;
            EX2MEM_ALU_Res <= EX_Final_Data;
            EX2MEM_Reg_Dest <= ID2EX_Reg_Dest;
        end

        // MA/WB
        if (!ID_Jal_Signal)
        begin
            MEM2WB_Ctr_Signal_Bus <= EX2MEM_Ctr_Signal_Bus[0];
            MEM2WB_Final_Data <= MA_Final_Data;
            MEM2WB_Reg_Dest <= EX2MEM_Reg_Dest;
        end
    else
        begin
            // 重置

             IF_PC = 0;
             IF2ID_Inst = 0;
             IF2ID_PC = 0;
             ID2EX_ALUOp = 0;
             ID2EX_Ctr_Signal_Bus = 0;
             ID2EX_Ext_Res = 0;
             ID2EX_Reg_Rs = 0;
             ID2EX_Reg_Rt = 0;
             ID2EX_Reg_Read_Data1 = 0;
             ID2EX_Reg_Read_Data2 = 0;
             ID2EX_Inst_Funct = 0;
             ID2EX_Inst_Shamt = 0;
             ID2EX_Reg_Dest = 0;
             EX2MEM_Ctr_Signal_Bus = 0;
             EX2MEM_ALU_Res = 0;
             EX2MEM_Reg_Read_Data_2 = 0;
             EX2MEM_Reg_Dest = 0;
             MEM2WB_Ctr_Signal_Bus = 0;
             MEM2WB_Final_Data = 0;
             MEM2WB_Reg_Dest = 0;
         end
end
```

图 5.15 顶层模块预测不转移&停顿部分

6. 结果验证

我们使用 VerilogHDL 硬件描述语言在 Vivado 中编写激励文件并进行仿真。我们使用助教提供的乘法实例程序对处理器进行仿真测试，该程序逻辑如图 6.1 所示。

```
(reset)
lw $1, 0($3)
lw $2, 4($3)
srl $4, $2, 1
sll $5, $4, 1
beq $5, $2, 2
add $8, $8, $1
srl $2, $2, 1
sll $1, $1,1
beq $2, $3,1
j 2
sw $8, 8($3)
```

图 6.1 乘法示例程序

将其转换为 MIPS 指令并根据实际情况进行修改，我们得到如图 6.2 所示二进制指令。

```
10001100011000010000000000000000
10001100011000010000000000000100
00000000000000010001000001000010
00000000000000010000101000001000000
00010000010001010000000000000001
00000001000000010100000000100000
00000000000000010000100001000010
00000000000000010000100001000000
00010000011000100000000000000001
00001000000000000000000000000010
10101100011010000000000000001000
```

图 6.2 乘法示例程序（二进制）

编写激励文件 Top_tb.v 并将图 6.2 所示指令和图 6.3 所示数据导入处理器开始运行。仿真结果如图 6.4 所示。

```
00000009
0000000D
00000002
00000003
00000004
00000005
00000006
00000007
00000008
```

图 6.3 乘法示例程序导入数据

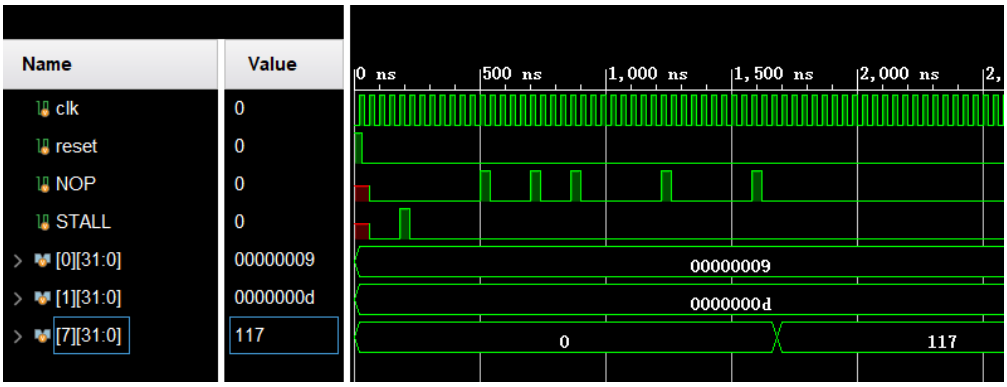

图 6.4 乘法示例程序仿真结果

由图 6.4 可知，程序成功读取了地址为 0 和 1 的 9 和 D（13）作为操作数，并将其乘法正确结果 117 存储在了地址为 2 的位置，证明程序的正确性。在这个示例程序中，几大类指令如运算（add）、条件跳转（beq），无条件跳转（j）以及逻辑位移（sll）均得到了测试，功能均正常，则其他同类型指令的正确性亦不难验证。亦可见 nop 和 stall 信号正常出现而次数不多，说明我们的数据转发和预测不转移成功提高了流水线运行的效率

7. 上板验证

7.1. 工程实现

1. 由于实验板板载 200MHz 时钟振荡器，属高频时钟，做下载验证时则需用到差分时钟以更好适应工程上的需要。原代码模块需做时钟方面的修改。创建顶层源文件 Top_Top.v 如图 7.1 所示。由于 lab6 和 lab5 只有内部实现不同，则上板前的准备工作完全相同。

```
module TopTop(
    input clk_p,
    input clk_n,
    input [3:0] a,
    input [3:0] b,
    input reset,

    output led_clk,
    output led_do,
    output led_en,

    output wire seg_clk,
    output wire seg_en,
    output wire seg_do
    );
    wire CLK_i;
    wire Clk_25M;

    IBUFGDS IBUFGDS_inst(
        .O(CLK_i),
        .I(clk_p),
        .IB(clk_n)
    );
```

```
reg [1:0] div;
reg [3:0] cnt;
always @ (posedge CLK_i)
begin
    div <= div + 1;
    cnt <= cnt + 1;
end
assign Clk_25M = div[1];

Top processor(
    .clk(Clk_25M),
    .reset(!reset),
    .a(a),
    .b(b)
    );


display display(
   .clk(Clk_25M),
   .rst(1'b0),
   .en(8'b00001111),
   .data({processor.DataMemory.memFile[2][7:0],8'b00000000 ,b, a}),
   .dot(8'b0),
   .led(~div),
   .led_clk(16'b0),
   .led_en(led_en),
   .led_do(led_do),
   .seg_clk(seg_clk),
   .seg_en(seg_en),
   .seg_do(seg_do)
   );
```

图 7.1 Top_Top.v 文件主要内容

2. 添加 display IP 核，将课程自带的 display.v 和 display.edif 添加进项目中。

3. 由于我不知道如何自动将用拨动开关表示 a 和 b 通过指令添加进 memFile，于是在这里我使用取巧的办法，修改 dataMemory.v，在每个时钟上升沿将 a 和 b 写入 memFile[0] 和 memFile[1]，从而让程序正常运行。

4. 此外，由于上板验证就无法让 FPGA 读取我磁盘内的指令，于是修改指令存储模块的实现，将指令直接写进内存，如图 6.2 所示。

```
initial begin
instFile[0] = 32'b11111111111111111111111111111111;
instFile[1] = 32'b10001100011000010000000000000000;
instFile[2] = 32'b10001100011000100000000000000001;
instFile[3] = 32'b00000000000000100010000001000010;
instFile[4] = 32'b00000000000000100001010000001000000;
instFile[5] = 32'b00010000010001010000000000000001;
instFile[6] = 32'b00000001000000010100000000100000;
instFile[7] = 32'b00000000000000100001000001000010;
instFile[8] = 32'b00000000000000010000100001000000;
instFile[9] = 32'b00010000011000100000000000000001;
instFile[10] = 32'b00001000000000000000000000000011;
instFile[11] = 32'b10101100011010000000000000000010;
end
```

图 7.2 指令存储模块写入指令

7.2. 管脚约束

添加约束文件 lab06_xdc 文件，内容如图 7.3 所示。

```
set_property PACKAGE_PIN AC18 [get_ports clk_p]
set_property IOSTANDARD LVDS [get_ports clk_p]

set_property PACKAGE_PIN AA12 [get_ports {a[3]}]
set_property PACKAGE_PIN AA13 [get_ports {a[2]}]
set_property PACKAGE_PIN AB10 [get_ports {a[1]}]
set_property PACKAGE_PIN AA10 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[3]}]

set_property PACKAGE_PIN AD10 [get_ports {b[3]}]
set_property PACKAGE_PIN AD11 [get_ports {b[2]}]
set_property PACKAGE_PIN Y12 [get_ports {b[1]}]
set_property PACKAGE_PIN Y13 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[3]}]

set_property PACKAGE_PIN N26 [get_ports led_clk]
set_property PACKAGE_PIN M26 [get_ports led_do]
set_property PACKAGE_PIN P18 [get_ports led_en]
set_property IOSTANDARD LVCMOS33 [get_ports led_clk]
set_property IOSTANDARD LVCMOS33 [get_ports led_do]
set_property IOSTANDARD LVCMOS33 [get_ports led_en]

set_property PACKAGE_PIN M24 [get_ports seg_clk]
set_property PACKAGE_PIN L24 [get_ports seg_do]
set_property PACKAGE_PIN R18 [get_ports seg_en]
set_property IOSTANDARD LVCMOS33 [get_ports seg_clk]
set_property IOSTANDARD LVCMOS33 [get_ports seg_do]
set_property IOSTANDARD LVCMOS33 [get_ports seg_en]
```

图 7.3 lab06_xdc 文件内容

7.3. 下载验证

在 Vivado 页面左侧的 Flow Navigator 中选择 Run Systhesis 进行综合，随后选择 Run Implementation 进行部署。全部完成后选择 Generate Bitstream 生成二进制码流文件。连接 FPGA 开发板并将其添加到设备列表。最后选择 Program Device 将码流文件烧录进 FPGA 开发板。结果如图 7.4 所示。

图 6.4 乘法程序上板验证

在图 7.4 中，我们将两位操作数分别通过拨动开关设置为 1001B（9）和 1101B（D），器件成功得到了结果 75H（117）并将操作数和结果显示在七位数码管中，可见器件的功能正确。

6. 总结与反思

在本次实验中，我实现了一个支持 31 条 MIPS 指令的完整类 MIPS 多周期流水线处理器。在实现过程中，我重用了 lab5 实现的类 MIPS 单周期处理器，让我免于实现很多模块进而省下了大量的时间，让我对计算机科学领域“不要重新造轮子”的理念有了更深刻的理解。

在实验的前段，我就对顶层模块的设计感到无从下手，完全不知道应该如何实现流水线，尝试进行几次连接之后把线路弄得非常混乱。之后我应用了 lab5 中通过画出处理器线路图后按照线路图连接线路的方法，这样不仅可以稳扎稳打地一步步连接线路，更方便了我后续进行器件调试。这段经历让我了解了硬件设计的基本流程，即先画出整体或局部器件线路图，再按照线路图进行连接。在完成基础的流水线后，一步步实现其余功能，并只对已经实现的功能进行测试，实验就显得简单了一些。

同时，为了对处理器进行调试，我对 MIPS 架构中 31 条指令的分段，各分段功能和行为有了进一步的理解。丰富了我对计算机系统结构理论知识的理解。

最后，在上板验证的过程中，我经过长时间的调试才最终将让 FPGA 显示出了我想要的结果，这段经历也告诉我，实践不像理论是停留在纸面上的，需要耐心与智慧才能成功，真是一段宝贵的经历！

自此，计算机系统结构实验全部 6 个 lab 都已经完成了，在这段宝贵的经历中，我从流

水灯开始，一点点从基础模块到复杂的多周期处理器，我真正体会到了计算机系统结构和硬件设计的乐趣。在此请允许我向计算机系统结构和计算机系统结构实验的老师、各位助教以及一同上课的同学们致以衷心的感谢！