



AES Behavioral Model Specification

*Author: scheng
schengopencores@opencores.org*

**Rev. 0.1
July 4, 2013**

This page has been intentionally left blank.

Revision History

Rev	Date	Author	Description
.			
0.1	07/04/2013	scheng	Initial release

Contents

INTRODUCTION.....	1
USAGE.....	2
CLASS PROPERTIES AND METHODS	4
PROPERTIES	4
METHODS.....	5
EXAMPLES	8
EXAMPLE 1 - VERIFICATION OF AN AES DECRYPTION IP	8
EXAMPLE 2 - VERIFICATION OF PER CYCLE (ROUND) OUTPUT OF AN AES IP	10
VERIFICATION.....	14
REFERENCES.....	15

1

Introduction

The AES behavioral model is an un-timed SystemVerilog class which implements the decryption algorithm described in the FIPS-197 specification. The objective is to provide a tool to facilitate the verification of AES IPs developed in HDL. This version only supports decryption. The model can be used as a golden model or test vector generator in HDL simulation of AES designs. It provides a quick and easy way to output known good values (plaintext) to be compared with the output of the unit under test (uut). Unlike other reference models written in C/C++, this model is implemented purely in SystemVerilog and can be instantiated directly in the testbench, eliminating the need to interface with an external language in an HDL simulation environment. The model is an un-timed SystemVerilog class with no timing information hard-coded. This allows the flexibility to further enclose the model in a wrapper with timing details to form a timed AES decryption behavioral model to cope with various simulation scenarios.

Highlights

- Implements AES decryption algorithm defined in FIPS-197
- Supports 128/192/256 bit key length
- Native SystemVerilog class eliminates the need to interface with external language in HDL testbench
- Runtime selectable single-round mode to generate per round intermediate result or run-through mode to generate the final plaintext directly
- Verified against selected test vectors provided in FIPS-197, SP800-38a, and AESAVS. Testbench provided.

2

Usage

The following pre-defined data types are provided for use in the testbench, each represents a SystemVerilog class of the corresponding key length

aes128_decrypt_t

aes192_decrypt_t

aes256_decrypt_t

To use the model in your testbench, declare a variable of the appropriate type and run the constructor, as shown in the example below

```
module my_testbench;
....
    aes128_decrypt_t my_decryptor; // Declare class variable
    ....
    initial begin
        my_decryptor = new; // Run constructor
    end
endmodule
```

To use the model in a testbench, follow the calling sequence below.

Example for 256-bit key, run-through mode.

```
aes256_decrypt_t my_aes_decryptor; // Class variable for 256-bit decryptor
bit [0:127] pt; // Plaintext
....
my_aes_decryptor = new; // Create a decryptor instance
my_aes_decryptor.KeyExpand(256'h.....); // Load 256-bit crypto key to model
my_aes_decryptor.LoadCt(128'h.....); // Load ciphertext
my_aes_decryptor.run(0); // Run through all decryption round
pt = my_aes_decryptor.GetState(); // Get plaintext
.....
```

Example for 128-bit key, single-round mode.

```
aes128_decrypt_t  my_aes_decryptor;    // Class variable for 128-bit decryptor
bit [0:127]       pt;                  // Plaintext
....
my_aes_decryptor = new;                // Create a decryptor instance
my_aes_decryptor.KeyExpand(128'h.....); // Load 128-bit crypto key to model
my_aes_decryptor.LoadCt(128'h.....);    // Load ciphertext
do
begin
    my_aes_decryptor.run(1);            // Run one round only
    $display("State=%h",my_aes_decryptor.GetState()); // Print per-round result
end
while (my_aes_decryptor.done == 0);
pt = my_aes_decryptor.GetState();      // Get plaintext
....
```

3

Class Properties and Methods

Inside the model the State, Key Schedule, and round counter are maintained as protected properties which can be accessed through dedicated methods. Other properties like the done and loaded flags are exposed to the outside world and can be accessed directly. By calling the exposed methods the model can be driven to generate known good results at different point during simulation for verification of the uut.

Properties

Name	done
Declaration	bit done
Description	Done flag indicated the end of decryption. Initialized to '0' in new() and LoadCt(). Set to '1' in run() after the last round is completed. When done='1', state contains valid plaintext. Testbench codes can check this flag to determine if the last round is completed.

Name	loaded
Declaration	bit loaded
Description	Flag indicates whether a valid ciphertext is loaded to the model. Initialized to '0' by new(). Set to '1' when LoadCt() is called. Reset to '0' by run() when the last round is finished. Testbench codes can check this flag to determine if the model is loaded with a valid ciphertext.

Methods

Name	LoadCt()
Declaration	task LoadCt(bit [0:127] ct)
Properties modified	state, loaded, done
Return value	None
Description	Load valid ciphertext to state. ct is a 128-bit vector holding the ciphertext, with 1 st byte in ct[0:7], 2 nd byte in ct[8:15], ...and so on. Make sure LoadCt() is called before calling run() to ensure run() doesn't work on garbage. LoadCt() sets loaded to '1' and clears done to '0'.

Name	GetState()
Declaration	function bit [0:127] GetState
Properties modified	None
Return value	Current state content in a 128-bit vector. Decrypted plaintext at the end of last round.
Description	Returns current State as a 128-bit vector. State[0][0] in bit[0:7], State[1][0] in bit [8:15],.. ..., and so on. Once the last are completed, State contains the decrypted plaintext. GetState() can be called at any time.

Name	KeyExpand
Declaration	task KeyExpand(bit [0:4*8*Nk-1] key)
Properties modified	keysch
Return value	None
Description	Load crypto key to model and compute Key Schedule. KeyExpand() should be called before calling run() to make sure that a valid Key Schedule is available for run() to use during decryption. Once KeyExpand() is completed, a valid Key Schedule is stored in the property keysch and stays there until KeyExpand() is invoked again. Therefore a single Key Schedule can be used in multiple decryption runs if there is no change of crypto key.

Name	GetCurrKsch
Declaration	function bit [0:127] GetCurrKsch
Properties modified	None
Return value	Key Schedule for the current round
Description	Returns the Key Schedule for the current round. . A protected property <code>curr_round</code> keeps track of which round the decryption process is in. More precisely, <code>curr_round</code> holds the round that will be executed next time <code>run()</code> is called. So the round key returned by <code>GetCurrKsch()</code> is the one that will be used in next call of <code>run()</code> .

Name	LookupKsch
Declaration	function bit [0:127] LookupKsch(int unsigned r);
Properties modified	None
Return value	Key Schedule for the specified round
Description	Returns the Key Schedule for round specified by <code>r</code> .

Name	GetCurrRound
Declaration	function int unsigned GetCurrRound
Properties modified	None
Return value	Unsigned integer indicating the current round
Description	Call <code>GetCurrRound()</code> to find out the round number which will be executed next time <code>run()</code> is called. This method is provided for use with single-round mode (see description on <code>run()</code> below) so that the testbench codes can monitor the progress of the decryption and tell exactly which round the model is in.

Name	<code>run</code>
Declaration	task <code>run(int mode)</code>
Properties modified	<code>state</code> , <code>loaded</code> , <code>done</code> , <code>curr_round</code> (protected)
Return value	None
Description	Run the decryption process. <code>Mode=0</code> for run-through mode, <code>mode=1</code> for single-round mode. In run-through mode <code>run()</code> runs from the current round all the way to the last round. In single-round mode <code>run()</code> runs one round and

returns. Single round mode is for scenarios where intermediate result for each round is needed, e.g. to verify the State of the uut at each clock cycle. Before calling `run()`, make sure the model is loaded with `LoadCt()` and `KeyExpand()`.

4

Examples

Example 1 - Verification of an AES128 decryption IP

This example shows a sample SystemVerilog testbench for verification of an AES decryption IP (uut). The uut output is compared against the reference model output. In this example we don't care about the intermediate results, so the model is run with run-through mode to get the plaintext right away.

```
`timescale 1ns/1ps

// Source code for our reference model
`include "aes_beh_model.sv"
// Source code of the AES IP to be verified
`include "aes128_dec.sv"

module aes128_dec_tb;

    logic    [0:127]  ct;        // Ciphertext input to uut
    logic    ct_vld;        // High indicates valid ciphertext present
    wire     ct_rdy;        // High indicates uut ready to accept new ciphertext

    logic    [0:127]  kt;        // Key text input to uut
    logic    kt_vld;        // High indicates valid key text present
    wire     kt_rdy;        // High indicates uut ready to accept new key text

    wire     [0:127]  pt;        // Plaintext output from uut
    wire     pt_vld;        // High indicates valid plaintext present from uut

    logic    clk;          // System clock
    logic    rst;          // Active high reset

    `define PERIOD 10
    `define T (`PERIOD/2)
    `define Tcko 1

    `define WAIT_N_CLK(num_of_clk) repeat(num_of_clk) @(posedge clk); #(`Tcko)

    // Declare a variable for our reference model. Output from the uut
    // will be verified against this reference model.
```

```

aes128_decrypt_t ref_model;

// Instantiate decryptor IP
aes128_dec uut( .clk(clk),
               .rst(rst),

               .ct(ct),           // Ciphertext
               .ct_vld(ct_vld),
               .ct_rdy(ct_rdy),

               .kt(kt),          // Key text
               .kt_vld(kt_vld),
               .kt_rdy(kt_rdy),

               .pt(pt),          // Plaintext
               .pt_vld(pt_vld)
               );

// Task for loading key text to uut
task set_kt(input [0:127] x);
    wait (kt_rdy);
    kt = x;
    kt_vld = 1;
    `WAIT_N_CLK(1);
    kt_vld = 0;
    `WAIT_N_CLK(1);
endtask

// Task for loading ciphertext to uut
task set_ct(input [0:127] x);
    wait (ct_rdy);
    ct = x;
    ct_vld = 1;
    `WAIT_N_CLK(1);
    ct_vld = 0;
    `WAIT_N_CLK(1);
endtask

// Clock generator
always
begin
    clk <= 1;
    #(`T);
    clk <= 0;
    #(`T);
end

initial begin
    // Create an instance of the reference model
    ref_model = new;

    // Initialize signals
    rst = 1;
    kt_vld = 0;
    
```

```
ct_vld = 0;
`WAIT_N_CLK(3);

// Deactivate reset
rst = 0;
`WAIT_N_CLK(1);

// Load key text to model
ref_model.KeyExpand(128'h000102030405060708090a0b0c0d0e0f);
// Load ciphertext to model
ref_model.LoadCt(128'h69c4e0d86a7b0430d8cdb78070b4c55a);

// Write key text to uut
set_kt(128'h000102030405060708090a0b0c0d0e0f);
// Write ciphertext to uut
set_ct(128'h69c4e0d86a7b0430d8cdb78070b4c55a);
// Wait until plaintext is available from uut
wait (pt_vld);
// Execute reference model to obtain known good result
ref_model.run(0);

// Print uut and model output
$display("pt=%h expected=%h",pt,ref_model.GetState());
// Verify uut output against model output
if (pt != ref_model.GetState()) $display("****Mismatch");

$stop;
end
endmodule
```

Example 2 - Verification of per cycle (round) output of an AES IP

This example demonstrates the use of single-round mode of the model. Here the output of the uut (which is the State) is verified against the reference model out on a per cycle basis. The model is driven with single-round mode so that run() return after finishing every round. In the following testbench run() is called at every clock cycle to obtain the value of the State after each round, which is then compared against the uut output.

There are two initial blocks in this testbench. The first one is a stimuli generator which feeds ciphertext and crypto key to the uut. The second one is a checker process which verifies the uut output against the reference model for every clock cycle.

```

`timescale 1ns/1ps

// Source code for our reference model
`include "aes_beh_model.sv"
// Source code of the AES IP to be verified
`include "aes128_dec.sv"

module aes128_dec_tb;

    logic    [0:127]  ct;
    logic    ct_vld;
    wire     ct_rdy;

    logic    [0:127]  kt;
    logic    kt_vld;
    wire     kt_rdy;

    wire     [0:127]  pt;
    wire     pt_vld;

    logic    clk;
    logic    rst;

    logic    [0:127]  round_key;

    `define PERIOD 10
    `define T (`PERIOD/2)
    `define Tcko 1

    `define WAIT_N_CLK(num_of_clk) repeat(num_of_clk) @(posedge clk); #(`Tcko)

    // Declare a variable for the 128 bit decryptor model. Output from the uut
    // will be verified against this reference model.
    aes128_decrypt_t_ref_model;

    // Instantiate decryptor IP
    aes128_dec uut( .clk(clk),
                   .rst(rst),

                   .ct(ct),                // Ciphertext
                   .ct_vld(ct_vld),
                   .ct_rdy(ct_rdy),

                   .kt(kt),                // Key text
                   .kt_vld(kt_vld),
                   .kt_rdy(kt_rdy),

                   .pt(pt),                // Plaintext
                   .pt_vld(pt_vld)
                   );

    // Task for loading key text to uut
    task set_kt(input [0:127] x);
        wait (kt_rdy);
    endtask

```

```
        kt = x;
        kt_vld = 1;
        `WAIT_N_CLK(1);
        kt_vld = 0;
        `WAIT_N_CLK(1);
    endtask

    // Task for loading ciphertext to uut
    task set_ct(input [0:127] x);
        wait (ct_rdy);
        ct = x;
        ct_vld = 1;
        `WAIT_N_CLK(1);
        ct_vld = 0;
        `WAIT_N_CLK(1);
    endtask

    // Clock generator
    always
    begin
        clk <= 1;
        #(`T);
        clk <= 0;
        #(`T);
    end

    // This initial block applies stimuli to the uut
    initial begin
        // Initialize signals
        rst = 1;
        kt_vld = 0;
        ct_vld = 0;
        `WAIT_N_CLK(3);

        // Deactivate reset
        rst = 0;
        `WAIT_N_CLK(1);

        // Write key text to uut
        set_kt(128'h000102030405060708090a0b0c0d0e0f);
        // Write ciphertext to uut. Decryption process starts immediately
        // once ciphertext is loaded to uut.
        set_ct(128'h69c4e0d86a7b0430d8cdb78070b4c55a);

        // All stimuli have been applied at this point
    end

    // This initial block is a checker process which monitors the uut output at each
    // clock cycle and verify against the reference model.
    initial begin
        // Create an instance of the reference model
        ref_model = new;
```



```
// Wait for testbench to write key text to uut
wait (kt_vld);
// Load same key text to reference model
ref_model.KeyExpand(kt);

// Wait for testbench to write ciphertext to uut
wait (ct_vld)
// Load same ciphertext to reference model
ref_model.LoadCt(ct);

// uut executes one decryption round per clock cycle. pt contains the State after each
// round. pt is compared against the reference model output after each clock cycle.
do begin
    `WAIT_N_CLK(1);
    // Get round key for the current round. After ref_model.run() is called the internal
    // round counter will be updated and points to next round.
    round_key = ref_model.GetCurrKsch();
    ref_model.run(1); // Run reference model for a single round

    // Print uut and model output
    $display("round key=%h State=%h expected=%h", round_key,pt,
ref_model.GetState(););
    // Compare uut output with reference model
    if (pt != ref_model.GetState();) $display("****Mismatch");
end
while (~pt_vld); // Repeat until uut finished all decryption rounds

// Print plaintext from uut and refrence model
$display("pt=%h expected=%h",pt,ref_model.GetState());

// Verify uut output against model output
if (pt != ref_model.GetState();) $display("****Mismatch");

$stop;
end
endmodule
```

5

Verification

This model has been verified with the following test vector sets

- FIPS-197, Appendix C
- NIST Special Publication 800-38A 2001 Edition (SP800-38a), Appendix F.1.2, F.1.4, F.1.6
- The Advanced Encryption Standard Algorithm Validation Suite (AESAVS), Appendix B, C, D, E

A sample testbench with modelsim do script is provided under the sim/ directory.

6

References

1. Advanced Encryption Standard (AES) (FIPS PUB 197)
2. NIST Special Publication 800-38A 2001 Edition
3. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)