Logic and High Level Synthesis on FPGA

Stefano Spellini

Acknowledgements

- **This lesson has been made possible thanks to the courtesy of Xilinx Inc.**
  - Leading company in the programmer logic market
  - Inventor of the FPGA (1985)
  - First semiconductor company with a fabless manufacturing model

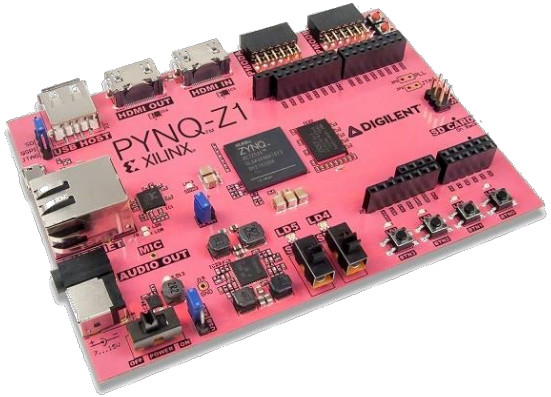
- Xilinx Inc. donated to the University of Verona:
  - 5 PYNQ-Z1 FPGAs,
  - 25 SDSoC licenses,
- And gave full access to their training material

- More information on the **Xilinx University Program**:
  - https://www.xilinx.com/support/university.html

Field Programmable Gate Array

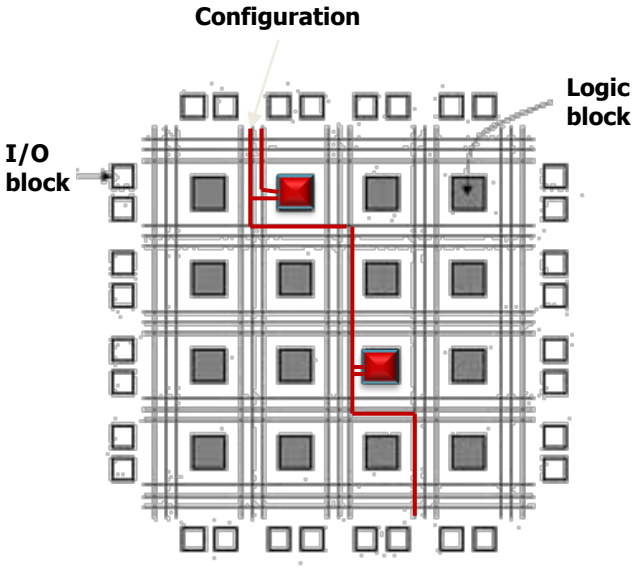
- Field-Programmable Gate Array
  - Integrated circuit designed to be configured by customer/designer after manufacturing
    - Can implement any functionality (as ASICs)
    - Can be re-configured
      - Update functionality after shipping
    - Ideal when volume of production is low
    - Used for validating designs before they are produced
      - Reduce time to market
      - Find bugs (pre-silicon and post-silicon)

- All Programmable SoC (APSoC)
  - FPGA + General Purpose CPU
  - SW/HW Co-design
    - VHDL/Verilog and C/C++
    - Python only for the PYNQ-Z1



Field Programmable Gate Array

- Contain
  - Logic blocks
    - Simple logic functions
    - Complex combinational functions
  - Recoinfigurable interconnects
    - Allow blocks to be wired together
  - Memory elements
    - Flip flops
    - Blocks of memory



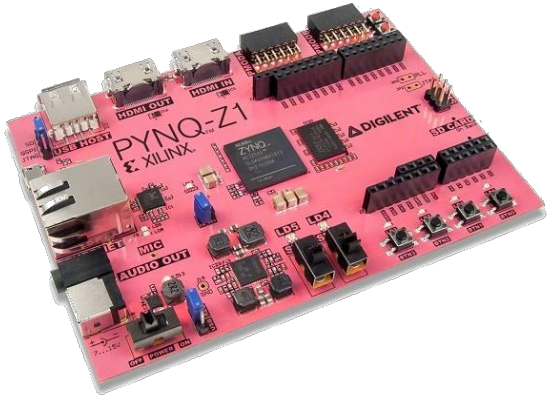
Field Programmable Gate Array

- Define the behavior
  - HDL or schematic design
  - Mapped netlist
    - Generated with an electronic design tool
  - Place-and-route
    - Fit the netlist to the actual FPGA
    - FPGA's company proprietary SW
  - FPGA configuration
    - Using the binary generated file
    - Transferred to the FPGA

- Result of logic synthesis strictly depends
  - Target architecture
  - Synthesis algorithm used
    - Synthesis tool

PYNQ-Z1: Python Productivity for Zynq (SoC)

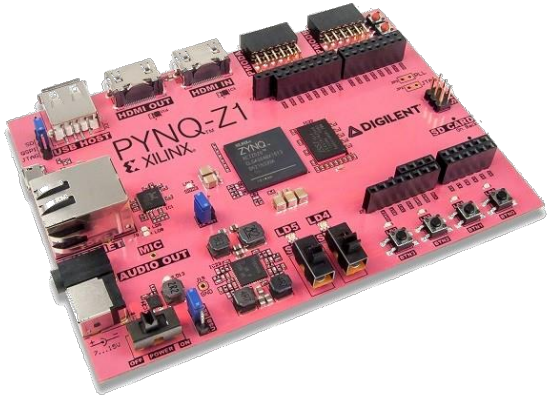
- **Dual ARM® Cortex™-A9 MPCore™ with CoreSight™**
  - 32 KB Instruction, 32 KB Data per processor L1 Cache
  - 512 KB unified L2 Cache
  - 256 KB On-Chip Memory
- **1.3 M reconfigurable gates**
  - **22 nm technology!**
- **512MB DDR3 / FLASH**
- Four clock management tiles220 DSP slices
- Internal clock speeds exceeding 450MHz



- http://www.pynq.io/

PYNQ- Python Productivity for Zynq (misc)

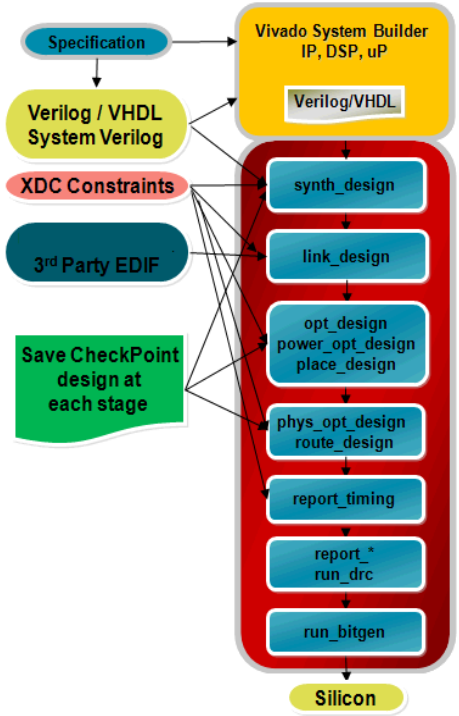
- I/O Interfaces
  - USB-JTAG Programming circuitry
  - USB OTG 2.0
  - USB-UART bridge
  - One 10/100/1G Ethernet
  - HDMI Input/Output
  - Microphone and3.5mm mono audio output jack
- Switches and LEDs
  - 2 Slide switches
  - 2 RGB LEDs
  - 4 LEDs
  - 4 Push-buttons
- Expansion ports
  - 2 Pmod ports
  - 1 Arduino Shield
  - 16 GPIO
  - 10 Single-ended Analog inputs to XADC



XILINX VIVADO

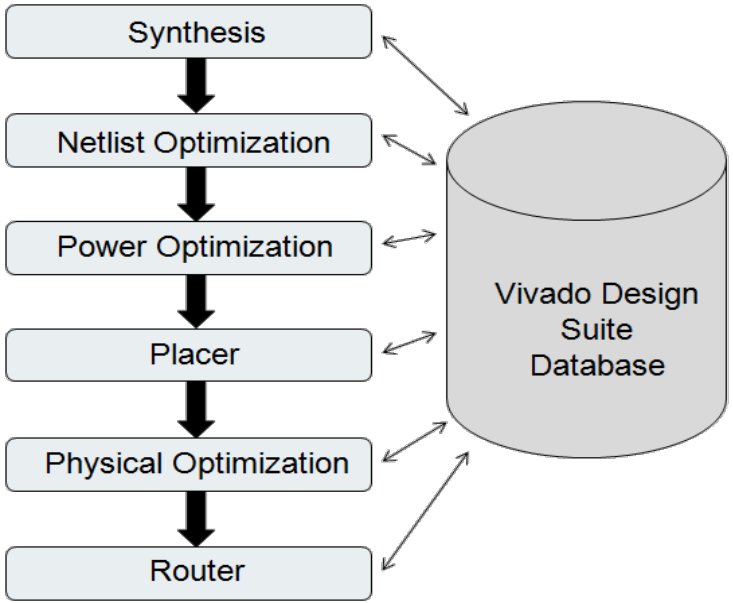
Typical vs Vivado Design Flow

- Interactive IP plug-n-play environment
  - AXI4, IP_XACT
- Common constraint language (XDC) throughout flow
  - Apply constraints at any stage
- Reporting at any stage
  - Robust Tcl API
- Common data model throughout the flow
  - “In memory” model improves speed
  - Generate reports at all stages
- Save checkpoint designs at any stage
  - Netlist, constraints, place and route results



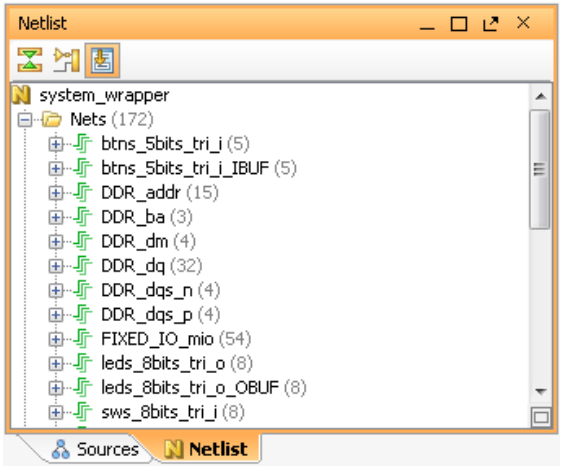
Design Database

- Processes access the underlying database of your design
  - Each process operates on a netlist and will modify the netlist or create a new netlist
- Different netlists are used throughout the design process
  - Elaborated
  - Synthesized
  - Implemented

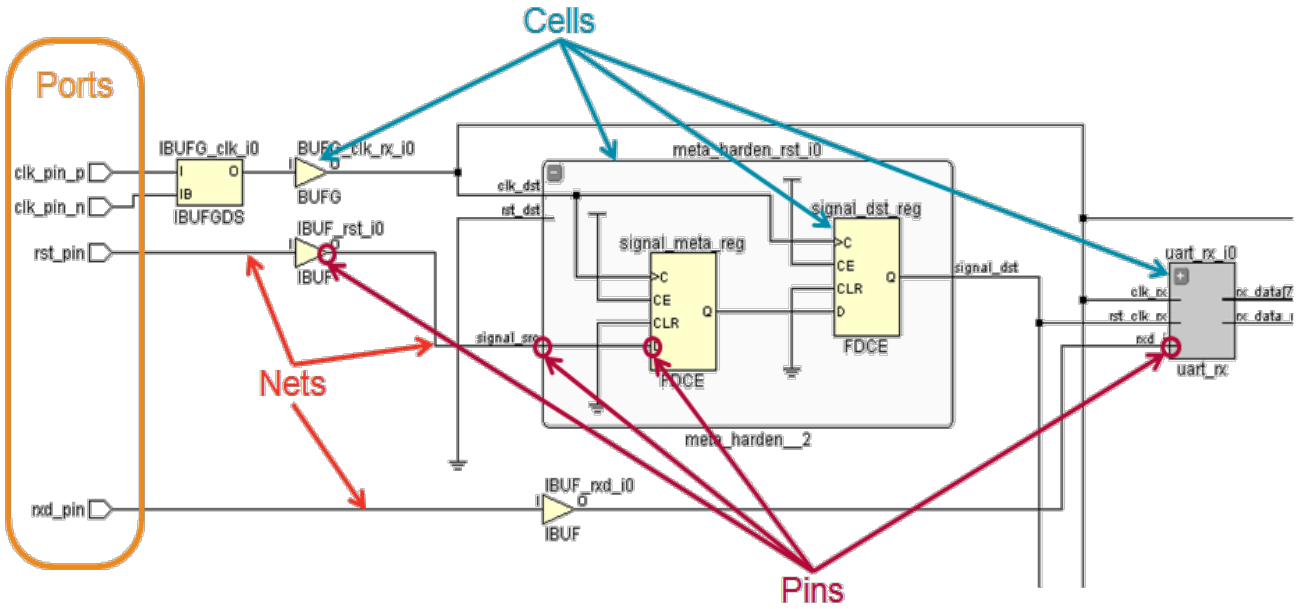


What is a Netlist?

- A Netlist is a lower-level description of your design
  - Consists of cells, pins, port and nets
  - Cells are design objects
    - Instances of user modules/entities
    - Instances of library Basic Elements (BELs)
      - LUTs, FF, RAMs, DSP cells, etc…
    - Generic technology representations of hardware functions
    - Black boxes
  - Pins are connection points on cells
  - Ports are the top level ports of your design
  - Nets make connections between pins and from pins to ports

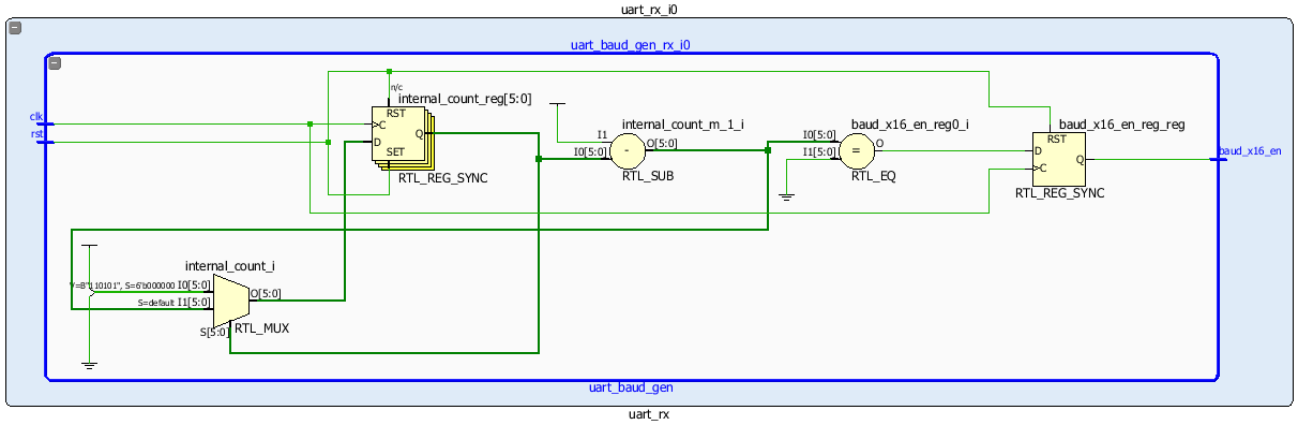


Netlist Objects



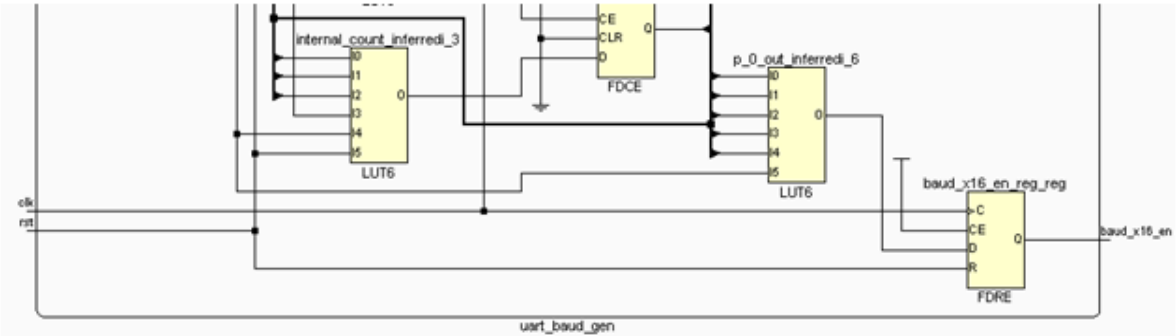
Elaborated Design

- Representation of the design before synthesis
  - Interconnected netlist of hierarchical and generic technology cells
    - Instances of modules/entities
    - Generic technology representations of hardware components
      - AND, OR, buffer, multiplexers, adders, comparators, etc…



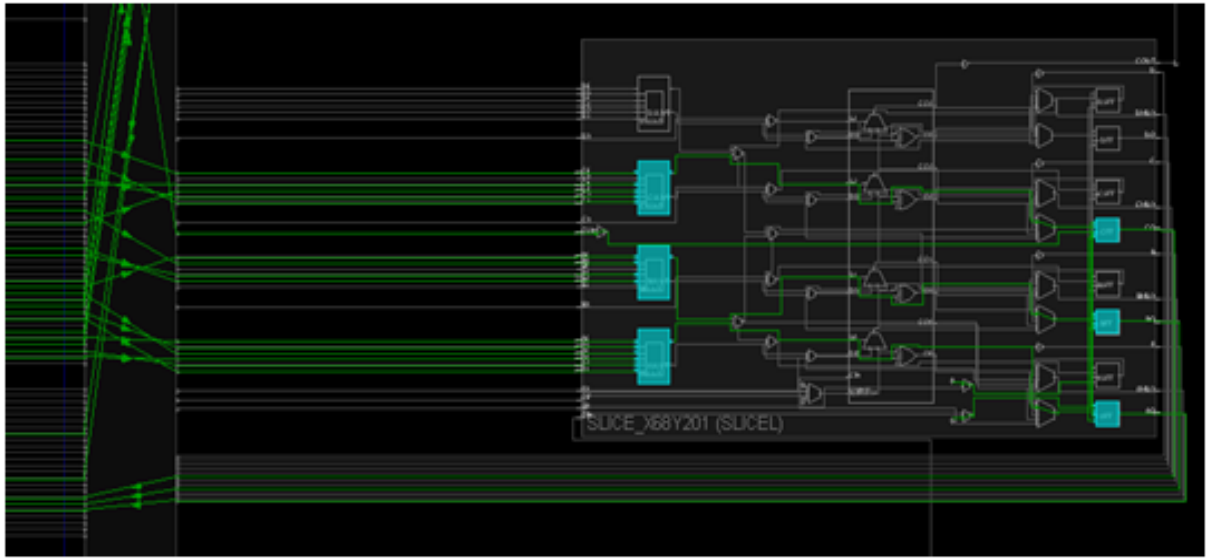
Synthesized Design

- Representation of the design after synthesis
  - Interconnected netlist of hierarchical and Basic Elements (BELs)
    - Instances of modules/entities
    - BELs
      - LUTs, flip-flops, carry chain elements, wide MUXes
      - Block RAMs, DSP cells
      - Clocking elements (BUFG, BUFR, MMCM, …)
      - I/O elements (IBUF, OBUF, I/O flip-flops)
- Object names are the same as names in the elaborated netlist when possible



Implemented Design

- Representation of the design during and after the implementation process
  - Structurally similar to the Synthesized Design
  - Cells have locations, and nets are mapped to specific routing channels

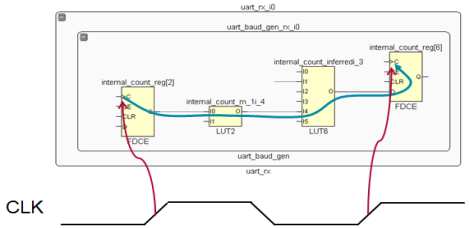


Implementation

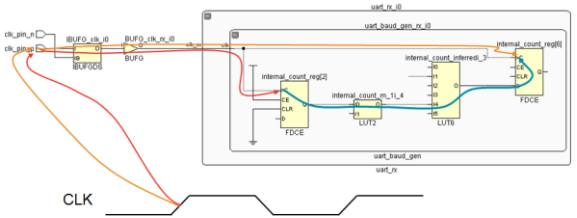
- Place and route the netlist of the design on the target FPGA
- Add constraints defined in the previous steps (.xdc file)

- Timing constraint
  - Use to set the clock of the design

- Setup time
  - Amount of time before the latching clock edge in which an input signal has to reach its expected value, so that the output signal will reach the expected logical value within a specific delay
- Hold time
  - Time after the latching clock edge in which an input signal should remain stable so that the output of flip-flop won't go into metastable state and can reach to its expected value



Setup check: Checks that a change in a clocked element has time to propagate to other clocked elements before the next clock event



Hold check: Checks that a change in a clocked element caused by a clock event does not propagate to a destination clocked element before the same clock event arrives at the destination element

Timing Metrics

Setup

- Worst Negative Slack (WNS) ⇒ THIS HAS TO BE POSITIVE
  - Corresponds to the worst slack of all the timing paths for max delay analysis. It can be positive or negative.
- Total Negative Slack (TNS)
  - Sum of all WNS violations, when considering only the worst violation of each timing path endpoint.
    - 0ns when all timing constraints are met for max delay analysis.
    - Negative when there are some violations

Hold

- Worst Hold Slack (WHS) ⇒ THIS HAS TO BE ≥ 0
  - Corresponds to the worst slack of all the timing paths for min delay analysis. It can be positive or negative.
- Total Hold Slack (THS)
  - Sum of all WHS violations, when considering only the worst violation of each timing path endpoint.

RTL to Silicon

LOGIC SYNTHESIS USING XILINX VIVADO

Vivado into action

- Download the tar.gz from the elearning
  - `$> tar xzf 05_sources.tar.gz`
  - `$> ls`

```
cpp/     systemc/    vhdl/
```

The directories contains three different implementations of the root

- Open Vivado
  - Click on **Create New Project**
    - A dialog window will appear, click on Next
  - Choose the **directory** where you want to save the project
    - E.g., /home/user/PSE/vivado_projects
  - Give a **name** to the project
    - E.g., root_rtl_synthesis
  - **Tick** the «create project subdirectory» option
  - **Next**

Vivado into action

- Project type: choose the **RTL Project** option
- **Next**
- **Click on the green cross**
  - **Add the root.vhdl file** in the vhdl directory you downloaded
  - Tick all the options
    - Scan and add RTL include files into project
    - **Copy sources into project**
  - Be sure that the Target the Simulator Languages **are set on VHDL**
  - Next
  - Skip the next page (Next again)
  - Skip the next page (Next again) [will be used in the next lecture]
  - Choose the PYNQ
    - Family: Zynq-7000
    - Package: clg400
    - Speed grade: -1
    - **Code: xc7z020clg400-1**
  - **Finish**

Fast lane for the silicon

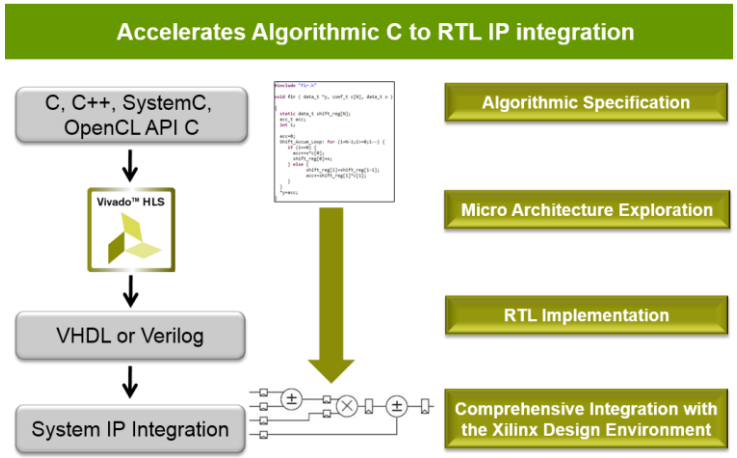
- Perform the RTL Analysis
  - Create the *Elaborated Design*
  - Click on **`Open Elaborated Design`**
- Perform the Synthesis
  - Create the *Synthesized Design*
  - Click on **`Run Synthesis`**
    - `Try adding timing constrain before Implementation`
    - `10ns clock period vs 1ns clock period`
    - `Report Timing`
- Implement the Design
  - Create the *Implemented Design*
  - Click on **`Run Implementation`**
- Create the Bitstream for the FPGA
  - Generate the «image» for the FPGA
  - Click on **`Generate Bitstream`**
- Deploy the component on the FPGA
  - **Next lecture!**

C to Silicon

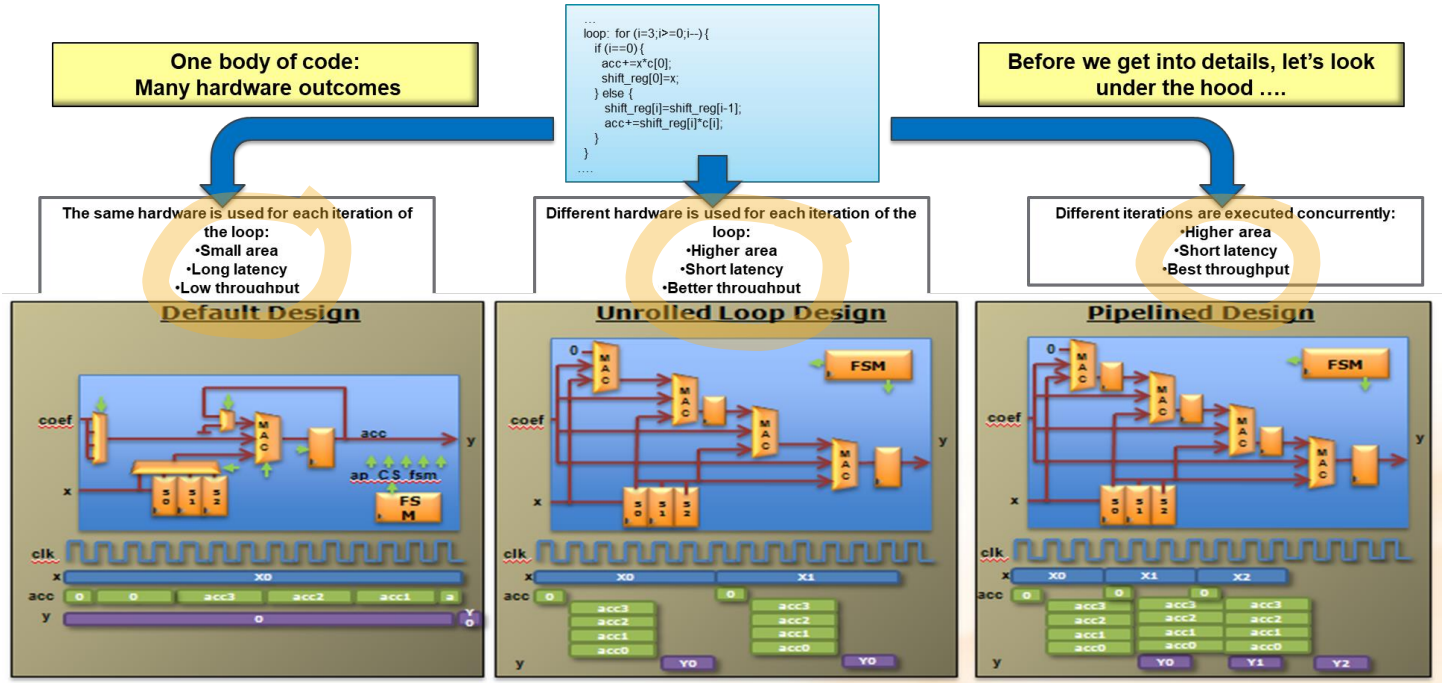
High-Level Synthesis using Xilinx Vivado HLS

High-Level Synthesis: HLS

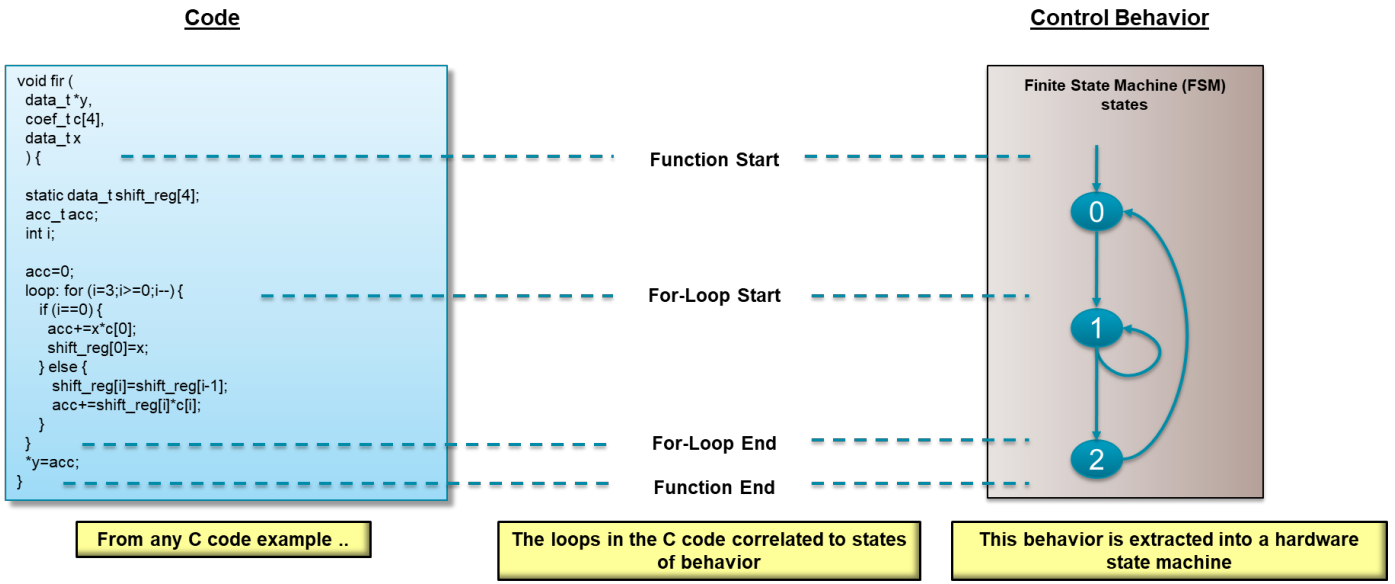
- High-Level Synthesis
  - Creates an RTL implementation from C, C++, System C, OpenCL API C kernel code
  - Extracts control and dataflow from the source code
  - Implements the design based on defaults and user applied directives
- Many implementation are possible from the same source description
  - Smaller designs, faster designs, optimal designs
  - Enables design exploration



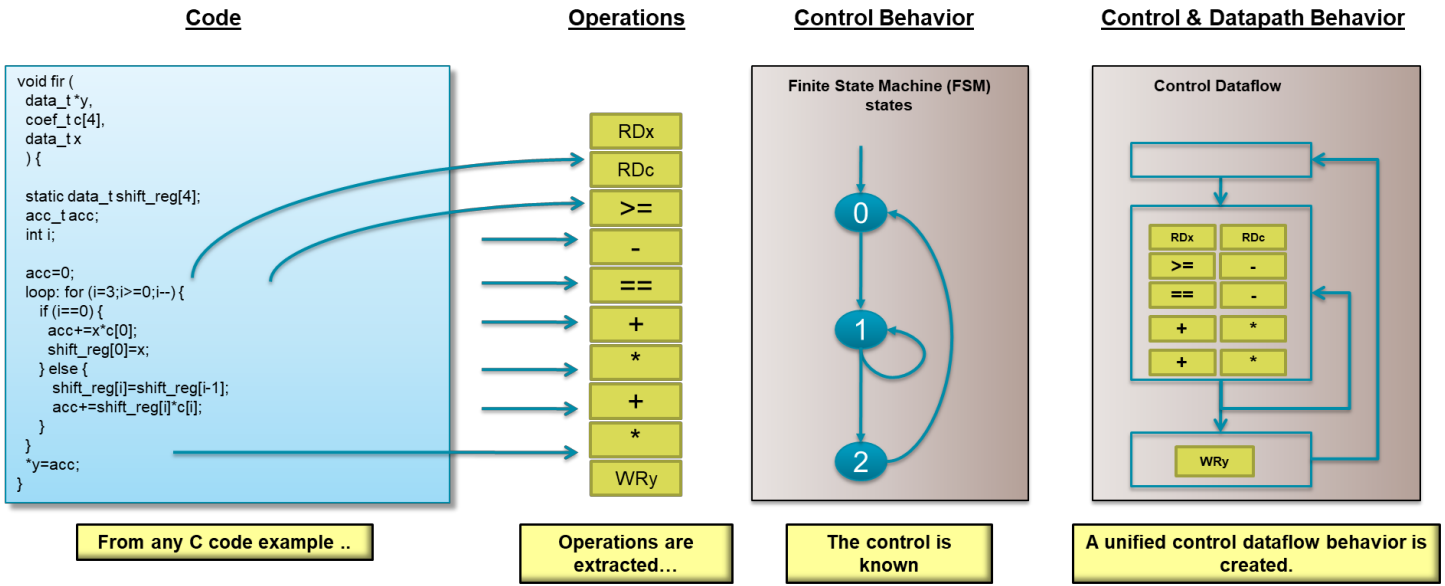
Design Exploration with Directives



HLS: Control Extraction

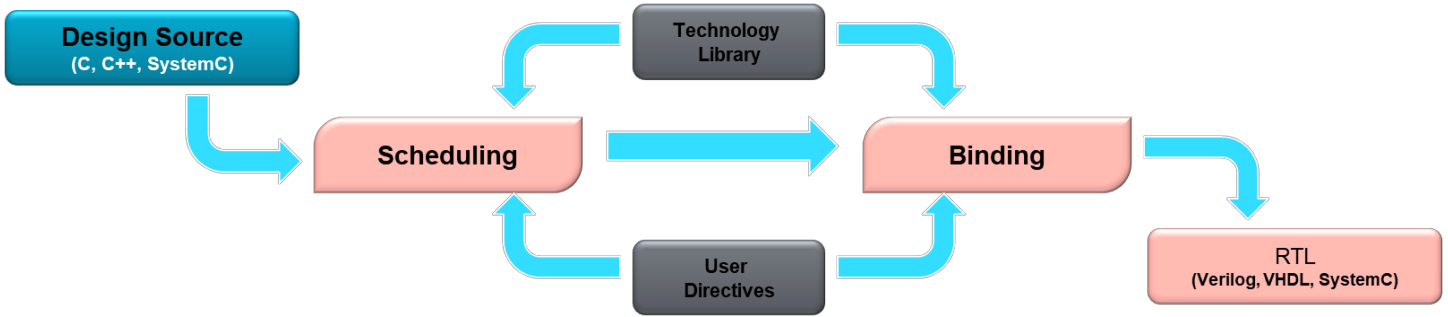


HLS: Control & Datapath Extraction



High-Level Synthesis: Scheduling & Binding

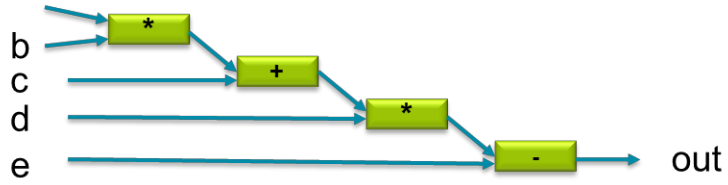
- Scheduling & Binding
  - Scheduling and Binding are at the heart of HLS
- Scheduling determines in which clock cycle an operation will occur
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- Binding determines which library cell is used for each operation
  - Takes into account component delays, user directives



Scheduling

- The operations in the control flow graph are mapped into clock cycles

```
void foo (
 …
  t1 = a * b;
  t2 = c + t1;
  t3 = d * t2;
  out = t3 – e;
}
```



Schedule 1

- The technology and user constraints impact the schedule
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle

Schedule 2

- The code also impacts the schedule
  - Code implications and data dependencies must be obeyed

Binding

- Binding is where operations are mapped to cores from the hardware library
  - Operators map to cores

- Binding Decision: to share
  - Given this schedule:

    - Binding must use 2 multipliers, since both are in the same cycle
    - It can decide to use an adder and subtractor or *share* one addsub

- Binding Decision: or not to share
  - Given this schedule:

    - Binding may decide to share the multipliers (each is used in a different cycle)
    - Or it may decide the cost of sharing (muxing) would impact timing and it may decide *not to share* them
    - It may make this same decision in the first example above too

The Key Attributes of C code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]  *   c[i];
}
  }
  *y=acc;
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Top Level IO : The arguments of the top-level function determine the hardware RTL interface ports

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance

Arrays: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

C, C++ and SystemC Support

- The vast majority of C, C++ and SystemC is supported
  - Provided it is statically defined at **compile time**
  - If it's not defined until **run time**, it won' be synthesizable

- Any of the three variants of C can be used
  - If C is used, Vivado HLS expects the file extensions to be .c
  - For C++ and SystemC it expects file extensions .cpp

Data types

- C and C++ have standard types created on the 8-bit boundary
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
    - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
    - Types: int8_t, uint16_t, uint32_t, int_64_t etc.
  - They result in hardware which is not bit-accurate and can give sub-standard QoR
- Vivado HLS provides bit-accurate types in both C and C++
  - Allow any arbitrary bit-width to be specified
  - Hence designers can improve the QoR of the hardware by specifying exact data widths
    - Can be specified in the code and simulated to ensure there is no loss of accuracy

Data types (cont.d)

- There are 4 basic types you can use for HLS (with bit accuracy)
  - Standard C/C++ Types
  - Vivado HLS enhancements to C: apint
  - Vivado HLS enhancements to C++: ap_int, ap_fixed
  - SystemC types

Type of C	C(C99) / C++	Vivado HLS ap_cint (bit-accurate with C)	Vivado HLS ap_int (bit-accurate with C++)	OSCI SystemC (IEEE 1666-2005 :bit-accurate)
Description		Used with standard C	Used with standard C++	IEEE standard
Requires		#include "ap_cint.h"	#include "ap_int.h" #include "ap_fixed.h" #include "hls_stream.h"	#include "systemc.h"
Pre-Synthesis Validation	gcc/g++		g++	g++
			Vivado HLS GUI	Vivado HLS GUI
Fixed Point	NA	NA	ap_fixed	#define SC_INCLUDE_FX sc_fixed
Signal Modeling	Variables	Variables	Variables Streams	Signals, Channels, TLM (1.0)

Floating Point Support

- Synthesis for floating point
  - Data types (IEEE-754 standard compliant)
    - Single-precision
      - 32 bit: 24-bit fraction, 8-bit exponent
    - Double-precision
      - 64 bit: 53-bit fraction, 11-bit exponent

- Support for Operators
  - Vivado HLS supports the Floating Point (FP) cores for each Xilinx technology (**PYNQ does**)
    - If Xilinx has a FP core, Vivado HLS supports it
    - It will automatically be synthesized
  - If there is no such FP core in the Xilinx technology, it will not be in the library
    - The design will be still synthesized by emulating the floating point

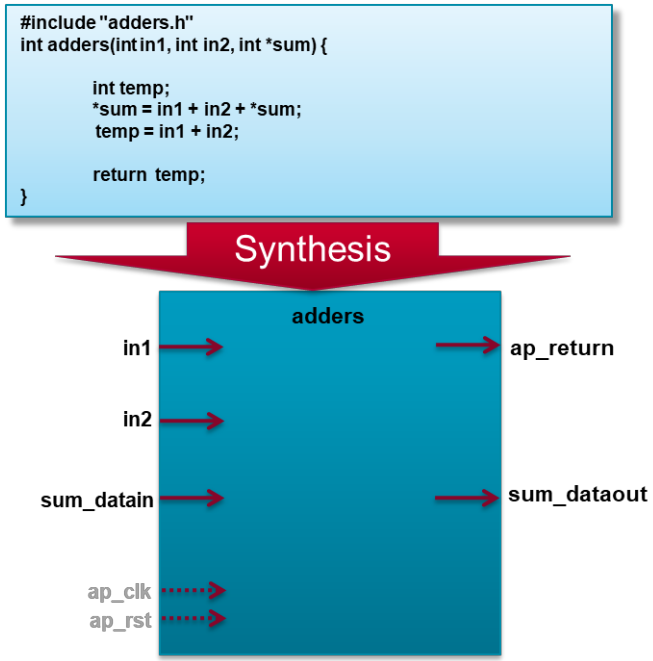
Input/Output

- The arguments of the top-level function must be transformed to hardware interfaces with an IO protocol

- Input:
  - Parameters passed as value to the function
  - Parameters passed by reference and read by the function

- Output
  - Return values
  - Parameters passed by reference and modified by the function

- Protocol and extra ports automatically generated by HLS algorithms



Vivado HLS into action (systemc)

- Download the tar.gz from the elearning
  - `$> tar xzf 05_sources.tar.gz`
  - `$> ls`

```
cpp/     systemc/    vhdl/
```

The directories contains three different implementations of the root

- Open Vivado HLS
  - Click on **Create a New Project**
    - A dialog window will appear, click on Next
  - Choose the **directory** where you want to save the project
    - E.g., /home/user/PSE/vivado_projects
  - Give a **name** to the project
    - E.g., root_systemc_hls
  - **Next**

Vivado HLS into action (systemc –cont.d)

- Click on **Add Files**
- Choose the `root_RTL.cpp` file previously downloaded
- Write **`root_RTL`** the **Top Function** field
- Click on Next
- Click on Next Again
- Click on the "…" button to choose a part
  - Choose the PYNQ (xc7z020clg400-1)
  - Necessary to choose a part, unknown library for HLS otherwise

- **Fast lane to the silicon**
  - **Click on "Run C Synthesis" button**

Vivado HLS into action (C++)

- Download the tar.gz from the elearning
  - `$> tar xzf 05_sources.tar.gz`
  - `$> ls`

```
cpp/     systemc/    vhdl/
```

The directories contains three different implementations of the root

- Open Vivado HLS
  - Click on **Create a New Project**
    - A dialog window will appear, click on Next
  - Choose the **directory** where you want to save the project
    - E.g., /home/user/PSE/vivado_projects
  - Give a **name** to the project
    - E.g., root_cpp_hls
  - **Next**

Vivado HLS into action (C++ –cont.d)

- Click on **Add Files**
- Choose the `root.cpp` file previously downloaded
- Choose the **`root`** function as **Top Function**
- Click on Next
- Click on Next Again
- Click on the "…" button to choose a part
  - Choose the PYNQ (xc7z020clg400-1)
  - Necessary to choose a part, unknown library for HLS otherwise

- **Fast lane to the silicon**
  - **Click on "Run C Synthesis" button**

Vivado HLS into action

- Export the RTL HLS as **IP Catalog**
  - IP archive created in the project folder, under *solutionx/impl/ip*
- Create a new Vivado Project
  - Without adding sources
  - Same board/device
- Select Project Settings, IP
  - Click '+' in the **Add Repository** and browse *solutionx/impl/ip* of the HLS design
- Now go to the **IP Catalog**
  - User repository, Vivado HLS IP
  - Double click on the IP, select 'Global' and leave the rest untouched
- **Your IP is now imported**
- **You can carry out synthesis and analysis as usual**

Literature (Logic Synthesis)

- Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. "*Multilevel logic synthesis.*" Proceedings of the IEEE 78.2 (1990): 264-300.

- Murgai, Rajeev, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. "*Logic synthesis for programmable gate arrays.*" In Proceedings of the 27th ACM/IEEE, Design Automation Conference*, 1990.* pp. 620-625. IEEE, 1990.

Literature (HLS)

- Coussy, Philippe, Daniel D. Gajski, Michael Meredith, and Andres Takach. "An introduction to high-level synthesis." *IEEE Design & Test of Computers* 26, no. 4 (2009): 8-17.

- Nane, Razvan, et al. "A survey and evaluation of FPGA high-level synthesis tools." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016): 1591-1604.

- Takach, Andres. "High-Level Synthesis: Status, Trends, and Future Directions." *IEEE Design & Test* 33.3 (2016): 116-124.

- Li, Huan, and Wenhua Ye. "Efficient implementation of FPGA based on Vivado High Level Synthesis." *Computer and Communications (ICCC), 2016 2nd IEEE International Conference on*. IEEE, 2016.

Lecture Assignment Lezione 16/11/2020

- Carry out the Logic Synthesis and the High Level Synthesis of the fixed-point multiplication
  - Logic Synthesis: the mult that you have developed (in Verilog or VHDL)
  - HLS: fixed-point multiplication (C/C++)
    - You must use the **'*'** operator
    - You must use the **Vivado HLS fixed point** type
- **Requirement: only the written report (2-4 pages) in .pdf in which you**
  - Compare the Schematics
    - # of Nets
    - # of Cells
    - …
  - Compare and comment the summaries of reports at the different stages
    - Timing
    - Power
    - Usage