A Fabric IP Netlist Generator for a Compiler-Approach to Fabric Integration

Arcel R. Salem[1], Erwil V. Pasia[2], and Allenn C. Lowaton[1]
[1] MSU-IIT/EECE, Iligan City, Philippines
[2] Lattice Semiconductor Corporation, Muntinlupa City, Philippines
Email: arcel.salem@g.msuiit.edu.ph; Erwil.Pasia@latticesemi.com; allenn.lowaton@g.msuiit.edu.ph

***Abstract*—Integrating fabric intellectual property (IP) netlists in field programmable gate array (FPGA) development can be very time-consuming especially if the design complexity and density increased significantly and the integration is done manually. Another issue in most integration techniques available is that it does not have a dedicated fabric IP netlists generator which makes the integration more tedious especially if there are changes and iterations needed in the design development like changing the design density. This paper introduces a fabric IP netlist generator that uses a compiler-approach to fabric integration. The compiler automatically connects the IP blocks and generates a fabric IP netlist depending on the user-input parameters. This tool provides the flexibility of changing the fabric IP array sizes and the IPs to be connected depending on the specifications required. It enables designers to design large densities of fabric IPs and also speeds up the FPGA development.**

***Index Terms*—compiler, CDL netlist, fabric IP, FPGA, integration, IP, Verilog netlist**

I. Introduction

Field-Programmable Gate Array (FPGA) is a type of integrated circuit which is designed to be programmed after manufacturing. Over the last decade, FPGAs have become one of the key digital circuit implementation media. The device's speed performance, area efficiency, and power consumption are remarkably shaped by FPGA's architecture. FPGA architecture is most commonly consist of an array of logic blocks, I/O pads, and routing channels [1]. These logic blocks are programmably interconnected to achieve certain desired functions or operations. These logic blocks are simply referred to as intellectual property (IP) core or IP blocks which are used as building blocks within an FPGA design. These IP blocks or IP cores are predesigned as well as pre-verified and are obtained from internal sources, or third parties and are combined on a single chip. FPGA core largely takes up the FPGA design which consist of the FPGA fabric [2].

FPGA's design complexity, in reality, is increasing more rapidly than the transistor count owing to the performance-enhancing features in FPGAs. With the increasing design complexity and density, the integration time also takes longer to complete. The integration process is not a simple operation. Selecting the optimal IP size, lower dynamic power, and faster speed is a critical design decision. Design engineers have to developed new methodologies and techniques in order to keep pace with the levels of integration. These techniques have to manage the increased complexity inherent to these large chips [3], [4].

Most integration techniques used in companies are script-based automation which involves manually modifying these scripts for changes and iterations. Lattice Semiconductor Company, for example, uses integration methodology which involves modifying the many script-based integration manually for the follow-on generation and port connections. Follow-on generation means new density for the same project or application [5].

One of the problems identified in the current integration techniques is that the designer must have a firm idea of the FPGA design floorplan before the integration takes place. Since the integration depends largely on the FPGA design floorplan, this technique is not desirable especially if the design abruptly changes to meet the desired specification. Similarly, the approach of compiler design used by the memory compiler can addressed the solution to this problem. Memory compilers can automatically instantiate the appropriate memory function based on the options the user choses.

Current studies in the area of increasing FPGA densities discussed only on integrating each physical IP blocks on the final chip [6], [7]. These studies do not focus on integrating the basic of IP design which is the netlists of each IP blocks. Netlist integration is as important as integrating the physical representation of each IP block since the integrated netlists will be the basis of how the final integration of IPs on the chip will be executed [8]-[10].

Harper J. *et al*. (2017) [11] in his paper stated that generating a file of the stitched system-on-chip (SOC) block specifies the characteristics and specifications of the final SOC design. The approach involves a user specified request to automatically stitch the SOC. The drawback, however, of this approach is that the user cannot have the freedom to choose which IP blocks will be connected and also the density of these IP blocks.

Manuscript received February 4, 2018; revised December 25, 2018; accepted December 25, 2018.
Corresponding author: Arcel R. Salem (email: arcel.salem@g.msuiit.edu.ph).

doi: 10.18178/ijeetc.8.3.181-187

The current available design compilers in the industry caters mostly to memory design mainly because they have regular layout structures, are highly repetitive, and are easily scalable. Since fabric IPs are also highly repetitive with regular layout structures, the same concept on design-automation method will be applied [12], [13].

II. Background

The FPGA core is made up of combinations of FPGA fabric consisting of repetitive IP blocks which serve as the key building blocks of FPGA fabric. These key building blocks shown in Fig. 1 are the Programmable Logic Cells (PLCs) and Common Interface Blocks (CIBs) [2].

Internally, fabric IP integration process is common to most industries which can be script-based automation or manually done. In Lattice' current process, for example, fabric IP netlist integration is done manually involving several script-based automations for integration for each IP blocks as reflected in Fig. 2.

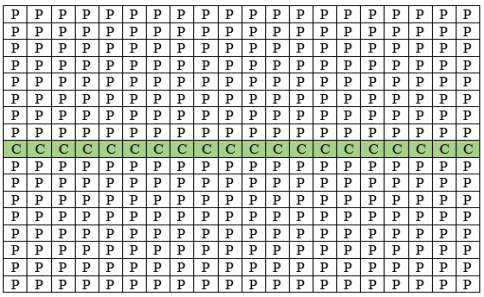

Fig. 1. FPGA fabric IPs.

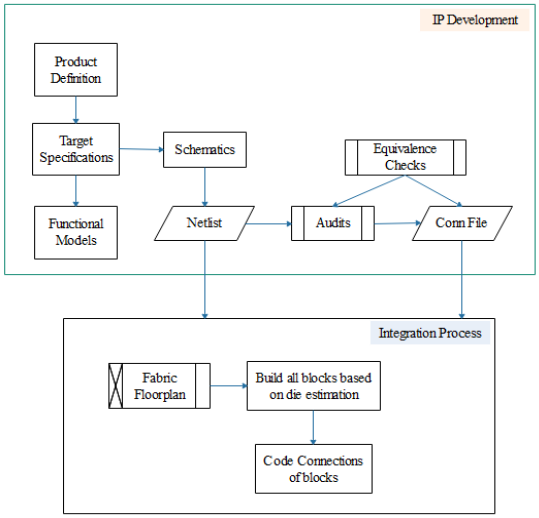

Fig. 2. Current netlist integration flow.

In the current process, the integration team will require the gate-level netlist of each IP block, then integrates it based on the designed specification and floorplan or the estimated die size. For example, the designer of the Programmable Logic Controller (PLC) block will submit the gate-level netlist of the verified IP to the integration team and it will then process the netlist to achieve the designed specification of the IP array such as the multipliers and size of the array. The integration of the IPs are done manually and separately per IP which involves many scripts to be modified for the block connections [3].

During follow-on generation, defined as creating the same FPGA design with different density requirements, the scripts that were used to integrate each IP will be modified to achieve the density requirement.

With the current fabric IP integration flow, several problems were identified that leads to this study. One of the major problem identified is the time-consuming integration of FPGA fabric IP netlists. The integration of FPGA fabric IP netlists takes several days to weeks to be completed simply because the scripts were manually modified when there are several iterations made for IPs correct functionality. Another problem identified is that before the fabric IP netlists can be integrated, the FPGA design density must be firmly defined in order to have an idea where the IPs will be placed and connected to other IPs. This methodology is not desirable especially if the design density abruptly changes or if new projects with different density will follow or the so-called follow-on generation.

To address these problems, memory compiler strategy will be leveraged which automatically instantiates the appropriate memory function based on the options chosen by the user. Since fabric IP blocks are highly repetitive and have regular layout structures, a new methodology that leverages the compiler approach strategy is suitable to overcome the problem of the current process. The principle of connecting these IP blocks by abutment is also one of the basis of this study.

III. Methodology

A. Proposed Fabric IP Netlist Generation Flow

The proposed fabric IP netlist generation leverages the compiler-approach strategy.

The proposed methodology eliminates the manual modification of the scripts used for the integration of fabric IPs. It also eliminates the design density estimation since the user has the capability of changing the IP array size automatically. The output deliverables will be used by the PnR (place & route) as input for automatic placement of and integration of the fabric IPs. The proposed methodology is reflected in Fig. 3.

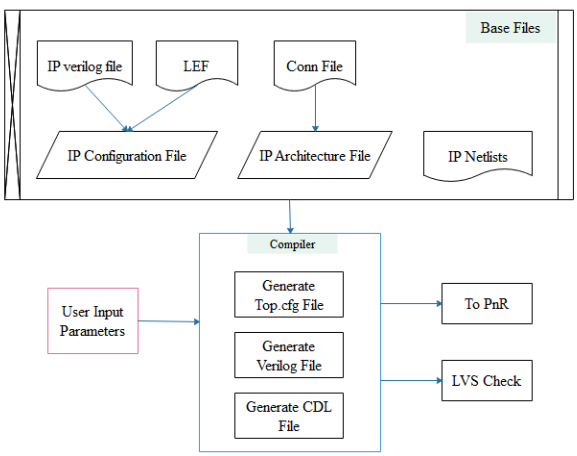

Fig. 3. Proposed fabric IP netlist generation.

The user input parameters consist of the number of rows and columns which determine the PLC block array size to be generated. Since the PLC blocks dictate the height and width of the neighboring blocks in the chip, it will be used as the ratio reference to other IPs. The user input parameters also include the location of CIB block to be connected to the PLC array and other IPs to be connected. CIB blocks are always placed at the bottom of the PLC array and at the left and right most of the core.

The whole fabric IP netlist generation proposed in this study begins through the establishment of the IP database consisting of the XML files namely, the configuration and architecture files. The configuration file defines the IP's port names, directions as well as the locations of each IP port in the physical representation. The architecture file, on the other hand, defines the connections of each IP port describing which port will be connected to which port. The architecture file contains the key information on how the IPs will be connected once the integration process takes place. These files serve as the database for the IPs' connectivity.

After all the input parameters and base files are processed, the Verilog gate-level netlist file will be generated including the wrapper. The Verilog file will be converted to CDL (circuit description language) netlist file using v2cdl tool. The CDL file will then be released for the LVS (layout versus schematic) check. This file format is the acceptable format for LVS check which is the reason the generated Verilog file must be converted.

B. Establishment of IP Configuration and Architecture Files

XML files from the IP database will be read by the tool to identify the IP ports and its property settings. These predefined internal files are first established before the tool operation starts. The IP configuration files are predefined files which are obtained according to each IP. The information contained by IP configuration file is established from IP Verilog file and LEF file. IP Verilog file contains the IPs port names and directions. The LEF file, on the other hand, contains the locations as well as the direction of the IP ports in the physical layout. This file reflects the exact location of the IP ports thus, the port's location will be determined, whether on the left, right, top, or bottom part of the block. Some ports are located both left and right or top and bottom part of the IP block, in this case, the configuration file will capture it as "right/left" or "top/bot".

The same process/methodology will be used on parsing the information contained by the IP architecture file. The information is based on the "Conn File" or connectivity file obtained from the IP development flow. This file contains the IP port names and how these ports will be connected to which other IP ports.

C. Generation of Top Configuration File

The top configuration file is an intuitive way to capture how the IPs will be connected. This file is generated after the user input parameters are processed. After determining the IP array size, the "top.cfg" file will be generated. This file shows the configuration of the array which consists of the integrated IPs and the array number. A script-based automation is responsible for the generation of this file. If the user includes the CIB block in the connection or additional IPs in the connection, the integrated IPs will be reflected in the "top.cfg" file. CIBs are always placed at the bottom of PLC array and at the left and right most of the core. The additional IPs to be connected will be placed after the CIB connection since CIBs serves as the physical interface between PLC and other IP blocks. This file also helps the user to easily check if the generated array is accurate especially if the array involved is large.

D. Verilog and CDL Netlist Generation

Verilog netlists generation of the top module of the integrated IP blocks is the main focus of the netlist generator tool. The operation starts with the tool reading the internal input files of the IP blocks. The main Verilog netlist generation includes three stages namely: (1) building wires between ports, (2) building instances of each ports, and (3) building the wrapper of the integrated netlist.

Connectivity of each ports which are reflected in the architecture file will be the bases of building the wires between the ports.

The wire direction can be horizontal or vertical which can drive either east, west, north, or south. Fig. 4 shows the generation of the Verilog netlist.

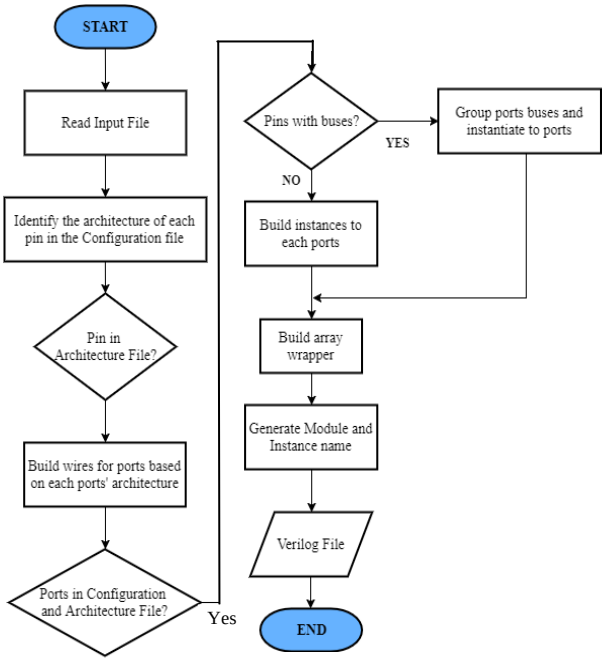

Fig. 4. Verilog netlist generation.

Some of the ports in an IP block consists of several bits, thus instantiating them can be tedious if not grouped. The netlist generator will group the corresponding port buses and instantiate them as one. Each port in the IP configuration file will be instantiated to corresponding ports based on the connectivity of the ports reflected in the architecture file. The top module wrapper will then be built after the wires and instantiations are finished.

Output Verilog netlist will be converted to equivalent

CDL netlist format. This netlist will be the main output of the tool since it will be used for the LVS check. LVS check determines whether a particular integrated circuit layout corresponds to the original schematic or circuit diagram of a design. The CDL format will be generated through the use of the Cadence tool called "v2cdl" which converts the Verilog file to CDL file. During the conversion, the IP netlists of each IP block will be included on the final CDL file. The IP netlists are obtained from the development process of each IP block. These files contain the internal information of each IP block such as the length, width, and multiples of the transistors used.

Also, the said file contains the technology used in the circuit design. All information from the design stage of the schematic of each IP block is captured in the IP netlist file. Fig. 5 shows the generation of the equivalent CDL netlist.

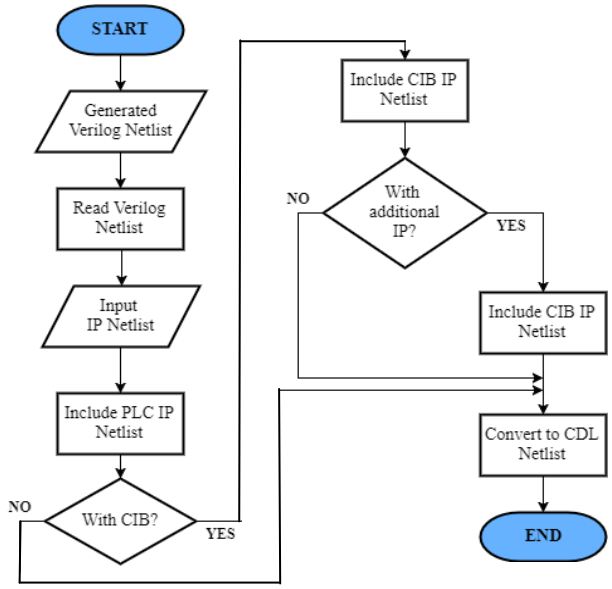

Fig. 5. CDL netlist generation.

IV. Results

IP database starts with the establishment of IP configuration and architecture files. The configuration file contains the IP's port name, direction, and location. The architecture file, on the other hand, contains the connection of each IP port which is based from the conn file obtained during the IP development process. The architecture file contains the key information on how the IP blocks will be connected. Sample content of these files are shown in Fig. 6.

User-input parameter includes the size of the IP array which is determined by the number of rows and columns. The tool's usage will be reflected once the script runs which acts as a GUI for the user. Fig. 7 shows the user guide that will appear once the tool is started. "-r" and "-c" represents the rows and columns to be inputted, "-d" represents the name of the module, "--cib" will determine the location of the CIB block to be connected to the PLC array, and "--ip" determines the IP block that will be added to the existing array combination.

```
<?xml version="1.0"?>
<ip name="plc">
   <ports>
      <port name="addr[2]" direction="input" location="top|bot"/>
      <port name="addr[1]" direction="input" location="top|bot"/>
      <port name="addr[0]" direction="input" location="top|bot"/>
      <port name="h02e0002" direction="input" location="left"/>
      <port name="h02w0102" direction="input" location="right"/>
      <port name="h06e0203" direction="input" location="left"/>
      <port name="prev_lut7" direction="input" location="left"/>
      <port name="v02n0002" direction="input" location="bot"/>
      <port name="v02s0402" direction="input" location="top"/>
      <port name="v06n0106" direction="input" location="bot"/>
      <port name="data[1]" direction="inout" location="left|right"/>
      <port name="data[0]" direction="inout" location="left|right"/>
      <port name="datan[1]" direction="inout" location="left|right"/>
      <port name="datan[0]" direction="inout" location="left|right"/>
      <port name="h02e0000" direction="output" location="right"/>
      <port name="h02w0100" direction="output" location="left"/>
      <port name="h06e0200" direction="output" location="right"/>
      <port name="next_lut7" direction="output" location="right"/>
      <port name="v02n0000" direction="output" location="top"/>
      <port name="v02s0400" direction="output" location="bot"/>
      <port name="v06n0100" direction="output" location="top"/>
   </ports>
</ip>
```

(a)

```
<?xml version="1.0"?>
<ip name="cib">
   <ports>
      <port name="addr[2]" direction="input" location="top|bot"/>
      <port name="addr[1]" direction="input" location="top|bot"/>
      <port name="addr[0]" direction="input" location="top|bot"/>
      <port name="fci" direction="input" location="left"/>
      <port name="h02e0202" direction="input" location="left"/>
      <port name="h02w0301" direction="input" location="right"/>
      <port name="v02n0502" direction="input" location="bot"/>
      <port name="v06n0006" direction="input" location="top"/>
      <port name="data[1]" direction="inout" location="left|right"/>
      <port name="data[0]" direction="inout" location="left|right"/>
      <port name="datan[1]" direction="inout" location="left|right"/>
      <port name="datan[0]" direction="inout" location="left|right"/>
      <port name="fcout" direction="output" location="right"/>
      <port name="h02e0200" direction="output" location="right"/>
      <port name="h02w0300" direction="output" location="left"/>
      <port name="v02n0500" direction="output" location="top"/>
      <port name="v06n0000" direction="output" location="bot"/>
   </ports>
</ip>
```

(b)

```
<?xml version = "1.0"?>
<arch>
   <types>
      <type name="tile1">
         <blocks>
            <block name="IP"/>
         </blocks>
         <connections>
            <connection out="(addr[7])" in="(addr[7])" />
            <connection out="(addr[6])" in="(addr[6])" />
            <connection out="(addr[5])" in="(addr[5])" />
            <connection out="(addr[4])" in="(addr[4])" />
            <connection out="(addr[3])" in="(addr[3])" />
            <connection out="(addr[2])" in="(addr[2])" />
            <connection out="(addr[1])" in="(addr[1])" />
            <connection out="(addr[0])" in="(addr[0])" />
            <connection out="(fcout)" in="(fci)" />
            <connection out="(h02e0700)" in="(h02e0702)" />
            <connection out="(h06w0300)" in="(h06w0306)" />
            <connection out="(h02w0700)" in="(h02w0702)" />
            <connection out="(h06e0300)" in="(h06e0306)" />
            <connection out="(next_lut7)" in="(prev_lut7)" />
            <connection out="(v01n0100)" in="(v01n0101)" />
            <connection out="(v06n0200)" in="(v06n0206)" />
            <connection out="(v06s0100)" in="(v06s0103)" />
            <connection out="(v01s0000)" in="(v01s0001)" />
            <connection out="(data[1])" in="(data[1])" />
            <connection out="(data[0])" in="(data[0])" />
            <connection out="(datan[1])" in="(datan[1])" />
            <connection out="(datan[0])" in="(datan[0])" />
         </connections>
      </type>
   </types>
</arch>
```

(c)

Fig. 6. Configuration and architecture files: (a) PLC configuration file (b) CIB configuration file (c) IP top architecture file.

```
Top Netlist Builder for Compiler-approach to Fabric Integration
0.1
Usage: python netgen.py -r [int] -c [int] -d [device name] --cib [cib location] --ip [ip to be connected]...
```

Fig. 7. Tool user guide.

```
plc(r1c1) | plc(r1c2) | plc(r1c3) | plc(r1c4) | plc(r1c5) | plc(r1c6)
plc(r2c1) | plc(r2c2) | plc(r2c3) | plc(r2c4) | plc(r2c5) | plc(r2c6)
plc(r3c1) | plc(r3c2) | plc(r3c3) | plc(r3c4) | plc(r3c5) | plc(r3c6)
plc(r4c1) | plc(r4c2) | plc(r4c3) | plc(r4c4) | plc(r4c5) | plc(r4c6)
plc(r5c1) | plc(r5c2) | plc(r5c3) | plc(r5c4) | plc(r5c5) | plc(r5c6)
plc(r6c1) | plc(r6c2) | plc(r6c3) | plc(r6c4) | plc(r6c5) | plc(r6c6)
```

(a)

```
plc(r1c1) | plc(r1c2) | plc(r1c3) | plc(r1c4) | plc(r1c5) | plc(r1c6)
plc(r2c1) | plc(r2c2) | plc(r2c3) | plc(r2c4) | plc(r2c5) | plc(r2c6)
plc(r3c1) | plc(r3c2) | plc(r3c3) | plc(r3c4) | plc(r3c5) | plc(r3c6)
plc(r4c1) | plc(r4c2) | plc(r4c3) | plc(r4c4) | plc(r4c5) | plc(r4c6)
plc(r5c1) | plc(r5c2) | plc(r5c3) | plc(r5c4) | plc(r5c5) | plc(r5c6)
plc(r6c1) | plc(r6c2) | plc(r6c3) | plc(r6c4) | plc(r6c5) | plc(r6c6)
cib(r7c1) | cib(r7c2) | cib(r7c3) | cib(r7c4) | cib(r7c5) | cib(r7c6)
```

(b)

```
plc(r1c1) | plc(r1c2) | plc(r1c3) | plc(r1c4) | plc(r1c5) | plc(r1c6)
plc(r2c1) | plc(r2c2) | plc(r2c3) | plc(r2c4) | plc(r2c5) | plc(r2c6)
plc(r3c1) | plc(r3c2) | plc(r3c3) | plc(r3c4) | plc(r3c5) | plc(r3c6)
plc(r4c1) | plc(r4c2) | plc(r4c3) | plc(r4c4) | plc(r4c5) | plc(r4c6)
plc(r5c1) | plc(r5c2) | plc(r5c3) | plc(r5c4) | plc(r5c5) | plc(r5c6)
plc(r6c1) | plc(r6c2) | plc(r6c3) | plc(r6c4) | plc(r6c5) | plc(r6c6)
cib(r7c1) | cib(r7c2) | cib(r7c3) | cib(r7c4) | cib(r7c5) | cib(r7c6)
plc(r8c1) | plc(r8c2) | plc(r8c3) | plc(r8c4) | plc(r8c5) | plc(r8c6)
plc(r9c1) | plc(r9c2) | plc(r9c3) | plc(r9c4) | plc(r9c5) | plc(r9c6)
plc(r10c1) | plc(r10c2) | plc(r10c3) | plc(r10c4) | plc(r10c5) | plc(r10c6)
plc(r11c1) | plc(r11c2) | plc(r11c3) | plc(r11c4) | plc(r11c5) | plc(r11c6)
plc(r12c1) | plc(r12c2) | plc(r12c3) | plc(r12c4) | plc(r12c5) | plc(r12c6)
plc(r13c1) | plc(r13c2) | plc(r13c3) | plc(r13c4) | plc(r13c5) | plc(r13c6)
```

(c)

Fig. 8. Generated Top.cfg file for different test cases: (a) 6x6 PLC array (b) PLC-CIB array (c) PLC-CIB-PLC array.

```
module plc6x6 (h02e0701, h02e0201, v01s0101, v06n0103, v02n0702, v02n0601, v02s0501, h06w0203,
h02w0501, h06w0106, v01s0100, h02e0500, v06n0200, h02e0601, v02s0602, v02n0701, v01s0000, h02w0001,
v02s0002, h02e0101, v02s0700, h06w0103, v06s0306, v06n0003, h06w0303, fci, h02e0100, v01n0000,
v02s0400, h02w0401, h02e0002, h06e0106, v02s0701, v02n0101, v01n0100, h06w0000, h01w0001, h02e0600,
v06s0300, h02e0401, h02e0702, v02n0501, v02s0401, h02e0502, h02w0202, h06w0003, v02n0002, h02e0602,
h06e0203, v06n0300, v06n0006, h06w0006, h02w0500, h06w0300, h06w0100, v06s0206, v02n0202, v06s0106,
h02e0102, v01s0001, v02n0302, h01e0000, h02e0402, v02s0000, h02w0302, h02w0601, h02w0700, v02s0601,
h02e0300, v02n0700, v02s0200, v02n0401, addr, v06s0203, v06n0000, v06s0103, h02e0400, v02s0402,
h06e0306, h06e0303, v02s0202, v02s0502, h02w0702, h06e0103, v02n0502, h02e0301, h02w0100, h01e0001,
fcout, h02w0402, v06s0303, v02s0300, v02s0100, h02w0600, datan, v02n0100, h02e0001, h06w0200,
h06e0200, h01w0101, h02w0002, v02s0500, h01e0100, v02n0102, v02n0201, h06e0206, h02w0300,
h02e0000, v06n0100, next_lut7, prev_lut7, h02w0301, v02s0001, h02w0602, v06s0003, h06e0006, h02e0202,
h06e0300, v06s0100, h02w0502, data, v01n0001, v06n0306, v06s0000, h06e0000, v06n0203, h01w0100,
h02e0200, v02s0101, v02s0301, v02n0200, h06e0100, v02n0600, v02n0500, h02w0400, v02n0400, h06e0003,
h06w0306, v02n0402, v02n0301, v02s0102, v02s0702, h06w0206, h01w0000, h02e0700, h02w0000, h01e0101,
h02w0201, v02n0001, v01n0101, en_pupn, h02w0200, v02s0201, v02s0600, v02n0000, v02s0302, v06s0200,
h02w0101, h02w0701, h02w0102, h02e0302, h02e0501, v06n0206, v06n0106, v02n0602, v06n0303, v06s0006);
```

(a)

```
PLC Xr1c1  (.h01w0101(r1c2_h01w0100), .h02w0202(r1c3_h02w0200), .h06w0203(r1c4_h06w0200),
.v02s0201(r1c1_v02n0200), .v02s0500(r1c1_v02s0500), .h02e0502(r1c2_h02w0500),
  .h02e0202(r1c2_h02w0200), .h02w0301(r1c2_h02w0300), .v02s0001(r1c1_v02n0000),
  .h02w0200(r1c1_h02w0200), .v02s0701(r1c1_v02n0700), .v06s0303(r3c1_v06n0300),
  .h06w0000(r1c1_h06w0000), .v01n0000(r1c1_v01n0000), .v02n0002(r3c1_v02n0000),
  .v02s0101(r1c1_v02n0100), .h06w0200(r1c1_h06w0200), .fcout(r1c1_fcout),
  .h02w0500(r1c1_h02w0500), .v01s0100(r1c1_v01s0100), .v06s0106(r6c1_v06n0100),
  .v06n0103(r4c1_v06n0100), .h02e0001(r1c1_h02w0000), .h02w0700(r1c1_h02w0700),
  .h02e0401(r1c1_h02w0400), .v06s0003(r3c1_v06n0000), .h06e0300(r1c1_h06e0300),
  .v02n0201(r2c1_v02n0200), .h06w0306(r1c6_h06e0300), .h06e0100(r1c1_h06e0100),
  .h02w0201(r1c2_h02w0200), .v06n0206(r6c1_v06s0200), .v02s0100(r1c1_v02s0100),
  .v01s0001(r1c1_v01n0000), .h06w0100(r1c1_h06w0100), .v06s0300(r1c1_v06s0300),
  .h02e0200(r1c1_h02e0200), .v02n0000(r1c1_v02n0000), .h06w0003(r1c4_h06w0000),
  .addr({addr[3], addr[2], addr[1], addr[0]}),
  .datan({datan[3], datan[2], datan[1], datan[0]}),
  .data({data[3], data[2], data[1], data[0]}));
```

(b)

Fig. 9. Output Verilog netlist file: (a) Verilog wrapper (b) instantiated IP ports.

```
$ Cdl netlist generated by v2cdl
$ Tool:      v2cdl 13.1.0-p286, sub-version p286
$ CmdLine: /tools/dist/cadence/PVS/latest/Linux/tools.lnx86/pvs/bin/64bit/v2cdl
            -v plc6x6.v -w 1 -o plc6x6.cdl

.SUBCKT macgyver h02e0701 h02e0201 v01s0101 v06n0103 v02n0702 v02n0601 v02s0501
+ h02e0601 v02s0602 v02n0701 v01s0000 h02w0001 v02s0002 h02e0101 v02s0700
+ h06w0103 v06s0306 v06n0003 h06w0303 fci h02e0100 v01n0000 v02s0400 h02w0401
+ h06w0003 v02n0002 h02e0602 h06e0203 v06n0300 v06n0006 h06w0006 h02w0500
+ h02e0300 hpbx1200 v02n0700 v02s0200 v02n0401 addr<7> addr<6>
+ addr<5> addr<4> addr<3> addr<2> addr<1> addr<0> v06s0203 v06n0000 v06s0103
+ h02e0400 v02s0402 h06e0306 h06e0303 v02s0202 v02s0502 h02w0702 h06e0103
+ v02n0502 h02e0301 h02w0100 hpbx1400 hpbx0000 h01e0001 fcout h02w0402 hpbx0700
+ v06s0303 v02s0300 v02s0100 h02w0600 datan<11> datan<10> datan<9> datan<8>
+ datan<7> datan<6> datan<5> datan<4> datan<3> datan<2> datan<1> datan<0>
+ v02n0100 h02e0001 h06w0200 hpbx0200 h06e0200 done_gwe h01w0101 hpbx1000
+ h02w0002 hpbx0400 v02s0500 h01e0100 hpbx1500 v02n0102 v02n0201 h06e0206
+ h02w0300 h02e0000 v06n0100 next_lut7 prev_lut7 h02w0301 v02s0001 h02w0602
+ v06s0003 h06e0006 h02e0202 hpbx0300 en_pupnb h06e0300 v06s0100 h02w0502
+ data<11> data<10> data<9> data<8> data<7> data<6> data<5> data<4> data<3>
+ data<2> data<1> data<0> v01n0001 v06n0306 v06s0000 h06e0000 v06n0203 h01w0100
+ h02e0200 v02s0101 v02s0301 v02n0200 h06e0100 v02n0600 gsrn v02n0500 h02w0400
+ v02n0400 h06e0003 h06w0306 v02n0402 v02n0301 v02s0102 v02s0702 h06w0206
+ h01w0000 hpbx1100 hpbx0600 hpbx0100 h02e0700 h02w0000 h01e0101 h02w0201
+ v02n0001 v01n0101 en_pupn h02w0200 v02s0201 v02s0600 v02n0000 v02s0302

XXr1c1 PLC  $PINS v06s0006=r1c1_v06n0000 h02e0201=r1c1_h02w0200
+ v01s0100=r1c1_v01s0100 v06n0200=r1c1_v06n0200 h02e0601=r1c1_h02w0600
+ v02s0602=r2c1_v02n0600 v02n0701=r2c1_v02n0700 v01s0000=r1c1_v01s0000
+ h02w0001=r1c2_h02w0000 v02s0002=r2c1_v02n0000 h02e0101=r1c1_h02w0100
+ v02s0700=r1c1_v02s0700 h06w0103=r1c1_h06e0100 v06s0306=r1c1_v06n0300
+ v06n0003=r3c1_v06s0000 h06w0303=r1c1_h06e0300 fci=r1c1_fcout
+ h02e0100=r1c1_h02e0100 v01n0000=r1c1_v01n0000 v02s0400=r1c1_v02s0400
```

Fig. 10. Output CDL netlist file.

After the user inputs the required parameters, the tool will then generate, as reflected in Fig. 8, the "top.cfg" file which intuitively captures the connection of the IP blocks. Verilog netlist file and the CDL file as reflected in Fig. 9 will also be generated.

Fig. 10 reflects the generated equivalent netlist. The CDL file will be delivered for the LVS checking.

```
################################################################################
#####
#####                                PVS LVS COMPARISON
#####
#####   Version                    :  14.14-s507
#####   NVN Run Start              :  Thu Nov  2 13:17:21  2017
#####   Extraction Report File     :   plc6x6.lvs
#####   Comparison Report File     :   plc6x6.lvs.cls
#####   Top Cell                   :  plc6x6  <vs>  plc6x6
#####
#####   Run Result                 :      MATCH
#####
#####   Run Summary                :  [INFO]  Extraction Clean
#####
#####   Layout Design              :  /$pathname/plc6x6.gds  (GDSII)
#####   Schematic                  :  /$pathname/plc6x6.cdl    (cdl)
#####   Rules File                 :  rules.lvs
#####   Pin Swap File              :  plc6x6.lvs.cps
#####
#####   Extraction CPU Time        :  0h 8m 43s - (523s)
#####   Extraction Exec Time       :  0h 8m 52s - (532s)
#####   Extraction Peak Memory Usage :  1431.00MB
#####   NVN CPU Time               :  0h 0m 5s - (5s)
#####   NVN Exec Time              :  0h 0m 7s - (7s)
#####   NVN Peak Memory Usage      :  474.07MB
#####   LVS Total CPU Time         :  0h 8m 48s - (528s)
#####   LVS Total Exec Time        :  0h 8m 59s - (539s)
#####   LVS Total Peak Memory Usage  :  1431.00MB
#####
################################################################################
```

Fig. 11. LVS Result.

TABLE I: COMPARISON OF CURRENT AND PROPOSED METHODOLOGIES

Comparison	Current integration	Compiler-approach
Fabric Density	Fixed Floorplan	Flexible
Development process	Manually integrate IPs	Automatically integrate IPs
Macro development Time	24 hours	0.5 hours
Dedicated fabric generator	No	Yes
Limitations on IP size	Yes	No
Follow-on generation	Code modifications	Base files modifications

The generated output Verilog netlist file will be used as input for the next process of FPGA development which will automatically place and integrate the physical representation of the IP blocks. A script will be used to automatically place the IP blocks. If the script will result to an error, the input file delivered, which is the generated Verilog file, contains an error. This process also serves as verification to the integrity of the generated output netlist. If the script returns no error, the generated Verilog file is tested accurate. Another test performed to validate the accuracy of the generated output Verilog and CDL netlists is the LVS check. The output netlist of the integrated PLC array will be compared to the output file of the physical layout of the integrated PLC array. The result of the LVS check is shown in Fig. 11.

Table I shows the comparison of the current to the proposed methodology. The development time reflected in Table I refers to PLC array. One of the advantages of the proposed methodology is its compiler capability which enables the user to flexibly change the size of the IP array based on the design specification density. The current process, on the other hand, based its density on the fixed design floorplan which, in case of design iterations, cannot automatically change the IP array size.

The proposed methodology is a decently dedicated fabric Generator while the current methodology does not have a dedicated fabric Generator since the integration is done manually per IP.

During design changes and iterations, the current process' integration code will be manually modified which consists of hundreds of lines and the characteristics to be changed will be manually looked for. The proposed process, on the other hand, will only change the base files base on the design specifications required.

With the introduction of the compiler-approach fabric IP netlist generator developed in this study, the time needed for the development and integration of fabric IP netlist as summarized in Table II and shown in Fig. 12 was greatly reduced. The overall time it takes for the integration of PLC array to finish takes only minutes compared to a day of integration using the current process. The integration time was greatly reduced by almost 97.92%. For the PLC-CIB array, the integration time was reduced by 97.20%. For the PLC-CIB-PLC array, the integration time was reduced by 96.53%.

TABLE II: SUMMARY OF FABRIC MACRO DEVELOPMENT

Fabric macro development (time: day)			
Array	Current integration	Compiler-approach	Percent (%) reduced
PLC	1	0.0208333	97.92%
PLC-CIB	1.5	0.042	97.20%
PLC-CIB-PLC	1.8	0.0625	96.53%

V. CONCLUSIONS

The methodology presented in this paper have been used and tested on a 40nm technology project of Lattice Semiconductor. In which the current integration process was also used.

The researcher has concluded that the proposed dedicated fabric IP netlist generation flow with the generation of configuration and top architecture file can deliver the accurate Verilog and CDL files of the integrated IP blocks. It also has greatly improved the speed of fabric IP netlist integration and the flexibility of the FPGA design. The improvement on speed of integration will enable the designers to complete the project sooner than the scheduled time of completion.

Using a lesser number of base files to be processed makes the integration faster since the tool will process fewer files. Also, using these base files will eliminate the need to manually modify the scripts every time new changes and iterations will follow. Moreover, the tool will automatically generate the connections of each IPs simultaneously which reduced its runtime to a significantly. The user also has the ability to change the size of the IP array depending on the desired design density and to follow-ons. In addition to this, the user can change the IPs to be connected to the existing array and the location of the IPs.

Moreover, the improvement on the speed of integration will enable the other section of the FPGA development to adjust the timeline on the verification and audit checks.

ACKNOWLEDGMENT

The authors would like to thank the DOST-ERDT program for the fund support in conducting this study; USAID-STRIDE and DOST-PCIEERD for the IC design tools. Also, the Lattice Semiconductor Inc. Philippines for the opportunity to work on this paper. Also to the IP Productization, Methodology and Quality team and Integration team for the guidance while conducting this study.

REFERENCES

[1] R. Saleh, S. Wilton, S. Mirabbasi, *et al*., "System-on-chip: Reuse and integration," *Proc. of the IEEE*, vol. 94, no. 6, pp. 1050-1069, 2006.

[2] *FPGA Design Guide*, (Version 7.2), Lattice Semiconductor Corporation, 2008.

[3] R. Rajsuman, *System-on-Chip Design and Test*, Norwood, USA: Artech House, Inc., 2000.

[4] L. Yang, S. Gurumani, D. Chen, and K. Rupnow, "Behavioral-level IP integration in high-level synthesis," in *Proc. Int. Conf. on Field Programmable Technology (ICFPT2015)*, 2015.

[5] "*Ecp5 product family qualification summary*," Lattice Semiconductor Corporation, August 2016.

[6] I. Kuon, A. Egier, and J. Rose, "Design, layout and verification of an FPGA using automated tools," in *Proc. ACM/SIGDA 13th Int. Symposium on the Field-Programmable Gate Arrays*, February 2005, pp. 215-226.

[7] S. Wu, X. Zheng, Z. Gao, and X. He, "A 65nm embedded low power SRAM compiler," in *Proc. 13th Int. Symposium on Circuits and Systems (ISCAS2007)*, April 2010, pp. 3756-3759.

[8] R. Goldman, *et al*., "Synopsys' educational generic memory compiler," in *Proc. 10th European Workshop on Microelectronics Education*, 2014, pp. 89-92.

[9] P. Coussy, A. Baganne, and E. Martin, "A design methodology for integrating IP into SoC systems," in *Proc. IEEE 2002 Custom Integrated Circuits Conf.* 2002, pp. 307-310.

[10] M. S. Kim, C. G. Kim, S. D. Kim, and J. L. Gaudiot, "Design of configurable I/O Pin control block for improving reusability in multimedia SoC platforms," *Multimedia Tools and Applications*, vol. 74, no. 20, pp. 9055-9066, 2013.

[11] J. Harper, *et al*., "Automating system on chip customized design integration, specification, and verification through a single integrated service," U.S. Patent Application Publication US 20170116355A1, April 27, 2017.

[12] M. Cheng and N. Bai, "An efficient and flexible embedded memory IP compiler," in *Proc. Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2012, pp. 268-273.

[13] M. R. Guthaus, J. E. Stine, S. Ataei, *et al*., "OpenRAM: An open-source memory compiler," in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2016, pp. 1-6.

Arcel R. Salem received her bachelor's degree in Electronics Engineering (BSECE) at Mindanao State University-Main Campus, Marawi City, Philippines, in 2011. She is currently in her 2nd year of Master's degree in Electrical Engineering major in Microelectronics in Mindanao State University-Iligan Institute of Technology, Iligan City, Philippines. She had her internship for her master's thesis at Lattice Semiconductor Philippines. Her research interests includes, RTL design, FPGA design, IC design and layout.

Erwil V. Pasia received his bachelor's degrees in Physics and in Computer Science (BSComSci) at Ateneo de Manila University (AdMU), Manila City, Philippines, in 2004. He is currently managing IP Productization, Methods, and Quality in Lattice Semiconductor Inc., a team he pioneered in 2012. He previously work at Canon Information Technologies Philippines as design verification and design engineer. His work interests includes ASIC and FPGA Designs, generation of secondary views of IP designs, develop efficient processes for faster integration and promote IP reuse.

Prof. Allenn C. Lowaton received his bachelor's degree in Electronics Engineering (BSECE) at Mindanao State University-Iligan Institute of Technology, Iligan City (MSU-IIT), Philippines, in 2008 and graduated Cum Laude. He finished his Master's degree in Electrical Engineering major in Integrated Circuits Design at National Taipei University, New Taipei City, Taiwan ROC in 2012 with a GPA of 4/4.

He had worked as a product engineer in Taiyo Yuden Philippines, Mactan Eco-Processing Zone at Lapu-Lapu City, Cebu from December 2008 to August 2009. Presently, he is a full-time assistant professor IV at the Electrical, Electronics and Computer Engineering (EECE) Department and is currently the program head of Electronics Engineering at MSU-IIT.