计算机系统结构实验报告　实验 6

简单的类 MIPS 多周期流水线处理器实现

黄骁 520021910388

2022 年 4 月 19 日

摘要

本实验实现了类 MIPS 多周期流水线处理器的几个重要部件，该类 MIPS 多周期流水线处理器的 CPU 支持全部 31 条 MIPS 指令。本实验以实验三、四实现的模块以及实验五实现的类 MIPS 单周期处理器为基础，对部分功能进行了修改，添加了部分控制信号；同时，为适应流水线结构，增加了段寄存器，支持通过前向通路 (forwarding) 与流水线停顿 (stall) 来解决流水线冒险；同时通过预测不转移 (predict-not-taken) 策略解决控制竞争问题所带来的高延迟，提高流水线性能。本实验通过软硬法仿真的形式进行实验结果的验证。

目录

1 实验目的

本次实验有如下七个实验目的：

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 支持的指令数量从 9 条或 16 条扩充为 31 条，使处理器功能更加丰富（选做）；
6. 在 5. 的基础上，增加协处理器 CP0，支持中断相关机制及中断指令（选做）；
7. 应用 Cache 原理，设计 Cache Line 并进行仿真验证（选做）；

2 原理分析

2.1 功能模块原理分析

2.1.1 主控制单元 (Ctr)、运算单元控制器 (ALUCtr) 原理分析

主控制单元 (Ctr) 的输入为指令的操作码 (opCode) 字段，操作码经过 Ctr 的译码，给 ALUCtr，DataMemory，Registers，Muxs 等功能单元输出正确的控制信号。

本实验中，主控制单元 (Ctr) 可以识别 R 型指令 (add, addu, sub, ,subu, and, or, xor, nor, slt, sltu, sll, srl, sra, sllv, srlv, srav, jr)、I 型指令 (addi, addiu, andi, ori, xori, lui, lw, sw, beq, bne, slti, sltiu)、J 型指令 (j, jar) 并输出对应的控制信号。

本次实验用到的控制信号与实验五类似，其中标注 (*) 为与实验五不同之处比如将 Branch 信号扩展为 BeqSignal 和 BneSignal，增添了 JrSign 和 LuiSign 信号，其中为了提高流水线执行 Jr 指令性能, Jr 指令需要在指令译码 (ID) 阶段完成，而实验五中 jrSign 信号由 ALUCtr 产生, 无法在 ID 阶段完成 Jr 指令的识别, 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。具体说明如表1所示。

信号	具体说明
*ALUOp	4 位信号，发送给 ALU 控制单元 (ALUCtr) 用来进一步解析运算类型的控制信号
ALUSrc	ALU 第二个操作数来源选择信号；低电平：选择 rt 寄存器值，高电平：选择立即数
Jump	无条件跳转信号，高电平说明当前指令是无条件跳转指令
MemToReg	写寄存器的数据来源选择信号；低电平：选择 ALU 运算结果，高电平：选择内存读取数据
MemRead	内存读使能信号，高电平有效
MemWrite	内存写使能信号，高电平有效
RegDst	目标寄存器的选择信号；低电平：写入 rt 寄存器；高电平：写入 rd 寄存器
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
ExtSign	符号扩展信号，高电平说明当前指令需要进行符号拓展
JalSign	跳转并链接指令信号，高电平说明当前指令是 jal 指令
*BeqSign	条件跳转 beq 指令信号，高电平说明当前指令是 beq 指令
*BneSign	条件跳转 bne 指令信号，高电平说明当前指令是 bne 指令
*JrSign	寄存器无条件跳转指令信号，高电平说明当前指令是 jr 指令
*LuiSign	载入立即数指令信号，高电平说明当前指令是 lui 指令

表 1: 主控制单元 (Ctr) 产生的控制信号

由于 ALUOp 信号包含四个二进制位以支持 31 条指令，而 ALUCtrOut 也是四个二进制位，我们可以直接将 ALUOp 和 ALUCtrOut 进行一一匹配，这样设计更加方便，调整后如表 2 所示。

ALUOp 的信号内容	指令	具体说明
0000	addi	ALU 执行带溢出检查的加法
0001	addiu	ALU 执行不带溢出检查的加法
0010	sub	ALU 执行带溢出检查的减法
0011	subu, beq, bne	ALU 执行不带溢出检查的减法
0100	and, andi	ALU 执行逻辑与运算
0101	or, ori	ALU 执行逻辑或运算
0110	xor, xori	ALU 执行逻辑异或运算
0111	nor	ALU 执行逻辑或非运算
1000	slt, slti	ALU 执行带符号小于时置位运算
1001	sltu, sltiu	ALU 执行无符号小于时置位运算
1010	sll, sllv, lui	ALU 执行逻辑左移运算
1011	srl, srlv	ALU 执行逻辑右移运算
1100	sra, srav	ALU 执行算术右移运算
1101	R-format	ALUCtr 结合指令 Funct 段决定最终操作
1110	j, jal, jr	ALU 不进行任何操作

表 2: ALUOp 信号的具体含义以及解析方式

与实验五类似，为了支持 jr 指令以及带有 shamt 操作数的 sll 与 srl 指令，我们新增了两个信号 jrSign 与 shamtSign，分别用来表示该指令是否为 jr、该指令操作数是否在 shamt 中。

2.1.2 ALU 控制单元 (ALUCtr) 原理分析

算数逻辑单元 ALU 的控制单元（ALUCtr）是根据主控制器的 ALUOp 控制信号来判断指令类型，并依据指令的后 6 位区分 R 型指令。综合这两种输入，以控制 ALU 做正确操作。

由于 ALUOp 信号的改动，ALU 控制单元 (ALUCtr) 也需要做出相应的更改，在本实验中，我们让 ALUOp 和 ALUCtrOut 进行一一匹配，其解析方式如表 3 所示。

指令	ALUOp	Funct	具体说明
addi	0000	xxxxxx	带溢出检查的加法运算
addiu, lw, sw	0001	xxxxxx	不带溢出检查的加法运算
beq, bne	0011	xxxxxx	不带溢出检查的减法运算
andi	0100	xxxxxx	逻辑与运算
ori	0101	xxxxxx	逻辑或运算
xori	0110	xxxxxx	逻辑异或运算
slti	1000	xxxxxx	带符号小于时置位运算
sltiu	1001	xxxxxx	无符号小于时置位运算
lui	1010	xxxxxx	逻辑左移运算
add	1101	100000	带溢出检查的加法运算
addu	1101	100001	不带溢出检查的加法运算
sub	1101	100010	带溢出检查的减法运算
subu	1101	100011	不带溢出检查的减法运算
and	1101	100100	逻辑与运算
or	1101	100101	逻辑或运算
xor	1101	100110	逻辑或非运算
nor	1101	100111	逻辑或非运算
slt	1101	101010	带符号小于时置位运算
sltu	1101	101011	无符号小于时置位运算
sll	1101	000000	逻辑左移运算
sllv	1101	000100	逻辑左移运算
srl	1101	000010	逻辑右移运算
srlv	1101	000110	逻辑右移运算
sra	1101	000011	算术右移运算
srav	1101	000111	算术右移运算
jr	1110	001000	不用进行运算
j, jar	1110	xxxxxx	不用进行运算

表 3: ALU 控制单元 (ALUCtr) 的解析方式

2.1.3 算术逻辑运算单元 (ALU) 原理分析

算术逻辑单元 ALU 根据 ALUCtr 的控制信号将两个输入执行与之对应的操作。ALURes 为输出结果。若减法操作 ALURes 的结果为 0 时，则 Zero 输出置为 1。该信号用于与 branch 指令结合判断是否满足转移条件。

与实验五有所扩充，ALU 执行的算术逻辑运算类型与运算单元控制信号 ALUCtrOut 的对应方式如表 4 所示。

ALUCtrOut	ALU 执行算术逻辑运算类型
0000	带溢出检查的加法 add*
0001	不带溢出检查的加法 add
0010	带溢出检查的减法 sub*
0011	不带溢出检查的减法 sub
0100	逻辑与 and
0101	逻辑或 or
0110	逻辑异或 xor
0111	逻辑或非 nor
1000	带符号小于时置位
1001	无符号小于时置位
1010	逻辑左移 shift logical left
1011	逻辑右移 shift logical right
1100	算术右移 shift arithmetic right

表 4: ALUCtrOut 与 ALU 操作的对应关系

2.1.4 寄存器 (Register)、存储器 (Data Memory) 及有符号扩展单元 (Sign Extension) 原理分析

本部分和实验四的原理类似，但需要注意的是模块一些部分的修改。

在寄存器 (Register) 中可以响应 reset 信号，当 reset 为高电平时，所有寄存器清零。增加了对于 jal 指令的支持，由于 jal 默认将下一条指令的地址保存至寄存器 31，于是我们在寄存器文件上增加了一个新的接口用于处理 jal 指令带来的特殊情况。

在有符号扩展单元 (Sign Extension) 中增添了带符号扩展信号 signExt，高电平代表进行带符号扩展，低电平代表无符号扩展。

2.1.5 指令存储器 (Instruction Memory) 原理分析

本部分和存储器 (Data Memory) 几乎相同，而且不需要支持修改操作。指令存储器接受一个 32 位地址输入, 输出一条 32 位指令。

2.1.6 程序计数器模块 (PC) 原理分析

在本次实验中，我们不再设置单独的 PC 模块，而是利用阻塞赋值完成延迟到时钟上升沿赋值的操作；所以在本次实验中，我们将这个模块简化为顶层模块中的一个全局寄存器 PC。

2.1.7 数据选择器 (Mux) 原理分析

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 Mux5 和 Mux32。Mux32 的输入与输出信号均为 32 位, 用于对进行数据进行选择。Mux5 的输出和输出信号为 5 位, 用于寄存器选取信号的选择。

2.2 流水线各阶段原理分析

2.2.1 取指阶段 (IF)

取指阶段主要包含程序计数器模块 (PC) 以及指令存储器模块 (Instruction Memory)，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单元模块 (Ctr)、寄存器模块 (Register)、符号扩展模块 (Sign Extension)。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)

访存阶段主要包括存储器 (Data Memory)，目的是进行内存的访问（读/写）。另外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 (Register), 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。我们将写回安排在时钟下降沿进行操作，这样可以有效解决一部分的数据冒险。

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，我们需要设置段寄存器来保存指令在上一阶段执行的结果下：

- **IF → ID 段寄存器**：主要包含当前指令及其 PC；
- **ID → EX 段寄存器**：主要包含控制信号（包括 ALUOp、ALUSrc、luiSign、beqSign、bneSign、memWrite、memRead、memToReg、regWrite 信号）、有符号扩展单元的结果、解析出的指令 rs、rt、funct、shamt 的值、读 rs 与 rt 寄存器得到的结果、当前指令 PC 与目标寄存器的地址；

- **EX → MA 段寄存器**：主要包含控制信号（包括 memWrite、memRead、memToReg、regWrite 信号）、ALU 的运算结果、rt 寄存器的值以及目标寄存器的地址。
- **MA → WB 段寄存器**：主要包含 regWrite 信号、最终写寄存器的数据以及目标寄存器的地址。

。

2.3.2 前向通路的设计

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。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 流水线停顿的设计

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“(load-use)”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 控制竞争的解决

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。对于非条件转移，我们将其前提至 ID 阶段进行处理，即可将非条件转移指令造成的流水线停顿降为零。

3 功能实现

3.1 各功能模块实现

本实验中用到的各个模块的功能实现与实验五中各个模块的功能实现基本相同，不同之处是增加了一些控制信号，同时重新设计了 ALUOp 与 ALUCtrOut 的编码方式；其余部分基本相同，具体参见附录 A.1。

3.2 顶层模块的功能实现

3.2.1 段寄存器的功能实现

```
1   //IF to ID
2   reg [31:0] IF2ID_INST;
3   reg [31:0] IF2ID_PC;
4   //ID to EX
5   reg [3:0] ID2EX_ALUOP;
6   reg [7:0] ID2EX_CTR_SIGNALS;
7   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;
```

其具体含义我们在第 2.3.1 节进行了具体阐述，这些段寄存器暂时存储前一个阶段的执行结果，以供后一个阶段的执行使用。

3.2.2 前向通路的功能实现

```
    wire[31:0] EX_FORWARDING_A_TEMP;
    wire[31:0] EX_FORWARDING_B_TEMP;
    Mux32 forward_A_mux1(
        .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RS)),
        .input2(ID2EX_REG_READ_DATA1),
        .input1(MA2WB_FINAL_DATA),
        .out(EX_FORWARDING_A_TEMP)
    );
    Mux32 forward_A_mux2(
        .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RS)),
        .input2(EX_FORWARDING_A_TEMP),
        .input1(EX2MA_ALU_RES),
        .out(FORWARDING_RES_A)
    );
    Mux32 forward_B_mux1(
        .select(WB_REG_WRITE & (MA2WB_REG_DEST == ID2EX_INST_RT)),
        .input2(ID2EX_REG_READ_DATA2),
        .input1(MA2WB_FINAL_DATA),
        .out(EX_FORWARDING_B_TEMP)
```

```
    );
    Mux32 forward_B_mux2(
        .select(MA_REG_WRITE & (EX2MA_REG_DEST == ID2EX_INST_RT)),
        .input2(EX_FORWARDING_B_TEMP),
        .input1(EX2MA_ALU_RES),
        .out(FORWARDING_RES_B)
    );
```

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。

3.2.3 PC 更新模块的实现

我们将 PC 模块嵌入了顶层模块后需要解决 PC 更新问题。由于不止条件转移指令会更新 PC, 因此 PC 的更新除判断条件转移指令（beq 与 bne）外，还需要判断 jr、jal、jr 指令。如下所示:

```
    wire[31:0] PC_AFTER_JUMP_MUX;
    Mux32 jump_mux(
        .select(ID_CTR_SIGNALS[12]),
        .input1(((IF2ID_PC + 4) & 32'hf0000000) + (IF2ID_INST [25 : 0] << 2)),
        .input2(IF_PC + 4),
        .out(PC_AFTER_JUMP_MUX)
    );
    wire[31:0] PC_AFTER_JR_MUX;
    Mux32 jr_mux(
        .select(ID_CTR_SIGNALS[11]),
        .input2(PC_AFTER_JUMP_MUX),
        .input1(ID_REG_READ_DATA1),
        .out(PC_AFTER_JR_MUX)
    );
    wire EX_BEQ_BRANCH = EX_BEQ_SIG & EX_ALU_ZERO;
    wire[31:0] PC_AFTER_BEQ_MUX;
    Mux32 beq_mux(
        .select(EX_BEQ_BRANCH),
        .input1(BRANCH_DEST),
        .input2(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;
    Mux32 bne_mux(
        .select(EX_BNE_BRANCH),
        .input1(BRANCH_DEST),
```

```
        .input2(PC_AFTER_BEQ_MUX),
        .out(PC_AFTER_BNE_MUX)
    );
```

3.2.4 段寄存器写入、分支预测、停顿功能实现

```
NOP = BRANCH | ID_CTR_SIGNALS[12] | ID_CTR_SIGNALS[11];
STALL = ID2EX_CTR_SIGNALS[2] & ((ID2EX_INST_RT == IF2ID_INST[25:21]) | (
    ID2EX_INST_RT == IF2ID_INST[20:16]));
        // IF - ID
        if(!STALL)
        begin
            if(!NOP)
            begin
                IF2ID_INST <= IF_INST;
                IF2ID_PC <= IF_PC;
                IF_PC <= NEXT_PC;
            end
            else
            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
        end
        // ID - EX
        if (!ID_CTR_SIGNALS[8])
        begin
            if (STALL | NOP)
            begin
                ID2EX_PC <= IF2ID_PC;
                ID2EX_ALUOP <= 4'b1110;
                ID2EX_CTR_SIGNALS <= 0;
```

```
35              ID2EX_EXT_RES <= 0;
36              ID2EX_INST_RS <= 0;
37              ID2EX_INST_RT <= 0;
38              ID2EX_REG_READ_DATA1 <= 0;
39              ID2EX_REG_READ_DATA2 <= 0;
40              ID2EX_INST_FUNCT <= 0;
41              ID2EX_INST_SHAMT <= 0;
42              ID2EX_REG_DEST <= 0;
43          end else
44          begin
45              ID2EX_PC <= IF2ID_PC;
46              ID2EX_ALUOP <= ID_CTR_SIGNAL_ALUOP;
47              ID2EX_CTR_SIGNALS <= ID_CTR_SIGNALS[7:0];
48              ID2EX_EXT_RES <= ID_EXT_RES;
49              ID2EX_INST_RS <= IF2ID_INST[25:21];
50              ID2EX_INST_RT <= IF2ID_INST[20:16];
51              ID2EX_REG_DEST <= ID_REG_DEST;
52              ID2EX_REG_READ_DATA1 <= ID_REG_READ_DATA1;
53              ID2EX_REG_READ_DATA2 <= ID_REG_READ_DATA2;
54              ID2EX_INST_FUNCT <= IF2ID_INST[5:0];
55              ID2EX_INST_SHAMT <= IF2ID_INST[10:6];
56          end
57      end
58      // EX - MA
59      if (!ID_CTR_SIGNALS[8])
60      begin
61          EX2MA_CTR_SIGNALS <= ID2EX_CTR_SIGNALS[3:0];
62          EX2MA_ALU_RES <= EX_FINAL_DATA;
63          EX2MA_REG_READ_DATA_2 <= FORWARDING_RES_B;
64          EX2MA_REG_DEST <= ID2EX_REG_DEST;
65      end
66      // MA - WB
67      if (!ID_CTR_SIGNALS[8])
68      begin
69          MA2WB_CTR_SIGNALS <= EX2MA_CTR_SIGNALS[0];
70          MA2WB_FINAL_DATA <= MA_FINAL_DATA;
71          MA2WB_REG_DEST <= EX2MA_REG_DEST;
72      end
```

4 结果验证

与实验五相同，我们编写如下汇编代码进行测试。

地址	数据	地址	数据	地址	数据	地址	数据
0x00	0x00000000	0x01	0x00000001	0x02	0x00000002	0x03	0x00000003
0x04	0x00000004	0x05	0x00000005	0x06	0x00000006	0x07	0x00000007

表 5: 内存中的初始值

指令地址	指令	指令解释	执行结果
0x00	100011 00000 00001 0000000000000000	lw $1, 0($0)	$1 = Mem[0] = 0
0x04	000010 00000000000000000000000010	j 2	go to 0x0c
0x08	000000 00000 00000 0000000000000000	nop	(not executed)
0x0c	100011 00000 00010 0000000000000001	lw $2, 2($0)	$2 = Mem[2] = 2
0x10	100011 00000 00010 0000000000000001	lw $2, 1($0)	$2 = Mem[1] = 1
0x14	100011 00000 00010 0000000000000001	lw $3, 3($0)	$3 = Mem[3] = 3
0x18	100011 00000 00011 0000000000000100	lw $3, 4($0)	$3 = Mem[4] = 4
0x1c	100011 00000 00100 0000000000000111	lw $4, 7($0)	$4 = Mem[7] = 7
0x20	000000 00001 00010 0001100000100000	add $4, $1, $2	$4 = 0 + 1 = 1
0x24	001000 00000 00100 0000000000000110	addi $4, $0, 6	$4 = 0 + 6 = 6
0x28	000000 00010 00011 00011 00000 100001	addu $3, $2, $3	$3 = 1 + 4 = 5
0x2c	000000 00011 00100 00100 00000 100010	sub $4, $3, $4	$4 = 5 - 6 = -1
0x30	000000 00001 00010 00011 00000 100011	subu $3, $1, $2	$3 = 1 - 2 = -1
0x34	000000 00001 00010 00100 00000 100101	or $4, $1, $2	$4 = 0 | 1 = 1
0x38	000000 00001 00010 00011 00000 100110	xor $3, $1, $2	$3 = 0 xor 1 = 1
0x3c	000000 00000 00010 00010 00001 000011	sra $2, $2, 1	$2 = 1 » 1 = 0
0x40	101011 00000 00010 0000000000000001	sw $2, 1($0)	Mem[1] = 0
0x44	000000 00000 00001 00010 00000 101011	sltu $1, $0, $2	$1 = 0 < 0 = 0
0x48	000000 00000 00011 00011 00010 000000	sll $3, $3, 2	$3 = 1 « 2 = 4
0x4c	000000 00011 00100 00010 00000000111	srav $2, $4, $3	$2 = 4 » 1 = 0
0x50	000100 00110 0000000000000000000001	beq $6, $0, 1	go to 0x58
0x54	000101 00110 0000000000000000000001	bne $6, $0, 1	(not executed)
0x58	000000 11111 00000000000000001000	jr $31	(not jump)

表 6: 汇编代码及其解释、执行结果

其中，执行结果中表明 “(not executed)” 表示该指令由于跳转等原因没有被执行。我们的测试汇编代码挑选测试了本实验要求的 31 条汇编指令.

我们在激励文件中使用 readmemh 命令将其读入存储器模块的对应位置进行测试。

```
1 $readmemh("D:/Archlabs/lab06/mem_data.txt", cpu.memory.memFile);
2 $readmemb("D:/Archlabs/lab06/mem_inst.txt", cpu.instMemory.instFile);
```

我们用上述汇编命令以及内存的初始情况进行测试，测试结果如图 1 所示。

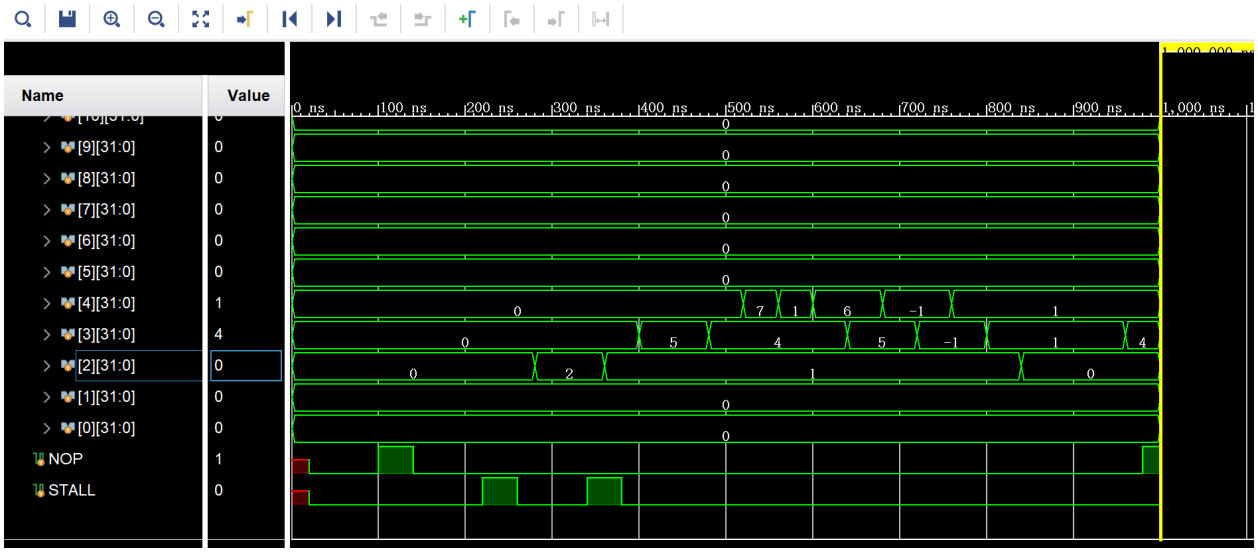

图 1: MIPS 多周期流水线处理器测试结果

对照图 1 与表 3 中的执行结果可知，运行结果完全正确，仿真成功，说明整个 MIPS 多周期流水线处理器实现正确。

5 总结与反思

本实验实现了一个完整的支持 31 指令的 MIPS 多周期流水线处理器，以实验三、四的模块以及实验五的单周期 MIPS 处理器为基础，对部分模块进行重新设计，添加了部分新的控制信号，并且较为完整地实现了流水线技术。

通过这次实验，我加深了对于 MIPS 处理器的数据通路、信号通路等的了解，同时对于流水线有了更加深入的认识，充分理解了流水线中的段寄存器 (segment registers)、前向通路 (forwarding)、流水线停顿 (stall) 以及预测不转移 (predict-not-taken) 等技术的原理以及作用，令我收获颇多。

在理论课中，一些知识学习得比较粗浅，缺乏深入的思考。在实验中，为了把功能正确地实现出来，我需要对课上所学的内容进行巩固和加深。例如处理器的流水化执行这一部分，由于线路复杂，指令之间相关的情况多，学理论课时总有一种没有吃透的感觉。在实现流水线处理器的过程中，我必须将所有这些的连线的条理全部弄清楚。在排错的过程中，更是要一遍遍地理清各阶段的关系，找到错误的源头。这对全面而细致地掌握体系结构时是十分有益处的。同时，在调试的过程中，我对 MIPS 处理器的完整 31 指令也有了更加深刻的理解；通过自行编写对应的汇编代码、手动模拟汇编代码的运行结果、与自己写的处理器的运行结果进行对照等过程，对汇编代码也有了更加深刻的理解

附录 A　设计文件代码实现

A.1　各模块的代码实现

- 主控制器 (Ctr) 参见代码文件 `Ctr.v`。
- 算术逻辑运算单元 (ALU) 参见代码文件 `ALUCtr.v`。
- 算术逻辑运算单元 (ALU) 参见代码文件 `ALU.v`。
- 寄存器 (Register) 参见代码文件 `Registers.v`。
- 存储器 (Data Memory) 参见代码文件 `DataMemory.v`。
- 指令存储器 (Instruction Memory) 参见代码文件 `instMemory.v`。
- 有符号扩展单元 (Sign Extension) 参见代码文件 `SignExt.v`。
- 数据选择器 (Mux) 参见代码文件 `Mux5.v` 与 `Mux32.v`，分别表示 5 位与 32 位数据选择器。
- 我们将程序计数器模块 (PC) 嵌入顶层模块 (Top) 中进行实现，因此没有单独的设计文件。

A.2　顶层模块 (Top) 的代码实现

参见代码文件 `Top.v`。

附录 B　激励文件代码实现

参见代码文件 `mips_tb.v`。