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

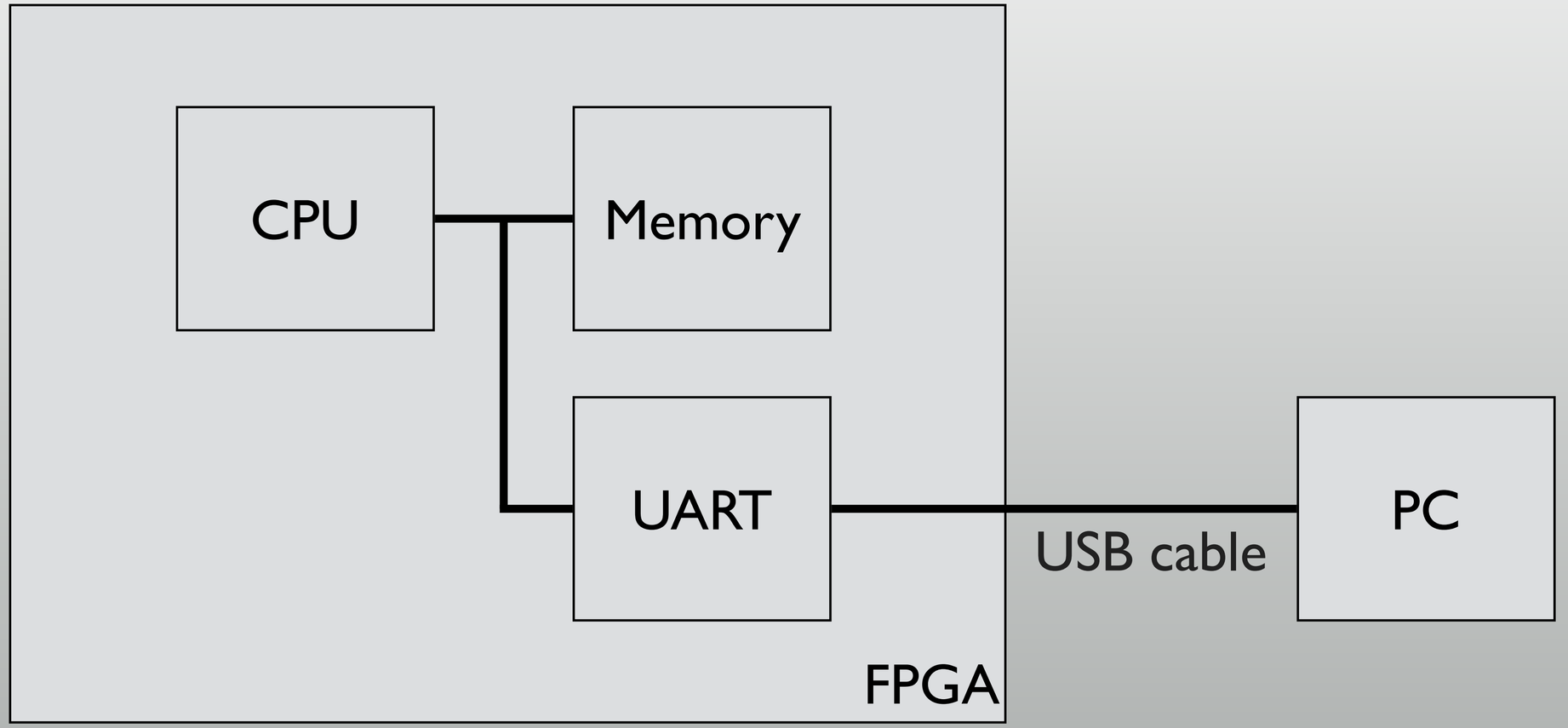
Processor-based systems

- 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 ←→ 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

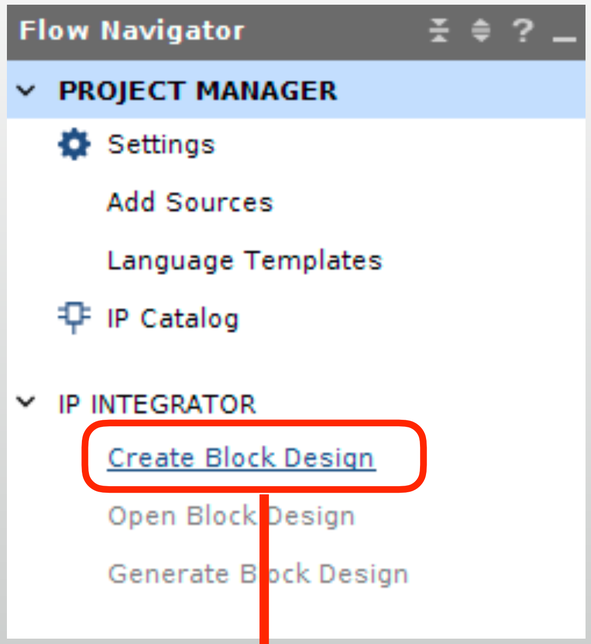


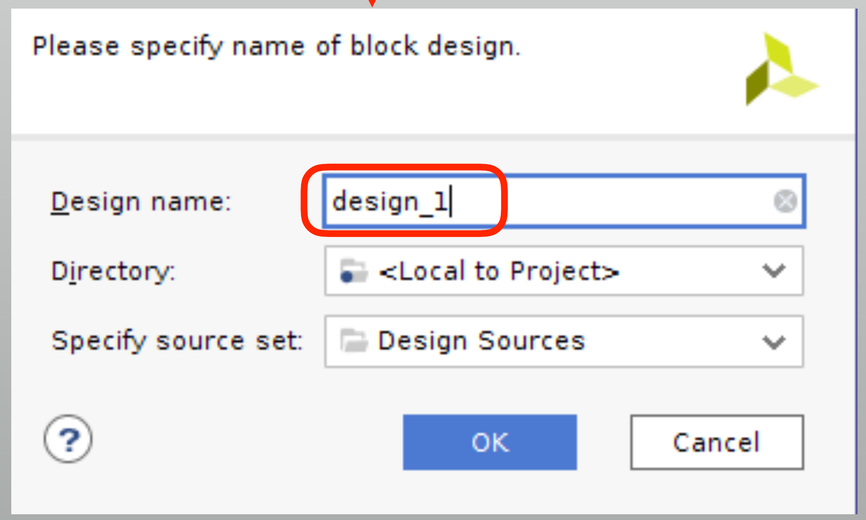
Hands-on: “Hello World” from FPGA

- 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

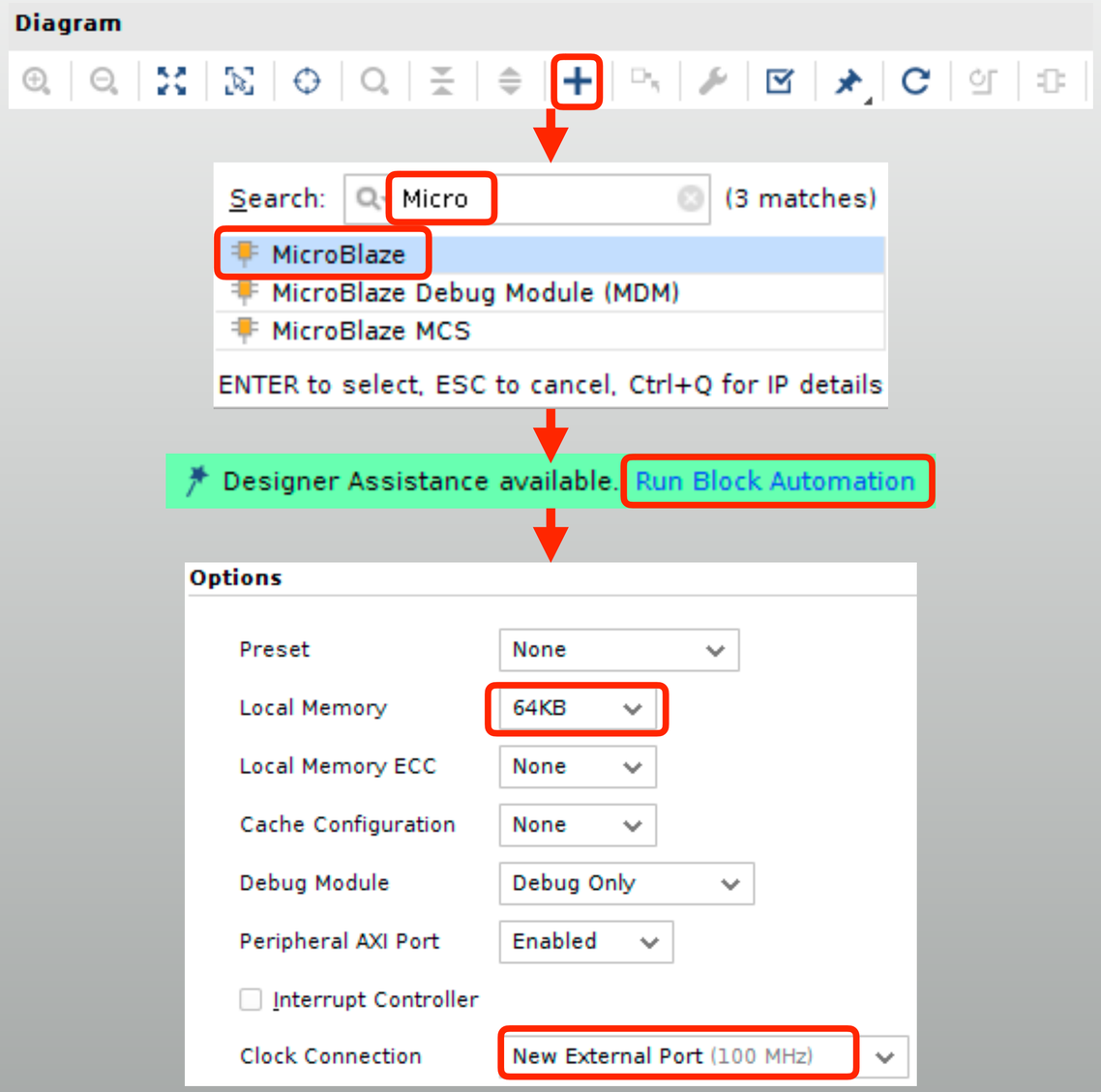
- Create a Vivado RTL project
  - Device: xc7a325tcsg-1
  - “Create Block Design” → “design_1”
    - Empty block diagram is open





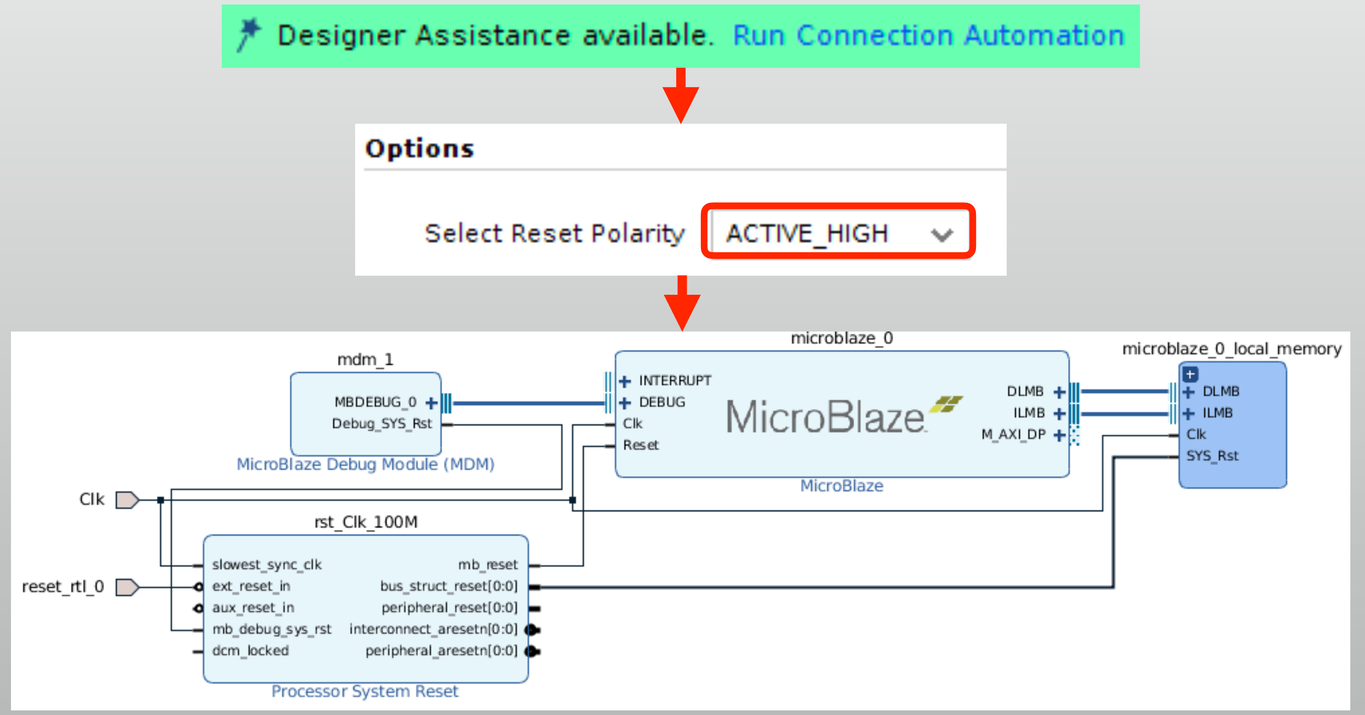
Add processor core

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



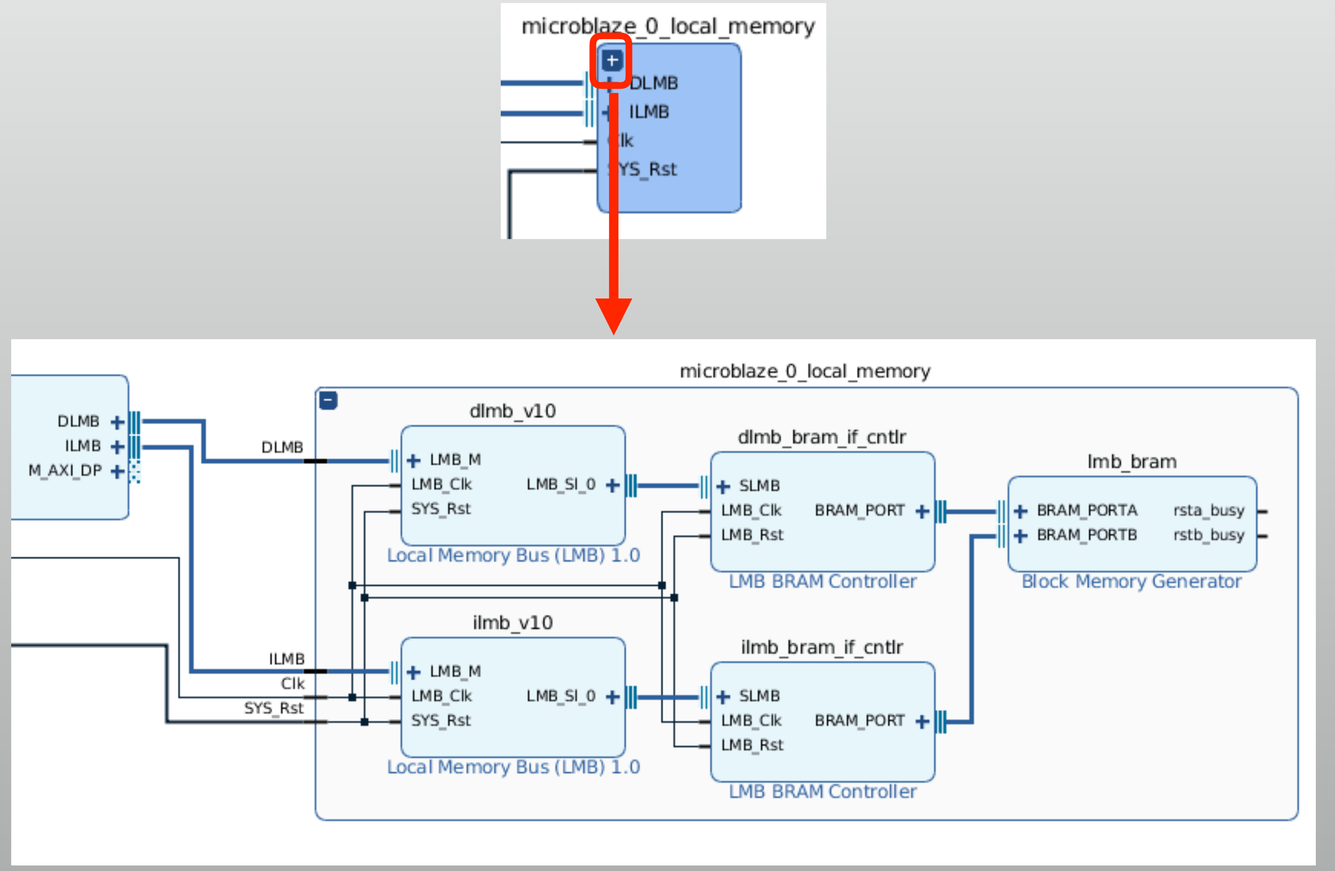
Reset settings

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



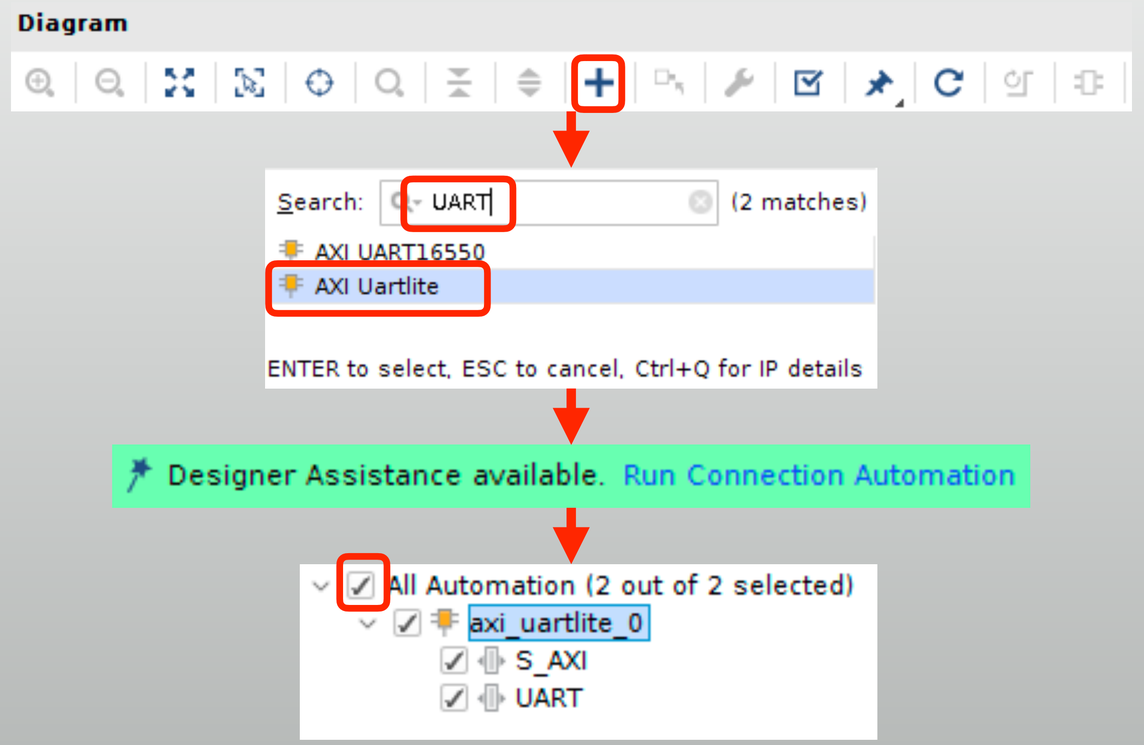
See inside RAM

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



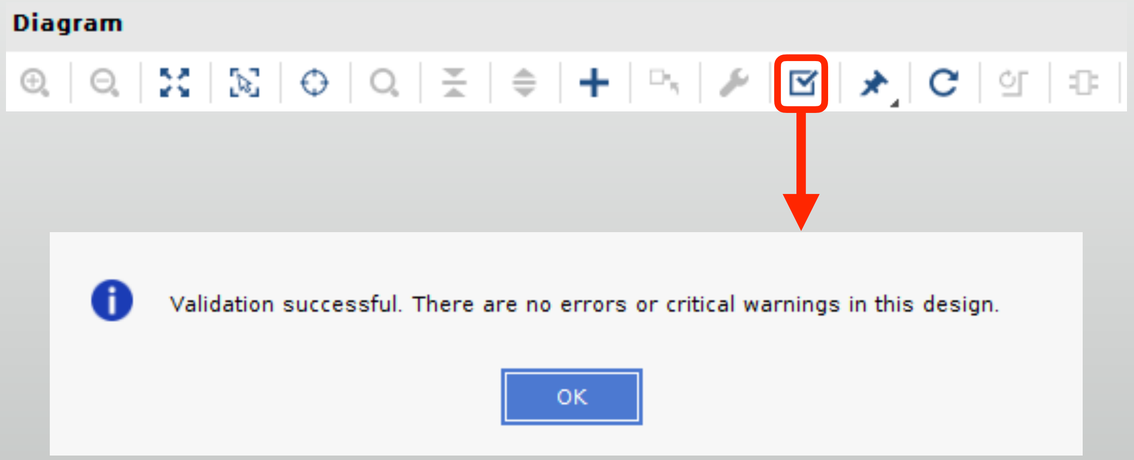
Add UART

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



Validate the design

- Let Vivado check design integrity

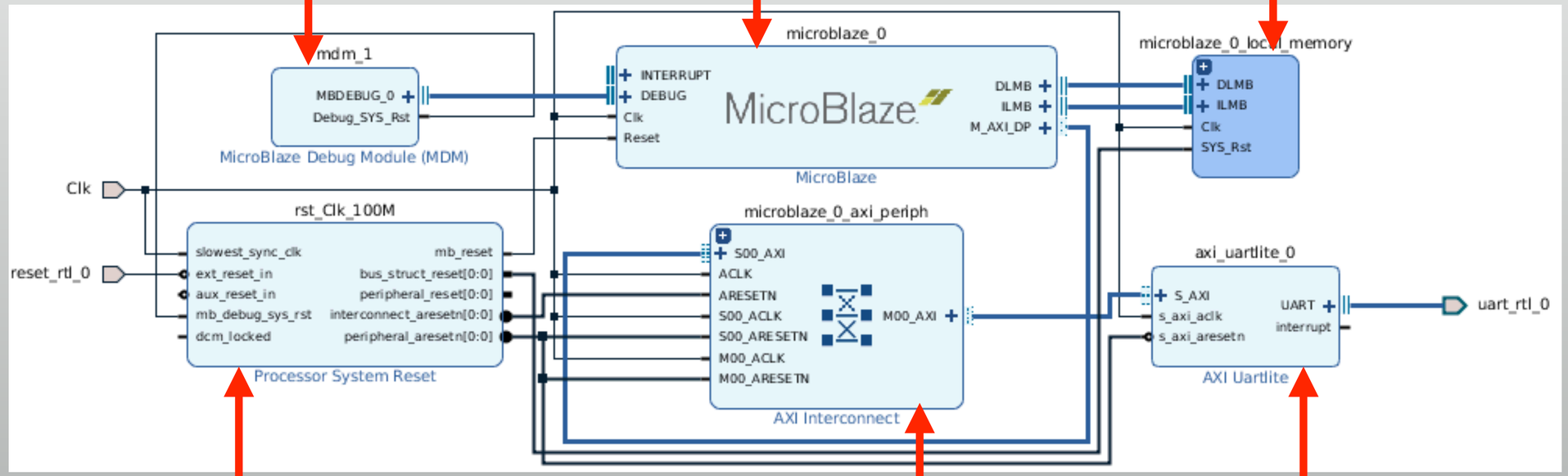


Your system is ready!

Debug module
to load programs

MicroBlaze CPU core

RAM



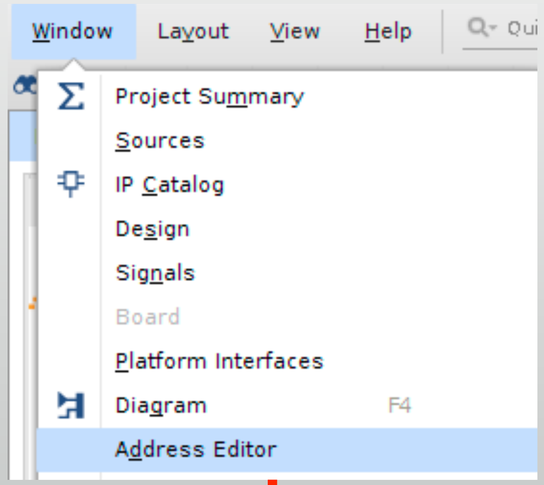
Reset module

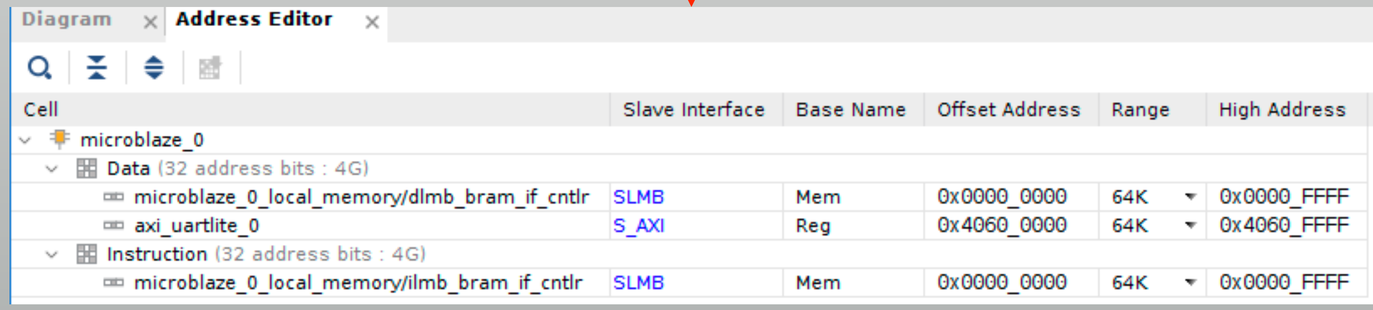
AXI interconnect
to connect peripherals

UART core

See the address space

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



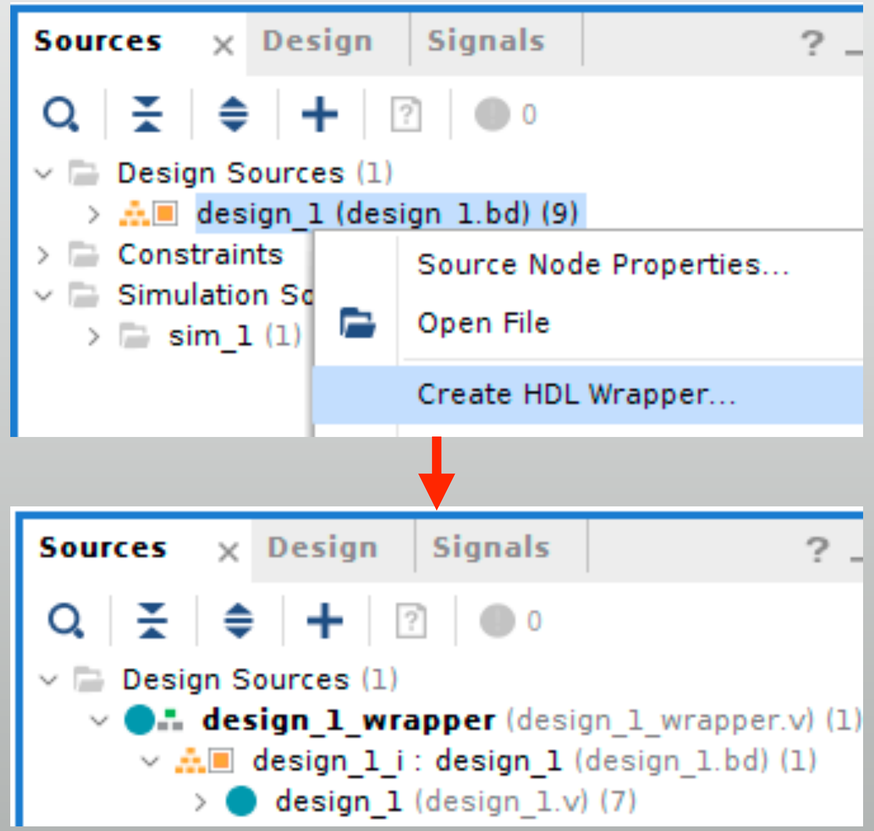


Implementing the system

- 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

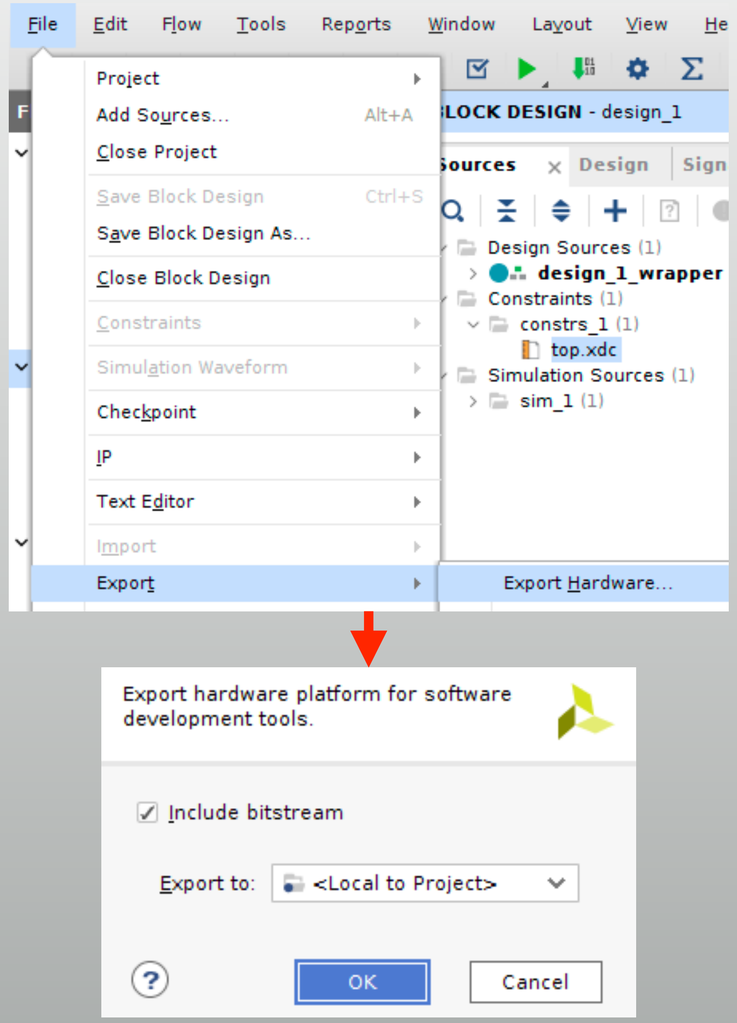
```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports Clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports Clk]

# 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;
```

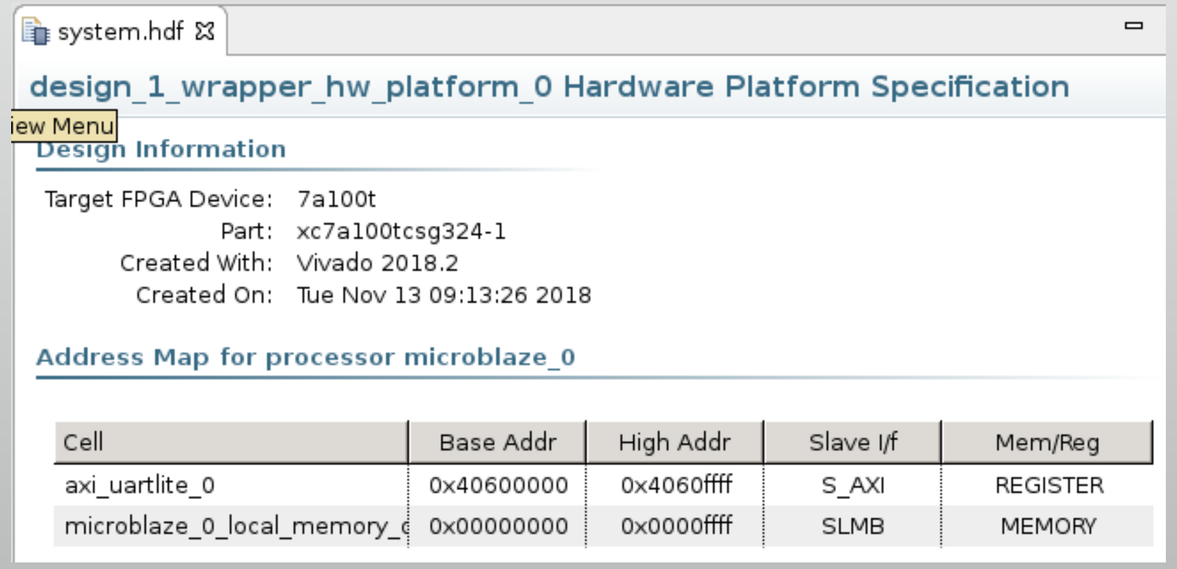
Get ready for SDK (Software Development Kit)

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



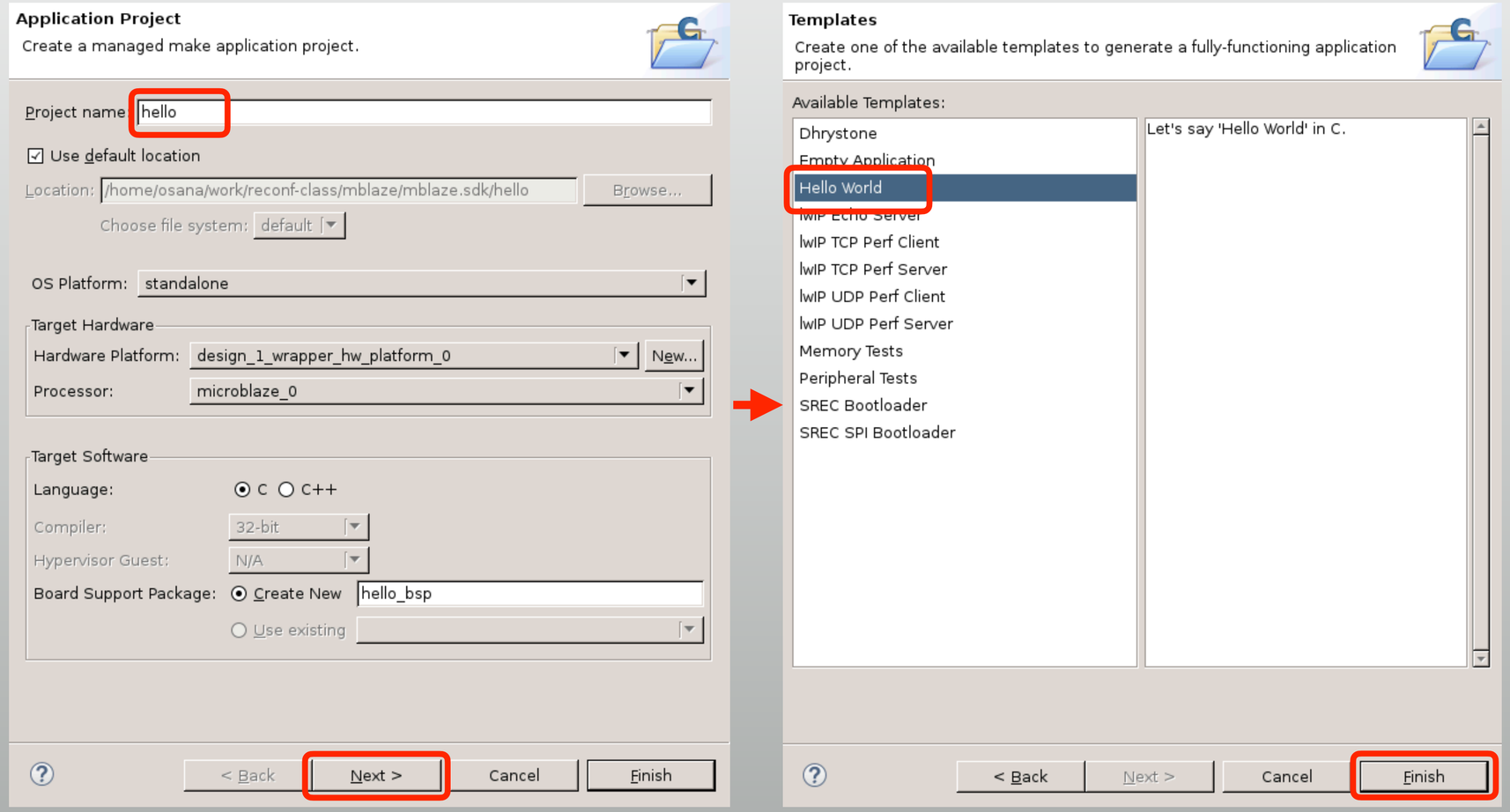
Hardware definitions in SDK

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



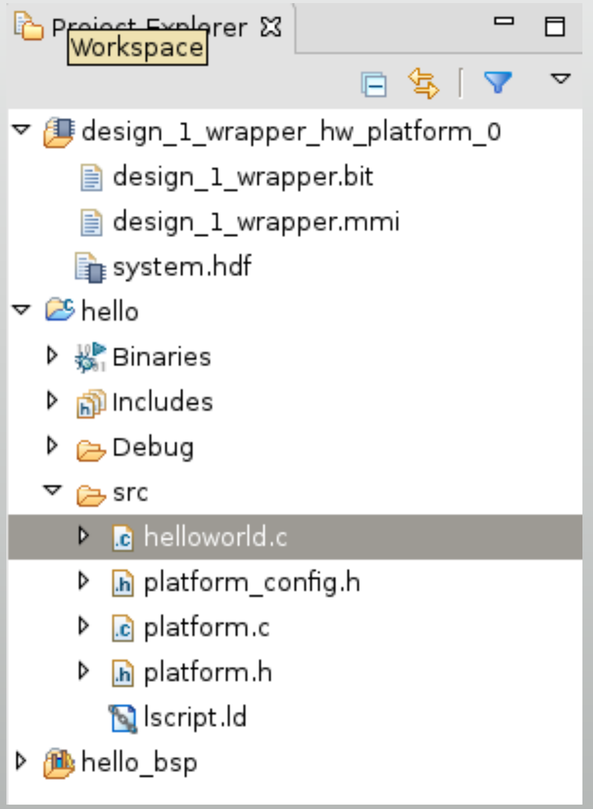
Create Application Project

- File → New → Application Project
  - Name project
  - Choose template "Hello World"



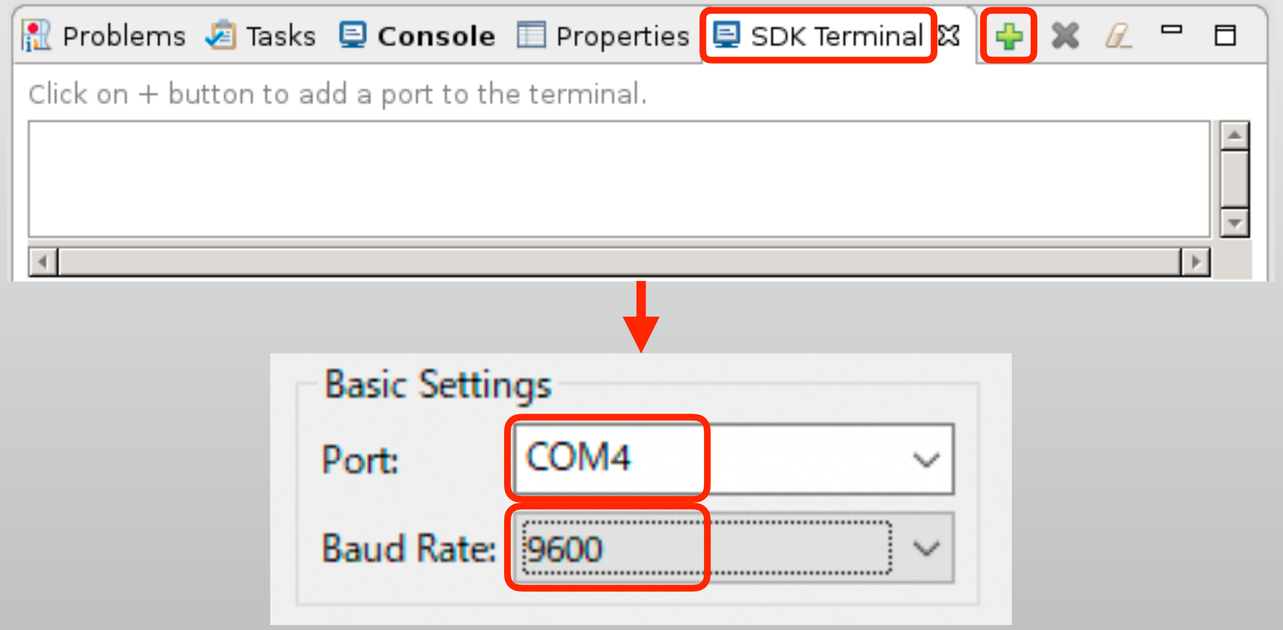
Application + BSP generated

- 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



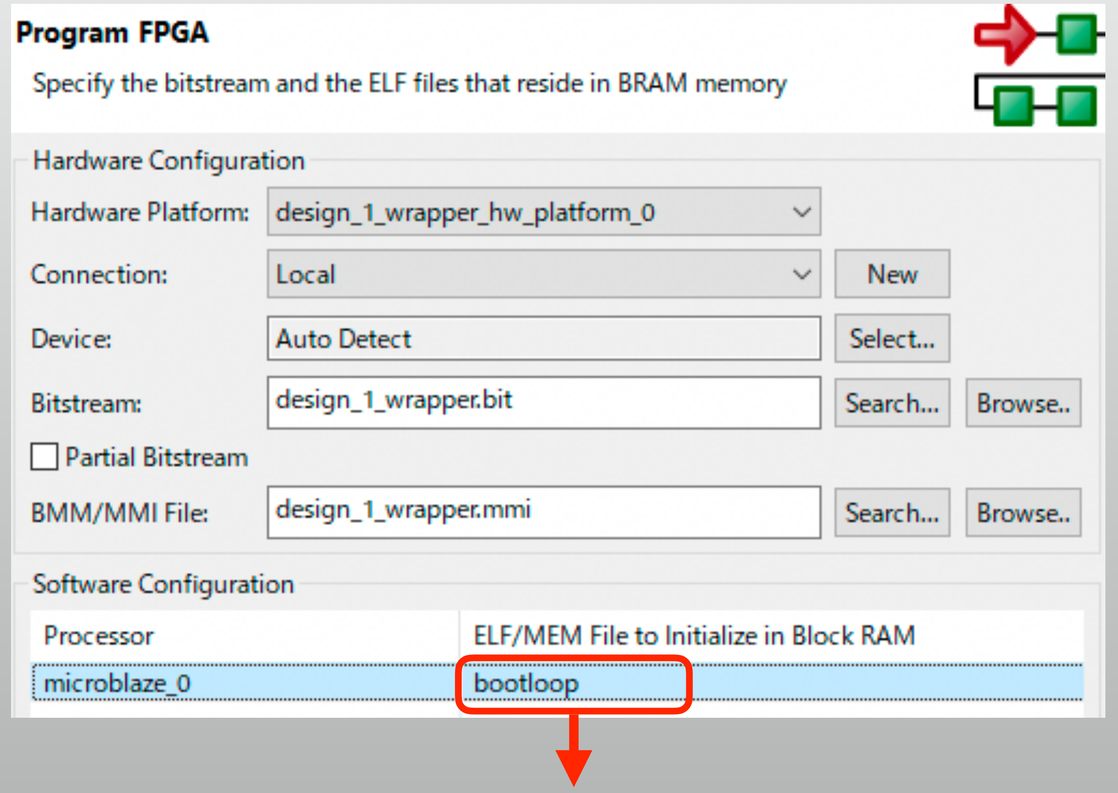
Open serial console

- 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