ISE

- ISE is ...

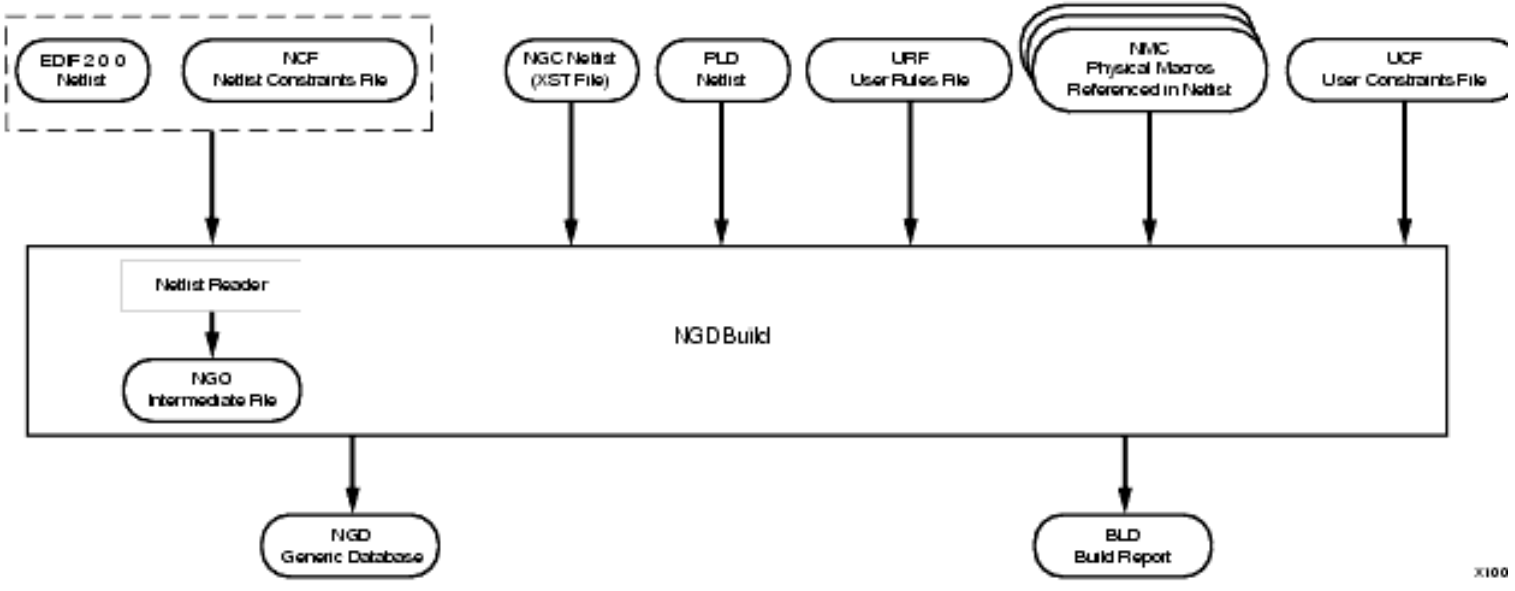
How Do You 'Program' a Platform FPGA?

- software reference design
- decompose application into
  - FPGA components
  - Processor components
- implement FPGA component(s)
- implement FPGA/Processor interface(s)
- cross your fingers and type `a.out`

Schematic Capture

- technically possible but consider a 64 bit PLB bus and one PPC core — several hundred signals to connect!
  - address, data, control lines
  - memory controller has to be added
  - ...

ngdbuild


EDIF 2 0 0
Netlist
NCF
Netlist Constraints File
NGC Netlist
(XST File)
PLD
Netlist
URF
User Rules File
NMC
Physical Macros
Referenced in Netlist
UCF
User Constraints File
Netlist Reader
NGO
Intermediate File
NGDBuild
NGD
Generic Database
BLD
Build Report
X100

ngdbuild

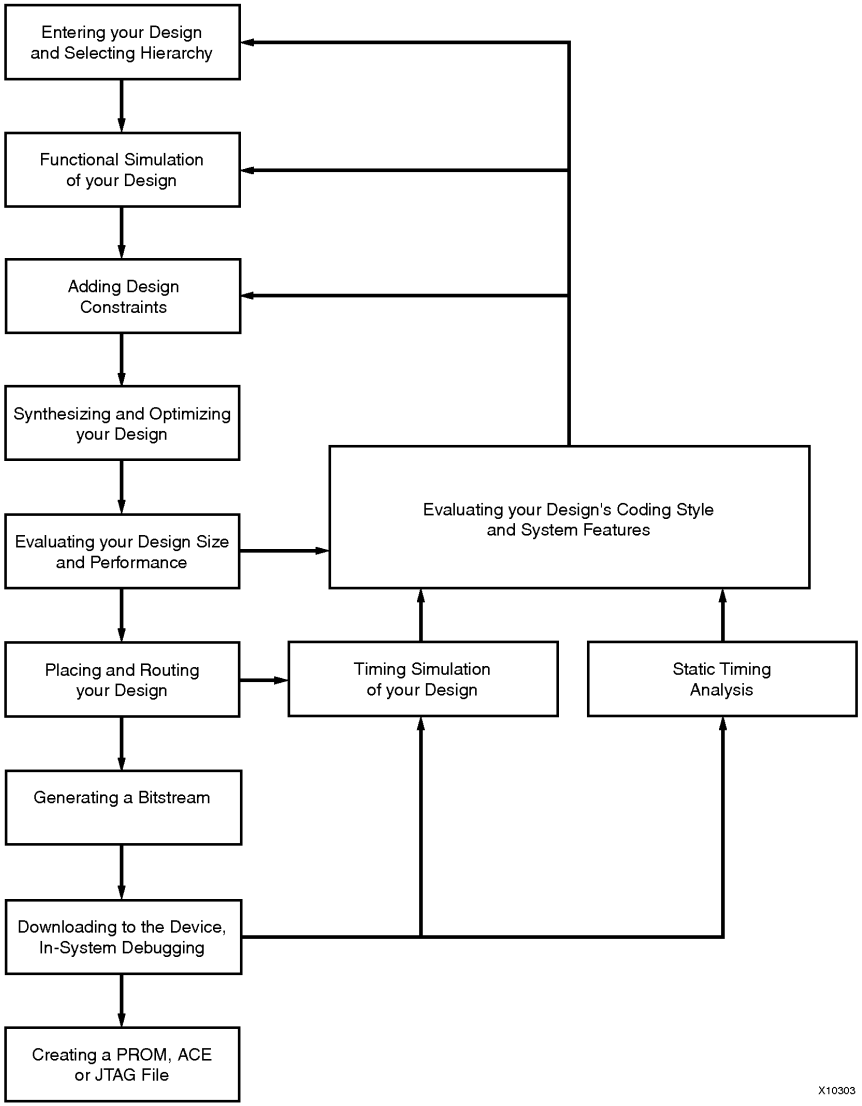


Figure 2-1: **Design Flow Overview Diagram**

Basic Terms

- ***configuring an FPGA*** means loading all of the individual SRAM cells
- an FPGA is ***configured*** either at power-on or at run-time
- the ***configuration*** is called a bitstream or a `.bit` file (for its file extension)
- a ***tool chain*** is used to generate the configuration

Tool Chain

- a ***tool chain*** is used to generate the configuration
  - Synthesize $\Rightarrow$
  - Technology Mapping $\Rightarrow$
  - Place & Route $\Rightarrow$
  - Bitstream Generation

Synthesis Options

- MAP/PAR/BITGEN three steps are FPGA vendor-specific; use vendor-supplied tools
- synthesis tools
  - vendor-supplied synthesis tools: `xst`
  - third-party synthesis tools: Synopsis, Synplicity
- input to synthesis tools
  - FPGA Editor
  - Schematic Capture
  - **Hardware Description Languages**
  - High-Level Languages (software)

FPGA Editor

- the lowest abstraction level is the FPGA Editor; vendor-specific tool
  - graphical/script tool that allows the user to set LUTs, Flip-Flops, routes individually
  - closest to designing in terms of AND, OR, NOT
  - least productive
- forgoes the MAP and PAR steps

Schematic Capture

- only slightly higher level than FPGA Editor; CAD tool allows designer to describe AND/OR/NOT gates graphically
- tools may or may not perform minimization
- CAD tool groups gates and maps to LUTs, handles place-and-route

Hardware Description Langauges

- ***Hardware Description Languages*** (HDLs) — use strings of characters (text-based files) that follow some language's syntax and semantics
- informally, describes hardware with words
- Traditional HDLs:
  - Verilog
  - VHDL

HDLs and Productivity

- HDLs are widely used for...
  - documenting behavior
  - testing and verifying circuits (simulation)
  - bulk of custom circuit layup
  - ASIC design
  - FPGA design
- for larger projects, far more productive than other techniques
- higher level of abstraction

Traditional HDL History

- goals, use, and capability of HDLs and HDL CAD tools has metamorphized over the last 25 years
  - originally: just used to document
  - simulation came next
  - synthesizing structural design
  - "high level synthesis" (HLS) or synthesizing behavior design is current

Alternative HDLs

although VHDL/Verilog are the most common; alternatives exist

- object-oriented circuit generators
- system-level (or co-design) languages

Object-Oriented Circuit Generators

- different approach to high-level languages and synthesis:
  use a modern high-level language (Java, C++) to describe circuits
  - PAM-Blox — uses C++ for PCI PAM board
  - JHDL — Java to generate netlists
- primarily structural

System-Level Description and Verification

- Examples:
  - SystemC — C++ and behavioral descriptions
  - SystemVerilog — extends Verilog to add C structures
- Goals: support *co-design*; that is, large systems that include hardware and software

Introduction to VHDL

- two major forms/styles of expressing logic
  - structural / data flow
  - behaviorial

Be Aware

- every VHDL code can be simulated
- two logically equivalent codes may perform completely different in simulation
- not every valid VHDL code can be synthesized!

VHDL Syntax

- shares a lot of conventions with Ada
- two major parts
  - entity declaration
  - followed by one or more architecture declarations

Entity

- describes the interface of the component
  - similar to the role of a function prototype in C
  - include file/class declaration in C++

Architecture

- provides and implementation of the component
  - a function (or method) in C (C++)
- may have multiple implementations for different roles (simulation v. synthesis; ASIC v. FPGA)

(Dataflow) Example

```
------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fa is port (
        a, b, cin  : in std_logic ;
        s : out std_logic ;
        cout : out std_logic ) ;
end fa ;
architecture fa_df of fa is
begin
        s <= a xor b xor cin ;
        cout <= (a and b) or (b and cin) or (a and cin) ;
end fa_df ;
------------------------------
```

Structural VHDL

- previous example is sometimes called `dataflow' approach
- structural is conceptually similar
  - both are spatial design that can describes networks of gates
  - structural can describe networks of larger components
  - different syntax

Structural Syntax

- declare entity as before
- in architecture
  - declare components and signals
  - instantiate units in begin/end block

Structural Example (1 of 2)

```
------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fa is port (
    a, b, cin  : in std_logic ;
    s : out std_logic ;
    cout : out std_logic ) ;
end fa ;
```

Structural Example (2 of 2)

```
architecture fa_struct of fa is
    component xorgate port ( x,y : in std_logic ;
        f : out std_logic ) ;
    end component;
    component andgate port ( x,y : in std_logic ;
        f : out std_logic ) ;
    end component;
    component or3gate port ( x,y,z : in std_logic ;
        f : out std_logic ) ;
    signal t1, t2, t3, t4 : std_logic ;
begin
    u0: xorgate port map (a,b,t1) ;
    u1: xorgate port map (t1,cin,s) ;
    u3: or3gate port map (t2,t3,t4,cout) ;
    u4: andgate port map (a,b,t2) ;
    u5: andgate ...
end fa_struct ;
-----------------------------
```

Behavioral VHDL

- in dataflow and structural; the order of the statements in begin/end doesn't matter
- in contrast, in behavioral VHDL uses a process block and the statements are executed sequentially
  - provides full-powered procedural language (Ada)
  - the last value `assigned' to a signal is the one produced by the process
  - if a signal is *not* assigned a value; it retains its previous value

Behavioral Example

```
------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fa is port (
    a, b, cin  : in std_logic ;
    s : out std_logic ;
    cout : out std_logic ) ;
end fa ;
architecture fa_behav of fa is
    signal result : std_logic_vector(1 downto 0);
    process(a,b,cin) begin
        result = ('0'&a) + ('0'&b) + ('0'&cin) ;
        s <= result(0) ;
        cout <= result(1) ;
    end process ;
end fa_behav ;
------------------------------
```

Behavioral to Logic Network

- behavioral can be directly used in simulation
- in contrast, in behavioral VHDL uses a process block and the statements are executed sequentially
  - provides full-powered procedural language (Ada)
  - the last value 'assigned' to a signal is the one produced by the process
  - if a signal is *not* assigned a value; it retains its previous value

Modelsim

demo modelsim and simulation

Custom Core Diagram

Using Xilinx's **Create or Import Peripheral Wizard** we create a custom hardware core and connect it to the OPB.

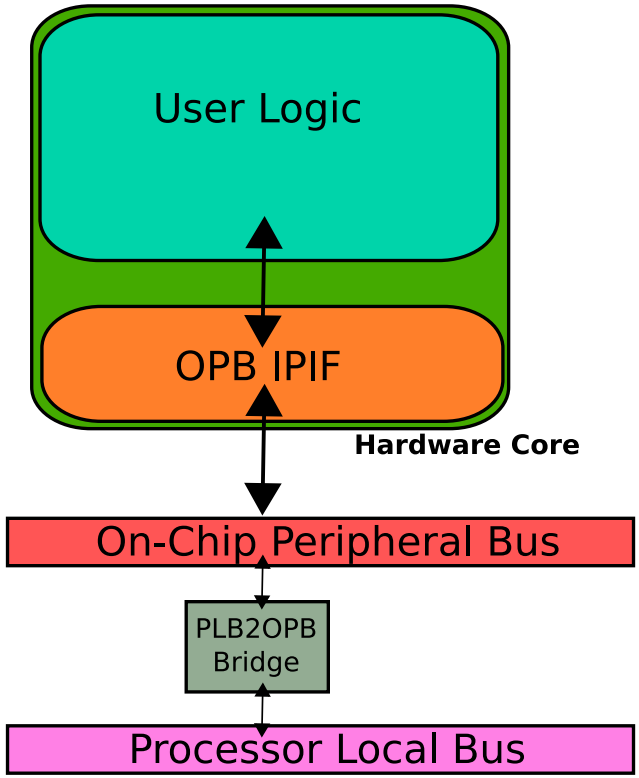


Figure 1: Custom Core Connected to OPB

Hardware Core's Slave Registers

Create a hardware core with four slave registers.

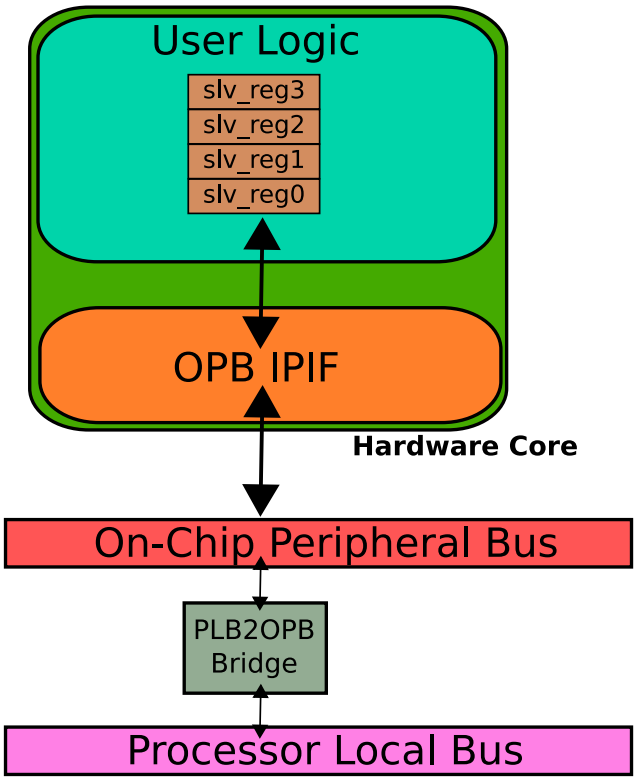


Figure 2: Custom Core Connected to OPB

Hardware Core's Address Range

When hooking up the hardware core we specify the address range to be **0x78000000 - 0x7800FFFF**. The Slave Registers are 32-bits wide. Now, each register is addressible.

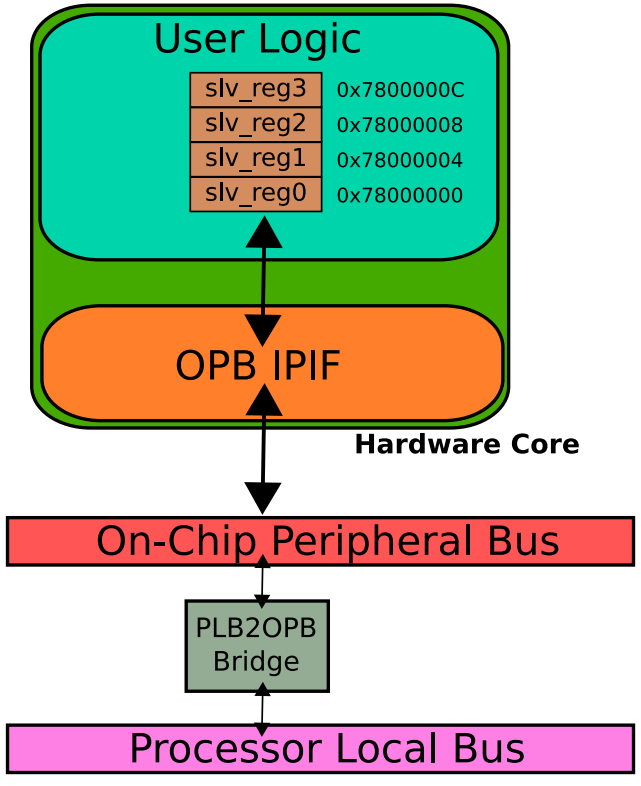


Figure 3: Custom Core Connected to OPB

Modifying Hardware Core's Functionality

Create Process to add 5 to the value in **Register 1** then store the result in **Register 3**.

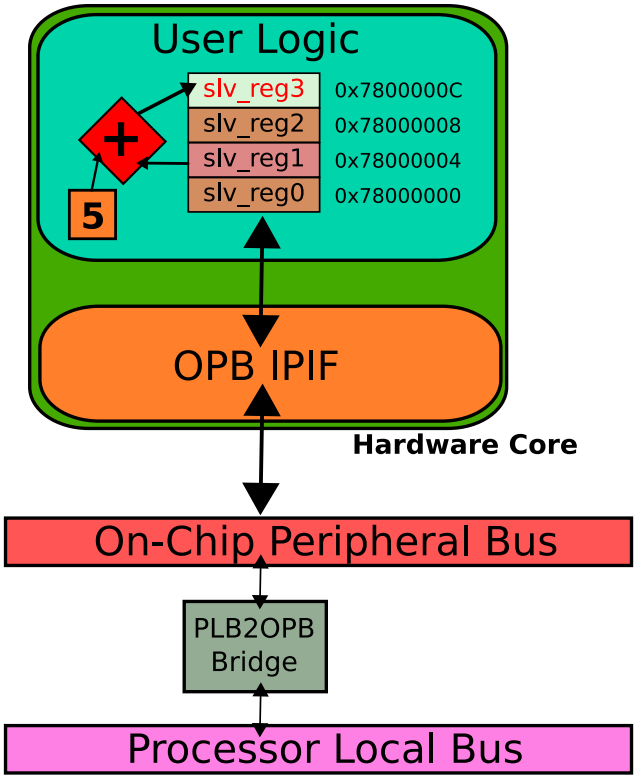


Figure 4: Custom Core Connected to OPB

User Logic File - user_logic.vhd

In build directory:

- pcores/my_test_core_v1_00_a/hdl/vhdl/user_logic.vhd
- Four important parts:
  1. at line 84: **entity user_logic**
     - Used to Connect User Logic to IP Interface (then to Bus)
  2. lines 131 - 134: Slave Registers assessible by other cores
     - Number of registers depends on choice during the Wizard
  3. at line 170: **SLAVE_REG_WRITE_PROC**
     - Process which stores data on the bus into specific register
     - For Example: PPC writes 0x10 to slv_reg0
  4. at line 213: **SLAVE_REG_READ_PROC**
     - Process which puts value in specific register onto bus
     - For Example: PPC reads data in slv_reg0

Adding A Process

We want to add 5 to the value in Register 1 and store the result in Register 3.

1. Create a new Process: **ADD_5_PROC**
2. Process only depends (sensitive) to Register 1 changing
3. Store Result in Register 3
   - Only one process can modify register value
   - Register 3 has two other places where it is being written to it

Sample Code

Process to Add 5 to Slave Register 1 and store result in Slave Register 3:

```
ADD_5_PROC : process( slv_reg1 ) is
begin
  -- Add 5 to Slave Register 1 and Store
  -- result in Slave Register 3;
  slv_reg3 <= slv_reg1 + 5;

end process Add_5_PROC;
```

C Code

From the PowerPC create an application (in C) that can write a value to **Slave Register 1** and then read back the result of the calculation which is stored in **Slave Register 3**

1. How do you write data to Slave Register 1?
2. How do you read data from Slave Register 3?
3. What is the Address of Slave Register 1?
4. What is the Address of Slave Register 3?

C Code Example

add example standalone C code here