



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 18

Adição de Novas Funcionalidades: Instruções e Contadores

1. INTRODUÇÃO

Neste laboratório, irá aplicar os conhecimentos adquiridos nos laboratórios anteriores para modificar o processador SweRV EH1 e adicionar as novas funcionalidades:

- **Adicionar instruções A-L:** Adicionar instruções Aritmética-Lógica de manipulação de bits disponível na nova extensão da arquitetura RISC-V.
- **Adicionar instruções de vírgula-flutuante:** Adicionar três instruções de vírgula-flutuante: adição, multiplicação e divisão. Em seguida, utilizá-las para calcular o algoritmo da bissecção.
- **Adicionar um contador:** Adicionar um novo contador em hardware que conta o número de instruções executadas do tipo I.

Nalguns destes exercícios, guiamo-lo através do processo de modificação do núcleo e, noutros, o utilizador descobrirá por si próprio o que precisa de ser feito.

2. EXERCÍCIOS

- 1) A extensão bit-manipulation (*bitmanip*) é composta por várias extensões de componentes da arquitetura RISC-V de base que se destinam a proporcionar uma combinação de redução do tamanho do código, melhoria do desempenho e redução de energia consumida. A especificação completa pode ser encontrada em <https://github.com/riscv/riscv-bitmanip>. O ficheiro <https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf> descreve em pormenor todas as instruções que pertencem a esta extensão.

Neste exercício, irá incluir uma nova instrução da extensão *bitmanip* no processador SweRV EH1. Especificamente, irá adicionar a instrução `minu`, que coloca o menor dos dois inteiros sem sinal de `rs1` e `rs2` em `rd`. O formato utilizado para esta instrução é apresentado na ilustração seguinte.

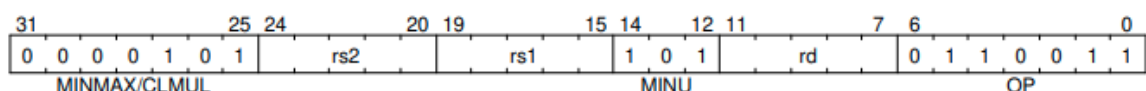


Figura 1. Formato utilizado para a instrução minu (figura obtida de <https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>).

Para incluir uma nova instrução Aritmética-Lógica, é necessário modificar duas partes principais do processador: a **Unidade de Controlo** e a **Unidade de Execução**. Figura 2 destaca a vermelho as estruturas específicas destas duas unidades que tem de modificar para incluir a instrução `minu` (lembre-se que esta figura foi incluída pela primeira vez no Lab 11 como Figura 4).

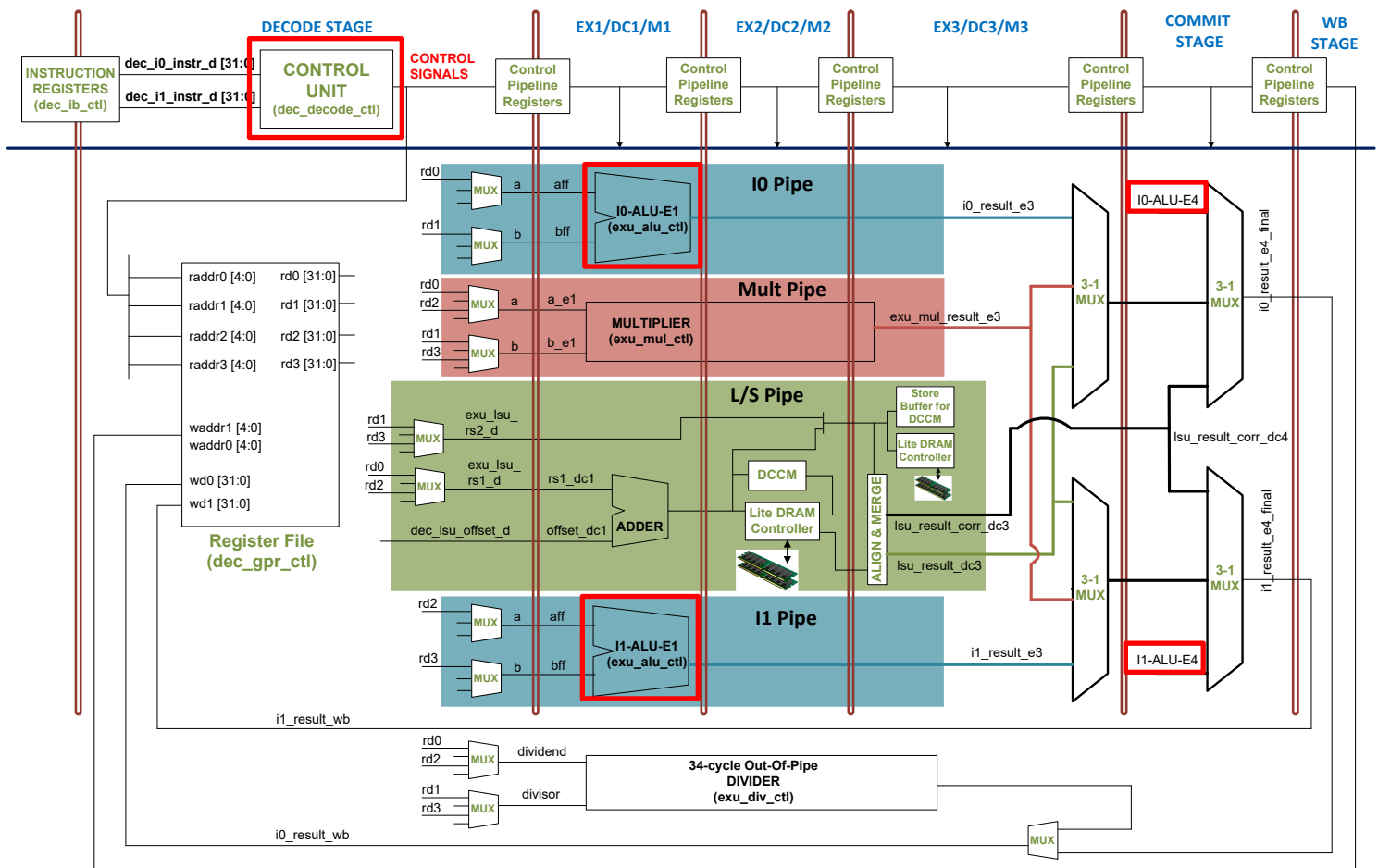


Figura 2. Andares de Decode, Execute, Commit e WriteBack do SweRV EH1

Neste exercício, damos instruções passo a passo sobre como adicionar uma nova instrução, neste caso `minu`. Depois, no Exercício 2, seguirá um procedimento semelhante para adicionar outras instruções `bitmanip`.

Modificações da Unidade de Controlo:

NOTA: Recomendamos que reveja a Secção 2.C.i do Lab 11 e a Secção 4 do documento `SweRVref.docx` antes de concluir os passos seguintes.

Agora vamos modificar/criar novos sinais de controlo necessários para suportar a nova instrução.

- Criar dois novos bits no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`. Estes dois bits informam o processador se uma instrução `minu` está a ser executada.
 - o Criar um novo bit, chamado `minu`, como parte da estrutura do tipo `dec_pkt_t` (Figura 3). Lembre-se que este é o principal tipo de estrutura utilizado na Unidade de Controlo.

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

Figura 3. Novo bit na estrutura `dec_pkt_t`

- Criar um novo bit, chamado *minu*, como parte da estrutura do tipo `alu_pkt_t` (Figura 4). Lembre-se que este é o tipo de estrutura específico utilizado para as instruções Aritmética-Lógica.

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

Figura 4. Novo bit na estrutura `alu_pkt_t`

- Atribuir um valor aos novos sinais de controlo no módulo `dec_decode_ctl` (implementado no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`).

- Atribuir o valor ao novo bit *minu* no andar de descodificação, usando os sinais `i0_dp_raw` e `i1_dp_raw`. Para isso, é preciso modificar as equações do módulo `dec_dec_ctl` (linhas 2497-2672 do ficheiro `dec_decode_ctl.sv`), como explicado a seguir (note que estas explicações estão resumidas nas linhas 2482-2495 do módulo `dec_decode_ctl`, de onde as obtivemos e estendemos um pouco):

1. O ficheiro

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode` é um ficheiro legível por pessoas que tem todos os descodificadores de instruções definidos no processador SweRV EH1, e que deve ser modificado como explicado a seguir para incluir a instrução *minu*.

- Na secção `.definition`, crie uma nova linha (Figura 5) para a nova instrução de acordo com o seu formato, mostrado na Figura 1.

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

Figura 5. Modificar a secção `.definition`

- Na secção `.output`, criar um novo bit chamado *minu* (Figura 6).

```
.output

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

Figura 6. Modificar a secção `.output`

- Na secção `.decode`, crie uma nova linha para a instrução `minu` (Figura 7). Para a nova instrução, devem ser ativados os mesmos bits que os ativados para uma instrução `add`, exceto o bit `add`. Ou seja: `alu`, `rs1`, `rs2`, `rd`, `pm_alu`. Além disso, o novo bit `minu` também deve ser ativado.

```
.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 }
```

Figura 7. Modificar a secção `.decode`

2. Na mesma pasta (`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/`), gerar as *equações gerais*, que, após a modificação do ficheiro de *descodificação*, incluem as instruções suportadas pelo SweRV EH1 mais a instrução `minu`.

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

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

Estes dois comandos irão gerar ficheiros *coredecode.e* e *equations*.

3. Na mesma pasta (`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/`), gerar a *equação legal*.

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

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

Estes dois comandos irão gerar os ficheiros *legal.e* e *legal_equations*.

4. Modificar o módulo `dec_dec_ctl`, substituindo as equações existentes (linhas 2497-2672 do ficheiro `dec_decode_ctl.sv`) pelas novas equações, tal como definidas nos ficheiros *equations* e *legal_equations*.

- No módulo **dec_decode_ctl**, atribuir um valor ao novo bit *minu* nos sinais *i0_ap* e *i1_ap*, utilizando os sinais *i0_dp* e *i1_dp* (Figura 8).

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

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

Figura 8. Atribuir valor aos bits *minu*

Estes passos descrevem o procedimento geral que deve ser seguido para modificar a unidade de controlo quando se inclui uma nova instrução no processador SweRV EH1.

Alterações na Unidade de Execução:

Em seguida, modifique a Unidade de Execução, que está implementada nos módulos **exu**, **exu_alu_ctl**, **exu_mul_ctl**, **exu_div_ctl** (os ficheiros que contêm estes módulos têm o nome dos módulos). Em exercícios futuros, analisaremos situações complexas em que é necessário um novo pipe completo. No entanto, neste exercício, apenas são necessárias algumas pequenas alterações no módulo **exu_alu_ctl** (Figura 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}} & lout[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});
```

Figura 9. Modificar as ALUs

Depois de concluir essas alterações, você está pronto para testar a nova instrução. Execute uma simulação no Verilator que ilustre o uso da nova instrução. Você pode usar o programa fornecido na Figura 10 ou pode criar seu próprio programa.

O programa da Figura 10 cria um ciclo infinito que calcula o valor mínimo de dois registos em cada iteração. Note-se que a nova instrução não pode ser usada normalmente (com uma mnemónica), mas tem de ser usada diretamente no formato de máquina, uma vez que o compilador RISC-V ainda não a suporta.

```
.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 # Repete o ciclo
    nop
.end
```

Figura 10 . Programa simples para testar a nova instrução, destacada a vermelho

Figura 11 mostra a simulação do Verilator (como de costume, usamos um script `.tc/` para incluir os sinais). A forma de onda mostra duas iterações do ciclo, o que mostra duas execuções da nova instrução (`ifu_i0_instr` ou `ifu_i1_instr` = 0x0BDE5F33). Os seus bits de controlo principais (`i0_dp_raw` ou `i1_dp_raw` = 0x7A000000000003) e os bits de controlo da ALU (`i0_ap` ou `i1_ap` = 0x180000) são os mesmos que numa instrução `add`, exceto o bit `minu` e o bit `add`. O resultado escrito em `t5` (mostrado na parte inferior da figura) é o valor mínimo dos dois números lidos em `t3` e `t4`. Note que a segunda execução de `minu` compara 0xFFFFFFFEE com 0x00000800; dado que é uma instrução *min sem sinal*, 0xFFFFFFFEE representa um número positivo grande, e assim o valor mínimo entre os dois é 0x00000800.

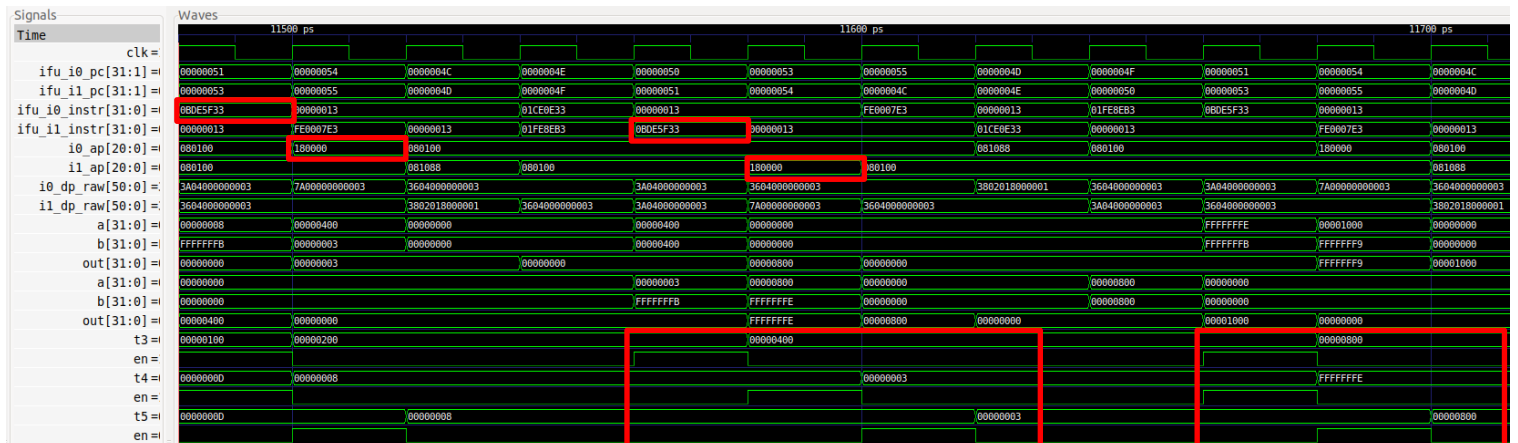


Figura 11 . Simulação Verilator do programa da Figura 10

Modifique o programa para efetuar comparações diferentes e, em seguida, simule o programa utilizando o Verilator.

Depois de ter verificado que a sua implementação funciona corretamente, gere o novo *bitstream* no Vivado e teste a nova instrução na placa utilizando qualquer um dos testes criados para simulação.

Construa um programa que leia os 16 interruptores e compare o valor binário dos 8 interruptores menos significativos com o valor binário dos 8 mais significativos, utilizando a nova instrução. Depois, apresente o valor mínimo nos mostradores de 7 segmentos.

Por fim, crie diferentes testes para confirmar que a instrução funciona como esperado e demonstre os resultados na placa.

2) Implementar outras instruções que pertencem à extensão *bitmanip* do RISC-V. Comece por completar as restantes instruções `min/max: min, max` e `maxu`.

3) Neste exercício, vai estender o processador SweRV EH1 para incluir três novas instruções que pertencem à extensão RISC-V Single-Precision Floating-Point (extensão F): `fadd.s`, `fmul.s` e `fdiv.s`.

- As instruções assumem que os operandos são representados no formato IEEE 754 de vírgula-flutuante de precisão simples (<https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>). Num número de vírgula flutuante, o registo está logicamente dividido em três campos: **Sinal** (1 bit), **Expoente** (8 bits) e **Mantissa** (23 bits).

Sinal | **E₇ ... E₀** | **M₂₂ ... M₀**

- A instrução `fadd.s rd, rs1, rs2` adiciona os dois valores de vírgula-flutuante em `rs1` e `rs2` e armazena o resultado em `rd`. A instrução `fmul.s rd, rs1, rs2` multiplica os dois valores de vírgula-flutuante em `rs1` e `rs2` e armazena o resultado em `rd`. Finalmente, a instrução `fdiv.s rd, rs1, rs2` divide os dois valores de vírgula-flutuante em `rs1` e `rs2` e armazena o resultado em `rd`.

- Os formatos utilizados para estas instruções, tal como definidos na extensão RISC-V F, são os seguintes

```
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
```

- Embora esta extensão pressuponha um processador com 32 registos de vírgula-flutuante, neste exercício, por uma questão de simplicidade, será utilizado o Register File existente utilizado por qualquer outra instrução (ou seja, os registos `x`). Além disso, assumimos outras simplificações: apenas uma instrução de vírgula-flutuante pode ser executada de cada vez e instruções de vírgula-flutuante são bloqueantes.

Para incluir o suporte para estas instruções no processador SweRV EH1, é necessário efetuar as seguintes modificações:

Alterações na Unidade de Execução:

Irá adicionar hardware para adição, multiplicação e divisão de vírgula-flutuante (pode encontrar algumas contribuições na Internet, como detalhamos abaixo). Depois vai usar este hardware quando uma instrução `fadd`, `fmul` ou `fdiv` for executada. Para o fazer, complete o seguinte:

- Descarregue o somador, multiplicador e divisor de vírgula-flutuante multi-ciclo fornecido em: <https://github.com/dawsonjon/fpu>. Estas são unidades não-

pipelined multi-ciclo semelhantes ao Divisor inteiro disponível no SweRV EH1.

- Apesar das novas unidades constituírem novos pipes e, portanto, poderem ser tratadas de forma independente, pode instanciar as três unidades de vírgula-flutuante dentro do módulo **exu_div_ctl**, dado que esta via de execução fornece alguns sinais que são úteis para suportar as novas instruções, tais como os sinais *finish* e *div_stall*. Se o fizer desta forma, deve ativar os mesmos bits de uma instrução *div*, mais os novos bits de vírgula-flutuante, ao gerar as equações para a Unidade de Controlo como explicado abaixo.

Alterações na Unidade de Controlo:

Modificar/criar novos sinais de controlo para suportar as novas instruções.

- Criar novos bits e tipos de estrutura no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`.
 - o Crie um novo tipo de estrutura chamado `fp_pkt_t` que inclui 3 bits: *fp_add*, *fp_mul* e *fp_div*, que indicam, respetivamente, se o processador está a executar uma adição de vírgula-flutuante, uma multiplicação de vírgula-flutuante ou uma divisão de vírgula-flutuante.
 - o Crie três novos bits, chamados *fp_add*, *fp_mul* e *fp_div*, que fazem parte do tipo de estrutura `dec_pkt_t`. Lembre-se que este é o principal tipo de estrutura utilizado na Unidade de Controlo.
- Atribuir um valor aos novos sinais de controlo no módulo **dec_decode_ctl** (implementado no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`).
 - o Atribua valores aos novos bits nos sinais *i0_dp_raw* e *i1_dp_raw*. Para o efeito, deve gerar novamente as equações do módulo **dec_dec_ctl**, como explicado no Exercício 1. Como mencionado acima, se gerir as novas instruções como uma instrução *div*, deve ativar os mesmos bits que uma instrução *div*, mais os novos bits de vírgula-flutuante, ao gerar as equações do módulo **dec_dec_ctl**.
 - o Criar um novo sinal do tipo `fp_pkt_t` chamado *fp_p*. Em seguida, atribua valores aos três bits dessa estrutura, usando os sinais *i0_dp* e *i1_dp*. Note que, à semelhança das instruções *mul* ou *div*, apenas é necessário um sinal deste tipo, porque apenas uma instrução de vírgula-flutuante pode ser executada num determinado ciclo.

Depois de modificar o hardware, efetue uma simulação no Verilator que ilustre a utilização das novas instruções. Pode utilizar o programa fornecido na Figura 12 ou pode criar o seu próprio programa. O programa da Figura 12 cria um ciclo infinito que calcula três instruções: adição, multiplicação e divisão de vírgula flutuante.

```
.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    # Repete o ciclo
    nop

.end
```

Figura 12 . Programa simples para testar as novas instruções, realçadas a vermelho

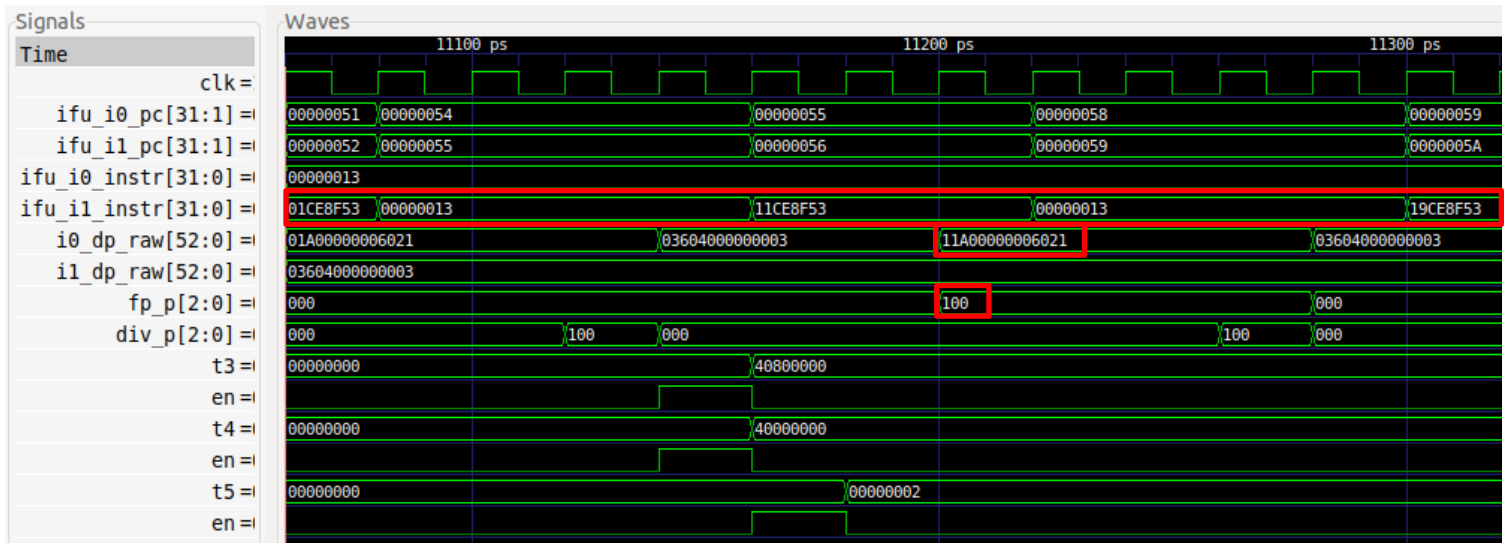
A Figura 13 mostra os resultados da simulação do Verilator. Para verificar os resultados, pode utilizar um conversor de vírgula-flutuante, como o que está disponível em: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.

Na Figura 13-a, as três instruções de vírgula-flutuante são introduzidas em `ifu_i0_instr` ou `ifu_i1_instr`. Os seus bits de controlo principais (`dec_pkt_t`) são os mesmos que os de uma instrução `div` (`i0_dp_raw` = 0x11A00000006021) com os três bits extra adicionados como descrito acima. Os bits de controlo FP (vírgula-flutuante) (`fp_pkt_t`) são 100 para `fadd` (como mostrado na figura), 010 para `fmul` e 001 para `fdiv`.

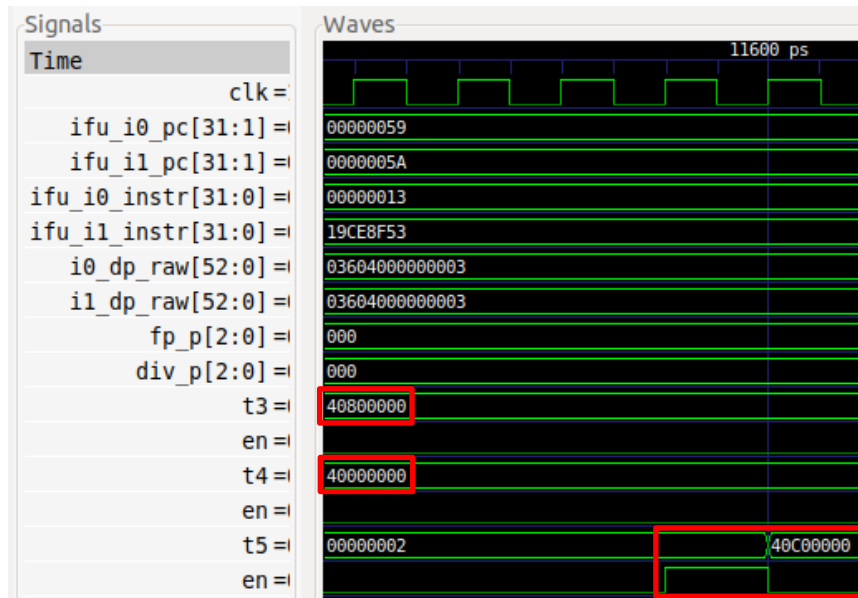
A Figura 13-b mostra a adição FP a escrever o seu resultado em `t5` vários ciclos depois. Note-se que os valores de entrada são 0x40800000 e 0x40000000, pelo que o resultado da adição é 0x40c00000.

A Figura 13-c mostra a multiplicação FP a escrever o seu resultado em `t5` vários ciclos depois. Note-se que os valores de entrada são 0x40800000 e 0x40000000, pelo que o resultado da multiplicação é 0x41000000.

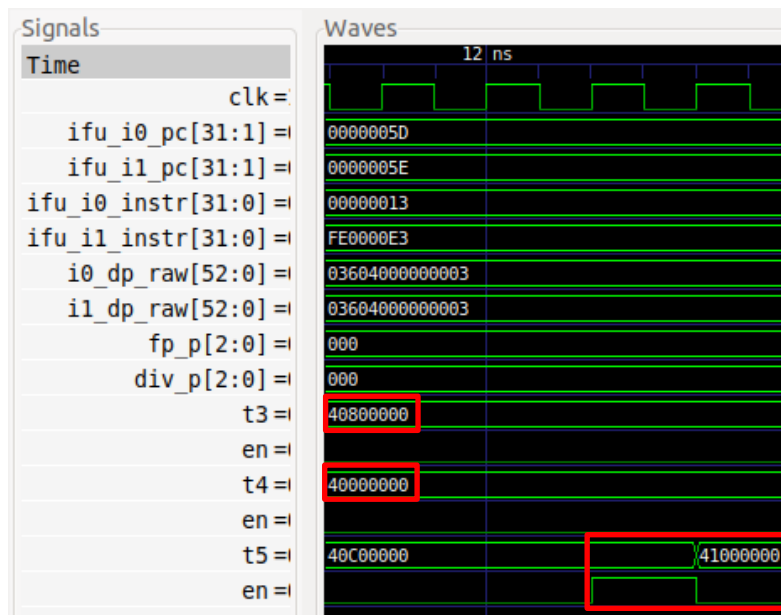
Finalmente, a Figura 13-d mostra a divisão FP a escrever o seu resultado em `t5` vários ciclos depois. Note-se que os valores de entrada são 0x40800000 e 0x40000000, pelo que o resultado da divisão é 0x40000000.



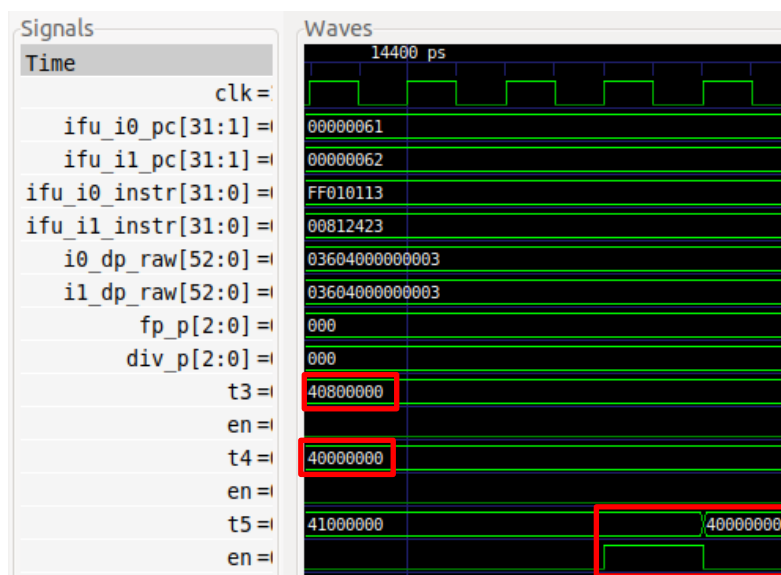
(a)



(b)



(c)



(d)

Figura 13 . Simulação Verilator do programa da Figura 12

Modificar o programa para testar outros casos e demonstrar que as instruções funcionam corretamente. Por exemplo, testar números negativos, dependências de dados com instruções anteriores/subsequentes, etc. Em seguida, simule-os utilizando o Verilator.

Em seguida, teste as novas instruções em hardware na placa. Para isso, programe o exemplo *DotProduct_C-Lang* fornecido no GSG, usando as novas instruções *fmul* e *fadd* para realizar os cálculos de virgula-flutuante. Compare a execução desse algoritmo quando as instruções de virgula-flutuante são emuladas vs. quando essas instruções são implementadas em hardware.

Também pode adicionar mais funcionalidades, como suporte para: outros formatos de vírgula-flutuante (como *precisão dupla*), outros modos de arredondamento de vírgula-flutuante, um novo Register File para os valores de vírgula-flutuante, a sua própria implementação da unidade FP, etc.

- 4) Implementar o método da bissecção. Pode encontrar muita informação sobre este algoritmo na Internet, por exemplo, em: https://en.wikipedia.org/wiki/Bisection_method.

Compare a execução deste algoritmo quando as instruções de vírgula-flutuante são emuladas e quando estas instruções são implementadas em hardware.

- 5) Implementar qualquer uma das instruções propostas nos exercícios do Capítulo 4 do livro "Computer Organization and Design - RISC-V Edition", de Patterson & Hennessy ([HePa]), como por exemplo

a. (de [HePa] Exercício 4.11):

- i. Instrução "Load With Increment": `lwi.d rd, rs1, rs2`
- ii. Interpretação: `rd = Mem[rs1 + rs2]`

b. (de [HePa] Exercício 4.12):

- i. Instrução "Swap": `swap rs1, rs2`
- ii. Interpretação: `rs2 = rs1; rs1 = rs2`

c. (de [HePa] Exercício 4.13):

- i. Instrução "Store Sum": `ss rs1, rs2, imm`
- ii. Interpretação: `Mem[rs1] = rs2 + imm`

- 6) À semelhança do exercício anterior, implemente as instruções propostas nos Exercícios 3-6 do Capítulo 7 do livro de S. Harris e D. Harris, "Digital Design and Computer Architecture: RISC-V Edition" [DDCARV]. Repetimos de seguida todas as instruções incluídas nestes quatro exercícios. Algumas delas já são suportadas pelo nosso processador SweRV EH1 e, nesse caso, em vez de as implementar, pode simplesmente explicar como são implementadas.

a. Exercício 3: `xor, sll, srl, bne`. (Já implementado no SweRV EH1)

b. Exercício 4: `lui, sra, lbu, blt, bltu, bge, bgeu, jalr, auipc, sb, slli, srai`. (Já implementado no SweRV EH1)

c. Exercício 5: `lwpostinc rd, imm(rs)` (a instrução é equivalente às duas instruções seguintes: `lw rd, 0(rs)` seguido de `addi rs, rs, imm`).

d. Exercício 6: `lwpreinc rd, imm(rs)` (a instrução é equivalente às duas instruções seguintes: `lw rd, imm(rs)` seguido de `addi rs, rs, imm`).

- 7) Incluir um novo evento para contar o número de instruções de tipo I executadas num programa. Fornecemos algumas orientações para o ajudar a realizar este exercício:

- É necessário modificar algumas estruturas do ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`. Especificamente, é necessário adicionar outro campo no seguinte tipo de estrutura:
 - Estrutura `inst_t`: novo campo para uma instrução de tipo I.
- Como sabe, os bits de controlo são atribuídos no módulo **dec_decode_ctl** (ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`). Modificar a atribuição dos sinais `i0_itype` e `i1_itype` para adicionar o novo tipo de instrução incluído no item anterior.
- Os contadores em hardware estão implementados no módulo **dec_tlu_ctl** (ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_tlu_ctl.sv`). Abra esse ficheiro e analise o código incluído nas linhas 1882 a 2143. Terá de modificar esta parte do código para incluir o novo contador.

Depois que o novo contador tiver sido incluído no código Verilog, depure a implementação usando o Verilator. Uma vez que a implementação tenha sido verificada através de simulação, gerar o novo fluxo de bits para o SoC e testar o funcionamento do novo contador em hardware na placa.