

Reconfigurable Architecture (7)

osana@eee.u-ryukyu.ac.jp

Today's contents

- * Previously in this class:
 - * HDL design: Combinational, Sequential + Testbench + Simulation / Implement
 - * IP-based design: **IP cores in RTL** design
- * Today: **RTL in IP**-based design
 - * Designing processor-based systems

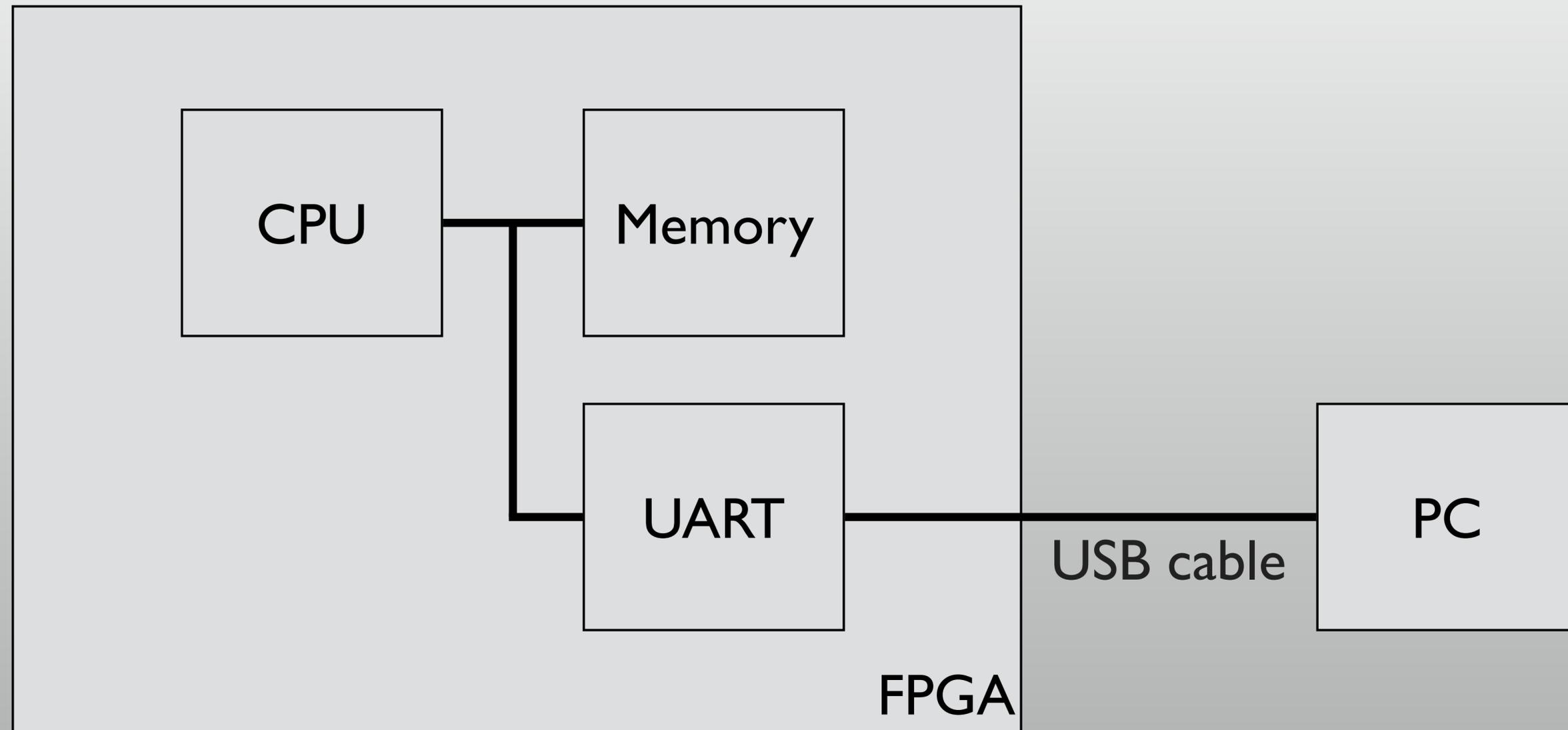
Processor-based systems

- * Has a microprocessor as the “core” of system
 - * Softcore processor is an IP core of FPGA
 - * Configurable along users’ requirement
 - * Some FPGAs have hardcore processor (ARM or PowerPC)
 - * Other IP cores and RTL modules are the peripherals

Xilinx MicroBlaze Processor

- * Highly customizable, 32-bit RISC microprocessor
 - * Area Optimized \leftrightarrow Performance Optimized + many options:
 - * Floating point units, Integer multiplier, etc.
 - * Instruction / Data caches for external DDR memory usages
 - * Exceptions and MMU (memory management unit) for Linux and other OS
 - * Various AXI peripherals

Simplest processor system



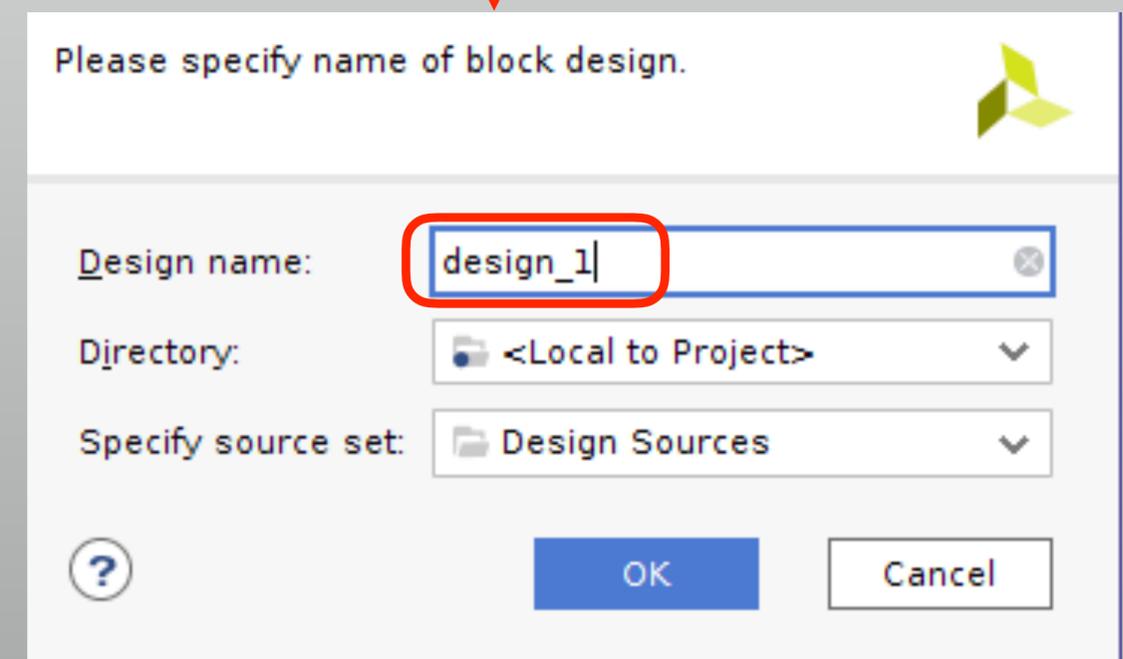
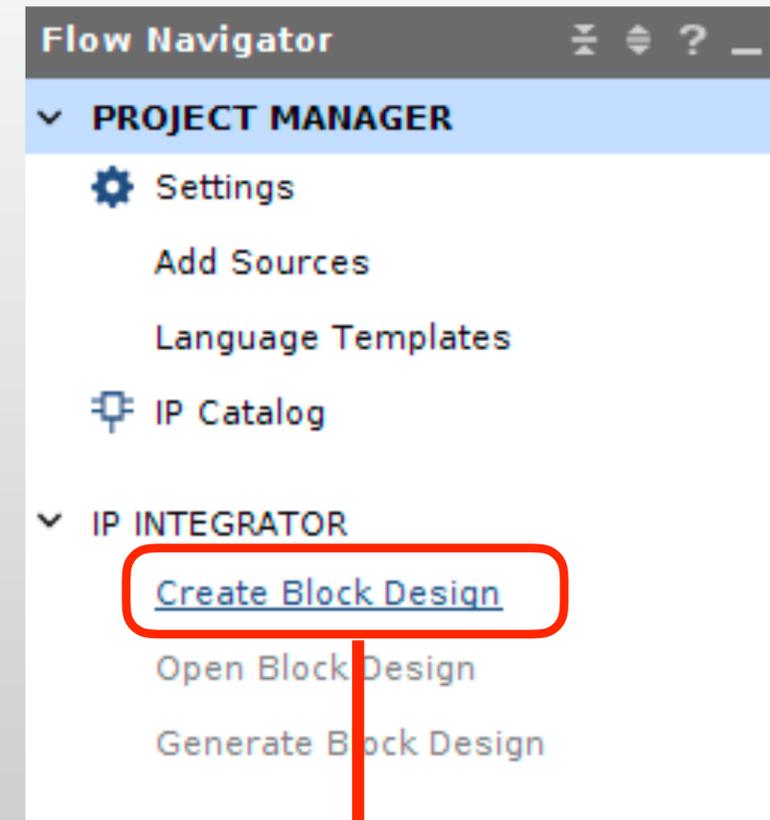
Hands-on: “Hello World” from FPGA

- * Using Xilinx’s MicroBlaze processor
- * With FPGA’s internal BlockRAM as the main memory
- * Xilinx’s UART-Lite core as the console device

Setting up

- * Create a Vivado RTL project
- * Device: xc7a325tcsg-1
- * “Create Block Design” → “design_1”
- * Empty block diagram is open

This design is empty. Press the **+** button to add IP.



Add processor core

- * “+” to add IP core
- * Find “MicroBlaze”
- * Run block automation
 - * Change Local memory size and clock connection

The screenshot illustrates the steps to add a processor core in the Xilinx IDE. It shows the 'Diagram' toolbar with the '+' icon highlighted. A search box contains 'Micro', showing three matches: 'MicroBlaze', 'MicroBlaze Debug Module (MDM)', and 'MicroBlaze MCS'. A green notification bar indicates 'Designer Assistance available' with a 'Run Block Automation' button. The 'Options' dialog is open, showing configuration settings for the core, with 'Local Memory' set to 64KB and 'Clock Connection' set to 'New External Port (100 MHz)'. Red boxes and arrows highlight the key elements in the workflow.

Diagram

Search: (3 matches)

- MicroBlaze
- MicroBlaze Debug Module (MDM)
- MicroBlaze MCS

ENTER to select, ESC to cancel, Ctrl+Q for IP details

Designer Assistance available. [Run Block Automation](#)

Options

Preset	None
Local Memory	64KB
Local Memory ECC	None
Cache Configuration	None
Debug Module	Debug Only
Peripheral AXI Port	Enabled
<input type="checkbox"/> Interrupt Controller	
Clock Connection	New External Port (100 MHz)

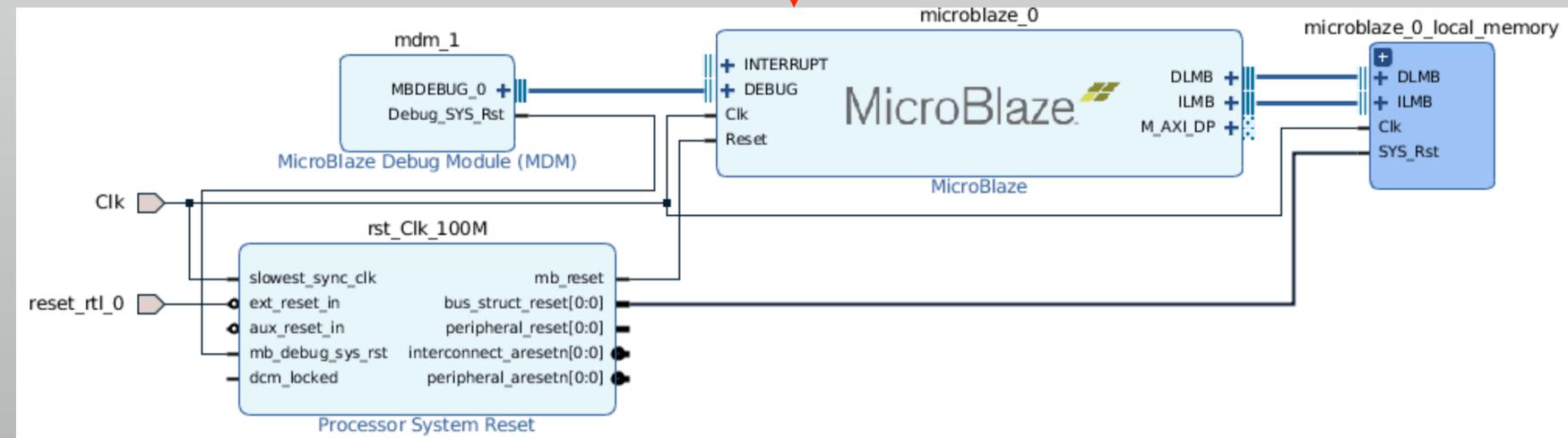
Reset settings

- * “Connection automation” then set reset polarity
- * And you’re done for base system with CPU and BRAM

★ Designer Assistance available. Run Connection Automation

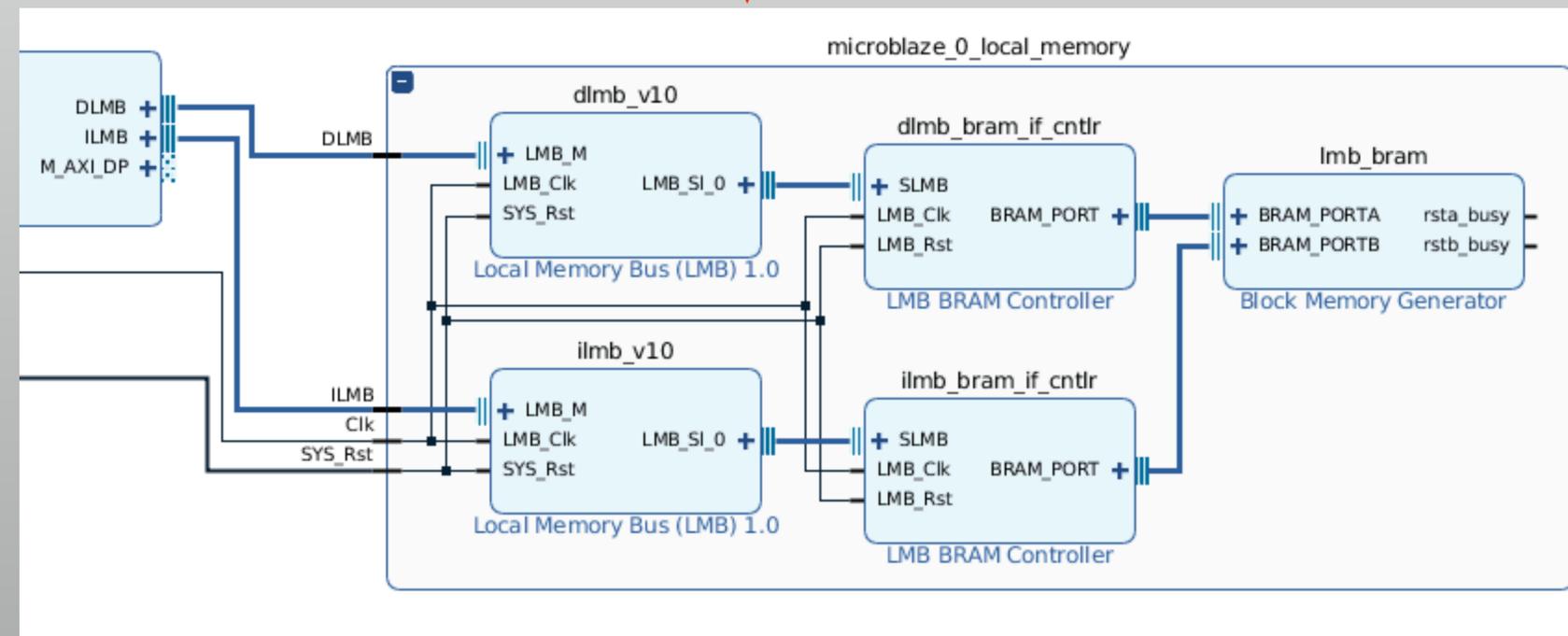
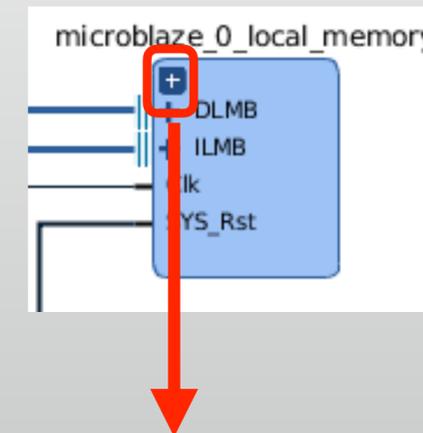
Options

Select Reset Polarity **ACTIVE_HIGH** ▾



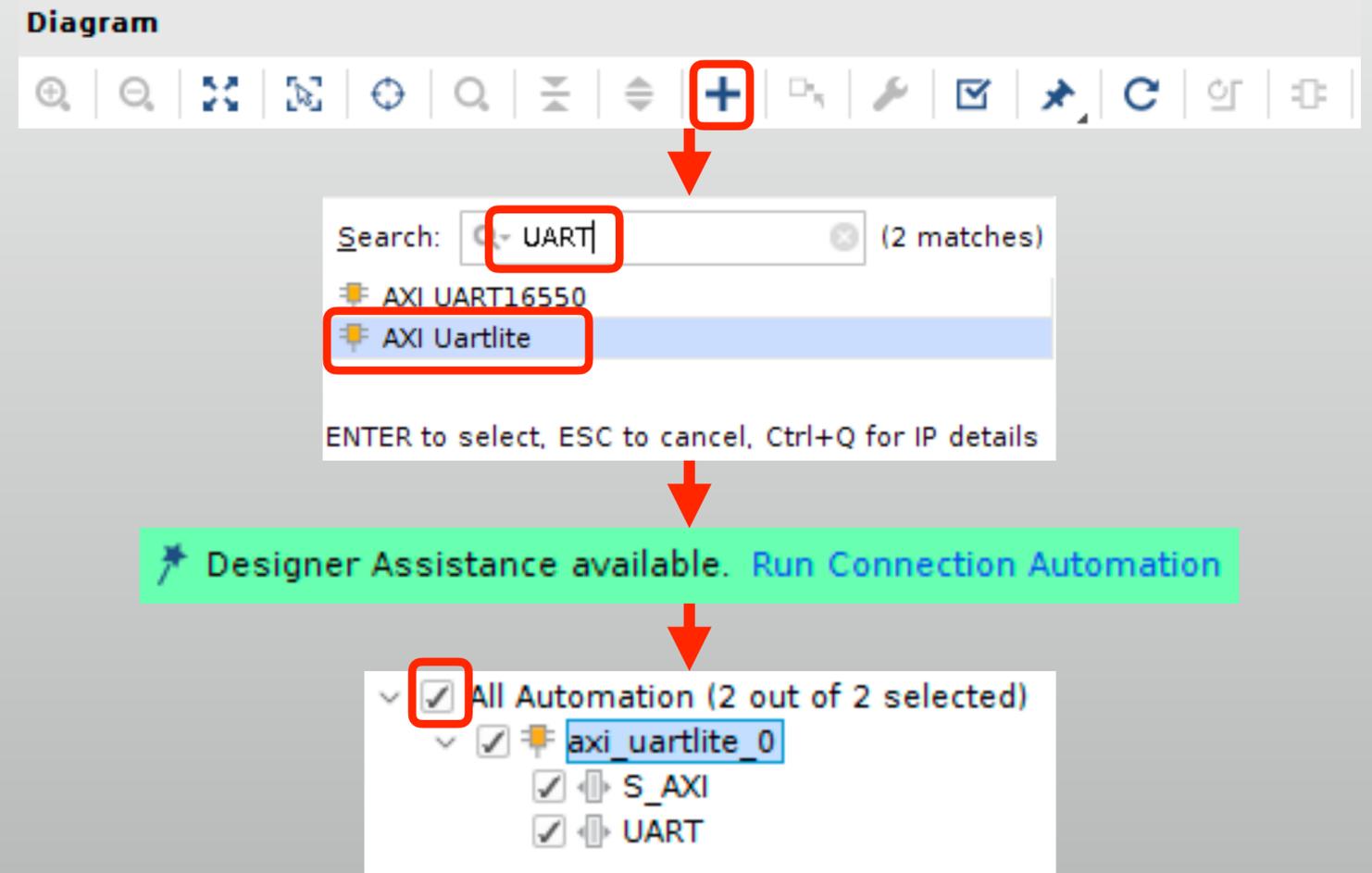
See inside RAM

- * Expand the local memory
 - * (LMB + BRAM controller) x2
 - * ILMB (Instruction LMB)
 - * DLMB (Data LMB)
- * Share the same dual-port BRAM



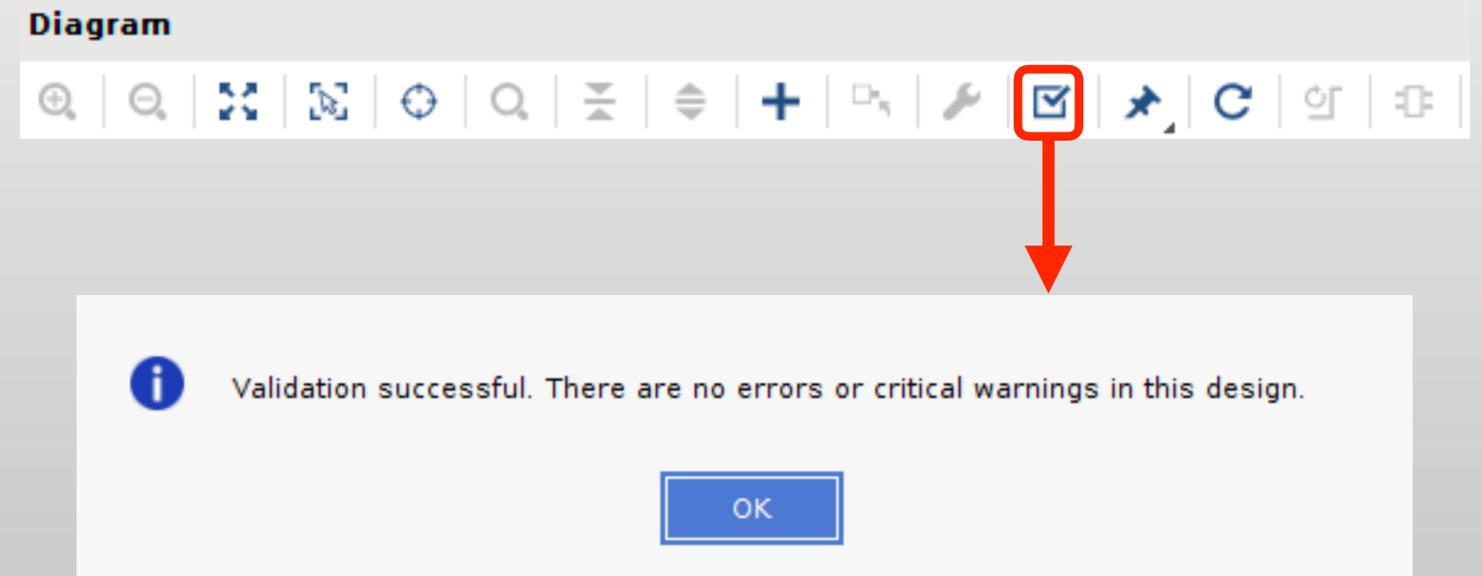
Add UART

- * Add “AXI Uartlite” core
- * Connect by automation
- * Check the uartlite_0 core

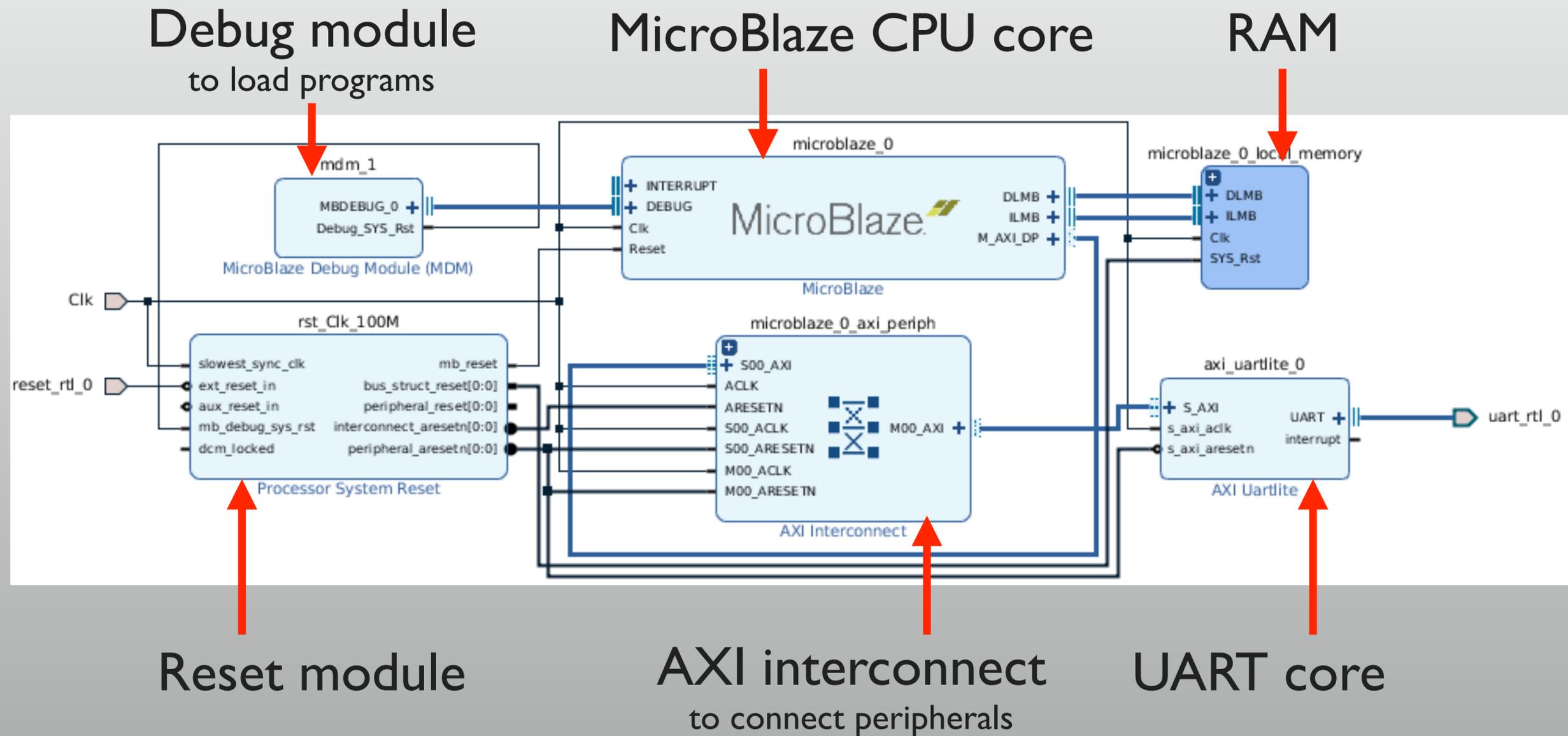


Validate the design

- * Let Vivado check design integrity

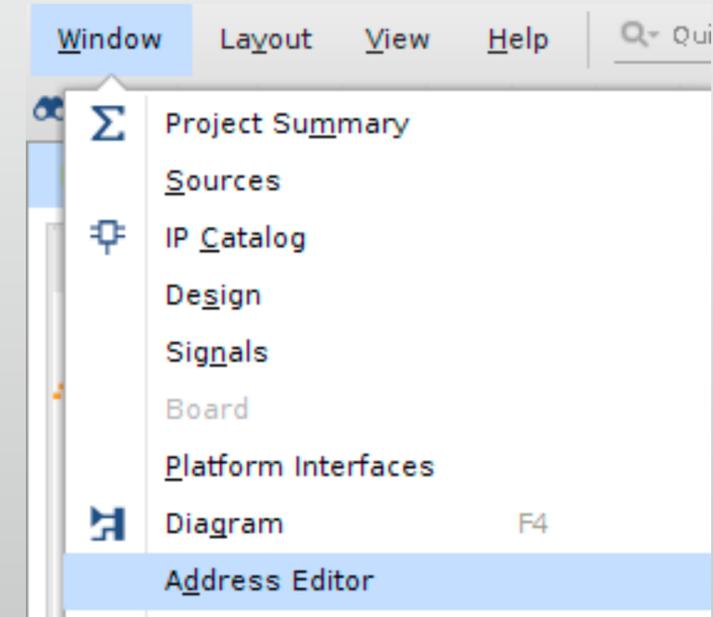


Your system is ready!



See the address space

- * Window → Address Editor
- * Address map is displayed
- * RAM at 0x0000_0000
- * UART at 0x4060_0000
- * Don't modify for now

A screenshot of the 'Address Editor' window. The window title is 'Address Editor'. It contains a table with the following columns: Cell, Slave Interface, Base Name, Offset Address, Range, and High Address. The table is expanded to show the address map for 'microblaze_0'.

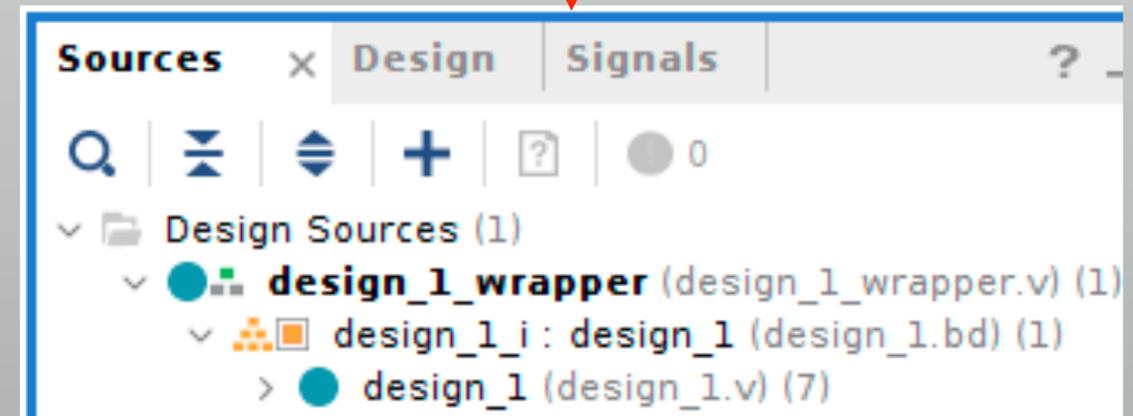
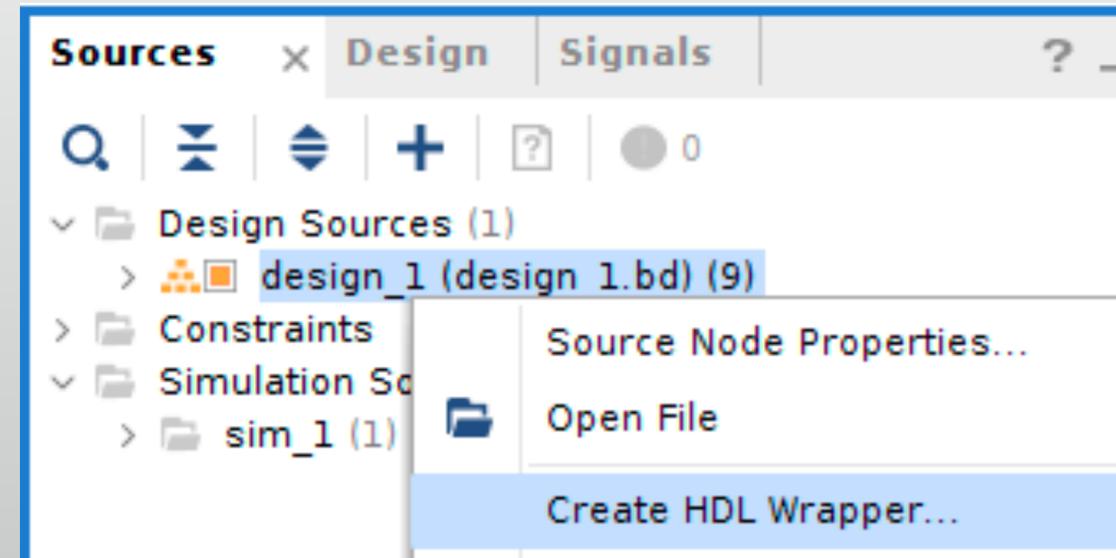
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF

Implementing the system

- * 2 problems to make the system work
 - * No top-level module yet
 - * No constraint file yet
- * Prepare them and just generate bitstream

Generate top module

- * Right-click on “design_1” and create HDL wrapper
- * Wrapper module for block design is generated
- * This will be the top module for today (or, may be your submodule)



Set the constraints

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports Clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports Clk]

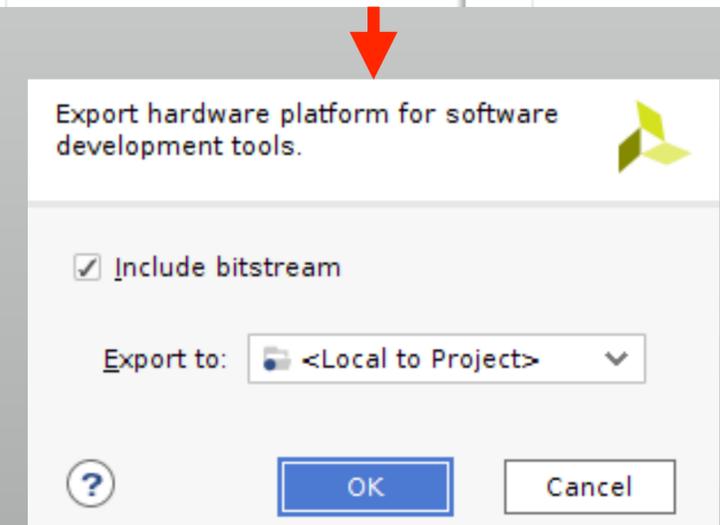
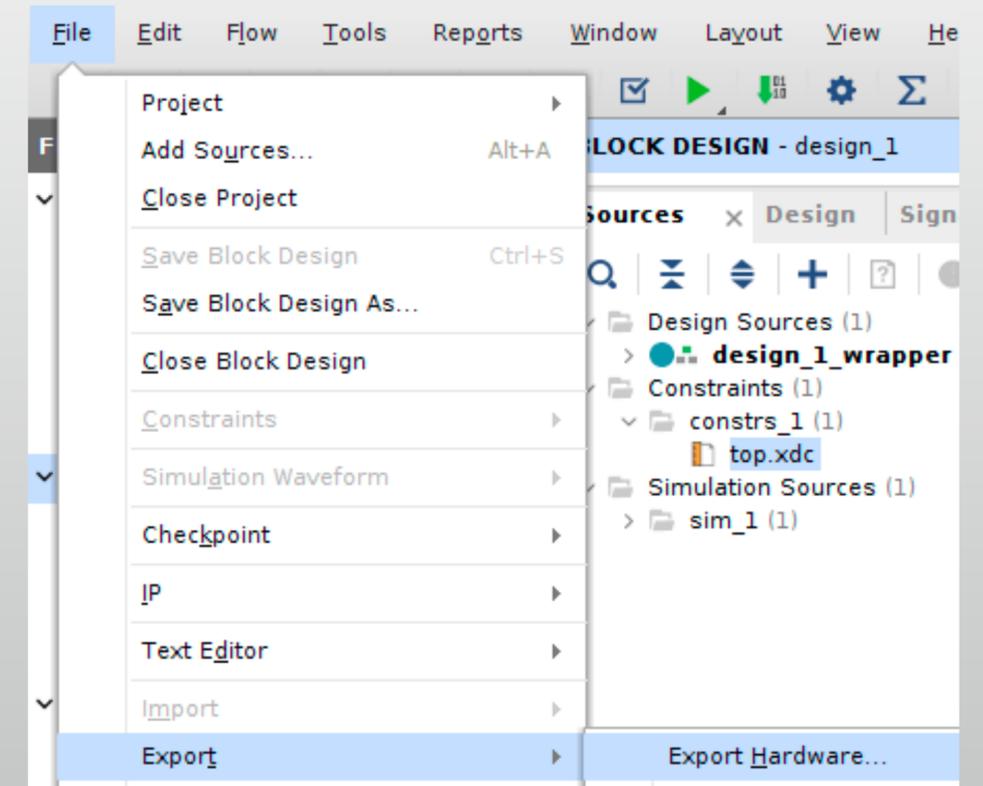
# Reset on Button C
set_property -dict { PACKAGE_PIN E16 IOSTANDARD LVCMOS33 } [get_ports reset_rtl_0]
set_property -dict { PACKAGE_PIN C4 IOSTANDARD LVCMOS33 } [get_ports uart_rtl_0_rxd]
set_property -dict { PACKAGE_PIN D4 IOSTANDARD LVCMOS33 } [get_ports uart_rtl_0_txd]
```

See the wrapper module

```
module design_1_wrapper
    (Clk,
     reset_rtl_0,
     uart_rtl_0_rxd,
     uart_rtl_0_txd);
    input Clk;
    input reset_rtl_0;
    input uart_rtl_0_rxd;
    output uart_rtl_0_txd;
```

Get ready for SDK (Software Development Kit)

- * File → Export → Export Hardware
- * “Include bitstream” is required
- * Address map + bitstream is handed to SDK
- * Then, File → Launch SDK to launch SDK



Hardware definitions in SDK

- * HDF (hardware definitions) is automatically loaded
- * Updated by “Export hardware” in Vivado
- * Update required when bitstream or the CPU’s address map had changed

The screenshot shows a window titled 'system.hdf' with a subtitle 'design_1_wrapper_hw_platform_0 Hardware Platform Specification'. Below the title bar, there is a 'View Menu' button. The main content is divided into two sections: 'Design Information' and 'Address Map for processor microblaze_0'. The 'Design Information' section lists: Target FPGA Device: 7a100t, Part: xc7a100tcsg324-1, Created With: Vivado 2018.2, and Created On: Tue Nov 13 09:13:26 2018. The 'Address Map for processor microblaze_0' section contains a table with the following data:

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
axi_uartlite_0	0x40600000	0x4060ffff	S_AXI	REGISTER
microblaze_0_local_memory_0	0x00000000	0x0000ffff	SLMB	MEMORY

Create Application Project

* File → New →
Application Project

* Name project

* Choose template
“Hello World”

Application Project
Create a managed make application project.

Project name:

Use default location

Location:

Choose file system:

OS Platform:

Target Hardware

Hardware Platform:

Processor:

Target Software

Language: C C++

Compiler:

Hypervisor Guest:

Board Support Package: Create New Use existing

Templates
Create one of the available templates to generate a fully-functioning application project.

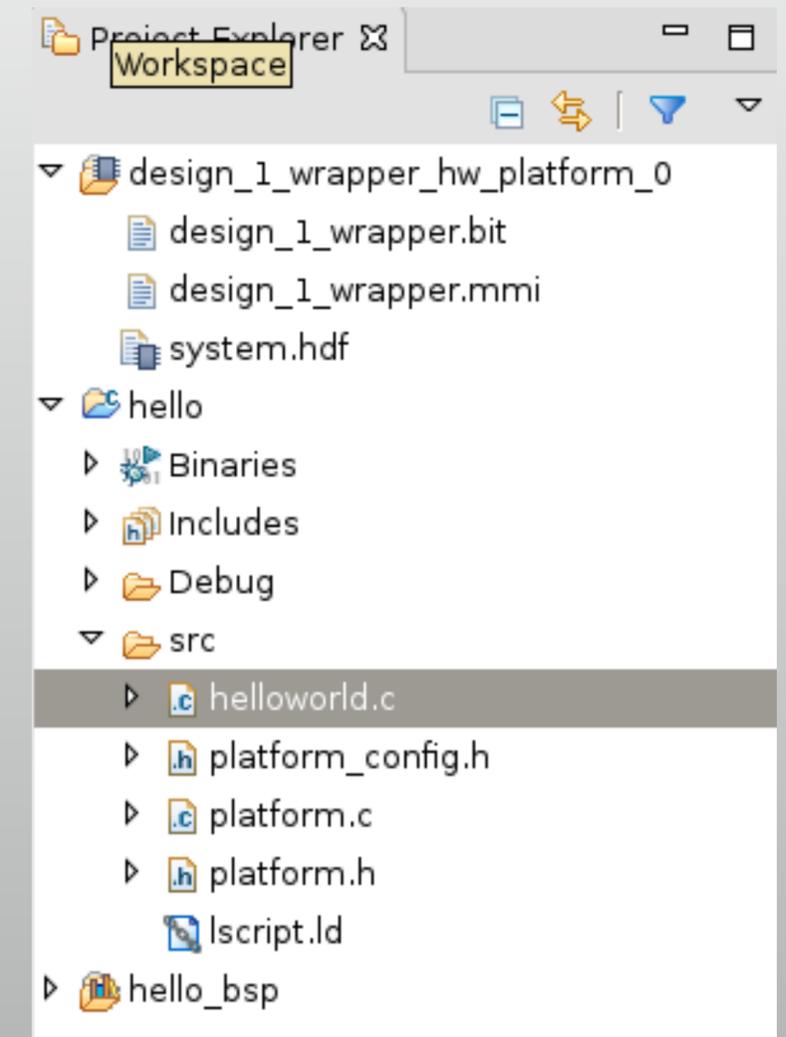
Available Templates:

- Dhrystone
- Empty Application
- Hello World
- lwIP Echo Server
- lwIP TCP Perf Client
- lwIP TCP Perf Server
- lwIP UDP Perf Client
- lwIP UDP Perf Server
- Memory Tests
- Peripheral Tests
- SREC Bootloader
- SREC SPI Bootloader

Let's say 'Hello World' in C.

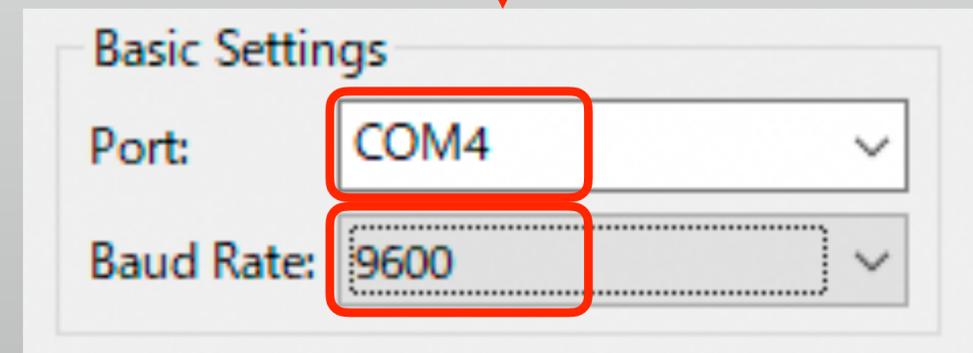
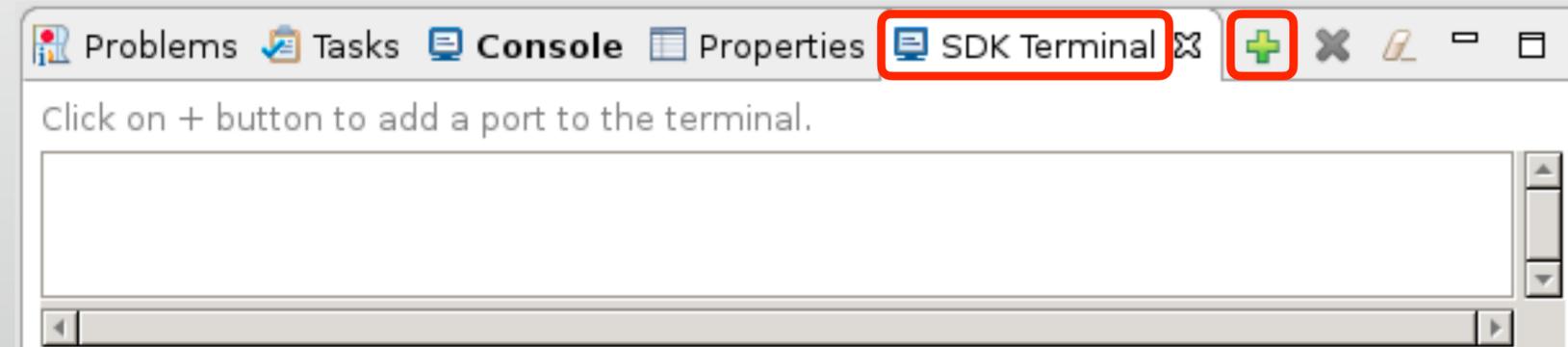
Application + BSP generated

- * Application Project: hello
 - * “helloworld.c” is the application code
 - * “platform.*” is hardware support code (don't modify)
- * Board Support Package: hello_bsp
 - * Minimum libraries as device drivers



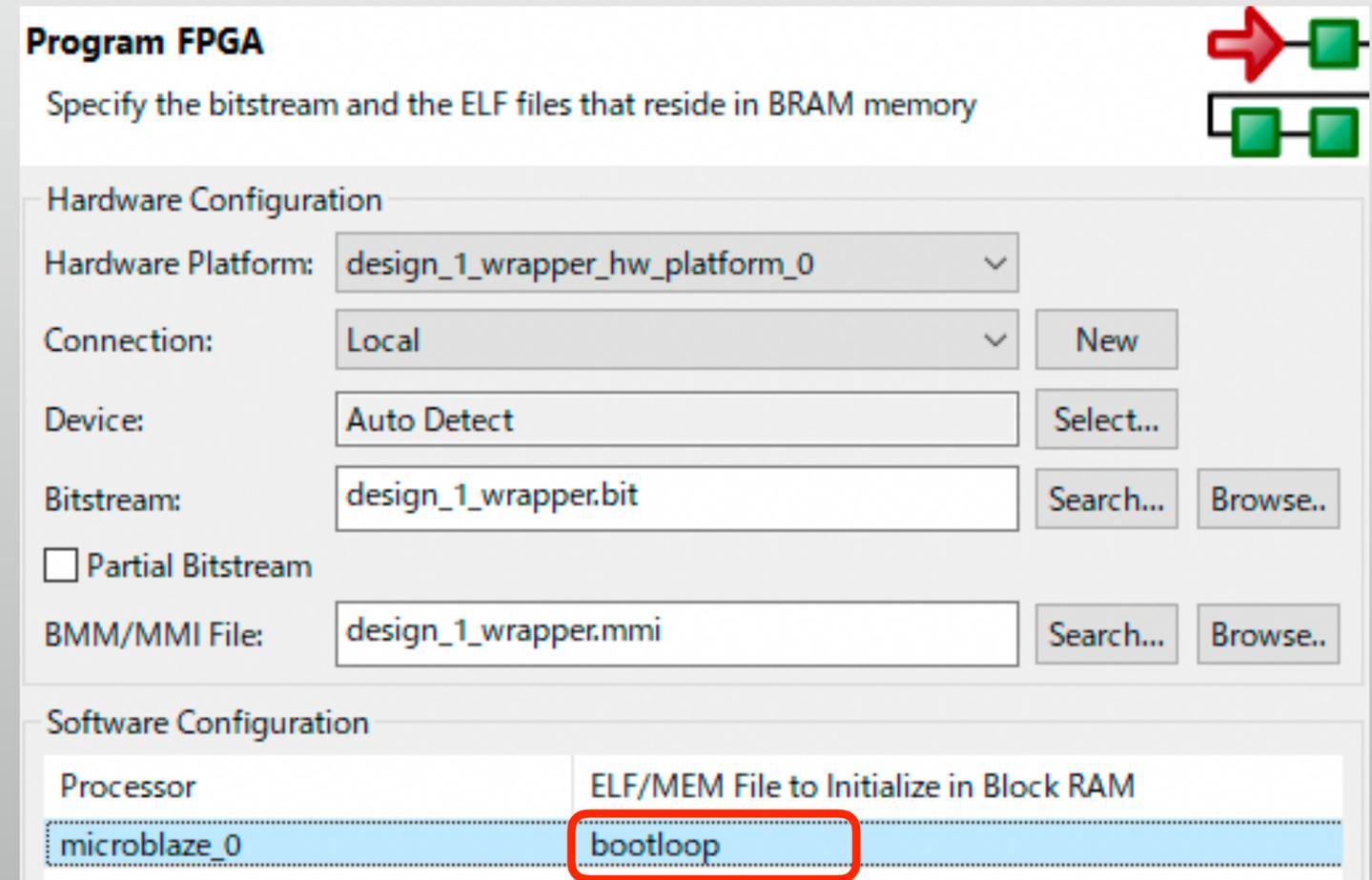
Open serial console

- * Communicate with UART on FPGA
- * SDK terminal → +
- * Choose serial device (COMx)
- * Baud rate is 9600
(settings in block design → UARTLite core)



Program FPGA

- * Xilinx → Program FPGA
 - * Change “ELF/MEM file” from “bootloop” to “hello.elf”
 - * Then click “Program” to launch
 - * FPGA is programmed with the bitstream + hello.elf
 - * Check SDK terminal



hello.elf (found in dropdown menu)

Modify program and check size

- * Modify program, Ctrl+B to build
- * SDK terminal can send text
- * Multiple “Hello world” appears
- * Check the code size
- * 24,560 bytes used in 64kB RAM

src/helloworld.c:

```
while(1){
    print("Hello World\n\r");
    getchar();
}
```

Debug/hello.elf.size:

text	data	bss	dec	hex	filename
20076	1308	3176	24560	5ff0	hello.elf

Code size in embedded programming

- * Code + variable size must not exceed the memory size (64kB for this time)
 - * Standard library functions (such as printf() and scanf()) is usually too large
- * See Xilinx Standalone Library Documentation:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/oslib_rm.pdf
 - * Document is very long, but first “Xilinx Standard C libraries” is sufficient
 - * printf() is larger than 64kB!
 - * Instead, print(), printnum() and xil_printf() is provided

Next week:

- * Integrating HDL peripherals with MicroBlaze processor
 - * + more AXI peripherals