计算机系统结构实验报告
实验 6

2021 年 6 月 23 日

摘要

本实验以实验 3、4 实现的功能模块以及实验 5 实现的类 MIPS 单周期处理器为基础，对部分功能模块进行了修改，增添高速缓存模块，并将支持的指令扩展到 31 条。此外，为实现流水线处理器，本实验增加了段寄存器，使用前向通路 (forwarding) 与流水线停顿 (stall) 来解决流水线冒险，通过预测不转移 (predict-not-taken) 策略提高流水线性能。最后，本实验将通过软件仿真的形式让处理器运行指令，以此进行实验结果的验证。

目录

1 实验目的

1. 理解 CPU Pipeline，了解流水线冒险 (hazard) 及相关性，设计基础流水线 CPU
2. 设计支持停顿的流水线 CPU，通过检测竞争并插入停顿 (Stall) 机制解决数据冒险、控制竞争和结构冒险
3. 增加前向传递机制解决数据竞争，减少因数据竞争带来的流水线停顿延时
4. 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
5. 将 CPU 支持的指令数量从 16 条扩充为 31 条，使处理器功能更加丰富
6. 设计并实现高速缓存
7. 功能仿真

2 原理分析

2.1 功能模块原理分析

2.1.1 主控制器模块

主控制器 (Ctr) 的输入为指令的操作码 (opCode) 字段，主控制器模块对操作码进行译码，向 ALU 控制器、寄存器、数据选择器等部件输出正确的控制信号。

本实验中，主控制器模块可以识别 R 型指令和 MIPS 指令集中其他所有指令并输出对应的控制信号。相比于实验 5 中的主控制器模块，为了提高流水线执行 JR 指令的性能，需要在指令译码 (ID) 阶段完成 JR 指令的识别，而实验 5 中 jrSign 信号由 ALUCtr 产生，无法在 ID 阶段完成，所以在本实验中，我们调整了主控制器模块的功能，使之产生 jrSign 信号。

主控制器模块产生的控制信号及说明如表1所示。表1中 aluOp 信号代表的含义如表2所示。

信号	内部寄存器	具体说明
regDst	RegDst	目标寄存器的选择信号；低电平：rt 寄存器；高电平：rd 寄存器
aluSrc	ALUSrc	ALU 第二个操作数来源选择信号；低电平：rt 寄存器值，高电平：立即数拓展结果
memToReg	MemToReg	写寄存器的数据来源选择信号；低电平：ALU 运算结果，高电平：内存读取结果
regWrite	RegWrite	寄存器写使能信号，高电平说明当前指令需要进行寄存器写入
memRead	MemRead	内存读使能信号，高电平有效
memWrite	MemWrite	内存写使能信号，高电平有效
aluOp	ALUOp	3 位信号，发送给运算单元控制器 (ALUCtr) 用来进一步解析运算类型的控制信号
extSign	ExtSign	带符号扩展信号，高电平将对立即数进行带符号扩展
luiSign	LuiSign	载入立即数指令 (LUI) 信号，高电平说明当前指令为 LUI 指令
jumpSign	JumpSign	无条件跳转指令 (J) 信号，高电平说明当前指令是 J 指令
jrSign	JrSign	寄存器无条件跳转指令 (JR) 信号，高电平说明当前指令是 JR 指令
jalSign	JalSign	跳转并链接指令 (JAL) 信号，高电平说明当前指令是 JAL 指令
beqSign	BeqSign	条件跳转指令 (BEQ) 信号，高电平说明当前指令是 BEQ 指令
bneSign	BneSign	条件跳转指令 (BNE) 信号，高电平说明当前指令是 BNE 指令

表 1: 主控制器产生的控制信号

aluOp 的信号内容	指令	具体说明
101	R	ALUCtr 结合指令 Funct 段决定最终操作
000	lw,sw,addi,addiu	ALU 执行加法
001	beq	ALU 执行减法
011	andi	ALU 执行逻辑与
100	ori	ALU 执行逻辑或
111	xori	ALU 执行逻辑异或
010	slti	ALU 执行带符号数大小比较
110	sltiu	ALU 执行无符号数大小比较

表 2: aluOp 信号的具体含义以及解析方式

主控制器模块 (Ctr) 产生的各种控制信号与指令 OpCode 段的对应方式如表 3 所示。

OpCode	指令	aluOp	aluSrc	memRead	memToReg	memWrite	regDst	regWrite	extSign	beqSign	bneSign	jumpSign	jalSign	jrSign	luiSign
000000	R 型指令	101	0	0	0	0	1	1	0	0	0	0	0	0	0
100011	lw	000	1	1	1	0	0	1	1	0	0	0	0	0	0
101011	sw	000	1	0	0	1	0	0	1	0	0	0	0	0	0
000100	beq	001	0	0	0	0	0	0	1	1	0	0	0	0	0
000101	bne	001	0	0	0	0	0	0	1	0	1	0	0	0	0
000010	j	000	0	0	0	0	0	0	0	0	0	1	0	0	0
000011	jal	000	0	0	0	0	0	1	0	0	0	1	1	0	0
000000	jr	101	0	0	0	0	1	0	0	0	0	0	0	1	0
001000	addi	000	1	0	0	0	0	1	1	0	0	0	0	0	0
001001	addiu	000	1	0	0	0	0	1	0	0	0	0	0	0	0
001100	andi	011	1	0	0	0	0	1	0	0	0	0	0	0	0
001010	xori	111	1	0	0	0	0	1	1	0	0	0	0	0	0
001101	ori	100	1	0	0	0	0	1	1	0	0	0	0	0	0
001010	slti	010	1	0	0	0	0	1	1	0	0	0	0	0	0
001011	sltiu	110	1	0	0	0	0	1	0	0	0	0	0	0	0
001111	lui	000	0	0	0	0	0	1	0	0	0	0	0	0	1

表 3: 各指令对应的主控制器 (Ctr) 控制信号

2.1.2 ALU 控制器模块

ALU 控制器模块 (ALUCtr) 接受指令中 Funct 段以及来自 Ctr 模块的 aluOp 信号，输出 aluCtr 信号，aluCtr 信号直接决定 ALU 模块进行的计算操作。

ALUCtr 信号输出与 ALUOp 及 Funct 的对应关系如表4所示。

除上表所示之外，本实验中 ALU 控制器模块还负责产生 shamtSign 信号，当指令为 sll、srl、sra 时 shamtSign 信号为高电平，其余情况下为低电平。

如2.1.1节中所述，本实验中 ALU 控制器模块不再产生 jrSign 信号。

2.1.3 ALU 模块

ALU 模块接受 aluCtr 信号，并根据此信号选择执行对于的 ALU 计算功能。ALU 功能与 ALUCtr 信号的对应关系如表5所示。

ALU 模块产生的输出包括 32 位的运算结果以及 1 位 zero 信号；当运算结果为 0 时，zero 处于高电平，其余时候 zero 处于低电平。

指令	ALUOp	Funct	ALUCtr	说明
lw,sw,addi,addiu	000	xxxxxx	0010	ALU 执行加法运算
beq	001	xxxxxx	0110	ALU 执行减法运算
stli	010	xxxxxx	0111	ALU 执行带符号数大小比较
stliu	110	xxxxxx	1000	ALU 执行无符号数大小比较
andi	011	xxxxxx	0000	ALU 执行逻辑与运算
ori	100	xxxxxx	0001	ALU 执行逻辑或运算
xori	111	xxxxxx	1011	ALU 执行逻辑异或运算
add	101	100000	0010	ALU 执行加法运算
addu	101	100001	0010	ALU 执行加法运算
sub	101	100010	0110	ALU 执行减法运算
subu	101	100011	0110	ALU 执行减法运算
and	101	100100	0000	ALU 执行逻辑与运算
or	101	100101	0001	ALU 执行逻辑或运算
xor	101	100110	1011	ALU 执行逻辑异或运算
nor	101	100111	1100	ALU 执行逻辑或非运算
slt	101	101010	0111	ALU 执行带符号数大小比较
sltu	101	101011	1000	ALU 执行无符号数大小比较
sll	101	000000	0011	ALU 执行逻辑左移运算
sllv	101	000100	0011	ALU 执行逻辑左移运算
srl	101	000010	0100	ALU 执行逻辑右移运算
srlv	101	000110	0100	ALU 执行逻辑右移运算
sra	101	000011	1110	ALU 执行算术右移运算
srav	101	000111	1110	ALU 执行算术右移运算

表 4: 运算单元控制器 (ALUCtr) 的输入与输出关系

ALUCtr	ALU 功能
0000	AND
0001	OR
0010	add
0011	Left Shift (logic)
0100	Right Shift (logic)
0110	sub
0111	set on less than (signed)
1000	set on less than (unsigned)
1011	XOR
1100	NOR
1110	Right Shift (arithmetic)

表 5: ALU 执行功能与 ALUCtr 信号的对应方式

2.1.4 寄存器模块

寄存器 (Register) 是中央处理器内用来暂存指令、数据和地址的存储器。寄存器的存贮容量有限，读写速度非常快。在计算机体系结构里，寄存器存储在已知时间点所作计算的中间结果，通过快速地访问数据来加速计算机程序的运行。

寄存器模块的输入与输出信号如表6所示。

输入信号	长度	说明
readReg1	5	读取寄存器 1 的编号
readReg2	5	读取寄存器 2 的编号
writeReg	5	写入寄存器编号
writeData	32	写入的数据
regWrite	1	写使能信号
clk	1	时钟信号
reset	1	初始化信号
输出信号	长度	说明
readData1	32	读取寄存器 1 的结果
readData2	32	读取寄存器 2 的结果

表 6: 寄存器模块输入、输出信号

对于读取操作，寄存器模块会根据寄存器选择信号 readReg1、readReg2 和寄存器内容立即响应并输出结果。对于写入操作，寄存器模块会在时钟下降沿且 regWrite 信号为高电平时，将 writeData 值写入 writeReg 指定的寄存器。

2.1.5 高速缓存模块

高速缓存 (Cache) 是用于减少处理器访问内存所需平均时间的部件，其容量远小于内存，但速度却可以接近处理器的频率。

本实验中的高速缓存采用全相联映射策略，块大小为 128 位，总共 16 个数据块，数据缓存总容量为 64 个字。对于读操作，首先检查 valid 位是否有效以及 tag 是否与内存地址一直，如果缓存命中，则直接返回缓存，如果未命中，则从内存中读取一个块的内容，覆盖缓存中对应块。对于写操作，采用直写策略，直接写入内存，同时将缓存中对应块 valid 位清空。

高速缓存的内存地址映射策略如图1所示。

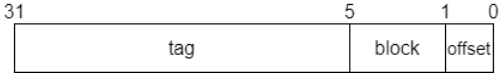

图 1: 高速缓存地址映射策略

高速缓存模块输入输出信号如表7所示。

本实验中高速缓存模块的输入输出与实验 5 中内存模块输入输出完全一致，这样的设计出于兼容性考虑，对于外部电路而言，使用高速缓存与直接使用内存模块在接口上没有差异，便于我们将本高速缓存模块安装在其他系统上。

输入信号	长度	说明
address	32	内存地址
writeData	32	写入数据
memWrite	1	内存写使能信号
memRead	1	内存读使能信号
clk	1	时钟信号
输出信号	长度	说明
readData	32	内存读取结果

表 7: 高速缓存模块输入、输出信号

2.1.6 内存单元模块

内存 (Memory) 是计算机的重要部件之一，也称内存储器和主存储器，它用于暂时存放 CPU 中的运算数据，与硬盘等外部存储器交换的数据。它是外存与 CPU 进行沟通的桥梁，计算机中所有程序的运行都在内存中进行，内存性能的强弱影响计算机整体发挥的水平。

在本实验中，内存单元模块按字进行寻址，同时，我们不考虑页表、虚拟地址与物理地址的转化，只考虑直接操作物理地址的情况。实验中我们的内存大小设定为 1024 个字，但是为了契合一般的处理器设计，我们的内存模块依然能接受 32 位地址输入，但是仅当内存地址小于 1024 时，内存操作才是有效的。

与实验 5 不同，本实验中我们加入了高速缓存，为了与高速缓存模块对接，本实验中的内存单元模块将会一次性返回 4 个字的内容。

内存模块输入输出信号如表8所示。

输入信号	长度	说明
address	32	内存地址
writeData	32	写入数据
memWrite	1	内存写使能信号
memRead	1	内存读使能信号
clk	1	时钟信号
输出信号	长度	说明
readData	128	内存读取结果

表 8: 内存模块输入、输出信号

与寄存器模块即时响应读取操作不同，内存模块仅在 memRead 处于高电平时才会进行读取操作。写入操作与寄存器模块相似，内存模块会在时钟下降沿且 memWrite 信号为高电平时，将 writeData 值写入 address 地址。

2.1.7 符号扩展模块

符号扩展模块可以根据主控制器模块 (Ctr) 的信号，以带符号扩展或无符号扩展对来自指令的 16 位数进行扩展，扩展结果为一个 32 位的数。

符号扩展模块的输入输出信号如表9所示。

输入信号	长度	说明
inst	16	指令中的立即数
signExt	1	高电平代表进行带括号扩展，否则无符号扩展
输出信号	长度	说明
data	32	扩展结果

表 9: 符号扩展模块输入、输出信号

2.1.8 数据选择器模块 (Mux/RegMux)

数据选择器模块接受两个输入信号和一个选择信号，产生一个输出信号。本实验中，我们使用了两种数据选择器，包括 Mux 和 RegMux。Mux 的输入与输出信号均为 32 位，用于对进行数据进行选择。RegMux 的输入和输出信号为 5 位，用于寄存器选取信号的选择。

数据选择器的电路模型如图2所示。

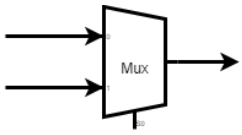

图 2: 数据选择器的电路模型

2.1.9 指令内存模块 (InstMem)

本实验中处理器采用哈佛架构，指令内存与数据内存分离。指令内存模块 (InstMem) 接受一个 32 位地址输入，输出一条 32 位指令。

2.2 流水线阶段原理分析

2.2.1 取指令阶段 (IF)

取指令阶段主要包括 PC 寄存器与指令内存模块，此阶段指令内存模块根据 PC 寄存器的值取出指令。

2.2.2 译码阶段 (ID)

译码阶段主要包括主控制器模块 (Ctr)、寄存器模块 (Register)、符号扩展模块。主控制器产生控制信号，寄存器模块读取寄存器数据，符号扩展模块将立即数扩展为 32 位。同时，在此阶段目标寄存器选择器还会在 rt 与 rd 中选择出写入寄存器，此阶段仅选择出写入寄存器后并不会执行写操作，选择结果会一直保存到 WB 阶段进行实际写入。提前完成写入寄存器选择工作，既便于之后的数据保存、传输，也方便进行数据冒险判断与前向通路实现。

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，以此来提高流水线的执行效率。

2.2.3 执行阶段 (EX)

执行阶段主要包括 ALU 控制器 (ALUCtr)、ALU 以及一系列用于选择 ALU 输入数据的选择器。本阶段中 ALU 会根据 ALU 控制器的控制信号与输入数据计算出结果，对于 beq、bne 指令，本阶段也会决定是否跳转。

对于 lui 指令，lui 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，ALU 的计算结果在此指令下会被丢弃。

2.2.4 访存阶段 (MA)

访存阶段主要包括高速缓存模块，高速缓存模块与内存模块相连，用以加速内存操作。对于需要进行访存的指令，将在本阶段完成访存操作。本阶段还有一个数据选择器，该选择器根据 MEM_TO_REG 控制信号，在访存结果与执行阶段结果中选择一个数据作为访存阶段的结果。

2.2.5 写回阶段 (WB)

写回阶段主要包括寄存器模块，对于需要寄存器写入的指令，本阶段将完成寄存器写操作。其中，写入寄存器的编号在 ID 阶段已经确定，写入的数据为 MA 阶段的结果。

2.3 顶层模块 (top) 原理分析

2.3.1 段寄存器

在流水线的两个阶段之间，需要通过段寄存器临时保存上一阶段的执行结果和控制信号。

- IF-ID 段寄存器：主要包含当前指令内容及 PC
- ID-EX 段寄存器：主要包含控制信号、符号扩展的结果，rs、rt 对应寄存器的编号，目标写入寄存器的编号，funct、shamt 的值、当前指令 PC
- EX-MA 段寄存器：主要包含控制信号、ALU 的运算结果、rt 寄存器的编号以及目标写入寄存器的编号
- MA-WB 段寄存器：主要包含 regWrite 控制信号、最终写入寄存器的数据以及目标写入寄存器的编号

控制信号在段之间的传递关系如图3所示。

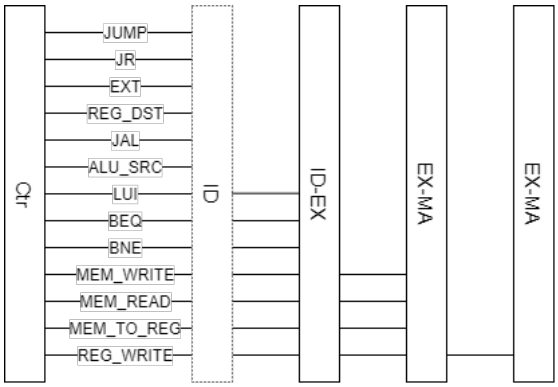

图 3: 控制信号的传递关系

2.3.2 数据前向传递原理分析

在流水线中，当前指令在 EX 阶段所需要的数据可能来自先前指令的写回结果，但是被依赖的指令可能才刚开始 MA 阶段或 WB 阶段。尽管当前指令可以等待依赖指令完全执行结束再从寄存器中获得需要的数据，但是这样的策略会导致大量流水线停滞，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 段寄存器获得需要的数据，可以不需要等待先前指令完成 WB 阶段，由此提高效率。

2.3.3 停顿机制原理分析

对于“读内存-使用”型数据冒险，前向传递无法避免停顿，这是由于 lw 指令需要在 MA 阶段完成后才能得到需要的数据，而下一条指令必须在 EX 阶段开始前获得需要的数据。因此，后一条指令必须在 EX 阶段开始前等待一个周期，待 lw 指令完成 MA 阶段后，使用前向数据通路将访存结果送回 EX 阶段。

在每条指令的 ID 阶段，我们均会检测该指令是否和前一条指令形成“读内存-使用”型数据冒险，如果存在这样的冒险，则发出 STALL 信号，使得 IF、ID 阶段停顿一个周期。

2.3.4 分支预测原理分析

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，即对于所有的指令均预测不会跳转，当预测错误时，则生成 NOP 信号请求清空 IF、ID 阶段。这种策略的正确性需要分情况说明：

- **无条件跳转指令**：这类指令会在 ID 阶段完成跳转，产生的 NOP 信号会清空 ID-EX 段寄存器，即跳转指令本身会在 ID 阶段结束后被清除，但由于跳转指令在 ID 阶段就已经完成了所有工作，因此并不会导致执行错误。
- **条件跳转指令**：这类指令会在 EX 阶段完成跳转，当跳转发生时，IF、ID 阶段的指令由于预测错误，理应被清除，因此在此情况下，我们的清除策略也是正确的。

在此，有以下两点需要特别说明：

- 当 IF-ID 段寄存器接收到 NOP 信号时，并不一定会清空段寄存器，此处我们额外加入了当前指令地址和跳转目标 PC 的比较，当两者相同时，指令会正常执行。这样的设计可以为短跳转（例如仅跳过一行指令的条件跳转）减少一个停顿周期。
- 在跳转目标 PC 的选择电路中，我们将条件跳转（beq、bne）的选择器安排在无条件跳转之后。这样的设计，当条件跳转语句后紧跟非条件跳转时，保证了跳转目标 PC 的正确性。

2.3.5 jal 指令实现原理分析

如先前所说，无条件跳转语句会在 ID 阶段完成所有工作，但是 jal 指令由于需要进行寄存器写入，可能会与处于 WB 阶段的指令产生结构冒险。为解决这个问题，处理 jal 指令时，将当前处于 EX、MA、WB 阶段的指令全部停顿一个周期，为 jal 指令让出寄存器写入端口。由于 jal 使用的 31 号寄存器不是通用寄存器，其余指令不会对此寄存器进行写入，改变寄存器写入顺序不会导致数据错误。

3 功能实现

3.1 功能模块的实现

3.1.1 主控制器模块的实现

本实验中，主控制器模块 (Ctr) 的输出结果由指令中 opCode 段决定。通过 case 语句，我们可以将指令与 opCode 对应，并根据每个指令的操作需求输出控制信号。

主控制器模块的完整实现见 Ctr.v，相比实验 5 中实现代码，本实验中增加了部分跳转指令与 lui 指令的控制信号，同时将 jrSign 信号的产生集成到主控制器模块中。部分核心代码如下：

```
always @(opCode or funct)
begin
    case(opCode)
    6'b000000: //R type
    begin
        RegDst = 1;
        ALUSrc = 0;
        MemToReg = 0;
        MemRead = 0;
        MemWrite = 0;
        BeqSign = 0;
        BneSign = 0;
        ExtSign = 0;
        LuiSign = 0;
        JalSign = 0;
        ALUOp = 3'b101;
        JumpSign = 0;
        if (funct == 6'b001000) begin
            RegWrite = 0;
            JrSign = 1;
        end else begin
            RegWrite = 1;
            JrSign = 0;
        end
    end
    6'b100011: //lw
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 1;
        RegWrite = 1;
        MemRead = 1;
```

```
            MemWrite = 0;
            BeqSign = 0;
            BneSign = 0;
            ExtSign = 1;
            LuiSign = 0;
            JalSign = 0;
            ALUOp = 3'b000;
            JumpSign = 0;
            JrSign = 0;
        end
        //后续代码类似，此处省略
        endcase
end
```

受限于篇幅，此处只展示 R 型指令与 lw 指令对应的实现。

代码中 RegDst、ALUSrc、MemToReg 等为控制信号寄存器，与对应输出信号线相连。

3.1.2 ALU 控制器模块的实现

ALU 控制器模块 (ALUCtr) 的输出由 aluOp 与 funct 共同决定，其行为与主控制器模块 (Ctr) 相似。不同的是，aluOp 与 funct 有部分位在部分指令中属于无关的位，因此我们使用 casex 替代 case。casex 中，可以用 x 表示我们不关心的位。

ALU 控制器模块的完整实现见 ALUCtr.v，相比实验 5 中实现，本实验中 ALU 控制器模块不再产生 jrSign 信号。核心代码如下：

```
always @ (aluOp or funct)
begin
    ShamtSign = 0;
    casex({aluOp,funct})
        9'b000xxxxxx:  // lw,sw,add,addiu
            ALUCtrOut = 4'b0010;     //add
        9'b001xxxxxx:  // beq,bne
            ALUCtrOut = 4'b0110;     //sub
        9'b010xxxxxx:   //stli
            ALUCtrOut = 4'b0111;
        9'b110xxxxxx:   //stliu
            ALUCtrOut = 4'b1000;
        9'b011xxxxxx:  // andi
            ALUCtrOut = 4'b0000;
        9'b100xxxxxx:  // ori
            ALUCtrOut = 4'b0001;
        9'b111xxxxxx:  // xori
            ALUCtrOut = 4'b1011;
```

```
19
20          //R type
21          9'b101001000:  // jr
22              ALUCtrOut = 4'b0101;
23          9'b101000000:  // sll
24          begin
25              ALUCtrOut = 4'b0011;
26              ShamtSign = 1;
27          end
28          9'b101000010:  // srl
29          begin
30              ALUCtrOut = 4'b0100;
31              ShamtSign = 1;
32          end
33          9'b101000011:  // sra
34          begin
35              ALUCtrOut = 4'b1110;
36              ShamtSign = 1;
37          end
38          9'b101000100:  // sllv
39              ALUCtrOut = 4'b0011;
40          9'b101000110:  // srlv
41              ALUCtrOut = 4'b0100;
42          9'b101000111:  // srav
43              ALUCtrOut = 4'b1110;
44          9'b101100000:  // add
45              ALUCtrOut = 4'b0010;
46          9'b101100001:  // addu
47              ALUCtrOut = 4'b0010;
48          9'b101100010:  // sub
49              ALUCtrOut = 4'b0110;
50          9'b101100011:  // subu
51              ALUCtrOut = 4'b0110;
52          9'b101100100:  // and
53              ALUCtrOut = 4'b0000;
54          9'b101100101:  // or
55              ALUCtrOut = 4'b0001;
56          9'b101100110:  // xor
57              ALUCtrOut = 4'b1011;
58          9'b101100111:  // nor
59              ALUCtrOut = 4'b1100;
```

```
60          9'b101101010:   // slt
61              ALUCtrOut = 4'b0111;
62          9'b101101011:   // sltu
63              ALUCtrOut = 4'b1000;
64      endcase
65  end
```

3.1.3 ALU 模块的实现

ALU 模块根据 aluCtr 信号完成指定的功能，可以使用 case 语句选择操作，利用 verilog 自带的运算符完成运算。

ALU 模块的完整实现见 ALU.v，本实验 ALU 模块实现与实验 5 完全一致，核心代码如下：

```
1  always @ (input1 or input2 or aluCtr)
2  begin
3      case(aluCtr)
4      4'b0000:      //AND
5          ALURes = input1 & input2;
6      4'b0001:      //OR
7          ALURes = input1 | input2;
8      4'b0010:      //ADD
9          ALURes = input1 + input2;
10     4'b0011:      //Left-shift
11         ALURes = input2 << input1;
12     4'b0100:      //Right-shift
13         ALURes = input2 >> input1;
14     4'b0101:
15         ALURes = input1;
16     4'b0110:      //SUB
17         ALURes = input1 - input2;
18     4'b0111:      //SLT
19         ALURes = ($signed(input1) < $signed(input2));
20     4'b1000:      //SLTU
21         ALURes = (input1 < input2);
22     4'b1011:      //xor
23         ALURes = input1 ^ input2;
24     4'b1100:      //nor
25         ALURes = ~(input1 | input2);
26     4'b1110:      //Right-shift-arithmetic
27         ALURes = ($signed(input2) >> input1);
28     endcase
29     if(ALURes==0)
```

```
30          Zero = 1;
31      else
32          Zero = 0;
33  end
```

SLT 运算功能的实现中，由于 Verilog 会默认以无符号数解释 wire 类型，需要通过 $signed 关键词将输入解释为带符号数后再进行比较。

在 ALU 模块的结尾，我们判断运算结果是否为 0，并以此设置 Zero 寄存器，最终作为 zero 信号输出。

3.1.4 寄存器模块的实现

寄存器会一直进行读操作，而由 regWrite 控制写操作。为实现信号同步，保证信号的完整性，写操作仅在时钟下降沿进行。由于读操作即时进行，寄存器内容被修改后，对应寄存器的读取结果也会同时更新。

相比与实验 4 中的实现，本实验中寄存器模块可以响应 reset 信号，当 reset 为高电平时，所有寄存器清零。

寄存器模块的完整实现见 Registers.v, 核心部分代码如下：

```
1  reg [31:0] RegFile[31:0];
2  integer i;
3
4  initial begin
5      RegFile[0] = 0;
6  end
7
8  assign readData1 = RegFile[readReg1];
9  assign readData2 = RegFile[readReg2];
10
11 always @ (negedge clk or reset)
12 begin
13     if(reset)
14     begin
15         for(i=0;i<32;i=i+1)
16             RegFile[i] = 0;
17     end
18     else begin
19         if(regWrite)
20             RegFile[writeReg] = writeData;
21     end
22 end
```

3.1.5 内存单元模块的实现

内存模块由 memRead 控制是否进行读取操作，当 memRead 或 address 信号发生变化或时钟处于下降沿时，内存模块会根据 address 指定的内存地址内容更新数据读取数据，并将其作为结果输出。本模块中，内存单元模块与高速缓存模块协同工作，内存模块会一次性返回连续 4 个字的数据。

写操作由 memWrite 信号控制。与寄存器模块相似，为实现信号同步，保证信号的完整性，写操作仅在时钟下降沿进行。

内存模块的完整实现见 dataMemory.v, 核心部分代码如下：

```
reg [31:0] memFile [0:1023];
reg [127:0] ReadData;
always @(memRead or address or memWrite)
begin
    if(memRead)
    begin
        if(address<1023)
            ReadData = {memFile[address],memFile[address+1],memFile[
                address+2],memFile[address+3]};
        else
            ReadData = 0;
    end
end

always @(negedge clk)
begin
    if(memWrite)
        if(address<1023)
            memFile[address] = writeData;
end

assign readData = ReadData;
```

3.1.6 高速缓存模块的实现

对于外部系统而言，高速缓存模块的接口与普通内存模块没有差异。进行读操作时，高速缓存模块会首先检查缓存是否有效，若缓存失效则从内存读取数据并保存至缓存。进行写操作时，直接写入内存模块，同时将缓存中对应块标记为无效。

当缓存失效而从内存读取数据时，需要添加延时以等待内存模块完成读取操作，待内存模块读取完成且缓存块更新之后再输出读取结果。

高速缓存模块的完整实现见 Cache.v, 核心部分代码如下：

```
reg[31:0] cacheFile[0:63];
reg validBit[0:15];
```

```
reg[25:0] tag[0:15];

reg[31:0] ReadData;

wire[127:0] dataFromMemFile;

wire[3:0] cacheAddr = address[5:2];
wire[31:0] MemFileAddress = {address[31:2],2'b00};
integer i;
dataMemory mem(
    .clk(clk),
    .address(MemFileAddress),
    .writeData(writeData),
    .memWrite(memWrite),
    .memRead(memRead),
    .readData(dataFromMemFile)
);

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

always @(memRead or address or memWrite)
begin
    if(memRead)
    begin
        if(validBit[cacheAddr] & tag[cacheAddr] == address[31:6])
            ReadData = cacheFile[address[5:0]];
        else begin
            tag[cacheAddr] = address[31:6];
            validBit[cacheAddr] = 1'b1;
            #5
            cacheFile[{cacheAddr,2'b11}] = dataFromMemFile[31:0];
            cacheFile[{cacheAddr,2'b10}] = dataFromMemFile[63:32];
            cacheFile[{cacheAddr,2'b01}] = dataFromMemFile[95:64];
            cacheFile[{cacheAddr,2'b00}] = dataFromMemFile[127:96];
            ReadData = cacheFile[address[5:0]];
        end
```

```
44      end
45  end
46
47  always @(negedge clk)
48  begin
49      if(memWrite)
50          validBit[cacheAddr] = 1'b0;
51  end
52
53  assign readData = ReadData;
```

3.1.7 符号扩展模块的实现

带符号扩展可以通过在高 16 位填入立即数第 15 位实现，无符号扩展可以通过在高 16 位填入 0 实现。为了在两种工作模式下切换，可以通过一个三目运算符根据 signExt 信号在两种扩展结果中进行选择。

符号扩展模块的完整实现见 signext.v, 核心部分代码如下：

```
1  assign data = signExt?{{16{inst[15]}},inst[15:0]}:{{16{0}},inst
       [15:0]};
```

3.1.8 数据选择器模块的实现

使用 Verilog 自带的三目运算符即可实现数据选择器的功能。Mux 与 RegMux 的差异仅在于输入输出数据长度不同。

数据选择器模块的完整实现见 Mux.v 与 RegMux.v，核心代码如下：

```
1  assign out = select?input1:input0;
```

3.1.9 指令内存模块的实现

指令内存模块只需要根据 PC 地址输出对应的指令，实现相对简单。

指令内存模块的完整实现见 InstMem.v，核心代码如下：

```
1  reg [31:0] instFile[0:1023];
2  assign inst = instFile[address/4];
```

3.2 顶层模块的实现

3.2.1 段寄存器实现

```
1  //IF to ID
```

```
reg [31:0] IF2ID_INST;
reg [31:0] IF2ID_PC;
//ID to EX
reg [2:0] ID2EX_ALUOP;
reg [7:0] ID2EX_CTR_SIGNALS;
reg [31:0] ID2EX_EXT_RES;
reg [4:0] ID2EX_INST_RS;
reg [4:0] ID2EX_INST_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;
//EX to MA
reg [3:0] EX2MA_CTR_SIGNALS;
reg [31:0] EX2MA_ALU_RES;
reg [31:0] EX2MA_REG_READ_DATA_2;
reg [4:0] EX2MA_REG_DEST;
//MA to WB
reg MA2WB_CTR_SIGNALS;
reg [31:0] MA2WB_FINAL_DATA;
reg [4:0] MA2WB_REG_DEST;
```

3.2.2 功能模块连接

每个阶段中的功能模块从段寄存器中获得输入信号，产生输出信号，输出信号将在下一个时钟上升沿写入下一阶段的段寄存器中。

以主控制器模块的连接为例，其实现如下：

```
wire [12:0] ID_CTR_SIGNALS;
wire [2:0] ID_CTR_SIGNAL_ALUOP;
wire ID_JUMP_SIG;
wire ID_JR_SIG;
wire ID_EXT_SIG;
wire ID_REG_DST_SIG;
wire ID_JAL_SIG;
wire ID_ALU_SRC_SIG;
wire ID_LUI_SIG;
wire ID_BEQ_SIG;
wire ID_BNE_SIG;
wire ID_MEM_WRITE_SIG;
```

```
13 wire ID_MEM_READ_SIG;
14 wire ID_MEM_TO_REG_SIG;
15 wire ID_REG_WRITE_SIG;
16 wire ID_ALU_OP;
17 Ctr main_ctr(
18     .opCode(IF2ID_INST[31:26]),
19     .funct(IF2ID_INST[5:0]),
20     .jumpSign(ID_JUMP_SIG),
21     .jrSign(ID_JR_SIG),
22     .extSign(ID_EXT_SIG),
23     .regDst(ID_REG_DST_SIG),
24     .jalSign(ID_JAL_SIG),
25     .aluSrc(ID_ALU_SRC_SIG),
26     .luiSign(ID_LUI_SIG),
27     .beqSign(ID_BEQ_SIG),
28     .bneSign(ID_BNE_SIG),
29     .memWrite(ID_MEM_WRITE_SIG),
30     .memRead(ID_MEM_READ_SIG),
31     .memToReg(ID_MEM_TO_REG_SIG),
32     .regWrite(ID_REG_WRITE_SIG),
33     .aluOp(ID_CTR_SIGNAL_ALUOP)
34 );
35 //将信号合并为总线，便于段寄存器读写
36 assign ID_CTR_SIGNALS[12] = ID_JUMP_SIG;
37 assign ID_CTR_SIGNALS[11] = ID_JR_SIG;
38 assign ID_CTR_SIGNALS[10] = ID_EXT_SIG;
39 assign ID_CTR_SIGNALS[9] = ID_REG_DST_SIG;
40 assign ID_CTR_SIGNALS[8] = ID_JAL_SIG;
41 assign ID_CTR_SIGNALS[7] = ID_ALU_SRC_SIG;
42 assign ID_CTR_SIGNALS[6] = ID_LUI_SIG;
43 assign ID_CTR_SIGNALS[5] = ID_BEQ_SIG;
44 assign ID_CTR_SIGNALS[4] = ID_BNE_SIG;
45 assign ID_CTR_SIGNALS[3] = ID_MEM_WRITE_SIG;
46 assign ID_CTR_SIGNALS[2] = ID_MEM_READ_SIG;
47 assign ID_CTR_SIGNALS[1] = ID_MEM_TO_REG_SIG;
48 assign ID_CTR_SIGNALS[0] = ID_REG_WRITE_SIG;
```

3.2.3 前向通路实现

当前指令处于 EX 阶段时，运算输入可能来自先前指令的写回结果，为提高流水线性能，我们引入前向数据通路。进行数据前向传递需要满足以下条件：

1. 先前指令需要进行寄存器写入操作。

2. 先前指令写入的目标寄存器与当前指令的读取寄存器相同。

前向数据通路的实现如下:

```
wire [31:0] EX_FORWARDING_A_TEMP;
wire [31:0] EX_FORWARDING_B_TEMP;
Mux forward_A_mux1(
    .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RS)),
    .input0(ID2EX_REG_READ_DATA1),
    .input1(MA2WB_FINAL_DATA),
    .out(EX_FORWARDING_A_TEMP)
);
Mux forward_A_mux2(
    .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RS)),
    .input0(EX_FORWARDING_A_TEMP),
    .input1(EX2MA_ALU_RES),
    .out(FORWARDING_RES_A)
);
Mux forward_B_mux1(
    .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RT)),
    .input0(ID2EX_REG_READ_DATA2),
    .input1(MA2WB_FINAL_DATA),
    .out(EX_FORWARDING_B_TEMP)
);
Mux forward_B_mux2(
    .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RT)),
    .input0(EX_FORWARDING_B_TEMP),
    .input1(EX2MA_ALU_RES),
    .out(FORWARDING_RES_B)
);
```

代码中 forward_A、forward_B 为两组前项通路，分别用于将数据传输 ALU 的两个输入端口，每组通路包括两个选择器，分别从 EX-MA 段寄存器与 MA-WB 段寄存器获取数据。

3.2.4 跳转目标 PC 选择的实现

我们使用 4 个数据选择器选择下一个 PC 地址，负责条件跳转的选择器在无条件跳转的选择器之后，其原因已在2.3.4节中说明。

PC 地址选择的实现代码如下：

```
// ID stage
wire [31:0] PC_AFTER_JUMP_MUX;
```

```
Mux jump_mux(
    .select(ID_JUMP_SIG),
    .input1(((IF2ID_PC + 4) & 32'hf0000000) + (IF2ID_INST[25 : 0]
        << 2)),
    .input0(IF_PC + 4),
    .out(PC_AFTER_JUMP_MUX)
);

wire[31:0] PC_AFTER_JR_MUX;
Mux jr_mux(
    .select(ID_JR_SIG),
    .input0(PC_AFTER_JUMP_MUX),
    .input1(ID_REG_READ_DATA1),
    .out(PC_AFTER_JR_MUX)
);

// EX stage
wire EX_BEQ_BRANCH = EX_BEQ_SIG & EX_ALU_ZERO;
wire[31:0] PC_AFTER_BEQ_MUX;
Mux beq_mux(
    .select(EX_BEQ_BRANCH),
    .input1(BRANCH_DEST),
    .input0(PC_AFTER_JR_MUX),
    .out(PC_AFTER_BEQ_MUX)
);

wire EX_BNE_BRANCH = EX_BNE_SIG & (~EX_ALU_ZERO);
wire[31:0] PC_AFTER_BNE_MUX;
Mux bne_mux(
    .select(EX_BNE_BRANCH),
    .input1(BRANCH_DEST),
    .input0(PC_AFTER_BEQ_MUX),
    .out(PC_AFTER_BNE_MUX)
);
wire[31:0] NEXT_PC = PC_AFTER_BNE_MUX;
wire BRANCH = EX_BEQ_BRANCH | EX_BNE_BRANCH;
```

3.2.5 流水线时序实现

本部分是流水线处理器实现的核心，段寄存器写入、分支预测、停顿等功能均在本部分实现。各个段寄存器的清空、停顿条件如下：

- **IF-ID 段寄存器：**如果 NOP 信号有效且当前指令地址与目标跳转 PC 不一致，说明分支预测失败，则清空段寄存器。
- **ID-EX 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID-EX 段寄存器维持原样，ID 阶段结果不写入段寄存器。STALL 信号有效时表明需要停顿，清空 ID-EX 段寄存器，避免 EX 阶段执行错误指令。NOP 指令有效时说明分支预测错误，清空 ID-EX 段寄存器。
- **EX-MA 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID 之后的阶段需要停顿一个周期，EX-MA 段寄存器维持原样。
- **MA-WB 段寄存器：**ID_JAL_SIG 信号有效时说明当前指令为 jal，ID 之后的阶段需要停顿一个周期，MA-WB 段寄存器维持原样。

流水线时序实现代码如下：

```
always @(posedge clk)
begin
    NOP = BRANCH | ID_JUMP_SIG | ID_JR_SIG;
    STALL = ID2EX_CTR_SIGNALS[2] & //该信号来自于ID_MEM_READ_SIG
                ((ID2EX_INST_RT == ID_REG_RS) |
                (ID2EX_INST_RT == ID_REG_RT)); //如果读取的两个寄存
                    器与内存读冲突则stall

    if(!STALL)
    begin
        if(NOP)
        begin
            if(IF_PC == NEXT_PC)
            begin
                IF2ID_INST <= IF_INST;
                IF2ID_PC <= IF_PC;
                IF_PC <= IF_PC + 4;
            end
            else begin
                IF2ID_INST <= 0;
                IF2ID_PC <= 0;
                IF_PC <= NEXT_PC;
            end
        end
        else begin
            IF2ID_INST <= IF_INST;
            IF2ID_PC <= IF_PC;
            IF_PC <= NEXT_PC;
        end
```

```
29      end
30
31      // ID - EX
32      if (!ID_JAL_SIG)
33      begin
34          if (STALL|NOP)
35          begin
36              //STALL：下一个周期不进行EX阶段，等待上条指令MA完成
37              //BRANCH：发生跳转，从这里截断命令
38              //JUMP的决定在ID阶段，不影响
39              ID2EX_PC <= IF2ID_PC;
40              ID2EX_ALUOP <= 3'b000;
41              ID2EX_CTR_SIGNALS <= 0;
42              ID2EX_EXT_RES <= 0;
43              ID2EX_INST_RS <= 0;
44              ID2EX_INST_RT <= 0;
45              ID2EX_REG_READ_DATA1 <= 0;
46              ID2EX_REG_READ_DATA2 <= 0;
47              ID2EX_INST_FUNCT <= 0;
48              ID2EX_INST_SHAMT <= 0;
49              ID2EX_REG_DEST <= 0;
50          end else
51          begin
52              ID2EX_PC <= IF2ID_PC;
53              ID2EX_ALUOP <= ID_CTR_SIGNAL_ALUOP;
54              ID2EX_CTR_SIGNALS <= ID_CTR_SIGNALS[7:0];
55              ID2EX_EXT_RES <= ID_EXT_RES;
56              ID2EX_INST_RS <= ID_REG_RS;
57              ID2EX_INST_RT <= ID_REG_RT;
58              ID2EX_REG_DEST <= ID_REG_DEST;
59              ID2EX_REG_READ_DATA1 <= ID_REG_READ_DATA1;
60              ID2EX_REG_READ_DATA2 <= ID_REG_READ_DATA2;
61              ID2EX_INST_FUNCT <= IF2ID_INST[5:0];
62              ID2EX_INST_SHAMT <= IF2ID_INST[10:6];
63          end
64      end
65
66      // EX - MA
67      if (!ID_JAL_SIG)
68      begin
69          EX2MA_CTR_SIGNALS <= ID2EX_CTR_SIGNALS[3:0];
```

```
70              EX2MA_ALU_RES <= EX_FINAL_DATA;
71              EX2MA_REG_READ_DATA_2 <= FORWARDING_RES_B;
72              EX2MA_REG_DEST <= ID2EX_REG_DEST;
73          end
74
75          // MA - WB
76          if (!ID_JAL_SIG)
77          begin
78              MA2WB_CTR_SIGNALS <= EX2MA_CTR_SIGNALS[0];
79              MA2WB_FINAL_DATA <= MA_FINAL_DATA;
80              MA2WB_REG_DEST <= EX2MA_REG_DEST;
81          end
82  end
```

4 结果验证

编写如表10所示的汇编代码进行测试。

地址	指令	汇编指令	预期执行结果		
			$0	$1	$2
0x00	00001000000000000000000000000100	j	跳转至 0x10		
0x04	00000000000000000000000000000000	nop	不执行		
0x08	00000000000000000000000000000000	nop	不执行		
0x0c	00000000000000000000000000000000	nop	不执行		
0x10	00001100000000000000000000000110	jal	跳转至 0x18, $31 = 0x14		
0x14	00000000000000000000000000000000	nop	空指令		
0x18	00111100000000101111111111111111	lui $2,65535	x	-65536	x
0x1c	10001100000000010000000000000010	lw $1,2($0)	2	-65536	x
0x20	10001100000000010000000000000011	lw $1,3($0)	3	-65536	x
0x24	10001100000000010000000000000100	lw $2,4($0)	3	4	x
0x28	00000000001000100001100000100000	add $3,$1,$2	3	4	7
0x2c	00000000001000100001100000100100	and $3,$1,$2	3	4	0
0x30	00100000000000110000000000000110	addi $3,$0,6	3	4	6
0x34	00000000001000100010000000100010	sub $4,$1,$2	$4 = -1		
0x38	00000000000000010000100001000000	sll $1,$1,1	6	4	6
0x3c	00000000000000011000110001100000	sll $3,$3,3	6	4	48
0x40	00000000001000110000100000100101	or $1,$3,$1	54	4	48
0x44	00100100000000010000000000000110	addiu $2,$0,6	54	6	48
0x48	00101000010000010000000000000001	slti $1,$1,1	0	6	48
0x4c	00110100010000010000000000000001	ori $1,$2,1	7	6	48

0x50	00000000010000110000100000000100	sllv $1,$3,$2	3072	6	48
0x54	00000000001000100001100000100001	addu $3,$1,$2	3072	6	3078
0x58	00000000001000100001100000100010	sub $3,$1,$2	3072	6	3066
0x5c	00000000001000100001100000100011	subu $3,$1,$2	3072	6	3066
0x60	00000000001000100001100000100101	or $3,$1,$2	3072	6	3078
0x64	00000000001000100001100000100110	xor $3,$1,$2	3072	6	3078
0x68	00000000001000100001100000100111	nor $3,$1,$2	3072	6	-3079
0x6c	00000000001000100001100000101010	slt $3,$1,$2	3072	6	0
0x70	00000000000000010000100001000011	sra $1,$1,1	1536	6	0
0x74	10101100000000100000000000000001	sw $2,1($0)	内存地址 0x01 写入数值 6		
0x78	00000000010000000000100000101011	sltu $1,$0,$2	0	6	0
0x7c	00000000000000010000100000100010	srl $2,$2,2	0	3	0
0x80	00000000010000100001000000000111	srav $2,$2,$2	0	0	0
0x84	00111000010000100000000000000010	xori $2,$2,2	0	2	0
0x88	00101100010000010000000000000100	sltiu $1,$2,4	1	2	0
0x8c	00010000101000000000000000000001	beq $5,$0,1	跳转至 0x94		
0x90	00010000101000000000000000000001	beq $5,$0,1	不执行		
0x94	00010100101000000000000000000001	bne $5,$0,1	不跳转		
0x98	00000011111000000000000000001000	jr $31	跳转至 0x14		

表 10: 仿真使用的指令

数据内存的初始值如表11所示。

地址	数据
0x00	0x00000000
0x01	0x00000001
0x02	0x00000002
0x03	0x00000003
0x04	0x00000004
0x05	0x00000005

表 11: 数据内存初始值

在激励文件中载入指令内存与数据内存数据，此处需要根据数据文件具体目录修改路径。

```
1 $readmemb("E:/course/arclab/lab06/mem_inst.dat",top.inst_mem.
      instFile);
2 $readmemh("E:/course/arclab/lab06/mem_data.dat",top.memory.mem.
      memFile);
```

仿真结果如图4所示。需要特别说明的是，由于显示缩放问题，部分数字可能没有显示完整。

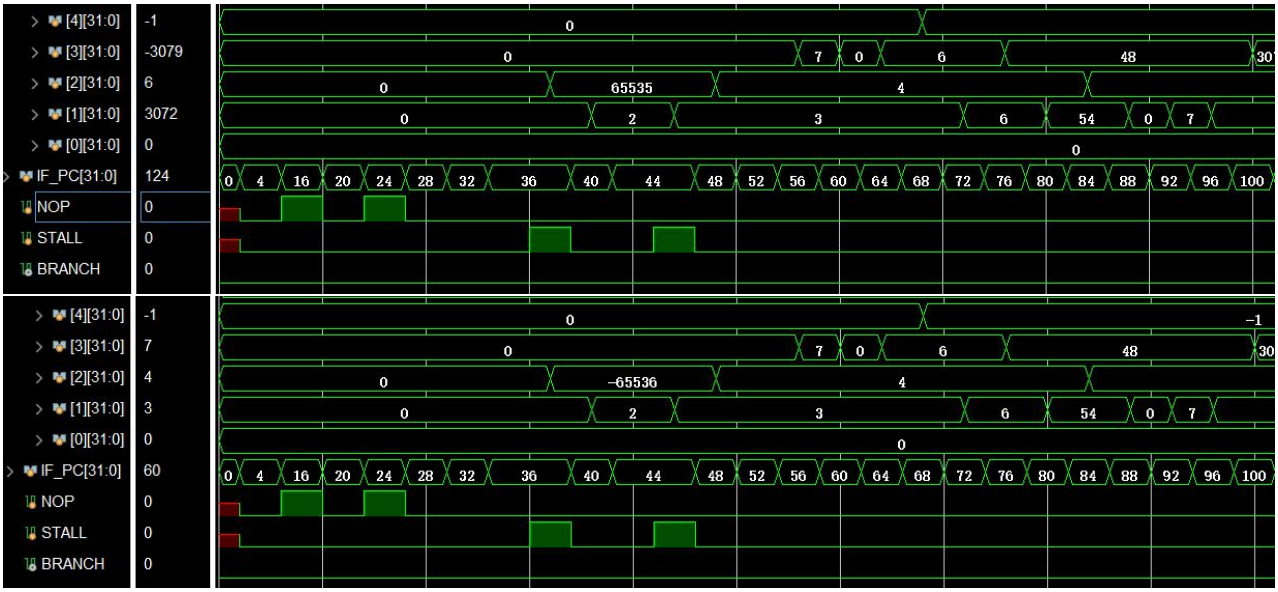

图 4: 仿真截图

可以看出，我们实现的类 MIPS 流水线处理器完成了设计的所有功能，执行结果符合预期。

5 总结与反思

本实验综合先前 5 个实验，完成了类 MIPS 流水线处理器的实现。在实验开始之前，我对流水线的实现毫无头绪，当时的我认为实现流水线处理器是一个毫无可能的任务；实验开始后，在实验指导书的提示下，我将复杂的流水线结构分解，逐一实现每个阶段的线路连接，我发现每个阶段的实现与单周期处理器非常接近，在此之后，只需要将线路整体连接到段寄存器上即可以实现基础的流水线。在完成基础的流水线之后，逐步添加前向通路、停顿、分支预测就显得相对简单。最终，我完成了整个流水线。

通过本实验，我加深了对流水器处理器结构的记忆与理解。调试段寄存器时序的过程，让我更加熟悉 Verilog 的语法以及 Vivado 软件的仿真调试功能。编写仿真指令的过程，我熟悉了各类 MIPS 指令的结构、功能与实现原理。完成实验报告的过程，让我对 latex 的使用更加熟练，也积累了使用 Latex 编写中文文章的经验。

最后，我想向计算机系统结构实验 (CS145) 的指导老师与助教、计算机系统结构课程 (CS359) 的授课老师以及实验过程中帮助过我的同学表示由衷的感谢，感谢你们的帮助与陪伴。