Lab06　简单的类 MIPS 多周期流水化处理器实现

w1049

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. 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）

2 原理分析

2.1 总体框架

本次实验的主要参考是图 1 （来自《计算机组成与设计：硬件/软件接口》），在其基础上进行部分改动（参考《深入理解计算机系统》），得到了最终的流水化 CPU。

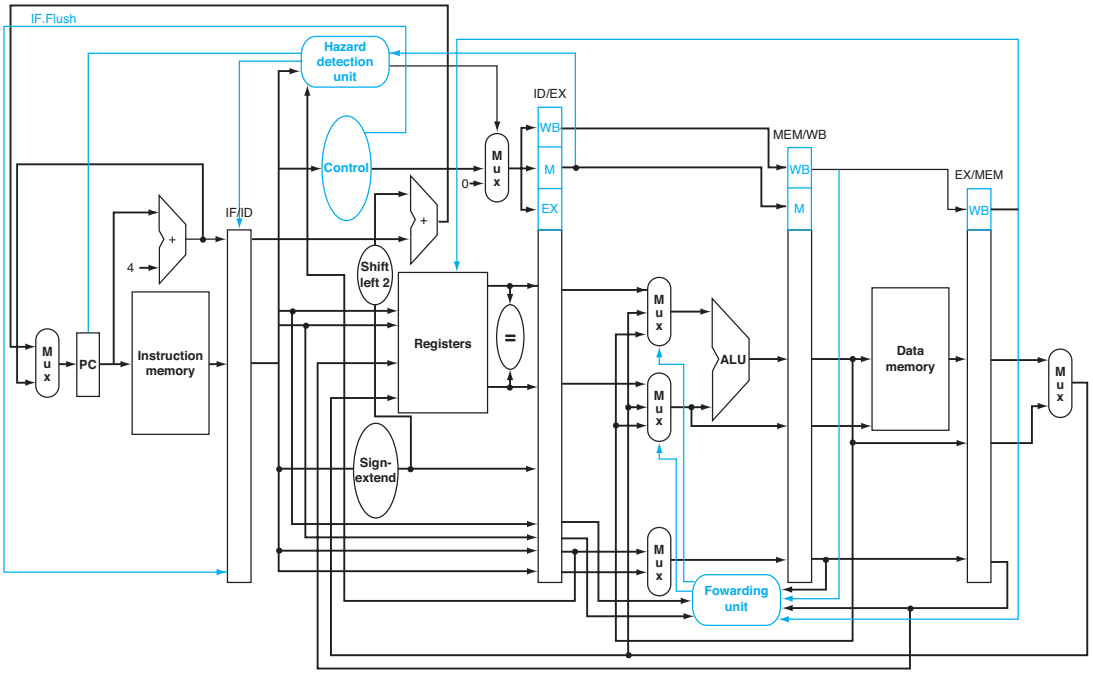

图 1: 流水线设计图

把单周期处理器改为多周期、流水线处理器，需要划分阶段，在阶段之间插入流水线寄存器。在实现了基础功能后，再进行优化，加入冒险检测、停顿、转发等功能，最终得到一个完整的流水线处理器。

3 功能实现

3.1 流水线寄存器

我将流水线寄存器和 PC 整合为一个模块，即一个可以控制位宽的、每周期更新的寄存器：

```
module Register#(parameter MSB = 31)(
    input Clk,
```

```
    input reset,
    input [MSB:0] in,
    output reg [MSB:0] out
    );
    always @(posedge Clk) begin
        if (reset) out <= 0;
        else out <= in;
    end
endmodule
```

PC 就可以定义成：

```
    Register#(.MSB(31)) PC (
        .Clk(Clk),
        .reset(reset),
        .in(IF_newpc),
        .out(IF_pc)
    );
```

其他寄存器也可以类似定义，只需要修改位宽即可。比如 IFID 寄存器：

```
    parameter IFID_MSB = 63;
    wire [IFID_MSB:0] IFID;
    // 31:0 inst 32
    // 63:32 pc+4 32
    Register#(.MSB(IFID_MSB)) IFID_reg (
        .Clk(Clk),
        .reset(reset),
        .in({IF_pc + 4, IF_inst}),
        .out(IFID)
    );
```

这么做的缺点很明显：需要记住下标范围与变量的对应关系。如果添加新的变量，应该添加到高位，这样不会影响低位已有的代码。使用 `parameter`，可以在一处修改位宽，而不用修改其他代码。

还有一种实现流水线寄存器的做法，是对每个寄存器定义一个模块，这样就可以直接使用变量名，但是代码量较大。

3.2 简单流水线

有了流水线寄存器，我们就可以划分阶段实现简单的流水线了。

3.2.1 数据通路

首先仍然是数据通路。把处理器的几个部件放到不同的阶段中，可以参考图 2：

- IF：指令内存、PC
- ID：寄存器堆
- EX：ALU
- MEM：数据内存
- WB：仍然是寄存器堆

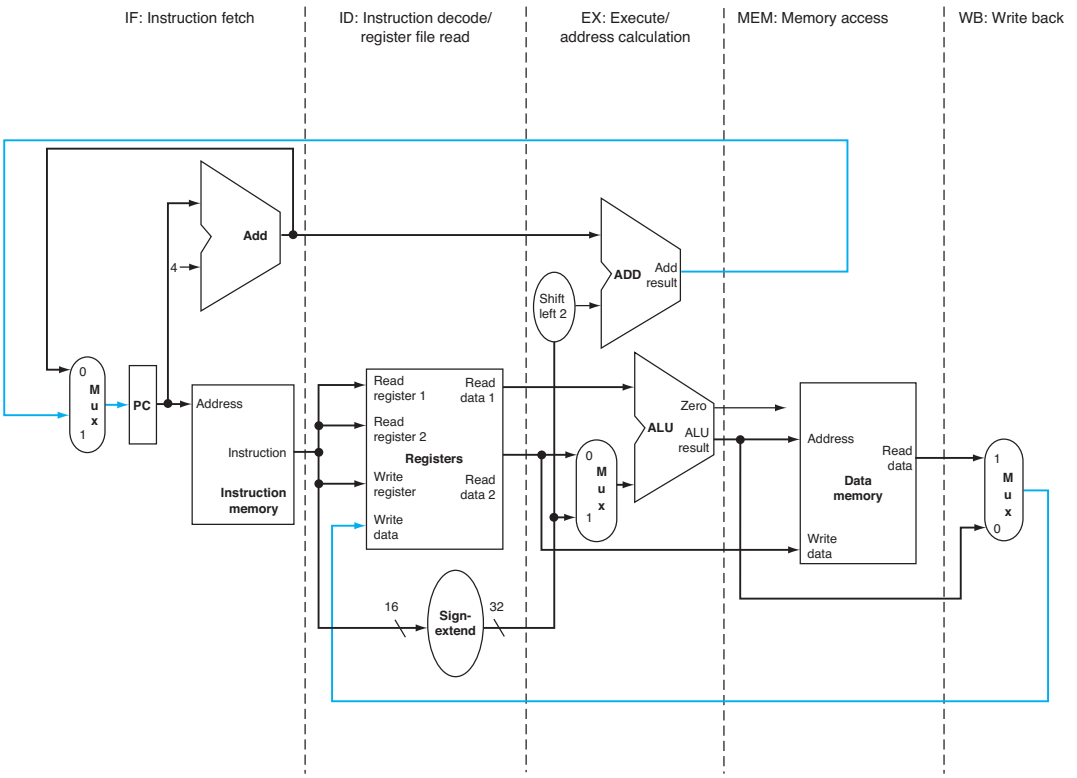

图 2: 数据通路

对于每个模块，把输入、输出变量全部加上阶段前缀：输入从流水线寄存器中获取，输出存放到下一阶段的流水线寄存器中。比如 IF 阶段：

```
    // IF
    wire [31:0] IF_newpc;
    wire [31:0] IF_pc;
    wire [31:0] IF_inst;

    wire pcSrc;
    wire [31:0] MEM_branchpc;
    assign IF_newpc = pcSrc ? MEM_branchpc : IF_pc + 4;

    InstMemory instMemory (
        .address(IF_pc),
        .inst(IF_inst)
    );

    Register#(.MSB(31)) PC (
        .Clk(Clk),
        .reset(reset),
        .in(IF_newpc),
        .out(IF_pc)
    );

////////////////////////////////////////////////
    parameter IFID_MSB = 63;
    wire [IFID_MSB:0] IFID;
    // 31:0 inst 32
    // 63:32 pc+4 32
    Register#(.MSB(IFID_MSB)) IFID_reg (
        .Clk(Clk),
```

```
        .reset(reset),
        .in({IF_pc + 4, IF_inst}),
        .out(IFID)
    );
```

不过在此时，我并不打算处理控制冒险——或者说，一切和跳转相关的指令，所以 newpc 没有什么用。如何决定是否把某个数据向后传递呢？答案还是根据输入，如果后续模块需要这个输入，就要放入流水线寄存器中。

3.2.2 控制信号

解决了数据通路后，就要考虑控制信号了。Ctr 模块在 ID 阶段计算出控制信号后，要把控制信号向后传递。如果每个控制信号都放到原本的流水线寄存器里，位宽就太多太复杂了。所以我使用了类似图 3 的方法，把某个阶段需要的控制信号打包在一起。如果遇到了不属于自己阶段的“包”，只要直接向后传递就可以了。

之前在 Lab05 中，许多一位信号被我改成了多位信号，在这里我把它们拆成了很多个一位信号，便于使用。比如 `regDst[1]`，被我改成 `regJal`，表示当前指令是 `jal`。

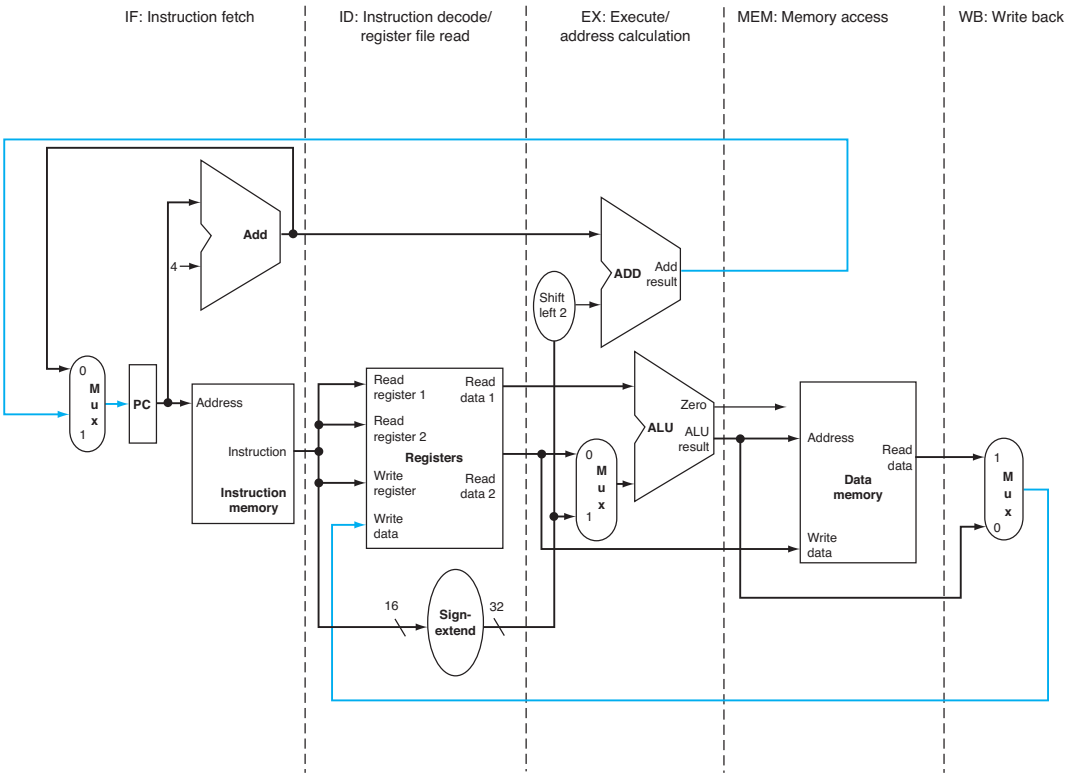

图 3: 控制信号的传递

所以，所有控制信号全部在 ID 阶段分发，代码如下：

```
parameter EX_MSB = 5;
parameter M_MSB = 2;
parameter WB_MSB = 2;
wire [EX_MSB:0] IDEX_EX;
// [2:0] aluOp 3
// [3] aluSrc 1
```

```
    // [4] regJal 1
    // [5] regDst 1

    wire [M_MSB:0] IDEX_M;
    // [0] branch 1
    // [1] memWrite 1
    // [2] memRead 1

    wire [WB_MSB:0] IDEX_WB;
    // [0] regWrite 1
    // [1] memToReg 1
    // [2] regJal 1

    Register#(.MSB(EX_MSB)) IDEX_EX_reg (
        .Clk(Clk),
        .reset(reset),
        .in({ID_regDst, ID_regJal, ID_aluSrc, ID_aluOp}),
        .out(IDEX_EX)
    );
    Register#(.MSB(M_MSB)) IDEX_M_reg (
        .Clk(Clk),
        .reset(reset),
        .in({ID_memRead, ID_memWrite, ID_branch}),
        .out(IDEX_M)
    );
    Register#(.MSB(WB_MSB)) IDEX_WB_reg (
        .Clk(Clk),
        .reset(reset),
        .in({ID_regJal, ID_memToReg, ID_regWrite}),
        .out(IDEX_WB)
    );
```

到了 EX 阶段，要把 MEM、WB 的控制信号向后传递，代码如下：

```
    wire [M_MSB:0] EXMEM_M;
    wire [WB_MSB:0] EXMEM_WB;

    Register#(.MSB(M_MSB)) EXMEM_M_reg (
        .Clk(Clk),
        .reset(reset),
        .in(IDEX_M),
        .out(EXMEM_M)
    );
    Register#(.MSB(WB_MSB)) EXMEM_WB_reg (
        .Clk(Clk),
        .reset(reset),
        .in(IDEX_WB),
        .out(EXMEM_WB)
    );
```

编写好数据通路与控制信号后，就可以测试没有任何冒险的程序了（仿真测试 4.1）。冒险的产生，根本原因是流水线并非一直向后运行，还存在着更新 PC、写回寄存器这两种数据向前流通的情况。其中更新 PC 导致控制冒险，写回寄存器导致数据冒险。下面分别介绍如何解决这两种冒险。

3.2.3 Forwarding

以写后读（RAW）数据冒险为例，如果不做任何处理，一条写指令会影响其后三条读指令，如图 4 所示。如果寄存器足够快，可以在一半周期内完成写操作，那么第三条指令不会受影响，实际只有两条受影响的指令。

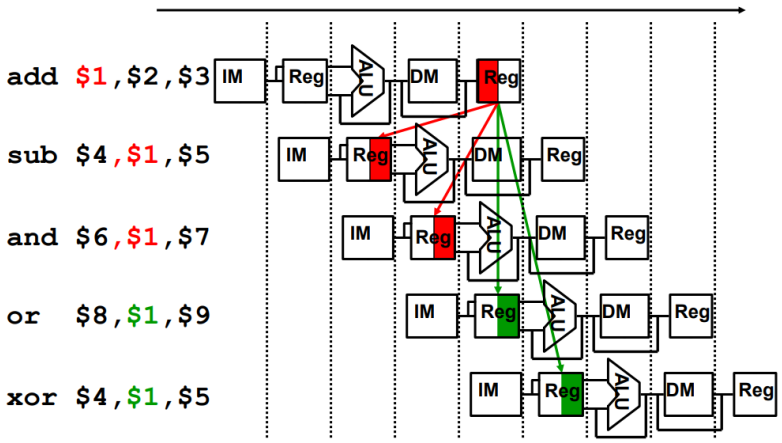

图 4: RAW 数据冒险

添加 Forwarding（转发、旁路、前向）可以解决这两条指令的数据冒险。从图 4 中可以看出，如果一条指令遇到了冒险，需要把数据从上一条、上两条转发到本条指令的 ALU 中。因此我新加入一个模块 Forward，按照书上内容计算出 `forwardA,forwardB`，分别代表 ALU 两个输入的数据来源。代码如下：

```
always @(*) begin
    forwardA = 2'b00;
    forwardB = 2'b00;

    // EX/MEM
    if (EXMEM_regWrite && EXMEM_rd != 0) begin
        if (EXMEM_rd == IDEX_rs)
            forwardA = 2'b10;
        if (EXMEM_rd == IDEX_rt)
            forwardB = 2'b10;
    end

    // MEM/WB
    if (MEMWB_regWrite && MEMWB_rd != 0 &&
      !(EXMEM_regWrite && EXMEM_rd != 0 && EXMEM_rd == IDEX_rs))
        if (MEMWB_rd == IDEX_rs)
            forwardA = 2'b01;

    if (MEMWB_regWrite && MEMWB_rd != 0 &&
      !(EXMEM_regWrite && EXMEM_rd != 0 && EXMEM_rd == IDEX_rt))
        if (MEMWB_rd == IDEX_rt)
            forwardB = 2'b01;
end
```

如果两种需要转发的情形同时发生（WAW），要注意选用更近、数据更新的那一条，所以上述代码在 MEM/WB 处有判断。

可以注意到，上述代码是从流水线寄存器中读取的数据，而不是直接从寄存器组的输出端或是 ALU 的输出端读取的，所以 Forward 单元放在了 EX 阶段（见图 1），这是《计算机组成与设计：硬件/软件接口》中的做法。在《深入理解计算机系统》中则不然，转发的来源就是输出端，Forward 单元放在了 ID 阶段。两种做法都可以达到要求，但是加上控制冒险后，第二种做法可以早一个阶段拿到转发的数据，在 ID 阶段即可判断是否需要跳转，把预测错误的惩罚降低到了一个周期，有优越性。

为什么第一本书要采用第一种做法呢？我猜测是阶段划分的原因。当我把转发功改为

第二种做法时，几乎所有的 Mux 都被提前了一个阶段，WB 阶段更是除了写入寄存器以外无事可做，或许这样会导致阶段划分的不平衡，影响处理器的最高频率。不过由于我们的仿真远远达不到最高频率，这个问题就不用考虑了，使用第二种做法可以带来解决控制冒险的优势。

在《深入理解计算机系统》中，还有一段话论述这么做是可行的：“注意，使用 ALU 的输出不会造成任何时序问题。译码阶段只要在时钟周期结束之前产生信号 valA 和 valB, 这样在时钟上升开始下一个周期时，流水线寄存器 E 就能装载来自译码阶段的值了。而在此之前 ALU 的输出已经是合法的了。”

所以，我决定就将 Forward 单元放置在了 ID 阶段。原本的许多判断都被提前了，比如 ALU 输入有两个选择，原本是 ID 阶段把两个选择全部放入流水线寄存器，再在 EX 阶段选择，现在变成了直接在寄存器组的出口选好，再放入寄存器。这种修改让很多传递到后面的控制信号变得无必要，但是我没有一一改过来，代码还有很大的优化空间。

代码几乎不用修改，区别仅在于给模块的输入不同。最后 Forward 单元的实例化有点不伦不类，不过可以实现功能：

```
Forward forward (
    .EXMEM_regWrite(IDEX_WB[0]),
    .MEMWB_regWrite(EXMEM_WB[0]),
    .EXMEM_rd(EX_writeReg),
    .MEMWB_rd(EXMEM[101:97]),
    .IDEX_rt(ID_inst[20:16]),
    .IDEX_rs(ID_inst[25:21]),
    .forwardA(forwardA),
    .forwardB(forwardB)
);
```

这时可以测试没有 Load-Use 冒险的程序（仿真测试 4.2）。

3.2.4 Stall

想要完全解决数据冒险，还需要 Stall 机制，因为 Load-Use 型冒险是无法通过转发解决的。Stall 的条件很简单，检测到上一条指令是 Load 指令，且当前指令的源操作数是上一条指令的目的操作数，就停顿一周期。和 Forward 一样，Stall 可以提前，代价是许多判断也会被提前：把 Stall 提前到 IF 阶段，相当于要在 IF 阶段就把指令部分解码了。如此，该指令在 IF，上一条 Load 指令在 ID，所以停顿操作就是：PC 不更新，IFID 写入全 0。

为了实现停顿操作，我为寄存器模块增加了一个输入 `write`，只有 `write == 1` 时才会写入。这样一来，正常寄存器的 `write` 始终为 1，而 PC 的 `write` 是`!stall`。写入全 0 很容易实现，在流水线寄存器的输入中使用三元运算符即可。

实现 Stall 后，所有含数据冒险的程序都可以运行了（仿真测试 4.3）。

3.2.5 控制冒险

我采用 predict-not-taken（预测不发生）的策略解决控制冒险。如果预测错误，应该做什么呢？首先分析每种跳转指令预测错误的代价。

- `beq`，`bne`：如果在 ID 阶段能判断是否跳转，代价是一个周期
- `jal`：IF 阶段即可判断，无代价
- `j`：IF 阶段即可判断，无代价
- `jr`：ID 阶段才能找到跳转地址，代价是一个周期

所以，如果在 ID 阶段前完成判断，最多只有一个周期的代价。如果采用 Forward 小节的第一种方法，这里的判断将非常麻烦；但如果使用第二种，判断就会很简单，因为转发的数据就是即将写入流水线寄存器的数据。J 型指令在 IF 阶段完成判断也需要部分解码。

```
// IF 阶段
assign IF_jump = IF_inst[31:26] == 6'b000010 || IF_inst[31:26] == 6'b000011;
assign IF_newpc = IF_jump ? {IF_pcplus4[31:28], IF_inst[25:0], 2'b0}
                : (pcSrc ? ID_branchpc : IF_pc + 4);
// ID 阶段
assign ID_branch = (ID_beq && regData1 == regData2); // 是否有 branch？后续可以加上 bne
assign pcSrc = ID_jr || ID_branch; // pc 是否要改变？
assign ID_branchpc = ID_jr ? regData1 :
    (ID_branch ? (ID_imm << 2) + ID_pcplus4 : 0); // 最后的 0 没用

// Forward 传来的数据，判断跳转地址时也需要用到
assign forwardALU = IDEX_regJal ? EX_pcplus4 : EX_aluRes; // jal == 1
assign regData1 = forwardA == 2'b10 ? forwardALU
    : (forwardA == 2'b01 ? MEM_writeData : ID_readData1);
assign regData2 = forwardB == 2'b10 ? forwardALU
    : (forwardB == 2'b01 ? MEM_writeData : ID_readData2);
```

如果预测错误，流水线需要插入 nop，但照常从 PC 中取指，因此 PC 不受影响；错误发生时，只需要清空 IFID 寄存器即可，将清空条件改为 `pcSrc || stall`。

此后，所有含控制冒险的程序都可以运行了（仿真测试 4.4）。

3.2.6 31 条指令

将指令增加到 31 条难点不多，只要根据指令的特性增加控制信号，或是增加 ALU 的功能就可以了。

我遇到的一个问题是，如果 `aluOp` 只有 3 位，将无法容纳 31 条指令的 9 种操作，而我又不想继续增加其位宽。所以，对于 `lui` 指令，我额外用了一个控制信号，将它传入 ALUCtr 中。最后 ALUCtr 的代码如下：

```
always @(aluOp or funct or lui) begin
    casex ({aluOp, funct})
    9'b000xxxxxx: begin aluCtrOut = 4'b0010; aluSrc1 = 0; end // addi
    9'b001xxxxxx: begin aluCtrOut = 4'b0110; aluSrc1 = 0; end // subi
    9'b100xxxxxx: begin aluCtrOut = 4'b0000; aluSrc1 = 0; end // andi
    9'b101xxxxxx: begin aluCtrOut = 4'b0001; aluSrc1 = 0; end // ori
    9'b110xxxxxx: begin aluCtrOut = 4'b1101; aluSrc1 = 0; end // xori
    9'b011xxxxxx: begin aluCtrOut = 4'b0111; aluSrc1 = 0; end // slti
    9'b111xxxxxx: begin aluCtrOut = 4'b1000; aluSrc1 = 0; end // sltiu
    // 以上是 7 种 I 型指令，再加上下面的 R 型，用完了 aluOp 的 3 位
    9'b010100001: begin aluCtrOut = 4'b0010; aluSrc1 = 0; end // addu
    // ... 其他 R 型指令
    9'b010000011: begin aluCtrOut = 4'b0101; aluSrc1 = 1; end // sra
    default: begin aluCtrOut = 4'b0; aluSrc1 = 0; end
    endcase
    if (lui) begin // lui 单独判断，因为 aluOp 位数不够了
        aluCtrOut = 4'b1111;
        aluSrc1 = 0;
    end
end
```

完成后，可以运行所有的指令了（仿真测试 4.5）。

4 仿真测试

4.1 无冒险

内存初始值 3 4 6 8，代码见附录 A.1，结果见图 5。

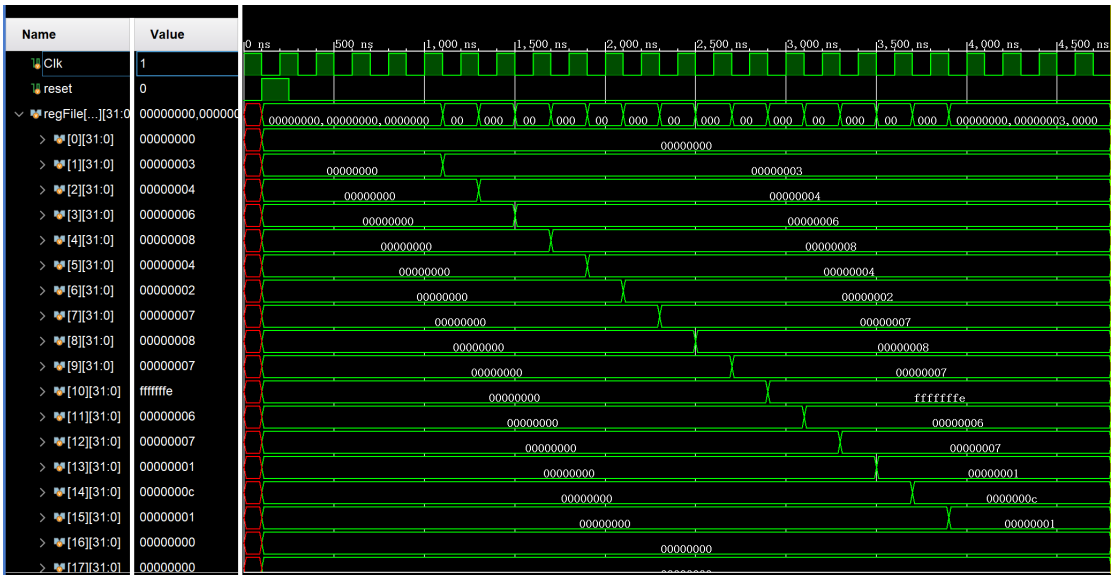

图 5: 波形图 A

4.2 无 Load-Use 冒险

代码见附录 A.2，结果见图 6。

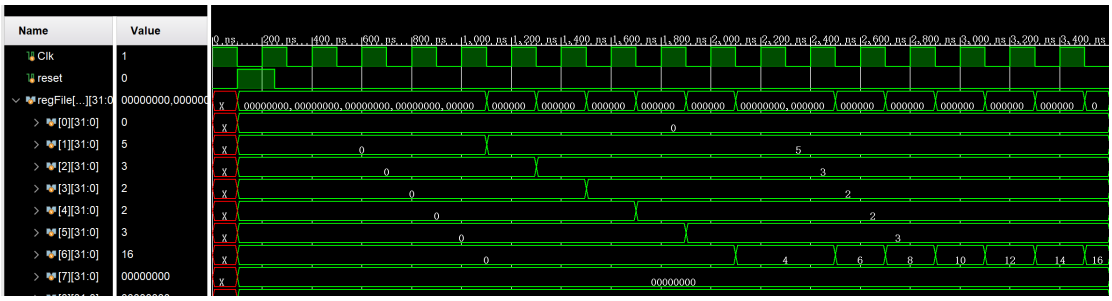

图 6: 波形图 B

4.3 Load-Use 冒险

内存初始值 3 4 6 8，代码见附录 A.3，结果见图 7。

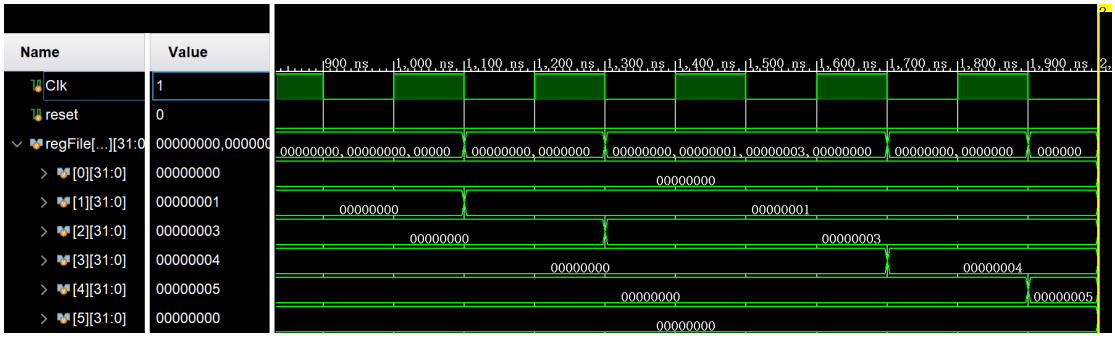

图 7: 波形图 C

4.4 控制冒险

代码见附录 A.4，结果见图 8。

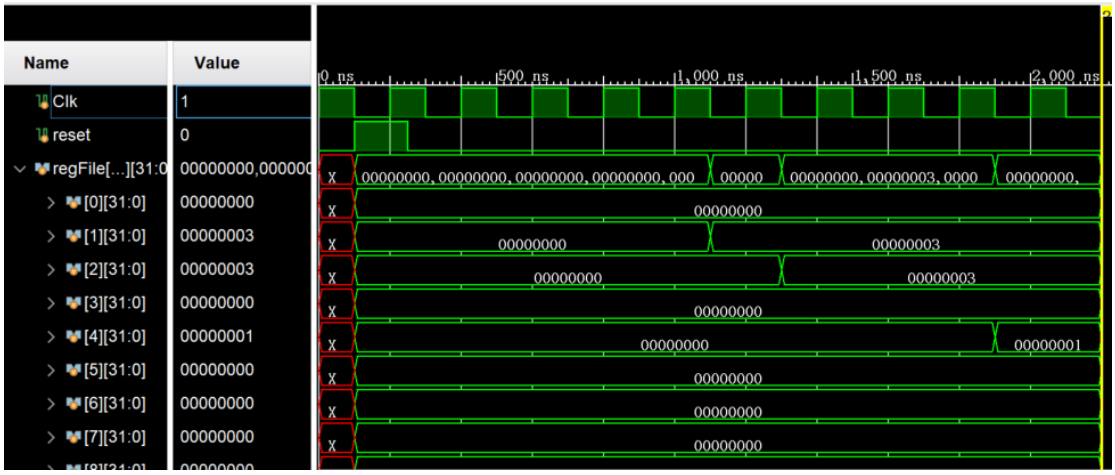

图 8: 波形图 D

4.5 31 条指令

内存初始值 0 1 2 3 4，代码见附录 A.5。本测试代码较长，我使用 MIPS 模拟器逐行验证了结果的正确性，最终结果（仅一部分）见图 9。

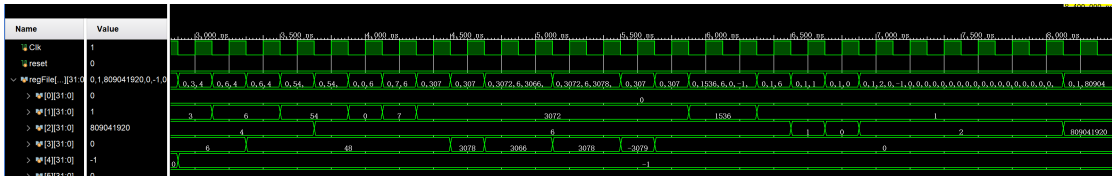

图 9: 波形图 E

5 总结与反思

本次实验我仍由简单到复杂，实现了一个支持 31 条指令的流水线处理器。在实现过程中我也遇到了困难，编写 Forward 模块时，我原本从一本书上学到的写法让解决控制冒险变得非常复杂，我只好到处寻找资料，在另一本书上找到了较好的做法；有趣的是，他们的做法对于本实验都是不完整的，需要组合起来才能得到最好的效果。

在实验开始时我选择把流水线寄存器作为通用的模块，一切数据都要手动指定位宽。这种设计在后面看来是不好的，极难发现某一个位宽写错了，导致其他数字无法对齐。如果重新写的话，我应该也会把每个寄存器单独模块化，省去处理几乎没有规律的数字。

在实验中，我每实现一部分功能，就编写对应的指令来随时测试，保证每一部分都是正确的；使用 git，保存每一阶段正确的代码。这为我省下了大量调试的时间：如果出现问题，那基本上是新加的代码有问题。测试用的汇编代码是我手动编写的，而对应的机器码是由 MARS 生成的，我再用 MARS 的结果对照自己的结果，就能知道实现是否正确。代码与相应的指令内存附在文后。

本次实验我没有上板验证，我觉得如果上板，效果应和上次实验无异，所以没有必要。

感谢助教和老师的悉心指导！

A 附录

A.1 无冒险

汇编代码：

```
lw $1, ($0)
lw $2, 4($0)
lw $3, 8($0)
lw $4, 12($0)

addi $5, $1, 1
addi $6, $2, -2
ori $7, $3, 3
andi $8, $4, 15

add $9, $1, $2
sub $10, $3, $4
and $11, $3, $7
or $12, $1, $3
slt $13, $6, $9

sll $14, $1, 2
srl $15, $1, 1

sw $8, 16($0)
sw $9, 20($0)
sw $10, 24($0)
sw $11, 28($0)
```

指令内存：

```
10001100000000010000000000000000
10001100000000010000000000000100
10001100000000011000000000001000
10001100000000010000000000001100
00100000001001010000000000000001
00100000010001101111111111111110
```

```
00110100011001110000000000000011
00110000100010000000000000001111
00000000001000100100100000100000
00000000011001000101000000100010
00000000011001110101100000100100
00000000001000110110000000100101
00000000110010010110100000101010
00000000000000001011100001000000
00000000000000001011110000100010
10101100000010000000000000010000
10101100000010010000000000010100
10101100000010100000000000011000
10101100000010110000000000011100
```

A.2 无 Load-Use 冒险

汇编代码：

```
addi $1, $0, 5
addi $2, $0, 3
sub $3, $1, $2
and $4, $2, $3
or $5, $2, $3
add $6, $3, $3
sw $1, 2($3)
add $6, $3, $6
add $6, $3, $6
add $6, $3, $6
add $6, $3, $6
add $6, $3, $6
add $6, $3, $6
```

指令内存：

```
00100000000000010000000000000101
00100000000000100000000000000011
00000000001000100001100000100010
00000000010000110010000000100100
00000000010000110010100000100101
00000000011000110011000000100000
10101100011000010000000000000010
00000000011001100011000000100000
00000000011001100011000000100000
00000000011001100011000000100000
00000000011001100011000000100000
00000000011001100011000000100000
00000000011001100011000000100000
```

A.3 Load-Use 冒险

汇编代码：

```
addi $1, $0, 1
lw $2, ($0)
addi $3, $2, 1
addi $4, $3, 1
```

指令内存：

```
00100000000000010000000000000001
10001100000000010000000000000000
00100000010000110000000000000001
00100000011001000000000000000001
```

A.4 控制冒险

汇编代码：

```
addi $1, $0, 3
addi $2, $0, 3
beq $1, $2, label
addi $3, $0, 1
label:
addi $4, $0, 1
```

指令内存：

```
00100000000000010000000000000011
00100000000000010000000000000011
00010000001000100000000000000001
00100000000000110000000000000001
00100000000001000000000000000001
```

A.5 31 条指令

汇编代码：

```
start:
    jal label
    lui $2, 12345
    j end
label:
    lw $1, 8($0)
    lw $1, 12($0)
    lw $2, 16($0)
    add $3, $1, $2
    and $3, $1, $2
    addi $3, $0, 6
    sub $4, $1, $2
    sll $1, $1, 1
    sll $3, $3, 3
    or $1, $3, $1
    addiu $2, $0, 6
    slti $1, $1, 1
    ori $1, $2, 1
    sllv $1, $3, $2
    addu $3, $1, $2
    sub $3, $1, $2
    subu $3, $1, $2
    or $3, $1, $2
    xor $3, $1, $2
    nor $3, $1, $2
    slt $3, $1, $2
    sra $1, $1, 1
    sw $2, 4($0)
    sltu $1, $0, $2
    srl $2, $2, 2
    srav $2, $2, $2
```

```
    xori $2, $2, 2
    sltiu $1, $2, 4
    beq $5, $0, label1
    bne $5, $0, label2
label1:
    jr $ra
label2:
    xor $1, $1, $1
    addi $1, $1, 666
end:
```

指令内存：

```
00001100000000000000000000000011
00111100000000100011000000111001
00001000000000000000000000100100
10001100000000010000000000001000
10001100000000010000000000001100
10001100000000010000000000010000
00000000001000100001100000100000
00000000001000100001100000100100
00100000000000110000000000000110
00000000001000100010000000100010
00000000000000001000010000001000000
00000000000000011000110001100000
00000000011000010000100000100101
00100100000000100000000000000110
00101000001000010000000000000001
00110100010000010000000000000001
00000000010000110000100000000100
00000000001000100001100000100001
00000000001000100001100000100010
00000000001000100001100000100011
00000000001000100001100000100101
00000000001000100001100000100110
00000000001000100001100000100111
00000000001000100001100000101010
00000000000000001000010000011000011
10101100000000100000000000000100
00000000000000010000010000101011
00000000000000010000100001000010
00000000010000100001000000000111
00111000010000100000000000000010
00101100010000010000000000000100
00010000101000000000000000000001
00010100101000000000000000000001
00000011111000000000000000001000
00000000001000010000100000100110
00100000001000010000001010011010
```