实验报告成绩：	成绩评定日期：

2021 ~ 2022 学年秋季学期

A3705060050《计算机系统》必修课

课程实验报告

班级：人工智能 1902 班

组长：赵家栋 20191522

组员：李峻锐 20193246

李家康 20190778

报告日期：2021.12.18

目　　录

1 成员分工

姓名	工作量	工作内容
赵家栋	**40%**	• 解决基础的数据相关和 load 指令产生的数据相关； • 添加 slt, slti, sltiu, j, add, addi, sub, and, andi, nor, xori, sllv, sra, srav, srl, srlv, bgez, bgtz, blez, bltz, bltzal, bgezal, jalr 指令； • 重写乘法器为 32 周期移位乘法器，并整合乘除法器，复用转换补码模块。
李峻锐	**40%**	• 添加 hilo 寄存器、除法器、乘法器指令； • 添加 mfhi, mflo, mthi, mtlo 指令； • 解决移动指令数据相关问题； • 添加 div_mul 模块； • 简单整合除法器和乘法器。
李家康	**20%**	• 添加 ori, lui, addiu, beq, subu 指令； • 添加 jr, jal, addu, sll, or, lw, xor , sw, bne 指令； • 撰写实验报告。

2 总体设计

2.1 设计任务与要求

该《计算机系统》课程实验要求开发一个支持 MIPS 基准指令集的 MIPS 微系统。

MIPS 指令系统是在 MIPS32 指令系统的基础之上进行一定程度地裁剪，在控制系统设计规模的前提下，保证简单系统的可实现性。概要来说，这套指令系统包含所 MIPS32 中的 34 条指令，不支持 CP0 寄存器和异常，不实现 TLB MMU 和特权等级。具体的指令系统规范可以参见《“系统能力培养大赛”MIPS 指令系统规范》。

《自己动手写 CPU》用 Verilog HDL 语言，详细介绍了 OpenMIPS 的开发过程，该书可以为我们对架构和相关定义的理解提供帮助。

本次实验将结合 2020 年龙芯杯比赛的要求，利用 Vivado 并结合指定参考文献，根据现有的 mycpu.xpr 项目进行开发，完成支持 MIPS 基准指令集的 CPU 设计。

2.2 CPU 的流水线结构

一个周期需要完成一批非阻塞赋值操作。如何将整个指令的步骤划分到不同的周期中去，是需要斟酌的。如果划分成的周期所需时间不同，很有可能无法取得优化效果。

我们设计的 CPU 将使用 MIPS 架构传统的五级流水线结构，各级分别是：取指、译码、执行、访问和写回。相比单周期 CPU 的设计，流水线能够更加充分地并行利用 CPU 内部不同功能的器件，使得 CPU 执行指令更有效率。

2.2.1 取值阶段 IF

取指阶段将从外部只读储存器中读取一条指令，并更新 PC，指向下一条指令。本 CPU 访问外部储存器时按照字节寻址，每次对齐地读入 32 位。

此外该阶段还负责响应由译码阶段传来的跳转指令：译码阶段给出跳转信号与新的程序地址，该阶段相应信号并将新的地址写入 PC，随即一个时钟周期内便按照新的 PC 进行取指。

2.2.2 译码阶段 ID

指令由操作码和地址码组成。操作码表示要执行的操作性质，地址码是操作码执行时的操作对象的地址。计算机执行一条指定的指令时，必须首先分析这条指令的操作码是什么，以决定操作的性质和方法，然后才能控制计算机其他各部件协同完成指令表达的功能。这个分析工作在译码阶段完成。

2.2.3 执行阶段 EX

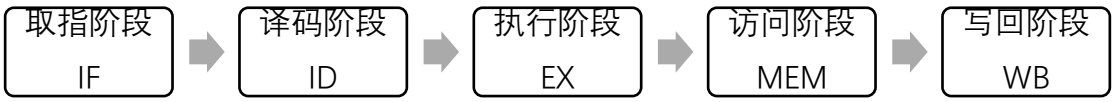
执行阶段是调用 ALU 进行真正的计算过程。本阶段将按照 ID 产生的信号对数据进行操作，即运算。CPU 支持的全部运算将在这一阶段执行，运算包括逻辑运算、位移运算、加减乘除等等。多周期运算执行时需要暂停流水线等待运算结束。

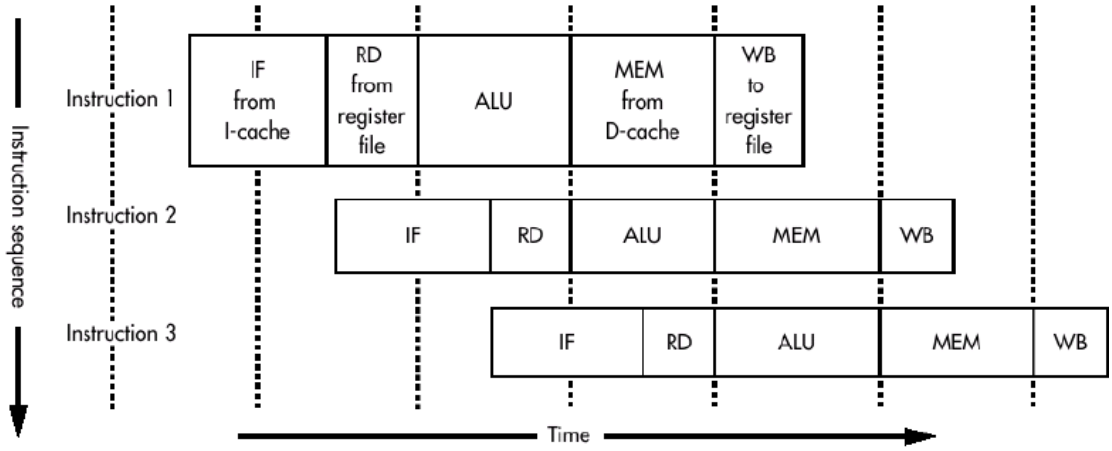
2.2.4 访问阶段 MEM

MIPS 为精简指令集，其指令设计十分简约：所有指令的操作数来源都只能来自于 CPU 内部的寄存器，只有访存指令能够将数据在内存与寄存器之间移动。因此本阶段将实际执行访存指令，与外部存储器交互。在这个阶段将会与一级的数据缓存交互。

2.2.5 写回阶段 WB

所有指令的数据写回将在该阶段完成，在这个阶段有结果的指令将会把数据存到对应的寄存器里。在空间上这个阶段并不存在实际的电路，因为实际的写入操作是在寄存器中完成的，只是在时间上单独划出一部分供寄存器写入数据。





流水线的工作原理可以在上图中形象体现：每个指令依次流过取指、译码、执行、访存、写回五个阶段，每一阶段只完成一个工作，经过五个阶段之后则一条指令被执行完毕。一般情况下每个指令将在 IF 阶段从 ROM 取出，随后在 ID 阶段（图中即 RD）译码产生控制信号，并在该阶段获取指令所需的操作数，按照指令的内容，实际运算将在 EX 阶段指令，访存类指令将在 MEM 阶段执行，最终所有指令的结果将在 WB 阶段实际写入目标寄存器。

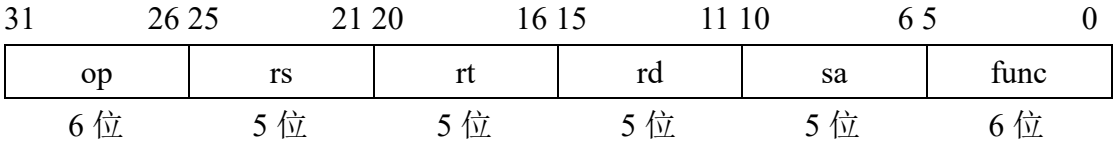
2.3 运行环境

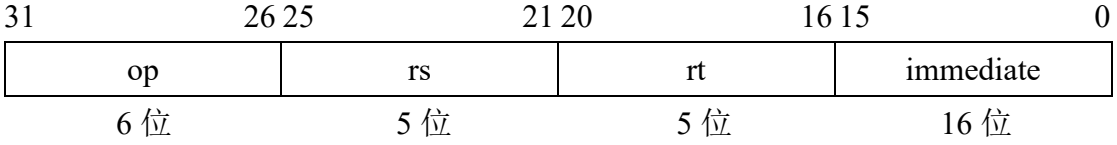
本实验基于 Verilog HDL 语言，利用 Vivado 2019.2 对 CPU 进行设计与仿真。

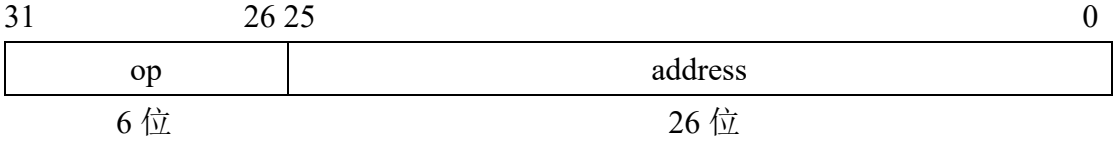
3 指令

3.1 指令格式

MIPS32 架构中所有指令都是 32 位，也就是 32 个 0、1 编码连在一起表示一条指令，有三种指令格式。如图所示。其中 op 是指令码，func 是功能码。







（1）R 类型：具体操作由 op、func 结合指定，rs 和 rt 是源寄存器的编号，rd 是目的寄存器的编号，比如：假设目的寄存器是￥3，那么对应的 rd 就是 00011（此处是二进制数）。MIPS32 架构中有 32 个通用寄存器，使用 5 位编码就可以全部表示，所以 rs、rt、rd 的宽度都是 5 位。sa 只有在移位指令中使用，用来指定移位位数。

（2）I 类型：具体操作由 op 指定，指令的低 16 位是立即数，运算时要将其扩展至 32 位，然后作为其中一个源操作数参与运算。

（3）J 类型：具体操作由 op 指定，一般是跳转指令，低 26 位是字地址，用于产生跳转的目标地址。

3.2 指令集

使用汇编程序将汇编指令翻译为计算机可以识别的 0、1 编码，也就是将汇编指令翻译为 3.1 中的指令格式，这样处理器就可以识别了。MIPS32 结构中定义的指令可以分为以下几类。

1. 逻辑操作指令

有 8 条指令：and、andi、or、ori、xor、xori、nor、lui，实现逻辑与、或、异

或、或非等运算。

2. 移位操作指令

有 6 条指令：sll、sllv、sra、srav、srl、srlv。实现逻辑左移、右移、算数右移等运算。

3. 移动操作指令

有 4 条指令：mfhi、mthi、mflo、mtlo，用于通用寄存器之间的数据移动，以及通用寄存器与 HI、LO 寄存器之间的数据移动。

4. 算术操作指令

有 15 条指令：add、addi、addiu、addu、sub、subu、slt、slti、sltiu、sltu、mul、mult、multu、div、divu。实现了加法、减法、乘法、除法等运算。

5. 转移指令

有 12 条指令：j、jr、jal、jalr、beq、bgez、bgezal、bgtz、blez、bltz、bltzal、bne。其中既有无条件转移，也有条件转移，用于程序转移到另一个地方执行。

6. 加载存储指令

有 8 条指令：lb、lbu、lh、lhu、lw、sb、sh、sw。以“l”开始的都是加载指令，以“s”开始的都是存储指令，这些指令用于从存储器中读取数据，或者向存储器中保存数据。

本实验设计的 CPU 共实现了 52 条 MIPS 指令，指令集如下表所示：

#	指令名称	功能描述
1	ORI	立即数位或
2	LUI	寄存器高半部分置立即数
3	ADDIU	加立即数(不产生溢出例外)
4	BEQ	相等转移
5	SUBU	减(不产生溢出例外)
6	JR	无条件寄存器跳转
7	JAL	无条件直接跳转至子程序并保存返回地址
8	ADDU	加(不产生溢出例外)
9	SLL	立即数逻辑左移
10	OR	位或
11	LW	取字
12	XOR	位异或
13	SW	存字
14	BNE	不等转移
15	SLTU	无符号小于设置

16	SLT	有符号小于置 1
17	SLTI	有符号小于立即数设置 1
18	SLTIU	无符号小于立即数设置 1
19	J	无条件直接跳转
20	ADD	加(可产生溢出例外)
21	ADDI	加立即数(可产生溢出例外)
22	SUB	减(可产生溢出例外)
23	AND	位与
24	ANDI	立即数位与
25	NOR	位或非
26	XORI	立即数位异或
27	SLLV	变量逻辑左移
28	SRA	立即数算术右移
29	SRAV	变量算术右移
30	SRL	立即数逻辑右移
31	SRLV	变量逻辑右
32	BGEZ	大于等于 0 转移
33	BGTZ	大于 0 转移
34	BLEZ	小于等于 0 转移
35	BLTZ	小于 0 转移
36	BLTZAL	小于 0 调用子程序并保存返回地址
37	BGEZAL	大于等于 0 调用子程序并保存返回地址
38	JALR	无条件寄存器跳转至子程序并保存返回地址下
39	MFLO	LO 寄存器至通用寄存器
40	MFHI	HI 寄存器至通用寄存器
41	MTLO	通用寄存器至 LO 寄存器
42	MTHI	通用寄存器至 HI 寄存器
43	DIV	有符号字除
44	DIVU	无符号字除
45	MULT	有符号字乘
46	MULTU	无符号字乘
47	LB	取字节有符号扩展
48	LBU	取字节无符号扩展
49	LH	取半字有符号扩展

50	LHU	取半字无符号扩展
51	SB	存字节
52	SH	存半字

3.3 指令的实现

3.3.1 以 ori 指令的实现为例

ori 指令是最简单的指令之一，是进行逻辑“或”运算的指令，其指令格式如图所示。

31 26	25 21	20 16	15 0
ORI 001101	rs	rt	immediate

从指令格式中可以知道，这是一个 I 类型的指令，ori 指令的指令码是 6’b001101，所以当处理器发现正在处理的指令的高 bit 是 6’b001101 时，就知道当前正在处理的是 ori 指令。

ori 指令的用法为：ori rs,rt,immediate，作用是将指令中的 16 位立即数 immediate 进行无符号扩展至 32 位，然后索引为 rs 的通用寄存器的值进行逻辑“或”运算，运算结果保存到索引为 rt 的通用寄存器中。

对于 2.2 的五级流水线结构，我们来分析 ori 指令在流水线中的数据处理过程。

• 取值：取出指令存储器中的指令，PC 值递增，准备取下一条指令。

• 译码：对指令进行译码，依据译码结果，从 32 通用寄存器中取出源操作数。对于 ori 指令，这里有两个 复用器，用于依据指令要求，确定参与运算的操作数，最终确定的两个操作数会送到执行阶段。

• 执行：依据译码阶段送入的源操作数、操作码，进行运算，对于 ori 指令而言，就是进行逻辑“或”运算，运算结果传递到访存阶段。

• 访存：对于 ori 指令，在访存阶段没有任何操作，直接将运算结果向下传递到回写阶段。

• 回写：将运算结果保存到目的寄存器。

在执行指令时，流水线中经常有一些被称为“相关”的情况发生，它使得指令序列中下一条指令无法按照设计的时钟周期执行，甚至无法正确的执行指令。这些“相关”会降低流水线的性能。接下来本报告将介绍我们设计的 CPU 中出现的相关问题以及解决这些问题的办法。

3.3.2 以 lw 指令的实现为例

加载指令用于从存储器中读取数据，本 CPU 中 MIPS 指令集的加载指令包含 lb、lbu、lh、lhu、lw 五条指令。

加载指令 lb、lbu、lh、lhu、lw 的格式如下图所示：

31 26 25 21 20 16 15 0

LB 100000	base	rt	offset
LBU 100100	base	rt	offset
LH 100001	base	rt	offset
LHU 100101	base	rt	offset
LW 100011	base	rt	offset

这五条加载指令可以根据指令中 26~31bit 的指令码加以区分。另外，加载指令的第 0~15bit 是 offset、第 21~15bit 是 base，加载地址的计算方法如下，先将 16 位的 offset 符号扩展至 32 位，然后与地址为 base 的通用寄存器的值相加，即可得到虚地址。

虚地址 = sighned_extended(offset) + GPR[base]

lw 指令的指令码为 6’b100011，是字加载指令。

该指令用法为：lw rt, offset(base)，lw 指令将 base 寄存器的值加上符号扩展后的立即数 offset 得到访存的虚地址，如果地址不是 4 的整数倍则触发地址错例外，否则据此虚地址从存储器中读取连续 4 个字节的值，写入到 rt 寄存器中。

接下来，本报告将以 lw 指令的实现为例，来分析在流水线中的数据处理过程。

• 取指：将 PC 值传入主存。

• 译码：从主存获取指令，得到要写入目的寄存器地址、offset 和 base。将这些信息传递到执行阶段。

• 执行：根据 offset 和 base 数据，计算虚地址。将其发送给主存。

• 访存：得到虚地址对应的数据，将该数据和 rt 寄存器的地址传递到写回阶段。

• 写回：将数据发送给 regfile 模块。

4 流水线中的数据相关问题

4.1 相关问题

流水线中经常有一些被称为“相关”的情况发生，它使得指令序列中下一条指令无法按照设计的时钟周期执行，这些“相关”会降低流水线的性能。流水线中的相关分为以下三种类型：结构相关、数据相关、控制相关。

（1）结构相关：指的是在指令执行的过程中，由于硬件资源满足不了指令执行的要求，发生硬件资源冲突而产生的相关。比如：指令和数据都共享一个存储器，在某个时钟周期，流水线既要完成某条指令对存储器中数据的访问操作，又要完成后续的取指令操作，这样就会发生存储器访问冲突，产生结构相关。

（2）数据相关：指的是在流水线中执行的几条指令中，一条指令依赖于前面指令的执行结果。

（3）控制相关：指的是流水线中的分支指令或者其他需要改写 PC 的指令造成的相关。

由于篇幅有限，报告只重点强调我们设计的 CPU 中出现的数据相关问题以及解决方法。

4.2 数据相关问题和解决办法

在我们设计的 CPU 中，众多数据相关问题只有写后读会影响流水线的正常运行。

考虑如下情况：第一条指令在 EX 阶段，计算结果要写回寄存器；第二条指令在 ID 阶段，需要使用寄存器的内容作为参数，而第一条指令的计算结果要到 WB 阶段才实际写入寄存器，而那时第二条指令已经进入 MEM 阶段，很显然已经晚了。

这种情况解决方法有两种：第一种是暂停流水线，使第二条指令在 ID 阶段等待，待第一条指令执行完 WB 后再恢复第二条指令的执行，此时第二条指令在 ID 阶段获取到的数据便是正确的；第二种方法是硬件上从 EX 阶段将写入的数据与寄存器编号反馈到 ID 阶段，在 ID 阶段对比是否有读写重叠，若有则优先使用 EX 阶段反馈来的最新数据，否则还是去寄存器堆里读。很显然第二种解决方法虽然增加了电路的复杂性，但是很大程度上节约了时间，减少了流水线空闲的时间，更具有效率。

再考虑另一种情况：第一条指令在 MEM 阶段且不是访存指令，要写入寄存器；第二条指令是 NOP，在 EX 阶段；第三条指令在 ID 阶段，需要使用寄

存器的内容作为参数。这种情况与上面的情况有些类似，解决方法也很相似：除了暂停流水线，可以在 MEM 处反馈给 EX 当前指令要写入的数据与寄存器编号，ID 先判断本条指令是否与 EX 阶段有读写重叠，没有则再检查与 MEM 的指令是否有读写重叠（这样的顺序是因为 EX 阶段产生的数据总是比 MEM 阶段的新），以此来解决数据相关的问题。

同理，若，某一条指令在 ID 阶段，需要寄存器的内容作为参数。然而写入该处数据的指令出于 WB 阶段，在 WB 段结束后才会将数据写入寄存器。此时可以通过在 WB 和 ID 间添加一条定向路径，使 ID 段可以获得需要的数据，而无需暂停。

除此以外，还存在一种情况，若第一条指令为 LW 指令，第二条指令为 XOR 指令，且 XOR 指令需要 LW 指令存入的数据，也就是 LOAD 相关问题。单单使用定向路径已经无法解决该问题，这是因为 LW 指令在 EX 段计算虚地址，MEM 段等到主存内对应的数据，故此时需要插入一个气泡，即暂停一个时钟周期。此时仍存在一些问题，在 ID 段插气泡时，相当于 ID 持续了两个时钟周期。由于 EX 段 ID 段间存在定向路径，MEM 和 ID 段也存在定向路径，且前者更加“优先”。故为了屏蔽 ID 段发送的虚地址，我们需要将前者屏蔽，从而达到获得想要的数据的目的。

5 流水段

5.1 流水段总述

在 2.2 中提到，我们设计的 CPU 使用了五级流水线结构。具体而言，五级流水线各个阶段的主要工作如下：

• 取值阶段 IF：从指令存储器读出指令，同时确定下一条指令地址。

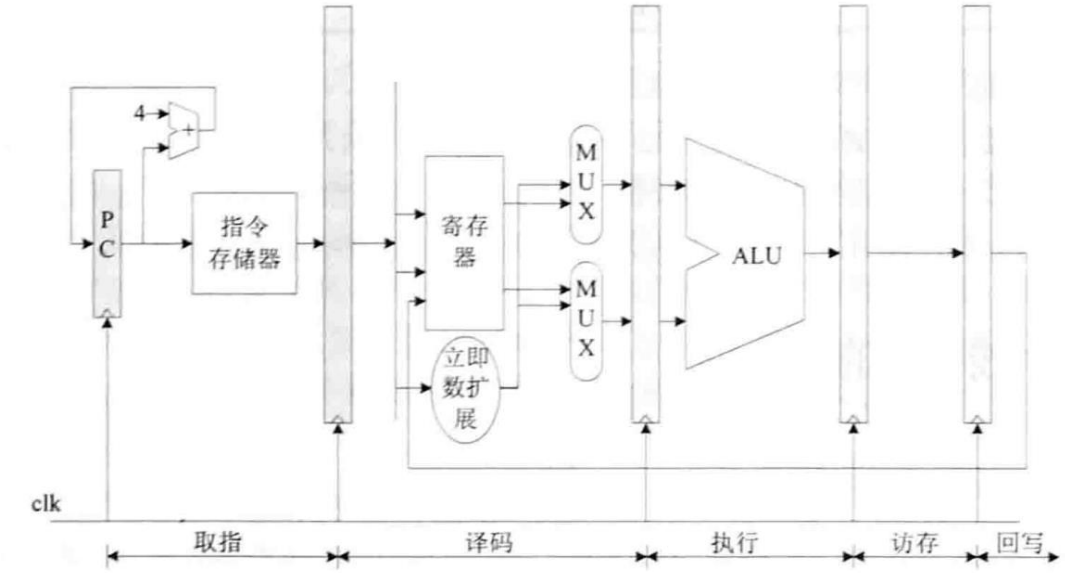
• 译码阶段 ID：对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。

• 执行阶段 EX：按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是 Load/Store 指令，那么还会计算 Load/Store 的目标地址。

• 访存阶段 MEM：如果是 Load/Store 指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行。

• 回写阶段 WB：将运算结果保存到目标寄存器。

根据这个五级流水线结构，以 ori 指令为例，我们可以得到实现 ori 指令的数据流图。



为了实现上述数据流图，我们需要完善我们的五级流水线结构。接下来，本文将详细介绍每个流水段的功能和实现原理。

5.2 取值阶段 IF

取指阶段取出指令储存器中的指令，同时 PC 值递增，准备取下一条指令。

取指阶段将从外部只读存储器中读取一条指令，并更新 PC 指向下一条指令。我们设计的 CPU 访问外部存储器时按照字节寻址，每次对齐地读入 32 位，即访问[0x00,0x03]中任意一个地址，都返回地址为 0x00~0x03 的 4 字节数据，这也是为了迎合 MIPS32 指令集中要求访存与取指的地址都必须对齐的要求。

5.2.1 PC 值的变化

这部分内容的作用在于给出指令地址，其接口描述见下表。

序号	接口名	输入/输出
1	rst	输入
2	clk	输入
3	pc_reg	输出
4	ce_reg	输出

重要代码如下：

```
assign next_pc = br_e ? br_addr : pc_reg + 32'h4;
```

当指令存储器禁用时，PC 的值保持为 0；当指令存储器能使用时，PC 的值会在每时钟周期加 4，表示下一条指令的地址，因为一条指令是 32 位，而我们设计的 CPU 是可以按照字节寻址的，一条指令对应 4 个字节，所以 PC 加 4 指向下一条指令地址。

5.3 译码阶段 ID

指令由操作码和地址码组成。操作码表示要执行的操作性质，地址码是操作码执行时的操作对象的地址。计算机执行一条指定的指令时，必须首先分析这条指令的操作码是什么，以决定操作的性质和方法，然后才能控制计算机其他各部件协同完成指令表达的功能。而这个分析工作，就是在译码阶段完成的。

总的来说，在译码阶段，将对取到的指令进行译码：给出要进行的运算类型，以及参与运算的操作数。

译码阶段包括 Regfile 模块和 hilo 模块。

5.3.1 Regfile 模块

Regfile 模块实现了 32 个 32 位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作，其接口描述见下表。

序号	接口名	输入/输出
1	rst	输入
2	clk	输入
3	waddr	输入
4	wdata	输入
5	we	输入
6	raddr1	输入
7	re1	输入
8	rdata1	输出
9	raddr2	输入
10	re2	输入
11	rdata2	输出

Regfile 模块在这里将会实现以下目标：

第一，获取 rs 寄存器和 rt 寄存器的数据。

第二，写入最终的结果。

在这个部分，根据 rs 的地址和 rt 的地址，得到 rs 和 rt 对应的值。

Regfile 模块的重要代码如下：

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
   ((ex_id_wreg == 1'b1) && (ex_id_waddr == raddr1)) ? ex_id_wdata :
   ((mem_id_wreg ==1'b1) && (mem_id_waddr == raddr1)) ? mem_id_wdata :
   ((wb_id_wreg ==1'b1) && (wb_id_waddr == raddr1)) ? wb_id_wdata :
reg_array[raddr1];

assign rdata2 = (raddr2 == 5'b0) ? 32'b0 :
   ((ex_id_wreg == 1'b1) && (ex_id_waddr == raddr2)) ? ex_id_wdata :
   ((mem_id_wreg ==1'b1) && (mem_id_waddr == raddr2)) ? mem_id_wdata :
   ((wb_id_wreg ==1'b1) && (wb_id_waddr == raddr2)) ? wb_id_wdata :
reg_array[raddr2];
```

首先判断如果数据相关的地址与当前需要的地址相同并且 wreg=1，就直接使用 wdata 的值。

如果以上条件不满足，就在寄存器中求值。

同时，Regfile 模块根据写回阶段的数据，将数据写回寄存器。

```
    always @ (posedge clk) begin
        if (we && waddr!=5'b0) begin
```

```
            reg_array[waddr] <= wdata;
        end
    end
```

5.3.2 hilo 模块

乘除法的结果是 64 位数据，而所有寄存器只能存放 32 位数据。因此乘除法结果无法存放在一个寄存器中。

为了解决这个问题，我们决定将乘除法结果的 64 位数据分成两部分，分别是高 32 位数据和低 32 位数据，分别放入 hi 寄存器和 lo 寄存器中。hilo 模块应运而生。

hilo 模块的功能与 Regfile 模块相同，它将乘法与除法计算后的 64 位数据写入 hi 寄存器和 lo 寄存器。其中高 32 位放入 hi 寄存器，低 32 位放入 lo 寄存器。其接口描述见下表。

序号	接口名	输入/输出
1	rst	输入
2	clk	输入
3	we	输入
4	hi_rdata	输入
5	lo_rdata	输入
6	hi_data1	输出
7	lo_data1	输出

hilo 模块的重要代码如下：

```
    always @(posedge clk) begin
        if (rst) begin
            hi_data1<=32'b0;
            lo_data1<=32'b0;
        end
        else if (we ==1'b1)begin
            hi_data1 <= hi_rdata;
            lo_data1 <= lo_rdata;
        end
     end
```

5.4 执行阶段 EX

执行阶段是调用 ALU 进行真正的计算过程。本阶段将按照 ID 产生的信号对数据进行操作，即运算。CPU 支持的全部运算将在这一阶段执行，运算包括逻

辑运算、位移运算、加减乘除等等。多周期运算执行时需要暂停流水线等待运算结束。

在此阶段将依据译码阶段的结果，对源操作数 1 和源操作数 2 进行指定的运算。

5.4.1 ALU 模块

ALU 模块会从译码阶段得到的运算类型、源操作数、要写的目的寄存器地址，根据这些数据进行运算。其接口描述见下表。

序号	接口名	输入/输出
1	alu_control1	输入
2	alu_src1	输入
3	alu_src2	输入
4	alu_result	输出

其中，alu_src1 和 alu_src2 是指源操作数 1 和源操作数 2，alu_result 为计算结果。

我们设计的 ALU 模块将实现以下运算：add、sub、slt、sltu、and、nor、or、xor、sll、srl、sra、lui。

实现运算的重要代码如下：

```
and    assign and_result = alu_src1 & alu_src2;
or     assign or_result = alu_src1 | alu_src2;
nor    assign nor_result = ~or_result;
xor    assign xor_result = alu_src1 ^ alu_src2;
lui    assign lui_result = {alu_src2[15:0], 16'b0};
slt    assign slt_result[31:1] = 31'b0;
       assign slt_result[0] = (alu_src1[31] & ~alu_src2[31])
                            | (~(alu_src1[31]^alu_src2[31]) & adder_result[31]);
sltu   assign sltu_result[31:1] = 31'b0;
       assign sltu_result[0] = ~adder_cout;
sll    assign sll_result = alu_src2 << alu_src1[4:0];
srl    assign srl_result = alu_src2 >> alu_src1[4:0];
sra    assign sra_result = ($signed(alu_src2)) >>> alu_src1[4:0];
add_sub  assign adder_a = alu_src1;
         assign adder_b = (op_sub | op_slt | op_sltu) ? ~alu_src2 : alu_src2;
         assign adder_cin = (op_sub | op_slt | op_sltu) ? 1'b1 : 1'b0;
         assign {adder_cout, adder_result} = adder_a + adder_b + adder_cin;
```

```
assign add_sub_result = adder_result;
```

5.4.2 div_mul 模块

div_mul 模块实现了乘法和除法的运算，且我们设计的 CPU 通过一个模块就实现了乘法与除法两种复杂运算。其接口描述见下表。

序号	接口名	输入/输出
1	rst	输入
2	clk	输入
3	dm_sighed	输入
4	opdata1_i	输入
5	opdata2_i	输入
6	start_i	输入
7	annul_i	输入
8	result_o	输出
9	ready_o	输出
10	choose_x	输入

其中 dm_sighed 表示是否为有符号除法运算。opdata_i 和 opdata2_i 分别表示被除数和除数。start_i 为是否开始运算，annul_i 为是否取消运算。result_o 为结果。ready_o 为是否结束。

div_mul 模块接收到乘除法的源操作数后，需要判断是否是有符号的乘除法。

如果是有符号的乘除法，并且传出的两个数中有负数，将其转化为补码形式。该部分内容，乘法与除法共用。转化代码如下：

```
if(dm_signed == 1'b1 && opdata1_i[31] == 1'b1) begin
        temp_op1 = ~opdata1_i + 1;
        mul_temp_op1 = {32'b0,~opdata1_i + 1};
end else begin
        temp_op1 = opdata1_i;
        mul_temp_op1 = opdata1_i;
```

接下来是乘除法的计算部分，计算部分乘法与除法分开。由于计算部分的代码过长，在这里不予展示。

计算结束以后，将结果转化为补码形式。

值得一提的是，我们设计的 CPU 通过 div_mul 一个模块就是实现了乘法与除法的运算，在设计过程中克服了乘法和除法运算周期不同带来的困难。

5.5 访存阶段 MEM

MIPS 为精简指令集，其指令设计十分简约：所有指令的操作数来源都只能

来自于 CPU 内部的寄存器，只有访存指令能够将数据在内存与寄存器之间移动。因此本阶段将实际执行访存指令，与外部存储器交互。

访存阶段 MEM 的接口如下表：

序号	接口名	输入/输出
1	clk	输入
2	rst	输入
3	data_sram_rdata	输入
4	inst_is_lb	输入
5	inst_is_lbu	输入
6	ex_to_mem_bus	输入
7	mem_to_id_bus	输出
8	mem_to_wb_bus	输出
9	ex_to_mem_hilo	输出
10	mem_to_wb_hilo	输出
11	mem_to_ex_hilo	输出

5.5.1 加载指令

加载指令 lb、lbu、lh、lhu 分别实现以下功能：

lb 指令是字节加载指令。从内存中指定的加载地址处，读取一个字节，然后符号扩展至 32 位，保存到地址位 rt 的通用寄存器中。

lbu 指令是无符号字节加载指令。从内存中指定的加载地址处，读取一个字节，然后无符号拓展至 32 位，保存到地址为 rt 的通用寄存器中。

lh 指令是半字节加载指令。从内存中指定的加载地址处，读取一个半字，然后符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为 0。

lhu 指令无符号半字节加载指令。从内存中指定的加载地址处，读取一个半字，然后无符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为 0。

以上 4 个加载指令的实现代码如下：

```
assign rf_wdata = sel_rf_res ? mem_result :
    inst_is_lb ? (choose_b == 2'b00 ? yituo:
                  choose_b == 2'b01 ? ertuo:
                  choose_b == 2'b10 ? santuo:
                                      situo):
    inst_is_lbu ? (choose_b == 2'b00 ? lingyituo:
```

```
                choose_b == 2'b01 ? lingertuo:
                choose_b == 2'b10 ? lingsantuo:
                                    lingsituo):
    inst_is_lh ? (choose_a ==2'b00 ? tuo:tuo1):
    inst_is_lhu ? (choose_a ==2'b00 ? tuo2:tuo3):
    (data_ram_en & (data_ram_wen == 4'b0000)) ? data_sram_rdata:ex_result;
```

5.6 写回阶段 WB

所有指令的数据写回将在该阶段完成，在这个阶段有结果的指令将会把数据存到对应的寄存器里。在空间上这个阶段并不存在实际的电路，因为实际的写入操作是在寄存器中完成的，只是在时间上单独划出一部分供寄存器写入数据。

实际上，在 WB 阶段做了数据准备工作，真正的“写回”，是在 WB 阶段结束以后，通过 Regfile 模块实现写回，具体代码可以参考 regfile 模块。

写回阶段 WB 的接口如下表：

序号	接口名	输入/输出
1	clk	输入
2	rst	输入
3	mem_to_wb_bus	输入
4	wb_to_rf_bus	输出
5	wb_to_id_bus	输出
6	debug_wb_pc	输出
7	debug_wb_rf_wen	输出
8	debug_wb_rf_wnum	输出
9	debug_wb_rf_wdata	输出
10	mem_to_wb_hilo	输入
11	wb_to_ex_hilo	输出
12	hi_rdata	输出
13	lo_rdata	输出
14	hilo_e	输出

6 心得感受与改进意见

6.1 赵家栋

本次计算机系统课程实验需要我们小组合力完成一个基于 MIPS 指令集的五级流水线 CPU 初步设计，并至少通过 64 个检查点。在本次实验中，我主要负责数据相关问题、load 相关和部分指令的添加。在刚开始实验时，由于对理论的掌握不够熟练，对助教提供的代码无从下手。在助教的讲解和不断研读代码之后，我逐渐掌握了添加指令的方法，随后我又通过添加定向路径的方法解决了数据相关，最后我又通过添加气泡解决了 load 相关问题。通过这次实验，不仅让我对流水线中的 IF、ID、EX、WEM、WB 各流水线段的工作原理以及各种指令在各段的具体实现有了更深的理解；同时也理解了流水线在实际应用中的实现方法和理论知识的区别。

6.2 李峻锐

本次课设让我对于流水线的功能和流程有了更深的认识和了解，取指，译码，执行，访存，回写，流水线还在每个阶段之间加入寄存器，接受并流水一个阶段产生的所有数据。在工作时流水线也带来分支冲突和数据冲突的问题，对于分支冲突，用延迟槽来解决这个问题，将一条与分支无关的指令放入分支指令的后面，让其也参与正常流水；对于数据冲突，在此基本上遇到的全是 RAW 冲突，对于 load 指令和移动指令来说会导致这样的问题，对此我们添加数据旁路，用定向技术将后阶段计算完成的数据转发给前面需要使用到此数据的阶段，以解燃眉之急。在设计过程中学会了如何接线，如何添加指令，如何解决冲突问题，尽管以前从未接触过相关硬件学习，出现很多错误，但在小组成员的努力合作，积极解决问题，在助教和老师的帮助下，成功完成了课设任务，学习了更多关于 CPU 设计的知识。

5.3 李家康

经历了本次计算机系统实验，我深刻明白了“知道与做到之间是有着巨大且难以逾越的鸿沟的”。懂得 CPU 的工作原理，并不一定可以真正动手做出一个 CPU。我也发现了：只有在真正的实践过程过，才能发现更多的问题，从而提高自己的知识面和能力。这次实验以后，我学习了一门新的语言 Verilog，了解了 Vivado 的使用，解决了很多在学习理论知识时遇到的问题，例如：如何解决数据相关问题等等。

借着写实验报告的机会，我也理清了 CPU 设计的脉络。询问队友负责的部分时，很感谢队友认真为我讲解原理和实践时的细节，让我对流水线有了更加深刻的认识和形象的理解。

7 参考资料

[1] 《自己动手做 CPU》雷思磊

[2] 《计算机体系结构 第二版》张晨曦等

[3] 《“系统能力培养大赛”MIPS 指令系统规范》龙芯杯比赛资料

[4] 《vivado 安装说明》龙芯杯比赛资料

[5] 《vivado 使用说明》龙芯杯比赛资料