



Flexible AES Crypto Cores (FLEX Series)

December 2018

Product Specification V1.0

Features

- ❖ Fully compliant with NIST FIPS-197 standard
- ❖ Flex series IP cores throughput summary
 - Over 7 Gbps for combo core (encrypt & decrypt)
 - Over 9 Gbps for stand-alone encrypt or decrypt
- ❖ Flexible architecture supports area/speed tradeoffs
 - User selectable number of AES engines allows an optimal balance of LUTs and throughput to be achieved
- ❖ Supports key size of 256 bits with hardware-based key expansion
- ❖ Supports stand-alone encryption, decryption or both using 128-bit data blocks in ECB format
- ❖ Powerful testbench verifies FIPS compliance through known answer test (KAT)
- ❖ Hardware verified using FPGA development kit

IP Core Facts

Provided with Core

Documentation	Core datasheet and testbench description
Design File Format	VHDL RTL or Verilog netlist
Constraint Files	SDC and PDC constraints
Verification	Testbench using Modelsim from Mentor

Synthesis Tool Used

Synplify Version L-2016.09M-2

Support

Provided by local sales channel

Table 1 - PolarFire Implementation Statistics

Core	AES Engines	LSRAM/ (uRAM)	LUT4	FF	Max Freq. (MHz) ¹	Cycles/AES Operation ²	Throughput (Gbps) ³
AES256_flex_combo (encrypt and decrypt)	1	16/24	3,533	3,374	141.6	14	1.30
	2	24/48	5,367	2,981	126.4	7	2.31
	3	32	8,308	6,318	129.7	5	3.32
	4	40	10,137	6,613	119.2	4	3.81
	5	48	11,580	6,971	118.2	3	5.04
	7	64	14,606	7,522	115.7	2	7.40
AES256_flex_enc (encrypt only)	1	12/12	2,020	1,483	174.0	14	1.59
	2	20/24	3,016	1,946	166.5	7	3.05
	3	28	3,826	3,695	162.0	5	4.15
	4	36	4,756	3,987	157.7	4	5.05
	5	44	5,707	4,309	159.0	3	6.79
	7	60	7,600	4,892	151.5	2	9.70
AES256_flex_dec (decrypt only)	1	12/12	1,975	1,491	161.2	14	1.47
	2	20/24	3,180	1,954	153.4	7	2.81
	3	28	5,586	3,831	153.4	5	3.93
	4	36	6,748	4,123	145.3	4	4.65
	5	44	7,500	4,445	135.4	3	5.78
	7	60	8,875	5,028	141.7	2	9.07

Notes:

- 1) Performance based on MPF300-1FCG1152 with single pass, STD effort timing-driven place and route
- 2) This value indicates the number of clock cycles required before the next input block can be processed
- 3) Throughput is calculated based on a 128-bit AES block processed = (128 x Max Freq.) / cycles per AES operation

Table 2 - Igloo2/SmartFusion2 Implementation Statistics

Core	AES Engines	LSRAM/ (uRAM)	LUT4	FF	Max Freq. (MHz) ¹	Cycles/AES Operation ²	Throughput (Gbps) ³
AES256_flex_combo (encrypt and decrypt)	1	16/24	TBD	TBD	TBD	14	TBD
	2	24/48	TBD	TBD	TBD	7	TBD
	3	32	TBD	TBD	TBD	5	TBD
	4	40	TBD	TBD	TBD	4	TBD
	5	48	TBD	TBD	TBD	3	TBD
	7	64	TBD	TBD	TBD	2	TBD
AES256_flex_enc (encrypt only)	1	12/12	TBD	TBD	TBD	14	TBD
	2	20/24	TBD	TBD	TBD	7	TBD
	3	28	TBD	TBD	TBD	5	TBD
	4	36	TBD	TBD	TBD	4	TBD
	5	44	TBD	TBD	TBD	3	TBD
	7	60	TBD	TBD	TBD	2	TBD
AES256_flex_dec (decrypt only)	1	12/12	TBD	TBD	TBD	14	TBD
	2	20/24	TBD	TBD	TBD	7	TBD
	3	28	TBD	TBD	TBD	5	TBD
	4	36	TBD	TBD	TBD	4	TBD
	5	44	TBD	TBD	TBD	3	TBD
	7	60	TBD	TBD	TBD	2	TBD

Notes:

- 1) Performance based on M2GL150T-1FC1152 with single pass, high effort timing-driven place and route
- 2) This value indicates the number of clock cycles required before the next input block can be processed
- 3) Throughput is calculated based on a 128-bit AES block processed = (128 x Max Freq.) / cycles per AES operation

AES Algorithm Overview

The Advanced Encryption Standard (AES) specifies a Federal Information Processing Standards (FIPS-197) approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt and decrypt information. Encryption converts plain-text data to an unintelligible form called cipher-text. Decrypting the cipher-text converts the data back into its original plain-text form.

The AES algorithm uses cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. For the FLEX series of IP cores however, only the 256-bit key size is supported and will be denoted from this point forward as “AES256”.

The AES algorithm requires an expansion of the 256-bit key material to then provide a unique 128-bit key for each of the 14 rounds of cryptography as specified by the FIPS-197 standard. This step to create these additional keys is called key expansion. For the FLEX series, the key expansion step is required each time a new key is used. Once a new key has been expanded, the 128-bit plain-text or cipher-text data can be input to the core continually.

Flex Series Combo Core

Figure 1 below illustrates the architecture of the AES256_flex_combo (combination) core which performs both encryption and decryption in a single block of RTL. As Figure 1 shows, the core has independent key expansion for both encryption and decryption which allows independent keys to be used for the encryption and decryption operations. The scalable architecture allows from one to seven AES engines to be allocated to the crypto process. Also, the dual datapath architecture supports cycle by cycle encryption or decryption operations without any switching delays. In other words, the core can be used in a time-division-multiplexing (TDM) fashion, where on successive cycles, the user can perform encrypt -> decrypt -> encrypt -> decrypt etc. In this use model, the

throughput for each encrypt and decrypt would be half of the values shown in Table 1 and Table 2, however the user would need only half the memory compared to using two stand-alone cores. For this reason, the combo core can also be viewed as a memory optimized core.

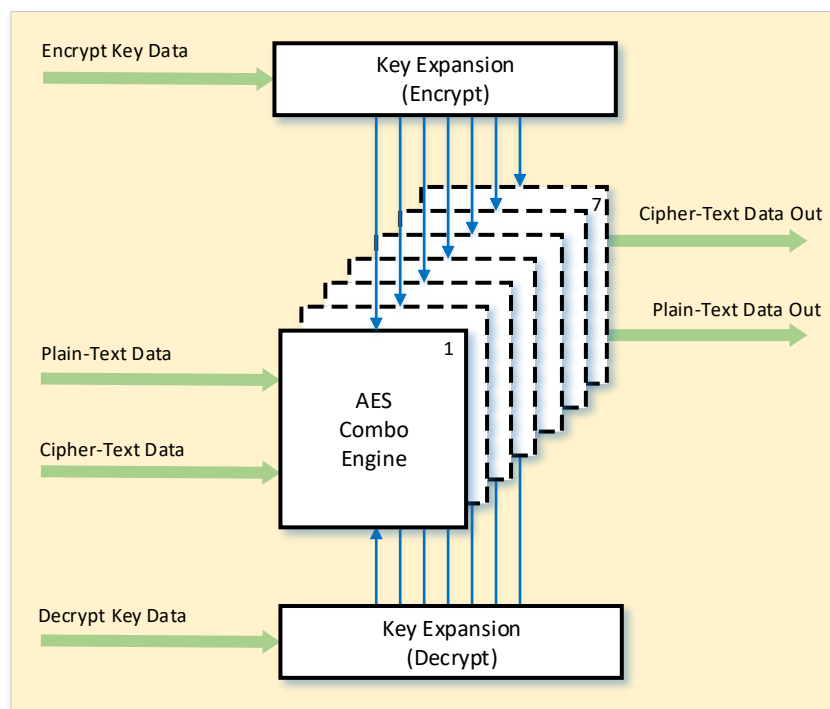


Figure 1 – AES256_flex_combo Block Diagram

Figure 2 below shows the architecture of the AES combo engine that is replicated in Figure 1. With independent 128-bit data paths for both plain-text and cipher-text, the core can be used as essentially two mostly independent encrypt and decrypt functions. The sharing of the SBOX memory allows both encrypt and decrypt to be performed by a single IP core without doubling the memory usage.

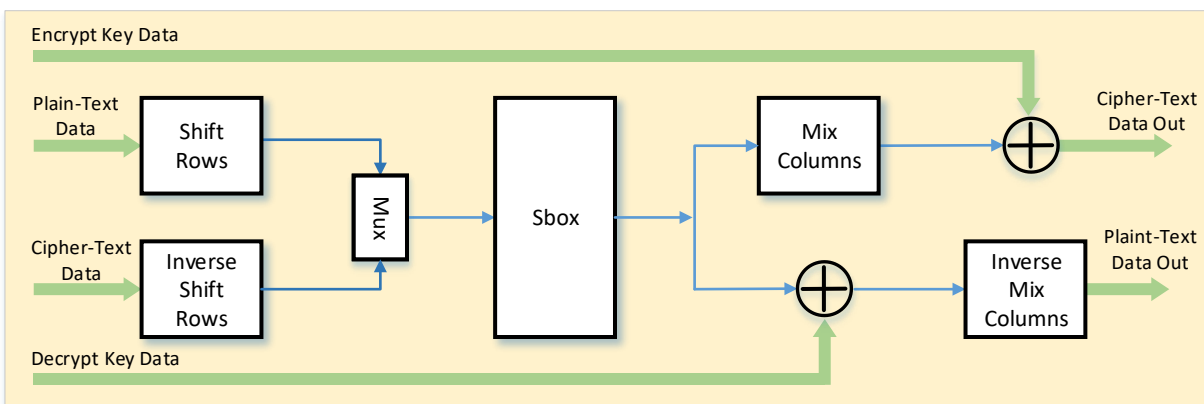


Figure 2 – AES Combo Engine Block Diagram

Flex Series Stand-alone Cores

If a separate encrypt or decrypt function is required or simultaneous (full duplex), full speed streaming is desired, then the AES256_flex_enc (encrypt) and AES256_flex_dec (decrypt) stand-alone cores are available to meet these requirements. Figure 3 below shows a block diagram of the encrypt/decrypt core. These stand-alone crypto cores have also been designed to allow the user to select the number of AES engines to allocate for the desired crypto function.

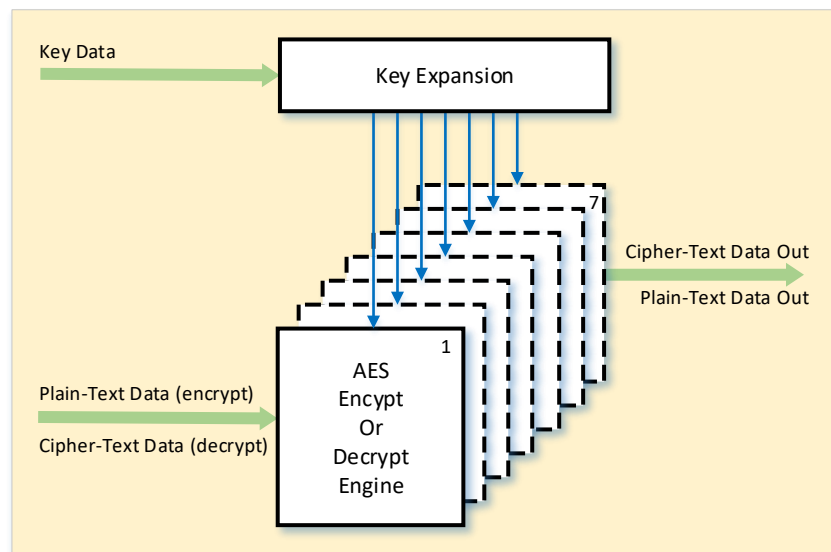


Figure 3 – AES256_flex_enc/dec Block Diagram

Figure 4 shows an example of how the stand-alone cores can be used in applications where full-speed independent encrypt and decrypt are required.

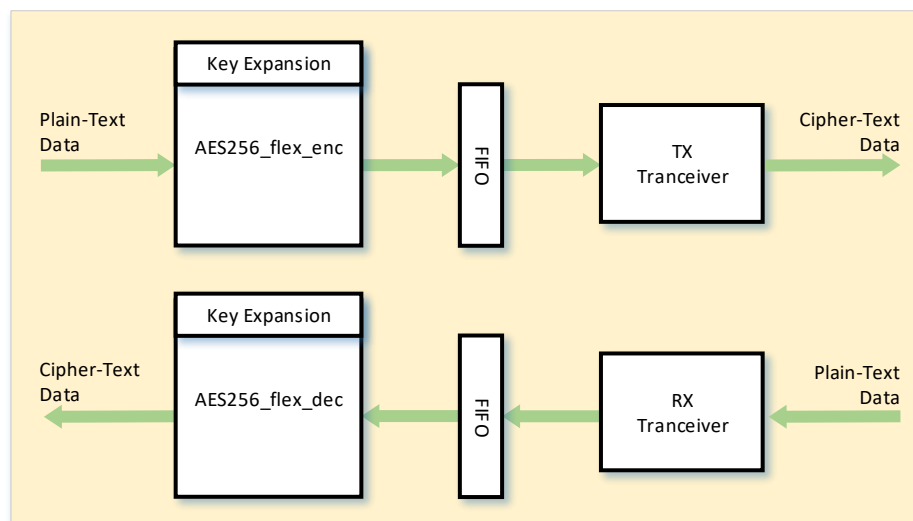


Figure 4 – Example of a Full Duplex Crypto Application

Generic Definitions

Table 3 shows the generic settings that need to be configured for the desired user operation of the AES core. Using generics provides maximum flexibility to the user by allowing a tradeoff of area versus performance to be made.

Table 3 - Generics for AES256_Flex (enc, dec, or combo)

Generic	Type	Values	Description
NUM_ENGINES	Integer	1, 2, 3, 4, 5, 7	Specifies the number of AES engine cores to be applied to the crypto function
KEY_PIPE_STAGES	Integer	1 or 2	Specifies how many pipe stages that will allocated to the key expansion function 1 = One pipe stage indicates the key expansion will take 14+1 cycles to complete 2 = Two pipe stages indicate the key expansion will take 28+1 cycles to complete

For applications where the critical timing path is in the key expansion engine, the user can select a 2nd pipe stage option which increases the performance of the expansion while only adding 14 cycles to the processing time.

Signal Description

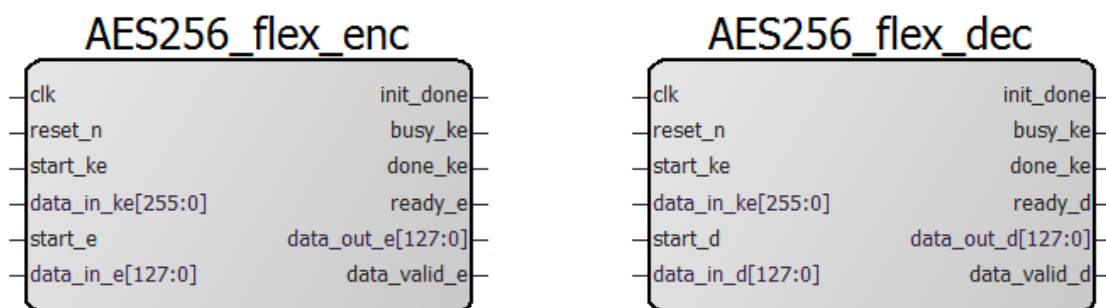


Figure 5 – AES256_flex Encrypt and Decrypt I/O Diagram

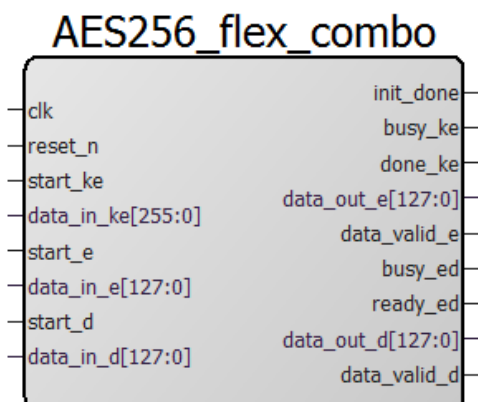


Figure 6 – AES256_flex_combo I/O Diagram

Table 4 - I/O Signal Description

Signal	Direction	Description
CLK	Input	Input clock to all registers and RAM
RESET_N	Input	LO active asynchronous clear of all registers
START_KE	Input	HI active signal used to start the key expansion process. Only needs to be asserted for one clock cycle to start key expansion.
DATA_IN_KE[255:0]	Input	256-bit input data for key expansion
START_E	Input	HI active signal that starts an encryption operation
DATA_IN_E[127:0]	Input	128-bit plain-text input for the encrypt function
START_D	Input	HI active signal that starts a decryption operation
DATA_IN_D[127:0]	Input	128-bit cipher-text input for the decrypt function
INIT_DONE	Output	HI active level signal indicating that the initialization of the SBOX memories has been completed. The SBOX initialization is done automatically after the de-assertion of the RESET_N signal
BUSY_KE	Output	HI active signal indicating that the key expansion process is being performed
DONE_KE	Output	HI active pulse signal indicating that the key expansion process has been completed
READY_E	Output	HI active signal indicating an encryption operation can be started
READY_D	Output	HI active signal indicating a decryption operation can be started
READY_ED	Output	HI active signal indicating an encryption or decryption operation can be started
DATA_OUT_E[127:0]	Output	128-bit cipher-text output

DATA_VALID_E	Output	HI active pulse signal indicating that the 128-bit cipher-text data is valid
DATA_OUT_D[127:0]	Output	128-bit plain-text output
DATA_VALID_D	Output	HI active pulse signal indicating that the 128-bit plain-text data is valid
BUSY_ED	Output	HI active signal indicating that the combo core is busy processing crypto data

Functional Description

Initialization

This IP core requires an initialization of the internal sbox ram tables before cryptographic functions can be started. This process is started immediately after the de-assertion of the RESET_N signal. For the combo core, after 512 plus one clock cycles, the INIT_DONE signal is asserted HI to indicate completion of the initialization process. For the stand-alone cores, after 256 plus one clock cycles, the INIT_DONE signal is asserted HI to indicate completion of the initialization process. The INIT_DONE signal remains HI until the next assertion of RESET_N. Figure 3 below illustrates the timing for sbox initialization from internal ROM. No other actions are required to start the init process other than the de-assertion of the RESET_N signal.

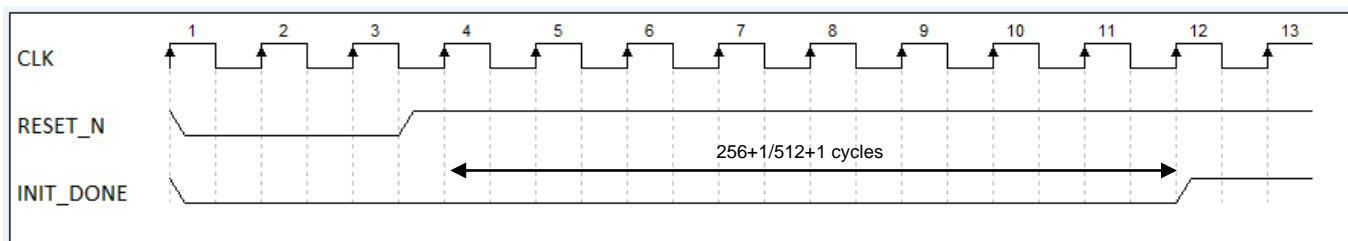


Figure 7 – SBOX Initialization Timing

Key Expansion

The key expansion step is required each time a new key is to be used in the cryptographic process. Before a cryptographic process is performed, the AES algorithm requires the 256-bit key to be expanded. During the key expansion step, the key data is input to the core where it gets passed through a logic chain 14 times to produce 14 sub-keys. These sub-keys are stored in either ram or registers.

The process of key expansion takes 14 or 28 clock cycles based on the generic setting plus one cycle for the output register. Key expansion and cryptographic functions can not be overlapped. The timing diagram shown below in Figure 8 shows that START_KE initiates the key expansion and KE_DONE indicates its completion.

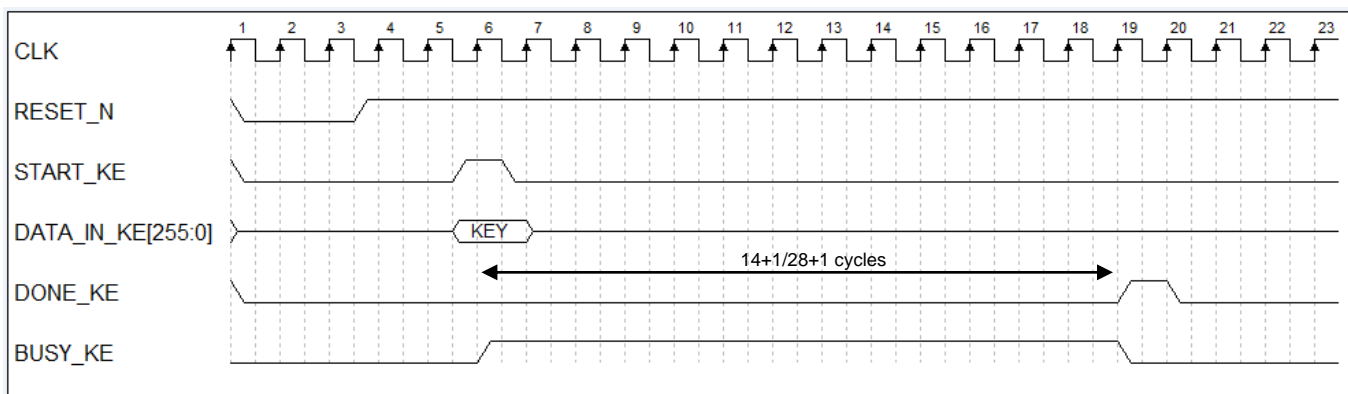


Figure 8 – Key Expansion Timing

AES Engine

This AES engine block is responsible for the actual encrypt and decrypt functions. Based on the number of engines allocated to the crypto function, the throughput can be increased at the expense of memory and gates. Table 5 below restates the options that are available with the combo or stand-alone cores. There are two considerations for assessing performance of these IP cores – latency and throughput. Since the cores are heavily pipelined, latency is defined as the number of clock cycles that it takes to get the first crypto result out of the core. It is the same number regardless of how many AES engines are used. The latency for all IP cores is 14 cycles (one for each round of AES256) plus one cycle for a pipeline register. Throughput is based on the number of cycles that it takes for each AES256 block (128-bits) to be processed. For example, if seven AES engines are used, the first crypto result would be output after 15 cycles followed by a new crypto result every 2 cycles after that (assuming the core is given new data to process every 2 cycles).

Table 5 – Performance vs Area Tradeoffs

# of AES Engines	# of Cycles per AES Operation	Latency
1	14	14+1
2	7	14+1
3	5	14+1
4	4	14+1
5	3	14+1
7	2	14+1

Figure 9 shows the timing required to perform an encryption and decryption operation with the combo core. The START_E signal at CLK5 begins the encrypt function on P1. 15 cycles later the cipher-text of P1 (called E1) is output synchronous with the DATA_VALID_E signal (the waveform below is compressed in showing the 15 cycles required for encryption/decryption). Four cycles after the last encrypt operation starts on CLK13, the first decrypt operation starts on CLK17 with cipher-text C1.

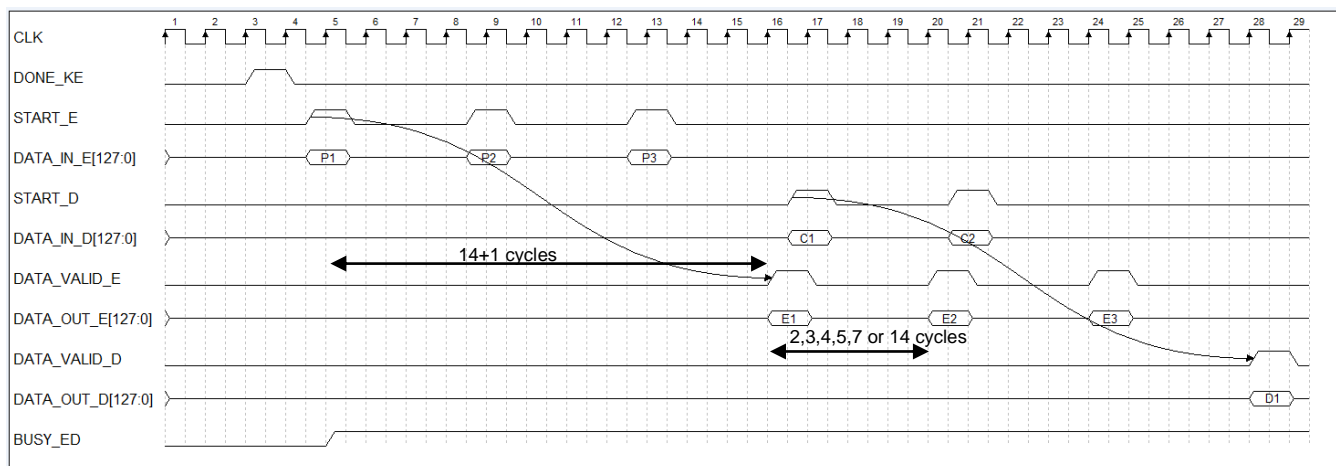


Figure 9 – Combo Core Crypto Timing (num_engines=4)

Figure 10 shows the timing required to perform an encryption or decryption operation with one of the stand-alone cores. The START_x signal at CLK5 begins the encrypt or decrypt function on P1. 15 cycles later the crypto text of P1 (called E1) is output synchronous with the DATA_VALID_x signal (the waveform below is compressed in showing the 15 cycles required for encryption/decryption).

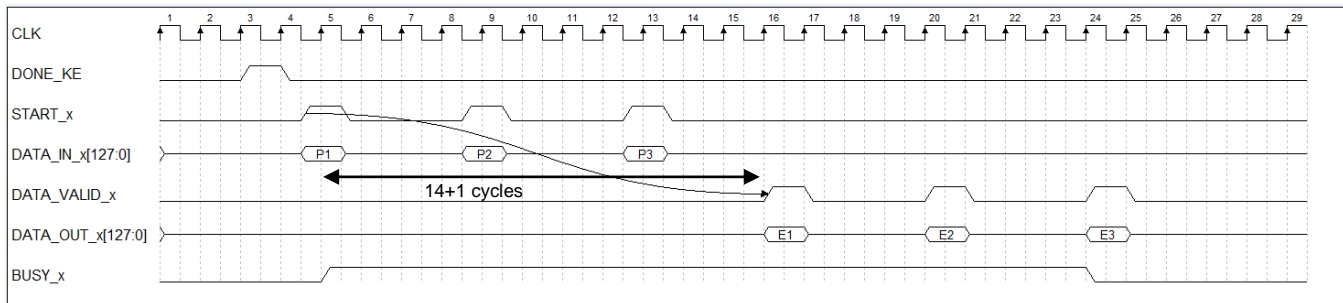


Figure 10 – Stand-alone Core Crypto Timing (num_engines=4)

Verification

Figures 11 and 12 shows the testbench architectures used to validate the Flex series AES256 cores.

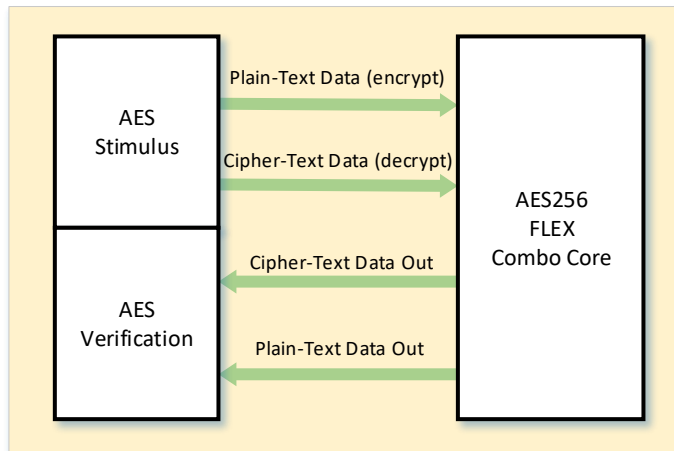


Figure 11 – Combo Core Verification

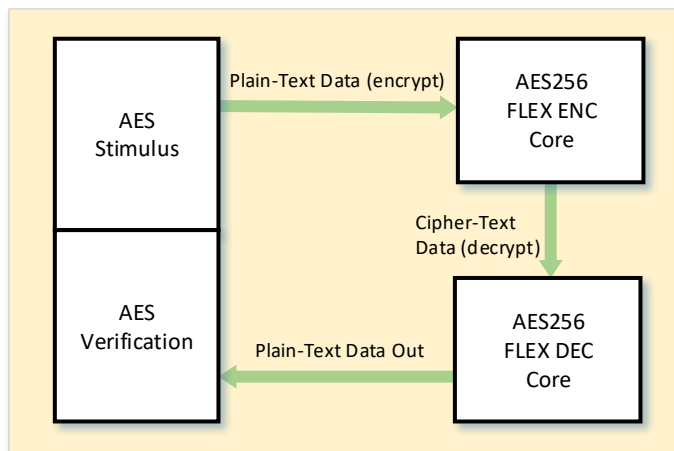


Figure 12 – Stand-alone Core Verification

Deliverables

Table 5 – AES256_flex_enc Source Code Hierarchy











 AES256_flex_enc (AES256_flex_enc.vhd) [work]
 AES256_engine_e_mc (AES256_engine_e_mc.vhd) [work]
 input_mux (input_mux.vhd) [work]
 mixcol (mixcol.vhd) [work]
 sbox_top_LSRAM_enc (sbox_top_LSRAM_enc.vhd) [work]
 KEY256_engine_e_mc (KEY256_engine_e_mc.vhd) [work]
 KEY256_regfile_e_mc (KEY256_regfile_e_mc.vhd) [work]
 KEY256_sbox_ed_mc (KEY256_sbox_ed_mc.vhd) [work]
 flex_sbox_init (flex_sbox_init.vhd) [work]
 ROM256x8_array (ROM256x8_array.vhd) [work]

Table 6 – AES256_flex_dec Source Code Hierarchy











 AES256_flex_dec (AES256_flex_dec.vhd) [work]
 AES256_engine_d_mc (AES256_engine_d_mc.vhd) [work]
 input_mux (input_mux.vhd) [work]
 inv_mixcol (inv_mixcol_net.vhd) [work]
 sbox_top_LSRAM_dec (sbox_top_LSRAM_dec.vhd) [work]
 KEY256_engine_d_mc (KEY256_engine_d_mc.vhd) [work]
 KEY256_regfile_d_mc (KEY256_regfile_d_mc.vhd) [work]
 KEY256_sbox_ed_mc (KEY256_sbox_ed_mc.vhd) [work]
 flex_sbox_init (flex_sbox_init.vhd) [work]
 ROM256x8_array (ROM256x8_array.vhd) [work]

Table 7 – AES256_flex_combo Source Code Hierarchy
















▲		AES256_flex_combo (AES256_flex_combo.vhd) [work]
▲		AES256_engine_ed_mc (AES256_engine_ed_mc.vhd) [work]
		input_mux (input_mux.vhd) [work]
		inv_mixcol (inv_mixcol_net.vhd) [work]
		mixcol (mixcol.vhd) [work]
		mux2to1_128 (mux2to1_128.vhd) [work]
		sbox_top_LSRAM_ed (sbox_top_LSRAM_ed.vhd) [work]
▲		KEY256_engine_d_mc (KEY256_engine_d_mc.vhd) [work]
		KEY256_regfile_d_mc (KEY256_regfile_d_mc.vhd) [work]
		KEY256_sbox_ed_mc (KEY256_sbox_ed_mc.vhd) [work]
▲		KEY256_engine_e_mc (KEY256_engine_e_mc.vhd) [work]
		KEY256_regfile_e_mc (KEY256_regfile_e_mc.vhd) [work]
		KEY256_sbox_ed_mc (KEY256_sbox_ed_mc.vhd) [work]
▲		ed_sbox_init (ed_sbox_init.vhd) [work]
		ROM256x8_array (ROM256x8_array.vhd) [work]

Table 8 –Simulation Files

Item		Description
Run_aes.do		Combo core top level do file for running AES simulation
	AES256flex_stim2.vhd	Testbench for reading and comparing multiple cipher/decipher functions – Pass/Fail notification
	wave_aes.do	Waveform file for examining signals in Modelsim wave window
Run_aes.do		Stand-alone core top level do file for running AES simulation
	AESmodem_stim2.vhd	Testbench for reading and comparing multiple cipher/decipher functions – Pass/Fail notification
	wave_aes.do	Waveform file for examining signals in Modelsim wave window