深 圳 大 学 实 验 报 告

课 程 名 称：计算机系统(3)

实验项目名称：新增指令实验

学　　院：计算机与软件学院

专　　业：软件工程（腾班）

指 导 教 师：王　毅

报告人：黄亮铭　学号：2022155028　班级：腾班

实 验 时 间：2024 年 12 月 13 日

实验报告提交时间：2025 年 1 月 3 日

教务处制

一、 实验目标

了解 RISC-V mini 处理器架构，在其基础之上新增一个指令，完成设计并观察指令执行。

二、实验内容

1）修改数据通路，新增指令 comb rs1,rs2,rd（采用 R 型指令格式），实现将 rs1 高 16 位和 rs2 低 16 位拼接成 32 位整数，并且保存到 rd 寄存器。

2）在处理器上执行该指令，观察仿真波形，验证功能是否正确。

3）自行设计其他功能指令，并验证设计是否正确。

三、实验环境

硬件：桌面 PC

软件：Chisel 开发环境

四、实验步骤及说明

学习 Chisel 数据通路的 Chisel 描述，特别是指令译码部分和 core 核心代码。然后按照下面指引完成指令译码器的修改，以及数据通路的修改。

分析：添加新指令 comb ，首先需要根据 riscv 指令格式，设置该指令各个字段的值，并在相应文件中添加该指令的比特模式。然后设置该指令的译码结果，接着在 ALU 中实现该指令的功能。最后让该指令在处理器上执行，验证功能是否正确。以下操作均在实验六下载下来的 riscv-mini 项目工程上进行。

1. 添加 comb 指令比特模式串

comb 为 R 型指令，riscv 的 R 型指令格式如图 1 所示：

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

图 1 riscv 指令格式

为了避免新加指令与 riscv-mini 已有指令冲突，这里我们将 comb 指令的 opcode、funct3 和 funct7 部分设置为 0110011、111、0000001。然后使用 BitPat() 函数设置 comb 指令的比特模式。请在 Instrutcions.scala 中添加 comb 指令比特模式串，并截图。

```
52  def CSRRCI = BitPat("b?????????????????111?????1110011")
53  // Change Level
54  def ECALL = BitPat("b00000000000000000000000001110011")
55  def EBREAK = BitPat("b00000000000100000000000001110011")
56  def ERET = BitPat("b00010000000000000000000001110011")
57  def WFI = BitPat("b00010000001000000000000001110011")
58
59  def NOP = BitPat.bitPatToUInt(BitPat("b00000000000000000000000000010011"))
70
71  // 新增指令comb
72  def COMB = BitPat("b0000001??????????111?????0110011")
73 }
```

图 2 在 Instructions.scala 中添加 comb 指令比特模式串

2. 添加 comb 指令的译码

comb 指令需要在 ALU 中将 rs1 高 16 位和 rs2 低 16 位拼接成 32 位整数，因此需要在 Alu.scala 文件中添加常量 ALU_COMB ，让译码器可以译码出正确的信号。请在 Alu.scala 中添加该常量，并截图。

```
object Alu {
  val ALU_ADD = 0.U(4.W)
  val ALU_SUB = 1.U(4.W)
  val ALU_AND = 2.U(4.W)
  val ALU_OR = 3.U(4.W)
  val ALU_XOR = 4.U(4.W)
  val ALU_SLT = 5.U(4.W)
  val ALU_SLL = 6.U(4.W)
  val ALU_SLTU = 7.U(4.W)
  val ALU_SRL = 8.U(4.W)
  val ALU_SRA = 9.U(4.W)
  val ALU_COPY_A = 10.U(4.W)
  val ALU_COPY_B = 11.U(4.W)
  val ALU_XXX = 15.U(4.W)
  // 新增常量
  val ALU_COMB - 12.U(4.W)
}
```

图 3　在 Alu.scala 中添加 ALU_COMB 常量

接下来在 Control.scala 文件中为 comb 指令添加对应的译码映射。comb 指令执行后 pc 需要加 4，并将从寄存器文件中读取的数据 rs1 和 rs2 进行拼接操作，然后将 ALU 输出的拼接结果写回到寄存器文件中。请在 Control.scala 中添加 comb 指令的译码映射，并截图。

```
    WFI   -> List(PC_4  , A_XXX,  B_XXX, IMM_X, ALU_XXX   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.N, N),
    // format: on
    // comb指令的译码映射
    COMB  -> List(PC_4  , A_RS1,  B_RS2, IMM_X, ALU_COMB  , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.N, N))
}
```

图 4　在 Control.scala 中添加 comb 指令的译码映射

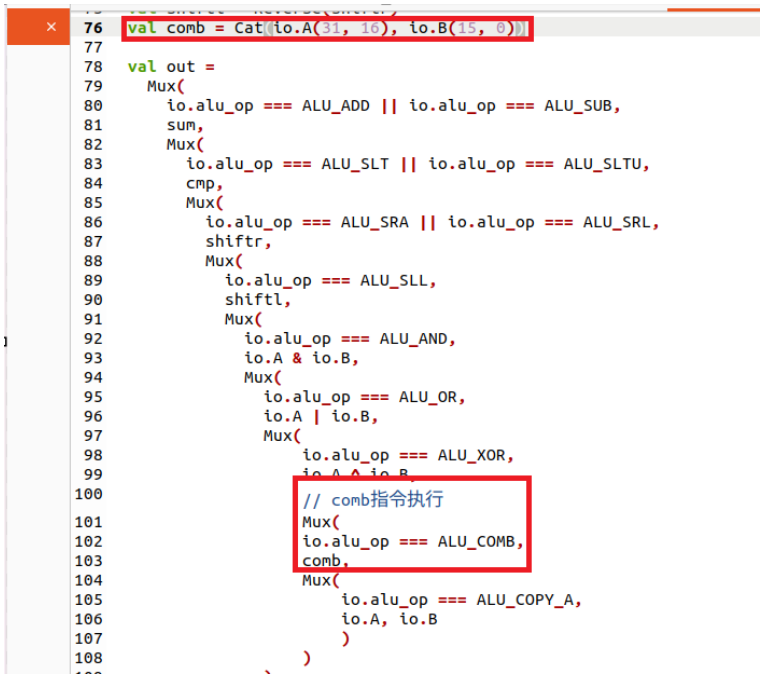
3. 实现 comb 指令的执行操作

在 Alu.scala 文件添加将 rs1 高 16 位和 rs2 低 16 位拼接成 32 位整数的操作。请在 Alu.scala 中实现该操作，并截图。

```
  val shamt = io.B(4, 0).asUInt

  io.out := MuxLookup(io.alu_op, io.B)(
    Seq(
      ALU_ADD -> (io.A + io.B),
      ALU_SUB -> (io.A - io.B),
      ALU_SRA -> (io.A.asSInt >> shamt).asUInt,
      ALU_SRL -> (io.A >> shamt),
      ALU_SLL -> (io.A << shamt),
      ALU_SLT -> (io.A.asSInt < io.B.asSInt),
      ALU_SLTU -> (io.A < io.B),
      ALU_AND -> (io.A & io.B),
      ALU_OR -> (io.A | io.B),
      ALU_XOR -> (io.A ^ io.B),
      ALU_COPY_A -> io.A,
      // 新增指令执行
      ALU_COMB -> Cat(io.A(31, 16), io.B(15, 0))
    )
  )
```

图 5　在 Alu.scala 中的 AluSimple 添加 comb 指令执行的逻辑



```
76  val comb = Cat(io.A(31, 16), io.B(15, 0))
77
78  val out =
79    Mux(
80      io.alu_op === ALU_ADD || io.alu_op === ALU_SUB,
81      sum,
82      Mux(
83        io.alu_op === ALU_SLT || io.alu_op === ALU_SLTU,
84        cmp,
85        Mux(
86          io.alu_op === ALU_SRA || io.alu_op === ALU_SRL,
87          shiftr,
88          Mux(
89            io.alu_op === ALU_SLL,
90            shiftl,
91            Mux(
92              io.alu_op === ALU_AND,
93              io.A & io.B,
94              Mux(
95                io.alu_op === ALU_OR,
96                io.A | io.B,
97                Mux(
98                    io.alu_op === ALU_XOR,
99                    io.A ^ io.B,
100                   // comb指令执行
101                   Mux(
102                   io.alu_op === ALU_COMB,
103                   comb,
104                   Mux(
105                       io.alu_op === ALU_COPY_A,
106                       io.A, io.B
107                       )
108                   )
```

图 6　在 Alu.scala 中的 AluArea 添加 comb 指令执行的逻辑和多选器逻辑

4. 对 comb 指令进行测试

参照实验六第二部分第一节，尝试编写一个汇编程序 comb.s，对 comb 指令进行测试。这里给出一个样例代码。在下面代码中，x6 寄存器中的数据为 0x00001000，x7 寄存器中的数据为 0x00002000，comb 命令将 x6 的高 16 位（即 0x0000）和 x7 的低 16 位（即 0x2000）拼接成一个 32 位的整数（即 0x00002000）。

```
1.          .text # Define beginning of text section
2.          .global _start # Define entry _start
3.  _start:
4.          lui x6, 1 # x6 = 0x00001000
5.          lui x7, 2 # x7 = 0x00002000
6.          # comb x5, x6, x7
7.  exit:
8.          csrw mtohost, 1
9.          j exit
10.         .end # End of file
```

我随机生成了一个随机数 0x12345678，装载进 x6 寄存器中，因为是 lui 的功能是把一个 20 位的立即数装载进寄存器的高 20 位。因此，在编写汇编代码的时候可以将 0x12345678 写成 0x45678（满足该数大于 0xFFFF 的要求）；我的学号为 2022155028，16 进制表示为 0x7887a314，与上同理，在编写汇编代码的时候可以将 0x7887a314 写成 0x7a314。

```
Control.scala                                    Alu.scala
1 .text # Define beginning of text section
2 .global _start # Define entry _start
3 _start:
4     lui x6, 0x45678 # x6 = 0x45678000
5     lui x7, 0x7a314 # x7 = 0x7a314000 2022155028 = 0x7887a314
6     # comb x5, x6, x7
7 exit:
8     csrw mtohost, 1
9     j exit
10 .end # End of file
```

请注意，因为 comb 为自己加入的指令，不能被汇编器汇编，所以这里先将其注释掉，到后面生成的 comb.hex 文件中再将 comb x5, x6, x7 的二进制添加进去。

编写完程序后，使用 riscv32-unknown-elf-gcc 编译得到 comb 二进制文件：

我的编译指令：

```
riscv32-unknown-elf-gcc -nostdlib -Ttext=0x200 -o comb comb.s
```

然后使用 elf2hx 将编译得到的 comb 二进制文件转换成 16 进制文件 comb.hex：

我的转换指令：

```
elf2hex 16 4096 comb > comb.hex
```

执行完上述两条指令之后，对应文件夹的目录如下，可以看到对应文件已经生成。

```
huangliangming_2022155028@ubuntu-2204:~/riscv-mini$ riscv32-unknown-elf-gcc -nos
tdlib -Ttext=0x200 -o comb comb.s
huangliangming_2022155028@ubuntu-2204:~/riscv-mini$ elf2hex 16 4096 comb > comb.
hex
huangliangming_2022155028@ubuntu-2204:~/riscv-mini$ ls
build-riscv-tools.sh  custom-bmark  factorial.hex  project     test.hex
build.sbt             diagram.pdf   factorial.vcd  README.md   test.s
comb                  diagram.png   generated-src  src         tests
comb.hex              factorial     LICENSE        target      test.vcd
comb.s                factorial.c   Makefile       test        VTile
huangliangming_2022155028@ubuntu-2204:~/riscv-mini$ S
```

在 comb.hex 文件中，可以找到 lui x6, 1 和 lui x7, 2 的机器码对应的十六进制形式，如图 7 所示：

```
24 00000000000000000000000000000000
25 00000000000000000000000000000000
26 00000000000000000000000000000000
27 00000000000000000000000000000000
28 00000000000000000000000000000000
29 00000000000000000000000000000000
30 00000000000000000000000000000000
31 00000000000000000000000000000000
32 00000000000000000000000000000000
33 ffdff06f7800d0737a3143b745678337S
34 00000000000000000000000000000000
35 00000000000000000000000000000000
36 00000000000000000000000000000000
37 00000000000000000000000000000000
38 00000000000000000000000000000000
39 00000000000000000000000000000000
```

图 7　在 comb.hex 中找到 lui x6,1 和 lui x7,2 的十六进制机器码

comb x5, x6, x7 转换成机器码的十六机制形式为 027372b3。因此处指令存储为小端模式，故我们需要将十六进制数插入到第一个红线的前面。修改后如图 8 所示：

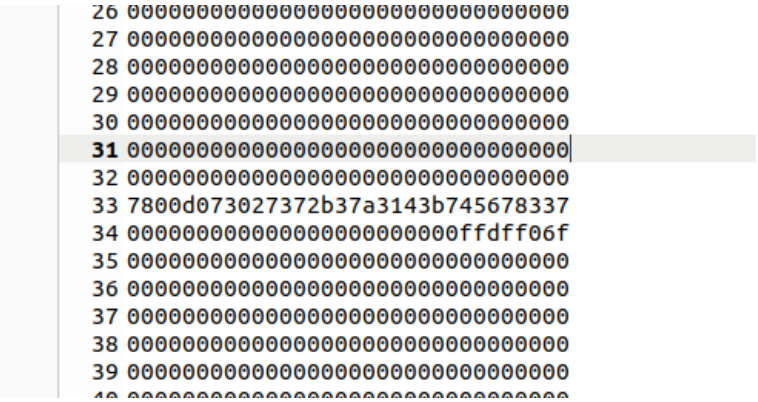


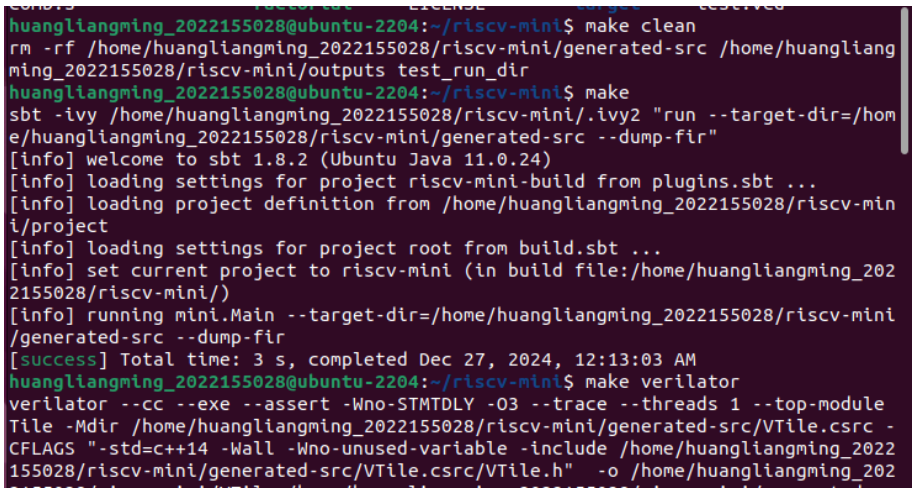
图 8　将 comb x5, x6, x7 汇编的十六进制机器码插入到 comb.hex 中

接着需要在主目录下一次执行 make 和 make verilator 命令（若之前已经执行过，则在此次操作之前需要执行 make clean），执行后会产生 VTile 可执行文件。然后执行下面命令，使 mini 处理器执行新建指令并产生波形文件。

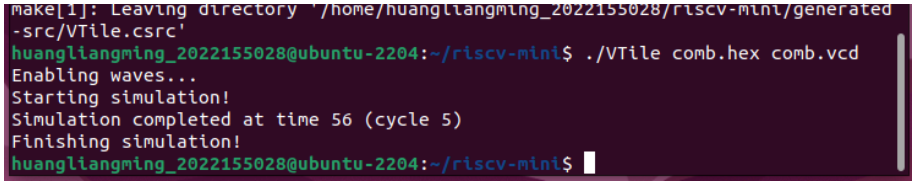
我执行了如下命令：

```
make clean
make
make verilator
```

执行结果如下图所示，说明命令运行成功。



使用 GTKWave 打开 comb.vcd 文件（对应指令为./VTile comb.hex comb.vcd，gtkwave comb.vcd），执行指令结果如下图所示。其波形图如图 9 所示。



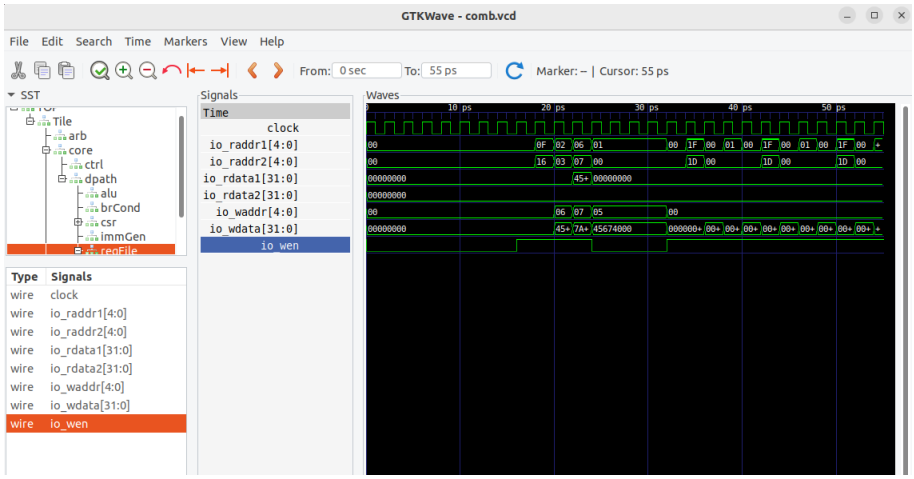


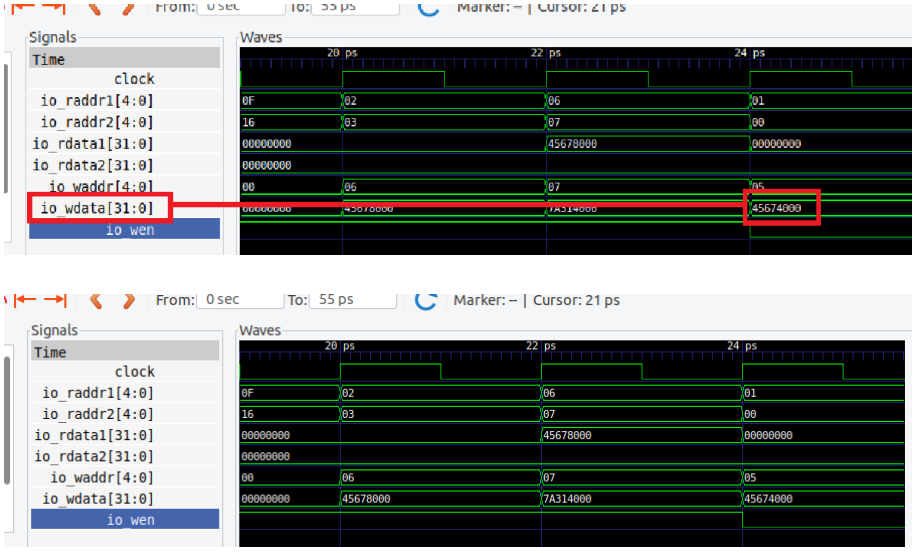
图 9　comb 测试程序的波形图

指令对应的十六进制形式见表 1 所示：

表 1　核心指令对应的十六进制形式

指令	十六进制表示	说明
lui x6, 1	45678337	x6 = 0x45678000
lui x7, 2	7a31423b7	x7 = 0x7a314000
comb x5, x6, x7	027372b3	x5 = cat(x6(31:16), x7(15:0))

从波形图中可以看出， comb 指令将拼接后的结果 0x45674000 写回到了 5 号寄存器中，故该指令执行正常。



可以看到，io_wdata 分别获得 x6 和 x7 的值，然后将其拼接，最后写入 x5 中。

五、实验结果

1)　本次实验我成功地添加了一条新增指令，用于将两个寄存器的数值进行拼接。

2)　本次实验，我在处理器上执行该指令，观察仿真波形，验证功能的正确性。

六、实验总结与体会

1)　在本次实验中，通过对数据通路的修改和指令新增，我更深入地理解了 RISC-V 处

理器的架构。

2) 通过观察仿真波形图，我学会了如何验证和调试新增的指令。
3) 通过对数据通路的修改，我了解了数字逻辑设计的基本原理。
4) 通过本次实验，我理解了如何在处理器中插入新指令。

本次实验我将文档中给定的 Terminal 图替换成我自己的 Terminal 图，图中的绿色的用户名为我的姓名的拼音+学号，即 huangliangming_2022155028。

指导教师批阅意见： **成绩评定：** 指导教师签字：　王毅 年　月　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
　　2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。