流水线设计大作业

王炜致 2022010542

目录

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 设计方案及设计过程（含仿真情况与关键代码）

2.1 总体设计

该流水线为经典的五级流水线，由《数字逻辑与处理器基础》课程中提供的单周期处理器改造而来；此外，还沿用了理论课中编写的排序代码。这两块基石的基本原理，在此不再赘述，以下展示的是利用单周期处理器运行排序代码的仿真结果，旨在证明单周期处理器代码及排序代码（为适配处理器，略有修改）的可靠性:

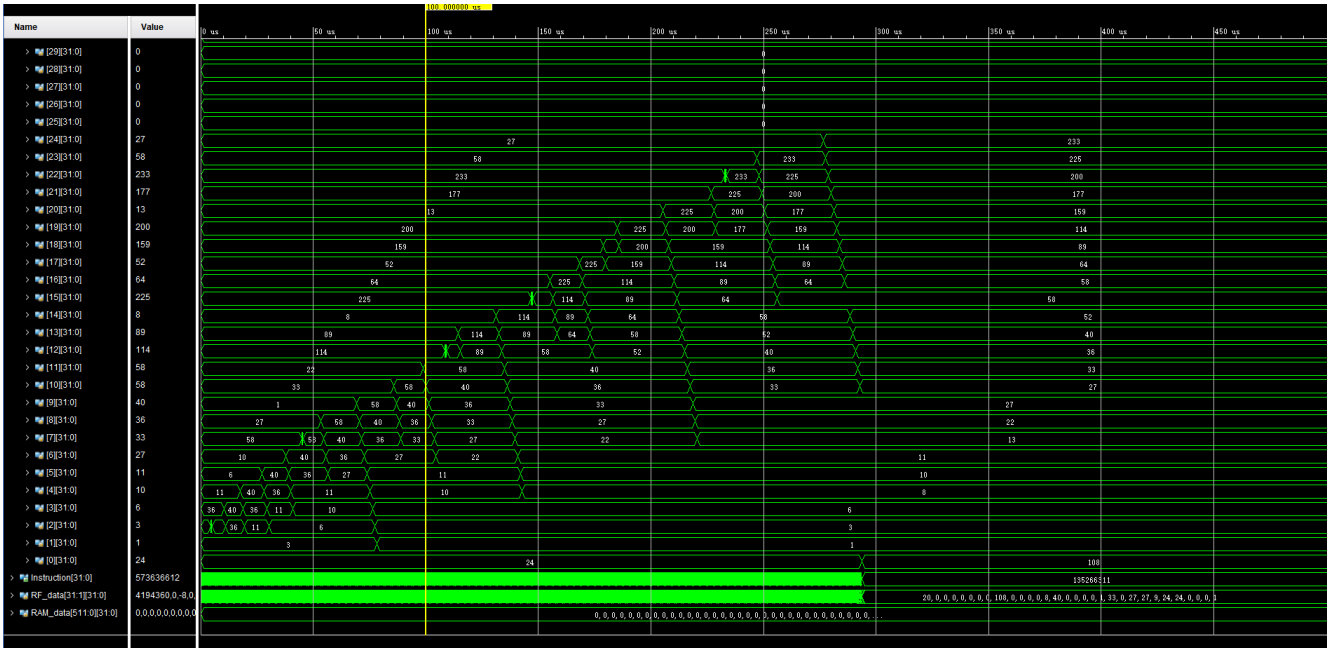


图 1: 单周期处理器运行插入排序代码

在改造的过程中，我在 CPU.v 模块中划分 IF,ID,EX,MEM,WB 五个阶段，在阶段之间插入 IFID,IDEX,EXMEM,MEMWB 四个寄存器堆以服务于数据转发等功能（详见“冒险处理设计”一节），这也是改造的主体部分；此外，为扩充指令集，Control.v 需要扩展或调整 Branch 等信号 (Branch[2:0])；为实现数据初始化及配合软件控制，DataMemory.v 也需要接受一定的改造与扩展；为在特定处理器条件下适应处理器局限性、执行给定的任务，汇编代码也需要经过修订与增补，再写入 InstructionMemory.v 中。

该流水线处理器的实现历时约五天（7.1-7.5）。预估到工作量庞大，为避免后期 debug 困难，我遵循小心谨慎、勤做仿真、步步为营的工作方针，因而总体设计历程层次分明：验证单周期处理器及代码可靠性、扩充指令集、插入寄存器堆（不考虑冒险）、处理结构冒险、处理数据冒险、处理控制冒险，到此先使用译码器方法上板验证，硬件调试成功后，再使用软件方法完成了实验。除串口部分外，实验内容均已完成并提前约 2 周通过验收。设计历程有存档，可以在附录文件中查看。

以下展示的是流水线处理器设计总框图，为使画面简明扼要，略去不画 MUX,Forwarding,Jump,Branch 单元的控制信号。相关信息将在下文具体介绍。由图可见，IF 阶段主要执行取指令及生成控制信号操作；ID 阶段主要执行后序指令的判断与跳转、生成 ALU 控制信号、读取寄存器堆操作；EX 阶段主要进行 ALU 运算操作；MEM 阶段主要进行读写存储器堆及控制外设（BCD）操作；WB 阶段主要进行写回操作。

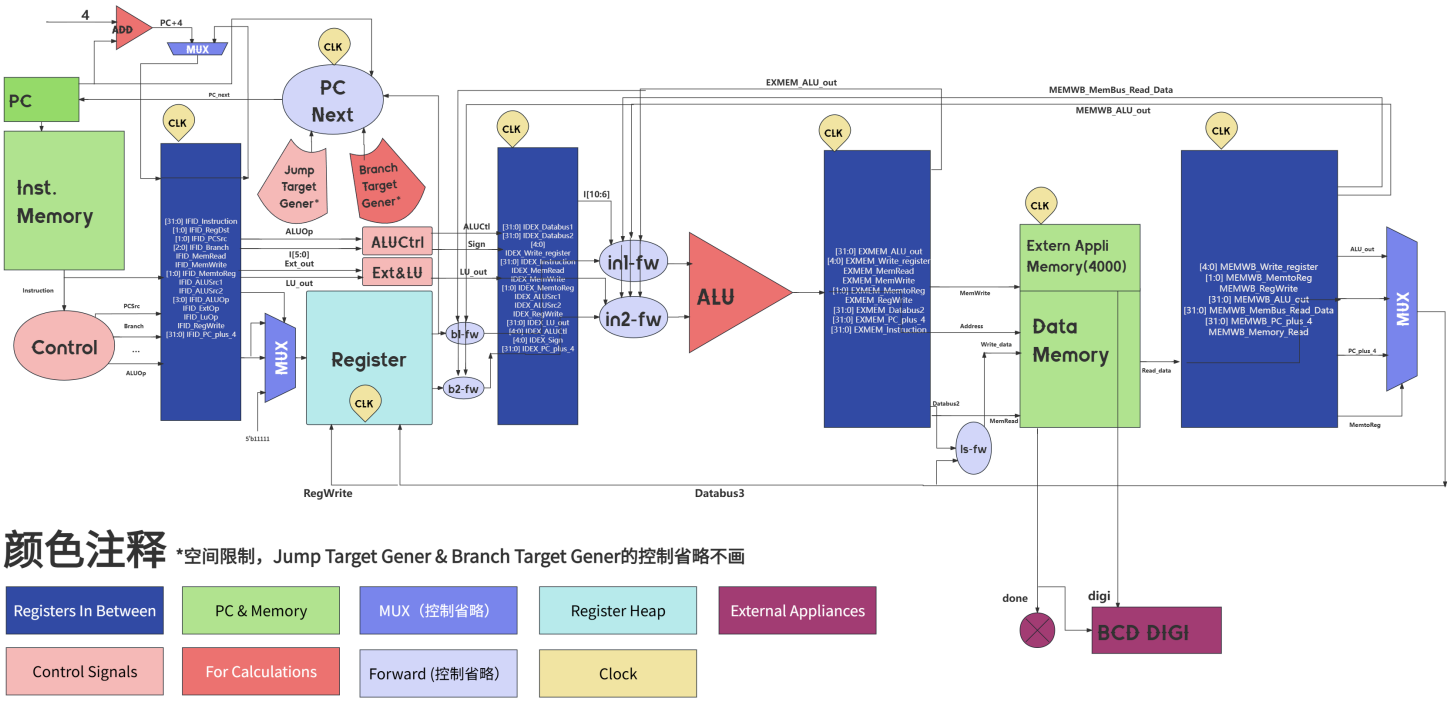


图 2: 流水线处理器设计框图（有省略）

2.2 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。

2.3 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 RAMdata[1:24]，作为测试样例。RAMdata[0] 存储的是待排序正整数的数量。

```
//DataMemory.v
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;
```

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 冒险处理设计

2.4.1 结构冒险

在前述框图中，InstructionMemory 与 DataMemory 已作分离；R 型指令 Mem 阶段已空置；ALU 已作功能疏解。结构冒险已经得到解决。

2.4.2 数据冒险

同时读写寄存器堆

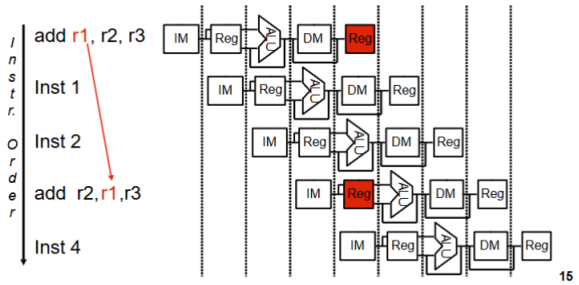


图 3: 数据冒险之同时读写寄存器堆

根据理论所学，应通过先写后读策略处理：

```
//RegisterFile.v
```

```
    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：

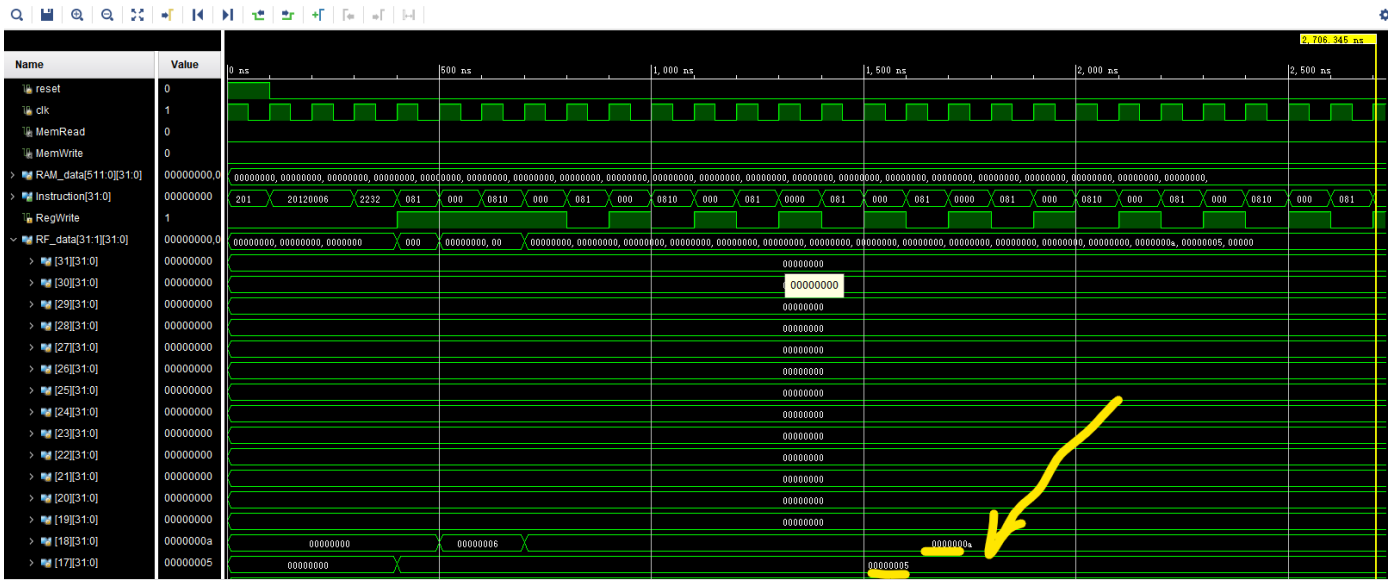


图 4: 先写后读前仿验证

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

图 5: 先写后读验证测试代码

但在布线过程中发现 always 块内阻塞赋值会报错，故后期改为转发实现。

时间顺序关联（ALU 输出）

对于 R 型指令，ALU 输出运算结果后尚未写回寄存器堆，而后序指令可能需要使用新数据，故应转发给 ALU 输入端。需要修改 ALU 的输入端（二路），增加 MUX 选择合适的输入信号：

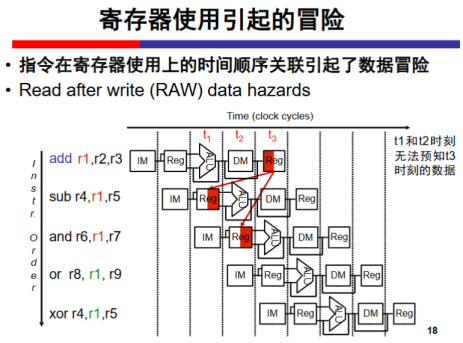


图 6: 数据冒险之 ALU 输出

```
//CPU.v
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（汇编代码及其对应的仿真结果）：

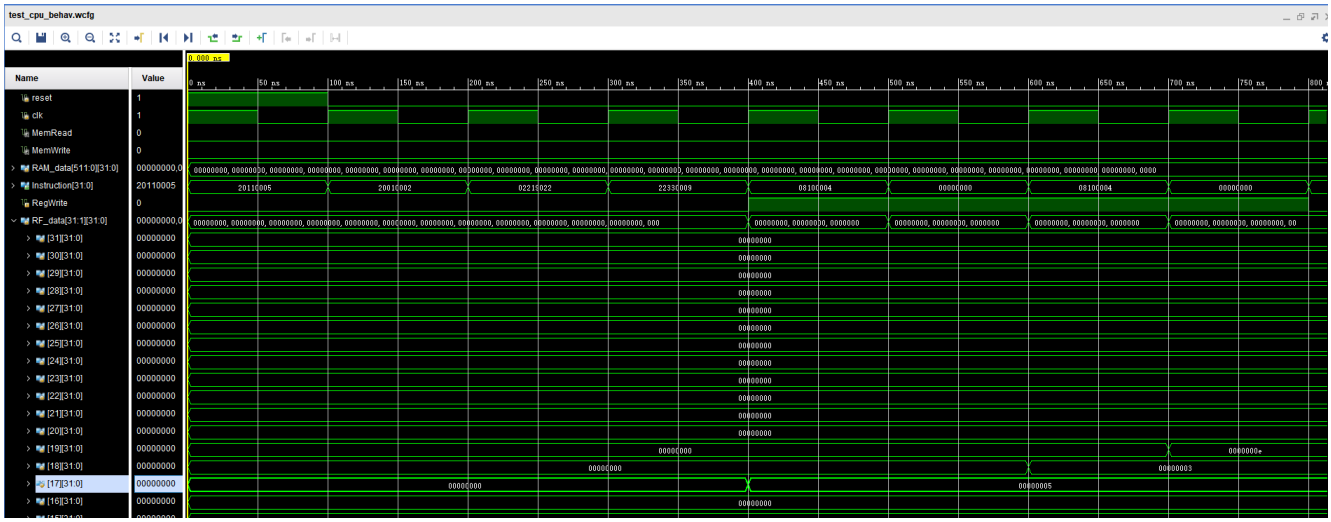


图 7: 数据转发前仿验证

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

图 8: 数据转发验证测试代码

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：

```
//CPU.v
assign Databus1_brc = (MEMWB_Write_register && MEMWB_RegWrite
    && MEMWB_Write_register == IFID_Instruction[25:21] &&
    (EXMEM_Write_register != IFID_Instruction[25:21] || ~EXMEM_RegWrite))? MEMWB_ALU_out:
    (EXMEM_Write_register && EXMEM_RegWrite
    && EXMEM_Write_register == IFID_Instruction[25:21])? EXMEM_ALU_out:
    Databus1;
assign Databus2_brc = (MEMWB_Write_register && MEMWB_RegWrite
    && MEMWB_Write_register == IFID_Instruction[20:16] &&
     (EXMEM_Write_register != IFID_Instruction[20:16] || ~EXMEM_RegWrite))? MEMWB_ALU_out:
    (EXMEM_Write_register && EXMEM_RegWrite
    && EXMEM_Write_register == IFID_Instruction[20:16])? EXMEM_ALU_out:
    Databus2;
```

验证如下（汇编代码及其对应的仿真结果）:

图 9: Branch 指令前自带 stall 前仿验证

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

图 10: Branch 指令前自带 stall 验证测试代码

2.4.3 load-use 冒险

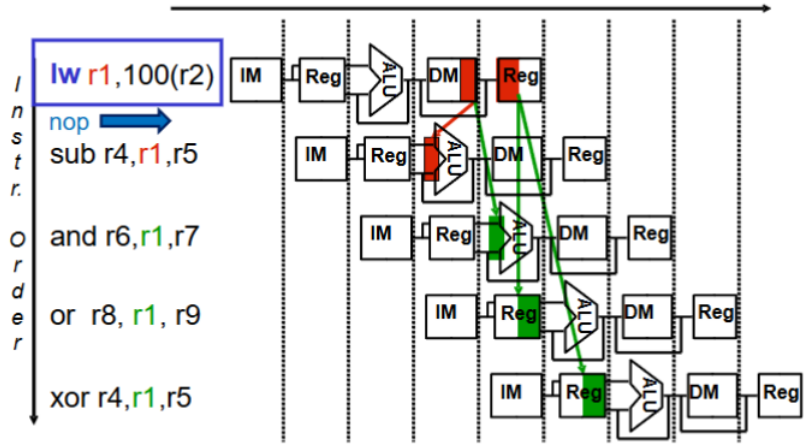


图 11: load-use 冒险

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 forwarding 判断:

```
//CPU.v
assign MemBus_Write_Data  = (Memory_Write && MEMWB_Write_register
                            && (MEMWB_Write_register == EXMEM_Instruction[20:16]))? Databus3
                            :EXMEM_Databus2;//for load-store special case:lw s1,x;sw s1,s1
```

在仿真测试时发现一种非常特殊的情况——形如 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：

```
//CPU.v
wire [32 -1:0] IDEX_Databus2_prevent_loadstore;
assign IDEX_Databus2_prevent_loadstore = (IDEX_MemWrite && MEMWB_Write_register
    && (MEMWB_Write_register == IDEX_Instruction[20:16]))? MEMWB_MemBus_Read_Data:
    IDEX_Databus2;
```

一般 load-use 处理效果如下（汇编代码及其对应的仿真结果）：

图 12: load-use 处理前仿验证

☐	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

图 13: load-use 处理验证测试代码

一般 load-store 处理效果如下（汇编代码及其对应的仿真结果）：

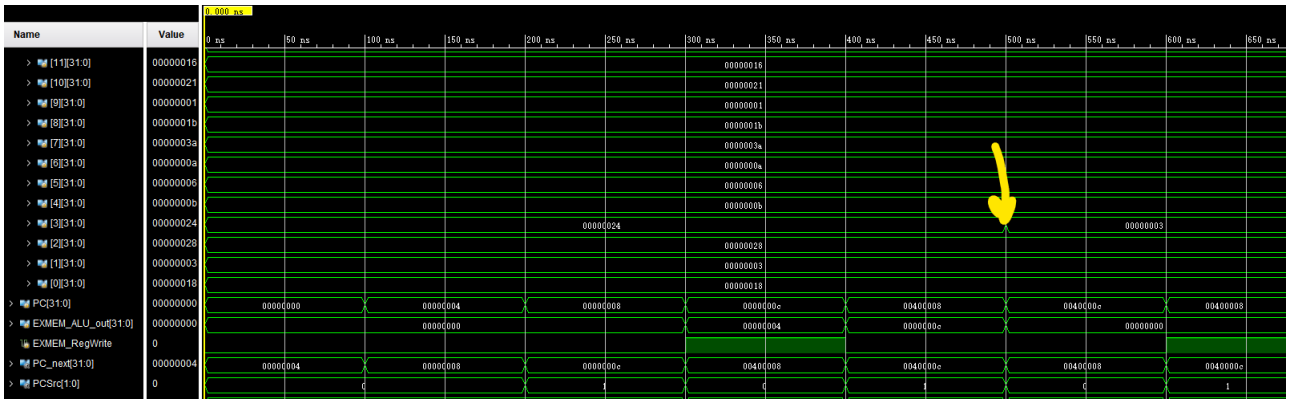


图 14: load-store 处理前仿验证

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

图 15: load-store 处理验证测试代码

特殊 load-store 处理效果如下（汇编代码及其对应的仿真结果）：

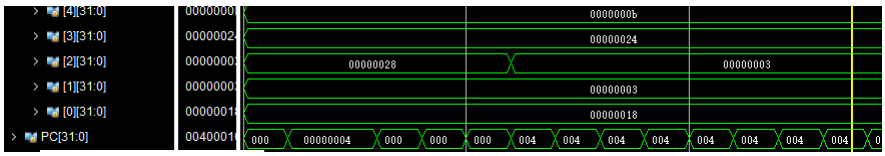


图 16: 特殊 load-store 处理验证测试代码

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

图 17: 特殊 load-store 处理验证测试代码

另外，load 后若紧跟 Branch 指令且发生冒险，则需要 stall 两个周期，在此不作处理，通过在汇编代码中添加 3 个 nop 指令（0x00000000）解决。至此，数据冒险基本解决。

2.4.4 控制冒险

由于提前到 ID 阶段进行新指令相关操作，接下来执行哪条指令需要等待两个周期才能出结果。故设定 Branch、Jump 指令下一条指令必为 nop，即等待一个周期，这样能够比较方便地规避控制冒险。

处理完上述冒险后，利用流水线处理器排序结果仿真如下，可见排序已能正确实现：

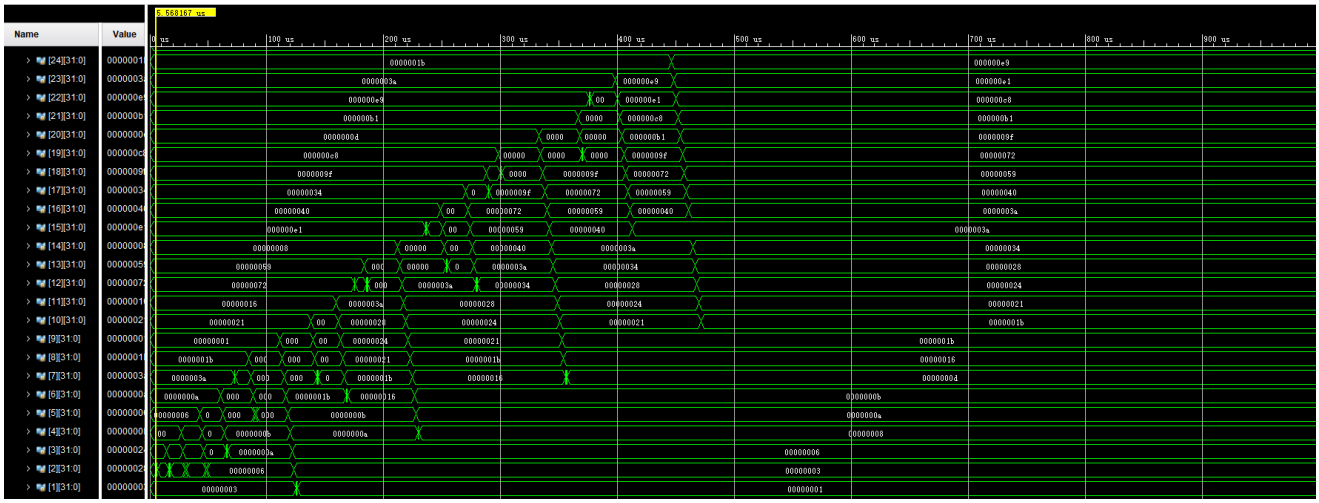


图 18: 冒险处理后排序效果的前仿验证

2.5 外设与 WELOG 实验板行为设计

2.5.1 外设设计

我的流水线外设设计主要参考了 PPT 中的设计，但也存在一大不同。我的汇编代码中不存在读取外设存储器的 lw 指令，verilog 代码中也省略了读外设存储器的硬件配套。事实上，只需要通过“写入外设存储器”的指令来维护外设控制信号，而“读取外设存储器”的操作，应当是不必要的。让外设存储器直接与外设相连，即直接控制外设，也能保证功能的正常实现，且更加简捷。

更多相关内容可参阅“算法指令”一节。

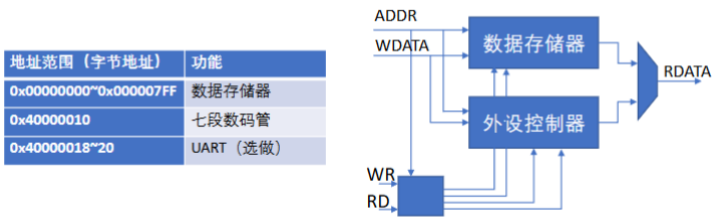


图 19: PPT 提供的外设参考设计

2.5.2 WELOG 实验板行为设计

排序程序执行完之后，一颗 LED 将点亮以指示完成排序，此后七段数码管将从小到大显示用十六进制表示的正整数，每隔 1 秒切换，无限循环显示。约定字母按如下规则显示：

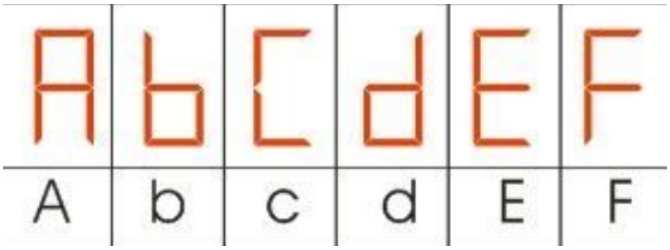


图 20: 十六进制数显示约定

3 算法指令

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。前者使用插入排序，不再赘述；下面将介绍后者，这段代码在排序代码运行完毕之后执行：

```
    #MIPS_Code.asm
    addi $a1,$a1,1
    sll $a1,$a1,2
    addi $s6,$zero,1
    sw $s6,0($a1)
    addi $a1,$a1,4 #saving current sorted data

    lui $s0,0x4000

    addi $s0,$s0,16 #40000010
    addi $t1,$zero,80

rounding:
   lw $s1,0($a1) #s1 0xabcd,renewed every second
   add $t0,$zero,$zero #zero,count 100000 then break
bcd4:#lowest
   addi $t0,$t0,1
   andi $s2,$s1,0x0000000f #get lowest 4 bits
   jal judging
   addi $s3,$s3,256 #add ano lowest
   sw $s3,0($s0)
   bne $t0,$t1,bcd4

   add $t0,$zero,$zero #zero,count 100000 then break
bcd3:
   addi $t0,$t0,1
   andi $s2,$s1,0x000000f0 #get 2-lowest 4 bits
   srl $s2,$s2,4
   jal judging
   addi $s3,$s3,512
   sw $s3,0($s0)
   bne $t0,$t1,bcd3
```

```
33
34      add $t0,$zero,$zero #zero,count 100000 then break
35  bcd2:
36      addi $t0,$t0,1
37      andi $s2,$s1,0x00000f00 #get 2-highest 4 bits
38      srl $s2,$s2,8
39      jal judging
40      addi $s3,$s3,1024
41      sw $s3,0($s0)
42      bne $t0,$t1,bcd2
43
44      add $t0,$zero,$zero #zero,count 100000 then break
45  bcd1:
46      addi $t0,$t0,1
47      andi $s2,$s1,0x0000f000 #get highest 4 bits
48      srl $s2,$s2,12
49      jal judging
50      addi $s3,$s3,2048 #add ano highest
51      sw $s3,0($s0)
52      bne $t0,$t1,bcd1
53
54      j rounding
55
56  judging:
57      beq $s2,15,f
58      beq $s2,14,e
59      beq $s2,13,d
60      beq $s2,12,c
61      beq $s2,11,B
62      beq $s2,10,a
63      beq $s2,9,ni
64      beq $s2,8,ei
65      beq $s2,7,se
66      beq $s2,6,si
67      beq $s2,5,fi
68      beq $s2,4,fo
69      beq $s2,3,th
70      beq $s2,2,tw
71      beq $s2,1,on
72      beq $s2,0,ze
73
74  f:addi $s3,$zero,113 #0111 0001
75      jr $ra
76  e:addi $s3,$zero,121
77      jr $ra
78  d:addi $s3,$zero,94
79      jr $ra
80  c:addi $s3,$zero,57
81      jr $ra
82  B:addi $s3,$zero,124
```

```
    jr $ra
a:addi $s3,$zero,119
    jr $ra
ni:addi $s3,$zero,111
    jr $ra
ei:addi $s3,$zero,127
    jr $ra
se:addi $s3,$zero,7
    jr $ra
si:addi $s3,$zero,125
    jr $ra
fi:addi $s3,$zero,109
    jr $ra
fo:addi $s3,$zero,102
    jr $ra
th:addi $s3,$zero,79
    jr $ra
tw:addi $s3,$zero,91
    jr $ra
on:addi $s3,$zero,6
    jr $ra
ze:addi $s3,$zero,63
    jr $ra
```

软件操作过程为：当排序完成后，在紧接正整数序列存储区域的下一单元存入“1”（即由 0 变为 1），表示排序完成。；“1”的下一个单元则用来存放当前需要展示的正整数，地址为 a1。

接下来进入轮询环节。verilog 部分负责控制正整数更新（见下方代码），即每隔 1 秒在 a1 地址中存入序列中新的正整数值，这样汇编代码中 s1 可以在每次 rounding 循环中自然地检测并获取数据更新。接着，需要扫描当前正整数的各个数位。通过 andi 分别读取正整数在 16 进制下的 4 个数位，跳转到 judging 处作（0-15）的判断，再跳转译码，给出数码管的控制信号。

```
//DataMemory.v
reg [16 -1:0] counting;//fetch data
parameter Hz = 100000000;//1s,based on 100MHz
reg [32 -1:0] cHz;//1s timer
always @(posedge reset or posedge clk)begin
        if (reset) begin
            RAM_data[0] <= 32'h00000018; //numbers=24
            ...
            cHz <= 32'h00000000;
            counting <= 16'b1;
            digi <= 12'b0;
        end
        else begin
            cHz <= (cHz == Hz-32'd1) ? 32'd0 : cHz + 32'd1;
            if (MemWrite) begin
                if(Address == 32'h40000010)
                        digi <= Write_data[11:0];//write into digi with address 40000010(sw 40000010)
                    else
                        RAM_data[Address[RAM_SIZE_BIT + 1:2]] <= Write_data;
            end
            else if (cHz == 32'd0 && done) begin
```

```
                    RAM_data[26] <= RAM_data[counting];//address 26 saves the current data
                if (counting < RAM_data[0])
                    counting <= counting+16'b1;
                else
                    counting <= 16'b1;
            end
        end
end
```

DataMemory.v 输出控制信号 [11:0]digi，与外设相连：

```
    wire [11:0] digi;//generate from DMem, designed using mips insts
    assign sel = digi[11:8];//output [3:0]sel
    assign leds = digi[7:0];//output [7:0]leds
```

4 关键代码和文件清单 (此处关键代码略)

提交的文件包含本实验报告和 2 个文件夹。

CODES 文件夹中包含 3 个源代码文件夹。其中 FINAL_VERSION_FOR_BOARD_RUNNING 文件夹为专用于上板运行的版本，为确保安全不超频，工作频率为 50MHz（用 2 分频模块处理 100MHz 板载时钟），此外还负责资源占用测量；FINAL_VERSION_FOR_TIMING & CPI 文件夹为测量最高主频的版本，此外还负责 CPI 测量（仿真频率为 50MHz）；SINGLE_CYCLE 文件夹为单周期处理器运行排序算法的版本；HIST 文件夹为历史版本记录，供参考。

PROJECTS 文件夹中包含上板运行（onboard）、测量最高主频（pipeline）两个项目版本。

以下是 FINAL_VERSION_FOR_BOARD_RUNNING 中的文件：

文件	功能
test_CPU.v	仿真文件
CPU.v	CPU 主模块，集成了段间寄存器
clk_50M.v	分频得到 50MHz 安全频率
InstructionMemory.v	存储机器语言表示的排序算法
Control.v	生成处理器各部件的控制信号
RegisterFile.v	寄存器堆
ALUControl.v	生成 ALU 的控制信号
ALU.v	计算模块
DataMemory.v	数据存储器和外设存储器
const.xdc	管脚约束文件
Instructions.txt	排序算法机器语言代码
MIPS_Code.asm	排序算法汇编语言代码
Mars4_5.jar	MIPS 编译器

除了一般的 verilog 文件，FINAL_VERSION_FOR_BOARD_RUNNING 还包括：

文件	功能
a.in	输入数据文件
gen.cpp	用于生成 a.in
MIPS_Code_CPI.asm	测量指令数用

5 综合情况

5.1 时序：最高主频

如图，经过反复测量，发现时钟周期为 13.5710ns（Vivado 四舍五入为 13.5710ns）时，WNS 几乎为 0，故可以计算最高时钟频率

$$f_m = \frac{1}{13.5710ns} \approx 73.687MHz$$

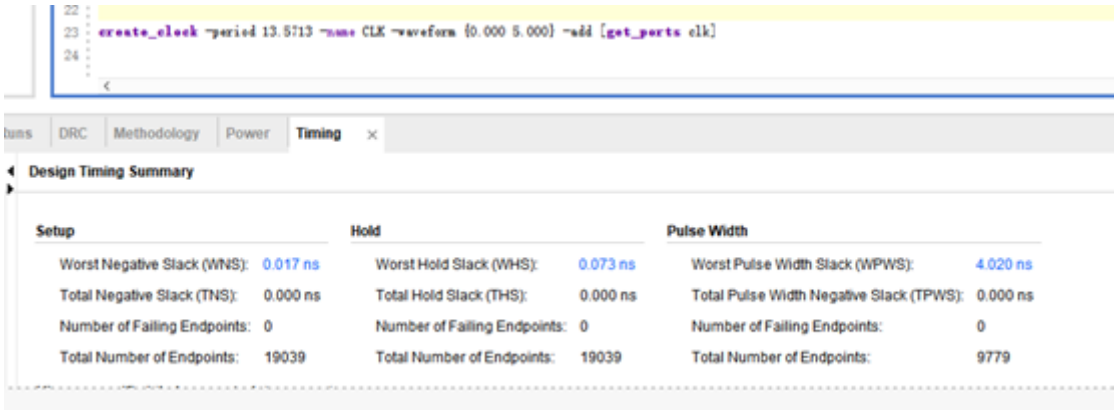


图 21: 测量流水线 CPU 最高主频（Implementation）

5.2 面积：资源占用

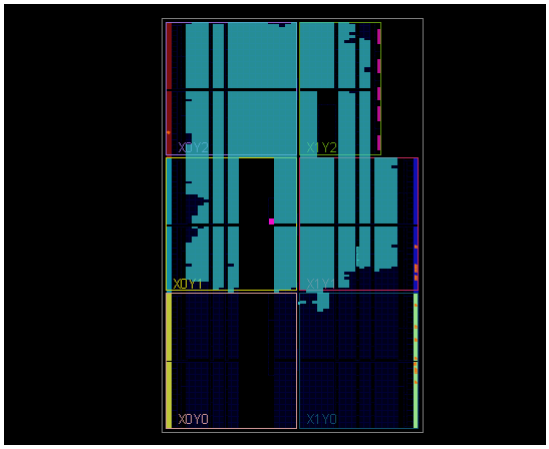
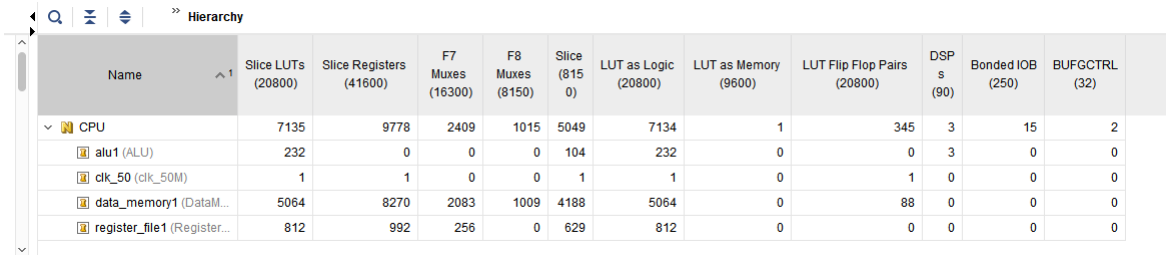


图 22: 流水线 FPGA 面积占用



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	DSPs (90)	Bonded IOB (250)	BUFGCTRL (32)
CPU	7135	9778	2409	1015	5049	7134	1	345	3	15	2
alu1 (ALU)	232	0	0	0	104	232	0	0	3	0	0
clk_50 (clk_50M)	1	1	0	0	1	1	0	1	0	0	0
data_memory1 (DataM...	5064	8270	2083	1009	4188	5064	0	88	0	0	0
register_file1 (Register...	812	992	256	0	629	812	0	0	0	0	0

图 23: 流水线资源占用数量

Summary

Resource	Utilization	Available	Utilization %
LUT	7135	20800	34.30
LUTRAM	1	9600	0.01
FF	9778	41600	23.50
DSP	3	90	3.33
IO	15	250	6.00

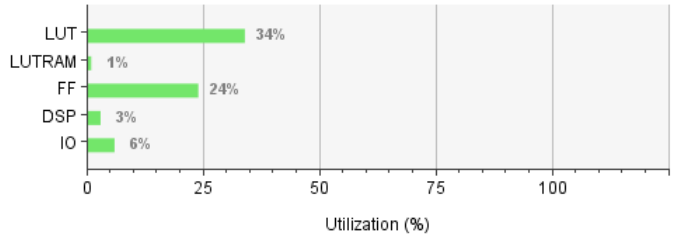


图 24: 流水线资源占用比例

可见，LUT 的占用率最高，达到 34.3%，存储器占据了大量的查找表资源；由于寄存器的大量存在，D 触发器的占用比例也较高，达到 23.50%。相比单周期处理器，LUT,FF 的使用都有所增加，是资源换时间的体现。

Summary

Resource	Utilization	Available	Utilization %
LUT	3461	20800	16.64
FF	8704	41600	20.92
IO	68	250	27.20

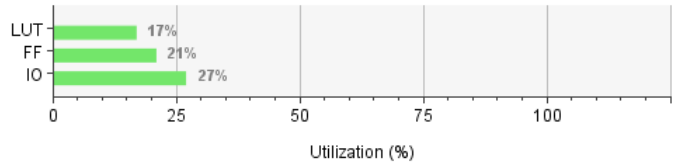


图 25: 单周期处理器资源占用情况

5.3 CPI 估算

利用 C 语言程序将 DataMemory.v 中初始化的正整数列制作为二进制文件 a.in，用于运行汇编代码。测量排序算法部分的指令数：

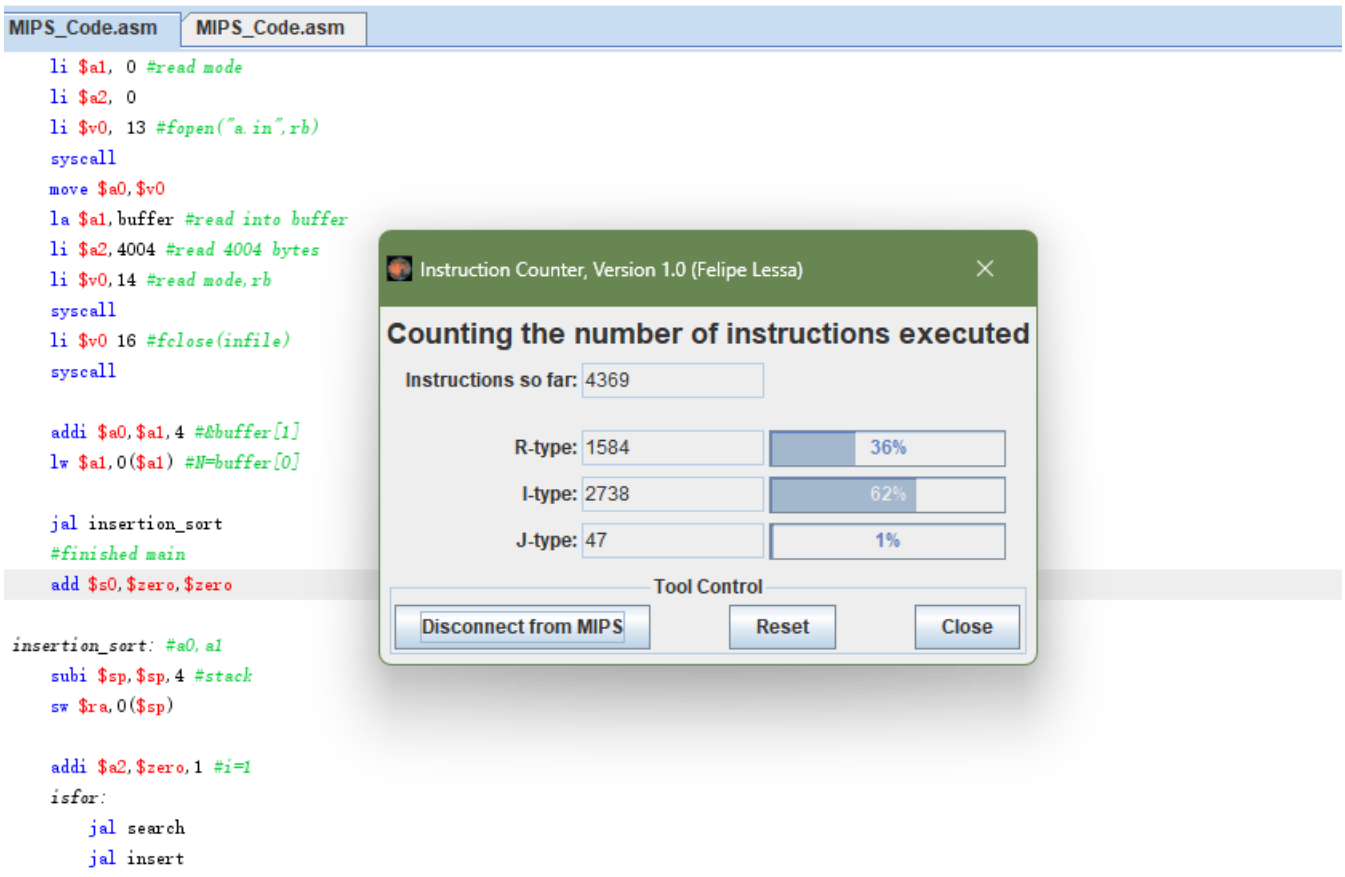


图 26: 排序算法指令数

注意该测试代码的数据输入方式有所修改，经修正，排序算法指令数 $I = 4369 - 12 = 4357$ 。
仿真测量排序算法部分的时钟周期数：

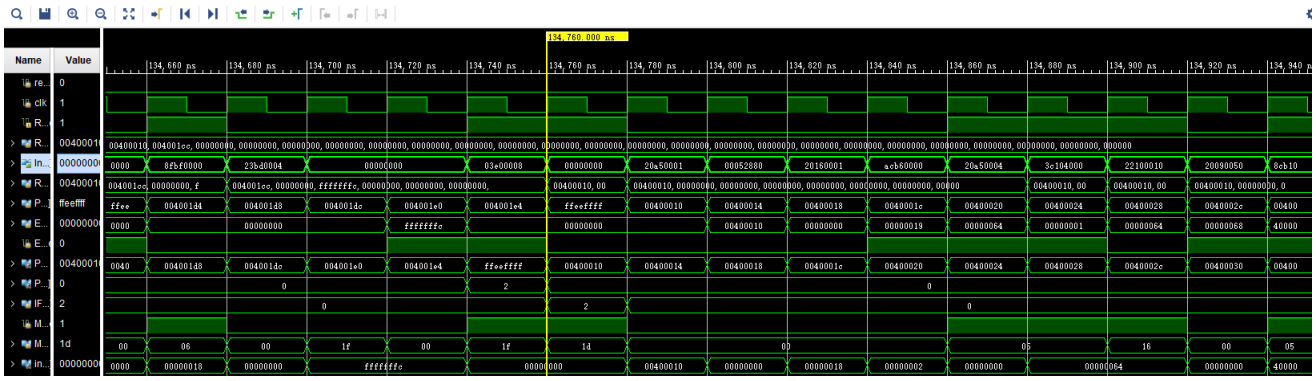


图 27: 排序算法运行周期数

其中指令 0x03e00008 为排序算法部分的最后一条指令。此处已经经历的周期数为

$$C_0 = \frac{134760ns}{20ns} = 6738$$

故排序算法所需周期数为 $C = C_0 + 4 = 6742$。

从而用该流水线处理器执行排序算法的 $CPI = \frac{C}{I} \approx 1.547$。更换测试数据，CPI 基本稳定在该值附近。CPI 大于 1 的主要缘故是 Branch,Jump 及其它冒险处理造成的阻塞。

6 硬件调试情况

在硬件调试（进行 synthesis,implementation,generate bitstream）过程中遇到的主要问题，是代码不规范引发的报错或板上行为异常，在 behavioral simulation 中不会观察到这些问题。因此，仿真验证成功到上板显示成功还有一段路要走。

在 DataMemory.v 的设计中——我曾使用两个 always 块初始化内存，这就产生了多驱动问题，致使 implementation 报错；在 RegisterFile.v 中，我原使用 always@(*) 组合逻辑块，应用阻塞赋值实现寄存器先写后读，但上板发现数码管与指示灯均不亮，代码运行可能出现死循环之类的问题。改用转发方式后，实验板能够正常工作；在使用软件方式扫描显示数位时，我使用了接近 MHz 量级的扫描频率而非原本的 1kHz——一方面，汇编代码书写更简易，另一方面，频率较低时发现数码管有明显的闪烁问题。

此外，在硬件调试过程中还出现了其他问题，如端口位数不匹配等，这些问题并没有在 behavioral simulation 的过程中被发现。我通过处理警告和报错一步步地解决了各种问题，最终完成了硬件调试，验收通过。

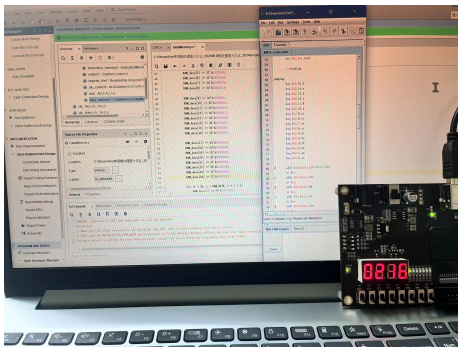


图 28: 硬件调试正常

7 思想体会

本次实验使我收获颇丰，对我有非常深刻的影响。

我充分积累了编写大项目的宝贵经验——步步为营、循序渐进、勤加验证。处理器流水线化的过程，是添加功能的过程。充分考虑应当实现的各种功能，相应地在程序上不断做加法即可。而只有做好规划，在功能实现上讲究一定的次序与逻辑，才能尽量避免细节疏漏，避免千里之堤溃于蚁穴。在编写代码的过程中，我领悟并应用了化整为零、逐个击破的技巧。在处理各种冒险时，我遵循《数字逻辑与处理器基础》讲义，分别处理三类冒险，且同步进行分开验证与合并验证，做到了全方位、无死角。我还认识到了备份及版本管理的重要性，遇到难以解决的 bug 时，可以在邻近旧版本的基础上重新开始，一步步检查问题出现在哪一块新增代码上。频繁排查，遇到问题及时解决，能够避免错误累积以至于积重难返。回想起来，如果没有上述“步步为营”“循序渐进”的习惯，只需要一个 bug，就可以让我迷失在庞大的代码块中，能不能按时完成任务都很成问题。

我充分培养了从事大工程所需要具备的良好品质——耐心谨慎、善于分析。本次流水线工程并非平地起高楼，而是有单周期处理器的基础。这固然大大减轻了我编程的负担，但也考验着我的阅读分析能力。真正领会单周期处理器代码的含义，是对之进行改造的前提。此外，耐心与分析能力是 debug 的必备品质。无论采用何种聪明的编程战略，大小代码块产生问题都是难以避免的。要将汇编代码（包含对应的机器语言）、verilog 代码综合起来，结合仿真波形，不辞辛苦地全面分析。我曾一步步排查了近百个时钟周期，尽管弄得眼睛疲劳、心情烦躁，终于锁定问题所在。解决问题之时，debug 过程所带来的一切折磨都换作了等量的轻松与喜悦。

本次实验全程由我独立完成。如果多加咨询老师助教、与他人多加交流参照，我的效率想必会更高，这是我应当反思改进的。然而，独立实验的难度更大，也更能大大锻炼我解决问题的能力、促使我形成更独立深切的思想体会。在此，我要感谢《数字逻辑与处理器基础实验》课程，在夏季学期为我提供了这样一次全面锻炼的机会！