From Verilog to Bluespec: Tales of an AES Implementation for FPGAs

Omar Choudary, University of Cambridge

Abstract

In this paper I present a combined Verilog and Bluespec implementation of the Advanced Encryption Standard (AES) for FPGAs. This solution relies on the fastest AES implementation for FPGAs, presented by Drimer et al. Bluespec is a versatile HDL, allowing an easy parametrization of a large design. Based on this property I show the simple use of two different AES implementations (AES32 and AES128) in a large Bluespec design.

Using the Altera DE2 board and Quartus software I synthesize the design and test it on a Cyclone II FPGA. I present the performance results obtained from several tests and I compare them against the results presented by Drimer et al. The source code for the complete design is made publicly available.

Index Terms

Algorithms, Performance, Security

I. Introduction

As part of my MPhil course *Advanced Computer Design*, I was required to choose or define a challenging hardware project and implement it on the Altera DE2 development board[1]. It was also recommended to use Bluespec[2] as the hardware definition language.

The Altera DE2 development board provides a Cyclone II FPGA[3], an LCD, LEDs, 7-segment displays, switches, buttons, 8MB SDRAM, 512K SRAM, 4MB Flash and many I/O devices including USB, Ethernet and Video In/Out. All these components can be glued together by using the Quartus[4] software. The DE2 kit comes with an embedded processor, the NIOS II[5], that is implemented within the Cyclone II FPGA and can be used to easily access most of the DE2 board components. Also, there is an integrated development environment (IDE) dedicated to NIOS II. Using this IDE the user can create traditional C++ programs to transfer data between the different components of the DE2 board, including the Cyclone II FPGA.

The Bluespec SystemVerilog (BSV) is a new type of Hardware Definition Language (HDL), different than traditional Verilog or VHDL. Instead of the usual synchronous always blocks, BSV uses **Rules** that express synthesizable behavior. *Rules* are meant to achieve correct concurrency and eliminate race conditions. BSV also enables a better overall system generation because all instances, including methods, rules, modules, interfaces and functions are considered first-class objects which means they can be used as arguments to other objects. Another advantage of BSV is that it allows formal verification.

The Advanced Encryption Standard (AES)[6] is a widely used block cipher having many implementations in both software and hardware. Drimer et al.[7] have made the fastest hardware implementation of AES for Xilinx Virtex-5 FPGAs[8]. I will refer to this implementation as **AES2** from now on. AES2 has been used as the basis of the work presented in this paper.

AES2 comes in three versions: a) AES32 which computes two 128-bit data blocks in 86 clock cycles using a 32-bit datapath (please see the original paper for more details); b) AES128 which uses 4 instances of AES32 to compute eight 128-bit data blocks within the 86 clocks; c) AES128U which represent a completely unrolled version of the design, using ten instances of AES128. All code is written in Verilog and is designed to run on Xilinx Virtex-5 FPGAs.

The main author of AES2, Saar Drimer, has proposed a project of exploration to make AES2 more flexible by taking advantage of the polymorphism capabilities provided by Bluespec. As I am interested in security and hardware, this project was a very tempting challenge.

To successfully transform AES2 I needed to: a) design the solution in Bluespec such that a larger Bluespec project could use AES2; b) make the design run on Altera's Cyclone II instead of Xilinx Virtex-5 FPGAs. In the remaining of this paper I refer to the resulting Bluespec AES2 implementation as **AES2B**.

In the following sections I present the steps and challenges involved in converting AES2 into a Bluespec design running on Cyclone II FPGAs. I also present the results from several real tests done using the Altera DE2 board on AES2B and I compare the performance with the results of AES2.

II. Implementation

The first challenge of this project was to decide what could be done in the relatively short time frame (about 80 hours of work).

I decided to use the NIOS II processor to control the data flow between AES2B and the other devices on the DE2 board. A *C* program controls all the execution flow, transferring data through the JTAG interface of the DE2 board. The Quartus software provides a tool named *SOPC Builder* to define the components to be used in a design, including RAM and ROM memories,

different versions of the NIOS II processor and user created modules. The Avalon framework (part of the DE2/Quartus suite) provides a standard interface to interconnect all the modules of a design. For components created using the SOPC Builder the connections are created automatically. However, for user defined modules (given via Verilog or VHDL files), the interface has to be defined manually and it needs to match the format of the Avalon framework. This can be a difficult task if the user does not have experience with the framework. Taking advantage of Bluespec's interface based design, my module lecturer Dr. Simon Moore has created a BSV interface to connect a module with the Avalon framework. BSV files can compile into Verilog code. Thus, any Bluespec design using Moore's interface will compile into a module having the correct setup to be included into a large design using the SOPC Builder.

Having a test platform in place, I needed to use AES2. I have considered three options: a) leave the AES2 code as it is, and just test it on the DE2 board; b) rewrite the entire AES2 implementation in Bluespec; c) import the different AES2 implementations in Bluespec and export a general interface. I quickly realized that (a) is not a scalable solution, is prone to errors and does not satisfy the project goals. Option (b) was not feasible in the short time frame and was not certain to work as AES2 relies on strict timing. Thus I continued with solution (c).

Using a Verilog module in Bluespec seems difficult the first time. This is because Verilog uses wires and registers to define the input and output signals of a module. On the other hand Bluespec uses the concept of a method to define input (arguments to a method) and output (return value of a method). This can be seen in Appendix, where I import the AES2 modules *aes32_dsp_8p* (AES32) and *aes128_dsp* (AES128). I first declare a general interface *AESIfc* (of parametrizable data width) containing two methods: *request* and *read_response*. Any module using this interface will have to define these methods. The two modules created, *mkAES32* and *mkAES128*, use this general interface and declare (but not define) the two methods. In this particular case mkAES32 and mkAES128 only declare the methods because their input arguments and return values are linked to the Verilog modules aes32_dsp_8p and aes128_dsp respectively. Thus, any Bluespec module using mkAES32 or mkAES128 will compile into a Verilog module using aes32_dsp_8p or aes_dsp respectively. The method *request (DIN, KEY) enable (START)* specifies that every time this method is called within a rule, the DIN and KEY wires will have the values given as parameter to the method and the signal START will be asserted high. This also implies that START will remain low every clock cycle the method is not called. This would happen if for example at a given time there is no rule containing a call to method *request* that will *fire* (a Bluespec rule is said to fire when it is executed).

The next step was to use the imported Bluespec modules into a Bluespec design, connect the design to the Avalon framework and test it on the DE2 board. The complete design, described in the following sections, can be obtain from my web page[9].

A. AvalonAES32

I started by using the imported AES32 module (mkAES32) as it represents the basic component of AES2. Bluespec provides a simulator, named **bluesim**, that can be used to easily simulate a Bluespec design. Thus I created a small BSV testbench just to test the correct functionality of mkAES32. Bluespec was about to show its limitations. It seems that bluesim does not currently support simulation of designs using imported Verilog modules. This limitation has been an important impediment in the design of this project as I was not able to perform any simulation.

Without the possibility of simulation I continued by creating the Bluespec module *mkAvalonAES32* that links mkAES32 to the Avalon framework. Bluespec's main advantage over Verilog is that I was able to create a general interface (AESIfc) for any instantiation of AES2 (AES32 or AES128). In order to use the 32-bit version I just need to provide the data width as parameter to the interface and call the correct module, as below:

```
AESIfc#(32) aes32 <- mkAES32;
```

If instead we wanted to use the 128-bit version then we just needed to change the previous line to this one:

```
AESIfc#(128) aes128 <- mkAES128;
```

Notice however that the names and parameters of the two methods (request and read_response) declared inside the AESIfc interface (see Appendix) remain the same. And because the interface is polymorphic (that is, its argument is a given parameter) all the wires of the resulted design have the correct width.

The AES32 Verilog implementation provided within AES2 requires input data (initial plain text, key or subkeys) each clock cycle. Thus I needed to provide this data to the mkAES32 module using the *request* method. Using the Avalon framework I provided the required data from the Nios II processor to the mkAvalonAES32 module. Then the mkAvalonAES32 module performs an encryption operation (when instructed to do so by the Nios II processor) by sending the necessary data at the correct time to the mkAES32 module. A diagram of this workflow is presented in Figure 1.

Successful compilation of the mkAvalonAES32 module was not an easy task. As mentioned earlier, Bluespec uses the concept of *Rules* instead of the traditional Verilog synchronous *always* blocks. Each such rule defines a set of actions to be taken if the rule is enabled. The rule is enabled (can fire) if all its conditions, either explicit (mentioned in the rule definition) or implicit (e.g. methods that can block, like our method read_response if the signal DONE is not high) are satisfied. The set of actions represent mainly traditional Verilog continuous assignments (<=) or method calls which translate into assignment of values to wires.

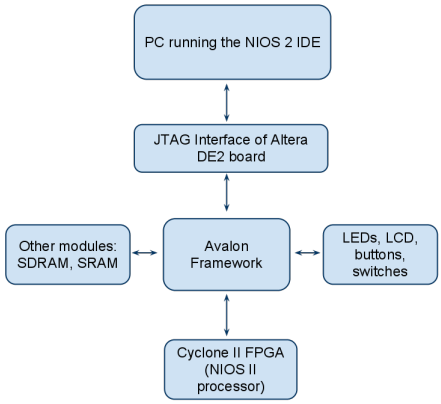

Fig. 1. *Diagram of data flow using the Avalon framework on the Altera DE2 board*

After successful compilation of the mkAvalonAES32 module, I used the resulted Verilog module into the SOPC Builder to create the full design. While building the new design I found new problems: the AES2 was designed to work on Xilinx Virtex-5 FPGAs and contained specific elements such as BRAMs DSPs. For DSPs the solution was simple: I replaced the DSP module instantiation with the simple XOR logic (code provided in AES2).

The main problem was the use of BRAMs. The Cyclone II FPGA has 105 internal memory blocks, called **M4K**. These memory blocks can be accessed dual-port just like the BRAMs used in AES2. However there are two major differences: a) The BRAMs initialization is done in the Verilog code using the *INIT_XX* parameter while the M4Ks are initialized using specific Memory Initialization File (MIF) files (standard Intel Hex files can also be used); b) the size of the BRAMs used in AES2 is 36 Kbits while the size of each M4K is 4 Kbits. Together these two problems made difficult the transformation of the BRAM data used in AES2 to my Bluespec implementation. In AES2B I have used Cyclone II's M4Ks explicitly by creating the Verilog instantiation module using Quartus' *MegaFunction* tool. As I imported the AES2 main modules (AES32 and AES128) into Bluespec I did not have access to the BRAM instantiation. Thus I had to edit some of the Verilog code in order to replace the use of BRAMs by M4Ks. For clarity I mention that there are at least two other approaches to using BRAMs efficiently in a Bluespec design. First, one can declare a register array in a Verilog module following a standard pattern and then import this Verilog module into Bluespec. This should compile into a code using the correct BRAM (if available). Secondly, it is possible to use the BRAM module provided by Bluespec. This should again, compile in a code using the correct BRAM. Unfortunately none of these methods were possible in my case. As mentioned above, I did not have access to the BRAM instantiation from Bluespec; that code was behind the imported Verilog files.

At this point the full 32-bit design was compiled and synthesized. In the following sections I present the implementation of the 128-bit design and then I continue with a description of the evaluation framework and the results obtained.

B. AvalonAES128

To use the 128-bit version of AES2 (AES128) I simply needed to change the data width parameter of AESIfc and instantiate mkAES128 instead of mkAES32, as shown above. In order to use both mkAES32 and mkAES128 in the complete design I decided to create a new module (*mkAvalonAES128*) that uses the mkAES128 module. The data flow between mkAvalonAES128 and the Avalon framework is exactly the same as with the mkAvalonAES32 module. However because I used the 128-bit version of AESIfc all the wires have now a width of 128 instead of 32. Thus I needed to modify the calls to methods *request* and *read_response* in order to provide four 32-bit words instead of one. The data path provided by the Avalon framework is 32-bit wide. Thus I had all the data (plain text, keys, subkeys and cipher) in 32-bit words. In Verilog it is possible to concatenate four 32-bit words together by simply using the {} operator. In Bluespec however things are a little more complicated because the language allows several types of data, most of which eventually compile into the same Verilog wires. The Avalon interface transfers data using the *UInt* type. However Bluespec defines the concatenation operator {} only for data of type *Bit*. Thus I needed to make a double conversion to use the 128-bit version of AESIfc, as shown below:

```
Vector#(4, Reg#(UInt#(32))) dataIn1 <- replicateM(mkReg(0));
Vector#(4, Reg#(UInt#(32))) key <- replicateM(mkReg(0));
...
aes128.request(
unpack({pack(dataIn1[0]), pack(dataIn1[1]),pack(dataIn1[2]), pack(dataIn1[3])}),
    unpack({pack(key[0]), pack(key[1]),pack(key[2]), pack(key[3])}) );
```

TABLE I
RESULTS FOR SOFTWARE TEST

Module Tested	128-bit words per encryption	Number of Cycles	Throughput at 50 Mhz	Throughput at 550 Mhz
mkAvalonAES32	2 (256 bits)	60271	210 Kbps	2.33 Mbps
mkAvalonAES128	8 (1024 bits)	60271	840 Kbps	9.3 Mbps

The operators *pack* and *unpack* are used in Bluespec to transform between the *UInt* and *Bit* types.

After a successful compilation of the mkAvalonAES128 module, I started the synthesis of the complete design which includes the mkAvalonAES32 and mkAvalonAES128 modules, the NIOS2 processor and other components used by the Avalon framework. I encountered a new problem. The new design was using more M4Ks than are available on the Cyclone II FPGA and thus was not able to fit. The AES32 design uses 8 T-tables (see the original AES2 paper), each of 8 Kbits. This was implemented in Xilinx Virtex-5 FPGAs by using two 36 Kbits BRAMs. As the M4Ks provide 4 Kbits of memory I needed to use 16 M4Ks for mkAES32. The AES128 design uses four AES32 modules which means that mkAES128 uses 64 M4Ks. Thus, together mkAES32 and mkAES128 use 80 M4Ks from a total of 105 available. The NIOS II processor uses 3 M4Ks (as stated by the SOPC Builder). The JTAG interface also used M4Ks for its data buffers (even though the number of M4Ks was not mentioned). As a result the current configuration was over the limit of available M4Ks. I discovered that the JTAG interface can be configured to use standard registers instead of M4Ks. By using this option the design successfully synthesized.

The problem described above shows the limitations of average FPGAs from a resource perspective. Thus it is very important to choose the correct AES2 implementation (AES32 or AES128) on a new design. Bluespec has been shown to facilitate such design decisions, because it allows a polymorphic interface declaration. Thus, the designer only needs to specify the correct data type of the inteface and instantiate the corresponding module to choose between different module implementations.

AES2 also provides an unrolled version of AES128 that is able to give 128-bit of output every clock cycle. However this version could not fit into the Cyclone II FPGA so I did not implement it.

Next I describe the test methodology for the complete design, the problems encountered and the results obtained.

III. TESTS AND RESULTS

With the complete design compiled and synthesized I decided to test its performance. For this purpose I have included the following capabilities in both mkAvalonAES32 and mkAvalonAES128: a) start a new encryption by sending a write command from the Avalon framework; b) retrieve and reset a clock counter; c) perform a number of successive encryption operations using the stored data.

Based on the above capabilities I created a *C* test bench in the NIOS II IDE. This test bench runs on the DE2 board by sendig instructions to the NIOS II processor (implemented on the Cyclone II FPGA). By using standard method calls, variable assignments and printf commands, the *C* test bench can be used to transfer data between all the DE2 components through the Avalon framework (see file aestest.c in my source code). The JTAG interface can be used to send data back to the NIOS II IDE and show this data in a terminal. The test bench implements two tests as described below.

The first test (called *software test*) sends input data (plain text, key and subkeys) for the AES algorithm, reads the result and checks the number of cycles elapsed and the validity of the data. I've designed the mkAvalonAES32 and mkAvalonAES128 modules such that any request through the Avalon framework will block until the last encryption requested completes. Thus the *software test* sends the input data, makes an encryption request, retrieves the clock counter value and makes a get_result request. As any request is blocked during an encryption operation, the clock counter value is guaranteed to be returned just after the encryption operation is done. In this way we get an accurate latency design value. The exclusive synchronization between encryption operations and data access through the Avalon framework is implemented by the use of Bluespec rules. I explicitly define under which conditions each rule is allowed to fire and thus I am able to control the overall execution.

The performance results of the *software test* are presented in Table I. The Cyclone II FPGA is run at 50 Mhz by the DE2 board so all the results obtained refer to an execution at this frequency. The performance of this test (210 Kbps for mkAvalonAES32 and 840 Kbps for mkAvalonAES128) is very different from the results presented in AES2 (1.67 Gbps for AES32 and 6.7 Gbps for AES128). This is caused by two factors. Firstly, the *software test* represents a real scenario where data for AES encryption is passed from other modules and devices, whereas the AES2 tests present an ideal (but not realistic) situation where the data is statically included in the FPGA along with the AES2 design. Secondly the AES2 results are obtained from an FPGA running at 550 Mhz while my results are obtained from an FPGA running at 50 Mhz. I also present in Table I the expected performance of my design on an FPGA running at 550 Mhz.

The second test (called *hardware test*) performs a number of successive encryption operations on a given data and then checks the clock counter and the validity of results. The input data is assumed to be sent via the Avalon framework prior to the *hardware test* (I send the data with the *software test* which is run before this test). Thus the test bench for the *hardware test* first sends the number of requested encryptions (which also starts the test) and then makes a request for the clock counter. This assures that when the clock counter value returns it contains an accurate value of the test latency.

TABLE II
RESULTS FOR HARDWARE TEST, N = 10000 ENCRYPTIONS

Module Tested	128-bit words per encryption	Number of Cycles	Throughput at 50 Mhz	Throughput at 550 Mhz
mkAvalonAES32	2 (256 bits)	950088	134.7 Mbps	1.48 Gbps
mkAvalonAES128	8 (1024 bits)	950088	538.9 Mbps	5.9 Gbps

The performance results of the *hardware test* for n = 10000 encryptions are shown in Table II. This test is very similar to the tests used for AES2, as the data is already present in the FPGA and the sequence of encryption operations runs uninterrupted. The results also showcase this aspect. The mkAvalonAES32 test gives a throughput of 134.7 Mbps (1.48 Gbps at 550 Mhz) and mkAvalonAES128 achieves 538.9 Mbps (5.9 Gbps at 550 Mhz). The difference betweeen the expected results at 550 Mhz and the reported results for AES2 (1.67 Gbps for AES32 and 6.7 Gbps for AES128) can be explained by the lack of DSP blocks in my design. The AES2 results are based on the use of specialized DSP blocks to compute the XOR operation between AES state columns. I chose to use the simpler XOR version of AES2 to avoid making further changes to the Verilog code. However I must mention that *Quartus MegaFunction* tool can be used to instantiate DSP blocks for the Cyclone II FPGAs. I have not tried this approach but the reader is encouraged to do so if the exact AES2 results need to be obtained.

To check the correctness of the resulted design I have used the data provided with the AES2 test benches. Neither the mkAvalonAES32 nor the mkAvalonAES128 provide the expected result. I tried to debug this problem but the lack of simulation possibilities have made this task very difficult. As I mentioned previously I cannot use bluesim to simulate my design because I use imported Verilog. Modelsim cannot be used neither because Quartus does not provide me the Verilog source file for the M4K instantiation. The only solution left for simulation was to use the Quartus simulation environment. As the complete design is too complex to be simulated manually (i.e. set up each signal with the correct value at each clock cycle), I decided to make another Bluespec test that uses mkAvalonAES32. Unfortunately after many trials I was not able to see the desired signals using the Quartus simulation tool. Thus I was left with the NIOS 2 interface and DE2 components (LCD, LEDs) to debug the problem. The need of specific input data each clock cycle in the AES2 design and the limited time for this project have made the debugging task even more difficult.

IV. CONCLUSION

AES2 provides a fast implementation of AES for FPGAs. I have used AES2 to create a Bluespec implementation that allows a more parameterizable design. I have shown the simplicity of using any AES2 implementation (AES32 or AES128) in a large Bluespec design. By means of the Altera DE2 board I was able to test the complete design.

The performance results showcase two scenarios: a) real application, where AES data is sent from a different module or device; b) ideal situation, where the data is already in the FPGA and the module only needs to run the encryption process. The results obtained for the second scenario are very close to the results presented in AES2. This suggests the correctness of our design.

Unfortunately the test benches did not validate the correctness of the encryption operations. The lack of simulation possibilities made it difficult to find the problem. For a future investigation I mention two possible sources of the problem: a) the M4K contents and the way data is retrieved compared to the BRAM access; b) the sequence in which data is passed from the mkAvalonAES32 and mkAvalonAES128 modules to mkAES32 and mkAES128 respectively.

Based on the experience with AES2, I think the following changes would improve the available code and facilitate its further use:

- Make the state machine for AES32 such that it automatically goes back to the initial state. In the current version it needs a reset signal after each encryption.
- Add comments to the test benches. It is not clear to me for example why the test bench for the AES128 sends the initial key with one clock cycle delay compared with the test bench for AES32.

Overall, Bluespec seems a good solution for larger designs, especially in situations where a parameterizable model is required. However it has some limitations that can become an important problem. First of all, the Bluespec simulator, bluesim, does not currently accept designs with imported Verilog files. Secondly, the assignment between different data types is cumbersome, requiring many conversions. Thirdly BSV rules are meant to fire *at most* one time per clock cycle. As such the designer does not have full control over the execution, as it is the case with Verilog.

APPENDIX
VERILOG TO BLUESPEC WRAPPER FOR AES32 AND AES128

```
package AES;

// Declare the Bluespec module interface
interface AESIfc#(numeric type data_width);
   method Action request(UInt#(data_width) dataIn, UInt#(data_width) keyIn);
   method UInt#(data_width) read_response;
endinterface: AESIfc

// Declare the wrapper for the AES32 module, which creates the
// the Bluespec interface
import "BVI" aes32_dsp_8p =
   module mkAES32 (AESIfc #(32));

   method request (DIN, KEY) enable (START);
   method DOUT read_response ready ((*reg*) DONE);
   default_clock clk(CLK);
   default_reset rst(RST_N);
   schedule (request) SB (read_response);
   schedule (read_response) CF (read_response);
   schedule (request) C (request);
endmodule

// Declare the wrapper for the AES128 module
import "BVI" aes128_dsp =
   module mkAES128 (AESIfc #(128));

   method request (DIN, KEY) enable (START);
   method DOUT read_response ready ((*reg*) DONE);
   default_clock clk(CLK);
   default_reset rst(RST_N);
   schedule (request) SB (read_response);
   schedule (read_response) CF (read_response);
   schedule (request) C (request);
endmodule


// See the Bluespec reference manual page 127
// for more information on importing modules

endpackage: AES
```

REFERENCES

[1] Altera DE2 Development Board, http://www.altera.com/education/univ/materials/boards/unv-de2-board-faq.html.
[2] Bluespec, http://www.bluespec.com/.
[3] Cyclone II FPGA, http://www.altera.com/products/devices/cyclone2/cy2-index.jsp.
[4] Quartus Web Edition, http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html.
[5] NIOS II Processor, http://www.altera.com/products/ip/processors/nios2/ni2-index.html.
[6] "Fips 197: Advanced encryption standard," NIST, Tech. Rep., 2001.
[7] S. Drimer, T. Guneysu, and C. Paar, "Dsps, brams and a pinch of logic: New recipes for aes on fpgas," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 99–108, 2008.
[8] UG190: Virtex-5 User Guide, http://www.xilinx.com/support/.
[9] Bluespec impementation of AES2 for Altera DE2, http://www.cl.cam.ac.uk/~osc22/aes2b/.