RVfpga

The Complete Course in Understanding Computer Architecture

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

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 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

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

Introduction

- RISC-V FPGA (**RVfpga**) is a teaching package that provides a set of instructions, tools, and labs that show how to:
  - **Target** a commercial RISC-V system-on-chip (SoC) to an **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.
- The RVfpga System is built around Chips Alliance's **SweRVolf SoC**, which is based on Western Digital's RISC-V **SweRV EH1** core.

RVfpga Overview

- The **RVfpga Package** provides:
  - a **comprehensive, freely distributed, complete RISC-V course**
  - a **hands-on** and **easily accessible** way to learn about RISC-V processors and the RISC-V ecosystem
  - a RISC-V system targeted to **low-cost FPGAs**, which are readily available at many universities and companies.
- After completing the RVfpga Course, users will walk away with a **working RISC-V processor, SoC, and ecosystem** that they understand and know how to use and modify.

RVfpga Course Contents

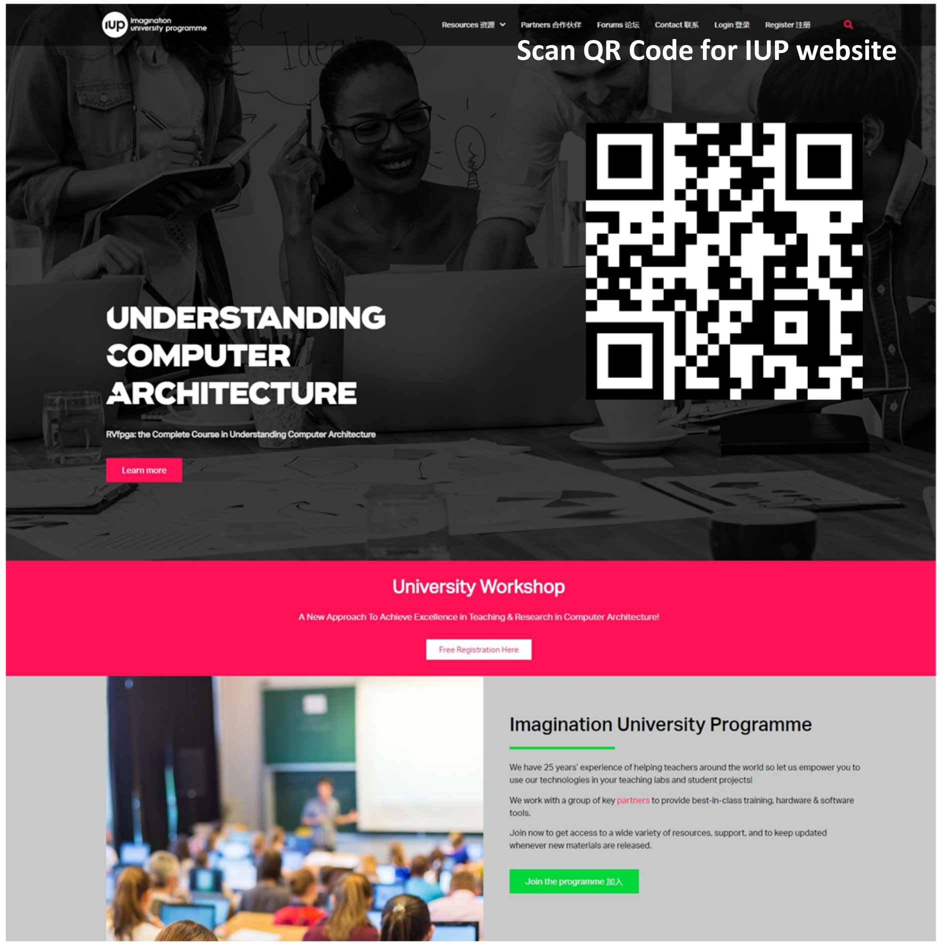
RVfpga Contents

- **Getting Started Guide**
  - Quick Start Guide
  - Overview of RISC-V Architecture and RVfpga
  - Installing Tools (VSCode, PlatformIO, Vivado, Verilator, and Whisper)
  - Running the RVfpga System in Hardware and Simulation
- **Labs**
  - **1-10:** Building the RVfpga System in Vivado, Programming and Extending the system by adding peripherals (released Nov 2020)
  - **11-20:** Analyzing and modifying RVfpga's RISC-V core and memory system (to be released Q4 2021)

RVfpga Course

- **2-4 Semester Course**
  - Undergraduate (Labs 1-10)
  - Master's/upper division (Labs 11-20)
- **Expected Prior Knowledge**
  - Fundamental understanding of **digital design**, **high-level programming** (preferably C), **instruction set architecture** and **assembly programming**, processor **microarchitecture**, **memory** systems (this material is covered in *Digital Design and Computer Architecture: RISC-V Edition*, Harris & Harris, © Elsevier, expected publication: summer 2021)
  - These topics will be expanded on and solidified with hands-on learning throughout the RVfpga course

How to Get RVfpga



Imagination University Programme Website

- **Register for Imagination University Programme (IUP)** – for teachers, researchers, and students worldwide:

  https://university.imgtec.com

  – Receive **updates** and **notifications** of release
  – **Request & download** materials
  – **Support Forums:** PowerVR, RVfpga, & AI Forums; IUP Forum for curriculum/teaching discussions
- **Social Media:**
  – **Robert Owen, IUP Director:** @UniPgm
  – **Imagination Technologies:** @ImaginationTech
  – **WeChat & Weibo:** ImaginationTech

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)

HARDWARE*

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

*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**

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

Supported Platforms

- Operating Systems
  - **Ubuntu 18.04** (although later versions likely also work)
  - **Windows 10**
  - **macOS**

RVfpga Software Tools

- **Xilinx's Vivado IDE**
  - View RVfpga source files (Verilog / SystemVerilog) and hierarchy
  - Create bitfile (FPGA configuration file) for RVfpga targeted to Nexys A7 board
- **PlatformIO** – an extension of Visual Studio Code (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

Nexys A7-100T FPGA Board

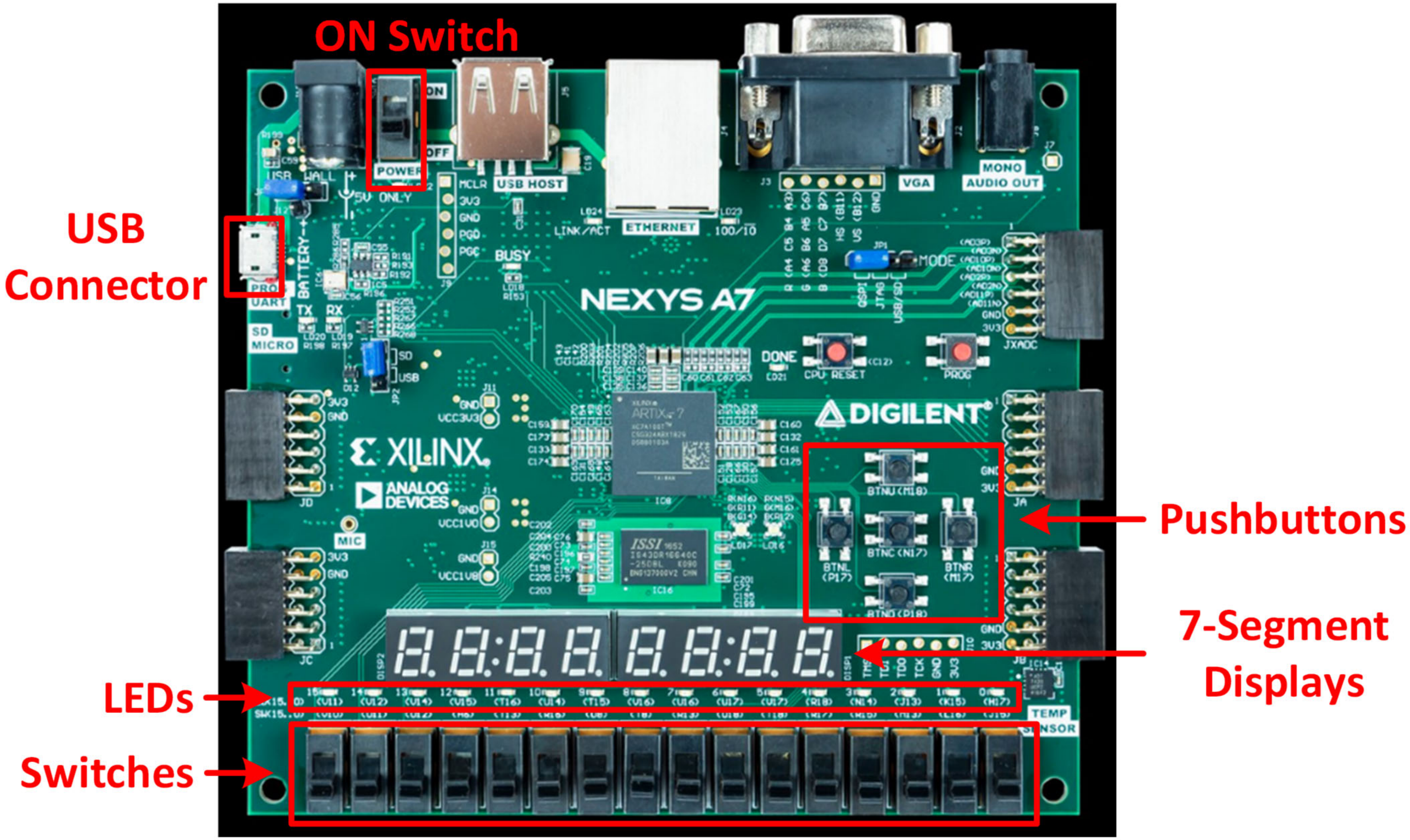
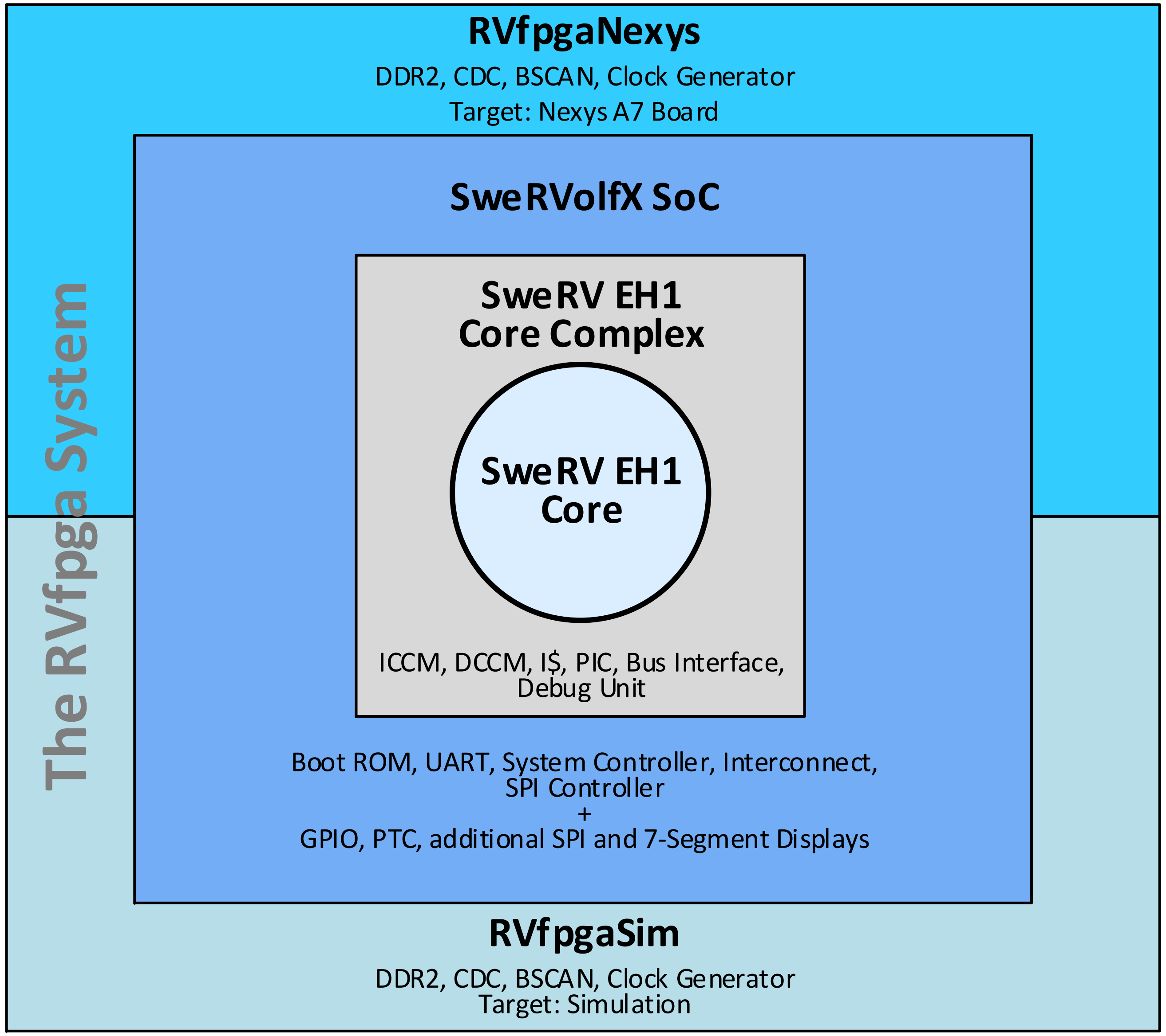


figure of board from https://reference.digilentinc.com/

- 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 **digilentinc.com** and other vendors

RISC-V Cores and SoCs

RVfpga Hierarchy



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.
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

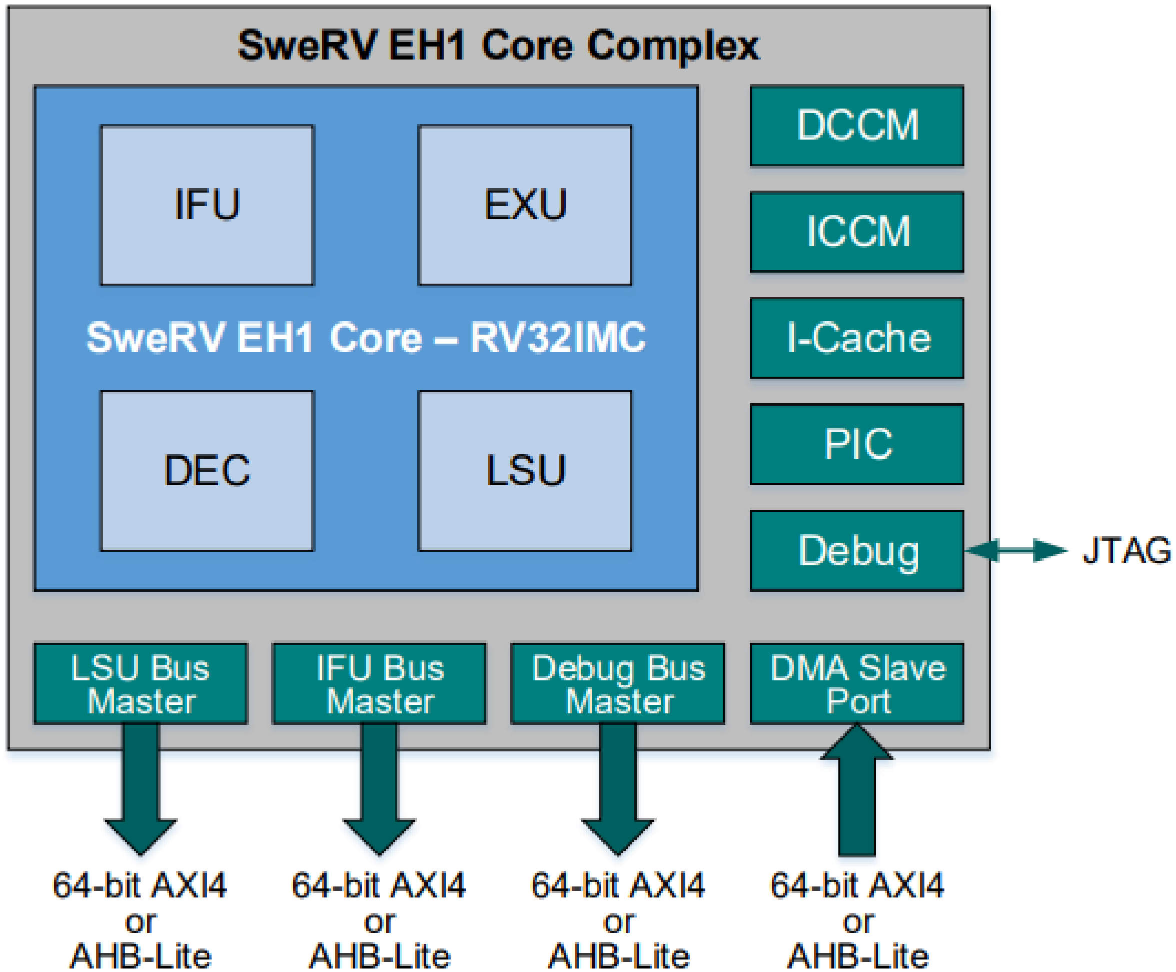
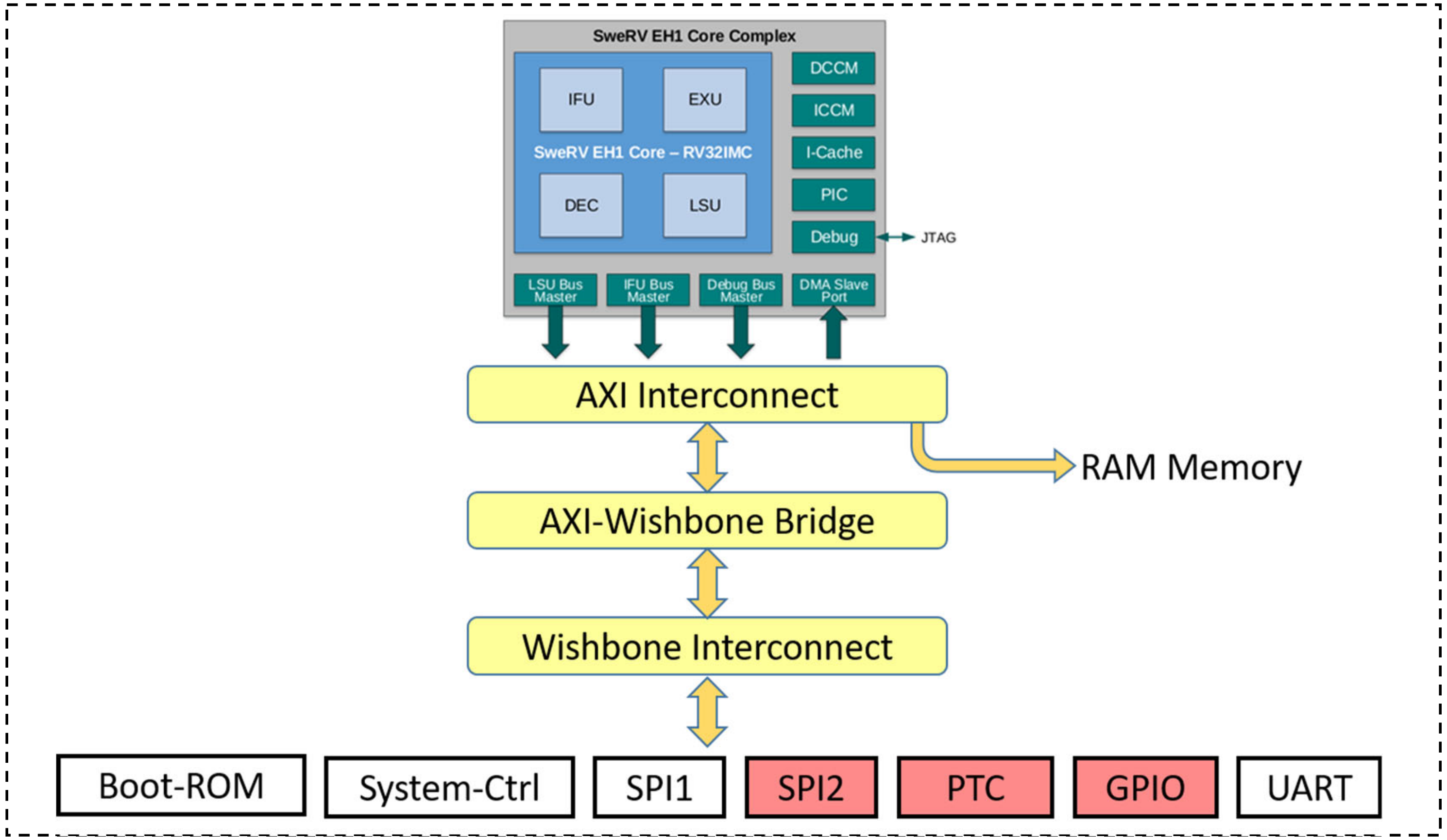


Figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

- 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

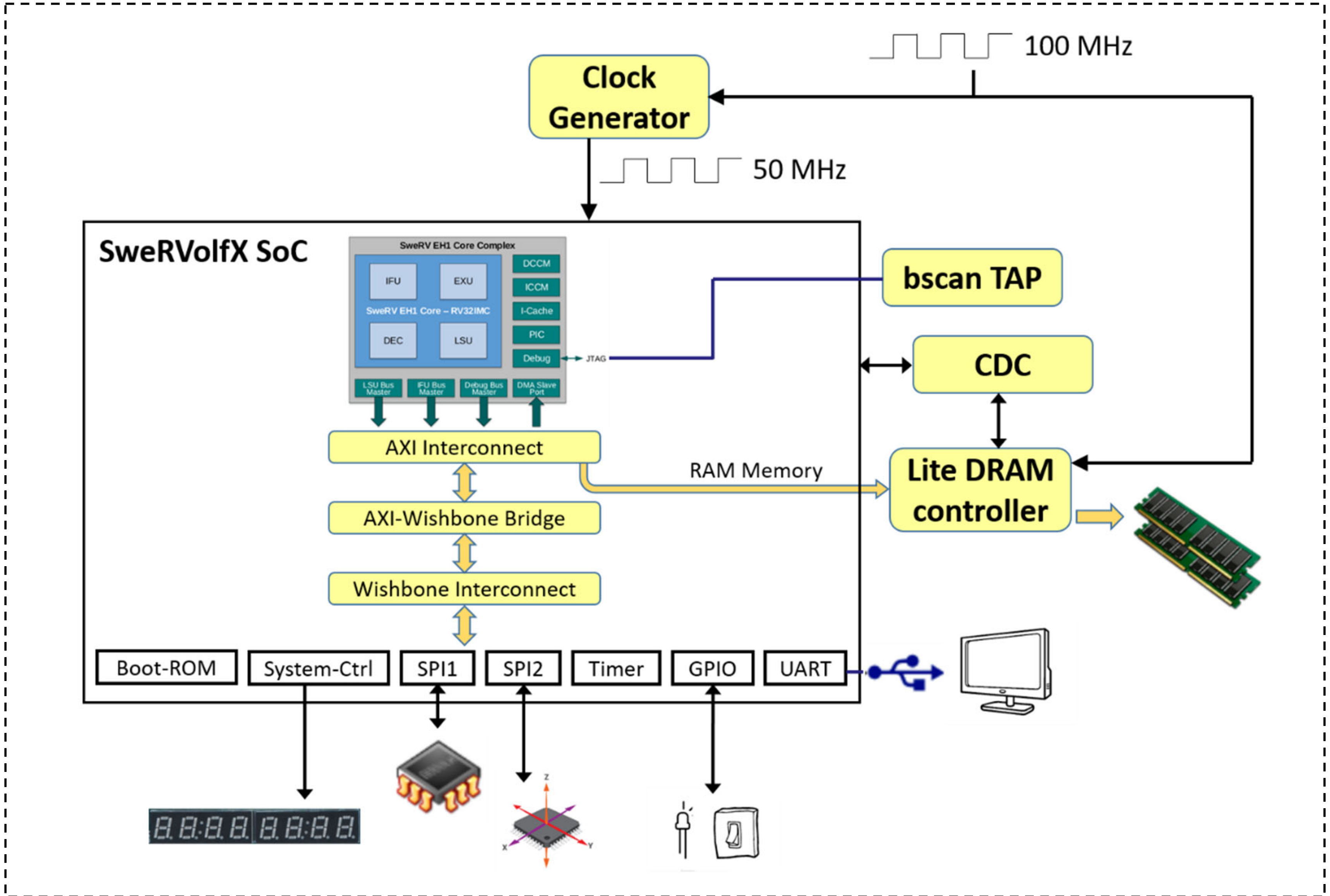


- Open-source system-on-chip (SoC) from Chips Alliance
- SweRVolf uses the SweRV EH1 Core. SweRVolf includes a Boot ROM, UART, and a System Controller and an SPI controller (SPI1)
- SweRVolfX extends SweRVolf with another SPI controller (SPI2), a GPIO (General Purpose Input/Output), 8-digit 7-Segment 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

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

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

- The SweRVolfX SoC can also include a Verilog wrapper to enable simulation.

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

Rvfpga System Extensions

- The Rvfpga System 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

Labs 1-10 (released Nov 2020)

- Vivado Project and Programming
- I/O Systems

Labs 11-20 (to be released Q4 2021)

- RISC-V Core
- RISC-V Memory Systems

All labs include **exercises** for using and/or modifying the RVfpga System to increase understanding through hands-on design.

RVfpga Labs 1-10

Show how to view the RVfpga System source code (Verilog/SV) and target it to an FPGA (Lab 1), write C and assembly programs (Labs 2-5), and modify RVfpgaNexys/RVfpgaSim to add peripherals (Labs 6-10).

- **Lab 0: Overview of RVfpga Labs**
- **Lab 1: Creating a Vivado Project**

Programming

- **Lab 2: C Programming**
- **Lab 3: RISC-V Assembly Language**
- **Lab 4: Function Calls**
- **Lab 5: Image Processing: C & Assembly**

I/O Systems

- **Lab 6: Introduction to I/O**
- **Lab 7: 7-Segment Displays**
- **Lab 8: Timers**
- **Lab 9: Interrupt-driven I/O**
- **Lab 10: Serial Buses**

RVfpga Labs 1-5: Vivado Project & Programming

- **Lab 1: Creating a Vivado Project:** Build a Vivado project to target RVfpgaNexys to an FPGA board and simulate RVfpgaSim in Verilator
- **Lab 2: 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 3: RISC-V Assembly Language:** Write a RISC-V assembly program in PlatformIO and run /debug it on RVfpgaNexys/RVfpgaSim/Whisper
- **Lab 4: Function Calls:** Introduction to function calls, C libraries, and the RISC-V calling convention
- **Lab 5: Image Processing: C & Assembly:** Embed assembly code with C code

RVfpga Labs 6-10: I/O & Peripherals

- **Lab 6: Introduction to I/O:** Introduction to memory-mapped I/O and 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 Rvfpga System interrupt support and use of interrupt-driven I/O
- **Lab 10: Serial Buses:** Introduction to serial interfaces (SPI, I2C, UART). Use on board accelerometer that uses an SPI interface

RVfpga Labs 11-15: The RISC-V Core

- **To be released in Q4 2021**
- Understanding the core structure
- Understanding instruction flow through the pipeline (Arithmetic/Logic, Memory, Jumps, and Branches)
- Understanding hazards and how to deal with them
- Implementing new instructions and executing them on the FPGA board
- Understanding and modifying the branch predictor
- Understanding superscalar processing

RVfpga Labs 16-20: RISC-V Memory Systems

- **To be released in Q4 2021**
- Understanding the operation of the memory hierarchy including cache hits and misses
- Modifying the cache: implementing different cache sizes, configurations, and management policies
- Understanding the cache controller
- Understanding the memories: ICCM (instruction closely coupled memory) and DCCM (data closely coupled memory)

RVfpga Timeline

RVfpga Availability	
November 2020	RVfpga Getting Started Guide RVfpga Labs 1-10
Q4 2021	RVfpga Labs 11-20
June 2021	Masters-level SoC Design Course
Languages	English & Chinese (Spanish & Japanese to follow)
Textbook	*Digital Design & Computer Architecture: RISC-V Edition 2021* by Sarah Harris and David Harris

- **Target Audience**
  - Undergraduates 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
Quick Start Guide

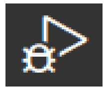
Quick Start Guide Overview

- Install VSCode & **PlatformIO**
- Run **Example Program** on **RVfpgaNexys**

Install PlatformIO & VSCode

- Install **VSCode**
  - https://code.visualstudio.com/Download
  - For Ubuntu and macOS, install **Python** (this step is not required for Windows)
- Install **PlatformIO** extension within VSCode
- Install Nexys A7 Board **drivers** (see RVfpga Getting Started Guide instructions)

Download RVfpgaNexys onto Board and Run Program

- In **PlatformIO**:
  - Open **example program** that writes value of switches to LEDs. Program is in: *[RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang*
  - Update directory location of **RVfpgaNexys bitfile** in PlatformIO initialization file (platformio.ini) – i.e., add this line to platformio.ini: *board_build.bitstream_file = [RVfpgaPath]/RVfpga/src/rvfpga.bit*
  - **Download RVfpgaNexys** onto Nexys A7 Board (Project Tasks → env:swervolf_nexys → Platform → Upload Bitstream)
  - **Compile, download, and run program on RVfpgaNexys** by pressing the Run/Debug button: 

[RVfpgaPath] is the location of the **RVfpga** folder on your machine. This folder was provided with the RVfpga package from the Imagination University Programme.

RVfpga Labs Descriptions

Lab 1:
Vivado Project

RVfpga Lab 1: 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
- Create a **Vivado project** that contains RVfpga System's source code. Synthesize RVfpgaNexys targeted to Nexys A7 board and create a **bitfile** (also called bitstream file) that contains information to configure the FPGA as RVfpgaNexys.
- You may also use **Verilator**, an HDL simulator, to simulate RVfpga System's source code and examine internal signals (see the RVfpga Getting Started Guide for instructions on how to use Verilator).
- Vivado and Verilator will be used extensively in RVfpga **Labs 6-10** for modifying and simulating the RVfpga System.

Lab 2:
C Programming

RVfpga Lab 2: C Programming

- Create **PlatformIO project**
- Add example **C program** to project
- **Download RVfpgaNexys** to Nexys A7 board
- **Download C program** onto RVfpgaNexys and **run/debug** program
- Complete some or all of **exercises** at end of lab
- Remember that you can also simulate the program in Verilator (using **RVfpgaSim**) or **Whisper**.

RVfpga Lab 2: 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;

  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.

RVfpga Lab 2: 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 2: Western Digital's BSP & PSP

- Western Digital provides:
  - **PSP:** processor 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 2: 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 **platform.ini** file:
  monitor_speed = 115200
- After program starts running, **open PlatformIO terminal** by pressing this button in the bottom of the window:

Lab 3: RISC-V Assembly

RVfpga Lab 3: RISC-V Assembly

- RISC-V Assembly Language **Overview**
- Create **PlatformIO project**
- Add example **RISC-V assembly program** to project
- **Download RVfpgaNexys** to Nexys A7 board
- **Download RISC-V assembly program** onto RVfpga and **run/debug** program
- Complete some or all of **exercises** at end of lab
- Remember that you can also simulate the program in Verilator (using **RVfpgaSim**) or **Whisper**.

RVfpga Lab 3: 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 3: 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 = Memory address where variable A is stored
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 3: 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 3: Example RISC-V Assembly Program

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

.globl main
main:

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
```

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

Lab 4: Function Calls

RVfpga Lab 4: Function Calls

- Write C programs with **function calls**
  - Functions are also called *procedures*
- Using **C libraries**
- RISC-V (Procedure) **Calling Convention**

RVfpga Lab 4: 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 4: 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 4: 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 4: 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 4: 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 4: 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 4: 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 4: 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 4: 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
```

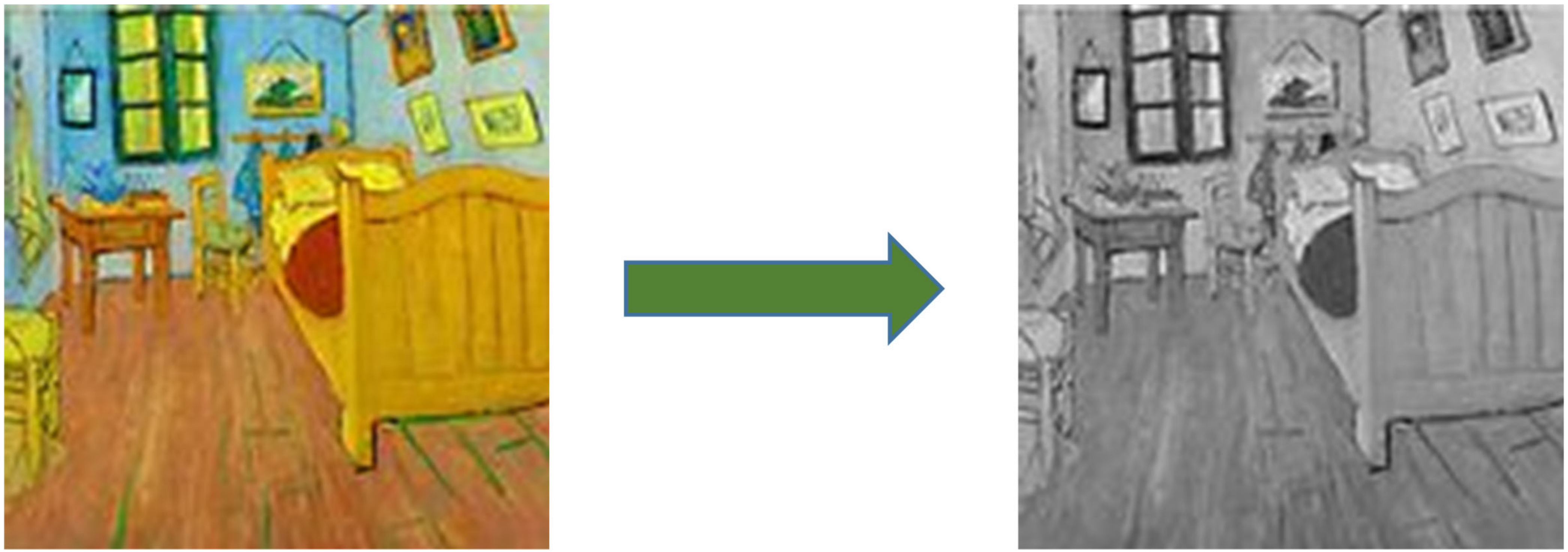
Lab 5:
C and Assembly

RVfpga Lab 5: Combining C and Assembly

- **Example:** Image processing program
- Some functions written in C and some in assembly

RVfpga Lab 5: Image Processing Program

- **Convert colour image to greyscale**



RVfpga Lab 5: 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 5: 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 `CoulourToGrey_Pixel` function visible to all files in project

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

RVfpga Lab 5: 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 5: Main Function

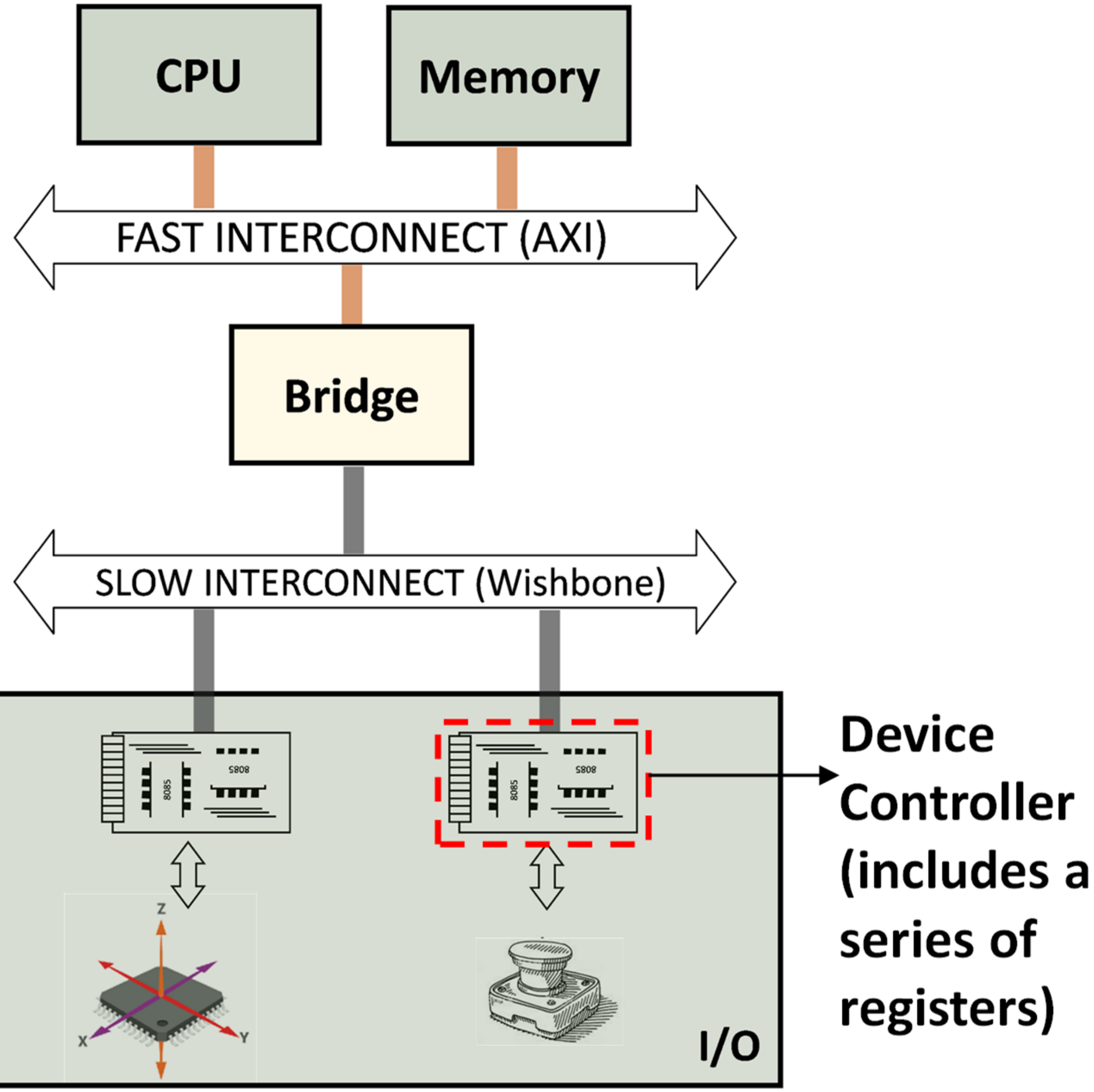
```
int main(void) {
  // Create an N x M matrix using the input image
  initColourImage(ColourImage);

  // 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);
}
```

Lab 6: Intro to I/O

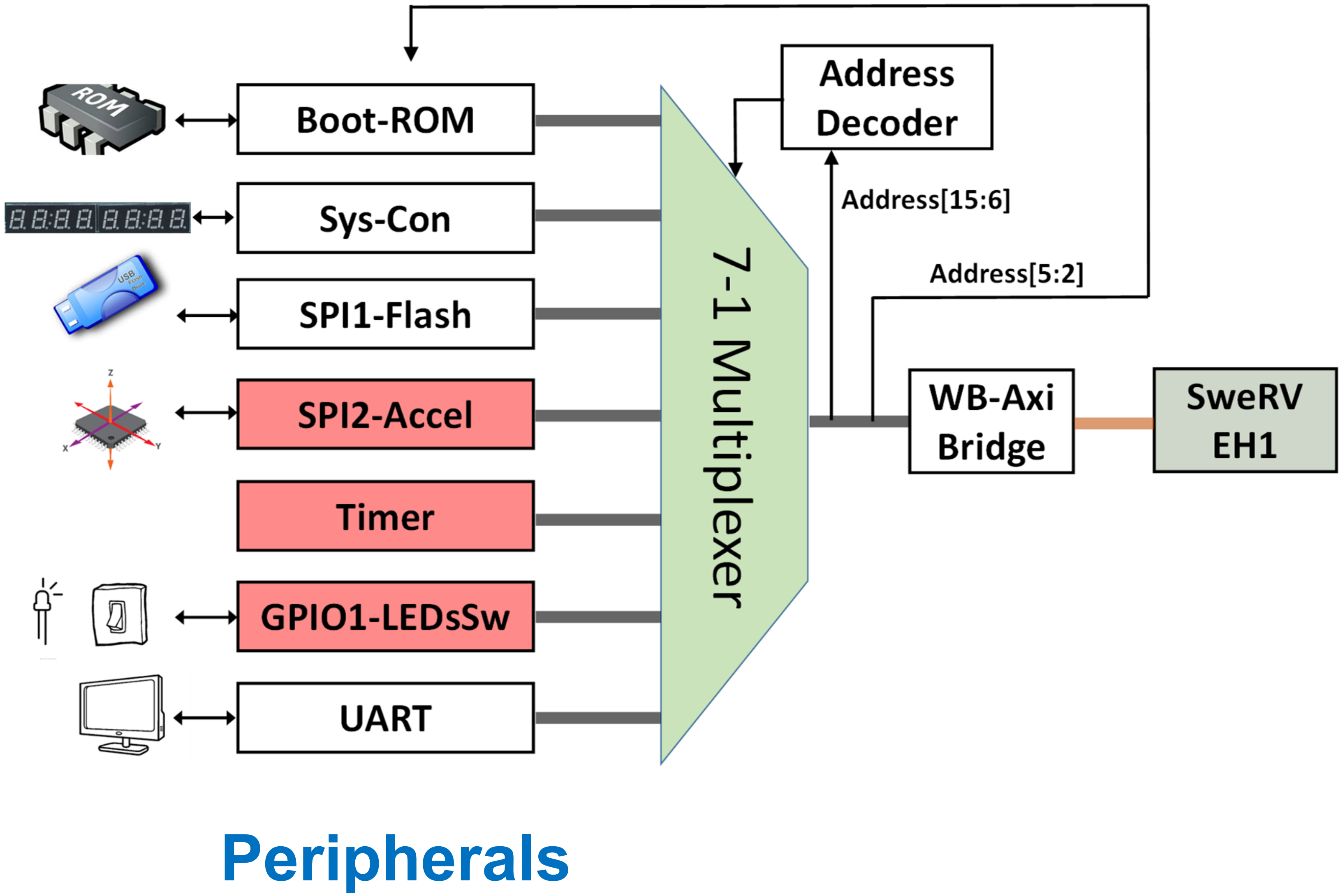
RVfpga Lab 6: Introduction to I/O

- Input/output (I/O) systems – also called *peripherals*
- General-purpose I/O (GPIO)
- GPIO controllers

RVfpga Lab 6: Generic Processor with I/O



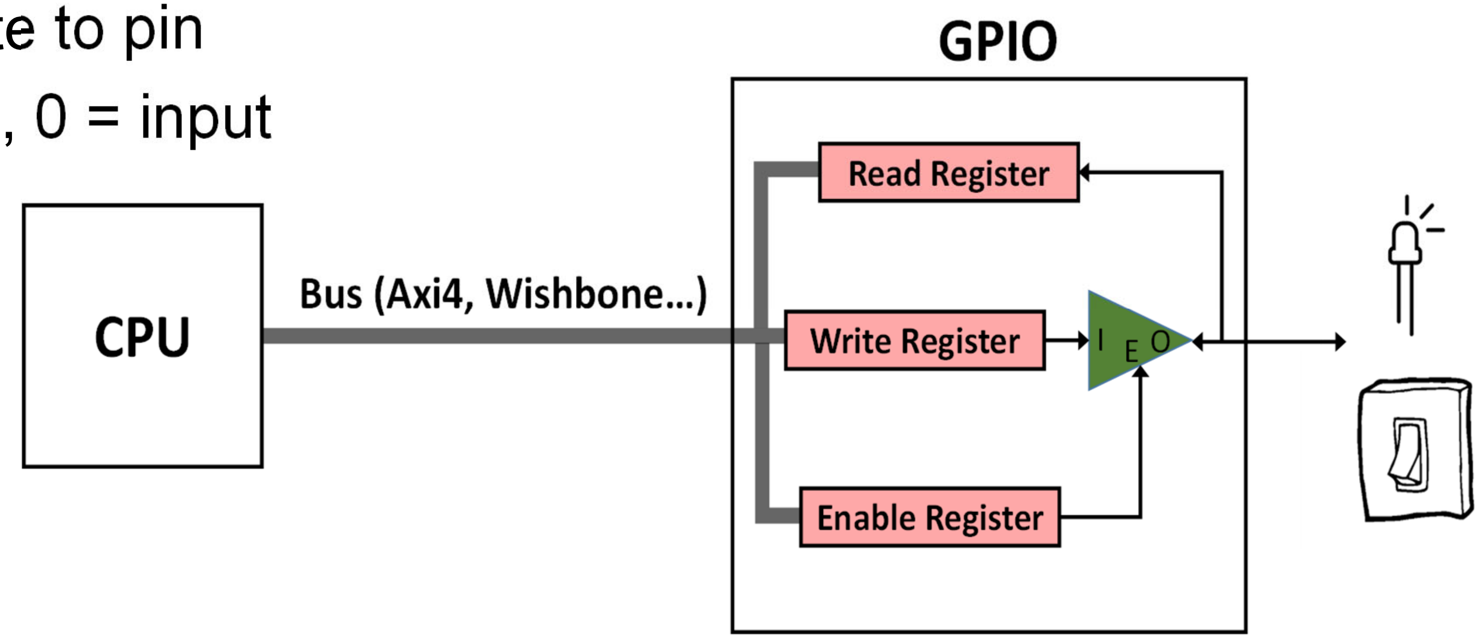
RVfpga Lab 6: Processor with I/O



- **SweRVolfX peripherals:**
  - Boot ROM
  - System Controller
  - SPI1 Flash Memory
  - UART
  - GPIO LEDs and switches
  - Timer
  - SPI2 Accelerometer
  - 7-segment displays (within System Controller: Sys-Con)

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



Peripherals

RVfpga Lab 6: Memory-Mapped Registers

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

- **Configure bits 15:0 of GPIO as outputs, 31:16 as inputs:**

```
li t0, 0x80001400   # t0 = 0x80001400
li t1, 0xFFFF       # 1 = output, 0 = input
sw t1, 8(t0)        # [15:0] = outputs, [31:16] = inputs
```

- **Reading I/O:**

```
lw t2, 0(t0)        # t2 = value of GPIO pins
```

- **Writing I/O:**

```
sw t3, 4(t0)        # GPIO pins = t3
```

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

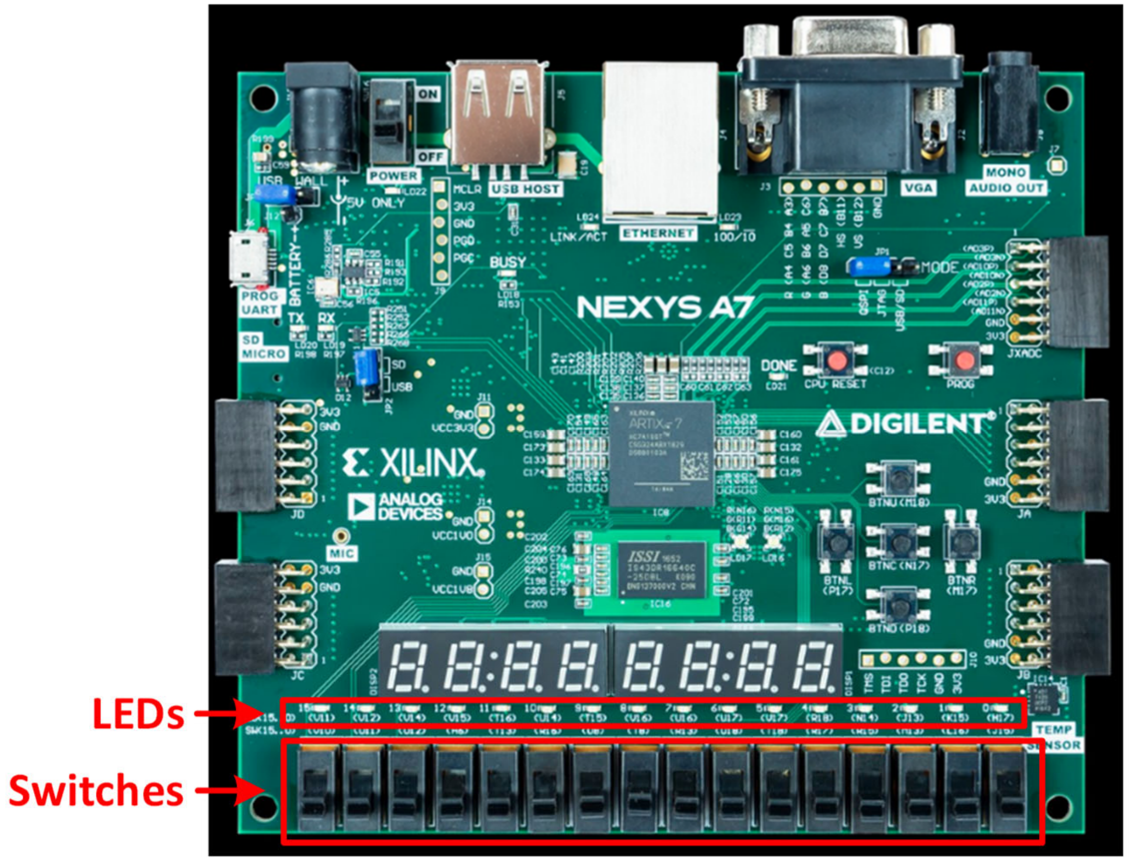


figure of board from https://reference.digilentinc.com/

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
```

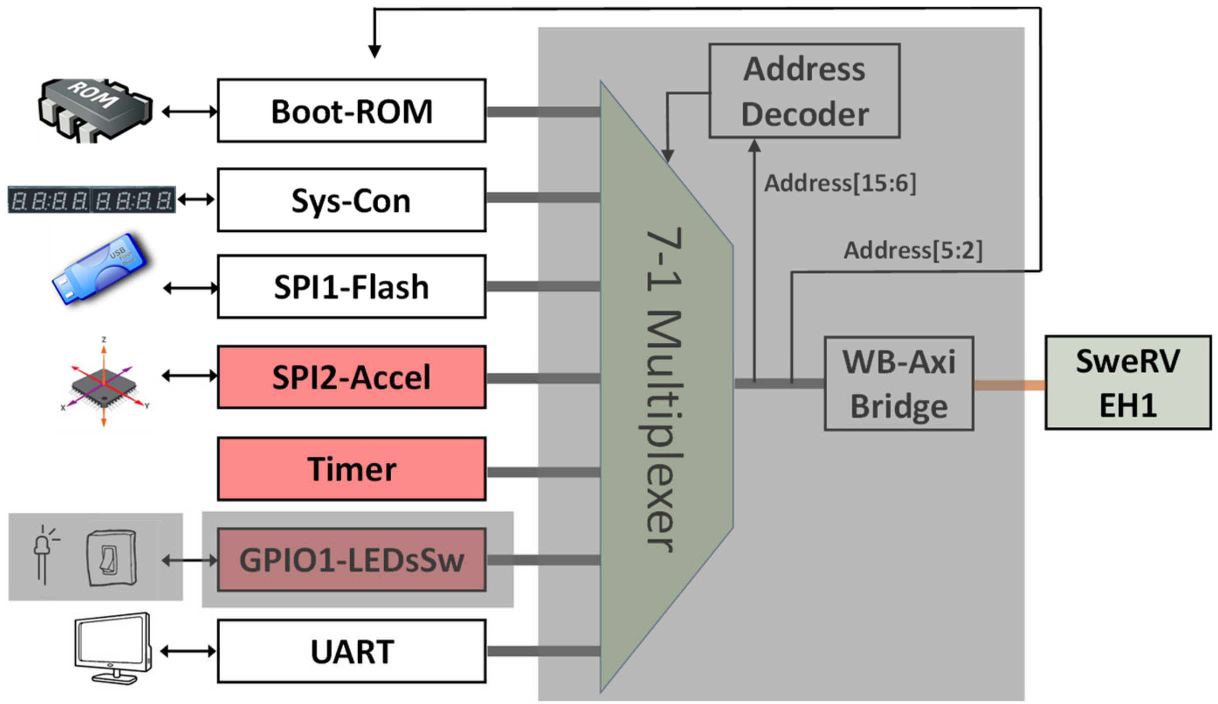
Read Switches:

- 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: 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

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { i_sw[0] }]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { i_sw[1] }]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { i_sw[2] }]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { i_sw[3] }]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { i_sw[4] }]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { i_sw[5] }]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { i_sw[6] }]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { i_sw[7] }]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { i_sw[8] }]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { i_sw[9] }]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { i_sw[10] }]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { i_sw[11] }]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { i_sw[12] }]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { i_sw[13] }]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { i_sw[14] }]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { i_sw[15] }]

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { o_led[0] }]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { o_led[1] }]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { o_led[2] }]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { o_led[3] }]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { o_led[4] }]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { o_led[5] }]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { o_led[6] }]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { o_led[7] }]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { o_led[8] }]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { o_led[9] }]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { o_led[10] }]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { o_led[11] }]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { o_led[12] }]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { o_led[13] }]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { o_led[14] }]
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

```
wb_mux
  #(.num_slaves (7),
    .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
    .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
 wb_mux_io
   (.wb_clk_i  (wb_clk_i),
    .wb_rst_i  (wb_rst_i),
    .wbm_adr_i (wb_io_adr_i),
    .wbm_dat_i (wb_io_dat_i),
    .wbm_sel_i (wb_io_sel_i),
    .wbm_we_i  (wb_io_we_i),
    .wbm_cyc_i (wb_io_cyc_i),
    .wbm_stb_i (wb_io_stb_i),
    .wbm_cti_i (wb_io_cti_i),
    .wbm_bte_i (wb_io_bte_i),
    .wbm_dat_o (wb_io_dat_o),
    .wbm_ack_o (wb_io_ack_o),
    .wbm_err_o (wb_io_err_o),
    .wbm_rty_o (wb_io_rty_o),
    .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}),
    .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}),
    .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}),
    .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 }),
    .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}),
    .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}),
    .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}),
    .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}),
    .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}),
    .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}),
    .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}),
    .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}));

endmodule
```

CPU/Controller Signals

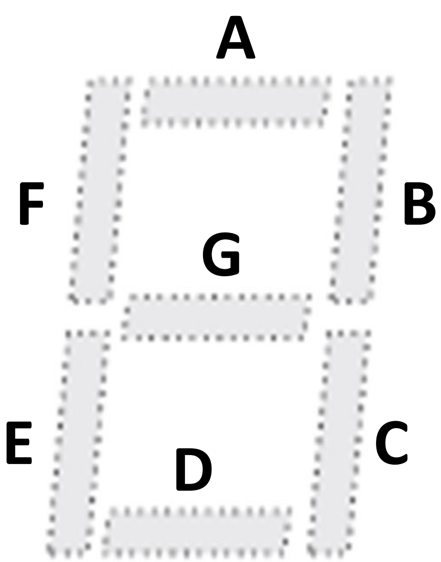
Peripheral Signals

Lab 7:
7-Segment Displays

RVfpga Lab 7: 7-Segment Displays

- Overview of 7-segment displays
- 7-segment display hardware

RVfpga Lab 7: Overview of 7-Segment Displays



- **7 LED segments:** A-G
- **Light up segments to create specific 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

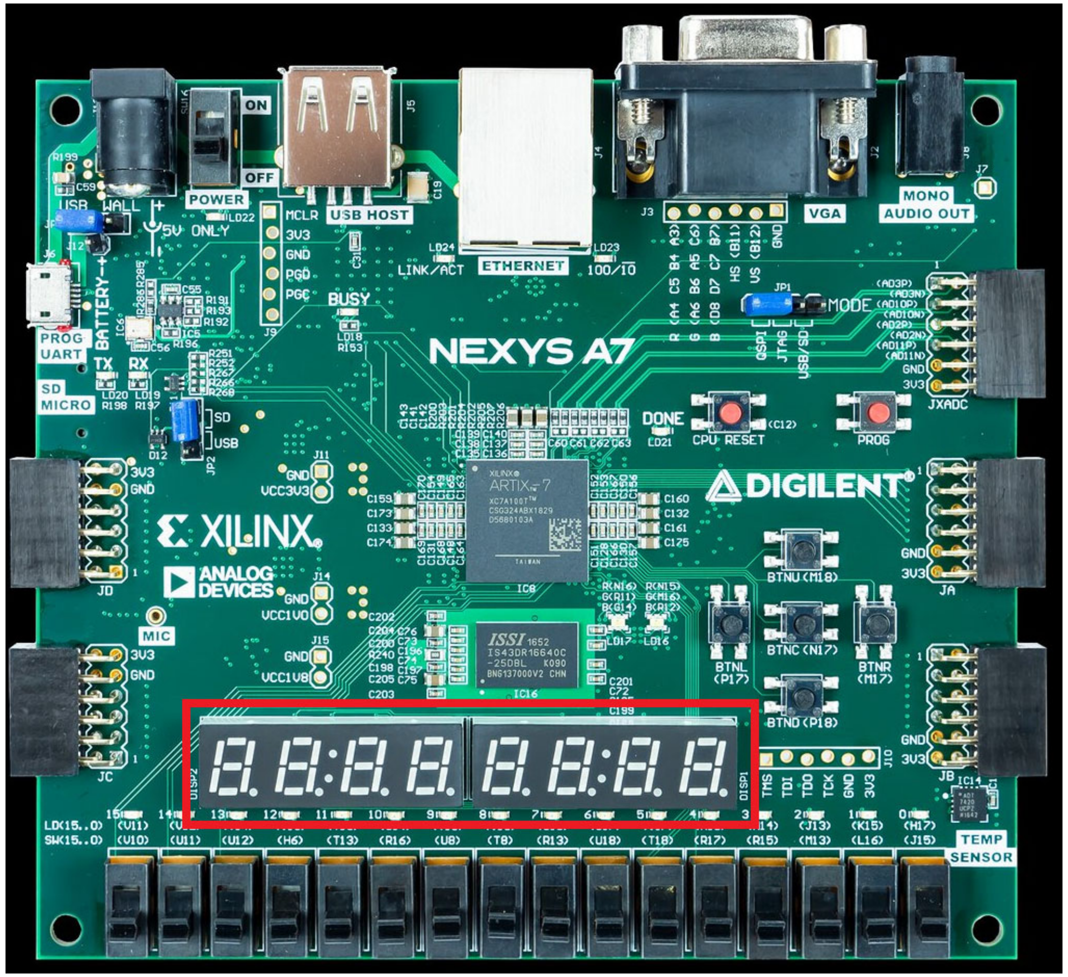


figure of board from https://reference.digilentinc.com/

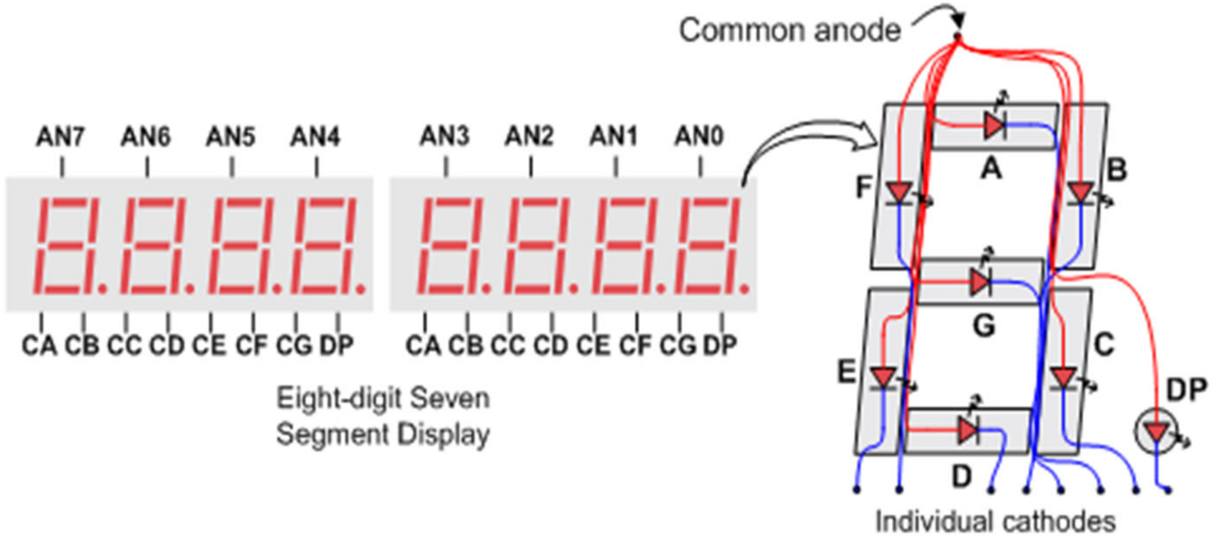
- **8-digit** 7-segment displays
- **Memory-mapped** access:
  - **Enables_Reg:** 0x80001038
  - **Digits_Reg:** 0x8000103C
- **Enables** are **low-asserted**
- **Example:** Display 71 on two right-most digits:
  - **Enables_Reg** = 0xFC (0b11111100: enable two right-most digits)
  - **Digits_Reg** =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

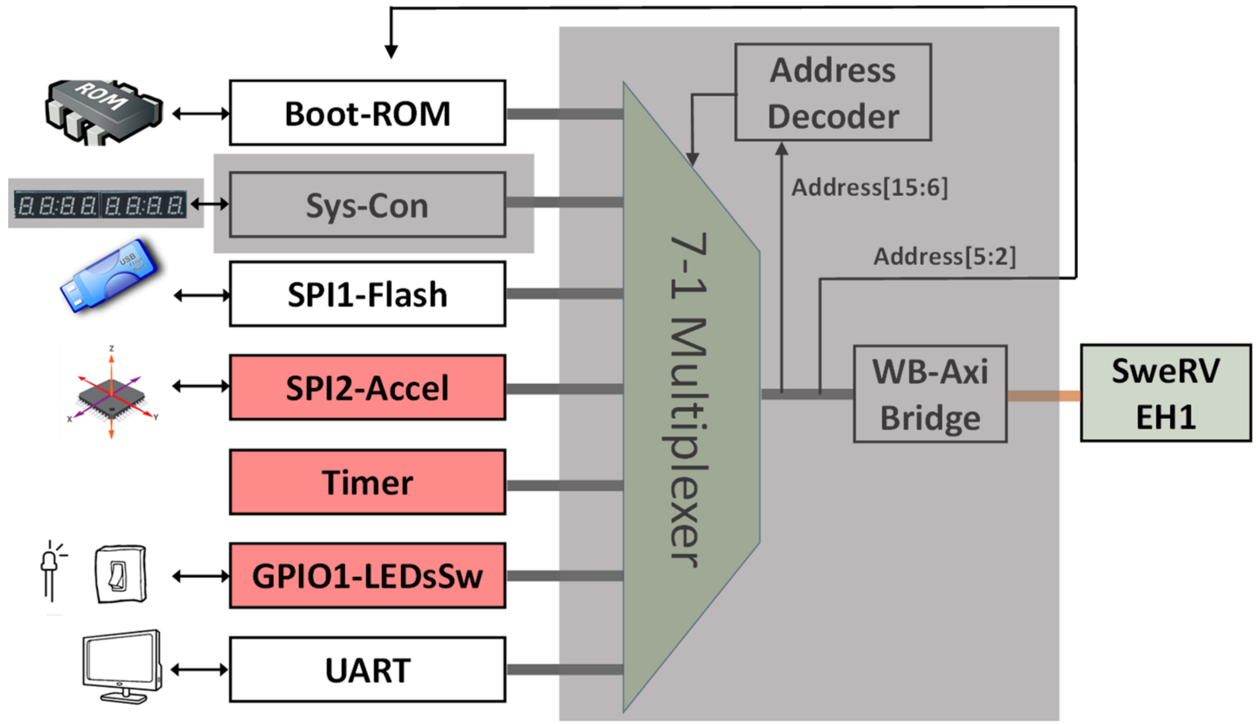
- Each digit is **common anode** (anodes of that digit's LEDs are tied together)
  - Anode signals act as **enables** (AN0 - AN7)
  - Drive **low** to enable digit (AN0 - AN7 go through an inverter (not shown) before being fed to LED)
- Each **segment** for all digits is **tied together**
  - Segments 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: 7-Seg. Disp. Low-Level Implementation

- **Divided in 3 main parts**
  - RvfpgaNexys's external connection to the on-board 7-seg. displays (left shaded region)
  - Integration of the 7-seg. displays module into SweRVolfX (middle shaded region)
  - Connection between the 7-seg. disp. and the SweRV EH1 (right shaded region)



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

```
##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
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 instantiation. The module receives, in addition to the clock signal (i_clk) and the reset signal (i_rst), two input signals, **Enables_Reg** and **Digits_Reg**, which are the two memory-mapped control registers described before. This module outputs two signals, **AN** and **Digits_Bits**, which are connected to the 7-segment displays on the board.

```
// 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: Integration into SweRVolfX

- The **SevSegDisplays_Controller** is also implemented in this file. It contains the following subunits:
  - Two multiplexers (module **SevSegMux**) that select the value to send to the AN and Digits_Bits signals every 2ms.
  - A counter (module **counter**) that creates the 2ms period.
  - A decoder (module **SevenSegDecoder**), which outputs the segment values for a given 4-bit hexadecimal value.

Lab 8: Timers

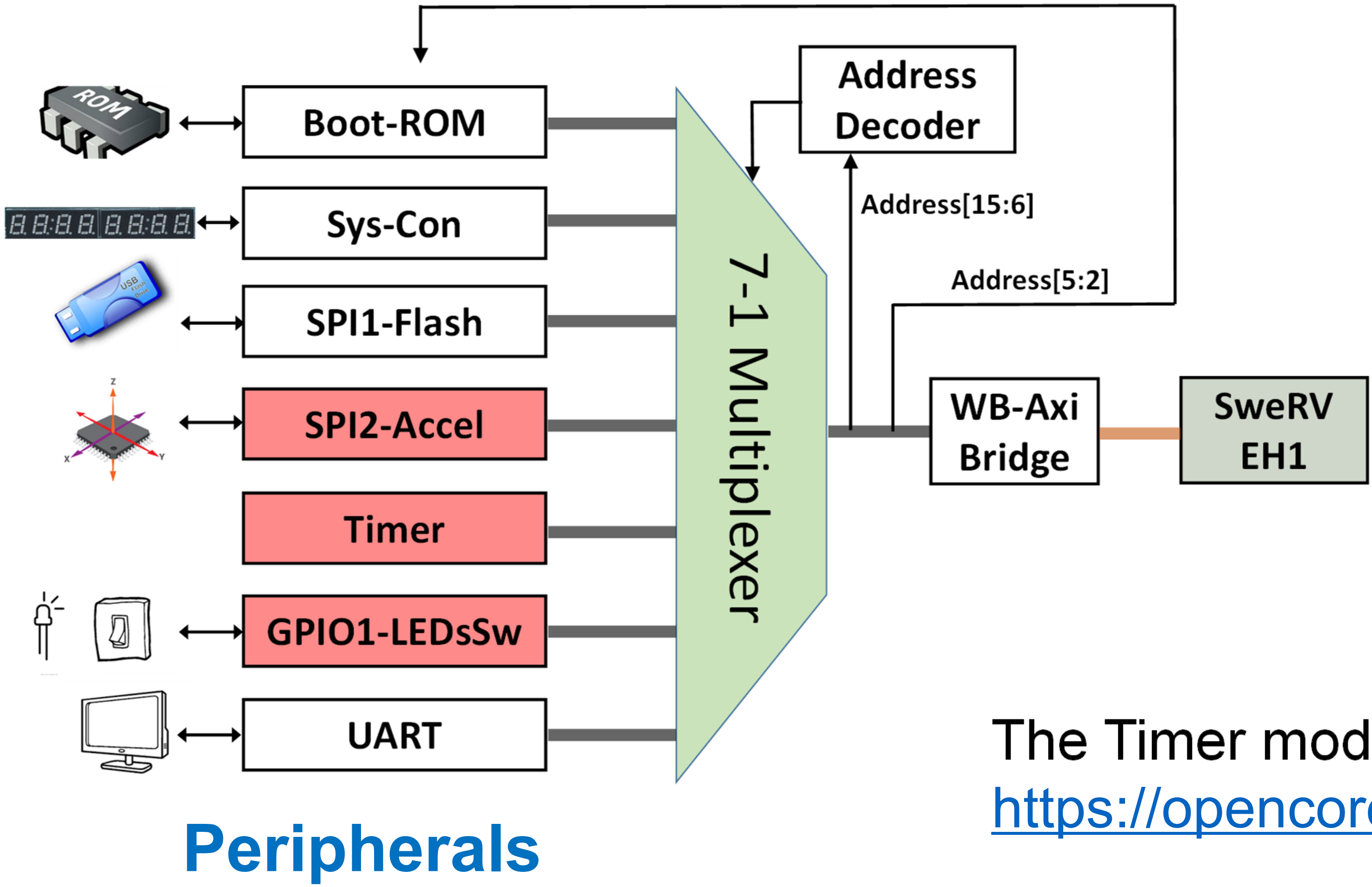
RVfpga Lab 8: Timers

- Overview of timers
- Timer Registers
- Timer Example

RVfpga Lab 8: Timers

- Timers **increment or decrement a counter** at a fixed frequency
- Commonly found in **microcontrollers** and SoCs
- Used to generate **precise timing**

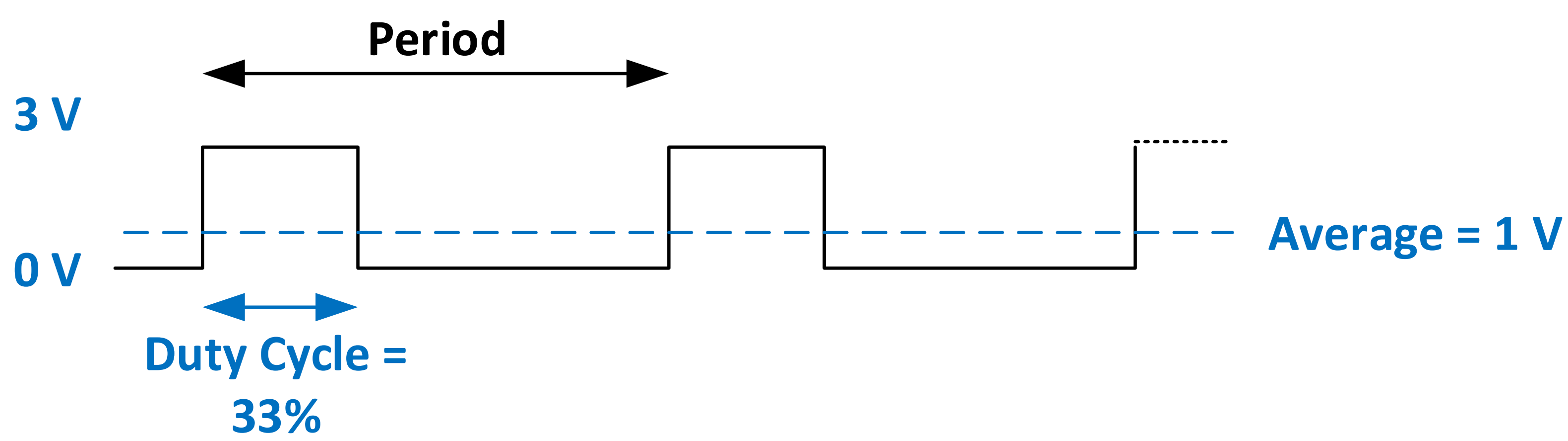
RVfpga Lab 8: Timers



The Timer module used is from OpenCores:
https://opencores.org/projects/ptc

RVfpga Lab 8: Timer (PTC) Module

- 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 (PTC) Control Register

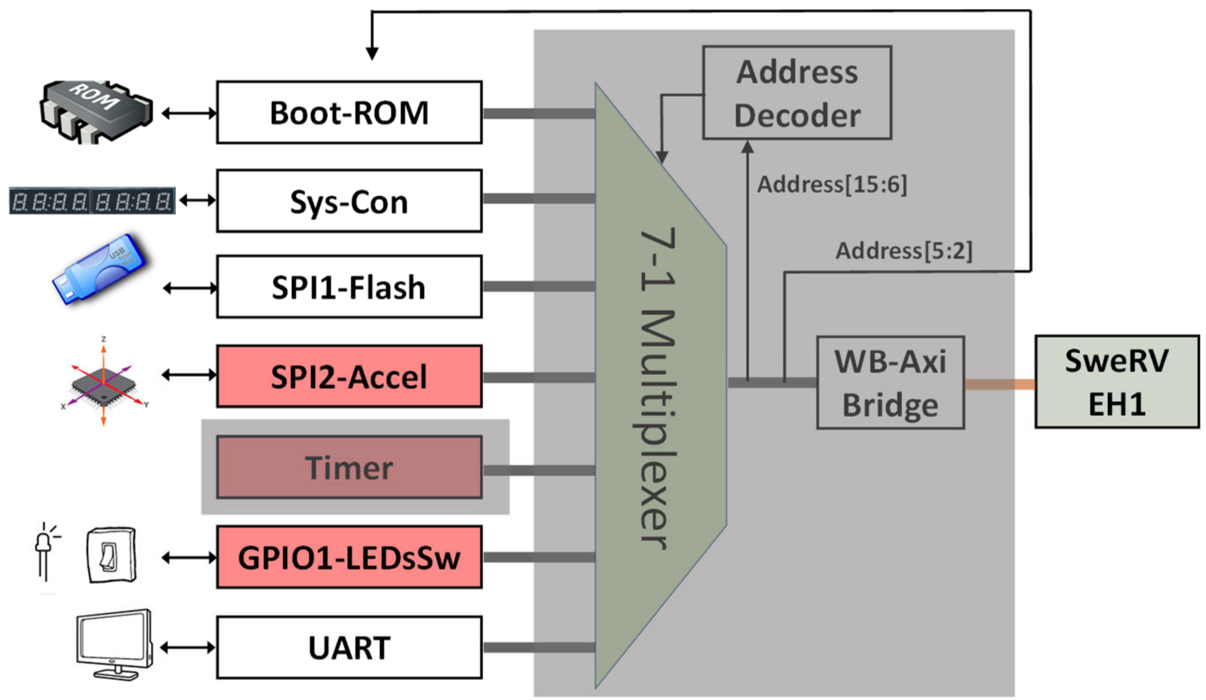
Bit	Access	Reset	Name & Description
0	R/W	0	**EN:** When set, RPTC_CNTR increments.
1	R/W	0	**ECLK:** Selects the clock signal: external clock, through ptc_ecgt (1), or system clock (0).
2	R/W	0	**NEC:** Used for selecting the negative/positive edge and low/high period of the external clock (ptc_ecgt).
3	R/W	0	**OE:** Enables PWM output driver.
4	R/W	0	**SINGLE:** When set, RPTC_CNTR is not incremented after it reaches value equal to the RPTC_LRC value. When cleared, RPTC_CNTR is restarted after it reaches value in the RPTC_LCR register.
5	R/W	0	**INTE:** When set, PTC asserts an interrupt when RPTC_CNTR value is equal to the value of RPTC_LRC or RPTC_HRC. When the signal is cleared, interrupts are masked.
6	R/W	0	**INT:** When read, this bit represents pending interrupt. When it is set, an interrupt is pending. When this bit is written with ‘1’, interrupt request is cleared.
7	R/W	0	**CNTRRST:** When set, RPTC_CNTR is reset. When cleared, normal operation of the counter occurs.
8	R/W	0	**CAPTE:** When set, RPTC_CNTR is captured into RPTC_LRC or RPTC_HRC registers. When cleared, capture function is masked.

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** (0b0**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: 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 ()
);
```

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 Peripherals Support and Board Support Packages (PSP and BSP)
- Interrupt Example

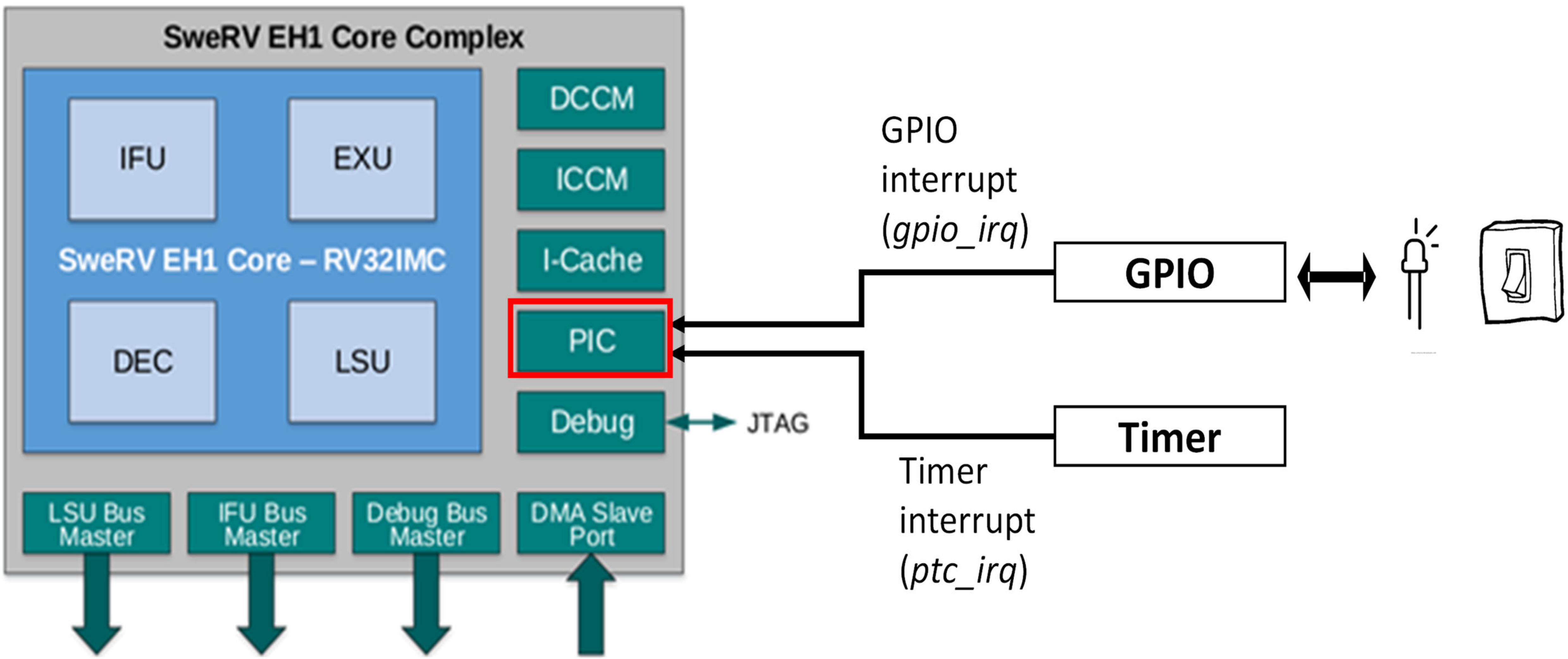
RVfpga Lab 9: Interrupt-Driven I/O Introduction

- **Programmed I/O:**
  - A program continuously polls a value (for example a switch) until the desired value is seen.
  - For example, this method was used to read the switches in prior labs.
  - This ties up the processor by its constantly polling a value – instead of being able to do other useful work.
- **Interrupt-driven I/O:**
  - An event (such as a pin asserting) causes the processor to jump to an interrupt service routine (ISR, also called an interrupt *handler*), which is like an unscheduled function call. The ISR handles the interrupt – for example, reads the value of the switches – and then returns to the regular program.
  - Until that event occurs, the processor can continue doing useful work.

RVfpga Lab 9: Handling Interrupts

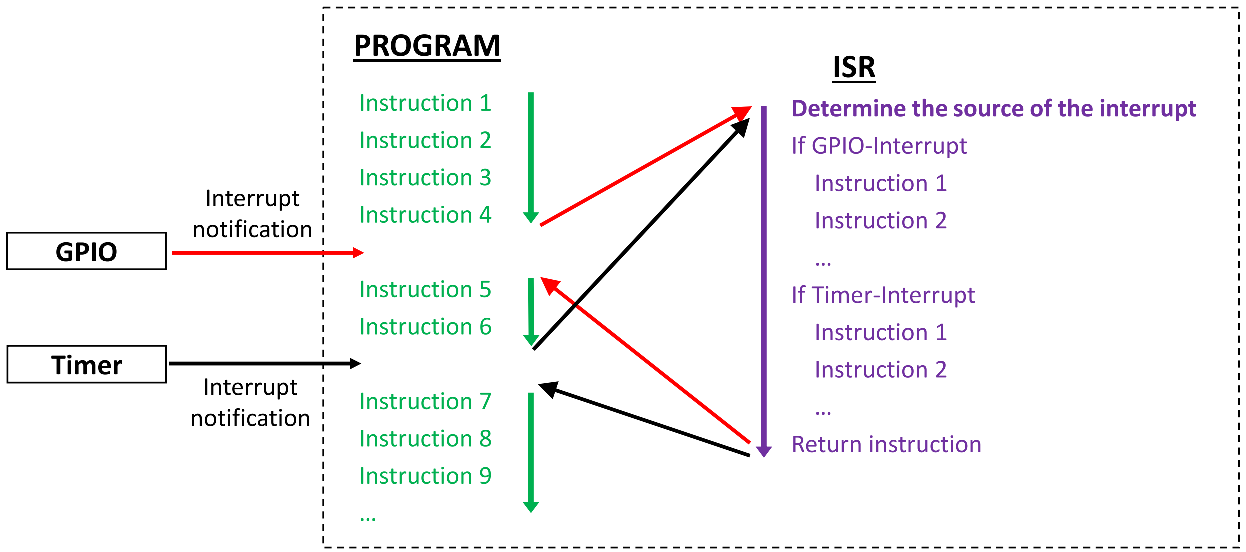
- Interrupts may be caused by **hardware** or **software**
- In this lab, we focus on **hardware interrupts**
- The SweRV EH1 core handles interrupts after RISC-V's PLIC (Platform-level interrupt controller) specification. It is referred to as the Programmable Interrupt Controller (**PIC**). It has: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

RVfpga Lab 9: Interrupt Hardware

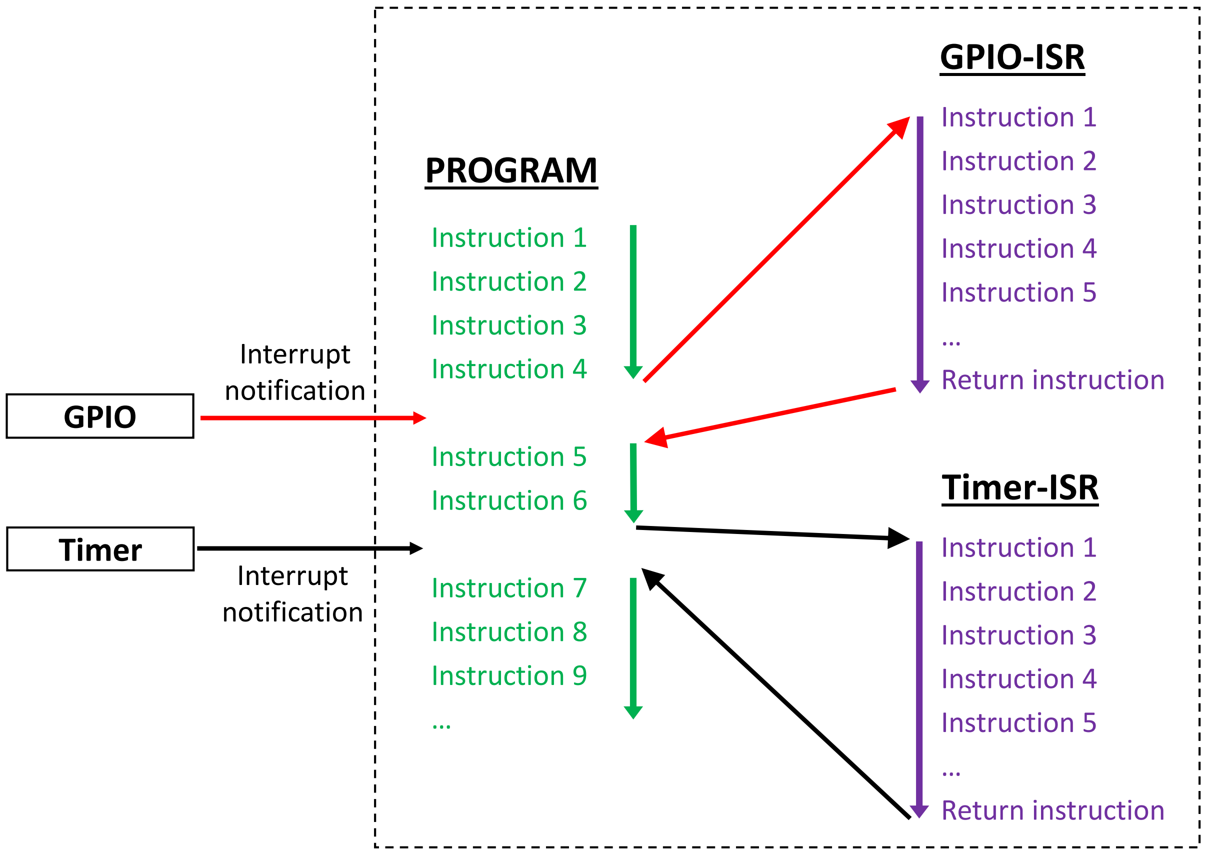


RVfpga Lab 9: Single-vector vs. Multi-vector mode

Single-vector mode example:



Multi-vector mode example:



RVfpga Lab 9: Handling Interrupts

- Using WD's PSP/BSP:
  - Initialize Interrupts using WD'S PSP/BSP
  - Initialize one or more of the 255 interrupts and provide name of ISR
  - Connect peripheral signal that should trigger interrupt with interrupt pin.
  - Enable all interrupts
  - Enable external interrupts

RVfpga Lab 9: Interrupt Example

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

Name	Address	Width	Access	Description
RGPIO_IN	0x80001400	1-32	R	GPIO input data
RGPIO_OUT	0x80001404	1-32	R/W	GPIO output data
RGPIO_OE	0x80001408	1-32	R/W	GPIO output driver enable
RGPIO_INTE	0x8000140C	1-32	R/W	**Interrupt enable**
RGPIO_PTRIG	0x80001410	1-32	R/W	**Type of event that triggers an interrupt**
RGPIO_AUX	0x80001414	1-32	R/W	Multiplex auxiliary inputs to GPIO outputs
RGPIO_CTRL	0x80001418	2	R/W	**Control register**
RGPIO_INTS	0x8000141C	1-32	R/W	**Interrupt status**
RGPIO_ECLK	0x80001420	1-32	R/W	Enable gpio_eclk to latch RGPIO_IN
RGPIO_NEC	0x80001424	1-32	R/W	Select active edge of gpio_eclk

For full code, see: [RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c

RVfpga Lab 9: Interrupt Example

- Setting up GPIO registers for 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)

RVfpga Lab 9: Interrupt Example

- GPIO ISR:

```
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 keep 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 */

  /* Stop the generation of this interrupt (IRQ4) */
  bspClearExtInterrupt(4);
}
```

RVfpga Lab 9: Interrupt Example

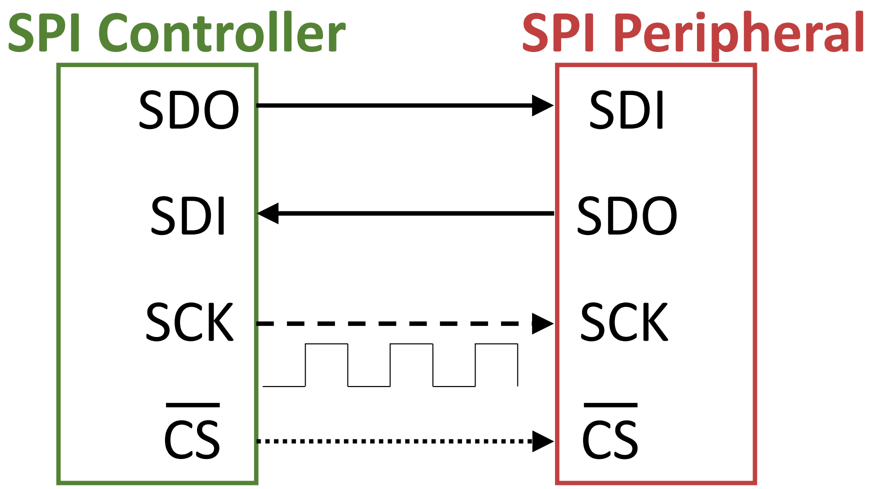
- Connect interrupt 4 (IRQ4) with interrupt from switch, and set interrupt service routine to be GPIO_ISR
- Memory-mapped register 0x80001018 = 0x1: connects GPIO interrupt to IRQ4
- Enable global interrupts

Lab 10: Serial Buses

RVfpga Lab 10: Serial Buses

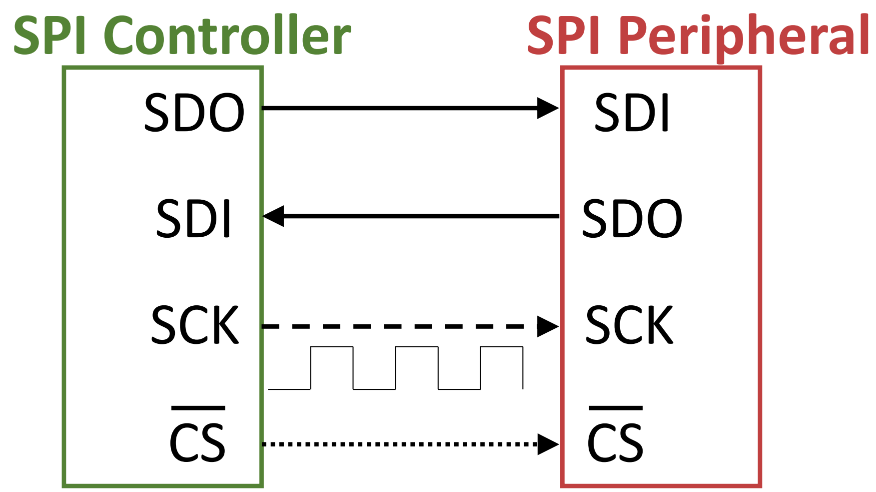
- Serial buses send **one bit at a time**
  - In contrast, parallel buses send **multiple bits at once**
- **Common serial buses**
  - **UART** (universal asynchronous receiver/transmitter)
  - **SPI** (serial peripheral interface)
  - **I2C** (inter-integrated circuit protocol)
- We focus on **SPI** in this lab

RVfpga Lab 10: Serial Buses

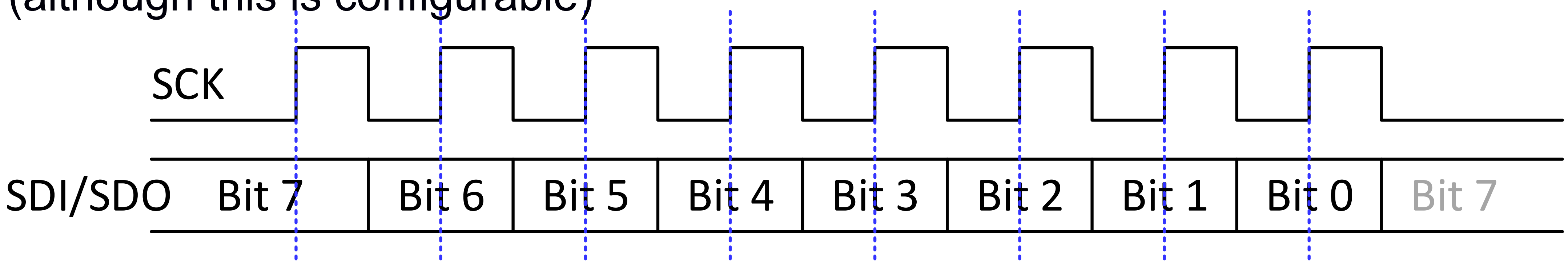


- **Controller:** sends clock, sends & receives data
- **Peripheral:** receives clock, sends & receives data
- **Signals:**
  - **SDO:** Serial Data Out
  - **SDI:** Serial Data In
  - **SCK:** SPI clock
  - **CSbar:** low-asserted chip select

RVfpga Lab 10: Serial Buses



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



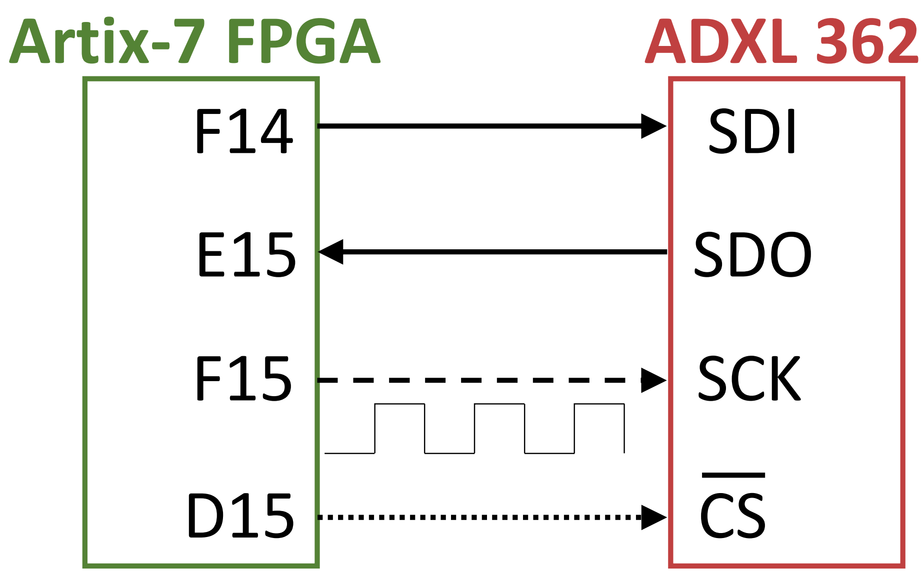
RVfpga Lab 10: Rvfpga System's SPI Module

- 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

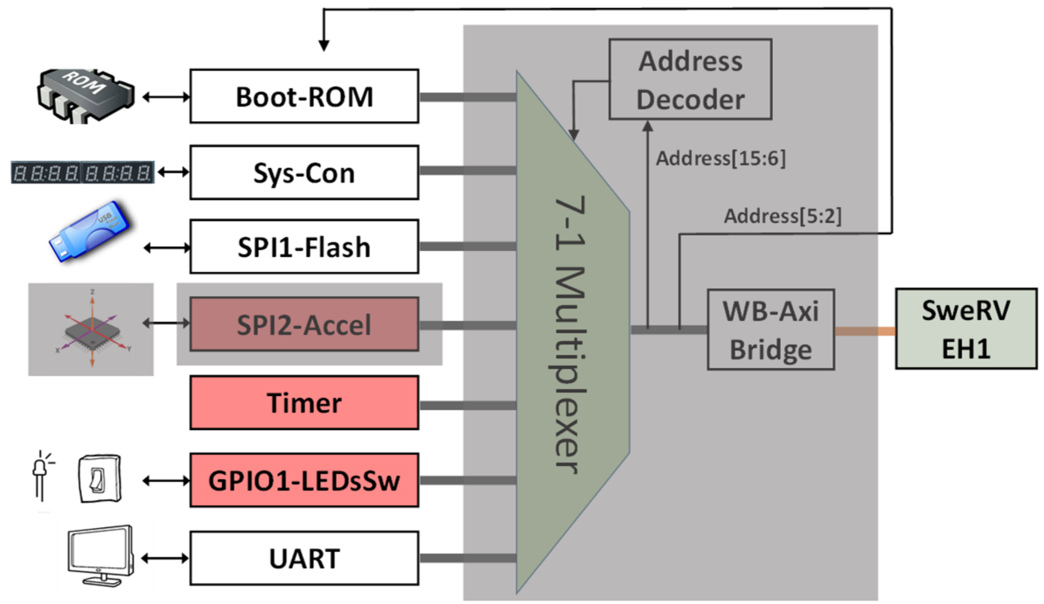
RVfpga Lab 10: ADXL362 Accelerometer

- The Nexys A7 board includes an 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: 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

```
##Accelerometer
set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
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));
```