计算机系统结构实验报告 Lab06：简单的类 MIPS 多周期流水化处理器实现

花振东　521021910653

2023 年 5 月 3 日

摘要

本实验在 Lab03、Lab04、Lab05 的基础上，先将支持的指令数由 16 条扩展成 31 条，然后再修改顶层设计模块以支持前向传递（Forwarding）、停顿（Stall）、预测不转移（Predict-not-taken）。这些功能帮助处理器解决了数据冒险、控制冒险和结构冒险，同时还能优化处理器的性能。最后利用 Vivado 软件，读取测试文件进行仿真实验来检验处理器是否能正确的运行。

目录

1 实验目的

本次实验有以下五个实验目的：

1. 理解 CPU Cacheline、流水线冒险（hazard）及相关性，在 Lab05 的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）机制来解决数据冒险、控制冒险和结构冒险。
3. 在 2. 的基础上，增加 Forwarding 机制来解决数据竞争，减少因数据竞争带来的流水线停顿时，提高流水线处理器的性能。
4. 在 3. 的基础上，通过 predict-not-taken 策略解决控制冒险，减少控制竞争带来的流水线停顿时，进一步提高处理器性能。
5. 在 4. 的基础上，将 CPU 支持的指令由 16 条扩充至 31 条，使处理器功能更加丰富。

2 原理分析

2.1 流水线（Pipeline）的原理

MIPS 的 CPU 包含了五级流水，分别是取指令阶段（IF）、译码阶段（ID）、执行阶段（EX）、访存阶段（MA）、写回阶段（WB）。流水线示意图如图1，本图来自 Lab06 实验指导书。接下来将详细介绍每个阶段的功能。

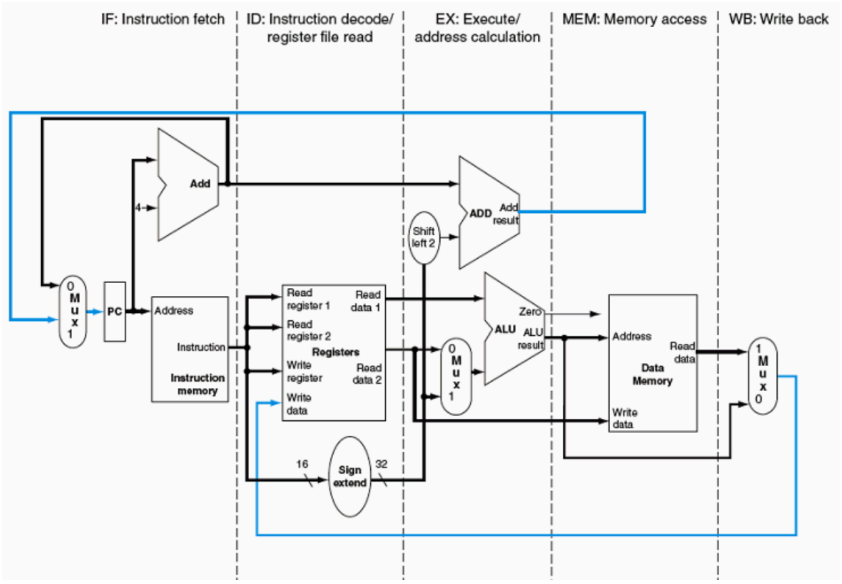

图 1: 流水线示意图

2.1.1 取指令阶段（IF）

在取指令阶段，CPU 会根据 PC 给出的地址，访问指令存储器，取出所对应的指令，并存放在 IF/ID 段寄存器中。除此以外还需要计算 PC+4 的值以更新 PC 的值。

2.1.2 译码阶段（ID）

在译码阶段，涉及到的子模块主要包括主控制器模块 (Ctr)、寄存器模块 (Register)、符号扩展模块。在这一阶段主要的功能有：主控制器（Ctr）产生控制信号，寄存器模块（Registers）读取寄存器数据，符号扩展模块（Sign Extension）将 16 位立即数扩展为 32 位。除此以外，根据 IF/ID 段寄存器中的指令，选择 rt 或者 rs 寄存器进行访问，并将指令解析从而决定是否需要进行写入操作。**需要特别注意的是**此阶段仅选择出写入寄存器，并不会实际执行写操作，选择结果会一直保存到写回阶段（WB）阶段才进行实际写入。

为了进一步提高流水线的效率，对于无条件跳转指令 j、jal、jr，其对应的所有操作都会在本阶段完成。

2.1.3 执行阶段（EX）

执行阶段，顾名思义，就是指令开始运行的阶段，涉及的子模块基本全是围绕 ALU 展开的，如 ALU 控制单元（ALUCtr），ALU 单元以及与之相关的多路选择器（Mux）。其主要功能是 ALU 会根据 ALUCtr 的输出 AluCtrOut 来对输入进行相应的算数运算。对于 beq、bne 指令，本阶段也会决定是否需要跳转。对于 lui 指令，lui 所对应的多路选择器会选取指令中立即数部分作为本阶段的执行结果的高 16 位，因此需要舍弃此时 ALU 的输出。**注：**因为我在本实验中将指令拓展为了 31 条，而新增加的指令包含了 bne 和 lui，所以需要做此说明。具体的扩展将在之后展开。

2.1.4 访存阶段（MA）

在访存阶段，主要进行内存模块的访问，主要涉及内存模块。如果该指令需要内存访问，则需要向数据存储器（Data Memory）提供地址，并保存数据存储器的输出。如果该指令为跳转指令 (beq、jr、jal 等)，则需要将跳转目标地址写入 PC，并且刷新流水线。

2.1.5 写回阶段（WB）

写回阶段主要包括寄存器模块。在执行前需要确定是否需要向寄存器写入以及写入的数据是 ALU 的运算结果还是访问内存得到的数据，但无论如何写入的数据一定会在访存阶段（MA）确定下来。除此以外，正如 2.1.2 节所描述的，写入寄存器的编号在译码阶段（ID）阶段也已经确定。

2.2 主控制器（Ctr）的原理

主控制单元根据输入指令的 [31 : 26] 位的 opCode，将指令分成 R 型（add，sub，and，or，slt）、I 型（lw，sw，beq）和 J 型（j，jal），并且输出相对应的信号至 regDst、aluSrc、memToReg、regWrite、memRead、memWrite、branch、aluOp 和 jump。考虑到此次支持的指令中包含了立即数运算，且 andi 和 ori 需要零扩展，所以还需要增加一个信号 extSign 来表示当前是否需要有符号扩展。指令中还包含了 jal，与 j 指令类似，也需要设置一个 jalSign 来表示当前指令是否是 jal。

除此以外，在 2.1.2 节提到，为了提高流水线的效率，无条件跳转指令 jr 的所有操作必须在译码阶段（ID）完成，而之前的 jrSign 是由在执行阶段（EX）才会使用的 ALUCtr 产生的。为了解决这个问题，我修改了主控制器的输出，使之能够输出 jrSign。因为本处理器支持 31 条指令，所以还添加了条件跳转指令的信号 beqSign 和 bnqSign，原来的条件跳转指令 branch 就无需存在了。对于需要舍弃 ALU 输出的 lui 指令，也需要专门的信号来表示，因此还添加了 luiSign。

根据理论课程知识可以得到输出信号的作用如下表1。

信号	作用
regDst	目标寄存器的选择信号，若为 0 则写入 rt 代表的寄存器，为 1 则写入 rd 代表的寄存器
aluSrc	取值为 0 表示使用 rt 作为输入，取值为 1 表示使用立即数作为输入
memToReg	取值为 0 表示使用 ALU 的输出作为写寄存器的输入数据，取值为 1 表示从内存中读取数据
regWrite	取值为 1 表示当前指令需要执行寄存器写入操作
memRead	取值为 1 表示当前指令需要从内存中读取数据（load）
memWrite	取值为 1 表示当前指令需要将数据写入内存（store）
aluOp[2:0]	输出至 AluCrtr 来进一步确定信号类型，将在下表中详细展开
jump	取值为 1 表示当前指令为无条件跳转指令（jump）
jalSign	取值为 1 表示当前指令为 jal 指令（jal）
extSign	取值为 1 表示当前指令需要有符号扩展
luiSign	取值为 1 表示当前指令为 lui 指令（lui）
beqSign	取值为 1 表示当前指令为 beq 指令（beq）
bneSign	取值为 1 表示当前指令为 bne 指令（bne）
jrSign	取值为 1 表示当前指令为 jr 指令（jr）

表 1: 主控制单元输出信号的作用

因为 aluOp 所需分辨的指令超过 4 条，无法继续使用 2 位二进制数来表示，因此 aluOp 需要 3 位二进制数来保存。不同的二进制位的组合对应了不同的指令操作。但是

在该指令实际运行之前，其还必须被输出至 AluCtr，最后才能被 Alu 模块所执行。在本实验中，aluOp 与 MIPS 自身的 aluOp 使用习惯保持一致，以减小之后出现差错的可能性。此处和 Lab05 保持一致，对应关系如表2。

aluOp	指令	具体操作
000	lw，sw，addi，addiu	ALU 进行加法操作
001	beq	ALU 进行减法操作
010	slti	ALU 进行有符号的比较
011	andi	ALU 进行逻辑与操作
100	ori	ALU 进行逻辑或操作
101	R	结合 Funct 域的六位进行判断
110	sltiu	ALU 进行无符号的比较
111	xori	ALU 进行逻辑异或

表 2: aluOp 与输出信号之间的对应关系

结合实验指导书，可以得出以下的 opCode 与输出信号之间的对应关系，如表3,4。

opCode	指令名称	aluOp	aluSrc	memRead	memToReg	memWrite	regDst	jrSign
000000	R-type	101	0	0	0	0	1	0
100011	lw	000	1	1	1	0	0	0
101011	sw	000	1	0	0	1	0	0
000100	beq	001	0	0	0	0	0	0
000101	bne	001	0	0	0	0	0	0
000010	j	101	0	0	0	0	0	0
000011	jal	101	0	0	0	0	0	0
000000	jr	101	0	0	0	0	0	1
001000	addi	000	1	0	0	0	0	0
001001	addiu	000	1	0	0	0	0	0
001100	andi	011	1	0	0	0	0	0
001010	xori	111	1	0	0	0	0	0
001101	ori	100	1	0	0	0	0	0
001010	slti	010	1	0	0	0	0	0
001011	sltiu	110	1	0	0	0	0	0
001111	lui	100	0	0	0	0	0	0

表 3: opCode 与输出信号之间的对应关系

opCode	指令名称	regWrite	extSign	beqSign	jump	jalSign	bneSign	luiSign
000000	R-type	1	0	0	0	0	0	0
100011	lw	1	1	0	0	0	0	0
101011	sw	0	1	0	0	0	0	0
000100	beq	0	1	1	0	0	0	0
000101	bne	0	1	0	0	0	1	0
000010	j	0	0	0	1	0	0	0
000011	jal	1	0	0	0	1	0	0
000000	jr	0	0	0	0	0	0	0
001000	addi	1	1	0	0	0	0	0
001001	addiu	1	0	0	0	0	0	0
001100	andi	1	0	0	0	0	0	0
001010	xori	1	1	0	0	0	0	0
001101	ori	1	1	0	0	0	0	0
001010	slti	1	1	0	0	0	0	0
001011	sltiu	1	0	0	0	0	0	0
001111	lui	1	0	0	0	0	0	1

表 4: opCode 与输出信号之间的对应关系

2.3 运算单元控制器（ALUCtr）的原理

算术逻辑单元 ALU 的控制单元 ALUCtr 是结合主控制器输出的 aluOp 和 Funct 字段，来正确的给每条指令输出相应的 AluCtrOut，这会决定数据在 ALU 模块中所进行的操作。**本实验为了支持 31 条指令，在此模块增加了如 addu，subu，sra，srav，sllv，srlv 等指令。同时因为 jrSign 在主控制器阶段已经产生，在此处就无需再产生 jrSign 了。**因为需要支持 srl，sll，sra 等移位指令，AluCtr 模块还必须产生 shamtSign 信号，用来表示当前指令是否为 srl、sll、sra 等需要进行移位操作的指令，如果是则为 1，且需要将指令的 shamt 段作为 ALU 的输入，表示移位的量。

ALU 控制单元的输入输出对应关系如表5。

指令操作	aluOp	Funct 域	AluCtrOut	具体操作
lw	000	XXXXXX	0010	加法操作
sw	000	XXXXXX	0010	加法操作
addi	000	XXXXXX	0010	加法操作
addiu	000	XXXXXX	0010	加法操作
beq	001	XXXXXX	0110	减法操作
slti	010	XXXXXX	0111	有符号大小比较
sltiu	110	XXXXXX	1000	无符号大小比较
andi	011	XXXXXX	0000	逻辑与运算
xori	111	XXXXXX	1011	逻辑异或运算
add	101	100000	0010	加法操作
addu	101	100001	0010	加法操作
sub	100	100010	0110	减法操作
subu	100	100011	0110	减法操作
and	101	100100	0000	逻辑与运算
or	101	100101	0001	逻辑或运算
slt	101	101010	0111	有符号大小比较
sltu	101	101011	1000	无符号大小比较
ori	100	XXXXXX	0001	逻辑或运算
sll	101	000000	0011	逻辑左移运算
sllv	101	000100	0011	逻辑左移运算
srl	101	000010	0100	逻辑右移运算
srlv	101	000110	0100	逻辑右移运算
sra	101	000011	1110	算术右移运算
srav	101	000111	1110	算术右移运算
xor	101	100110	1011	逻辑异或运算
nor	101	100111	1100	逻辑或非运算

表 5: ALUCtr 控制单元输入输出对应关系表

2.4 算术逻辑运算单元（ALU）的原理

算术逻辑单元 ALU 根据 ALUCtr 的输出信号 AluCtrOut 将两个输入执行与之对应的操作，将结果输出至 ALURes。该输出结合 branch 指令可以控制转移条件。在实际实现的过程中，如果 AluRes 的值为 0 那么 Zero 置为 1，否则为 0。AluCtrOut 的值和 ALU 操作的对应关系如表6。ALUCtrOut 的值的设定和 MIPS 指令集对于该操作的设置一样，尽可能减少自己设计会导致的一系列的问题。此模块和 Lab05 基本一致。

AluCtrOut	ALU 操作
0000	逻辑与（and）
0001	逻辑或（or）
0010	加法（add）
0011	逻辑左移（logical left shift）
0100	逻辑右移（logical right shift）
0110	减法（subtract）
0111	set on less than signed
1000	set on less than unsigned
1011	异或（xor）
1100	或非（nor）
1110	shift right arithmetic

表 6: ALU 控制单元输入操作对应表

2.5 寄存器（Register）的原理

寄存器是 CPU 内部的元件，有存储容量小、访问速度极快的特性，一般用于存储中间结果以便程序之后的快速访问。

在本实验中，一共有 32 个寄存器，因此需要 5 个二进制位来表示寄存器的编号，所以用来表示寄存器编号的输入端口 Read register1、Read register2 和 Write register 的长度均为 5。而寄存器所能存储的数据在本实验中为 32 位二进制位，因此与数据读写有关的输入输出端口 Write Data、Read data1、Read data2 的长度均为 32 位。

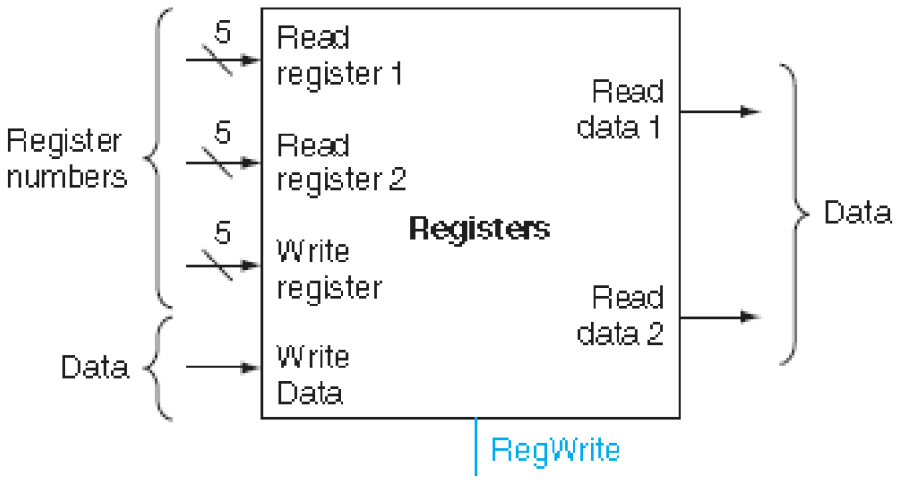

图 2: 寄存器模块的输入输出端口示意图

对于读取操作，寄存器会立刻响应，根据 Read register1 和 Read register2 中的值

以及寄存器中的数据输出相应的结果。对于写入操作，由于不确定 Write register、Write Data、RegWrite 信号的先后顺序，为了避免发生写入错误，所以将时钟的下降沿设置为写入操作的同步信号。即只有在时钟的下降沿且 regWrite 信号为高电平的时候才讲 Write Data 的值写入 Write register 所对应的寄存器。**除此以外，**还需要添加一个 Reset 信号输入，当其取值为 1 的时候要清空寄存器，为了保持同步，将清零的过程也放在时钟的下降沿和写操作合并在一起。

寄存器模块的输入输出端口示意图如图2所示。本图来自于 Lab04 的实验指导书。此模块和 Lab05 基本一致。

2.6 数据存储器（Data Memory）的原理

数据存储器的功能是存储海量数据，同时支持数据的读取与写入，其输入输出端口如图3所示。本图来自于 Lab04 的实验指导书。

值得注意的是 Address 输入端口为 32 位，可以表示 2^{32} 个存储单元。但结合操作系统的内存相关的知识，内存的容量是有限的，32 位的长度有一部分是表示页表的编号，同时由于虚拟内存的存在，导致了实际的物理内存其实远小于这个数字。在这里我仅考虑了 64 个存储单元，只有存储单元编号满足设定要求，即 0～63 才能正确访问。超出该范围的编号将不执行写操作，同时读操作也会返回 0。

在该模块中，当 MemWrite 信号为高电平时，将 WriteData 的值写入 Address 所对应的存储单元。当 MemRead 信号为高电平时，将 Address 所对应的存储单元的值输出至 ReadData。当 MemWrite 和 MemRead 信号均为低电平时，不执行任何操作。

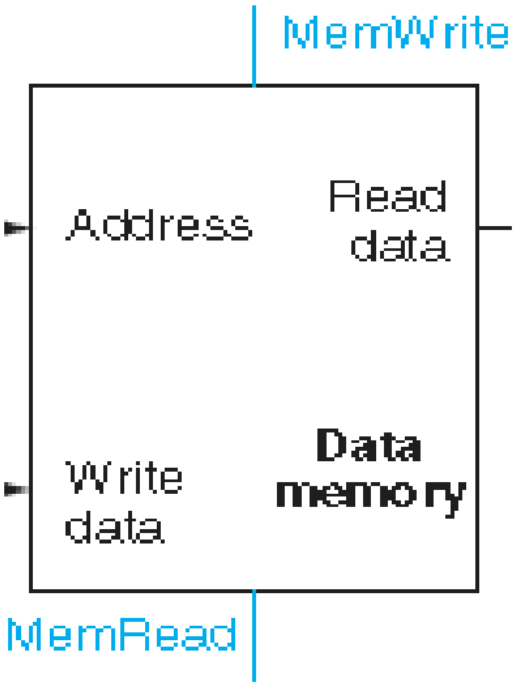

图 3: 数据存储器模块的输入输出端口示意图

2.7 有符号扩展单元（Sign Extension）的原理

有符号扩展单元的主要功能是将 16 位立即数扩展为 32 位立即数，并且根据 extSign 是否为 1 来进行有符号扩展/无符号扩展。若 extSign 为 1，那么进行有符号扩展。在输出的 32 位中，高 16 位全部都是输入的 16 位有符号数的符号位，而低 16 位就是输入的 16 位符号数本身。若 extSign 为 0，那么进行无符号扩展，输出的 32 位立即数的高 16 位全部为 0，低 16 位为原本输入的 16 位立即数。其输入输出端口如图4所示。本图来自于 Lab04 的实验指导书。此模块和 Lab05 基本一致。

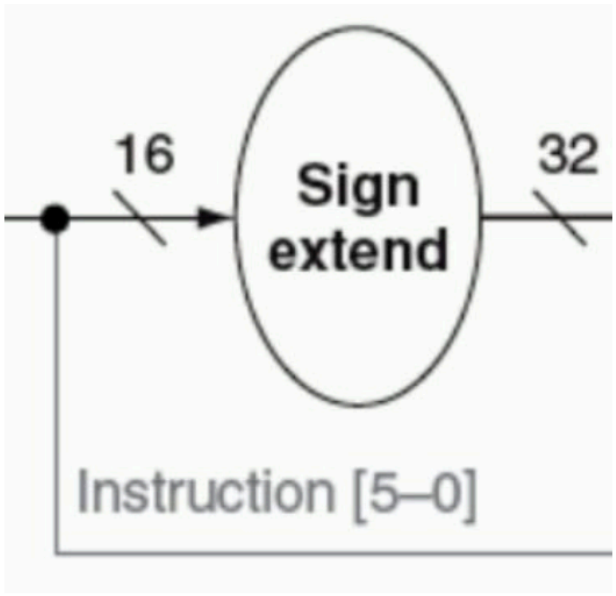

图 4: 有符号扩展单元模块的输入输出端口示意图

2.8 指令存储器（Instruction Memory）的原理

指令存储器的输入 Read address 是 32 位的地址，输出的是其所指向的一条 32 位的指令 Instruction[31 : 0]。其输入输出端口如图5所示，本图来自于 Lab05 的实验指导书。此模块和 Lab05 基本一致。

2.9 多路选择器（Mux）的原理

多路选择器的主要功能是从输入中选择其中之一输出。如图6所示，多路选择器有两个输入 INPUT1 和 INPUT2。如果另外一个输入 SEL 的取值为 1，那么输出信号 OUT 的取值为 INPUT1，反之如果 SEL 的取值为 0，那么输出信号 OUT 的取值为 INPUT0。本图来自于 Lab05 的实验指导书。此模块和 Lab05 基本一致。

在本实验中设计了两种不同的多路选择器，一种的输入输出均为 32 位，另外一种的输入输出均为 5 位。后者的使用对象是寄存器，因为一共有 32 个寄存器可以用 5 个二进制位来表示。

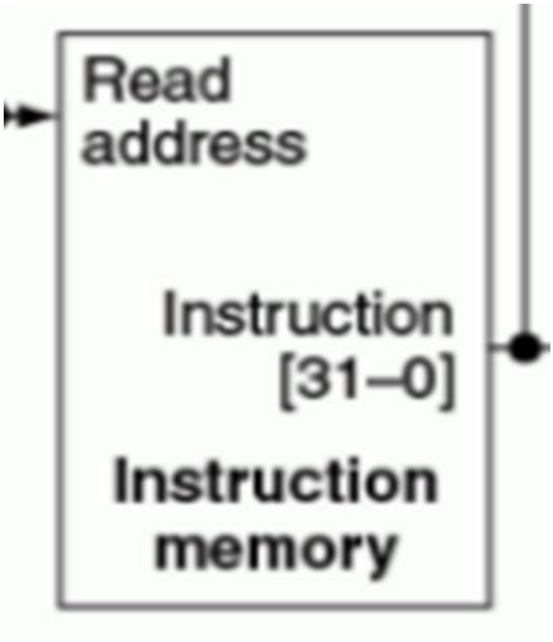

图 5: 指令存储器模块的输入输出端口示意图

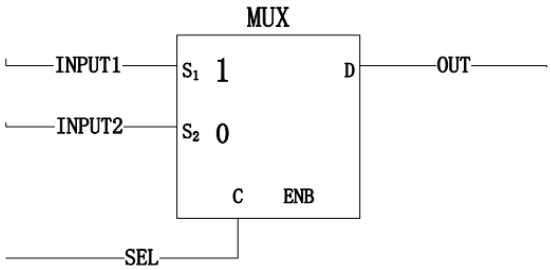

图 6: 多路选择器的输入输出端口示意图

2.10 程序计数寄存器（PC）的原理

程序计数寄存器是用来管理程序地址，即 PC（program counter）的模块。它在时钟的上升沿将输入 pcIn 写入程序计数寄存器。输出的 pcOut 是当前 PC 所在的位置，需要与程序计数寄存器的值时刻保持一致。除此以外，当 Reset 取值为 1 的时候，需要将寄存器清零。此模块和 Lab05 基本一致。

2.11 顶层模块（Top）的原理

2.11.1 基本通路

顶层模块，顾名思义就是作为统领将之前设计的所有零散的子模块串联在一起，以实现单周期处理器的功能。在顶层模块的连线中包含了数据通路和控制通路，前者用于在模块之间传输数据，后者用于在模块之间传输控制信号。两者缺一不可。如图7所示

为单周期处理器的数据通路和控制通路。[1] 实现过程中的连线将按照图中示意进行。

在本实验中，还需要在此图的基础上添加段寄存器，用于判断特定指令如 beq、bne、jr 等的多路选择器，以及为了解决数据冒险且提高流水线效率引入的前向传递机制，暂停机制以及预测不转移机制。

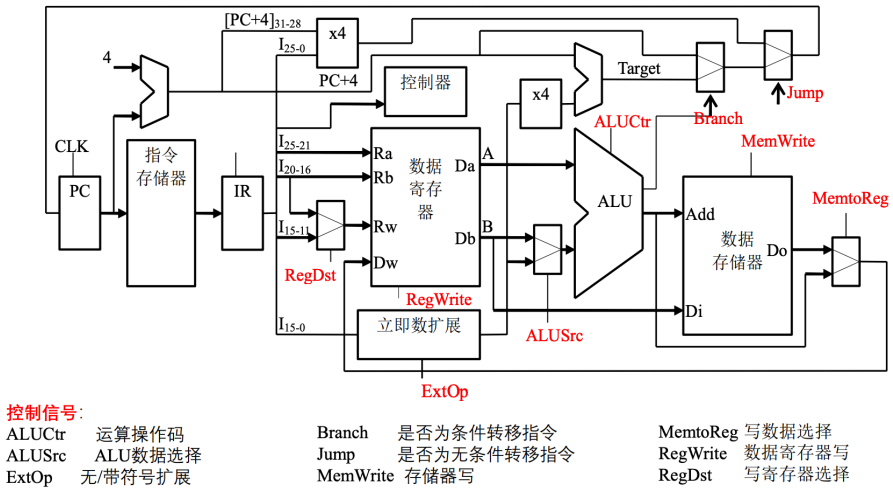

图 7: 数据通路和控制通路示意图

2.11.2 段寄存器

在 CPU 流水线的相邻两个阶段之间，需要用一个段寄存器来保存上一阶段的输出结果以及控制信号。主要有以下四个段寄存器，其功能分别与 2.1 节中每个阶段主要执行的任务相对应：

- IF/ID 段寄存器：保存 IF 阶段的输出结果以及控制信号，主要包含指令内容和 PC 的值。
- ID/EX 段寄存器：保存 ID 阶段的输出结果以及控制信号，主要包含主控制器（Ctr）产生的控制信号，符号扩展模块（Sign Extension）的 32 位立即数输出，rs、rt 寄存器的编号，写入寄存器的编号，当前 PC 的值等。
- EX/MA 段寄存器：保存 EX 阶段的输出结果以及控制信号，主要包含算术逻辑运算单元（ALU）的运算结果，写入寄存器的编号，rt 寄存器的编号等。
- MA/WB 段寄存器：保存 MA 阶段的输出结果以及控制信号，主要包含写回寄存器的数据，写入寄存器的编号，以及控制是否写入的 regWrite 信号。

段寄存器的数据通路如图8所示，本图来自 Lab06 实验指导书。

[1]图片来自邓倩妮老师课件，https://oc.sjtu.edu.cn/courses/52843/files/6621677?module_item_-id=947833

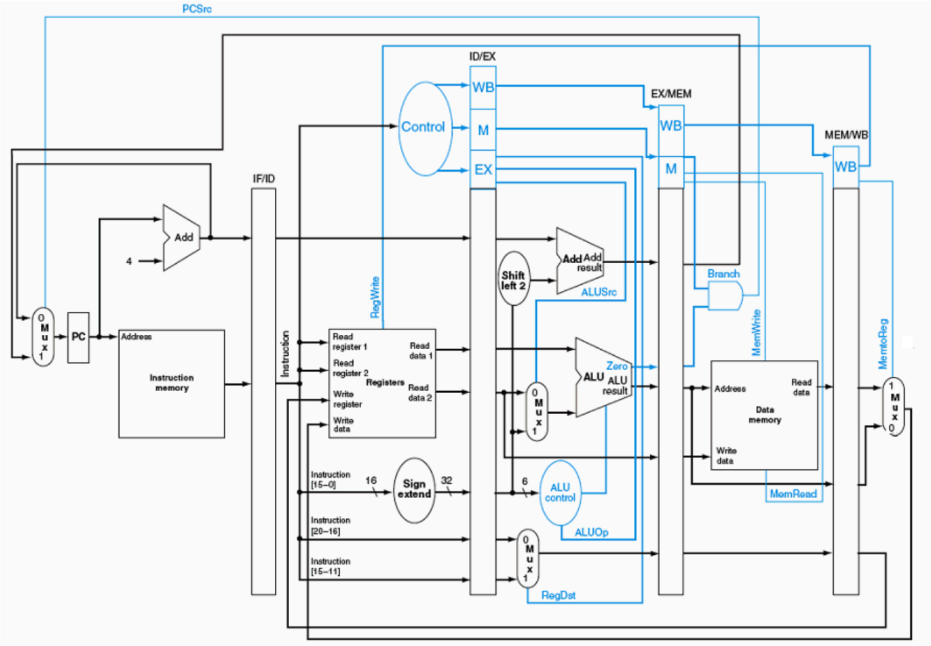

图 8: 段寄存器的数据通路

2.11.3 前向传递（Forwarding）的原理

在流水线中，当前指令在执行阶段（EX）所需要的数据可能来自先前指令的写回结果，即可能需要等到某条相关指令执行完写回操作（WB）才能继续进行。但那条指令可能才刚开始访存阶段（MA）或写回阶段（WB）。等待该指令执行完再继续执行当前指令会导致大量流水线停滞，大大降低处理器效率。通过添加前向数据通路，从 EX/MA 段寄存器、MA/WB 段寄存器就可以获得需要的数据，避免了耗时的等待，由此提高流水线效率。

2.11.4 停顿（Stall）的原理

对于“load and use”型数据冒险，即 lw 指令，前向传递无法避免停顿，这是由于 lw 指令需要在访存阶段（MA）**完成后**才能得到需要的数据，而下一条指令必须在执行阶段（EX）**开始前**获得需要的数据。因此，后一条指令必须在执行阶段（EX）开始前暂停一个周期，等待 lw 指令完成访存阶段（MA）后，使用 EX/MA 前向数据通路使得在执行阶段（EX）能够访问到所需的数据。在每条指令的译码阶段（ID），程序均会检测该指令是否和前一条指令形成“load and use”型数据冒险，如果存在冒险，则发出 STALL 信号暂停一个周期。

2.11.5 转移预测（predict-not-taken）的原理

对于跳转指令，我们通过预测不转移 (predict-not-taken) 来解决条件转移带来的控制竞争，当预测错误时，则清楚转移指令后的指令以及相关寄存器并且转移至正确的地

址执行。接下来分无条件跳转指令和条件跳转指令两种情况讨论其合理性。这种策略的正确性需要分情况说明。首先是**无条件跳转指令**，在 2.1.2 节中提到诸如 j、jal、jr 这类无条件跳转指令将在译码阶段（ID）完成所有操作，包括跳转。假设发生预测错误，那么会清除 ID/EX 寄存器。但是此时整个跳转指令已经完成，即使清除也不会造成跳转错误。其次是**条件跳转指令**。这类指令会在执行阶段（EX）完成跳转，所以当发生预测错误的时候，此时在执行阶段（EX）之后，清除取指令阶段（IF）和译码阶段（ID）的指令是合理的，会影响至多之后的两条指令。

3 功能实现

3.1 主控制器（Ctr）的功能实现

在 2.2 节中已经详细介绍了主控制器实现背后的原理。与 Lab05 中实现主控制器模块的想法相似，在这里仍然是对输入的 opCode 进行判断，然后根据不同的 opCode 输出不同的控制信号。不同之处在于这里的模块中多了 BeqSign、BneSign、LuiSign、JrSign，少了 Branch。具体代码如下，仍然使用 case 语句进行判断。因为代码十分长，所以只选择了一部分具有典型输出信号值的情况进行展示。

```
1    always @(opCode or funct)
2    begin
3        case(opCode)
4        6'b000000:                  // R type instructions
5        begin
6            RegDst = 1;
7            ALUSrc = 0;
8            MemToReg = 0;
9            MemRead = 0;
10           MemWrite = 0;
11           BeqSign = 0;
12           BneSign = 0;
13           ExtSign = 0;
14           LuiSign = 0;
15           JalSign = 0;
16           ALUOp = 3'b101;
17           JumpSign = 0;
18           if (funct == 6'b001000) begin
19               RegWrite = 0;
20               JrSign = 1;         // jr
21           end else begin
```

```
22                  RegWrite = 1;
23                  JrSign = 0;
24              end
25          end
26          6'b100011:                  // lw
27          begin
28              RegDst = 0;
29              ALUSrc = 1;
30              MemToReg = 1;
31              RegWrite = 1;
32              MemRead = 1;
33              MemWrite = 0;
34              BeqSign = 0;
35              BneSign = 0;
36              ExtSign = 1;
37              LuiSign = 0;
38              JalSign = 0;
39              ALUOp = 3'b000;
40              JumpSign = 0;
41              JrSign = 0;
42          end
43          6'b101011:                  //sw
44          begin
45              RegDst = 0;
46              ALUSrc = 1;
47              MemToReg = 0;
48              RegWrite = 0;
49              MemRead = 0;
50              MemWrite = 1;
51              BeqSign = 0;
52              BneSign = 0;
53              ExtSign = 1;
54              LuiSign = 0;
55              JalSign = 0;
56              ALUOp = 3'b000;
57              JumpSign = 0;
58              JrSign = 0;
59          end
60          6'b000100:                  // beq
```

```
61          begin
62              RegDst = 0;
63              ALUSrc = 0;
64              MemToReg = 0;
65              RegWrite = 0;
66              MemRead = 0;
67              MemWrite = 0;
68              BeqSign = 1;            // BeqSign is 1
69              BneSign = 0;
70              ExtSign = 1;
71              LuiSign = 0;
72              JalSign = 0;
73              ALUOp = 3'b001;
74              JumpSign = 0;
75              JrSign = 0;
76          end
77          6'b000101:                  // bne
78          begin
79              RegDst = 0; //x
80              ALUSrc = 0;
81              MemToReg = 0;   //x
82              RegWrite = 0;
83              MemRead = 0;
84              MemWrite = 0;
85              BeqSign = 0;
86              BneSign = 1;            // BneSign is 1
87              ExtSign = 1;
88              LuiSign = 0;
89              JalSign = 0;
90              ALUOp = 3'b001;
91              JumpSign = 0;
92              JrSign = 0;
93          end
94          6'b000010:                  // jump
95          begin
96              RegDst = 0;
97              ALUSrc = 0;
98              MemToReg = 0;
99              RegWrite = 0;
```

```
100             MemRead = 0;
101             MemWrite = 0;
102             BeqSign = 0;
103             BneSign = 0;
104             ExtSign = 0;
105             LuiSign = 0;
106             JalSign = 0;
107             ALUOp = 3'b000;
108             JumpSign = 1;           // JumpSign is 1
109             JrSign = 0;
110         end
111         6'b001000:                  // addi
112         begin
113             RegDst = 0;
114             ALUSrc = 1;
115             MemToReg = 0;
116             RegWrite = 1;
117             MemRead = 0;
118             MemWrite = 0;
119             BeqSign = 0;
120             BneSign = 0;
121             ExtSign = 1;
122             LuiSign = 0;
123             JalSign = 0;
124             ALUOp = 3'b000; //add
125             JumpSign = 0;
126             JrSign = 0;
127         end
128         6'b001110:                  // xori
129         begin
130             RegDst = 0;
131             ALUSrc = 1;
132             MemToReg = 0;
133             RegWrite = 1;
134             MemRead = 0;
135             MemWrite = 0;
136             BeqSign = 0;
137             BneSign = 0;
138             ExtSign = 0;
```

```
139             LuiSign = 0;
140             JalSign = 0;
141             ALUOp = 3'b111;
142             JumpSign = 0;
143             JrSign = 0;
144         end
145         6'b001111:                  // lui
146         begin
147             RegDst = 0;
148             ALUSrc = 0;
149             MemToReg = 0;
150             RegWrite = 1;
151             MemRead = 0;
152             MemWrite = 0;
153             BeqSign = 0;
154             BneSign = 0;
155             ExtSign = 0;
156             LuiSign = 1;            // LuiSign is 1
157             JalSign = 0;
158             ALUOp = 3'b000;
159             JumpSign = 0;
160             JrSign = 0;
161         end
162         6'b001010:                  // slti
163         begin
164             RegDst = 0;
165             ALUSrc = 1;
166             MemToReg = 0;
167             RegWrite = 1;
168             MemRead = 0;
169             MemWrite = 0;
170             BeqSign = 0;
171             BneSign = 0;
172             ExtSign = 1;
173             LuiSign = 0;
174             JalSign = 0;
175             ALUOp = 3'b010;
176             JumpSign = 0;
177             JrSign = 0;
```

```
178         end
179         default:
180         begin
181             RegDst = 0;
182             ALUSrc = 0;
183             MemToReg = 0;
184             RegWrite = 0;
185             MemRead = 0;
186             MemWrite = 0;
187             ALUOp = 2'b00;
188             JumpSign = 0;
189             JrSign = 0;
190         end
191         endcase
192     end
```

3.2 运算单元控制器（ALUCtr）的功能实现

在 2.3 节中已经详细介绍了运算单元控制器实现背后的原理。与 Lab05 中实现主控制器模块的想法相似，在这里仍然是对输入的 aluOp 和 Funct 先位拼接然后再进行判断，根据不同的组合输出不同的 AluCtrOut。不同之处在于因为 JrSign 在主控制器模块产生，所以此处不再需要了。同时因为本处理器将支持的 16 条指令扩展为 31 条，所以需要增加扩展出来的指令的判断。具体代码如下。因为拼接后存在通配符 X，所以仍然使用 casex 语句进行判断。

```
1     always @ (aluOp or funct)
2     begin
3     // Initialization of "Sign of shift amount"
4         ShamtSign = 0;
5     // switch case
6         casex({aluOp,funct})
7             9'b000xxxxxx:                   // lw, sw, add, addiu
8                 ALUCtrOut = 4'b0010;
9             9'b001xxxxxx:                   // beq, bne
10                ALUCtrOut = 4'b0110;
11            9'b010xxxxxx:                   // slti
12                ALUCtrOut = 4'b0111;
13            9'b110xxxxxx:                   // sltiu
14                ALUCtrOut = 4'b1000;
```

```
15              9'b011xxxxxx:                   // andi
16                  ALUCtrOut = 4'b0000;
17              9'b100xxxxxx:                   // ori
18                  ALUCtrOut = 4'b0001;
19              9'b111xxxxxx:                   // xori
20                  ALUCtrOut = 4'b1011;
21              9'b101001000:                   // jr
22                  ALUCtrOut = 4'b0101;
23
24              // Shift amount related are as below
25              9'b101000000:                   // sll
26              begin
27                  ALUCtrOut = 4'b0011;
28                  ShamtSign = 1;
29              end
30              9'b101000010:                   // srl
31              begin
32                  ALUCtrOut = 4'b0100;
33                  ShamtSign = 1;
34              end
35              9'b101000011:                   // sra
36              begin
37                  ALUCtrOut = 4'b1110;
38                  ShamtSign = 1;
39              end
40              9'b101100000:                   // add
41                  ALUCtrOut = 4'b0010;
42              9'b101100011:                   // subu
43                  ALUCtrOut = 4'b0110;
44              9'b101100100:                   // and
45                  ALUCtrOut = 4'b0000;
46              9'b101100101:                   // or
47                  ALUCtrOut = 4'b0001;
48              9'b101100110:                   // xor
49                  ALUCtrOut = 4'b1011;
50              9'b101100001:                   // addu
51                  ALUCtrOut = 4'b0010;
52              9'b101100010:                   // sub
53                  ALUCtrOut = 4'b0110;
```

```
            9'b101100111:                   // nor
                ALUCtrOut = 4'b1100;
            9'b101101010:                   // slt
                ALUCtrOut = 4'b0111;
            9'b101101011:                   // sltu
                ALUCtrOut = 4'b1000;
            9'b101000100:                   // sllv
                ALUCtrOut = 4'b0011;
            9'b101000110:                   // srlv
                ALUCtrOut = 4'b0100;
            9'b101000111:                   // srav
                ALUCtrOut = 4'b1110;
        endcase
    end
```

3.3 算术逻辑运算单元（ALU）的功能实现

在 2.3 节中已经详细介绍了算数逻辑运算单元实现背后的原理。与 Lab05 中实现主控制器模块的想法相似，在这里仍然是对输入的 aluCtrOut 进行判断，从而对两个输入 input1 和 input2 进行相应的算数操作。最后根据结果 ALURes 是否为 0 来判断是否需要将 Zero 置 1。具体代码如下。

值得注意的是 Verilog 会自动把 wire 类型解释为无符号数，因此想进行有符号的运算时，必须进行强制类型转换，即在需要转换的变量外面套上 `$signed()`。

```
    always @ (input1 or input2 or aluCtrOut)
    begin
        case(aluCtrOut)
        4'b0000:                        // and
            ALURes = input1 & input2;
        4'b0001:                        // or
            ALURes = input1 | input2;
        4'b0010:                        // add
            ALURes = input1 + input2;
        4'b0011:                        // left shift
            ALURes = input2 << input1;
        4'b0100:                        // right shift
            ALURes = input2 >> input1;
        4'b0110:                        // sub
            ALURes = input1 - input2;
```

```
        4'b0111:                        // slt signed
            ALURes = (input1 < input2);
        4'b1000:                        // slt unsigned
            ALURes = ($signed(input1) < $signed(input2));
        4'b1011:                        // xor
            ALURes = input1 ^ input2;
        4'b1100:                        // nor
            ALURes = ~(input1 | input2);
        4'b1110:                        // shift right arithmetic
            ALURes = ($signed(input2) >> input1);
        endcase
        if(ALURes==0)
            Zero = 1;
        else
            Zero = 0;
    end
```

3.4 寄存器（Register）的功能实现

在 2.4 节中已经详细介绍了寄存器实现背后的原理。寄存器的实现与 Lab04 基本一致，唯一的区别在于在这里需要添加 Reset 信号，当其值为 1 的时候，要清空寄存器。具体代码如下。

```
module Registers(
    input [25 : 21] readReg1,
    input [20 : 16] readReg2,
    input [4 : 0] writeReg,
    input [31 : 0] writeData,
    input regWrite,
    input reset,
    input clk,
    output reg [31 : 0] readData1,
    output reg [31 : 0] readData2
    );

    reg [31:0] RegFile[31:0];
    // Initialization of RegFile
    integer i;
    initial begin
```

```
17          RegFile[0] = 0;
18      end
19
20      always @ (readReg1 or readReg2) begin
21          begin
22          readData1 = RegFile[readReg1];
23          readData2 = RegFile[readReg2];
24          end
25      end
26
27      always @ (negedge clk or reset)
28      begin
29          if(reset)                                   // reset the register
30          begin
31              for(i = 0; i < 32; i = i + 1)
32                  RegFile[i] = 0;
33          end
34          else begin
35              if(regWrite)
36                  RegFile[writeReg] = writeData;
37          end
38      end
39
40  endmodule
```

3.5 数据存储器（Data Memory）的功能实现

在 2.5 节已经详细介绍了数据存储器的原理。对于写入操作将由时钟的下降沿进行同步，将 WriteData 的值写入 Address 所对应的存储单元。对于读操作，每当 memRead 这个读操作的标志以及 address 发生改变的时候，就需要重新读取 address 所指向的内存空间的数据，并将其输出。具体代码如下。

```
1  module dataMemory (
2      input clock,
3      input [31 : 0] address,
4      input [31 : 0] writeData,
5      input memWrite,
6      input memRead,
7      output [31 : 0] readData
```

```
);
    reg [31 : 0] memFile [0 : 63];
    reg [31 : 0] ReadData;
    integer i;

    // initialization
    initial begin
    ReadData = 0;
    for (i = 0; i < 64; i=i+1)
        memFile[i]=0;
    end

    // Read operation is performed when memRead is high
    always @ (address or memRead)
        begin
            if (memRead == 1)
                if (address <= 63)              // address less than 64 is
    valid
                    ReadData = memFile[address];
                else                            // invalid address
                    ReadData = 0;

        end

    // Write operation is performed when memWrite is high
    always @ (negedge clock)
        begin
            if (memWrite == 1 && address <= 63)
                memFile[address] = writeData;
        end

    assign readData = ReadData
    endmodule
```

3.6 有符号扩展单元（Sign Extension）的功能实现

在 2.6 节已经详细介绍了有符号扩展单元的原理。与 Lab04 中的实现类似，仍然采用三目运算符?: 来简化代码。不同之处在于需要额外加入 extSign 信号用于判断是否需要有符号扩展即可。具体代码如下。

```
    module signext(
    input [15 : 0] inst,
    input signExt,
    output [31 : 0] data
    );
    assign data = (signExt) ? {{16{inst[15]}},inst[15:0]} : {{16{0}},inst
    [15:0]};
```

3.7 指令存储器（Instruction Memory）的功能实现

在 2.7 节已经详细介绍了指令存储器的原理。它仅仅只需要根据输入的地址去输出所对应的指令即可。为了整体的鲁棒性，还需判断地址是否有效，若不在有效范围之内，那么就将结果置 0。具体代码如下。

```
module InstMemory(
        input[31 : 0] address,
        output[31 : 0] instructiion
    );

    reg [31 : 0] instFile[0 : 63];
    // check for validity of address
    assign instructiion = ((address / 4) <= 63) ? instFile[address / 4] : 0;
endmodule
```

3.8 多路选择器（Mux）的功能实现

在 2.8 节已经详细介绍了多路选择器的原理。使用三目运算符就可以轻松的实现从两个输入中选择一个输出的功能。同时支持 32 位二进制数的多路选择器和支持 5 位二进制数的多路选择器的唯一区别就在于输入的位数不同，在实现逻辑上无任何差异。具体代码如下。

```
module Mux(
    input [31:0] input0,
    input [31:0] input1,
    input select,
    output [31:0] out
    );
    assign out = select ? input1 : input0;
endmodule
```

3.9 程序计数模块（PC）的功能实现

在 2.9 节已经详细介绍了程序计数模块的原理。在实现的过程中，在时钟的上边缘将输入 pcIn 写进寄存器。除此以外，如果 reset 的值为 1，那么 PC 的值将置 0。除此以外，输出 pcOut 的值始终得和寄存器中的值保持一致，那么就需要用到 assign 语句。具体代码如下。

```
module PC(
    input [31:0] pcIn,
    input clk,
    input reset,
    output [31:0] pcOut
);

initial begin
    PC = 0;
end

always @ (posedge clk or reset)
    begin
        if(reset == 1)
            PC = 0;                 // PC needs to be reset
        else
            PC = pcIn;
    end
assign pcOut = PC;                  // pcOut keeps synchronized with PC
endmodule
```

3.10 顶层模块（Top）的功能实现

由于顶层模块的组成很复杂，我将分为段寄存器、模块实例化、前向传递的实现、流水线的实现来展开。

3.10.1 段寄存器

值得一提的是在编写代码的时候并不需要将所有的段寄存器全部定义在一起，最好是在每个阶段前定义该阶段对应的段寄存器，方便使用。

```
    // IF/ID Registers
    reg [31 : 0] IF_TO_ID_INST;
    reg [31 : 0] IF_TO_ID_PC;

    // ID/EX Registers
    reg [2 : 0] ID_TO_EX_ALUOP;
    reg [7 : 0] ID_TO_EX_CTR_SIGNALS;
    reg [31 : 0] ID_TO_EX_EXT_RES;
    reg [4 : 0] ID_TO_EX_INST_RS;
    reg [4 : 0] ID_TO_EX_INST_RT;
    reg [31 : 0] ID_TO_EX_REG_READ_DATA1;
    reg [31 : 0] ID_TO_EX_REG_READ_DATA2;
    reg [5 : 0] ID_TO_EX_INST_FUNCT;
    reg [4 : 0] ID_TO_EX_INST_SHAMT;
    reg [4 : 0] ID_TO_EX_REG_DEST;
    reg [31 : 0] ID_TO_EX_PC;

    // EX/MA Registers
    reg [3 : 0] EX_TO_MA_CTR_SIGNALS;
    reg [31 : 0] EX_TO_MA_ALU_RES;
    reg [31 : 0] EX_TO_MA_REG_READ_DATA_2;
    reg [4 : 0] EX_TO_MA_REG_DEST;

    // MA/WB Registers
    reg MA_TO_WB_CTR_SIGNALS;
    reg [31 : 0] MA_TO_WB_FINAL_DATA;
    reg [4 : 0] MA_TO_WB_REG_DEST;
```

3.10.2 子模块初始化

此部分与 Lab05 类似，模块的实例化方法仍然是用类似于.opCode(OPCODE) 的方式，这样可以避免模块内的变量顺序和实例化时的数据顺序不一致导致的问题。具体代码如下。值得注意的是跳转指令所对应的多路选择器的初始化，在 2.1 节中已经说明对于无条件跳转指令如 j、jr，跳转将在 ID 阶段完成，而对于条件跳转指令是在 EX 阶段后才执行完。因此在实例化模块时可以放在每个阶段所对应的代码段中。

```

    wire [12 : 0] ID_CTR_SIGNALS;
    wire [2 : 0] ID_CTR_SIGNAL_ALUOP;
```

```
    wire ID_JUMP_SIGN;
    wire ID_JR_SIGN;
    wire ID_EXT_SIGN;
    wire ID_REG_DST_SIGN;
    wire ID_JAL_SIGN;
    wire ID_ALU_SRC_SIGN;
    wire ID_LUI_SIGN;
    wire ID_BEQ_SIGN;
    wire ID_BNE_SIGN;
    wire ID_MEM_WRITE_SIGN;
    wire ID_MEM_READ_SIGN;
    wire ID_MEM_TO_REG_SIGN;
    wire ID_REG_WRITE_SIGN;
    wire ID_ALU_OP;
    // In ID stage, initialize Ctr module
    Ctr ctr(
        .opCode(IF_TO_ID_INST[31:26]),
        .funct(IF_TO_ID_INST[5:0]),
        .jumpSign(ID_JUMP_SIGN),
        .jrSign(ID_JR_SIGN),
        .extSign(ID_EXT_SIGN),
        .regDst(ID_REG_DST_SIGN),
        .jalSign(ID_JAL_SIGN),
        .aluSrc(ID_ALU_SRC_SIGN),
        .luiSign(ID_LUI_SIGN),
        .beqSign(ID_BEQ_SIGN),
        .bneSign(ID_BNE_SIGN),
        .memWrite(ID_MEM_WRITE_SIGN),
        .memRead(ID_MEM_READ_SIGN),
        .memToReg(ID_MEM_TO_REG_SIGN),
        .regWrite(ID_REG_WRITE_SIGN),
        .aluOp(ID_CTR_SIGNAL_ALUOP)
    );

    wire[31 : 0] ID_REG_READ_DATA1;
    wire[31 : 0] ID_REG_READ_DATA2;
    wire[4 : 0] WB_WRITE_REG_ID;
    wire[4 : 0] WB_WRITE_ID;              // Register ID used to write back
    after jal mux
```

```
42      wire[31 : 0] WB_REG_WRITE_DATA;
43      wire[31 : 0] WB_REG_DATA;              // Data written back after jal mux
44      wire WB_REG_WRITE;
45      wire [4 : 0] ID_REG_DEST;
46      wire [4 : 0] ID_REG_RS = IF_TO_ID_INST[25 : 21];
47      wire [4 : 0] ID_REG_RT = IF_TO_ID_INST[20 : 16];
48      wire [4 : 0] ID_REG_RD = IF_TO_ID_INST[15 : 11];
49
50      Mux jal_data_mux(
51          .select(ID_JAL_SIGN),
52          .input0(WB_REG_WRITE_DATA),
53          .input1(IF_TO_ID_PC + 4),
54          .out(WB_REG_DATA)
55      );
56
57      Mux jal_reg_id_mux(
58          .select(ID_JAL_SIGN),
59          .input0(WB_WRITE_REG_ID),
60          .input1(5'b11111),
61          .out(WB_WRITE_ID)
62      );
63
64      // As illustrated in 2.1.2, we need to determine the write back register
        in ID stage
65      // Register takes only 5 bits to denote, so we use 5-bit mux
66      Mux_5bits reg_dst_mux(
67          .select(ID_CTR_SIGNALS[9]),
68          .input0(ID_REG_RT),
69          .input1(ID_REG_RD),
70          .out(ID_REG_DEST)
71      );
72
73      Registers reg_file(
74          .readReg1(ID_REG_RS),
75          .readReg2(IF_TO_ID_INST[20:16]),
76          .writeReg(WB_WRITE_ID),
77          .writeData(WB_REG_DATA),
78          .regWrite(WB_REG_WRITE),
79          .clk(clk),
```

```
80          .reset(reset),
81          .readData1(ID_REG_READ_DATA1),
82          .readData2(ID_REG_READ_DATA2)
83      );
84
85      wire [31:0] ID_EXT_RES;
86      // Sign extension operation also happens in ID stage
87      signext sign_ext(
88          .inst(IF_TO_ID_INST[15:0]),
89          .signExt(ID_EXT_SIG),
90          .data(ID_EXT_RES)
91      );
92
93      wire EX_ALU_SRC_SIG = ID_TO_EX_CTR_SIGNALS[7];
94      wire EX_LUI_SIG = ID_TO_EX_CTR_SIGNALS[6];
95      wire EX_BEQ_SIG = ID_TO_EX_CTR_SIGNALS[5];
96      wire EX_BNE_SIG = ID_TO_EX_CTR_SIGNALS[4];
97
98      wire [3:0] EX_ALU_CTR_OUT;                  // ALUCtr's output in EX stage
99      wire EX_SHAMT_SIGN;                         // shift amount sign in EX
        stage
100
101     // In EX stage, modules relative to ALU will be initialized, such as
        ALUCtr, ALU
102
103     ALUCtr alu_ctr(
104         .aluOp(ID_TO_EX_ALUOP),
105         .funct(ID_TO_EX_INST_FUNCT),
106         .shamtSign(EX_SHAMT_SIGN),
107         .aluCtrOut(EX_ALU_CTR_OUT)
108     );
109
110     wire EX_ALU_ZERO;                           // Zero sign in ALU module
111     wire [31 : 0] EX_ALU_RES;                   // ALU's result
112     wire [31 : 0] EX_LUI_DATA;                  // Special case for LUI
        instruction
113     ALU alu(
114         .input1(EX_ALU_INPUT1),
115         .input2(EX_ALU_INPUT2),
```

```
116         .aluCtr(EX_ALU_CTR_OUT),
117         .aluRes(EX_ALU_RES),
118         .zero(EX_ALU_ZERO)
119     );
120 
121     // Recall in 2.1.3, we have stated that for lui, it needs to throw the
        output of ALU
122 
123     Mux lui_mux(
124         .select(EX_LUI_SIGN),
125         .input0(EX_ALU_RES),
126         .input1({ID_TO_EX_EXT_RES[15 : 0], 16'b0}),
127         .out(EX_LUI_DATA)
128     );
129 
130     // In MA stage, modules related to Memory are initialized
131 
132     wire MA_MEM_WRITE = EX_TO_MA_CTR_SIGNALS[3];
133     wire MA_MEM_READ = EX_TO_MA_CTR_SIGNALS[2];
134     wire MA_MEM_TO_REG = EX_TO_MA_CTR_SIGNALS[1];
135     wire MA_REG_WRITE = EX_TO_MA_CTR_SIGNALS[0];
136 
137 
138     wire [31:0] MA_OUTPUT_DATA;
139     // As is illustrated in 2.1.4, instruction may have access to memory in MA
         stage
140     // Need to choose between result of ALU or data from memory to output
141     Mux mem_to_reg_mux(
142         .input0(EX_TO_MA_ALU_RES),
143         .input1(MA_MEM_READ_DATA),
144         .select(MA_MEM_TO_REG),
145         .out(MA_OUTPUT_DATA)
146     );
147 
148     // The following is the Initialization of SHIFT instructions, including
        JUMP, JR, BNE, BEQ
149 
150     // Recall thayt UNCONDITIONAL SHIFT will finish in ID stage !!! Including
        J and JR !
```

```
151     wire[31 : 0] PC_AFTER_JUMP_MUX;
152     Mux jump_mux(
153         .select(ID_JUMP_SIGN),
154         .input1(((IF_TO_ID_PC + 4) & 32'hf0000000) + (IF2ID_INST [25 : 0] <<
    2)),
155         .input0(IF_PC + 4),
156         .out(PC_AFTER_JUMP_MUX)
157     );
158 
159     wire[31:0] PC_AFTER_JR_MUX;
160     Mux jr_mux(
161         .select(ID_JR_SIG),
162         .input0(PC_AFTER_JUMP_MUX),
163         .input1(ID_REG_READ_DATA1),
164         .out(PC_AFTER_JR_MUX)
165     );
166 
167     // For CONDITIONAL SHIFT, things will happen in EX stage, including BEQ
    and BNE
168     wire EX_BEQ_BRANCH = EX_BEQ_SIG & EX_ALU_ZERO;
169     wire[31 : 0] PC_AFTER_BEQ_MUX;
170     Mux beq_mux(
171         .select(EX_BEQ_BRANCH),
172         .input1(BRANCH_DEST),
173         .input0(PC_AFTER_JR_MUX),
174         .out(PC_AFTER_BEQ_MUX)
175     );
176 
177     wire EX_BNE_BRANCH = EX_BNE_SIG & (~ EX_ALU_ZERO);
178     wire[31 : 0] PC_AFTER_BNE_MUX;
179     Mux bne_mux(
180         .select(EX_BNE_BRANCH),
181         .input1(BRANCH_DEST),
182         .input0(PC_AFTER_BEQ_MUX),
183         .out(PC_AFTER_BNE_MUX)
184     );
```

3.10.3 前向传递的实现

在 2.11.3 节中已经详细说明了前向传递的原理，在实现的过程中考虑到 EX 阶段主要处理与 ALU 相关的任务，而且 ALU 有两个输入口都可能需要前向传递的数据。所以设置了两组前项通路，其中每一组中又包含两个多路选择器，分别从 EX/MA 和 MA/WB 段寄存器中读取所需数据。同时前向传递还得满足当前的读取寄存器和先前的写入寄存器是一样的前提。具体代码如下。

```
    wire[31:0] EX_FORWARDING_P1_TEMP;
    wire[31:0] EX_FORWARDING_P2_TEMP;
    // 1st forwarding mux pair
    // mux1 reads from MA/WB
    Mux forward_input1_MAWB_mux1(
        .select(WB_REG_WRITE & (MA_TO_WB_REG_DEST == ID_TO_EX_INST_RS)),
        .input0(ID_TO_EX_REG_READ_DATA1),
        .input1(MA_TO_WB_FINAL_DATA),
        .out(EX_FORWARDING_P1_TEMP)
    );
    // mux2 reads from EX/MA
    Mux forward_input1_EXMA_mux2(
        .select(MA_REG_WRITE & (EX_TO_MA_REG_DEST == ID_TO_EX_INST_RS)),
        .input0(EX_FORWARDING_A_TEMP),
        .input1(EX_TO_MA_ALU_RES),
        .out(FORWARDING_RES_P1)
    );

    // 2nd forwarding mux pair
    // mux1 reads from MA/WB
    Mux forward_input2_MAWB_mux1(
        .select(WB_REG_WRITE & (MA_TO_WB_REG_DEST == ID_TO_EX_INST_RT)),
        .input0(ID_TO_EX_REG_READ_DATA2),
        .input1(MA_TO_WB_FINAL_DATA),
        .out(EX_FORWARDING_P2_TEMP)
    );
    // mux2 reads from EX/MA
    Mux forward_input2_EXMA_mux2(
        .select(MA_REG_WRITE & (EX_TO_MA_REG_DEST == ID_TO_EX_INST_RT)),
        .input0(EX_FORWARDING_B_TEMP),
        .input1(EX_TO_MA_ALU_RES),
        .out(FORWARDING_RES_P2)
```

```
33     );
```

3.10.4 流水线的实现

在这个部分，实现了转移预测、停顿、流水线等功能。在这之中段寄存器何时清空，何时产生停顿比较重要，做如下分析。在讨论之前首先需要明确 NOP 信号是预测错误的标志，STALL 是暂停的标志。在这里需要考虑 jal 这样一种特殊的指令，它的执行过程横跨所有阶段，由它引起的暂停和其他指令的暂停对寄存器是否需要清空有不同的影响。

首先是 IF/ID 段寄存器。当 NOP 信号为 1 时，说明转移预测有误，此时需要清空 IF/ID 段寄存器。

接下来是 ID/EX 段寄存器。当 NOP 信号为 1 时，说明转移预测有误，此时需要清空此寄存器。当 STALL 信号为 1 时，说明此时需要暂停，同样也需要清空当前的寄存器，避免暂停结束后继续执行时产生错误。若此时的指令是 jal，则不需要清除 ID/EX 段寄存器。

最后是 EX/MA 段寄存器和 MA/WB 段寄存器。当指令是 jal 时，译码阶段（ID）后会暂停一个周期，此时无需清除 EX/MA、MA/WB 段寄存器。

```
1      always @(posedge clk)
2      begin
3          // NOP stands for the circumstance where the prediction is wrong.
4          // Only those Shift instructions such as j, jr, beq, bne may lead to
       wrong prediction.
5          NOP = BRANCH | ID_JUMP_SIGN | ID_JR_SIGN;
6          // STALL means that it needs to stop for a circle.
7          STALL = ID_TO_EX_CTR_SIGNALS[2] & ((ID_TO_EX_INST_RT == ID_REG_RS) | (
       ID_TO_EX_INST_RT == ID_REG_RT));
8
9          // Unless there are special cases when the registers should be emptied
       , registers should all be able to written into.
10
11         // IF/ID Register
12         if(!STALL)
13         begin
14         // when NOP = 1, IF/ID register should be emptied.
15             if(NOP)
16             begin
17                 if(IF_PC == NEXT_PC)
18                 begin
```

```
19                      IF_TO_ID_INST <= IF_INST;
20                      IF_TO_ID_PC <= IF_PC;
21                      IF_PC <= IF_PC + 4;
22                  end
23                  else begin
24                      IF_TO_ID_INST <= 0;
25                      IF_TO_ID_PC <= 0;
26                      IF_PC <= NEXT_PC;
27                  end
28              end
29              else begin
30                  IF_TO_ID_INST <= IF_INST;
31                  IF_TO_ID_PC <= IF_PC;
32                  IF_PC <= NEXT_PC;
33              end
34          end
35
36          // ID/EX Register
37          // Recall that JAL instruction is a special case when dealing with NOP
        and STALL.
38          // It does not need to empty any pipe registers.
39          if (!ID_JAL_SIG)
40          begin
41          // No matter STALL or NOP, this register should be emptied.
42              if (STALL | NOP)
43              begin
44              // For UNCONDITIONAL SHIFT like J and JR, no need to care
45              // For CONDITIONAL SHIFT like BNE and BEQ, instructions should be
        interrupted here if something wrong happens.
46                  ID_TO_EX_PC <= IF2ID_PC;
47                  ID_TO_EX_ALUOP <= 3'b000;
48                  ID_TO_EX_CTR_SIGNALS <= 0;
49                  ID_TO_EX_EXT_RES <= 0;
50                  ID_TO_EX_INST_RS <= 0;
51                  ID_TO_EX_INST_RT <= 0;
52                  ID_TO_EX_REG_READ_DATA1 <= 0;
53                  ID_TO_EX_REG_READ_DATA2 <= 0;
54                  ID_TO_EX_INST_FUNCT <= 0;
55                  ID_TO_EX_INST_SHAMT <= 0;
```

```
56                  ID_TO_EX_REG_DEST <= 0;
57              end else
58              begin
59                  ID_TO_EX_PC <= IF_TO_ID_PC;
60                  ID_TO_EX_ALUOP <= ID_CTR_SIGNAL_ALUOP;
61                  ID_TO_EX_CTR_SIGNALS <= ID_CTR_SIGNALS[7 : 0];
62                  ID_TO_EX_EXT_RES <= ID_EXT_RES;
63                  ID_TO_EX_INST_RS <= ID_REG_RS;
64                  ID_TO_EX_INST_RT <= ID_REG_RT;
65                  ID_TO_EX_REG_DEST <= ID_REG_DEST;
66                  ID_TO_EX_REG_READ_DATA1 <= ID_REG_READ_DATA1;
67                  ID_TO_EX_REG_READ_DATA2 <= ID_REG_READ_DATA2;
68                  ID_TO_EX_INST_FUNCT <= IF_TO_ID_INST[5 : 0];
69                  ID_TO_EX_INST_SHAMT <= IF_TO_ID_INST[10 : 6];
70              end
71          end
72
73          // EX/MA
74          // Again, special case for JAL instruction.
75          if (!ID_JAL_SIG)
76          begin
77              EX_TO_MA_CTR_SIGNALS <= ID_TO_EX_CTR_SIGNALS[3 : 0];
78              EX_TO_MA_ALU_RES <= EX_FINAL_DATA;
79              EX_TO_MA_REG_READ_DATA_2 <= FORWARDING_RES_B;
80              EX_TO_MA_REG_DEST <= ID_TO_EX_REG_DEST;
81          end
82
83          // MA/WB
84          // Again, special case for JAL instruction.
85          if (!ID_JAL_SIG)
86          begin
87              MA_TO_WB_CTR_SIGNALS <= EX_TO_MA_CTR_SIGNALS[0];
88              MA_TO_WB_FINAL_DATA <= MA_FINAL_DATA;
89              MA_TO_WB_REG_DEST <= EX_TO_MA_REG_DEST;
90          end
91
92      end
```

4 结果验证

在仿真验证时，我采用了微信群中分享的指令集和数据集，如图9，10所示。

```
00001000000000000000000000000100    //j
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00001100000000000000000000000110    //jal
00000000000000000000000000000000    //                  $1  $2  $3  $4
00111100000000101111111111111111    //lui $2,65535
10001100000000010000000000000010    //lw $1,2($0)       2   0   0
10001100000000010000000000000011    //lw $1,3($0)       3   0   0
10001100000000010000000000000100    //lw $2,4($0)       3   4   0
00000000001000100001100000100000    //add $3,$1,$2      3   4   7
00000000001000100001100000100100    //and $3,$1,$2      3   4   0
00100000000000110000000000000110    //addi $3,$0,6      3   4   6
00000000001000100010000000100010    //sub $4,$1,$2      3   4   6   -1
00000000000000010000100001000000    //sll $1,$1,1       6   4   6
00000000000000110001100011000000    //sll $3,$3,3       6   4   48
00000000001000110000100000100101    //or $1,$3,$1       54  4   48
00100100000000100000000000000110    //addiu $2,$0,6     54  6   48
00101000010000010000000000000001    //slti $1,$1,1      0   6   48
00110100010000010000000000000001    //ori $1,$2,1       7   6   48
00000000010000110000100000000100    //sllv $1,$3,$2     3072 6  48
00000000001000100001100000100001    //addu $3,$1,$2     3072 6  3078
00000000001000100001100000100010    //sub $3,$1,$2      3072 6  3066
00000000001000100001100000100011    //subu $3,$1,$2     3072 6  3066
00000000001000100001100000100101    //or $3,$1,$2       3072 6  3078
00000000001000100001100000100110    //xor $3,$1,$2      3072 6  3078
00000000001000100001100000100111    //nor $3,$1,$2      3072 6  -3079
00000000001000100001100000101010    //slt $3,$1,$2      3072 6  0
00000000000000010000100001000011    //sra $1,$1,1       1536 6  0
10101100000000100000000000000001    //sw $2,1($0)
00000000010000000000100000101011    //sltu $1,$0,$2     0   6   0
00000000000000010000100000001000    //srl $2,$2,2       0   3   0
00000000010000100001000000000111    //srav $2,$2,$2     0   0   0
00111000010000100000000000000010    //xori $2,$2,2      0   2   0
00101100010000010000000000000100    //sltiu $1,$2,4     1   2   0
00010000101000000000000000000001    //beq $5,$0,1
00010000101000000000000000000001    //beq $5,$0,1
00010100101000000000000000000001    //bne $5,$0,1
00000011111000000000000000001000    //jr $31
```

图 9: 仿真测试指令集

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
```

图 10: 仿真测试数据集

在编写激励代码的时候，依然使用了 Verilog 中的 `$readmemb` 函数，这个函数可以将文件中的数据读入到指定的变量中。激励代码如下。

```
module single_cycle_cpu_tb();

    reg clk;
    reg reset;

    Top top(
        .clk(clk),
        .reset(reset)
    );

    initial begin
        $readmemb("mem_inst.txt", top.inst_mem.instFile);
        $readmemb("mem_data.txt", top.memory.mem.memFile);
        reset = 1;
        clk = 0;
    end

    always #20 clk = ~clk;

    initial begin
        #30 reset = 0;
        #3000;
        $finish;
    end
endmodule
```

在运行仿真之后得到了如图11，12所示的仿真结果。

可见结果符合预期，即当前实现的支持 31 条指令的简单多周期处理器的功能正确无误。

5 总结与反思

在本实验中，我学习并实现了前向传递（Forwarding）、停顿（Stall）、预测不转移（Predict-not-taken），了解了在实现流水线处理器的时候如何解决以跳转指令为典型的可能带来数据冒险、控制冒险和结构冒险，同时对处理器的效率进行优化。除此以外，我还将本处理器支持的指令数由 16 条扩展成 31 条。主要的改动是增加了现有指令的

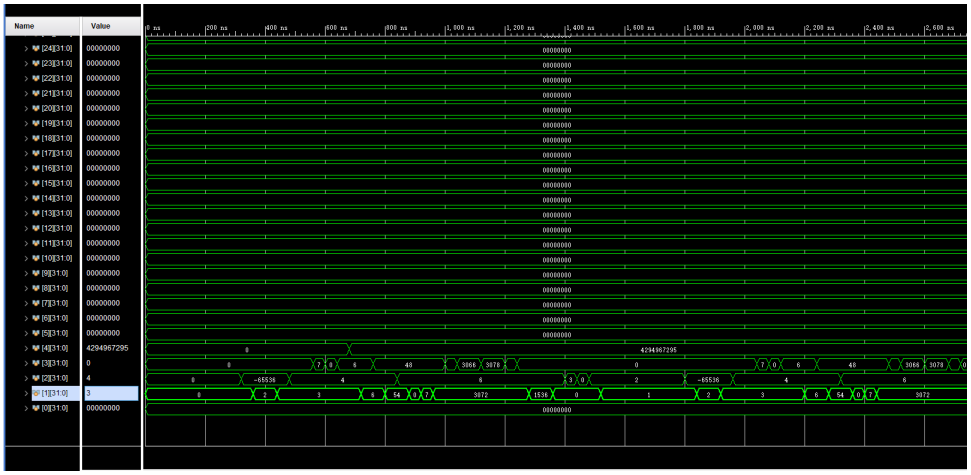

图 11: 仿真结果截图

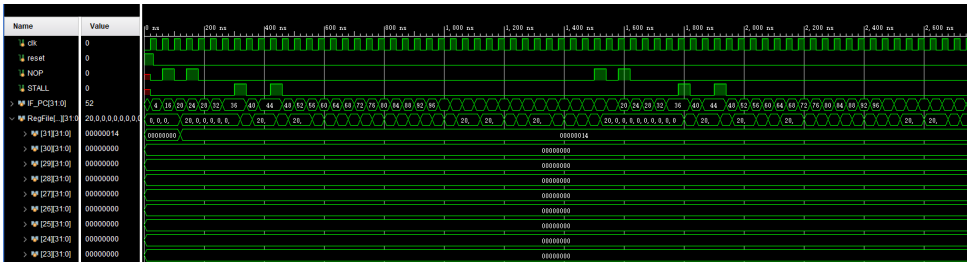

图 12: 仿真结果截图

无符号版本，这也充分利用了有符号扩展单元（Sign extension）会根据是否需要有符号扩展的性质，所以对整体的改动并不大，只需要修改主控制模块（Ctr）和运算单元控制模块（ALUCtr）即可。

经过六次课的循序渐进的铺垫与练习，我最终实现了简单的多周期流水线处理器。此次实验让我充分的意识到了模块化开发的重要性，将一个大的模块划分成若干基本子模块，利用 wire 进行模块间的通信。最后利用顶层模块串联子模块，并且实现复杂的控制逻辑，如停顿、前向传递等。

除此以外，经过六次课的练习使我对硬件编程语言 Verilog 和软件 Vivado 有了较为深入的理解，在日后想进行相关方面的研究时会更加轻松的上手。这些收获让我获益匪浅。

6 致谢

感谢刘老师和数位助教老师的耐心指导与答疑；

感谢课程相关老师提供了详细的实验指导书来辅助同学更好地理解 Verilog 语言和 Vivado 软件；

感谢电子信息与电气工程学院提供了上机编程和上板验证的环境与条件。

再次感谢每一位老师和同学！