# TAREFAS

**TAREFA:** Verificar se esses 32 bits (0x01de0e33) correspondem à instrução add t3,t3,t4 na arquitetura RISC-V.

**0x01de0e33 🡪 0000000 11101 11100 000 11100 0110011**

**funct7 = 0000000**

**rs2 = 11101 = x29 (t4)**

**rs1 = 11100 = x28 (t3)**

**funct3 = 000**

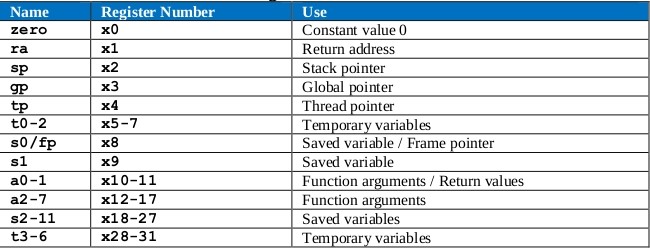
**rd = 11100 = x28 (t3)**

**op = 0110011**

Do Apêndice B do DDCARV:







|  |
| --- |
| **TAREFA:** Replicar a simulação da Figura 3 no seu computador. Para fazer isso, siga as próximas etapas (conforme descrito em detalhes na Seção 7 do GSG):   * Se necessário, gere o binário de simulação (*Vrvfpgasim*). * No PlatformIO, abra o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction*. * Defina o caminho correto para o binário de simulação do RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*. * Gere o trace da simulação com o Verilator (Generate Trace). * Abra o trace no GTKWave. * Use o ficheiro *test\_1.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/*) para abrir os mesmos sinais que os mostrados na Figura 3. Para isso, no GTKWave, clique em *File - Read Tcl Script File* e selecione o ficheiro *test\_1.tcl*. * Clique em *Zoom In* () várias vezes e vá para 15000ps. |

Solução fornecida no documento principal do Lab 12.

**TAREFA:** Localize as principais estruturas e sinais da Figura 6 nos ficheiros Verilog do processador SweRV EH1.

* Unidade de controlo no módulo **dec\_decode\_ctl**
* Register File:
  + Instanciação na linha 525 do módulo **dec**.
  + Implementação no módulo **dec\_gpr\_ctl**.
* Muxes 3:1 no andarcode: Linha 279 do módulo **exu**.
* Registos de pipeline para sinais de controlo: Distribuídos em vários módulos.
* Registos aff e bff: Linhas 90 e 92 do módulo **exu\_alu\_ctl**.
* I0 ALU em EX1:
  + Instanciação na linha 401 do módulo **exu**.
  + Implementação no módulo **exu\_alu\_ctl**.
* Registos do pipeline com o resultado da operação (i0e2resultff, i0e3resultff, i0e4resultff, i0wbresultff): Linhas 2260-2283 do módulo **dec\_decode\_ctl**.
* Mux 3:1 no andar EX3: Linha 2268 do módulo **dec\_decode\_ctl**.
* Mux 3:1 no andar EX4: Linha 2277 do módulo **dec\_decode\_ctl**.
* Mux 2:1 no andar Writeback: Linha 2286 do módulo **dec\_decode\_ctl**.

**TAREFA:** Encontre no código Verilog (módulo **dec\_decode\_ctl**) como o sinal de controlo i0r é usado para ler o Register File.

* Os identificadores de registo são obtidos da instrução de 32 bits na Via-0: sinal i0[31:0] = dec\_i0\_instr\_d[31:0].

Numa instrução do tipo R, eles estão localizados nos seguintes campos:

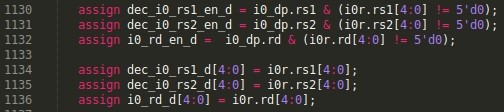


No módulo **dec\_decode\_ctl**:



* Os identificadores de registo e os sinais de habilitação de leitura são atribuídos a dec\_i0\_rs1\_d/dec\_i0\_rs2\_d e dec\_i0\_rs1\_en\_d/ dec\_i0\_rs2\_en\_d.

Estes sinais são enviados do módulo **dec** para o módulo **dec\_decode\_ctl**. No módulo **dec\_decode\_ctl**:

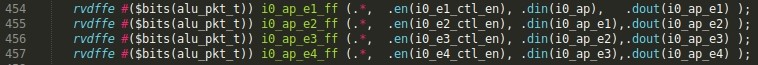


* Os identificadores de registo e os sinais de habilitação de leitura são fornecidos ao Register File, que é instanciado no módulo **dec**. No módulo **dec**:



**TAREFA:** Encontre no código Verilog (módulo **exu**) como os sinais de controlo i0\_ap e dd são propagados do andar Decode para o andar Execução. Além disso, descubra como o sinal de controlo dd é usado pelo Register File no andar Write-Back, depois de passar por todos os andares Decode a Writeback.

O sinal i0\_ap é obtido no módulo **dec\_decode\_ctl**. Ele é fornecido ao módulo exu, onde é propagado para EX1, EX2, EX3 e Commit (EX4). No módulo **exu**:



O sinal dd é obtido no módulo **dec\_decode\_ctl** e propagado para EX1, EX2, EX3, Commit (EX4) e WB (EX5). No módulo **dec\_decode\_ctl**:



Note que a saída de cada registo é ligeiramente modificada (e, portanto, renomeada) antes de entrar no próximo registo. Pode consultar o código Verilog se quiser verificar os detalhes.

O identificador de registo para o operando de saída é atribuído no andar Decode:



O sinal dd é propagado da Decodificação para o Writeback, conforme mostrado acima: dd 🡪 e1d 🡪 e2d 🡪 e3d 🡪 e4d 🡪 wbd. Em seguida, o registo de destino é fornecido ao Register File no andar Writeback:





**TAREFA:** A geração desses dois sinais (i0\_e1\_ctl\_en e dec\_i0\_alu\_decode\_d) é um processo bastante complexo que não explicamos aqui em detalhe, mas que pode analisar por conta própria nos módulos **dec\_decode\_ctl** e **exu**.

Solução não fornecida.

**TAREFA:** Localize no código Verilog (módulo **exu**) o multiplexer 3:1 na parte inferior (segundo operando de entrada) e tente encontrar a origem de suas entradas (na Figura 6, apenas a entrada proveniente do Register File é mostrada). Não é necessário examinar as entradas com muita atenção, pois elas serão analisadas nos exercícios propostos na Seção 3 e em Labs futuros.



Estes muxes 3:1 recebem 3 entradas:

* Um do Register File (gpr\_i0\_rs2\_d)
* Um do Instruction Register de 32 bits, que constitui o imediato (dec\_i0\_immed\_d)
* Um da lógica de bypass, que analisamos no Lab 15 (i0\_rs2\_bypass\_data\_d)

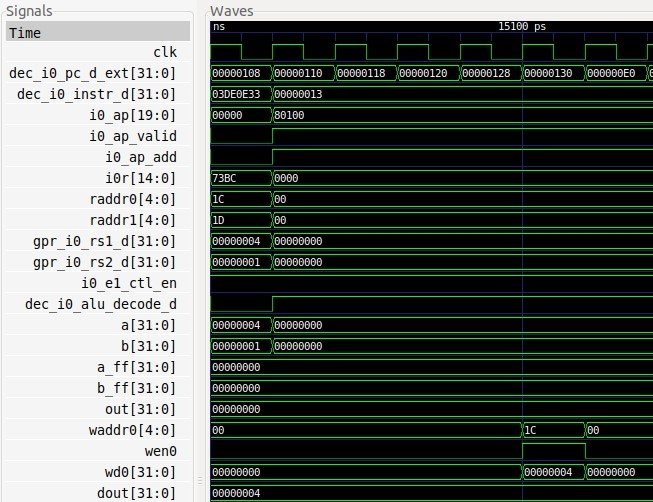
**TAREFA:** Replicar a simulação da Figura 7 no seu computador. Pode usar o script *.tcl* fornecido em:

*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/test\_2.tcl*. Observe que os aliases são usados nesse ficheiro *.tcl* para alguns dos bits de controlo.

Solução fornecida no documento principal do Lab 12.

**TAREFA:** No exemplo da Figura 2, substitua a instrução add por uma instrução não A-L (como uma instrução mul). Verifique se o sinal i0\_ap tem todos os seus campos iguais a 0 e se isso faz com que a ALU I0 não funcione (verá que os sinais a\_ff e b\_ff para o Pipe I0 no andar EX1 permanecem estáveis para essa instrução). Pode usar o mesmo ficheiro *test\_2.tcl* usado no exemplo da Figura 7.

Por exemplo, a simulação de mul t3, t3, t4 (0x03de0e33) fornece os seguintes resultados:

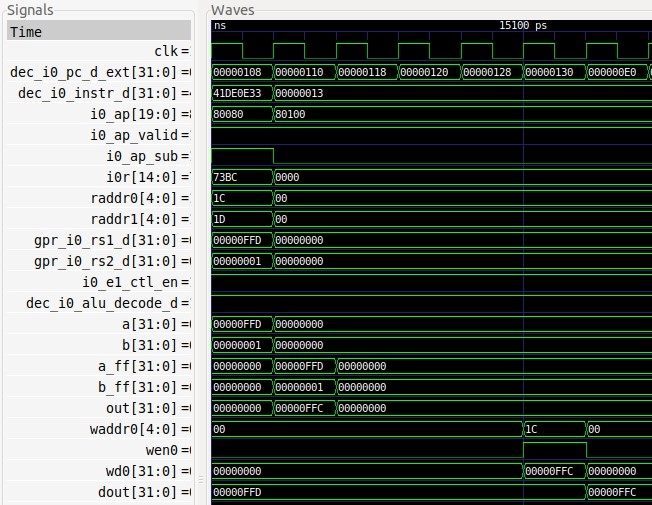


**TAREFA:** Inclua os novos sinais analisados nesta seção na simulação da Figura 7.

Solução não fornecida.

**TAREFA:** Realize uma simulação de uma sub-instrução semelhante à da Figura 7. Lembre-se de que pode incluir novos sinais na simulação por meio do ficheiro *.tcl*.

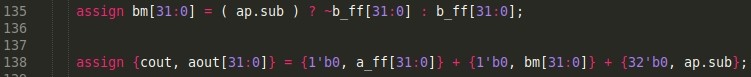
Por exemplo, a simulação de sub t3, t3, t4 (0x41de0e33) fornece os seguintes resultados:



**TAREFA:** Analisar a implementação Verilog do somador/subtrator implementado no módulo **exu\_alu\_ctl**. A Figura 8 lhe dá alguma ajuda, mostrando a lógica diretamente relacionada às operações de adição e subtração.

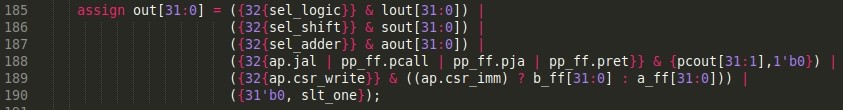


Os operandos de entrada são propagados do andar Decode (a e b) para o andar Execução (a\_ff e b\_ff).



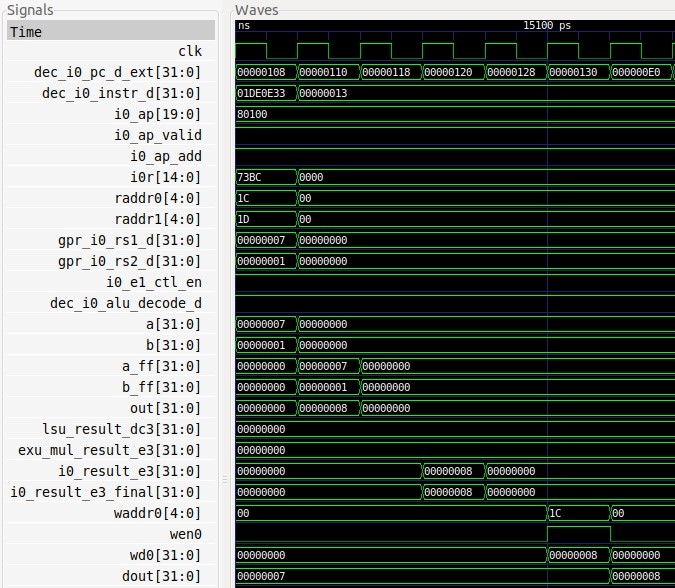
Esse é o somador/subtrator.

* Se a instrução for uma adição, aout = a\_ff + b\_ff
* Se a instrução for uma subtração, b\_ff será primeiro complementado por dois e, em seguida, a\_out será computado.

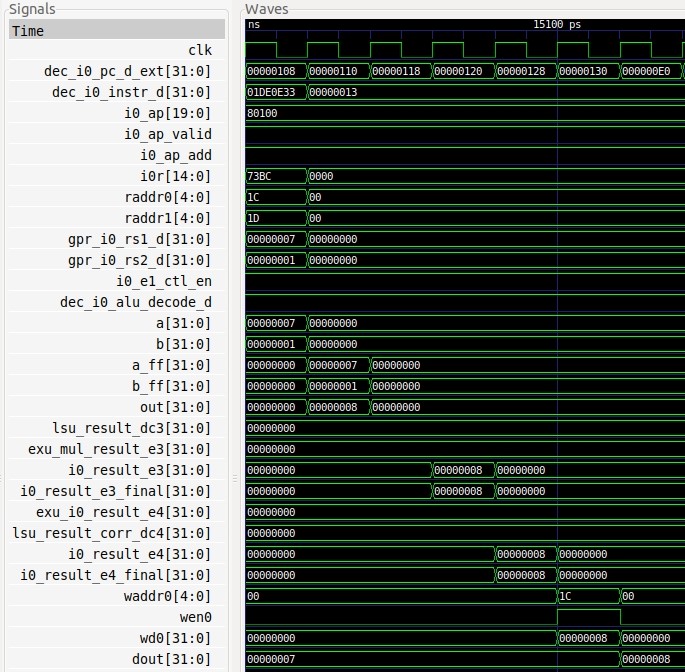


Se a instrução for uma adição ou uma subtração, então out = aout.

**TAREFA:** Verificar na simulação se esse multiplexer seleciona o resultado do Pipe esperado para a instrução add, para o exemplo da Figura 2.



**TAREFA:** Verificar na simulação se esse multiplexer seleciona o resultado da fonte de entrada adequada (i0\_result\_e4) para a instrução add do nosso exemplo da Figura 2.



**TAREFA:** No código Verilog, analise como os sinais wen0 e waddr0 são gerados no andar Decode e propagados para o andar Writeback.



# EXERCÍCIOS

1) Faça uma análise semelhante à apresentada neste Lab para instruções lógicas (and, or, xor).

O exemplo a seguir, fornecido em *[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction*, ilustra a execução de uma instrução and contida num ciclo que se repete para sempre. Como no exemplo da instrução add, a instrução and (destacada em vermelho) é rodeada por várias instruções nop. Duas instruções são incluídas no final do ciclo para modificar os valores armazenados em t3 e t4.

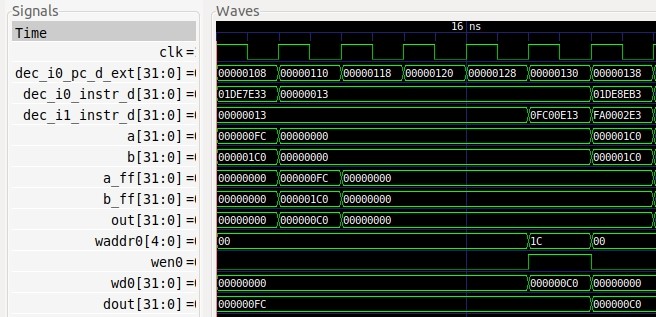
|  |
| --- |
| #define INSERT\_NOPS\_1 nop;  #define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1  #define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2  #define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3  #define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4  #define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5  #define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6  #define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7  #define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8  #define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9  .globl main  main:  li t3, 0xFC # t3 = 0xFC  li t4, 0x7 # t4 = 0x7    REPEAT:  INSERT\_NOPS\_10  **and t3, t3, t4** # t3 = t3 & t4  INSERT\_NOPS\_10  li t3, 0xFC # t3 = 0xFC  add t4, t4, t4  beq zero, zero, REPEAT # Repete o ciclo  .end |

Se abrir o projeto no PlatformIO, compilá-lo e abrir o ficheiro Disassembly (disponível em

*[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction/.pio/build/swervolf\_nexys/firmware.dis*) irá ver que a instrução and é colocada no endereço 0x00000108, e também pode ver o código de máquina da instrução (0x01de7e33):

**0x00000108: 01de7e33 and t3,t3,t4**

Em seguida, simulamos o programa no Verilator e abrimos o ficheiro trace gerado pelo simulador no GTKWave. Vá para qualquer iteração do ciclo, exceto a primeira.



Analise a forma de onda (os valores destacados em vermelho correspondem à instrução and). Neste Lab, saltamos os andares Fetch e Align, que serão explicados num próximo Lab.

* Andar **Decode**: O sinal dec\_i0\_pc\_d\_ext contém o endereço da instrução (nos livros, isso geralmente é chamado de Program Counter), que para o and é 0x00000108, e o sinal dec\_i0\_instr\_d contém a instrução de máquina de 32 bits 0x01DE7E33 (nos livros, isso geralmente é chamado de Instruction Register).

No RISC-V, o opcode para a instrução and é (consulte o Apêndice B de [Harris&Harris]):

0000000 | rs2 | rs1 | 111 | rd | 0110011

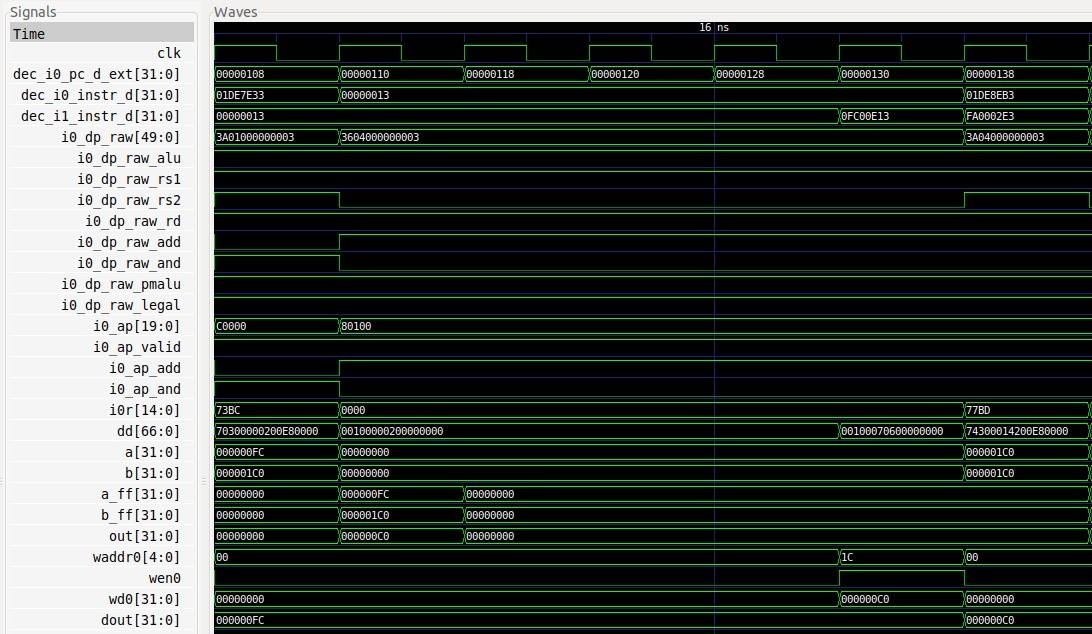
Assim, pode verificar facilmente que 0x01DEFE33 corresponde a: and t3, t3, t4 (lembre-se de que t3=x28 e t4=x29).

Neste andar, os **sinais de controlo do pipeline são gerados** (mostraremos alguns detalhes na próxima seção). Além disso, os

**O Register File é lido** nesse andar. Os sinais a e b contêm as entradas para a ALU, que, nesse caso, coincidem com os valores lidos do Register File (noutros casos que analisaremos nos próximos Labs, esse não será o caso).

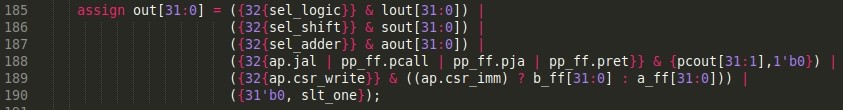
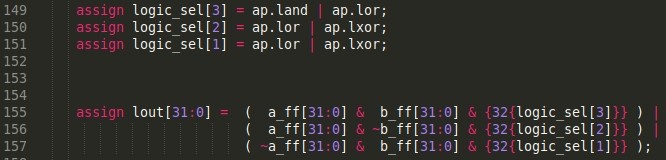
* Andar **EX1**: no próximo ciclo, a instrução and é **executada**. Os sinais a\_ff e b\_ff contêm as entradas para a ALU (0xFC e 0x1C0, respectivamente), enquanto out contém o resultado da adição (0xC0).
* Andar **EX5**, também chamado de **Writeback**: Finalmente, 4 ciclos depois, o resultado da adição é **escrito de volta** no Register File por meio do sinal wd0=0xC0, que contém os dados a serem gravados. Dado que wen0=1 (habilitação de gravação), o resultado da operação and é gravado no final desse ciclo no registo x28 (o índice do registo, waddr0=0x1C). É possível observar que, no ciclo seguinte (último ciclo mostrado na figura), o registo x28 contém o novo valor (dout=0xC0).

Em seguida, adicionamos os sinais de controlo à simulação anterior:



Pode ver que o bit de controlo para a instrução and é 1 no primeiro ciclo.

Os segmentos de código Verilog a seguir mostram a unidade lógica do SweRV EH1.



Quando o bit de controlo da and é 1, o resultado da operação and é selecionado:

logic\_sel[3]=1 and logic\_sel[2]=logic\_sel[1]=0 🡪 lout = a\_ff & b\_ff

|  |
| --- |
| 2) (*O exercício a seguir é baseado no exercício 4.1 do livro "Computer Organization and Design - RISC-V Edition", de Patterson & Hennessy ([HePa]*)*.*  Considere a seguinte instrução: and rd, rs1, rs2   1. Quais são os valores dos sinais de controlo gerados pelo SweRV EH1 para essa instrução? 2. Quais recursos (blocos) que desempenham uma função útil para essa instrução? 3. Quais recursos (blocos) que não produzem saída para essa instrução? Quais recursos produzem saída que não é usada? |

Solução não fornecida.

3) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, as instruções de *deslocamento à esquerda/direita* disponíveis no conjunto de instruções do RV32I Base Integer: srl, sra e sll.

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0xEEEEEEEE

li t4, 0x1

REPEAT:

srl t0, t3, t4

INSERT\_NOPS\_7

sra t1, t3, t4

INSERT\_NOPS\_7

sll t2, t3, t4

INSERT\_NOPS\_6

beq zero, zero, REPEAT # Repete o ciclo

.end

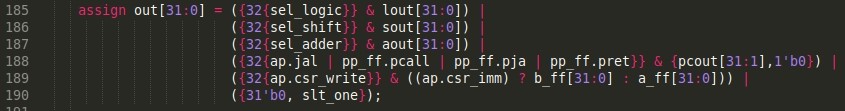


Os segmentos de código Verilog a seguir mostram a unidade de deslocamento (Shift Unit) do SweRV EH1.









4) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, as instruções *set-less-than* disponíveis no Instruction Set do RV32I Base Integer: slt e sltu.

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0x80000007

li t4, 0x6

REPEAT:

slt t0, t3, t4

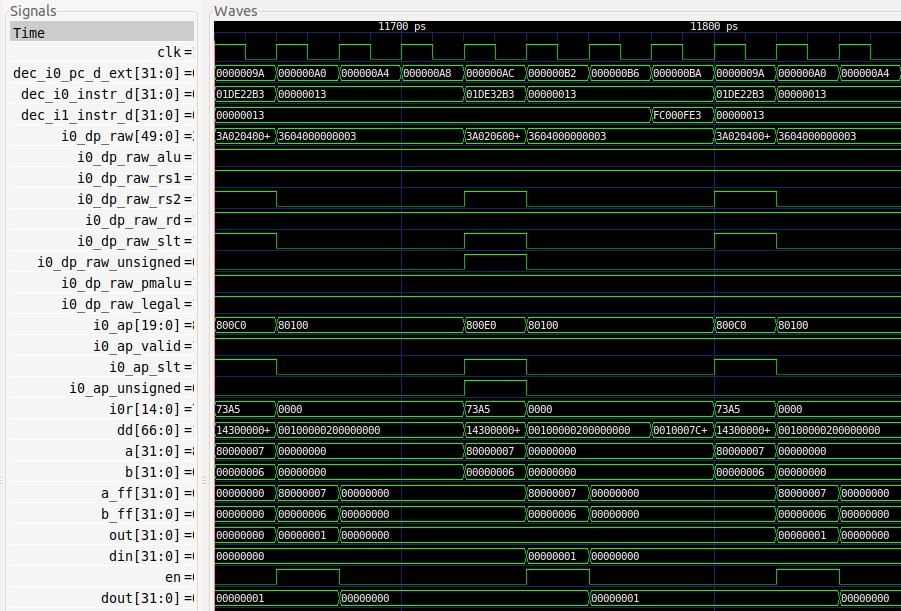
INSERT\_NOPS\_7

sltu t0, t3, t4

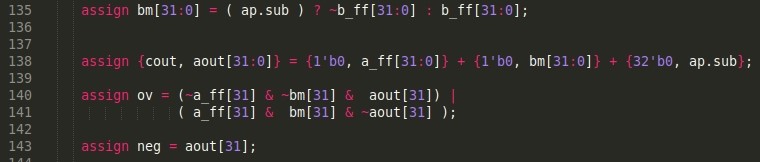
INSERT\_NOPS\_6

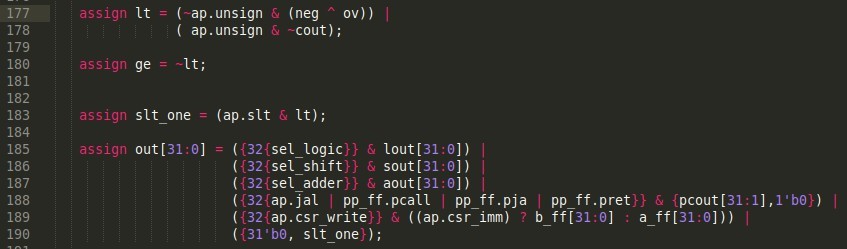
beq zero, zero, REPEAT # Repete o ciclo

.end



Os segmentos de código Verilog a seguir mostram a lógica que executa essas operações no SweRV EH1.





5) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, algumas das instruções *imediatas* disponíveis no Instruction Set do RV32I Base Integer: addi, andi, ori, xori, srli, srai, slli, slti e sltui.

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0x4 # t3 = 4

INSERT\_NOPS\_1

REPEAT:

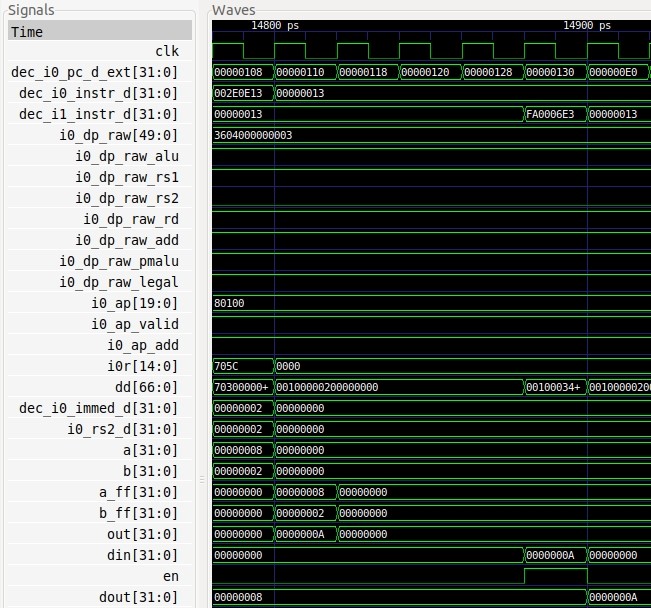
INSERT\_NOPS\_10

addi t3, t3, 2 # t3 = t3 + t4

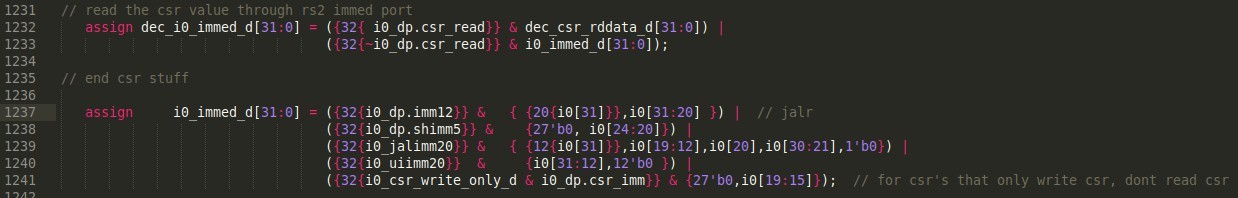
INSERT\_NOPS\_10

beq zero, zero, REPEAT # Repete o ciclo

.end



No módulo **dec\_decode\_ctl,** o imediato de 32 bits é computado.



No módulo **exu,** a entrada *rs2* adequada é selecionada. Nesse caso, usamos *dec\_i0\_immed\_d*.

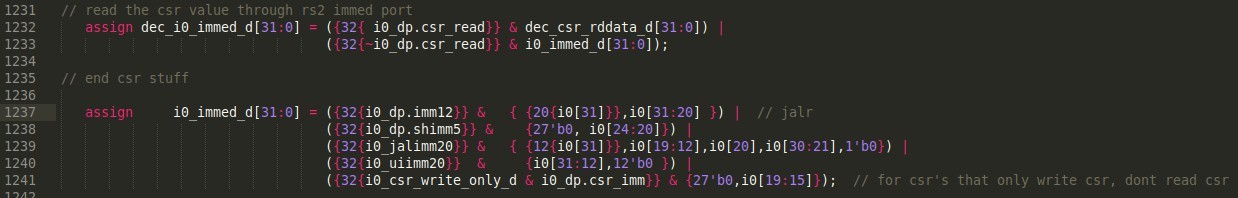


No módulo **dec\_gpr\_ctl**, o sinal de habilitação *rden1* determina se o ficheiro de registo é acessado para o segundo operando ou não. Se uma instrução usar um operando imediato: i0\_dp.rs2=0 🡪 rden1=0 🡪 rd1[31:0]=0x00000000 🡪 gpr\_i0\_rs2\_d[31:0]=0x00000000.



|  |
| --- |
| 6) (*O exercício a seguir é baseado no exercício 4.6 de [HePa]*)*.*  A Figura 5 não aborda instruções do tipo I, como addi ou andi.   1. Quais blocos lógicos adicionais, se houver, são necessários para dar suporte à execução de instruções do tipo I no SweRV EH1? Adicione os blocos lógicos necessários à Figura 5 e explique sua finalidade. 2. Liste os valores dos sinais gerados pela unidade de controlo para addi. |

Uma das entradas para os dois multiplexers 3:1 no andar Decode vem do imediato no sinal dec\_i0\_immed\_d[31:0]. O imediato é um sinal de 32 bits que é computado de forma diferente, dependendo da instrução I-Type que é executada. É um subconjunto de 32 bits que compõem a instrução, que são selecionados e estendidos por sinal da seguinte forma:



Os valores dos sinais de controlo para o addi podem ser vistos na simulação do Exercício 5.

|  |
| --- |
| 7) (*O exercício a seguir é baseado no exercício 4.4 de [HePa] e no exercício 1 do Capítulo 7 do livro de S. Harris e D. Harris, "Digital Design and Computer Architecture": RISC-V Edition*" *[DDCARV]*)*.*  Quando os chips de silício são fabricados, os defeitos nos materiais (por exemplo, silício) e os erros de fabricação podem resultar em circuitos defeituosos. Um defeito muito comum é uma ligação de um sinal ficar "quebrada" e registrar sempre um 0 lógico, o que é chamado de falha permanente ou "stuck-at-0".  Determine o efeito de cada um dos bits de controlo incluídos no sinal i0\_ap (um sinal do tipo alu\_pkt\_t) ficar preso em 0. |

O tipo de estrutura é definido no ficheiro swerv\_types.sv:

typedef struct packed {

logic valid;

logic land;

logic lor;

logic lxor;

logic sll;

logic srl;

logic sra;

logic beq;

logic bne;

logic blt;

logic bge;

logic add;

logic sub;

logic slt;

logic unsign;

logic jal;

logic predict\_t;

logic predict\_nt;

logic csr\_write;

logic csr\_imm;

} alu\_pkt\_t;

* Sinal valid em ”stuck-at-0”: não seria possível executar nenhuma instrução A-L, pois qualquer instrução A-L seria considerada inválida.
* Sinais land, lor, lxor, sll, srl, sra, beq, bne, blt, bge, add, sub, slt e jal em ”stuck-at-0”: para cada um desses bits, não seria possível executar a instrução A-L correspondente; por exemplo, se land estiver em ”stuck-at-0”, não será possível executar uma instrução and.
* Sinal unsign em ”stuck-at-0”: não seria possível comunicar ao processador que a operação deve ser sem sinal.
* Sinais predict\_t e predict\_nt: Não seria possível comunicar ao processador que um salto foi previsto como tomado ou não tomado.
* Sinais csr\_write e csr\_imm: não seria possível escrever ou operar com um imediato no Registo CSR.