



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 12

Instruções Aritméticas/Lógicas:

A instrução add

1. INTRODUÇÃO

Neste laboratório, analisamos o fluxo de instruções aritméticas e lógicas através dos andares do pipeline do SweRV EH1. Figura 1 mostra uma visão de alto-nível da microarquitetura do EH1, com andares que analisamos neste laboratório destacados a vermelho: Decode, EX1, EX2, EX3, Commit (às vezes chamado de EX4) e Writeback do Pipe I0. (O Pipe I1 é quase idêntico ao Pipe I0, mas adiamos a sua análise detalhada para o Lab 17, quando estudaremos o processamento superescalar. Também analisamos os andares Fetch e Align nos Labs 11 e 16).

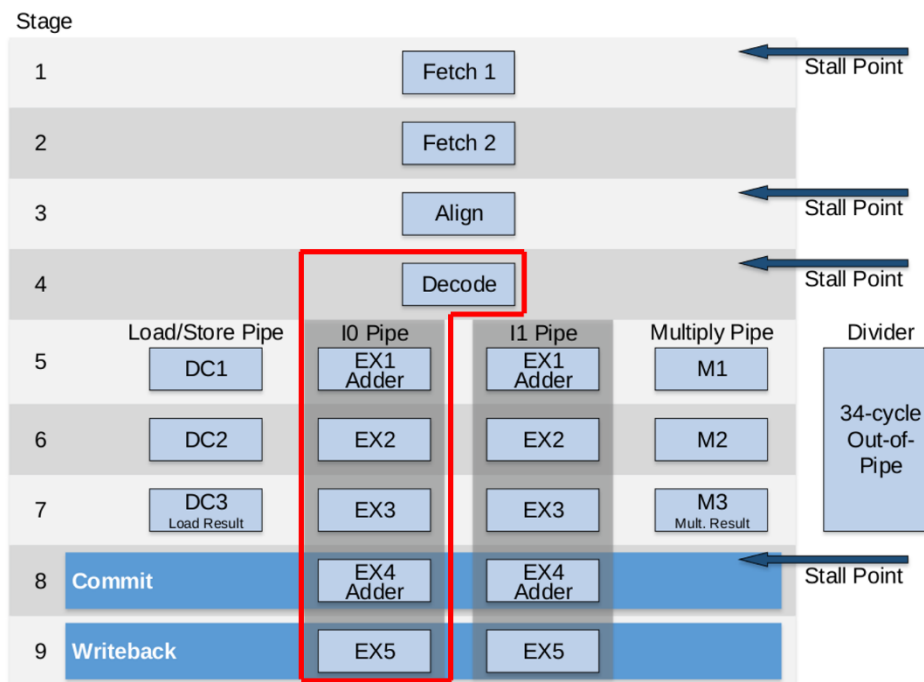


Figura 1. Pipeline SweRV EH1: andares 4-9 da instrução adição em destaque

Na secção 2 analisamos a instrução `add` desde o andar de Decode até ao andar de Writeback, quando escreve o resultado no Register File. Durante as explicações, intercalamos uma simulação de uma instrução `add` que deve replicar no seu próprio computador. Na Secção 3, apresentamos exercícios de análise de outras instruções Aritmético-Lógicas seguindo um procedimento semelhante ao descrito para a instrução `add`.

2. ANÁLISE DO NÚCLEO SweRV EH1 PARA UMA INSTRUÇÃO ADIÇÃO

Ao longo desta secção, trabalharemos com o exemplo apresentado na Figura 2 que executa uma instrução `add` dentro de um ciclo que se repete para sempre. A pasta `[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction` fornece o projeto PlatformIO para que se possa analisar, simular e alterar o programa como desejado. Como explicado na Secção 2 do documento SweRVref, por uma questão de simplicidade, neste projeto descartamos a utilização de instruções comprimidas. Além disso, por conveniência, inserimos a instrução `add` num ciclo infinito, o que nos permite inspecionar o programa sem *cache miss* de instruções (I\$) se evitarmos a primeira iteração para a nossa análise. Isto também facilita a localização da região de interesse na simulação. Finalmente, como também fizemos no

exemplo incluído nesse laboratório, a instrução `add` (destacada a vermelho na Figura 2) é rodeada por várias instruções `nop` (no-operation) de modo a isolá-la das instruções `add` precedentes/subsequentes que pertencem a outras iterações do ciclo.

```
.globl principal
principal:

li t3, 0x4 # t3 = 4
li t4, 0x1 # t4 = 1

REPETIR:
    INSERT_NOPS_10
    adicionar t3, t3, t4 # t3 = t3 + t4
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repete o ciclo

.fim
```

Figura 2. Exemplo de uma instrução de `add`

Se abrir o projeto no PlatformIO, construí-lo e abrir o ficheiro de Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/.pio/build/swervolf_nexys/firmware.dis`) verá que a instrução `add` (0x01de0e33) é colocada no endereço 0x00000108 neste programa.

```
0x00000108: 01de0e33  addt3 ,t3,t4
```

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

A. Análise Básica da Instrução `add`

Figura 3 mostra a simulação do Verilator do programa da Figura 2 para a execução da instrução `add` na quarta iteração do laço. A figura inclui alguns sinais associados aos andares Decode, EX1 e Writeback (WB). Os valores destacados a vermelho correspondem à instrução `add` à medida que atravessa estes três andares através do Pipe I0. Note que os sinais mostrados na figura correspondem ao Pipe I0.

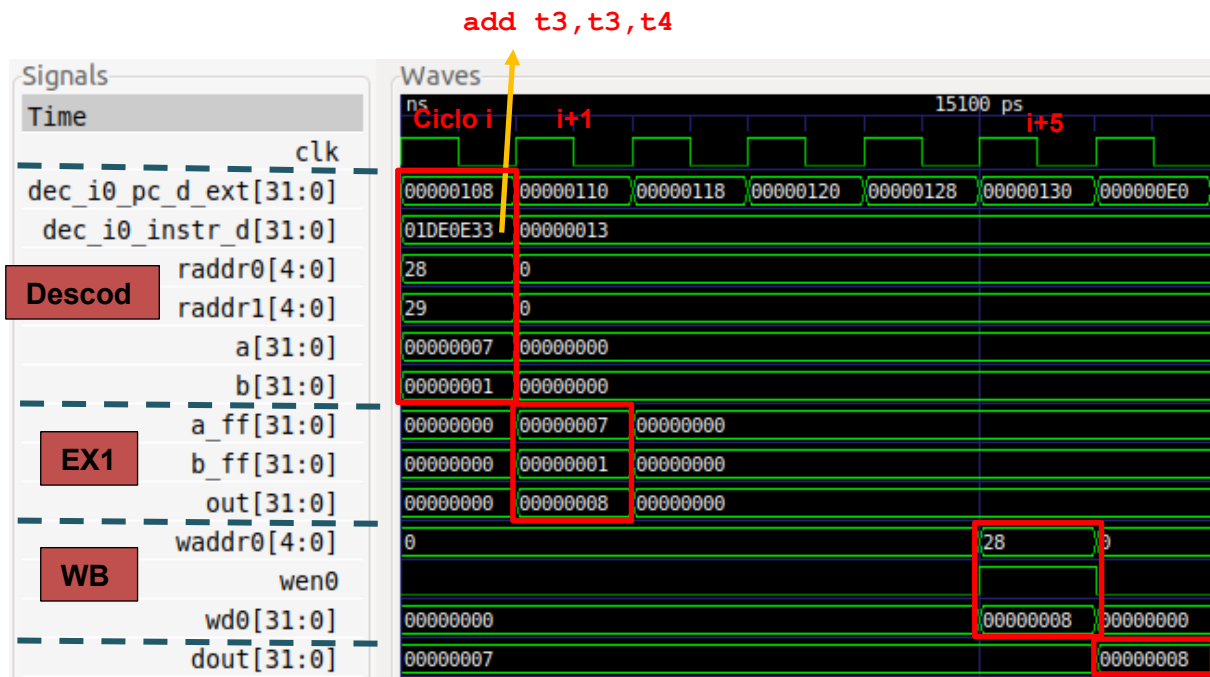


Figura 3. Simulação do Verilator para o programa de exemplo da Figura 2

Figura 4 mostra um diagrama simplificado do pipeline SweRV EH1 executando a instrução `add` durante a quarta iteração do ciclo (ver programa em Figura 2) através do Pipe I0. Note-se que a figura junta o estado do processador em diferentes ciclos:

- **Ciclo i:** **Descodificação:** A instrução é descodificada e o Register File é lido. A instrução `add` é enviada através do Pipe I0.
- **Ciclo i+1:** **EX1:** A adição é calculada pela ALU.
- **Ciclo i+5:** **Writeback:** O resultado da adição é escrito no Register File utilizando o porto de escrita 0.

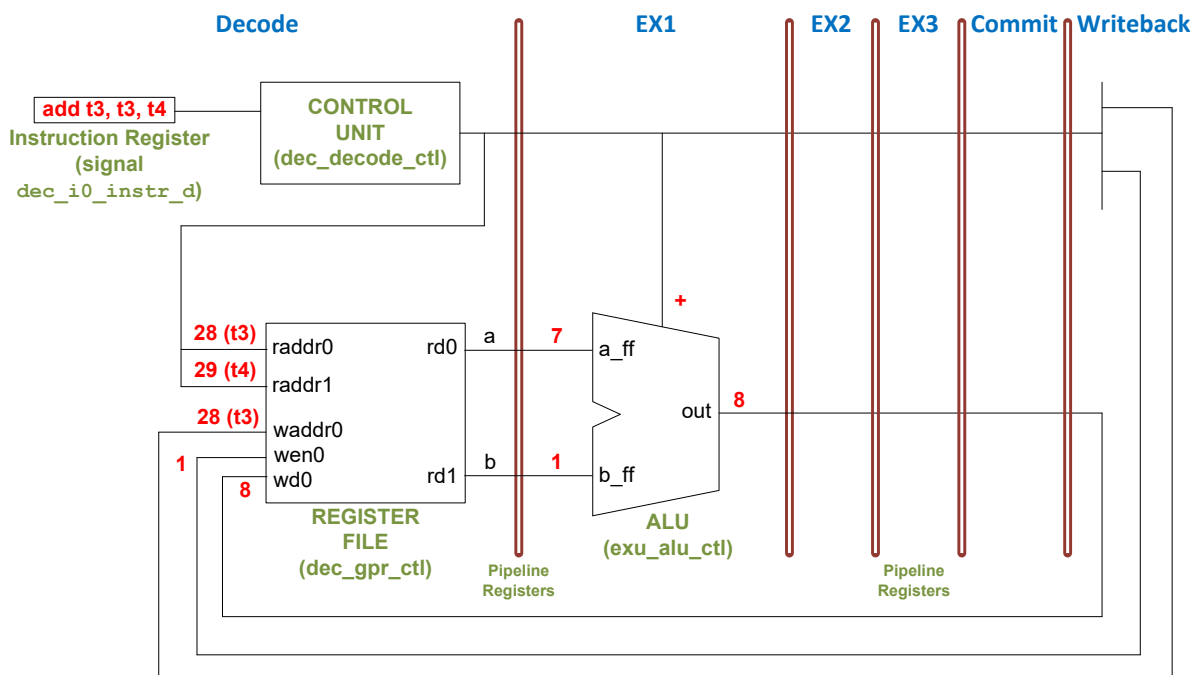



Figura 4. Pipeline SweRV EH1 a executar uma instrução de add

TAREFA: Replicar a simulação da Figura 3 no seu próprio computador. Para o fazer, siga os passos seguintes (descritos em pormenor na Secção 7 das GSG):

- Se necessário, gerar o binário de simulação (*Vrvfpgasim*).
- Na PlatformIO, abra o projeto fornecido em:
[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction.
- Estabelecer o caminho correto para o binário de simulação RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
- Gerar o *trace* da simulação com o Verilator (Generate Trace).
- Abrir o *trace* no GTKWave.
- Usar 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*.
- Clicar várias vezes em *Zoom In* () e passar para 15000ps.

Seguimos a instrução `add` através do pipeline, analisando a forma de onda da Figura 3 e o diagrama da Figura 4 ao mesmo tempo, e como descrito abaixo.

- **Ciclo i: Decode:** O sinal `dec_i0_instr_d` contém a instrução de máquina de 32 bits `0x01DE0E33`. Em RISC-V, o opcode para a instrução `add` é (ver Apêndice B de [DDCARV]):

00 | rs1 | 000 | rd | 0110011

Podemos facilmente verificar que `0x01DE0E33` corresponde a: `add t3, t3, t4` (lembre-se que `t3=x28` e `t4=x29`).

Durante este andar, são gerados **sinais de controlo** e o **ficheiro de registos é lido**. No andar seguinte (EX1), os operandos serão enviados para a ALU no pipe l0. Os sinais

`raddr0` e `raddr1` (mostrados em decimal nas figuras) contêm os dois números de registo de origem da instrução `add`, e os sinais `a` e `b` contêm os valores que serão enviados para a ALU no andar seguinte (EX1). Neste caso, `a` e `b` são os valores lidos do Register File. Para outras instruções, `a` e `b` podem ser valores diferentes; por exemplo, `b` pode ser um imediato. Analisaremos outras instruções nos laboratórios posteriores.

- Ciclo `i+1`: **EX1**: A instrução `add` é **executada**. Os sinais `a_ff` e `b_ff` contêm as entradas para a ALU (neste caso, 7 e 1, respetivamente), enquanto o sinal `out` contém o resultado da adição (8).
- Ciclo `i+5`: **Writeback**: Finalmente, 4 ciclos depois, o resultado da adição é **escrito de novo** no Register File através do sinal `wd0 = 0x8`, que contém os dados a escrever. Dado que `wen0 = 1` (write enable) neste ciclo, o resultado da adição é escrito no final do ciclo no registo `x28` (indicado em decimal, `waddr0 = 28`). Pode observar que, no ciclo seguinte (último ciclo mostrado na figura), o registo `x28` foi atualizado com o novo valor (`dout = 8`).

Lembre-se que o GTKwave permite-lhe alterar facilmente o formato dos dados de um sinal. Para isso, coloque o cursor sobre o sinal, clique com o botão direito do rato e seleccione o "Formato de dados" desejado. Por exemplo, pode ser mais conveniente ver `waddr0` em formato decimal (28) em vez de hexadecimal (0x1C), como mostrado na Figura 5.



Figura 5. Sinal `waddr0` apresentado em formato decimal.

B. Análise avançada da instrução `add`

Nesta secção, analisamos os andares percorridos pela instrução `add`, desde o Decode até ao Writeback, com mais pormenor do que na secção A e adicionamos progressivamente mais sinais à simulação a partir de Figura 3.

Figura 6 mostra um diagrama detalhado dos principais elementos que uma instrução `add` atravessa durante a sua execução através do Pipe I0. Isto já foi ilustrado na Figura 4 do Lab 11 (recomendamos a comparação de ambas as figuras), mas agora concentramo-nos apenas no Pipe I0 e fornecemos detalhes relacionados com a instrução `add`. Pode ser necessário fazer *zoom* na figura para ver os detalhes. Os nomes dos sinais de controlo são apresentados a vermelho, enquanto os nomes dos sinais de dados são apresentados a preto. Estes nomes são os nomes atuais utilizados nos módulos Verilog do SweRV EH1. Símbolos iguais (=) representam atribuições de sinais no código Verilog.

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

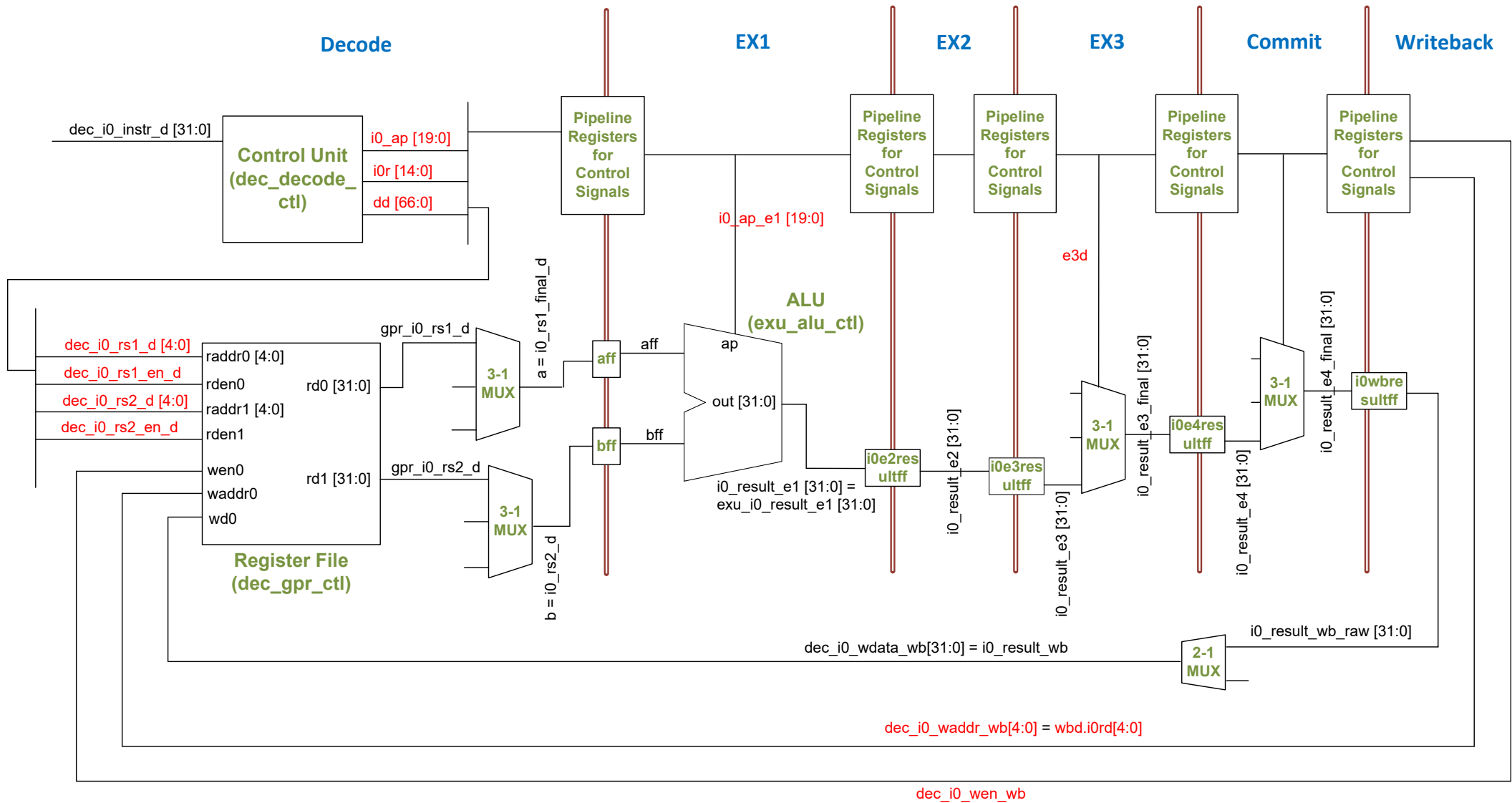


Figura 6. Principais unidades utilizadas pelas instruções aritmético-lógicas que passam pelo Pipe I0

i. Andar de Decode

Como explicado no Lab 11, o andar de descodificação é responsável por duas tarefas principais:

- **Descodificar as instruções e gerar sinais de controlo.**
- **Ler ou juntar os operandos de origem e enviar as instruções para os Pipes apropriados.**

Em seguida, analisamos cada uma destas tarefas para a instrução `add` e adicionamos alguns sinais relacionados com a simulação.

Descodificar as instruções e gerar sinais de controlo:

Tal como explicado na secção 2.C.i do Lab 11, estão definidas várias estruturas em `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` para agrupar os bits de controlo. Três destas estruturas estão diretamente relacionadas com as instruções Aritmética-Lógica (A-L):

- **`alu_pkt_t`:** Esta é a estrutura principal para as instruções A-L:

```
190 typedef struct packed {
191     logic valid;
192     logic land;
193     logic lor;
194     logic lxor;
195     logic sll;
196     logic srl;
197     logic sra;
198     logic beq;
199     logic bne;
200     logic blt;
201     logic bge;
202     logic add;
203     logic sub;
204     logic slt;
205     logic unsign;
206     logic jal;
207     logic predict_t;
208     logic predict_nt;
209     logic csr_write;
210     logic csr_imm;
211 } alu_pkt_t;
```

Dois sinais deste tipo, designados `i0_ap` (para o modo-0) e `i1_ap` (para o modo-1), são definidos e atribuídos no módulo **`dec_decode_ctl`** no andar de Decode, e propagados através dos andares de Execute subsequentes (EX1-4) (sinais `i0_ap_e1`, `i0_ap_e2`, `i0_ap_e3` e `i0_ap_e4` para o modo-0, e sinais `i1_ap_e1`, `i1_ap_e2`, `i1_ap_e3` e `i1_ap_e4` para o modo-1). Estes sinais contêm os sinais de controlo para informar a ALU da operação que deve realizar. Quando uma instrução `add` é executada, todos os bits de `i0_ap/i1_ap` são definidos como 0, exceto os seguintes:

- `valid`: indicando que se trata de uma instrução ALU válida
- `add`: indicando que se trata de uma instrução `add`

Quando a instrução na Via-0/1 não é uma instrução A-L, todos os bits do sinal `i0_ap/i1_ap` são 0 (e especificamente `valid = 0`), o que faz com que a ALU I0/I1 não funcione de todo.

- **reg_pkt_t**: Dois sinais deste tipo, denominados `i0r` (para Via-0) e `i1r` (para Via-1), são definidos, atribuídos e utilizados no módulo **dec_decode_ctl**. Contêm os números dos dois registos de origem (campos `rs1` e `rs2`) e do registo de destino (campo `rd`):

```
183 typedef struct packed {
184     logic [4:0] rs1;
185     logic [4:0] rs2;
186     logic [4:0] rd;
187 } reg_pkt_t;
```

- **dest_pkt_t**: Um sinal deste tipo, chamado `dd`, é definido e atribuído dentro do módulo **dec_decode_ctl** durante o andar de Decode. O sinal é propagado através de todos os restantes andares (sinais `e1d`, `e2d`, `e3d`, `e4d`, e `wbd`). Contém vários campos, como o registo de destino das instruções na Via-0 e Via-1: `i0rd[4:0]` e `i1rd[4:0]` respetivamente.

Alguns destes sinais são utilizados no andar de Decode e não são propagados através dos registos do circuito de controlo para andares posteriores. É o caso de `i0r.rs1/i1r.rs1` e `i0r.rs2/i1r.rs2` que são fornecidos diretamente ao Register File durante o andar de Decode para a leitura dos dois operandos de entrada (sinais `raddr0`, `raddr1`, `raddr2`, `raddr3`).

TAREFA: Encontre no código Verilog (módulo **dec_decode_ctl**) como é que o sinal de controlo `i0r` é utilizado para ler o Register File no andar de Decode.

No entanto, outros sinais de controlo devem ser propagados para andares posteriores. É o caso de `i0_ap/i1_ap`, que são utilizados pela ALU para saber a operação que deve efetuar (no nosso caso, uma adição), ou do sinal `dd`, que é utilizado, entre outras coisas, pelo Ficheiro de Registos para escrever os dois resultados.

TAREFA: Descubra no código Verilog (módulo **exu**) como é que os sinais de controlo `i0_ap` e `dd` são propagados do andar de Decode para o andar de Execute (EX1). Além disso, descubra como é que o sinal de controlo `dd` é utilizado pelo Register File durante o andar de Write-Back, depois de passar por todas as andares desde o Decode até ao Writeback.

Ler ou juntar os operandos de origem e enviar as instruções para os pipes apropriados:

Como explicado no Lab 11, o processador SweRV EH1 inclui vários Pipes para a execução das instruções. No andar de Decode, as instruções, uma vez decodificadas, devem ser programadas através do pipe apropriado. Especificamente, se uma instrução A-L estiver na Via-0, deve ser enviada, se possível, para o pipe I0; da mesma forma, se uma instrução A-L estiver na Via-1, deve ser enviada, se possível, para o pipe I1. No programa que estamos a analisar neste laboratório (Figura 2), uma vez que o processador decodificou na Via-0 a instrução `add` (ou seja, "sabe" que é uma instrução A-L e por isso deve enviá-la para o pipe I0), deve verificar se estão reunidas todas as condições para a execução através do pipe I0: Decodificação válida?, 2 operandos de entrada disponíveis?, Pipeline não bloqueado?... No nosso caso, o resultado desta verificação é enviado para o Pipe I0 através de dois sinais de estado que são computados no módulo **dec_decode_ctl** e que são usados pela ALU no

módulo **exu** (na próxima subsecção vamos explicar a ALU com mais detalhe). Estes dois sinais de estado são:

- `i0_e1_ctl_en` (renomeado `enable` dentro da ALU): Este sinal depende de `dec_i0_ctl_en[4:1]`, que estabelece, em tempo de decodificação, se cada um dos andares de execução (EX1-3) e o andar de Commit da Via-0 devem ser activadas (1) ou não (0). Note-se que a instrução pode ser ilegal, ou ter o pipeline bloqueado, ser descarregada, etc., devido a circunstâncias diferentes (predição errada de salto, cálculo de divisão, etc.), que desativariam o pipeline.
- `dec_i0_alu_decode_d` (renomeado como `válido` dentro da ALU): Este sinal é 1 se a instrução na Via-0 foi legalmente decodificada, é uma instrução Aritmética/Lógica e não utiliza a ALU Secundária (explicaremos esta estrutura no Laboratório 15).

Ambos os sinais têm de ser 1 para que a ALU efetue a operação `add` no andar seguinte (andar EX1).

TAREFA: A geração destes dois sinais (`i0_e1_ctl_en` e `dec_i0_alu_decode_d`) é um processo bastante complexo que não explicamos aqui em pormenor, mas que pode analisar mais profundamente nos módulos `dec_decode_ctl` e `exu`.

Como também explicado no Lab 11, os operandos de entrada são fornecidos ao Pipe I0 (`i0_rs1_final_d` e `i0_rs2_d`) através de dois multiplexers 3:1 implementados no andar de Decode (ver Figura 6). Na instrução `add` do nosso exemplo, ambos os operandos de entrada são obtidos diretamente do Register File:

- Primeiro operando de entrada: `i0_rs1_final_d[31:0] = gpr_i0_rs1_d[31:0]`
- Segundo operando de entrada: `i0_rs2_d[31:0] = gpr_i0_rs2_d[31:0]`

TAREFA: Localizar no código Verilog (módulo **exu**) o multiplexer 3:1 na parte inferior (segundo operando de entrada) e tentar encontrar a origem das suas entradas (na Figura 6 só é mostrada a entrada que vem do Register File). Não é necessário analisar as entradas com muita atenção, uma vez que estas serão analisadas nos exercícios propostos na Secção 3 e em laboratórios futuros.

A Figura 7 estende a simulação do Verilator da Figura 3 acrescentando os sinais introduzidos acima:

- `i0_ap[19:0]`
- `i0_ap.valid` (designado na figura como `i0_ap_valid` por meio de um aliás no script `.tcl` descrito abaixo e disponível em `[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl`)
- `i0_ap.add` (nomeado na figura como `i0_ap_add` por meio de um aliás no script `.tcl` descrito abaixo)
- `i0r[14:0]`
- `raddr0`
- `raddr1`
- `i0_e1_ctl_en` (renomeado como `enable` na ALU)
- `dec_i0_alu_decode_d` (renomeado como `válido` na ALU)
- `gpr_i0_rs1_d[31:0]`

- `gpr_i0_rs2_d[31:0]`

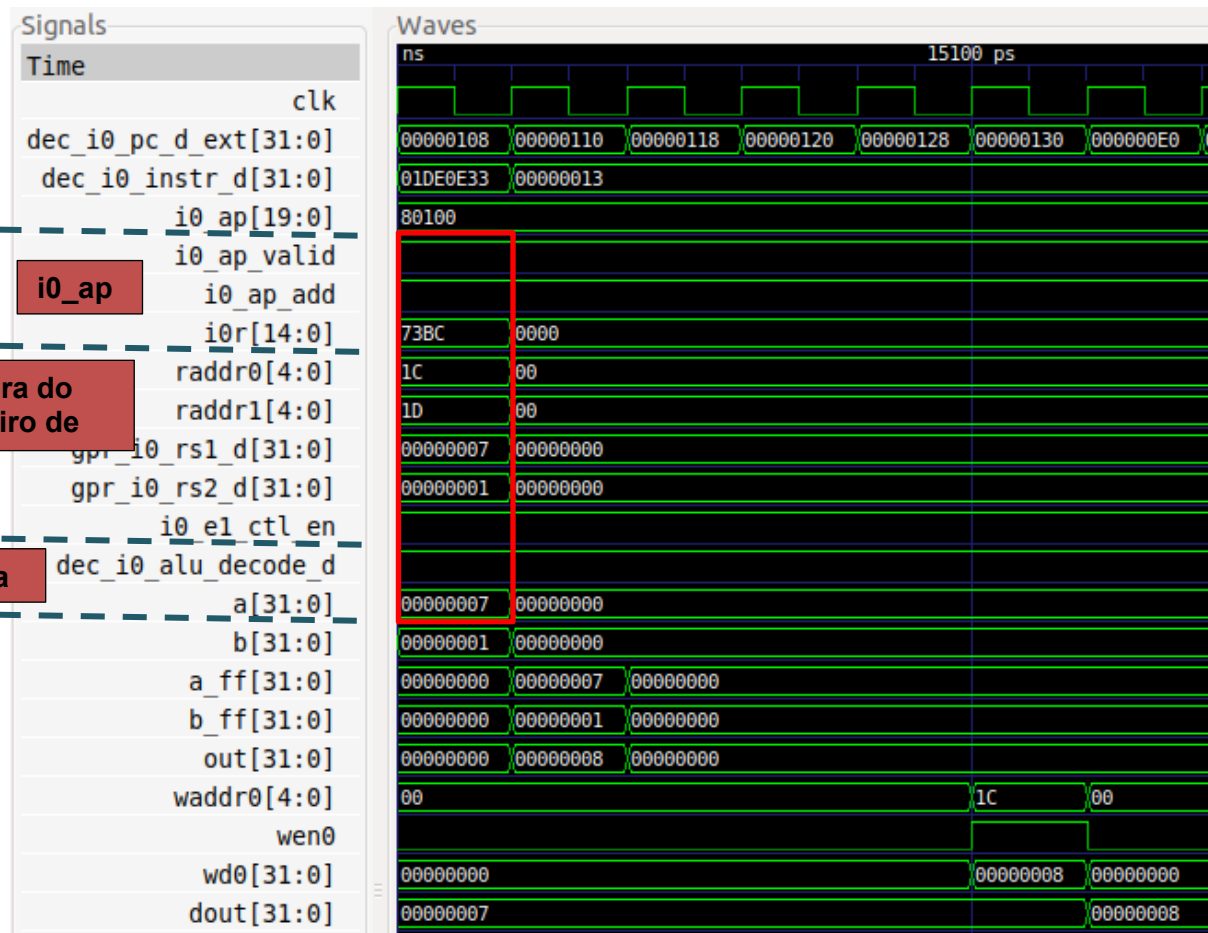


Figura 7. Simulação Verilog do programa de exemplo da Figura 2 incluindo sinais de controlo e portas de leitura do ficheiro de registo

TAREFA: Replicar a simulação da Figura 7 no seu próprio computador. Pode utilizar o script `.tcl` fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl`. Note que neste ficheiro `.tcl` são utilizados aliases para alguns dos bits de controlo.

Analisar a forma de onda da Figura 7. Como explicado anteriormente, todos os bits de `i0_ap` são 0, exceto os bits `valid` e de `add`. Como também explicado, o sinal `i0r` contém os identificadores dos dois registos de origem e um de destino da instrução de adição. Usando `i0r.rs1` e `i0r.rs2`, o Register File é acedido (ver sinais `raddr0` e `raddr1`) e os valores lidos são fornecidos ao Pipe I0: `gpr_i0_rs1_d = a = 0x7` e `gpr_i0_rs2_d = b = 0x1`. Finalmente, pode ver que ambos os sinais de validação e ativação são 1, pelo que a ALU do Pipe I0 será usada no próximo ciclo.

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 isto 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 esta instrução). Pode usar o mesmo ficheiro `test_2.tcl` usado no exemplo da Figura 7.

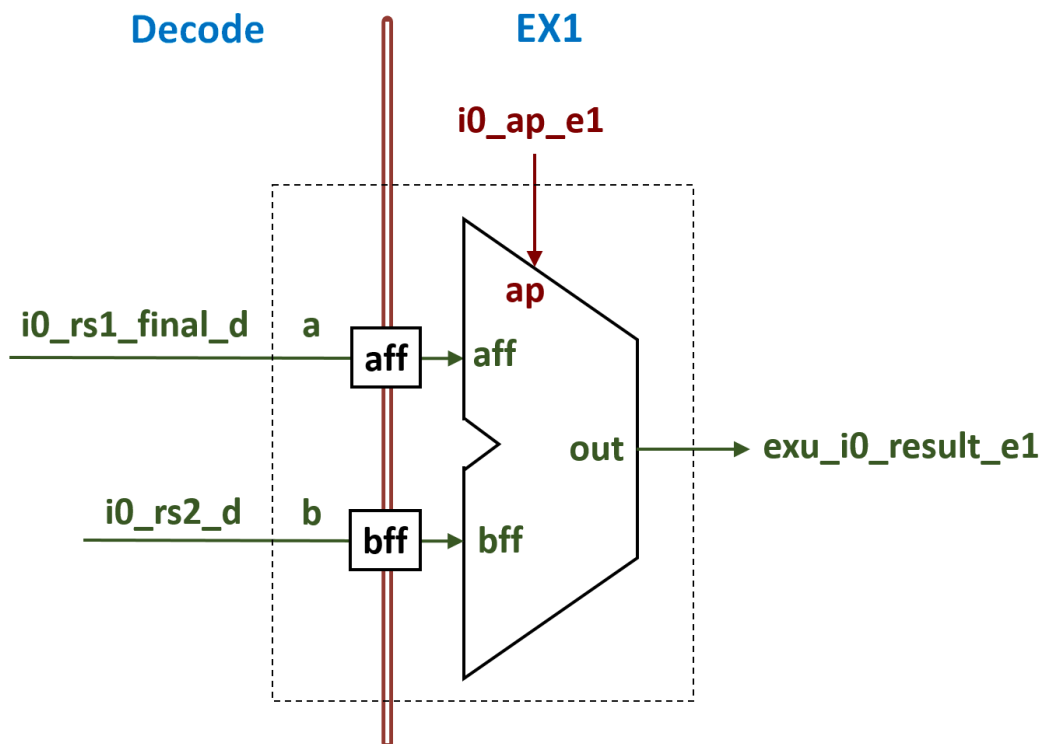
ii. Andar de Execute

Como explicado no Lab 11, o SweRV EH1 inclui quatro pipes de execução (ver Figura 4 no Lab 11): Pipes I0/I1, Multiplicação, e L/S. Para além disso, contém um Divisor não pipelined. Cada um dos pipes está dividido em 3 andares: **EX1/EX2/EX3** (Pipes I0/I1), **M1/M2/M3** (Pipe Multiply), **DC1/DC2/DC3** (L/S Pipe). Neste lab, vamos concentrar-nos no Pipe I0, onde a instrução `add` é executada. A principal tarefa do pipe I0 para uma instrução de `add` é computar a adição na ALU e propagá-la para o andar de Commit.

i. Andar EX1

Neste andar, é efetuada a operação da ALU - neste caso, uma adição. A unidade aritmético-lógica (ALU) do SweRV EH1 está implementada no módulo **exu_alu_ctl** (que pode ser encontrado em `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_alu_ctl.sv`), e é instanciado no módulo **exu** (que pode ser encontrado em `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu.sv`).

Figura 8 mostra a instanciação da ALU incluída no andar EX1 do Pipe I0, e um diagrama simplificado da ALU com alguns dos seus portos de entrada/saída. Note-se que a maioria dos sinais de entrada/saída são renomeados na ALU.



```

401     exu_alu_ctl i0_alu_e1 (.*,
402         .freeze      ( freeze                ), // I
403         .enable      ( i0_e1_ctl_en          ), // I
404         .predict_p   ( i0_predict_newp_d     ), // I
405         .valid       ( dec_i0_alu_decode_d   ), // I
406         .flush       ( exu_flush_final       ), // I
407         .a           ( i0_rs1_final_d[31:0]   ), // I
408         .b           ( i0_rs2_d[31:0]        ), // I
409         .pc          ( dec_i0_pc_d[31:1]     ), // I
410         .brimm       ( dec_i0_br_immed_d[12:1]), // I
411         .ap          ( i0_ap_e1              ), // I
412         .out         ( exu_i0_result_e1[31:0]), // 0
413         .flush_upper ( exu_i0_flush_upper_e1 ), // 0
414         .flush_path  ( exu_i0_flush_path_e1[31:1]), // 0
415         .predict_p_ff ( i0_predict_p_e1      ), // 0
416         .pc_ff       ( exu_i0_pc_e1[31:1]    ), // 0
417         .pred_correct ( i0_pred_correct_upper_e1 ), // 0
418     );

```

Figura 8. ALU de I0 (módulo exu_alu_ctl): Diagrama de alto-nível e código Verilog

Entradas da ALU: As entradas da ALU (a e b) são selecionadas no andar de Decode pelos dois multiplexers 3:1 apresentados na Figura 6 como explicado na secção anterior. No módulo **exu_alu_ctl**, dois registos (a_{ff} e b_{ff}) propagam os operandos do andar de Decode para o andar EX1 quando os sinais de **valid** e de **enable** são 1.

Sinais de controlo da ALU: A ALU é governada pelos bits de controlo gerados no andar de Decode no sinal **i0_ap** (lembre-se que esta é uma estrutura do tipo **alu_pkt_t**). Este sinal é propagado através dos Registos do Pipeline como explicado na secção anterior. Em EX1, este sinal chama-se **i0_ap_e1** e é renomeado como **ap** dentro da ALU (ver Figura 8).

Saída da ALU: A saída da ALU é obtida no sinal **exu_i0_result_e1** (ver Figura 8). Este sinal é propagado para EX2 utilizando um novo registo de Pipeline (ver Figura 6), que pode ser encontrado no módulo **dec_decode_ctl** (o sinal é primeiro atribuído a **i0_result_e1**):

```

2256     assign i0_result_e1[31:0] = exu_i0_result_e1[31:0];
2260     rvdffe #(32) i0e2resultff (.*, .en(i0_e2_data_en), .din(i0_result_e1[31:0]), .dout(i0_result_e2[31:0]));

```

TAREFA: Incluir os novos sinais analisados nesta secção na simulação da Figura 7.

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

TAREFA: Analisar a implementação Verilog do somador/subtrator implementado no módulo **exu_alu_ctl**. Figura 9 dá-lhe alguma ajuda, mostrando a lógica diretamente relacionada com as operações de adição e subtração. Pode utilizar uma simulação do Verilator como ajuda.

```

90     rvdffe #(32) a_ff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92     rvdffe #(32) b_ff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));

```

```

135     assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138     assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
139
172     assign sel_adder = (ap.add | ap.sub) & ~ap.slt;

185     assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                       ({32{sel_shift}} & sout[31:0]) |
187                       ({32{sel_adder}} & aout[31:0]) |
188                       ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                       ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                       ({31'b0, slt_one});

```

Figura 9 . Somador dentro de exu_alu_ctl

ii. Andares EX2 e EX3

Estes andares realizam poucas tarefas nas instruções aritméticas-lógicas; no entanto, são necessários para sincronizar estas instruções com outros tipos de instruções (como leituras, escritas, multiplicações, etc.) que requerem três ciclos para computar as suas operações. Recorde-se que, numa conceção multi-ciclo, cada instrução pode ter um número diferente de estágios, mas num processador com pipelines de ordem, como o SweRV EH1, todas as instruções têm de percorrer o mesmo número de andares.

No nosso exemplo, o resultado da adição é propagado através de novos registos de pipeline, que podem ser encontrados no módulo **dec_decode_ctl**. No multiplexer 3:1 de EX3, a entrada `i0_result_e3` é selecionada (ver Figura 6). Este multiplexer 3:1 também foi mostrado na Figura 4 e na Figura 8 do Laboratório 11. Como explicado nesse laboratório, ele seleciona o resultado do Pipe adequado, que no nosso exemplo da Figura 2 é o resultado fornecido pelo Pipe I0: (`i0_resultado_e3_final = i0_resultado_e3`).

```

2263     rvdffe #(32) i0e3resultff (., .en(i0_e3_data_en), .din(i0_result_e2[31:0]), .dout(i0_result_e3[31:0]));
2274     rvdffe #(32) i0e4resultff (., .en(i0_e4_data_en), .din(i0_result_e3_final[31:0]), .dout(i0_result_e4[31:0]));

```

TAREFA: Verificar na simulação que este multiplexer seleciona o resultado do pipe esperado para a instrução `add`, para o exemplo da Figura 2.

iii. Andar de Commit

À semelhança do que acontece com EX2 e EX3, este andar faz pouco para uma instrução `add` independente (no Lab 15 analisaremos um cenário em que uma instrução `add` que depende de uma instrução anterior tem de recalculer a adição na ALU Secundária, não mostrada na Figura 6). Neste exemplo, a entrada `i0_resultado_e4` é selecionada pelo multiplexer 3:1 disponível neste andar. Este multiplexer 3:1 também foi mostrado na Figura 4 e na Figura 9 do Laboratório 11. No nosso exemplo da Figura 2o valor selecionado é novamente o resultado fornecido pelo Pipe I0 (`i0_result_e4_final = i0_result_e4`).

TAREFA: Verificar na simulação que este multiplexer seleciona o resultado da fonte de entrada correta (`i0_result_e4`) para a instrução `add` do nosso exemplo da Figura 2.

iv. Andar de Writeback

No último andar, o resultado da instrução `add` é escrito no Register File, como se mostra na Figura 6. O resultado de 32 bits (`i0_resultado_wb_raw[31:0]`) foi computado no andar EX1 e foi propagado para este andar. Passa por um multiplexer 2:1 antes de ser passado para o Register File (a outra entrada deste multiplexer vem do Divisor, como analisaremos no Lab 14). O endereço do registo (na Figura 7 o sinal `waddr0` é mostrado em hexadecimal, mas poderia ser mostrado em decimal, como explicado anteriormente) e os sinais de permissão de escrita são fornecidos através dos Registos do Pipeline de Controlo.

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

3. EXERCÍCIOS

- 1) Efetue uma análise semelhante à apresentada neste laboratório para as instruções lógicas: `and`, `or` e `xor`.
- 2) (O exercício seguinte baseia-se no exercício 4.1 do livro "*Computer Organization and Design - RISC-V Edition*", de Patterson & Hennessy ([PaHe])).
Considere a seguinte instrução: `e rd, rs1, rs2`
 - a. Quais são os valores dos sinais de controlo gerados pelo SweRV EH1 para esta instrução?
 - b. Que recursos (blocos) desempenham uma função útil para esta instrução?
 - c. Que recursos (blocos) não produzem qualquer saída para esta instrução? Que recursos produzem uma saída que não é utilizada?
- 3) Analisar, numa simulação Verilator e diretamente no código Verilog, as instruções *shift left/right* disponíveis no conjunto de instruções RV32I Base Integer: `srl`, `sra` e `sll`.
- 4) Analisar, tanto numa simulação Verilator como diretamente no código Verilog, o conjunto de instruções *less than* disponíveis no RV32I Base Integer Instruction Set: `slt` e `sltu`.
- 5) Analisar, tanto numa simulação Verilator como diretamente no código Verilog, algumas das instruções *imediatas* disponíveis no Conjunto de Instruções de Base Inteira RV32I: `addi`, `andi`, `ori`, `xori`, `srl`, `srai`, `slli`, `slti` e `sltui`.
- 6) (O exercício seguinte é baseado no exercício 4.6 de [PaHe]).
Figura 6 não aborda instruções do tipo I, como `addi` ou `andi`.
 - a. Que blocos lógicos adicionais, se existirem, são necessários para suportar a execução de instruções de tipo I no SweRV EH1? Acrescentar os blocos lógicos necessários para Figura 6 e explique o seu objetivo.
 - b. Enumerar os valores dos sinais gerados pela unidade de controlo para aditivar.
- 7) (O exercício seguinte baseia-se no exercício 4.4 de [PaHe] 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 fabrico podem resultar em circuitos defeituosos. Um defeito muito comum é um fio de sinal ficar "partido" e registar sempre um 0 lógico. Determine o efeito de cada um dos bits de controlo incluídos no sinal `i0_ap` (um sinal do tipo `alu_pkt_t`) ficar preso no 0.