AES Decryption Core for FPGA

Specification

Rev. 0.1.2 Preliminary

Author

scheng

schengopencores@opencores.org

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK

REVISION HISTORY

Rev.	Date	Author	Description
0.1	27 Jan 2014	scheng	First release
0.1.1	11 Feb 2014	scheng	Revised device utilization no. for 192-bit
0.1.2	7 May 2015	scheng	Updated benchmark data with Vivado 2015.1 results. Added benchmark for Kintex UltraScale device.

CONTENT

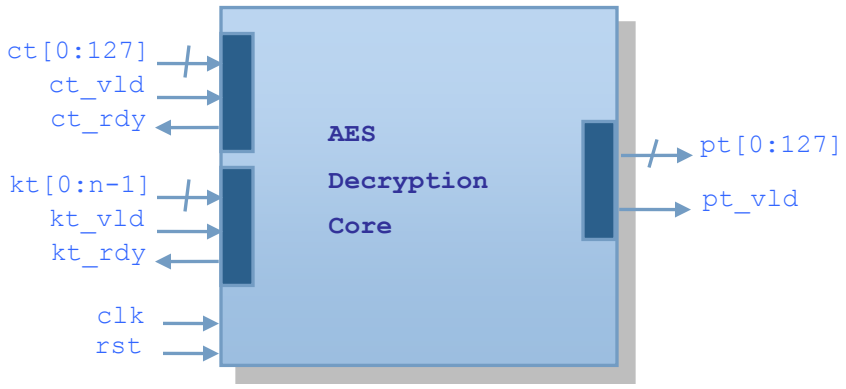
INTRODUCTION

The AES Decryption Core for FPGA implements the decryption portion of the AES (a.k.a. Rijndael) algorithm described in the FIPS-197 specification. Key lengths of 128 / 192 / 256 bits are supports, each with a separate instantiation wrapper. The core logic is carefully designed to take advantage of 6-input lookup table (LUT6) based FPGA architecture. As a result, it can achieve a peak throughput of over 3Gbps for 256-bit key, yet occupies about 2000 LUTs only. The core has been verified with random test vectors as well as selected test vectors in FIPS-197, SP-800a, and AESAVS specifications.

HIGHLIGHTS

- Supports 128/192/256-bit AES decryption.
- Separate wrappers for each key length. Not changeable at runtime.
- Key expansion takes 11/13/15 clock cycles for 128/192/256-bit key. The computed key schedule is stored internally and can be used on multiple ciphertext.
- Once the key schedule is computed, decryption of each 128-bit ciphertext takes 11/13/15 clock cycles.
- Separate interfaces for ciphertext, key text, and plaintext, with simple valid/ready style handshaking.
- Fully synchronous design with one clock domain only.
- Source code in SystemVerilog

TOP LEVEL SYMBOL



BENCHMARKS

Xilinx Kintex xc7k325tffg900-3

	128-bit	192-bit	256-bit
LUT	1865	2350	2033
FF	310	443	448
BRAM	0	0	0
Latency w/ key switching	22 clk	26 clk	30 clk
w/o key switching	11 clk	13 clk	15 clk
Fmax	369MHz	361MHz	365MHz
Peak throughput	4.293Gbps	3.554Gbps	3.114Gbps

Xilinx Kintex UltraScale xcku040-ffva1156-2-e

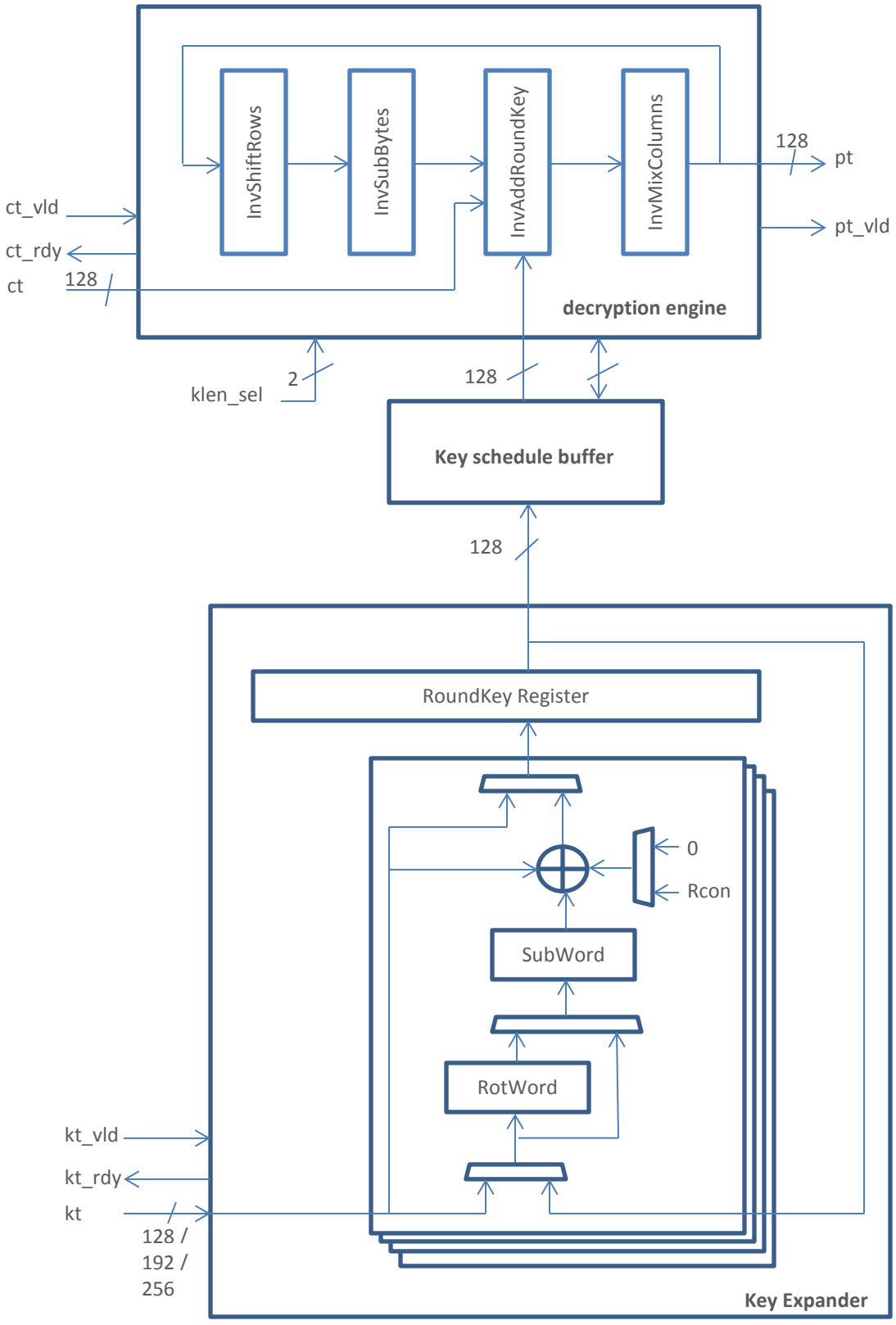
	128-bit	192-bit	256-bit
LUT	1791	2269	1969
FF	299	441	438
BRAM	0	0	0
Latency w/ key switching	22 clk	26 clk	30 clk
w/o key switching	11 clk	13 clk	15 clk
Fmax	380MHz	364MHz	375MHz
Peak throughput	4.421Gbps	3.584Gbps	3.2Gbps

Test conditions

For the purpose of benchmarking, the core is wrapped in a shift-register-like structure to reduce I/O pin count, synthesized and implemented with Xilinx Vivado 2015.1 using “Performance_Explore” implementation strategy with a period constraint. Target devices are Xilinx Kintex family xc7k325tffg900-3 and Kintex UltraScale xcku040-ffva1156-2-e.

The latency is a measure of the no. of clock cycles starting from the arrival of the key text and ciphertext to the clock edge when the plaintext is available at the output. That is the sum of the key expansion latency and the decryption engine latency. For subsequent ciphertext blocks which use the same key text as before, key expansion latency is zero since the previously computed key schedule will be re-used, only the decryption engine latency counts.

ARCHITECTURE



The block diagram of the AES decryption core is shown in the figure above. The core accepts ciphertext and key text from their respective interface, performs the AES decryption algorithm described in FIPS-197 specification, and outputs the plaintext at the pt interface. The decryption engine and key schedule buffer are common to all key lengths, while there is a separate key expander for each supported key length. For ease of use, a separate wrapper is provided for each key length which instantiates the proper key expander and other modules. Dynamic switching of key length at runtime is not supported.

The decryption engine implements the inverse cipher algorithm in figure 12 of the FIPS-197 specification. Key length is selected via klen_sel[0:1], which is pulled to the right value in the wrappers provided. The transformations InvShiftRows, InvSubBytes, InvAddRoundKey, and InvMixColumns in the inverse cipher algorithm are implemented as separate blocks. Ciphertext enters the decrypt engine from the ct interface and loops through 11/13/15 rounds for 128/192/256 bit key respectively. During each round the decryption engine consumes one round key from the key schedule buffer. Intermediate result of each round is stored in a 128-bit state register. Plaintext is presented at the end of the last round at the pt interface.

The key expander implements the key expansion algorithm in figure 11 of the FIPS-197 specification. It expands the key text into a key schedule which is then exported to the key schedule buffer. The key text loops through the key expander for a pre-defined number of rounds. Each round involves a number of transformations such as SubWord, RotWord, and XOR with round constant Rcon. While most of the processing in the key expansion algorithm is common to all key lengths, there are still minor differences for each key length. In order to achieve the highest Fmax, the decision here is to implement a separate key expander for each supported key length. This eliminates the need for extra multiplexors which increases the critical path delay. The whole key schedule is generated in 11/13/15 clock cycles for 128/192/256 bit keys.

The key schedule buffer sits between the key expander and the decrypt engine. It is a 16-deep by 128-wide dual port RAM with associated read and write pointers and handshake logic to interface with the key expander and the decryption engine. The key schedule buffer is needed because the inverse cipher algorithm consumes round keys in reversed order than they are generated by the key expansion algorithm. As a result, the whole key schedule has to be stored in a buffer as it exits from the key expander to allow the decrypt engine to access in reversed order.

I/O PORTS

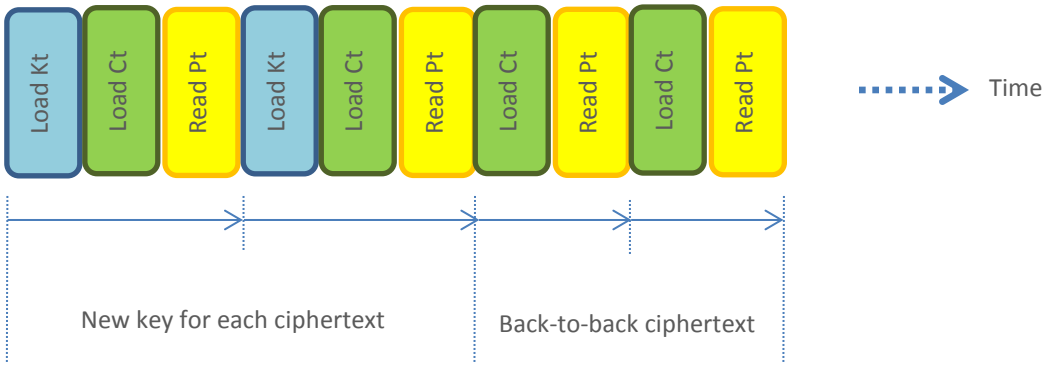
Ports	Width	Direction	Description
clk	1	Input	Core clock. All logic is synchronous to the rising edge of clk.
rst	1	Input	Core reset. Active high **synchronous** reset. This signal must be asserted for at least one clock cycle to reset the core.
kt[0:n]	128/ 192/ 256	Input	Key input. Width equals to the selected key length. A key must be loaded to the core first before a decryption can start. Once loaded, the same key can be used on multiple ciphertext.
kt_vld	1	Input	Key valid. Active high. This signal is driven high by the application to tell the core that a valid key is present on kt[0:n]. Key transfer occurs at the clock rising edge when both kt_vld and kt_rdy are high.
kt_rdy	1	output	Kt interface ready. Active high. This signal is driven high by the core when it is ready to accept a new key.
ct[0:127]	128	Input	Ciphertext input.
ct_vld	1	Input	Ciphertext valid. Active high. This signal is driven high by the application to indicate the presence of a valid ciphertext on ct[0:127]. The ciphertext is transferred to the core at the clock rising edge when both ct_vld and ct_rdy are high.
ct_rdy	1	output	Ct interface ready. Active high. This signal is driven high by the core to indicate that it is ready to accept a new ciphertext.
pt[0:127]	128	output	Plaintext output
pt_vld	1	output	Plaintext valid. Active high. This signal is driven high by the core when it has placed a valid plaintext on pt[0:127].

OPERATION

The basic decryption cycle involves 3 steps

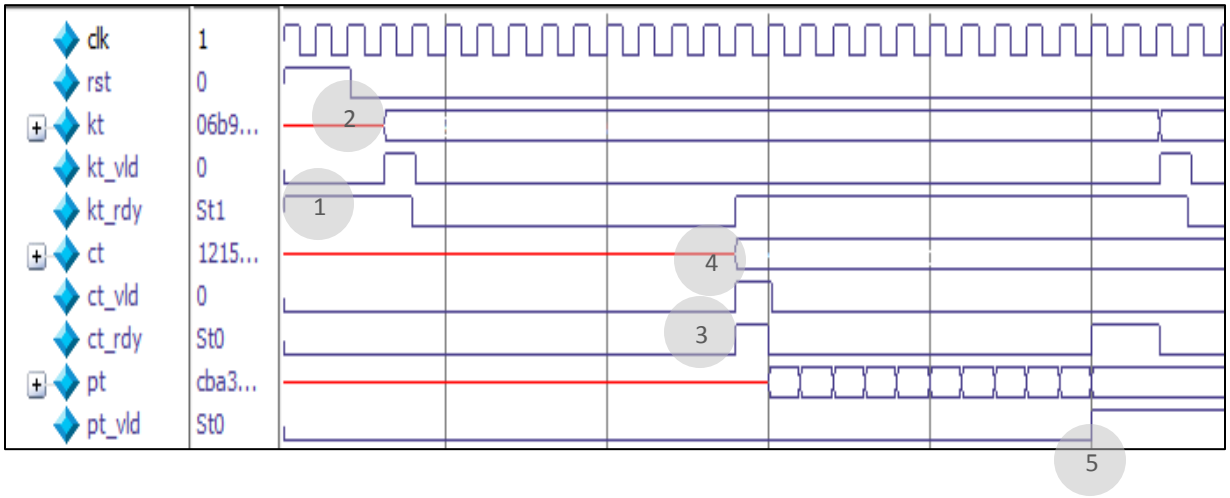
1. Load crypto key
2. Load ciphertext
3. Read plaintext

Once the plaintext is available at the pt interface, the next decryption cycle can starts by loading either the next key or ciphertext. In case the next ciphertext uses the same key as before, there is no need to load the key again since the previous key schedule is already stored in the key schedule buffer.



If a new ciphertext is loaded before the previous plaintext is read, the core will start to decrypt the new ciphetext immediately and the previous plaintext will be over-written. Loading a new key will only start a key expansion cycle internally and will not alter the previous plaintext.

BASIC 128-BIT DECRYPTION CYCLE



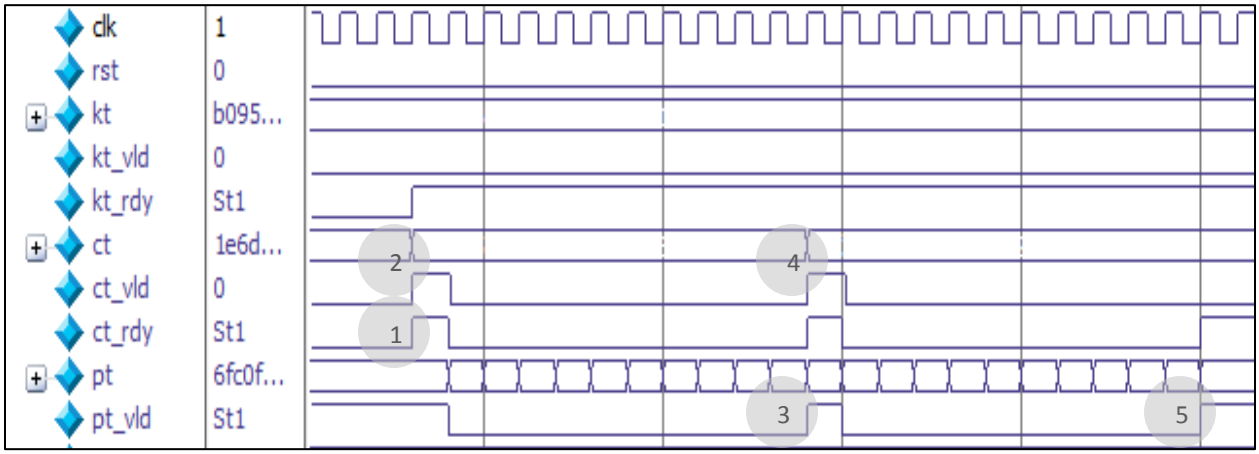
The timing diagram of a basic 128-bit decryption cycle is shown above.

1. The core asserts kt_rdy to high when it is ready to accept a new key.
2. The application presents the key to kt and asserts kt_vld to high to inform the core that a valid key is present.
3. The core asserts ct_rdy to high when it is ready to accept new ciphertext.
4. The application presents the ciphertext to ct and asserts ct_vld to high to inform the core that a valid ciphertext is present.
5. The core presents the plaintext to pt and asserts pt_vld to high when the decryption process is finished.

The key expansion starts when kt_vld is high and finishes when kt_rdy goes high again. This process takes 11 clock cycles for 128-bit key. The decryption engines starts when a valid key schedule is present and both ct_vld and ct_rdy are high and finishes when pt_vld is high. This process takes 11 clock cycles for 128-bit key.

Decryption cycle for 192 and 256-bit key are similar except the latencies are different. Refer to the benchmark section for exact latency numbers.

DECRYPTION CYCLE FOR BACK-TO-BACK CIPHERTEXT



The timing diagram above shows the decryption cycle for back-to-back ciphertext.

1. The core asserts ct_rdy to high when it is ready to accept new ciphertext.
2. The application drives the ciphertext to ct and asserts ct_vld to high to inform the core that a valid ciphertext is present.
3. The core asserts pt_vld to high when a valid plaintext is available on pt. At the same time it also asserts ct_rdy to high again to indicate it is ready to accept a new ciphertext.
4. The application drives the next ciphertext to ct and asserts ct_vld to high to inform the core that a new ciphertext is present.
5. The core asserts pt_vld to high when the second plaintext is available on pt.

It can be seen that the availability of the plaintext (pt_vld at 3) and the loading of next ciphertext (ct_vld at 4) can be overlapped. No dead cycle is incurred.

SIMULATION

The core is verified against selected test vectors from FIPS-197, AESAVS, and SP-800a. It is also tested against an AES behavioral model with random test vectors. All the necessary files for simulation are provided under the bench/ and sim/ directory so that the verification result can be reproduced.

TESTBENCH

The testbenches are located under the bench/ directory. There is a separate testbench for each supported key length, while the test set is common to all key lengths. The tests performed are listed below.

1. FIPS-197 sample vector test
2. Back-to-back ciphertext test
3. ECB-AES128/192/256.Decrypt sample vector test. SP800-38a appendix F
4. GFSbox Known Answer Test. AESAVS appendix B
5. KeySbox Known Answer Test. AESAVS appendix C
6. VarTxt Known Answer Test. AESAVS appendix D
7. VarKey Known Answer Test. AESAVS appendix E
8. Random vector test

For back-to-back ciphertext test and random vector test, the core is driven with random vectors and the output verified against the AES SystemVerilog Behavioral Model available from Opencores.org, by the same designer of this core. Source code of the behavioral model can be found under the sim/rtl_sim/src/ directory. Users are encouraged to checkout the latest version of the model from Opencores.org.

The testbench compares the core output against either known good results or golden model output. In case of a mismatch, it prints an error message and the simulation continues. Once all tests are finished either “OK” or “Failed” will be printed to indicate whether all tests are passed.

RUNNING SIMULATION WITH MODELSIM

Shell scripts for simulation and Modelsim .do files are provided under the sim/rtl_sim/bin/ directory. Simulation can be run either directly from the shell or from the Modelsim GUI. As the simulation runs, messages are dumped to the screen and at the same time to a log file in the sim/rtl_sim/out/ directory.

Make sure the path to the Modelsim executable (vsim in this case) is included in your PATH environment before you execute the shell script. Also, you may need to add execute permission to the shell script to be able run from UNIX shell.

Here is an example of simulating the 128-bit core from the Windows command prompt

1. Change directory to sim/rtl_sim/bin
2. Type “sim128.bat” from the command prompt
3. Examine sim/rtl_sim/out/sim128.out for simulation results

To simulate the 128-bit core from Modelsim GUI

1. Launch Modelsim
2. Change directory to sim/rtl_sim/bin from Modelsim prompt
3. Type “do sim128.do” from Modelsim prompt

To include a waveform view in Modelsim GUI, uncomment the line “add wave *” from the file sim128.do.

To run simulation for other key lengths, replace **sim128.*** above with **sim192/256.*.**

RETARGETING GUIDELINES

The core is designed with the objective of maximizing performance and resource utilization when implemented on modern LUT6 based FPGA. This is realized by carefully written source codes which limit combinational logic to use at most 6 input signals whenever possible so that they can fit well into LUT6s. Other than that, the source code is technology independent and portable to FPGA architecture of different vendors. This section describes the recommended modifications to the core for retargeting to bring out the full performance of the target technology.

Inclusion of generic_muxfx.v

The source file “generic_muxfx.v” located under rtl/verilog/generic/ directory defines technology independent 2-to-1 multiplexors MUXF7 and MUXF8 which are used in the source file “Sbox.sv”. This file should NOT be included while targeting Xilinx FPGA to allow the synthesis tool to use the MUXF7 and MUXF8 in the Xilinx library. When targeting other FPGA technologies, either provides a technology specific definition of those multiplexors, or include “generic/generic_muxfx.v” if a technology specific version is not available.

Tool specific synthesis attributes

Xilinx Vivado synthesis attributes are used throughout the source codes to hint the inference of specific logic resources and to preserve the design hierarchy for better packing into FPGA slices. Those attributes are essential to achieve maximum performance on the target FPGA.

The table below shows the Vivado synthesis attributes that need to be replaced when retargeting to other FPGA technologies or using a different synthesis tool.

Vivado synthesis attribute	Used in	Description
(* RAM_STYLE="distributed" *)	KschBuffer.sv	Infer LUT RAM for the key schedule buffer.
(* KEEP_HIRARACHY = "yes" *)	decrypt128_wrapper.sv decrypt192_wrapper.sv decrypt256_wrapper.sv decrypt.sv InvSbox.sv Sbox.sv	Maintain the design hierarchy as specified in the source code.
(* keep = "true", max_fanout = 1 *)	KeyExpand128.sv KeyExpand192.sv KeyExpand256.sv	This combination hints the synthesizer not to optimize away the target signal and to limit the fanout of that signal to one. The objective is to force the synthesizer to infer separate logic to drive every signal which this attribute combination is applied. Refer to the comments in the corresponding source codes for details.