数字逻辑与处理器基础
流水线处理器

无 81 马啸阳 2018011054

2020 年 9 月 20 日

1 设计方案

1.1 设计功能

- 五级流水线 MIPS 处理器
  - 使用转发电路解决数据冒险
  - Load-use 冒险阻塞一个周期并进行转发
  - 分支指令在 EX 阶段判断，分支发生时取消 IF 和 ID 阶段的两条指令
  - J 类指令在 ID 阶段判断，并取消 IF 阶段的指令
- 支持扩充的分支指令（beq、bne、blez、bgtz、bltz）和跳转指令（j、jal、jr、jalr）
- 支持未定义异常和中断
  - 支持区分内核态与用户态的监督位，内核态下禁止中断
  - 异常或中断发生的时刻，处理器负责记录当前正在执行的指令 PC，跳转到异常或中断处理例程，运行完毕后恢复
- 数据存储地址划分为数据存储器使用地址与外设地址两部分
- 定时器、七段译码管、LED、系统时钟计数器等外设

1.2 总体设计

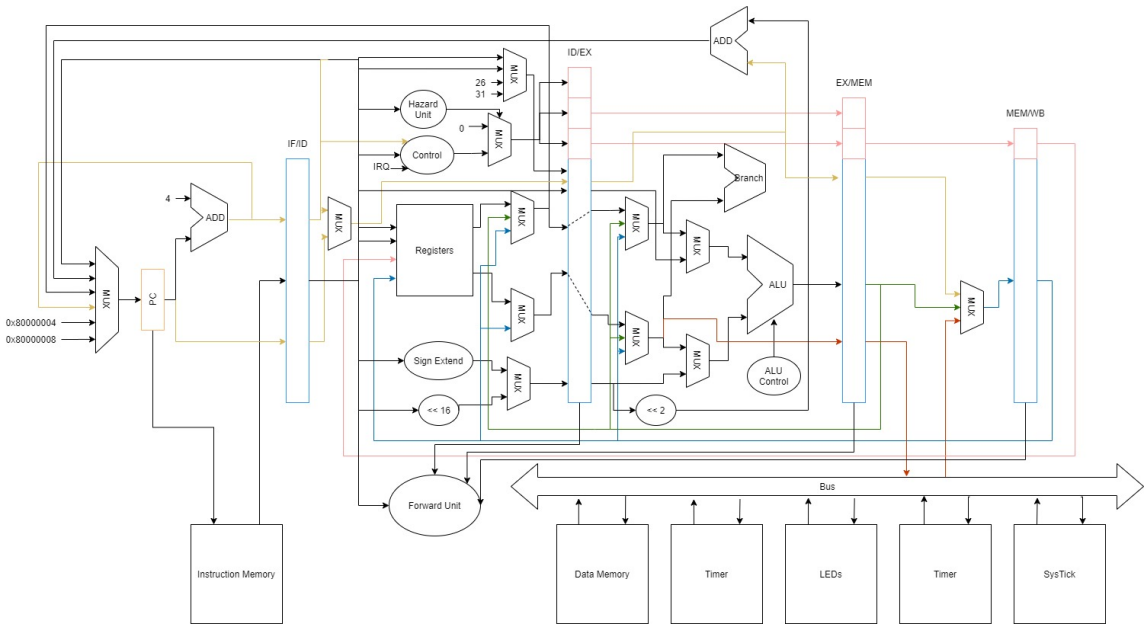


图 1: 流水线处理器设计框图

总体设计如图 1所示，其中主要模块单元与单周期处理器接近，含有程序计数器、指令存储器、寄存器堆、控制单元、ALU、数据存储器等。较单周期处理器增加的内部结构包括：五级流水的级间寄存器、数据转发单元及其通路、冒险检测单元、用以连接外设与数据存储器的总线等。框图中各控制信号与多路选择器的连接略去。特别需要注意的是，这里使用的寄存器堆不支持先写后读。

其中各级流水之间的安排中，分支在 EX 阶段完成，J 型指令在 ID 阶段完成，并且分支单元独立于 ALU 完成以更方便地支持扩展分支指令。对 WB 阶段数据来源的判断提前至 MEM 阶段完成（这是在时序分析后发现 MEM 阶段略有冗余后做出的调整）。具体地，各阶段工作如下。

- IF 阶段：通过 PC 中所存地址取指令，计算 PC+4，同时根据 PCSrc 控制信号更新 PC 中所存的地址。
- ID 阶段：根据指令生成各控制信号，同时读寄存器，判断是否有 load-use 冒险，对立即数进行扩展或左移，并判断是否为 J 型指令。若为 J 型指令，进行跳转并取消 IF 阶段阶段指令。其中读寄存器需要数据转发。

- EX 阶段：生成 ALU 控制信号，进行 ALU 操作（ALU 数据来源需要转发），分支指令进行分支判断，同时计算跳转的地址，分支发生时取消 IF 和 ID 阶段的两条指令。
- MEM 阶段：读写数据存储器或操作外设，随后进行选择需要写入寄存器堆的内容。
- WB 阶段：写寄存器堆

各级间寄存器所存内容及 flush 或 stall 条件如下，详细可见 CPU.v 中代码靠上的各级寄存器定义。

- IF_ID 寄存器：存储当前 PC（由于有中断与 flush 机制，故此 PC 指的是当前实际执行的指令地址，在后文中断设计一节中详细介绍），存储 PC+4 的值用于分支跳转或 jal 与 jalr 指令，存储所取出的指令。
- ID_EX 寄存器：存储 PC+4 的值（实际也可能是当前 PC 以在中断时刻存入 $26 寄存器），存储各控制信号，存储立即数、所读的寄存器值及其地址，存储 ALU 所用的其它控制信号。
- EX_MEM 寄存器：存储 PC+4 的值（同上），存储 ALU 输出、Rd 寄存器地址、RegWrite 信号、Rt 寄存器内容（sw 指令写入的内容）、MemtoReg 信号、MemRead 信号、MemWrite 信号。
- MEM_WB 寄存器：存储 RegWrite 信号、写入寄存器的内容、写入寄存器的地址（Rd 寄存器）。

当发生分支或所读指令为 J 型指令或发生中断或异常时（即 PC 的下一个取值不为 PC+4 时均需 flush），将 IF_ID 寄存器 flush 掉（IF_ID_PC 除外，见后）。当发生分支或需要阻塞时，将 ID_EX 寄存器 flush 掉。后续两级寄存器不会被 flush。阻塞的机制为：当阻塞发生时（由 Hazard 单元检测），PC 不更新，并将 ID_EX 寄存器 flush 掉，此后 IF_ID 寄存器重新读入当前 PC 所产生的内容，达到阻塞目的。具体的相关代码如下：

```
assign IF_ID_Flush = Branch_out || |PCSrc;
assign ID_EX_Flush = Stall || Branch_out;

always @(posedge reset or posedge clk)
        if (reset)
                PC <= 32'h80000000;
        else if (~Stall)
                PC <= PC_next;
```

1.3 数据转发

数据转发通路见图 1中的蓝线与绿线。所需要的数据转发通路，有 ALU 的输入与 jr 与 jalr 所读寄存器输出需要数据转发；同时由于寄存器堆不支持先写后读，因此还需要来自 MEM_WB 寄存器的写入寄存器的转发，转发到 ID 阶段读寄存器内容处。这其中，ID 阶段的转发，Rs 需要转发 EX_MEM 阶段与 MEM_WB 阶段的（因为 jr 与 jalr 指令需要），Rt 仅需转发 MEM_WB 阶段的，因为若等到 EX 阶段这个寄存器已经不在各级流水中了，而 EX_MEM 阶段的与 EX 阶段转发相重复。具体代码如下，需判断写入地址是否与读地址相同，是否写入寄存器，以及是否是 0，同时先判断更近一条指令是否需要转发，就不必再判断更远一条指令是否也转发了。以下代码在 ForwardUnit.v 中。

```
// ForwardA_ID:
// 2'b00 No forward
// 2'b01 EX_MEM_ALU_out
// 2'b10 MEM_WB_Write_data
assign ForwardA_ID = (
        EX_MEM_RegWrite
                        && EX_MEM_RegisterRd != 0
                        && EX_MEM_RegisterRd == RegisterRs
                )? 2'b01: (
                        MEM_WB_RegWrite
                        && MEM_WB_RegisterRd != 0
                        && MEM_WB_RegisterRd == RegisterRs
                )? 2'b10: 2'b00;

// ForwardB_ID:
// 1'b0 No forward
// 1'b1 MEM_WB_Write_data
assign ForwardB_ID = (
                        MEM_WB_RegWrite
                        && MEM_WB_RegisterRd != 0
                        && MEM_WB_RegisterRd == RegisterRt
                )? 1'b1: 1'b0;

// ForwardA_EX:
// 2'b00 No forward
// 2'b01 EX_MEM_ALU_out
// 2'b10 MEM_WB_Write_data
assign ForwardA_EX = (
```

```
                        EX_MEM_RegWrite
                        && EX_MEM_RegisterRd != 0
                        && EX_MEM_RegisterRd == ID_EX_RegisterRs
                )? 2'b01: (
                        MEM_WB_RegWrite
                        && MEM_WB_RegisterRd != 0
                        && MEM_WB_RegisterRd == ID_EX_RegisterRs
                )? 2'b10: 2'b00;

// ForwardB_EX:
// 2'b00 No forward
// 2'b01 EX_MEM_ALU_out
// 2'b10 MEM_WB_Write_data
assign ForwardB_EX = (
                        EX_MEM_RegWrite
                        && EX_MEM_RegisterRd != 0
                        && EX_MEM_RegisterRd == ID_EX_RegisterRt
                )? 2'b01: (
                        MEM_WB_RegWrite
                        && MEM_WB_RegisterRd != 0
                        && MEM_WB_RegisterRd == ID_EX_RegisterRt
                )? 2'b10: 2'b00;
```

1.4 冒险检测

此部分负责检测 Load-use 冒险。若发生，则阻塞一个周期。总体而言 Load-use 冒险是 EX 阶段的指令为 lw 指令（MemRead==1），同时 ID 阶段的指令需要读这个寄存器。但其中有一个特殊情况在于 lw 指令后接一个 jr 或 jalr 指令，并且 lw 读取数据载入了 $ra。这种情况下，由于 jr 或 jalr 指令在 ID 阶段就需要寄存器内容，而不能等到 EX 阶段的数据转发，因此还需要一个周期的阻塞。具体见下代码，此部分代码在 HazardUnit.v 中，其中 EX_Stall 指正常的阻塞，EX 阶段中的指令为 lw，而 MEM_Stall 指 lw 至 jr 的阻塞，MEM 阶段中的指令为 lw。

```
wire EX_Stall, MEM_Stall; // EX_Stall for lw, and MEM_Stall for lw -> jr
assign EX_Stall = ID_EX_MemRead && (ID_EX_RegisterRd != 0) &&
            (ID_EX_RegisterRd == RegisterRs || ID_EX_RegisterRd == RegisterRt);
assign MEM_Stall = EX_MEM_MemRead && (EX_MEM_RegisterRd != 0) &&
            (EX_MEM_RegisterRd == RegisterRs);
```

```
6 assign Stall =  EX_Stall || (PCSrc == 3'b010 && MEM_Stall);
```

1.5 中断与异常

本实验所完成的处理器检测的异常仅为未定义异常，其检测即在控制单元中检测指令的 Funct 与 OpCode 字段是否为支持的指令。而中断由外设触发。

中断与异常主要机制为：在中断或异常发生时，保存当前执行的指令 PC 记录到 $k0 寄存器（这是通过改变当前执行到 ID 阶段的指令为一个写入寄存器 $k0 的指令实现的），并将 PC 置为 0x80000004（中断处理例程地址）或 0x80000008（异常处理例程地址），此处 PC 最高位为 1 为内核态，屏蔽其它中断或异常请求。在例程处理完毕后跳转至 $k0 寄存器恢复原执行的指令，由于 jr 指令由跳转地址决定 PC 监督位，故恢复了原有的内核态或用户态。

中断与异常处理例程本身由编写的汇编码完成，硬件部分的主要难度在于保存实际执行到的 PC 地址，这在 J 型指令与分支指令发生时需要注意。具体地，该地址存储在 IF_ID_PC 中，正常情况下（包括 Stall 的情形）更新为 PC 内的地址，在 ID 阶段的指令是 J 型指令或分支指令或者发生异常或中断时（即所有会 flush IF_ID 寄存器的情形除分支发生外）IF_ID_PC 的值不变（实际是 Latch），而在 EX 阶段的分支指令发生了分支的情况下 IF_ID_PC 更新为 PC_Branch 的值。

具体来看，中断与异常会将当前 ID 指令变更为写入寄存器 $k0 的指令，IF 指令变更为处理例程指令，因此实际执行了的最后一条指令是目前在 EX 中的指令（即上一周期中在 ID 中的指令），IF_ID_PC（正常情况下即 ID 阶段被替代的指令的 PC）是第一条未执行的指令，应写入 $k0。若上一周期 EX 阶段的分支指令发生了跳转（即 Branch_out 为高电平），则下一条本该执行的指令应为 PC_Branch，因而将其放入 IF_ID_PC 以存入 $k0。而若上一周期 ID 阶段的指令为 J 型或分支指令（即目前在 EX 中的指令），并且并未发生上述上一周期 EX 阶段分支发生跳转的情形，则保持 IF_ID_PC 不变使得该指令的地址被存入 $k0，即之后重新执行该条 J 型或分支指令。若发生了异常或中断，由于进入了内核态，则不必更新 IF_ID_PC（这时实际可以任意更新 IF_ID_PC 因为其必然不起作用，但为了简化代码，假定其保持不变）。因此 IF_ID 寄存器的具体更新方法如下所示（以下代码在 CPU.v 中）。

```
1 assign IF_ID_Flush = Branch_out || |PCSrc;
2         always @(posedge reset or posedge clk)
3                 if (reset || IF_ID_Flush) begin
4                         IF_ID_Instruction <= 32'h00000000;
5                         if (reset) begin
6                                 IF_ID_PC <= 32'h00000000;
7                         end else if (Branch_out) begin
8                                 IF_ID_PC <= PC_branch;
```

```
                        end else begin
                                IF_ID_PC <= IF_ID_PC;
                        end
                        IF_ID_PC_plus_4 <= 32'h00000000;
                end else if (~Stall) begin
                        IF_ID_Instruction <= Instruction;
                        IF_ID_PC <= PC;
                        IF_ID_PC_plus_4 <= PC_plus_4;
                end
```

同时，对于 PC 地址的更新，优先级最高的应为中断与异常，然后是成功的分支指令（因为较其它指令领先一个周期），然后是 J 型指令与普通指令，因此 PC 的更新方式代码如下，顺序不可任意调换。

```
assign PC_next = (PCSrc == 3'b100)? 32'h80000004:
                (PCSrc == 3'b101)? 32'h80000008:
                Branch_out? PC_branch:
                (PCSrc == 3'b001)? PC_jump:
                (PCSrc == 3'b010)? PC_jr:
                PC_plus_4;
        always @(posedge reset or posedge clk)
                if (reset)
                        PC <= 32'h80000000;
                else if (~Stall)
                        PC <= PC_next;
```

1.6 外设

将原本处理器中的数据存储器由虚拟总线替代。从处理器角度来看，虚拟总线与原先的数据存储器一样。而在虚拟总线内部会根据指令中的访存地址决定待访问的外设，同时中断信号引回处理器。具体地，虚拟总线代码如下，在 Bus.v 中，分别有数据存储器、定时器、LED、七段数码管、系统时钟计数器，且各外设有自身的使能端。

```
module Bus(reset, clk, Address, Write_data, Read_data, MemRead,
        MemWrite, leds, digits, IRQ);
        input reset, clk;
        input [31:0] Address, Write_data;
        input MemRead, MemWrite;
        output [31:0] Read_data;
```

```
7         output [7:0] leds;
8         output [11:0] digits;
9         output IRQ;
10
11        // Enable signals
12        wire EN_DataMemory;
13        wire EN_Timer;
14        wire EN_LED;
15        wire EN_BCD7;
16        wire EN_SysTick;
17
18        assign EN_DataMemory = Address <= 32'h000007ff;
19        assign EN_Timer = Address >= 32'h40000000 && Address <= 32'h40000008;
20        assign EN_LED = Address == 32'h4000000c;
21        assign EN_BCD7 = Address == 32'h40000010;
22        assign EN_SysTick = Address == 32'h40000014;
23
24        wire [31:0] Read_data_DataMemory, Read_data_Timer, Read_data_SysTick;
25
26        DataMemory data_memory1(.reset(reset), .clk(clk), .Address(Address),
27                .Write_data(Write_data), .Read_data(Read_data_DataMemory),
28                .MemRead(MemRead && EN_DataMemory),
29                .MemWrite(MemWrite && EN_DataMemory));
30
31        Timer timer1(.reset(reset), .clk(clk), .Address(Address[3:2]),
32            .Read_data(Read_data_Timer), .MemRead(MemRead && EN_Timer),
33                .Write_data(Write_data),
34                .MemWrite(MemWrite && EN_Timer), .IRQ(IRQ));
35
36        LED led1(.reset(reset), .clk(clk),
37            .Write_data(Write_data[7:0]),
38            .MemWrite(MemWrite && EN_LED), .leds(leds));
39
40        BCD7 bcd7(.reset(reset), .clk(clk),
41            .Write_data(Write_data[11:0]),
42            .MemWrite(MemWrite && EN_BCD7), .digits(digits));
43
44        SysTick sys_tick1(.reset(reset), .clk(clk), .count(Read_data_SysTick));
45
```

```
46          assign Read_data = MemRead?
47                             (EN_DataMemory? Read_data_DataMemory:
48                          EN_Timer? Read_data_Timer:
49                              EN_LED? {24'h000000, leds}:
50                              EN_BCD7? {20'h00000, digits}:
51                          EN_SysTick? Read_data_SysTick:
52                              32'h00000000):
53                         32'h00000000;
54
55  endmodule
```

各外设代码在 peripherals 文件夹下，具体代码均略去。主要有以下外设：

- 数据存储器：地址范围 0x00000000~0x000007FF，512 个 32 位数，全部可写可读
- 定时器：地址范围 0x40000000~0x4000000B，全部可写可读
  - 0x40000000：TH，当 TL 全 1 后将 TH 装载到 TL
  - 0x40000004：TL，随时钟递增
  - 0x40000008：TCon，定时器控制，0bit 为定时器使能控制，1bit 为定时器中断控制（均为高电平使能），2bit 标记中断，当 TL 全 1 后此位置 1
- 外部 LEDs：地址范围 0x4000000C，可写不可读
- 七段数码管：地址范围 0x40000010，7-0bit 为七段数码管控制信号，11-8bit 使能信号，可写不可读
- 系统时钟计数器：地址范围 0x40000014，复位时开始计数时钟，忽略溢出，可读不可写

1.7 汇编代码

本实验中，对 128 个无符号 32 位二进制数进行排序，排序完毕后将系统时钟计数器所统计的总执行周期数用七段数码管显示。选择的排序方式是快速排序。具体完整代码在 qsort.asm 中。

1.7.1 中断与异常处理例程

异常处理例程为进入死循环，如下。

```
1  exception:
2      j exception
```

中断（由定时器触发）处理的目的在于，刷新七段数码管的数字。不必保存现场（即各寄存器值）因为中断是事实上的最后一段内容。$s3、$s2、$s1、$s0 分别存储着待输出总指令数的十六进制四位，$s6 标记着当前应当刷新的使能位。

整个汇编例程代码的结构为（此时处理器处于内核态，监督位为 1）：首先定时器中断禁止，同时中断状态清零（TCon 的 1-2bit 置零，即 TCon 置 1）；再根据 $s6 的值进行分支，决定存入数码管的使能位；将准备写入的数写入 $t1 后，跳转到 digits 对 $t1 进行查表，当 $t1 与相应的十六进制数相等时，$t2 中放有先前置入的查表得到的七段数码管控制信号，并跳转到 interruptExit，然后写入七段数码管进行刷新显示；然后将 $s6 增 1 并模 4（实际与 3）准备刷新下一个位置；最后将 TCon 置回 3，使能中断，并跳转回 $k0，结束中断例程。具体代码如下。

```
interrupt:
    # Ignore saving registers since showing digits is the last part of program

    # $s7 = 0x40000000
    li $t0, 1
    sw $t0, 8($s7)          # TCon = 1

    beqz $s6, show0
    subi $t0, $s6, 1
    beqz $t0, show1
    subi $t0, $t0, 1
    beqz $t0, show2
    subi $t0, $t0, 1
    beqz $t0, show3

show0:
    li $t0, 0x00000100
    move $t1, $s0
    j digits

show1:
    li $t0, 0x00000200
    move $t1, $s1
    j digits

show2:
    li $t0, 0x00000400
```

```
28      move $t1, $s2
29      j digits
30
31  show3:
32      li $t0, 0x00000800
33      move $t1, $s3
34      j digits
35
36  digits:
37      # show digits with num in $t1 and en in $t0
38      andi $t1, $t1, 0xf
39      addi $t2, $zero, 0x00c0
40      beq  $t1, 0, interruptExit
41      addi $t2, $zero, 0x00f9
42      beq  $t1, 1, interruptExit
43      addi $t2, $zero, 0x00a4
44      beq  $t1, 2, interruptExit
45      addi $t2, $zero, 0x00b0
46      beq  $t1, 3, interruptExit
47      addi $t2, $zero, 0x0099
48      beq  $t1, 4, interruptExit
49      addi $t2, $zero, 0x0092
50      beq  $t1, 5, interruptExit
51      addi $t2, $zero, 0x0082
52      beq  $t1, 6, interruptExit
53      addi $t2, $zero, 0x00f8
54      beq  $t1, 7, interruptExit
55      addi $t2, $zero, 0x0080
56      beq  $t1, 8, interruptExit
57      addi $t2, $zero, 0x0090
58      beq  $t1, 9, interruptExit
59      addi $t2, $zero, 0x0088
60      beq  $t1, 10, interruptExit
61      addi $t2, $zero, 0x0083
62      beq  $t1, 11, interruptExit
63      addi $t2, $zero, 0x00c6
64      beq  $t1, 12, interruptExit
65      addi $t2, $zero, 0x00a1
66      beq  $t1, 13, interruptExit
```

```
67      addi $t2, $zero, 0x0086
68      beq  $t1, 14, interruptExit
69      addi $t2, $zero, 0x008e
70      beq  $t1, 15, interruptExit
71      addi $t2, $zero, 0x00ff
72
73  interruptExit:
74      addi $s6, $s6, 1
75      andi $s6, $s6, 3
76
77      add $t0, $t0, $t2
78      sw $t0, 16($s7)
79      li $t0, 3
80      sw $t0, 8($s7)          # TCon = 3
81      jr $k0
```

1.7.2 主程序

主程序的第一步在于，清除监督位（刚复位时，PC 为 0x80000000，处于内核态），因此需要通过一个 jr 指令，清除监督位。

然后主要部分为快速排序部分，此处不加赘述。在进行快速排序之前，首先将系统时间计数器的值载入 $s7 中。需要注意的是，由于数据存储器地址有限，故需要将 $sp 初始值置一个非零的位置（此处置 1024，防止栈与存储的数据冲突）。

在快速排序完成后，再将系统时间计数器的值载入 $s6 中，作差载入 $s5，并分别将每四位组成的十六进制数依次存入 $s3、$s2、$s1、$s0。

此时拉高 led[0] 以示标记，并将使能的 $s6 置初值 0 以显示最低位。然后将定时器 TL 和 TH 赋值后，将 TCon 置 3 启动定时器。此处设定 TH=0xfffff01，即每 15 周期触发一次中断以节约仿真时间。实际中，TH 应设定更小以限制刷新频率。

在中断例程返回后，进入死循环 j loop，以等待下一次中断。

汇编的完整代码如下。

```
1   .text
2       j main
3       j interrupt
4       j exception
```

```
5
6  main:
7      # clean PC[31]
8      la $ra, userMain
9      jr $ra
10
11 userMain:
12     # load SysTick
13     lui $s7, 0x00004000      # $s7 = 0x40000000
14     lw $s6, 20($s7)
15
16     li $sp, 1024             # prevent $sp grow out of range
17
18     # call quickSort
19     li $a0, 0            # $a0 = 0x00000000 as start address
20     li $a1, 0            # $a1 = 0
21     li $a2, 127          # $a2 = 127
22     jal quickSort
23
24     # load SysTick again, $s7 = 0x40000000
25     lw $s5, 20($s7)
26     sub $s4, $s5, $s6
27
28     # put lower 16 bits of $s4 in $s0, $s1, $s2, $s3
29     li $t0, 0xf
30     and $s0, $t0, $s4
31     srl $s4, $s4, 4
32     and $s1, $t0, $s4
33     srl $s4, $s4, 4
34     and $s2, $t0, $s4
35     srl $s4, $s4, 4
36     and $s3, $t0, $s4
37
38     # use led[0] to show finish
39     li $t0, 1
40     sw $t0, 12($s7)
41
42     li $s6, 0
43
```

```
44      # Timer interrupt
45      subi $t0, $zero, 0x000f
46      sw $t0, 0($s7)          # TH = 0xffffff01
47      subi $t0, $zero, 1
48      sw $t0, 4($s7)          # TL = 0xffffffff
49      li $t0, 3
50      sw $t0, 8($s7)          # TCon = 3
51
52  loop:
53      j loop
54
55  quickSort:
56      # save registers, no need to save $s3, see comments below
57      addi $sp, $sp, -16
58      sw $ra, 12($sp)
59      sw $s2, 8($sp)
60      sw $s1, 4($sp)
61      sw $s0, 0($sp)
62
63      move $s0, $a0       # store arr in $s0
64      move $s1, $a1       # store left in $s1
65      move $s2, $a2       # store right in $s2
66      move $t0, $s1       # $t0 stores i
67      move $t1, $s2       # $t1 stores j
68      sll $t2, $s1, 2
69      add $t2, $s0, $t2   # $t2 = array + 4 * left
70      lw $s3, 0($t2)      # store key in $s3, key only used before
71                          # calling quickSort, so no need to save register
72
73      outerloop:
74          innerloop1:
75              sll $t2, $t1, 2
76              add $t2, $s0, $t2
77              lw $t3, 0($t2)              # $t3 = arr[j]
78              bltu $t3, $s3, innerloop2   # exit if arr[j] < key
79              bge $t0, $t1, innerloop2    # exit if i >= j
80              addi $t1, $t1, -1           # j--
81              j innerloop1
82          innerloop2:
```

```
83              sll $t2, $t0, 2
84              add $t2, $s0, $t2
85              lw $t3, 0($t2)              # $t3 = arr[i]
86              bgtu $t3, $s3, innerexit        # exit if arr[i] > key
87              bge $t0, $t1, innerexit     # exit if i >= j
88              addi $t0, $t0, 1            # i++
89              j innerloop2
90          innerexit:
91          bge $t0, $t1, exit              # break if i >= j
92          sll $t2, $t0, 2
93          add $t2, $s0, $t2               # $t2 = arr + 4 * i
94          sll $t3, $t1, 2
95          add $t3, $s0, $t3               # $t3 = arr + 4 * j
96          lw $t4, 0($t2)                  # $t4 = arr[i]
97          lw $t5, 0($t3)                  # $t5 = arr[j]
98          sw $t4, 0($t3)                  # arr[j] = $t4
99          sw $t5, 0($t2)                  # arr[i] = $t5
100         j outerloop
101
102     exit:
103     sll $t2, $s1, 2
104     add $t2, $s0, $t2                   # $t2 = arr + 4 * left
105     sll $t3, $t0, 2
106     add $t3, $s0, $t3                   # $t3 = arr + 4 * i
107     lw $t4, 0($t3)                      # $t4 = arr[i]
108     sw $t4, 0($t2)                      # arr[left] = $t4
109     sw $s3, 0($t3)                      # arr[i] = key
110
111     addi $t2, $t0, -1                   # $t2 = i - 1
112     bge $s1, $t2, exit1                 # exit if left >= i - 1
113     move $a0, $s0                       # $a0 = arr
114     move $a1, $s1                       # $a1 = left
115     move $a2, $t2                       # $a2 = i - 1
116     jal quickSort
117
118     exit1:
119     addi $t2, $t0, 1                    # $t2 = i + 1
120     bge $t2, $s2, exit2                 # exit if i + 1 >= right
121     move $a0, $s0                       # $a0 = arr
```

```
122     move $a1, $t2                   # $a1 = i + 1
123     move $a2, $s2                   # $a2 = right
124     jal quickSort
125
126     exit2:
127     # restore registers and return
128     lw $ra, 12($sp)
129     lw $s2, 8($sp)
130     lw $s1, 4($sp)
131     lw $s0, 0($sp)
132     addi $sp, $sp, 16
133     jr $ra
134
135 interrupt:
136     # Ignore saving registers since showing digits is the last part of program
137
138     # $s7 = 0x40000000
139     li $t0, 1
140     sw $t0, 8($s7)          # TCon = 1
141
142     beqz $s6, show0
143     subi $t0, $s6, 1
144     beqz $t0, show1
145     subi $t0, $t0, 1
146     beqz $t0, show2
147     subi $t0, $t0, 1
148     beqz $t0, show3
149
150 show0:
151     li $t0, 0x00000100
152     move $t1, $s0
153     j digits
154
155 show1:
156     li $t0, 0x00000200
157     move $t1, $s1
158     j digits
159
160 show2:
```

```
161     li $t0, 0x00000400
162     move $t1, $s2
163     j digits
164 
165 show3:
166     li $t0, 0x00000800
167     move $t1, $s3
168     j digits
169 
170 digits:
171     # show digits with num in $t1 and en in $t0
172     andi $t1, $t1, 0xf
173     addi $t2, $zero, 0x00c0
174     beq  $t1, 0, interruptExit
175     addi $t2, $zero, 0x00f9
176     beq  $t1, 1, interruptExit
177     addi $t2, $zero, 0x00a4
178     beq  $t1, 2, interruptExit
179     addi $t2, $zero, 0x00b0
180     beq  $t1, 3, interruptExit
181     addi $t2, $zero, 0x0099
182     beq  $t1, 4, interruptExit
183     addi $t2, $zero, 0x0092
184     beq  $t1, 5, interruptExit
185     addi $t2, $zero, 0x0082
186     beq  $t1, 6, interruptExit
187     addi $t2, $zero, 0x00f8
188     beq  $t1, 7, interruptExit
189     addi $t2, $zero, 0x0080
190     beq  $t1, 8, interruptExit
191     addi $t2, $zero, 0x0090
192     beq  $t1, 9, interruptExit
193     addi $t2, $zero, 0x0088
194     beq  $t1, 10, interruptExit
195     addi $t2, $zero, 0x0083
196     beq  $t1, 11, interruptExit
197     addi $t2, $zero, 0x00c6
198     beq  $t1, 12, interruptExit
199     addi $t2, $zero, 0x00a1
```

```
200     beq  $t1, 13, interruptExit
201     addi $t2, $zero, 0x0086
202     beq  $t1, 14, interruptExit
203     addi $t2, $zero, 0x008e
204     beq  $t1, 15, interruptExit
205     addi $t2, $zero, 0x00ff
206
207 interruptExit:
208     addi $s6, $s6, 1
209     andi $s6, $s6, 3
210
211     add $t0, $t0, $t2
212     sw $t0, 16($s7)
213     li $t0, 3
214     sw $t0, 8($s7)          # TCon = 3
215     jr $k0
216
217 exception:
218     j exception
```

1.8 测试模块

测试文件执行编写的快速排序及中断汇编代码，可有效测试本处理器的各项功能。转发、冒险、分支、J 型指令的正确运行由快速排序内部本身的代码复杂程度保证。中断的正确性也由汇编代码中的中断处理例程保证，由于中断处理例程执行完毕后进行了返回，并等待下一次中断，同时中断发生时甚至是复杂的 J 型指令执行时，因而由指令的正确跳转（即 $k0 保存地址的正确性）测试了中断处理过程的正确性。

具体地，前仿测试模块 CPU_tb_behav.v 与 DataMemory_behav.v 和 InstructionMemory_behav.v 配套使用，这两个存储器内部是无初始值的而由测试模块赋初值。具体地，由 readmemh 分别从 instruction.txt 和 data.txt 读入指令和未排序的 128 个 32 位无符号整数，然后进行排序。排序完成时，led[0] 拉高，将此时处理器内部的数据由 writememh 写入 data_sorted.txt 验证排序正确性，然后延时 5us，等待中断处理例程操纵七段译码管后结束。具体代码如下。

```
1 `timescale 1ns/1ps
2 `define PERIOD 10
3
4 module CPU_tb;
```

```
reg reset;
reg clk;
wire [7:0] leds;
wire [11:0] digits;

CPU cpu1(reset, clk, leds, digits);

initial begin
    reset = 1;
    clk = 1;
    $readmemh("instruction.txt", cpu1.instruction_memory1.RAM_data);
    $display("Instructions loaded");
    #100 reset = 0;
    $readmemh("data.txt", cpu1.bus1.data_memory1.RAM_data);
    $display("Data loaded");
end

always @(posedge leds[0]) begin
    $writememh("data_sorted.txt", cpu1.bus1.data_memory1.RAM_data);
    #5000 $finish;
end

always #(`PERIOD/2) clk = ~clk;

endmodule
```

而后仿中无法使用 readmemh 直接写入存储器中（存储器模块结构被优化而不再以原本形式存在），采用直接将指令写入 InstructionMemory.v，数据在 DataMemory.v 中 initial 初始化进行综合实现。由于快速排序指令数与待排数据高度相关，且快速排序是递归的（即中途某时刻的数据会影响后续排序的指令数），因此可以通过验证最后七段数码管上显示的总指令数验证排序的正确性。具体测试模块如下。

```
`timescale 1ns/1ps
`define PERIOD 10

module CPU_tb;
reg reset;
reg clk;
wire [7:0] leds;
wire [11:0] digits;
```

```
9
10  CPU cpu1(reset, clk, leds, digits);
11
12  initial begin
13      reset = 1;
14      clk = 1;
15      #100 reset = 0;
16  end
17
18  always @(posedge leds[0]) begin
19      #5000 $finish;
20  end
21
22  always #(`PERIOD/2) clk = ~clk;
23
24  endmodule
```

2 文件清单

```
.
└── code ..... 源代码
    ├── assembly ..... 汇编代码及随机数文件
    │   ├── data.in ..... 随机数十六进制文件
    │   ├── data.txt ..... 随机数文件
    │   ├── data_sorted.txt ..... 快排完毕后的存储器的值
    │   ├── instruction.txt ..... 指令文件
    │   └── qsort.asm ..... 汇编代码
    └── src ..... 流水线处理器代码
        ├── peripherals ..... 外设
        │   ├── BCD7.v ..... 七段数码管
        │   ├── LED.v ..... LED
        │   ├── SysTick.v ..... 系统时钟计数器
        │   └── Timer.v ..... 定时器
        ├── ALU.v ..... ALU
        └── ALUControl.v ..... ALU 控制器
```

```
├── BranchUnit.v ..... 分支控制单元
├── Bus.v ..... 虚拟总线
├── Control.v ..... 控制单元
├── CPU.v ..... 单周期 CPU
├── CPU.xdc ..... CPU 管脚约束
├── CPU_tb.v ..... CPU 测试模块
├── CPU_tb_behav.v ..... CPU 前仿测试模块
├── DataMemory.v ..... 数据存储器
├── DataMemory_behav.v ..... 前仿数据存储器
├── ForwardUnit.v ..... 转发单元
├── HazardUnit.v ..... 冒险单元
├── InstructionMemory.v ..... 指令存储器
├── InstructionMemory_behav.v ..... 前仿指令存储器
└── RegisterFile.v ..... 寄存器堆
```

此处 CPU_tb_behav.v、DataMemory_behav.v 与 InstructionMemory_behav.v 是配套用于前仿的，这两个存储器内部是无初始值的而由 testbench 赋初值。前仿时，用这三个文件替换相应的文件进行仿真。而实现后难以进行类似操作，需使用 CPU_tb.v、DataMemory.v 与 InstructionMemory.v 进行综合实现及测试。

3 综合与实现情况

选择打通层次（flatten-hierarchy 为 full）以提升时序性能，同时对于面积分析附上未打通层次的综合与实现情况以进行参考。

3.1 面积分析

由于打通了层次，因而无法查看各模块的资源使用情况，总体使用情况如图 2所示，总计使用了 6904 个 LUT 和 18320 个寄存器。

Name 1	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)
N CPU	6904	18320	2440	1120	6812	6904	22	1

图 2: 资源占用情况

以未打通层次的处理器综合实现参考，可以观察出各个模块对资源的消耗情况。资源占用

如图 3所示，总计使用了 6704 个 LUT 和 17978 个寄存器。可以看到的是，主要资源都消耗在了数据存储器上与寄存器堆上（数据存储器的 512 个 32 位数就占用了 16384 个寄存器），而其它模块及处理器本身并不消耗过多资源（扣除所有下设模块，处理器本身用了 473 个 LUT 和 463 个寄存器）。

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)
N CPU	6704	17978	2432	1056	6253	6704	22	1
alu1 (ALU)	388	0	0	0	120	388	0	0
alu_control1 (ALUControl)	10	0	0	0	5	10	0	0
branch_unit1 (BranchUnit)	36	0	0	0	14	36	0	0
bus1 (Bus)	5020	16523	2176	1056	5686	5020	0	0
bcd7 (BCD7)	0	24	0	0	7	0	0	0
data_memory1 (DataMemory)	4871	16384	2176	1056	5630	4871	0	0
led1 (LED)	0	16	0	0	5	0	0	0
sys_tick1 (SysTick)	1	32	0	0	8	1	0	0
timer1 (Timer)	78	67	0	0	43	78	0	0
control1 (Control)	26	0	0	0	11	26	0	0
forward_unit1 (ForwardUnit)	22	0	0	0	15	22	0	0
hazard_unit1 (HazardUnit)	10	0	0	0	4	10	0	0
instruction_memory1 (InstructionMemory)	112	0	0	0	31	112	0	0
register_file1 (RegisterFile)	607	992	256	0	443	607	0	0

图 3: 不打通层次的资源占用情况

另外与单周期处理器的资源占用情况进行对比，注意该单周期处理器的数据存储器仅存储 256 个字（是流水线处理器的一半），数据存储器资源占用也大致是一半，其它资源总体而言使用无明显区别，处理器本身扣除下设模块使用了 164 个 LUT 和 38 个寄存器，这一点上较流水线处理器节约较多，因为流水线处理器需要大量的级间寄存器。但是总体而言，由于数据存储器是资源占用的主要部分，单周期处理器与流水线处理器资源占用无明显差异。

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)
N CPU	3882	8710	1287	512	2926	3882	12	1
alu1 (ALU)	446	0	39	0	166	446	0	0
alu_control1 (ALUControl)	11	0	0	0	5	11	0	0
control1 (Control)	18	0	0	0	14	18	0	0
data_memory1 (DataMemory)	2547	8192	1056	512	2780	2547	0	0
instruction_memory1 (InstructionMemory)	38	0	0	0	22	38	0	0
register_file1 (RegisterFile)	658	480	192	0	263	658	0	0

图 4: 单周期处理器的资源占用情况

3.2 时序性能

时序性能如图 5所示，建立时间裕量 0.215ns，可工作的最短周期 10-0.215=9.785ns，最大工作频率 1/9.785ns=102.2MHz。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.215 ns	Worst Hold Slack (WHS):	0.047 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35930	Total Number of Endpoints:	35930	Total Number of Endpoints:	18321

All user specified timing constraints are met.

图 5: 时序性能

关键路径如图 6所示，可以看到这条路径发生在 EX 阶段，其中 ALU 的第二个输入是由 EX_MEM_ALU_out 或 ID_EX_Rt_data 转发而来，并且转发控制信号的产生生成也需要一定时间。

4 仿真情况和分析

首先由前所述，仿真中执行编写的快速排序及中断汇编代码，可有效测试本处理器的各项功能。转发、冒险、分支、J 型指令的正确运行由快速排序内部本身的代码复杂程度保证。中断的正确性也由汇编代码中的中断处理例程保证，由于中断处理例程执行完毕后进行了返回，并等待下一次中断，同时中断发生时甚至是复杂的 J 型指令执行时，因而由指令的正确跳转（即 $k0 保存地址的正确性）测试了中断处理过程的正确性。后仿仅验证执行周期数，由于快速排序指令数与待排数据高度相关，且快速排序是递归的（即中途某时刻的数据会影响后续排序的指令数），因此可以通过验证最后七段数码管上显示的总周期数验证排序的正确性。

行为级仿真由 readmemh 与 writememh 完成，具体波形如图 7所示，led[0] 拉高标志排序完成，此时可从 $s4 中查看系统时钟计数器的差值（即排序指令所用的周期数，此波形中略去）。然后将 $s4 各位依次由七段数码管显示。可以从波形中看到，能够正确循环显示 0x1a1、0x288、0x4c0、0x880，其中最高四位为使能位，依次为 1、2、4、8，而低八位中 0xa1、0x88、0xc0、0x80 分别代表七段数码管上显示 0xd、0xa、0x0、0x8，即显示的总周期数为 0x80ad=32941 个周期。同时，下方所显示的一部分数据存储器中的存储数据可看见排序确实完成了。具体的排序数据可在 data_sorted.txt 中验证排序正确（该文件打印了整个数据存储器，其中包括了快速排序所使用的栈，在数据之后未被清除）。

而对于实现后时序仿真，波形如图 8所示。由前所述，我们只需验证执行的周期数。此处的

Summary

Name	Path 1
Slack	0.215ns
Source	MEM_WB_RegisterRd_reg[3]/C (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	EX_MEM_ALU_out_reg[5]_rep__3/D (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (CLK rise@10.000ns - CLK rise@0.000ns)
Data Path Delay	9.576ns (logic 1.586ns (16.561%) route 7.990ns (83.439%))
Logic Levels	8 (LUT3=1 LUT5=2 LUT6=5)
Clock Path Skew	-0.126ns
Clock Un...rtainty	0.035ns

Source Clock Path

Delay Type	Incr (ns)	Path ...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: W5	clk
net (fo=0)	0.000	0.000		clk
IBUF (Prop_ibuf_I_O)	(r) 1.458	1.458	Site: W5	clk_IBUF_inst/O
net (fo=1, routed)	1.967	3.425		clk_IBUF
BUFG (Prop_bufg_I_O)	(r) 0.096	3.521	Site: BUF...TRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=18320, routed)	1.565	5.086		clk_IBUF_BUFG
FDCE			Site: SLICE_X36Y45	MEM_WB_RegisterRd_reg[3]/C

Data Path

Delay Type	Incr (ns)	Path (...	Location	Netlist Resource(s)
FDCE (Prop_fdce_C_Q)	(r) 0.419	5.505	Site: SLICE_X36Y45	MEM_WB_RegisterRd_reg[3]/Q
net (fo=36, routed)	0.867	6.372		MEM_WB_RegisterRd[3]
LUT6 (Prop_lut6_I0_O)	(r) 0.299	6.671	Site: SLICE_X36Y46	ID_EX_Rt_data[31]_i_10/O
net (fo=5, routed)	1.003	7.673		forward_unit1/p_8_in
LUT6 (Prop_lut6_I0_O)	(r) 0.124	7.797	Site: SLICE_X34Y48	EX_MEM_Rt_data[31]_i_2/O
net (fo=88, routed)	0.892	8.690		EX_RtSrc[1]
LUT5 (Prop_lut5_I1_O)	(r) 0.124	8.814	Site: SLICE_X39Y51	EX_MEM_Rt_data[20]_i_1/O
net (fo=9, routed)	0.311	9.125		EX_MEM_Rt_data[20]_i_1_n_0
LUT3 (Prop_lut3_I2_O)	(r) 0.124	9.249	Site: SLICE_X37Y51	EX_MEM_ALU_out[31]_i_54/O
net (fo=16, routed)	1.400	10.649		ALU_in2[20]
LUT6 (Prop_lut6_I0_O)	(r) 0.124	10.773	Site: SLICE_X37Y57	EX_MEM_ALU_out[12]_i_13/O
net (fo=4, routed)	0.649	11.422		EX_MEM_ALU_out[12]_i_13_n_0
LUT6 (Prop_lut6_I0_O)	(r) 0.124	11.546	Site: SLICE_X37Y58	EX_MEM_ALU_out[6]_i_8/O
net (fo=2, routed)	0.827	12.373		EX_MEM_ALU_out[6]_i_8_n_0
LUT6 (Prop_lut6_I1_O)	(r) 0.124	12.497	Site: SLICE_X36Y59	EX_MEM_ALU_out[5]_i_4/O
net (fo=12, routed)	0.844	13.342		EX_MEM_ALU_out[5]_i_4_n_0
LUT5 (Prop_lut5_I2_O)	(r) 0.124	13.466	Site: SLICE_X33Y59	EX_MEM_ALU_out[5]_rep__3_i_1/O
net (fo=1, routed)	1.197	14.663		EX_MEM_ALU_out[5]_rep__3_i_1_n_0
FDCE			Site: SLICE_X53Y59	EX_MEM_ALU_out_reg[5]_rep__3/D
Arrival Time		14.663		

Destination Clock Path

Delay Type	Incr (ns)	Path (...	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 10.000	10.000		
	(r) 0.000	10.000	Site: W5	clk
net (fo=0)	0.000	10.000		clk
IBUF (Prop_ibuf_I_O)	(r) 1.388	11.388	Site: W5	clk_IBUF_inst/O
net (fo=1, routed)	1.862	13.250		clk_IBUF
BUFG (Prop_bufg_I_O)	(r) 0.091	13.341	Site: BUF...TRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=18320, routed)	1.440	14.781		clk_IBUF_BUFG
FDCE			Site: SLICE_X53Y59	EX_MEM_ALU_out_reg[5]_rep__3/C
clock pessimism	0.180	14.961		
clock uncertainty	-0.035	14.925		
FDCE (Set...dce_C_D)	-0.047	14.878	Site: SLICE_X53Y59	EX_MEM_ALU_out_reg[5]_rep__3
Required Time		14.878		

图 6: 关键路径

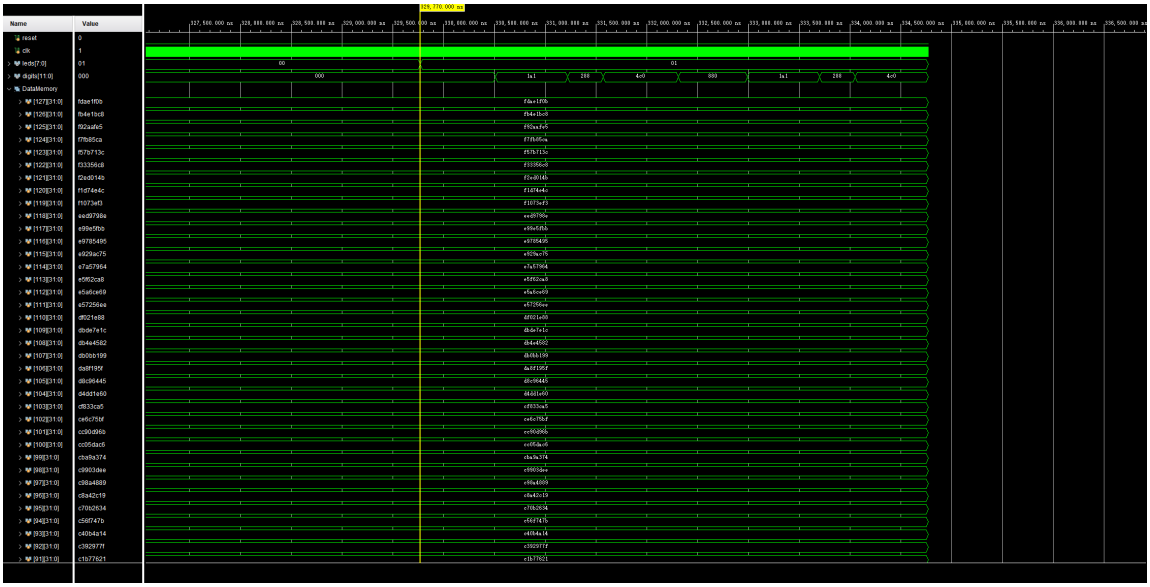


图 7: 行为级仿真波形

特殊情况是，虽然前面所附时序分析表明时序满足要求，但实际的实现后时序仿真中，以 10ns 为周期将无法准确运行。我们将测试模块中的时钟周期改为 20ns，则可正确运行，波形如图所示。波形中可见，在 led[0] 拉高后，BCD 译码管周期性输出 0x1a1、0x288、0x4c0、0x880，并且 led[0] 的拉高时刻是一致的（659540ns，恰为 329770ns 的两倍），因此后仿与前仿结果一致，可认为数据排序正确。

对于后仿的时序要求与时序性能不一致的原因，猜测可能是仿真本身与时序分析流程有所不同，造成一定时序差异。并且，裕量过小也可能是造成此类情况的原因之一。在计算最大工作频率时，以时序报告为准。

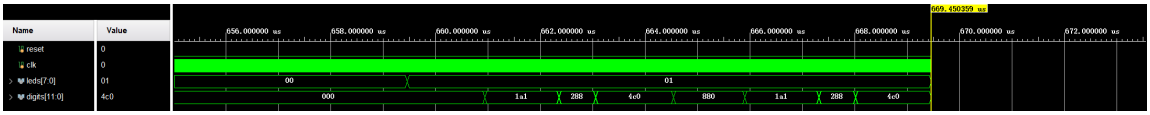
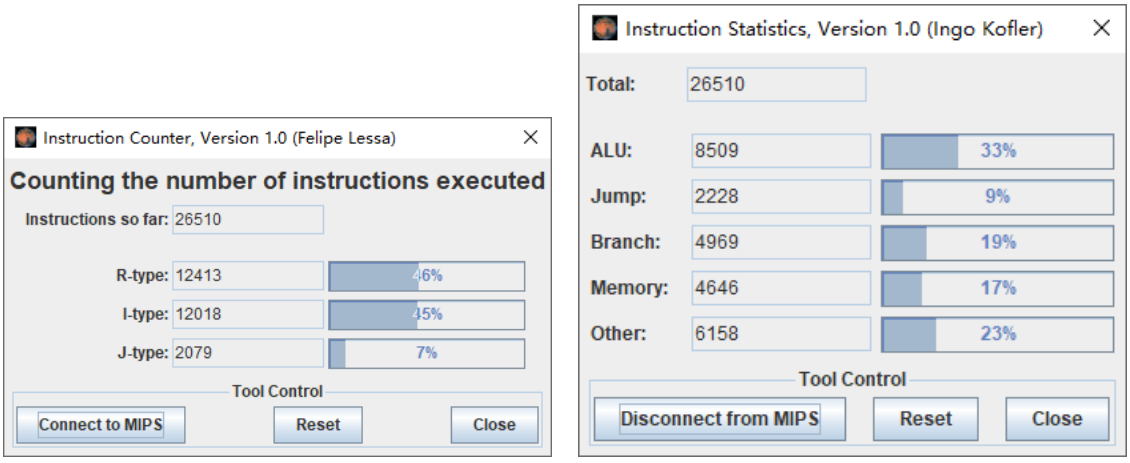


图 8: 实现后时序仿真波形

由图 9所示，使用相同的排序数据，对排序的汇编代码进行统计，共执行了 26510 条汇编指令。因此此处理器在执行这段快速排序代码时 CPI=32941/26510=1.243。这大致符合理论估计：J 型指令占用两个时钟周期，分支指令在跳转成功时占用三个时钟周期，故当假定分支跳转成功率 50% 时，由汇编指令类型统计可得，总 CPI 约为 $0.33+0.09\times2+0.19\times2+0.17+0.23=1.29$。



(a) 汇编指令计数 (b) 汇编指令类型

图 9: 汇编指令统计结果

综上所述，该流水线处理器最大工作频率 102.2MHz，执行这段快速排序汇编代码 CPI 为 1.243，实际最大工作频率 102.2MHz/1.243=82.22MHz（每秒执行约 8200 万条指令）。相比之下，单周期处理器最大工作频率 58.86MHz（每秒执行约 5900 万条指令）。流水线处理器较单周期处理器性能提升了约 39.7%。

5 经验体会

今年由于情况特殊，因此无法实际上板测试，并且时间也较为紧凑，给同学们带来了较大的压力。总体而言，在单周期处理器的基础上，完成一个能够执行基础指令的流水线处理器并不困难，只需按照流水线处理器的原理，正确添加级间寄存器即可完成。这里一些较小的困难是数据转发（由于寄存器不支持先写后读，故与课堂上教授的方案有所区别），以及分支跳转与冒险的处理。

我遇到较多困难的部分是中断与异常的处理。这一部分课堂上讲述得不多，因此需要自己进行自西得思考，完成设计，同时还要编写汇编处理例程来配合硬件完成整个任务，其中有相当多的特殊情况（如分支、J 型指令中出现中断）需要仔细思考，不遗漏任何特殊情况。

总体的调试上，前仿中有比较多的方式进行调试，可以将各模块依次拉出进行观察。主要的问题在于对于流水线处理器，某一时刻的 PC 并不是当前真正执行的指令，甚至可能被取消，需要认真分辨出错的指令执行到哪一个阶段，出现了什么错误，观察相应的级间寄存器。

后仿中出现了一些难以解决的问题，即前述的时序报告完全符合要求，但后仿失败，只能将周期延长而获得正确结构。这出乎我的意料，并且到最后也未能完全解决。

总体而言，本次实验加深了我对于处理器本身原理的认识，并且更充分理解了软硬件协同工作的原理。课堂上讲述的处理器，在实际情形下进行适当调整，实现了一个完整的处理器，使得我获得了较大的乐趣。