Advanced Encryption Standard (AES) Tiny Encryptor (8-bit Datapath)

December 23rd, 2012 Product Specification V1.1 for G4

Features

- Fully compliant with NIST (FIPS-197) standards
- Supports a key size of 128 or 256 bits with hardware-based key expansion
- Supports AES encryption, using 128-bit data blocks in ECB or CBC format
  - Optional support for CFB and OFB formats
- Supports RAM or ROM-based sbox tables
  - Supports RAM initialization by either ROM or external resource
- 8-bit data interface simplifies loading of keys and data
- Supports burst operations with optional full duplex fifo
- Optimized for smallest possible area
- Testbench verifies FIPS compliance through known answer test (KAT)

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 D2009.12A	
Support	
Provided by local sales channel	

Table 1 – VTinyAESe_8 Implementation Statistics for SmartFusion 2/Igloo 2

MODE	Key Size (bits)	SBOX[1]		FIFO	uRAM Blocks	Logic Cells		Speed[2] (MHz)	Throughput (Mbps)
		Implemented As	Initialized by			Lut4	Seq		
ECB	128	ROM	n/a	No	3	525	210	145	20
		RAM	ROM	No	5	509	268	213	29
			External	No	5	254	200	214	29
		ROM	n/a	Yes	5	620	275	150	20
		RAM	ROM	Yes	7	611	332	202	27
			External	Yes	7	334	259	207	28
	256	ROM	n/a	No	3	525	210	145	14
		RAM	ROM	No	5	515	272	226	22
			External	No	5	254	200	214	21
		ROM	n/a	Yes	5	620	275	150	15
		RAM	ROM	Yes	7	611	332	202	20
			External	Yes	7	334	259	207	20
CBC	128	ROM	n/a	No	3	525	353	142	19
		RAM	ROM	No	5	562	398	194	26
			External	No	5	266	345	236	32
		ROM	n/a	Yes	5	600	419	153	21
		RAM	ROM	Yes	7	615	480	183	25
			External	Yes	7	348	402	191	26
	256	ROM	n/a	No	3	538	355	150	15
		RAM	ROM	No	5	562	401	217	21
			External	No	5	289	341	218	21
		ROM	n/a	Yes	5	600	419	154	15
		RAM	ROM	Yes	7	615	480	193	19
			External	Yes	7	348	402	195	19

1) This column refers to the how the internal SBOX tables are implemented and whether an initialization process applies or not
2) All performance numbers are based on M2S050T-1FG896 with single pass TDPR

AES Algorithm Overview

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

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. The algorithm is used with the three different key lengths indicated above, and therefore these different "flavors" are referred to as "AES-128", "AES-192", and "AES-256". For the AES algorithm, the amount of processing or number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by ***Nr***, where

Nr =10 when for AES-128, ***Nr*** = 12 for AES-192, and ***Nr*** = 14 for AES-256.

Table 2 illustrates the breakdown of processing steps based on the different key sizes. The overall throughput is therefore decreased as the key size is increased. In other words, there are four additional rounds of processing to handle a 256-bit key. Table 1 shows how the key size affects the core throughput.

Table 2 - AES Algorithm

Version	Key Size	Block Size	Rounds (Nr)
AES-128	128 bits	128 bits	10
AES-256	256 bits	128 bits	14

This IP core implementation of the AES algorithm (VTinyAESe_8) supports a 128-bit or 256-bit key size and encryption operations only.

The AES algorithm requires an expansion of the input key to provide a unique key for each round of the encryption process. The step to create these additional keys is called key expansion. For the VTinyAESe_8 core, the key expansion step is required each time a new key is used. Once a new key has been expanded, then 128-bit data blocks (16 bytes) can be input to the core for encryption. To allow up 32 continous blocks to be encrypted, an optional full duplex fifo can be used on the front end of the VTinyAESe_8 core. Figure 1 below illustrates the architecture of the VTinyAESe_8 core. Microsemi's flash-based FPGAs are a perfect fit for AES data security applications due to their inherent device security and non-volatile attributes.

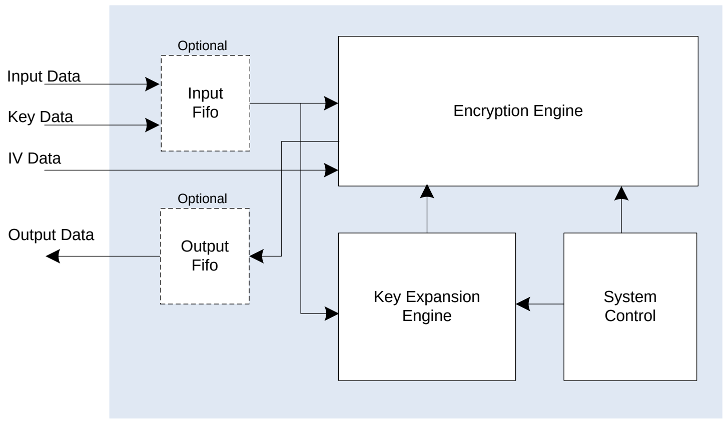


Figure 1 – VTinyAESe_8 Block Diagram

Generic Definitions

Table 2 shows the generic settings that need to be configured for the desired user operation of the AES core. Using generics, maximum flexibility is obtained allowing a balance of area versus feature tradeoffs to be made.

Table 2 - Generics for VTinyAESe_8

Generic	Type	Values	Description
MODE	Integer	0 or 1	Specifies Algorithm type 0 = Electronic Code Book (ECB) mode 1= Cypher Block Chaining (CBC) mode
KEYSIZE	Integer	128 or 256	Specifies the size of the key to be used in the encryption process 128 = use 128-bit key for encryption 256 = use 256-bit key for encryption
USE_ROM_SBOX	Integer	0 or 1	Specifies whether a 256 byte ROM will be used in the core for the SBOX table or whether the ROM will initialize RAM used for the SBOX table (if USE_RAM_SBOX is 1) 0 = no ROM used 1 = no ROM used
USE_RAM_SBOX	Integer	0 or 1	Specifies whether RAM will be used in the core for the SBOX table 0 = no RAM used 1 = RAM used
USE_FRONTEND_FIFO	Integer	0 or 1	Specifies whether two 512x8 bit fifos are used in the core for data into and out of the AES core 0 = no fifo used 1 = use fifo

Notes:

Sbox Generics

Using these two generic settings, three different configurations of the sbox table implementation are supported. These configurations are shown below in Table 3.

Table 3 – SBOX Generic Description

SBOX[2]		Use_rom_sbox	Use_ram_sbox
Implemented As	Initialized by		
ROM	n/a	1	0
RAM	ROM	1	1
	External	0	1

It should be noted that it is an illegal combination to have 0 for the two generics above. The sbox must be based on rom, ram or both. If ROM only is selected, then RAM blocks can be saved at the expense of performance since the ROM block is slower that using dedicated RAM. If external RAM initialization is selected, then this will yield the smallest possible core, since it is left to the user to load the 256 byte sbox table into the RAM.

Signal Description

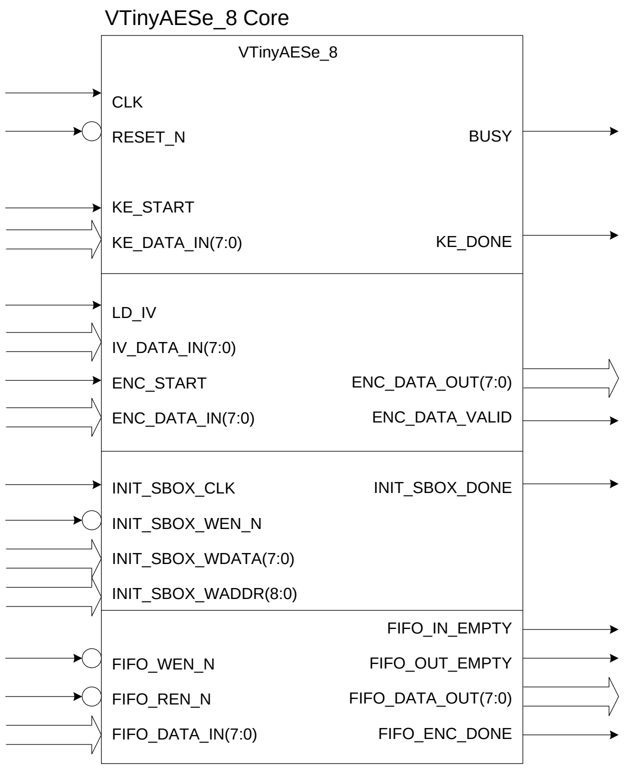


Figure 2 – VTinyAESe_8 I/O Diagram

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
BUSY	Output	HI active signal indicating the core is busy doing key expansion or an encryption operation
KE_START	Input	HI active signal used to start key expansion. Only needs to be asserted for one clock cycle.
KE_DATA_IN(7:0)	Input	8-bit input for the key to be loaded (16 or 32 bytes in sequence)
KE_DONE	Output	HI active signal indicating the completion of key expansion
LD_IV	Input	HI active signal used to load the initialization vector used for CBC encryption mode. This signal needs to be asserted for 16 clock cycles.
IV_DATA_IN(7:0)	Input	8-bit initialization vector input
ENC_START	Input	HI active signal used to start an encryption operation. Only needs to be asserted for one clock cycle.
ENC_DATA_IN(7:0)	Input	8-bit plaintext input (16 bytes in sequence)
ENC_DATA_OUT(7:0)	Output	8-bit encrypted data output (16 bytes in sequence)
ENC_DATA_VALID	Output	HI active signal indicating when the data output bus has valid encrypted data on it
INIT_SBOX_CLK	Input	Input clock to SBOX ram. Must be less than or equal to CLK frequency.
INIT_SBOX_WEN_N	Input	LO active write enable to the SBOX ram
INIT_SBOX_WDATA(7:0)	Input	8-bit write data to the SBOX ram
INIT_SBOX_WADDR(7:0)	Input	8-bit write address to the SBOX ram (256 byte addresses)
INIT_SBOX_DONE	Output	HI active signal indicating the SBOX ram table has been initialized after reset.
FIFO_WEN_N	Input	LO active signal that allows the fifo to be written to
FIFO_REN_N	Input	LO active signal that allows the fifo to be read from
FIFO_DATA_IN(7:0)	Output	8-bit input data for inbound Fifo (key and plaintext data share this input)
FIFO_DATA_OUT(7:0)	Output	8-bit encrypted data output from outbound Fifo

FIFO_IN_EMPTY	Output	HI active signal indicating that the input fifo is empty
FIFO_OUT_EMPTY	Output	HI active signal indicating that the output fifo is empty
FIFO_ENC_DONE	Output	HI active signal indicating that the core has completed processing the data blocks in the input Fifo

It is important to understand the effect of the generic settings on the user I/O of the VTinyAESe_8 core. Table 5 below illustrates when an I/O pin is used based on the generic setting.

Table 5 - Generic Settings and Effect on I/O

Signal	Mode	Use_rom_sbox	Use_ram_sbox	Use_frontend_fifo
INIT_SBOX_CLK	Has No Effect	0	1	Has No Effect
INIT_SBOX_WEN_N	Has No Effect	0	1	Has No Effect
INIT_SBOX_WDATA(7:0)	Has No Effect	0	1	Has No Effect
INIT_SBOX_WADDR(7:0)	Has No Effect	0	1	Has No Effect
INIT_SBOX_DONE	Has No Effect	0	1	Has No Effect
FIFO_WEN_N	Has No Effect	Has No Effect	Has No Effect	1
FIFO_REN_N	Has No Effect	Has No Effect	Has No Effect	1
FIFO_DATA_IN(7:0)	Has No Effect	Has No Effect	Has No Effect	1
FIFO_DATA_OUT(7:0)	Has No Effect	Has No Effect	Has No Effect	1
FIFO_IN_EMPTY	Has No Effect	Has No Effect	Has No Effect	1
FIFO_OUT_EMPTY	Has No Effect	Has No Effect	Has No Effect	1
FIFO_ENC_DONE	Has No Effect	Has No Effect	Has No Effect	1
KE_DATA_IN(7:0)	Has No Effect	Has No Effect	Has No Effect	0
LD_IV	1	Has No Effect	Has No Effect	Has No Effect
IV_DATA_IN(7:0)	1	Has No Effect	Has No Effect	Has No Effect
ENC_DATA_IN(7:0)	Has No Effect			0
ENC_DATA_OUT(7:0)	Has No Effect			0
ENC_DATA_VALID	Has No Effect			0

The I/O signals not shown above are always necessary for proper operation of the VTinyAESe_8 core. The I/Os above that have a NO effect or 0 can be left unconnected on the core given the generic setting shown. For example, the INIT signals are only used when the ROM_sbox and RAM_sbox generics have the setting shown. Also, the direct KE data and ENC data signals are not used when the FIFO is selected. Of note here in the FIFO mode is the IV data does not go through the FIFO where the Key and crypto data in and out does (see Figure 1).

Functional Description

RAM-based Sbox Initialization (USE_RAM_SBOX=1)
The VTinyAESe_8 core requires an initialization of the internal sbox ram table before cryptographic functions can be started. There are two ways that this can be done. If USE_SBOX_ROM=1, then the ram table will be automatically loaded from an internal 256 byte rom.

As shown below in Figure 3, this process is started immediately after the de-assertion of the RESET_N signal. After 256 clock cycles, the INIT_SBOX_DONE signal is asserted HI to indicate completion of the initialization process. The INIT_SBOX_DONE signal remains HI until the next assertion of RESET_N. No other actions are required to start the initialization process other than the de-assertion of the RESET_N signal

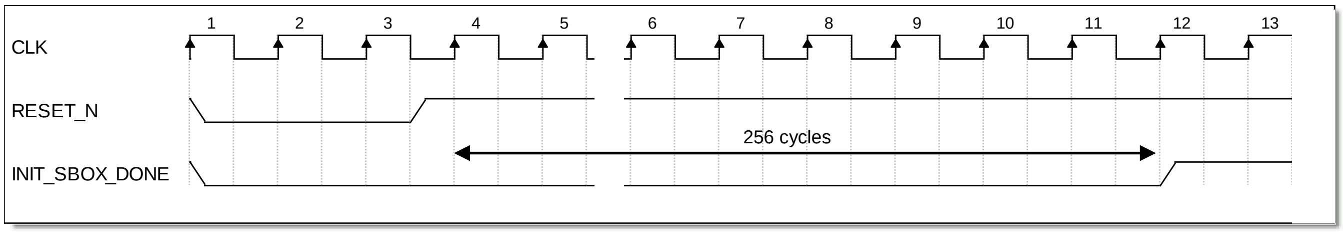


Figure 3 – ROM-based Initialization Timing (USE_ROM_SBOX=1)

In an effort to offer further area efficiency for the VTinyAESe_8 core, an external memory interface is available to load the sbox table from a processor flash or from on-chip non-volatile flash memory (NVM). By setting the INIT_SBOX_ROM=0, the ROM block is excluded from the core build and a 50% logic cell reduction is obtained. However, in this case the user is now responsible for loading 256 bytes of data through the memory interface before any encryption operations are performed. During this setting, the INIT_SBOX_DONE signal is unused. Figure 4 shows the timing required to initialize the sbox ram table using the external memory interface and Appendix A contains the data that needs to be loaded via the memory interface. A separate clock has been provided to be used to load the ram table (INIT_SBOX_CLK) and it does not need to be synchronous to the CLK input signal. However, it can be connected to the main CLK signal but that is the user's choice.

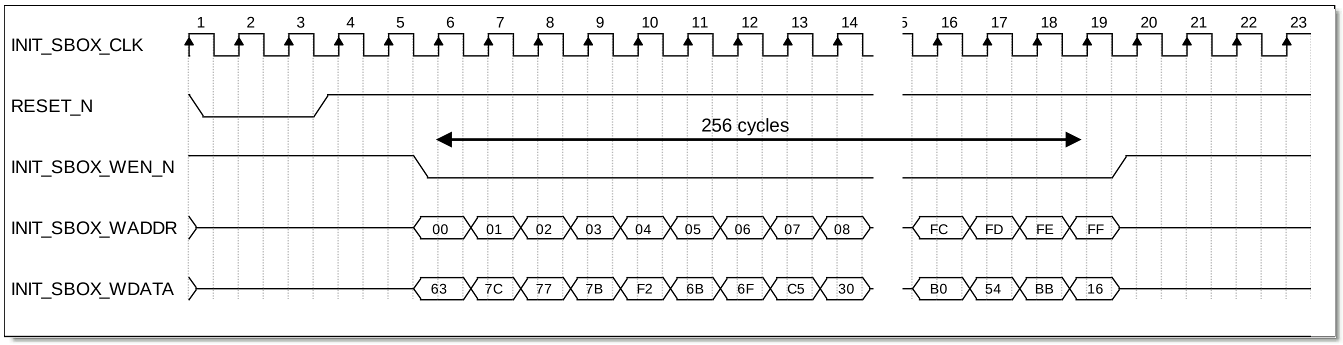


Figure 4 – External Ram Initialization Timing

If the user chooses too, this process of loading the sbox ram table can be repeated at any time during or after an encryption operation. The advantage being that the sbox ram table can be refreshed in case any bit errors occurred in the ram block due to neutron effects.

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 chosen key (regardless of size) to be expanded. During the key expansion step, the key is input to the core and stored in ram where it then gets passed through a logic chain ten times to produce ten sub-keys (in the case of a 128-bit key). These additional keys are also stored in ram to be later used in the actual encryption function (see Figure 5). The yellow shading indicates register stages that are present.

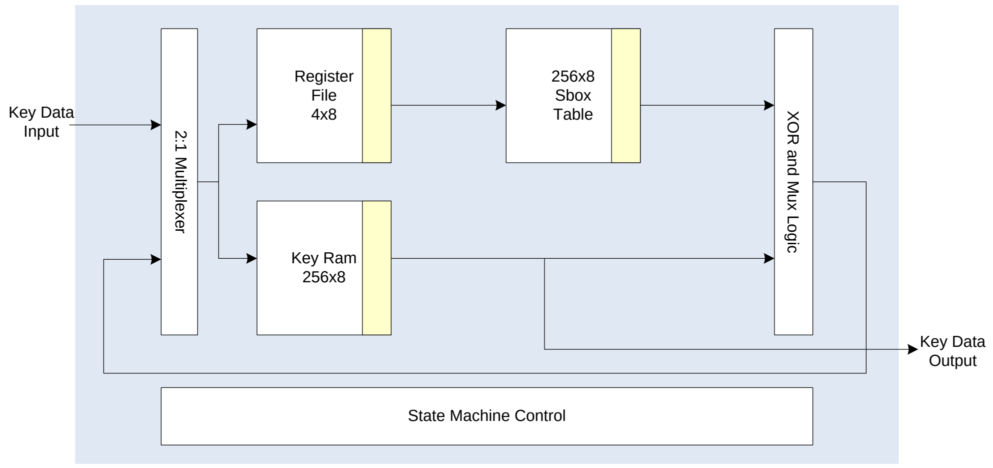


Figure 5 - Key Expansion Engine Block Diagram

Key expansion only needs to be done once before an encryption function is started. The only time it needs to be run again is when a different key is desired. The process of key expansion takes 198 (128 bit key), or 279 (256-bit key) clock cycles. Key expansion and encryption functions cannot be overlapped for this IP core. The timing diagram shown below in Figure 6 shows that KE_START initiates the key expansion and KE_DONE indicates its completion.

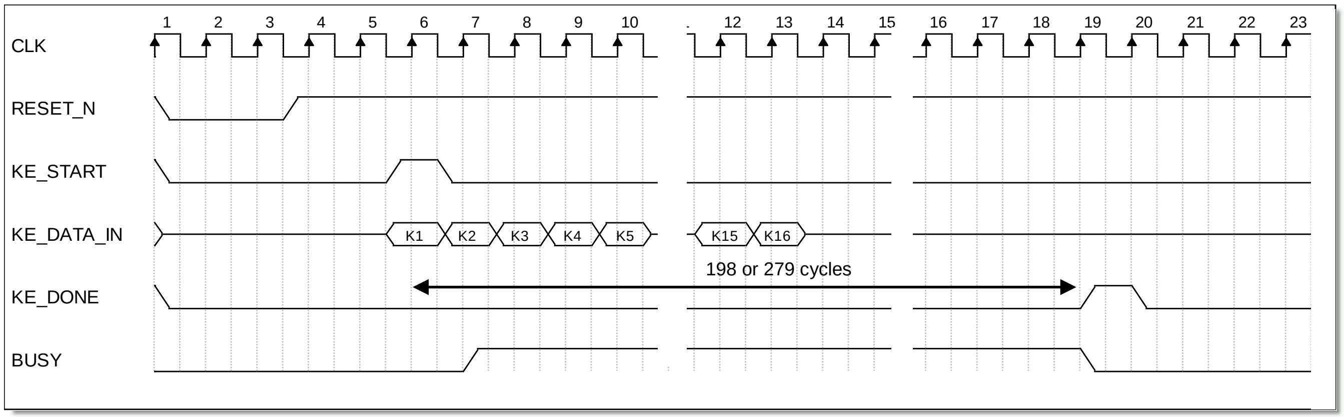


Figure 6 – Key Expansion Timing

Encryption Engine

This block is responsible for the AES algorithm encryption function. The stage sequencer shown in Figure 7 controls the AES processing steps. Similar to key expansion, multiple rounds are required for a complete AES encrypt function (10 or 14 rounds per Table 2). Since this core contains one encryption engine, each round must pass through the engine sequentially. The VTinyAESe_8 core uses an 8-bit data path for processing the 128-bit AES data block. Figure 7 below shows a block diagram of the encryption engine. To save resources, a ram block has been used to store the results from each round rather than registers. The sbox table shown below is shared with the key expansion sbox table from Figure 5.

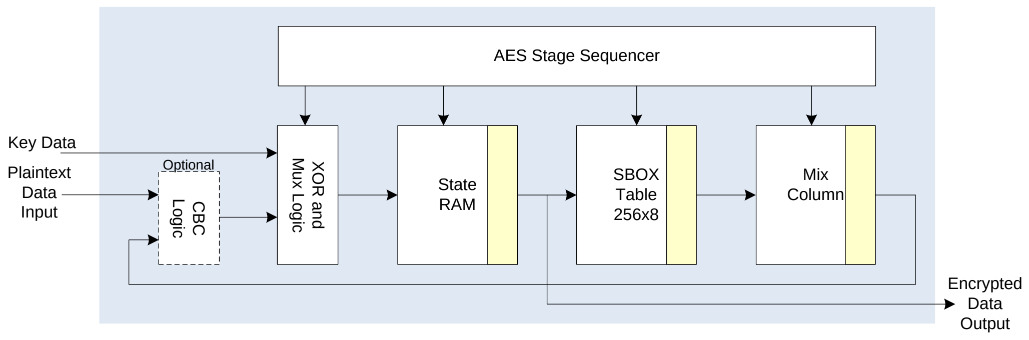


Figure 7 - Encryption Engine Block Diagram

The equation below shows the total number of clock cycles required for processing a 128-bit AES data block through the VTinyAESe_8 core.

(1) $\#cycles = (Nr \times Ce) + Ohd$

where Nr = the number of rounds (10 for 128-bit key, 14 for 256-key)
where Ce = the number of cycles to pass through the encryption engine (93)
where Ohd = the number of overhead cycles (17)

Using equation (1), we can compute the number of clock cycles required to encrypt a 128-bit block of data.

Table 6 - VTinyAESe_8 Cycle Count

Key Size	# Cycles
128 bits	(10 x 93) + 18 = 949
256 bits	(14 x 93) + 18 = 1321

The data throughput of VTinyAESe_8 can be calculated using the results from Table 7 and equation (2) below. Together with the operating frequency values from Table 1, the data throughput rate for each of the core variants, can be calculated for a 128-bit block of data. The throughput is measured in bits-per-second (bps).

(2) $\text{Throughput(bps)} = \text{frequency} \times (\#cycles)^{-1} \times 128$

For example with a 128-bit key size, if the speed of the chosen core variant runs at 183 MHz, then the throughput would be:

$$25 \text{ Mbps} = 183 \times 10^{6} \times (1/949) \times 128$$

25 Mega-bits-per-second indicates the sustained input data rate that can be supported by the VTinyAESe_8 while running at 183 MHz using a 128-bit input key.

Figure 8 shows the timing required to perform an encryption operation. The ENC_START signal begins the operation on the next rising edge of the clock. The first of the 16 bytes of plaintext words to be processed are input on the same rising edge as the ENC_START with the next 15 bytes input on successive clock cycles as shown by P2 through P16. After 947 clock cycles (with 128-bit key size), the ENC_DATA_VALID signal is asserted for 16 clock cycles to indicate that the ENC_DATA_OUT bus contains valid data. For each ENC_DATA_VALID clock cycle, a new byte is driven out on the ENC_DATA_OUT bus as shown by E1 through E16.

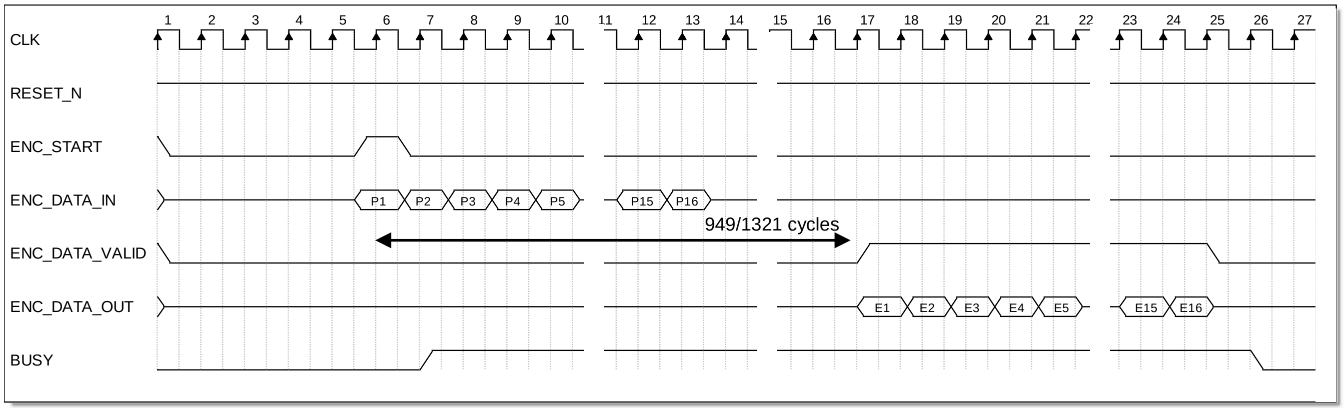


Figure 8 – Encryption Timing Without Fifo

The FIFO option is implemented as a full duplex 128x8 Fifo (which means a 128x8 input fifo and a 128x8 output fifo). The user can read results from the output fifo while the core is still busy. In the same way, plaintext data can be loaded into the input fifo while the core is busy. The user is responsible for monitoring the fifo flags because if data is not read from the output fifo then overflow can occur if the input fifo is continually loaded with data. The core will continue to run automatically, as long as the input fifo is not empty. Whenever the user is accessing the fifos, care must be taken to always read or write 16 bytes at a time to ensure that the flags are accurate and that the fifos are synchronized properly with the encryption engine and the key expansion engine.

Figure 9 below shows the required signal timing to interface to the input and output fifo blocks. Up to 8 128-bit plaintext data blocks can be preloaded into the input fifo before the ENC_START is asserted. The output fifo can begin to be read for encrypted data as soon as the FIFO_OUT_EMPTY goes LO to indicate the fifo is not empty.

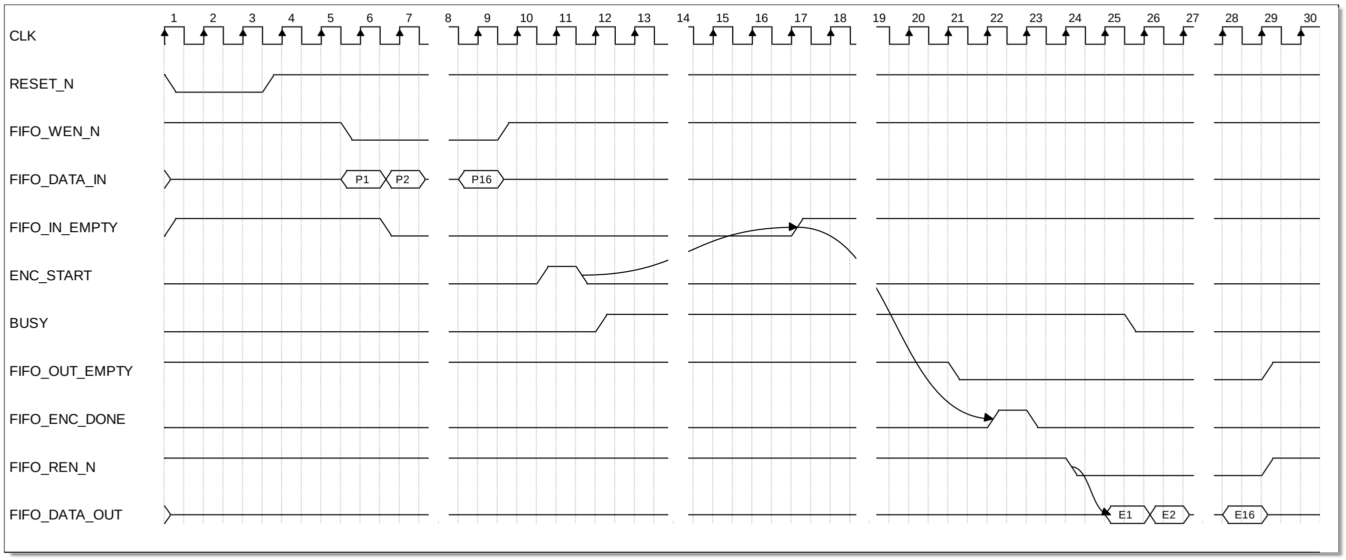


Figure 9 – Encryption Timing with Fifo

Data Formatting

The format and ordering of input data for key expansion and encryption is very important because a mismatch of expected data could occur if the user does not follow the convention described here. First, we will look at an example of key expansion and then an example of encryption.

128-bit Key Expansion Example

Per the FIPS-197 specification, an example of a sample 128-bit key is given by:

Key = 000102030405060708090a0b0c0d0e0f

Let us consider the key as 16 bytes of data organized from least significant byte on the left to most significant byte on the right as shown below:

LS Byte MS Byte

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15
00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f

We will consider each byte as an independent entity with the LSB (bit) of the byte to be the rightmost bit position and MSB (bit) to be the leftmost bit position as illustrated below. The entire key must be input sequentially starting with byte 0, byte 1 and so on until byte 15 has been input.

Clock	KE_DATA_IN(7:0)	
	7………………..0 (hex)	7………………..0 (binary)
1	00	0000 0000
2	01	0000 0001
3	02	0000 0010
4	03	0000 0011
5	04	0000 0100
6	05	0000 0101
7	06	0000 0110
8	07	0000 0111
9	08	0000 1000
10	09	0000 1001
11	0a	0000 1010
12	0b	0000 1011
13	0c	0000 1100
14	0d	0000 1101
15	0e	0000 1110
16	0f	0000 1111

Following this sequence will generate a key expansion schedule of:

Key Round	Key Value
0	000102030405060708090a0b0c0d0e0f
1	d6aa74fdd2af72fadaa678f1d6ab76fe
2	b692cf0b643dbdf1be9bc5006830b3fe
3	b6ff744ed2c2c9bf6c590cbf0469bf41

4	47f7f7bc95353e03f96c32bcfd058dfd
5	3caaa3e8a99f9deb50f3af57adf622aa
6	5e390f7df7a69296a7553dc10aa31f6b
7	14f9701ae35fe28c440adf4d4ea9c026
8	47438735a41c65b9e016baf4aebf7ad2
9	549932d1f08557681093ed9cbe2c974e
10	13111d7fe3944a17f307a78b4d2b30c5

Since the expanded keys generated are central to the integrity of the AES algorithm, the ram used for key storage is not accessible from outside the VTinyAESe_8 core. In fact, even during calculation and storage of the sub-keys, no key information is exposed outside of the core.

128-bit Encryption with ECB Example
Per the FIPS-197 specification, an example of a sample 128-bit plaintext input block with an expected AES encrypted output block (using the same key as above) is given by:

Key	= 000102030405060708090a0b0c0d0e0f
Plaintext	= 00112233445566778899aabbccddeeff
Encrypted	= 69c4e0d86a7b0430d8cdb78070b4c55a

Let us consider the input block as 16 bytes of plaintext data organized from least significant byte on the left to most significant byte on the right as shown below (same for encrypted output):

LS Byte MS Byte

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15
00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff

We will consider each byte as an independent entity with the LSB (bit) of the byte to be the rightmost bit position and MSB (bit) to be the leftmost bit position as illustrated below. The entire 16 byte block must be input via ENC_DATA_IN(7:0) sequentially starting with byte 0, byte 1 and so on until byte 15 has been input. Encrypted data is output on ENC_DATA_OUT(7:0) with byte 0, followed by byte 1, and so on until byte 15 has been output.

Clock	ENC_DATA_IN(7:0)		ENC_DATA_OUT(7:0)	
	7………………..0 (hex)	7………………..0 (binary)	7………………..0 (hex)	7………………..0 (binary)
1	00	0000 0000	69	0110 1001
2	11	0001 0001	c4	1100 0100
3	22	0010 0010	e0	1110 0000
4	33	0011 0011	d8	1101 1000
5	44	0100 0100	6a	0110 1010
6	55	0101 0101	7b	0111 1011
7	66	0110 0110	04	0000 0100
8	77	0111 0111	30	0011 0000
9	88	1000 1000	d8	1101 1000
10	99	1001 1001	cd	1100 1101
11	aa	1010 1010	b7	1011 0111
12	bb	1011 1011	80	1000 0000
13	cc	1100 1100	70	0111 0000
14	dd	1101 1101	b4	1011 0100
15	ee	1110 1110	c5	1100 0101

16	ff	1111 1111	5a	0101 1010

128-bit Encryption with CBC Example

Per the FIPS-197 specification, an example of a sample 128-bit plaintext input block with an expected AES encrypted output block (using the same key as above) is given by:

Key	= 2b7e151628aed2a6abf7158809cf4f3c
IV	= 000102030405060708090a0b0c0d0e0f
Plaintext	= 6bc1bee22e409f96e93d7e117393172a
Encrypted	= 7649abac8119b246cee98e9b12e9197d

First, expand the key by loading it and expanding it. Next, by asserting LD_IV and
Let us consider the input block as 16 bytes of plaintext data organized from least significant byte on the left to most significant byte on the right as shown below (same for encrypted output):

We will consider each byte as an independent entity with the LSB (bit) of the byte to be the rightmost bit position and MSB (bit) to be the leftmost bit position as illustrated below. The entire 16 byte block must be input via ENC_DATA_IN(7:0) sequentially starting with byte 0, byte 1 and so on until byte 15 has been input. Encrypted data is output on ENC_DATA_OUT(7:0) with byte 0, followed by byte 1, and so on until byte 15 has been output.

1st Step			2nd Step			3rd Step	
LD_IV	**IV_DATA**		**ENC START**	**DATA_IN**		**ENC DATA_VALID**	**DATA_OUT**
	7………………0			**7……………..0**			**7………..……..0**
1	00		1	6B		1	76
1	01		0	C1		1	49
1	02		0	BE		1	AB
1	03		0	E2		1	AC
1	04		0	2E		1	81
1	05		0	40		1	19
1	06		0	9F		1	B2
1	07		0	96		1	46
1	08		0	E9		1	CE
1	09		0	3D		1	E9
1	0a		0	7E		1	8E
1	0b		0	11		1	9B
1	0c		0	73		1	12
1	0d		0	93		1	E9
1	0e		0	17		1	19
1	0f		0	2A		1	7D

Verification

Figure 11 shows the architecture of the verification testbench used to validate the VTinyAESe_8 core variants.

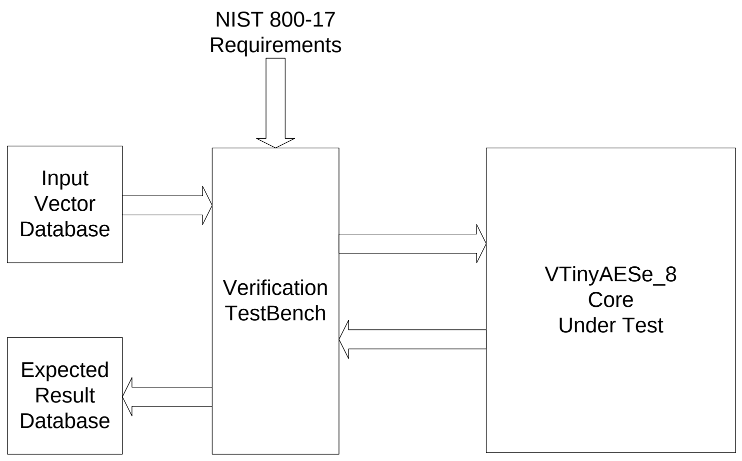
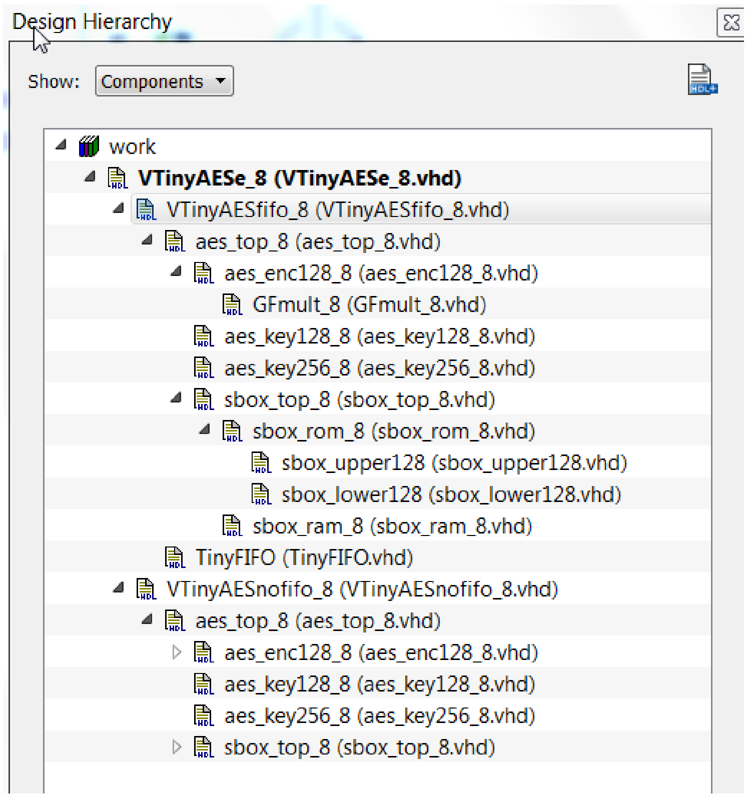


Figure 10 – Verification Block Diagram

Deliverables

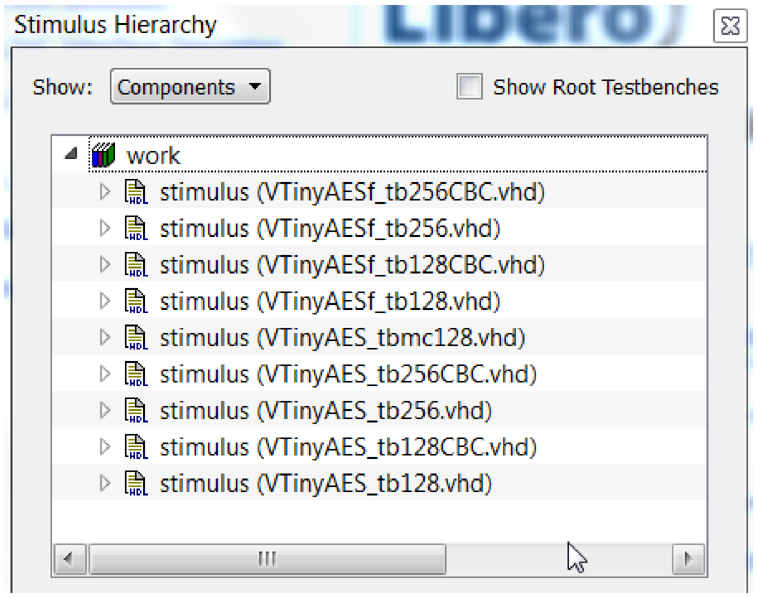
Table 8 below shows the source code hierarchy of the VTinyAESe_8 core. The current RTL deliverable is VHDL, however, verilog RTL can be provided on request.



```
Design Hierarchy
Show: Components

◢ work
  ◢ VTinyAESe_8 (VTinyAESe_8.vhd)
    ◢ VTinyAESfifo_8 (VTinyAESfifo_8.vhd)
      ◢ aes_top_8 (aes_top_8.vhd)
        ◢ aes_enc128_8 (aes_enc128_8.vhd)
            GFmult_8 (GFmult_8.vhd)
          aes_key128_8 (aes_key128_8.vhd)
          aes_key256_8 (aes_key256_8.vhd)
        ◢ sbox_top_8 (sbox_top_8.vhd)
          ◢ sbox_rom_8 (sbox_rom_8.vhd)
              sbox_upper128 (sbox_upper128.vhd)
              sbox_lower128 (sbox_lower128.vhd)
            sbox_ram_8 (sbox_ram_8.vhd)
        TinyFIFO (TinyFIFO.vhd)
    ◢ VTinyAESnofifo_8 (VTinyAESnofifo_8.vhd)
      ◢ aes_top_8 (aes_top_8.vhd)
        ▷ aes_enc128_8 (aes_enc128_8.vhd)
          aes_key128_8 (aes_key128_8.vhd)
          aes_key256_8 (aes_key256_8.vhd)
        ▷ sbox_top_8 (sbox_top_8.vhd)
```

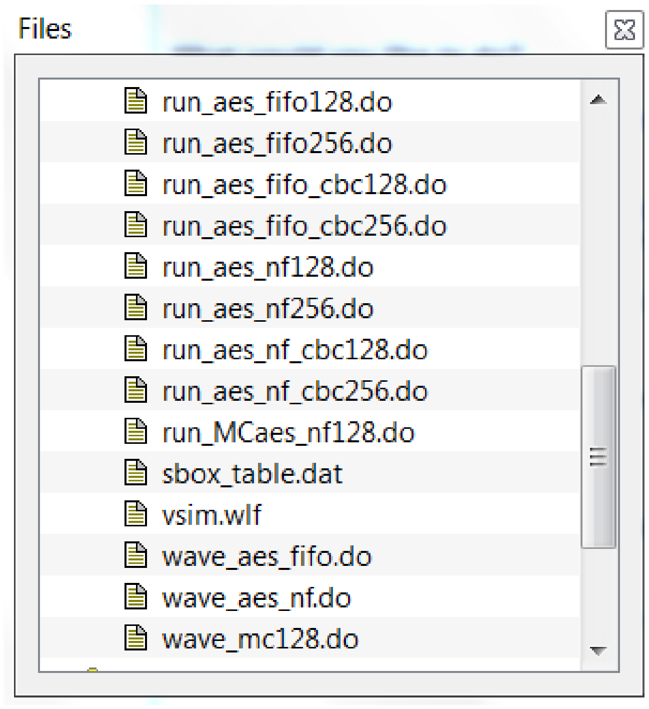
Figure 11 - Source Code Hierarchy



```
Stimulus Hierarchy
Show: Components          ☐ Show Root Testbenches

◢ work
  ▷ stimulus (VTinyAESf_tb256CBC.vhd)
  ▷ stimulus (VTinyAESf_tb256.vhd)
  ▷ stimulus (VTinyAESf_tb128CBC.vhd)
  ▷ stimulus (VTinyAESf_tb128.vhd)
  ▷ stimulus (VTinyAES_tbmc128.vhd)
  ▷ stimulus (VTinyAES_tb256CBC.vhd)
  ▷ stimulus (VTinyAES_tb256.vhd)
  ▷ stimulus (VTinyAES_tb128CBC.vhd)
  ▷ stimulus (VTinyAES_tb128.vhd)
```

Figure 12 - Testbench Files


Files

run_aes_fifo128.do
run_aes_fifo256.do
run_aes_fifo_cbc128.do
run_aes_fifo_cbc256.do
run_aes_nf128.do
run_aes_nf256.do
run_aes_nf_cbc128.do
run_aes_nf_cbc256.do
run_MCaes_nf128.do
sbox_table.dat
vsim.wlf
wave_aes_fifo.do
wave_aes_nf.do
wave_mc128.do

Figure 13 - Simulation Files

Appendix A – External Initialization of SBOX Ram

Sbox RAM Load Pattern (256 bytes)

ADDR	DATA
0	63
1	7C
2	77
3	7B
4	F2
5	6B
6	6F
7	C5
8	30
9	01
0A	67
0B	2B
0C	FE
0D	D7
0E	AB
0F	76
10	CA
11	82
12	C9
13	7D
14	FA
15	59
16	47
17	F0
18	AD
19	D4
1A	A2
1B	AF
1C	9C
1D	A4
1E	72
1F	C0
20	B7
21	FD
22	93
23	26
24	36
25	3F
26	F7
27	CC
28	34
29	A5
2A	E5
2B	F1
2C	71
2D	D8
2E	31
2F	15
30	04
31	C7
32	23
33	C3
34	18
35	96
36	05
37	9A
38	07
39	12
3A	80
3B	E2
3C	EB
3D	27
3E	B2
3F	75
40	09
41	83
42	2C
43	1A
44	1B
45	6E
46	5A
47	A0
48	52
49	3B
4A	D6
4B	B3
4C	29
4D	E3
4E	2F
4F	84
50	53
51	D1
52	00
53	ED
54	20
55	FC
56	B1
57	5B
58	6A
59	CB
5A	BE
5B	39
5C	4A
5D	4C
5E	58
5F	CF
60	D0
61	EF
62	AA
63	FB
64	43
65	4D
66	33
67	85
68	45
69	F9
6A	02
6B	7F
6C	50
6D	3C
6E	9F
6F	A8
70	51
71	A3
72	40
73	8F
74	92
75	9D
76	38
77	F5
78	BC
79	B6
7A	DA
7B	21
7C	10
7D	FF
7E	F3
7F	D2
80	CD
81	0C
82	13
83	EC
84	5F
85	97
86	44
87	17
88	C4
89	A7
8A	7E
8B	3D
8C	64
8D	5D
8E	19
8F	73
90	60
91	81
92	4F
93	DC
94	22
95	2A
96	90
97	88
98	46
99	EE
9A	B8
9B	14
9C	DE
9D	5E
9E	0B
9F	DB
A0	E0
A1	32
A2	3A
A3	0A
A4	49
A5	06
A6	24
A7	5C
A8	C2
A9	D3
AA	AC
AB	62
AC	91
AD	95
AE	E4
AF	79
B0	E7
B1	C8
B2	37
B3	6D
B4	8D
B5	D5
B6	4E
B7	A9
B8	6C
B9	56
BA	F4
BB	EA
BC	65
BD	7A
BE	AE
BF	08
C0	BA
C1	78
C2	25
C3	2E
C4	1C
C5	A6
C6	B4
C7	C6
C8	E8
C9	DD
CA	74
CB	1F
CC	4B
CD	BD
CE	8B
CF	8A
D0	70
D1	3E
D2	B5
D3	66
D4	48
D5	03
D6	F6
D7	0E
D8	61
D9	35
DA	57
DB	B9
DC	86
DD	C1
DE	1D
DF	9E
E0	E1
E1	F8
E2	98
E3	11
E4	69
E5	D9
E6	8E
E7	94
E8	9B
E9	1E
EA	87
EB	E9
EC	CE
ED	55
EE	28
EF	DF
F0	8C
F1	A1
F2	89
F3	0D
F4	BF
F5	E6
F6	42
F7	68
F8	41
F9	99
FA	2D
FB	0F
FC	B0
FD	54
FE	BB
FF	16