CENICS 2019

The Twelfth International Conference on Advances in Circuits, Electronics and Micro-electronics

ISBN: 978-1-61208-748-1

October 27 - 31, 2019

Nice, France

CENICS 2019 Editors

Sandra Sendra, Universidad de Granada, Spain
Pascal Lorenz, University of Haute Alsace, France

CENICS 2019

Forward

The Twelfth International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS 2019), held between October 27, 2019 and October 31, 2019 in Nice, France, continued a series of events initiated in 2008, capturing the advances on special circuits, electronics, and micro-electronics on both theory and practice, from fabrication to applications using these special circuits and systems. The topics covered fundamentals of design and implementation, techniques for deployment in various applications, and advances in signal processing.

Innovations in special circuits, electronics and micro-electronics are the key support for a large spectrum of applications. The conference is focusing on several complementary aspects and targets the advances in each on it: signal processing and electronics for high speed processing, micro- and nano-electronics, special electronics for implantable and wearable devices, sensor related electronics focusing on low energy consumption, and special applications domains of telemedicine and ehealth, bio-systems, navigation systems, automotive systems, home-oriented electronics, bio-systems, etc. These applications led to special design and implementation techniques, reconfigurable and self-reconfigurable devices, and require particular methodologies to be integrated on already existing Internet-based communications and applications. Special care is required for particular devices intended to work directly with human body (implantable, wearable, ehealth), or in a human-close environment (telemedicine, house-oriented, navigation, automotive). The mini-size required by such devices confronted the scientists with special signal processing requirements.

We take here the opportunity to warmly thank all the members of the CENICS 2019 technical program committee, as well as all the reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and effort to contribute to CENICS 2019. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

We also thank the members of the CENICS 2019 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope that CENICS 2019 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the field of circuits, electronics and micro-electronics. We also hope that Nice, France provided a pleasant environment during the conference and everyone saved some time to enjoy the charm of the city.

CENICS 2019 Chairs

CENICS Steering Committee
Falk Salewski, Muenster University of Applied Sciences, Germany
Chun-Hsi Huang, University of Connecticut, USA
Diego Ettore Liberati, National Research Council of Italy, Italy
Julio Sahuquillo, Universitat Politècnica de València, Spain
Sergei Sawitzki, FH Wedel (University of Applied Sciences), Germany
Manuel José Cabral dos Santos Reis, University of Trás-os-Montes e Alto Douro, Portugal
Bartolomeo Montrucchio, Politecnico di Torino, Italy
Petr Hanáček, Brno University of Technology, Czech Republic

CENICS 2019
Committee

CENICS Steering Committee

Falk Salewski, Muenster University of Applied Sciences, Germany
Chun-Hsi Huang, University of Connecticut, USA
Diego Ettore Liberati, National Research Council of Italy, Italy
Julio Sahuquillo, Universitat Politècnica de València, Spain
Sergei Sawitzki, FH Wedel (University of Applied Sciences), Germany
Manuel José Cabral dos Santos Reis, University of Trás-os-Montes e Alto Douro, Portugal
Bartolomeo Montrucchio, Politecnico di Torino, Italy
Petr Hanáček, Brno University of Technology, Czech Republic

CENICS Research/Industry Committee

John Vardakas, Iquadrat Informatica, Barcelona, Spain
Laurent Fesquet, TIMA laboratory | Grenoble Institute of Technology, France
Christian Wögerer, PROFACTOR GmbH, Austria
Miroslav Velev, Aries Design Automation, USA
Ivo Stachiv, Institute of Physics | Czech Academy of Sciences, Prague, Czech Republic / Harbin Institute of Technology | Shenzhen Graduate School, Shenzhen, China
Amir Shah Abdul Aziz, TM Research & Development, Malaysia

CENICS 2019 Technical Program Committee

Francesco Aggogeri, University of Brescia, Italy
Adel Al-Jumaily, University of Technology, Sydney, Australia
Mohammad Amin Amiri, Malek Ashtar University of Technology, Islamic Republic of Iran
Patroklos Anagnostou, Leclanché SA, Switzerland
Nihar Athreyas, Spero Devices, Inc., USA
Amir Shah Abdul Aziz, TM Research & Development, Malaysia
Payman Behnam, University of Utah, USA
Fayçal Bensaali, Qatar University, Qatar
Vincent Beroulle, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), France
Timm Bostelmann, FH Wedel (University of Applied Sciences), Germany
Hamza Bouzeria, Constantine - 1- University, Algeria
Khalid Bouziane, Université Internationale de Rabat, Morocco
Luca Calderoni, University of Bologna, Italy
David Cordeau, XLIM UMR CNRS 7252, France
Nicola D'Ambrosio, Laboratori Nazionali del Gran Sasso (LNGS) – INFN, Italy
Jamal Deen, Academy of Science - Royal Society of Canada / McMaster University, Canada
Javier Diaz-Carmona, Technoligical Institute of Celaya, Mexico
Alie El-Din Mady, United Technologies Research Center, Cork, Ireland
Diego Ettore Liberati, National Research Council of Italy, Italy
Maher Fakih, OFFIS e.V. Institut für Informatik, Oldenburg, Germany
Francisco Falcone, ISC-UPNA, Spain

Table of Contents

VHDL Design Tool Flow for Portable FPGA Implementation

Vijaykumar Guddad, Alexandra Kourfali and Dirk Stroobandt
ELIS department, Computer Systems Lab, Ghent University
iGent, Technologiepark Zwijnaarde 126, B-9052 Ghent - Belgium
Email: {Vijaykumar.Guddad, Alexandra.Kourfali, Dirk.Stroobandt}@UGent.be

***Abstract*—In Field-Programmable Gate Array (FPGA) design, the coding style has a considerable impact on how an application is implemented and how it performs. Many popular Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) logic synthesis tools like Vivado by Xilinx, Quartus II by Altera, and IspLever by Lattice Semiconductor, have significantly improved the optimization algorithm for FPGA synthesis. However, the designer still has to generate synthesizable VHDL code that leads the synthesis tools and achieves the required result for a given hardware architecture. To meet the required performance, VHDL based hardware designers follow their own rules of thumb, and there are many research papers which suggest best practices for VHDL hardware designers. However, as many trade-offs have to be made and results depend on the combination of optimized implementations and optimized hardware architectures, final implementation decisions may have to change over time. In this paper, we present a VHDL design tool flow that makes portability of the design to new design requirements easier. It helps to generate automated portable VHDL design implementations and customized portable VHDL design implementations. This tool flow helps the VHDL hardware designers to generate a single VHDL design file, with multiple design parameters. It also helps the end-users of VHDL hardware designs in choosing the right parameter settings for a given hardware architecture and generating the right bit file corresponding to these parameter settings, according to their requirements.**

***Keywords*–FPGA; VHDL; Toolflow; VIVADO.**

I. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are attractive platforms for custom hardware implementation. They have been used in accelerating high-performance applications in which the complexity is significantly reduced by employing custom hardware to parts of the problem. They have been attractive for many new applications in which their flexibility and configurability are in high demand. The FPGA's strength comes from the fact that hardware developers can program it to deliver exactly what they need for their design, and massive spatial parallelism at low energy gives FPGAs the potential to be core components in large scale High-Performance Computing (HPC).

Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) is a Hardware Description Language (HDL) that is used to program FPGAs. It describes the behaviour of an electronic circuit or system, from which the physical circuit or system can then be implemented. FPGAs work on configuration bits that define the functionality. To generate configuration bits from HDL, an FPGA tool flow is used. An FPGA tool flow aims to produce a configuration for the target FPGA that implements the functionality described in the HDL design. The current FPGA tool flow consists of synthesis, technology mapping, packing placement and routing. In the synthesis step, the HDL code is translated from a human-readable form to a gate-level logic circuit. The synthesis tool is also responsible for optimising the circuits depending on the needs of the designer. During technology mapping, the gate-level circuit generated by the synthesis step is mapped onto the resource primitives (ex. Lookup tables (LUTs), Flipflops (FFs), DSP blocks (DSPs), BlockRAMs) available in the target FPGA architecture. In packing, LUT primitives and FFs from the mapped netlist are clustered into Configuration Logic Blocks (CLBs) according to their interconnectivity. During placement and routing, CLBs are assigned to physical logic blocks on the FPGA, and these CLBs are connected using switch blocks and wires. Finally, the configuration bitstream is generated. Recently, increasing research has been performed in the field of placement and routing, and also commercial FPGA tools (Vivado by Xilinx, Quartus II by Altera, IspLever by Lattice Semiconductor, Encounter RTL compiler by Cadence Design Systems, LeonardoSpectrum, Precision by Mentor Graphics, and Synplify by Synopsys) optimised their algorithm for better results [1]–[3].

The tool flow proposed in this paper is mainly related to targetting the synthesis step. The coding style can have a severe impact on the resource utilisation of an FPGA architecture, and how it performs on the target board. The designer can write HDL code that forces the synthesis step to make use of available FPGA resources or not to use the specific resources. The designer can also write HDL code that has higher or lower throughput or a different design operating frequency. There are many research papers and books [4]–[6] which suggest best practices and techniques; also, each commercial FPGA vendor has their own coding guidelines [7] [8]. Apart from all these guidelines, each designer and company follows their own rules of thumb in order to achieve the required results according to the design requirements.These techniques and guidelines are specific to the particular FPGA architecture and model. As we can see in the current market, FPGA architectures and models keep changing to fulfill a new market need (e.g., currently, most FPGA vendors are designing their boards to target machine learning, deep neural networks, and data server requirements). From the above discussion, we can categorize these coding techniques and methods into three main categories: i) technology independent coding styles, ii) performance driven coding, iii) technolonology specific coding techniques.

In this paper, we present a method to combine all possible coding techniques, methods, and rules of thumb in a single VHDL design file. The tool flow processes the VHDL design file with all possible techniques, methods, and rules of thumb, which is independent of the FPGA vendor and the architecture. In Section II and Section III, we present different types of VHDL coding techniques and methods and describe a method to combine all possible techniques in a single VHDL design file. In Section IV, we present the portable tool flow integrated with a synthesis tool to process input VHDL design files. In Section V, we give results and a conclusion.

II. VHDL CODING TECHNIQUES AND METHODS

In this section, we will discuss three main coding techniques and methods with an example. In the end, we describe a method to combine these techniques and methods in a single VHDL design file.

A. Technology independent coding styles

As the name suggests, technology independent coding techniques are independent of the FPGA architecture, vendor, and technology. Here, we will discuss a few techniques [9]–[12].

1) Sequential devices design techniques: In sequential devices, we have two main types of memory devices: a latch and a flip-flop. A latch is a level-sensitive memory device and a flip-flop is an edge triggered memory device [3] [13].

Data-Latches: Here we will see different ways, of using Data-latches (D-latches).

D-Latch with data and enable:

```
begin
process (enable, data) begin
   if (enable= '1') then
   y<=data;
```

D-Latch with gated asynchronous data:

```
process (enable, gate, data) begin
    if (enable = '1') then
       q <= data and gate;
```

D-Latch with gated enable:

```
process (enable, gate, d) begin
    if ((enable and gate) = '1') then
        q <= d;
```

D-Latch with asynchronous reset:

```
process (enable, data, reset) begin
    if (reset = '0') then
       q <= '0';
    elsif (enable = '1') then
       q <= data;
```

2) Datapath: Datapath logic is a structured repetitive function. These structures can be modelled in a different implementation depending upon timing and area constraints. The following synthesis tools generate optimal implementations for the target technology depending upon the datapath model used in the VHDL code [9] [13].

(i) Using if-then-else and case statement: An if-then-else statement is used to execute sequential statements based on a condition. Each of the if-then-else statements is checked until a true condition is found. Statements associated with a true condition are executed and the rest of the statement is ignored. Using if-then-else statements in VHDL code forces synthesis tools to realize the circuit in a way shown in Figure 1.

A case statement implies parallel encoding and a case statement is used to select one of several alternative statement

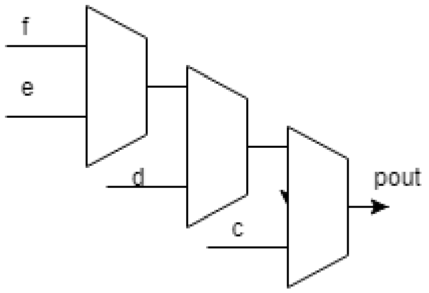


Figure 1. Effect on synthesis tool using an if-then-else statement

sequences based on the value of a condition. The condition is checked against each choice in the case statement until the match is found. Using case statements in VHDL code forces synthesis tools to realise the circuit in a way shown in Figure 2.

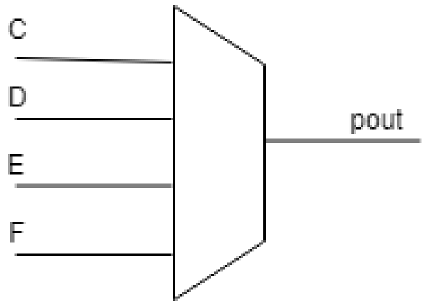


Figure 2. Effect on synthesis tool using a case statement

While writing VHDL code, it is difficult to predict how using an if-then-else statement or a case statement will effect the critical path of the final design or the design throughput or design requirements. The optimisation level of each statement varies from one synthesis tool to the other. In such cases, we can define both statements in a VHDL design file using the keyword - -# using_case and - -#using_if_else. Later, we can make choices at the synthesis step depending upon requirements using the tool flow presented in the next section.

(ii) Designing Multiplexers: While writing VHDL code, we can force the synthesis tool to make use of 4:1 or 6:1 or 12:1mux. The LUT inputs vary with the architecture, thus optimizing for different mux types can affect the synthesis to architecture step [10] [14].

(iii) Counters: Counters count the number of occurrences of an event that occurs either at regular intervals or randomly. Counters can be designed in one of the following ways: i) a counter with count enable and asynchronous reset, ii) a counter with load and asynchronous reset, and iii) a counter with load, count enable, and asynchronous reset. However, most synthesis tools cannot find the optimal implementation of counters higher than 8- bits. If the counter is in the critical path of a speed and area critical design, it is better to redesign using one of the ways mentioned above or to use a pre-instantiated counter provided by the vendor [9] [10].

3) Input-output buffers: We can infer or instantiate an Input/Output buffer in the VHDL design depending upon design requirements. The usage of inference and instantiation

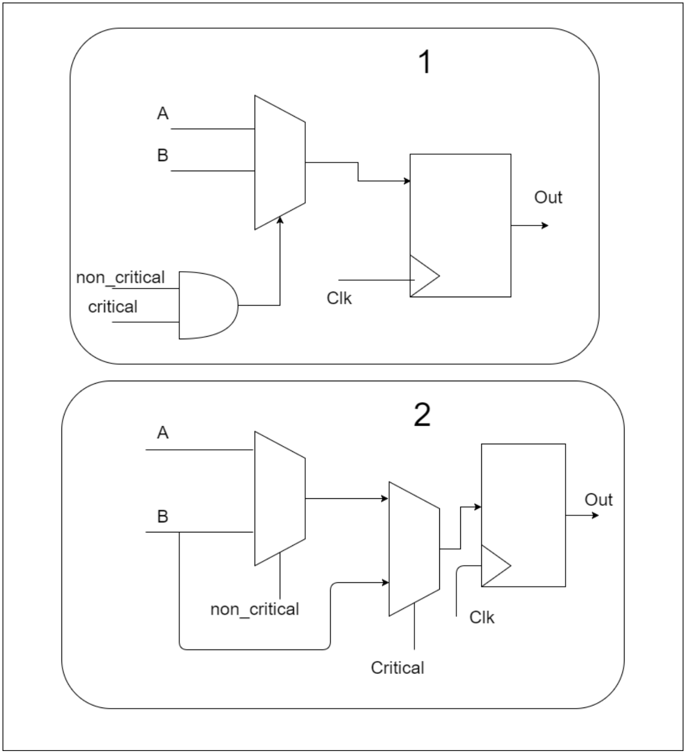


Figure 3. Example circuit designs

has its own advantages and disadvantages. For example, in the inference method, we define a tri-state buffer using an entity port in the design, whereas in the instantiation method, we make use of the tristate component design provided by synthesis tools [4] [9].

B. Performance driven coding

In the FPGA, each logic level used in the design path can add a delay. As a result, meeting timing and area constraints on a critical path with many logic levels becomes difficult. Using an efficient coding style is important because it dictates the synthesis logic implementation. In this section, we will discuss a few essential techniques [9] [15].

1) Reducing logic levels on critical paths: Consider a small circuit design, as shown in Figure 3. Here, we have two circuit designs with the same functionality but designed differently.

In circuit 1 of Figure 3, the signal ”critical” goes through two logic gates.

```
if (clk'event and clk ='1') then
     if (non_critical and critical) then
        out1 <= A
     else
        out1 <= B
     end if;
end if;
```

To reduce the logic gate usage on “critical” signals, multiplex inputs “A” and “B” based on “non_critical” and call this output “out_temp”. Then multiplex “out_temp” and “B” based on “critical”. As a result, the signal “critical” goes through one logic gate as shown in circuit 2 of Figure 3.

```
if (clk'event and clk ='1') then
     if (non_critical and critical) then
        out1 <= A
     else
        out1 <= B
     end if;
end if;
```

2) Resource sharing: The resource sharing technique is used to reduce the number of logic modules needed to implement VHDL operations. Here, we have two pieces of VHDL code: one makes use of four adders and another uses two adders.

```
--Example implementation with 4 Adders
if (...(siz == 1)...)
    count = count + 1;
else if (...((siz ==2)...)
    count = count + 2;
else if (...(siz == 3)...)
    count = count + 3;
else if (...(siz == 0)...)
     count = count + 4;
--Example implementation with 2 Adders
if (...(siz == 0)...)
   count = count + 4;
else
   count = count + siz
```

C. Technology specific coding techniques

These coding techniques are used to take advantage of the specific FPGA architecture, to improve speed and area utilization of the design. These techniques have their own coding guidelines to take advantage of their FPGA architectures [8] [16].

III. METHOD TO COMBINE ALL POSSIBLE DESIGN TECHNIQUES IN A SINGLE VHDL DESIGN FILE

From the above discussions, we observe that the VHDL coding style has a considerable impact on how an FPGA design is implemented, and ultimately, how it performs. Here, we present a method to combine all possible VHDL design methods and techniques in a single design file. Let us consider a VHDL design file where we have two processes in a behavioral architecture and each process has to be described with different pipeline stages. Later, before the synthesis step, we plan to select the pipeline stages between each process, and a method to combine this would be as follows:

```
architecture behavioral of <identifier> is
begin
 process(<signal>)    -- Process 1
begin
  --#pipelined=0
  --   < code>
 --#pipelined=0
 --#pipelined=1
      < code >
 --#pipelined=1
 process(<signal>)    --Process 2
begin
  --#pipelined=0
     < code>
```

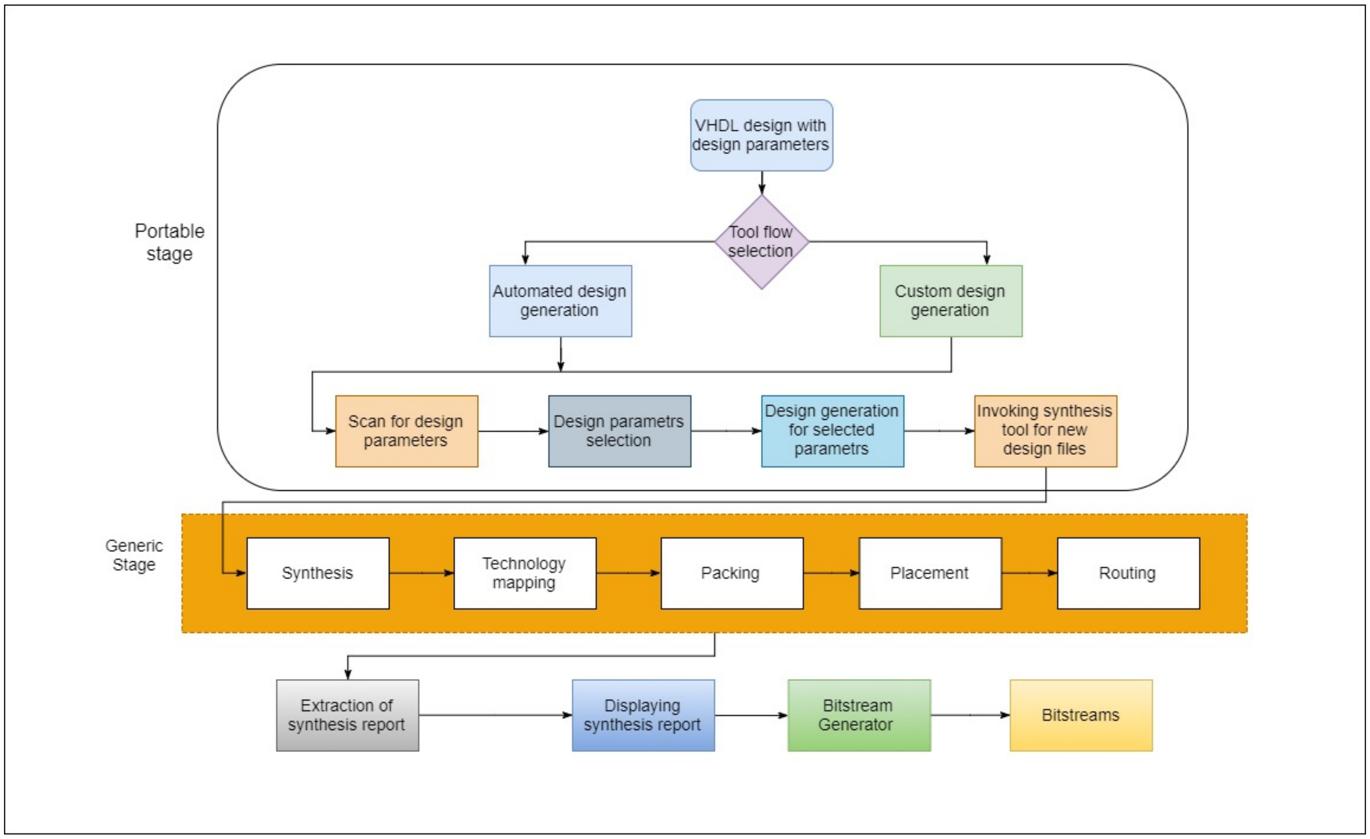

Portable stage
VHDL design with design parameters
Tool flow selection
Automated design generation
Custom design generation
Scan for design parameters
Design parametrs selection
Design generation for selected parametrs
Invoking synthesis tool for new design files
Generic Stage
Synthesis
Technology mapping
Packing
Placement
Routing
Extraction of synthesis report
Displaying synthesis report
Bitstream Generator
Bitstreams

Figure 4. VHDL design tool flow for portable FPGA implementation

```
--#pipelined=0
--#pipelined=1
--     < code >
--#pipelined=1
```

Here, we make use of the keyword "- -#" followed by a design parameter instance. We can use this keyword to define any part of the code like input-output ports, signals, or the architecture part of the code. We can use any possible name to define the design parameter instance and there is no syntax rule for the names. The only condition we put forward for different parameter design parts is that they should start and end with the same design parameter name (from the above code design, a parameter means –#pipelined=1, –#pipelined=2). From the above code, we can observe in process 1 we commented the pipelined=0 and kept the pipelined=1 as default. In process 2, we kept pipelined=0 as default. It will help in processing the design files without our tool flow.

IV. VHDL Design Tool Flow for Portable Design Generation

As seen in Section III, now we have a way to combine multiple design techniques and methods in a single VHDL design file. However, this type of VHDL design file cannot be synthesised by regular synthesis tools. Therefore, we need a tool flow which can guide the synthesis tool in implementing combined VHDL design files, and which can allow the user to decide and generate portable VHDL designs.

In Figure 4, we present our proposed tool flow. As it can be observed, the tool flow is designed in two stages: the portable stage and the generic stage. The portable stage processes the input design files and allows the user to make the selection between different design parameters. The generic stage processes the new design files generated by the portable stage and generates a synthesis report and a bit-file.

In the portable stage, the tool flow receives the input design files. The tool flow scans for the design parameters available in the design files and prints them to the user. Then, the tool flow allows the user to select either automated design or customised design. In the automated design, the user can apply a selected set of design parameters or techniques to all available design files. In the customised design selection, the user can choose between different design files, for using selected design parameters. After successful selection of the design parameters available from the input design files, the tool flow searches for the user selected design parameters in each design file and extracts the design parameter content from each design file. If the user selected design parameter does not match with parameters in the design file, the tool flow will keep the default design parameter. After this the tool flow generates the new design files. The newly generated design files are processed further with the generic stage to generate a synthesis report. Next, the synthesis report is extracted and displayed to the user, and finally, the bit-file is generated. The portable tool flow proposed in this paper is independent of the used FPGA architecture. This can be integrated with any FPGA architecture, just by changing the invoking statement in the code. For example, to invoke the Xilinx Vivado design suite, the following code is used:

```
vivado-mode batch -source design.tcl
```

One can think it is a lot of manual work to define all possible design parameters in a design file, but as we discussed in the introduction, our idea is to combine all possible coding techniques, methods, and rules of thumb in a single design file. It is easier for a designer to propose different smaller design options in sub-parts of the design than to provide different complete designs full of different choices. So the designer does not have to worry about how different choices are combined (as this is done automatically) but can focus on the individual different options. This is a huge difference.

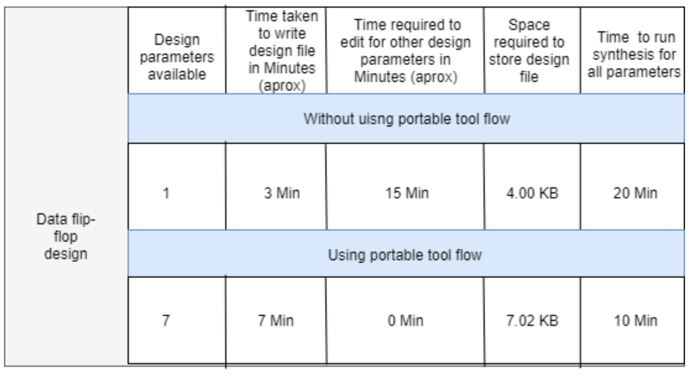
V. Results and discussion

In this section, we will evaluate the portable VHDL design tool flow and present the results [17]. In our experiments, we used an 8 core CPU system. We integrated our portable VHDL design tool flow with the Xilinx synthesis tool (Vivado 2018.3) [18], to generate the synthesis report and the bit-file.

A. Evaluating the portable VHDL design tool flow for a single design file (using technology independent coding styles)

Here, we considered the data flip-flop VHDL design file with seven design parameters (techniques) in a single VHDL design file. We compared our results with the standard VHDL design with a single design parameter. These results are tabulated in Table I.

TABLE I. RESULTS USING PORTABLE VHDL DESIGN TOOL FLOW AND GRAPHICAL USER INTERFACE OF SYNTHESIS TOOL



	Design parameters available	Time taken to write design file in Minutes (aprox)	Time required to edit for other design parameters in Minutes (aprox)	Space required to store design file	Time to run synthesis for all parameters
Data flip-flop design	Without uisng portable tool flow				
	1	3 Min	15 Min	4.00 KB	20 Min
	Using portable tool flow				
	7	7 Min	0 Min	7.02 KB	10 Min

From Table I, we can observe that we have two different implementation results: one using our portable VHDL design tool flow, and another one using the Graphical User Interface (GUI) of Vivado design suite. In this experiment, we considered the seven design techniques (parameters) to design a data flip-flop. These are i) rising edge flip-flop, ii) rising edge flip-flop with asynchronous reset, iii) rising edge flip-flop with asynchronous preset, iv) rising edge flip-flop with asynchronous reset and preset, v) rising edge flip-flop with synchronous reset, vi) rising edge flip-flop with synchronous preset, vii) rising edge flip-flop with asynchronous reset and clock enable. While using the GUI, we designed the data flip-flop considering one design technique at a time and subsequently edited the design for other parameters to get the corresponding synthesis report and the bit-file. In the second instance, we used the portable VHDL design tool flow, combining all techniques in a single design file and generating the synthesis report and the bit-file.

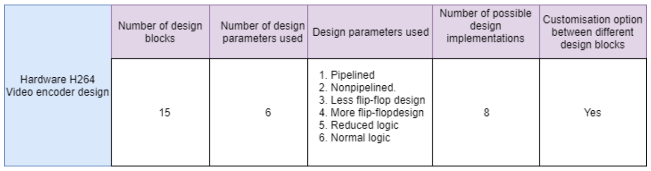
From the results in Table I, we can observe that writing a combination of required VHDL design techniques in a single file takes more time, but if we want to edit for other parameters or design techniques, it will take an extra 15 minutes at later stages without the use of our tool flow. Using our portable tool flow, we can generate the synthesis results and the bit-file for all seven design parameters at the same time, so we can easily compare the results and choose the right implementation. Otherwise, we need to edit the design file each time and run the synthesis tool.

B. Evaluating the portable VHDL design tool flow for multiple design files (H264 Video encoder design)

In this section, we evaluate our tool flow for the complete hardware H264 video encoder design, which consists of 15 different design blocks, as shown in Table II. Here, we added six design parameters in the design blocks. Using 6 different design parameters in the H264 encoder design, we are able to generate 8 different combinations, by selecting one design parameter each time or multiple design parameters in various combinations, which leads to different implementations and results. Using our tool flow here, we have the option of customisation by making parameter selections to a few design blocks.

Apart from portability options, we provide the automation option in running the complete synthesis steps. Using our tool flow, we can check the resource utilization for each design block. Our tool flow runs the synthesis step in parallel. A few more comparisons are tabulated in Table III.

TABLE II. EVALUATION OF HARDWARE VIDEO ENCODER H264 USING PORTABLE VHDL DESIGN TOOL FLOW



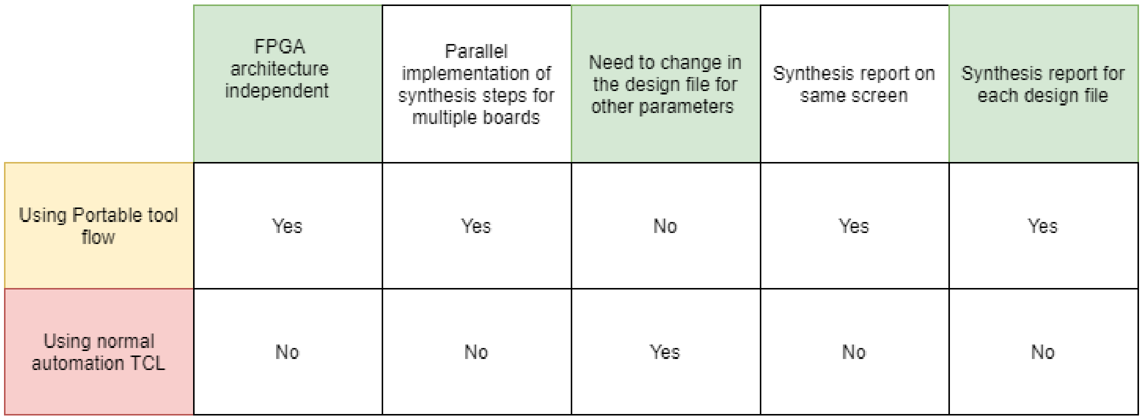
	Number of design blocks	Number of design parameters used	Design parameters used	Number of possible design implementations	Customisation option between different design blocks
Hardware H264 Video encoder design	15	6	1. Pipelined 2. Nonpipelined. 3. Less flip-flop design 4. More flip-flopdesign 5. Reduced logic 6. Normal logic	8	Yes

VI. Conclusion

In this paper, we proposed a way to combine the different possible VHDL design techniques and methods in a single VHDL design file. This can be evaluated into multiple (functionally equivalent) design files, that can be compared to allow the designer to choose between different implementation techniques, based on the achieved trade-off between FPGA resource utilisation and performance. In that way, by evaluating the same VHDL design file, the designer can estimate which design option is the better choice.

Additionally, we proposed a tool flow that allows the user to generate the design bit-files of all possible combinations automatically. This tool flow can be integrated into any other synthesis tool. Hence, it forms a pre-synthesis step, that provides the user with the flexibility of having multiple design

TABLE III. COMPARISON OF PORTABLE VHDL DESIGN TOOL FLOW OVER USUAL TOOL COMMAND LANGUAGE (TCL) AUTOMATION



	FPGA architecture independent	Parallel implementation of synthesis steps for multiple boards	Need to change in the design file for other parameters	Synthesis report on same screen	Synthesis report for each design file
Using Portable tool flow	Yes	Yes	No	Yes	Yes
Using normal automation TCL	No	No	Yes	No	No

options, but without having to redesign them, only to generate the new file, based on a set of parameters.

Using our tool flow, designers can easily redo the design exploration when the underlying FPGA architecture for the design changes. They do not have to delve into the VHDL source code for this. For the final design parameter choice, the tool can automatically generate the bitstream. Hence, this tool flow significantly enhances the portability of designs to new FPGA devices.

ACKNOWLEDGMENT

This work was supported by the Help Video! imec.icon research Project, funded by imec and the Flemish government (Agentschap innoveren and ondernemen).

REFERENCES

[1] J. de Fine Licht, M. Blott, and T. Hoefler, “Designing scalable FPGA architectures using high-level synthesis,” Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2018, pp. 403–404. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178527

[2] K.Kuusilinna, Timo.Hmlinen, and Jukka.Saarinen, “Practical VHDL optimization for timing critical FPGA applications,” Microprocessors and Microsystems, vol. 23, no. 8, pp. 459–469.

[3] P. I. Necsulescu and V. Groza, “Automatic generation of VHDL hardware code from data flow graphs,” 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), May 2011, pp. 523–528.

[4] P. P. Chu, “Coding for efficiency, portability, and scalability,” hardware design using VHDL, 2006, pp 50-100, ISSN:13: 978-0-471-72092-8.

[5] Xilinx corporation ”Xilinx product guide”, 1.1, 2006, URL: https://bit.ly/2Gz2RO9/ [accessed: 2019-09-17].

[6] T. Davidson, K. Bruneel, and D. Stroobandt, “Identifying opportunities for dynamic circuit specialization,” 2012, workshop on Self-Awareness in Reconfigurable Computing Systems Proceedings, Oslo Norway, p-p 18-21. [Online]. Available: http://srcs12.doc.ic.ac.uk/docs/srcs_proceedings.pdf

[7] M. Arora, “The art of hardware architecture, design methods and techniques for digital circuits,” Design Methods and Techniques for Digital Circuits, springer-Verlag New York, 2011, DOI:10.1007/978-1-4614-0397-5.

[8] R. Jasinski, Effective Coding with VHDL: Principles and Best Practice. The MIT Press, 2016.

[9] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, “Improvement of fault injection techniques based on VHDL code modification,” Tenth IEEE International High-Level Design Validation and Test Workshop, 2005., Nov 2005, pp. 19–26.

[10] R. P. P. Singh, P. Kumar, and B. Singh, “Performance analysis of fast adders using VHDL,” 2009 International Conference on Advances in Recent Technologies in Communication and Computing, Oct 2009, pp. 189–193.

[11] Z. Zhang, Q. Yu, L. Njilla, and C. Kamhoua, “FPGA-oriented moving target defense against security threats from malicious FPGA tools,” 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), April 2018, pp. 163–166.

[12] Z. Jia, B. Qi, L. Chen, H. Chen, and L. Ma, “Relative radiometric correction for remote sensing images based on VIVADO HLS,” IET International Radar Conference 2015, Oct 2015, pp. 1–4.

[13] V. S. Rosa, F. F. Daitx, E. Costa, and S. Bampi, “Design flow for the generation of optimized FIR filters,” Dec 2009, pp. 1000–1003.

[14] G. Donzellini and D. Ponta, “From gates to FPGA learning digital design with deeds,” March 2013, pp. 41–48.

[15] M. S. Sutaone and S. C. Badwaik, “Performance evaluation of VHDL coding techniques for optimized implementation of ieee 802.3 transmitter,” Jan 2008, pp. 287–293.

[16] Altera corporation, Altera product guide, 9.1, 2009, URL: http://bit.do/eJsnK/ [accessed: 2019-09-17].

[17] Github link, URL: https://bit.ly/2Sh1kgU/.

[18] Xilinx vivado URL: https://bit.ly/2AVvccx.

Implementation of an FPGA - Raspberry Pi SPI Connection

Haissam Hajjar
Department of Applied Business Computer,
Faculty of Technology, Lebanese University
Saîda, Lebanon
haissamh@ul.edu.lb

Hussein Mourad
Department of Applied Business Computer,
Faculty of Technology, Lebanese University
Saîda, Lebanon
mourad_hussein@hotmail.com

***Abstract*— The use of Field Programmable Gate Arrays (FPGAs) requires low level programming. This makes it difficult to have a friendly user interface. The presented work explains FPGA techniques in detail. There are few works demonstrating an application integrating FPGA and ergonomic user-interface techniques. This article describes the connection of an FPGA to a Raspberry PI using a Serial Peripheral Interface (SPI) link. A Python SPI driver is developed on the Raspberry side. A Very High-Speed Integrated Circuit Hardware Description Language (VHDL) driver is developed on the FPGA side. A Web client-server application is developed to demonstrate the usage of SPI link and its integration with a standard Web application to control the FPGA inputs and outputs.**

Keywords—SPI VHDL driver; VHDL; Raspberry PI; Altera Cyclone II; Python VHDL communication; Python-PHP socket communication.

I. INTRODUCTION

FPGAs are typically used in electronic circuits. Usually, they are programed in VHDL or Verilog [1]. This is well suited to stay at the hardware level but remains very poor and complex when developing a user-friendly human-machine interface.

The VHDL implementation of SPI protocol is developed in some previous works [2][3]. However, these works focus their efforts on the electronic aspect by neglecting the application aspect.

The objective of this paper is to connect an FPGA to a Raspberry PI so that one side can use the FPGA for the electronic part, while the Raspberry PI can be used to develop a friendly user interface using common well-known techniques. The utilization of a Raspberry PI is taken to demonstrate a low-cost solution for this implementation.

As the Raspberry runs under Linux operating system and the FPGA is programmed at an electronic level, we elected to use the SPI standard that does not need to use a common clock (see Figure 1). The communication is synchronized by a clock signal delivered by the SPI Master, independently of the internal clock frequency of each side.

On the programming language level, we choose to use Python for the Linux side (Raspberry) and VHDL for the FPGA side. So, the SPI driver can be integrated with the commonly used frameworks on the Linux side.

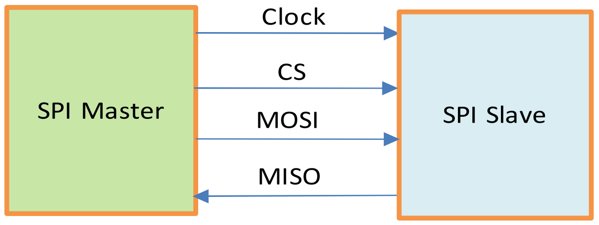


Figure 1. SPI Single Master – Single Slave signals - Chip Select (CS), Master Out Slave In (MOSI), Master In Slave Out (MISO)

This paper covers the following topics: Section I has provided an introduction. In Section II, we give a functional description of the implemented system. In Section III, we describe the hardware implementation and the materials used. In Section IV, we develop the SPI implementation on the Master level and the Slave level. In Section V, we present testing results of the SPI. In Section VI, we describe a high-level user interface developed to illustrate the SPI utilization. Finally, a conclusion is included in Section VII.

II. FUNCTIONAL DESCRIPTION

Figure 2 illustrates a functional representation of the whole system:

A. Raspberry PI

A Raspberry PI 3 [4] functions as Master of the SPI link. The Raspberry PI is equipped with a General Purpose Input/Output (GPIO). The SPI was implemented using 4 lines of this GPIO. A Python implementation of SPI Master is utilized. An Apache Web server is implemented inside the Raspberry to allow implementation of ergonomic and easy to use interface for testing and demonstration purposes. A SPI Master driver is developed using Python language and libraries. As the system must work efficiently with regards to real time response time, two independent processes were implemented within the Raspberry PI system:

- To handle the user requests: an Apache Web server is implemented using PHP scripting for the Web server side.
- To handle the SPI link during the communication with the FPGA. This driver is written using Python.

- The communication between these two processes is executed using TCP/IP socket communication.

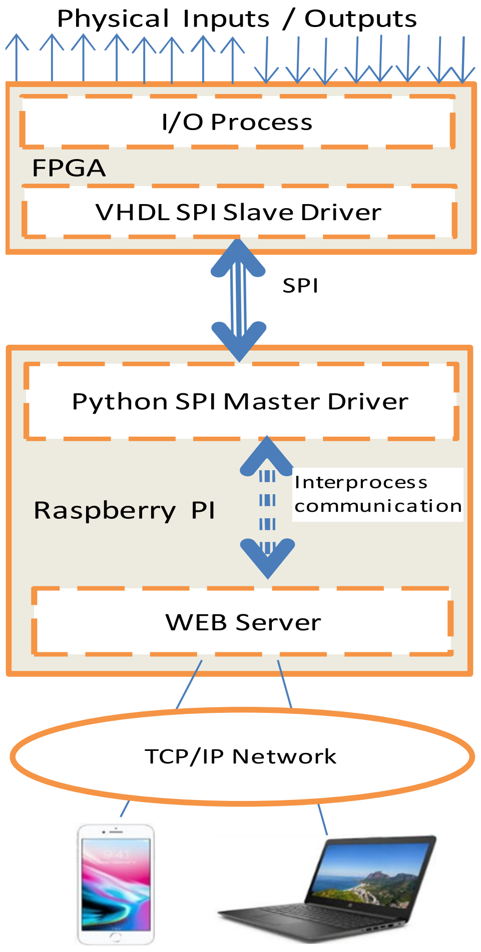


Figure 2. Functional representation

B. FPGA

An ‘Altera DE2’ Development and Education Board’ [5] is used to implement the FPGA part. This board is built on a Cyclone II EP2C35F672C6 FPGA working up to 50 MHz clock frequency. This board has a 2 lines/16-character LCD display, a set of 18 toggle switches for digital inputs, a set of 4 pushbuttons, a set of 18 red led for digital outputs and 2 forty lines extension headers for external connection. The SPI is implemented using one of these extension headers.

To illustrate the successful operation of the SPI link, we devise three functional usages:

- Send order, starting from the user interface, to drive the 18 FPGA board digital outputs
- Receive the status of the 18 lines of digital input to display on the user screen.
- Send 32 bytes, entered on the user screen, in order to be displayed on the FPGA LCD 2 lines display.

Compared to OSI communication layer, we can consider the SPI drivers as the physical layer and these functional usages as a link layer. So, this can be extended to implement other functional types of messages exchanged between the FPGA and the Raspberry.

***TCP/IP network*:** The Raspberry PI has an 802.11 Wi-Fi 2.4 GHz interface. This interface is used to allow the connection of a Web-based client using a standard browser. A Web-based user interface is developed to allow the usage of the previously mentioned illustration function for the use of the SPI connection.

III. PHYSICAL IMPLEMENTATION

Figure 3 represents the physical implementation:

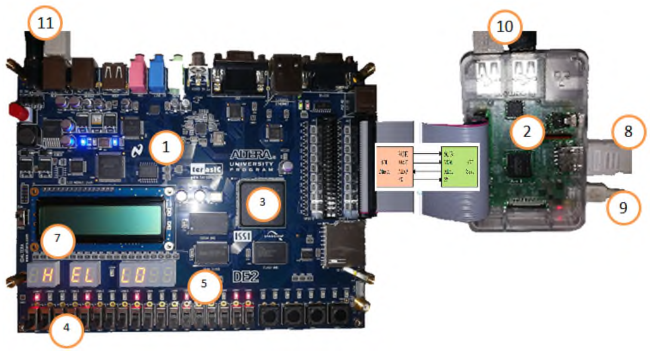


Figure 3. Physical implementation

1. DE2 Development and Education Board: we use this board for the FPGA implementation part. For detailed documentation, refer to the Intel official documentation [5]
2. Raspberry Pi 3 board [3]: this board is equipped with 1 GB Ram, processor Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz.
3. Cyclone® II 2C35 FPGA in a 672-pin package, working at 50 MHz
4. 18 switches used as digital inputs
5. 18 LEDs used as digital outputs
6. 40 pins flat cable used as connector between the Raspberry GPIO and the extension header of the DE2 board. This cable is used to implements the SPI connection between the Raspberry PI and the Altera DE2 FPGA evaluation board.
7. 2x16 LCD and eight 7 segment digital display
8. HDMI connector for Raspberry PI
9. Power supply for Raspberry PI
10. Mouse and keyboard USB connectors
11. Power Supply and Programmer connection

IV. SPI IMPLEMENTATION

The implemented system is represented in Figure 4. A single Master with single Slave scenario is shown in the following paragraphs.

A. *Physical interface*

Figure 4 presents the SPI signals. The Raspberry PI is the master and the FPGA is the slave. A configuration of one Master/one Slave is implemented:

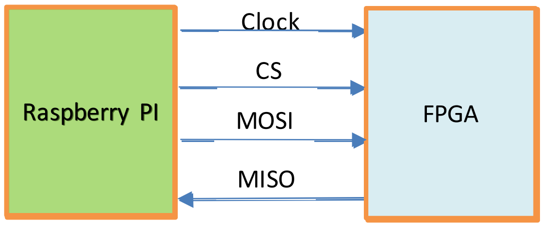


Figure 4. SPI implementation

- Clock: the clock is generated by the Master. This signal drives the communication in both directions.
- CS (Chip Select) high when the FPGA is not selected: No communication; low when selected.
- MOSI: Master Out Slave In: data transferred from the Master to the Slave.
- MISO: Master In Slave Out: data transmitted from the Slave to the Master.

B. *Implementation principle*

The communication is driven by the Master. The first byte determines the type of communication. To illustrate the usage of the drivers, three types of messages were implemented:

- Send Memory: The Master sends to the Slave 32 bytes of data for displaying on the LCD.
- Send Outputs: The Master sends to the Slave the order to set its digital outputs ON or OFF
- Receive Inputs: The Slave sends to the Master the status of its digital inputs.

C. *Master Driver*

This driver is based on the RPI.GPIO Python library [9]. After initialization, two functions are available for an upper level usage:

Sendbyte: send a byte from Master to Slave.

Receivebyte: receive a byte from Slave to Master.

To send a bit, MOSI is set, and then a Clock is generated (SCLK from Low to High).

To receive a bit, a Clock rising is generated, and then the MISO line level is read.

SPI Initialization (Figure 5): the GPIO of the Raspberry includes 2 SPI lines. We had trouble driving these lines with the standard Raspberry library. We opted to drive the SPI signals directly through our program. This allowed us to control the CS and Clock lines easily and to reach the maximum possible communication speed with a Python driver.

```
import RPi.GPIO as GPIO
# Line definition
  MOSI = 5
  MISO = 10
  SCLK = 15
  CE0  = 7
#
# SPI line initialization
def initspi():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setwarnings(False)
    GPIO.setup(MOSI, GPIO.OUT)
    GPIO.setup(MISO, GPIO.IN)
    GPIO.setup(SCLK, GPIO.OUT)
    GPIO.setup(CE0, GPIO.OUT)
```

Figure 5. Master driver - SPI initialization

Send byte (Figure 6): The transmission of a byte starts with the change of the signal CS (CS low). This will initiate the reception process on the VHDL side. The MOSI level is set according to the bits to be sent and a clock signal is generated. After transmission of the 8 bits, this CS signal returns to the high level.

```
def sendbyte(cc):
    c=ord(cc)
    # select slave
    GPIO.output(CE0, GPIO.LOW)
    bitsx = [0,0,0,0,0,0,0,0]
    # determine bits 0/1
    for x in range(8):
        bitsx[7-x] = int(c % 2)
        c = int((c - bitsx[7-x])/2)
    # set Mosi signal level
    for x in range(8):
        if (bitsx[x]>0):
            GPIO.output(MOSI,
GPIO.HIGH)
        else:
            GPIO.output(MOSI, GPIO.LOW)
    # clock
        GPIO.output(SCLK, GPIO.LOW)
        GPIO.output(SCLK, GPIO.HIGH)
    # end of byte transmission
    GPIO.output(CE0, GPIO.HIGH)
    GPIO.output(SCLK, GPIO.LOW)
    GPIO.output(SCLK, GPIO.HIGH)
```

Figure 6. Master driver – Send byte

Receive byte (Figure 7): The reception of a byte starts with the change of the signal CS (CS low). This will initiate the transmission process on the VHDL side. The MISO level is read according to the bits received each clock signal generated. After reception of the 8 bits, this CS signal returns to the high level.

```
def receivebyte():
    GPIO.output(SCLK, GPIO.LOW)
    # select slave
    GPIO.output(CE0, GPIO.LOW)
    out = 0b0
    # read 8 bits on MISO
    for x in range(8):
        GPIO.output(SCLK, GPIO.LOW)
        GPIO.output(SCLK, GPIO.HIGH)
        out = out*2
        if GPIO.input(MISO):
            out = out + 1
    GPIO.output(CE0, GPIO.HIGH)
    return out
```

Figure 7. Master driver - Receivebyte

D. Slave Driver

Figure 8 presents the process handling the communication on the FPGA.

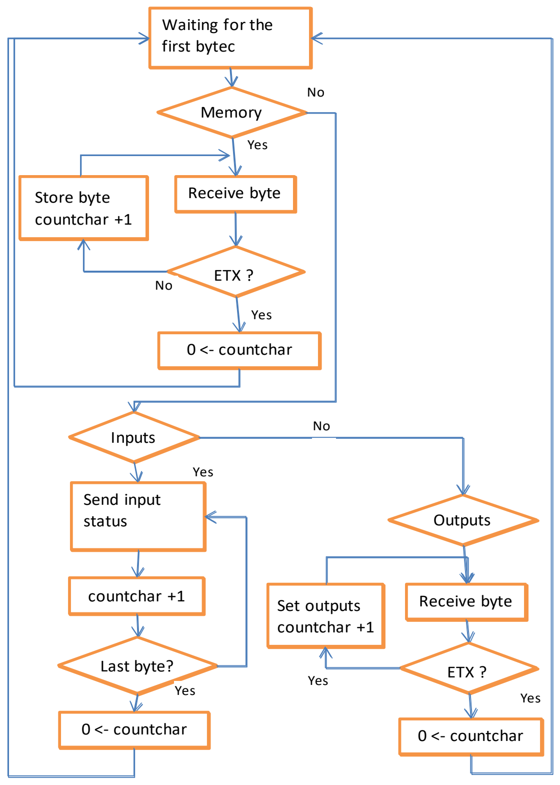


Figure 8. Master driver - Receivebyte

The process is normally in a waiting state. It is activated by the CS signal. When the CS is down, the FPGA reads the bits set on MOSI signal on rising edge of Clock signal.

```
process (SCK,CS,reset)
begin
  if (CS = '1' and octet=MasterToSlaveMemory) then
    countbit <= 0; countchar <= 0;
    direction <= RECEIVE_MEMORY;
  elsif (CS = '1' and octet=MasterToSlaveOutput) then
    countbit <= 0;        countchar <= 0;
    direction <= RECEIVE_OUTPUTS;
  elsif (CS = '1' and octet=SlaveToMaster) then
    countbit <= 0; countchar <= 0;
    direction <= SEND_INPUTS;
  elsif .....
```

Figure 9. Slave driver - Message type detection

When the first byte is received, it is tested. Three cases are considered:

- 'Send Memory': the FPGA continues the reception of the following bytes. The received bytes are stored in an internal memory indexed by a reception counter. This will end when an EOT is received.

```
-- Receive memory
if (rising_edge(SCK) and CS='0'
    and direction=RECEIVE_MEMORY) then
      octet <= octet(size-2 downto 0) & mosi;
      countbit <= countbit+1;
      if (countbit=7) then
        memory(countchar) <= octet(size-2 downto 0) &
mosi;
        if memory(countchar) = EOT then
            countchar <= 0;
        else
          countchar <= countchar+1;
        end if;
        countbit <= 0;
    end if;
end if;
```

Figure 10. Slave driver - Receive memory

- 'Send outputs': the FPGA continues the reception of data. The outputs are set/unset according to the received data.

```
-- Receive Outputs
if (rising_edge(SCK) and CS='0' and
    direction=RECEIVE_OUTPUTS) then
-- Receive 3 bytes [18 bits only valid] for digital outputs
      outputs(countchar)(7-countbit) <= mosi;
      countbit <= countbit+1;
      if (countbit=7) then
        if memory(countchar) = EOT then
            countchar <= 0;
        else
          countchar <= countchar+1;
        end if;
        countbit <= 0;
      end if; end if;
```

Figure 11. Slave driver - Receive outputs

- 'Receive inputs': the FPGA sends the status of its digital inputs [3 bytes for 18 inputs] using the MISO line. An EOT is sent to inform the Master that the end of sending is reached.

```
if (rising_edge(SCK) and CS='0'
          and direction=RECEIVE_OUTPUTS) then
 -- Receive 3 bytes [18 bits only valid] for digital outputs
   outputs(countchar)(7-countbit) <= mosi;
   countbit <= countbit+1;
   if (countbit=7) then
          if memory(countchar) = EOT then
              countchar <= 0;
          else
             countchar <= countchar+1;
         end if;
      countchar <= countchar+1;
       countbit <= 0;
   end if;
end if;
```

Figure 12. Slave driver - Request to send inputs

V. TESTING AND RESULTS

We present three tests executed to validate communication using this implementation of SPI.

A. Setting outputs

Send order from the Master (Raspberry PI) to the Slave (Altera FPGA DE2 board) to set/unset its digital outputs: 'Oxxx': Message of 4 bytes. The first byte represents the type of message; the following bytes represent the required outputs status.

Figure 13 shows the signals observed on the SPI lines. The message sent from the Master: the first byte represents the ASCII representation of the character O (01101111) used as identifier for this message. As described in Section IV, the following 3 bytes represent the value to be set on the digital outputs. The following 3 bytes represent the requested status of FPGA 18 lines output.

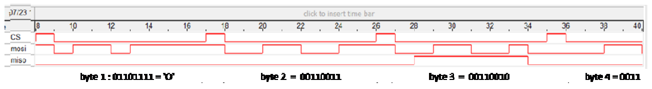


Figure 13. Signals on the SPI lines for sending outputs

Each byte starts when the CS comes down and is sent when the CS goes up again. Figure 14 shows the LEDs corresponding to the signal shown in Figure 13. The LED is on when '1' is received and is off when '0' is received.

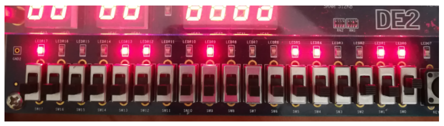


Figure 14: Led status on the FPGA board

B. Read inputs

Send order from the Master (Raspberry PI) to the Slave (Altera FPGA DE2 board) 'r': This message asks the FPGA to send back to the Raspberry the status of its digital inputs. The Raspberry (Master) must continue to generate the clock. The next bytes are sent by the FPGA (Slave) to the master over the MISO line. As we have 18 inputs, three bytes are used for this function.

Figure 15 shows the SPI signals: the clock is always given by the Master. The Master sends the first 'r' byte (01110100) over the MOSI signal. Then, the Slave sends back three bytes.

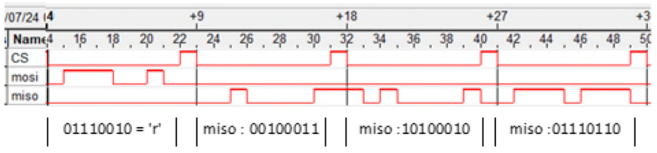


Figure 15: Signals on the SPI lines for read inputs outputs

Figure 16 shows the input switches generating the signals shown in Figure 15.

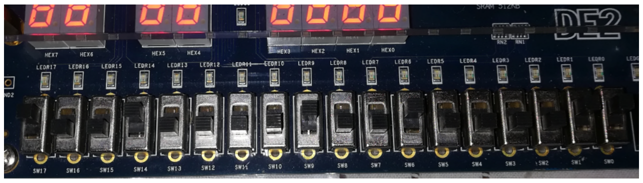


Figure 16. Input switch corresponding to Schema 6 signals

C. Performance

The performance of this link depends on the SPI Master. For the Raspberry III utilized, the speed of 100kb/s was reached.

VI. APPLICATION TESTING

An application is developed to show a concreate utilization of this work. Figure 17 shows a functional representation of this realization.

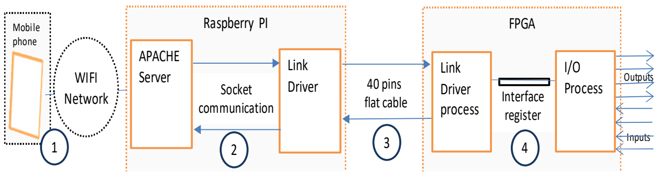


Figure 17. Signals on the SPI lines for sending outputs

The implementation inside the Raspberry PI is performed using two processes: an Apache server and the SPI link driver. This is done for real time constraints. The communication between these two processes is executed using client/server socket communication. The Apache side is developed in PHP and the SPI link driver side is developed in Python. Figure 18 presents the principle of this communication.

Figure 19 shows the user-interface on a smartphone screen using a standard Web browser.

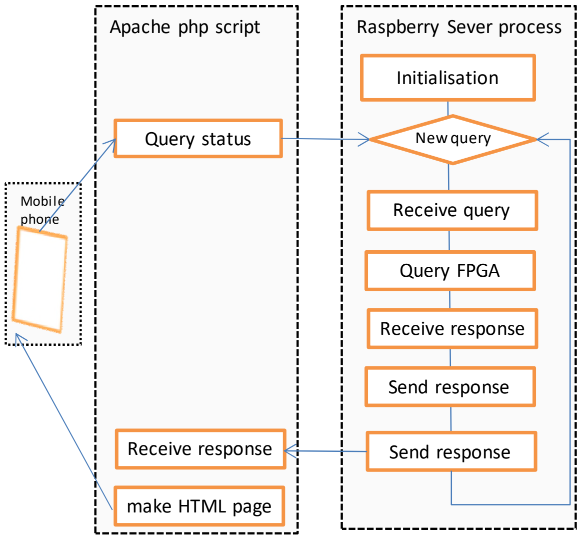
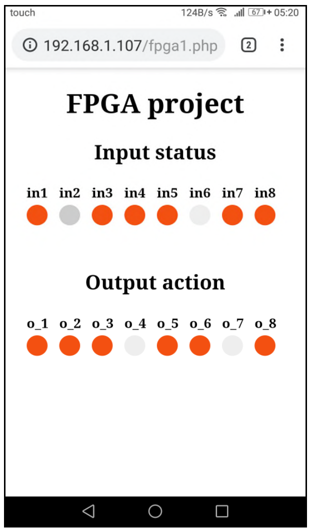


Figure 18. Client/server communication in Raspberry PI

The number of inputs and outputs are reduced to 8 to have an ergonomic user-interface on smartphones. The status of the digital inputs of the FPGA is reported on the user screen. The digital outputs of the FPGA are set according to the radio-button.



FPGA project

Input status

in1	in2	in3	in4	in5	in6	in7	in8
●	○	●	●	●	○	●	●

Output action

o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
●	●	●	○	●	●	○	●

Figure 19. User interface print screen

VII. Conclusion

In this work, FPGA-Raspberry Pi communication is developed using the SPI protocol. A high level application is developed using this link. This demonstrates a solution that works by using a low level technique (VHDL) on the FPGA side and using a high level technique on the user interface side.

We have limited the application to digital inputs / outputs. The work can be extended to other functions of the FPGA. This opens the possibility of modifying the behavior of an FPGA dynamically. The job can also be completed in the sense of increasing the transmission speed, which is somehow proportional to the Master's clock frequency.

References

[1] https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools (9/2019)

[2] N.Q.B.M. Noor and A. Saparon, "FPGA implementation of high speed serial peripheral interface for motion controller," in Proc. 2012 IEEE Symposium on Industrial Electronics and Applications (ISIEA), pp.78-83, Sept. 2012.

[3] Raspberi Pi official site: https://www.raspberrypi.org/ documentation/hardware/raspberrypi/spi/ (1/2019)

[4] Rapberry Pi 3 board - Official documentation https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ (1/2019)

[5] Altera DE2-115 Development and Education Board - https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html

[6] Python documentation, https://www.python.org/ (1/2019)

[7] https://www.php.net/manual/fr/ (1/2019)

[8] Quartus II Handbook: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf (4/2019)

[9] GPIO Raspberry installation and usage : https://www.raspberrypi-spy.co.uk/2012/05/install-rpi-gpio-python-library/ (6/2019)

[10] SPI Tutorial – COREIS https://www.corelis.com/ education/tutorials/spi-tutorial/ (5/2019)

Accelerating FPGA-Placement With a Gradient Descent Based Algorithm

Timm Bostelmann, Tobias Thiemann and Sergei Sawitzki

FH Wedel (University of Applied Sciences)
Wedel, Germany
Email: {bos,inf103917,saw}@fh-wedel.de

***Abstract*—Programmable circuits and, nowadays, especially Field-Programmable Gate Arrays (FPGAs) are widely applied in computationally demanding signal processing applications. Considering modern, agile hardware / software codesign approaches, an Electronic Design Automation (EDA) process not only needs to deliver high quality results, but also has to be swift because software compilation is already distinctly faster. Slow EDA tools can in fact act as a kind of show-stopper for an agile development process. One of the major problems in EDA is the placement of the technology-mapped netlist to the target architecture. In this work, a method to reduce the runtime of the netlist placement for FPGAs is evaluated. The approach is a variation of analytical placement, with the distinction that a gradient descent is used for the optimization of the placement. This work is based on previous publications of the authors, in which a placement algorithm using self-organizing maps is introduced and optimized. In comparison, the gradient placement approach is shown to be up to 3.8 times faster than the simulated annealing based reference with about the same quality regarding the bounding-box and routing-resource costs.**

Keywords–EDA; FPGA; placement; gradient descent.

I. Introduction

The ever-growing complexity of Field-Programmable Gate Arrays (FPGAs) has a high impact on the performance of Electronic Design Automation (EDA) tools. A complete compilation from a hardware description language to a bitstream can take several hours. One step highly affected by the vast size of netlists is the NP-equivalent placement process. It consists of selecting a resource cell (position) on the FPGA for every cell of the applications netlist. In previous publications of the authors, a placement algorithm for FPGAs based on a self-organizing map [1] was presented [2] and optimized [3]. With that approach, placements of high quality were produced. However, it was relatively slow for large netlists, even when accelerated using a Graphics Processing Unit (GPU) [4]. Therefore, in this work, a faster approach for netlist placement based on a gradient descent is presented as an updated version of the authors' previous work [4].

Due to the complexity of the netlist placement problem, many current algorithms work in an iterative manner. A well known example is simulated annealing [5], which starts with a random initial placement and swaps blocks stepwise. The result of every step is evaluated by a cost function. A step is always accepted, if it reduces the cost. If it increases the cost, it is accepted with a probability that declines with time (cooling down). An annealing schedule determines the gradual decrease of the temperature, where a low temperature means a low acceptance rate and a high temperature means a high acceptance rate. Generally, the temperature is described by an exponentially falling function like

$$T_n = \alpha^n \cdot T_0, \tag{1}$$

where typically $0.7 \leq \alpha \leq 0.95$. However, there has been a lot of research on the optimization of the annealing schedule like in [6][7]. As a result, there are many variations available for any related problem.

Analytical placement is a different approach, where the problem is described as a system of equations. By solving this system of equations, the optimal position for every element can be derived. However, solving such large equation systems takes much time. Therefore, Vansteenkiste et al. [8] have introduced a method to approximate the solution of the equation system by the steepest gradient descent. This approach is shown to be two times faster than a conventional analytical placement on average, without any penalties in quality.

In this work, a simplified implementation of the steepest gradient descent placement is described and benchmarked extensively. It is not compared to other analytical placement methods. Instead, the established implementation of the simulated annealing approach of the Versatile Place and Route (VPR) tool [9] for FPGAs is used as reference.

In Section II, the problem of netlist placement for FPGAs is introduced and the principle of netlist placement with a gradient descent is described. In Section III, the proposed algorithm is described including some details of its implementation. In Section IV, the results of the proposed algorithm are presented. As representation for real world applications, a set of twenty Microelectronics Center of North Carolina (MCNC) benchmarks [10] is used. Finally, in Section V, the results of this work are summarized and a prospect to further work is given.

II. Background

This section is separated into two parts. First, the problem of netlist placement for FPGAs is introduced. Second, the general idea of using a gradient descent for the placement of netlists for FPGAs is described.

A. Netlist Placement for FPGAs

The problem of netlist placement for FPGAs can be roughly described as selecting a resource cell (a position) on the target FPGA for every cell of the given netlist. In Figure 1, an exemplary graph of a netlist is defined. An exemplary

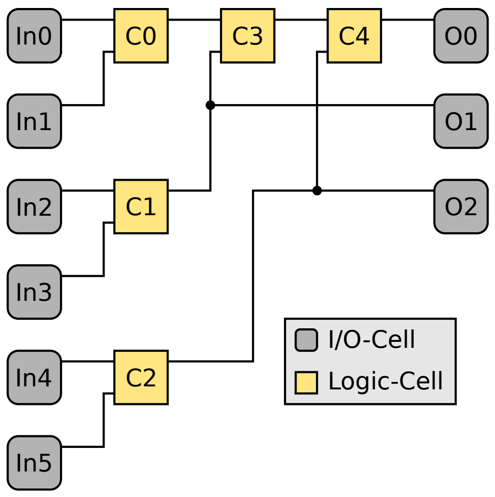


Figure 1. An exemplary graph of a netlist consisting of input-, output-, and logic-cells.

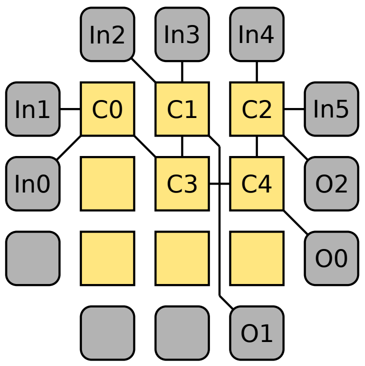


Figure 2. A valid placement for the graph in Figure 1 on a simple island-style FPGA architecture.

placement for this netlist is presented in Figure 2. The positions must be chosen in a way that:

1) Every cell of the netlist is assigned to a resource cell of the fitting type (e.g., Input / Output or Logic).
2) No resource cell is occupied by more than one cell of the netlist.
3) The cells are arranged in a way that allows the best possible routing.

The first two rules are necessary constraints. A placement that is failing at least one of these two constraints is illegal and, therefore, unusable. The third rule is a quality constraint, which is typically described by a cost function. The goal of a placement algorithm is to optimize the placement regarding this function without violating one of the necessary constraints. Usually, the length of the critical path and the routability are covered by the cost function.

B. *Netlist Placement With a Gradient Descent*

The netlist placement with a gradient descent is done by iteratively optimizing the positions of all elements of the netlist in the direction of the steepest gradient descent. During this process, the nodes are not bound to the grid of the FPGA architecture. Instead, they are positioned in a continuous space. To generate a valid placement – without overlapping and under consideration of the FPGA's architecture – in this approach, a cycle of optimization and legalization is used. This procedure is customary for analytical placement methods for FPGAs, like Gort and Anderson have introduced in [11]. A different approach would be to generate only valid placements by exclusively moving the nodes on the architectural grid of the FPGA.

III. Implementation

A. *Gradient Calculation*

At the beginning of every optimization step, the bounding-box size of every net in the netlist is determined. This is a necessary preparation for the cost-function, which is described later in this section. To determine the size of a net, all nodes with a connection to the net are determined. For all these nodes, the minimum and maximum of the horizontal positions (X_i) and the vertical positions (Y_i) are determined and stored for the calculation of the gradient. Additionally, the sum of all sizes in X and Y direction is calculated, as a metric for the global quality of the current placement.

The goal of every optimization step is to move the nodes in a direction that leads to a reduction of the bounding-box size of the containing net. A cost-function is necessary to determine the influence of every node on the size of the corresponding net. The gradient of this cost-function can then be used to determine the direction of the movement of each node. All nodes of the netlist are moved towards the steepest gradient descent to reduce the global cost.

An intuitive approach would be to use the sum of the bounding-box sizes of all nets as cost-function. However, with this metric, only the outermost nodes would be moved and even nodes that are very near to the bounding-box would be ignored. Furthermore, the min and max functions contained in the metric can not be derived to calculate the gradient.

To solve these issues, an exponential function over the distance between the position of the node and the bounding-box of the net is chosen as basis of the cost-function. The cost-function for a node with the index k is

$$C_k = \alpha_2 \cdot \sum_{n \in N_k} \Big(e^{\alpha_1 \cdot (x_k - \max_x(n))} + e^{\alpha_1 \cdot (\min_x(n) - x_k)} + e^{\alpha_1 \cdot (y_k - \max_y(n))} + e^{\alpha_1 \cdot (\min_y(n) - y_k)} \Big) , \tag{2}$$

where x_k and y_k describe the X and Y coordinates of the current node, N_k describes the set of all nets that contain the node and $\min_x$, $\max_x$, $\min_y$ and $\max_y$ are the minimal and maximal coordinates of the current net (i.e., the bounding-box). α_1 and α_2 are parameters for the cost-function, which allow to influence the behavior of the function. With α_1, it can be determined how large the distance between the node and the bounding-box must be to reduce its influence in the cost-function. The influence of α_1 on the gradient is shown in Figure 3 for the X coordinate of a node, assuming a net with the boundaries $\min_x = 1$ and $\max_x = 7$. With α_2, the cost can be increased or reduced to influence the steepness of the gradient.

Based on (2), the gradients for the X and Y coordinates

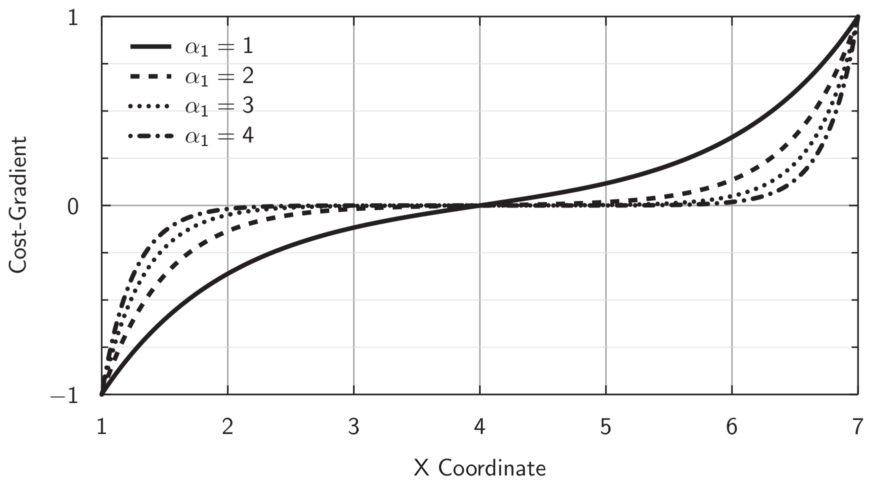


Figure 3. Exemplary plot of possible gradients for the X coordinate of a node, assuming a net with the boundaries $\min_x = 1$ and $\max_x = 7$.

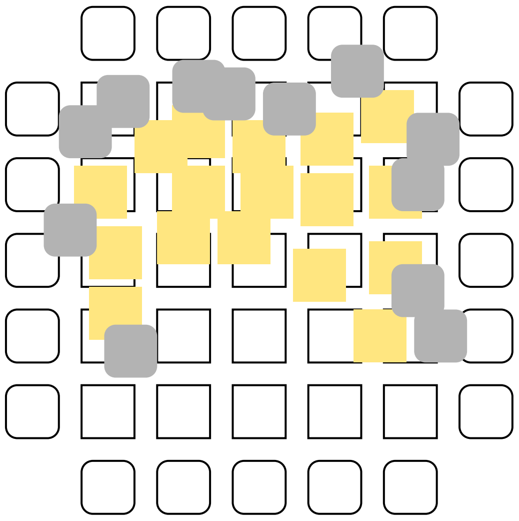


Figure 4. Exemplary placement before the legalization step.

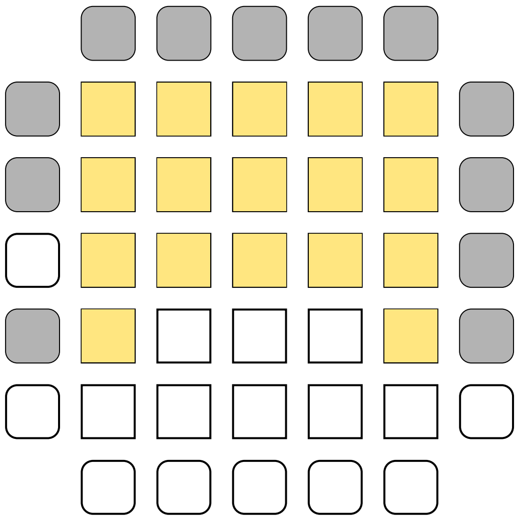


Figure 5. Exemplary placement after the legalization step.

can be calculated as

$$\frac{\partial C_k}{\partial x_k} = \alpha_2 \cdot \sum_{n \in N_k} \left(e^{\alpha_1 \cdot (x_k - \max_x(n))} - e^{\alpha_1 \cdot (\min_x(n) - x_k)} \right), \qquad (3)$$

$$\frac{\partial C_k}{\partial y_k} = \alpha_2 \cdot \sum_{n \in N_k} \left(e^{\alpha_1 \cdot (y_k - \max_y(n))} - e^{\alpha_1 \cdot (\min_y(n) - y_k)} \right). \qquad (4)$$

As a result, the coordinates of nodes that are near the bounding-box of their containing net have a gradient of $\pm\alpha_2$, where the coordinates of nodes with a larger distance to the bounding-box have a much lower gradient, as shown in Figure 3. Consequentially, nodes with a larger gradient value must be moved further to improve the placement optimally.

B. *Legalization*

During the optimization step, the nodes can take any position. Thereby, illegal placements are produced, due to overlapping of nodes, as well as violation of the architectural grid of the FPGA. Therefore, the optimized placement must be legalized in a separate step. This is done by finding the nearest valid position for every node, as depicted in Figure 4 (before the legalization) and Figure 5 (after the legalization).

The algorithm for the legalization is inspired by the work of Gort and Anderson [11]. The basic idea of that approach is to find regions that contain more nodes than the corresponding region of the FPGA provides. Then, those regions are gradually expanded. When two regions overlap, they are merged. This is done until the regions are large enough to place all contained nodes to a proper resource cell of the FPGA. In the next step, the regions are split recursively and the nodes are assigned to the new sections by their position. This is repeated until a region contains no more nodes, or only one node. In the latter case, the position of the single remaining node is set to the position of its containing region.

In this work, the search for regions that contain more nodes than the corresponding region of the FPGA provides and the following expansion and merge phases are skipped. Instead, all nodes are assigned to one large region from the start and the phase of recursive splitting starts directly. By this measure, the computational effort for the legalization is reduced significantly without a dramatic impact on the global quality. This is because – especially when a large amount of the available resources is used – the result of the expansion phase is containing usually very few large regions or often only one large region anyway.

C. Optimization

For the optimization, the algorithm Adam – which was introduced by Kingma and Ba in [12] – is used. The used update rules are:

$$\begin{aligned}
g_t &= \Delta\phi_t && \text{Gradient of the variable}\\
m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t && \text{Running average force one}\\
v_t &= \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2 && \text{Running average force two}\\
\hat{m}_t &= m_t / \left(1-\beta_1^t\right) && \text{Bias corrected force one}\\
\hat{v}_t &= v_t / \left(1-\beta_2^t\right) && \text{Bias corrected force two}\\
\phi_t &= \phi_{t-1} - S_a \cdot \hat{m}_t / \left(\sqrt{\hat{v}_t} + \epsilon\right) && \text{Update of the variable}
\end{aligned}$$

The constants β_1 and β_2 define how fast the averages of the first and second forces change. In this work, the constants were defined as $\beta_1 = 0.96$ and $\beta_2 = 0.998$. The variable S_a defines the learning rate or, more specifically, the step-width. It starts at $S_a = 1.5$, but changes over time (i.e., in the different phases of the placement).

D. Placement Phases

The previously described steps are executed for every iteration. The placement process is separated into five phases, with different parameters. Each phase consists of a given number of iterations. The number of iterations per phase was determined empirically and is fixed (i.e., independent of the size of the design). The phases are:

1) Presorting (5000 iterations)
   In this phase, all nodes are moved with a high step width in the general direction of their final position.
2) Grid placement (1000 iterations)
   In this phase, the force of the legalization is increased. Thereby, the nodes are pulled harder towards legal positions (i.e., to fitting cells of the architecture). This is necessary – for example – to prevent input and output cells from getting stuck in the logic block section of the architecture.
3) Initial detailed placement (1000 iterations)
   In this phase, the global step-width is reduced to one tenth of the initial value. This influences the legalization and the optimization equally, so that the balance between those two steps is not changed. However, the changes are much smaller, resulting in a more precise outcome.
4) Detailed placement (5000 iterations)
   In this phase, the step-width of the optimization is reduced linearly to 20 percent of its original value. Thereby, the nodes are pulled relatively harder towards their final positions in the grid.
5) Final placement (100 iterations)
   In this phase the influence of the optimization is reduced to zero, so that effectively only the legalization is active. Hence, the nodes are moved to their final position in the grid.

TABLE I. A LIST OF THE USED BENCHMARKS AND THEIR CHARACTERISTICS, THE NUMBER OF CLBS, INPUT BLOCKS, OUTPUT BLOCKS AND THE GLOBAL BLOCK COUNT

Name	Inputs	Outputs	CLBs	Blocks
ex5p	8	63	1064	1135
tseng	52	122	1047	1221
apex4	9	19	1262	1290
misex3	14	14	1397	1425
alu4	14	8	1522	1544
diffeq	64	39	1497	1600
dsip	229	197	1370	1796
seq	41	35	1750	1826
apex2	38	3	1878	1919
s298	4	6	1931	1941
des	256	245	1591	2092
bigkey	229	197	1707	2133
frisc	20	116	3556	3692
spla	16	46	3690	3752
elliptic	131	114	3604	3849
ex1010	10	10	4598	4618
pdc	16	40	4575	4631
s38417	29	106	6406	6541
s38584.1	38	304	6447	6789
clma	62	82	8383	8527

IV. RESULTS

In this section, the benchmark results of the previously described placement algorithm are presented. VPR is used as reference for the comparison of the placement results, as well as for the routing and timing analysis.

All used MCNC benchmarks [10] and their characteristics, namely, the number of Configurable Logic Blocks (CLBs), input blocks, output blocks and the sum of all blocks are listed in Table I, sorted by ascending complexity (i.e., the global block count). The netlists are placed on a homogeneous island-style architecture with four input lookup tables.

A. Bounding-Box Costs

The standard metric used for the approximation of the quality of a placement in VPR is the bounding-box cost. It is basically the sum of the half perimeter of the bounding-boxes (i.e., length plus width) of all nets. As introduced by Betz and Rose in [9], the bounding-box metric can be described as

$$Cost = \sum_{n=1}^{N_{nets}} q(n) \left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)} \right], \tag{5}$$

where $bb_x(n)$ and $bb_y(n)$ describe the horizontal and vertical size of the net n. $C_{av,x}(n)$ and $C_{av,y}(n)$ describe the average capacity of horizontal and vertical channels in the region of the net (in the considered case, the capacity is homogeneous over the whole architecture, so these values are constant. $q(n)$ corrects the effort for nets with more than three nodes, because it would otherwise be approximated to low.

In Table II, the bounding-box costs for the previously introduced benchmark netlists are presented. The results of VPR and the gradient placer are shown as absolute values and in relation to each other:

$$Cost_{Relative} = \frac{Cost_{VPR}}{Cost_{Gradient}} \cdot 100\,\% \tag{6}$$

TABLE II. COMPARISON OF THE BOUNDING-BOX COSTS BETWEEN THE GRADIENT PLACEMENT AND THE SIMULATED ANNEALING OF VPR

Netlist	VPR	Gradient	Relative / %
ex5p	180.599	173.701	96.18
tseng	102.398	101.112	98.74
apex4	195.338	190.657	97.60
misex3	200.456	199.160	99.35
alu4	204.692	200.965	98.18
diffeq	155.531	156.375	100.54
dsip	199.845	179.254	89.70
seq	260.789	267.686	102.64
apex2	280.120	293.168	104.66
s298	225.344	217.479	96.51
des	257.643	268.889	104.36
bigkey	209.470	201.344	96.12
frisc	587.227	593.630	101.09
spla	628.155	672.990	107.14
elliptic	497.645	503.854	101.25
ex1010	684.798	720.589	105.23
pdc	939.813	976.890	103.95
s38417	687.198	784.862	114.21
s38584.1	684.220	774.451	113.19
clma	1502.330	1598.670	106.41
Average			101.85

It can be seen that especially the smaller netlists profit from the gradient placement. Remarkably, for all netlists with less than 1600 nodes, the bounding-box costs are less with the gradient placer than with VPR. If the larger netlists are included, the costs for the gradient placer are only 1.85 percent higher on average, which is almost equal.

B. Channel Width

After their generation, the placements were routed with the VPR router and the Channel Width (CW), as well as the amount of necessary wire elements as a measure for the total Wire Length (WL) were determined. The results are shown in Table III. The differences in the channel width are given as a simple delta between the results:

$$\Delta CW = CW_{VPR} - CW_{Gradient} \tag{7}$$

The differences in the wire length are given as ratio between the results in percent:

$$WL_{Relative} = \frac{WL_{VPR}}{WL_{Gradient}} \cdot 100\,\% \tag{8}$$

The needed channel width of the gradient method is on average 0.5 channels smaller than the reference, whereas its total wire length is 0.09 percent longer. Both values are considered to be almost equal to the reference.

C. Runtime

In the previous sections, it was shown that the gradient placer produces a similar placement quality as VPR in regard of the bounding-box cost and the required routing resources. In this section, the runtime of both algorithms is measured and evaluated. The configuration of the system that has been used for the benchmarking is provided in Table IV.

The results are shown in Table V. The presented numbers are each an average of ten measurements. All single measurements varied less than two percent of the average of the measurement series.

TABLE III. COMPARISON OF THE MINIMAL CHANNEL WIDTH (CW) AND THE TOTAL WIRE LENGTH (WL) BETWEEN THE GRADIENT BASED PLACEMENT ALGORITHM AND THE SIMULATED ANNEALING OF VPR

Netlist	VPR		Gradient		Relative	
	CW	WL	CW	WL	ΔCW	WL / %
ex5p	15	20034	14	19541	-1	97.54
tseng	8	10200	7	9463	-1	92.77
apex4	15	22215	13	22116	-2	99.55
misex3	13	21884	12	21820	-1	99.71
alu4	12	22319	11	21261	-1	95.26
diffeq	9	15369	8	15292	-1	99.50
dsip	7	18065	7	15260	0	84.47
seq	12	28469	13	28977	1	101.78
apex2	12	30826	12	31905	0	103.50
s298	8	22335	9	21801	1	97.61
des	9	28084	9	28764	0	102.42
bigkey	8	21424	7	20315	-1	94.82
frisc	17	63146	14	64220	-3	101.70
spla	16	68364	16	72288	0	105.74
elliptic	11	44742	12	51127	1	114.27
ex1010	13	71891	12	73653	-1	102.45
pdc	19	104065	19	106057	0	101.91
s38417	8	64626	9	68999	1	106.77
s38584.1	10	64626	9	64180	-1	99.31
clma	14	141660	14	142695	0	100.73
Average					-0.5	100.09

TABLE IV. CONFIGURATION OF THE SYSTEM THAT HAS BEEN USED FOR THE BENCHMARKING OF THE GRADIENT ALGORITHM AND VPR

Property	Value
Processor	Intel® Core™ i7-4510U
Cores	2
Threads	4
Base Frequency	2.00 GHz
Turbo Frequency	3.10 GHz
Cache	4 MB
RAM	16 GB

On average, the gradient based placement algorithm needs less than half of the time of the simulated annealing placer of VPR. Furthermore, the ratio is even better for large netlists, as can be seen clearly in Figure 6. For example, the largest netlist in this benchmark series – the clma netlist – is placed 3.8 times faster with the gradient based approach.

V. CONCLUSION AND FUTURE WORK

In this work, a fast approach for netlist placement based on a gradient descent was presented. The gradient placer was compared to the simulated annealing based placer of VPR. It has been shown that the quality of the placement in regard of the bounding-box cost and the occupation of routing resources (i.e., channel width and total wire length) is equal to the reference within a reasonable margin of error, as proven by placing twenty prominent benchmarking netlists of different complexity. Notably, the presented approach is shown to be up to 3.8 times faster than the reference. On average, it needs less than half of the time to compute the result. However, preliminary results show that the resulting length of the critical path is worse with the gradient placer (about twenty percent for the largest netlist in this work). This would need to be addressed in future work.

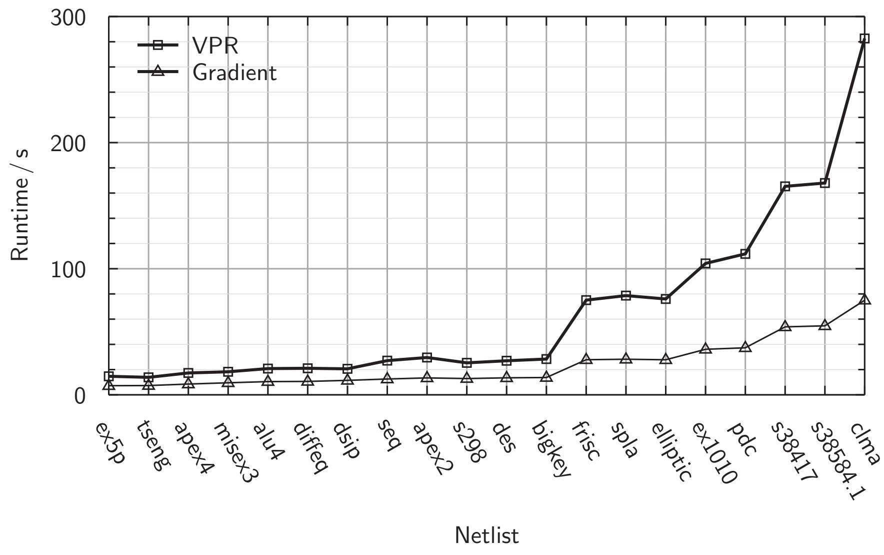


Figure 6. Diagram of the runtime as average of ten measurements between the gradient based placement algorithm and the simulated annealing of VPR.

TABLE V. COMPARISON OF THE RUNTIME AS AVERAGE OF TEN MEASUREMENTS BETWEEN THE GRADIENT BASED PLACEMENT ALGORITHM AND THE SIMULATED ANNEALING OF VPR

Netlist	VPR / s	Gradient / s	Relative / %
ex5p	14.69	7.23	49.23
tseng	13.86	7.34	53.00
apex4	17.34	8.53	49.16
misex3	18.27	9.54	52.20
alu4	20.81	10.48	50.36
diffeq	21.05	10.63	50.47
dsip	20.62	11.45	55.54
seq	27.12	12.53	46.21
apex2	29.60	13.41	45.31
s298	25.35	12.90	50.90
des	27.01	13.48	49.92
bigkey	28.36	13.72	48.36
frisc	75.10	27.83	37.05
spla	78.67	28.21	35.85
elliptic	76.02	27.79	36.56
ex1010	104.21	36.11	34.65
pdc	111.76	37.30	33.37
s38417	165.32	53.89	32.60
s38584.1	167.96	54.72	32.58
clma	282.60	75.04	26.55
Average			43.49

As the current implementation of the gradient placer is executed only single-threaded, the next logic step would be to parallelize its execution to make it even faster. The calculation of the gradients could be executed in parallel on node level, and even large parts of the legalization (e.g., the assignment of nodes to the regions) could be parallelized. Hence, a multi-threaded implementation would be beneficial and even a GPU-computing approach seems to be promising.

Even though the gradient placement approach was shown to be comparably fast for large netlists, a more recent set of benchmarks like the one included in [13] – containing much larger netlists – could be used to underline the scalability of the approach.

REFERENCES

[1] T. Kohonen, Self-Organizing Maps. Springer, 1995.

[2] T. Bostelmann and S. Sawitzki, “Improving FPGA placement with a self-organizing map,” in International Conference on Reconfigurable Computing and FPGAs (ReConFig), December 2013, pp. 1–6.

[3] T. Bostelmann and S. Sawitzki, “Improving the performance of a SOM-based FPGA-placement-algorithm using SIMD-hardware,” in The Ninth International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS), July 2016, pp. 13–15.

[4] T. Bostelmann, P. Kewisch, L. Bublies, and S. Sawitzki, “Improving FPGA-placement with a self-organizing map accelerated by GPU-computing,” International Journal On Advances in Systems and Measurements, vol. 10, no. 1 & 2, 2017, pp. 45–55.

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” Science, vol. 220, May 1983, pp. 671–680.

[6] L. Ingber, “Adaptive simulated annealing (ASA): Lessons learned,” Control and Cybernetics, vol. 25, 1996, pp. 33–54.

[7] M. M. Atiqullah, “An efficient simple cooling schedule for simulated annealing,” in International Conference on Computational Science and Its Applications (ICCSA). Springer, 2004, pp. 396–404.

[8] E. Vansteenkiste, S. Lenders, and D. Stroobandt, “Liquid: Fast placement prototyping through steepest gradient descent movement,” in 2016 26th International Conference on Field Programmable Logic and Applications (FPL), August 2016, pp. 1–4.

[9] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” in International Conference on Field Programmable Logic and Applications (FPL). Springer, 1997, pp. 213–222.

[10] S. Yang, “Logic synthesis and optimization benchmarks user guide version 3.0,” Microelectronics Center of North Carolina, Tech. Rep., 1991.

[11] M. Gort and J. H. Anderson, “Analytical placement for heterogeneous FPGAs,” in 22nd International Conference on Field Programmable Logic and Applications (FPL), August 2012, pp. 143–150.

[12] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015, pp. 1–15.

[13] J. Luu et al., “VTR 7.0: Next generation architecture and CAD system for FPGAs,” ACM Trans. Reconfigurable Technol. Syst., vol. 7, no. 2, June 2014, pp. 6:1–6:30.

60 GHz Low-Noise Amplifier in a 70 nm GaAs m-HEMT Technology for Multi-band Impulse Detection System

Pape Sanoussy Diao, Thierry Alves, Benoît Poussot and Martine Villegas

Université Paris-Est, ESYCOM (FRE2028), CNAM, CNRS, ESIEE Paris, Université Paris-Est Marne-la-Vallée
F-77454 Marne-la-Vallée, France
Email: pape-sanoussy.diao@esiee.fr, thierry.alves@esiee.fr

***Abstract*—In this paper, we present a 60 GHz Low-Noise Amplifier (LNA) to improve the performance of multi-band detection systems. The LNA is designed in 70 nm GaAs metamorphic High Electron Mobility Transistor (m-HEMT) technology and occupies an area of 1.47 x 1.0 mm^2. The inductive degeneration technique is used for a suitable trade-off between gain and noise. The three-stage LNA achieves a gain of 14.3 dB and a noise factor of 2.1 dB at 60.2 GHz, while consuming 13.5 mW. The simulated non-linear characteristics show an IP_{1dB} (Input 1 dB compression Point) of -9.6 dBm and an $IIP3$ (Input third-order Intercept Point) of -4.8 dBm.**

Keywords–LNA; 60 GHz; GaAs m-HEMT; Millimeter wave technology; Multi-band detection system.

I. INTRODUCTION

Advances in integrated circuit technologies are generating great interest in the evolution of standards in millimeter-wave bands. The availability of unlicensed bandwidth around 60 GHz in several regions of the world (57-66 GHz in Europe) is a real opportunity for new systems development and frequency harmonisation. Advances in SiGe [1] [2] and III-V [3]- [5] technologies nowadays allow the production of devices and systems for a variety of applications in millimeter-wave bands. These opportunities create a need for increasingly high-performance devices. In this way, this work adresses the design of a wideband amplifier, wich has low-noise, low power consumption and is small in size in order to improve the performance of detection systems.

This study is part of the development of an Ultra-WideBand (UWB) millimeter-wave detection system for short-range applications. We consider the monostatic radar context, Figure 1, with a cylindrical metallic target of radius r and height h $(r; h)$. The detection system is schematized by a transceiver (TX-RX) using the same antenna. The incidence angle θ is determined by the orientation of the target with respect to the antenna boresight.

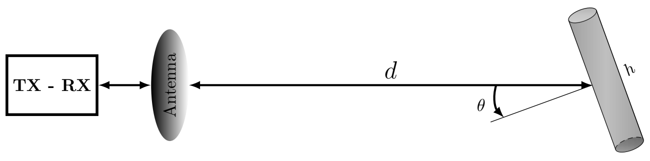


Figure 1. Context of the detection

This paper proposes the design of a low-noise amplifier with the 70 nm GaAs m-HEMT technology from OMMIC. It is structured as follows: Section II presents the principle of detection. Section III presents the sizing of the system and LNA specifications. Choice and technology description are presented in Section IV. Then, the design of the circuit is detailed in Section V. In Section VI, we present the results of the post-layout electromagnetic simulations in comparison with those in the literature before concluding in Section VII.

II. PRINCIPLE OF DETECTION

The angular dependence of the Radar Cross Section (RCS) of objects often results in a limitation of the detection range when moving away from the normal incidence. To overcome this limitation, we use frequency diversity [6]. The proposed detection principle is then based on the impulse technique, using a multi-band approach [7] [8], to improve detection coverage, particularly the continuity of detection according to the target orientation angle θ. Frequencies around 60 GHz were chosen for spectrum availability, but also for short wavelengths (5 mm at 60 GHz) to detect small objects (< 10 cm). The architecture associated with the detection principle is shown in Figure 2 for a dual-band system at 57.8 GHz and 62.8 GHz. In the transmitter, Differential Structure Power Amplifiers (DSPAs) whose operation is based on the even mode rejection are used. DSPA$_1$ provides both signal division

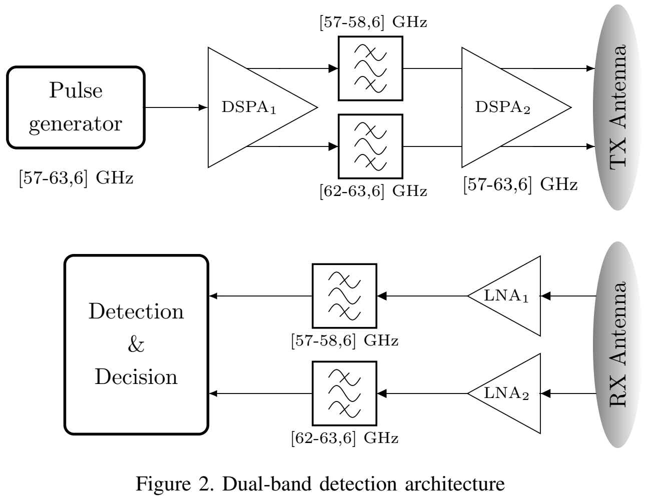


Figure 2. Dual-band detection architecture

and pre-amplification. The signals selected by the filter bank are then amplified by DSPA$_2$ and transmitted by the same antenna. In reception, simple LNAs are used for better noise performance. The received signals are selected by a filter bank identical to the one in transmission before being subjected to the detection and decision process. More details of the architecture operation are given in [8]. This architecture can be applied in a more general case with N bands. The processing of the received signals can be done by different techniques, such as: selection combining, cumulative detection [8], or non-coherent integration [9].

III. SYSTEM SIZING AND LNA SPECIFICATIONS

The sizing of the system is based on the monostatic radar equation expressing the maximum detection range (R_{max}) as a function of the minimum detectable power S_{min} [10]:

$$R^4_{max} = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 S_{min}} \quad (1)$$

where P_t is the transmitted power, G is the antenna gain, λ the operating wavelength and σ the RCS of the target.

Since S_{min} is related to the thermal noise power of the receiver and the minimum Signal-to-Noise Ratio (SNR) required to detect a target, the radar equation (1) can be written in the form:

$$R^4_{max} = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 \cdot kT\Delta f \cdot F \cdot SNR_r} \quad (2)$$

where k is the Boltzmann constant, T the temperature, Δf the receiver bandwidth, F its noise factor and SNR_r the required SNR at the output of the receiver to ensure detection.

In the equation (2), the transmitted power and the antenna gain are determined by standardization. The wavelength λ is chosen according to the application and the dimensions of the targets to be detected. The receiver bandwidth Δf is defined by the bandwidth of the front-end filter which will set the range resolution ΔR of the system ($\Delta R = c/2\Delta f$). The SNR_r is defined by the desired performance in terms of detection and false alarm probabilities. The proposed detection principle is based on the frequency and angle variations of the RCS. Thus, the receiver noise factor F is the only adjustable parameter to maximize system performance. Due to the position of the LNA in the receiver architecture, shown in Figure 2, the impact of its noise factor is more significant over that of the total RF chain according to the Friis equation:

$$F = F_1 + \frac{F_2 - 1}{G_1} + \frac{F_3 - 1}{G_1 G_2} + ... + \frac{F_n - 1}{G_1 G_2 ... G_{n-1}} \quad (3)$$

where F_i and G_i are the noise factor and the power gain, respectively, of the i-th stage, and n is the number of stages. By setting the objective of detecting a cylindrical target ($r =$ 0.6 cm; $h = 5.4$ cm), up to 2 m at normal incidence, we will determine the characteristics of the receiver stage and in particular those of the LNA. For this purpose, we consider a system with four bands in 57-66 GHz, distributed around the frequencies 57.8 GHz, 60.2 GHz, 62.8 GHz and 65.2 GHz. The output power of each channel of the DSPA$_2$ is set at 15 dBm (taking into account frequency bands standardization) and the gain of the antennas at 12 dBi (which can be achieved with 4 patches of 6 dBi each). The bandwidth of each band is 1.6 GHz and the SNR_r depends on the detection technique used. Filter losses are set at 3.5 dB [11]. For a conventional single-band configuration, the SNR_r to ensure the detection of a nonfluctuating target with a detection probability of 90% and a false alarm probability of 10^{-6} is 13.2 dB [10]. In the case of a non-coherent integration of 4 pulses in 4 sufficiently spaced frequency bands, the SNR_r is only 8.3 dB for the same detection and false alarm probabilities. Based on this case of non-coherent integration, using (2), we established the technical specifications of the LNA given in Table I, to ensure the targeted detection.

TABLE I. TECHNICAL SPECIFICATIONS OF THE LNA

Parameters	*Values*
Bandwidth BW	$\geq$ 1.6 GHz
Power gain G	$\geq$ 12.5 dB
Noise factor NF	$<$ 3 dB
S_{11} & S_{22}	< -10 dB

IV. CHOICE AND DESCRIPTION OF THE TECHNOLOGY

To realize the LNA, we have two design technologies: SG13S from IHP (Innovations for High Performance) Micro-electronics [12] and D007IH from OMMIC [13]. The choice of technology was first based on a study of passive elements. This revealed that SG13S is better suited for highly integrable components (small capacitors and resistors). On the other hand, it has very high losses ($\approx$ 0.7 dB/mm @ 60 GHz) for low-noise applications. Unlike SG13S, the D007IH has the advantage of lower losses ($\approx$ 0.22 dB/mm @ 60 GHz) and more integrable inductors in terms of shape. For example, Figure 3 shows simulation results of grounded transmission lines of the same characteristic impedance of 75 Ω. It can be seen that the D007IH has a quality factor (Q-factor) more than 5 times higher than that of the SG13S at 60 GHz for the same inductance of 268 pH.

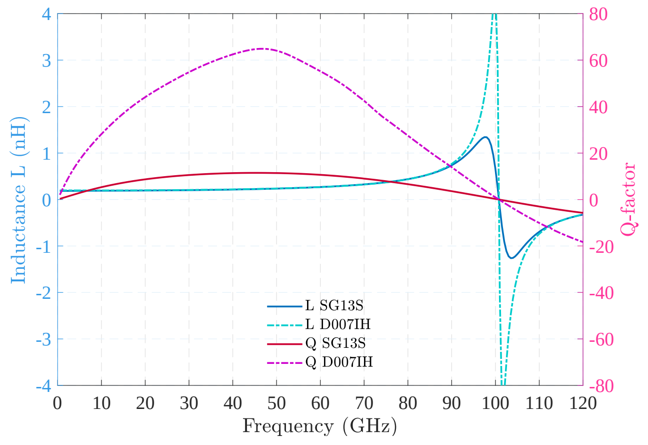


Figure 3. Comparison of grounded transmission lines

At high frequencies and especially at 60 GHz, the use of transmission lines is often preferred, so the D007IH seems more suitable for low noise applications. However, in microwaves, the overall performance of a circuit depends not only on the passive elements, but also on the characteristics of the transistors. Thus, the choice of technology must be based on an analysis of the global circuit. For this purpose, we compared single-stage amplifiers designed with both technologies, Table II. This comparison clearly shows that the m-HEMT is

TABLE II. SINGLE STAGE AMPLIFIERS COMPARISON

Technology	*Transistor*	*Gain (dB)*	*NF (dB)*	P_{DC} *(mW)*
SG13S	Bipolar	4.0	2.6	2.0
SG13S	MOS	3.1	2.6	9.2
D007IH	m-HEMT	3.9	1.2	4.1

the only one of the three transistor models that can meet the technical specifications of the LNA, particularly in terms of noise. To satisfy the gain performance, the use of a multi-stage structure will be necessary. In addition, with regard to sizing, it appeared that the noise factor is more significant over the gain because its influence is much greater in the range of the system. So, D007IH technology is chosen for the design of the LNA.

D007IH is a 70 nm gate length GaAs technology providing f_T/f_{max} of 300 GHz/450 GHz. It offers a depletion transistor m-HEMT with very high transconductance of $g_{m_{max}} = 1600$ mS/mm, that can support a voltage $V_{DS_{max}}$ of 3 V and a maximum current $I_{DSS_{max}}$ of 400 mA/μm. This type of transistor offers good performance in terms of noise, with a noise factor of only 0.5 dB at 30 GHz, giving it a privilege for security applications (millimeter-band imaging), telecommunications or radars. The process of the technology consists of a 3.5 μm metal in its underside, a 100 μm thick GaAs substrate above which different metallization levels can be distinguished. The most used metal layer for transmission lines realization is the IN metal with a thickness of 1.25 μm. It is also possible to associate this layer with a gold layer of the same thickness for less losses.

V. Circuit Design

The circuit is designed with Keysight Advanced Design System (ADS). The LNA consists of a three-stage structure using identical transistors. The size of the transistor is chosen so as to ensure a better trade-off between gain and noise. A parametric study allowed us to choose a transistor of 2 x 25 μm grid development. The optimal bias point then corresponds to a voltage $V_{DS} = 1\ V$ for an I_{DS} current of about 4.1 mA.

Unconditional stability has been ensured both by inductive degeneration of the source, but also by the use of a resistance in the bias circuit. This resistance allows stability at low frequency; its value must be chosen meticulously because it influences the gain and noise [14]. The degeneration also brings the circles of gain and noise closer together and thus facilitates the input matching [15].

The first two stages are almost identical and are matched with a good trade-off between gain and noise. Matching of the third stage is optimized in gain because its influence is less on the noise factor of the whole structure (3). The LNA's first stage schematic is presented in Figure 4.

The bias circuits are made with quarter-wave transmission lines. They include GaAs implanted resistors (R_D and R_G) for improving low frequency stability and by-pass capacitors (C_D) to short RF (Radio Frequency) leakage to the ground. The degeneration of the transistor source is ensured by the shorted transmission line TL$_S$. The capacitor C_1 and the transmission lines TL$_1$ and TL$_2$ form the input matching network. TL$_1$ is an open line smaller than $\lambda/4$ and therefore acts as a parallel capacitor. C_2, TL$_3$ and C_3 form the inter-stage matching and DC (Direct Current) isolation between stages 1 and 2. The same topology is used between stages 2 and 3. The output matching of the LNA is performed by a simple L-C series topology. The output inductor is achieved by a transmission line whose dimensions are optimized for a better gain, without degrading the noise factor, while respecting the design rules.

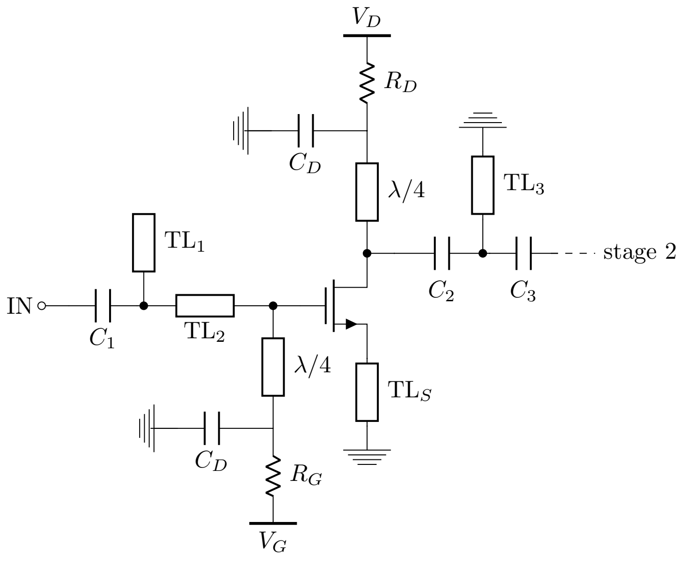


Figure 4. Schematic of the first stage of the LNA

VI. Post-Layout Simulations Results

The layout of the LNA is shown in Figure 5 and it includes RF and DC pads. All transmission lines are made with the same IN metal layer. Ground connections are made using vias holes. The size of the circuit is 1.47 x 1.0 mm^2.

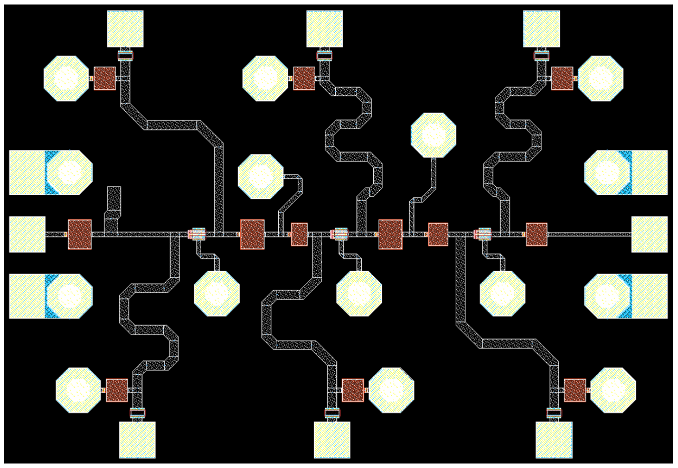


Figure 5. Layout of the LNA

For layout realization, small transmission lines have been added to transistor accesses (gate and drain) to reduce the coupling between components. The resistor in the drain bias drops the voltage at the DC pad from 1.1 V to 1 V at the transistor drain for each stage. Thus, the overall consumption of the circuit is approximately 13.5 mW. Layout simulations are done with Momentum Microwave. The layout was done in two main steps. Initially, electromagnetic simulations were carried out in a partial way, i.e., by considering each element separately (p index). Then, in a second step, we simulated the whole structure of the LNA (g index). The results obtained then present differences related to the coupling phenomena between the different elements of the LNA, which are not taken into account in the partial simulations of these elements. These couplings result in a more or less pronounced degradation of the circuit's performance. In our case, we were able to distinguish the degradation of reflection coefficients both at the input S_{11} and output S_{22}. Since then, we have re-optimized the layout in order to minimize these degradations, but also to improve its gain and noise performance, see Figure 6 and Figure 7.

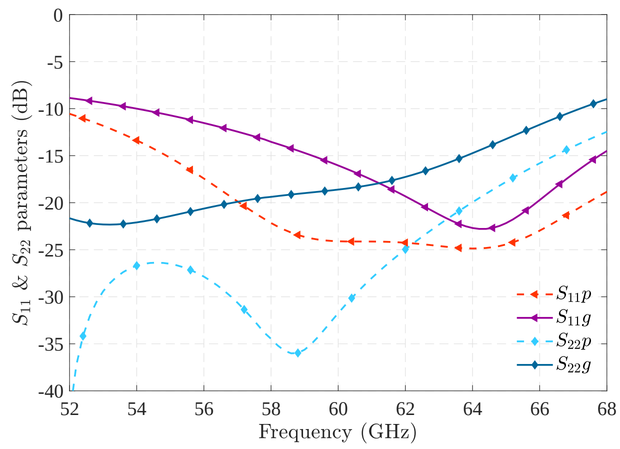


Figure 6. S_{11} and S_{22} with partial and global layout simulations

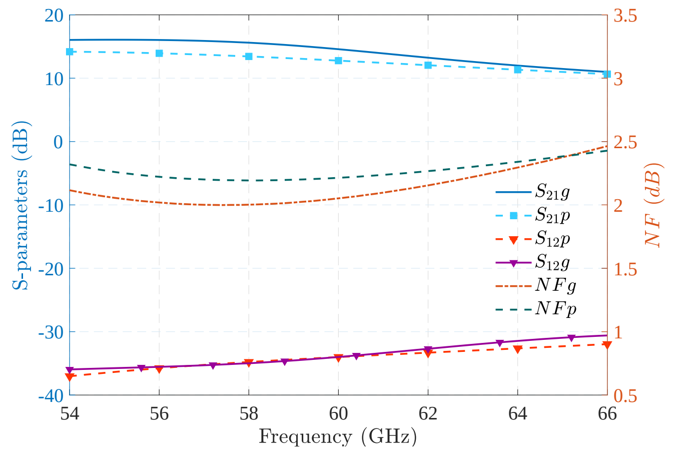


Figure 7. S_{21}, S_{12} and NF with partial and global layout simulations

A final optimization of the layout was done to reduce the size of the circuit which was initially 1.29 x 1.56 mm^2. This allowed us to obtain the layout seen in Figure 5. The optimized LNA thus offers unconditional stability over a wide frequency range, see Figure 8. Post-layout simulations results are presented in Figure 9. There is a gain of 14.3 dB and a noise factor of 2.1 dB at 60.2 GHz. The reverse isolation is about -34 dB at 60.2 GHz and the reflection coefficients S_{11} and S_{22} are less than -15 dB at 60.2 GHz. The 3 dB bandwidth (for $S_{11} < -10$ dB and $S_{22} < -10$ dB) ranges from 54 to 62.5 GHz with a fluctuation of about 0.17 dB in noise factor. The non-linear characteristics of the LNA are shown in Figure 10 and Figure 11. The simulated input 1 dB compression point of the LNA is $IP_{1dB} = -9.6$ dBm, while the input third order intercept point is $IIP3 = -4.8$ dBm.

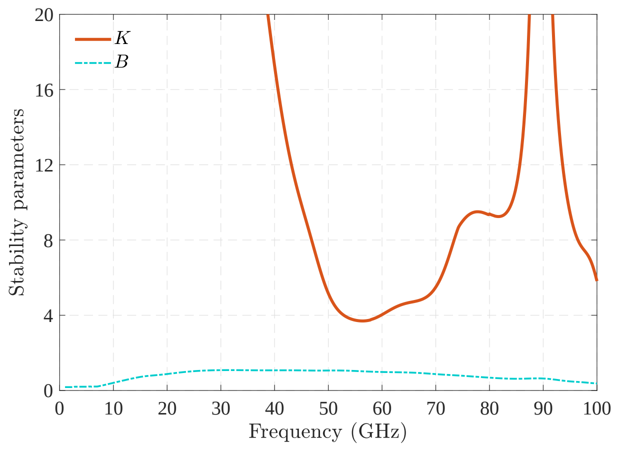


Figure 8. Stability parameters K and B

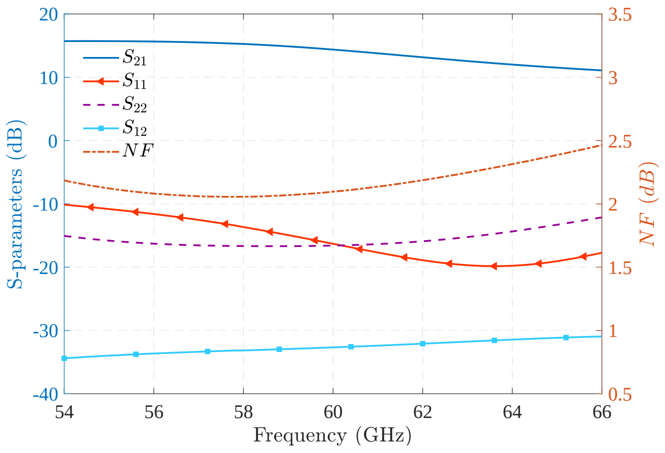


Figure 9. S-parameters and noise factor

The performance parameters of the designed LNA compared with other LNAs in the state-of-the-art are given in Table III. Our LNA shows good performance, especially in terms of noise factor, even if some of the results in the Table III are based on measurements. With a moderate power consumption compared to same type of technologies [4] [14] or even the 40 nm CMOS (Complementary Metal Oxide Semiconductor) [16], it presents a good gain to meet the targeted detection objectives. More gain can be achieved by increasing the drain voltage of the output stage or adding a fourth transistor. This would increase the power consumption and a little more the noise factor. The non-linear characteristics of the designed LNA are much better than those of the m-HEMT and CMOS technologies presented in Table III. For

TABLE III. PERFORMANCE COMPARISON

Ref.	*Technology*	*Freq. (GHz)*	*Gain (dB)*	*NF (dB)*	*IP_{1dB} (dBm)*	*IIP3 (dBm)*	*P_{DC} (mW)*	Area (mm^2)
[2]	130 nm SiGe BiCMOS*	57-66	20.5	4.3	-17,8	-11,1	9.8	0.41 x 0.32
[4]	50 nm GaAs m-HEMT$^+$	60-90	27	2.6	-26	-	45	1.6 x 2.3
[14]	100 nm GaAs m-HEMT$^+$	60-90	19	2.5	-	-	56	3.5 x 1.0
[16]	40 nm CMOS$^+$	60	12.5	3.8	-	-15	20.4	0.63 x 0.31
[17]	65 nm CMOS$^+$	60	23	4	-26	-	8	0.35 x 0.14
[18]	90 nm CMOS$^+$	58-77	11.2	4.8	-18.7	-7,4	10	0.72 x 0.76
[19]	65 nm CMOS$^+$	60	20.2	5.2	-25	-	28	0.54 x 0.80
[20]	65 nm LP CMOS*	61	22	5.5	-	-10,7	26	0.71 x 0.46
This work	**70 nm GaAs m-HEMT***	**60**	**14.3**	**2.1**	**-9.6**	**-4.8**	**13.5**	**1.47 x 1.0**

$^+$ Measures
$*$ Simulations

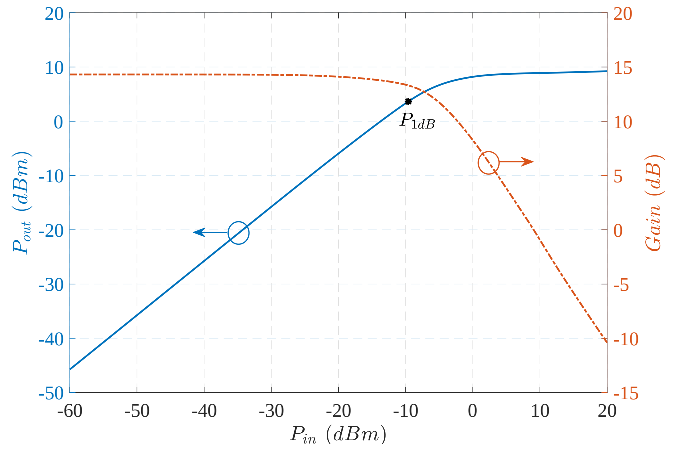


Figure 10. Gain and 1 dB compression point

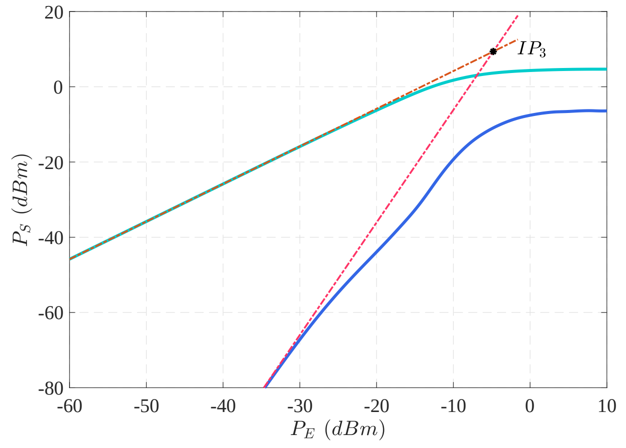


Figure 11. Three order intercept point (IP3)

example, its IP_{1dB} is -9.6 dBm, while it is less than -18 dBm for [4] [17]- [19]. It is the same for input three order intercept point ($IIP3$). The LNA occupies less space (1.47 x 1.0 mm^2) compared to the 50 and 100 nm GaAs m-HEMT technologies which occupy 2.3 x 1.6 mm^2 and 3.5 x 1.0 mm^2, respectively.

The performances thus obtained satisfy the specifications established in Table I, and allow, with the multi-band system, to detect up to 2.3 m the considered target (metallic cylinder of radius $r = 0.6$ cm and height $h = 5.4$ cm) at normal incidence, with a non-coherent integration. This represents an improvement of 30% in range compared to the conventional single-band detection system. In addition, by referring to [8], the overall detection coverage is also improved.

VII. Conclusion

A 60 GHz LNA designed in 70 nm GaAs m-HEMT technology was presented. The design was done with ADS Keysight in D007IH technology from OMMIC. Inductive degeneration of the source and inserting resistance in the bias circuit was used to better scale the transistor with a good trade-off between gain and noise, while ensuring unconditionnal stability. Our design was compared to other recently published milimeter-wave LNAs. Post-layout electromagnetic simulation results with momentum microwave show good performance, especially in terms of noise. With a noise factor of 2.1 dB at 60.2 GHz, our LNA is much better that those commonly found in the state-of-the-art. For a moderate power consumption of 13.5 mW, which is relativeley low for III-V's technologies, it presents 14.3 dB of gain at 60.2 GHz. The reflection coefficients of the designed LNA are less than -10 dB in 54-68 GHz. The input power at 1 dB compression point is $IP_{1dB} = -9.6$ dBm and the input third order intercept point is $IIP3 = -4.8$ dBm. The results of the designed LNA show the potential of III-V's technologies, especially the 70 nm GaAs m-HEMT for very low noise applications, particularly to improve the performance of detection systems.

References

[1] A. C. Ulusoy et al., "A SiGe D-Band Low-Noise Amplifier Utilizing Gain-Boosting Technique," IEEE Microwave and Wireless Components Letters, vol. 25, no. 1, 2015, pp. 61–63.

[2] M. Pallesen, "Design of a 60 GHz Low Noise Amplifier in a 0.13 μm SiGe BiCMOS Process," Master's thesis, The University of Bergen, 2016, URL: http://bora.uib.no/handle/1956/12595 [accessed: 2017-10-24].

[3] A. Dyskin, D. Ritter, and I. Kallfass, "Ultra wideband cascaded low noise amplifier implemented in 100-nm GaAs metamorphic-HEMT technology," in Proceedings of the International Symposium on Signals, Systems, and Electronics (ISSSE) Oct. 3–5, 2012, Potsdam, Germany. IEEE, Dec. 2012, pp. 1–4.

[4] P. M. Smith et al., "A 50 nm MHEMT millimeter-wave MMIC LNA with wideband noise and gain performance," in Proceedings of the IEEE MTT-S International Microwave Symposium (IMS2014) June 1–6, 2014, Tampa, FL, USA. IEEE, Jul. 2014, pp. 1–4.

[5] Y. Chen et al., “OMMIC 70 nm mHEMT LNA design,” in Proceedings of the IEEE Asia Pacific Microwave Conference (APMC) Nov. 13–16, 2014, Kuala Lumpar, Malaysia. IEEE, Jan. 2018, pp. 1192–1195.

[6] D. K. Barton, Frequency Diversity Theory. Artech House Inc., 1977, vol. 6 of Radars, section 2, pp. 35–114, in Frequency Agility and Diversity, ISBN: 0-89006-067-3.

[7] P. S. Diao, T. Alves, B. Poussot, and M. Villegas, “A new method and transceiver architecture dedicated to continuous detection of very small metallic object,” in Proceedings of the 10^{th} Global Symposium on Millimeter-Waves (GSMM) May 24–26, 2017, Hong Kong, China. IEEE, Jul. 2017, pp. 169–171.

[8] P. S. Diao, T. Alves, M. Villegas, and B. Poussot, “Compact millimeter wave architecture dedicated to object detection using dual band-dual polarization and impulse method,” in Proceedings of the 13^{th} Conference on Ph.D. Research in Microelectronics and Electronics (PRIME) June 12–15, 2017, Giardini Naxos, Italy. IEEE, Jul. 2017, pp. 161–164.

[9] P. Surendran, J.-H. Lee, and S. J. Ko, Performance of Non-coherent Detectors for Ultra Wide Band Short Range Radar in Automobile Applications. Springer-Verlag Berlin Heidelberg, 2012, vol. 377, pp. 185–195 in Software Engineering Research, Management and Applications 2011, ISBN: 978-3-642-23201-5.

[10] M. I. Skolnik, The Radar Equation, 2nd ed. McGraw Hill, Inc., 1980, chapter 2, pp. 15–67, in Introduction to Radar Systems, ISBN: 0-07-057909-1.

[11] R. Abdaoui, M. Villegas, G. Baudoin, and A. Diet, “Microstrip band pass filter bank for 60 GHz UWB impulse radio multi band architectures,” in Proceedings of the IEEE MTT-S International Microwave Workshop Series on Millimeter Wave Integration Technologies Sept. 15–16, 2011, Sitges, Spain. IEEE, Oct. 2011, pp. 192–195.

[12] “SG13S Process Specification Rev. 1.06,” July 2016, URL: https://www.ihp-microelectronics.com/en/services/mpw-prototyping/sigec-bicmos-technologies.html [accessed: 2017-10-18].

[13] “D007IH Design Manual - OM-CI/008/MG,” Oct. 2017, URL: http://www.ommic.fr/site/mpw-4r [accessed: 2018-07-20].

[14] A. Bessemoulin, J. Grunenputt, P. Felton, A. Tessmann, and E. Kohn, “Coplanar W-band low noise amplifier MMIC using 100-nm gate-length GaAs PHEMTs,” in Proceedings of the 34^{th} European Microwave Conference Oct. 12–14, 2004, Amsterdam, The Netherlands, vol. 1. IEEE, 2005, pp. 25–28.

[15] S. P. Voinigescu et al., “A scalable high-frequency noise model for bipolar transistors with application to optimal transistor sizing for low-noise amplifier design,” IEEE Journal of Solid-State Circuits, vol. 32, no. 9, 1997, pp. 1430–1439.

[16] H. Gao et al., “A 4861 GHz LNA in 40-nm CMOS with 3.6 dB minimum NF employing a metal slotting method,” in Proceedings of the IEEE Radio Frequency Integrated Circuits Symposium (RFIC) May 22–24, 2016, San Francisco, CA, USA. IEEE, Jul. 2016, pp. 154–157.

[17] E. Cohen, O. Degani, and D. Ritter, “A wideband gain-boosting 8 mW LNA with 23 dB gain and 4 dB NF in 65 nm CMOS process for 60 GHz applications,” in Proceedings of the IEEE Radio Frequency Integrated Circuits Symposium June 17–19, 2012, Montreal, QC, Canada. IEEE, Jul. 2012, pp. 207–210.

[18] Y.-S. Lin, C.-Y. Lee, and C.-C. Chen, “A 9.99 mW low-noise amplifier for 60 GHz WPAN system and 77 GHz automobile radar system in 90 nm CMOS,” in Proceedings of the IEEE Radio and Wireless Symposium (RWS) Jan. 25–28, 2015, San Diego, CA, USA. IEEE, 2015, pp. 65–67.

[19] C. So and S. Hong, “60 GHz variable gain LNA with small NF variation,” in Proceedings of the IEEE International Symposium on Radio-Frequency Integration Technology (RFIT) 30 Aug.–1 Sept., 2017, Seoul, South Korea. IEEE, Sep. 2017, pp. 171–173.

[20] A. Wang, L. Li, and T. Cui, “A transformer neutralization based 60 GHz LNA in 65 nm LP CMOS with 22 dB gain and 5.5 dB NF,” in Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS2013) May 19–23, 2013, Beijing, China. IEEE, Aug. 2012, pp. 1111–1114.

A Convolutional Neural Network Accelerator for Power-Efficient Real-Time Vision Processing

Junghee Lee
School of Cybersecurity
Korea University, Seoul, Korea
Email: j_lee@korea.ac.kr

Chrysostomos Nicopoulos
Department of Electrical and Computer Engineering
University of Cyprus, Nicosia, Cyprus
Email: nicopoulos@ucy.ac.cy

***Abstract*—Deep Convolutional Neural Networks (CNN) constitute a promising framework for many applications. Such networks are often employed for vision processing algorithms, because CNNs offer better accuracy than traditional signal processing algorithms. However, it is challenging to apply high-accuracy deep CNNs for *real-time* vision processing, because they require high computational power and large data movement. Since general-purpose processors do not efficiently support CNNs, various hardware accelerators have been proposed. While it is required to support all the layers of the CNN for real-time vision processing, the large amount of weights (more than 100s of MB) limit the speedup of hardware acceleration, because the performance is largely bounded by memory access times. Recent CNN architectures, such as SqueezeNet and GoogLeNet, address this problem by employing narrow layers. However, their irregular architecture necessitates a re-design of hardware accelerators. In this paper, we propose a novel hardware accelerator for advanced CNNs aimed at realizing real-time vision processing with high accuracy.**

Keywords–Convolutional Neural Network; Hardware Accelerator; Scheduling.

I. Introduction

As unmanned vehicles and robotics keep evolving, there is a growing demand for power-efficient real-time vision processing. While deep Convolutional Neural Networks (CNN) offer high accuracy and are applicable to various vision processing algorithms, they are very challenging to employ for *real-time* vision processing, because of their high demand on computation and data movement. Various types of accelerators have been proposed based on Graphics Processing Units (GPU) [1], Multiprocessor Systems-on-Chip (MPSoC) [2], reconfigurable architectures [3], Field-Programmable Gate Arrays (FPGA) [4]–[6], in-memory computation [7], and dedicated hardware acceleration through Application Specific Integrated Circuits (ASIC) [8] [9].

A typical CNN architecture consists of a stack of convolutional and pooling layers, followed by classifier layers, as shown in Figure 1(a). To realize *real-time* vision processing, all layers of the CNN should run on an accelerator. Otherwise, the data transfer time between the host and the accelerator cancels out the acceleration in the computation itself. The challenge is in the processing of the classifier layer, where all neurons are fully connected. Award-winning high-accuracy CNNs (such as AlexNet [10], which won the 2012 ImageNet contest) usually require a huge number of weights (up to 100s of MB [7]) and weights are not reused.

This challenge is being addressed by recent CNN architectures. Two representative examples are SqueezeNet [11] and GoogLeNet [12]. SqueezeNet offers comparable accuracy to AlexNet, but it uses 510 times fewer weights. GoogLeNet took the first place in the 2014 ILSVRC Classification contest. GoogLeNet employs narrow layers to minimize the number of weights, while offering high accuracy by using a large number of such narrow layers (more than 100). As shown in Figures 1(b) and (c), the SqueezeNet [11] and GoogLeNet [12] architectures are not as regular as the traditional CNN architecture of Figure 1(a).

To realize real-time vision processing, all layers of the CNN should run on the accelerator seamlessly. For example, Eyeriss [13] [8] requires reconfiguration of the accelerator for each layer. It takes 0.1 ms to configure one layer. If there are 100 layers, it takes 10 ms only for reconfiguration. ShiDianNao [9] addresses this by using hierarchical finite state machines. However, it is not proven with large-scale CNNs, such as SqueezeNet and GoogLeNet. Approaches using GPUs and FPGAs can execute all layers of the CNN quickly, but they consume an order of magnitude more power than ASIC designs. DaDianNao [14] offers low latency for all the layers of large-scale CNNs, but it consumes as much power as an FPGA, which may not be suitable for power-efficient vision processing. In general, an FPGA-based design cannot simply be implemented in an ASIC to boost power efficiency, due to the fundamental differences in the underlying design principles. Since the FPGA is programmable, the design can typically be customized to suit a particular CNN. This customization is not feasible in an ASIC. To support advanced CNNs like SqueezeNet and GoogLeNet in ASIC for real-time vision processing, we need a flexible – yet power-efficient – design that does not require run-time reconfiguration.

The proposed accelerator aims to achieve this goal by employing ***data-driven scheduling*** and ***modular design***. These two key features constitute *the novel contributions of this work*, since they enable the handling of *advanced CNNs without the need for reconfiguration*. The operation and destination of a Processing Element (PE) is determined at run-time upon receipt of data. The data is accompanied by metadata indicating the meaning of the data. By interpreting the metadata, a PE determines its schedule at run-time, which makes it easier to handle irregular CNN architectures. To achieve scalability, a modular design concept is employed with no shared resources and global synchronization being assumed. Each PE can only access its own local memory, and communicates only with its neighbors. Modular design facilitates deep pipelining, which enables further latency improvements by increasing the clock frequency. As a result, it is demonstrated by experiments that the proposed accelerator executes all layers of SqueezeNet and GoogLeNet in 14.30 and 27.12 million cycles with 64 processing elements. Assuming a 1 GHz clock speed, these latencies correspond to 14.30 ms and 27.12 ms, respectively, which is comparable to high-performance FPGA-based approaches (range of 1.06 ms to 262.9 ms [5] [6]). It is estimated that the proposed accelerator consumes 2.47 W and 2.51 W for SqueezeNet and GoogLeNet, respectively, which may be higher than power-efficient ASIC-based approaches (consuming 0.278 to 0.320 W [13] [9]), but it is significantly lower than FPGA-based approaches (that consume 8 to 18.61 W [4]) and DaDianNao [14] (that consumes 15.97 W).

After discussing related works in Section II, we present

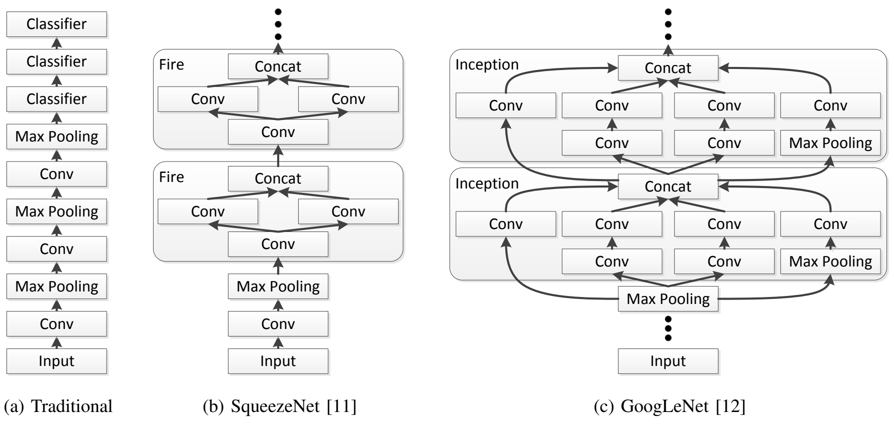


Figure 1. Three different types of CNN architectures. The left one represents the traditional (generic) approach, while the other two represent two existing state-of-the-art approaches.

the proposed accelerator in Section III, and the details of the employed data-driven scheduling in Section IV. Section V provides experimental results, and Section VI concludes the paper.

II. Background and Related Work

Research in neural networks has a long history. Over the last several years, various types of approaches for the acceleration of CNNs have been studied. There is a trade-off between latency and power consumption among these accelerators. The GPU approach achieves 0.19 ms latency at 227 W [1], while FPGAs offer a range of 1.06 ms to 262.9 ms at 8 W to 18.61 W [4]–[6]. These values are measured under AlexNet [10] or VGG-16 (Visual Geometry Group 16) [15]. On the contrary, dedicated hardware accelerators implemented in ASIC target power-efficient implementations of small-scale CNNs, or the convolutional layers of large-scale CNNs [9]. For example Eyeriss [8] executes the convolutional layers of AlexNet [10] in 115.3 ms at 0.278 W [13].

Compared to two state-of-the-art CNN accelerators, the proposed accelerator offers lower latency and better scalability with the number of processing elements and clock frequency. Compared to Eyeriss [8], the proposed accelerator offers significantly lower latency through its modular design (that allows for higher clock frequencies), weight prefetching (optimized memory access patterns to Dynamic Random Access Memory (DRAM)), and by using larger on-chip memory. Additionally, the data-driven scheduling enables seamless execution of all layers without reconfiguration. ShiDianNao [9] also supports seamless execution of all layers, by storing all weights and feature maps in on-chip memory. However, the ShiDianNao [9] architecture was evaluated only with small-scale CNNs whose weights and feature map sizes fit into on-chip memory. Furthermore, both Eyeriss [8] and ShiDianNao [9] employ global shared memory, which renders their scalability questionable. In contrast, the modular design concept of the architecture proposed in this work enables high clock frequencies through pipelining. Even though the proposed accelerator requires more hardware and memory space to accommodate its data-driven scheduling and modular design, it is still significantly more power-efficient than FPGA-based approaches.

III. Overview of the Proposed Accelerator

A. Functional Requirements

The current implementation of the proposed accelerator supports three types of layers, and four types of layer connections. The four layers are: (1) convolutional layer, (2) max pooling layer, and (3) average pooling layer. The classifier layer can be implemented as a special case of the convolutional layer. SqeezeNet and GoogLeNet still use the classifier layer, even though it is not as big as those in traditional CNNs.

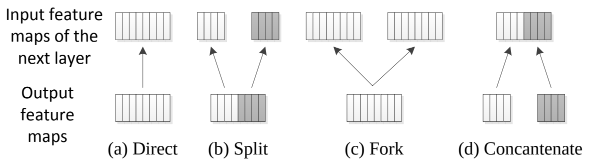


Figure 2. The 4 different types of layer connections supported by the proposed CNN accelerator that can be used to implement various CNN architectures.

To support a traditional/generic CNN, only one type of layer connection is enough, which is shown in Figure 2(a). To support more advanced CNN architectures, the proposed accelerator supports three other types of connections. The feature maps of a layer can be split and sent to different layers, as shown in Figure 2(b), and all feature maps can be sent to multiple layers, as shown in Figure 2(c). Finally, output feature maps of different layers can be concatenated as input feature maps of a layer, as shown in Figure 2(d).

The data-driven scheduling and modular design make it easy to support various types of layers and connections. Since the abovementioned three layers and four connections are enough to support SqueezeNet and GoogLeNet, the proposed accelerator only implements these for now, but it can be easily extended to cover other types of layers and connections. It is also possible to use heterogeneous PEs. These extension possibilities – and more – of the accelerator will be explored in our future work.

B. Architecture

For real-time vision processing, the speed of the feed-forward process is more important than that of the backward process, because the backward process is usually performed off-line during training. Thus, the proposed accelerator is focused on accelerating the feed-forward process.

Figure 3 illustrates the architecture of the proposed accelerator and presents the high-level details of one PE module. We assume that the accelerator is implemented as a separate chip. It receives inputs from and sends outputs to the host through

TABLE I. CONFIGURATION OF A LAYER TO BE STORED IN CONFIGURATION MEMORY.

Parameter	Description
R	Number of rows of an output feature map
C	Number of columns of an output feature map
M	Number of output feature maps
N	Number of input feature maps
K	Filter size
S	Stride
O	Number of next layers connected with this layer
T_n	The layer number of $n-th$ connected layer
F_n^{start}	Start feature map number of the $n-th$ connected layer
F_n^{end}	End feature map number of the $n-th$ connected layer
F_n^{shift}	Feature map number shift of the $n-th$ connected layer

a standard bus interface. It has its own main memory (e.g., DRAM), which is used to store weights.

The proposed accelerator consists of a number of PEs. All PEs are the same, but one of them is designated as an interface PE, which interacts with the host and memory. The PEs are connected by 1D rings. Two rings are used for data (activation) transfer, and the third ring is used for weight prefetching.

A PE consists of a communication interface, matching logic, functional units (multiplier and adder), an output Finite State Machine (FSM), and local memories for weights and feature maps. The matching logic determines whether the incoming activation is assigned to the PE or not. The matching logic makes a decision based on the mapping information, which is presented in the next section (subsection IV-A). If the incoming activation is accepted, it is pushed to a queue and processed by the functional unit. If the queue is full, the incoming activation cannot be accepted, even though it is destined to this PE. By interpreting the metadata accompanied by the activation, the corresponding functional unit is triggered. The result is stored in the local feature map memory, and transferred to other PEs when the computation is done.

IV. DATA-DRIVEN SCHEDULING

The heart of the proposed accelerator and its key novelty is *data-driven scheduling*. It enables the execution of advanced CNN architectures without reconfiguration. Each PE determines whether to accept an activation and the subsequent schedule of operations, based on metadata and the CNN's configuration. The metadata is accompanied by the activation coming from the interconnection network. The CNN configuration is transferred from the host through the interface PE, and stored in the local configuration memory.

Figure 4 shows examples of the metadata. The format of the metadata depends on the type of data. For example, for activations, the metadata includes the layer, feature map, and the position (row and column) of the activation. The position of an activation in the input feature map is denoted as `y` and `x`, that of a neuron in the output feature map is denoted as `row` and `col`, and that of a weight in a filter is denoted as `i` and `j` throughout this paper.

The configuration of layers is broadcasted to all PEs at initialization time, and it is stored in the local configuration memory of each PE. The configuration of one layer is shown in Table I.

The parameters R, C, M, N, K, and S are basic parameters of the CNN. Specifically, O and F are used to specify the connection, while F^{start} and F^{end} are used to support splits, and F^{shift} is used to support concatenation. For example, if a layer has 64 output feature maps, and 32 of them are sent to layer 1, and the remaining 32 are sent to layer 2, then O=2, T_0=1, F_0^{start}=0, F_0^{end}=31, F_0^{shift}=0, T_1=2, F_1^{start}=32, F_1^{end}=63, and F_1^{shift}=-32. In this case, F_1^{shift} is used to convert the feature map numbers 32–63 of the current layer to the feature map numbers 0–31 of the next layer. In a similar way, when feature maps of multiple layers are concatenated, the feature map numbers can be adjusted to become linear, by using the F^{shift} parameter.

A. Mapping

In the proposed accelerator architecture, the granularity of mapping is a feature map. A PE processes all neurons in its assigned feature maps. In this way, we can *avoid the sharing of weights among PEs*, which facilitates modular design. In other words, if a PE processes all the neurons of its assigned feature maps, it can store their weights in its local memory and other PEs do not need to access them.

Feature maps are assigned as a combination of input and output feature maps. As a toy example, let us suppose a layer has 2 input feature maps (`ifm0` and `ifm1`), and 2 output feature maps (`ofm0` and `ofm1`). If there are 2 PEs, one PE is assigned to `ifm0-ofm0` and `ifm1-ofm0`, and the other PE is assigned to `ifm0-ofm1` and `ifm1-ofm1`. In other words, each PE processes all input feature maps of its assigned output feature map. If there are 4 PEs, feature maps are spread out as PE0 to `ifm0-ofm0`, PE1 to `ifm1-ofm0`, PE2 to `ifm0-ofm1`, and PE3 to `ifm1-ofm1`. PE0 and PE1 produce partial sums of neurons for `ofm0`, and one of them must accumulate them. In the proposed accelerator, the PE processing the last input feature map of an output feature map is responsible to collect the partial sums from other PEs that are assigned to the same output feature map. In our toy example, PE0 should send its partial sums to PE1, so that PE1 can collect them and generate the final `ofm0`, while PE2 should send its partial sums to PE3, so that PE3 can generate the final `ofm1`.

To generalize this concept, we compute a feature map index for each combination of input and output feature maps, and a range of indices is assigned to PEs. The feature map index is computed as `index = ifm + ofm` × `M`, where `ifm` denotes the input feature map number, `ofm` is the output feature map number, and `M` is the total number of input feature maps. In the above toy example, the index of `ifm0-ofm0` is 0, `ifm1-ofm0` is 1, `ifm0-ofm1` is 2, and `ifm1-ofm1` is 3. If there are 2 PEs, PE0 is assigned to the range of indices from 0 to 1, and PE1 to indices from 2 to 3. If there are 3 PEs, PE0 is assigned to 0 and 1, PE1 to 2, and PE2 to 3. Thus, feature maps are not evenly distributed. If there are 4 PEs, each PE is assigned to each index.

The matching logic accepts an incoming activation, if its feature map falls within the range of the assigned indices. Recall that an activation is accompanied by metadata that includes the input feature map number, as shown in Figure 4. The pseudo code in Figure 5 shows how to determine if an activation, whose index is `ifm`, should be accepted or not, given a range of indices from `index_start` to `index_end`. Again, `M` indicates the total number of input feature maps.

Even if the activation is accepted, it should be forwarded to the next PE, because it may be used by the next PE. In fact, if there is a high enough number of output feature maps, as compared to the number of PEs, all PEs would need all input feature maps. Coming back to the toy example,

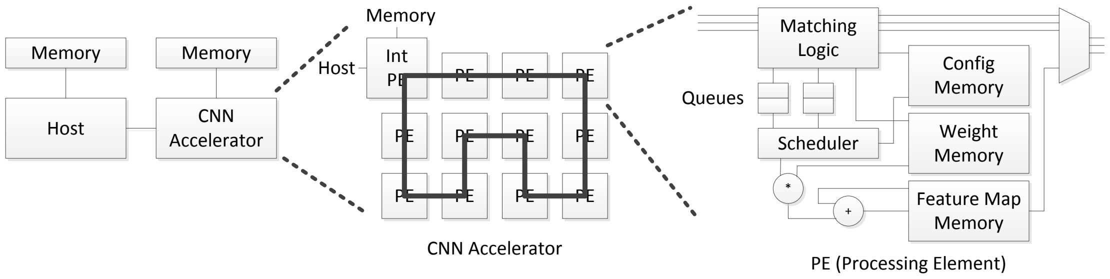


Figure 3. The architecture of the proposed accelerator and a high-level overview of one processing element. Note that the memory connected to the CNN accelerator on the leftmost diagram is connected only to the interface PE, i.e. it is not a shared memory. The pseudo codes of the 'Matching Logic' and the 'Scheduler' modules are presented, respectively, in Figure 5 and Figure 7.

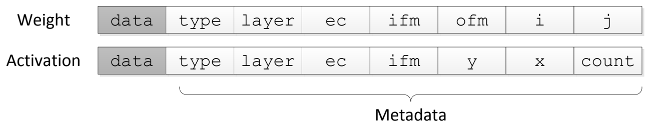


Figure 4. Examples of message formats, including the pertinent metadata. [ec: Escape channel; ifm: Input feature map number; ofm: Output feature map number.]

```
ofm_start =
   index_start % M <= ifm ?
   index_start / M : index_start / M + 1;
ofm_end =
   ifm <= index_end % M ?
   index_end / M : index_end / M - 1;
if(ofm_end >= ofm_start)
   activation accepted;
```

Figure 5. The pseudo code of the matching logic. The code determines if an activation should be accepted or not.

let us suppose there are 2 PEs. PE0 processes `ifm0-ofm0` and `ifm1-ofm0`, while PE1 processes `ifm0-ofm1` and `ifm1-ofm1`. Thus, both PE0 and PE1 need all input feature maps (`ifm0` and `ifm1`). Therefore, we designed the accelerator in such a way that activations are broadcast, and PEs determine if they are to be accepted. This is in contrast to sending activations to specific target destinations.

Due to resource constraints, an activation may not be accepted, even if it is destined to the particular PE. Because of this, we need to maintain two types of counters. One counter is to determine when the activation should be removed from the network. When the activation is injected into the network, the total number of output feature maps is attached to the metadata. Whenever a PE accepts the activation, it decrements this counter by the number of assigned output feature maps and forwards it to the next PE. When this counter reaches zero, it is no longer forwarded (i.e., it is removed from the network).

The other type of counter is for determining if the activation has already been accepted, or not. Because a ring is used as a communication fabric in the proposed accelerator, the same activation may arrive at the PE more than once, if it is not removed from the network. To check for this, a PE maintains a counter for each input feature map of a layer. The activations of an input feature map are accepted in a pre-determined order. In our implementation, all columns of a row are accepted in an increasing order of their column index, and those of the next rows are accepted in the same way. The counter counts how many activations of the input feature map have been accepted. Since activations are accepted in a specific order, if a PE knows how many have been accepted, the PE can determine

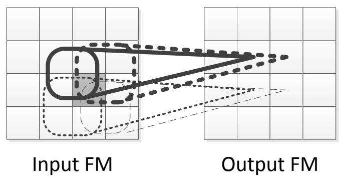


Figure 6. Illustration of how an activation is used for multiple filters.

what should come next. The activation is accepted only if the incoming activation is what the PE is expecting. In this way, the PE avoids accepting the same activation more than once.

In case of the max and average pooling layers, the number of input and output feature maps is always the same. An output feature map only needs one corresponding input feature map. Thus, those PEs that generate the final output feature map of the previous layer (which is the input feature map of the pooling layer) are assigned to process the corresponding output feature map of the pooling layer. In this way, we can eliminate unnecessary activation transfers.

B. Scheduling

Once an activation is accepted, all operations that need the activation are scheduled. To compute a neuron, its neighboring activations are required. The exact number of required activations depends on the size of a filter. In other words, an activation should be used by multiple filters.

Figure 6 shows an example. Let us suppose the filter size is 2 by 2 and the stride is 1. To compute a neuron at (1,1) of an output feature map, we need activations (neurons of input feature map) at (1,1), (1,2), (2,1), and (2,2). Similarly, neurons at (1,2), (2,1), and (2,2) of the output feature map need the same activation at (2,2) of the input feature map. If multiple output feature maps are assigned to the PE, neurons in other feature maps also need the incoming activation.

The pseudo code in Figure 7 shows how Multiply-And-Accumulate (MAC) operations are scheduled for an incoming activation. The `ofm_start` and `ofm_end` parameters are computed as shown in Figure 5. As shown in Figure 4, the position of the activation is given by `y` and `x`. The same mechanism is used for pooling layers. Instead of MAC operations, comparison (max pooling) or accumulation (average pooling) operations are scheduled.

The pseudo code is implemented as an FSM in the functional units. The FSM pops an activation from the queue located in-between the functional units and the matching logic in Figure 3. Once the FSM finishes all the scheduled operations, it pops the next activation from the queue. A functional unit

```
for(ofm=ofm_start; ofm<=ofm_end; ofm++)
  for(row=MIN(y/S, R-1); row>(y-K)/S && row>=0; row--)
   for(col=MIN(x/S, C-1); col>(x-K)/S && col>=0; col--) {
    i = y-row*S;
    j = x-col*S;
    feature_map[layer][ofm][row][col] +=
     weights[ofm][ifm][i][j] *
     activation
   }
```

Figure 7. The schedule of operations when an activation is accepted. [R: Number of rows of the output feature map; C: Number of columns of the output feature map; K: Filter size; S: Stride. All of the R, C, K, and S are of the current layer.]

TABLE II. THE DEFAULT SIMULATION PARAMETERS USED IN ALL EXPERIMENTS.

Parameter	SqueezeNet	GoogLeNet
Number of PEs	64	
Average memory access cycle	1	
Pipeline stages of communication channel	1	
Pipeline stages of functional units	1	
Queue depth	16	
Number of rings	3	
Configuration memory size	0.021 MB	0.092 MB
Weight memory size	1.289 MB	4.119 MB
Feature map memory size	9.132 MB	3.333 MB
Bit width of one activation ring	68	71
Bit width of the weight ring	58	61
Number of escape channels	10	46

accesses the weight memory and the feature map memory to perform its operation, and the result is stored in the feature map memory. To determine if accumulation is finished for one neuron, a counter is maintained for every neuron in the output feature map. The counter is stored in the feature map memory. The overhead of the memory will be discussed in Section V.

V. EVALUATION

A. Experimental Setup

We developed a cycle-level in-house simulator using SystemC [16]. The default simulation parameters are shown in Table II.

The proposed accelerator can take full advantage of the DRAM bandwidth, because the access pattern is always sequential. All feature maps are stored in the on-chip memory by adopting a sliding window technique, and the external DRAM is used only for weights. Since weights are prefetched in the order of layers, there is no need for random accesses to DRAM. Assuming the proposed accelerator runs at 1 GHz, then a 2 GB/s throughput is required to fetch one weight (16 bits) per cycle. According to the DDR4 standard, the maximum throughput can be up to 25.6 GB/s. Therefore, the DRAM throughput is high enough to easily supply one weight every cycle.

B. Performance Analysis

Table III shows the number of cycles required to execute *all layers* of SqueezeNet and GoogLeNet. Under the assumption that the proposed accelerator runs at 1 GHz (since ShiDianNao [9] also runs at 1 GHz), these results correspond to 14.30 ms and 27.12 ms for SqueezeNet and GoogLeNet, respectively.

Even though a direct comparison may not be meaningful due to fundamental differences in the design goals (low power vs. low latency) and benchmark (different CNNs), Eyeriss [13] is reported to execute the convolutional layers of AlexNet in 115.3 ms, and the convolutional layers of VGG-16 in 4309.5 ms. While a GPU executes all layers of these CNNs in 0.19 ms, FPGAs require 1.06 ms to 262.9 ms [1] [4]–[6]. The performance of the proposed accelerator is comparable to FPGA-based techniques. DaDianNao [14] offers even lower latency, but its power consumption is comparable to FPGA-based techniques. This is because it targets high-performance implementations supporting all the layers of large-scale CNNs and both the forward and backward processing steps.

TABLE III. NUMBER OF CYCLES REQUIRED TO EXECUTE ALL LAYERS OF THE CNN.

CNN	Number of cycles	Execution time*
SqeezeNet [11]	14,303,612	14.30 ms
GoogLeNet [12]	27,122,439	27.12 ms

* 1 GHz clock frequency is assumed.

TABLE IV. THE MAXIMUM SUPPORTED VALUES OF THE VARIOUS CNN CONFIGURATION PARAMETERS.

Parameter	Meaning	SqueezeNet	GoogLeNet
R	Rows	224	224
C	Columns	224	224
M	Input feature maps	1000	1000
N	Output feature maps	1000	1000
K	Filter size	7	7
S	Stride	2	2
O	Connections of a layer	2	4
T_n	Next layer	33	106
F_n^{start}	Start feature map	1000	1000
F_n^{end}	End feature map	1000	1000
F_n^{shift}	Feature map shift	1000	1000
Total number of layers		33	106
Total number of connections		40	204

It should also be noted that the proposed accelerator offers flexibility in that it can support SqueezeNet and GoogLeNet without run-time reconfiguration. Since SqueezeNet and GoogLeNet offer comparable accuracy with AlexNet and VGG-16, we believe they are good alternatives for power-efficient real-time vision processing.

On the other hand, ShiDianNao [9] reports 0.047 ms to execute all layers of ConvNN [17]. However, ConvNN is much smaller. For example, GoogLeNet requires 1502 million MAC operations, whereas ConvNN only needs 0.6 million. While it demonstrates an efficient implementation of small-scale CNNs, it is not proven with large-scale CNNs for high-accuracy vision processing algorithms.

C. Cost Analysis

To compute the *minimum* required memory size and the minimum required bit-width for the rings, it is essential to assess the *maximum* supported values of the parameters of the CNN configurations under investigation. These parameters are summarized in Table IV. The total number of layers used for the proposed accelerator is different from the number assumed in the original implementations of the CNN architectures. We slightly changed the architecture – in a mathematically equivalent manner – to better fit the underlying architecture of the accelerator. Specifically, instead of introducing an explicit concatenation layer, the output feature maps are directly connected to the next layer to reduce the memory requirement. Thus, if a pooling layer is followed by a concatenation layer, the pooling layer has to be split into the previous layers, because pooling layers are processed by the same PE where the output feature map is generated.

In the configuration memory, the basic parameters (R, C, M, N, K, S, and O) are stored for each layer and the connection parameters (T, F^{start}, F^{end}, and F^{shift}) are

TABLE V. THE MINIMUM REQUIRED MEMORY SIZES UNDER TWO DIFFERENT NUMBER REPRESENTATIONS.

Memory	SqueezeNet		GoogLeNet	
	16 bits	6 bits	16 bits	6 bits
Weight memory	1.289 MB	0.483 MB	4.119 MB	1.544 MB
Feature-map memory	9.132 MB	5.619 MB	3.333 MB	2.051 MB

stored for each connection. The total number of bits to required to store all of these is 2,793 and 12,106 for SqueezeNet and GoogLeNet, respectively. Since all PEs need to store them, the sum of the configuration memory size of all PEs is 0.021 MB and 0.092 MB for SqueezeNet and GoogLeNet, respectively, as shown in Table II.

The minimum size of the weight and feature-map memories varies for different PEs, depending on the feature map assignment. For regularity, we used the same memory size across all PEs. The proposed accelerator does not depend on the type of number representation. All analysis results shown so far is based on 16-bit fixed-point representation, which is the most popular setup in previous efforts. If, instead, we adopt 6-bit representation [18], the memory size can be further reduced. Table V shows both cases.

Obviously, the memory size required for the proposed accelerator is significantly larger than that of existing accelerators. This is because the design goal of the proposed accelerator is to minimize latency as much as possible at a reasonable hardware cost. Considering the fact that recent Intel processors employ 8 MB of L3 cache and multiple 256 KB L2 and 32 KB L1 caches and DaDianNao [14] has a 36 MB embedded on-chip DRAM, we believe that 10 MB of on-chip memory is affordable for a stand-alone hardware-based CNN accelerator.

D. Power Estimation

It is estimated that the power consumption of the proposed accelerator is similar to ShiDianNao [9], which consumes 320.10 mW (except for the memory power, which will be discussed shortly), assuming an operating frequency of 1 GHz. Both designs run at the same clock frequency, employ the same number of PEs (64), and use the same types of functional units (multipliers and adders). The overhead of the control logic would obviously be different, but according to the analysis in Eyeriss [13], the power consumption of the control logic corresponds to only 9.5% to 10.0% of the total power budget. In general, the biggest consumer of power is the on-chip memory. Since the proposed accelerator employs a significantly larger memory, it consumes more power than ShiDianNao, which has a 288 KB on-chip memory. By using the per-access energy model of CACTI [19] and the number of memory accesses obtained through simulation, the power consumption of both the on-chip memory and DRAM can be estimated. Including the power consumption of the other components reported by ShiDianNao, the total power consumption (including DRAM accesses) of the proposed accelerator is estimated as 2.47 W and 2.51 W for SqueezeNet and GoogLeNet, respectively. Despite the fact that these numbers are based solely on estimation, it is clear that the power consumption of the proposed accelerator is significantly lower than FPGA-based approaches (that consume 8 to 18.61 W) and DaDianNao's 15.97 W [14].

VI. CONCLUSIONS

This paper proposes a novel hardware-based accelerator for deep CNNs used to realize *power-efficient real-time* vision processing. This attribute is enabled by *modular design*, optimized memory access patterns due to *weight prefetching*, and larger on-chip memory. More importantly, the new accelerator can execute *all layers* of SqueezeNet and GoogLeNet in 14.30 ms and 27.12 ms, respectively, which are comparable to high-performance FPGA-based approaches, but with significantly lower power consumption at 2.47 W and 2.51 W, respectively. The use of *data-driven scheduling* can seamlessly support advanced CNN architectures without any reconfiguration.

REFERENCES

[1] nVIDIA, "Tesla m4 gpu accelerator," 2016.

[2] C. Wang et al., "Cnn-based object detection solutions for embedded heterogeneous multicore socs," in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Jan 2017, pp. 105–110.

[3] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept 2010, pp. 273–283.

[4] J. Qiu et al., "Going deeper with embedded fpga platform for convolutional neural network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35.

[5] X. Wei et al., "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in ACM/EDAC/IEEE Design Automation Conference (DAC), June 2017, pp. 1–6.

[6] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™deep learning accelerator on arria 10," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: ACM, 2017, pp. 55–64.

[7] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: scalable and efficient neural network acceleration with 3d memory," in Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 751–764.

[8] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in Proceedings of the 43rd International Symposium on Computer Architecture, 2016, pp. 367–379.

[9] Z. Du et al., "ShiDianNao: shifting vision processing closer to the sensor," in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), June 2015, pp. 92–104.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems 25, 2012, pp. 1097–1105.

[11] F. N. Iandola et al., "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," CoRR, vol. abs/1602.07360, 2016.

[12] C. Szegedy et al., "Going deeper with convolutions," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015, pp. 1–9.

[13] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, vol. 52, no. 1, Jan 2017, pp. 127–138.

[14] Y. Chen et al., "Dadiannao: A machine-learning supercomputer," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2014, pp. 609–622.

[15] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," CoRR, vol. abs/1409.1556, 2014.

[16] Accellera, "Systemc 2.3.3," November 2018.

[17] M. Delakis and C. Garcia, "Text detection with convolutional neural networks," in International Conference on Computer Vision Theory and Applications, 2008, pp. 290–294.

[18] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," CoRR, vol. abs/1603.01025, 2016.

[19] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," IEEE Journal of Solid-State Circuits, vol. 31, no. 5, May 1996, pp. 677–688.