Course:
Advanced Digital System Design

Professor: HyungWon Kim

TUTORIAL:
Software/Hardware Co-design Using Xilinx Zynq SoC

TAs: Hossam Hassan and Saad Arslan

Email: {hossam,saad}@cbnu.ac.kr

Office: E10-516

Lab Objectives

- **Understand What and Why Software/Hardware Co-design**
- **Understand the Xilinx Zynq SoC SoC development flow**
- **Utilize the Xilinx embedded systems tools to**
  - Design a Zynq AP SoC System
  - Develop Software Applications

Xilinx Embedded Tool Flow

- PlanAhead
  - Central Design Cockpit
  - HDL Development
  - Design Implementation
- Xilinx Platform Studio (XPS)
  - Processor Configuration and Customization
- Software Development Kit (SDK)
  - C/C++ Integrated Development Environment

LAB Contents

- Getting Started with Software/Hardware Co-design using PlanAhead and Zynq
  - Zynq Quick-Start Tutorial
    - Part 1: Setting up a new project
    - Part 2: Setting up Zynq Processor
    - Part 3: Setting up Top Level HDL
    - Part 4: Setting Software

Let’s Start Our Journey

Before Starting

If you have ANY question

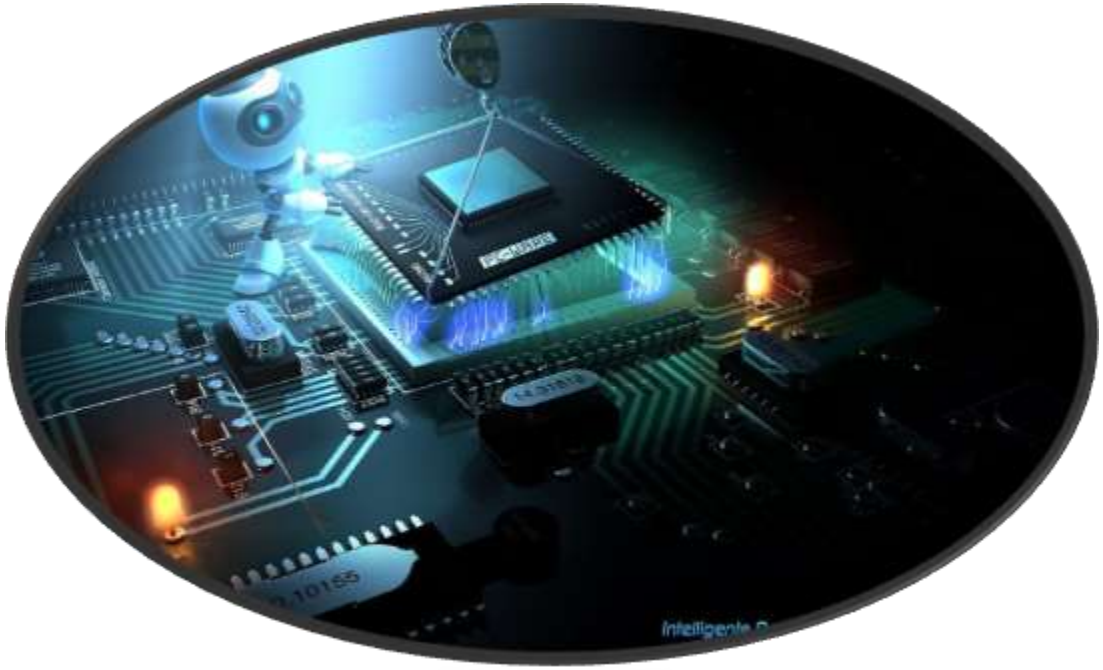
Please write it on your notebook.

By the end of the presentation YOU can ASK w hatever you want.

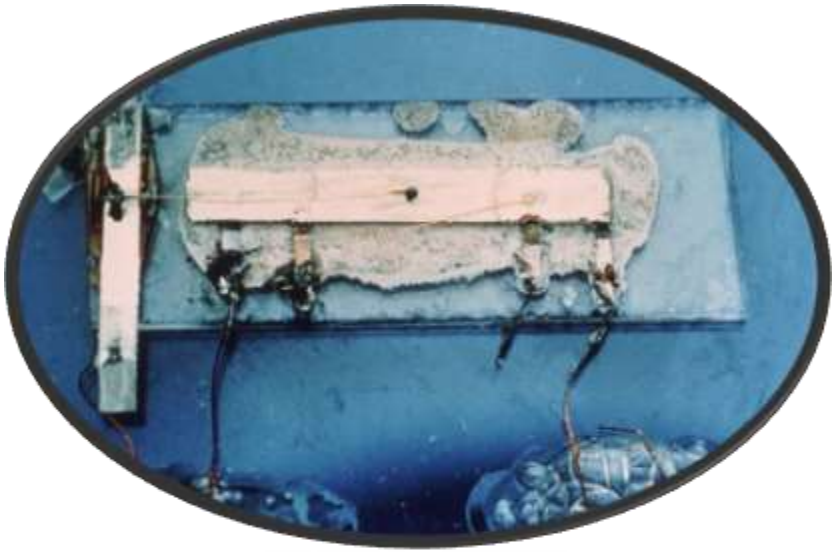
Maybe the answer for your question will be in one of the following slides.

Thank you for Your understanding

Let’s Enjoy Our Journey



Some History and Insight about Integrated Circuits



The first working integrated circuit was created by Jack Kilby in 1958.
It contains a single transistor

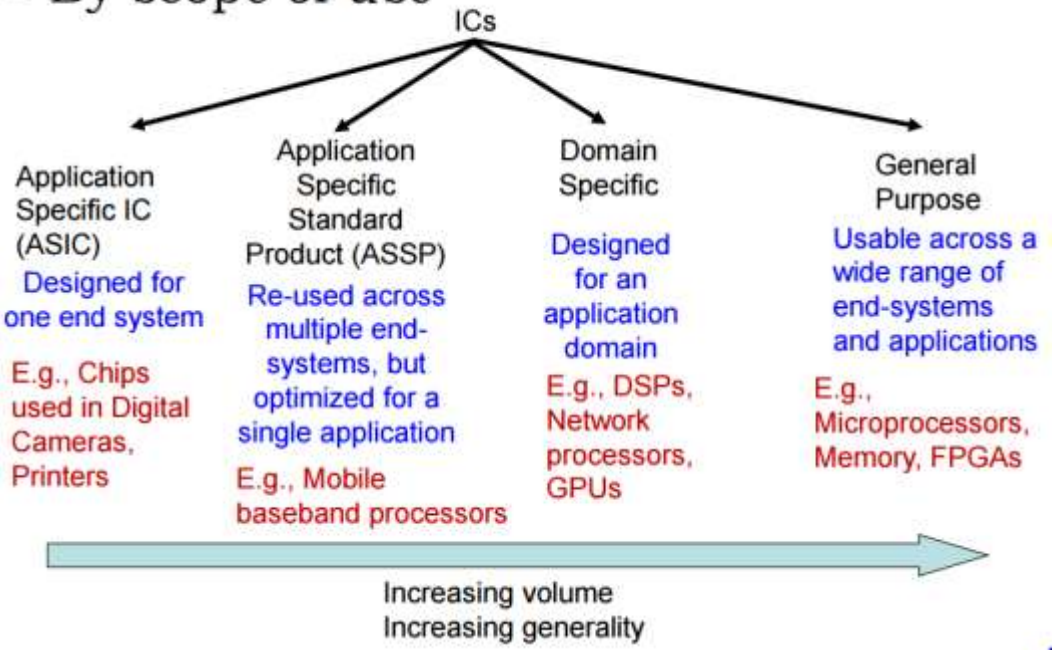
What is Integrated Circuit?

- It is a set of electronic circuits on one small flat piece (or "chip ") of semiconductor material, normally silicon. [wikipedia]

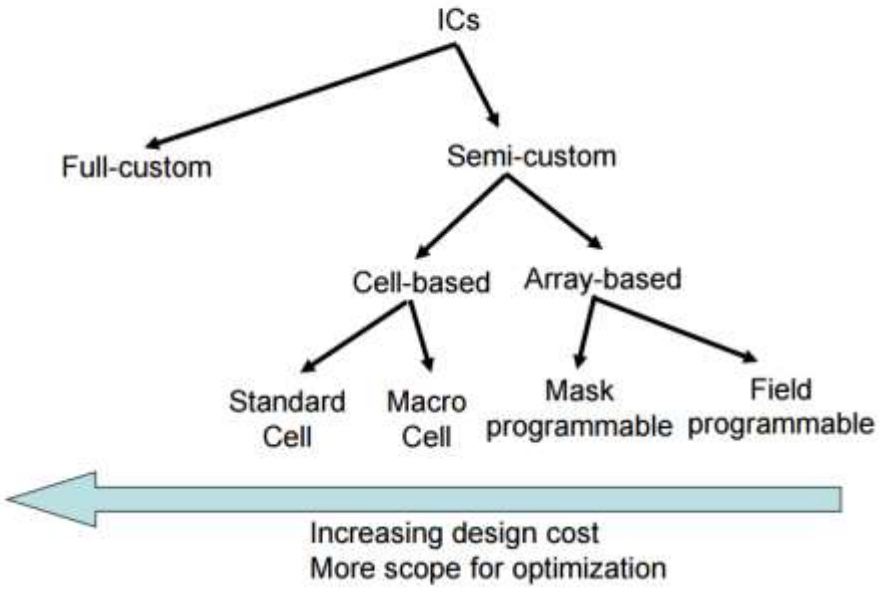
Name	Signification	Year	Transistors number	Logic gates number
SSI	*small-scale integration*	1964	1 to 10	1 to 12
MSI	*medium-scale integration*	1968	10 to 500	13 to 99
LSI	*large-scale integration*	1971	500 to 20,000	100 to 9,999
VLSI	*very large-scale integration*	1980	20,000 to 1,000,000	10,000 to 99,999
ULSI	*ultra-large-scale integration*	1984	1,000,000 and more	100,000 and more

Taxonomy of Integrated Circuits

- By scope of use



- By degree of design customization



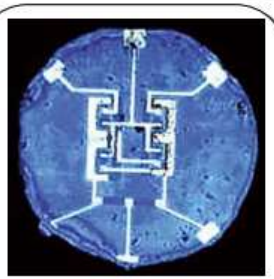
ECE 695R: System-on-Chip Design

Evolution of IC Design Abstraction

1 Transistor
3 Resistors
1 Capacitor

Germanium IC
Jack Kilby
Texas Instruments

From

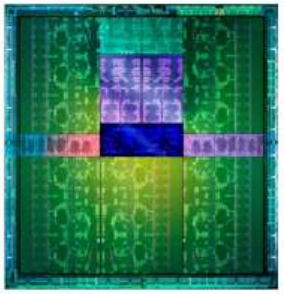


Silicon IC
Robert Noyce
Fairchild
Semiconductor

circa 1958

To

Intel Poulson (8-core microprocessor)
3.1 Billion transistors
(32 nm)



NVIDIA GK110 Kepler
(Graphics processor with 2880 "cores")
7.1 Billion transistors
(28 nm)

circa 2013

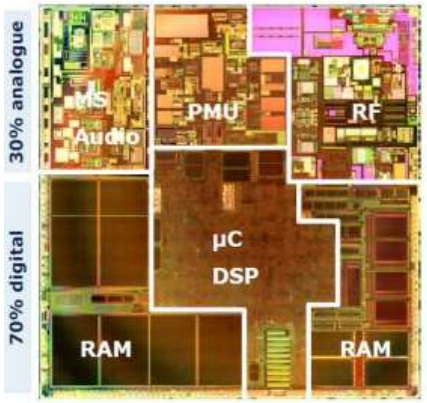
A 3,000,000,000 fold increase in 55 years

ECE 695R: System-on-Chip Design

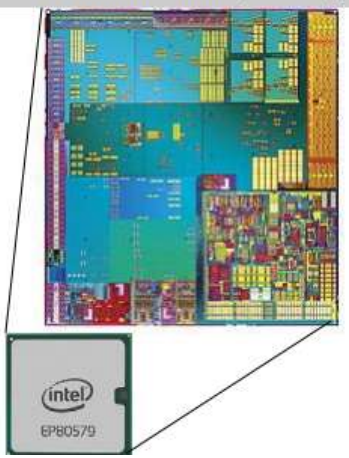
What are Systems-on-chip?

- Direct consequence of increasing scales of integration (Moore's Law)
  - Integrate all or most system components into a single chip
  - Benefits: cost, size, power consumption, performance

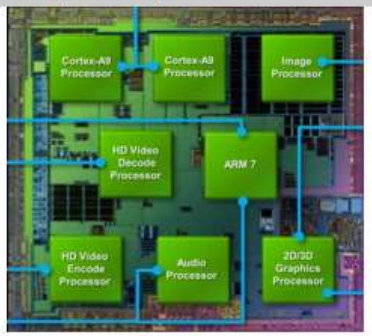
Infineon E-GOLDVoice "Phone-on-a-chip"



Intel EP80579 SoC for embedded systems (security, communications, storage)



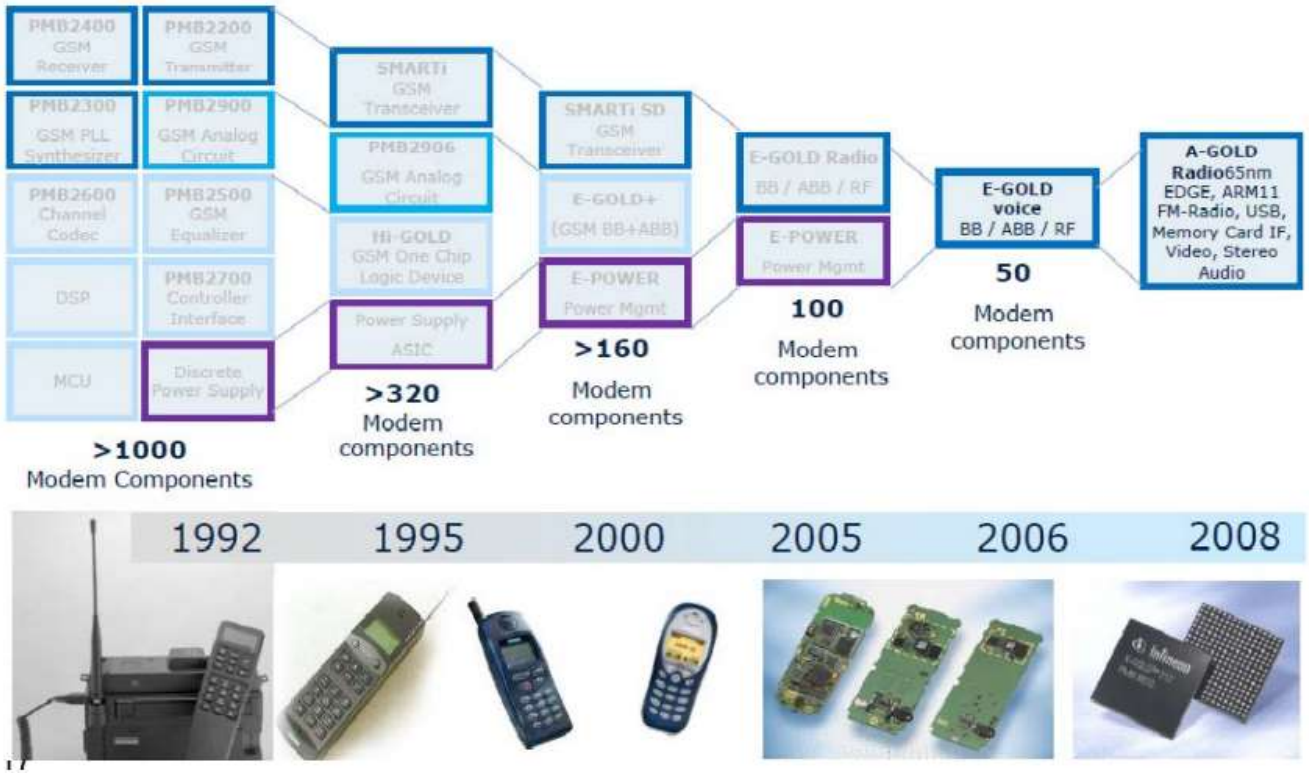
NVIDIA Tegra2 SoC for tablets, smartphones, cars, set-top boxes



ECE 695R: System-on-Chip Design

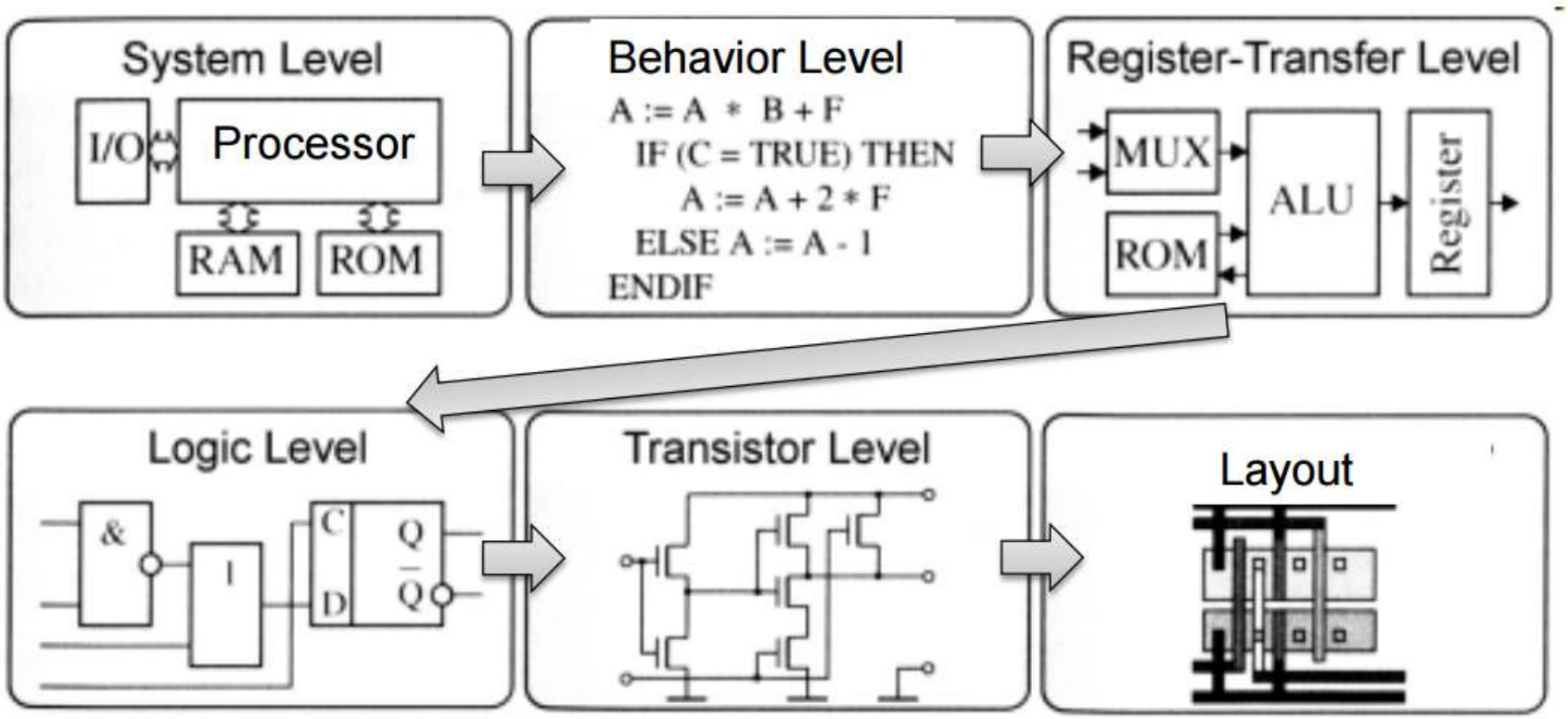
Benefits of SoC Integration: Size

- Impact of SoCs on mobile phones



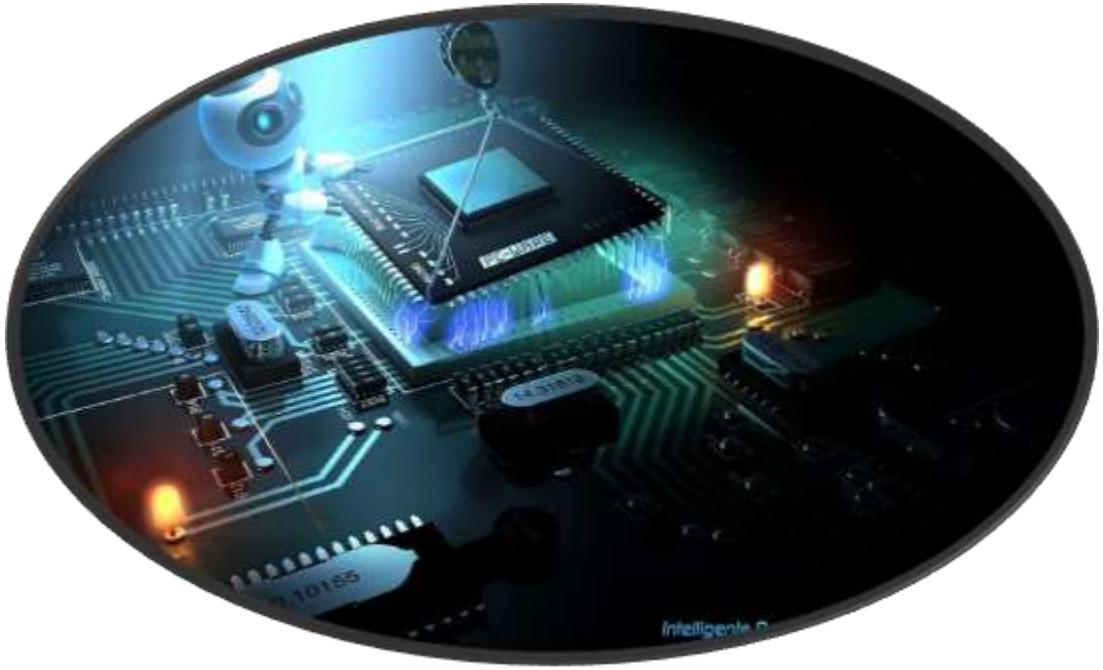
- "Single-chip" cell phone

IC Design : Levels of Abstraction



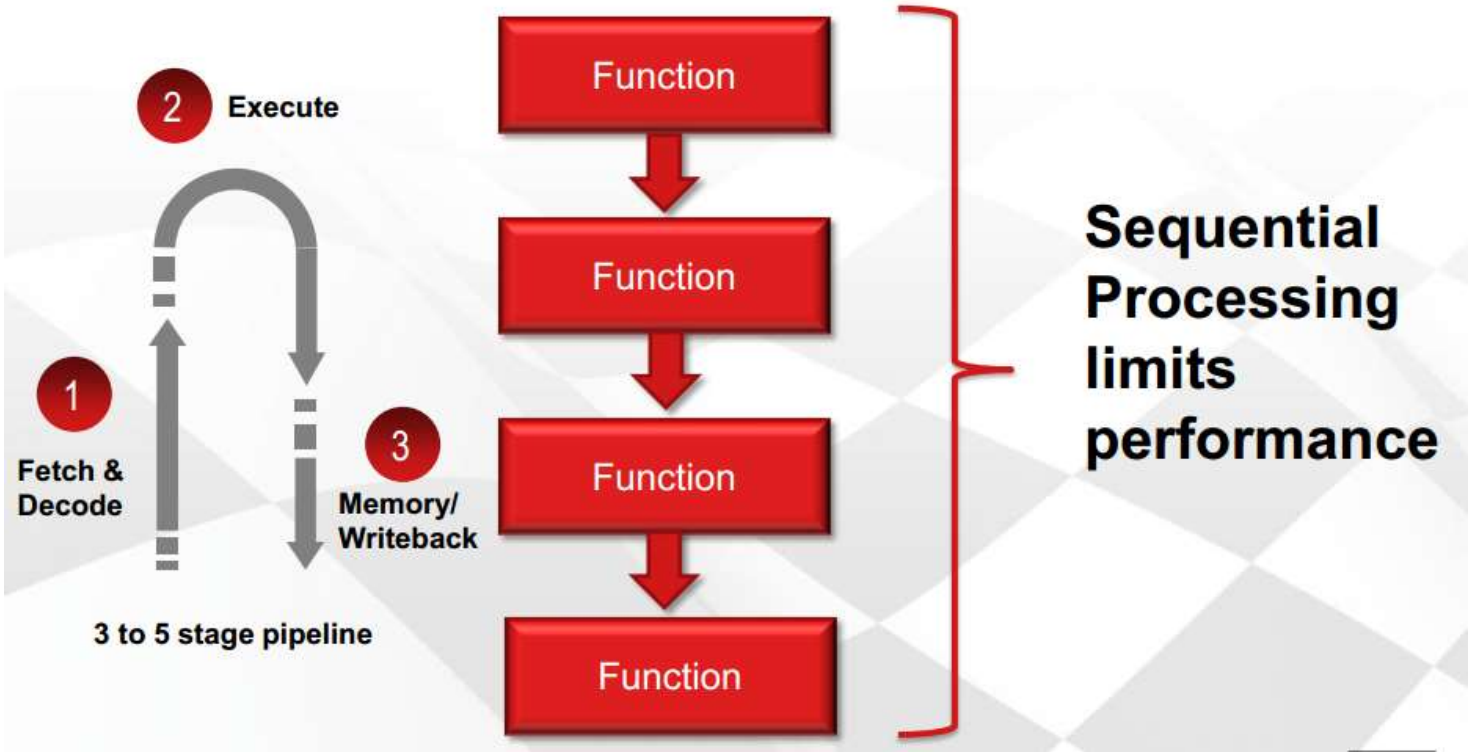
ECE 695R: System-on-Chip Design

Background



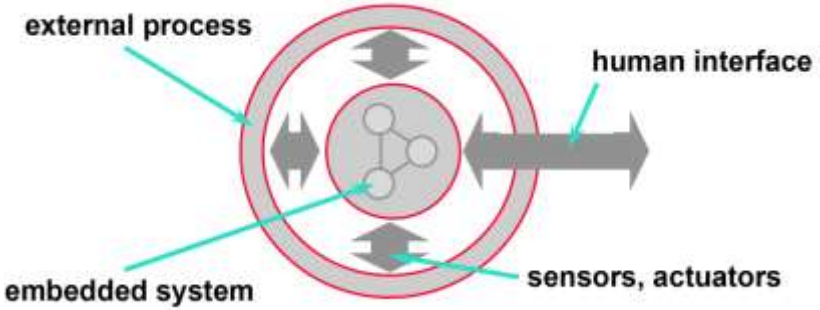
Sequential World

Software Engineers: Stuck in a Sequential World



Embedded System

Idea of an Embedded System



Software vs. Hardware Trade-offs

Improve Performance
Improve Energy Efficiency
Reduce Power Density

Implement more in Hardware

Manage Design Complexity
Reduce Design Cost
Stick to Design Schedule
Handle Deep Submicron

Implement more in Software

Embedded Systems vs. General-Purpose Computing

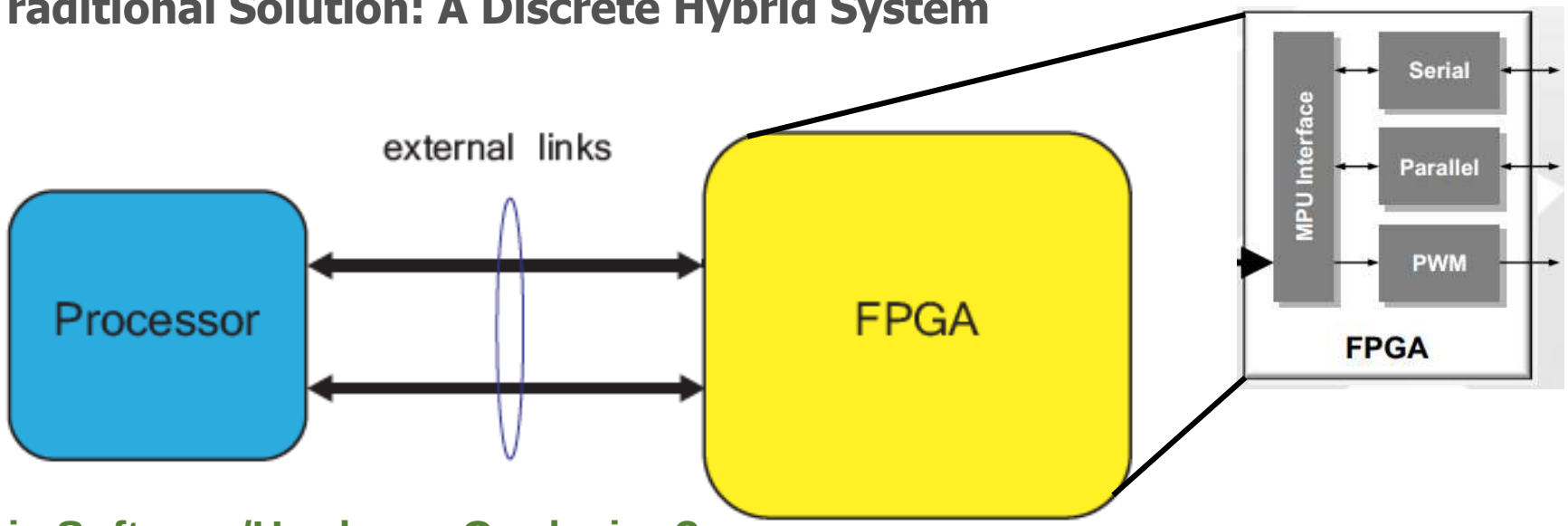
- Embedded Systems
  - Few applications that are known at design-time.
  - Not programmable by end user.
  - Fixed run-time requirements (additional computing power not useful).
  - Criteria:
    - cost
    - power consumption
    - predictability
    - meeting time bounds
- General Purpose Computing
  - Broad class of applications.
  - Programmable by end user.
  - Faster is better.
  - Criteria:
    - cost
    - average speed

Hardware and software developed separately in past

Co-Design reduces number of prototypes and time-to-market

Analysis of HW/SW boundaries and interfaces

The Traditional Solution: A Discrete Hybrid System

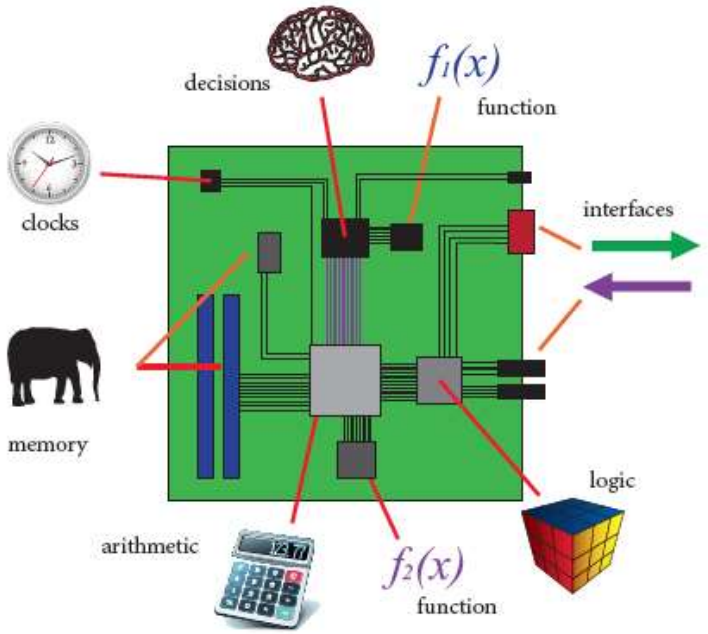


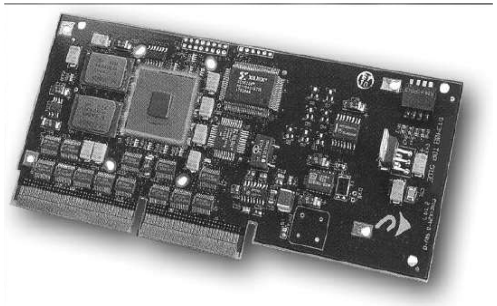
What is Software/Hardware Co-design?

Integrated design of systems that consist of hardware and software components

System-on-a-Board Vs System-on-Chip (SoC)

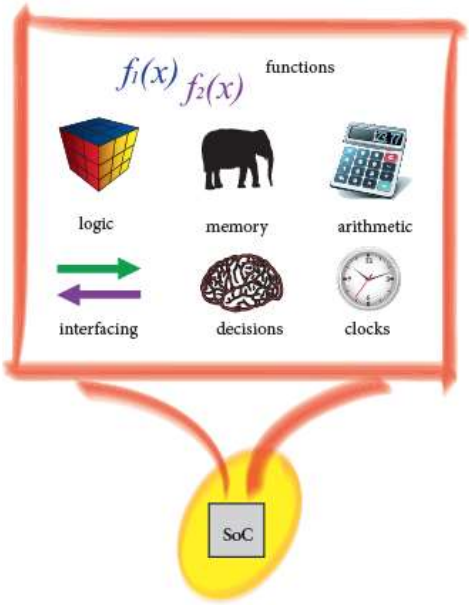
System-on-a-Board





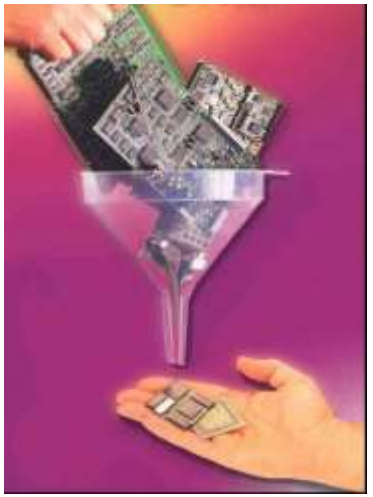
Vs.

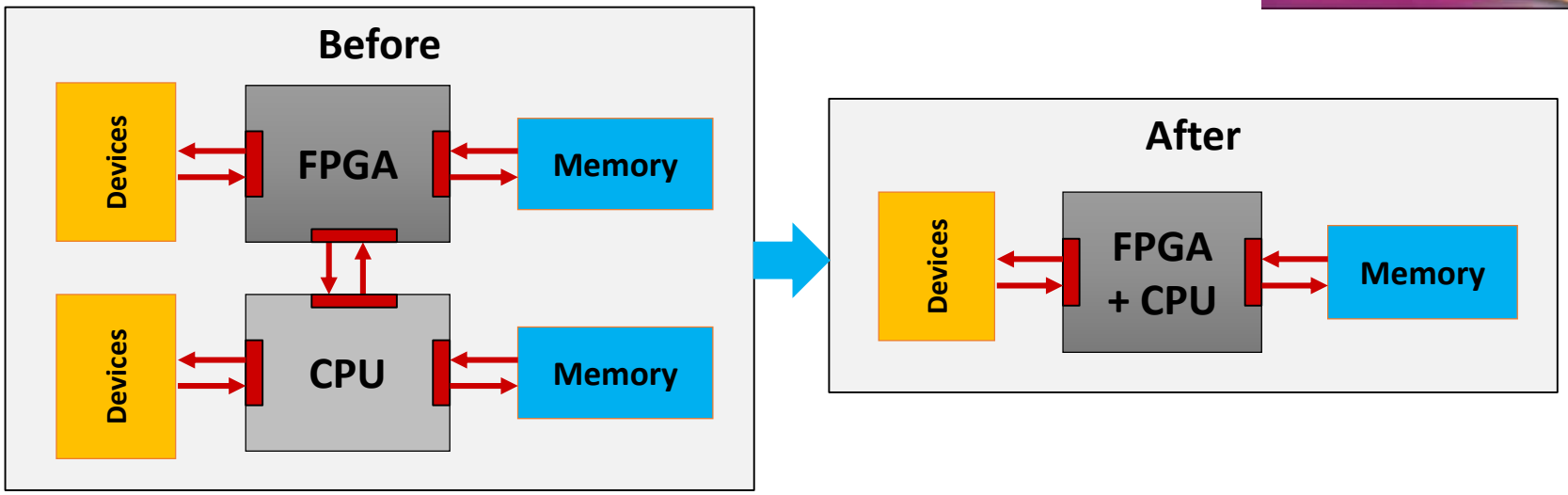
System-on-Chip (SoC)



System-on-Chip (SoC)

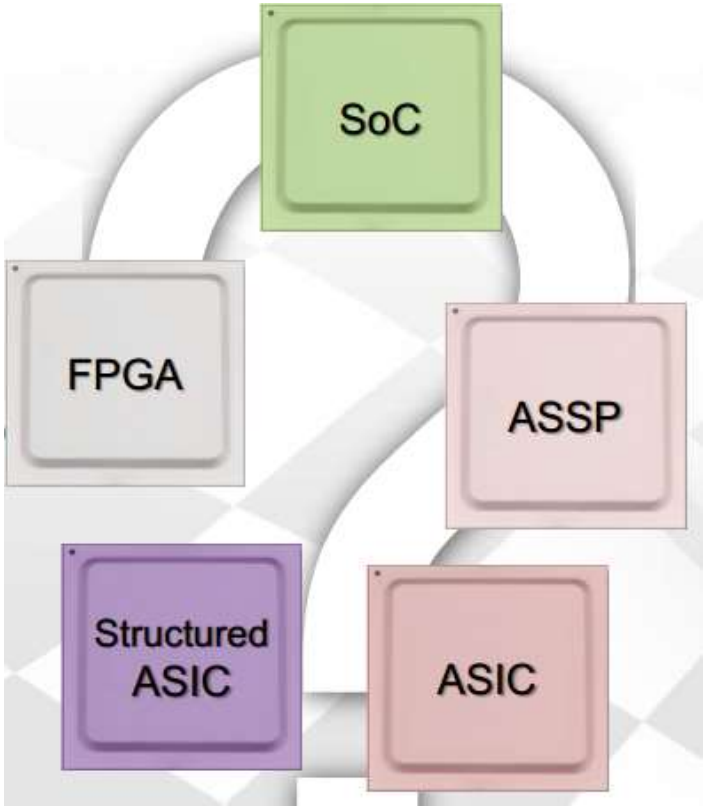
SoC FPGAs integrate ARM-based processor system with FPGA fabric.
Benefits are higher performance, lower power, lo wer cost, and smaller board space of a single chip s olution.



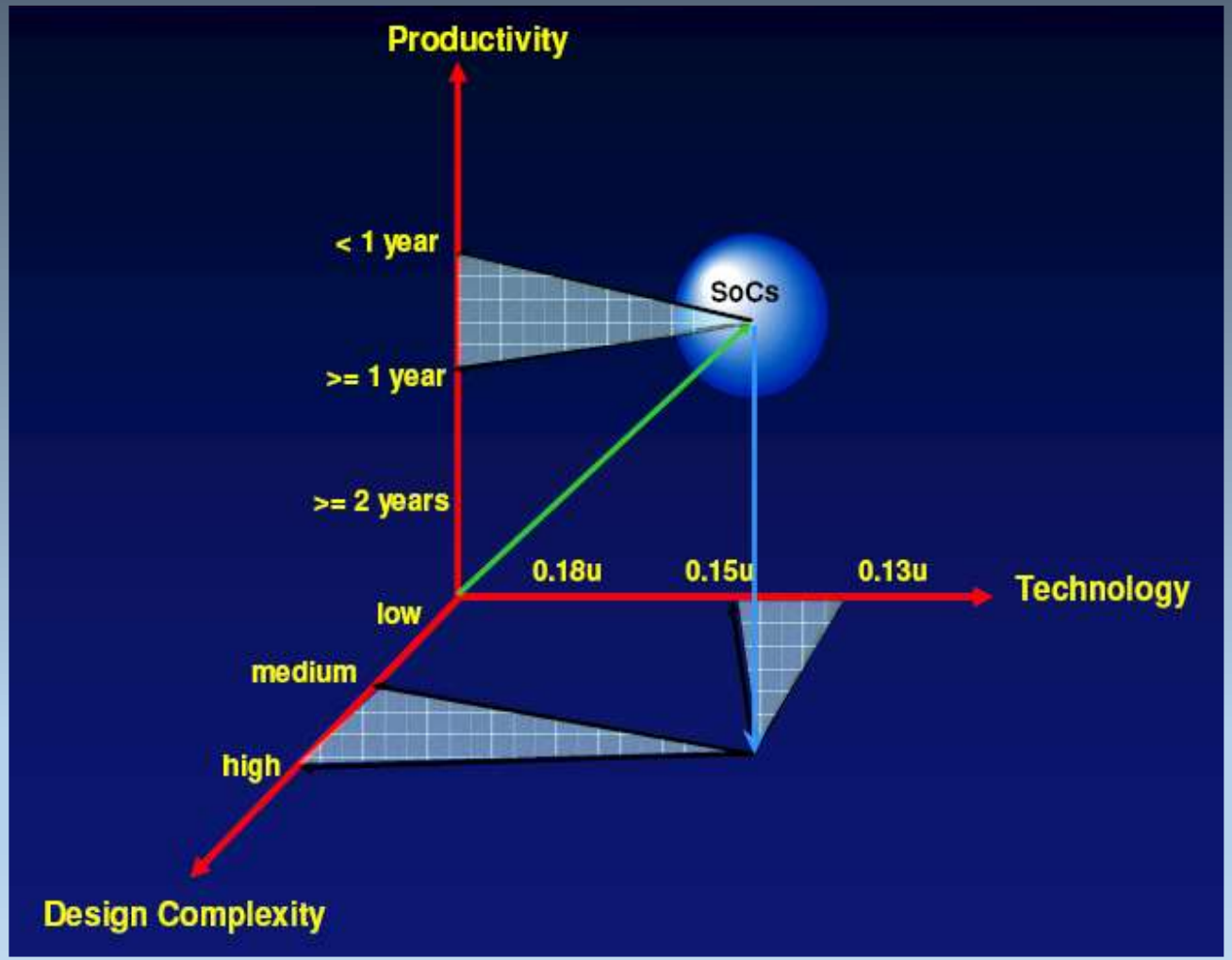


Demands of Today's Technology

- **Which Technology Should I Choose?**



Technology vs. Productivity vs. Complexity



Processors inside FPGAs?

Alternative Solutions

Xilinx Zynq
Zynq-7000 All Programmable SoCs with Cortex-A9 MPCore

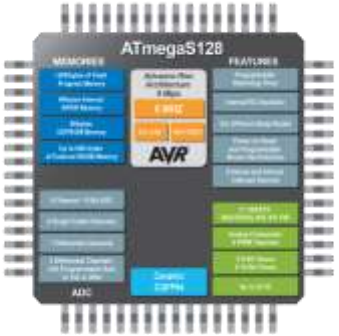
Altera Arria V & Cyclone V
Hard processor system (HPS) with Cortex-A9 MPCore

Microsemi Smartfusion2
Cortex M3

FPGAs inside Processors?

Cypress Semiconductor PSoC® 5 programmable embedded System-on-Chip solutions based on the ARM® Cortex®-M processor, high-performance programmable analog blocks, PLD-based programmable digital blocks, programmable interconnect and routing, and CapSense®.

Alternative Solutions



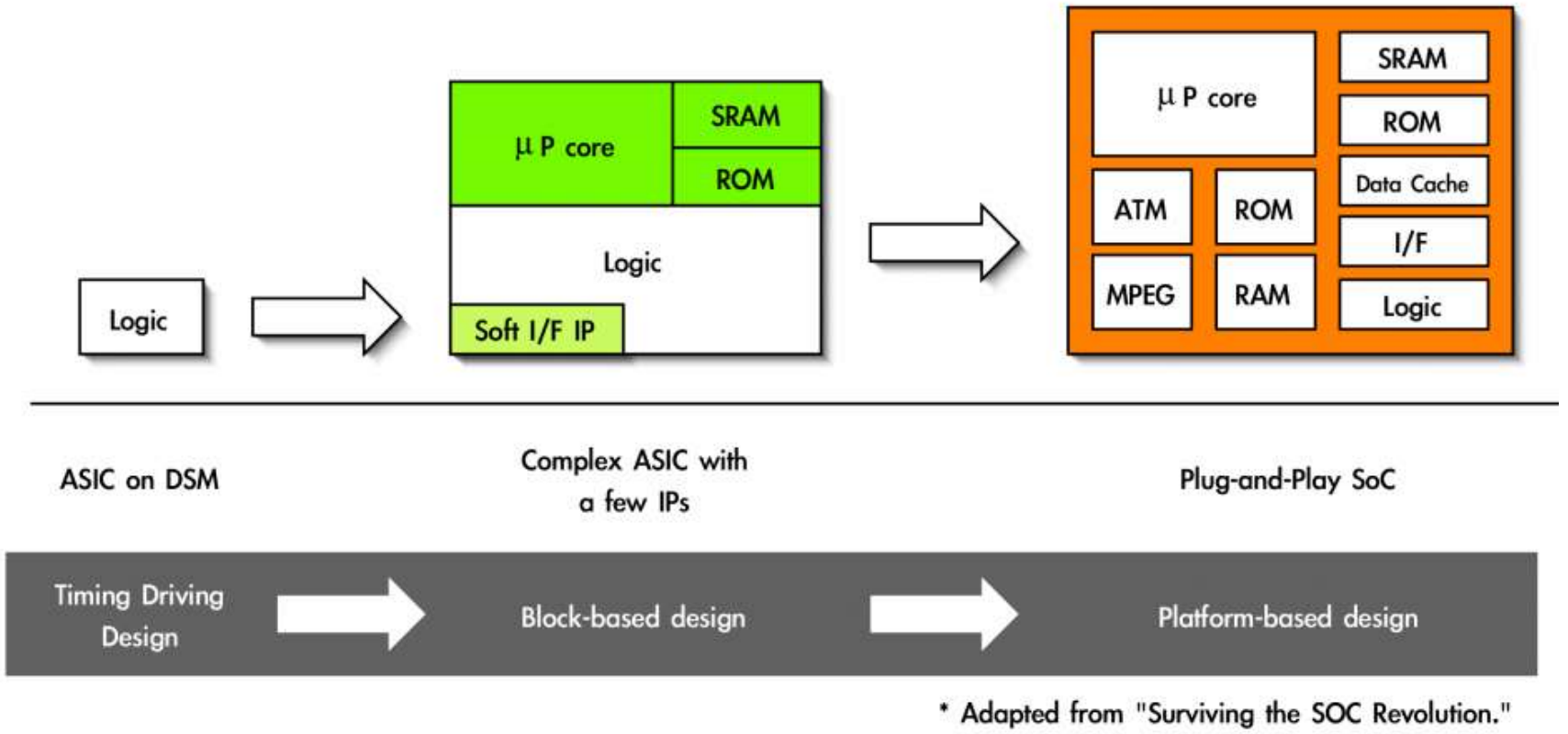
The Microchip AT40KAL Series co-processor FPGAs range from 5K to 50K usable gates and are designed for high-density, compute-intensive DSP and other fast logic designs .

An SoC …

- usually contains
  - reusable IP
  - embedded processor, memory
  - real-world interface
  - mixed-signal blocks
  - programmable hardware

So, What is the SoC Design Flow?

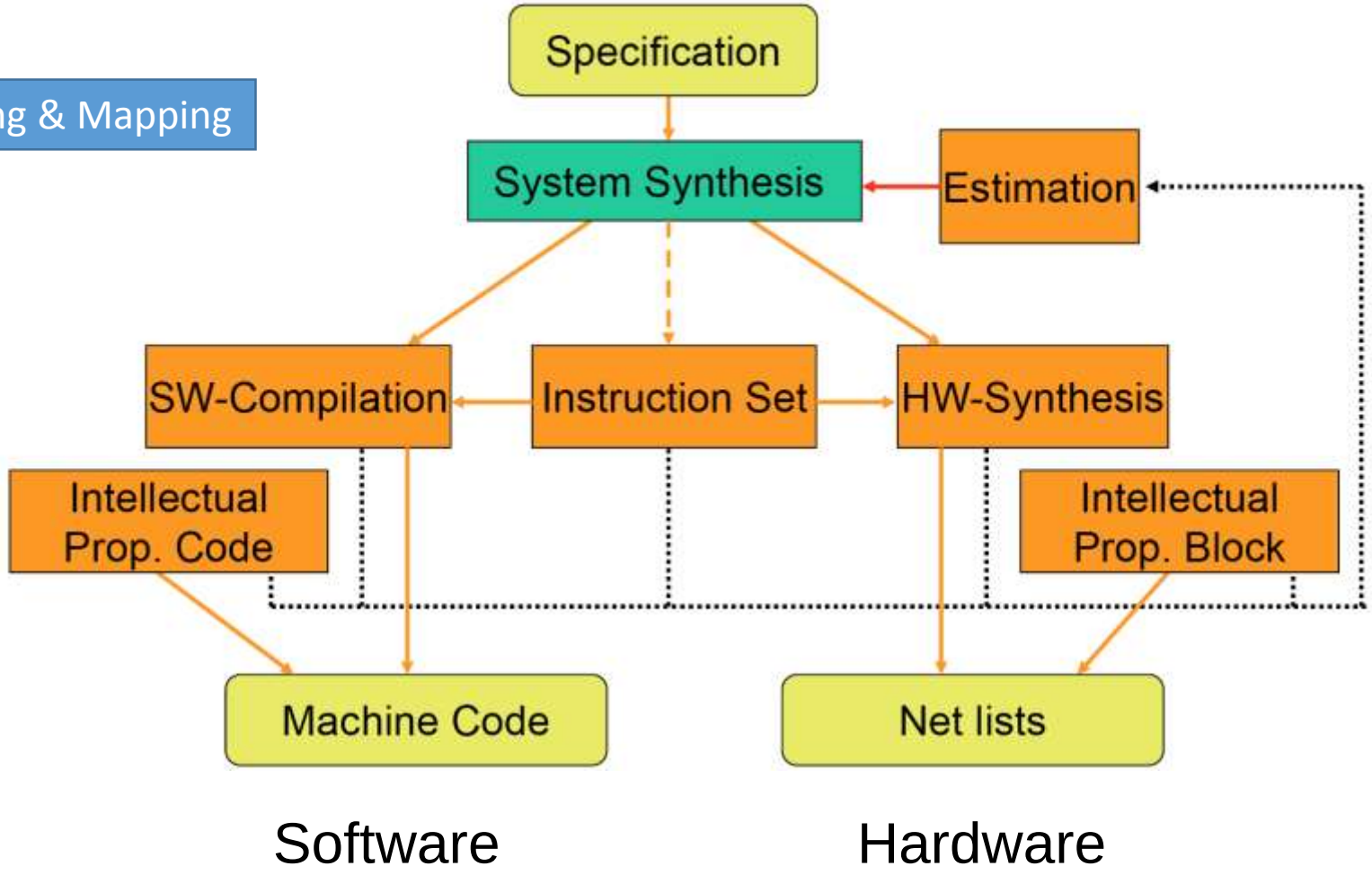
Design Methodology Transition



* Adapted from "Surviving the SOC Revolution."

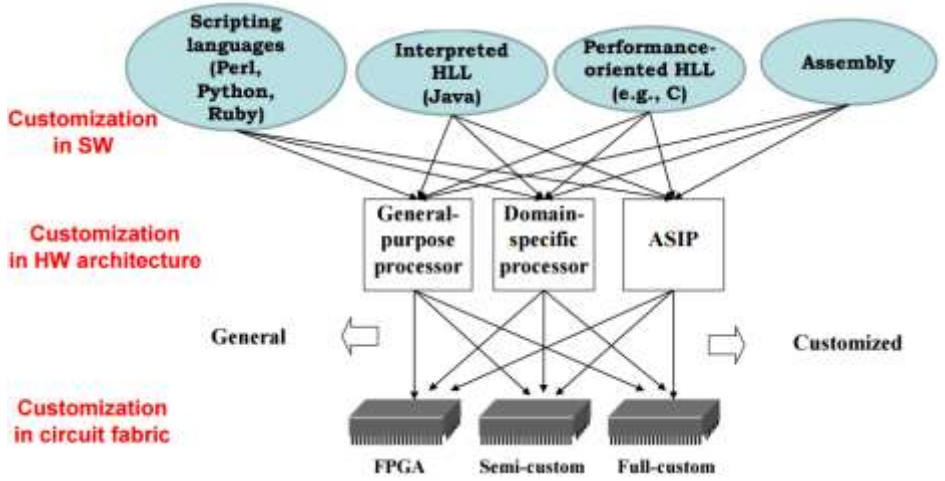
SoC Design Flow

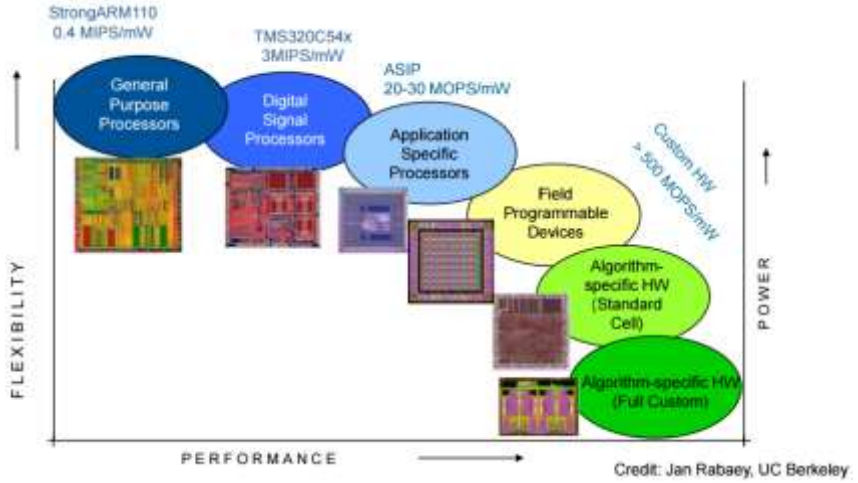
HW/SW Partitioning & Mapping



SoC Design Flow

- Hardware/Software Partitioning
  - Choosing the best combination of hardware and software to implement a give n function
- Customization
  - Tradeoff between efficiency and (design effort / generality / flexibility)
  - Fundamental recurring tradeoff at different levels of abstraction
  - Orders of magnitude in performance and energy efficiency





ECE 695R: System-on-Chip Design

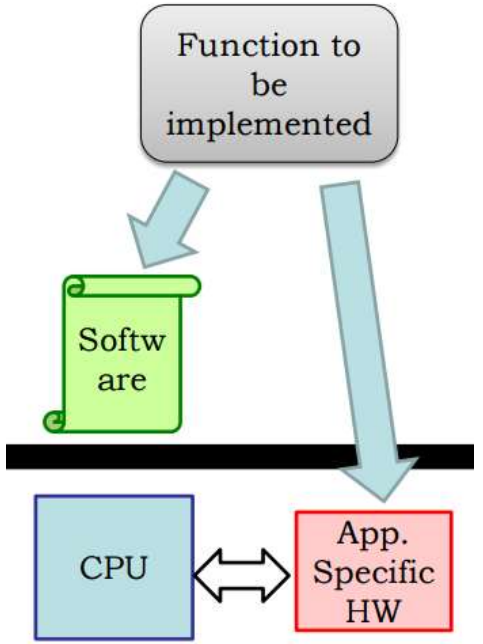
SoC Design Flow: HW/SW Partitioning

- Determining the right mix of “hardware” and “software” to implement a gi ven function
  - Software → Runs on a general-purpose programmable processor
  - Hardware → Some degree of specialization to the function being implemented

Objectives

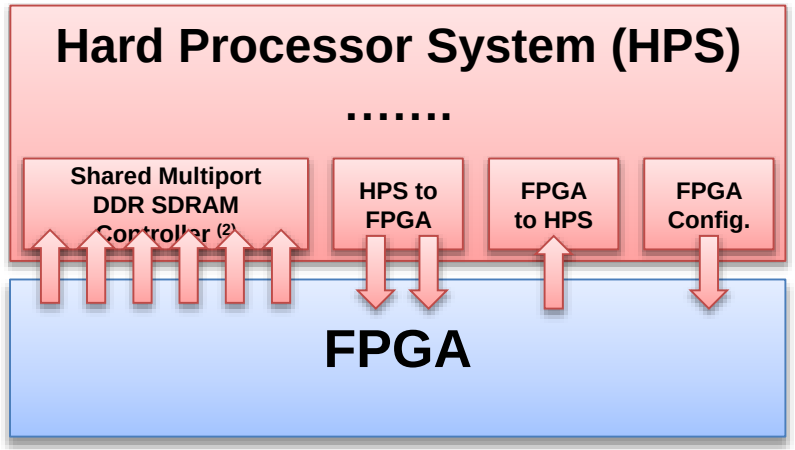
– Meet performance target with minimum hardware added – Minimize energy/power while meeting performance constraint

– Maximum performance while keeping certain functions flexible



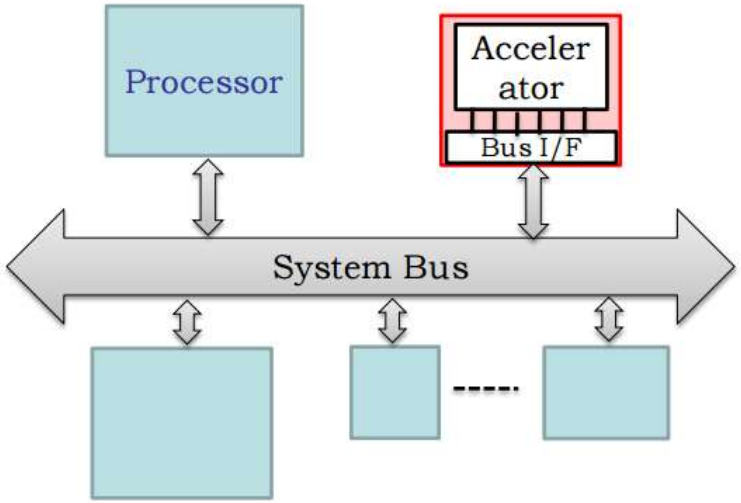
System Architecting

- HW/SW design partitioning (cont)
  - Data coherency maximizes partitioning options
    - Increase power efficient computing
      - Co-processing certain tasks in the FPGA fabric can not only benefit from a per formance standpoint but also reduces the per power computational element
    - Decreased and much finer grained latency potentials (do not have to deal with protocol overheads)
    - Saves on board level debugging
    - Not I/O restrictive and removes having to commit to a board level CPU to FPGA interface
    - Expedites board design faster



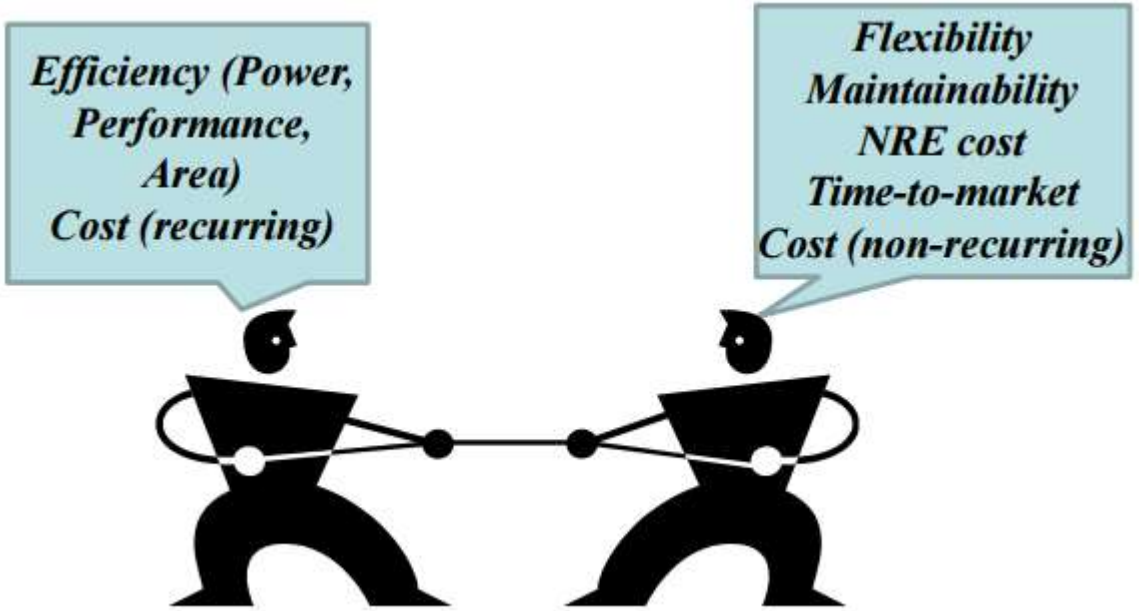
HW/SW Interfacing Basics

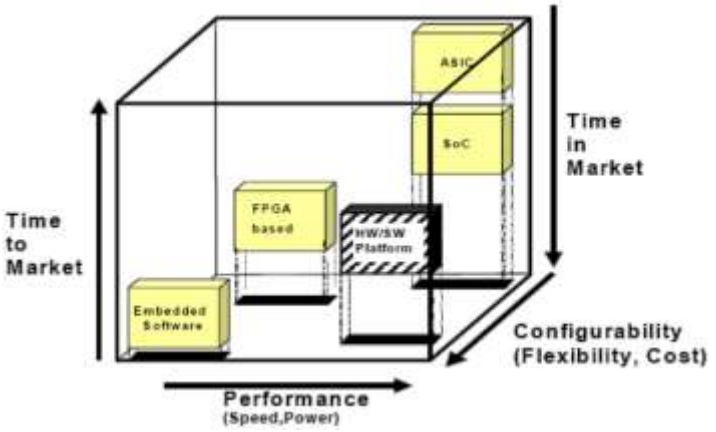
- Hardware view: Connect the accelerator to the communication architectur e (system bus)



What is the right level of customization?

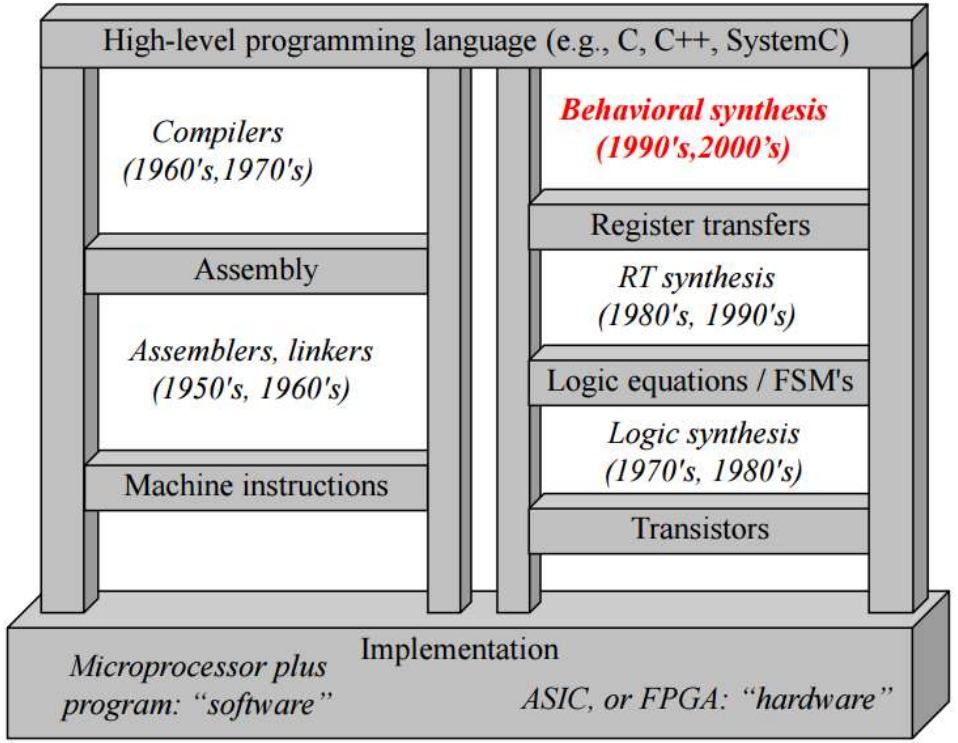
- SoC design requires complex tradeoffs between different desig n considerations!



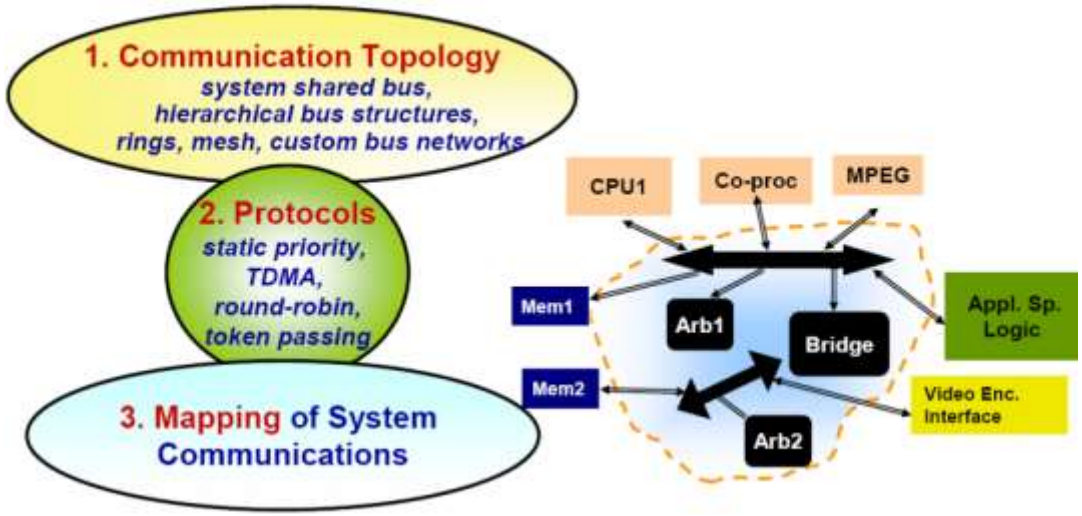


Abstraction levels for specifying hardware and software

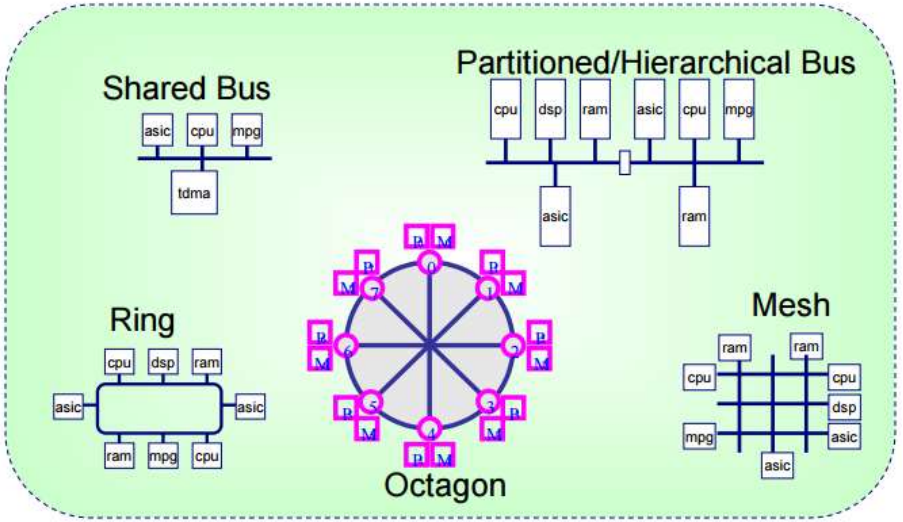
- Recent developments allow hardware to be specified using sof tware-like descriptions
  - Need to automatically translate to lower levels of abstraction



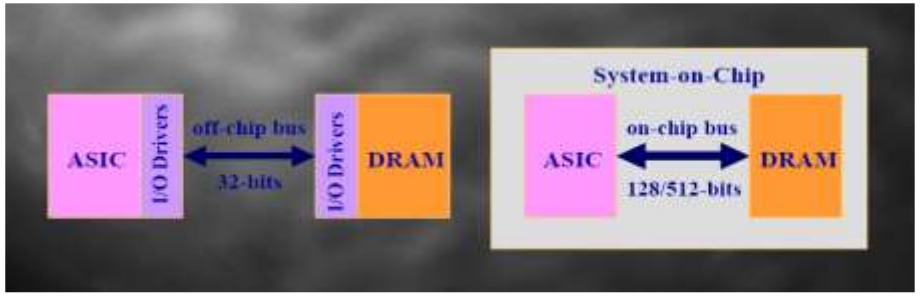
On-Chip Communication Architecture Design



Topologies range from fully sequential (single shared bus) to fully parallel (point-to-point)



Internal Bandwidth vs External Bandwidth

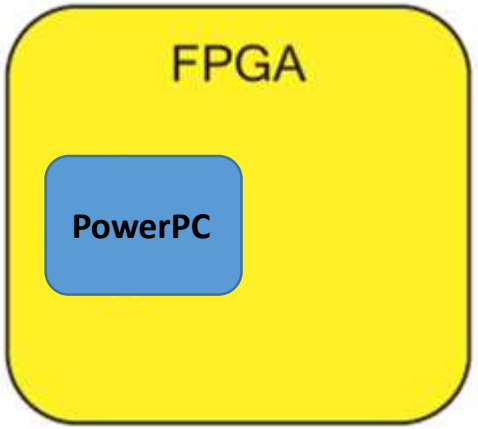


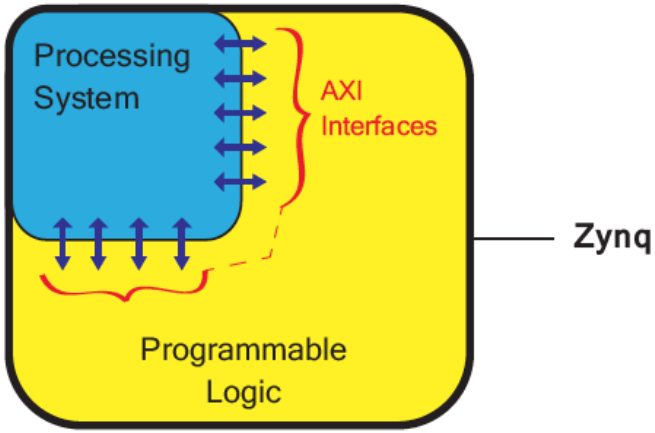
Let's Explore Xilinx Zynq

Processors inside FPGAs?

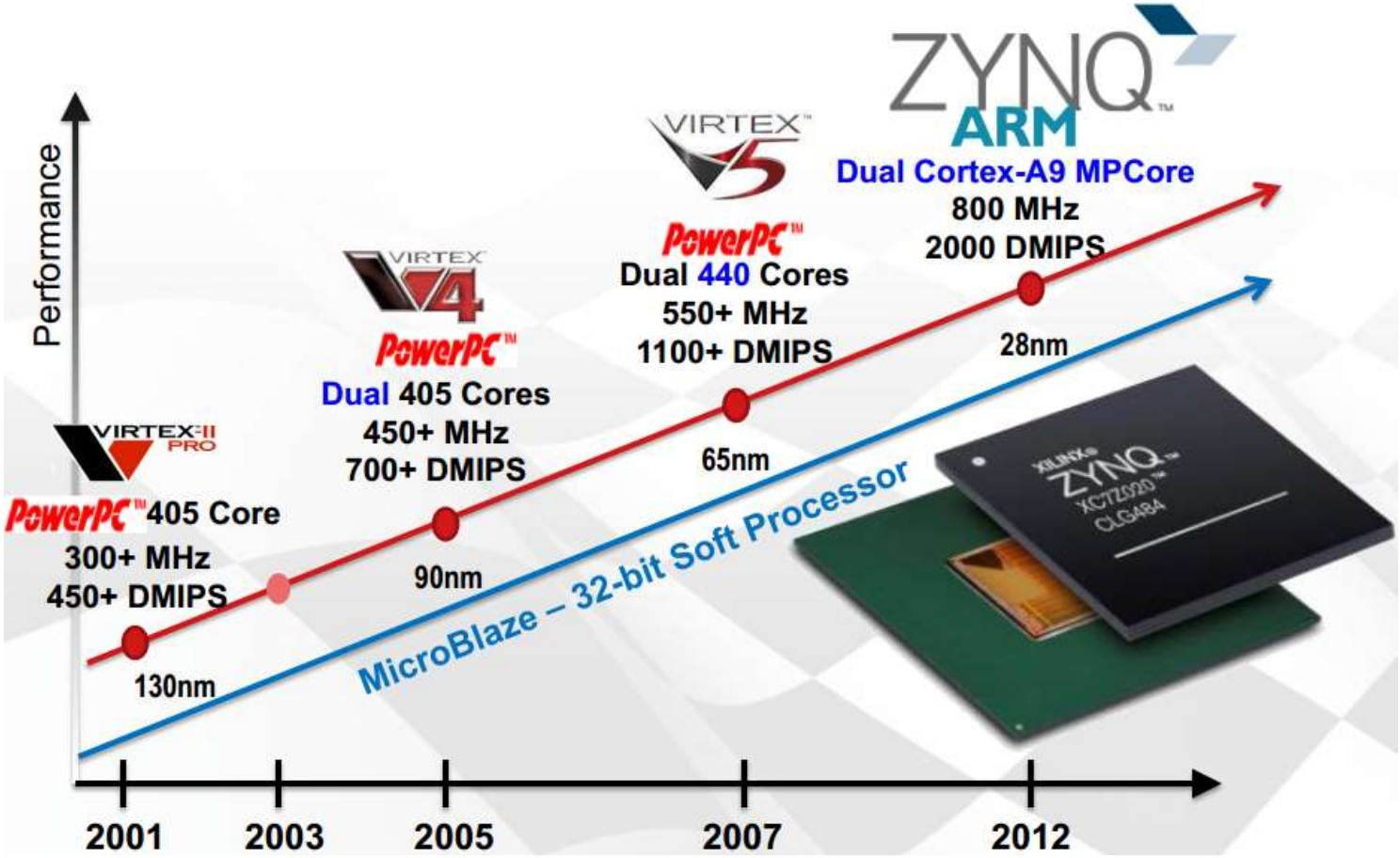
FPGA with Soft Processor Core

FPGA with Hard Processor Core





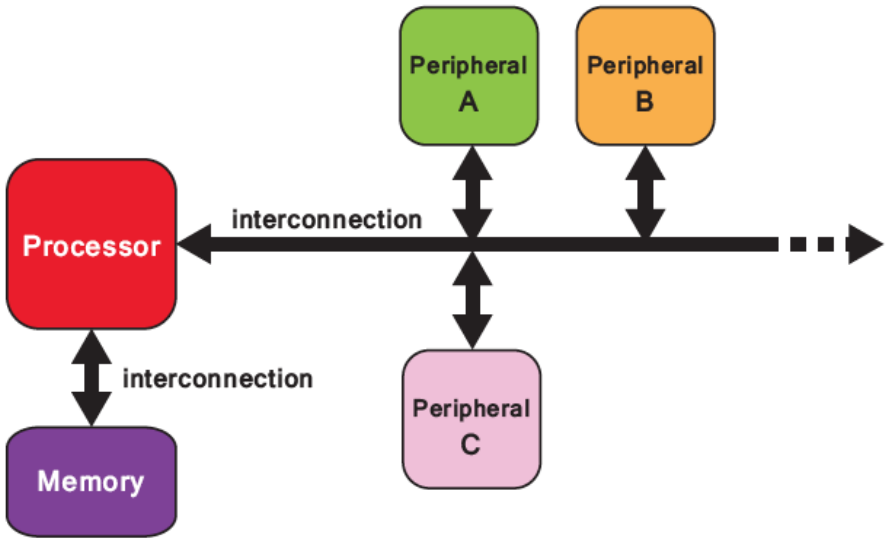
Xilinx Processing Heritage



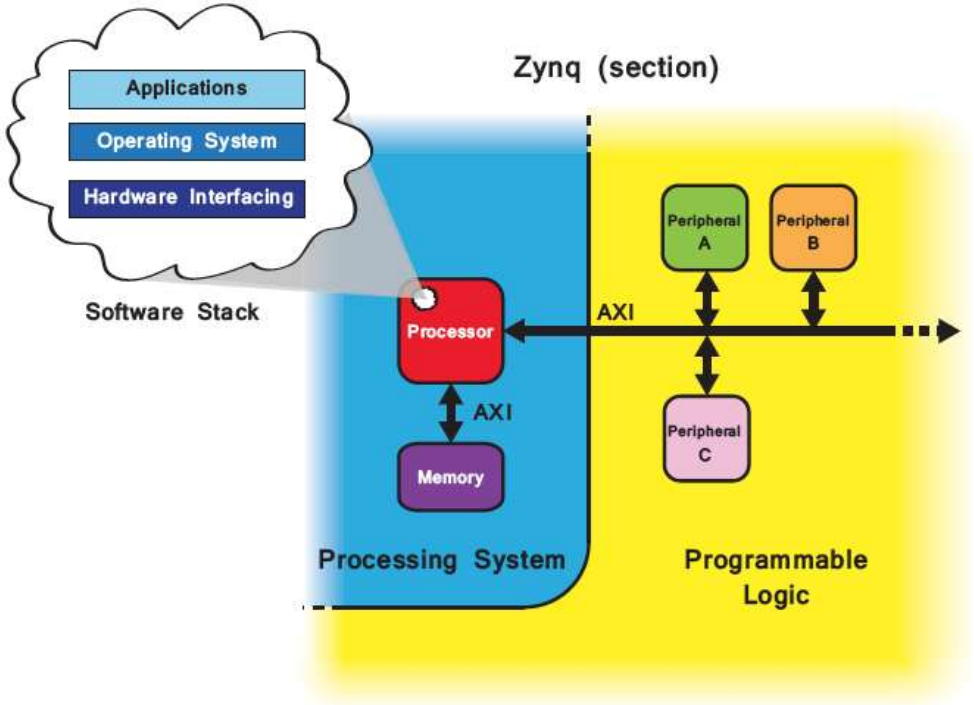
10+ years, 4 Generations

Hardware Architecture of an Embedded SoC

Simplified Hardware Architecture of an Embedded SoC



Mapping of an Embedded SoC Hardware Architecture to Zynq



Comparison with Alternative Solutions

	ASIC	ASSP	2 Chip Solution	Zynq
Performance	+	+	■	+
Power	+	+	−	+
Unit Cost	+	+	−	■
Total Cost of Ownership	■	+	+	+
Risk	−	+	+	+
Time to Market	−	+	+	+
Flexibility	−	−	+	+
Scalability	−	■	+	+

+ positive, − negative, ■ neutral

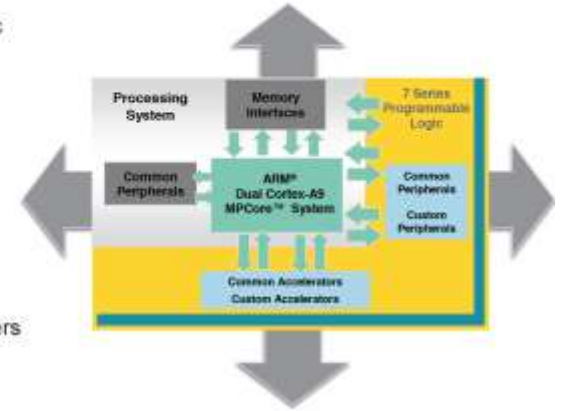
Zynq Highlights

- **Not an ordinary FPGA!**
- **Not an ordinary microprocessor!**
- **Unique blend of the two technologies**
  - Dual Cortex™-A9 processors +caches + robust peripheral set
  - Xilinx 7 series fabric
  - Powerful interconnects between the two

It's no longer just an FPGA, it's a processing system with programmable logic!

ZYNQ™

- Complete ARM®-based Processing System
  - Dual ARM Cortex™-A9 MPCore™, processor centric
  - Integrated memory controllers & peripherals
  - Fully autonomous to the Programmable Logic
- Tightly Integrated Programmable Logic
  - Used to extend Processing System
  - High performance ARM AXI interfaces
  - Scalable density and performance
- Flexible Array of I/O
  - Wide range of external multi-standard I/O
  - High performance integrated serial transceivers
  - Analog-to-Digital Converter inputs



Programmable Systems Integration

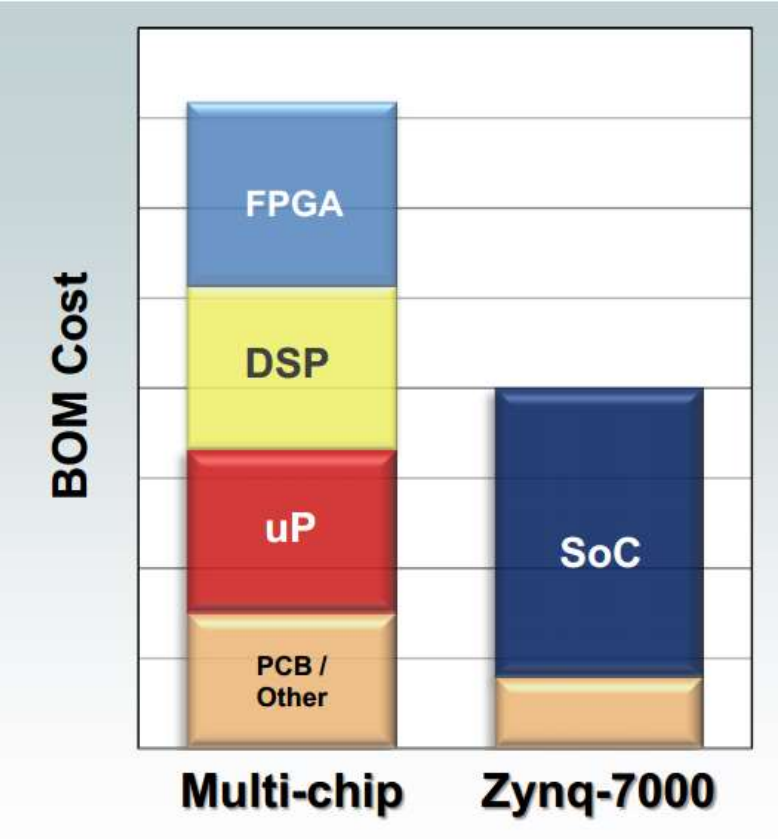
- **All Programmable Platform Integrating Multiple Components**
  - Hardware and Software programmable
  - Board component reduction
  - Security & reliability
  - Manufacturing benefits

- **Ultimate Flexibility**
  - Create custom, flexible SoC to meet exact project needs in a single device
  - HW / SW partitioning optimized to specific application requirements

ARM Programmability +
FPGA Flexibility in a Single Chip

BOM* Cost Reduction

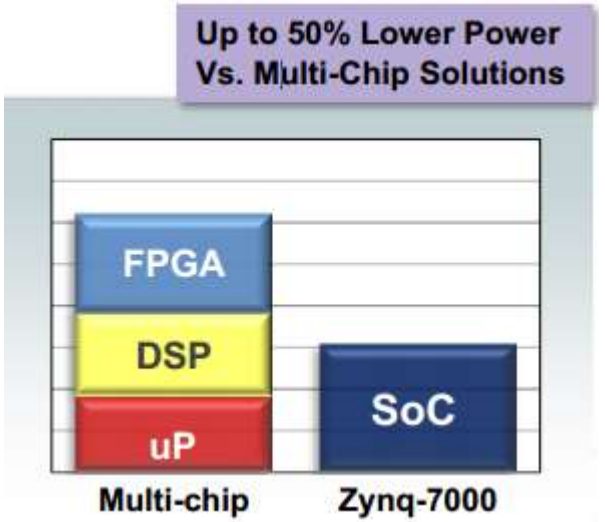
- **Reduce Devices Per Board**
  - Processor
  - PLD
  - DSP
  - A/D
  - Power
- **Reduced PCB Complexity**
  - Fewer traces = Fewer Layers
  - Fewer terminations
  - Faster Design
- **In-System Reconfiguration**

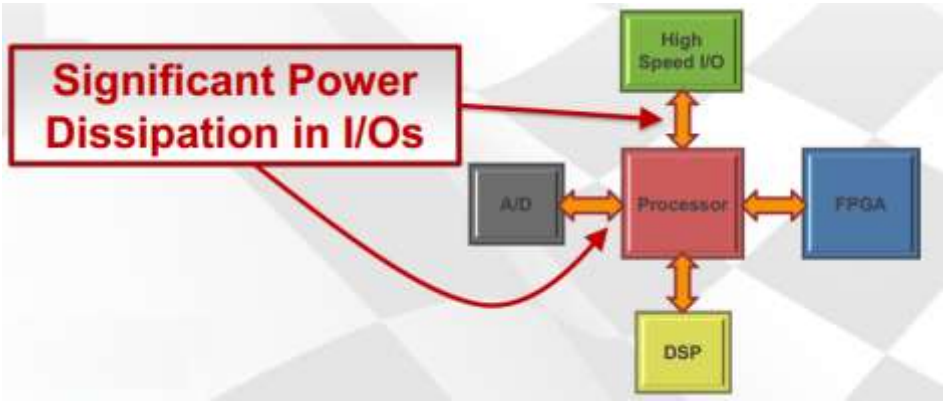


*BOM: Bill Of Materials

Total Power Reduction

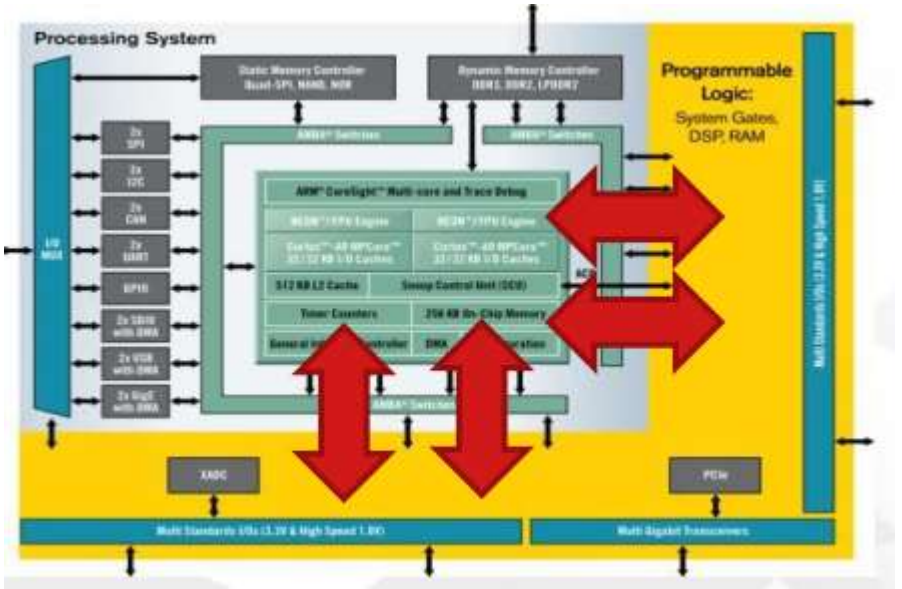
- **Flexible/Tunable Power Envelope**
  - Adjusted frequencies
  - Clock gating
  - ARM low Power States
- **Integration = Power Reduction**
  - Static Power
  - I/O Power





Increased System Performance

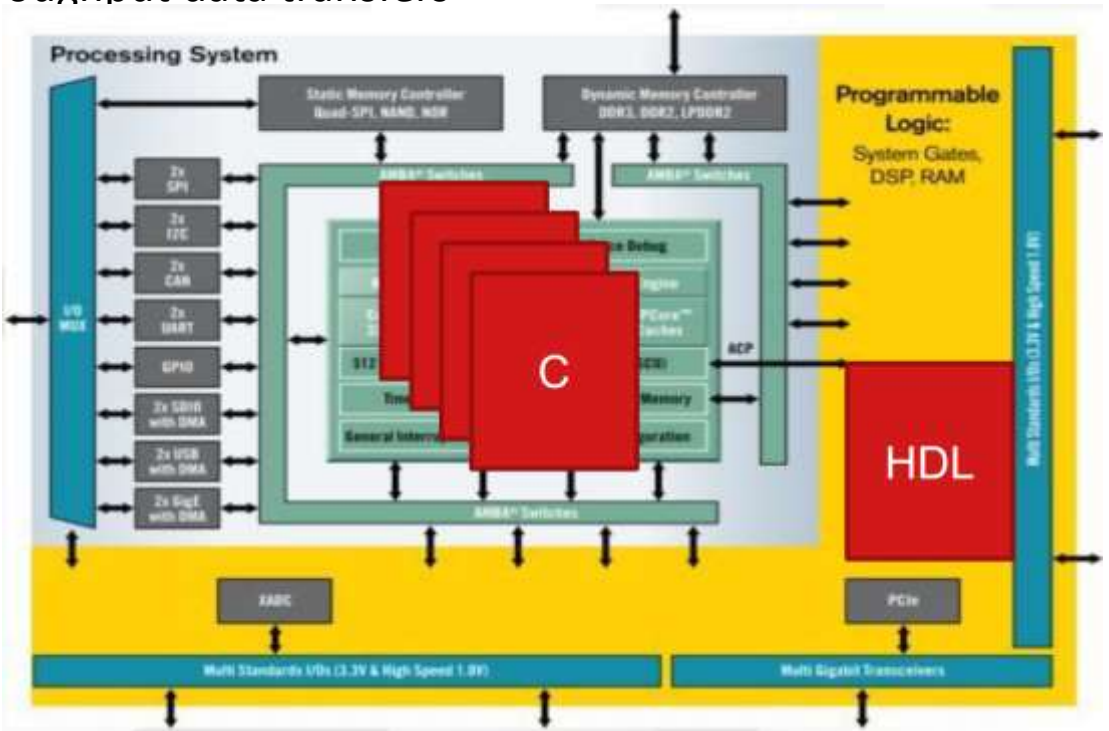
- **Meet HW and SW Processing Performance Needs**
  - Programmable Logic
  - Massive DSP processing
  - High throughput AXI
  - High performance I/Os
  - Gigabit transceivers



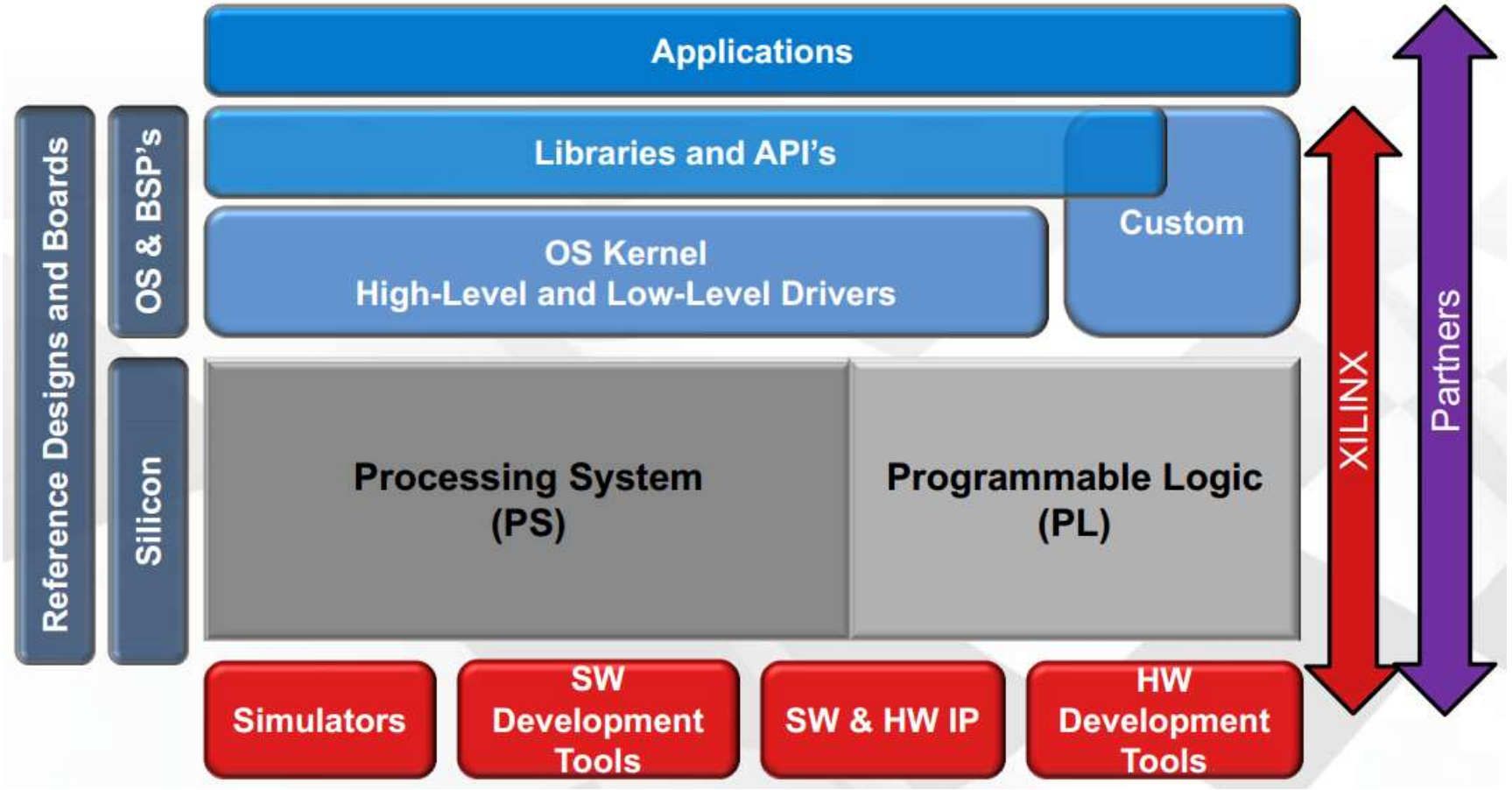
Elements	Performance (up to)
Processors (each)	1 GHz
PL Fabric/ DSP Fmax	741 MHz
DSP (aggregate)	1080 GMACs
Transceivers (each)	12.5Gbps

Increased System Performance

- **Optimized & Simplified HW/SW Partitioning**
  - HW acceleration enables scaling SW performance to address many application s
  - Low latency interfacing for efficient co-processor implementation and high thr oughput data transfers



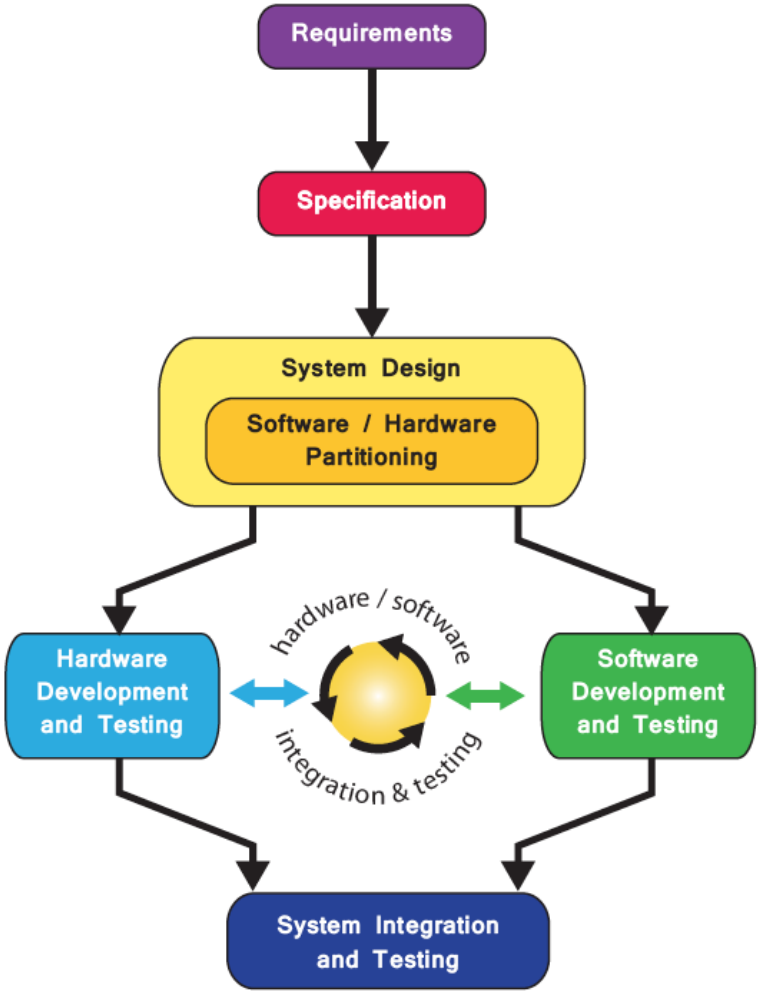
Zynq-7000 AP SoC Programming Model



Zynq-7000 OS Support

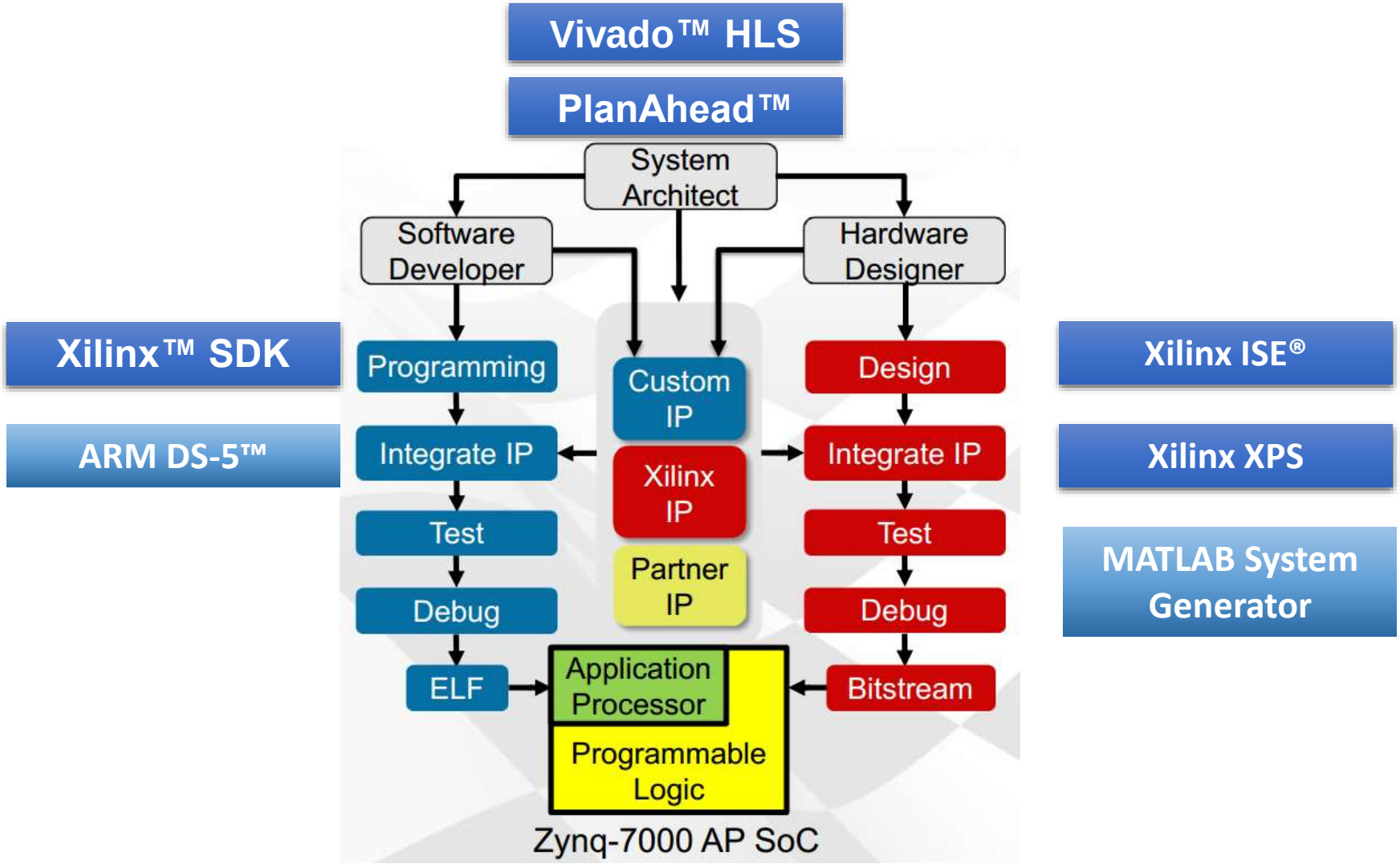
- **The Zynq™-7000 family of devices of fers a comprehensive collection of o perating systems to suit your system needs**
- Open Source Linux
  - Xilinx provides a freely downloadabl e Linux solution
- Open Source Android
  - Freely downloadable Android 2.3 sol ution
- Open Source FreeRTOS
  - light-weight real-time OS
  - used in applications that demand de terministic and real-time responsive ness to events in the system

Zynq SoC Embedded Design Flow

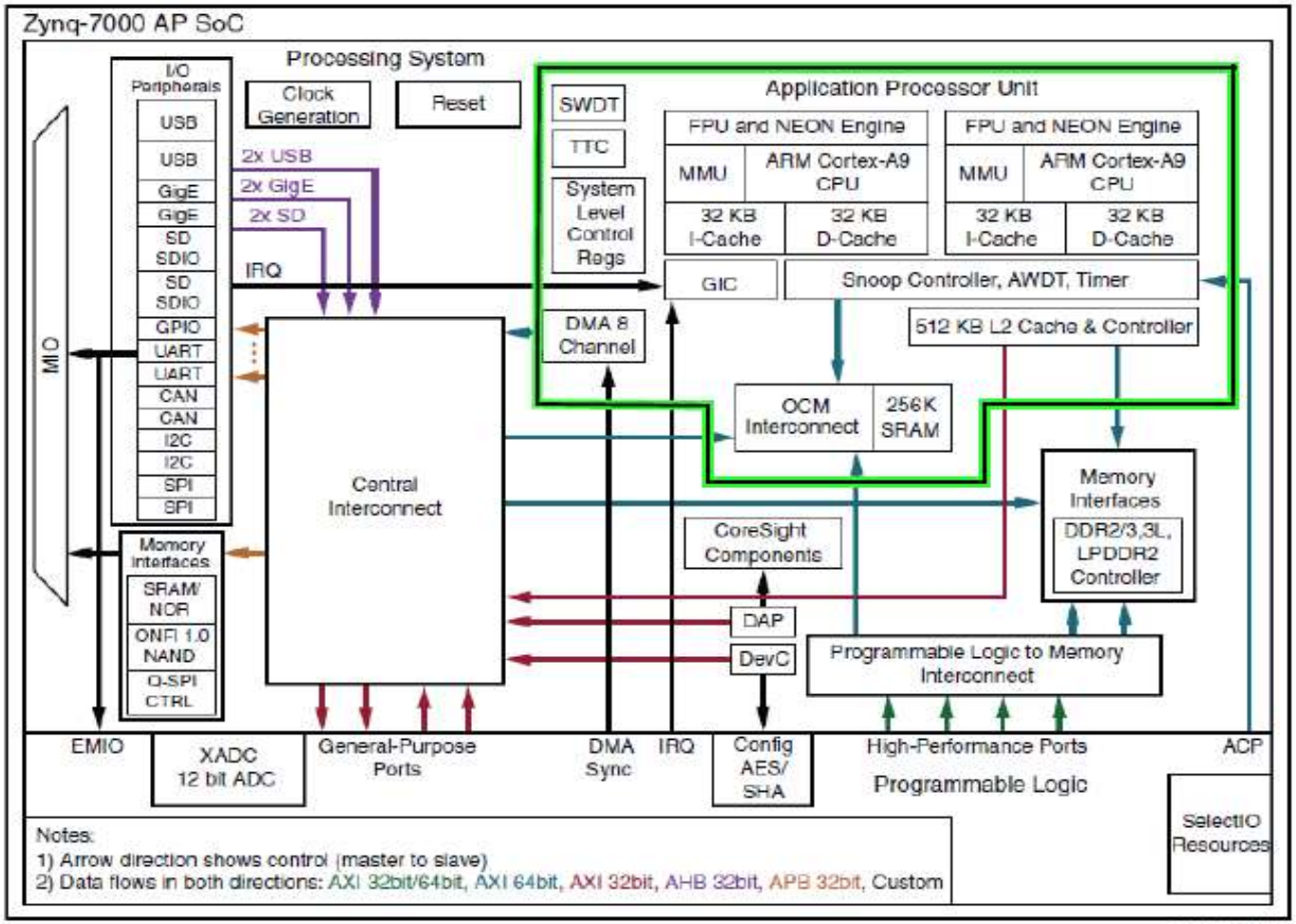


Zynq SoC Embedded Design Flow

Tools Point of View



The Zynq Processing System



Applications

- Automotive Applications
  - Lane and Road Sign Recognition
- Computer Vision
  - Detection of Cars at a Junction
- Smart Home
- Software Defined Radio (SDR)
- Communication Systems
  - Wireless Base-station
  - Satellite Ground-station
  - Wired Network Switches
- Control and Instrumentation Systems
- Medical Applications

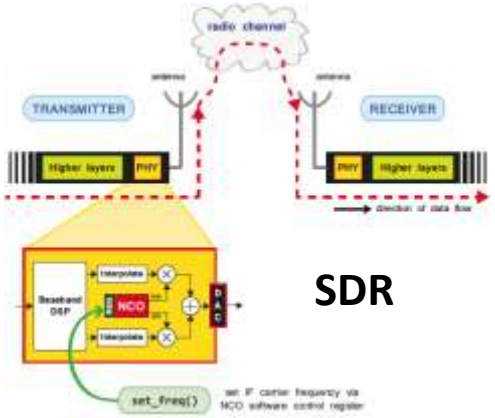






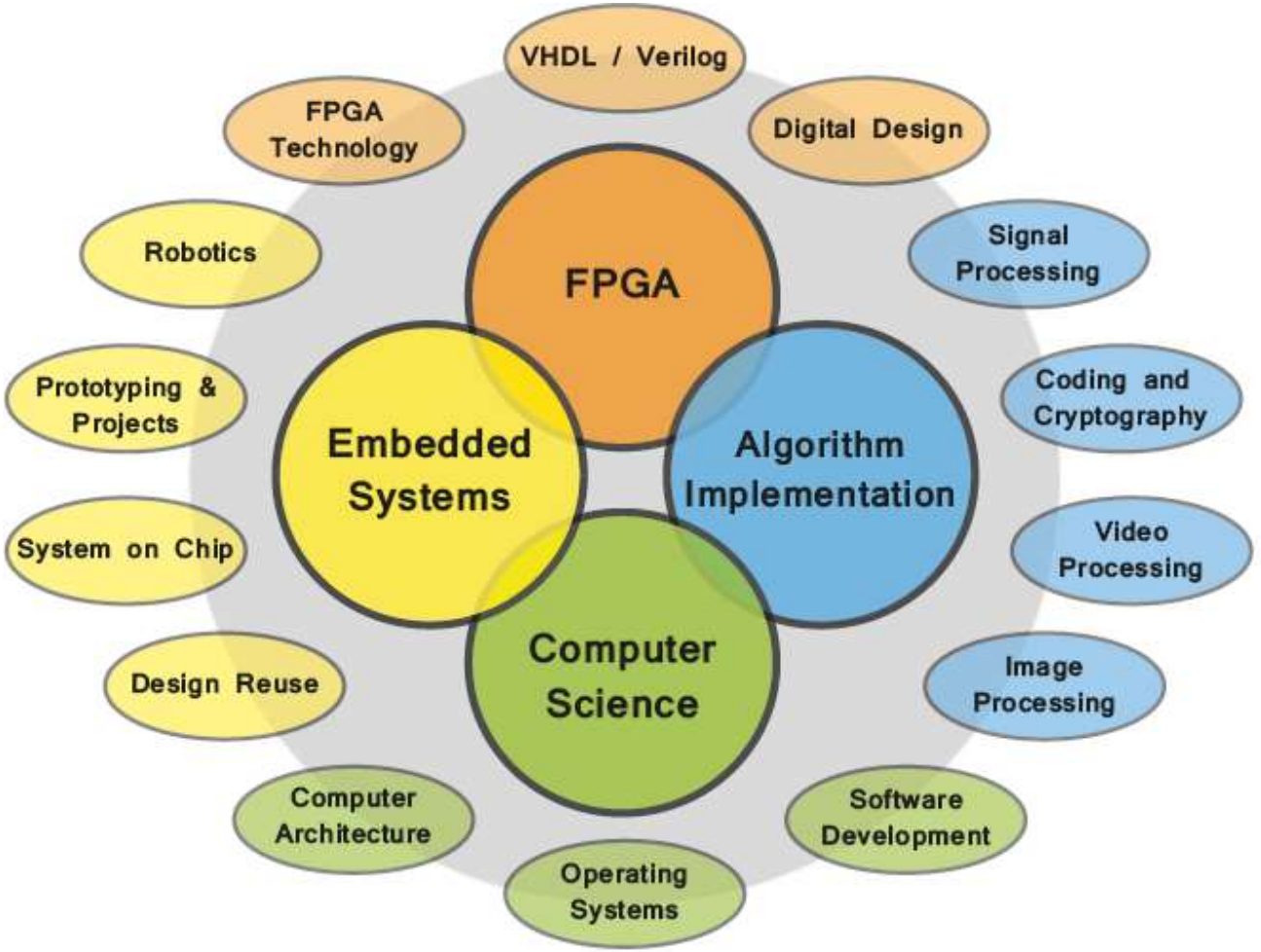




SDR

Academic Subjects to which Zynq is Relevant



Choice Among Various Implementation Platforms

	Total System Cost	Flexibility	Differentiation	Time-to-Market	Cost of Derivatives	Risk
Zynq SoC	Low + best value	Most flexible: HW and SW programmable + programmable I/O	Highest degree of programmability, HW/SW co-design	Fastest for integrated HW & SW differentiation	Lowest due to HW & SW programmability	Predictably low risk
ASSP + FPGA	Higher than Zynq SoC (system dependent)	Highly flexible but ASSP I/O limited compared to Zynq SoC	HW and SW programmable, ASSP-dependent	Fastest if ASSP requires HW differentiation	Low to high depending on FPGA vendor	Low to high depending on FPGA vendor
ASSP	Lowest if SW-only programmability is sufficient	Good but SW-programmable only	Limilted to SW programmable only - easy cloning	Fastest if SW-only differentia-tion required	Lowest if SW-only derivatives needed	Can be Lowest if SW-only programmability is sufficient
ASIC	High to prohibitive	Once manufactured only limited SW flexibility	Best HW differentiation but limited SW differentiation	Lowest & riskiest	Highest	Terrible (respins)

Advantages of Zynq

Lowest NRE, Best Risk Mitigation	Greatest Flexibility & Differentiation	Streamlined Productivity & Fast TTM	Lowest Cost of Derivatives & Highest Profitability
✔ Already manufactured silicon	✔ All Programmable HW, SW & I/O	✔ Instant HW/SW co-development	✔ IP standardized on ARM AMBA AXI4
✔ Negligible development & design tool costs	✔ Anytime field programmable	✔ All Programmable Abstractions (C, C++, OpenCV, OpenCL, HDL, model-based entry)	✔ Reuse precertified code (ISO, FCC, etc.)
✔ Xilinx IP library + third-party IP	✔ Partial reconfiguration	✔ Vivado Design Suite, Vivado HLS, IP Integrator & UltraFast Methodology	✔ Reuse & refine code & testbenches
✔ Extensive development boards	✔ System Secure (encryption)	✔ Broad and open OS & IDE support (Open-source Linux & Android, FreeRTOS, Windows Embedded, Wind River, Green Hills, & many others)	✔ Volume silicon, power circuity, PCBs & IP licensing

The Development Boards

Entry Level

199$

ZYBO

Low Cost

199$

MicroZed

Versatile

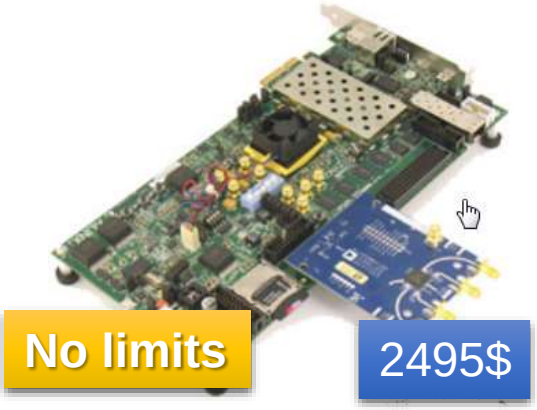
395$

Zedboard

Powerful

895$

ZC702



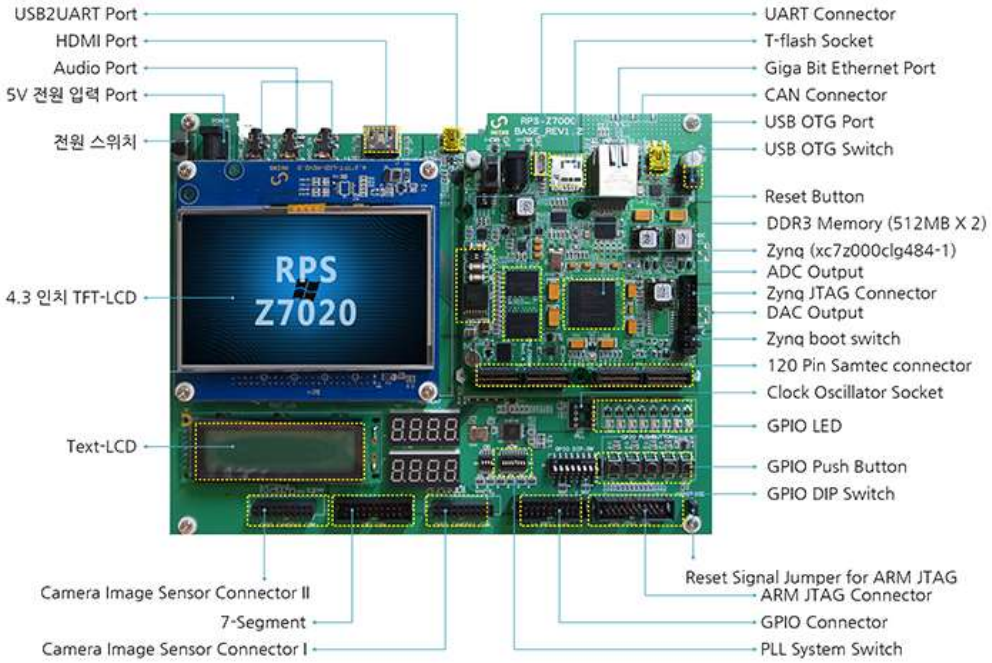
No limits

2495$

ZC706

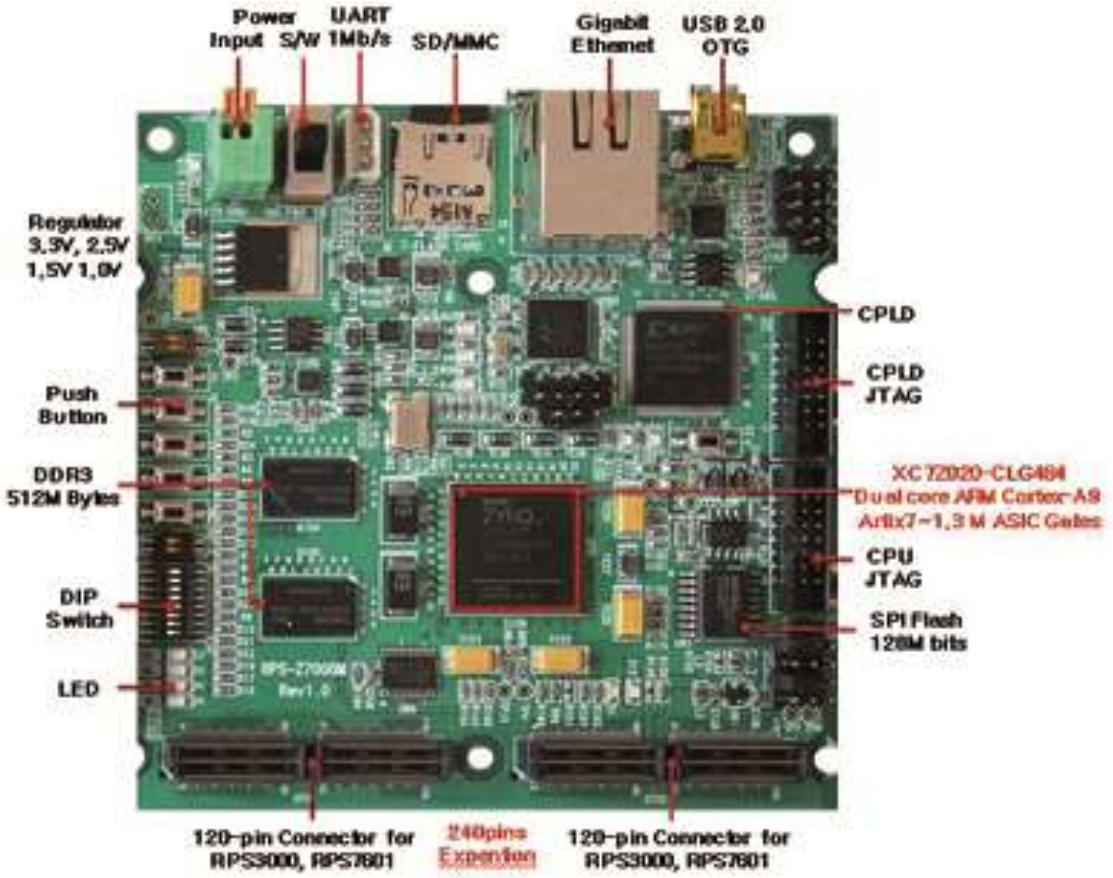
http://www.fpgadeveloper.com/2014/03/comparison-of-zynq-boards.html

Our Development Board



http://www.huins.com/new/sub/goods_view.php?it_id=1428652397&ca_id=20&ca_id2=20&n=2&sn=3

Our Development Board

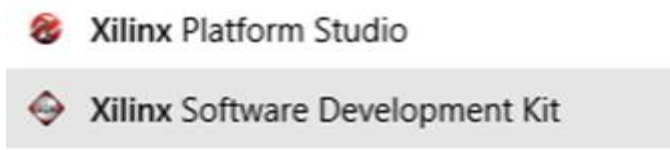


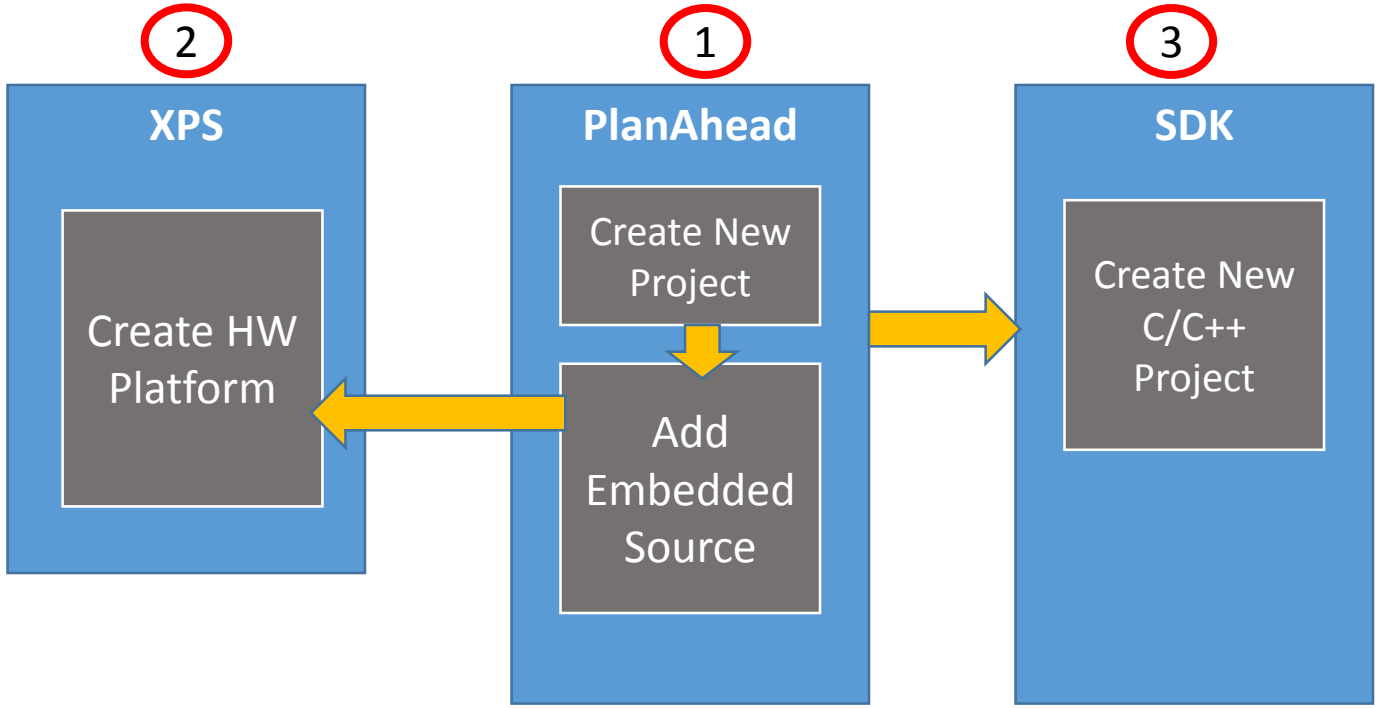
Getting Started with Software/Hardware Co-design using PlanAhead and Zynq

Lab 1: Creating a Zynq Project from PlanAhead
Lab 2: PS Config Part 1 – Hello World
Lab 3: Adding a Custom PL Peripheral
Lab 4: Writing Software for a Custom PL Peripheral
Lab 5: Program the FPGA

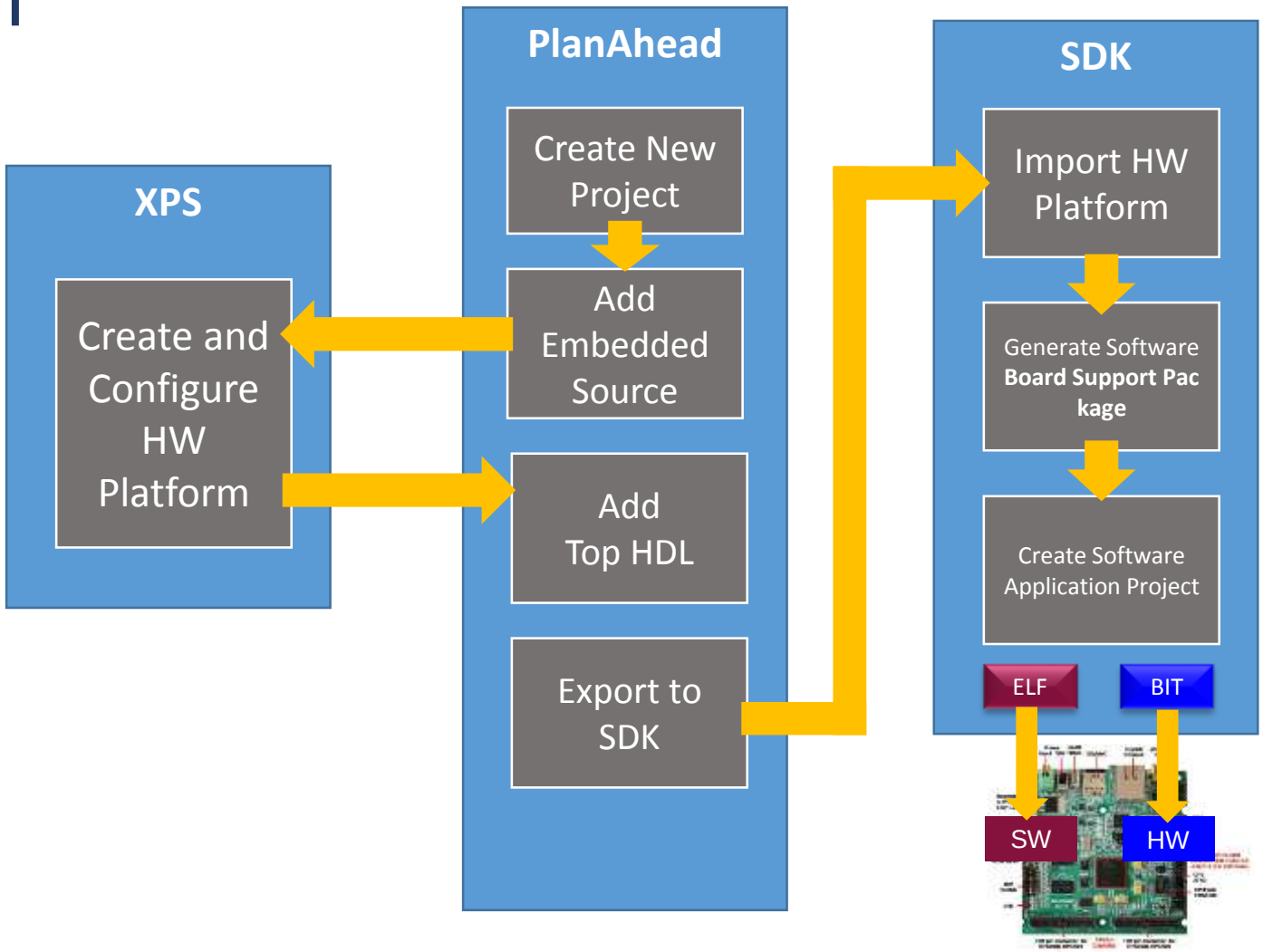
Our Tools

- Xilinx PlanAhead.
- Xilinx Platform Studio (XPS).
- Xilinx Software Development Kit (XSDK)





Detailed Configuration



LAB 1: CREATING A ZYNQ PROJECT FROM PLANAHEAD

Objectives

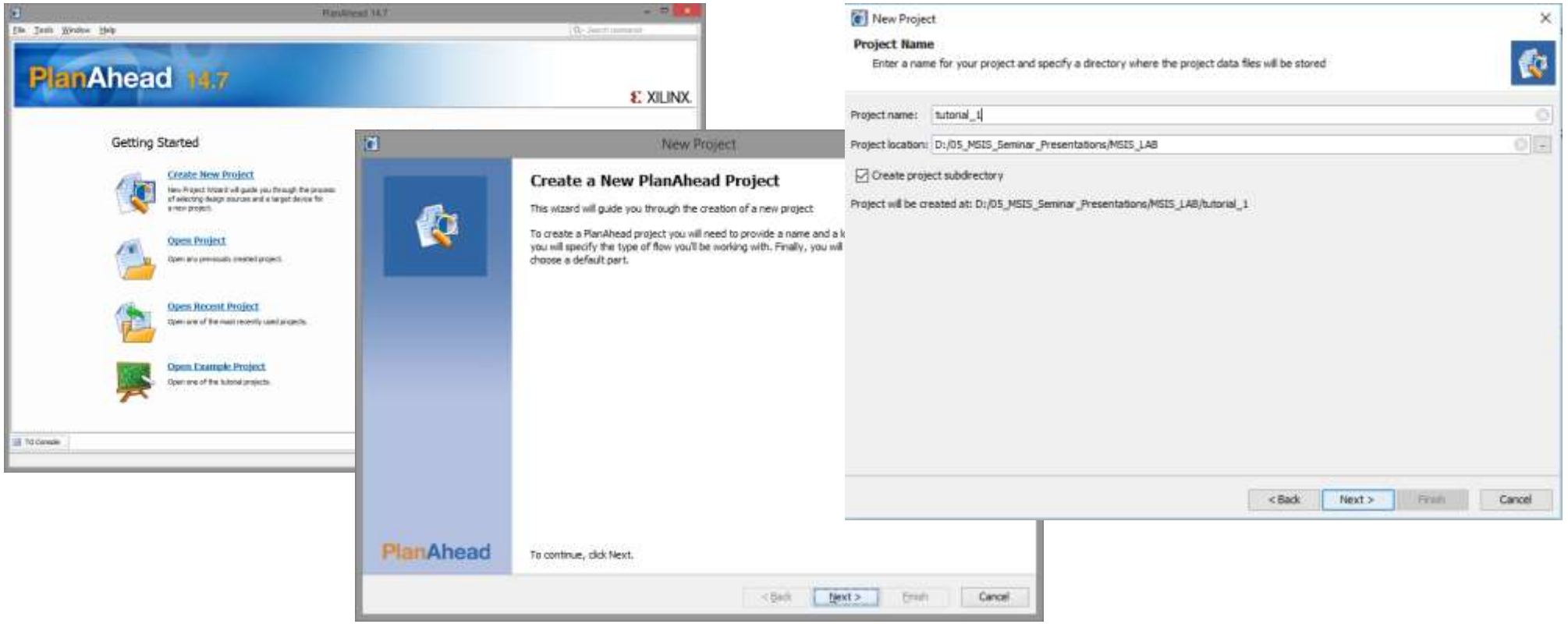
- When you have completed Lab 1, you will know how to do the following:
  - Create a new project in PlanAhead (PA)
  - Target the Zynq 7Z020 device
  - Add an Embedded ARM source to the PA project
  - Launch Xilinx Platform Studio (XPS) for editing of the Embedded Source

Create a New Zynq Project in PlanAhead (PA)

- The Zynq Processing System (PS) may be used without anything programmed in the Programmable Logic (PL).
  - However, in order to use any soft IP in the PL, or to route PS dedicated peripherals t o device pins for the PL, programming of the PL is required.
- A Zynq PS-only project can be completed in XPS and SDK standalone.
  - However, once any stitching of PL logic with the PS is required, an over-reaching de sign cockpit like PlanAhead or Project Navigator is required.
- PlanAhead is the recommended tool by Xilinx, so that is what we'll use in these labs.
- PlanAhead software provides a central cockpit for design entry in RTL, synthesis an d verification.
- PlanAhead offers integration with XPS for embedded processor design (including ac cess to the Xilinx IP catalog), and SDK to complete the embedded processor softwar e design.

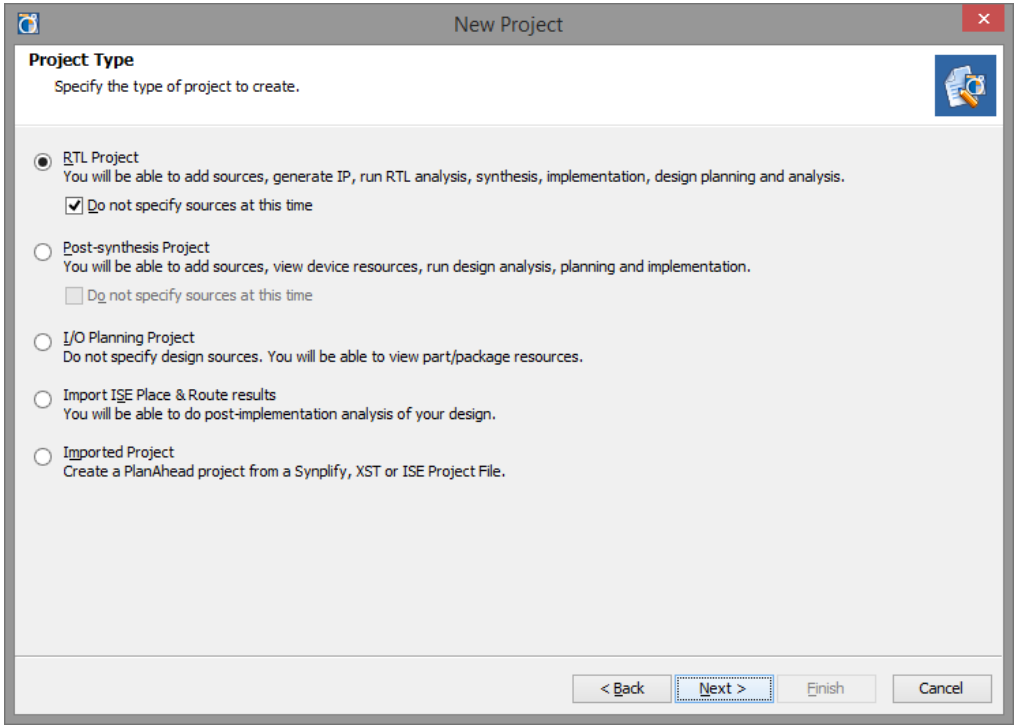
Create a New Zynq Project in PlanAhead (PA)

- Launch PlanAhead.
- Create a new project called tutorial_1 targeting the 7Z020-1CLG484 device



Create a New Zynq Project in PlanAhead (PA)

- The project will be RTL based, so leave the radio button for *RTL Project* sele cted.
- Since this is a brand new project, check the box for ***Do not specify sources at this time***. Click **Next >**.

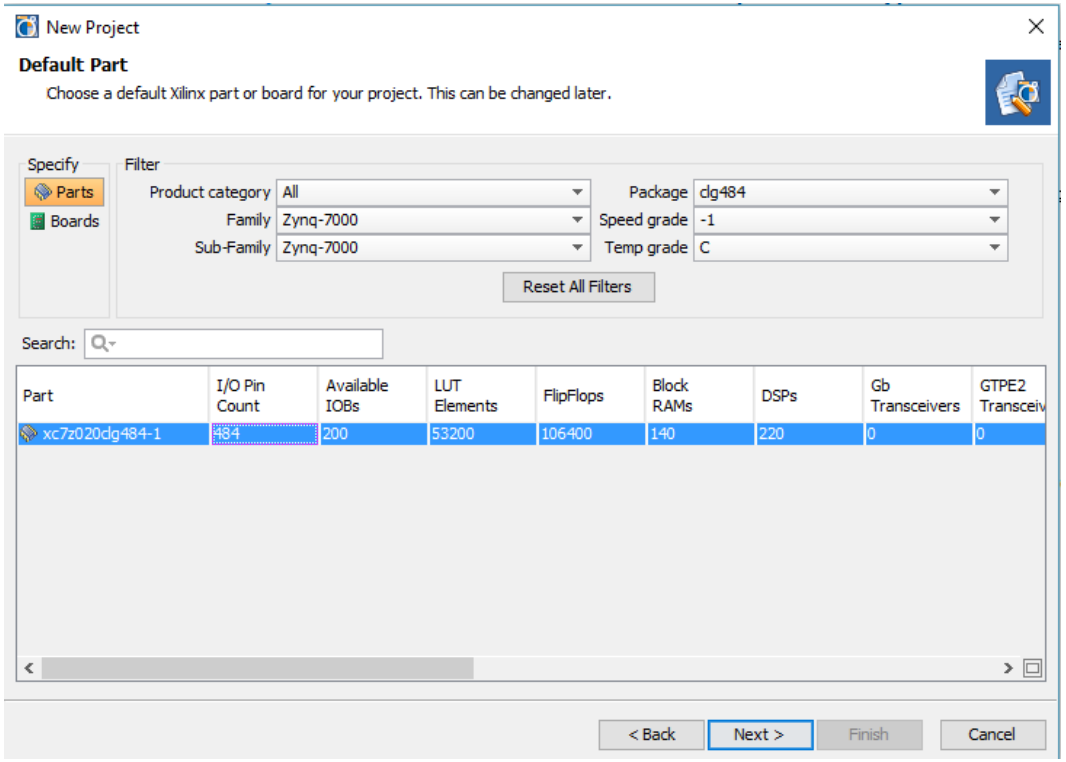


Create a New Zynq Project in PlanAhead (PA)

Now we get to select the board.

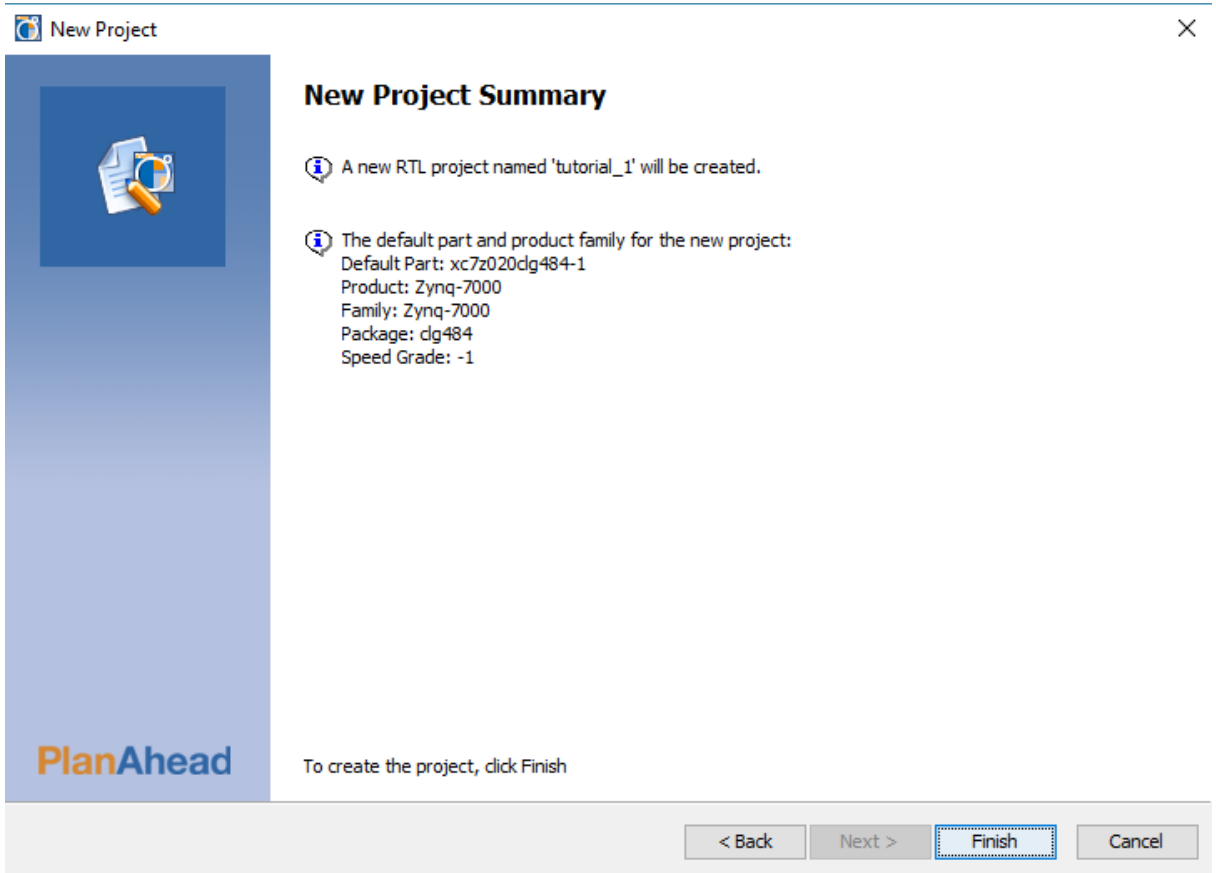
We have to select the chip at the time being. From looking at the board we can see that this is a Zynq -7000 series commercial grade CLG484 chip, with speed grade -1.

Using the filters we'll eventually find the xc7z0 20clg484-1 part, which is the Zynq-7020. Select it and click Next.



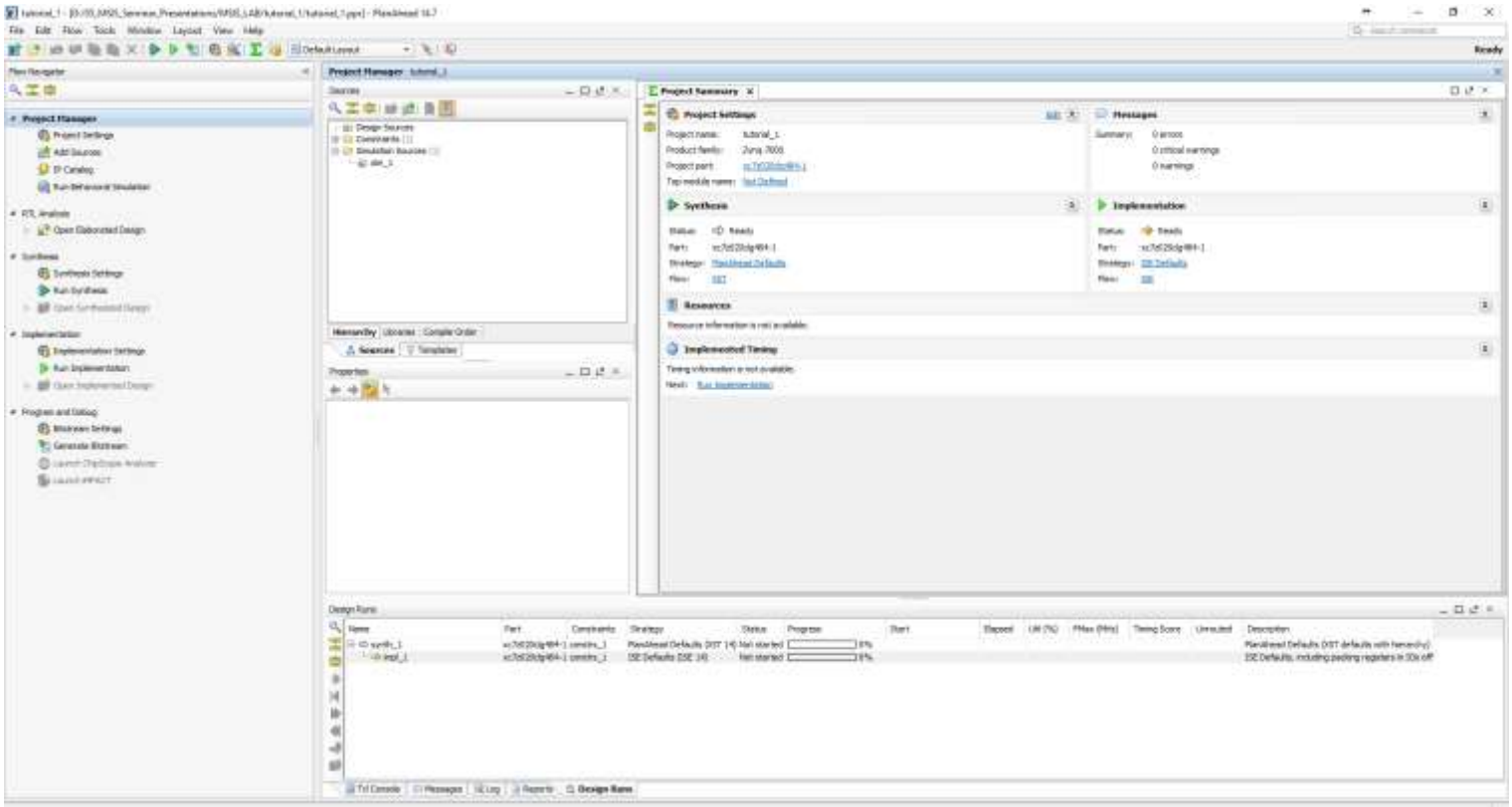
Create a New Zynq Project in PlanAhead (PA)

On the summary screen we confirm the selection and click **Finish** to be taken to the Project Manager.



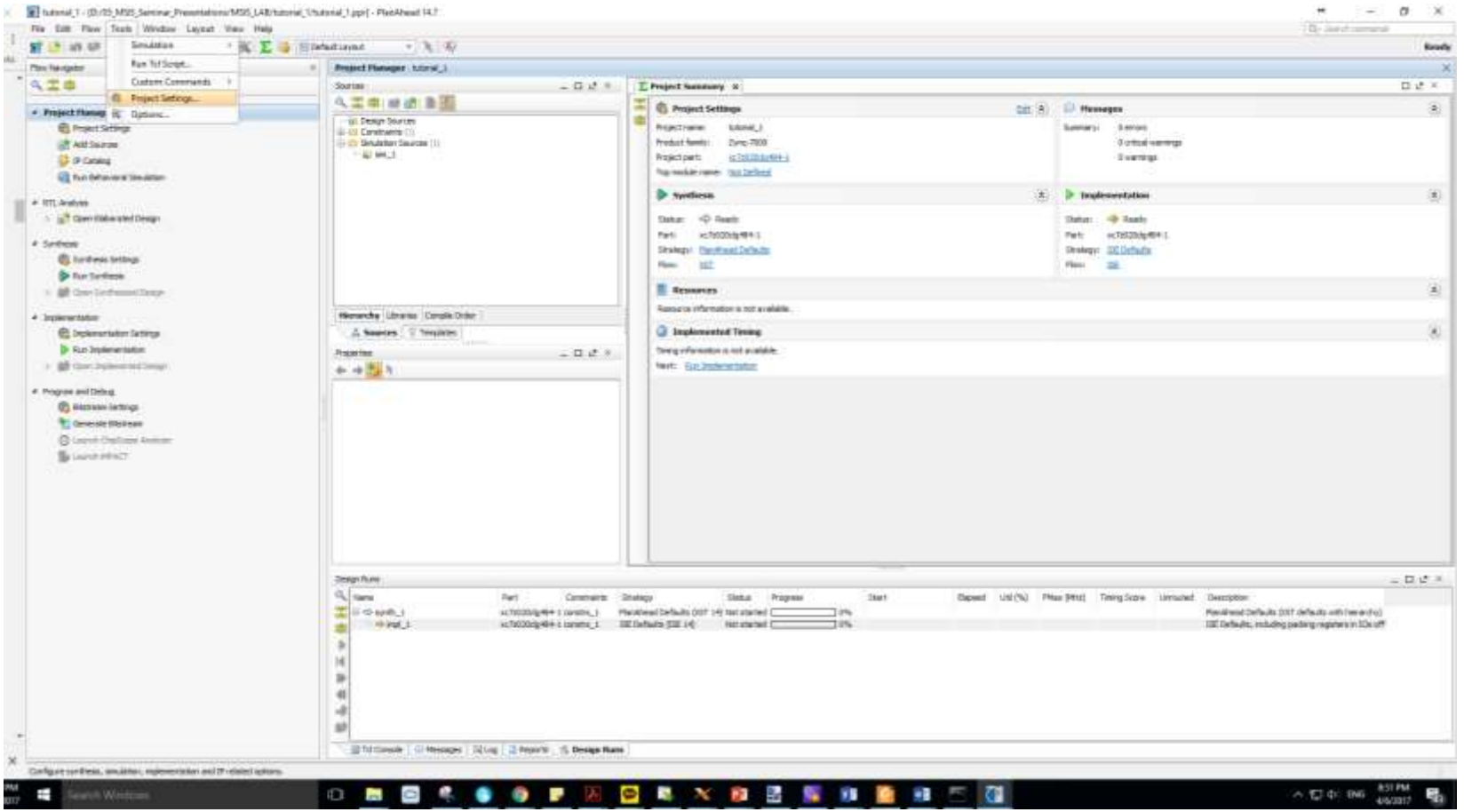
Create a New Zynq Project in PlanAhead (PA)

The Project Manager is part of the main screen of PlanAhead and shows the current target as well as projec t settings. It also presents shortcuts to the common steps in the workflow on the left, in the Flow Navigator.



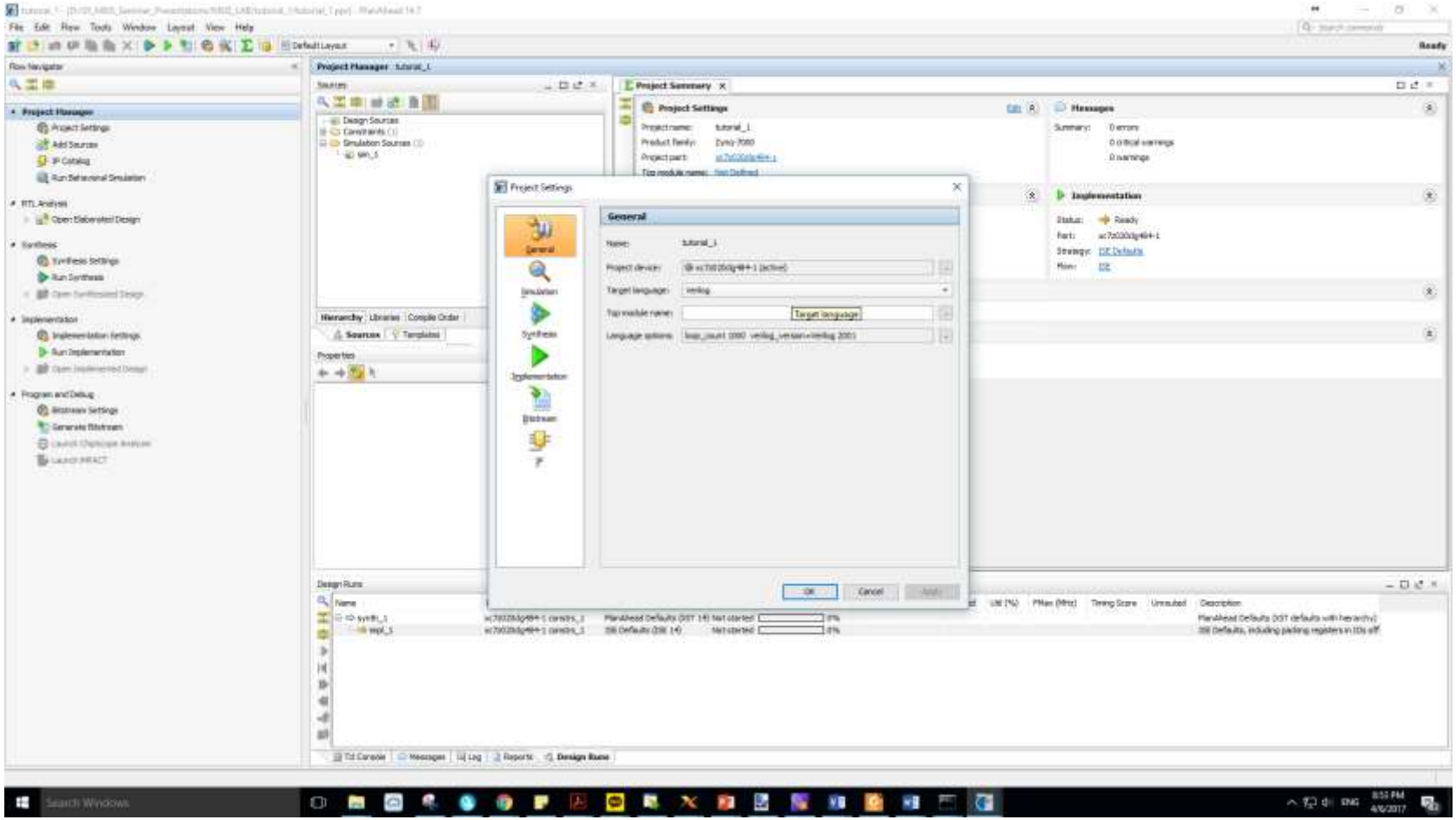
Create a New Zynq Project in PlanAhead (PA)

Before we can begin, we will verify the HDL language settings in the **Tools**, **Project Settings** menu.



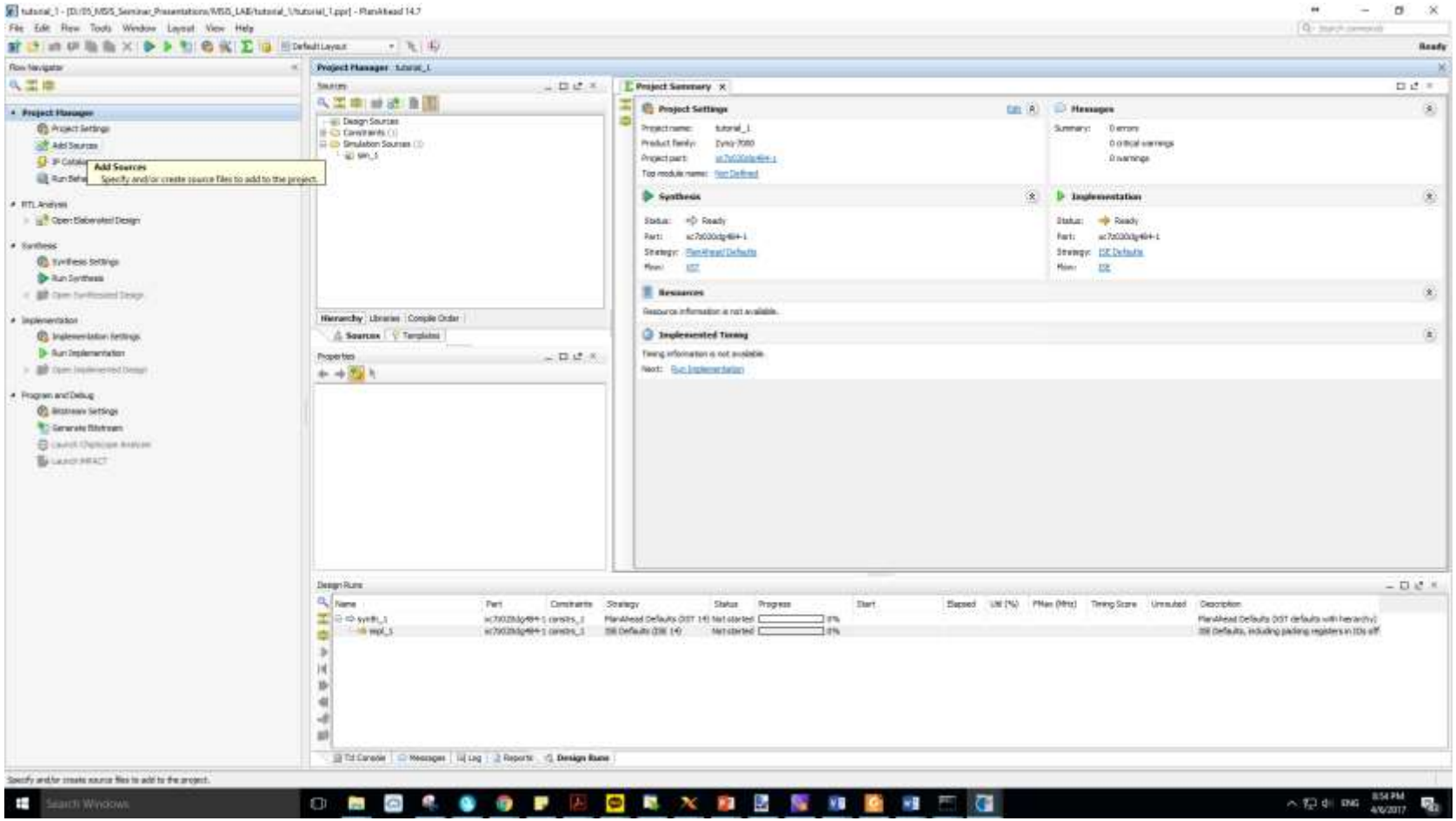
Create a New Zynq Project in PlanAhead (PA)

Make sure that the Target language is set to either **Verilog** or **VHDL**, whichever you pr efer. In this tutorial I will assume you selected **Verilog** here. Click **OK** to continue.



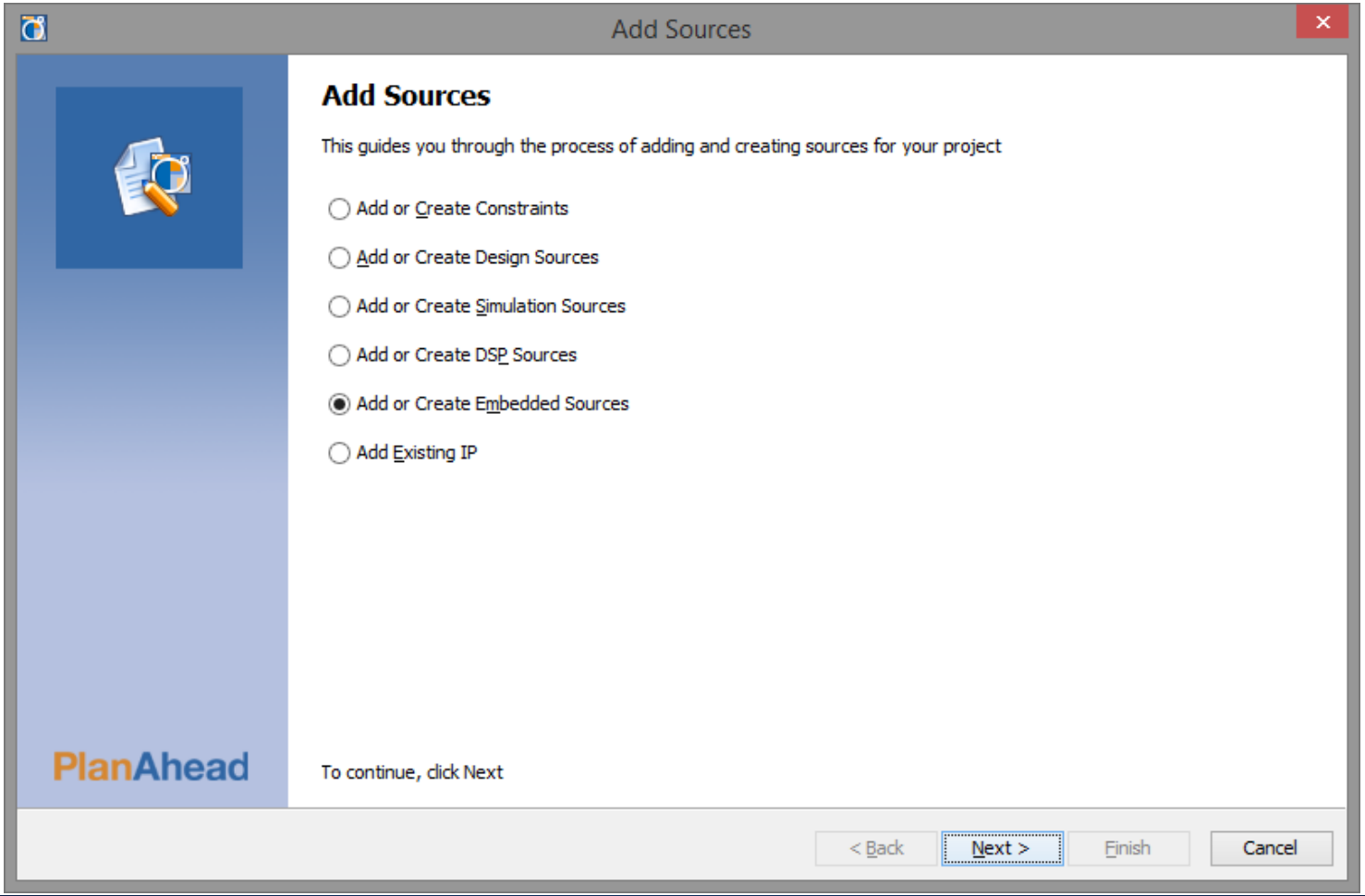
Add an Embedded Source and Launch XPS

Next we will add some sources to the project. Select **Add Sources** in the Project Mana ger tab of the Flow Navigator on the left..



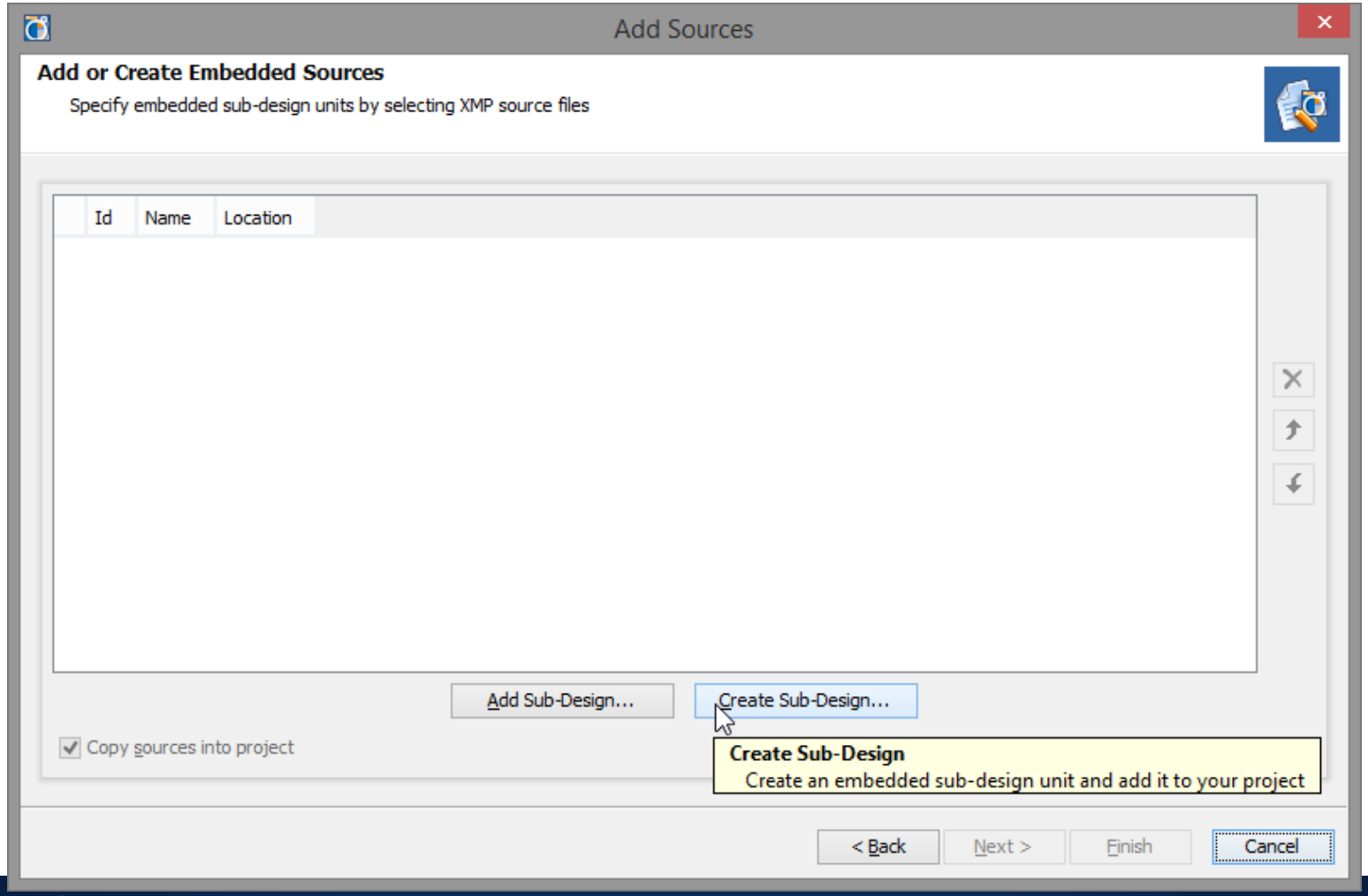
Add an Embedded Source and Launch XPS

Select **Add or Create Embedded Sources** and click **Next** to continue.



Add an Embedded Source and Launch XPS

On the following screen, click **Create Sub-Design**.



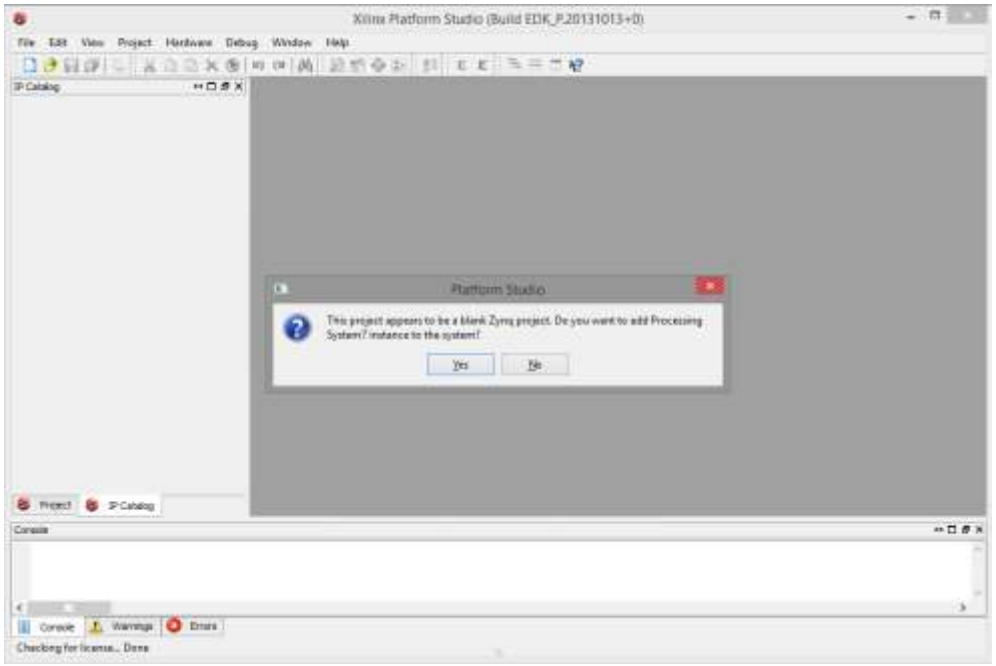
Add an Embedded Source and Launch XPS

Enter the name of the new module. I'll use the default, **module-1** here.

You will find the new module and its location in the list. Click Finish to be taken to **Xilinx Platform Studio (XPS)**

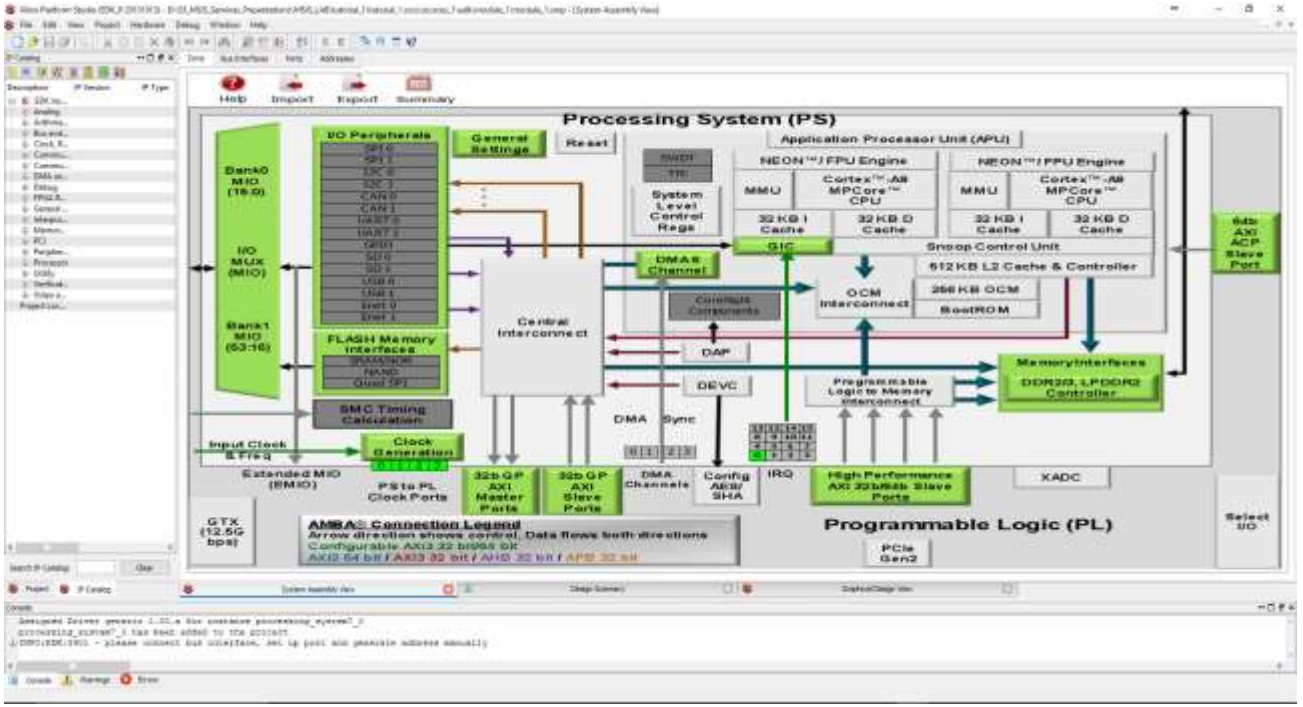
.

After XPS has opened, you'll be asked to ad d a new Processing System7 instance to th e otherwise empty board. Confirm to do so by clicking **Yes**.



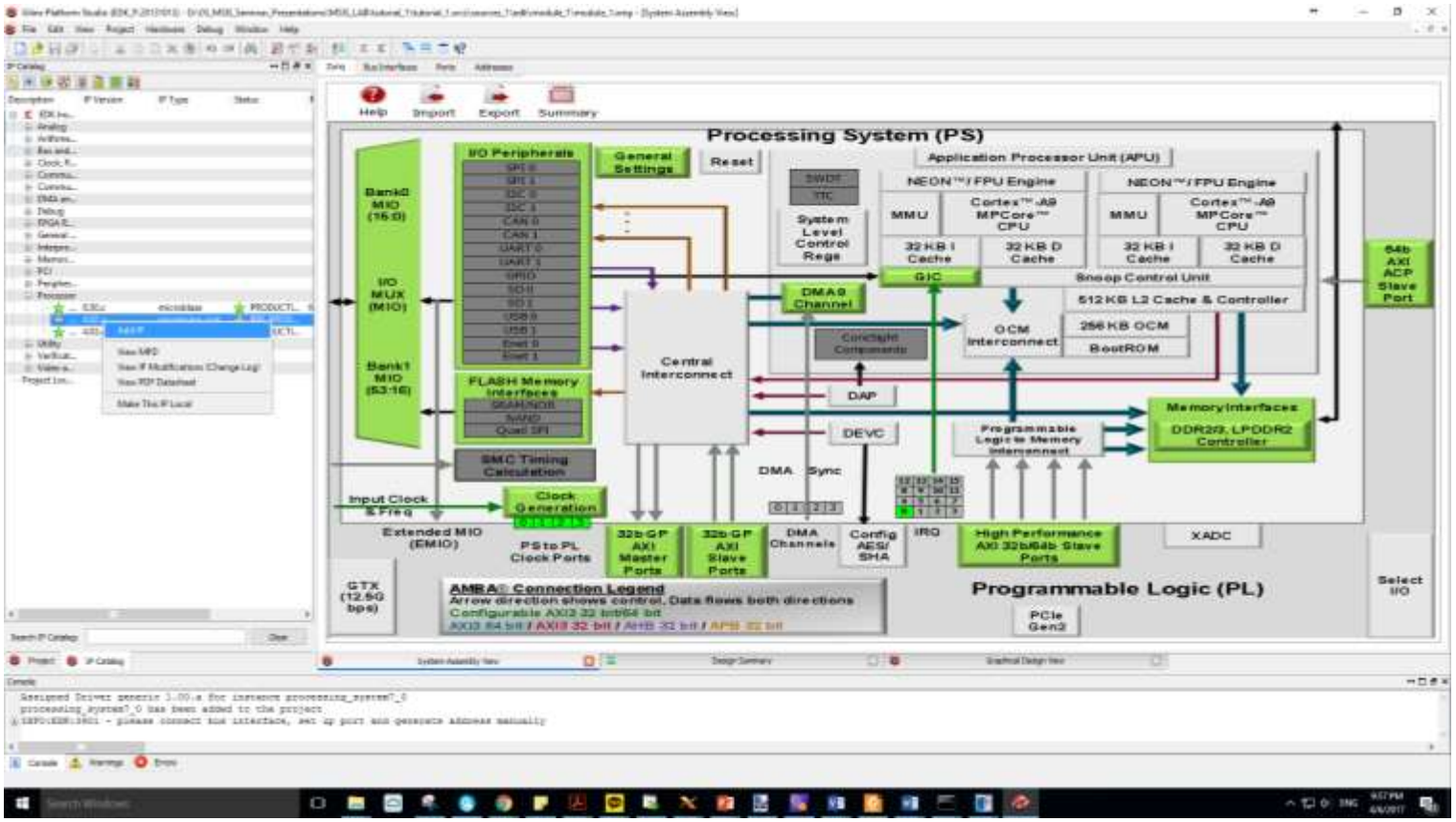
Add an Embedded Source and Launch XPS

You will be presented with the Zynq tab of the System Assembly View. This is the place to configure the Zynq peripherals like the interrupt and memory controllers, clock gen erators etc. We won't use any of that in this tutorial.
Within the Zynq tab, click the Import button to import a board description.



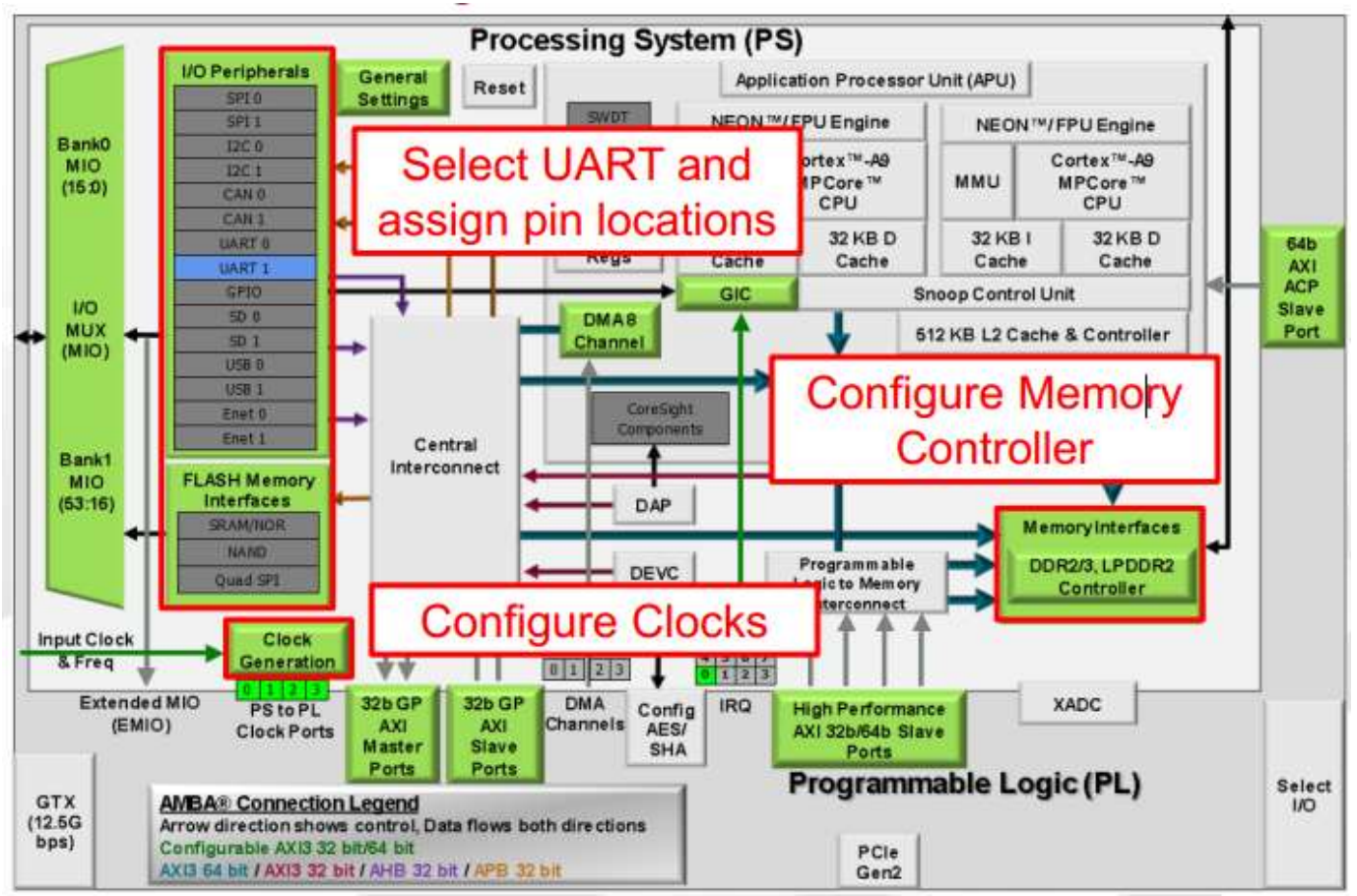
Add an Embedded Source and Launch XPS

You Also can add pro cessor IP as shown



LAB 2: PS CONFIG PART 1 – HELLO WORLD

PS Configurations

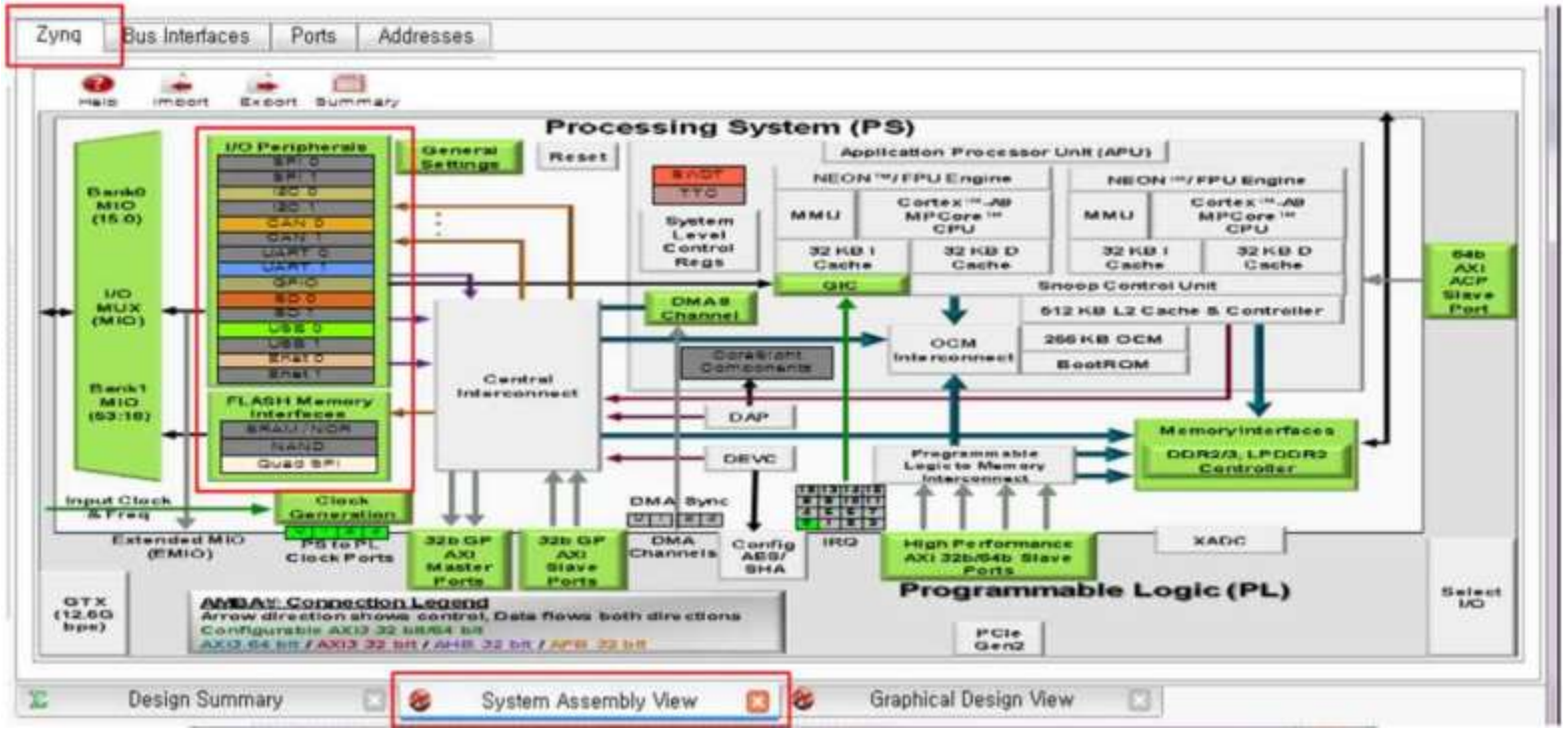


Objectives

- When you have completed Lab 2, you will know how to do the following:
  - Enable and map a Zynq PS UART peripheral
  - Configure Memory and Clocks for the Zynq PS
  - Build the hardware platform and export to SDK
  - Create and run a Hello World application

Enable and map a Zynq PS UART peripheral

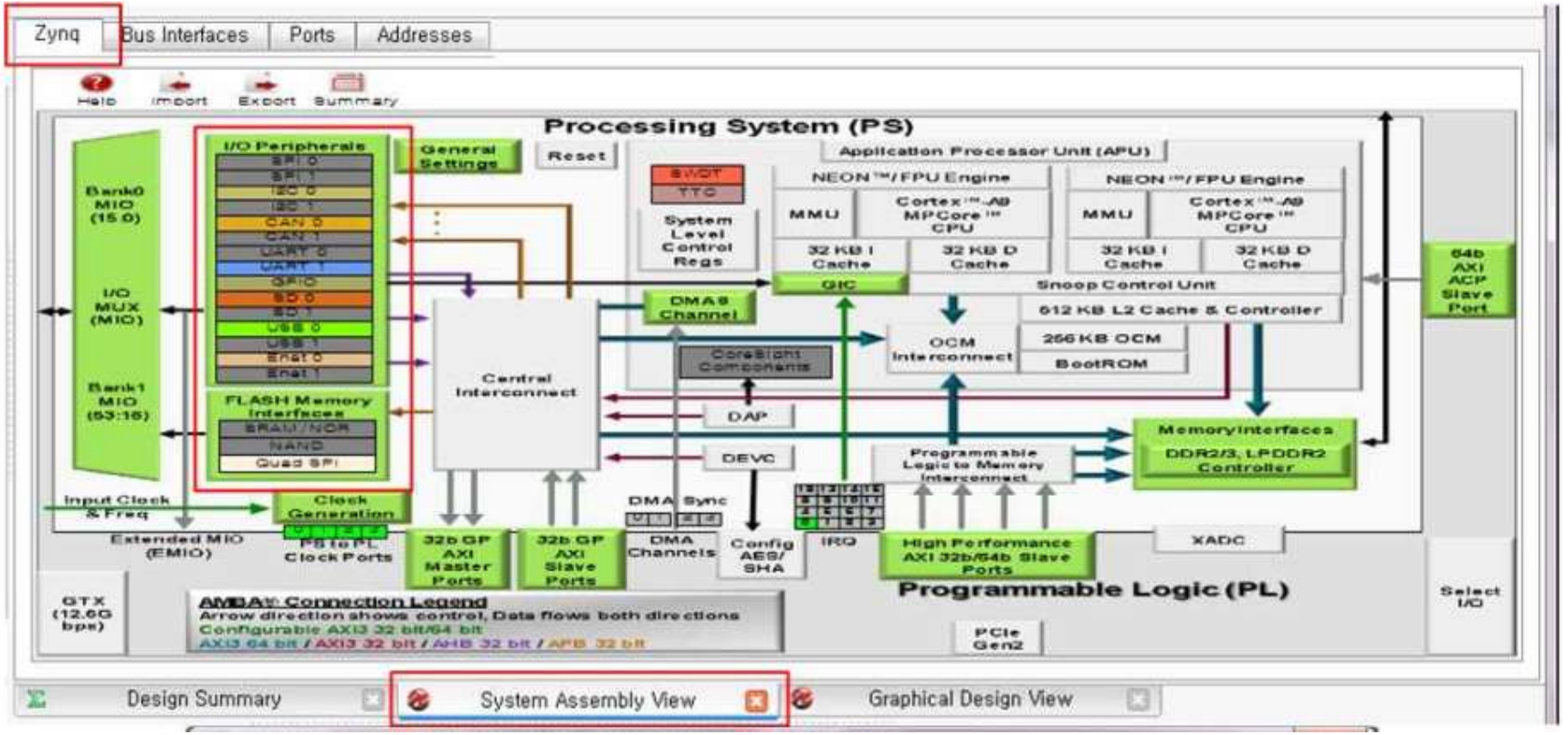
- At the conclusion of Lab 1, an ARM Processing System was added to the Embedded Source.
- The Zynq configuration tool became visible in the **System Assembly View: Zynq** tabs.
- We'll do something very simple in this step by enabling a single UART peripheral in the design an d mapping it to the Multiplexed I/O (MIO).



Zynq Hardware Configuration in XPS

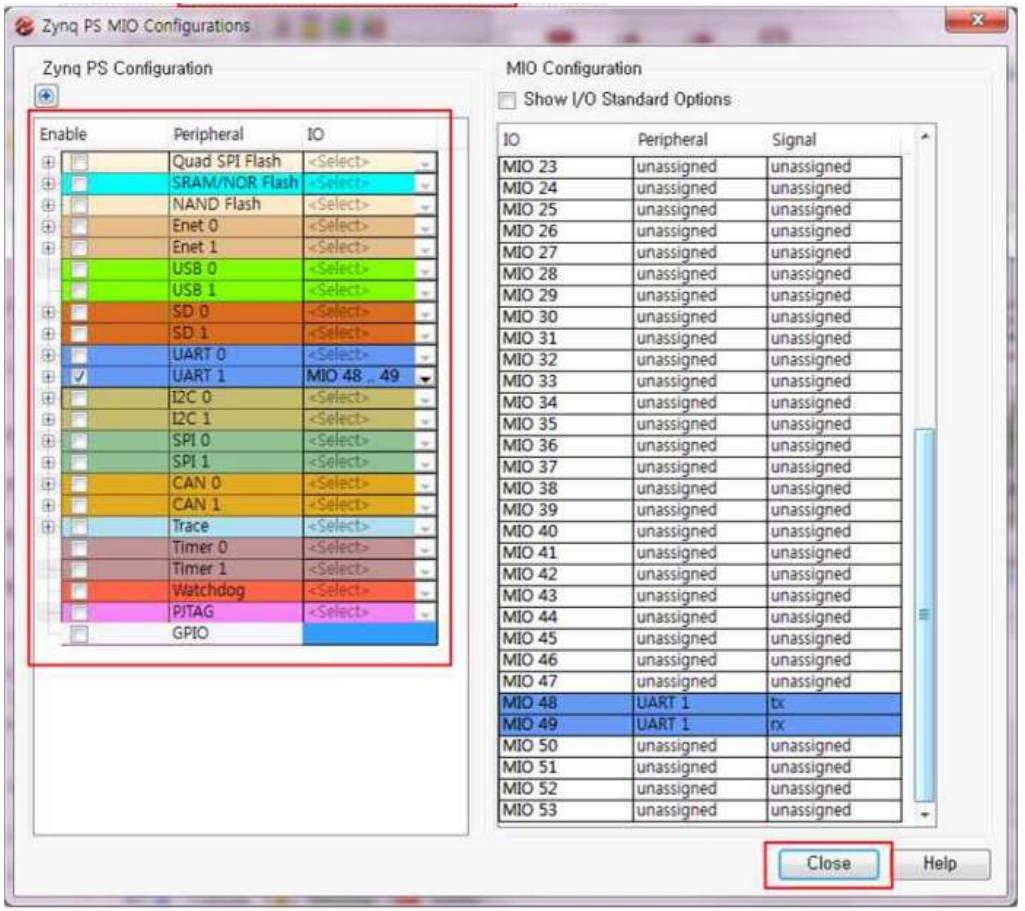
Enable and map a Zynq PS UART peripheral

Zynq PS MIO Configuration



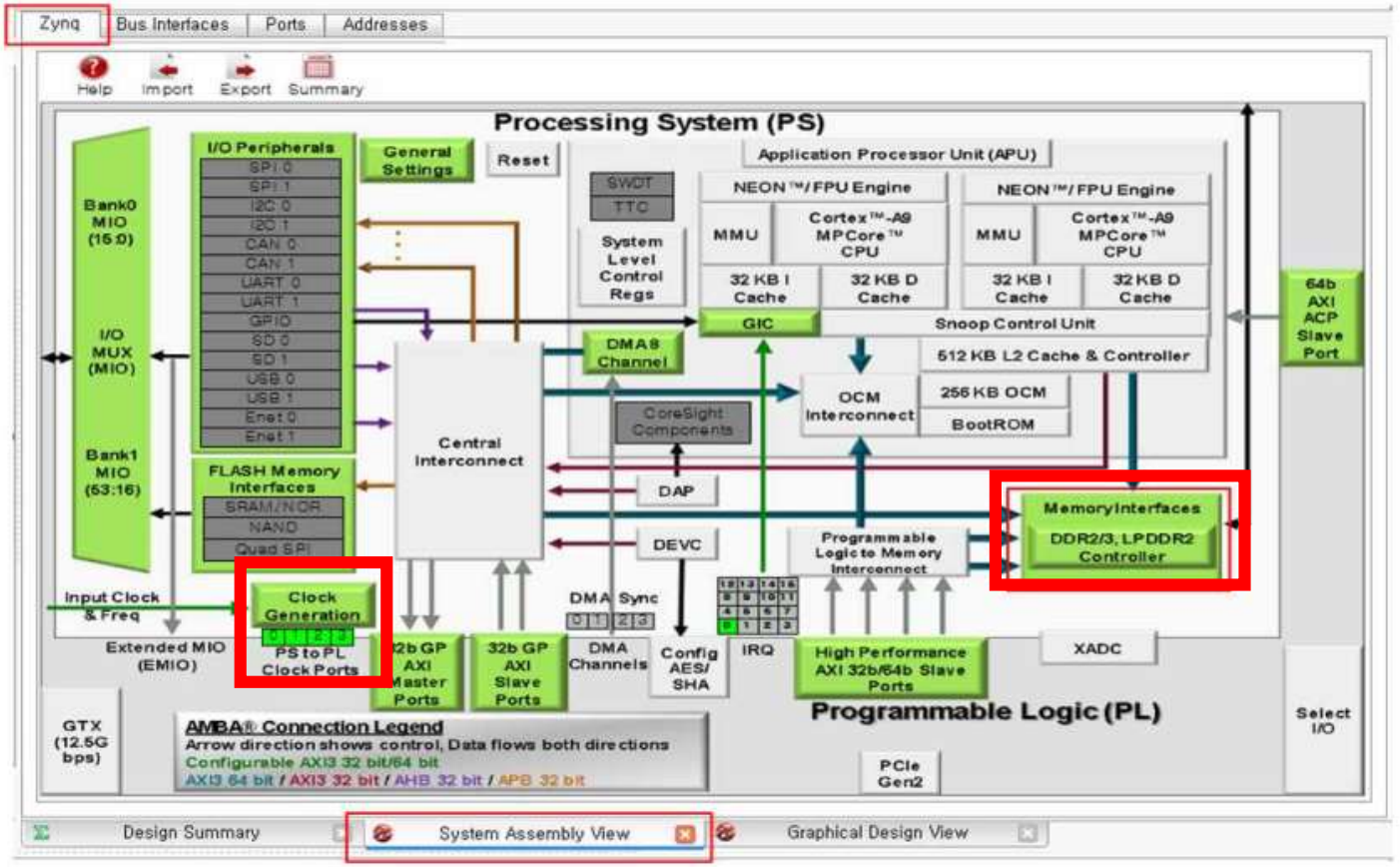
Enable and map a Zynq PS UART peripheral

Select UART 1



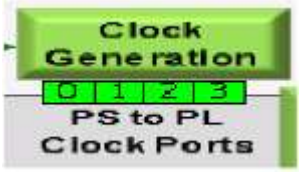
Configure Memory and Clocks for the Zynq PS

- A few critical Zynq PS elements must be configured before even a simple Hello World can be run.
- This includes the DDR3 memory, as it is the RAM that will execute the Zynq PS applications. Also, the system clocks must be configured correctly.

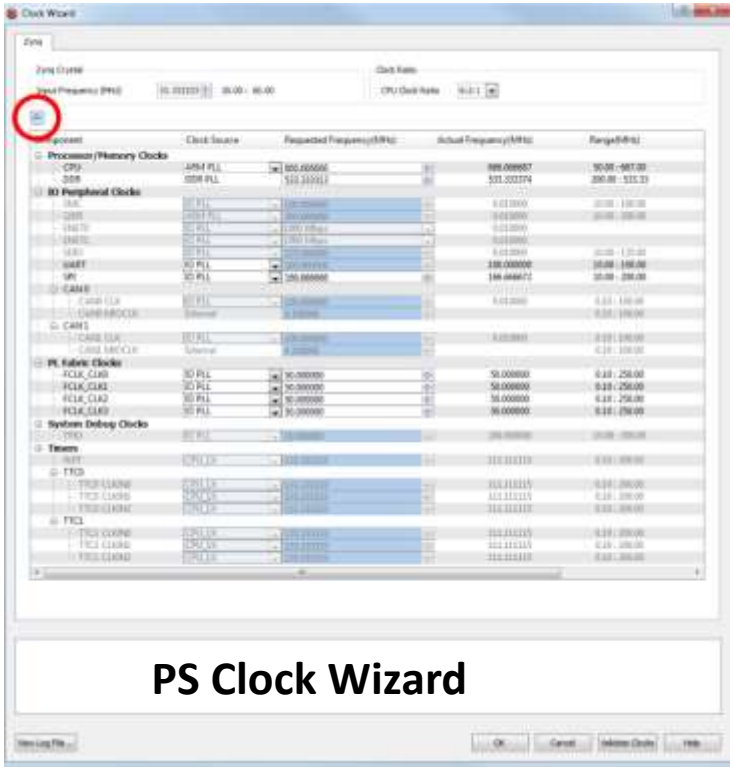


Configure Memory and Clocks for the Zynq PS

PS Clock Generation

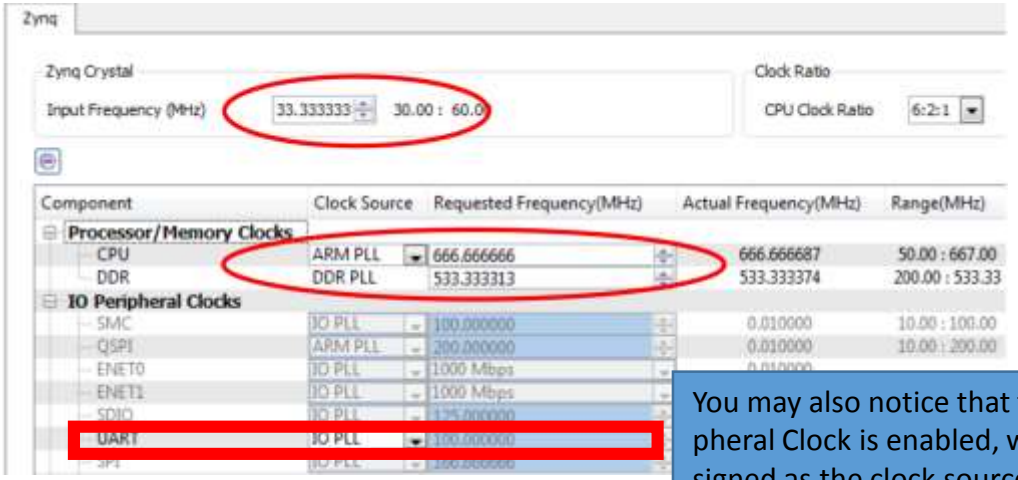


- Click on the box for **Clock Generation**.
- Expand all the clocks, You should see the same view as shown in Figure



PS Clock Wizard

- **Verify the following:**
  - Input frequency is **33.33333** MHz
  - CPU frequency is **666.666666** MHz
  - DDR frequency is **533.333313** MHz



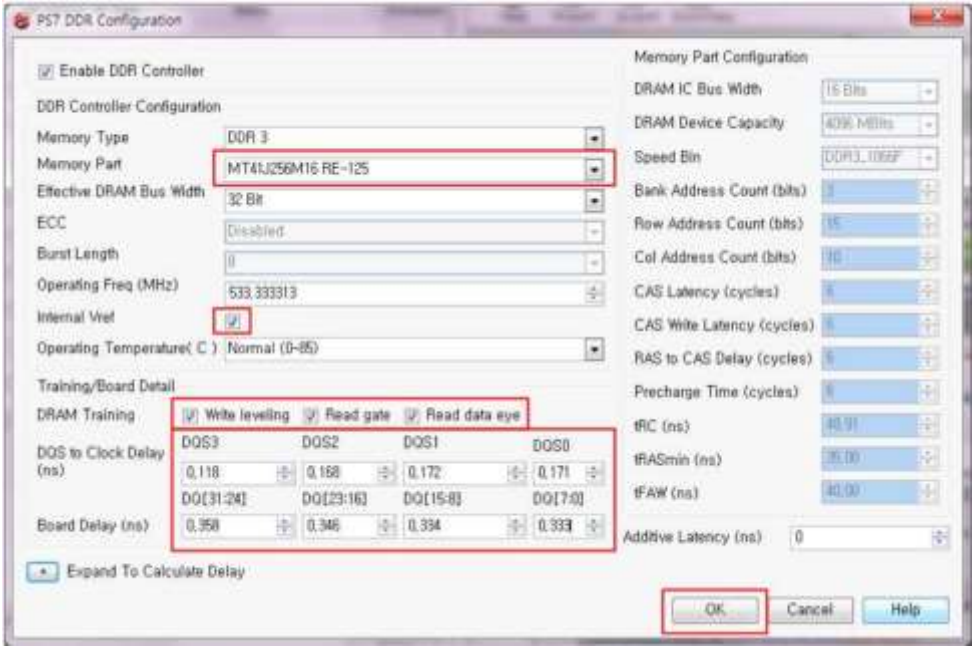
You may also notice that the UART IO Peri pheral Clock is enabled, with the IO PLL as signed as the clock source. However, the UART Peripheral Clock is fixed at 100 MHz.

Configure Memory and Clocks for the Zynq PS

- **Next, we will configure the memory controller.**
  - Click on the box labeled Memory Interfaces.

Launch PS7 DDR Configuration

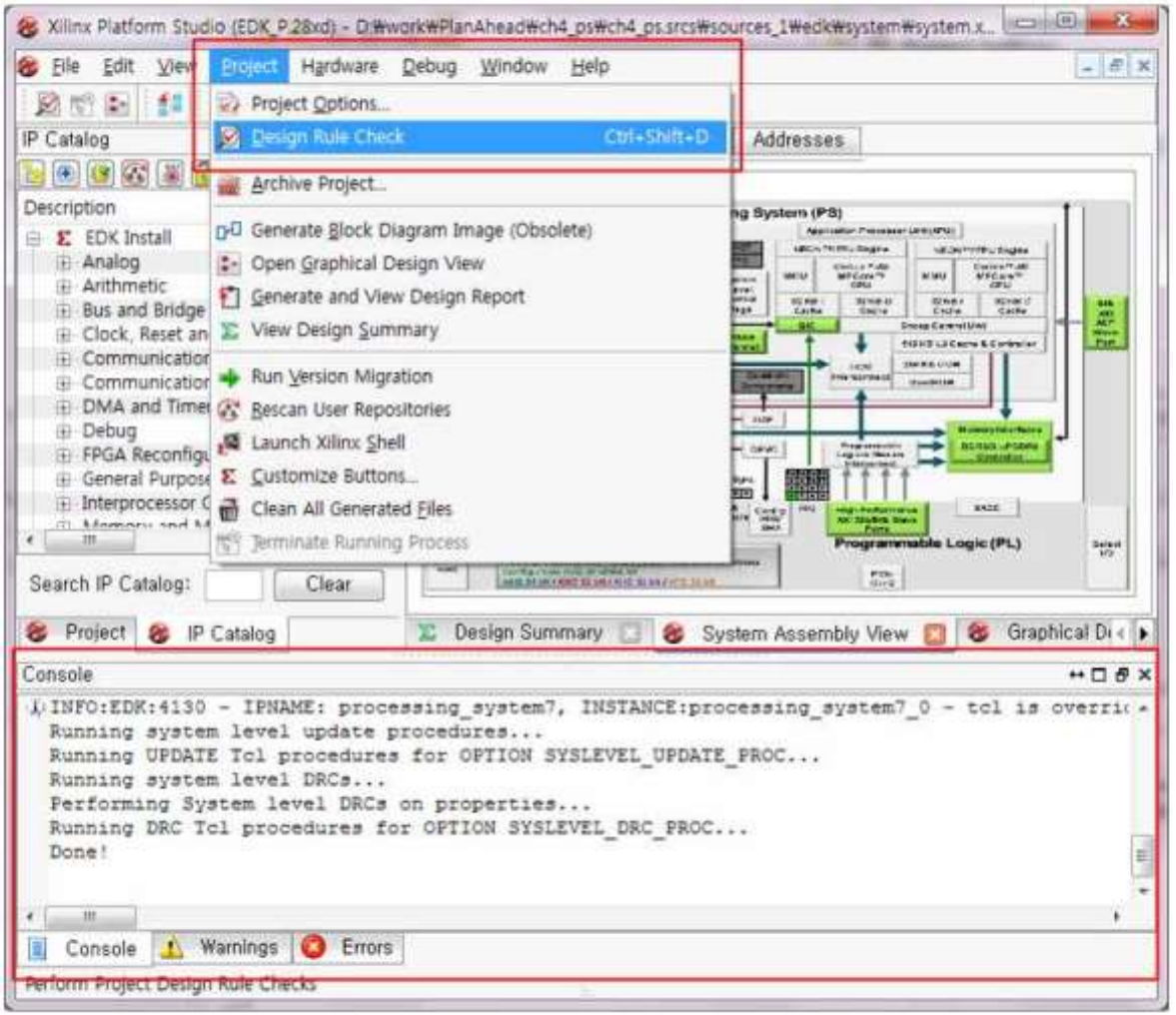
The PS7 DDR Configuration screen allows for configuration of the DDR Controller.



Set Up Settings for PS7 DDR Configuration

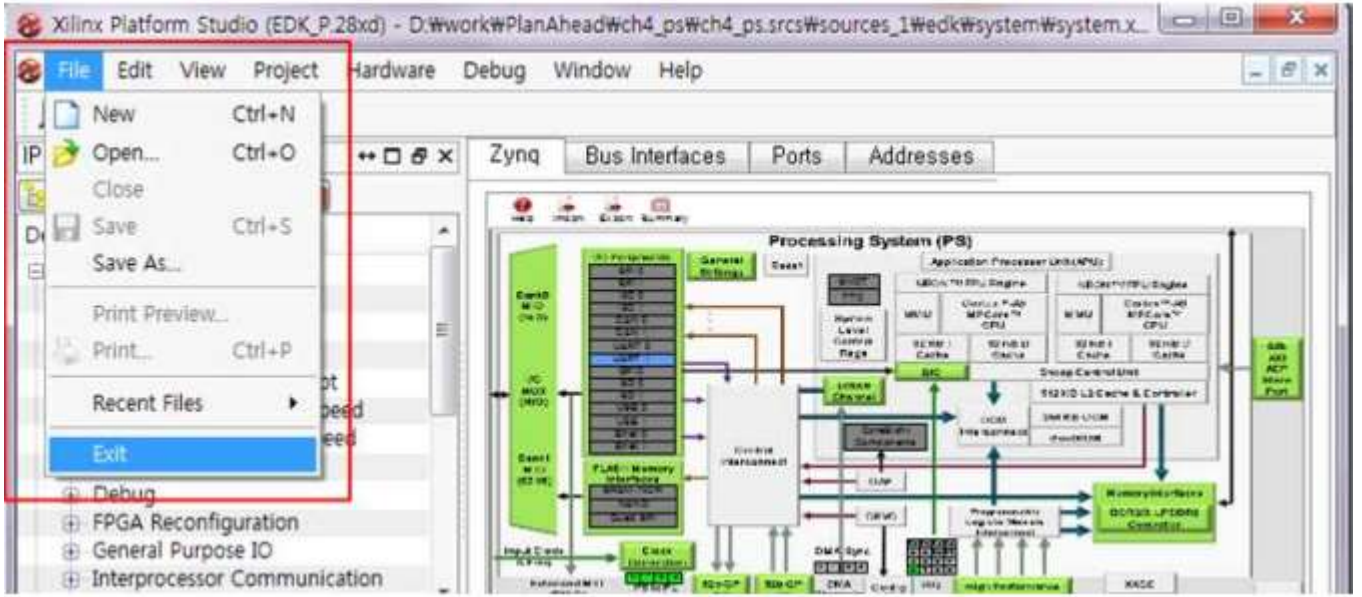
Finalize Setting up Zynq Processor

Project > Design Rule Check



Finalize Setting up Zynq Processor

File > Exit

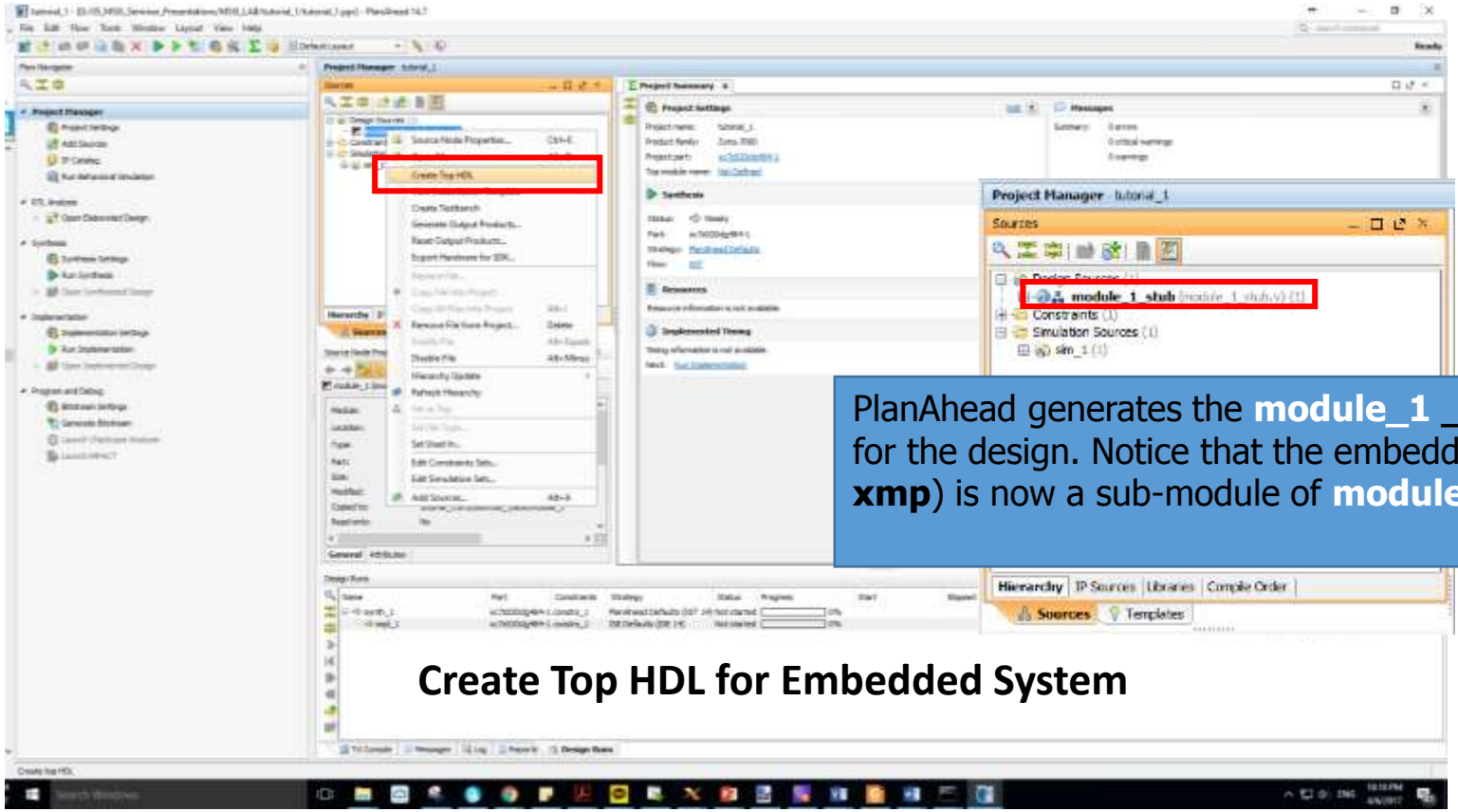


Build the hardware platform and export to SDK

- A basic ARM hardware platform is now configured.
  - The configuration includes clock and DDR controller settings.
  - It also enables and maps a UART peripheral.
- Now we'll build the hardware platform and export to the Software Develop ment Kit (SDK) so that an application can be developed.

Build the hardware platform and export to SDK

From PalanAhead "Project Manager – Sources" - right-click on **module_1** and select **Create Top HDL**.



PlanAhead generates the **module_1 _stub.v** top-level module for the design. Notice that the embedded system (**module_1 . xmp**) is now a sub-module of **module_1 _stub**.

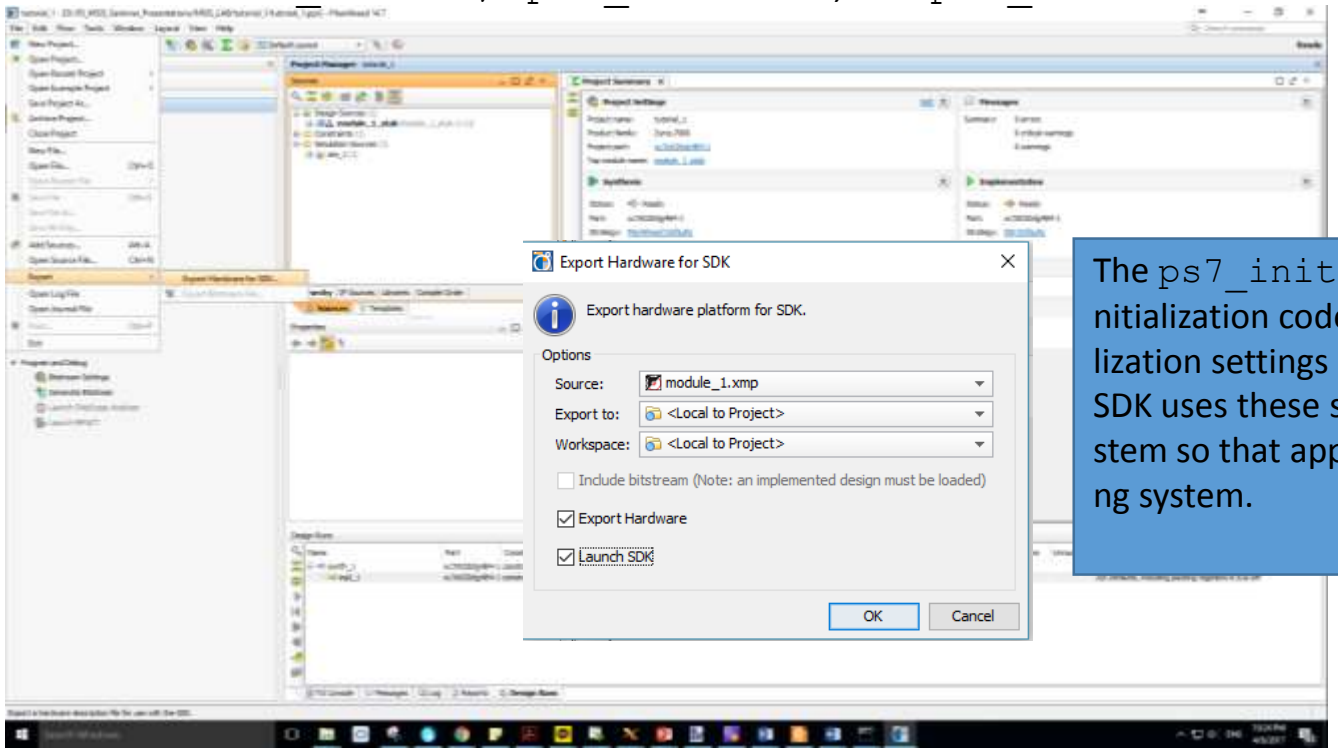
Create Top HDL for Embedded System

Build the hardware platform and export to SDK

File > Export > Export Hardware for SDK

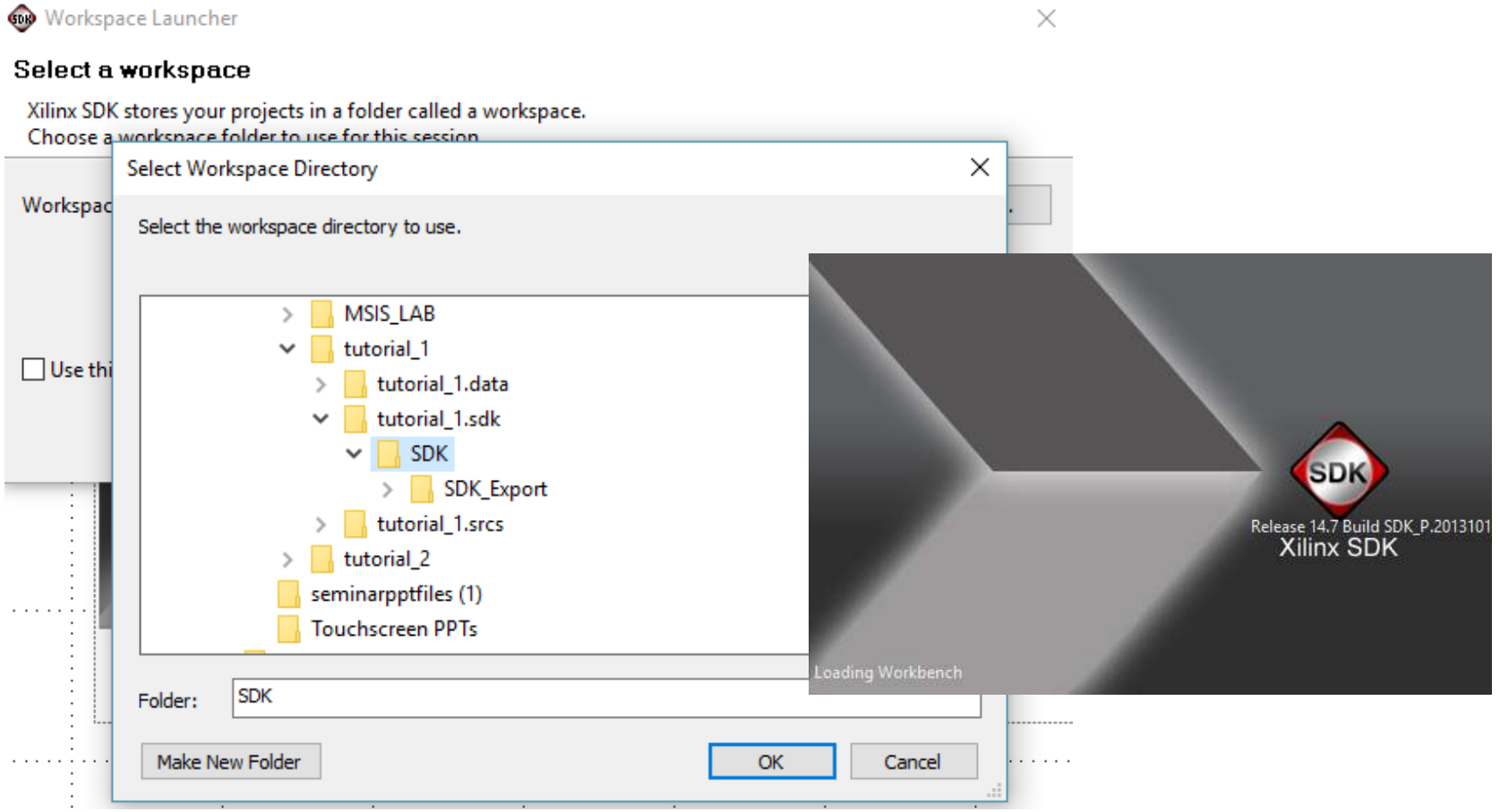
The PlanAhead design tool exported the Hardware Platform Specification for your design (**module_1.xml** in this example) to SDK.

In addition to system.xml, there are four more files exported to SDK. They are `ps7_init.c`, `ps7_init.h`, `ps7_init.tcl`, and `ps7_init.html`.



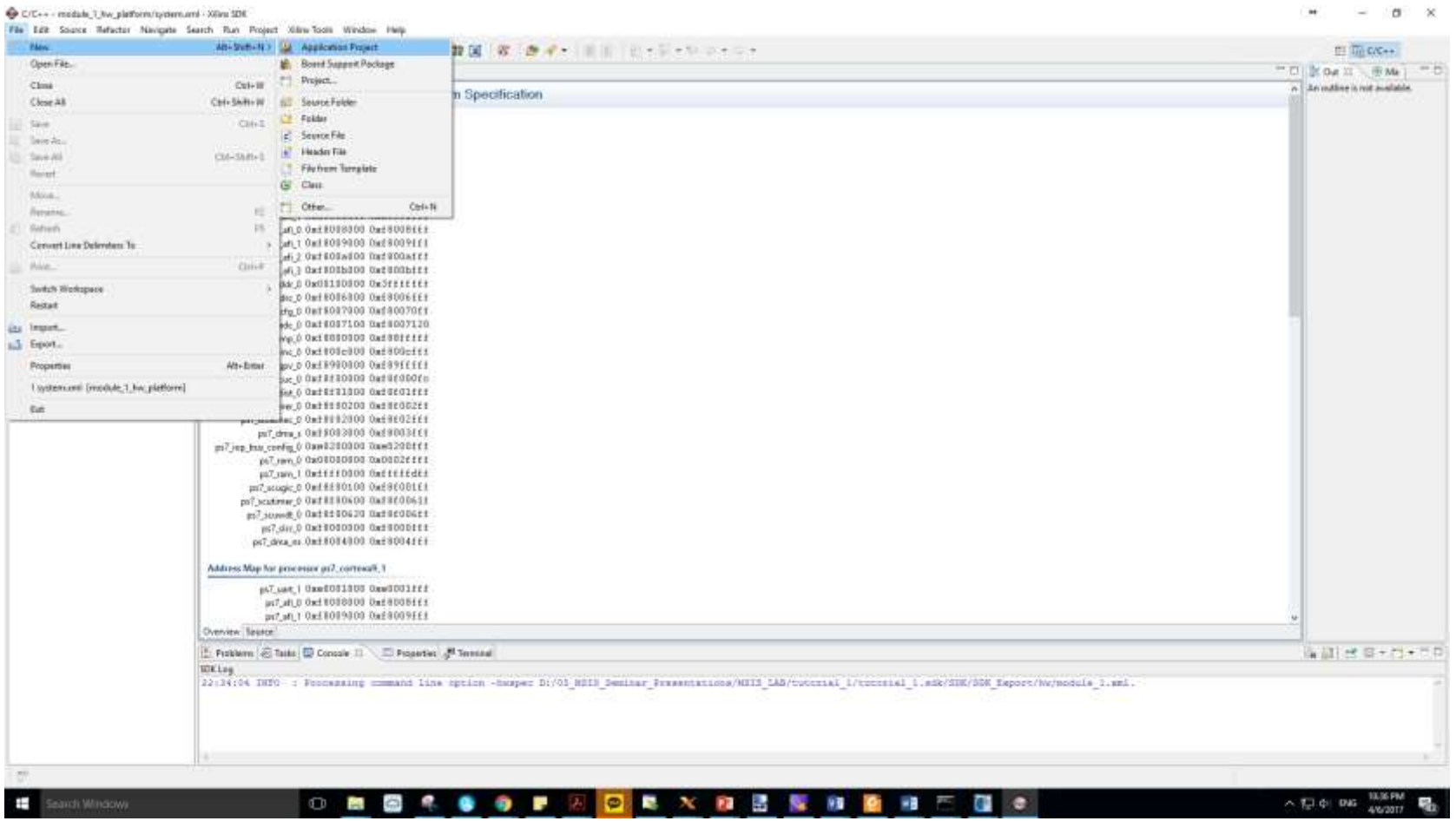
The `ps7_init.c` and `ps7_init.h` files contain the initialization code for the Zynq Processing System and initialization settings for DDR, clocks, plls, and MIOs.
SDK uses these settings when initializing the processing system so that applications can be run on top of the processing system.

Create and Run a Hello World application

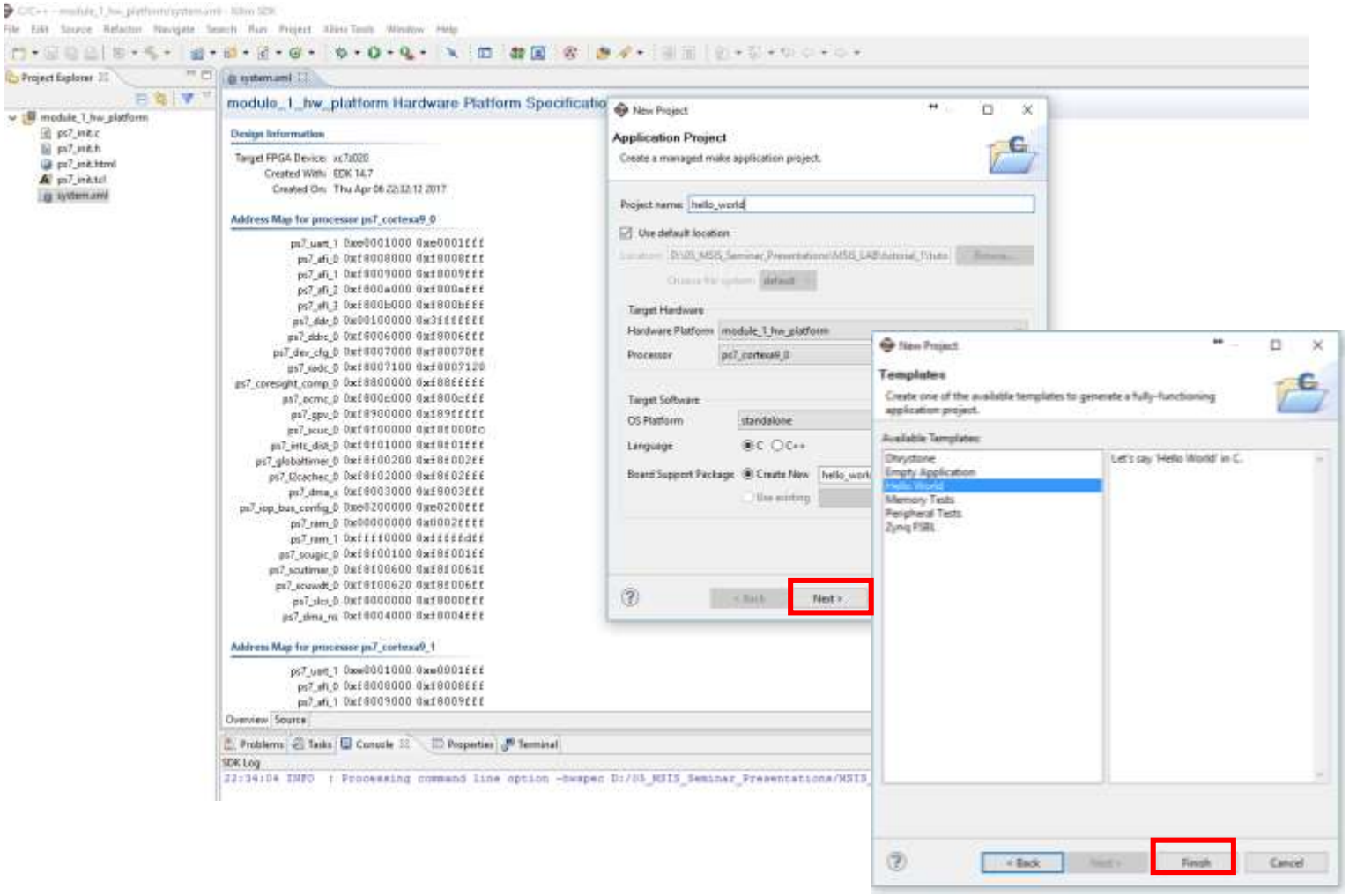


Create and Run a Hello World application

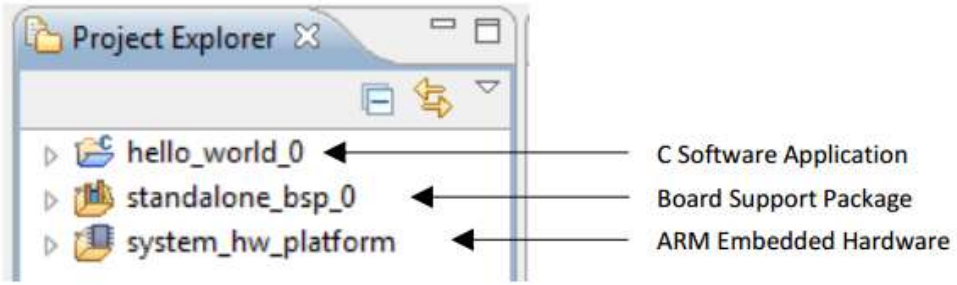
"File > New > Xilinx Application Project



Create and Run a Hello World application



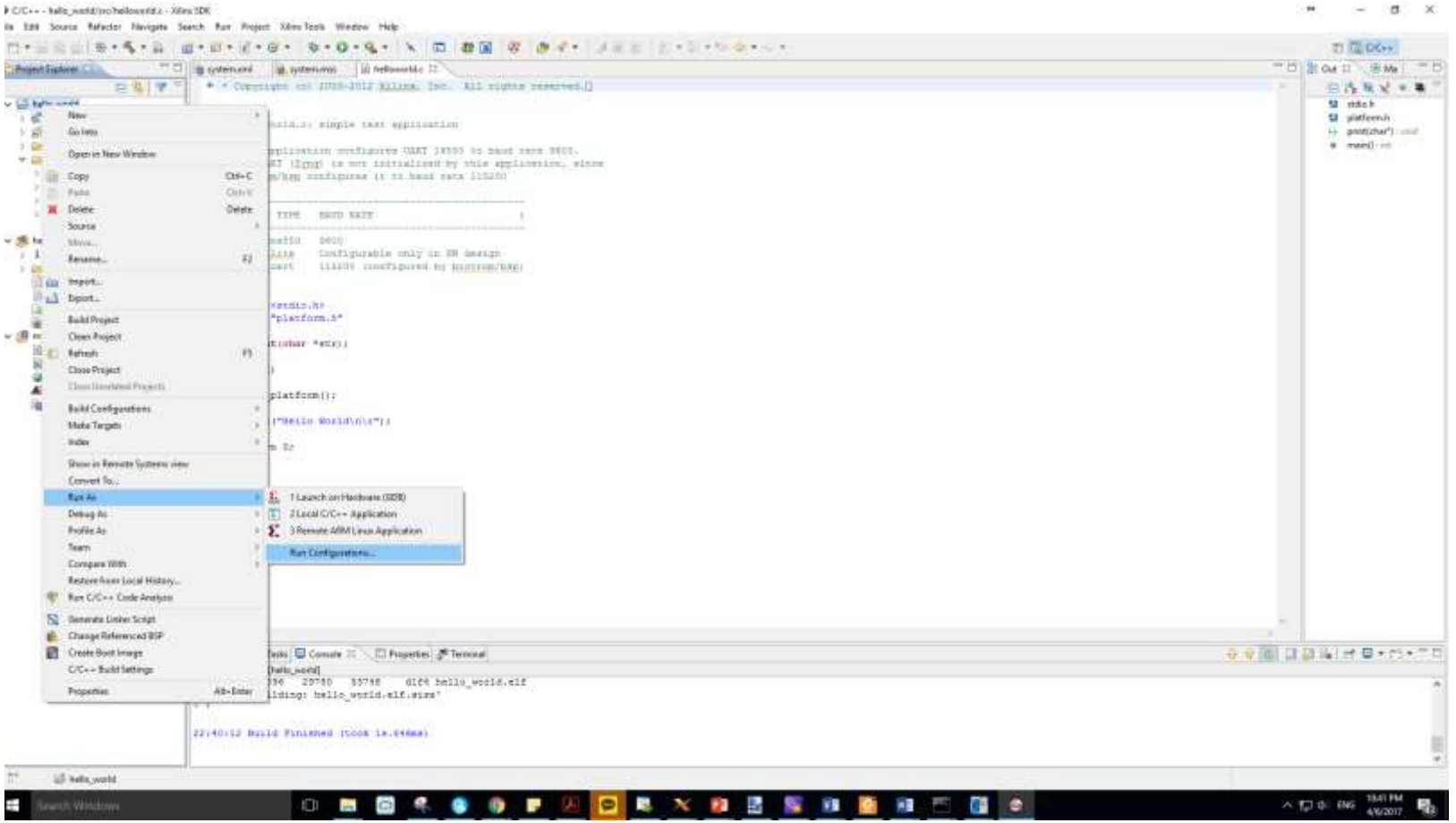
Create and Run a Hello World application



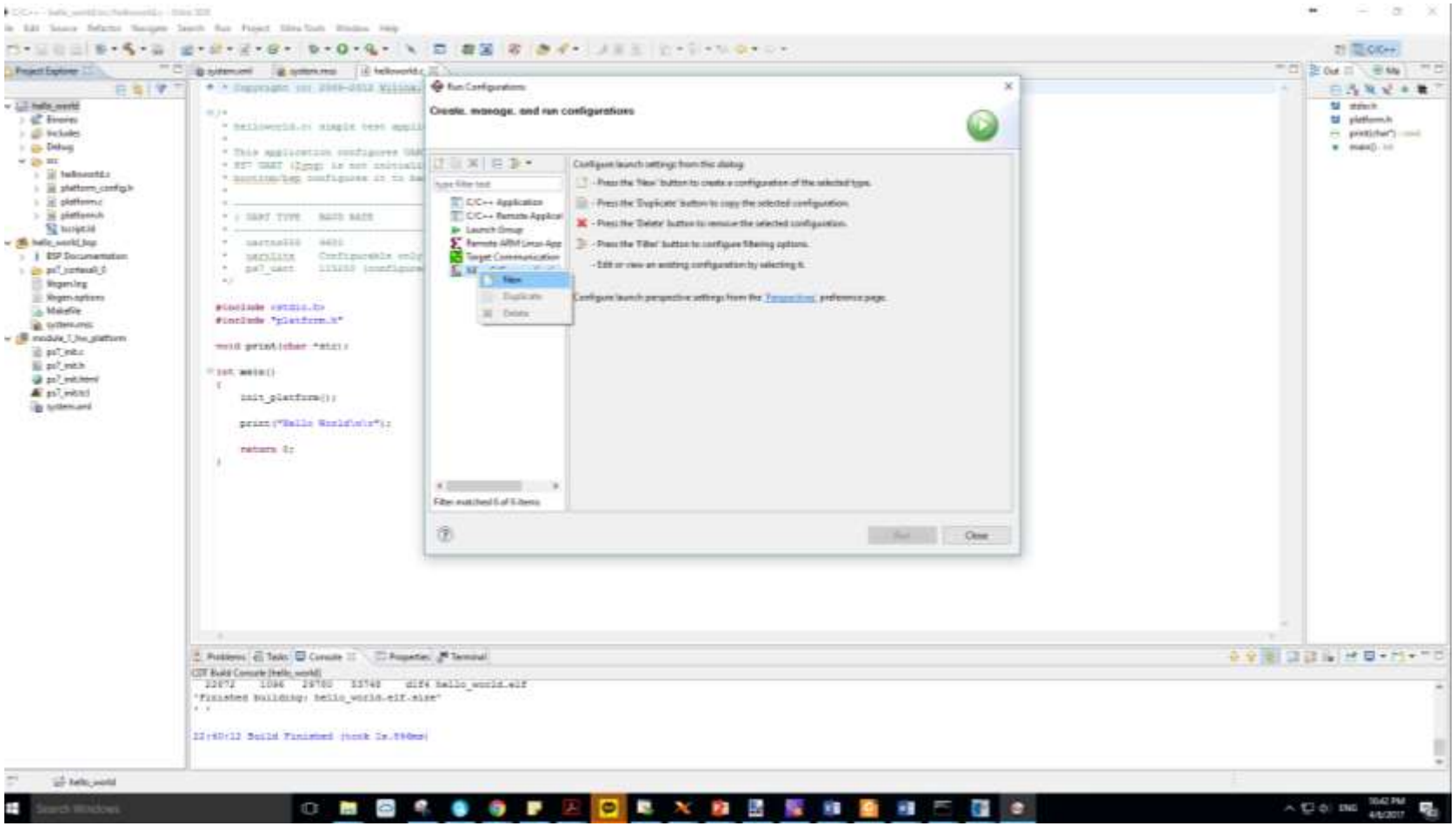
Project Explorer View with Hello World C Application Added

hello_world_0 Application Expanded

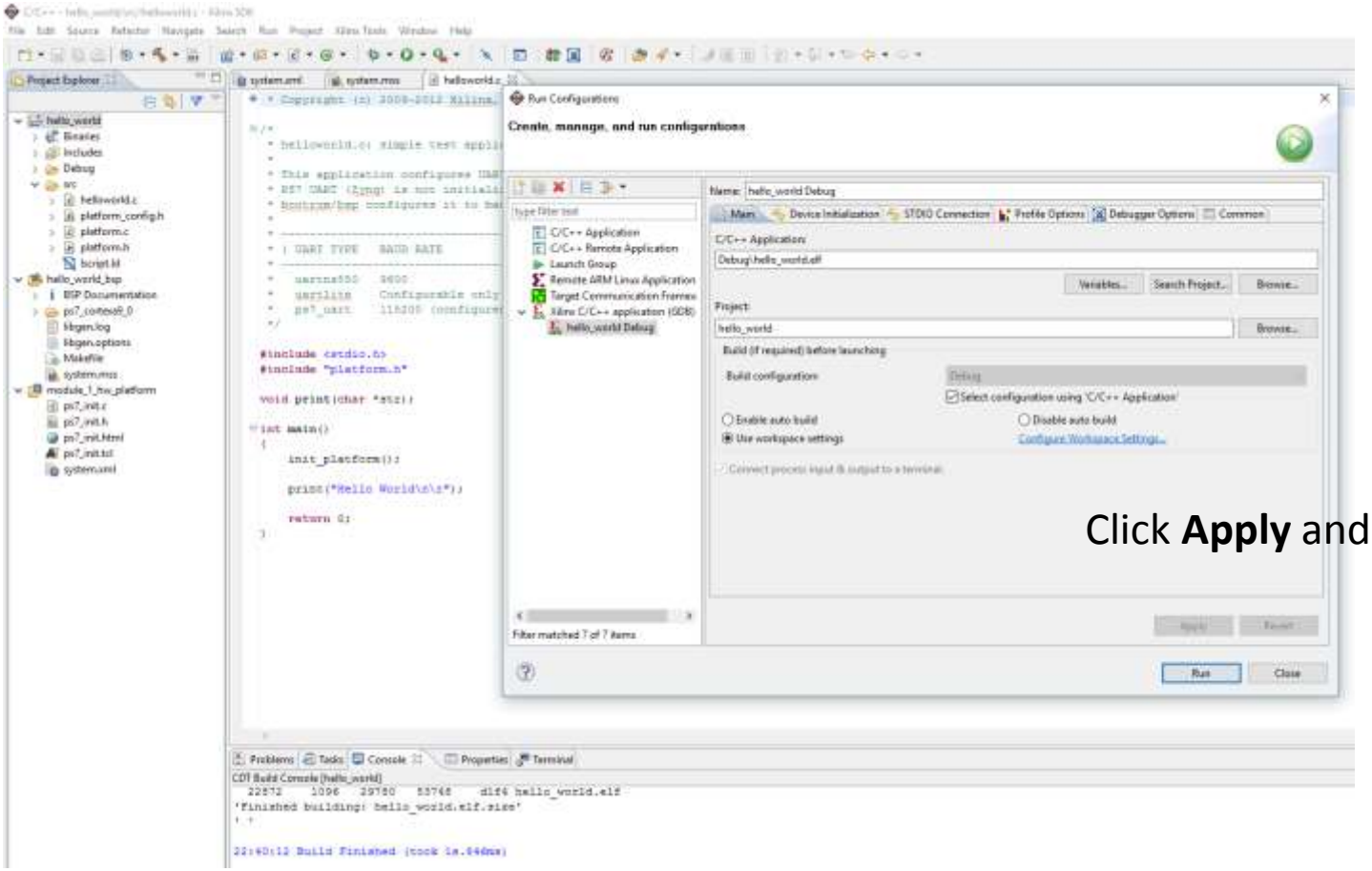
Create and Run a Hello World application



Create and Run a Hello World application



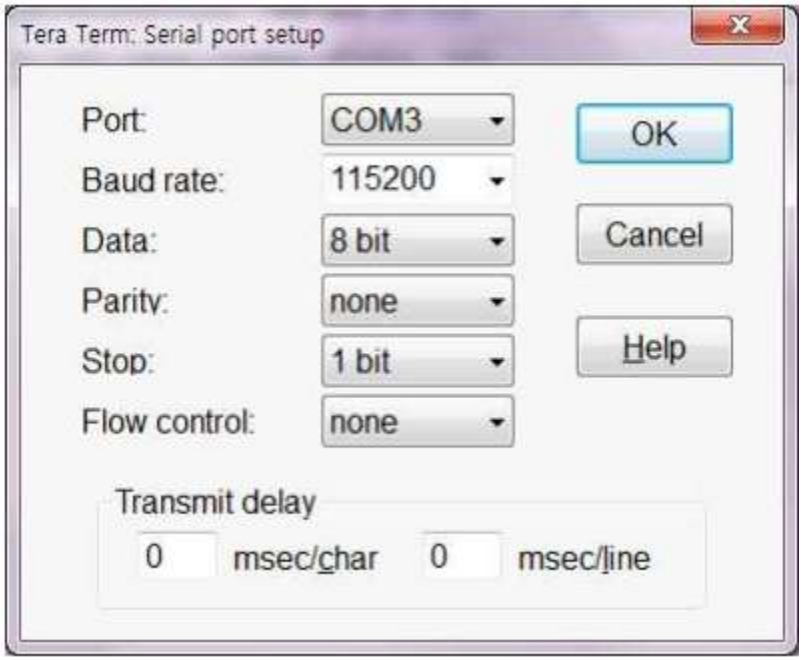
Create and Run a Hello World application



Click **Apply** and then **Run**.

Create and Run a Hello World application

UART Setting



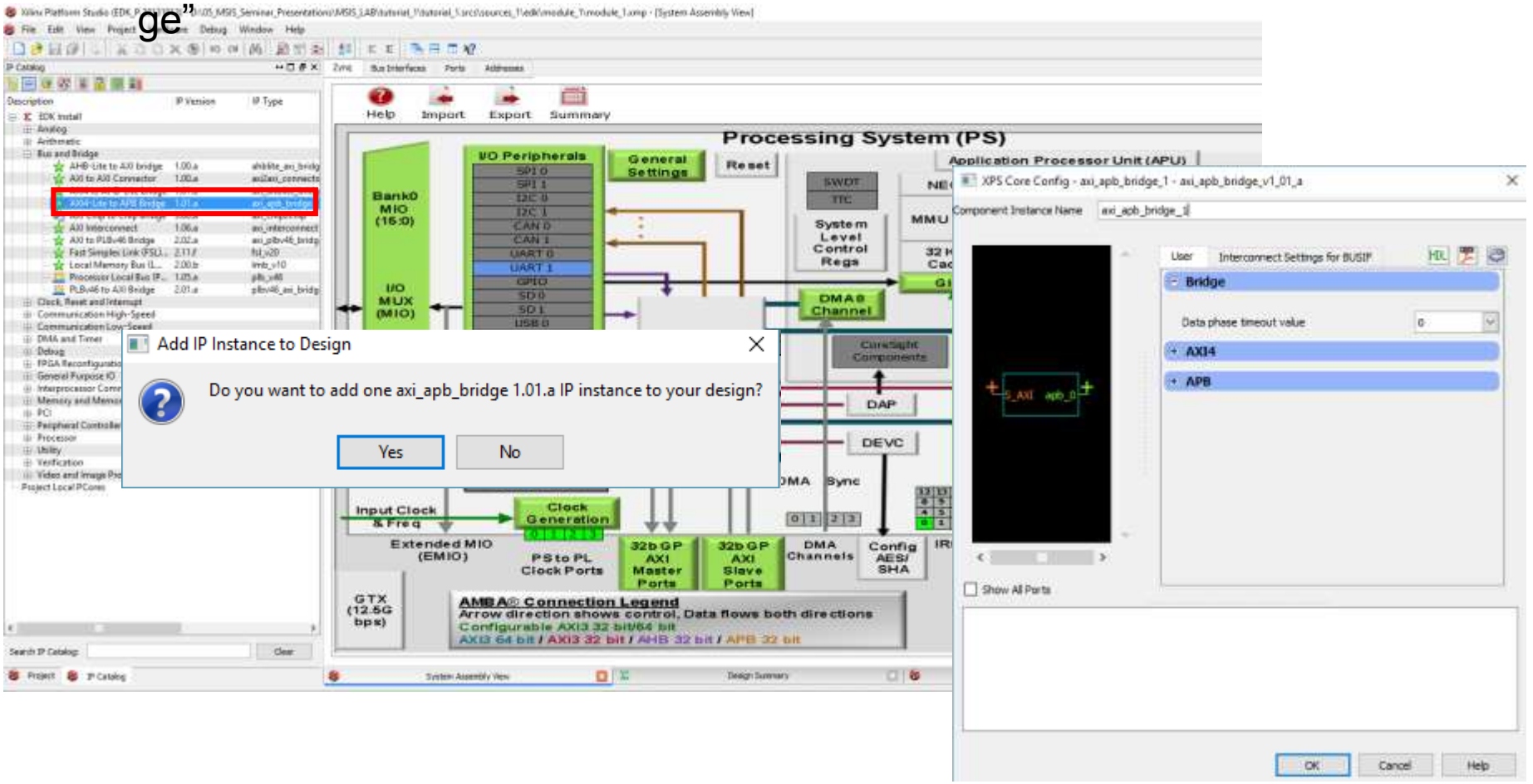
LAB 3: ADDING A CUSTOM PL PERIPHERAL

Objectives

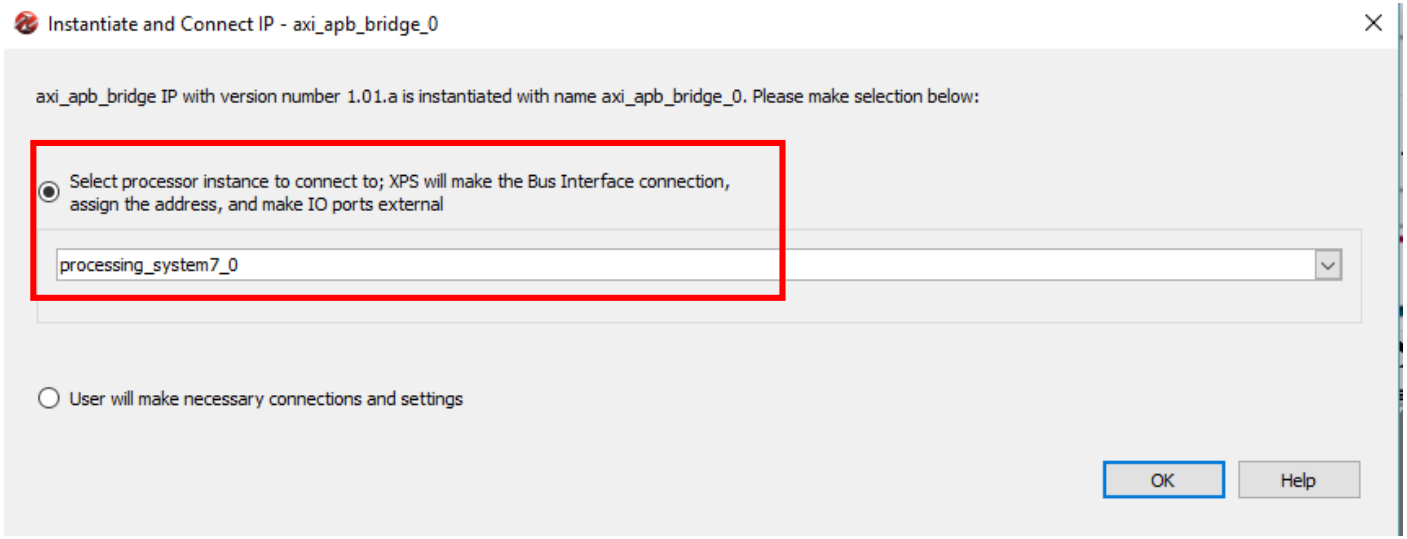
- This tutorial demonstrates how to add a custom PL peripheral to the embe dded system.
- Using the PlanAhead and Xilinx Platform Studio (XPS), this lab will show:
  - Adding a GPIO interface to the Processor Subsystem (PS).
  - Exporting signals, including clocks from the PS to the Programmable Logic subs ection (PL).
  - Modifying the top level HDL module
  - Creating and connecting a new custom HDL module

Add a GPIO Peripheral in XPS

From XPS Select "EDK install > Bus and Bridge > AXI4-Lite to APB Bridge"

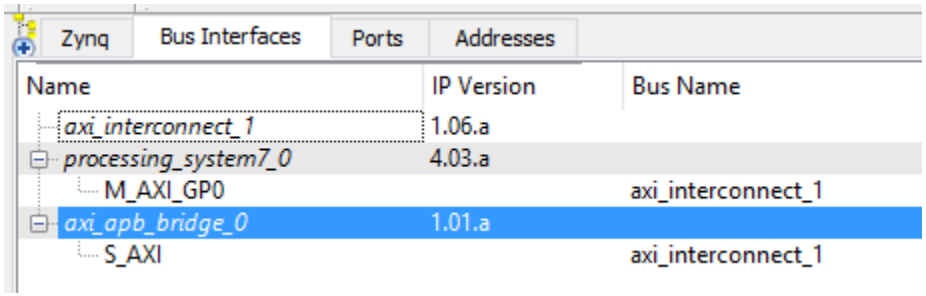


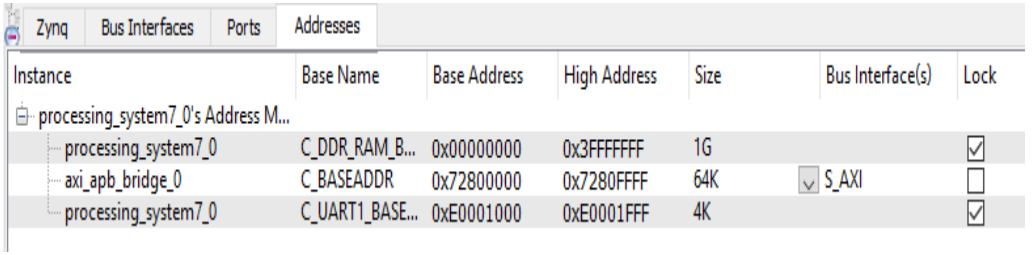
Add a GPIO Peripheral in XPS

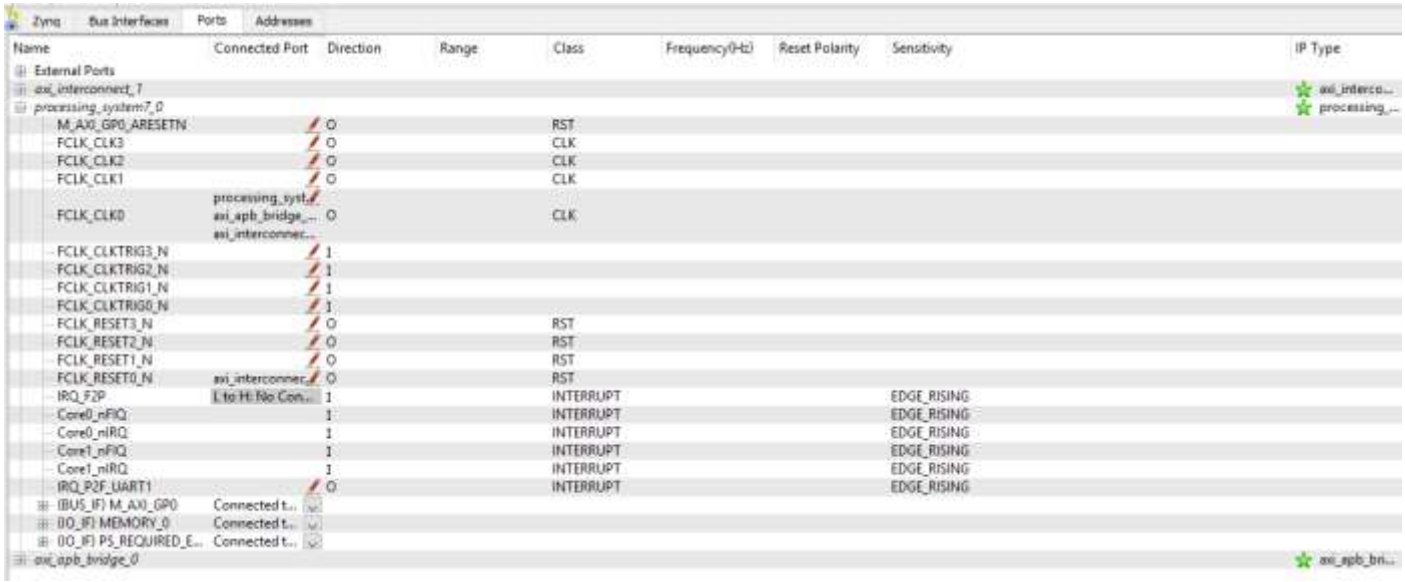


Press OK then Go to “System Assembly View → Addresses”
Default AXI2APB Bridge Address is 0x72800000

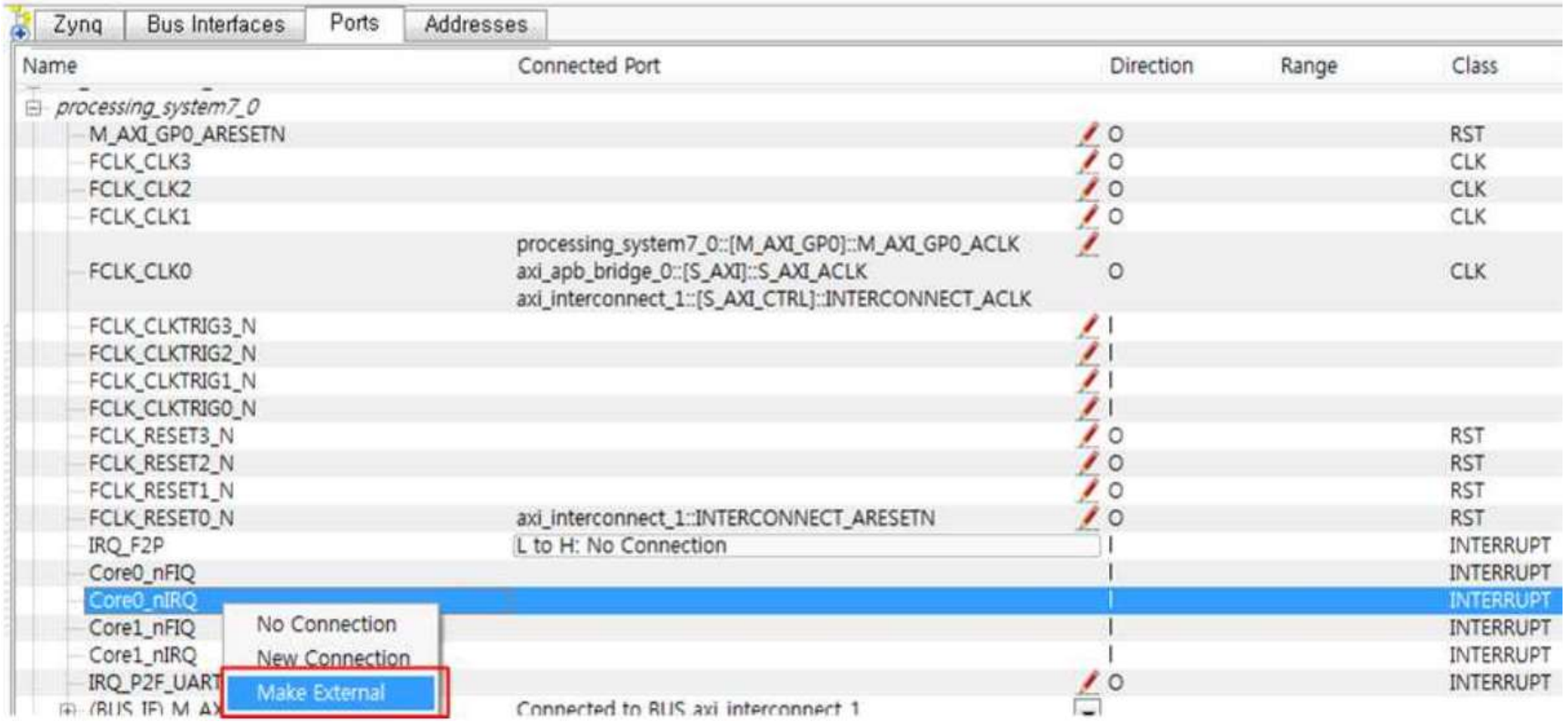
Add a GPIO Peripheral in XPS



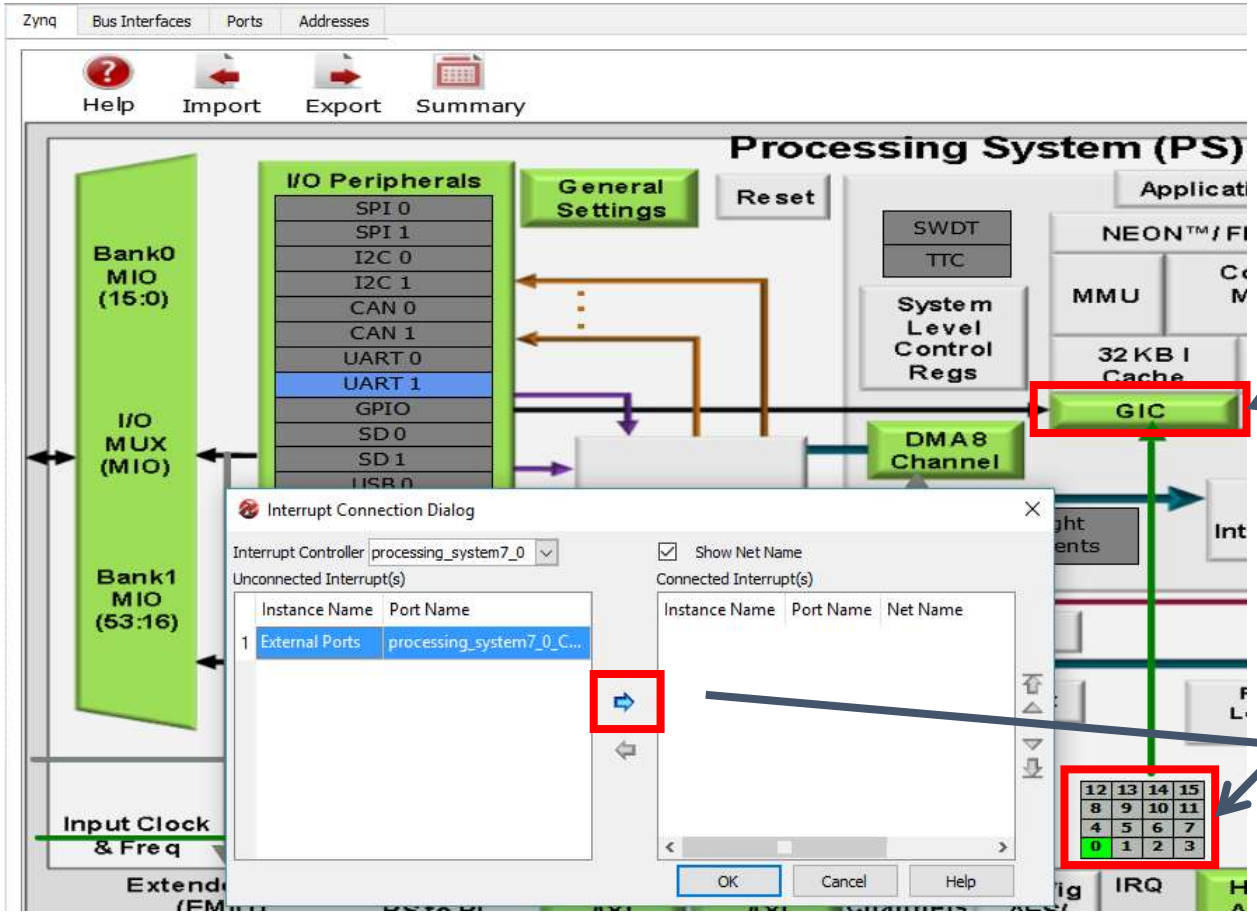




Add a GPIO Peripheral in XPS

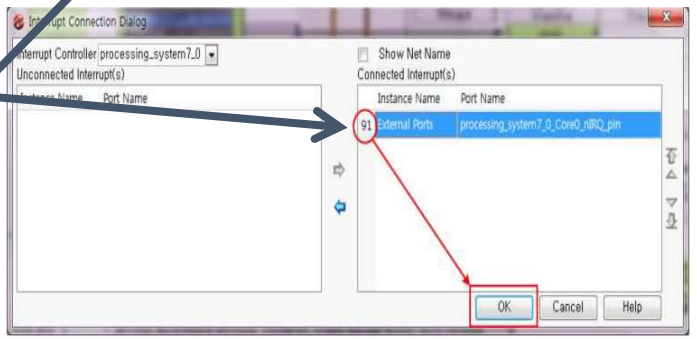


Add a GPIO Peripheral in XPS

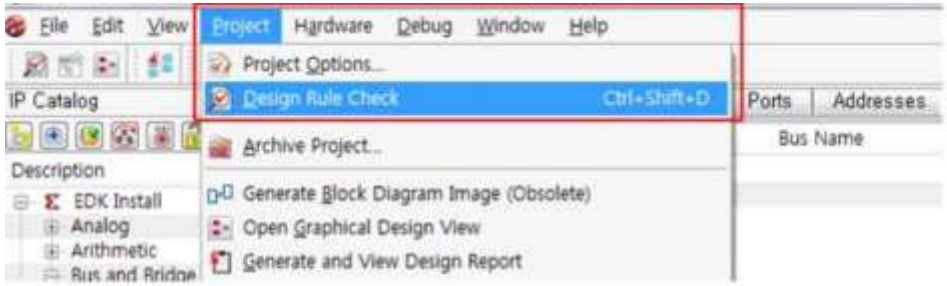


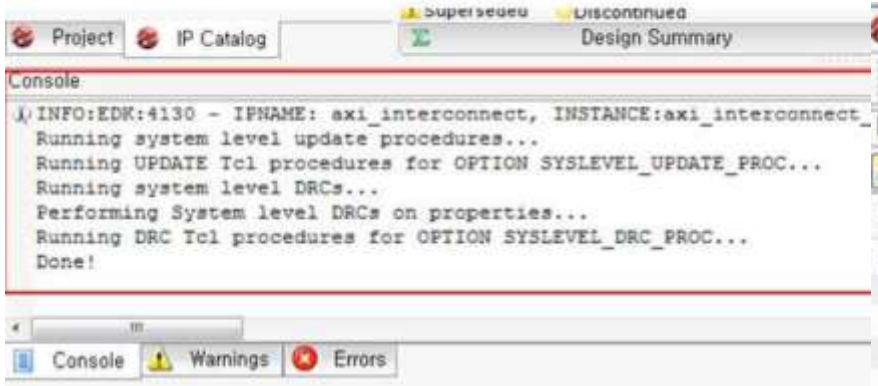
Press

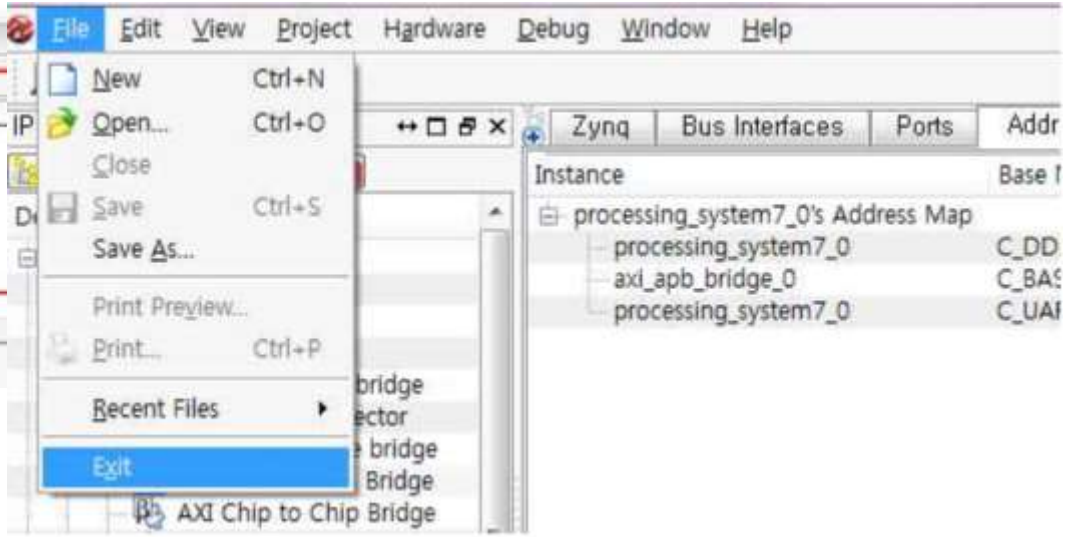
OR



Add a GPIO Peripheral in XPS

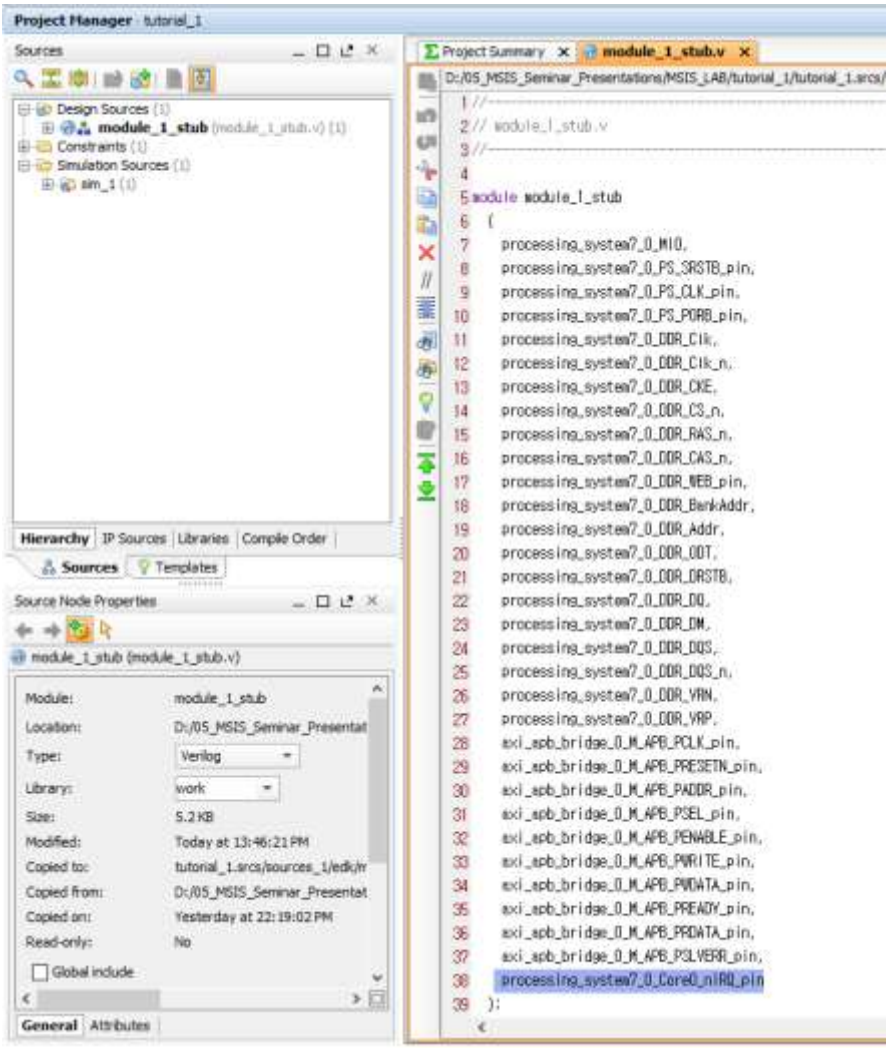






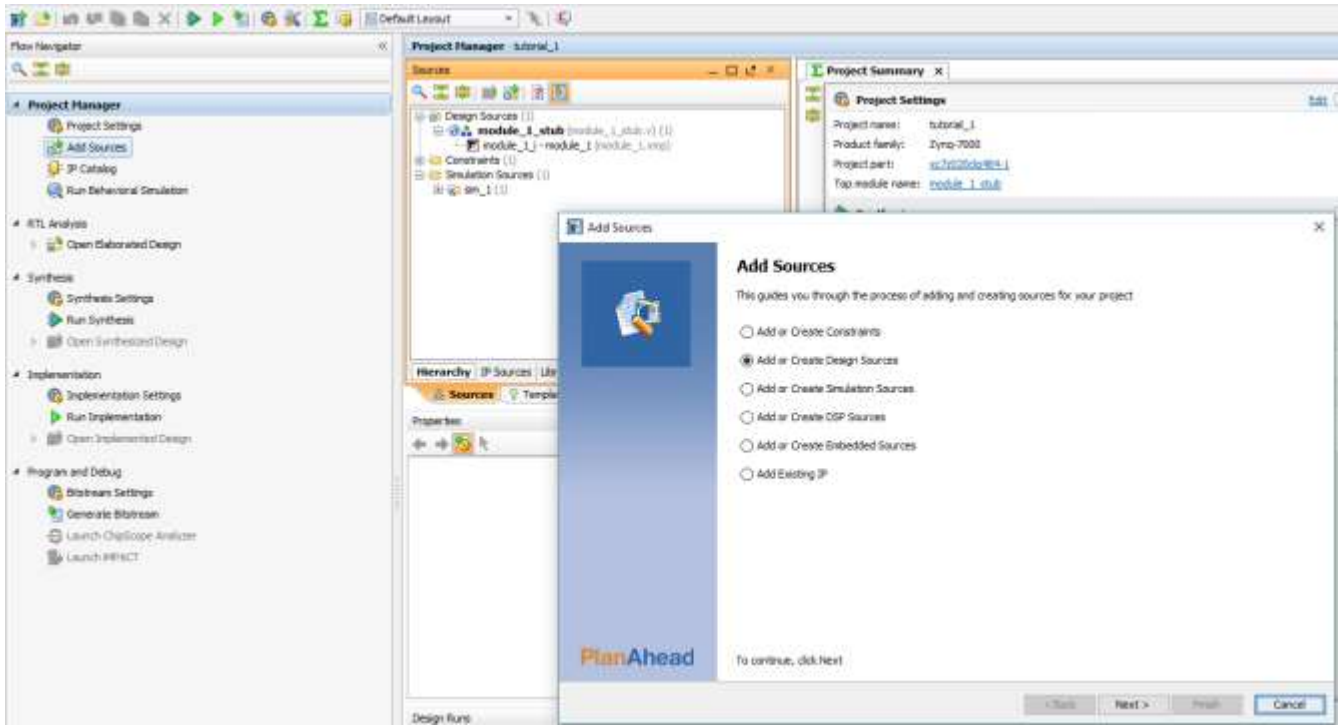
Add a GPIO Peripheral in XPS

- In PlanAhead, double-click **module_1_stub** to open the instantiation HDL file.
- Scroll down to the **module_1** instantiation.
- Right-click on **module_1– module_1 (module_1.xmp)** and select **Create Top HDL**.
- This will overwrite the existing top level HDL wi th an updated HDL module that adds the new external port to the Zynq PS system.
- Double-click **module_1 _stub** to open the new Top-level HDL file.

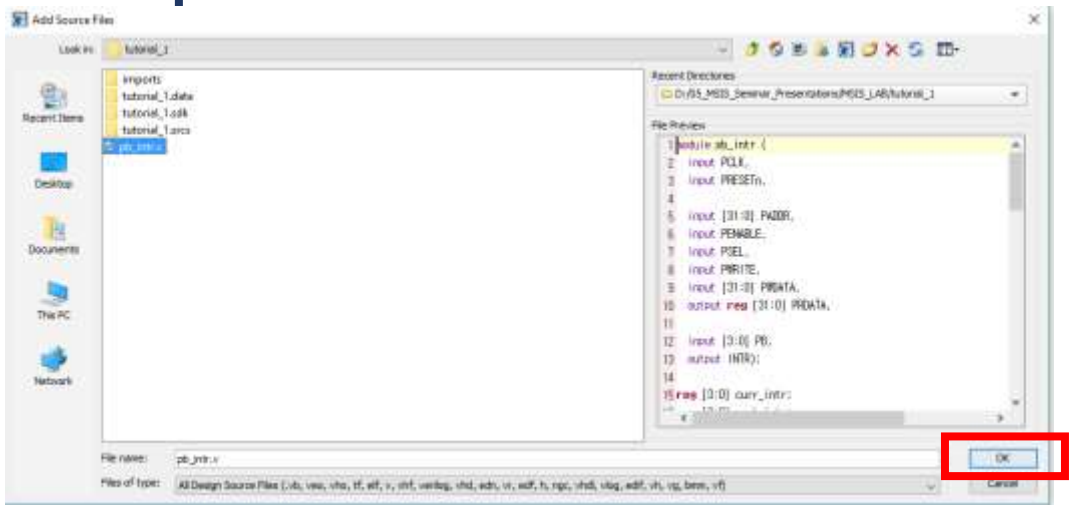


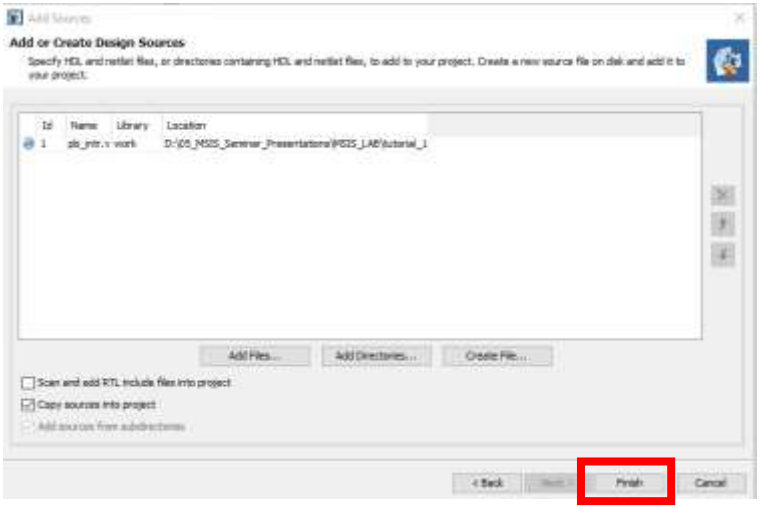
Add a GPIO Peripheral in XPS

- From PlanAhead
  - "Add Sources" → "Create or Add Design Sources"



Add a GPIO Peripheral in XPS



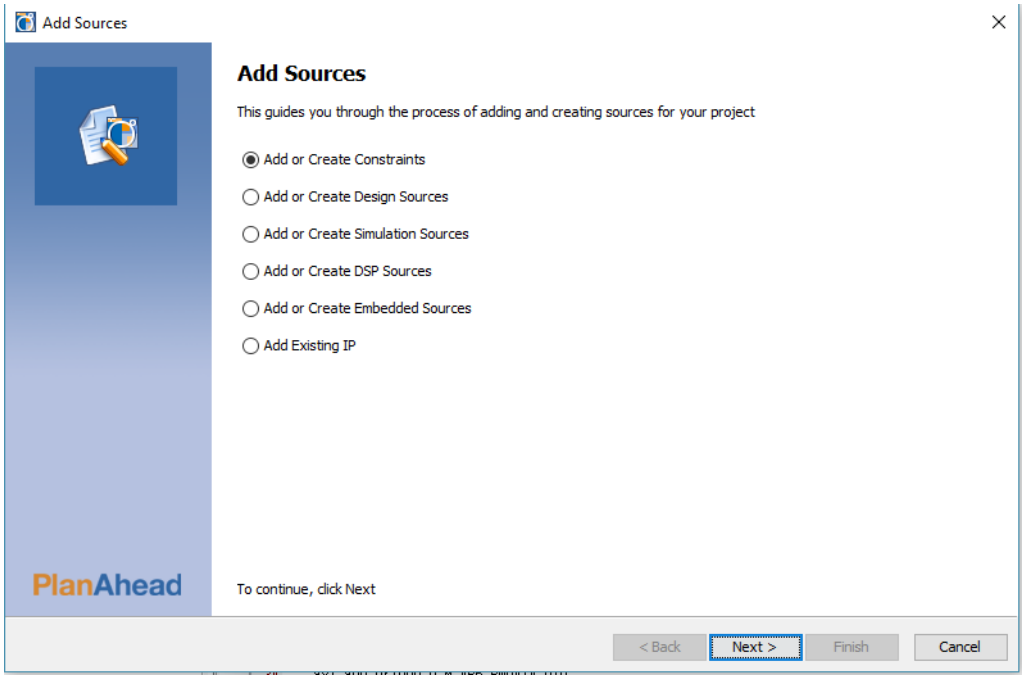


Add a GPIO Peripheral in XPS

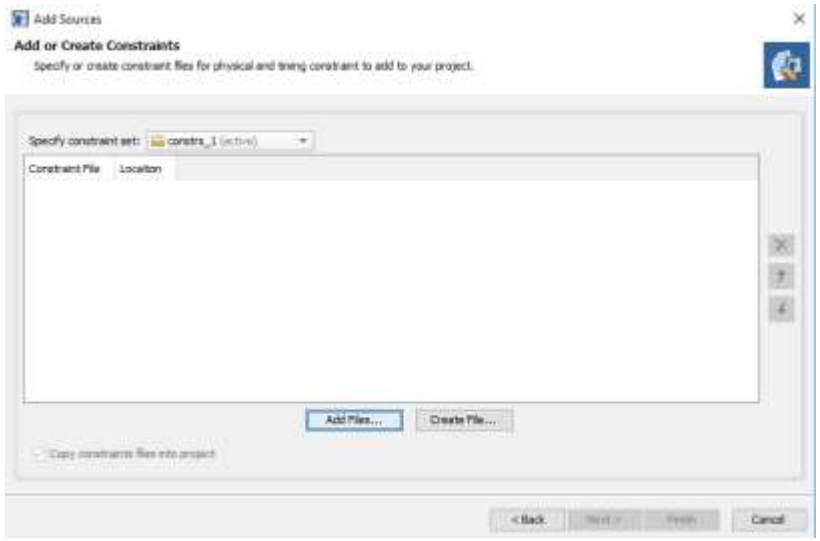
- At this point, we added a new custom PL peripheral and connected to our PS subsystem.
- We need to modify the UCF to connect the new Ports output to external PL pins. Then, the design must be implemented and exported to SDK.

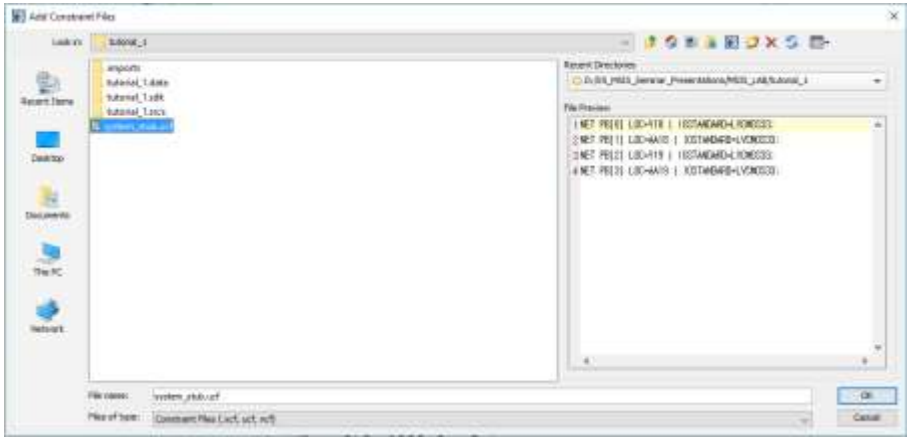
From PlanAhead

"Add Sources" → "Create or Add Constraints"



Add a GPIO Peripheral in XPS





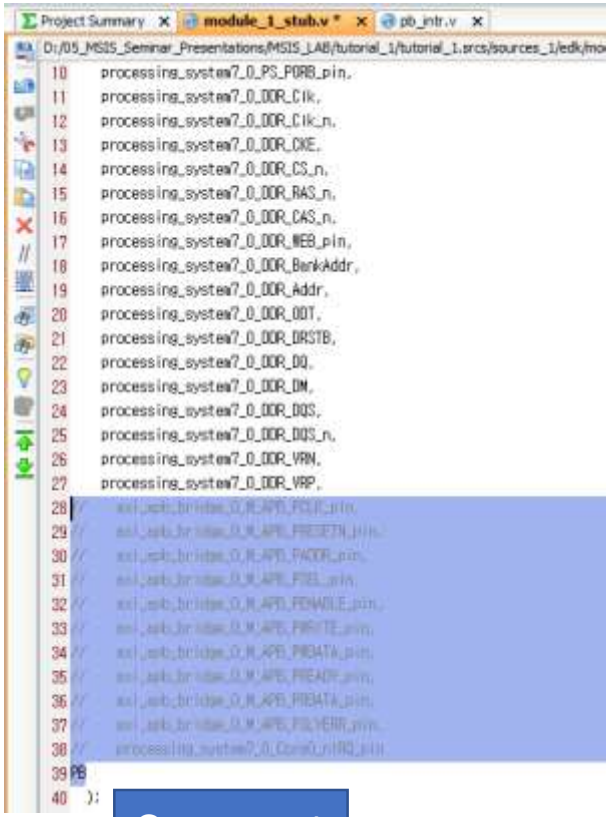
Project Summary | module_1_stub.v | pb_intr.v | system_stub.ucf

D:/05_MSIS_Seminar_Presentations/MSIS_LAB/tutorial_1/tutorial_1.srcs/constrs_1/imports/tutoria

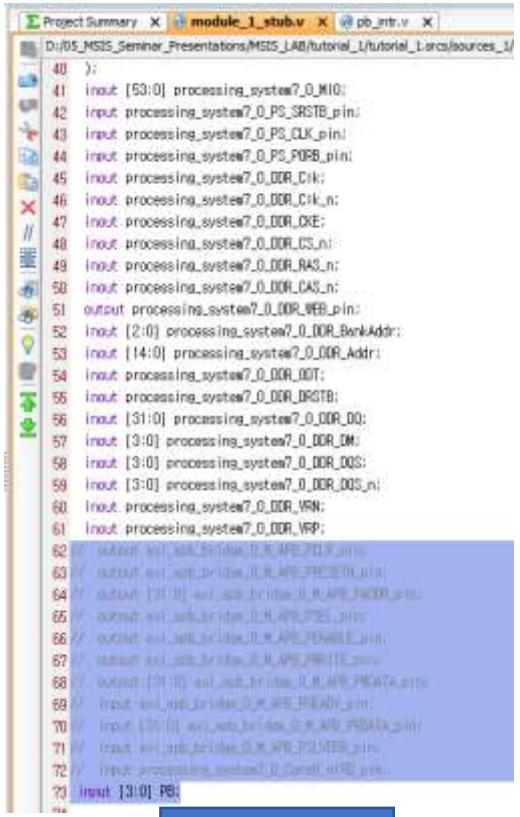
```
1 NET PB[0] LOC=Y18 | IOSTANDARD=LVCMOS33;
2 NET PB[1] LOC=AA18 | IOSTANDARD=LVCMOS33;
3 NET PB[2] LOC=Y19 | IOSTANDARD=LVCMOS33;
4 NET PB[3] LOC=AA19 | IOSTANDARD=LVCMOS33;
```

Add a GPIO Peripheral in XPS

- Modify the code as follow



Comment



Comment

```
75 wire axi_apb_bridge_0_M_APB_PCLK_pin;
76 wire axi_apb_bridge_0_M_APB_PRESETN_pin;
77 wire [31:0] axi_apb_bridge_0_M_APB_PADDR_pin;
78 wire axi_apb_bridge_0_M_APB_PSEL_pin;
79 wire axi_apb_bridge_0_M_APB_PENABLE_pin;
80 wire axi_apb_bridge_0_M_APB_PWRITE_pin;
81 wire [31:0] axi_apb_bridge_0_M_APB_PWDATA_pin;
82 wire axi_apb_bridge_0_M_APB_PREADY_pin;
83 wire [31:0] axi_apb_bridge_0_M_APB_PRDATA_pin;
84 wire axi_apb_bridge_0_M_APB_PSLVERR_pin;
85 wire processing_system7_0_Core0_nIRQ_pin;
```

Add this

Add a GPIO Peripheral in XPS

- Instantiate Your PL

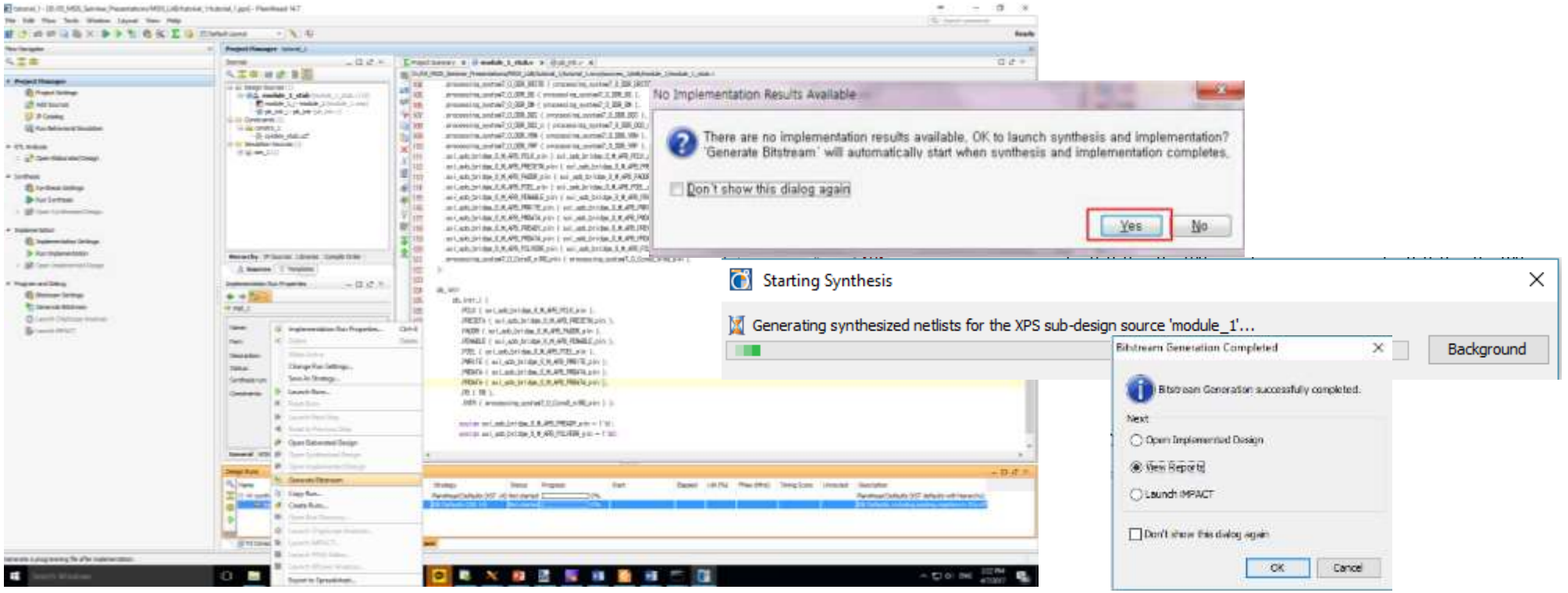
```
124   pb_intr
125         pb_intr_i (
126           .PCLK ( axi_apb_bridge_0_M_APB_PCLK_pin ),
127           .PRESETn ( axi_apb_bridge_0_M_APB_PRESETN_pin ),
128           .PADDR ( axi_apb_bridge_0_M_APB_PADDR_pin ),
129           .PENABLE ( axi_apb_bridge_0_M_APB_PENABLE_pin ),
130           .PSEL ( axi_apb_bridge_0_M_APB_PSEL_pin ),
131           .PWRITE ( axi_apb_bridge_0_M_APB_PWRITE_pin ),
132           .PWDATA ( axi_apb_bridge_0_M_APB_PWDATA_pin ),
133           .PRDATA ( axi_apb_bridge_0_M_APB_PRDATA_pin ),
134           .PB ( PB ),
135           .INTR ( processing_system7_0_Core0_nIRQ_pin ) );
136
137           assign axi_apb_bridge_0_M_APB_PREADY_pin = 1'b1;
138           assign axi_apb_bridge_0_M_APB_PSLVERR_pin = 1'b0;
```

```
1  module pb_intr (
2    input PCLK,
3    input PRESETn,
4
5    input [31:0] PADDR,
6    input PENABLE,
7    input PSEL,
8    input PWRITE,
9    input [31:0] PWDATA,
10   output reg [31:0] PRDATA,
11
12   input [3:0] PB,
13   output INTR);
```

The APB interface of the pb_intr does not have the Ready port and the Error port, but the APB interface of the system module have th ese ports.
System module receives Ready signal and Error signal, Set Ready si gnal is High ('1') and Set Error signal to Low ('0').

Add a GPIO Peripheral in XPS

- Under Flow Navigator, click **Generate Bitstream**.
  - Click **Yes** to run synthesis and implementation.
    - This will take a few minutes to complete. Ignore and close any warnings.



Add a GPIO Peripheral in XPS

- Export the design to SDK. **Click File → Export → Export Hardware for SDK…**
- Select the checkbox for **Launch SDK**, so all 3 checkboxes should be checked. Click **OK**.

- You will do same steps as previous.

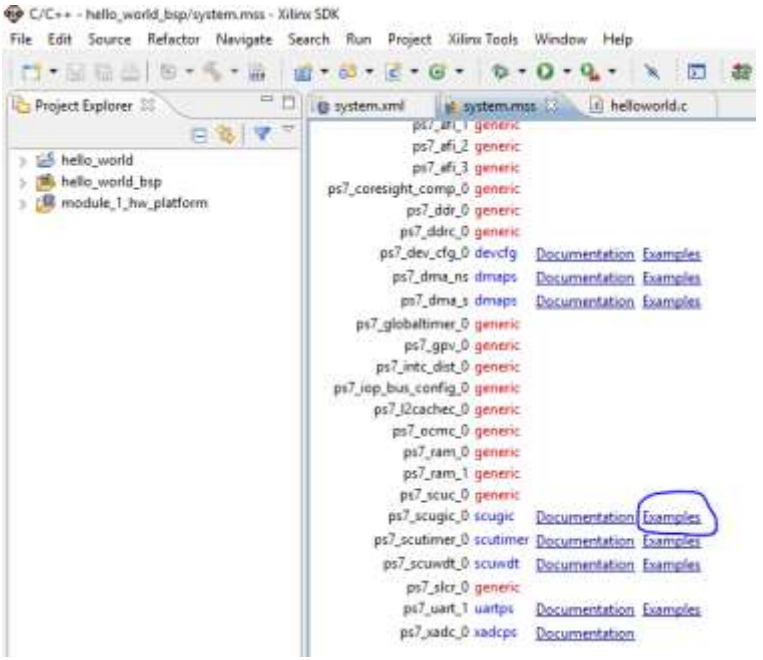
LAB 4: WRITING SOFTWARE FOR A CUSTOM PL PERIPHERAL

Objectives

- This tutorial demonstrates how to write software for the new custom PL pe ripheral. Using SDK this lab will show how to:
  - Create a new software project
  - Access addresses and parameters defined in the Xparameters.h file
  - Write a simple C application
  - Download and test the software application in hardware
  - Use SDK debugger to step through our software application

Create a Software Project

- We will use SDK to create a new software proje ct.
- In SDK, **File → New → Xilinx Application Projec t**
- The standalone_bsp_0 definition is defined in **s ystem.mss**, which is already open.
  - Click on the **system.mss** tab in the primary window.
  - This information page about the BSP provides a number of useful links to Documentation and Examples. Under **Peripheral Drivers**, you will see the G PIO peripheral, **ps7_scugic_0**. Next to tha t is a hyperlink for examples. Click **Exa mples**.



Create a Software Project

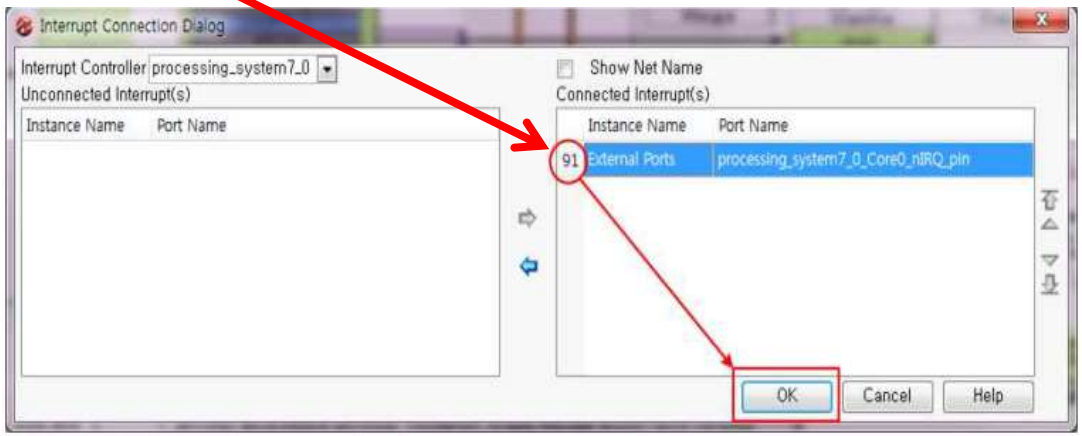
Open the file to view it.
Select xscugic_example.c and Copy its contents.

From Xilinx SDK → Project Explorer → Opent helloworld.c
Paste xscugic_example.c contents

Create a Software Project

- The next thing of interest is the define statement for **DEVICE_ID**

```
]/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID
//#define INTC_DEVICE_INT_ID    0x0E
#define INTC_DEVICE_INT_ID  91
```

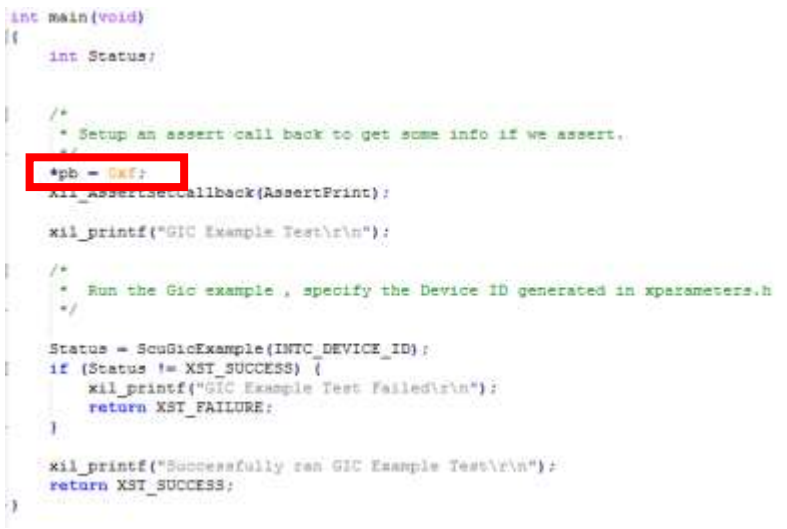


Create a Software Project

- To access the Interrupt Generator IP, declare the address as a pointer.
- When you add an Interrupt Generator IP, XPS sets the address of the Interrupt Gen erator IP to 0x72800000
- Because you've assigned it, you declared the address 0x72800000 in your C code.

```
/************************** Variable Definitions *****************************/

XScuGic InterruptController;         /* Instance of the Interrupt Controller */
static XScuGic_Config *GicConfig;    /* The configuration parameters of the
                                        controller */

volatile unsigned int *pb = 0x72800000;
```

In the main function, write 0xf to Inter rupt Generator IP to initialize Interrupt Generator IP.



Create a Software Project

- Because the actual interrupt signal goes into the system module, the ScuGi cExample function, is commented.

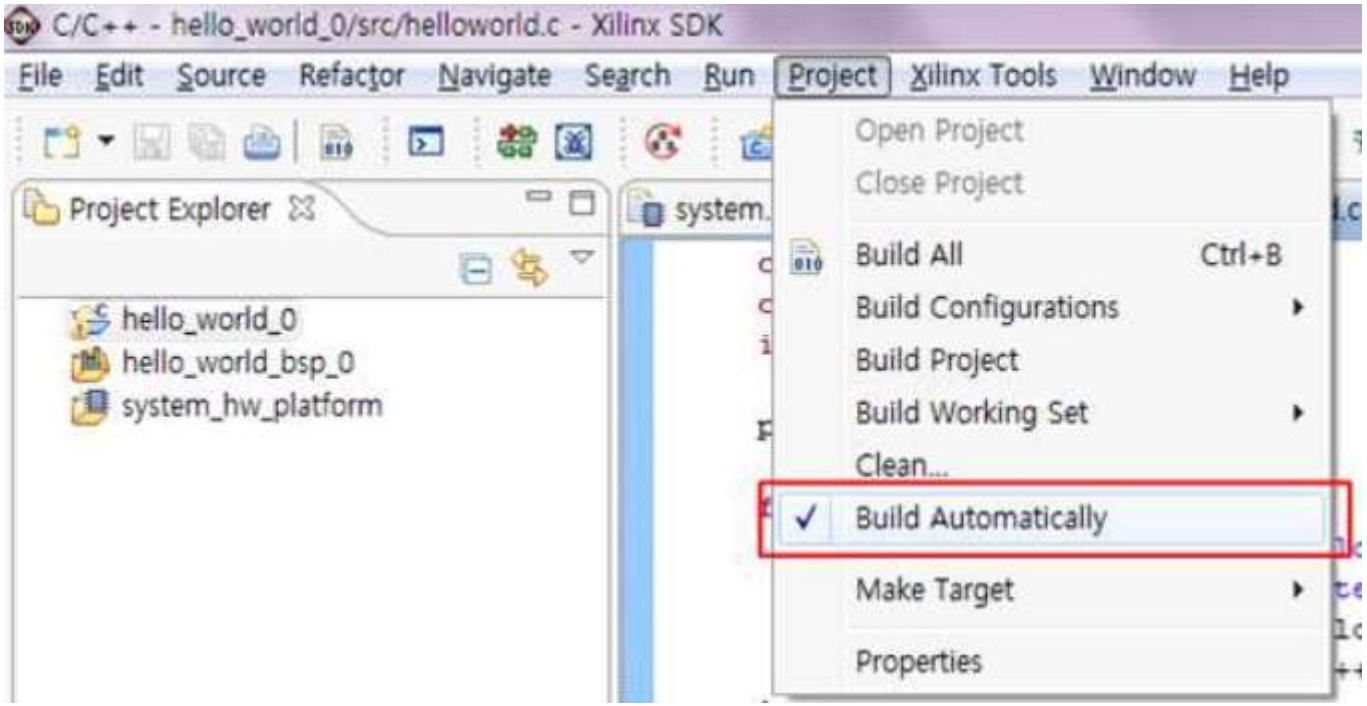
```
    /*
     *  Simulate the Interrupt
     */
//  Status = XScuGic_SoftwareIntr(&InterruptController,
//                   INTC_DEVICE_INT_ID,
//                   XSCUGIC_SPI_CPU0_MASK);
//  if (Status != XST_SUCCESS) {
//      return XST_FAILURE;
//  }
```

Create a Software Project

- Add code to the interrupt handler function (DeviceDriverHandler).
- When an interrupt occurs, the processor executes the interrupt handler function.
- The first part of the code to be added identifies which Push Button was pressed as a Register of IP.
- The second is to modify the Register so that IP no longer causes interrupts.
- Third, UART outputs the Push Button that generated the interrupt.

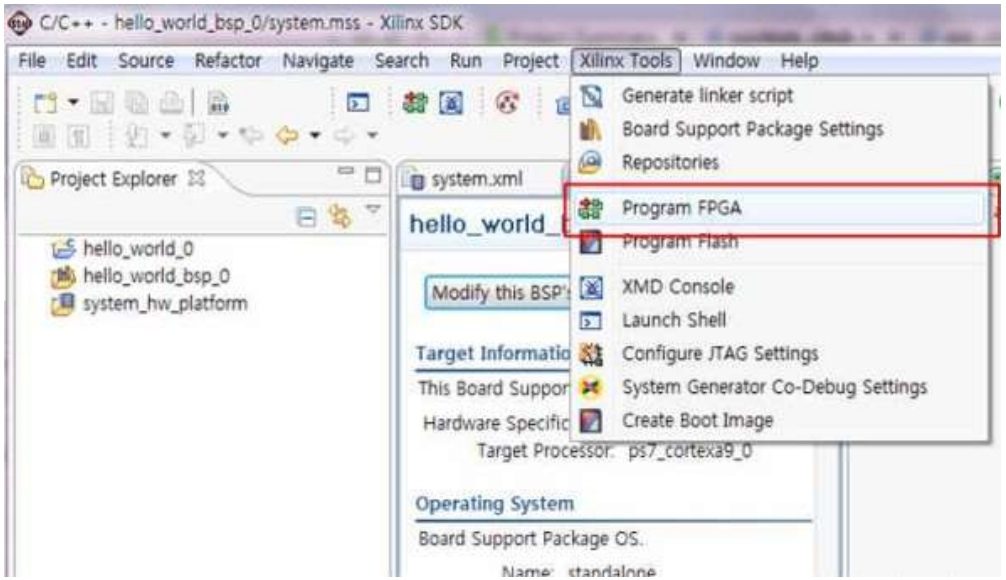
```
void DeviceDriverHandler(void *CallbackRef)
{
    /*
     * Indicate the interrupt has been processed using a shared variable
     */
    if ((*pb & 1) == 0) {
        *pb = *pb + 1;
        xil_printf("S1 Switch is pushed\r\n");
    }
    else if ((*pb & 2) == 0) {
        *pb = *pb + 2;
        xil_printf("S2 Switch is pushed\r\n");
    }
    else if ((*pb & 4) == 0) {
        *pb = *pb + 4;
        xil_printf("S3 Switch is pushed\r\n");
    }
    else if ((*pb & 8) == 0) {
        *pb = *pb + 8;
        xil_printf("S4 Switch is pushed\r\n");
    }

    InterruptProcessed = TRUE;
    xil_printf("xx");
}
```

Create a Software Project



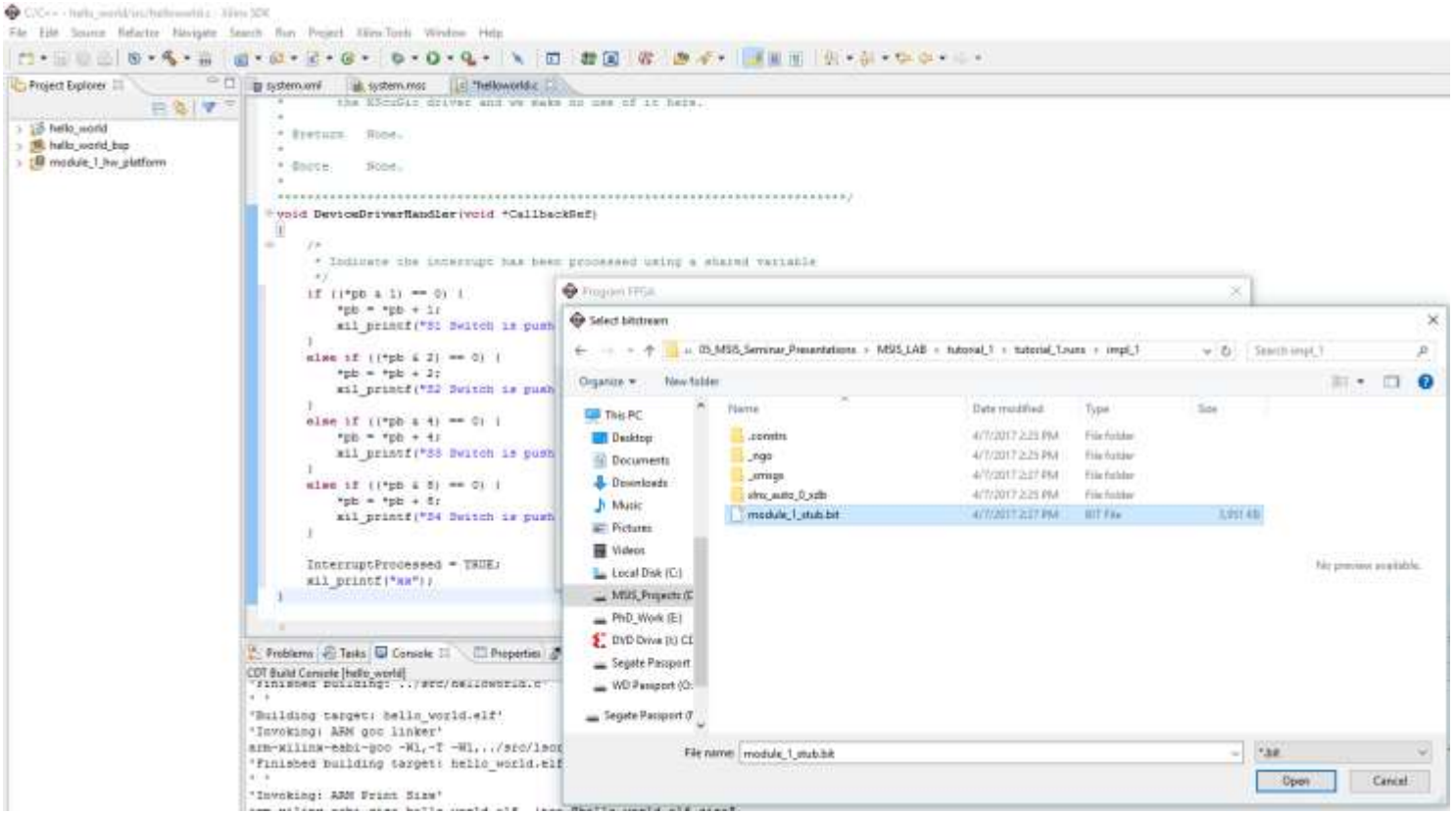
LAB 5: PROGRAM THE FPGA

Program the FPGA

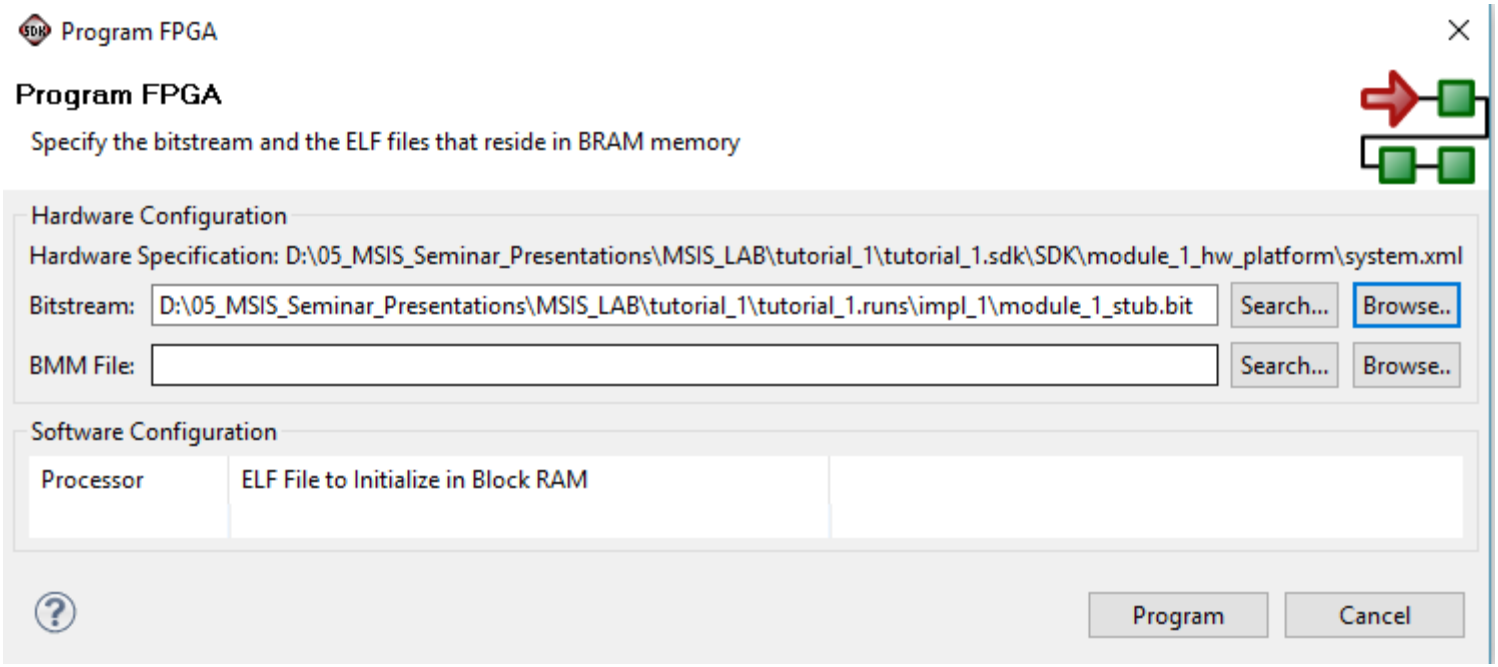


Program the FPGA

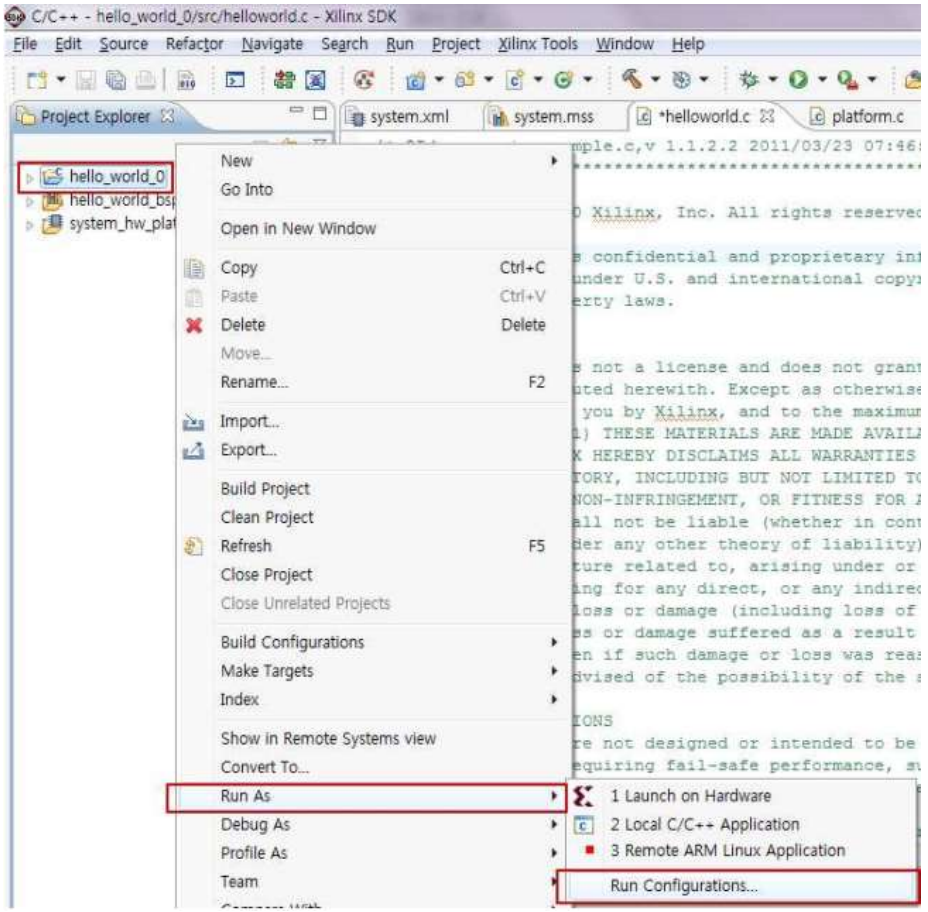
- Verify the correct bitstream is selected and click **Program**



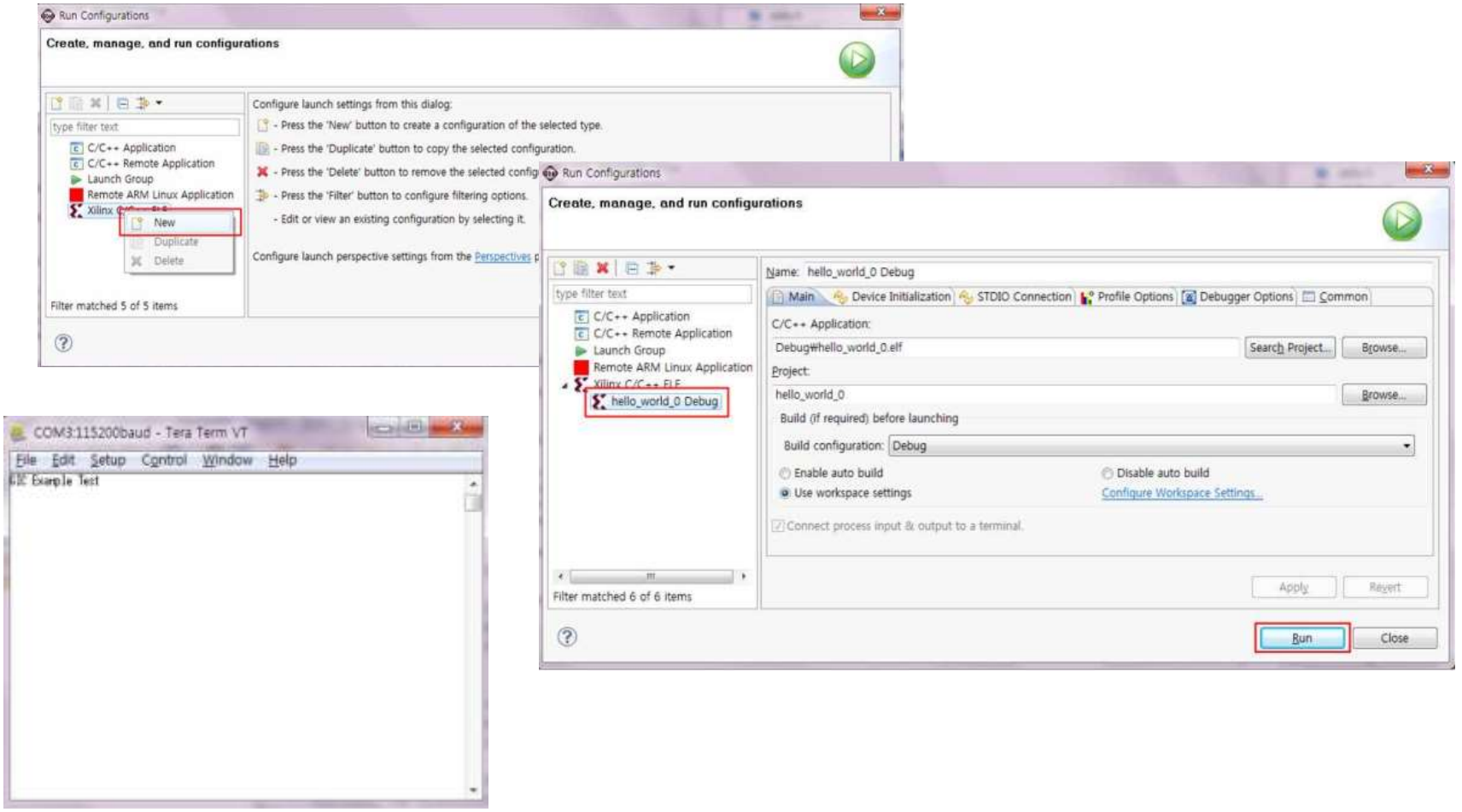
Program the FPGA



Run your Software

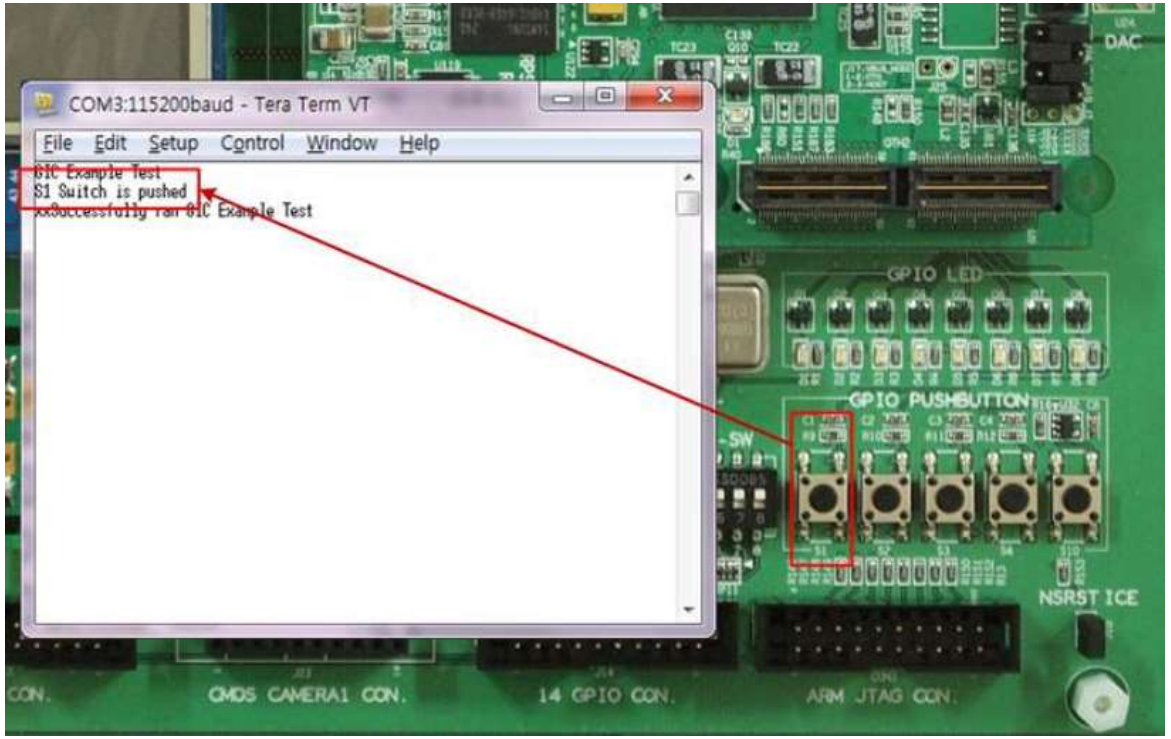


Run your Software



Hardware/Software Co-Design

- When you press the S1 Push Button, "S1 Switch is pushed" is di splayed in the Terminal program,
- Then processor operation is terminated.



Resources

- www.Xilinx.com
- The ZYNQ Book
- Introduction to Zynq-7000™ All Programmable SoC
- Huins RPS_Z7020_Education
- https://www.beyond-circuits.com/wordpress/tutorial/
- http://sunsided.github.io/zybo-tutorial/