RVfpga

The Complete Course in Understanding Computer Architecture

RVfpga Introduction

Acknowledgements

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

ADVISER
Prof. David Patterson

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

ASSOCIATES
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 Ataur 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

Sponsors and Supporters

RVfpga Introduction

- RISC-V FPGA (**RVfpga**): course that shows how to:
  - **Target** SweRV **commercial RISC-V core** & system-on-chip (SoC) to **FPGA**
  - **Program** the RISC-V SoC
  - **Add more functionality** to the RISC-V SoC
  - **Analyze and modify** the RISC-V core and memory hierarchy
- The package is being developed by **Imagination Technologies** and its academic and industry partners.
- After completing the RVfpga Course, users will walk away with a **commercial RISC-V processor, SoC, and ecosystem** that they understand and know how to use and modify.

This section of slides covers material in the **Getting Started Guide (GSG) Section 1**.

Two RVfpga Courses

- Imagination Technologies offers two courses based on the RISC-V architecture:
  - **RVfpga:** the course covered in these slides, about the RISC-V core, memory system, and peripherals
  - **RVfpga-SoC:** a second course that shows how to:
    - **Build a RISC-V SoC** from building blocks
    - Install the **Zephyr** RTOS (real-time operating system)
    - Run **programs** on Zephyr
    - Run simple **Tensorflow** programs

    (RVfpga-SoC is not covered in these slides)

Download Material from Imagination Technologies

- Both courses (RVfpga and RVfpga-SoC) are available as separate downloads (free upon registration) at:

  https://university.imgtec.com/rvfpga/

- This EdX course covers the **RVfpga** Course.

RVfpga Audience & Track Record

- **Target Audience**
  - Undergraduates and master's students in electrical engineering, computer science, or computer engineering
  - Academics & industry professionals interested in learning the RISC-V architecture
- **Imagination University Programme (IUP) Track Record:** Developed MIPSfpga Program:
  - Launched in April 2015
  - Engaged 800 universities
  - Winner: Elektra Best Educational Support Award, Europe 2015

RVfpga
Course Overview

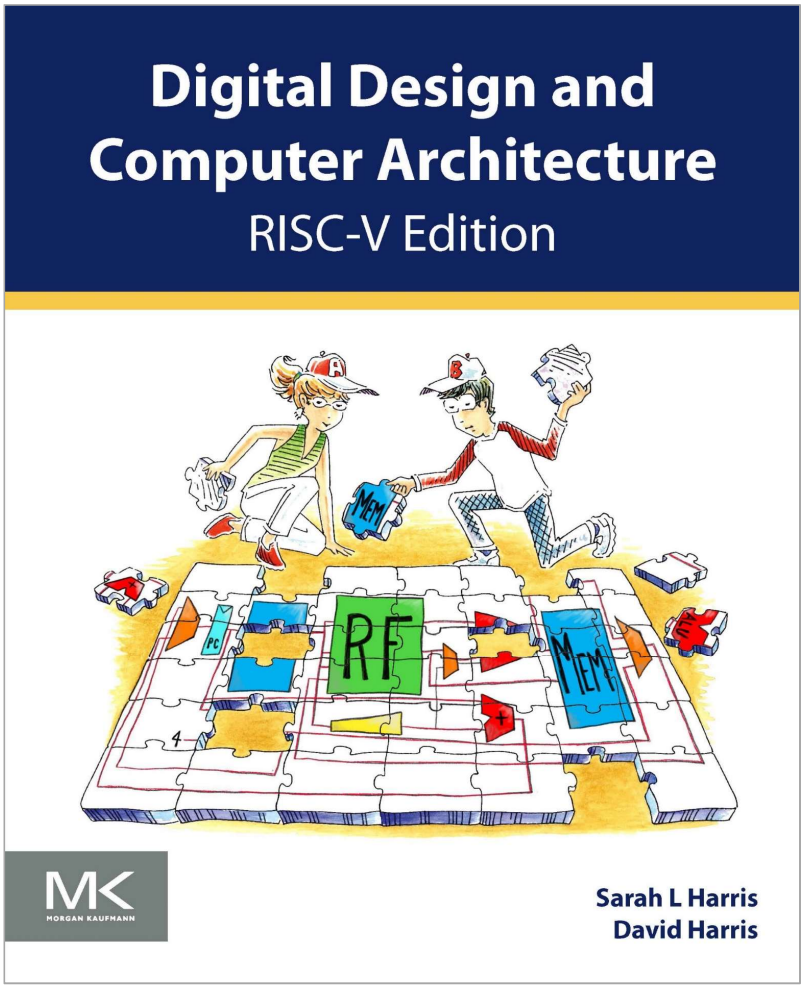
RVfpga Course

- **Typically 2-3 Semester Course**
  - Undergraduate (Labs 1-10)
  - Master's / upper division (Labs 11-20)
- **Expected Prior Knowledge**
  - Digital design
  - High-level programming (preferably C)
  - Instruction set architecture / assembly programming
  - Microarchitecture
  - Memory systems
  - This material is covered in ***Digital Design and Computer Architecture: RISC-V Edition***, Harris & Harris, © Elsevier 2021.
  - These topics are expanded on with hands-on learning throughout RVfpga course.

This section of slides covers material in the **Getting Started Guide (GSG) Section 1**.

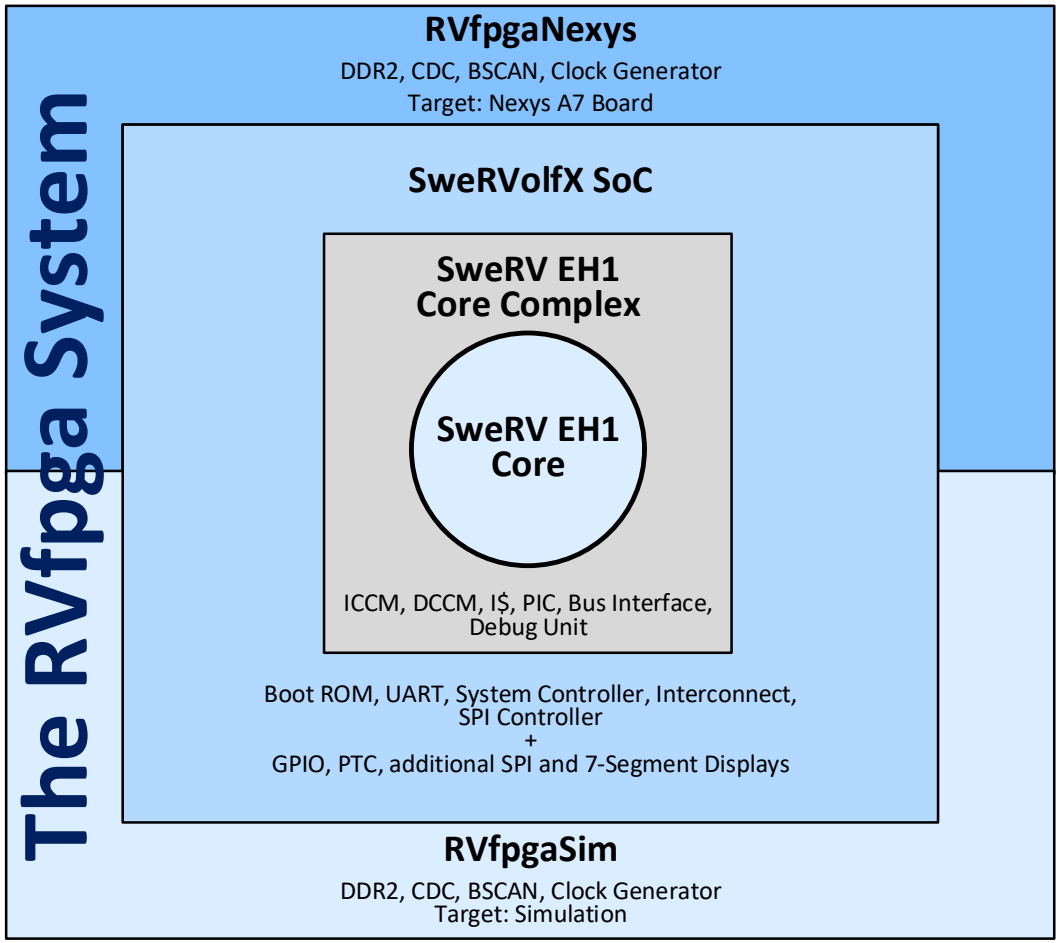
Textbook

Recommended text to understand before starting the RVfpga course: ***Digital Design and Computer Architecture: RISC-V Edition***, Harris & Harris, © Elsevier, 2021



ISBN-10: 0128200642 ISBN-13: 978-0128200643

RVfpga System



SweRV Core™

- Open-source core from Western Digital
- 2-way superscalar core
- 9-stage pipeline
- In-order
- RV32IMC

RVfpga Required Software and Hardware

SOFTWARE

Xilinx **Vivado** 2019.2 WebPACK

PlatformIO – an extension of Microsoft's **Visual Studio Code –** with Chips Alliance platform, which includes: RISC-V Toolchain, OpenOCD, Verilator HDL Simulator, WD Whisper instruction set simulator (ISS) **Verilator** and **GTKWave**

HARDWARE*

Digilent's **Nexys A7** / Nexys 4 DDR FPGA Board

*Optional: All labs can be completed in simulation only; so this hardware is recommended but not required.

RISC-V CORE & SOC

Core: Western Digital's **SweRV EH1****

SoC: Chips Alliance's **SweRVolf****

**Open-source – and provided as part of RVfpga package.

All are free except the optional Nexys A7 FPGA board, which costs $265 (academic price: $199)

Nexys A7-100T FPGA Board: Optional

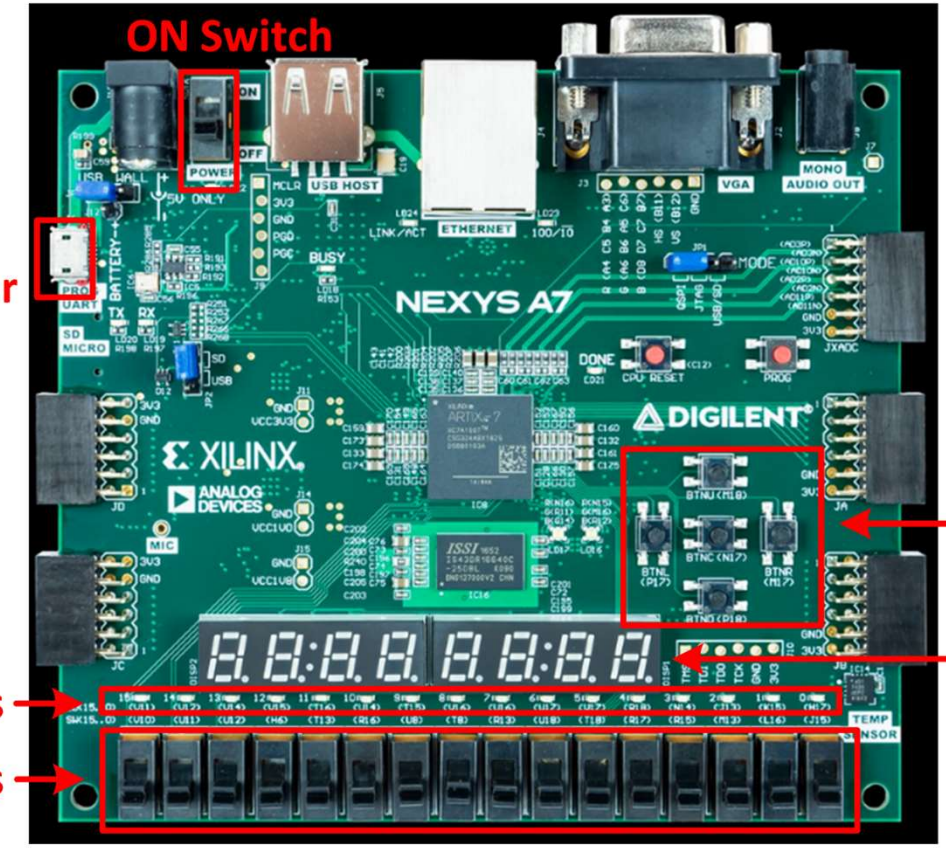


Figure of board from https://reference.digilentinc.com/

- Contains **Artix-7** field programmable gate array (FPGA)
- Includes **peripherals** (i.e., LEDs, switches, pushbuttons, 7-segment displays, accelerometer, temperature sensor, microphone, etc.)
- Available for purchase at **digkey.com**, **digilentinc.com**, and other vendors

We will support the **Basys3** FPGA board soon.

Supported Platforms

- **Operating Systems**
  - **Ubuntu 18.04** and **20.04**
  - **Windows 10**
  - **macOS**

RVfpga Course Overview

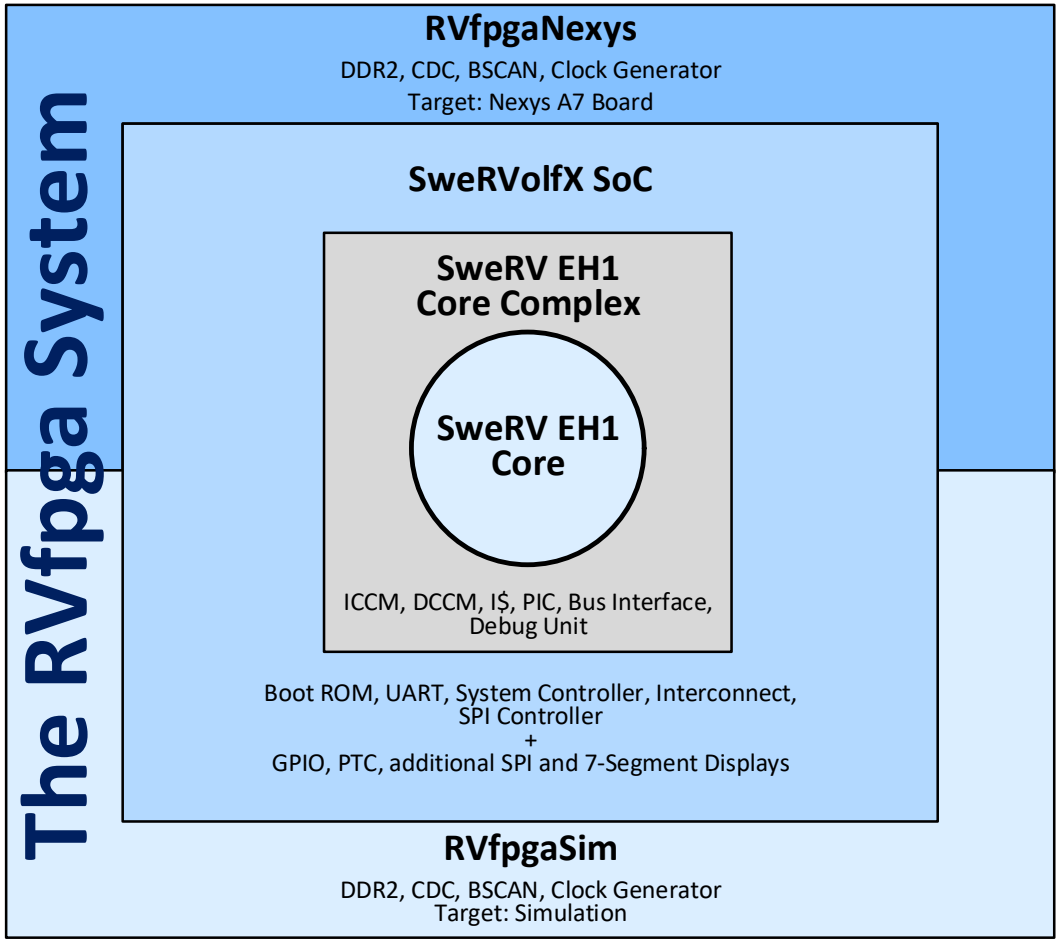
Introduction

- The RVfpga System is an extension of Chips Alliance's **SweRVolf SoC**, which is based on Western Digital's RISC-V **SweRV EH1** core.
- The source code for the SoC and core are provided with the RVfpga download from Imagination Technologies.
- The RVfpga System is also simply referred to as "RVfpga".

This section of slides covers material in the **Getting Started Guide (GSG) Section 1**.

RVfpga Hierarchy

- **SweRV EH1 Core / Core Complex**
  - Includes processor, memory, and bus interface
- **SweRVolfX SoC**
  - Extended version of SweRVolf
  - Adds peripherals
- **RVfpga System**
  - **RVfpgaNexys:** SweRVolfX targeted to hardware (Nexys A7 FPGA board, with on-board memory, clock, etc.)
  - **RVfpgaSim:** SweRVolfX targeted to simulation



RVfpga Hierarchy

Name	Description
SweRV EH1 Core	Open-source commercial RISC-V core developed be Western Digital (https://github.com/chipsalliance/Cores-SweRV).
SweRV EH1 Core Complex	SweRV EH1 core with added memory (ICCM, DCCM, and instruction cache), programmable interrupt controller (PIC), bus interfaces, and debug unit (https://github.com/chipsalliance/Cores-SweRV).
SweRVolfX (Extended SweRVolf)	The System on Chip that we use in the RVfpga course. It is an extension of SweRVolf. SweRVolf (https://github.com/chipsalliance/Cores-SweRVolf): An open-source SoC built around the SweRV EH1 Core Complex. It adds a boot ROM, UART interface, system controller, interconnect (AXI Interconnect, Wishbone Interconnect, and AXI-to-Wishbone bridge), and an SPI controller. SweRVolfX: It adds 4 new peripherals to SweRVolf: a GPIO, a PTC, an additional SPI and a controller for the 8 digit 7-Segment Displays.

RVfpga Hierarchy

Name	Description
RVfpgaNexys	The SweRVolfX SoC targeted to the Nexys A7 board and its peripherals. It adds a DDR2 interface, CDC (clock domain crossing) unit, BSCAN logic (for the JTAG interface), and clock generator. RVfpgaNexys is the same as SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.
RVfpgaSim	The SweRVolfX SoC with a testbench wrapper and AXI memory intended for simulation. RVfpgaSim is the same as SweRVolf sim, (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.

SweRV EH1 Core and SweRV EH1 Core Complex

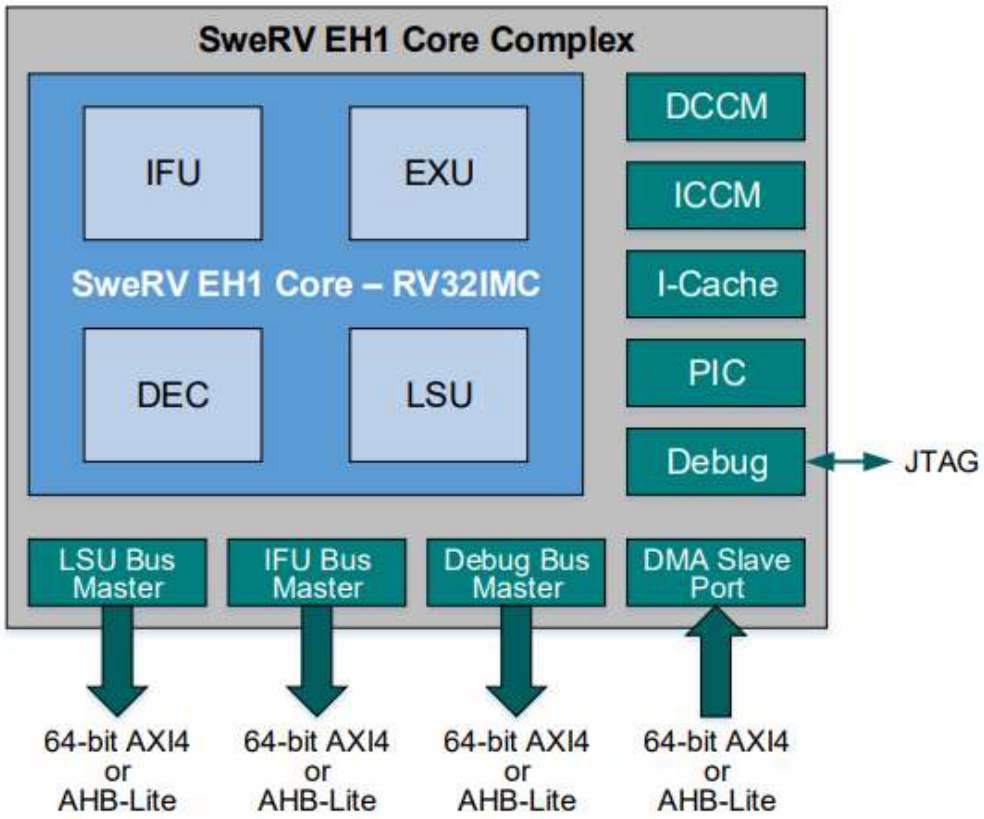
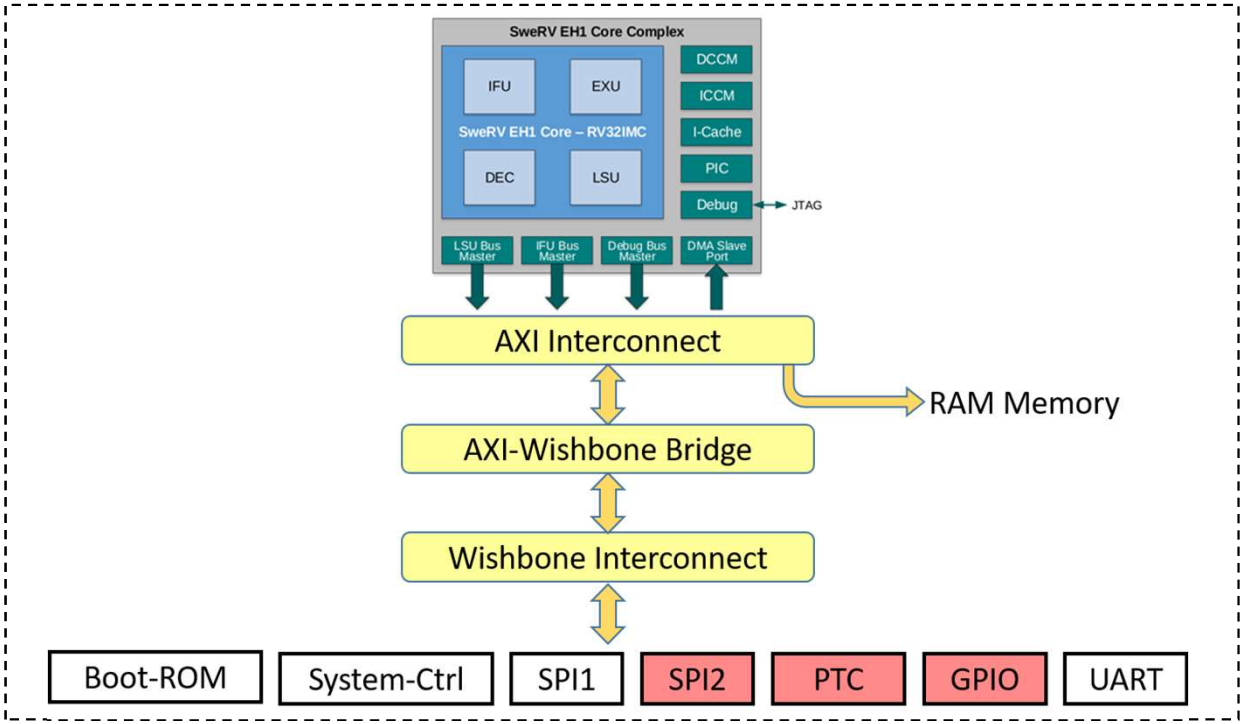


Figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

- Open-source core from Western Digital
- 32-bit (RV32ICM) superscalar core, with dual-issue 9-stage pipeline
- Separate instruction and data memories (ICCM and DCCM) tightly coupled to the core
- 4-way set-associative I$ with parity or ECC protection
- Programmable Interrupt Controller
- Core Debug Unit compliant with the RISC-V Debug specification
- System Bus: AXI4 or AHB-Lite

SweRVolfX SoC

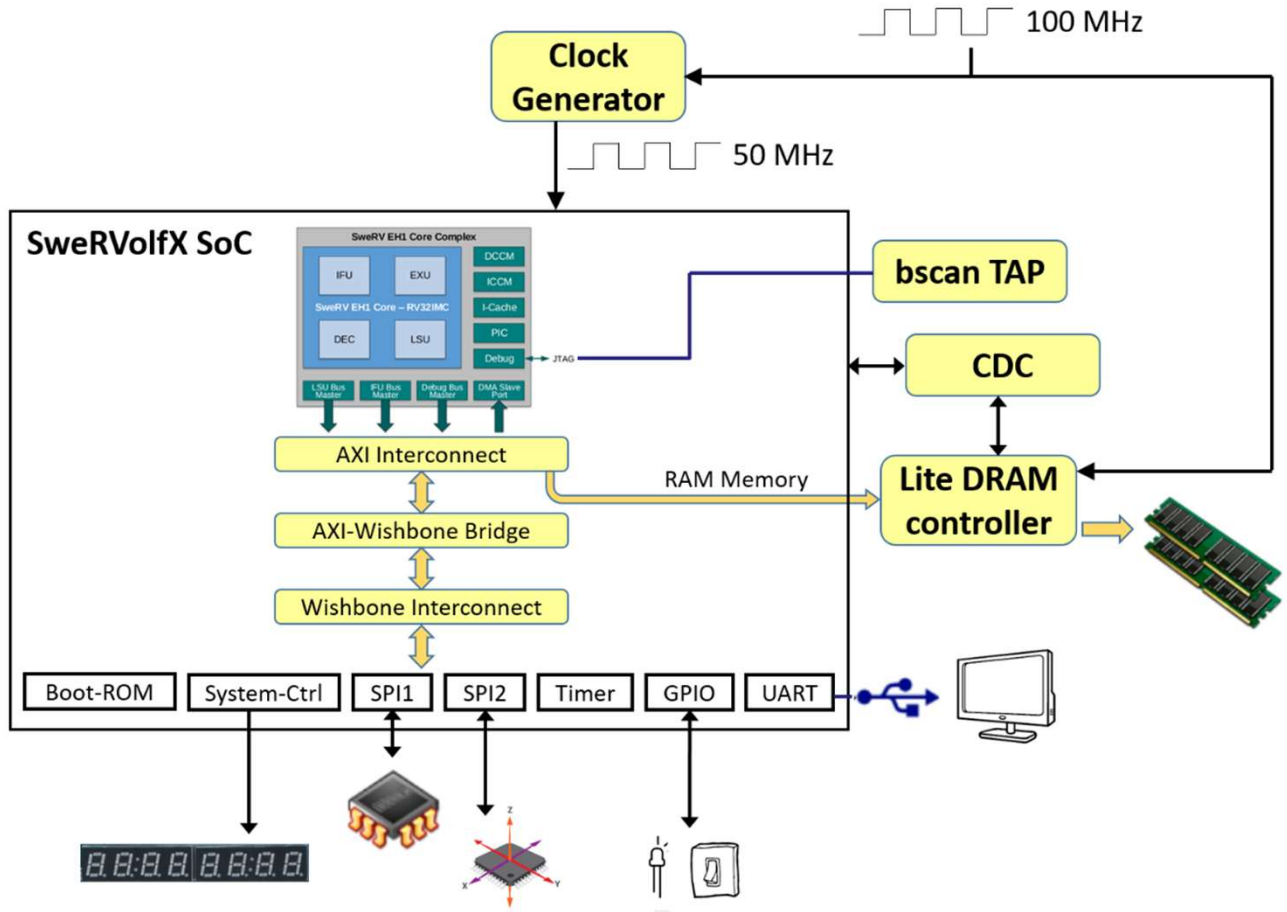


SweRVolfX Memory Map

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

- Open-source system-on-chip (SoC) from Chips Alliance
- SweRVolf uses the SweRV EH1 Core. SweRVolf includes a Boot ROM, UART, and a System Controller and an SPI controller (SPI1)
- SweRVolfX extends SweRVolf with another SPI controller (SPI2), a GPIO (General Purpose Input/Output), 8-digit 7-Segment Displays and a PTC (shown in red).
- SweRV EH1 Core uses an AXI bus and peripherals use a Wishbone bus, so the SoC also has an AXI to Wishbone Bridge

RVfpgaNexys



- **RVfpgaNexys:** SweRVolfX SoC targeted to Nexys A7 FPGA board with added peripherals:
  - **Core & System:**
    - SweRVolfX SoC
    - Lite DRAM controller
    - Clock Generator, Clock Domain and BSCAN logic for the JTAG port
  - **Peripherals** used on Nexys A7 FPGA board**:**
    - DDR2 memory
    - UART via USB connection
    - SPI Flash memory
    - 16 LEDs and 16 switches
    - SPI Accelerometer
    - 8-digit 7-segment displays

RVfpgaSim

- **RVfpgaSim** is the SweRVolfX SoC wrapped in a testbench to be used by HDL simulators.

RVfpga System Extensions

- The SweRVolfX SoC is **further extended** in Labs 6-10:
  - Another **GPIO** controller to interface with the on-board Nexys A7 **pushbuttons**
  - Modification of the **7-segment displays** controller
  - New **timer** modules for using the on-board **tri-color LEDs**
  - New **external interrupt sources**

RVfpga Labs Overview

RVfpga Labs Overview

Part 1: Labs 1-10

- Programming
- Vivado Project & I/O Systems

Part 2: Labs 11-20

- RISC-V Core
- RISC-V Memory Systems
- RISC-V Benchmarking & Performance Monitoring

All labs include **exercises** for using and/or modifying the RVfpga System to increase understanding through hands-on design. RVfpga includes C and assembly example programs and solutions.

RVfpga Labs 1-4: Programming

- **Lab 1: C Programming:** Write a C program in PlatformIO, and run / debug it on RVfpgaNexys/RVfpgaSim/Whisper. Also introduce Western Digital's Board Support and Platform Support Packages (BSP and PSP) for supporting operations such as printing to the terminal.
- **Lab 2: RISC-V Assembly Language:** Write a RISC-V assembly program in PlatformIO and run /debug it on RVfpgaNexys/RVfpgaSim/Whisper.
- **Lab 3: Function Calls:** Introduction to function calls, C libraries, and the RISC-V calling convention.
- **Lab 4: Image Processing: C & Assembly:** Embed assembly code with C code.

RVfpga Labs 5-10: I/O & Peripherals

- **Lab 5: Creating a Vivado Project:** Build a Vivado project to target RVfpgaNexys to an FPGA board and simulate RVfpgaSim in Verilator.
- **Lab 6: Introduction to I/O:** Introduction to memory-mapped I/O and the RVfpga System's open-source GPIO module.
- **Lab 7: 7-Segment Displays:** Build a 7-segment display decoder and integrate it into the RVfpga System.
- **Lab 8: Timers:** Understand and use Timers and a Timer controller.
- **Lab 9: Interrupt-driven I/O:** Introduction to the RVfpga System's interrupt support and use of interrupt-driven I/O.
- **Lab 10: Serial Buses:** Introduction to serial interfaces (SPI, I2C, and UART). Show how to use the onboard accelerometer that uses an SPI interface.

RVfpga Labs 11-20: The RISC-V Core

- **Lab 11:** Understanding the SweRV EH1 configuration, core structure, and performance monitoring.
- **Labs 12, 13, 16:** Examining instruction flow through the pipeline (Arithmetic/Logic, Memory, Jumps, and Branches).
- **Labs 14-16:** Understanding hazards and how to deal with them
- **Lab 16:** Understanding and modifying the branch predictor
- **Lab 17:** Exploring superscalar execution.
- **Lab 18:** Adding new instructions and hardware counters.
- **Lab 19:** Understanding the memory hierarchy and I$.
- **Lab 20:** Enabling the ICCM and DCCM (instruction and data closely-coupled memories) and using benchmarking to compare performance.

Directories Hierarchy

Directory hierarchy of RVfpga materials

- *RVfpga/Documents* folder: Documents (GSG, Slides, Labs, …).
  - *LabInstructions* folder:
    - Instructions for each Lab + Figures used in the instructions for each lab.
- *RVfpga/verilatorSIM* folder: Sources for Verilator simulator
- *RVfpga/src* folder: Verilog sources for the SoC
- *RVfpga/examples* folder: PlatformIO projects for the GSG examples
- *RVfpga/Labs* folder:
  - Folders *Lab1*,…, *Lab20*: Resources to be used while completing the labs.
  - *RVfpgaLabsSolutions* folder: Exercise solutions for each of the labs.
    - *ProgramsAndDocuments* folder: Solutions for the proposed tasks and exercises.
    - *Modified_RVfpgaSystem* folder: Modified RVfpga System as guided by the exercises in Labs 6-10 and Lab 18. Solutions for the exercises + Instructions.

RVfpga Installing Tools

RVfpga Software Tools (Section 5 GSG)

- **Xilinx's Vivado IDE**
  - View RVfpga source files (Verilog / SystemVerilog) and hierarchy
  - Create bitfile (FPGA configuration file) for RVfpga targeted to Nexys A7 board
- **Visual Studio Code (VSCode) + PlatformIO**
  - PlatformIO: an extension of VSCode
  - Download the RVfpga System onto the Nexys A7 board
  - Compile, download, run, and debug C and assembly programs on the RVfpga System
- **Verilator** – an HDL (hardware description language) simulator
  - Simulate the RVfpga System at HDL (low) level to analyze its internal signals
- **GTKWave** – wave viewer

This section of slides covers material in the **Getting Started Guide (GSG) Sections 2 & 5**.

Installation of minimal tools (Section 2 GSG)

- VSCode
  - Download VSCode for Linux, Windows or MacOS
  - Install it in your system
- Install PlatformIO on top of VSCode
  - In Linux: Install python3 utilities
  - Extensions Icon: Look for PlatformIO and install it
  - Install Nexys A7 drivers:
    - Linux: Use provided folder
    - Windows: Use Zadig application (Appendix)
    - Mac OS: Not necessary
- Install GTKWave following the instructions for your OS

Initial Examples

LEDs-Switches – Execution on the Board (Sections 6.A and 6.E of GSG)

- Connect the Nexys A7 board.
- Open VSCode and PlatformIO.
- Click on **File → Open Folder** and select:

  [RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang
- Analyse the source code of the program: *src/LedsSwitches_C-Lang.c.*
- The first time an RVfpga example opens in PlatformIO, the Chips Alliance platform gets automatically installed. It includes the pre-built RISC-V toolchain, OpenOCD, the Verilator simulator, etc.
- Download RVfpgaNexys to the Nexys A7 board. You first have to update the path. You may need to refresh the Project Tasks window.
- Download and execute example LEDs-Switches.

This section of slides covers material in the **Getting Started Guide (GSG) Sections 6-8**.

AL_Operations Example – Execution on the Board (Section 6.B of GSG)

- If not opened yet, open VSCode and PlatformIO
- Click on **File → Close Folder** from the top file menu. Then click on **File → Open Folder** and select:

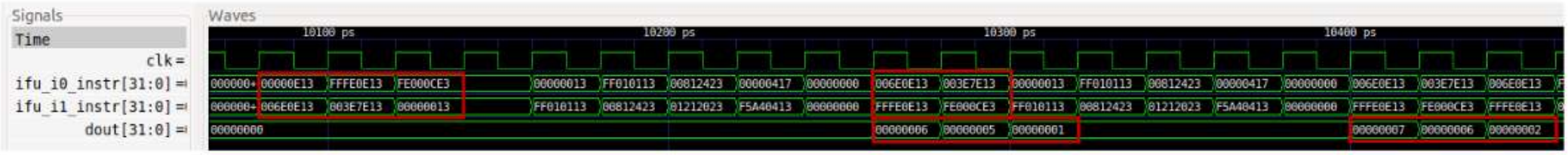
  [RVfpgaPath]\RVfpga\examples\AL_Operations
- Analyse the source code of the program
- If necessary, download RVfpgaNexys to the Nexys A7 board.
- Download, execute and debug example AL_Operations.

AL_Operations Example – Simulation in Whisper (Section 8 of GSG)

- Click on *File* → *Open File* and double-click on *[RVfpgaPath]/RVfpga/examples/AL_Operations/platformio.ini*, and set **whisper** as the debug tool by uncommenting line 17.
- Launch the debugger as usual
- You can now debug the program exactly as you did in Section 6.B, but this time the program is running in simulation on Whisper instead of on the Nexys A7 FPGA board.

AL_Operations Example – Simulation in Verilator (Section 7 of GSG)

- Open file *platformio.ini*. Establish the path to RVfpgaSim.
- Run the simulation by clicking on the PlatformIO icon 
- Expand Project Tasks → env:swervolf_nexys → Platform and click on Generate Trace
- A few seconds after the previous step, file *trace.vcd* should have been generated and you can open it with GTKWave
- Add signals: click on *File – Read Tcl Script File* and select *[RVfpgaPath]/RVfpga/examples/AL_Operations/test.tcl*



HelloWorld Example (Section 6.F of GSG)

1. Open VSCode and PlatformIO. Click on **File → Close Folder**. Click on **File → Open Folder** from the top file menu and select:

   [RVfpgaPath]\RVfpga\examples\HelloWorld_C-Lang

2. Configure the system:
   - PlatformIO serial monitor: Use *monitor_speed* parameter in file *platformio.ini*
   - In Linux, add yourself to the *dialout*, *tty* and *uucp* groups
3. Download and execute example HelloWorld. When the program starts to run, open the serial monitor, by clicking on the *plug* button available on the bottom of VS Code.

HelloWorld – Simulation in Whisper (Section 8 of GSG)

- Click on *File* → *Open File* and double-click on *[RVfpgaPath]/RVfpga/examples/HelloWorld_C-Lang/platformio.ini*, and set **whisper** as the debug tool by uncommenting line 17.
- Launch the debugger as usual. You can now debug the program exactly as you did in Section 6.B, but this time the program is running in simulation on Whisper instead of on the Nexys A7 FPGA board.
- Given that this program uses the *printfNexys* function in Whisper, you should not open the PlatformIO serial monitor, as messages are shown in the DEBUG console instead.

Lab 1:
C Programming

RVfpga Lab 1: C Programming

- Create **PlatformIO project** from scratch
- Add example **C programs** to the project:
  - **LedsSwitches**
  - **HelloWorld**
- **Run** and **debug** the two programs:
  - On the board
  - On Whisper
- Complete the **exercises** at end of lab

RVfpga Lab 1: Memory-Mapped I/O Addresses

Device	Memory-Mapped I/O Address
Switches (16 on Nexys A7 board)	0x80001400 (upper 16 bits)
LEDs (16 on Nexys A7 board)	0x80001404 (lower 16 bits)
Input/Output of GPIO (1 = output, 0 = input)	0x80001408

RVfpga Lab 1: Example C Program

```
// memory-mapped I/O addresses
#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); }

int main ( void )
{
  int En_Value=0xFFFF, switches_value;      // Upper 16 bits are inputs, lower 16 are outputs

  WRITE_GPIO(GPIO_INOUT, En_Value);

  while (1) {
    switches_value = READ_GPIO(GPIO_SWs);   // read value on switches
    switches_value = switches_value >> 16;  // shift into lower 16 bits
    WRITE_GPIO(GPIO_LEDs, switches_value);  // display switch value on LEDs
  }

  return(0);
}
```

This program writes the value of the switches to the LEDs.

Folder Location: *[RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang*

RVfpga Lab 1: Western Digital's BSP & PSP

- Western Digital provides:
  - **PSP:** platform support package
  - **BSP:** board support package

- These provide common functions for a given processor (SweRV EH1 core) and board (Nexys A7 FPGA board).
  - **Example:** `printfNexys` (like `printf` function in C)

RVfpga Lab 1: Using UART to Print to Terminal

```
#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++;
   }
}
```

- Add this line to **platformio.ini** file:
  monitor_speed = 115200
- After program starts running, **open PlatformIO terminal** by pressing this button in the bottom of the window:

Folder Location: *[RVfpgaPath]\RVfpga\examples\HelloWorld_C-Lang*

RVfpga Lab 1: Exercises Sample – Input/Output

- **Exercise 1**. Write a C program that flashes the value of the switches onto the LEDs. The value should pulse on and off slow enough that a person can view the flashing.
- **Exercise 2**. Write a C program that displays the inverse value of the switches on the LEDs. For example, if the switches are (in binary): 0101010101010101, then the LEDs should display: 1010101010101010; if the switches are: 1111000011110000, then the LEDs should display: 0000111100001111...
- **Exercise 4**. Write a C program that displays the unsigned 4-bit addition of the 4 least significant bits of the switches and the 4 most significant bits of the switches. Display the result on the 4 least significant (right-most) bits of the LEDs. The fifth bit of the LEDs should light up when unsigned overflow occurs (that is when the carry out is 1).

RVfpga Lab 1: Exercises Sample – Algorithms

- **Exercise 5**. Write a C program that finds the greatest common divisor of two numbers, a and b, according to the Euclidean algorithm. The values a and b should be statically defined variables in the program.
- **Exercise 9**. Implement the bubble sort algorithm in C. This algorithm sorts the components of a vector in ascending order by means of the following procedure:
  1. Traverse the vector repeatedly until done.
  2. Interchanging any pair of adjacent components if V(i) > V(i+1).
  3. The algorithm stops when every pair of consecutive components is in order.
- **Exercise 10**. Write a program in C that computes the factorial of a given non-negative number, n, by means of iterative multiplications. While you should test your program for multiple values of n, your final submission should be for n = 7. The program should print out the value of factorial(n) at the end of the program. n should be a variable that is statically defined within the program.

Lab 2:
RISC-V Assembly

RVfpga Lab 2: RISC-V Assembly Programming

- Create **PlatformIO project** from scratch
- Add example **RISC-V Assembly program** to the project:
  - **LedsSwitches**
- **Run** and **debug** the program:
  - On the board
  - On Whisper
- Complete the **exercises** at end of lab
- Create a project from scratch

RVfpga Lab 2: RISC-V Assembly Instructions

Common RISC-V Assembly Instructions & Pseudoinstructions

RISC-V Assembly	Description	Operation
`add  s0, s1, s2`	Add	s0 = s1 + s2
`sub  s0, s1, s2`	Subtract	s0 = s1 - s2
`addi t3, t1, -10`	Add immediate	t3 = t1 – 10
`mul  t0, t2, t3`	32-bit multiply	t0 = t2 * t3
`div  s9, t5, t6`	Division	t9 = t5 / t6
`rem  s4, s1, s2`	Remainder	s4 = s1 % s2
`and  t0, t1, t2`	Bit-wise AND	t0 = t1 & t2
`or   t0, t1, t5`	Bit-wise OR	t0 = t1 | t5
`xor  s3, s4, s5`	Bit-wise XOR	s3 = s4 ^ s5
`andi t1, t2, 0xFFB`	Bit-wise AND immediate	t1 = t2 & 0xFFFFFFFB
`ori  t0, t1, 0x2C`	Bit-wise OR immediate	t0 = t1 | 0x2C
`xori s3, s4, 0xABC`	Bit-wise XOR immediate	s3 = s4 ^ 0xFFFFFABC
`sll  t0, t1, t2`	Shift left logical	t0 = t1 << t2
`srl  t0, t1, t5`	Shift right logical	t0 = t1 >> t5
`sra  s3, s4, s5`	Shift right arithmetic	s3 = s4 >>> s5
`slli t1, t2, 30`	Shift left logical immediate	t1 = t2 << 30
`srli t0, t1, 5`	Shift right logical immediate	t0 = t1 >> 5
`srai s3, s4, 31`	Shift right arithmetic immediate	s3 = s4 >>> 31

RVfpga Lab 2: RISC-V Assembly Instructions

Common RISC-V Assembly Instructions & Pseudoinstructions (continued)

RISC-V Assembly	Description	Operation
`lw s7, 0x2C(t1)`	Load word	s7 = memory[t1+0x2C]
`lh s5, 0x5A(s3)`	Load half-word	s5 = SignExt(memory[s3+0x5A]$_{15:0}$)
`lb s1, -3(t4)`	Load byte	s1 = SignExt(memory[t4-3]$_{7:0}$)
`sw t2, 0x7C(t1)`	Store word	memory[t1+0x7C] = t2
`sh t3, 22(s3)`	Store half-word	memory[s3+22]$_{15:0}$ = t3$_{15:0}$
`sb t4, 5(s4)`	Store byte	memory[s4+5]$_{7:0}$ = t4$_{7:0}$
`beq s1, s2, L1`	Branch if equal	if (s1==s2), PC = L1
`bne t3, t4, Loop`	Branch if not equal	if (s1!=s2), PC = Loop
`blt t4, t5, L3`	Branch if less than	if (t4 < t5), PC = L3
`bge s8, s9, Done`	Branch if not equal	if (s8>=s9), PC = Done
`li s1, 0xABCDEF12`	Load immediate	s1 = 0xABCDEF12
`la s1, A`	Load address	s1 = Variable A's memory address (location)
`nop`	Nop	no operation
`mv s3, s7`	Move	s3 = s7
`not t1, t2`	Not (Invert)	t1 = ~t2
`neg s1, s3`	Negate	s1 = -s3
`j Label`	Jump	PC = Label
`jal L7`	Jump and link	PC = L7; ra = PC + 4
`jr s1`	Jump register	PC = s1

RVfpga Lab 2: RISC-V Registers

32 32-bit registers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

RVfpga Lab 2: Example RISC-V Assembly Program

```
// memory-mapped I/O addresses
# GPIO_SWs    = 0x80001400
# GPIO_LEDs   = 0x80001404
# GPIO_INOUT = 0x80001408

.globl main
```

This program writes the value of the switches to the LEDs.

Folder Location: *[RVfpgaPath]\RVfpga\Labs\Lab02*

```
main:
  li t0, 0x80001400    # base address of GPIO memory-mapped registers
  li t1, 0xFFFF        # set direction of GPIOs
                       # upper half = switches (inputs)   (=0)
                       # lower half = outputs (LEDs)      (=1)
  sw t1, 8(t0)         # GPIO_INOUT = 0xFFFF

repeat:
  lw   t1, 0(t0)       # read switches: t1 = GPIO_SWs
  srli t1, t1, 16      # shift val to the right by 16 bits
  sw   t1, 4(t0)       # write value to LEDs: GPIO_LEDs = t1
  j    repeat          # repeat loop
```

RVfpga Lab 2: Same Exercises as in Lab 1 - Sample

- **Exercise 4**. Write a C program that displays the unsigned 4-bit addition of the 4 least significant bits of the switches and the 4 most significant bits of the switches. Display the result on the 4 least significant (right-most) bits of the LEDs. The fifth bit of the LEDs should light up when unsigned overflow occurs (that is when the carry out is 1).

Lab 3:
Function Calls

RVfpga Lab 3: Function Calls

- Write C programs with **function** (procedure) **calls**
- Write C programs with calls to library functions:
  - Use of standard libraries
  - Use of WD libraries, specific for RVfpga
- RISC-V (Procedure) **Calling Convention**

RVfpga Lab 3: Example Program with Functions

```
// memory-mapped I/O addresses
#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: Example Program with Functions

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

unsigned int getSwitchVal()
{
  unsigned int val;

  val = READ_GPIO(GPIO_SWs);   // read value on switches
  val = val >> 16;  // shift into lower 16 bits

  return val;
}

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

RVfpga Lab 3: C Libraries

- **Libraries**
  - Collection of commonly used functions
  - Provided so that common functions are readily available (save programming time)
- **Example C libraries:**
  - **math.h** (math library): includes functions such as sqrt (square root), cos (cosine), etc.
  - **stdio.h** (standard I/O library): includes functions for printing values to the screen (printf), reading values from users (scanf), etc.
  - **stdlib.h** (standard library): includes functions for generating random numbers (rand).
  - **Many others…** (google C libraries)

RVfpga Lab 3: Example Program using C Library

```
#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);
}
```

This program writes a random number between 0 and 65535 to the LEDs.

RVfpga Lab 3: WD Libraries

- We've used **printfNexys** in many programs
- Lab 9: Use of WD functions for handling **interrupts**

RVfpga Lab 3: RISC-V Calling Convention

- **Call a function**

```
jal function_label
```

- **Return from a function**

```
jr ra
```

- **Arguments**
  - **placed in registers** `a0-a7`
- **Return value**
  - **placed in register** `a0`

RVfpga Lab 3: RISC-V Calling Convention Example

C Code

```
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;
}
```

RISC-V Assembly

```
# y is in s0
main:
   ...
   addi a0, zero, 1   # put values in argument registers
   addi a1, zero, 2
   addi a2, zero, 3
   jal  func1         # call function func1
   add  s0, s0, a0    # y = y + return value
   addi s0, s0, 1     # y = y++
   ...

# sum is in s0
func1:
     add s0, a0, a1   # sum = a + b
     add s0, s0, a2   # sum = a + b + c
     addi a0, s0, 0   # return value = sum
     jr   ra          # return
```

RVfpga Lab 3: The Stack

- **Scratch space** in memory used to save register values
- The stack pointer (`sp`) holds the address of the top of the stack
- The **stack grows downward** in memory. So, for example, to make space for 4 words (16 bytes) on the stack the following code is used:

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

- **Two categories of registers:**
  - **Preserved registers:** register contents must be **preserved** across function calls (i.e., contain the same value before and after a function call)
  - **Non-preserved registers:** register contents must not be **preserved** across function calls (i.e., the register does not need to be the same before and after a function call)
  - Saved registers (`s0-s11`), the return address register (`ra`), and the stack pointer (`sp`) are **preserved** registers. All other registers are not preserved.

RVfpga Lab 3: Preserved / Nonpreserved Registers

Name	Register Number	Use	Preserved
zero	x0	Constant value 0	-
ra	x1	Return address	**Yes**
sp	x2	Stack pointer	**Yes**
gp	x3	Global pointer	-
tp	x4	Thread pointer	-
t0-2	x5-7	Temporary variables	No
s0/fp	x8	Saved variable / Frame pointer	**Yes**
s1	x9	Saved variable	**Yes**
a0-1	x10-11	Function arguments / Return values	No
a2-7	x12-17	Function arguments	No
s2-11	x18-27	Saved variables	**Yes**
t3-6	x28-31	Temporary variables	No

RVfpga Lab 3: The Stack – Revised Assembly Code

C Code

```
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;
}
```

RISC-V Assembly

```
# y is in s0
main: ...
      addi a0, zero, 1   # put values in argument registers
      addi a1, zero, 2
      addi a2, zero, 3
      jal  func1         # call function func1
      add  s0, s0, a0    # y = y + return value
      addi s0, s0, 1     # y = y++
      ...

# sum is in s0
func1:addi sp, sp, -4 # make room on stack
      sw   s0, 0(sp)  # save s0 on stack
      add  s0, a0, a1 # sum = a + b
      add  s0, s0, a2 # sum = a + b + c
      addi a0, s0, 0  # return value = sum
      lw   s0, 0(sp)  # restore s0 from stack
      addi sp, sp, 4  # restore stack pointer
      jr   ra         # return
```

RVfpga Lab 3: Exercises Sample – C programs

- **Exercise 3**. Write a C program that measures reaction time. Your program should time how long it takes for a person to switch on the right-most switch (SW[0]) after all of the LEDs light up. You will use the rand() function from the stdlib.h library to generate a random amount of time to delay between each time the user attempts to test their reaction time.

RVfpga Lab 3: Exercises Sample – Assembly programs

- **Exercise 8**. Write a RISC-V assembly program called Filter.S (the program must be compliant with the standard for function management studied before). You can use the following pseudo-code:

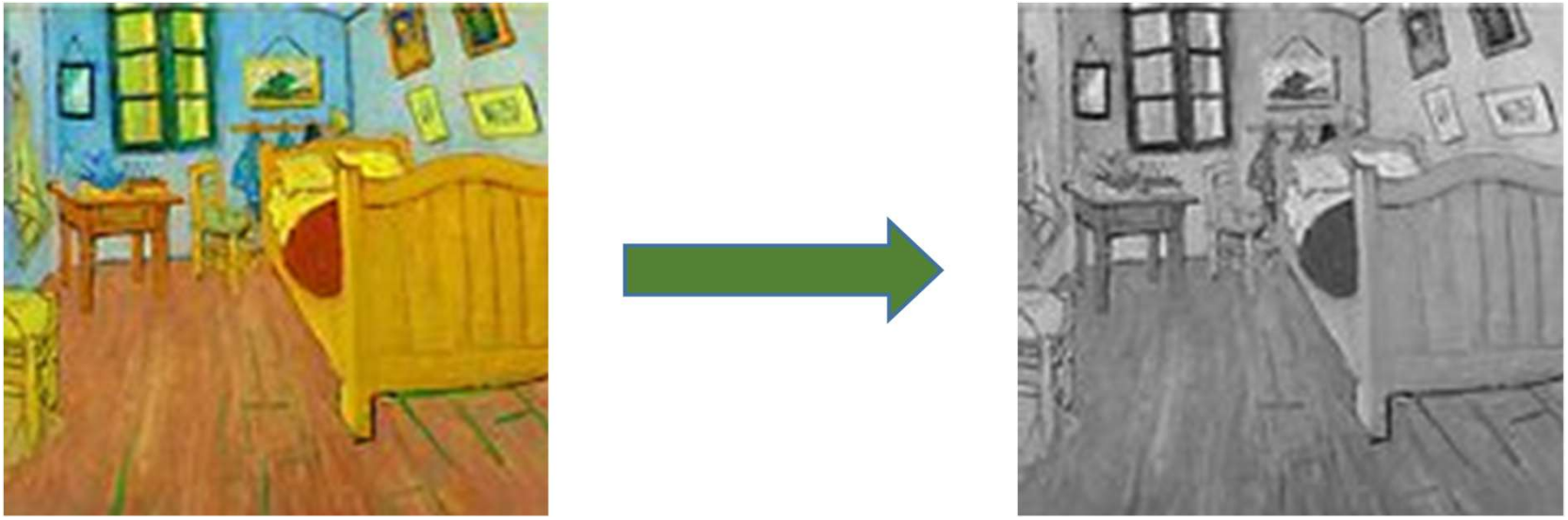
```
#define N 6
int i, j=0, A[N]={48,64,56,80,96,48}, B[N];
for (i=0; i<(N-1); i++){
   if( (myFilter(A[i],A[i+1])) == 1){
      B[j]=A[i]+ A[i+1] + 2;
      j++;
   }
}
```

  - Write the equivalent RISC-V assembly code, including any directives required to reserve memory space, and declaring the corresponding sections (.data, .bss and .text). Function myFilter returns the value 1 if the first argument is a multiple of 16 and the second is greater than the first; otherwise, it returns a 0.
  - Write the assembly code of the function myFilter.

Lab 4:
C and Assembly

RVfpga Lab 4: Combining C and Assembly

- **Example:** Image processing program
- Some functions written in C and some in assembly
- Convert colour image to greyscale



RVfpga Lab 4: Image Processing Program

- Each pixel stored as three 8-bit colours: **R** = red, **G** = green, **B** = blue
- Any colour can be created by **varying R, G,** and **B** values
- To **convert image to an 8-bit greyscale** (`grey`), each pixel is transformed as follows:

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

- RGB weights add up to 1024 (306 + 601 + 117 = 1024), so to get back to an 8-bit range (0-255), the result is divided by 1024 (i.e., shifted right by 10 bits: `>> 10`)
- For more details about the algorithm, see:

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

RVfpga Lab 4: Assembly Function

```
.globl ColourToGrey_Pixel
.text
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
```

.globl makes ColourToGrey_Pixel function visible to all files in project

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

RVfpga Lab 4: Structs and Arrays

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

extern unsigned char VanGogh_128x128[]; // 1D array of individual RGB values
RGB ColourImage[N][M];                  // 2D array of RGB struct (colour image)
unsigned char GreyImage[N][M];          // 2D array of greyscale image

// 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: Main Function

```
int main(void) {
  // Create an N x M matrix using the input image
  initColourImage(ColourImage);

  // Transform Colour Image to Grey Image
  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: Provided project and Exercises

- **Exercise 1**. Execute the program on a different input image.
- **Exercise 2**. Create a C function that counts the number of close to white (>235) and close to black (<20) elements in the VanGogh greyscale image. Print the two numbers on the serial console.
- **Exercise 3**. Transform the **ColourToGrey_Pixel** assembly subroutine into a C function, and the C function **ColourToGrey** into an assembly subroutine that invokes the **ColourToGrey_Pixel** C function.
- **Exercise 4**. Apply a Blur Filter to the VanGogh colour image.

Lab 5:
Vivado Project

RVfpga Lab 5: RVfpga Vivado Project

- **Vivado** is a Xilinx tool for viewing, modifying, and synthesizing the source (Verilog) code for the RVfpga System.
- RVfpga System's source code is in:

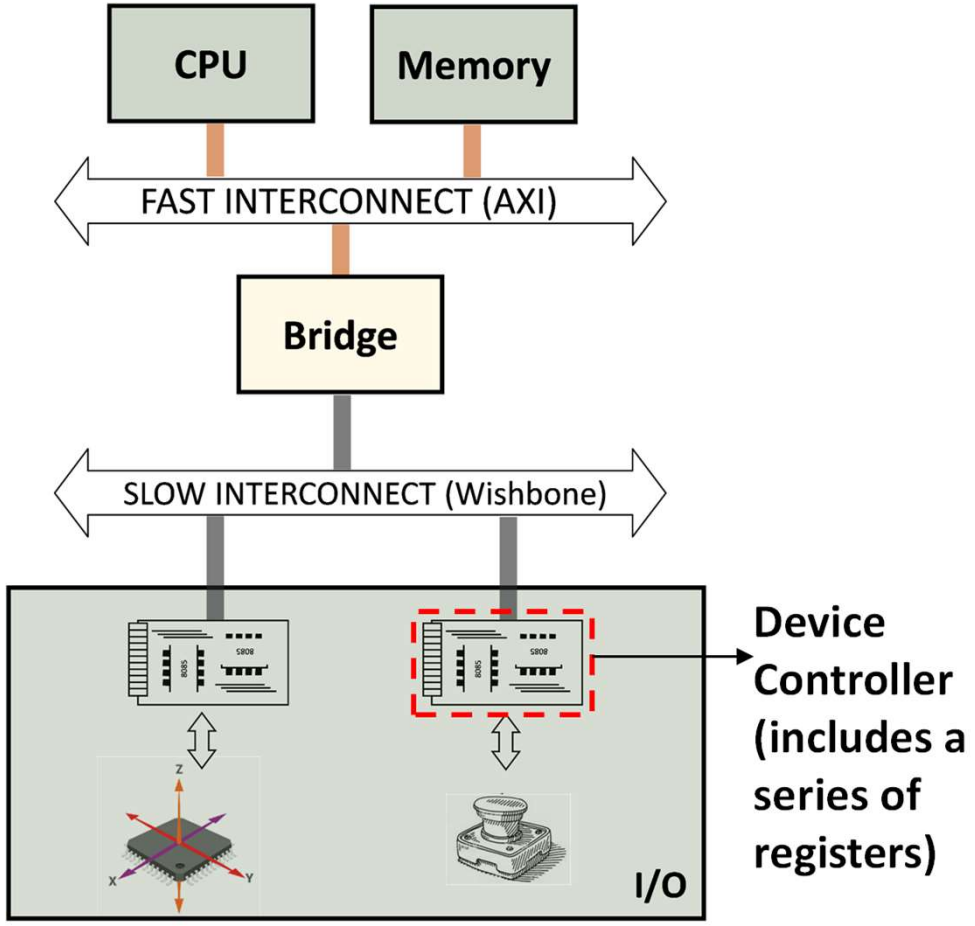
  [RVfpgaPath]/RVfpga/src
- In this lab, users create a **Vivado project** that contains RVfpga System's source code, synthesize RVfpgaNexys targeted to Nexys A7 board and create a **bitfile** that contains information to configure the FPGA as RVfpgaNexys.
- Vivado (and Verilator) are used extensively in RVfpga **Labs 6-20** for modifying and simulating the RVfpga System.

Lab 6:
Intro to I/O

RVfpga Lab 6: Introduction to I/O

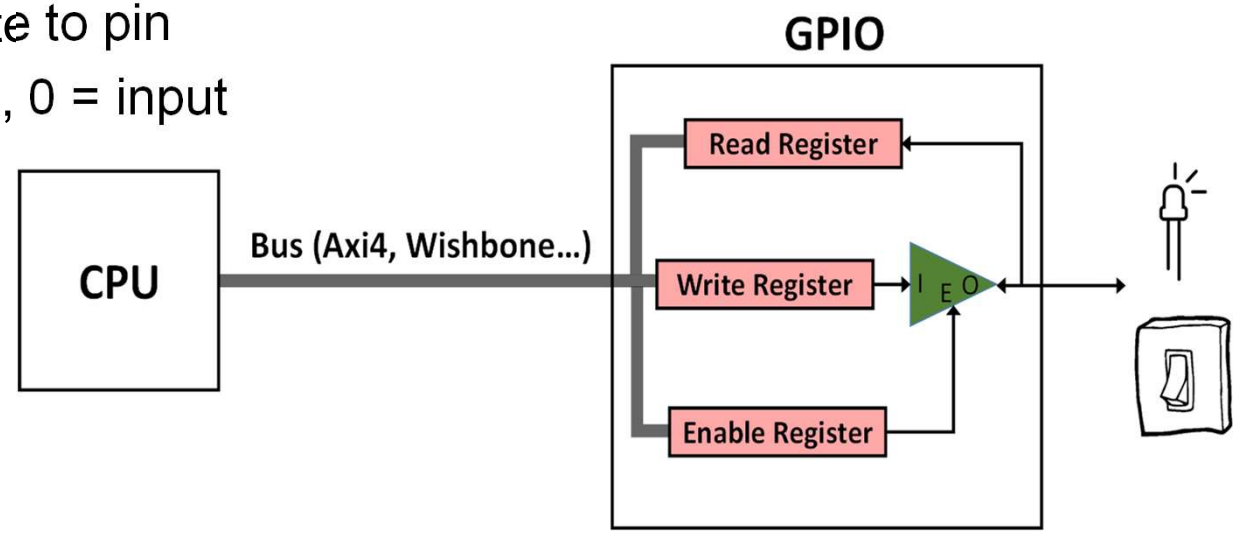
- Main features of a general-purpose I/O system and the one used in the RVfpga System
- Simplified theoretical version of a generic GPIO controller
- GPIO controller used in the SweRVolfX SoC:
  - We first analyse its high-level specification and introduce fundamental exercises
  - We then analyse its low-level implementation, simulating RVfpgaSim in Verilator, and introducing advanced exercises

RVfpga Lab 6: Generic Processor with I/O



RVfpga Lab 6: General-Purpose I/O (GPIO)

- **General-purpose I/O:**
  - Allows processor to read/write pins connected to peripherals (like switches and LEDs)
  - Each pin can be configured as an input or output using tri-state
- **Three memory-mapped registers:**
  - **Read Register:** value read from pin
  - **Write Register:** value to write to pin
  - **Enable Register:** 1 = output, 0 = input



RVfpga Lab 6: SweRVolfX GPIO Module

- **GPIO Module from OpenCores**

  https://opencores.org/projects/gpio

- **Allows up to 32 GPIO pins**
  - All pins can be individually configured as inputs (enable = 0) or outputs (enable = 1)
  - Configuration can change throughout program

Register	Memory-Mapped Address
Read Register	0x80001400
Write Register	0x80001404
Enable Register	0x80001408

RVfpga Lab 6: Memory-Mapped Registers

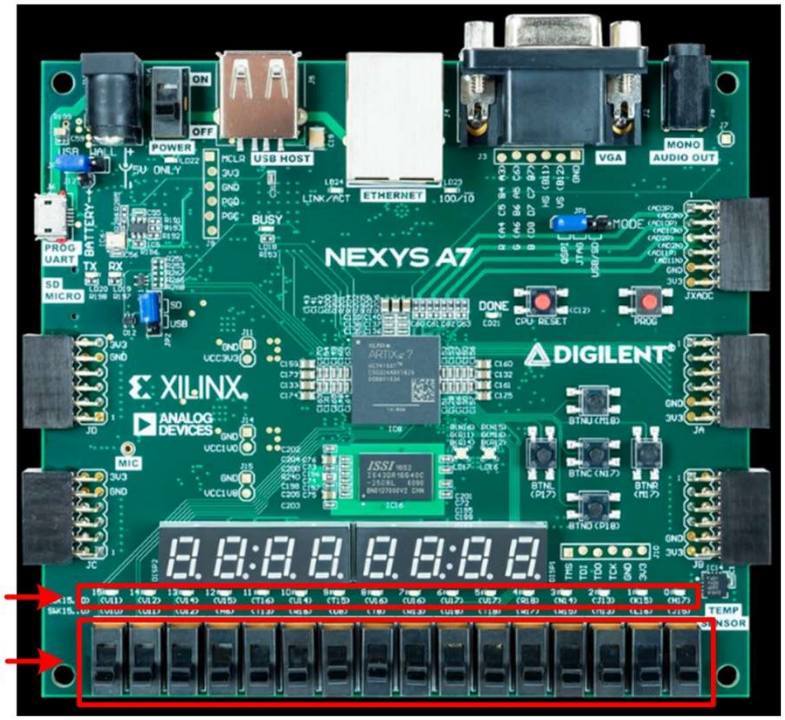


figure of board from https://reference.digilentinc.com/

Mapping LEDs & Switches to GPIO pins:

- **LEDS:** pins **[15:0]** (outputs of processor)
- **Switches:** pins **[31:16]** (inputs to processor)

Configure GPIO:

- Enable Register = 0x0000FFFF (1 = output, 0 = input)

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

Write LEDs:

- Write value in [15:0] to address 0x80001404

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

Read Switches:

- Read switches in bits [31:16] from address 0x80001400
- Shift right by 16 bits to put value in lower 16 bits

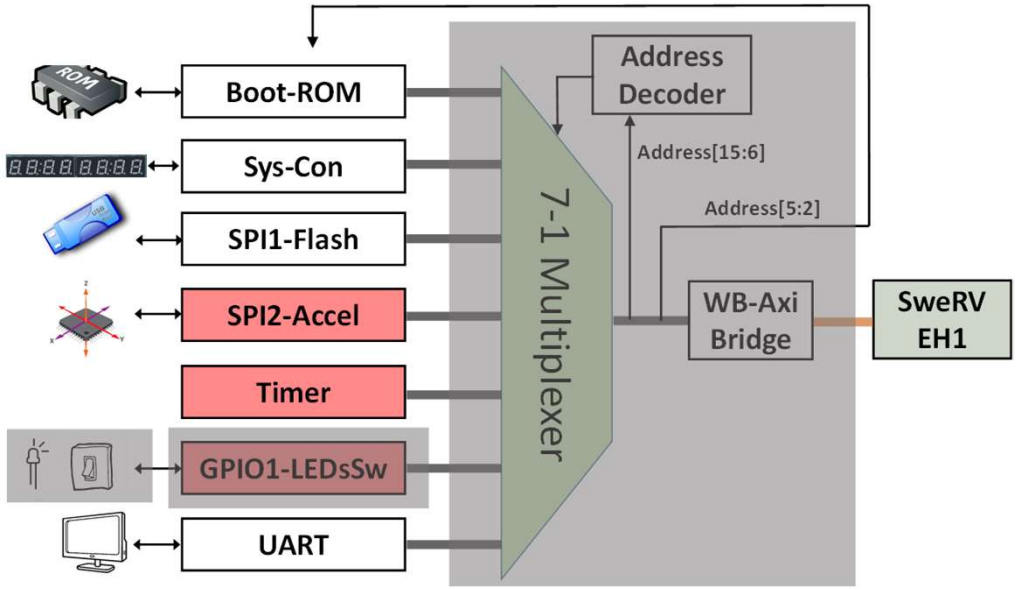
```
lw   t5, 0(t0)  # [t5]31:16 = switch values
srli t5, t5, 16 # [t5]15:0 = switch values
```

RVfpga Lab 6: Fundamental Exercises - Sample

- **Exercise 1.** Write a RISC-V assembly program and a C program that shows a block of four lit LEDs that repeatedly moves from one side of the 16 LEDs available on the board to the other. Also include two switches that control the speed and direction. Switch[0] changes the speed and Switch[1] changes the direction as follows:
  - If Switch[0] is ON (high), the lit LEDs should move quickly. Otherwise, the lit LEDs should move slowly. You may define what “quickly” and “slowly” mean, but either speed must be visible, and you must be able to detect a difference in speed just by looking at it.
  - If Switch[1] is ON (high), the lit LEDs should repeatedly move from right-to-left (they start back at the right when they reach the left-most LED). Otherwise, the lit LEDs should repeatedly move from left-to-right.

RVfpga Lab 6: GPIO Low-Level Implementation

- **Divided in 3 main parts**
  - RVfpgaNexys's external connection to the on-board LEDs/Switches (left shaded region)
  - Integration of the GPIO module into SweRVolfX (middle shaded region)
  - Connection between the GPIO and the SweRV EH1 (right shaded region)



RVfpga Lab 6: External connection

File **rvfpganexys.xdc**: Defines the connection of i_sw[15:0] with the on-board switches and o_led[15:0] with the on-board LEDs

```
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

File **swervolf_core.v**: Tri-state buffers and GPIO module instantiation

```
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

File **wb_intercon.v**: 7-1 Multiplexer implementation

```
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'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
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: Advanced Exercises - Sample

- **Exercise 3.** Expand **RVfpgaNexys** to support the five on-board pushbuttons. The pushbuttons are shown in Figure 22. The five buttons are named according to their location: up, down, left, right, and center – BTNU, BTND, BTNL, BTNR, BTNC.

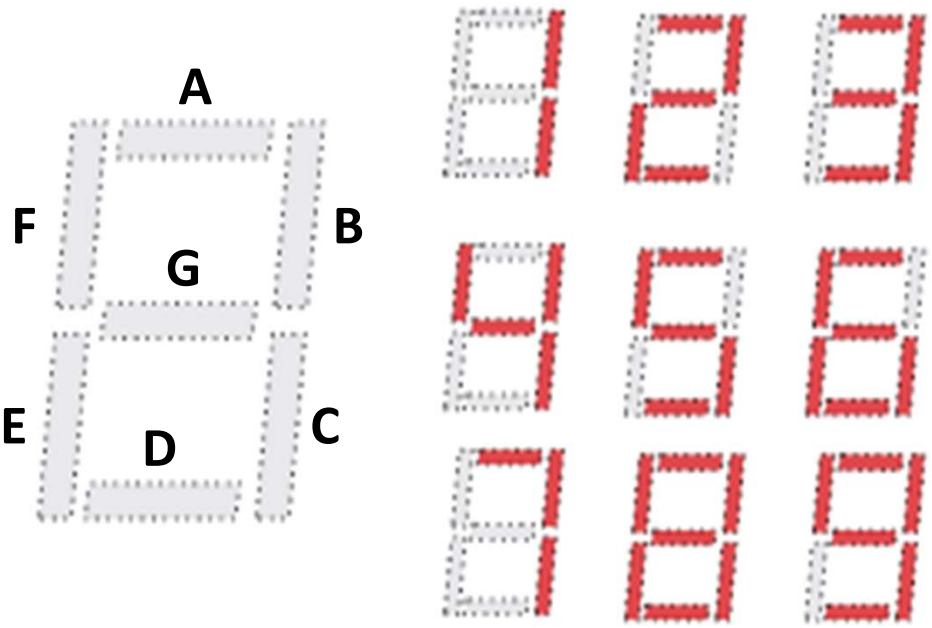
- **Exercise 5.** Write a RISC-V assembly program and a C program that displays an increasingly incrementing binary count on the LEDs, starting at 1. Use BTNC to change the speed of the count, and BTNU to restart the count whenever it is pressed.

Lab 7:
7-Segment Displays

RVfpga Lab 7: 7-Segment Displays

- Describes how the RVfpga System was extended to work with 7-segment displays and shows how to modify the 7-segment display controller.
- We first describe how the 7-segment display controller works
- Then we analyse the high-level specification of the 8-digit 7-segment display controller included in the RVfpga System and provide some fundamental exercises.
- Finally, we analyse the low-level implementation of this controller, perform a Verilator simulation and provide additional exercises where you will modify and experiment with the controller implementation.

RVfpga Lab 7: Overview of 7-Segment Displays



- **7 LED segments:** A-G
- **Light up segments to create digit**
  - **1:** segments B and C
  - **2:** segments A, B, D, E, G
  - **3:** segments A, B, C, D, G
  - etc.

RVfpga Lab 7: 7-Segment Displays on Nexys A7

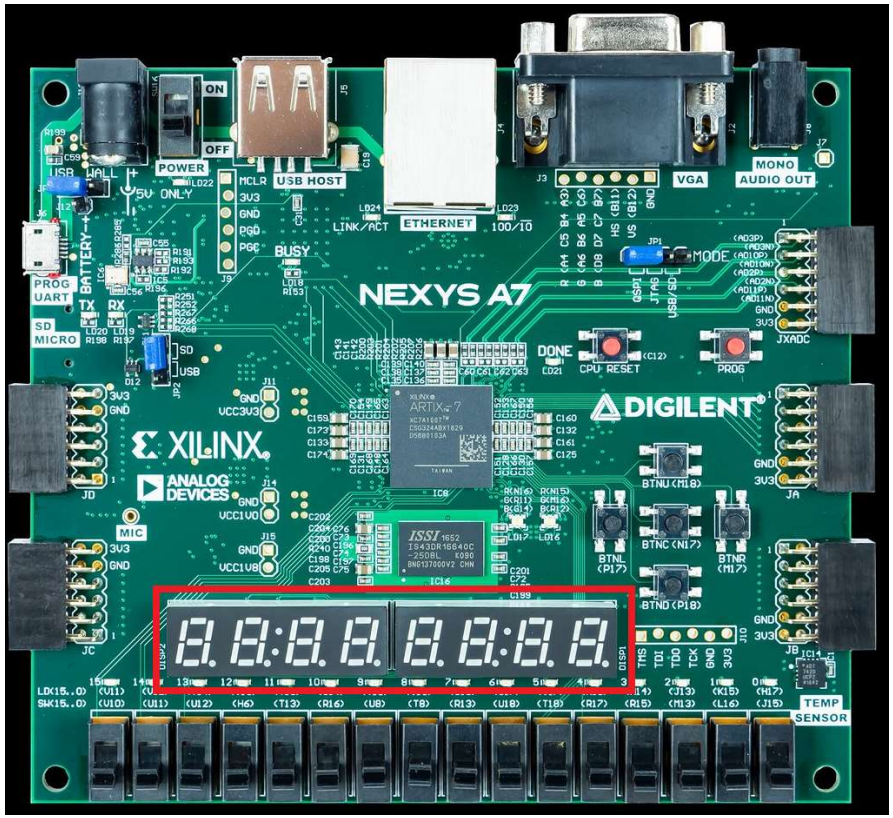


figure of board from https://reference.digilentinc.com/

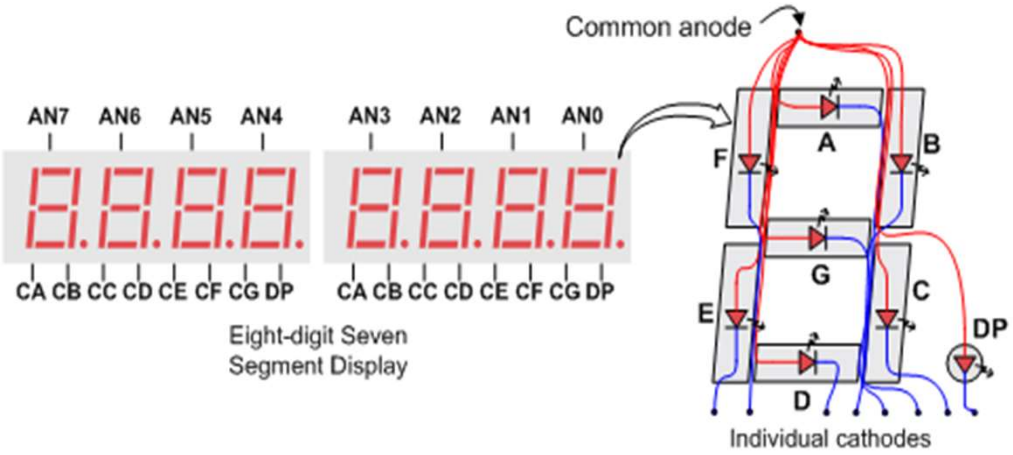
- **8-digit** 7-segment displays
- **Memory-mapped** access:
  - **Enables_Reg:** 0x80001038
  - **Digits_Reg:** 0x8000103C
- **Enables** are **low-asserted**
- **Example:** Display 71 on two right-most digits:
  - **Enables_Reg** = 0xFC (0b11111100: enable two right-most digits)
  - **Digits_Reg** = 0x71
  - **Assembly:**

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

RVfpga Lab 7: 7-Segment Display Hardware

- Each digit is **common anode** (anodes of that digit's LEDs are tied together)
  - Anode signals act as **enables** (AN0 - AN7)
  - Drive **low** to enable digit (AN0 - AN7 go through an inverter (not shown) before being fed to LED)
- Each **segment** for all digits is **tied together**
  - Segments are driven **low** to turn them on
  - **Time-multiplexing** of AN0 - AN7 signals allows unique values to be displayed on each digit
  - A digit's AN signal (AN0 - AN7) must go low every **1-16 ms** to be bright

8-Digit 7-Segment Displays



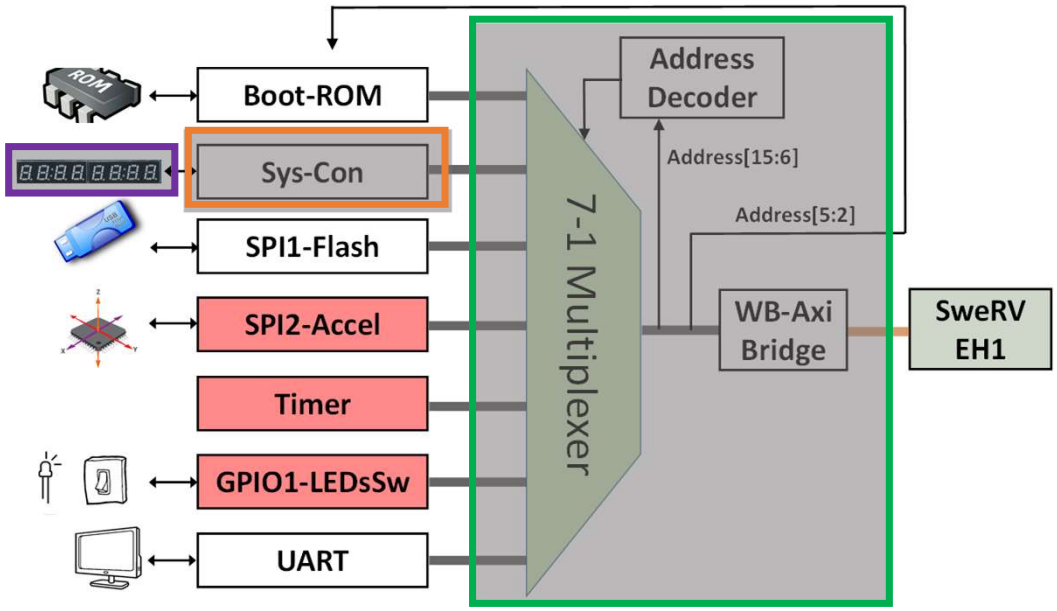
RVfpga Lab 7: Fundamental Exercises - Sample

- **Exercise 1.** Write a RISC-V assembly program and a C program that shows the value of the switches on the four right-most digits of the 7-segment displays.

- **Exercise 2.** Write a RISC-V assembly program and a C program that shows the string “0-1-2-3-4-5-6-7-8” moving from the right to the left of the 8-digit 7-segment displays.

RVfpga Lab 7: 7-Seg. Disp. Low-Level Implementation

- **Three parts:**
  - Connection to **7-segment displays**
  - 7-segment displays decoder in System Control (**Sys-Con**) module
  - **SweRV EH1 bus** interface



RVfpga Lab 7: External connection

File **rvfpganexys.xdc**: Defines the connection of **CA-CG** (called Digits_Bits[i] in the SoC) and **AN[i]** with the on-board 7-segment displays

```
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_FOE_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]
```

RVfpga Lab 7: Integration into SweRVolfX

- File **swervolf_syscon.v**: 7-segment displays controller instance:
  - **Inputs:** i_clk (clock), reset (i_rst)
  - **Memory-mapped inputs:** **Enables_Reg** (which digits on board are enabled), **Digits_Reg** (number to display)
  - **Outputs:** **AN** (which digit on board to drive), **Digits_Bits** (which segments to assert).

```
// 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: Advanced Exercises - Sample

- **Exercise 3.** Modify the controller described in this lab so that the 8-digit 7-segment displays can show any combination of ON/OFF LEDs.

- **Exercise 4.** Use the new controller for printing the following on the 8-digit 7-segment displays: “I SAY HI”. As usual, implement both RISC-V assembly and C versions of the program.

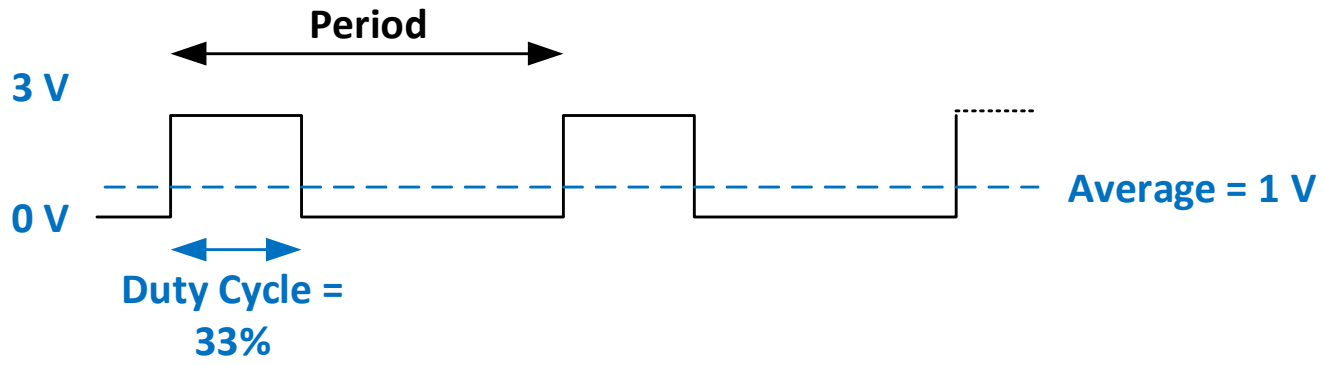
Lab 8:
Timers

RVfpga Lab 8: Timers

- Generate precise timing: Timers increment or decrement a counter at a fixed frequency, which is often configurable, and then interrupt the processor when the counter reaches zero or a predefined value.
- More sophisticated timers can also perform other functions, such as generating pulse-width modulated (PWM) waveforms to control the speed of a motor or the brightness of a light.
- In this lab, we first describe the high-level specification of the timer included in the RVfpga System and then explain its low-level implementation. Both fundamental and advanced exercises are proposed that show how to both use and modify a timer.

RVfpga Lab 8: Timer (PTC) Module

- The Timer module used is from OpenCores: https://opencores.org/projects/ptc
- Timer module (also called the **PTC** module) is used for:
  - **T**imer/**C**ounter: counts clock edges (or edges of another signal, also called events)
  - **P**ulse-width modulation (PWM):
    - Vary high duration (called *duty cycle*) of an output
    - Used to approximate an analog voltage digitally
    - **PWM example:** 33% duty cycle (signal is high 1/3rd of the time). If high level is 3 V, analog voltage (average voltage of signal) is 3 V * 0.33 = 1 V



RVfpga Lab 8: Timer (PTC) Registers

Name	Address	Width	Access	Description
RPTC_CNTR	0x80001200	1-32	R/W	Main PTC counter
RPTC_HRC	0x80001204	1-32	R/W	PTC HI Reference/Capture register
RPTC_LRC	0x80001208	1-32	R/W	PTC LO Reference/Capture register
RPTC_CTRL	0x8000120C	9	R/W	Control register

- **RPTC_CNTR:** Counter (value of the counter)
- **RPTC_HRC:** High reference capture – indicates the number of cycles (after reset) when the output should go high in PWM mode
- **RPTC_LRC:** Low reference capture – indicates the number of cycles (after reset) when the count is complete in counter/timer mode; indicates the number of clock cycles (after reset) when the output should go low in PWM mode.
- **RPTC_CTRL:** Control register

RVfpga Lab 8: Timer Example

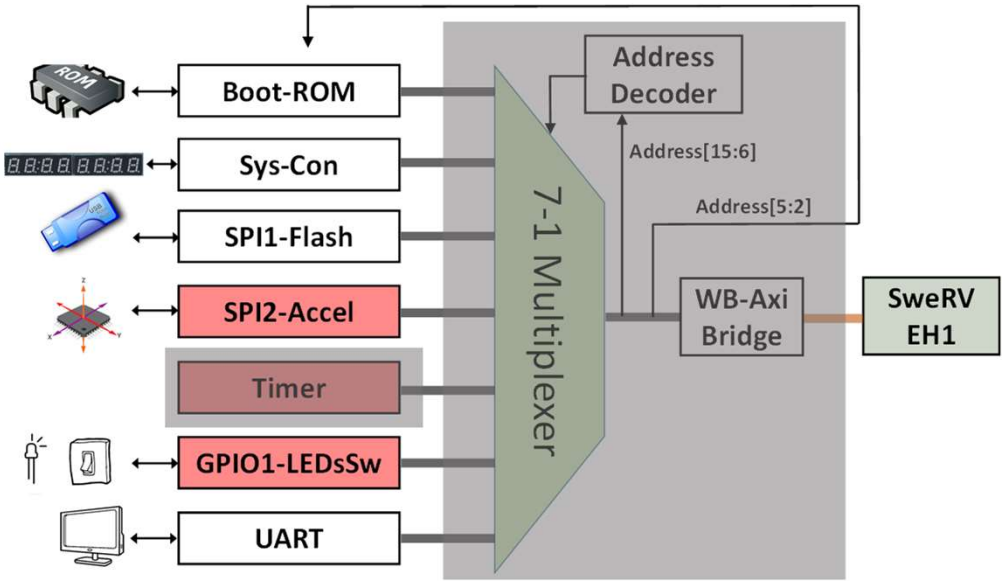
- **Set RPTC_LRC** to number of cycles to count
- **Set control bits (RPTC_CTRL) to configure timer:**
  - **Reset counter and clear interrupts: RPTC_CTRL = 0xC0** (0b**11**000000): CNTRRST (bit 7) = 1: counter is reset (RPTC_CNTR = 0); INT (bit 6) = 1: interrupt request cleared.
  - **Enable counter and interrupts: RPTC_CTRL = 0x21** (0b000**1**0000**1**): EN (bit 0) = 1: counter is enabled, so RPTC_CNTR increments; INTE (bit 5) = 1: PTC asserts an interrupt when RPTC_CNTR == RPTC_LRC.
- Program reads interrupt bit in control register (**INT** is **bit 6** of **RPTC_CTRL**) until it is 1 (indicating that RPTC_CNTR == RPTC_LRC).
- This algorithm **does not use interrupts**, but it does read the interrupt bit (INT, bit 6 of RPTC_CTRL) to determine when the correct number of clock cycles have been reached. We show how to use interrupts in Lab 9.

RVfpga Lab 8: Fundamental Exercises - Sample

- **Exercise 1.** Write a program that displays an ascending count on the 8-digit 7-segment displays. The value should change about once per second and, for creating this delay, you must use the timer module.
  - First, write the program in RISC-V assembly language and run it on the Nexys A7 board.
  - Then, perform a simulation in Verilator with the same program.
  - Now write the program in C and run it on the Nexys A7 board.
  - Simulate your C program in Verilator, as in part (b) for the RISC-V assembly program.

RVfpga Lab 8: Timer Low-Level Implementation

- **Divided in 2 main parts**
  - (No external connection)
  - Integration of the Timer module into SweRVolfX (left shaded region)
  - Connection between the Timer and the SweRV EH1 (right shaded region)



RVfpga Lab 8: Integration into SweRVolfX

File **swervolf_core.v**: PTC module instantiation

```
// 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: Advanced Exercises - Sample

- **Exercise 2.** Modify RVfpgaNexys for connecting the PWM output signal of the timer (*pwm_pad_o*) to one of the two tri-colour LEDs available on the Nexys A7 board.

- **Exercise 3.** Implement a program that uses the new peripheral for controlling the tri-colour LED, using the value provided by the 16 switches.

Lab 9:
Interrupt-Driven I/O

RVfpga Lab 9: Interrupt-Driven I/O

- Interrupt-driven I/O vs. Programmed I/O
- RVfpga System's Interrupt Controller
- How to configure interrupts using Western Digital's Platform Support and Board Support Packages (PSP and BSP)
- Interrupt Example and Exercises

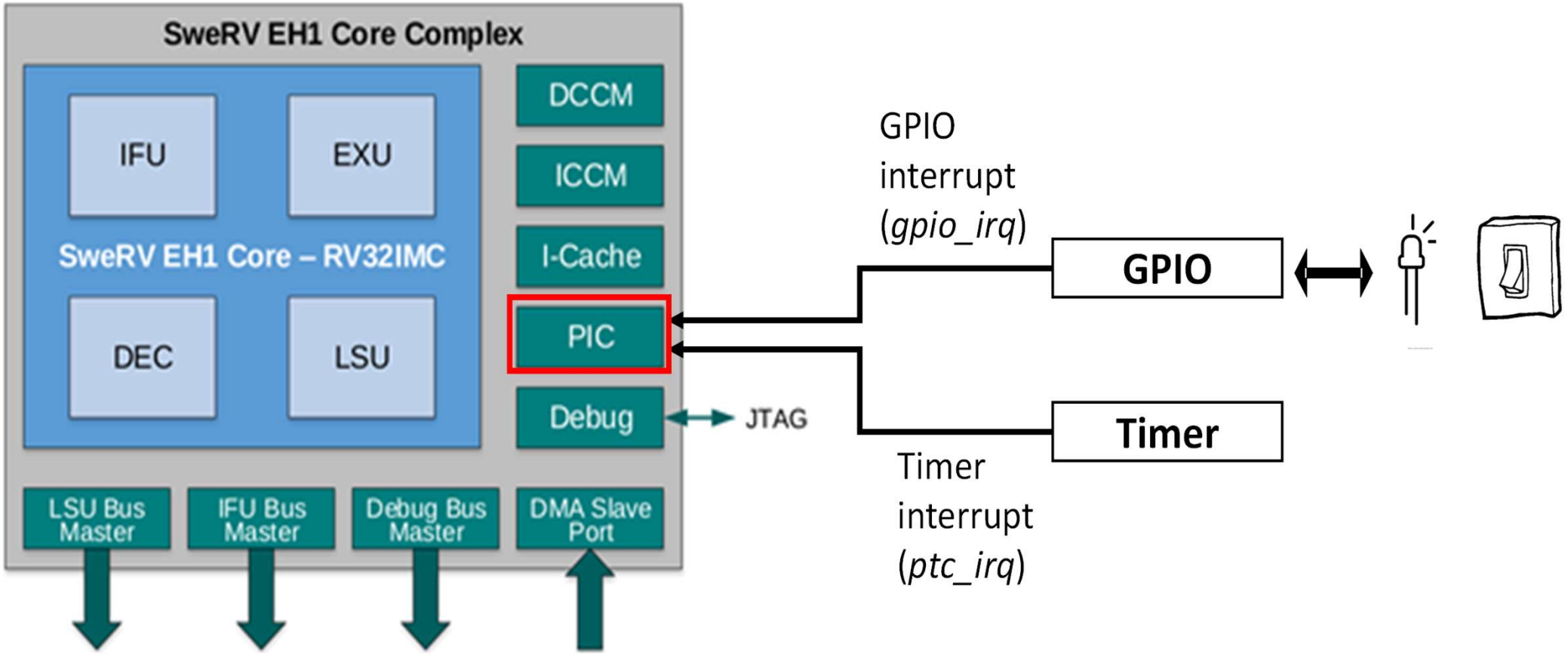
RVfpga Lab 9: Interrupt-Driven I/O Introduction

- **Programmed I/O:**
  - Program continuously **polls** a value (i.e., switches)
  - **Processor busy** doing this – instead of other work
- **Interrupt-driven I/O:**
  - An **event** (i.e., a switch asserting) makes processor jump to an **interrupt service routine** (ISR, also called an interrupt *handler*), which handles the event and then returns to the program.
  - The **processor does other work** between events.

RVfpga Lab 9: Handling Interrupts

- Interrupts may be caused by **hardware** or **software**
- In this lab, we focus on **hardware interrupts**
- The SweRV EH1 core handles interrupts after RISC-V's PLIC (Platform-level interrupt controller) specification. It is referred to as the Programmable Interrupt Controller (**PIC**). It has:
  - 255 interrupt sources
  - 15 priority levels

RVfpga Lab 9: Interrupt Hardware



RVfpga Lab 9: WD's PSP/BSP functions for handling interrupts

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: Interrupt Example using WD's PSP/BSP

Use interrupts to read value of Switch[0] – only on rising edge (0→1 transition)

- Steps for configuring the system using PSP/BSP functions:
  1. Initialize the interrupt system
  2. Initialize external interrupt line IRQ4 and connect with GPIO
  3. Initialize the GPIO registers for using interrupts:
     - RGPIO_INTE = 0x10000 (enable interrupt for Switch[0])
     - RGPIO_PTRIG = 0x10000 (interrupt triggered on rising-edge of Switch[0])
     - RGPIO_INTS = 0x0 (clears all interrupts)
     - RGPIO_CTRL = 0x1 (enables GPIO interrupts)
  4. Enable global interrupts
- GPIO_ISR (see next slide): Invoked when an interrupt is triggered at the GPIO
  1. The current state of the LEDs is read
  2. The LEDs are inverted and masked
  3. The LEDs are written with the new value
  4. The GPIO interrupt is cleared
  5. The IRQ4 external interrupt is cleared
- For full code, see: *[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c*

RVfpga Lab 9: Example ISR to invert right-most LED when switch 0→1

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

  /* Invert LED value */
  i = M_PSP_READ_REGISTER_32(GPIO_LEDs);     // RGPIO_OUT
  i = !i;                                    // Invert the LEDs
  i = i & 0x1;                               // Only change right-most LED

  M_PSP_WRITE_REGISTER_32(GPIO_LEDs, i)      // RGPIO_OUT

  /* Clear GPIO interrupt */
  M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); // RGPIO_INTS

  /* Clear this interrupt (IRQ4) */
  bspClearExtInterrupt(4);
}
```

RVfpga Lab 9: Exercises - Sample

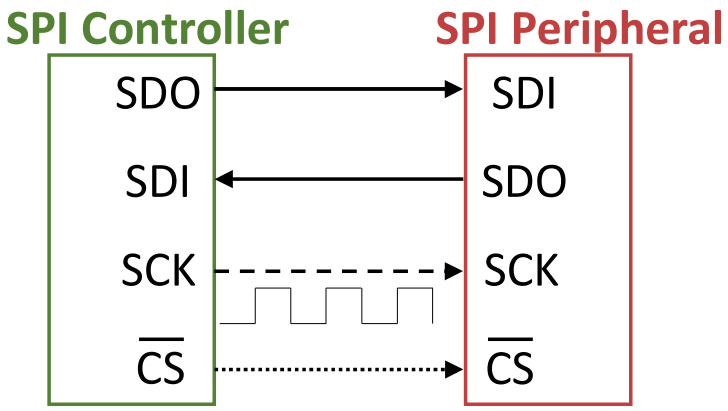
- **Exercise 1.** Modify the *LED-Switch_7SegDispl_Interrupts_C-Lang* program to include a second interrupt source, in this case generated by the timer.
- **Exercise 2.** Modify RVfpgaNexys to include a third interrupt source coming from the second GPIO that you designed in Lab 6 for controlling the on-board pushbuttons (GPIO2).
- **Exercise 3.** Use the extended RVfpgaNexys version that you designed in the previous exercise to implement a C program that displays an increasingly incrementing binary count on the LEDs, starting at 1.
  - Create a delay with the timer, using interrupts, for waiting between displaying each incremented value so that the values are viewable by the human eye.
  - Read BTNC and use it to change the speed of the count.
  - Read Switch[0] and use it to restart the count whenever it is pressed.

Lab 10:
Serial Buses

RVfpga Lab 10: Serial Buses

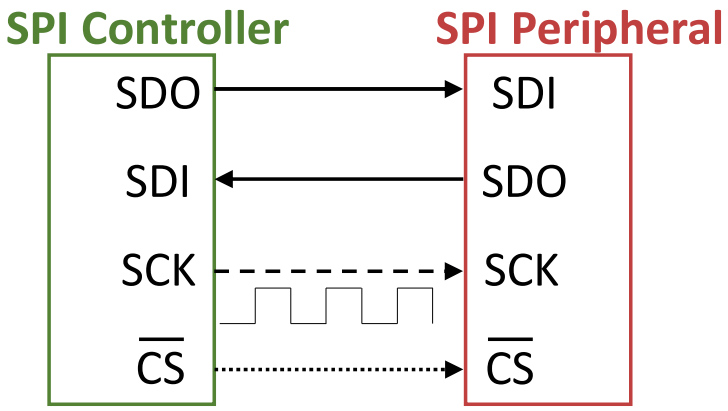
- We first describe how serial buses work.
  - Serial buses send **one bit at a time**.
  - Parallel buses send **multiple bits at once**
- Common serial buses
  - **UART** (universal asynchronous receiver/transmitter)
  - **SPI** (serial peripheral interface)
  - **I2C** (inter-integrated circuit protocol)
- We then focus on the SPI accelerometer available on the Nexys A7 board:
  - Analysis of the high-level specification + fundamental exercises
  - Analysis of the low-level implementation + advanced exercises
- More advanced exercises with UART and I2C

RVfpga Lab 10: Serial Buses – SPI

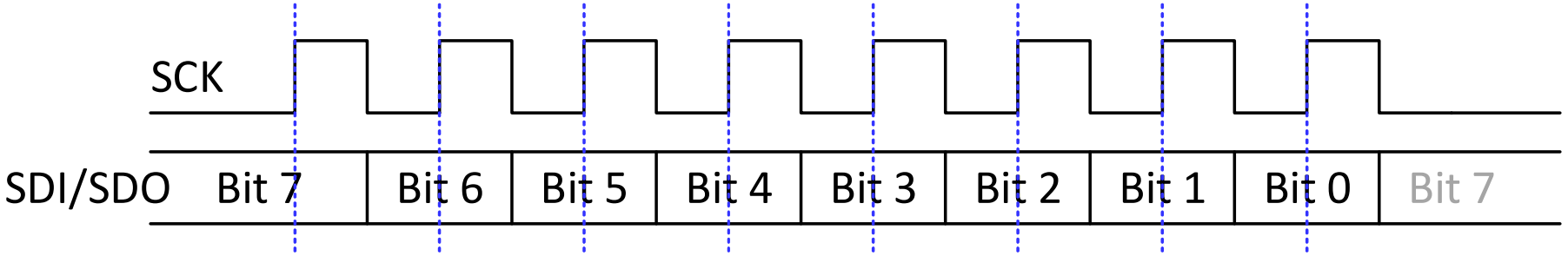


- **Controller:** sends clock, sends & receives data
- **Peripheral:** receives clock, sends & receives data
- **Signals:**
  - **SDO:** Serial Data Out
  - **SDI:** Serial Data In
  - **SCK:** SPI clock
  - **CSbar:** low-asserted chip select

RVfpga Lab 10: Serial Buses – SPI



- **SCK idles**
- When controller sends an **edge on SCK**, both the controller and peripheral **sample and send data**. Data is changed (sent) on falling edge and sampled on rising edge (although this is configurable)



RVfpga Lab 10: RVfpga System's SPI Module

- RVfpga System's SPI module is from OpenCores

  https://opencores.org/projects/simple_spi

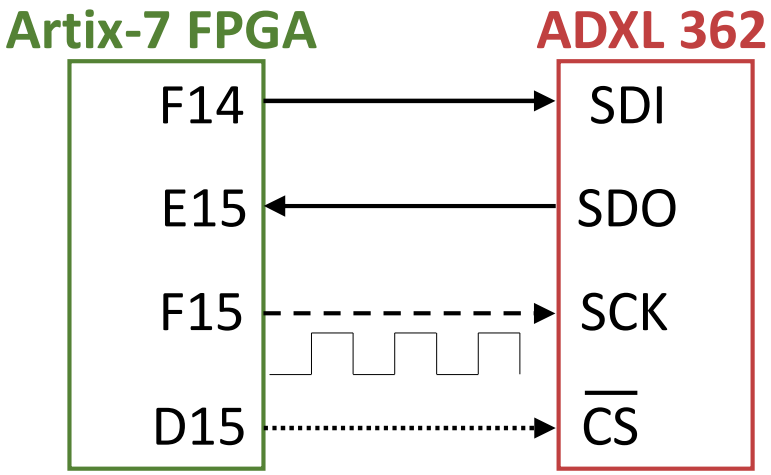
- 4-entry read and write buffers
- SPI Registers:

Name	Address	Width	Access	Description
SPCR	0x80001100	8	R/W	Control register
SPSR	0x80001108	8	R/W	Status register
SPDR	0x80001110	8	R/W	Data register
SPER	0x80001118	8	R/W	Extensions register
SPCS	0x80001120	8	R/W	CS register

RVfpga Lab 10: SPI ADXL362 Accelerometer

- The Nexys A7 board includes an SPI Analog Devices ADXL362 accelerometer. You can find the complete information at:

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

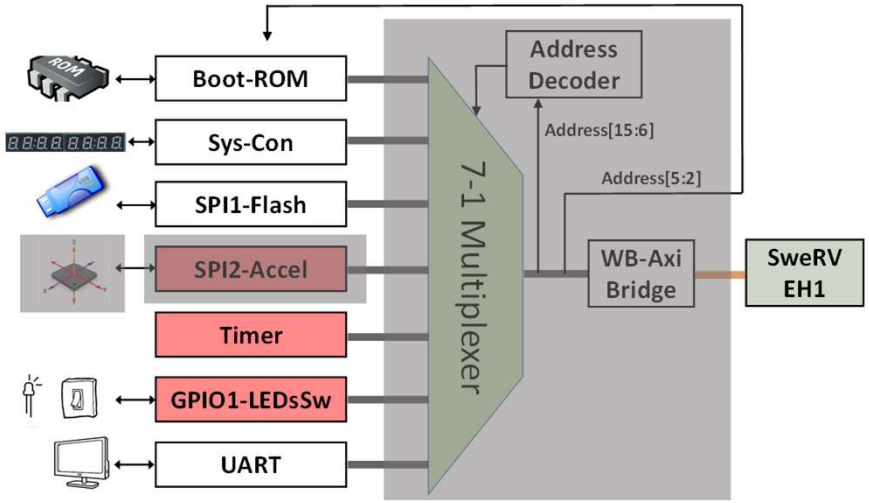


RVfpga Lab 10: Fundamental Exercises - Sample

- **Exercise 1.** Create a RISC-V assembly program that reads the eight most significant bits of the X-axis, Y-axis, and Z-axis acceleration data and then displays those values on the 8-digit 7-Segment Displays.

RVfpga Lab 10: Accel. Low-Level Implementation

- **Divided in 3 main parts**
  - RVfpgaNexys external connection to the on-board accelerometer (left shaded region)
  - Integration of the new SPI module into SweRVolfX (middle shaded region)
  - Connection between the accelerometer and the SweRV EH1 (right shaded region)



RVfpga Lab 10: External Connection

File **rvfpganexys.xdc**: Defines the connection of the SPI signals used in the SoC with the corresponding on-board accelerometer pins

```
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: Integration into SweRVolfX

File **swervolf_core.v**: Tri-state buffers and GPIO module instantiation

```
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: Advanced Exercises - Sample

- **Exercise 2.** The Universal Asynchronous Receiver-Transmitter (UART) is an asynchronous serial communication protocol. First, analyse the low-level implementation of this module in Rvfpga. Then, create a RISC-V assembly program that prints a message to the PlatformIO shell through the serial port.

- **Exercise 3.** Implement the three following functions in the C language:
  - char uart_getchar(void): This function waits for the keyboard to send a character through the UART to the Nexys A7 board and then returns this character as an output parameter.
  - int uart_putchar(char c): This function receives a character as an input argument and displays it on the serial console through the UART.
  - int SevSegDispl(char c): This function receives a character as an input argument and displays it on the right-most digit of the 7-segment displays, shifting the remaining digits one position to the left (the left-most digit is lost).

Labs 11-20
Understanding the Core and Memory Systems

RVfpga Labs 11-20 Overview

- Labs 11–20 dive down to the microarchitectural level and analyse how the SweRV EH1 processor and cache/memory hierarchy work.
- Each lab is divided into two parts:
  - Theoretical explanation of the concepts
  - Illustration of the concepts using figures and a Verilator simulation of an example program to illustrate the concept.

  We also provide exercises to deepen understanding of and gain experience with the described concepts.

RVfpga SweRVref

- In addition to these 10 labs, which we describe next, the RVfpga_SweRVref document provides extra instructions on the following topics:
  - Section 1: **Sigasi Studio**
  - Section 2: **Configuration** of the **SweRV EH1** processor
  - Section 3: **RVfpga System hierarchy of modules** and their most relevant **signals**
  - Section 4: **Main structures/types** for grouping **control bits**
  - Section 5: **RISC-V compressed instructions**
  - Section 6: **Real Benchmarks**

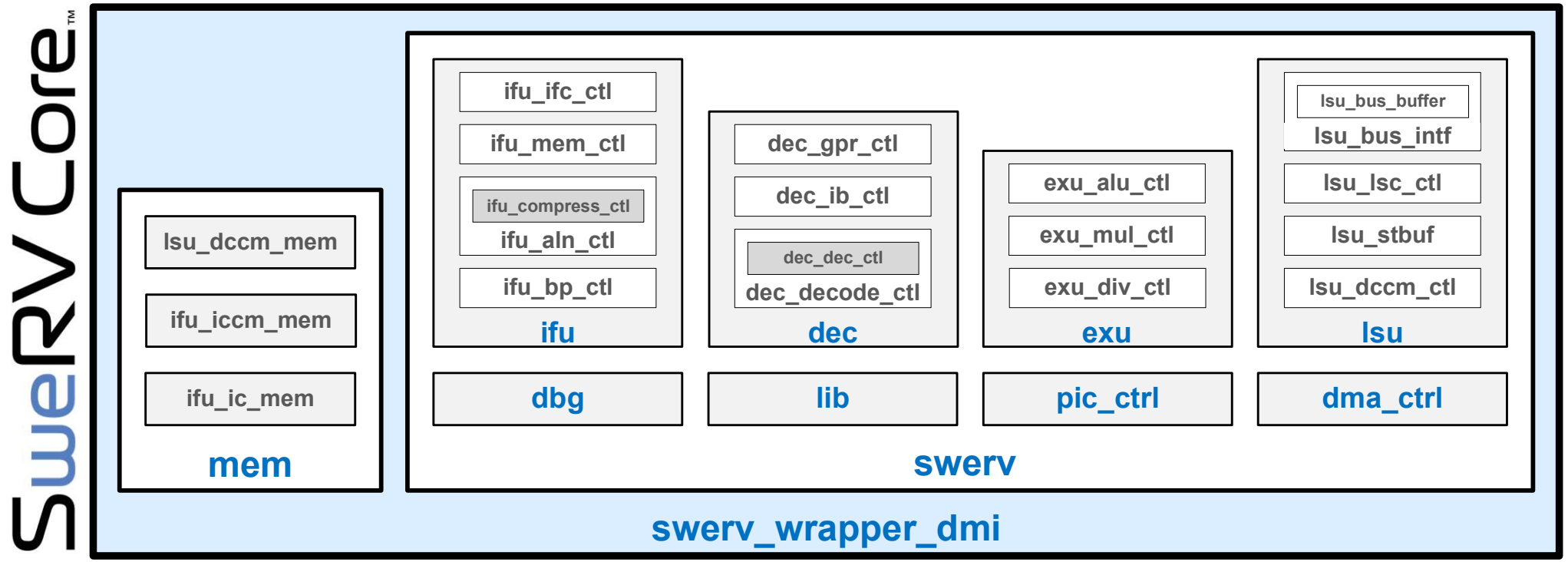
Lab 11:
SweRV EH1 Configuration, Organization, and Performance Monitoring

RVfpga Lab 11: The SweRV EH1 processor

- In this lab we begin to analyse the SweRV EH1 processor. Specifically:
  - We first describe the Verilog RTL organization and details of each pipeline stage.
  - We then show how to use the SweRV EH1 performance counters to analyse processor performance.

RVfpga Lab 11: SweRV EH1 Verilog Modules

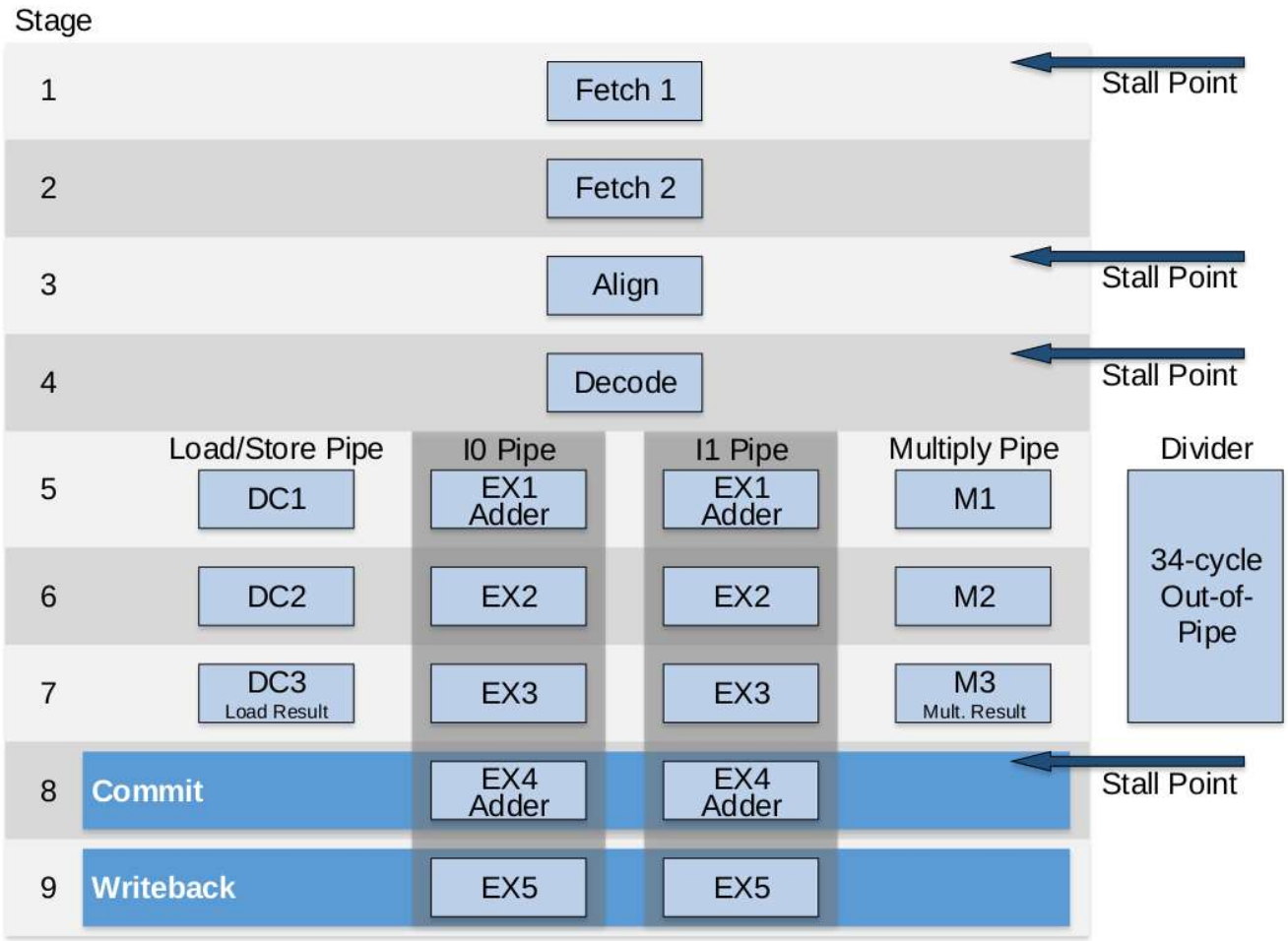
- **swerv** module: CPU
- **mem** module (memory hierarchy): instruction/data closely-coupled memories (ICCM, DCCM) and instruction cache (I$)



RVfpga Lab 11: The SweRV EH1 Pipeline

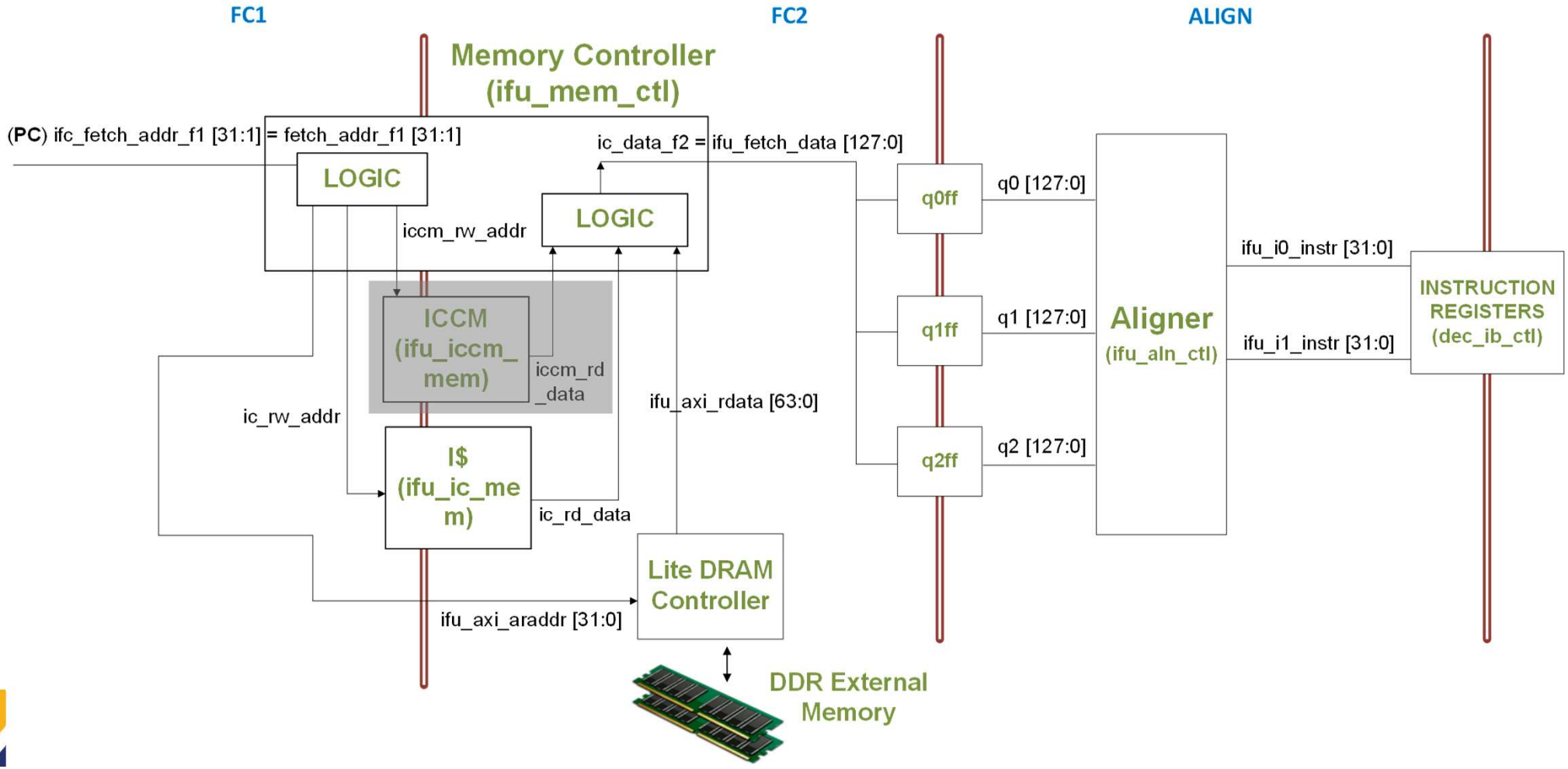
SweRV Core™

SweRV EH1 is a 32-bit 2-way superscalar 9-stage pipelined in-order processor.



RVfpga Lab 11: Fetch (FC1 and FC2) and Align Stages

- First three stages: two Fetch stages (FC1 and FC2) and an Align stage

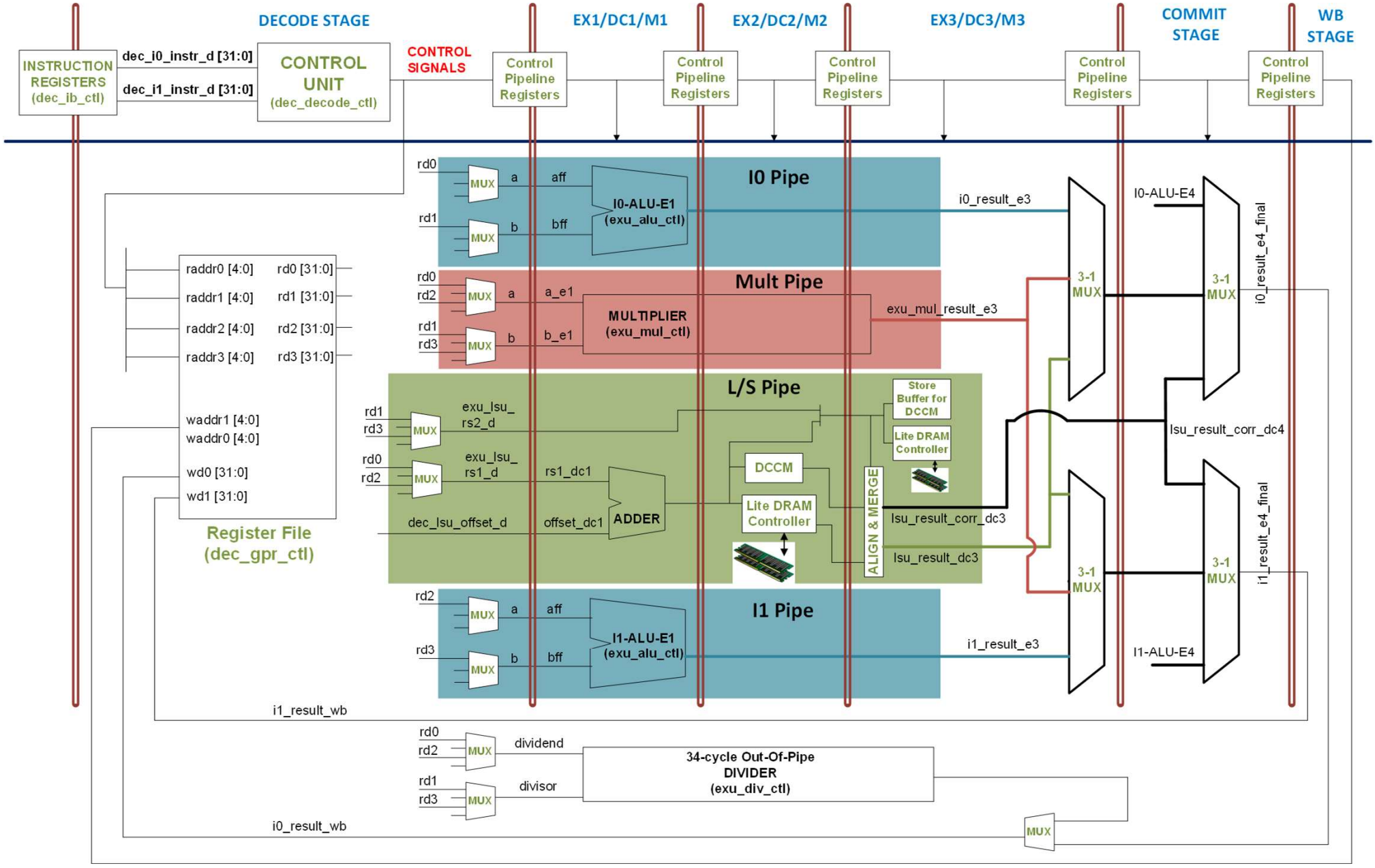


RVfpga Lab 11: Fetch (FC1 and FC2) and Align Stages

- In RVfpga, the Instruction Memory consists of:
  - 16 KiB Instruction Cache
  - 128 MiB DDR External Memory
- **Fetch** stages: read instructions from the Instruction Memory
  - **FC1**: Computes the instruction address (`ifc_fetch_addr_f1`)
  - **FC2**: Reads instruction from the I$ or the DDR External Memory. (The I$ only caches memory within the Main Memory address range.)
- The **Align** stage performs two main tasks:
  - **Provide two 32-bit instructions per cycle to the Decode stage**: Extracts two instructions per cycle from the 128-bit bundles provided by the Instruction Memory and assigns them to each of the two ways available in SweRV EH1.
  - **Uncompress instructions**: The Align stage uncompresses 16-bit instructions into 32-bit instructions.

RVfpga Lab 11: Decode, EX1/2/3, Commit and WB Stages

- The figure on the next slide shows the last six stages of the pipeline: the **Decode** stage, three **Execution** stages, the **Commit** stage, and the **Writeback** (WB) stage.


DECODE STAGE
EX1/DC1/M1
EX2/DC2/M2
EX3/DC3/M3
COMMIT STAGE
WB STAGE
INSTRUCTION REGISTERS (dec_ib_ctl)
dec_i0_instr_d [31:0]
dec_i1_instr_d [31:0]
CONTROL UNIT (dec_decode_ctl)
CONTROL SIGNALS
Control Pipeline Registers
I0 Pipe
I0-ALU-E1 (exu_alu_ctl)
i0_result_e3
I0-ALU-E4
i0_result_e4_final
Mult Pipe
MULTIPLIER (exu_mul_ctl)
exu_mul_result_e3
L/S Pipe
Store Buffer for DCCM
Lite DRAM Controller
DCCM
ADDER
ALIGN & MERGE
lsu_result_corr_dc4
lsu_result_corr_dc3
lsu_result_dc3
I1 Pipe
I1-ALU-E1 (exu_alu_ctl)
i1_result_e3
I1-ALU-E4
i1_result_e4_final
3-1 MUX
Register File (dec_gpr_ctl)
raddr0 [4:0]
raddr1 [4:0]
raddr2 [4:0]
raddr3 [4:0]
rd0 [31:0]
rd1 [31:0]
rd2 [31:0]
rd3 [31:0]
waddr1 [4:0]
waddr0 [4:0]
wd0 [31:0]
wd1 [31:0]
i1_result_wb
i0_result_wb
34-cycle Out-Of-Pipe DIVIDER (exu_div_ctl)
dividend
divisor

RVfpga Lab 11: Decode Stage

- The Decode stage performs two main tasks:
  - **Decode the instructions and generate the control signals** (performed by the Control Unit)
  - **Distribute the instructions and operands to the appropriate pipes**:
    - Pipes:
      - Two Integer pipes: I0 and I1
      - Multiply pipe
      - Load/Store pipe (L/S)
      - Out-of-pipe 34-cycle Divider
    - Several multiplexers select among possible operands, which may come from:
      - Bypass logic
      - Immediate
      - Register File

RVfpga Lab 11: Execution stages – 3 Pipes and a Divider

The SweRV EH1 processor has the following pipes:

- **I0/I1 Pipes**:
  - Two integer pipes which have three stages (EX1, EX2, and EX3).
  - EX1 performs the ALU operation.
- **Multiply Pipe**: The multiply pipe contains a 3-cycle integer multiplier using three stages (M1, M2, and M3).
- **Load/Store (L/S) Pipe**: The L/S pipe executes both load and store instructions.
  - DC1: an adder calculates the address by adding the register base address and the offset
- **Divider**: The divider is a non-pipelined 34-cycle integer divider.

At the end of the third execution stage (EX3/DC3/M3), the result of the instructions is selected from the proper pipe (I0/I1, MUL, or L/S) using two 3:1 multiplexers, one for each way. The Divider has its own path to the Register File.

RVfpga Lab 11: Commit and Writeback Stages

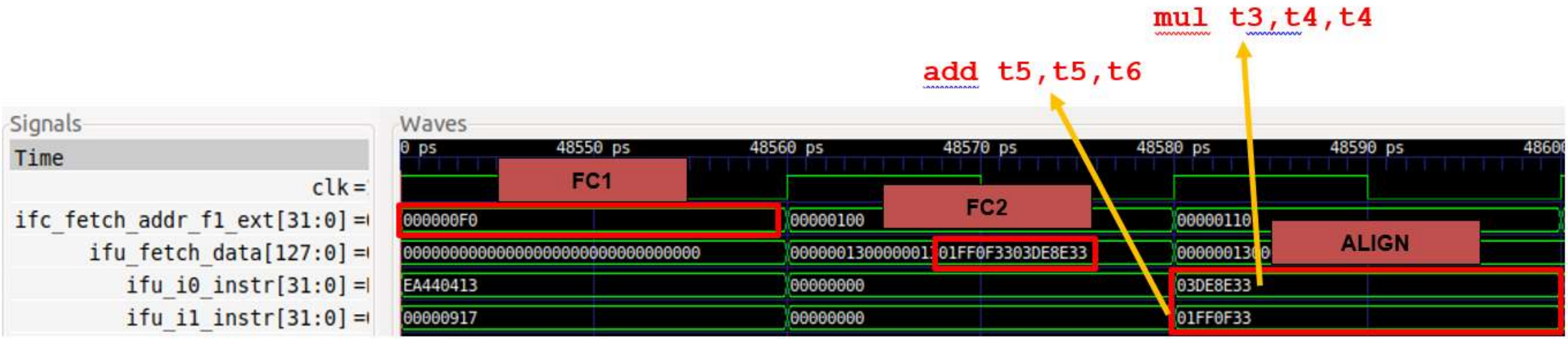
- **Commit Stage:** Selects the result to write back to the register file.

- **Writeback Stage:**
  - Writes the results to the Register File using write ports 0 and 1.
  - The Control Pipeline Registers supply the register identifiers and the enable signals (which were generated in the Decode stage).

RVfpga Lab 11: Example Program – Verilator Simulation

```
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 # Repeat the loop
```

RVfpga Lab 11: Simulation – FC1, FC2, Align Stages

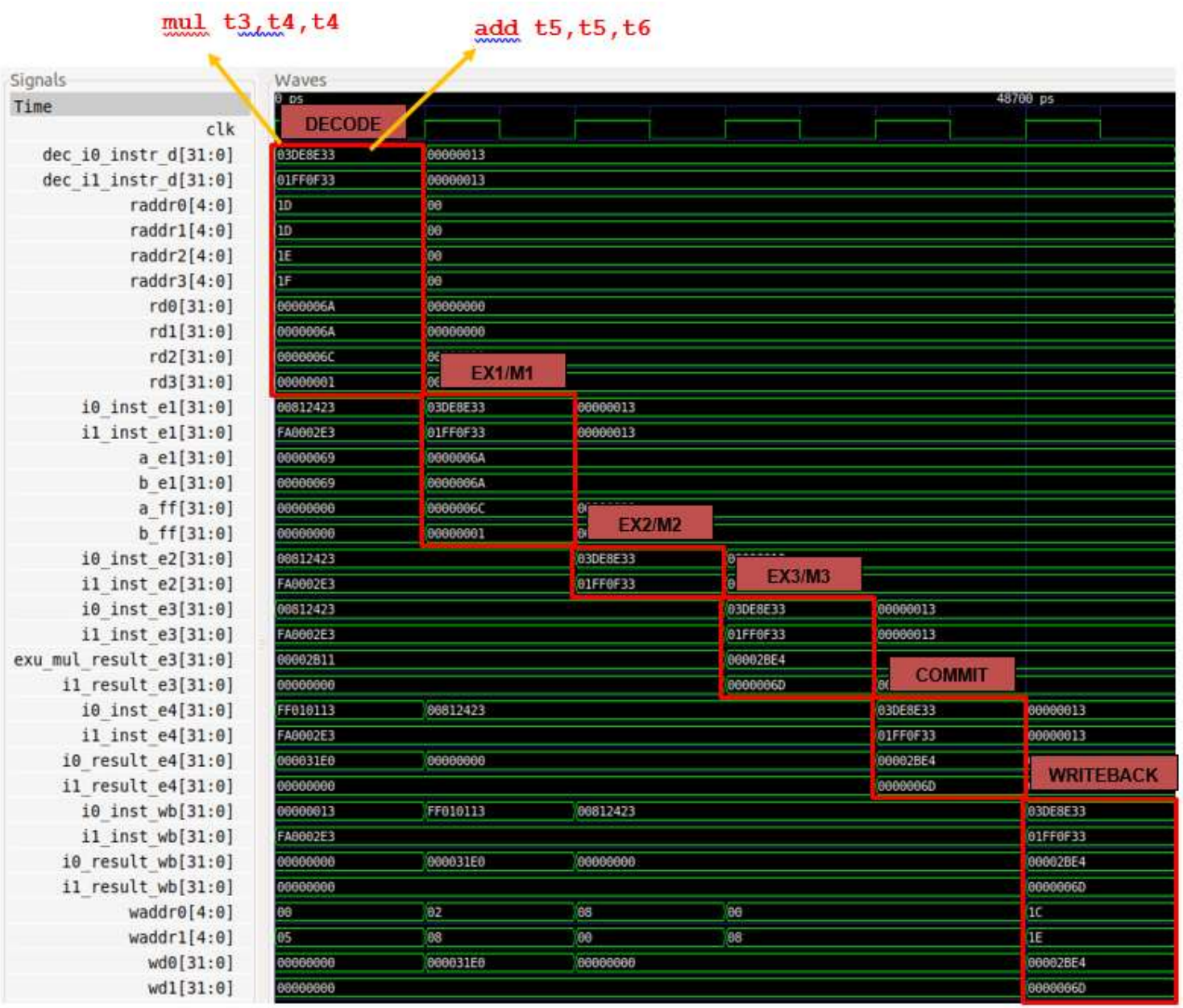


RVfpga Lab 11: Analysis of Simulation

- **FC1**: Computes the address of the `mul` instruction:
  - `ifc_fetch_addr_f1_ext` = 0x000000F0
- **FC2**: Extracts two instructions (shown in red) from the Instruction Memory's 128-bit bundle:
  - `ifu_fetch_data` = 0x00000013000000131**01FF0F3**303DE8E33
- **Align**: The two instructions are extracted and distributed to the two ways of SweRV EH1.
  - Way 0: `ifu_i0_instr` = 0x03DE8E33 (`mul` instruction)
  - Way 1: `ifu_i1_instr` = 0x01FF0F33 (`add` instruction)

Simulation:

- Decode
- EX1/2/3
- Commit
- Writeback



RVfpga Lab 11: Analysis of Simulation

- **Decode**: read operands from Register File and send to Mult and I1 pipes
- **EX1/2/3 and Commit**: Compute result (addition and multiplication)
  - `i0_result_e4` = `exu_mul_result_e3` = 0x6A * 0x6A = 0x2BE4
  - `i1_result_e4` = `i1_result_e3` = 0x6C + 0x01 = 0x6D
- **Writeback**: Write results back to Register File
  - `waddr0` = 0x1C `wd0` = 0x2BE4
  - `waddr1` = 0x1E `wd1` = 0x6D

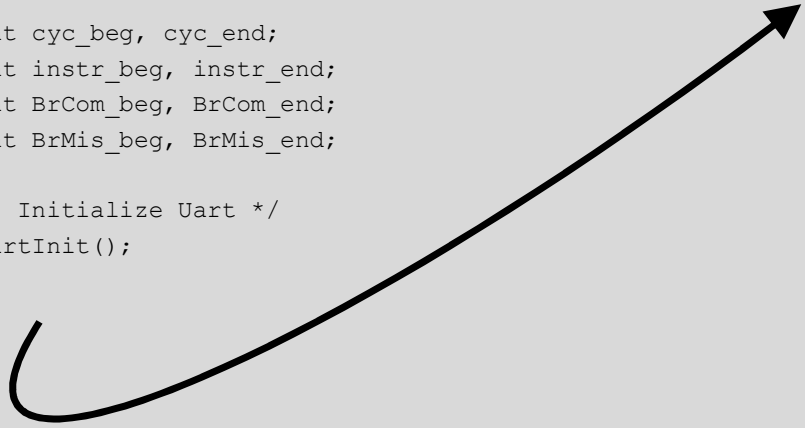
RVfpga Lab 11: Hardware Counters

- Hardware counters are a set of special-purpose registers included in most current processors to record the metrics shown in the table.

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 commmited	21	Fence	38	Timer interrupts taken
5	Instrs commited 16-b	22	Fence.i	39	Exteranal interrupts taken
6	Instrs commited 32-b	23	Mret	40	TLU flushes
7	Instrs aligned	24	Branches commited	41	Branch error flushes
8	Instrs decoded	25	Branches mispredicted	42	I-bus transactions – instr
9	Muls commited	26	Branches taken	43	D-bus transactions – ld/st
10	Divs commited	27	Unpredictable branches	44	D-bus transactions misaligned
11	Loads commited	28	Cycles fetch stalled	45	I-bus errors
12	Stores commited	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 commited	32	Cycles presync stalled	49	Cycles interrutps disabled
16	CSR read	33	Cycles frozen	50	Cycles interrupts stalled while disabled

RVfpga Lab 11: Use of the Performance Counters by means of Western Digital's PSP

```
#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: Tasks - Sample

- **TASK.** The Register File is implemented in module `dec_gpr_ctl` and it is instantiated in module `dec`. Analyse both the Verilog code and the simulation of the main signals of module `dec_gpr_ctl` in order to understand how it works.

- **TASK.** Execute the program from Figure 13 on the Nexys A7 board as explained in the GSG. Measure other events in the Hardware Counters for this program.

Lab 12: Arithmetic/Logic Instructions: The add Instruction

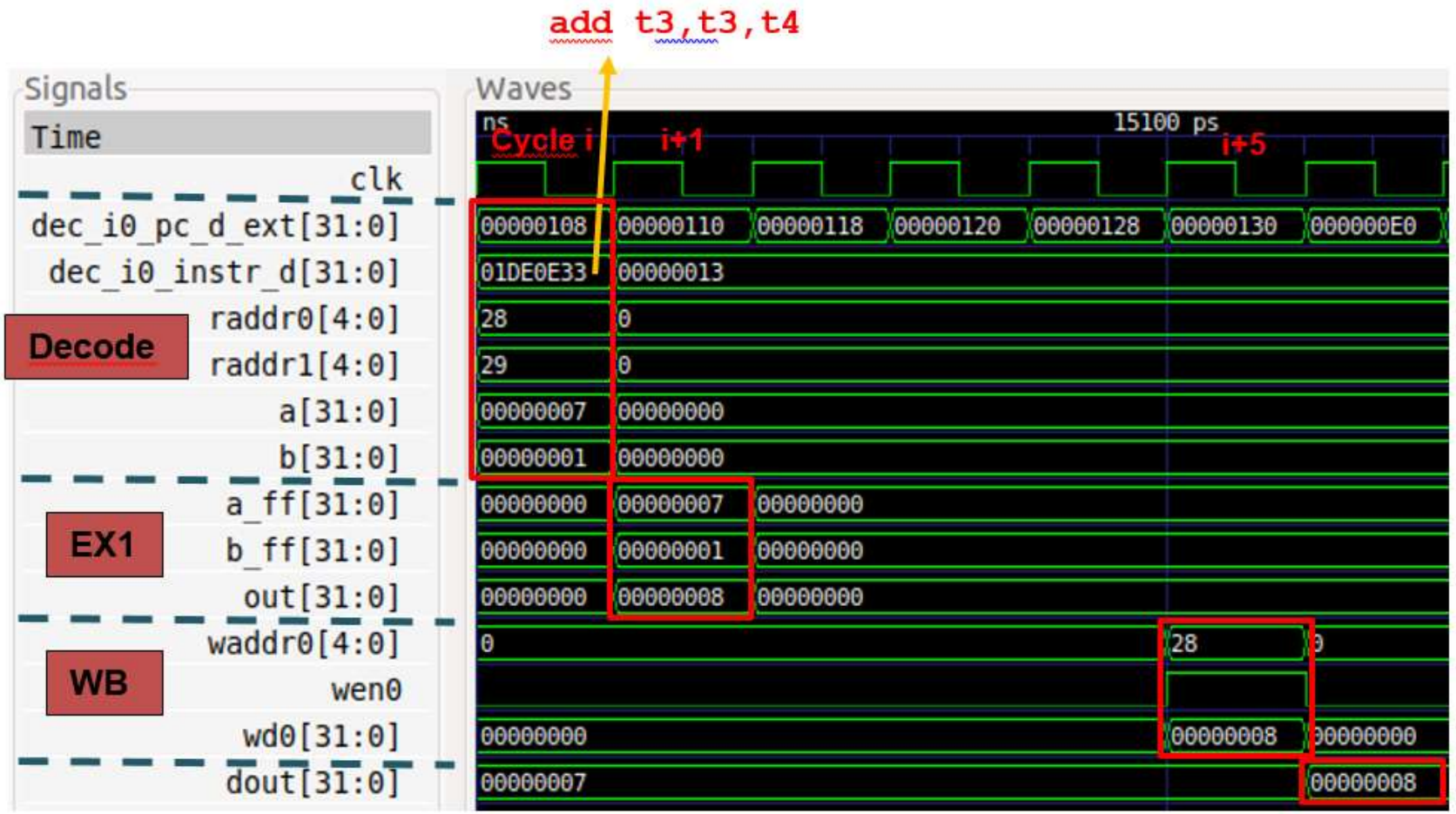
RVfpga Lab 12: Introduction

- This lab analyses the flow of arithmetic and logical instructions through the SweRV EH1 pipeline, focusing on the `add` instruction.
- Two sections:
  - **Basic analysis** of an `add` instruction
  - **Advanced analysis** of an `add` instruction
- The two analyses use the same example program, shown on the next slide.

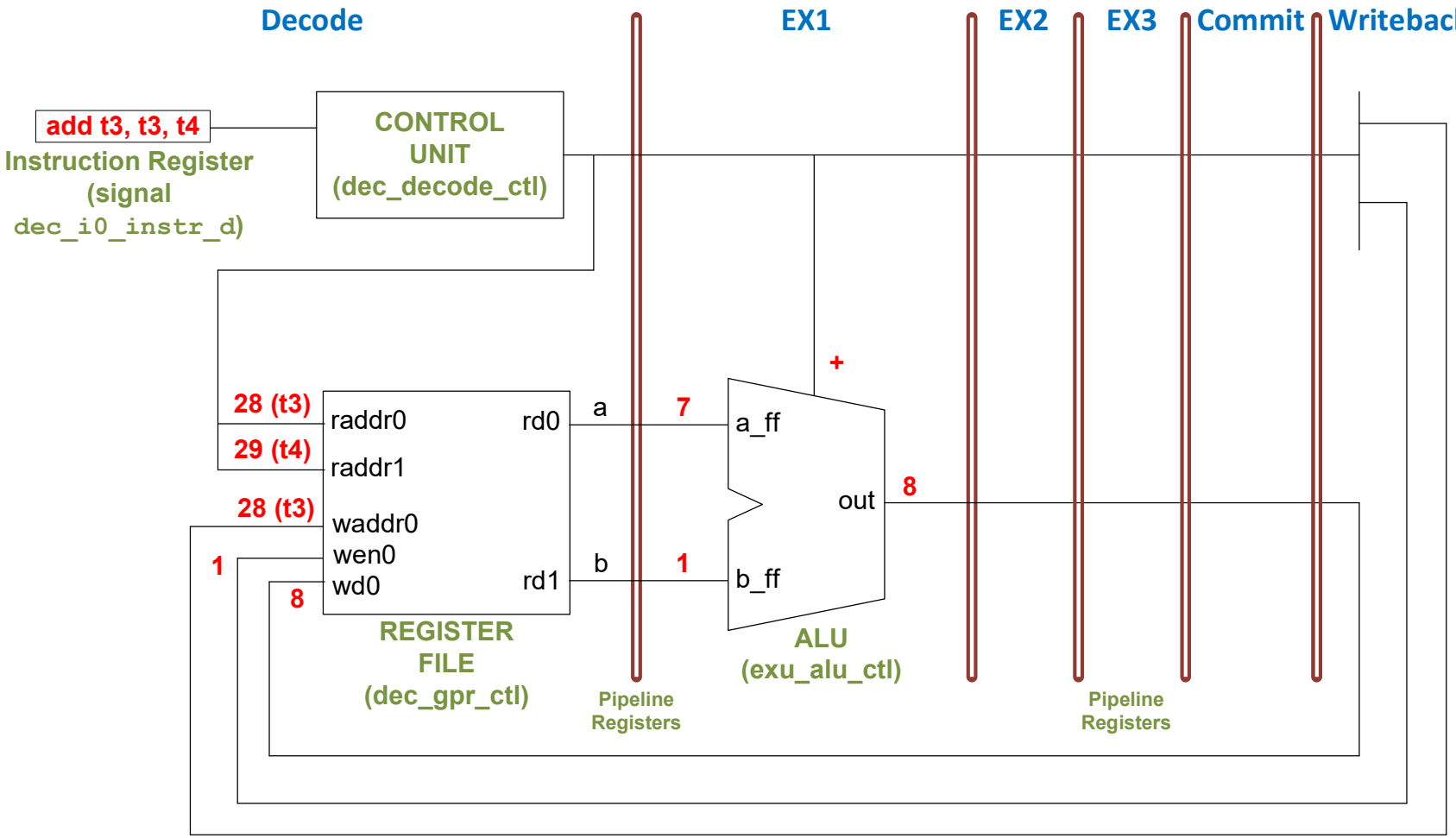
RVfpga Lab 12: Example program

```
.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 # Repeat the loop

.end
```

RVfpga Lab 12: Basic Analysis – Simulation



RVfpga Lab 12: Basic Analysis – SweRV EH1 pipeline



RVfpga Lab 12: Basic Analysis – Simulation

- Cycle i: **Decode:** Signal `dec_i0_instr_d` contains the 32-bit machine instruction 0x01DE0E33. In RISC-V, the fields for the `add` instruction are: `00 | rs1 | 000 | rd | 0110011`

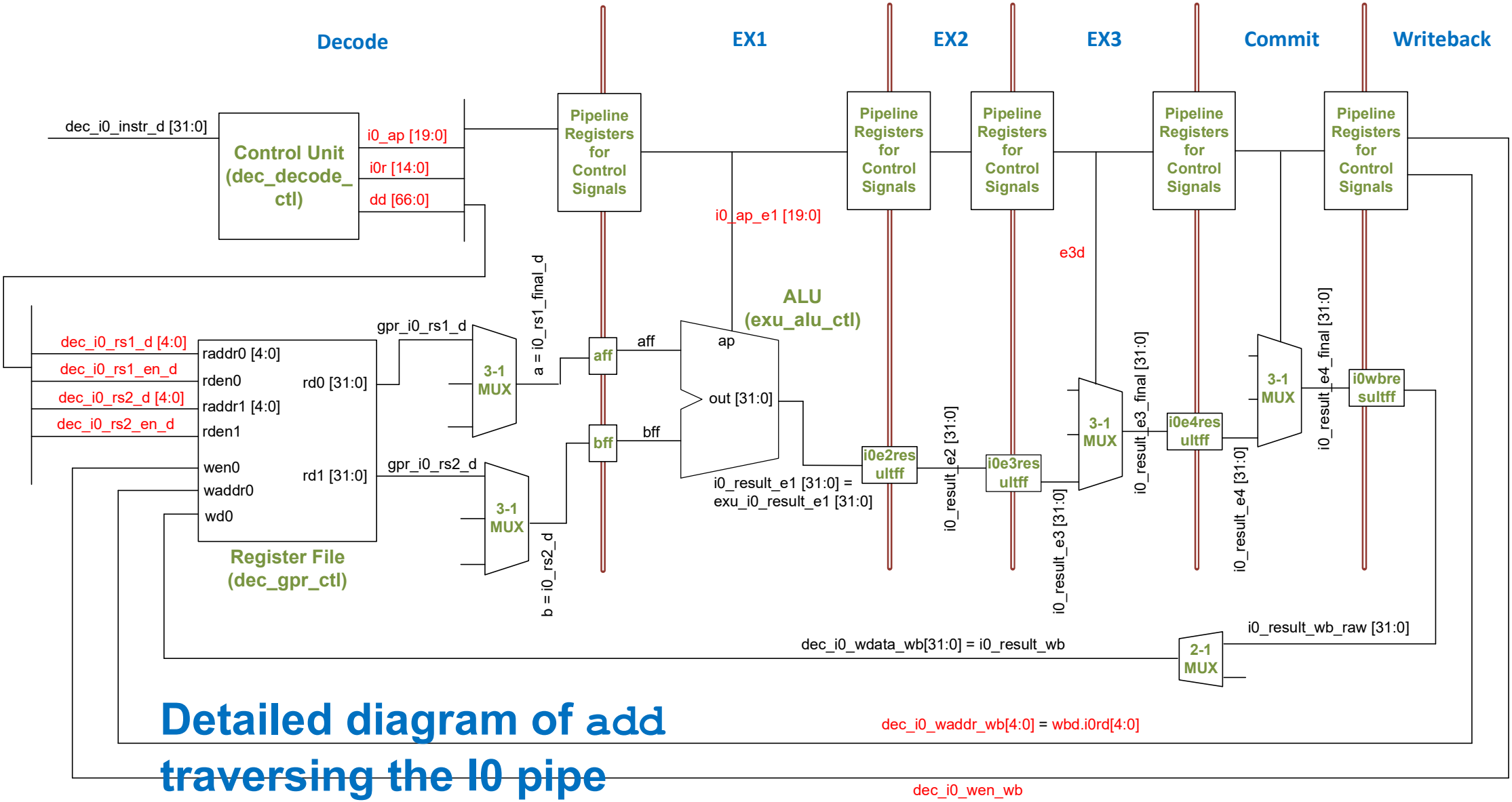
  During this stage, **control signals** are generated and the **Register File is read**. Moreover, the operands are propagated to the **I0 Pipe**.

- Cycle i+1: **EX1:** The add instruction is **executed**. The result of the addition is provided as an output of the ALU in signal `out` = 8.

- Cycle i+5: **Writeback**: The result of the addition is **written back** to the Register File: `wd0` = 0x8, `wen0` = 1 and `waddr0` = 0x28

RVfpga Lab 12: Advanced Analysis

- Figure on next page shows a detailed diagram of `add` instruction traversing the I0 pipe.



Detailed diagram of add traversing the I0 pipe

RVfpga Lab 12: Tasks and Exercises - Sample

- **TASK.** In the example from Figure 2, replace the `add` instruction with a non A-L instruction (such as a `mul` instruction). Verify that the `i0_ap` signal has all its fields equal to 0 and that this makes the I0 ALU not work.
- **TASK**. Perform a simulation of a sub instruction similar to the one from Figure 7.
- **TASK**. Analyse the Verilog implementation of the adder/subtractor implemented in module `exu_alu_ctl`.
- **TASK**. In the Verilog code, analyse how signals `wen0` and `waddr0` are generated in the Decode stage and propagated to the Writeback stage.

- **Exercises 1, 3, 4, 5.** Perform a similar analysis to the one presented in this lab for other instructions such as: `and`, `or`, `xor` , `srl`, `sra`, `sll`, `slt`, `sltu`, `addi`, `andi`, `ori`, `xori`, `srli`, `srai`, `slli`, `slti`, and `sltui`.
- **Exercises 2, 6, 7.** Exercises based on different exercises the two main reference books:
  - "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy.
  - "Digital Design and Computer Architecture: RISC-V Edition" by S. Harris and D. Harris.

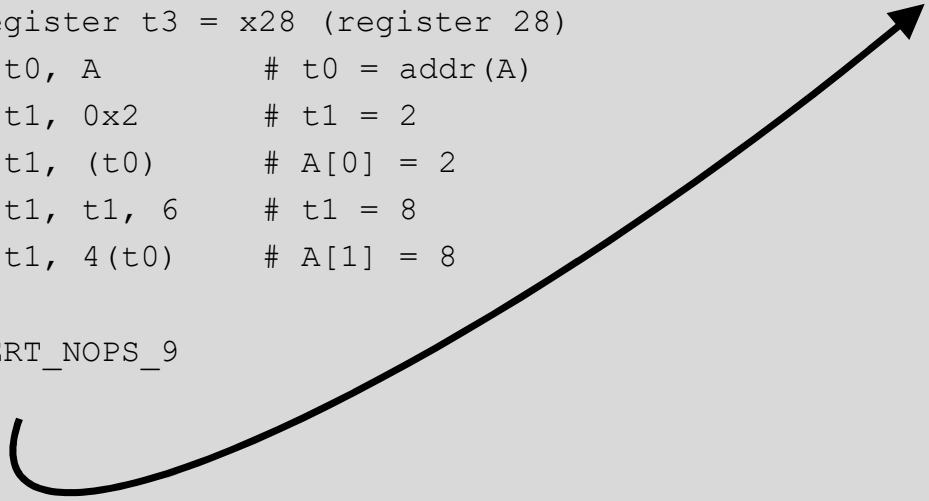
Lab 13:
Memory Instructions:
`lw` and `sw` Instructions

RVfpga Lab 13: Introduction

- Lab 13 analyses memory reads and writes.
- Three parts:
  - **Low-latency Loads:** Examine Load/Store pipe when reading low-latency DCCM (does not stall the processor).
  - **Low-latency Stores:** Examine stores to the DCCM.
  - **High-latency loads and stores:** Repeat previous analyses when reading/writing the DDR main memory available on the Nexys A7 board.

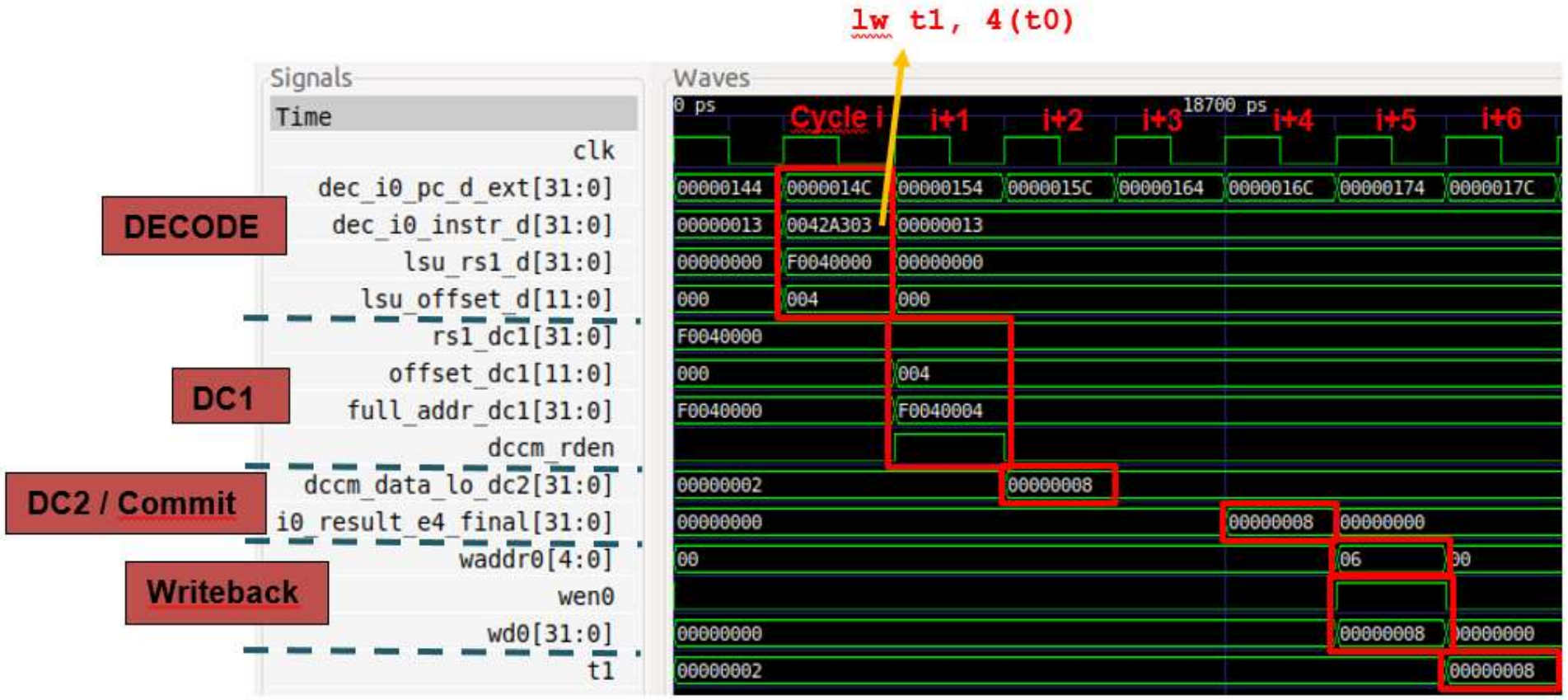
RVfpga Lab 13: Loads – Example Program

```
.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
```

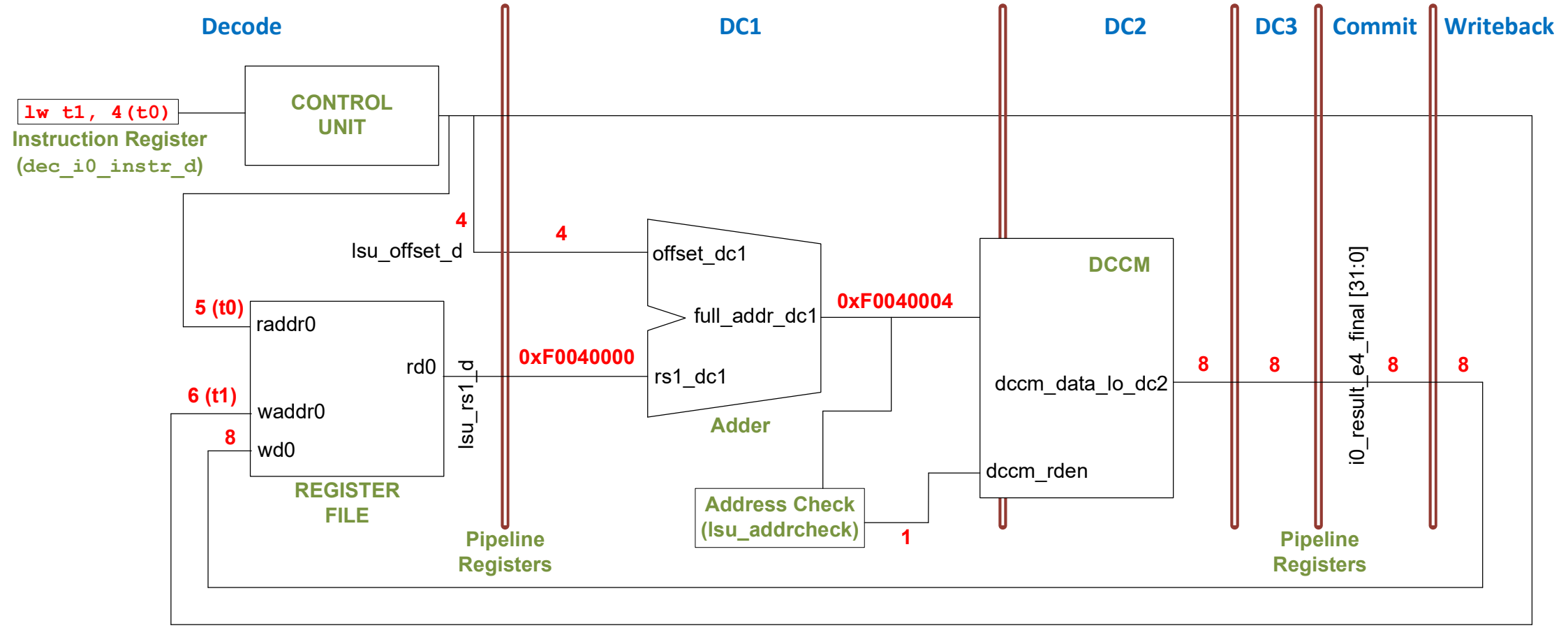


```
REPEAT:
    INSERT_NOPS_1
    lw t1, (t0)
    INSERT_NOPS_9
    INSERT_NOPS_4
    lw t1, 4(t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    beq  zero, zero, REPEAT   # Repeat the loop

.end
```

RVfpga Lab 13: Low-latency loads – Simulation



RVfpga Lab 13: Low-latency Loads – SweRV EH1 pipeline



RVfpga Lab 13: Low-latency Loads – Analysis

- **Cycle i:** **Decode:** generates **control signals** and reads **operands**:
  - t0 = 0xF0040000
  - Offset = 0x004

- **Cycle i+1:** **DC1**:
  - Computes address: `full_addr_dc1` = 0xF0040004
  - Finds memory region of the access → `dccm_rden` asserts

- **Cycle i+2:** **DC2**: the DCCM is read → `dccm_data_lo_dc2` = 0x8

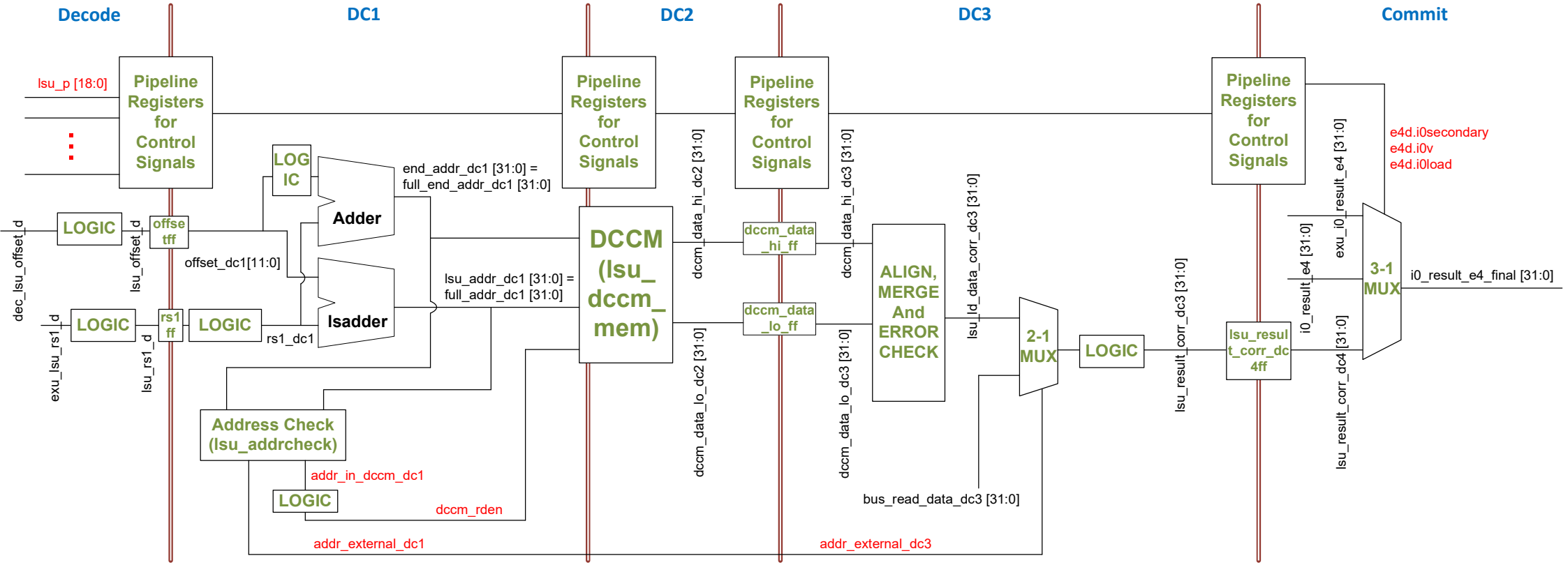
- **Cycle i+5:** **Writeback**: The value read from memory is **written back** to the register file:
  - `wd0` = 0x8
  - `wen0` = 1
  - `waddr0` = 0x6

RVfpga Lab 13: Detailed low-latency Load Analysis

- The figure in the next slide shows a detailed diagram of the main elements that a `lw` instruction traverses during its execution through the I0 Pipe.

- This was already illustrated in Lab 11, but the new figure focuses only on the LSU Pipe and provides details related to the `lw` instruction.

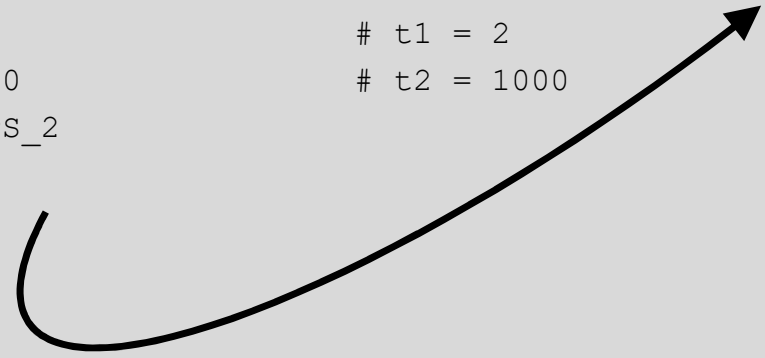
- The document for Lab 13 provides deep explanations, not included here, about each stage shown in the figure for the execution of the `lw` instruction.



Detailed diagram of `lw` traversing the I0 pipe

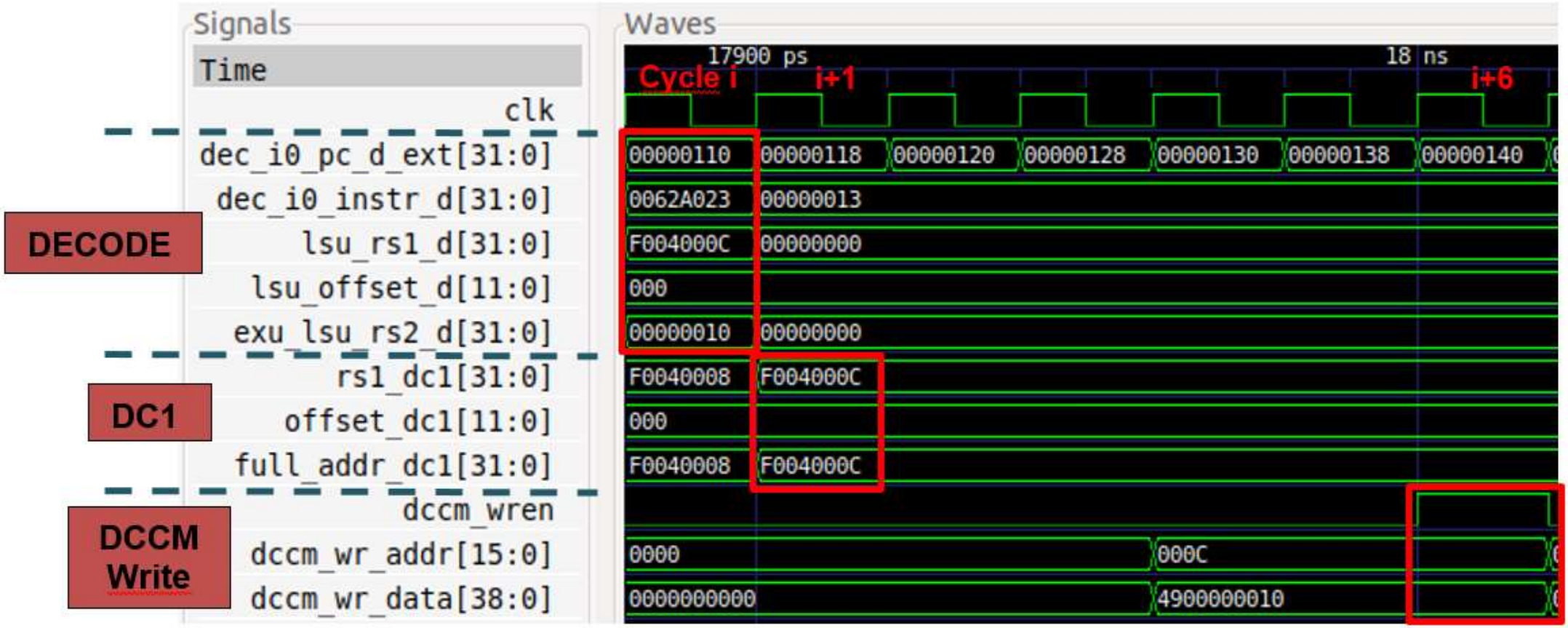
RVfpga Lab 13: Stores – Example program

```
.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
```

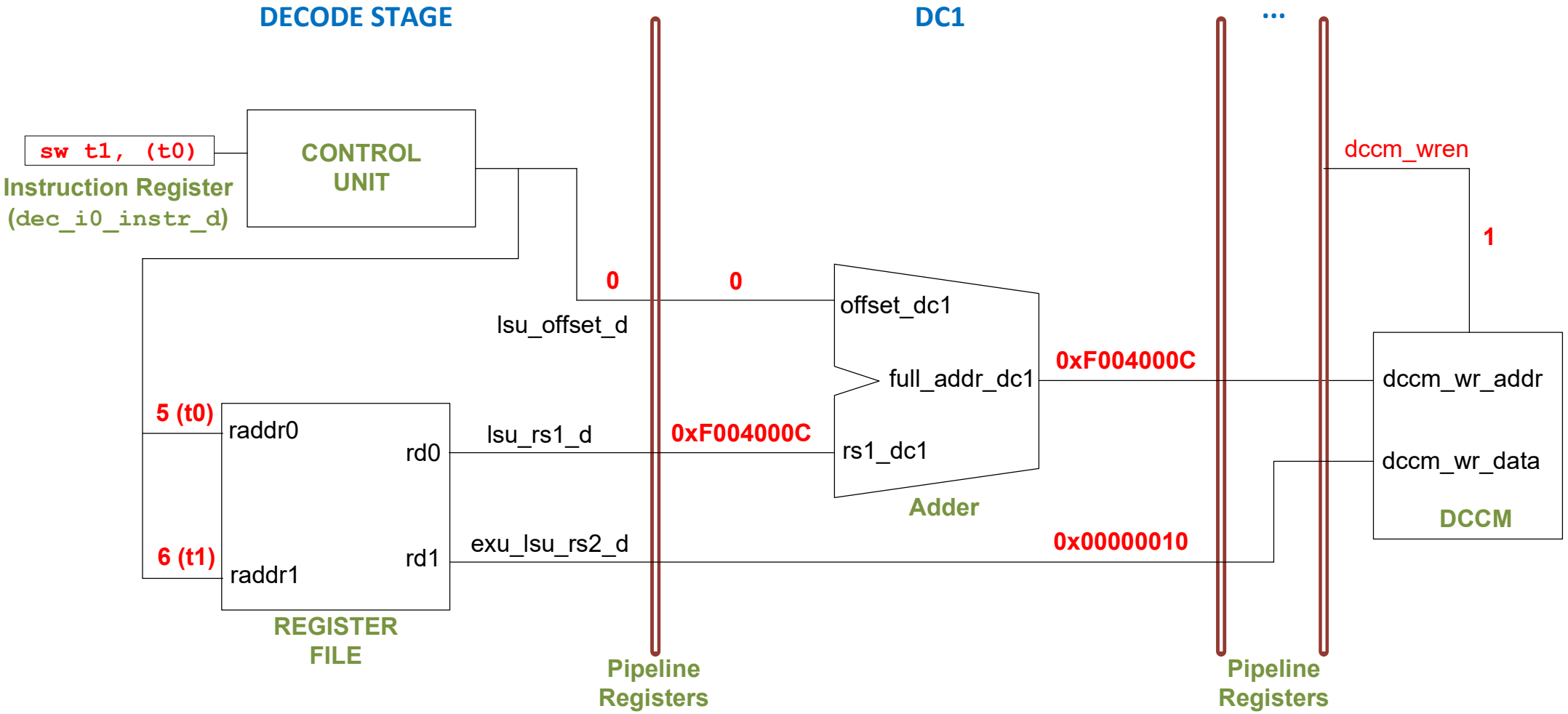


```
REPEAT:
    sw t1, (t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    lw t1, (t0)
    INSERT_NOPS_10
    add t1,t1,t1
    add t0,t0,0x04
    add t2,t2,-1
    INSERT_NOPS_10
    bne  t2, zero, REPEAT    # Repeat the loop
    nop
    nop

.end
```

RVfpga Lab 13: Low-latency Store – Simulation



RVfpga Lab 13: Low-latency Store – SweRV EH1 pipeline



RVfpga Lab 13: Store basic analysis – Simulation

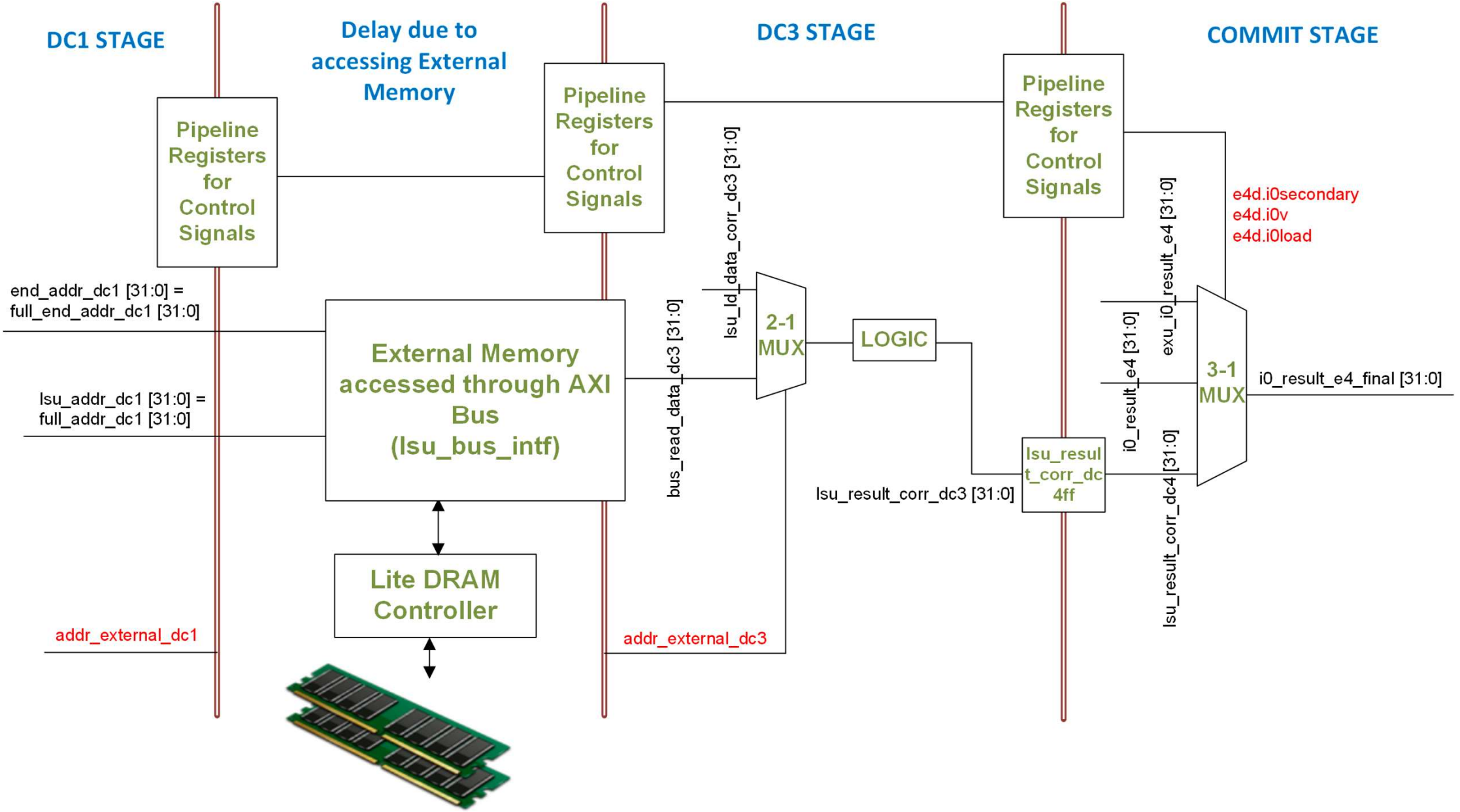
- **Cycle i: Decode:** generates **control signals** and reads **operands**:
  - `t0` = 0xF004000C
  - Offset = 0x000
  - `t1` = 0x10

- **Cycle i+1: DC1**:
  - Computes address: `full_addr_dc1` = 0xF004000C

- **Cycle i+6: DCCM write**:
  - `dccm_wr_addr` = 0x000C
  - `dccm_wr_data` = 0x4900000010

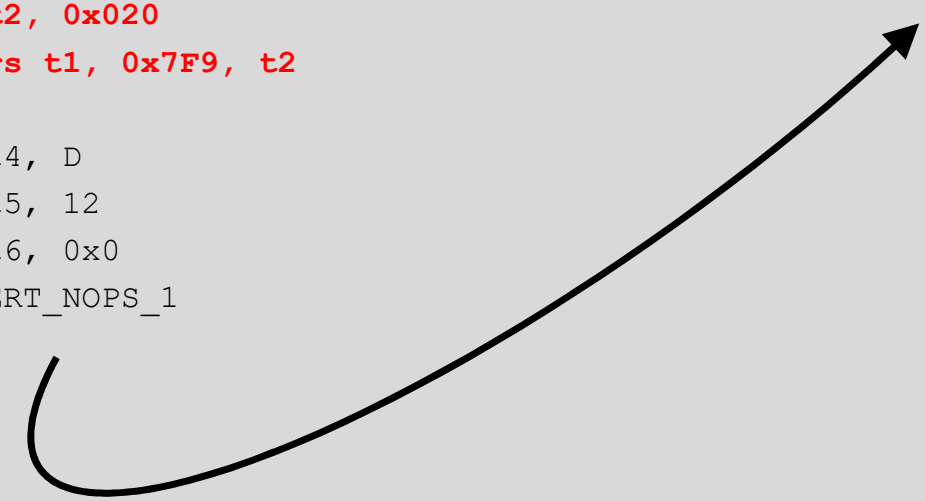
RVfpga Lab 13: Load to the External Memory

- The figure on the next slide shows the main path the `lw` instruction traverses to read Main Memory.
- The processor must stall waiting for data from the External Memory.
- The External Memory is accessed through the AXI bus, which provides the address to the Lite DRAM controller and, some cycles later, aligns and sends the requested data to the DC3 stage.
- A 2:1 multiplexer in the DC3 stage selects the data coming from the External Memory, instead of the data coming from the 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
end_addr_dc1 [31:0] = full_end_addr_dc1 [31:0]
lsu_addr_dc1 [31:0] = full_addr_dc1 [31:0]
External Memory accessed through AXI Bus (lsu_bus_intf)
Lite DRAM Controller
addr_external_dc1
addr_external_dc3
bus_read_data_dc3 [31:0]
lsu_ld_data_corr_dc3 [31:0]
2-1 MUX
LOGIC
lsu_result_corr_dc3 [31:0]
lsu_result_corr_dc4ff
exu_i0_result_e4 [31:0]
i0_result_e4 [31:0]
lsu_result_corr_dc4 [31:0]
3-1 MUX
i0_result_e4_final [31:0]
e4d.i0secondary
e4d.i0v
e4d.i0load

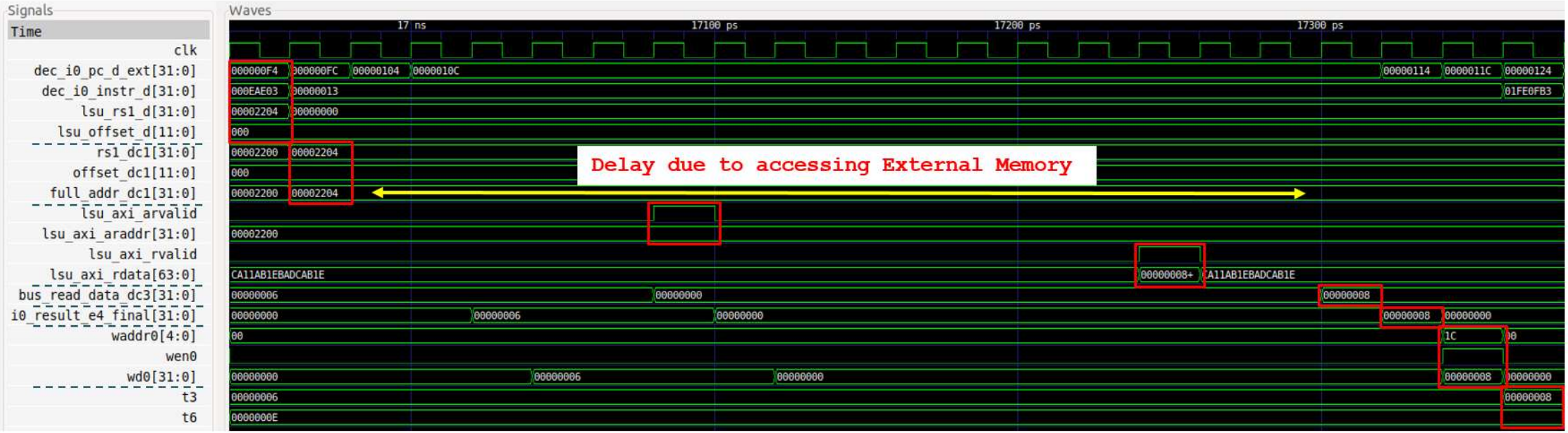
RVfpga Lab 13: External Memory – Example program

```
.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: External Memory – Simulation



RVfpga Lab 13: External Memory – Analysis

- The Decode stage computes the address, which in the fourth iteration of the example is 0x00002204.
- Then, the address is sent to the external memory through the AXI bus:
  - `lsu_axi_arvalid` = 1
  - `lsu_axi_araddr` = 0x00002200
- Some cycles later, the external memory returns 64-bit data read through the AXI Bus
  - `lsu_axi_rdata` = 0x0000000800000006
  - `lsu_axi_rvalid` = 1
- Finally, the requested 32-bit data is extracted from the 64-bit data, inserted in the main pipeline path, and written into the Register File.

RVfpga Lab 13: Tasks - Sample

- **TASK**. Include signal `lsu_p` in the simulation from Figure 4 and analyse its bits.
- **TASK**. Analyse in the Verilog code the path followed by the two inputs to the LSU (`exu_lsu_rs1_d` and `dec_lsu_offset_d`) from the sources where they are obtained. Several modules are involved in this process: `dec`, `exu`, `lsu`.
- **TASK**. Analyse the implementation of the two adders from the DC1 stage, which are instantiated in module `lsu_lsc_ctl`.
- **TASK**. In the program from Figure 2, try different access sizes (byte, half-word) and unaligned accesses. To do so, change the offset or the access type from `lw` to lb (load byte) or lh (load half-word). For example, if you change the offset from 4 to 3, the load word instruction performs an unaligned access to the 32-bits starting at address 0xF0040003, as shown in Figure 8. Analyse the value of signals `lsu_addr_dc1[31:0]` (or `full_addr_dc1[31:0]`) and `end_addr_dc1[31:0]` under these different situations.
- **TASK**. Analyse unaligned stores to the DCCM, as well as sub-word stores: store byte (`sb`) or store half-word (`sh`).
- **TASK**. It can also be interesting to analyse the AXI Bus implementation for accessing the DRAM Controller, for which you can inspect the `lsu_bus_intf` module.

Lab 14:
Structural Hazards

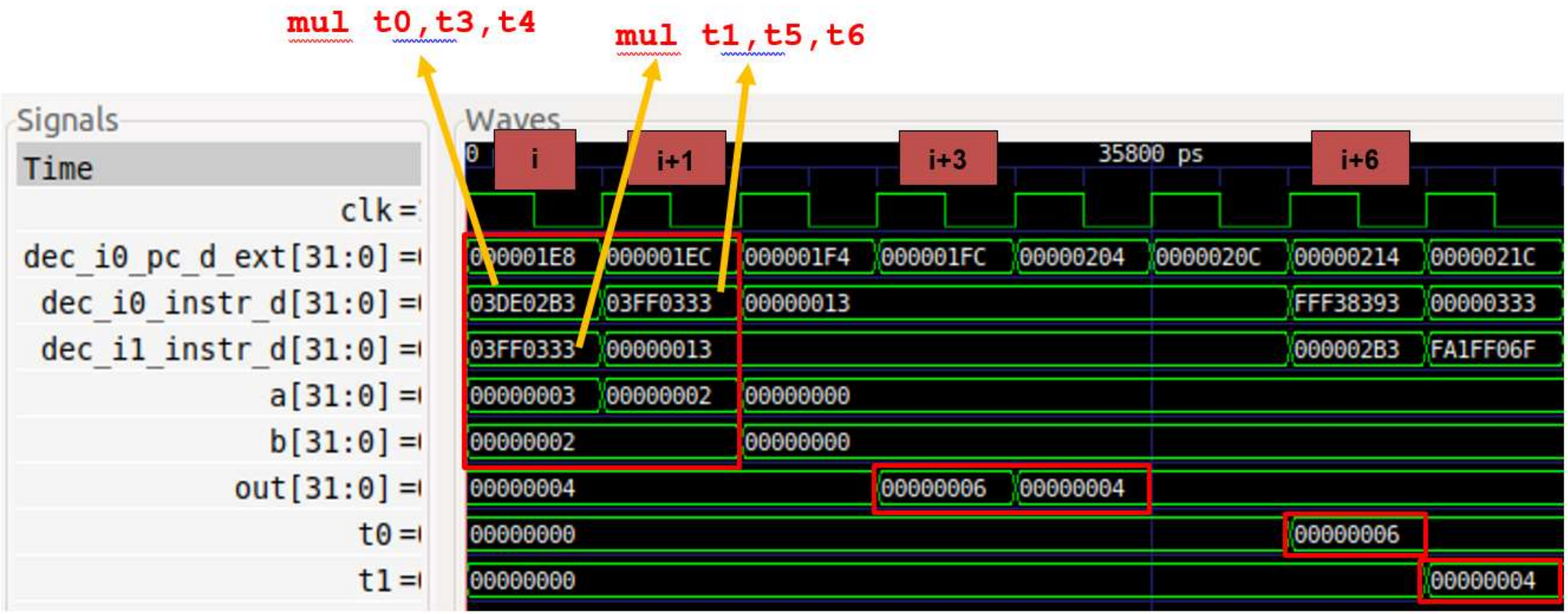
RVfpga Lab 14: Introduction

- Lab 14 illustrates **two structural hazards** (which have different performance-cost trade-offs).
  - **Unit conflict:** two `mul` instructions arrive at the Decode stage in the same cycle. The multiplier is pipelined, so the second `mul` instruction is only delayed by one cycle. Hardware cost and performance degradation (only one cycle) are low.
  - **Register File write port conflict:** Three instructions arrive at the Writeback stage in the same cycle, one of them being a non-blocking load executed several cycles earlier. SweRV EH1 has three (instead of two) write ports. The structural hazard is avoided (resulting in no performance loss), but it has high hardware cost due to the extra register file port.
  - Note that the `div` instruction can also cause hazards, which is discussed in the lab's Appendix.

RVfpga Lab 14: 2 `mul` Instructions – Example Program

```
.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      # Stay in the loop?
   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
```

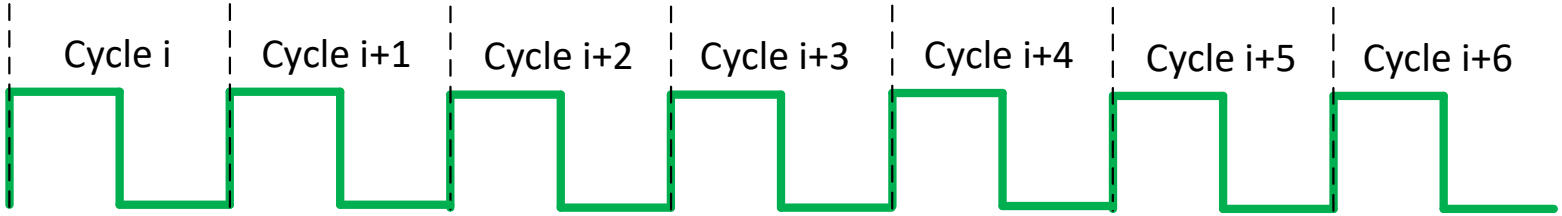
RVfpga Lab 14: 2 `mul` Instructions – Simulation



RVfpga Lab 14: 2 `mul` instructions – Analysis

- **Cycle i**: The two `mul` instructions arrive at the Decode stage in the same cycle. A Structural Hazard prevents the second `mul` instruction from advancing.
- **Cycle i+1**: The first `mul` instruction executes in the first stage of the pipelined multiplier (M1), while the second `mul` instruction waits in the Decode stage.
- **Cycle i+2**: The first `mul` instruction executes in the second stage of the pipelined multiplier (M2) and the second `mul` executes in the first stage (M1).
- **Cycle i+3**: The first `mul` instruction obtains the result: `out` = 0x6.
- **Cycle i+4**: The second `mul` instruction obtains the result: `out` = 0x4.
- **Cycle i+6**: The register file is updated with the result of the first `mul` (`t0` = 0x6).
- **Cycle i+7**: The register file is updated with the result of the second `mul` (`t1` = 0x4).

RVfpga Lab 14: 2 `mul` Instructions – Diagram



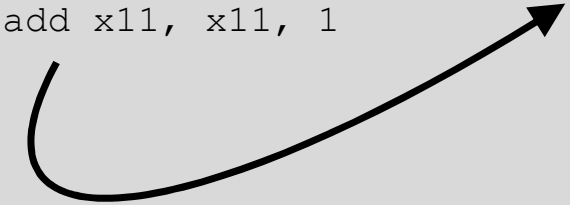
	Cycle i	Cycle i+1	Cycle i+2	Cycle i+3	Cycle i+4	Cycle i+5	Cycle i+6
`mul t0, t3, t4 (03de02b3)`	D Way0	M1	M2	M3	C Way0	WB Way0	
`mul t1, t5, t6 (03ff0333)`	Dst Way1	D Way0	M1	M2	M3	C Way0	WB Way0
`nop (00000013)`	A Way0	D Way1	EX1 I1	EX2 I1	EX3 I1	C Way1	WB Way1
`nop (00000013)`	Ast Way1	A Way0	D Way0	EX1 I0	EX2 I0	EX3 I0	C Way0

RVfpga Lab 14: 3 Simultaneous Writes – Example Program

```
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
```



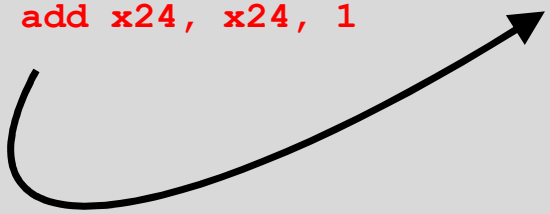
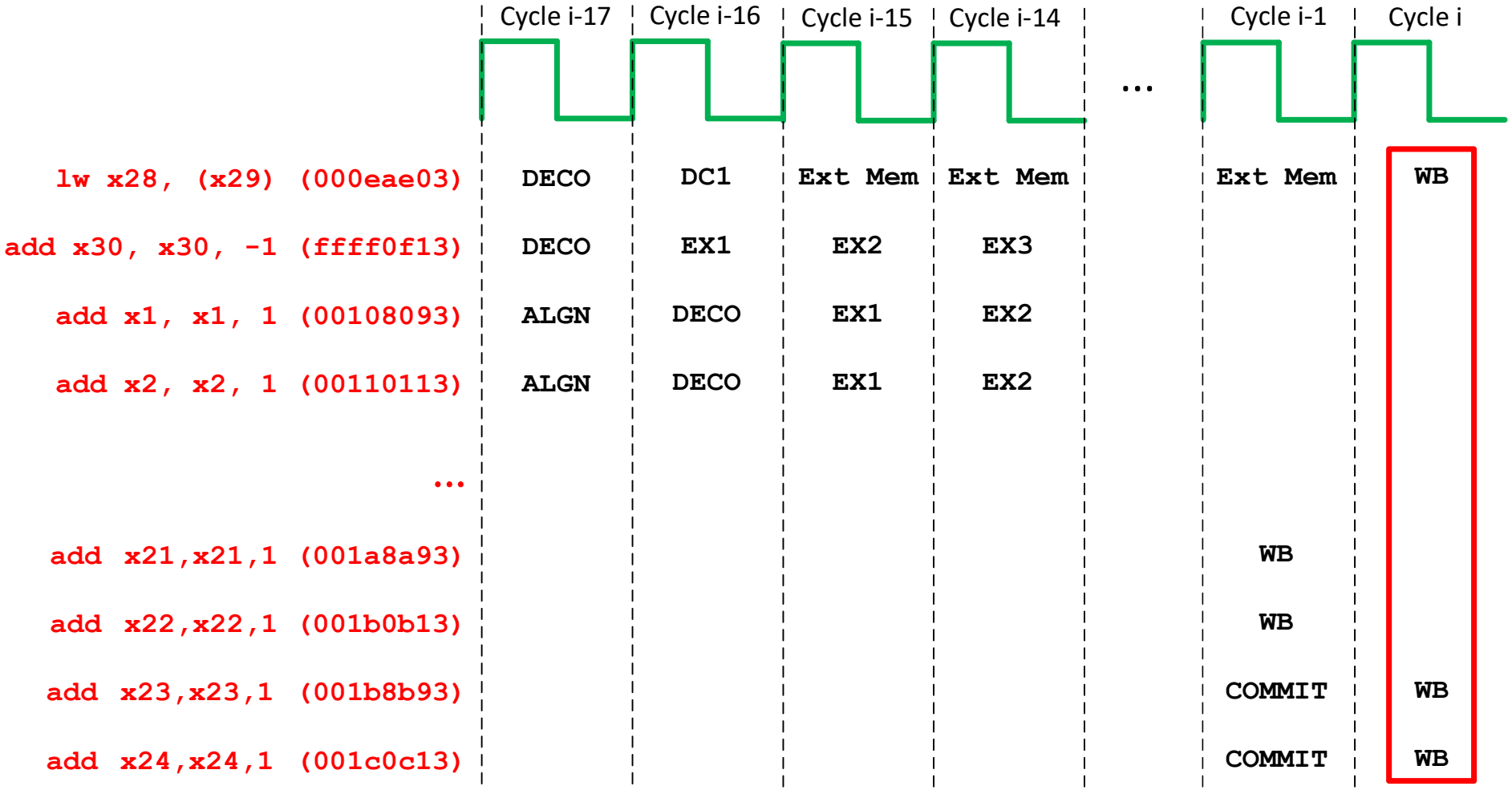


Figure 6. Verilator simulation for the example from Figure 4

RVfpga Lab 14: 3 Simultaneous Writes – Analysis

- **Cycle i-17**: The `lw` instruction is at the Decode stage.
- **Cycle i-16**: The effective memory address is computed and sent to the External Memory through the AXI Bus. The `load` instruction waits several cycles for the External Memory to supply the data.
- **Cycle i-5**: The two conflicting `add` instructions are decoded.
- **Cycle i**: The `lw` instruction and the two conflicting `add` instructions proceed to the Writeback stage, where they write the register file, which is possible because the Register File has three write ports.

RVfpga Lab 14: 3 Simultaneous Writes – Diagram



RVfpga Lab 14: Tasks and Exercises - Sample

- **TASK**: Inspect the Verilog code from `exu_mul_ctl` and see how the multiplication is computed. Remember that RISC-V includes 4 multiply instructions (`mul, mulh, mulhsu` and `mulhu`), and all of them must be supported by the hardware.
- **TASK**: Remove the nop instructions included within the loop from Figure 1 and measure different events (cycles, instructions/multiplies committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 2? Justify your answer. Now reorder the code within the loop trying to reach the ideal throughput. Justify the results obtained in the original code and in the reordered one.
- **TASK**: Modify the program from Figure 1, replacing the two mul instructions for two lw instructions to the DCCM. You should observe a structural hazard analogous to the one analysed in this section and resolved in a similar way.
- **TASK**: Compare the simulation shown in Figure 6 (non-blocking load) with the simulation shown in Figure 14 of Lab 13 (blocking load).
- **Exercise 1**. Analyse, both in simulation and on the board, the structural hazard that happens between two consecutive memory instructions (you can analyse any combination of two consecutive memory instructions such as loads and stores) that arrive at the L/S Pipe in the same cycle.
- **Exercise 2**. This following exercise is based on exercise 4.22 from the book “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([PaHe]).

Lab 15:
Data Hazards

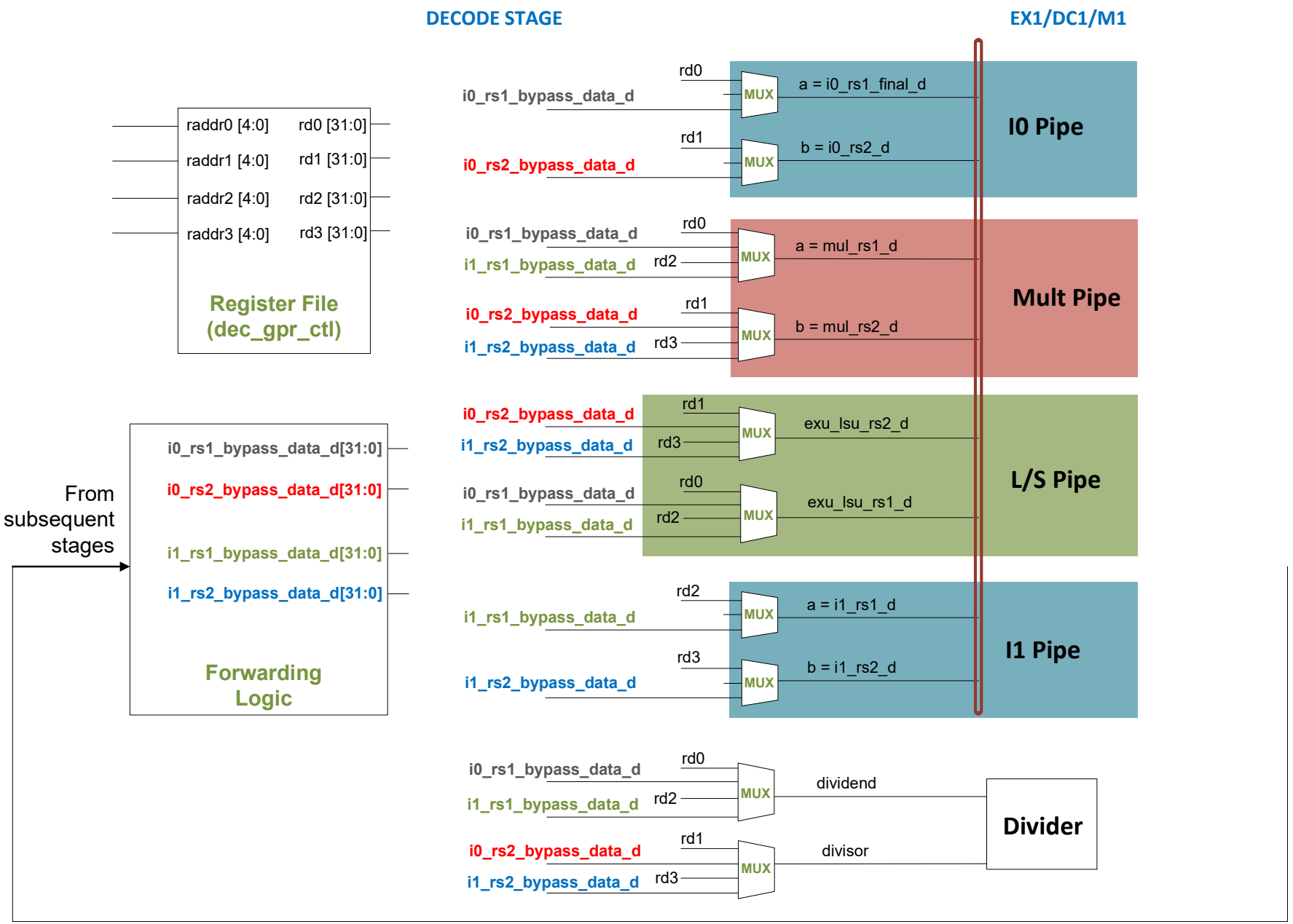
RVfpga Lab 15: Introduction

- Lab 15 analyses how **RAW data hazards** are resolved.
- RAW data hazards are resolved by **stalling** the processor or **forwarding** (also called bypassing) the value from an instruction executing in a later stage.
- **Two scenarios** analysed:
  - RAW data hazards resolved by **forwarding** to the Decode stage (using several new multiplexers)
  - RAW data hazards resolved in the Commit stage using **two additional ALUs**

RVfpga Lab 15: Solving Data Hazards by Forwarding

- Forwarding to the Decode stage requires adding multiplexers in front of the Functional Units (ALUs, Multiplier, Adder that computes the Effective Address in DC1, etc.) to select operands from either the Register File or from subsequent stages.

- The figure on the next slide shows the forwarded values in the Decode stage. The Forwarding Logic produces bypass signals for each of the two source operands in each of the Ways

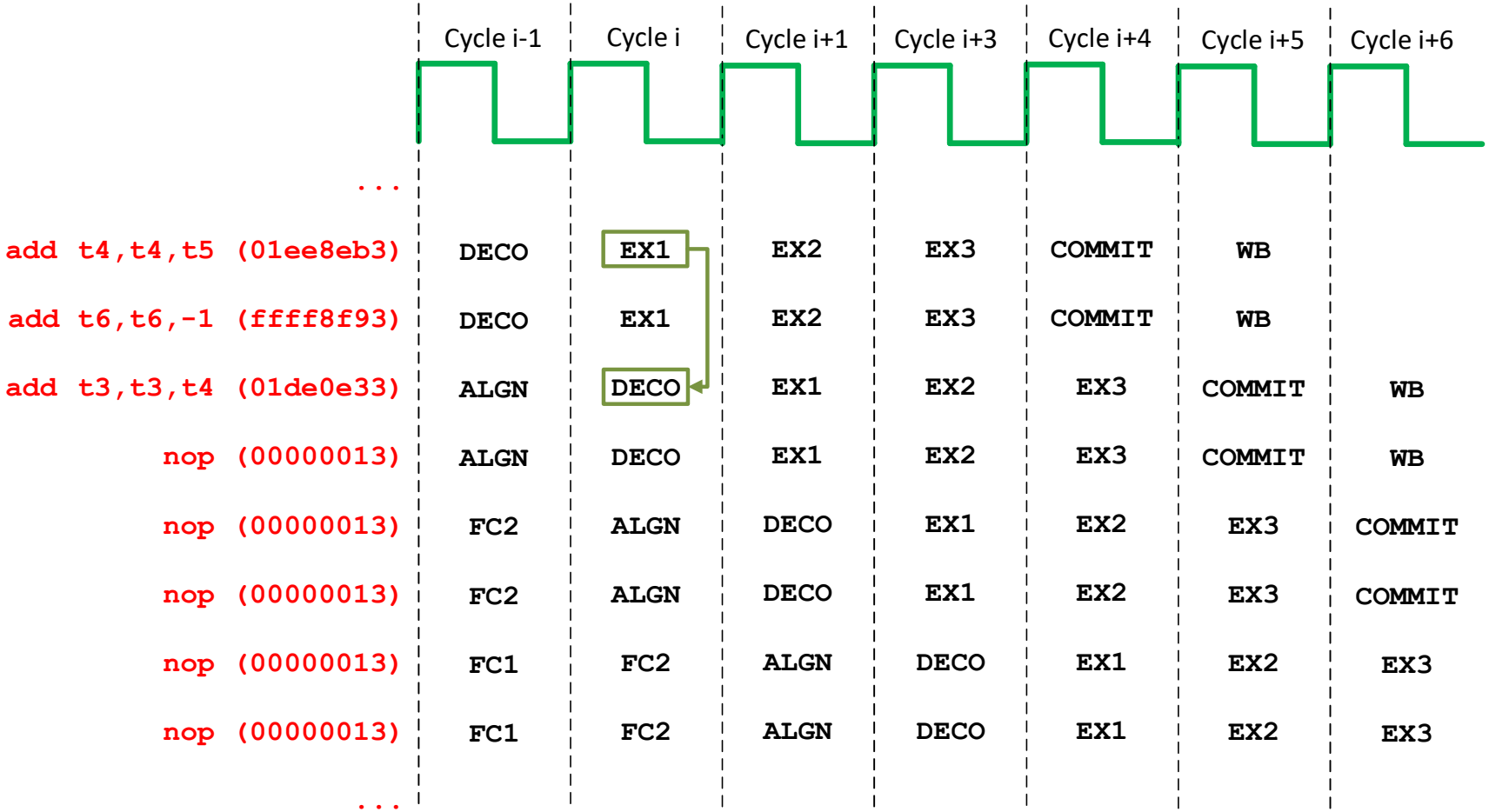

DECODE STAGE
EX1/DC1/M1
raddr0 [4:0]
raddr1 [4:0]
raddr2 [4:0]
raddr3 [4:0]
rd0 [31:0]
rd1 [31:0]
rd2 [31:0]
rd3 [31:0]
Register File
(dec_gpr_ctl)
From
subsequent
stages
i0_rs1_bypass_data_d[31:0]
i0_rs2_bypass_data_d[31:0]
i1_rs1_bypass_data_d[31:0]
i1_rs2_bypass_data_d[31:0]
Forwarding
Logic
i0_rs1_bypass_data_d
i0_rs2_bypass_data_d
i1_rs1_bypass_data_d
i1_rs2_bypass_data_d
rd0
rd1
rd2
rd3
MUX
a = i0_rs1_final_d
b = i0_rs2_d
I0 Pipe
a = mul_rs1_d
b = mul_rs2_d
Mult Pipe
exu_lsu_rs2_d
exu_lsu_rs1_d
L/S Pipe
a = i1_rs1_d
b = i1_rs2_d
I1 Pipe
dividend
divisor
Divider

RVfpga Lab 15: Solving Data hazards by Forwarding – Example

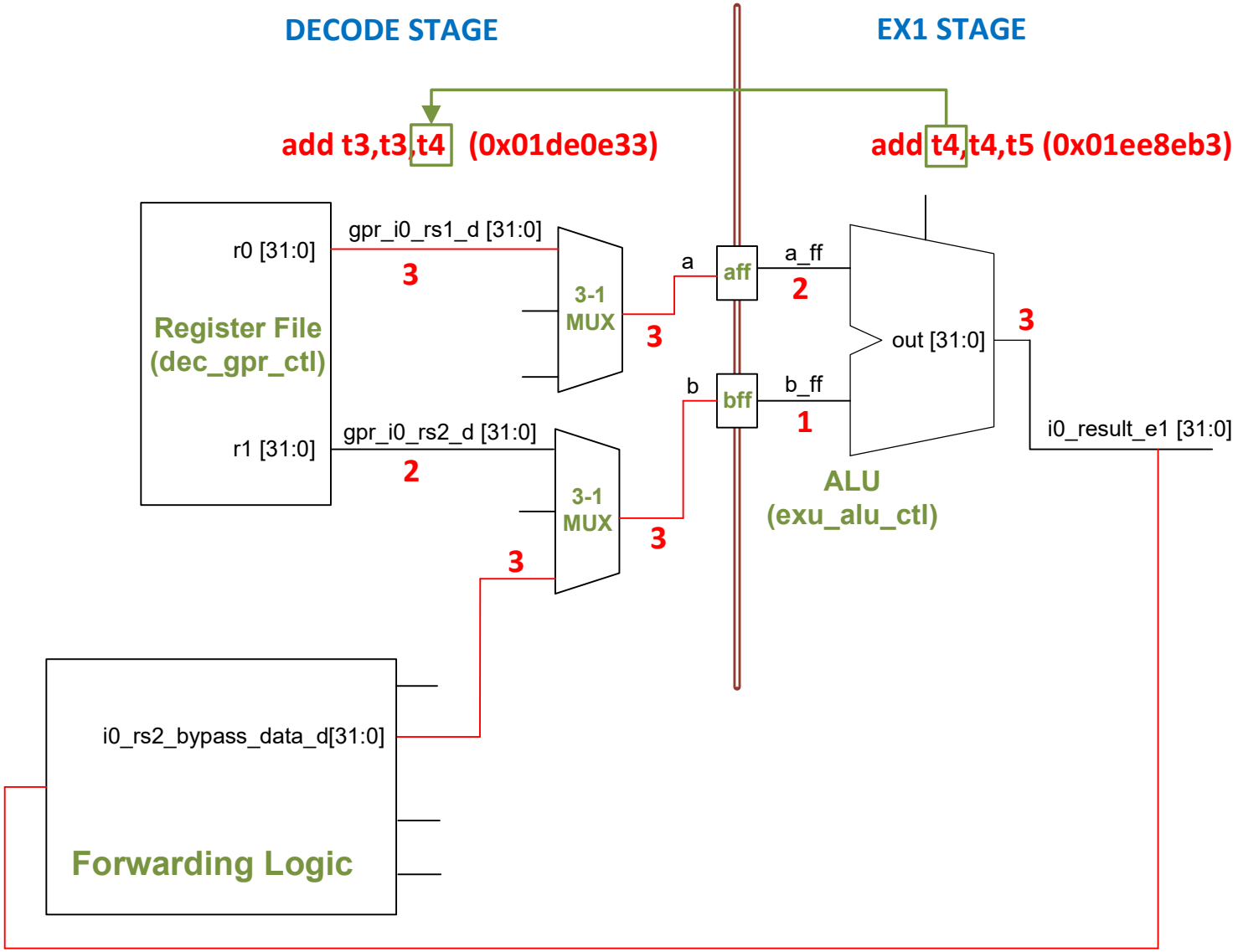
```
.globl Test_Assembly
.text

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    # Repeat the loop
```

```
00000180 <REPEAT>:
 180:    01ee8eb3        add     t4,t4,t5
 184:    ffff8f93        addi    t6,t6,-1
 188:    01de0e33        add     t3,t3,t4
 18c:    00300e13        li      t3,3
 190:    00200e93        li      t4,2
 194:    00100f13        li      t5,1
 198:    fe0f94e3        bnez    t6,180 <REPEAT>
```

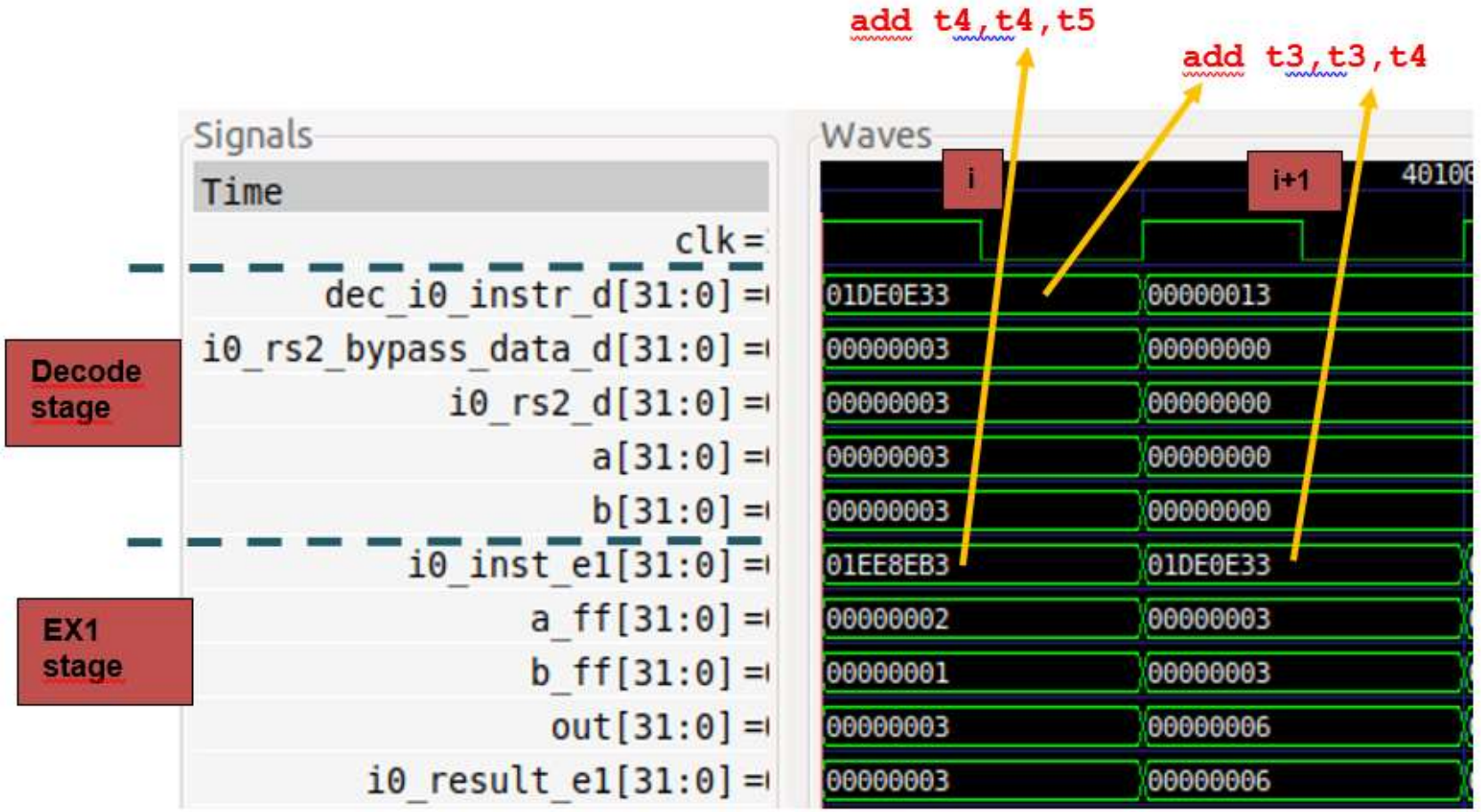
RVfpga Lab 15: Solving Data Hazards by Forwarding – Diagram



RVfpga Lab 15: Solving Data Hazards by Forwarding – Pipeline – Cycle *i*

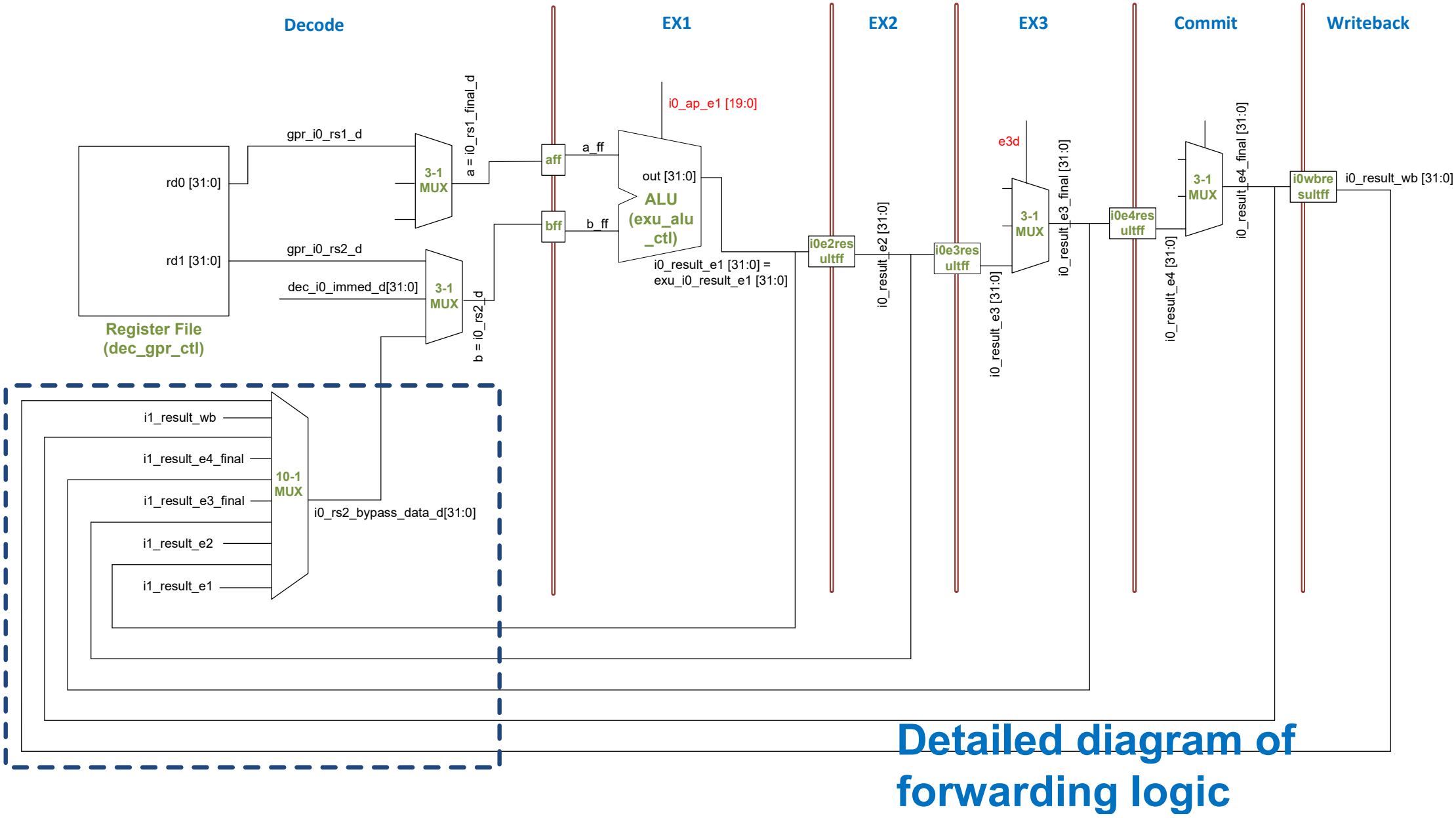


RVfpga Lab 15: Solving Data Hazards by Forwarding – Simulation



RVfpga Lab 15: Solving Data Hazards by Forwarding – Analysis

- Instruction `add **t4**,t4,t5` (0x01ee8eb3):
  - **Cycle *i*:** This `add` instruction is in the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01ee8eb3). It computes the following addition in the ALU:
    - a_ff (2) + b_ff (1) = out (3)
  - The result is sent to the Forwarding Logic in the Decode stage.
- Instruction `add t3,t3,**t4**` (0x01de0e33):
  - **Cycle *i*:** This `add` instruction is in the Decode stage of Way-0 (`dec_i0_instr_d` = 0x01de0e33). The Forwarding Logic forwards the result from EX1 (`i0_result_e1`) to the Decode stage (`i0_rs2_bypass_data_d`). Two 3:1 multiplexers produce the operands, specifically:
    - Operand a = 3 (from the Register File)
    - Operand b = 3 (from the ALU output in the EX1 stage of the I0 Pipe, through the Forwarding Logic)
  - **Cycle *i+1*:** This `add` instruction is in the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01de0e33). It computes the correct addition in the ALU:
    - a_ff (3) + b_ff (3) = out (6)



Detailed diagram of forwarding logic

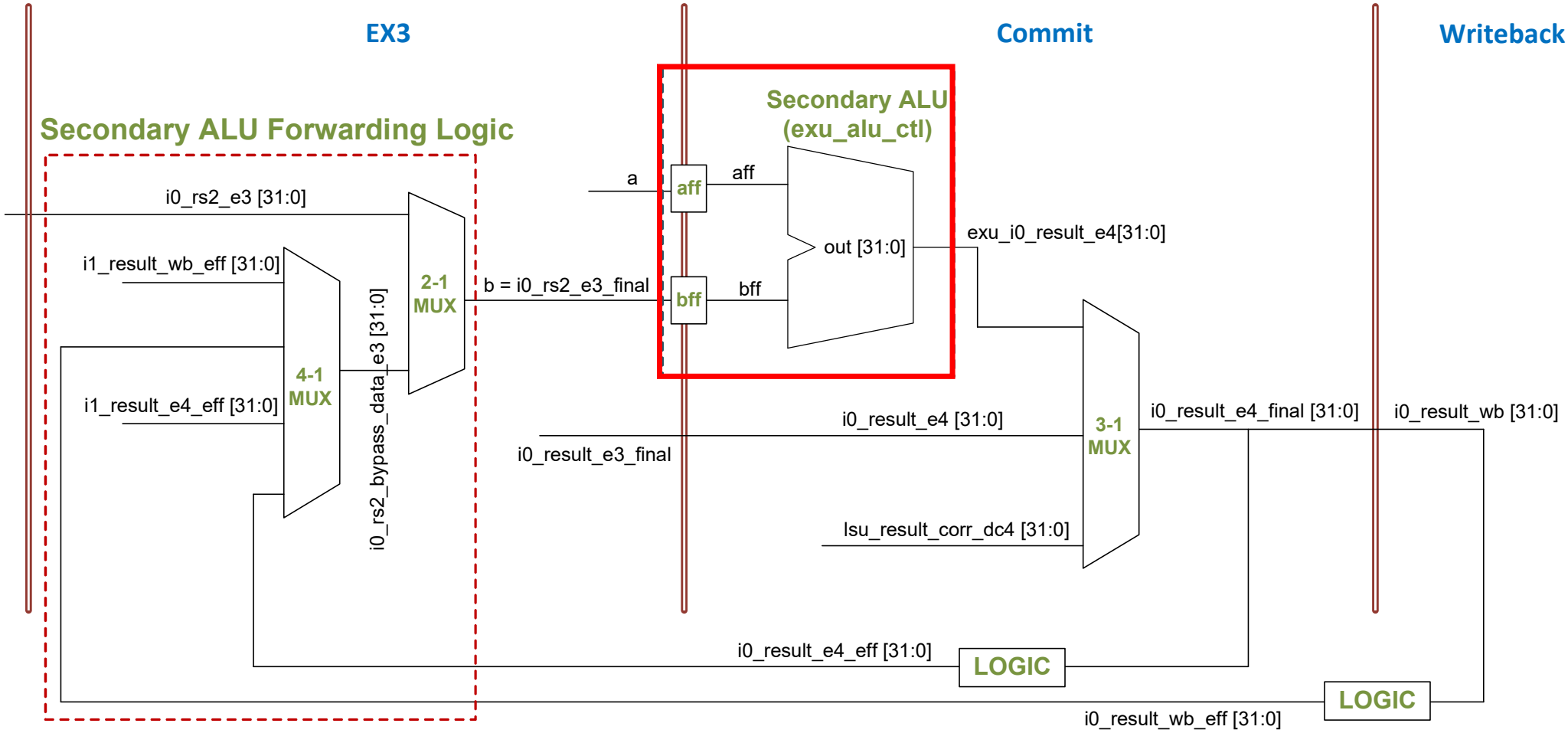
RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit

- Instructions that need several cycles to obtain the result (i.e. a multi-cycle operation, such as a `lw`, `mul`, and `div`) **cannot forward to Decode** stage.

- But SweRV EH1 adds an extra ALU (the **Secondary ALU**) in the Commit stage of each way. This ALU recalculates the arithmetic-logic operation with the proper inputs when necessary.

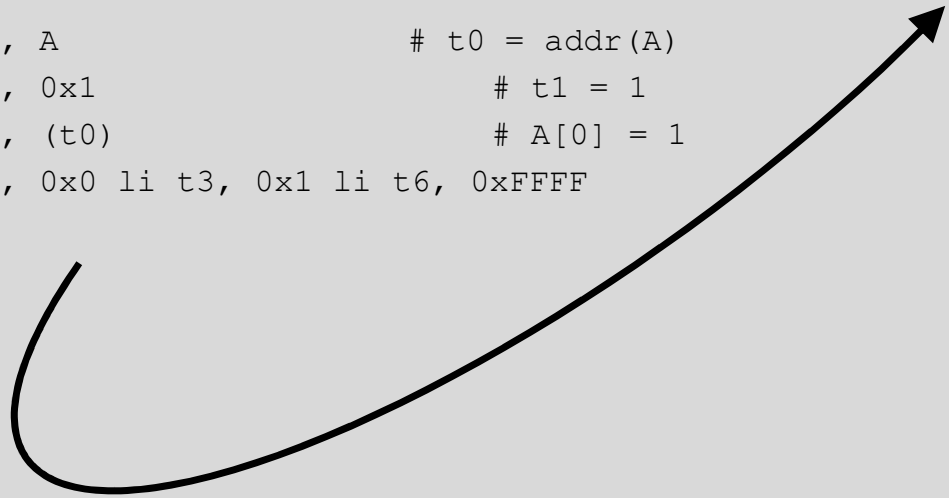
- Thus, **no cycles** are lost due to stalling – but the cost is **two extra ALUs** (one per way) as well as added control signals and logic.

RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit – Pipeline



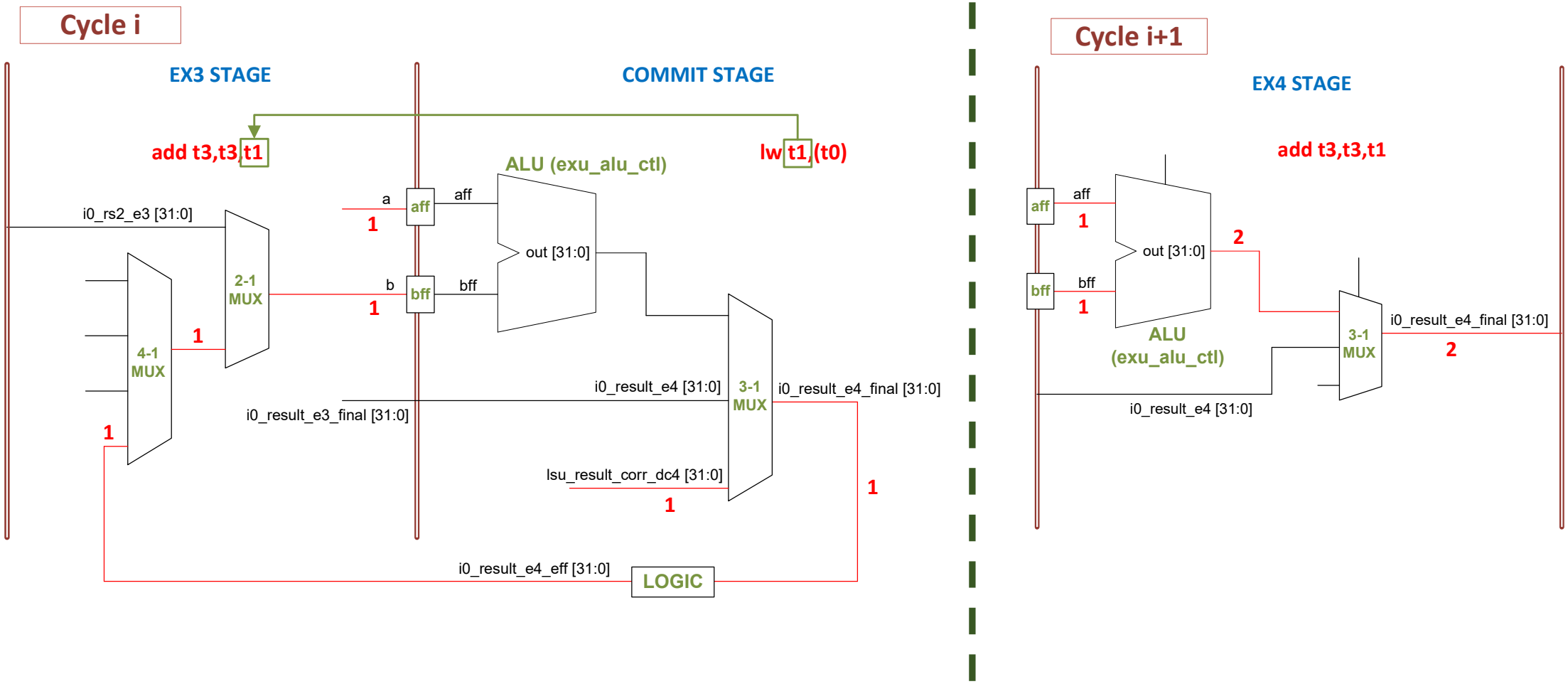
RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit – Example

```
.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
```

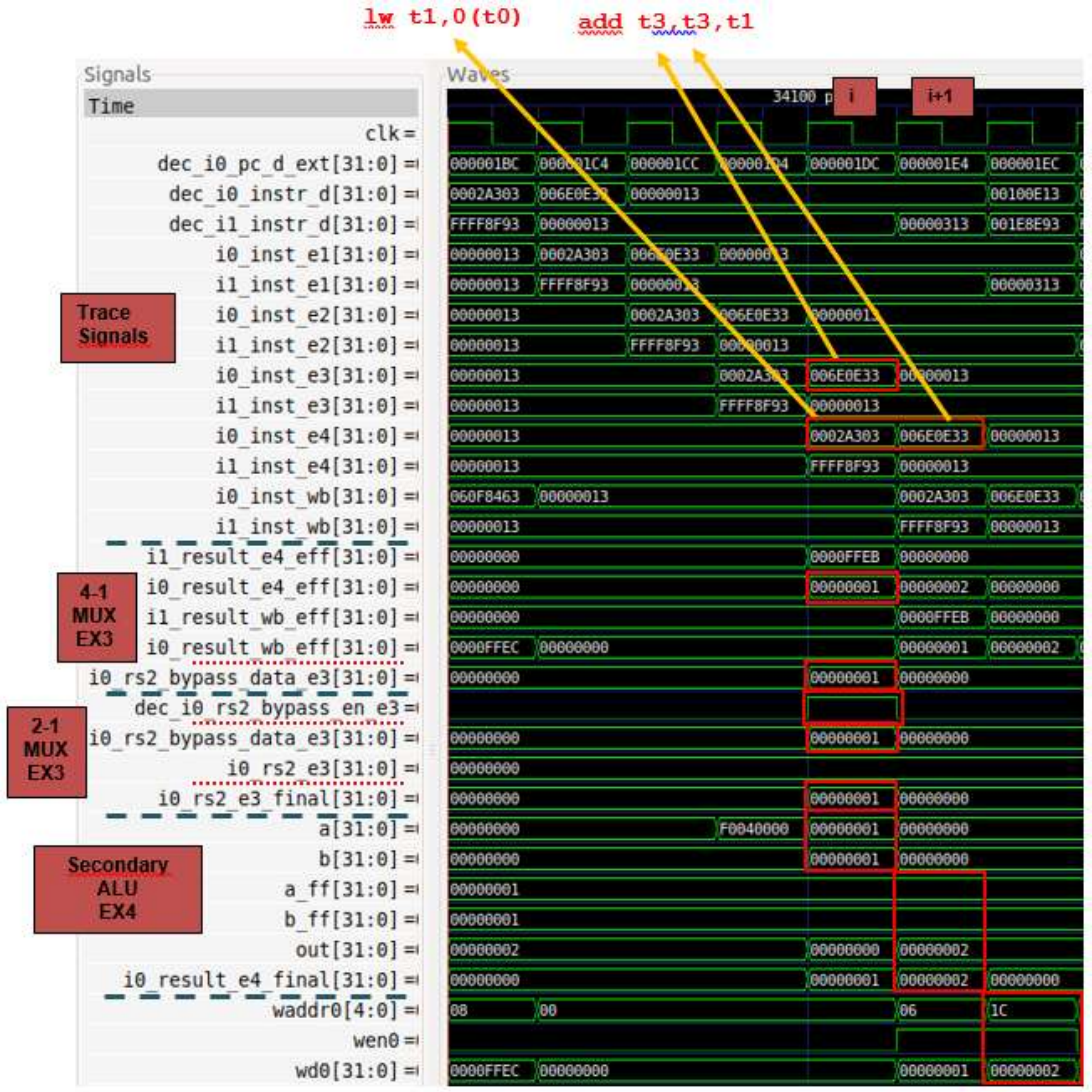


```
REPEAT:
   beq t6, zero, OUT       # Stay in the loop?
   INSERT_NOPS_9
   lw t1, (t0)
   add t6, t6, -1
   add t3, t3, t1           # t3 = t3 + t1
   INSERT_NOPS_8
   li t1, 0x0
   li t3, 0x1
   add t4, t4, 0x1
   add t5, t5, 0x1
j REPEAT

OUT:

.end
```

RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit – Pipeline



RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit – Simulation



RVfpga Lab 15: Solving Data Hazards by Forwarding at Commit – Simulation

- **Trace Signals**
  - **Cycle *i*:** the `add` instruction is in the EX3 stage of Way 0 (`i0_inst_e3` = 0x006E0E33), and the `lw` instruction is in the Commit stage of the I0 Pipe (`i0_inst_e4` = 0x0002A303).
  - **Cycle *i+1*:** the `add` instruction is in the Commit stage of Way 0 (`i0_inst_e4` = 0x006E0E33).
- **4:1 Multiplexer**
  - **Cycle *i*:** the result from the `lw` instruction (in the Commit stage), is selected:

    `i0_rs2_bypass_data_e3` = `i0_result_e4_eff` = 0x00000001
- **2:1 Multiplexer**
  - **Cycle *i*:** the bypass value is selected due to the dependency between the `lw` and the `add`:

    `i0_rs2_e3_final` = `i0_rs2_bypass_data_e3` = 0x00000001
- **Commit stage ALU**
  - **Cycle *i+1*:** the `add` operation is recomputed using the correct values:

    `out` = `a_ff` + `b_ff` = 0x00000001 + 0x00000001 = 0x00000002
- **3:1 multiplexer**
  - **Cycle *i+1*:** The Secondary ALU's output is selected (`exu_i0_result_e4`). (When no dependency exists, `i0_result_e4` is selected.)

RVfpga Lab 15: Tasks – Sample

- **TASK**: Remove all `nop` instructions in the example from Figure 2. Draw a figure similar to Figure 3 for two consecutive iterations of the loop, then analyse and confirm that the figure is correct by comparing it to a Verilator simulation, and finally compute the IPC by using the Performance Counters while executing the program on the board.
- **TASK**: In the example from Figure 2, remove all `nop` instructions and move the `add t6,t6,-1` instruction after the `add t3,t3,t4` instruction, and then re-examine the program both in simulation and on the board. In this reordered program, the two dependent add instructions (`add t4,t4,t5` and `add t3,t3,t4`) arrive at the Decode stage in the same cycle, and this has an impact in performance. Explain the impact of these changes, using both simulation and execution on the board.
- **TASK**: Compare the equations for the 10:1 multiplexer in the Forwarding Logic with the ones explained for the pipelined processor from DDCARV.
- **TASK**: Remove the `nop` instructions in the example from Figure 11 and obtain the IPC using the HW Counters.
- **TASK**: Disable the Secondary ALU as explained in Lab 11 and analyse the example from Figure 11 both with a Verilator simulation and with an execution on the board.

RVfpga Lab 15: Exercises – Sample

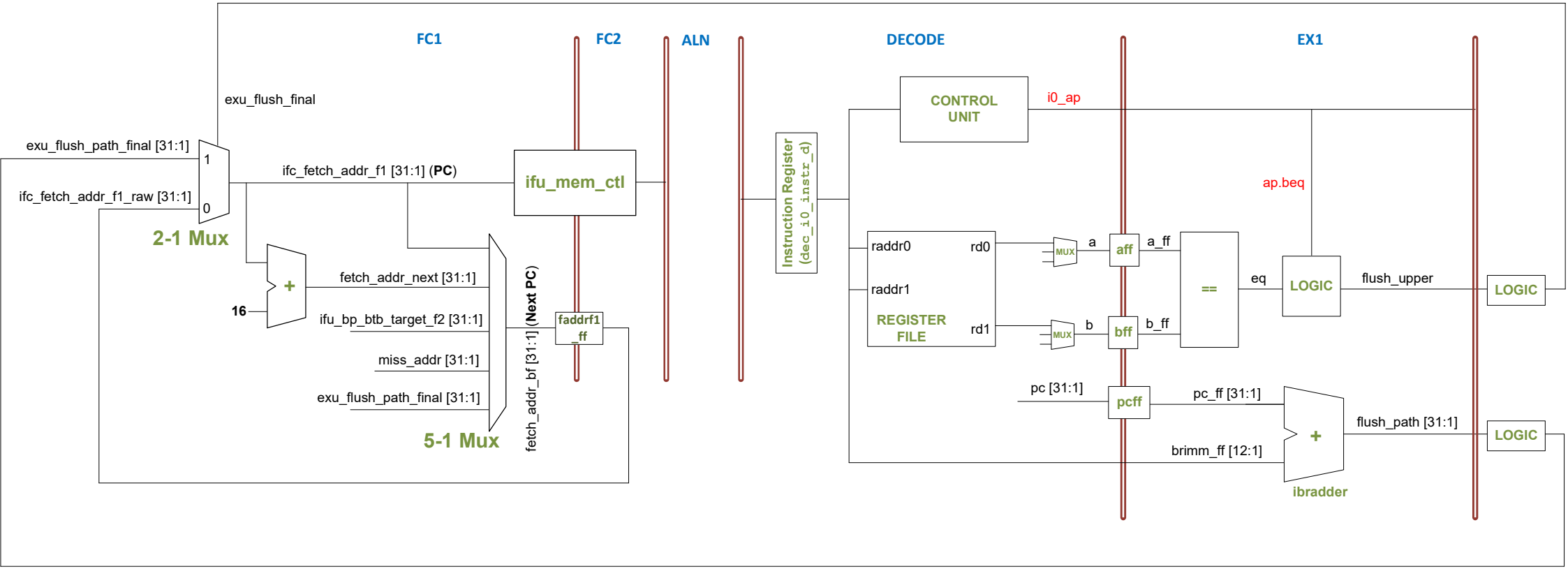
- **Exercise 1**. Modify the program used in Section 3 by adding an extra arithmetic-logic instruction that depends on the result of the `add` instruction. Analyse the Verilator simulation and explain how data hazards are handled for the new A-L instruction. Then remove all nop instructions and analyse the results provided by the HW counters.
- **Exercise 2**. Analyse the same situation as the one described in Section 3 for a mul instruction followed by an add instruction that uses the result of the multiplication. In the program from Figure 11 you can simply substitute the lw for a mul that writes to register t1.
- **Exercise 5**. In the program from Section 2.C of Lab 14, replace instruction `add x1, x1, 1` with `add x28, x1, 1`. This introduces a WAW hazard between the modified add instruction and the non-blocking load at the beginning of the loop (`lw x28, (x29)`). Analyse in simulation how this hazard is handled in SweRV EH1, for which you can look at the value of signal `wen2` in the Register File. Try to understand how this signal is computed in the Control Unit (module `dec`).
- **Exercise 7.** In the program from Section 2.C of Lab 14, replace instruction `add x1, x1, 1` with `add x1, x28, 1`, and instruction `add x7, x7, 1` with `add x28, x7, 1`. This causes both a RAW and a WAW hazard to occur. Analyse in simulation how these two hazards are handled.
- **Exercise 8 - Store to Load Forwarding**: This is a very interesting situation that we have not analysed in this lab and that you will analyse in this exercise.

Lab 16:
Control Hazards and Branch Instructions

RVfpga Lab 16: Control Hazards & Branches

- Branch instructions **calculate the address of the next instruction** after its fetch.
- Control hazards may:
  - **Stall** the pipeline until next instruction address is calculated, or
  - **Predict** whether the branch will be taken and fetch instructions from the predicted path.
- SweRV EH1 has two possible branch predictors (BPs) :
  - **Naïve Branch Predictor:** always predicts branch not taken. Has poor performance but at no hardware cost.
  - **Gshare Branch Predictor:** offers higher performance at the cost of extra hardware.
- This lab analyzes the execution of a `beq` instruction using both the naïve and Gshare BP.

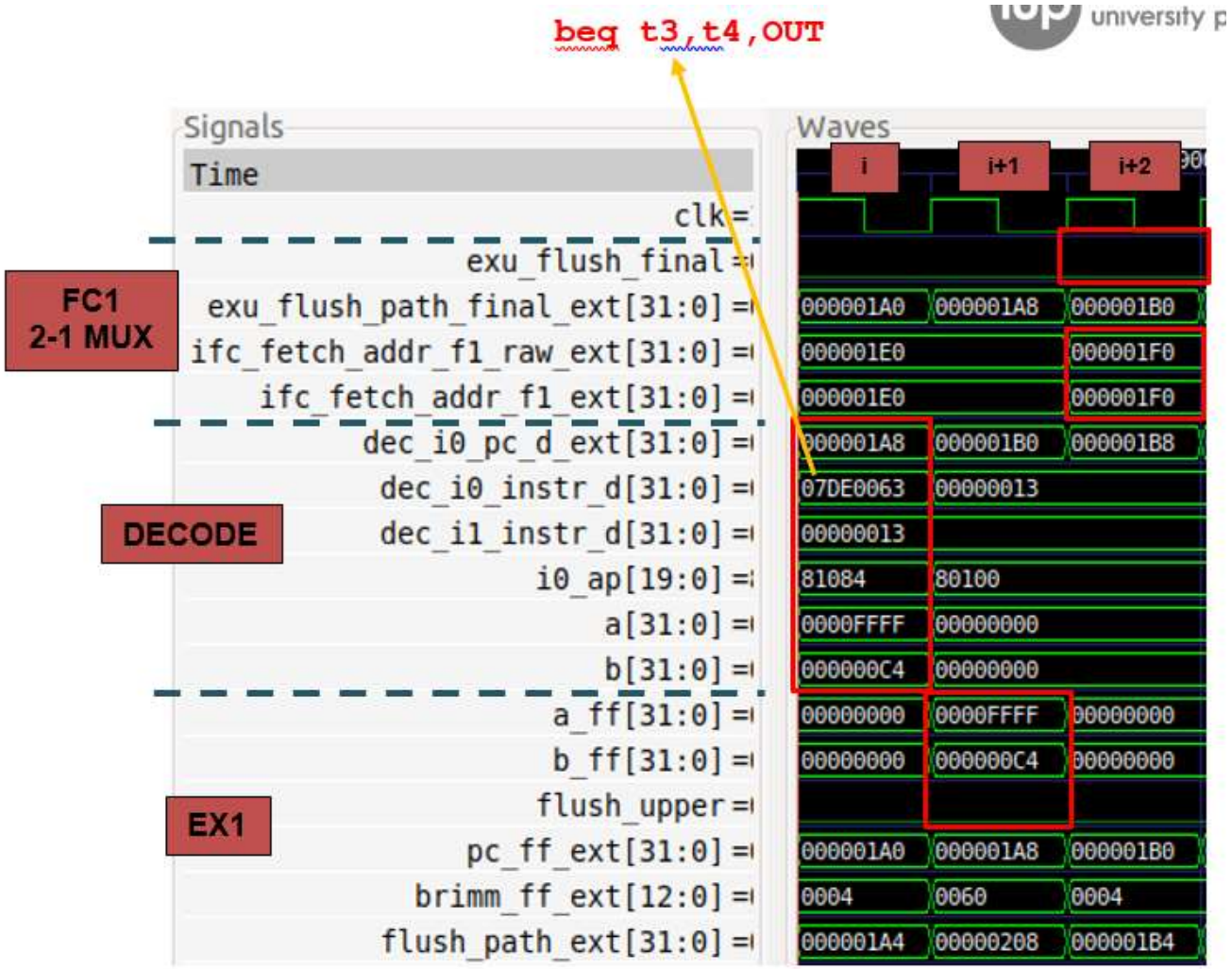
RVfpga Lab 16: Execution of a beq Instruction and PC Calculation



RVfpga Lab 16: Execution of a beq Instruction and PC Calculation – Example

```
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: Execution of First beq Instruction – Simulation



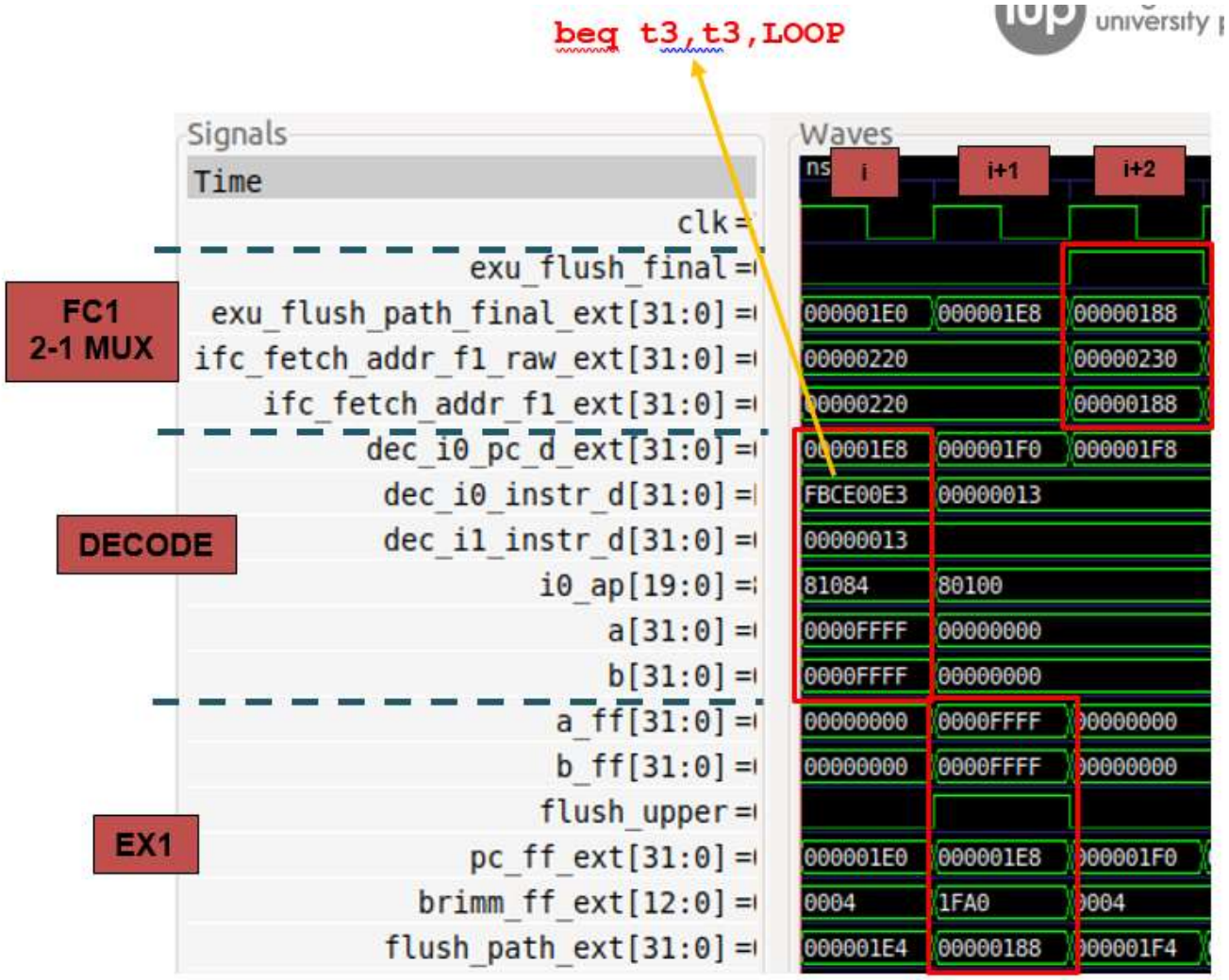
RVfpga Lab 16: Execution of First `beq` Instruction – Analysis

- **Cycle *i* - Decode stage for the beq instruction**: The first `beq` (0x07DE0063) is decoded in Way 0. **Control signals are generated**, the **Register File is read,** and **the branch instruction is routed to the I0 Pipe**. Signals `a` and `b` (0xFFFF and 0xC4, respectively, in this example) contain the inputs to the comparator used in the next stage.

- **Cycle *i+1* - EX1 stage for the beq instruction**: The beq instruction is **executed**. Signals `a_ff` and `b_ff` are compared. The two numbers (0xFFFF and 0xC4) are different, so the branch is not taken. In this example the Gshare predictor is disabled, thus all branches are predicted *not taken* (`i0_ap.predict_nt` = 1). Thus, the branch has been predicted correctly, and the pipeline is **not** flushed (`flush_upper` = 0).

- **Cycle *i+2* - FC1 stage**: Given that the branch was predicted and resolved as not taken, fetching simply continues sequentially. Notice that `exu_flush_final` = 0 and `ifc_fetch_addr_f1_ext[31:0]` = `ifc_fetch_addr_f1_raw_ext[31:0]` = 0x000001F0. This address points to the next sequential 128-bit bundle of instructions.

RVfpga Lab 16: Execution of Second beq Instruction – Simulation



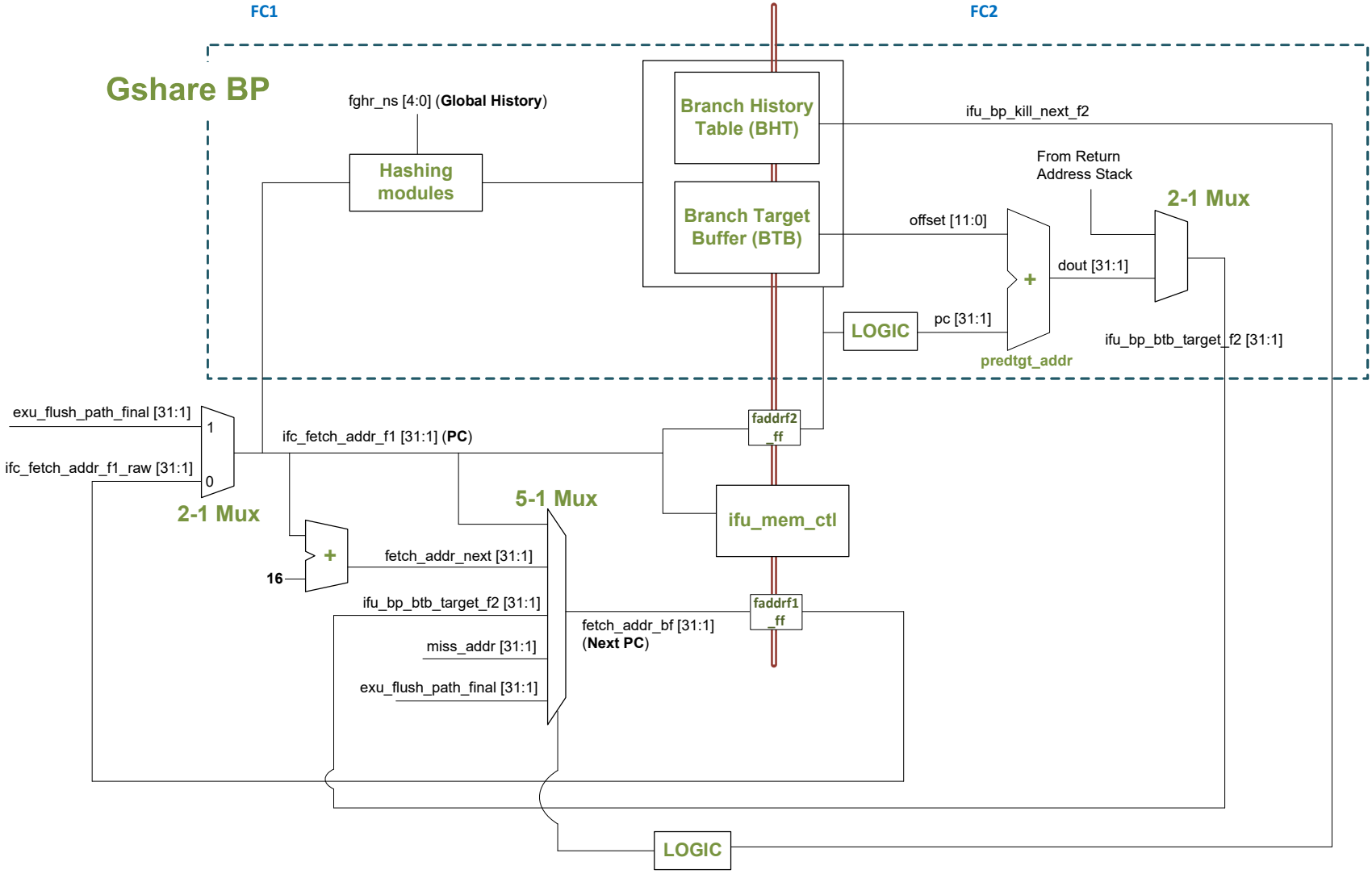
RVfpga Lab 16: Execution of Second `beq` Instruction – Analysis

- **Cycle *i* - Decode stage for the beq instruction**: The second `beq` (0xFBCE00E3) is decoded in Way 0. **Pipeline control signals are generated**, the **Register File is read**, and **the branch instruction is routed to the I0 Pipe**. Signals `a` and `b` (0xFFFF for both of them, in this example) contain the inputs to the comparator used in the next stage.

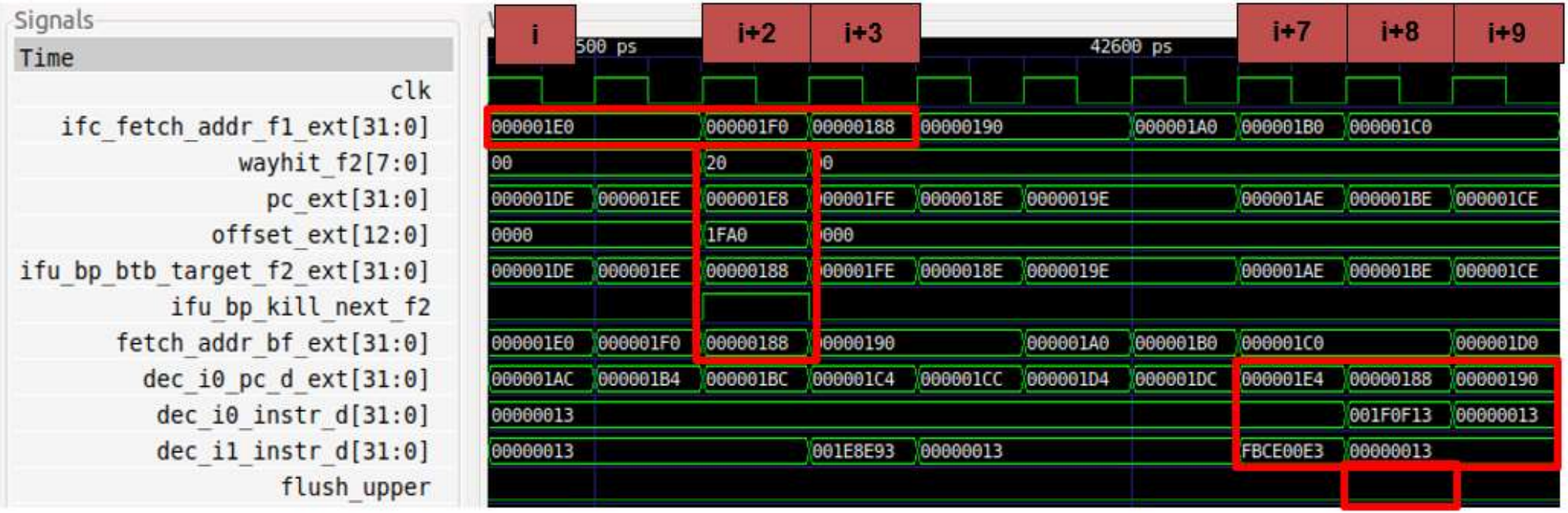
- **Cycle *i+1* - EX1 stage for the beq instruction**: The beq instruction is **executed**. Signals `a_ff` and `b_ff` are compared. The two numbers are equal, so the branch is taken. However, the Naïve BP predicts all branches as *not taken* (`i0_ap.predict_nt` = 1). So, the branch has been mispredicted, and the fetched instructions must be flushed (`flush_upper` = 1)..

- **Cycle *i+2* - FC1 stage**: Execution must continue at the branch target address. `exu_flush_final` = 1 and `ifc_fetch_addr_f1_ext` = `exu_flush_path_final_ext` = 0x00000188. This address corresponds to the branch target address, which is the address of the first instruction of the loop.

RVfpga Lab 16: The Gshare Branch Predictor used by SweRV EH1



RVfpga Lab 16: The Gshare Branch Predictor for the Second beq



RVfpga Lab 16: The Gshare Branch Predictor for the second beq

- **Cycle *i***: The address of the bundle that contains the second branch is provided to the Instruction Cache: `ifc_fetch_addr_f1_ext` = 0x000001E0. The Branch Target Buffer (BTB) is read using this address.
- **Cycle *i+2***: A hit takes place in the BTB: `wayhit_f2` = 0x20. The address of the branch (`pc_ext` = 0x000001E8) is added to the offset provided by the BTB (`offset_ext` = 0x1FA0, which is a negative value), which results in the predicted target address (`ifu_bp_btb_target_f2_ext` = 0x00000188). Given that the branch is predicted taken by the BHT (`ifu_bp_kill_next_f2` = 1), it is used as the Next Fetch PC (`fetch_addr_bf_ext` = 0x00000188).
- **Cycle *i+3***: The Fetch Address is the predicted target address of the branch, which was computed in the previous cycle: `ifc_fetch_addr_f1_ext` = 0x00000188.
- **Cycle *i+7***: The branch is decoded in Way 1 (`dec_i1_instr_d` = 0xFBCE00E3).
- **Cycle *i+8***: The branch executes. The prediction was correct, so no flush needs to be triggered (`flush_upper` = 0).
- **Cycle *i+9***: Execution continues normally through the branch target address given that the prediction was correct.

RVfpga Lab 16: Tasks and Exercises - Sample

- **TASK**: In Lab 15, we analysed how RAW data hazards are resolved in the Commit stage by means of the Secondary ALUs. Similar to the A-L instructions that we studied in that lab, a conditional branch instruction can have a RAW data hazard with a previous multi-cycle operation that must be resolved at commit time. If the branch is determined to have been mispredicted, the pipeline must be flushed and redirected from the Commit stage. Analyse this situation using a slightly modified version of the program from Figure 2.
- **TASK**: In the example from Figure 2, remove all the nop instructions and analyse the simulation. Then compute the IPC with the Performance Counters by executing the program on the board. Enable the branch predictor used in SweRV EH1 (by commenting out the two initial instructions in Figure 2) and analyse the simulation and the execution on the board. Compare the two experiments and explain the results.
- **TASK**: Explain how the Global History Register is updated at module `ifu_bp_ctl`.
- **Exercise 1**) Implement a Bimodal Branch Predictor and compare its performance to the Gshare BP.
- **Exercise 2**) This exercise is based on exercise 4.25 from the book “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([HePa]).

Lab 17:
Superscalar Execution

RVfpga Lab 17: Introduction

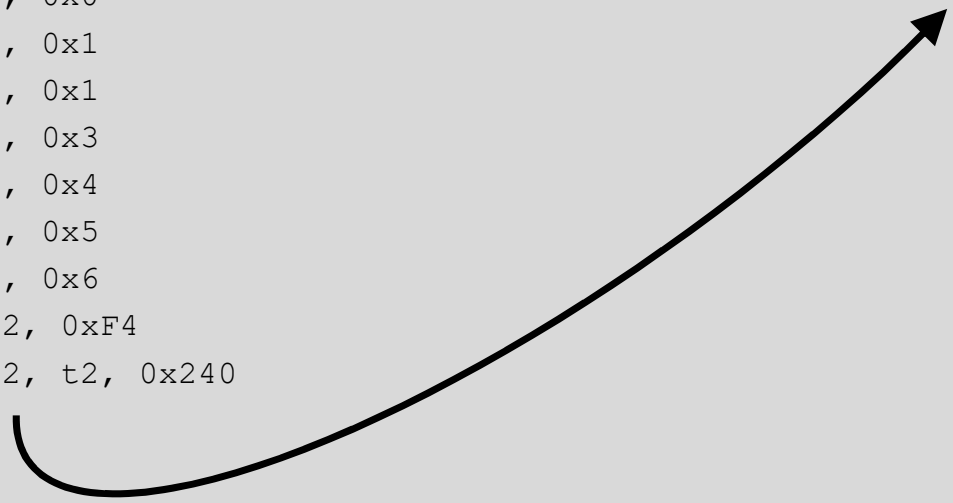
- Western Digital's SweRV EH1 processor is a 9-stage pipelined 32-bit **2-way superscalar** core.
- A superscalar processor contains **multiple copies of the datapath** hardware to execute multiple instructions simultaneously.
- The **latency** of executing a single instruction is the same as a scalar processor, but the processor can execute and commit more instructions per cycle, thus **improving its throughput**.
- SweRV EH1 does not include support for dynamic instruction scheduling with out-of-order execution, except for the non-blocking loads. However, it is possible to statically reorder the code in order to better exploit the resources, including the two ways of the pipeline.

RVfpga Lab 17: Introduction

- SweRV EH1 is a **2-way superscalar** processor.
  - It fetches, executes, and commits up to **two instructions per cycle**.
  - The **multi-ported register file** reads up to four source operands and writes two values back in each cycle (plus one more value coming from a non-blocking load, as analysed in Lab 15).
  - Each way contains **independent pipes**: two Integer pipes, one Multiply pipe, one Load-Store pipe, and one non-pipelined Divider.
- Ideally, in a 2-way superscalar processor, throughput (IPC) doubles compared to a single-issue processor. Unfortunately, actual programs typically exhibit performance improvements of 1.3x-1.5x when going from 1- to 2-way processors; however, adding the second way requires much more hardware.
- In this lab, we analyse two simple programs, comparing the behaviour when using single-issue and dual-issue configurations of SweRV EH1.

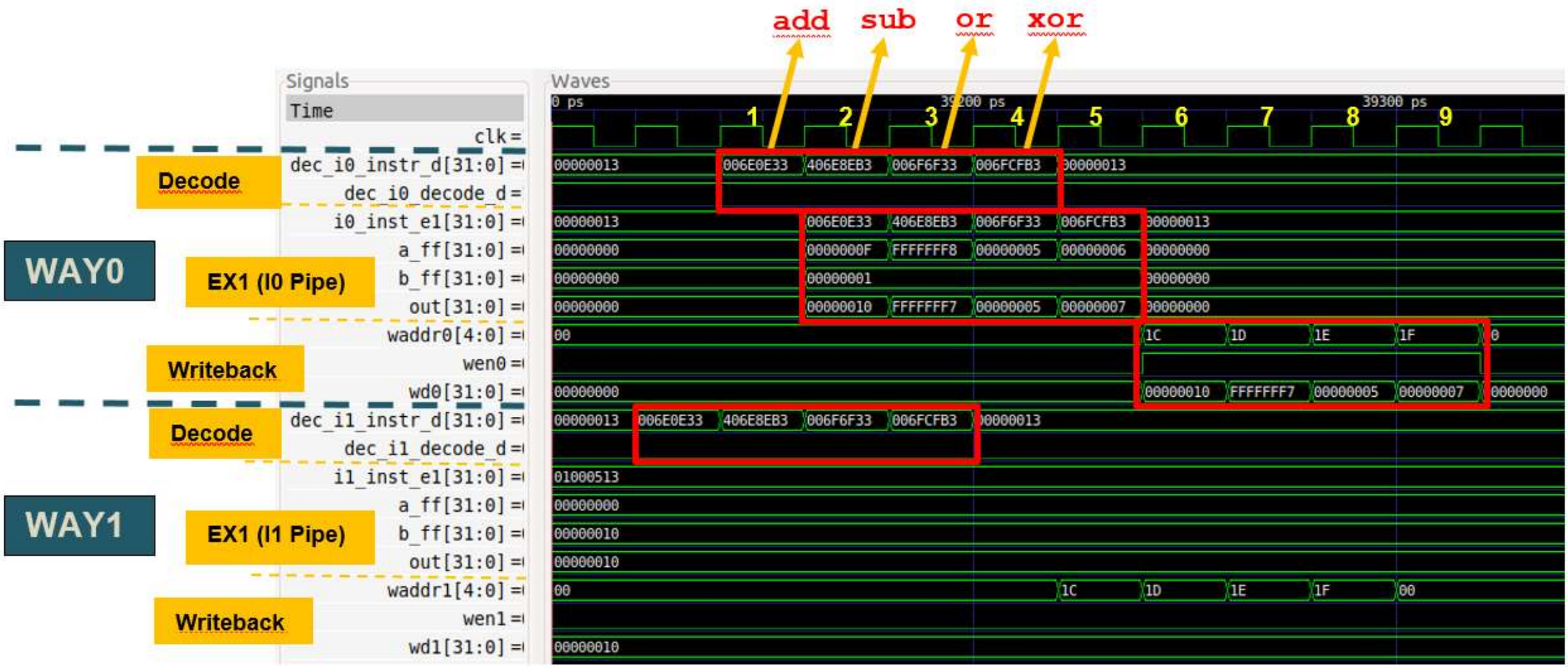
RVfpga Lab 17: Four Independent A-L Instructions – Example – Single-Issue

```
.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: Four Independent A-L Instructions – Simulation – Single-Issue

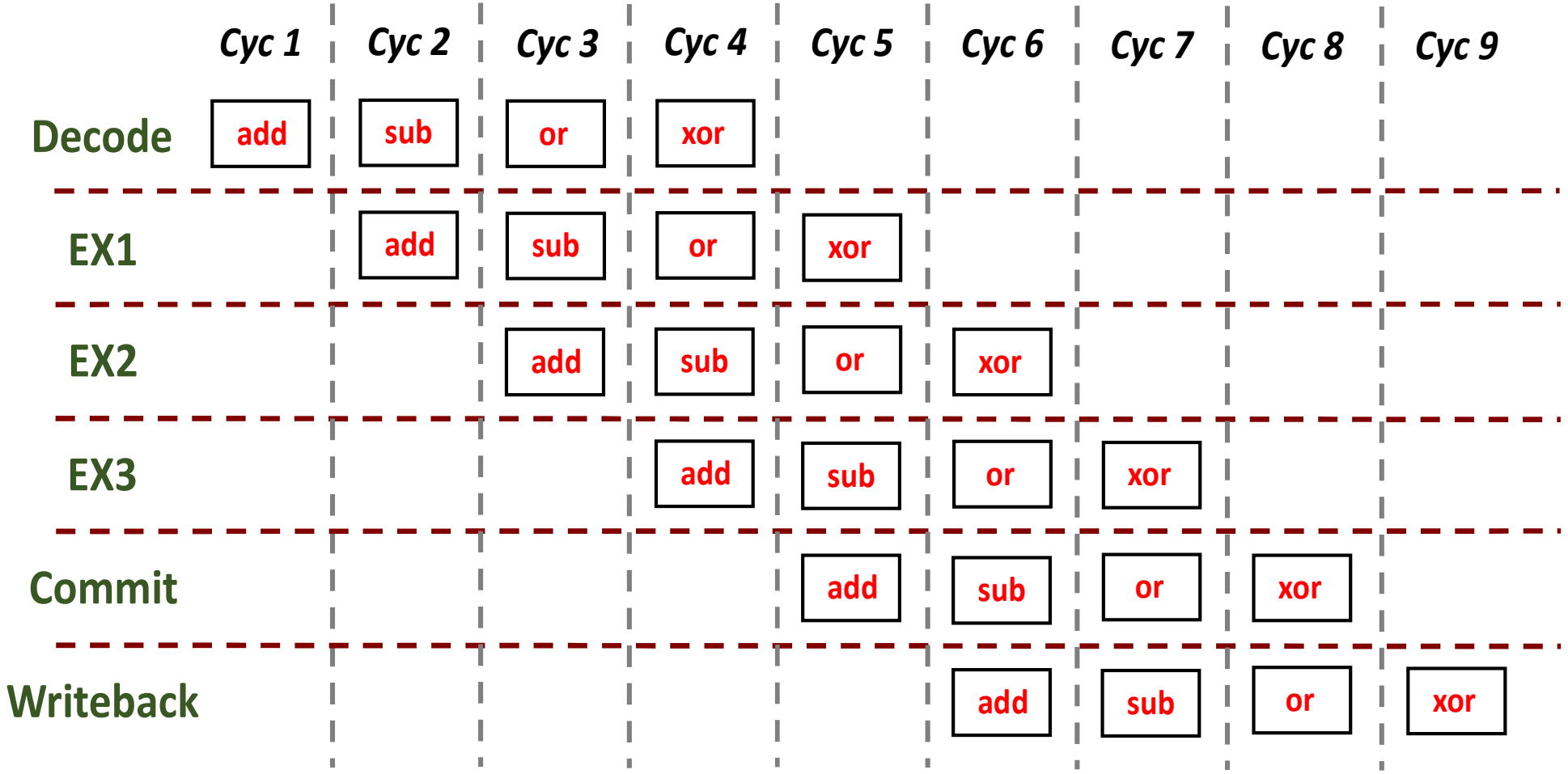


RVfpga Lab 17: Four Independent A-L Instructions – Simulation – Single-Issue

- The instructions are received in both ways at decode time, but they are only sent to execution in Way 0, because Way 1 is disabled.
  - **Way 0**:
    - Signal `dec_i0_decode_d` is always 1 in our example; specifically, it is 1 for the four AL instructions under analysis.
    - The instruction in the Decode Stage **is propagated** to the I0 Pipe (`i0_inst_e1[31:0]`)
  - **Way 1**:
    - Signal `dec_i1_decode_d` is always 0 in our example; specifically, it is 0 for the four AL instructions under analysis.
    - The instruction at the Decode Stage **is NOT propagated** (`i1_inst_e1[31:0]`) to the Execution Stage.

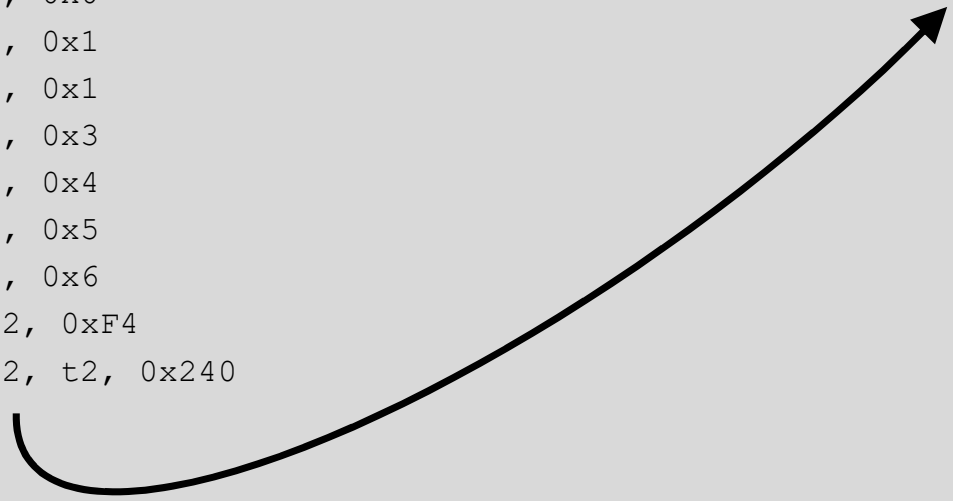
- Accordingly, only the ALU from the I0 Pipe is used (see signals `aff`, `bff` and `out` in both ways) and only write port 0 of the Register File is used (see signals `waddr`, `wen` and `wd` in both ways).

RVfpga Lab 17: Four Independent A-L Instructions – Diagram – Single-Issue



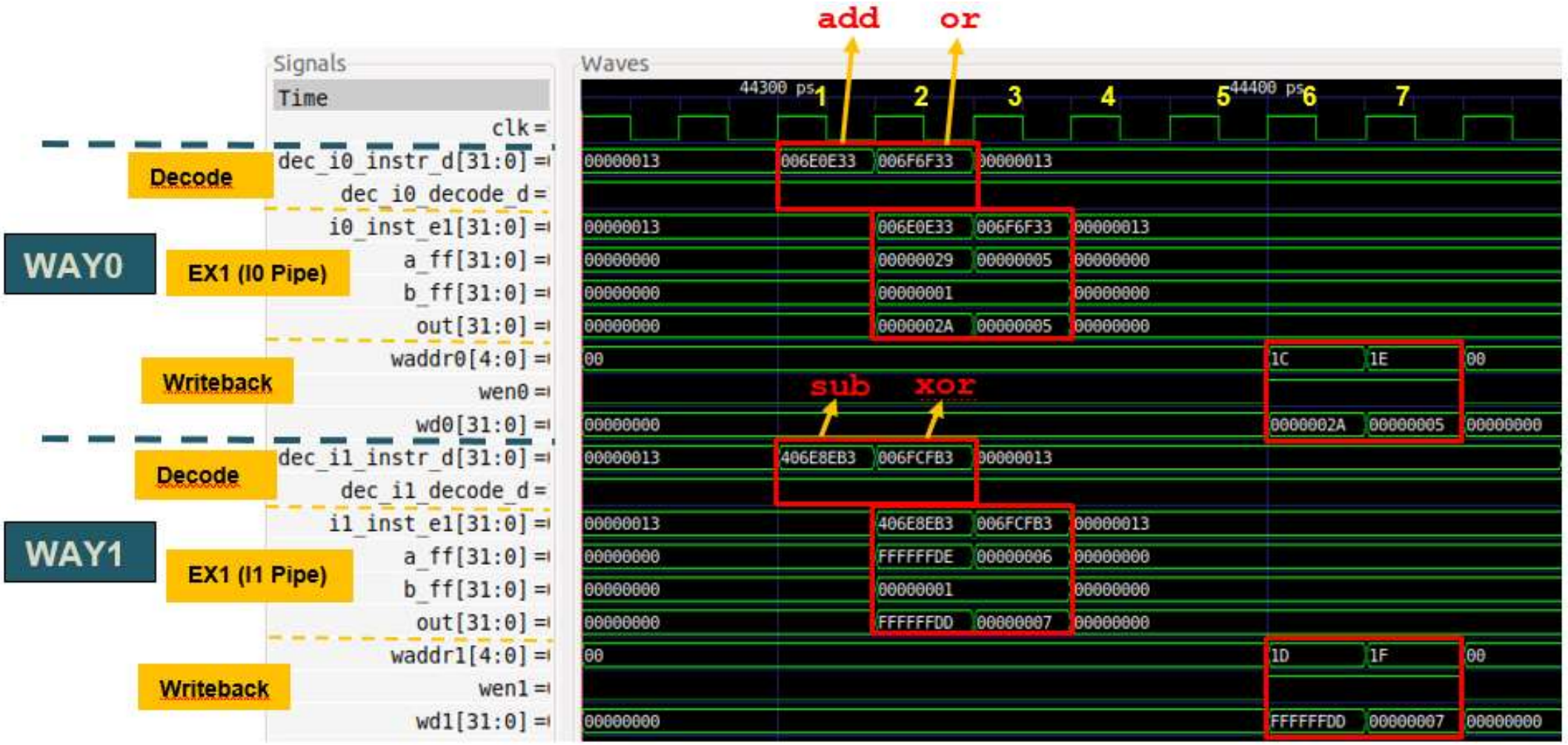
RVfpga Lab 17: Four Independent A-L Instructions – Example – Dual-Issue

```
.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
```

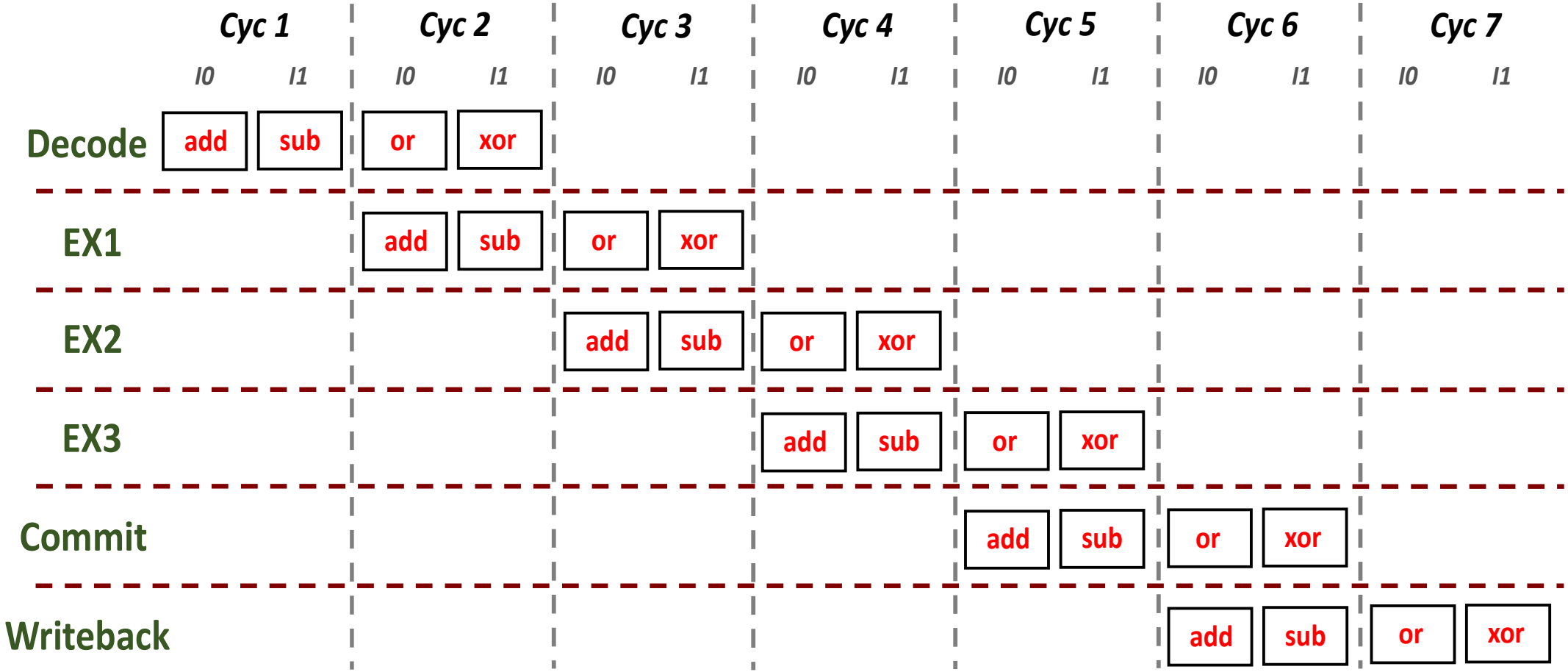
RVfpga Lab 17: Four Independent A-L Instructions – Simulation – Dual-Issue



RVfpga Lab 17: Four Independent A-L Instructions – Analysis – Dual-Issue

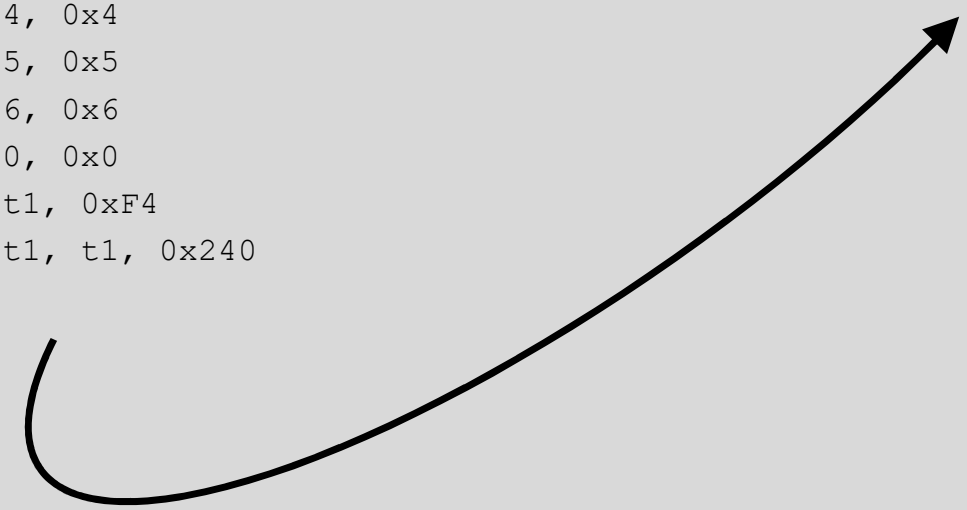
- In each cycle, two instructions are decoded, one in each way, and two instructions are sent to the Execute stages, one through the I0 pipe and the other through I1.
  - **Way 0**:
    - Signal `dec_i0_decode_d` is always 1 – being true for two of the four A-L instructions of our example (the other two A-L instructions are decoded in Way 1).
    - The instruction in the Decode stage is propagated to the I0 Pipe (`i0_inst_e1[31:0]`).
  - **Way 1**:
    - Signal `dec_i1_decode_d` is always 1 – being true for two of the four A-L instructions of our example (the other two A-L instructions are decoded in Way 0).
    - The instruction in the Decode stage is propagated to the I1 Pipe (`i1_inst_e1[31:0]`).
- Thus, the ALUs in both pipes (I0 and I1) are used (see signals `aff`, `bff`, and `out` in both ways), and both Register File write ports are used (see signals `waddr`, `wen` and `wd` in both ways).

RVfpga Lab 17: Four Independent A-L Instructions – Diagram – Dual-Issue



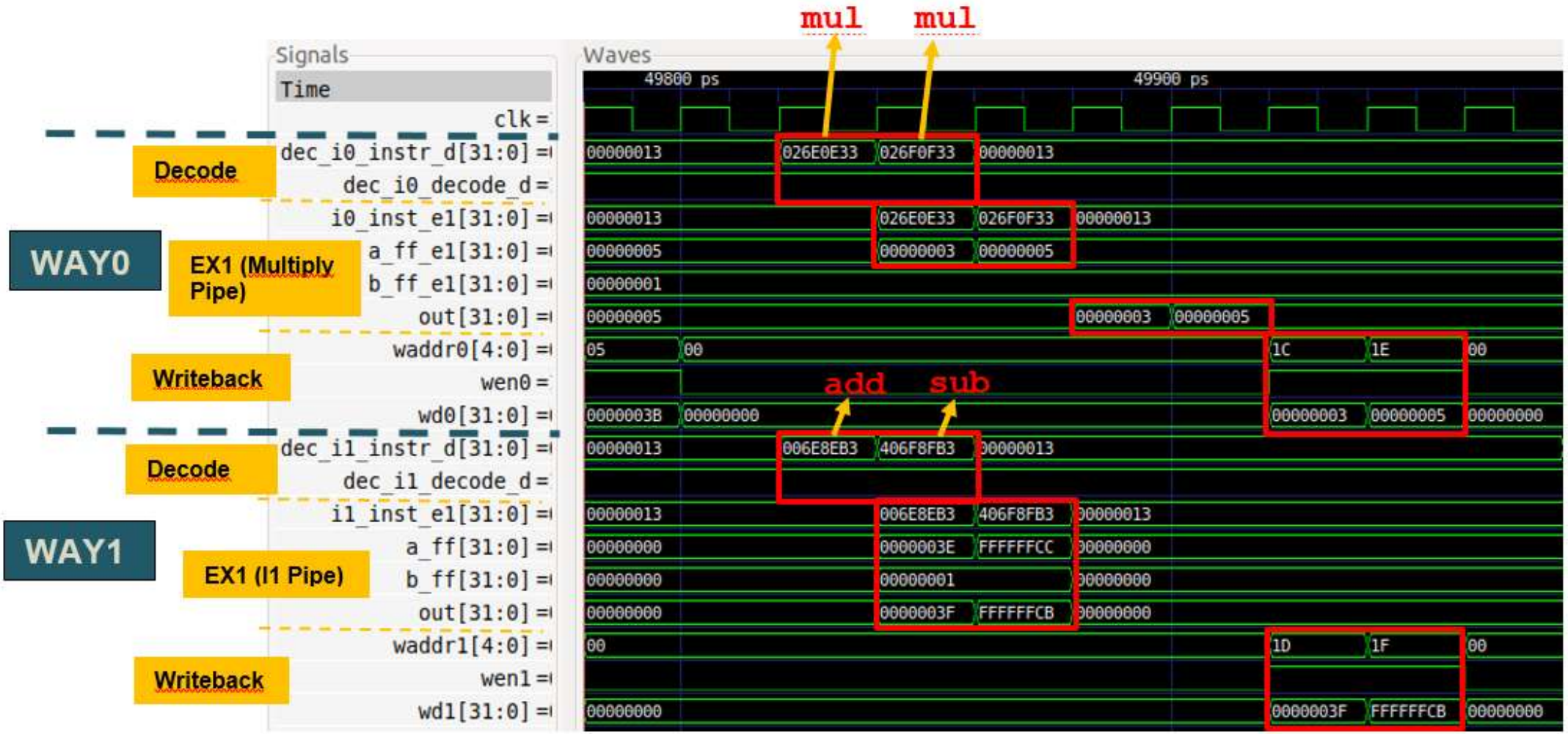
RVfpga Lab 17: Two `mul` Instructions Interleaved with Two A-L Instructions – Example – Dual-Issue

```
.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: Two `mul` Instructions Interleaved with Two A-L Instructions – Simulation – Dual-Issue



RVfpga Lab 17: Two `mul` Instructions Interleaved with Two A-L Instructions – Analysis – Dual-Issue

- The instructions are received in both ways at decode time and are sent to the execution stages in both ways.
  - **Way 0**:
    - Signal `dec_i0_decode_d` is always 1 – for two of the four instructions analysed in our example (the other two instructions are decoded in Way 1).
    - The instruction in the Decode stage is sent to the Multiply pipe (`i0_inst_e1[31:0]`)
  - **Way 1**:
    - Signal `dec_i1_decode_d` is always 1 – for two of the four instructions analysed in our example (the other two instructions are decoded in Way 1).
    - The instruction in DECODE (`dec_i1_instr_d[31:0]`) is propagated to the I1 Pipe (`i1_inst_e1[31:0]`)
- Thus, the ALU from the I1 pipe and the Multiplier are used (see signals `a_ff_e1`, `b_ff_e1`, and `out` and signals `a_ff`, `b_ff`, and `out`), and both Register File write ports are used (see signals `waddr`, `wen`, and `wd` in both ways).

RVfpga Lab 17: Tasks and Exercises - Sample

- **TASK**: Remove all the `nop` instructions within the body of the loop from Figure 2. Repeat the simulation from Figure 3. What is the expected IPC for this program? Execute the program on the board and verify that the IPC obtained is the one that you expected.
- **Exercise 2**) Analyse the differences between the (dual-issue) SweRV EH1 processor and the example superscalar processor proposed in Section 7.7.4 of the textbook by S. Harris and D. Harris, "Digital Design and Computer Architecture: RISC-V Edition" [DDCARV] (shown in Figure 1 for convenience).
- **Exercise 3**) Analyse the program from Figure 7.70 in Section 7.7.4 of DDCARV, which is provided in a PlatformIO project. Run the program on SweRV EH1, both in simulation and on the board (for the latter remove the nop instructions). Explain the results. If necessary, reorder the program trying to obtain the optimal IPC. Next, disable the dual-issue execution as explained in this lab – and in SweRVref.docx (Section 2). Compare the simulation and the results obtained on the board when compared to when the dual-issue feature is enabled.
- **Exercises 5, 6** and **7**) These exercises are based on exercises from the books:
  - "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:
Adding New Features:
Instructions and Counters

RVfpga Lab 18: Adding Instructions & Features

- In this lab, you will apply the knowledge acquired in previous labs to modify the SweRV EH1 processor to add the following new features:
  - **Add A-L instructions:** Add Arithmetic-Logic instructions from the new bit manipulation extension available in the RISC-V architecture.
  - **Add floating-point instructions:** Add three floating point instructions: add, multiply, and divide. Then use them to compute the bisection algorithm.
  - **Add counter:** Add a new hardware counter that counts the number of I-Type instructions executed.
- In some of these exercises we guide you through the process of modifying the core, and in others you will figure out on your own what needs to be done.

Lab 19:
Instruction Cache

RVfpga Lab 19: Introduction

- This lab describes and explores the memory system of the RVfpga System. RVfpga's Memory System has the following elements:
  - External DDR Main Memory
  - Cache for instructions (I$)
  - Two Scratchpad memories (also called closely-coupled memories), one for data (DCCM) and one for instructions (ICCM). The ICCM is disabled in the default system.
- In this lab, we first describe how data are read from and written to the DDR External Memory, and then we delve into the operation and management of the I$ available in the RVfpga System.

RVfpga Lab 19: Data read and write to Memory – Example

```
.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
```

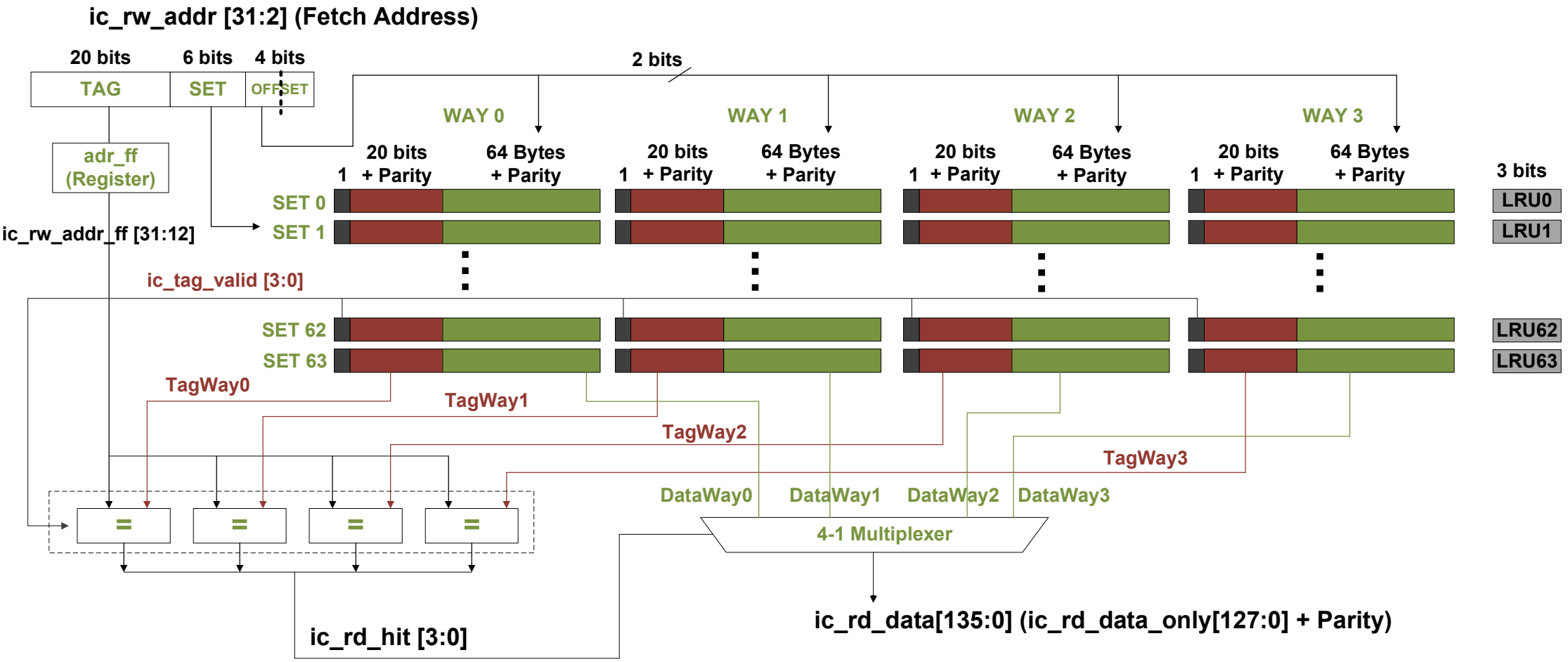
```
REPEAT:
lw t3, (t4)
add t3, t3, t5
sw t3, (t4)
add t4, t4, 4
bne t4, t6, REPEAT
```

RVfpga Lab 19: Data read and write to Memory – Simulation

The example illustrates a program that includes a load instruction followed by a store instruction

- Cycle *i* – *i+8*: The processor reads data from the DDR External Memory (yellow square) into `t3`, through the bus.
- Cycle *i+16* – *i+21*: The processor writes the value of `t3` to the DDR External Memory (red square), through the bus.

RVfpga Lab 19: I$ Configuration and Operation



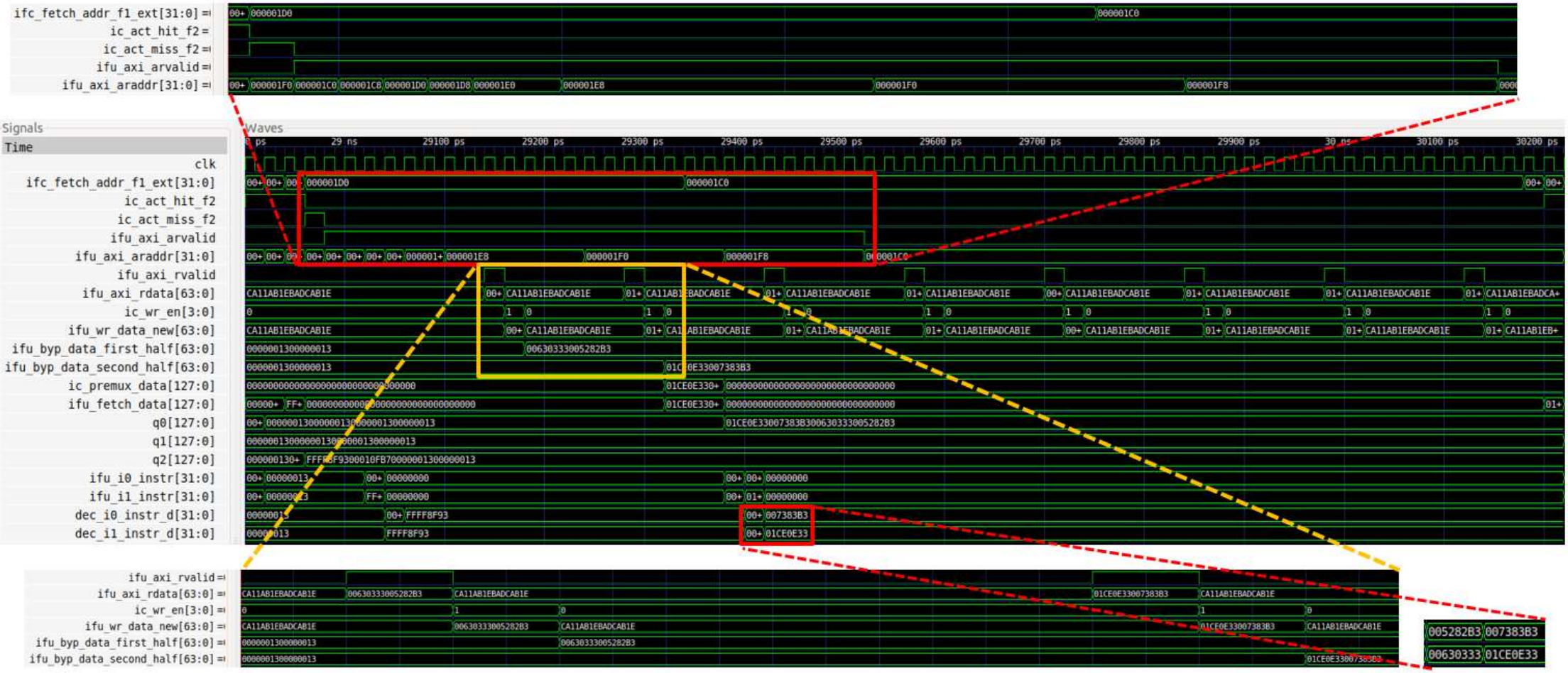
RVfpga Lab 19: I$ Miss and Hit Management – Example

```
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: I$ Miss Management - Simulation



RVfpga Lab 19: I$ Miss Management - Analysis

- The simulation shows the fetch of the 16 `add` instructions the first time they are executed. Given that these instructions are not in the I$ yet, a miss is triggered in the I$ and the instructions must be copied from the DDR External Memory into the I$.
  - An I$ miss is signalled at around 29ns (`ic_act_miss_f2` = 1), which triggers the request of the block through the AXI bus (`ifu_axi_arvalid` = 1).
  - The eight 64-bit chunks that make up the target block are requested sequentially through the AXI bus.
    - Signal `ifu_axi_arvalid` goes high for 27 cycles. This signal indicates that the channel is signalling valid read address and control information.
    - During these 27 cycles where `ifu_axi_arvalid` = 1 the initial addresses of the eight 64-bit chunks are provided sequentially through the AXI bus using signal `ifu_axi_araddr`, which provides the 8 addresses that must be read from the DDR Memory.

RVfpga Lab 19: I$ Miss Management - Analysis

- The middle figure shows the eight 64-bit chunks arriving sequentially to the processor through the AXI bus in signal `ifu_axi_rdata`.
  - Signal `ifu_axi_rvalid`, which indicates that the channel is signalling the required read data, goes high for one cycle every 7 cycles.
  - Each of the eight 64-bit chunks (each containing two instructions) is provided in signal `ifu_axi_rdata`.
- The two bottom figures show that each of the eight 64-bit chunks is written into the I$ right after their arrival to the cache controller.
- Finally, you can see that the four instructions are bypassed from the I$ controller to the pipeline so that it can restart execution as soon as possible after the I$ miss. Several cycles later, the four instructions arrive at the Decode Stage.

RVfpga Lab 19: I$ Hit Management - Simulation

RVfpga Lab 19: I$ Hit Management - Analysis

- In the previous simulation you can see a hit in the I$.
  - **Cycle i:** The address of the first add instruction (`add t0,t0,t0`) is given in signal `ifc_fetch_addr_f1_ext`. This signal is passed to the I$ except for its two least significant bits, which are not needed because instructions are 4-byte (32-bit) aligned. Thus, `ic_rw_addr` = 0x0000070. The Tag and Data Arrays use a subset of the Fetch Address.
  - **Cycle i+1:** The four tags, one per cache way, are in signals `TagWay0-TagWay3`. These are compared to the TAG field of the Fetch Address. In this case, all tags are the same as the TAG field, however only one way (Way 0) is valid (`ic_tag_valid` = 0001), thus a hit is signalled in Way 0: `ic_rd_hit` = 0001. Also, four 128-bit bundles are in signals `DataWay0-DataWay3`: `ic_rd_data_only` = 0x01ce0e33007383b300630333005282b3
  - **Cycle i+2:** The first and second `add` instructions are extracted in the Align stage from buffer q1: `ifu_i0_instr` = 0x005282b3 and `ifu_i1_instr` = 0x00630333
  - **Cycle i+3:** The third and fourth `add` instructions are extracted in the Align stage and, at the same time, the first and second `add` instructions are decoded: `ifu_i0_instr` = 0x007383b3, `ifu_i1_instr` = 0x01ce0e33, `dec_i0_instr_d` = 0x005282b3 and `dec_i1_instr_d` = 0x00630333
  - **Cycle i+4:** Finally, the third and fourth `add` instructions are decoded: `dec_i0_instr_d` = 0x007383b3 and `dec_i1_instr_d` = 0x01ce0e33

RVfpga Lab 19: I$ Replacement Policy

- Most associative caches have a least recently used (LRU) replacement policy. However, tracking the least recently used way becomes complicated, thus approximate LRU policies (usually called Pseudo LRU) are often used and are good enough in practice.
- SweRV EH1 uses an approximate policy called **Binary Tree Pseudo LRU**.
  - It requires N-1 bits per set (which we call LRU State) in an N-way associative cache. This translates into 3 bits per set in SweRV EH1's I$.

Block Replacement

LRU State	Way to replace
x00	Way 0
x10	Way 1
0x1	Way 2
1x1	Way 3

LRU State Updating

Written Way	Next LRU state
Way 0	-11
Way 1	-01
Way 2	1-0
Way 3	0-0

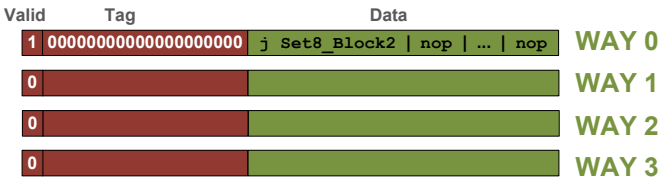
RVfpga Lab 19: I$ Replacement Policy – Example

- The example below accesses five different I$ blocks inside an infinite loop. All five blocks map to the same I$ set: SET = 8.
- The infinite loop contains five `j` (jump) instructions, where each pair of `j` instructions is separated by 1023 `nop`s. The `j` instruction plus the `nop`s occupy 4 KiB, which is equal to the size of each Way in the 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
```

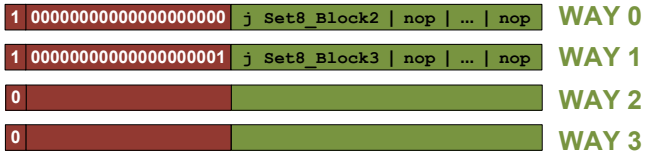
RVfpga Lab 19: I$ Replacement Policy – Evolution of SET 8 of the I$

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



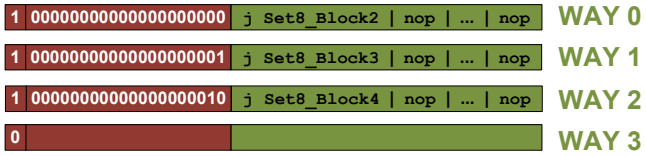
LRU STATE = 011

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



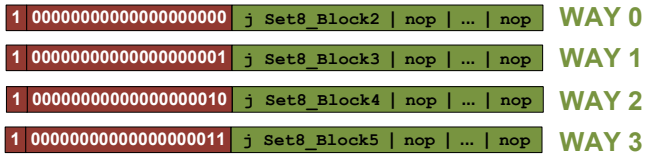
LRU STATE = 001

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



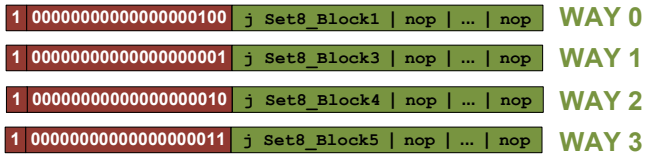
LRU STATE = 100

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



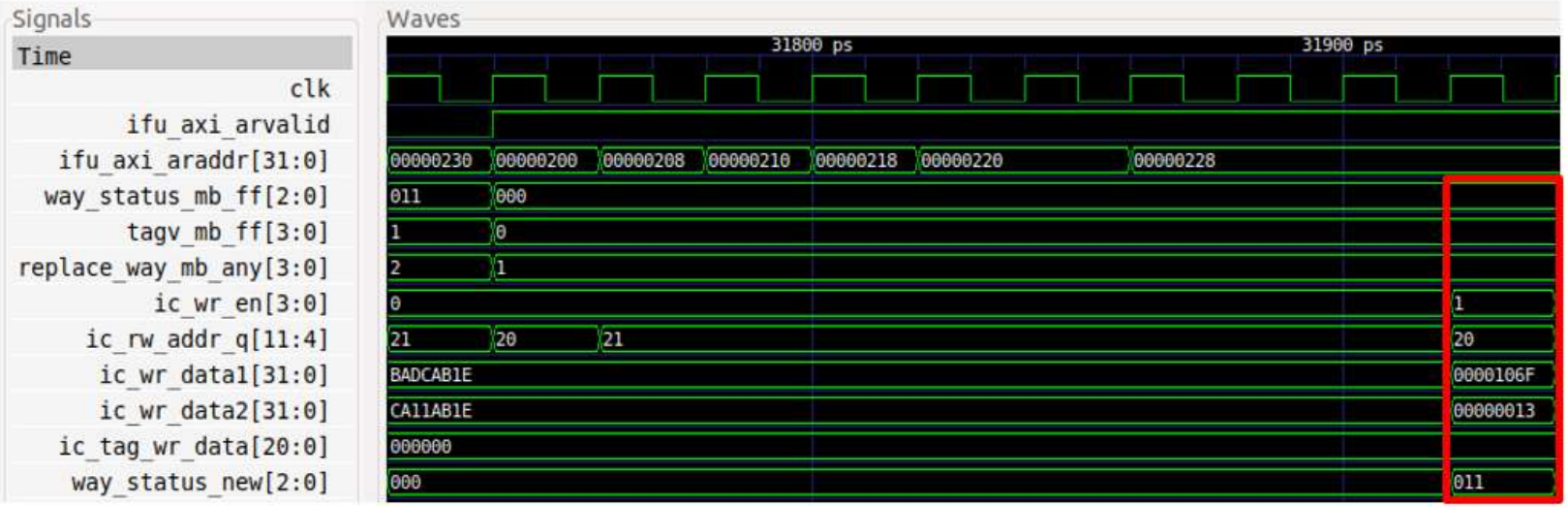
LRU STATE = 000

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



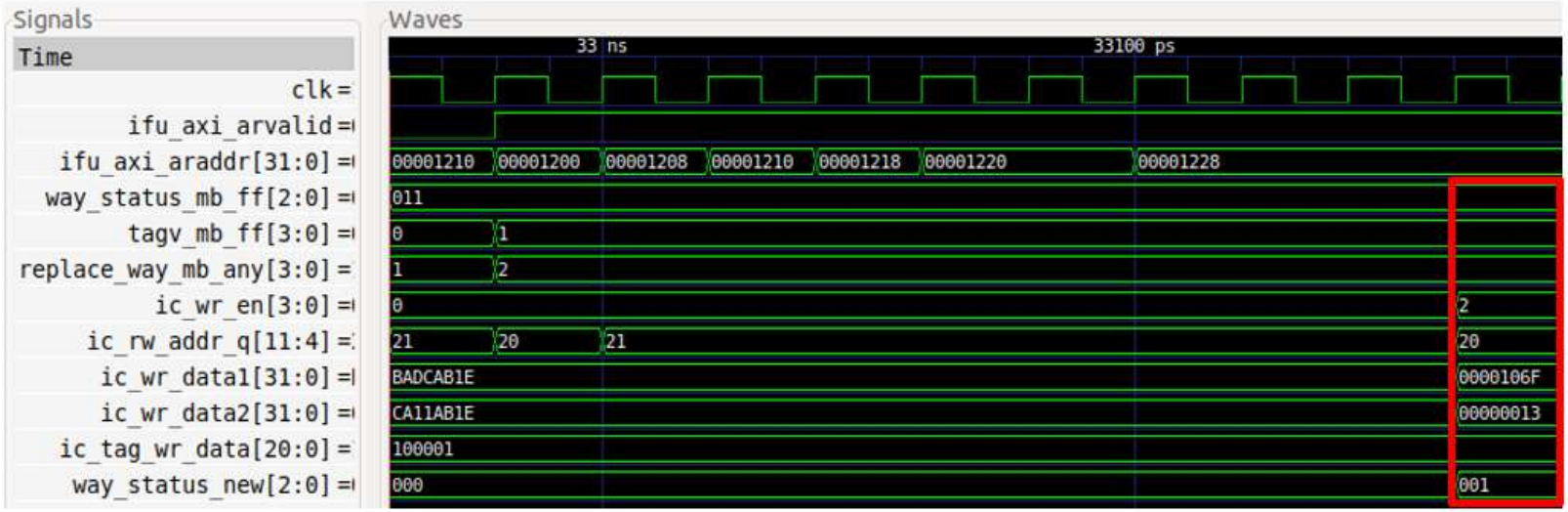
LRU STATE = 011

RVfpga Lab 19: I$ Replacement Policy – 1st Jump



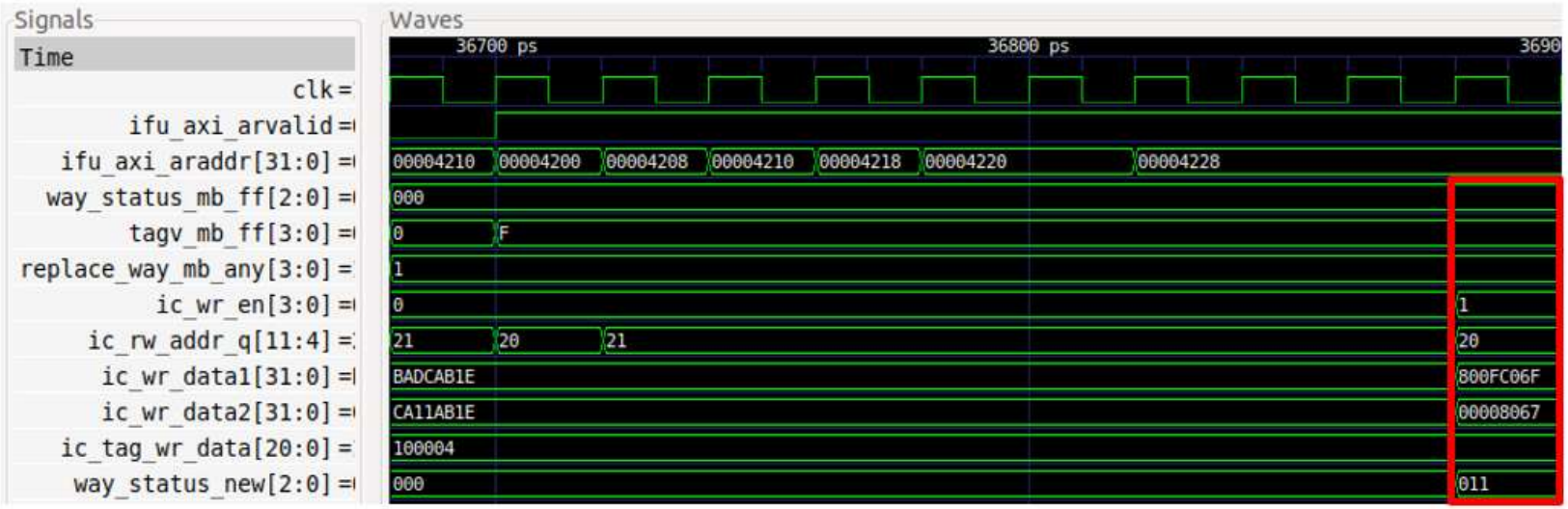
- The first jump's address (0x200) maps to Set 8 of the I$. That set is initially empty, thus, the new block must be written in Way 0: `replace_way_mb_any` = `ic_wr_en` = 0001. The LRU state of Set 8 is updated as follows: `way_status_new` = 011.
- The I$ block is read from the DDR Memory and written into the I$ in 64-bit chunks. The figure illustrates the write of the tag and the two first instructions of the new block into 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$ Replacement Policy – 2nd Jump



- The second jump's address (0x1200) also maps to Set 8 of the I$. Only way 0 is valid in that set: `tagv_mb_ff` = 0001. Thus, the new block must be written in Way 1: `replace_way_mb_any` = `ic_wr_en` = 0010. The LRU state of Set 8 is updated as follows: `way_status_new` = 001.
- The I$ block is read from the DDR Memory and written into the I$ in 64-bit chunks. The figure illustrates the write of the tag and the two first instructions of the new block into 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: I$ Replacement Policy – 5th Jump



- The fifth jump's address (0x4200) also maps to Set 8 of the I$. However, in this case the set is full: `tagv_mb_ff` = 1111. Thus, the new block must be written to Way 1: `replace_way_mb_any` = `ic_wr_en` = 0001. The LRU state of Set 8 is updated as follows: `way_status_new` = 011.
- The I$ block is read from the DDR Memory and written into the I$ in 64-bit chunks. The figure illustrates the write of the tag and the two first instructions of the new block into 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: Tasks and Exercises – Sample

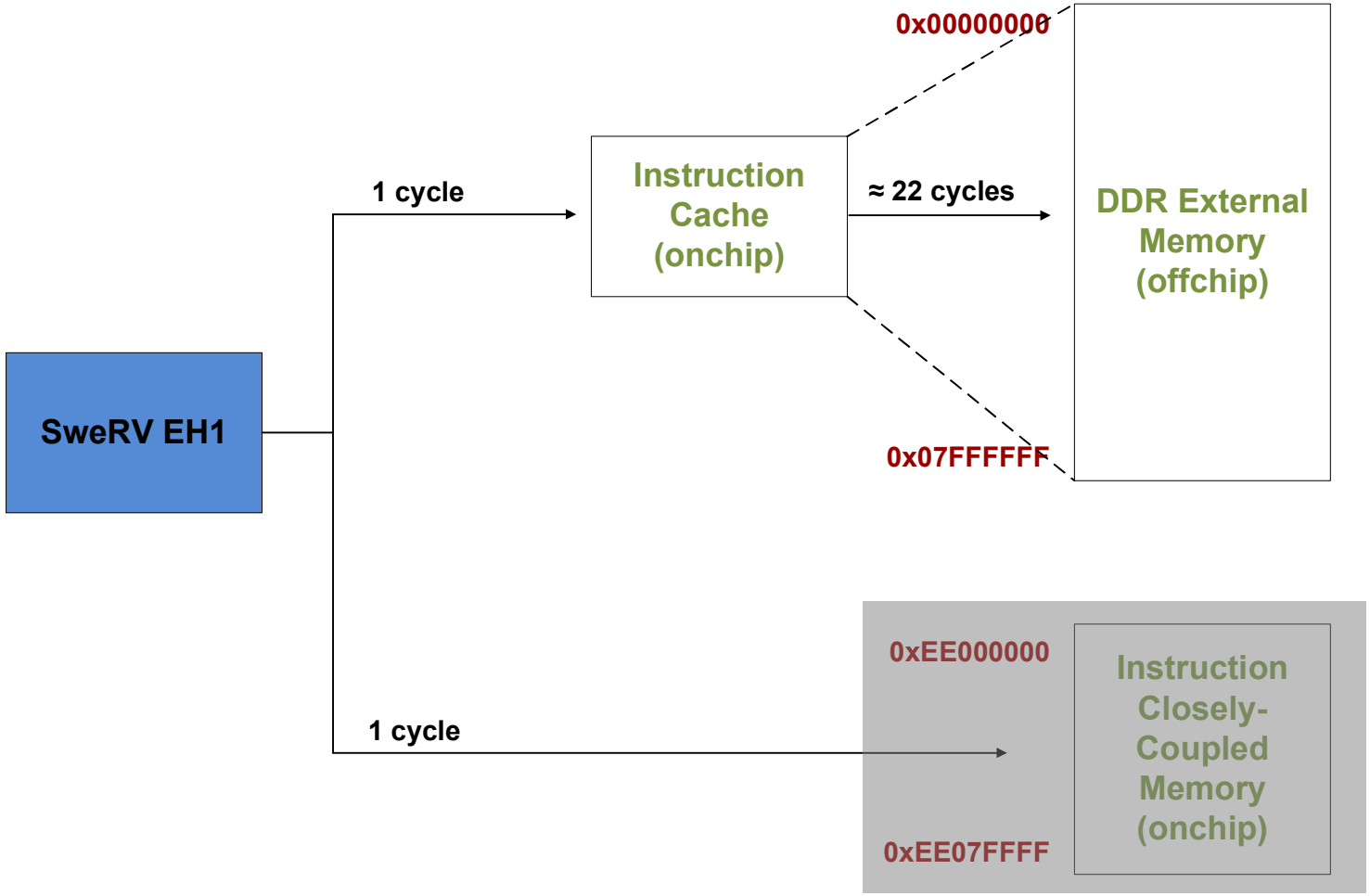
- **TASK**: Using the HW Counters, measure the number of cycles, instructions, loads and stores in the program from Figure 2. How much time in total (both for reading and writing) does it take to access the DDR External Memory?
- **TASK**: Use the example from *[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem* to estimate the DDR External Memory read latency using the HW Counters.
- **TASK**: A quite complex but very interesting exercise is to analyse the Memory Controller used in the RVfpga System. Remember that you can find the modules that make up this controller in folder *[RVfpgaPath]/RVfpga/src/LiteDRAM*, and that the top module is implemented in file litedram_top.v inside that folder. You can start with the simulation from Figure 3 and add and analyse some signals from the LiteDRAM controller.
- **Exercise 4**) Analyse in simulation and on the board other I$ configurations, such as an I$ with a different block size. Recall that the number of ways cannot be modified.
- **Exercise 5**) Analyse the logic that checks the correctness of the parity information from the Data Array and from the Tag Array.

Lab 20:
ICCM, DCCM, and Benchmarking

RVfpga Lab 20: ICCM, DDCM & Benchmarking

- **Scratchpad memories:**
  - Instruction Closely-Coupled Memory (**ICCM**)
  - Data Closely-Coupled Memory (**DCCM**)

RVfpga Lab 20: Address Space (Instructions)



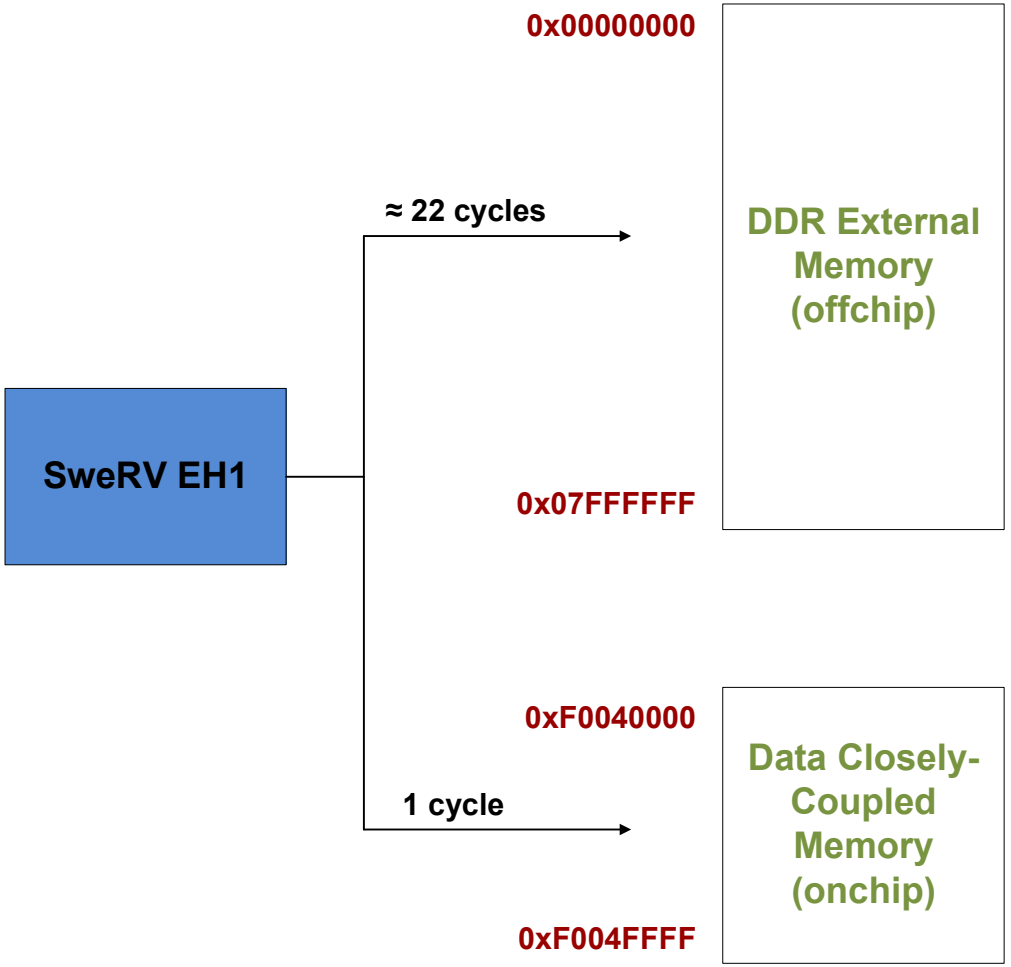
RVfpga Lab 20: Address Space (Data)

- **External memory (~22 cycles):**

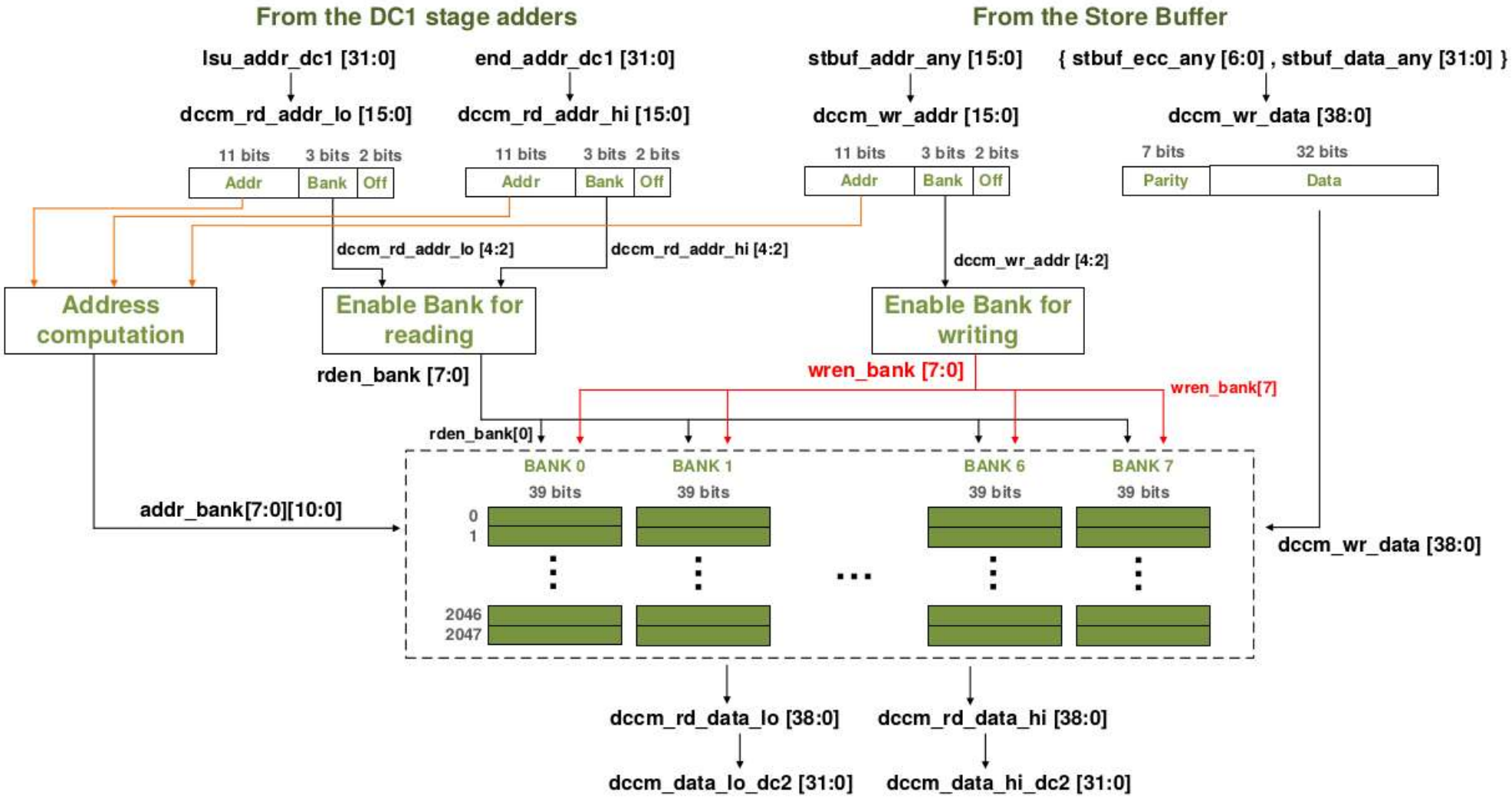
  `0x00000000 – 0x07FFFFFF`

- **On-chip memory (DCCM, ~1 cycle):**

  `0xF0040000 – 0xF004FFFF`



RVfpga Lab 20: ICCM Configuration and Operation

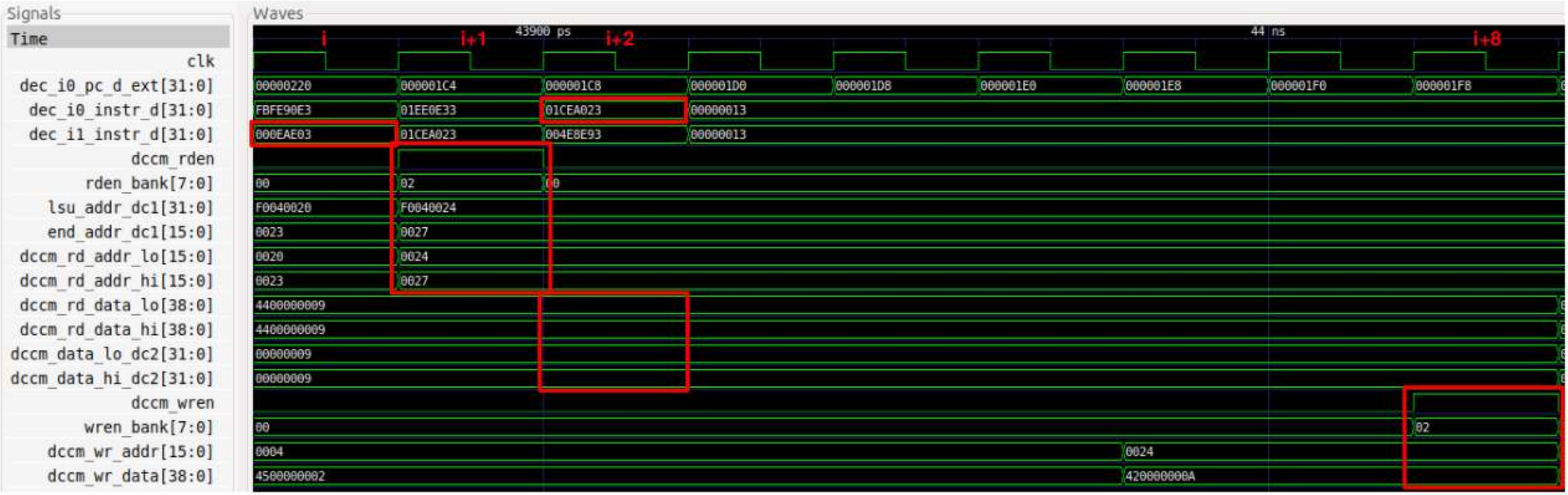


RVfpga Lab 20: Accessing the ICCM – Example

```
// 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: Accessing the ICCM – Example



RVfpga Lab 20: Accessing the ICCM – Example

- **Cycle *i***: The `lw` instruction is decoded in Way 1: `dec_i1_instr_d` = 0x000eae03.
- **Cycle *i+1***: The address is generated in the DC1 stage and provided to the DCCM. As a result of the address check, reading the DCCM is enabled: `dccm_rden` = 1. This signal is provided to the DCCM and, along with the 3-bit Bank field of the address, determines the bank that must be read.
- **Cycle *i+2***: The read data is obtained from the DCCM and provided to the core.
- **Cycle *i+8***: After adding 1 (the immediate) to the read value (0x00000009 + 1 = 0x0000000A) and traversing the Store Buffer, as explained in Lab 13, the data and address are provided to the DCCM, and writing of the correct bank is enabled.

RVfpga Lab 20: Benchmarking

- **Benchmarks:**
  - Run set of programs on processor
  - Compare processors
- **Two common benchmarks:**
  - CoreMark
  - Dhrystone
- Benchmarks use **hardware counters** (HW Counters) to measure events (such as number of instructions, number of cycles).

RISC-V Hardware Counters

- Special-purpose registers to record performance and other metrics (shown below).

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 commmited	21	Fence	38	Timer interrupts taken
5	Instrs commited 16-b	22	Fence.i	39	Exteranal interrupts taken
6	Instrs commited 32-b	23	Mret	40	TLU flushes
7	Instrs aligned	24	Branches commited	41	Branch error flushes
8	Instrs decoded	25	Branches mispredicted	42	I-bus transactions – instr
9	Muls commited	26	Branches taken	43	D-bus transactions – ld/st
10	Divs commited	27	Unpredictable branches	44	D-bus transactions misaligned
11	Loads commited	28	Cycles fetch stalled	45	I-bus errors
12	Stores commited	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 commited	32	Cycles presync stalled	49	Cycles interrutps disabled
16	CSR read	33	Cycles frozen	50	Cycles interrupts stalled while disabled

Table 7-2 in SweRV EH1 Programmer's Reference Manual: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

How to Use and Initialize Hardware Counters

- **Include Western Digital's PSP (Platform Support Package):**
  - `#include <psp_api.h>`
- **Enable counters:**
  - `pspEnableAllPerformanceMonitor(1);`
- **Set counters to measure various metrics:**
  - `pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);`
  - `pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);`
- **Read metrics:**
  - `cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);`
  - `instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);`
- **Print metrics:**
  - `printfNexys("Cycles = %d", cyc_end - cyc_beg);`
  - `printfNexys("Instructions = %d", instr_end - instr_beg);`

Example Program with Hardware Counters

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

   uartInit();

   pspEnableAllPerformanceMonitor(1);                                  // enable counters

   pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE); // assign
   pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);  // counters

   cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);                 // read counters
   instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);

   Test_Assembly();

   cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);                 // read counters
   instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);

   printfNexys("Cycles = %d", cyc_end-cyc_beg);                        // print values
   printfNexys("Instructions = %d", instr_end-instr_beg);
}
```

RVfpga Lab 20: Metrics

- **CoreMark Metrics**
  - CoreMark runs multiple iterations of a loop.
  - **CoreMark Score (CM):** The number of iterations it completes per second (i.e., the iterations/second).
  - **CM/MHz:** CM divided by the clock frequency in MHz (also called Iterat/Sec/MHz or iterations/second/MHz).
- Recall, **ideal IPC** (instructions per cycle) is **2** for SweRV EH1 because it is 2-way superscalar.

RVfpga Lab 20: CoreMark Performance

	Compiler = debug External Memory	Compiler = debug DCCM	Compiler = optimized DCCM
CM/MHz	0.47	1.88	3.47
# Instructions	~0.5 million	~0.5 million	0.309 million
# Cycles	~2 million	~0.5 million	0.288 million
IPC (instructions/cycle)	0.25	~1	~1
Data Bus Transactions	~133,000 (all go to external memory)	0 (due to DCCM)	0 (due to DCCM)
Instruction Bus Transactions	392 (due to I$)	392 (due to I$)	392 (due to I$)

RVfpga Lab 20: Tasks and Exercises – Sample

- **TASK**: Using the instructions provided in Lab 1, implement a new RVfpga System that includes a 64 KiB ICCM.
- **TASK**: Simulate an unaligned read to the DCCM and analyse how it is handled inside the DCCM.
- **TASK**: Simulate a DCCM bank conflict by modifying the program from Figure 4.
- **TASK**: Modify file platformio.ini to use both the DCCM for storing most data and the ICCM for storing the instructions. Execute the *CoreMark* benchmark and compare the results with the ones obtained in this section.
- **TASK**: Modify the compilation optimization to -O3 and explain the results.
- **Exercise 1**) Do the same analysis as was done for *CoreMark* but this time using the *Dhrystone* benchmark.
- **Exercise 2**) Do the same analysis as was done for *CoreMark* but this time for the *ImageProcessing* application from Lab 4.
- **Exercise 3**) Enable/disable various core features. Compare the performance results. Run all three programs (CoreMark, Dhrystone, and ImageProcessing) on these modified RVfpga Systems on the Nexys A7 board.

Acknowledgements

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

ADVISER
Prof. David Patterson

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

ASSOCIATES
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 Ataur 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

Sponsors and Supporters

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