



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 11

Configuração, Organização e Monitorização do Desempenho do SweRV EH1

1. INTRODUÇÃO

Nos primeiros 10 laboratórios RVfpga (Laboratórios 1-10), apresentámos a arquitetura RISC-V e como comunicar o SweRV EH1 Core utilizando vários periféricos. Nos próximos dez laboratórios (Laboratórios 11-20), vamos descer ao nível da microarquitetura e analisar como o processador SweRV EH1 funciona internamente e como funciona a hierarquia da cache/memória.

SIGASI STUDIO: Nestes laboratórios vamos lidar com um projeto Verilog extenso: o SweRV EH1 Core RTL. Uma forma de analisar os vários módulos e sinais é utilizar um editor de texto como o Sublime Text (<https://www.sublimetext.com/>), que oferece funcionalidades interessantes para navegar num projeto, inspecionar os ficheiros, procurar strings, etc. No entanto, existem alternativas mais adequadas e específicas, como o **Sigasi Studio** (<https://www.sigasi.com/>), que recomendamos vivamente. Um documento suplementar, **RVfpga_SweRVref.docx**, mostra, entre outras coisas, como instalar e utilizar o Sigasi Studio (Secção 1 do documento RVfpga_SweRVref).

Tal como explicado no Guia de Iniciação do RVfpga (GSG), o SweRV EH1 é um processador superescalar de 32 bits, de 2 vias, com 9 andares de pipeline em ordem. A Figura 1 mostra uma visão de alto nível da microarquitetura SweRV EH1. A SweRV EH1 suporta as extensões RISC-V de inteiros (I), instruções comprimidas (C) e multiplicação e divisão de inteiros (M). O seu desempenho impressionantemente elevado por MHz (4,9 CM/MHz) é conseguido graças à inclusão de várias técnicas microarquitetónicas, desde as mais básicas e comuns, como o pipelining e uma cache de instruções, a outras técnicas mais específicas e avançadas, como a execução superescalar, leituras e divisões não bloqueantes, duas ALUs secundárias que permitem a repetição de instruções aritmético-lógicas quando necessário devido a conflitos de dados (ver Laboratório 15 para mais pormenores), leituras e escritas desalinhadas, memórias "scratch pad" para instruções e dados e previsão avançada de ramificações. Todas estas técnicas serão amplamente analisadas nestes laboratórios.

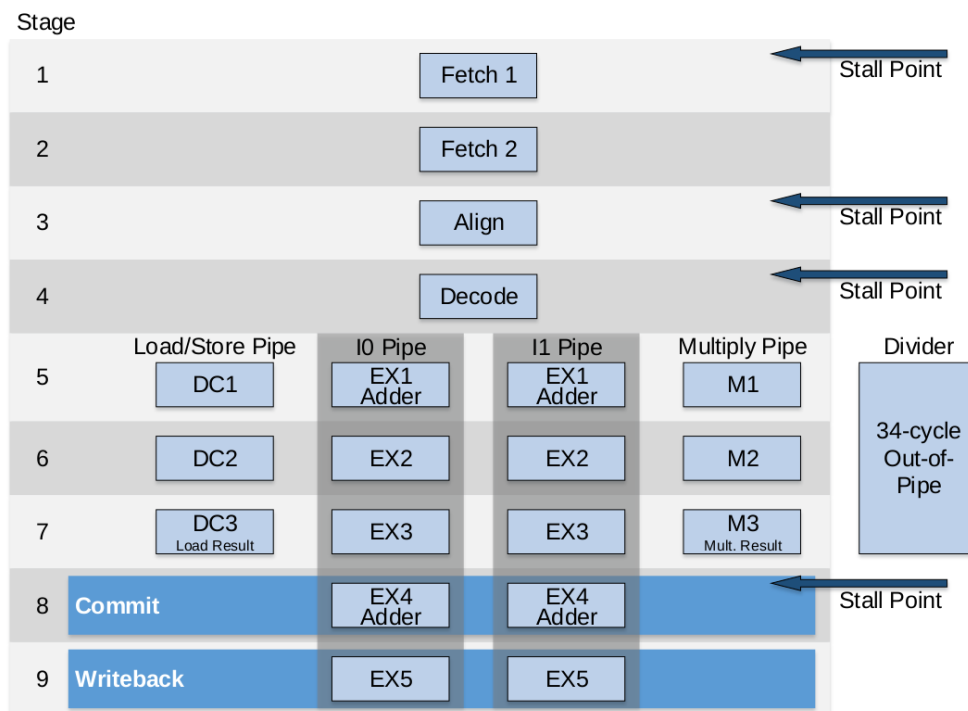


Figura 1. Microarquitetura do core SweRV EH1

(Figura de https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)

NOTA: Antes de iniciar este conjunto de laboratórios, recomendamos que leia atentamente os capítulos 7 e 8 do livro *Digital Design and Computer Architecture: RISC-V Edition* de S. Harris e D. Harris (Morgan Kaufmann © 2021). Alguns dos conteúdos destes laboratórios são inspirados nesse livro. Vamos nos referir ao livro como DDCARV.

A maioria dos laboratórios está dividida em duas partes: uma secção de fundamentos seguida de uma secção avançada. Além disso, dada a elevada complexidade de algumas partes de um processador real, como o SweRV EH1, alguns detalhes são transferidos para o apêndice de um determinado laboratório. Desta forma, os utilizadores podem optar por completar apenas a secção fundamental, completar as secções fundamental e avançada, ou mesmo aprofundar os apêndices e compreender as partes mais complexas do processador.

Os laboratórios 11-20 começam com uma explicação teórica dos conceitos e depois ilustram os conceitos usando Figuras e uma simulação do Verilator de um programa de exemplo. Estes são programas de exemplo que têm apenas o objetivo de ilustrar o conceito. Também fornecemos exercícios para aprofundar a compreensão e a experiência com os conceitos descritos.

É possível completar apenas um subconjunto dos laboratórios, dependendo do objetivo e da profundidade do curso. Os conceitos de pipelining, organização de memória e microarquitetura avançada/hierarquia de memória são abordados nos seguintes laboratórios:

- **Pipelining:** Labs 11, 12, 14, 15 e primeira parte do 16 (instruções de salto)
- **Memória:** Labs 11, 13 e 19
- **Microarquitetura avançada e hierarquia de memória:** Labs 17, 18, 20 e a segunda parte do 16 (preditor de saltos)

Neste laboratório (Laboratório 11), começamos a analisar o processador SweRV EH1. Especificamente:

- **Secção 2** descreve a organização Verilog RTL e os pormenores de cada andar do pipeline.
- **Secção 3** mostra como utilizar os contadores de desempenho para analisar o desempenho do processador.

O documento suplementar (**RVfpga_SweRVref.docx**) descreve:

- **Secção 1:** Uso do Sigasi Studio.
- **Secção 2:** Configuração do processador SweRV EH1.
- **Secção 3:** Sistema Hierarquia de módulos do sistema RVfpga e seus sinais mais relevantes
- **Secção 4:** Estruturas e tipos para agrupar bits de controlo
- **Secção 5:** instruções comprimidas RISC-V
- **Secção 6:** Avaliações de desempenho (Benchmarks) reais

Após esta abordagem inicial, alargamos esta análise nos Labs 12-20 a várias unidades do processador. Especificamente:

- **Lab 12** centra-se em instruções **aritmética-lógica** aprofundando os andares de Descodificação, EX1/EX2/EX3 e Writeback.
- **Lab 13** descreve **instruções de memória** (leituras e escritas) centrando-se nos andares DC1/DC2/DC3.
- **Lab 14** discute **conflitos estruturais** centrando-se na instrução de multiplicação em 3 ciclos pipeline e num caso específico relacionado com leituras não bloqueantes. O laboratório também analisa a instrução de divisão não pipelined de 34 ciclos num apêndice.
- **Lab 15** analisa **conflitos de dados** descrevendo os caminhos de *bypass* do processador.
- **Lab 16** descreve **conflitos de controlo**, instruções de salto e o preditor de saltos, para o qual nos centraremos nas fases Fetch 1 e Fetch 2 do processador SweRV EH1.
- Enquanto nos laboratórios anteriores apenas uma via do processador é usada na maioria dos casos, o **Lab 17** descreve processadores superescalares de 2 vias, como o SweRV EH1.
- **Lab 18** é um laboratório prático onde se adicionam novas instruções e contadores de hardware ao núcleo do SweRV EH1.
- **Labs 19 e 20** centram-se nas várias memórias de baixa latência disponíveis no processador: a cache de instruções (I\$) e as memórias de instruções e de dados estreitamente acopladas (ICCM e DCCM).

2. UMA APROXIMAÇÃO INICIAL À MICROARQUITECTURA SweRV EH1

O processador descrito no DDCARV tem 5 andares de pipeline, que são chamados de *Fetch*, *Decode*, *Execute*, *Memory* e *Writeback*. Em contrapartida, o pipeline SweRV EH1

está dividido em 9 andares (Figura 1): *Fetch1*, *Fetch2*, *Align*, *Decode*, *EX1/DC1/M1*, *EX2/DC2/M2*, *EX3/DC3/M3*, *Commit*, e *Writeback*. Ao comparar os dois processadores, alguns estágios são equivalentes, como os andares de *Decode* e *Writeback*. Mas o SweRV EH1 acrescenta caminhos paralelos (pipelines de leitura/escrita vs. inteiros vs. multiplicação), divide alguns andares em vários andares (o *Fetch* tem 2 andares e o *Execute* tem 3 andares) e acrescenta andares (os andares *Commit* e *Align*).

O resto desta secção descreve a organização Verilog RTL e os detalhes de cada fase do pipeline. A Secção A descreve a hierarquia dos módulos Verilog do SweRV EH1. As Secções B e C discutem a microarquitetura do SweRV EH1 andar a andar. Finalmente, a Secção D apresenta um exemplo prático das explicações teóricas dadas nas Secções B e C.

CONFIGURAÇÃO DO PROCESSADOR SWERV EH1: Muitas das estruturas e características do processador SweRV EH1 podem ser configuradas ou ativadas/desativadas. O documento suplementar, *RVfpga_SweRVref.docx*, explica as diferentes opções na Secção 2, que irá utilizar frequentemente nos Laboratórios 12-20.

A. Hierarquia dos Módulos Verilog do SweRV EH1

A Figura 2 mostra a hierarquia dos principais módulos Verilog (alguns módulos não estão incluídos na figura) que compõem o processador SweRV EH1. Esta Figura expande a Figura 29 do GSG, onde mostramos a hierarquia dos módulos Verilog que compõem o Sistema RVfpga. Estes módulos estão localizados nos ficheiros com o mesmo nome no diretório: *[RVfpgaPath]/RVfpga/src/SweRVofSoC/SweRVEh1CoreComplex*.

O módulo **mem** instancia as estruturas que constituem a hierarquia de memória do processador SweRV EH1: ICCM, DCCM e I\$. O módulo **swerv** é o CPU global; instancia os módulos que compõem o processador SweRV EH1: Instruction Fetch Unit (**ifu**), Decode Unit (**dec**), Execution Unit (**exu**), Load/Store Unit (**lsu**)...

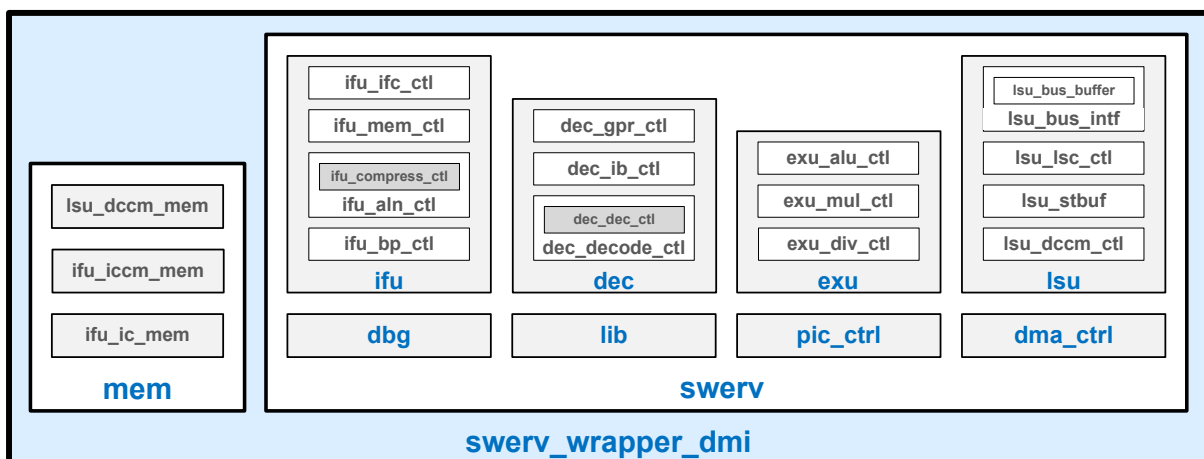


Figura 2. Módulos principais do SweRV EH1

PRINCIPAIS SINAIS DO NÚCLEO SweRV EH1: O documento suplementar, *RVfpga_SweRVref.docx*, fornece, na Secção 3, os principais sinais de entrada/saída de/para os módulos do processador SweRV EH1. Pode utilizá-lo como referência para completar os Labs 11-20.

B. Andares Fetch (FC1 e FC2) e Align

Nesta secção analisamos os três primeiros andares do pipeline: os dois andares Fetch (FC1 e FC2) e o andar Align do pipeline do SweRV EH1. A Figura 3 ilustra uma visão muito simplificada destes andares.

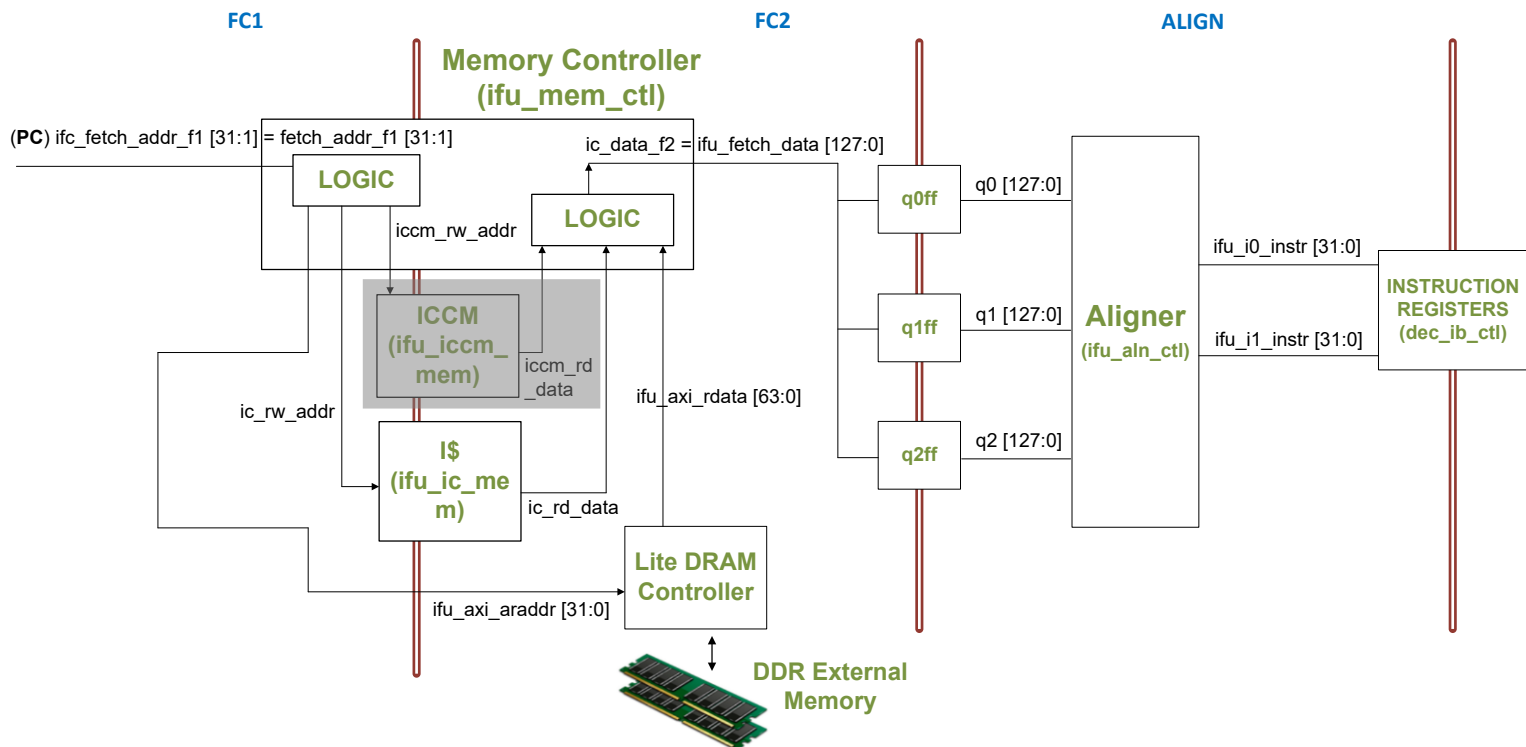


Figura 3. Vista simplificada dos andares FC1, FC2 e Align. Note que o ICCM está sombreado, indicando que está desativado no nosso sistema RVfpga.

i. Andares Fetch (FC1 e FC2)

Em cada ciclo, o andar Fetch é responsável por **ler as instruções da Memória de Instruções**. Na nossa configuração, a memória de instruções é composta por um ICCM (implementado no módulo `ifu_iccm_mem`), uma Cache de Instruções (I\$, implementada no módulo `ifu_ic_mem`) e a memória externa DDR. Tanto o I\$ como o ICCM são controlados a partir de um controlador de memória unificado (`ifu_mem_ctl`), enquanto a Memória Externa é controlada a partir do Controlador Lite DRAM. No nosso sistema RVfpga por omissão, o ICCM está desativado, mas pode facilmente incluí-lo como explicado no Lab 20.

Como se pode ver na Figura 3, o endereço da instrução (chamado endereço de fetch, `ifc_fetch_addr_f1`) é calculado no primeiro andar de Fetch (**FC1**) como será discutido mais adiante no Lab 16. Este endereço é fornecido ao controlador da memória de instruções (implementado no módulo `ifu_mem_ctl`): `fetch_addr_f1 = ifu_fetch_addr_f1`.

Os sinais têm normalmente um prefixo correspondente à unidade de que fazem parte. Por exemplo, “ifu” representa Instruction Fetch Unit. Os sinais acrescentam o andar a que estão associados. Por exemplo, “f1” indica o andar FC1.

A instrução é lida durante o segundo andar de Fetch (**FC2**) da Memória Principal (ou seja, Memória Externa DDR) ou do ICCM. Se o endereço da instrução estiver dentro do intervalo de endereços da memória principal, o I\$ fornece a instrução. Se o endereço da instrução

estiver dentro do intervalo de endereços da memória principal, o I\$ fornece a instrução. Se o endereço da instrução estiver dentro do intervalo de endereços da memória principal, o I\$ fornece a instrução. Se o endereço da instrução estiver dentro do intervalo de endereços ICCM, a instrução é fornecida com baixa latência a partir do ICCM através de um multiplexer implementado dentro do módulo **ifu_ic_mem**.

A memória de instruções do sistema RVfpga está configurada da seguinte forma (esta configuração pode ser modificada, como mostraremos em laboratórios futuros):

- 16 KiB de Cache de instruções
- 512 KiB ICCM (desativado): gama de endereços: 0xEE000000 – 0xEE07FFFF
- 128 MiB Memória externa: gama de endereços: 0x00000000 – 0x07FFFFFF

Se o programa não tiver *stalls* (ou seja, não há conflitos de controlo, de dados ou estruturais, não há I\$ misses, etc.), são lidas quatro instruções de 32 bits (128 bits no total) de dois em dois ciclos: ver sinal `ifu_fetch_data[127:0]`. Isto é suficiente para manter o pipeline superescalar de 2 vias a funcionar com o seu débito máximo de 2 instruções por ciclo. Três buffers (`q0ff`, `q1ff` e `q2ff`) pode armazenar até três destes pacotes de 128 bits.

ii. Andar Align

O andar Align, que se segue às duas fases Fetch (ver Figura 3), é implementado no módulo **ifu_aln_ctl**. O andar Align é responsável pela execução de duas tarefas principais:

- **Fornecer duas instruções de 32 bits por ciclo ao andar de decodificação:** O andar Align extrai duas instruções por ciclo dos pacotes de 128 bits fornecidos pela Memória de Instruções e que são temporariamente armazenados em buffers `q0ff`, `q1ff` e `q2ff`. Estas duas instruções são atribuídas a cada uma das duas vias disponíveis no SweRV EH1 através de sinais `ifu_i0_instr[31:0]` (Via 0) e `ifu_i1_instr[31:0]` (Via 1), e são depois armazenadas nos dois Instruction Registers (IR) implementados no módulo **dec_ib_ctl**.
- **Descomprimir instruções:** A extensão de instrução comprimida (RVC) do RISC-V reduz o tamanho das instruções inteiras e de vírgula flutuante comuns para 16 bits, reduzindo os tamanhos dos campos de controlo, imediato e de registo e tirando partido de registos redundantes ou implícitos. Este tamanho reduzido de instrução diminui o custo, a potência e a memória necessária (ver Secção 6.6.5 do DDCARV). O andar Align descomprime estas instruções de 16 bits, quando necessário, antes de as passar para o andar Decode, que apenas decodifica instruções de 32 bits. Isto é efetuado pelo módulo **ifu_compress_ctl**, que é instanciado dentro do Aligner (módulo **ifu_aln_ctl**).

INSTRUÇÕES COMPRIMIDAS: O documento suplementar, RVfpga_SweRVref.docx, explica, na Secção 5, a execução de instruções comprimidas no SweRV EH1 e propõe algumas novas tarefas.

C. Andares Decode, Execution, Commit e Writeback

Nesta secção analisamos os andares de "Decode", "Execution", "Commit" e "Writeback" do pipeline SweRV EH1. A Figura 4 ilustra uma visão simplificada destes andares, que alargaremos em futuros laboratórios.

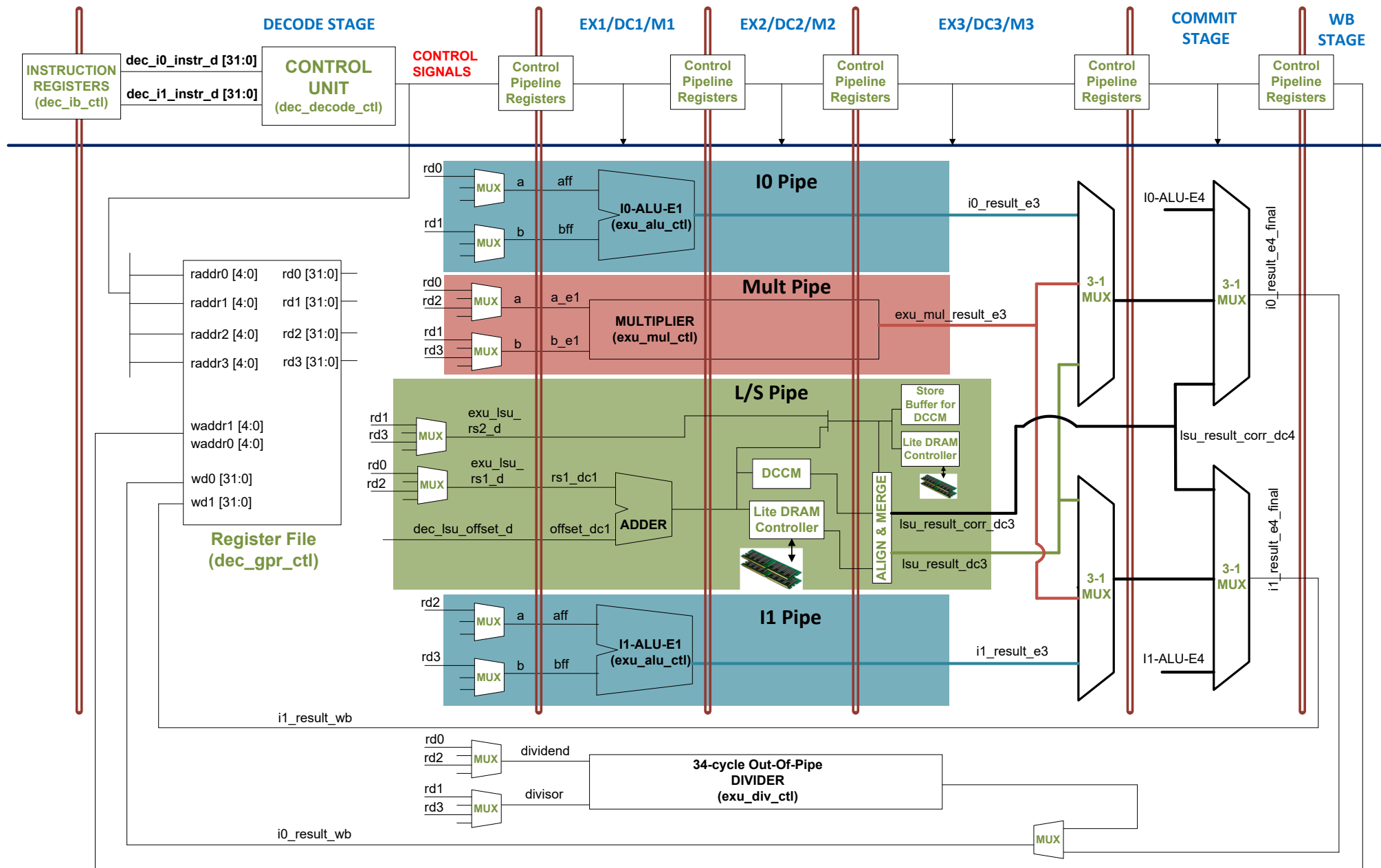


Figura 4. Visão simplificada dos andares de Decode, Execution, Commit e WB

i. Andar Decode

Os módulos Verilog para este andar estão na pasta `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec`. Em cada ciclo, o andar de descodificação é responsável por duas tarefas principais:

- **Descodificar as instruções e gerar os sinais de controlo:** Os sinais de controlo estão organizados em vários tipos, tal como definido no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`. Cada estrutura/tipo está relacionada com uma determinada unidade: ALU (`alu_pkt_t`), Unidade de multiplicação (`mul_pkt_t`), Unidade de Divisão (`div_pkt_t`), Registos (`reg_pkt_t`), etc.

ESTRUTURAS UTILIZADAS PARA OS BITS DE CONTROLO: O documento suplementar, `RVfpga_SweRVref.docx`, estende, na Secção 4, a descrição das principais estruturas/tipos usados no processador SweRV EH1 para agrupar os sinais de controlo e propõe algumas novas tarefas. Em laboratórios posteriores, concentrar-nos-emos nos tipos relacionados com a unidade discutida.

A Unidade de Controlo (Control Unit), implementada no módulo **dec_decode_ctl**, recebe as duas instruções de 32 bits obtidas, descomprimidas, alinhadas e atribuídas a cada via nos andares anteriores (sinais `dec_i0_instr_d[31:0]` para a Via 0 e `dec_i1_instr_d[31:0]` para a Via 1) e descodifica-os, gerando os sinais de controlo para cada instrução. A Figura 5 mostra uma visão de alto-nível da Control Unit (módulo **dec_decode_ctl**), que gera sinais de controlo nos dois andares: Os dois primeiros módulos (**i0_dec** e **i1_dec**) usam as instruções (**i0** e **i1**) para produzir sinais de controlo globais (**i0_dp** e **i1_dp**, ambos do tipo `dec_pkt_t`), e depois a segunda unidade (**decode**) utiliza esses sinais para gerar sinais de controlo para cada segmento do pipeline, também designados por “pipes” (**i0_ap**, **i1_ap**, **lsu_p**, **mul_p**, etc.).

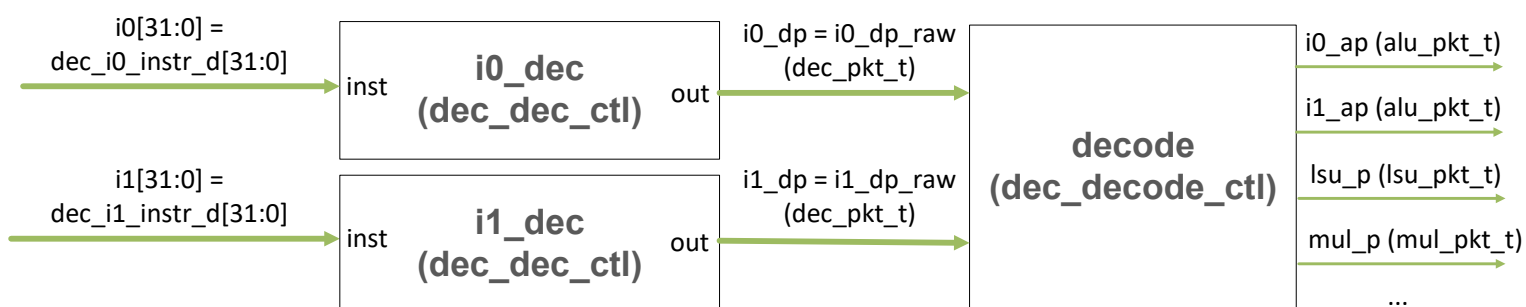


Figura 5. Unidade de Controlo

A Unidade de Controlo propaga estes sinais de controlo para os andares posteriores do pipeline utilizando registos do pipeline (identificados como **Control Pipeline Registers** na Figura 4), que são colocados entre cada andar do pipeline.

- **Distribuir as instruções pelos pipes apropriados e fornecer os operandos:** Como se pode ver na Figura 4, o SweRV EH1 inclui dois pipes de inteiros (I0 e I1), um pipe Multiply e um pipe de Load/Store (L/S). Além disso, inclui um divisor de 34 ciclos que se encontra fora do pipeline. Quando cada instrução é descodificada, o processador envia-a para um de quatro pipelines separados:

- As instruções aritméticas-lógicas e de salto são executadas nos pipes I0/I1.
- As leituras e escritas são executados nos pipes L/S.
- As instruções de multiplicação são executadas através do pipe Multiply.
- Divide as instruções executadas através do pipe Divider.

Dado que são decodificadas até duas instruções em cada ciclo, uma na Via 0 e outra na Via 1, ambas são programadas para execução sempre que possível. Por exemplo, algumas combinações possíveis são:

- Duas instruções Aritmética-Lógica independentes são enviadas para os pipes I0 e I1.
- Uma instrução Aritmética-Lógica e uma instrução de multiplicação (`mul`) são enviados para os pipes I0 (ou I1) e Multiply, respetivamente.
- Uma instrução de memória (Leitura ou Escrita) é executada no pipe L/S, e uma instrução de multiplicação é executada no pipe Multiply.

Infelizmente, existem algumas situações (como os conflitos, que analisamos nos Laboratórios 14-16) em que uma ou as duas instruções têm de ser interrompidas. Estas situações são também determinadas no andar da Descodificação. Por exemplo:

- Se duas instruções `mul` são decodificados no mesmo ciclo, o conflito estrutural é resolvido atrasando a segunda instrução `mul` num ciclo (isso será analisado em detalhes no Lab 14).
- Se duas instruções Aritmética-Lógica (A-L) dependentes forem decodificadas no mesmo ciclo, o conflito de dados RAW é resolvido atrasando a segunda instrução A-L em um ciclo (isto será analisado mais detalhadamente no Lab 15).

Para além de programar as instruções, os pipes devem ser dotados dos operandos correspondentes. Para esse efeito, vários multiplexers 3:1 e 4:1 (ver Figura 4) selecionam entre os operandos possíveis e propagam-nos para os andares seguintes utilizando registos de pipeline. Esses multiplexers são implementados nas linhas 279-328 do módulo **exu** (embora os multiplexers estejam dentro do módulo **exu**, eles operam no andar de decodificação). Os seus operandos de entrada podem vir de vários sítios:

- **Bypass Logic:** A maioria das dependências de dados são resolvidas no andar de decodificação através de bypasses, como analisaremos no Laboratório 15. As entradas provenientes da lógica de bypass não são identificadas nos multiplexers 3:1 e 4:1 da Figura 4 por uma questão de simplicidade - apenas são apresentados fios em branco.
- **Immediate:** Algumas instruções RISC-V utilizam o modo de endereçamento imediato, no qual o operando é fornecido diretamente a partir dos bits da instrução. As entradas provenientes do imediato não são mostradas nos multiplexers 3:1 e 4:1 da Figura 4 – apenas é apresentado um fio de entrada em branco).
- **Register File:** O Register File disponível no processador SweRV EH1 (Figura 6) tem 4 portos de leitura e 3 portos de escrita (note-se que o terceiro porto de escrita é ignorado no ficheiro de registo incluído na Figura 4 pois só é utilizada em situações específicas que analisaremos em laboratórios futuros). Estes portos de leitura/escrita permitem a execução de duas instruções por ciclo. As entradas provenientes do Register File são mostradas nos multiplexers 3:1 e 4:1

de Figura 4 utilizando apenas os nomes dos sinais. As ligações com o "Register File" não são mostradas por uma questão de simplicidade.

Cada porto de leitura/escrita tem um endereço de 5 bits ($raddr0 \dots raddr3$, $waddr0 \dots waddr2$), e um sinal de ativação de 1 bit ($rden0 \dots rden3$, $wen0 \dots wen2$) não apresentado na Figura 4. Os portos de escrita também têm uma entrada de dados de escrita de 32 bits ($wd0 \dots wd2$), e os portos de leitura têm uma saída de dados de leitura de 32 bits ($rd0 \dots rd3$). O Register File contém 32 registos de 32 bits, designados $x0-x31$, com $x0$ ligado a 0.



Figura 6. Register File disponível no SweRV EH1

TAREFA: O Register File é implementado no módulo **dec_gpr_ctl** e é instanciado no módulo **dec** (ver Figura 7). Analisar o código Verilog e a simulação dos principais sinais do módulo **dec_gpr_ctl** (disponível no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv`), para compreender o seu funcionamento. Note-se que o processador SweRV EH1 permite a inclusão de vários Register Files, mas a configuração utilizada no Sistema RVfpga apenas utiliza um Register File (ver linha 402 do ficheiro `dec.sv`: `localparam GPR_BANKS = 1;`).

```

525 dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526             .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.,
527             // inputs
528             .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529             .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530             .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531             .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533             .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534             .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535             .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537             // outputs
538             .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539             .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540             );

```

Figura 7. Instanciação do Register File dentro do módulo dec

ii. Andares de Execução

Nesta subsecção, analisamos versões simplificadas dos pipes disponíveis no SweRV EH1: dois **Pipes de Inteiros (I0 Pipe e I1 Pipe)**, um **Pipe Multiplicação**, um **Pipe Leitura/Escrita**, e um **Divisor** de 34 ciclos não pipelined.

Pipes I0/I1: Os dois pipes de inteiros são mostrados a azul na Figura 4. Estão divididos em três andares denominados EX1, EX2 e EX3. Cada um destes dois pipes inclui uma ALU de latência de 1 ciclo em **EX1**, que é capaz de efetuar operações aritméticas como a adição ou a subtração, bem como operações lógicas como *and* e *or*. Os andares EX2 e EX3 executam poucas tarefas mas são necessários para sincronizar as instruções A-L com os outros tipos de instruções (como loads, stores, multiplicações, etc.) que requerem três ciclos para executar as suas operações. No Lab 12, analisaremos os pipes I0/I1 em mais pormenor.

Pipe de Multiplicação: O pipe multiply é mostrado a vermelho na Figura 4. Divide-se em três andares: **M1**, **M2**, e **M3**. Este pipe inclui um multiplicador de 3 ciclos capaz de efetuar multiplicações de números inteiros. No Lab 14, analisaremos o pipe Multiply em detalhe.

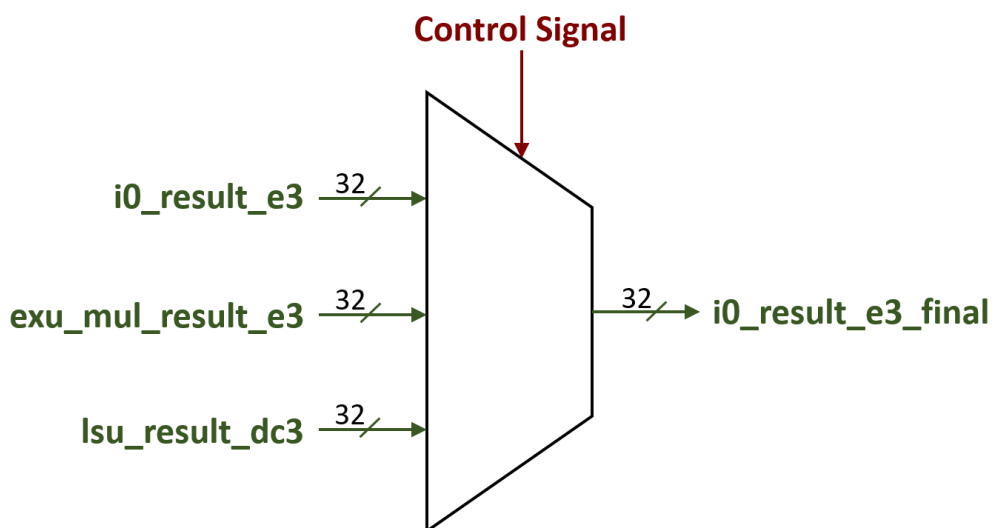
Pipe de Load/Store (L/S): O pipe L/S é mostrado a verde na Figura 4. No Lab 13 exploramos este caminho do pipeline em profundidade. As instruções de carregamento e armazenamento são executadas através do pipe L/S. Inclui 3 andares:

- **DC1:** No primeiro andar, a Unidade Adder calcula o endereço adicionando o endereço base do registo e o deslocamento imediato.
- **DC2:** No segundo andar, as instruções de load lêem a memória utilizando o endereço calculado em DC1. Se o endereço for mapeado para o DCCM, a latência de acesso é de apenas 1 ciclo e o pipeline continua sem paragens (stalls). No entanto, se o acesso for mapeado para a memória principal, o pipeline pode ter de ser interrompido durante vários ciclos, dependendo da utilização de loads bloqueantes/não bloqueantes e da existência de dependências, como analisaremos em laboratórios futuros.
- **DC3:** No terceiro andar, os dados são alinhados e fundidos (por exemplo, se um store anterior para o mesmo endereço ainda estiver a ser executado, os dados desse store podem ter de ser encaminhados para o load). Neste andar, as instruções store começam a escrever na memória, o que continuará durante vários ciclos. Se a escrita for mapeada para o DCCM, tanto os dados como o endereço são colocados em buffer no Store Buffer antes de serem enviados para o DCCM, como analisamos no Lab 13; se a escrita for mapeada para a Memória Principal, tanto os dados como o endereço são enviados para a Memória Externa através do barramento AXI (o controlador DRAM Lite gere os acessos a esta memória).

Divisor: O divisor é apresentado a branco na Figura 4. É uma unidade sem pipeline que requer até 34 ciclos para calcular o seu resultado. O Laboratório 14 analisa o divisor em mais pormenor.

Dois multiplexers 3:1: No final da terceira fase de execução (EX3/DC3/M3), como ilustrado na Figura 4, o resultado das instruções é selecionado a partir do pipe apropriado (I0/I1, MUL, ou L/S) usando dois multiplexers 3:1, um para cada via. Estes multiplexers estão localizados no módulo **dec_decode_ctl**. O multiplexer superior, associado à via 0, é mostrado na Figura 8. As três entradas para este multiplexer são:

1. **Resultado do pipe I0:** `i0_result_e3`. O Lab 12 analisa este caminho.
2. **Resultado do pipe L/S:** `lsu_result_dc3`. O Lab 13 analisa este caminho.
3. **Resultado do pipe de Multiplicação:** `exu_mul_result_e3`. O Lab 14 analisa este caminho.



```
2268 assign i0_result_e3_final[31:0] = (e3d.i0v & e3d.i0load) ? lsu_result_dc3[31:0] : (e3d.i0v & e3d.i0mul) ? exu_mul_result_e3[31:0] : i0_result_e3[31:0];
```

Figura 8. 3:1 multiplexer para selecionar o resultado EX3: diagrama e Verilog

TAREFA: Analisar os bits de controlo do multiplexer da Figura 8. Note-se que os bits de controlo estão no sinal `e3d`, que foi pipelined do sinal `dd`, que foi gerado no andar de decodificação pela unidade de controlo (ver `RVfpga_SweRVref.docx` para descrição dos bits de controlo).

iii. Andar de Commit

No andar de Commit, dois multiplexers 3:1, um por via, selecionam o resultado a escrever no ficheiro de registo (ver Figura 4). O multiplexer superior, associado à via 0, é mostrado na Figura 9. Tem três entradas:

1. **Resultado EX3:** `i0_result_e4`. (A saída do multiplexer 3:1 do EX3).
2. **Dados de leitura corrigidos:** `lsu_result_corr_dc4`. Lab 13 analisa este

caminho.

3. **Resultado da ALU secundária:** `exu_i0_result_e4`. Estas ALUs não são mostradas na Figura 4 para simplificar. Como já referimos, permitem que as instruções de aritmética-lógica sejam repetidas quando necessário devido a conflitos de dados (ver Lab 15 para mais pormenores).

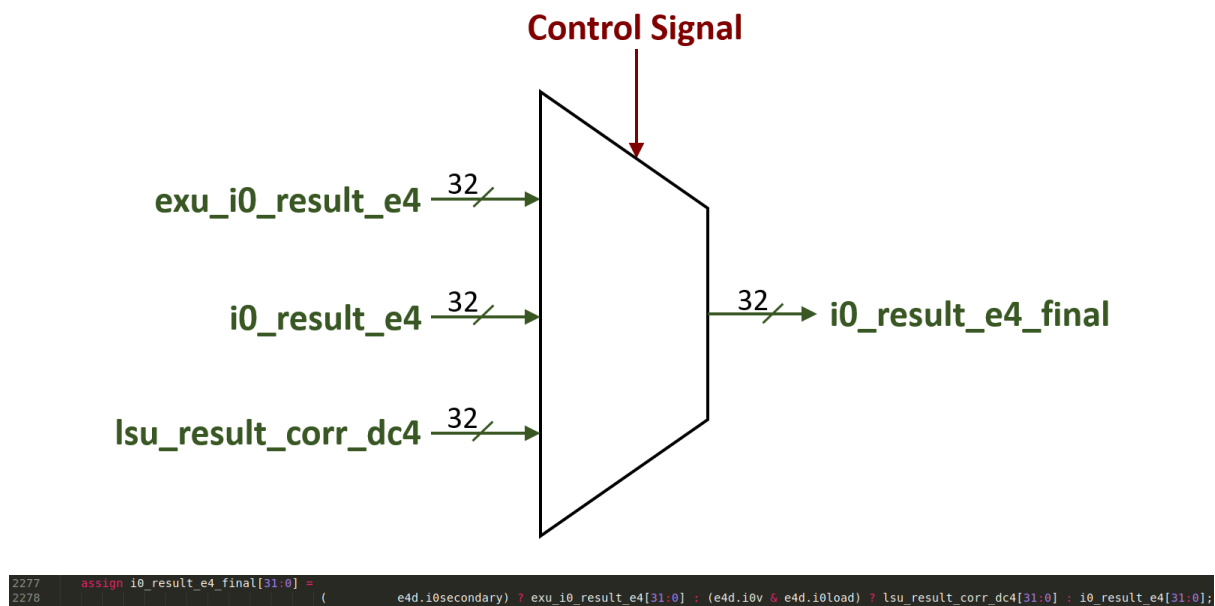


Figura 9. Multiplexer 3:1 para seleccionar o resultado final: diagrama e Verilog


TAREFA: Analisar os bits de controlo do multiplexer da Figura 9, que pode ser encontrado no módulo `dec_decode_ctl`.

iv. Andar Writeback

A fase final, a fase de Writeback, escreve os resultados no Register File utilizando os dois primeiros portos de escrita (0 e 1) ilustrados na Figura 6 (no Lab 14 veremos quando o terceiro porto de escrita - 2 - for utilizado). Nem todos os ciclos escreverão dois resultados: algumas instruções não escrevem um registo (i.e., instruções de salto, instruções de escrita...), e nem todos os ciclos executam duas instruções. Os identificadores de registo e os sinais de habilitação foram gerados no andar de descodificação e são fornecidos pelos Registos de Controlo do Pipeline.

D. Exemplo de Simulação com o Verilator

Nesta secção, ilustramos a simulação de duas instruções a executar em paralelo no pipeline SweRV EH1, mostrando os sinais introduzidos nas secções anteriores. Os laboratórios futuros também usarão simulações do Verilator para visualizar os sinais internos do processador e para ilustrar as explicações teóricas.

De seguida, executamos o código de exemplo apresentado na Figura 10, concentrando-se nas instruções `mul` e `add` (destacadas a vermelho), que fazem parte do ciclo infinito. A pasta `[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram` fornece o projeto PlatformIO para que possa analisar, simular e alterar o programa como desejar. Abra o projeto no PlatformIO e construa-o (lembre-se de que, no Guia de Iniciação, pode construir o projeto clicando no botão , localizado na parte inferior do VSCode). O ficheiro disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram/.pio/build/swervolf_nexys/firmware.dis`) mostra os endereços e o código de máquina. Observe que as duas instruções estão nos endereços `0x000000f0` e `0x000000f4`:

<code>0x000000f0:</code>	<code>03de8e33</code>	<code>mul</code>	<code>t3,t4,t4</code>
<code>0x000000f4:</code>	<code>01ff0f33</code>	<code>add</code>	<code>t5,t5,t6</code>

Estas duas instruções estão envoltas em várias instruções `nop` (no-operation) a fim de as isolar de outras instruções e poder analisá-las melhor. A instrução `nop` não altera o estado do sistema. No RISC-V, `nop` é traduzido para `addi x0,x0,0`, que é codificada como uma instrução de máquina de 32 bits com o valor `0x00000013`. Neste código, definimos várias macros para inserir um número de instruções `nop` (de 1 a 10) no nosso código (para simplificar, as definições das macros não estão incluídas na Figura 10 mas podem ser vistas no projeto PlatformIO).

Para maior clareza, desativamos o Preditor de Saltos (Branch Predictor) e as instruções comprimidas, seguindo o procedimento que explicamos na Secção 2 do documento `RVfpga_SweRVref`.

```
li x28, 0x1
li x29, 0x2
li x30, 0x4
li x31, 0x1

REPEAT:
    mul x28, x29, x29    # x28 = 2 * 2 = 4 (later iterations: 3*3=9, 4*4=16, ...)
    add x30, x30, x31    # x30 = 4 + 1 = 5 (later iterations: 5+1=6, 6+1=7, ...)
    INSERT_NOPS_10
    add x29, x29, 1      # x29 = x29 + 1
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repete o ciclo
```

Figura 10. Programa de exemplo que contém as instruções `mul` e `add` dentro de um ciclo

A Figura 11 e 12 mostra as formas de onda do Verilator dos sinais do processador durante a execução do programa da Figura 10. A Figura 11 mostra os sinais dos três primeiros andares do pipeline (FC1, FC2 e Align - ver Figura 3). A Figura 12 mostra os sinais dos restantes andares (ver Figura 4). Dividimos os resultados em duas figuras por uma questão de coerência com a Figura 3 e a Figura 4, mas lembrem-se que estas duas instruções partem dos andares Align (à direita da Figura 11) para o andar de Decode (à esquerda da Figura 12).

Os seguintes sinais são incluídos nas figuras para seguir as instruções à medida que progridem no pipeline (`ifu` para as instruções nos andares de Align, `dec` para as instruções no andar de Decode, `eX` para as instruções nos andares X (X = primeiro, segundo, terceiro) andar de Execute, `e4` para as instruções no andar de Commit, e `wb` para as instruções no andar de Writeback) e saber a que via estão associados (`i0` para a Via 0 e `i1` para a Via 1).

- ifu_i0_instr **e** ifu_i1_instr → instruções no andar de Align
- dec_i0_instr_d **e** dec_i1_instr_d → instruções no andar de Decode
- i0_inst_e1 **e** i1_inst_e1 → instruções no andar de EX1
- i0_inst_e2 **e** i1_inst_e2 → instruções no andar de EX2
- i0_inst_e3 **e** i1_inst_e3 → instruções no andar de EX3
- i0_inst_e4 **e** i1_inst_e4 → instruções no andar de Commit
- i0_inst_wb **e** i1_inst_wb → instruções no andar de Writeback

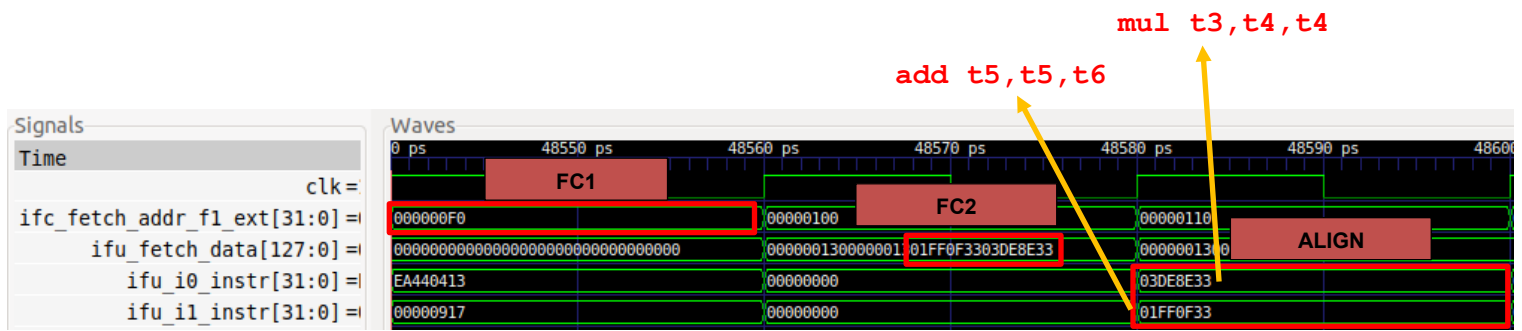


Figura 11. Simulação dos três primeiros andares do pipeline: FC1, FC2 e Align

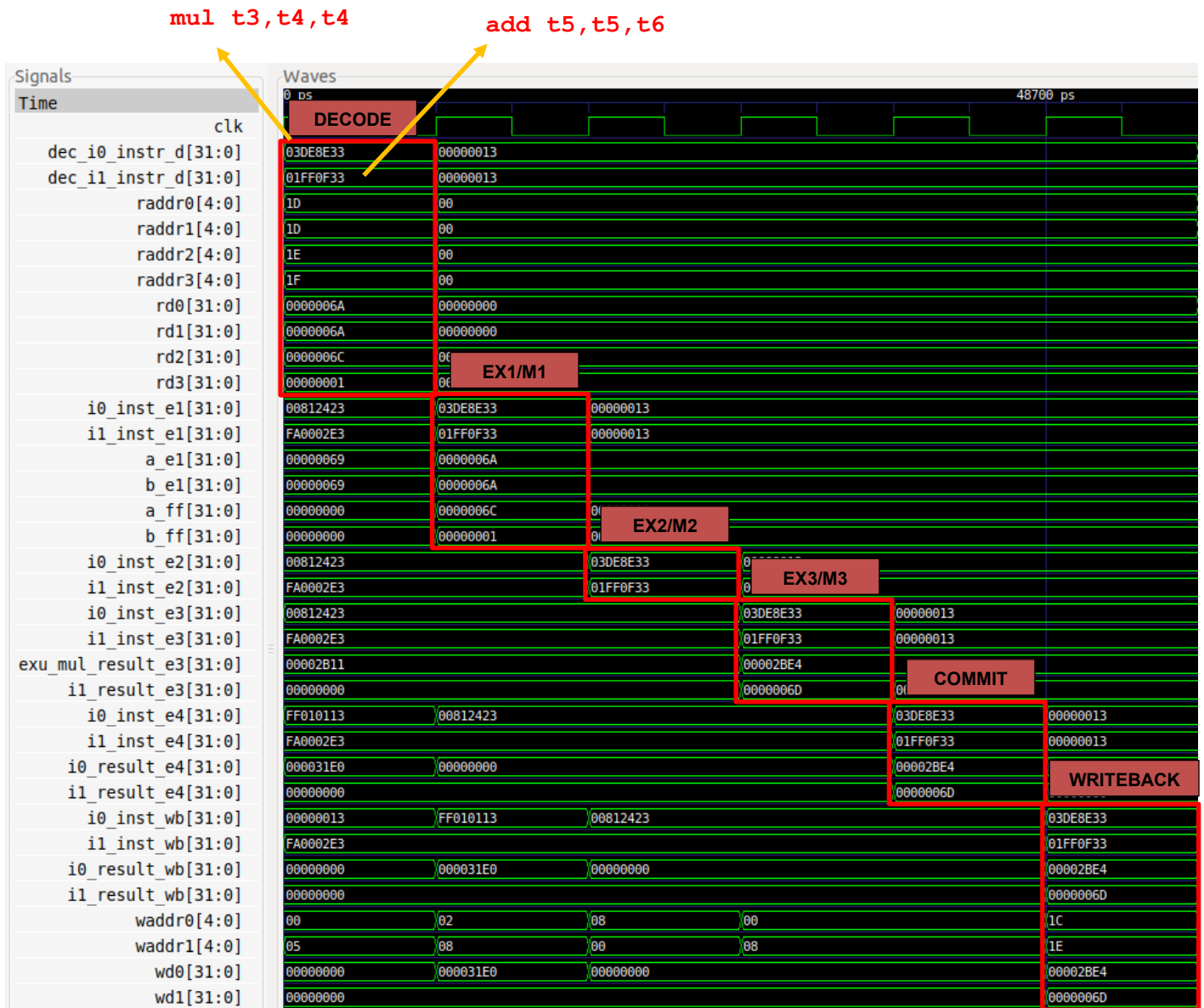



Figura 12. Simulação dos andares finais: Decode, EX1/M1, EX2/M2, EX3/M3, Commit, Writeback

TAREFA: Replicar a simulação da Figura 11 e da Figura 12 no seu próprio computador, seguindo estes passos (como descrito em pormenor na Secção 7 do GSG):

- Se necessário, gerar o binário de simulação (*Vrvfpgasim*).
- No PlatformIO, abra o projeto fornecido em:
[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram.
- Definir 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 com GTKWave.
- Utilizar os ficheiros *test_1.tcl* e *test_2.tcl* (fornecidos em *[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram*) para abrir os mesmos sinais que os apresentados na Figura 11 e na Figura 12. Para o efeito, no GTKWave, clique em *File → Read Tcl Script File* e seleccionar o ficheiro *test_1.tcl* ou *test_2.tcl*.
- Clique em *Zoom In* () várias vezes e passe para 48600ps (ou qualquer outra iteração do ciclo, exceto a primeira).

Analisar o diagrama de forma de onda da Figura 11 e Figura 12 e os diagramas da Figura 3 e Figura 4 em simultâneo. As figuras incluem alguns sinais associados com cada um dos andares do pipeline. Os valores assinalados a vermelho correspondem às duas instruções (*mul* e *add*) à medida que passam pelo pipeline.

- **FC1:** No primeiro ciclo da Figura 11, o sinal *ifc_fetch_addr_f1_ext[31:0]* (o Program Counter, que é fornecido à memória de instruções) contém o endereço de (i.e., *aponta para*) a instrução *mul* (*ifc_fetch_addr_f1_ext* = 0x000000F0).
- **FC2:** No segundo ciclo da Figura 11, a Memória de Instruções fornece um novo sinal de 128 bits que inclui as duas instruções que estamos a analisar no exemplo (*mul* é apresentado a verde e *add* é apresentado a vermelho):

```
ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33
```

- **Align:** No último ciclo da Figura 11, as duas instruções são extraídas do novo sinal de 128 bits e distribuídas pelas duas vias que o SweRV EH1 possui.

```
ifu_i0_instr = 0x03DE8E33 (Way 0)
ifu_i1_instr = 0x01FF0F33 (Way 1)
```

- **Decode:** No primeiro ciclo da Figura 12, as duas instruções são decodificadas - isto é, os valores de registo das instruções são lidos a partir do Register File, e os bits de controlo são gerados (não mostrados na figura, mas pode adicionar alguns deles como descrito em *RVfpga_SweRVref.docx*). Os operandos (valores de registo) são colocados em *rd0*, *rd1*, *rd2*, e *rd3*.

```
rd0 = 0x0000006A
rd1 = 0x0000006A
rd2 = 0x0000006C
rd3 = 0x00000001
```

- **EX1/M1, EX2/M2, EX3/M3 e Commit:** Nos três ciclos seguintes da Figura 12, a adição e a multiplicação são efectuadas. No final do EX3/M3, os resultados são seleccionados utilizando os dois multiplexers 3:1 e depois propagados para o andar de Commit.

```
i0_result_e4 = exu_mul_result_e3 = 0x6A * 0x6A = 0x2BE4
i1_result_e4 = i1_result_e3      = 0x6C + 0x01 = 0x6D
```

- **Writeback:** No último ciclo da Figura 12, os resultados são escritos de volta no Register File.

```
waddr0 = 0x1C      wd0 = 0x2BE4
waddr1 = 0x1E      wd1 = 0x6D
```

3. CONTADORES EM HARDWARE NO SWERV EH1

De seguida, mostramos como utilizar os contadores de desempenho para analisar o desempenho do processador. Os contadores em hardware são um conjunto de registos para fins especiais incluídos na maioria dos processadores atuais para registar uma variedade de métricas, tais como o número de instruções executadas, o número de ciclos executados, a média de ciclos de relógio por instrução (CPI), o número de acertos/erros da cache de instruções, o número de saltos previstos certos/errados, etc.

Nos Labs 12-20, utilizaremos regularmente os contadores de desempenho disponíveis no SweRV EH1 para medir e comparar as diferentes magnitudes.

BENCHMARKS REAIS: Na pasta `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks` fornecemos três aplicações reais (CoreMark, Dhrystone e Processamento de Imagem) que irá utilizar no Lab 20 para testar as diferentes características do nosso processador SweRV EH1. O documento suplementar, `RVfpga_SweRVref.docx`, descreve brevemente estas aplicações na Secção 6, e o Laboratório 20 estende estas descrições e propõe várias tarefas.

A. Contadores de Desempenho no SweRV EH1

O Manual de Referência do Programador RISC-V SweRV EH1

(https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) descreve as capacidades básicas de monitorização do

desempenho do hardware de um processador RISC-V. Devem ser implementados os seguintes contadores de desempenho, que são também registos de controlo e de estado (CSR):

- *mcycle*: número de ciclos de relógio do *hart* (hardware thread) foi executado desde um momento arbitrário no passado.
- *minstret*: número de instruções que o *hart* já retirou desde um momento arbitrário no passado.
- *mhpmcounter3–mhpmcounter31*: 29 outros contadores de eventos. O seletor de eventos CSRs, *mhpmevent3–mhpmevent31*, são registos WARL (Write Any value, Read Legal values - escrever qualquer valor, ler valores válidos) que controlam qual o evento que provoca o incremento do contador correspondente. O significado destes eventos é definido pela plataforma, mas o evento 0 está reservado para significar "nenhum evento".

Nem todos os contadores precisam de ser implementados. É permitido ligar o contador e o seu correspondente seletor de eventos a 0. Especificamente, no SweRV EH1, apenas os

contadores de eventos 3 a 6 (*mhpcounter3-mhpcounter6*) e os seus correspondentes seletores de eventos (*mhpmevent3-mhpmevent6*) são funcionais, enquanto que os contadores de eventos 7 a 31 (*mhpcounter7-mhpcounter31*) e os seus correspondentes seletores de eventos (*mhpmevent7-mhpmevent31*) são ligados a "0". A ativação destes contadores é controlada através do bit 0 do registo mgpmc (0 = desativar, 1 = ativar).

O capítulo 7 do Manual de Referência do Programador SweRV EH1 (https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) descreve em pormenor as características e o funcionamento dos quatro contadores de desempenho disponíveis no SweRV EH1:

- Quatro contadores de eventos standard de 64 bits de largura
- Seleção standard de eventos separados para cada contador
- Controlabilidade de ativação/desativação de contagem seletiva standard
- Controlabilidade de ativação/desativação do contador sincronizado
- Contador de ciclos standard
- Contador de instruções standard retiradas
- Suporte para registos standard de temporizadores de baseados em máquina SoC

A Tabela 7-2 nesse documento lista os 50 eventos contabilizáveis disponíveis no SweRV EH1, que estão resumidos na Tabela 1.

Tabela 1. Lista de eventos contáveis no SweRV EH1

0	Reserved	17	CSR read/write	34	Cycles SB/WB stalled
1	Cycles clock active	18	CSR write rd==0	35	Cycles DMA DCCM transaction stalled
2	I-Cache hits	19	Ebreak	36	Cycles DMA ICCM transaction stalled
3	I-Cache misses	20	Ecall	37	Exceptions taken
4	Instrs committed	21	Fence	38	Timer interrupts taken
5	Instrs committed 16-b	22	Fence.i	39	External interrupts taken
6	Instrs committed 32-b	23	Mret	40	TLU flushes
7	Instrs aligned	24	Branches committed	41	Branch error flushes
8	Instrs decoded	25	Branches mispredicted	42	I-bus transactions – instr
9	Muls committed	26	Branches taken	43	D-bus transactions – ld/st
10	Divs committed	27	Unpredictable branches	44	D-bus transactions misaligned
11	Loads committed	28	Cycles fetch stalled	45	I-bus errors
12	Stores committed	29	Cycles aligner stalled	46	D-bus errors
13	Misaligned loads	30	Cycles decode stalled	47	Cycles stalled due to I-bus busy
14	Misaligned stores	31	Cycles postsync stalled	48	Cycles stalled due to D-bus busy
15	Alus committed	32	Cycles presync stalled	49	Cycles interrupts disabled
16	CSR read	33	Cycles frozen	50	Cycles interrupts stalled while disabled

B. Utilização dos Contadores de Desempenho Através do Processor Support Package (PSP) da Western Digital

Utilizar o sistema de monitorização do desempenho ao nível do registo seria um pouco complexo; felizmente, o PSP da WD (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) inclui várias funções que fornecem uma abordagem muito mais simples para o monitorização do desempenho. Se instalou o PlatformIO seguindo as instruções do GSG, deve encontrar os dois ficheiros seguintes no seu sistema Ubuntu:

- `~/.platformio/packages/framework-wd-riscv-sdk/psp/psp_performance_monitor_eh1.c`
- `~/.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h`

Windows: A pasta *.platformio* está localizada dentro da sua pasta de utilizador (C:\Users\<USER>). Note que poderá ser necessário ativar o sistema para visualizar ficheiros/pastas ocultos.

macOS: Como no Linux, a pasta `.platformio` está localizada dentro da sua pasta home (`~/platformio`).

O ficheiro `.c` (`psp_performance_monitor_eh1.c`) implementa funções que permitem fazer coisas como ativar/desativar o monitor de desempenho do grupo (`pspEnableAllPerformanceMonitor`), associar um contador a um evento (`pspPerformanceCounterSet`) ou obter o valor do contador (`pspPerformanceCounterGet`).

O ficheiro `.h` (`psp_performance_monitor_eh1.h`) fornece nomes para cada um dos eventos de Tabela 1 em: `typedef enum pspPerformanceMonitorEvents`.

O seguinte exemplo (Figura 13), fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab11/HwCounters_Example`, ilustra a utilização dos quatro contadores em hardware disponíveis no SweRV EH1 para medir: *ciclos*, *instruções*, e *saltos efetuados e mal previstos*. A função `main`:

- Inicializa a UART (`uartInit()`)
- Ativa os contadores em hardware (`pspEnableAllPerformanceMonitor(1)`)
- Atribui os eventos a serem medidos (*ciclos*, *instruções* e *saltos efetuados e mal previstos*) a cada contador (`D_PSP_COUNTER0` - `D_PSP_COUNTER3`)
- Lê os contadores (`pspPerformanceCounterGet(D_PSP_COUNTER0)`)
- Chama um programa Assembly simples (`Test_Assembly()`) e lê os contadores novamente
- Imprime o valor de cada contador utilizando a função `printfNexys`.

A função `Test_Assembly()`, após algumas inicializações de registos, repete um ciclo 1.000.000 de vezes; o ciclo contém cinco instruções aritméticas-lógicas (A-L) e um salto condicional. O ficheiro `disassembly` também é mostrado no final da Figura 13 para que saiba o valor das instruções de máquina de 32 bits que compõem o corpo do ciclo.

File Test.C

```
#if defined(D_NEXYS_A7)
#include <bsp_printf.h>
#include <bsp_mem_map.h>
#include <bsp_version.h>
#else
    PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>

extern void Test_Assembly(void);

int main(void)
{
    int cyc_beg, cyc_end;
    int instr_beg, instr_end;
    int BrCom_beg, BrCom_end;
    int BrMis_beg, BrMis_end;

    /* Initialize Uart */

    uartInit();

    pspEnableAllPerformanceMonitor(1);
```

```

pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);

cyc_beg  = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);

Test_Assembly();

cyc_end  = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);

printfNexys("Cycles = %d", cyc_end-cyc_beg);
printfNexys("Instructions = %d", instr_end-instr_beg);
printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
printfNexys("BrMis = %d", BrMis_end-BrMis_beg);

while(1);
}

```

File Test_Assembly.S

```

.globl Test_Assembly

.text

Test_Assembly:

li t1, 0x1
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
li a0, 0x0
lui a1, 0xF4
add a1, a1, 0x240
nop

REPEAT:
    add a0, a0, 1
    add t3, t3, t1
    sub t4, t4, t1
    or  t5, t5, t1
    xor t6, t6, t1
    bne a0, a1, REPEAT # Repeat the loop

.end

```

File firmware.dis

```

000001e4 <Test_Assembly>:
1e4: 00100313      li      t1,1
1e8: 00300e13      li      t3,3
1ec: 00400e93      li      t4,4
1f0: 00500f13      li      t5,5
1f4: 00600f93      li      t6,6

```

```

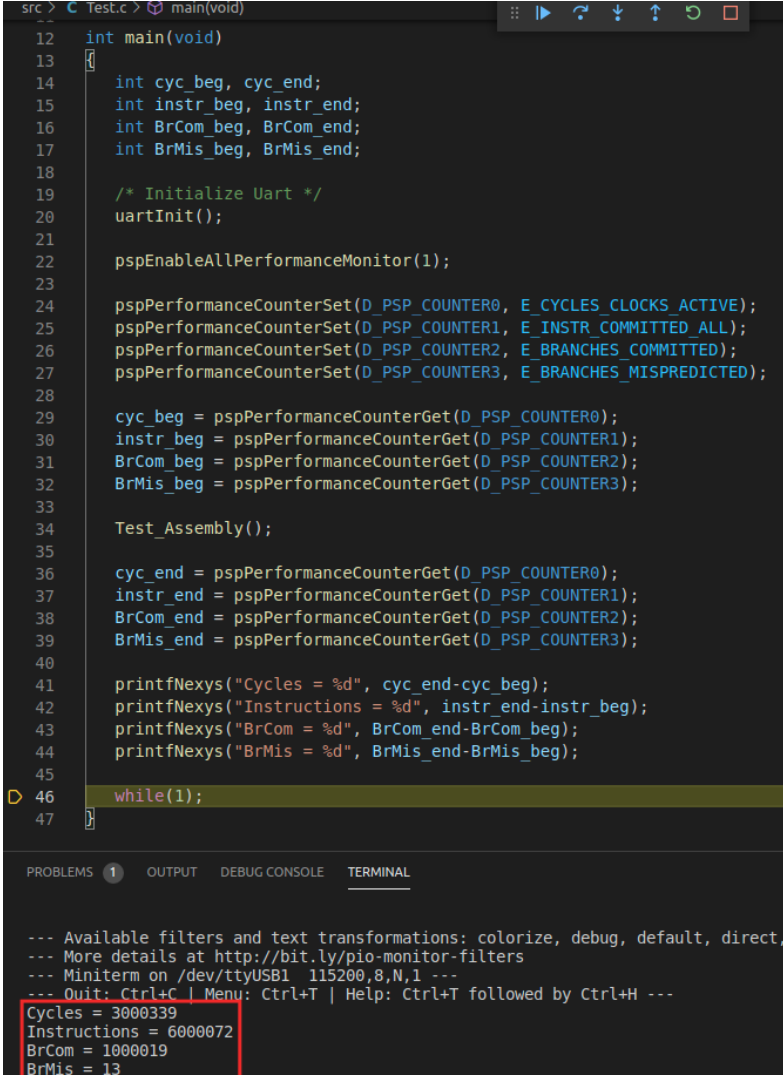
1f8: 00000513      li    a0,0
1fc: 000f45b7      lui    a1,0xf4
200: 24058593      addi   a1,a1,576 # f4240 <_sp+0xf0788>
204: 00000013      nop

00000208 <REPEAT>:
208: 00150513      addi   a0,a0,1
20c: 006e0e33      add    t3,t3,t1
210: 406e8eb3      sub    t4,t4,t1
214: 006f6f33      or     t5,t5,t1
218: 006fcfb3      xor    t6,t6,t1
21c: feb516e3      bne    a0,a1,208 <REPEAT>

```

Figura 13. Test.C, Test_Assembly.S e firmware.dis

TAREFA: Executar o programa da Figura 13 na placa Nexys A7, como explicado no GSG. Deverá obter os resultados apresentados na Figura 14 para os quatro acontecimentos medidos. Explicar e justificar os resultados.



```

src > C Test.C > main(void)
12 int main(void)
13 {
14     int cyc_beg, cyc_end;
15     int instr_beg, instr_end;
16     int BrCom_beg, BrCom_end;
17     int BrMis_beg, BrMis_end;
18
19     /* Initialize Uart */
20     uartInit();
21
22     pspEnableAllPerformanceMonitor(1);
23
24     pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
25     pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
26     pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
27     pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);
28
29     cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
30     instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
31     BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
32     BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
33
34     Test_Assembly();
35
36     cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
37     instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
38     BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
39     BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
40
41     printfNexys("Cycles = %d", cyc_end-cyc_beg);
42     printfNexys("Instructions = %d", instr_end-instr_beg);
43     printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
44     printfNexys("BrMis = %d", BrMis_end-BrMis_beg);
45
46     while(1);
47 }

```

```

--- Available filters and text transformations: colorize, debug, default, direct,
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Cycles = 3000339
Instructions = 6000072
BrCom = 1000019
BrMis = 13

```

Figura 14. Execução do Test.C

TAREFA: Medir outros eventos nos contadores em hardware para o programa da Figura 13. Para o efeito, é necessário modificar no ficheiro *Test.c* a configuração dos acontecimentos a medir com a função `pspPerformanceCounterSet`. Note-se que os diferentes eventos (mostrados na Tabela 1) pode ser configurado utilizando as macros definidas no ficheiro PSP da WD: *.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h*. Por exemplo, se quiser medir o número de falhas de l\$ em vez do número de falhas de saltos, deve substituir no ficheiro *Test.c* a linha: `pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);` pela linha: `pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);`

TAREFA: Propor outros programas no âmbito da função `Test_Assembly` e verificar se os diferentes eventos fornecem os resultados esperados. Pode tentar outras instruções, como leituras, escritas, multiplicações, divisões... bem como conflitos que provoquem paragens do pipeline.