实验七 新增指令实验

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。

三、实验环境

- Ubuntu20.4
- chisel3.5.1

四、实验步骤及说明

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。

1. 在**Instrutcions.scala**文件中添加 `comb` 指令比特模式串

`comb` 为R型指令，riscv的R型指令格式如下：

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

为了避免新加指令与riscv-mini已有指令冲突，这里我将 `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。具体代码如下

```
// Instructions.scala
package min

import chisel3.util.BitPat

object Instructions {
  /*
   * 此处省略RISCV-mini已定义的指令
   */
  // 新增指令COMB
  def COMB = BitPat("b0000001??????????111????0110011")
}
```

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。具体代码如下：

```
// Alu.scala
package mini

import chisel3._
import chisel3.util._

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)
    // 新加常量
```

```
    val ALU_COMB = 12.U(4.W)
    val ALU_XXX = 15.U(4.W)
}
/*
 * 此处省略RISCV-mini的ALU实现部分
 */
```

接下来为`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。在**Control.scala**文件中添加的具体代码如下：

```
// Control.scala
package mini

import chisel3._
import chisel3.util._

object Control {
    /*
     * 此处省略常量定义部分
     */
    // format: off
    val default =


    //                                                                 kill                        wb_en  illegal

    //             pc_sel  A_sel   B_sel  imm_sel   alu_op   br_type  |  st_type ld_type wb_sel  | csr_cmd |
    //               |       |       |     |          |          |    |     |       |       |    |  |      |
              List(PC_4  , A_XXX,  B_XXX, IMM_X, ALU_XXX   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.N, Y)
    val map = Array(
      LUI   -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_COPY_B, BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.N, N),
      AUIPC -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_ADD   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.N, N),
      JAL   -> List(PC_ALU, A_PC,   B_IMM, IMM_J, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.N, N),
      /*
       * 此处省略部分指令译码映射
       */

      // 这是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))

    // format: on
}
/*
 * 此处省略RISCV-mini的控制实现部分
 */
```

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，具体代码如下：

```
// Alu.scala
package mini

import chisel3._
import chisel3.util._
/*
 * 此处省略Alu常量定义和端口声明部分
 */
class AluSimple(val width: Int) extends Alu {
  val io = IO(new AluIO(width))

  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,
      // COMB指令执行
```

```
      ALU_COMB -> Cat(io.A(31,16), io.B(15,0))
    )
  )

  io.sum := io.A + Mux(io.alu_op(0), -io.B, io.B)
}

class AluArea(val width: Int) extends Alu {
  val io = IO(new AluIO(width))
  val sum = io.A + Mux(io.alu_op(0), -io.B, io.B)
  val cmp =
    Mux(io.A(width - 1) === io.B(width - 1), sum(width - 1), Mux(io.alu_op(1), io.B(width - 1), io.A(width -
1)))
  val shamt = io.B(4, 0).asUInt
  val shin = Mux(io.alu_op(3), io.A, Reverse(io.A))
  val shiftr = (Cat(io.alu_op(0) && shin(width - 1), shin).asSInt >> shamt)(width - 1, 0)
  val shiftl = Reverse(shiftr)

  // 将A(rs1)的高16位与B(rs2)的低16位拼接
  val comb = Cat(io.A(31,16), io.B(15,0))

  val out =
    Mux(
      io.alu_op === ALU_ADD || io.alu_op === ALU_SUB,
      sum,
      Mux(
        io.alu_op === ALU_SLT || io.alu_op === ALU_SLTU,
        cmp,
        Mux(
          io.alu_op === ALU_SRA || io.alu_op === ALU_SRL,
          shiftr,
          Mux(
            io.alu_op === ALU_SLL,
            shiftl,
            Mux(
              io.alu_op === ALU_AND,
              io.A & io.B,
              Mux(
                io.alu_op === ALU_OR,
                io.A | io.B,
                Mux(io.alu_op === ALU_XOR, io.A ^ io.B,
                  // COMB指令执行
                  Mux(io.alu_op === ALU_COMB, comb, Mux(io.alu_op === ALU_COPY_A, io.A, io.B)))
              )
            )
          )
        )
      )
    )

  io.out := out
  io.sum := sum
}
```

3. 对 `comb` 指令进行测试

首先创建**comb.s**文件，编写如下的汇编程序：

```
                .text # Define beginning of text section


                .global _start # Define entry _start
    _start:
                lui x6, 1 # x6 = 0x00001000
                lui x7, 2 # x7 = 0x00002000
                # comb x5, x6, x7
    exit:
                csrw mtohost, 1
                j exit
                .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。

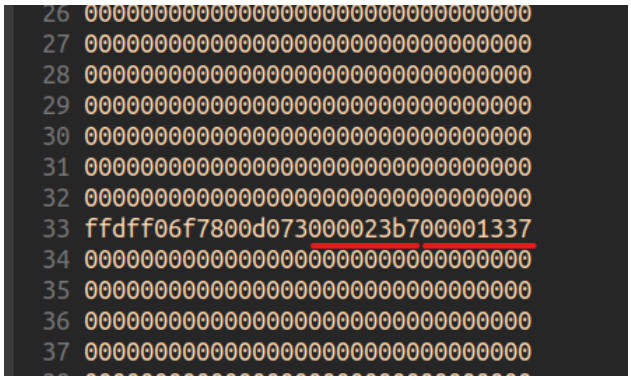
编写完程序后，使用如下命令进行编译：

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

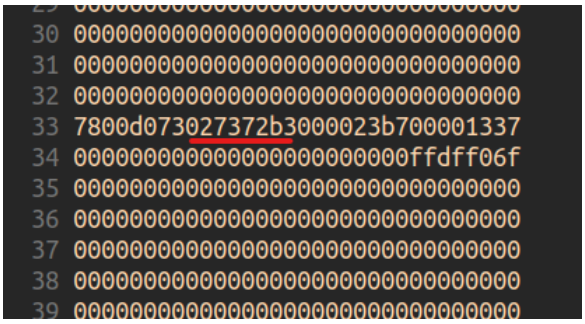
然后使用 `elf2hx` 命令将**comb**二进制文件转换成十六进制：

> $ elf2hex 16 4096 comb > comb.hex

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：



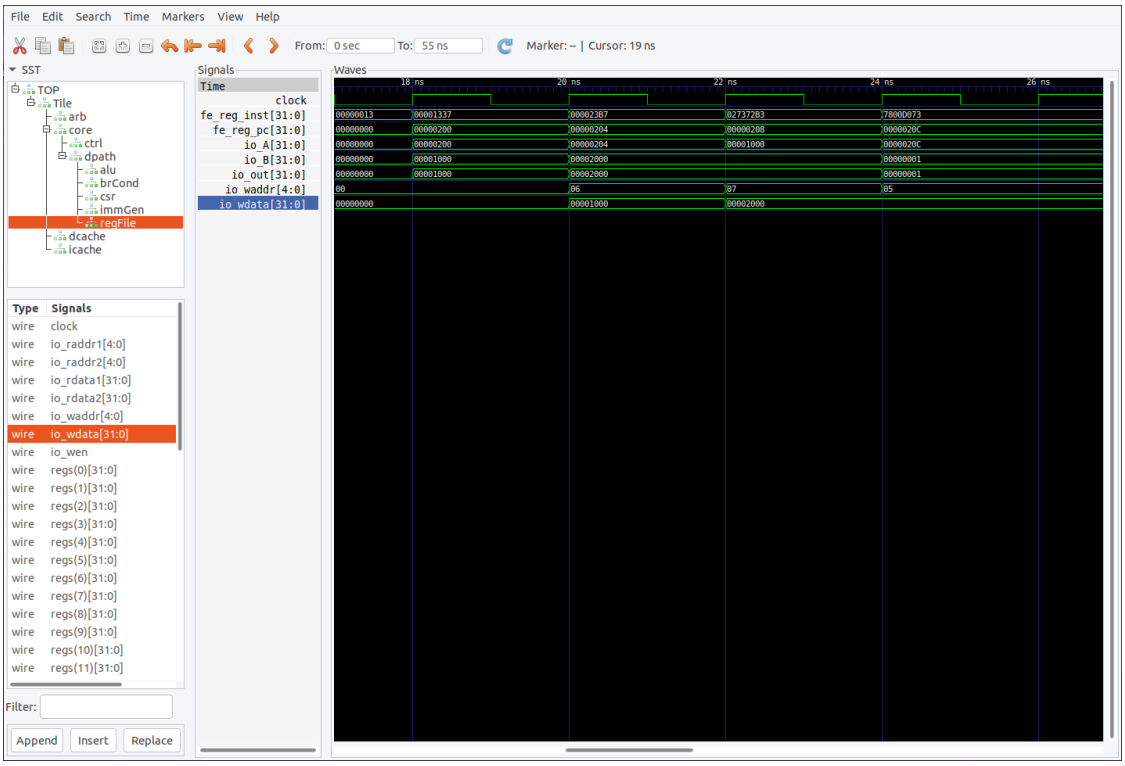
`comb x5, x6, x7`转换成机器码的十六机制形式为 027372b3。因此处指令存储为小端模式，故我们需要将十六进制数插入到第一个红线的前面。修改后如下：



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。

> $ ./VTile comb.hex comb.vcd

使用GTKWave打开**comb.vcd**文件，其波形图如下：



指令对应的十六进制形式见下表：

指令	十六进制形式	说明
lui x6, 1	00001337	x6 = 0x00001000
lui x7, 2	000023b7	x7 = 0x00002000
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0x00002000写回到了5号寄存器中，故该指令执行正常。