COD_Lab5 流水线 CPU 设计

PB18111707 吕瑞

报告中的测试指令 test1 和 test2 均来自于 2020 年春季计算机组成原理实验课的助教。

一、实验目标

1. 理解流水线CPU的组成结构和工作原理；
2. 掌握数字系统的设计和调试方法；
3. 熟练掌握数据通路和控制器的设计和描述方法。

二、实验内容

（0）流水线 CPU 概述：

流水线 CPU，即采用流水线的方式执行指令。通常 MIPS 指令包括以下 5 个处理步骤：

1. 从指令存储器中读取指令。
2. 指令译码的同时读取寄存器。MIPS 的指令格式允许同时进行指令译码和读寄存器。
3. 执行操作或计算地址。
4. 从数据存储器中读取操作数。
5. 将结果过写回寄存器。

流水线的奇妙之处在于，对于单独的指令，取指、译码、执行、访存、写回，这整个过程的处理时间并没有缩短，而对于多条指令需要执行时，他们可以并行进行。因此，流水线上单位时间内能够完成执行的指令条数就会大大增加。改善了 cpu 的吞吐率。在理想状态下（没有流水线阻塞），其得到的速度提高倍数与指令处理所需步骤相同，为 5 倍。

即有公式：

$$指令执行时间（流水线）= 指令执行时间（非流水线）/流水线级数$$

图 0. 来自 COD 课本 204 页：

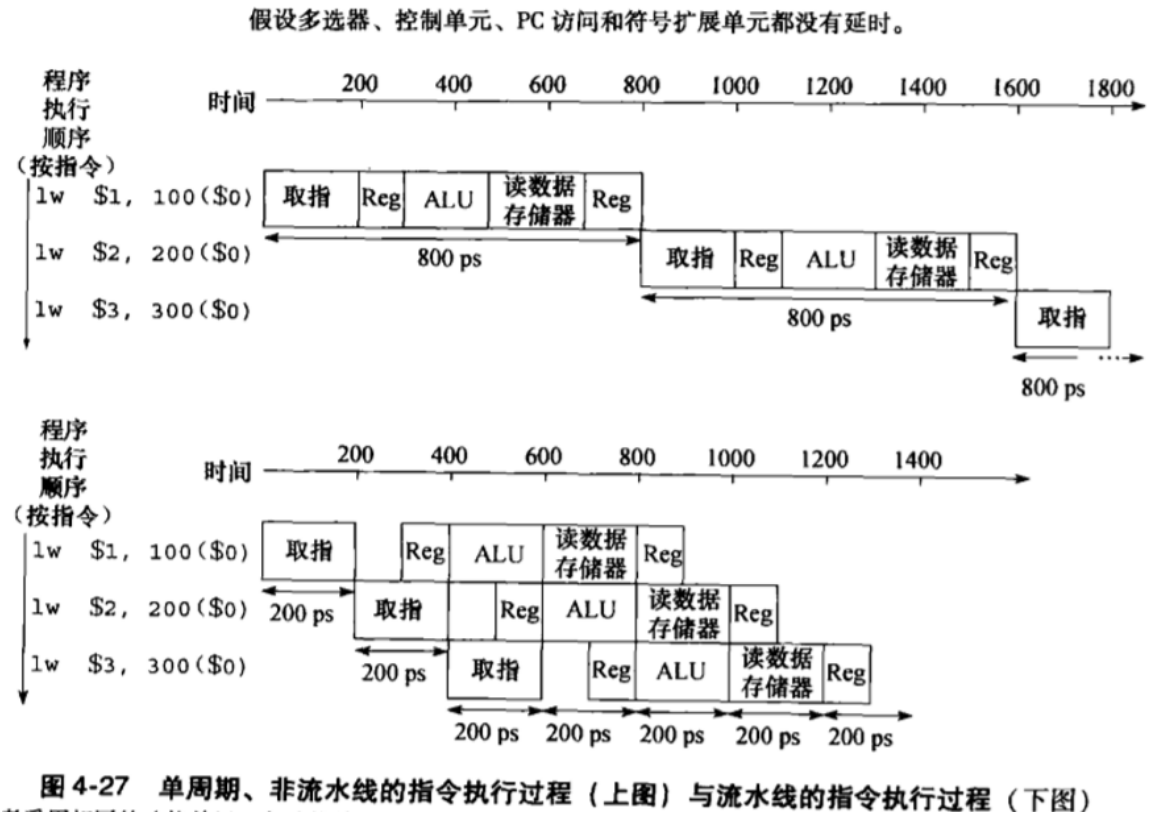


图 4-27 单周期、非流水线的指令执行过程（上图）与流水线的指令执行过程（下图）

流水线冒险：

流水线冒险（hazard）：在下一个时钟周期中，下一条指令不能执行。

1. 结构冒险

硬件不支持多条指令在同一时钟周期进行。

发生情景：在同一寄存器内，需要在同一个时钟周期中读写数据。

解决方法：让寄存器提前半个时钟周期写入数据，在另半个时钟周期内读出数据。

> 比如在时钟下降沿触发数据写入。数据异步读出。

2. 数据冒险

一条指令必须等待另一条指令的完成。

发生情景：当前指令的操作数，来自上条指令需要写回的结果。

解决方法：转发（旁路），即设计额外的数据通路，将已经产生的结果作用于需要它们的流水级。

> 流水线转发并不能解决数据冒险的所有情况，必要时还是需要做流水线阻塞来保证指令执行的正确性。
>
> 具体见下文 HazardUnit 单元的实现部分。

3. 控制冒险

决策依赖于一条指令的结果，而其正在进行中。

发生情景：分支指令和跳转指令。

解决方法：流水线阻塞（停顿），或做分支预测。

> 对于分支预测，分为静态分支预测和动态分支预测。两种方法本质上都是提前预估 beq 指令的执行状态，对下一条指令的地址做出判断。避免流水线的停顿。

(1) 指令 -- 6 条

1. 设计实现多周期 CPU ， 可执行如下六条指令：

• **add**: rd <- rs + rt; op = 000000, funct = 100000

R类型：

31-26	25-21	20-16	15-11	10-6	5-0
op	rs	rt	rd	shamt	func
6位	5位	5位	5位	5位	6位

• **addi**: rt <- rs + imm; op = 001000

lw: rt <- M(rs + addr); op = 100011

sw: M(rs + addr) <- rt; op = 101011

beq: if (rs = rt) then pc <- pc + 4 + addr << 2

else pc <- pc + 4; op = 000100

I类型：

31-26	25-21	20-16	15-0
op	rs	rt	immediate/addr
6位	5位	5位	16位

• **j**: pc <- (pc+4)[31:28] | (add<<2)[27:0]; op = 000010

J类型：

31-26	25-0
op	address
6位	26位

2. 分析指令功能，设计数据通路和控制器。
3. 指令和数据存储分别用两个 256x32 的分布式单端口 RAM 存储器，采用 IP 例化实现。

(2) 数据通路

图 1. 教材给出：

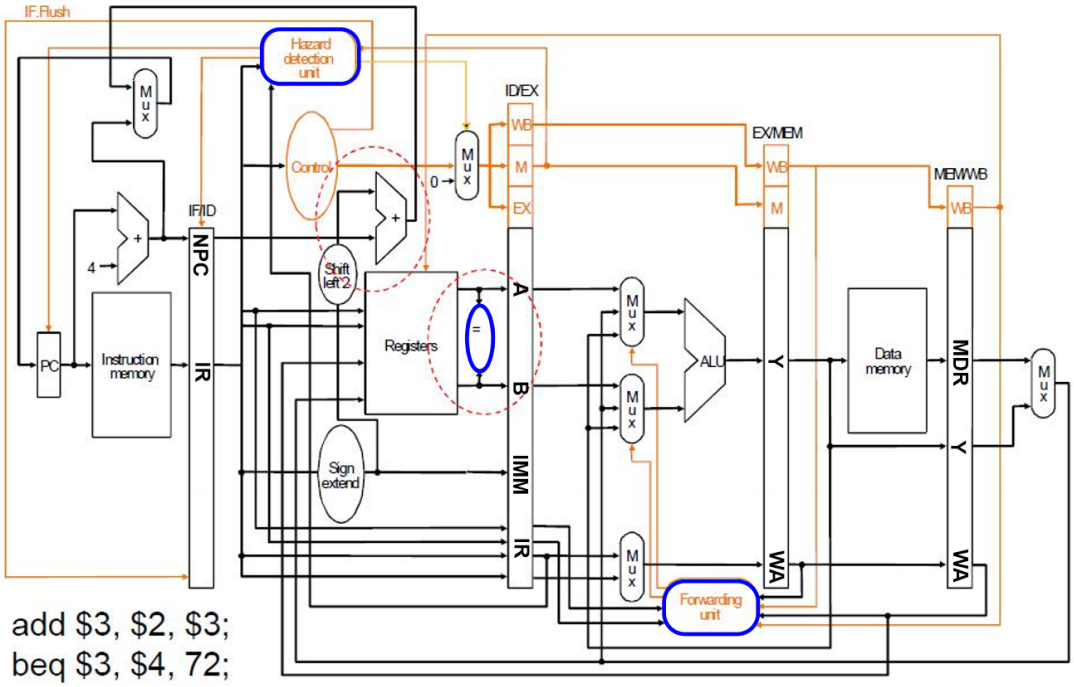
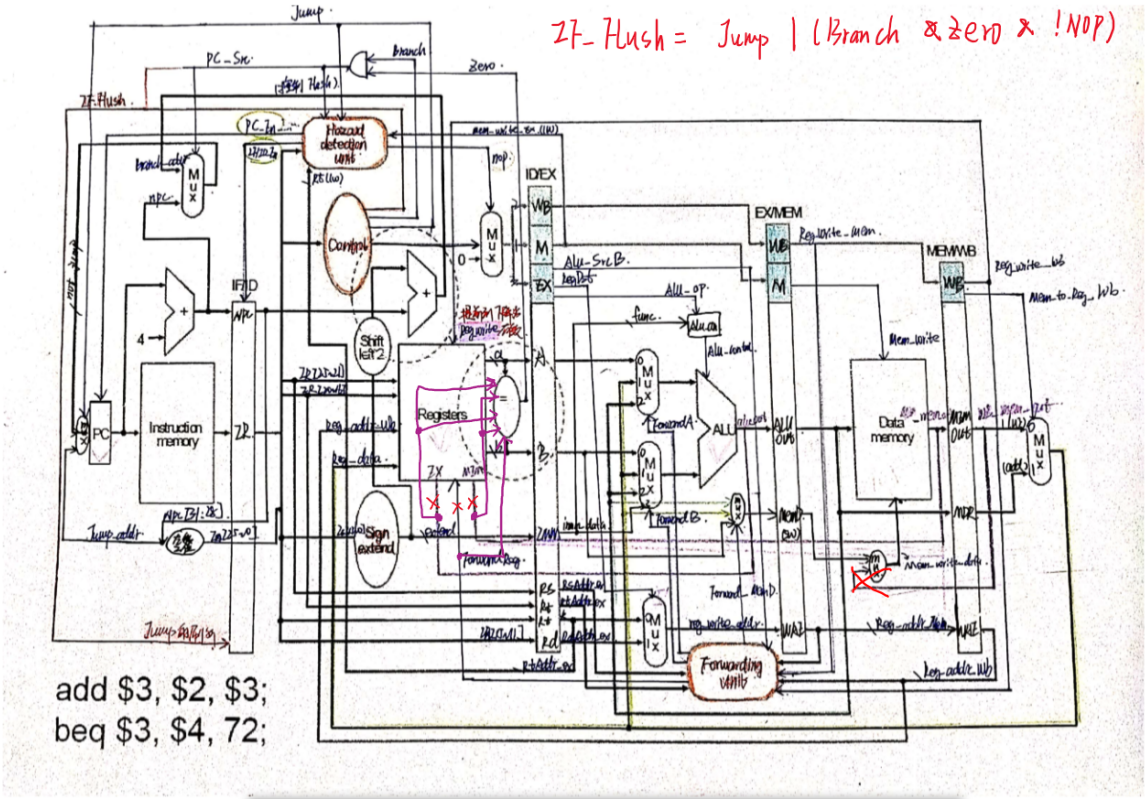


图 2. 根据指令特点改造后的数据通路：



（3）各大模块设计

控制单元 -- ControlUnit

表-1 总控制信号

指令	RegDst	Jump	Branch	MemRead	MemtoReg	MemWrite	ALUSrcB	RegWrite	ALUOp
add	1	0	0	0	0	0	0	1	10
addi	0	0	0	0	0	0	1	1	11
lw	0	0	0	1	1	0	1	1	00
sw	x	0	0	0	x	1	1	0	00
beq	x	0	1	0	x	0	0	0	01
j	x	1	x	0	x	0	x	0	x

基本运算单元 -- ALU

表-2 ALU 模块功能表

m	y
000	a + b
001	a - b
010	a & b
011	a | b
100	a ^ b
其他	x

表-3 ALU 控制单元

ALUOP	funct（6bits）/funct_imm(3bits)	ALUControl	ind
10	100000	000	add
10	-	001	sub
10	-	010	and
10	-	011	or
10	-	100	xor
00	-	000	lw,sw
01	-	001	beq
11	-	000	addi

段间寄存器 -- IF_ID_Reg

```
module IF_ID_Reg #(parameter N = 32)
   (input [N-1:0] PC4,
    input [N-1:0] Instruct,
    input en,
    input clk,
    input rst,
    input Flush,
    output reg [N-1:0] NPC,
    output reg [N-1:0] IR
    );
```

段间寄存器 -- ID_EX_Reg

```
module ID_EX_Reg #(parameter N = 32)
   (input clk,
    input [N-1:0] RegData_A_id,
    input [N-1:0] RegData_B_id,
    input [N-1:0] ImmData_id,
    input [4:0] RsAddr_id,
    input [4:0] RtAddr_id,
    input [4:0] RdAddr_id,
    input RegWrite_id,
    input MemToReg_id,
    input MemWrite_id,
    input MemRead_id,
    input AluSrcB_id,
    input RegDst_id,
    input [1:0] AluOp_id,

    output reg [N-1:0] RegData_A_ex,
    output reg [N-1:0] RegData_B_ex,
    output reg [N-1:0] ImmData_ex,
    output reg [4:0] RsAddr_ex,
    output reg [4:0] RtAddr_ex,
    output reg [4:0] RdAddr_ex,
    output reg RegWrite_ex,
    output reg MemToReg_ex,
    output reg MemWrite_ex,
    output reg MemRead_ex,
    output reg AluSrcB_ex,
    output reg RegDst_ex,
    output reg [1:0] AluOp_ex
    );
```

段间寄存器 -- EX_MEM_Reg

```
module EX_MEM_Reg #(parameter N = 32)
   (input clk,
    input [N-1:0] AluOut_ex,
    input [N-1:0] MemWriteData_ex,
    input [4:0] RegWriteAddr_ex,
    input RegWrite_ex,
    input MemToReg_ex,
    input MemWrite_ex,
    input MemRead_ex,
```

```
    // Êä³ö¶Ë¿Ú
    output reg [N-1:0] AluOut_mem,
    output reg [N-1:0] MemWriteData_mem,
    output reg [4:0] RegWriteAddr_mem,
    output reg RegWrite_mem,
    output reg MemToReg_mem,
    output reg MemWrite_mem,
    output reg MemRead_mem
    );
```

段间寄存器 -- MEM_WB_Reg

```
module MEM_WB_Reg #(parameter N = 32)
   (input clk,
    input [N-1:0] MemOut_mem,
    input [N-1:0] RTypeWriteBackData_mem,
    input [4:0] RegWriteAddr_mem,
    input RegWrite_mem,
    input MemToReg_mem,
    // Êä³ö¶Ë¿Ú
    output reg [N-1:0] MemOut_wb,
    output reg [N-1:0] RTypeWriteBackData_wb,
    output reg [4:0] RegWriteAddr_wb,
    output reg RegWrite_wb,
    output reg MemToReg_wb
    );
```

有符号数的扩展单元 -- SignExtend

```
    assign extend[15:0] = imm; // 低 16 位保留
    assign extend[31:16] = (imm[15] == 1)? 16'hffff:16'h0000; // 有符号数，高 16 位补 imm[15]
```

前推转发控制单元 -- ForwardUnit

需要前推转发的数据冒险有三种：

- 分支指令的数据冒险：在数据通路的设计上，我们把分支预测的工作提前到了 ID 阶段，所以需要在 ID 阶段对 Rs / Rt 寄存器中的数据进行相等性检测。在这里需要考虑 EX 数据冒险和 MEM 数据冒险，即需要超前读取这两个阶段所得出的数据的冒险。

  > 注意首先需要判断是否是和需要写回的数据有读写冲突，下同。

```
    //----------分支指令的数据冒险----------//
    // Forward_RegWriteData_A_id
        if (RegWrite_ex && RegWriteAddr_ex && (RegWriteAddr_ex == RsAddr_id))
            Forward_RegWriteData_A_id = 2'b10; // ex 段冒险
        else if (RegWrite_mem && RegWriteAddr_mem && (RegWriteAddr_mem == RsAddr_id))
            Forward_RegWriteData_A_id = 2'b01; // mem 段冒险
        else
            Forward_RegWriteData_A_id = 2'b00;

    // Forward_RegWriteData_B_id
```

```
        if (RegWrite_ex && RegWriteAddr_ex && (RegWriteAddr_ex == RtAddr_id))
            Forward_RegWriteData_B_id = 2'b10; // ex 段冒险
        else if (RegWrite_mem && RegWriteAddr_mem && (RegWriteAddr_mem == RtAddr_id))
            Forward_RegWriteData_B_id = 2'b01; // mem 段冒险
        else
            Forward_RegWriteData_B_id = 2'b00;
```

- ALU 操作数的数据冒险：ALU 的操作数可能来自于 EX 段、MEM 段和 WB 段。

```
    //----------ALU 操作数的冒险----------//
    // Forward_A_ex
        if (RegWrite_mem && RegWriteAddr_mem && (RegWriteAddr_mem == RsAddr_ex))
            Forward_A_ex = 2'b10; // ex 段冒险
        else if (RegWrite_wb && RegWriteAddr_wb && (RegWriteAddr_wb == RsAddr_ex))
            Forward_A_ex = 2'b01; // mem 段冒险
        else
            Forward_A_ex = 2'b00;

        // Forward_B_ex
        if(AluSrcB_ex == 0)
            Forward_B_ex = 2'b11; // 选择立即数，没有数据冒险
        else if (AluSrcB_ex && RegWrite_mem && RegWriteAddr_mem && (RegWriteAddr_mem == RtAddr_ex))
            Forward_B_ex = 2'b10; // ex 段冒险
        else if (AluSrcB_ex && RegWrite_wb && RegWriteAddr_wb && (RegWriteAddr_wb == RtAddr_ex))
            Forward_B_ex = 2'b01; // mem 段冒险
        else
            Forward_B_ex = 2'b00;
```

- 写入 memory 的数据冒险：这里我们在 EX 段就可以对这类数据冒险进行处理，写入 memory 的数据可能来源于 EX 、MEM 和 WB 阶段。

```
    //----------sw 写入 memory 的数据冒险----------//
       if(RegWrite_mem && RegWriteAddr_mem && (RtAddr_ex == RegWriteAddr_mem))
           Forward_MemWriteData_ex = 2'b00; // ex 段冒险
       else if(RegWrite_wb && RegWriteAddr_wb && (RtAddr_ex == RegWriteAddr_wb))
           Forward_MemWriteData_ex = 2'b01; // mem 段冒险
       else
           Forward_MemWriteData_ex = 2'b10; // 没有冒险
```

流水线阻塞 -- HazardDetection

- lw 的目标寄存器号 == 下一条指令的读寄存器号的阻塞。

```
    assign NOP = ((RsAddr_id == RtAddr_ex) || (RtAddr_id == RtAddr_ex)) && MemRead_ex; // NOP == 1 插入流水线气泡
    assign PCEn = ~NOP; // 流水线阻塞1，指令不再更新
    assign IF_ID_En = ~NOP; // 流水线阻塞2，段间寄存器不再更新
```

图 3. 来自 COD 课本 233 页：

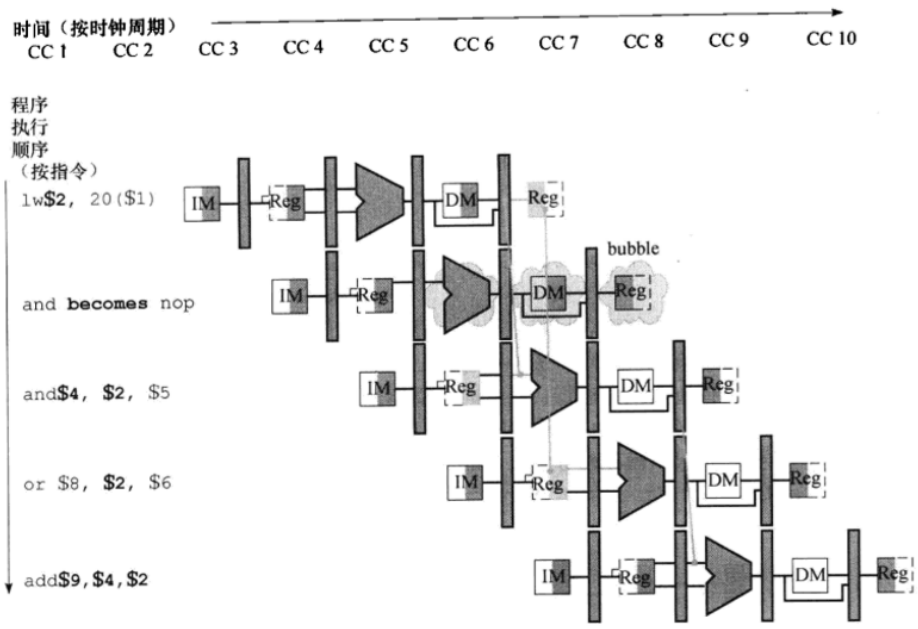


图 4-59　在流水线中插入阻塞的方法

在第 4 个时钟周期中，通过将 and 指令变成 nop 插入了一个气泡。注意，and 指令的 IF 和 ID 级在第 2 个和第 3 个时钟周期，但它的 EX 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。与此类似，OR 指令的 IF 级在第 3 个时钟周期，但它的 ID 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。在插入气泡后，所有的相关性沿时间前进，冒险不再发生。

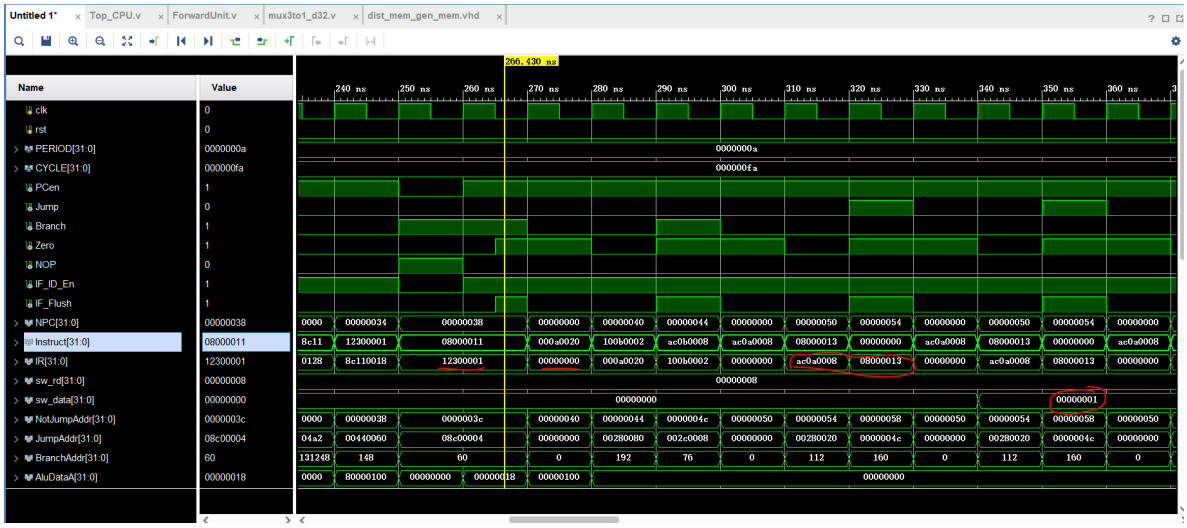
（4）仿真结果

test1

```
# 本文档存储器以字节编址
# 本文档存储器以字节编址
# 本文档存储器以字节编址
# 初始PC = 0x00000000

.data
    .word 0,6,0,16,0x80000000,0x80000100,0x100,5,0    #编译成机器码时，编译器会在前面多加个0，所以后面lw指令地址会多加4

_start:
        addi $t0,$0,3           #t0=3    0
        addi $t1,$t0,2          #t1=5    4
        addi $t2,$0,1           #t2=1    8
        addi $t3,$0,0           #t3=0    12

        add  $s0,$t1,$t0        #s0=t1+t0=8  测试add指令     16
        add  $s0,$s0,$s0        #s0=s0+s0=16                  20
        lw   $s1,12($0)         #                             24
        beq  $s1,$s0,_next1     #正确跳到_next                28

        j _fail
```

```
_next1:
        lw $t0, 16($0)          #t0 = 0x80000000    36
        lw $t1, 20($0)          #t1 = 0x80000100    40

        add  $s0,$t1,$t0        #s0 = 0x00000100 = 256  44
        lw $s1, 24($0)          #                       48
        beq  $s1,$s0,_next2     #正确跳到_success       52

        j _fail

_next2:
        add $0, $0, $t2         #$0应该一直为0          60
        beq $0,$t3,_success     #                       64


_fail:
        sw   $t3,8($0) #失败通过看存储器地址0x08里值，若为0则测试不通过，最初地址
0x08里值为0   64
        j    _fail
68

_success:
        sw   $t2,8($0)    #全部测试通过，存储器地址0x08里值为1
72
        j   _success
76

                        #判断测试通过的条件是最后存储器地址0x08里值为1，说明全部通过测
试
```

检查仿真中，最后存储器地址0x08里值为1，说明测试已经通过。

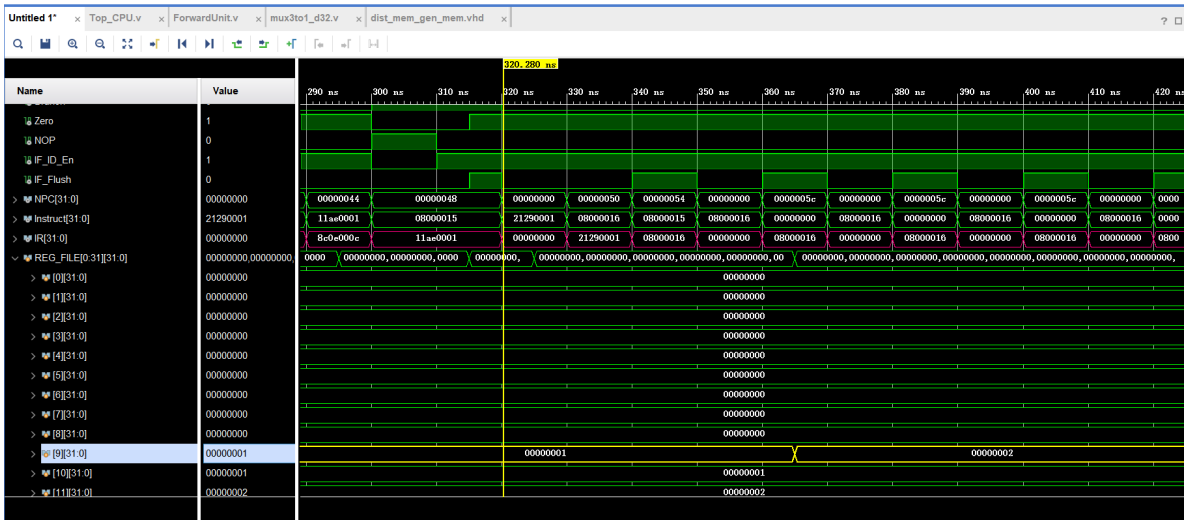


test2

```
# Test cases for MIPS 5-Stage pipeline

.data
    .word 0,1,2,3,0x80000000,0x80000100,0x100,5,0

_start:
    add $t1, $0, $0     # $t1 = 0
```

```
    j _test0

_test0:
    addi $t2, $0, 1      # $t2 = 1
    addi $t2, $t2, 1      # $t2 = 2
    add $t2, $t2, $t2      # $t2 = 4
    addi $t2, $t2, -4     # $t2 = 0
    beq $t2, $0, _next0   # if $t2 == $0: $t1++, go next testcase, else: go fail
    j _fail
_next0:
    addi $t1, $t1, 1      # $t1++
    j _test1

_test1:
    addi $0, $0, 4       # $0 += 4
    lw $t2, 4($0)        # $t2 = MEM[1]
    lw $t3, 8($0)        # $t3 = MEM[2]
    add $t4, $t2, $t3
    sw $t4, 0($0)        # MEM[0] = $t4
    lw $t5, 0($0)        # $t5 = MEM[0]
    lw $t6, 12($0)       # $t6 = MEM[3]
    beq $t5, $t6, _next1
    j _fail

_next1:
    addi $t1, $t1, 1
    j _success

_fail:
    j _fail

_success:
    j _success    # if success: $t1 == 2
```

检查仿真中，寄存器 $t1 的值，为 2，说明测试通过。



三、实验总结

本次实验实现的主要困难在于：数据通路的设计，流水线阻塞控制单元的判断条件设计，几种需要前推转发来避免流水线阻塞的情况的判断和实现，以及**静态分支预测和缩短分支延迟时间**的设计。

1. 数据通路：在教材已经给出的数据通路的基础上，根据需要实现的指令的定向通路，做一些修改即可。
2. 流水线阻塞控制单元：lw 的目标寄存器号 == 下一条指令的读寄存器号的阻塞时，流水线必须阻塞。
3. 前推转发设计单元：分为，分支预测前推、ALU 操作数前推和写入 memory 数据前推三种情况考虑。
4. 静态预测和缩短分支延迟时间的设计：

   静态分支预测，即默认每次的分支指令都不会发生跳转，指令寄存器继续读取下一条指令 pc+4，如果分支指令发生了，则需要在下一个时钟周期清空刚才读出的指令，换成分支跳转地址下的指令。所以这里要设计一个 flush 信号。

   `IF_Flush = Jump | (Branch & Zero & !NOP)`

   > jump 指令同样需要清空 pc+4，换成跳转地址下的指令。
   >
   > NOP 为流水线阻塞信号，阻塞信号是不需要清空寄存器的，只需要阻止指令寄存器在下一个时钟周期的写入即可。

   缩短分支预测延迟时间的设计：即，把原先在 EX 阶段才能发生的分支预测操作提前到 ID 阶段，从而达到缩短分支预测延迟时间的目的，能够将静态分支预测错误的代价尽可能的降低。这种方法同时需要在 ID 段设计前推转发的数据通路来解决数据冒险问题。

流水线的总体设计并不难，比较费时间的是调试工作，需要细心和耐心，一个不注意就会导致最终结果的错误。