

Project Description

Task: Implement algorithms using Vitis HLS and Vivado on FPGA board

1. Background knowledge

1.1 Fundamentals about FPGA (Field Programmable Gate Arrays)

- FPGA are circuits that are programmable on the field
- Components of FPGA
 - Flip-Flops (FF), small memory component able to store a bit, typically used as a fast register to store data
 - Look-Up Tables (LUT), small memories used to store truth tables and perform logic functions, typically used to perform operation
 - Digital Signal Processor (DSP), small processor able to quickly perform mathematical operation on streaming digital signals, typically used for multiplication and additions
 - Block RAM (BRAM), memory able to store data, can store a fair amount of data, but slow and with a limited number of ports limiting memory throughput

1.2 How do we use FPGA traditionally?

We write codes in Verilog or VHDL for blocks, through Vivado, transfer codes into FPGA board and verify whether it is identical between FPGA result and the original function. Traditionally, it is essential for us to rewrite codes if more specific restrictions are added.

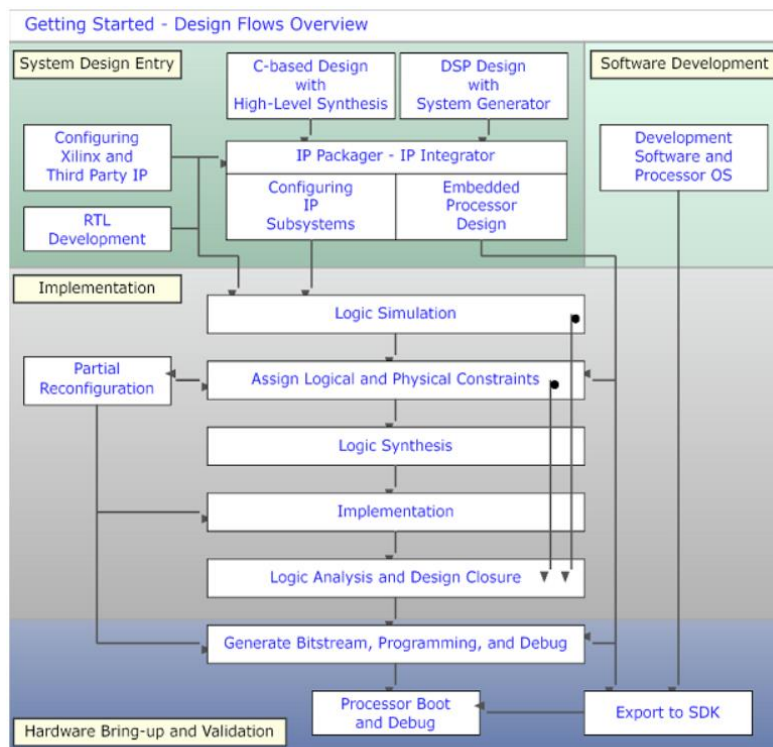


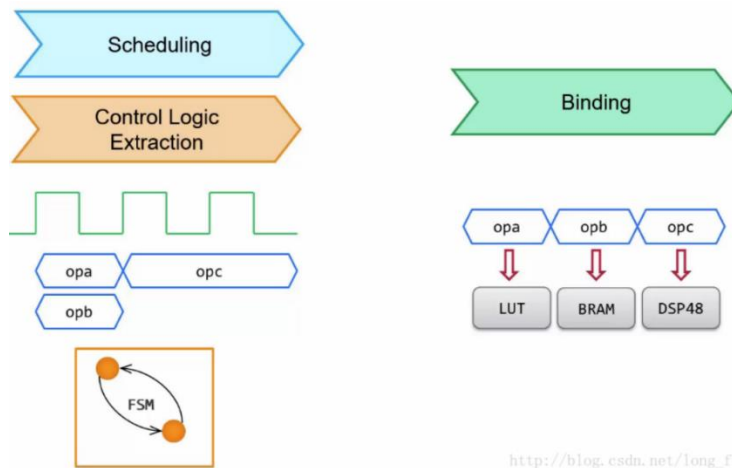
Figure 1-1: Vivado Design Suite High-Level Design Flow

It can be seen from the figure that there are two independent ways to implement codes on FPGA boards, one traditional way and another with HLS.

1.3 What is HLS, or more specifically Vitis HLS?

Vitis HLS is a high-level synthesis tool that allows C, C++, and OpenCL™ functions to become hardwired onto the device logic fabric and RAM/DSP blocks. Vitis HLS implements hardware kernels in the Vitis application acceleration development flow and uses C/C++ code for developing RTL IP core for designs in Vivado. Users can use Vitis HLS to develop FPGA modules using C/C++.

1.4 How does Vitis HLS work?

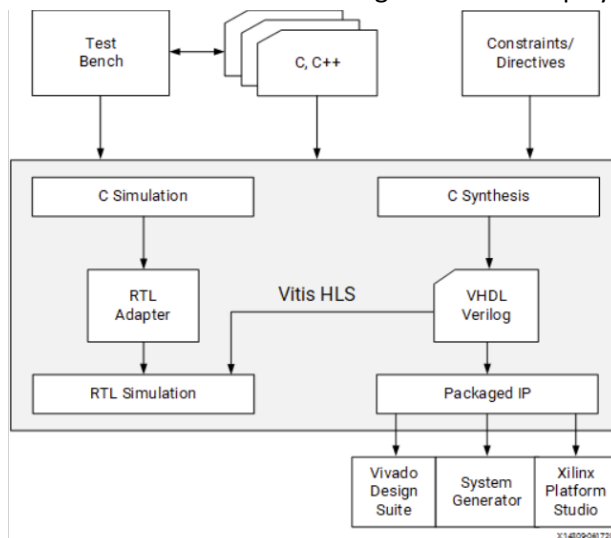


From the figure, we know that transforming codes in C++ or C to Verilog consists of two main steps, scheduling and binding.

Scheduling determines which operations to occur in each clock cycle.

Binding assigns hardware resources to implement each scheduled operation, and maps operators to specific RTL implementations.

More detailed manual of using Vitis HLS is displayed in the following figure.



1.5 How to examine codes' performance?

After C-synthesis, a detailed report is generated. Based on that, we can examine how the overall performance is.

Vitis HLS Report Comparison				
All Compared Solutions				
solution1: xcu200-fsgd2104-2-e				
solution2: xcu200-fsgd2104-2-e				
solution3: xcu200-fsgd2104-2-e				
Performance Estimates				
Timing				
Clock		solution1	solution2	solution3
ap_clk Target		8.00 ns	5.00 ns	10.00 ns
	Estimated	5.840 ns	3.650 ns	7.300 ns
Latency				
Latency (cycles)	min	648	648	648
	max	648	648	648
Latency (absolute)	min	5.184 us	3.240 us	6.480 us
	max	5.184 us	3.240 us	6.480 us
Interval (cycles)	min	649	649	649
	max	649	649	649
Utilization Estimates				
		solution1	solution2	solution3
BRAM_18K	34	34	34	
DSP	16	16	16	

Above is a typical report. Based on latency, we can infer how long the program and each loop last for. To be more intuitive, the larger latency, the longer time the algorithm needs. Based on II, we know whether reading in data is done at the highest efficiency after pipeline process of HLS. Ideally, II should be 1 so that reading input is executed in every cycle. Based on the numbers of LUT, FF and DSP, we know how many resources used in executing algorithms.

In further improvement, two aspects we are going to focus on is to balance between resources and time.

1.6 How to improve kernels' performance utilizing Vitis HLS?

There are two equivalent ways to improve performance: adding pragmas and setting directives.

- Pragmas are added to the source code to enable the optimization or change. Every time the code is synthesized, it is implemented in ways according to the specified pragmas.
- Optimization Directives, or the set_directive commands, can be specified as Tcl commands that are associated with a specific solution, or set of solutions, allowing you to customize the synthesis results for the same code base across different solutions.

According to Xilinx official guides, there are various pragmas including unrolling, partition, pipeline and etc. Adding directives or pragmas to loops enables users to lift algorithms' performance or occupy less resources in FPGA or CPU.

1.7 How to export HLS's result to FPGA board?

Since hardware board need I/O port initializations, it is essential for us to specify information like type and width of each I/O port through pragmas. The main interface of port is AXI4. The AXI4 interfaces supported by Vitis HLS include the AXI4-Stream interface (axis), AXI4-Lite (s_axilite), and AXI4 master (m_axi) interfaces.

According to official guides, there are different definition of input and output ports. With proper initialization of each port, stream can be transferred in a rather efficient way.

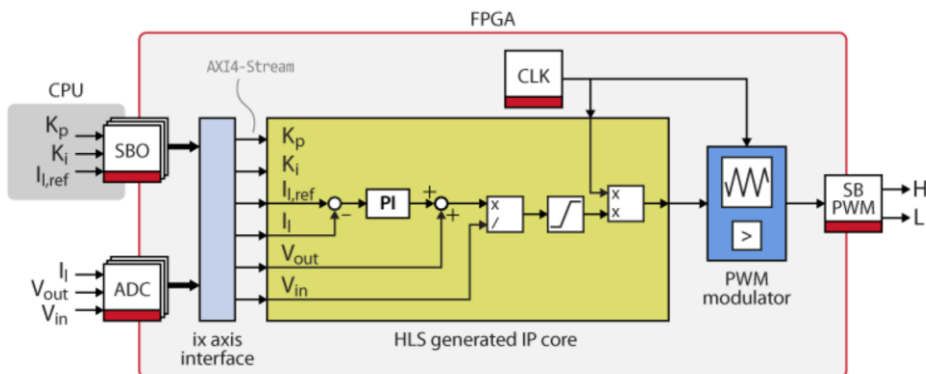
Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Input	Return	I	I/O	O	I	I/O	O	
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Supported
 D = Default Interface
 Not Supported

X14293

Figure 1-39: Data Type and Interface Synthesis Support

After complete steps of synthesis and simulation, the algorithms written in C++ can be verified to be executable. Then, it is the time to export it into hardware board to be further tested.



We use “exporting to RTL design” to generate IP core. We open it in Vivado as usual and create block design and bitstream and finally to FPGA.

2. Overall Process

2.1 Writing C++ codes

It is really a basic step and should end up with no more explanation. We write a function or algorithms using C++. For further test, it is essential to writes a simple test function as the main function.

2.2 Transforming C++ codes to IP core in Vitis HLS

2.2.1 C-synthesis

To synthesize the active solution of the project, select the Run C Synthesis command. When synthesis completes, the Simplified Synthesis report for the top-level function opens automatically. Performance can be quickly evaluated based on the metrics displayed in the report to determine if the design meets our requirements. Transformed Verilog files is also accessible in the “impl” folders and we can even use the Verilog code directly.

2.2.2 Co-simulation

If added a test bench to the project for simulation purposes, we can run C/RTL Co-simulation to verify whether the RTL is functionally identical to the C++ source code.

The test bench verifies output from the top-level function for synthesis, and returns zero to the main function of the test bench if the output is correct. Vitis HLS uses the same return value for both C simulation and C/RTL co-simulation to determine if the results are correct. If the C test bench returns a non-zero value, Vitis HLS reports that the simulation failed. The Co-simulation Report showing the pass or fail status and the measured statistics on latency and II.

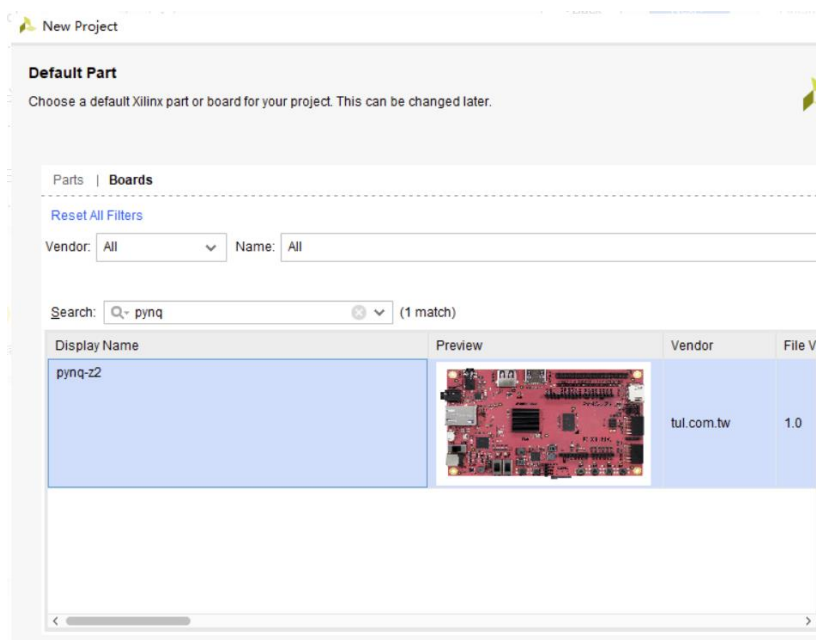
2.3 Using IP core in Vivado and transform it into FPGA boards

2.3.1 Generate executable bitstream using Vivado

The process is similar t the traditional way. The several main steps will be introduced in this part.

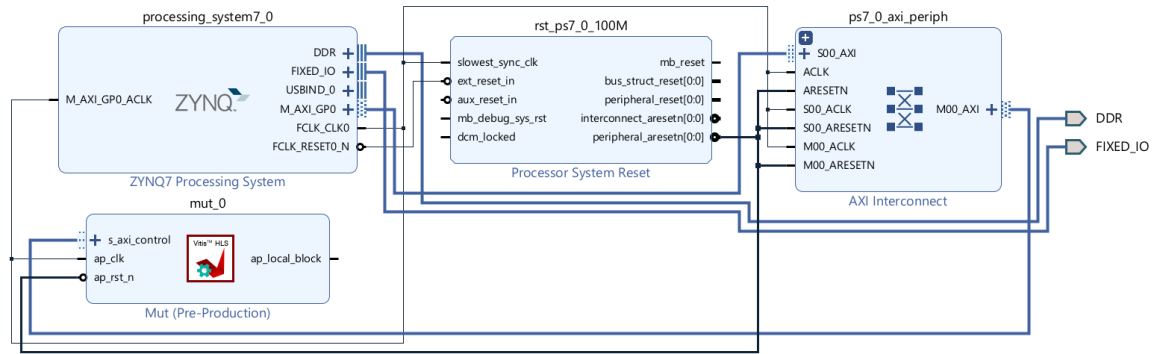
- Choose Boards

In this project, the board we used is Pynq-Z2.



- Create Block Design

Since we have already generated IP core, in this step we can directly import the IP core and it will be automatically displayed and connected in block design.



- Generate bitstream and export block design

After all these steps, it is the time to generate executable overlay so that we can test in the FPGA board. We need to generate bitstream and block wrapper. Finally, we just export them out and steps using Vivado is done.

2.3.2 Test IP using FPGA and Jupyter notebook

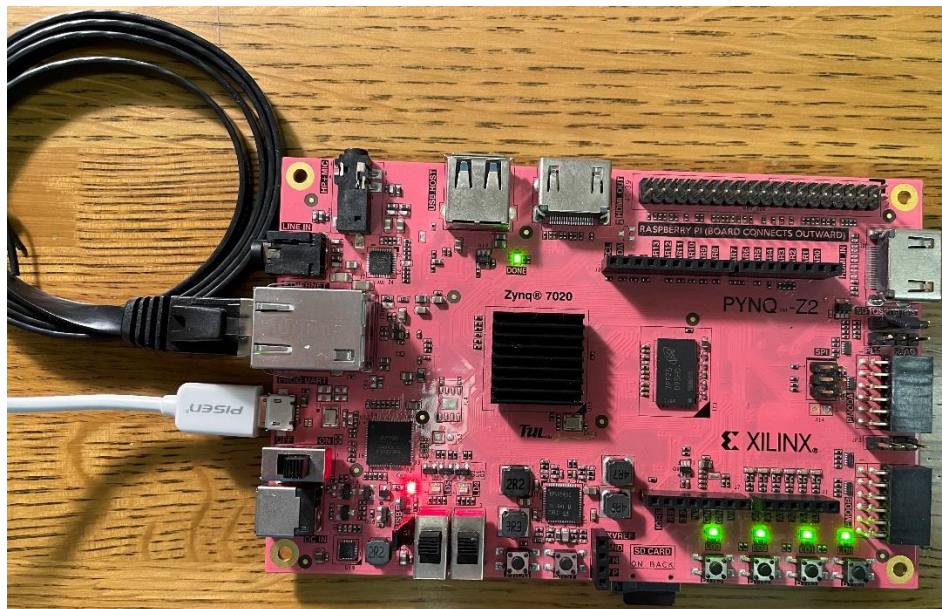
In this part, we need to set up and connect Pynq-Z2 with laptop and use Jupyter notebook to test.

2.3.2.1 Setup Pynq-Z2

A detailed guide can be referred to the official website.

https://pynq.readthedocs.io/en/v2.3/getting_started/pynq_z2_setup.html

The final connection diagram is displayed as follows.



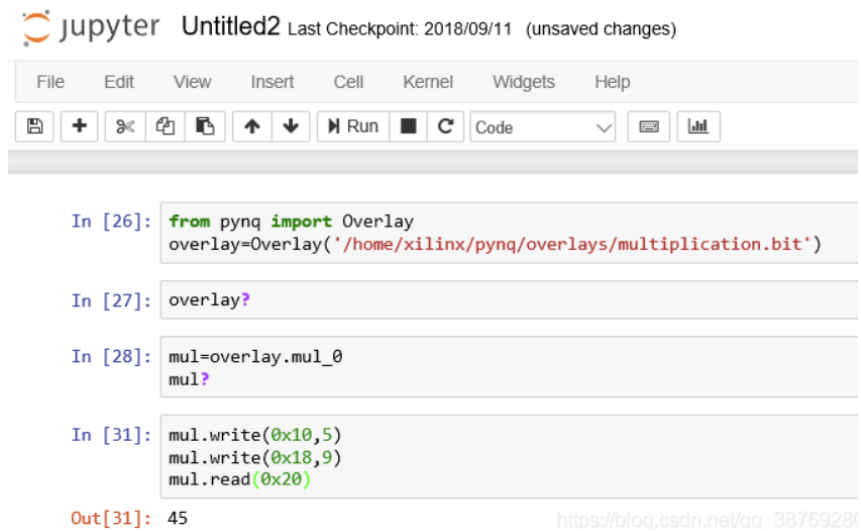
2.3.2.2 Test on Jupyter notebook

A detailed guide can be referred to the official website.

https://pynq.readthedocs.io/en/v2.3/getting_started.html#configuring-pynq

Following these steps, we can finally get the result as this, which works as a multiplication operator. During this process, what should be noticed is that there are

three files needed for the test with type of mut.bit, mut.hwh and mut.tcl. If there is any file which are not copied to the overlay folder under Xilinx Pynq folder, test will not succeed.



The image shows a Jupyter Notebook interface with the title 'Untitled2' and a last checkpoint of '2018/09/11 (unsaved changes)'. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, zooming, copying, pasting, undo, redo, and running code. The notebook contains four input cells and one output cell. The first input cell (In [26]) contains code to import the 'Overlay' class from the 'pynq' module and create an 'overlay' object pointing to a file at '/home/xilinx/pynq/overlays/multiplication.bit'. The second input cell (In [27]) contains the text 'overlay?'. The third input cell (In [28]) contains code to assign 'mul' to 'overlay.mul_0' and then 'mul?'. The fourth input cell (In [31]) contains code to write the values 5 and 9 to memory locations 0x10 and 0x18, and then read the value at 0x20. The output cell (Out[31]) shows the value 45. A URL is visible at the bottom right of the output cell.

```
In [26]: from pynq import Overlay
overlay=Overlay('/home/xilinx/pynq/overlays/multiplication.bit')
```

```
In [27]: overlay?
```

```
In [28]: mul=overlay.mul_0
mul?
```

```
In [31]: mul.write(0x10,5)
mul.write(0x18,9)
mul.read(0x20)
```

Out[31]: 45 https://blog.csdn.net/qq_38769280