



Creating Processor System

Vivado HLS 2013.3 Version

Objectives

➤ After completing this module, you will be able to:

- Describe embedded system development flow in Zynq
- List the steps involved in creating adapter
- State how adapter created in Vivado HLS is used in Vivado Design Suite

Outline

- *Embedded System Design in Zynq using IP Integrator*
- **Creating IP-XACT Adapters**
- **Integrating the IP-XACT Adapter in AXI System**
- **Summary**

Embedded Design Architecture in Zynq

➤ Embedded design in Zynq is based on:

- Processor and peripherals
 - Dual ARM® Cortex™ -A9 processors of Zynq-7000 AP SoC
 - AXI interconnect
 - AXI component peripherals
 - Reset, clocking, debug ports
- Software platform for processing system
 - Standalone OS
 - C language support
 - Processor services
 - C drivers for hardware
- User application
 - Interrupt service routines (optional)

The PS and the PL

➤ The Zynq-7000 AP SoC architecture consists of two major sections

- PS: Processing system
 - Dual ARM Cortex-A9 processor based
 - Multiple peripherals
 - Hard silicon core
- PL: Programmable logic
 - Uses the same 7 series programmable logic
 - Artix™-based devices: Z-7010, Z-7015, and Z-7020 (high-range I/O banks only)
 - Kintex™-based devices: Z-7030, Z-7045, and Z-7100 (mix of high-range and high-performance I/O banks)

➤ What are Vivado, IP Integrator and SDK?

- Vivado is the tool suite for Xilinx FPGA design and includes capability for embedded system design
 - IP Integrator, is part of Vivado and allows block level design of the hardware part of an Embedded system
 - Integrated into Vivado
 - Vivado includes all the tools, IP, and documentation that are required for designing systems with the Zynq-7000 AP SoC hard core and/or Xilinx MicroBlaze soft core processor
 - Vivado + IPI replaces ISE/EDK
- SDK is an Eclipse-based software design environment
 - Enables the integration of hardware and software components
 - Links from Vivado

➤ Vivado is the overall project manager and is used for developing non-embedded hardware and instantiating embedded systems

- Vivado/IP Integrator flow is recommended for developing Zynq embedded systems using 2013.2 and later

Embedded System Tools: Hardware

➤ Hardware and software development tools

- IP Integrator
- IP Packager
- Hardware netlist generation
- Simulation model generation
- Xilinx Microprocessor Debugger (XMD)
- Hardware debugging using Vivado analyzer

Embedded System Tools: Software

➤ Eclipse IDE-based Software Development Kit (SDK)

- Board support package creation
- GNU software development tools
- C/C++ compiler for the MicroBlaze and ARM Cortex-A9 processors (gcc)
- Debugger for the MicroBlaze and ARM Cortex-A9 processors (gdb)
- TCF framework – multicore debug

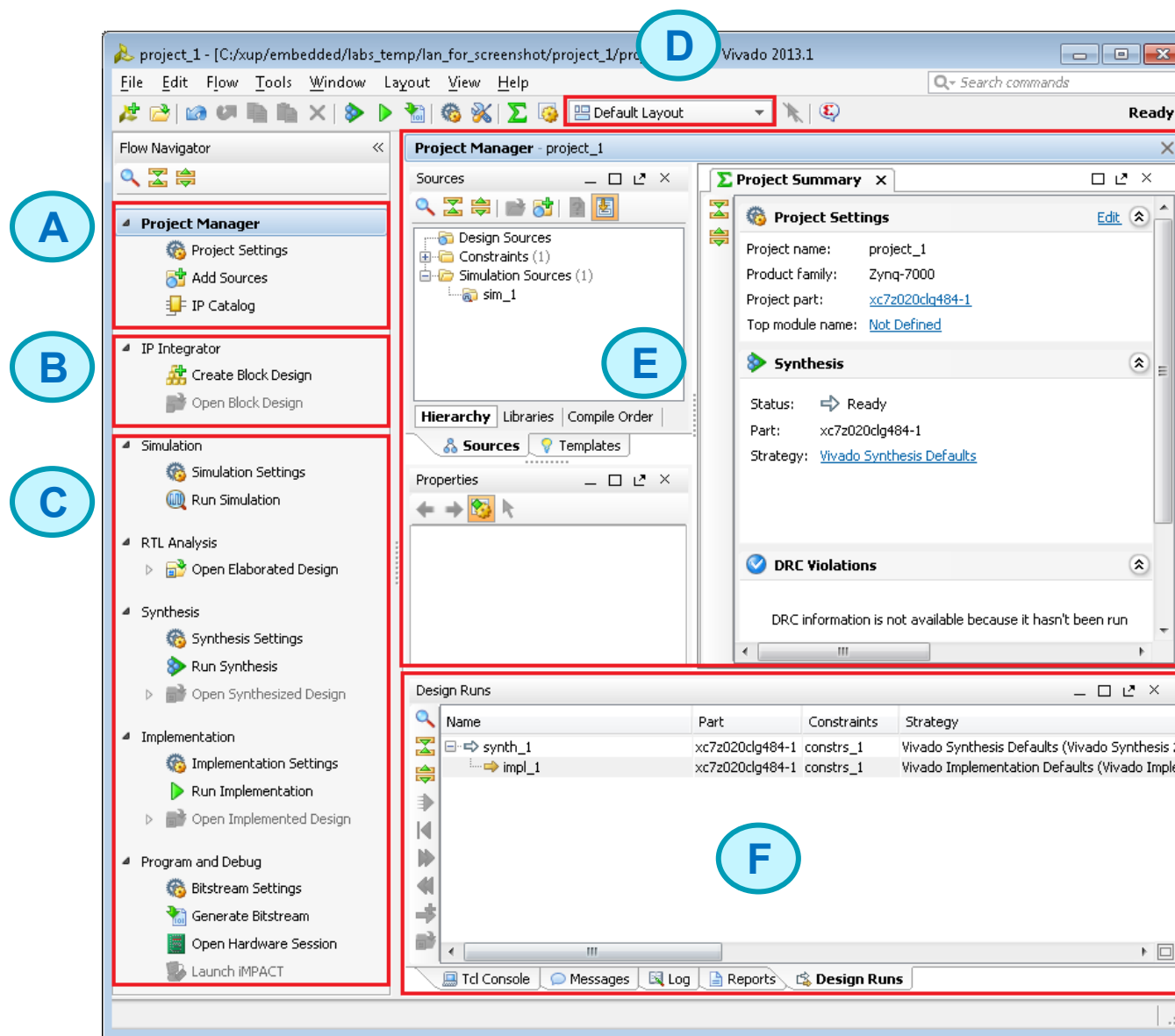
➤ Board support packages (BSPs)

- Stand-alone BSP
 - Free basic device drivers and utilities from Xilinx
 - NOT an RTOS

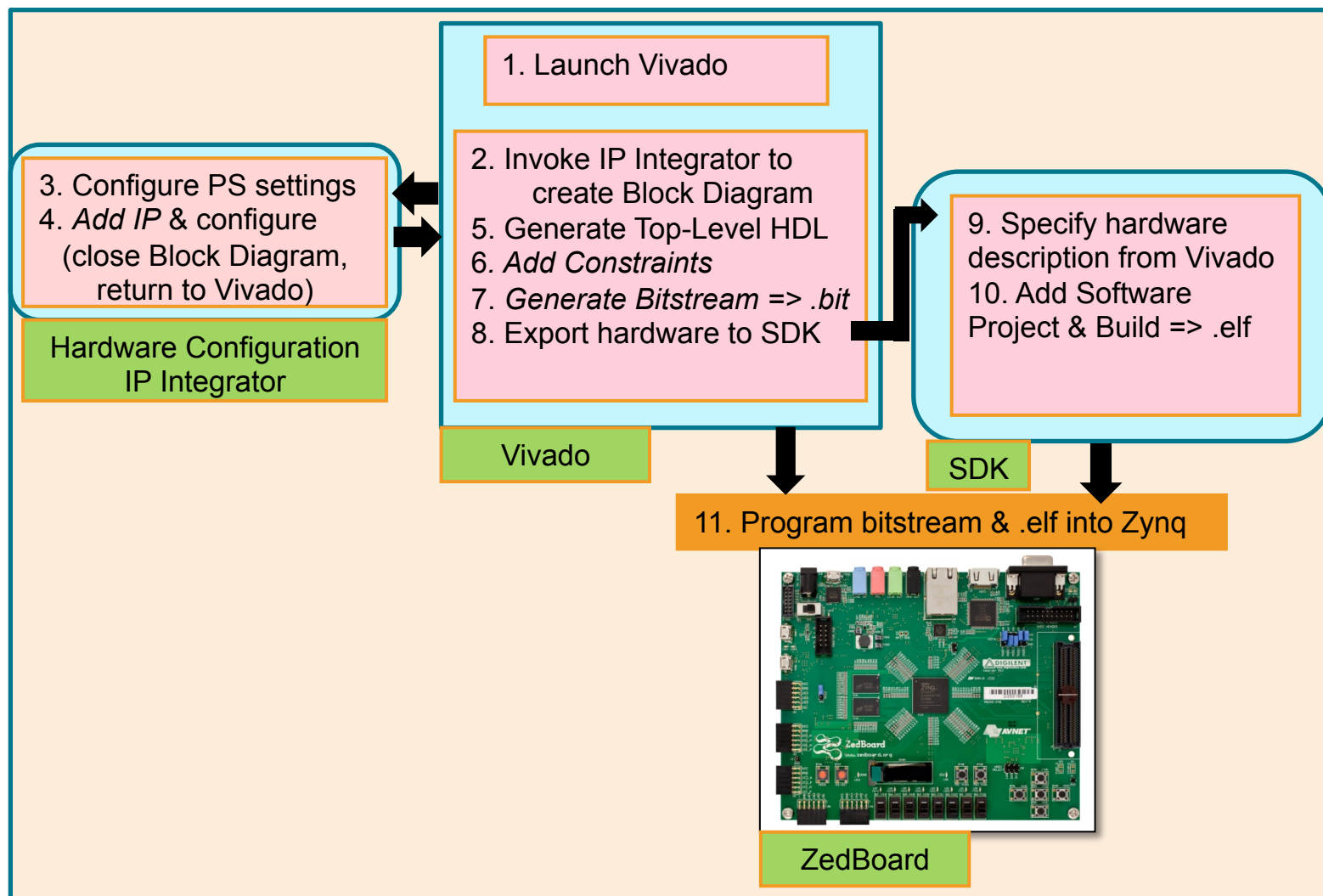
Vivado View

➤ Customizable panels

- A: Project Management
- B: IP Integrator
- C: FPGA Flow
- D: Layout Selection
- E: Project view/Preview Panel
- F: Console, Messages, Logs

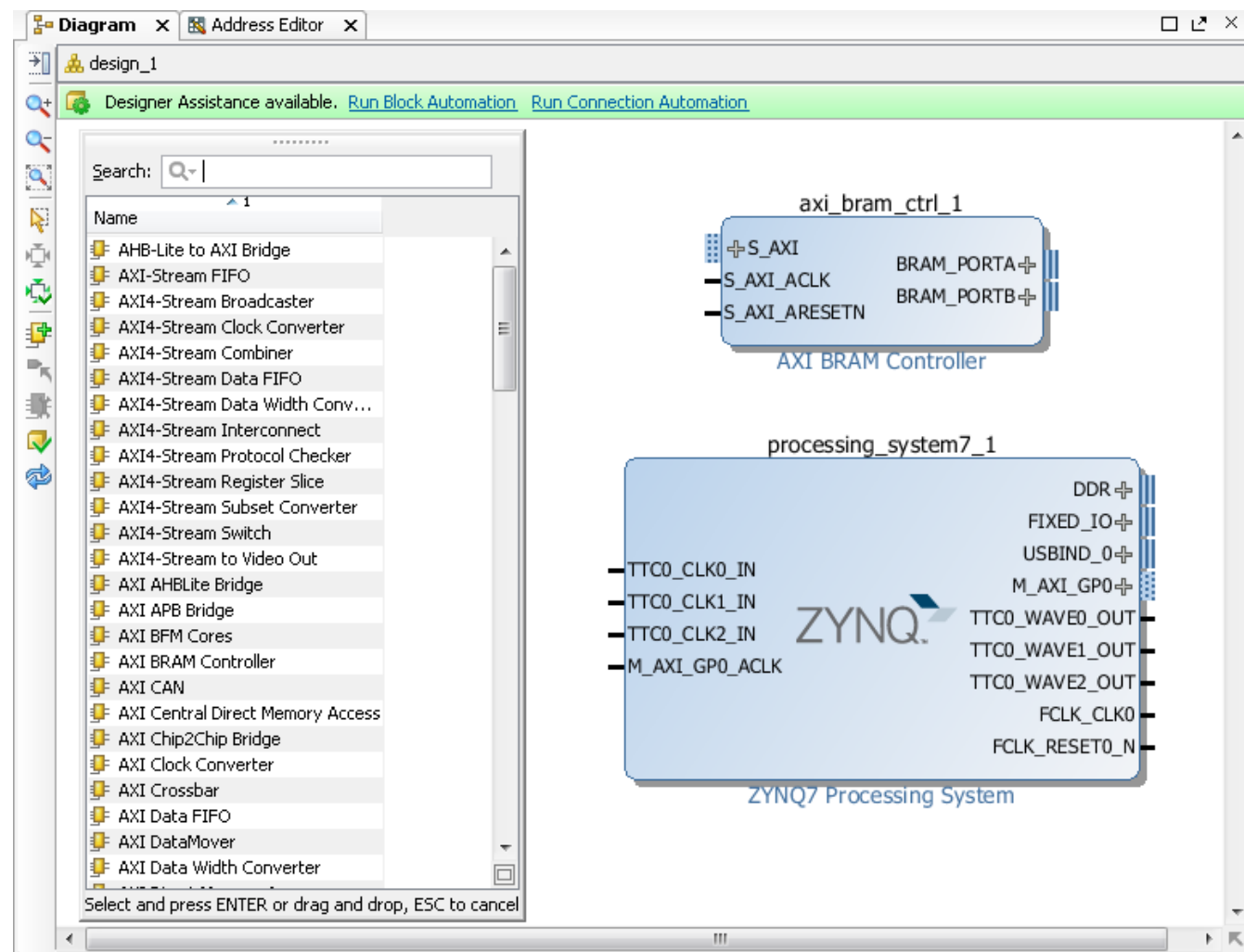


Embedded System Design using Vivado



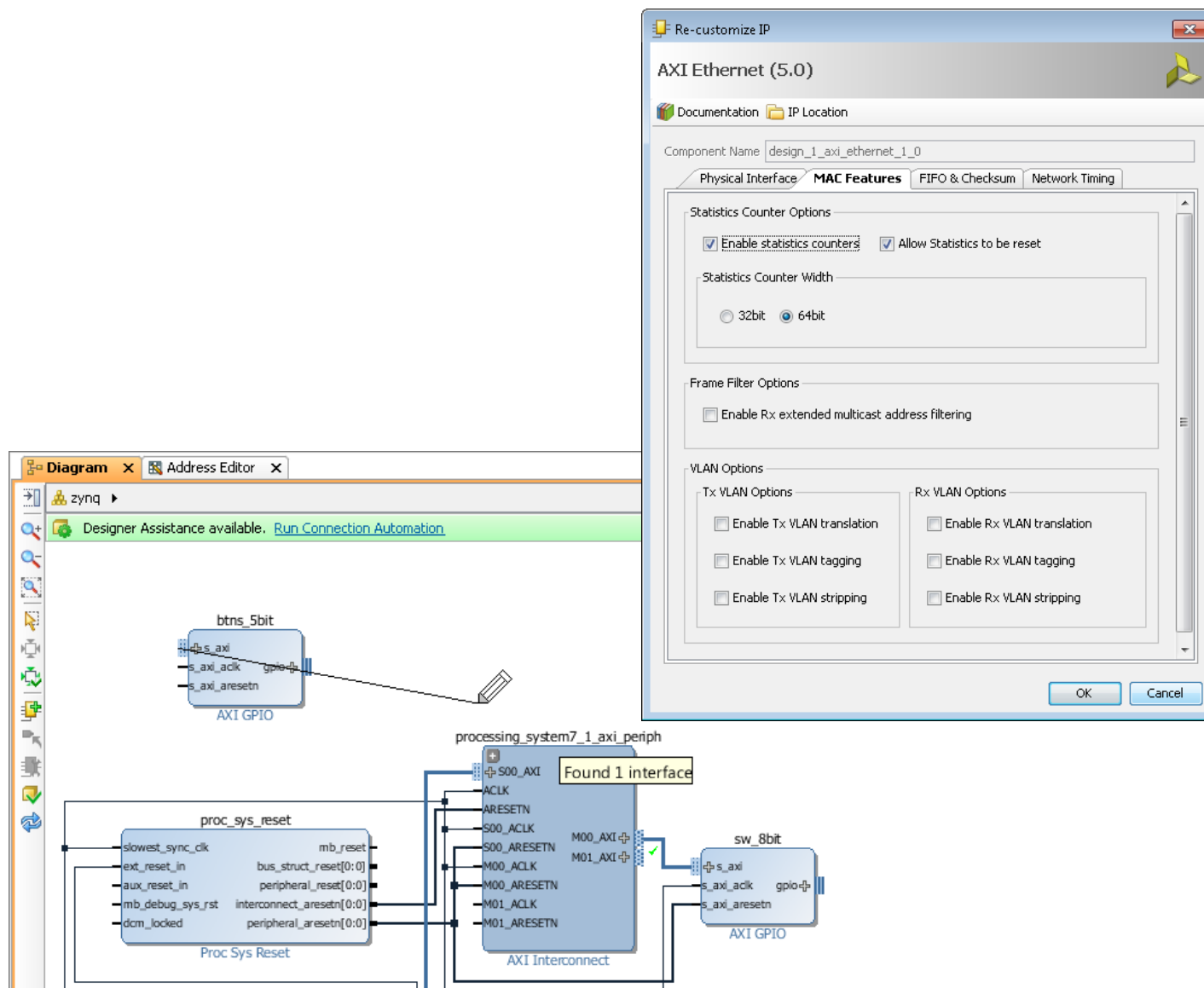
Add IP Integrator Block Diagram

- IP Integrator Block Diagram opens a blank canvas
- IP can be added from the IP catalog
- Drag and drop interface
- Intelligent Design environment
 - Design Assistance
 - Connection automation
 - Highlights valid connections
 - Group, create hierarchal blocks
- Can import custom IP using IP Packager



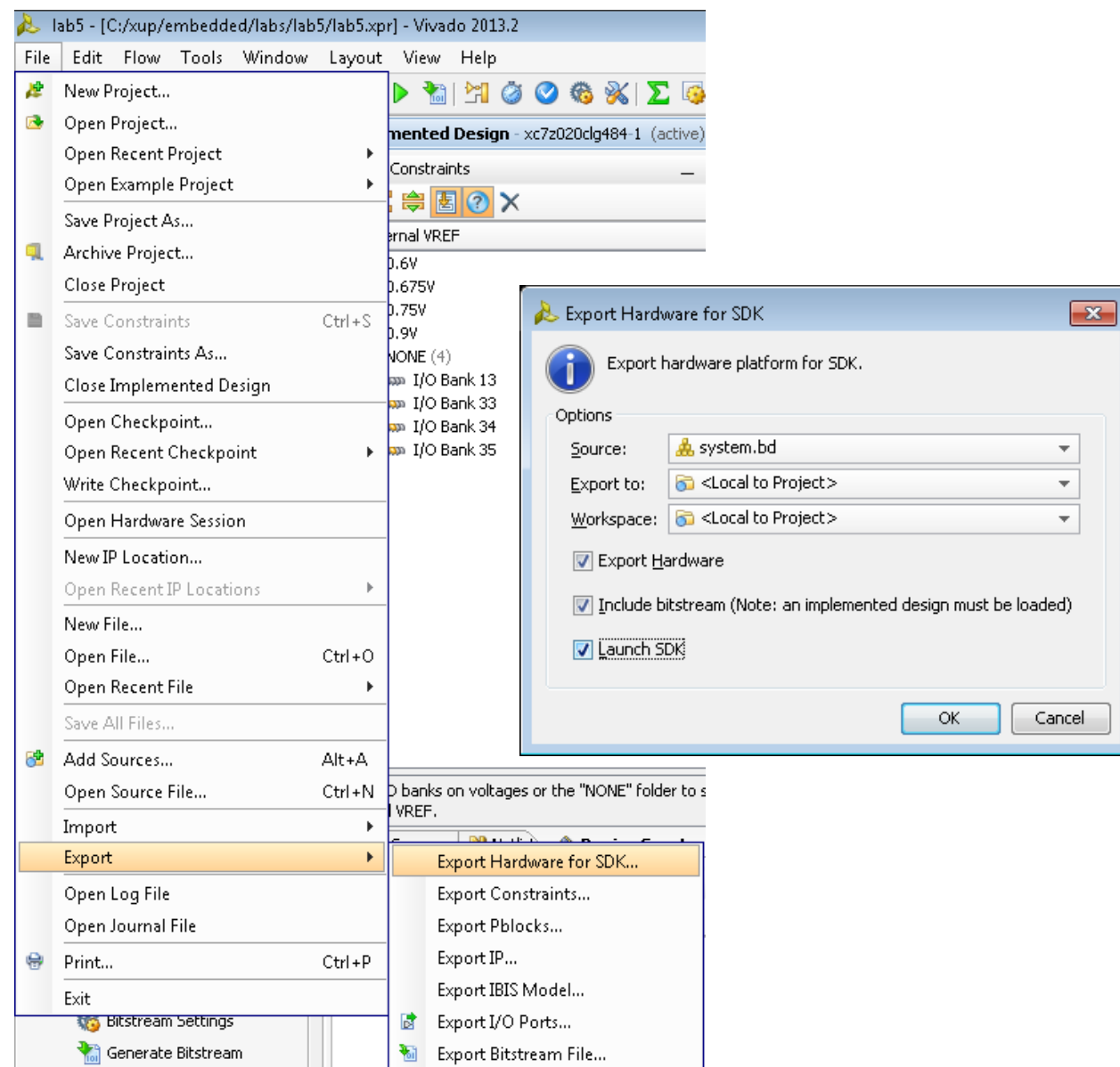
Configuring and Connecting Hardware in IP Integrator

- Double click blocks to access configuration options
- Drag pointer to make connections
 - Highlights valid connections
- Connection Automation
 - Automatically connect recognised interfaces
- Automatically Redraw system



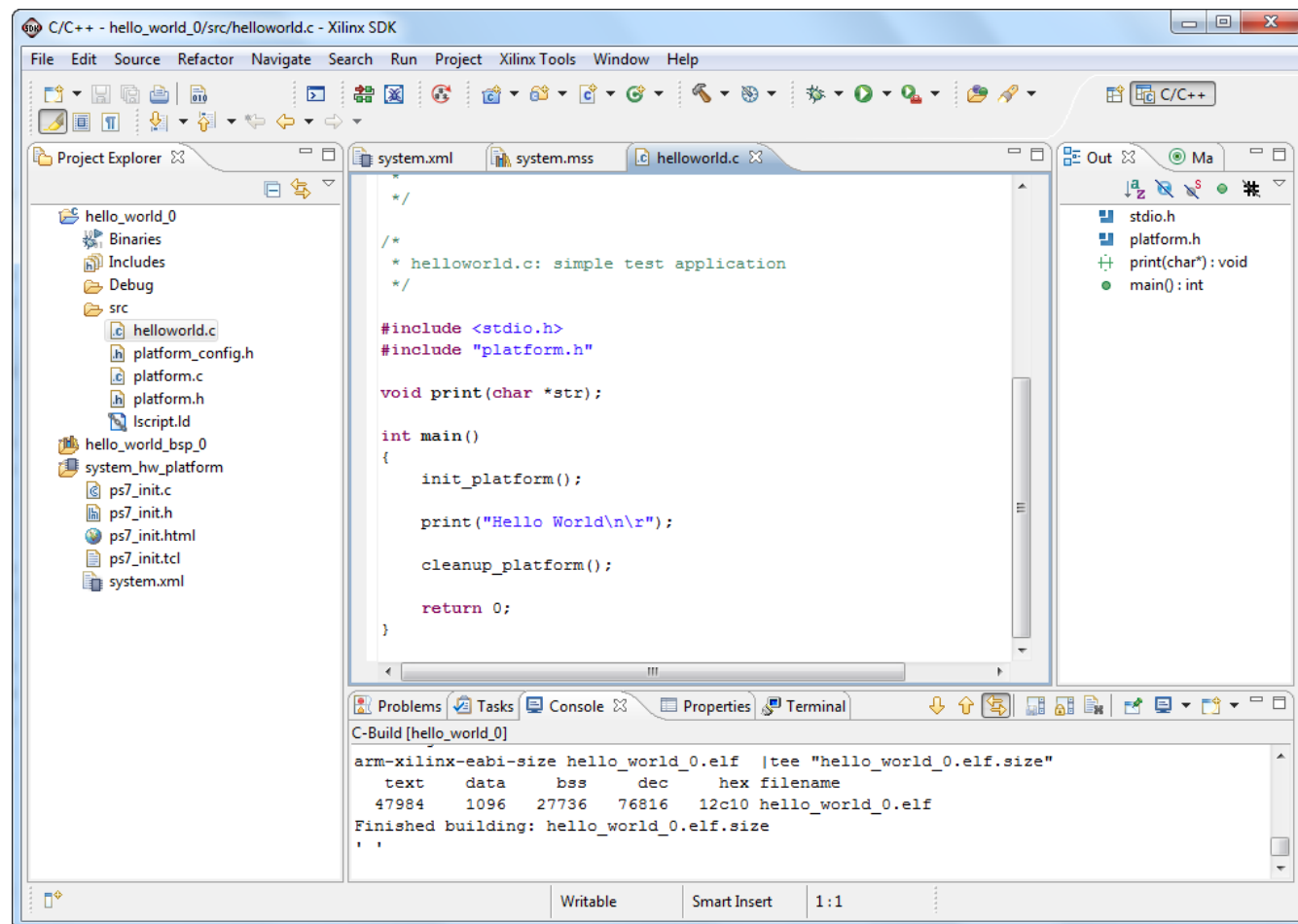
Exporting to SDK

- **Software development is performed with the Xilinx Software Development Kit tool (SDK)**
 - Hardware configuration in Vivado
- **An XML description of the hardware is exported to SDK**
 - The hardware platform is built on this description
 - Only one hardware platform for each SDK project
 - (Optionally export bitstream)
- **The SDK tool will then associate user software projects to hardware**



Software Development Flow

- **Create hardware platform project**
 - Automatically performed when SDK tool is launched from Vivado project
- **Create BSP**
 - System software, board support package
- **Create software application**
- **Create linker script**
- **Build project**
 - compile, assemble, link output file `<app_project>.elf`



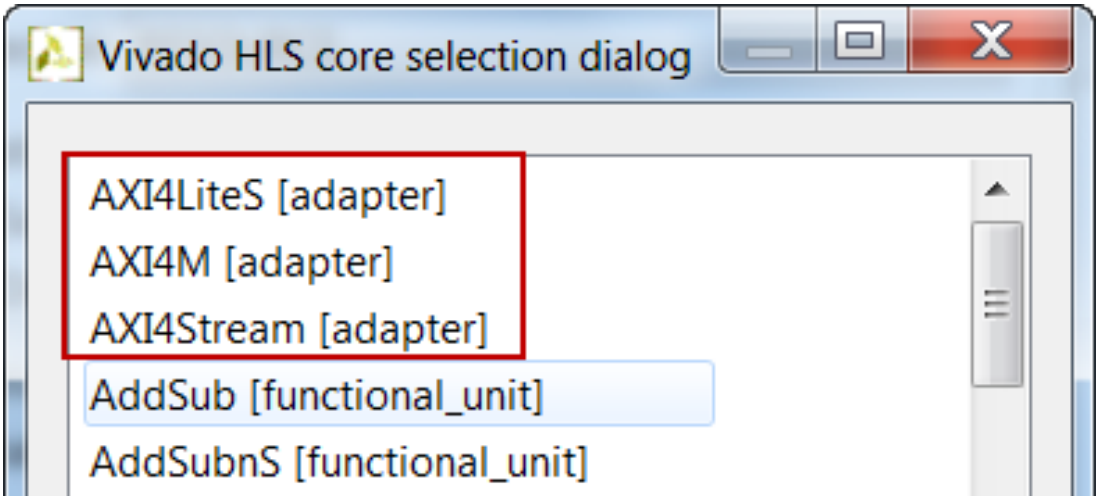
Outline

- Embedded System Design in Zynq using IP Integrator
- ***Creating IP-XACT Adapters***
- Integrating the IP-XACT Adapter in AXI System
- Summary

Adapter Cores

➤ **List of Adapter Cores for reference**

Type	Core Name	Description
AXI4	AXI4M	AXI4 Master interface
	AXI4Stream	AXI4 stream interface
	AXI4LiteS	AXI4Lite slave interface



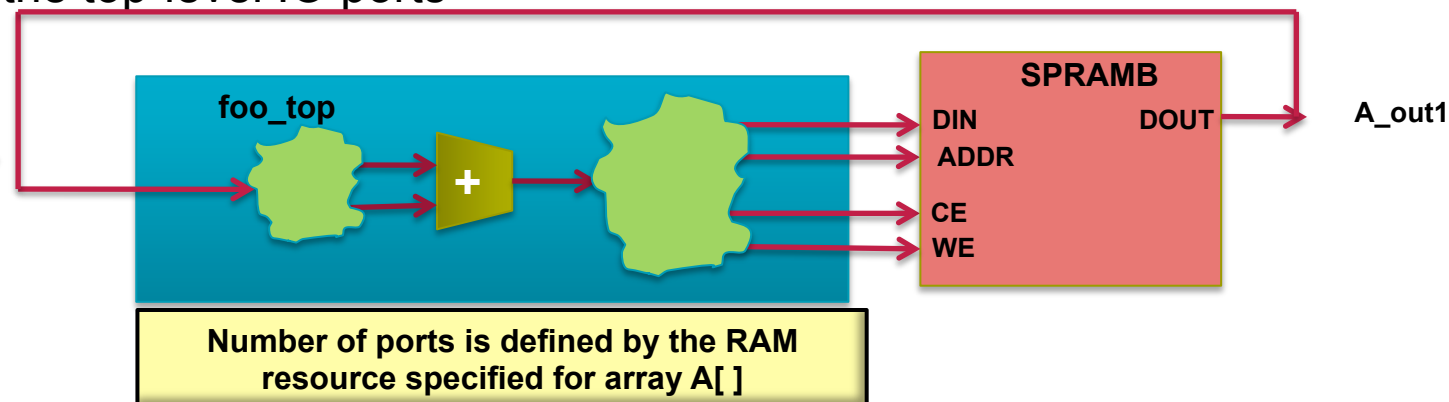
Using RESOURCE for external connections

➤ When the array is an argument of the top-level function

- The array/RAM is “off-chip”
- The type of RESOURCE determines the top-level IO ports

```
void foo_top( int A[3*N] , int x)
{
  L1: for (i = 0; i < N; i++)
    A[i+x] = A[i] + i;
}
```

Synthesis



➤ Add adapters to the RTL and create a bus interface

- The principle is the same
- We define a resource for the port
 - Plus there are a few other options we'll cover

Adding Adapters 1) Check the RTL & C IO

➤ The first step in adding an adapter

- Make sure you have a suitable port on the RTL
- Example,
 - If you want an AXI Stream interface → must have an ap_fifo on the RTL
 - If an ap_fifo on the RTL is needed → pass pointer variable, array, or reference variable by reference (in, out, but not in/out)

Bus Interfaces														
AXI4														
Stream	Lite	Master												
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔											
			↔			</								

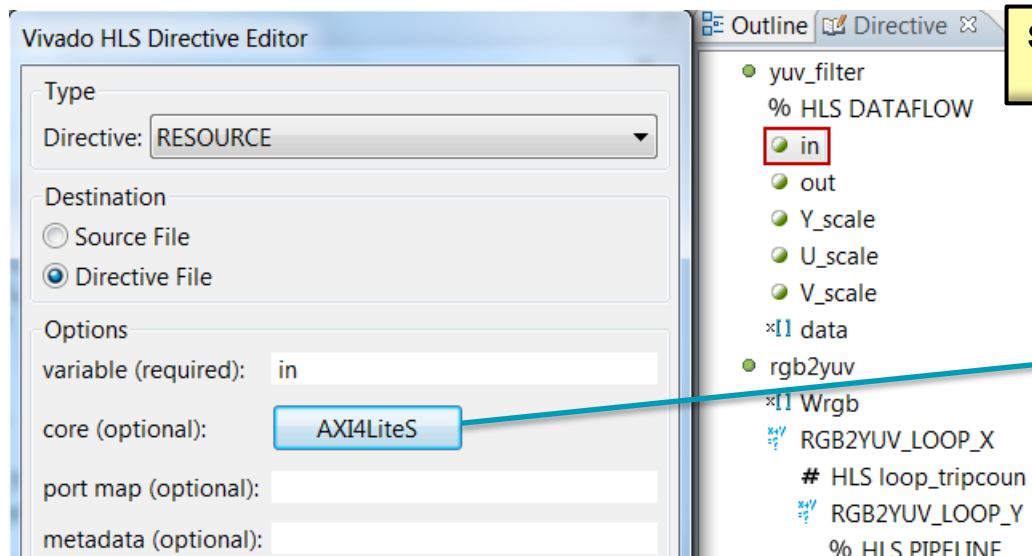
2) Assign an adapter Core to each Port

➤ Once the I/O adapter is known

- Add the core as a resource on the RTL port

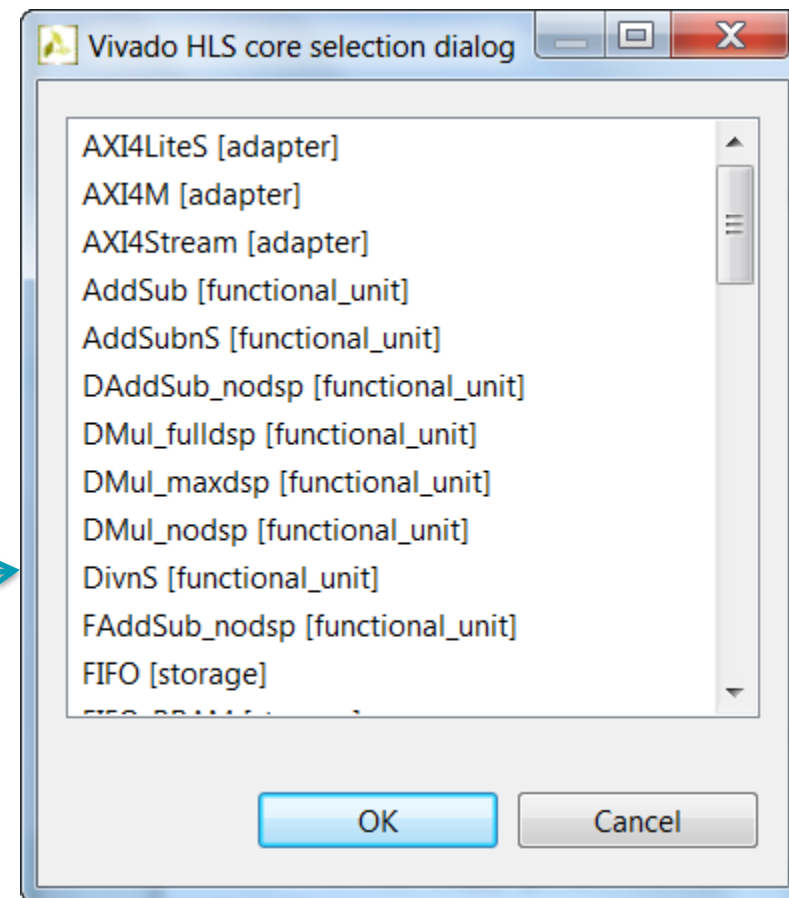
➤ The cores are listed in the Vivado HLS Library Guide

- And available from the menu in the GUI



Select the IO port & RESOURCE

Adapter cores are noted → (adapter)



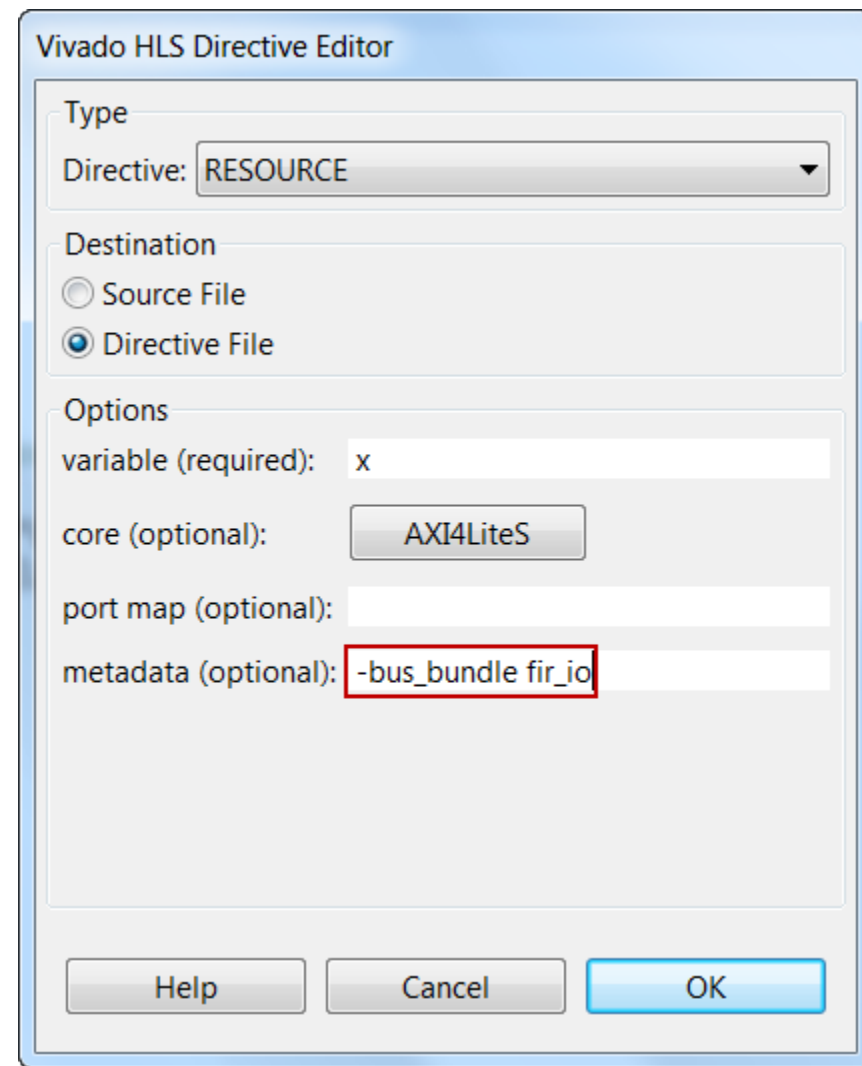
3) Optionally group and rename buses

➤ Metadata bundle buses

- Allow buses to be grouped with new names
 - Will be grouped into the same bus
- In this example, port “x” will be
 - AXI4 Slave interface
 - And renamed fir_io
 - Any other adapter named fir_io will be grouped with this port

➤ Port Name Mappings

- Optionally, port may be name mapped in the port map field
 - {{output} {output_hs_vld, output_ap_ack}}
- The RTL port names are renamed on the adapter interface
 - In the above example, it will be renamed as output



The image shows the Vivado HLS Directive Editor dialog box. It has a title bar 'Vivado HLS Directive Editor'. Inside, there are three main sections: 'Type', 'Destination', and 'Options'. The 'Type' section has a dropdown menu with 'RESOURCE' selected. The 'Destination' section has two radio buttons: 'Source File' and 'Directive File', with 'Directive File' selected. The 'Options' section has four fields: 'variable (required):' with the value 'x', 'core (optional):' with a button labeled 'AXI4LiteS', 'port map (optional):' which is empty, and 'metadata (optional):' with the value '-bus_bundle fir_io' which is highlighted with a red rectangle. At the bottom, there are three buttons: 'Help', 'Cancel', and 'OK'.

Vivado HLS Directive Editor

Type
Directive: RESOURCE

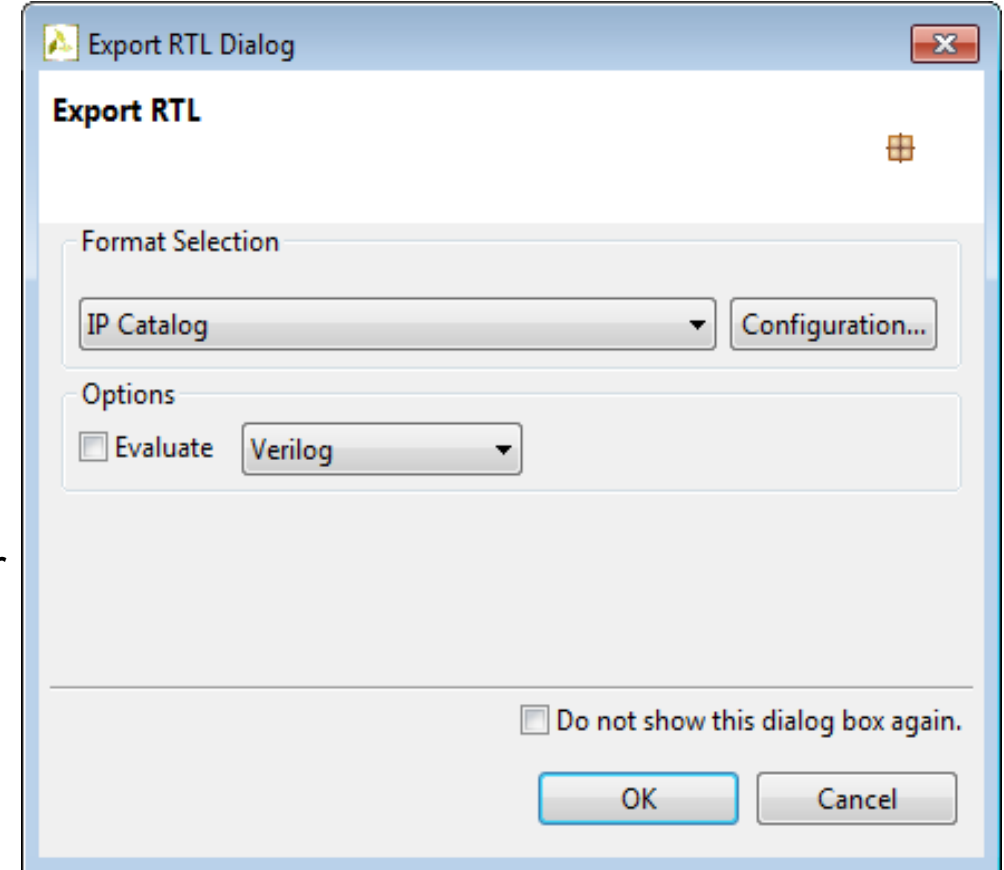
Destination
☐ Source File
☒ Directive File

Options
variable (required): x
core (optional): AXI4LiteS
port map (optional):
metadata (optional): -bus_bundle fir_io

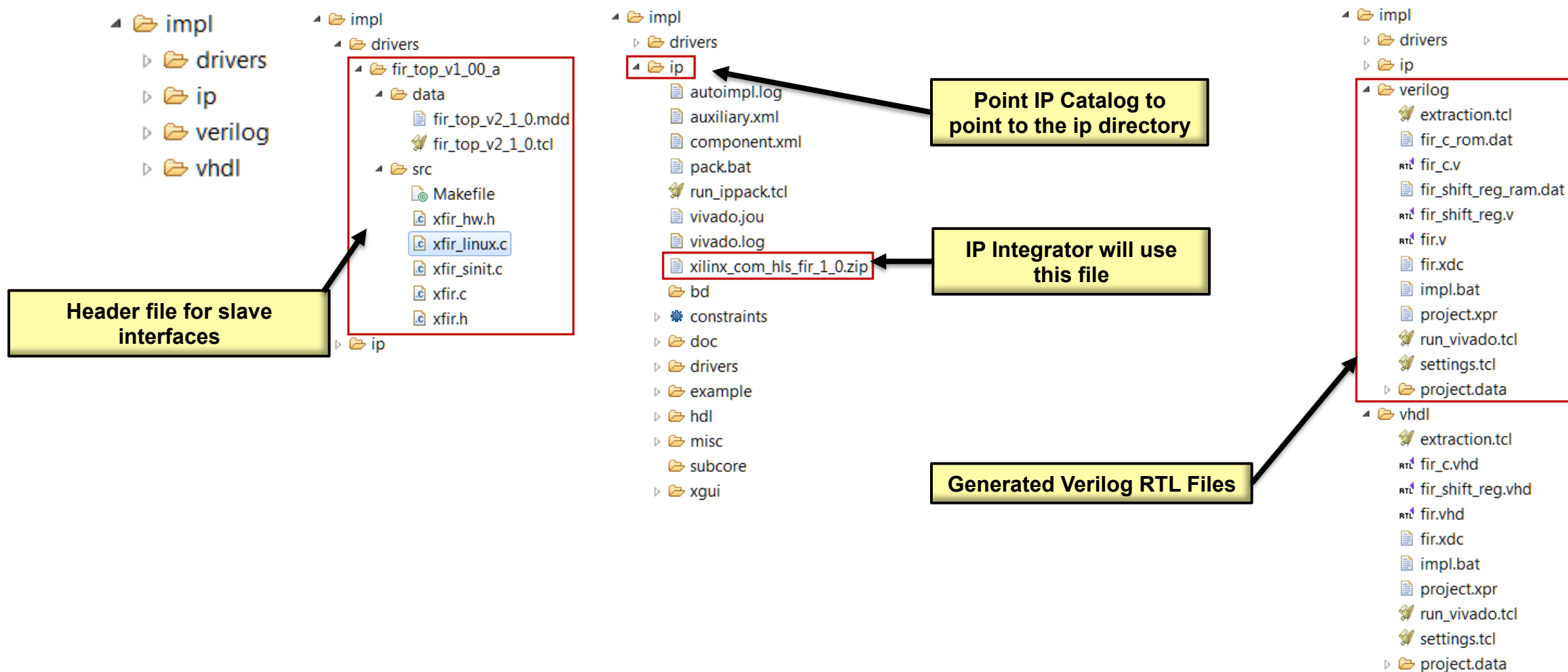
Help Cancel OK

4) Generate the Adapter

- **Select Solution > Export RTL**
- **Select IP Catalog, System Generator for Vivado or PCore for EDK**
- **Click on Configuration... if you want to change the version number or other information**
 - Default is v1_00_a
- **Click on OK**
 - The directory (ip) will be generated under the impl folder under the current project directory and current solution
 - Only RTL code in Verilog will be generated, even if you select VHDL language



Generated impl Directory



Slave Interface Example

➤ Simple Example

```
int foo_top (int *a, int *b, int *c, int *d) {  
  
    // Define the RTL interfaces  
    #pragma HLS interface ap_hs port=a  
    #pragma HLS interface ap_none port=b  
    #pragma HLS interface ap_vld port=c  
    #pragma HLS interface ap_ack port=d  
    #pragma HLS interface ap_ctrl_hs port=return register  
  
    // Define the pcore interfaces and group into AXI4 slave "slv0"  
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a  
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=b  
  
    // Define the pcore interfaces and group into AXI4 slave "slv1"  
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=return  
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=c  
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=d  
  
    *a += *b;  
    return (*c + *d);  
}
```

Port a: synthesized with two-way handshake (ap_hs)
Port b: synthesized with no IO protocol (ap_none)
Port c: synthesized with input valid protocol (ap_vld)
Port d: synthesized with output acknowledge (ap_ack)
Block IO protocols signal added to the design

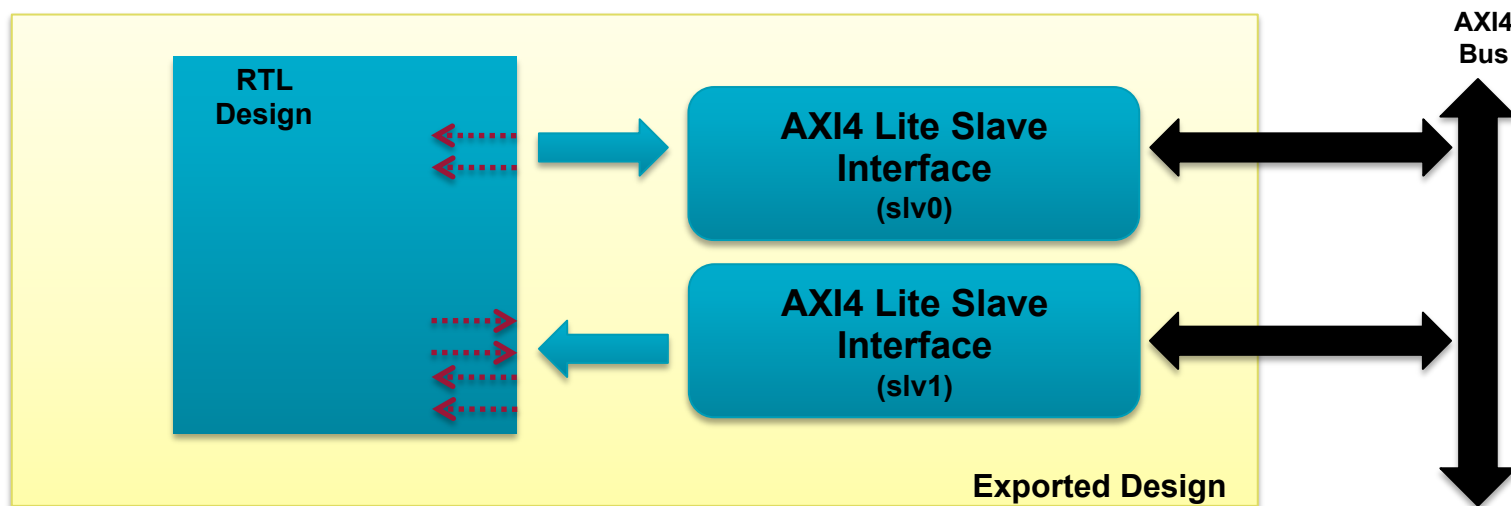
Ports a and b grouped into slave adapter slv0

Ports c, d, return port and the block level IO protocol (associated with the return port) grouped into slave adapter slv1

Slave Interface Example RTL

➤ The output IP consists of the RTL design and two slave interfaces

- The block can be controlled via AXI4 slave interface slv1
 - This is where the Block IO protocol signals are grouped
- Individual ports can be controlled via their associated AXI4 slave interface



Slave Interface Example Software

➤ The following files are available in the include directory

- xfoo_top_slv0.h xfoo_top_slv1.h
- xfoo_top.h xfoo_top.c

➤ The functions in the files can be used on CPU C program

```
// Use this function to instantiate an instance of XFoo_Top in the program space:
int XFoo_top_Initialize(XFoo_top *InstancePtr, XFoo_top_Config *ConfigPtr);

// The following functions set the value on the ports with no handshake protocol
void XFoo_top_SetB(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetD(XFoo_top *InstancePtr, u32 Data);

// The block can be started by issuing a call to the start function : pulses the ap_start signal
// (first, it may be worth checking the status of the idle signal to confirm the last process is indeed finished):
u32 XFoo_top_IsIdle(XFoo_top *InstancePtr);
void XFoo_top_Start(XFoo_top *InstancePtr);

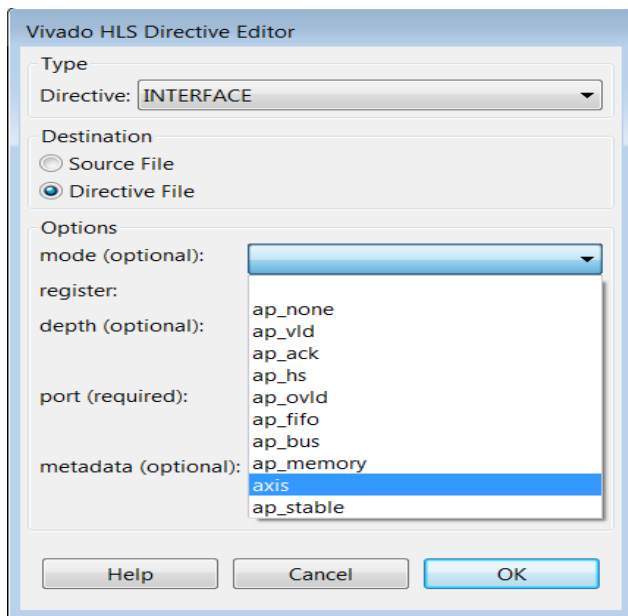
// Write values to ports a and c and set the valid port (or the device will stall)
void XFoo_top_SetA_i(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetA_iVld(XFoo_top *InstancePtr);
void XFoo_top_SetC(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetCVld(XFoo_top *InstancePtr);
```

A complete example is provided in Application Note
Processor Control of Vivado HLS Designs

AXI4 Stream Interface: Ease of Use (Starting in 2013.3)

➤ Native Support for AXI4 Stream Interfaces

- Native = An AXI4 Stream can be specified with `set_directive_interface`
 - No longer required to set the interface then add a resource
 - This AXI4 Stream interface is part of the HDL after synthesis
 - This AXI4 Stream interface is simulated by RTL co-simulation



Interface Type “axis” is AXI4 Stream

`set_directive_interface -mode axis “foo” portA`
Or
`#pragma HLS interface axis port=portA`

Pre-2013.3 Approach to AXI Streams

➤ Existing Functionality Deprecated

- BUT NOT REMOVED!!
- We don't want to break existing designs

➤ Warning:

- If you use the method for adding AXI4 Streams before 2013.3
 - This is where you set the interface as a FIFO then add an AXI Resource
- You will get a FIFO interface in the RTL
- And the AXI4 Stream adapter is added during export_design

➤ Recommendation

- Change existing AXI4 Stream directives to use the INTERFACE directive

```
#if 1
// Use New Method
#pragma HLS interface axis port=portA

#else

// Or use old Method
#pragma HLS interface ap_fifo port=portA
#pragma HLS resource core=AXI4Stream variable=portA \
                    metadata="-bus_bundle Agroup"
#endif
```

AXI4 Master Interface: Pipeline Support

- **Transaction involving an AXI4 Master Interface is now Pipelined**
 - Prior to 2013.3 this interface would not pipeline
 - Each transfer was an “atomic” process
 - The for-loop/memcpy waits until a transfer completes before starting next transfer
 - This was the limiting factor in the pipeline interval
- **Improved performance in 2013.3**
 - Accesses to an AXI master interface can now be pipelined
 - The performance will be much better than before

AXI4 Master Interface: Pipeline Details

➤ The memcpy operation is now automatically pipelined

- Even when no pipeline directive is added
- This can be seen in the log file
- Ensures the performance of the data transfer is optimal

➤ Currently no way to name these loops

- The loop is embedded inside the memcpy
- Names are sequential (1,2,3, etc.) from top to bottom
 - This includes unnamed for-loops: the Analysis Perspective can help identify

```
void example(volatile int *a){  
  
#pragma HLS INTERFACE ap_bus port=a depth=50  
#pragma HLS RESOURCE variable=a core=AXI4M  
  
    int i;  
    int buff[50];  
  
    memcpy(buff,(const int*)a,50*sizeof(int));  
    ....  
}
```

```
@I [SCHED-11] Starting scheduling ...  
@I [SCHED-61] Pipelining loop 'Loop 1'.  
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 1, Depth: 8.
```

AXI4 Master Interface: Pipeline Details (II)

➤ The auto-pipeline of memcpy is ignored

- If the region containing the memcpy is pipelined
- This can be seen in the log file
- Individual accesses can be seen in the Analysis Perspective

```
void example(volatile int *a){  
  
#pragma HLS INTERFACE ap_bus port=a depth=50  
#pragma HLS RESOURCE variable=a core=AXI4M  
#pragma HLS PIPELINE  
    int i;  
    int buff[50];  
  
    memcpy(buff,(const int*)a,50*sizeof(int));  
    ....  
}
```

example.cpp Performance - example

Current Module : example

	Operation\Control S...	C5	C6	C7	C8	C9	C10	C11	C12
1	a req 0(readreq)								
2	buff 0(read)								
3	buff 1(read)								
4	buff 2(read)								
5	buff 3(read)								
6	buff 4(read)								
7	buff 5(read)								
8	buff 6(read)								
9	buff 7(read)								
10	buff 8(read)								
11	buff 9(read)								
12	buff 10(read)								
13	buff 11(read)								
14	buff 12(read)								

@I [HLS-10] Starting code transformations ...

@I [XFORM-502] Unrolling all loops for pipelining in function 'example' (example.cpp:53).

@W [XFORM-505] Ignored pipeline directive for loop 'Loop-1' because its parent loop or function is pipelined.

AXI4 Master Interface: Pipeline Details (III)

➤ The auto-pipeline of memcpy can be disabled

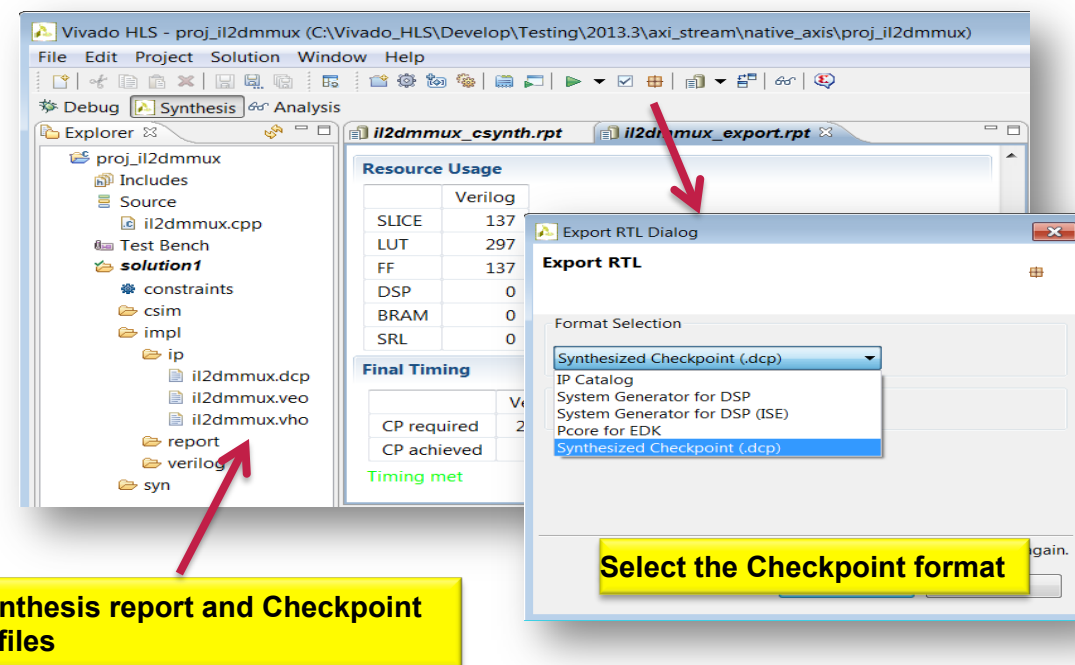
- Create a region around the memcpy
- Use the UNROLL directive in this region
- **NOTE**: Must use the region option
- As with unrolling explicit loops, the burst length must be constant

```
void example(volatile int *a){  
  
#pragma HLS INTERFACE ap_bus port=a depth=50  
#pragma HLS RESOURCE variable=a core=AXI4M  
  
    int i;  
    int buff[50];  
  
    my_region: {  
#pragma HLS UNROLL region  
        memcpy(buff,(const int*)a,50*sizeof(int));  
    }  
    ....  
}
```

No messages about pipelining the memcpy

Export to a Synthesized Checkpoint

- **Design Checkpoint format is now supported**
 - This will appear as one of the options for 7-series & Zynq
 - This is a Vivado format and not supported in ISE flow
- **RTL will be synthesized prior to be packaged**
 - Expect longer run time to export
- **No required to add the IP to IP catalog**
 - Simply read it into Vivado
- **This format does not support AXI interface adapters**



Outline

- Embedded System Design in Zynq using IP Integrator
- Creating IP-XACT Adapters
- *Integrating the IP-XACT Adapter in AXI System*
- Summary

Embedded System Design using Vivado

- **Create a new Vivado project, or open an existing project**
- **Invoke IP Integrator**
- **Construct(modify) the hardware portion of the embedded design by adding the IP-XACT adapter created in Vivado HLS**
- **Create (Update) top level HDL wrapper**
- **[optional] Synthesize any non-embedded components and implement in Vivado**
- **Export the hardware description, and launch SDK**
- **Create a new software board support package and application projects in the SDK**
- **Compile the software with the GNU cross-compiler in SDK**
- **[optional] Download the programmable logic's completed bitstream using iMPACT**
- **[optional] Use SDK to download the program (the ELF file)**

Outline

- Embedded System Design in Zynq using IP Integrator
- Creating IP-XACT Adapters
- Integrating the IP-XACT Adapter in AXI System
- *Summary*

Summary

- **Embedded system development flow in FPGA involves**
 - Developing hardware using IP Integrator and Vivado
 - Developing software using SDK
- **Adapter provides wide support of AXI interfaces, System Generator design, and Pcore for EDK**
 - Assign as an external resource, just like a RAM
 - The choice of adapter is a function of the C variable type (pointer, etc.)
- **Start with the correct C argument type**
 - Verify the design at the C level
 - Accept the default block-level I/O protocol
 - Select the port-level I/O protocol that gives the required pcore adapter interface
 - Specify the port to have the appropriate adapter RESOURCE
 - Optionally group and rename ports