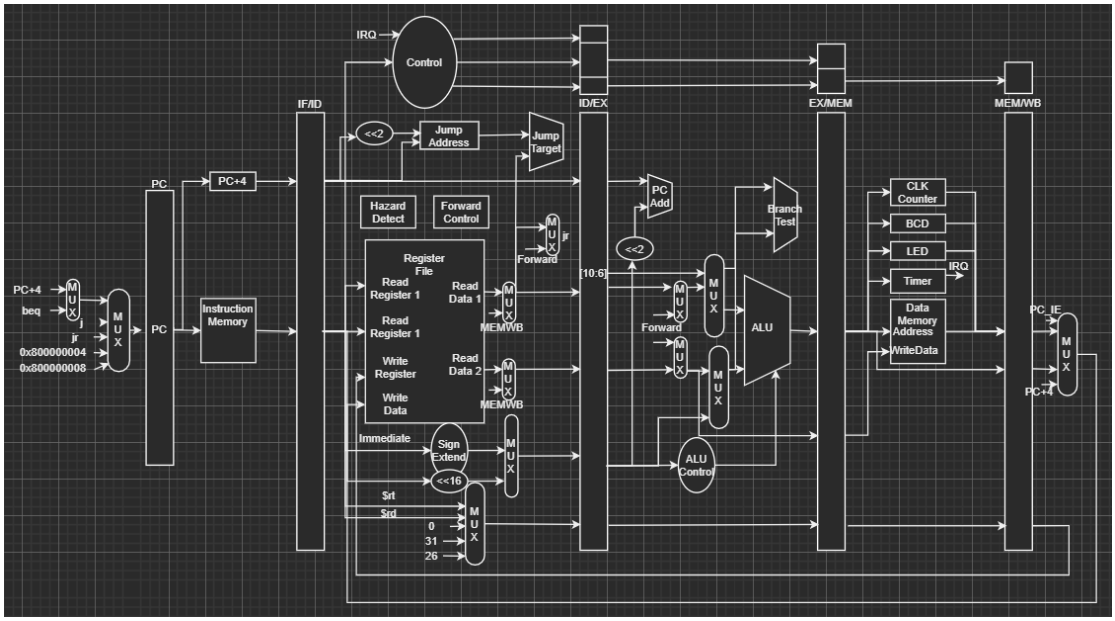
流水线 MIPS 处理器报告

一、 处理器主要性能

1. **107MHz 理论主频**
2. **5 级流水**
3. **支持 30 条 MIPS 指令**
   空指令：nop
   存储访问指令：lw, sw, lui
   算术指令：add, addu, sub, subu, addi, addiu
   逻辑指令：and, or, xor, nor, andi, sll, srl, sra, slt, slti, sltiu
   分支和跳转指令：beq, bne, blez, bgtz, bltz, j, jal, jr, jalr
4. **完整的转发机制**
   5 个转发起始点，5 个转发结束点
   可解决大部分数据冒险
5. **完整的冒险探测机制**
   数据冒险：load-use, load-jr/jalr, load-?-jr/jalr, $-jr/jalr
   控制冒险：j/jr/jal/jalr, branch
   j 类在 ID 阶段判断，branch 类在 EX 阶段判断
6. **支持定时器的软件中断和未定义指令异常**
   有效的 PC 监督位写入控制
   中断处理指令自定义
   异常处理采用死循环
   中断和异常结束后可返回原地址继续执行
7. **支持多个外设，采用 lw 和 sw 访问**
   定时器 Timer：与汇编指令结合实现定时中断
   系统时钟计数器 CLKCounter：记录周期数
   七段数码管 BCD：与汇编指令结合显示周期数
   LED

二、 流水线设计

本流水线处理器的数据通路设计草图如下：



注：大部分的控制信号因过于琐碎未在图中标出。

1. 控制信号

流水线处理器的控制信号在单周期处理器的控制信号基础上修改而来，如下表所示：

	PCSrc	Branch	RegWrite	RegDst	MemRead	MemWrite	MemtoReg	ALUSrc1	ALUSrc2	Extop	Luop
lw	000	000	1	00	1	0	01	0	1	1	0
sw	000	000	0	xx	0	1	xx	0	1	1	0
lui	000	000	1	00	0	0	00	0	1	x	1
add	000	000	1	01	0	0	00	0	0	x	x
addu	000	000	1	01	0	0	00	0	0	x	x
sub	000	000	1	01	0	0	00	0	0	x	x
subu	000	000	1	01	0	0	00	0	0	x	x
addi	000	000	0	00	0	0	00	0	1	1	0
addiu	000	000	0	00	0	0	00	0	1	1	0
and	000	000	1	01	0	0	00	0	0	x	x
or	000	000	1	01	0	0	00	0	0	x	x
xor	000	000	1	01	0	0	00	0	0	x	x
nor	000	000	1	01	0	0	00	0	0	x	x
andi	000	000	0	00	0	0	00	0	1	0	0
sll	000	000	1	01	0	0	00	1	0	x	x
srl	000	000	1	01	0	0	00	1	0	x	x
sra	000	000	1	01	0	0	00	1	0	x	x
slt	000	000	1	01	0	0	00	0	0	x	x
sltu	000	000	1	01	0	0	00	0	0	x	x
slti	000	000	0	00	0	0	00	0	1	1	0
sltiu	000	000	0	00	0	0	00	0	1	1	0

beq	000	001	0	xx	0	0	xx	0	0	1	0
bne	000	010	0	xx	0	0	xx	0	0	1	0
blez	000	011	0	xx	0	0	xx	0	0	1	0
bgtz	000	100	0	xx	0	0	xx	0	0	1	0
bltz	000	101	0	xx	0	0	xx	0	0	1	0
j	001	xxx	0	xx	0	0	xx	x	x	x	x
jal	001	xxxx	1	10	0	0	10	x	x	x	x
jr	010	xxx	0	xx	0	0	xx	x	x	x	x
jalr	010	xxx	1	01	0	0	10	x	x	x	x
中断时刻	011	000	1	11	0	0	11	x	x	x	x
异常时刻	100	000	1	11	0	0	11	x	x	x	x

相比于单周期的控制信号，流水线的控制信号对新添加的指令赋予了相应控制信号，对 PCSrc 和 Branch 进行了增宽，同时为满足中断和异常的需要对 MemtoReg 进行了修改。

2. 中间寄存器

流水线处理器相比于单周期处理器最大的区别在于加入了中间寄存器，实现了数据信号和控制信号的流水线化。

本处理器采用 5 级流水，分为 IF, ID, EX, MEM, WB 这 5 个阶段，同时 PC 也需要由 PC 寄存器保存。因而本处理器设置了 5 个中间寄存器，分别为：PC 寄存器、IFID 寄存器、IDEX 寄存器、EXMEM 寄存器、MEMWB 寄存器。各中间寄存器中存的数据如下：

(1) PC 寄存器

PC

(2) IFID 寄存器

PC, PC_plus_4, Instruction, nop_label

(3) IDEX 寄存器

PC, PC_plus_4, Databus1, Databus2, LUout, shamt, WriteRegister, RegisterRs, RegisterRt, PC_Interrupt, nop_label, PC_IE, ALUForward1, ALUForward2

PCSrc, Branch, ALUSrc1, ALUSrc2, MemRead, MemWrite, RegWrite, MemtoReg, ALUCtl, Sign

(4) EXMEM 寄存器

ALUout, WriteData, WriteRegister, PC_plus_4, PC_Interrupt, nop_label, PC_IE

MemRead, MemWrite, RegWrite, MemtoReg

(5) MEMWB 寄存器

MDR, PC, PC_plus_4, WriteRegister, ALUout, PC_Interrupt, nop_label, PC_IE

RegWrite, MemtoReg

中间寄存器通过 reset 和 clk 进行控制。reset=1 时则把中间寄存器置成 nop 指令，reset=0 时中间寄存器在 clk 上升沿处写入新数据，再把新数据传给后续处理单元，处理完的数据在下一个 clk 上升沿处写入后一级寄存器。开启流水线前需要先 reset，以保证流水线运转起来。

中间寄存器中存储的数据基本与理论课上讲的一致，仅新添加了部分数据。nop_label 和 PC_IE 存储中断跳转前的 PC。

3. PC 写入设计

PC 的设计如下：

```
//PC
assign PC_next = (reset == 1'b1)? 32'h80000000:
                 (BranchResult==1'b1)? BranchTarget: //beq类
                 (PCSrc == 3'b001)? JumpTarget: //j, jal
                 (PCSrc == 3'b010)? JrTarget: //jr, jalr
                 (PCSrc == 3'b011)? 32'h80000004: //中断
                 (PCSrc == 3'b100)? 32'h80000008: //异常
                 PC_plus_4;    //默认PC+4
assign core = PC[31];
always @(posedge reset or posedge clk) //reset为1时复位PC
begin
    if (reset) begin
        PC <= 32'h80000000;
    end
    else begin
        PC <= PC_stall? PC: PC_next;
    end
end
```

每个周期内，PC_next 计算出下一条指令的 PC，在下一个周期的上升沿处写入 PC。

新 PC 主要有以下几种情况：PC_plus_4, branch, j/jal, jr/jalr, 中断，异常。j 类指令在 ID 阶段获得跳转地址，branch 类指令在 EX 阶段判断是否跳转，因而 PCSrc 和 BranchResult 都是在刚获得的周期内被用掉，不需要存在寄存器中。当然这里会出现控制冒险，需要额外处理。

4. 转发设计

在流水线处理器中加入转发以处理数据冒险。转发信号统一在 ForwardControl 模块产生。

转发的起始位点有以下 5 个：EXMEM_ALUout, EXMEM_PC_plus_4, MEMWB_ALUout, MDR, MEMWB_PC_plus_4。转发 PC_plus_4 是针对 jal-use 类语句，jal 在 ID 阶段求出跳转地址，需要 stall 1 个周期，但$ra 的写入 WB 阶段才能完成，因而有必要对 PC_plus_4 进行转发。最差的情况是 jal 后面是 jr $ra（虽然几乎不可能出现），即使 stall 1 个周期 jr 也需要从 EXMEM_PC_plus_4 获得转发。

转发的目的地有以下 5 个：RegisterFile 后的 Databus1, Databus2，jr 指令的 JrTarget，ALU 的 ALUin1, ALUin2。转发到 Databus 是为了实现寄存器的先写后读，由于寄存器堆在 clk 上升沿处读入，而 MEMWB 中间寄存器也是在 clk 上升沿处读入，因而实际写入寄存器堆是在 WB 阶段后面的 clk 上升沿处，故需要转发 MEMWB 阶段的数据：MEMWB_ALUout, MEMWB_MDR, MEMWB_PC_plus_4。

本处理器共有 3 种共 5 个转发信号，如下：

```
output reg [1:0] RFForward1;
output reg [1:0] RFForward2;
output reg [2:0] ALUForward1;
output reg [2:0] ALUForward2;
output reg [2:0] JrForward;
```

RFForward 为向 Databus 的转发，ALUForward 为向 ALUin 的转发，JrForward 为向 JrTarget 的转发。

考虑到 EX 阶段的用时比 ID 阶段大，我将部分转发的判断提前到 ID 阶段，将判断出的转发信号写入 IDEX 中间寄存器。

5. 冒险处理设计

在本指令集中和转发设计下，可能产生冒险的有以下几种语句组合方式：load-use, load-jr/jalr, load-?-jr/jalr, $-jr/jalr, j/jr/jaljalr-, branch-。其中 branch 的判断在 EX 阶段进行，仅在确定要跳后才会触发冒险。将各个冒险按照冒险类型进行分类：

数据冒险 DataHazard	load-use, load-jr/jalr, load-?-jr/jalr
控制冒险 ControlHazard	j/jr/jal/jalr-, branch-

数据冒险中后一条指令不会变，仅需要 stall 1 个周期，因而对于数据冒险需要保持 PC 寄存器不变，保持 IFID 寄存器不变，同时把 IDEX 寄存器 flush 掉。

控制冒险仅在确定会改变 PC 时才会触发，因而提前读入的指令一定不能执行。对于在 ID 阶段判断的 j 类指令，需要把 IFID 寄存器 flush 掉；对于在 EX 阶段判断的 branch 类指令，需要把 IFID 和 IDEX 寄存器全部 flush 掉。

注意若同时出现了数据冒险和控制冒险（如 load-jr）则应先按照数据冒险的处理方式处理（即 stall），当数据冒险被消除后再处理控制冒险。

本处理器的冒险探测统一在 HazardDetect 模块中实现。

```
//DataHazard
wire correlation_1;
wire correlation_2;
assign correlation_1 = (IDEX_RegWrite) && (IDEX_WriteRegister != 5'd0) && (IDEX_WriteRegister == IFID_RegisterRs || IDEX_WriteRegister ==IFID_RegisterRt);
assign correlation_2 = (EXMEM_RegWrite) && (EXMEM_WriteRegister != 5'd0) &&(EXMEM_WriteRegister == IFID_RegisterRs || EXMEM_WriteRegister == IFID_RegisterRt);
wire DataHazard_lw, DataHazard_jr, DataHazard;
assign DataHazard_lw = IDEX_MemRead && correlation_1; //一般load-use
assign DataHazard_jr = (PCSrc == 3'b010) && (correlation_1 || (correlation_2 && EXMEM_MemRead)); //load-jr, load-?-jr, $-jr
assign DataHazard = DataHazard_lw || DataHazard_jr;
//ControlHazard
wire ControlHazard_jump, ControlHazard_branch;
assign ControlHazard_branch = (DataHazard == 1'b0) && BranchResult;
assign ControlHazard_jump = (!ControlHazard_branch) && (DataHazard == 1'b0) && (PCSrc >= 3'b001) && (PCSrc <= 3'b010);
```

判断数据冒险时，先通过寄存器编号判断相邻指令是否可能有数据相关，即 correlation，然后在代入 2 条指令的控制信号判断是否真的构成了冒险。判断控制冒险则较为简单，只需要代入 ID 阶段指令的控制信号。

自此本处理器已经能完成给定指令集下的各类运算，我采用之前单周期求$\sum_{i=0}^{n} i$的指令进行测试，取得了成功，详见后面的仿真测试。

三、 软件中断异常设计

1. PC 监督位

设置 PC[31]作为监督位。当 PC[31]=0 时 CPU 处于用户态，允许中断和异常；当 PC[31]=1 时 CPU 处于内核态，不允许中断和异常。仅部分指令对 PC[31]有着修改权，如下：

仅允许设置 PC[31]=1	reset、中断、异常
仅允许设置 PC[31]=0	jr, jalr
不允许修改 PC[31]	PC+4, j, jal, branch

本处理器在硬件层面上实现了监督位的功能。定义 core=PC[31]内核态/用户态。具体设计方法如下：

对 reset 后的 PC 设置为 0x80000000：

```
always @(posedge reset or posedge clk) //reset为1时复位PC
begin
    if (reset) begin
        PC <= 32'h80000000;
    end
    else begin
        PC <= PC_stall? PC: PC_next;
    end
end
```

中断和异常的 PC 分别设置为 0x80000004 和 0x80000008：

```
assign PC_next = (reset == 1'b1)? 32'h80000000:
                 (BranchResult==1'b1)? BranchTarget: //beq类
                 (PCSrc == 3'b001)? JumpTarget: //j, jal
                 (PCSrc == 3'b010)? JrTarget: //jr, jalr
                 (PCSrc == 3'b011)? 32'h80000004: //中断
                 (PCSrc == 3'b100)? 32'h80000008: //异常
                 PC_plus_4;    //默认PC+4
```

PC+4 的第 31 位始终等于 PC[31]：

```
assign PC_plus_4_temp = (reset == 1'b1)? 32'h80000000: (PC + 32'd4);
assign PC_plus_4 = {PC[31],PC_plus_4_temp[30:0]};
```

2. 中断设计

中断触发信号 IRQ 送入 Control 模块中，当处于处理器用户态时引发中断，Interrupt 置 1。

整体上，发生中断时处理器 EX 及之后的指令照常执行，将 ID 阶段的指令改为将 PC 写入$26，同时将 IF 阶段的指令改为 nop，PCSrc 置 011。这样下一个周期中 IF 阶段读出的是 0x80000004 处的指令。0x80000004 处写的是 j Interrupt，这样之后处理器就会开始执行中断处理指令。

具体实现时，当 ID 阶段探测到 IRQ=1 后，判断能否引发中断并将结果存在 Interrupt 中。若 Interrupt=1，将控制信号替换为将 PC 写入$26 的控制信号，将 IFID_flush 置为 1，将 PCSrc 置为 011。

写入$26 的 PC 由 PC_IE 模块提供，找到的。正常情况下（包括 stall）PC 取 IFID_PC，因为 IFID 中的指令一定紧跟前一条 EX 阶段及之后的指令。若 ID 阶段正好在执行分支跳转

语句 flush 出的 nop 指令，即 IFID 被 flush 过，则依次向后面找第 1 个没有被 flush 过的中间寄存器，取该寄存器中的 PC，这样跳出中断后会先执行这个分支跳转语句。

3. 异常设计

本处理器可处理的异常仅限未定义指令。处理器根据 OpCode 和 Funct 来判断是否在指令集中，若不在指令集中并且处理器处于用户态时，引发异常，Exception 置 1。引发异常后处理器会执行死循环。

硬件层次上的异常处理与中断处理基本一致，仅 PCSrc 上有区别：引发异常后 PCSrc=100。

4. 中断异常的软件设计框架

带有中断异常处理的汇编指令应有如下框架：

```
j Main
j Interrupt
j Exception

Main:
addi $ra $0 20
jr $0 #退出内核态
add $ra $0 $0
#此处写入待排序数据

#此处写入排序

Interrupt: #此处写入中断处理
jr $26

Exception: #此处写入异常处理
jr $26
```

四、 外设设计

1. 概览

处理器的各个外设及其地址设计如下：

地址范围（字节地址）	功能
0x00000000~0x000007FF	数据存储器
0x40000000~0x4000000B	定时器
0x4000000C	外部LEDs
0x40000010	七段数码管
0x40000014	系统时钟计数器

访问外设通过 lw 和 sw 完成。

部分外设取消了读端口或者写端口。定时器 Timer 不具有读端口，内部数据可通过 lw

修改。LED 和七段数码管 BCD 不具有读端口，其输出 led 和 digi 直接连到 CPU 中，这样就不需要通过 lw 才能访问点亮数据。系统时钟计数器 CLKCounter 不具有写端口，仅能通过 reset 置 0。

2. 定时器 Timer

定时器主要在排序完成并进入死循环后引发中断，进入中断处理程序。中断处理程序中写有点亮数码管的指令。每进入 1 次中断就重置定时器，点亮完数码管后处理器重新执行死循环，直到定时器再一次引发中断，进而点亮另 1 个数码管。如此往复实现 4 个数码管的来回点亮。

Timer 的定时功能通过 TH, TL, TCON 实现。3 个信号的设计如下：

地址范围	名称	功能
0x40000000	[31:0] TH	0xffffffff-TH 决定了 2 次中断的周期
0x40000004	[31:0] TL	TL 计数器，每当 TL 全 1，则 TL<=TH
0x40000008	[2:0] TCON	TCON[0]=1 为开始计数；TCON[0]=0 为停止计数 TCON[1]=1 允许 IRQ=1；TCON[1]=0 不允许 IRQ=1 TCON[2]=1 则 IRQ=1；TCON[2]=0 则 IRQ=0

只要 TCON[0]=1，每个 clk 上升沿，TL 就加 1。当 TL 为全 1 时，TL 赋值为 TH，同时若 TCON[1]=1 则 TCON[2]=1，IRQ=1。

实际使用时，先加载 TCON=011，等到 TL 全 1 后引发中断，加载 TCON=001，点亮数码管，然后再加载 TCON=011，跳回死循环，等待下一次中断。因而 Timer 需要保留写端口，不需要保留读端口。

CPU 可正常工作于 100MHz 的时钟下，为保证数码管 1kHz 的刷新率，TH 应设定为 0xfffe795f。实际仿真时，取 TH=0xffffff99

Timer 设计代码如下：

```
always @(posedge reset or posedge clk) begin
    if (reset) begin
        TCON <= 3'b000;
        TH <= 32'h00000000;
        TL <= 32'h00000000;
    end
    else begin
        if (MemWrite) begin
            case(Address)
                32'h40000000: TH <= Write_data;
                32'h40000004: TL <= Write_data;
                32'h40000008: TCON <= Write_data[2:0];
                default: ;
             endcase
        end
        else begin
            if (TCON[0]) begin
                if(TL == 32'hffffffff) begin
                    TL <= TH;
                    if(TCON[1]) begin
                        TCON[2] <= 1'b1; //IRQ enabled
                    end
                end
                else begin
                    TL <= TL + 1'b1;
                end
            end
        end
```

3. 七段数码管 BCD

七段数码管共 12bit，低 8bit 控制数码管点亮的数字，高 4bit 控制点亮哪个数码管。这 12bit 均为低电平点亮。

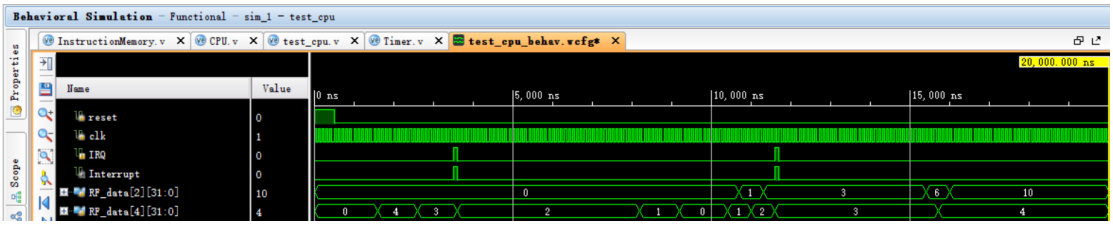
数码管使用十六进制显示总排序数的低 4 位，即[15:0]。各数字的编码方式如下：

数字	编码	数字	编码	数字	编码	数字	编码
0	11000000	4	10011001	8	10000000	c	11000110
1	11111001	5	10010010	9	10010000	d	10100001
2	10100100	6	10000010	a	10001000	e	10000110
3	10110000	7	11111000	b	10000011	f	10001110

五、 流水线测试

本部分测试主要为验证流水线是否能正确处理指令，以及中断异常的硬件部分是否设计正确。

测试代码采用单周期 CPU 的求$\sum_{i=0}^{n} i$代码。此时 IRQ 由 testbench 提供，中断处理指令为空循环一段时间，前仿如下：



注意到处理器检测到了 2 次 IRQ 峰值，成功地进行了中断处理，最终成功地计算出了结果 10。

此外，在写后续代码的时候，mars 多次把 lui 拆分成 lui+ori，而 ori 不在设计的指令集中。仿真时 Exception 会在读入 ori 时拉高，系统进入死循环，我也因此查出了汇编代码的 bug。

由此可知，流水线基本功能均可实现。

六、 排序测试

1. 排序代码设计思路

排序代码整体上可以分为 4 个部分：排序前准备部分、排序部分、点亮前准备部分、中断点亮数码管部分。其中前 3 部分写在 Main 里，第 4 部分写在 Interrupt 里。

(1) 排序前准备部分

排序前准备部分需要完成的任务如下：

reset复位
Timer禁止计数，TCON=000
加载待排序数据
记录CLKCount的输出于$v1

其中使用 lw 向 DataMemory 加载待排序数据。此部分代码较多，我写了 RandomNum.cpp 以自动生成加载随机数的指令。

(2) 排序部分

我采用选择排序，每次排序循环需要从 DataMemory 中读出数据，排好后再把数据写回 DataMemory。

(3) 点亮前准备部分

点亮前准备部分需要完成的任务如下：

记录CLKCount的输出于$v0
CLKCount之差记录在$v0中
将数码管查找表加载进DataMemory

（不断将排好的数lw到$t0中）

Timer加载TH、TL
Timer开始计数，TCON=011
$k1写入0
进入Main死循环

其中“不断将排好的数 lw 到$t0 中”是专门为仿真设计的，这样就可以通过查看$t0 的变化以确定处理器是否完成了排序。实际代码不需要此步。

(4) 中断点亮数码管部分

中断点亮数码管部分需要完成的任务如下：

#点亮数码管1
判断$k1<4
从$k1读出编号并给an赋值
在$v0中找到要显示的数字（使用与运算）
找到要显示的数字对应的地址
读出该数字的编码
将an与数字编码拼接
写入数码管中
$k1+1

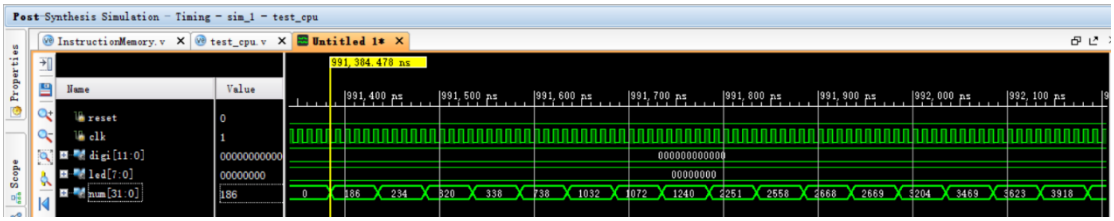
Timer允许中断，TCON=011
$k1=4后重新回到0
跳回Main死循环

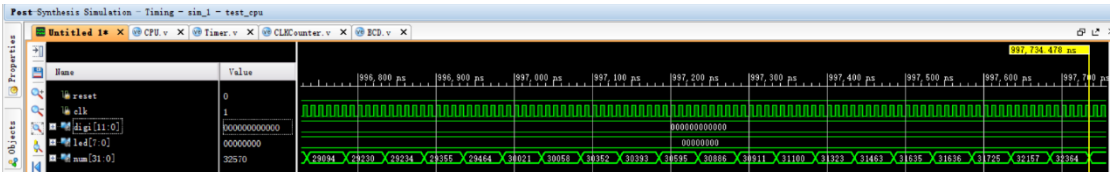
其中$k0 存的是返回 PC，$k1 存的是当前点亮的数码管的编号，an 为 digi 的高 4bit。

2. 128 个数综合后仿

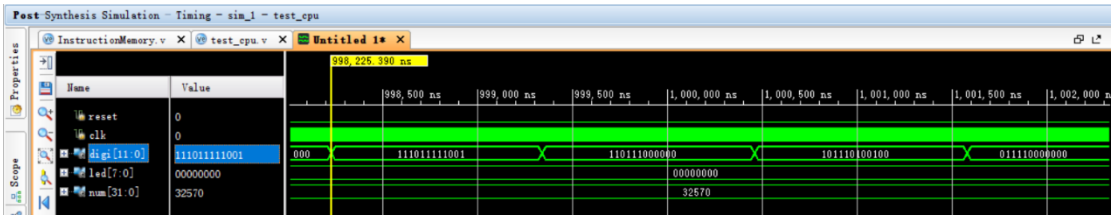
为方便后仿，我将排好的数依次 lw 到$a0 中，然后将$a0 接到顶层模块的 num 中，这样可方便查看是否完成排序。以下 2 个后仿均遵循该设计。

Post-Synthesis Timing Simulation 如下：

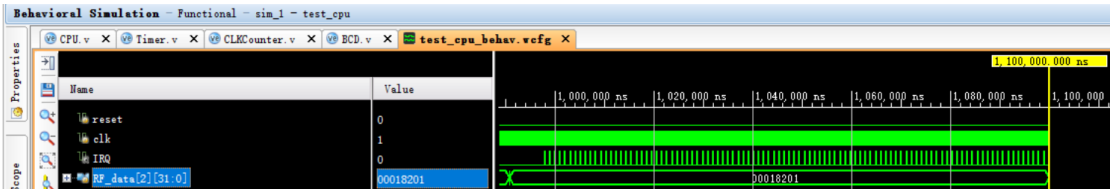




num 为排好序后从 DataMemory 中依次 lw 出的数，可以看到已完成排序。



可以看到 digi 也在周期性地不断刷新，数码管上显示的数为 8201（16 进制）。这是总指令数的低 4 位，实际上总指令数为 0x18201，后仿时管脚不足导致无法查看$v0 中存储的总排序数。前仿验证如下：

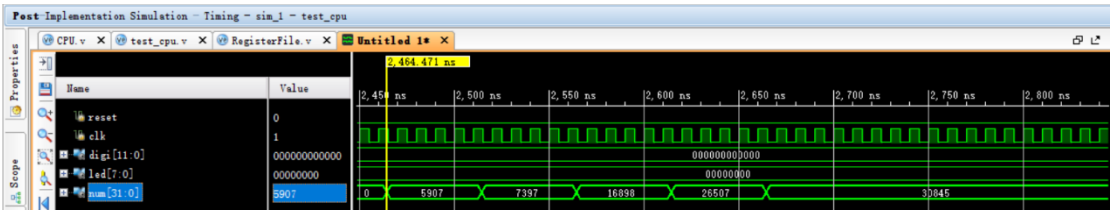


可以看到总排序数为 0x18201。注意 IRQ 出现周期性的峰值，与设计相符。

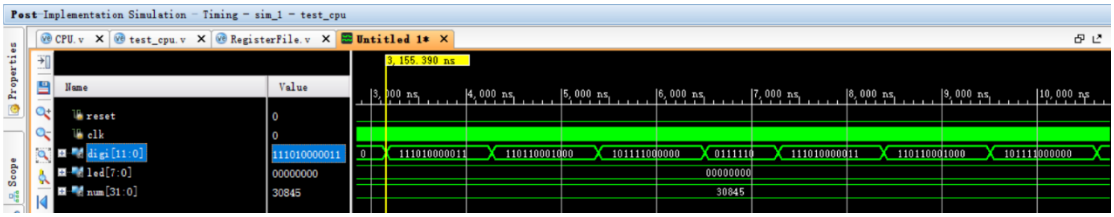
3. 5 个数实现后仿

由于 128 个数实现后仿太费时间，我只采用 5 个数实现后仿以验证排序功能的正确性。

Post-Implementation Timing Simulation 如下：



可以看到处理器正确完成了排序。

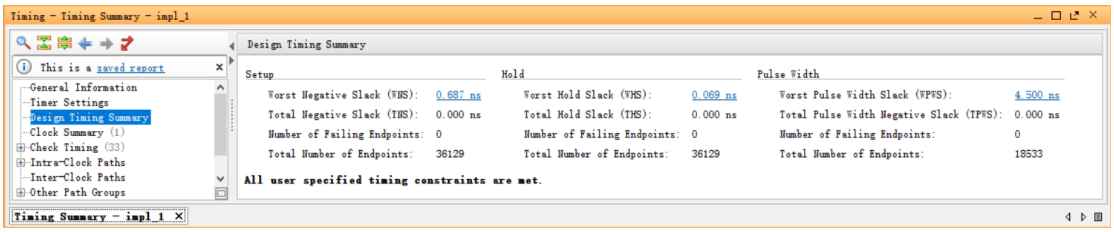


同时 digi 也在周期性地不断刷新数码管点亮数据，显示的数为 00ab（十六进制）。由此可验证处理器设计正确！

七、 处理器性能分析

1. 主频

实现后的时序报告如下：



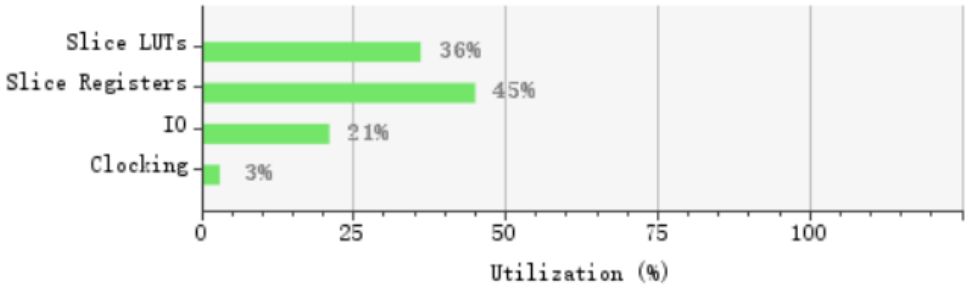
WNS=0.687ns，故理论最高频率为 107.38MHz。

注意到 InstructionMemory 中存的指令数也会影响时序报告。上述报告是在 InstructionMemory 为排 128 个数的情况下产生的。

2. 资源使用情况

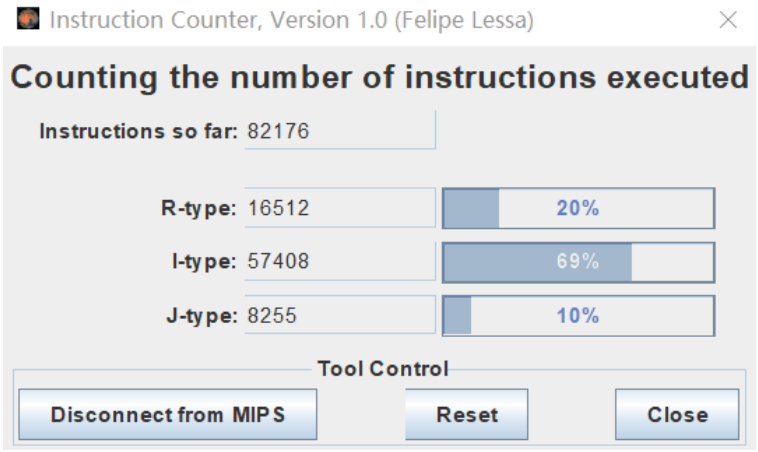
Summary

Resource	Utilization	Available	Utilization %
Slice LUTs	7442	20800	35.78
Slice Registers	18538	41600	44.56
IO	22	106	20.75
Clocking	1	32	3.12



3. CPI

使用 mars 分析 128 个数排序所需指令数，如下：



故总指令数 C=82176。

仿真中得出的总周期数 I=0x18201=82176。

因而 CPI=1.202。

八、 实验心得

本次作业是在大二暑假小学期最后和大三上开学第 1 周写的，整体时间非常紧。同时流水线 CPU 的实现难度大，软件中断的设计复杂，因而我感觉此次大作业难度还是蛮大的，时间限制非常近。我的开学第 1 周几乎全部都投给了 CPU，不过好在最后写完了，效果也还行。

写 CPU 的时候我采取步步为营的策略。首先我花了一天半的时间设计了整个 CPU 的草图，之后我先将 CPU 流水线化，再加入转发，然后写冒险，期间我每写完一步就写一段汇编指令验证，提前找 bug。这样我就写出了一个可运行各种指令的 CPU，我将之前单周期的代码写入流水线 CPU，进一步 debug。接着我开始着手写软件中断和异常，最后写外设。最终仿真时我先从排 5 个数试起，最终扩展到 128 个数。这一步步走来虽然工作量很麻烦，却最大程度地减小了出 bug 的可能性，我最后仿真时遇到的 bug 几乎全是汇编指令的 bug，如果硬件上还有 bug 那 de 起来会相当麻烦和烦人。

另一个重要的收获是版本控制。我每实现一个大模块就保留一次副本，并专门记录这些副本实现的 feature。这些副本发挥了很大的作用，尤其是在最后优化时序性能的时候。

最后在此感谢老师和助教的努力和付出！

九、 文件清单

1. Verilog

ALU: ALU 模块

ALUControl: ALUControl 模块

BCD: 七段数码管模块

BranchTest: 判断各个 branch 指令是否跳转

CLKCounter: 系统时钟计数器模块

Control: Control 模块

CPU: 设计的顶层模块

DataMemory: 数据存储器模块

ForwardControl: 转发模块

HazardDetect: 冒险处理模块
InstructionMemory: 指令存储器模块
LED: LED 模块
PcforIE: 给出中断异常后返回地址
RegisterFile: 寄存器堆模块
test_cpu: 仿真测试模块，给出 reset 和 clk
Timer: 定时器模块

2. XDC

CPU.xdc: 约束文件

3. 汇编指令

mips_pip_comp_5: 对 5 个数的排序，符合软件中断规范
mips_pip_comp_128: 对 128 个数的排序，符合软件中断规范
mips_pip_comp_single: 求$\sum_{i=0}^{n} i$，符合软件中断规范

4. 辅助代码

InstructionCompilation: 将机器码转化为可被 Verilog 读取的指令
RandomNum: 产生随机数，并转化为 mips 指令