Simple SIMD Processor

Tanvir Hoque
dept. of Electrical and Electronic Engineering
(of Ahsanullah University of Science and Technology)
Dhaka, Bangladesh
email: 180205041@aust.edu

Avishek Debnath
dept. of Electrical and Electronic Engineering
(of Ahsanullah University of Science and Technology)
Dhaka, Bangladesh
email: 180205014@aust.edu

Akfa Sultana Mithika
dept. of Electrical and Electronic Engineering
(of Ahsanullah University of Science and Technology)
Dhaka, Bangladesh
email: 180205005@aust.edu

***Abstract*— SIMD (single instruction multiple data)-type processors have been found very efficient in image processing applications, because their repetitive structure is able to exploit the huge amount of data-level parallelism in pixel-type operations, operating at a relatively low energy consumption rate. However, current SIMD architectures lack support for dynamic communication between processing elements, which is needed to efficiently map a set of non-linear algorithms. An architecture for dynamic communication support has been proposed, but this architecture needs large amounts of buffering to function properly. In this paper, we are going to implement the basic IC of simple SIMD processor with the use of verilog codes and cadence tool (genus, encounter). The design is based on ALU (arithmetic logic unit). Processor optimization with several test algorithms have shown with a performance improvement of up to 5x compared to a locally connected SIMD-processor. Also, detailed area models have been developed, estimating the proposed architectures to have an area overhead of 30-70%compared to a locally connected SIMD architecture (like the IMAP).**

Keywords—component, formatting, style, styling, insert (*key words****)***

I. Introduction (*Heading 1*)

This project is to implement a simple SIMD processor, core of which is a 16-bit SIMD ALU. 2's compliment calculations are implemented in this ALU. The ALU operation will take two clocks. The first clock cycle will be used to load values into the registers. The second will be for performing the operations. 6-bit opcodes are used to select the functions. The instruction code, including the opcode, will be 18-bit.

The ALU will be embedded into a simple processor based on 5-stage, delay of each stage will be 1 cycle, meeting the delay of ALU, as shown in the figure below. The 5 typical stages are IF, ID, EX, MEM and WB, without pipeline. In the stage IF, a 10-bit address will be sent to an instruction Block-RAM (BRAM) to fetch 18-bit instruction. In the stage IF, a 10-bit address will be sent to an instruction Block-RAM (BRAM) to fetch 18-bit instruction. In the stage ID, the instruction will be decoded and some of control registers will be set to control the following stage. In the stage EX, ALU will process data in registers or implement some control commands, e.g. jump. In the stage MEM, if the instruction is "store" or "load", data would be read from/ written to data BRAM, based on instruction and address. Finally, in the stage WB, data will be written back to register. The pins of clock, reset, address, data and BRAM enable will be exposed on the interface of processor. The architecture of processor is shown in **Figure 1.**

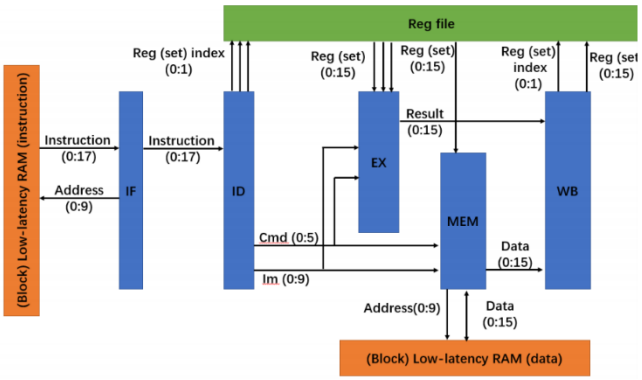
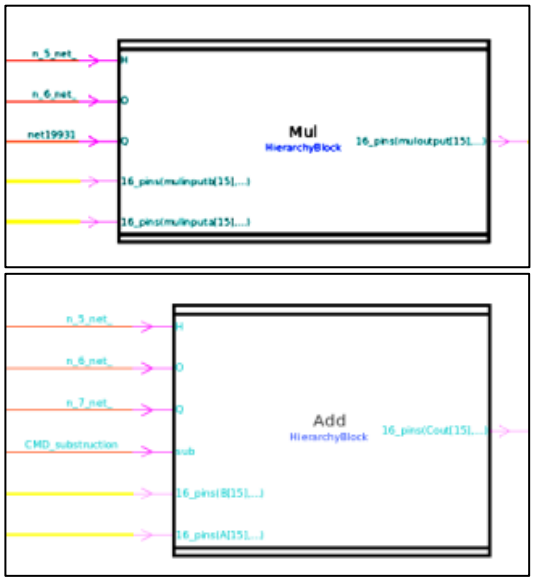


Figure 1: *Simple SIMD Processor Architecture*

II. Design of **SIMD** ALU

The ALU consists of three SIMD basic computation units: SIMD adder, SIMD multiplier and SIMD shifter. These three components can be reused for the computation of data with different width (4-bit, 8-bit or 16-bit) and can handle all the instructions (listed in Section 4) in the design specification. Each of them is controlled by three input port, H (for 16-bit), O (for 8-bit) and Q (for 4-bit), indicating the kind of input data, so that the unit can handle the data properly, as shown below.



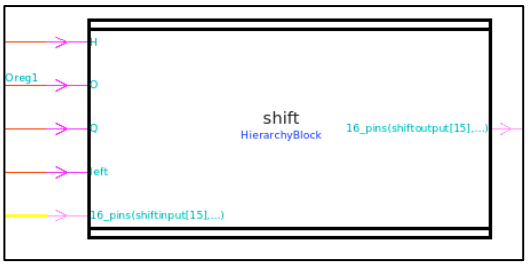


***Figure 2**: The Interface of ALU Components*

A. *SIMD Adder*

The SIMD adder is implemented based on 4 4-bit adders. To reuse the adder for data with different width, the input signals, H, O and Q, control the forwarding of the carries between the adders. Moreover, the adder can also support subtraction, by flipping the second operand and add one to it when the signal sub is set. The data path of the SIMD adder is shown in **Figure 3.**

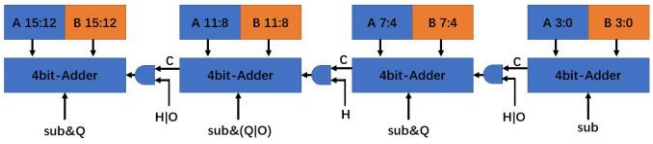


***Figure 3**: Datapath of SIMD Adder*

B. *SIMD Shifter*

The SIMD shifter can also support data with different width. It is based on two 16-bit shifters and necessary overstep-correct logic. To illustrate the implementation of the shifter, the example based on logic-left-shift is shown in **Figure 4**. The shifter will determine whether the MSB of 4-bit block should be set to 0 or just inherit the value from the LSB from 4-bit block in front of it, according to the input signal, H and O.

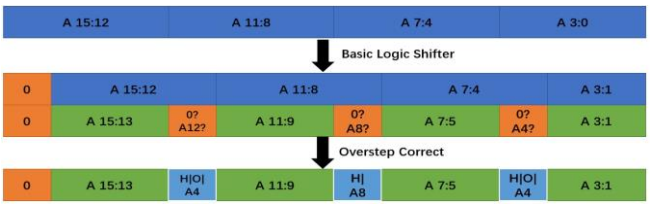


***Figure 4**: Example of SIMD Shifter*

C. *SIMD Multiplier*

The SIMD multiplier is implemented with adders and shifters. To make it support multiplication with different data width, the input signal, H, O and Q, are used to control the input operands of the adders and select corresponding outputs for the target data width. The 1x16x16 multiplication is implemented as a typical multiplier, as **Figure 5**, where adder tree is utilized. The least significant 16 bits of the sum is the output of multiplication.

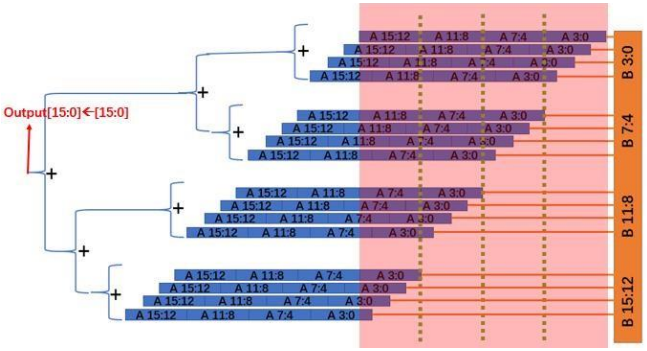


***Figure 5**: 1x16x16 multiplication*

To reuse the structure of 1x16x16 multiplication implemented as above, for the input signal, H, O and Q, are used to select the inputs of the adders, to control the adders to only sum up the range of bits. The 4x4x4 multiplication is implemented as shown **Figure 6**, where for each adder, just a part of input bit is valid while other bits will be set to 0. Similarly, the 2x8x8 multiplication is shown in **Figure 7**.

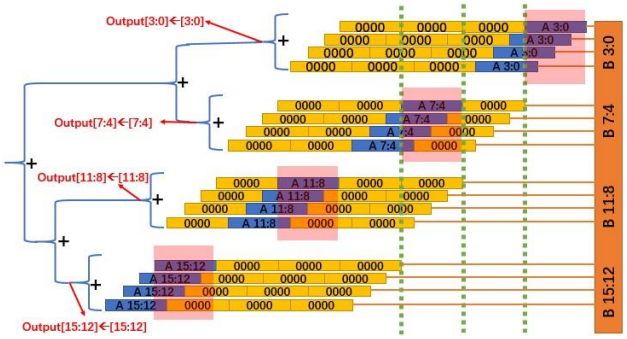


***Figure 6**: 4x4x4 multiplication*

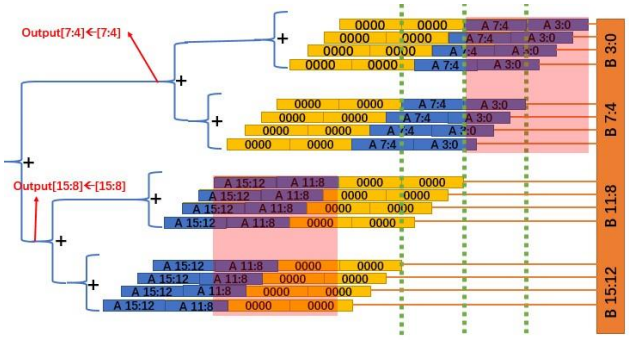


***Figure 7**: 2x8x8 multiplication*

III. REGISTER FILES

There are registers which are used to store the data for computation.

16-bit registers: H1, H2, H3, H4

8-bit registers (3 sets): {O1, O2}, {O3, O4}, {O5, O6}

4-bit registers (3 sets): {Q1, Q2, Q3, Q4}, {Q5, Q6, Q7, Q8}, {Q9, Q10, Q11, Q12}

Loop counter (for loop control): LC.

IV. Conclusion and Future work

An SIMD processor can be fabricated through many ways. We tried to synthesis this IC by the use of cadence tools and verilog codes. Although the design was slower than a hard-wired design, the development time for an application using this IC will be significantly lower since designers can adopt a purely programming based model and need not be concerned with lower levels details such as data path design, control design, hardware patterns etc. An SIMD processor can be applied to the RC4 key search problem and able to achieve a high level of parallelism as well as utilize the higher memory bandwidth available on the device. The SIMD approach also has benefits in that the design can be amortized over many different applications potentially resulting in a large overall savings in development effort. Finally, using this approach, there is potential to customize the instruction set of the processor as well as to add coprocessing elements to further accelerate application.

V. References

[1] Y. Fujita, S. Kyo, N. Yamashita, and S. Okazaki. A 10 GIPS SIMD Processor for PC-based Real-Time Vision Applications Architecture, Algorithm Implementation and Lan- guage Support. In *In Proceedings of the 4th International Workshop of the Computer Architecture for Machine Per- ception, (CAMP)*, pages 22–32, Washington, DC, USA, October 1997. IEEE Computer Society.

[2] H. Fatemi, H. Corporaal, T. Basten, R. Kleihorst, and P. Jonker. Designing Area and Performance Constrained SIMD/VLIW Image Processing Architectures. In *Proceed- ings of Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pages 689–696, Antwerp, Belgium, September2005. Springer-Verlag, Berlin, Germany, 2005. K. Elissa, “Title of paper if known,” unpublished.

[3] https://github.com/zslwyuan/Basic-SIMD-Processor-Verilog-Tutorial

VI. Result (Data Representation)

A. Constraints Tables

Clock Period	18 ns
Clock Uncertainty setup	0.5
Clock Uncertainty hold	0.5
Max Transition	4
Clock Transition minimum fall	0.5
Clock Transition minimum rise	0.5
Clock Transition maximum fall	0.5
Clock Transition maximum rise	0.5
DRIVING CELL	BUFX8
DRIVE PIN	Y
Max Fan Out	20
Load	0.5
Operating Conditions	SLOW
Input delay max	0.5
Output delay max	0.5

B. Result Table

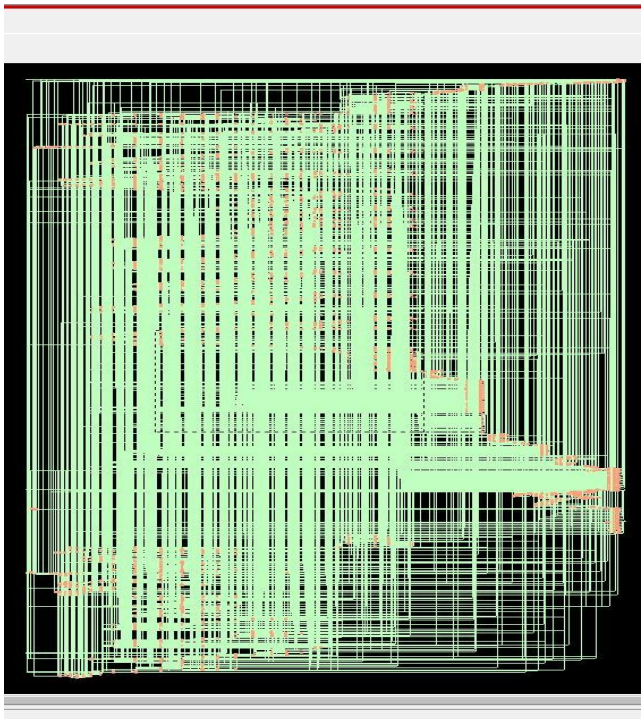
Final density	106.21 %
Total placed cells	4354
Remaining Violations (DRC)	5
Total Gates	12462
Area	12786.4 um^{-2}

VII. Result (Pictorial representation)

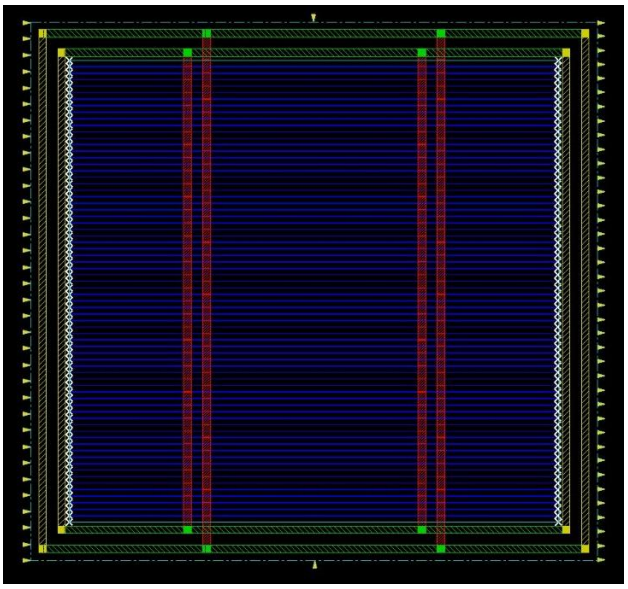
A. Synthesis Result

```
Info    : Done incrementally optimizing. [SYNTH-8]
        : Done incrementally optimizing 'SIMD'.
     flow.cputime  flow.realtime  timing.setup.tns  timing.setup.wns  snapshot
UM:            52             41              0 ps           6439.3 ps  synthesize
Finished SDC export (command execution time mm:ss (real) = 00:00).
genus@design:SIMD>
```

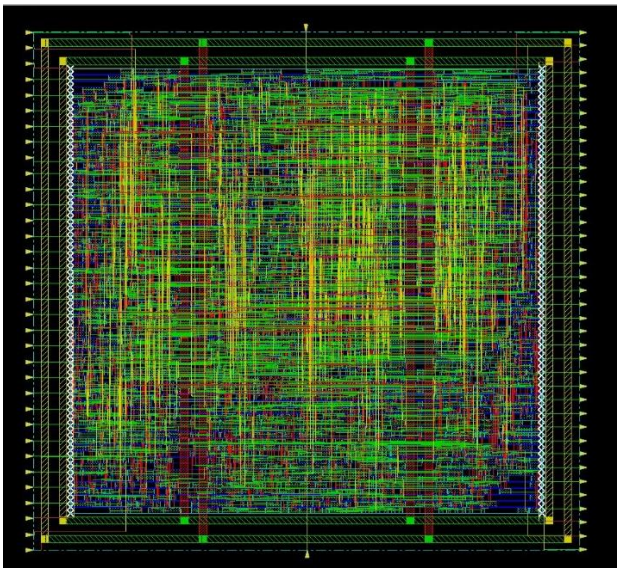
B. Synthesized Circuit



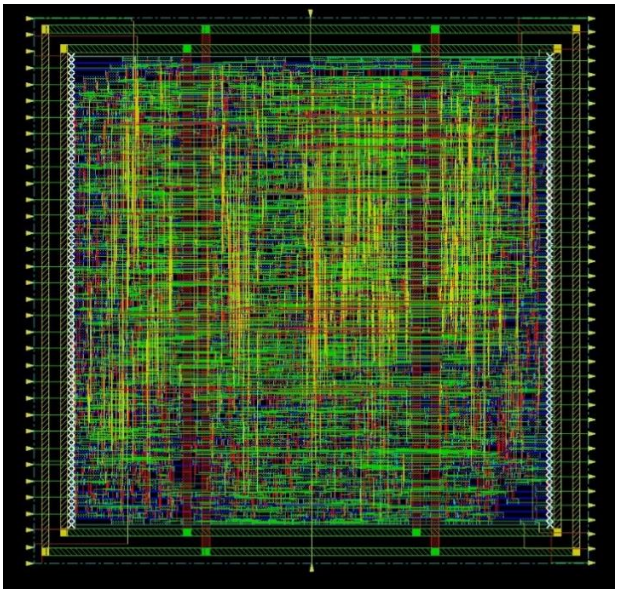
C. PNR Output (Placement)



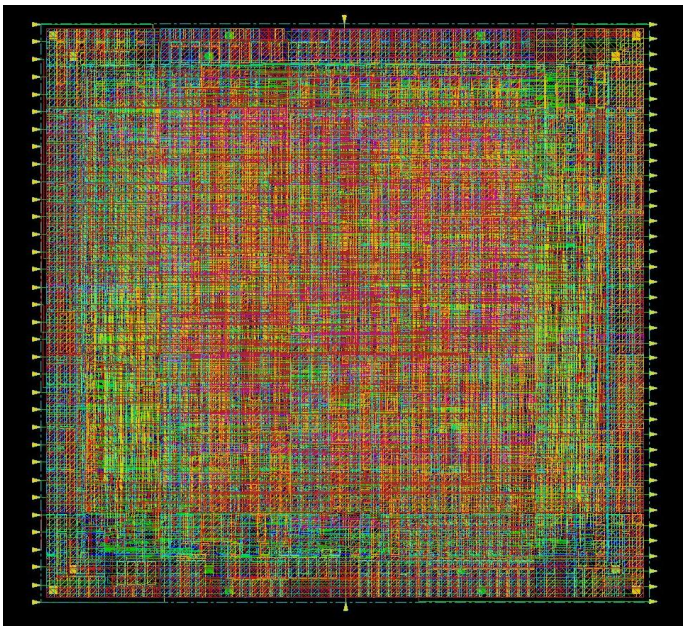
D. PNR Output (After Route)



E. Plcaement Optimization



F. Final Design (After Adding Filllers)

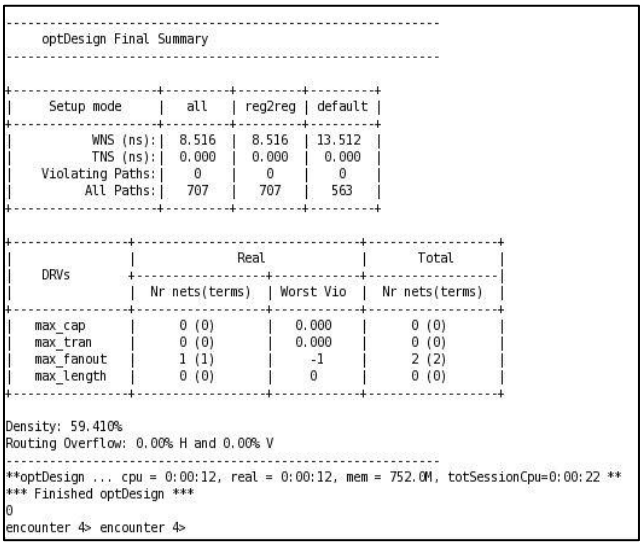


G. STA Reports (Before Violation Cleaning)

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  8.830  |  8.830  | 13.695  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   707   |   707   |   563   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      3 (31)      |   -0.382   |      3 (31)      |
|   max_fanout   |      0 (0)       |     0      |      1 (1)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 58.928%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------
Reported timing to dir ./timingReports
Total CPU time: 1.52 sec
Total Real time: 2.0 sec
Total Memory Usage: 672.058594 Mbytes
```

The above image shows the summary of existing setup and DRC violations in the pre-CTS stage. In the summary report, there is a negative value in **max_tran** which is indicates that there is a violation.

H. STA Reports (After Violation Cleaning)



```
------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------

+--------------------+--------+--------+---------+
|     Setup mode     |   all  | reg2reg | default |
+--------------------+--------+--------+---------+
|           WNS (ns):|  8.516 |  8.516 | 13.512  |
|           TNS (ns):|  0.000 |  0.000 |  0.000  |
|    Violating Paths:|    0   |    0   |    0    |
|          All Paths:|   707  |   707  |   563   |
+--------------------+--------+--------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +----------------+--------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+----------------+--------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      1 (1)       |     -1     |      2 (2)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+----------------+--------------+------------------+

Density: 59.410%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------
**optDesign ... cpu = 0:00:12, real = 0:00:12, mem = 752.0M, totSessionCpu=0:00:22 **
*** Finished optDesign ***
0
encounter 4> encounter 4>
```

We solve this violation by **optDesign –preCTS** command in encounter terminal for optimization. Optimized summary report is in above figure.