计算机系统结构实验

lab6: 类 MIPS 多周期流水化处理器实现

周旭东 521021910829

2023 April

摘要

在 lab6：类 MIPS 多周期流水化处理器实现中，我学习了 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；在此基础上，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并增加 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；此外，还通过 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。最后通过仿真测试证明了流水线处理器的正确性。

目录

1 实验目的

1. 理解 CPU Pipeline、流水线冒险 (hazard) 及相关性，在 lab5 基础上设计简单流水线 CPU

2. 在 1. 的基础上设计支持 Stall 的流水线 CPU。通过检测竞争并插入停顿 (Stall) 机制解决数据冒险/竞争、控制冒险和结构冒险

3. 在 2. 的基础上，增加 Forwarding 机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能

4. 在 3. 的基础上，通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能

5. 允许考虑将 Stall 与 Forwarding 结合起来实现或将 2.、3. 和 4. 合并起来设计

2 实验原理

MIPS 多周期流水线处理器是一种基于 MIPS 指令集架构的处理器设计，其设计目的是提高处理器的执行效率。该处理器采用了流水线的设计思想，将指令的执行分为若干个步骤，并通过不同的模块实现不同的处理逻辑，从而实现指令的并行执行。

总体设计思路分为一下部分：

- 模块划分：将整个处理器划分为若干个模块，每个模块负责处理指令的不同阶段。
- 流水线划分：将整个处理器的指令执行过程划分为五个阶段：取指令（IF）、指令译码（ID）、执行（EX）、访存（MEM）、写回（WB）。每个阶段对应上面所述的模块。流水线的五个阶段在时钟的不同沿上分别进行。
- 时序设计：每个模块的时序都需要进行设计。例如，PC 计数器模块需要在时钟上升沿时计算下一条指令的地址；指令译码器模块需要在时钟上升沿时进行译码等等。
- 控制信号的生成：每个阶段的控制信号需要在不同的阶段生成。例如，指令译码阶段需要根据指令的类型生成不同的控制信号，而执行阶段需要根据不同的操作码执行不同的操作。
- 冒险与前推：在流水线中，可能会出现冒险问题，例如数据冒险和控制冒险。为了解决这些问题，可以采用前推的技术。前推可以将结果直接传递给需要使用的模块，从而避免数据冒险，提高处理器的执行效率。

2.1 处理器设计原理

进行处理器设计有五个关键步骤

1. 分析指令，得出对数据通路的需求
2. 选择数据通路上合适的组件
3. 连接组件构成数据通路
4. 分析每一条指令的实现，以确定控制信号
5. 集成控制信号，完成控制逻辑

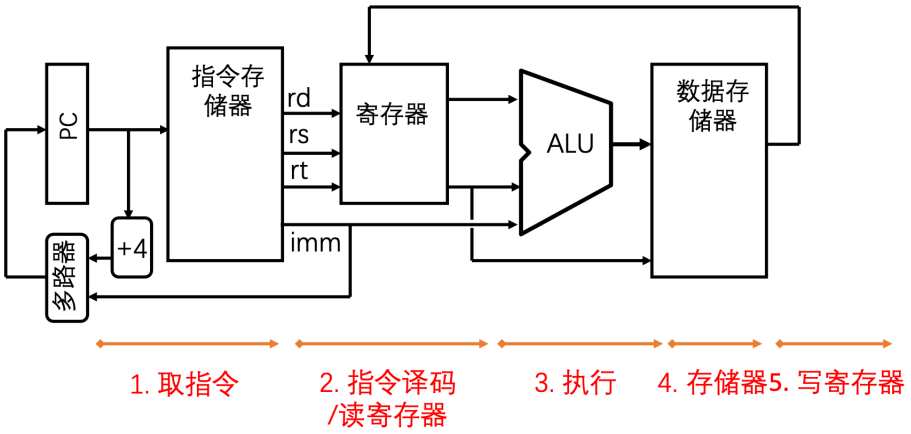

图 1: 数据通路结构

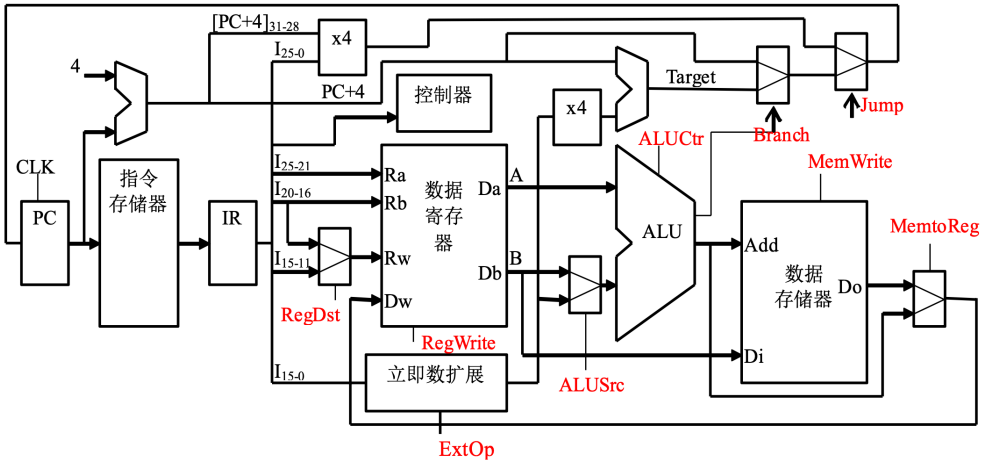

图 2: MIPS 处理器结构图

助记符	指令格式					
Bit #	31..26	25..21	20..16	15..11	10..6	5..0
R-type	op	rs	rt	rd	shamt	func
add	0	rs	rt	rd	0	100000
sub	0	rs	rt	rd	0	100010
and	0	rs	rt	rd	0	100100
or	0	rs	rt	rd	0	100101
slt	0	rs	rt	rd	0	101010
sll	0	0	rt	rd	shamt	0
srl	0	0	rt	rd	shamt	10
jr	0	rs	0	0	0	1000
I-type	op	rs	rt	immediate		
addi	1000	rs	rt	immediate		
andi	1100	rs	rt	immediate		
ori	1101	rs	rt	immediate		
lw	100011	rs	rt	immediate		
sw	101011	rs	rt	immediate		
beq	100	rs	rt	immediate		
J-type	op	address				
j	10	address				
jal	11	address				

表 1: 需要支持的 16 条指令及其相关信息

本次实验要求完成支持 16 条指令的处理器设计。表格 1给出了本次试验需要考虑的指令及其相应格式。根据这 16 条指令，给出相应完整通路结构图 2。图 1展示了单周期处理器的数据通路结构。

在结构图的帮助下，首先实现各个模块，其次通过程序连接各个数据线。失序性的模块例如 PC 和存储器 DataMemery 需要和系统时钟 clk 相关联，当下降沿到来时做出相应改变。非时序性模块例如多路选择器 MUX 和符号扩展器 SignExt 则将信号线内容实时更新。

2.2 流水化构建原理

将整个处理器的指令执行过程划分为五个阶段：取指令（IF）、指令译码（ID）、执行（EX）、访存（ME）、写回（WB）。五个阶段在时钟的不同沿上分别进行。

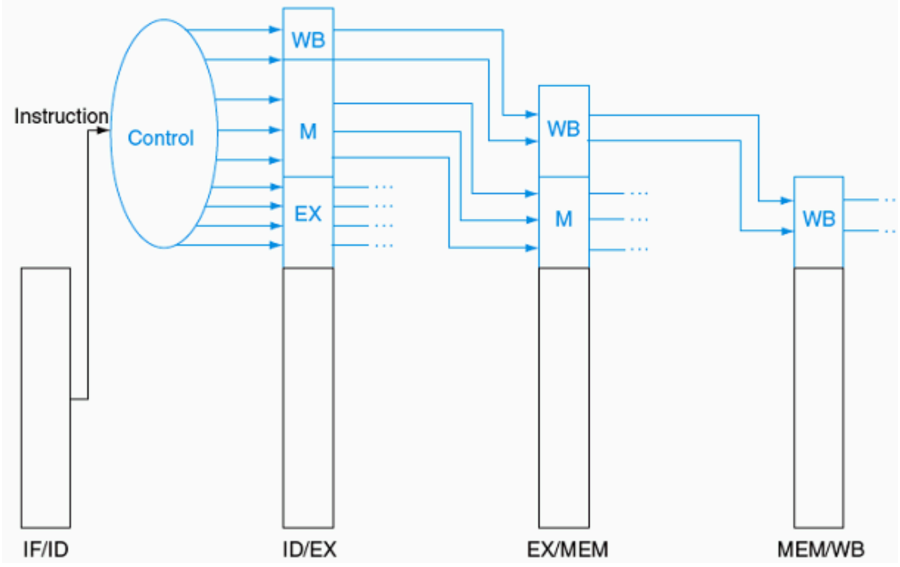

图 3: 数据通路结构

图 3展示了流水线处理器的数据通路结构。

2.2.1 取指令 IF 阶段工作原理

MIPS 流水处理器的 IF（Instruction Fetch）段是多级流水线中的第一级，主要负责从指令存储器中取出指令并传递给下一级流水线。包含有主要模块为指令存储器（在 IF 段的开始，PC 中保存的地址将被送入指令存储器，并将相应的指令读取到流水线中。）为方便进行转移预测的设计，移除 PC 与 PCupdate 模块。

在 IF 段中，由于只需要进行指令的读取和传递，因此 IF 段不需要使用寄存器进行状态保存，这使得 IF 段可以实现为组合逻辑电路，从而实现更高的时钟频率。然而，在多级流水线中，由于后续阶段需要等待 IF 段传递的指令，因此 IF 段需要能够在每个时钟周期结束时及时传递指令到下一级流水线，以保证后续阶段的正常运行。

2.2.2 指令译码 ID 阶段工作原理

MIPS 流水处理器的 ID（Instruction Decode）段是多级流水线中的第二级，主要完成指令的解码和寄存器操作数的读取。在 ID 阶段中，指令的 opcode 被提取并传递到控制单元进行指令译码和控制信号的生成，同时根据指令中的寄存器编号读取寄存器文件中的数据，并传递给流水线的下一阶段进行操作数的计算。

ID 阶段的主要任务包括以下几个方面：

- 从指令寄存器 IF/ID 中读取指令，并对其进行解码；

- 根据指令中包含的寄存器编号，从寄存器文件中读取操作数；
- 生成控制信号，并将其传递给流水线的下一阶段；

在 MIPS 流水处理器中，ID 阶段主要的模块包括指令寄存器 IF/ID、寄存器堆、控制单元、符号拓展器。其中指令寄存器用于存储从 IF 阶段传来的指令，寄存器堆用于读取指令中的寄存器编号对应的操作数，并将其传递给流水线的下一阶段，控制单元用于根据指令译码结果生成相应的控制信号。

ID 阶段是流水处理器中非常重要的一个阶段，它的正确性直接影响着整个处理器的正确性和性能。在 ID 阶段，需要完成诸如指令译码、寄存器读取、控制信号生成等一系列的任务，这些任务的正确性和性能都需要得到充分的考虑和优化，以确保流水处理器的正常运行和高效性能。

2.2.3 执行 EX 阶段工作原理

MIPS 流水处理器的 EX（Execution）段是多级流水线中的第三级，主要包括算术逻辑单元 (ALU) 的计算、数据存储器的读/写操作、控制单元对指令解码后的信号的生成等操作。在该阶段中，需要对指令的操作类型进行判断，并根据不同类型的指令执行不同的操作。常见的指令类型包括算术指令、逻辑指令、移位指令、分支指令等。

在执行指令时，需要对操作数进行选择和读取，同时进行运算并写回结果。对于寄存器-寄存器指令，操作数可以直接从寄存器文件中读取；对于寄存器-立即数指令，需要将立即数进行符号拓展后与寄存器内容进行运算。针对不同的操作数选择和读取方式，可以采用多路选择器进行控制。此外，还需要判断指令的跳转条件，并在满足跳转条件时更新 PC 寄存器的值，以实现分支、跳转等指令。

EX 阶段是 MIPS 流水线中最为复杂的一个阶段，需要进行多个操作，包括指令解码、ALU 计算、数据存储器读/写、控制信号生成等，同时还需要考虑分支跳转等问题，因此需要设计合理的流水线控制机制。

2.2.4 访存 ME 阶段工作原理

MIPS 流水处理器的 MEM（Memory）段是多级流水线中的第四级，其主要任务是执行访存操作，即将数据写入数据存储器中或者从数据存储器中读取数据。

在 MEM 段中，ALU 的输出作为访存地址，并将从数据存储器中读取到的数据存储在寄存器中，以便后续阶段的使用。此外，MEM 段还负责处理指令中的访存相关操作，例如 load 和 store 指令。

执行 load 指令时，MEM 段从数据存储器中读取数据，并将其存储在寄存器中。而在执行 store 指令时，MEM 段从寄存器中获取要写入数据存储器的数据，并将其写入数据存储器中。

2.2.5 写回 WB 阶段工作原理

MIPS 流水处理器的 WB（Write Back）段是多级流水线中的第五级，执行阶段计算的结果会被写回寄存器堆中，同时，流水线的状态会被更新。WB 段是 MIPS 流水线中的最后一个阶段。

在 WB 阶段中，执行阶段计算出的结果存储在一个内部的寄存器中，它需要被写回到寄存器堆中。写回的操作发生在时钟的下降沿。此时，需要通过控制信号 regWrite 来选择是否将执行阶段计算的结果写回到寄存器堆中。同时，在 WB 阶段，还需要更新流水线的一些状态，比如 PC 寄存器中存储的下一条指令地址，以及控制信号等。这些更新操作需要在时钟下降沿之后进行，以确保在下一次时钟上升沿时，新的指令可以被取出。

WB 阶段完成后，整个指令的执行过程就结束了。下一条指令将会在 IF 阶段被取出，整个流水线会重新开始执行。

2.3 冒险与相关性

在流水线处理器中，由于指令的执行存在数据相关性问题，可能会导致数据冒险（Data Hazard）和控制冒险（Control Hazard），从而影响程序的正确性和执行效率。

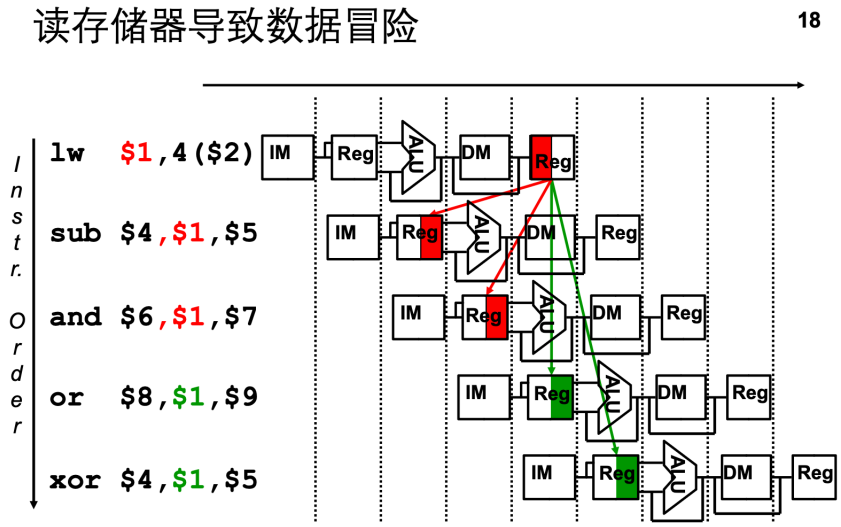

图 4: 数据冒险的一个例子

数据相关性问题是由于前一条指令的结果需要用于后一条指令的运算，但前一条指令还没有完成执行，所以后一条指令需要等待前一条指令执行完成后才能继续执行。这种情况下，如果后一条指令需要访问前一条指令正在使用的寄存器或内存，就会产生数据冒险，可能会导致后一条指令读取到不正确的数据，从而产生错误的结果。

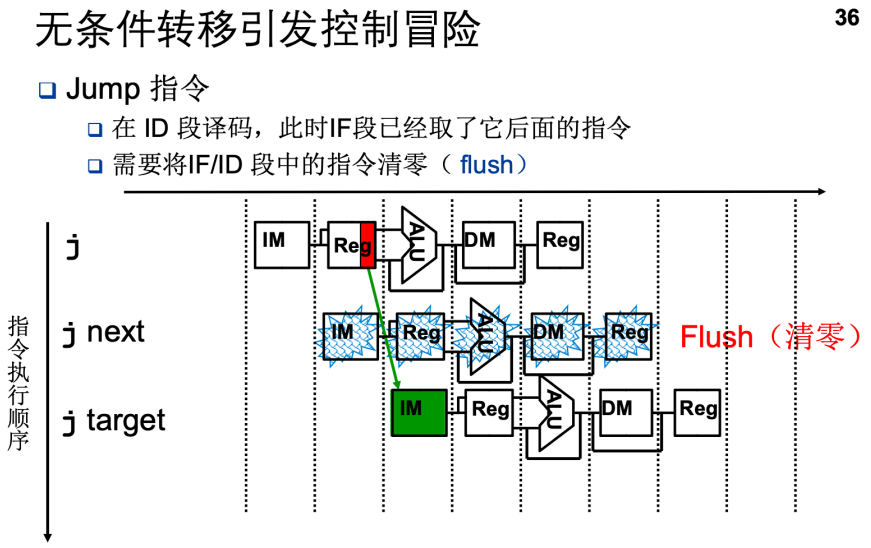

图 5: 控制冒险的一个例子

控制相关性问题是由于分支指令的存在，使得后续的指令流可能会受到分支指令的影响而产生跳转，从而导致程序的执行路径发生变化，需要在分支指令执行之前就预测分支的结果，否则会产生控制冒险，浪费 CPU 的时钟周期。

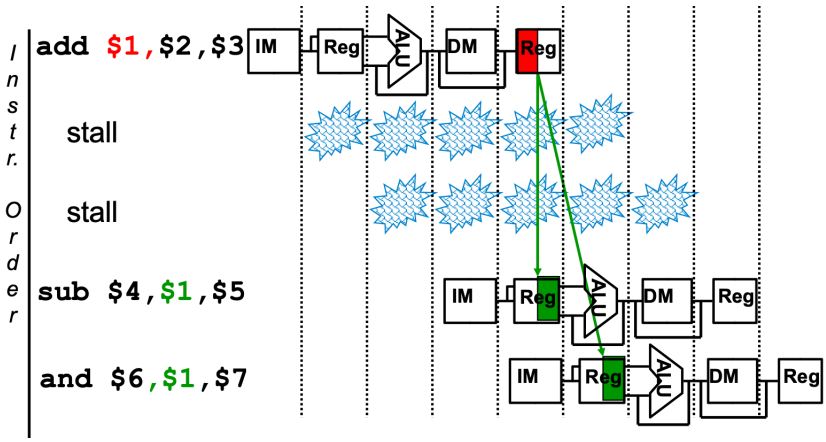

图 6: STALL 机制示例

为了解决这些问题，需要使用一些技术来避免或减少冒险的影响。其中一个重要的技术是流水线停顿 STALL 机制，即在出现冒险时暂停流水线的进程，等待相关的数据准备就绪后再继续执行。在 STALL 机制中，流水线中的某些阶段会发出一个暂停指令（STALL 信号），告诉前一级不要产生新的指令，等待数据准备好后再继续执行。当数据准备好后，暂

停信号解除，前一级的指令重新开始产生。这种机制可以避免数据冒险的问题，但是会浪费一些时钟周期，从而降低处理器的执行效率。

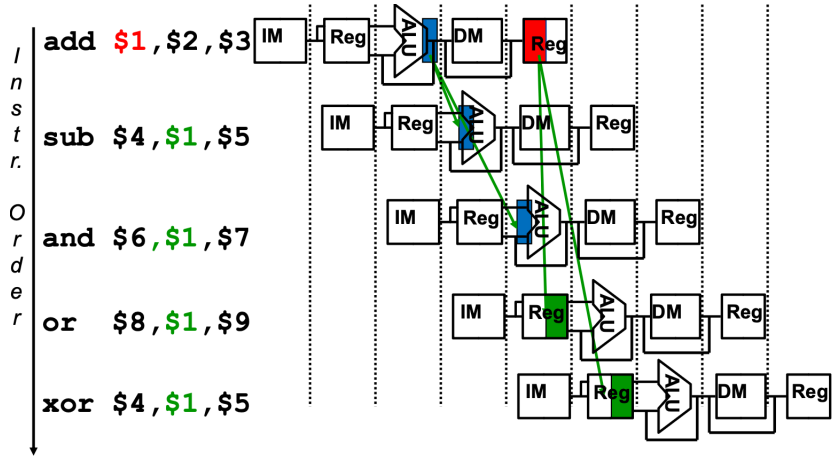

图 7: Forwarding 机制示例

另一种优化方案是使用向前传递 Forwarding 机制，也称为数据旁路机制。当一个指令需要从另一个指令计算的结果中读取数据时，可以通过直接将结果从上一个执行阶段中的寄存器或执行单元转发到下一个阶段中的寄存器或执行单元，避免了需要等待内存读写的情况，从而减少流水线的停顿，提高了流水线的效率。

需要注意的是，forwarding 机制只能用于对数据的读取操作，对于数据的写入操作仍然需要等待前面指令的执行完成。

2.4 predict-not-taken 转移预测

在 MIPS 流水线处理器中，predict-not-taken 转移预测的设计原理是基于分支指令的特性和历史执行情况。当 CPU 执行分支指令时，通常需要暂停流水线，并等待分支条件的确定。为了减少暂停流水线的时间，流水线处理器使用预测器来预测分支的执行结果。predict-not-taken 转移预测就是基于分支指令的历史执行情况，预测分支不会被执行，即预测分支条件为假。

具体实现上，predict-not-taken 转移预测使用一个简单的状态机。状态机有两个状态，一个是预测分支不会被执行的状态，另一个是预测分支会被执行的状态。当遇到分支指令时，状态机切换到预测分支会被执行的状态。如果分支条件为真，则预测正确，状态机保持在预测分支会被执行的状态，CPU 流水线继续按照预测执行；如果分支条件为假，则预测错误，状态机切换回预测分支不会被执行的状态，CPU 流水线进行重启，从预测分支不会被执行的状态重新执行。

predict-not-taken 转移预测的设计原理是基于分支指令历史执行情况和对 CPU 流水线

暂停时间的优化，可以有效地提高 CPU 流水线的执行效率。然而，在实际应用中，由于分支指令执行结果的难以预测，predict-not-taken 转移预测仅能达到一定的效果，无法完全消除 CPU 流水线暂停的影响。

2.5 命名规则及所需模块

为避免重名和方便辨认，首先定义命名规则如下：

- 修改数据通路信号线定义为 `wire [a:b] IF_INST`。即 所处的流水段-下划线-全部大写字母。
- 模块命名为 `Register register_module()`。即 小写字母_module。
- 模块内信号线命名为 `input [31:0] data0`。即 大小写混杂（需要reg时开头小写）。
- 模块内寄存器命名为 `reg [3:0] ALUCtrOut`。即将所 `assign` 的命名改为部分大写，通常开头大写。

所需模块如表 2所示。

模块	描述
InstMem	指令寄存器，得到 PC 后给出相应位置指令
Ctr	控制单元，处理指令给出各种控制信号
ALUCtr	ALU 控制单元，通过 Ctr 给出的 ALUOP 决定 ALU 操作
ALU	通过 ALUCtr 给出的信号进行运算
signext	进行带符号或不带符号拓展
dataMemory	存储器，下降沿时进行写操作，根据 Ctr 进行读操作
MUX	32 位的多路选择器
MUX_5	5 位的多路选择器

表 2: 所需模块及其描述

3 流水线基本功能实现

3.1 取指令 IF 阶段功能实现

定义了一个包含 32 位地址 IF_PCIN 和 32 位指令 IF_INST 的寄存器和线路变量。同时，该段代码还包括了一个指令存储器模块 InstMem，用于读取指令存储器中存储的指令并将其输出到 IF_INST 中。以此完成了从指令存储器中取出指令的功能。

```
    // Instruction Fetch Segement
    reg [31:0] IF_PCIN;
    wire [31:0] IF_INST;

    // instruction memory
    InstMem instmem_module (
        .readAddr(IF_PCIN),
        .clk(clk),
        .reset(reset),
        .instr(IF_INST)
    );
```

构建 IF/ID 段的两个段寄存器。

```
    // IF/ID Register
    reg [31:0] IFID_PC;
    reg [31:0] IFID_INST;
```

用于存储 IF 阶段得到的指令地址和指令内容。

3.2 指令译码 ID 阶段功能实现

仿照 lab5 的代码，将相应部分的信号线更换为新定义的信号线，并将需要传到其后相应阶段寄存器的内容导向段寄存器。

实现代码如下：

```
    // Instruction Decode Segement
    wire ID_REGDST;
    wire ID_ALUSRC;
    wire ID_MEMTOREG;
    wire ID_MEMREAD;
    wire ID_REGWRITE;
    wire ID_MEMWRITE;
    wire ID_BRANCH;
    wire ID_EXTSIGN;
    wire ID_JALSIGN;
    wire ID_JUMP;
    wire [2:0] ID_ALUOP;

    // Control Unit
    Ctr ctr_module (
```

```
16          .opCode(IFID_INST[31 : 26]),
17          .regDst(ID_REGDST),
18          .aluSrc(ID_ALUSRC),
19          .memToReg(ID_MEMTOREG),
20          .regWrite(ID_REGWRITE),
21          .memRead(ID_MEMREAD),
22          .memWrite(ID_MEMWRITE),
23          .branch(ID_BRANCH),
24          .aluOp(ID_ALUOP),
25          .jump(ID_JUMP),
26          .extSign(ID_EXTSIGN),
27          .jalSign(ID_JALSIGN)
28      );
29
30      wire [31:0] ID_EXTRES;
31
32      // EXTSIGN
33      signext signext_module (
34          .extSign(ID_EXTSIGN),
35          .inst(IFID_INST[15 : 0]),
36          .data(ID_EXTRES)
37      );
38
39      wire [4:0] WB_WRITEREG;
40      wire [31:0] ID_REGOUT1;
41      wire [31:0] ID_REGOUT2;
42      wire [31:0] WB_REG_WRITE_DATA;
43      wire WB_REGWRITE;
44
45      // register
46      Register register_module (
47          .readReg1(IFID_INST[25 : 21]),
48          .readReg2(IFID_INST[20 : 16]),
49          .writeReg(WB_WRITEREG),
50          .writeData(WB_REG_WRITE_DATA),
51          .regWrite(WB_REGWRITE),
52          .clk(clk),
53          .reset(reset),
54          .readData1(ID_REGOUT1),
55          .readData2(ID_REGOUT2)
56      );
57
```

```
    wire [4 : 0] ID_WRITE_REG_TEMP;
    // rd or rt
    MUX_5 rt_rd_selector (
        .Select(ID_REGDST),
        .data0(IFID_INST[15 : 11]),
        .data1(IFID_INST[20 : 16]),
        .data(ID_WRITE_REG_TEMP)
    );
```

构建 ID/EX 段的两个段寄存器。

```
    // ID/EX register
    reg [2:0] IDEX_ALUOP;
    reg IDEX_ALUSRC;
    reg IDEX_MEMTOREG;
    reg IDEX_MEMREAD;
    reg IDEX_REGWRITE;
    reg IDEX_MEMWRITE;
    reg IDEX_BRANCH;
    reg [31:0] IDEX_EXTRES;
    reg [4:0] IDEX_RS;
    reg [4:0] IDEX_RT;
    reg [31:0] IDEX_REGOUT1;
    reg [31:0] IDEX_REGOUT2;
    reg [5:0] IDEX_FUNCT;
    reg [4:0] IDEX_SHAMT;
    reg [4:0] IDEX_REGDEST;
    reg [31:0] IDEX_PC;
```

用于存储 ID 阶段得到的指令地址和指令内容。

3.3 执行 EX 阶段功能实现

仿照 lab5 的代码，将相应部分的信号线更换为新定义的信号线，并将需要传到其后相应阶段寄存器的内容导向段寄存器。

实现代码如下：

```
    // Execution Segment
    wire [3:0] EX_ALUCTR;
    wire EX_SHAMTSIGN;

    // ALU Controller
```

```
    ALUCtr aluctr_module (
        .aluOp(IDEX_ALUOP),
        .funct(IDEX_FUNCT),
        .aluCtrOut(EX_ALUCTR),
        .shamtSign(EX_SHAMTSIGN),
        .jrSign(EX_JRSIGN)
    );

    wire [31:0] EX_ALUIN1;
    wire [31:0] EX_ALUIN2;
    wire EX_ALUZERO;
    wire [31:0] EX_ALURES;

    // INST[10:6] or rs
    MUX rs_shamt_selector (
        .Select(EX_SHAMTSIGN),
        .data0({27'b000000000000000000000000000, IDEX_SHAMT}),
        .data1(IDEX_REGOUT1),
        .data(EX_ALUIN1)
    );

    // EXTRES or rt
    MUX rt_ext_selector (
        .Select(IDEX_ALUSRC),
        .data0(IDEX_EXTRES),
        .data1(IDEX_REGOUT2),
        .data(EX_ALUIN2)
    );

    // ALU
    ALU alu_module (
        .input1(EX_ALUIN1),
        .input2(EX_ALUIN2),
        .aluCtr(EX_ALUCTR),
        .zero(EX_ALUZERO),
        .aluRes(EX_ALURES)
    );

    wire [31 : 0] BRANCH_DEST = IDEX_PC + 4 + (IDEX_EXTRES << 2);
```

构建 EX/MA 段的两个段寄存器。

```
    // EX/MA register
    reg EXMA_MEMTOREG;
    reg EXMA_MEMREAD;
    reg EXMA_REGWRITE;
    reg EXMA_MEMWRITE;
    reg [31:0] EXMA_ALURES;
    reg [31:0] EXMA_REGOUT2;
    reg [4:0] EXMA_REGDEST;
```

用于存储 EX 阶段得到的指令地址和指令内容。

3.4 访存 ME 阶段功能实现

仿照 lab5 的代码，将相应部分的信号线更换为新定义的信号线，并将需要传到其后相应阶段寄存器的内容导向段寄存器。

实现代码如下：

```
    // Memery Segment
    wire [31:0] MA_MEM_READ_DATA;
    // data memory
    dataMemory datamemory_module (
        .clk(clk),
        .address(EXMA_ALURES),
        .writeData(EXMA_REGOUT2),
        .memWrite(EXMA_MEMWRITE),
        .memRead(EXMA_MEMREAD),
        .readData(MA_MEM_READ_DATA)
    );

    wire [31:0] MA_REG_WRITE_DATA_T;

    // MEM READ DATA or ALURES
    MUX mem_alu_selector (
        .Select(EXMA_MEMTOREG),
        .data0(MA_MEM_READ_DATA),
        .data1(EXMA_ALURES),
        .data(MA_REG_WRITE_DATA_T)
    );
```

构建 ME/WB 段的两个段寄存器。

```
    // ME/WB register
```

```
    reg MAWB_REGWRITE;
    reg [31:0] MAWB_FINALDATA;
    reg [4:0] MAWB_REGDEST;
```

用于存储 ME 阶段得到的指令地址和指令内容。

3.5 写回 WB 阶段功能实现

仿照 lab5 的代码，将相应部分的信号线更换为新定义的信号线，并将需要传到其后相应阶段寄存器的内容导向段寄存器。

实现代码如下：

```
    // Write Back Segment
    MUX jal_selector (
        .Select(ID_JALSIGN),
        .data0(IFID_PC + 4),
        .data1(MAWB_FINALDATA),
        .data(WB_REG_WRITE_DATA)
    );

    assign WB_WRITEREG = MAWB_REGDEST;
    assign WB_REGWRITE = MAWB_REGWRITE;
```

3.6 信号线与寄存器传递部分实现

在 reset 信号与时钟的控制下，将各部分的信号与寄存器进行传递，代码如下：

```
    always @(reset)begin
        if (reset) begin
            IF_PCIN = 0;
            IFID_INST = 0;
            IFID_PC = 0;
            IDEX_ALUOP = 0;
            IDEX_ALUSRC = 0;
            IDEX_MEMTOREG = 0;
            IDEX_MEMREAD = 0;
            IDEX_REGWRITE = 0;
            IDEX_MEMWRITE = 0;
            IDEX_BRANCH = 0;
            IDEX_EXTRES = 0;
            IDEX_RS = 0;
```

```
            IDEX_RT = 0;
            IDEX_REGOUT1 = 0;
            IDEX_REGOUT2 = 0;
            IDEX_FUNCT = 0;
            IDEX_SHAMT = 0;
            IDEX_REGDEST = 0;
            IDEX_PC = 0;
            EXMA_MEMTOREG = 0;
            EXMA_MEMREAD = 0;
            EXMA_REGWRITE = 0;
            EXMA_MEMWRITE = 0;
            EXMA_ALURES = 0;
            EXMA_REGOUT2 = 0;
            EXMA_REGDEST = 0;
            MAWB_REGWRITE = 0;
            MAWB_FINALDATA = 0;
            MAWB_REGDEST = 0;
        end
    end

    always @(posedge clk)
    begin
        // EX/MA -> MA/WB
        MAWB_REGWRITE <= EXMA_REGWRITE;
        MAWB_REGDEST <= EXMA_REGDEST;
        MAWB_FINALDATA <= MA_REG_WRITE_DATA_T;

        // ID/EX -> EX/MA
        EXMA_MEMTOREG <= IDEX_MEMTOREG;
        EXMA_MEMREAD <= IDEX_MEMREAD;
        EXMA_REGWRITE <= IDEX_REGWRITE;
        EXMA_MEMWRITE <= IDEX_MEMWRITE;
        EXMA_ALURES <= EX_ALURES;
        EXMA_REGOUT2 <= IDEX_REGOUT2;
        EXMA_REGDEST <= IDEX_REGDEST;

        // IF/ID -> ID/EX
        IDEX_ALUOP <= ID_ALUOP;
        IDEX_ALUSRC <= ID_ALUSRC;
        IDEX_MEMTOREG <= ID_MEMTOREG;
        IDEX_MEMREAD <= ID_MEMREAD;
        IDEX_REGWRITE <= ID_REGWRITE;
```

```
57          IDEX_MEMWRITE <= ID_MEMWRITE;
58          IDEX_BRANCH <= ID_BRANCH;
59          IDEX_EXTRES <= ID_EXTRES;
60          IDEX_RS <= IFID_INST [25 : 21];
61          IDEX_RT <= IFID_INST [20 : 16];
62          IDEX_REGOUT1 <= ID_REGOUT1;
63          IDEX_REGOUT2 <= ID_REGOUT2;
64          IDEX_FUNCT <= IFID_INST [5 : 0];
65          IDEX_SHAMT <= IFID_INST [10 : 6];
66          IDEX_REGDEST <= ID_WRITE_REG_TEMP;
67          IDEX_PC <= IFID_PC;
68
69          // PCupdate -> IF/ID
70          IF_PCIN <= PC_BEQ_END;
71          IFID_INST <= IF_INST;
72          IFID_PC <= IF_PCIN;
73
74      end
```

4 流水线特殊机制功能实现

考虑将 Forwarding，Stall，predict-not-taken 统合设计。

4.1 Forwarding 机制实现

Forwarding 机制主要涉及到了 EX 段和 MEM/WB 段产生的数据，因此 forwarding 机制主要在 EX 段中实现。

设计四个 MUX 模块。其中，forward1_1 模块的作用是判断是否有 MA/WB 阶段写回的寄存器等待 ID/EX 阶段的寄存器读取（即是否存在数据相关），如果有，则将 MA/WB 阶段的数据（MAWB_FINALDATA）发送给 EX 段需要的寄存器，否则传递 ID/EX 段的数据（IDEX_REGOUT1）给 EX 段。同样的，forward1_2 模块的作用是判断是否有 EX/MEM 阶段写回的寄存器等待 ID/EX 阶段的寄存器读取，如果有，则将 EX/MEM 阶段的数据（EXMA_ALURES）发送给 EX 段需要的寄存器，否则传递 forward1_1 模块的结果。

代码中的 forward2_1 和 forward2_2 模块与 forward1_1 和 forward1_2 类似，但是针对的是第二个需要读取的寄存器，即 ID/EX 段指令中的 rt 字段，与 IDEX_REGOUT2 有关。

通过这些 MUX 模块的组合，可以实现数据的转发，避免了数据冒险带来的性能损失。

实现代码如下；

```
    // Forwarding
    wire [31 : 0] EX_ALU1_FORTEMP;
    wire [31 : 0] EX_ALU2_FORTEMP;
    MUX forward1_1(
        .Select(MAWB_REGWRITE & (MAWB_REGDEST == IDEX_RS)),
        .data0(MAWB_FINALDATA),
        .data1(IDEX_REGOUT1),
        .data(EX_ALU1_FORTEMP));
    MUX forward1_2(
        .Select(EXMA_REGWRITE & (EXMA_REGDEST == IDEX_RS)),
        .data0(EXMA_ALURES),
        .data1(EX_ALU1_FORTEMP),
        .data(EX_ALU1_FORWARDING));

    MUX forward2_1(
        .Select(MAWB_REGWRITE & (MAWB_REGDEST == IDEX_RT)),
        .data0(MAWB_FINALDATA),
        .data1(IDEX_REGOUT2),
        .data(EX_ALU2_FORTEMP));
    MUX forward2_2(
        .Select(EXMA_REGWRITE & (EXMA_REGDEST == IDEX_RT)),
        .data0(EXMA_ALURES),
        .data1(EX_ALU2_FORTEMP),
        .data(EX_ALU2_FORWARDING));
```

4.2 Stall 机制实现

在 Forwarding 机制实现后，需要 Stall 的情况大大减少。

新定义 Stall 信号，当 IDEX 阶段正在进行内存读取操作并且读取到的数据将被写入到寄存器中，同时在 IFID 阶段中的指令操作数（寄存器地址）与 IDEX 阶段的结果寄存器地址匹配时，需要使用 stall 策略来暂停 IFID 和 IDEX 之间的流水线，以避免读写操作之间的冲突。

在暂停操作时，将 IDEX 阶段的所有寄存器重置为 0，以防止冲突。这样，IFID 和 IDEX 之间的流水线被暂停，直到冲突解决并且 stall 标志被清除为止，才可以继续执行下一条指令。实现代码如下：

```
    // Stall
    wire STALL = IDEX_MEMREAD &
        ((IDEX_RT == IFID_INST [25 : 21]) |
```

```
        (IDEX_RT == IFID_INST [20 : 16]));

always @(reset)
...
always @(posedge clk)
    begin
        // IF/ID -> ID/EX
        if (STALL || (IDEX_BRANCH & EX_ALUZERO))
        begin
            IDEX_ALUOP <= 4'b1111;
            IDEX_ALUSRC <= 0;
            IDEX_MEMTOREG <= 0;
            IDEX_MEMREAD <= 0;
            IDEX_REGWRITE <= 0;
            IDEX_MEMWRITE <= 0;
            IDEX_BRANCH <= 0;
            IDEX_EXTRES <= 0;
            IDEX_RS <= 0;
            IDEX_RT <= 0;
            IDEX_REGOUT1 <= 0;
            IDEX_REGOUT2 <= 0;
            IDEX_FUNCT <= 0;
            IDEX_SHAMT <= 0;
            IDEX_REGDEST <= 0;
            IDEX_PC <= IFID_PC;
        end
        else begin
            IDEX_ALUOP <= ID_ALUOP;
            IDEX_ALUSRC <= ID_ALUSRC;
            IDEX_MEMTOREG <= ID_MEMTOREG;
            IDEX_MEMREAD <= ID_MEMREAD;
            IDEX_REGWRITE <= ID_REGWRITE;
            IDEX_MEMWRITE <= ID_MEMWRITE;
            IDEX_BRANCH <= ID_BRANCH;
            IDEX_EXTRES <= ID_EXTRES;
            IDEX_RS <= IFID_INST [25 : 21];
            IDEX_RT <= IFID_INST [20 : 16];
            IDEX_REGOUT1 <= ID_REGOUT1;
            IDEX_REGOUT2 <= ID_REGOUT2;
            IDEX_FUNCT <= IFID_INST [5 : 0];
            IDEX_SHAMT <= IFID_INST [10 : 6];
            IDEX_REGDEST <= ID_WRITE_REG_TEMP;
```

```
            IDEX_PC <= IFID_PC;
        end

        // ID/EX -> EX/MA
        EXMA_MEMTOREG <= IDEX_MEMTOREG;
        EXMA_MEMREAD <= IDEX_MEMREAD;
        EXMA_REGWRITE <= IDEX_REGWRITE;
        EXMA_MEMWRITE <= IDEX_MEMWRITE;
        EXMA_ALURES <= EX_ALURES;
        EXMA_REGOUT2 <= IDEX_REGOUT2;
        EXMA_REGDEST <= IDEX_REGDEST;

        // EX/MA -> MA/WB
        MAWB_REGWRITE <= EXMA_REGWRITE;
        MAWB_REGDEST <= EXMA_REGDEST;
        MAWB_FINALDATA <= MA_REG_WRITE_DATA_T;

        // PCupdate -> IF/ID
        if (! STALL)
        begin
            IF_PCIN <= PC_BEQ_END;
            IFID_INST <= IF_INST;
            IFID_PC <= IF_PCIN;
        end
    end
```

4.3 predict-not-taken 机制实现

在我的设计中，MUX jump_selector 实现了当 ID_JUMP 为 1 时，选择跳转目标地址 (IFID_INST [25 : 0] « 2)，否则选择下一条指令的地址 (IF_PCIN + 4)。当分支指令发生时，MUX beq_selector 选择跳转目标地址 (BRANCH_DEST)，否则选择 PC_JUMP_END，即由 jump_selector 选择的下一条指令的地址。

如果预测错误，即发生了分支指令，此时需要清空 IF/ID 和 ID/EX 寄存器中的所有值，避免错误的指令执行。此处采用的方法是将 STALL 赋值为 1，此时 if 语句块内的代码将被执行，清空 ID/EX 寄存器中的所有值。这样，处理器就能在分支指令发生时正确地更新 PC，并执行分支指令。

实现代码如下：

```
    // PC update
    wire [31:0] PC_JUMP_END;
```

```
    MUX jump_selector(
        .Select(ID_JUMP),
        .data0(((IFID_PC + 4) & 32'hf0000000) + (IFID_INST [25 : 0] << 2)),
        .data1(IF_PCIN + 4),
        .data(PC_JUMP_END));

    wire BEQ_BRANCH = IDEX_BRANCH & EX_ALUZERO;
    wire [31 : 0] PC_BEQ_END;
    MUX beq_selector(
        .Select(BEQ_BRANCH),
        .data0(BRANCH_DEST),
        .data1(PC_JUMP_END),
        .data(PC_BEQ_END));
```

5 仿真测试

5.1 基础流水线处理器测试

根据给定的汇编语言参考样例编写机器语言代码:

```
10001100000000010000000000000010 // lw $2, 2($0)
10001100000000010000000000000001 // lw $1, 1($0)
10001100000000110000000000000011 // lw $3, 2($0)
00000000011000010010100000100010 // sub $5, $3, $1
00000000001000100010000000100000 // add $4, $1, $2
00000000010000010011000000100100 // and $6, $2, $1
10001100000010100000000000000001 // lw $10, 1($0)
10001100000010100000000000000001 // lw $10, 1($0)
10001100000010100000000000000001 // lw $10, 1($0)
00000000011000010011100000100101 // or $7, $3, $1
00000000011000010100000000101010 // slt $8, $3, $1
00010000000000000000000000000010 // beq $0, $0, end
00000000111010000100100000100000 // add $9, $7, $8
10001100000010100000000000000011 // end: lw $10, 3($0)
10001100000010100000000000000010 // lw $10, 2($0)
10001100000010100000000000000001 // lw $10, 1($0)
10001100000010100000000000000001 // lw $10, 1($0)
10001100000010100000000000000001 // lw $10, 1($0)
```

初始化存储器内容为

```
00000000
00000001
00000005
00000008
```

得到基础流水线测试结果如图 8。

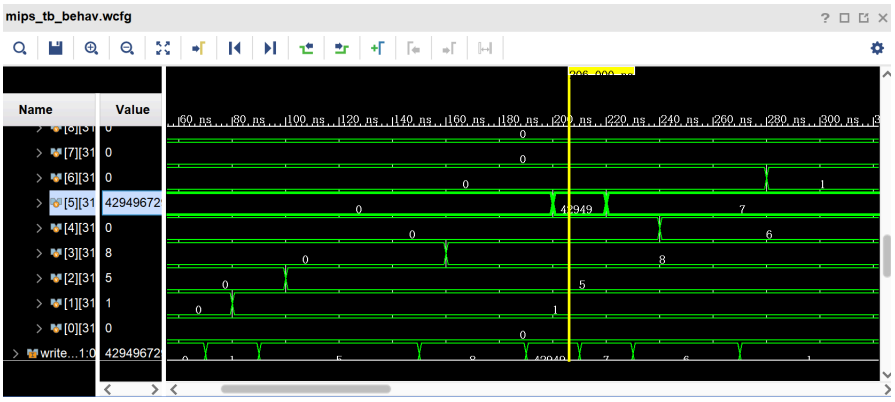

图 8: 基础流水线处理器测试

可以看到，发生了 hazard 情况，其余部分符合要求。

5.2 优化后流水线处理器测试

增加 Stall，Forwarding，predict-not-taken 机制后进行仿真测试，得到结果如图 9。

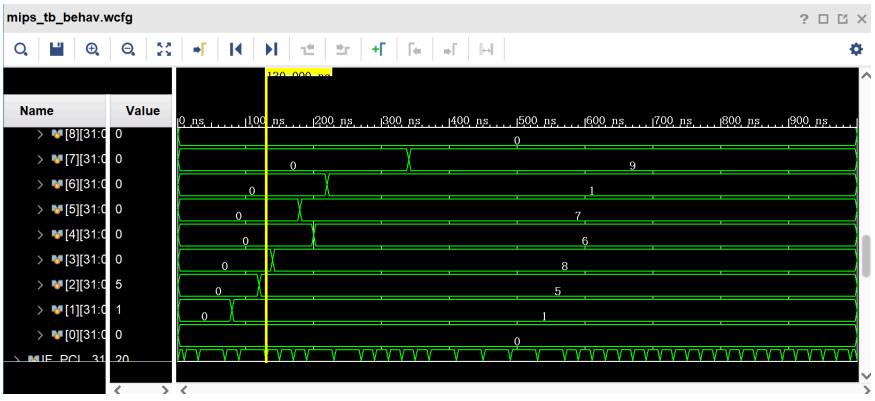

图 9: 优化后流水线处理器测试

可以看到，hazard 被消除，仿真波形与预期结果相同（由于增加了对 branch 的测试，仿真波形与参考波形略有不同）。

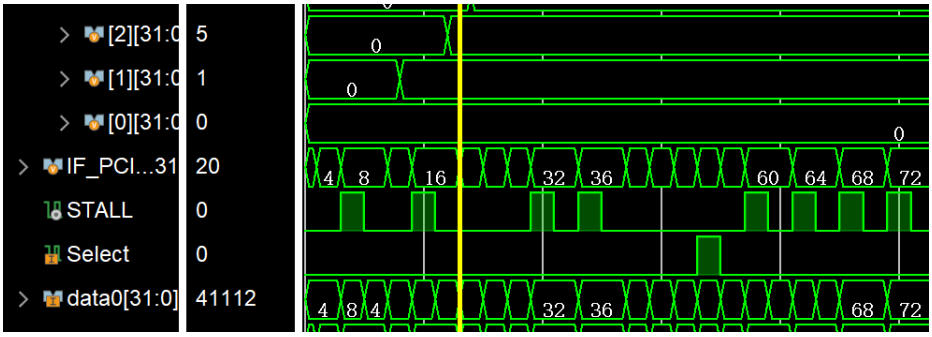

图 10: Stall 信号输出波形

此外，图 10展示了 Stall 信号的输出波形，说明 Stall 机制正确性。

6 遇到的问题与解决

6.1 仿真文件读取问题

在进行仿真测试与代码调式时，忘记采用的读取文件方式为绝对路径，而在备份文件夹后，修改文件夹内指令代码时出现预料之外的结果。这一错误十分低级，但忽视之后给我带来不小的困扰。最终在仿真时对照 InstMem 内容才慢慢发现问题所在。

6.2 调试工作

在进行信号线替换时，有时会不知道从何处获取该信号，由于调试工作的抽象性，总结如下调试和检查方法：

- 对照结构图和流水线示意图，检查各个信号线的来源与去向
- 进行仿真，逐一检查所需模块的信号是否符合预期
- 修改指令文件和存储内容，通过增添和删减所需指令，对照仿真情况能够更好观察各个指令的执行情况
- 使用 IP 核的 debug 功能实时监控信号值的变化，查看处理器的状态信息

7 总结与反思

在本次实验中，我学习了 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；在此基础上，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并增加 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；此外，还通过 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。最后通过仿真测试证明了流水线处理器的正确性。

lab6 是对 MIPS 处理器的又一次扩充，也是对处理器流水线的进一步学习。本次实验在软硬件协同的基础上加深我对计算机处理器流水线的认识和理解，同时锻炼了自己通过更抽象的方式编写，调试代码的能力。

在调试调试过程中，我通过学习和研究总结出了软硬件协同调试的方法，也发现一些初级错误可能带来的惯性后果，加深了对计算机编程，工作和细致度的理解。收获颇丰。

8 致谢

刘雨桐老师为实验提供了良好的讲解，在课程中时常做出对于任务难度和时间上的提醒，并十分体谅学生，能理解我们的困难并给予帮助，深表感谢。

实验过程中，助教蔡明昕、黄正翔老师多次解决我的困惑和实验中遇到的问题，并且在困难的时候给予我鼓励和帮助，对此深表感激。

感谢黄小平老师及实验室提供的资源和硬件支持。

感谢在实验过程中给予我帮助的同学。

邓倩妮老师为使得教学效果更好，将课程体系由 RISC-V 改为 MIPS，在实验中提供很多帮助，并且本文一部分图片来自邓倩妮老师计算机体系结构课程 ppt，对此表示感谢。