



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

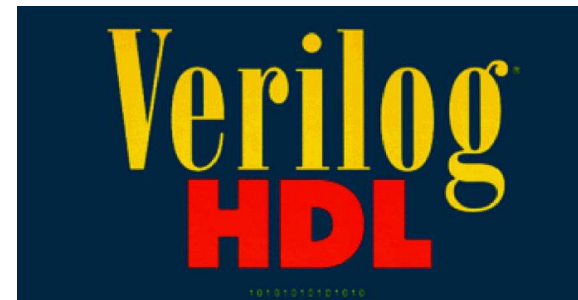
mcyang@cse.cuhk.edu.hk



High-level vs Hardware Description Lang.



VHDL
Very High Speed Integrated Circuit
Hardware Description Language



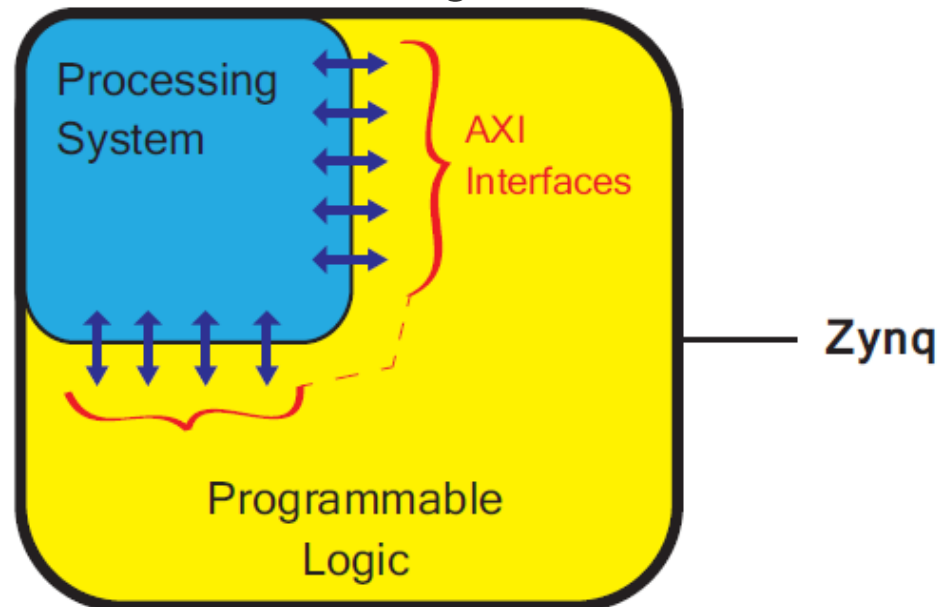


- Rapid Prototyping with Zynq
- Rapid Prototyping (I): Integration of ARM and FPGA
 - Case Study: Lab 08 Software Stopwatch
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware

Zynq Features



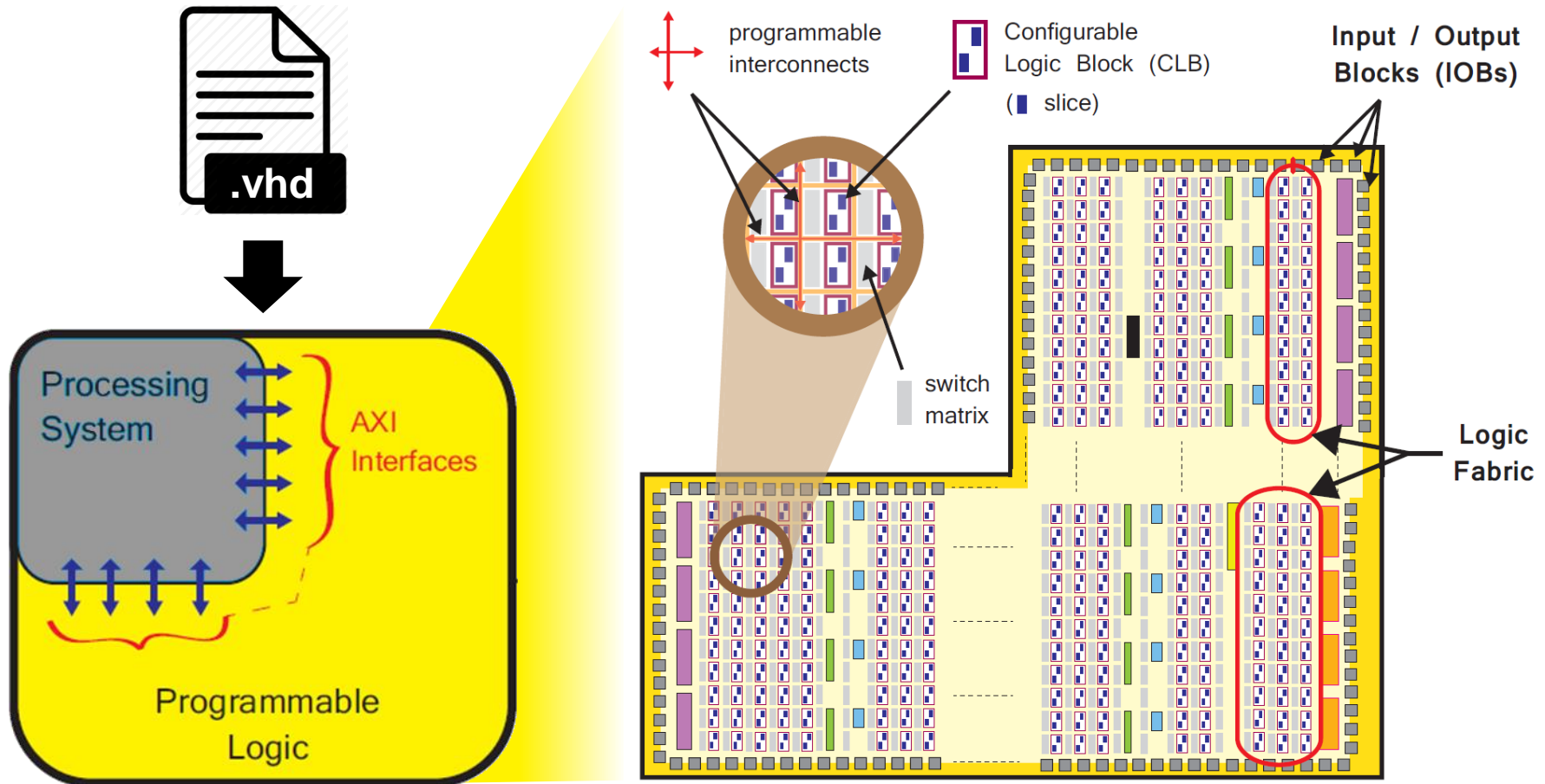
- The defining features of Zynq:
 - **Processing System (PS)**: Dual-core ARM Cortex-A9 CPU
 - **Programmable Logic (PL)**: Equivalent traditional FPGA
 - **Advanced eXtensible Interface (AXI)**: High bandwidth, low latency connections between PS and PL.
 - *PS and PL can each be used for what they do best, without the overhead of interfacing 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 usually **hard** to implement **complicated logic or software**.



Rapid Prototyping with Zynq: PS + PL



PS for Software:

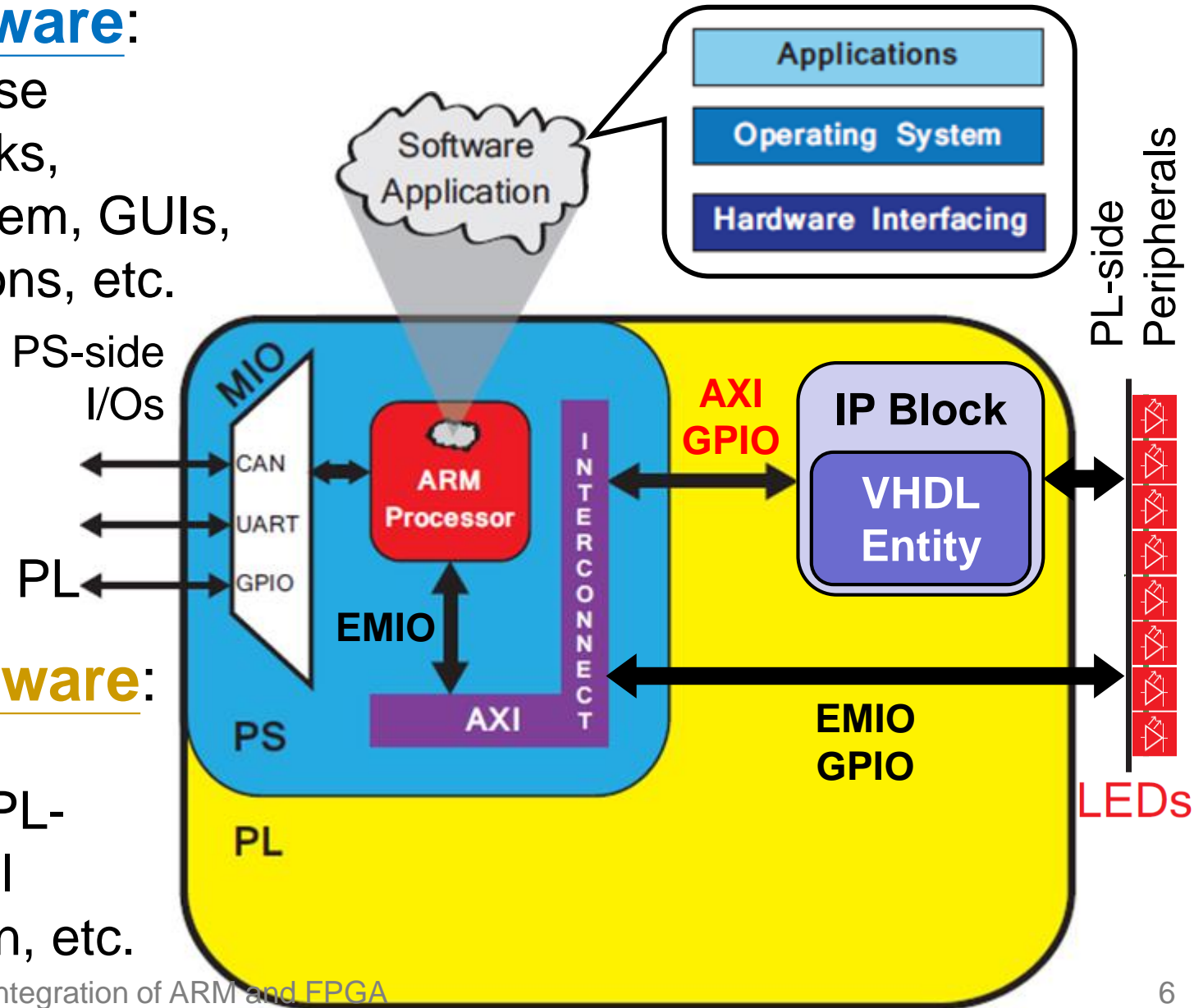
general purpose sequential tasks, operating system, GUIs, user applications, etc.

AXI:

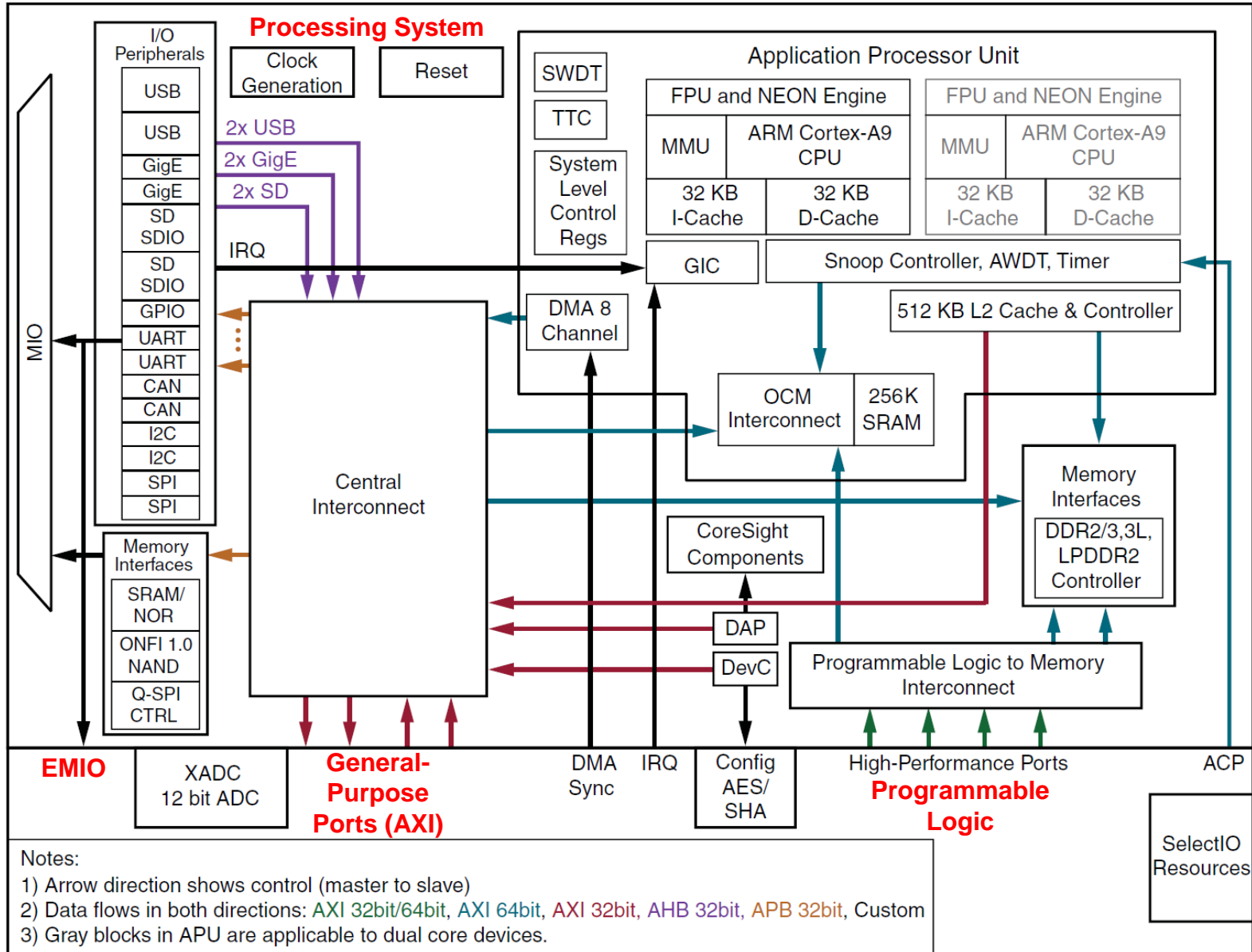
hardware interfacing between PS & PL

PL for Hardware:

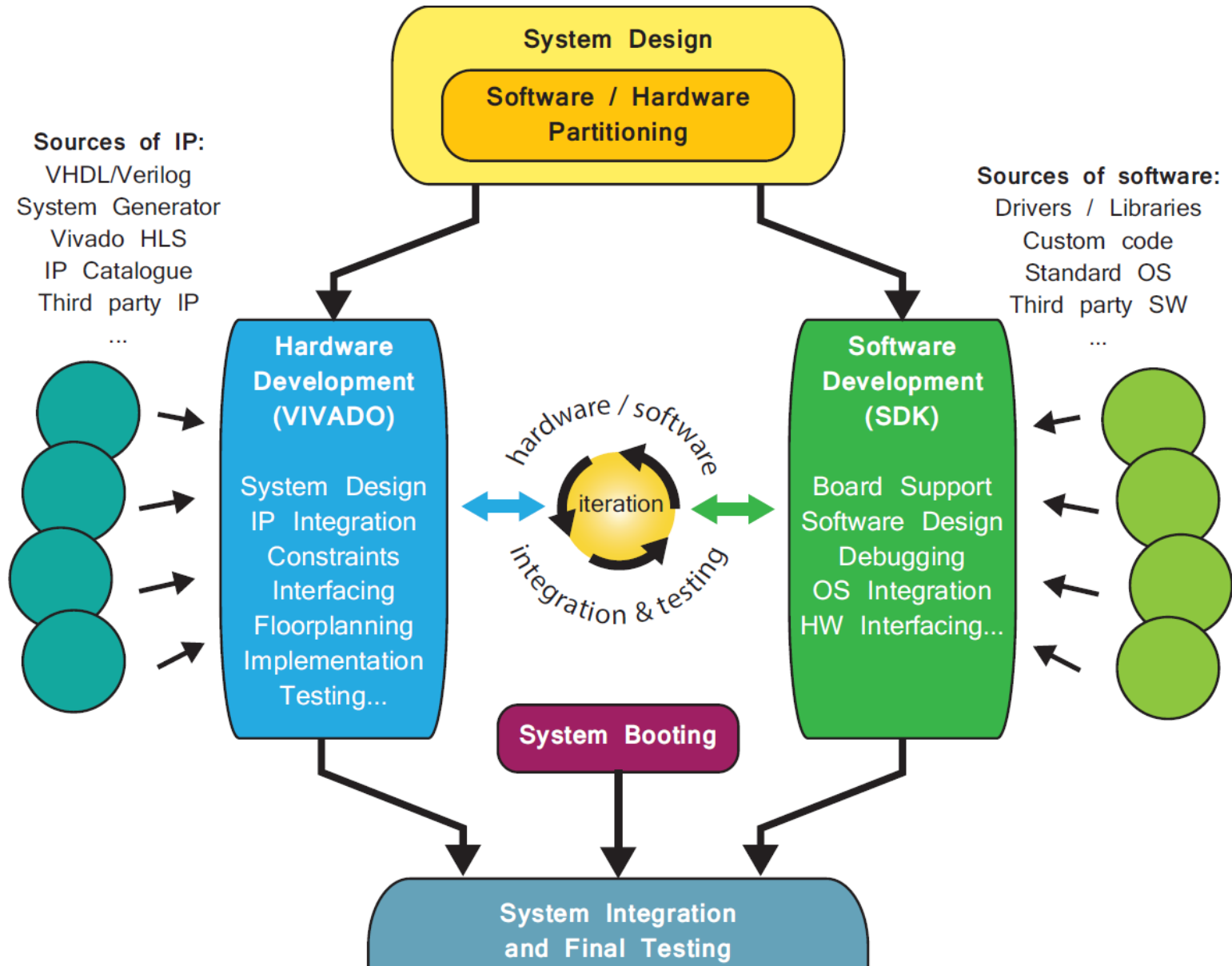
intensive data computation, PL-side peripheral communication, etc.



Zynq-7000 SoC



Design Flow with Zynq



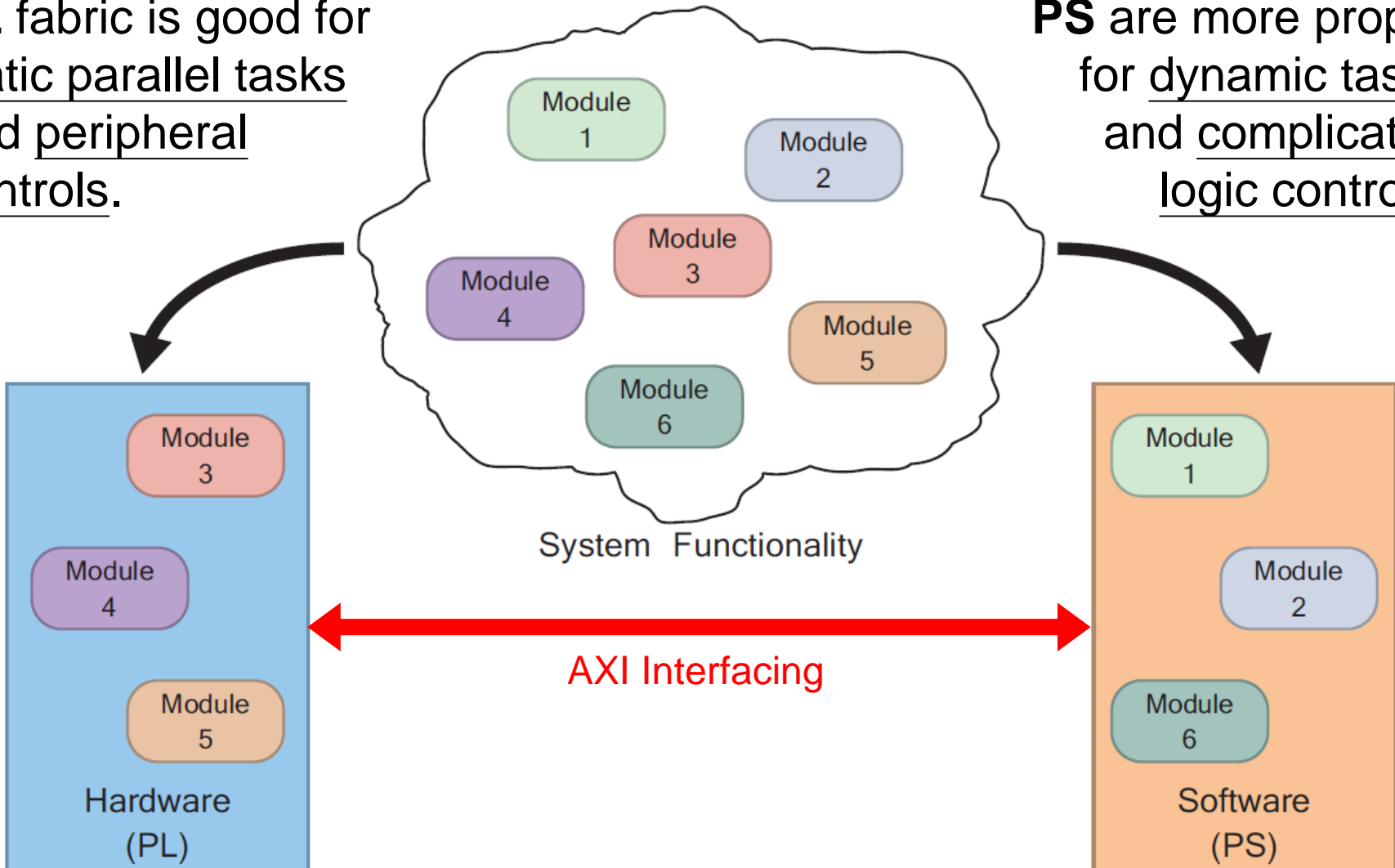
Key: Hardware/Software Partitioning



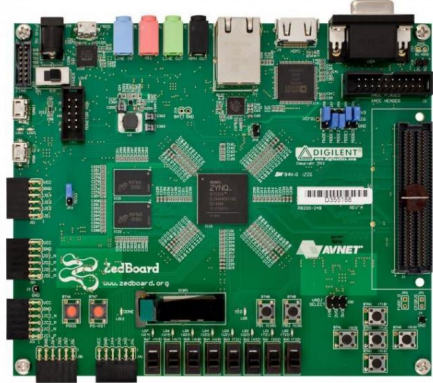
- **PS** and **PL** can each be used for what they do best.

PL fabric is good for static parallel tasks and peripheral controls.

PS are more proper for dynamic tasks and complicated logic controls.



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
hardware

Xilinx Vivado
(HDL)

Programmable Logic Design

Hardware Base System

Hardware Base System

Program Logic (PL)

Style 1)
FPGA (PL)

VHDL or Verilog Programming
(Lec. 01~06 & 10)

Style 2)
ARM + FPGA

ARM Programming & IP Block Design
(Lec. 07 & 09)

Style 3)
Embedded OS

Shell Script & sysfs EMIO GPIO
(Lec. 08)



- Rapid Prototyping with Zynq
- **Rapid Prototyping (I): Integration of ARM and FPGA**
 - Case Study: Lab 08 Software Stopwatch
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware

Integration of ARM and FPGA

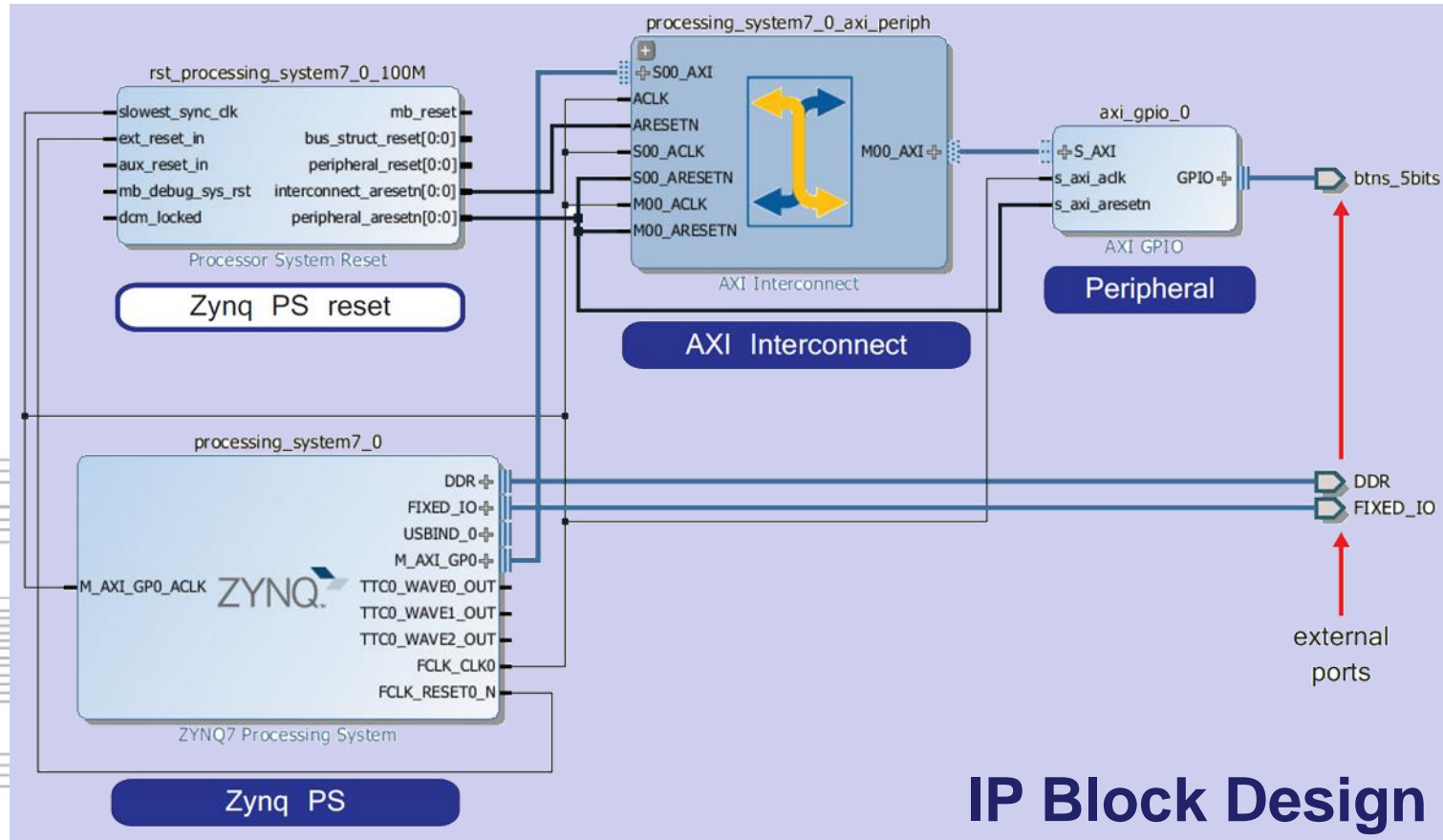
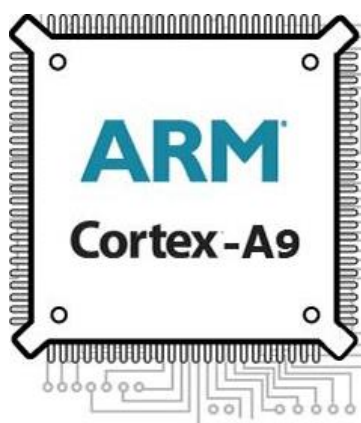


- To integrate ARM and FPGA, we need to do:
 - IP Block Design** on Xilinx Vivado using HDL
 - ARM Programming** on Xilinx SDK using C/C++



ARM

Programming



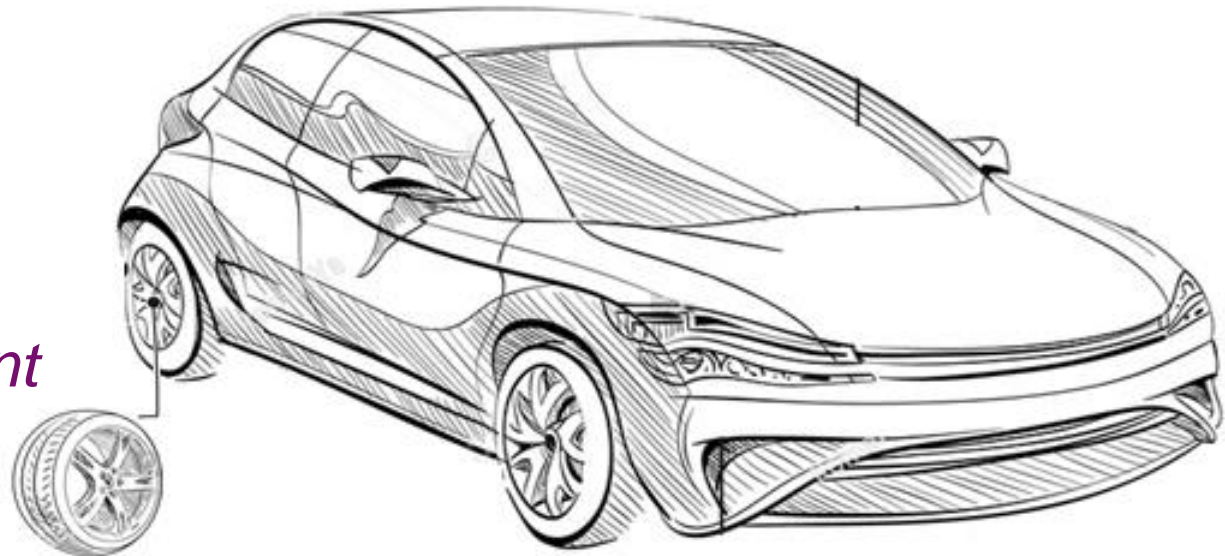
IP Block Design

Intellectual Property (IP) Block



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

Why reinvent the wheel?



Hard vs. Soft IP Block



- **Hard IP Block:** Permit **no** realistic method of modification by end users.
 - **Firm IP Block:** An IP block already *undergone full synthesis, place and route design flow for a targeted FPGA/ASIC.*
- **Soft IP Block:** **Allow** end users to customize the IP by controlling the synthesis, place and route design flow.
 - The highest level of soft IP block customization is available when the **source HDL code** is provided.
 - Such as the **C source code** is provided.
 - Soft IP block can be also provided as a **gate-level netlist**.
 - Such as the compiled **assembly code listing** is provided.

Sources of IP Block



- **IP Libraries:** Xilinx provides an extensive catalogue of **soft 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)**.

Steps of ARM-FPGA Integration

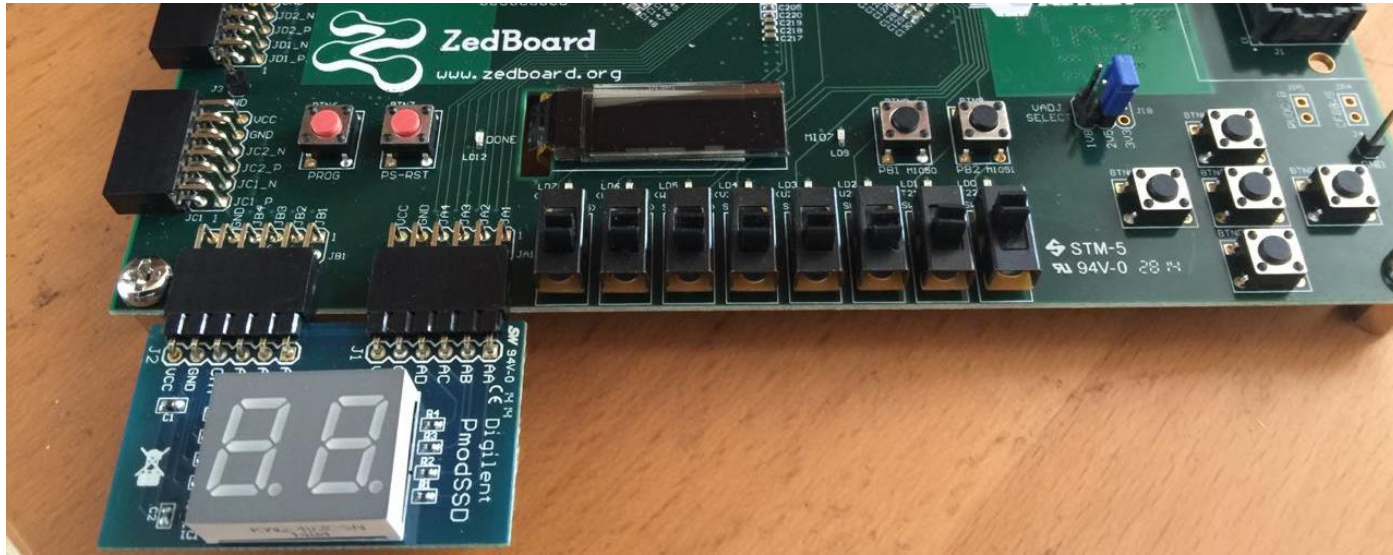


- **PART 1: IP Block Design** (Software: **Xilinx Vivado**)
 - ① **Create and Package** the **PL** logic blocks into **intellectual property (IP) block** with **AXI4 Interface**.
 - With AXI4, data can be exchanged via shared 32-bit registers.
 - ② **Integrate** the customized (or pre-developed) IP block 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** (Software: **Xilinx SDK**)
 - ⑤ **Design** the bare-metal **application** in **C/C++ language**.
 - ⑥ **Launch on Hardware (GDB)**: Run the code on ARM core.



- Rapid Prototyping with Zynq
- **Rapid Prototyping (I): Integration of ARM and FPGA**
 - **Case Study: Lab 08 Software Stopwatch**
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware

Recall: What we have done in Lab06



entity sevensseg is

```
port (
    clk : in STD_LOGIC;
    switch : in STD_LOGIC_VECTOR (7 downto 0);
    btn : in STD_LOGIC;
    ssd : out STD_LOGIC_VECTOR (6 downto 0);
    ssdcats : out STD_LOGIC );
```

end sevensseg;

- *Task1: Display the input number (XY) in hexadecimal*
- *Task2: Count down from the input number (XY) to (00)*

Recall: Lab06 (1/2)



```
-- generate 1ms and 1s clocks
process (clk)
begin
  if rising_edge(clk) then
    if (s_count = 49999999) then
      s_pulse <= not s_pulse;
      s_count <= 0;
    else
      s_count <= s_count + 1;
    end if;
    if (ms_count = 49999) then
      ms_pulse <= not ms_pulse;
      ms_count <= 0;
    else
      ms_count <= ms_count + 1;
    end if;
  end if;
end process;

-- read button (combinational)
process (btn)
begin
  if rising_edge(btn) then
    counter_en <= not counter_en;
  end if;
end process;
```

```
-- count down
process (s_pulse)
begin
  if rising_edge(s_pulse) then
    if (counter_en = '1') then
      if (counter=0) then
        counter <= to_integer(
          unsigned(switch));
      else
        counter <= counter - 1;
      end if;
    end if;

    counter_vec <= std_logic_vector(
      to_unsigned(counter, 8));
  end if;
end process;
```

```
-- update the seven segment display
process (ms_pulse)
  ... -- see the next page
end process;
```

```
-- output ssd (combinational)
process (digit)
  ... -- see the next page
end process;
```

Recall: Lab06 (2/2)



```
-- generate 1ms and 1s clocks
process (clk)
begin
  if rising_edge(clk) then
    if (s_count = 49999999) then
      s_pulse <= not s_pulse;
      s_count <= 0;
    else
      s_count <= s_count + 1;
    end if;
    if (ms_count = 49999) then
      ms_pulse <= not ms_pulse;
      ms_count <= 0;
    else
      ms_count <= ms_count + 1;
    end if;
  end if;
end process;

-- read button (combinational)
process (btn)
begin
  if rising_edge(btn) then
    counter_en <= not counter_en;
  end if;
end process;
```

```
-- count down
process (s_pulse)
  .. -- see the previous page
end process;

-- update the seven segment display
process (ms_pulse)
begin
  if ms_pulse='1' then
    if (counter_en = '1') then
      digit <= counter_vec(7 downto 4);
    else
      digit <= switch(7 downto 4);
    end if;
  else
    if (counter_en = '1') then
      digit <= counter_vec(3 downto 0);
    else
      digit <= switch(3 downto 0);
    end if;
  end if;
  ssdcat <= ms_pulse; -- select display
end process;

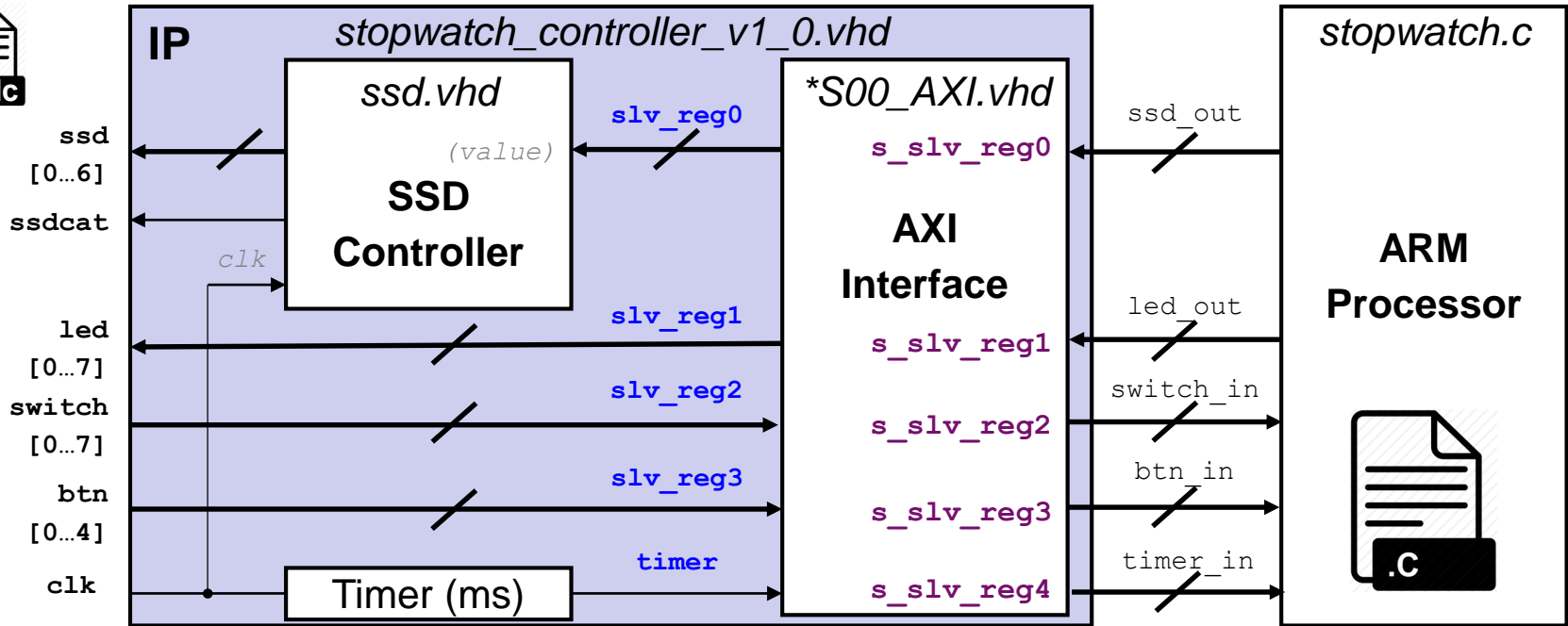
-- output ssd (combinational)
process (digit)
begin
  case digit is
    when "0000" => ssd <= "1111110";
    ...
  end case;
end process;
```

Hardware vs. Software Stopwatch



- In Lab 06, what we've done is a **hardware stopwatch** in which the FPGA (PL) is responsible for **both**:
 - **Hardware**: Interfacing with the user via `switch` and `btn`.
 - **Software**: Generating the time to be shown on `ssd` and dealing with different user inputs.
- In Lab 08, we will design a **software stopwatch** through ARM-FPGA integration as follows:
 - **Hardware**: FPGA (PL) is **only** responsible for hardware interfacing with the user via `switch`, `btn`, and `led`.
 - **Software**: ARM (PS) is responsible for generating the values to be shown on `ssd` and `led`, and dealing with different user inputs or events.
 - By ARM programming, an even more complicated control logic can be realized in an **easier way**.

Lab08: Design Specification (1/2)



| Register | AXI Port / Signal Name | Related Port on FPGA | Function |
|----------|------------------------|----------------------|--|
| 0 | s_slv_reg0 / slv_reg0 | ssd | SSD output |
| 1 | s_slv_reg1 / slv_reg1 | led | LED output |
| 2 | s_slv_reg2 / slv_reg2 | switch | Switch input |
| 3 | s_slv_reg3 / slv_reg3 | btn | Button input |
| 4 | s_slv_reg4 / slv_reg4 | timer | Timer signal input (generated by FPGA) |
| N/A | N/A | clk | 100MHz clock signal from PL |
| N/A | N/A | ssdcats | 7-segment digit selection |
| N/A | N/A / timer | N/A | 1kHz clock divided from clk |

Lab08: Design Specification (2/2)



- We need five AXI slave registers (`s_slv_reg0~4`) for *exchanging data between ARM and FPGA*:
 - The ARM processor reads the input value from the switches and the buttons, as well as a 1 KHz timer signal.
 - `s_slv_reg2`: Switch input
 - `s_slv_reg3`: Button input
 - `s_slv_reg4`: 1 KHz clock divided from 1 MHz `clk` of PL.
 - The C program runs on the ARM processor, calculates the stopwatch's time based on the data input, generates values to be displayed on the 7-segment displays and the LEDs, and sends the data back to the FPGA for display.
 - `s_slv_reg0`: SSD output
 - `s_slv_reg1`: Led output



- Rapid Prototyping with Zynq
- Rapid Prototyping (I): Integration of ARM and FPGA
 - Case Study: Lab 08 Software Stopwatch
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware

① IP Block Creation

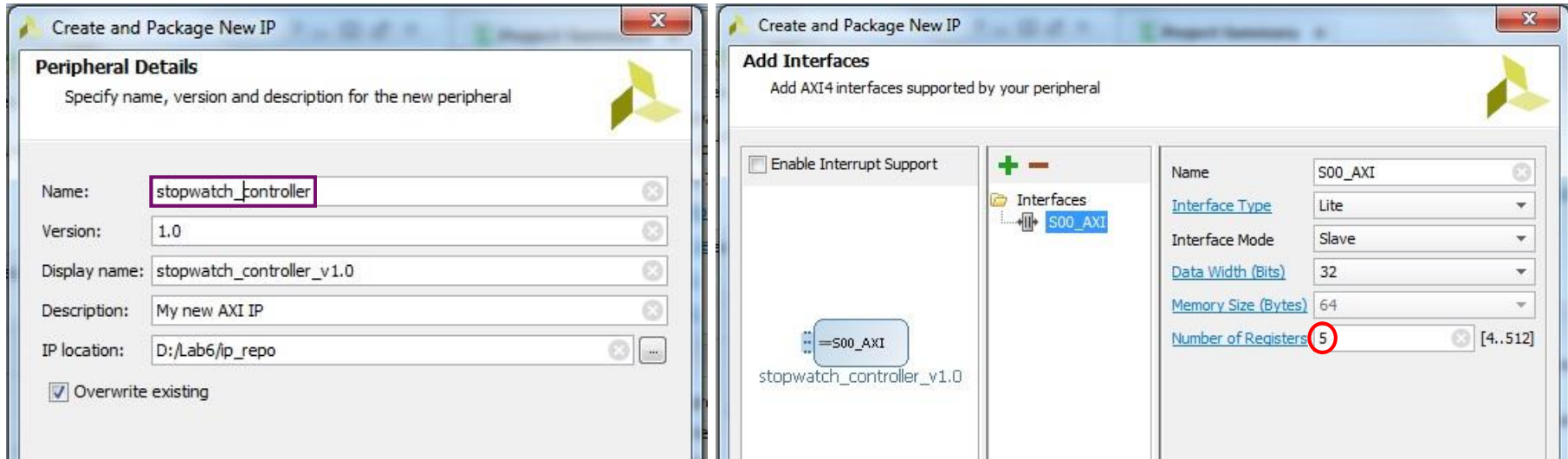


- **IP Block Creation in HDL**
 - Hardware description languages (HDLs), such as VHDL and Verilog, are specialized programming languages.
 - HDLs describe the operation and structure of digital circuits.
 - The ability to create IP cores in HDL allows you the **maximum control** over the functionality of your peripheral.
- **IP Block Creation in Vivado High-Level Synthesis**
 - Vivado HLS is a tool provided by Xilinx.
 - **HLS** is capable of converting C-based designs into RTL design files for implementation of Xilinx All Programmable devices (see Lecture 09).
 - C-based Designs: C, C++, or SystemC
 - RTL Designs: VHDL, Verilog, or SystemC

① IP Block Creation



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

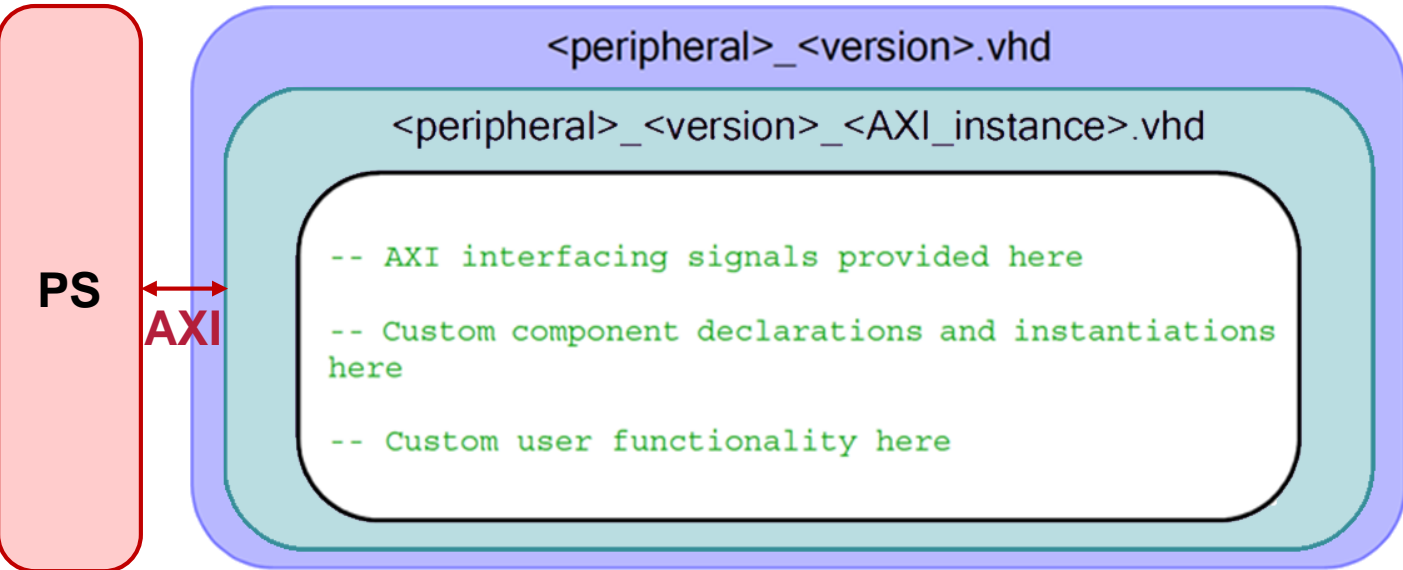


- Two .vhd files will be generated automatically:
 - **stopwatch_controller_v1_0.vhd**: This file instantiates the AXI-Lite interface and contain the **stopwatch functionality**.
 - **stopwatch_controller_v1_0_S00_AXI.vhd**: This file contains only the AXI-Lite **bus functionality**.

① AXI Interfacing



- IP blocks designed in HDL are communicated the processing system (PS) via an **AXI interface**.
 - Vivado will *auto-create* the following source files for editing:
 - **<peripheral>_<version>.vhd**: the top-level file **defines the design interface**, lists connections and ports for the AXI interface, as well as **implements the functionality of user-defined entities**.
 - **<peripheral>_<version>_<AXI_instance>.vhd**: describes an instance of **AXI interface** for this IP block for integrating into PS.



Where:
<peripheral> is the name of the IP to be created.
<version> is the current version, i.e. v1_0.
<AXI_instance> is the AXI4 master (M) or slave (S) interface instance, i.e. M00_AXI or S00_AXI. (There may be multiple instances per peripheral).

stopwatch_controller_v1_0_S00_AXI.vhd (1/2)

- Vivado will **auto-declare slave registers** (as internal signals) based on the number entered by users:

```
-----  
---- Signals for user logic register space example  
-----  
---- Number of Slave Registers 5  
signal slv_reg0: std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
signal slv_reg1: std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
signal slv_reg2: std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
signal slv_reg3: std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
signal slv_reg4: std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
```

- But we still need to **define ports** for these registers:

```
-- Users to add ports here  
s_slv_reg0: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
s_slv_reg1: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
s_slv_reg2: in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
s_slv_reg3: in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
s_slv_reg4: in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);  
-- User ports ends
```

stopwatch_controller_v1_0_S00_AXI.vhd (2/2)

- Then we **interconnect** the internal slave **registers** and the user-defined **ports**:

```
-- Add user logic here
s_slv_reg0 <= slv_reg0; ← SSD output
s_slv_reg1 <= slv_reg1; ← LED output
slv_reg2 <= s_slv_reg2; ← Switch input
slv_reg3 <= s_slv_reg3; ← Button input
slv_reg4 <= s_slv_reg4; ← Timer input
-- User logic ends
```

- Besides, we also need to **disable** some auto-generated “**write logic**” for **slv_reg2 ~ slv_reg4** (i.e., switch, button, and timer), since:
 - Their values would be read-only from the FPGA, and
 - The user application (i.e., stopwatch.c) cannot change their values.



- Next, we complete the **stopwatch functionality**:
 - 1) We first **define ports** in **entity** of stopwatch_controller_v1_0:

```
-- Users to add ports here
clk : in std_logic;
btn : in std_logic_vector(4 downto 0);
switch : in std_logic_vector(7 downto 0);
ssdcnt : out std_logic;
ssd : out std_logic_vector(6 downto 0);
led : out std_logic_vector(7 downto 0);
-- User ports ends
```
 - 2) The following changes should be also made:
 - Add **generic parameters** (if any),
 - Add ports in **component** of stopwatch_controller_v1_0_S00_AXI,
 - Since we define new ports for the five registers in *AXI.vhd
 - Add other **user-defined components** (if any), and
 - Add required **internal signals** for user logic and functionality.
- (Note: Please refer to the lab sheet for more detailed instructions.)*



- Next, we complete the stopwatch functionality:
 - 3) Then we create and connect `stopwatch_AXI` and `ssd_controller` components in the architecture body of `stopwatch_controller_v1_0` as follows:

```
stopwatch_controller_v1_0_S00_AXI
```

```
...
port map (
  -- Users to add port map
  s_slv_reg0 => slv_reg0,
  s_slv_reg1 => slv_reg1,
  s_slv_reg2 => slv_reg2,
  s_slv_reg3 => slv_reg3,
  s_slv_reg4 => timer,
  -- User port map ends
  ...
)
```

```
-- Add user logic here
```

```
ssd_controller
```

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

*Note: VHDL allows the designer to parametrize the entity during the component instantiation via **generic map**. It is used here to indicate the value for counting 1 ms in ZedBoard.*

stopwatch_controller_v1_0.vhd (3/3)



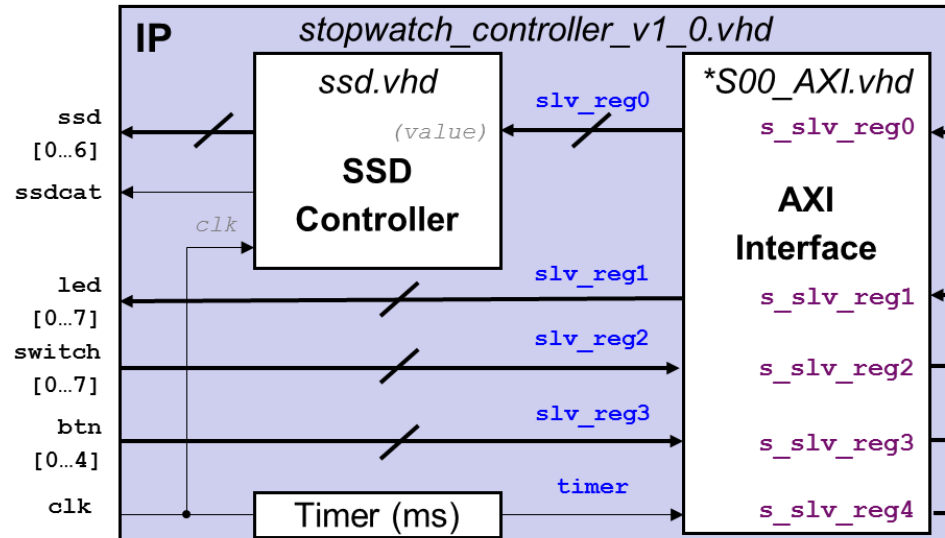
- Next, we complete the stopwatch functionality:
 - Last, we implement the stopwatch logic in the architecture body of stopwatch_controller_v1_0 as follows:

`ssd_value` <= `slv_reg0` (7 downto 0); ← `ssd_controller` will take over the `ssd` display!
`led` <= `slv_reg1` (7 downto 0); ← LED output

`slv_reg2` <= (C_S00_AXI_DATA_WIDTH-1 downto 8 => '0') & `switch`; ← Switch input

`slv_reg3` <= (C_S00_AXI_DATA_WIDTH-1 downto 5 => '0') & `btn`; ← Button input

```
process(clk, ms_count, timer) begin
  if (clk'event and clk='1') then
    if (ms_count = C_MS_LIMIT-1) then
      ms_count <= (OTHERS => '0');
      timer <= timer + 1; ← Timer input
    else
      ms_count <= ms_count + 1;
    end if;
  end if;
end process;
-- User logic ends
```



ssd.vhd



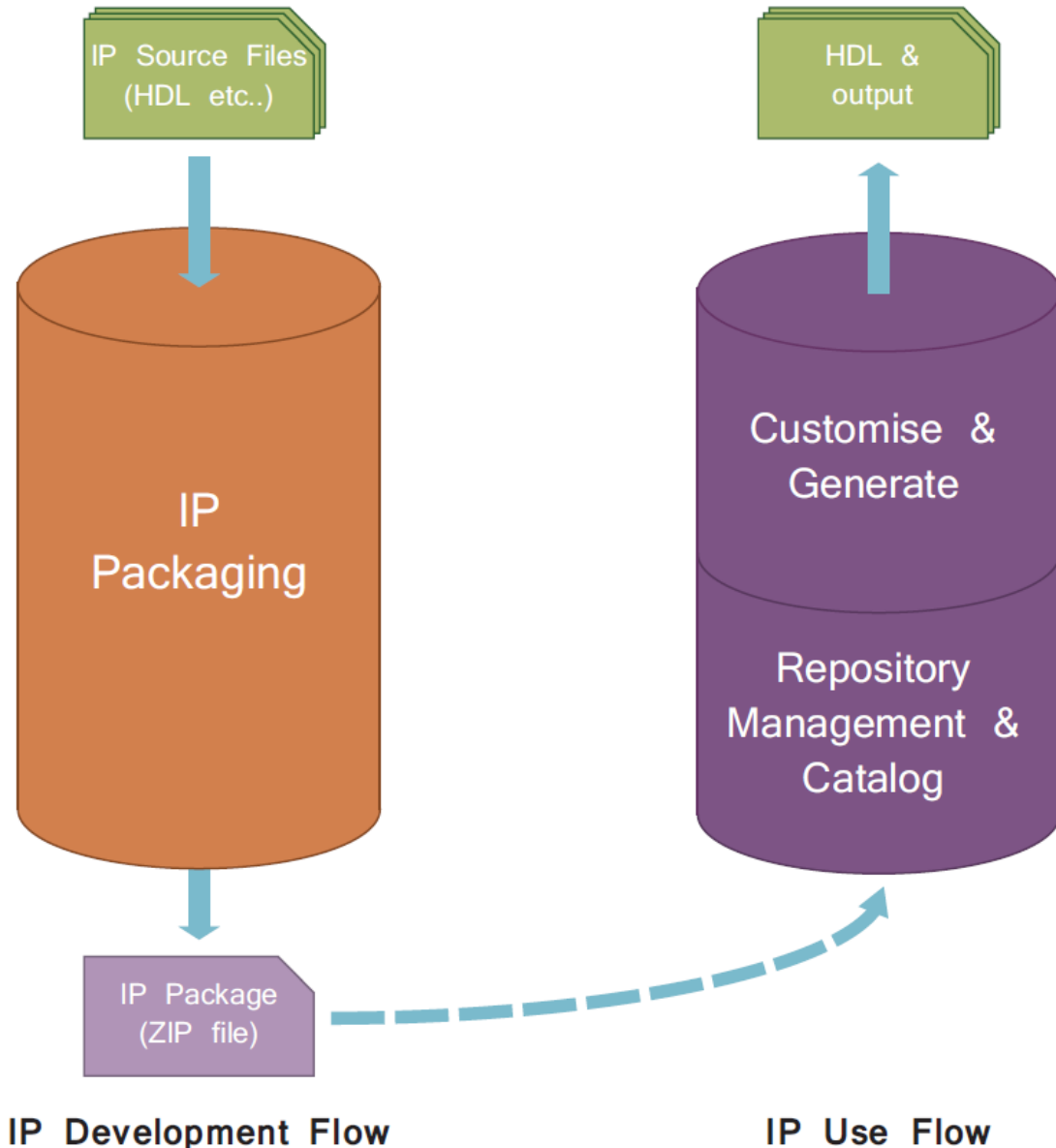
```
-- count 1 ms (generic: cat_period)
process(clk, count)
begin
  if (clk'event and clk='1') then
    if (count = cat_period-1) then
      count <= 0;
      ms_pulse <= '1';
    else
      count <= count + 1;
      ms_pulse <= '0';
    end if;
  end if;
end process;
-- negate sel every 1 ms
process(clk, sel, ms_pulse)
begin
  if (clk'event and clk='1') then
    if (ms_pulse = '1') then
      sel <= not sel;
    else
      sel <= sel;
    end if;
  end if;
end process;
-- output ssdcat
ssdcat <= sel; ← ssdcat output

-- assign digit based on sel
digit <= value(7 downto 4) when sel='1'
      else value(3 downto 0);
-- display digit on ssd
process(clk, digit) begin
  if (clk'event and clk='1') then
    case digit is
      when x"0" => ssd <= b"1111110";
      when x"1" => ssd <= b"0110000";
      when x"2" => ssd <= b"1101101";
      when x"3" => ssd <= b"1111001";
      when x"4" => ssd <= b"0110011";
      when x"5" => ssd <= b"1011011";
      when x"6" => ssd <= b"1011111";
      when x"7" => ssd <= b"1110000";
      when x"8" => ssd <= b"1111111";
      when x"9" => ssd <= b"1110011";
      when x"a" => ssd <= b"1110111";
      when x"b" => ssd <= b"0011111";
      when x"c" => ssd <= b"1001110";
      when x"d" => ssd <= b"0111101";
      when x"e" => ssd <= b"1001111";
      when x"f" => ssd <= b"1000111";
      when others => ssd <= b"0000000";
    end case;
  end if;
end process;
↑ SSD output
```

① IP Packager and IP Catalog



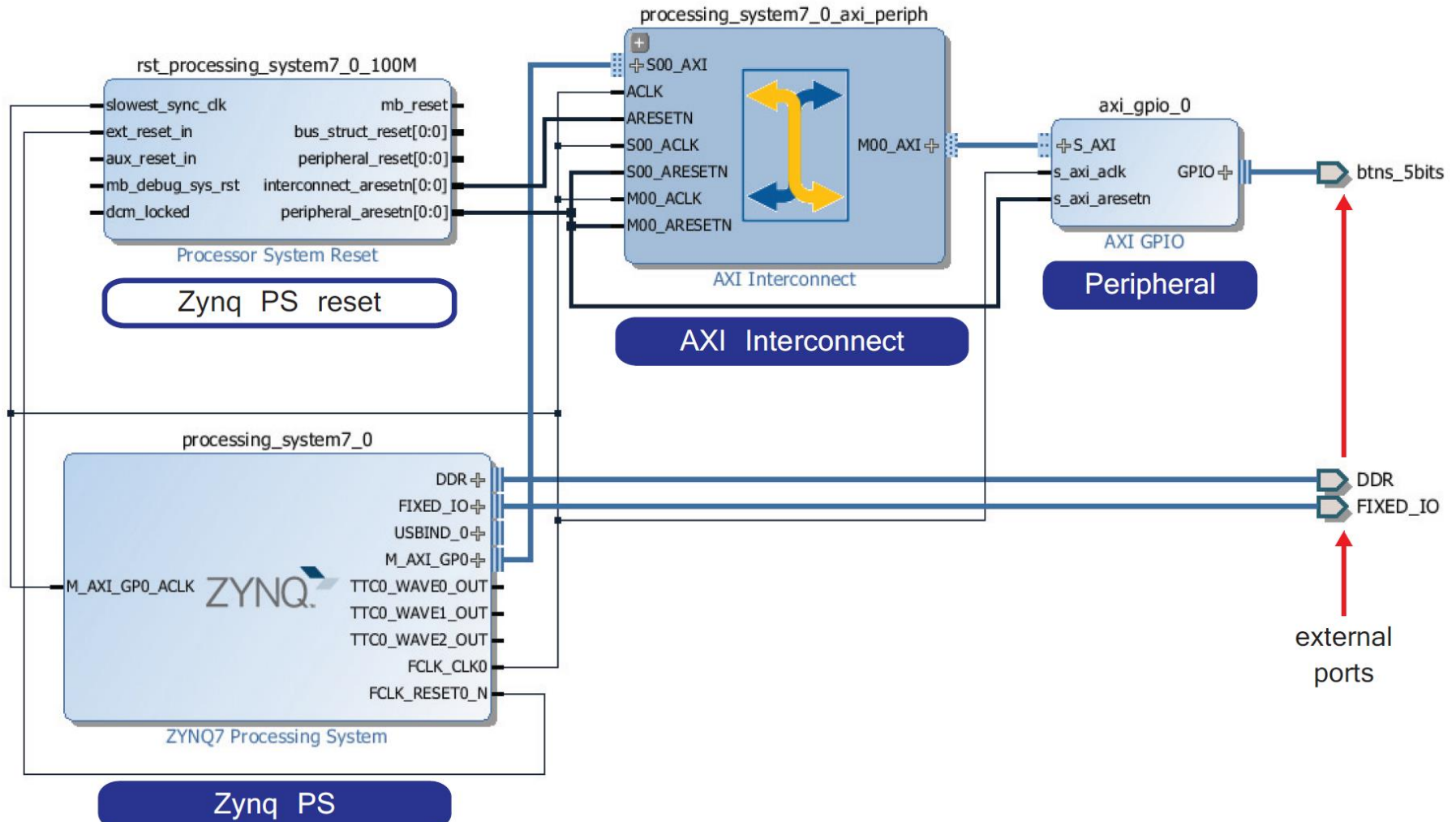
- Vivado IP Packager enables developers to quickly prepare IP for integration in the Vivado IP Catalog.
- Once the IP is selected in a Vivado project, the IP is treated like any other IP module from the **IP Catalog**.



② IP Integration



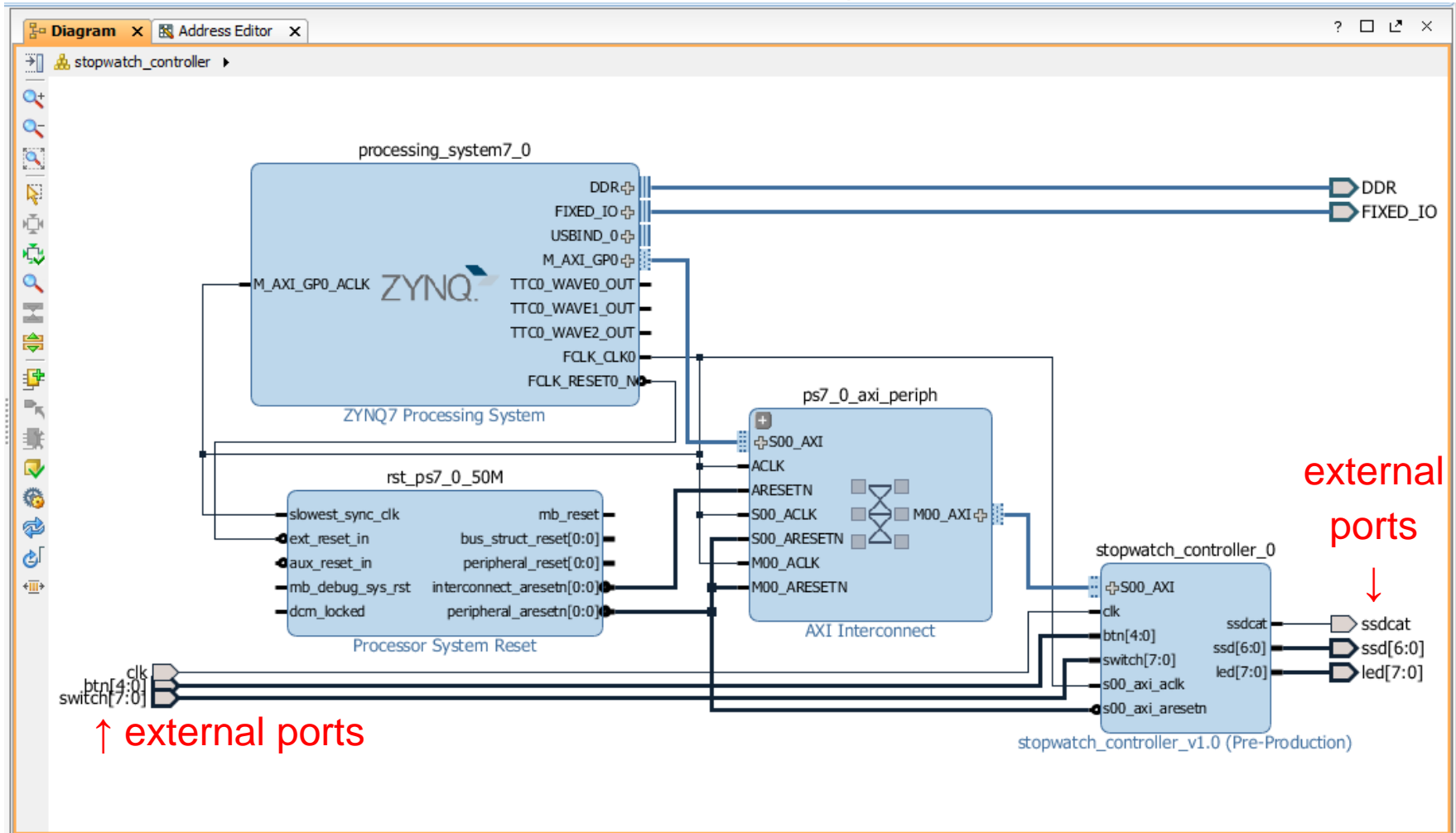
- Vivado IP Integrator provides a graphical “canvas” to configure IP blocks in an *automated* development flow.



Block Design for Stopwatch System

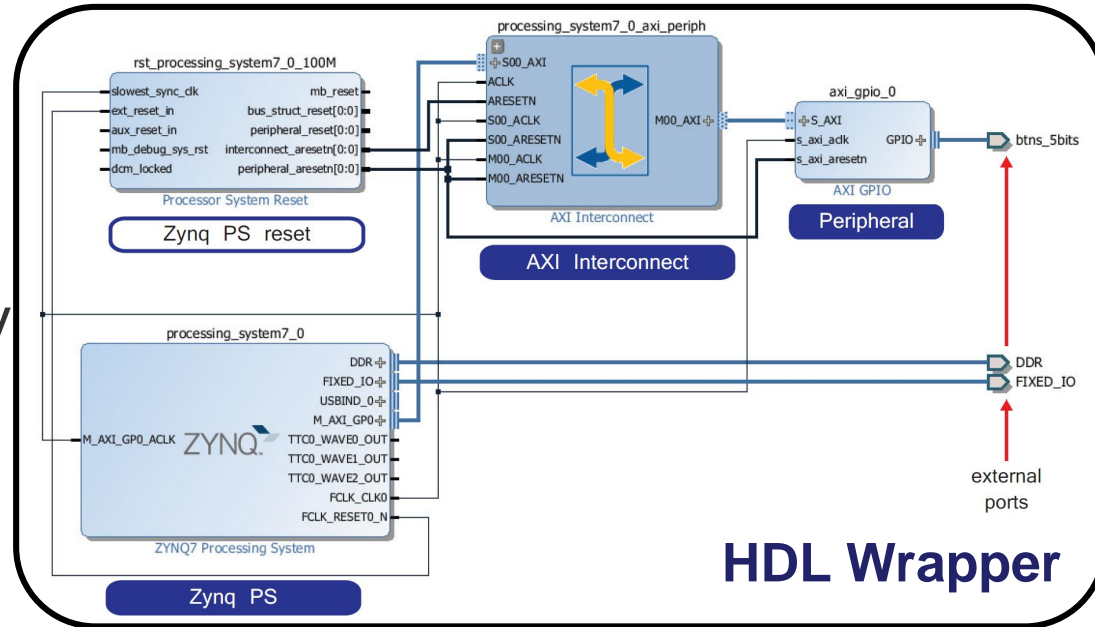


- Vidado will help us to **auto-connect** the stopwatch and the ARM processor through AXI interface.

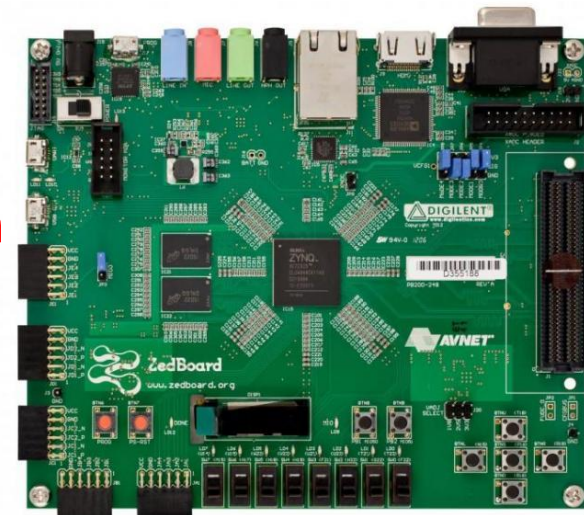
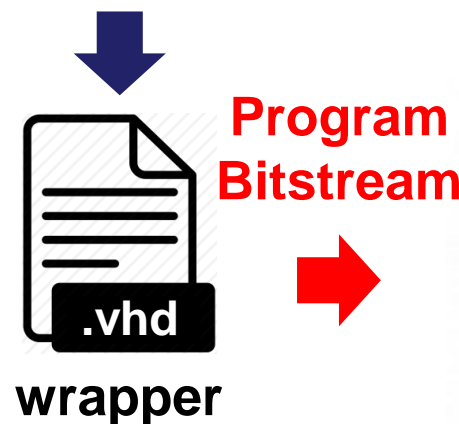


③ HDL Wrapper & ④ Generate Bitstream

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



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





- Rapid Prototyping with Zynq
- Rapid Prototyping (I): Integration of ARM and FPGA
 - Case Study: Lab 08 Software Stopwatch
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware

⑤ ARM Programming



- We need some **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"`
 - `#include "stopwatch_controller.h"`
- 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`
 - ...
- We are creating a **bare metal software program**.
 - There is *nothing but our program* running on the ARM.
 - Thus, our program should really never exit (How? By loop!).

Key: Interfacing via Registers



stopwatch.c

while(1) ← Infinite loop

```
{
    /* INPUT */
    /* btn & switch & time */
    btn_in = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, BTN_ADDR);
    ...
                                     ↑ read register    SWITCH_ADDR

    /* time */
    timer_in = STOPWATCH_CONTROLLER_mReadReg(SW_BASE, TIMER_ADDR);
    u32 time_display;
...
    ... ← User logic for determining the time to be displayed on LED and SSD

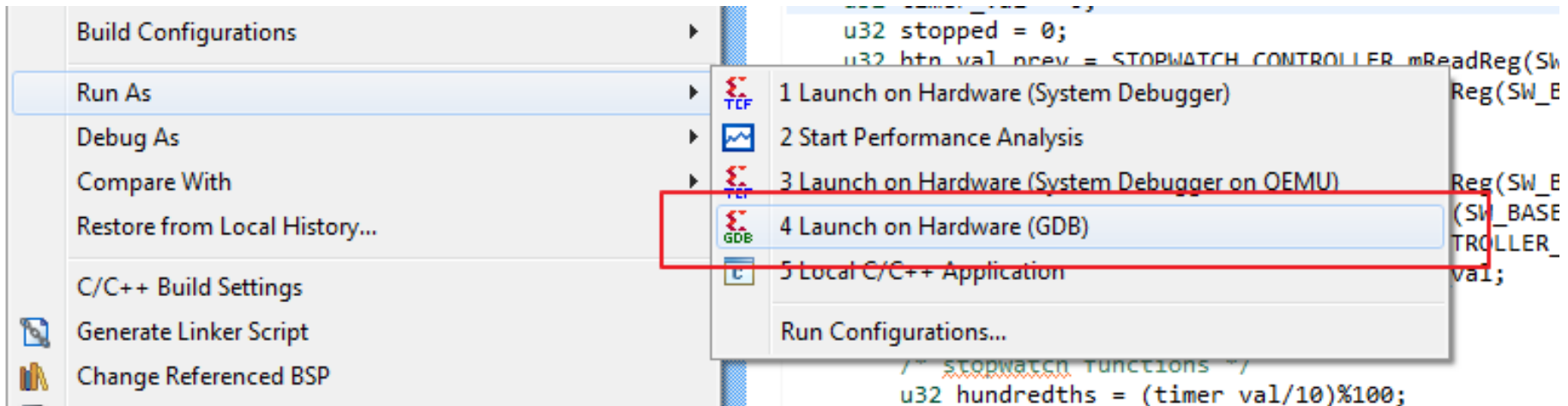
    /* OUTPUT */
    /* led & ssd */
    led_out = time_display;
    STOPWATCH_CONTROLLER_mWriteReg(SW_BASE, LED_ADDR, led_out);
    ...
                                     ↑ write register    SSD_ADDR

    /* FEEDBACK */ ← Like the states for FSMs
    btn_in_prev = btn_in;
    switch_in_prev = switch_in;
}
```


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 to **automatically** compile, link, and load your program.

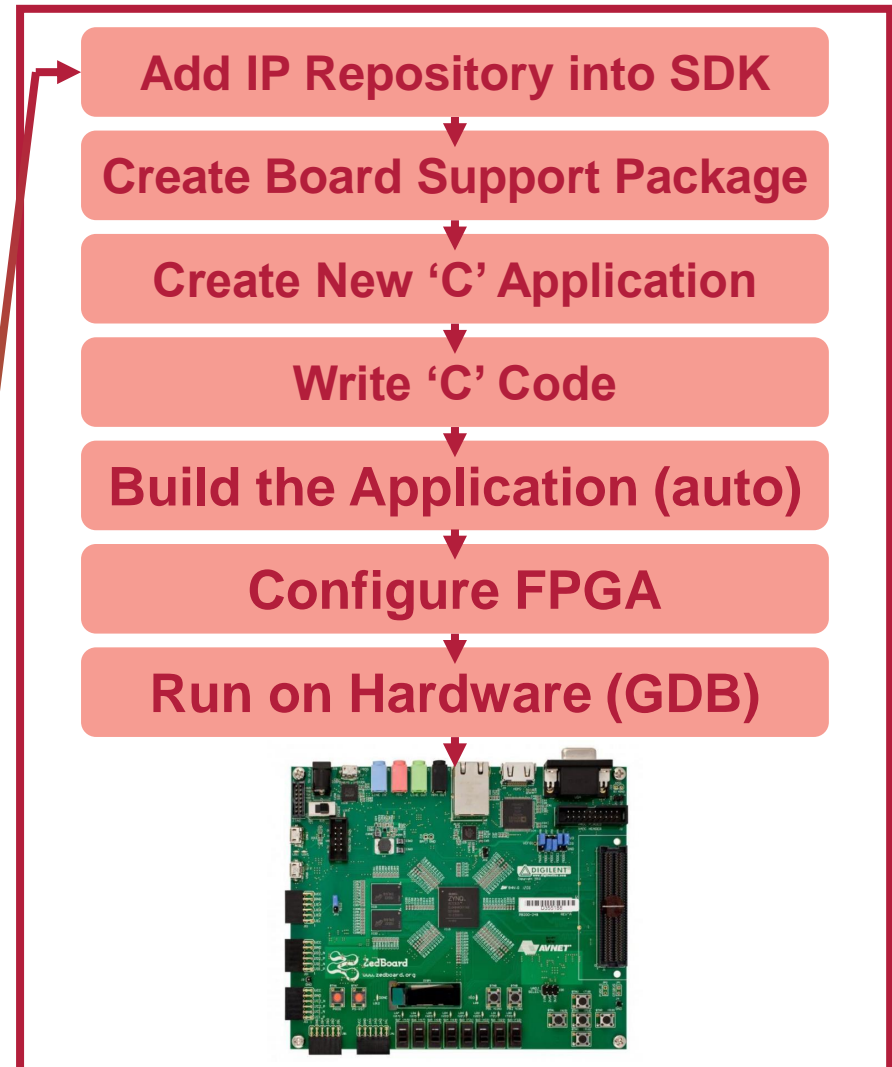


Design Flow of ARM-FPGA Integration



VIVADO

SDK
Software Development Kit





- Rapid Prototyping with Zynq
- Rapid Prototyping (I): Integration of ARM and FPGA
 - Case Study: Lab 08 Software Stopwatch
 - IP Block Design (Xilinx Vavido)
 - ① IP Block Creation & AXI Interfacing
 - ② IP Integration
 - ③ HDL Wrapper
 - ④ Generate Bitstream
 - ARM Programming (Xilinx SDK)
 - ⑤ ARM Programming
 - ⑥ Launch on Hardware