



Imagination大学计划

# RVfpga实验18

添加新功能：指令和计数器

## 1. 简介

本实验将应用在先前实验中获得的知识来修改SweRV EH1处理器，向其添加以下新功能：

- **添加A-L指令：**通过RISC-V架构中提供的全新位操作扩展来添加算术逻辑指令。
- **添加浮点指令：**添加三条浮点指令：加、乘、除。然后使用这些指令进行二分法计算。
- **添加计数器：**添加一个新的硬件计数器，用于计算执行的I型指令数量。

在一些练习中，我们将指导您完成修改内核的过程，在其他练习中，您必须自行确定所需的操作。

## 2. 练习

- 1) 位操作（*bitmanip*）扩展由基本RISC-V架构的数个组件扩展组成，旨在缩减代码长度、改进性能并降低能耗。可访问<https://github.com/riscv/riscv-bitmanip>获取完整的规范。文件<https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>详细介绍了该扩展包含的所有指令。

在本练习中，您将在SweRV EH1处理器中添加一个来自*bitmanip*扩展的新指令。具体而言，您将添加**minu**指令，该指令会将rs1和rs2中较小的一个无符号整数放入rd。该指令使用的格式如下图所示。

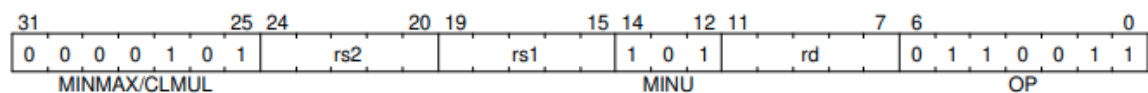


图1. minu指令使用的格式

（图片来源：<https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>）

要加入新的算术逻辑指令，必须修改处理器的两个主要部分：**控制单元**和**执行单元**。图2中以红色突出显示了添加minu指令前必须在上述两个单元中修改的结构（请记住，此图最初用作实验11中的图4）。

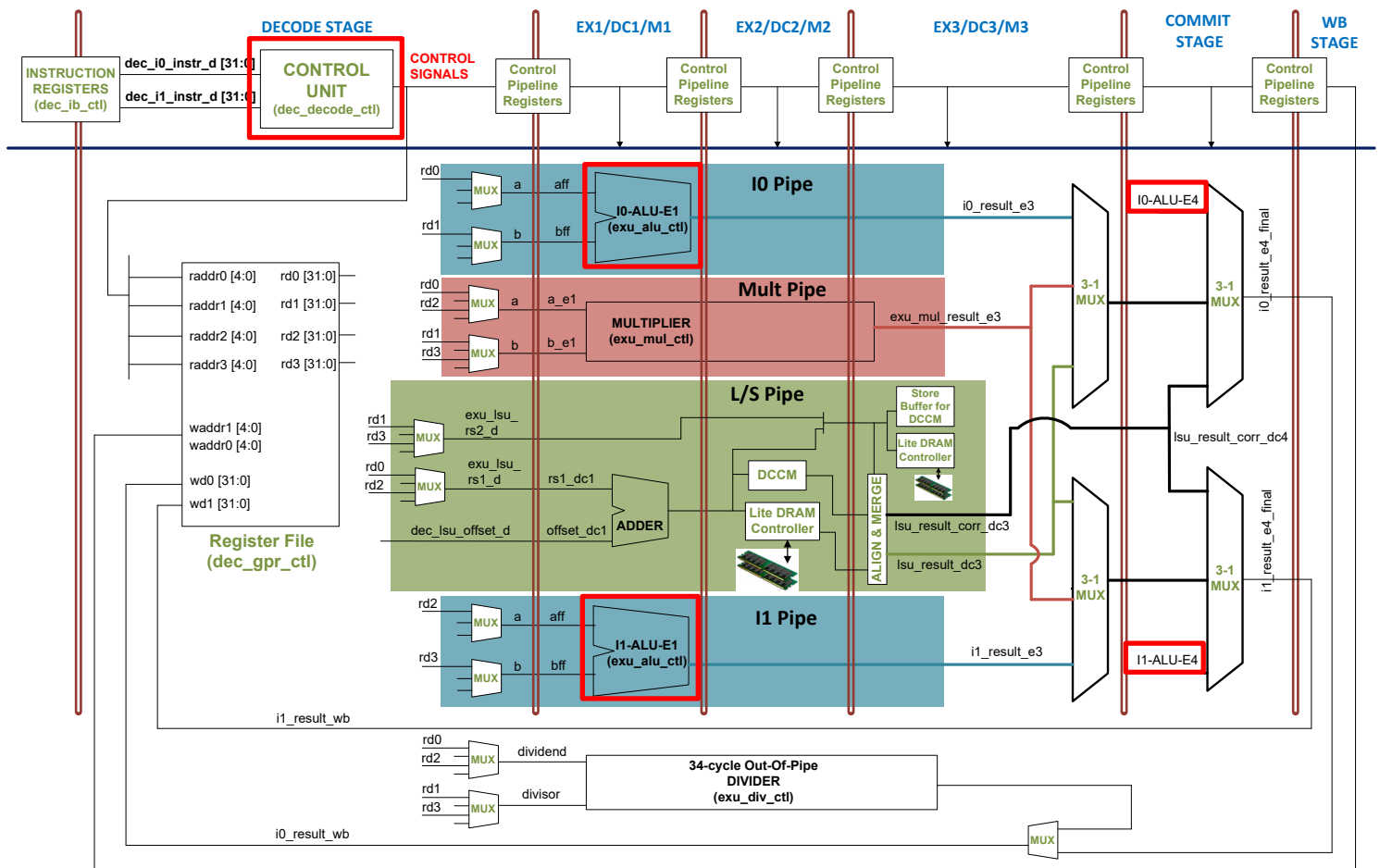


图2. SweRV EH1的译码、执行、提交和回写阶段

在本练习中，我们将逐步指导您添加新的minu指令。在接下来的练习2中，您将遵循相似的步骤添加其他bitmanip指令。

### 修改控制单元:

**注：**在执行下面的步骤之前，建议您先回顾一下实验11的第2.C.i部分和SweRVref.docx的第4部分。

现在，我们将修改/增加支持新指令所需的控制信号。

- 在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` 中创建两个新控制位。这两控制位将用于通知处理器minu指令是否正在执行。

- o 创建名为minu的控制位，作为结构类型dec\_pkt\_t的一部分（图3）。请记住，这是控制单元使用的主要结构类型。

```
typedef struct packed {
    // MINU Instruction
    logic minu;
    logic alu;
    logic rs1;
    logic rs2;
}
```

图3. 结构dec\_pkt\_t中的新位

- 创建名为`minu`的控制位，作为结构类型`alu_pkt_t`的一部分（图4）。请记住，这是算术逻辑指令专用的结构类型。

```
typedef struct packed {
    // MINU Instruction
    logic minu;
    logic valid;
    logic land;
    logic lor;
}
```

图4. 结构alu\_pkt\_t中的新位

- 在模块`dec_decode_ctl`（在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv` 中实现）中，为新控制信号分配值。

- 在译码阶段使用信号`i0_dp_raw`和`i1_dp_raw`为新增的`minu`位分配值。为此，必须修改模块`dec_dec_ctl`中的公式（文件`dec_decode_ctl.sv`的第2497至第2672行），详见下文（请注意，相关说明源自模块`dec_decode_ctl`的第2482至第2495行，我们对其进行了一些扩充）：

#### 1. 文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/decode` 是人类可读的文件，其中包含SweRV EH1处理器中定义的所有指令编码，必须按照下述步骤修改该文件，向其中添加`minu`指令。

- 在`.definition`部分，根据新指令的格式（如图1所示），为新指令创建新的一行（图5）。

```
.definition
minu = [0000101.....101.....0110011]
add = [0000000.....000.....0110011]
addi = [.....000.....0010011]
sub = [0100000.....000.....0110011]
```

图5. 修改`.definition`部分

- 在`.output`部分，创建名为`minu`的新比特位（图6）。

```
.output

rv32i = {
    minu
    alu
    rs1
    rs2
    imm12
}
```

图6. 修改.output部分

- 在.decode部分，为minu指令创建新的一行（图7）。应参照为add指令使能的位，为新指令使能同样的位（add位除外）。也就是说，应使能以下位：alu、rs1、rs2、rd、pm\_alu。此外，还应使能新增的minu位。

```
.decode

rv32i[minu] = { alu rs1 rs2 rd pm_alu minu }

rv32i[mul]   = { mul rs1 rs2 rd low }
rv32i[mulh]  = { mul rs1 rs2 rd rs1_sign rs2_sign }
rv32i[mulhu] = { mul rs1 rs2 rd }
rv32i[mulhsu] = { mul rs1 rs2 rd rs1_sign }
```

图7. 修改.decode部分

2. 在同一文件夹  
([RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/) 中生成通用公式，.decode文件修改后，其中将包含SweRV EH1支持的指令和minu指令。

```
./coredecode -in decode > coredecode.e
```

```
./espresso.linux -Dso -oeqntott coredecode.e |
./addassign -pre out. > equations
```

上述两条命令将生成文件coredecode.e和equations。

3. 在同一文件夹  
([RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/) 中生成合法等式。

```
./coredecode -in decode -legal > legal.e
```

```
./espresso.linux -Dso -oeqntott legal.e |
./addassign -pre out.> legal_equation
```

上述两条命令将生成文件legal.e和legal\_equations。

4. 修改dec\_dec\_ctl模块，将现有等式（文件dec\_decode\_ctl.sv的第2497至第2672行）替换为文件equations和legal\_equations中定义的新等式。

- 在模块 **dec\_decode\_ctl** 中，使用信号 *i0\_dp* 和 *i1\_dp*（图8）为信号 *i0\_ap* 和 *i1\_ap* 中新增的 *minu* 位分配值。

```
// MINU Instruction
assign i0_ap.minu = i0_dp.minu;

// MINU Instruction
assign i1_ap.minu = i1_dp.minu;
```

图8. 为 *minu* 位分配值

上述步骤描述了向 SweRV EH1 处理器中添加新指令时，修改控制单元必须遵循的一般程序。

### 修改执行单元：

接下来修改执行单元，该单元是在模块 **exu**、**exu\_alu\_ctl**、**exu\_mul\_ctl** 和 **exu\_div\_ctl** 中实现的（包含这些模块的文件与模块同名）。在后续练习中，我们将分析需要一整条新管道的复杂情况。本练习则只需对模块 **exu\_alu\_ctl** 进行少量修改（图9）。

```
// MINU Instruction
logic sel_minu;

// MINU Instruction
assign sel_minu = ap.minu;

// MINU Instruction
assign out[31:0] = sel_minu ? ((a_ff < b_ff) ? a_ff : b_ff) :
    ({32{sel_logic}} & tout[31:0]) |
    ({32{sel_shift}} & sout[31:0]) |
    ({32{sel_adder}} & aout[31:0]) |
    ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1], 1'b0}) |
    ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
    ({31'b0, slt_one});
```

图9. 修改ALU

完成上述更改后，即可测试新的指令。在 Verilator 中执行仿真，演示新指令的执行情况。可以使用图10中提供的程序，也可自行创建程序。

图10中的程序会形成一个无限循环，在每次迭代中计算两个寄存器中的最小值。请注意，新指令不能以常规方式（使用助记符）使用，而是必须以机器格式直接使用，因为 RISC-V 编译器尚不支持助记符。

```
.globl main
main:
```

```

li t3, 0x2
li t4, 0x30
li t6, -0x5

REPEAT:
    nop
    nop
    add t3, t3, t3
    add t4, t4, t6
    nop
    .word 0x0bde5f33 # minu t5, t4, t3    0000 101 | 1 1101 | 1110 0 | 101 | 1111 0 | 011 0011
    nop
    nop
    beq zero, zero, REPEAT    # Repeat the loop
    nop
.end

```

图10. 新指令（以红色突出显示）的简单测试程序

图11所示为Verilator中的仿真结果（与往常一样，我们使用`.tcl`脚本来包含信号）。波形为循环的两次迭代，其中展示了新指令的两次执行（`ifu_i0_instr`或`ifu_i1_instr = 0x0BDE5F33`）。新指令的主控制位（`i0_dp_raw`或`i1_dp_raw = 0x7A00000000003`）及ALU控制位（`i0_ap`或`i1_ap = 0x180000`）与`add`指令基本相同，唯一的区别在于前者的`minu`位与后者的`add`位。向`t5`中写入的结果（如图底部所示）是从`t3`和`t4`读取的两个数字中的较小值。请注意，第二次`minu`执行会比较`0xFFFFFFFFE`与`0x00000800`的大小；如果该指令为`unsigned min`指令，则`0xFFFFFFFFE`代表一个很大的正数，因此两者之间的较小值为`0x00000800`。

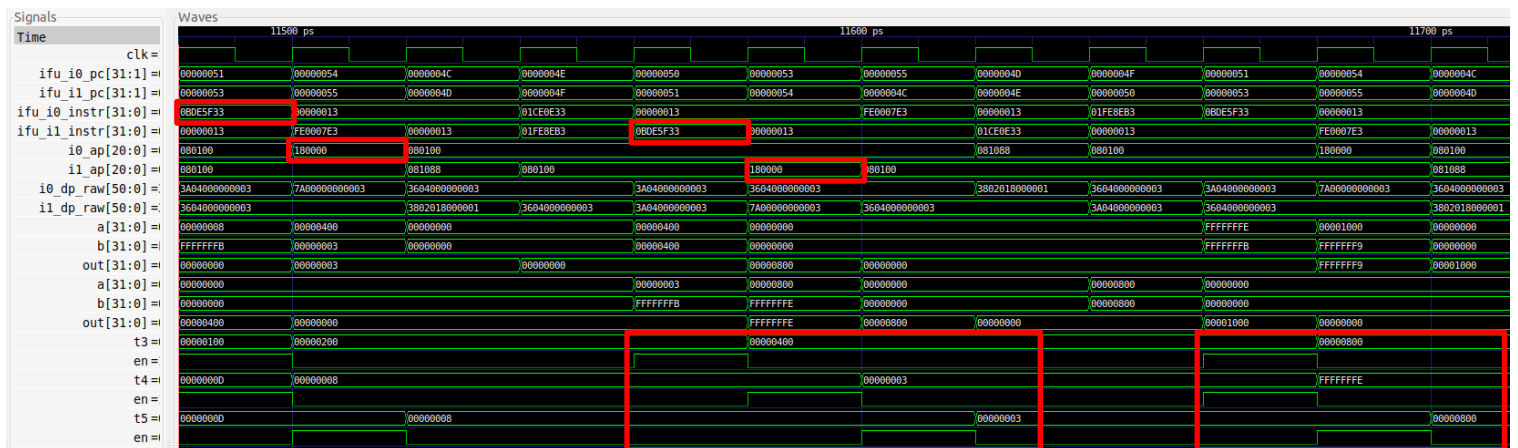


图11. 图10中程序的Verilator仿真

修改程序，使其执行不同的比较，然后使用Verilator仿真程序。

验证程序可正确实施后，在Vivado中生成新的比特流，并使用任意仿真测试在开发板上测试新指令。

构建一个程序，该程序应能够使用新指令读取16个开关，并能够比较8个最低有效开关的二进制值与8个最高有效开关的二进制值。然后在7段显示屏上显示其中的最小值。

最后，构建不同的测试以确认指令能够按预期运行，并在开发板上演示结果。

2) 实现RISC-V *bitmanip*扩展中包含的其他指令。首先是剩余的min/max指令：min、max和maxu。

3) 在本练习中，您需要扩展SweRV EH1处理器，向其中添加三条属于RISC-V单精度浮点扩展（F扩展）的新指令：fadd.s、fmul.s和fdiv.s。

- 这些指令假定操作数以IEEE 754标准所定义的单精度浮点格式表示（<https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>）。对于浮点数，寄存器在逻辑上分为三个字段：**Sign**（1位）、**Exponent**（8位）和**Mantissa**（23位）。

**Sign** | **E<sub>7</sub> ... E<sub>0</sub>** | **M<sub>22</sub> ... M<sub>0</sub>**

- 指令fadd.s rd, rs1, rs2会将rs1和rs2中的两个浮点值相加，并将结果存储在rd中。指令fmul.s rd, rs1, rs2会将rs1和rs2中的两个浮点值相乘，并将结果存储在rd中。最后，指令fdiv.s rd, rs1, rs2会将rs1和rs2中的两个浮点值相除，并将结果存储在rd中。

- 根据RISC-V F扩展中的定义，这些指令使用的格式如下：

```
fadd.s: 0000000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fmul.s: 0001000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fdiv.s: 0001100 | rs2 | rs1 | Rounding-Mode | rd | 1010011
```

- 该扩展假设处理器具有32个浮点寄存器，但为了简单起见，在本练习中，您将使用其他指令所使用的现有寄存器文件（即x寄存器）。此外，我们还进行了其他简化的假设：一次只能执行一条浮点指令，且浮点指令会阻塞。

要在SweRV EH1处理器中添加对这些指令的支持，必须进行以下修改：

### 修改执行单元：

应添加用于支持浮点加法、乘法和除法的硬件（部分资源可通过互联网获取，如下所述）。后续执行fadd、fmul或fdiv指令时会用到该硬件。为此，请完成以下步骤：

- 访问以下链接，下载多周期浮点加法器、减法器 and 除法器：  
<https://github.com/dawsonjon/fpu>。这些单元均为非流水线多周期单元，类似于SweRV EH1中提供的整数除法器。



- 新单元构成了新的管道，因此可以单独处理，但由于该执行管道提供了一些有助于支持新指令的信号，例如信号 *finish* 和 *div\_stall*，您也可以将这三个浮点单元在 **exu\_div\_ctl** 模块中进行实例化。如果选择后一种方式，则在为控制单元生成等式时，应参照为 *div* 指令使能的位，为新指令使能同样的位，同时使能新的浮点位。

### 修改控制单元:

修改/新建支持新指令所需的控制信号。

- 在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` 中创建新的位和结构类型。
  - 创建一个名为 *fp\_pkt\_t* 的新结构类型，其中包含 *fp\_add*、*fp\_mul* 和 *fp\_div* 三个位，分别指示处理器是否正在执行浮点加法、浮点乘法或浮点除法。
  - 创建名为 *fp\_add*、*fp\_mul* 和 *fp\_div* 的三个新位，作为结构类型 *dec\_pkt\_t* 的一部分。请记住，这是控制单元使用的主要结构类型。
- 在模块 **dec\_decode\_ctl**（在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv` 中实现）中，为新控制信号分配值。
  - 为信号 *i0\_dp\_raw* 和 *i1\_dp\_raw* 中的新位分配值。为此，必须根据练习1中的说明，在模块 **dec\_dec\_ctl** 中重新生成等式。如上所述，如果将新指令作为 *div* 指令处理，则在模块 **dec\_dec\_ctl** 中生成等式时，必须参照为 *div* 指令使能的位，为新指令使能同样的位，同时使能新的浮点位。
  - 新建一个名为 *fp\_p* 的 *fp\_pkt\_t* 型信号。然后使用信号 *i0\_dp* 和 *i1\_dp* 为该结构的三个位分配值。请注意，与 *mul* 或 *div* 指令类似，新指令只需要一个此类信号，因为在给定的周期内只能执行一条浮点指令。

修改硬件后，在 **Verilator** 中执行仿真，演示新指令的执行情况。可以使用图12中提供的程序，也可自行创建程序。图12中的程序会形成一个无限循环，以执行浮点加法、乘法和除法三条指令的运算。

```
.globl main
main:

li t0, 0x4
li t1, 0x2
li t3, 0x40800000
li t4, 0x40000000
```

```

REPEAT:
    div t5, t0, t1
    nop
    nop
    .word 0x01ce8f53      # fadd.s 00000000 | 11100 | 11101 | 000 | 11110 | 1010011
    nop
    nop
    .word 0x11ce8f53      # fmul.s 00010000 | 11100 | 11101 | 000 | 11110 | 1010011
    nop
    nop
    .word 0x19ce8f53      # fdiv.s 00011000 | 11100 | 11101 | 000 | 11110 | 1010011
    nop
    nop
    beq zero, zero, REPEAT    # Repeat the loop
    nop
.end

```

**图12. 新指令（以红色突出显示）的简单测试程序**

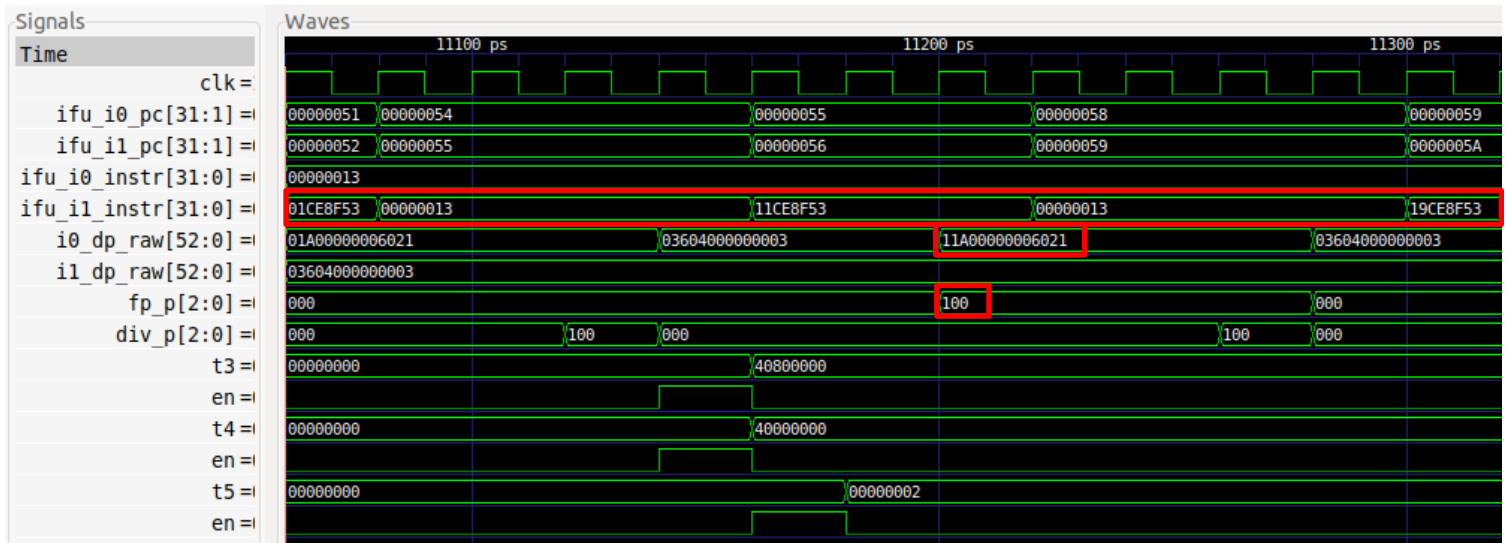
图13所示为Verilator中的仿真结果。可以使用浮点转换器检查结果，如以下链接中提供的浮点转换器：<https://www.h-schmidt.net/FloatConverter/IEEE754.html>。

在图13-a中，三条浮点指令被取指到ifu\_i0\_instr或ifu\_i1\_instr中。其主控制位（dec\_pkt\_t）是在div指令（i0\_dp\_raw = 0x11A000000006021）的主控制位的基础上额外增加了三位，如上文所述。fadd、fmul和fdiv的FP（浮点）控制位（fp\_pkt\_t）为100（如图所示）、010和001。

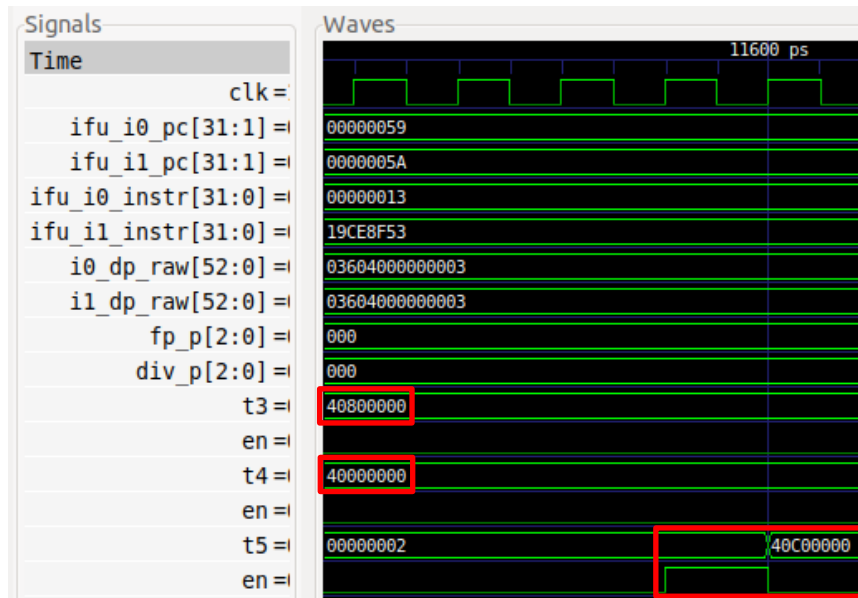
图13-b显示了浮点加法指令在几个周期后将运算结果写入t5。请注意，输入值为0x40800000和0x40000000，因此加法运算的结果为0x40c00000。

图13-c显示了浮点乘法指令在几个周期后将运算结果写入t5。请注意，输入值为0x40800000和0x40000000，因此乘法运算的结果为0x41000000。

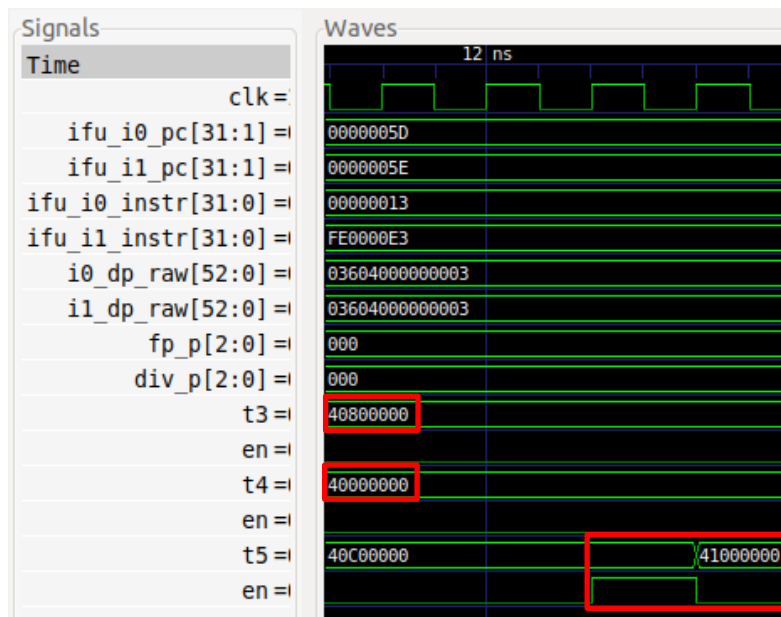
最后，图13-d显示了浮点除法指令在几个周期后将运算结果写入t5。请注意，输入值为0x40800000和0x40000000，因此除法运算的结果为0x40000000。



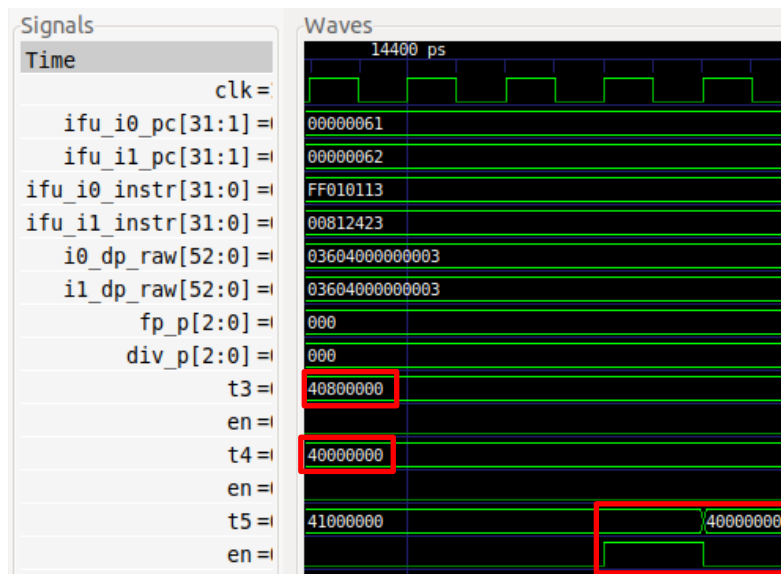
(a)



(b)



(c)



(d)

图13. 图12中程序的Verilator仿真

修改程序以测试其他情况，演示指令能够正确执行。例如，测试数值为负数的情况以及先/后续指令的数据相关性等。然后使用Verilator进行仿真。

接下来，在开发板上的硬件中测试新指令。为此，需使用新的fmul和fadd指令对GSG中提供的示例*DotProduct\_C-Lang*进行编程，以执行浮点计算。比较模拟浮点指令时以及在硬件中具体实现这些指令时该算法的执行情况。

还可以添加更多功能，例如提供以下支持：其他浮点格式（如双精度）、其他浮点舍入模式、用于存储浮点值的新寄存器文件、实现自己的FP单元等。

- 4) 实现二分法计算。您可以在网上找到有关该寻根算法的许多信息，例如，可参阅以下链接：  
[https://en.wikipedia.org/wiki/Bisection\\_method](https://en.wikipedia.org/wiki/Bisection_method)。

比较模拟浮点指令时以及在硬件中具体实现这些指令时该算法的执行情况。

- 5) 实现《计算机组织结构和设计》（RISC-V版本，Patterson & Hennessy ([HePa])）第4章的练习中提到的指令，例如：

a. ([HePa]练习4.11)：

- i. “增量装载”指令： `lwi.d rd, rs1, rs2`
- ii. 说明： `rd = Mem[rs1 + rs2]`

b. ([HePa]练习4.12)：

- i. “交换”指令： `swap rs1, rs2`
- ii. 说明： `rs2 = rs1; rs1 = rs2`

c. ([HePa]练习4.13)：

- i. “存储和”指令： `ss rs1, rs2, imm`
- ii. 说明： `Mem[rs1] = rs2 + imm`

- 6) 仿照上一练习，实现S.Harris和D.Harris所著教材《数字设计和计算机体系结构》（RISC-V版本，简称[DDCARV]）第7章的练习3至练习6中提到的指令。我们会在下文中再次列出这四个练习中的所有指令。其中一些指令已得到SweRV EH1处理器的支持，对于这些指令，您只需简单说明实现方法，无需具体实现。

a. 练习3: `xor, sll, srl, bne`。（已在SweRV EH1中实现）

b. 练习4: `lui, sra, lbu, blt, bltu, bge, bgeu, jalr, auipc, sb, slli, srai`。（已在SweRV EH1中实现）

c. 练习5: `lwpostinc rd, imm(rs)`（相当于下面两条指令：在 `lw rd, 0(rs)` 后添加 `addi rs, rs, imm`）。

d. 练习6: `lwpreinc rd, imm(rs)`（相当于下面两条指令：在 `lw rd, imm(rs)` 后添加 `addi rs, rs, imm`）。

- 7) 包含一个新的事件，用于计算程序中执行的I型指令数量。我们会提供一些指导，帮助您完成此练习：

○ 您需要修改文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_`

*types.sv*中的部分结构。具体而言，应在以下结构类型中另外添加一个字段：

- 结构 `inst_t`：用于I型指令的新字段。

- 如您所知，需在模块 `dec_decode_ctl`（文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`）中分配控制位。修改信号 `i0_itype` 和 `i1_itype` 的分配，添加先前所包含的新指令类型。
- 需在模块 `dec_tlu_ctl`（文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_tlu_ctl.sv`）中实现硬件计数器。打开该文件，分析第1882至第2143行中包含的代码。必须修改这部分代码，才能包含新计数器。

在Verilog代码中包含新计数器后，使用Verilator进行仿真调试。通过仿真验证实现结果后，为SoC生成新的比特流，并在开发板上测试硬件中新计数器的工作情况。