A Template Based Interactive AES Generator

Donghua Wang1, Biman Tang2, Jin Sha2

*1Shanghai Jiao Tong University, alex\_wang@126.com*

*2Shanghai Jiao Tong University*

Abstract: A template based AES generator was proposed in this work. We explored a new language construction called *map* to describe the reconfigurable components of the AES circuit. A precompiler could parse the language constructions been used in the template, and then translates them into Verilog code depending on the user inputs during the interactive compiling process. Thus, users can gain tradeoffs between area and time easily, without changing any code. Circuits generated by this system vary in architecture as well as in resource consuming and time cost. 15 implements were obtained under 180*MHz* frequency from the template, by using the TSMC 0.18*u*m library. The maximum throughput is 860*Mbps*, and the minimum area is only 6004 nand2 gates.

*Keywords:* AES; Precompiler; Verilog RTL; Template; Time-Space Trade-off; Interactive

# INTRODUCTION

Hardware description language[1] (HDL) based IP synthesis has been the industry standard in recent years. However, it has a number of problems. First, Verilog and VHDL are so low level that they are often compared to assembly languages in terms of programmability. Second, the simulation and verification is the most time-consuming step in IP design cycle. Third, while most HDLs offer design libraries of basic building blocks, these are largely limited to the hardware circuit level, and are often inadequate for programming large, complex but common algorithms such as those used in cryptography and image processing. Today, designs for these algorithms require thousands of lines of Verilog code which is extremely expensive to debug and maintain. Last, because HDLs do not offer the capability of high level abstraction, it is not easy to reconfigure the functionality of an existing design. For example, for a given design of an AES[2] algorithm, if the user prefers to trade die space for speed, a common approach is to unroll a loop a number of times and execute it in parallel within a clock cycle. Such unrolling cannot be achieved in Verilog without substantial code change.

To address some of these problems, a range of new solutions have been proposed. There are two main categories, new and powerful languages to make programming easier and generators[9] to help users to get a circuit with several clicks. SystemC[8] introduces an event driven simulation kernel and some ability in describing hardware in C. SystemVerilog[13], on the other hand, aids hardware designer by raising the abstract level of Verilog with convenient programming constructs and some object-oriented concepts. Spiral[4][11] project, which develops systems that generate hardware designs from high level mathematical representations of certain DSP transforms. However, none of the above methods provides opportunities to users to re-configure the circuit to gain time-space tradeoffs.

In our work, a template based interactive generator was explored. Users can seek tradeoffs during the generating process via selecting the number of s-box and mix-column sub-module instantiated in the architecture. A new language construction *map* was proposed to describe circuit in algorithm level that depicts the mathematical computation meaning of the circuit. The *map* construction can be embedded with Verilog code, so it can be used anywhere as long as there are such circuit structures.

# Circuit Component and Language construction

We explored the reconfigurable circuit component in AES architecture and designed a new language construct *map* accordingly to describe it, which portraits the computation meaning of the reconfigurable structure.

The *map* is used to model the circuit that does some operations upon a list of data. It's also similar to higher order function “map” in Haskell[3], which takes two inputs - a function, and a list, and then applies this function to every element in the list. In Verilog we don't have list type. However we can view a *reg* or *wire* as a bit list, and every time we applies the function (actually we should call it a *module* in circuit design) to a slice of the bits. Figure 1 shows the *map* structure, as an example, we can think “Module” has the functionality that adds 1 to the input data list. Here comes the syntax of *map*:

*map(md,data,datalen,maplen)*

where *md* is a module identifier, *data* is the bit-list name (variable name of a *reg* or *wire*), *datalen* is the length of the bit-list and *maplen* is the length of the bit slice that applied to the module each time.

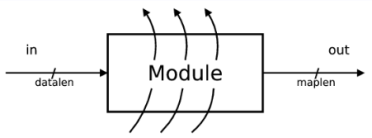


Figure 1: Map circuit structure

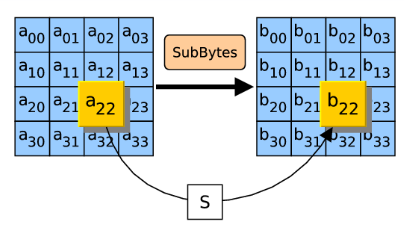
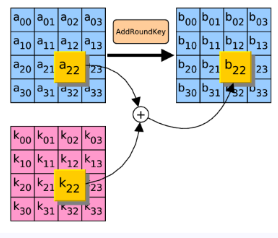
# The AES Generator

Basically, our generator consists of two parts, the template and a precompiler.

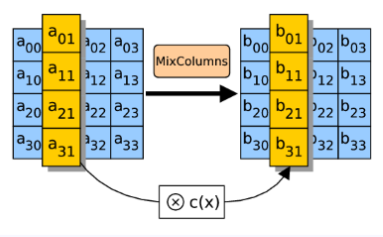
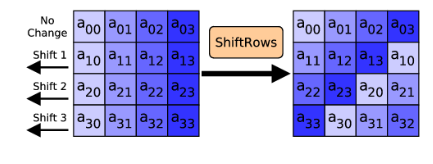
1. *AES*

The AES algorithm is a symmetric block cipher that processes data block of 128 bits. Basically, AES operates on a 4\*4 column-major order matrix (called *state*) of bytes. The key size used for an AES cipher can be different, and it specifies the number of repetitions of transformation rounds that converts the input, called the plaintext, into the final output, called the ciphertext. In each round there are four operations:

* AddRoundkey(A.R.) - A round key is added to the *state* by a simple bitwise XOR operation. See Figure 2(a).
* SubBytes(S.B.) – The SubBytes operations is a non-linear byte substitution that operates on each byte of the *state* using s substitution table, *s-box*. See Figure 2(b).
* ShiftRows(S.R.) – In the ShiftRows operation, the bytes in the last three rows of the *state* are cyclically shifted over different number of bytes. See Figure 2(c)
* MixColumns(M.C.) – Operates on the columns of the *state* using a linear transformation. See Figure 2(d).



(a) AddRoundkey (b) SubBytes



(c) ShiftRows (d) MixColumn

Figure 2. AES operations

1. AES Template

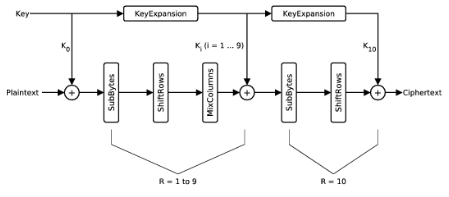


Figure 3: AES dataflow

Figure 3 shows the dataflow of AES with 128-bit key. We should be aware of that the size of input text is 128-bit, however SubBytes operation gets 8-bit data flow into the s-box each. So there come different configurations of the number of *s-box* instances. For example, we can use only 1 *s-box* and do 16 times of SubBytes operations, or use 4 *s-boxes* and do just 4 times of SubBytes operations. Same kinds of configurations were found in the MixColumn step.

By using the *map* language construction, we can describe this kind of reconfigurability very well. Code snippet below is the key points of the template:

always @(posedge clk)

begin

if(load\_i) data <= data\_i;

else if(addkey\_i) data <= data\_w;

else if(subbytes\_i) map(sbox,data,128,8);

else if(mixcolumn\_i) map(mix,data,128,8);

end

Where *sbox* and *mix* are two sub-modules, signals *subbytes\_i* and *mixcolumn\_i* control the invoking of these two sub-modules. Designers don’t have to concern the details of the invoking process, such as how data connects with *sbox* and *mix* in desired order. Furthermore, designers can choose the number of sub-module instances during the compiling stage.

1. *The Precompiler*

Basically, the precompiler has the template as input and outputs a copy of Verilog RTL code. However, as an interactive compiler, ours works different from traditional compilers. This precompiler can discover the *map* constructions in the template and dig out all the possible configurations, so that users could make their decision on how many module instances they want to use in the circuit architecture via interacting with the precompiler. And users don’t have to concern the module schedule issues.

In this work, there are two *maps* in the template, so two configurations stages exists during the compiling process. Figure 4 shows the procedure.

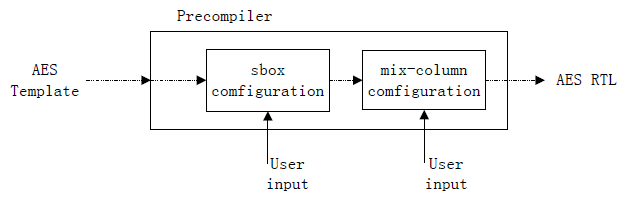


Figure 4: Compiling process

The template file is parsed by the precompiler. At first, the *sbox* *map* is discovered. The precompiler pauses and provides the configuration information and asks for user’s optimization choice. Then the *mix map* is handled as the same. Finally, based on the user input, the generator makes modifications on the template and outputs the AES RTL descriptions.

# Results and Discussion

15 kinds of optimizations choices for AES are explored in our experiments. The synthesis results of these implementations vary in architectures as well as in resource consuming and time cost. We synthesized these RTL descriptions in Synopsys using TSMC 0.18*u*m standard cell library. Table 1 shows the main measuring dimensions of these circuits. Column “*conf.”* is the number of *s-box* and *mix-column* sub-modules instances pair, column “Gates” is the number of nand2-gate cost of the circuit and column “Cycles” means the clock cycles needed to encrypt 128-bit data by this circuit each time. We could see that, configuration with the most modules has 2.5 times of gates number of that with fewest modules and only 20% of the cycles. Thus, the reconfigurations do make sense, since users can get considerable trade-offs between space and time.

Table 1: Area and Time cost of AES implementations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Conf. | Gates | Cycles | Conf. | Gates | Cycles |
| (1,1) | 6004 | 218 | (4,4) | 7828 | 71 |
| (1,2) | 6314 | 200 | (8,1) | 8599 | 78 |
| (1,4) | 6824 | 191 | (8,2) | 8875 | 60 |
| (2,1) | 6398 | 138 | (8,4) | 9315 | 51 |
| (2,2) | 6742 | 120 | (16,1) | 11310 | 63 |
| (2,4) | 7164 | 111 | (16,2) | 11686 | 50 |
| (4,1) | 7057 | 98 | (16,4) | 12069 | 41 |
| (4,2) | 7350 | 80 |  |  |  |

Table 2 is about the comparison with other implement-tations. Ours provide many kinds of configurations which have different gates, and can run under a high frequency.

Table 2: Comparison with other AESs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Work | Tech | Gates | Freq | Throughput | Conf |
| [12] | 0.18um | 7226 | 138MHZ | 327Mbps | 1 |
| [6] | 0.35um | 3595 | 100KHZ | 12.6Kbps |
| [10] | 0.60um | 8541 | 50MHZ | 70Mbps |
| [5] | 0.18um | 6986 | 180MHZ | 114-360Mbps | 3 |
| Ours | 0.18um | 6004-12069 | 180MHZ | 161-860Mbps | 15 |

For silicon technology the area-time tradeoff has been nicely formalized by theorists as AT (area and time) bounds. Author in paper[7] shows that:

Figure 5 is the two-dimension diagram between area-time of our AES implementations. The blue line is about gates and cycles of each implementation. And the green line is function *xy=750000*. It reveals that the area-time bounds of our implementations approximately in keeping with, when *n=1* and *constant = 750000*. That is to say, circuits generated by our system are efficient in theory.

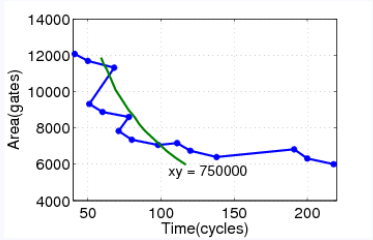


Figure 5: Area-Time diagram

# Conclusion

We proposed a new language construction that can be embedded with Verilog, and a precompiler that translates this language construction into Verilog. With these two technologies, we created an AES template and an interactive generator that gives users the ability to do optimizations during generating the circuit. Experiments reveal that those implementations based on this template are fairly good.

# References

1. R. Waxman and M. Israel. Hardware description languages. pages 768-773, 2003.
2. N.I. of Standard and Technology (U.S.). *Advanced Encryption Standard* (AES). FIPS Publication, 2001.
3. J. G. Bryan O'Sullivan, Don Stewart. *Real World Haskell*. Southeast University Press, 2010.
4. Spiral project website. [www.spiral.net](http://www.spiral.net).
5. Y. Fan, T. Ikenaga, Y. Tsunoo, and S. Goto. A low-cost reconfigurable architecture for aes algorithm. In World Academy of Science, Engineering and Technology, pages 41-43, 2008.
6. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for rfid systems using the aes algorithm. In CHES, pages 357-370, 2004.
7. M. Flynn. Area-time-power and design effort: the basic tradeoffs in application specific systems. 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors, 2005.
8. T. Grötker, S. Liao, G. Martin, and S. Swan. System Design with SystemC. Springer, 2002.
9. G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel. Automatic generation of customized discrete fourier transform IPs. In DAC, pages 471-474, 2005.
10. N. Pramstaller, S. Mangard, S. Dominikus, and J.Wolkerstorfer. Efficient aes implementations on asics and fpgas. In AES Conference, pages 98-112, 2004.
11. M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y.Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. In Proceedings of IEEE, pages 232-275, 2005.
12. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact rijndael hardware architecture with s-box optimization. pages 239-254, 2001.
13. S. Sutherland, S. Davidmann, and P. Flake. SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling. Springer, second edition, 2006.