Creating Processor System

Objectives

- **After completing this module, you will be able to:**
  - Describe embedded system development flow in Zynq
  - List the steps involved in creating hardware accelerator
  - State how hardware accelerator created in Vivado HLS is used in Vivado Design Suite

Outline

- *Embedded System Design in Zynq using IP Integrator*
- Creating IP-XACT Hardware Accelerator
- Integrating the IP-XACT Hardware Accelerator 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 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 SoC architecture consists of two major sections**
  - PS: Processing system
    - Single/Dual ARM Cortex-A9 processor based (Single core versions available)
    - Multiple peripherals
    - Hard silicon core
  - PL: Programmable logic
    - Uses the same 7 series programmable logic

Features	Zynq-7000S	Zynq-7000	
Devices	Z-7007S, Z-7012S, Z-7014S	Z-7010, Z-7015, Z-7020	Z-7030, Z-7035, Z-7045, Z-7100
Processor Core	Single-core ARM® Cortex™-A9 MPCore™	Dual-core ARM Cortex-A9 MPCore	
Maximum Frequency	Up to 766MHz	Up to 866 MHz	Up to 1GHz
External Memory Support	DDR3, DDR3L, DDR2, LPDDR2		
Key Peripherals	USB 2.0, Gigabit Ethernet, SD/SDIO		
Dedicated Peripheral Pins	Up to 128	Up to 128	128

Vivado

- **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 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

Embedded System Tools: Hardware

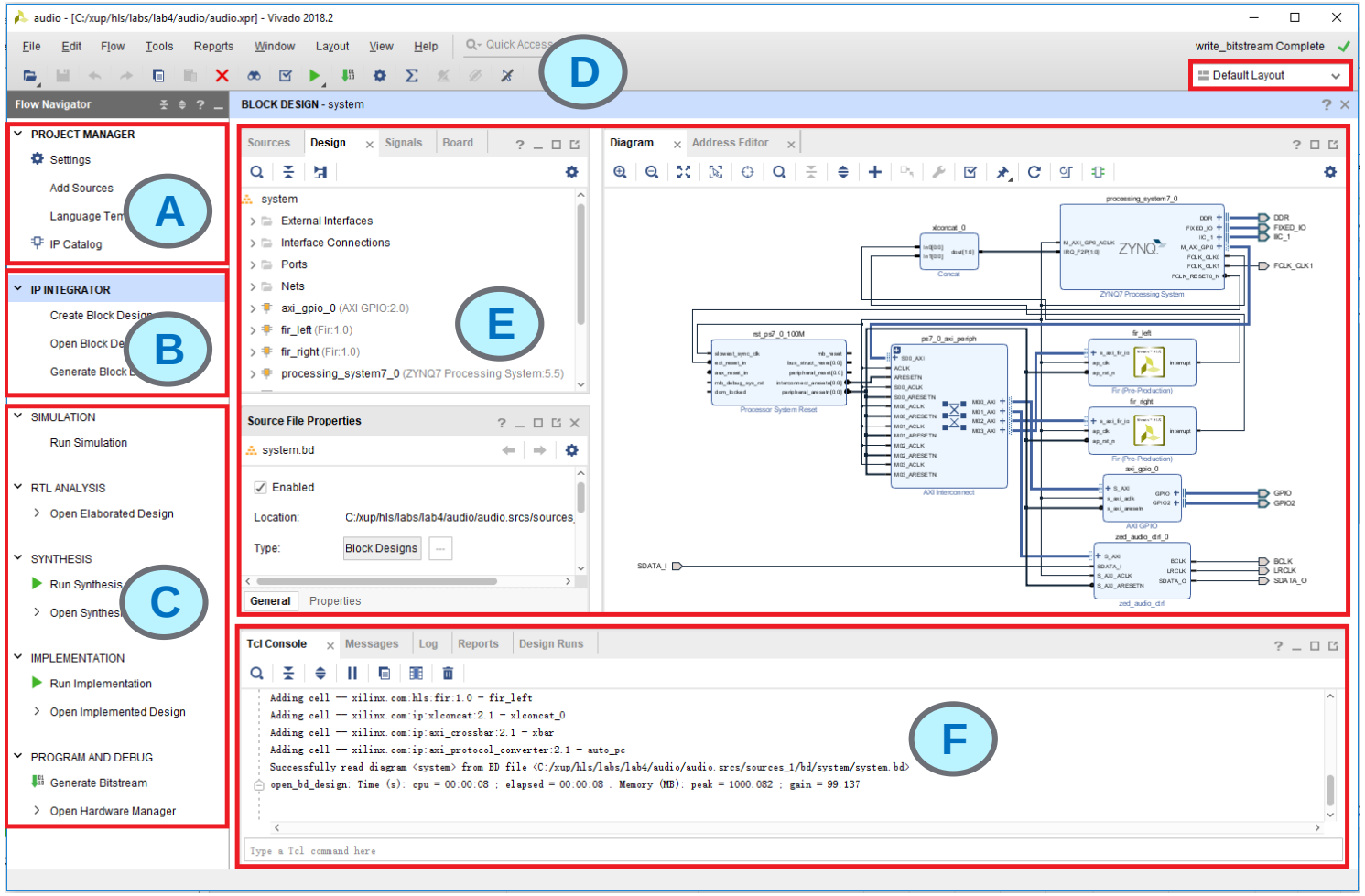
- **Hardware development tools**
  - IP Integrator
  - IP Packager
  - Hardware netlist generation
  - Simulation model generation
  - Hardware debugging using Vivado analyzer cores

Embedded System Tools: Software

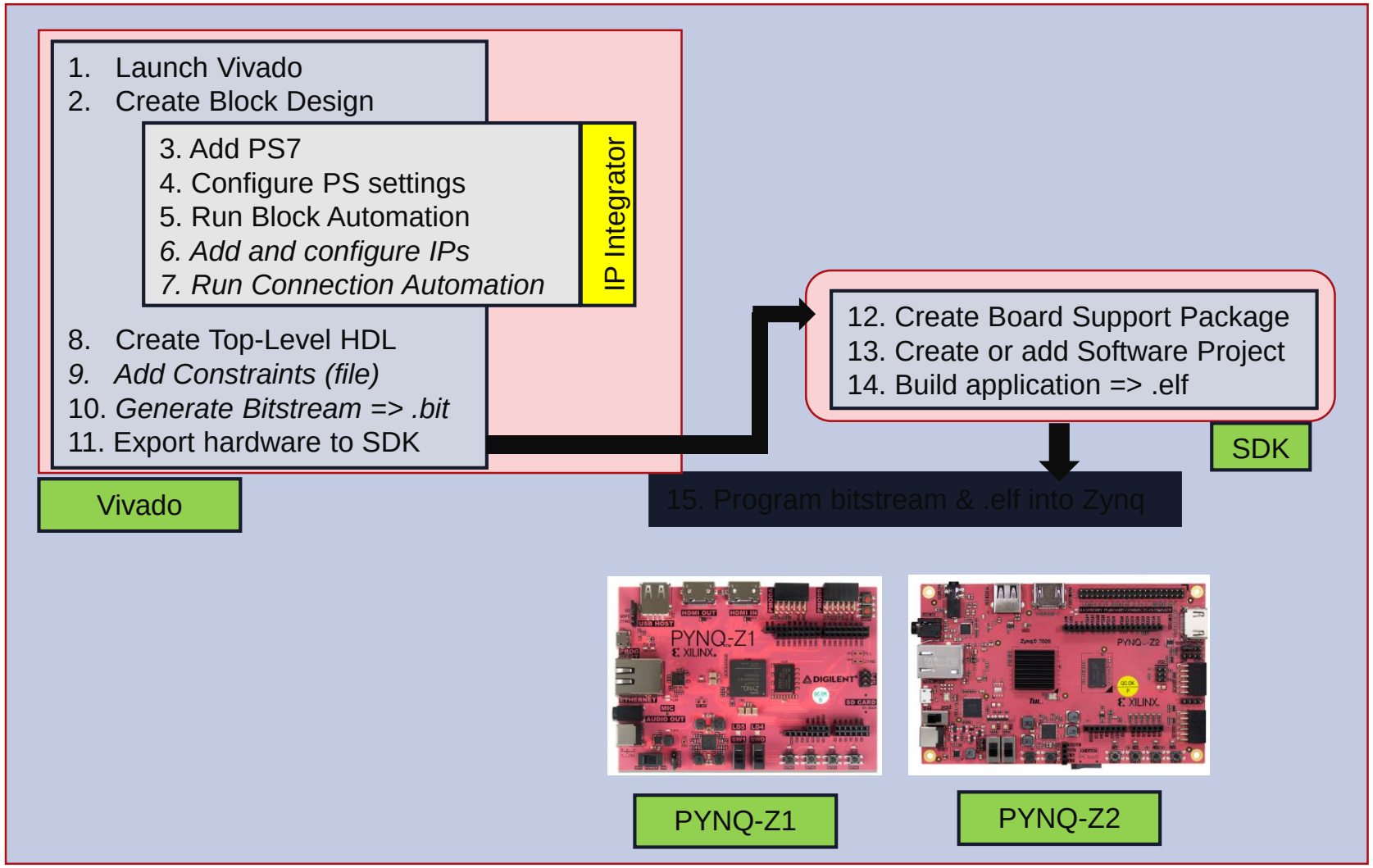
- **Eclipse IDE-based Software Development Kit (SDK)**
  - Board support package creation
  - GNU software development tools
  - C/C++ compiler for the MicroBlaze and ARM Cortex-A9 processors (gcc)
  - Debugger for the MicroBlaze and ARM Cortex-A9 processors (system debugger)
  - TCF framework – multicore 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

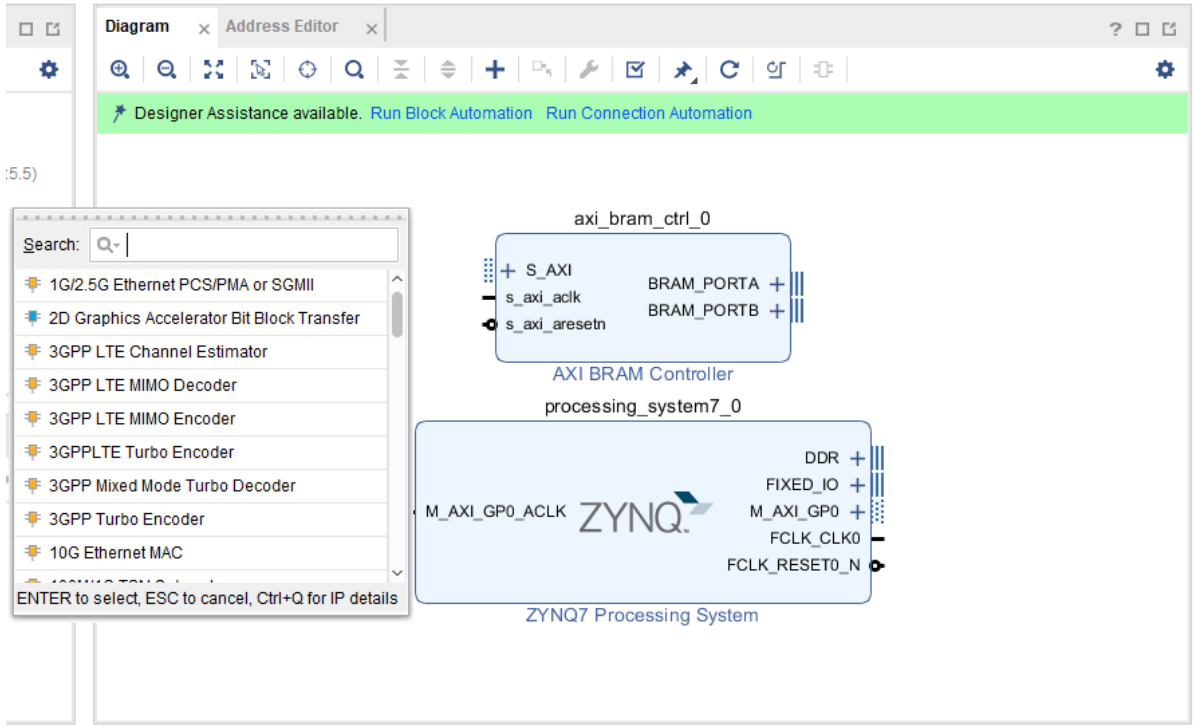


Embedded System Design using Vivado



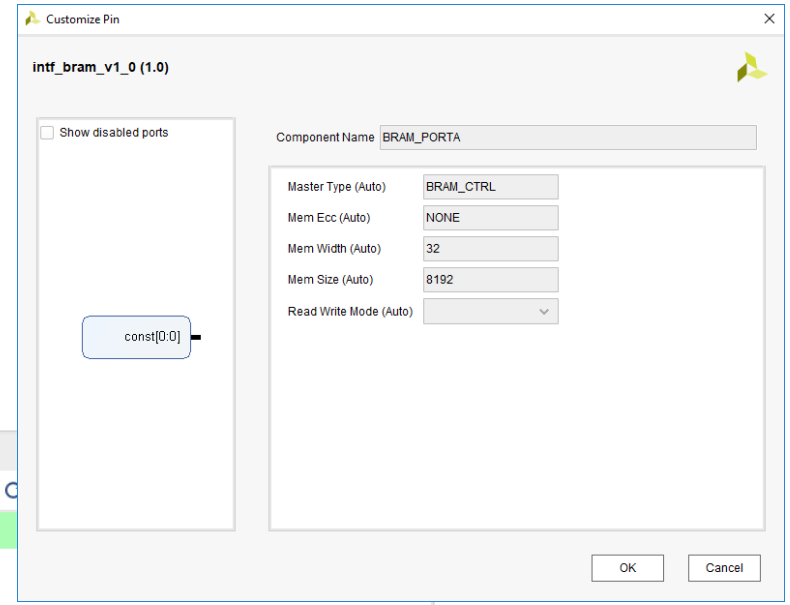
Add IP Integrator Block Diagram

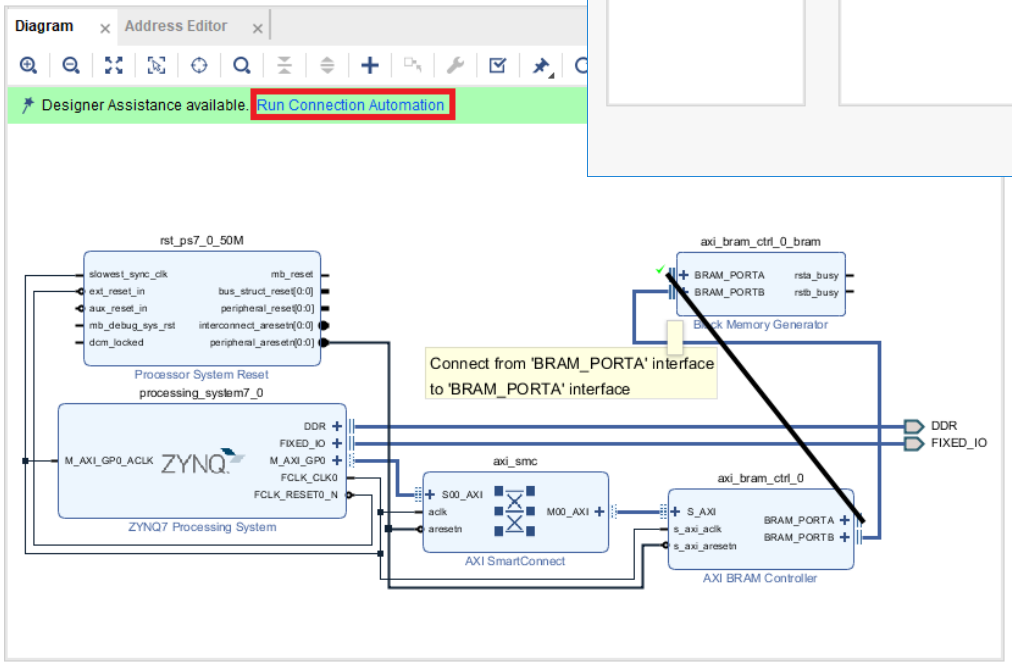
- **IP Integrator Block Diagram opens a blank canvas**
- **IP can be added from the IP catalog**
- **Drag and drop interface**
- **Intelligent Design environment**
  - Design Assistance
  - Connection automation
  - Highlights valid connections
  - Group, create hierarchal blocks
- **Can import custom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

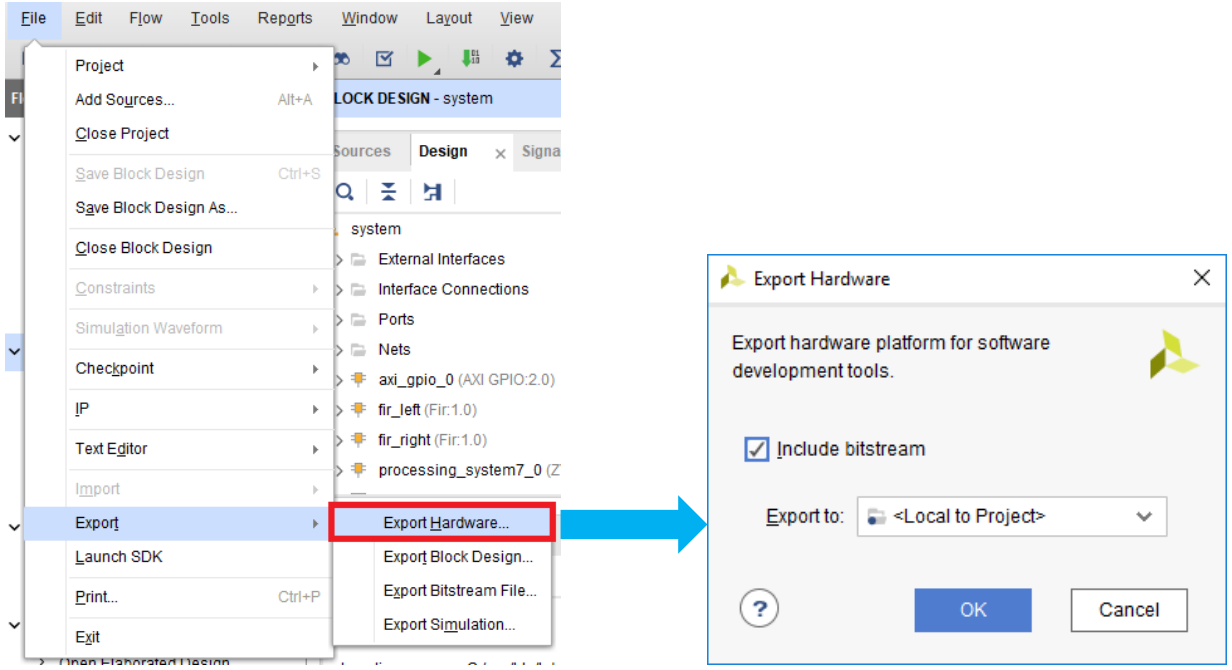


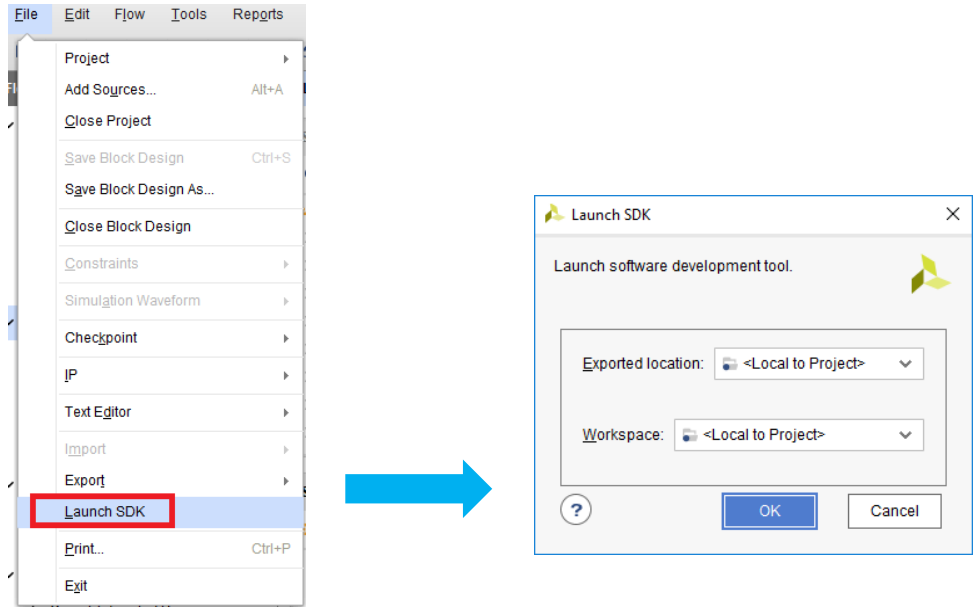


XILINX.

Exporting to XSDK

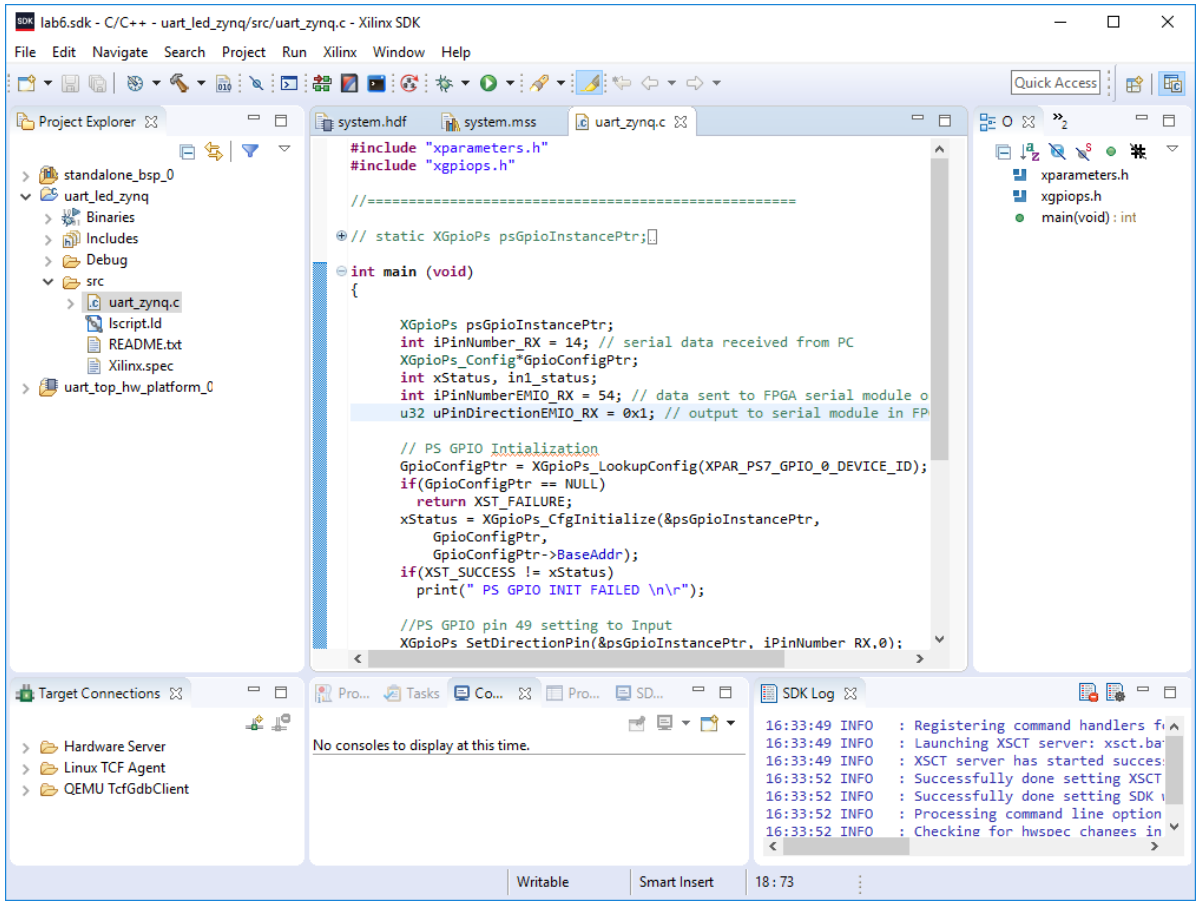
- **Export hardware first**
  - The Hardware Description File (hdf) format file containing all the relevant information will be created and placed under the *.sdk directory
  - Include bitstream if generated
- **Launch XSDK**
  - Software development is performed with the Xilinx Software Development Kit tool (XSDK)
- **The XSDK tool will then associate user software projects to hardware**





Software Development Flow

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

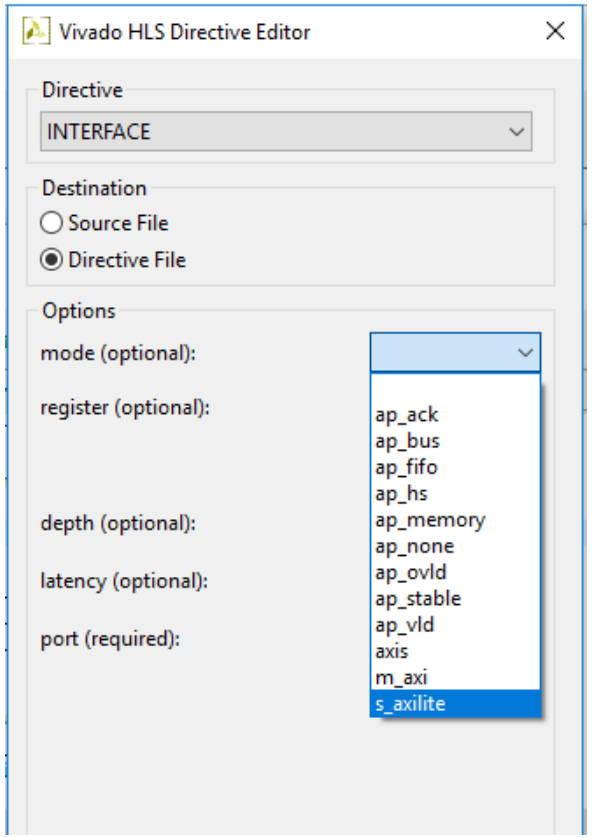


Creating IP-XACT Hardware Accelerator

Port-Level Interfaces

- **The AXI4 interfaces supported by Vivado HLS include**
  - The AXI4-Stream (axis)
    - Specify on input arguments or output arguments only, not on input/output arguments
  - The AXI4 master (m_axi)
    - Specify on arrays and pointers (and references in C++) only. You can group multiple arguments into the same AXI4-Lite interface using the `bundle` option
  - The AXI4-Lite (s_axilite)
    - Specify on any type of argument except arrays. You can group multiple arguments into the same AXI4-Lite interface using the `bundle` option

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=a      bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=b      bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=c      bundle=BUS_A offset=0x0400
#pragma HLS INTERFACE ap_vld port=b

  *c += *a + *b;
}
```



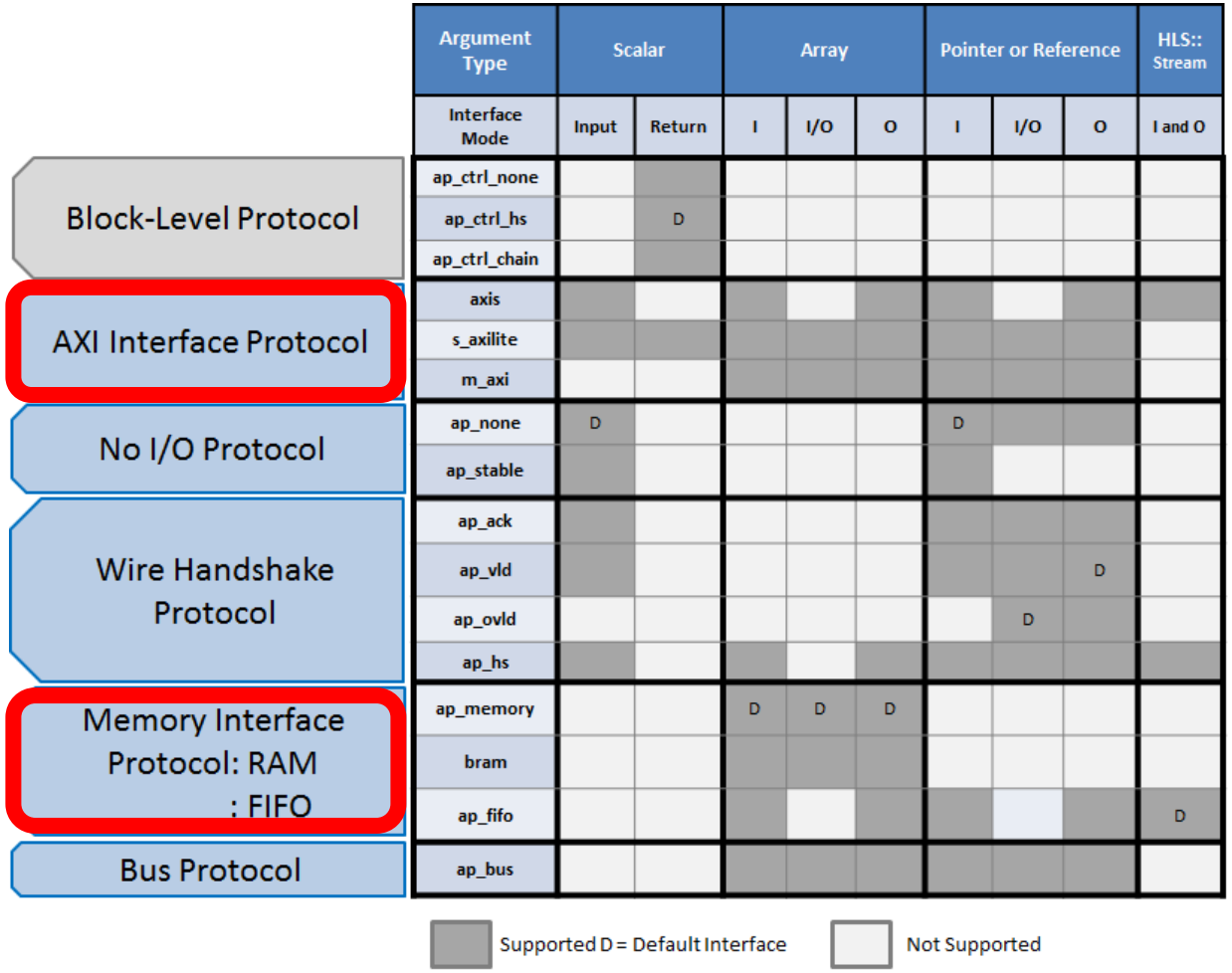
Interface Modes

> Native AXI Interfaces

- >> AXI4 Slave Lite, AXI4 Master, AXI Stream supported by INTERFACE directive
  - – Provided in RTL after Synthesis
  - – Supported by C/RTL Co-simulation
  - – Supported for Verilog and VHDL

> BRAM Memory Interface

- >> Identical IO protocol to ap_memory
- >> Bundled differently in IP Integrator
  - – Provides easier integration to memories with BRAM interface

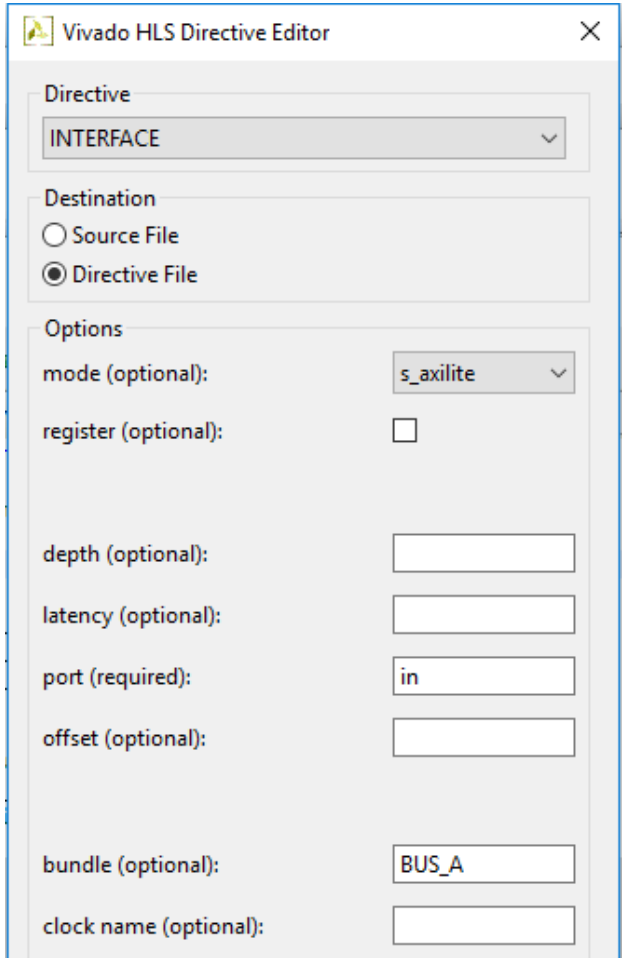


XILINX.

Native AXI Slave Lite Interface

- **Interface Mode: s_axilite**
  - Supported with INTERFACE directive
  - Multiple ports may be grouped into the same Slave Lite interface
    - All ports which use the same bundle name are grouped
- **Grouped Ports**
  - Default mode is ap_none for input ports
  - Default mode is ap_vld for output ports
  - Default mode ap_ctrl_hs for function (return port)
  - Default mode can be changed with additional INTERFACE directives

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE s_axilite port=return  bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=a       bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=b       bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=c       bundle=BUS_A
#pragma HLS INTERFACE ap_hs     port=a
#pragma HLS INTERFACE ap_vld    port=b
#pragma HLS INTERFACE ap_none   port=c register
```



Controllable Register Maps in AXI4 Lite

- **Assigning offset to array (RAM) interfaces**
  - Specified value is offset to base of array
  - Array's address space is always contiguous and linear

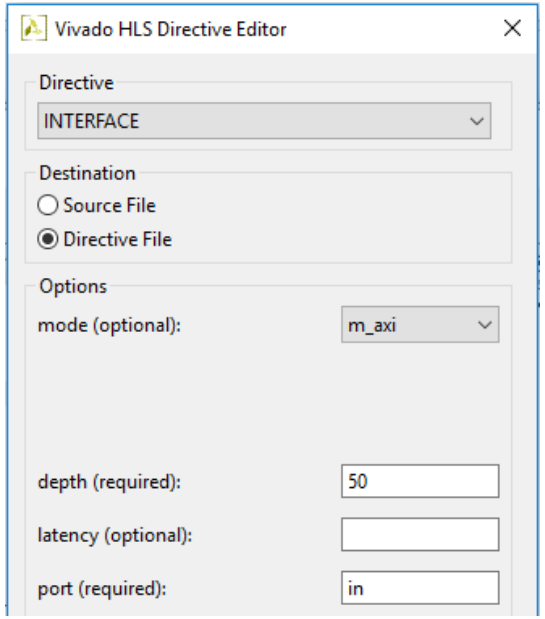
```
void hls_sig_gen_bram2axis(hls::stream<axis_last_t<data_t> >& dout,
                           data_t sig_buf[MAX_SIG_PERIOD], short sig_period)
{
#pragma HLS INTERFACE port=return     s_axilite bundle=ctrl
#pragma HLS INTERFACE port=sig_buf    s_axilite bundle=ctrl offset=0x1000
#pragma HLS INTERFACE port=sig_period s_axilite bundle=ctrl offset=0x0400
```

- **C Driver Files include offset information**
  - In generated driver file xhls_sig_gen_bram2axis.h

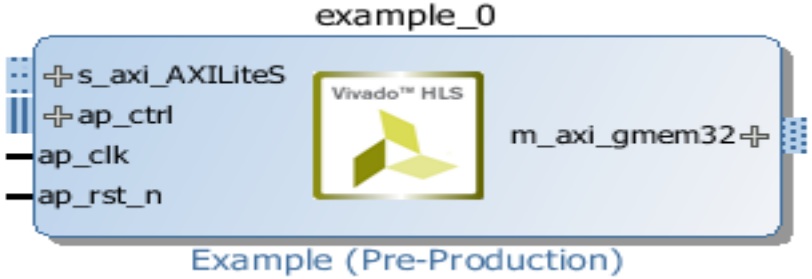
```
...
#define XHLS_SIG_GEN_BRAM2AXIS_CTRL_ADDR_SIG_PERIOD_DATA 0x0400
...
#define XHLS_SIG_GEN_BRAM2AXIS_CTRL_ADDR_SIG_BUF_BASE    0x1000
#define XHLS_SIG_GEN_BRAM2AXIS_CTRL_ADDR_SIG_BUF_HIGH    0x17ff
...
```

Native AXI4 Master

- **Interface Mode: m_axi**
  - Supported with INTERFACE directive
- **Options**
  - Multiple ports may be grouped into the same AXI4 Master interface
    - All ports which use the same bundle name are grouped
  - Depth option is required for C/RTL co-simulation
    - Required for pointers, not arrays
    - Set to the number of values read/written
  - Option to support offset or base address

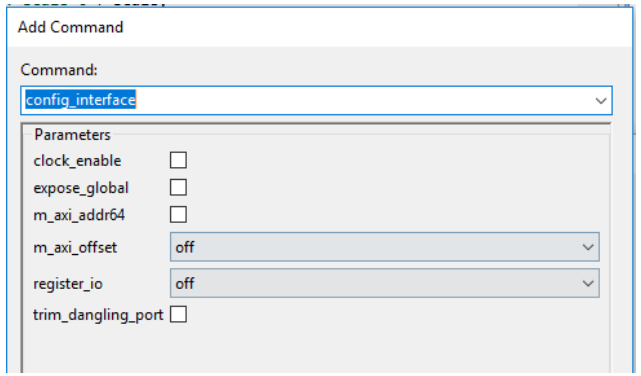


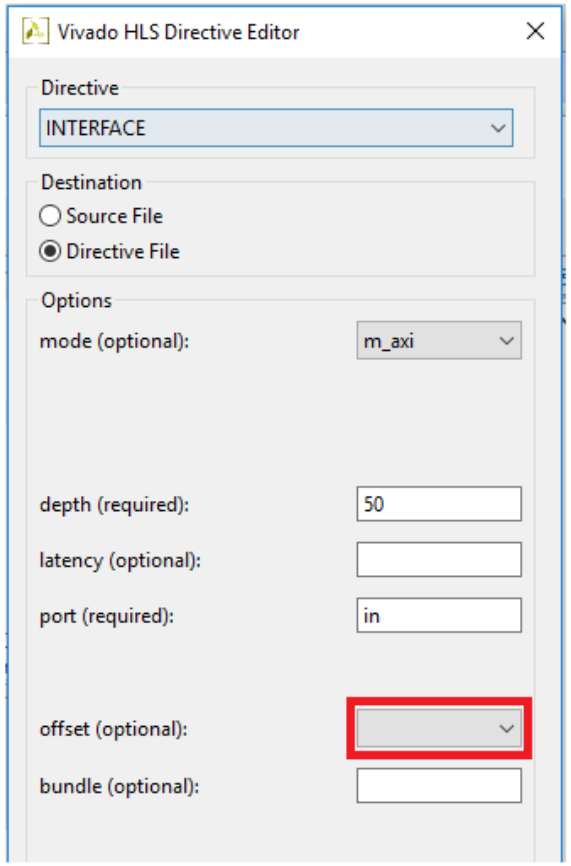

```
void example(volatile int *a)
{

#pragma HLS INTERFACE m_axi depth=50 port=a
```



Native AXI4 Master : Offset Support

- **Address Offset / Base Address Support**
  - Support provided for address offset
- **Port Offset**
  - Defines the offset for the port
  - May be set on individual interfaces using the INTERFACE directive
- **Global Offset**
  - Globally controls the offset ports of all M_AXI interface in the design
  - May be set using the interface configuration
    - Using Tcl command config_interface -m_axi_offset option

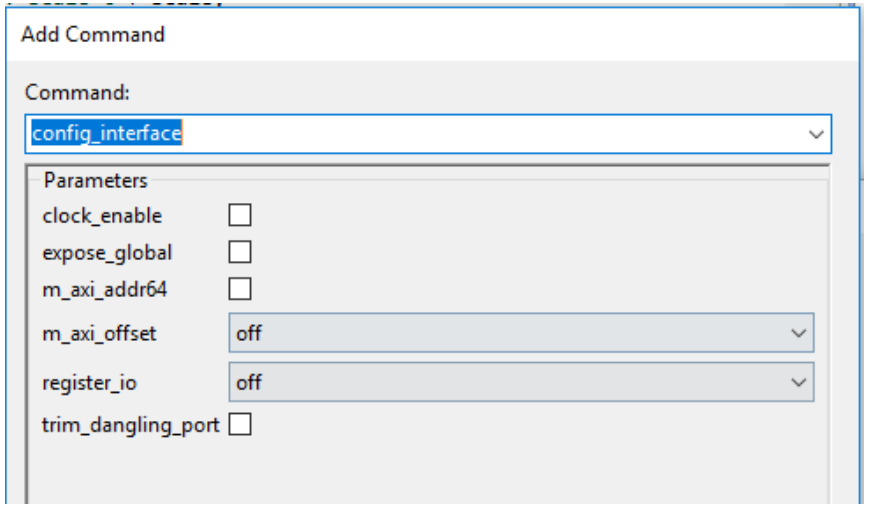


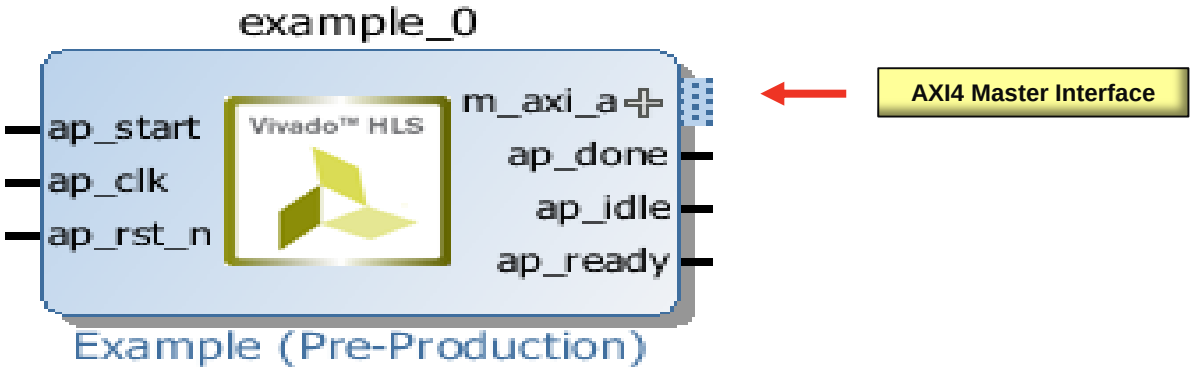


XILINX

Native AXI4 Master: Offset=off (default)

- **Default AXI4 Master Interface**
  - No offset is provided for the address
    - Same as existing behavior
  - The offset (BASEADDR) is set IPI
    - Using IP customization GUI
  - The offset can not be changed on the fly



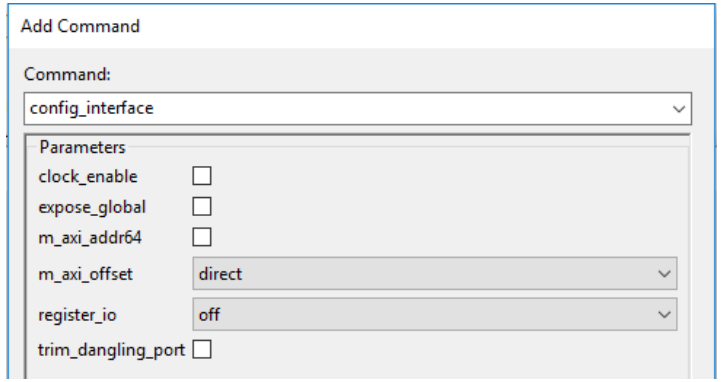


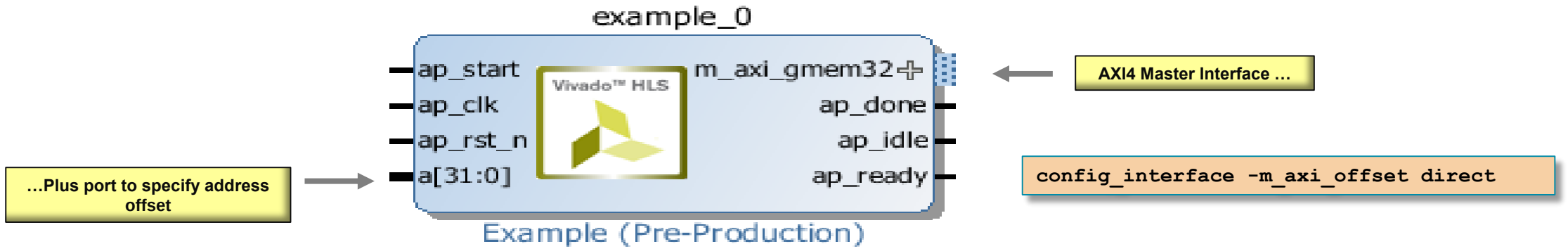
AXI4 Master Interface

```
config_interface -m_axi_offset off
```

Native AXI4 Master: Offset=direct

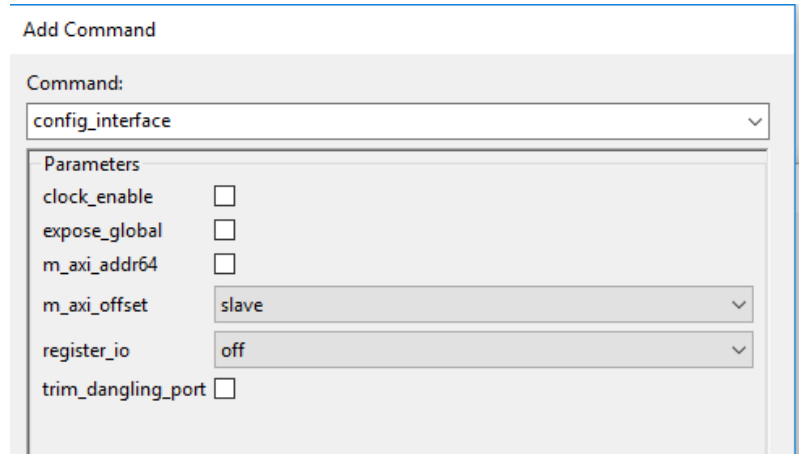
- **Direct Interface**
  - Generates a scalar input offset port
  - The offset is set by driving the input port
  - It can be changed on the fly by driving the port with a different value

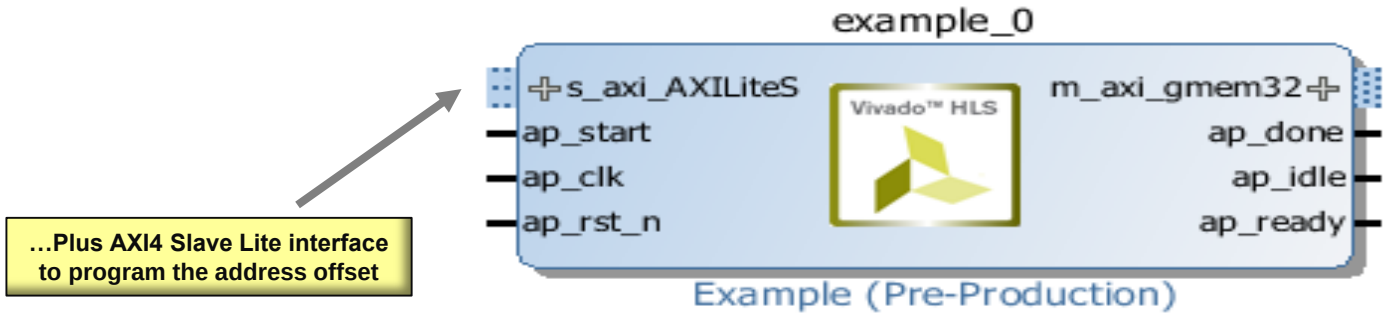




Native AXI4 Master: Offset=slave

- **Direct Interface**
  - Generates an offset port and automatically maps it to an AXI4 Slave Lite interface
  - User must program the offset before starting transactions on the AXI4 Master interface
  - It can changed on the fly by re-programming the offset register





AXI4 Master Interface …

…Plus AXI4 Slave Lite interface to program the address offset

```
config_interface -m_axi_offset slave
```


XILINX

Burst Accesses Inferred for AXI4 Master

- **There are two types of accesses on an AXI Master**

Single Access

```
void example(int *a)
{
#pragma HLS INTERFACE m_axi port=a depth=...
   ...
   int val[i] = *(a + i);
   ...
```

Burst Access

```
void example(int *a)
{
#pragma HLS INTERFACE m_axi port=a depth=...
   ...
   memcpy(vals, a, N * sizeof(int));
   ...
```

  - Burst accesses are more efficient
  - Burst access has until now required the use of memcpy()

- **Burst Accesses are now inferred**
  - From operations in a for-loop and from sequential operations in the code
  - However: there are some limitations
    - Single for-loops only, no nested loops

Byte-Enable Accesses on AXI4 Master

- **Byte-Enable Accesses Support on AXI4 Master Interfaces**
  - Single bytes are now written and read
  - Improved AXI4 Master performance
- **Improved Performance**
  - This code uses 8-bit data

```
void example(volatile char *a) {

#pragma HLS INTERFACE m_axi depth=50 port=a
```

    - Previously, accessing this required reading/writing full 32-bit
    - This implied a required read-modify-write behavior: Impacted performance
  - Similar performance improvement when accessing struct members
    - Also often implied read-modify-write behavior
  - Improved Port Bundling
    - Variables of different sizes can be grouped into same AXI4 Master port

AXI4 Port Bundling

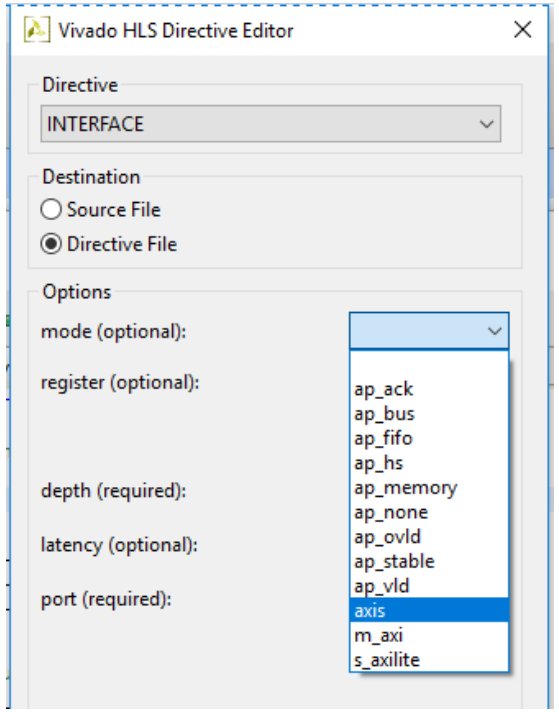
- **AXI4 Master and Lite Port Bundling**
  - The bundle options groups arguments into the same AXI4 port
  - For example, group 3 arguments into AXI4 port “ctrl” :

```
void hls_sig_gen_bram2axis(hls::stream<data_t>& dout,
      data_t sig_buf[MAX_SIG_PERIOD], short sig_period)
{
#pragma HLS INTERFACE port=return     s_axilite bundle=ctrl
#pragma HLS INTERFACE port=sig_buf    s_axilite bundle=ctrl
#pragma HLS INTERFACE port=sig_period s_axilite bundle=ctrl
#pragma HLS INTERFACE port=dout axis
```

- **Arguments can be Bundled into AXI4 Master and AXI4 Lite ports**
  - If no bundle name is used a default name is used for all arguments
    - All go into a single AXI4 Master or AXI4 Lite
    - Default name applied if no –bundle option is used
  - Group different sized variables into an AXI4 Master port

AXI4 Stream Interface: Ease of Use

- **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

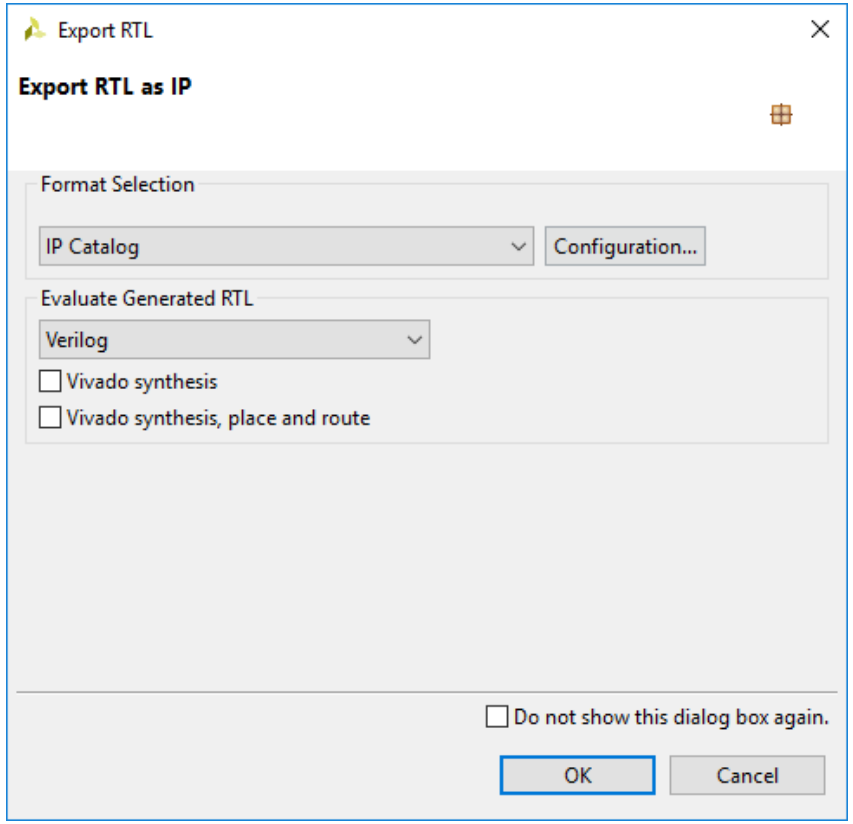


Interface Type "axis" is AXI4 Stream

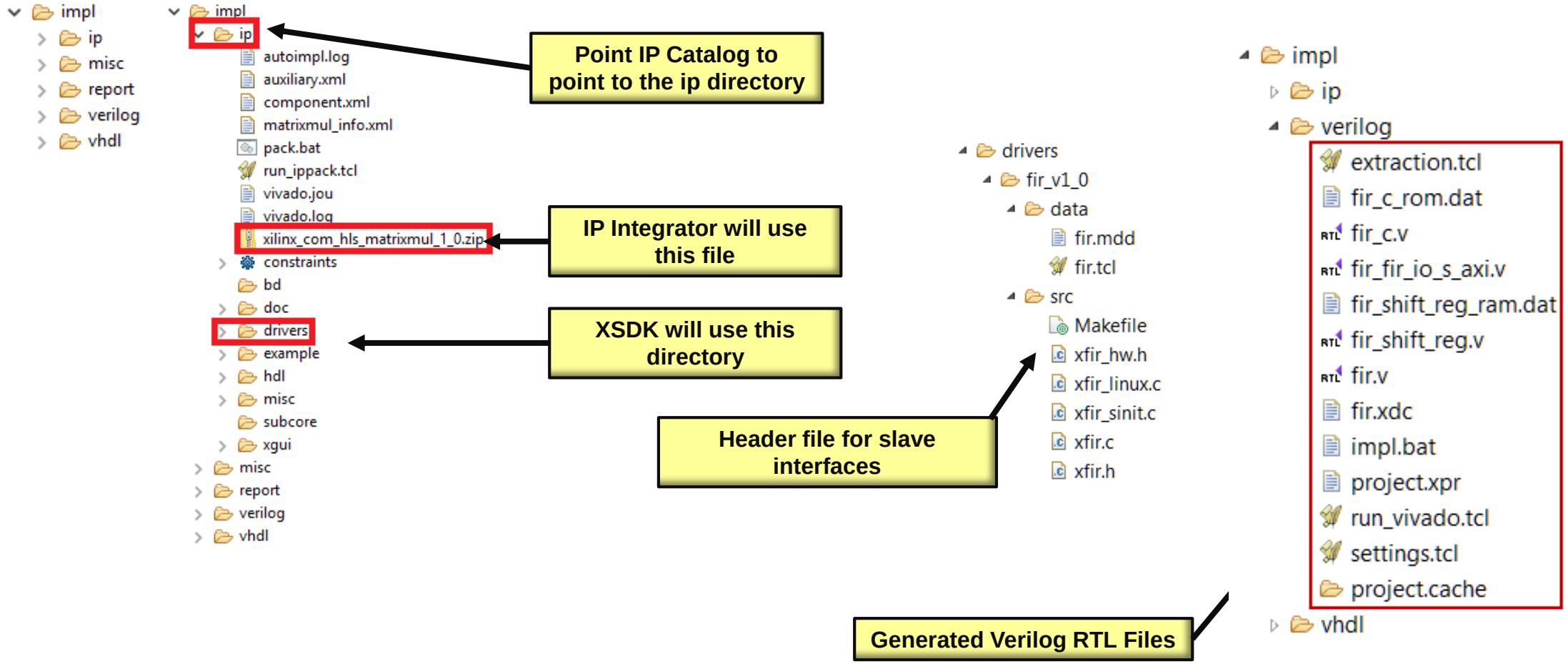
```
set_directive_interface –mode axis "foo" portA
Or
#pragma HLS interface axis port=portA
```

Generate the hardware accelerator

- **Select Solution > Export RTL**
- **Select IP Catalog, System Generator for Vivado or design check point (dcp)**
- **Click on Configuration… if you want to change the version number or other information**
  - Default is v1.0
- **Click on OK**
  - The directory (ip) will be generated under the impl folder under the current project directory and current solution
  - RTL code will be generated, both for Verilog and VHDL languages in their respective folders



Generated impl Directory



C Driver API for AXI4-Lite Interface

API Function	Description
XExample_Initialize	This API will write value to InstancePtr which then can be used in other APIs. It is recommended to call this API to initialize a device except when an MMU is used in the system.
XExample_CfgInitialize	Initialize a device configuration. When a MMU is used in the system, replace the base address in the XDut_Config variable with virtual base address before calling this function. Not for use on Linux systems.
XExample_LookupConfig	Used to obtain the configuration information of the device by ID. The configuration information contain the physical base address. Not for user on Linux.
XExample_Release	Release the uio device in linux. Delete the mappings by munmap: the mapping will automatically be deleted if the process terminated. Only for use on Linux systems.
XExample_Start	Start the device. This function will assert the `ap_start` port on the device. Available only if there is `ap_start` port on the device.
XExample_IsDone	Check if the device has finished the previous execution: this function will return the value of the ap_done port on the device. Available only if there is an ap_done port on the device.
XExample_IsIdle	Check if the device is in idle state: this function will return the value of the ap_idle port. Available only if there is an ap_idle port on the device.
XExample_IsReady	Check if the device is ready for the next input: this function will return the value of the ap_ready port. Available only if there is an ap_ready port on the device.
XExample_Continue	Assert port ap_continue. Available only if there is an ap_continue port on the device.
XExample_EnableAutoRestart	Enables "auto restart" on device. When this is set the device will automatically start the next transaction when the current transaction completes.
XExample_DisableAutoRestart	Disable the "auto restart" function.
XExample_Set_ARG	Write a value to port ARG (a scalar argument of the top function). Available only if ARG is input port.
XExample_Set_ARG_vld	Assert port ARG_vld. Available only if ARG is an input port and implemented with an ap_hs or ap_vld interface protocol.
XExample_Set_ARG_ack	Assert port ARG_ack. Available only if ARG is an output port and implemented with an ap_hs or ap_ack interface protocol.
XExample_Get_ARG	Read a value from ARG. Only available if port ARG is an output port on the device.
XExample_Get_ARG_vld	Read a value from ARG_vld. Only available if port ARG is an output port on the device and implemented with an ap_hs or ap_vld interface protocol.
XExample_InterruptGlobalEnable	Enable the interrupt output. Interrupt functions are available only if there is ap_start.
XExample_InterruptGlobalDisable	Disable the interrupt output.
XExample_InterruptEnable	Enable the interrupt source. There may be at most 2 interrupt sources (source 0 for ap_done and source 1 for ap_ready)
XExample_InterruptDisable	Disable the interrupt source.
XExample_InterruptClear	Clear the interrupt status.
XExample_InterruptGetEnabled	Check which interrupt sources are enabled.
XExample_InterruptGetStatus	Check which interrupt sources are triggered.

Integrating the IP-XACT Hardware Accelerator in AXI System

XILINX

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 hardware accelerator created in Vivado HLS
- Create (Update) top level HDL wrapper
- Synthesize any non-embedded components and implement in Vivado
- Export the hardware description, and launch XSDK
- Create a new software board support package and application projects in the XSDK
- Compile the software with the GNU cross-compiler in XSDK
- Download the programmable logic's completed bitstream using Xilinx Tools > Program FPGA in XSDK
- Use XSDK to download and execute the program (the ELF file)

Summary

Summary

- **Embedded system development flow in FPGA involves**
  - Developing hardware using IP Integrator and Vivado
  - Developing software using XSDK
- **hardware accelerator provides wide support of AXI interfaces, System Generator design, and design check point(dcp)**
  - Use the INTERFACE directive
  - The choice of hardware accelerator 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 hardware accelerator interface
  - Optionally group ports