

MIPS流水线处理器综合实验报告

姓名：陈彦旭

学号：2022010597 班级：无24

目录：

MIPS流水线处理器综合实验报告

一、实验内容

二、实验要求

三、实验设计

 支持的指令集

 控制信号

 级间寄存器

 数据冒险

 Case 1: EX_MEM to EX

 Case 2: MEM_WB to EX

 Case 3: load-use

 Case 4: load-store

 控制冒险

 分支指令冒险

 跳转指令冒险

 解决冒险的方法总结

 总体设计

 IF stage

 ID stage

 EX stage

 MEM stage

 WB stage

 外设部分

四、关键代码

五、调试情况

六、仿真结果

七、CPI 计算

八、FPGA 运行结果

九、性能分析

 静态时序分析

 资源使用情况

十、实验总结

附录、文件清单

一、实验内容

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，本实验选择的算法为直接插入排序。

理论课大作业中提交的可直接运行的汇编代码 `insert_sort.asm`，将其改为用于可执行流水线与 WELOG 开发板的代码 `test.asm`。

二、实验要求

设计一个 5 级流水线的 MIPS 处理器，建议采用如下方法解决竞争问题：

- a) 采用完全的 forwarding 电路解决数据关联问题。
- b) 对于 Load-use 类竞争采取阻塞一个周期 + Forwarding 的方法解决。
- c) 对于分支指令在 EX 阶段判断（提前判断也可以），在分支发生时刻取消 ID 和 IF 阶段的两条指令。
- d) 对于 J 类指令在 ID 阶段判断，并取消 IF 阶段指令。

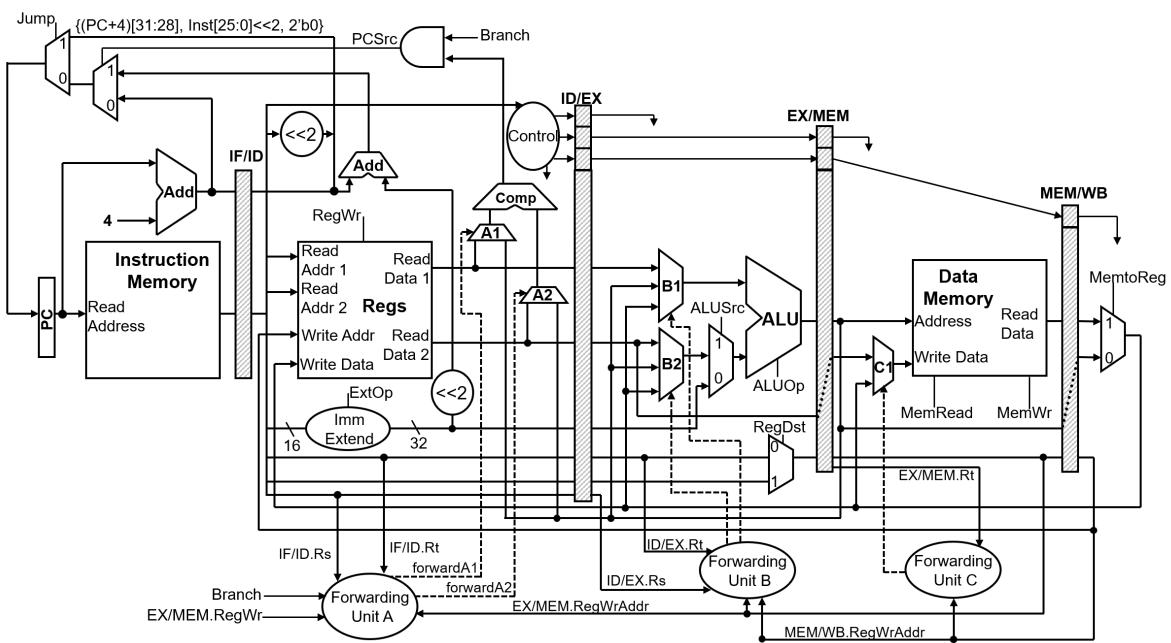
分支和跳转指令做如下扩充：分支指令 (beq、bne、blez、bgtz、bltz) 和跳转指令(j、jal、jr、jalr)；

数据存储的地址空间被划分为 2 部分：0x00000000 ~ 0x3FFFFFFF (字节地址) 为数据 RAM，可以提供数据存储功能；0x40000000 ~ 0x7FFFFFFF (字节地址) 为外设地址空间，对其地址的读写对应到相应的外设资源。除说明外，外设地址和描述不得更改或额外添加。

地址 (字节地址)	功能	描述
0x00000000~0x000007FF	数据存储器	512×32bits (可以根据需要自行调整大小)
0x40000010	七段数码管	0bit: CA 1bit: CB 7bit: DP 8bit: AN0 9bit: AN1 10bit: AN2 11bit: AN3

三、实验设计

流水线的整体框图为：



五级流水线执行指令可以分为五个阶段，每个阶段都为一个时钟周期：

1. IF, instruction fetch, 取指令。在 PC 所指向的指令存储单元处，取出指令传输到处理器中。
2. ID, instruction decode, 指令译码。译码单元根据指令类型进行分析，生成相应的控制信号（寄存器地址，立即数、多路选择器信号等）。寄存器接收到访问地址后将寄存器中数据传输到数据线上。
3. EX, execution, 操作执行。算术逻辑单元 ALU 根据上一阶段读取的寄存器值和决定运算类型的控制信号进行计算，输出计算结果。
4. MEM, memory access, 内存访问。将数据写入内存或从内存中读取数据。
5. WB, write back to register, 写回寄存器。将数据写回寄存器堆。

回顾：控制冒险解决方法

- 在ID阶段加入分支判断
- 提前分支判断

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	
WB	Reg[IR[15:11]] <= ALUOut	Reg[IR[20:16]] <= MDR		

支持的指令集

本实验所设计的处理器支持的是 MIPS32 指令集的一个子集：

1. 算术指令：

R-type: `add, addu, sub, subu, mul, and, or, nor, xor, sll, srl, sra,slt, sltu`

。

I-type: `lui, addi, addiu, andi, ori, slti, sltiu`。

2. 内存访问指令： `lw, sw`。

3. 分支指令： `beq, bne, blez, bgtz, bltz, bgez`。

4. 跳转指令： `j, jal, jr, jalr`。

5. 空指令 `nop`, 即 `sll, $0, $0, 0`。

对于一些指令的解释：

移位指令 `sll, srl, sra`：格式为 `sll rd, rt, shamt: R[rd] = R[rt] << shamt`，其中 `rs` 为 0。

加载高位立即数指令 `lui`：格式为 `lui rt, imm: R[rt] = {imm, 16'b0}`，其中 `rs` 为 0。

分支指令 `blez, bgtz, bltz, bgez` 的指令格式较为复杂，查询 MIPS 指令集手册得知格式为 `blez rs, offset: if(R[rs] = 0) PC = PC + 4 + offset << 2`。其中 `bgtz, blez` 的格式为指令中前 6 位为正常的 `OpCode[5:0]`，其中 `rt` 为 0。而对于 `bltz, bgez` 指令，前 6 位为 `RegImm[5:0] = Instruction[31:26] = 6'h01`，而在原先 `rs = Instruction[20:16]` 位置处为区分这两条指令的字段。

跳转回寄存器指令 `jr`：格式为 `jr rs: PC = R[rs]`，其中 `rt, rd` 为 0。

跳转回寄存器并链接指令 `jalr`：格式为 `jalr rd, rs: PC = R[rs], R[rd] = PC + 8`，其中 `rt` 为 0。

控制信号

在 ID 阶段根据下面两个指令字段译码：

`OpCode[5:0]`：6 位操作码，即 `Instruction[31:26]`。

`Funct[5:0]`：6 位功能码，即 `Instruction[5:0]`。

生成各类控制信号：

`PCSrc[1:0]`：下一个 PC 值的来源。0—顺序执行的 PC+4，1—分支地址，2—跳转地址，3—寄存器中的地址。

`RegWrite`：是否写入寄存器。1—写入；0—不写入。

`RegDst[1:0]`：写入寄存器的类型。0—寄存器 `rt`；1—寄存器 `rd`；2—第 31 号寄存器 `ra`。

`MemRead`：是否读取内存。1—读取；0—不读取。只有 `lw` 指令为 1，其余为 0。

`MemWrite`：是否写入内存。1—写入；0—不写入。只有 `sw` 指令为 1，其余为 0。

`MemtoReg[1:0]`：写入寄存器的数据来源。0—ALU输出结果；1—内存读取结果；2—PC+4。

`ALUSrc1`：控制 ALU 第一个操作数来源，1—位移量，0—寄存器 `rs` 读取数据。

`ALUSrc2`：控制 ALU 第二个操作数来源，1—立即数，0—寄存器 `rt` 读取数据。

`ExtOp`：是否进行有符号扩展。1—有符号扩展；0—无符号扩展。只有逻辑运算指令 andi 为1，其余为0。

`Luop`：是否选择高位立即数。1—选择加载到高位的立即数；0—选择立即数。只有 lui 指令为1，其余为0。

`ALUOp[3:0]`：控制 ALU 的操作类型。

级间寄存器

级间寄存器中存储各类控制信号和数据，并随着流水线向前移动。5级流水线有4个级间寄存器：

`IF_ID`, `ID_EX`, `EX_MEM`, `MEM_WB`。

处理器执行指令各个阶段需要使用到的控制信号：

- ID: `PCSrc`, `RegDst`, `ExtOp`, `LuOp`。
- EX: `ALUOp`, `ALUSrc1`, `ALUSrc2`。
- MEM: `MemRead`, `MemWrite`。
- WB: `MemtoReg`, `RegWrite`。

注：为了区分流水线在不同阶段、不同模块使用的信号与数据，同一个信号在不同阶段有不同的变量名称，例如写入寄存器控制信号 `RegWrite`，在 ID、EX、MEM、WB 四个阶段分别为：`ID_RegWrite`, `EX_RegWrite`, `MEM_RegWrite`, `WB_RegWrite`，同一个周期内不同的模块使用对应的控制信号进行操作，例如 WB 阶段使用 `WB_RegWrite` 进行寄存器堆写入操作；而同一个指令的控制信号在不同周期内是不同的变量，这样可以随着时钟沿由上一级寄存器传递到下一级寄存器。

数据冒险

数据冒险是指不同指令的操作数存在依赖关系造成的冒险，指令所需要的数据依赖其他指令的结果。通常采用数据转发 (Forwarding) 来解决数据冒险，可能还需要对流水线进行阻塞 (Stall)。

如果在同一周期内 ID 级和 WB 级分别需要读写相同的寄存器，我们可以设置寄存器堆来解决这样的数据冒险：写入寄存器发生在时钟上升沿（一个时钟周期的开始），使用 always 语句非阻塞赋值，而读取寄存器操作实际上是组合逻辑，使用 assign 语句赋值，读取端口数据紧随着寄存器编号或寄存器内的值的改变而变化，不需要等待时钟沿的到来，如此能保证一个周期内寄存器堆先写后读（写后读，RAW），实现“内部转发”。

Case 1: EX_MEM to EX

连续两条算术指令，本条指令要读取的寄存器 `RegRs` or `RegRt` 是上一条指令要写入的寄存器 `RegWrAddr`。

例如：

```

1 # example 1
2 add $t1, $t2, $t3 # Inst 1
3 add $t4, $t1, $t2 # Inst 2

```

对于 `example 1`，第二条指令 `Inst 2`，需要在 ID 阶段读取寄存器 `$t1` 中的值，而此时上一条指令 `Inst 1` 还处于 EX 阶段，它在 WB 阶段才会将结果写入寄存器 `$t1`，数据的产生晚于需要使用的时刻，造成数据冒险。

指令 `Inst 1` 的 EX 阶段的周期内，ALU 的计算结果 `EX_ALUOut` 在该周期前结束前就可以准备好，该周期结束时所有数据进入级间寄存器 `EX_MEM`。在下一个周期，指令 `Inst 1` 进入 MEM 阶段，指令 `Inst 2` 进入 EX 阶段，需要准备 ALU 操作数 `$t1` 的值，我们在此时进行转发，将指令 `Inst 1` 的计算结果 `MEM_ALUOut` 转发到 ALU 输入端，因此转发通路为：`EX_MEM --> EX`，以 ALU 的第一个操作数为例，转发逻辑为：

1. 上一条指令的计算结果要写入寄存器：`MEM_RegWrite == 1`。
2. 上一条指令要写入的寄存器不是 0 号寄存器：`MEM_RegWrAddr != 0`。
3. 上一条要写入的寄存器与这一条指令要读取的 `rs` 寄存器相同：`EX_RegRs == MEM_RegWrAddr`
- 。

Case 2: MEM_WB to EX

相隔一条指令的两条算术指令，本条指令要读取的寄存器 `RegRs or RegRt` 是上上一条指令要写入的寄存器 `RegWrAddr`。

例如：

```

1 # example 2
2 add $t1, $t2, $t3 # Inst 1
3 add $t4, $t1, $t2 # Inst 2
4 add $t5, $t1, $t2 # Inst 3
5
6 # example 3
7 add $t1, $t2, $t3 # Inst 1
8 add $t1, $t1, $t4 # Inst 2
9 add $t5, $t1, $t2 # Inst 3

```

对于 `example 2`，第三条指令 `Inst 3` 需要在 ID 阶段读取寄存器 `$t1` 中的值，而此时上上一条指令 `Inst 1` 还处于 MEM 阶段，它在 WB 阶段才会将结果写入寄存器 `$t1`，数据的产生晚于需要使用的时刻，造成数据冒险。

指令 `Inst 1` 在 EX 阶段的周期内，计算出 ALU 的结果 `EX_ALUOut`，在 MEM 阶段并未被使用（因为是算术指令），当进入下下一个周期时开始 WB 阶段，此时指令 `Inst 3` 进入 EX 阶段，需要准备好操作数 `$t1` 的值，我们在此时进行转发，将指令 `Inst 1` 的计算结果 `WB_ALUOut` 转发到 ALU 输入端，因此转发通路为：`MEM_WB --> EX`，以 ALU 的第一个操作数为例，转发逻辑为：

1. 上上一条指令的计算结果要写入寄存器：`WB_RegWrite == 1`。
2. 上上一条指令要写入的寄存器不是 0 号寄存器：`WB_RegWrAddr != 0`。
3. 上上一条要写入的寄存器与这一条指令要读取的 `rs` 寄存器相等：`EX_RegRs == WB_RegWrAddr`
- 。

4. 我们还需考虑 `example 3`，虽然 `Inst 1` 和 `Inst 3` 同样存在满足上面的数据依赖，但是从 `Inst 1` 转发过来的数据不是最新的，此时应该从 `Inst 2` 转发，实际上属于 **Case 1**（从上一条指令转发）。因此需要排除掉种情况（或者设置判断 Case 1 的优先级更高）。上一条指令与该指令不能写入相同的寄存器，否则转发的数据不是最新的：`MEM_RegWrAddr != EX_RegRs || ~MEM_RegWrite`。

综合“Case 1”与“Case 2”，ALU 输入端的转发逻辑用 Verilog 语言描述为：

```

1 // 00: read data from the Register File
2 // 01: forward ALUOut from EX_MEM to EX
3 // 10: forward ALUOut from MEM_WB to EX
4 assign ALU_forwardA =
5     (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRs == MEM_RegWrAddr)
6 ? 1 :
7     (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRs == WB_RegWrAddr) ?
8 : 0;
9
10 assign ALU_forwardB =
11     (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRt == MEM_RegWrAddr)
12 ? 1 :
13     (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRt == WB_RegWrAddr) ?
14 : 0;

```

Case 3: load-use

一条 load 指令之后，紧随着几条指令，它们需要读取的寄存器与 load 指令写入的那个寄存器相同。

例如：

```

1 # example 4
2 lw $t1, 0($t2) # Inst 1
3 add $t2, $t1, $t2 # Inst 2
4 sub $t3, $t1, $t3 # Inst 3

```

对于 `example 4`，第二条指令 `Inst 2` 在 ID 阶段需要读取寄存器 `$t1` 中的值，而此时上一条指令 `Inst 1` 还处在 EX 阶段，它在 WB 阶段才会将结果写入寄存器 `$t1`。同理第三条指令 `Inst 3` 在 ID 阶段需要读取寄存器 `$t1` 中的值，而此时上一条指令 `Inst 1` 还处在 MEM 阶段，未将结果写入寄存器 `$t1`。数据的产生晚于需要使用的时刻，造成数据冒险。

由于内存读取的数据在 MEM 阶段结束前就可以准备好，但此时下一条指令在 EX 阶段开始时就需要该内存读取数据，所以我们必须让流水线阻塞一个周期，阻塞 load 之后的下一条指令，然后才能进行转发。

指令 `Inst 2` 执行到 ID 阶段时，可以判断出与正处于 EX 阶段的 load 指令之间的依赖，该周期结束前生成阻塞信号，在下一个周期将流水线的 IF 和 ID 阶段阻塞。阻塞的这个周期内，load 指令进入 MEM 阶段，而 `Inst 2` 在 IF 阶段“原地踏步”。阻塞 IF 与 ID 级的逻辑为：

1. 上一条指令为 load 指令： `EX_MemRead == 1`。
2. 上一条指令写入的寄存器不为 0 号寄存器： `EX_RegWrAddr != 0`。
3. 上一条指令写入的寄存器与这一条指令读取的寄存器相同： `EX_RegWrAddr == ID_RegRs || EX_RegWrAddr == ID_RegRt`。

用 Verilog 语言描述为：

```

1  (!ID_MemWrite && EX_MemRead && EX_RegWrAddr != 0 && (EX_RegWrAddr == ID_RegRs
|| EX_RegWrAddr == ID_RegRt))
2  ||
3  (ID_MemWrite && EX_MemRead && EX_RegWrAddr != 0 && EX_RegWrAddr == ID_RegRs)

```

阻塞的这一个周期结束时，load 指令已经得到内存读取数据。阻塞后的第一个周期，load 指令进入 WB 阶段，指令 Inst 2 进入 EX 阶段，需要准备好 ALU 操作数，在此时进行转发，转发通路为： MEM_WB --> EX，此情况与“Case 2：从上上条指令转发”类似，转发通路相同，只是转发的数据变为 WB_MemReadData 而非 WB_ALUout，此处需要增加额外的条件判断。修改后，综合考虑 Case 1, Case 2, Case 3，完整的的 ALU 输入端转发逻辑用 Verilog 描述为：

```

1  // 00: read data from the Register File
2  // 01: forward ALUOut from EX_MEM to EX
3  // 10: forward ALUOut from MEM_WB to EX
4  // 11: forward MemReadData from MEM_WB to EX
5  assign ALU_forwardA =
6      (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRs == MEM_RegWrAddr)
? 1 :
7      (WB_MemRead && WB_RegWrAddr != 0 && EX_RegRs == WB_RegWrAddr) ? 3
:
8      (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRs == WB_RegWrAddr) ?
2 : 0;
9
10 assign ALU_forwardB =
11     (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRt == MEM_RegWrAddr)
? 1 :
12     (WB_MemRead && WB_RegWrAddr != 0 && EX_RegRt == WB_RegWrAddr) ? 3
:
13     (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRt == WB_RegWrAddr) ?
2 : 0;

```

在同一个周期内，指令 Inst 3 在 ID 阶段，由于寄存器堆支持先写后读，因此 load 指令与指令 Inst 3 之间无需进行转发。

又例如：

```

1 # example 5
2 lw $t1, 0($t2) # Inst 1
3 add $t5, $t3, $t4 # Inst 2
4 sub $t3, $t1, $t3 # Inst 3

```

对于 example 5，第一条 load 指令与第二条指令 Inst 2 不存在数据依赖，而与第三条指令 Inst 3 存在 load-use 冒险。此时执行中间间隔的指令 Inst 2 就相当于使指令 Inst 3 阻塞了一个周期，因此我们只需在 load 指令进入 WB 阶段时将内存读取数据转发到指令 Inst 3 的 ALU 输入端即可，转发通路和逻辑与前面讨论的相同。

Case 4: load-store

相邻的两条 load 指令和 store 指令读写同一个寄存器，例如：

```
1 # example 6
2 lw $t1, 0($t2) # Inst 1
3 sw $t1, 4($t2) # Inst 2
```

对于 example 6，第二条 store 指令 Inst 2 在 ID 阶段需要读取寄存器 ID_RegRt = \$t1 的值，此时第一条 load 指令 Inst 1 处于 EX 阶段，它在 WB 阶段才会将内存读取数据写入寄存器 \$t1。数据的产生晚于需要使用的时刻，造成数据冒险。

好在 store 指令在 EX 阶段时，我们期望得到的 \$t1 的值不参与计算（不会被使用到），因此我们不必阻塞 store 指令，只需在 store 指令处于 MEM 阶段的开始时准备好写入内存的数据即可，此时 load 指令处于 WB 阶段，已经得到将要写入 \$t1 的数据，我们进行转发：MEM_WB --> MEM，内存写入端口的转发逻辑为：

1. 上一条指令是 load 指令：WB_MemRead。
2. 这一条指令是 store 指令：MEM_Memwrite。
3. load 指令写入的寄存器与 store 指令读取的寄存器相同：WB_RegWrAddr == MEM_RegRt。
4. load 指令要写入的寄存器不为 0 号寄存器：WB_RegWrAddr != 0。

用 Verilog 语言描述为：

```
1 // 0: read data from the Register File
2 // 1: forward MemReadData from MEM_WB to MEM
3 assign MEM_forward =
4     (WB_MemRead && MEM_Memwrite && WB_RegWrAddr != 0 && MEM_RegRt ==
      WB_RegWrAddr) ? 1 : 0;
```

又例如：

```
1 # example 7
2 lw $t1, 0($t2) # Inst 1
3 sw $t3, 4($t1) # Inst 2
```

对于 example 7，第一条 load 指令 Inst 1 写入寄存器 \$t1 的数据，在第二条 store 指令 Inst 2 中被用来计算内存访问的地址，此情形属于 load-use 冒险，可并入“Case 3”，使用阻塞 + 转发解决。

控制冒险

取指令的 PC 依赖于其他指令的结果，由分支指令和跳转指令造成。与数据冒险不同的是，流水线发现可能会出错的时候，错误已经发生（已经取出错误的指令），因此除了转发、阻塞以外，还需要进行清除（Flush），清除流水线中错误的指令。

分支指令冒险

以 `beq` 为例：

```

1 | # example 8
2 | beq $t1, $t2, label # Inst 1
3 | addi $t3, $t3, 1 # Inst 2
4 | ...
5 | Label:
6 | addi $t1, $t1, 1 # Inst 3

```

一般情况下，分支指令在 EX 阶段计算分支结果，在 MEM 阶段更新 PC 值。由于我们事先不知道是否执行分支，因此必须等待分支指令执行完 MEM 阶段、更新好正确的 PC 值后，我们才能在下一个周期取指令。

对于 `example 8`，我们需要在 `beq` 指令 `Inst 1` 之后阻塞 3 个周期，但这会极大影响流水线效率，因为汇编程序中循环是用分支指令实现的。

因此我们采取**提前分支判断**的方式：

在分支指令的 ID 阶段读取寄存器数据后立即计算分支结果并更新 PC，这样就可以在下一个周期取出下一条指令 `Inst 2 or Inst 3`。同时，因为该周期内分支指令的“顺序下一条”指令 `Inst 2` 已经进入 IF 级，如果执行分支，下一条应该执行的是 `Inst 3`，因此我们需要清除 IF 级的指令 `Inst 2`，如果不执行分支，就正常进行流水即可。

分支执行时清除 IF 级的逻辑为：

```
1 | ID_PCSrc == 2'b00 || (ID_PCSrc == 2'b01 && branch_taken == 0)
```

但与此同时，由于分支的提前判断，等价于提前执行了 EX 阶段，此时新的数据可能还未产生，会引发新的数据冒险，我们需要考虑更多转发的情况。分支指令最多可能与前面两条指令产生数据冒险（与前面第三条指令，也就是上上上条指令之间的数据依赖由寄存器堆的先写后读保证）。

例如：

```

1 | # example 9
2 | addi $t1, $t1, 1 # Inst 1
3 | addi $t2, $t2, 2 # Inst 2
4 | beq $t1, $t3, label # Inst 3

```

此为“情形一：分支指令的上上一条指令为算术指令且不为 load 指令”。对于 `example 9`，分支指令 `Inst 3` 在 ID 阶段需要读取寄存器 `$t1` 中的值，指令 `Inst 1` 在上个周期 EX 阶段已经计算出新的 `$t1` 的值，该周期正处于 MEM 阶段，在此时进行转发：`EX_MEM --> ID`，分支判断单元输入端的转发逻辑为：

1. 上上一条指令写入寄存器 `MEM_RegWrite`：
2. 上上一条指令写入的寄存器不为 0 号寄存器 `MEM_RegWrAddr != 0`：
3. 上上一条指令写入的寄存器与这一条指令读取的寄存器相同 `MEM_RegWrAddr == ID_RegRs`：

用 Verilog 语言描述为：

```

1 // 0: no forwarding
2 // 1: forward from EX_MEM to ID
3 assign branch_forward1 =
4     (MEM_RegWrite && MEM_RegWrAddr != 0 && ID_RegRs == MEM_RegWrAddr)
5 ? 1 : 0;
6
7 assign branch_forward2 =
8     (MEM_RegWrite && MEM_RegWrAddr != 0 && ID_RegRt == MEM_RegWrAddr)
9 ? 1 : 0;

```

下面再考虑只通过转发无法解决的情况：

```

1 # example 10
2 lw $t1, 0($t2) # Inst 1
3 addi $t2, $t2, 2 # Inst 2
4 beq $t1, $t3, label # Inst 3
5
6 # example 11
7 addi $t1, $t1, 2 # Inst 1
8 beq $t1, $t3, label # Inst 2

```

以上两种情况 `example 10`, `example 11` 分别对应“**情形二：分支指令的上一条指令是 load 指令**”和“**情形三：分支指令的上一条指令是算术指令且不为 load 指令**”。此时没有直接的转发通路，必须使 ID 阶段的分支指令阻塞一个周期。

在阻塞的这一个周期内：

对于 `example 10`，load 指令进入 WB 阶段，分支指令保持在 ID 阶段，寄存器堆先写后读，无需进行转发。

对于 `example 11`，变为“**情形一：分支指令的上一条指令为算术指令且不为 load 指令**”，按照前面的方法进行转发 `EX_MEM --> ID` 即可。

分支指令阻塞 IF 与 ID 级的逻辑为：

1. 这一条指令为分支指令：`ID_PCSrc[1:0] == 2'b01`。
2. 上一条为算术指令，且写入寄存器与这一条指令读取寄存器相同且不为0号，或者上上一条为 load 指令，且写入寄存器与这一条指令读取寄存器相同且不为0号。

用 Verilog 语言描述为：

```

1 ID_PCSrc == 2'b01 // data hazard caused by branch
2 &&
3 (
4     (EX_RegWrite && EX_RegWrAddr != 0 && (EX_RegWrAddr == ID_RegRs ||
5         EX_RegWrAddr == ID_RegRt))
6     ||
7     (MEM_MemRead && MEM_RegWrAddr != 0 && (MEM_RegWrAddr == ID_RegRs ||
8         MEM_RegWrAddr == ID_RegRt))
9 )

```

又例如：

```

1 # example 12
2 lw $t1, 0($t2) # Inst 1
3 beq $t1, $t3, label # Inst 2

```

此为“情形四：分支指令的上一条指令是 load 指令”。其实这同时也属于 load-use 冒险，因此 load 指令 Inst 1 之后的分支指令 Inst 2 会阻塞一个周期，这时会自动退化为前面的“情形二：分支指令的上一条指令是 load 指令”，再阻塞一个周期，最后总共阻塞两个周期，此时 \$t1 的值由寄存器堆先写后读保证，无需进行转发。

跳转指令冒险

跳转指令的目标地址在 ID 阶段计算出来，而此时下一条指令已经进入流水线 IF 阶段，因此一定会造成控制冒险（除非跳转的目标地址就是 PC+4，但这样的跳转没有意义），在软件层面上我们可以在跳转指令后加一条空指令 nop，在本实验中我们从硬件层面上考虑解决。

```

1 # example 13
2 j label # or jr $ra, Inst 1
3 add $t3, $t1, $t2 # Inst 2
4 ...
5 label:
6 sub $t3, $t1, $t2 # Inst 3

```

对于 example 13，流水线在 ID 阶段译码后才知道是指令 Inst 1 为跳转指令，此时下一条指令 Inst 2 已经进入 IF 阶段，此时如果我们只是阻塞 IF 级一个周期，并不能阻止指令 Inst 2 的执行，因此我们需要清除 IF 阶段的指令 Inst 2，使级间寄存器 IF_ID 的内容变为0，在下一周期就可以根据更新好的 PC（跳转地址）取出应该执行的指令 Inst 3。跳转指令清除 IF 级的逻辑为“ID 级的指令为跳转指令”，用 Verilog 语言描述为：

```
1 ID_PCSrc == 2'b10 || ID_PCSrc == 2'b11
```

又例如：

```

1 # example 14
2 jal label # or jalr $s1, $s2, Inst 1
3 add $t3, $t1, $t2 # Inst 2
4 ...
5 label:
6 sub $t3, $t1, $t2 # Inst 3

```

对于“跳转并链接类指令” jal, jalr，还需要将当前“跳转并链接类指令”的“顺序下一条指令的 PC ”存入寄存器，用于函数返回时继续执行主函数。对于 example 14，指令 Inst 1 为 jal or jalr，在 ID 阶段更新跳转地址后，还需要将其 PC（跳转并链接类指令的 PC，不是跳转目标地址的 PC）继续保持到 WB 阶段，然后将 PC + 4 写入到寄存器 \$31 or WB_RegWrAddr。

```

1 # example 15
2 addi $t1, $t1, 1 # Inst 1
3 jr $t1 # Inst 2

```

我们注意到，跳转回寄存器类指令 `jr`, `jalr` 与分支指令相同，都需要在 ID 阶段提前读取寄存器的值，因此造成的数据冒险类型相同：

1. 跳转回寄存器类指令的上一条指令为算术指令且不为 load 指令——直接转发。
2. 跳转回寄存器类的上一条指令是 load 指令——阻塞 + 转发。
3. 跳转回寄存器类的上一条指令为算术指令且不为 load 指令——阻塞。
4. 跳转回寄存器类的上一条指令是 load 指令——阻塞。

因此，我们采取与“分支提前判断导致的数据冒险”相同的解决办法，这里不再赘述。

解决冒险的方法总结

何时转发：数据冒险。

1. 转发到 ALU 输入端；
2. 转发到内存写入端口；
3. 分支提前判断，转发到分支比较单元的输入端。
4. 跳转回寄存器类指令，转发最新的寄存器值。

何时阻塞：

1. load-use 数据冒险中，阻塞 load 指令之后紧随的第一个指令；
2. 分支提前判断导致数据冒险时，阻塞分支指令。
3. 跳转回寄存器类指令导致数据冒险时，阻塞跳转回寄存器类指令。

何时清除：

1. 清除跳转指令后面紧随的第一个指令。
2. 若执行分支，清除分支指令后面紧随的第一个指令。

阻塞与清除的具体实现：

由于我们所支持的指令级有限、没有考虑中断异常等复杂情况，我们所遇到的控制冒险均可以通过“清除 IF 级”和“阻塞 IF 和 ID 级”来解决。事实上 IF 级和 ID 级是“不平权”的，原因在于 IF 级并没有前级寄存器，且 IF 仅仅是取指令，没有做任何计算和转发。**对于“清除 IF 级”，**我们需要做的是在级间寄存器 `IF_ID` 中将输出的所有 ID 级信号全部置为 0，也就是流水线仍然前进，只是要清除的那个信号变为 `nop`。**对于“阻塞 IF 和 ID 级”，**IF 级中只需保持 `IF_PC` 即可，在级间寄存器 `IF_ID` 中，输出的所有 ID 级信号给自身赋值（保持），`ID_EX` 中，输出的所有 EX 级信号全部置为 0，代替 ID 级的指令执行下一周期。下面表格显示了在 ID 级指令判断出需要阻塞或清楚的时候，在下一个周期开始时，流水线前三级的信号更新情况：

<code>flush_IF</code>	<code>stall_IF_ID</code>	<code>IF signals</code>	<code>ID signals</code>	<code>EX signals</code>
0	0	$IF_PC \leq IF_PC_next$	$ID \leq IF$	$EX \leq ID$
0	1	$IF_PC \leq IF_PC$	$ID \leq ID$	$EX \leq 0$
1	0	$IF_PC \leq IF_PC_next$	$ID \leq 0$	$EX \leq ID$
1	1	$IF_PC \leq IF_PC$	$ID \leq ID$	$EX \leq 0$

总体设计

上面的分析中，我们大多是从时间角度考虑，将一个指令分解为5个阶段，随着指令在流水线中的前进，分析每个阶段中该指令使用不同的模块的执行情况。下面我们将从空间角度考虑，将流水线各个模块分离出来，考虑在同一个时钟周期内，各个独立的模块如何使用不同的控制信号和数据完成该级流水线的任务。

IF stage

该周期所使用的（用于取指令的）PC值为`IF_PC`。首先根据控制信号`ID_PCSrc`决定下一个周期的PC：`IF_PC_next`，可能是顺序执行的下一个地址、分支地址、跳转地址、寄存器中的地址。在周期开始时的时钟上升沿更新`IF_PC <= IF_PC_next`。

在指令存储器`InstructionMemory.v`中，以`IF_PC`为地址取出指令`IF_Instruction`。

该周期结束时，传入级间寄存器`IF_ID`的有：`reset, clk, flush_IF, stall_IF_ID, IF_Instruction, IF_PC`。

ID stage

(1) 首先进行指令译码：在控制信号模块`Control.v`中，根据指令的操作码`OpCode = ID_Instruction[31:26]`和功能码`Funct = ID_Instruction[5:0]`生成`ID_PCSrc, ID_RegWrite`等控制信号。

(2) 访问寄存器堆在模块`RegisterFile.v`中实现：寄存器堆的两个读取端口分别为`ID_RegRs = ID_Instruction[25:21], ID_RegRt = ID_Instruction[20:16]`，分别读取到寄存器中的数据`ID_RegReadDataA, ID_RegReadDataB`。

需要注意的是：ID阶段以后出现的`RegReadDataA, RegReadDataB`和EX阶段以后的`RegRsData, RegRtData`意思虽然相近但是不完全相同，前者是按照指令中寄存器编号`RegRs, RegRt`直接从寄存器堆中读取出来的数据，但不一定是最新的，而后者是考虑转发的情况下该指令实际上需要使用的最新值。

(3) 立即数单元根据控制信号`ID_ExtOp, ID_LuOp`进行立即数扩展，得到结果`ID_ExtImm`，它可能是有符号或无符号扩展后的立即数(I-type指令)或者高位立即数(lui指令)。扩展完成后，上述两个控制信号不需要再向下一级流水线传递。同时，移位指令需要用到的位移量`shamt`其实可以不用单独作为变量，它可以表示为`ID_ExtImm[10:6]`，因此我们只需向下级传递`ID_ExtImm`即可。

(4) 对于分支指令，为了读取最新的寄存器值，在阻塞的前提下，需要判断是否进行转发，分支比较单元的输入端根据转发信号选择直接从寄存器中读取的值、ALU输出端转发过来的值，输出分支判断结果。根据是否执行分支，计算分支地址`BranchAddr`。

(5) 对于J-type的跳转指令`j, jal`，可直接根据`ID_PC, ID_Instruction`计算出跳转地址`JumpAddr`。

对于R-type的跳转会寄存器指令`jr, jalr`，跳转地址`RegisterAddr`需要从寄存器堆中读取，此时仍需考虑转发，事实上它们读取的寄存器值`ID_RegReadDataA`，与分支指令读取的相同，可以直接复用。`ID_PC`仍需要向下级流水线传递，因为`jal, jalr`需要在WB阶段将`WB_PC + 4`写入寄存器`WB_RegWrAddr`。

由于分支指令和跳转指令在该阶段已经计算出新的PC，因此控制信号`PCSsrc`不需要向下级流水线传递。

(6) 控制冒险检测单元 `Hazardunit.v` 中，根据是否是跳转指令或是否执行分支指令，生成清除信号 `flush`。根据是否出现 load-use 冒险、分支与跳转指令导致的数据冒险，生成阻塞信号 `stall`。

(7) 由于转发单元需要判断该指令是否与上几条指令（位于流水线下面几级的指令）是否存在数据冒险，因此要向下级流水线传递读取寄存器编号 `ID_RegRs`, `ID_RegRt`，以及写入寄存器编号 `ID_RegWrAddr = ID_RegRt or ID_RegRd or $31`，由控制信号 `ID_RegDst` 获取。这里我们不传递 `ID_RegRd` 的原因是：当指令中需要用到 `rd` 寄存器编号时，最后写入的寄存器编号一定为 `rd` (R-type)，同时传递写入寄存器编号更为直接，也更便于转发逻辑的判断，因为数据冒险的本质是写入的那个寄存器未得到最新值的时候就要被使用，最后在 WB 阶段我们只关心写入的寄存器编号就可以了。

该周期结束时，传入级间寄存器 `ID_EX` 的有：

```
reset, clk, stall_IF_ID ;
ID_PC, ID_RegWrite, ID_MemRead, ID_MemWrite, ID_MemtoReg, ID_ALUSrc1, ID_ALUSrc2,
ID_ALUOp ;
ID_ExtImm, ID_RegReadDataA, ID_RegReadDataB, ID_RegRs, ID_RegRt, ID_RegWrAddr .
```

EX stage

该阶段进行 ALU 单元的计算。

(1) 首先考虑转发，在 `ForwardingUnit.v` 中完成。由于从寄存器堆直接读取到的数据 `EX_RegReadDataA`, `EX_RegReadDataB` 不一定是最新的，因此我们将 MEM 级的 `MEM_ALUOut`、WB 级的 `WB_ALUOut`, `WB_MemReadData` 转发过来，判断并选择寄存器的最新值，称为 `EX_RegRsData`, `EX_RegRtData`，他们才是在 EX 级指令之前的指令完整执行后，编号为 `EX_RegRs`, `EX_RegRt` 中真正的值。

(2) ALU 控制单元 `ALUControl.v`，根据操作码 `EX_OpCode` 和控制信号 `EX_ALUOp` (由指令 `EX_PC` 中字段获取)，选择 ALU 的运算类型 (加减乘、移位、逻辑与或等)。

(3)

ALU 的第一个操作数，根据控制信号 `EX_ALUSrc1`，选择数据 `EX_RegRsData` 或移位量 `shamt = EX_ExtImm[10:6]`。

ALU 的第二个操作数，根据控制信号 `EX_ALUSrc2`，选择数据 `EX_RegRtData` 或立即数 `EX_ExtImm`。

以两个操作数为输入，在模块 `ALU.v` 中完成运算，输出计算结果 `EX_ALUOut`。

(4) 该级使用过的 `EX_ALUSrc1`, `EX_ALUSrc2`, `EX_ALUOp`, `EX_ExtImm`, `EX_RegRsData` 不用向下级传递，而 `EX_RegRtData` 可能是 store 指令将要写入内存的数据，需要传递。

该周期结束时，传入级间寄存器 `EX_MEM` 的有：

```
reset, clk ;
EX_PC, EX_RegWrite, EX_MemRead, EX_MemWrite, EX_MemtoReg ;
EX_ALUOut, EX_RegRtData, EX_RegRt, EX_RegWrAddr .
```

MEM stage

(1) 首先考虑转发，在 `ForwardingUnit.v` 中完成。store 指令将要写入内存的数据 `MEM_RegRtData` 不一定是最新的 (load-use 冒险)，因此要转发 WB 级的内存读取数据 `WB_MemReadData`，判断并选择真正要写入内存的数据 `MEM_MemwriteData`。

(2) 在数据存储器 `DataMemory.v` 中进行内存读写：

对于 store 指令，在控制信号 `MEM_MemWrite` 有效条件下，以 `MEM_ALUout` 作为访存地址，将 `MEM_MemwriteData` 写入改地址中。

对于 load 指令，在控制信号 `MEM_MemRead` 有效条件下，以 `MEM_ALUout` 作为访存地址，读取到该地址存储的数据，记为 `MEM_MemReadData`。

(3) 该周期使用过的 `MEM_Memwrite`, `MEM_RegRtData`, `MemRegRt` 不需要向下级传递，但由于与 WB 级共同判断 load-use 冒险，因此需要传递 `MEM_MemRead`。

该周期结束时，传入级间寄存器 `MEM_WB` 的有：

```
reset, clk ;
MEM_PC, MEM_RegWrite, MEM_MemRead, MEM_MemtoReg ;
MEM_ALUOut, MEM_MemReadData, MEM_RegWrAddr .
```

WB stage

寄存器堆写入端口为 `WB_RegWrAddr`，写入数据为 `WB_RegwriteData`，控制信号 `WB_MemtoReg` 决定写入数据的来源。该模块 `RegisterFile.v` 已经在 ID 级实现过。

对于算术指令 `WB_MemtoReg == 00`，将 `WB_ALUout` 写入到编号为 `WB_RegWrAddr` 的寄存器中。

对于 load 指令 `WB_MemtoReg == 01`，将内存读取结果 `WB_MemReadData` 写入到编号为 `WB_RegWrAddr` 的寄存器中。

对于跳转并链接类指令 `WB_MemtoReg == 10`，将 `WB_PC + 4` 写入到编号为 `WB_RegWrAddr` 的寄存器中。

外设部分

最后要使用软件方法，将排序结果依次显示在 FPGA 开发板的4位八段数码管上。我们在数据存储器 `DataMemory.v` 中曾将高位地址 0x40000010 设为数码管专用空间，该地址存储的字中，低8位依次为数码管的 8 个段 `tube_segment[7:0]`，12-9 位为数码管显示的使能信号 `tube_select[4:0]`。四个数码管只能显示 4 个 4 bit 数，即一个 16 bit 数，点亮数码管，相当于对数码管对应内存地址进行 store word 操作，只是这个字数据的高 16 位应为0。

首先我们将所有 4 bit 数 0-F 与八段数码管点亮之间的映射关系，提前存储在数据存储器的一片空闲区域内，以便我们能够根据得到的 4 bit 数直接在内存中取出它所对应的八段数码管显示模式。我选取的这一片内存的首地址为 400。

4 bit 数	tube_segment	4 bit 数	tube_segment
0	8'b0011_1111	8	8'b0111_1111
1	8'b0000_0110	9	8'b0110_1111
2	8'b0101_1011	A	8'b0111_0111
3	8'b0100_1111	B	8'b0111_1100
4	8'b0110_0110	C	8'b0011_1001
5	8'b0110_1101	D	8'b0101_1110
6	8'b0111_1101	E	8'b0111_1001
7	8'b0000_0111	F	8'b0111_0001

```

1  li $t0, 400 # first address to store BCD display
2  li $t1, 0x3f # 0: 0x3f
3  sw $t1, 0($t0)
4  li $t1, 0x06 # 1: 0x06
5  sw $t1, 4($t0)
6  ...
7  li $t1, 0x79 # E: 0x79
8  sw $t1, 56($t0)
9  li $t1, 0x71 # F: 0x71
10 sw $t1, 60($t0)

```

显示一个 16 bit 数时，使用动态扫描的方法，每一次只用一个数码管显示一个 4 bit 数，然后循环点亮每一个数码管。将 16 bit 数的 4 个 4 bit 数取出，需要使用“逻辑与+移位”的方法：

```

1      # t1 is the 16 bit data to display
2      andi $s3, $t1, 0xf000
3      srl $s3, $s3, 12 # s3=data[15:12]
4      # find tube display mode in DataMemory
5      sll $s3, $s3, 2
6      add $s3, $s1, $s3 # s1=400
7      lw $s3, 0($s3)
8      addi $s3, $s3, 0x800 # s3={4'b1000, tube_segment[7:0]}

```

每个数据持续显示 1s，我设定扫描频率为 1KHz，也即每个数据在 4 个数码管上循环显示 1000 次，每次持续 1ms，4 个数码管各显示 0.25 ms。流水线 CPU 使用的是系统时钟分频出的 50MHz 时钟，指令执行周期为 20ns。忽略 store 指令的执行时间，我们需要在每一个数码管完成点亮之后（store 写入内存完成）循环执行 $0.25\text{ms}/20\text{ns} = 12500$ 个空指令，因此有如下代码：

```

1      # t2=100
2      # s2=4000_0010
3      select_begin:
4          li $t3, 2500 # 12500/5=2500
5      loop1:
6          sw $s3, 0($s2) # s3={4'b1000, tube_segment[7:0]}
7          nop1:
8              addi $t3, $t3, -1
9              nop
10             nop
11             bnez $t3, nop1
12             ...

```

```

13      li $t3, 2500
14  Loop4:
15      sw $s6, 0($s2) # s6={4'b0001, tube_segment[7:0]}
16      nop4:
17          addi $t3, $t3, -1
18          nop
19          nop
20          bnez $t3, nop4
21
22      addi $t2, $t2, -1
23      bnez $t2, select_begin # if times<0, break

```

考虑到每一个循环中有 `addi` 指令、两个 `nop`、`bnez` 指令及其之后清除的指令，每一次循环占用了 5 个时钟周期，因此循环计数器 $\$t3 = 12500/5 = 2500$ ，如此可实现一个数码管持续点亮 0.25ms，四个数码管交替点亮 1ms，在外层循环中的计数器控制下循环 1000 次，实现了每个数据显示 1s。

四、关键代码

转发单元 `ForwardingUnit.v`：

```

1 // 00: read data from the Register File
2 // 01: forward ALUOUT from EX_MEM to EX
3 // 10: forward ALUOUT from MEM_WB to EX
4 // 11: forward MemReadData from MEM_WB to EX
5 assign ALU_forwardA =
6     (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRs == MEM_RegWrAddr) ? 1
7     :
8     (WB_MemRead && WB_RegWrAddr != 0 && EX_RegRs == WB_RegWrAddr) ? 3 :
9     (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRs == WB_RegWrAddr) ? 2 :
10    0;
11
12 assign ALU_forwardB =
13     (MEM_RegWrite && MEM_RegWrAddr != 0 && EX_RegRt == MEM_RegWrAddr) ? 1
14     :
15     (WB_MemRead && WB_RegWrAddr != 0 && EX_RegRt == WB_RegWrAddr) ? 3 :
16     (WB_RegWrite && WB_RegWrAddr != 0 && EX_RegRt == WB_RegWrAddr) ? 2 :
17    0;
18
19 // 0: read data from the Register File
20 // 1: forward MemReadData from MEM_WB to MEM
21 assign MEM_forward =
22     (WB_MemRead && MEM_MemWrite && WB_RegWrAddr != 0 && MEM_RegRt ==
23     WB_RegWrAddr) ? 1 : 0;

```

控制冒险检测单元 `HazardUnit.v`：

```

1 // when to flush IF: jump hazard, branch hazard
2 assign flush_IF = (ID_PCSrc == 2'b00 || (ID_PCSrc == 2'b01 &&
3 branch_taken == 0)) ? 0 : 1;
4
4 // when to stall IF and ID: load-use hazard, data hazard caused by
5 branch or jump

```

```

5   assign stall_IF_ID =
6     (!ID_MemWrite && EX_MemRead && EX_RegWrAddr != 0 && (EX_RegWrAddr
7 == ID_RegRs || EX_RegWrAddr == ID_RegRt)) // Load-use, except load-store
8     ||
9     (!ID_MemWrite && EX_MemRead && EX_RegWrAddr != 0 && EX_RegWrAddr
10 == ID_RegRs) // sw use the reg that lw writes in for calculating the address
11     ||
12     (
13       ID_PCSrc == 2'b01 // data hazard caused by branch
14       &&
15       (
16         (EX_RegWrite && EX_RegWrAddr != 0 && (EX_RegWrAddr ==
17 ID_RegRs || EX_RegWrAddr == ID_RegRt))
18         ||
19         (MEM_MemRead && MEM_RegWrAddr != 0 && (MEM_RegWrAddr ==
20 ID_RegRs || MEM_RegWrAddr == ID_RegRt))
21       )
22     )
23     ||
24     (
25       ID_PCSrc == 2'b11 // data hazard caused by jr or jalr
26       &&
27       (
28         (EX_RegWrite && EX_RegWrAddr != 0 && EX_RegWrAddr ==
ID_RegRs)
29         ||
30         (MEM_MemRead && MEM_RegWrAddr != 0 && MEM_RegWrAddr ==
ID_RegRs)
31       )
32     )
33   ) ? 1 : 0;

```

分支和跳转转发单元 `BranchJumpForwarding.v` :

```

1 // 0: no forwarding
2 // 1: forward from EX_MEMORY to ID
3 assign Forward1 = (MEM_RegWrite && MEM_RegWrAddr != 0 && ID_RegRs ==
MEM_RegWrAddr) ? 1 : 0;
4 assign Forward2 = (MEM_RegWrite && MEM_RegWrAddr != 0 && ID_RegRt ==
MEM_RegWrAddr) ? 1 : 0;

```

PC 的更新与 `stall_IF_ID` 有关, 与 `flush_IF` 无关:

```

1 always @(`posedge reset or posedge clk)
2   if (reset)
3     IF_PC <= 32'h0;
4   else begin
5     if (stall_IF_ID)
6       IF_PC <= IF_PC;
7     else
8       IF_PC <=
9         (ID_PCSrc == 2'b01) ? BranchAddr :
10        (ID_PCSrc == 2'b10) ? JumpAddr :
11        (ID_PCSrc == 2'b11) ? RegisterAddr : IF_PC + 32'h4;
12   end

```

级间寄存器 IF_ID_Reg.v 更新逻辑：

```

1  always @(`posedge clk or posedge reset) begin
2      if (reset) begin
3          ID_PC <= 32'b0;
4          ID_Instruction <= 32'b0;
5      end
6      else begin
7          if (flush_IF && !stall_IF_ID) begin
8              ID_PC <= 32'b0;
9              ID_Instruction <= 32'b0;
10         end
11         else if (stall_IF_ID) begin
12             ID_PC <= ID_PC;
13             ID_Instruction <= ID_Instruction;
14         end
15         else begin
16             ID_PC <= IF_PC;
17             ID_Instruction <= IF_Instruction;
18         end
19     end
20 end

```

数据存储器的空间划分：

```

1  else if (MemWrite) begin
2      if (Address == 32'h4000_0010) begin
3          tube_select <= write_data[11:8];
4          tube_segment <= write_data[7:0];
5      end
6      else if (Address < 32'h4000_0000) begin
7          RAM_data[Address[RAM_SIZE_BIT + 1 : 2]] <= write_data;
8      end
9  end

```

寄存器堆的先写后读（内部转发）：

```

1 // read data from RF_data as Read_data1 and Read_data2, RAW at the same time
2 assign Read_data1 = (Read_register1 == 5'b00000) ? 32'h00000000 :
3     (RegWrite && Write_register != 0 && (Write_register == Read_register1)) ?
4     Write_data : RF_data[Read_register1];
5 assign Read_data2 = (Read_register2 == 5'b00000) ? 32'h00000000 :
6     (RegWrite && Write_register != 0 && (Write_register == Read_register2)) ?
7     Write_data : RF_data[Read_register2];

```

五、调试情况

(1) 关于级间寄存器的更新，我最开始想当然地认为：级间寄存器在复位或者本级清除的情况下置为0，另外在不阻塞的情况下进行更新，于是对于每个级间寄存器，我都写出了下面这样的代码：

```

1 // ./src/IF_ID_Reg.v
2   always @(posedge clk or posedge reset) begin
3     if (reset || IF_flush) begin
4       ID_Instruction <= 32'b0;
5       ID_PC <= 32'b0;
6     end
7     else if (!IF_stall) begin
8       ID_Instruction <= IF_Instruction;
9       ID_PC <= IF_PC;
10    end
11  end

```

但是进行仿真时我遇到了一个这样的问题：运行单周期处理器大作业时的指令时，出现了 `stall` 信号拉高后，经过一个周期后没有变回0，而是一直保持为高，而且之后每次取的指令都是其中一个分支指令之后的第一条指令。我仔细检查了仿真波形，发现第一个原因：

对于分支指令，在 ID 阶段判断出不执行分支的时候，代码正常顺序执行，不需要清除 IF 级，此时的分支地址应为 `ID_PC + 8` 而不是 `ID_PC + 4`，因为 `ID_PC + 4` 指向的指令已经进入 IF 级了。修改后的分支地址为：

```

1 assign BranchAddr = ID_PC + 32'h0000_0004 + (branch_taken ? {ID_ExtImm[29:0],
2 'b00} : 32'h0000_0004);

```

(2) 解决上面的问题后，`stall` 信号仍一直为高无法复原，于是我找出第二个原因在于级间寄存器的更新逻辑。下面我们考虑流水线阻塞和清除的本质：阻塞是保持该指令及其前级指令，使它们在下一个周期内在同一级重复上一周期的行为，在该指令的下一级产生全零信号来替代它，等价于在该指令前面“凭空插入”一个空指令 `nop`；清除是使得该指令及其前级指令在下一周期进入下一级的时候全部被强制变为0，等价于将该指令变为空指令，消除了它们的影响，从而阻止了它们在流水线中继续执行。

我们考虑下面这样一个比较“极端”的情况：

```

1 lw $t1, 0($t2)
2 beq $t1, $a1, label
3 ...
4 label: ...

```

它出现了多种冒险：load-use、分支提前判断引发的冒险、控制冒险。`beq` 需要阻塞两个周期，并且在分支执行后还需要清除 IF。在此期间阻塞和清除信号均为高，这时下一个周期的各级信号更新情况就比较复杂，我最开始只是考虑单独在阻塞和清除的作用下的更新逻辑，而忽略了两者同时作用。修改后的逻辑见“解决冒险的方法总结”。

(3) 完成代码之后准备上板子进行验证，我最开始写的汇编代码中，点亮数码管的方式是这样的：每次 `store` 指令执行后仅仅将计数器递减和进行分支判断，然后就执行下一个 `store` 指令，也就是每次点亮一个数码管之后立刻点亮下一个。这样数码管仅仅点亮了 5 个时钟周期=100ns，扫描频率高达 2.5MHz，频率太快导致数码管过亮，难以区分显示的数字。后来我查阅资料得知扫描频率不宜过快或过慢，过快会难以辨认，过慢则会闪烁。因此我参照《实验二：反应速度测试仪》将扫描频率降低到 1KHz，在每个 `store` 指令后加入大量的空指令，最终在板子上显示效果较好。

六、仿真结果

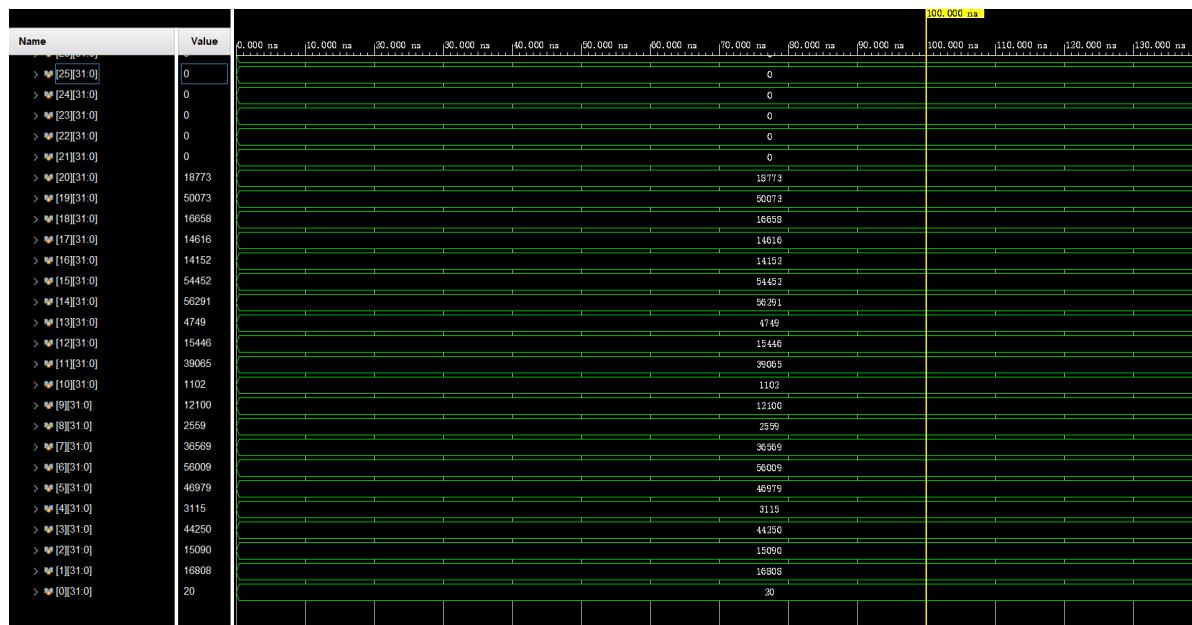
输入文件 `a.in` 中的内容是：

address	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	int32	unsigned	bigendian
00000000	14	00	00	00	a8	41	00	00	f2	3a	00	00	da	ac	00	00	20	16808	15090
00000010	2b	0c	00	00	83	b7	00	00	c9	da	00	00	d9	8e	00	00	3115	46979	56009
00000020	ff	09	00	00	44	2f	00	00	4e	04	00	00	99	98	00	00	2559	12100	1102
00000030	56	3c	00	00	8d	12	00	00	e3	db	00	00	b4	d4	00	00	15446	4749	56291
00000040	48	37	00	00	18	39	00	00	12	41	00	00	99	c3	00	00	14152	14616	16658
00000050	55	49	00	00													18773		50073

理论上排序后的输出文件 `a.out` 为：

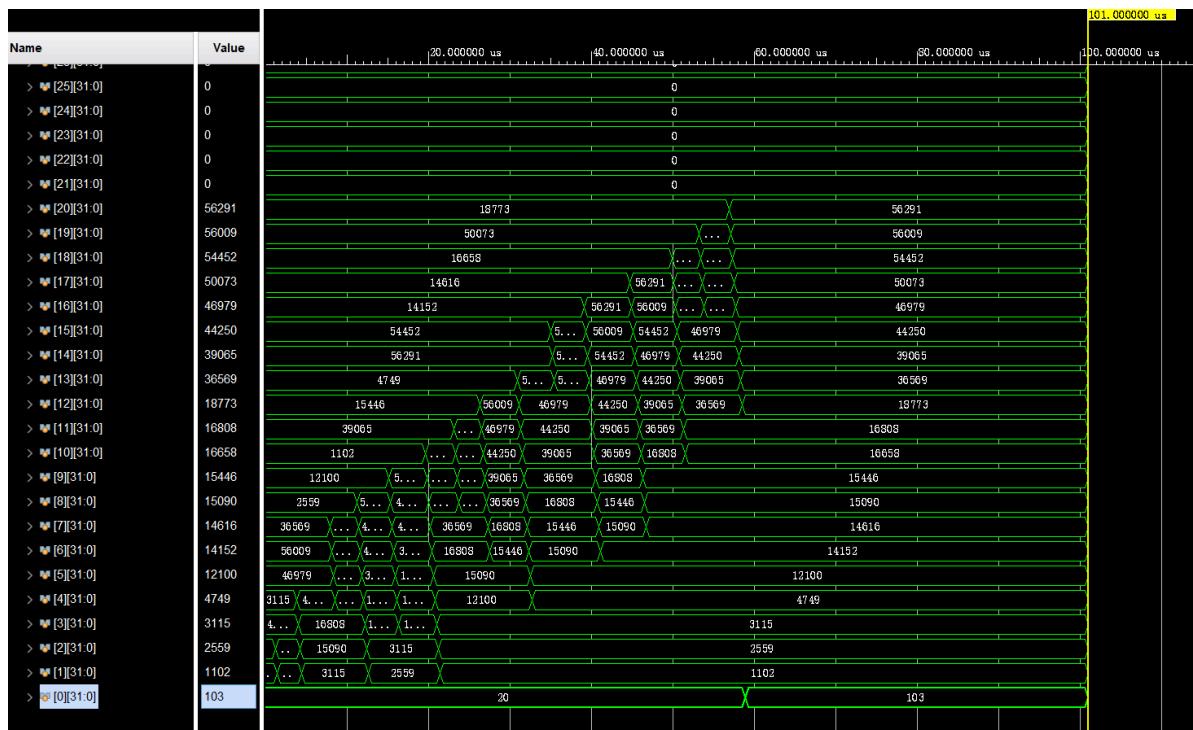
address	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	int32	unsigned	bigendian
00000000	67	00	00	00	4e	04	00	00	ff	09	00	00	2b	0c	00	00	103	1102	2559
00000010	8d	12	00	00	44	2f	00	00	48	37	00	00	18	39	00	00	4749	12100	14152
00000020	f2	3a	00	00	56	3c	00	00	12	41	00	00	a8	41	00	00	15090	15446	16658
00000030	55	49	00	00	d9	8e	00	00	99	98	00	00	da	ac	00	00	18773	36569	39065
00000040	83	b7	00	00	99	c3	00	00	b4	d4	00	00	c9	da	00	00	46979	50073	54452
00000050	e3	db	00	00													56291		56009

0-100ns 排序前的内存，（十进制显示），第一个数20为排序数据个数，从低到高依次为待排序数据：



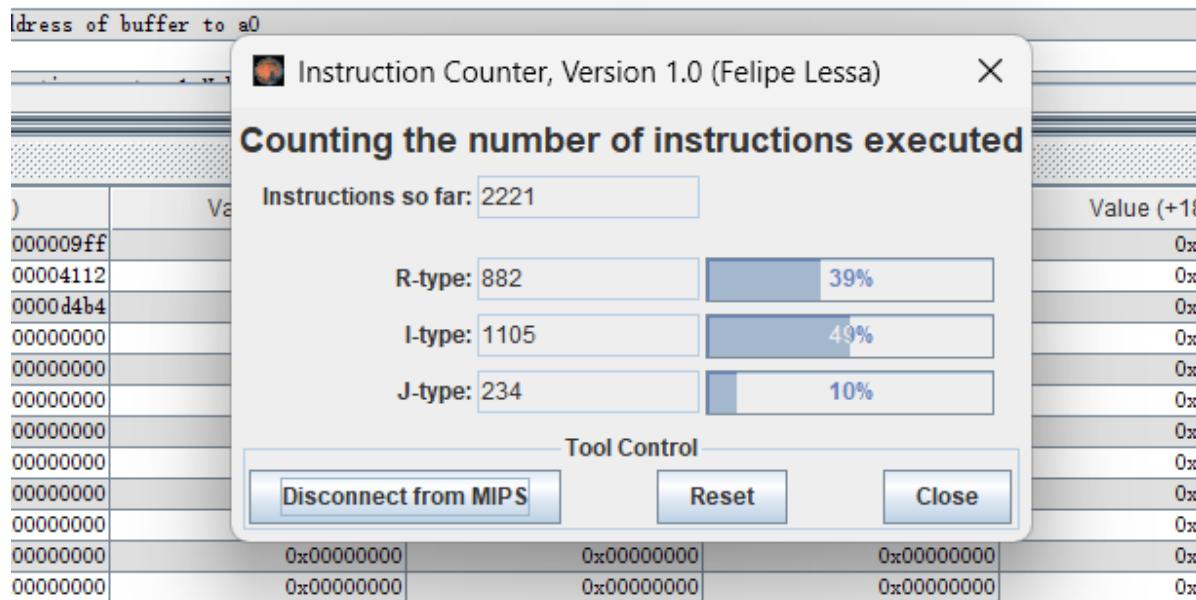
排序后的内存（十进制显示），可见插入排序过程，其中地址 0 处存储的值为排序比较次数

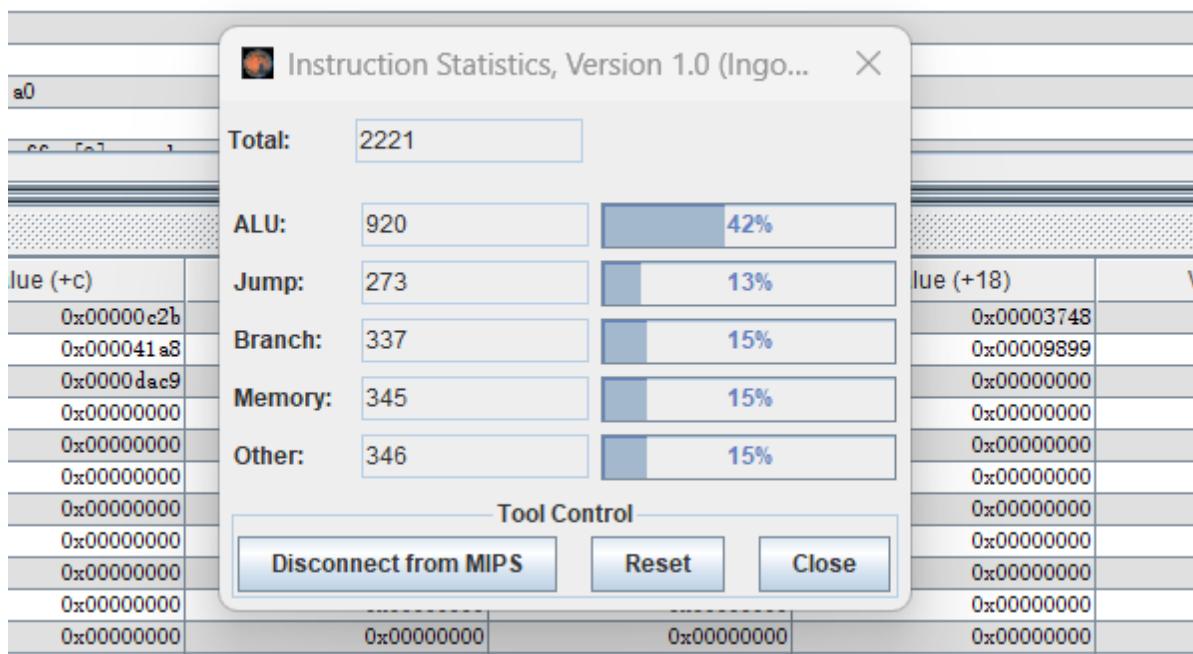
0x67=103，排序后的数据从低到高依次增大排列：



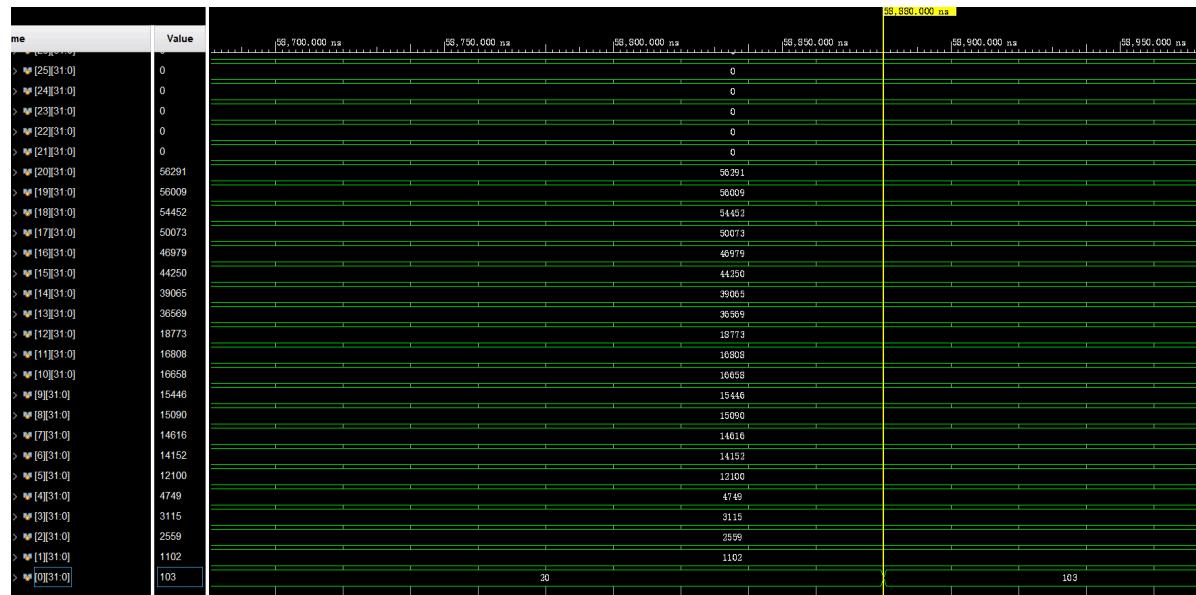
七、CPI 计算

在 MARS 仿真器中运行文件 `insert_sort.asm`，得到指令数为 2221：





仿真所用的代码指令中，排序过程以将比较次数写入内存 0 地址处（`sw $s0, 0($zero)`）为结束，如下图：

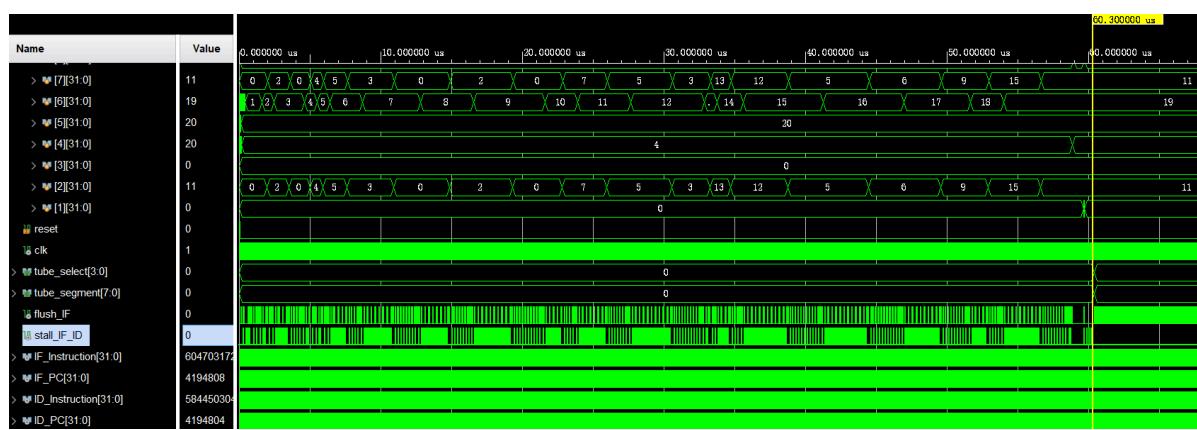


可见内存 0 地址处在 58880ns 时刻存储的数据发生改变，此为 store 指令 MEM 阶段的结束时刻，整个指令完成执行还需要经过一个周期的 WB 阶段，因此排序过程总执行时间为 $58880 + 20 = 58900$ ns，总的周期数为：

$$C = \frac{58900 \text{ ns}}{20 \text{ ns}} = 2945 \quad (1)$$

由此计算出 CPI 为：

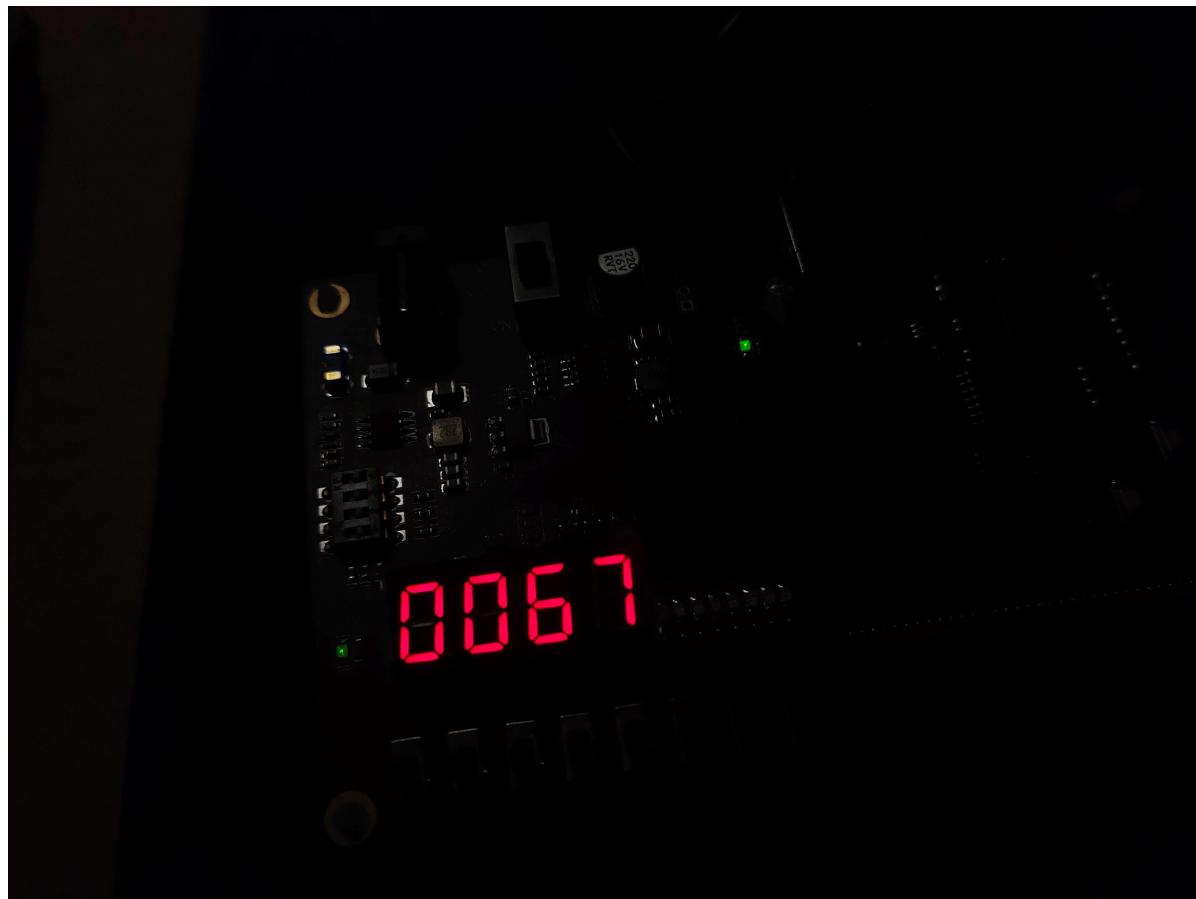
$$\text{CPI} = \frac{2945}{2221} = 1.326 \quad (2)$$



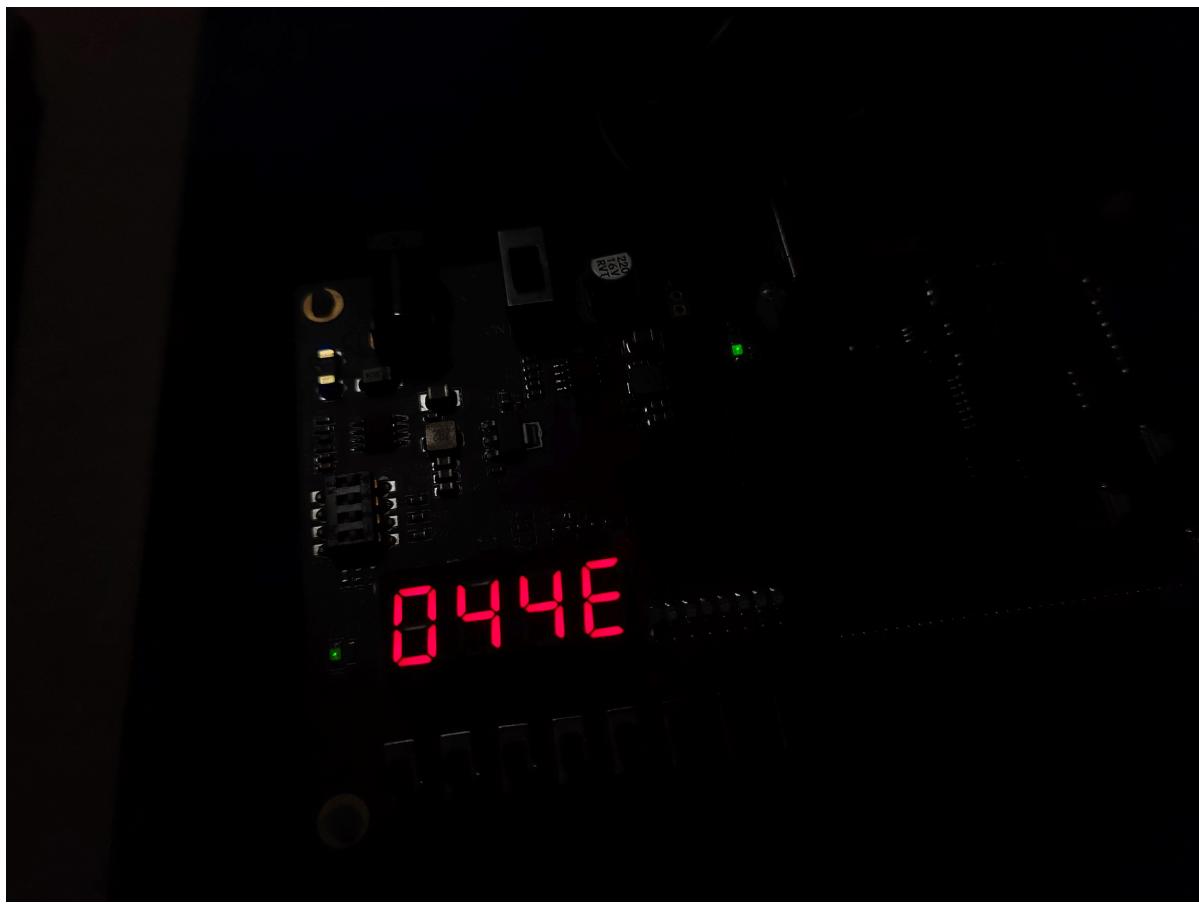
由上图可知，排序过程中阻塞信号 `stall_IF_ID` 和清除信号 `flush_IF` 为高的时间较为密集，主要是由于排序过程中的大量循环造成分支和跳转次数较多，降低了最后的 CPI。

八、FPGA 运行结果

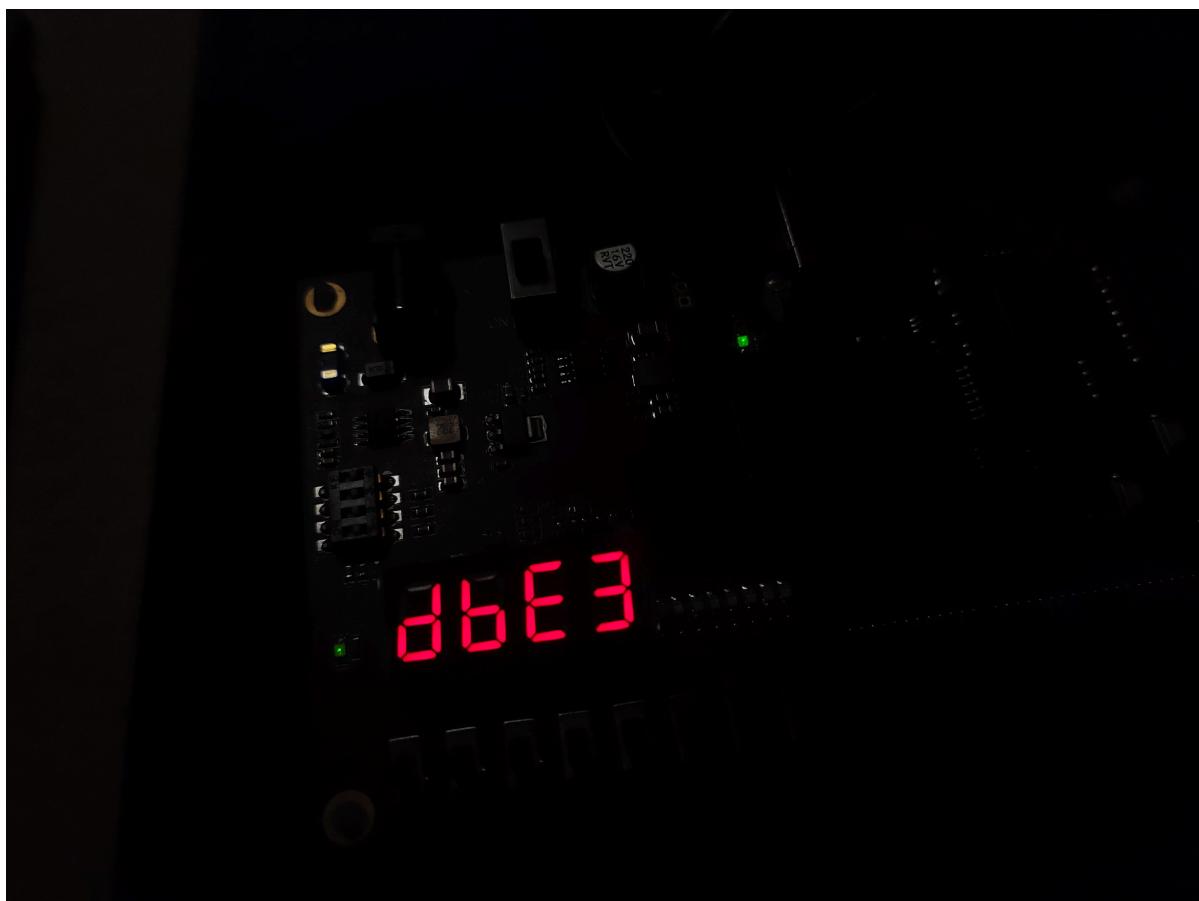
显示第一个数——比较次数：



显示排序后的第一个数据——最小的数据 0x044E：



显示排序后的最后一个数据——最大的数据 0xDBE3：



九、性能分析

静态时序分析

综合后：

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -1.121 ns	Worst Hold Slack (WHS): 0.134 ns	Worst Pulse Width Slack (WPWS): 6.550 ns
Total Negative Slack (TNS): -6.859 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 14	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 35368	Total Number of Endpoints: 35368	Total Number of Endpoints: 17886
Timing constraints are not met.		

实现后：

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.075 ns	Worst Hold Slack (WHS): 0.151 ns	Worst Pulse Width Slack (WPWS): 6.550 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: 35414	Total Number of Endpoints: 35414	Total Number of Endpoints: 17932
All user specified timing constraints are met.		

为了尽可能得到流水线的最高运行频率，我设置时钟周期不断逼近流水线的关键路径延时，“迫使” Vivado 优化以满足时序要求。最终，我将时钟周期设置为 14.1ns 时，综合 (Synthesis) 后略微超出时序要求，但是实现 (Implementation) 之后能够满足时序要求，且时序余量 WNS 仅为 0.075ns，因此最终得到最高时钟频率为：

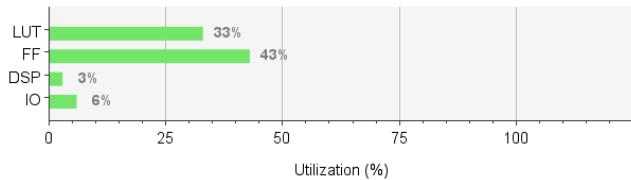
$$f_{max} = \frac{1}{14.1 \text{ ns} - 0.075 \text{ ns}} = 71.30 \text{ MHz} \quad (3)$$

资源使用情况

Hierarchy										
Name	1	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	DSPs (90)	Bonded IOB (250)	BUFGCTRL (32)
CPU	6886	17886	2310	1088	7070	6886	3	14	2	
u_ALU (ALU)	15	0	0	0	4	15	3	0	0	0
u_DataMemory (DataMemory)	4384	16396	2176	1024	6463	4384	0	0	0	0
u_EX_MEM_Reg (EX_MEM_Reg)	975	145	0	0	443	975	0	0	0	0
u_GenerateCLK (GenerateCLK)	1	1	0	0	1	1	0	0	0	0
u_ID_EX_Reg (ID_EX_Reg)	132	153	0	0	141	132	0	0	0	0
u_IF_ID_Reg (IF_ID_Reg)	215	63	0	0	89	215	0	0	0	0
u_InstructionMemory (InstructionMemory)	108	0	6	0	37	108	0	0	0	0
u_MEM_WB_Reg (MEM_WB_Reg)	484	104	0	0	320	484	0	0	0	0
u_PC (PC)	1	32	0	0	23	1	0	0	0	0
u_RegisterFile (RegisterFile)	576	992	128	64	412	576	0	0	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	6886	20800	33.11
FF	17886	41600	43.00
DSP	3	90	3.33
IO	14	250	5.60



十、实验总结

实验的最开始，我就在思考：单周期处理器的结构如何才能变成流水线处理器？单周期在时钟周期内完成一条指令的所有操作，而流水线处理器在一个时钟周期内要使用不同模块完成多个指令的操作。后来我想明白，流水线的核心就是级间寄存器，与单周期相比，级间寄存器实现了指令和控制信号在时间和空间上的分离：同一条指令所对应的信号，在不同周期内有不同名称。流水线中同时存在着五个阶段的信号，他们通过前缀和信号名称区分，这是我对变量命名的考虑——以 `EX_RegWrAddr` 为例，前缀表示这个信号在哪一个模块中被使用，后边的名称 `RegWrAddr` 表示它是一个写入寄存器的编号，而 `MEM_RegWrAddr` 就表示它的上一条指令的写入寄存器编号。等到下一个周期，所有这些信号会通过级间寄存器传递到另一个变量，“改变前缀”，从而在下一个周期为下一个模块所使用，这就实现了“流水”——信号以级间寄存器为起点和终点不断流动。

本实验还给我带来的一个启示是：在开始正式写代码之前，一定要先厘清流水线的结构，要有一个整体的框架、每个阶段和每个模块都需要使用什么信号。我就是前期先开始写这份实验报告，详细分析流水线各阶段的行为、需要添加的控制信号、级间寄存器传递哪些信号、解决冒险的逻辑，此时我思考的层面是在流水线的硬件结构上。又了框架之后，在此基础上写代码效率就比较高。在写代码时我思考的层面是在 Verilog 语言抽象描述硬件的层面上，与之前思考的角度不同，因此可能有一些细节之前没有考虑到，或者实际电路结构和硬件描述语言没有很好的对应，此时我再进行一些细调和修改，最后写出来的程序基本上实现了“一次跑通”。

在本实验中，我实现了分支指令在 ID 阶段提前进行判断，一方面是考虑到可以减少控制冒险发生时阻塞的周期数，减少运行的周期数从而提高 CPI，另一方面是由于跳转指令 `jr`, `jalr` 本身就在 ID 阶段读取寄存器计算跳转地址，仅仅转发到 EX 级无法解决问题，需要考虑新的数据冒险类型。如果将分支提前到 ID 阶段判断，就可以使分支指令和跳转指令可以共用一套阻塞、转发和清除逻辑，更具有统一性。当数据需要转发到 ID 阶段时，用于分支判断的两个寄存器数据中的第一个，也作为跳转至寄存器指令需要读取的数据可以直接复用。另外，提前分支判断也造成了额外的比较单元、控制信号等电路，可能会延长 ID 阶段的关键路径而影响最高时钟频率。

要善于利用仿真。Vivado 提供的仿真波形对 debug 较为友好，我遇到问题时，常常先找出是哪一条指令结果出现了错误，如果是这条指令本身的执行没有问题，而是获取了错误的数据导致结果出错，那么就继续向前追溯最早产生错误数据的指令，如果是这条指令本身的问题，那么就观察控制信号、阻塞清除等是否正确。对于测试用的排序算法，我曾经遇到一个这样的问题：内存地址 0 处原本为排序数据个数 20，但是经过一段时间后内存几个地址处数据全变成了 20，于是我在汇编代码中找到所有 `store` 指令，发现是 `load-store` 转发的问题，我原本在冒险检测单元中考虑到这种情况，它不需要阻塞 `store` 指令，只需要转发到 MEM 阶段，但是它满足 `load-use` 阻塞的条件，因此要在 `load-use` 阻塞逻辑中排除掉这种情况。

最后，我独立完成了实验内容，除了参考和复用单周期处理器大作业中 `DataMemory.v`, `InstructionMemory.v`, `ALU.v`, `ALUControl.v`, `RegisterFile.v` 以外，其他的转发、控制冒险、分支判断转发、级间寄存器等均为自己完成，然后完全重写顶层文件 `CPU.v`，最后手动将数码管显示翻译为汇编语言。流水线大作业是对理论课知识和实验课硬件编程的一次高强度考验，虽然我耗时许久，但最终找出所有 bug 并成功显示正确结果，丰富了我的硬件开发和调试经验，较有收获感。

附录、文件清单

```

1 .
2 |-- LICENSE
3 |-- README.md
4 |-- assembly
5 |   |-- Mars4_5.jar
6 |   |-- a.in
7 |   |-- a.out
8 |   |-- convert.txt
9 |   |-- default.txt
10 |  |-- insert_sort.asm
11 |  |-- insert_sort.cpp
12 |  |-- instruction_convert.py
13 |  |-- mem_data.txt
14 |  `-- test.asm
15 |-- docs
16 |   |-- reference
17 |   |   |-- Introduction to the MIPS32 Architecture.pdf
18 |   |   |-- MIPS Calling Conventions Summary.pdf
19 |   |   `-- MIPS32 Instruction Set Manual.pdf
20 |   |-- requirements.pdf
21 |   |-- singlecycleCPU_2024.pdf
22 |   `-- singlecycle_datapath.pptx
23 |-- prj
24 |   |-- simulation
25 |   |   `-- icarus
26 |   |       `-- out.vvp
27 |   `-- xilinx
28 |-- report
29 |   |-- 4bit_tube.png
30 |   |-- a_in.png
31 |   |-- a_out.png
32 |   |-- frame.png
33 |   |-- imp_timing.png
34 |   |-- imp_uzi1.png
35 |   |-- imp_uzi2.png
36 |   |-- impl_timing.png
37 |   |-- impl_utilization1.png
38 |   |-- impl_utilization2.png
39 |   |-- inst_counter.png
40 |   |-- inst_statistics.png
41 |   |-- memory_space.png
42 |   |-- pipeline_stages.png
43 |   |-- report.md
44 |   |-- report.pdf
45 |   |-- sim_after_sort.png

```

```

46 |   |-- sim_before_sort.png
47 |   |-- sim_flush.png
48 |   |-- sim_sort_end.png
49 |   |-- syn_timing.png
50 |   |-- tube0067.jpg
51 |   |-- tube044E.jpg
52 |   `-- tubeDBE3.jpg
53 `-- user
54     |-- data
55     |   '-- top.xdc
56     |-- sim
57     |   '-- testbench.v
58     '-- src
59         |-- ALU.v
60         |-- ALUControl.v
61         |-- BranchJumpForwarding.v
62         |-- BranchResolve.v
63         |-- CPU.v
64         |-- Control.v
65         |-- DataMemory.v
66         |-- EX_MEMORY_Reg.v
67         |-- ForwardingUnit.v
68         |-- GenerateCLK.v
69         |-- HazardUnit.v
70         |-- ID_EX_Reg.v
71         |-- IF_ID_Reg.v
72         |-- InstructionMemory.v
73         |-- MEM_WB_Reg.v
74         |-- PC.v
75         '-- RegisterFile.v

```

MIPS流水线处理器综合实验报告

一、实验内容

二、实验要求

三、实验设计

支持的指令集

控制信号

级间寄存器

数据冒险

Case 1: EX_MEMORY to EX

Case 2: MEM_WB to EX

Case 3: load-use

Case 4: load-store

控制冒险

分支指令冒险

跳转指令冒险

解决冒险的方法总结

总体设计

IF stage

ID stage

EX stage

MEM stage

WB stage

- 外设部分
- 四、关键代码
- 五、调试情况
- 六、仿真结果
- 七、CPI 计算
- 八、FPGA 运行结果
- 九、性能分析
 - 静态时序分析
 - 资源使用情况
- 十、实验总结
- 附录、文件清单