



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



MAGYARORSZÁG  
KORMÁNYA

**Európai Unió**  
Európai Strukturális  
és Beruházási Alapok



**BEFEKTETÉS A JÖVŐBE**



# 4. XILINX VIVADO

Creating BSB - Base System Build and Board „bring-up”

**SZÉCHENYI**  2020



MAGYARORSZÁG  
KORMÁNYA

**Európai Unió**  
Európai Strukturális  
és Beruházási Alapok



**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 (\*\*):



Letölthető gyakorlati anyag

Digilent ZyBo Fejlesztőkártyákhoz (BSP, XDC):  
XDC (FPGA lábkiosztás - GIT master):  
[Zybo-Master.xdc](#)  
Base System Pack (BSP):  
**[Vivado Board Files \(2020.1\)](#)**

Digilent Zybo hivatalos weboldal:  
[Zybo Zynq-7000 ARM/FPGA SoC Trainer Board](#)

Xilinx Vivado/VITIS telepítési útmutató:  
[ESD\\_00\\_VITIS\\_Vivado\\_Installation\\_guide](#)

Vivado HW manager - próba .bit:  
[TM\\_osszeado\\_3bit](#) Alaptesztek:  
hello world / memóriateszt/ periféria teszt.

# 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** version of board files!



# XILINX VIVADO DESIGN SUITE

Creating Embedded Base System

**SZÉCHENYI**  2020



MAGYARORSZÁG  
KORMÁNYA

**Európai Unió**  
Európai Strukturális  
és Beruházási Alapok



**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

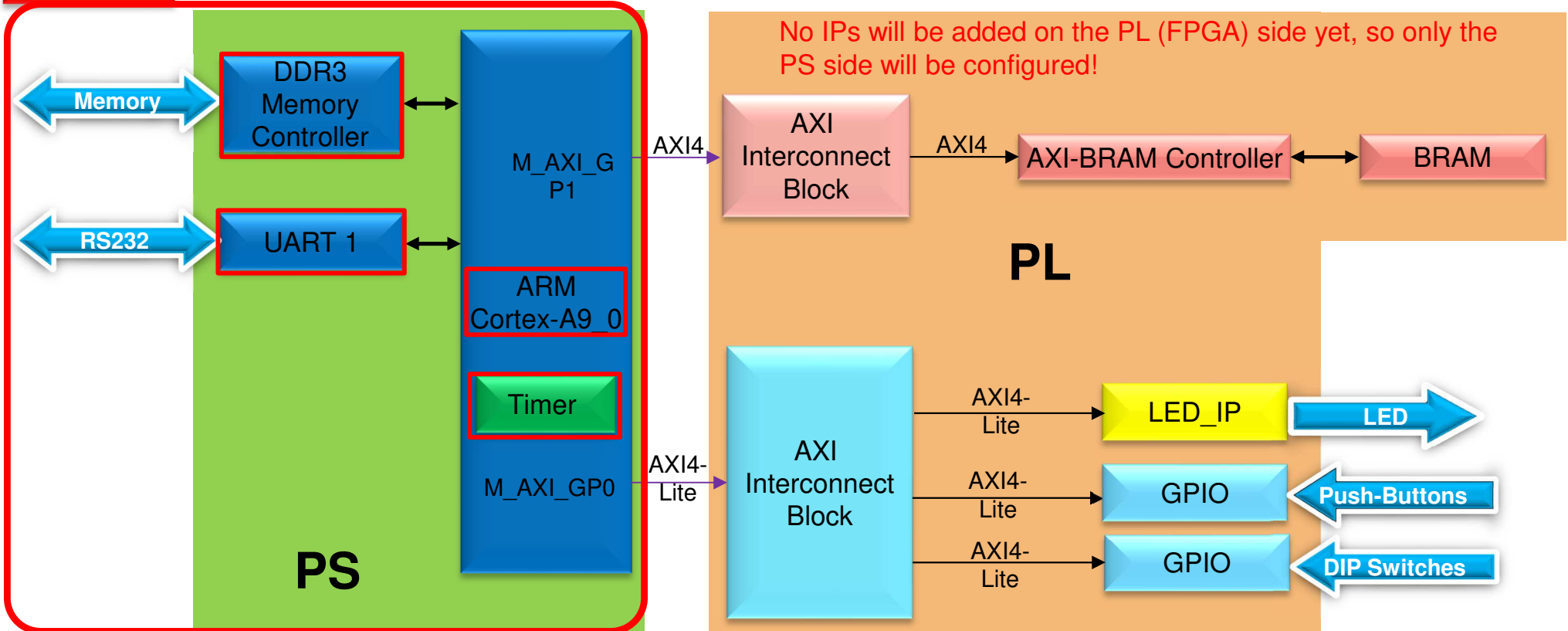


# Main steps to solve the task

- 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

LAB\_01



## 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



Vivado 2020.1

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

Not Xilinx Vivado HLS !

Vivado 2020.1

File Flow Tools Window Help Q- 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 >

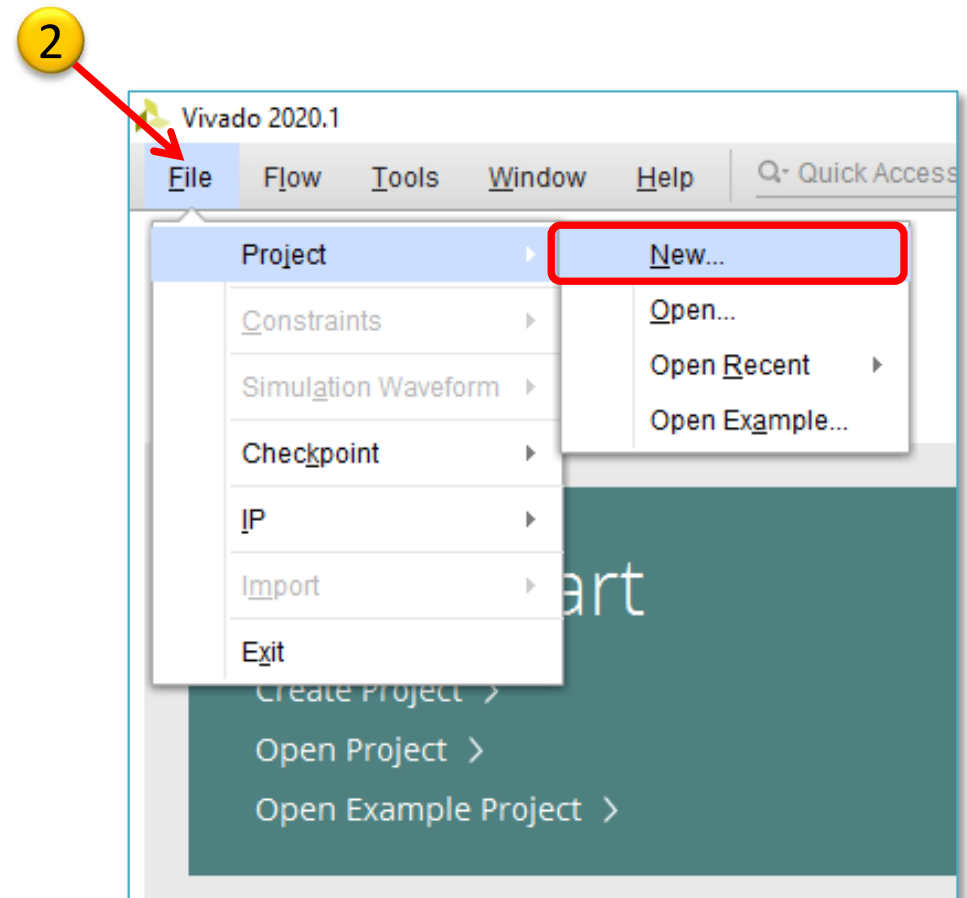
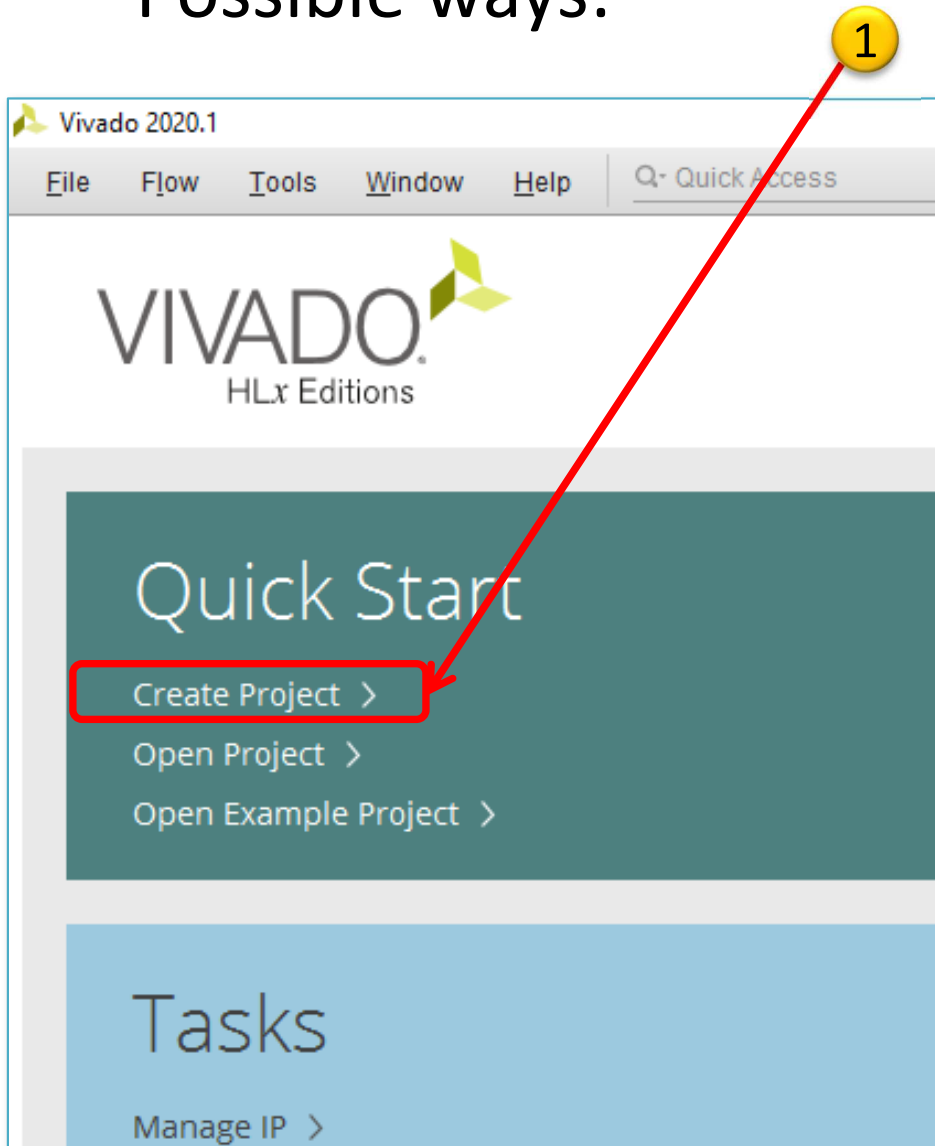
#### Recent Projects

- 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\_V1344D\_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\_V1344D\_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.)

**New Project**

**Project Name**  
Enter a name for your project and specify a directory where the project data files will be stored.

Project name: lab01

Project location: F:/Vivado\_2020.1

Create project subdirectory

Project will be created at: F:/Vivado\_2020.1/lab01

< Back   **Next >**   Finish   Cancel

Set the path and name of a new project <name.xpr>

 lab01 xpr

# Create a new project (cont.)

New Project

**Project Type**  
Specify the type of project to create.

**RTL Project**  
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

Do not specify sources at this time

Project is an extensible Vitis platform

**Post-synthesis Project**  
You will be able to add sources, view device resources, run design analysis, planning and implementation.

Do not specify sources at this time

**I/O Planning Project**  
Do not specify design sources. You will be able to view part/package resources.

**Imported Project**  
Create a Vivado project from a Synplify, XST or ISE Project File.

**Example Project**  
Create a new Vivado project from a predefined template.

? < Back **Next >** Finish Cancel

Select „RTL project” (for later block design and IP integration).

Do not add other resources (since they don't exist yet :)

\*Extensible VITIS platform means „new” development flow. We do not want to use it!

# Choose HW development board

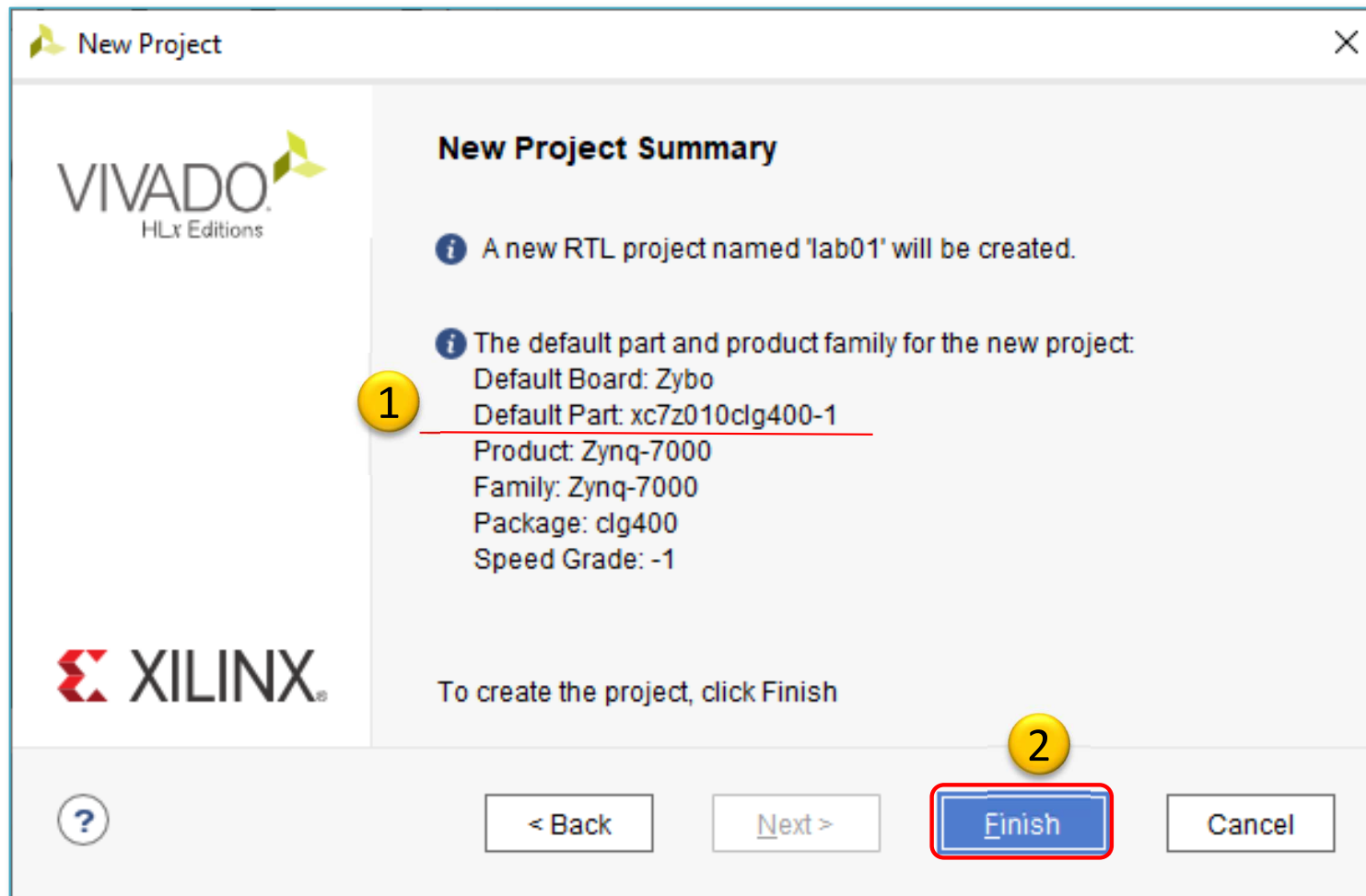
\*\* „Zybo” can only be selected after installing the support package (see slides 5-6)

The screenshot shows the 'New Project' dialog box with the 'Default Part' section. The 'Boards' tab is selected. The search filters are set to Vendor: digilentinc.com, Name: Zybo, and Board Rev: Latest. The 'Zybo' board is highlighted in the table. The 'Next >' button is highlighted.

Display Name	Preview	Vendor	File Version	Part	I/O Pin Count	Board Rev
Zybo		digilentinc.com	2.0	xc7z010clg400-1	400	B.4

\*\*

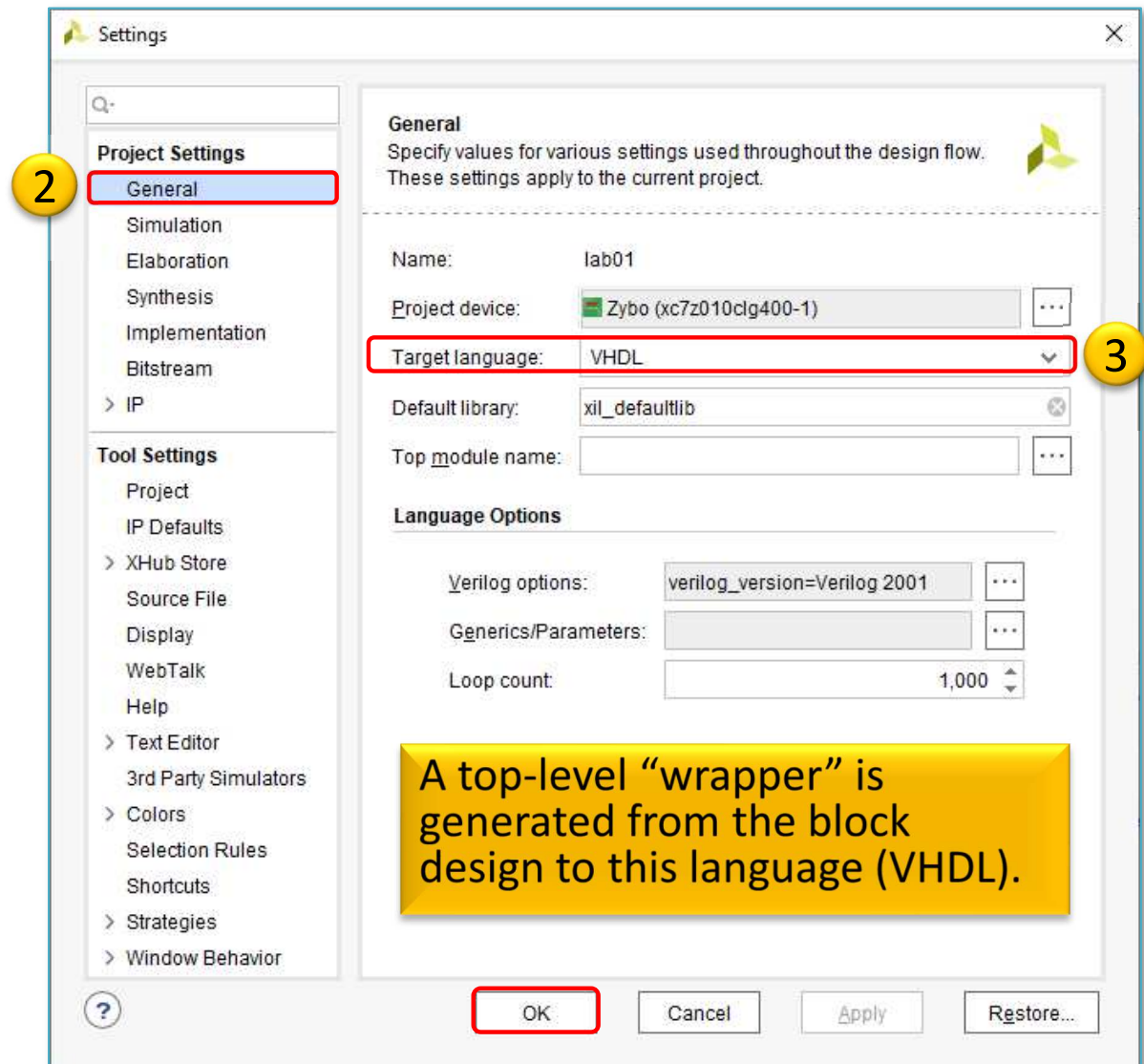
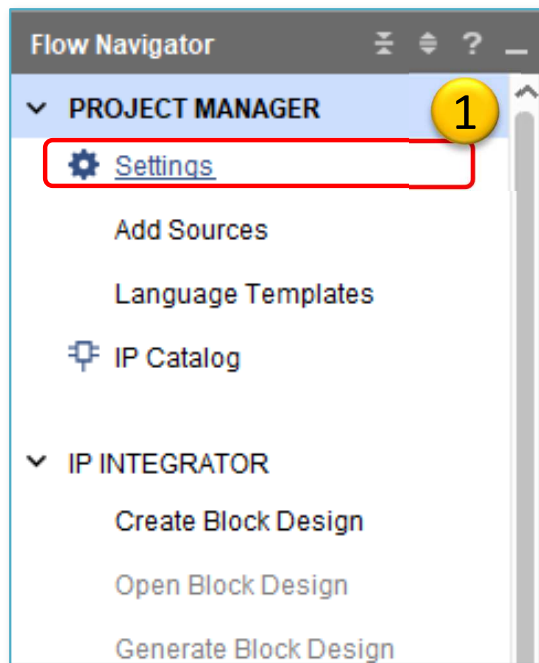
# Project summary





# Project settings - VHDL

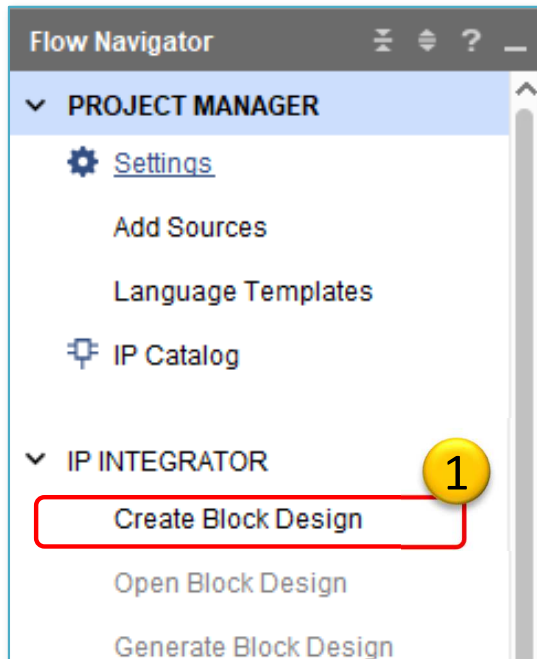
- Project Manager → Settings → General → Select VHDL, then OK.



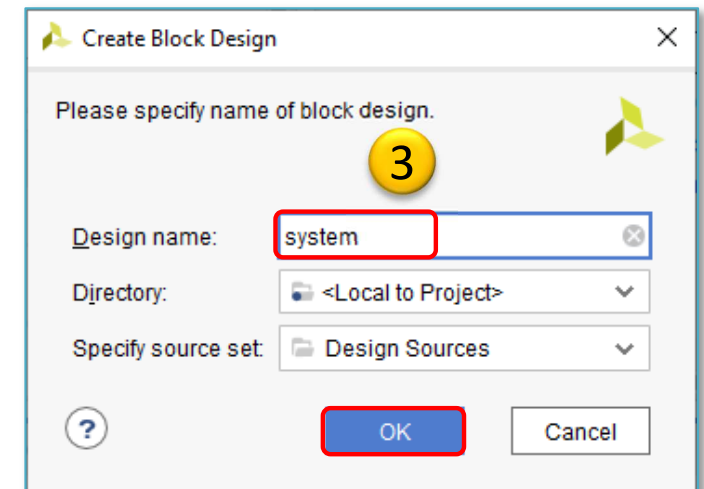
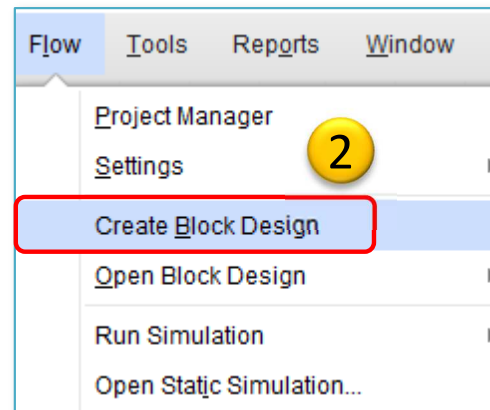
# Embedded System – IP Integrator (IPI)

## 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.

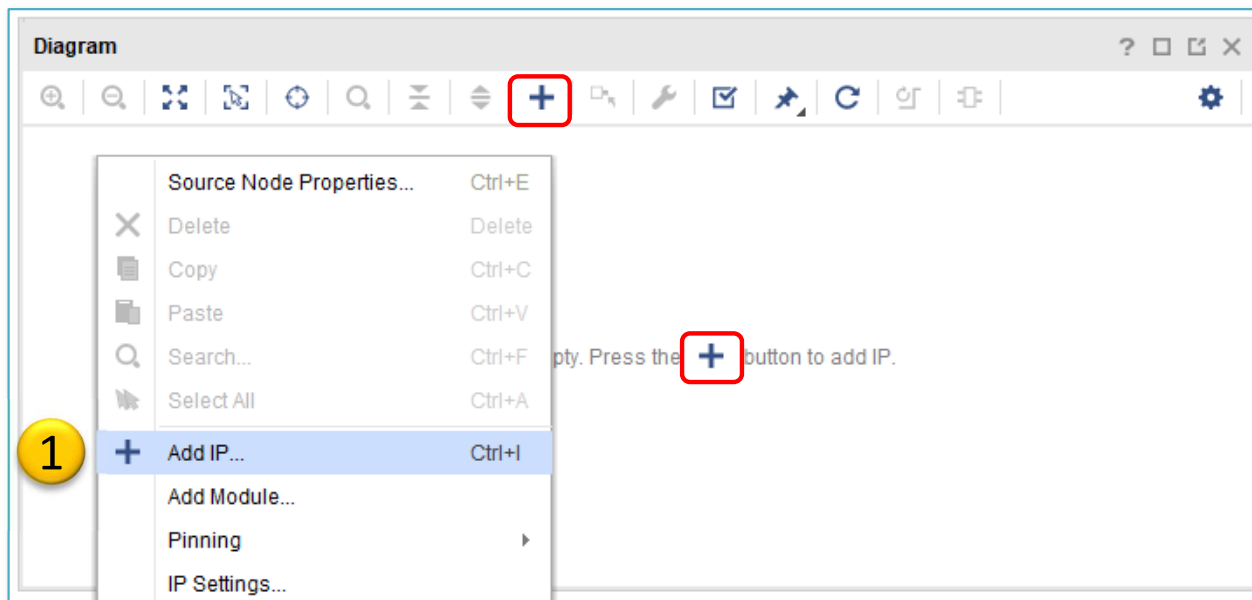
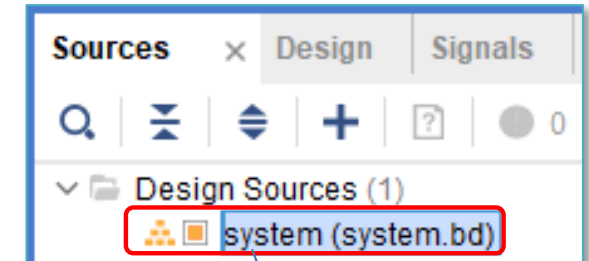


OR

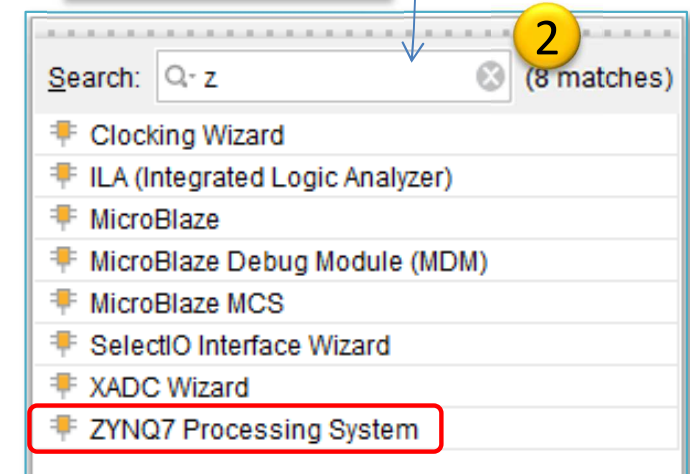


# Embedded System – Add IP

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



Filtering search





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:

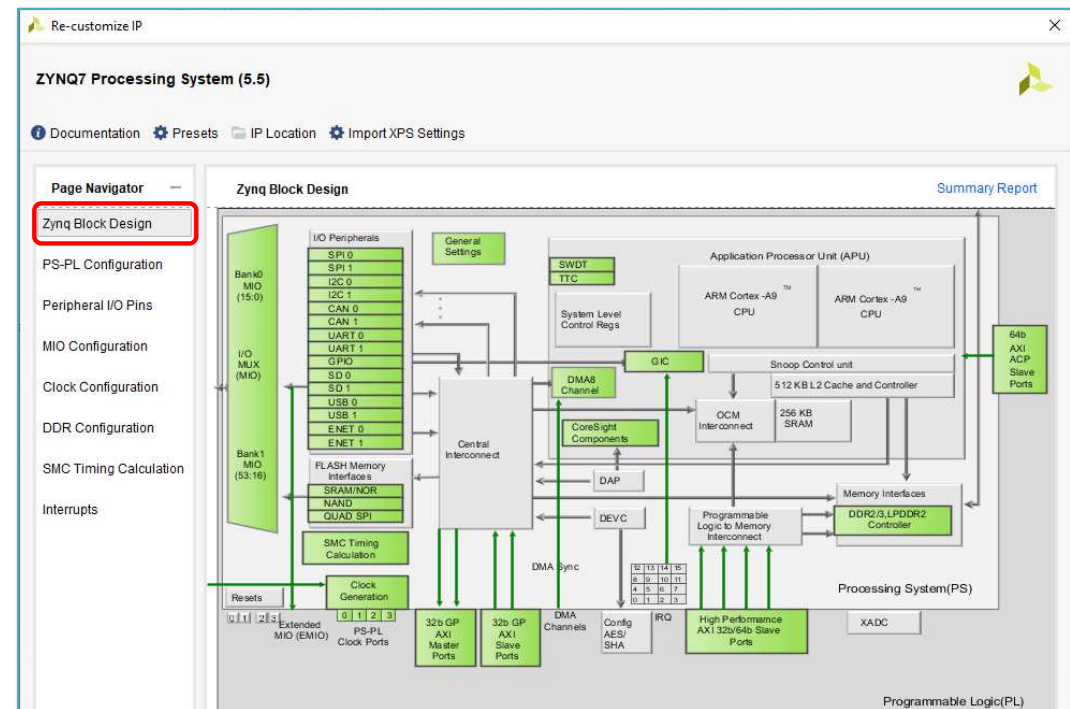
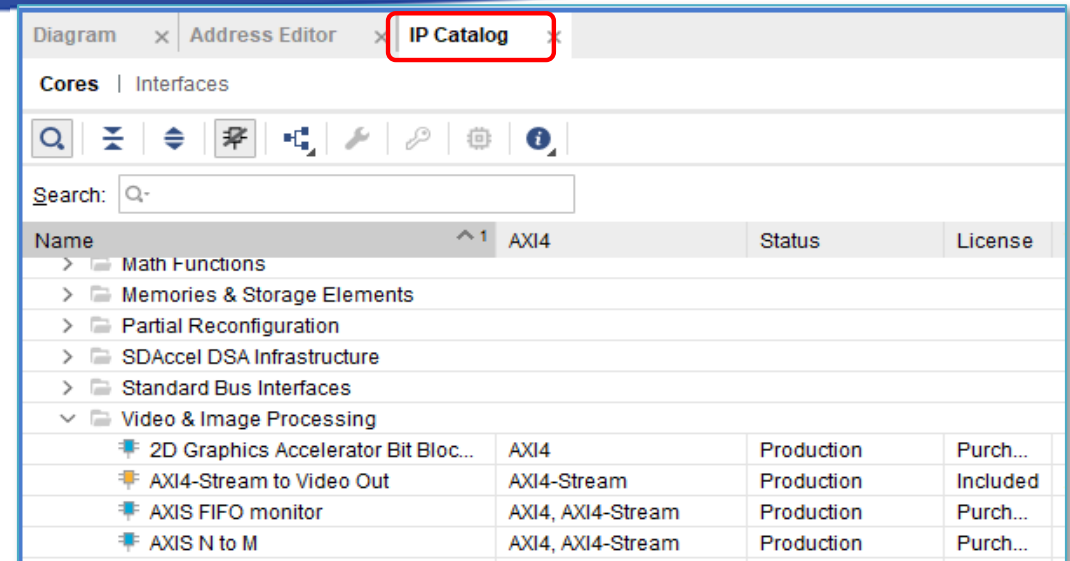
- compatibility 

- 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:

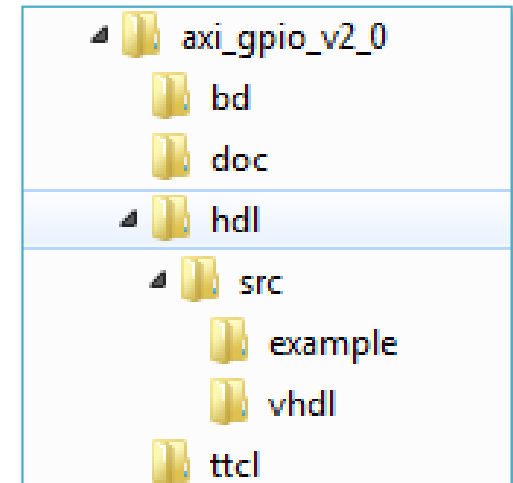
**Project Manager** →

**Settings** →

**IP Defaults / Repository** tab

- `%XILINX_INSTALL%\Vivado\2020.X\data\ip`

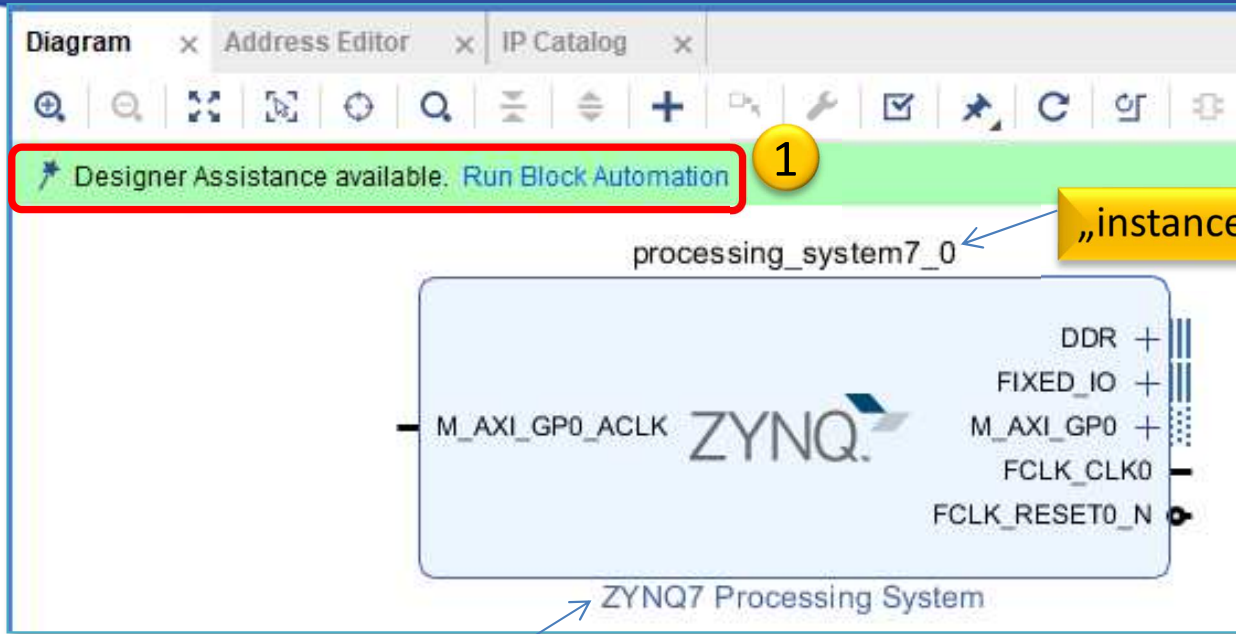
PI:



PI:

```
<?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" xml:
  ..<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



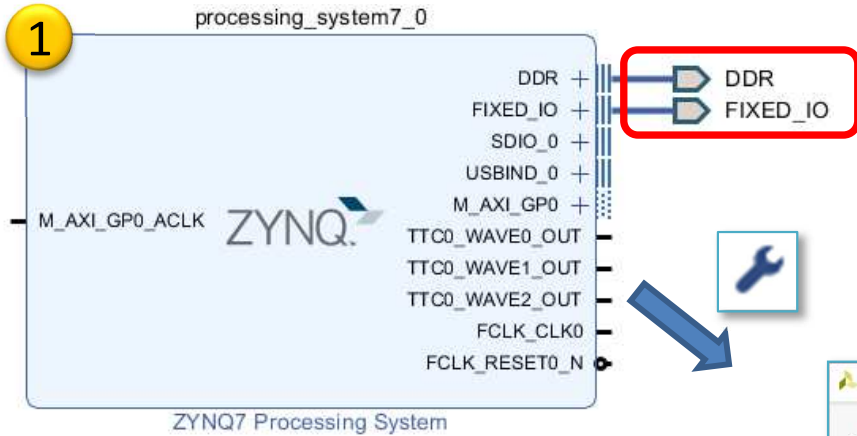
IP catalog name (fixed)

„instance” name (arbitrary)

Keep everything at the default setting. OK.

The screenshot shows the 'Run Block Automation' dialog box. The 'All Automation (1 out of 1 selected)' checkbox is checked, and the 'processing\_system7\_0' instance is selected. The 'Description' section explains that this option sets the board preset on the Processing System. The 'Options' section shows 'Make Interface External' set to 'FIXED\_IO, DDR', 'Apply Board Preset' checked, and 'Cross Trigger In' and 'Cross Trigger Out' both set to 'Disable'. A yellow callout box labeled '2' points to the 'OK' button.

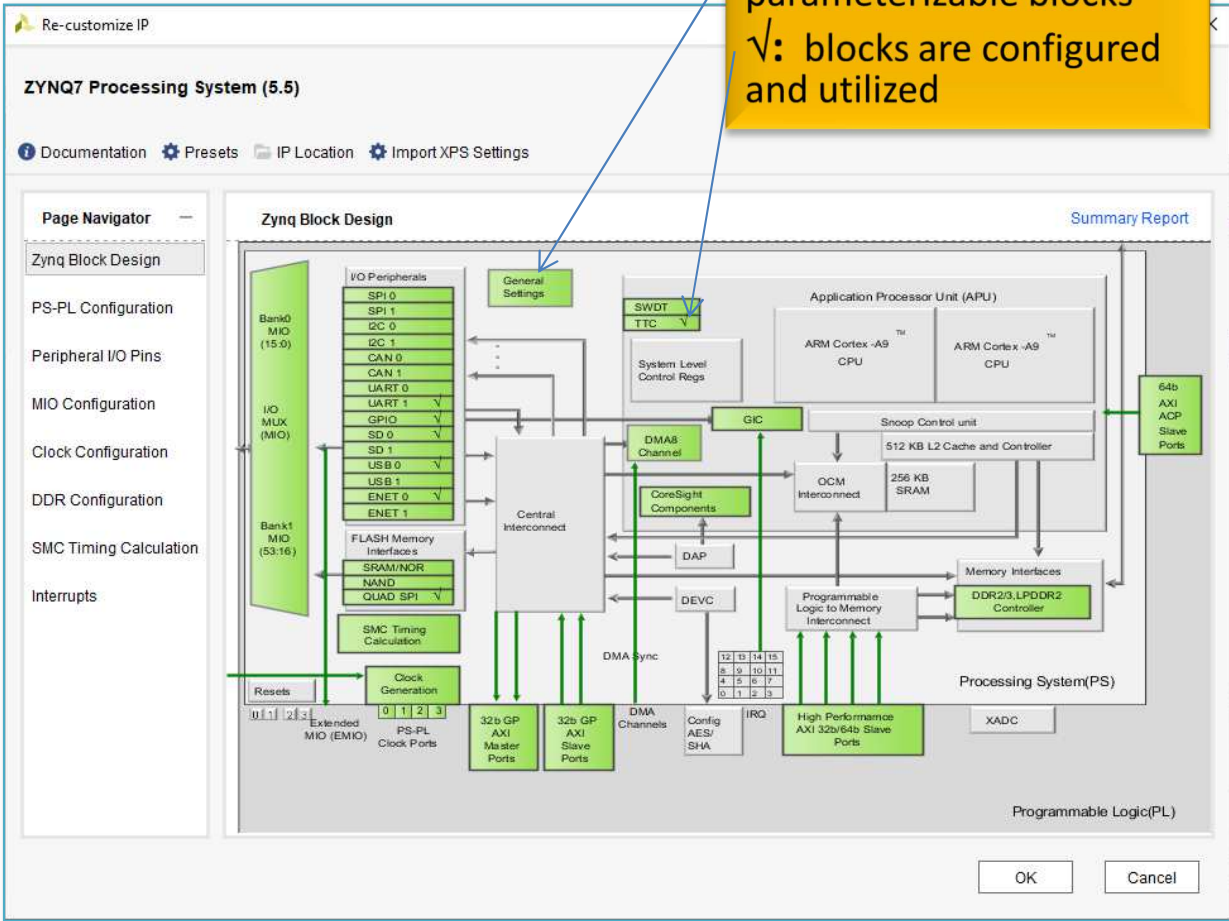
# Zynq Blokk Design (DDR and I/O ports)



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)





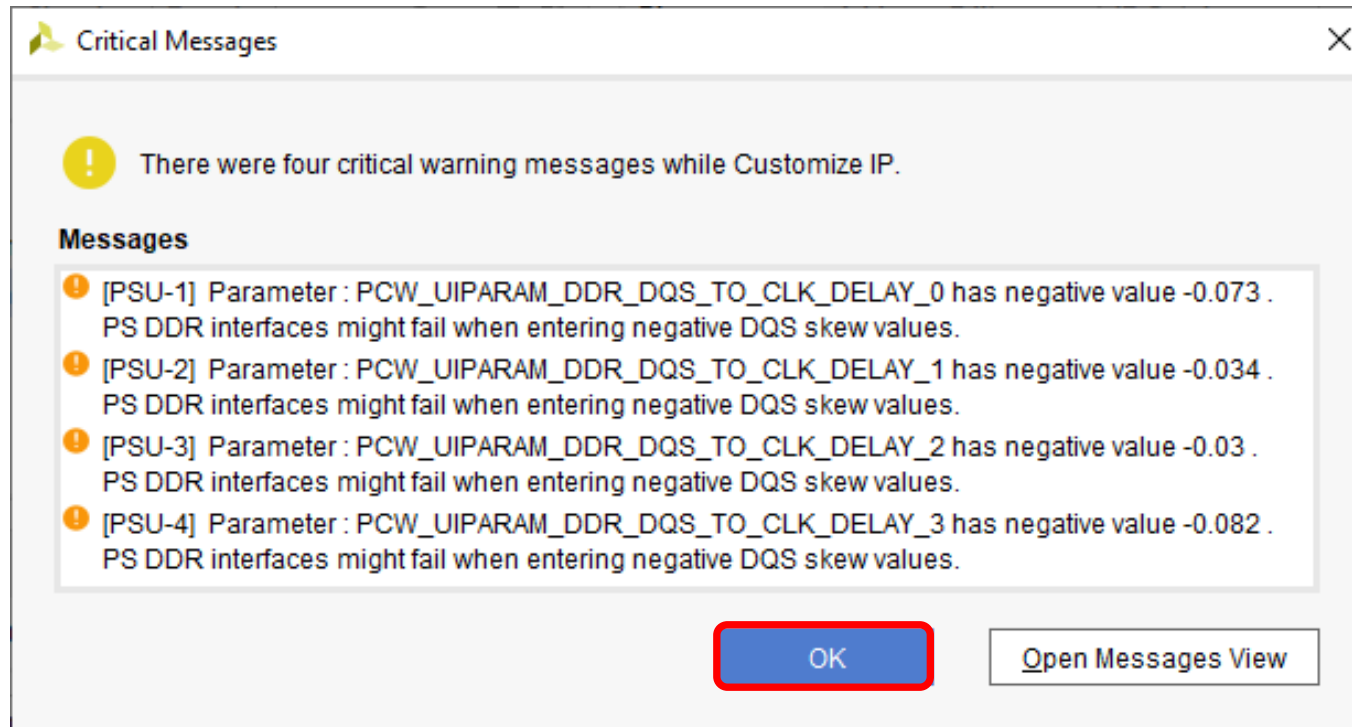
# Zynq – PS-PL configuration

1 PS-PL Configuration

Name	Select	Description
> General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
> GP Master AXI Interface		
> M AXI GP0 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 0
> M AXI GP1 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 1
> GP Slave AXI Interface		
> HP Slave AXI Interface		
> ACP Slave AXI Interface		
> DMA Controller		
> PS-PL Cross Trigger interface	<input type="checkbox"/>	Enables PL cross trigger signals to PS and vice-versa

- *General* [+] ->
  - *UART1 Baud rate: 115.200 ?*
  - *Enable Clock Resets : disable FCLK\_RESETO\_N.*
- *AXI Non Secure Enablement* [+] ->
  - *GP Master AXI interface: disable M AXI GPO interface.*

# Zynq – Critical warning messages

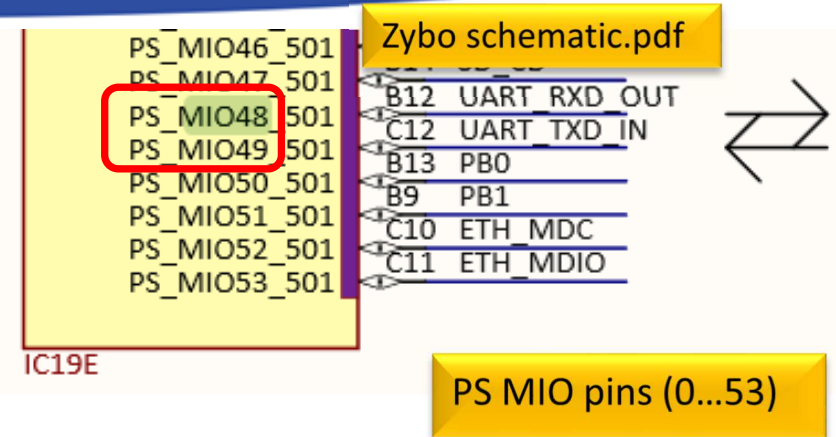


- This is coming from the board definition file for ZYBO. This warning might be considered only for new/custom board design, so it can be ignored! Just click OK.
- *Reference:* [https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual?\\_ga=2.21480123.1048852157.1597218516-1725411217.1597064829#hardware\\_errata](https://reference.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



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

UART1: can be enabled by clicking on it.

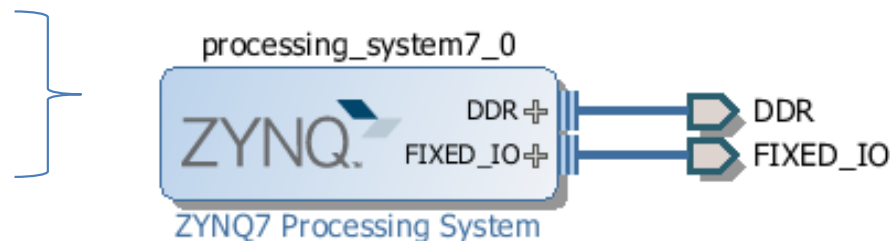
# Zynq – Clock configuration

- PL Fabric Clocks (FPGA) →
  - disable FCLK\_CLK0


The screenshot shows the 'Clock Configuration' window with the 'PL Fabric Clocks' section expanded. A yellow circle with the number '1' highlights the 'Clock Configuration' item in the left-hand 'Page Navigator'. A red box highlights the unchecked checkbox for 'FCLK\_CLK0' in the table below.

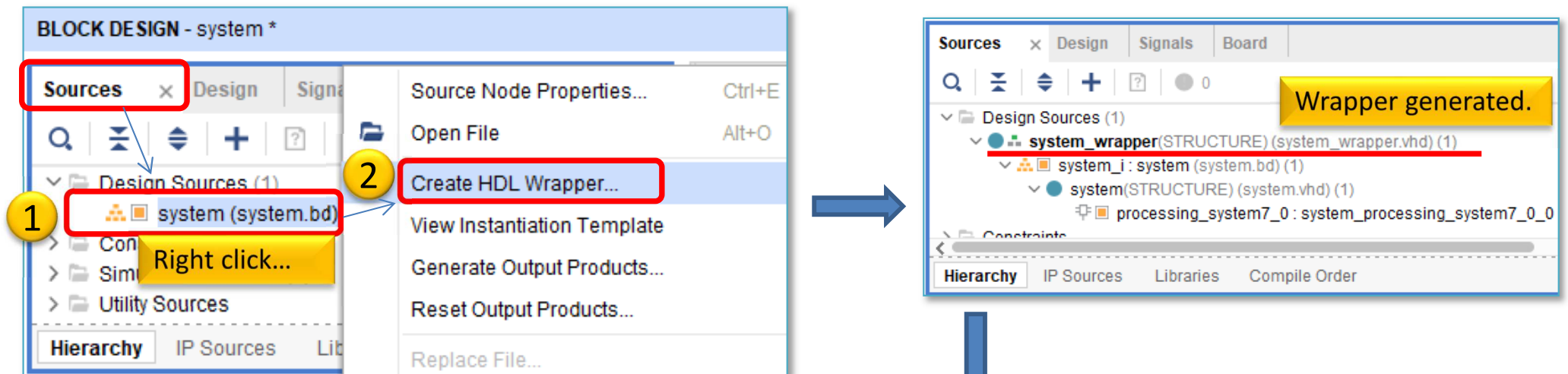
Component	Clock Source	Requested Frequ...	Actual Frequency(...)	Range(MHz)
Processor/Memory Clocks				
CPU	ARM PLL	650	650.000000	50.0 : 667.0
DDR	DDR PLL	525	525.000000	200.000000 : 534.000...
IO Peripheral Clocks				
SMC	IO PLL	100	10.000000	10.000000 : 100.000000
QSPI	IO PLL	200	10.000000	10.000000 : 200.000000
ENET0	IO PLL	1000 Mbps	10.000000	
ENET1	IO PLL	1000 Mbps	10.000000	
SDIO	IO PLL	100	10.000000	10.000000 : 125.000000
SPI	IO PLL	166.666666	10.000000	0.000000 : 200.000000
> CAN				
PL Fabric Clocks				
<input type="checkbox"/> FCLK_CLK0	IO PLL	100	10.000000	0.100000 : 250.000000

- 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!



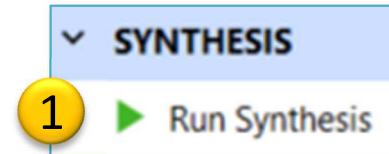
System\_wrapper.vhd generated:

```
Address Editor x IP Catalog x system_wrapper.vhd x <> ? □
2018.3/lab01/project_1/project_1.srscs/sources_1/bd/system/hdl/system_wrapper.vhd x
🔍 ⏪ ⏩ ✂ 📄 📄 ✖ // 📄 💡 ⚙
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 );
```

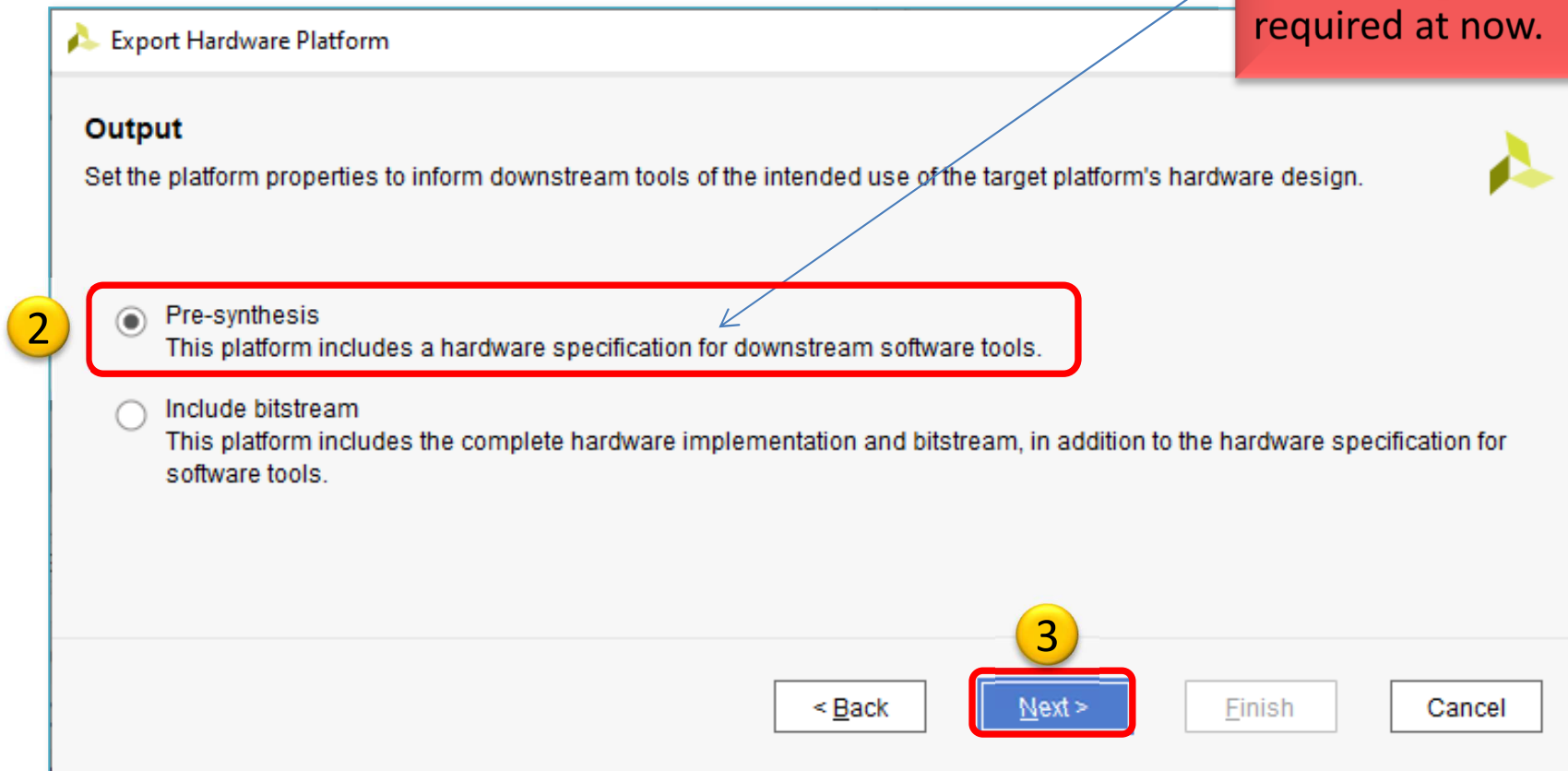
# Export HW → VITIS (~SDK)

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-synthesis as an option**

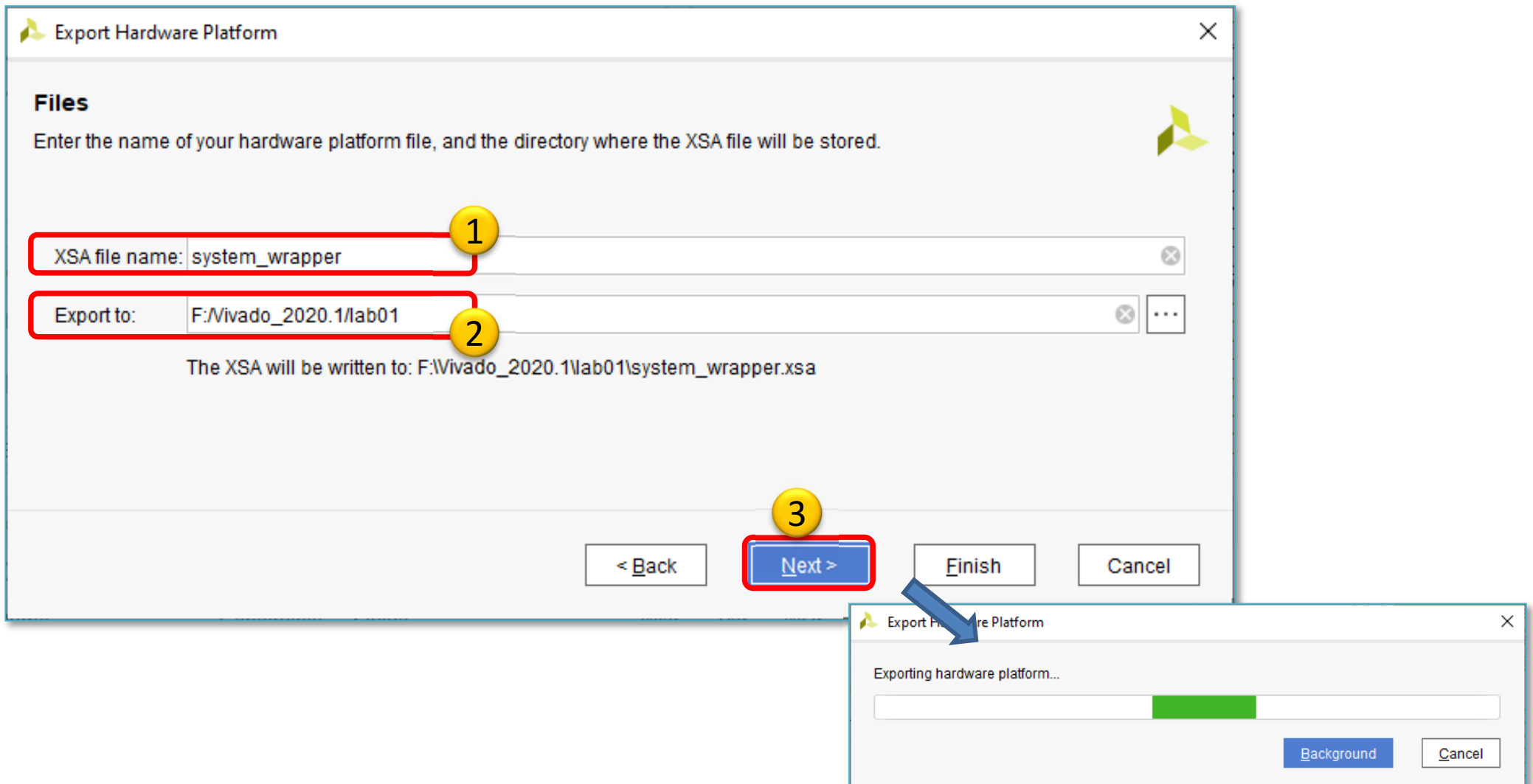


Since the PL (FPGA) side has not been configured, a bitstream generation is not required at now.



# Export HW → VITIS (cont.)

Set **XSA**\* file name and export directory path:



# 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

**SZÉCHENYI**  2020




MAGYARORSZÁG  
KORMÁNYA

**Európai Unió**  
Európai Strukturális  
és Beruházási Alapok



**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



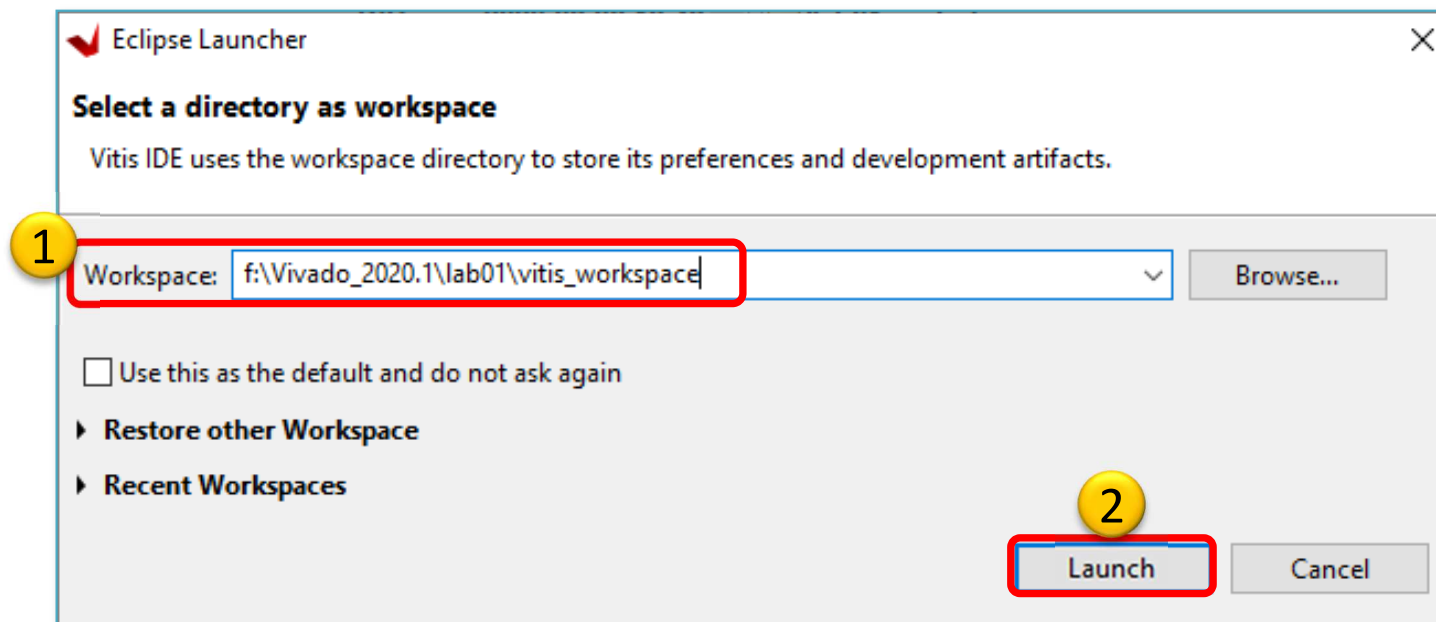
From Vivado: Tools menu → Launch VITIS IDE

OR externally

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

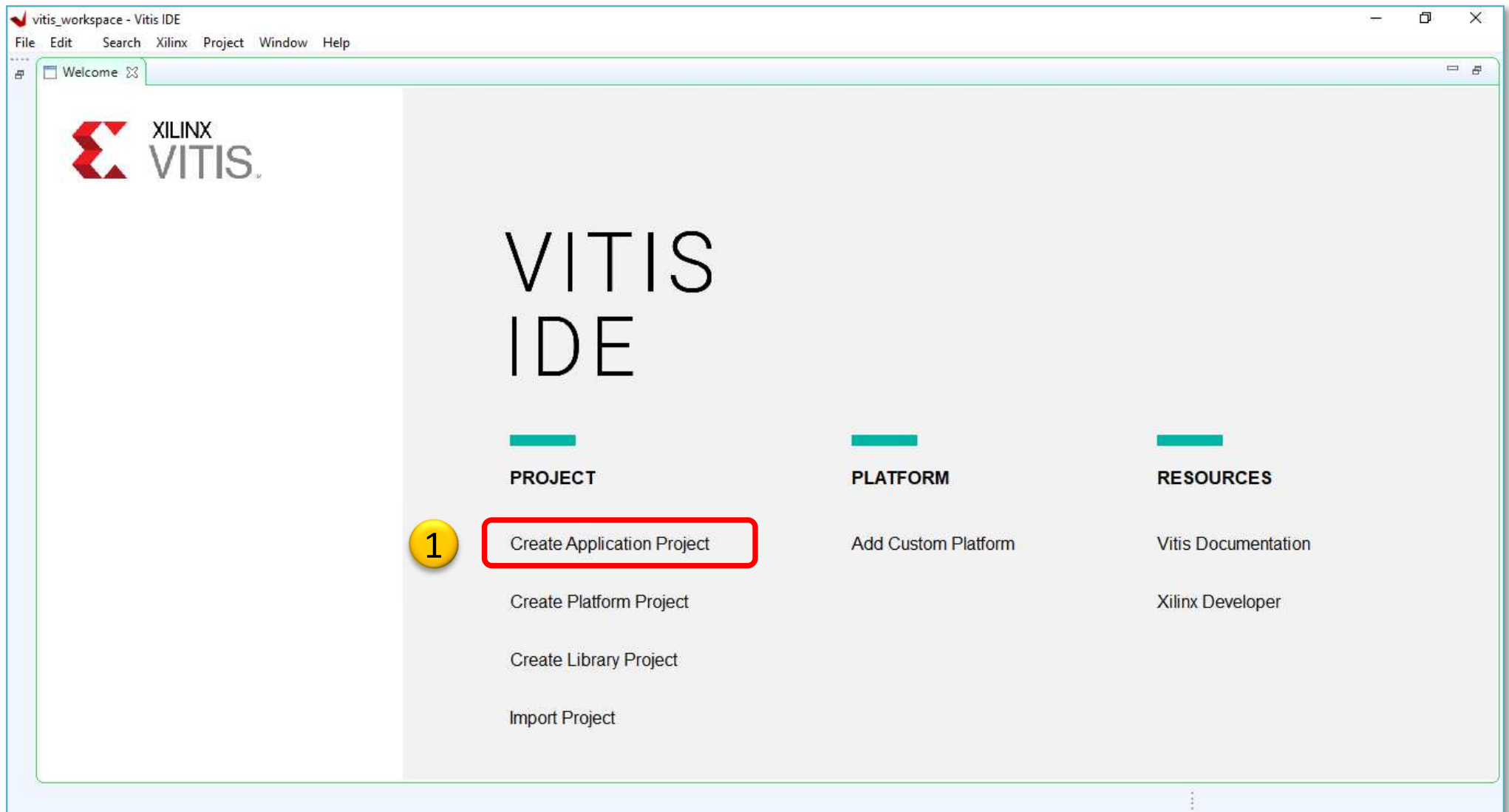
Do Not run Xilinx VITIS HLS 2020.x !

- 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

New Application Project

## Create a New Application Project

This wizard will guide you through the 4 steps of creating new application projects.

1. Choose a **platform** or create a **platform project** from Vivado exported XSA
2. Put application project in a **system project**, associate it with a processor
3. Prepare the application runtime – **domain**
4. Choose a template for application to quick start development

HW / FW Processor

Platform Project

BSP Domain

XSA !

System Project

SW App

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.)

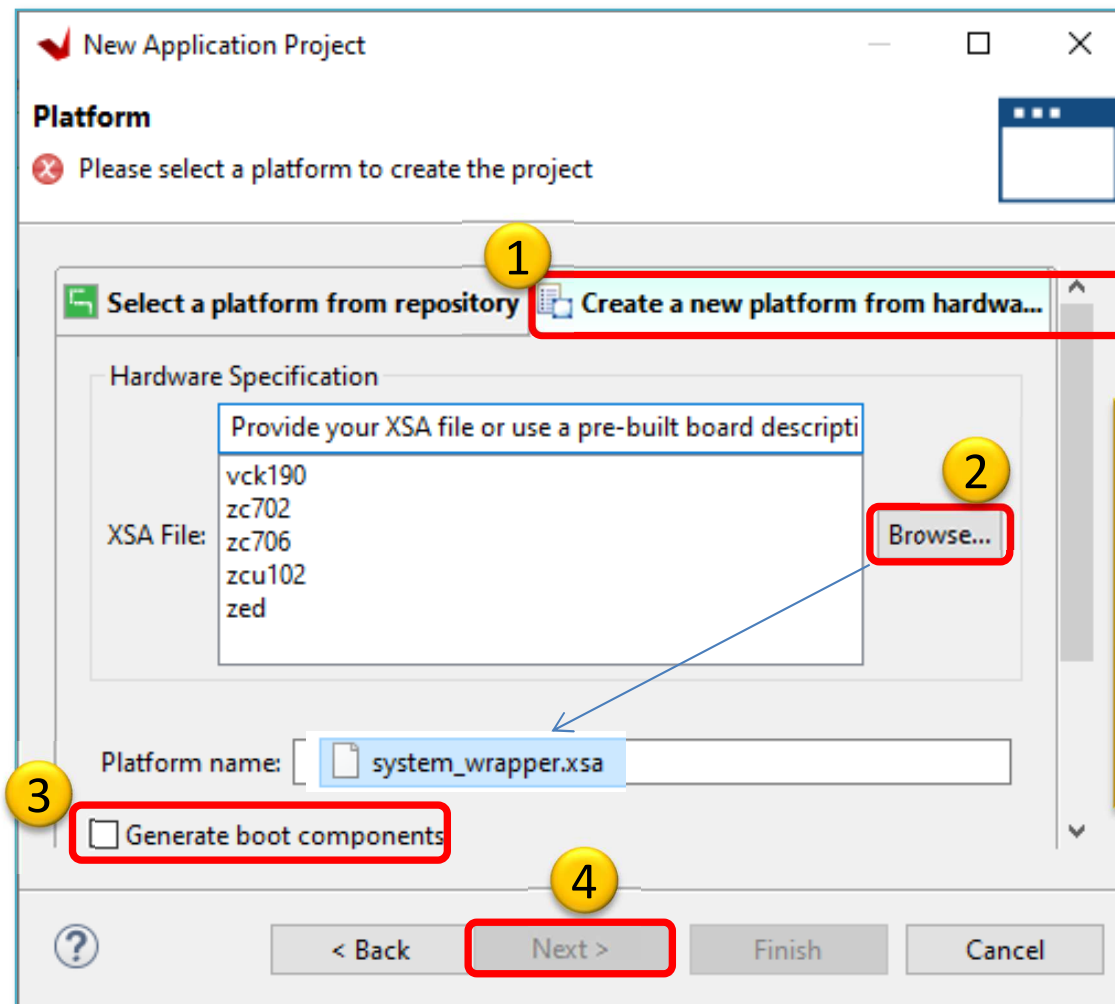
- A platform provides hardware information and software environment settings.
- A system project contains one or more applications that run at the same time.
- A domain provides runtime for applications, such as operating system or BSP.
- A workspace can contain unlimited platforms and unlimited system projects.

Skip welcome page next time. (Can be reached with Back button)

1

< Back **Next >** Finish Cancel

# Import XSA



1. Create a new platform by importing XSA (created in Vivado!)
2. Choose proper XSA
3. Do Not select „Generate boot components” now.
4. NEXT.

# Create Application project

**New Application Project**

**Application Project Details**  
Specify the application project name and its system project properties

1  
Application project name:

System Project  
Create a new system project for the application or select an existing one from the workspace

Select a system project

System project details  
2  
System project name:

Target processor  
3  
Select target processor for the Application project.

Processor	Associated applications
<input type="text" value="ps7_cortexa9_0"/>	Zybo_test
ps7_cortexa9_1	
ps7_cortexa9 SMP	

Show all processors in the hardware specification

4

1. Type an app. name: „Zybo\_test”
2. Leave system project name as default.
3. Select ARM Cortex-0 core
4. NEXT.

# Create Application project – Domain

**New Application Project**

**Domain**

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain  
+ Create new...

Domain details

Name: domain\_ps7\_cortexa9\_0

Display Name: domain\_ps7\_cortexa9\_0

Operating System: standalone 1

Processor: ps7\_cortexa9\_0

Architecture: 32-bit

2

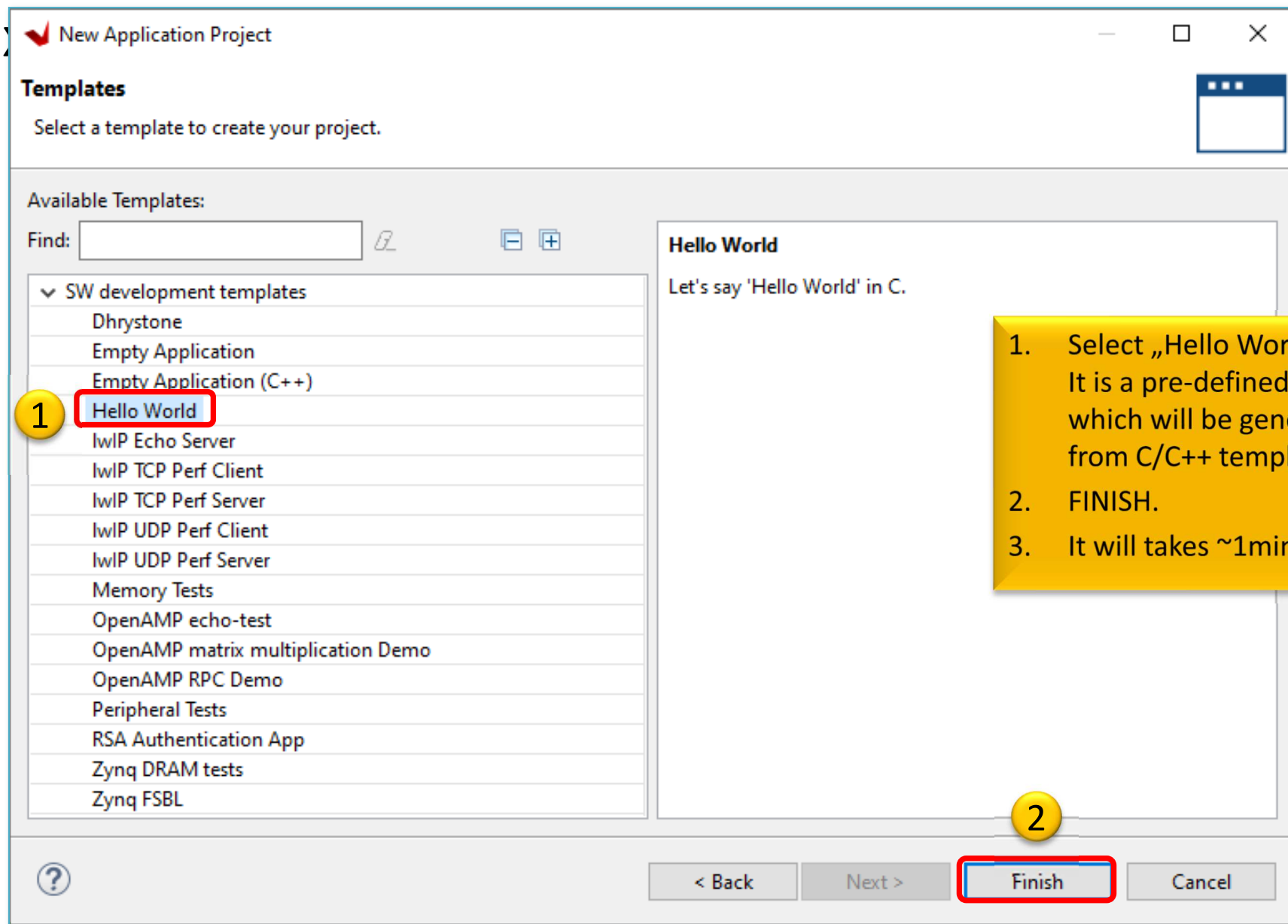
< Back Next > Finish Cancel

1. Select „standalone“ OS, because we have not a full featured OS (e.g. Linux) at now.
2. NEXT.



# Example I.) Creating HelloWorld application from template

- New Application Project



# VITIS GUI – Main window

The screenshot displays the Vitis IDE interface for a project named 'Zybo\_test'. The 'Application Project Settings' window is open, showing the following configuration:

- Project name: Zybo\_test
- Platform: system\_wrapper
- Runtime: cpp
- Domain: domain\_ps7\_cortexa9\_0
- CPU: ps7\_cortexa9\_0
- OS: standalone

The 'Hardware Specification' link is highlighted with a red box and a blue arrow pointing to the callout box. The callout box contains the following text:

**Standalone** is a simple, low-level software layer. It provides access to

- **basic processor features:** \$ caches, IRQ interrupts and exceptions
- and **basic features of a hosted environment**, e.g. standard I/O, profiling, abort and exit

# VITIS – HW platform

The screenshot displays the Vitis IDE interface for a project named 'system\_wrapper'. The Explorer on the left shows the project structure, with 'system\_wrapper.xsa' highlighted. The main editor shows the 'Hardware Platform Specification' for 'system\_wrapper.xsa'. The 'Design Information' section lists the target FPGA device as '7z010' (Part: xc7z010clg400-1) and the creation date as 'Wed Aug 12 13:03:24 2020'. Below this is the 'Address Map for processor ps7\_cortexa9[0-1]', which contains a table of hardware components and their memory addresses. A red box highlights the 'Base Address' and 'High Address' columns of this table. Two yellow callout boxes provide additional context: one points to the 'system\_wrapper.xsa' file in the Explorer, and another points to the table, indicating it lists and versions used PS peripherals. The console at the bottom shows the build status for the 'system\_wrapper' platform.

**HW platform from Vivado, description of elaborated embedded system**

**Memory address map (PS)**

**List and versions of used PS peripherals (below)**

Cell	Base Address	High Address	Slave Interface	Addr Range Type
ps7_ram_0	0x00000000	0x0002ffff	-	memory
ps7_ddr_0	0x00100000	0x1fffffff	-	memory
ps7_uart_1	0xe0001000	0xe0001fff	-	register
ps7_iop_bus_config_0	0xe0200000	0xe0200fff	-	register
ps7_slcr_0	0xf8000000	0xf8000fff	-	register
ps7_dma_s	0xf8003000	0xf8003fff	-	register
ps7_dma_ns	0xf8004000	0xf8004fff	-	register
ps7_ddrc_0	0xf8006000	0xf8006fff	-	register
ps7_dev_cfg_0	0xf8007000	0xf8007fff	-	register
ps7_xadc_0	0xf8007100	0xf8007120	-	register

Build Console [system\_wrapper]  
Nothing to build in platform 'system\_wrapper'

# 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:  $0x0002\ ffff-0x0000\ 0000 = 192\ \text{KByte}$
  - ps7\_ddr\_0:  $0x1fff\ ffff-0x0010\ 0000 = 511\ \text{MByte}$
  - ps7\_ram\_1:  $0xffff\ 0000-0xffff\ fdff = 63\ \text{KByte}$

# BSP – system.mss (graphical view)

**.MSS: Microprocessor Software Specification (system.mss)**

The screenshot shows the TI CCS IDE interface. On the left, the Explorer pane shows the project structure with 'platform.spr' highlighted. The Assistant pane shows the project configuration. The main window displays the 'Board Support Package' configuration for 'system.mss'. A red circle highlights the 'Modify BSP Settings...' button and the 'Device drivers' section. A yellow box highlights the 'Device drivers' text.

**Board Support Package**

View current BSP settings, or configure settings like STDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

[Modify BSP Settings...](#) [Reset BSP Sources](#)

A BSP settings file is generated with the user options selected in the settings dialog. To use existing settings, click the below link. This operation clears any existing modifications done. All the subsequent changes are applied on top of the loaded settings.

[Load BSP settings from file](#)

**Operating System**

Name: standalone  
Version: 7.2  
Description: Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.  
Documentation: [standalone v7.2](#)

**Device drivers.**

Name	Driver	Documentation
ps7_coresight_comp_0	coresightps_dcc	<a href="#">Documentation Link</a>
ps7_ddr_0	ddrps	<a href="#">Documentation Link</a>
ps7_ddrc_0	generic	-

# BSP – system.mss (text view)

The image shows a screenshot of an IDE with three main components:

- File Explorer (left):** Shows a project structure. A yellow circle with the number '1' highlights the 'system\_wrapper' folder. A red box highlights the 'system.mss' file, with a yellow circle and the number '2' next to it.
- Assistant (bottom left):** Shows the project hierarchy with 'Zybo\_test\_system [System]' and 'system\_wrapper [Platform]'.
- Code Editor (right):** Shows the content of 'system.mss'. A red box highlights the tab title 'system.mss' with a yellow circle and the number '3' above it. The code is as follows:

```
BEGIN OS # basic embedded OS - „standalone“
PARAMETER OS_NAME = standalone
PARAMETER OS_VER = 7.2
PARAMETER PROC_INSTANCE = ps7_cortexa9_0
    #instance name (arbitrary)
PARAMETER STDIN = ps7_uart_1
PARAMETER STDOUT = ps7_uart_1    #login to a
    serial port
END

.....

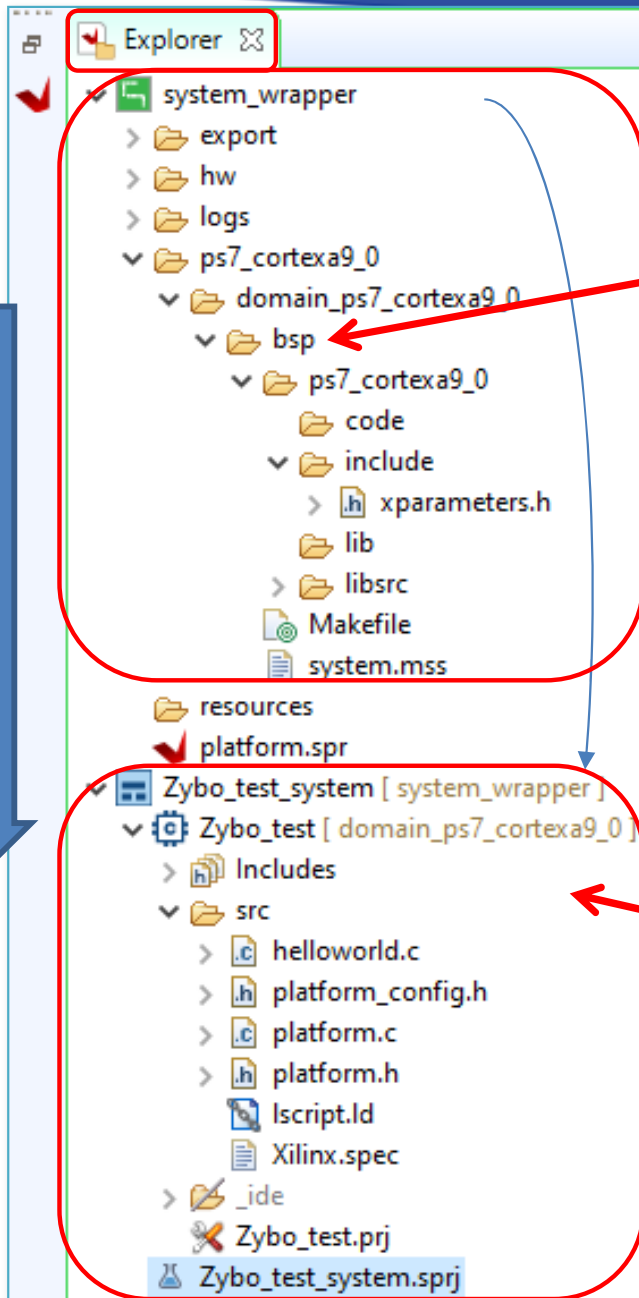
... # Xilinx driver for PS Uart1 IP
BEGIN DRIVER
PARAMETER DRIVER_NAME = uartps #driver name
    (fixed)
PARAMETER DRIVER_VER = 3.9
PARAMETER HW_INSTANCE = ps7_uart_1
END


...


```

A yellow button with the text "Device drivers." is located at the bottom right of the code editor area.

# 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



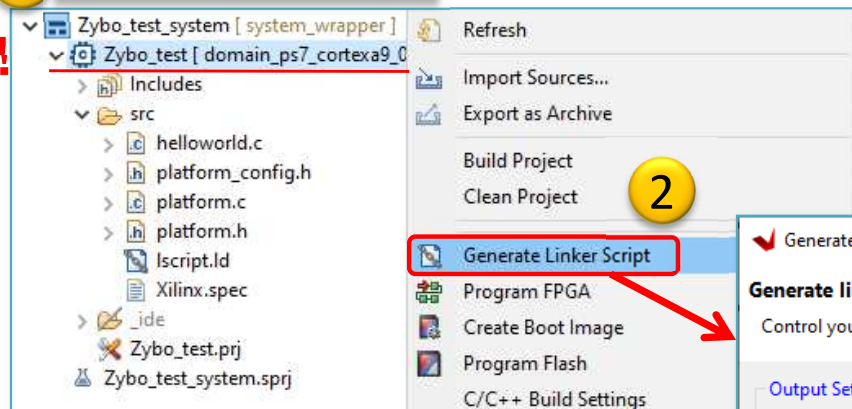
**SW:** zybo\_test (SW application)

- \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)

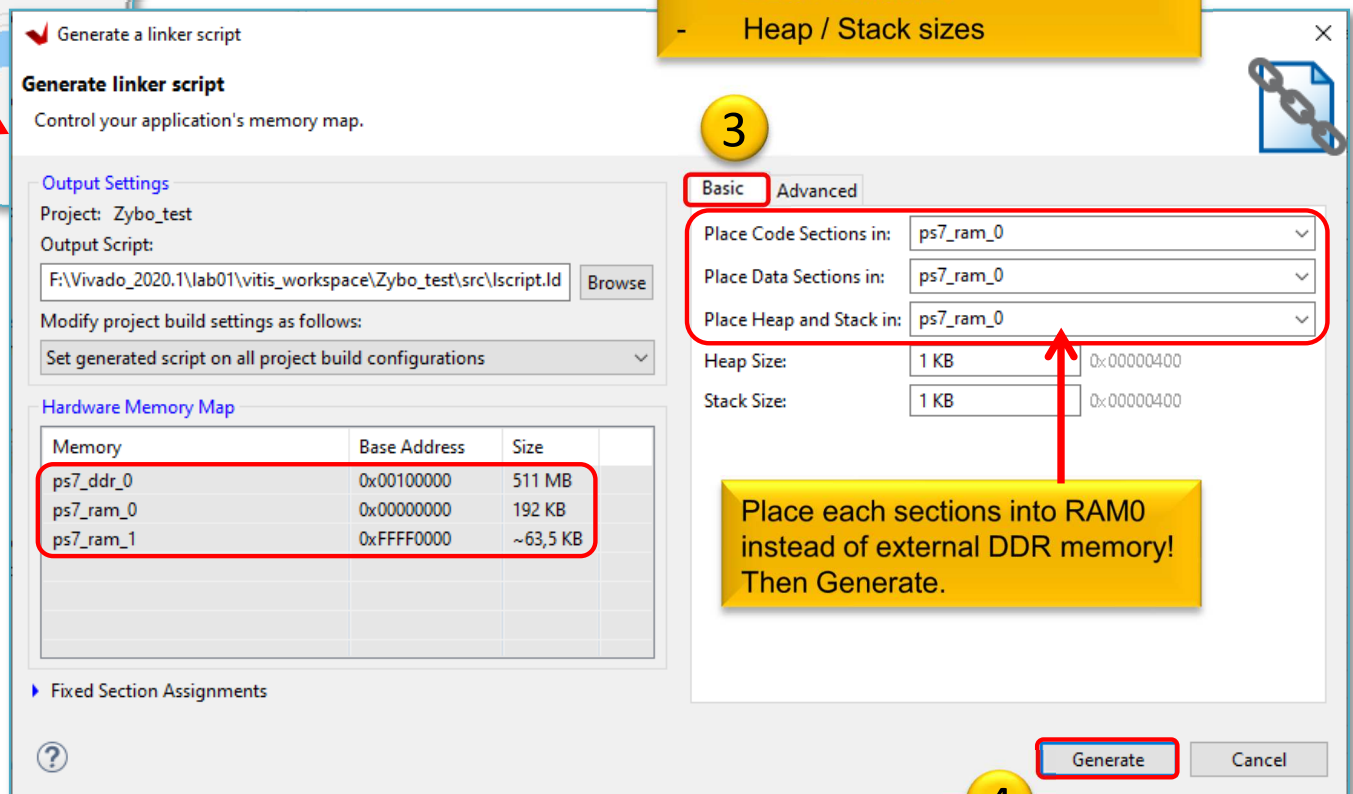
- Xilinx menu → Generate Linker Script (`lscript.ld`)

1 Right click on application.



Choose between Internal RAM0/1 vs. External memory space

- Instructions / Program codes
- Data / Variables
- Heap / Stack sizes



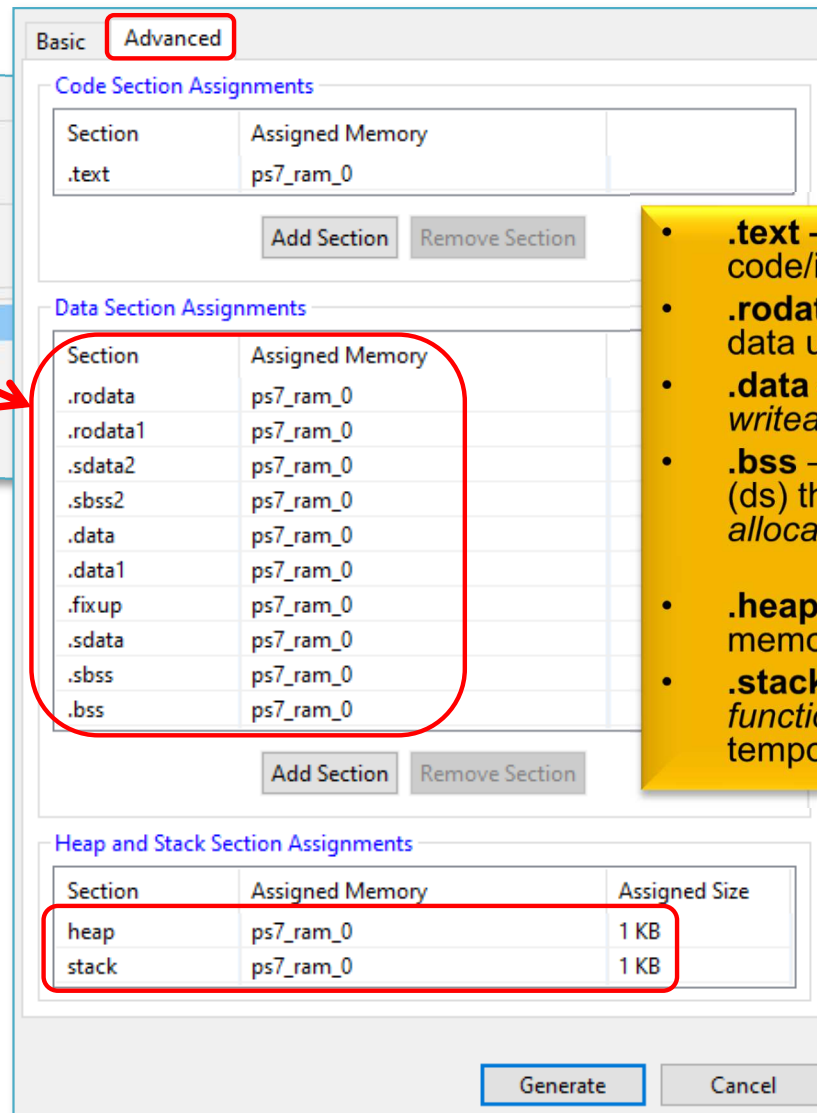
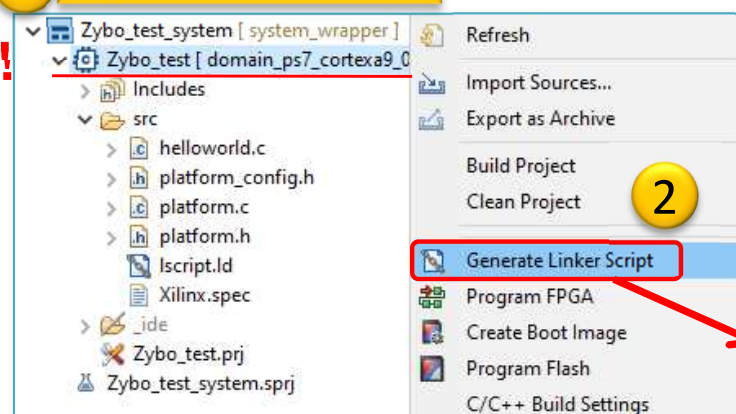
Place each sections into RAM0 instead of external DDR memory! Then Generate.



# Linker Script generation (Advanced)

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

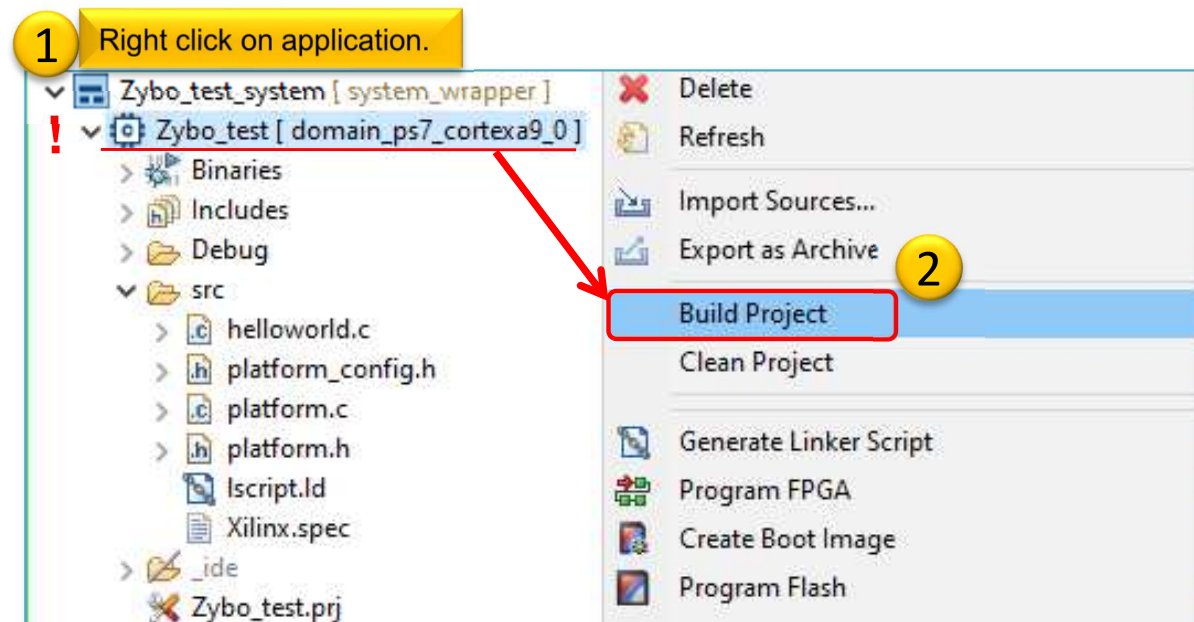
1 Right click on application.



- **.text** — contains „executable code/instruction”
- **.rodata** — contains any “read-only” data used to execute program code
- **.data** — contains „readable-writeable” variables and „pointers”
- **.bss** — a part of a data segment (ds) that contains “statically allocable” variables
- **.heap** — „dinamically allocated” memory
- **.stack** — contains parameters of function call (CALL) and other temporary data

# 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 -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.

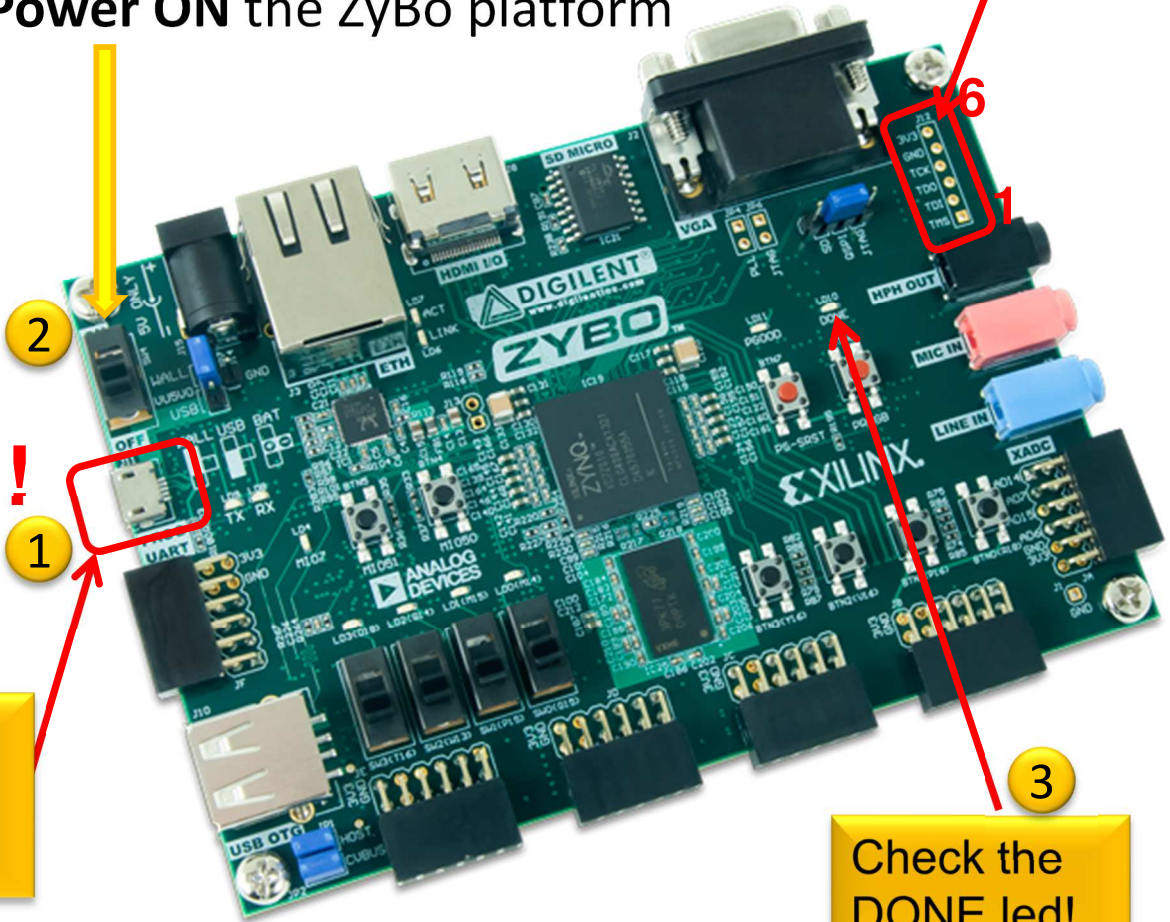
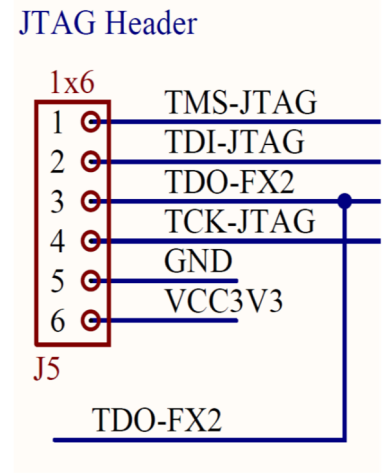
# Embedded system and software test verification

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!)



We use the USB-serial connector.

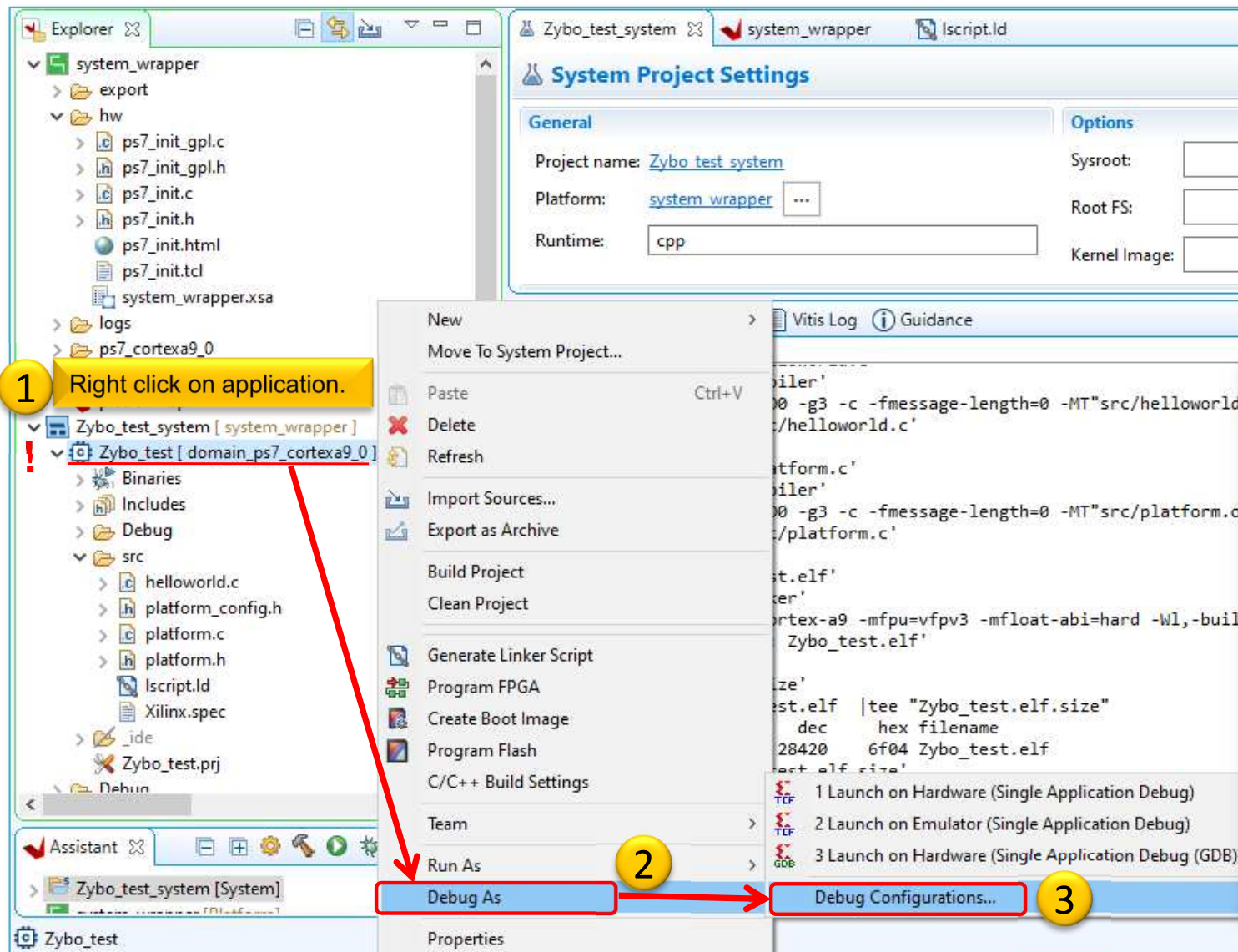
Check the DONE led!

ZyBo – Xilinx USB programming cable

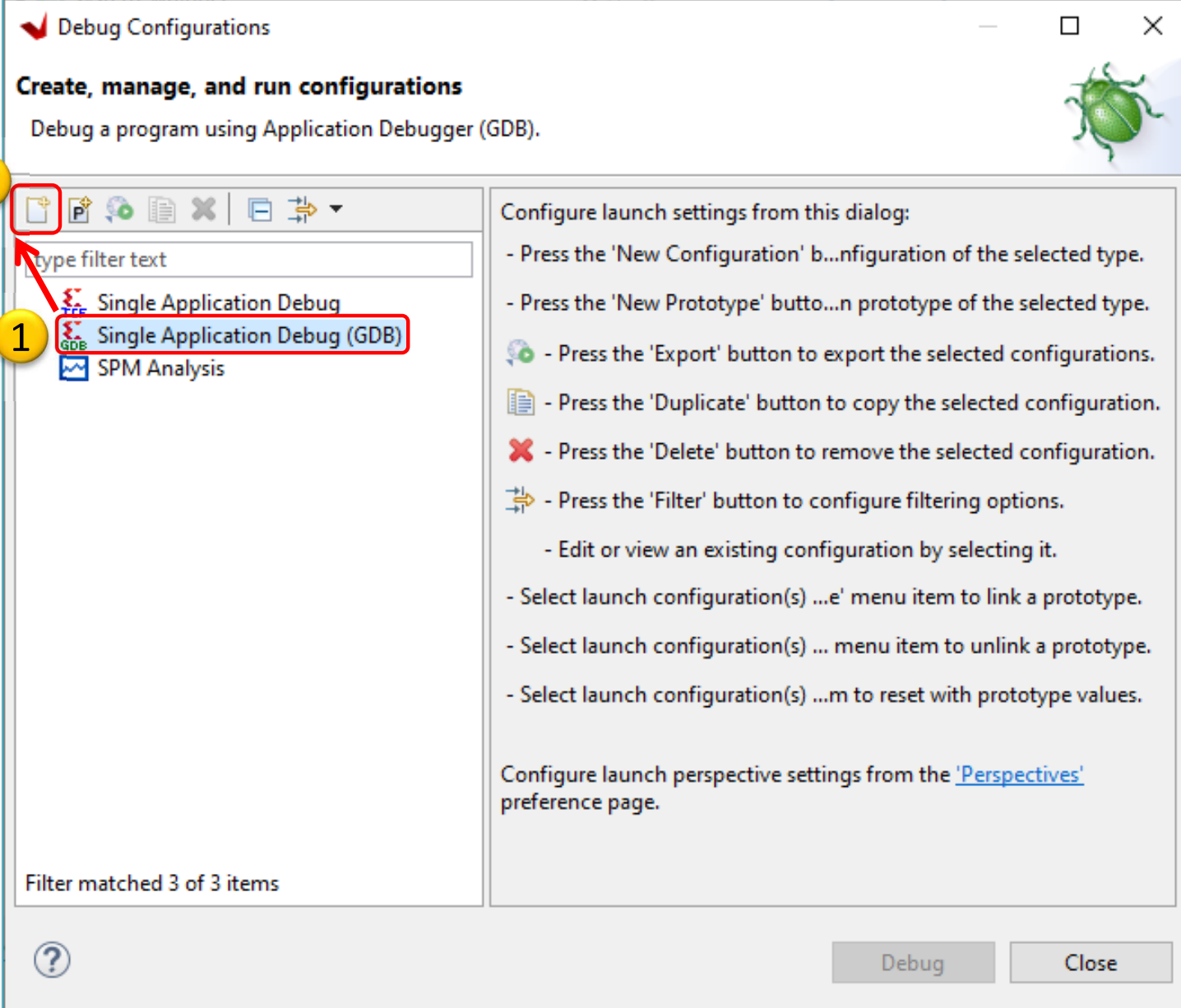
- VCC3V3 – red VREF (6)
- GND – black GND (5)
- TCK-JTAG – yellow TCK (4)
- TDO-FX2 – lilac – TDO (3)
- TDI-JTAG – white TDI (2)
- TMS-JTAG – green TMS (1)

# Creating Debug Configuration

- **Select the application** (Zybo\_test) in the Project Explorer



# Create a new GDB configuration



**Debug Configurations**

Create, manage, and run configurations

Debug a program using Application Debugger (GDB).

2

1

type filter text

- Single Application Debug
- Single Application Debug (GDB)
- SPM Analysis

Filter matched 3 of 3 items

Configure launch settings from this dialog:

- Press the 'New Configuration' button to create a new configuration of the selected type.
- Press the 'New Prototype' button to create a new prototype of the selected type.
- Press the 'Export' button to export the selected configurations.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
  - Edit or view an existing configuration by selecting it.
- Select launch configuration(s) ...e' menu item to link a prototype.
- Select launch configuration(s) ... menu item to unlink a prototype.
- Select launch configuration(s) ...m to reset with prototype values.

Configure launch perspective settings from the ['Perspectives'](#) preference page.

Debug Close

# Create a new GDB configuration (cont)

Debug Configurations

Create, manage, and run configurations

Debug a program using Application Debugger (GDB).

Name: Debugger\_Zybo\_test-GDB

type filter text

- Single Application Debug
- Single Application Debug (GDB)
- Debugger\_Zybo\_test-GDB**
- SPM Analysis

Filter matched 4 of 4 items

Main Application Target Setup Debugger Common

Debug Type: Standalone Application Debug

Connection: Local New

Project: Zybo\_test Browse...

Configuration: Debug

Emulation

Check all GDB settings.

Revert Apply

Debug Close

# Lunching Debugger

1

2

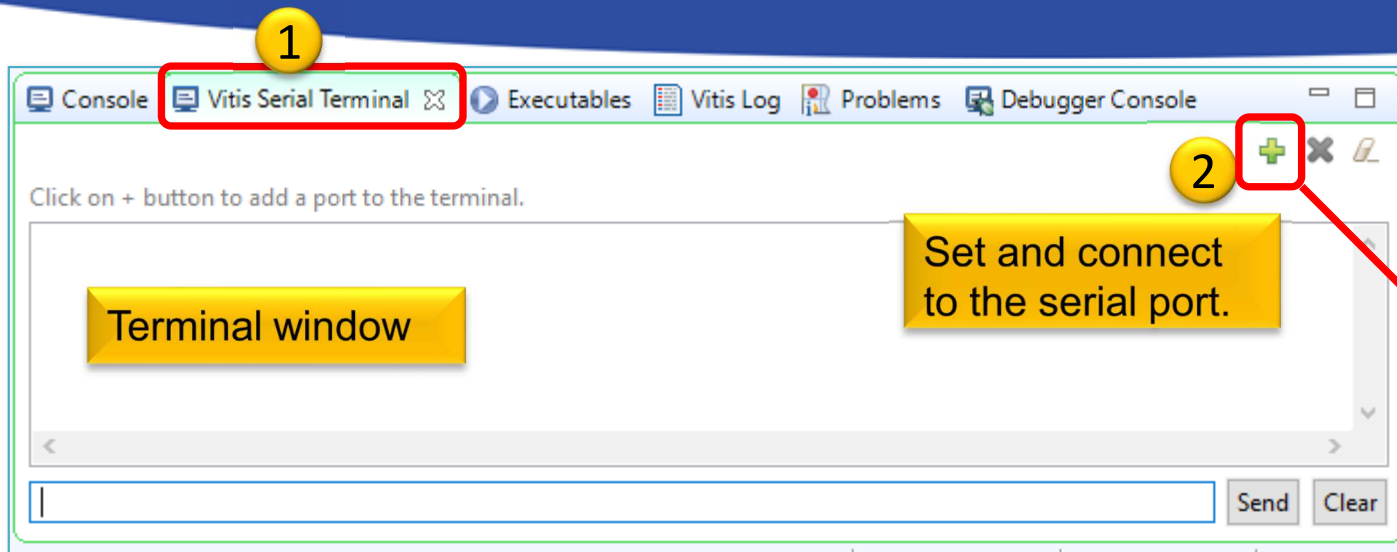
3

Debugger step into this entry point = first instruction of the source code. (helloworld.c)

```
vitis_workspace - Zybo_test/src/helloworld.c - Vitis IDE
File Edit Run Search Xilinx Project Window Help
[Icons]
Debug [X]
Debugger_Zybo_test-GDB [Single Application Debug (GDB)]
  Zybo_test.elf [1]
    Thread #1 1 (ARM Cortex-A9 MPCore #0: Hardware E
      main() at helloworld.c:55 0x58c
      arm-none-eabi-gdb (8.3.0.20190709)
Zybo_test_system system_wrapper lscript.ld helloworld.c [X]
40 *
41 * | UART TYPE  BAUD RATE
42 * -----
43 *   uartns550  9600
44 *   uartlite   Configurable only in HW design
45 *   ps7_uart   115200 (configured by bootrom/bsp)
46 */
47
48 #include <stdio.h>
49 #include "platform.h"
50 #include "xil_printf.h"
51
52
53 int main()
54 {
55   init_platform();
56
57   print("Hello World\n\r");
58   print("Successfully ran Hello World application");
59   cleanup_platform();
60   return 0;
61 }
62
Console [X] Vitis Serial Terminal [X] Executables Vitis Log Problems
Launching Debugger_Zy...t-GDB: (98%)
```



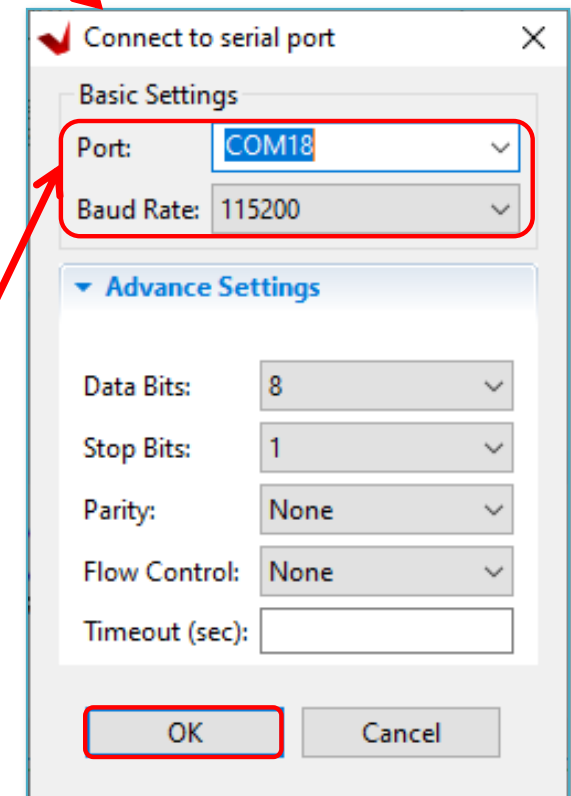
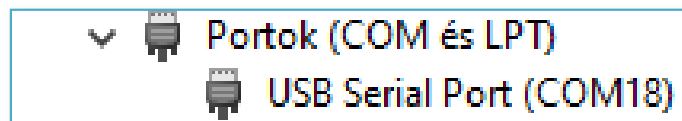
# Set Debug-serial port (VITIS terminal)



Possible ways to login via serial port:

1. **VITIS Serial Terminal: integrated** or
2. using external program: (HyperTerminal, Putty etc.)

- 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

Debug tools

Viewer (variables)

The screenshot displays the Vitis IDE interface for hardware debugging. The main window shows the source code for `helloworld.c` with a breakpoint set at line 59, `cleanup_platform();`. The code includes headers for `stdio.h`, `platform.h`, and `xil_printf.h`, and contains a `main` function that prints "Hello World" and "Successfully ran Hello World application" before calling `cleanup_platform()`.

The **Debug tools** toolbar is highlighted with a red box. The **Viewer (variables)** window is also highlighted with a red box, showing a table with columns for Name, Type, and Value.

The **Breakpoint(s)** window shows a single breakpoint at line 59.

The **ARM/MicroBlaze logging window (VITIS terminal)** shows the following output:

```
Connected to: Serial ( COM18, 115200, 0, 8 )
Connected to COM18 at 115200
Hello World
Successfully ran Hello World application
```

The **XSCt Console** shows the following output:

```
XSCt Process
58: print("Successfully ran Hello World application");
xsct% Info: ARM Cortex-A9 MPCore #0 (t:
58: print("Successfully ran Hello World application");
xsct% Info: ARM Cortex-A9 MPCore #0 (t:
59: cleanup_platform();
xsct% Info: ARM Cortex-A9 MPCore #0 (t:
59: cleanup_platform();
xsct%
xsct%
```

# Terminate Debug process

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

The screenshot shows the Vitis IDE interface. The top toolbar contains various icons for file operations and debugging. The 'Debug' toolbar is active, showing a dropdown menu for the current debug configuration: 'Debugger\_Zybo\_test-GDB [Single Application Debug (GDB)]'. A red box highlights this dropdown, with a yellow callout box labeled '1' pointing to it. The callout box contains the text 'Right click on GDB.' Below the dropdown, the 'Debug Console' shows the execution of 'main() at helloworld.c:61 0x5b0' and the debugger version 'arm-none-eabi-gdb (8.3.0.20190709)'. The 'Explorer' view at the bottom shows the project structure, including 'Zybo\_test\_system' and 'Zybo\_test' sub-projects. The 'Assistant' view is also visible. A right-click context menu is open over the 'Debugger\_Zybo\_test-GDB' entry, with a red box highlighting the 'Terminate and Remove' option. A yellow callout box labeled '2' points to this option. The callout box contains the text 'Stop running the debug configuration and remove it (otherwise it would constantly consume memory!)'. The 'Properties' view is visible at the bottom right of the context menu.

1 Right click on GDB.

2 Stop running the debug configuration and remove it (otherwise it would constantly consume memory!)

# Example I.) Questions

1. Modify and rebuild 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()
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 ?

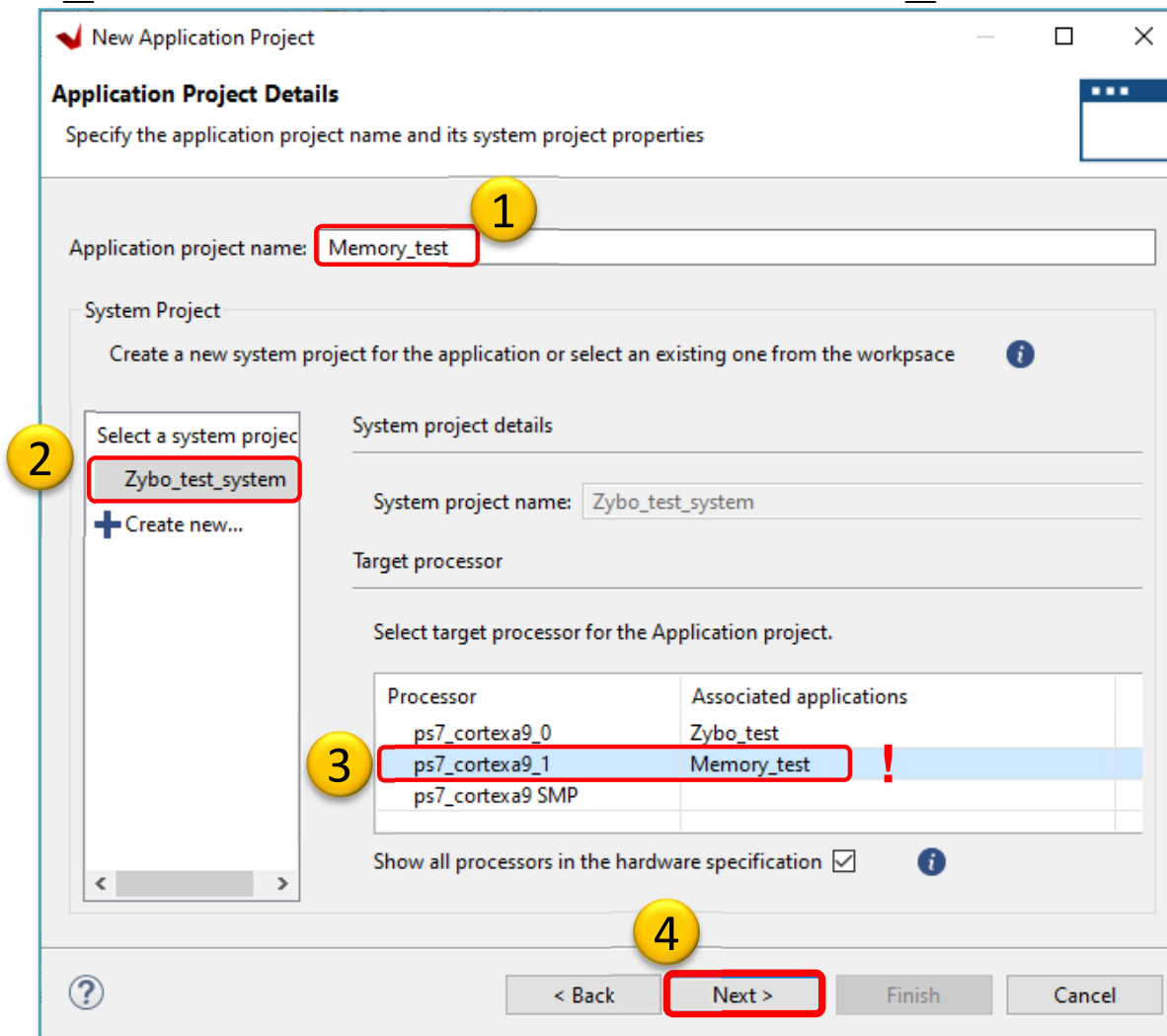
- `xil_printf()` OR `print()`: small memory consumption – FPGA-optimized function
- `printf()` = large memory consumption (non-FPGA optimized, it supports unnecessary formats)

# 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.

New Application Project

**Domain**

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain

+ Create new...

Domain details

Name: domain\_ps7\_cortexa9\_1

Display Name: domain\_ps7\_cortexa9\_1

Operating System: standalone

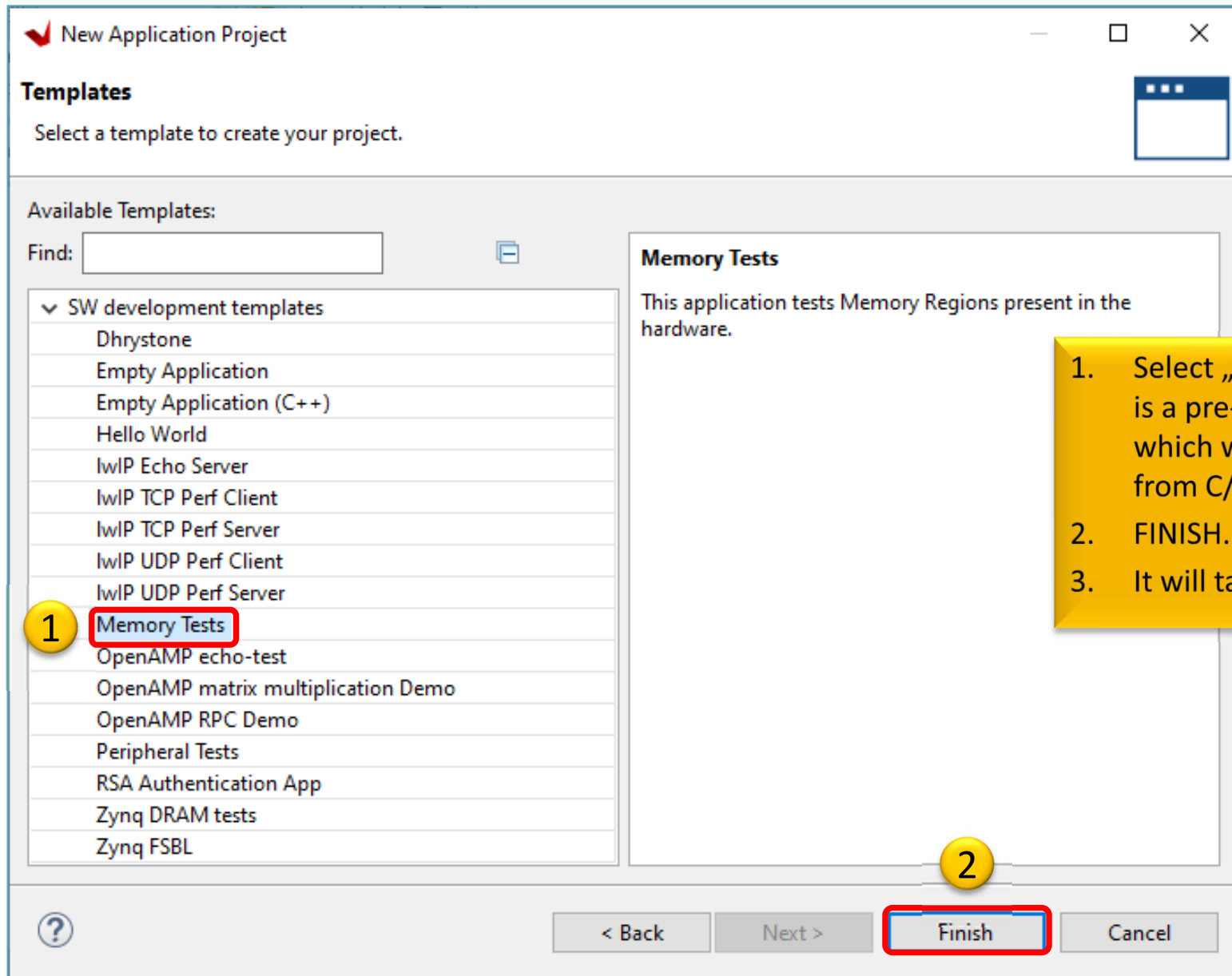
Processor: ps7\_cortexa9\_1

Architecture: 32-bit

< Back Next > Finish Cancel



# 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:

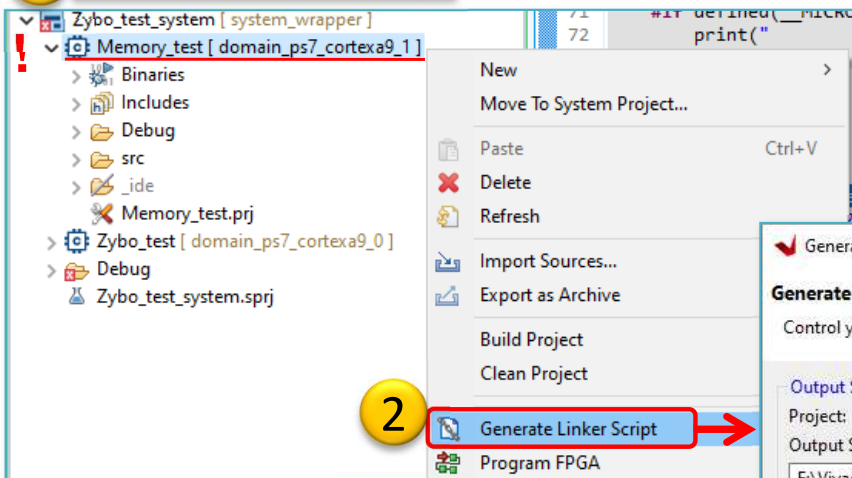
```
<xilinx_install_dir>\VITIS\2020.x\data\embeddedsdsw\lib\bsp\  
standalone_v7_2\src\common\xil_testmem.c
```

**s32 Xil\_TestMem32(u32 \*Addr, u32 Words, u32 Pattern, u8 Subtest);**

# Linker Script generation (Basic)

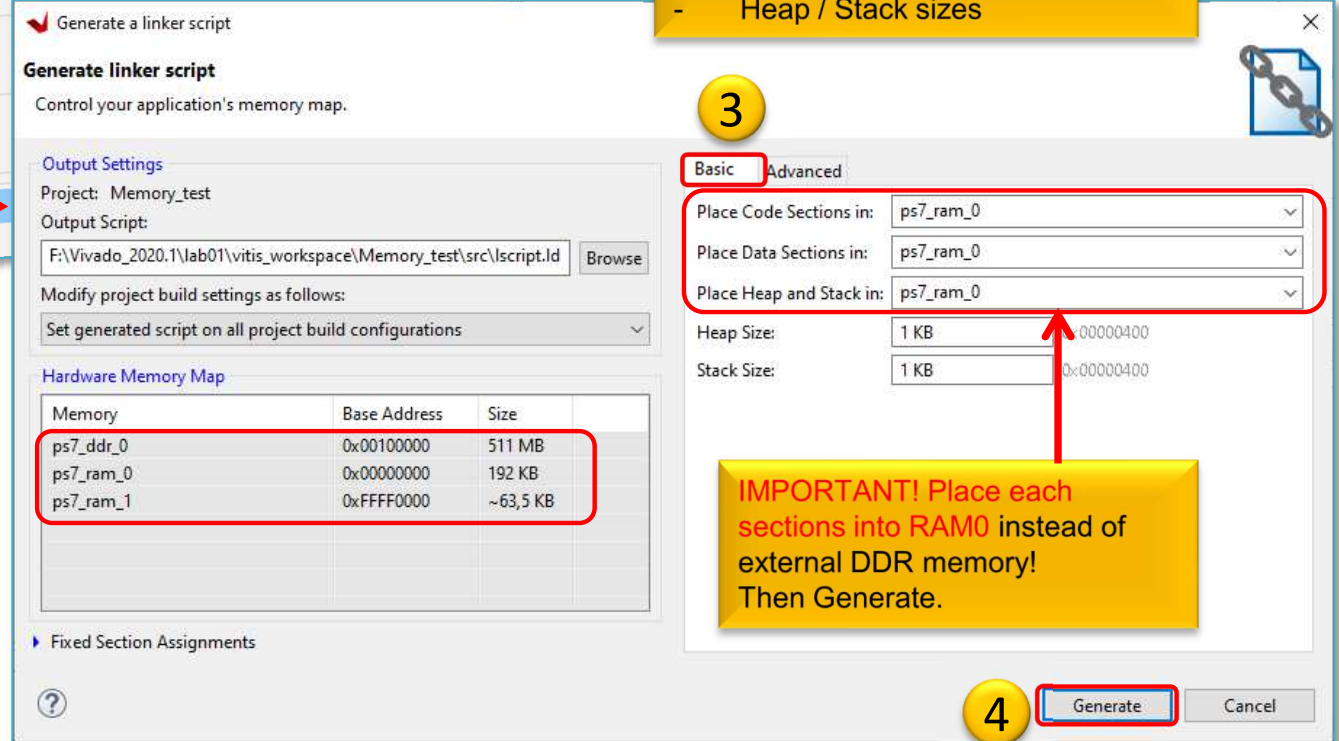
- Xilinx menu → Generate Linker Script (`lscript.ld`) - **RAM0**

1 Right click on application.



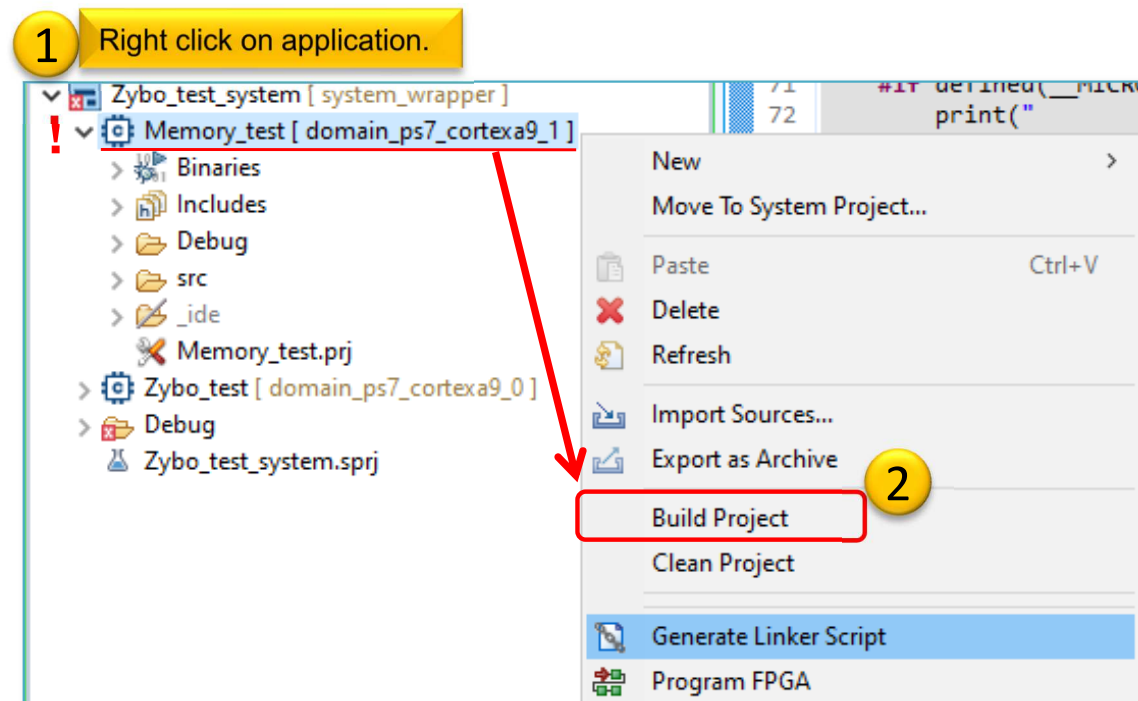
Choose between Internal RAM0/1 vs. External memory space

- Instructions / Program codes
- Data / Variables
- Heap / Stack sizes



# 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_dds_0
Memory Controller: ps7_dds_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 II.) Modify Memory Test

- Modify the memory test application so that
  - it tests a larger external `ps7_dds_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 II.) 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 II.) Question & Answer


- What is the size of the modified Memory\_test application?
  - ~37 Kbyte

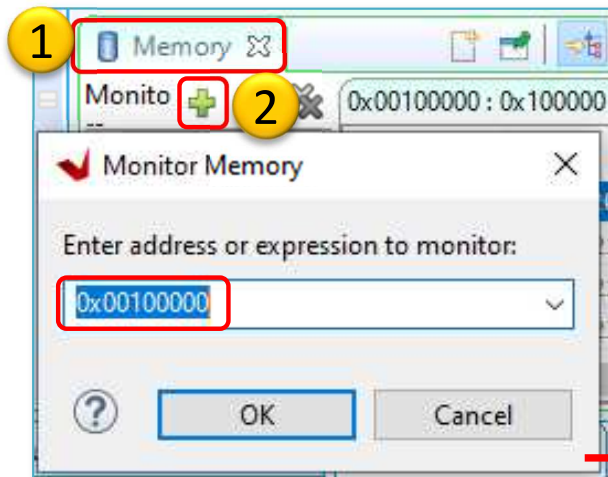
```
'Invoking: ARM v7 Print Size'  
arm-none-eabi-size Memory_test.elf |tee "Memory_test.elf.size"  
  text    data    bss     dec     hex filename  
 27960   1184   8248   37392   9210 Memory_test.elf  
'Finished building: Memory_test.elf.size'
```

# Example II.) Build and Debug

- 
1. Build the modified Memory Test application (.elf)
  2. Launch 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)



Address	0 - 3	4 - 7	8 - B	C - F
00100000	00000000	00000000	00000000	00000000
00100010	0x100000	00000000	00000000	00000000
00100020	00000000	00000000	00000000	00000000

Address	0 - 3	4 - 7	8 - B	C - F
00100000	01020304	05060708	090A0B0C	0D0E0F10
00100010	11121314	15161718	191A1B1C	1D1E1F20

# 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:

```
#include "xtime_1.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);

    ...
}
```

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.



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének  
együttes javítása a Pannon Egyetemen

# THANK YOU FOR YOUR KIND ATTENTION!

**SZÉCHENYI** 2020



MAGYARORSZÁG  
KORMÁNYA

**Európai Unió**  
Európai Strukturális  
és Beruházási Alapok



**BEFEKTETÉS A JÖVŐBE**