



IMAGINATION大學計劃

RVfpga實驗18

新增新功能：指令和計數器

1. 簡介

本實驗將應用在先前實驗中獲得的知識來修改SweRV EH1處理器，向其新增以下新功能：

- **新增A-L指令：**透過RISC-V架構中提供的全新位元操作延伸功能來新增算術邏輯指令。
- **新增浮點指令：**新增三條浮點指令：加、乘、除。然後使用這些指令進行二分法計算。
- **新增計數器：**新增一個新的硬體計數器，用於計算執行的I型指令數量。

在一些練習中，我們將指導您完成修改核心的過程，在其他練習中，您必須自行確定所需的操作。

2. 練習

- 1) 位元操作 (*bitmanip*) 延伸功能由基本RISC-V架構的數個元件延伸功能組成，旨在縮減程式碼長度、改進效能並降低能耗。可前往<https://github.com/riscv/riscv-bitmanip>取得完整的規格。檔案<https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>詳細介紹了該延伸功能包含的所有指令。

在本練習中，您將在SweRV EH1處理器中新增一個來自`bitmanip`延伸功能的新指令。具體而言，您將新增`minu`指令，該指令會將rs1和rs2中較小的一個無符號整數放入rd。該指令使用的格式如下圖所示。

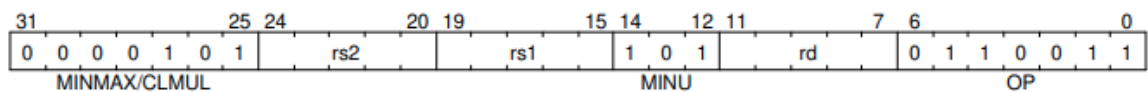


圖1. minu指令使用的格式

(圖片來源：<https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>)

要加入新的算術邏輯指令，必須修改處理器的兩個主要部分：**控制單元**和**執行單元**。圖2中以紅色強調顯示了新增minu指令前必須在上述兩個單元中修改的結構（請記住，此圖最初用作實驗11中的圖4）。

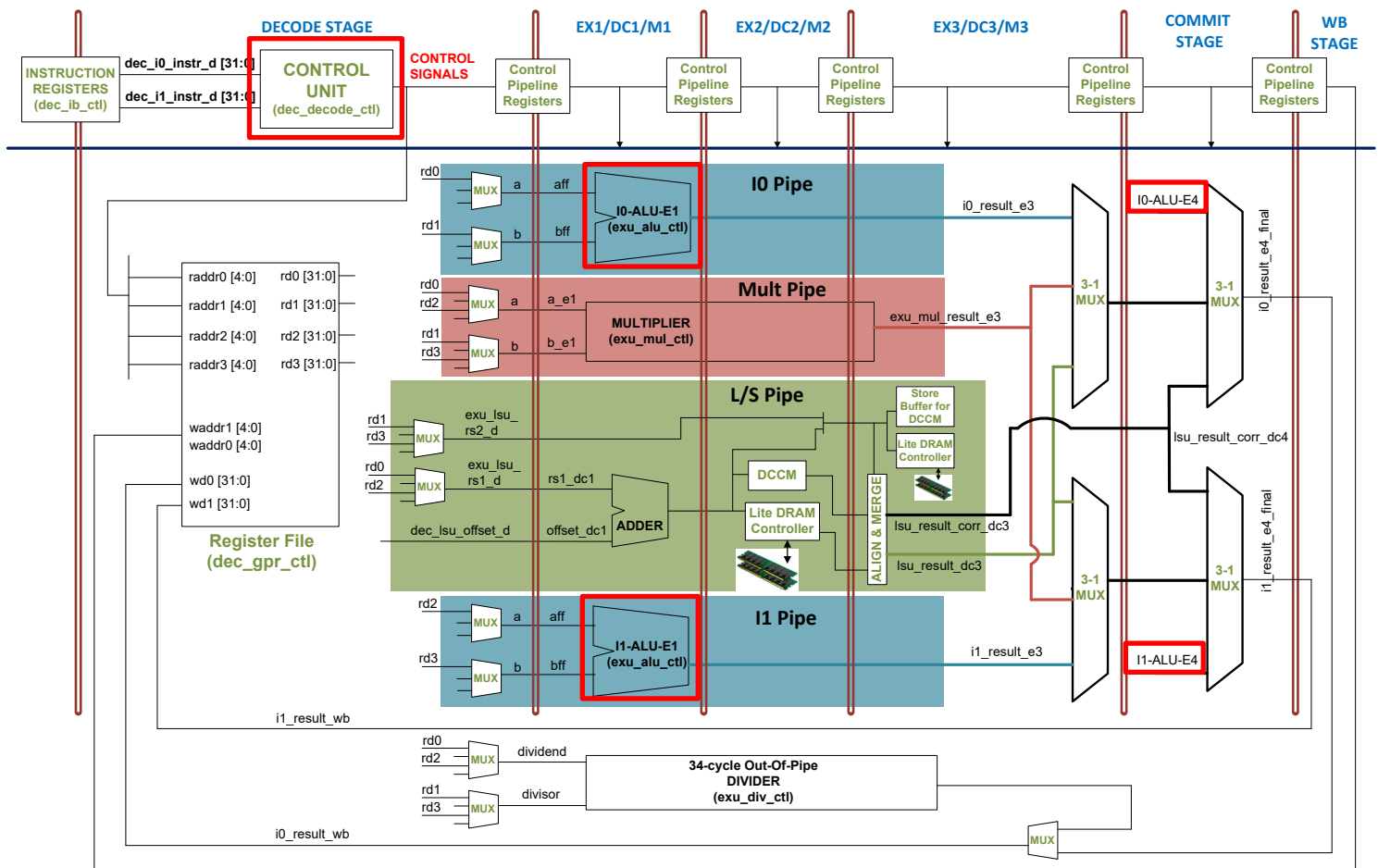


圖2. SweRV EH1的解碼、執行、提交和寫回階段

在本練習中，我們將逐步指導您新增新的minu指令。在接下來的練習2中，您將遵循相似的步驟新增其他bitmanip指令。

修改控制單元：

附註：在執行下面的步驟之前，建議您先回顧一下實驗11的第2.C.i部分和SweRVref.docx的第4部分。

現在，我們將修改/增加支援新指令所需的新控制訊號。

- 在檔案
[RVfpgaPath]/RVfpga/src/SweRVofSoC/SweRVEh1CoreComplex/include/swerv_types.sv
中建立兩個新控制位元。這兩個控制位元將用於通知處理器minu指令是否正在執行。
- 建立名為minu的控制位元，作為結構類型dec_pkt_t的一部分（圖3）。請記住，這是控制單元使用的主要結構類型。

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

圖3. 結構dec_pkt_t中的新位元

- 建立名為`minu`的控制位元，作為結構類型`alu_pkt_t`的一部分（圖4）。請記住，這是算術邏輯指令專用的結構類型。

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

圖4. 結構alu_pkt_t中的新位元

- 在模組`dec_decode_ctl`（在檔案`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`中實作）中，為新控制訊號指定值。

- 在解碼階段使用訊號`i0_dp_raw`和`i1_dp_raw`為新增的`minu`位元指定值。為此，必須修改模組`dec_dec_ctl`中的公式（檔案`dec_decode_ctl.sv`的第2497至第2672行），詳見下文（請注意，相關說明源自模組`dec_decode_ctl`的第2482至第2495行，我們對其進行了一些擴充）：

1. 檔案

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode`是人類可讀的檔案，其中包含SweRV EH1處理器中定義的所有指令編碼，必須按照下述步驟修改該檔案，向其中新增`minu`指令。

- 在`.definition`部分，根據新指令的格式（如圖1所示），為新指令建立新的一行（圖5）。

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

圖5. 修改`.definition`部分

- 在`.output`部分，建立名為`minu`的新位元（圖6）。

```
.output

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

圖6. 修改.output部分

- 在.decode部分，為minu指令建立新的一行（圖7）。應參照為add指令啟用的位元，為新指令啟用同樣的位元（add位元除外）。也就是說，應啟用以下位元：alu、rs1、rs2、rd、pm_alu。此外，還應啟用新增的minu位元。

```
.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 }
```

圖7. 修改.decode部分

2. 在同一資料夾
（[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/）
中產生一般公式，.decode檔案修改後，其中將包含SweRV EH1支援的指令和minu指令。

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

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

上述兩條命令將產生檔案coredecode.e和equations。

3. 在同一資料夾
（[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/）
中產生合法等式。

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

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

上述兩條命令將產生檔案legal.e和legal_equations。

4. 修改dec_dec_ctl模組，將現有等式（檔案dec_decode_ctl.sv的第2497至第2672行）替換為檔案equations和legal_equations中定義的新等式。

- 在模組 **dec_decode_ctl** 中，使用訊號 *i0_dp* 和 *i1_dp*（圖8）為訊號 *i0_ap* 和 *i1_ap* 中新增的 *minu* 位元指定值。

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

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

圖8. 為 *minu* 位元指定值

上述步驟描述了向 SweRV EH1 處理器中新增新指令時，修改控制單元必須遵循的一般程序。

修改執行單元：

接下來修改執行單元，該單元是在模組 **exu**、**exu_alu_ctl**、**exu_mul_ctl** 和 **exu_div_ctl** 中實作的（包含這些模組的檔案與模組同名）。在後續練習中，我們將分析需要一整條新管道的複雜情況。本練習則只需對模組 **exu_alu_ctl** 進行少量修改（圖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});
```

圖9. 修改ALU

完成上述更改後，即可測試新的指令。在 Verilator 中執行模擬，展示新指令的執行情況。可以使用圖10中提供的程式，也可自行建立程式。

圖10中的程式會形成一個無限迴圈，在每次迭代中計算兩個暫存器中的最小值。請注意，新指令不能以正常方式（使用助憶鍵）使用，而是必須以機器格式直接使用，因為 RISC-V 編譯器尚不支援助憶鍵。

```
.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

```

圖10. 新指令（以紅色強調顯示）的簡單測試程式

圖11所示為Verilator中的模擬結果（與往常一樣，我們使用`.tc`指令碼來包含訊號）。波形為迴圈的兩次迭代，其中顯示了新指令的兩次執行（`ifu_i0_instr`或`ifu_i1_instr = 0x0BDE5F33`）。新指令的主控制位元（`i0_dp_raw`或`i1_dp_raw = 0x7A000000000003`）及ALU控制位元（`i0_ap`或`i1_ap = 0x180000`）與`add`指令基本相同，唯一的區別在於前者的`minu`位元與後者的`add`位元。向`t5`中寫入的結果（如圖底部所示）是從`t3`和`t4`讀取的兩個數字中的較小值。請注意，第二次`minu`執行會比較`0xFFFFFFFEE`與`0x00000800`的大小；如果該指令為`unsigned min`指令，則`0xFFFFFFFEE`代表一個很大的正數，因此兩者之間的較小值為`0x00000800`。

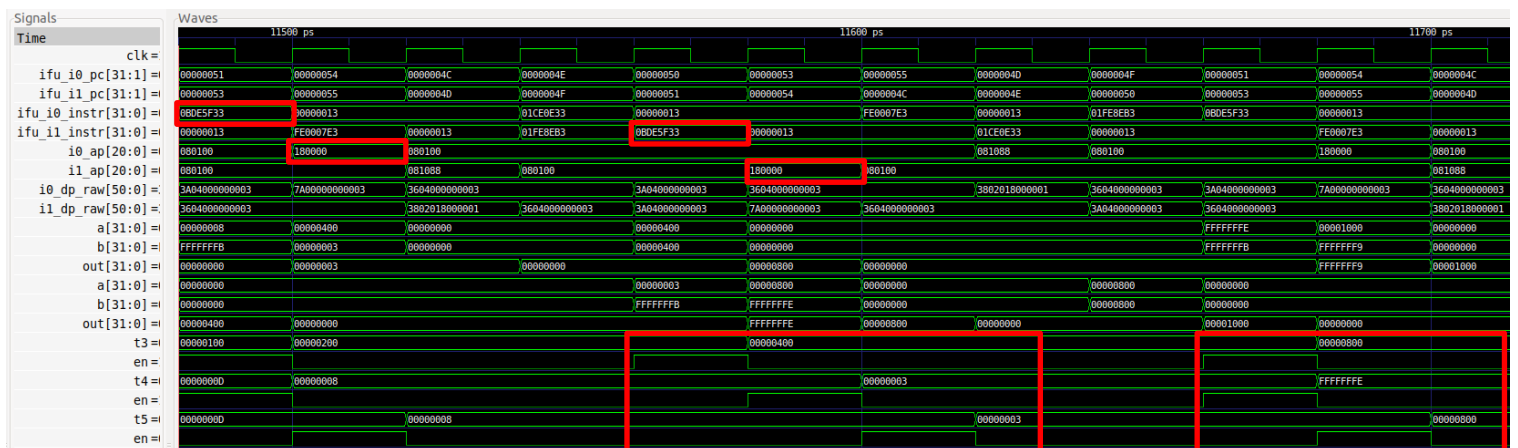


圖11. 圖10中程式的Verilator模擬

修改程式，使其執行不同的比較，然後使用Verilator模擬程式。

驗證程式可正確運作後，在Vivado中產生新的位元流，並使用任意模擬測試在開發板上測試新指令。

建立一個程式，該程式應能夠使用新指令讀取16個開關，並能夠比較8個最低有效開關的二進位值與8個最高有效開關的二進位值。然後在7段顯示器上顯示其中的最小值。

最後，建立不同的測試以確認指令能夠按預期執行，並在開發板上展示結果。

2) 實作RISC-V *bitmanip* 延伸功能中包含的其他指令。首先是剩餘的min/max指令：min、max和Imaxu。

3) 在本練習中，您需要延伸SweRV EH1處理器，向其中新增三條屬於RISC-V單精度浮點延伸功能（F延伸功能）的新指令：fadd.s、fmul.s和fdiv.s。

- 這些指令假定運算元以IEEE 754標準所定義的單精度浮點格式表示（<https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>）。對於浮點數，暫存器在邏輯上分為三個欄位：**Sign**（1位元）、**Exponent**（8位元）和**Mantissa**（23位元）。

Sign | **E₇ ... E₀** | **M₂₂ ... M₀**

- 指令fadd.s rd, rs1, rs2會將rs1和rs2中的兩個浮點值相加，並將結果儲存在rd中。指令fmul.s rd, rs1, rs2會將rs1和rs2中的兩個浮點值相乘，並將結果儲存在rd中。最後，指令fdiv.s rd, rs1, rs2會將rs1和rs2中的兩個浮點值相除，並將結果儲存在rd中。

- 根據RISC-V F延伸功能中的定義，這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
```

- 該延伸功能假設處理器具有32個浮點暫存器，但為了簡單起見，在本練習中，您將使用其他指令所使用的現有暫存器檔案（即x暫存器）。此外，我們還進行了其他簡化的假設：一次只能執行一條浮點指令，且浮點指令會阻塞。

要在SweRV EH1處理器中新增對這些指令的支援，必須進行以下修改：

修改執行單元：

應新增用於支援浮點加法、乘法和除法的硬體（部分資源可透過網際網路取得，如下所述）。後續執行fadd、fmul或fdiv指令時會用到該硬體。為此，請完成以下步驟：

- 前往以下連結，下載多週期浮點加法器、減法器 and 除法器：
<https://github.com/dawsonjon/fpu>。這些單元均為非管線多週期單元，類似於SweRV EH1中提供的整數除法器。

- 新單元構成了新的管道，因此可以單獨處理，但由於該執行管道提供了一些有助於支援新指令的訊號，例如訊號 *finish* 和 *div_stall*，您也可以將這三個浮點單元在 **exu_div_ctl** 模組中進行實例化。如果選擇後一種方式，則在為控制單元產生等式時，應參照為 *div* 指令啟用的位元，為新指令啟用同樣的位元，同時啟用新的浮點位元。

修改控制單元：

修改/建立支援新指令所需的控制訊號。

- 在檔案 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` 中建立新的位元和結構類型。
 - 建立一個名為 *fp_pkt_t* 的新結構類型，其中包含 *fp_add*、*fp_mul* 和 *fp_div* 三個位元，分別指示處理器是否正在執行浮點加法、浮點乘法或浮點除法。
 - 建立名為 *fp_add*、*fp_mul* 和 *fp_div* 的三個新位元，作為結構類型 *dec_pkt_t* 的一部分。請記住，這是控制單元使用的主要結構類型。
- 在模組 **dec_decode_ctl**（在檔案 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv` 中實作）中，為新控制訊號指定值。
 - 為訊號 *i0_dp_raw* 和 *i1_dp_raw* 中的新位元指定值。為此，必須根據練習 1 中的說明，在模組 **dec_dec_ctl** 中重新產生等式。如上所述，如果將新指令作為 *div* 指令處理，則在模組 **dec_dec_ctl** 中產生等式時，必須參照為 *div* 指令啟用的位元，為新指令啟用同樣的位元，同時啟用新的浮點位元。
 - 建立一個名為 *fp_p* 的 *fp_pkt_t* 型新訊號。然後使用訊號 *i0_dp* 和 *i1_dp* 為該結構的三個位元指定值。請注意，與 *mul* 或 *div* 指令類似，新指令只需要一個此類訊號，因為在指定的週期內只能執行一條浮點指令。

修改硬體後，在 **Verilator** 中執行模擬，描繪新指令的執行情況。可以使用圖 12 中提供的程式，也可自行建立程式。圖 12 中的程式會形成一個無限迴圈，以執行浮點加法、乘法和除法三條指令的運算。

```
.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 0001000 | 11100 | 11101 | 000 | 11110 | 1010011
    nop
    nop
    .word 0x19ce8f53      # fdiv.s 0001100 | 11100 | 11101 | 000 | 11110 | 1010011
    nop
    nop
    beq zero, zero, REPEAT    # Repeat the loop
    nop
.end

```

圖12. 新指令（以紅色強調顯示）的簡單測試程式

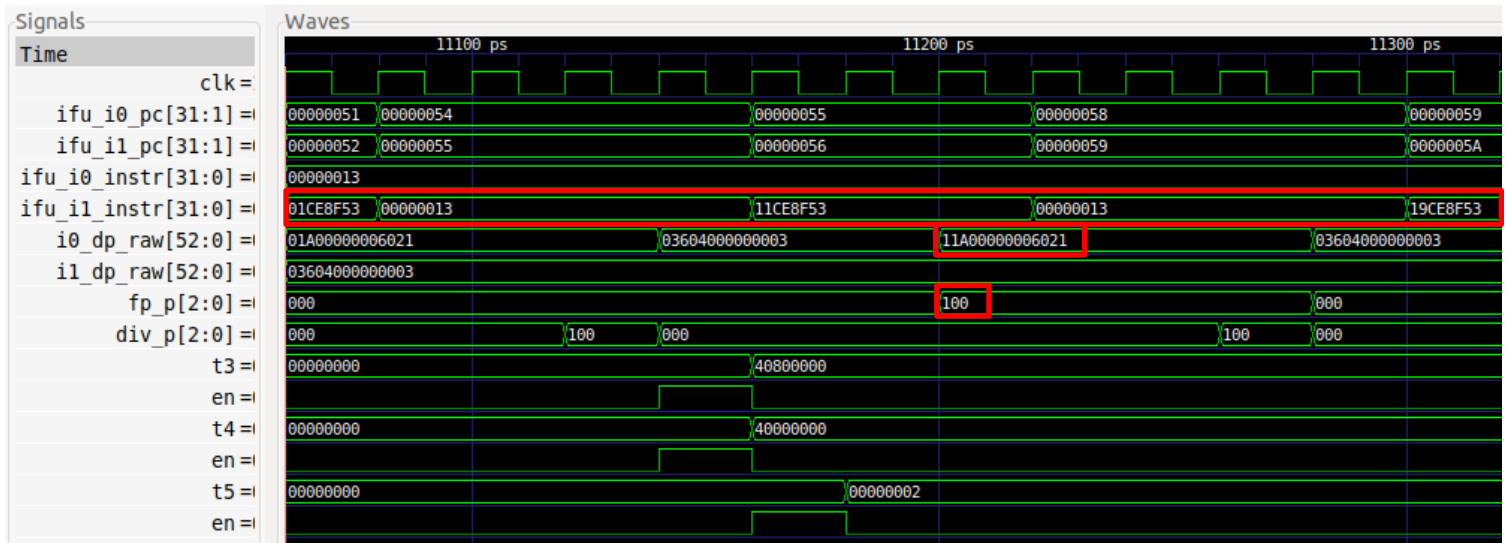
圖13所示為Verilator中的模擬結果。可以使用浮點轉換器檢查結果，如以下連結中提供的浮點轉換器：<https://www.h-schmidt.net/FloatConverter/IEEE754.html>。

在圖13-a中，三條浮點指令被擷取到ifu_i0_instr或ifu_i1_instr中。其主控制位元（dec_pkt_t）是在div指令（i0_dp_raw = 0x11A00000006021）的主控制位元的基礎上額外增加了三位元，如上文所述。fadd、fmul和fdiv的FP（浮點）控制位元（fp_pkt_t）為100（如圖所示）、010和001。

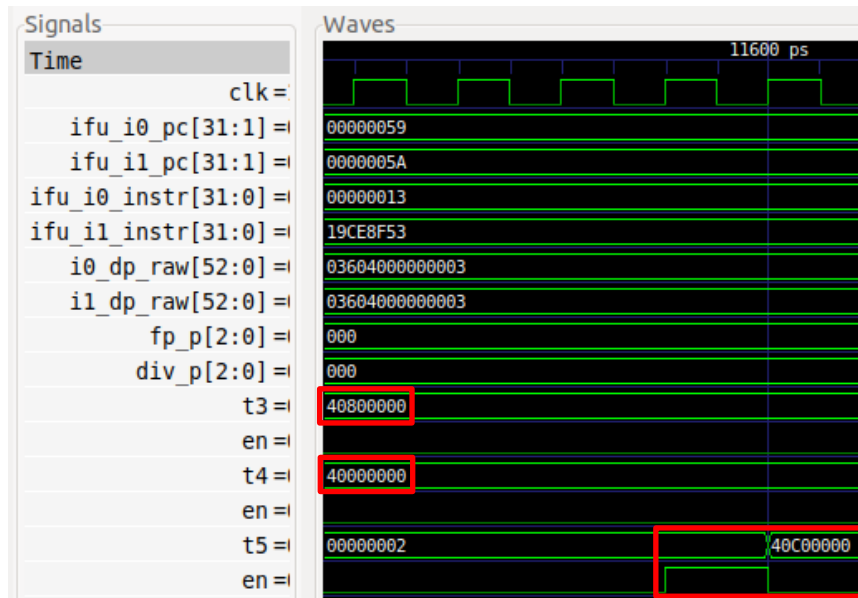
圖13-b顯示了浮點加法指令在幾個週期後將運算結果寫入t5。請注意，輸入值為0x40800000和0x40000000，因此加法運算的結果為0x40c00000。

圖13-c顯示了浮點乘法指令在幾個週期後將運算結果寫入t5。請注意，輸入值為0x40800000和0x40000000，因此乘法運算的結果為0x41000000。

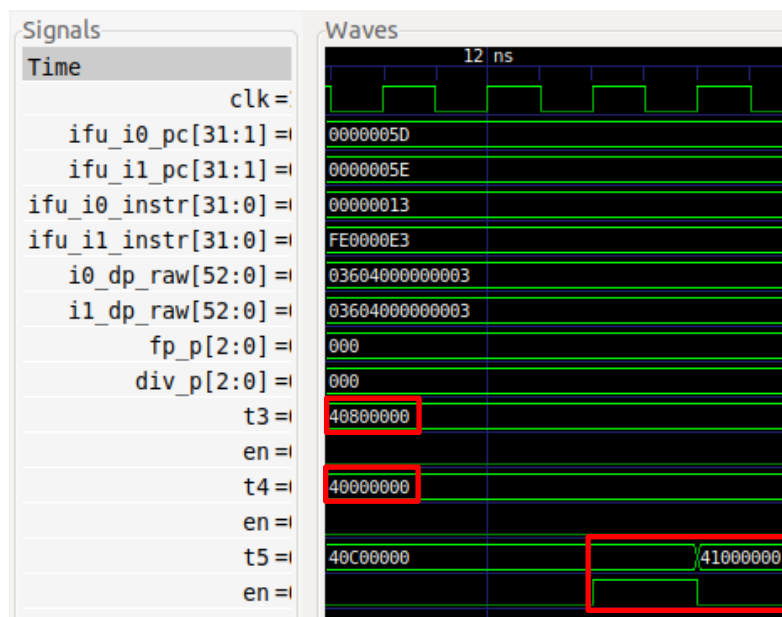
最後，圖13-d顯示了浮點除法指令在幾個週期後將運算結果寫入t5。請注意，輸入值為0x40800000和0x40000000，因此除法運算的結果為0x40000000。



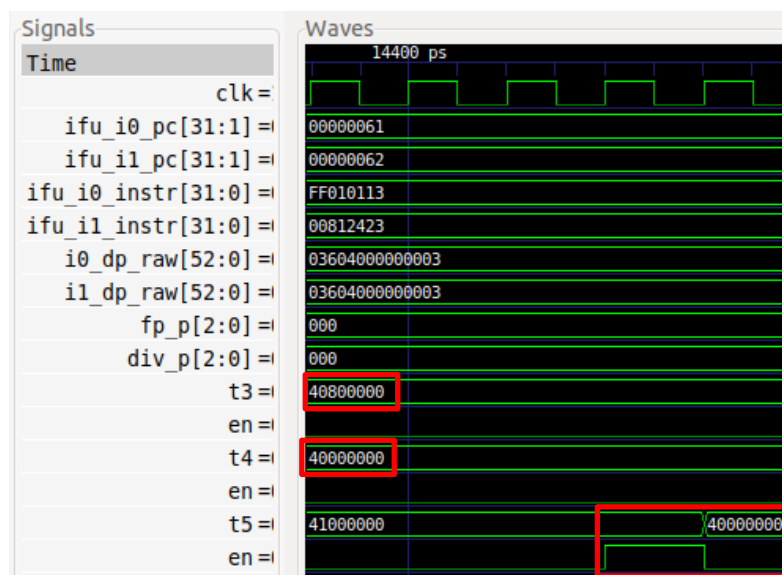
(a)



(b)



(c)



(d)

圖13. 圖12中程式的Verilator模擬

修改程式以測試其他情況，展示指令能夠正確執行。例如，測試數值為負數的情況以及與先前/後續指令的資料相關性等。然後使用Verilator進行模擬。

接下來，在開發板上的硬體中測試新指令。為此，需使用新的fmul和fadd指令對GSG中提供的範例*DotProduct_C-Lang*進行程式編寫，以執行浮點計算。比較模擬浮點指令時以及在硬體中具體實作這些指令時該演算法的執行情況。

還可以新增更多功能，例如提供以下支援：其他浮點格式（如雙精度）、其他浮點捨入模式、用於儲存浮點值的新暫存器檔案、實作自己的FP單元等。

- 4) 實作二分法計算。您可以在網際網路上找到有關該尋根演算法的許多資訊，例如，可參閱以下連結：https://en.wikipedia.org/wiki/Bisection_method。

比較模擬浮點指令時以及在硬體中具體實作這些指令時該演算法的執行情況。

- 5) 實作《電腦組織結構和設計》（RISC-V版本，Patterson & Hennessy ([HePa])）第4章的練習中提到的指令，例如：

a. ([HePa]練習4.11)：

- i. 「增量載入」指令：`lwi.d rd, rs1, rs2`
- ii. 說明：`rd = Mem[rs1 + rs2]`

b. ([HePa]練習4.12)：

- i. 「交換」指令：`swap rs1, rs2`
- ii. 說明：`rs2 = rs1; rs1 = rs2`

c. ([HePa]練習4.13)：

- i. 「儲存和」指令：`ss rs1, rs2, imm`
- ii. 說明：`Mem[rs1] = rs2 + imm`

- 6) 仿照上一練習，實作S.Harris和D.Harris所著教材《數位設計和電腦體系結構》（RISC-V版本，簡稱[DDCARV]）第7章的練習3至練習6中提到的指令。我們會在下文再次列出這四個練習中的所有指令。其中一些指令已得到SweRV EH1處理器的支援，對於這些指令，您只需簡單說明實作方法，無需具體實作。

a. 練習3：`xor, sll, srl, bne`。（已在SweRV EH1中實作）

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

c. 練習5：`lwpostinc rd, imm(rs)`（相當於下面兩條指令：在`lw rd, 0(rs)`後新增`addi rs, rs, imm`）。

d. 練習6：`lwpreinc rd, imm(rs)`（相當於下面兩條指令：在`lw rd, imm(rs)`後新增`addi rs, rs, imm`）。

- 7) 包含一個新的事件，用於計算程式中執行的I型指令數量。我們會提供一些指導，幫助您完成此練習：

○ 您需要修改檔案

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`中的部分結構。具體而言，應在以下結構類型中另外新增一個欄位：

- 結構 `inst_t`：用於I型指令的新欄位。

- 如您所知，需在模組 `dec_decode_ctl`（檔案 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`）中指定控制位元。修改訊號 `i0_itype` 和 `i1_itype` 的指定，新增先前所包含的新指令類型。
- 需在模組 `dec_tlu_ctl`（檔案 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_tlu_ctl.sv`）中實作硬體計數器。開啟該檔案，分析第1882至第2143行中包含的程式碼。必須修改這部分程式碼，才能包含新計數器。

在Verilog程式碼中包含新計數器後，使用Verilator進行模擬除錯。透過模擬驗證實作結果後，為SoC產生新的位元流，並在開發板上測試硬體中新計數器的工作情況。