香港中文大學
The Chinese University of Hong Kong

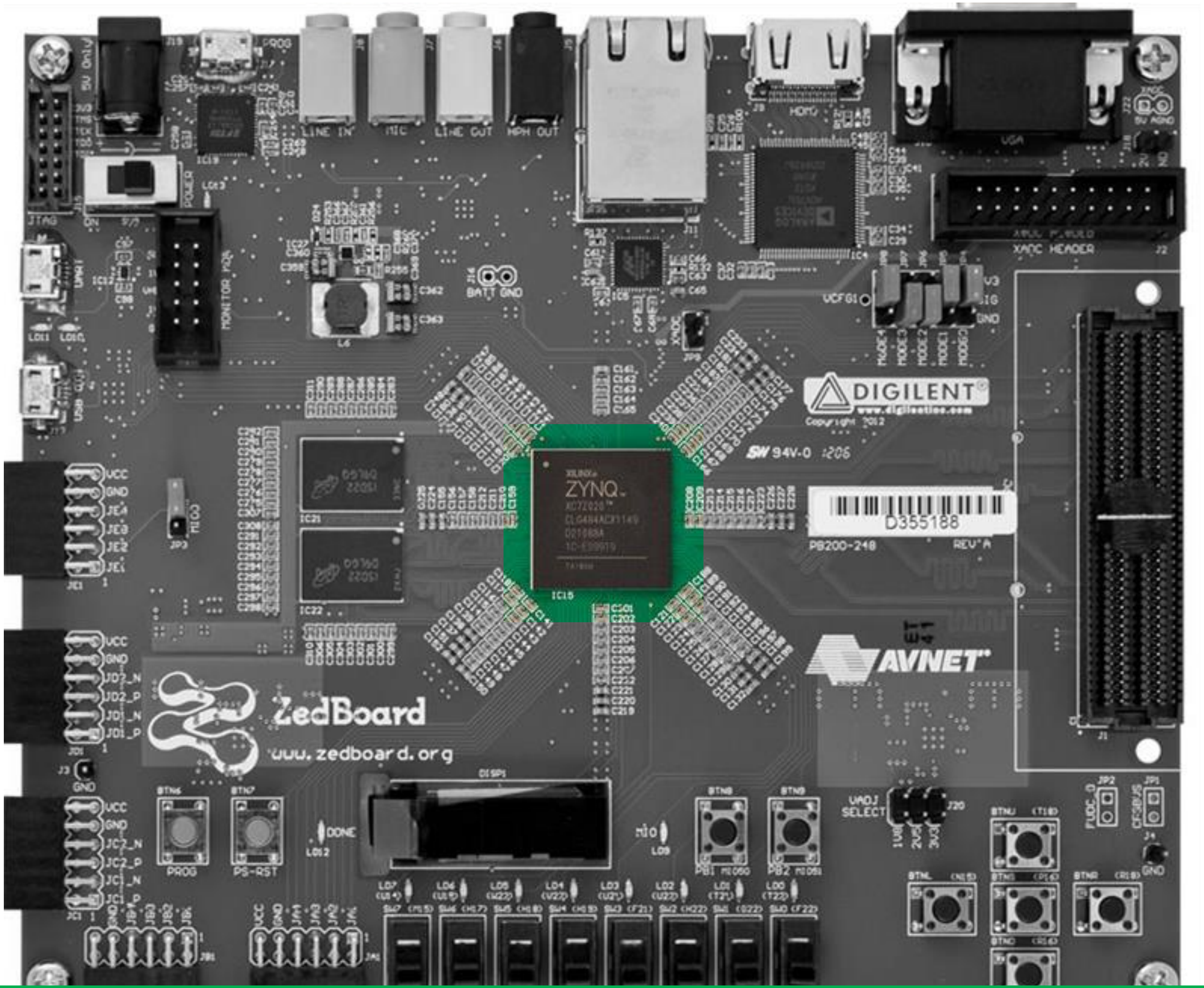
CENG3430 Rapid Prototyping of Digital Systems

Lecture 07:
Rapid Prototyping (I) – Integration of ARM and FPGA

Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Rapid Prototyping with Zynq Zedboard?



ZedBoard features a ZC7Z020 “Zynq” device.

Outline

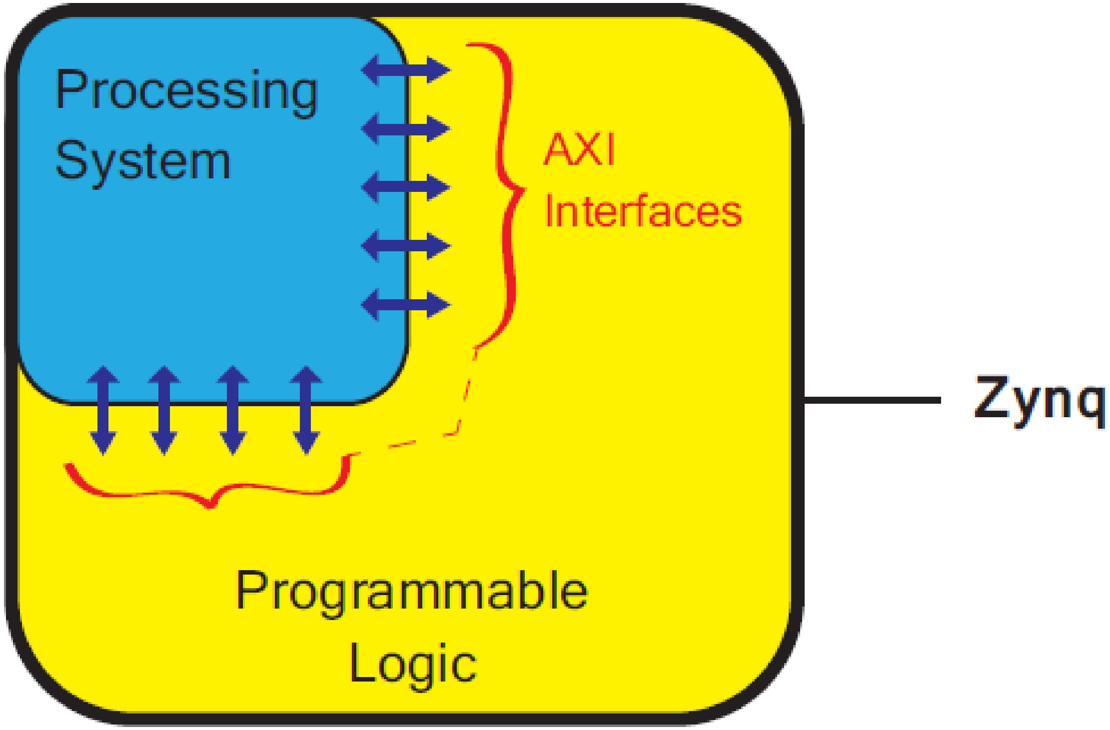
- Rapid Prototyping with Zynq

- Rapid Prototyping (I): Integration of ARM and FPGA
  - PART 1: IP Block Design (Xilinx Vavido)
    - ① IP Block Creation
    - ② IP Integration
    - ③ HDL Wrapper
    - ④ Generate Bitstream
  - PART 2: ARM Programming (Xilinx SDK)
    - ⑤ ARM Programming
    - ⑥ Launch on Hardware

- Case Study: Software Stopwatch

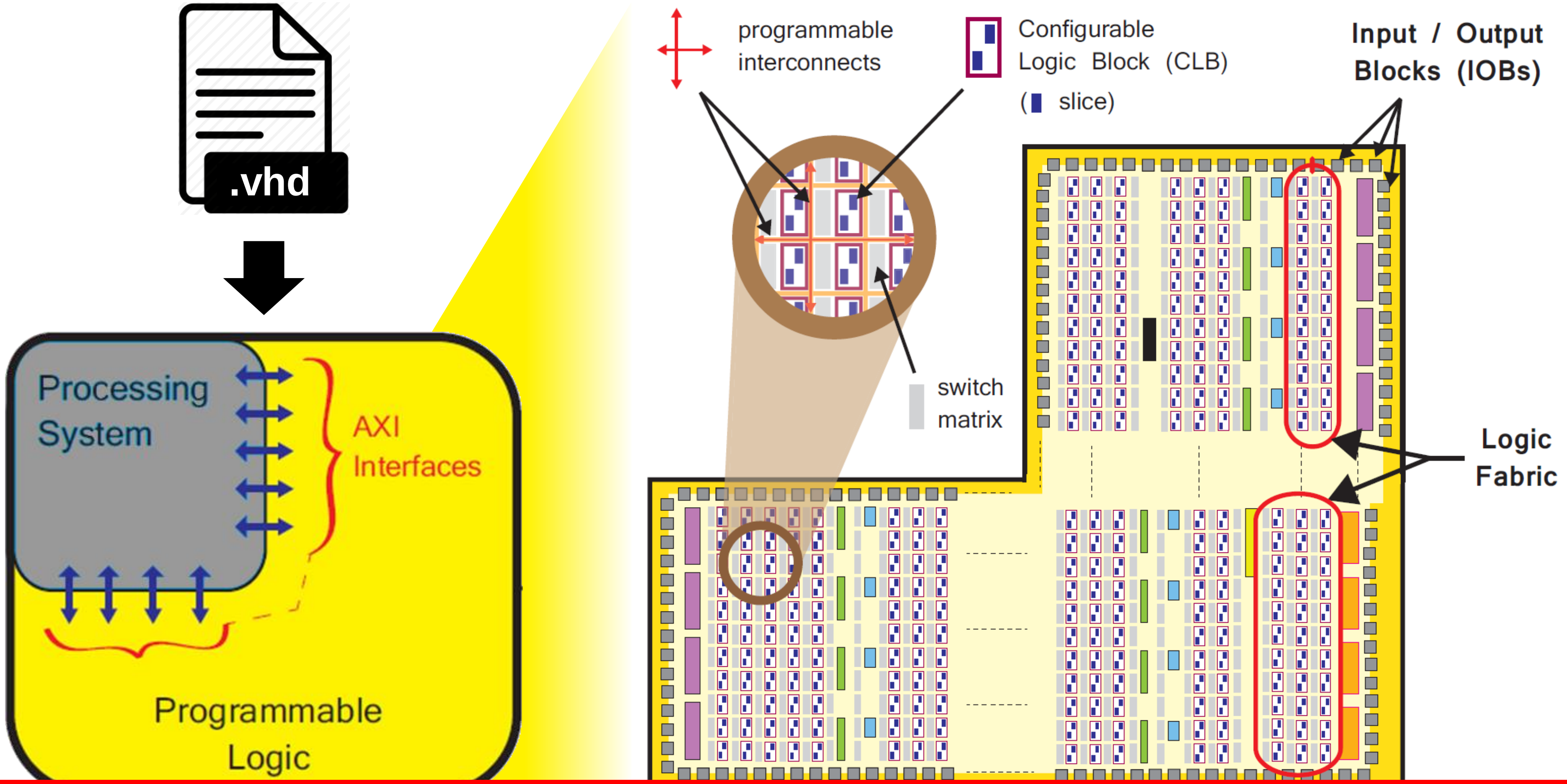
What is Zynq?

- The defining features of Zynq family:
  - **Processing System (PS)**: Dual-core ARM Cortex-A9 CPU
  - **Programmable Logic (PL)**: Equivalent traditional FPGA
  - **Advanced eXtensible Interface (AXI)**: High bandwidth, low latency connections between PS and PL.



Prototyping with FPGA: PL Only

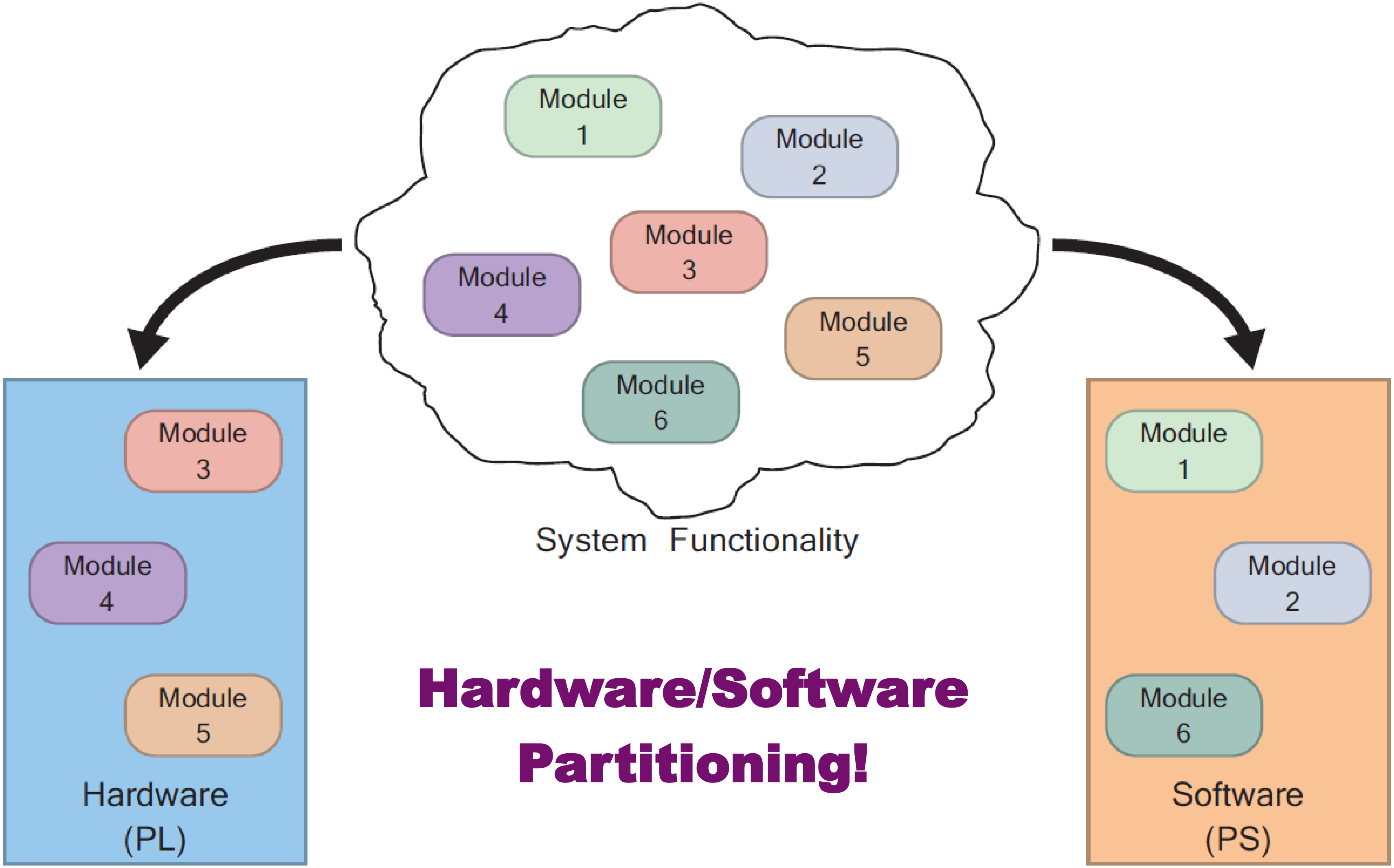
- However, so far, our designs are implemented ***only*** using the **programmable logic** of Zynq with **VHDL**.



It is challenging to implement "complicated" design!

Key to Rapid Prototyping?

- **PL** and **PS** shall each be used for *what they do best*.



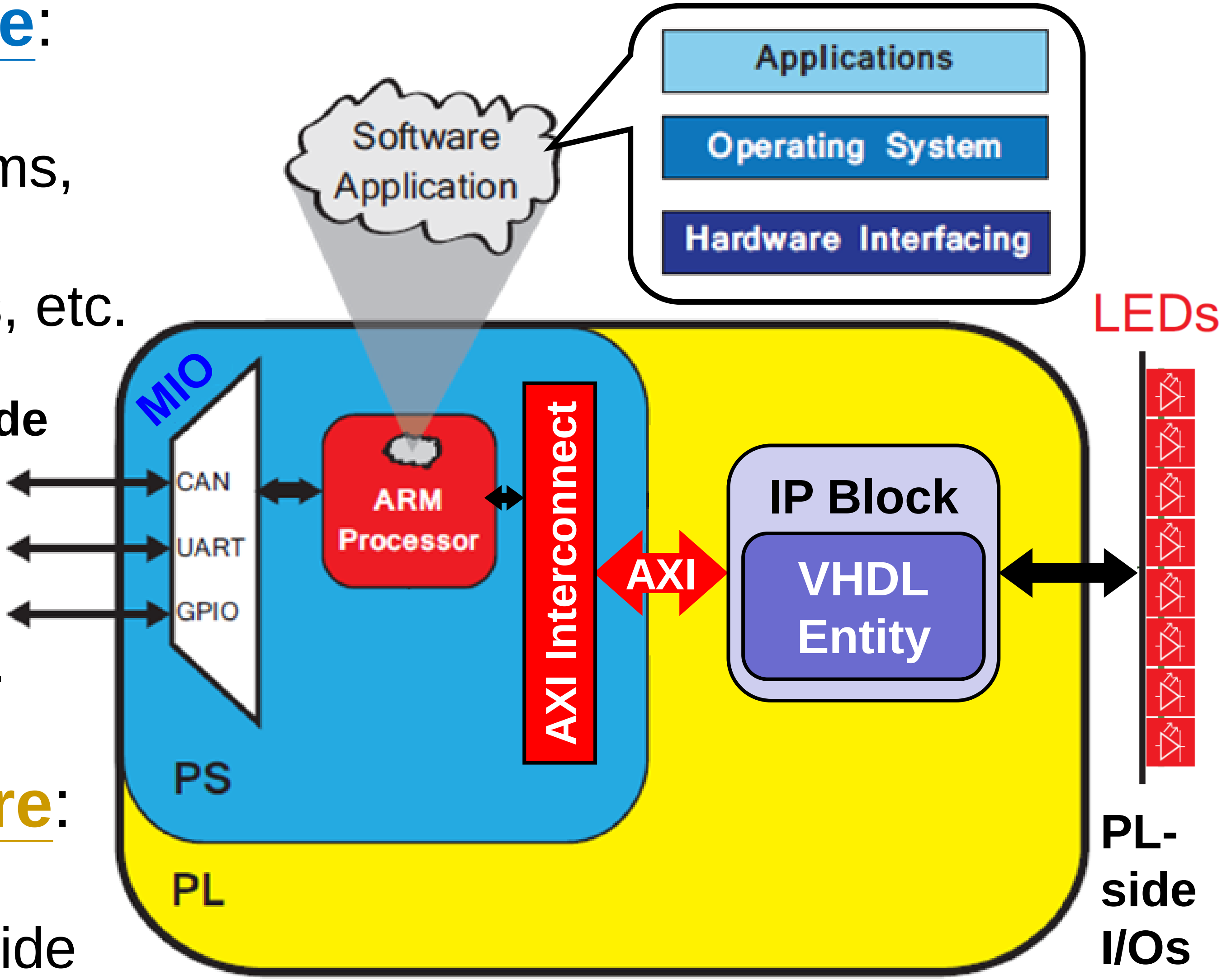
Hardware/Software Partitioning!

Rapid Prototyping with Zynq: PS + PL

PS for Software:
general purpose sequential programs, operating system, GUIs, applications, etc.

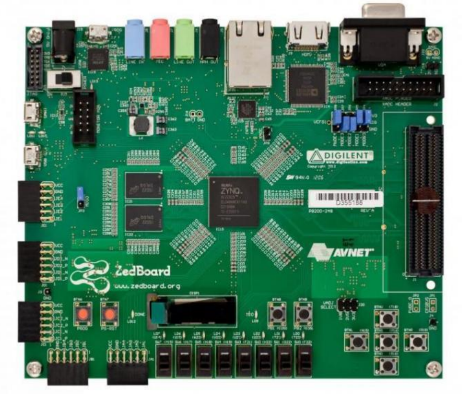
AXI:
a means of communication between **PS** & **PL**.

PL for Hardware:
intensive data computation, PL-side peripheral control, etc.



Note: ***AXI*** *stands for* ***A****dvanced e****X****tensible* ***I****nterface.*

Prototyping Styles with Zynq ZedBoard



Xilinx SDK (C/C++)		**Bare-metal Applications**	**Applications**	**SDK** (Shell, C, Java, …)
			Operating System	**Process System (PS)**
		Board Support Package	Board Support Package	*software*
Xilinx Vivado (HDL)	**Programmable Logic Design**	**Hardware Base System**	Hardware Base System	*hardware* **Program Logic (PL)**
	Style 1) **FPGA (PL)**	**Style 2)** **ARM + FPGA**	**Style 3)** **Embedded OS**	
	VHDL or Verilog Programming	ARM Programming & IP Block Design	Shell Script Programming	

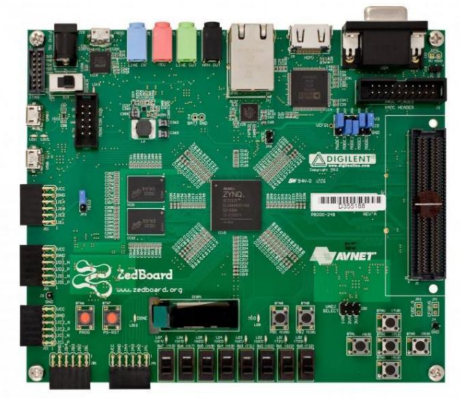
Outline

- Rapid Prototyping with Zynq

- Rapid Prototyping (I): Integration of ARM and FPGA
  - PART 1: IP Block Design (Xilinx Vavido)
    - ① IP Block Creation
    - ② IP Integration
    - ③ HDL Wrapper
    - ④ Generate Bitstream
  - PART 2: ARM Programming (Xilinx SDK)
    - ⑤ ARM Programming
    - ⑥ Launch on Hardware

- Case Study: Software Stopwatch

Integration of ARM and FPGA (1/2)



Xilinx SDK (C/C++)

Xilinx Vivado (HDL)

Programmable Logic Design

Style 1) FPGA (PL)

VHDL or Verilog Programming

Bare-metal Applications

Board Support Package

Hardware Base System

Style 2) ARM + FPGA

ARM Programming & IP Block Design

Applications

Operating System

Board Support Package

Hardware Base System

Style 3) Embedded OS

Shell Script Programming

SDK (Shell, C, Java, …)

Process System (PS)

software

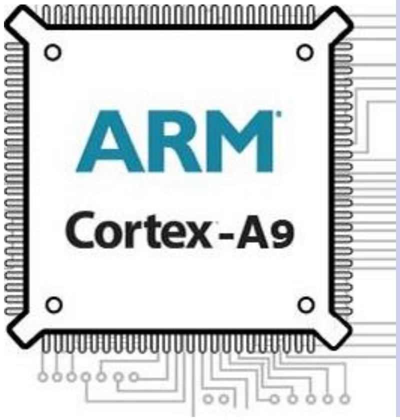
hardware

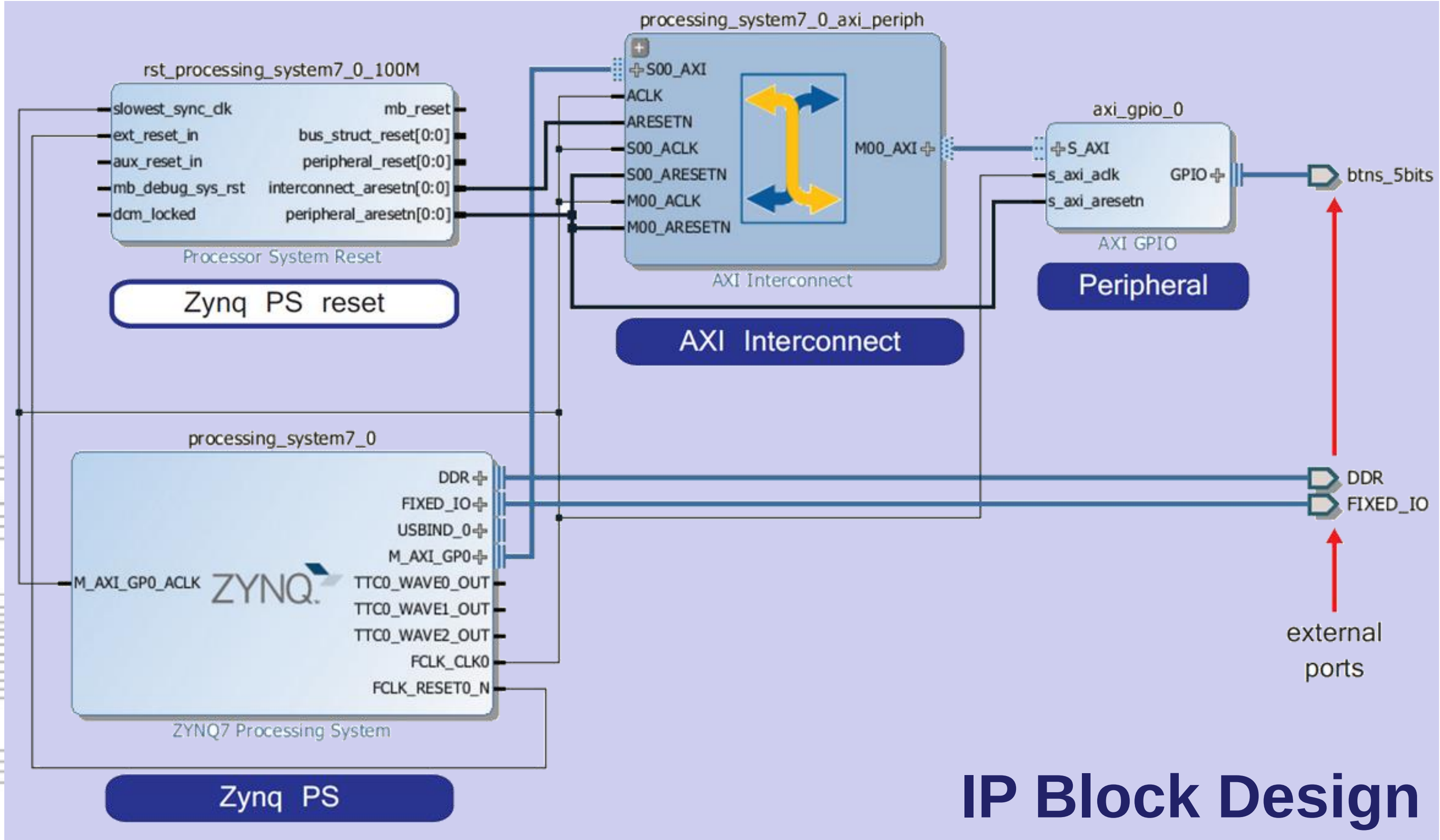
Program Logic (PL)

Integration of ARM and FPGA (2/2)

- To integrate ARM and FPGA, we need to do:
  1. **“IP Block” Design** on Xilinx Vivado using HDL
  2. **ARM Programming** on Xilinx SDK using C/C++

ARM Programming

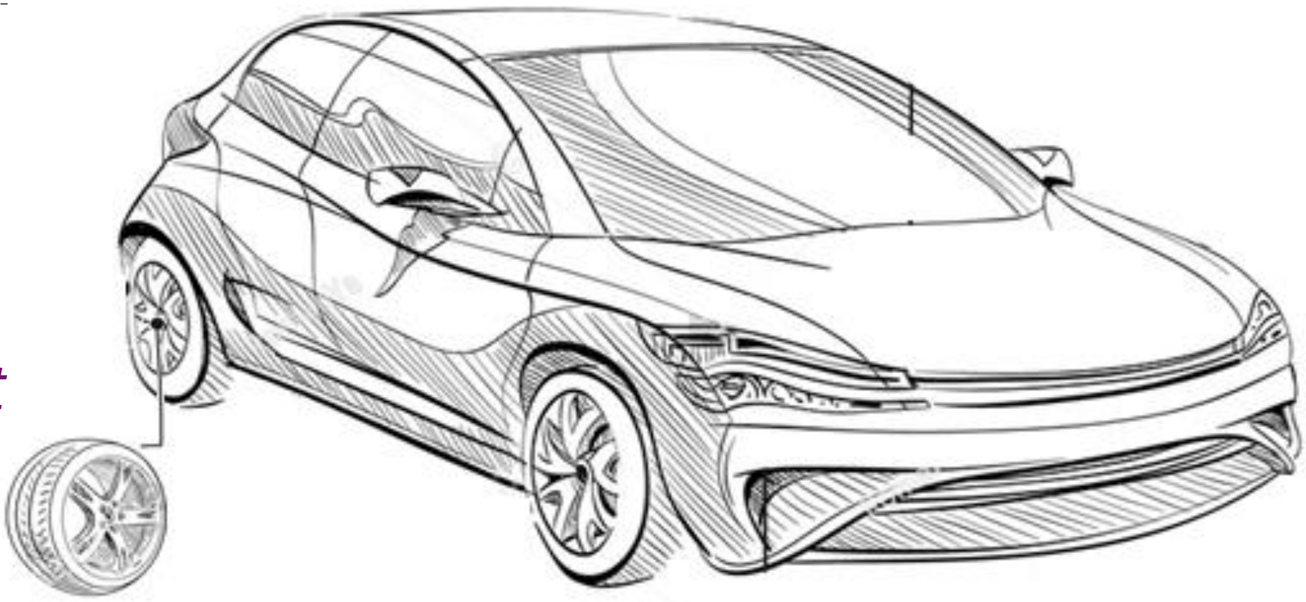




IP Block Design

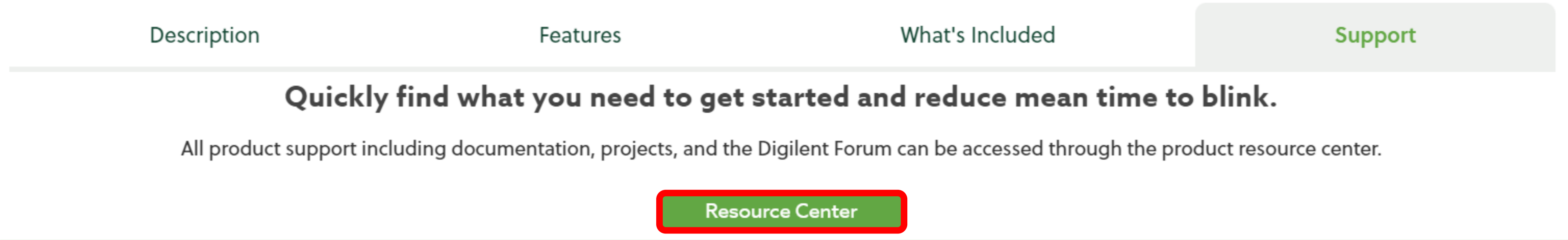
Intellectual Property (IP) Block

- **IP Block** (or **IP Core**) is a hardware specification used to configure the resources of an FPGA.
  - – IP allows system designers to pick-and-choose from a wide range of pre-developed, re-useable **design blocks**.
  - – IP saves development time, as well as provides guaranteed functionality without the need for extensive testing.
- An Analogy



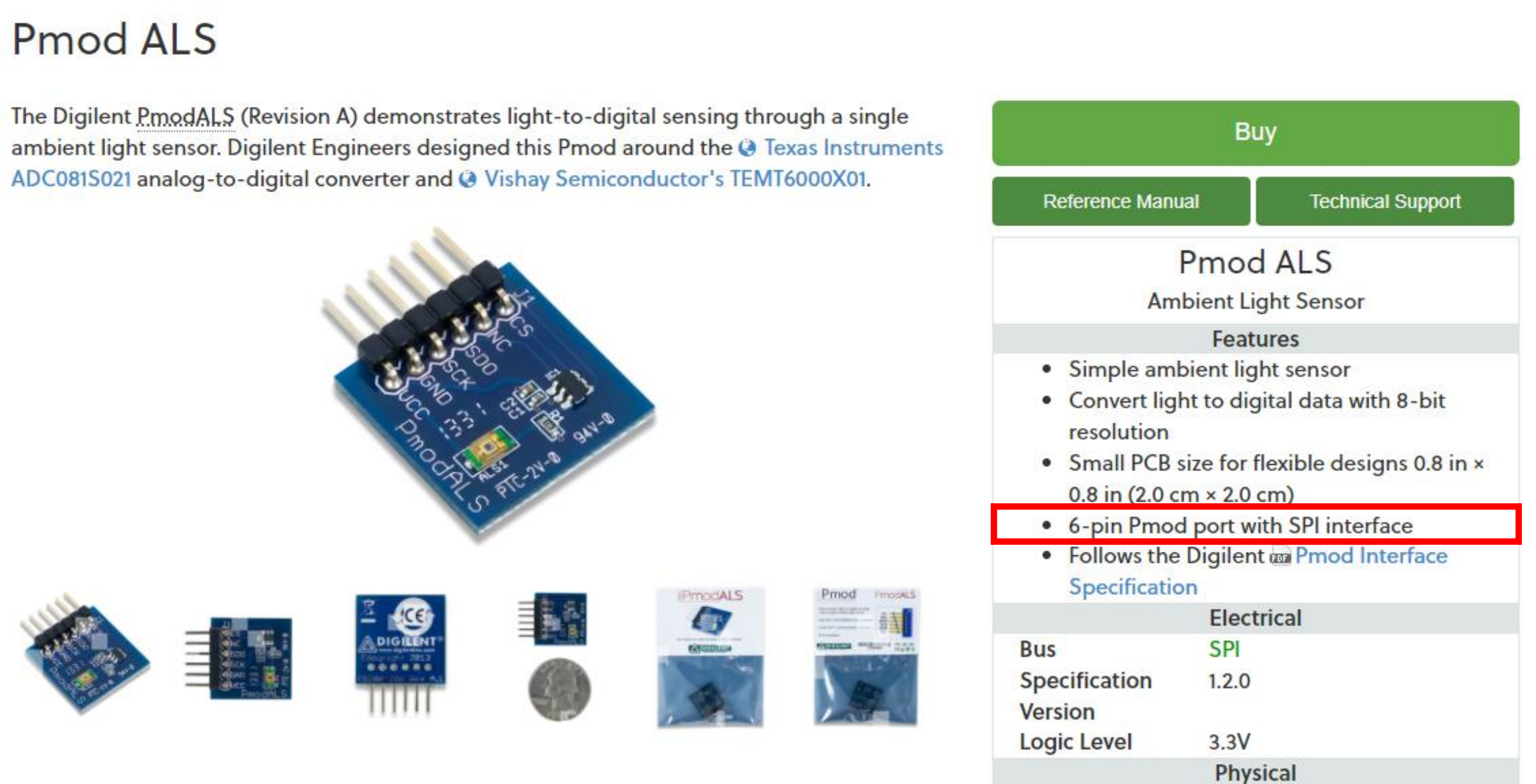
Why reinvent the wheel?

Revisit: Pmod ALS (1/2)



Quickly find what you need to get started and reduce mean time to blink.

All product support including documentation, projects, and the Digilent Forum can be accessed through the product resource center.



Pmod ALS

The Digilent PmodALS (Revision A) demonstrates light-to-digital sensing through a single ambient light sensor. Digilent Engineers designed this Pmod around the Texas Instruments ADC081S021 analog-to-digital converter and Vishay Semiconductor's TEMT6000X01.

Pmod ALS

Ambient Light Sensor

Features

- Simple ambient light sensor
- Convert light to digital data with 8-bit resolution
- Small PCB size for flexible designs 0.8 in × 0.8 in (2.0 cm × 2.0 cm)
- 6-pin Pmod port with SPI interface
- Follows the Digilent Pmod Interface Specification

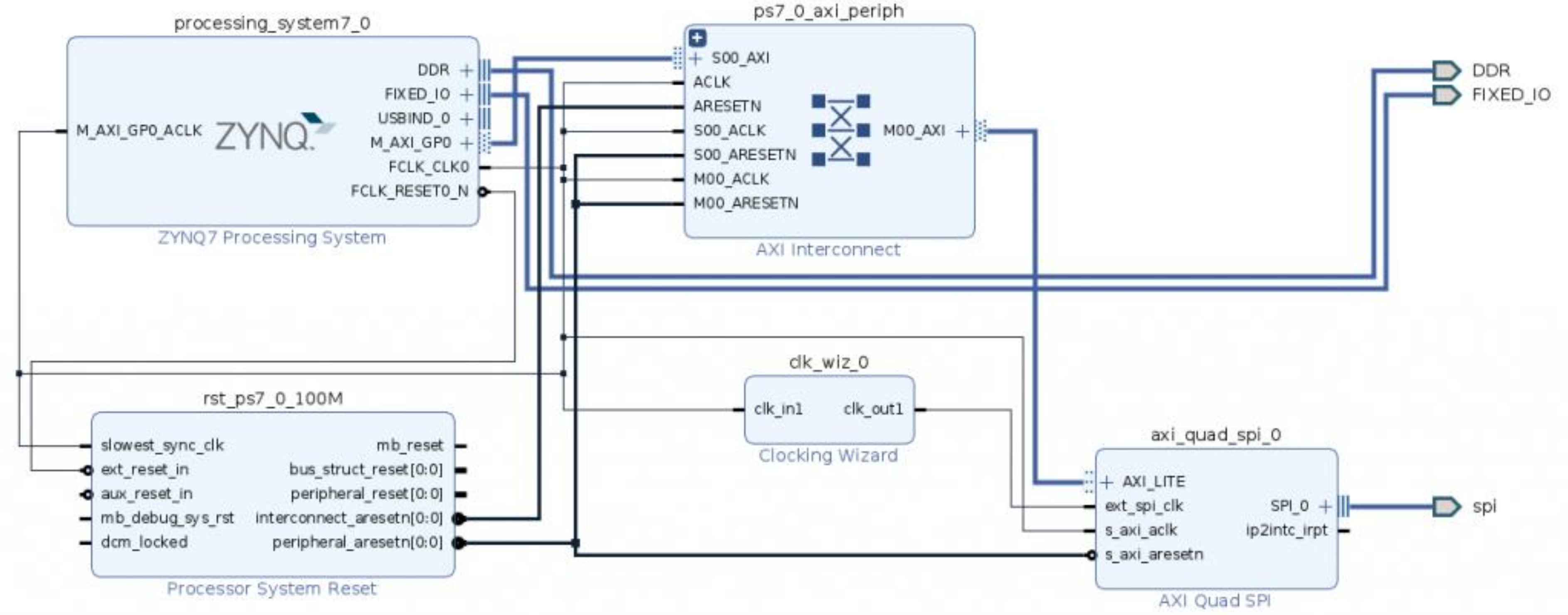
Electrical

Bus	SPI
Specification Version	1.2.0
Logic Level	3.3V

Physical

Revisit: Pmod ALS (2/2)

- The host board can alternatively communicate with Pmod ALS using the IP block (e.g., AXI Quad SPI).
  - Similar methods can be applied to other SPI, I2C, or UART based Pmod devices as well.



https://www.makarenalabs.com/spi-i2c-uart-on-pynq-a-pl-approach/
https://www.xilinx.com/products/intellectual-property/axi_quadspi.html

Sources of IP Block

- **IP Libraries**: Xilinx provides an extensive catalogue of IP cores for the Zynq-7000 AP family.
  - Ranging from building blocks (such as FIFOs and arithmetic operators) up to fully functional processor blocks.

- **Third-party IP** is also available, both commercially and from the open-source community.

🎯**IP Creation**: The final option is to create by yourself.

- The most traditional method of IP creation is for it to be developed in HDLs (such as VHDL or Verilog).
- Recently, other methods of IP creation have also been introduced to Vivado, such as High Level Synthesis (HLS).

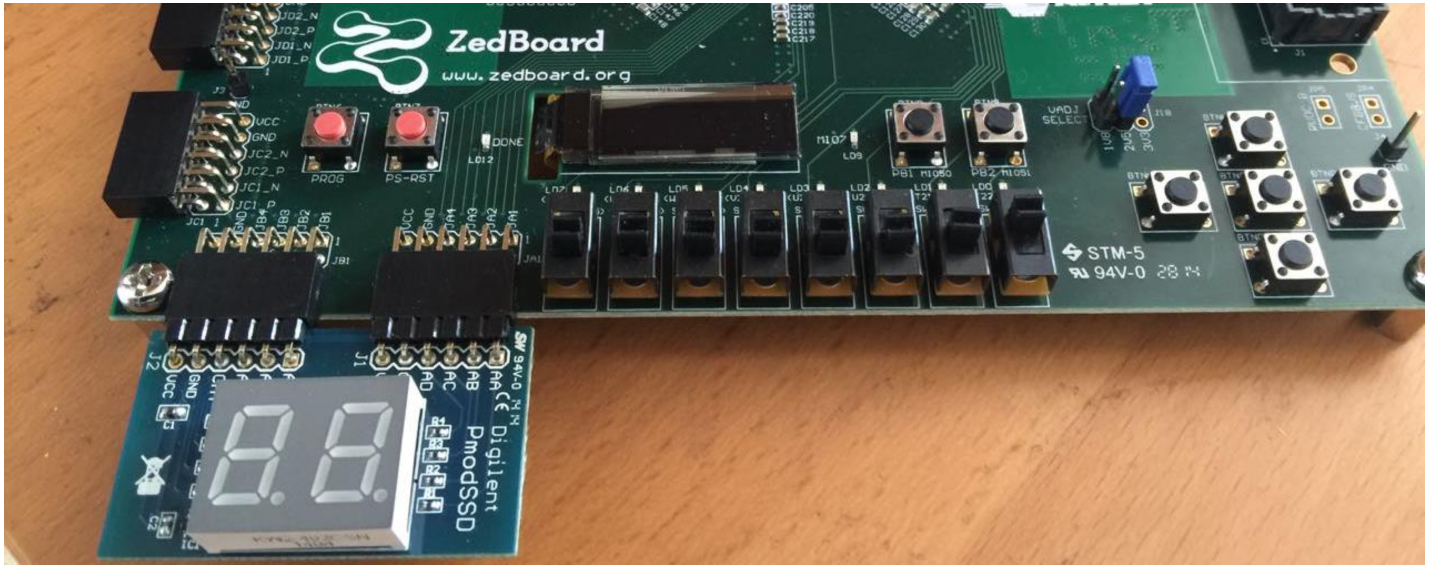
Outline

- Rapid Prototyping with Zynq

- Rapid Prototyping (I): Integration of ARM and FPGA
  - PART 1: IP Block Design (Xilinx Vavido)
    - ① IP Block Creation
    - ② IP Integration
    - ③ HDL Wrapper
    - ④ Generate Bitstream
  - PART 2: ARM Programming (Xilinx SDK)
    - ⑤ ARM Programming
    - ⑥ Launch on Hardware

- Case Study: Software Stopwatch

Case Study: Stopwatch



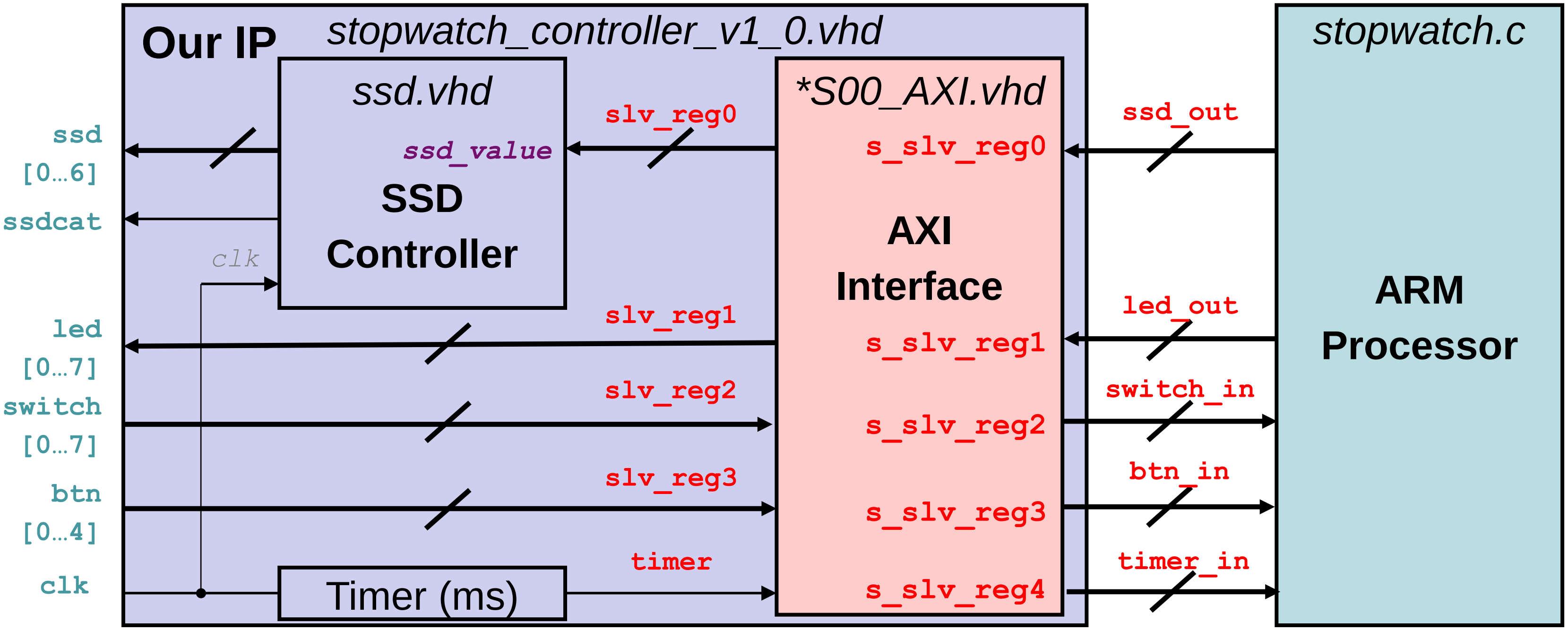
```
entity stopwatch is
port(     clk: in STD_LOGIC;
       switch: in STD_LOGIC_VECTOR (7 downto 0);
          btn: in STD_LOGIC_VECTOR (4 downto 0);
          led: out STD_LOGIC_VECTOR (7 downto 0);
          ssd: out STD_LOGIC_VECTOR (6 downto 0);
       ssdsel: out STD_LOGIC );
end stopwatch;
```

Task: Count down from the input number (XY) to (00)

Hardware vs. Software Stopwatch

- We can build a hardware stopwatch in which the FPGA (PL) is responsible for ***both***:
  - Interfacing with hardware (**`clk`**/**`switch`**/**`btn`** /**`led`**/**`PmodSSD`**);
  - Controlling the values to be shown on **`led`**/**`PmodSSD`** based on user inputs or events.
- In Lab 07, we are going to develop a software stopwatch through ARM-FPGA integration:
  - **Hardware**: FPGA (PL) is ***only*** responsible for hardware interfacing with **`clk`**/**`switch`**/**`btn`**/**`led`**/**`PmodSSD`**.
    - We can reuse the hardware interfacing for different designs.
  - **Software**: ARM (PS) determines the values to be shown on **`led`**/**`PmodSSD`** based on user inputs or events.
    - We can easily realize a complex control logic via ARM programming.

Overall Design of Software Stopwatch



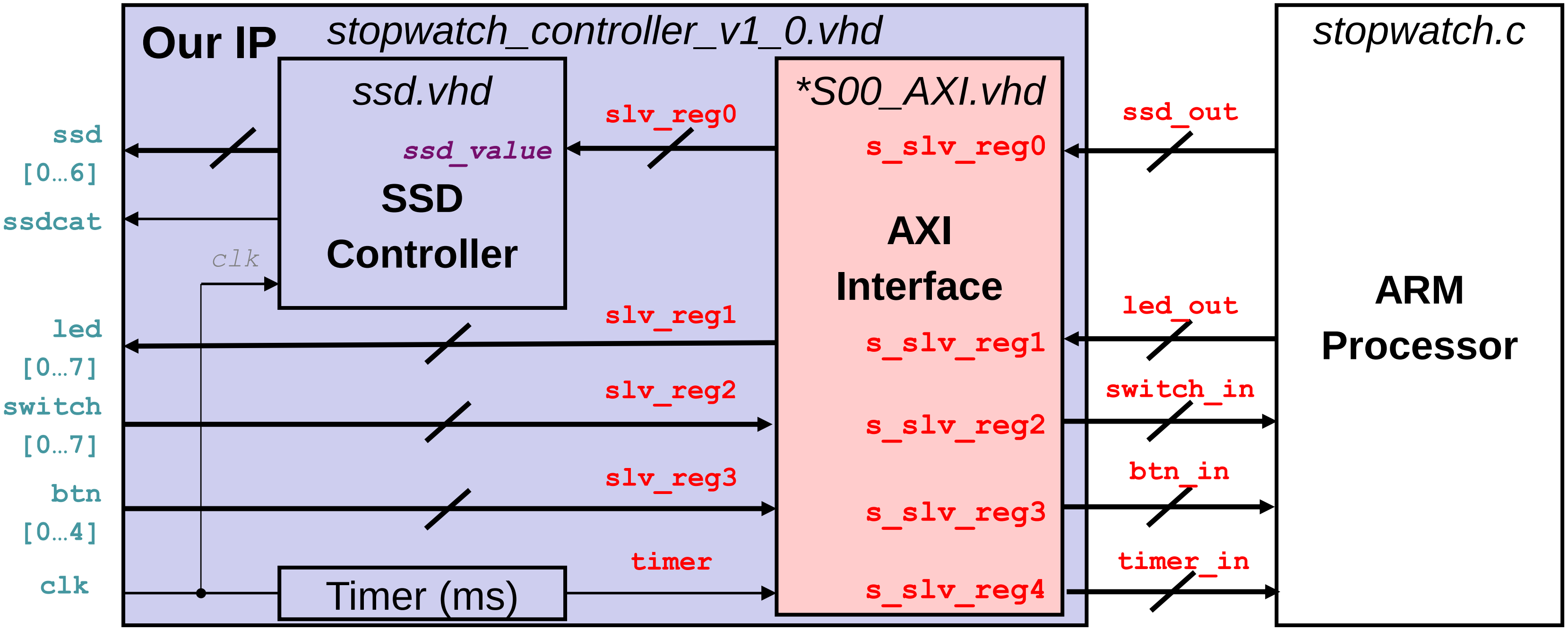
- **Hardware**: The **stopwatch IP block** is responsible for hardware interfacing with **`clk`/`switch`/`btn`/`led`/`PmodSSD`**.
- **Software**: The **ARM processor** determines the values to be shown on **`led`** and **`PmodSSD`** based on user inputs or events.
- The **ARM processor** communicates with the **IP block** via the **AXI slave registers**.

Key Steps of ARM-FPGA Integration

- **PART 1: IP Block Design** (using Xilinx Vivado)
  - ① **Create** and **Package** the **PL** logic blocks into **intellectual property (IP) block** with **AXI4 Interface**.
    - With AXI4, data can be exchanged via shared 32-bit registers.
  - ② **Integrate** the customized (or pre-developed) IP block with ZYNQ7 Processing System (**PS**) via **IP Block Design**.
    - Vivado can auto-connect IP block and ARM core via AXI interface.
  - ③ **Create HDL Wrapper** and **Add Constraints** to automatically generate the HDL code (VHDL or Verilog).
  - ④ **Generate and Program Bitstream** into the board.

- **PART 2: ARM Programming** (using Xilinx SDK)
  - ⑤ **Design** the bare-metal **application** in C/C++ language.
  - ⑥ **Launch on Hardware** (GDB): Run the code on ARM core.

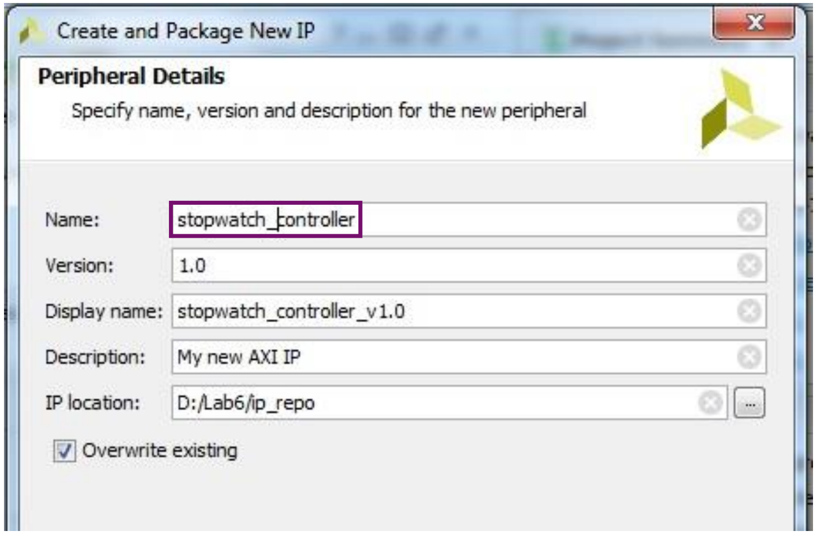
PART 1: IP Block Design



- Five AXI slave registers are used for data exchange:
  - `s_slv_reg0`: value to be displayed on the Pmod SSD (←)
  - `s_slv_reg1`: value to be displayed on the LEDs (←)
  - `s_slv_reg2`: value inputted from the switches (→)
  - `s_slv_reg3`: value inputted from the buttons (→)
  - `s_slv_reg4`: the number of milliseconds elapsed (→)

① IP Block Creation: New IP

- According to our design specification, we need to have five AXI registers for exchanging data:


Create and Package New IP

Peripheral Details

Specify name, version and description for the new peripheral

Name: stopwatch_controller

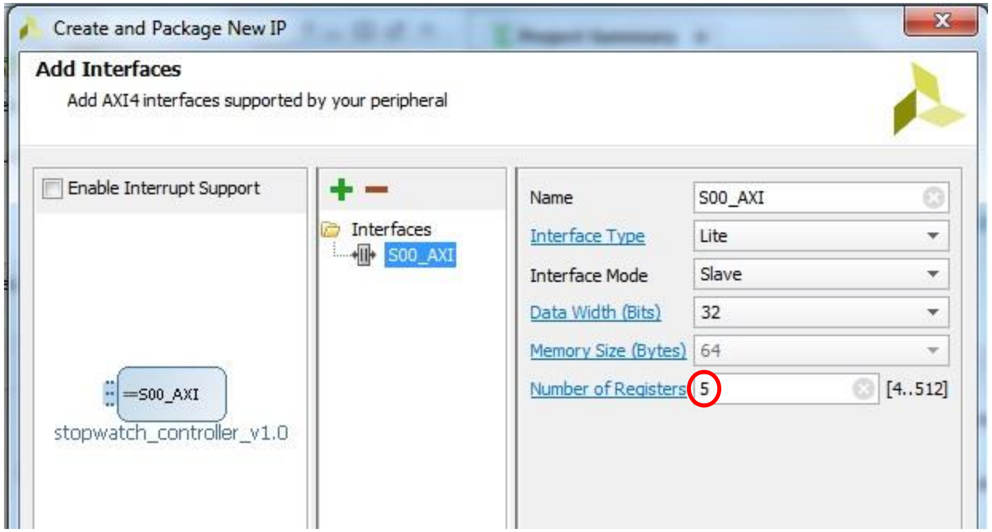
Version: 1.0

Display name: stopwatch_controller_v1.0

Description: My new AXI IP

IP location: D:/Lab6/ip_repo

☑ Overwrite existing


Create and Package New IP

Add Interfaces

Add AXI4 interfaces supported by your peripheral

☐ Enable Interrupt Support

Name: S00_AXI

Interface Type: Lite

Interface Mode: Slave

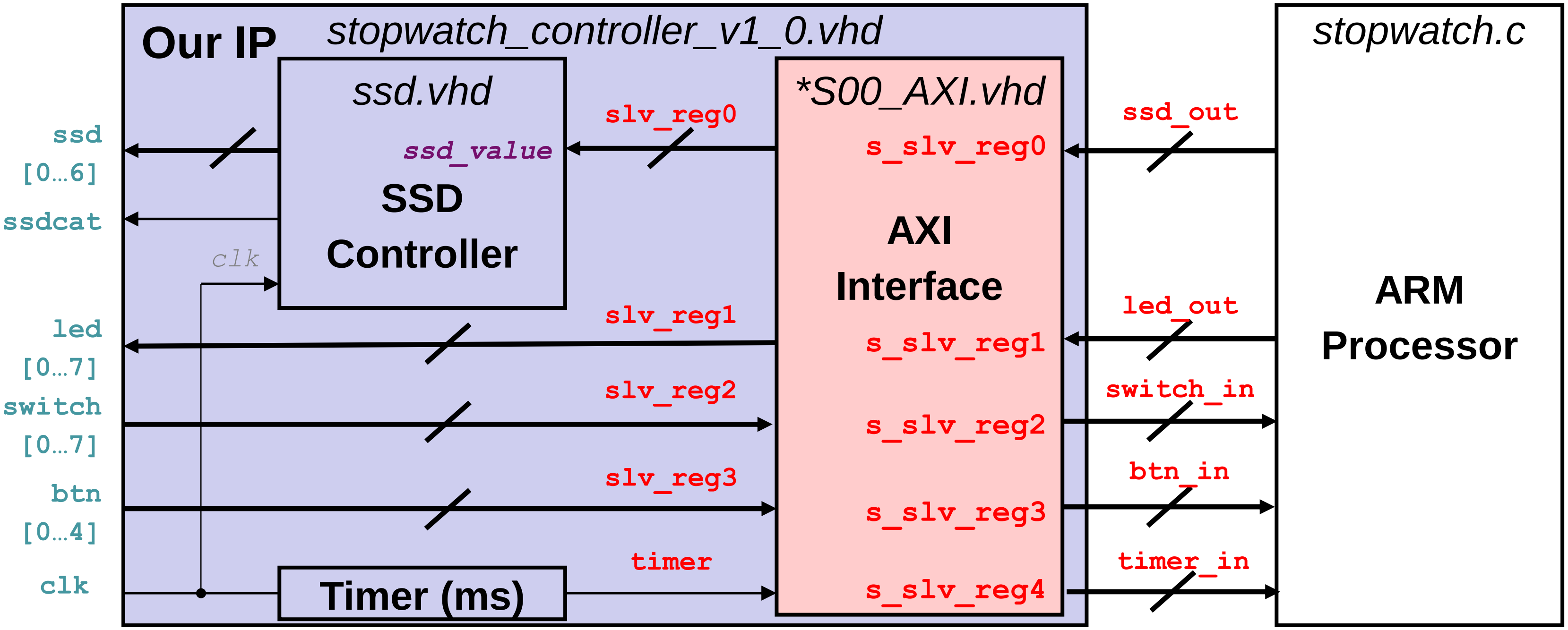
Data Width (Bits): 32

Memory Size (Bytes): 64

Number of Registers: 5 [4..512]

- Two .vhd templates will be generated automatically:
  - **stopwatch_controller_v1_0.vhd**: This file instantiates the AXI-Lite interface and contain the required functionality.
  - **stopwatch_controller_v1_0_S00_AXI.vhd**: This file contains only the AXI-Lite bus functionality.

① IP Block Creation: Implementation (1/2



- **stopwatch_controller_v1_0.vhd:**
  - Define the design interface, implement the required functionality (including ssd.vhd for Pmod SSD), and instantiate the AXI interface.
- **stopwatch_controller_v1_0_S00_AXI.vhd:**
  - Describe a five-register AXI interface for this IP block.

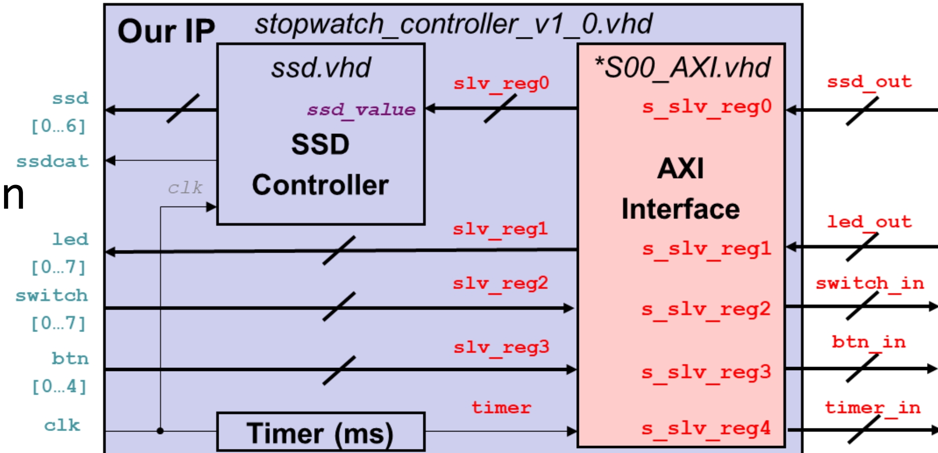
(Note: Please refer to the lab sheet for detailed instructions.)

① IP Block Creation: Implementation (2/2)

```
stopwatch_controller_v1_0_S00_AXI
  port map (
    s_slv_reg0 => slv_reg0, -- ssd
    s_slv_reg1 => slv_reg1, -- led
    s_slv_reg2 => slv_reg2, -- sw
    s_slv_reg3 => slv_reg3, -- btn
    s_slv_reg4 => timer );  -- clk
```

```
ssd_controller
  generic map (
    cat_period => C_MS_LIMIT )
  port map (
    clk => clk,
    value => ssd_value,
    ssd => ssd,
    ssdcat => ssdcat );
```

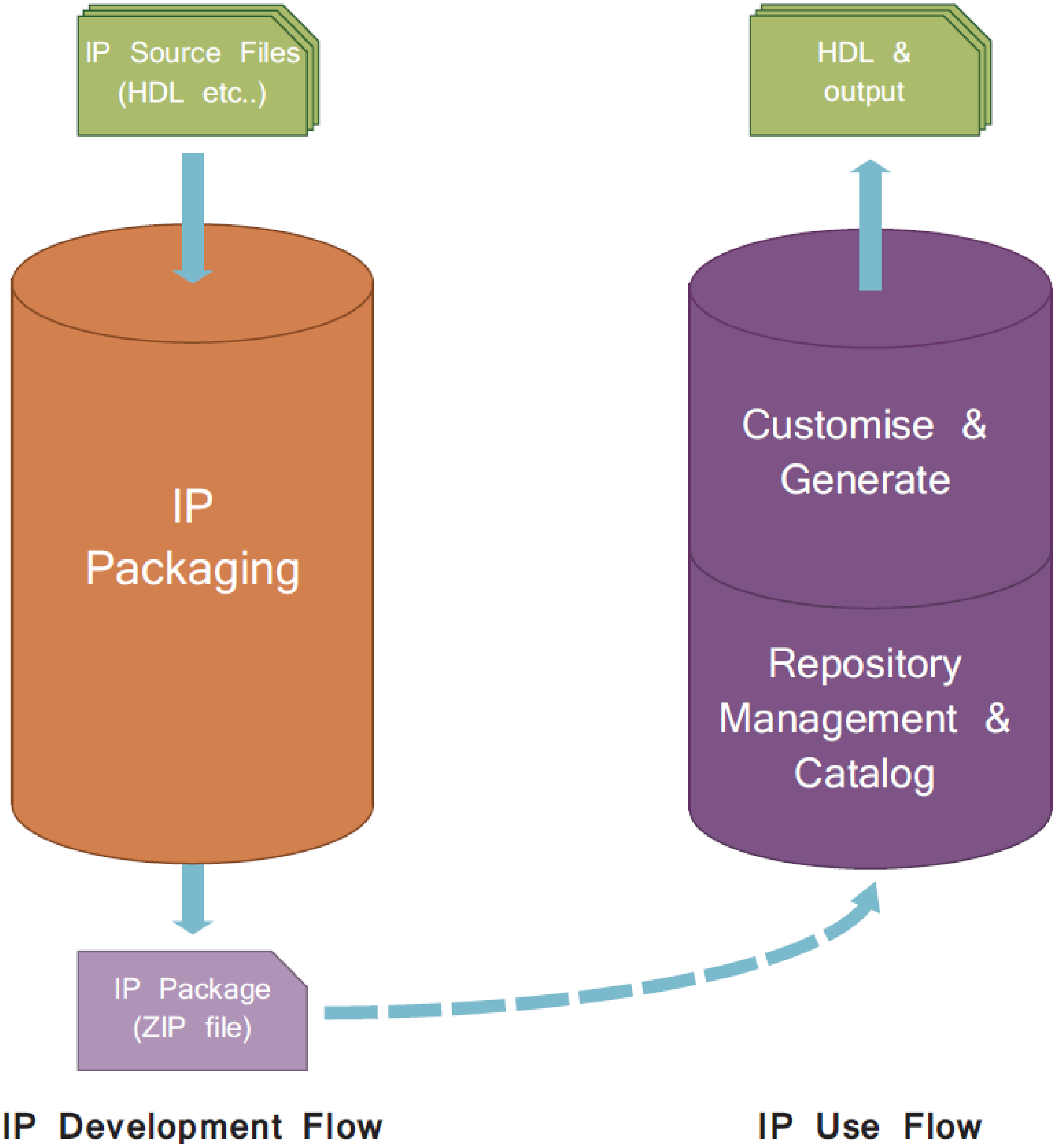
```
-- get the ssd and led values from ARM processor for display (←)
ssd_value <= slv_reg0(7 downto 0); -- pass to ssd_controller
led <= slv_reg1(7 downto 0); -- light up led
-- pass switch, btn, timer values to ARM processor for processing (→)
slv_reg2 <= (C_S00_AXI_DATA_WIDTH-1 downto 8 => ‘0’) & switch;
slv_reg3 <= (C_S00_AXI_DATA_WIDTH-1 downto 5 => '0') & btn;
process( clk, ms_count, timer )
begin
  if ( rising_edge(clk) ) then
    if (ms_count = C_MS_LIMIT-1) then
      ms_count <= (OTHERS => '0');
      timer <= timer + 1; -- ms
    else
      ...
```



① IP Block Creation: IP Packaging

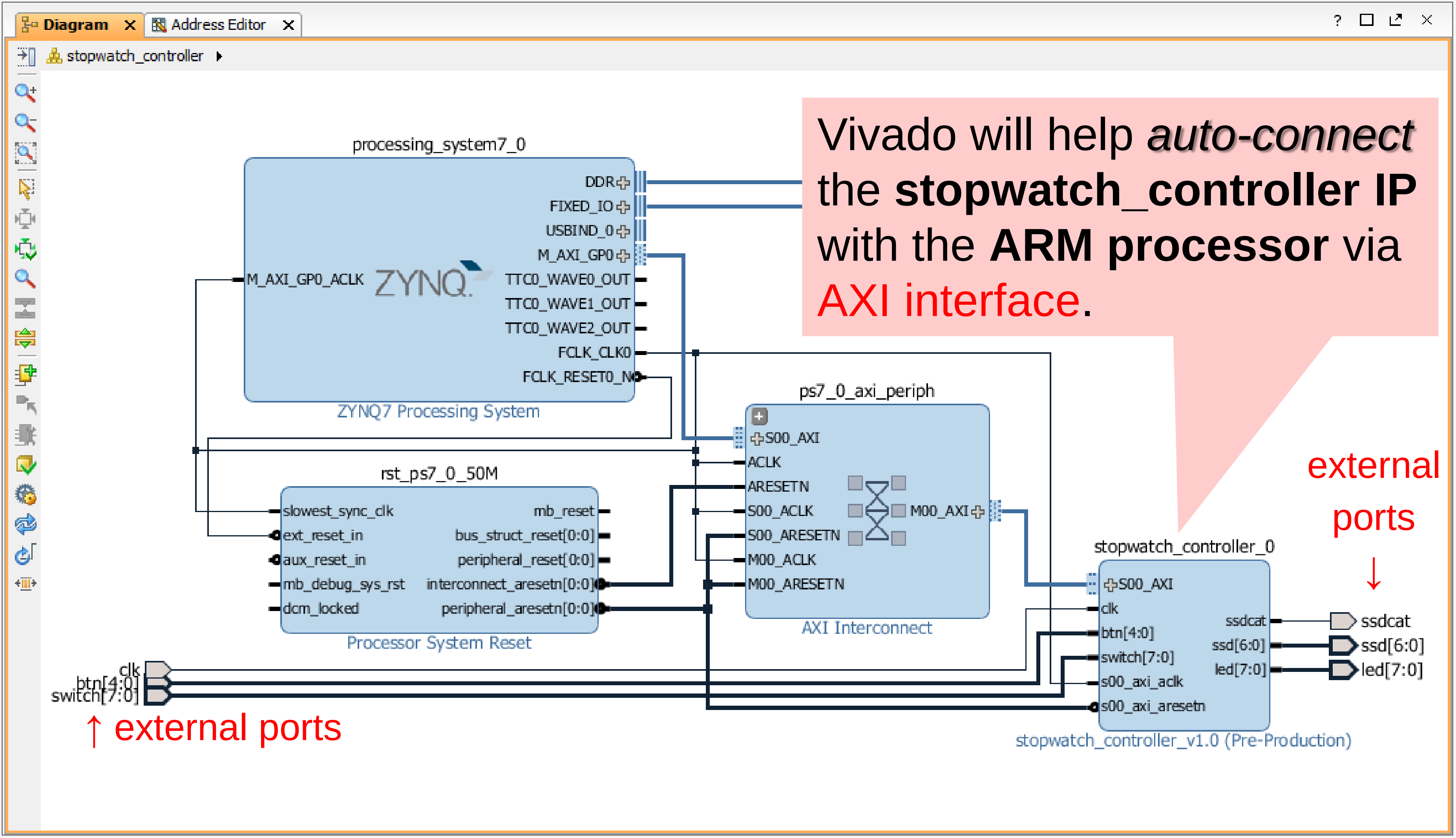
- Vivado IP Packager enables developers to quickly prepare IP for integration in the Vivado IP Catalog.

- Once the IP is selected in a Vivado project, the IP is treated like any other IP module from the IP Catalog.



② IP Integration

- Vivado IP Integrator provides a graphical “canvas” to configure IP blocks in an *automated* development flow.



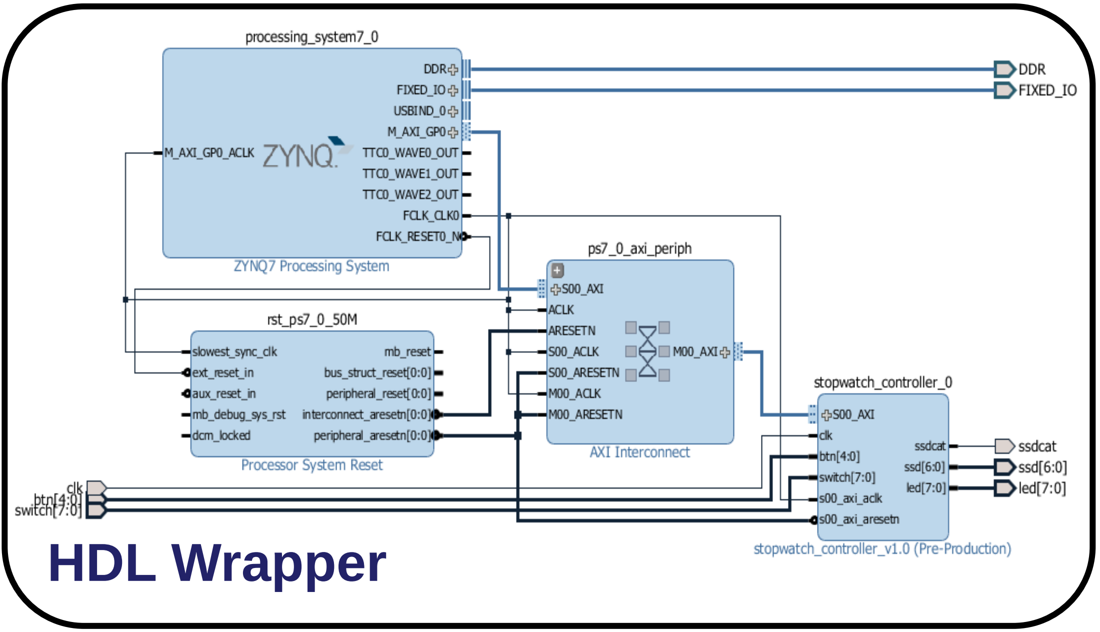
Vivado will help *auto-connect* the **stopwatch_controller IP** with the **ARM processor** via AXI interface.

external ports ↓

↑ external ports

③ HDL Wrapper & ④ Generate Bitstream

- Vivado will also help to create a top-level HDL Wrapper.
  - This will automatically generate the VHDL code for the whole block design.

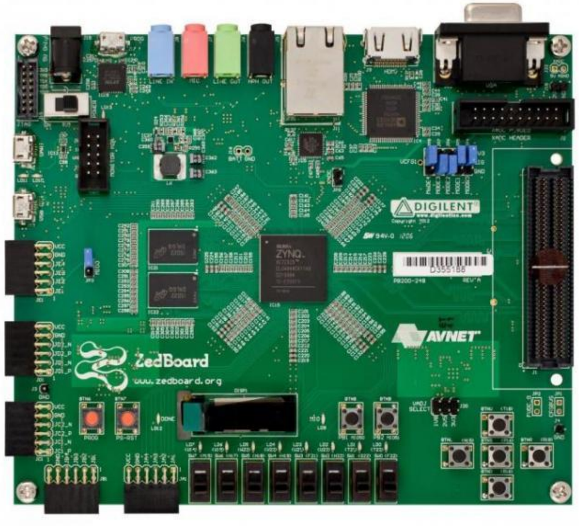


HDL Wrapper

- With a constraint file, the Bitstream can be generated and downloaded into the targeted board.

wrapper

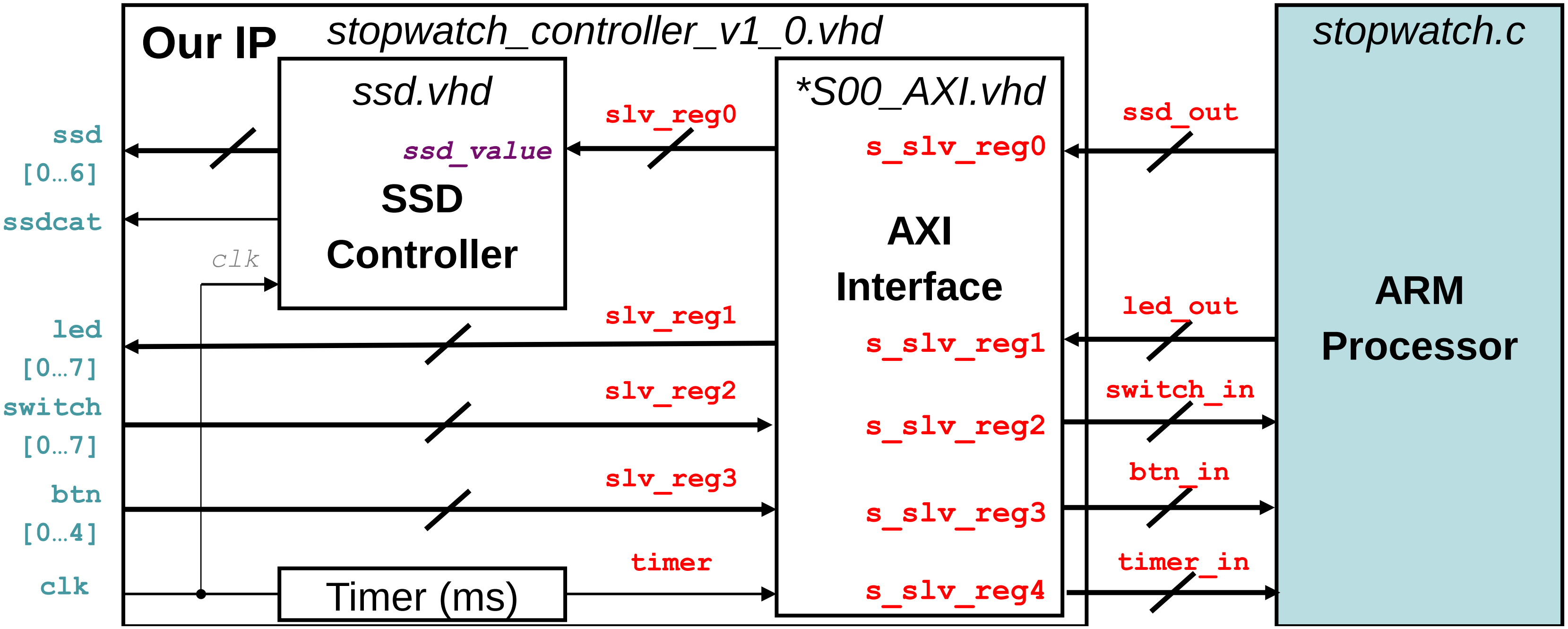
Program Bitstream



Key Steps of ARM-FPGA Integration

- **PART 1: IP Block Design** (using Xilinx Vivado)
  - ① **Create** and **Package** the **PL** logic blocks into **intellectual property (IP) block** with **AXI4 Interface**.
    - With AXI4, data can be exchanged via shared 32-bit registers.
  - ② **Integrate** the customized (or pre-developed) IP block with ZYNQ7 Processing System (**PS**) via **IP Block Design**.
    - Vivado can auto-connect IP block and ARM core via AXI interface.
  - ③ **Create HDL Wrapper** and **Add Constraints** to automatically generate the HDL code (VHDL or Verilog).
  - ④ **Generate and Program Bitstream** into the board.
- **PART 2: ARM Programming** (using Xilinx SDK)
  - ⑤ **Design** the bare-metal **application** in C/C++ language.
  - ⑥ **Launch on Hardware** (GDB): Run the code on ARM core.

PART 2: ARM Programming



- Five AXI slave registers are used for data exchange:
  - **s_slv_reg0**: value to be displayed on the Pmod SSD (←)
  - **s_slv_reg1**: value to be displayed on the LEDs (←)
  - **s_slv_reg2**: value inputted from the switches (→)
  - **s_slv_reg3**: value inputted from the buttons (→)
  - **s_slv_reg4**: the number of milliseconds elapsed (→)

⑤ ARM Programming

- We need two header files: one for controlling the ZYNQ processor in general, and the other to bring in items specific to our stopwatch controller.

```
#include “xparameters.h” // it is auto-generated
#include “stopwatch_controller.h” // it is auto-generated
```

- Then, we can make some simple names for the addresses of the registers in our IP block.

```
#define SW_BASE XPAR_STOPWATCH_CONTROLLER_0_S00_AXI_BASEADDR
#define SSD_ADDR STOPWATCH_CONTROLLER_S00_AXI_SLV_REG0_OFFSET
#define LED_ADDR STOPWATCH_CONTROLLER_S00_AXI_SLV_REG1_OFFSET
...
```

- Finally, we create a bare metal software program.
  - There is ***nothing but a sole program*** running on the ARM.
  - Thus, the program should ***never ever exit***. (How?)

Key: Interfacing via Registers (1/3)

```
/*** STATES ***/
u32 stopped, btn_in_prev, switch_in_pre, timer_zero;
// logic for initializing internal states
while(1) // infinite loop
{
       /*** INPUT ***/
       btn_in = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, BTN_ADDR);
       switch_in = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, SWITCH_ADDR);
       timer_in = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);

       /*** CONTROL ***/
       // logic for detecting btn and switch events
       u32 time_display;
       // logic for determining the time for led and ssd display

       /*** OUTPUT ***/
       STOPWATCH_CONTROLLER_mWriteReg(SW_BASE, LED_ADDR, time_display);
       STOPWATCH_CONTROLLER_mWriteReg(SW_BASE, SSD_ADDR, time_display);

       /*** FEEDBACK ***/
       btn_in_prev = btn_in; // btn_in_prev keeps previous btn
       switch_in_prev = switch_in; // switch_in_prev keeps previous sw
}
```

Key: Interfacing via Registers (2/3)

```
/* CONTROL: btn */
// determine whether BTN_C is pressed?
u32 btn_rise = ~btn_in_prev & btn_in;
if (btn_rise & BTN_C) stopped = ( stopped==1? 0 : 1);
```

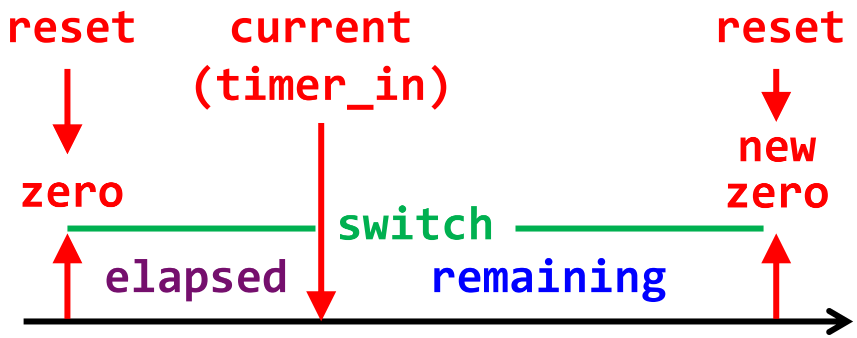
```
                                      CDRUL                     CDRUL
  #define BTN_C 16        btn_in_prev 00000         btn_in_prev 00000
  #define BTN_D 8                        ↓                         ↓
  #define BTN_R 4        ~btn_in_prev 11111        ~btn_in_prev 11111
  #define BTN_U 2    &)   btn_in      10000    &)   btn_in      00000
  #define BTN_L 1    ----------------------    ----------------------
                          btn_rise    10000         btn_rise    01000
                                    rising                  not rising
```

```
/* CONTROL: switch */
// determine whether any of switches has been changed?
if (switch_in != switch_in_prev) stopped = 1;
```

```
                           switch_in_prev 0000 0000
                  compare)      switch_in 0010 0000
                  ---------------------------------
                                               TRUE (otherwise: FALSE)
```

Key: Interfacing via Registers (3/3)

```
/* CONTROL: time */
int time_display; // the “remaining” time for display
if( stopped )
{
    // reset time_display by switches and timer_zero by current time
    time_display = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, SWITCH_ADDR);
    timer_zero = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);
}else
{
    // calculate the “elapsed” and “remaining” time (in seconds)
    u32 time_elapsed = (timer_in - timer_zero) / 1000; // “elapsed”
    time_display = switch_in - time_elapsed; // “remaining”
    if(time_display < 0)
    {
        // reset timer_zero by the “current” time to restart count-down
        timer_zero = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);
    }
}
```



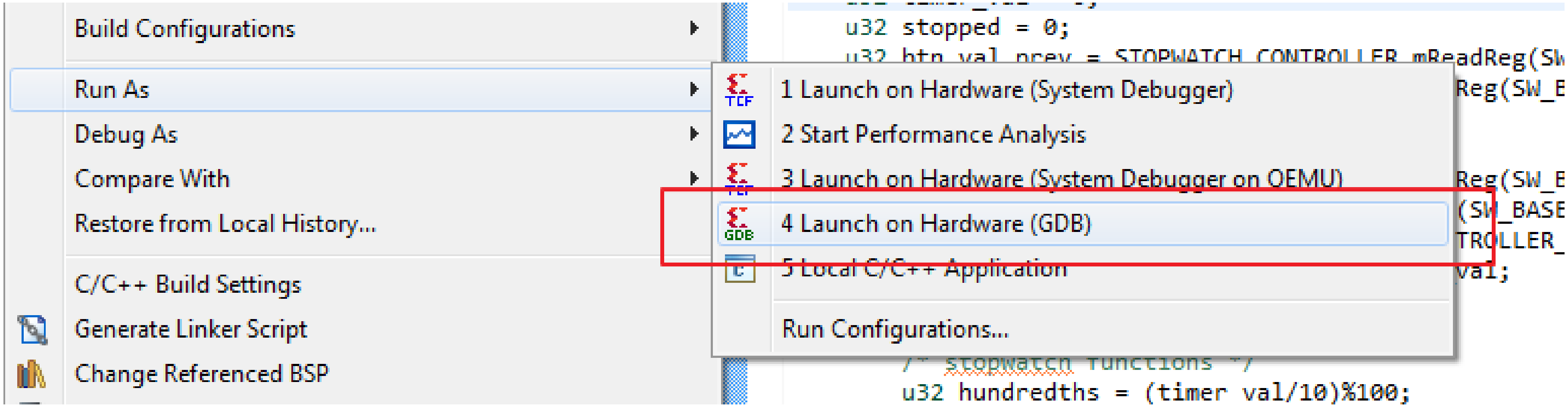
Class Exercise 7.1

- The stopwatch originally counts down at the rate of one number per second (1 Hz). Modify the high-lighted line to let it count-down at the rate of 0.5 Hz.

```
/* CONTROL: time */
int time_display; // the “remaining” time for display
if( stopped )
{
    // reset time_display by switches and timer_zero by current time
    time_display = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, SWITCH_ADDR);
    timer_zero = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);
}else
{
    // calculate the “elapsed” and “remaining” time (in seconds)
    u32 time_elapsed = (timer_in - timer_zero) / 1000; // “elapsed”
    time_display = switch_in - time_elapsed; // “remaining”
    if(time_display < 0)
    {
        // reset timer_zero by the “current” time to restart count-down
        timer_zero = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);
    }
}
```

⑥ Launch on Hardware (GDB)

- Finally, after the software stopwatch (`.c`) is ready, you can run it on ARM by Launch on Hardware (GDB).
  - **GDB**: GNU Debugger is the most popular debugger for UNIX systems to debug C and C++ programs.
  - Vivado will help automatically compile, link, and load your C program.

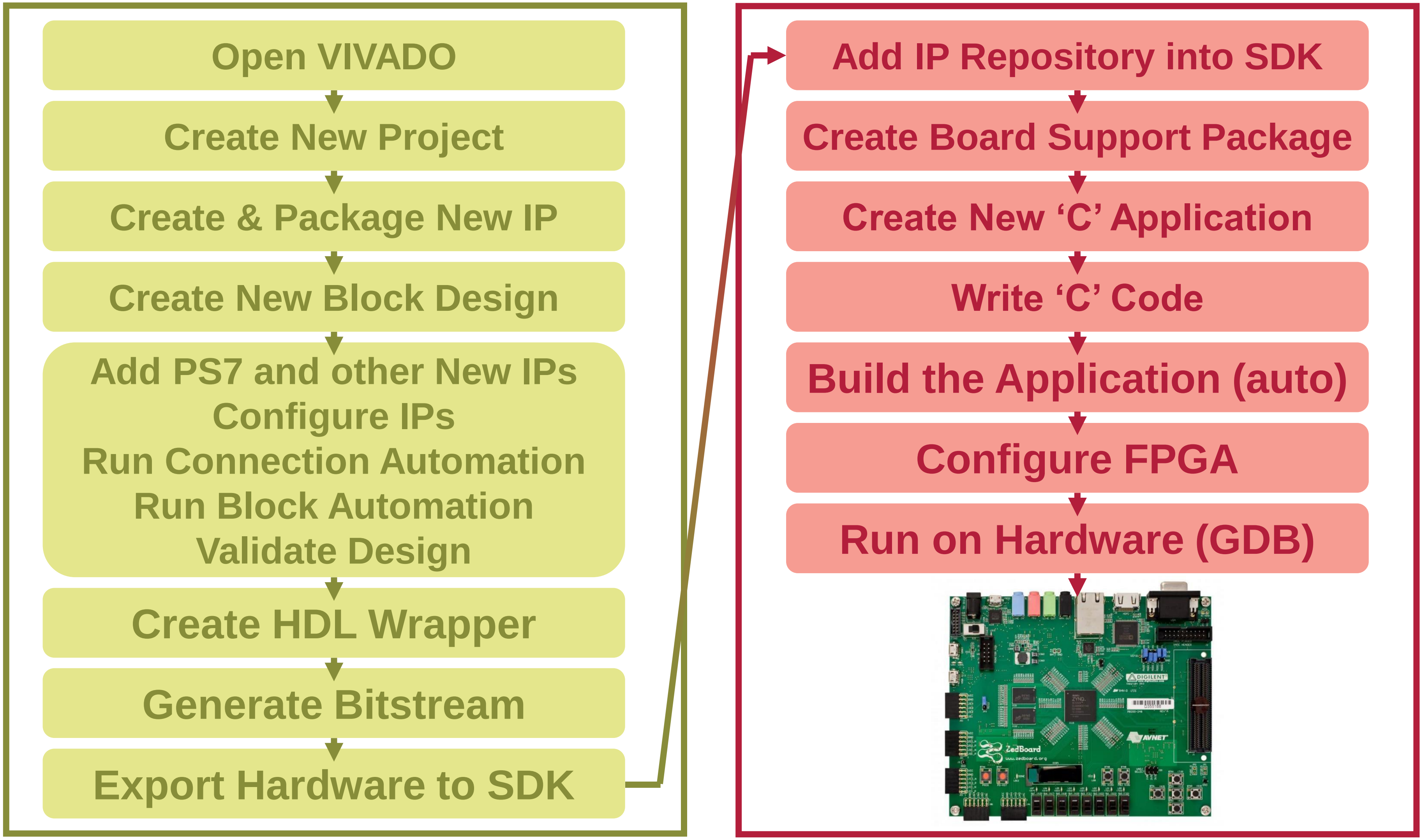


```
u32 stopped = 0;
u32 hundredths = (timer val/10)%100;
```

Summary of ARM-FPGA Integration

VIVADO

SDK
Software Development Kit



- Open VIVADO
- Create New Project
- Create & Package New IP
- Create New Block Design
- Add PS7 and other New IPs
  Configure IPs
  Run Connection Automation
  Run Block Automation
  Validate Design
- Create HDL Wrapper
- Generate Bitstream
- Export Hardware to SDK

- Add IP Repository into SDK
- Create Board Support Package
- Create New ‘C’ Application
- Write ‘C’ Code
- Build the Application (auto)
- Configure FPGA
- Run on Hardware (GDB)

Summary

- Rapid Prototyping with Zynq

- Rapid Prototyping (I): Integration of ARM and FPGA
  - PART 1: IP Block Design (Xilinx Vavido)
    - ① IP Block Creation
    - ② IP Integration
    - ③ HDL Wrapper
    - ④ Generate Bitstream
  - PART 2: ARM Programming (Xilinx SDK)
    - ⑤ ARM Programming
    - ⑥ Launch on Hardware

- Case Study: Software Stopwatch