数字逻辑预处理器基础实验报告书
流水线 MIPS 处理器的设计

姓名：黎佳维

学号：2022010540

一、实验目的

将理论课处理器大作业中设计的单周期 MIPS 处理器改进为流水线结构，并利用此处理器完成排序算法。将理论课学习的内容以实践的形式熟悉与深化，并增强同学们 verilog 代码与汇编语言的读写与纠错能力。

二、设计方案（原理说明及框图）

处理器基于理论课程学习的框架进行设计开发，整体采用了冯诺依曼存储与计算分离的架构，分为五级流水线，有 IF-ID-EX-MEM-WB 五个阶段。处理器 Main.v 从指令存储器 InstructionMemory 中读指令，数据存储器 DataMemory 读写数据，进行交互并对数据进行处理。执行的程序以机器码的形式提前写入指令存储器中，要排序的数据写入在 DataMemory 中；排序算法采用了插入排序，从小到大进行排序。

在细节上，有如下设计：

(1) 在处理冒险的问题中，采用了如下方案：

结构冒险：采用了数据存储器与指令存储器分离的哈佛架构；

数据冒险：在硬件上实现了寄存器堆的先写后读；采用了 forwarding 电路解决数据关联问题：对于 Load-use 类竞争，采取了 stall 一个周期+Forwarding 的方法解决；

控制冒险：实现了提前判断，分支指令与跳转指令都提前到 ID 阶段进行判断。

(2) 对分支指令与跳转指令进行了如下扩充：

分支指令：beq、bne、blez、bgtz、bltz；

跳转指令：j、jal、jr、jalr。

(3) 地址空间的划分：

0x00000000～0x3FFFFFFF：数据 RAM，具有数据存储功能；

0x40000000～0x7FFFFFFF：外设地址空间。

(4) 算法指令可见附件 InstructionMemory 中，主要分为两个部分，前半部分为插入排序算法，直接翻译自理论课上给出的 C++代码，分为 main、insertion_sort、serach、insert 四个函数模块，依次找到当前数可以插入的位置后进行逐个插入；后半部分为数码管显示功能，将排序好的数字按顺序显示到数码管上，并循环显示。

(5) 指令与控制信号设计：

处理器一共实现了 33=16 条 R 型+15 条 I 型+2 条 J 型指令。控制信号共有 8 个，如下图所示，其中标红是为了设计时方便进行特例的赋值，以此简化选择语句的设计。

				OP(HEX)	FUN(HEX)	MemWrite	MemRead	RegWrite	Regread1	Regread2	MemtoReg	Branchtype	PCSrc[1:0]
addu	rd=rs+rt	R		0	21	0	0	1	1	1	0	0	0
or	rd=rs|rt	R		0	25	0	0	1	1	1	0	0	0
xor	rd=rs xor rt	R		0	26	0	0	1	1	1	0	0	0
sll	rd=rt<<shamt(logic)	R		0	0	0	0	1	0	1	0	0	0
srav	rd=rt>>rs(arithmetic)	R		0	7	0	0	1	1	1	0	0	0
srl	rd=rt>>shamt(logic)	R		0	2	0	0	1	0	1	0	0	0
and	rd=rs&rt	R		0	24	0	0	1	1	1	0	0	0
jr	PC=rs	R		0	8	0	0	0	1	0	x	x	3
add	rd=rs+rt	R		0	20	0	0	1	1	1	0	0	0
sub	rd=rs-rt	R		0	22	0	0	1	1	1	0	0	0
subu	rd=rs-rt	R		0	23	0	0	1	1	1	0	0	0
nor	rd=rs nor rt	R		0	27	0	0	1	1	1	0	0	0
sra	rd=rt >>shamt(arithmetic)	R		0	3	0	0	1	0	1	0	0	0
slt	if (rs<rt) rd=1 else rd=0	R		0	2a	0	0	1	1	1	0	0	0
sltu	if (rs<rt) rd=1 else rd=0	R		0	2b	0	0	1	1	1	0	0	0
jalr	PC=rs; r31=PC+8	R		0	9	0	0	1	1	0	0	0	3
lui	rt={imm,16'b0}	I		f		0	0	1	0	0	0	0	0
ori	rt=rs|ZeroExtImm	I		d		0	0	1	1	0	0	0	0
lw	rt=Mem[rs+SignExtImm]	I		23		0	1	1	1	0	1	0	0
sw	Mem[rs+SignExtImm]=rt	I		2b		1	0	0	1	1	x	0	0
andi	rt=rs&ZeroExtImm	I		c		0	0	1	1	0	0	0	0
addiu	rt=rs+SignExtImm	I		9		0	0	1	1	0	0	0	0
addi	rt=rs+SignExtImm	I		8		0	0	1	1	0	0	0	0
xori	rt=rs xor ZeroExtImm	I		e		0	0	1	1	1	0	0	0
bne	if(rs!=rt) PC=PC+4+BrAddr	I		5		0	0	0	1	1	x	1	1
beq	if(rs==rt) PC=PC+4+BrAddr	I		4		0	0	0	1	1	x	2	1
blez	if(rs<=0) PC=PC+4+BrAddr	I		6		0	0	0	1	0	x	3	1
bgtz	if(rs>0) PC=PC+4+BrAddr	I		7		0	0	0	1	0	x	4	1
bltz	if(rs<0) PC=PC+4+BrAddr	I		1		0	0	0	1	0	x	5	1
slti	if (rs<imm) rd=1 else rd=0	I		a		0	0	1	1	0	0	0	0
sltiu	if (rs<imm) rd=1 else rd=0	I		b		0	0	1	1	0	0	0	0
j	PC=JumpAddr	J		2		0	0	0	0	0	x	0	2
jal	PC=JumpAddr; r31=PC+8	J		3		0	0	1	0	0	0	0	2

①memwrite: 0: write disable 1: write enable

②memread: 0:read disable 1:read enable

③regwrite: 0:write disable 1:write enable

④regread1: 0:read disable 1:read enable

⑤regread2: 0:read disable 1:read enable

⑥memtoreg: 0:from ALU result 1:from MEM x:memwrite=0

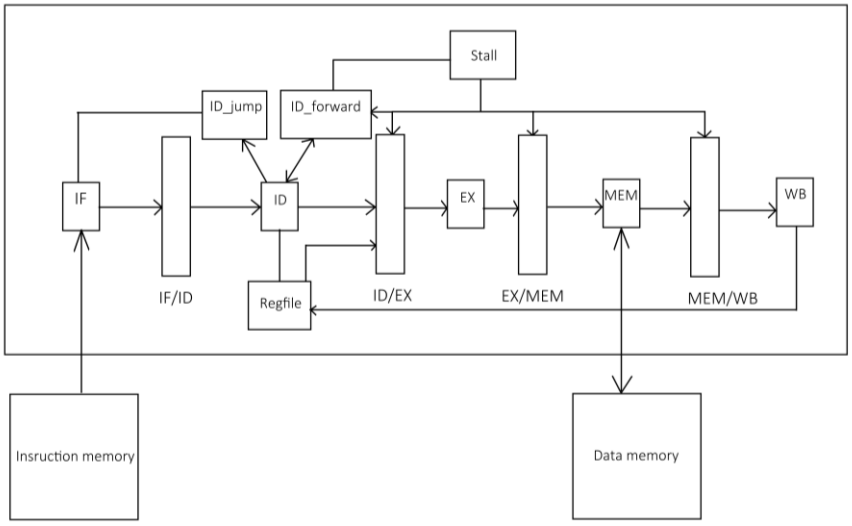
⑦branchtype: 1:bne 2:beq 3:blez 4:bgtz 5:bltz

⑧pcsrc: 0:PC+4 1:PC+4+BrAddr 2:JumpAddr 3:rs

⑨aluop: 决定了后续 ex 阶段中需要执行的操作。

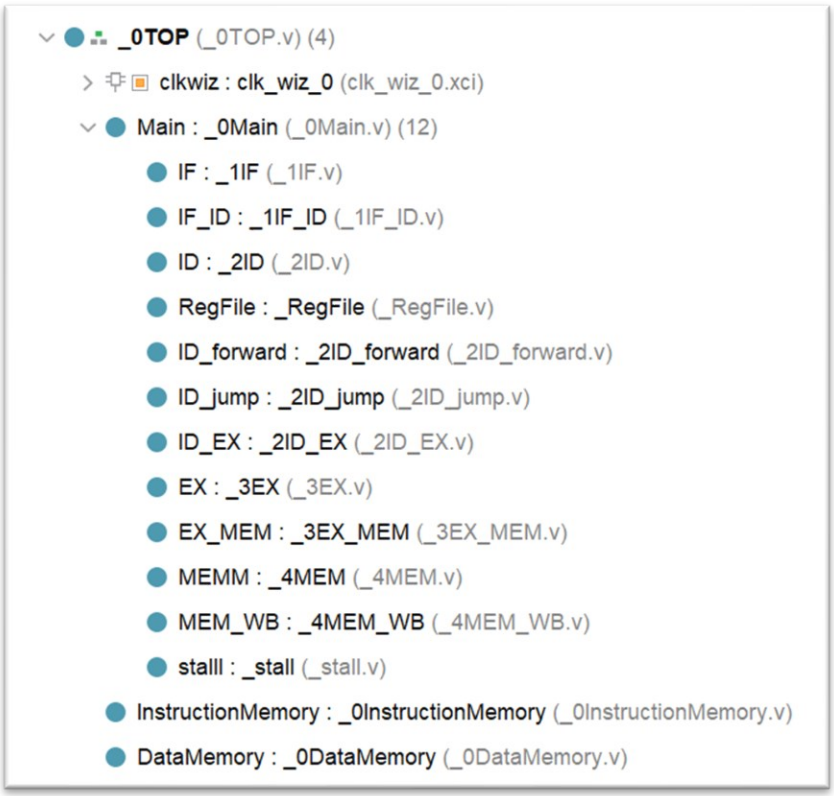
此外，图中还附带了各指令的内容阐释与识别 code，设计与标准 mips 指令设计相一致。

设计的整体框图图下所示，各部分对应文件清单中的各组成，其详细功能在关键代码中阐释：



三、关键代码及文件清单

文件清单(此外还有约束文件_0.xdc、测试模块 tb.v、自己设计的指令模式生成代码 generate.py)：



0 开头的部分文件，为该处理器的顶层模块：

TOP 实例化处理器模块（Main）与存储模块（DataMemory、InstructionMemory），是处理器的顶层模块，连接了计算与存储功能，同时输出控制信号到数码管上进行显示。

Main 实例化了处理器中的五个阶段及其附带的模块，实现了五级流水线各个阶段的处理功能，以及附带的数据转发、中断模块。

InstructionMem 部分根据所给的 pc 进行指令读取，DataMem 部分根据所给的地址对数据进行读写。

关键代码：

1、1IF.v：

该模块控制 pc 取值，实现读取下一条指令或者到达跳转位置，rst 时从初始位置读取。

```
always @(posedge clk) begin
    if(rst) begin
        pc <= 32'h00400000; //reset to default address
    end
    else if (stall != 1) begin
        if(jumpif) begin
```

```
            pc <= jumpaddr;
        end
        else begin
            pc <= pc + 4'h4;
        end
    end
end
```

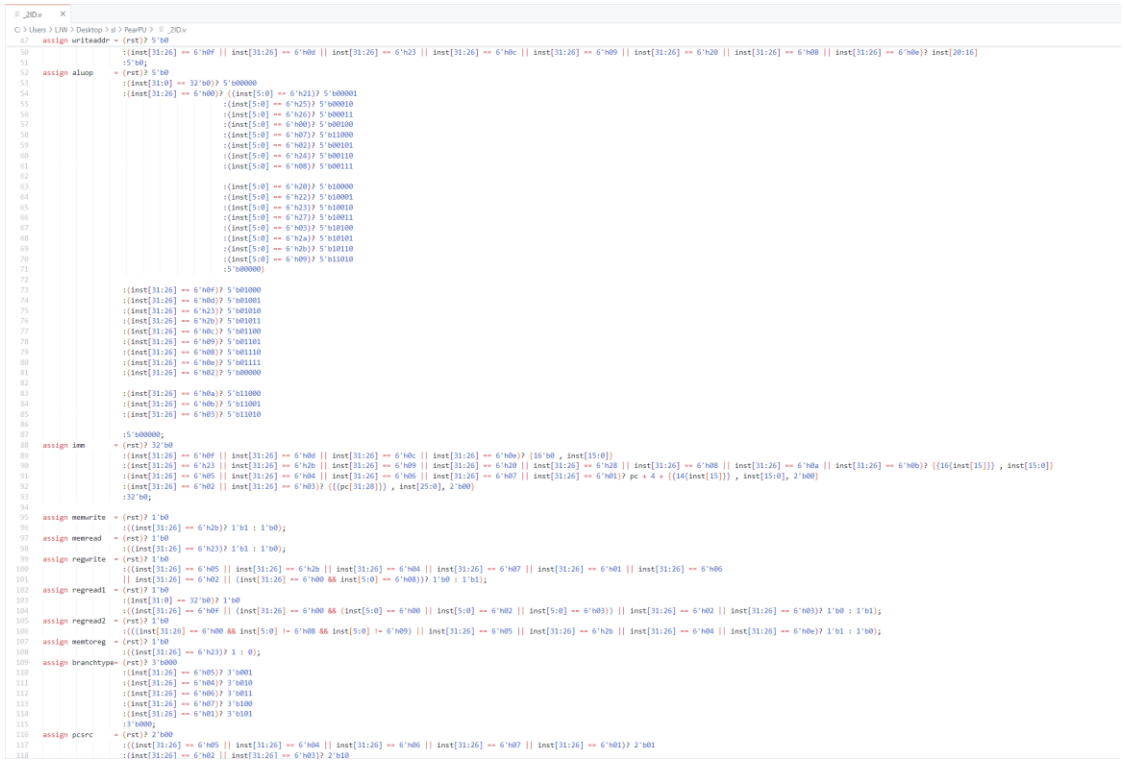
2、1IF_ID.v：

与该模块类似名为 A_B 的模块，即为处理器中各个流水线级间的数据寄存器，存储上一级数据并作为下一级的输出，同时还受到 rst 或者 stall 信号的控制重置或者保持不变。

```
always @(posedge clk) begin
    if(rst) begin
        ID_pc <= 32'h80000000;
        ID_inst <= 1'b0;
    end
    else if (stall!=1) begin
        ID_pc <= IF_pc;
        ID_inst <= IF_inst;
    end
end
```

3、2ID.v：

此模块为设计中最大的模块之一，实现了指令到控制信号的解码，主要采用选择语句的形式进行译码，解码的控制信号在后续的模块中决定了操作的类型以及目标，各个控制信号的具体含义可见上文。



4、2ID_forward.v：

此模块实现了数据的转发，转发的原则是：如果当前执行的指令所需要的寄存器号码与前两条指令修改的寄存器号码任意一条一致，则进行转发，同时更靠近的一条（即前一条）具有优先转发权（因为更靠近的指令所得到的值才是接下来要使用的）。

此外，模块还实现了对于 load-use 冒险的检测，如果指令为 load 且读与用的指令是同一条，则流水线 stall 一个周期。

```
assign stall = isload && ((EX_MEM_regwrite && ID_regread1 &&
EX_MEM_regwriteaddr == ID_regreadaddr1) || (EX_MEM_regwrite &&
ID_regread2 && EX_MEM_regwriteaddr == ID_regreadaddr2));
assign ID_regreaddata1 = rst? 31'b0
                       :(EX_MEM_regwrite && ID_regread1 &&
EX_MEM_regwriteaddr == ID_regreadaddr1) ? EX_MEM_regwritedata
                       :(MEM_WB_regwrite && ID_regread1 &&
MEM_WB_regwriteaddr == ID_regreadaddr1) ? ((MEM_WB_memtoreg)?
MEM_WB_memreaddata: MEM_WB_aluresult)
                       :ID_regreaddata10;
assign ID_regreaddata2 = rst? 31'b0
                       :(EX_MEM_regwrite && ID_regread2 &&
EX_MEM_regwriteaddr == ID_regreadaddr2) ? EX_MEM_regwritedata
                       :(MEM_WB_regwrite && ID_regread2 &&
MEM_WB_regwriteaddr == ID_regreadaddr2) ? ((MEM_WB_memtoreg)?
MEM_WB_memreaddata: MEM_WB_aluresult)
                       :ID_regreaddata20;
```

5、2ID_jump.v：

此模块实现了分支指令与跳转指令的提前判断，若根据 branchtype 的类型对应的判断方法为正确，或检测到 j 型指令，则分支或跳转成立，导入到 if 模块中根据 jumpaddress 修改 pc 的值。

```
assign jumpif = rst? 1'b0
              :(pcsrc == 2'b01)?
              (((branchtype == 3'b001 && readdata1 != readdata2) ||
              (branchtype == 3'b010 && readdata1 == readdata2) ||
              (branchtype == 3'b011 && readdata1 <= 0) ||
              (branchtype == 3'b100 && readdata1 > 0) ||
              (branchtype == 3'b101 && readdata1 < 0))? 1'b1: 1'b0)
              :((pcsrc == 2'b10 || pcsrc == 2'b11)? 1'b1: 1'b0);
assign jumpaddr = (pcsrc == 2'b01)?
              (((branchtype == 3'b001 && readdata1 != readdata2) ||
              (branchtype == 3'b010 && readdata1 == readdata2) ||
              (branchtype == 3'b011 && readdata1 <= 0) ||
```

```
                (branchtype == 3'b100 && readdata1 > 0) ||
                (branchtype == 3'b101 && readdata1 < 0))? (imm): 32'b0)
                :((pcsrc == 2'b10)? (imm): (pcsrc == 2'b11)? (readdata1):
32'b0);
```

5、3EX.v：

此模块根据各个指令不同的 opcode 计算得到了所需要的值输出到 aluresult 中。

```
assign aluresult = rst? 31'b0
                :(aluop == 5'b00001)? readdata1 + readdata2
                :(aluop == 5'b00010)? readdata1 | readdata2
                :(aluop == 5'b00011)? readdata1 ^ readdata2
                :(aluop == 5'b00100)? readdata2 << shamt
                :(aluop == 5'b11000)? readdata2 >>> readdata1
                :(aluop == 5'b00101)? readdata2 >> shamt
                :(aluop == 5'b00110)? readdata1 & readdata2
                :(aluop == 5'b00111)? readdata1
                :(aluop == 5'b01000)? {imm[15:0] , 16'b0}
                :(aluop == 5'b01001)? readdata1 | imm
                :(aluop == 5'b01010)? readdata1 + imm
                :(aluop == 5'b01011)? readdata1 + imm
                :(aluop == 5'b01100)? readdata1 & imm
                :(aluop == 5'b01101)? readdata1 + imm
                :(aluop == 5'b01110)? readdata1 + imm
                :(aluop == 5'b01111)? readdata1 ^ imm
                :(aluop == 5'b10000)? readdata1 + readdata2
                :(aluop == 5'b10001)? readdata1 - readdata2
                :(aluop == 5'b10010)? readdata1 - readdata2
                :(aluop == 5'b10011)? ~(readdata1 | readdata2)
                :(aluop == 5'b10100)? readdata2 >>> shamt
                :(aluop == 5'b10101)? readdata1 < readdata2
                :(aluop == 5'b10110)? readdata1 < readdata2
                :(aluop == 5'b10111)? readdata1
                :(aluop == 5'b11010)? pc + 5'd8
                :(aluop == 5'b11000)? readdata1 < imm
                :(aluop == 5'b11001)? readdata1 < imm
                :31'b0;
```

6、4MEM.v：

此模块根据输入的控制信号与数据找到和读写内存的地址以及数据，进行数据从内存中的读写操作。

```
assign memaddr        = rst?32'b0
```

```
                        :(aluop == 5'b01010 || aluop ==
5'b01011)?aluresult
                        :32'b0;
assign memwritedata    = rst?32'b0
                        :(memread || memwrite)? swdata
                        :32'b0;
```

7、Regfile.v：

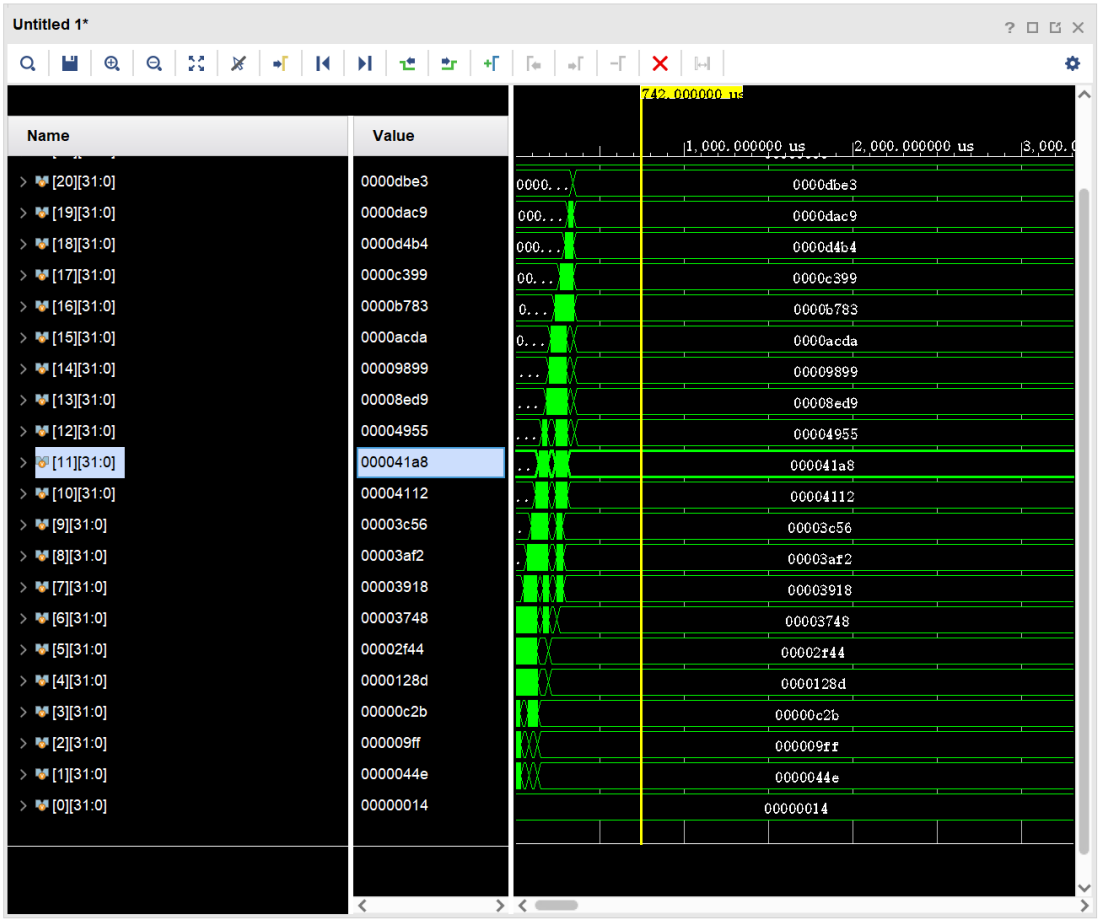
此模块存储了 32 个寄存器值，并根据控制信号进行初始化与读写。

```
always @(posedge clk) begin
    if(rst) begin
        for(i = 0; i <= 31; i = i + 1) begin
            registers[i] <= 31'b0;
        end
    end
    else begin
        if(regwrite && writeaddr != 0) begin
            registers[writeaddr] <= writedata;
        end
    end
end

assign readdata1 = rst? 32'b0
                   :(regread1)? ((regwrite && writeaddr == readaddr1) ?
writedata
                   :((readaddr1 == 0)? 32'b0
                   :registers[readaddr1]))
                   :32'b0;
assign readdata2 = rst? 32'b0
                   :(regread2)? ((regwrite && writeaddr == readaddr2) ?
writedata
                   :((readaddr2 == 0)? 32'b0
                   :registers[readaddr2]))
                   :32'b0;
```

四、仿真结果及分析

输入数据如右图所示：

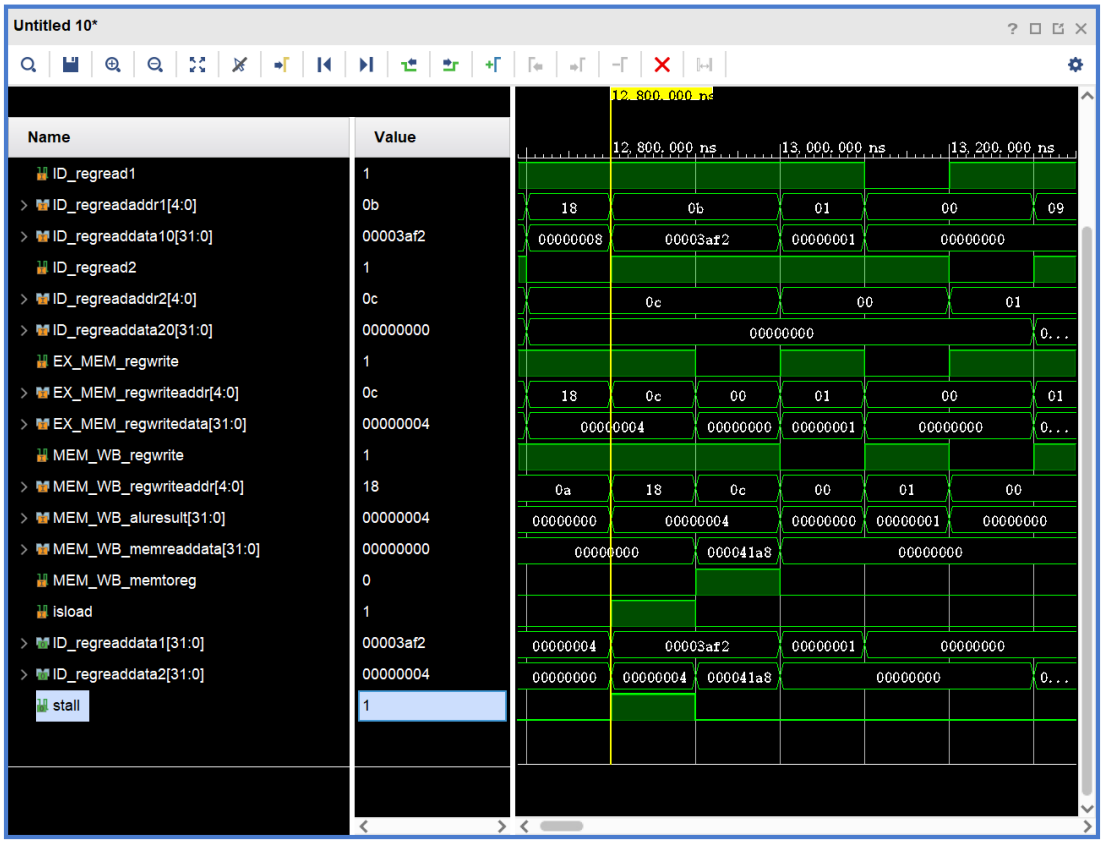
```
RAM_data[0] <= 32'h00000014;

RAM_data[1] <= 32'h000041A8;
RAM_data[2] <= 32'h00003AF2;
RAM_data[3] <= 32'h0000ACDA;
RAM_data[4] <= 32'h00000C2B;
RAM_data[5] <= 32'h0000B783;
RAM_data[6] <= 32'h0000DAC9;
RAM_data[7] <= 32'h00008ED9;
RAM_data[8] <= 32'h000009FF;
RAM_data[9] <= 32'h00002F44;
RAM_data[10] <= 32'h0000044E;
RAM_data[11] <= 32'h00009899;
RAM_data[12] <= 32'h00003C56;
RAM_data[13] <= 32'h0000128D;
RAM_data[14] <= 32'h0000DBE3;
RAM_data[15] <= 32'h0000D4B4;
RAM_data[16] <= 32'h00003748;
RAM_data[17] <= 32'h00003918;
RAM_data[18] <= 32'h00004112;
RAM_data[19] <= 32'h0000C399;
RAM_data[20] <= 32'h00004955;
```



如上图所示，使用设计的 tb.v 测试处理器，在数据存储的 1~20 位原来存储的数字经过插入排序后按照从小到大的顺序完成了排列，第 0 位表示的是 16 进制的数字 20，代表排序的数字有 20 个。

从仿真波形的特性还可以看出插入排序的特性：靠后的数字开始改变靠后，与最终确定的时间相对靠前，这是因为该插入排序算法设计从小到大开始排，每个数字排序完成后才进行下一位排序，因此后面的数据位置使用也更靠后；同时，若 n 位置的数字发生变化，那么其后面的数字（参与排序的）都要发生变化，且发生变化的顺序是从后往前。

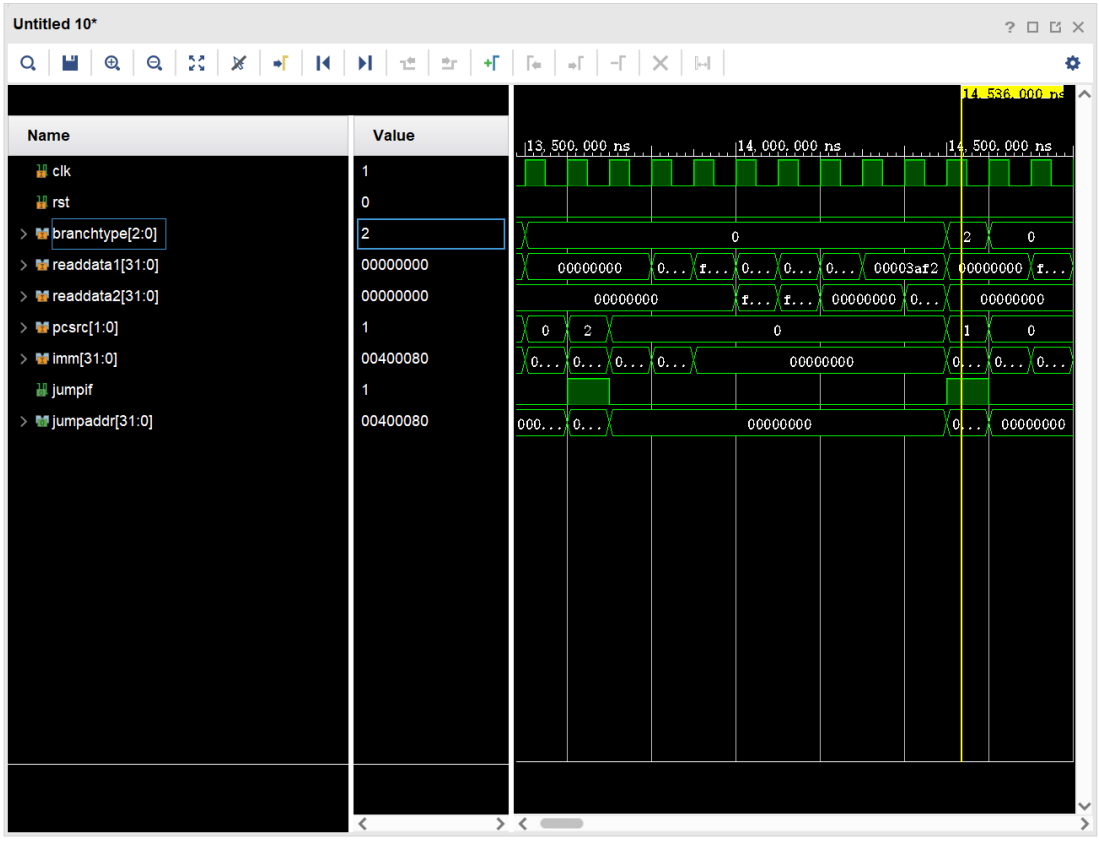
冒险的解决：



在算法中，有这样一段代码：

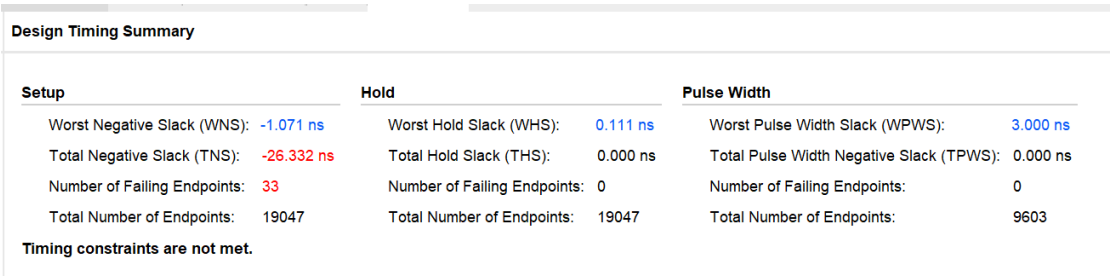
```
...
lw $t3 0($t2)
sw $t3 0($t1)
...
```

此处 sw 指令要使用的数据即为上一段 lw 要取的数据，在此情况下，处理器 stall 了一个周期后，数据转发到此处，解决了 load-use 的冒险问题。



在此处，有两处 jumpif 值为 1，前者为检测到指令为 j 型指令，故实现跳转；后者 branchtype=2，即为 beq 指令，由于读取的数据 1、2 相等，因此实现了分支。二者的实现说明了处理器完成了分支与跳转指令在 ID 阶段提前判断的功能。

五、综合情况（面积和时序性能）



如图为 100MHz 下的时序性能。根据计算最高频率为：

$$10 + 1.071 = 11.071ns$$

$$\frac{1}{11.071ns} = 90.33MHz$$

处理器未能达到 100MHz，猜测可能的原因是处理器设计的指令较多，在 ID 模块使用了较多且比较复杂的选择语句；同时由于算法设计的指令较多，InstructionMemory 中存储的数据较多，读取消耗时间较长。

根据计算所得将频率调低后达到要求，此时的资源利用率如下图所示：

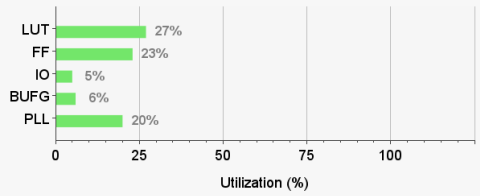
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.016 ns	Worst Hold Slack (WHS):	0.118 ns	Worst Pulse Width Slack (WPWS):	2.633 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:	19047	Total Number of Endpoints:	19047	Total Number of Endpoints:	9603

All user specified timing constraints are met.

Utilization Post-Synthesis | **Post-Implementation**

Graph | Table



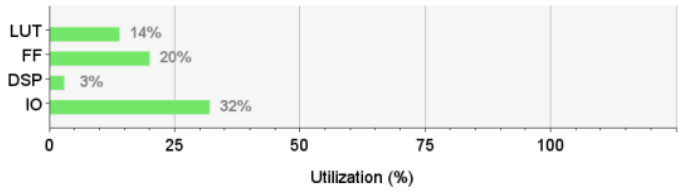
Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	5679	20800	27.30
FF	9597	41600	23.07
IO	13	250	5.20
BUFG	2	32	6.25
PLL	1	5	20.00

下左图为单周期处理器的资源消耗，二者对比，明显可见流水线处理器所使用的资源更多，与理论课上学习到的知识相一致：流水线处理器在形成多级流水以提高性能的同时，形成的级间寄存器以及其他的处理单元使用了更多资源。

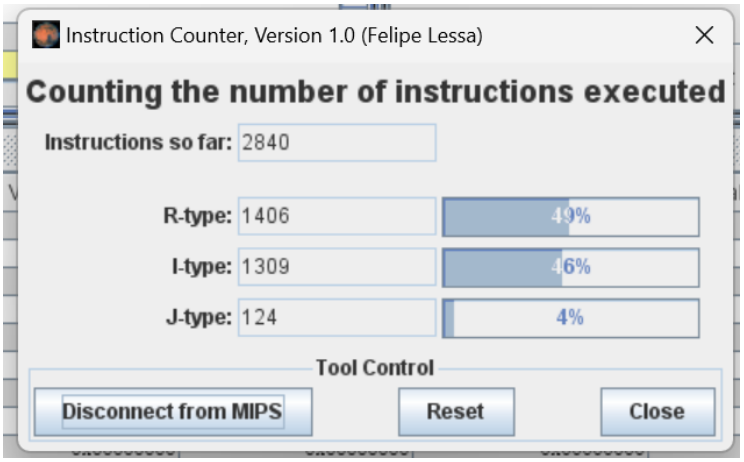
Resource	Utilization	Available	Utilization %
LUT	2947	20800	14.17
FF	8392	41600	20.17
DSP	3	90	3.33
IO	68	210	32.38

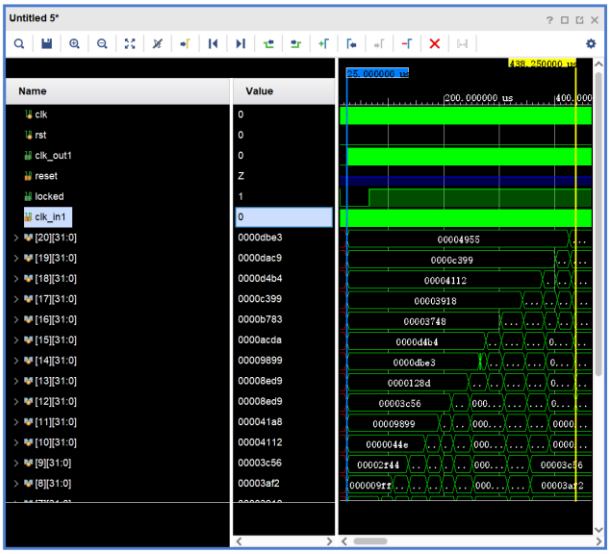


在 CPI 测算环节，由于自主设计的处理器中地址存储单元与 mars 模拟器的默认存储单元不一致，因此我对原来的汇编代码进行了一定修改使其能够在 mars 模拟器上运行，该部分代码与在处理器上运行的代码具有一定的差异，但仅在最开始的数据读写部分，差异较小。在 84MHz 的测试频率下运行程序，经过测算左下图为程序指令统计，右下图为运行时间，程序的 CPI 为：

$$CPI = \frac{\frac{413.25us}{1}}{\frac{84Mhz}{2840}} \approx 1.222$$

由理论课上学习可知，上 CPI 不可能刚好等于 1，因为流水级存在初始化等操作。在实际情况下，由于程序中存在冒险引发的 stall 等操作，同时还存在如上所述修改等原因，导致该处 CPI 约为 1.22。





六、硬件调试情况

经调试后，烧录程序到板上能够正确按从小到到的顺序显示排序的各个数字。

在调试过程中，也遇到了诸多的问题并逐一解决：如末尾位数不显示：该问题最终追溯到算法实现中，显示到数码管上时位数写入错误；只显示最小的数字：取值的计数器未能有效自增，最终也追溯到算法实现上等等。

本以为硬件调试会很复杂，但实际在 test bench 中自主设计与查看相关信号的过程中，就能精准地定位到出错的信号上并对相应可能出错的硬件设计或算法设计进行修复；同时也可见硬件与算法的设计共通之处。硬件设计过程中寻找问题要比软件算法上的代码设计寻找问题的过程复杂很多：自主设计 test bench、查看各个信号，但这种 debug 的能力也使得我进一步学会了寻找问题的分析方法，其与编写软件代码中的寻找问题方法其质一也。

七、思想体会

从理论课的学习到实验课的上手操作，我才真正体会到处理器设计的难处。尽管我们面对的问题仅仅是真实处理器的一个小小的子集，但其中的解决方案之多与可能出错的地方之多也需要我们作为初学者花费较多的时间与精力才能完成一个正常运行的设计。

除了挑战之外，我也有许多收获：

(1) 硬件描述语言的深入理解：开学时傻傻分不清 verilog 与 c++，在一个学期到暑假的编写过程中才体会到用硬件描述语言编写程序实际上就是一个绘制“图形”的过程，本次设计也让我体会到了硬件设计语言的一些本质所在与好用之处。相比于模拟电路手动绘制版图，数字电路的自动化设计对于大规模的电路设计更是有着缺其不可的重要性所在。

(2) 寻找问题的能力：

在不断的 debug 过程中，我逐渐掌握了编写 testbench，从诸多的信号中溯源，辨识关键信号的能力。一处小错误可能是硬件上的描述错误，也可能是算法上的谬误，甚至可能牵一发而动全身，诸如上文提到的从硬件的输出信号，找到输出信号的前者信号，再找到前者信号的计算过程，因而又溯源到汇编代码部分，这个过程的重复使我更能抓住重点所在。同时，我想有机会的话也许能开发或者利用现有的更智能的、更显式的信号溯源等功能。

(3) 工程设计能力：

流水线处理器的设计应该可以说是目前为止最“大”的大作业了，面对大作业，草草地上手就去做不一定就是对的，进行大工程的设计，我想预先进行良好的规划是非常有用的，良好的计划能事半功倍。

感谢老师与助教近半年的辛勤付出与指导！