EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

FPGA-BASED EMBEDDED SYSTEM DEVELOPMENT

(VEMIVIB334BR)

Created by Zsolt Voroshazi, PhD

voroshazi.zsolt@mik.uni-pannon.hu

Updated: 6. Mar. 2024.

SZÉCHENYI 2020

BEFEKTETÉS A JÖVŐBE

4. XILINX VIVADO

Creating **BSB** - **B**ase **S**ystem **B**uild and Board „bring-up”

BEFEKTETÉS A JÖVŐBE

Topics covered

1. Introduction – Embedded Systems
2. FPGAs, Digilent ZyBo development platform
3. Embedded System - Firmware development environment (Xilinx Vivado – „EDK” Embedded Development)
4. Embedded System - Software development environment (Xilinx VITIS – „SDK”)
5. **Embedded Base System Build (and Board Bring-Up)**
6. Adding Peripherals (from IP database) to BSB
7. Adding Custom (=own) Peripherals to BSB
8. Development, testing and debugging of software applications – Xilinx VITIS (SDK)
9. Design and Development of Complex IP cores and applications (e.g. camera/video/ audio controllers)
10. HW-SW co-simulation and testing(Xilinx Vivado ChipScope)
11. Embedded Operation System I.: Application development, testing, device drivers, and booting
12. Embedded Operation System II.: setting and starting Linux system on ARM/MicroBlaze processor

Important notes & Tips

- Make sure that the path of the Vivado/VITIS project to be created does NOT contain **accented** letters or "White-space" characters!
- Have permissions on the drive you are working on:
  - If possible, DO NOT work on a network / USB drive!
- The name of the project and source files should NOT start with a number, but they can contain a number! (due to VHDL)
- Use case-sensitive letters consistently in source file and project!
- If possible, the name of the project directory, project and source file(s) should be different and refer to their function for easier identification of error messages.
- The directory path should be no longer than 256 characters!

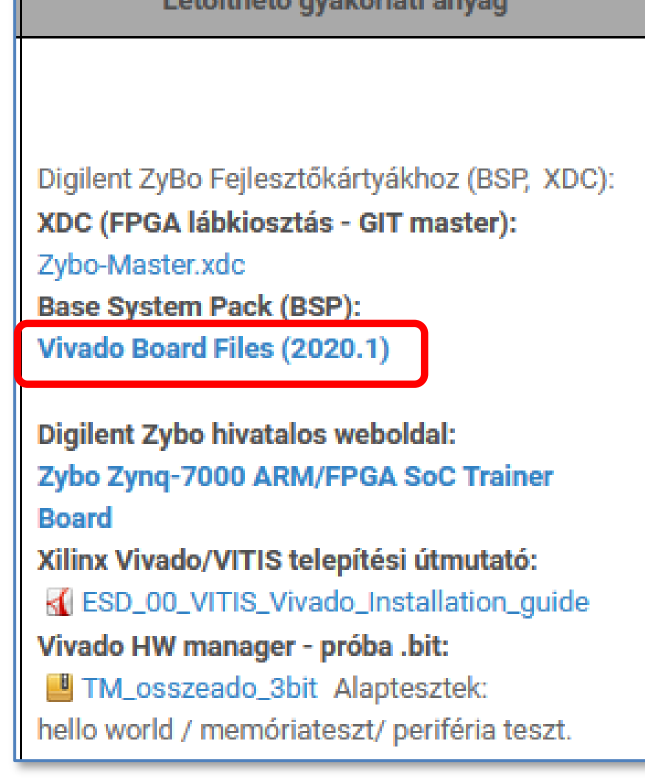
Set Hardware platform I.

- In order our „**ZyBo**” as HW platform can be selectable in the Vivado development environment, the following steps must be taken (for the first time only):
- Step 1.) Download the appropriate support package from the Laboratory’s webpage:

  (Link will redirect to the Digilent website):

  Digilent ZYBO

  – Support package (**):



Set Hardware platform II. (cont.)

- ** Direct Link for Digilent board support package: https://github.com/Digilent/vivado-boards/archive/master.zip

Step 2.)

- COPY the entire contents of the **\ vivado-zybo-master \ new** subdirectory in the package to the Xilinx subdirectory by keeping the directory structure in it
- TO → `<Xilinx_install_dir> \ Vivado \` **`202x.y`** `\ data \ boards \ board_files`

Subdirectory.

- We will use the latest **`B.4`** verison of board files!

XILINX VIVADO DESIGN SUITE

Creating Embedded Base System

BEFEKTETÉS A JÖVŐBE

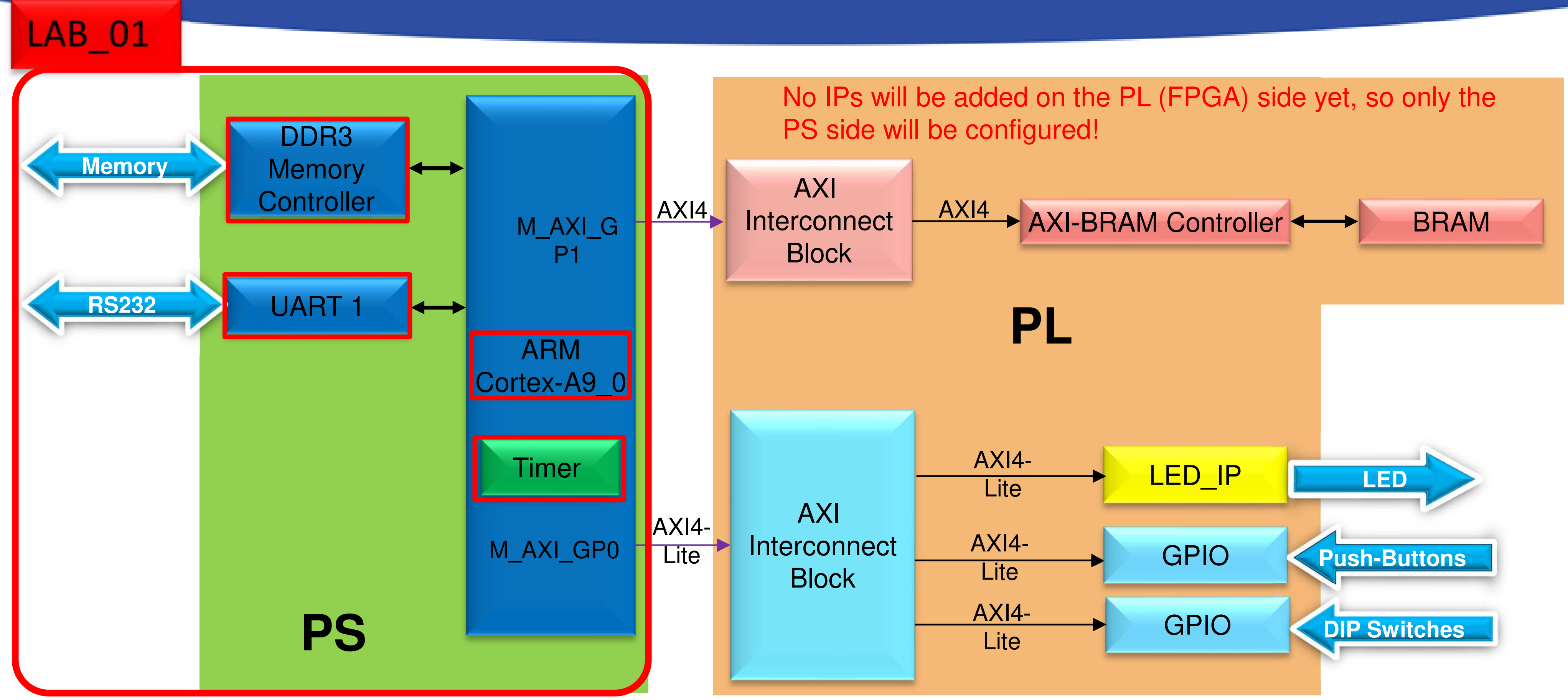
Task

- Creating a new project in Xilinx Vivado
  - Base System Builder (BSB) - Block Designer
  - Create a simple **ARM / AXI** based embedded base system (BSB)
  - with the addition of **AXI-Lite** “bus” interface-based **Xilinx IPs** (Intellectual Property).
- Creating a software application (from pre-defined templated) in Xilinx VITIS

Main steps to solve the task

- Create a new project using the **Xilinx Vivado (IPI)** embedded system designer,
- Overview of the created project,
  - *(Implementation and Bitstream generation if PL side is also configured!)*
- Create a *„Hello world"* and *„Memory Test"* applications running on ARM by using the Xilinx VITIS environment (~SDK),
- Verification of the completed embedded system and software application test on Digilent ZyBo.

Test system to be implemented



PS side:

- **ARM hard-processor (Core0)**
- **Internal OnChip-RAM controller**
- **UART1 (serial) interface / Global Timer**
- **External DDR3 memory controller**

PL (FPGA) side is empty, not configured: pl.

- GPIOs, LED_IPs,
- AXI interconnection, etc. not used.

Starting Vivado

Start menu → Programs → Xilinx Design Tools → Vivado 2020.x

Not Xilinx Vivado HLS !

Vivado 2020.1

File Flow Tools Window Help Quick Access

VIVADO HLx Editions

XILINX

Quick Start

Create Project >
Open Project >
Open Example Project >

Tasks

Manage IP >
Open Hardware Manager >
XHub Stores >

Learning Center

Documentation and Tutorials >

Recent Projects

project_1
F:/Vivado_2020.1/project_1

project_2_zed
E:/Vivado_2019.2/project_2_zed

project_1
E:/Vivado_2019.2/project_1

counter4div
E:/__PE__/_VIRT_/IPARI_SZKG_2020_Bemutato/peldak/counter4div

traffic_moore_delay_dialight
E:/__PE__/_VIRT_/IPARI_SZKG_2020_Bemutato/peldak/traffic_moore_delay_dialight

project_1
E:/__PE__/___OKTATAS___/TERV_MOD_PLD_VIB544T/2019_vivado/_ZHk_/project_1

traffic_moore_dia_led
e:/__PE__/___OKTATAS___/DIGITALIS_ARAMKOROK_II_VI344D_2019_osz/hatter_Addings_/vivado_traffic_m...

traffic_moore
E:/__PE__/___OKTATAS___/TERV_MOD_PLD_VIB544T/2018_vivado/E_TMPLD_2018_Vivado/traffic_moore

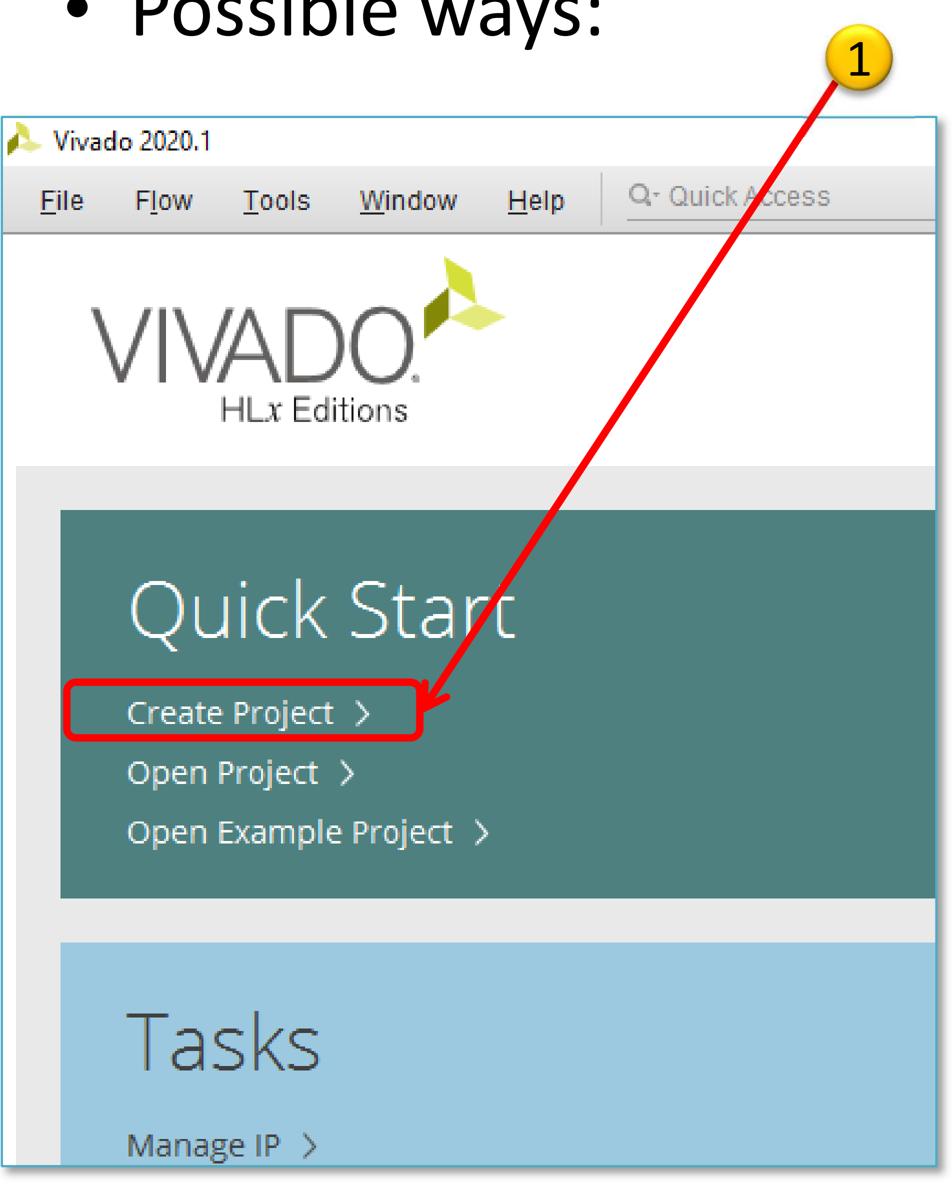
mux2_1
E:/__PE__/___OKTATAS___/DIGITALIS_ARAMKOROK_II_VI344D_2019_osz/hatter_Addings_/vivado_static_h...

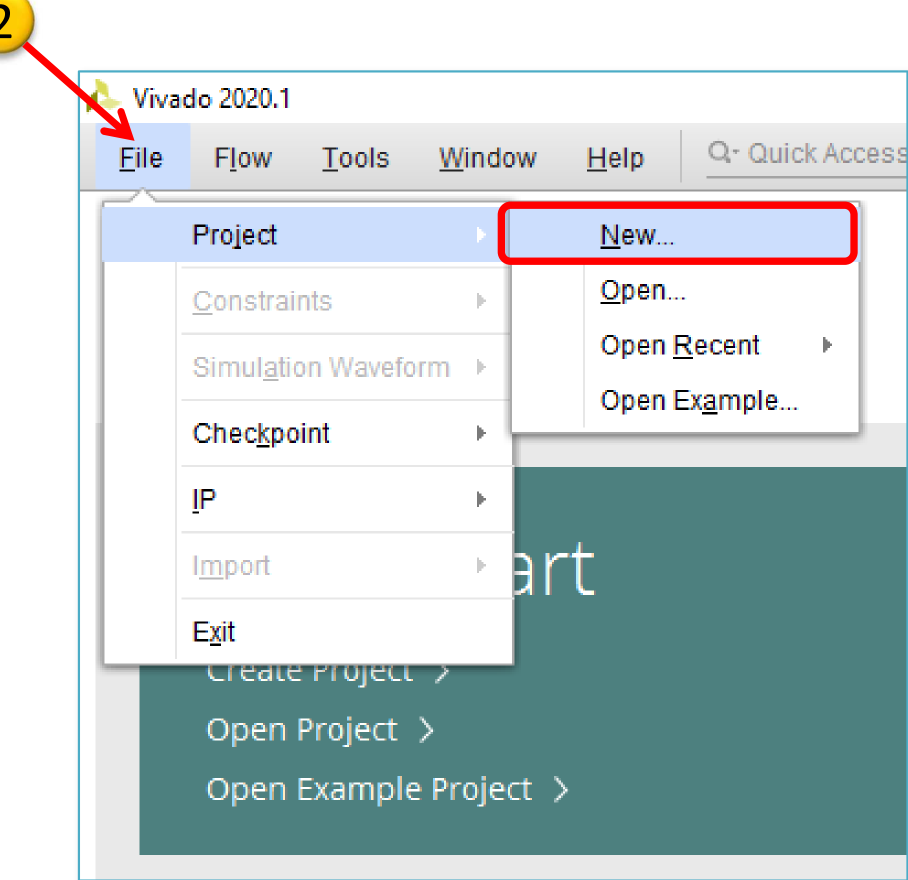
traffic
E:/__PE__/___OKTATAS___/TERV_MOD_PLD_VIB544T/2019_vivado/Radakovics/traffic

Tcl Console

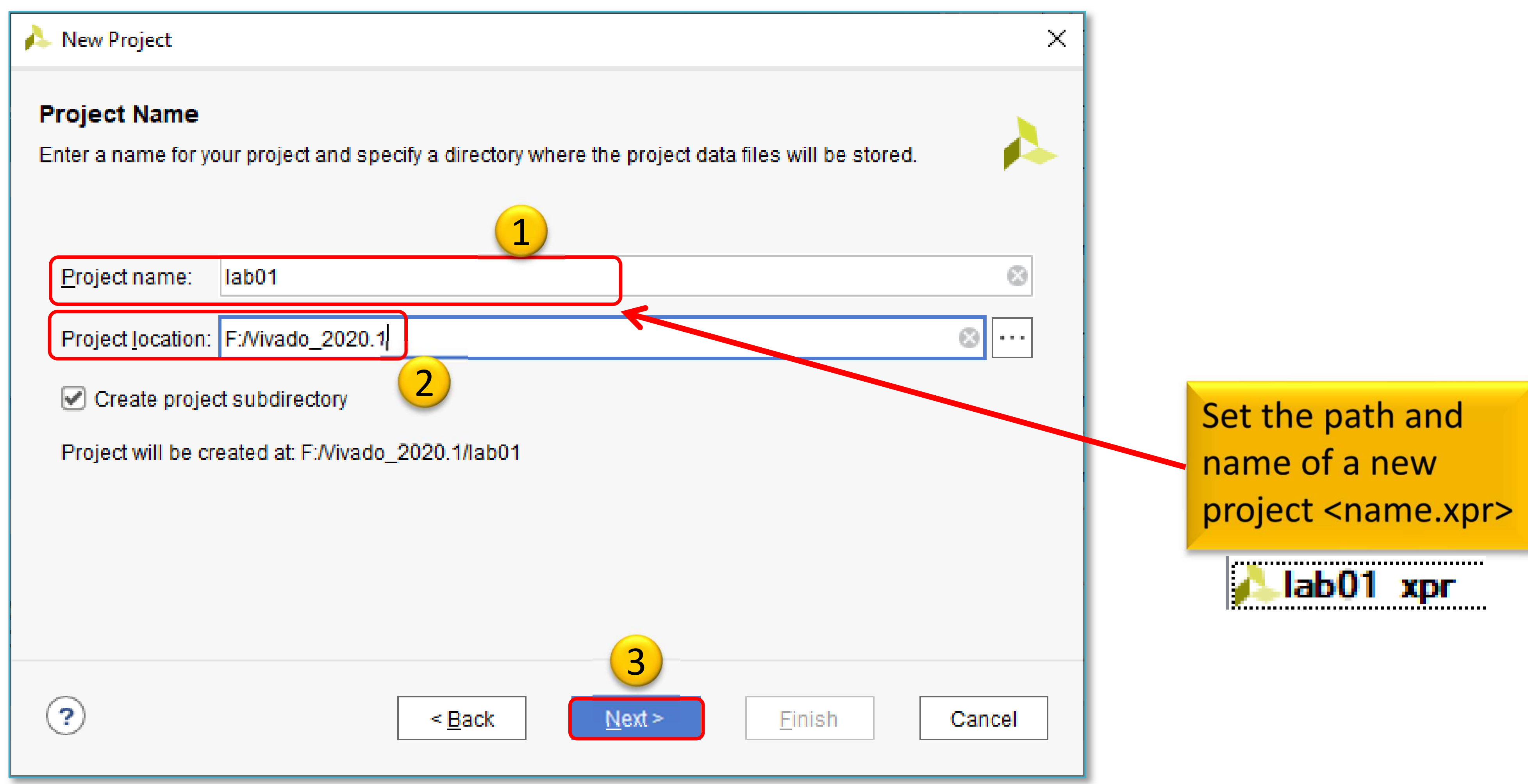
Create a new project

- Possible ways:

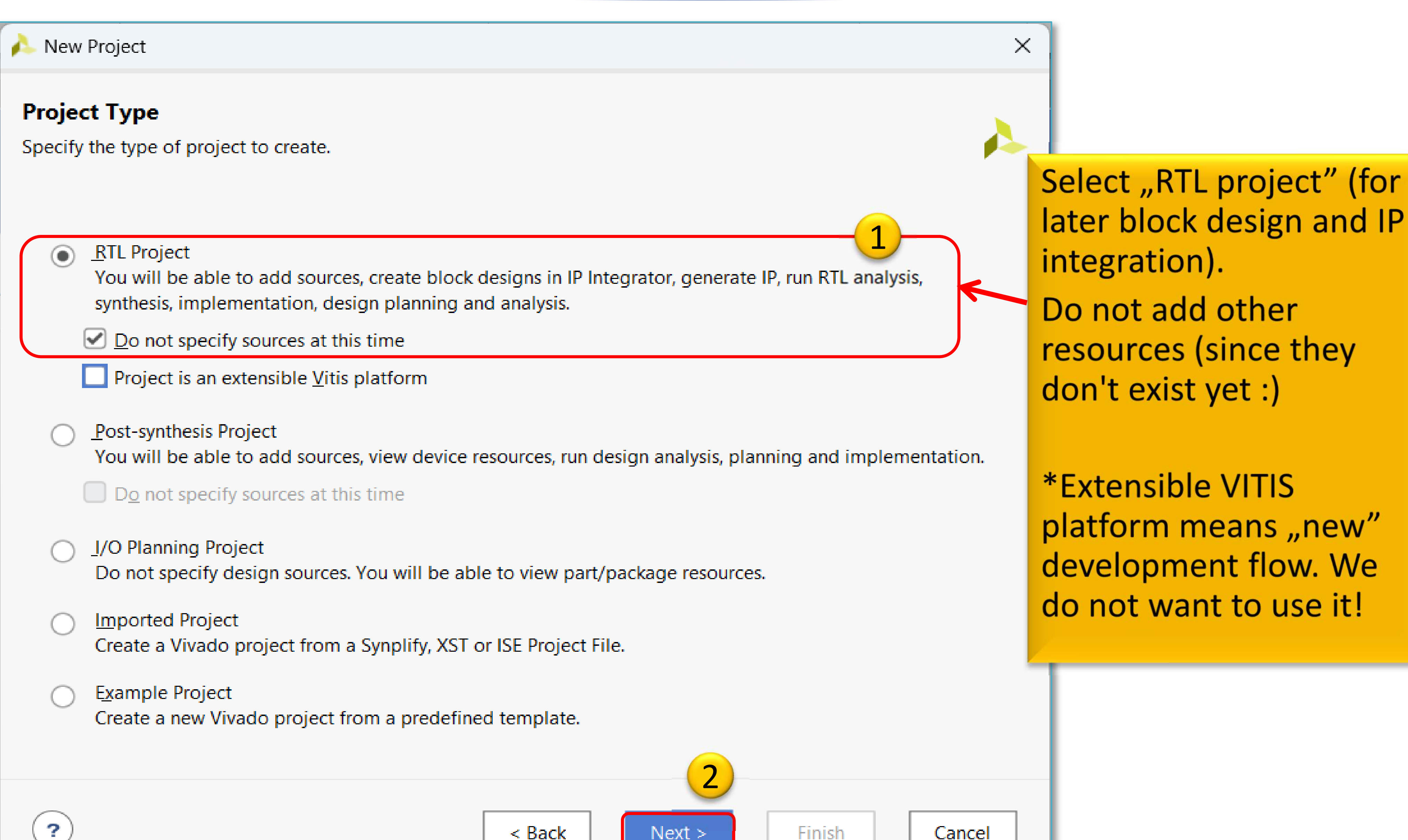




Create a new project (cont.)

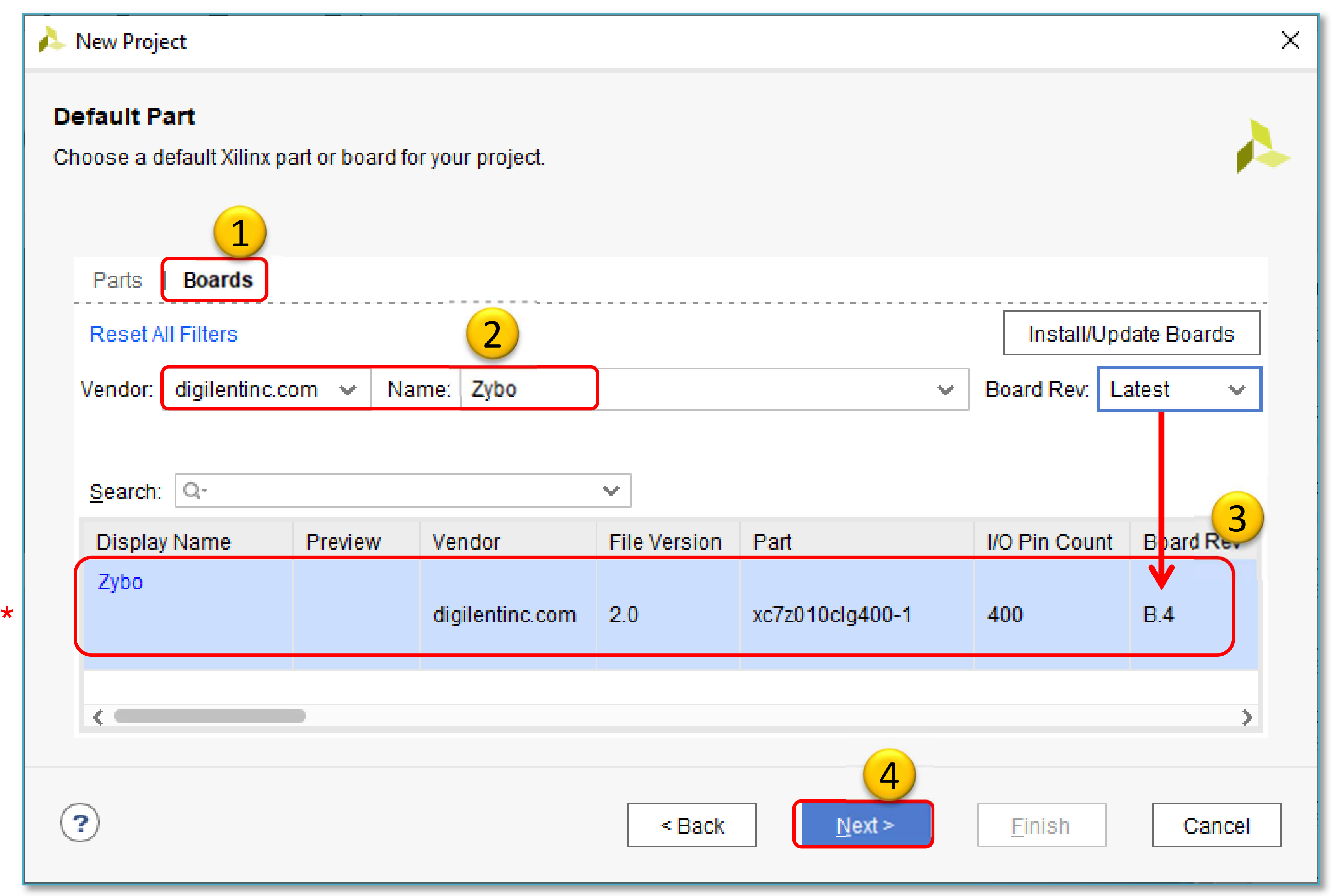


Create a new project (cont.)

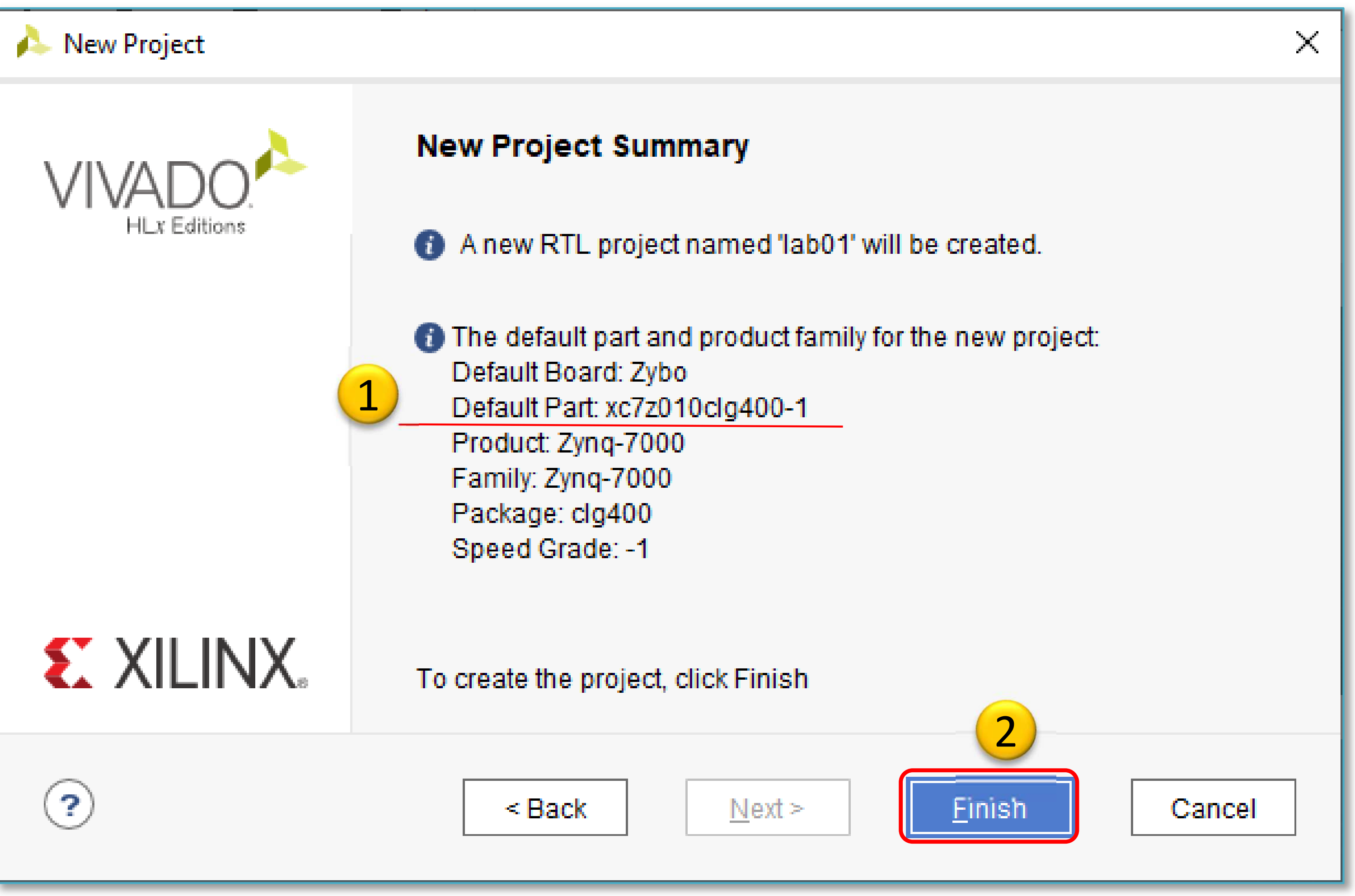


Choose HW development board

** „Zybo" can only be selected after installing the support package (see slides 5-6)

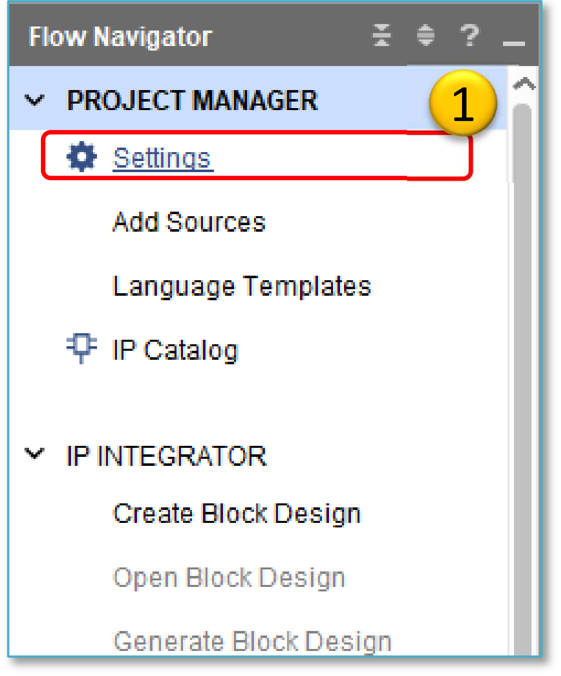


Project summary



Project settings - VHDL

- Project Manager → Settings → General → Select VHDL, then OK.



Settings

2

Project Settings
General
Simulation
Elaboration
Synthesis
Implementation
Bitstream
IP

Tool Settings
Project
IP Defaults
XHub Store
Source File
Display
WebTalk
Help
Text Editor
3rd Party Simulators
Colors
Selection Rules
Shortcuts
Strategies
Window Behavior

General
Specify values for various settings used throughout the design flow.
These settings apply to the current project.

Name: lab01
Project device: Zybo (xc7z010clg400-1)
Target language: VHDL
3
Default library: xil_defaultlib
Top module name:

Language Options
Verilog options: verilog_version=Verilog 2001
Generics/Parameters:
Loop count: 1,000

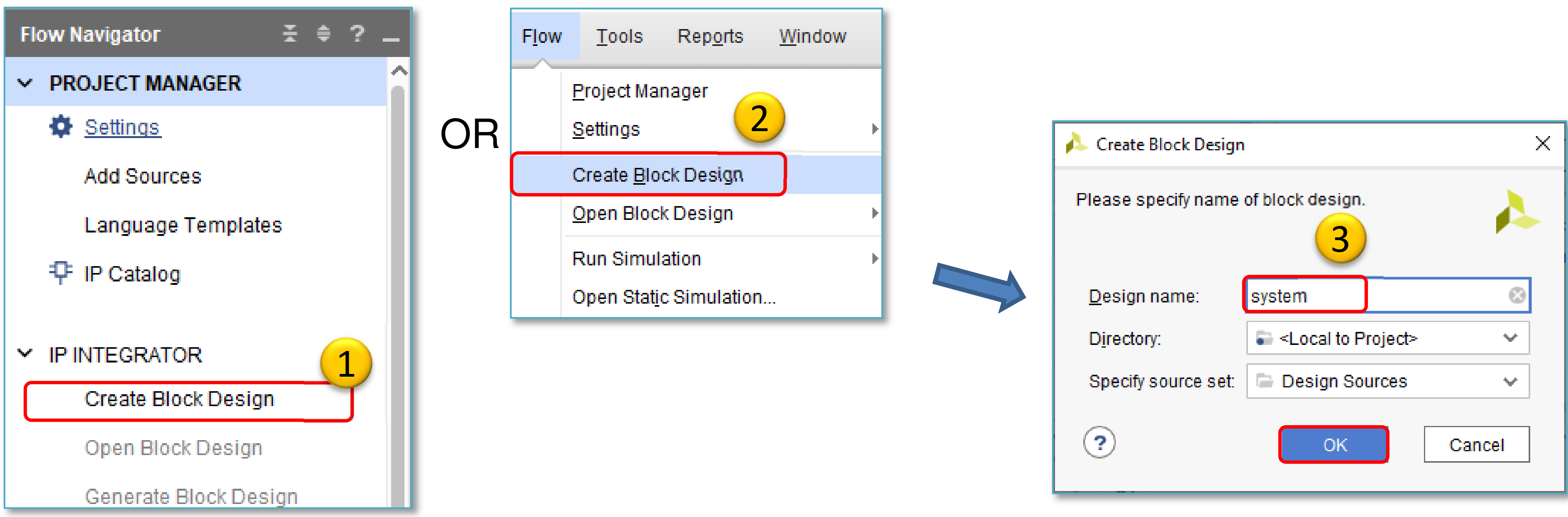
A top-level “wrapper” is generated from the block design to this language (VHDL).

OK Cancel Apply Restore...

Embedded System – IP Integrator (IPI)

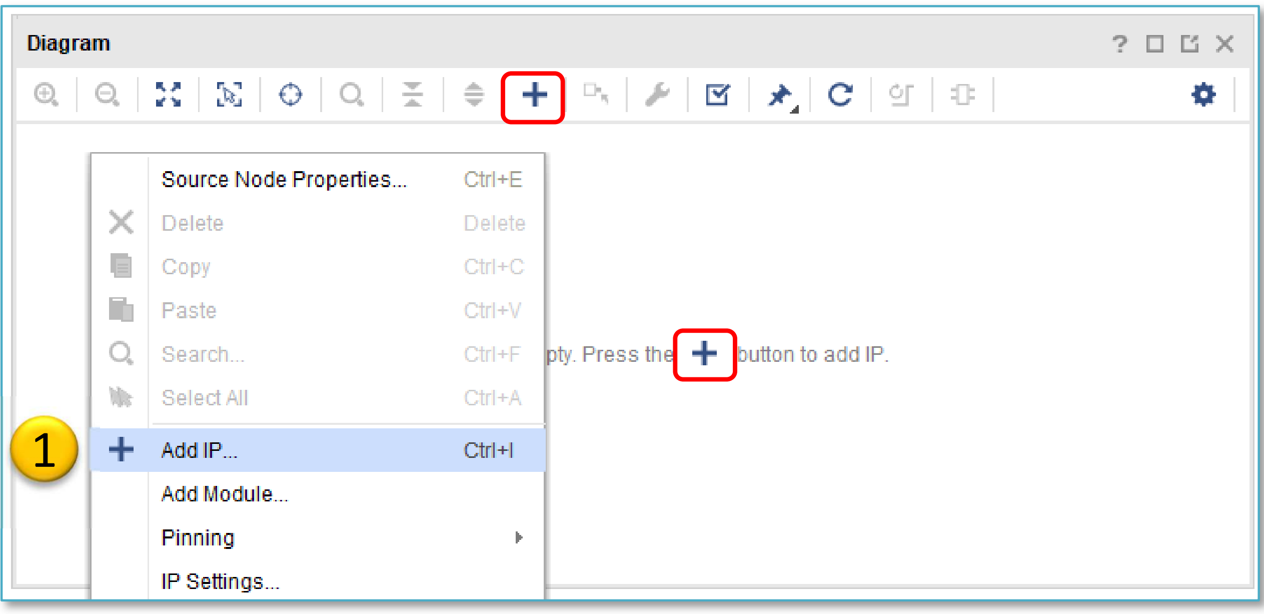
IP Integrator - Create a new Block Design

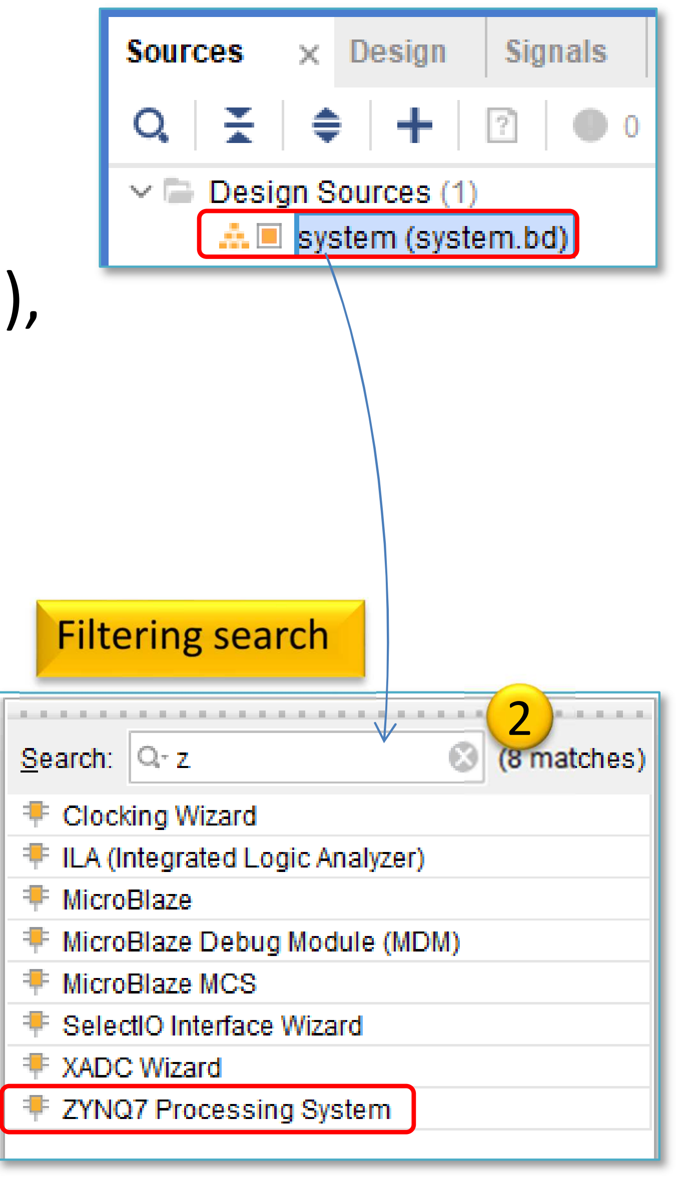
1. Create Block Design, OR
2. Flow menu → Create Block Design,
3. Let it be called as “**system**”. Finally OK.



Embedded System – Add IP

- Select block diagramm (`system.bd`)
- Add IP (CTRL+I) +
  - Select "ZYNQ7 Processing System" (= PS),
  - then double click on it.





CTRL+Q: IP details

IP Catalog – IP database

Open the IP Catalog:

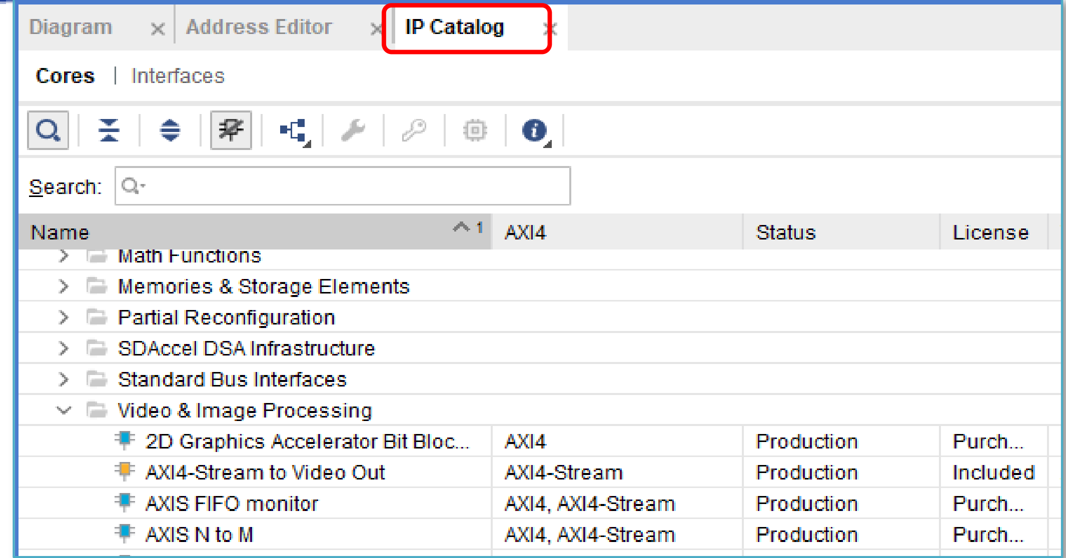
- Project Manager → IP Catalog

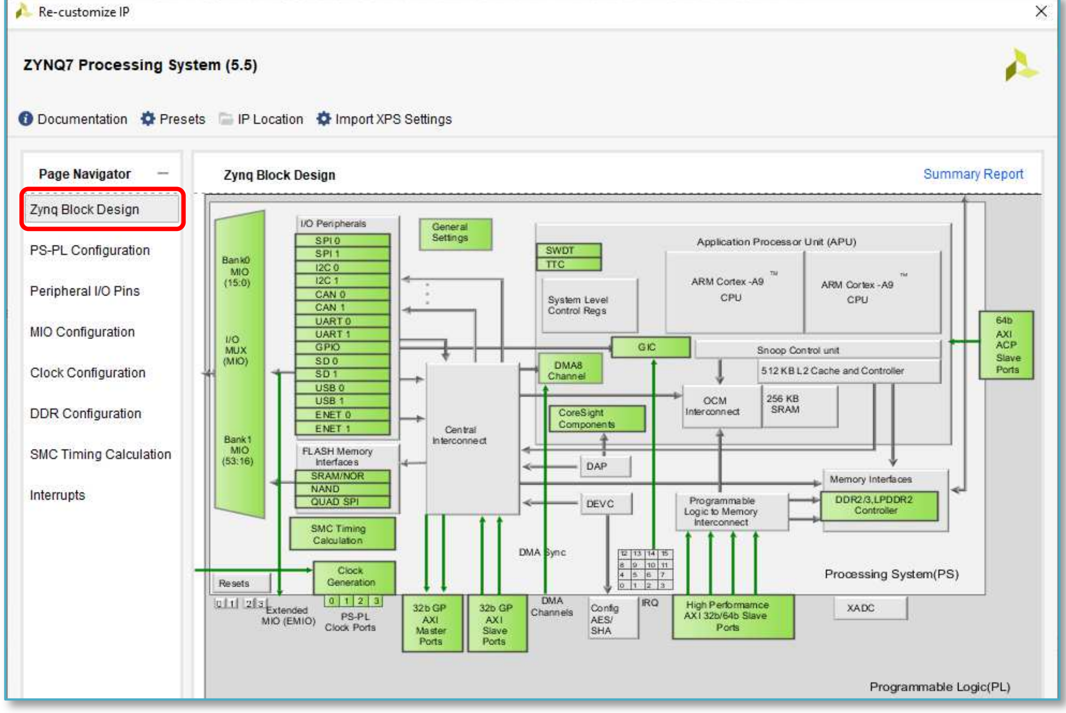
IP integration support

„included” = free-of-charge vs.

„purchased” = licensable (trial period ~90 days)

- fast IP parameterization
- Vivado IP GUI:
  - compatibility
  - access to datasheets
  - changelog, webpage
- Vivado “design flow”: Support for Synthesis and Implementation
- .Tcl support





IP Peripherals

- Bus and bridge controllers
  - AXI to AXI connector
  - Local Memory Bus (LMB)
  - AXI Chip to Chip
  - AHB-Lite to AXI
  - AXI4-Lite to APB
  - AXI4 to AHB-Lite…
- Debug cores
  - Integrated Logic Analyzer
- DMA and Timers
  - Watchdog, fixed interval
- Inter-processor communication
  - Mailbox, Mutex
  - ….
- External peripheral controller Memory and memory controller
- High-speed and low-speed communication peripherals
  - AXI 10/100 Ethernet MAC controller
  - Hard-core tri-mode Ethernet MAC
  - AXI IIC
  - AXI SPI
  - AXI UART…
- Other cores
  - System monitor
  - Xilinx Analog-to-Digital Converter (XADC)
  - Clock generator, System reset module
  - interrupt controller
  - Traffic Generator, Performance monitor
  - ….

IP repository (directory structure)

- Directory structure of IP cores (two options: local project directory or global directory = Repository)
  - **{component}.xml**–based descriptor
  - /MyProcessorIPLib directory (user defined)
    - Repo's subdirectories:

    Project Manager →
    Settings →
    IP Defaults / Repository tab
  - *%XILINX_INSTALL%\Vivado\2020.X\data\ip*

PI:

```
axi_gpio_v2_0
  bd
  doc
  hdl
    src
      example
      vhdl
  ttcl
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component xmlns:xilinx="http://www.xilinx.com" xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009" xmlr
  <spirit:vendor>xilinx.com</spirit:vendor>
  <spirit:library>ip</spirit:library>
  <spirit:name>axi_gpio</spirit:name>
  <spirit:version>2.0</spirit:version>
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>S_AXI</spirit:name>
      <spirit:displayName>S_AXI</spirit:displayName>
      <spirit:busType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm" spirit:version="1.0"/>
      <spirit:abstractionType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm_rtl" spirit:version="1.0"/>
      <spirit:slave/>
      <spirit:portMaps>
        <spirit:portMap>
          <spirit:logicalPort>
            <spirit:name>ARADDR</spirit:name>
          </spirit:logicalPort>
```

Block diagram – Run Block Automation

Diagram | Address Editor | IP Catalog

Designer Assistance available. Run Block Automation

1

„instance” name (arbitrary)

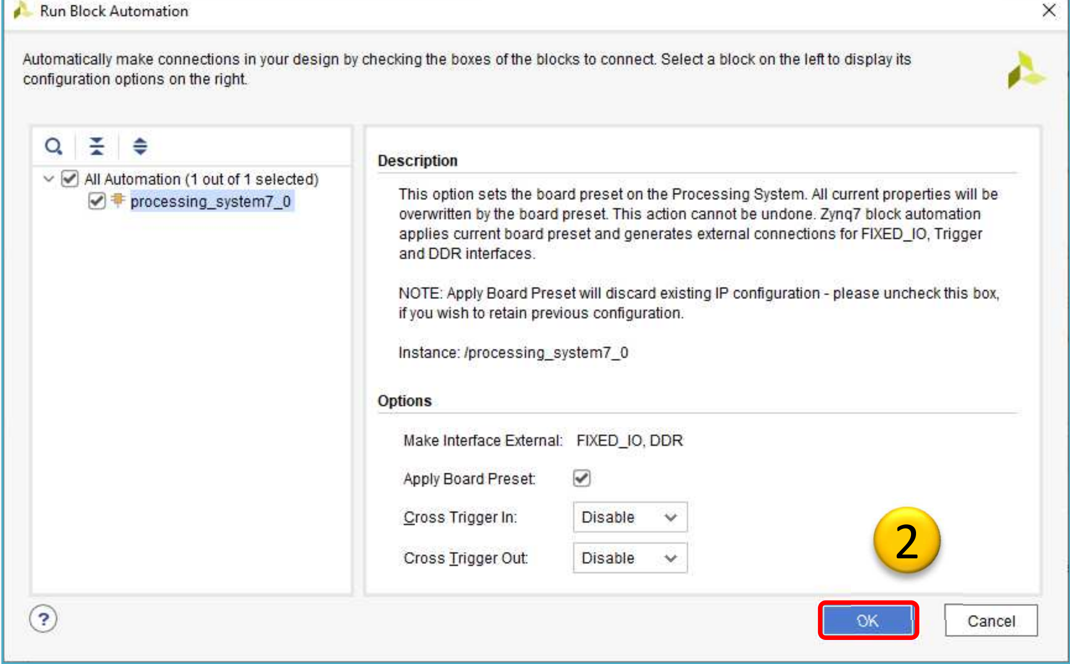
processing_system7_0

DDR
FIXED_IO
M_AXI_GP0_ACLK
ZYNQ
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N

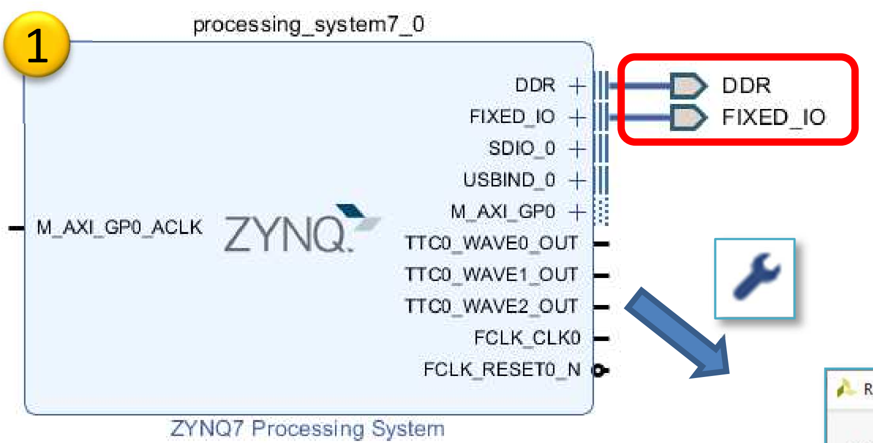
ZYNQ7 Processing System

IP catalog name (fixed)

Keep everything at the default setting. OK.



Zynq Blokk Design (DDR and I/O ports)

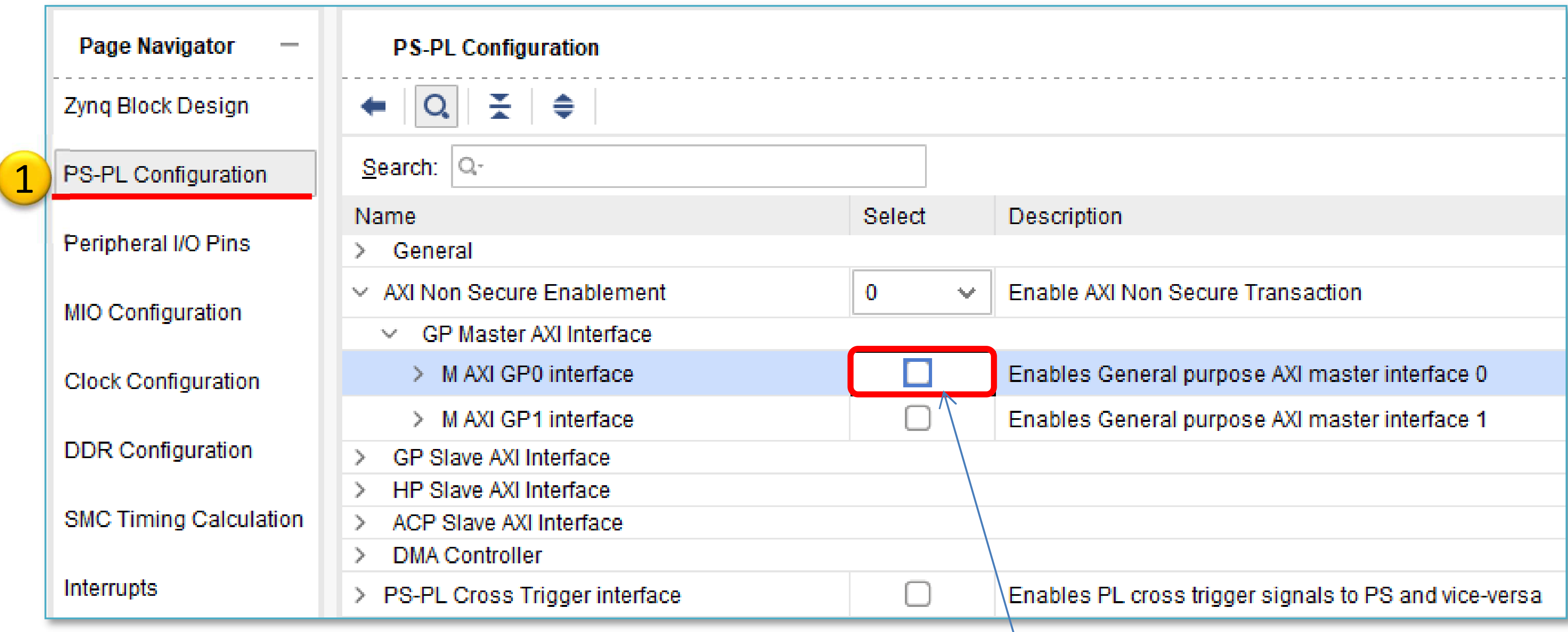


External DDR and IO ports become visible

Green block: configurable, parameterizable blocks
√: blocks are configured and utilized

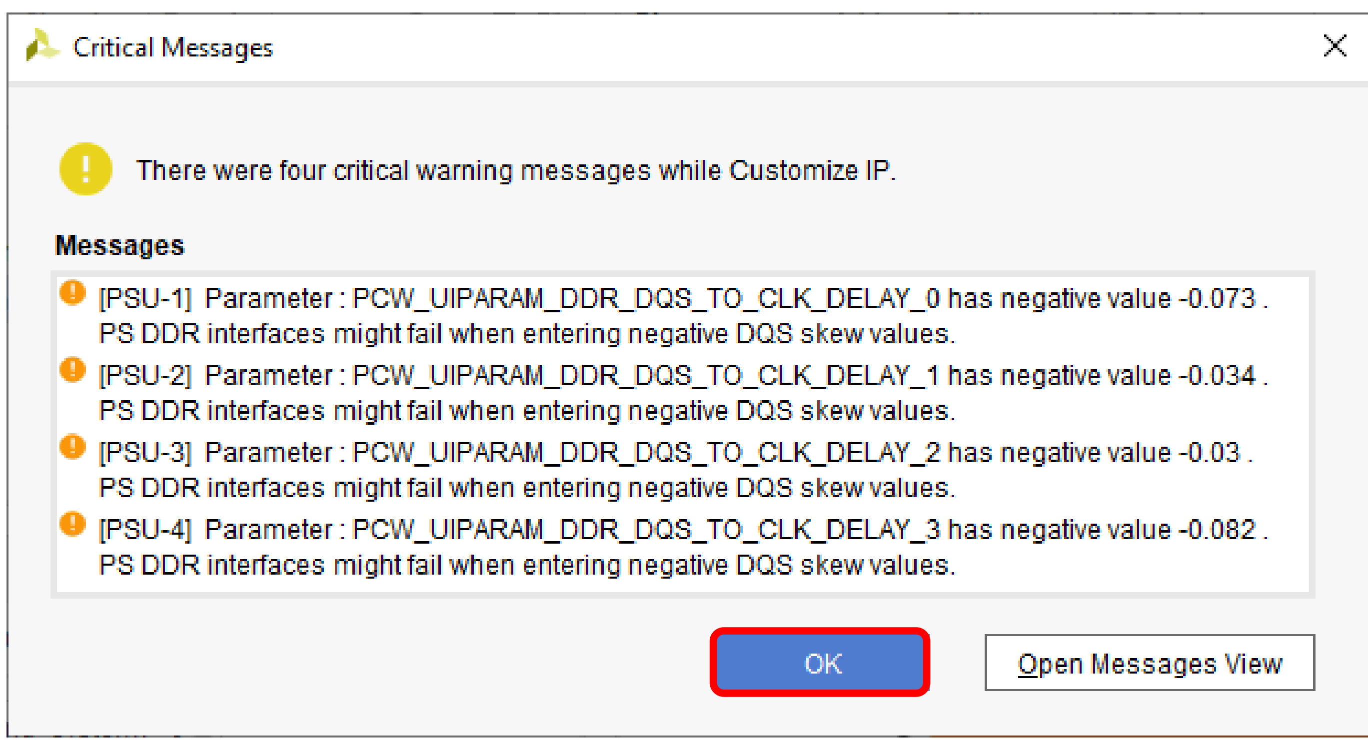
- *Import XPS settings:* It is possible to import the settings (update the factory default preset **.xml** file)

Zynq – PS-PL configuration



- *General [+] ->*
  - *UART**1** Baud rate: 115.200 ?*
  - *Enable Clock Resets* : disable **FCLK_RESET0_N**.
- *AXI Non Secure Enablement [+] ->*
  - *GP Master AXI interface*: disable **M AXI GP0** interface.

Zynq – Critical warning messages



- This is coming from the board definition file for ZYBO. This warning might be considered only for new/custom board design, so it can be ignored! Just click OK.
- *Reference:* https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual?_ga=2.21480123.1048852157.1597218516-1725411217.1597064829#hardware_errata

Zynq PS – Peripheral I/O pins

- Set (√) *UART1* to serial logging.
- All the other blocks are NOT enabled for this time, i.e. uncheck them:
  - **Memory Interfaces** - *Quad SPI Flash*
  - *ENET 0*
  - *USB 0*
  - *SD 0*
  - **Application Processor Unit** - *Timer 0 (TTC0)*
  - **GPIO** - *GPIO MIO*

Zybo schematic.pdf

PS_MIO46_501
PS_MIO47_501
PS_MIO48_501 B12 UART_RXD_OUT
PS_MIO49_501 C12 UART_TXD_IN
PS_MIO50_501 B13 PB0
PS_MIO51_501 B9 PB1
PS_MIO52_501 C10 ETH_MDC
PS_MIO53_501 C11 ETH_MDIO
IC19E

PS MIO pins (0…53)

Page Navigator
Zynq Block Design
PS-PL Configuration
1 Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts

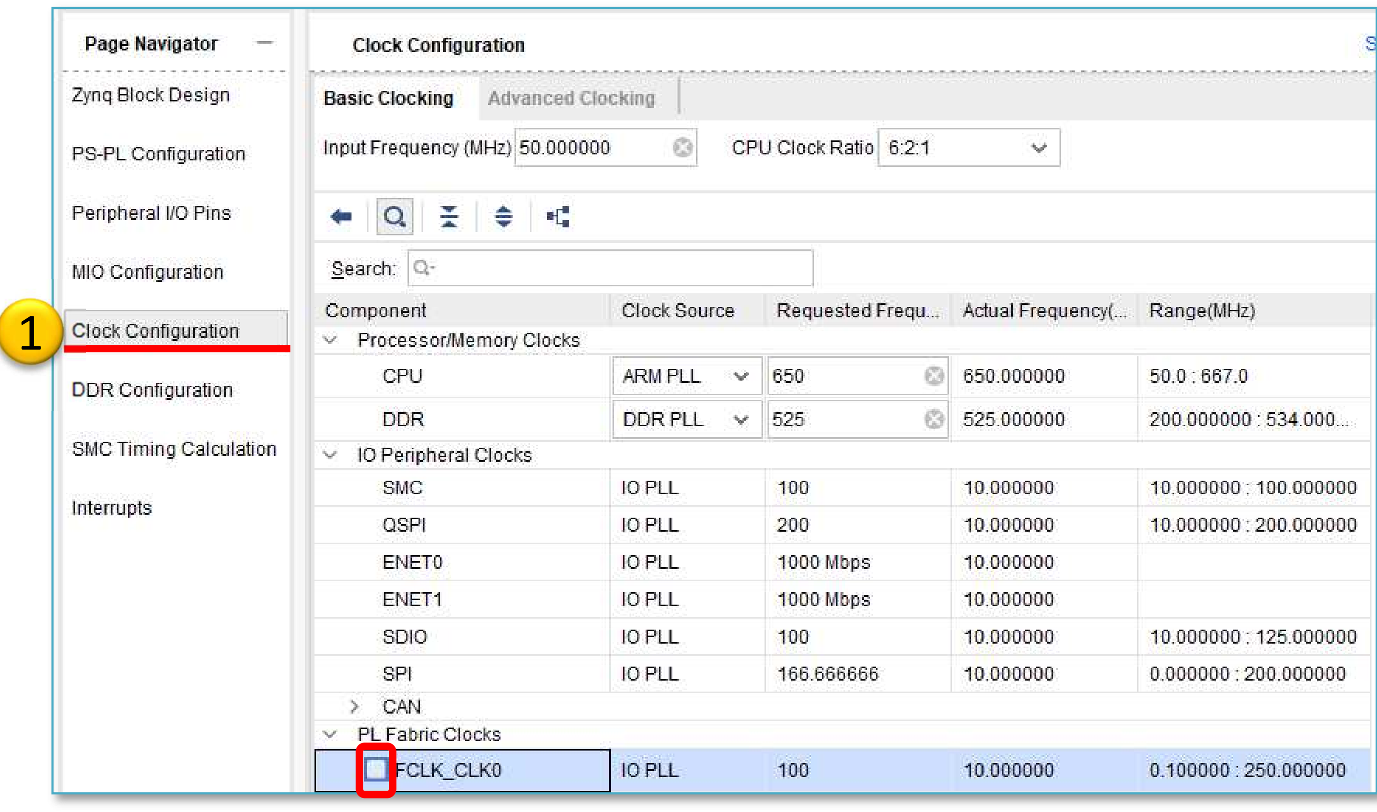
Peripheral I/O Pins
Summary Report
Search:
Peripherals
Ethernet 1
USB 0
USB 1
SD 0
SD 1
SPI 0
SPI 1
2 UART 0
UART 1
I2C 0
I2C 1

UART1: can be enabled by clicking on it.

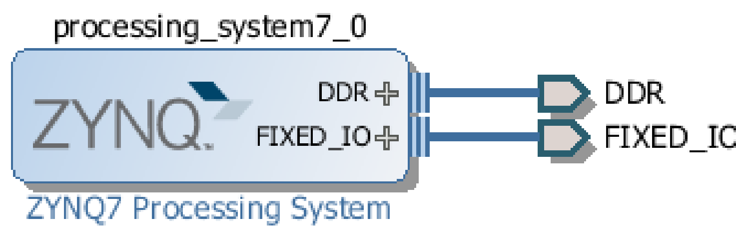
Conflict View: if there is a peripheral MIO conflicts, it indicates an error.

Zynq – Clock configuration

- PL Fabric Clocks (FPGA) →
  - disable FCLK_CLK0



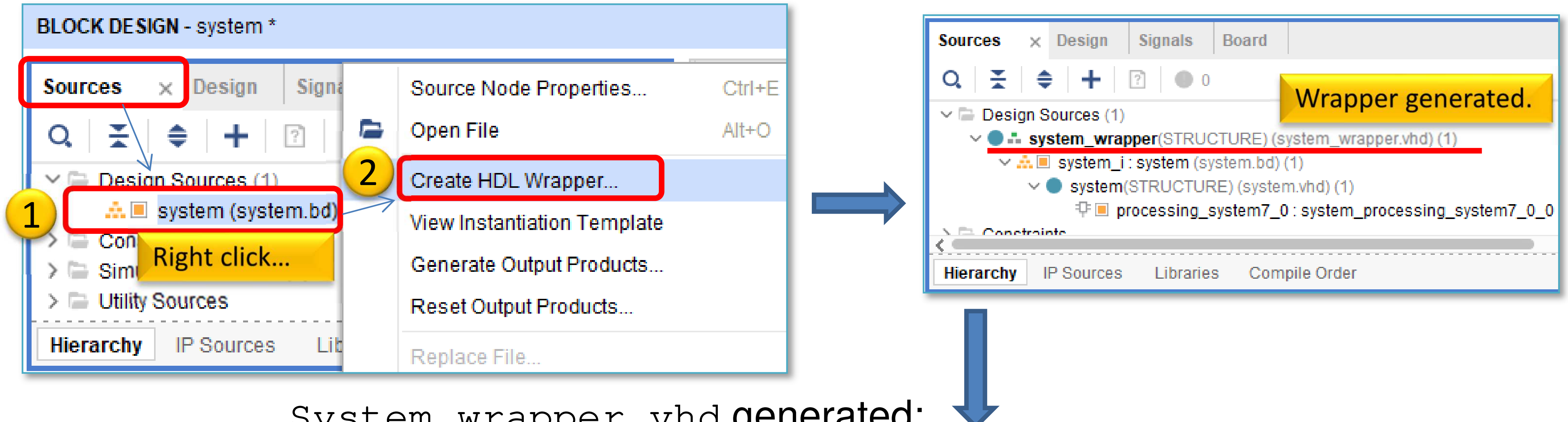
- Finally: OK.
  - Regenerate Layout
  - Validate Design (DRC)



Generate top-level HDL (~wrapper)

- Design Sources → Create HDL wrapper →
  - *„Let Vivado manager wrapper and auto-update"*

Note: without a top-level wrapper, the implementation would not run!



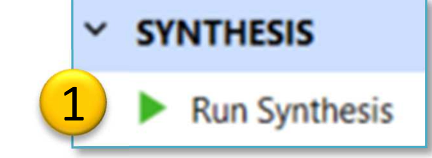
`System_wrapper.vhd` generated:

```
Address Editor   IP Catalog   system_wrapper.vhd
2018.3/lab01/project_1/project_1.srcs/sources_1/bd/system/hdl/system_wrapper.vhd
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  library UNISIM;
13  use UNISIM.VCOMPONENTS.ALL;
14  entity system_wrapper is
15    port (
16      DDR_addr : inout STD_LOGIC_VECTOR ( 14 downto 0 );
```

Export HW → VITIS (~SDK)

Vivado 2020.2: at least an Synthesized Design must be able to be exported to HW!

- **SYNTHESIS→ Run Synthesis**
- **File → Export → Export Hardware…**
- **Select Pre-sythesis as an option**



Since the PL (FPGA) side has not been configured, a bitstream generation is not required at now.

Export Hardware Platform

Output

Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.

2

Pre-synthesis
This platform includes a hardware specification for downstream software tools.

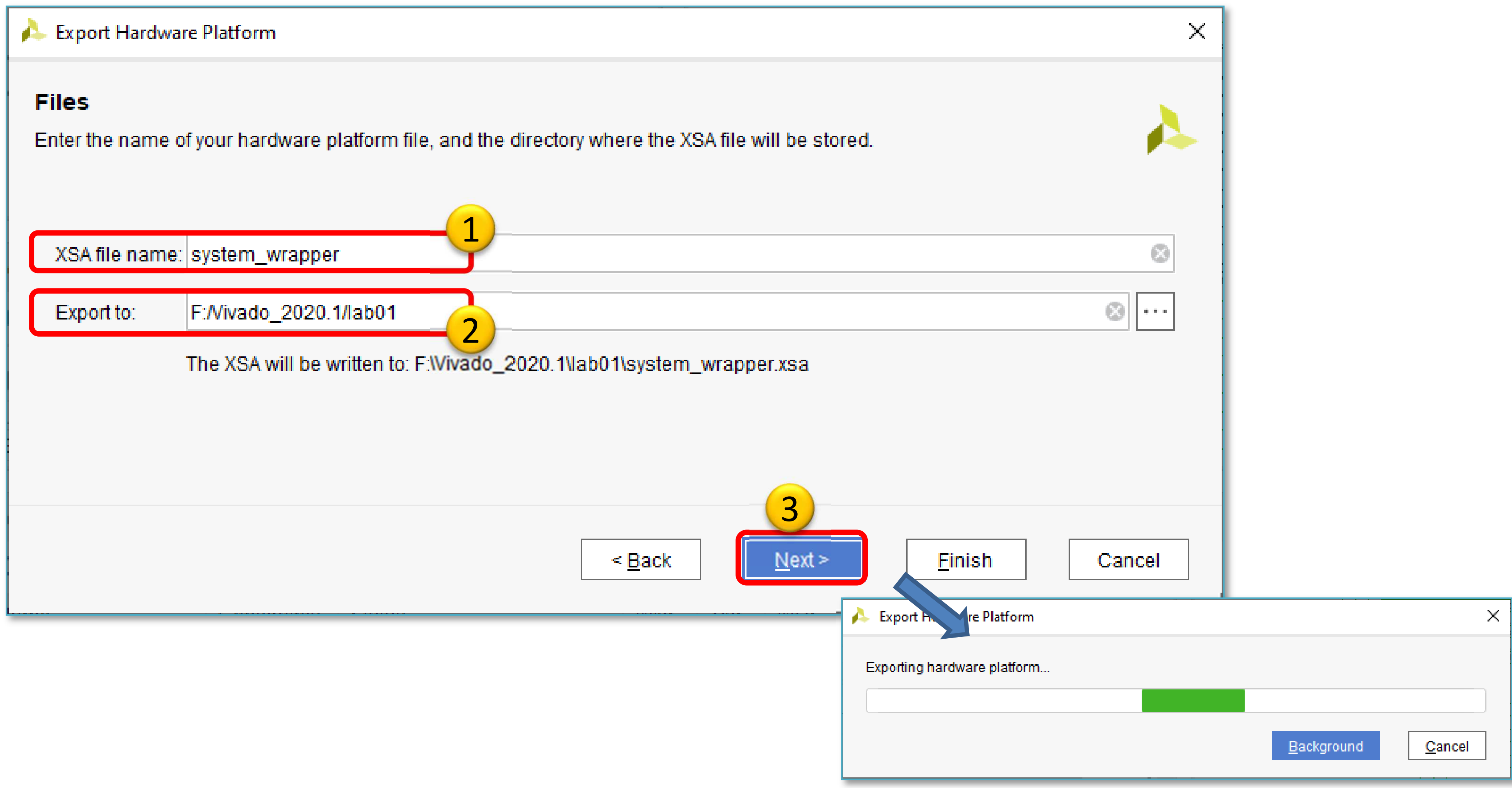
Include bitstream
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.

3

< Back | Next > | Finish | Cancel

Export HW → VITIS (cont.)

Set XSA* file name and export directory path:



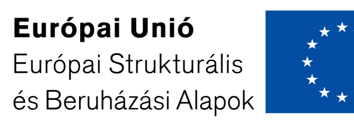
*Xilinx® Support Archive: new hw descriptor format since VITIS 2020.x (see the next slide)

Vivado/VITIS – XSA format

- **.XSA = Xilinx Support Archive = Xilinx proprietary file format, a container. Contains:**
  - One or more .hwh files
    - Vivado® tool version, part, and board tag information
    - IP - instance, name, VLNV (stands for **v**endor, **l**ibrary, **n**ame, and **v**ersion), and parameters
    - Memory Map information of the processors
    - Internal Connectivity information (including interrupts, clocks, etc.) and external ports information
  - BMM/MMI and BIT files
  - User and HLS driver files
  - Other meta-data files

USING XILINX VITIS

Creating a software test application



BEFEKTETÉS A JÖVŐBE

VITIS – General steps of application development

1. Creating a Vivado project, then Export HW → VITIS, √
2. Creating an empty application or an application generated from a C/C ++ template (e.g. Hello World or Memory Test):
   a. Importing `.XSA`
   b. Generating and compiling an application project containing a platform and a domain inside (~BSP: Board Support Package),
   c. Generating a Linker Script (specifying memory sections, `.LD`),
   d. Writing / generating and compiling the SW application
3. Setup a Serial terminal/Console (USB-serial port),
4. Connecting and setup a JTAG-USB programmer,
   – Configuring the FPGA (`.BIT` if PL-side existing)
5. Creating a 'Debug Configuration' for hardware debugging
6. Debug (insert breakpoints, stepping, run, etc.)

Starting VITIS

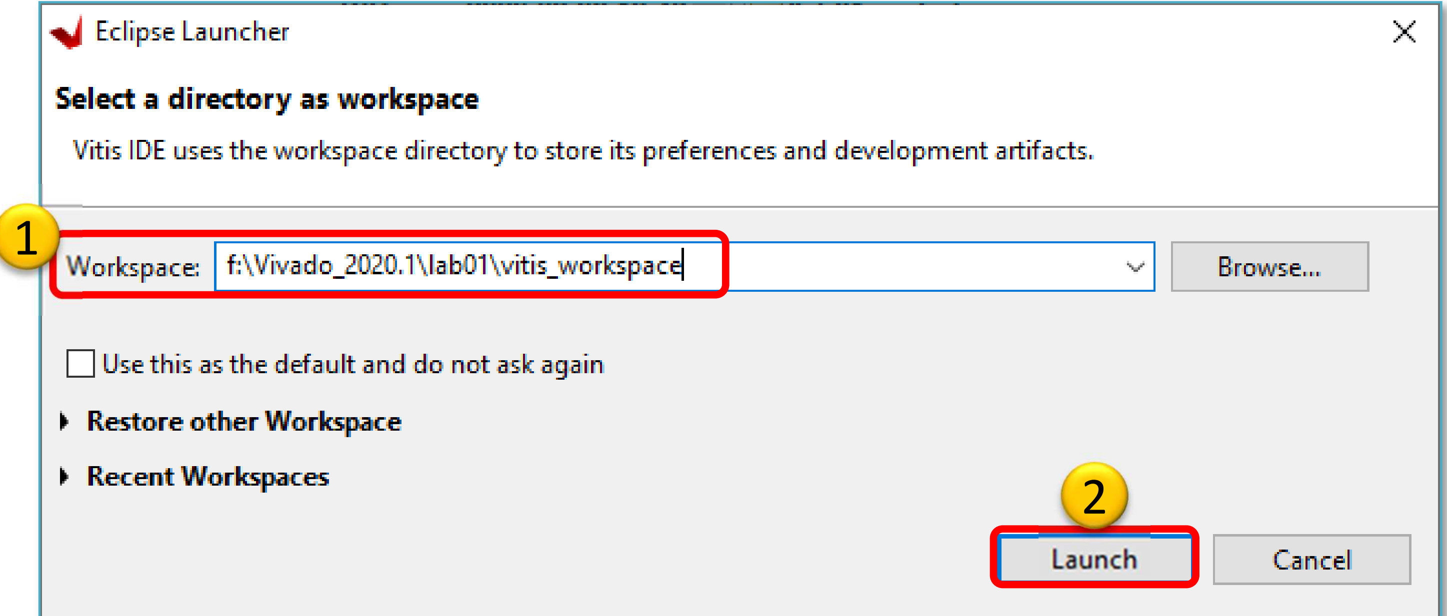
From Vivado: Tools menu → Launch VITIS IDE

OR externally

Start menu → Programs → Xilinx Design Tools → Xilinx VITIS 2020.x

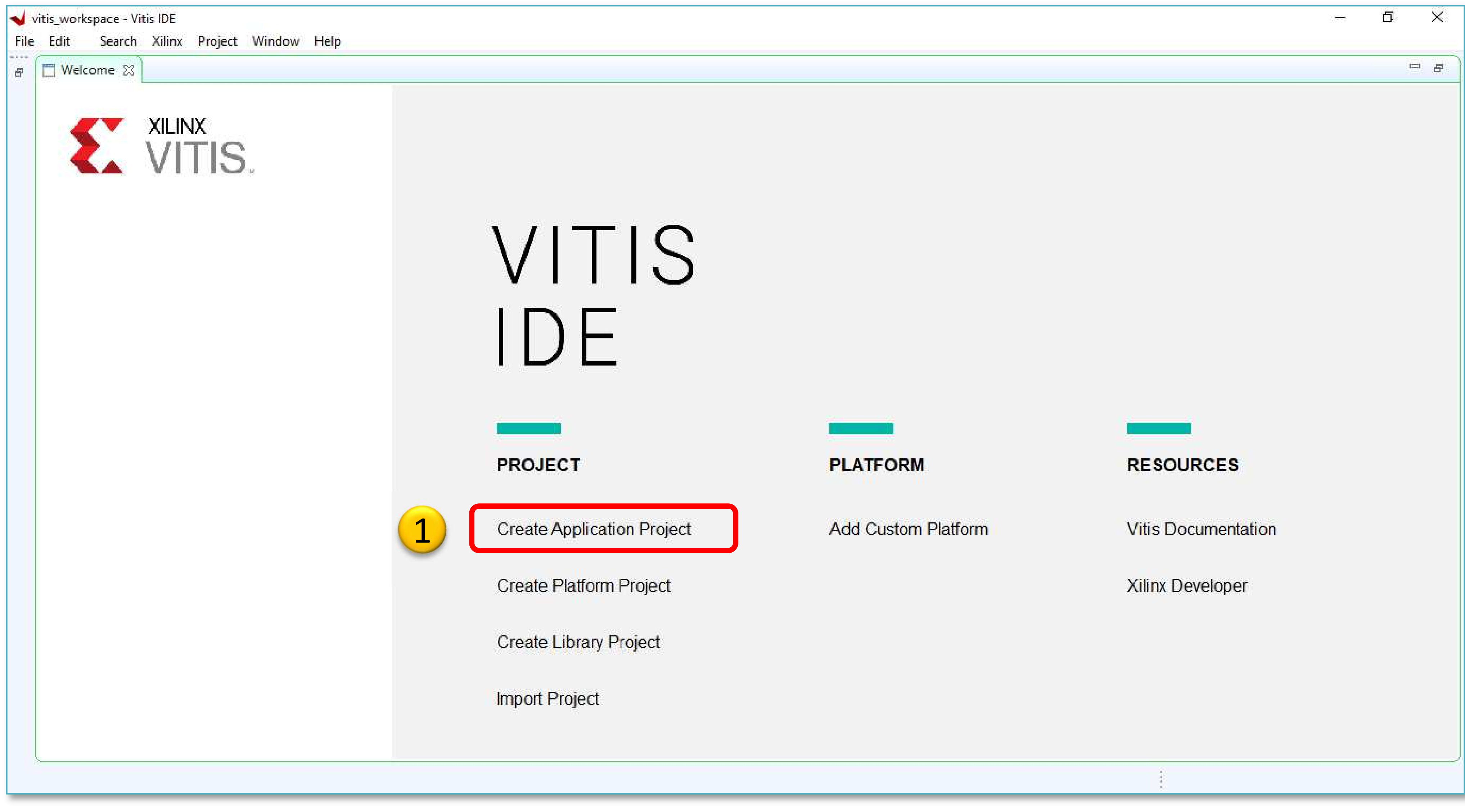
Do Not run Xilinx VITIS HLS 2020.x !

- Set workspace directory properly (`lab01`):
  - Recommended to use `vitis_workspace` as a subdirectory in your lab folder. Then Launch...

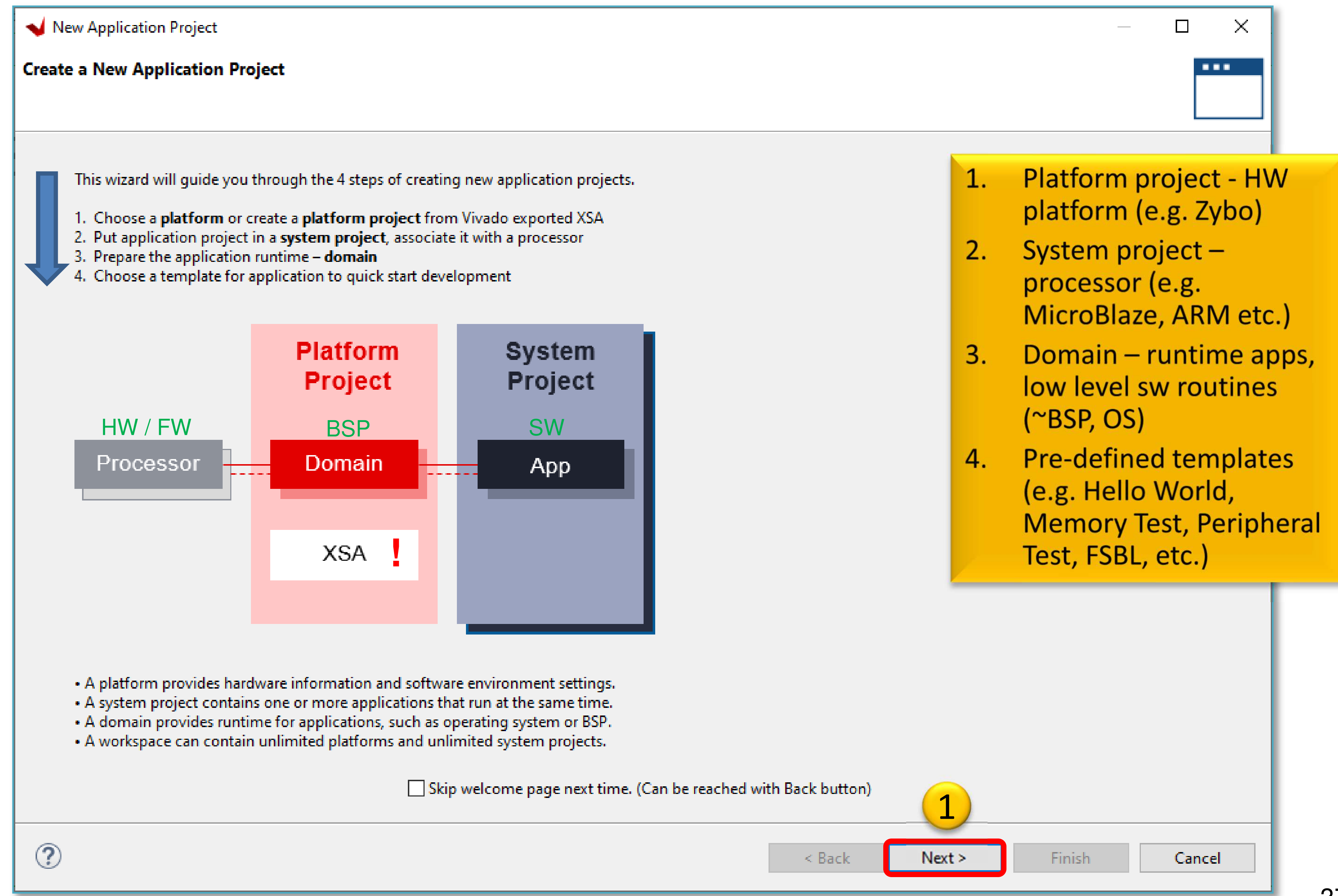


Xilinx VITIS – Create Application

- Create a new application project

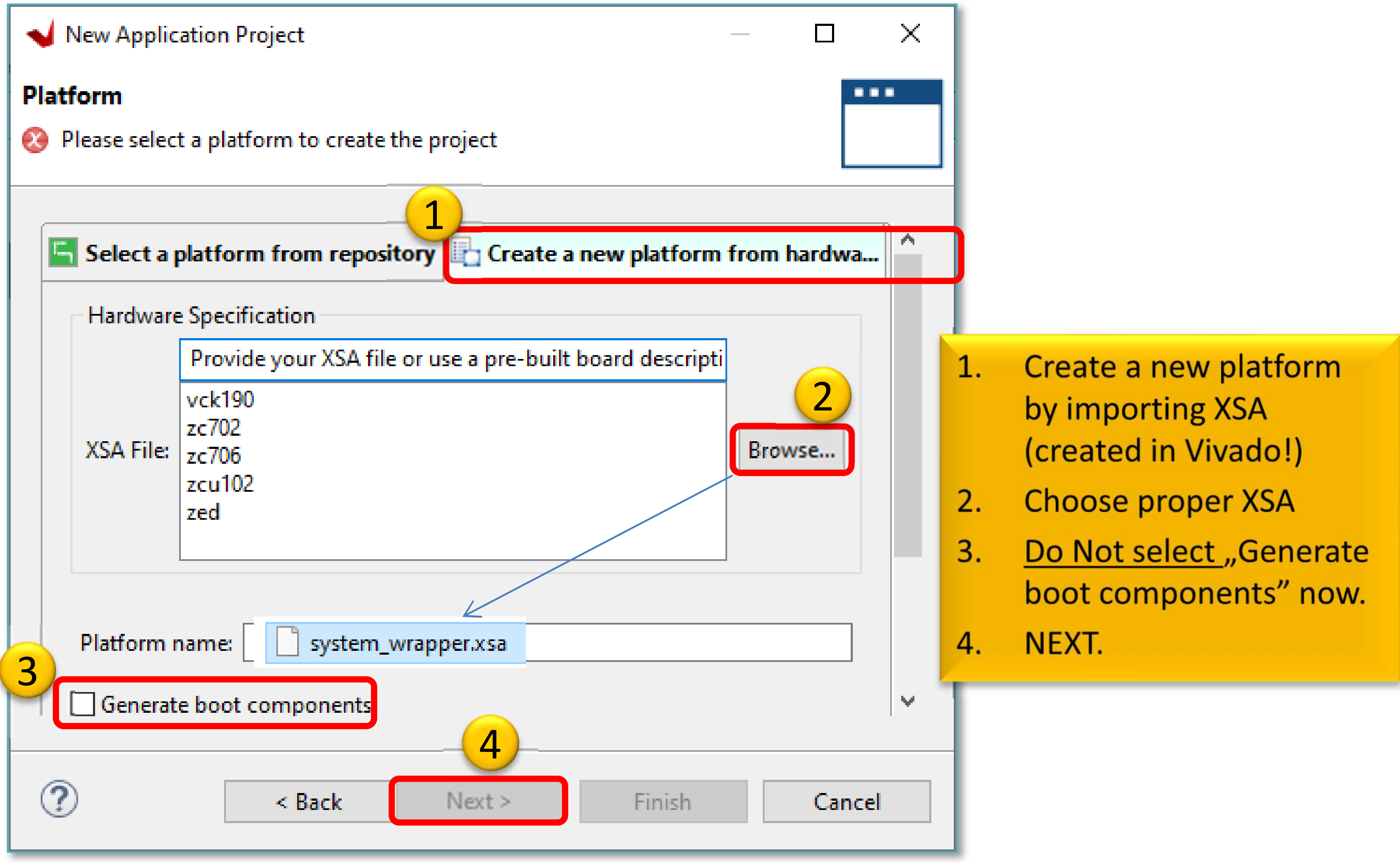


Xilinx VITIS – Create Application

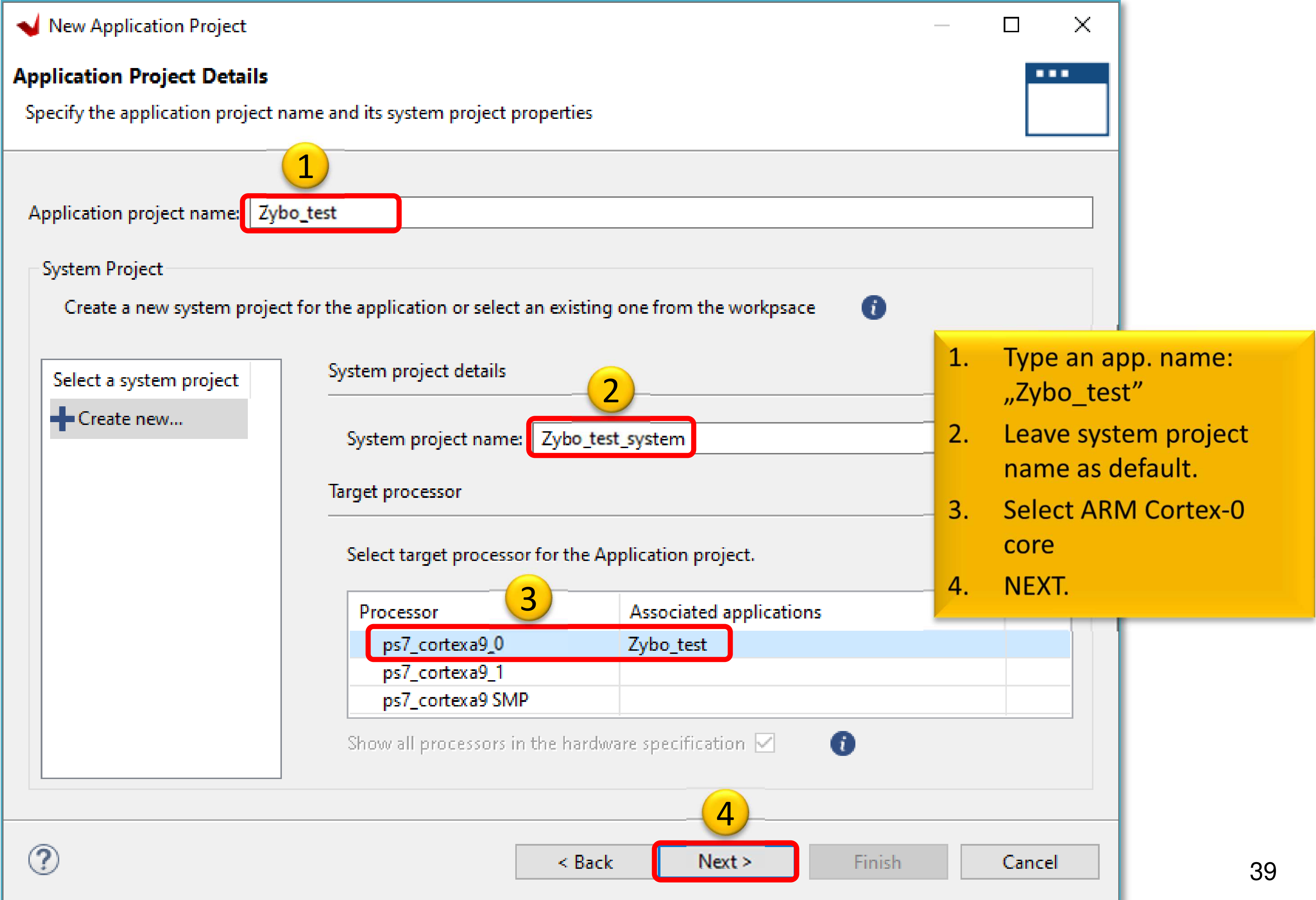


1. Platform project - HW platform (e.g. Zybo)
2. System project – processor (e.g. MicroBlaze, ARM etc.)
3. Domain – runtime apps, low level sw routines (~BSP, OS)
4. Pre-defined templates (e.g. Hello World, Memory Test, Peripheral Test, FSBL, etc.)

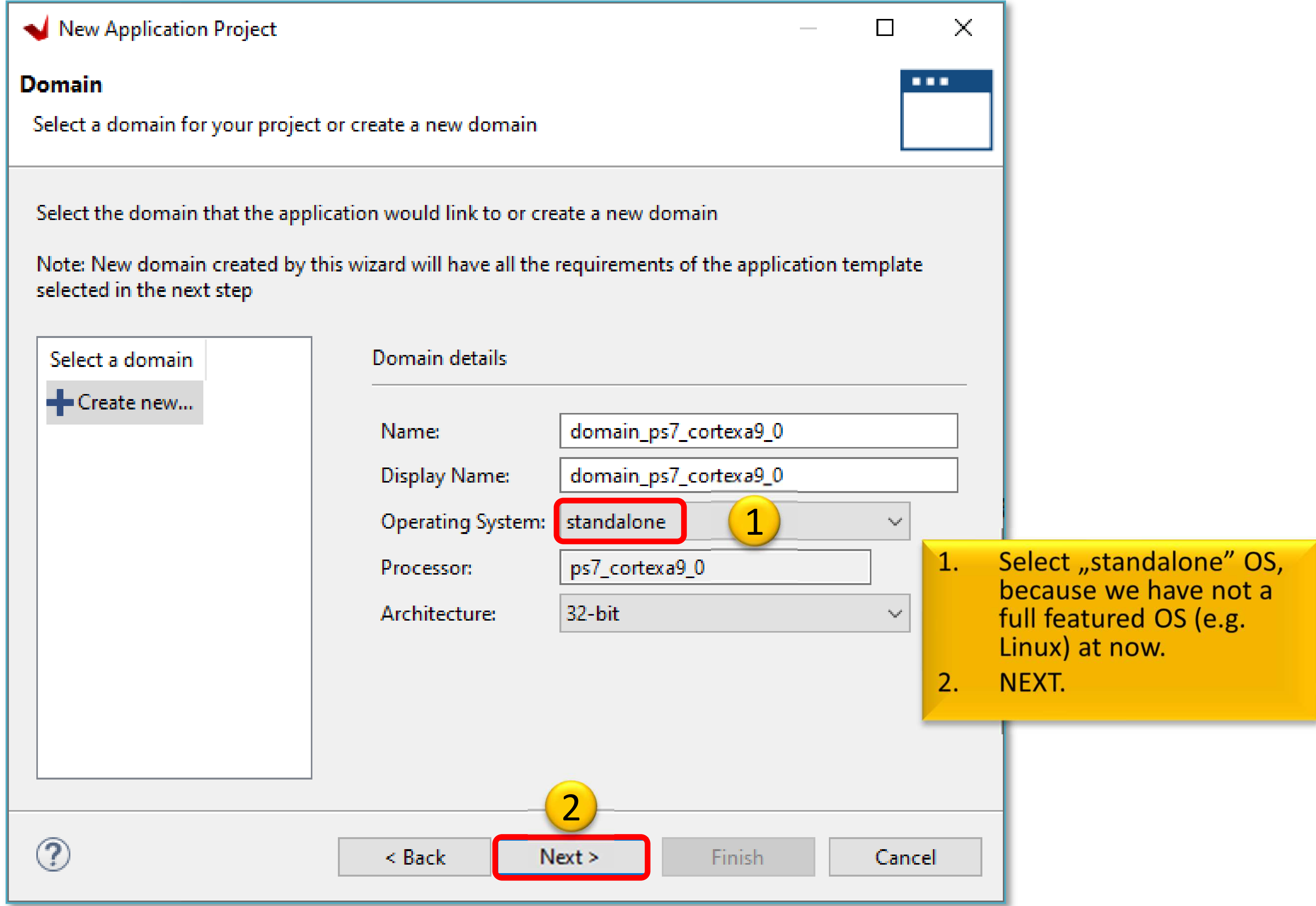
Import XSA



Create Application project

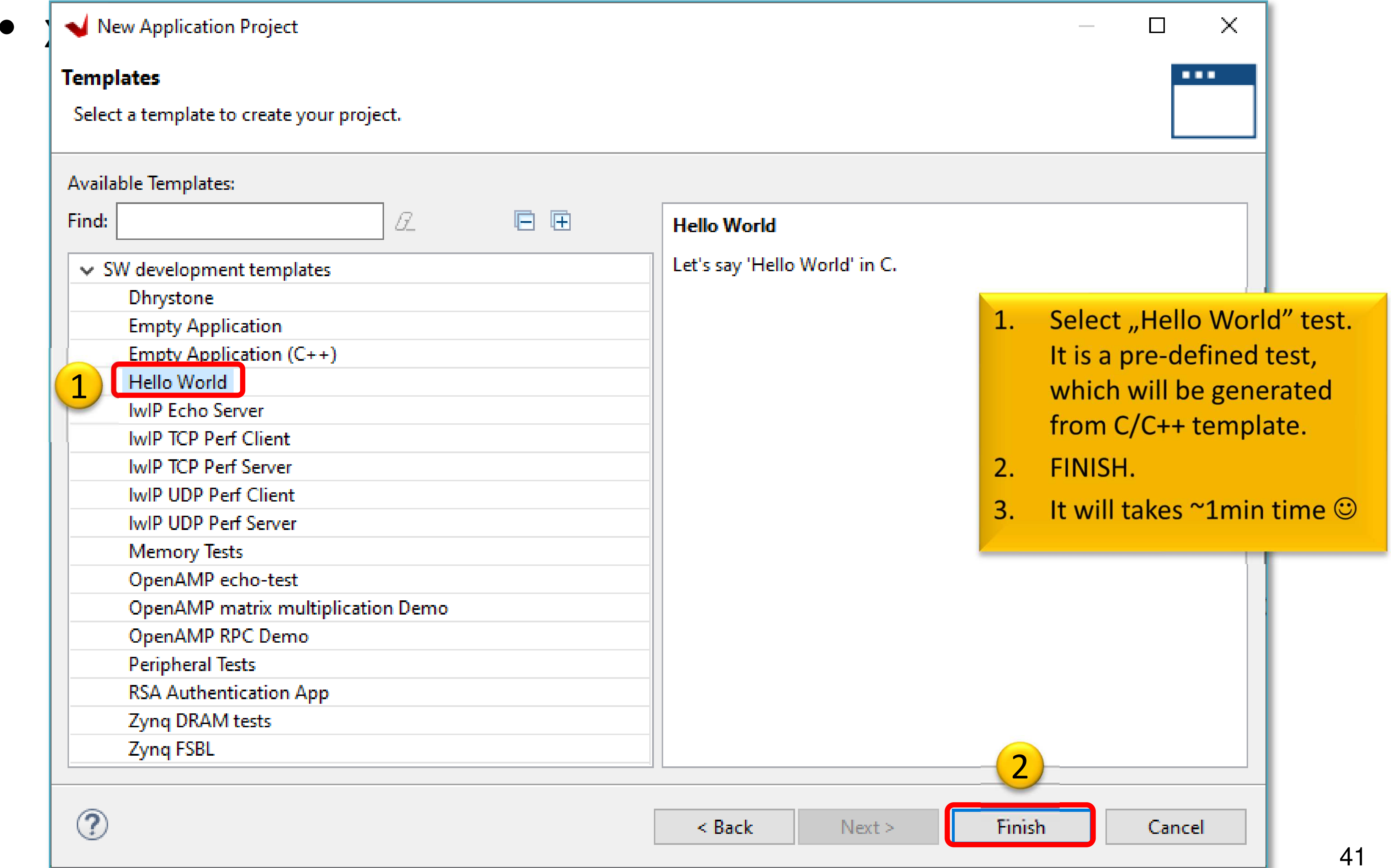


Create Application project – Domain

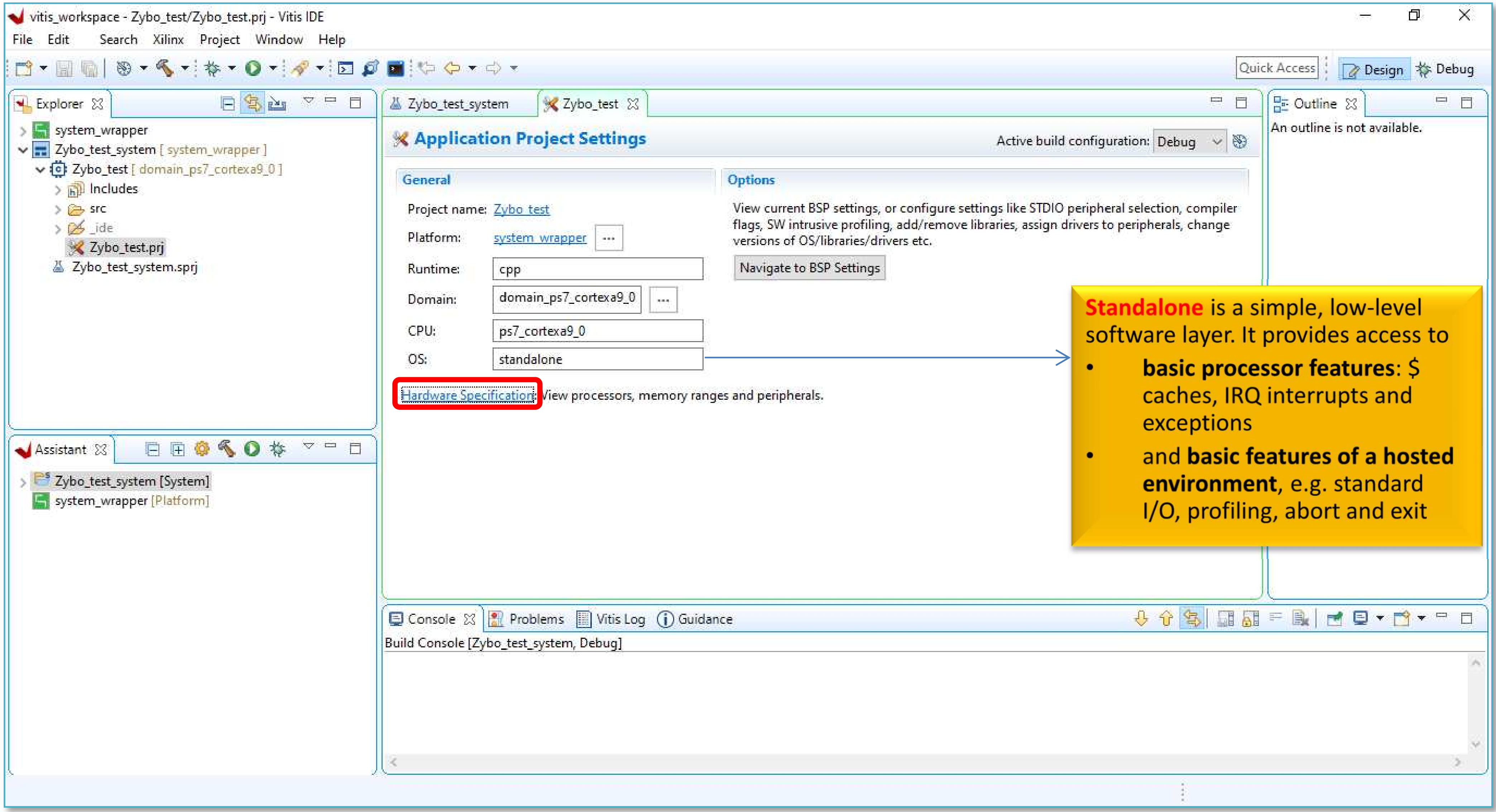


Example I.) Creating HelloWorld application from template

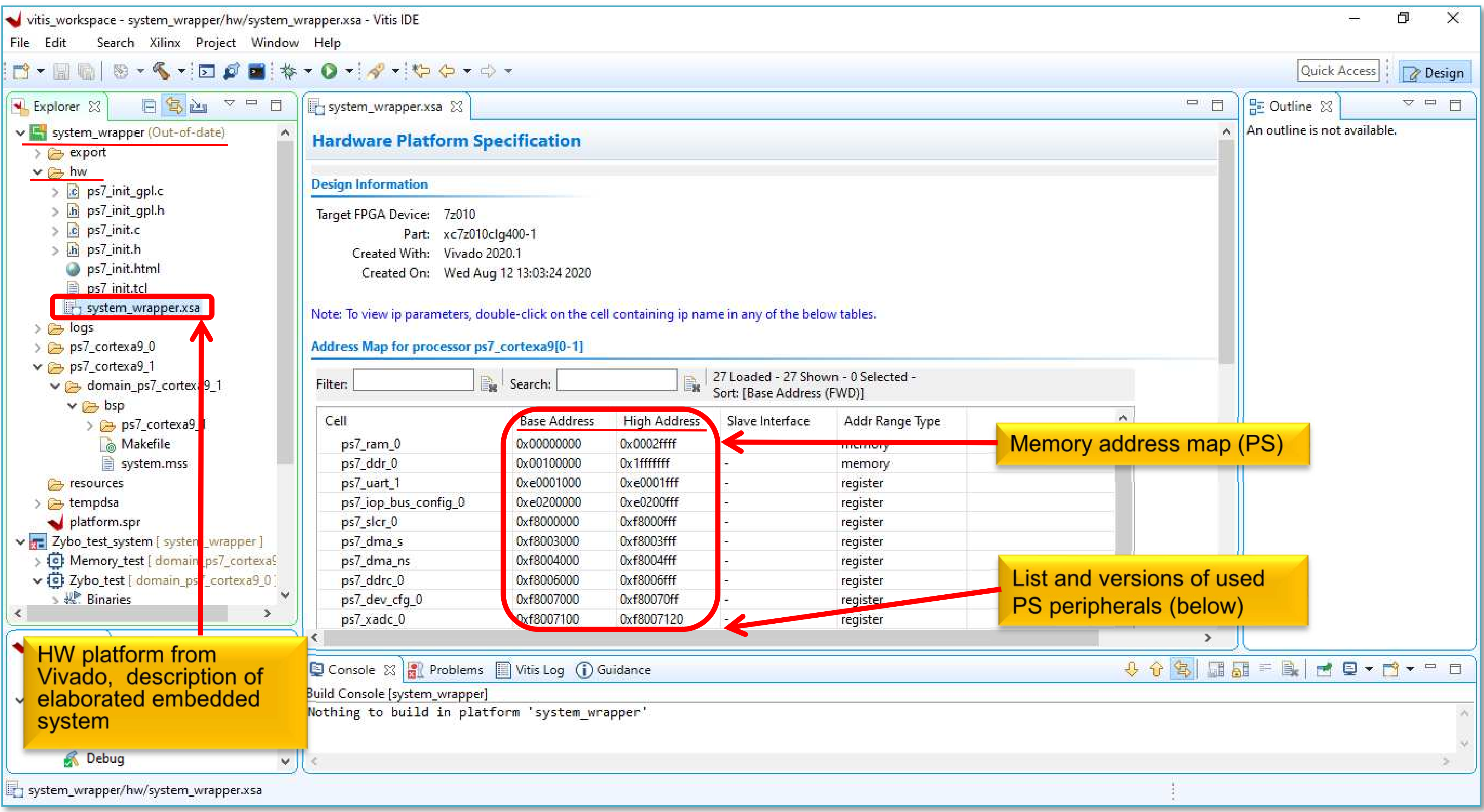
•



VITIS GUI – Main window



VITIS – HW platform

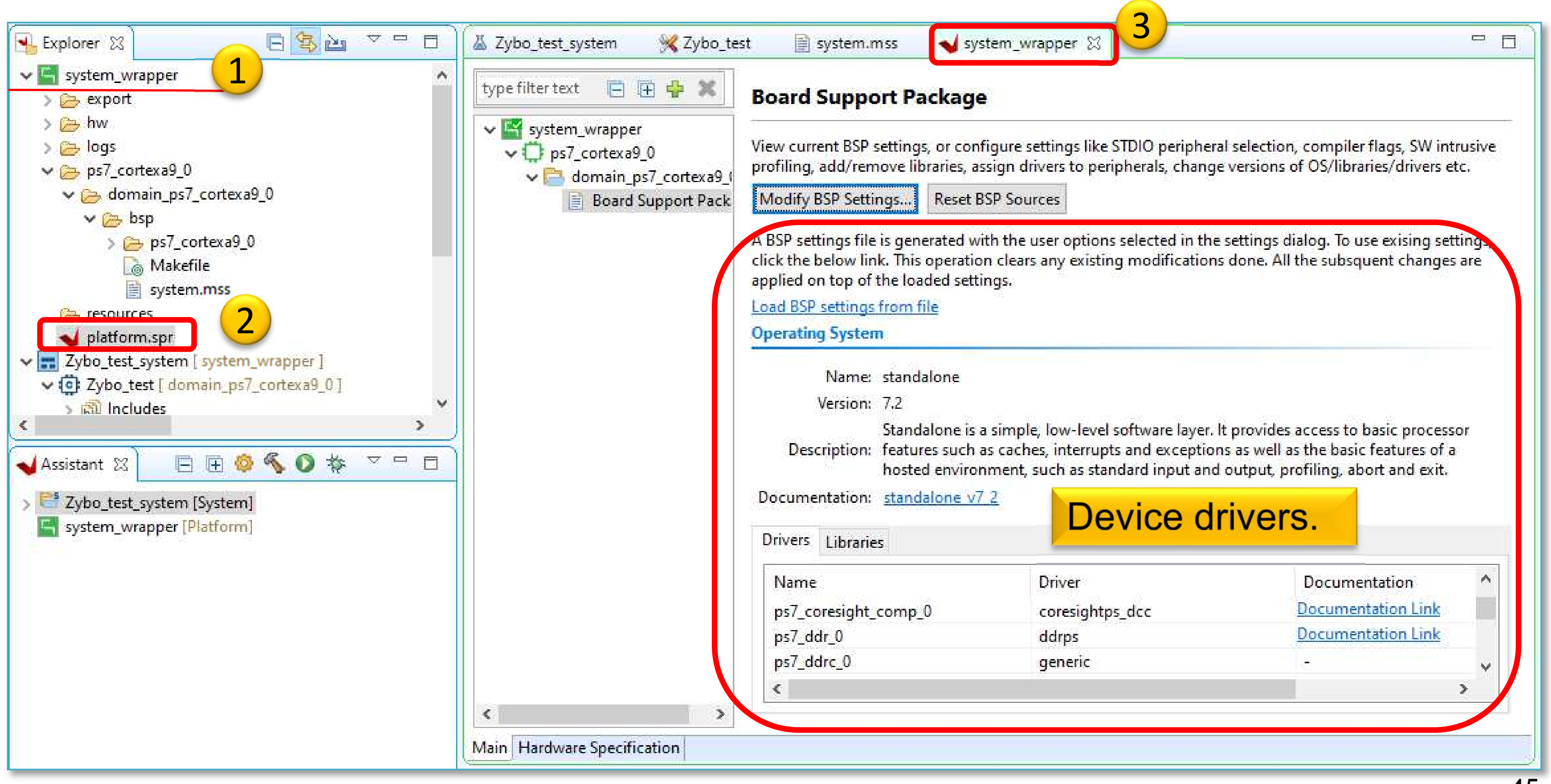


Q & A 1.)

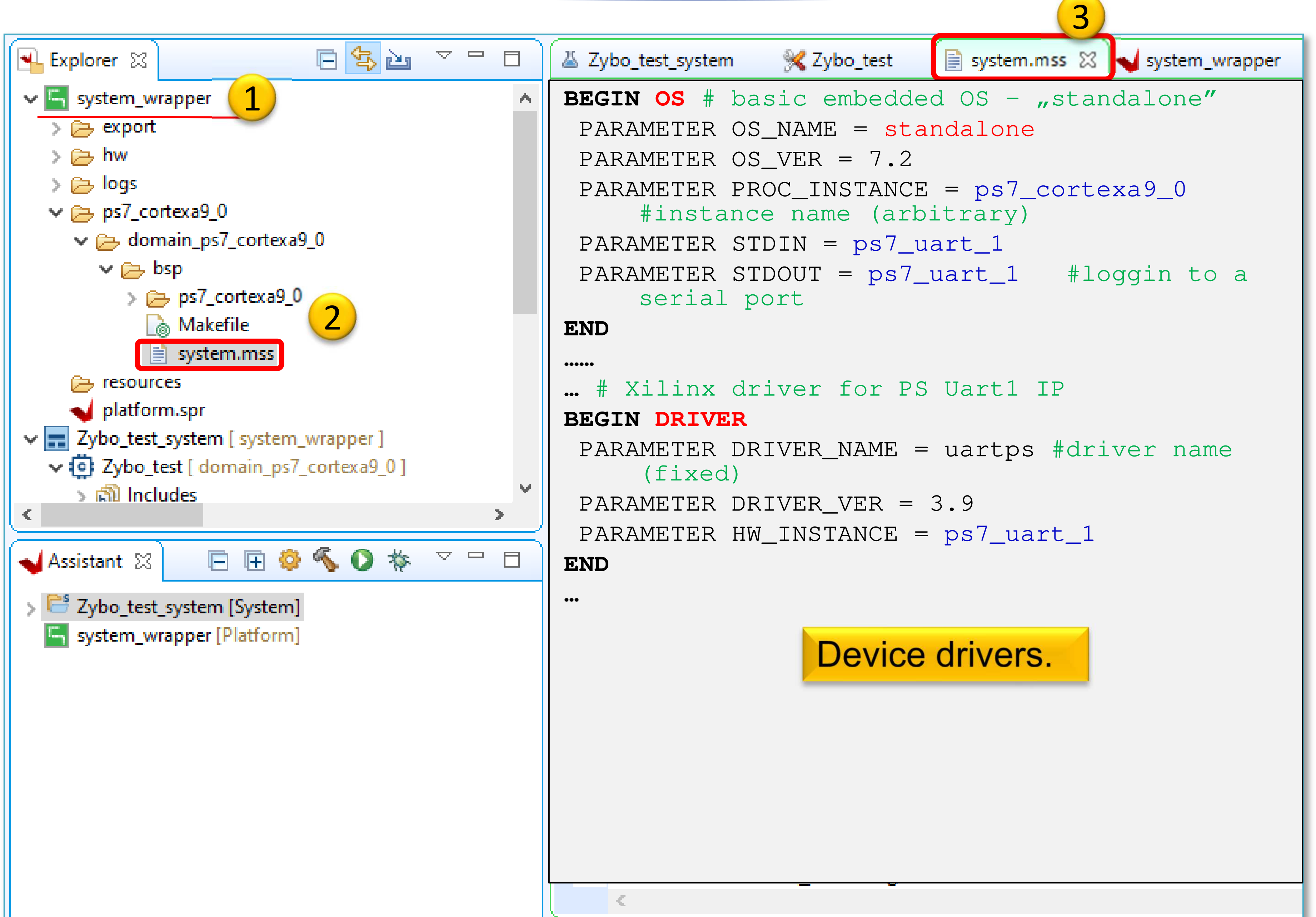
- **How many memory typed address range(s) are in the memory address map?**
  - 3
- **What are their cell names?**
  - ps7_ram_0
  - ps7_ddr_0
  - ps7_ram_1
- **Calculate what size they are?**
  - ps7_ram_0: `0x0002 ffff-0x0000 0000` = 192 KByte
  - ps7_ddr_0 : `0x1fff ffff-0x0010 0000` = 511 MByte
  - ps7_ram_1 : `0xffff 0000-0xffff fdff` = 63 KByte

BSP – system.mss (graphical view)

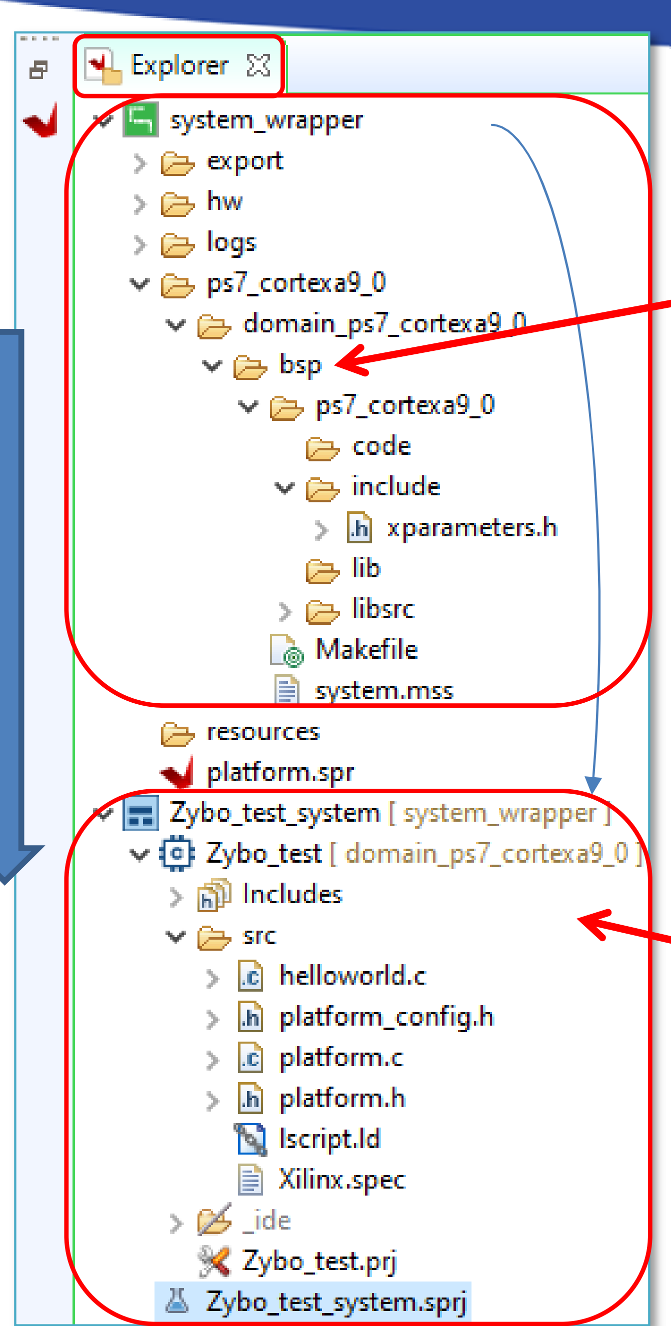
.MSS: **M**icroprocessor **S**oftware **S**pecification (system.mss)



BSP – system.mss (text view)



VITIS – Project Explorer / Hierarchy



`system_wrapper` as **HW** platform was exported from Vivado (`.xsa, .bit, etc.`) contains:

- **BSP**: (OS routines, device drivers, etc.)
  - **MSS: Microprocessor software/driver descriptor (`system.mss`)**
  - **`/includes/xparameters.h !!!` (all related #define and address ranges are defined here)**

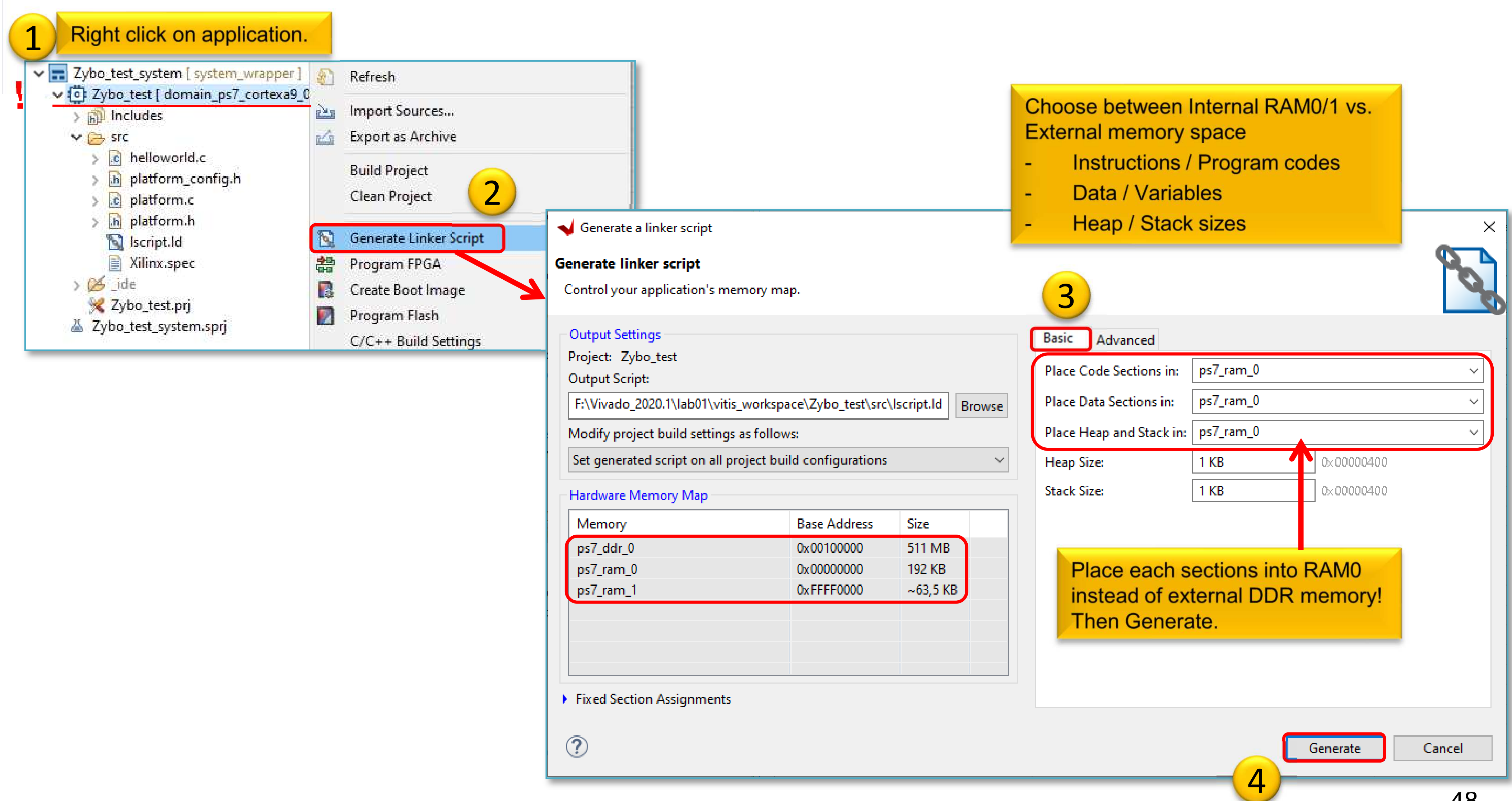
`Zybo_test_system` **as system_project** contains

SW: `zybo_test` (SW application)

- `\Binaries` (**executable load file as `.elf`** object file)
- `\Includes (factory default headers)`
- `\Debug`
- `\Src` = collection of .h, .c, .cpp sources
- **`.ld` = linker script!**
- **`Main()` entry point in the helloworld.c file.**

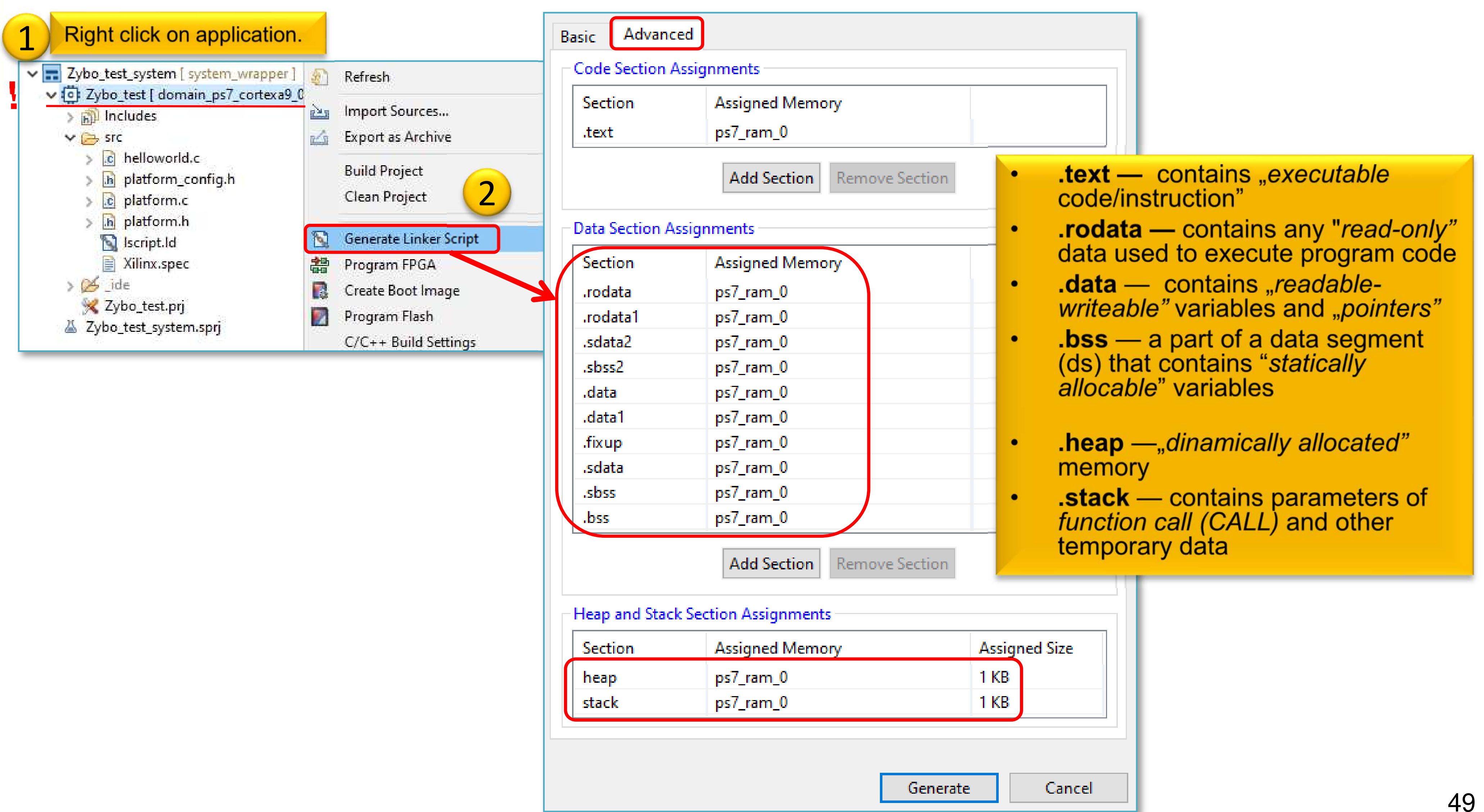
Linker Script generation (Basic)

- Xilinx menu → Generate Linker Script (`lscript.ld`)



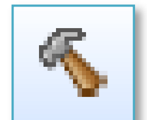
Linker Script generation (Advanced)

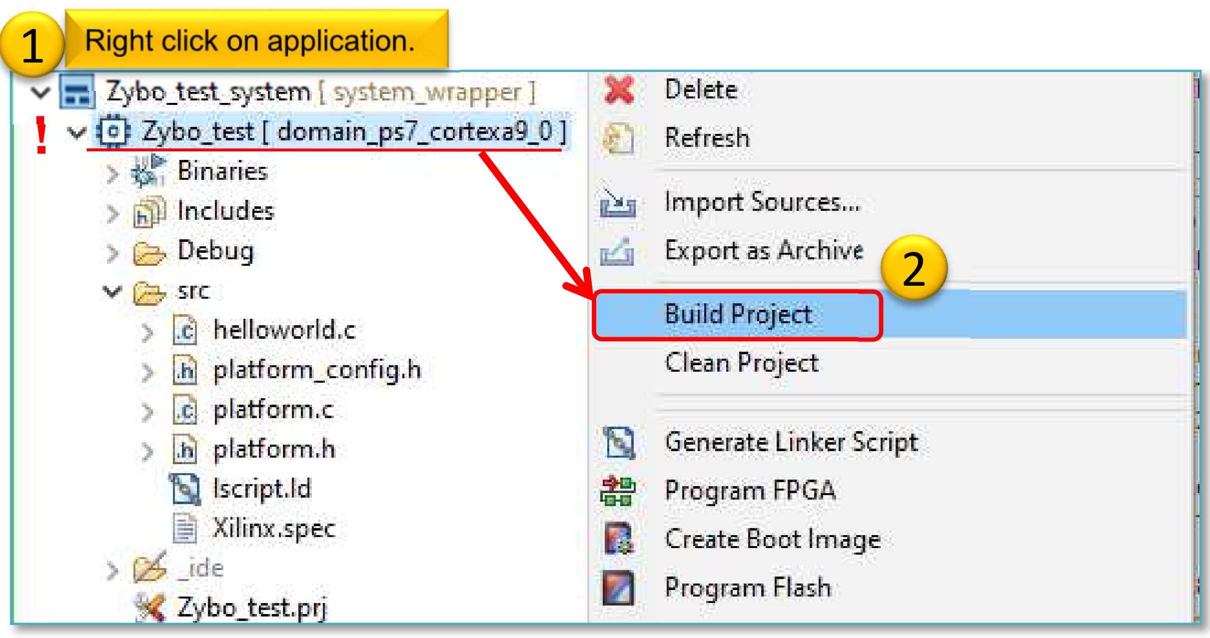
- Xilinx menu → Generate Linker Script (`lscript.ld`)



Build project

- 1. Select Application project (e.g. `Zybo_test`)
- 2. Project menu → Build Project... in two steps:
  - Build BSP (`system_wrapper`)
  - Build software application





Build project - Result

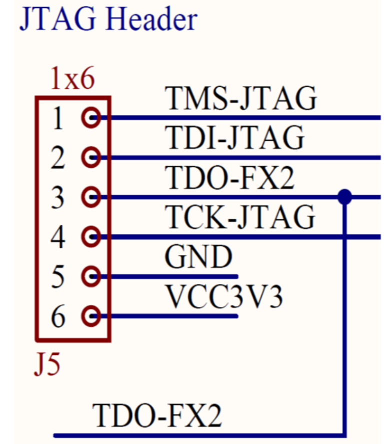
```
'Building target: Zybo_test.elf'
'Invoking: ARM v7 gcc linker'
arm-none-eabi-gcc -mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard -Wl,-build-id=none -specs=Xilinx.spec -Wl,-T -Wl,../src/lscript.ld -LF:/Vivado_2020.1/lab01/vitis_workspace/system_wrapper/export/system_wrapper/sw/system_wrapper/domain_ps7_cortexa9_0/bsplib/lib -o "Zybo_test.elf"  ./src/helloworld.o ./src/platform.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: Zybo_test.elf'
' '
'Invoking: ARM v7 Print Size'
arm-none-eabi-size Zybo_test.elf  |tee "Zybo_test.elf.size"
   text    data     bss     dec     hex filename
  19044    1144    8232   28420    6f04 Zybo_test.elf
'Finished building: Zybo_test.elf.size'
```

Decimal size: 28 420 byte ~28 KByte . The entire program can be placed both the internal on-chip RAM 0/1 and the external DDR RAM. (On the PL / FPGA-side, however, this amount of BRAM memory should be reserved). Therefore, the executable `.elf` file was also generated successfully.

Embedded system and software test verification

1. **Connect** the USB-serial cable (power+programmer functionality). Please check:
   - JP7 jumper = USB power!
   - JP5 jumper = JTAG mode!
2. Now **Power ON** the ZyBo platform

JTAG programming port (optional, but we don't use it!)



ZyBo – Xilinx USB programming cable

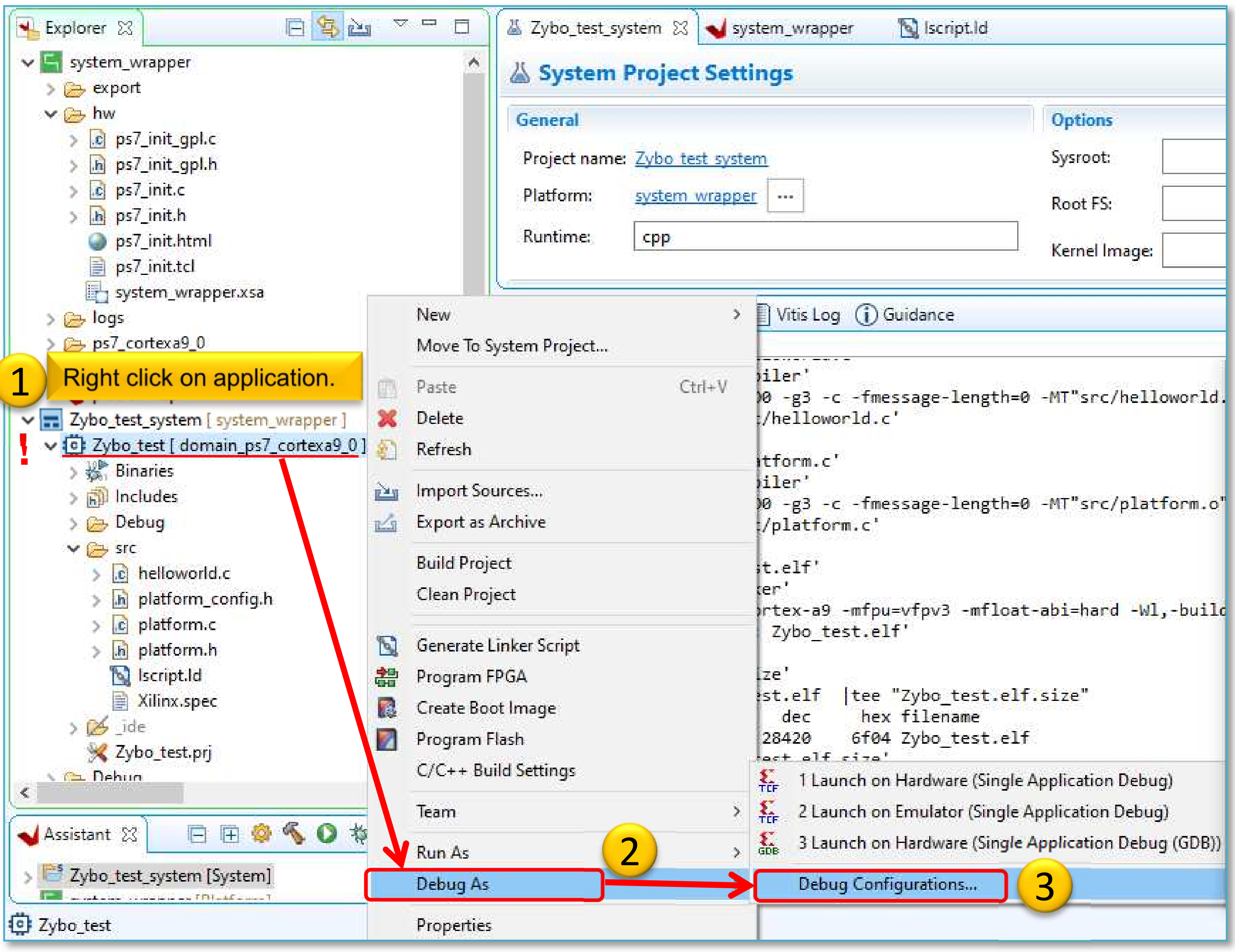
VCC3V3 – red VREF (6)
GND – black GND (5)
TCK-JTAG – yellow TCK (4)
TDO-FX2 – lilac – TDO (3)
TDI-JTAG – white TDI (2)
TMS-JTAG – green TMS (1)

We use the USB-serial connector.

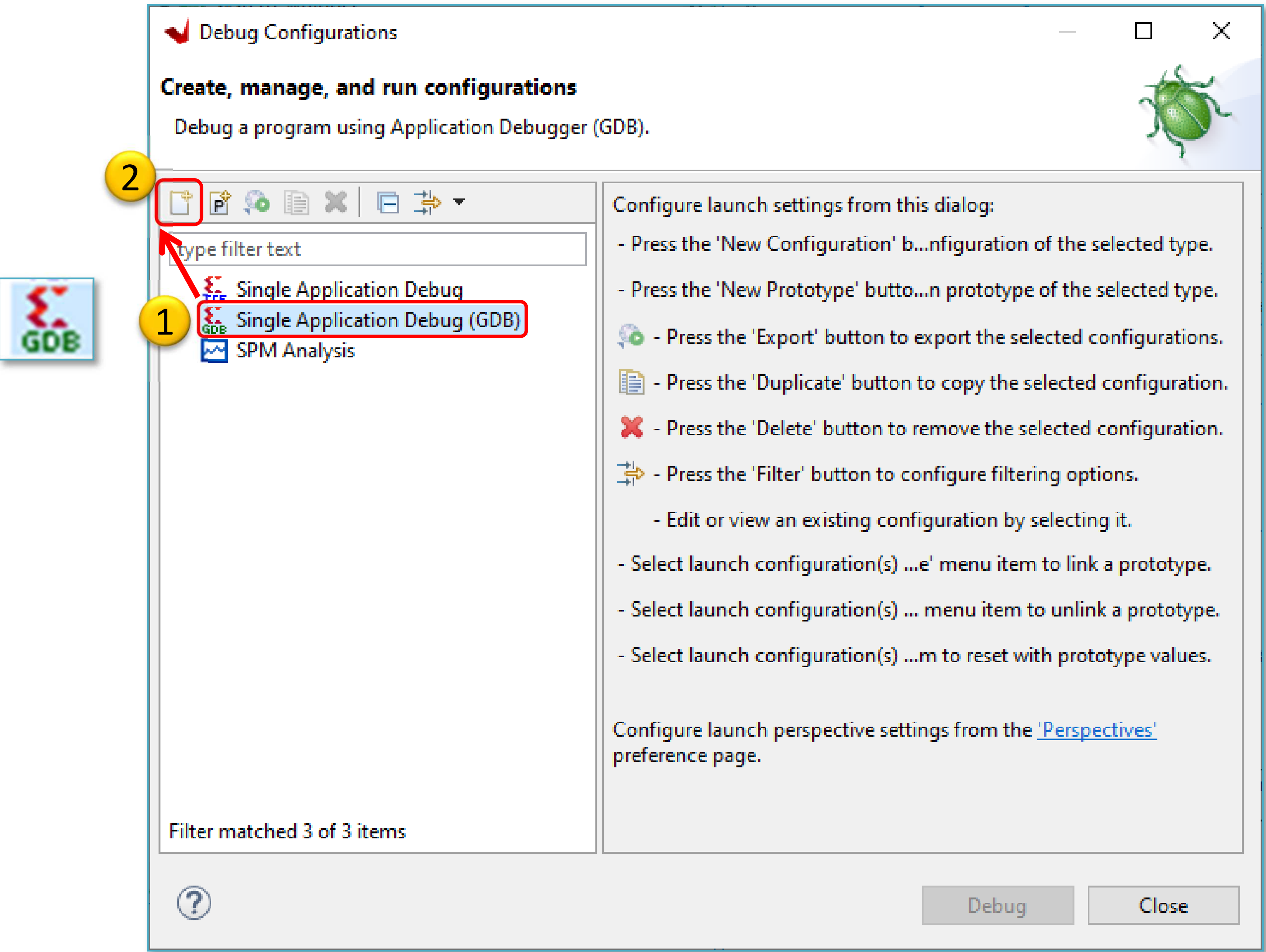
Check the DONE led!

Creating Debug Configuration

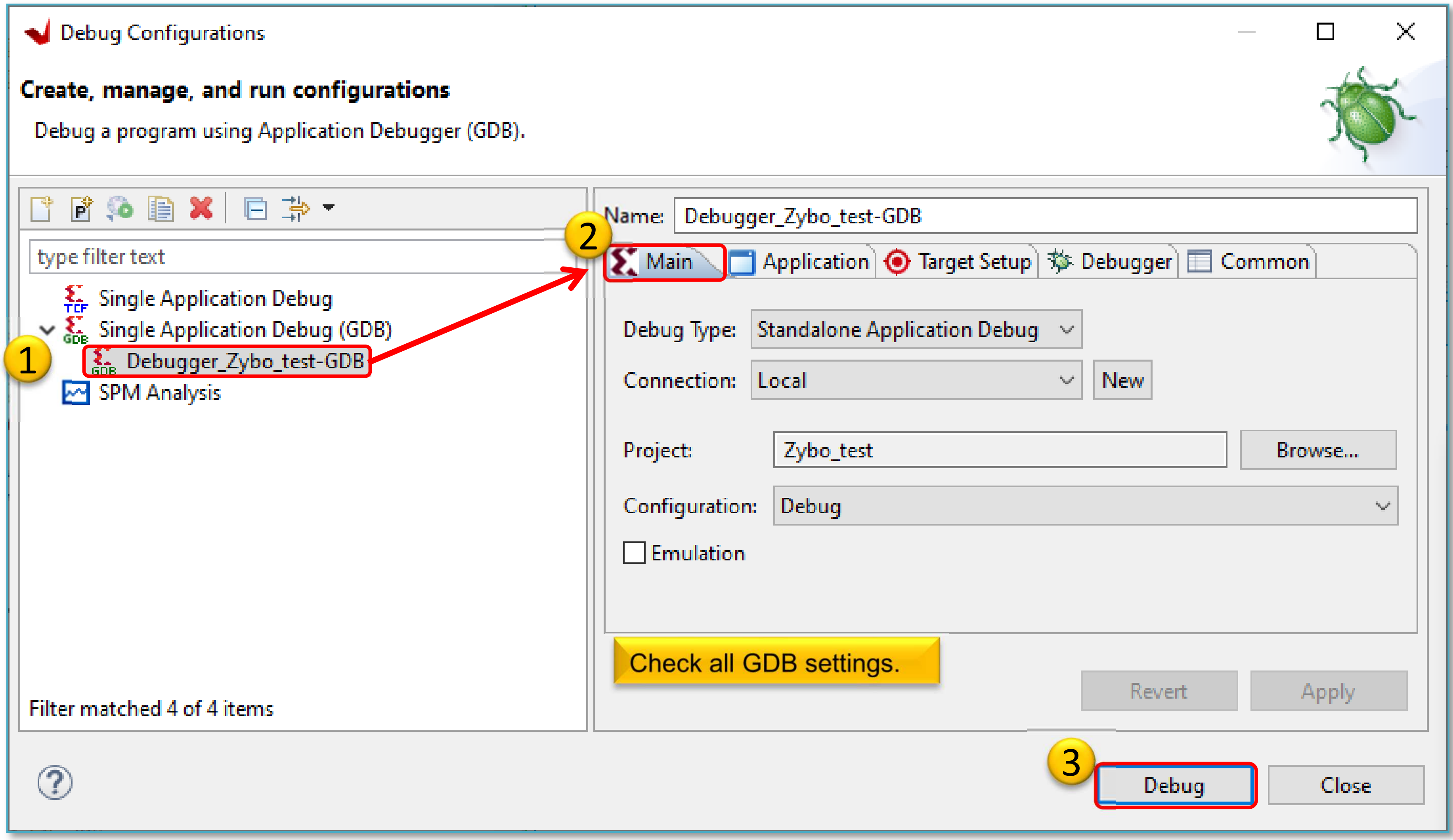
- **Select the application** (`Zybo_test`) in the Project Explorer



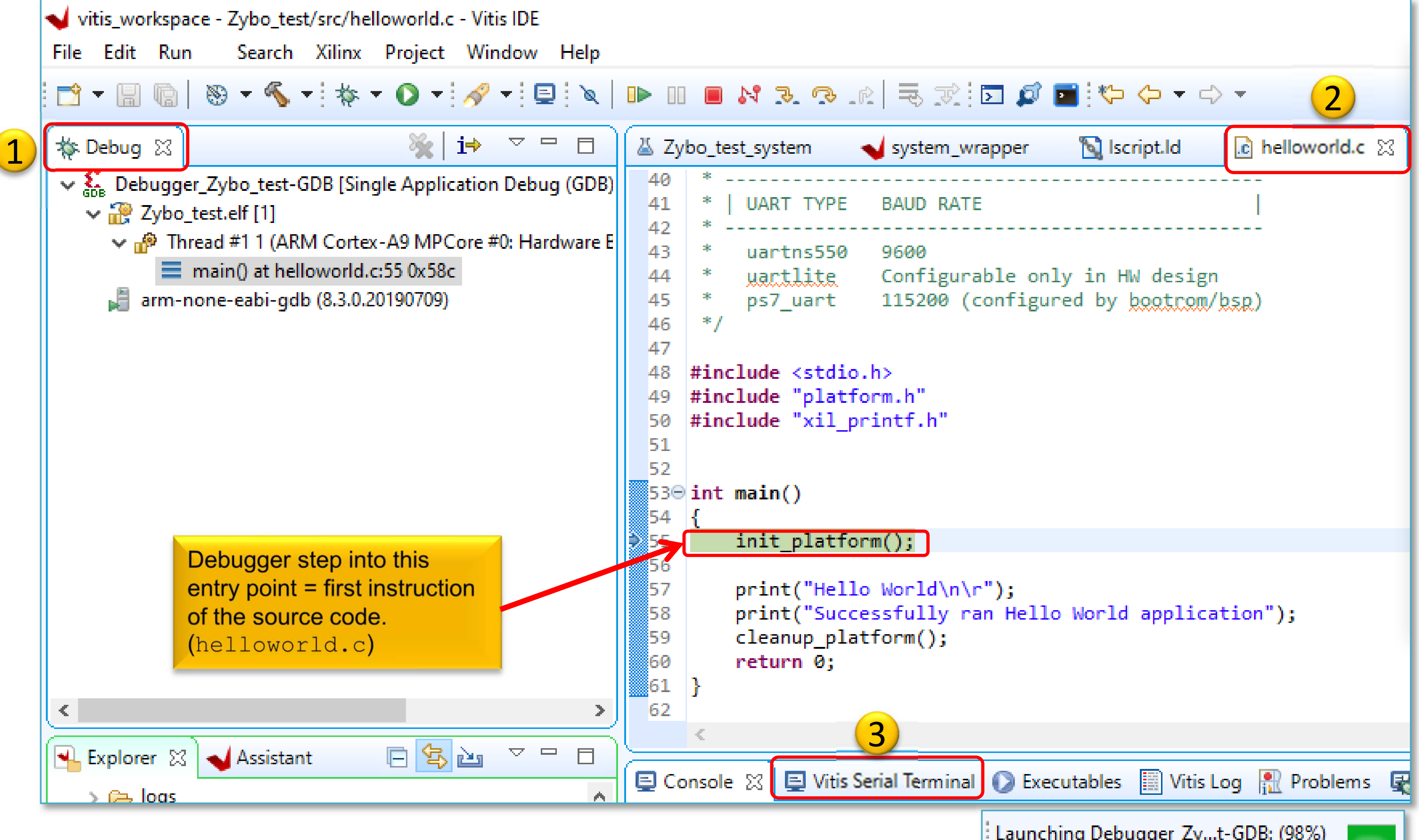
Create a new GDB configuration



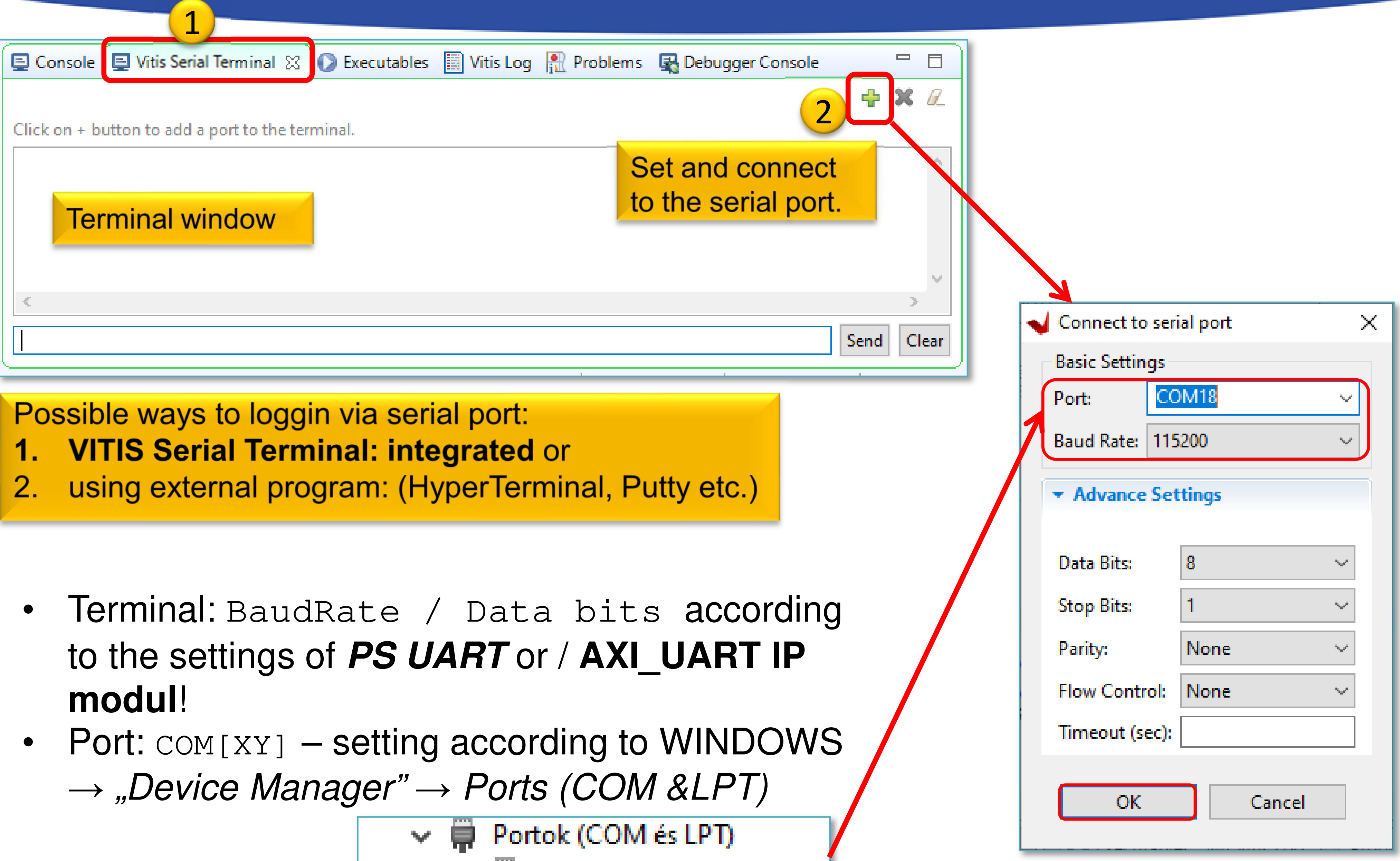
Create a new GDB configuration (cont)



Lunching Debugger

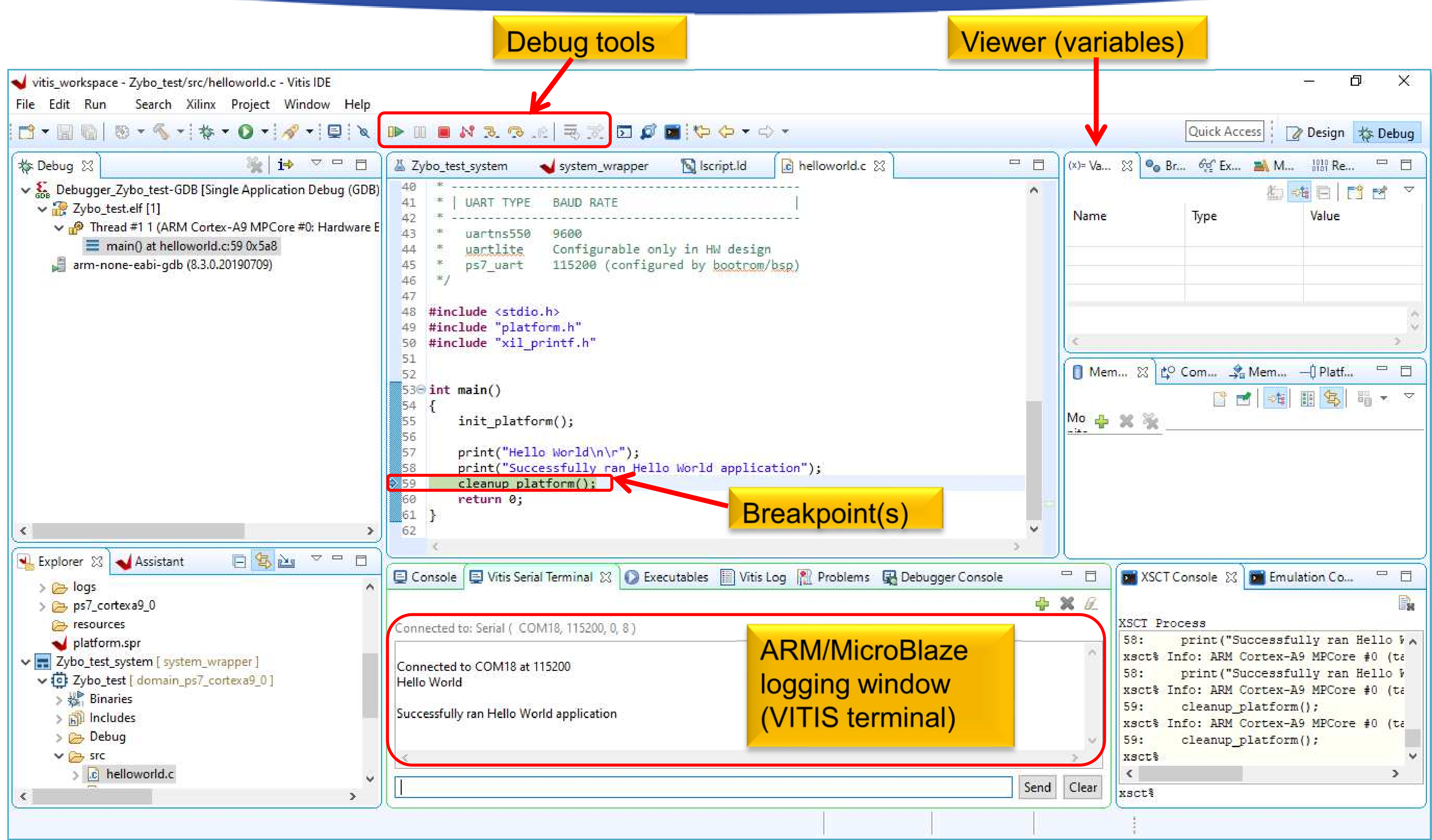


Set Debug-serial port (VITIS terminal)



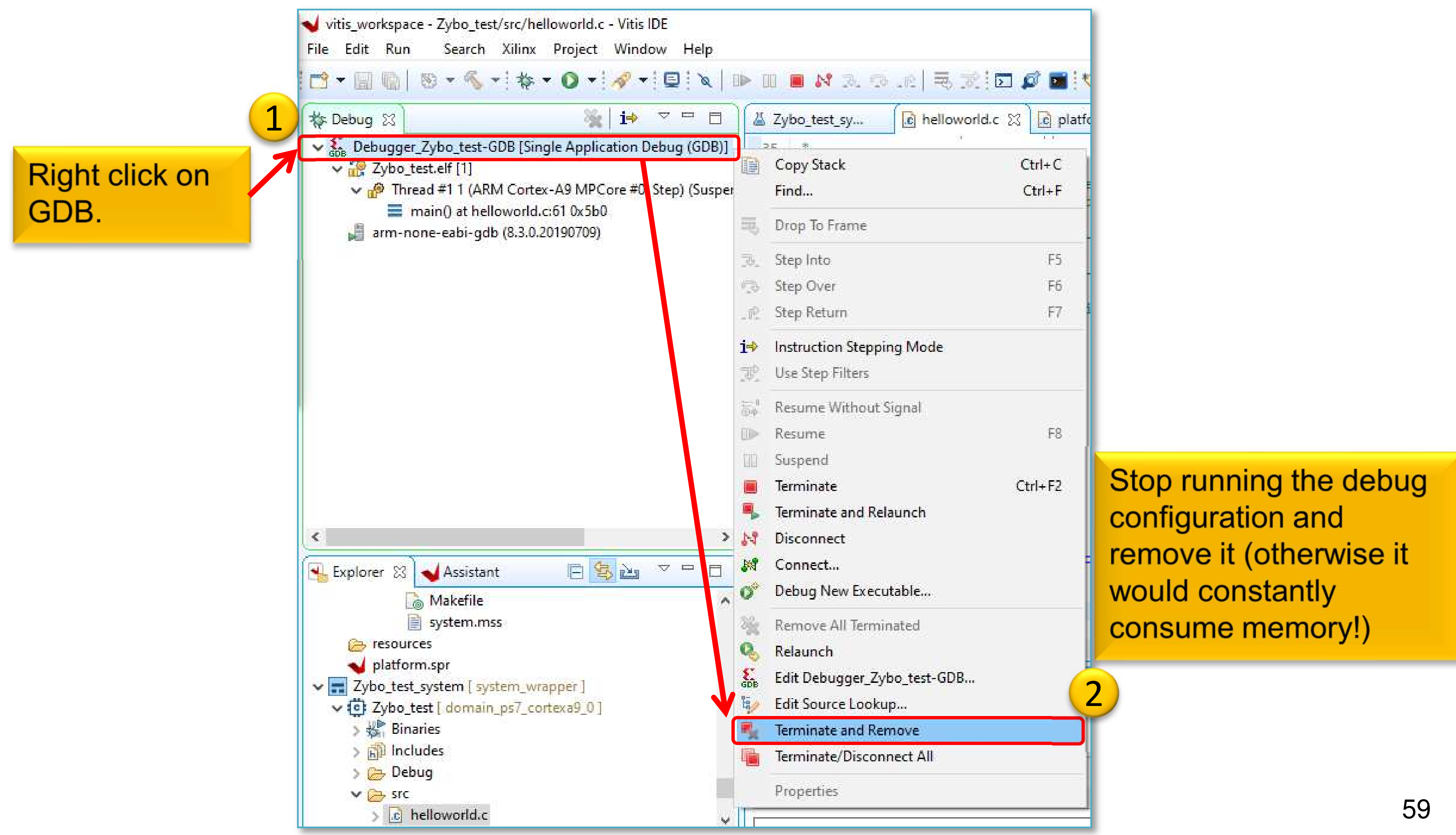
- Terminal: `BaudRate / Data bits` according to the settings of ***PS UART*** or / **AXI_UART IP modul**!
- Port: `COM[XY]` – setting according to WINDOWS → *„Device Manager”* → *Ports (COM &LPT)*

HW debugging - helloworld



Terminate Debug process

- IMPORTANT! At the end of the HW debug, the running debug configuration must be *Terminated and Removed*!



Example I.) Questions

1. Modify and rebulild the Zybo_test helloworld example according to the code snippet below:

```
int main()
{
    init_platform();
    unsigned int i = 0;
    do{
        xil_printf("Hello World (%d)\n\r",i);  //xil_print() OR print()
small   mem consumption - FPGA-optimized function
        //printf("Hello World (%d)\n\r",i);  //printf()  = large mem
consumption
        i++;
    }while(1);
    cleanup_platform();
    return 0;
}
```

   – What will be the size of this rebuilt Zybo_test.elf file?

2. Modify the main function to use `printf() //comment out`
   – What is your experience?

Example I.) Answers

1. Program size

with `xil_printf()` ?

~ 31 Kbyte

```
'Invoking: ARM v7 Print Size'
arm-none-eabi-size Zybo_test.elf  |tee "Zybo_test.elf.size"
   text    data     bss     dec     hex filename
  21512    1144    8232   30888    78a8 Zybo_test.elf
'Finished building: Zybo_test.elf.size'
```

with `printf()` ?

~ 52 Kbyte

```
'Invoking: ARM v7 Print Size'
arm-none-eabi-size Zybo_test.elf  |tee "Zybo_test.elf.size"
   text    data     bss     dec     hex filename
  40864    2548    8304   51716    ca04 Zybo_test.elf
'Finished building: Zybo_test.elf.size'
```

2. Experience ?

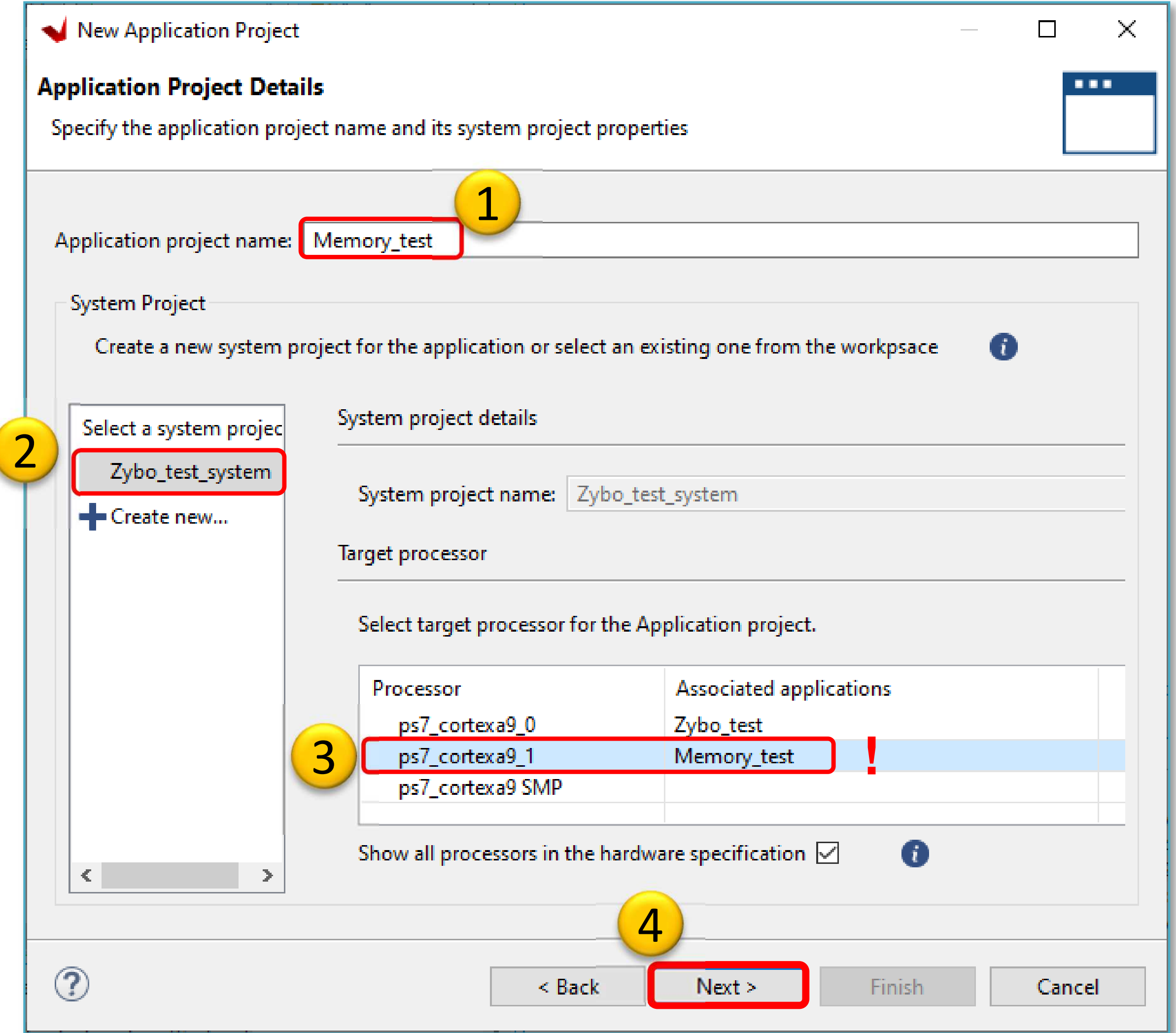
- `xil_printf()` OR `print()`: small memory consumption – FPGA-optimized function
- `printf()` = large memory consumption (non-FPGA optimized, it supports unnecessary formats)

LAB01 – Summary

- **Vivado's Block Designer** makes it easy and simple to create ARM / MicroBlaze processor-based embedded systems
- As part of the process, several implementation files are created, including a **.XSA** file describing the system.
- You can use the different views of Vivado **IP Integrator** to configure the components and various parameters of the assembled embedded system.
- After assembling the system, you can generate the configuration file (bitstream, if there is PL content!).
- Based on the created BSP, we can use the **VITIS** (~SDK) to create a software application with pre-defined templates.
- The correct operation of HW / FW and SW can be verified by downloading the bitstream **.BIT** + executable to **.ELF** FPGA.

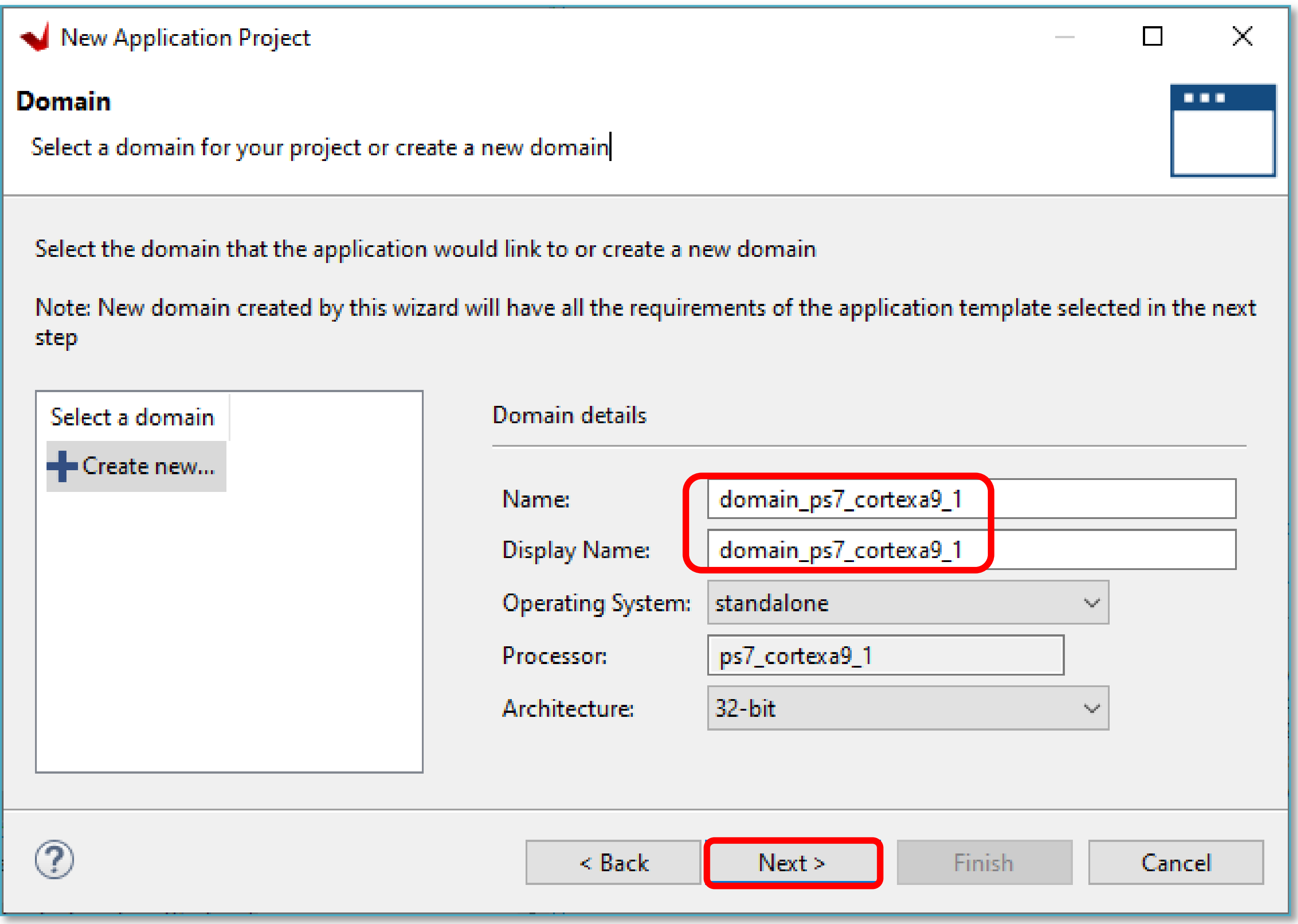
Example II.) Memory Test

- File → New → Application Project … → Next >
- Select `system_wrapper.xsa` as platform then
- Type `Memory_test` as project name. Select `PS7_cortexa9_1` (as Core ”1”)

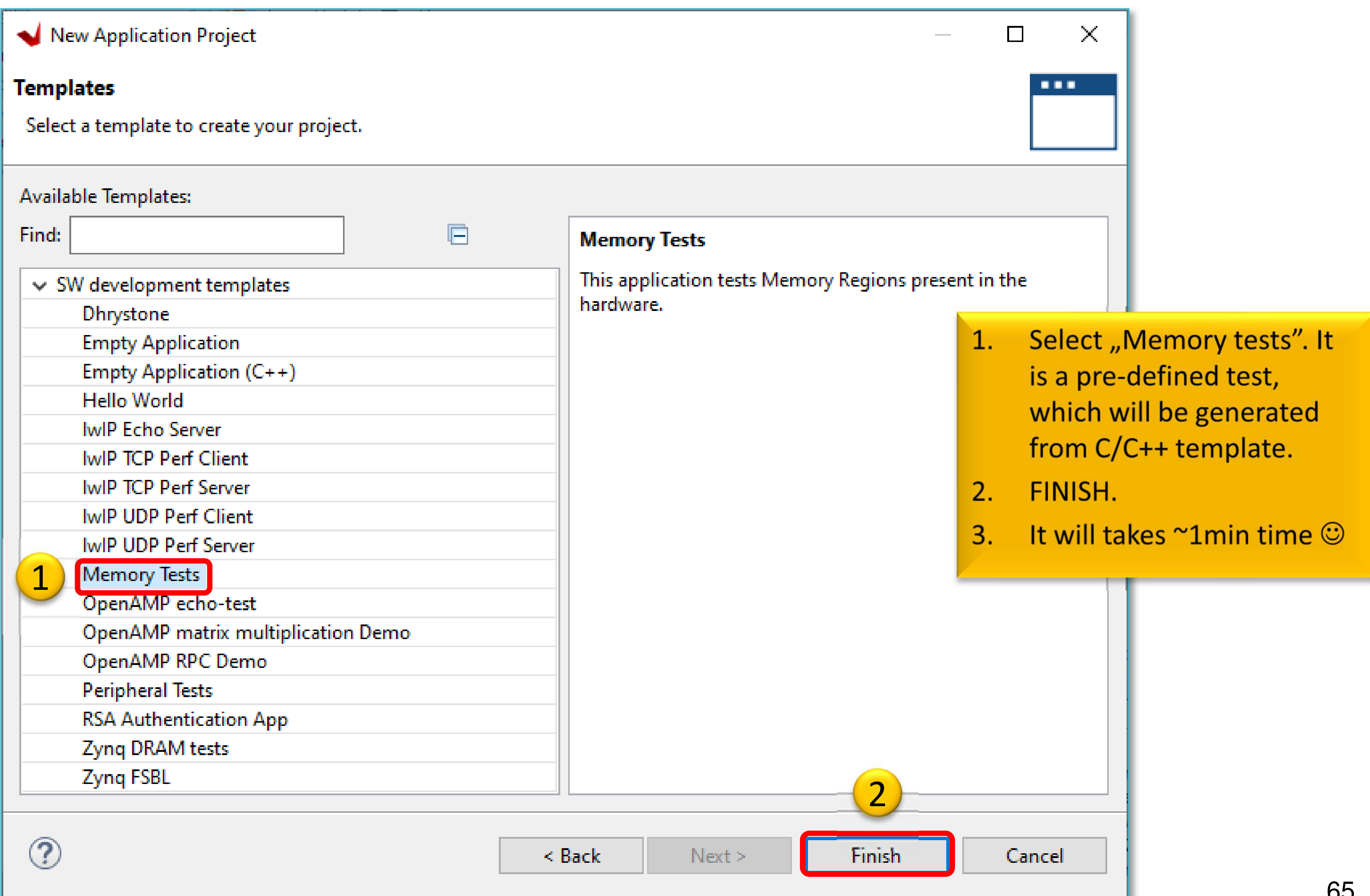


Domain – Cortex A9-1

Leave parameters as default.



MemoryTest application from template



SW application: MemoryTest

- **xparameters.h** (defines, addresses)
- `init_platform();`
  - `enable_caches();`
- **`test_memory_range`**`(`**`struct`** `memory_range_s *range)`
- `test_memory_range(&memory_ranges[i])`
  - `n_memory_ranges = 2` (defined in `memory_config.h`)
- `cleanup_platform();`
  - `disable_caches();  //ARM enables by default`

Xil_TestMemN() function

```
status = Xil_TestMem32((u32*)range->base, 1024, 0xAAAA5555,
   XIL_TESTMEM_ALLMEMTESTS);
    print("            32-bit test: "); print(status == XST_SUCCESS?
   "PASSED!":"FAILED!"); print("\n\r");

status = Xil_TestMem16((u16*)range->base, 2048, 0xAA55,
   XIL_TESTMEM_ALLMEMTESTS);
    print("            16-bit test: "); print(status == XST_SUCCESS?
   "PASSED!":"FAILED!"); print("\n\r");

status = Xil_TestMem8((u8*)range->base, 4096, 0xA5,
   XIL_TESTMEM_ALLMEMTESTS);
    print("             8-bit test: "); print(status == XST_SUCCESS?
   "PASSED!":"FAILED!"); print("\n\r");
```

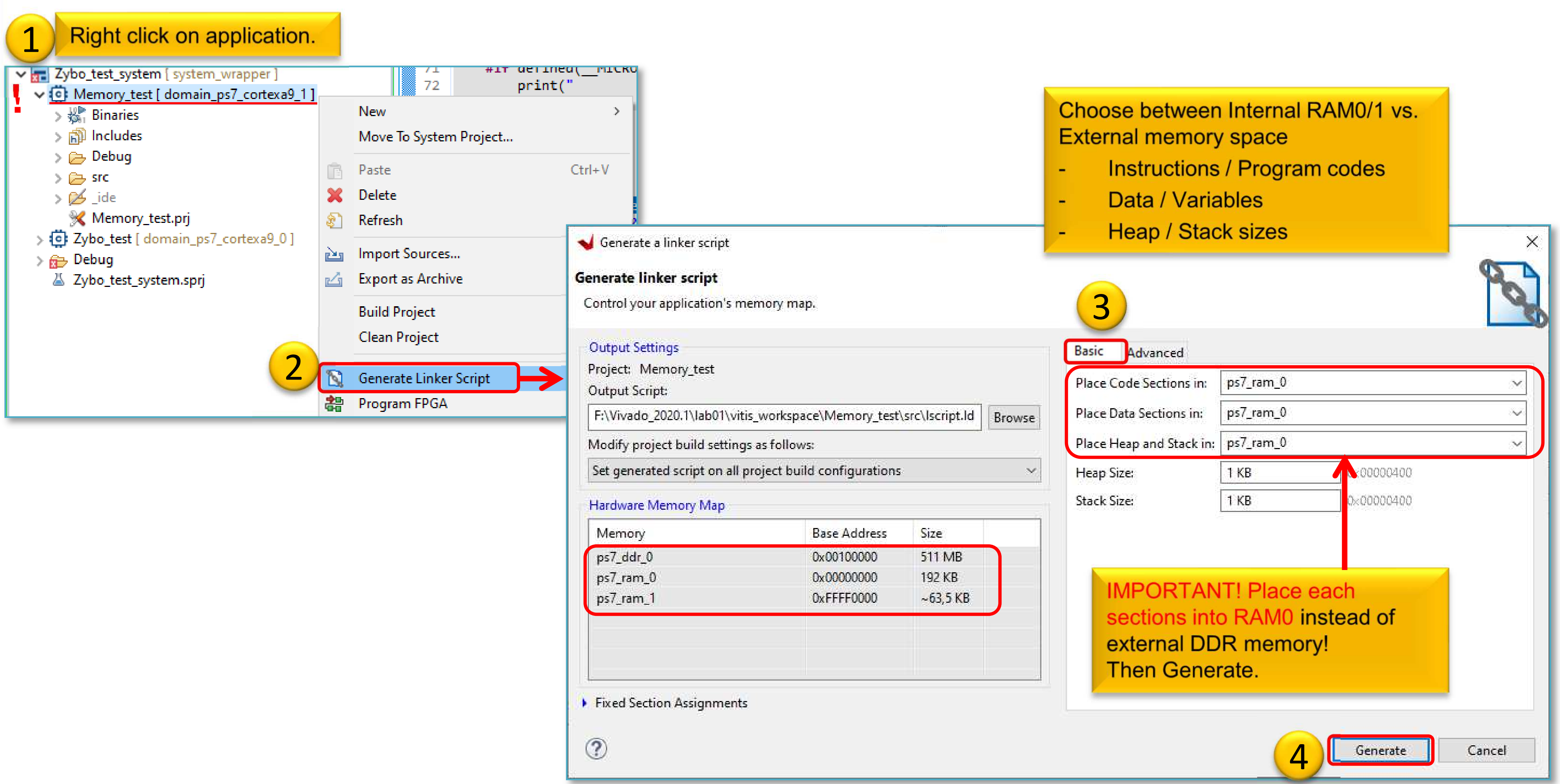
Xil_TestMemN() function declaration can be found here:

`<xilinx_install_dir>\VITIS\2020.x\data\embeddedsw\lib\bsp\standalone_v7_2\src\common\xil_testmem.c`

s32 Xil_TestMem32(u32 *Addr, u32 Words, u32 Pattern, u8 Subtest);

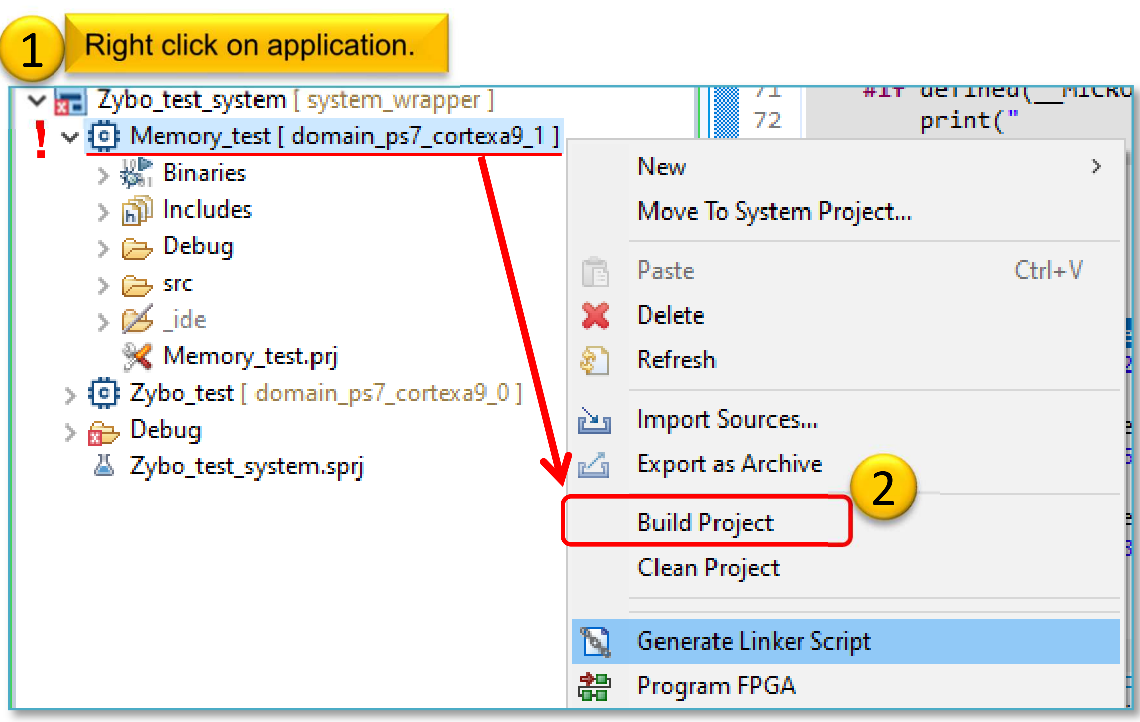
Linker Script generation (Basic)

- Xilinx menu → Generate Linker Script (`lscript.ld`) - RAM0



Build project

- 1. Select Application project (e.g. `Memory_test`)
- 2. Project menu → Build Project…   in two steps:
  - Build BSP (`system_wrapper`)
  - Build software application



Example II.) Question & Answer

- What is the size of Memory_test application?
  - ~37 Kbyte

```
'Invoking: ARM v7 Print Size'
arm-none-eabi-size Memory_test.elf  |tee "Memory_test.elf.size"
   text    data     bss     dec     hex filename
  27872    1184    8248   37304    91b8 Memory_test.elf
'Finished building: Memory_test.elf.size'
```

Memory Test – Code analysis

- Examine the source codes for **MemoryTest** SW application:
- **\src**
  - memory_config.h (structure definition)
    - `memory_range_s`
  - memory_config_g.c (structure `ps7_ddr_0` vs. `ps7_ram_1`)
  - *memorytest.c* (`main()` function)
  - platform_config.h
  - platform.c
- **\<*_bsp>\ps7_cortexa9_x\include**
  - xparameters.h (`#defines` based on .xsa/.xml)

HW debugging steps – MemoryTest

1. Create a new Debug Configuration (GDB) for Memory Test
2. Lunch Debugger
3. Set-up Debug-serial port (VITIS terminal)
4. HW debug – Memory Test
5. Examine results – serial logs
6. Terminate and remove debug process!

Serial log in VITIS terminal.

```
Connected to COM18 at 115200
--Starting Memory Test Application--
NOTE: This application runs with D-Cache disabled.As a result, cacheline requests will not be
generated

Testing memory region: ps7_ddr_0
    Memory Controller: ps7_ddr_0
         Base Address: 0x100000
                 Size: 0x1FF00000 bytes
          32-bit test: PASSED!
          16-bit test: PASSED!
           8-bit test: PASSED!

Testing memory region: ps7_ram_1
    Memory Controller: ps7_ram_1
         Base Address: 0xFFFF0000
                 Size: 0xFE00 bytes
          32-bit test: PASSED!
          16-bit test: PASSED!
           8-bit test: PASSED!

--Memory Test Application Complete--
Successfully ran Memory Test Application
```

Example III.) Modify Memory Test

- Modify the memory test application so that
  - it tests a larger external `ps7_ddr_0` memory range (up to *8-24* … and max. *511 MByte*, instead of *1024* data words!),
  - and for different 8- / 16- / 32-bits of data widths.

- HINT: 511 MByte = `0x1FF0_0000` =
  - `range->size`
  - TestMem32(): `(range->size)/64` (?), or
  - TestMem16(): `(range->size)/32`,
  - TestMem8(): `(range->size)/16`.

Example III.) Memory Test (cont.)

- Use the "`memset()`" function to reset/null the regions to be tested and add it to the `memorytest.c` application code:

Name https://linux.die.net/man/3/memset
memset - fill memory with a constant byte
Synopsis
`#include <string.h> void *memset(void *s, int c, size_t n);`
Description
The **`memset()`** function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

```
#include "string.h"
...
memset((u32*)range->base, 0x0, 1024);
status = Xil_TestMem32((u32*)range->base, 1024, 0xAAAA5555, XIL_TESTMEM_INCREMENT);
print("          32-bit test: "); print(status == XST_SUCCESS? "PASSED!":"FAILED!");
print("\n\r");

memset((u16*)range->base, 0x0, 2048);
status = Xil_TestMem16((u16*)range->base, 2048, 0xAA55, XIL_TESTMEM_INCREMENT);
print("          16-bit test: "); print(status == XST_SUCCESS? "PASSED!":"FAILED!");
print("\n\r");

memset((u8*)range->base, 0x0, 4096);
status = Xil_TestMem8((u8*)range->base, 4096, 0xA5, XIL_TESTMEM_INCREMENT);
print("           8-bit test: "); print(status == XST_SUCCESS? "PASSED!":"FAILED!");
print("\n\r");
```

Example III.) Question & Answer

- What is the size of the modified Memory_test application?
  - ~37 Kbyte

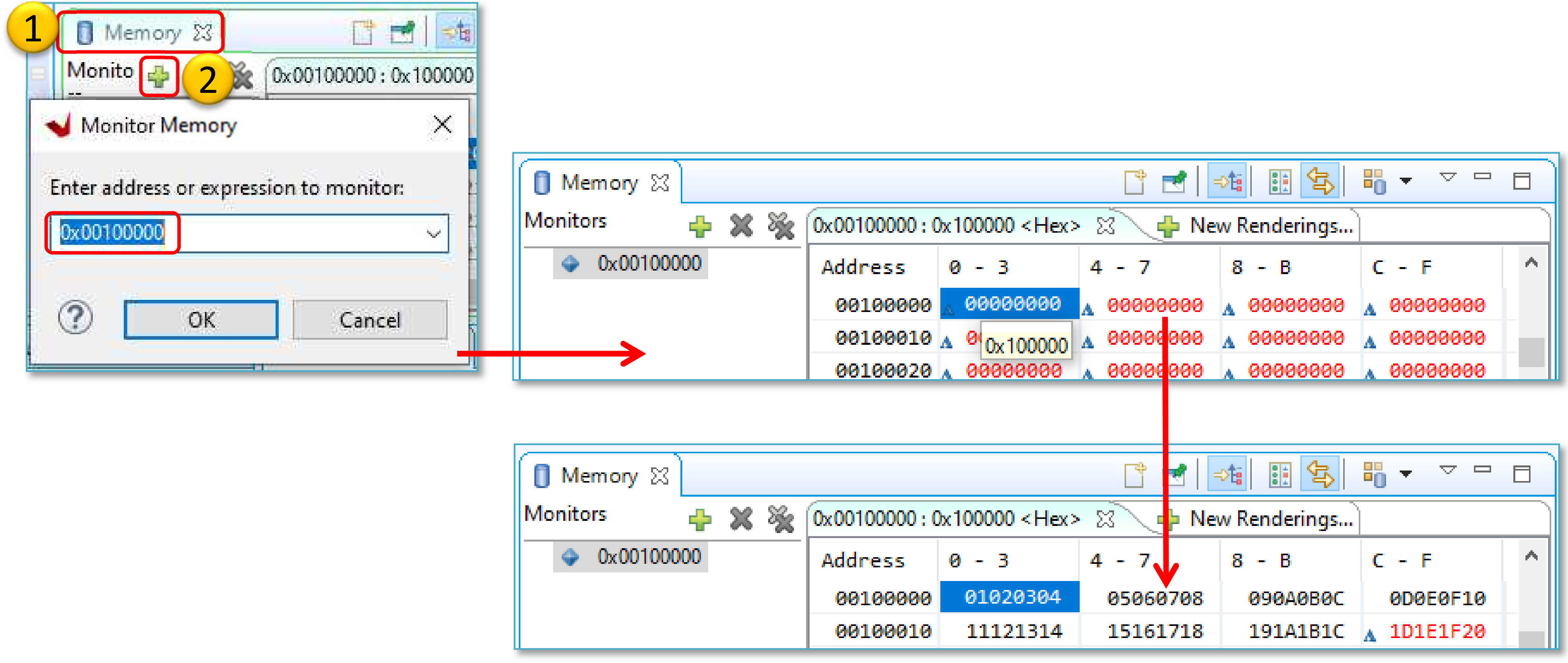
```
'Invoking: ARM v7 Print Size'
arm-none-eabi-size Memory_test.elf  |tee "Memory_test.elf.size"
   text    data     bss     dec     hex filename
  27960    1184    8248   37392    9210 Memory_test.elf
'Finished building: Memory_test.elf.size'
```

Example III.) Build and Debug

1. Build the modified Memory Test application (.elf)
2. Lunch the proper Debug configuration (GDB) for hardware debugging
3. Setup the VITIS serial terminal/Console (USB-serial port),
4. Connecting and setup a JTAG-USB programmer,
   - Configuring the FPGA (`.BIT` if PL-side existing)
5. Debug procedure (insert breakpoints, stepping, run, etc.)
   - **Watching variables and examine memory monitor !**
6. At the end of debug procedure do not forget to Terminate and Remove the actual Debug configuration (GDB)!
7. That’s all :D

HW debug – Memory monitoring

- Set a „Memory monitor” during the HW debug Memory
  - Window → Show View → Debug → Memory (if not visible)
    - [+] Add section: **0x0010 0000 (or range->base)**



Example IV.) Memory Test Timing

- Modify the memory test application so that
  - it measures the elapsed time in each test cases (for 8-/16-/32-bits os data). HINT:

> Note: COUNTS_PER_SECOND - Global Timer is always clocked at half of the CPU frequency! (650/2 = 325 MHz) Therefore multiplication by 2 is necessary.

```
#include "xtime_l.h"

void test_memory_range(struct memory_range_s *range) {

    XTime tStart, tEnd;
    …
    xil_printf("Global timer clock freq: %d [MHz]\n", COUNTS_PER_SECOND/1000000);
    xil_printf("Note: Global Timer is always clocked at half of the CPU freq\n");
    xil_printf("-> ARM PLL clock freq: %d [MHz]\n", 2*COUNTS_PER_SECOND/1000000);
    XTime_GetTime(&tStart);
    status = Xil_TestMem32((u32*)range->base, 1024, 0xAAAA5555,
                                                        XIL_TESTMEM_INCREMENT);
    XTime_GetTime(&tEnd);

    printf("Output took %llu [clock cycles].\n", 2*(tEnd - tStart));
    int time = (tEnd - tStart) / (COUNTS_PER_SECOND/1000000);
    printf("Output took %d [us].\n", time);

    …
}
```

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR YOUR KIND ATTENTION!

BEFEKTETÉS A JÖVŐBE