

AES Behavioral Model  
Specification

*Author: scheng*

*schengopencores@opencores.org*

**Rev. 0.2**

**August 12, 2013**

*This page has been intentionally left blank.*

**Revision History**

| **Rev.** | **Date** | **Author** | **Description** |
| --- | --- | --- | --- |
| 0.1 | 07/04/2013 | scheng | Initial release. Decryption only. |
| 0.2 | 08/12/2013 | scheng | Added support for encryption.  Typedefs for decryption unchanged. Three new typedefs added for encryption.  Additional testbench and test vectors for encryption included. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Contents**

Introduction 1

Usage 2

Class properties and methods 4

properties 4

methods 5

examples 8

Example1 - verification of an aes decryption ip 8

example 2 - verification of per cycle (round) output of an aes IP 10

verification 14

references 15

Introduction

The AES behavioral model is an un-timed SystemVerilog class which implements the encryption and 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. 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 / ciphertext) 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 SystemVeriog 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 behavioral model to cope with various simulation scenarios.

### Highlights

* Implements AES encryption and 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 ciphertext/plaintext directly
* Verified against selected test vectors provided in FIPS-197, SP800-38a, and AESAVS. Testbench provided.

Usage

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

|  |  |
| --- | --- |
| Encryption | Decryption |
| aes128\_encrypt\_t | aes128\_decrypt\_t |
| aes192\_encrypt\_t | aes192\_decrypt\_t |
| aes256\_encrypt\_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

Example calling sequences for use in testbench are shown below.

Example for 256-bit decryption, 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\_descryptor.run(0); // Run through all decryption round

pt = my\_aes\_descryptor.GetState(); // Get plaintext

…..

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

aes128\_encrypt\_t my\_aes\_encryptor; // Class variable for 128-bit encryptor

bit [0:127] ct; // Ciphertext

....

my\_aes\_encryptor = new; // Create a encryptor instance

my\_aes\_encryptor.KeyExpand(128'h.......); // Load 128-bit crypto key to model

my\_aes\_encryptor.LoadPt(128'h.........); // Load plaintext

do

begin

my\_aes\_encryptor.run(1); // Run one round only

$display(“State=%h”,my\_aes\_encryptor.GetState()); // Print per-round result

end

while (my\_aes\_encryptor.done == 0);

ct = my\_aes\_encryptor.GetState(); // Get ciphertext

….

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(), LoadCt() and LoadPt(). 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() or LoadPt() 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 or plaintext. |

### Methods

|  |  |
| --- | --- |
| Name | LoadCt() |
| Declaration | task LoadCt(bit [0:127] ct) |
| Properties modified | state, loaded, done |
| Return value | None |
| Description | For decryption model only. Load valid ciphertext to state. ct is a 128-bit vector holding the ciphertext, with 1st byte in ct[0:7], 2nd 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 | LoadPt() |
| Declaration | task LoadPt(bit [0:127] pt) |
| Properties modified | state, loaded, done |
| Return value | None |
| Description | For encryption model only. Load valid plainrtext to state. pt is a 128-bit vector holding the plaintext, with 1st byte in pt[0:7], 2nd byte in pt[8:15], …and so on. Make sure LoadPt() is called before calling run() to ensure run() doesn’t work on garbage. LoadPt() 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.  For encryption model, State holds the final ciphertext at the last round.  For decryption model, State holds the final plaintext at the 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. For encryption call GetState() to obtain the ciphertext at the last round. For decryption call GetState() to obtain the plaintext at the last round. 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 encryption/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 encryption/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 curr\_round keeps track of which round the encryption/decryption process is in. More precisely, curr\_round holds the round that will be executed next time run() is called. So the round key returned by GetCurrKsch() is the one that will be used in next call of run(). |

|  |  |
| --- | --- |
| 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 r. |

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

|  |  |
| --- | --- |
| Name | run |
| Declaration | task run(int mode) |
| Properties modified | state, loaded, done, curr\_round (protected) |
| Return value | None |
| Description | Run the encryption/decryption process. Mode=0 for run-through mode, mode=1 for single-round mode. In run-through mode run() runs from the current round all the way to the last round. In single-round mode run() 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 either LoadPt() or LoadCt() and KeyExpand(). |

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);

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 refence 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

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.1.1-1.6
* The Advanced Encryption Standard Algorithm Validation Suite (AESAVS), Appendix B, C, D, E

Two sample testbenches, one for encryption and the other for decryption, are provided the bench/ directory. Modelsim do scripts are included in the sim/ directory.

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)