流水线设计大作业

王炜致 2022010542

1 实验目的

通过将单周期处理器改造为流水线处理器的实践操作，深刻领会并巩固《数字逻辑与处理器基础》课程中所学处理器相关知识，重点掌握汇编指令、冒险处理、外设等的架构与设计，提高汇编语言及 verilog 硬件描述语言的理解与编程能力，体会处理器的设计及优化升级过程。

2 设计方案

2.1 指令集扩充设计

根据指导书要求，参阅 MIPS 文档，扩充跳转与分支指令如下：

	PCSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc2	ALUSrc1	ExtOp	LuOp
bne	0(00)	1	0	x(xx)	x	0	x(xx)	0	0	1	x
blez	0(00)	1	0	x(xx)	x	0	x(xx)	0	0	1	x
bgtz	0(00)	1	0	x(xx)	x	0	x(xx)	0	0	1	x
bltz	0(00)	1	0	x(xx)	x	0	x(xx)	0	0	1	x
jalr	2(10)	x	1	1(01)	x	0	2(10)	x	x	x	x

相应地修改了控制信号生成模块 Control.v 及模块端口，详见所附源代码。

2.2 测试数据设计

设置数据内存大小为 256 个字。按要求，准备 24 个待排序正整数，在 DataMemory.v 中初始化 RAM，作为测试样例。

```
parameter RAM_SIZE = 512;
parameter RAM_SIZE_BIT = 8;
reg [31:0] RAM_data [RAM_SIZE - 1: 0];
...
if (reset) begin
        RAM_data[0] <= 32'h00000018; //numbers=24

        RAM_data[1] <= 32'h00000003;
        RAM_data[2] <= 32'h00000028;
        RAM_data[3] <= 32'h00000024;
        RAM_data[4] <= 32'h0000020B;
        RAM_data[5] <= 32'h00003406;
        RAM_data[6] <= 32'h00000A0A;
        RAM_data[7] <= 32'h0000F03A;
        RAM_data[8] <= 32'h0000001B;

        RAM_data[9] <= 32'h00000001;
```

```
            RAM_data[10] <= 32'h00009021;
            RAM_data[11] <= 32'h00000216;
            RAM_data[12] <= 32'h00000072;
            RAM_data[13] <= 32'h00000059;
            RAM_data[14] <= 32'h00000007;
            RAM_data[15] <= 32'h000003E1;
            RAM_data[16] <= 32'h00000B40;

            RAM_data[17] <= 32'h00000034;
            RAM_data[18] <= 32'h0000009F;
            RAM_data[19] <= 32'h000000C8;
            RAM_data[20] <= 32'h0000000D;
            RAM_data[21] <= 32'h000000B1;
            RAM_data[22] <= 32'h000000E9;
            RAM_data[23] <= 32'h0000003a; ;
            RAM_data[24] <= 32'h0000001B;

            for (i = 25; i < RAM_SIZE; i = i + 1)
                RAM_data[i] <= 32'h00000000;
```

2.3 MIPS 指令设计

MIPS 指令主要分为两个部分：排序功能实现部分以及软件形式数显控制部分。
排序部分使用《数字逻辑与处理器基础》课程中设计的插入排序算法，详见所附源代码。排序完成后，

2.4 WELOG 实验板行为设计

采用之前设计的插入排序代码 (insert_sort.asm)，根据处理器实况略作修改。主函数体为：

```
addu $s7,$zero,$zero //compare_count
addu $a1,$zero,$zero //the address of the sequence
addi $a0,$a1,4 //&buffer[1]
lw $a1,0($a1) //N=buffer[0]

jal insertion_sort

addu $a0,$zero,$zero
sw $s7,0($a0) //buffer[0]=compare_count
end:
j end
```

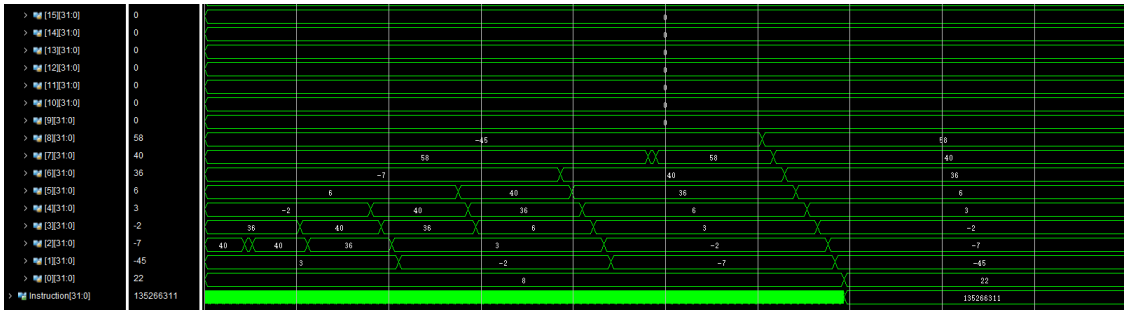
先用单周期处理器验证指令是否能正常运行，再改装为流水线。以下是写入 InstructionMemory.v 的排序算法的机器语言——直接在 DataMemory 中进行数据排序，将排序次数存入 RAM_data[0] 中。

```
8'd0:    Instruction <= 32'h0000b821;
8'd1:     Instruction <= 32'h00002821;
8'd2:    Instruction <= 32'h20a40004;
8'd3:    Instruction <= 32'h8ca50000;

8'd4:    Instruction <= 32'h0c100008;
8'd5:    Instruction <= 32'h00002021;
8'd6:    Instruction <= 32'hac970000;
```

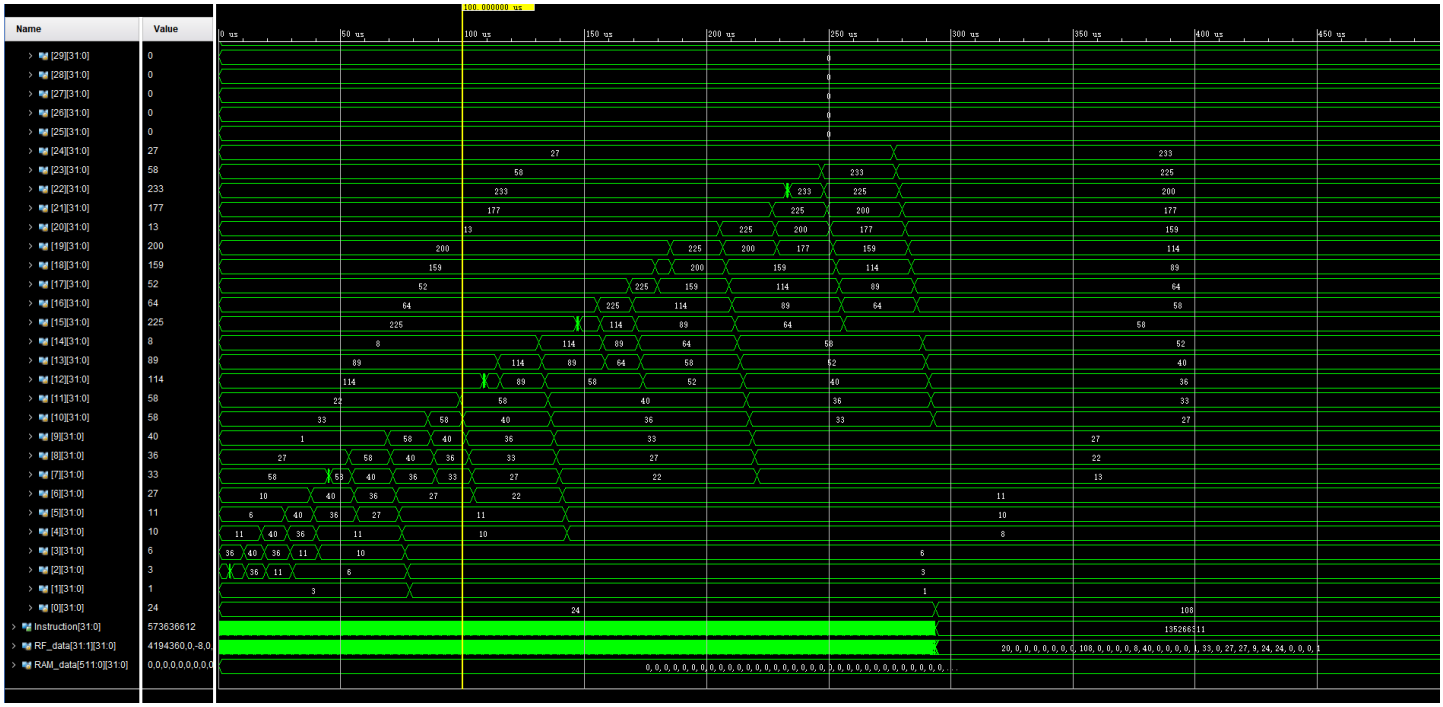
```
8'd7:    Instruction <= 32'h08100007;

8'd8:    Instruction <= 32'h20010004;
8'd9:    Instruction <= 32'h03a1e822;
8'd10:    Instruction <= 32'hafbf0000;
8'd11:    Instruction <= 32'h20060001;

8'd12:    Instruction <= 32'h0c100013;
8'd13:    Instruction <= 32'h0c10002b;
8'd14:    Instruction <= 32'h20c60001;
8'd15:    Instruction <= 32'h14c5fffc;

8'd16:    Instruction <= 32'h8fbf0000;
8'd17:    Instruction <= 32'h23bd0004;
8'd18:    Instruction <= 32'h03e00008;
8'd19:    Instruction <= 32'h20010004;

8'd20:    Instruction <= 32'h03a1e822;
8'd21:    Instruction <= 32'hafbf0000;
8'd22:    Instruction <= 32'h00068880;
8'd23:    Instruction <= 32'h00918821;

8'd24:    Instruction <= 32'h8e280000;
8'd25:    Instruction <= 32'h20010001;
8'd26:    Instruction <= 32'h00c19022;
8'd27:    Instruction <= 32'h22f70001;

8'd28:    Instruction <= 32'h00128880;
8'd29:    Instruction <= 32'h00918821;
8'd30:    Instruction <= 32'h8e290000;
8'd31:    Instruction <= 32'h11280007;

8'd32:    Instruction <= 32'h0128502a;
8'd33:    Instruction <= 32'h20010001;
8'd34:    Instruction <= 32'h102a0004;
8'd35:    Instruction <= 32'h20010001;

8'd36:    Instruction <= 32'h02419022;
8'd37:    Instruction <= 32'h2001ffff;
8'd38:    Instruction <= 32'h1432fff4;
8'd39:    Instruction <= 32'h22470001;

8'd40:    Instruction <= 32'h8fbf0000;
8'd41:    Instruction <= 32'h23bd0004;
8'd42:    Instruction <= 32'h03e00008;
8'd43:    Instruction <= 32'h20010004;

8'd44:    Instruction <= 32'h03a1e822;
8'd45:    Instruction <= 32'hafbf0000;
8'd46:    Instruction <= 32'h00068880;
```

```
8'd47:    Instruction <= 32'h00918821;

8'd48:    Instruction <= 32'h8e280000;
8'd49:    Instruction <= 32'h20010001;
8'd50:    Instruction <= 32'h00c19022;
8'd51:    Instruction <= 32'h00128880;

8'd52:    Instruction <= 32'h00918821;
8'd53:    Instruction <= 32'h8e2b0000;
8'd54:    Instruction <= 32'h22310004;
8'd55:    Instruction <= 32'hae2b0000;

8'd56:    Instruction <= 32'h20010001;
8'd57:    Instruction <= 32'h02419022;
8'd58:    Instruction <= 32'h0247602a;
8'd59:    Instruction <= 32'h20010001;

8'd60:    Instruction <= 32'h102c0002;
8'd61:    Instruction <= 32'h1247fff5;
8'd62:    Instruction <= 32'h1647fff4;
8'd63:    Instruction <= 32'h00078880;

8'd64:    Instruction <= 32'h00918821;
8'd65:    Instruction <= 32'hae280000;
8'd66:    Instruction <= 32'h8fbf0000;
8'd67:    Instruction <= 32'h23bd0004;

8'd68:    Instruction <= 32'h03e00008;
```

排序效果良好，如下：



根据验收要求，进一步验证，对 24 个正整数排序如下：


Name
Value
[29][31:0] 0
[28][31:0] 0
[27][31:0] 0
[26][31:0] 0
[25][31:0] 0
[24][31:0] 27
[23][31:0] 58
[22][31:0] 233
[21][31:0] 177
[20][31:0] 13
[19][31:0] 200
[18][31:0] 159
[17][31:0] 52
[16][31:0] 64
[15][31:0] 225
[14][31:0] 8
[13][31:0] 89
[12][31:0] 114
[11][31:0] 58
[10][31:0] 58
[9][31:0] 40
[8][31:0] 36
[7][31:0] 33
[6][31:0] 27
[5][31:0] 11
[4][31:0] 10
[3][31:0] 6
[2][31:0] 3
[1][31:0] 1
[0][31:0] 24
Instruction[31:0] 573636612
RF_data[31:1][31:0] 4194360,0,-8,0,
RAM_data[511:0][31:0] 0,0,0,0,0,0,0,0,
0 us
50 us
100 us
150 us
200 us
250 us
300 us
350 us
400 us
450 us
100.000000 us

3 流水线改造（无冒险处理）

Stage	R-type	Load	Store	Branch
IF	IR <= MemInst[PC]; PC <= PC+4			
ID	op <= IR[31:26]; A <= Reg[IR[25:21]]; B <= Reg[IR[20:16]]; **Branch: If(A == B) PC <= PC + signext(IR[15:0]) << 2** **Jump: If(JUMP) PC<=NewPC** **EX/MEM.ALUOut**			
EX	ALUOut <= A op B **EX/MEM.ALUOut** **MEM/WB.ALUOut** **MDR**	ALUOut <= A + signext(IR[15:0]) **EX/MEM.ALUOut** **MEM/WB.ALUOut** **MDR**		
MEM		MDR <= MemData[ALUOut];	MemData[ALUout] <= B **MDR**	
WB	Reg[IR[15:11]] <= ALUOut	Reg[IR[20:16]] <= MDR		

CPU.v 的改动最大，需要在各模块输入/输出之间添加寄存器，以下介绍寄存器的初步添加，不考虑冒险。

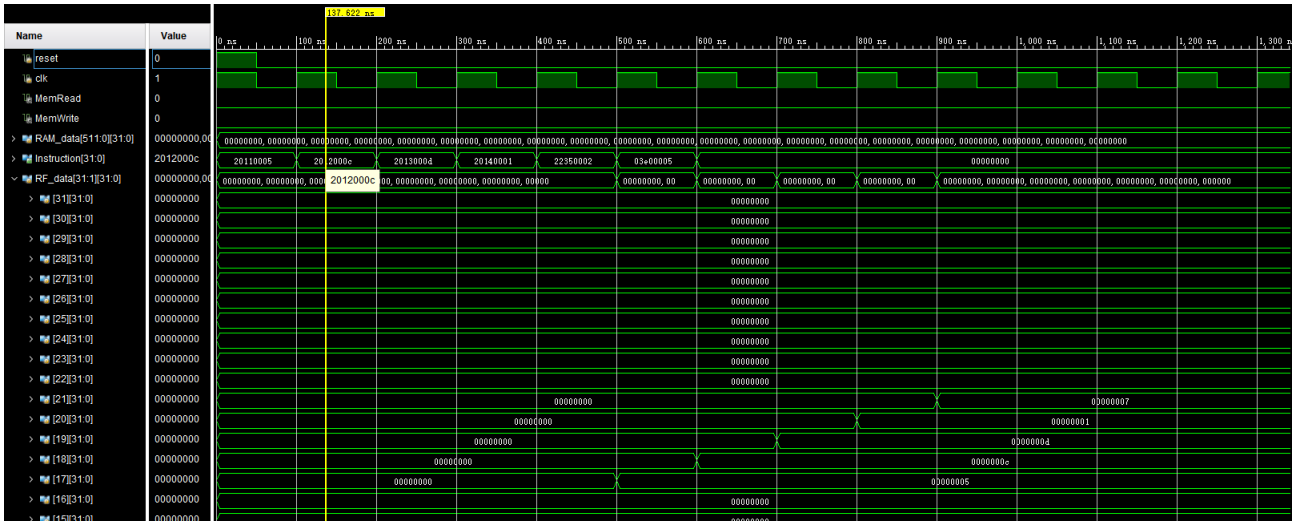
Control.v 即控制信号模块在 IF 阶段调用，产生控制信号，为传递控制信号，在 CPU.v 中插入 IF/ID 寄存器：

```
    always @(posedge reset or posedge clk) begin//IF/ID PLUGIN BEGIN ↑IF
       if (reset) begin
           IFID_Instruction <= 32'b0;
           IFID_RegDst <= 2'b00;
           IFID_PCSrc <= 2'b00;
           IFID_Branch <= 3'b000;
           IFID_MemRead <= 1'b0;
           IFID_MemWrite <= 1'b0;
           IFID_MemtoReg <= 2'b00;
           IFID_ALUSrc1 <= 1'b0;
           IFID_ALUSrc2 <= 1'b0;
           IFID_ALUOp <= 4'b0000;
           IFID_ExtOp <= 1'b0;
           IFID_LuOp <= 1'b0;
           IFID_RegWrite <= 1'b0;
       end     //more to consider
       else begin
            IFID_Instruction <= Instruction;
            IFID_RegDst <= RegDst;
            IFID_PCSrc <= PCSrc;
            IFID_Branch <= Branch;
            IFID_MemRead <= MemRead;
            IFID_MemWrite <= MemWrite;
            IFID_MemtoReg <= MemtoReg;
            IFID_ALUSrc1 <= ALUSrc1;
            IFID_ALUSrc2 <= ALUSrc2;
            IFID_ALUOp <= ALUOp;
            IFID_ExtOp <= ExtOp;
            IFID_LuOp <= LuOp;
            IFID_RegWrite <= RegWrite;
       end
    end//IF/ID PLUGIN END ↓ID
```

ID 阶段调用寄存器堆 (读)，注意 WB 阶段亦涉及调用寄存器堆 (写)，考虑遵循先写后读原则 (在 RegisterFile.v 中修改)。插入 ID/EX 寄存器：

将 Jump,Branch 均置于 ID 阶段执行，则 ALU 不需要 Zero。无冒险处理的流水线例程测试如下：

Bkpt	Address	Code	Basic	
	4194304	0x20110005	addi $17, $0, 5	1: addi $s1, $zero, 5
	4194308	0x2012000c	addi $18, $0, 12	2: addi $s2, $zero, 12
	4194312	0x2013000d	addi $19, $0, 13	3: addi $s3, $zero, 13
	4194316	0x20140001	addi $20, $0, 1	4: addi $s4, $zero, 1
	4194320	0x22350002	addi $21, $17, 2	5: addi $s5, $s1, 2
	4194324	0x08100005	j 4194324	7: j end



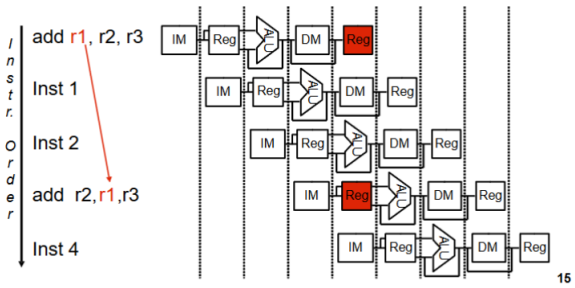
4 冒险处理

4.1 结构冒险

InstructionMemory 与 DataMemory 已作分离；R 型指令 Mem 阶段已空置；ALU 已作功能疏解。

4.2 数据冒险

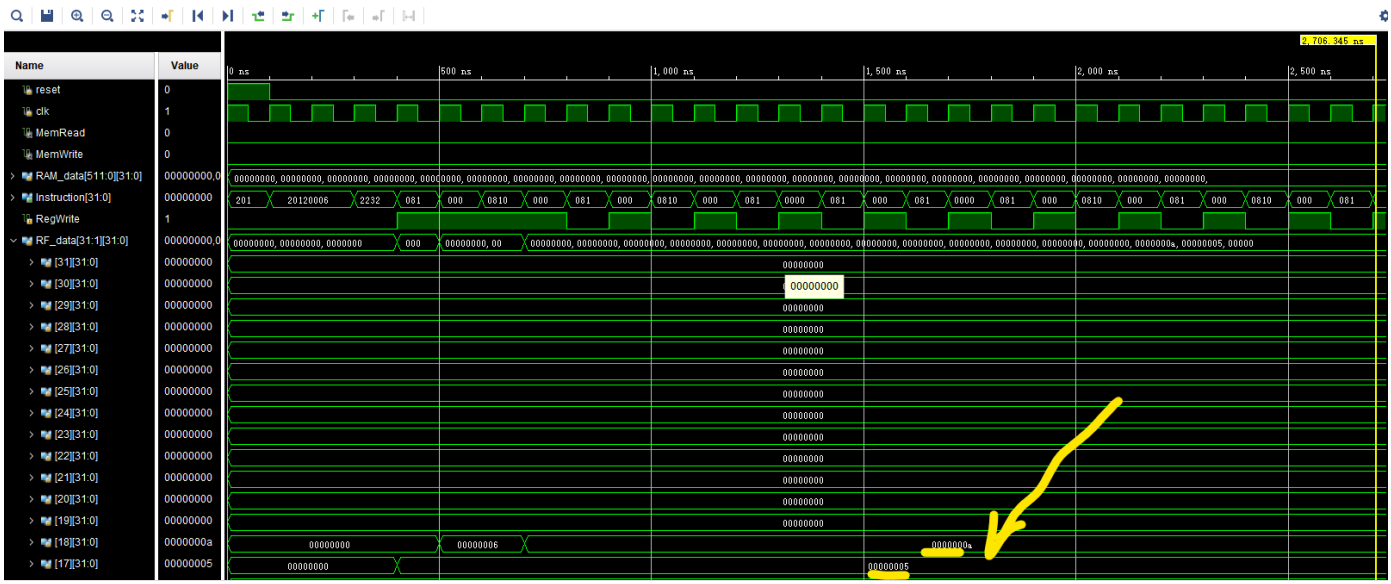
4.2.1 同时读写寄存器堆



通过先写后读策略处理。利用 always 块的阻塞赋值实现代码执行的先后顺序。

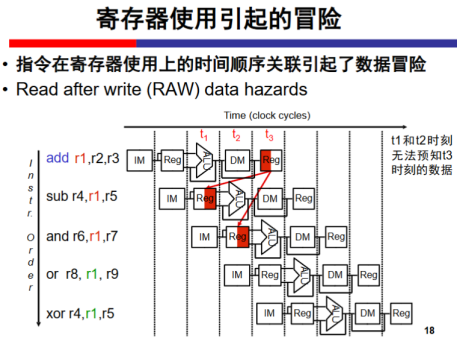
```
always @(*) begin
    if (RegWrite && (Write_register != 5'b00000))
        RF_data[Write_register] = Write_data;//first write
    Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: RF_data[Read_register1];
    Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000: RF_data[Read_register2];
end
```

验证如下：



Bkpt	Address	Code	Basic	
	4194304	0x20110005	addi $17, $0, 5	1: addi $s1, $zero, 5
	4194308	0x20120006	addi $18, $0, 6	2: addi $s2, $zero, 6
	4194312	0x20120006	addi $18, $0, 6	3: addi $s2, $zero, 6
	4194316	0x22320005	addi $18, $17, 5	4: addi $s2, $s1, 5
	4194320	0x08100004	j 4194320	6: j end

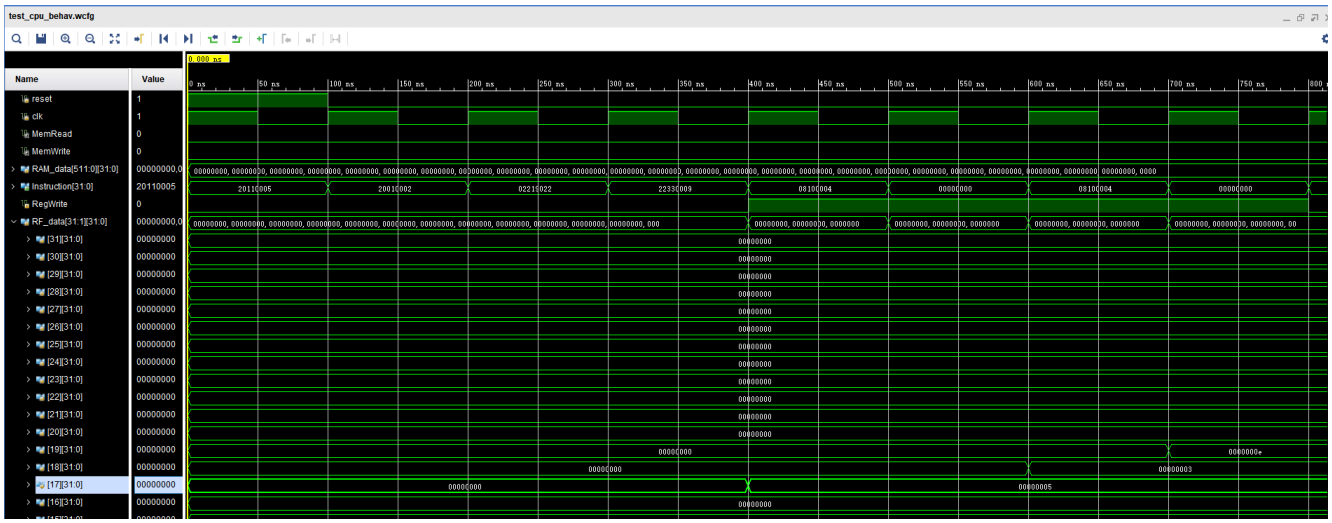
4.2.2 时间顺序关联（ALU 输出）



ALU 输出运算结果后，立刻转发给 ALU 输入端。需要修改 ALU 的输入端（二路），增加 MUX 选择合适皂的输入信号。

```
wire [32 -1:0] in1,in2;//rs,rt forwarding
assign in1 = (~IDEX_ALUSrc1 && MEMWB_Memory_Read && MEMWB_Write_register
    && MEMWB_Write_register == IDEX_Instruction[25:21])? MEMWB_MemBus_Read_Data:
    (~IDEX_ALUSrc1 && MEMWB_RegWrite && MEMWB_Write_register
    && (MEMWB_Write_register == IDEX_Instruction[25:21])
    && (EXMEM_Write_register != IDEX_Instruction[25:21]
    || ~EXMEM_RegWrite))? MEMWB_ALU_out:
    (~IDEX_ALUSrc1 && EXMEM_RegWrite && EXMEM_Write_register
    && (EXMEM_Write_register == IDEX_Instruction[25:21]))? EXMEM_ALU_out:
    ALU_in1;
assign in2 = (~IDEX_ALUSrc2 && MEMWB_Memory_Read && MEMWB_Write_register
    && MEMWB_Write_register == IDEX_Instruction[20:16])? MEMWB_MemBus_Read_Data:
    (~IDEX_ALUSrc2 && MEMWB_RegWrite && MEMWB_Write_register
    && (MEMWB_Write_register == IDEX_Instruction[20:16])
    && (EXMEM_Write_register != IDEX_Instruction[20:16]
    || ~EXMEM_RegWrite))? MEMWB_ALU_out://mind load-store
    (~IDEX_ALUSrc2 && EXMEM_RegWrite && EXMEM_Write_register
    && (EXMEM_Write_register == IDEX_Instruction[20:16]))? EXMEM_ALU_out:
    ALU_in2;
```

验证如下：



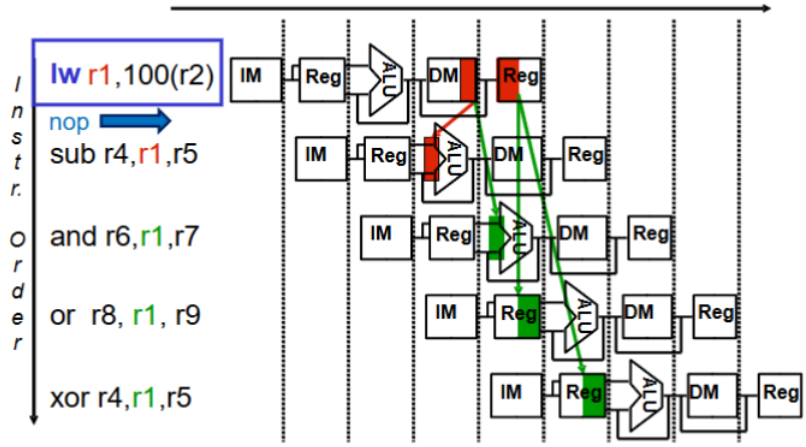
Text Segment

Bkpt	Address	Code	Basic	
	4194304	0x20110005	addi $17,$0,5	1: addi $s1,$zero,5
	4194308	0x20010002	addi $1,$0,2	2: subi $s2,$s1,2
	4194312	0x02219022	sub $18,$17,$1	
	4194316	0x22330009	addi $19,$17,9	3: addi $s3,$s1,9
	4194320	0x08100004	j 4194320	5: j end

此外，若下一条指令为 Branch（为简便，默认 jr,jalr 使用 $31，故不需要处理），则不能不 stall 一个周期，再将数据转发到 ID 阶段。验证如下：

Bkpt	Address	Code	Basic	
	4194304	0x20110001	addi $17,$0,1	2: addi $s1,$zero,1
	4194308	0x1e20fffe	bgtz $17,-2	3: bgtz $s1,again
	4194312	0x08100002	j 4194312	5: j end

4.2.3 load-use 冒险



load-use 冒险包含 load-R 类和 load-store 类冒险，前者不可避免地需要 stall 一个周期。故需要将读出数据作转发。处理 load-store 冒险时，在 ALU 输入 forwarding 条件中需要分辨立即数加法与寄存值加法；DataMemory 写端口也需要引入 MUX 作 forwarding 判断。

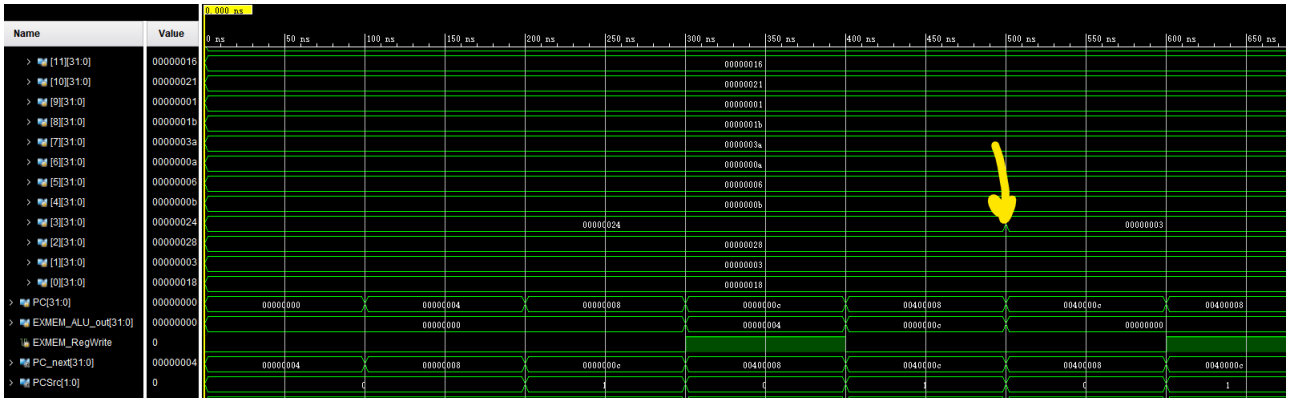
非常特殊的情况：lw $s1,4($zero)，sw $s1,0($s1)，由于 stall 一个周期，无法由 MEM/WB 转发，同时 $s1 尚未更新。故需要特殊处理：

```
1    assign IDEX_Databus2_prevent_loadstore = (IDEX_MemWrite && MEMWB_Write_register
2        && (MEMWB_Write_register == IDEX_Instruction[20:16]))? MEMWB_MemBus_Read_Data:
3        IDEX_Databus2;
```

一般 load-use 处理效果如下：

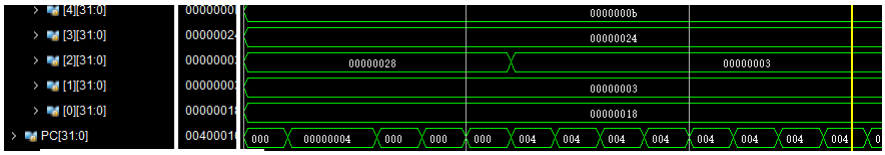
4194304	0x8c110004	lw $17,4($0)	1: lw $s1,4($zero)#s1=3
4194308	0x02318820	add $17,$17,$17	2: add $s1,$s1,$s1
4194312	0x08100002	j 4194312	4: j end

一般 load-store 处理效果如下：



Bkpt	Address	Code	Basic	
☐	4194304	0x8c110004	lw $17,4($0)	1: lw $s1,4($zero)
☐	4194308	0xac110008	sw $17,8($0)	2: sw $s1,8($zero)
☐	4194312	0x08100002	j 4194312	4: j end

特殊 load-store 处理效果如下：



Text Segment

Bkpt	Address	Code	Basic	
☐	4194304	0x8c110004	lw $17,4($0)	1: lw $s1,4($zero)#s1=3
☐	4194308	0xae310005	sw $17,5($17)	2: sw $s1,5($s1)#40=3
☐	4194312	0x8e320005	lw $18,5($17)	3: lw $s2,5($s1)#s2=3
☐	4194316	0x08100003	j 4194316	5: j end

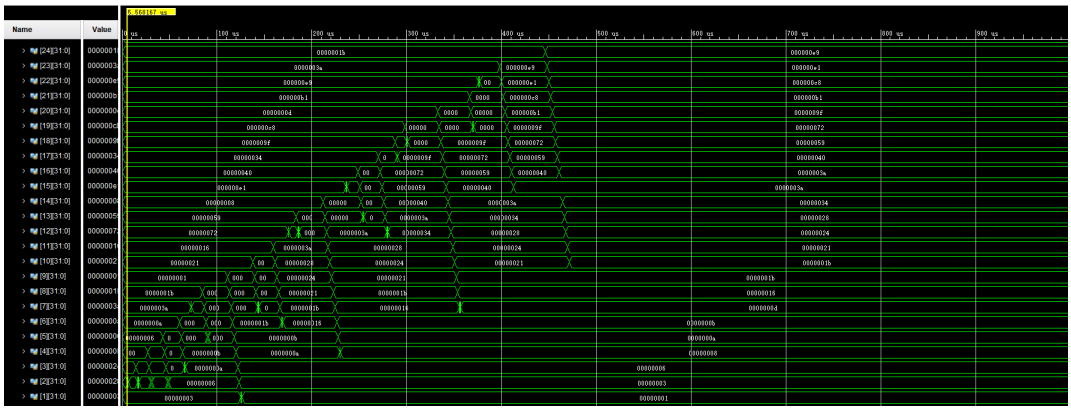
另外，load 后若紧跟 Branch 指令且发生冒险，则需要 stall 两个周期，暂不作处理，通过代码添加 3 个 nop 指令（0x00000000）解决。

至此，数据冒险基本解决。

4.3 控制冒险

由于提前到 ID 阶段判断，分支指令、跳跃指令的下一条指令必然为 nop。据此设定修改即可。

利用流水线处理器排序结果如下：



5 外设配置