Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 11(2): 112-117, Jun. 2013 **Regular paper**

Design Approach with Higher Levels of Abstraction: Implementing Heterogeneous Multiplication Server Farms

Sangook Moon*, ***Member, KIICE***

Department of Electronic Engineering, Mokwon University, Daejeon 302-729, Korea

Abstract

In order to reuse a register transfer level (RTL)-based IP block, it takes another architectural exploration in which the RTL will be put, and it also takes virtual platforms to develop the driver and applications software. Due to the increasing demands of new technology, the hardware and software complexity of organizing embedded systems is growing rapidly. Accordingly, the traditional design methodology cannot stand up forever to designing complex devices. In this paper, I introduce an electronic system level (ESL)-based approach to designing complex hardware with a derivative of SystemVerilog. I adopted the concept of reuse with higher levels of abstraction of the ESL language than traditional HDLs to design multiplication server farms. Using the concept of ESL, I successfully implemented server farms as well as a test bench in one simulation environment. It would have cost a number of Verilog/C simulations if I had followed the traditional way, which would have required much more time and effort.

Index Terms: Embedded systems, Electronic system level, Multiplication server farm, SystemVerilog

I. INTRODUCTION

A. Motivation

Electronic system level (ESL) design includes hardware and software interactions with higher levels of abstraction for system-level transactions. ESL methodologies have evolved from algorithmic modeling, such as architectural explorations, and proving concepts as supplementary techniques, such as design of embedded systems, system-level test bench development, hardware/software co-simulation, and high-level synthesis of ASIC/FPGA designs. Efficient ESL design includes the ability to proceed from the concept to the optimal implementation of architectural functionality, as well as verification. As one of the most evolved ESL languages, Bluespec SystemVerilog (BSV) has introduced to the system hardware architects a new way to simplify implementing complicated control logic while at the same time not losing control over the architecture and efficiency of the design. According to [1], a reduction of over 50% in time can be achieved in verifying a design and fewer than 50% of the bugs can be found compared to traditional register transfer level (RTL) design. BSV differs from traditional Verilog or VHDL, or even from SystemVerilog in many aspects. Instead of using the traditional procedural statements, which implement concurrency such as *always*, BSV uses statements named *rules* that refer to fully synthesizable behavior. BSV enables better overall system generation than via traditional ways since all instances such as methods, rules, modules, interfaces, and functions are regarded as first-class objects. This means that the objects can be used as arguments in other objects.

Received 06 November 2012, **Revised** 18 March 2013, **Accepted** 01 April 2013
***Corresponding Author** Sangook Moon (E-mail: smoon@mokwon.ac.kr, Tel: 042-829-7637)
Department of Electronic Engineering, Mokwon University, 88 Doanbuk-ro, Seo-gu, Daejeon 302-729, Korea.

Open Access **http://dx.doi.org/10.6109/jicce.2013.11.2.112** print ISSN: 2234-8255 online ISSN: 2234-8883

B. Related Work

This work is an extended version of a conference proceeding version [2]. In this version I explain the multiplier architecture, points on extracting a higher level of abstraction, and the reorder buffer more in detail. To build a server farm in the traditional way, a hardware finite state machine must be defined in HDL, which costs time and effort [3]. In the high-level abstraction languages such as BSV however, it is much simpler to define the state machine logic. At this point, I experimentally implemented server farms using BSV with a few existing Verilog IP logics such as multipliers and random number generators, to build the multiplication server farms. I first discuss the custom designed modified Booth multiplier, and then I discuss how to utilize the IPs in the BSV test bench program. Next, after considering how I achieved the higher levels of abstraction, I present the random number generator and the architecture of the server farms, and finally discuss the results.

II. MODIFIED BOOTH MULTIPLIER WITH FULL-CUSTOM LAYOUT

A. Modified Booth Multiplier

Fig. 1 presents the custom layout of the modified Booth multiplier. In order to accommodate a fast clock frequency, I used 2-stage pipeline architecture. In the Wallace tree module I used a 4:2 carry save adder (CSA) for a regular layout. The multiplier can perform 2's complement-represented multiplication in 9.5 ns with LG 0.6 μm 3-metal N-well CMOS technology. The multiplier is composed of 9115 transistors and occupies 1135 × 1545 μm^2 in the die size [4]. The IP was also written in Verilog HDL for reuse in the BSV ESL design.

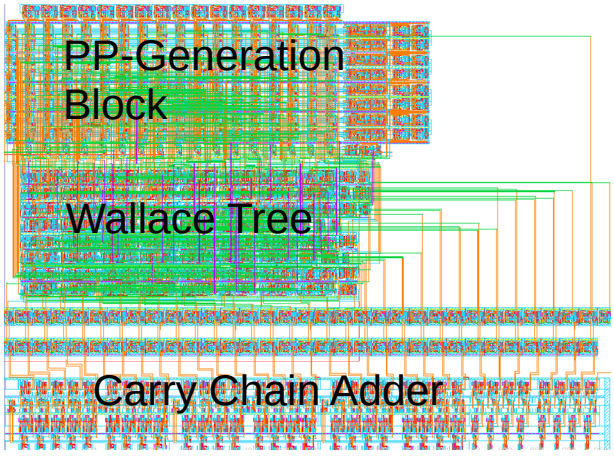

Fig. 1. Custom layout of modified Booth multiplier.

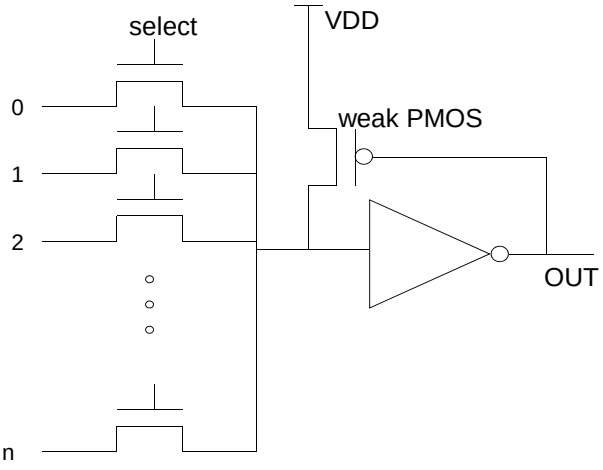

Fig. 2. Size-reduced structure of n-to-1 multiplexer. VDD: virtual device driver, PMOS: p-channel metal oxide semiconductor.

B. PP-Generation Block

The partial product (PP) generation block includes the Booth encoder. According to the modified Booth's algorithm, a variation of multiplicand (A) among [+2A, +1A, 0A, -1A, -2A] must be chosen in reference to the last three digits of the multiplier input (B). +{1,2}/-{1,2} operation is implemented by shift or inversion. To reduce the area, I customized the size of the multiplexors using a weak pull-up pMOS transistor. In Fig. 2, instead of using a complementary MOS transistor system, I used nMOS switches only, in order to reduce the size needed. To prevent the side-effect of using only nMOS, I appended a weak pMOS pull-up transistor between the input and output of the buffer so that the multiplexer output maintains both strong high and low.

C. Wallace Tree

The 9 pieces of partial products generated are handled by 4:2 CSAs. The reason I chose the 4:2 CSA structure was that 4:2 structure shows good regularity compared to the 3:2 structured types. I also used the *sign-generation method* to avoid unnecessary calculations incurred by the sign bits in the Wallace tree.

D. Carry Chain Adder

The carry vector and the sum vector are first stored in buffers, which were designed with a master/slave structure. In the next clock cycle, the register outputs are transferred into the 33-bit adder blocks. I used carry chain adders as the basic adder cell. The final addition results are transmitted through output buffers to the pads.

E. IP Wrapping to SystemVerilog

Verilog transformed hardware uses wires and registers to define the input and output signals of a module. The BSV utilizes an object-oriented programming interface-based design to connect between modules. I adopted the BSV concept of a method to define input arguments of a method and return the value of a method to build an appropriate wrapper for the modified Booth multiplier IP. I used the "import BVI" mechanism to create a wrapper around the RTL module so that the IP looks like a BSV module. Instead of ports, the wrapped module uses methods and interfaces.

III. HIGHER LEVELS OF ABSTRACTION

A. Strong Type System

Languages with a higher level of abstraction such as Haskell suggest a strong type system [5]. Every single type in every expression is determined before compile time, which leads to safe code. In such languages, a program in which a Boolean number is divided by an integer fails to be compiled. This characteristic is recommended because it is better to catch this kind of error during compile time rather than having a dead lock during execution time. BSV complies with the Haskell type system and every keyword belongs to a specific type. Therefore, during compile time, the compiler is able to recognize such errors, by which the system designer can save time.

```
Example 1.
if a<b then c := True
Else c := False

is translated as

if a<b -> c := True
[] a ≥ b  ⟶ c : False
fi

Example 2.
if error = True then x := 0

is translated as

if error = True ⟶ x := 0
[] error = False  ⟶ skip
fi
```

Fig. 3. Some examples of guarded command language.

```
module mul17_b (result, multiplicand, multiplier,
                                        clock, reset);
output [32:0] result;
input [16:0] multiplicand, multiplier;
input clock, reset;

//structural model of the custom multiplier

endmodule
```

Fig. 4. Verilog-description of the modified Booth multiplier.

B. Guarded Command Language

BSV also adopts the characteristics of guarded commanded language invented by Dijkstra and introduced in [6]. Since this language is rather a theory than a physical language, there is no specific compiler. The theory makes the programming concept succinctly integrated, so that we can easily determine the correctness of the program using Hoare logic [7].

Guarded command means that commands used in programs are guarded, as the name implies. The guard is a proposition, and it must be true before executing the commands. If the guard is false, the command is not executed. Guarded command easily helps prove whether a program satisfies a certain specification or not, as shown in Fig. 3.

```
interface Mult#(type t);
        method Action start(t a, t b);
        method t result();
endinterface

import "BVI" mul17_booth =
module mkMult (Mult#(t))
        provisos (Bits#(t, st));

default_clock clk(clock);
default_reset rst(reset);

method start (A, B) enable((*inhigh*) unused1);
method X result;

schedule start C start;
schedule start SBR result;
schedule result CF result;

endmodule
```

Fig. 5. Implemented wrapper of interfaces, modules, and methods with Bluespec.

C. IP Wrapping to BSV

To extract the highest level of abstraction existing, I transformed the custom designed multiplier into Verilog HDL, which is shown in Fig. 4. Then I created the wrapper module to be used in the Bluespec simulator. With respect to reusing a well-defined IP in higher-level system design, this is the most favorable benefit of implementation in a high level of abstraction. The inputs and outputs should be defined as methods and interfaces using import "BVI" as shown in Fig. 5.

IV. IMPLEMENTATION OF SERVER FARMS

A. Multiplication Server Farms

A server farm or server cluster is a collection of computer servers with an arbiter to accomplish server needs far beyond the capability of one machine. The arbiter allocates incoming jobs to any server that becomes available and sends the result back to its caller. I implemented two multiplication server farms, one for simple paper-and-pencil pipelined multipliers and the other for the modified Booth multiplier, which I described in Section II-A (Fig. 6). In the test bench, the arbitration controller sends the same random jobs to each farm, and checks to see if the corresponding results returned from each multiplication farm are correct. For both servers, the time required to complete each job depends on the value of the multiplicand and the multiplier, so it remains uncertain whether results will be available in the order the jobs were started. However, the arbiter should return the results to its caller in the order the jobs were received. Therefore, a reorder buffer that fits this specification is required.

B. Reorder Buffers

Reorder buffers should be considered for out-of-order job executions. In BSV, the reorder buffer is implemented in the library package as a reusable parameterized IP. The concept with multiple update ports is shown in Fig. 7 [8]. The IP named completion buffer provides the function of reorder buffers. The interface of the IP offers three methods. The reserve method allows the caller to reserve a slot in the buffer, returning a token holding the identity of the slot.

When a job finishes, the complete method allows the result to be stored in the reorder buffer. The drain method returns results in the order where the tokens were assigned from the first. In this way, the results of swiftly completed jobs can wait in the buffer until a time-consuming job ahead of them finishes.

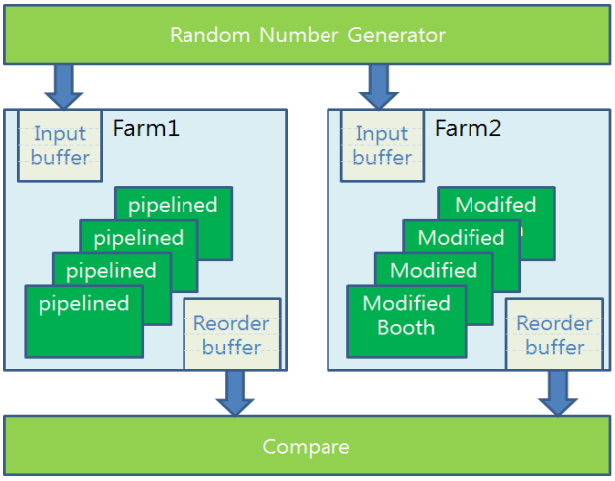

Fig. 6. Top level block diagram of the multiplication server farm and the test bench.

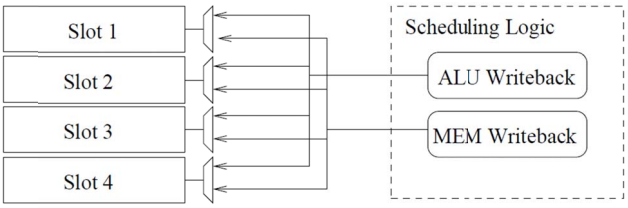

Fig. 7. Buffer reordering concept with multiple update ports. ALU: arithmetic-logic unit, MEM: memory.

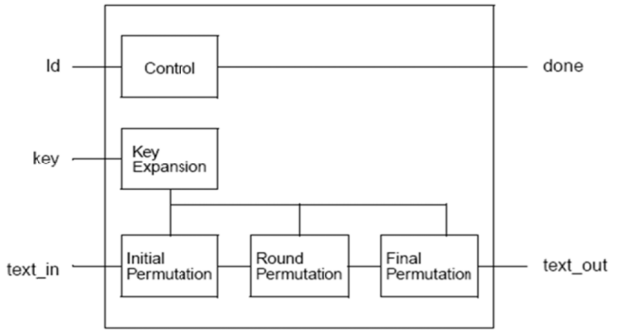

Fig. 8. Advanced Encryption Standard ciphering core block diagram.

C. Random Number Generator

The public Advanced Encryption Standard (AES) cryptographic algorithm to achieve a National Institute of Standards and Technology-recommended secure random number generator [9] was reused and described with Verilog HDL. Thanks to Drimer et al. [10], I was able to utilize the fastest AES Verilog HDL module to make the random number generator. As shown in Fig. 8, complex and repetitive permutations enable the ciphering algorithm applicable to random number generation.

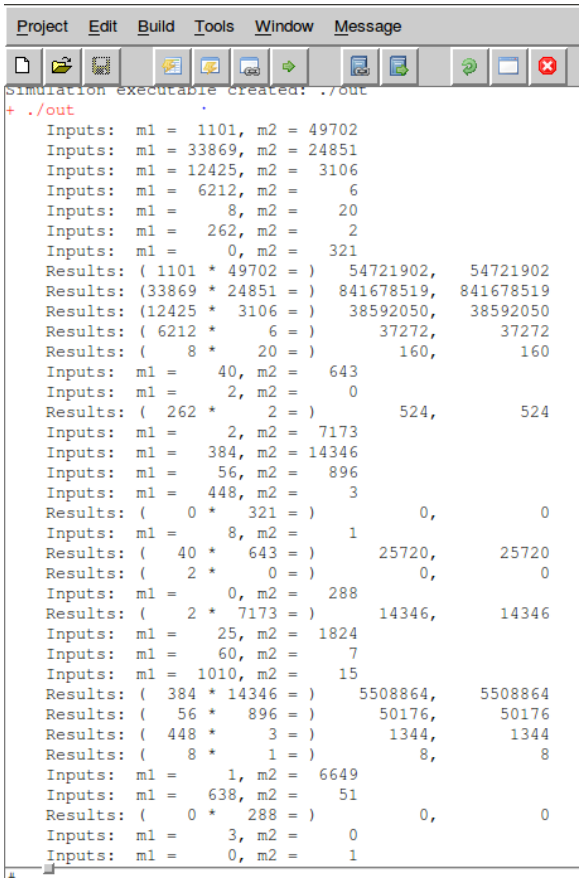

Fig. 9. Multiplication server farm test results.

According to the cipher theory, encrypted data with enough high security density behave as random numbers because after the repetitive permutation, data retain the characteristics of whitening. I then devised a wrapper for the AES Verilog IP to fit in the BSV test bench program. The test bench program instantiates 2 multiplication server farms and dispatches the same jobs to each farm. I utilized the technique to use the fully filled first-in-first-outs to evenly distribute the data elements one at a time. The test result is shown in Fig. 9. As you can see, the multiplication server farm accomplishes consecutive jobs in the order the data were received.

V. CONCLUSIONS

ESL methodologies have evolved from algorithmic modeling such as architectural explorations and proving concepts as supplementary techniques such as design of embedded systems, system level test bench development, hardware/software co-simulation, and high-level synthesis of ASIC/FPGA designs. To rapidly prototype the multiplication server farms, simple pipelined multipliers and modified Booth multipliers are implemented together with input buffers and reorder buffers, each for out-of-order completion. An AES cryptographic function module is reused to produce random number test vectors. By means of the BSV, I was able to utilize a higher system level of abstraction than the traditional HDLs to attain multiplication server farms with two farms. If I parameterize the number of farms and the number of multipliers, I could obtain additional server farms with ease, while it would have taken several times more time and effort to build the same implementation using Verilog or VHDL.

REFERENCES

[1] Bluespec System Verilog Reference Guide, Revision 2012 [Internet], Available: http://www.bluespec.com.

[2] S. Moon, “System Verilog-based approach of a design of multiplication server farms,” in *Proceedings of the 4th International Conference on Ubiquitous and Future Networks*, Phuket, Thailand, pp. 478-479, 2012.

[3] S. Moon, “Design of an FPGA-based IP using SPARTAN-3E embedded system,” *Journal of Maritime Information and Communication Sciences*, vol. 9, no. 4, pp. 428-430, 2011.

[4] S. Moon, B. Moon, and Y. Lee, “Design of a full-custom 17b*17b multiplier and its efficient test methodology,” *Journal of Korea Information and Communication Society*, vol. 26, no. 3B, pp. 362-368, 2001.

[5] M. Lipovaca, *Learn You A Haskell for Great Good: A Beginner’s Guide*. San Francisco, CA: No Starch Press, 2011.

[6] Wikipedia, Guarded command language [Internet], Available: http://en.wikipedia.org/wiki/Guarded_Command_Language.

[7] Wikipedia, Hoare logic [Internet], Available: http://en. wikipedia. org /wiki/Hoare_logic.

[8] N. Dave, “Designing a reorder buffer in Bluespec,” in *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design,* San Diego: CA, pp. 93-102, 2004.

[9] S. S. Keller, NIST-recommended random number generator based on ANSI X9.31 appendix A.2.4 using the 3-key triple DES and AES algorithms [Internet], Available: http://csrc.nist.gov/groups/STM/cavp/documents/rng/931rngext.pdf.

[10] S. Drimer, T. Guneysu, and C. Parr, ″DSPs, BRAMs, and a pinch of logic: extended recipes for AES on FPGAs,″ *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 1, article no. 3, 2010.

Sangook Moon
was born in Korea, in 1971. He received B.S., M.S. and Ph.D. degrees in electronic engineering from Yonsei University, Korea in 1995, 1997, and 2002 respectively. After graduation, he had been working at Hynix Semiconductor as a senior engineer. In 2004, he joined the Department of Electronic Engineering at Mokwon University, where he is currently an associate professor. His current research interests include VLSI architecture, crypto-processors, computer arithmetic, SoC, and embedded systems.