

RVfpga

Um Curso Completo para Compreender Arquiteturas de Computadores



RVfpga v2.2 © 2022 <1>
Imagination Technologies



RVfpga Introdução

Agradecimentos

AUTORES

Prof. Sarah Harris
Prof. Daniel Chaver
Zubair Kakakhel
M. Hamza Liaqat

ORIENTADOR

Prof. David Patterson

COLABORADORES

Robert Owen
Olof Kindgren
Prof. Luis Piñuel
Ivan Kravets
Valerii Koval
Ted Marena
Prof. Roy Kravitz

ASSOCIADOS

Prof. José Ignacio Gómez	Prof. Francisco Tirado	Gage Elerding
Prof. Christian Tenllado	Prof. Román Hermida	Prof. Brian Cruickshank
Prof. Daniel León	Prof. Ataur Patwary	Deepen Parmar
Prof. Katzalin Olcoz	Cathal McCabe	Thong Doan
Prof. Alberto del Barrio	Dan Hugo	Oliver Rew
Prof. Fernando Castro	Braden Harwood	Niko Nikolay
Prof. Manuel Prieto	Prof. David Burnett	Guanyang He

Patrocinadores e Apoiantes

Western Digital.

 **Imagination**

 **CHIPS
ALLIANCE**

 **RISC-V®**

 **DIGILENT®**
A National Instruments Company

 **XILINX.**
| UNIVERSITY PROGRAM

 **Digi-Key®**
ELECTRONICS

 **Esperanto**
TECHNOLOGIES

 **codasip®**


硬禾学堂

 **ANDES**
TECHNOLOGY

 **PLATFORMIO.ORG**

 **RISC-V®**

RVfpga v2.2 © 2022 <3>
Imagination Technologies

 **Imagination**

Introdução

- RISC-V FPGA (**RVfpga**) é um curso que mostra como:
 - Realizar um **core RISC-V comercial** SweRV & um System-on-Chip (SoC) numa FPGA
 - Programar o SoC RISC-V
 - Adicionar funcionalidades extra ao SoC RISC-V
 - Analisar e modificar o núcleo do RISC-V e a hierarquia de memória
- O material didático está a ser desenvolvido pela **Imagination Technologies** e pelos parceiros académicos e industriais.
- Depois de concluírem o Curso RVfpga, os participantes terão um **processador RISC-V comercial, SoC e ecossistema** que compreendem e sabem como utilizar e modificar

Esta secção de slides abrange o material da Secção 1 do **Guia de Introdução (GSG)**.

Dois cursos RVfpga

- A Imagination Technologies oferece dois cursos baseados na arquitetura RISC-V:
 - **RVfpga**: o curso abordado nestes diapositivos, sobre o núcleo RISC-V, o sistema de memória e os periféricos
 - **RVfpga-SoC**: um segundo curso que mostra como:
 - **Construir um SoC RISC-V** a partir de blocos base
 - Instalar o RTOS (sistema operativo em tempo real) **Zephyr**
 - Executar programas no Zephyr
 - Executar programas **Tensorflow** simples(RVfpga-SoC não é abordado nestes slides)

Descarregar o Material da Imagination Technologies

- Ambos os cursos (RVfpga e RVfpga-SoC) estão disponíveis em downloads separados (gratuitos após inscrição) em:
<https://university.imgtec.com/rvfpga/>
- O resto destes slides centram-se no **Curso RVfpga**.

RVfpga Audiência e Historial

- **Público-alvo**

- Estudantes de licenciatura e de mestrado em engenharia eletrotécnica, ciências informáticas ou engenharia informática
- Académicos e profissionais da indústria interessados em aprender a arquitetura RISC-V

- **Historial do Programa Universitário da Imagination**

(IUP): Desenvolveu o programa MIPSfpga:

- Lançado em Abril de 2015
- Envolveu 800 universidades
- Vencedor: Prémio para o Melhor Apoio Educativo Elektra, Europa 2015

Conteúdo do Curso RVfpga

Curso RVfpga

- **Curso de 2-3 Semestres**

- Licenciatura (Labs 1-10)
- Mestrado ou mais avançado (Labs 11-20)

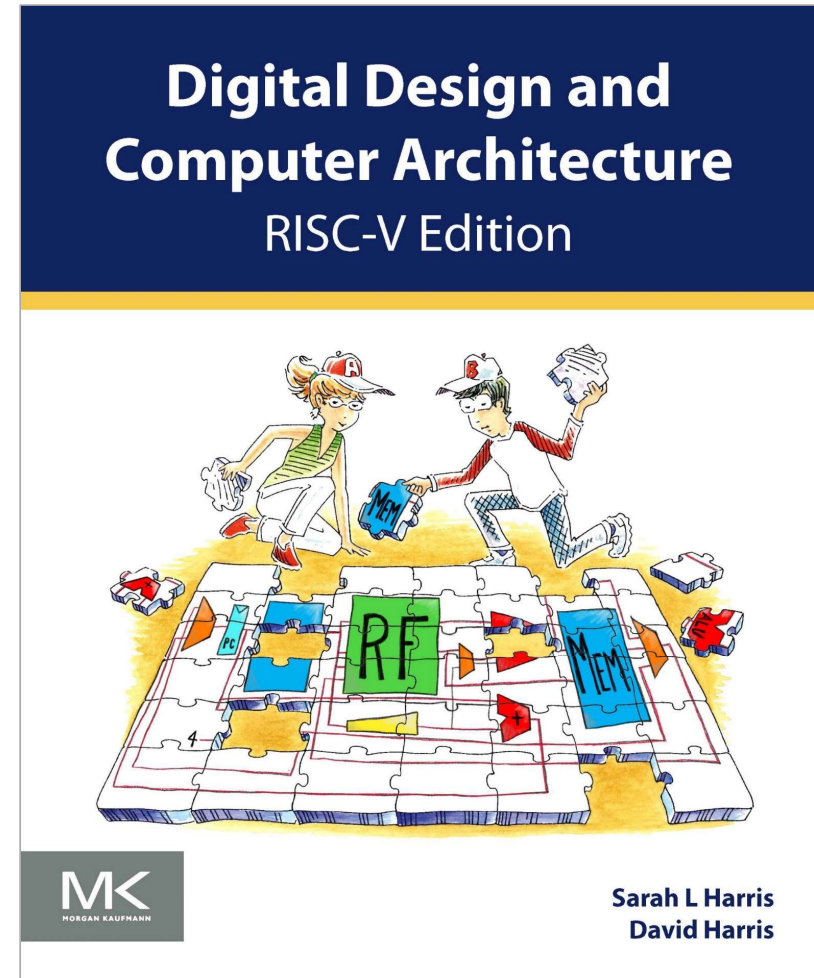
Esta secção de slides cobre o material da
Secção 1 do Guia de Introdução (GSG).

- **Conhecimentos Prévios Desejados**

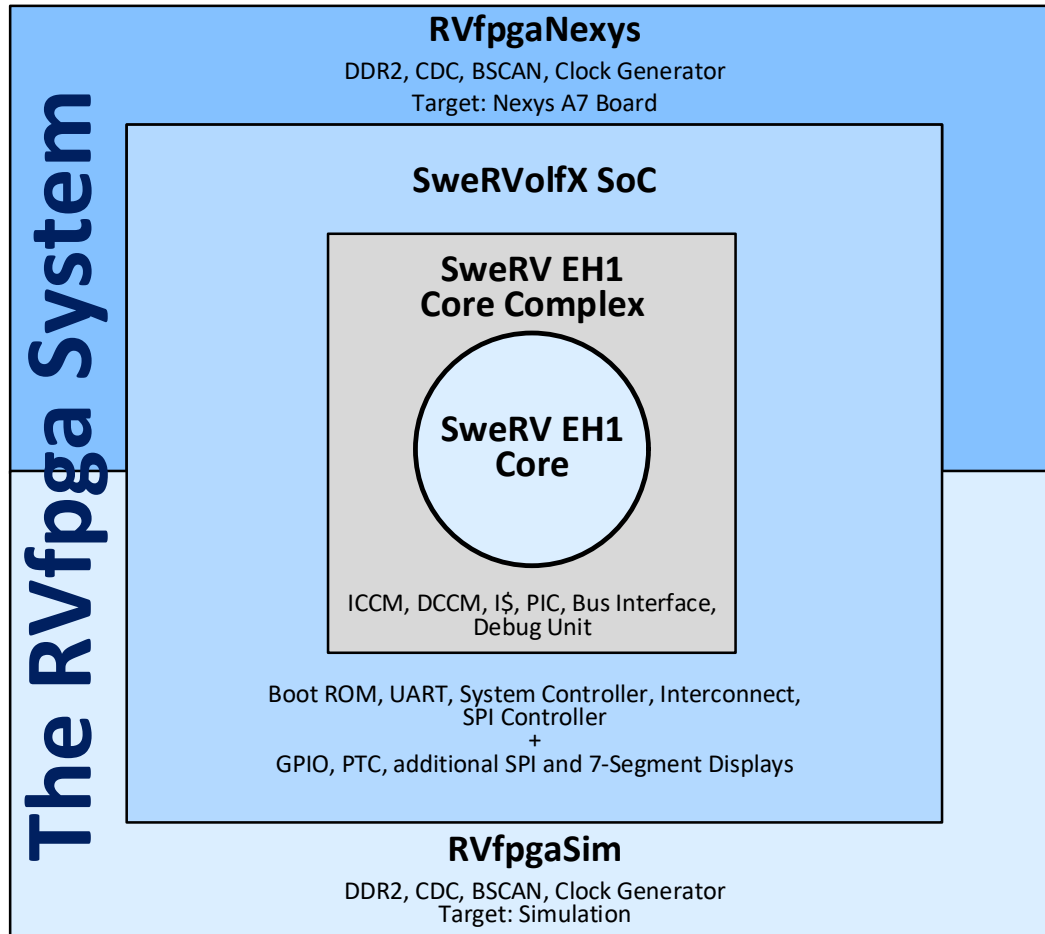
- Projeto de sistemas digitais
- Programação de alto nível (de preferência C)
- Arquitetura de conjuntos de instruções e programação Assembly
- Microarquiteturas de processadores
- Sistemas de memória
- Este material é coberto em Design Digital e Arquitetura de Computadores: Edição RISC-V, Harris & Harris, © Elsevier, publicação prevista: Verão 2021)
- Estes tópicos serão alargados e solidificados com a aprendizagem prática ao longo do curso RVfpga

Livro de Estudo

Leitura recomendada
antes de iniciar o curso
RVfpga: ***Digital Design
and Computer
Architecture: RISC-V
Edition***, Harris & Harris,
© Elsevier, 2021



Sistema RVfpga



SweRV Core™

- Núcleo de código aberto da Western Digital
- Núcleo super-escalar de 2-vias
- Pipeline de 9 andares
- Execução ordenada (*in-order*)
- RV32IMC

Software e Hardware Necessários para o RVfpga

SOFTWARE

Xilinx **Vivado** 2019.2 WebPACK

PlatformIO – uma extensão do **Visual Studio Code** da Microsoft - com a plataforma Chips Alliance, que inclui: RISC-V Toolchain, OpenOCD, Verilator HDL Simulator, WD Whisper simulador de conjunto de instruções (ISS)

Todos são gratuitos, exceto a placa FPGA, que custa \$265 (preço acadêmico: \$199)

HARDWARE*

Placa FPGA **Nexys A7** / Nexys 4 DDR da Digilent

*Opcional: Todos os laboratórios podem ser completados apenas em simulação; por isso, este hardware é recomendado mas não é necessário.

CORE RISC-V & SOC

Core: SweRV EH1** da Western Digital

SoC: SweRVolf** da Chips Alliance

**Open-source - e fornecido no material RVfpga.

Plataformas suportadas

- **Sistemas operativos**
 - Ubuntu 18.04 e 20.04
 - Windows 10
 - macOS

Cores e SoCs

RISC-V

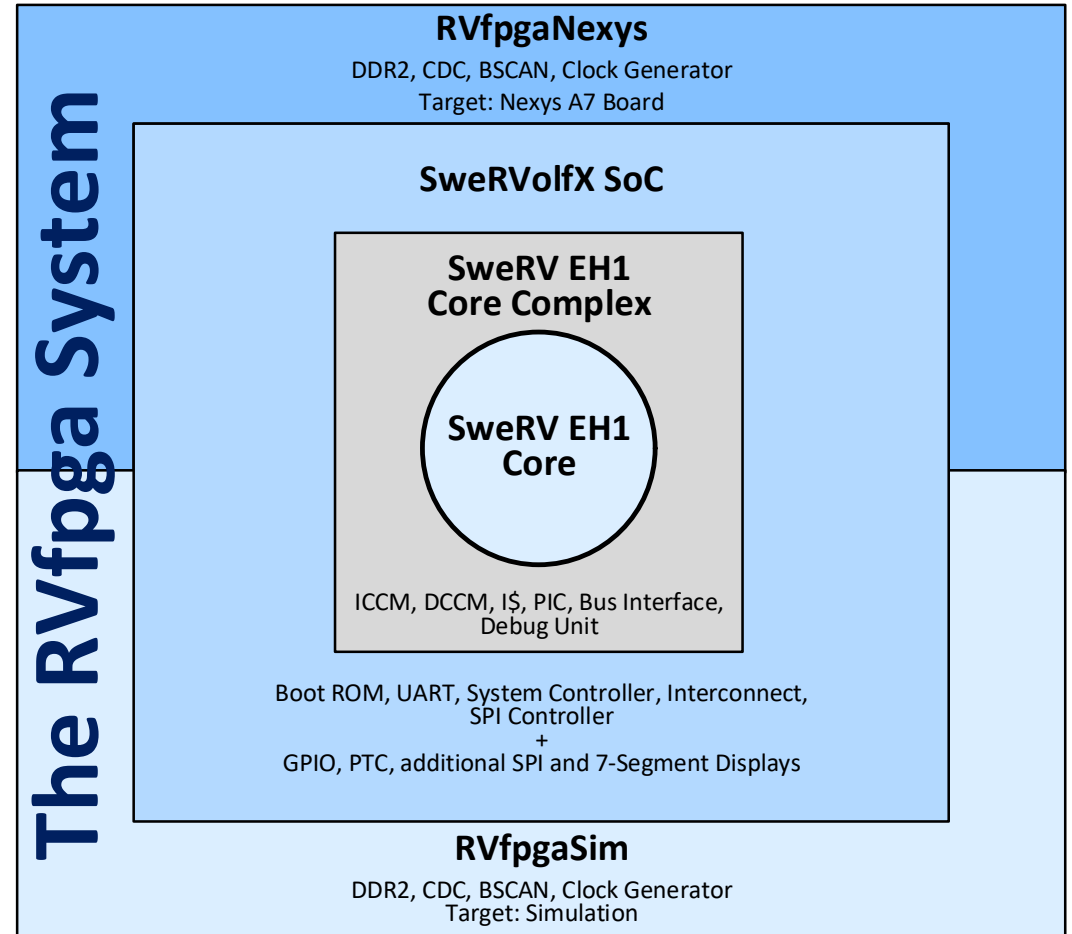
Introdução

- O sistema RVfpga é uma extensão do **SoC SweRVolf** da Chips Alliance, que se baseia no núcleo RISC-V **SweRV EH1** da Western Digital.
- O código-fonte do SoC e do núcleo é fornecido com o download do RVfpga da Imagination Technologies.
- O sistema RVfpga é também designado simplesmente por "RVfpga".

Esta secção de diapositivos abrange o material da **Secção 1 do Guia de Iniciação**.

Hierarquia RVfpga

- **SweRV EH1 Core / Core Complex**
 - Inclui processador, memória e barramento de interface
- **SoC SweRVolfX**
 - Versão estendida do SweRVolf
 - Adiciona periféricos
- **Sistema RVfpga**
 - **RVfpgaNexys**: SweRVolfX direccionado para hardware (placa FPGA Nexys A7, com memória integrada, relógio, etc.)
 - **RVfpgaSim**: SweRVolfX direccionado para a simulação



Hierarquia RVfpga

Nome	Descrição
SweRV EH1 Core	Núcleo comercial de código aberto RISC-V desenvolvido pela Western Digital (https://github.com/chipsalliance/Cores-SweRV).
SweRV EH1 Core Complex	SweRV EH1 core com memória adicional (ICCM, DCCM, e cache de instruções), controlador de interrupção programável (PIC), interfaces de barramentos, e unidade de depuração (https://github.com/chipsalliance/Cores-SweRV).
SweRVolfX (Extended SweRVolf)	System-on-a-Chip utilizado no curso de RVfpga. É uma extensão do SweRVolf. <u>SweRVolf</u> (https://github.com/chipsalliance/Cores-SweRVolf): Um SoC de código aberto construído em torno do SweRV EH1 Core Complex. Adiciona uma ROM de arranque, interface UART, controlador de sistema, interligação (AXI Interconnect, Wishbone Interconnect, e ponte AXI-para-Wishbone), e um controlador SPI. SweRVolfX: Adiciona 4 novos periféricos ao SweRVolf: um GPIO, um PTC, um SPI adicional e um controlador para os 8 mostradores de 7 segmentos.

Hierarquia RVfpga

Nome	Descrição
RVfpgaNexys	O SoC SweRVolfX foi concebido para a placa Nexys A7 e os seus periféricos. Acrescenta uma interface DDR2, unidade CDC (clock domain crossing), lógica BSCAN (para a interface JTAG) e gerador de relógio. RVfpgaNexys é o mesmo que SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), excepto que este último é baseado no SweRVolf.
RVfpgaSim	O SoC SweRVolfX com um encapsulamento de banco de ensaio (<i>testbench wrapper</i>) e memória AXI destinado à simulação. RVfpgaSim é o mesmo que SweRVolf sim, (https://github.com/chipsalliance/Cores-SweRVolf), exceto que este último é baseado no SweRVolf.

SweRV EH1 Core e SweRV EH1 Core Complex

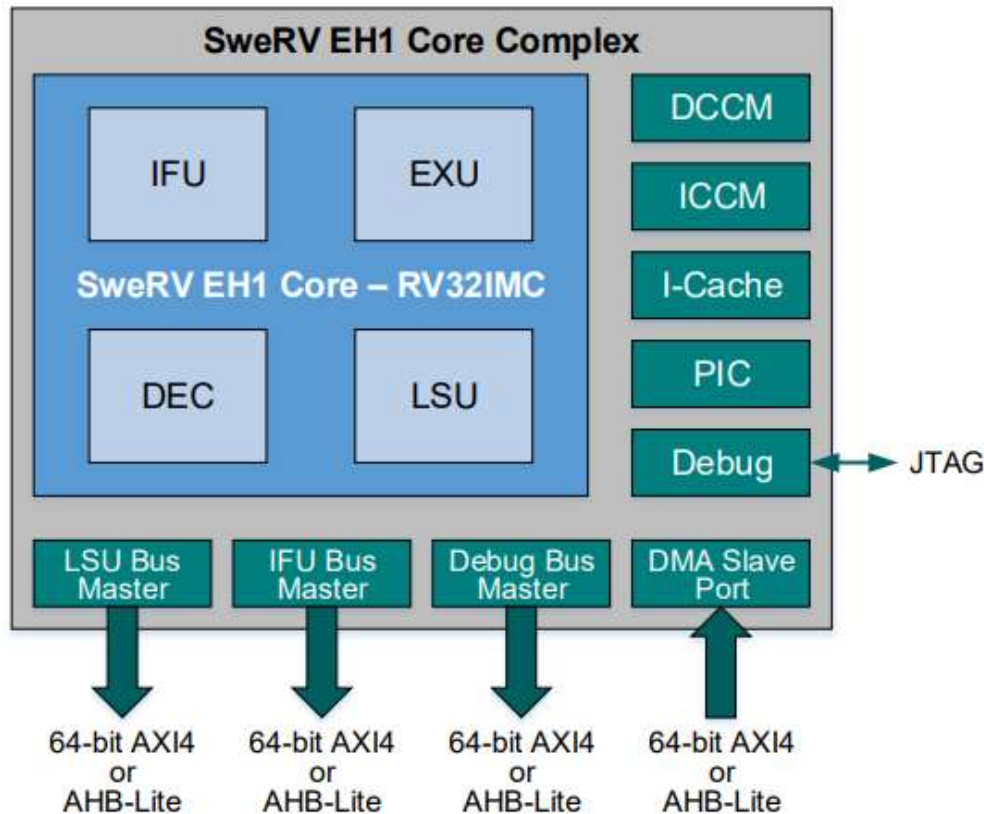
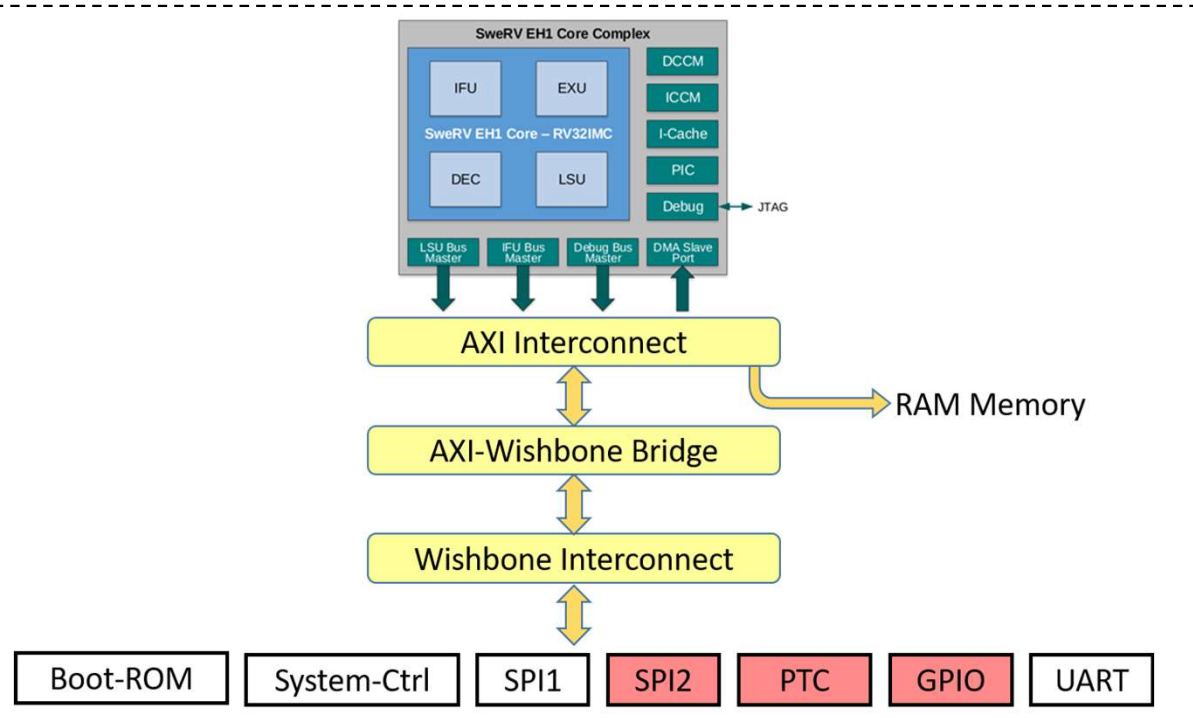


Figura de https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

- Núcleo de código aberto da Western Digital
- Núcleo superescalar de 32 bits (RV32IMC), com pipeline de 9 fases e dupla emissão de instruções
- Memórias de Instruções e dados separadas (ICCM e DCCM) e acopladas diretamente ao núcleo
- Cache de Instruções I\$ associativa de 4 vias com paridade ou proteção ECC
- Controlador de Interrupção Programável
- Unidade de Depuração Central em conformidade com a especificação RISC-V Debug
- System Bus: AXI4 ou AHB-Lite

SweRVolfX SoC

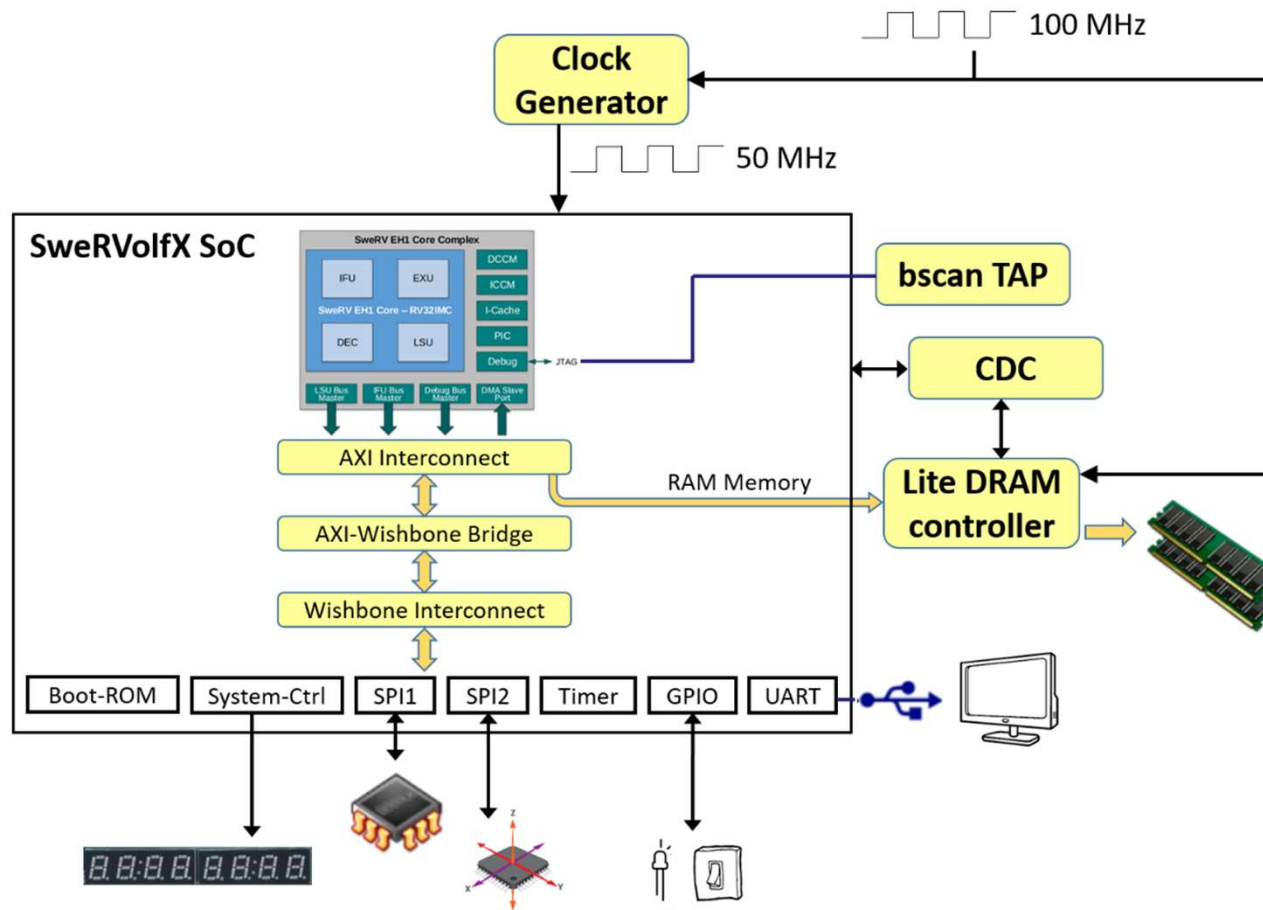


Mapa de Memória
do SweRVolfX

Sistema	Endereço
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
Timer	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

- System-on-chip (SoC) de código aberto da Chips Alliance
- SweRVolf usa o núcleo SweRV EH1. O SweRVolf inclui uma Boot ROM, UART, um Controlador de Sistema e um controlador SPI (SPI1)
- SweRVolfX estende o SweRVolf com outro controlador SPI (SPI2), e GPIO (General Purpose Input/Output), 8 mostradores de 7-Segmentos e um PTC (a vermelho).
- SweRV EH1 Core usa um barramento AXI e os periféricos usam um barramento Wishbone bus, pelo que o SoC também tem uma ponte AXI-Wishbone

RVfpgaNexys



- **RVfpgaNexys:** SweRVolfX SoC para a placa FPGANexys A7 com os periféricos:

- **Core & Sistema:**

- SweRVolfX SoC
- Controlador DRAM Lite
- Gerador de sinal e domínio de relógio, e lógica BSCAN para o porto JTAG

- **Periféricos** usados na placa FPGA Nexys A7 :

- Memória DDR2
- Ligação UART via USB
- Memória Flash SPI
- 16 LEDs e 16 interruptores
- Acelerómetro SPI
- 8 mostradores de 7-segmentos

RVfpgaSim

- **RVfpgaSim** é o SweRVolfX SoC acoplado a um banco de ensaio (*testbench*) para ser utilizado por simuladores HDL.

Extensões do Sistema RVfpga

- O Sistema Rvfpga é **aprofundado** nos laboratórios 6-10 :
 - Um controlador **GPIO** adicional para fazer interface com os **botões** na placa Nexys A7
 - Modificação do controlador dos mostradores de 7 segmentos
 - Novos módulos **temporizadores** para utilização dos **LEDs tricolores** existentes na placa
 - Novas **fontes de interrupção externa**

Panorama dos Labs RVfpga

Panorama dos Labs RVfpga

Parte 1: Labs 1-10

- Projeto e Programação com o Vivado
- Sistemas de E/S

Parte 2: Labs 11-20

- Core do RISC-V
- Sistemas de Memória do RISC-V
- Desempenho do RISC-V

Todos os laboratórios incluem **exercícios** de utilização e/ou modificação do Sistema RVfpga para aumentar a compreensão através da concepção prática. O RVfpga inclui programas e soluções de exemplo em C e Assembly

RVfpga Labs 1-4: Programação

- **Lab 1: Programação em C:** Escrever um programa em C no PlatformIO e execute-o / depure-o no RVfpgaNexys/RVfpgaSim/Whisper. Introduce também os pacotes de suporte de placa e de suporte de plataforma (BSP e PSP) da Western Digital para operações de suporte, como a impressão no terminal.
- **Lab 2: Linguagem Assembly RISC-V :** Escrever um programa em Assembly RISC-V no PlatformIO e executá-lo/depurá-lo no RVfpgaNexys/ RVfpgaSim/ Whisper.
- **Lab 3: Chamadas a Funções:** Introdução às chamadas de função, às bibliotecas C e à convenção de chamadas do RISC-V.
- **Lab 4: Processamento de Imagem: C & Assembly:** Incorporar código Assembly com código C.

RVfpga Labs 5-10: E/S & Periféricos

- **Lab 5: Criando um projeto no Vivado:** Criar um projeto no Vivado para implementar o RVfpgaNexys numa placa FPGA e simular o RVfpgaSim no Verilator.
- **Lab 6: Introdução às E/S:** Introdução às E/S mapeadas na memória e ao módulo GPIO de código-aberto do sistema RVfpga.
- **Lab 7: Mostradores de 7 segmentos:** Construir um decodificador de mostrador de 7 segmentos e integrá-lo no sistema RVfpga.
- **Lab 8: Temporizadores:** Compreender e utilizar temporizadores e um controlador de temporizador.
- **Lab 9: E/S Orientadas à Interrupção:** Introdução ao suporte de interrupções do sistema RVfpga e à utilização de E/S acionadas por interrupções.
- **Lab 10: Barramentos Série:** Introdução às interfaces de série (SPI, I2C e UART). Mostrar como utilizar o acelerómetro integrado na placa que utiliza uma interface SPI.

RVfpga Labs 11-20: O Núcleo RISC-V

- **Lab 11:** Compreender a configuração do SweRV EH1, a estrutura do núcleo e a monitorização de desempenho.
- **Labs 12, 13, 16:** Examinar o fluxo de instruções através do pipeline (aritmética/lógica, memória, e saltos condicionais e incondicionais).
- **Labs 14-16:** Compreender os conflitos e como lidar com eles.
- **Lab 16:** Compreender e modificar o preditor de saltos.
- **Lab 17:** Explorar a execução superescalar.
- **Lab 18:** Adição de novas instruções e contadores de hardware.
- **Lab 19:** Compreender a hierarquia da memória e I\$.
- **Lab 20:** Ativar ICCM e DCCM (memórias de instruções e dados estreitamente acopladas) e comparar desempenhos com benchmarks.

Hierarquia das Pastas

Hierarquia das Pastas dos Materiais RVfpga

- *RVfpga/Documents*: Documentos (GSG, Slides, Labs, ...).
 - *LabInstructions*:
 - Instruções para cada laboratório + Figuras utilizadas nas instruções de cada laboratório.
- *RVfpga/verilatorSIM*: Fontes para o simulador Verilator
- *RVfpga/src* folder: Fontes Verilog para o SoC
- *RVfpga/examples*: projetos PlatformIO para os exemplos GSG
- *RVfpga/Labs*:
 - *Lab1,..., Lab20*: Recursos a utilizar durante a realização dos laboratórios.
 - *RVfpgaLabsSolutions*: Soluções de exercícios para os laboratórios.
 - *ProgramsAndDocuments*: Soluções para as tarefas e exercícios propostos.
 - *Modified_RVfpgaSystem*: Sistema RVfpga modificado conforme orientado pelos exercícios dos Labs 6-10 e Lab 18. Soluções para os exercícios + Instruções.

Instalação das Ferramentas RVfpga

Ferramentas de Software RVfpga (Sec. 5 GSG)

- **Vivado IDE da Xilinx**
 - Ver ficheiros fonte RVfpga (Verilog / SystemVerilog) e hierarquia do projeto
 - Criar um bitfile (ficheiro de configuração da FPGA) para o RVfpga na Nexys A7
- **Visual Studio Code (VSCode) + PlatformIO**
 - PlatformIO: uma extensão do VSCode
 - Configurar o sistema RVfpga na placa Nexys A7
 - Compilar, descarregar, executar e depurar programas C e Assembly no sistema RVfpga
- **Verilator** –simulador HDL (linguagem de descrição de hardware)
 - Simular o sistema RVfpga a baixo-nível (HDL) para analisar os seus sinais internos
- **GTKWave** – visualizador de formas de ondas

Esta secção de slides abrange o material das **Secções 2 e 5 do Guia de Iniciação (GSG).**

Instalação mínima de ferramentas (Sec. 2 GSG)

- VSCode
 - Descarregar o VSCode para Linux, Windows ou MacOS
 - Instalar no seu sistema
- Instalar o PlatformIO dentro do VSCode
 - No Linux: Instalar os utilitários python3
 - Ícone de Extensões: Procure por PlatformIO e instale-o
 - Instalar os drivers da Nexys A7 :
 - Linux: Utilizar a pasta fornecida
 - Windows: Utilizar a aplicação Zadig (ver Apêndice)
 - Mac OS: Não é necessário
- Instale o GTKWave seguindo as instruções para seu SO

Exemplos Iniciais

LEDs-Switches – Execução na Placa (Secções 6.A e 6.E do GSG)

- Ligar a placa Nexys A7.
- Abrir o VSCode e o PlatformIO.
- Clique em **File** → **Open Folder** e seleccionar:
[RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang
- Analisar o código fonte do programa: *src/LedsSwitches_C-Lang.c*.
- Na primeira vez que um exemplo RVfpga é aberto no PlatformIO, a plataforma Chips Alliance é instalada automaticamente. Ela inclui as ferramentas RISC-V pré-construídas, o OpenOCD, o simulador Verilator, etc.
- Descarregar o RVfpgaNexys para a placa Nexys A7. Primeiro, é necessário atualizar o caminho. Pode ser necessário atualizar a janela Project Tasks.
- Descarregar e executar o exemplo LEDs-Switches.

Esta secção de slides cobre o material das **Secções 6-8 do Guia de Iniciação (GSG)**.

Exemplo AL_Operations – Execução na Placa (Sec. 6.B do GSG)

- Se ainda não estiver aberto, abra o VSCode e o PlatformIO
- Clique em **File** → **Close Folder** no menu ficheiro no topo. Em seguida, clique em **File** → **Open Folder** e selecione:


[RVfpgaPath]\RVfpga\examples\AL_Operations

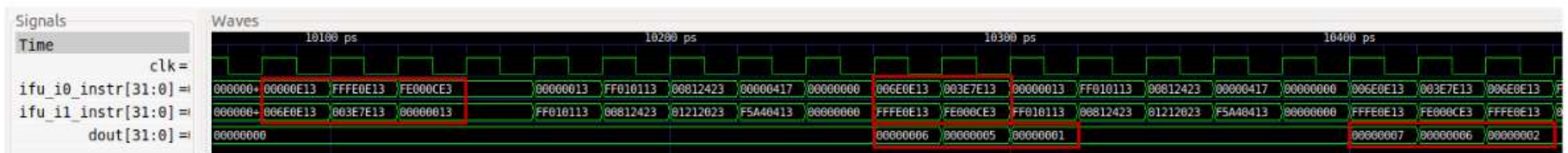
- Analisar o código fonte do programa
- Se necessário, configure o RVfpgaNexys na placa Nexys A7.
- Descarregar, executar e depurar exemplos de AL_Operations.

Exemplo AL_Operations – Simulação no Whisper (Sec. 8 do GSG)

- Clique em *File* → *Open File* e faça duplo clique em *[RVfpgaPath]/RVfpga/examples/AL_Operations/platformio.ini*, e defina **whisper** como a ferramenta de depuração, descomentando a linha 17.
- Lançar o depurador como habitualmente
- Agora pode depurar o programa exactamente como fez na Secção 6.B, mas desta vez o programa está a ser executado em simulação no Whisper em vez de na placa FPGA Nexys A7.

Exemplo AL_Operations – Simulação no Verilator (Sec. 7 do GSG)

- Abrir o ficheiro *platformio.ini*. Definir o caminho para o RVfpgaSim.
- Executar a simulação clicando no ícone PlatformIO 
- Expandir Project Tasks → env:swervolf_nexys → Platform e clicar em Generate Trace
- Alguns segundos após o passo anterior, o ficheiro *trace.vcd* deve ter sido gerado e pode abri-lo com o GTKWave
- Adicionar sinais: clique em *File – Read Tcl Script File* e seleccionar *[RVfpgaPath]/RVfpga/examples/AL_Operations/test.tcl*



Exemplo HelloWorld (Sec. 6.F do GSG)

1. Abra o VSCode e o PlatformIO. Clique em **File → Close Folder**. Clique em **File → Open Folder** no menu de ficheiro superior e seleccionar:

```
[RVfpgaPath]\RVfpga\examples\HelloWorld_C-Lang
```
2. Configurar o sistema:
 - Monitor série do PlatformIO: Usar o parâmetro *monitor_speed* no ficheiro *platformio.ini*
 - Em Linux, adicione-se aos grupos *dialout*, *tty* e *uucp*
3. Descarregar e executar o exemplo HelloWorld. Quando o programa começar a ser executado, abra o monitor de série, clicando no botão de ligação disponível na parte inferior do VS Code.

HelloWorld – Simulação no Whisper (Sec. 8 do GSG)

- Clique em *File* → *Open File* e fazer duplo clique em *[RVfpgaPath]/RVfpga/examples/HelloWorld_C-Lang/platformio.ini*, e definir o **whisper** como a ferramenta de depuração descomentando a linha 17.
- Inicie o depurador como habitualmente. Pode agora depurar o programa exactamente como fez na Secção 6.B, mas desta vez o programa está a ser executado em simulação no Whisper em vez de na placa FPGA Nexys A7.
- Dado que este programa usa a função *printfNexys* no Whisper, não deve abrir o monitor de série PlatformIO, uma vez que as mensagens são mostradas na consola DEBUG.

Lab 1:

Programação em C

RVfpga Lab 1: Programação em C

- Criar um **projeto no PlatformIO**
- Adicionar o **programa em C** de exemplo ao projeto
- **Configurar o RVfppgaNexys** na placa Nexys A7
- **Descarregar o programa em C** para o RVfpgaNexys e executar/depurar o programa
- Completar alguns ou todos os **exercícios** no final do Lab
- Lembre-se que também pode simular o programa em Verilator (com o **RVfpgaSim**) ou **Whisper**.

RVfpga Lab 1: Exemplo de Programa em C

```
// Endereços de E/S mapeados em memória
```

```
#define GPIO_SWs      0x80001400
```

```
#define GPIO_LEDs     0x80001404
```

```
#define GPIO_INOUT    0x80001408
```

Este programa escreve o valor dos interruptores nos LEDs.

```
#define READ_GPIO(dir) (*(volatile unsigned *)dir)
```

```
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }
```

```
int main ( void )
```

```
{
```

```
    int En_Value=0xFFFF, switches_value;
```

```
    WRITE_GPIO(GPIO_INOUT, En_Value);
```

```
    while (1) {
```

```
        switches_value = READ_GPIO(GPIO_SWs);    // ler valor nos interruptores
```

```
        switches_value = switches_value >> 16;    // deslocar para 16 bits inferiores
```

```
        WRITE_GPIO(GPIO_LEDs, switches_value);    // mostrar o valor dos interruptores nos LEDs
```

```
    }
```

```
    return(0);
```

```
}
```

RVfpga Lab 1: Endereços de E/S Mapeados em Memória

Dispositivo	Endereço de E/S mapeado em memória
Interruptores (16 na placa Nexys A7)	0x80001400 (16 bits superiores)
LEDs (16 na placa Nexys A7)	0x80001404 (16 bits inferiores)
Entrada/Saída do GPIO (1 = saída, 0 = entrada)	0x80001408

RVfpga Lab 1: BSP & PSP da Western Digital

- A Western Digital fornece:
 - **PSP:** *Processor Support Package* - pacote de apoio ao processador
 - **BSP:** *Board Support Package* - pacote de apoio à placa
- Estes fornecem funções comuns para um determinado processador (core SweRV EH1) e a placa FPGA (Nexys A7).
 - **Exemplo:** `printfNexys` (tal como a função `printf` em C)

RVfpga Lab 1: escrever num terminal via UART

```
#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>
#define DELAY 10000000

int main(void) {
    int i, j = 0;

    // Initialize UART
    uartInit();
    while (1) {
        printfNexys("Hello RVfpga users! Iteration: %d\n", j);
        for (i=0; i < DELAY; i++) ; // delay between printf's
        j++;
    }
}
```

- Adicione esta linha ao ficheiro **platform.ini** :
monitor_speed = 115200
- Depois do programa começar a correr, **abrir o terminal do PlatformIO** premindo este botão na parte inferior da janela:



RVfpga Lab 1: Exemplos de exercícios – E/S

- **Exercício 1.** Escreva um programa em C que faça piscar o valor dos interruptores nos LEDs. O valor deve ser ativado e desativado de forma suficientemente lenta para que uma pessoa possa ver a intermitência.
- **Exercício 2.** Escreva um programa em C que mostre o valor inverso dos interruptores nos LEDs. Por exemplo, se os interruptores forem (em binário): 010101010101010101, então os LEDs devem mostrar: 101010101010101010; se os interruptores forem: 11110000111110000, então os LEDs devem exibir : 0000111100001111...
- **Exercício 4.** Escreva um programa em C que mostre a adição sem sinal de 4 bits dos 4 bits menos significativos dos interruptores e dos 4 bits mais significativos dos interruptores. Mostre o resultado nos 4 bits menos significativos (mais à direita) dos LEDs. O quinto bit dos LEDs deve acender-se quando ocorre um overflow sem sinal (ou seja, quando o carry out é 1).

RVfpga Lab 1: Exemplos de exercícios – Algoritmos

- **Exercício 5.** Escreva um programa em C que encontre o maior divisor comum de dois números, a e b , de acordo com o algoritmo de Euclides. Os valores a e b devem ser variáveis definidas estaticamente no programa.
- **Exercício 9.** Implemente o algoritmo *bubble sort* em C. Este algoritmo ordena os elementos de um vector por ordem crescente através do seguinte procedimento:
 1. Percorrer o vector repetidamente até terminar.
 2. Trocar qualquer par de elementos adjacentes se $V(i) > V(i+1)$.
 3. O algoritmo pára quando todos os pares de elementos consecutivos estão em ordem.
- **Exercício 10.** Escreva um programa em C que calcule o factorial de um dado número não negativo, n , através de multiplicações iterativas. Embora deva testar o seu programa para vários valores de n , a sua apresentação final deve ser para $n = 7$. O programa deve imprimir o valor de $\text{factorial}(n)$ no final do programa. n deve ser uma variável definida estaticamente no programa.

Lab 2:

Assembly RISC-V

RVfpga Lab 2: Assembly RISC-V

- Crie um **projeto no PlatformIO** de raiz
- Adicionar **exemplo de programa de Assembly RISC-V** ao projeto
 - LedsSwitches
- **Executar e depurar** o programa:
 - Na placa Nexys A7
 - No Whisper
- Realize os exercícios no final do lab
- Crie um projeto de raiz

RVfpga Lab 3: Instruções do Assembly RISC-V

Instruções/Pseudoinstruções comuns do Assembly RISC-V

Assembly RISC-V	Descrição	Operação
add s0, s1, s2	Adição	$s0 = s1 + s2$
sub s0, s1, s2	Subtração	$s0 = s1 - s2$
addi t3, t1, -10	Adição com constante	$t3 = t1 - 10$
mul t0, t2, t3	Multiplicação a 32 bits	$t0 = t2 * t3$
div s9, t5, t6	Divisão	$t9 = t5 / t6$
rem s4, s1, s2	Resto (ou módulo)	$s4 = s1 \% s2$
and t0, t1, t2	AND bit-a-bit	$t0 = t1 \& t2$
or t0, t1, t5	OR bit-a-bit	$t0 = t1 t5$
xor s3, s4, s5	XOR bit-a-bit	$s3 = s4 \wedge s5$
andi t1, t2, 0xFFB	AND bit-a-bit com constante	$t1 = t2 \& 0xFFFFFBB$
ori t0, t1, 0x2C	OR bit-a-bit com constante	$t0 = t1 0x2C$
xori s3, s4, 0xABC	XOR bit-a-bit com constante	$s3 = s4 \wedge 0xFFFFFABC$
sll t0, t1, t2	Deslocamento lógico para a esquerda	$t0 = t1 \ll t2$
srl t0, t1, t5	Deslocamento lógico para a direita	$t0 = t1 \gg t5$
sra s3, s4, s5	Deslocamento aritmético para a direita	$s3 = s4 \ggg s5$
slli t1, t2, 30	Deslocação lógica à esquerda com constante	$t1 = t2 \ll 30$
srli t0, t1, 5	Deslocamento lógico para a direita com constante	$t0 = t1 \gg 5$
srai s3, s4, 31	Deslocamento aritmético para a direita com constante	$s3 = s4 \ggg 31$

RVfpga Lab 3: Instruções do Assembly RISC-V

Instruções/Pseudoinstruções comuns do Assembly RISC-V (continuação)

Assembly RISC-V	Descrição	Operação
lw s7, 0x2C(t1)	Carregar palavra	$s7 = \text{memory}[t1+0x2C]$
lh s5, 0x5A(s3)	Carregar meia-palavra	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
lb s1, -3(t4)	Carregar byte	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
sw t2, 0x7C(t1)	Guardar palavra	$\text{memory}[t1+0x7C] = t2$
sh t3, 22(s3)	Guardar meia-palavra	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
sb t4, 5(s4)	Guardar byte	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
beq s1, s2, L1	Saltar se igual	if $(s1 == s2)$, PC = L1
bne t3, t4, Loop	Saltar se diferente	if $(s1 \neq s2)$, PC = Loop
blt t4, t5, L3	Saltar se for menor	if $(t4 < t5)$, PC = L3
bge s8, s9, Done	Saltar se for maior ou igual	if $(s8 \geq s9)$, PC = Done
li s1, 0xABCDEF12	Carregar constante	$s1 = 0xABCDEF12$
la s1, A	Carregar endereço	$s1 = \text{Memory address where variable A is stored}$
nop	Instrução nula (Nop)	no operation
mv s3, s7	Copiar valor de registo	$s3 = s7$
not t1, t2	Negação bit (Inverter bit-a-bit)	$t1 = \sim t2$
neg s1, s3	Negar	$s1 = -s3$
j Label	Saltar	PC = Label
jal L7	Saltar e guardar endereço de retorno	PC = L7; ra = PC + 4
jr s1	Saltar para o endereço no registo	PC = s1

RVfpga Lab 3: Registos RISC-V

32 registos de 32 bits

Nome	Número de Registo	Uso
zero	x0	Valor constante 0
ra	x1	Endereço de retorno
sp	x2	Ponteiro da pilha
gp	x3	Ponteiro global
tp	x4	Ponteiro de <i>thread</i>
t0-2	x5-7	Variáveis temporárias
s0/fp	x8	Variável preservada/Ponteiro da <i>frame</i>
s1	x9	Variável preservada
a0-1	x10-11	Argumentos de função/Valores de retorno
a2-7	x12-17	Argumentos de função
s2-11	x18-27	Variável preservada
t3-6	x28-31	Variável temporária

RVfpga Lab 2: Exemplo de programa Assembly RISC-V

- // Endereços de E/S mapeados em memória
- # GPIO_SWs = 0x80001400
- # GPIO_LEDs = 0x80001404
- # GPIO_INOUT = 0x80001408

Este programa escreve o valor dos interruptores nos LEDs.

- .globl main
- main:

Localização da pasta: *[RVfpgaPath]\RVfpga\Labs\Lab02*

- main:
- li t0, 0x80001400 # endereço base dos registos de GPIO mapeados em memória
- li t1, 0xFFFF # definir direcção dos GPIOs
- # metade superior = interruptores (entradas) (=0)
- # metade inferior = saídas (LEDs) (=1)
- sw t1, 8(t0) # GPIO_INOUT = 0xFFFF
-
- repeat:
- lw t1, 0(t0) # ler interruptores: t1 = GPIO_SWs
- srli t1, t1, 16 # desloca val para a direita em 16 bits
- sw t1, 4(t0) # escreve valor para os LEDs: GPIO_LEDs = t1
- j repeat # repete o ciclo

RVfpga Lab 2: Os mesmos exercícios do laboratório 1 - Exemplo

- **Exercício 4.** Escreva um programa em C que apresente a soma de 4 bits sem sinal dos 4 bits menos significativos dos interruptores e dos 4 bits mais significativos dos interruptores. Mostre o resultado nos 4 bits menos significativos (mais à direita) dos LEDs. O quinto bit dos LEDs deve acender-se quando ocorre um overflow sem sinal (ou seja, quando o carry out é 1).

Lab 3:

Chamadas a

Funções

RVfpga Lab 3: Chamadas a Funções

- Escrever programas em C com **chamadas a funções** (procedimentos)
- Escrever programas em C com chamadas a funções em bibliotecas:
 - Usando bibliotecas standard
 - Usando as bibliotecas da WD, específicas para o RVfpga
- **Convenção de chamada de funções** (procedimentos) no RISC-V

RVfpga Lab 3: Exemplo de Programa com Funções

```
// Endereços de E/S mapeados em memória
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408
#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main ( void ) {
    unsigned int switches_val;

    IOsetup();
    while (1) {
        switches_val = getSwitchVal();
        writeValtoLEDs(switches_val);
    }

    return(0);
}
```

RVfpga Lab 3: Exemplo de Programa com Funções

```
void IOsetup()
{
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal()
{
    unsigned int val;

    val = READ_GPIO(GPIO_SWs);    // ler valor dos interruptores
    val = val >> 16;               // deslocar para os 16 bits menos significativos

    return val;
}

void writeValtoLEDs(unsigned int val)
{
    WRITE_GPIO(GPIO_LEDs, val);    // apresentar val nos LEDs
}
```

RVfpga Lab 3: Bibliotecas C

- **Bibliotecas**
 - Recolha de funções de uso corrente
 - Providenciado para que as funções comuns estejam disponíveis prontamente (poupar tempo de programação)
- **Exemplo de bibliotecas C:**
 - **math.h** (biblioteca matemática): inclui funções tais como sqrt (raiz quadrada), cos (cosseno), etc.
 - **stdio.h** (biblioteca de E/S comum): inclui funções para fazer a impressão de valores no ecrã (printf), leitura de valores dos utilizadores (scanf), etc.
 - **stdlib.h** (biblioteca standard): inclui funções para gerar números aleatórios (rand), e conversão de dados (atoi,atof,atol,strtod,strtol).
 - **Muitas outras...** (pesquisar “C libraries”)

RVfpga Lab 3: Exemplo de programa com Biblioteca C

```
#include <stdlib.h>
```

```
...
```

```
int main(void) {  
    unsigned int val;  
    volatile unsigned int i;  
  
    IOsetup();  
    while (1) {  
        val = rand() % 65536;  
        writeValtoLEDs(val);  
        for (i = 0; i < DELAY; i++)  
            ;  
    }  
    return(0);  
}
```

Este programa escreve um número aleatório entre 0 e 65535 para os LEDs.

RVfpga Lab 3: Bibliotecas WD

- Já usámos a **printfNexys** em muitos programas
- Lab 9: Utilização de funções WD para o tratamento de interrupções

RVfpga Lab 3: Convenção de Chamadas em RISC-V

- **Chamar uma função**

`jal function_label`

- **Retorno de uma função**

`jr ra`

- **Argumentos**

- colocados nos registros `a0–a7`

- **Valor de retorno**

- colocado no registro `a0`

RVfpga Lab 3: Exemplo da Convenção de Chamadas em RISC-V

Código C

```
int main() {  
    ...  
    int y = y + func1(1, 2, 3)  
    y++;  
    ...  
}  
  
int func1(int a, int b, int c) {  
    int sum;  
    sum = a + b + c;  
    return sum;  
}
```

Assembly RISC-V

```
# y está em s0  
main:  
    ...  
    addi a0, zero, 1 #valores em registos de argumentos  
    addi a1, zero, 2  
    addi a2, zero, 3  
    jal  func1        # chamar função func1  
    add  s0, s0, a0    # y = y + valor de retorno  
    addi s0, s0, 1     # y = y++  
    ...  
  
# sum está em s0  
func1:  
    add s0, a0, a1     # sum = a + b  
    add s0, s0, a2     # sum = a + b + c  
    addi a0, s0, 0     # valor de retorno = sum  
    jr   ra            # retornar
```

RVfpga Lab 3: A Pilha (Stack)

- **Espaço de rascunho** em memória utilizado para guardar valores de registo
- O Stack Pointer (sp) contém o endereço do topo da pilha
- A **pilha cresce para baixo** na memória. Assim, por exemplo, para criar espaço para 4 palavras (16 bytes) na pilha é utilizado o seguinte código:

```
addi sp, sp, -16
```

- **Duas categorias de registos:**
 - **Registos preservados:** o conteúdo do registo **deve ser preservado** entre chamadas de funções (i.e., contêm o mesmo valor antes e depois de uma chamada de função)
 - **Registos não preservados:** o conteúdo do registo não deve ser preservado nas chamadas de função (ou seja, o registo não precisa de ser o mesmo antes e depois de uma chamada da função)
 - Registos salvos ($s0-s11$), o registo do endereço de retorno (ra), e o ponteiro da pilha (sp) são registos **preservados**. Todos os outros registos não são preservados.

RVfpga Lab 4: Registos Preservados/Não Preservados

Nome	Número de Registo	Uso	Preservado
zero	x0	Valor constante 0	-
ra	x1	Endereço de retorno	Sim
sp	x2	Ponteiro da pilha	Sim
gp	x3	Ponteiro global	-
tp	x4	Ponteiro de <i>thread</i>	-
t0-2	x5-7	Variáveis temporárias	Não
s0/fp	x8	Variável preservada/Ponteiro da <i>frame</i>	Sim
s1	x9	Variável preservada	Sim
a0-1	x10-11	Argumentos de função/Valores de retorno	Não
a2-7	x12-17	Argumentos de função	Não
s2-11	x18-27	Variável preservada	Sim
t3-6	x28-31	Variável temporária	Não

RVfpga Lab 4: A Pilha – Código Assembly Revisto

Código C

```
int main() {  
    ...  
    int y = y + func1(1, 2, 3)  
    y++;  
    ...  
}  
  
int func1(int a, int b, int c) {  
    int sum;  
  
    sum = a + b + c;  
    return sum;  
}
```

Assembly RISC-V

```
# y is in s0  
main: ...  
    addi a0, zero, 1 # por valores nos registos de args  
    addi a1, zero, 2  
    addi a2, zero, 3  
    jal  func1        # função de chamada func1  
    add  s0, s0, a0    # y = y + valor de retorno  
    addi s0, s0, 1     # y = y++  
    ...  
  
# sum is in s0  
func1: addi sp, sp, -4 # reservar espaço na pilha  
       sw  s0, 0(sp) # guardar s0 na pilha  
    add s0, a0, a1 # sum = a + b  
    add s0, s0, a2 # sum = a + b + c  
    addi a0, s0, 0 # valor de retorno = sum  
       lw  s0, 0(sp) # recuperar s0 da pilha  
       addi sp, sp, 4 # recuperar o ponteiro da pilha  
    jr  ra        # retornar
```

RVfpga Lab 3: Exemplos de exercícios - Programas em C

- **Exercício 3.** Escreva um programa em C que meça o tempo de reacção. O seu programa deve medir o tempo que uma pessoa demora a ligar o interruptor mais à direita (SW[0]) depois de todos os LEDs se acenderem. Utilize a função rand() da biblioteca stdlib.h para gerar um período de tempo aleatório entre cada tentativa do utilizador para testar o seu tempo de reacção.

RVfpga Lab 3: Exemplos de exercícios - Programas em C

- **Exercício 8.** Escreva um programa RISC-V em assembly chamado Filter.S (o programa deve estar em conformidade com o modelo de gestão de funções estudado anteriormente). Pode utilizar o seguinte pseudo-código:

```
#define N 6
int i, j=0, A[N]={48,64,56,80,96,48}, B[N];
for (i=0; i<(N-1); i++){
    if( (myFilter(A[i],A[i+1])) == 1){
        B[j]=A[i]+ A[i+1] + 2;
        j++;
    }
}
```

- Escreva o código Assembly RISC-V equivalente, incluindo quaisquer directivas necessárias para reservar espaço de memória e declarando as secções correspondentes (.data, .bss e .text). A função myFilter retorna o valor 1 se o primeiro argumento for múltiplo de 16 e o segundo for maior que o primeiro; caso contrário, retorna 0.
- Escreva o código Assembly da função myFilter.

Lab 4:

C e Assembly

RVfpga Lab 4: Combinando C e Assembly

- **Exemplo:** Programa de processamento de imagem
- Algumas funções escritas em C e outras em Assembly
- Converter uma imagem a cores noutra em escala de cinza



RVfpga Lab 4: Programa de Processamento de Imagem

- Cada pixel é armazenado como três cores com 8 bits : **R** = vermelho (red), **G** = verde (green), **B** = azul (blue)
- Qualquer cor pode ser criada através da **variação dos valores R, G, e B**
- Para converter a imagem para uma escala de cinzentos de 8 bits (**grey**), cada pixel é transformado da seguinte forma :

$$\text{grey} = (306 * \text{R} + 601 * \text{G} + 117 * \text{B}) \gg 10$$

- Os pesos RGB somam até 1024 ($306 + 601 + 117 = 1024$), para voltar à gama de 8 bits (0-255), o resultado é dividido por 1024 (i.e., deslocado para a direita por 10 bits : $\gg 10$)
- Para mais detalhes sobre o algoritmo, ver:

<https://www.mathworks.com/help/matlab/ref/rgb2gray.html>

RVfpga Lab 4: Funções em Assembly

`.globl ColourToGrey_Pixel` ← `.globl` torna a função `ColourToGrey_Pixel` visível a todos os ficheiros no projeto

`ColourToGrey_Pixel:`

```
    li x28, 306          # a0 = R * 306
    mul a0, a0, x28
    li x28, 601          # a1 = G * 601
    mul a1, a1, x28
    li x28, 117          # a2 = B * 117
    mul a2, a2, x28
    add a0, a0, a1        # grey = a0 + a1 + a2
    add a0, a0, a2
    srl a0, a0, 10        # grey = grey / 1024
    ret                  # return
.end
```

`grey = (306*R + 601*G + 117*B) >> 10`

RVfpga Lab 4: Estruturas (Structs) e Arrays

```
typedef struct {
    unsigned char R;
    unsigned char G;
    unsigned char B;
} RGB;

extern unsigned char VanGogh_128x128[]; // Array 1D com valores individuais RGB
RGB ColourImage[N][M]; // Array 2D da estrutura RGB (imagem a cores)
unsigned char GreyImage[N][M]; // Array 2D da Imagem em escala de cinzentos

// VanGogh_128.c
unsigned char VanGogh_128x128[] = {    157, // R (pixel [0][0])
                                       182, // G (pixel [0][0])
                                       161, // B (pixel [0][0])
                                       171, // R (pixel [0][1])
                                       195, // G (pixel [0][1])
                                       173, // B (pixel [0][1])
                                       173, // R (pixel [0][2])
                                       ...
                                       }
```

RVfpga Lab 4: Função Principal (Main)

```
int main(void) {
    // Criar uma matriz N x M usando a imagem de entrada
    initColourImage(ColourImage);

    // Transformar imagem colorida em imagem cinzenta
    ColourToGrey(ColourImage, GreyImage);
    ...
}

void ColourToGrey(RGB Colour[N][M], unsigned char Grey[N][M]) {
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<M; j++)
            Grey[i][j] = ColourToGrey_Pixel(Colour[i][j].R, Colour[i][j].G,
                                             Colour[i][j].B);
}
```

RVfpga Lab 4: Projeto e Exercícios Fornecidos

- **Exercício 1.** Executar o programa numa imagem de entrada diferente.
- **Exercício 2.** Crie uma função C que conte o número de pixels próximos do branco (>235) e próximos do preto (<20) na imagem em escala de cinzentos de VanGogh. Imprima os dois números na consola série.
- **Exercício 3.** Transformar a sub-rotina Assembly **ColourToGrey_Pixel** numa função C, e a função C **ColourToGrey** numa sub-rotina Assembly que invoca a função em C **ColourToGrey_Pixel**.
- **Exercício 4.** Aplicar um filtro de desfocagem (*Blur*) à imagem a cores de VanGogh.

Lab 5:

Projeto em

Vivado

RVfpga Lab 5: Projeto RVfpga em Vivado

- **Vivado** é uma ferramenta da Xilinx para visualizar, modificar e sintetizar o código fonte (Verilog) do sistema RVfpga.
- O código fonte do sistema RVfpga está em :
[RVfpgaPath]/RVfpga/src
- Neste laboratório, os utilizadores criam um **projeto em Vivado** que contém o código-fonte do sistema RVfpga, sintetize o RVfpgaNexys destinado à placa Nexys A7 e crie um **bitfile** que contém informações para configurar a FPGA como RVfpgaNexys.
- O Vivado (e o Verilator) são utilizados extensivamente nos **Labs 6-20** para modificar e simular o sistema RVfpga.

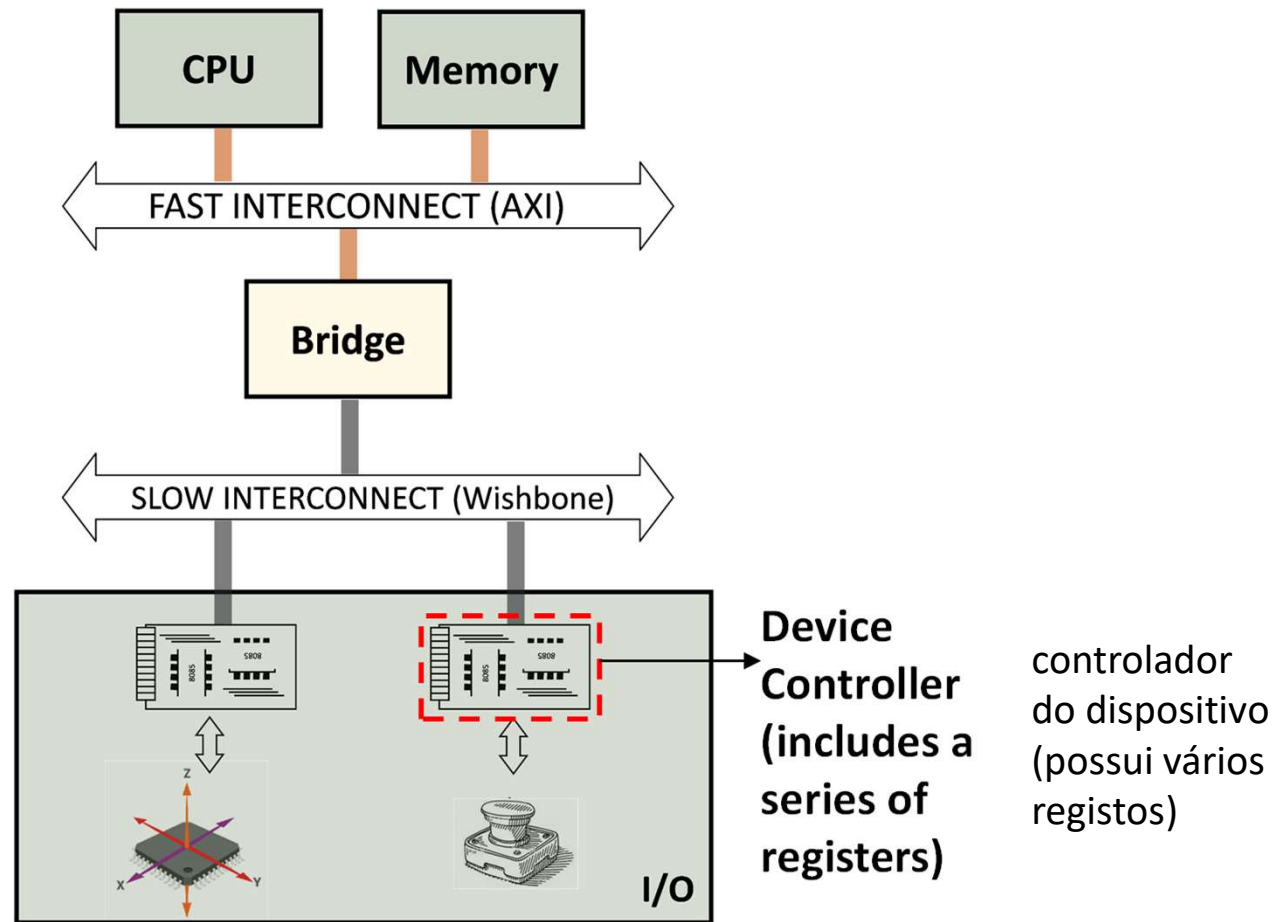
Lab 6:

Introdução às E/S

RVfpga Lab 6: Introdução às E/S

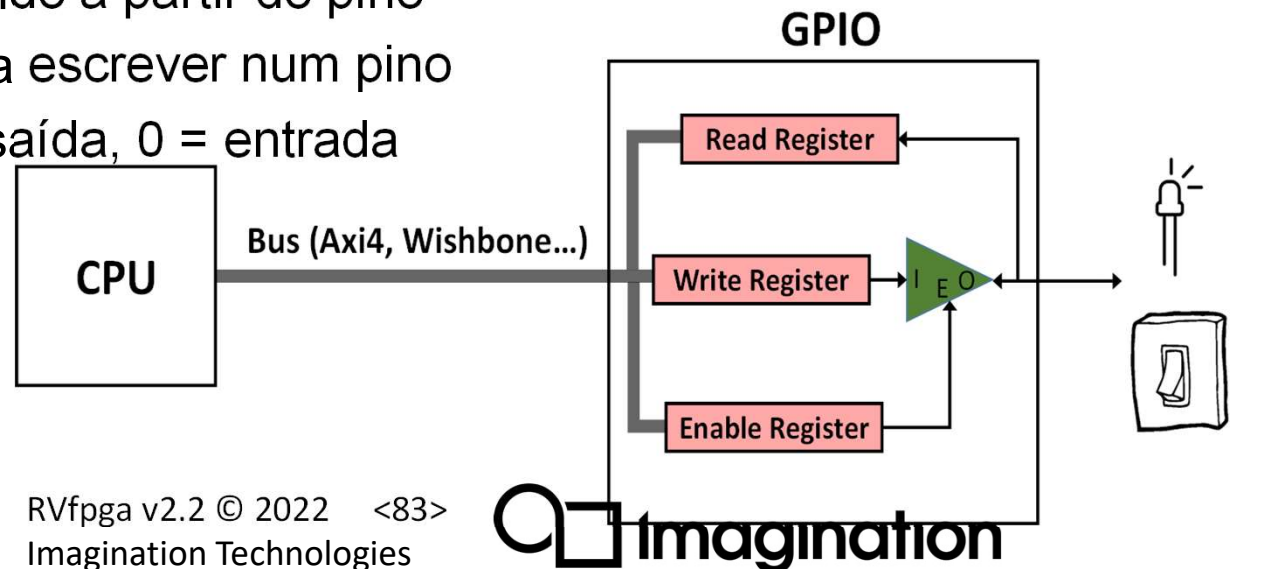
- Principais características de um sistema de E/S de uso geral e do sistema utilizado no sistema RVfpga
- Versão teórica simplificada de um controlador GPIO genérico
- Controlador GPIO utilizado no SoC SweRVolfX:
 - Analisamos primeiro a sua especificação de alto nível e introduzimos exercícios elementares
 - Em seguida, analisamos a sua implementação de baixo nível, simulando o RVfpgaSim no Verilator e introduzindo exercícios avançados

RVfpga Lab 6: Processador Genérico com E/S



RVfpga Lab 6: E/S de uso geral (GPIO)

- **E/S de uso geral :**
 - Permite ao processador ler/escrever em pinos ligados a periféricos (interruptores e LEDs)
 - Cada pino pode ser configurado como entrada ou saída usando um *buffer* de três estados (*tri-state*)
- **Três registros mapeados em memória :**
 - **Registro de Leitura:** valor lido a partir do pino
 - **Registro de Escrita:** valor a escrever num pino
 - **Registro de ativação:** 1 = saída, 0 = entrada



RVfpga Lab 6: Módulo GPIO SweRVolfX

- **Módulo GPIO do OpenCores**

<https://opencores.org/projects/gpio>

- **Permite até 32 pinos de GPIO**

- Todos os pinos podem ser configurados individualmente como entradas (enable = 0) ou saídas (enable = 1)
- A configuração pode mudar ao longo do programa

Registro	Endereço Mapeado em Memória
Registro de Leitura	0x80001400
Registro de Escrita	0x80001404
Registro de Ativação	0x80001408

RVfpga Lab 6: Registos Mapeados em Memória

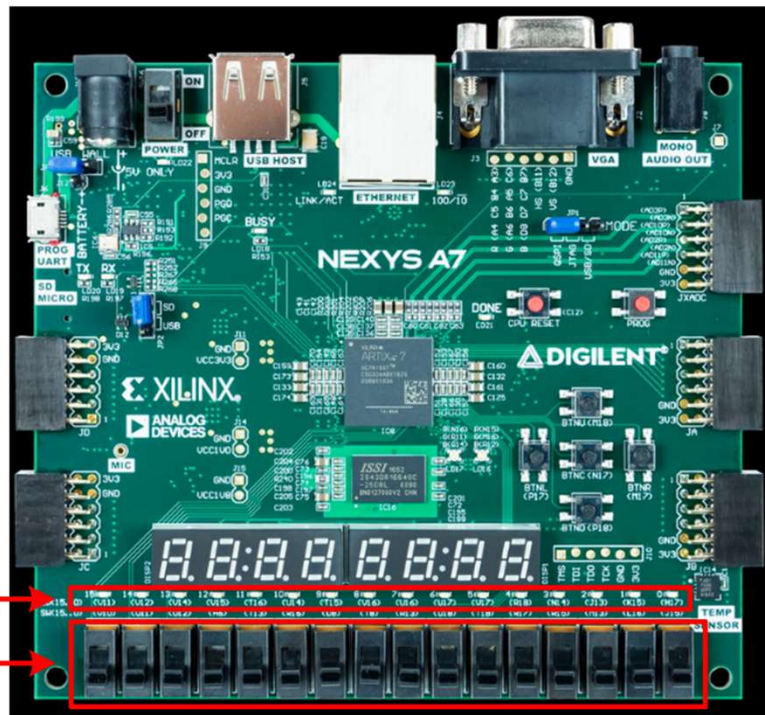


Figura da placa de <https://reference.digilentinc.com/>

Maapeamento de LEDs e interruptores para pinos GPIO:

- LEDs: pinos [15:0] (saídas do processador)
- Interruptores: pinos [31:16] (entradas para o processador)

Configure o GPIO :

- Registo de ativação = 0x0000FFFF (1 = saída, 0 = entrada)

```
li t0, 0x80001400  
li t1, 0xFFFF  
sw t1, 8(t0) # Enable Register = 0xFFFF
```

Escrita nos LEDs:

- Escrever valor em [15:0] no endereço 0x80001404

```
sw t3, 4(t0) # LEDs = [t3]15:0
```

Leitura dos interruptores:

- Ler os interruptores nos bits [31:16] do endereço 0x80001400
- Deslocar 16 bits para a direita para colocar o valor nos 16 bits mais baixos

```
lw t5, 0(t0) # [t5]31:16 = valor dos interruptores  
srli t5, t5, 16 # [t5]15:0 = valor dos interruptores
```

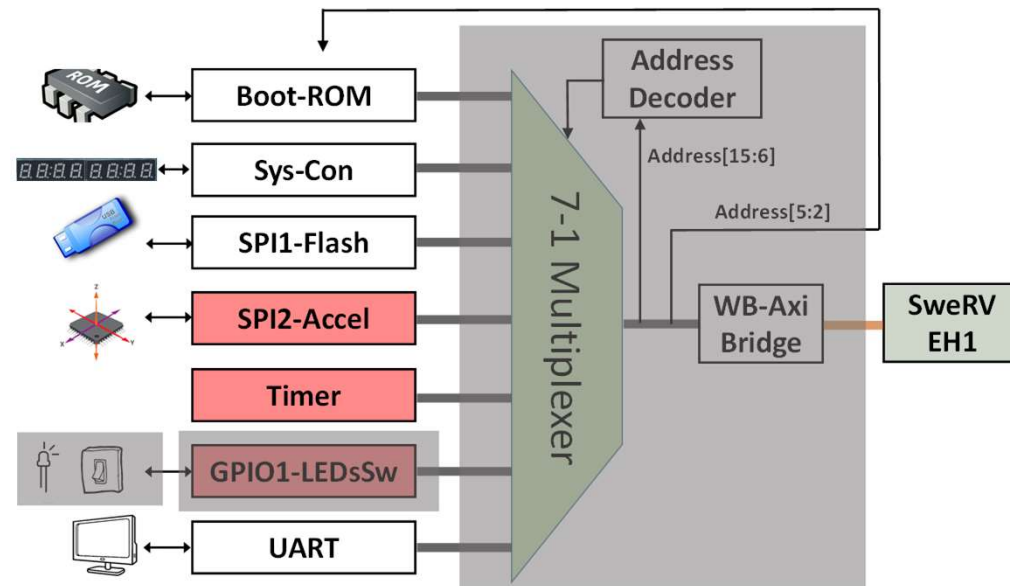
RVfpga Lab 6: Exercícios elementares - Exemplo

- **Exercício 1.** Escreva um programa em Assembly RISC-V e um programa em C que mostre um bloco de quatro LEDs acesos que se move repetidamente de um lado para o outro nos 16 LEDs disponíveis na placa. Inclua também dois interruptores que controlam a velocidade e a direção. O Switch[0] altera a velocidade e o Switch[1] altera a direção tal que:
 - Se o Switch[0] estiver ON (nível alto), os LEDs acesos devem mover rapidamente. Caso contrário, os LEDs acesos devem mover lentamente. Pode definir o que é "rápido" e "lento", mas qualquer uma das velocidades deve ser visível e a diferença na velocidade detectada olhando para os LEDs.
 - Se o Switch[1] estiver ON, os LEDs acesos devem mover repetidamente da direita para a esquerda (voltam à direita quando alcançam o LED mais à esquerda). Caso contrário, os LEDs acesos devem mover repetidamente da esquerda para a direita.

RVfpga Lab 6: Implementação de Baixo Nível do GPIO

- **Dividido em 3 partes principais**

- Ligação externa do RVfppgaNexys aos LEDs/Switches da placa (região sombreada à esquerda)
- Integração do módulo GPIO no SweRVolfX (região sombreada no meio)
- Ligação entre a GPIO e a SweRV EH1 (região sombreada à direita)



RVfpga Lab 6: Ligação Externa

Ficheiro **rvfpganexys.xdc**: Define a ligação de `i_sw[15:0]` com os interruptores da placa e `o_led[15:0]` com os LEDs da placa

```
26 set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[0] }]
27 set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[1] }]
28 set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[2] }]
29 set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[3] }]
30 set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { i_sw[4] }]
31 set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[5] }]
32 set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[6] }]
33 set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[7] }]
34 set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[8] }]
35 set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[9] }]
36 set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[10] }]
37 set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[11] }]
38 set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { i_sw[12] }]
39 set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { i_sw[13] }]
40 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { i_sw[14] }]
41 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { i_sw[15] }]
42
43 set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { o_led[0] }]
44 set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { o_led[1] }]
45 set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { o_led[2] }]
46 set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { o_led[3] }]
47 set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { o_led[4] }]
48 set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { o_led[5] }]
49 set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { o_led[6] }]
50 set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { o_led[7] }]
51 set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { o_led[8] }]
52 set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { o_led[9] }]
53 set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { o_led[10] }]
54 set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { o_led[11] }]
55 set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { o_led[12] }]
56 set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { o_led[13] }]
57 set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { o_led[14] }]
58 set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { o_led[15] }]
```

RVfpga Lab 6: Integration into RVfpga

Ficheiro **swervolf_core.v**: Instanciação do módulo GPIO e *buffers* de três estados

```
bidirec gpio0 (.oe(en_gpio[0]), .inp(o_gpio[0]), .outp(i_gpio[0]), .bidir(io_data[0]));
bidirec gpio1 (.oe(en_gpio[1]), .inp(o_gpio[1]), .outp(i_gpio[1]), .bidir(io_data[1]));
bidirec gpio2 (.oe(en_gpio[2]), .inp(o_gpio[2]), .outp(i_gpio[2]), .bidir(io_data[2]));
bidirec gpio3 (.oe(en_gpio[3]), .inp(o_gpio[3]), .outp(i_gpio[3]), .bidir(io_data[3]));
bidirec gpio4 (.oe(en_gpio[4]), .inp(o_gpio[4]), .outp(i_gpio[4]), .bidir(io_data[4]));
bidirec gpio5 (.oe(en_gpio[5]), .inp(o_gpio[5]), .outp(i_gpio[5]), .bidir(io_data[5]));
bidirec gpio6 (.oe(en_gpio[6]), .inp(o_gpio[6]), .outp(i_gpio[6]), .bidir(io_data[6]));
bidirec gpio7 (.oe(en_gpio[7]), .inp(o_gpio[7]), .outp(i_gpio[7]), .bidir(io_data[7]));
bidirec gpio8 (.oe(en_gpio[8]), .inp(o_gpio[8]), .outp(i_gpio[8]), .bidir(io_data[8]));
bidirec gpio9 (.oe(en_gpio[9]), .inp(o_gpio[9]), .outp(i_gpio[9]), .bidir(io_data[9]));
bidirec gpio10 (.oe(en_gpio[10]), .inp(o_gpio[10]), .outp(i_gpio[10]), .bidir(io_data[10]));
bidirec gpio11 (.oe(en_gpio[11]), .inp(o_gpio[11]), .outp(i_gpio[11]), .bidir(io_data[11]));
bidirec gpio12 (.oe(en_gpio[12]), .inp(o_gpio[12]), .outp(i_gpio[12]), .bidir(io_data[12]));
bidirec gpio13 (.oe(en_gpio[13]), .inp(o_gpio[13]), .outp(i_gpio[13]), .bidir(io_data[13]));
bidirec gpio14 (.oe(en_gpio[14]), .inp(o_gpio[14]), .outp(i_gpio[14]), .bidir(io_data[14]));
bidirec gpio15 (.oe(en_gpio[15]), .inp(o_gpio[15]), .outp(i_gpio[15]), .bidir(io_data[15]));
bidirec gpio16 (.oe(en_gpio[16]), .inp(o_gpio[16]), .outp(i_gpio[16]), .bidir(io_data[16]));
bidirec gpio17 (.oe(en_gpio[17]), .inp(o_gpio[17]), .outp(i_gpio[17]), .bidir(io_data[17]));
bidirec gpio18 (.oe(en_gpio[18]), .inp(o_gpio[18]), .outp(i_gpio[18]), .bidir(io_data[18]));
bidirec gpio19 (.oe(en_gpio[19]), .inp(o_gpio[19]), .outp(i_gpio[19]), .bidir(io_data[19]));
bidirec gpio20 (.oe(en_gpio[20]), .inp(o_gpio[20]), .outp(i_gpio[20]), .bidir(io_data[20]));
bidirec gpio21 (.oe(en_gpio[21]), .inp(o_gpio[21]), .outp(i_gpio[21]), .bidir(io_data[21]));
bidirec gpio22 (.oe(en_gpio[22]), .inp(o_gpio[22]), .outp(i_gpio[22]), .bidir(io_data[22]));
bidirec gpio23 (.oe(en_gpio[23]), .inp(o_gpio[23]), .outp(i_gpio[23]), .bidir(io_data[23]));
bidirec gpio24 (.oe(en_gpio[24]), .inp(o_gpio[24]), .outp(i_gpio[24]), .bidir(io_data[24]));
bidirec gpio25 (.oe(en_gpio[25]), .inp(o_gpio[25]), .outp(i_gpio[25]), .bidir(io_data[25]));
bidirec gpio26 (.oe(en_gpio[26]), .inp(o_gpio[26]), .outp(i_gpio[26]), .bidir(io_data[26]));
bidirec gpio27 (.oe(en_gpio[27]), .inp(o_gpio[27]), .outp(i_gpio[27]), .bidir(io_data[27]));
bidirec gpio28 (.oe(en_gpio[28]), .inp(o_gpio[28]), .outp(i_gpio[28]), .bidir(io_data[28]));
bidirec gpio29 (.oe(en_gpio[29]), .inp(o_gpio[29]), .outp(i_gpio[29]), .bidir(io_data[29]));
bidirec gpio30 (.oe(en_gpio[30]), .inp(o_gpio[30]), .outp(i_gpio[30]), .bidir(io_data[30]));
bidirec gpio31 (.oe(en_gpio[31]), .inp(o_gpio[31]), .outp(i_gpio[31]), .bidir(io_data[31]));
```

```
gpio_top gpio_module(
    .wb_clk_i      (clk),
    .wb_rst_i      (wb_rst),
    .wb_cyc_i      (wb_m2s_gpio_cyc),
    .wb_adr_i      ({2'b0,wb_m2s_gpio_adr[5:2],2'b0}),
    .wb_dat_i      (wb_m2s_gpio_dat),
    .wb_sel_i      (4'b1111),
    .wb_we_i       (wb_m2s_gpio_we),
    .wb_stb_i      (wb_m2s_gpio_stb),
    .wb_dat_o      (wb_s2m_gpio_dat),
    .wb_ack_o      (wb_s2m_gpio_ack),
    .wb_err_o      (wb_s2m_gpio_err),
    .wb_inta_o     (gpio_irq),
    // External GPIO Interface
    .ext_pad_i     (i_gpio[31:0]),
    .ext_pad_o     (o_gpio[31:0]),
    .ext_padoe_o   (en_gpio));
```

RVfpga Lab 6: Connection with SweRV EH1

Ficheiro `wb_intercon.v`: Implementação do Multiplexer 7-1

```
108 wb_mux
109 #(.num_slaves (7),
110   .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111   .MATCH_MASK ({32'hffffff00, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffff00}))
112 wb_mux_io
113 (.wb_clk_i (wb_clk_i),
114  .wb_rst_i (wb_rst_i),
115  .wbm_adr_i (wb_io_adr_i),
116  .wbm_dat_i (wb_io_dat_i),
117  .wbm_sel_i (wb_io_sel_i),
118  .wbm_we_i (wb_io_we_i),
119  .wbm_cyc_i (wb_io_cyc_i),
120  .wbm_stb_i (wb_io_stb_i),
121  .wbm_cti_i (wb_io_cti_i),
122  .wbm_bte_i (wb_io_bte_i),
123  .wbm_dat_o (wb_io_dat_o),
124  .wbm_ack_o (wb_io_ack_o),
125  .wbm_err_o (wb_io_err_o),
126  .wbm_rty_o (wb_io_rty_o),
127  .wbs_adr_o (wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_o),
128  .wbs_dat_o (wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o),
129  .wbs_sel_o (wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o),
130  .wbs_we_o (wb_rom_we_o, wb_sys_we_o, wb_spi_flash_we_o, wb_spi_accel_we_o, wb_ptc_we_o, wb_gpio_we_o, wb_uart_we_o),
131  .wbs_cyc_o (wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o),
132  .wbs_stb_o (wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o),
133  .wbs_cti_o (wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o),
134  .wbs_bte_o (wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o),
135  .wbs_dat_i (wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i),
136  .wbs_ack_i (wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i),
137  .wbs_err_i (wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i),
138  .wbs_rty_i (wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i));
139
140 endmodule
```

CPU/Controller Signals

Peripheral Signals

RVfpga Lab 6: Exercícios Avançados - Exemplos

- **Exercício 3.** Expanda o **RVfpgaNexys** para suportar os cinco botões de pressão existentes na placa. Os botões de pressão são mostrados na Figura 22. Os cinco botões são designados de acordo com a sua localização: cima, baixo, esquerda, direita e centro – BTNU, BTND, BTNL, BTNR, BTNC.
- **Exercício 5.** Escreva um programa RISC-V em Assembly e um programa C que apresente uma contagem binária crescente nos LEDs, começando em 1. Utilize BTNC para alterar a velocidade da contagem e BTNU para reiniciar a contagem quando for premido.

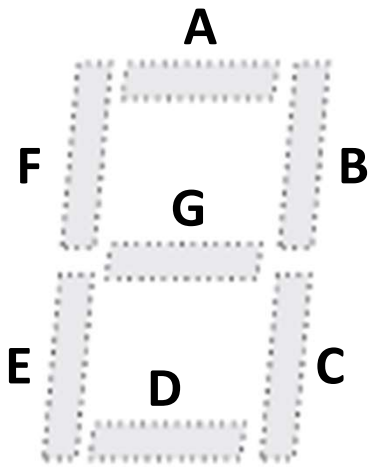
Lab 7:

Mostradores de 7-Segmentos

RVfpga Lab 7: Mostradores de 7-Segmentos

- Descreve como o sistema RVfpga foi expandido para funcionar com mostradores de 7 segmentos e mostra como modificar o controlador do mostrador de 7 segmentos.
- Começamos por descrever o funcionamento do controlador do mostrador de 7 segmentos
- Em seguida, analisamos a especificação de alto nível do controlador do mostrador de 8 dígitos de 7 segmentos incluído no sistema RVfpga e fornecemos alguns exercícios elementares.
- Por fim, analisamos a implementação de baixo nível deste controlador, efectuamos uma simulação no Verilator e fornecemos exercícios adicionais em que se modifica e experimenta a implementação do controlador.

RVfpga Lab 7: Revisão dos Mostradores de 7-Segmentos



- **7 segmentos LED: A-G**
- **Iluminar segmentos para criar dígitos específicos**
 - **1:** segmentos B e C
 - **2:** segmentos A, B, D, E, G
 - **3:** segmentos A, B, C, D, G
 - etc.

RVfpga Lab 7: Mostradores de 7-Segmentos na Nexys A7

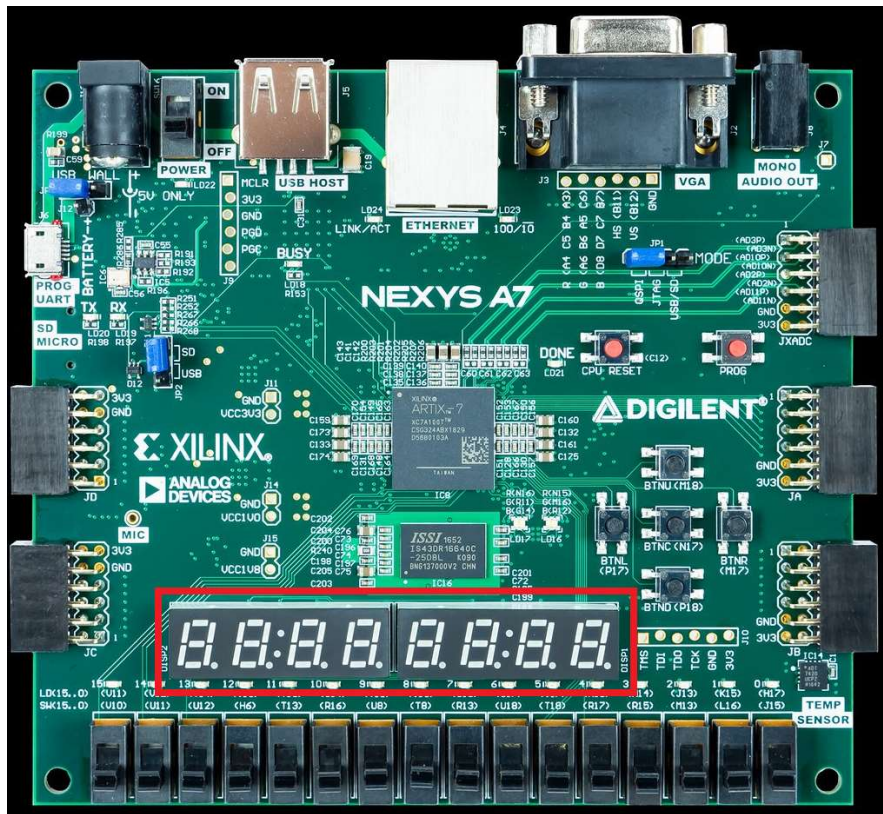


Figura da placa de <https://reference.digilentinc.com/>

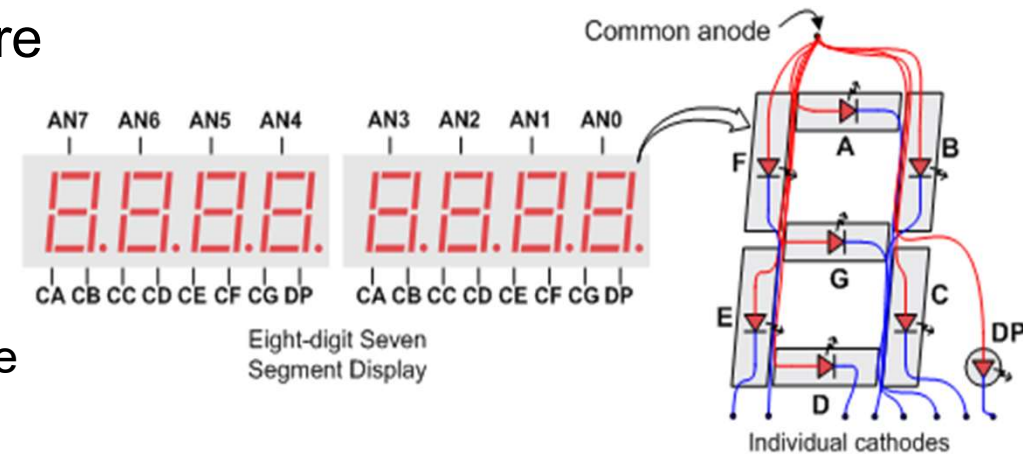
- 8 mostradores de 7-segmentos
- Acesso por memória mapeada:
 - **Enables_Reg:** 0x80001038
 - **Digits_Reg:** 0x8000103C
- Habilitações são a nível zero
- Exemplo: Mostrar 71 nos dois dígitos mais à direita:
 - **Enables_Reg** = 0xFC (0b11111100: ativar os dois dígitos mais à direita)
 - **Digits_Reg** = 0x71
 - **Assembly:**

```
li t0, 0x80001038
li t1, 0xFC
li t2, 0x71
sw t1, 0(t0)
sw t2, 4(t0)
```

RVfpga Lab 7: Circuito dos Mostradores de 7-Segmentos

8 Mostradores de 7-Segmentos

- Os dígitos têm **ânodo comum** (os ânodos dos LED's de cada dígito estão ligados entre si)
 - Os sinais de ânodo atuam como **ativadores** (AN0 - AN7)
 - Coloque a zero para ativar o dígito (AN0 - AN7 passam por um inversor, não mostrado, antes de ser alimentado por LED)
- Os mesmos **segmentos** estão ligados **juntos**
 - Os segmentos são ativados a **zero**
 - Multiplexagem-temporal** dos sinais AN0 - AN7 permite que valores únicos sejam exibidos em cada dígito
 - O sinal de AN de cada dígito (AN0 - AN7) deve ir a zero a cada **1-16 ms** para brilhar



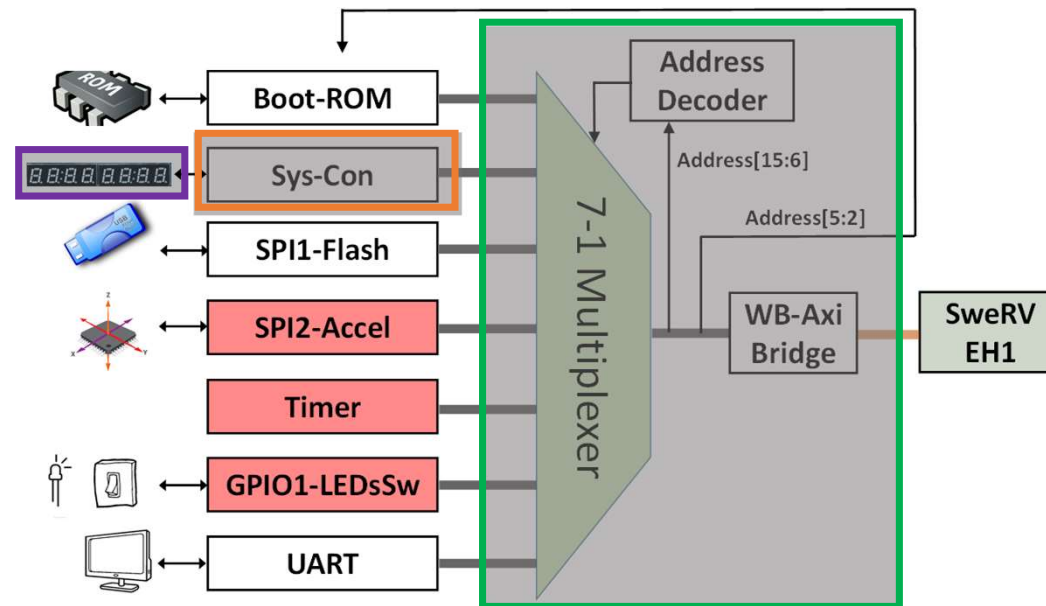
RVfpga Lab 7: Exercícios Elementares - Exemplos

- **Exercício 1.** Escreva um programa Assembly RISC-V e um programa C que mostrem o valor dos interruptores nos quatro dígitos mais à direita dos mostradores de 7-segmentos.
- **Exercício 2.** Escreva um programa em Assembly RISC-V e um programa em C que mostre "0-1-2-3-4-5-6-7-8" a mover-se da direita para a esquerda dos mostradores de 8 dígitos de 7-segmentos.

RVfpga Lab 7: Implementação de Baixo-Nível dos Mostradores de 7-Segmentos

- **3 partes principais:**

- A ligação aos **mostradores de 7-segmentos** na placa
- Descodificador de mostradores de 7 segmentos no System Control (**Sys-Con**)
- Interface ao **barramento do SweRV EH1**



RVfpga Lab 7: Ligação Externa

Ficheiro **rvfpganexys.xdc**: Define a ligação de **CA-CG** (chamados Digits_Bits[i] no SoC) e **AN[i]** com os mostradores de 7 segmentos na placa

```
60 ##7 segment display
61 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
62 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
63 set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
64 set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
65 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
66 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
67 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
68 #set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
69 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_F0E_B_15 Sch=an[0]
70 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
71 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
72 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
73 set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
74 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
75 set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
76 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
77
```

RVfpga Lab 7: Integração no SweRVolfX

- Ficheiro **swervolf_syscon.v**: instanciação do controlador de mostradores de 7-segmentos.
 - **Entradas:** **i_clk** (sinal do relógio), **i_rst** (sinal de reset)
 - **Memory-mapped inputs:** **Enables_Reg** (que dígitos da placa estão ativados) e **Digits_Reg** (número a mostrar)
 - **Saídas:** **AN** (dígito a ativar) e **Digits_Bits** (que segmentos a ativar)

```
// Eight-Digit 7 Segment Displays

reg [ 7:0] Enables_Reg;
reg [31:0] Digits_Reg;

SevSegDisplays_Controller SegDispl_Ctr(
    .clk           (i_clk),
    .rst_n         (i_rst),
    .Enables_Reg   (Enables_Reg),
    .Digits_Reg    (Digits_Reg),
    .AN            (AN),
    .Digits_Bits   (Digits_Bits)
);
```

RVfpga Lab 7: Exercícios Avançados - Exemplos

- **Exercício 3.** Modifique o controlador descrito neste Lab para que os mostradores de 8 dígitos de 7 segmentos possam mostrar qualquer combinação de LEDs ON/OFF.
- **Exercício 4.** Utilizar o novo controlador para exibir o seguinte no mostrador de 8 dígitos de 7 segmentos: "I SAY HI". Como de costume, implemente as versões Assembly RISC-V e C do programa.

Lab 8:

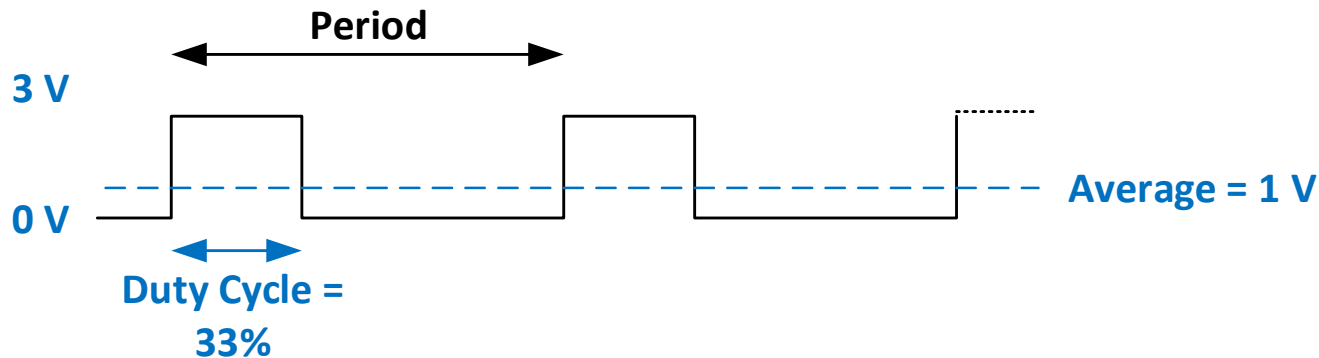
Temporizadores

RVfpga Lab 8: Temporizadores

- Gerar temporização precisa: Os temporizadores incrementam ou decrementam um contador a uma frequência fixa, que é normalmente configurável, e interrompem o processador quando o contador atinge zero ou um valor predefinido.
- Temporizadores mais sofisticados também podem executar outras funções, como gerar formas de onda moduladas por largura de pulso (PWM) para controlar a velocidade de um motor ou o brilho de uma luz.
- Neste laboratório, começamos por descrever a especificação de alto nível do temporizador incluído no sistema RVfpga e, em seguida, explicamos a sua implementação de baixo nível. São propostos exercícios elementares e avançados que mostram como utilizar e modificar um temporizador.

RVfpga Lab 8: Módulo Temporizador (PTC)

- O módulo temporizador utilizado é da OpenCores:
<https://opencores.org/projects/ptc>
- O módulo temporizador (também chamado módulo **PTC**) é utilizado para :
 - Temporizador/Contador: conta as transições do relógio (ou de outro sinal, também chamados eventos)
 - Modulação por largura de Pulso (PWM):
 - Varia a largura do pulso (ou ciclo de trabalho)
 - Usado para aproximar uma tensão analógica digitalmente
 - **Exemplo PWM:** 33% duty cycle (o sinal é activo 1/3rd do tempo). Se o nível alto for 3 V, tensão analógica (tensão média do sinal) é $3\text{ V} * 0.33 = 1\text{ V}$



RVfpga Lab 8: Registos do Temporizador (PTC)

Nome	Endereço	Largura	Acesso	Descrição
RPTC_CNTR	0x80001200	1-32	L/E	Contador principal do PTC
RPTC_HRC	0x80001204	1-32	L/E	Registo Referencia/Captura PTC HI
RPTC_LRC	0x80001208	1-32	L/E	Registo Referencia/Captura PTC LO
RPTC_CTRL	0x8000120C	9	L/E	Registo de Controlo

- **RPTC_CNTR:** Contador (valor do contador)
- **RPTC_HRC:** Captura de referência elevada - indica o número de ciclos (após reposição) quando a saída deve ser ativa no modo PWM
- **RPTC_LRC:** Captura de referência baixa – indica o número de ciclos (após reposição) quando a contagem está completa no modo contador/timer; indica o número de ciclos do relógio (após reposição) quando a saída deve ser zero no modo PWM.
- **RPTC_CTRL:** Registo de Controlo

RVfpga Lab 8: Exemplo de Uso do Temporizador

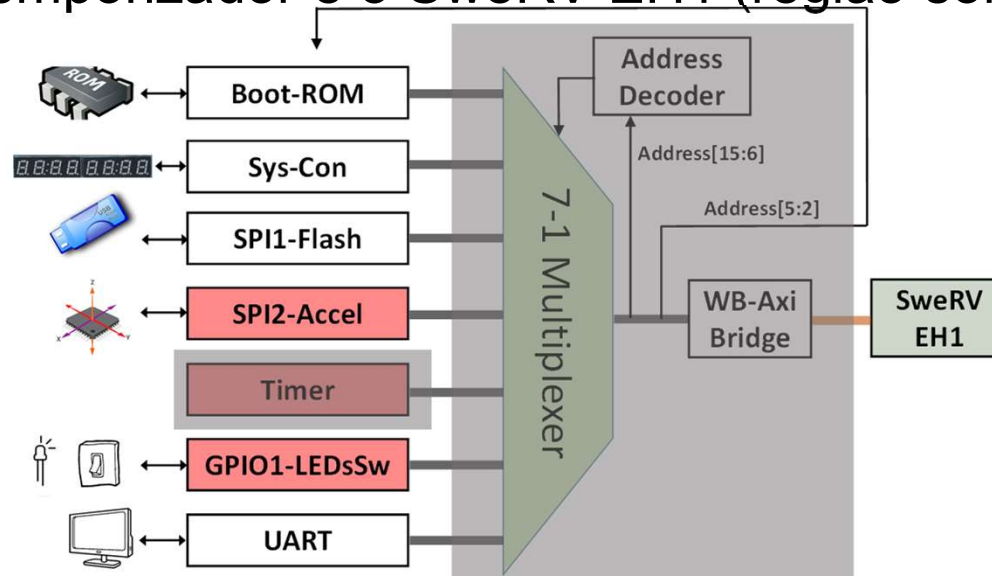
- Definir **RPTC_LRC** para o número de ciclos a contar
- Definir bits de controlo (**RPTC_CTRL**) para configurar o temporizador:
 - Repor o contador e limpar as interrupções: **RPTC_CTRL = 0xC0** (0b011000000): CNTRRST (bit 7) = 1: o Contador é reposto (RPTC_CNTR = 0); INT (bit 6) = 1: o pedido de interrupção é limpo.
 - Habilitar contador e interrupções: **RPTC_CTRL = 0x21** (0b000100001): EN (bit 0) = 1: o contador está ativado, então o RPTC_CNTR incrementa; INTE (bit 5) = 1: PTC estabelece uma interrupção quando RPTC_CNTR == RPTC_LRC.
- Programa lê bit de interrupção no registo de controlo (**INT** é o **bit 6** de **RPTC_CTRL**) até que seja 1 (indicando que RPTC_CNTR == RPTC_LRC).
- Este algoritmo **não utiliza interrupções**, mas lê o bit de interrupção (INT, bit 6 de RPTC_CTRL) para determinar quando foi atingido o número correto de ciclos de relógio. Mostramos como utilizar as interrupções no Lab 9.

RVfpga Lab 8: Exercícios Elementares - Amostra

- **Exercício 1.** Escreva um programa que mostre uma contagem crescente nos mostradores de 8 dígitos de 7 segmentos. O valor deve mudar cerca de uma vez por segundo e, para criar este atraso, deve utilizar o módulo temporizador.
 - Primeiro, escreva o programa em linguagem assembly RISC-V e execute-o na placa Nexys A7.
 - Em seguida, efectue uma simulação no Verilator com o mesmo programa.
 - Agora escreva o programa em C e execute-o na placa Nexys A7.
 - Simule o seu programa C no Verilator, como na parte (b) para o programa Assembly RISC-V.

RVfpga Lab 8: Implementação de baixo nível do temporizador

- **Dividido em 2 partes principais**
 - (Nenhuma ligação externa)
 - Integração do módulo temporizador no SweRVolfX (região sombreada à esquerda)
 - Ligação entre o temporizador e o SweRV EH1 (região sombreada à direita)



RVfpga Lab 8: Integração no SweRVolfX

Ficheiro **swervolf_core.v**: instanciação do módulo PTC

```
// PTC
wire      ptc_irq;

ptc_top timer_ptc(
    .wb_clk_i      (clk),
    .wb_rst_i      (wb_rst),
    .wb_cyc_i      (wb_m2s_ptc_cyc),
    .wb_adr_i      ({2'b0,wb_m2s_ptc_adr[5:2],2'b0}),
    .wb_dat_i      (wb_m2s_ptc_dat),
    .wb_sel_i      (4'b1111),
    .wb_we_i      (wb_m2s_ptc_we),
    .wb_stb_i      (wb_m2s_ptc_stb),
    .wb_dat_o      (wb_s2m_ptc_dat),
    .wb_ack_o      (wb_s2m_ptc_ack),
    .wb_err_o      (wb_s2m_ptc_err),
    .wb_inta_o     (ptc_irq),
    // External PTC Interface
    .gate_clk_pad_i (),
    .capt_pad_i     (),
    .pwm_pad_o      (),
    .oen_padoen_o   ()
);
```

RVfpga Lab 8: Exercícios Avançados - Exemplo

- **Exercício 2.** Modificar o RVfpgaNexys para ligar o sinal de saída PWM do temporizador (*pwm_pad_o*) a um dos dois LED tricolores disponíveis na placa Nexys A7.
- **Exercício 3.** Implementar um programa que utilize o novo periférico para controlar o LED tricolor, utilizando o valor fornecido pelos 16 interruptores.

Lab 9:

E/S com

Interrupções

RVfpga Lab 9: E/S com Interrupções

- E/S com Interrupções vs E/S programadas
- Controlador de Interrupção do Sistema Rvfpga
- Como configurar as interrupções usando os Pacotes de Suporte de Periféricos e Suporte de Placas da Western Digital (PSP e BSP)
- Exemplo de uso de Interrupções

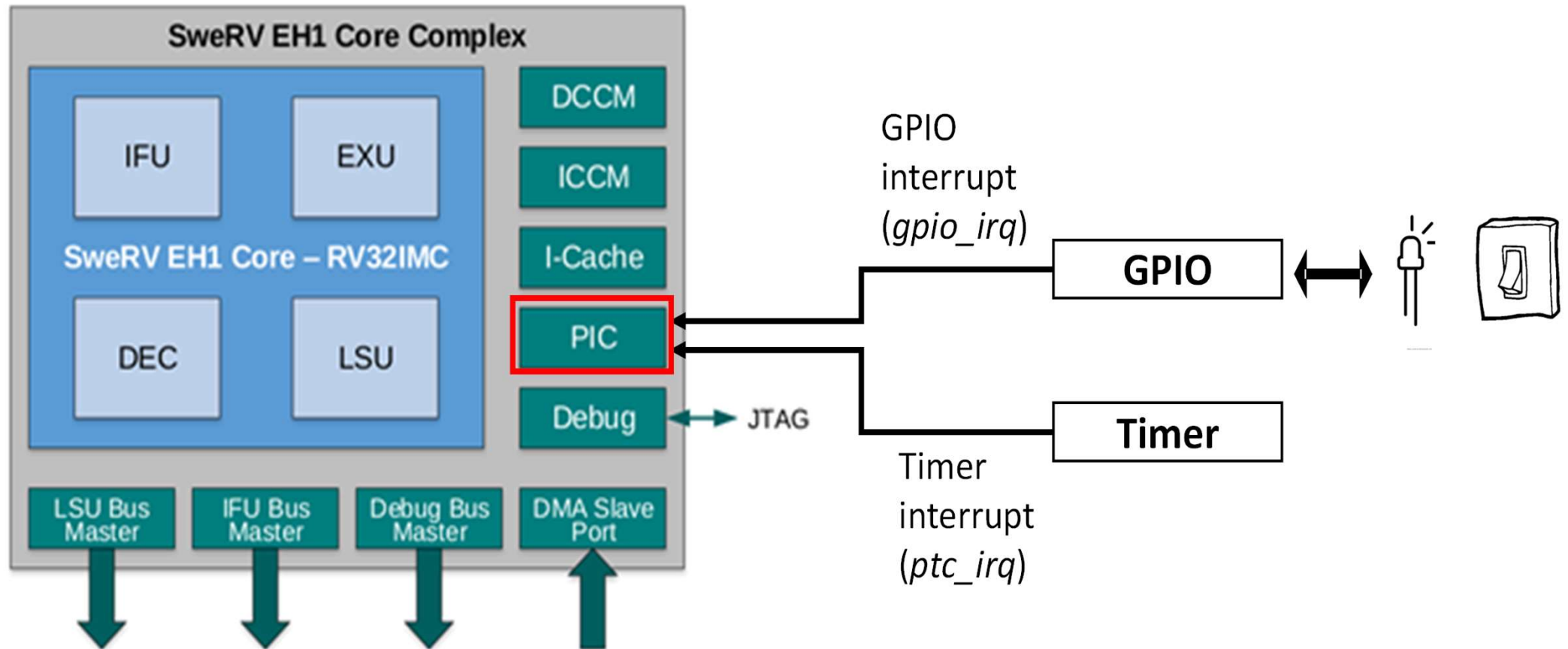
RVfpga Lab 9: Introdução a E/S com Interrupções

- **E/S programadas:**
 - Um programa **testa continuamente** o valor de um sinal (por exemplo, um interruptor) até que um valor desejado seja observado.
 - O **processador fica ocupado** neste processo - em vez de fazer outro trabalho útil.
- **E/S com interrupções:**
 - Um **evento** (como a asserção de um interruptor) faz com que o processador salte para uma rotina de serviço de interrupção - **interrupt service routine** (ISR, também chamada de "*interrupt handler*"), que trata do evento e depois regressa ao programa normal.
 - Até que esse evento ocorra, o **processador pode realizar outro trabalho** útil.

RVfpga Lab 9: Tratamento de Interrupções

- As interrupções podem ser causadas por **hardware** ou **software**
- Neste lab, concentramo-nos nas **interrupções por hardware**
- O núcleo SweRV EH1 interrompe após a indicação do PLIC o controlador de interrupção a nível de plataforma (*platform-level interrupt controller*) do RISC-V. É conhecido como o Controlador de Interrupção Programável (PIC). Ele tem:
 - 255 fontes de interrupção
 - 15 níveis de prioridade

RVfpga Lab 9: Interrupções por Hardware



RVfpga Lab 9: Funções PSP/BSP da WD para tratamento de interrupções

Header	Description
void pspInterruptsSetVectorTableAddress (void* pVectTable);	Prepares vector-table address
void pspExternalInterruptSetVectorTableAddress (void* pExtIntVectTable);	Prepares external interrupts vector-table address
void bspInitializeGenerationRegister (u32_t uiExtInterruptPolarity)	Put the Generation-Register in its initial state
void bspClearExtInterrupt (u32_t uiExtInterruptNumber)	Clear the trigger that generates external interrupt
void pspExtInterruptSetPriorityOrder (u32_t uiPriorityOrder);	Sets Priority Order (Standard or Reserved)
void pspExtInterruptsSetThreshold (u32_t uiThreshold);	Sets the priority threshold of the external interrupts in the PIC
void pspExtInterruptsSetNestingPriorityThreshold (u32_t uiNestingPriorityThreshold);	Sets the nesting priority threshold of the external interrupts in the PIC
void pspExtInterruptSetPolarity (u32_t uiIntNum, u32_t uiPolarity);	Sets the polarity (active-high or active-low) of a specified interrupt line
void pspExtInterruptSetType (u32_t uiIntNum, u32_t uiIntType);	Sets the type (Level-triggered or Edge-triggered) of a specified interrupt line
void pspExtInterruptClearPendingInt (u32_t uiIntNum);	Clears the indication of pending interrupt for the specified interrupt line
void pspExtInterruptSetPriority (u32_t uiIntNum, u32_t uiPriority);	Sets the priority of a specified interrupt line
void pspExternalInterruptEnableNumber (u32_t uiIntNum);	Enables a specified interrupt line in the PIC
void pspInterruptsEnable (void);	Enable interrupts (in all privilege levels) regardless their previous state
void pspInterruptsDisable (u32_t *pOutPrevIntState);	Disables interrupts and return the current interrupt state in each one of the privileged levels

RVfpga Lab 9: Exemplo de Interrupção Com o PSP/BSP da WD

Utilizar interrupções para ler o valor de Switch[0] - apenas na transição ascendente (transição 0→1)

- Passos para configurar o sistema utilizando as funções do PSP/BSP:
 1. Inicializar o sistema de interrupções
 2. Inicializar a linha de interrupção externa IRQ4 e ligá-la ao GPIO
 3. Inicializar os registos GPIO para utilizar as interrupções :
 - RGPI0_INTE = 0x10000 (ativar a interrupção para Switch[0])
 - RGPI0_PTRIG = 0x10000 (interrupção despoletada na transição ascendente de Switch[0])
 - RGPI0_INTS = 0x0 (limpa todas as interrupções)
 - RGPI0_CTRL = 0x1 (ativa as interrupções do GPIO)
 4. Ativar as interrupções globais
- GPIO_ISR (ver o próximo slide): Invocado quando uma interrupção é acionada no GPIO
 1. O estado atual dos LEDs é lido
 2. Os LEDs são invertidos e mascarados
 3. Os LEDs são escritos com o novo valor
 4. A interrupção GPIO é limpa
 5. A interrupção externa IRQ4 é limpa

Código completo: `[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c`

RVfpga Lab 9: Exemplo de ISR para inverter o LED à direita se SW 0→1

```
void GPIO_ISR(void) {
    unsigned int i;

    /* Invert LED value */
    i = M_PSP_READ_REGISTER_32(GPIO_LEDs); // RGPIO_OUT
    i = !i;                                // Inverter os LEDs
    i = i & 0x1;                           // Apenas altera o LED da direita

    M_PSP_WRITE_REGISTER_32(GPIO_LEDs, i) // RGPIO_OUT

    /* Limpar a interrupção do GPIO */
    M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); // RGPIO_INTS

    /* Limpar esta interrupção (IRQ4) */
    bspClearExtInterrupt(4);
}
```

RVfpga Lab 9: Exercícios - Exemplos

- **Exercício 1.** Modificar o programa *LED-Switch_7SegDispl_Interrupts_C-Lang* para adicionar uma segunda fonte de interrupção, neste caso gerada pelo temporizador.
- **Exercício 2.** Modifique o RVfpgaNexys para incluir uma terceira fonte de interrupção proveniente do segundo GPIO que concebeu no Lab 6 para controlar os botões de pressão da placa (GPIO2).
- **Exercício 3.** Utilize a versão alargada do RVfpgaNexys que concebeu no exercício anterior para implementar um programa C que apresenta uma contagem binária crescente nos LEDs, começando em 1.
 - Criar um atraso com o temporizador, utilizando interrupções, entre cada valor incrementado, de modo a que os valores sejam visíveis ao olho humano.
 - Ler BTNC e utilizá-lo para alterar a velocidade da contagem.
 - Ler Switch[0] e utilizá-lo para reiniciar a contagem sempre que for premido.

Lab 10:

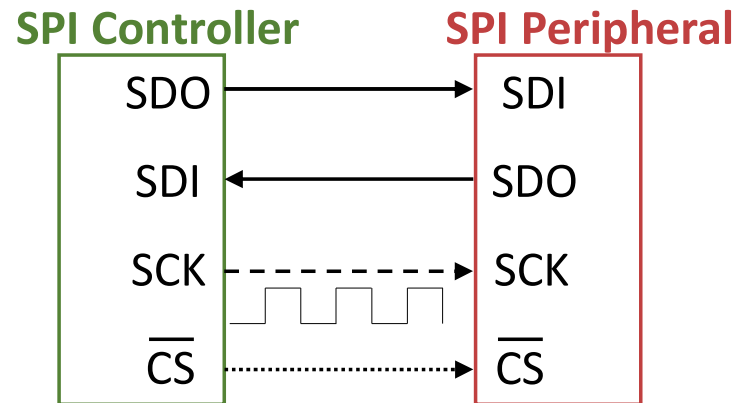
Barramentos

Série

RVfpga Lab 10: Barramentos Série

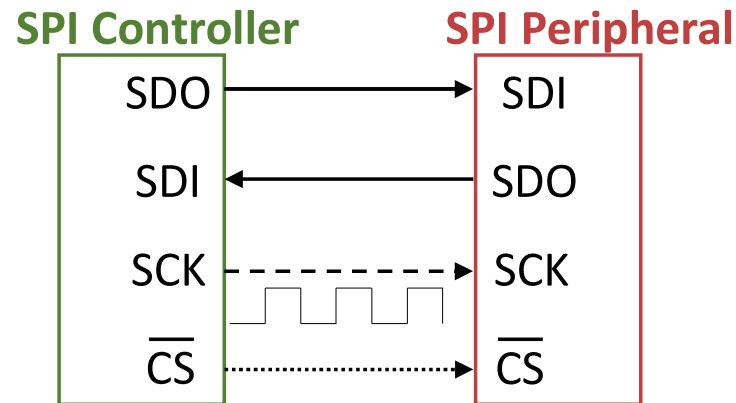
- Começamos por descrever o funcionamento dos barramentos série
 - Os barramentos série enviam **um bit de cada vez**
 - Em contraste, os barramentos paralelos enviam **vários bits ao mesmo tempo**
- **Barramentos série mais comuns**
 - **UART** (universal asynchronous receiver/transmitter)
 - **SPI** (serial peripheral interface)
 - **I2C** (inter-integrated circuit protocol)
- Concentramo-nos no acelerómetro SPI disponível na placa Nexys A7:
 - Análise da especificação de alto nível + exercícios elementares
 - Análise da implementação de baixo nível + exercícios avançados
- Exercícios mais avançados com UART e I2C

RVfpga Lab 10: Barramentos Série - SPI

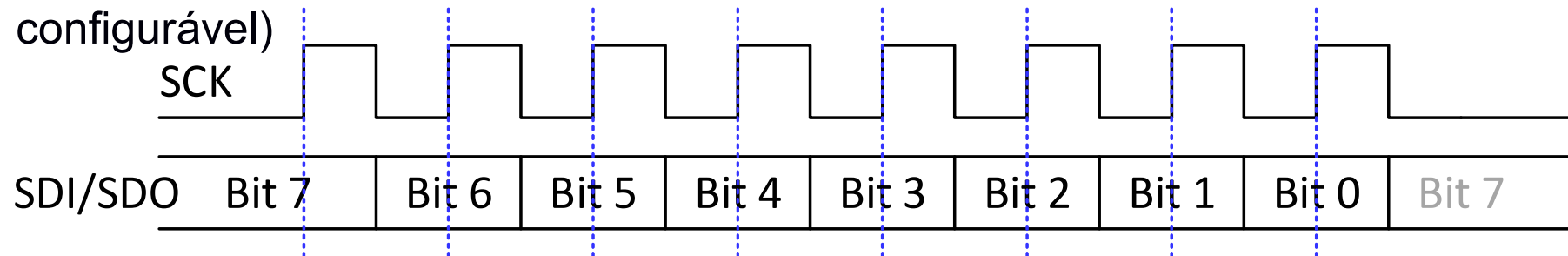


- **Controlador:** envia sinal de relógio, envia e recebe dados
- **Periférico:** recebe sinal de relógio, envia & recebe dados
- **Sinais:**
 - **SDO:** Serial Data Out - Saída de dados em série
 - **SDI:** Serial Data In - Entrada de dados em série
 - **SCK:** SPI clock - Relógio de sincronização SPI
 - **CSbar:** low-asserted chip select – ativação de nível baixo do periférico

RVfpga Lab 10: Barramentos Série



- **SCK inativo**
- Quando o controlador envia uma **transição do SCK**, tanto o controlador como o periférico **amostram e enviam dados**. Os dados são trocados (enviados) na transição descendente e amostrados na transição ascendente (embora esta seja configurável)



RVfpga Lab 10: Módulo SPI do Sistema Rvfpga

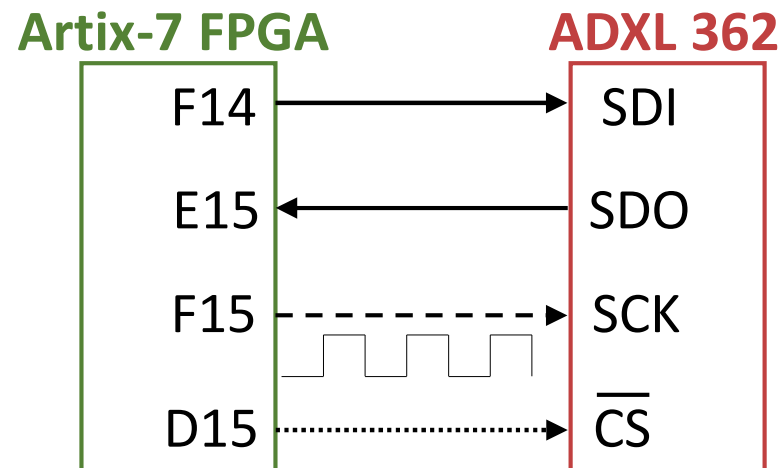
- Módulo SPI do Sistema Rvfpga é proveniente do OpenCores https://opencores.org/projects/simple_spi
- buffers de leitura e escrita com 4 entradas
- Registos SPI :

Nome	Endereço	Largura	Acesso	Descrição
SPCR	0x80001100	8	R/W	Registo de Controlo
SPSR	0x80001108	8	R/W	Registo de Status
SPDR	0x80001110	8	R/W	Registo de Dados
SPER	0x80001118	8	R/W	Registo de Extensões
SPCS	0x80001120	8	R/W	Registo CS

RVfpga Lab 10: Acelerómetro ADXL362 SPI

- A placa Nexys A7 inclui um acelerómetro analógico ADXL362. Pode encontrar a informação completa em:

<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>



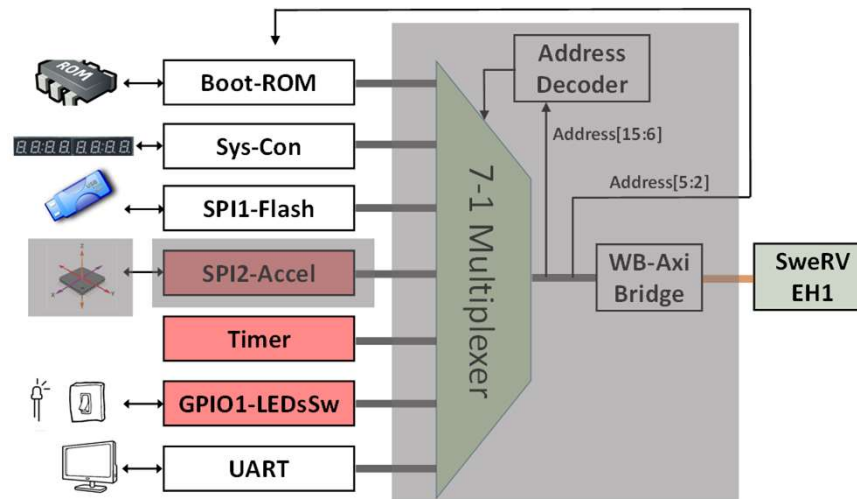
RVfpga Lab 10: Exercícios Elementares - Exemplo

- **Exercício 1.** Crie um programa Assembly RISC-V que leia os oito bits mais significativos dos dados de aceleração dos eixos X, Y e Z e, em seguida, exiba esses valores no mostrador de 8 dígitos de 7 segmentos.

RVfpga Lab 10: Implementação do Acelerómetro

- **Dividida em 3 partes principais**

- Ligação externa do RVfpgaNexys ao acelerómetro na placa (região sombreada à esquerda)
- Integração do novo módulo SPI no SweRVolfX (região sombreada média)
- Ligação entre o acelerómetro e o SweRV EH1 (região sombreada à direita)



RVfpga Lab 10: Ligação externa

Ficheiro **rvfpganexys.xdc**: Define a ligação dos sinais SPI utilizados no SoC com os correspondentes pinos do acelerómetro na placa

```
78 ##Accelerometer
79 set_property -dict { PACKAGE_PIN E15    IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
80 set_property -dict { PACKAGE_PIN F14    IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
81 set_property -dict { PACKAGE_PIN F15    IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
82 set_property -dict { PACKAGE_PIN D15    IOSTANDARD LVCMOS33 } [get_ports { o_accel_cs_n }];
```

RVfpga Lab 10: Integração no SweRVolfX

Ficheiro **swervolf_core.v**: Buffers de três estados (*tri-state buffers*) e instância do módulo GPIO

```
simple_spi spi2
  // Wishbone slave interface
  .clk_i  (clk),
  .rst_i  (wb_rst),
  .adr_i  (wb_m2s_spi_accel_adr[2] ? 3'd0 : wb_m2s_spi_accel_adr[5:3]),
  .dat_i  (wb_m2s_spi_accel_dat[7:0]),
  .we_i   (wb_m2s_spi_accel_we),
  .cyc_i  (wb_m2s_spi_accel_cyc),
  .stb_i  (wb_m2s_spi_accel_stb),
  .dat_o  (spi2_rdt),
  .ack_o  (wb_s2m_spi_accel_ack),
  .inta_o (spi2_irq),
  // SPI interface
  .sck_o  (o_accel_sclk),
  .ss_o   (o_accel_cs_n),
  .mosi_o (o_accel_mosi),
  .miso_i (i_accel_miso));
```

RVfpga Lab 10: Exercícios Avançados - Exemplos

- **Exercício 2.** A UART (Universal Asynchronous Receiver-Transmitter) é um módulo de comunicação serial assíncrono. Primeiro, analise a implementação de baixo nível deste módulo no Rvfpga. Em seguida, crie um programa em Assembly RISC-V que imprima uma mensagem para o terminal do PlatformIO através da porta série.
- **Exercício 3.** Implemente as três funções seguintes na linguagem C:
 - `char uart_getchar(void)`: Esta função espera que o teclado envie um carácter através da UART para a placa Nexys A7 e, em seguida, devolve esse carácter como valor de retorno.
 - `int uart_putchar(char c)`: Esta função recebe um carácter como argumento de entrada e apresenta-o na consola de série através da UART.
 - `int SevSegDispl(char c)`: Esta função recebe um carácter como argumento de entrada e apresenta-o no dígito mais à direita dos mostradores de 7 segmentos, deslocando os restantes dígitos uma posição para a esquerda (o dígito mais à esquerda perde-se).

Labs 11-20

Compreender o Núcleo e os Sistemas de Memória

RVfpga Panorâmica dos Labs 11-20

- Os Labs 11-20 descem ao nível microarquitetónico e analisam como funciona o processador SweRV EH1 e a hierarquia cache/memória.
- Cada laboratório está dividido em duas partes :
 - Explicação teórica dos conceitos
 - Ilustração dos conceitos utilizando figuras e uma simulação Verilator de um programa de exemplo para ilustrar o conceito.

Também fornecemos exercícios para aprofundar a compreensão e ganhar experiência com os conceitos descritos.

RVfpga SweRVref

- Para além destes 10 laboratórios, que descrevemos a seguir, o documento RVfpga_SweRVref fornece instruções adicionais sobre os seguintes tópicos :
 - Secção 1: **Sigasi Studio**
 - Secção 2: **Configuração** do processador **SweRV EH1**
 - Secção 3: **RVfpga Hierarquia de módulos do sistema** e seus **sinais** mais relevantes
 - Secção 4: **Principais estruturas/tipos** de agrupamento de **bits de controlo**
 - Secção 5: **Instruções comprimidas RISC-V**
 - Secção 6: **Indicadores de desempenho (*Benchmarks*) reais**

Lab 11:

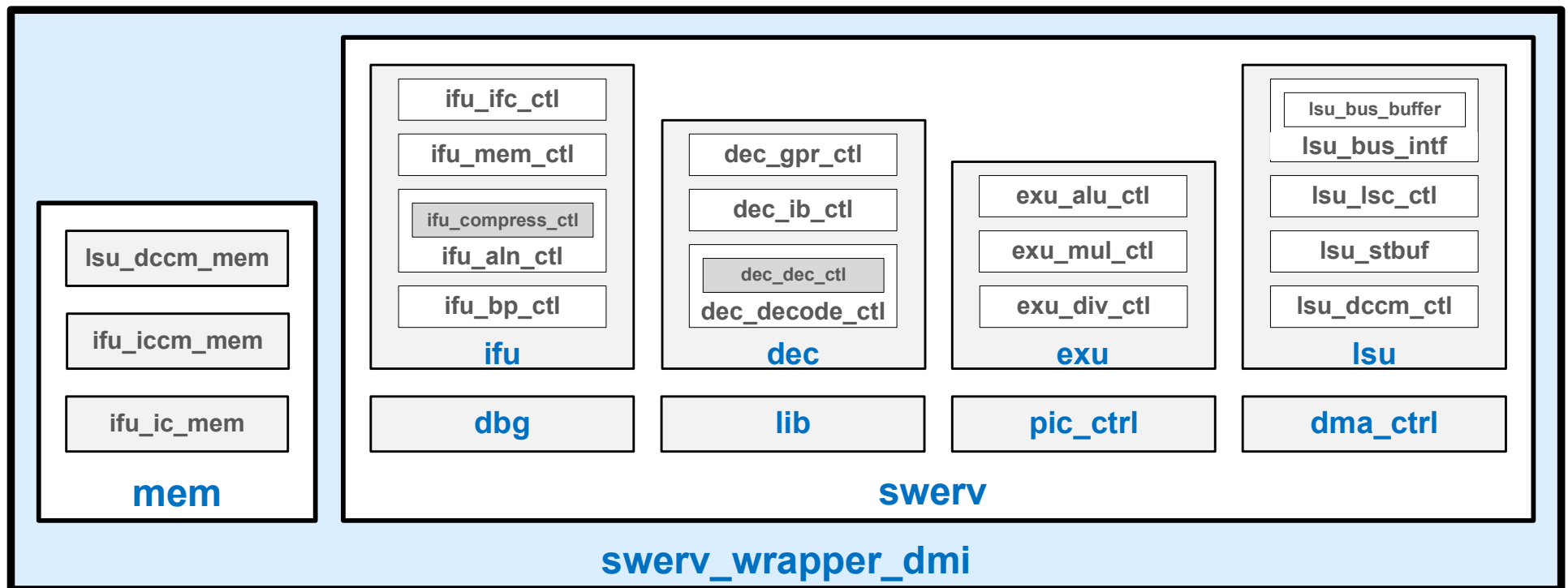
Configuração, Organização, e Monitorização do Desempenho do SweRV EH1

RVfpga Lab 11: O Processador SweRV EH1

- Neste laboratório, começamos a analisar o processador SweRV EH1. Especificamente:
 - Começamos por descrever a organização RTL em Verilog e os pormenores de cada andar do *pipeline*.
 - Em seguida, mostramos como utilizar os contadores de desempenho do SweRV EH1 para analisar o desempenho do processador.

RVfpga Lab 11: Módulos Verilog do SweRV EH1

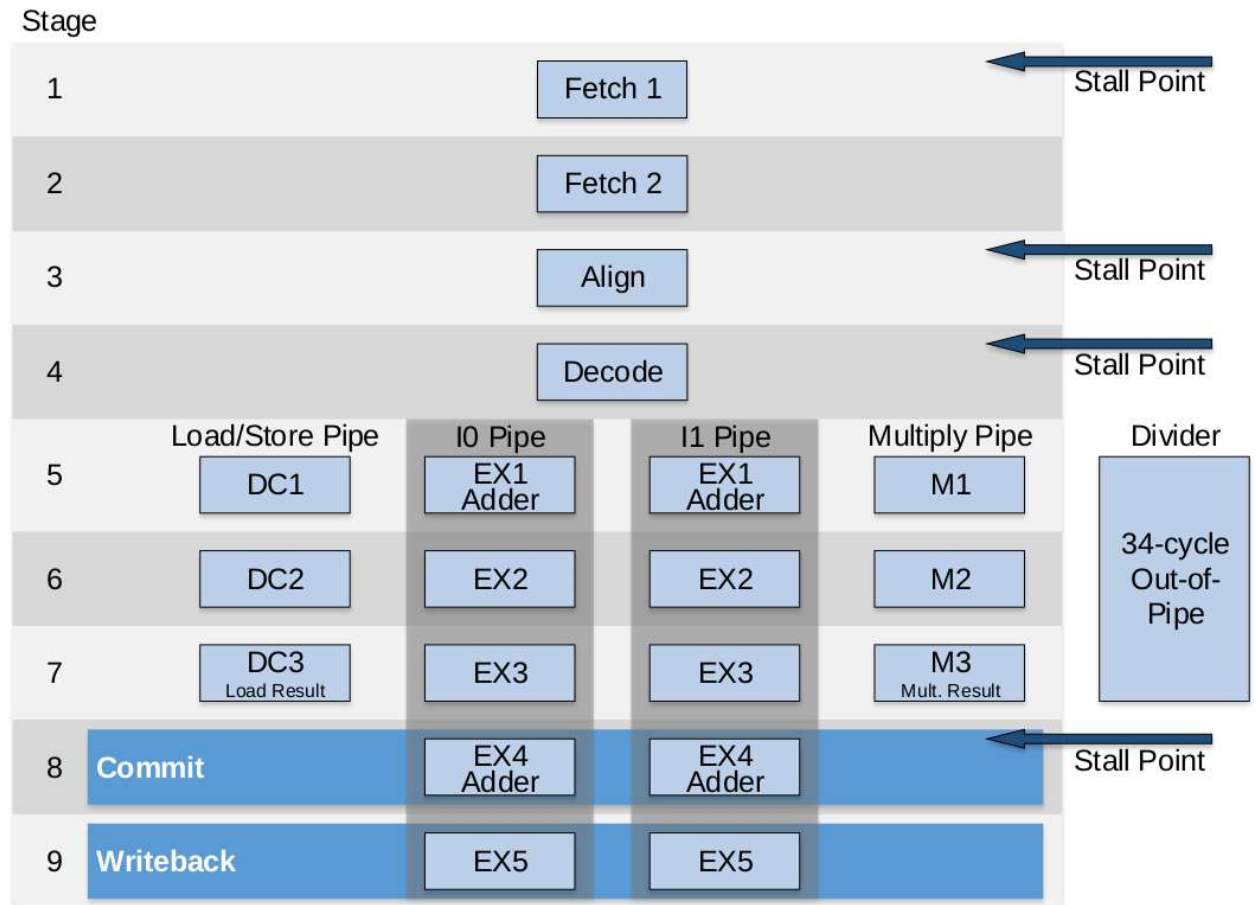
- Módulo **swerv** : CPU.
- Módulo **mem** (hierarquia de memória): memórias de instruções/dados estreitamente acopladas ICCM, DCCM, e cache de instruções I\$.



RVfpga Lab 11: O Pipeline SweRV EH1

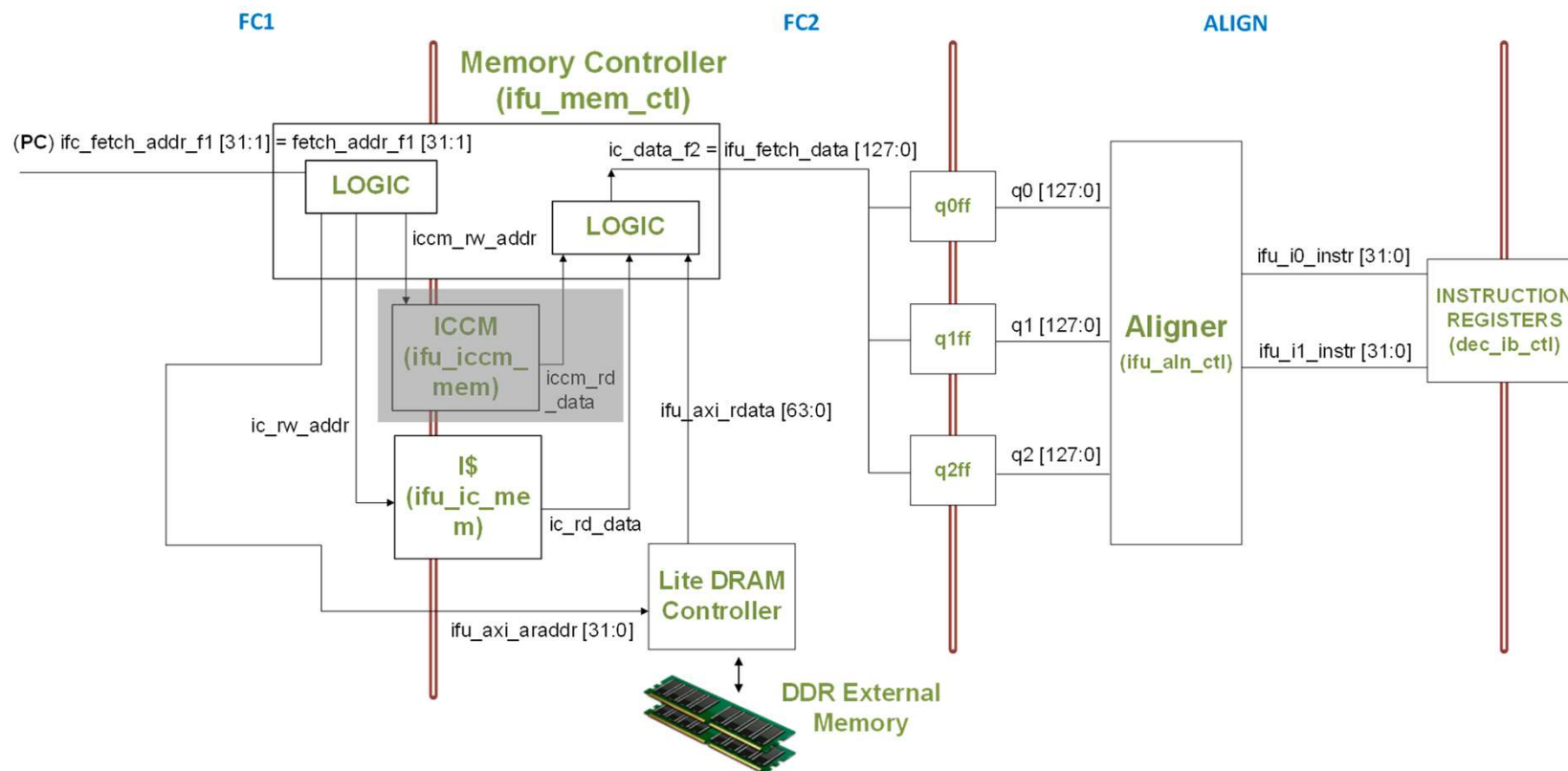
SweRV Core™

SweRV EH1 é um processador superescalar de 32 bits com 2 vias e pipeline com 9 andares de processamento em ordem.



RVfpga Lab 11: Andares de Fetch (FC1 e FC2) e Align

- Os três primeiros andares: dois andares de Fetch (FC1 e FC2) e um de Align

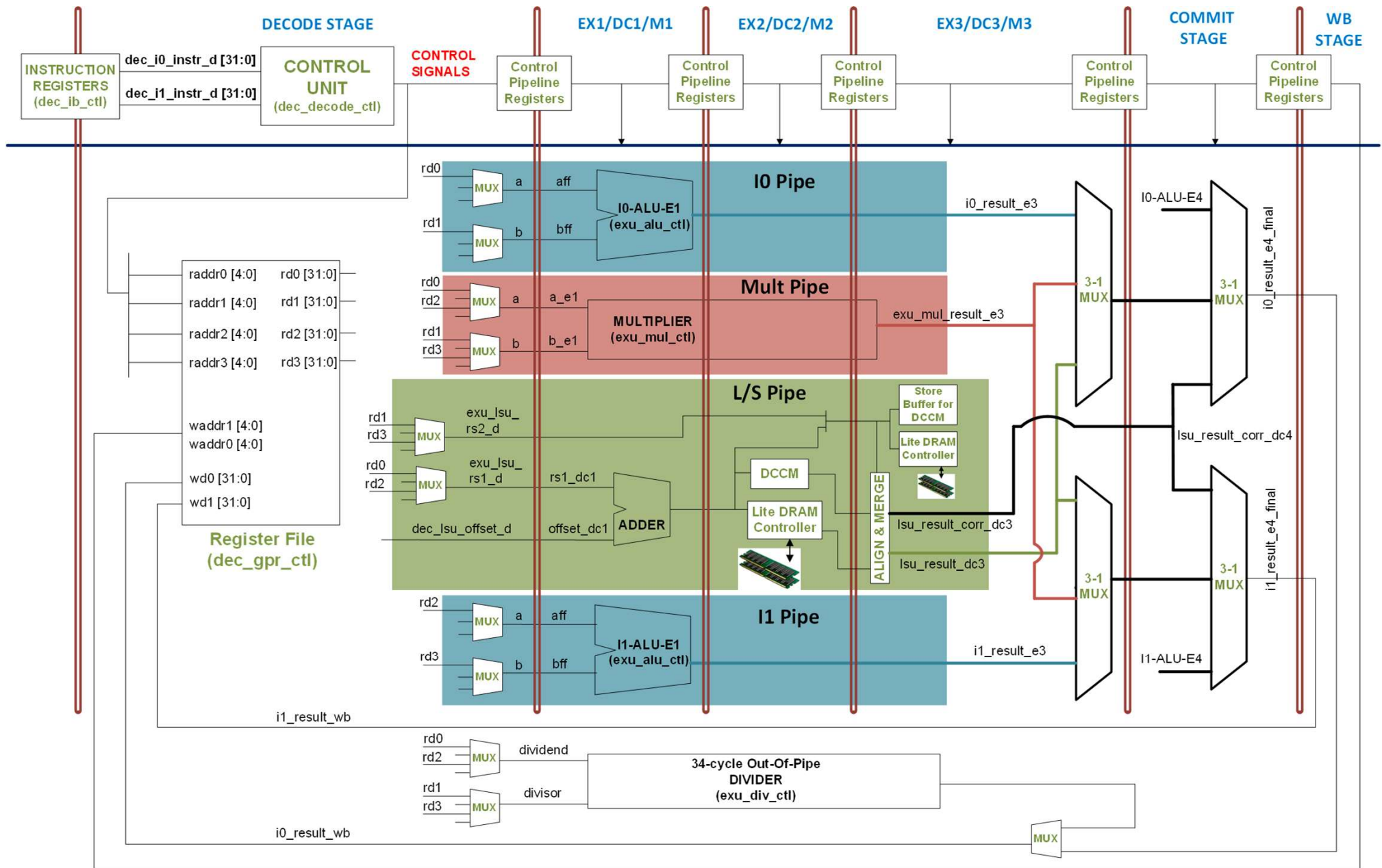


RVfpga Lab 11: Andares de Fetch (FC1 e FC2)

- No RVfpga, a Memória de Instruções consiste em:
 - Cache de Instruções 16 KiB
 - Memória externa 128 MiB DDR
- **Andares de Fetch:** ler instruções da Memória de Instruções
 - **FC1:** Calcula o endereço da instrução (`ifc_fetch_addr_f1`)
 - **FC2:** Lê instruções do I\$, DDR Memória Externa, ou ICCM. (O I\$ apenas armazena a memória dentro da gama de endereços da Memória Principal)
- O andar de Align executa duas tarefas principais:
 - **Fornecer duas instruções de 32 bits por ciclo para o andar de Decode:** Extrai duas instruções por ciclo dos pacotes de 128 bits fornecidos pela Memória de Instruções e atribui-as a cada uma das duas vias disponíveis no SweRV EH1.
 - **Descomprime instruções:** o andar de Align descomprime instruções de 16 bits em instruções de 32 bits

RVfpga Lab 11: Andares de Decode, EX1/2/3, Commit e WB

- A figura do próximo slide mostra as últimas seis fases do pipeline: o andar de **Decode**, três fases de execução (**Execution**), o andar de **Commit** e o andar de **Writeback** (WB).



RVfpga Lab 11: Andar de Decode

- O andar de Decode executa duas tarefas principais:
 - **Descodificar as instruções e gerar os sinais de controlo** (realizado pela Unidade de Controlo)
 - **Distribuir as instruções e operandos pelos canais apropriados:**
 - Canais (*pipes*):
 - 2 canais de inteiros: I0 e I1
 - Canal de Multiplicação
 - Canal Leitura/Escrita (L/E)
 - Divisor de 34 ciclos “*fora do pipeline*”
 - Vários Multiplexers seleccionam entre possíveis operandos, que podem ser provenientes de:
 - Lógica de desvio (*bypass*)
 - Valor Imediato
 - Register File

RVfpga Lab 11: Andares de Execute – 3 Pipes e um Divisor

- **Pipes I0/I1**: Dois canais de inteiros têm três fases (EX1, EX2, e EX3). EX1 realiza a operação da ALU.
- **Pipe de Multiplicação**: O canal de multiplicação contém um multiplicador inteiro de 3 ciclos usando três andares (M1, M2, e M3).
- **Pipe de Leitura/Escrita (L/E)**: O canal de L/E executa instruções de leitura e escrita.
- **Divisor**: O divisor é um divisor de inteiros de 34 ciclos, *fora-do-pipeline*.

No final do terceiro andar de execução (EX3/DC3/M3), o resultado das instruções é selecionado a partir do canal adequado (I0/I1, MUL, ou L/S) utilizando dois Multiplexers 3:1, um para cada via. O Divisor tem o seu próprio caminho para o Ficheiro de Registo.

RVfpga Lab 11: Andares de Commit e WriteBack

- **Andar de Commit:** Seleciona o resultado para ser escrito para o Register File.
- **Andar de Writeback:** Escreve os resultados no Register File utilizando os Portos 0 e 1. Os Registos de Controlo de Pipeline fornecem os identificadores de registo e os sinais de ativação (que foram gerados no andar de Decode).

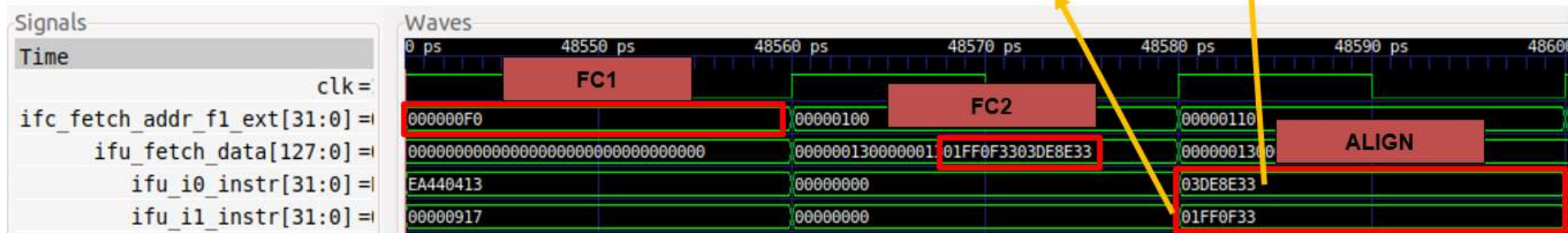
RVfpga Lab 11: Programa de Exemplo - Simulação no Verilator

```
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, ...)
add x30, x30, x31      # x30 = 4+1 = 5 (later iterations: 5+1=6, ...)
INSERT_NOPS_10
add x29, x29, 1        # x29 = x29 + 1
INSERT_NOPS_10
beq zero, zero, REPEAT # Repete o ciclo
```

RVfpga Lab 11: Simulação – Andares FC1, FC2, e Align

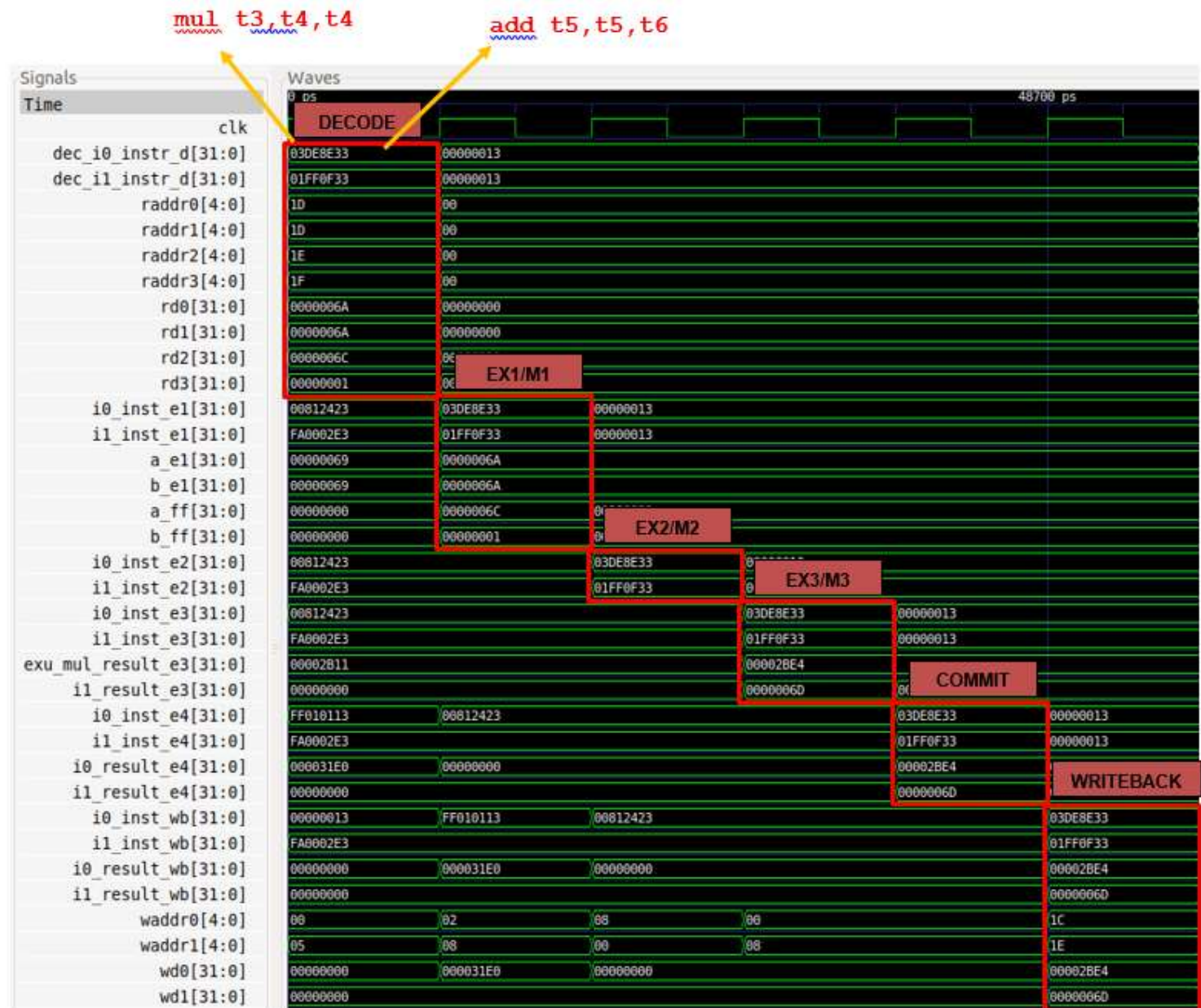


RVfpga Lab 11: Análise da Simulação

- **FC1:** Calcula o endereço da instrução `mul`:
 - `ifc_fetch_addr_f1_ext = 0x000000F0`
- **FC2:** Extrai duas instruções (mostradas a vermelho) do pacote de 128 bits da Memória de Instrução:
 - `ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33`
- **Align:** As duas instruções são extraídas e distribuídas às duas vias do SweRV EH1.
 - Via 0: `ifu_i0_instr = 0x03DE8E33` (instrução `mul`)
 - Via 1: `ifu_i1_instr = 0x01FF0F33` (instrução `add`)

Simulação:

- Descodificação
- EX1/2/3
- Commit
- Writeback



RVfpga Lab 11: Análise da Simulação

- **Decode:** Os operandos das instruções são lidos do Register File e fornecidos aos canais Mult e I1.
- **EX1/2/3 e Commit:** A adição e a multiplicação são computadas.
 - $i0_result_e4 = exu_mul_result_e3 = 0x6A * 0x6A = 0x2BE4$
 - $i1_result_e4 = i1_result_e3 = 0x6C + 0x01 = 0x6D$
- **WriteBack:** Os resultados são escritos de volta para o Register File.
 - $waddr0 = 0x1C$ $wd0 = 0x2BE4$
 - $waddr1 = 0x1E$ $wd1 = 0x6D$

RVfpga Lab 11: Contadores Internos (Hardware)

- Os contadores internos são um conjunto de registos de uso especial incluídos na maioria dos processadores atuais para registar as métricas apresentadas na tabela.

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

RVfpga Lab 11: Utilização dos Contadores de Desempenho via PSP da Western Digital

```
#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);
}
```

RVfpga Lab 11: Tarefas - Exemplos

- **TAREFA.** O Register File é implementado no módulo `dec_gpr_ctl` e é instanciado no módulo `dec`. Analise o código Verilog e a simulação dos principais sinais do módulo `dec_gpr_ctl` para compreender o seu funcionamento.
- **TAREFA.** Execute o programa da figura 13 na placa Nexys A7 como explicado no GSG. Medir outros eventos nos Contadores de Hardware para este programa.

Lab 12:

Instruções

Aritmética/Lógica:

A Instrução add

RVfpga Lab 12: Introdução

- Este laboratório analisa o fluxo das instruções aritméticas e lógicas através do pipeline do SweRV EH1, focando na instrução de soma.
- Duas secções:
 - **Análise básica** da instrução `add`
 - **Análise avançada** da instrução `add`
- As duas análises utilizam o mesmo programa de exemplo, mostrado na página seguinte.

RVfpga Lab 12: Programa de Exemplo

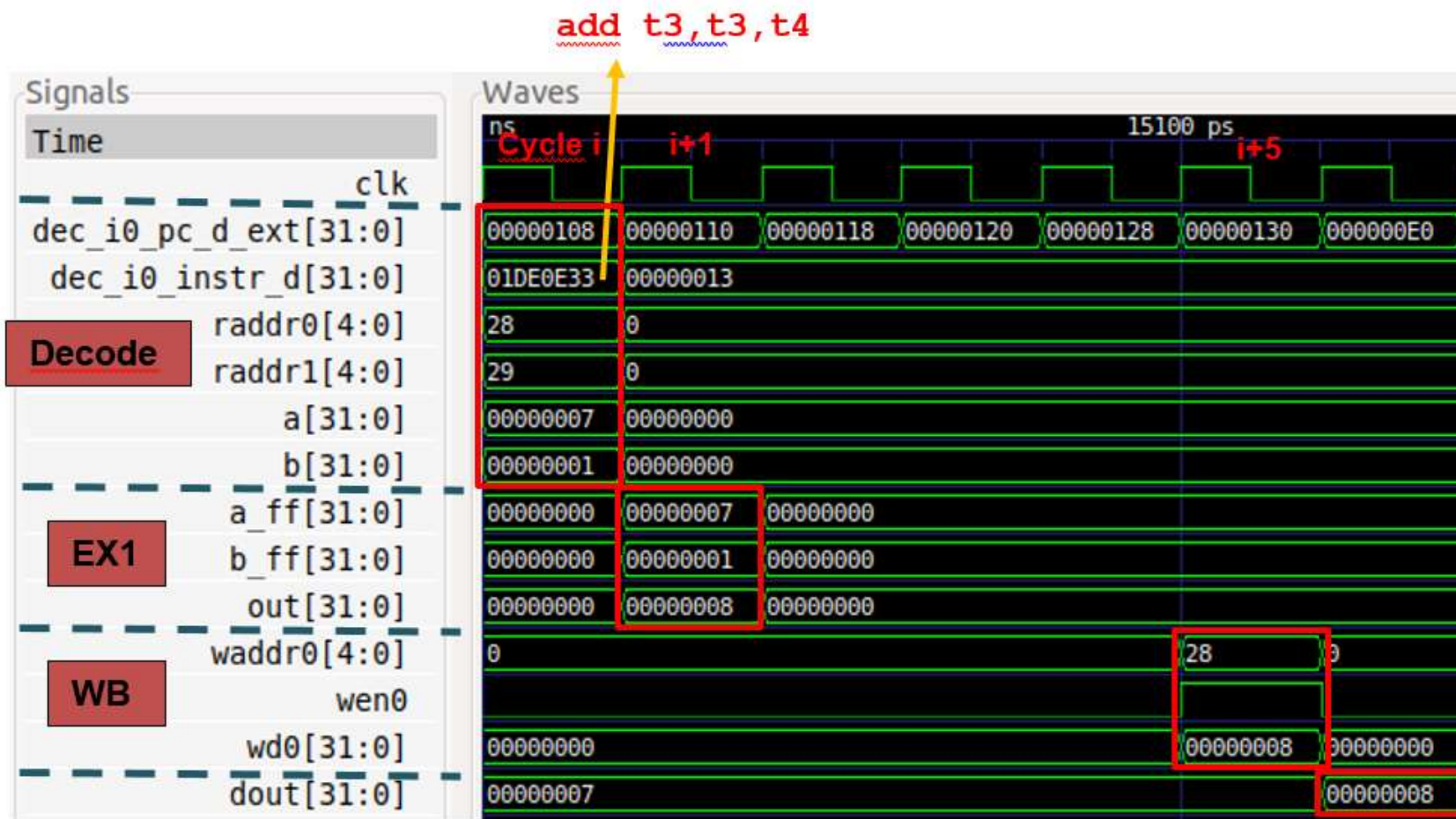
```
.globl main
main:

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

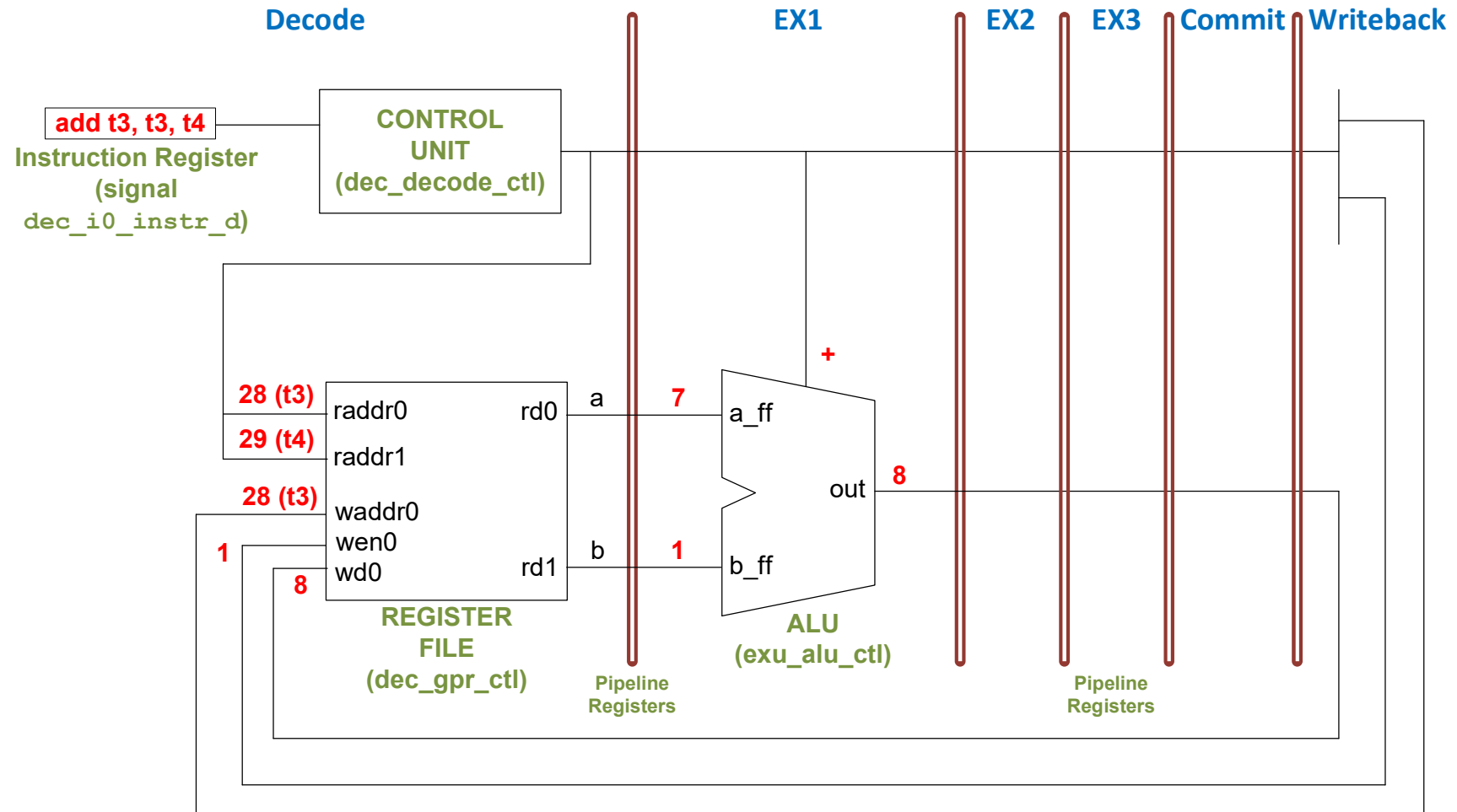
REPEAT:
    INSERT_NOPS_10
    add t3, t3, t4     # t3 = t3 + t4
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repete o ciclo

.end
```

RVfpga Lab 12: Análise Básica - Simulação



RVfpga Lab 12: Análise Básica – Pipeline SweRV EH1



RVfpga Lab 12: Análise Básica - Simulação

- **Ciclo i: Decode:** O sinal `dec_i0_instr_d` contém a instrução da máquina de 32 bits `0x01DE0E33`. No RISC-V, os campos para a instrução `add` são: `00 | rs1 | 000 | rd | 0110011`

Neste andar, **os sinais de controlo** são gerados e o **Register File** é **lido**. Além disso, os operandos são propagados para o canal **I0**.

- **Ciclo i+1: EX1:** A instrução de `add` é **executada**. O resultado da adição é fornecido como uma saída da ALU no sinal `out = 8`.
- **Ciclo i+5: Writeback:** O resultado da adição é **escrito de volta** para o Register File: `wd0 = 0x8`, `wen0 = 1` e `waddr0 = 0x28`

RVfpga Lab 12: Análise Avançada

- A figura na página seguinte mostra um diagrama detalhado da instrução de adição a atravessar o canal I0.

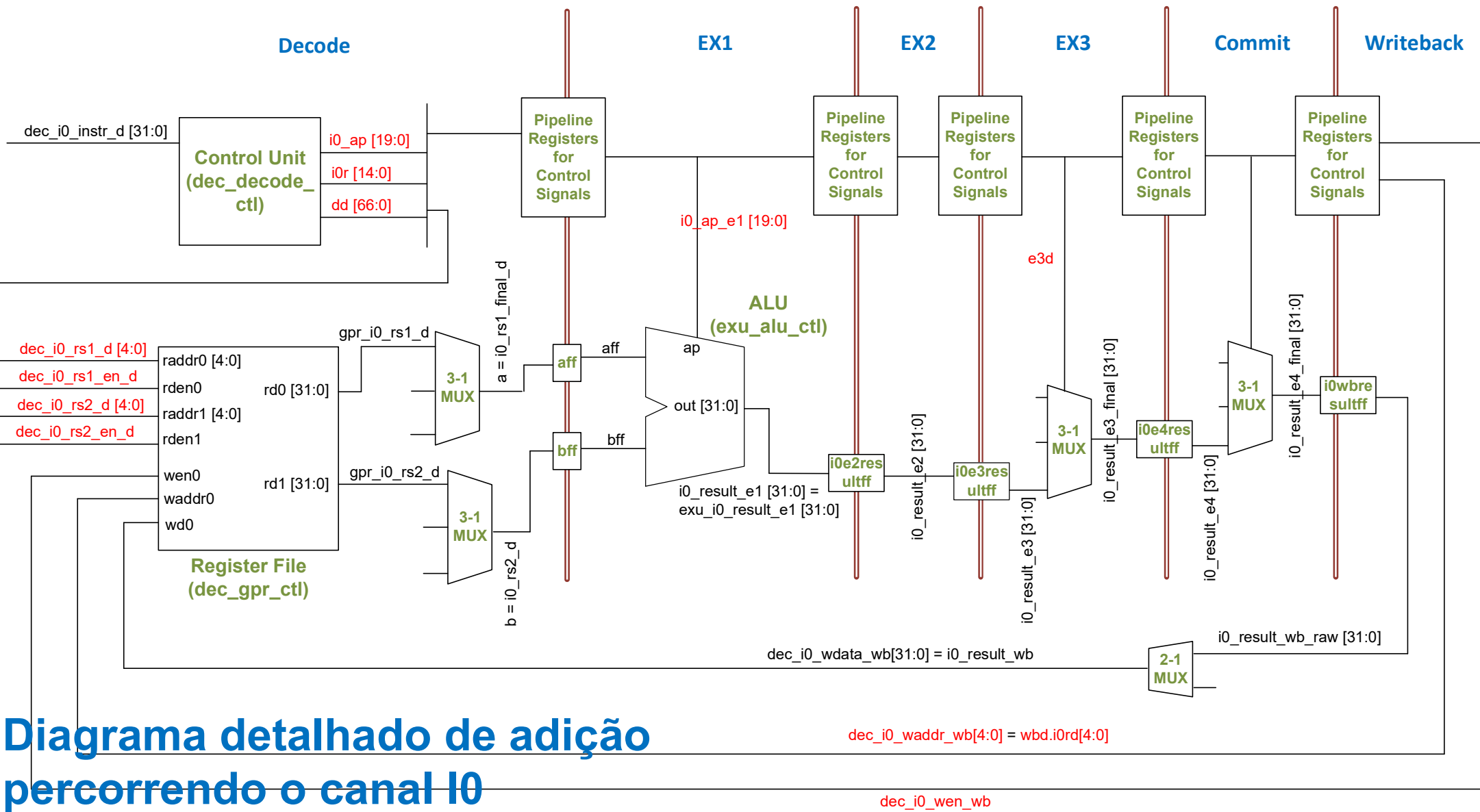


Diagrama detalhado de adiç o
percorrendo o canal i0

RVfpga Lab 12: Tarefas e Exercícios - Exemplos

- **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.
- **TAREFA.** Simule uma sub-instrução semelhante à da Figura 7.
- **TAREFA.** Analise a implementação Verilog do somador/subtrator implementado no módulo `exu_alu_ctl`.
- **TAREFA.** No código Verilog, analise a forma como os sinais `wen0` e `waddr0` são gerados no andar de Decode e propagados para o andar de Writeback.
- **Exercícios 1, 3, 4, 5.** Efetue uma análise semelhante à apresentada neste laboratório para outras instruções, tais como: `and`, `or`, `xor`, `srl`, `sra`, `sll`, `slt`, `sltu`, `addi`, `andi`, `ori`, `xori`, `srli`, `srai`, `slli`, `slti`, e `sltui`.
- **Exercícios 2, 6, 7.** Exercícios baseados em exercícios dos dois principais livros de referência:
 - “Computer Organization and Design – RISC-V Edition”, por Patterson & Hennessy.
 - “Digital Design and Computer Architecture: RISC-V Edition” por S. Harris and D. Harris.

Lab 13:

Instruções de Memória:

Instruções lw e sw

RVfpga Lab 13: Introdução

- Lab 13 analisa a leitura e a escrita na memória.
- Três partes:
 - **Leituras de baixa latência:** Estuda o pipe de leitura/escrita na leitura da DCCM de baixa latência (não pára o processador).
 - **Escritas de baixa latência :** Estuda escritas para a DCCM.
 - **Leituras e Escritas de elevada latência :** Repetir análises anteriores ao ler/escrever na memória principal DDR disponível na placa Nexys A7.

RVfpga Lab 13: Leituras – Programa de Exemplo

```
.globl main
```

```
.section .midccm
```

```
A: .space 8
```

```
.text
```

```
main:
```

```
# Register t3 = x28 (register 28)
```

```
la t0, A      # t0 = addr(A)
```

```
li t1, 0x2    # t1 = 2
```

```
sw t1, (t0)   # A[0] = 2
```

```
add t1, t1, 6  # t1 = 8
```

```
sw t1, 4(t0)  # A[1] = 8
```

```
INSERT_NOPS_9
```

```
REPEAT:
```

```
INSERT_NOPS_1
```

```
lw t1, (t0)
```

```
INSERT_NOPS_9
```

```
INSERT_NOPS_4
```

```
lw t1, 4(t0)
```

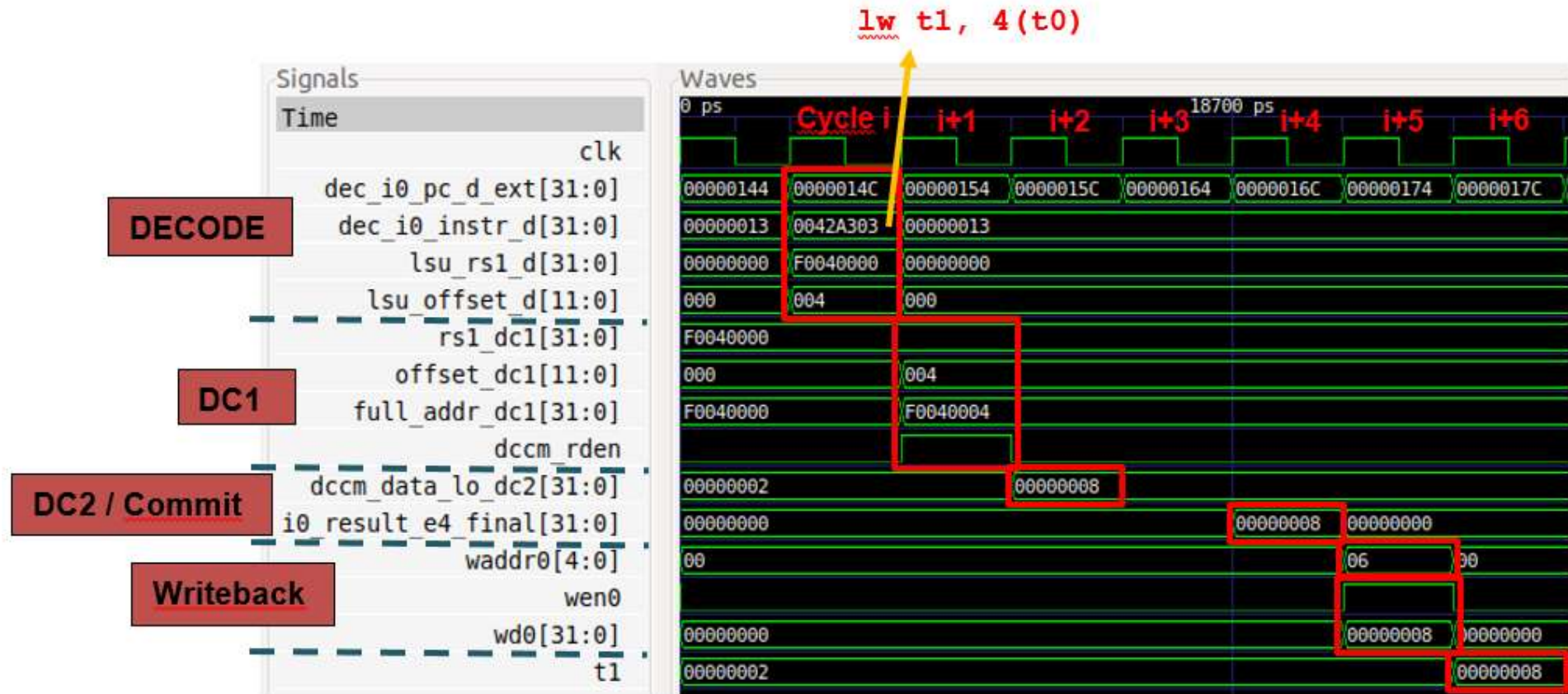
```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

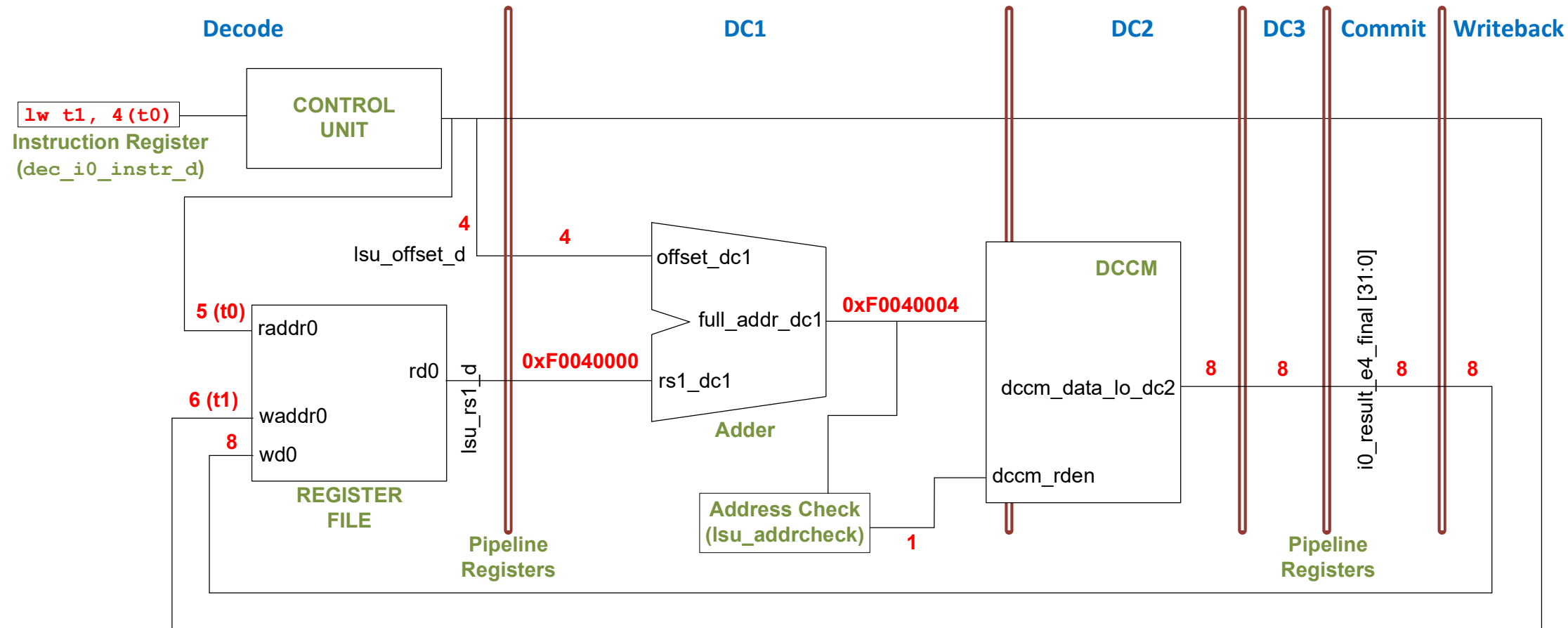
```
beq zero, zero, REPEAT # Repeat the loop
```

```
.end
```

RVfpga Lab 13: Leituras de Baixa Latência - Simulação



RVfpga Lab 13: Leituras de Baixa Latência – Pipeline SweRV EH1



RVfpga Lab 13: Leituras de Baixa Latência - Análise

- **Ciclo i:** **Descodificação:** gera os sinais de controlo e lê operandos:
 - $t0 = 0xF0040000$
 - $\text{Offset} = 0x004$
- **Ciclo i+1:** **DC1:**
 - Calcula o endereço: $\text{full_addr_dc1} = 0xF0040004$
 - Encontra a região de memória do acesso → confirma dccm_rden
- **Ciclo i+2:** **DC2:** o DCCM é lido → $\text{dccm_data_lo_dc2} = 0x8$
- **Ciclo i+5:** **Writeback:** O valor lido a partir da memória é escrito no Register File:
 - $\text{wd0} = 0x8$
 - $\text{wen0} = 1$
 - $\text{waddr0} = 0x6$

RVfpga Lab 13: Análise Detalhada de Leituras de Baixa Latência

- A figura no slide seguinte mostra um diagrama detalhado dos principais elementos que uma instrução l_w atravessa durante a sua execução através do canal I0.
- Isto já foi ilustrado no Laboratório 11, mas a nova figura centra-se apenas no canal LSU e fornece detalhes relacionados com a instrução l_w .
- O documento para o Laboratório 13 fornece explicações detalhadas, não incluídas aqui, sobre cada etapa mostrada na figura para a execução da instrução l_w .

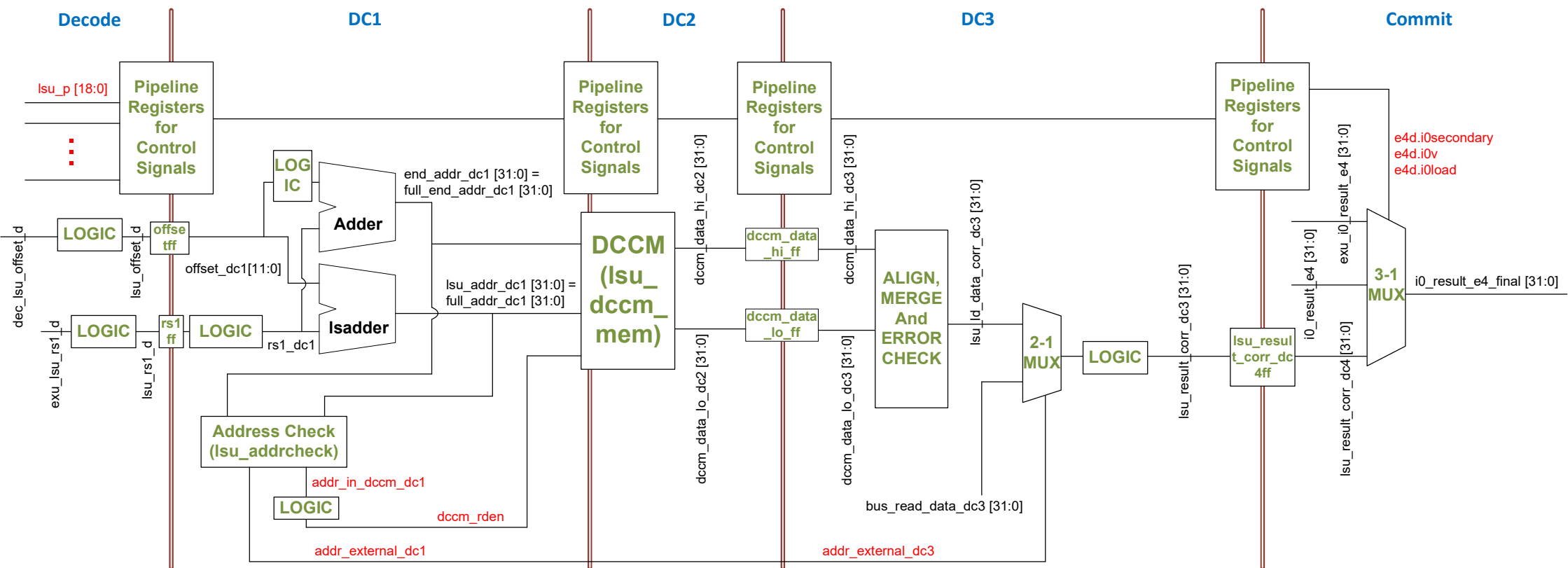


Diagrama detalhado do 1w ao atravessar o canal I0

RVfpga Lab 13: Escritas – Programa de Exemplo

```
.globl main
```

```
.section .midccm
```

```
A: .space 4000
```

```
.text
```

```
main:
```

```
la t0, A           # t0 = addr(A)
```

```
li t1, 0x2         # t1 = 2
```

```
li t2, 1000        # t2 = 1000
```

```
INSERT_NOPS_2
```

```
REPEAT:
```

```
sw t1, (t0)
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

```
lw t1, (t0)
```

```
INSERT_NOPS_10
```

```
add t1,t1,t1
```

```
add t0,t0,0x04
```

```
add t2,t2,-1
```

```
INSERT_NOPS_10
```

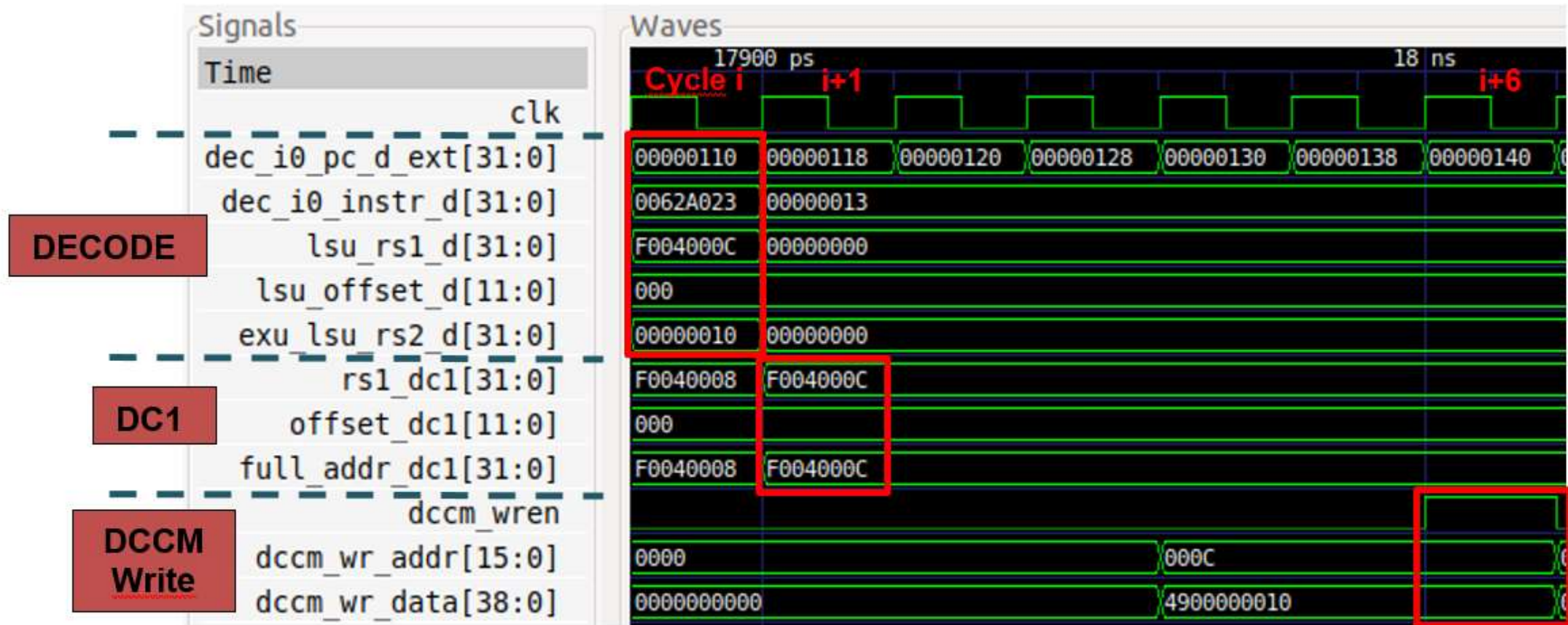
```
bne t2, zero, REPEAT # Repeat the loop
```

```
nop
```

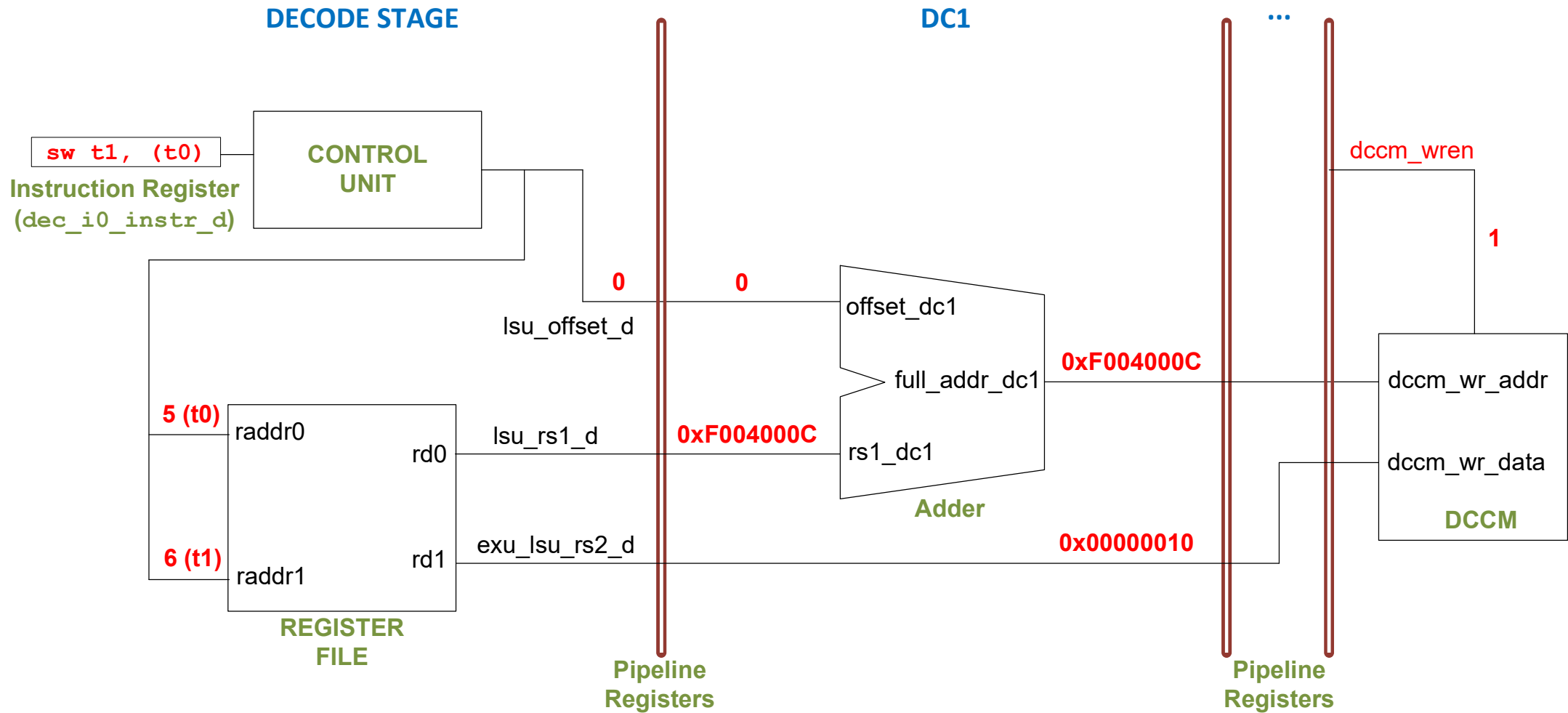
```
nop
```

```
.end
```

RVfpga Lab 13: Escrita de Baixa-Latência – Simulação



RVfpga Lab 13: Escrita de Baixa-Latência – pipeline do SweRV EH1



RVfpga Lab 13: Análise Básica da Escrita - Simulação

- **Ciclo i: Decode:** gera os sinais de controlo e lê os operandos:
 - $t0 = 0xF004000C$
 - Offset = 0x000
 - $t1 = 0x10$
- **Ciclo i+1: DC1:**
 - Calcula o endereço : $full_addr_dc1 = 0xF004000C$
- **Ciclo i+6: Escrita DCCM:**
 - $dccm_wr_addr = 0x000C$
 - $dccm_wr_data = 0x10$

RVfpga Lab 13: Leitura da Memória Externa

- A figura no diapositivo seguinte mostra o caminho principal que a instrução \perp_w percorre para ler a Memória Principal.
- O processador deve parar à espera de dados da Memória Externa.
- A Memória Externa é acedida através do barramento AXI, que fornece o endereço ao controlador DRAM Lite e, alguns ciclos mais tarde, alinha e envia os dados pedidos para a etapa DC3.
- O multiplexer 2:1 na etapa DC3 seleciona os dados provenientes da Memória Externa, em vez dos dados provenientes do DCCM.

DC1 STAGE

Delay due to
accessing External
Memory

DC3 STAGE

COMMIT STAGE

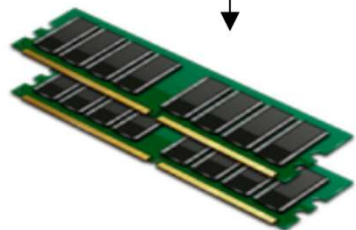
Pipeline
Registers
for
Control
Signals

Pipeline
Registers
for
Control
Signals

Pipeline
Registers
for
Control
Signals

External Memory
accessed through AXI
Bus
(Isu_bus_intf)

Lite DRAM
Controller



end_addr_dc1 [31:0] =
full_end_addr_dc1 [31:0]

Isu_addr_dc1 [31:0] =
full_addr_dc1 [31:0]

addr_external_dc1

bus_read_data_dc3 [31:0]

addr_external_dc3

Isu_id_data_corr_dc3 [31:0]

2-1
MUX

LOGIC

Isu_result_corr_dc3 [31:0]

Isu_resul
t_corr_dc
4ff

exu_i0_result_e4 [31:0]

i0_result_e4 [31:0]

Isu_result_corr_dc4 [31:0]

3-1
MUX

i0_result_e4_final [31:0]

e4d.i0secondary
e4d.i0v
e4d.i0load

RVfpga Lab 13: Memória Externa - Programa de Exemplo

```
.globl main
```

```
.data
```

```
D: .word 3,5,6,8,7,10,12,2,1,4,11,9
```

```
.text
```

```
main:
```

```
li t2, 0x020
```

```
csrrs t1, 0x7F9, t2
```

```
la t4, D
```

```
li t5, 12
```

```
li t6, 0x0
```

```
INSERT_NOPS_1
```

```
REPEAT:
```

```
lw t3, (t4)
```

```
add t5, t5, -1
```

```
INSERT_NOPS_10
```

```
add t6, t3, t6
```

```
add t4, t4, 4
```

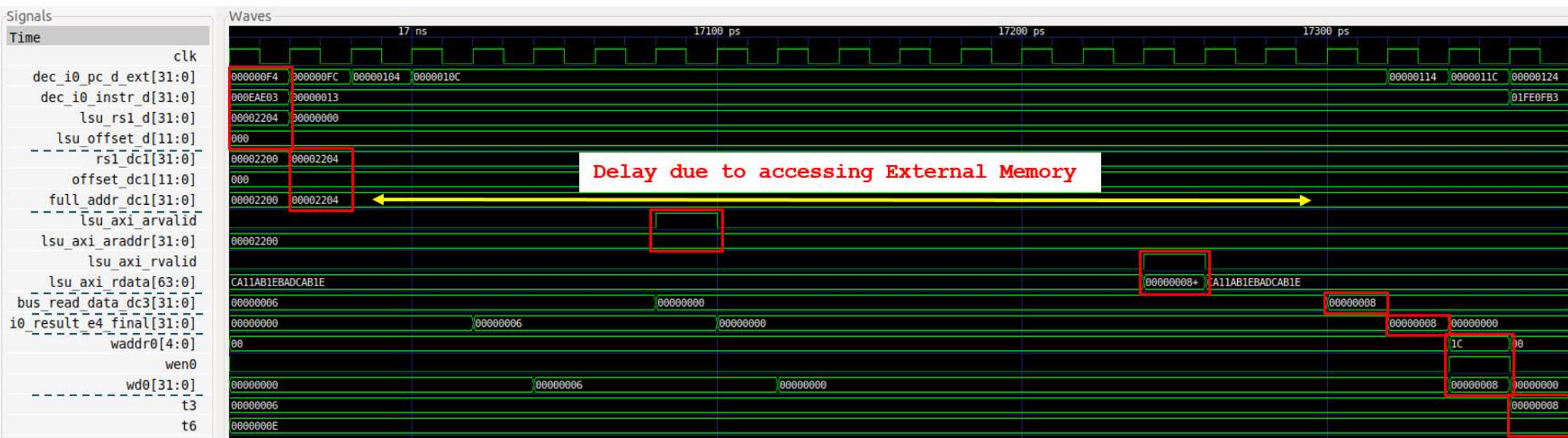
```
INSERT_NOPS_9
```

```
bne t5, zero, REPEAT # Repeat the loop
```

```
INSERT_NOPS_4
```

```
.end
```

RVfpga Lab 13: Memória Externa - Simulação



RVfpga Lab 13: Memória Externa – Análise

- No andar de Decode, o endereço, que na quarta iteração do exemplo é 0x00002204.
- Depois, o endereço é enviado para a memória externa através do barramento AXI:
 - `lsu_axi_arvalid = 1`
 - `lsu_axi_araddr = 0x00002200`
- Alguns ciclos mais tarde, a memória externa devolve dados de 64 bits lidos através do barramento AXI
 - `lsu_axi_rdata = 0x0000000800000006`
 - `lsu_axi_rvalid = 1`
- Finalmente, os dados de 32 bits pedidos são extraídos dos dados de 64 bits, inseridos no caminho principal do pipeline, e escritos no Register File.

RVfpga Lab 13: Tarefas - Exemplos

- **TAREFA.** Incluir o sinal `lsu_p` na simulação da Figura 4 e analisar os seus bits.
- **TAREFA.** Analisar no código Verilog o caminho seguido pelas duas entradas da LSU (`exu_lsu_rsl_d` e `dec_lsu_offset_d`) das origens onde são provenientes. Este processo envolve vários módulos: `dec`, `exu`, `lsu`.
- **TAREFA.** Analise a implementação dos dois somadores do andar DC1, que estão instanciados no módulo `lsu_lsc_ctl`.
- **TAREFA.** No programa da Figura 2, experimente diferentes tamanhos de acesso (byte, half-word) e acessos não alinhados. Para isso, altere o deslocamento ou o tipo de acesso de `lw` para `lb` (load byte) ou `lh` (load half-word). Por exemplo, se alterar o *offset* de 4 para 3, a instrução `load word` efectua um acesso não alinhado aos 32 bits que começam no endereço `0xF0040003`, como mostra a Figura 8. Analisar o valor dos sinais `lsu_addr_dc1[31:0]` (ou `full_addr_dc1[31:0]`) e `end_addr_dc1[31:0]` nas diferentes situações.
- **TAREFA.** Analisar as escritas não alinhadas com o DCCM, bem como as escritas de palavras menores: *store byte* (`sb`) ou *store half-word* (`sh`).
- **TAREFA.** Também pode ser interessante analisar a implementação do barramento AXI para aceder ao controlador DRAM, para o qual pode inspeccionar o módulo `lsu_bus_intf`.

Lab 14:

Conflitos Estruturais

RVfpga Lab 14: Introdução

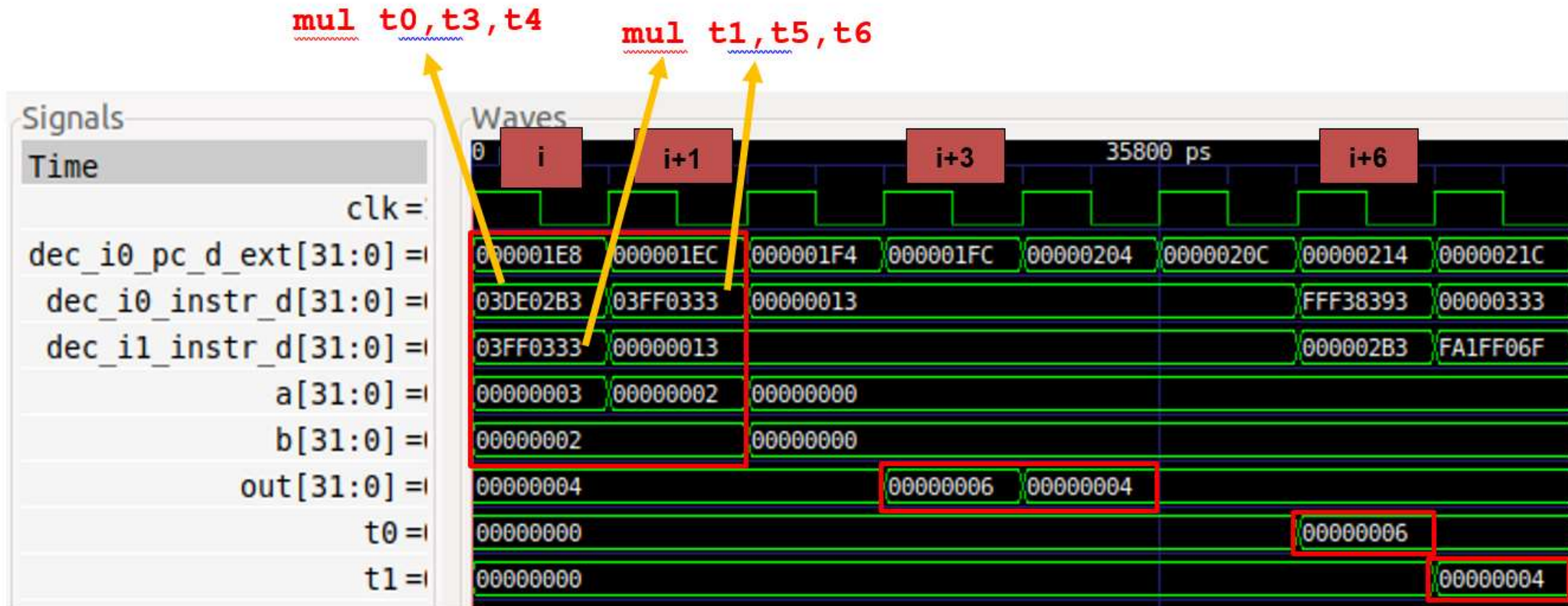
- O Lab 14 ilustra **dois conflitos estruturais** (que têm diferentes compensações entre desempenho e custo).
 - **Conflito de unidade:** duas instruções `mul` chegam ao andar de Descodificação no mesmo ciclo. O multiplicador é encadeado, pelo que a segunda instrução `mul` é atrasada apenas por um ciclo. O custo do hardware e a degradação do desempenho (apenas um ciclo) são baixos.
 - **Conflito de portas de escrita no Register File:** Três instruções chegam ao andar de Writeback no mesmo ciclo, sendo uma delas uma leitura não bloqueante executada vários ciclos antes. O SweRV EH1 tem três (em vez de duas) portas de escrita. O conflito estrutural é evitado (resultando em nenhuma perda de desempenho), mas tem um elevado custo de hardware devido à porta extra do Register File.
 - Note-se que a instrução `div` também pode causar conflitos, o que é discutido no Apêndice do laboratório.

RVfpga Lab 14: 2 Instruções mul – Programa de Exemplo

```
.globl Test_Assembly
Test_Assembly:
li t2, 0xFFFF
li t3, 0x3
li t4, 0x2
li t5, 0x2
li t6, 0x2
REPEAT:
    beq t2, zero, OUT    # Permanecer no ciclo?
    INSERT_NOPS_9
    mul t0, t3, t4        # t0 = t3 * t4
    mul t1, t5, t6        # t1 = t5 * t6
    INSERT_NOPS_9
    add t2, t2, -1
    add t0, zero, zero
    add t1, zero, zero
    j REPEAT
OUT:
.end
```



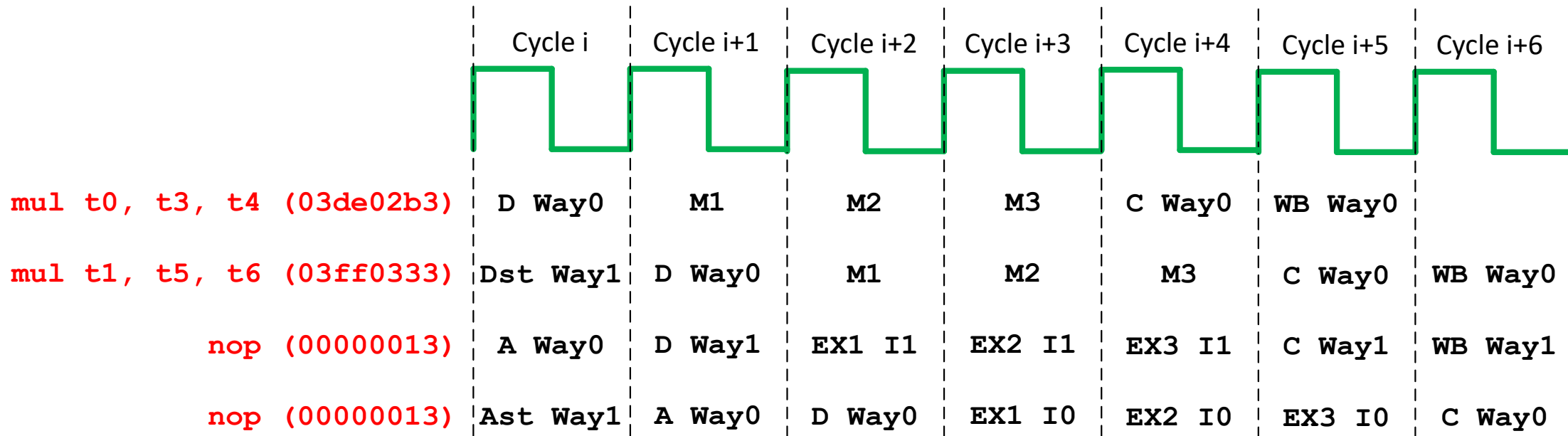
RVfpga Lab 14: 2 Instruções mul – Simulação



RVfpga Lab 14: 2 Instruções mul – Análise

- **Ciclo i:** As duas instruções `mul` chegam ao andar de Descodificação no mesmo ciclo. Um Conflito Estrutural impede que a segunda instrução `mul` avance.
- **Ciclo i+1:** A primeira instrução `mul` executa no primeiro andar do multiplicador encadeado (M1), enquanto a segunda instrução `mul` espera no andar de Decodificação.
- **Ciclo i+2:** A primeira instrução `mul` executa no segundo andar do multiplicador encadeado (M2) e a segunda `mul` executa na primeira etapa (M1).
- **Ciclo i+3:** A primeira instrução `mul` obtém o resultado : `out = 0x6`.
- **Ciclo i+4:** A segunda instrução `mul` obtém o resultado : `out = 0x4`.
- **Ciclo i+6:** O Register File é actualizado com o resultado da primeira `mul` (`t0 = 0x6`).
- **Ciclo i+7:** O Register File é actualizado com o resultado da segunda `mul` (`t1 = 0x4`).

RVfpga Lab 14: 2 Instruções mul – Diagrama



RVfpga Lab 14: 3 Escritas Simultâneas – Programa de Exemplo

REPEAT:

lw x28, (x29)

add x30, x30, -1

add x1, x1, 1

add x31, x31, 1

add x3, x3, 1

add x4, x4, 1

add x5, x5, 1

add x6, x6, 1

add x7, x7, 1

add x8, x8, 1

add x9, x9, 1

add x10, x10, 1

add x11, x11, 1

add x12, x12, 1

add x13, x13, 1

add x14, x14, 1

add x15, x15, 1

add x16, x16, 1

add x17, x17, 1

add x18, x18, 1

add x19, x19, 1

add x20, x20, 1

add x21, x21, 1

add x22, x22, 1

add x23, x23, 1

add x24, x24, 1

add x25, x25, 1

add x26, x26, 1

add x27, x27, 1

add x31, x31, 1

add x3, x3, 1

add x4, x4, 1

add x5, x5, 1

add x6, x6, 1

add x25, x25, 1

add x26, x26, 1

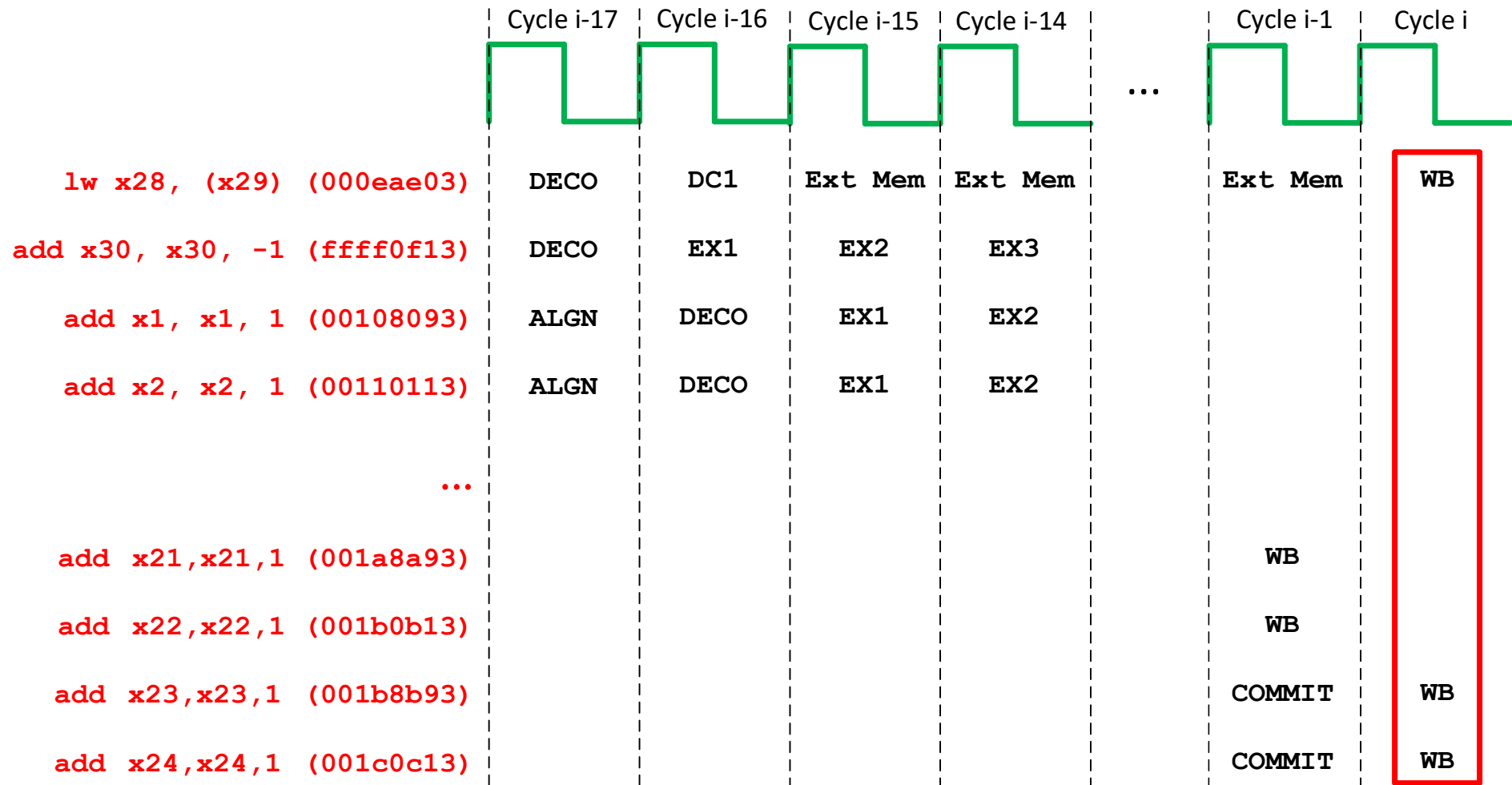
add x27, x27, 1

bne x30, zero, REPEAT

RVfpga Lab 14: 3 Escritas Simultâneas – Análise

- **Ciclo i-17:** A instrução `lw` está no andar de Descodificação.
- **Ciclo i-16:** O endereço efetivo da memória é calculado e enviado para a Memória Externa através do barramento AXI. A instrução de leitura `load` aguarda vários ciclos para que a Memória Externa forneça os dados.
- **Ciclo i-5:** As duas instruções `add` conflituosas são descodificadas.
- **Ciclo i:** A instrução `lw` e as duas instruções `add` em conflito prosseguem para o andar de Writeback, onde escrevem no Register File, o que é possível porque o ficheiro de registo tem três portas de escrita.

RVfpga Lab 14: 3 Escritas Simultâneas – Diagrama



RVfpga Lab 14: Tarefas e Exercícios – Exemplos

- **TAREFA:** Inspeccione o código Verilog de `exu_mul_ctl` e veja como a multiplicação é calculada. Lembre-se que o RISC-V inclui 4 instruções de multiplicação (`mul`, `mulh`, `mulhsu` e `mulhu`), e todas elas devem ser suportadas pelo hardware.
- **TAREFA :** Remova as instruções `nop` incluídas no ciclo da Figura 1 e meça os diferentes eventos (ciclos, instruções/multiplicação cometidas, etc.) usando os Contadores de Desempenho disponíveis no SweRV EH1, como explicado no Laboratório 11. O número de ciclos é o esperado depois de analisar a simulação da Figura 2? Justifique a sua resposta. Agora reordene o código dentro do ciclo tentando atingir a taxa de transferência ideal. Justifique os resultados obtidos no código original e no código reordenado.
- **TAREFA :** Modifique o programa da Figura 1, substituindo as duas instruções `mul` por duas instruções `lw` para o DCCM. Deverá observar um conflito estrutural análogo ao analisado nesta secção e resolvido de forma semelhante.
- **TAREFA :** Compare a simulação apresentada na Figura 6 (leitura não-bloqueante) com a simulação apresentada na Figura 14 do Lab 13 (leitura bloqueante).
- **Exercício 1.** Analise, tanto em simulação como na placa, o conflito estrutural que ocorre entre duas instruções de memória consecutivas (pode analisar qualquer combinação de duas instruções de memória consecutivas, como leituras e escritas) que chegam ao canal L/S no mesmo ciclo.
- **Exercício 2.** Este exercício é baseado no exercício 4.22 do livro “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([PaHe]).

Lab 15:

Conflitos de Datos

RVfpga Lab 15: Introdução

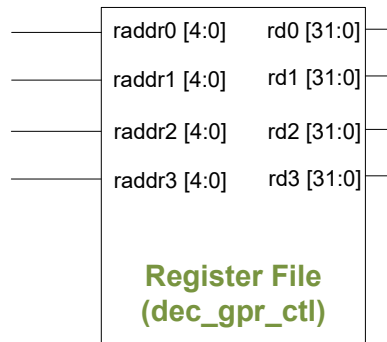
- O Lab 15 analisa como os **conflitos de dados** (*read-after-write*) RAW são resolvidos.
- Os conflitos de dados RAW são resolvidos **bloqueando** o processador ou **encaminhando** (também chamado por “forwarding” ou “bypassing”) o valor de uma instrução executada num andar posterior.
- **Dois cenários** analisados:
 - Os conflitos de dados RAW são resolvidos através do **forwarding** para o andar de Decode (utilizando vários novos multiplexers)
 - Conflitos de dados RAW resolvidos no andar de Commit usando **duas ALUs adicionais**

RVfpga Lab 15: Resolução de Conflitos de Dados por Encaminhamento

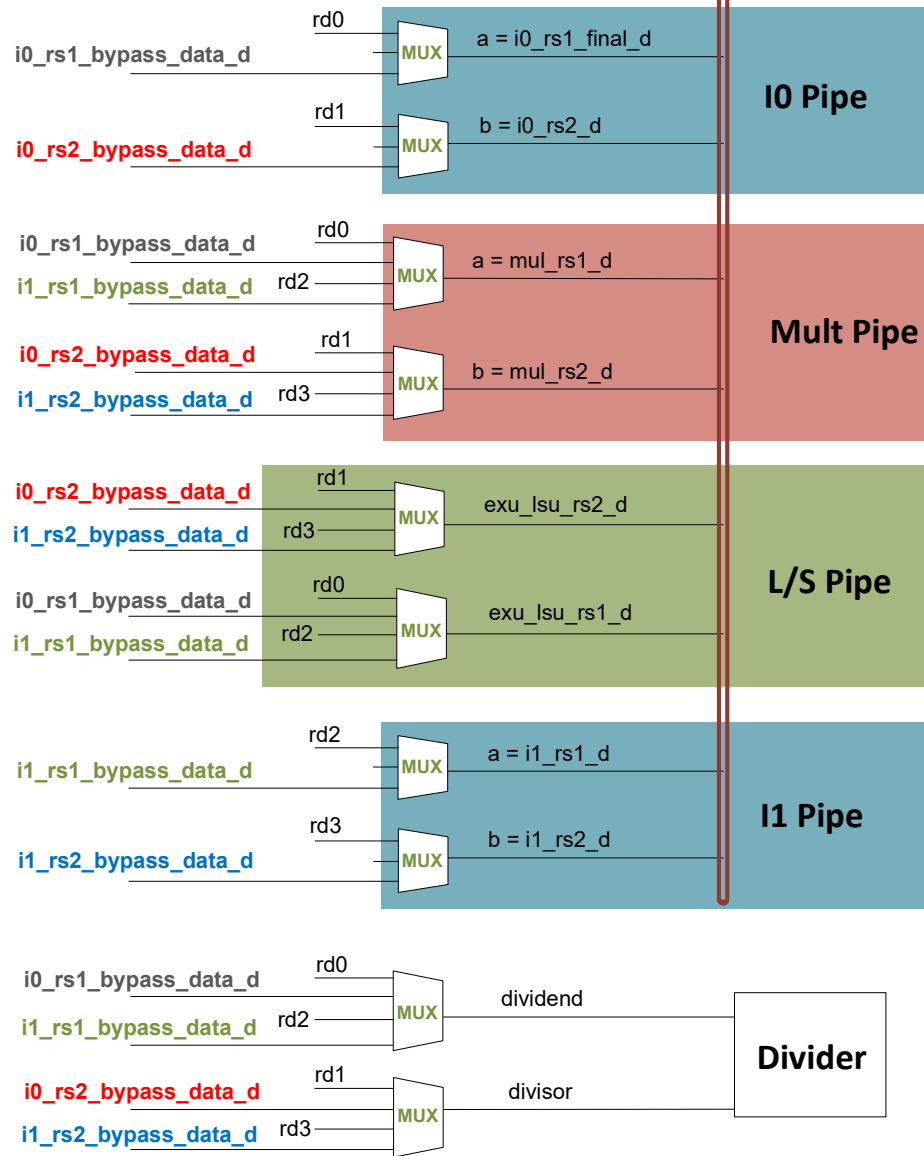
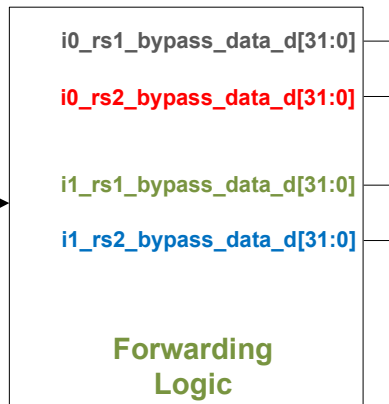
- O encaminhamento (*Forwarding*) para o andar de Decode requer a adição de Multiplexers em frente das Unidades Funcionais (ALUs, Multiplicador, e o Somador que calcula o Endereço Efetivo em DC1, etc.) para selecionar os operandos quer do Register File quer dos andares subsequentes.
- A figura no slide seguinte mostra os valores transmitidos no andar de Descodificação. A Lógica de Forwarding produz sinais de *bypass* para cada um dos dois operandos de origem em cada uma das vias.

DECODE STAGE

EX1/DC1/M1



From subsequent stages



RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding – Exemplo

```
.globl Test_Assembly
.text

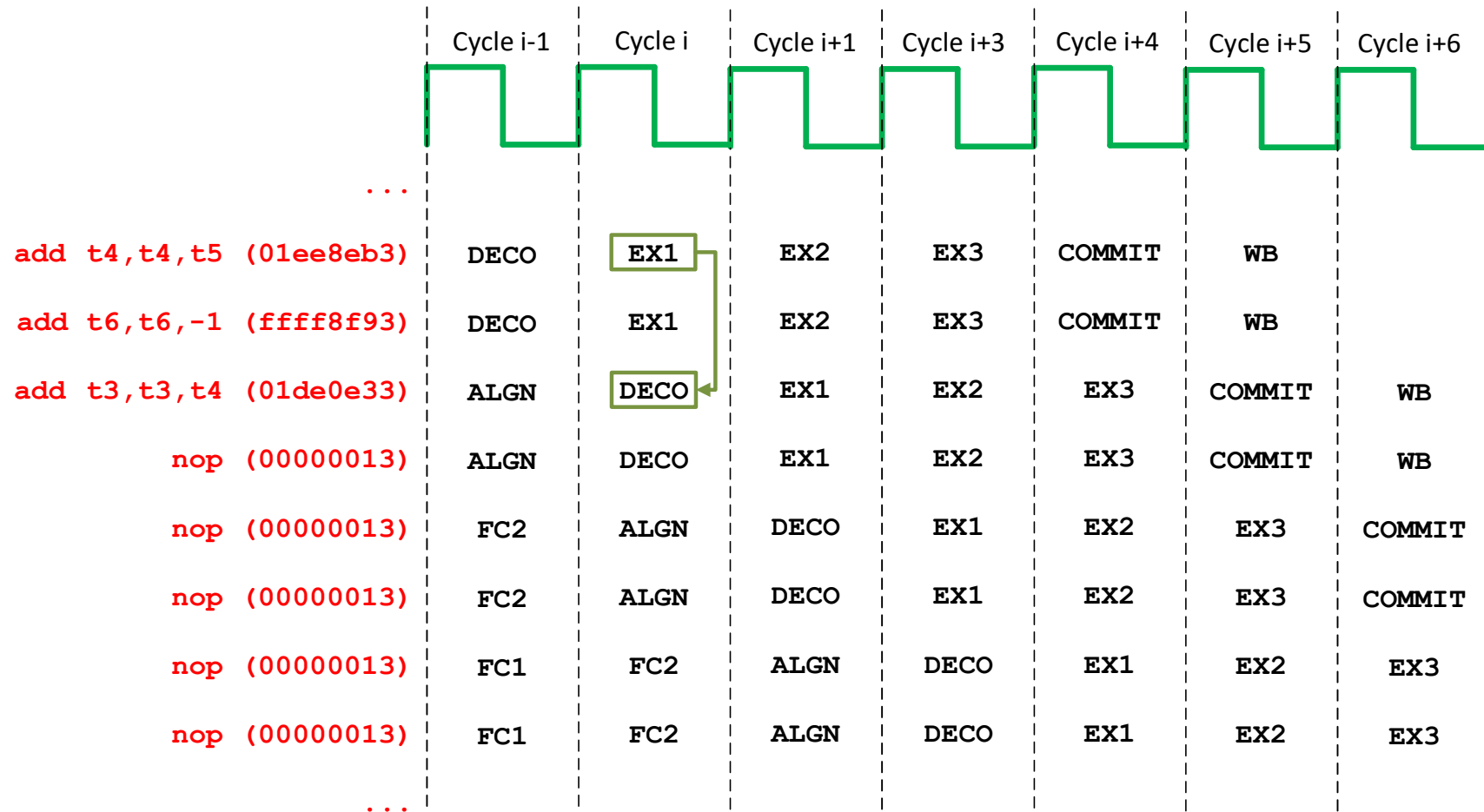
Test_Assembly:

li t3, 0x3
li t4, 0x2
li t5, 0x1
li t6, 0xFFFF

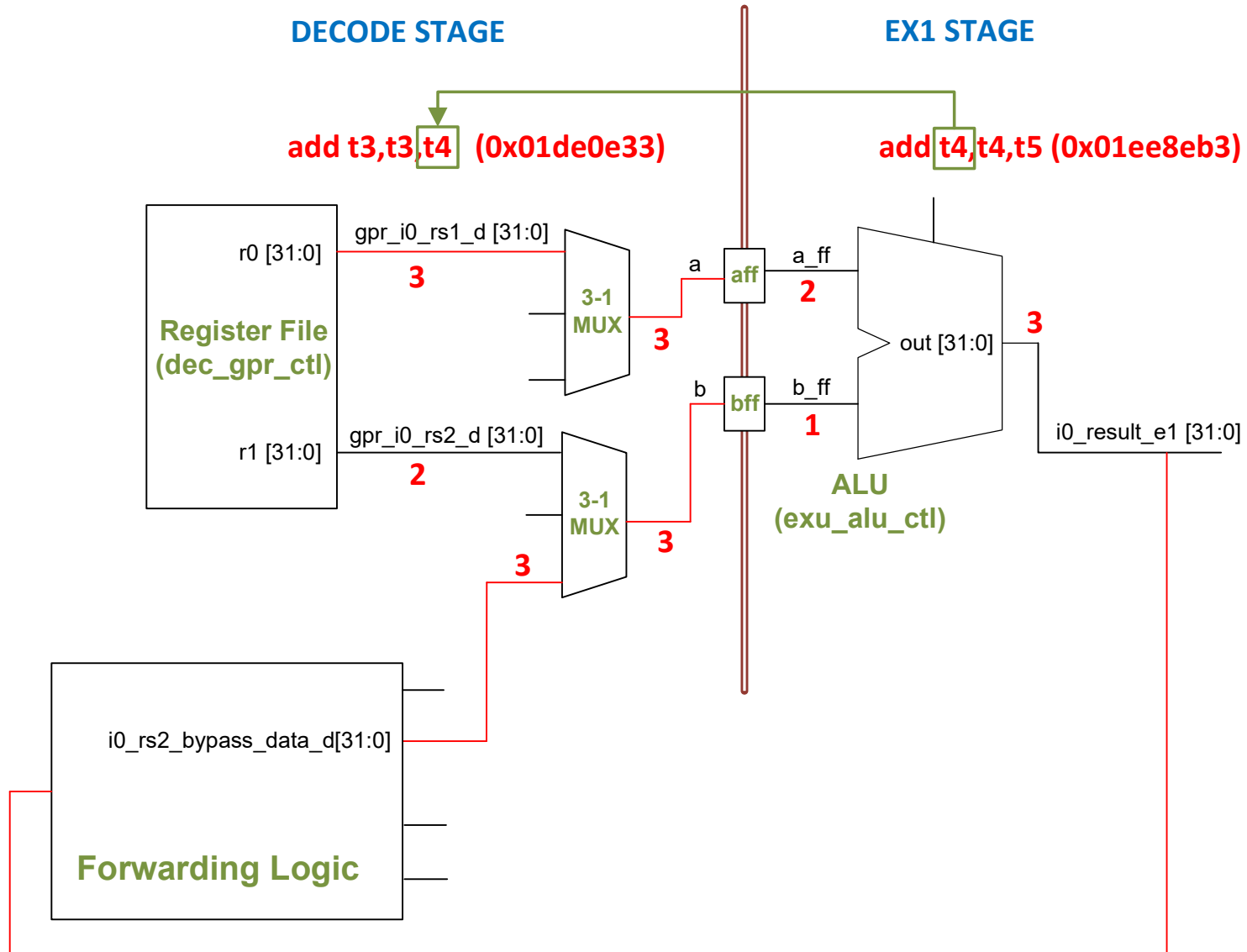
REPEAT:
    INSERT_NOPS_8
    add t4, t4, t5          # t4 = t4 + t5 (t4 = 2 + 1)
    add t6, t6, -1
    add t3, t3, t4          # t3 = t3 + t4 (t3 = 3 + 3)
    INSERT_NOPS_9
    li t3, 0x3
    li t4, 0x2    li t5, 0x1    bne t6, zero, REPEAT    # Repete o ciclo

.end
```

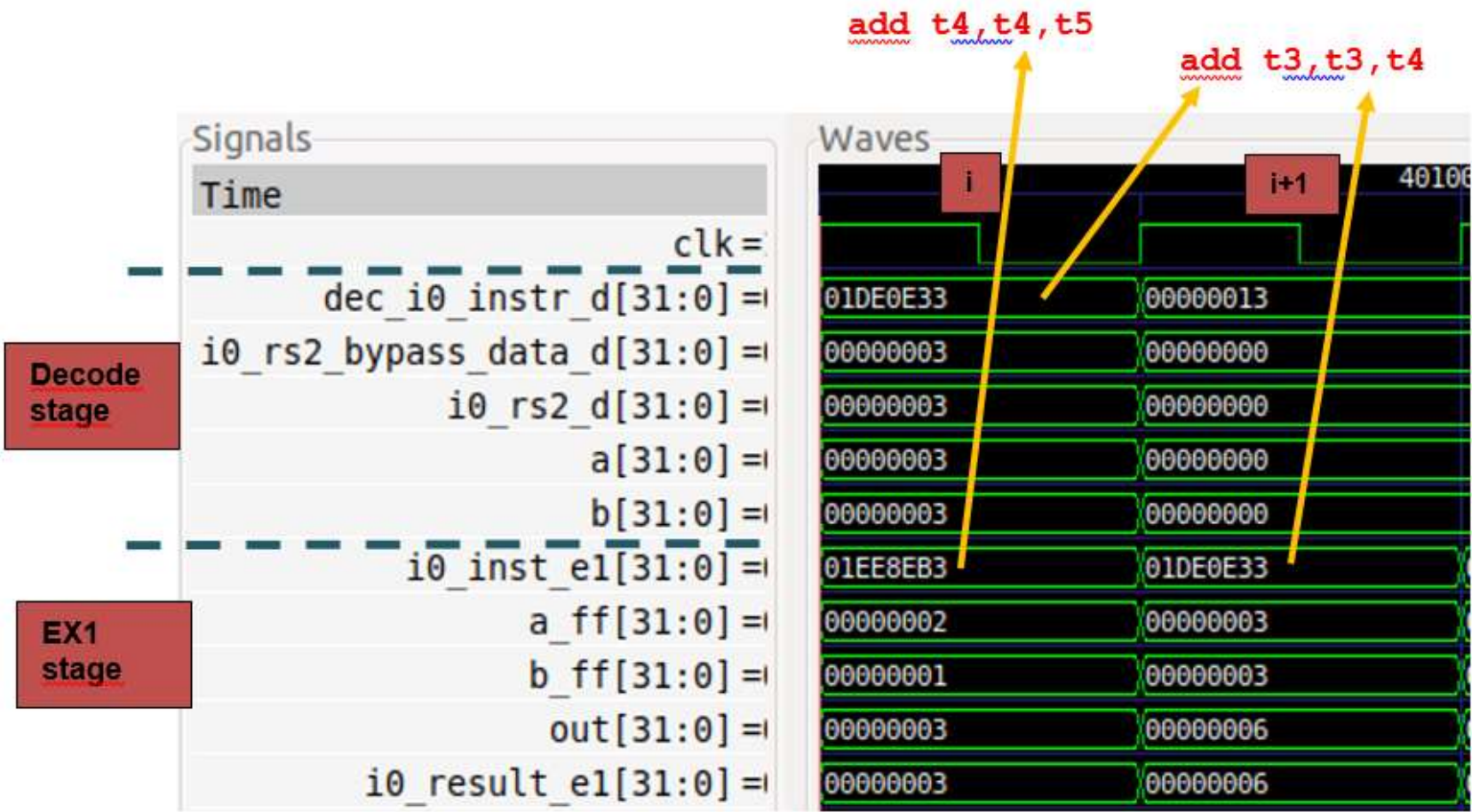
RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding – Diagrama



RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding – Pipeline – Ciclo *i*

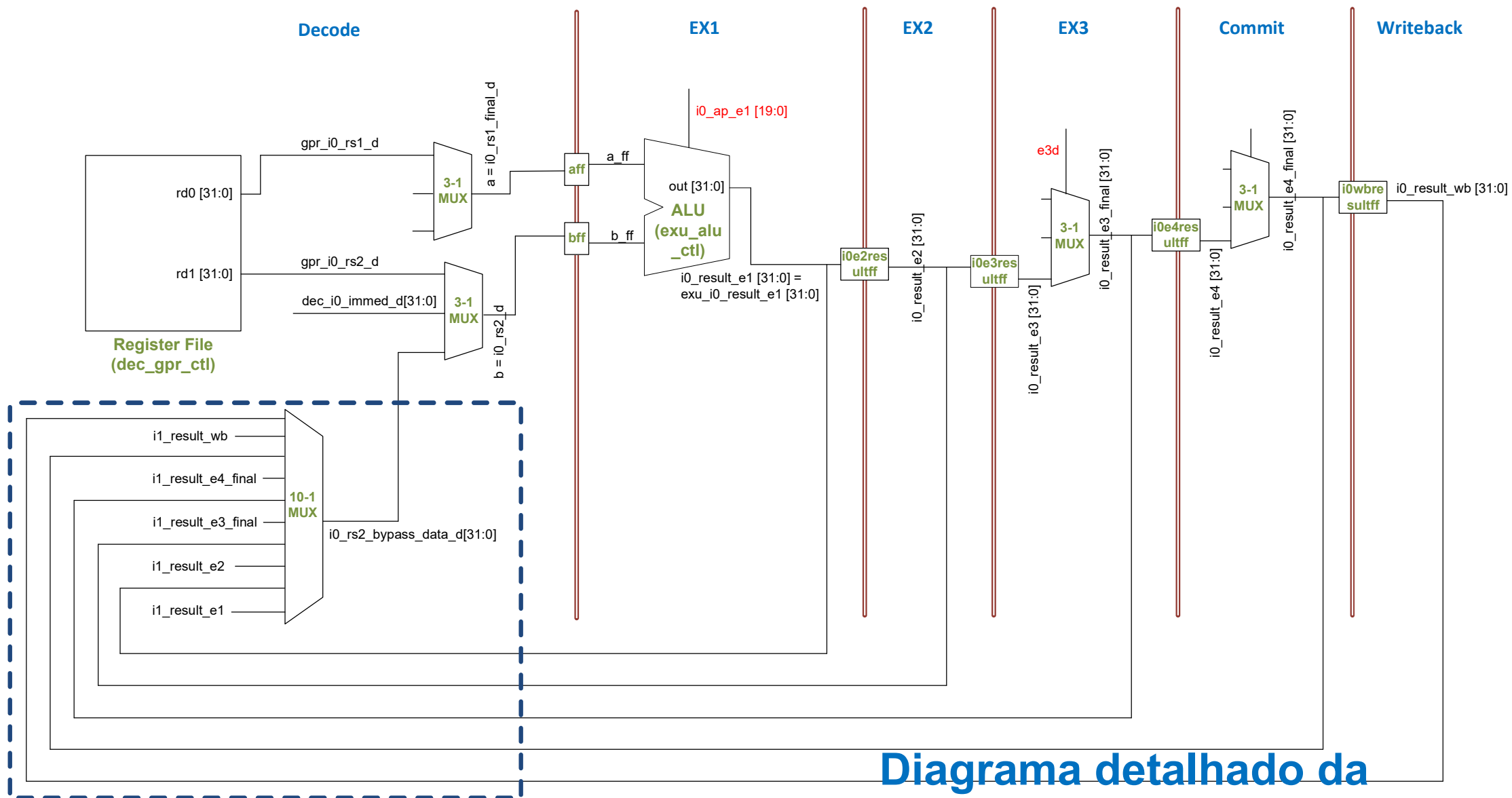


RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding – Simulação



RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding – Análise

- Instrução `add t4, t4, t5 (0x01ee8eb3)`:
 - **Ciclo *i*:** Esta instrução `add` está no andar EX1 do canal (*pipe*) I0 (`i0_inst_e1 = 0x01ee8eb3`). Calcula a seguinte adição na ALU:
 - $a_ff(2) + b_ff(1) = out(3)$
 - O resultado é enviado para a Lógica de Forwarding no andar de Decode.
- Instrução `add t3, t3, t4 (0x01de0e33)`:
 - **Ciclo *i*:** Esta instrução `add` está no andar de Decode na Via 0 (`dec_i0_instr_d = 0x01de0e33`). A Lógica de Forwarding encaminha o resultado de EX1 (`i0_result_e1`) para o andar de Decode (`i0_rs2_bypass_data_d`). Dois multiplexers 3:1 produzem os operandos, em concreto:
 - Operando a = 3 (do Register File)
 - Operando b = 3 (da saída da ALU na etapa EX1 do canal I0, através da lógica de Forwarding)
 - **Ciclo *i+1*:** Esta instrução `add` está no andar EX1 do canal I0 (`i0_inst_e1 = 0x01de0e33`). Calcula a adição correta na ALU:
 - $a_ff(3) + b_ff(3) = out(6)$

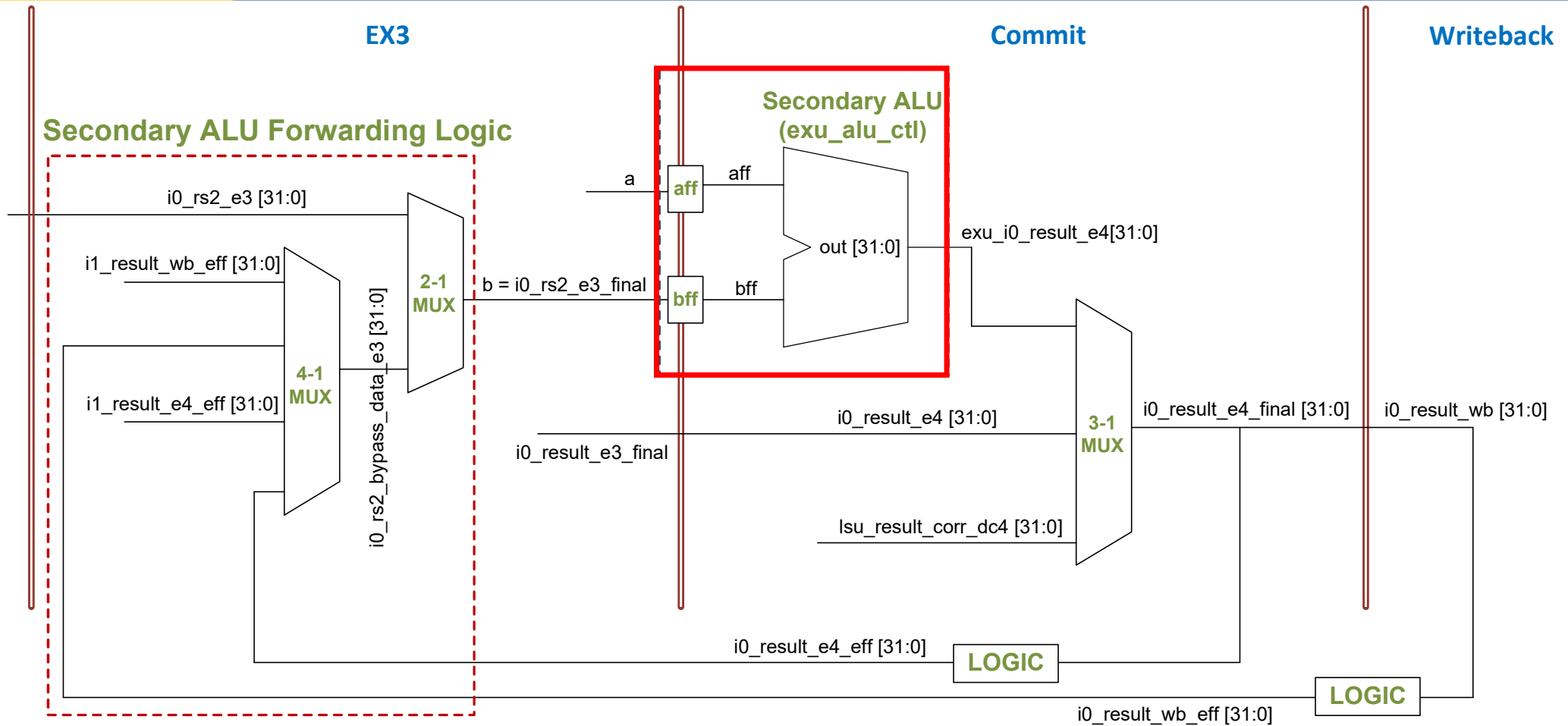


**Diagrama detalhado da
Lógica de Forwarding**

RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding no Andar de Commit

- Instruções que necessitam de vários ciclos para obter o resultado (ou seja, uma operação de vários ciclos, como por exemplo `lw`, `mul`, e `div`) **não pode avançar para o andar de Decode.**
- Mas o SweRV EH1 adiciona uma ALU extra (a **ALU Secundária**) no andar de Commit de cada via. Esta ALU recalcula a operação aritmética-lógica com as entradas adequadas, quando necessário..
- Assim, **não se perdem ciclos** devido à suspensão - mas o custo é de **duas ALUs extra** (uma por via), bem como sinais de controlo e lógica adicional.

RVfpga Lab 15: Resolução de Conflitos de Dados via Forwarding em Commit - Pipeline



RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding em Commit - Exemplo

```
.globl Test_Assembly
```

```
.section .midccm
```

```
A: .space 4
```

```
.text
```

```
Test_Assembly:
```

```
la t0, A                # t0 = addr(A)
```

```
li t1, 0x1              # t1 = 1
```

```
sw t1, (t0)             # A[0] = 1
```

```
li t1, 0x0 li t3, 0x1 li t6, 0xFFFF
```

```
REPEAT:
```

```
beq t6, zero, OUT      # Stay in the loop?
```

```
INSERT_NOPS_9
```

```
lw t1, (t0)
```

```
add t6, t6, -1
```

```
add t3, t3, t1          # t3 = t3 + t1
```

```
INSERT_NOPS_8
```

```
li t1, 0x0
```

```
li t3, 0x1
```

```
add t4, t4, 0x1
```

```
add t5, t5, 0x1
```

```
j REPEAT
```

```
OUT:
```

```
.end
```

RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding no *Commit* – Pipeline

Cycle i

EX3 STAGE

COMMIT STAGE

add t3,t3,t1

lw t1,(t0)

i0_rs2_e3 [31:0]

i0_result_e3_final [31:0]

i0_result_e4 [31:0]

i0_result_e4_final [31:0]

lsu_result_corr_dc4 [31:0]

i0_result_e4_eff [31:0]

LOGIC

Cycle i+1

EX4 STAGE

add t3,t3,t1

aff

1

bff

1

out [31:0]

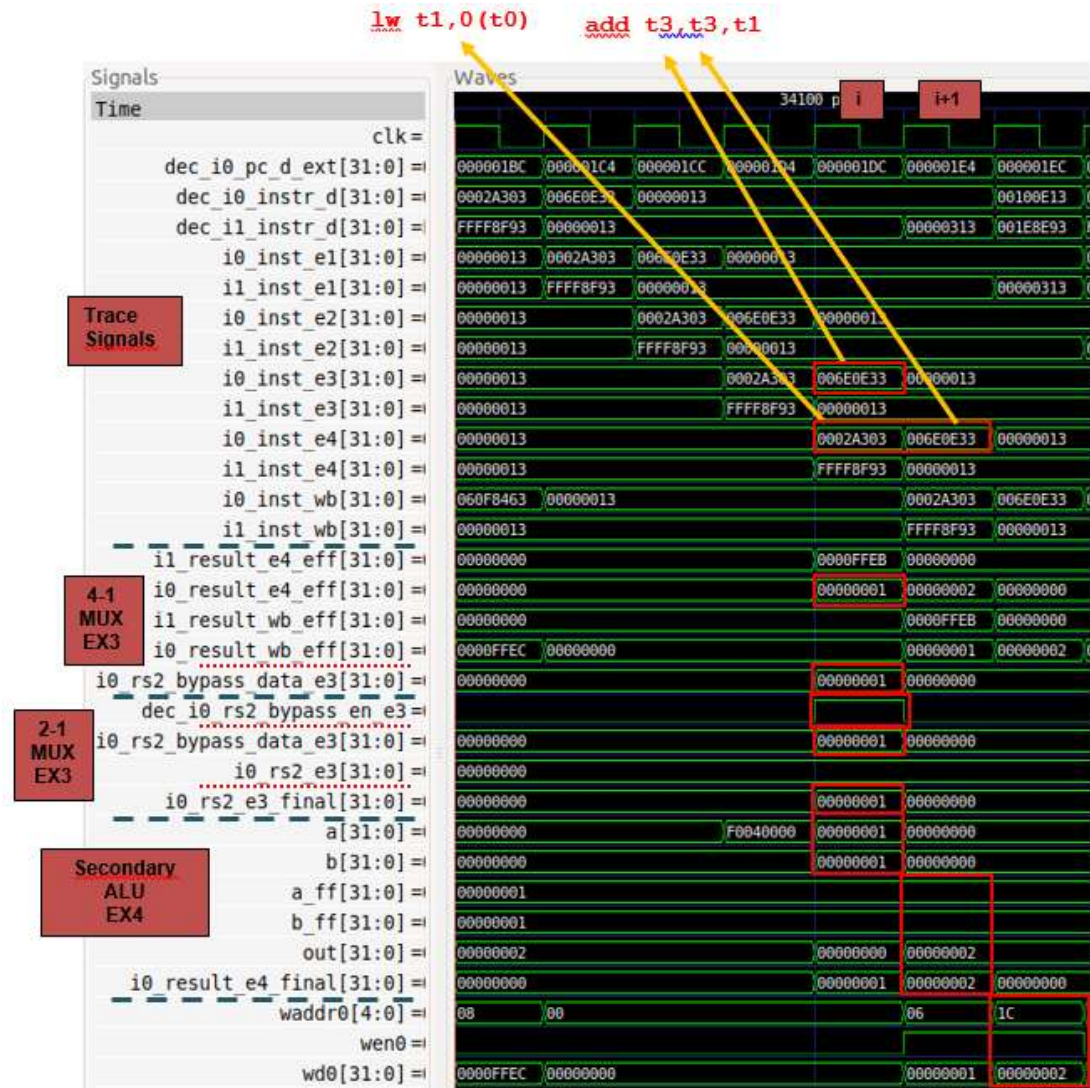
ALU (exu_alu_ctl)

i0_result_e4 [31:0]

i0_result_e4_final [31:0]

2

RVfpga Lab 15: Resolução de Conflitos de Dados por Forwarding no *Commit* - Simulação



RVfpga Lab 15: Resolução de Conflitos de Dados via Forwarding no *Commit* - Simulação

- **Sinais de *Trace***

- **Cycle i :** a instrução `add` está no andar EX3 da Via 0 (`i0_inst_e3 = 0x006E0E33`), e a instrução `lw` está no andar Commit do canal I0 (`i0_inst_e4 = 0x0002A303`).
- **Cycle $i+1$:** a instrução `add` está no andar Commit da Via 0 (`i0_inst_e4 = 0x006E0E33`).

- **Multiplexer 4:1**

- **Cycle i :** o resultado da instrução `lw` (no andar de Commit), é selecionada:

`i0_rs2_bypass_data_e3 = i0_result_e4_eff = 0x00000001`

- **Multiplexer 2:1**

- **Cycle i :** o valor encaminhado é selecionado devido à dependência entre `lw` e `add`:

`i0_rs2_e3_final = i0_rs2_bypass_data_e3 = 0x00000001`

- **Etapas Commit ALU**

- **Cycle $i+1$:** a operação `add` é recalculada usando os valores corretos:

`out = a_ff + b_ff = 0x00000001 + 0x00000001 = 0x00000002`

- **Multiplexer 3:1**

- **Cycle $i+1$:** A saída da ALU secundária é selecionada (`exu_i0_result_e4`). (Quando não existem dependências, `i0_result_e4` é selecionado.)

RVfpga Lab 15: Tarefas – Exemplos

- **TAREFA:** Remova todas as instruções `nop` do exemplo da Figura 2. Desenhe uma figura semelhante à Figura 3 para duas iterações consecutivas do ciclo, depois analise e confirme que a figura está correta comparando-a com uma simulação do Verilator e, finalmente, calcule o IPC utilizando os contadores de desempenho enquanto executa o programa na placa.
- **TAREFA:** No exemplo da Figura 2, remova todas as instruções `nop` e mova a instrução `add t6, t6, -1` após a instrução `add t3, t3, t4` e, em seguida, reexamine o programa tanto na simulação quanto na placa. Neste programa reordenado, as duas instruções de adição dependentes (`add t4, t4, t5` e `add t3, t3, t4`) chegam à fase de decodificação no mesmo ciclo, o que tem um impacto no desempenho. Explique o impacto destas alterações, utilizando tanto a simulação como a execução na placa.
- **TAREFA:** Compare as equações para o multiplexer 10:1 na lógica de encaminhamento com as equações explicadas para o processador em cadeia do DDCARV.
- **TAREFA:** Remova as instruções `nop` no exemplo da Figura 11 e obtenha o IPC usando os contadores HW.
- **TAREFA:** Desativar a ALU secundária como explicado no Laboratório 11 e analisar o exemplo da Figura 11 com uma simulação no Verilator e com uma execução na placa.

RVfpga Lab 15: Exercícios - Exemplos

- **Exercício 1.** Modifique o programa utilizado na Secção 3, acrescentando uma instrução aritmética-lógica extra que depende do resultado da instrução `add`. Analise a simulação do Verilator e explique como são tratados os conflitos de dados para a nova instrução A-L. Em seguida, remova todas as instruções `nop` e analise os resultados fornecidos pelos contadores HW.
- **Exercício 2.** Analise a mesma situação que a descrita na Secção 3 para uma instrução `mul` seguida de uma instrução `add` que usa o resultado da multiplicação. No programa da Figura 11 pode simplesmente substituir o `lw` por um `mul` que escreve no registo `t1`.
- **Exercício 5.** No programa da Secção 2.C do Laboratório 14, substitua a instrução `add x1, x1, 1` por `add x28, x1, 1`. Isto introduz um conflito WAW entre a instrução `add` modificada e a leitura não bloqueante no início do ciclo (`lw x28, (x29)`). Analise em simulação como este conflito é tratado no SweRV EH1, para o qual pode olhar para o valor do sinal `wen2` no Register File. Tente compreender como este sinal é calculado na Unidade de Controlo (módulo `dec`).
- **Exercício 7.** No programa da Secção 2.C do Lab 14, substitua a instrução `add x1, x1, 1` por `add x1, x28, 1`, e a instrução `add x7, x7, 1` por `add x28, x7, 1`. Isto faz com que ocorra um conflito RAW e um conflito WAW. Analise em simulação como estes dois conflitos são tratados.
- **Exercício 8 - Forwarding de escrita para leitura:** Esta é uma situação muito interessante que não analisámos neste laboratório e que irá analisar neste exercício.

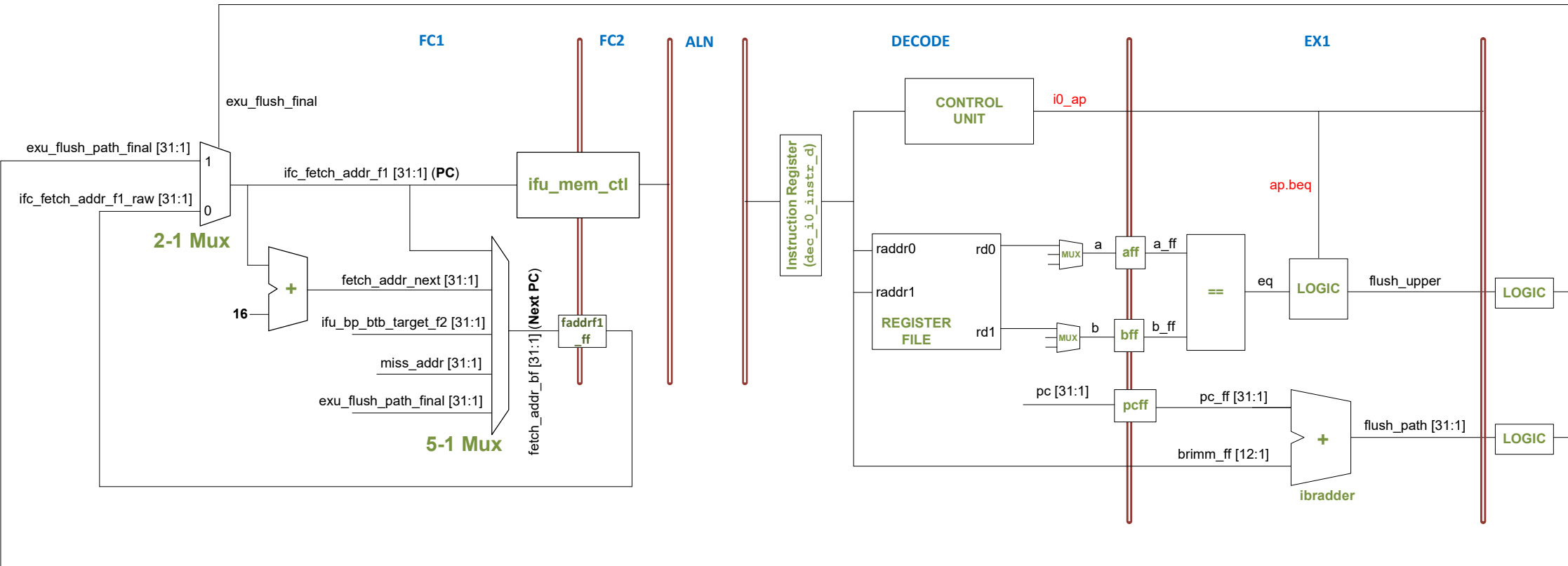
Lab 16:

Conflitos de Controle e Instruções de Salto

RVfpga Lab 16: Conflitos de Controlo e Saltos

- As instruções de salto calculam o endereço da instrução seguinte após o *fetch*.
- Os Conflitos de Controlo podem:
 - **Parar** o pipeline até que o próximo endereço de instrução seja calculado, ou
 - **Prever** se um salto será tomado, e carregar as instruções do caminho previsto.
- O SweRV EH1 tem dois preditores de salto ou *branch predictors* (BPs) possíveis:
 - **Preditor de Salto Simples (ou Naïve)**: prevê sempre o salto não tomado. Tem fraco desempenho mas sem custos de circuito.
 - **Preditor de Salto Gshare**: oferece maior desempenho ao custo de circuito adicional.
- Este Lab analisa a execução da instrução `beq` usando os preditores (BPs) naïve e Gshare.

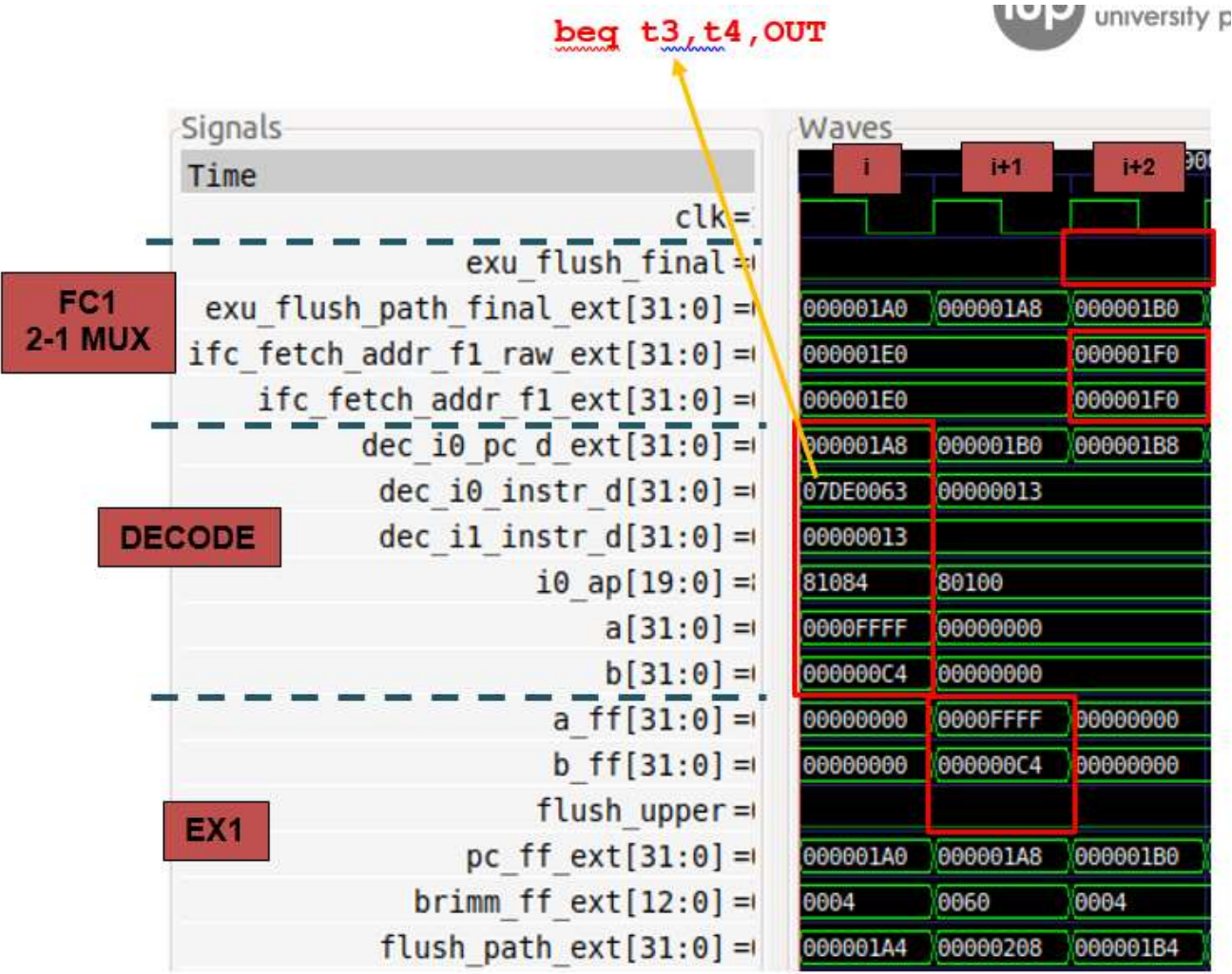
RVfpga Lab 16: Execução de uma Instrução beq e cálculo do PC



RVfpga Lab 16: Execução de uma Instrução beq e cálculo do PC– Exemplo

```
Test_Assembly:
  li t2, 0x008                # Disable Branch Predictor
  csrrs t1, 0x7F9, t2
  li t3, 0xFFFF
  li t4, 0x1
  li t5, 0x0
  li t6, 0x0
LOOP:
  add t5, t5, 1
  INSERT_NOPS_7
  beq t3, t4, OUT
  INSERT_NOPS_7
  add t4, t4, 1
  INSERT_NOPS_7
  beq t3, t3, LOOP
  INSERT_NOPS_7
OUT:
  INSERT_NOPS_8
.end
```

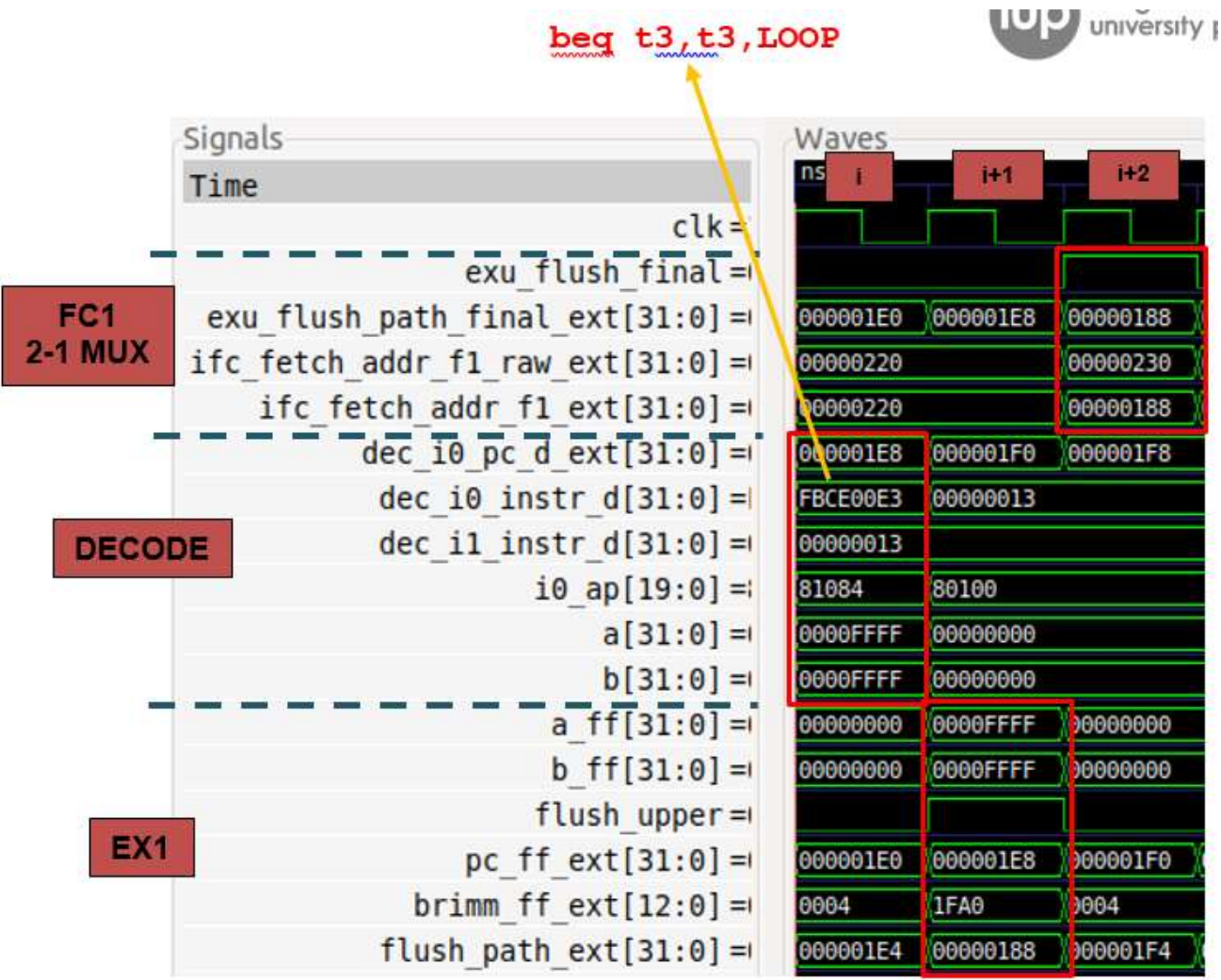
RVfpga Lab 16: Execução da **Primeira** Instrução beq - Simulação



RVfpga Lab 16: Execução da **Primeira** Instrução beq – Análise

- **Ciclo i** - Etapa de Decode para a instrução beq: O primeiro beq (0x07DE0063) é decodificado na Via 0. Os sinais de controlo são gerados, o O Register File é lido, e a instrução do salto é encaminhada para o Pipe I0. Os sinais a e b (0xFFFF e 0xC4, respetivamente, neste exemplo) contêm os argumentos para o comparador utilizado na etapa seguinte
- **Ciclo $i+1$** - Etapa EX1 para a instrução beq: A instrução beq é executada. Os sinais a_ff e b_ff são comparados. Os dois números (0xFFFF e 0xC4) são diferentes, pelo que o salto não é realizado. Neste exemplo, o indicador Gshare é desativado, pelo que todos os saltos são previstos como *não realizados* (i0_ap.predict_nt = 1). Assim, o salto foi corretamente previsto, e o pipeline **não** é descarregado (flush_upper = 0).
- **Ciclo $i+2$** – Etapa FC1: Dado que o salto foi previsto e determinado como não tomado, o carregamento das instruções continua sequencialmente. Note-se que exu_flush_final = 0 e ifc_fetch_addr_f1_ext[31:0] = ifc_fetch_addr_f1_raw_ext[31:0] = 0x000001F0. Este endereço aponta para a próxima sequência de instruções de 128 bits.

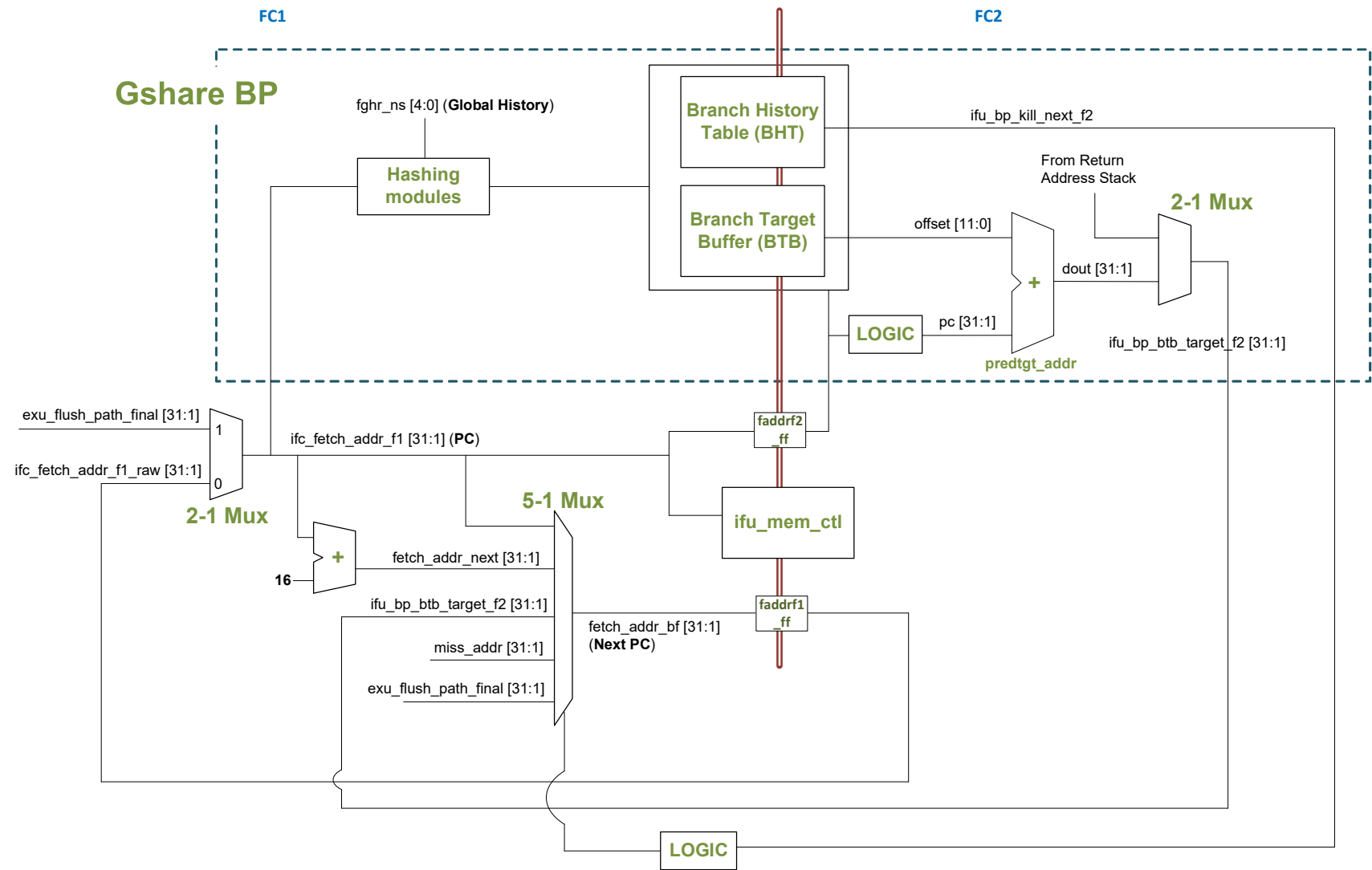
RVfpga Lab 16: Execução da Segunda Instrução beq – Simulação



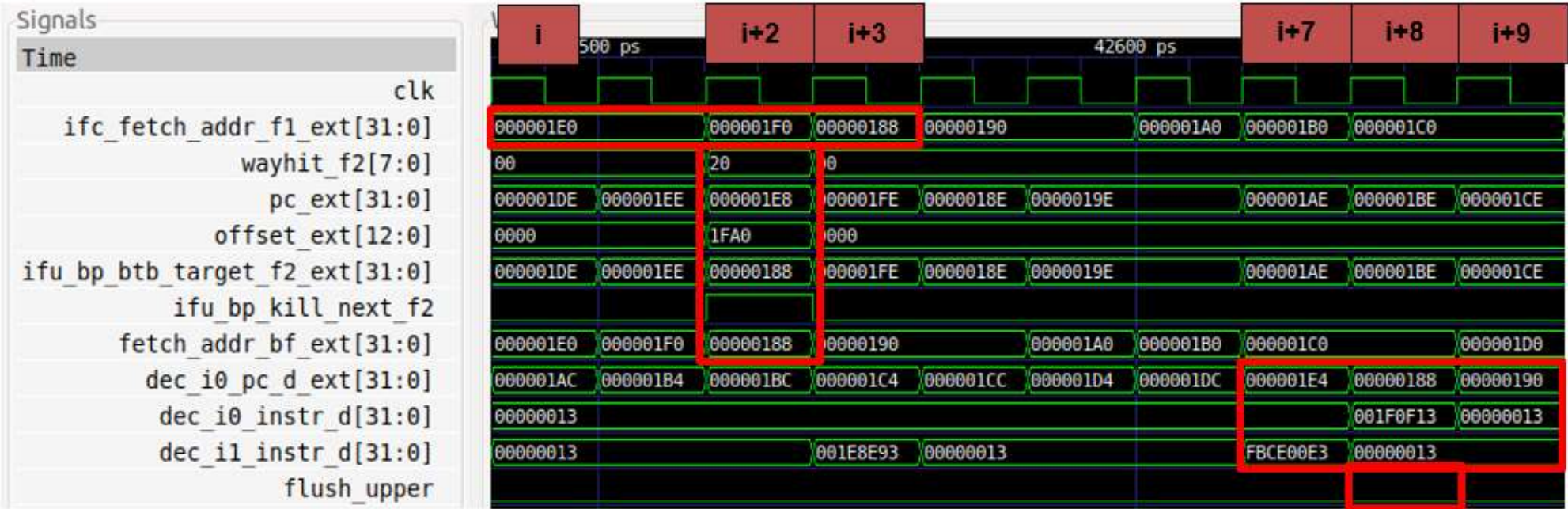
RVfpga Lab 16: Execução da **Segunda** Instrução beq – Análise

- **Ciclo i** – Etapa de Decode para a instrução beq: O segundo beq (0xFBCE00E3) é decodificado na Via 0. São gerados sinais de controlo do pipeline, o Register File é lido, e a instrução do salto é encaminhada para o Pipe I0. Os sinais a e b (0xFFFF para ambos, neste exemplo) contêm as entradas para o comparador utilizado na etapa seguinte.
- **Ciclo $i+1$** - Etapa EX1 para a instrução beq: A instrução beq é executada. Os sinais a_ff e b_ff são comparados. Os dois números são iguais, pelo que o salto é tomado. No entanto, o Naïve BP prevê todos os ramos como *não tomados* (i0_ap.predict_nt = 1). Assim, o salto foi mal previsto, e as instruções carregadas devem ser descartadas (flush_upper = 1)..
- **Ciclo $i+2$ - FC1 stage:** A execução deve continuar no endereço de destino do salto. exu_flush_final = 1 e ifc_fetch_addr_fl_ext = exu_flush_path_final_ext = 0x00000188. Este endereço corresponde ao endereço de destino do salto, que é o endereço da primeira instrução do ciclo.

RVfpga Lab 16: O preditor de saltos Gshare



RVfpga Lab 16: O preditor de saltos Gshare para o Segundo beq



RVfpga Lab 16: O preditor de saltos Gshare para o **Segundo** beq

- **Ciclo *i*:** O endereço do pacote que contém o segundo ramo é fornecido à Cache de Instrução : `ifc_fetch_addr_f1_ext = 0x000001E0`. O Branch Target Buffer (BTB) é lido utilizando este endereço.
- **Ciclo *i+2*:** Ocorre um *hit* no BTB: `wayhit_f2 = 0x20`. O endereço do salto (`pc_ext = 0x000001E8`) é adicionado ao deslocamento fornecido pelo BTB (`offset_ext = 0x1FA0`, que é um valor negativo), o que resulta no endereço alvo previsto (`ifu_bp_btb_target_f2_ext = 0x00000188`). Dado que o salto é previsto pelo BHT (`ifu_bp_kill_next_f2 = 1`), é utilizado como o Próximo Carregamento do PC (`fetch_addr_bf_ext = 0x00000188`).
- **Ciclo *i+3*:** O Endereço de Fetch é o endereço alvo previsto do salto, que foi calculado no ciclo anterior: `ifc_fetch_addr_f1_ext = 0x00000188`.
- **Ciclo *i+7*:** O ramo é descodificado na Via 1 (`dec_i1_instr_d = 0xFBCE00E3`).
- **Ciclo *i+8*:** O salto executa. A previsão estava correta, pelo que não é necessário ativar a descarga (`flush_upper = 0`).
- **Ciclo *i+9*:** A execução continua normalmente através do endereço de destino do salto, dado que a previsão estava correta.

RVfpga Lab 16: Tarefas e exercícios - Exemplos

- **TAREFA:** No Lab 15, analisámos como os conflitos de dados RAW são resolvidos no andar de *Commit* através das ALUs secundárias. À semelhança das instruções A-L que estudámos nesse lab, uma instrução de salto condicional pode ter um conflito de dados RAW com uma operação multi-ciclo anterior que tem de ser resolvido no momento do Commit. Se for determinado que o salto foi mal previsto, o pipeline deve ser descarregado e redirecionado a partir do andar de Commit. Analise esta situação utilizando uma versão ligeiramente modificada do programa da Figura 2.
- **TAREFA :** No exemplo da Figura 2, remova todas as instruções `nop` e analise a simulação. Em seguida, calcule o IPC com os contadores de desempenho, executando o programa na placa. Active o preditor de ramos utilizado no SweRV EH1 (comentando as duas instruções iniciais da Figura 2) e analise a simulação e a execução na placa. Compare as duas experiências e explique os resultados.
- **TAREFA:** Explicar como o Global History Register é atualizado no módulo `ifu_bp_ctl`.
- **Exercício 1)** Implemente um Preditor de Salto Bimodal e compare o seu desempenho com o BP Gshare.
- **Exercício 2)** Este exercício é baseado no exercício 4.25 do livro “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([HePa]).

Lab 17:

Execução Super-escalar

RVfpga Lab 17: Introdução

- O processador SweRV EH1 da Western Digital é um núcleo super escalar de 2 vias de 32 bits com 9 andares de pipeline.
- Um processador super-escalar contém **múltiplas cópias do circuito** para executar múltiplas instruções simultaneamente.
- A **latência** da execução de uma única instrução é a mesma de um processador escalar, mas o processador pode executar e efectivar mais instruções por ciclo, **melhorando assim o seu rendimento**.
- O SweRV EH1 não inclui suporte para escalonamento dinâmico de instruções com execução fora de ordem, excepto para as leituras não bloqueantes. No entanto, é possível reordenar estaticamente o código de modo a explorar melhor os recursos, incluindo as duas vias do pipeline

RVfpga Lab 17: Introdução

- SweRV EH1 é um processador **super-escalar de 2 vias**.
 - Carrega, executa e efectiva até **duas instruções por ciclo**.
 - O Register File multi-portas lê até quatro operandos de entrada e escreve dois valores em cada ciclo (mais um valor proveniente de uma leitura não bloqueante, tal como analisado no Lab 15).
 - Cada via contém canais independentes: dois canais para Inteiros, um canal para Multiplicação, um canal para Leitura-Escrita, e um Divisor (*non-pipelined*).
- Idealmente, num processador super-escalar de 2 vias, o débito (IPC) duplica em comparação com um processador de uma única via. Infelizmente, os programas reais normalmente apresentam melhorias de desempenho de 1,3x-1,5x ao passar de processadores de 1 para 2 vias; no entanto, adicionar a segunda via requer muito mais hardware.
- Neste laboratório, analisamos dois programas simples, comparando o comportamento ao utilizar configurações de uma e duas vias de SweRV EH1.

RVfpga Lab 17: Quatro Instruções A-L Independentes – Exemplo – Single-Issue

```
.globl Test_Assembly
```

```
.text
```

```
Test_Assembly:
```

```
li t2, 0x400          # Disable Dual-Issue Execution  
csrrs t1, 0x7F9, t2
```

```
li t0, 0x0
```

```
li t1, 0x1
```

```
li t2, 0x1
```

```
li t3, 0x3
```

```
li t4, 0x4
```

```
li t5, 0x5
```

```
li t6, 0x6
```

```
lui t2, 0xF4
```

```
add t2, t2, 0x240
```

```
REPEAT:
```

```
add t0, t0, 1
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

```
add t3, t3, t1
```

```
sub t4, t4, t1
```

```
or t5, t5, t1
```

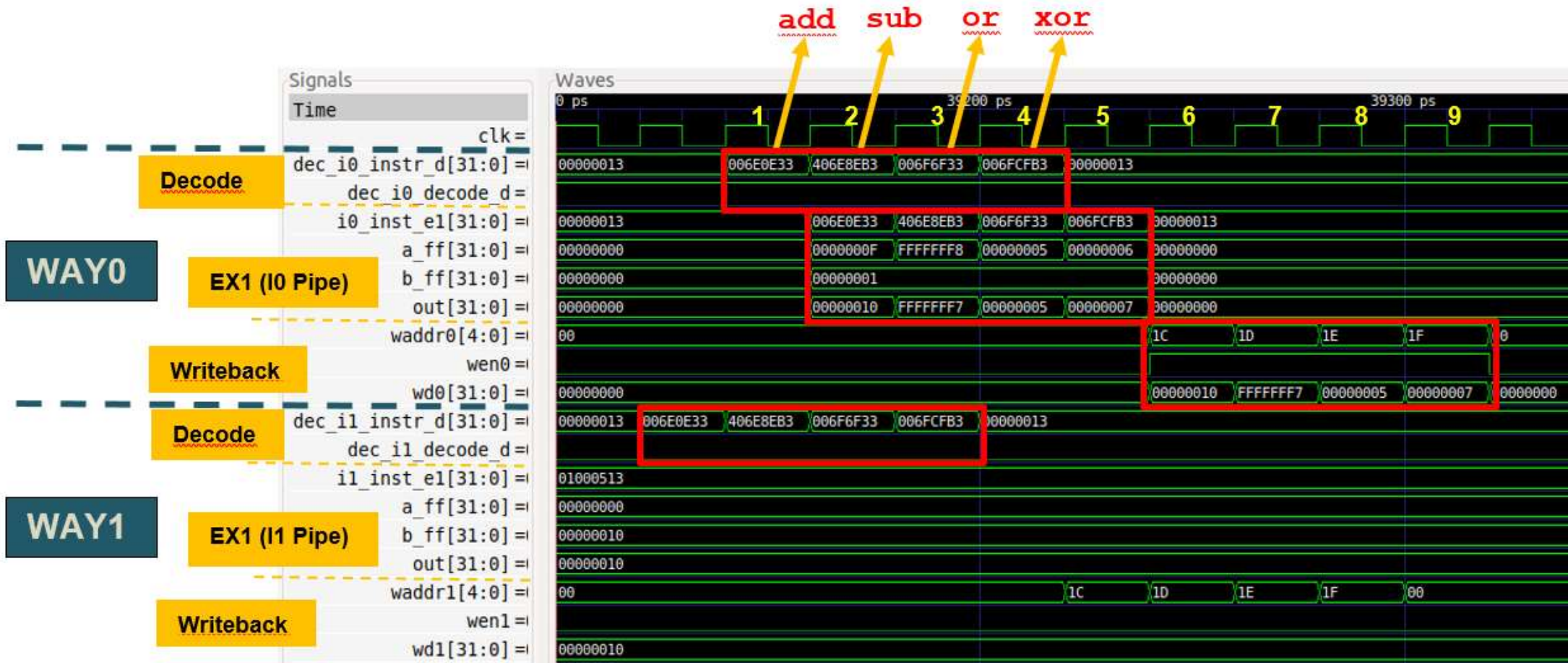
```
xor t6, t6, t1
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_3
```

```
bne t0, t2, REPEAT # Repeat the loop
```

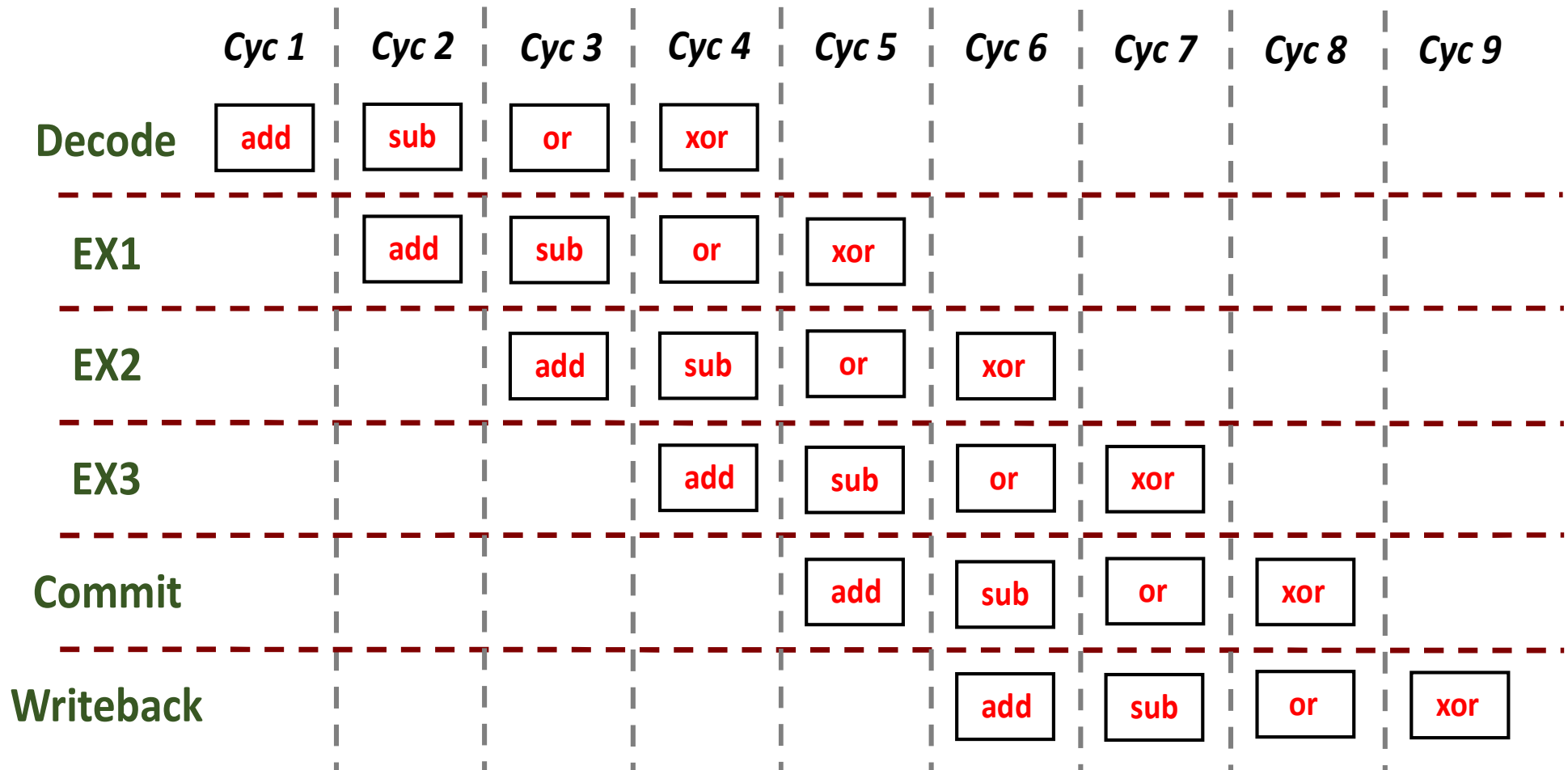
RVfpga Lab 17: Quatro Instruções A-L Independentes - Simulação - **Single-Issue**



RVfpga Lab 17: Quatro Instruções A-L Independentes – Simulação – **Single-Issue**

- As instruções são recebidas de ambas as vias no momento da decodificação, mas só são enviadas para execução na Via 0, porque a Via 1 está desativada.
 - **Via 0:**
 - O sinal `dec_i0_decode_d` é sempre 1 no nosso exemplo; especificamente, é 1 para as quatro instruções AL em análise.
 - A instrução no andar de Decode **é propagada** para o canal I0 (`i0_inst_e1[31:0]`)
 - **Via 1:**
 - O sinal `dec_i1_decode_d` é sempre 0 no nosso exemplo; especificamente, é 0 para as quatro instruções AL em análise.
 - A instrução no andar de Decode **NÃO é propagada** (`i1_inst_e1[31:0]`) para o andar de Execute.
- Consequentemente, apenas a ALU do canal I0 é utilizada (ver sinais `aff`, `bff` e `out` em ambas as vias) e só é utilizada a porta 0 do Register File (ver sinais `waddr`, `wen` e `wd` em ambas as vias).

RVfpga Lab 17: Quatro Instruções A-L Independentes - Diagrama - **Single-Issue**



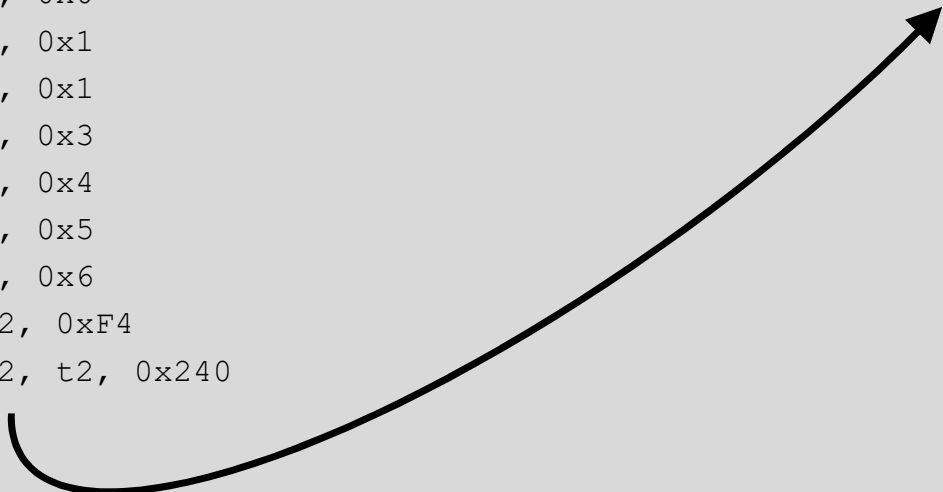
RVfpga Lab 17: Quatro Instruções A-L Independentes – Exemplo – Dual-Issue

```
.globl Test_Assembly

.text
Test_Assembly:

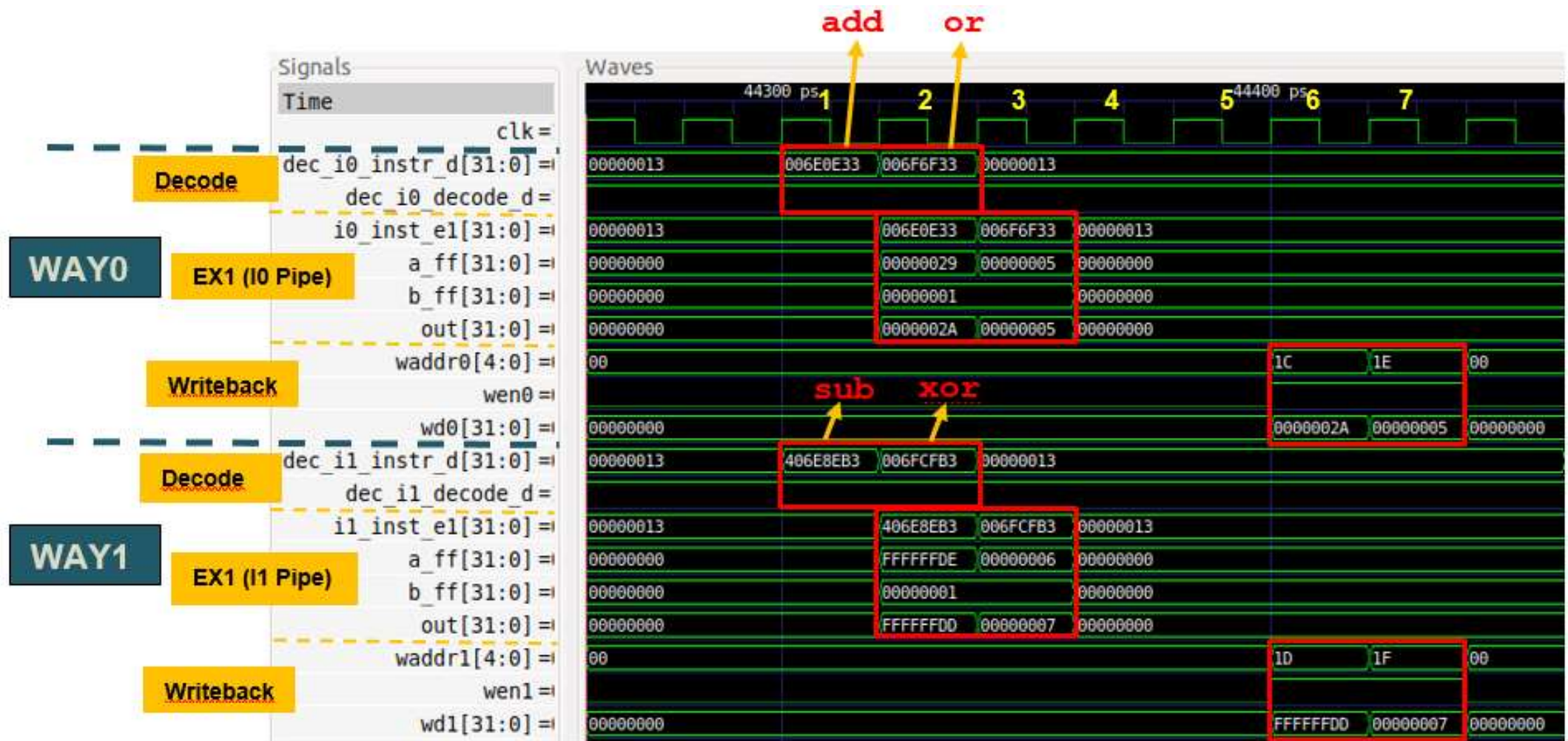
# li t2, 0x400          # Disable Dual-Issue Execution
# csrrs t1, 0x7F9, t2

li t0, 0x0
li t1, 0x1
li t2, 0x1
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
lui t2, 0xF4
add t2, t2, 0x240
```



```
REPEAT:
    add t0, t0, 1
    INSERT_NOPS_10
    INSERT_NOPS_4
    add t3, t3, t1
    sub t4, t4, t1
    or  t5, t5, t1
    xor t6, t6, t1
    INSERT_NOPS_10
    INSERT_NOPS_3
    bne t0, t2, REPEAT # Repeat the loop
```

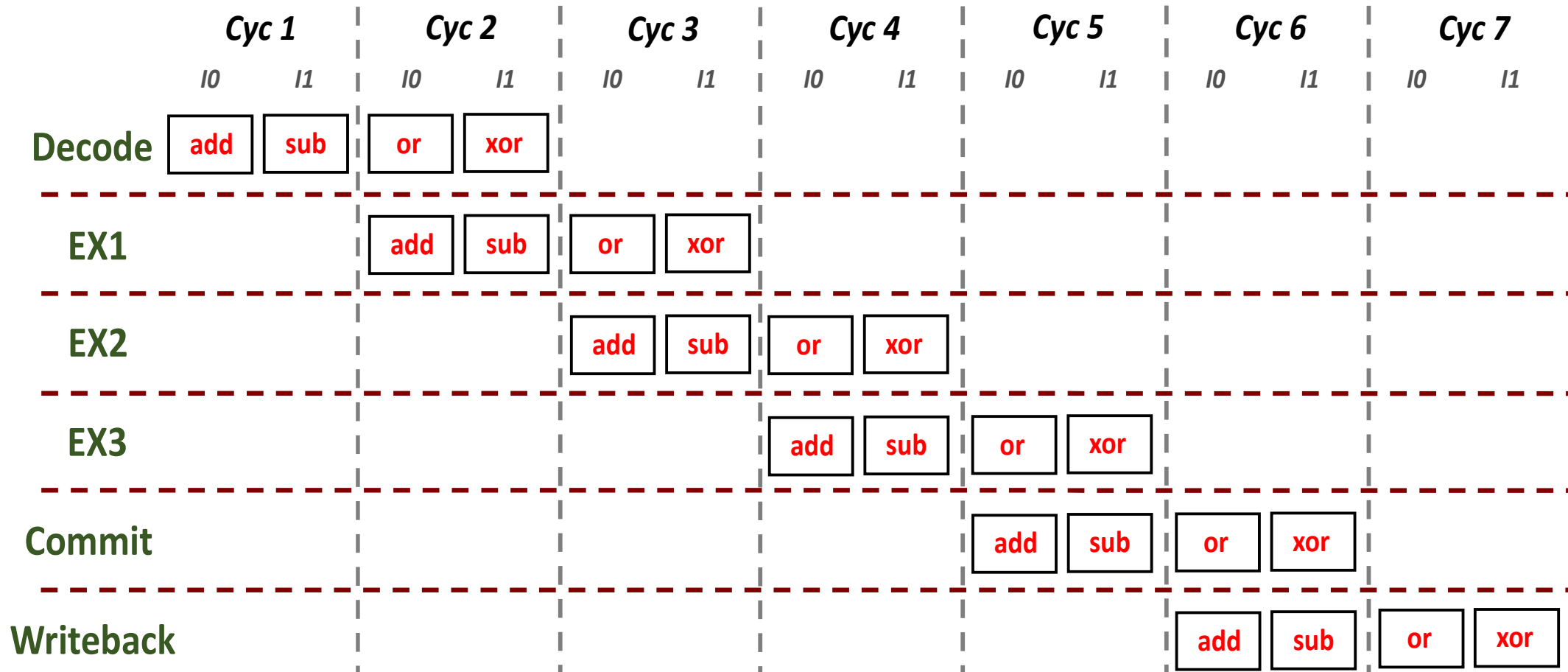
RVfpga Lab 17: Quatro Instruções A-L Independentes - Simulação- Dual-Issue



RVfpga Lab 17: Quatro Instruções A-L Independentes – Análise – Dual-Issue

- Em cada ciclo, duas instruções são decodificadas, uma de cada via, e duas instruções são enviadas para os andares Execute, uma através do canal I0 e a outra através de I1.
 - **Via 0:**
 - O sinal `dec_i0_decode_d` é sempre 1 – sendo verdade para duas das quatro instruções A-L do nosso exemplo (as outras duas instruções A-L são decodificadas na Via 1).
 - A instrução no andar de Decode é propagada para o canal I0 (`i0_inst_e1[31:0]`).
 - **Via 1:**
 - O sinal `dec_i1_decode_d` é sempre 1 – sendo verdade para duas das quatro instruções A-L do nosso exemplo (as outras duas instruções A-L são decodificadas no Caminho 0).
 - A instrução no andar de Decode é propagada para o canal I1 (`i1_inst_e1[31:0]`).
- Assim, são utilizadas as ALUs em ambos os canais (I0 e I1) (ver sinais `aff`, `bff`, e `out` em ambas as vias), e são utilizadas as duas portas de escrita do registo de ficheiros (ver sinais `waddr`, `wen` e `wd` em ambas as vias).

RVfpga Lab 17: Quatro Instruções A-L Independentes - Diagrama - Dual-Issue



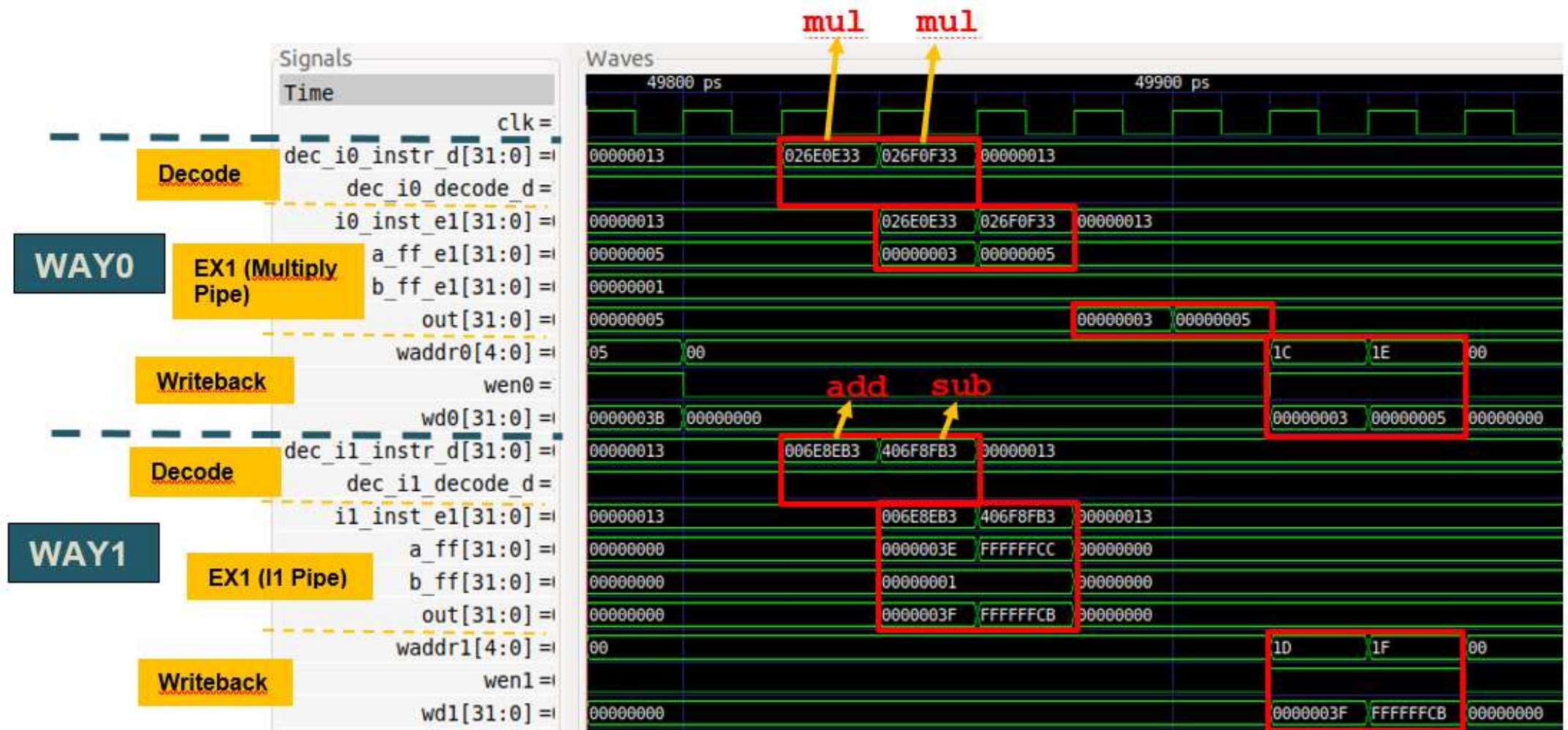
RVfpga Lab 17: Duas Instruções mul Entrelaçadas com Duas Instruções A-L – Exemplo – Dual-Issue

```
.globl Test_Assembly
.text
Test_Assembly:
# li t2, 0x400      # Disable Dual-Issue Execution
# csrrs t1, 0x7F9, t2
```

```
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
li t0, 0x0
lui t1, 0xF4
add t1, t1, 0x240
```

```
REPEAT:
    add t0, t0, 1
    INSERT_NOPS_10
    INSERT_NOPS_4
    mul t3, t3, t1
    add t4, t4, t1
    mul t5, t5, t1
    sub t6, t6, t1
    INSERT_NOPS_10
    INSERT_NOPS_3
    bne t0, t1, REPEAT # Repeat the loop
.end
```

RVfpga Lab 17: Duas Instruções mul Entrelaçadas com Duas Instruções A-L– Simulação – Dual-Issue



RVfpga Lab 17: Duas Instruções mul Entrelaçadas com Duas Instruções A-L - Análise – Dual-Issue

- As instruções são recebidas de ambas as vias no momento da descodificação e são enviadas para os andares de execução de ambas as vias.
 - **Via 0:**
 - O sinal `dec_i0_decode_d` é sempre 1 – para duas das quatro instruções analisadas no nosso exemplo (as outras duas instruções são descodificadas na Via 1).
 - A instrução no andar de Decode é enviada para o canal da Multiplicação (`i0_inst_e1[31:0]`)
 - **Via 1:**
 - O sinal `dec_i1_decode_d` é sempre 1 – para duas das quatro instruções analisadas no nosso exemplo (as outras duas instruções são descodificadas na Via 1).
 - A instrução em DECODE (`dec_i1_instr_d[31:0]`) é propagada para o canal I1 (`i1_inst_e1[31:0]`)
- Assim, são utilizadas as ALU na Via I1 e o Multiplicador (ver sinais `a_ff_e1`, `b_ff_e1`, `out`, `a_ff`, `b_ff`, e `out`), e são utilizadas as duas portas de escrita de ficheiros de registo (ver sinais `waddr`, `wen`, e `wd` em ambos os sentidos).

RVfpga Lab 17: Tarefas e exercícios - Exemplos

- **TAREFA:** Remova todas as instruções nop dentro do corpo do ciclo da Figura 2. Repita a simulação da Figura 3. Qual é o IPC esperado para este programa? Execute o programa na placa e verifique se o IPC obtido é o esperado.
- **Exercício 2)** Analise as diferenças entre o processador SweRV EH1 (dual-issue) e o exemplo de processador superescalar proposto na Secção 7.7.4 do livro S. Harris and D. Harris, “Digital Design and Computer Architecture: RISC-V Edition” [DDCARV] (mostrado na Figura 1 para maior conveniência).
- **Exercício 3)** Analise o programa da Figura 7.70 na Secção 7.7.4 do DDCARV, que é fornecido num projeto PlatformIO. Execute o programa no SweRV EH1, tanto em simulação como na placa (para esta última, remova as instruções nop). Explique os resultados. Se necessário, reordene o programa tentando obter o IPC óptimo. Em seguida, desative a execução em dual-issue como explicado neste laboratório - e em SweRVref.docx (Secção 2). Compare a simulação e os resultados obtidos na placa quando o *dual-issue* está ativado.
- **Exercícios 5, 6 e 7)** Estes exercícios são baseados nos exercícios dos livros:
 - “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy.
 - “Digital Design and Computer Architecture: RISC-V Edition”, by S. Harris and D. Harris.

Lab 18:

Adicionando Novas Características: Instruções e Contadores

RVfpga Lab 18: Adicionar Instruções e Funcionalidades

- Neste lab, aplicará os conhecimentos adquiridos em laboratórios anteriores para modificar o processador SweRV EH1 para adicionar as seguintes novas características:
 - **Adicionar instruções A-L:** Adicionar instruções aritmético-lógicas da nova extensão de manipulação de bits disponível na arquitetura RISC-V.
 - **Adicionar instruções de vírgula-flutuante:** Adicionar três instruções de vírgula-flutuante: adicionar, multiplicar e dividir. Depois utilizá-las para calcular o algoritmo de bissecção.
 - **Adicionar Contador HW :** Adicionar um novo contador em hardware que conta o número de instruções do tipo I executadas.
- Em alguns destes exercícios guiamo-lo através do processo de modificação do núcleo, e em outros descobrirá o que precisa de ser feito. Pode encontrar todos os detalhes no documento do laboratório.

Lab 19:

Cache de Instruções

RVfpga Lab 19: Introdução

- Este laboratório descreve e explora o sistema de memória do Sistema RVfpga. O Sistema de Memória da Rvfpga tem os seguintes elementos:
 - Memória principal externa DDR
 - Cache de instruções (I\$)
 - Duas memórias *Scratchpad* (também chamadas memórias estreitamente acopladas), uma para dados (DCCM) e uma para instruções (ICCM). A ICCM está desativada no sistema base.
- Neste laboratório, primeiro descrevemos como os dados são lidos e gravados na memória externa DDR, e depois aprofundamos o funcionamento e a gestão da I\$ disponível no sistema RVfpga.

RVfpga Lab 19: Leitura e escrita de dados na memória - Exemplo

```
.data
D: .space 40000

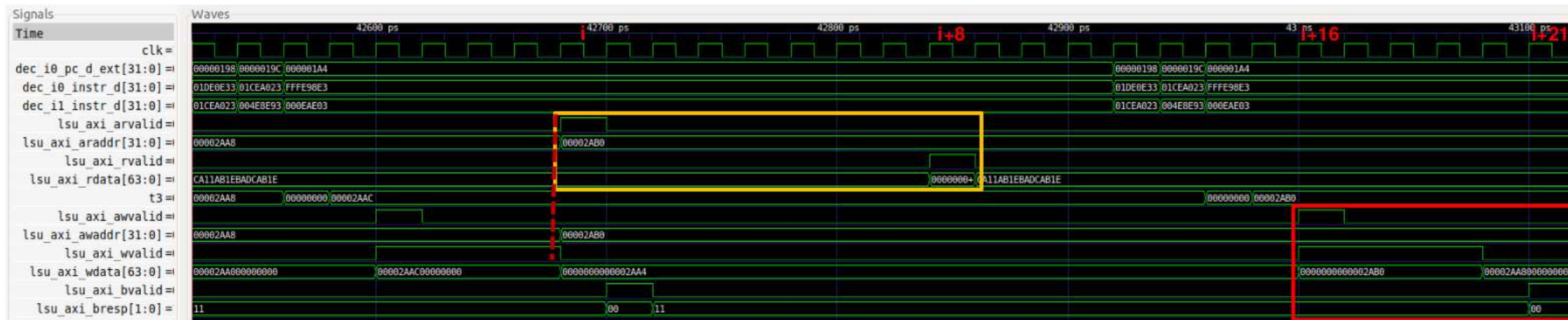
.text
Test_Assembly:
li t2, 0x000
csrrs t1, 0x7F9, t2
la t4, D
li t5, 50
li t0, 40000
la t6, D
add t6, t6, t0
```

```
REPEAT:
lw t3, (t4)
add t3, t3, t5
sw t3, (t4)
add t4, t4, 4
bne t4, t6, REPEAT
```



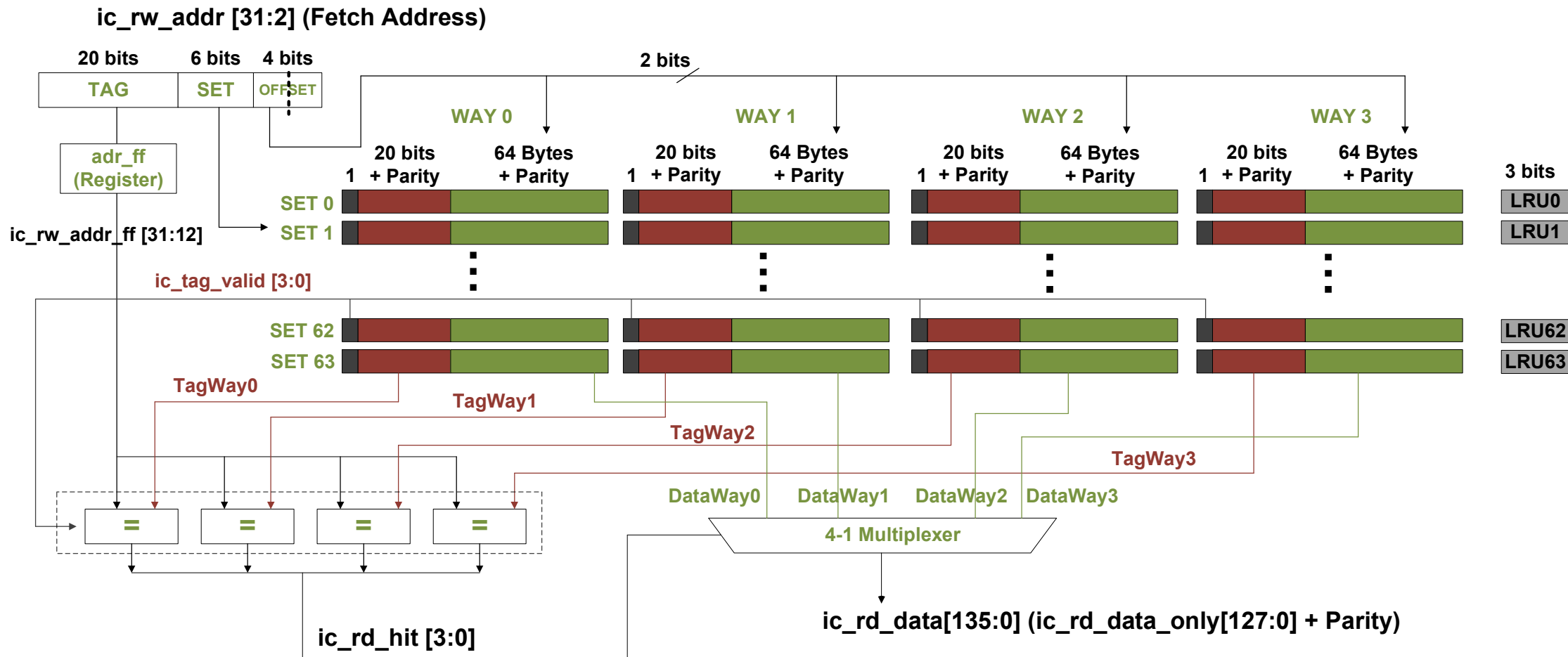
RVfpga Lab 19: Leitura e escrita de dados na memória - Simulação

O exemplo ilustra um programa que inclui uma instrução *load* seguida de uma instrução *store*



- Ciclo $i - i+8$: O processador lê dados da memória externa DDR (quadrado amarelo) em t_3 , através do barramento.
- Ciclo $i+16 - i+21$: O processador escreve o valor de t_3 na memória externa DDR (quadrado vermelho), através do barramento.

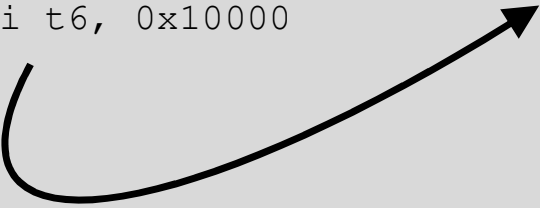
RVfpga Lab 19: Configuração e Funcionamento I\$



RVfpga Lab 19: Gestão de Hits e Misses I\$ – Exemplo

Test_Assembly:


```
INSERT_NOPS_3  
INSERT_NOPS_8  
INSERT_NOPS_8  
li t6, 0x10000
```



REPEAT:

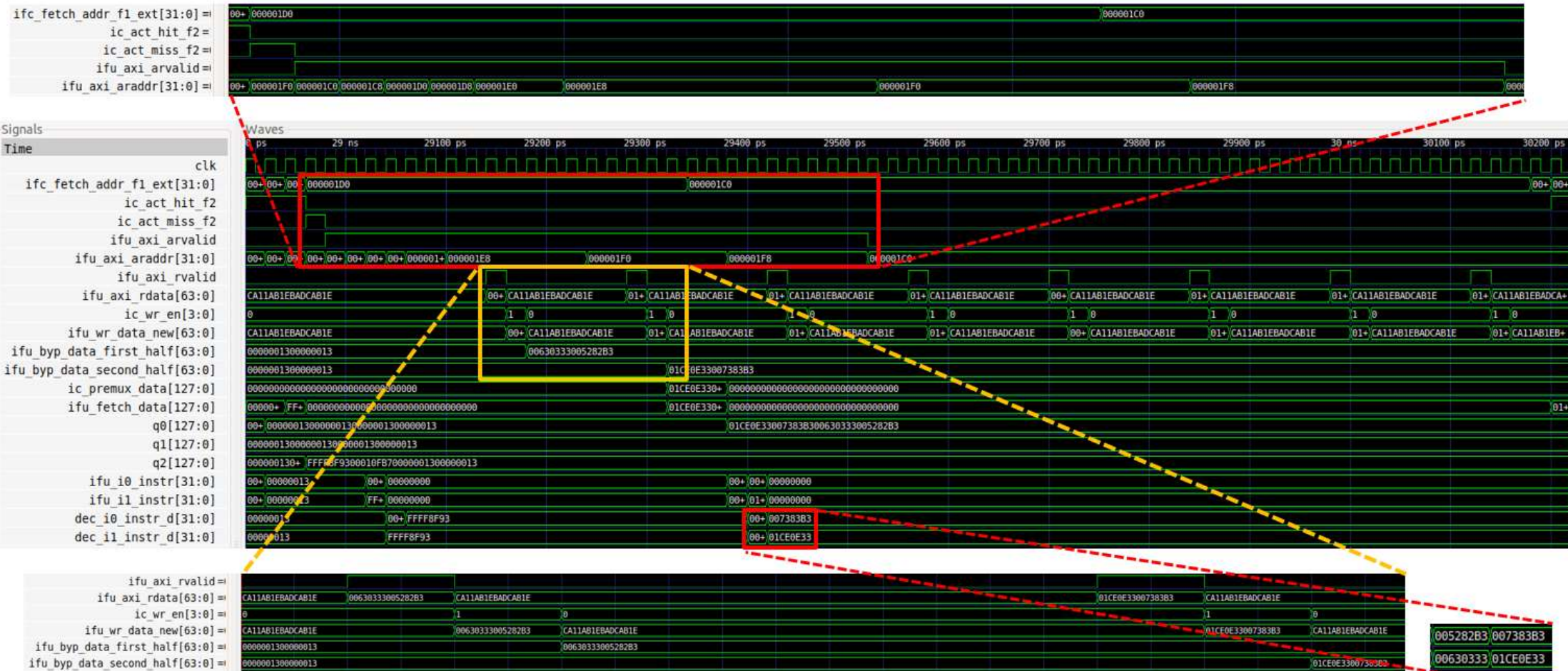
```
add t6, t6, -1
```

```
add t0, t0, t0  
add t1, t1, t1  
add t2, t2, t2  
add t3, t3, t3  
add t4, t4, t4  
add t5, t5, t5  
add t6, t6, t6  
add a7, a7, a7  
add t0, t0, t0  
add t2, t2, t2  
add t1, t1, t1  
add t3, t3, t3  
add t4, t4, t4  
add t6, t6, t6  
add t5, t5, t5  
add a7, a7, a7
```



```
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
bne t6, zero, REPEAT  
  
ret
```

RVfpga Lab 19: Gestão de **Misses** I\$ - Simulação



RVfpga Lab 19: Gestão de **Misses** I\$ - Análise

- A simulação mostra o carregamento das 16 instruções `add` a primeira vez que são executadas. Dado que estas instruções ainda não estão no I\$, um Miss é acionado na I\$ e as instruções devem ser copiadas da Memória Externa da DDR para a I\$.
 - Um Miss de I\$ é assinalado em cerca de 29ns (`ic_act_miss_f2 = 1`), o que desencadeia o pedido do bloco através do barramento AXI (`ifu_axi_arvalid = 1`).
 - Os oito pedaços de 64 bits que compõem o bloco alvo são solicitados sequencialmente através do barramento AXI.
 - O sinal `ifu_axi_arvalid` fica ativo durante 27 ciclos. Este sinal indica que o canal está a sinalizar um endereço de leitura válido e informação de controlo.
 - Durante estes 27 ciclos onde `ifu_axi_arvalid = 1` os endereços iniciais dos oito pedaços de 64 bits são fornecidos sequencialmente através do bus AXI utilizando o sinal `ifu_axi_araddr`, que fornece os 8 endereços que devem ser lidos a partir da memória DDR.

RVfpga Lab 19: Gestão de **Misses** I\$ - Análise

- A figura do meio mostra os oito pedaços de 64 bits que chegam sequencialmente ao processador através do barramento AXI no sinal `ifu_axi_rdata`.
 - O sinal `ifu_axi_rvalid`, indica que o canal está a sinalizar os dados lidos necessários, vai alto durante um ciclo a cada 7 ciclos.
 - Cada um dos oito pedaços de 64 bits (cada um contendo duas instruções) é fornecido no sinal `ifu_axi_rdata`.
- As duas figuras de baixo mostram que cada um dos oito pedaços de 64 bits está escrito no I\$ logo após a sua chegada ao controlador de cache.
- Finalmente, pode-se ver que as quatro instruções são desviadas do controlador I\$ para o pipeline, para que este possa reiniciar a execução o mais cedo possível após o Miss de I\$. Vários ciclos mais tarde, as quatro instruções chegam à fase de decodificação.

RVfpga Lab 19: Gestão de Hits I\$ - Simulação



RVfpga Lab 19: Gestão de Hits I\$ - Análise

- Na simulação anterior pode ver-se um Hit na I\$.
 - **Ciclo i:** O endereço da primeira instrução de adição (`add t0, t0, t0`) é dado no sinal `ifc_fetch_addr_f1_ext`. Este sinal é passado para o I\$ exceto pelos seus dois bits menos significativos, que não são necessários porque as instruções estão alinhadas em 4 bytes (32 bits). Assim, `ic_rw_addr = 0x0000070`. Os vectores Tag e Data usam um subconjunto do endereço Fetch.
 - **Ciclo i+1:** As quatro etiquetas, uma por via de cache, estão nos sinais `TagWay0-TagWay3`. Estes são comparados com o campo TAG do endereço Fetch. Neste caso, todas as etiquetas são as mesmas que o campo TAG, contudo apenas uma via (Via 0) é válida (`ic_tag_valid = 0001`), assim, é sinalizado um Hit na Via 0: `ic_rd_hit = 0001`. Além disso, quatro pacotes de 128 bits estão nos sinais `DataWay0-DataWay3`: `ic_rd_data_only = 0x01ce0e33007383b300630333005282b3`
 - **Ciclo i+2:** A primeira e a segunda instruções `add` são extraídas no andar de Align do buffer q1: `ifu_i0_instr = 0x005282b3` e `ifu_i1_instr = 0x00630333`
 - **Ciclo i+3:** A terceira e a quarta instruções `add` são extraídos no andar de Align e, ao mesmo tempo, a primeira e a segunda instruções `add` são decodificadas: `ifu_i0_instr = 0x007383b3`, `ifu_i1_instr = 0x01ce0e33`, `dec_i0_instr_d = 0x005282b3` e `dec_i1_instr_d = 0x00630333`
 - **Ciclo i+4:** Finalmente, a terceira e quarta instruções `add` são decodificadas: `dec_i0_instr_d = 0x007383b3` e `dec_i1_instr_d = 0x01ce0e33`

RVfpga Lab 19: Política de Substituição I\$

- A maioria das caches associativas tem uma política de substituição dos dados menos recentemente utilizada (LRU). No entanto, o rastreo da forma menos recentemente utilizada torna-se complicado, pelo que as políticas de LRU aproximadas (normalmente chamadas Pseudo LRU) são frequentemente utilizadas e são suficientemente boas na prática.
- SweRV EH1 utiliza uma política aproximada chamada **Binary Tree Pseudo LRU**.
 - Requer N-1 bits por Set (a que chamamos Estado LRU) numa cache associativa N-Vias. Isto traduz-se em 3 bits por Set no I\$ de SweRV EH1.

Substituição de Bloco

Estado LRU	Via a Substituir
x00	Way 0
x10	Way 1
0x1	Way 2
1x1	Way 3

Atualização do Estado LRU

Via Escrita	Próximo Estado LRU
Via 0	-11
Via 1	-01
Via 2	1-0
Via 3	0-0

RVfpga Lab 19: Política de Substituição I\$ – Exemplo

- O exemplo abaixo acede a cinco blocos diferentes de I\$ dentro de um laço infinito. Todos os cinco blocos mapeiam para o mesmo conjunto de I\$: SET = 8.
- O loop infinito contém cinco instruções `j` (salto), onde cada par de instruções `j` é separado por 1023 `nops`. A instrução `j` mais os `nops` ocupam 4 KiB, o que é igual ao tamanho de cada Via no I\$.

```
Set8_Block1:  j Set8_Block2      # This j instruction is at address 0x00000200
               INSERT_NOPS_1023
Set8_Block2:  j Set8_Block3      # This j instruction is at address 0x00001200
               INSERT_NOPS_1023
Set8_Block3:  j Set8_Block4      # This j instruction is at address 0x00002200
               INSERT_NOPS_1023
Set8_Block4:  j Set8_Block5      # This j instruction is at address 0x00003200
               INSERT_NOPS_1023
Set8_Block5:  j Set8_Block1      # This j instruction is at address 0x00004200
```

SET 8 after execution of the first j instruction at 0x200

Valid	Tag	Data	
1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
0			WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 011

SET 8 after execution of the second j instruction at 0x1200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 001

SET 8 after execution of the third j instruction at 0x2200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2
0			WAY 3

LRU STATE = 100

SET 8 after execution of the fourth j instruction at 0x3200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2
1	00000000000000000011	j Set8_Block5 nop ... nop	WAY 3

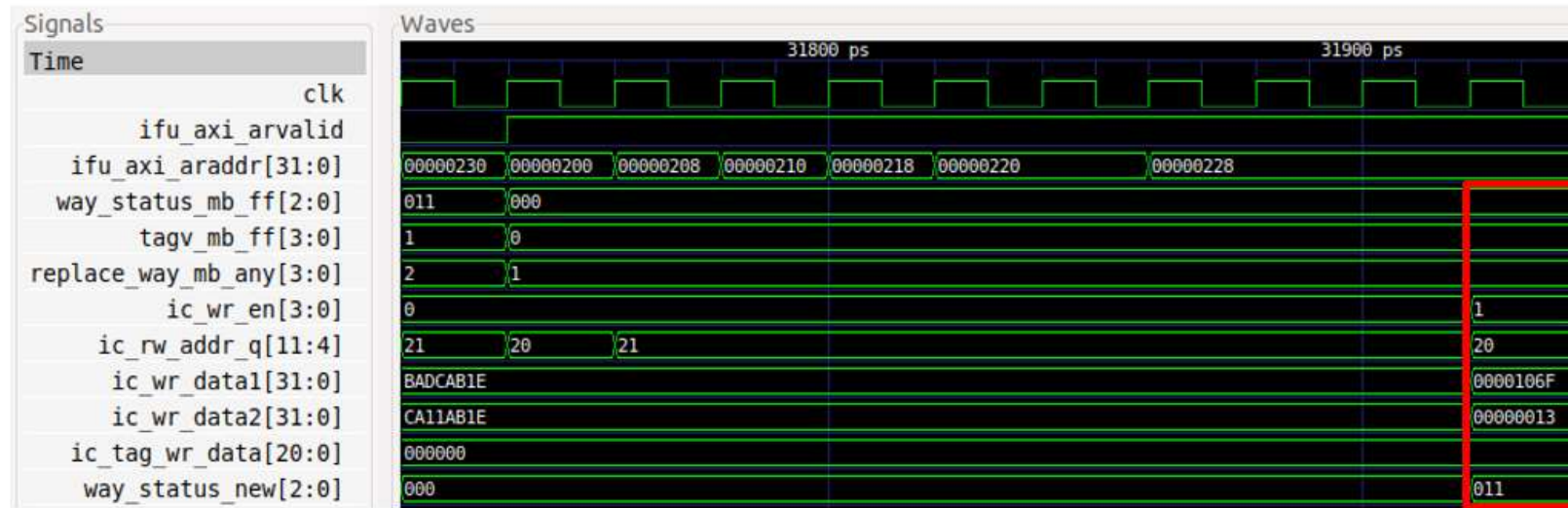
LRU STATE = 000

SET 8 after execution of the fifth j instruction at 0x4200

1	000000000000000000100	j Set8_Block1 nop ... nop	WAY 0
1	000000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	000000000000000000010	j Set8_Block4 nop ... nop	WAY 2
1	000000000000000000011	j Set8_Block5 nop ... nop	WAY 3

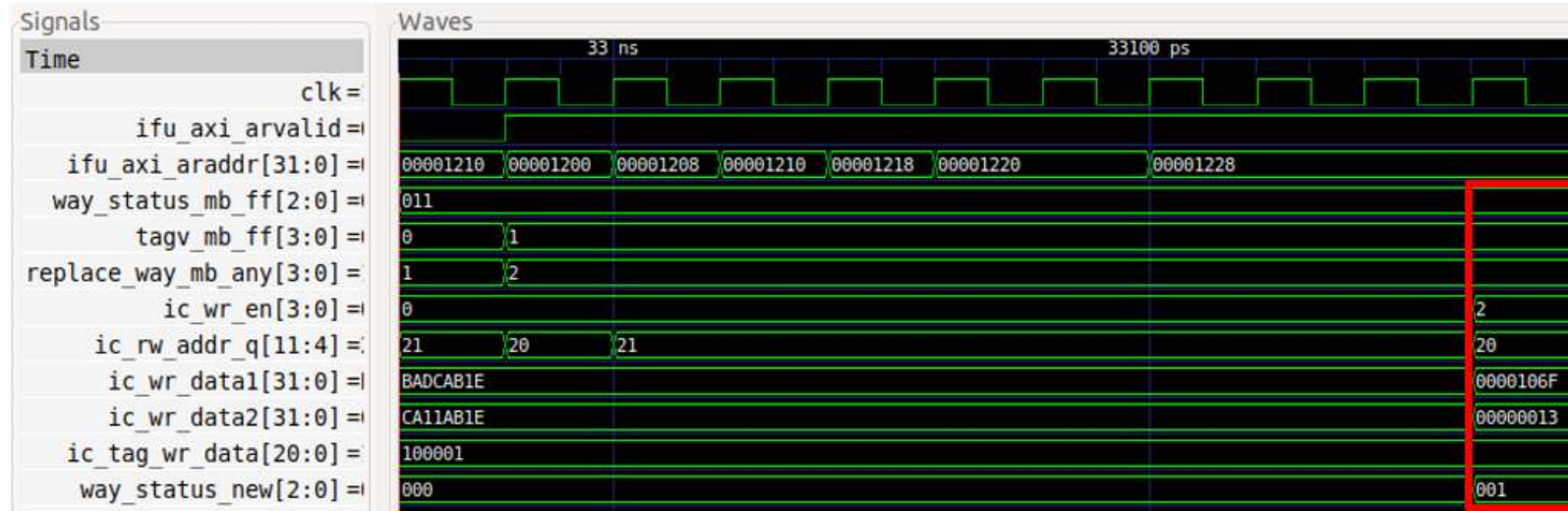
LRU STATE = 011

RVfpga Lab 19: Política de Substituição I\$ – 1º Salto



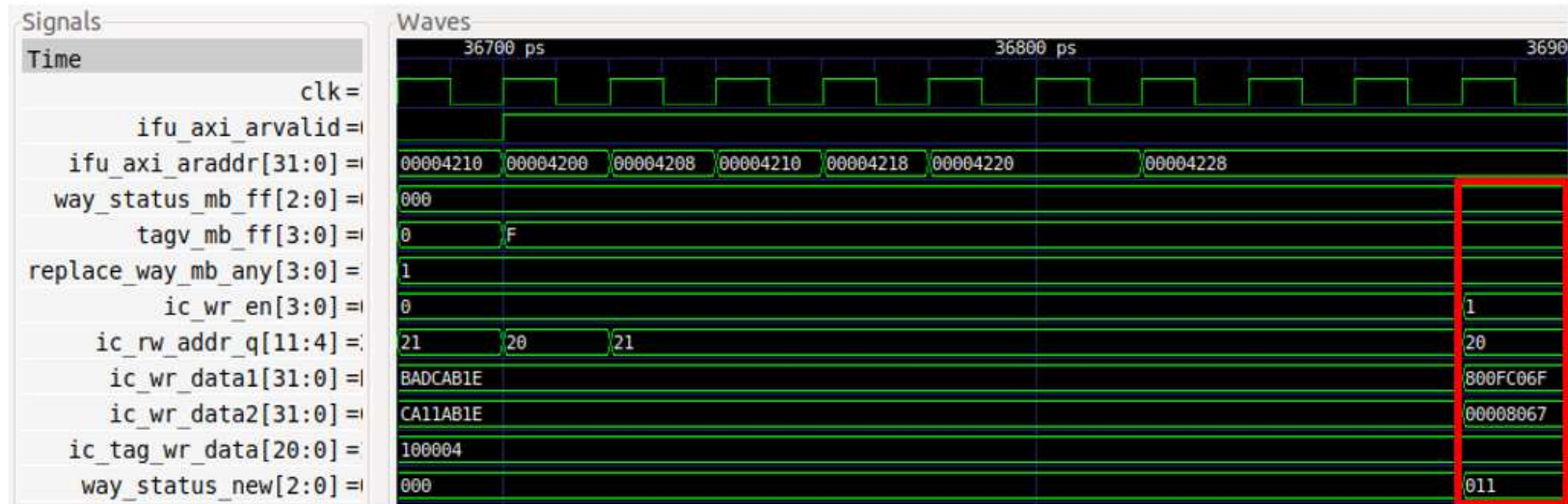
- O primeiro endereço do salto (0x200) mapeia o conjunto 8 dos I\$. Esse conjunto está inicialmente vazio, portanto, o novo bloco deve ser escrito na Via 0: `replace_way_mb_any = ic_wr_en = 0001`. O estado do set 8 da LRU é atualizado da seguinte forma: `way_status_new = 011`.
- O bloco I\$ é lido a partir da Memória DDR e escrito no I\$ em pedaços de 64 bits. A figura ilustra a escrita da etiqueta e as duas primeiras instruções do novo bloco no Set 8:
 - `ic_rw_addr_q[11:4] = 00100000` (SET 8)
 - `ic_tag_wr_data[19:0] = 0x0`
 - `ic_wr_data1[31:0] = 0x0000106F` (j Set8_Block2)
 - `ic_wr_data2[31:0] = 0x00000013` (nop)

RVfpga Lab 19: I\$ Política de Substituição – 2º Salto



- O endereço do segundo salto (0x1200) também mapeia o Set 8 de I\$. Apenas a Via 0 é válida nesse conjunto: tagv_mb_ff = 0001. Assim, o novo bloco deve ser escrito na Via 1: replace_way_mb_any = 1, ic_wr_en = 0010. O estado do Set 8 da LRU é actualizado da seguinte forma: way_status_new = 001.
- O bloco I\$ é lido a partir da Memória DDR e escrito no I\$ em blocos de 64 bits. A figura ilustra a escrita da etiqueta e as duas primeiras instruções do novo bloco no SET 8:
 - ic_rw_addr_q[11:4] = 00100000 (SET 8)
 - ic_tag_wr_data[19:0] = 0x1
 - ic_wr_data1[31:0] = 0x0000106F (j Set8_Block3)
 - ic_wr_data2[31:0] = 0x00000013 (nop)

RVfpga Lab 19: Política de Substituição I\$ – 5º Salto



- O endereço do quinto salto (0x4200) também mapeia o Set 8 de I\$. No entanto, neste caso, o conjunto (Set) está cheio : `tagv_mb_ff` = 1111. Assim, o novo bloco deve ser escrito no Via 1: `replace_way_mb_any` = `ic_wr_en` = 0001. O estado do Set 8 da LRU é actualizado da seguinte forma : `way_status_new` = 011.
- O bloco I\$ é lido da Memória DDR e escrito no I\$ em pedaços de 64 bits. A figura ilustra a escrita da etiqueta e as duas primeiras instruções do novo bloco no SET 8:
 - `ic_rw_addr_q[11:4]` = 00100000 (SET 8)
 - `ic_tag_wr_data[19:0]` = 0x4
 - `ic_wr_data1[31:0]` = 0x800fc06f (j Set8_Block1)
 - `ic_wr_data2[31:0]` = 0x00008067 (ret)

RVfpga Lab 19: Tarefas e Exercícios - Exemplos

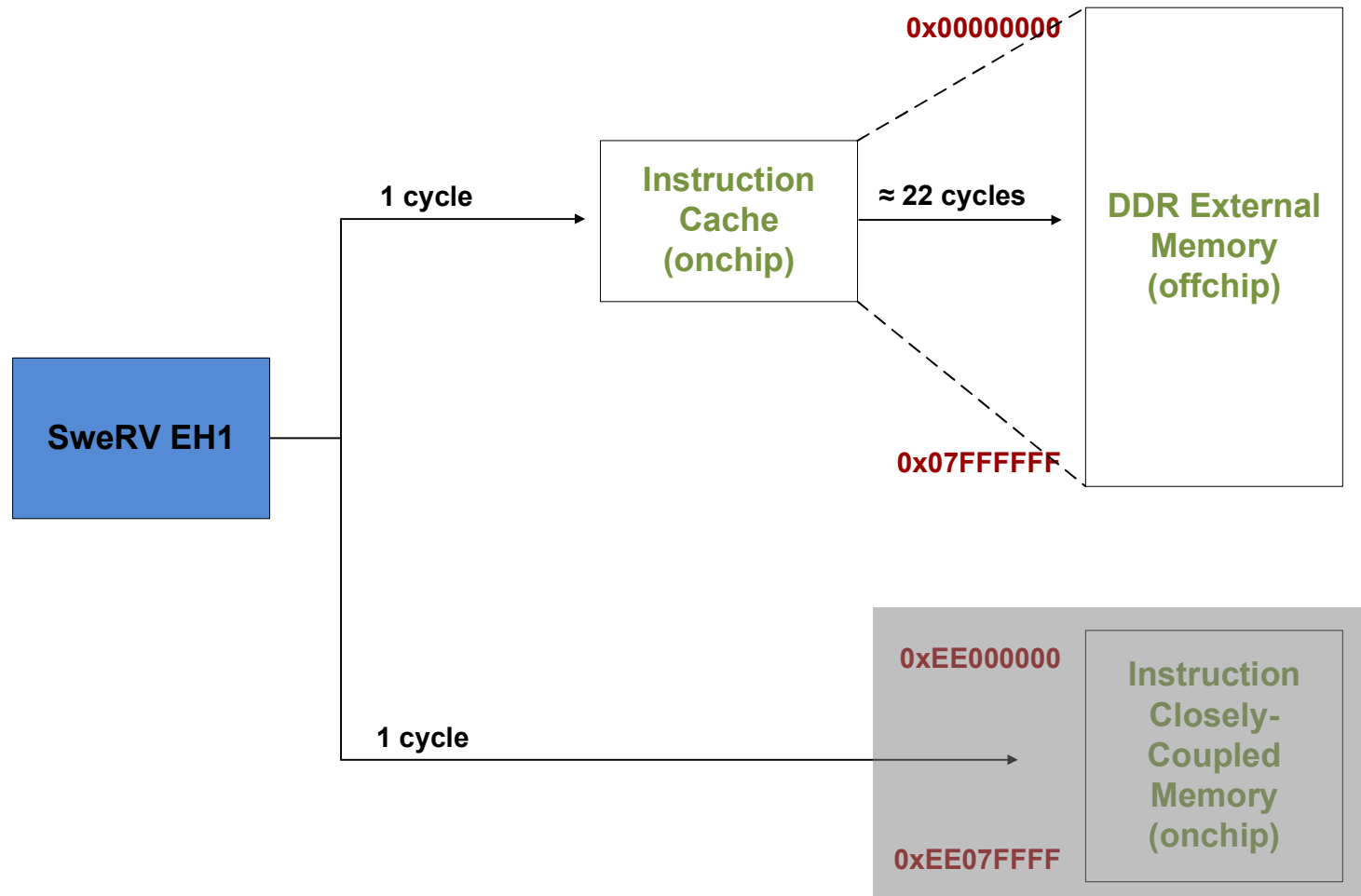
- **TAREFA:** Usando os contadores HW, meça o número de ciclos, instruções, leituras e escritas no programa da Figura 2. Quanto tempo no total (tanto para leitura como para escrita) é necessário para aceder à memória externa DDR?
- **TAREFA:** Utilize o exemplo de `[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem` para estimar a latência de leitura da memória externa DDR utilizando os contadores HW.
- **TAREFA:** Um exercício bastante complexo mas muito interessante é analisar o controlador de memória usado no sistema RVfpga. Lembre-se que pode encontrar os módulos que compõem este controlador na pasta `[RVfpgaPath]/RVfpga/src/LiteDRAM`, e que o módulo de topo está implementado no ficheiro `litedram_top.v` dentro dessa pasta. Pode começar com a simulação da Figura 3 e adicionar e analisar alguns sinais do controlador LiteDRAM.
- **Exercício 4)** Analise em simulação e na placa outras configurações de I\$, como um I\$ com um tamanho de bloco diferente. Recorde-se que o número de vias não pode ser modificado.
- **Exercício 5)** Analise a lógica que verifica a correção das informações de paridade do *Data Array* e do *Tag Array*.

Lab 20: ICCM, DCCM, e Benchmarking

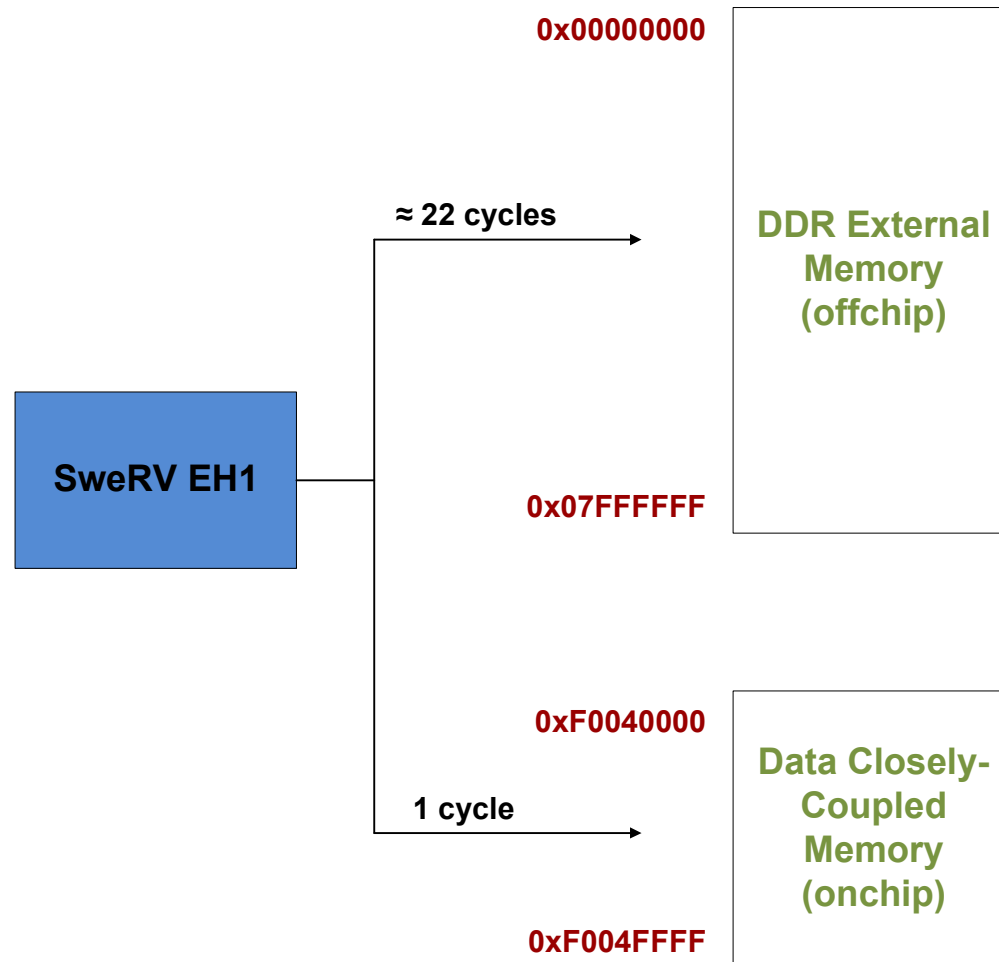
RVfpga Lab 20: ICCM, DDCM e Benchmarking

- **Memórias Locais (*Scratchpad*) SPM:**
 - Memória de instruções estreitamente acoplada - *Instruction Closely-Coupled Memory (ICCM)*
 - Memória de dados estreitamente acoplada - *Data Closely-Coupled Memory (DDCM)*

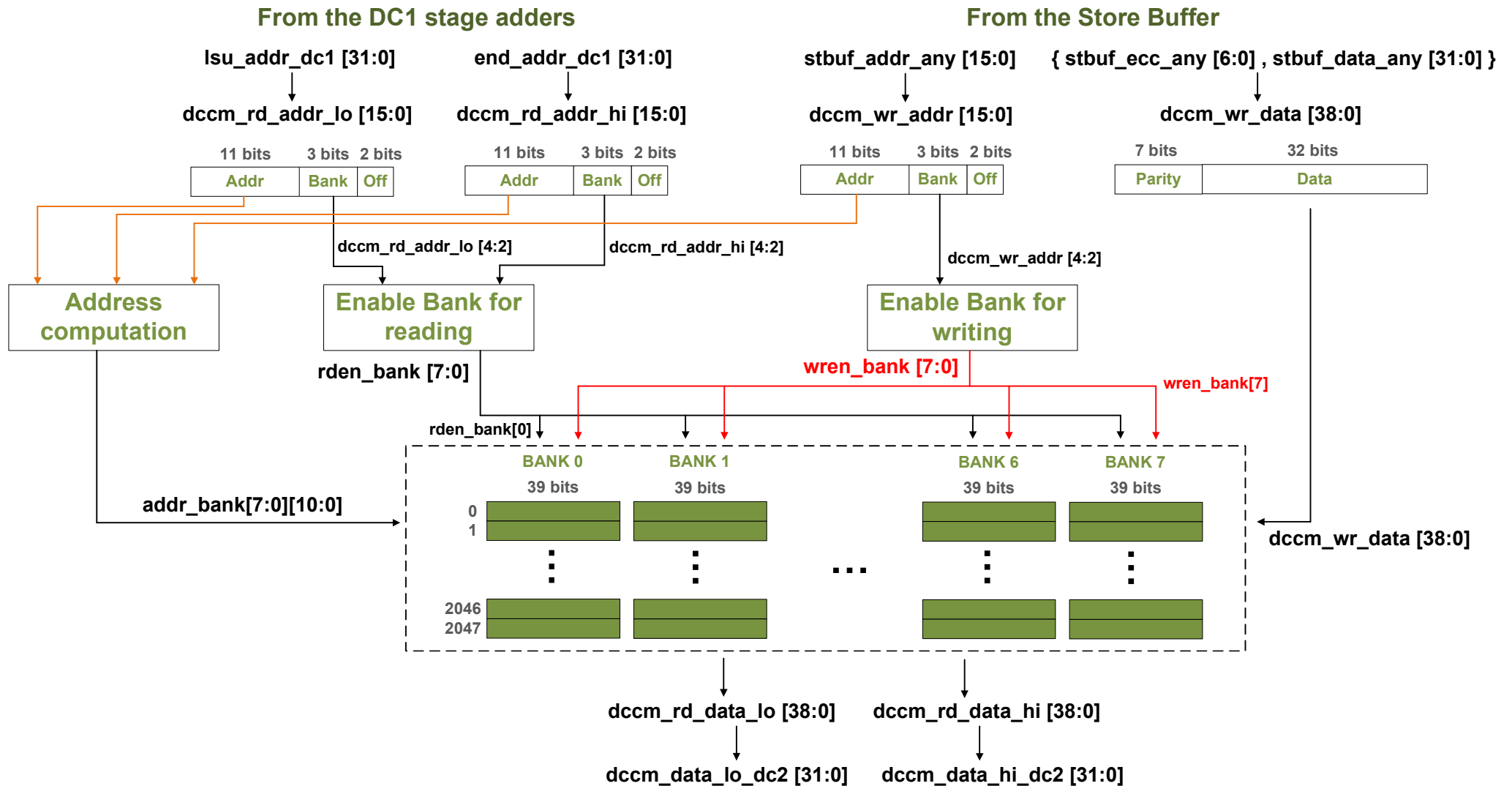
RVfpga Lab 20: Espaço de Endereçamento (Instruções)



RVfpga Lab 20: Espaço de Endereçamento (Dados)



RVfpga Lab 20: Configuração e Operação da ICCM

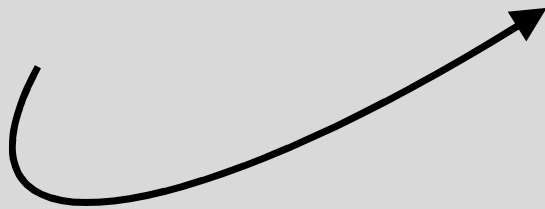


RVfpga Lab 20: Accesso à ICCM - Exemplo

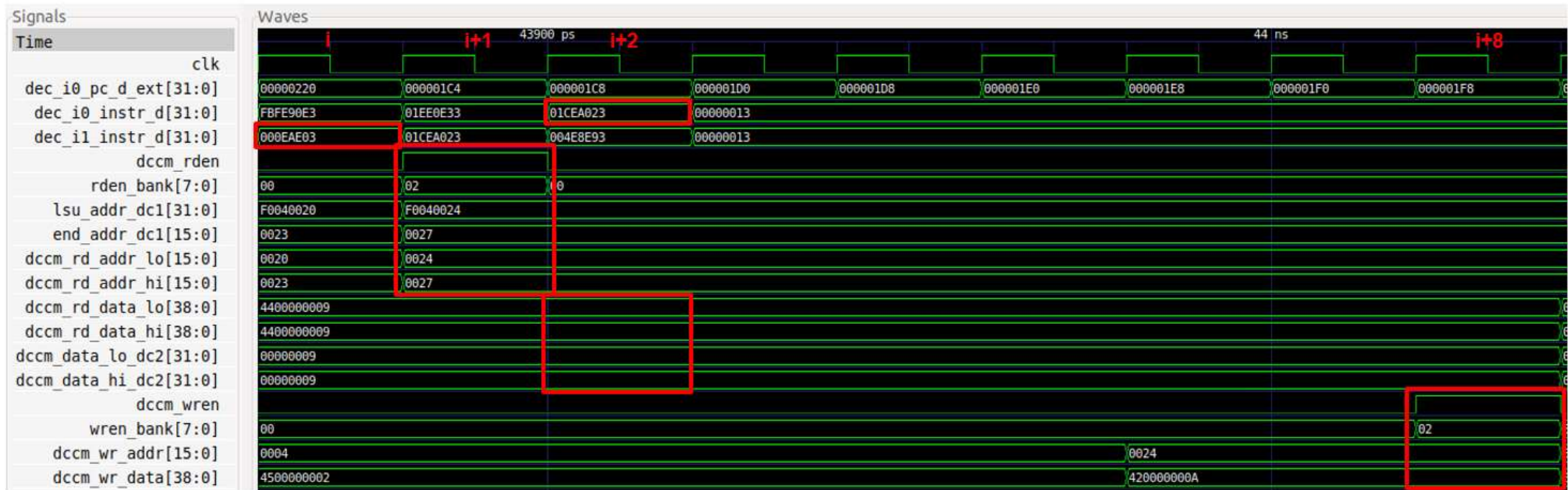
```
// Access array  
la t4, D  
li t5, 50  
li t0, 1000  
la t6, D  
add t6, t6, t0  
li t5, 1
```

REPEAT_Access:

```
lw t3, (t4)  
add t3, t3, t5  
sw t3, (t4)  
add t4, t4, 4  
INSERT_NOPS_10  
INSERT_NOPS_10  
bne t4, t6, REPEAT_Access
```



RVfpga Lab 20: Accesso à ICCM - Exemplo



RVfpga Lab 20: Acesso à ICCM - Exemplo

- **Ciclo i :** A instrução `lw` é decodificado na Via #1 : `dec_i1_instr_d = 0x000eae03`.
- **Ciclo $i+1$:** O endereço é gerado na etapa DC1 e fornecido ao DCCM :
Como resultado da verificação de endereço, a leitura do DCCM é ativada: `dccm_rden = 1`. Apenas o segundo banco é lido, pois o acesso é alinhado por palavras: `rden_bank = 0x02` (em binário 00000010).
- **Ciclo $i+2$:** Os dados lidos são obtidos a partir da DCCM e fornecidos ao núcleo.
- **Ciclo $i+8$:** Depois de adicionar 1 (o imediato) ao valor de leitura (`0x00000009 + 1 = 0x0000000A`) e atravessar o Store Buffer, como explicado no Laboratório 13, os dados e o endereço são fornecidos ao DCCM e a escrita do banco correcto é ativada.

RVfpga Lab 20: Benchmarking

- **Medidas de Desempenho (Benchmarks):**
 - Executar um conjunto de programas no processador
 - Comparar processadores
- **Dois indicadores comuns:**
 - CoreMark
 - Dhrystone
- Benchmarks usam **contadores hardware** (HW Counters) para medir eventos (como o número de instruções, o número de ciclos).

Contadores hardware no RISC-V

- Registos para fins especiais para registar o desempenho e outros parâmetros (apresentado em baixo).

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

Tabela 7-2 do Manual de Referência do Programador SweRV EH1: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

Como Usar e Inicializar Contadores de Hardware

- **Incluir o PSP (Platform Support Package) da Western Digital:**
 - `#include <psp_api.h>`
- **ativar contadores:**
 - `pspEnableAllPerformanceMonitor(1);`
- **Definir contadores para medir várias métricas :**
 - `pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);`
 - `pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);`
- **Ler métricas:**
 - `cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);`
 - `instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);`
- **Impressão de métricas:**
 - `printfNexys("Cycles = %d", cyc_end - cyc_beg);`
 - `printfNexys("Instructions = %d", instr_end - instr_beg);`

Programa de Exemplo com Contadores em Hardware

```
// Inclui ficheiros header do bsp (board support package) - ver ficheiros Lab 20
#include <psp_api.h>
int main(void) {
    int cyc_beg, cyc_end, instr_beg, instr_end;

    uartInit();

    pspEnableAllPerformanceMonitor(1); // ativar contadores

    pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE); // atribuir
    pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL); // contadores

    cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0); // ler contadores
    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);

    Test_Assembly();

    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0); // ler contadores
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);

    printfNexys("Ciclos = %d", cyc_end-cyc_beg); // imprimir valores
    printfNexys("Instruções = %d", instr_end-instr_beg);
}
```



RISC-V®

RVfpga v2.2 © 2022 <267>
Imagination Technologies



RVfpga Lab 20: Métricas

- **Métricas CoreMark**
 - O CoreMark executa várias iterações de um ciclo.
 - **CoreMark Score (CM)**: O número de iterações que completa por segundo (ou seja, as iterações/segundo).
 - **CM/MHz**: CM dividido pela frequência do relógio em MHz (também designado por Iterat/Sec/MHz ou iterações/segundo/MHz).
- Recordar: **IPC ideal** (instruções por ciclo) é **2** para o SweRV EH1 porque é um super-escalar com 2-vias.

RVfpga Lab 20: CoreMark Sob Várias Condições

	Compilador = depuração Memoria Externa	Compilador = depuração DCCM	Compilador = DCCM otimizada
CM/MHz	0.47	1.88	3.47
# Instruções	~0.5 milhões	~0.5 milhões	0.309 milhões
# Ciclos	~2 milhões	~0.5 milhões	0.288 milhões
IPC (instruções/ciclos)	0.25	~1	~1
Transacções do Barramento de Dados	~133,000 (todos vão para a memória externa)	0 (devido a DCCM)	0 (devido a DCCM)
Transacções do Barramento de Instruções	392 (devido a I\$)	392 (devido a I\$)	392 (devido a I\$)

RVfpga Lab 20: Tarefas e Exercícios - Exemplos

- **TAREFA:** Usando as instruções fornecidas no Lab 1, implemente um novo sistema RVfpga que inclua uma ICCM de 64 KiB.
- **TAREFA:** Simular uma leitura não alinhada para a DCCM e analisar o modo como esta é tratada no interior da DCCM.
- **TAREFA:** Simular um conflito de banco DCCM modificando o programa da Figura 4.
- **TAREFA:** Modifique o ficheiro platformio.ini para utilizar a DCCM para armazenar a maioria dos dados e a ICCM para armazenar as instruções. Execute o teste de referência *CoreMark* e compare os resultados com os obtidos nesta secção.
- **TAREFA:** Modificar a optimização de compilação para -O3 e explicar os resultados.
- **Exercício 1)** Faça a mesma análise que foi feita para o *CoreMark*, mas desta vez usando o benchmark *Dhrystone*.
- **Exercício 2)** Faça a mesma análise que foi efectuada para o *CoreMark*, mas desta vez para a aplicação *ImageProcessing* do Laboratório 4.
- **Exercício 3)** ativar/desativar várias funcionalidades principais. Compare os resultados de desempenho. Execute os três programas (*CoreMark*, *Dhrystone* e *ImageProcessing*) nestes sistemas RVfpga modificados na placa Nexys A7.

Agradecimentos

AUTORES

Prof. Sarah Harris
Prof. Daniel Chaver
Zubair Kakakhel
M. Hamza Liaqat

ORIENTADOR

Prof. David Patterson

COLABORADORES

Robert Owen
Olof Kindgren
Prof. Luis Piñuel
Ivan Kravets
Valerii Koval
Ted Marena
Prof. Roy Kravitz
Prof. Peng Liu

ASSOCIADOS

Prof. José Ignacio Gómez
Prof. Christian Tenllado
Prof. Daniel León
Prof. Katzalin Olcoz
Prof. Alberto del Barrio
Prof. Fernando Castro
Prof. Manuel Prieto

Prof. Francisco Tirado
Prof. Román Hermida
Prof. Julio Villalba
Prof. Aatur Patwary
Cathal McCabe
Dan Hugo
Braden Harwood
Prof. David Burnett

Gage Elerding
Prof. Brian Cruickshank
Deepen Parmar
Thong Doan
Oliver Rew
Niko Nikolay
Guanyang He
Prof. Peng Liu

Patrocinadores e Apoiantes

Western Digital

 **Imagination**

 **CHIPS
ALLIANCE**

 **RISC-V**

 **DIGILENT**
A National Instruments Company

 **XILINX**
| UNIVERSITY PROGRAM

 **Digi-Key**
ELECTRONICS

 **Esperanto**
TECHNOLOGIES

 **codasip**

 硬禾学堂

 **ANDES**
TECHNOLOGY

 **PLATFORMIO.ORG**

 **RISC-V**

RVfpga v2.2 © 2022 <271>
Imagination Technologies

 **Imagination**