EEE Department

AY23-24 RVFpga Projects

Update Meeting 11th June 2024

Exams Done! - Full time on the Project through September

Execution starts now…….

How are you getting on with the Labs

- Getting Started?
- Progress through the Labs?

- You will find it difficult to figure out how to INTEGRATE until you understand how the platform works
- You will have to control and interact with your specific block/function from the processor.

- … BUT remember this is only prep - you have to.
  - Design a Unique block – *including* figuring out how to integrate

3

Have you agreed a final *scope* for your block?

- Specific I/O e.g I2C
- Specific Accelerator – e.g AES
- Extension to the SoC Platform Core functionality

THIS MEANS YOU WILL ALSO HAVE TO RESEARCH ITS FUNCTIONALITY AND PROPOSE AN MICROARCHITECTURE FOR THE BLOCK

- In addition to integrating the block you will need to **architect** it
  - What specific functionality will be offloaded into hardware
    - Will some of it be carried out in software on the processor?
  - How will the block be controlled
  - How will it return status
  - Where will data come from or go to in the system

THIS MEANS YOU WILL ALSO HAVE TO RESEARCH ITS FUNCTIONALITY AND PROPOSE AN MICROARCHITECTURE FOR THE BLOCK

- You will have to write NEW RTL for you specific block
- You will have to write RTL for the control/status/data interface
- You will have to *test* it to see if it works and demonstrate functionality

3

Do you have a plan?

- Logbook?GANT or other planning chart?
- Risks?
- Requirements and Specification Agreed & Documented?

- Microarchitecture for block
- (Integration) architecture.

- Starting to think about testing/demonstration strategy
  - What will you test at Unit level
  - What will you test in an integration level simulation
  - What will you test/demonstrate on the FPGA

  - How will you write up the RESULTS of your project

Hint – you can demonstrate functionality. Record aspects of its performance. Describes how many resource (area) it consumes

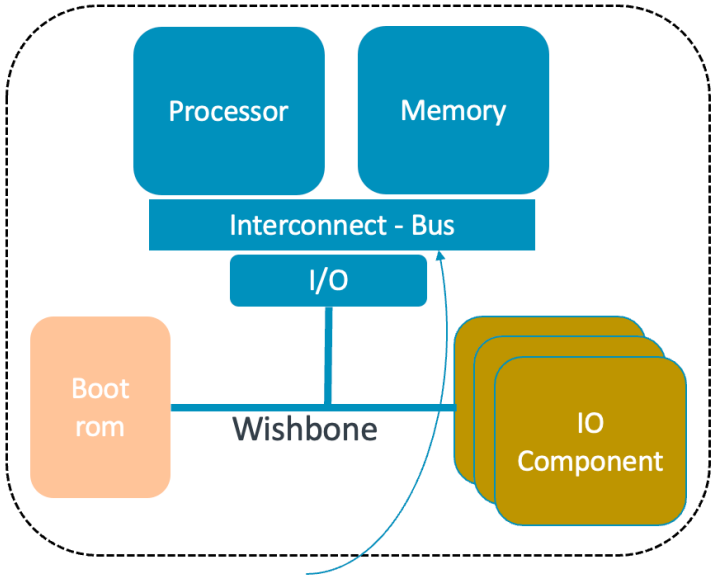
SoC 101

System on Chip

- SoC Architecture
  - Processor Memory
  - I/O
  - System Support
- Memory Map
- Bus Transaction

System on Chip (SoC) Projects

Your project will be *extending* an SoC example - RVfpga



https://github.com/pulp-platform/axi/blob/master/scripts/axi_intercon_gen.py

RV-FPGA is an **SoC Architecture**

It is a complete set of RTL (Verilog HDL) which describes the structure and behavior of an example SoC

It *instantiates* a SWERV RISC-V Processor Core

This is a micro-architectural implementation of the RISC-V Instruction Set Architecture (ISA)

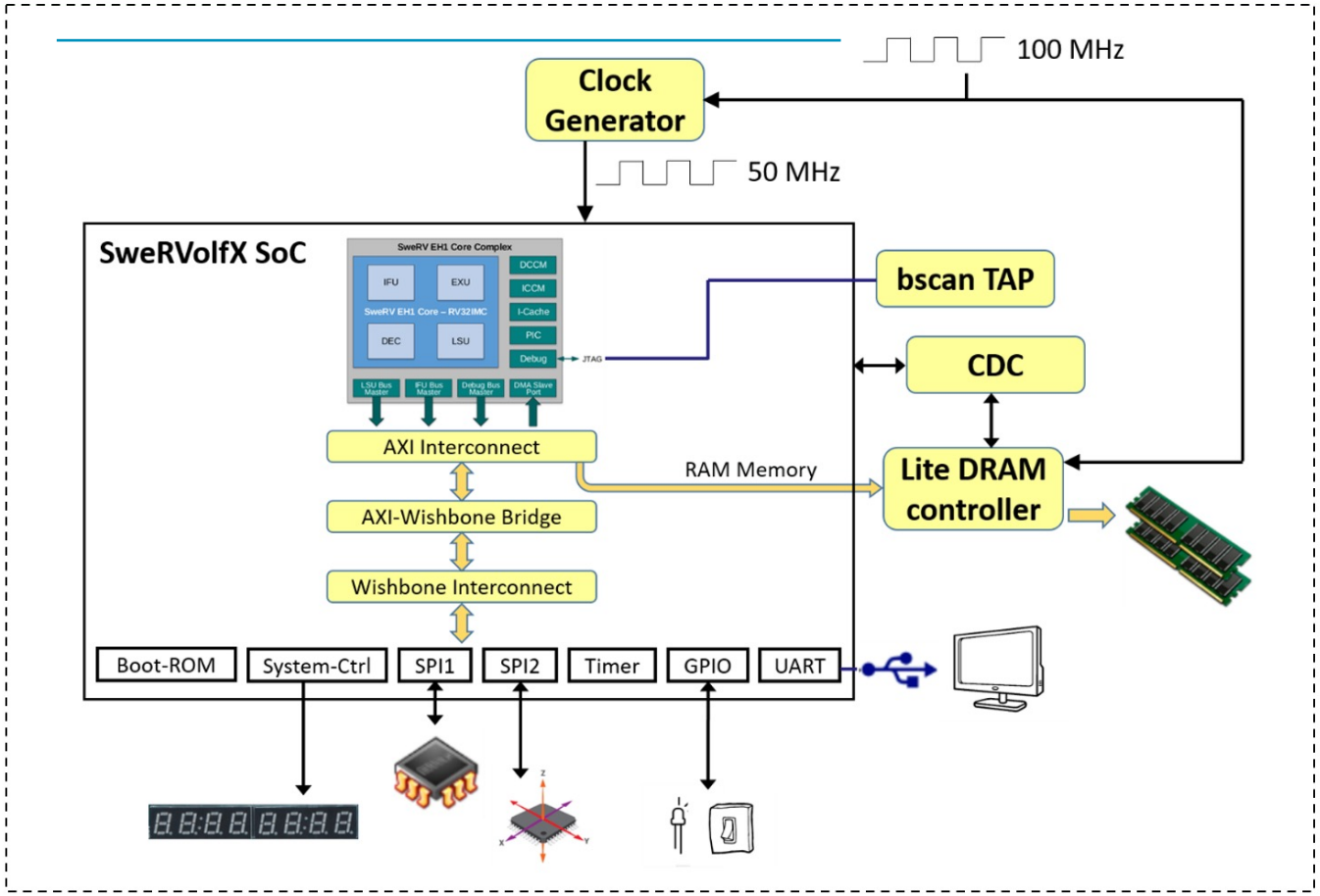
The example (which you will extend) will run code that is compiled and loaded into its memory

The design can be simulated as a Verilog HDL model

Or it can be **emulated** on an FPGA boards

Top Level of the FPGA Hierarchy

RVfpgaNexys



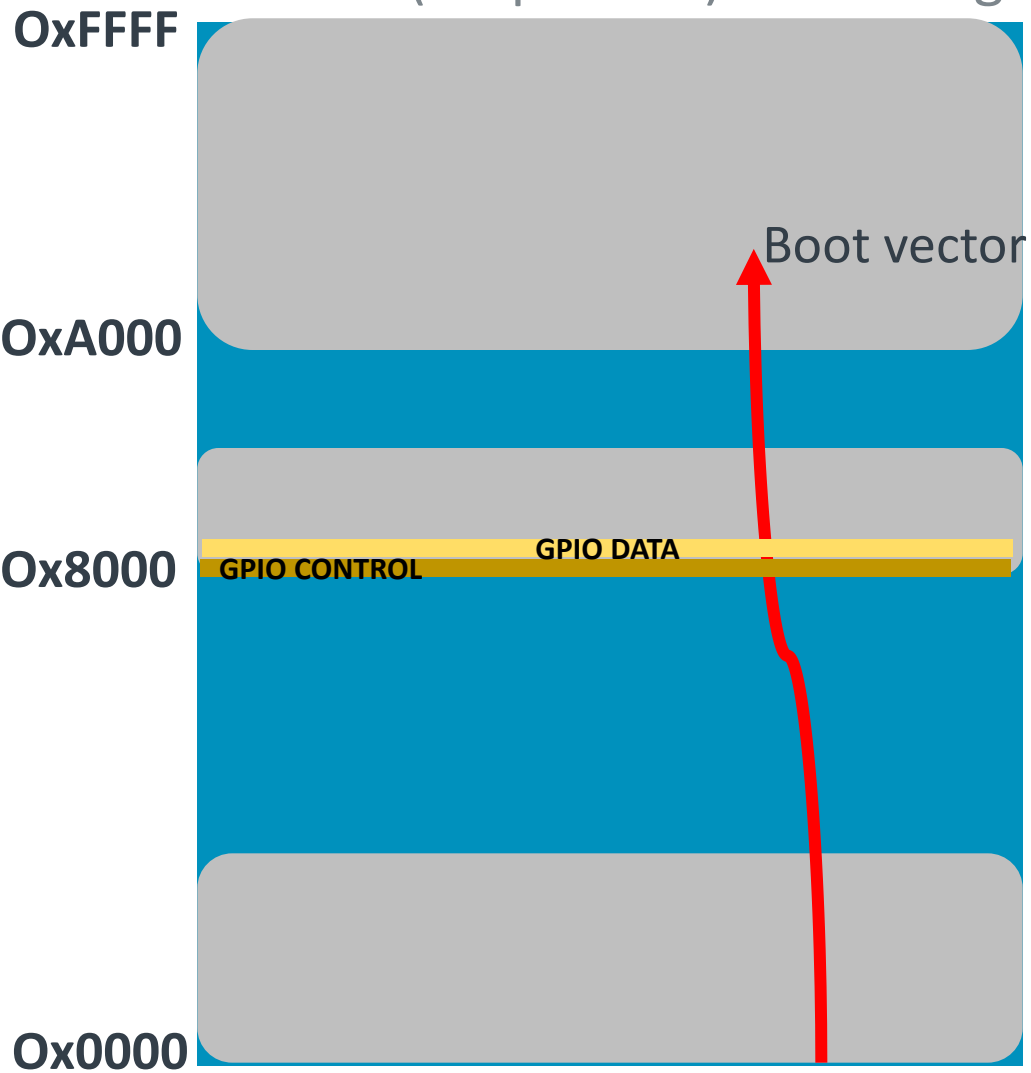
- **RVfpgaNexys:** SweRVolfX SoC targeted to Nexys A7 FPGA board with added peripherals:
  - **Core & System:**
    - SweRVolfX SoC
    - Lite DRAM controller
    - Clock Generator, Clock Domain

    and BSCAN logic for the JTAG port
  - **Peripherals** used on Nexys A7 FPGA board:
    - DDR2 memory
    - UART via USB connection
    - SPI Flash memory
    - 16 LEDs and 16 switches
    - SPI Accelerometer
    - 8-digit 7-segment displays

Memory Map

You will need to add new registers to the memory map specific to your block

How words (bit pattern) are arranged - Links hardware specific functions to software



External Memory

IO Region

Mem map can be non contiguous

Boot Memory

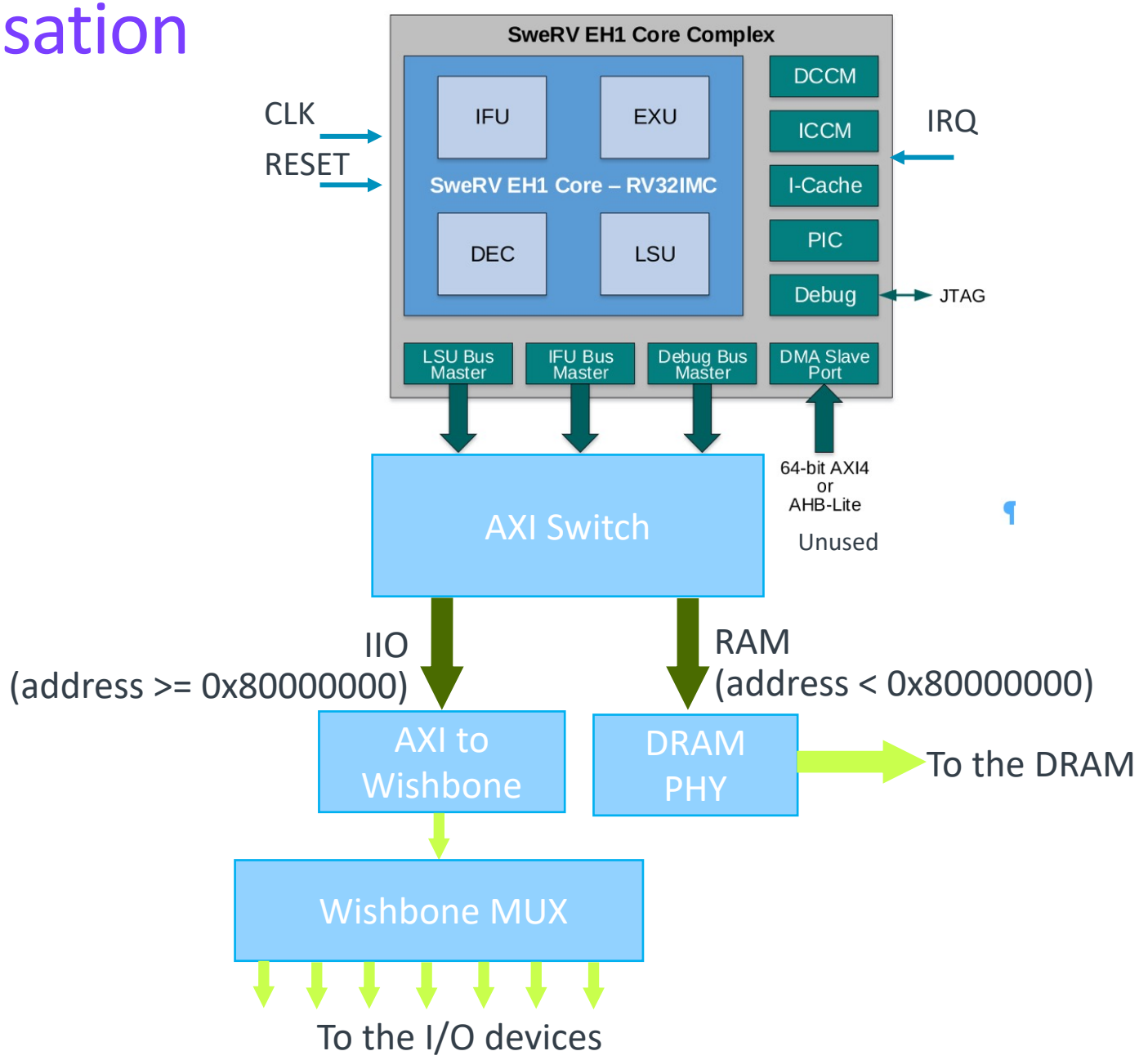
- Allocation of Hardware Resource to Unique location for a load or store
- The **Address Decoder** selects ONE word (*matches address*)
- Processor Loads or Stores a word to a ONE Register or Location in Memory
- Processor reads (program) instructions from memory
- Processor Reads it first instruction from bottom of boot memory

AXI Bus

- The AXI bus is an ARM standard for a high-performance, multi-master spilt-transaction bus
- It consists of separate ‘channels’ to transfer information:
  - Read Address
  - Read Data
  - Write Address
  - Write Data
  - Write Response

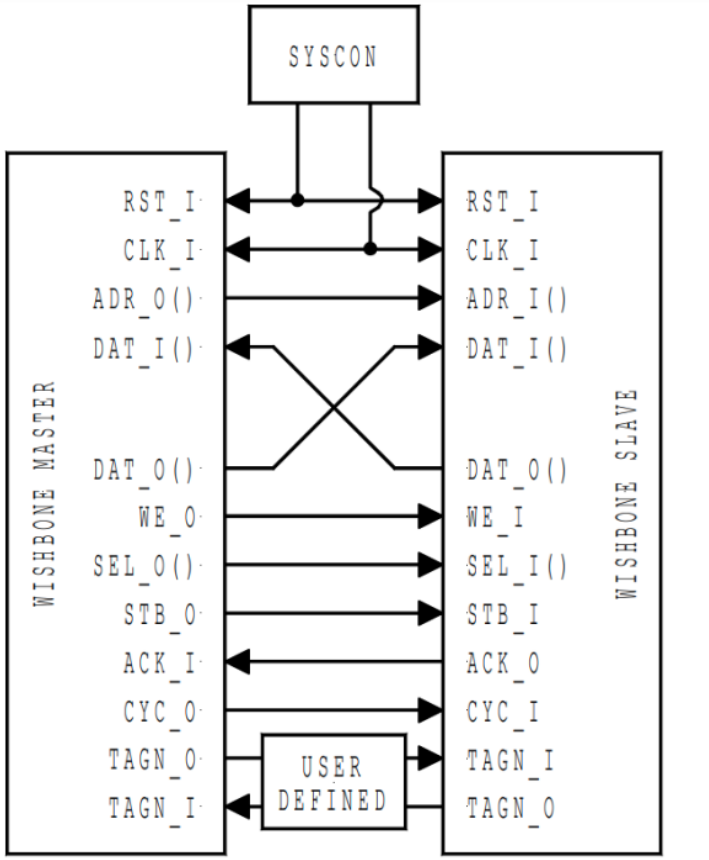
Signal	Source: master/ slave	Input/ Output	Description
Aclk	Global	Input	Global clock signal.
AResetn	Global	Input	Global reset signal
AWID[3:0]	Master	Input	Write address ID.
AWADDR[31:0]	Master	Input	Write address.
AWLEN[3:0]	Master	Input	Write burst length.
AWSIZE[2:0]	Master	Input	Write burst size.
AWBURST[1:0]	Master	Input	Write burst type.
AWLOCK[1:0]	Master	Input	Write lock type.
AWCACHE[3:0]	Master	Input	Write cache type.
AWPROT[2:0]	Master	Input	Write protection type.
WDATA[31:0]	Master	Input	Write data.
ARID[3:0]	Master	Input	Read address ID.
ARADDR[31:0]	Master	Input	Read address.
ARLEN[3:0]	Master	Input	Read Burst length.
ARSIZE[2:0]	Master	Input	Read Burst size.
ARLOCK[1:0]	Master	Input	Read Lock type.
ARCACHE[3:0]	Master	Input	Read Cache type.
ARPROT[2:0]	Master	Input	Read Protection type.
RDATA[31:0]	Master	Input	Read data.
WLAST	Master	Input	Write last.
RLAST	Slave	Output	Read last.
AWVALID	Master	Output	Write address valid.
AWREADY	Slave	Output	Write address ready.
WVALID	Master	Output	Write valid.
RAVLID	Slave	Output	Read valid.
WREADY	Slave	Output	Write ready.
BID[3:0]	Slave	Output	Write Response ID.
RID[3:0]	Slave	Output	Read response ID.
BRESP[1:0]	Slave	Output	Write response.
RRESP[1:0]	Slave	Output	Read response.
BVALID	Slave	Output	Write response valid.

Bus Organisation

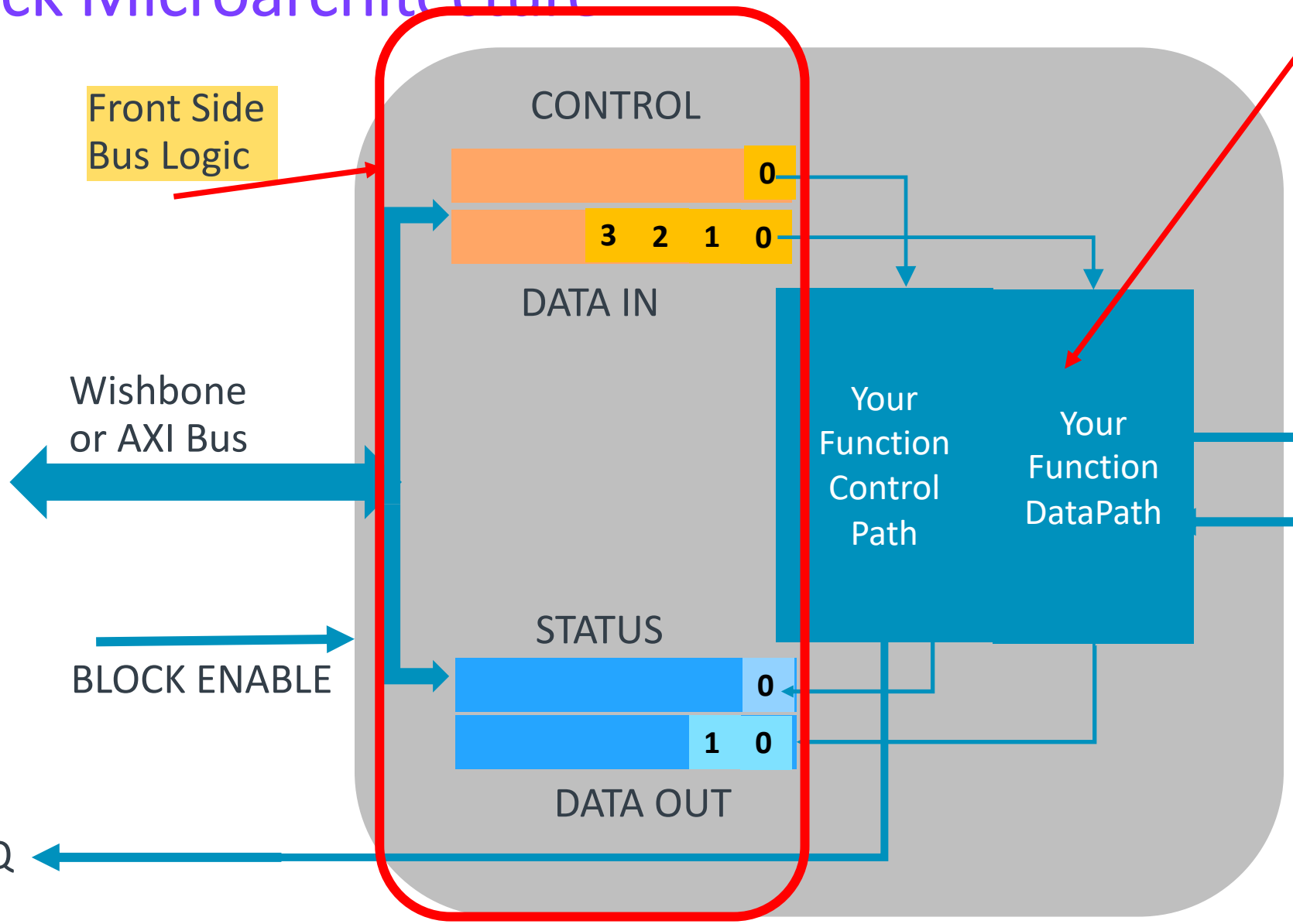


Wishbone Interface

- The Wishbone interface is a much simpler bus interface than the AXI bus and is used to connect from the processor to the I/O devices
- It is a simple master/slave bus
  - The CPU (master) initiates a data transfer
  - The I/O device (slave) responds
  - A memory cycle begins when CYC_O and STB_O go active along with a valid ADR_O and WE_O (0 for read, 1 for write)
  - If it's a write cycle DAT_O also drives the data to be written and SEL_O will determine which of the 4 bytes making up the 32 bits are to be written
  - A memory cycle concludes when the slave returns ACK at which point data has been written or read data on DAT_I is valid and will be sampled on the clock edge
  - After this CYC_O, STB_O, SEL_O, WE_O, and DAT_O are released and the cycle has finished



Block Microarchitecture



- Implement your function
- FSMs
- Arithmetic and Logical Code

May not have any primary IO

A register does NOT need to use all the bits

SoC Modification

You will need to MODIFY RVFPGA Source Files

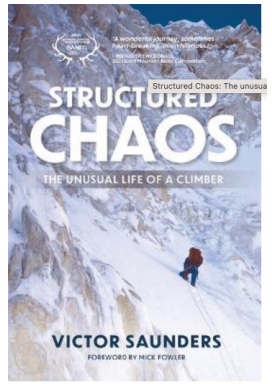
- To instantiate your new block (structural HDL code)
- To Connect it to the bus
- To extend the Address Decoder
- To extend the Interrupt Controller (possibly)

- To encode the Memory Map in and software (.h file)

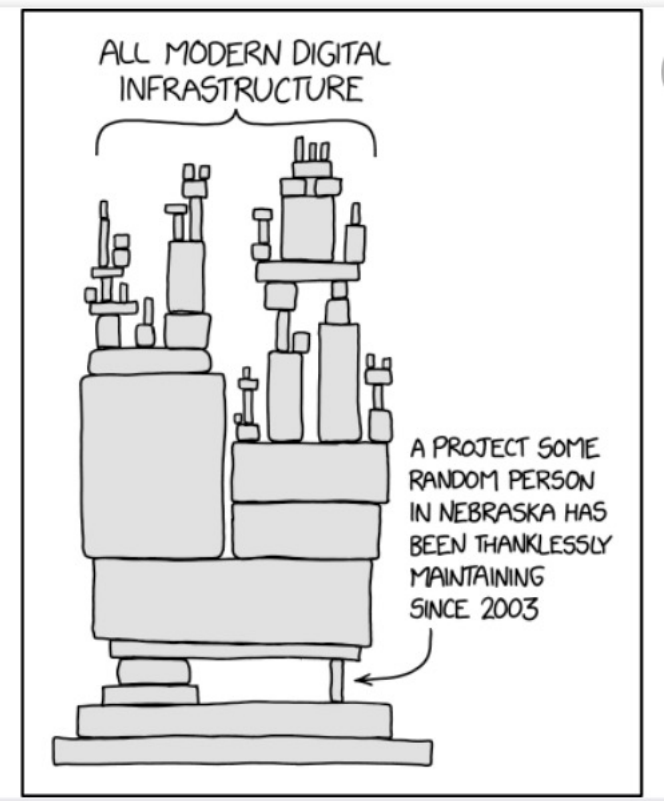
- Your HARDWARE and SOFTWARE memory map need to be consistent
  - One is an ,h include file for your assembler or c code
  - One is a fixed function or better still and include .vh file for your verilog

A *very* important note on complexity……..

The Computer you run on – there are choices – none of them are perfectg



Uniquification



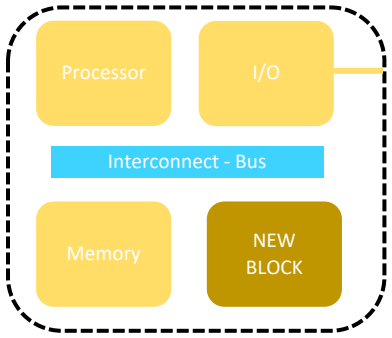
- IP Mix
- SW Mix
- Design Methodology
- Tool Chains
- Target Technology/Cost
- Service Lifetime

- **Design Patterns**
- Problem Solving Approaches
- Abstractions

- Team
- Supply chain
- Budget

Platform.io helps manage configurations of tools

System on Chip Project



Block Diagram

C code

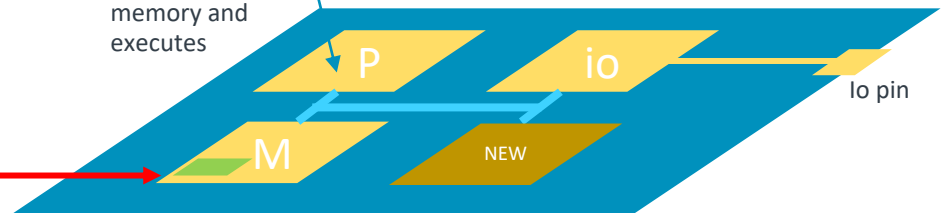
C Compiler

.asm – human-readable assembly code

Binary file with bit pattern that encodes instructions for the processor

Binary code is loaded into memory

Processor fetches instruction from memory and executes



Io pin

la t0, Ox100

Example assembler pseudo instruction to load the contents of address Ox100 into processor register t0. This reads a specific location in the memory map

Logical Architecture HDL Code <filename>.v

Vivado synthesizes the HDL code into a bit file

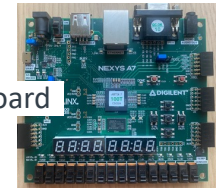
Vivado

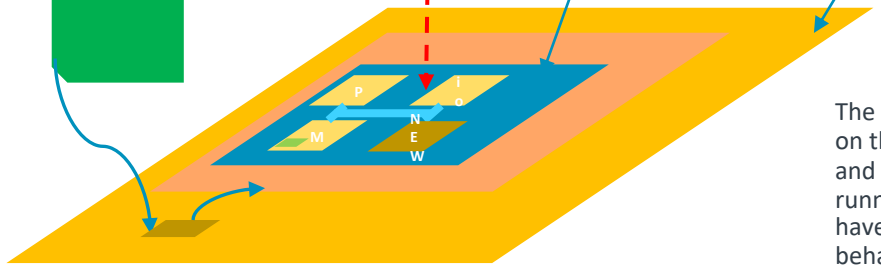
bitfile

The Bitfile configures the FPGA hardware to implement the Logical Architecture including io pin mapping.

Artix7 FPGA

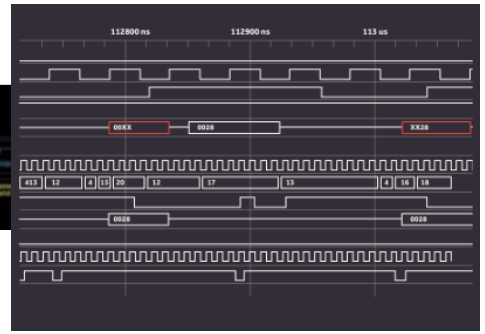
Nexys7 FPGA Board





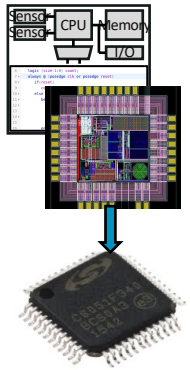
Microcontroller on the Nexys Board Loads Bitfile onto the unconfigured FPGA chip – this is all taken care of in the Vivado suite

The implementation on the FPGA board and the Simulation running in Verilator have the same logical behaviour



Emulation

Simulation



The same HDL can go through synthesis and place and route to create custom silicon

Test Planning

You will NEED a test plan to check your implementation works AND to Demonstrate Results

- Start small – You can test the function of your block or a sub-part of it in a BLOCK LEVEL TESTBENCH
- Be thoughtful about your integration – maybe start by seeing if you can write to a control register and read from a status register (look at the examples in the other IO blocks
  - You will need to write some .asm or .c code to do this
- Check it in simulation before synthesis and putting on the FPGA
- Synthesizing to FPGA ensures that your HDL can be implemented
  - Behavioral Code cannot – **Synthesizing is part of your test plan will give you area results as well.**
- The FPGA is great for running system-level integration tests
  - Maybe you can run a system-level loop back or connect to something ***real***

- ***BE THOUGHTFUL AND STRUCTURED ABOUT YOUR TEST PLAN IT WILL SAVE YOU TIME***

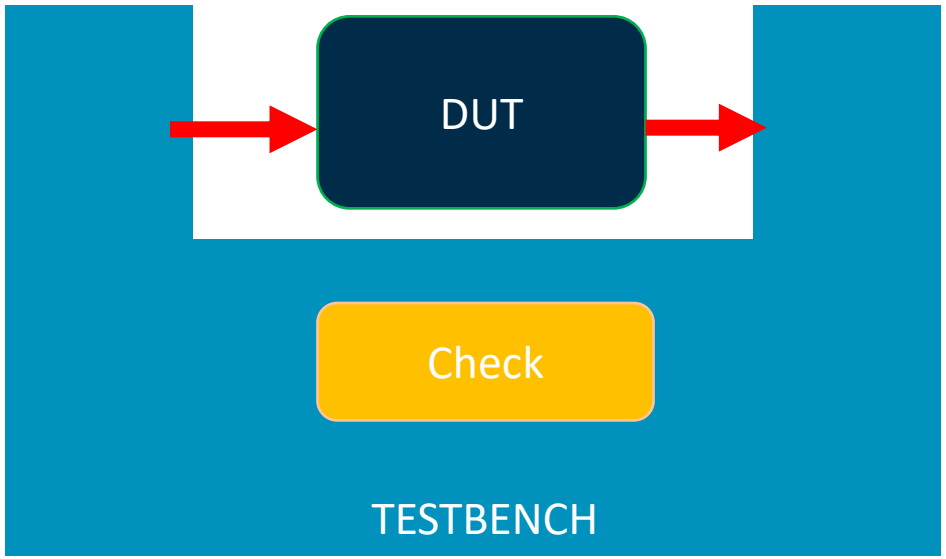
Testing you HDL

RECOMMEND TESTING COMPLEX FUNCTIONS BEFORE YOU INTEGRATE

IF YOU WRITE NEW HDL YOU WILL NEED TO TEST IT

You might *not* want to jump straight to the integration

- DUT – device under test
- Testing your testbench…..
  - Behavioral checking code for your function?
- Functionality



The DUT can be simple or complex – e,g complete RVFPGAsim