



***Implementing the XTS-AES Standard on
Xtensa® Processors***
Application Note

Cadence Design Systems, Inc.
2566 Seely Ave.
San Jose, CA 95134

www.cadence.com

© 2015 Cadence Design Systems, Inc.

All rights reserved worldwide.

This publication is provided "AS IS." Cadence Design Systems, Inc. (hereafter "Cadence") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Information in this document is provided solely to enable system and software developers to use our processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Cadence integrated circuits or integrated circuits based on the information in this document. Cadence does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

© 2014 Cadence, the Cadence logo, Allegro, Assura, Broadband Spice, CDNLIVE!, Celtic, Chipestimate.com, Conformal, Connections, Denali, Diva, Dracula, Encounter, Flashpoint, FLIX, First Encounter, Incisive, Incyte, InstallScape, NanoRoute, NC-Verilog, OrCAD, OSKit, Palladium, PowerForward, PowerSI, PSpice, Purespec, Puresuite, Quickcycles, SignalStorm, Sigrity, SKILL, SoC Encounter, SourceLink, Spectre, Specman, Specman-Elite, SpeedBridge, Stars & Strikes, Tensilica, TripleCheck, TurboXim, Vectra, Virtuoso, VoltageStorm, Xplorer, Xtensa, and Xtreme are either trademarks or registered trademarks of Cadence Design Systems, Inc. in the United States and/or other jurisdictions. All other trademarks are the property of their respective holders.

OSCI, SystemC, Open SystemC, Open SystemC Initiative, and SystemC Initiative are registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Document Change History:

December 2002
February 2005
August 2005
February 2013
November 2015 – format only

Contents

1.	Introduction: What is AES?.....	1
1.1	XTS Extension.....	1
2.	AES-ECB Encryption.....	2
3.	Xtensa Processor Extensions for XTS-AES	5
4.	The AES Register File	7
5.	128-bit AES Encryption	9
5.1	The ShiftRow Transformation	9
5.2	The SubBytes Transformation.....	12
5.3	The AddRoundKey Transformation	13
5.4	The MixColumns Transformation.....	14
5.5	Programming the AES Engine.....	17
6.	Key Expansion.....	19
7.	XTS Extensions	23
8.	Optimization: Instruction Combining	25
9.	Integrating XTS Extensions into AES	31
10.	Timing and Clock Period Considerations.....	33
10.1	AES Decryption	37
11.	Conclusion.....	38
	Example XTS-AES Test Bench.....	39
11.1	Unpacking Example and Attaching TIE to a Processor	39
11.2	Running the Example XTS-AES Application.....	40

Figures

Figure 1: The State Array	2
Figure 2: Basic AES Encryption	3
Figure 3: The Xtensa Processor Extended With XTS-AES Cipher Functionality	5
Figure 4: Shiftrow Transformation On State Array	9
Figure 5: 128-BIT AES Key Expansion	19
Figure 6: XTS-AES Extensions	23
Figure 7: 128-Bit LFSR For T Key Update	23
Figure 8: Combined XTS-AES Instructions	26
Figure 9: Block Diagram of the aes_encode TIE Semantic.....	29
Figure 10: Default AES Operation Schedule.....	34
Figure 11: Pipeline Diagram Of AES Encryption, Default Operation Schedule.....	35
Figure 12: Improved AES Operation Schedule	36
Figure 13: Pipeline Diagram of AES Encryption, Improved Operation Schedule	37

Abstract

This application note examines the Advanced Encryption Standard (AES) cipher and illustrates the power of a configurable processor in accelerating encryption and decryption. Xtensa-based application-specific processors are quickly designed, simulated, and instantiated in silicon. These processors offer performance that rivals hardware solutions along with the benefits of flexibility, programmability, and ease of verification found with purely software implementations.

The structure of this application note begins with a brief introduction of the AES cipher and then details the process of creating a highly efficient application-specific processor for this algorithm. An Xtensa processor configuration is defined and basic extensions using the Tensilica Instruction Extension (TIE) Language are designed to accelerate the AES cipher. Then, general optimization techniques such as instruction combining, pipelining and parallel processing, are explored to further improve the performance of the AES instructions. It is important to note that these optimization techniques are general and can apply to many algorithms other than the AES cipher.

Finally, the XTS storage extensions to the AES cipher are developed, and included in the example TIE source and software test bench that demonstrates these techniques.

This application note is written for the reader who is familiar with the Tensilica Instruction Extension description language, PERL, and the Xtensa Instruction Set Architecture. Please refer to the *Xtensa ISA Reference Manual* and the *Tensilica Instruction Extension (TIE) Language User's Guide* for additional information. The application note also assumes that the reader is familiar with the AES cipher and concepts of cryptography and Galois Field Arithmetic. This application note is not intended as a tutorial for AES, nor does it provide any theoretical basis for assessing the security of the cipher.

1. Introduction: What is AES?

The Data Encryption Standard (DES) cipher has been widely used for secure network communications since the 1970s. At that time, the security provided by DES was considered more than sufficient. Assuming that the best attack on DES is an exhaustive search of its 56-bit key, cracking DES was considered impractical with the available technology of the time. However, recent advances in technology have enabled recovery of DES keys in as little as a few hours. Many secure networks have responded to this real security threat by implementing triple DES, which is the serial application of three DES ciphers, each with its own individual keys. This provided an effective security of a 112-bit key, but at the expense of tripling the bandwidth. With the ever-increasing need for high bandwidth and secure communications, the cipher can easily become the data bottleneck. The DES cipher is quickly becoming obsolete.

In 1997, the NIST (U.S. Department of Commerce, National Institute of Standards and Technology) held a public contest to design the Advanced Encryption Standard (AES) cipher, a successor to the aging DES cipher. This effort involved the cooperation of the U.S. government, the private sector, and academia from around the world. In October 2000, the Rijndael block cipher was chosen amongst several candidate ciphers.

Dr. Joan Daemen and Dr. Vincent Rijmen developed the Rijndael block cipher. The Rijndael cipher was chosen for the AES, on the basis of its simplicity, security and flexibility. The Rijndael cipher is a cipher that can be easily implemented in both hardware and software. In addition, the Rijndael cipher structure has excellent potential for parallelism, lending itself to many performance optimization techniques (as will be seen in the upcoming sections). The Rijndael cipher is highly secure and flexible by virtue of variable block lengths and wide key lengths. However, AES limits the use of keys to a length of 128, 192, or 256 bits to encrypt a 128-bit block.

1.1 XTS Extension

The XTS extension to AES is a tweakable XOR-Encrypt_XOR (XEX) mode of AES that takes a tweak value derived from a combination of the disk sector and logical block numbers. The plain text block is XORed with the tweak value before AES encryption of the block, and the result is also XORed with the tweak value before storing the cipher block. In addition, the XTS mode allows Cypher Text Stealing for sectors that are not evenly divisible by the block size. These algorithms are described in IEEE Specification 1619-2007, and the XTS encryption mode was approved for disks by NIST in 2010 (Publication 800-38E).

This application note examines the process of designing an “XTS-AES engine”, an application-specific processor developed around Xtensa technology. Specifically, the Tensilica Instruction Extensions (TIE) language is used to design an optimized processor for AES. For ease of exposition, the focus of this application note is primarily on 128-bit XTS-AES encryption using 256-bit keys, treated as two 128-bit keys. However, the solution proposed in this application note can be used to support XTS-AES encryption and decryption using other key and encryption block sizes as defined in the IEEE specification.

Before delving into the design of this AES engine, take a moment to review and understand the algorithmic structure of AES encryption.

2. AES-ECB Encryption

The AES cipher is done on a block-by-block basis, encrypting a *plaintext* (original message) block to a *ciphertext* (encrypted) block. Each block of plaintext is 128 bits (16 bytes) in length and is conceptually stored and operated upon in a state array, which is a 4x4 byte array. This is shown in Figure 1.

16 bytes of plain text

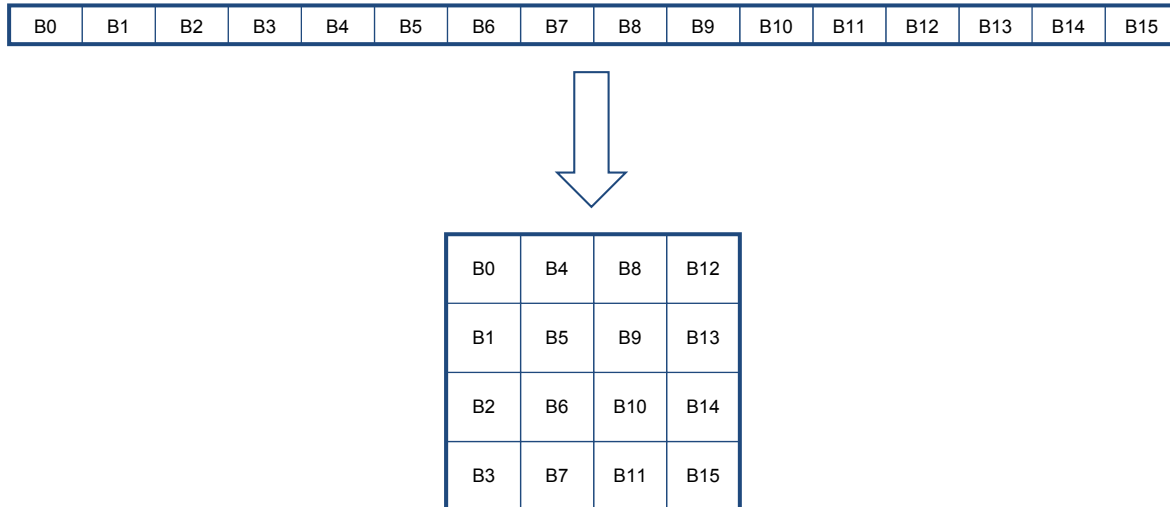


FIGURE 1: THE STATE ARRAY

Note the order in which data is stored into the state array. Bytes of the plaintext are stored in the upper left square, moving downward in the left column, and then proceeding to the next columns until the lower right square is filled. Byte placement is important because AES cipher transformations operate on rows or columns of the state array.

Like the DES cipher, the AES cipher is based on a secret key. The AES uses either a 128- or 256-bit key to encrypt the plaintext into ciphertext. The same key must be used to decrypt the ciphertext back into plaintext. The XTS extensions to AES require a second key, and the two keys are generally treated as two 128-bit halves of a 256-bit key. The techniques described here are for 128-bit keys only, or half of 256-bit keys.

Prior to encryption the 128-bit key is expanded to 10 additional 128-bit keys. The original key, along with the 10 expanded keys, is referred to as the key schedule. All keys of the key schedule will be used during encryption.

AES encryption consists of four transformations that are briefly described below.

- ◆ SubBytes – a substitution transformation which does a bitwise substitution for each byte in the state array using a fixed table
- ◆ ShiftRows – a transposition transformation which rearranges the placement of bytes within the state array
- ◆ MixColumns – a permutation transformation that operates on the state array, treating each column of the state array as a polynomial and performing a Galois field multiplication with a fixed 4x4 matrix

- ◆ AddRoundKey – a permutation operation that performs a bitwise Galois field addition with bytes of a key schedule

These transformations are applied to the state array in the order shown in the flowchart in Figure 2.

The AddRoundKey transformation is applied prior to the encryption rounds (round 0). The following encryption rounds apply the same set of transformations with the exception of the last round (round 10), in which the MixColumns transformation is excluded.

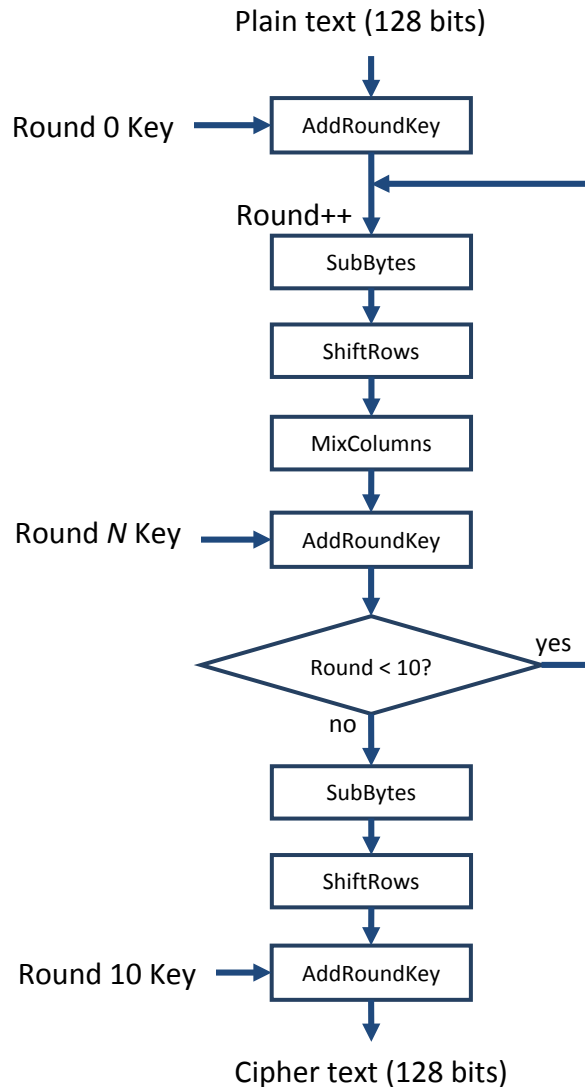


FIGURE 2: BASIC AES ENCRYPTION

AES has a very simple structure that is typically implemented on both hardware and software. A System-on-Chip (SOC) will provide much more functionality than simply performing a cipher. The SOC may be required to handle interface protocols, memory management, security, routing, miscellaneous network functions, operating system functions, and user interface. A high performance processor can perform these functions along with handling the AES cipher. Porting the AES cipher to a general-purpose processor is simple; however, the

performance of such an implementation (encryption/decryption throughput) would be poor because it takes too many instructions to complete. Due to this issue, software-only implementations will generally be ruled out except for some secure applications that have very low bandwidth requirements.

A hardware solution would consist of a hardware block outside of the processor. Combinational logic can be designed to support the AES transformations, while a state machine can be used to schedule the transformations. The main benefit of a hardware solution is the high performance and power efficiency that can be achieved when compared to a software-only solution. However, hardware acceleration does introduce significant complexity to the design, debugging, and verification of an SOC.

An alternative approach is to use an application-specific processor. An application-specific processor can provide performance that rivals hardware solutions and also offers the flexibility and ease of design found in software solutions.

3. Xtensa Processor Extensions for XTS-AES

The Xtensa processor is configurable, allowing you to create a processor that has a feature set optimized to your application. Moreover, the Xtensa processor is also extensible, allowing you to extend the processor by defining application-specific instructions in the Tensilica Instruction Extension (TIE) language. TIE is compiled into hardware inside the Xtensa processor and its software tools are automatically extended to support the new instructions. Let's consider the design of an application-specific processor for AES, by taking advantage of the configurability and extensibility of the Xtensa processor. A block diagram of the AES engine is shown in Figure 3.

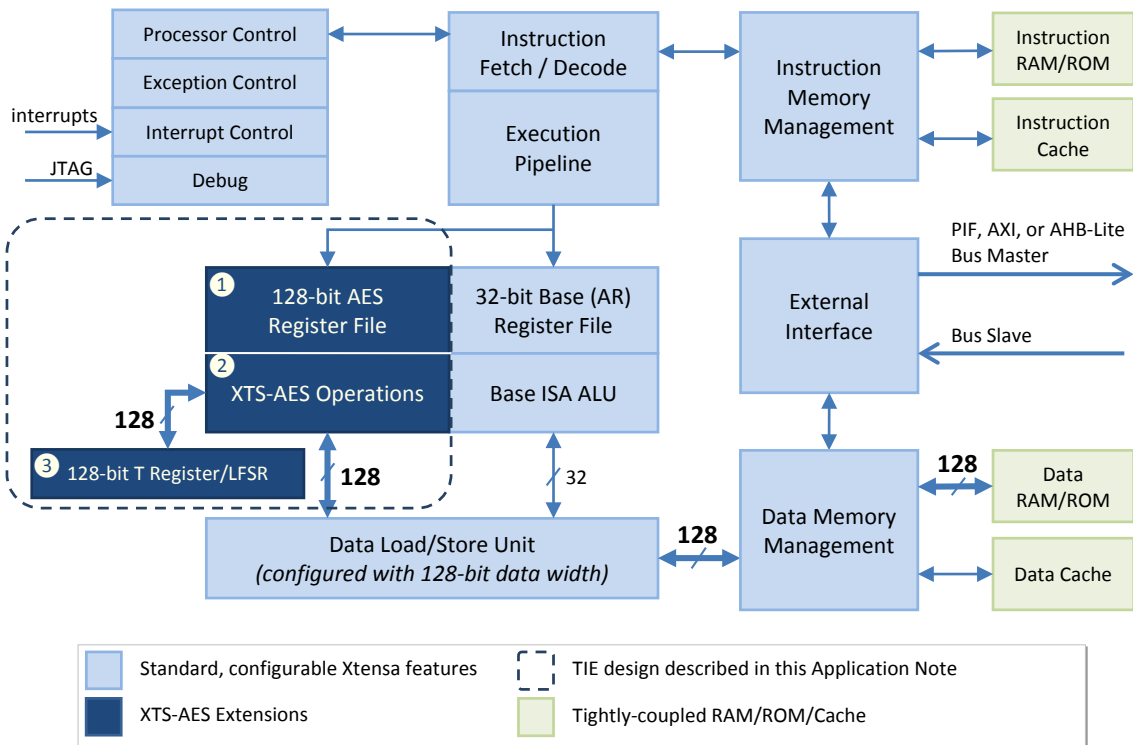


FIGURE 3: THE XTENSA PROCESSOR EXTENDED WITH XTS-AES CIPHER FUNCTIONALITY

Figure 3 shows an Xtensa processor block diagram, configured and extended for XTS-AES functions. Most blocks represent features of the Xtensa processor that are part of every Xtensa processor – basic features like instruction fetch, the execution pipeline, and the base ISA ALU. Many features of the Xtensa processor are configurable, such as the number of tightly-coupled RAM or ROM elements, and the presence of I- or D-Cache and their attributes like total size, cache line size, and associativity. The lightly-shaded blocks on the right of the diagram represent instances of ASIC SRAM connected to Xtensa. The open arrows on the right indicate Xtensa's interface to a system bus, which may be PIF, AXI, or AHB-Lite (the arrow points in the direction of the bus request channel). Xtensa may also be configured to have a system bus slave, called *Inbound PIF*, which gives external requestors (like DMA engines or other processors) direct access to the local memories of the processor.

Among the configuration options, the data load/store unit may be configured with data widths wider than the standard 32-bit (default) width. In the case of the custom operations needed for XTS-AES, a 128-bit load/store unit enables encryption text blocks and keys to be loaded

or stored into local RAM/DCache using a single memory operation. This 128-bit data load/store unit width is essentially the only processor configuration choice required for the implementation of XTS-AES described here. The TIE design that is included in the associated workspace can be attached to virtually any Xtensa processor, provided it has 128-bit load/store unit width, and a 5-stage instruction pipeline.

The dashed line in Figure 3 delineates the extent of the TIE-language design for XTS-AES described in this Application Note. First, a 128-bit AES register file ❶ is created to handle 128-bit quantities. This register file is associated with 128-bit C-types for use in C programming. AES-specific operations ❷ are designed to perform the various transformations on the 128-bit data types stored in the AES register file, including wide loads and stores. In addition, a 128-bit hardware register and linear-feedback shift register (LFSR) ❸ is added to the processor to hold the XTS encoded twiddle factor T.

The TIE language is used to describe processor extensions that are directly compiled by the TIE Compiler (TC) into hardware descriptions, and integrated into the RTL of the Xtensa core. The instruction fetch and decode functions of the processor are extended to include the new operations defined in TIE, as well as operand data paths and register file ports. In addition, the Xtensa software tools are automatically extended to support the new operations and C-types defined in the TIE design source.

All of the hardware and software features described here can operate on data residing in system memory accessed through the system bus master (PIF, AXI or AHB-Lite), but each reference (or cache miss) would experience the system bus latency. However, the description of XTS-AES operations that follows, and their performance, assumes that all data resides in a tightly-coupled local data RAM, or is resident in the DCache. We assume that the AES key schedule is created in local memory, all plain text blocks begin in local memory, and encrypted cipher text blocks are also stored back to local memory. The method of obtaining the encryption key, as well as the method of moving data blocks in and out of local data memory, is beyond the scope of this Application Note. This data could be loaded into local memory from the system bus by running `memcpy` operations on the processor, or an external requestor could deposit the data directly into the processor's data RAM by way of Inbound PIF.

An encryption key is initially loaded to the local data RAM. The processor then performs key expansion, and the resulting key schedule is stored into local data RAM memory. Each iteration (or round) of the AES encryption algorithm reads the key for the round from local memory.

Encryption begins after blocks of plaintext are loaded into the local data memory. An interrupt to the Xtensa core is asserted to wake the processor if it is in low power idle mode, or force a task switch if the Xtensa core is handling some lower priority function. The Xtensa processor then loads the 128-bit plaintext using a single TIE load instruction into the AES register file. Encryption rounds will operate on the AES register file until encryption is completed. Finally, the 128-bit ciphertext is written from the AES register file to the local data memory using a single TIE store instruction.

The following sections of this application note focuses on individual AES transformations and extensions added to the Xtensa processor to accelerate AES encryption rounds.

4. The AES Register File

The base Xtensa processor has a 32-bit wide register file. Given that AES performs transformations on a 128-bit state array, it would be ideal if the processor had a 128-bit register file specifically for AES transformations. A 128-bit register file will allow instructions to operate on all 128 bits of a state array concurrently, as opposed to 32 bits at a time. Combined with the wide data bus, the wide register file will also enable the state array to be transferred to and from memory with a single memory access, rather than in four 32-bit accesses. This section will cover the design of an application-specific register file and associated load/store/move instructions to support the register file. This register file will be used as the source and destination for custom transformation instructions that are covered in later sections.

The TIE language enables the Xtensa processor to be extended with custom register files. The custom register file can be up to 1024 bits wide and from 2 to 64 registers deep. The AES engine can take advantage of this capability by extending the processor with a 128-bit wide register file, named the AES register file.

It is not necessary to write assembly language to support the AES register file. The C compiler is automatically extended to support the AES register file. The TIE language allows you to define a custom data type that can be used within the software program. Just like C's built-in data types (int, short, char, etc.), the compiler manages register allocation of the program variables of the custom data type to the custom register file. If a software program uses many instances of the custom data type, you could improve performance by increasing the number of registers in the register file. Due to the simplicity of the AES cipher, a two-register AES register file is chosen to minimize the core area. The AES register file is easily defined in TIE as follows:

```
//Declare 128bit regfile aes
regfile aes 128 2 aes
```

A register file requires several basic instructions to allow the compiler to manage the register file. One of the required instructions is a move instruction that transfers data from one register in the AES register file to another register. The other required instructions are indexed load/store instructions that transfer data between the external memory space and the register file. An indexed load/store instruction generates its effective address by adding an immediate index in the instruction with a base address in the Xtensa processor's AR register file. These instructions are automatically created by the TIE compiler for every register file that is declared, and for the AES register file, they are called MOV.AES, LD.AES, and ST.AES. The C/C++ compiler will automatically use these instructions to facilitate variables of C-type "aes" associated with the register file.

This sample C code shows how the `aes` data type is used to load/store data between a register in the AES register file and memory.

```
char ciphertext[16] __attribute__((aligned(16)));
char plaintext[16] __attribute__((aligned(16))) =
{0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x99,0xaa,0xbb,0xcc,0xdd,
0xee,0xff};

// the "aes" C-Type is associated with the AES register file
aes mystate; // this variable is in the AES register file
```

```
// pointers to AES register values in memory
aes *plaintext_buf, *ciphertext_buf;

// use type casting to assign aes pointers to char buffers
plaintext_buf = (aes *)plaintext;
ciphertext_buf = (aes *)ciphertext;

mystate = *plaintext_buf; // compiler infers LD.AES load

. . . . perform encryption on mystate

*ciphertext_buf = mystate // compiler infers ST.AES store
```

A 16-byte char array is declared for plaintext and ciphertext. Recall that the Xtensa processor requires that 128-bit (16-byte) memory access be performed on 16-byte boundaries. Therefore, the alignment of these arrays must be enforced using a declaration attribute (`aligned`) to access the array using 128-bit load/store instructions.

AES pointers, `plaintext_buf` and `ciphertext_buf`, are declared and assigned the address of plaintext and ciphertext respectively. The `mystate` variable is allocated a register in the AES register file and loaded with 16 bytes from `plaintext_buf`. After some transformations are performed on the `mystate` variable, `mystate` is stored to `ciphertext_buf`.

Note that use of the register file is done without any use of assembly language. The TIE description defines how the compiler manages the `aes` C-type which is automatically generated for the AES register files, as it is for every register file. Additional C-types can be associated with the AES register file as well; the TIE language describes both hardware elements and compiler extensions to use them.

Now that the AES register file is specified, individual AES transformations that operate on the register file can be described. The following sections cover each AES transformation and explore how these transformations can be accelerated with TIE instructions that operate on the AES register file.

5. 128-bit AES Encryption

5.1 The ShiftRow Transformation

ShiftRow is a transposition transformation that rearranges the order of bytes within the state array. As the name of the transformation implies, shifting (rotating) is performed on bytes within a row in the state array. A diagram of the ShiftRow transformation is shown in the figure below.

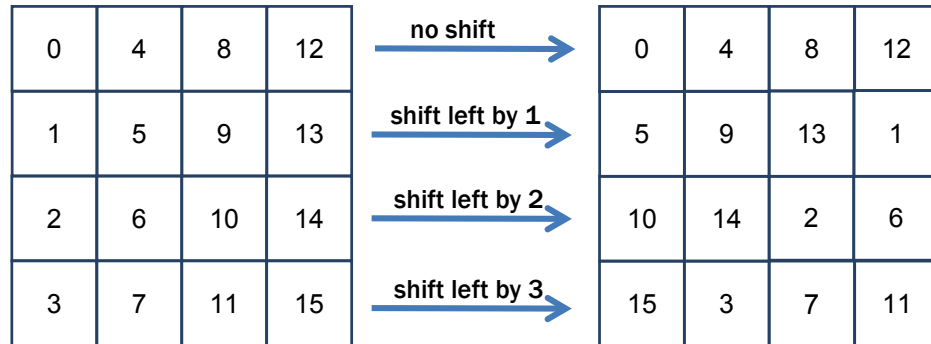


FIGURE 4: SHIFTRW TRANSFORMATION ON STATE ARRAY

The uppermost row is not shifted. The second, third, and fourth row are shifted left by 1, 2, and 3 bytes respectively. This transformation can be performed in C using the code below:

```
void shiftrow(char input[], char output[])
{
    output[0] = input[0] ; //transpose row 1
    output[4] = input[4] ;
    output[8] = input[8] ;
    output[12] = input[12] ;

    output[1] = input[5] ; //transpose row 2
    output[5] = input[9] ;
    output[9] = input[13] ;
    output[13] = input[1] ;

    output[2] = input[10] ; //transpose row 3
    output[6] = input[14] ;
    output[10] = input[2] ;
    output[14] = input[6] ;

    output[3] = input[15] ; //transpose row 4
    output[7] = input[3] ;
    output[11] = input[7] ;
    output[15] = input[11] ;

    return ;
}
```

The ShiftRow transposition of the input state array is performed using bitwise assignment. Each assignment requires a load and store operation for a total of 32 operations when performed in software. This transformation can be significantly accelerated in TIE by taking advantage of the 128-bit data path, and performing all byte transpositions in parallel, instead of performing the transformation a byte at a time. This transformation can be done with a single TIE instruction that executes in a single cycle. The TIE description for the SHIFTRW instruction is shown below.

```
operation SHIFTRW {inout aes aesr} {}
{
  //split aesr into individual bytes
  wire [7:0] in_0 = aesr[127:120];
  wire [7:0] in_1 = aesr[119:112];
  wire [7:0] in_2 = aesr[111:104];
  wire [7:0] in_3 = aesr[103:96];
  wire [7:0] in_4 = aesr[95:88];
  wire [7:0] in_5 = aesr[87:80];
  wire [7:0] in_6 = aesr[79:72];
  wire [7:0] in_7 = aesr[71:64];
  wire [7:0] in_8 = aesr[63:56];
  wire [7:0] in_9 = aesr[55:48];
  wire [7:0] in_10 = aesr[47:40];
  wire [7:0] in_11 = aesr[39:32];
  wire [7:0] in_12 = aesr[31:24];
  wire [7:0] in_13 = aesr[23:16];
  wire [7:0] in_14 = aesr[15:8];
  wire [7:0] in_15 = aesr[7:0];

  //transpose row 1
  wire [7:0] out_0 = in_0;
  wire [7:0] out_4 = in_4;
  wire [7:0] out_8 = in_8;
  wire [7:0] out_12 = in_12;

  //transpose row 2
  wire [7:0] out_1 = in_5;
  wire [7:0] out_5 = in_9;
  wire [7:0] out_9 = in_13;
  wire [7:0] out_13 = in_1;

  //transpose row 3
  wire [7:0] out_2 = in_10;
  wire [7:0] out_6 = in_14;
  wire [7:0] out_10 = in_2;
  wire [7:0] out_14 = in_6;

  //transpose row 4
  wire [7:0] out_3 = in_15;
```



```

wire [7:0] out_7 = in_3;
wire [7:0] out_11 = in_7;
wire [7:0] out_15 = in_11;

//write back transposed state array to aesr
wire [127:0] shiftrw_result = {out_0, out_1, out_2, out_3, out_4, out_5,
out_6, out_7, out_8, out_9, out_10, out_11, out_12, out_13, out_14,
out_15};

assign aesr = shiftrw_result;
}

```

In the ShiftRow operation, the input AES register is split up into individual byte size wires labeled `in_0` to `in_15`. Each byte is assigned to the output wires labeled `out_0` to `out_15` based on the transposition shown in the C reference. Finally, `out_0` to `out_15` are concatenated into a contiguous 128-bit word and is written back to the AES register file.

It is often useful to take advantage of PERL expressions in the TIE description. The use of PERL is a great convenience, allowing for more compact code that is less error-prone. In the previous TIE example, 16 individual lines of TIE code were used to split the AESR operand into 16 individual bytes.

```

//split aesr into individual bytes
wire [7:0] in_0 = aesr[127:120];
wire [7:0] in_1 = aesr[119:112];
wire [7:0] in_2 = aesr[111:104];
wire [7:0] in_3 = aesr[103:96];
wire [7:0] in_4 = aesr[95:88];
wire [7:0] in_5 = aesr[87:80];
wire [7:0] in_6 = aesr[79:72];
wire [7:0] in_7 = aesr[71:64];
wire [7:0] in_8 = aesr[63:56];
wire [7:0] in_9 = aesr[55:48];
wire [7:0] in_10 = aesr[47:40];
wire [7:0] in_11 = aesr[39:32];
wire [7:0] in_12 = aesr[31:24];
wire [7:0] in_13 = aesr[23:16];
wire [7:0] in_14 = aesr[15:8];
wire [7:0] in_15 = aesr[7:0];

```

Not only is this tedious, but it is easy to make mistakes when defining the specific bit locations of each byte of the AESR register. This same TIE code can be simplified using PERL as shown below.

```

//split aesr into individual bytes
;for ($i=0;$i<16;$i++) {
;    $hibit = 127 - $i*8;
;    $lobit = $hibit - 7;
;    wire [7:0] in_`$i` = aesr[`${hibit}`:`${lobit}`];
;}

```

Note that a semi-colon precedes PERL expressions. The TIE compiler pre-processes the TIE and evaluates the PERL expressions prior to compiling it. PERL variables inside TIE expressions (i.e., '\$i') are evaluated and replaced at compile time. PERL expressions are used throughout the upcoming TIE examples to simplify repetitive TIE expressions.

5.2 The SubBytes Transformation

The SubBytes transformation is a substitution transformation that performs a bitwise substitution for each byte in the state array using a fixed 256-byte table defined by the AES standard. The C code for the SubBytes transformation is shown below.

```
void subbytes(char input[], char output[])
{
    unsigned int i;
    for (i=0;i<16;i++)
        output[i] = sub_table[(input[i])];
    return ;
}
```

The code above would require a byte load, a table lookup, followed by a store for each byte of the state array. The table lookup is composed of a load operation in which the effective address is determined by adding the loaded byte value with the base address of the `sub_table` array. This transformation requires 4 operations per byte for a total of 64 operations when done completely in software.

This transformation can be optimized using TIE tables. TIE tables are “hard-wired” inside the processor, allowing very efficient access to the data in the tables; however, the data cannot be dynamically changed. As opposed to tables in memory, TIE tables are accessible in an instruction’s execution stage and do not require memory accesses. Moreover, a single instruction can access many TIE tables simultaneously.

The TIE description below shows how TIE tables are used to optimize the SubBytes transformation. Note the definition of `sub_table` at the beginning of the TIE example (only partial data is shown). The TIE table is defined as containing 256 elements and each element is 8 bits in width.

```
//partial data only
table sub_table 8 256 {8'h63, 8'h7c, 8'h77,
//...
//<snip> remainder of 256-entry table described here
}

operation SUBBYTES {inout aes aesr} {}
{
//split aesr into individual bytes
;for ($i=0;$i<16;$i++) {
;    $hibit = 127 - $i*8;
;    $lobit = $hibit - 7;
    wire [7:0] in_`$i` = aesr[`$hibit`:`$lobit`];
;}

//byte substitution using sub_table
;for ($i=0;$i<16;$i++) {
```

```

        wire [7:0] out_`$i` = sub_table[in_`$i`];
    };

    //combine bytes to 128 bit word and write to aesr
    wire [127:0] subbytes_result = {
    ;for ($i=0;$i<15;$i++) {
        out_`$i`,
    ;}

        out_15};

    assign aesr = subbytes_result;
}

```

The TIE example of the SUBBYTES instruction shows how the 16-byte substitutions can be performed in parallel with a single TIE instruction. Like the TIE example for the SHIFTRW instruction, the first set of PERL expressions is used to split the 128-bit `aesr` operand into individual bytes. Each byte is then used as an offset into the TIE table. Finally, the substituted bytes are combined and written back to the `aesr` operand.

5.3 The AddRoundKey Transformation

The AddRoundKey transformation is a permutation operation that performs a bitwise Galois field addition with bytes of the key schedule. This is equivalent to performing a simple bitwise XOR operation on all 128 bits of the state array with all 128 bits of the key schedule.

```

void roundkey(char input[], char keyschedule[], char output[])
{
    unsigned int i;
    for (i=0;i<16;i++)
        output[i] = (input[i] ^ keyschedule[i]);
    return ;
}

```

The C implementation above can be optimized to perform the XOR operations on the native 32-bit word of the Xtensa processor. In this case, four loads are required to read the state array, four loads are required to read all 128 bits of the key schedule, four XOR operations, and four stores are required to write out the results. Therefore, 16 operations are required when the transform is done in software. A TIE instruction can provide significantly better performance by loading the 128-bit key schedule from memory and performing the bitwise XOR with a state array held in the AES register. The TIE description for the ROUNDKEY instruction is shown below.

```

operation ROUNDKEY {inout aes aesr, inout AR ars}
    {out VAddr, in MemDataIn128}
{
    assign VAddr=ars;
    assign ars = ars + 16;
    wire [127:0] keyschedule = MemDataIn128;

    wire [127:0] roundkey_result = aesr ^ keyschedule;

    assign aesr = roundkey_result;
}

```

The `ROUNDKEY` instruction takes two operands: `aesr` and `ars`. This instruction accesses two separate register files. Recall that the `aesr` operand indexes the 128-bit AES register file. The `ars` operand specifies the register in the Xtensa processor's base AR register file. Prior to executing this instruction, `ars` is loaded with the effective address of the key schedule. Once a key from the key schedule is loaded, it is XORed with the `aesr` operand, and finally written to the AES register file. In preparation for the next `ROUNDKEY` instruction, the `ars` operand is post incremented (by 16) to the address of the next key in the key schedule.

5.4 The MixColumns Transformation

The MixColumns transformation is a permutation of the state array that is performed on each of the four columns in the state array. Each column, consisting of four bytes is multiplied with a constant 4x4 matrix. The result of the matrix multiplication becomes the transformed column of the state array. This is shown in the following matrix equation.

$$\begin{bmatrix} out0 \\ out1 \\ out2 \\ out3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} in0 \\ in1 \\ in2 \\ in3 \end{bmatrix}$$

This gets expanded to four equations for each byte of the column:

$$out0 = (2 \bullet in0) \oplus (3 \bullet in1) \oplus (1 \bullet in2) \oplus (1 \bullet in3)$$

$$out1 = (1 \bullet in0) \oplus (2 \bullet in1) \oplus (3 \bullet in2) \oplus (1 \bullet in3)$$

$$out2 = (1 \bullet in0) \oplus (1 \bullet in1) \oplus (2 \bullet in2) \oplus (3 \bullet in3)$$

$$out3 = (3 \bullet in0) \oplus (1 \bullet in1) \oplus (1 \bullet in2) \oplus (2 \bullet in3)$$

In the equations above, the \oplus (Galois field add) operator is achieved by a simple XOR operation. Explaining how \bullet (Galois field multiplication) operator works is somewhat more complex. Rather than going into detail on how Galois Field multiplication works, one can take a more pragmatic approach. Notice in the equations above, that multiplication occurs only with the factors 1, 2, or 3. This fact makes the multiplication quite simple.

As would be expected, multiplication with the constant 1 results in an output identical to the input. Just like binary arithmetic, multiplication with the constant 2 is achieved by a left shift. The only difference is that the result is XORed with value 0x1B if the most significant bit of the output is 1. This is described by the `xtimes2` function in the C code below.

```
char xtimes2 (unsigned char in)
{
    unsigned char out;
    out = in << 1;
    if (in < 0x80) return (out);
    else return (out ^ 0x1B);
}
```

In binary arithmetic, you can calculate multiplication with 3 as follows:

$$X \cdot 3 = (X \cdot 1) + (X \cdot 2)$$

Similarly, Galois field multiplication with the constant 3 is accomplished by XORing the input byte (times 1) with the result of `xtimes2`.

$$X \bullet 3 = (X \bullet 1) \oplus (X \bullet 2)$$

The C code for the MixColumns transformation is shown below.

```

void
mixcolumn(
    char    input[],
    char    output[])
{
    int i;
    char x2[16];
    char x3[16];

    //Generate all x2 & x3 arrays first
    for (i=0; i<16; i++)
        {
            x2[i]=xtimes2(input[i]);
            x3[i]=x2[i]^input[i];
        }

    //Calculate column 0
    output[0] = x2[0]^x3[1]^input[2]^input[3] ;
    output[1] = input[0]^x2[1]^x3[2]^input[3] ;
    output[2] = input[0]^input[1]^x2[2]^x3[3] ;
    output[3] = x3[0]^input[1]^input[2]^x2[3] ;

    //Calculate column 1
    output[4] = x2[4]^x3[5]^input[6]^input[7] ;
    output[5] = input[4]^x2[5]^x3[6]^input[7] ;
    output[6] = input[4]^input[5]^x2[6]^x3[7] ;
    output[7] = x3[4]^input[5]^input[6]^x2[7] ;

    //Calculate column 2
    output[8] = x2[8]^x3[9]^input[10]^input[11] ;
    output[9] = input[8]^x2[9]^x3[10]^input[11] ;
    output[10] = input[8]^input[9]^x2[10]^x3[11] ;
    output[11] = x3[8]^input[9]^input[10]^x2[11] ;

    //Calculate column 3
    output[12] = x2[12]^x3[13]^input[14]^input[15] ;
    output[13] = input[12]^x2[13]^x3[14]^input[15] ;
    output[14] = input[12]^input[13]^x2[14]^x3[15] ;
    output[15] = x3[12]^input[13]^input[14]^x2[15] ;

    return ;
}

```

The `mixcolumn` function initially takes each byte of the state array and multiplies it with constants 2 and 3, storing the results in arrays `x2` and `x3` respectively. Then, the new columns are calculated as described in the previous equations.

Of all the AES cipher transforms, this transform requires the most processor cycles when done in pure software. When compiled, the MixColumns transform consists of approximately

250 individual operations consisting of byte loads/stores, XORs, comparisons, and shifts. A single TIE instruction can be designed to perform all of these operations in a single clock cycle. The TIE description for the MIXCOLUMN instruction is shown below.

```

operation MIXCOLUMN {inout aes aesr} {}
{
  //split aesr into individual bytes
  ;for ($i=0;$i<16;$i++) {
  ;   $hibit = 127 - $i*8;
  ;   $lobit = $hibit - 7;
    wire [7:0] in_`$i` = aesr[`$hibit`:`$lobit`];
  ;}

  // calculate x2 and x3 arrays
  ;for ($i=0;$i<16;$i++) {
    wire [7:0] x2_`$i`a = {in_`$i`[6:0],1'b0};
    wire [7:0] x2_`$i`b = x2_`$i`a ^ 8'h1b;
    wire [7:0] x2_`$i` = (in_`$i`[7])? x2_`$i`b : x2_`$i`a;
    wire [7:0] x3_`$i` = x2_`$i` ^ in_`$i`;
  ;}

  // calculate new column from previous column and x2, x3 arrays
  ;for ($i=0;$i<16;$i=$i+4) {
    wire [7:0] mixout_`$i` = x2_`$i` ^ x3_`$i+1` ^
                          in_`$i+2` ^ in_`$i+3`;
    wire [7:0] mixout_`$i+1` = in_`$i` ^ x2_`$i+1` ^
                          x3_`$i+2` ^ in_`$i+3`;
    wire [7:0] mixout_`$i+2` = in_`$i` ^ in_`$i+1`
                          ^ x2_`$i+2` ^ x3_`$i+3`;
    wire [7:0] mixout_`$i+3` = x3_`$i` ^ in_`$i+1` ^
                          in_`$i+2` ^ x2_`$i+3`;
  ;}

  // combine bytes into 128 bit word
    wire [127:0] mixout = {
  ;for ($i=0;$i<15;$i++) {
    mixout_`$i`,
  ;}
    mixout_15};

    assign aesr = mixout;
  }

```

The TIE code above shows that the state array is initially split into bytes, then the x2 and x3 wire groups are calculated just as the x2 and x3 arrays were calculated in the C reference. Once the data in x2 and x3 wire groups are available, the new columns are calculated in parallel. Finally, the bytes are combined to a 128-bit word and written back to the AES register file.

Alternatively, hard TIE tables can be used to generate multiply tables rather than calculate x^2 and x^3 wire groups on the fly. In this case, two 256-entry TIE tables would be required for Galois Field Multiplication by factors of 2 and 3, using the input byte as an index to the table.

5.5 Programming the AES Engine

Writing software that takes advantage of TIE is straightforward. As mentioned earlier, it is not necessary to write any assembly since the C compiler automatically supports all TIE extensions. The C compiler handles allocation of variables of `AES` type to the AES register file. The transform instructions that were previously described in TIE are automatically made available to software by means of intrinsic functions. Therefore, the code required for the AES cipher is quite simple. The C code below performs encryption on a single block of plaintext.

```

aes *p_in, *p_out, my_state;
char *p_key;

//initialize aes pointers to plaintext & ciphertext buffers
p_in = (aes*)plaintext_buf;
p_out = (aes*)ciphertext_buf;

//initialize char pointer to start of keyschedule
p_key = keyschedule;

//loads plaintext from memory to AES register file
my_state = *p_in;

ROUNDKEY(my_state, p_key);           //encryption round 0

for (i=1; i<10; i++)                //encryption rounds 1-9
{
    SUBBYTES(my_state);
    SHIFTRW(my_state);
    MIXCOLUMNS(my_state);
    ROUNDKEY(my_state, p_key);
}

SUBBYTES(my_state);                 //encryption round 10
SHIFTRW(my_state);
ROUNDKEY(my_state, p_key);          //ciphertext now in state

//stores ciphertext from AES register file to memory
*p_out = my_state;

```

Prior to encryption, some initialization is performed. Assuming that the encryption key has already been expanded into the key schedule, the pointer `p_key` is initialized with the base address of the key schedule. Also, the pointer `p_in` is initialized with the address of plaintext input buffer.

Encryption begins with assigning plaintext from the `p_in` pointer into the variable `my_state`. The compiler allocates a register in the AES register file to the `my_state` variable and infers the `aes_li` instruction to load the plaintext data into the allocated register.

The `my_state` variable is used as an argument to the AES intrinsic functions: `SUBBYTES`, `SHIFTRW`, `MIXCOLUMNS`, and `ROUNDKEY`. The compiler then automatically translates the `my_state` variable to the appropriate register operand for each intrinsic function. After all 10 encryption rounds have completed, the ciphertext result is written to the ciphertext output buffer.

6. Key Expansion

AES encryption and decryption are iterative algorithms that make use of a different key for each iteration or “round”. The particular key used in a given round is called the “round key”. The set of round keys that are used to encrypt or decrypt a block is called the “key schedule”, and each key in the schedule is derived from the initial key. Prior to encryption or decryption, the 128-bit key is expanded into 10 additional round keys which are stored in memory. Each iteration of the algorithm reads a round key from the key schedule in memory and uses it in the calculations for that round.

Previous sections have treated this key schedule in memory as a given, preferring to describe the AES-ECB algorithm separate from the process of the key expansion which precedes it. As we look toward the XTS extensions to AES, which adds a second encryption key for the so-called twiddle factor, we return to the topic of key expansion.

The example TIE design provided with this Application Note uses a TIE operation to accelerate the generation of the key schedule prior to encryption or decryption. Without a custom operation, many shifts, byte masks, and bit-wise C/C++ operations would be required to generate each key. Using TIE, we can generate each key in the key schedule using a single processor instruction. We will now discuss the key schedule generation algorithm for the purposes of describing the TIE that implements it. For more detailed algorithmic information about the key schedule, see IEEE Specification 1619-2007.

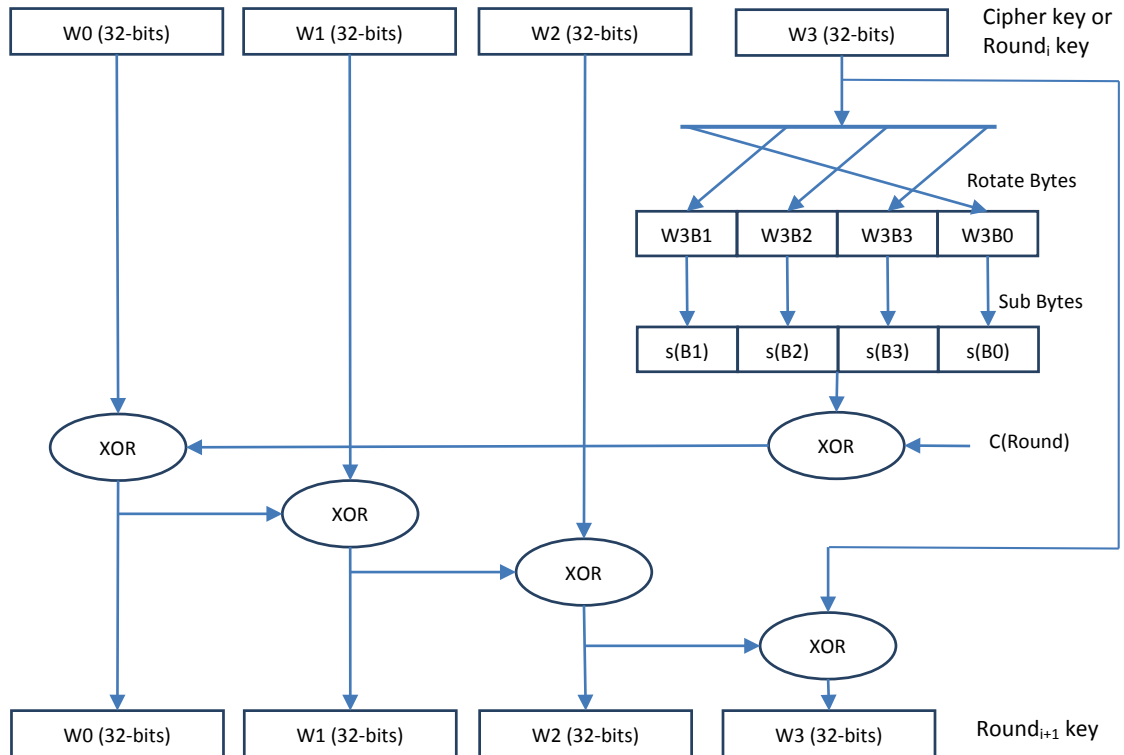


FIGURE 5: 128-BIT AES KEY EXPANSION

Figure 5 depicts one iteration of AES key expansion, which is repeated to generate each key to be used in one round of the AES encryption or decryption algorithm. The cipher key, or a key generated for a previous round (Round_i key), is shown at the top of the diagram, in big-endian byte order. Operations on the 128-bit key are performed on four 32-bit words, W0-W3.

The lowest-order word is transformed in a two-step process that first rotates the bytes in W3 (Rotate Bytes), and then substitutes each byte with another byte from a lookup table (Sub Bytes). The resulting 32-bit pattern is XORed bitwise with a constant, C(Round) that is related to the round (iteration number) of the key being generated. A network of XOR operations produces each 32-bit slice of the key for the next round (Round_{i+1} key).

When the cipher key changes, the new key schedule is calculated and the keys are stored in local memory for use during each round of encryption. A TIE operation implementing one iteration of the key expansion of Figure 5 can perform the byte transformations as well as store the resulting round key to memory. The `KeyExpansion` TIE operation accomplishes this.

```

immediate_range immKey 0 240 16

operation KeyExpansion {in AR ars, in AR *addr, in immKey offset,
                        inout aes cKeyH}
                        {out VAddr, out MemDataOut128}
{
  // break the current round key into bytes
  ;for(my$i=0; $i<16; $i++) {
  ;my $h=127-$i*8;
  ;my $l=$h-7;
    wire [7:0] key_`$i` = cKeyH[`$h`:`$l`];
  ;}

  // 1. Compute the first word of the round key, which involves the
  // Rotate bytes and SubBytes transformations

  wire [7:0] rotbyte12 = key_13;
  wire [7:0] rotbyte13 = key_14;
  wire [7:0] rotbyte14 = key_15;
  wire [7:0] rotbyte15 = key_12;

  wire [7:0] subbyte12 = sub_table[rotbyte12];
  wire [7:0] subbyte13 = sub_table[rotbyte13];
  wire [7:0] subbyte14 = sub_table[rotbyte14];
  wire [7:0] subbyte15 = sub_table[rotbyte15];

  // 2. Calculation of C(Round) constant
  // AR ars register contains round #

  wire z0 = ars[7:0]==8'd1;
  wire z1 = ars[7:0]==8'd2;
  wire z2 = ars[7:0]==8'd3;
  wire z3 = ars[7:0]==8'd4;
  wire z4 = ars[7:0]==8'd5;
  wire z5 = ars[7:0]==8'd6;
  wire z6 = ars[7:0]==8'd7;
}

```

```

wire z7 = ars[7:0]==8'd8;
wire z8 = ars[7:0]==8'd9;
wire z9 = ars[7:0]==8'd10;
wire z10 = ars[7:0] == 8'd0;

// Select appropriate constant by round #
// (constants defined by XTS-AES specification)
wire [7:0] rcon = TIEsel(z0, 8'h01, z1, 8'h02, z2, 8'h04, z3, 8'h08,
                        z4, 8'h10, z5, 8'h20, z6, 8'h40, z7, 8'h80,
                        z8, 8'h1b, z9, 8'h36);

// 3. now XOR the round constant with the SubBytes result
// The upper three bytes of all the round constant values are always
// zero, so no XOR is needed

wire [31:0] temp0 = { subbyte12,
                    subbyte13,
                    subbyte14,
                    subbyte15 ^ rcon
                    };

// 4. XOR with highest-order word
wire [31:0] W0 = {key_0, key_1, key_2, key_3} ^ temp0;

// Subsequent words of the round key are more straightforward.
wire [31:0] W1 = {key_4, key_5, key_6, key_7} ^ W0;
wire [31:0] W2 = {key_8, key_9, key_10, key_11} ^ W1;
wire [31:0] W3 = {key_12, key_13, key_14, key_15} ^ W2;

wire [127:0] new_key = {W0, W1, W2, W3};

// update the AES register with new key value
assign cKeyH = z10? cKeyH : new_key;

// 5. write the new round key to memory
assign VAddr = addr + offset;
assign MemDataOut128 = z10? cKeyH : new_key;
}

```

The KeyExpansion operation begins with the Rotate Bytes and SubBytes transformations (1). The Rotate Bytes transformation is simply a byte-lane transfer of bits which produces no gates. The SubByte transformation is accomplished with the de-referencing of the same `sub_bytes[]` array of constants that is used by the SubBytes transformation of the AES encryption algorithm. Second, the constant `C(Round)` defined by the XTS-AES standard is determined (2) by decoding the round number passed to KeyExpansion in an AR register and selecting the corresponding constant with a built-in `TIEsel` function. Third, the constant thus selected is XORed with the results of the SubByte transformation (3), followed by the cascading XOR operations (4) for each 32-bit slice of the key shown at the bottom of Figure 5.

Now that round key $i+1$ has been calculated, it is stored back to the AES register where key i was passed into the KeyExpansion operation. In addition, the new key is stored to memory (5), using indexed addressing. The effective store address is calculated by adding the immediate offset operand to KeyExpansion to an address passed through an AR register.

```
aes_encrypt_key128(unsigned char *key, unsigned char *keysched_base)
{
    aes cKey;
    aes* p_cKey;

    // load cipher key into 128-bit AES register "current Key cKey"
    p_cKey = (aes*) key;
    cKey = *p_cKey;

    // Each of these operations performs a 16-byte store to memory
    // cKey is also updated to contain the Round(i+1) key
    //
    // operands: round #, store address, store byte offset, Round(i) key
    KeyExpansion(0, keysched_base, 0, cKey);
    KeyExpansion(1, keysched_base, 16, cKey);
    KeyExpansion(2, keysched_base, 32, cKey);
    KeyExpansion(3, keysched_base, 48, cKey);
    KeyExpansion(4, keysched_base, 64, cKey);
    KeyExpansion(5, keysched_base, 80, cKey);
    KeyExpansion(6, keysched_base, 96, cKey);
    KeyExpansion(7, keysched_base, 112, cKey);
    KeyExpansion(8, keysched_base, 128, cKey);
    KeyExpansion(9, keysched_base, 144, cKey);
    KeyExpansion(10, keysched_base, 160, cKey);
}
```

The reference C program requires more than 300 cycles to do 128-bit key expansion. With the KeyExpansion TIE operation, the same can be done with 35 cycles, including function overhead in both cases.

7. XTS Extensions

So far we have discussed the most basic AES encryption, AES-ECB. The XTS extension to AES adds a second key and provides an algorithm for storage devices to include logical block information as part of the encryption scheme. The second encryption key, key2, is used to encrypt the logical block address t , and this encrypted key T is added (XORed) before, and after the AES algorithm encryption of each plain text block.

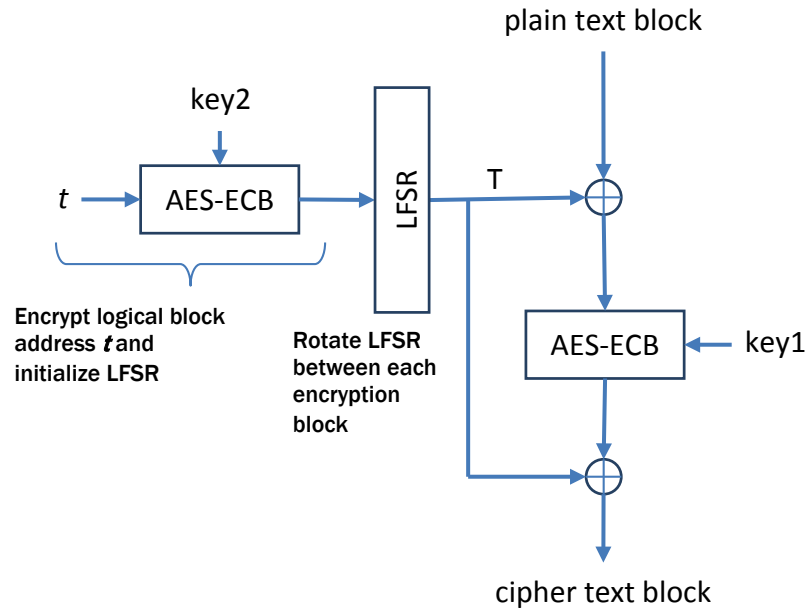


FIGURE 6: XTS-AES EXTENSIONS

In addition, the encrypted tweak value T is rotated through a linear feedback shift register (LFSR) between each plain text block, to alter T pseudo randomly for each block. In this way, each cipher block will be different, even if the input plain text was the same.

The shift-update of the LFSR in between blocks is particularly well suited for a new TIE operation. Updating the LFSR data requires a 128-bit shift, conditionally adding a constant to the lowest byte to implement the polynomial $(x^{128}+x^7+x^2+x^1+1)$, as defined in the XTS-AES specification. In C, this requires several byte shifts, masks and bit-wise XOR to accomplish.

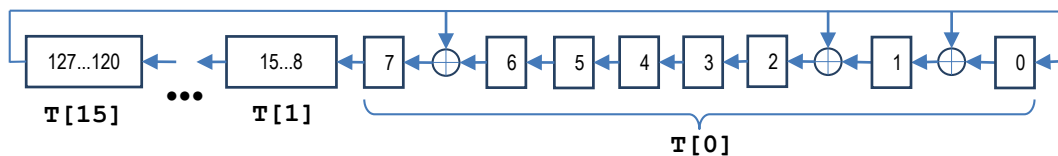


FIGURE 7: 128-BIT LFSR FOR T KEY UPDATE

```
Cin = 0;
for (j=0; j<16 ;j++) {
    Cout = (T[j]>>7) & 1; //T[] is byte data
```

```

T[j] = ((T[j]<<1)+Cin) & 0xff;
Cin = Cout;
}
if (Cout)
T[0] ^= 0x87;

```

The actual software algorithm in the example workspace (called `gf_mulx()` in `xts.c`) is more efficient than the loop above, but nonetheless it requires about 35 cycles to execute.

In the example implementation, the encoded twiddle `T` is kept in a 128-bit TIE state register called `cKeyL`. Using a single TIE operation, the entire LFSR update can be accomplished in a single cycle.

```

state cKeyL 128

operation tupdateE {} {inout cKeyL}
{
    wire [7:0] s_lobyte = {cKeyL[6:0], 1'b0};
    wire [7:0] lobyte = cKeyL[127] ? (s_lobyte ^ 8'h87) : s_lobyte;
    assign cKeyL = {cKeyL[126:7], lobyte};
}

```

In addition to the LFSR update operation on the encoded twiddle `T`, we also need operations to load and store the 128-bit TIE state register that holds `T`. Unlike the load, store, and move operations related to the AES register file that can be automatically generated by the TIE compiler, TIE state registers have no associated C-type, and so specific operations must be written to initialize and update them. In the case of the 128-bit `T` register (`cKeyL`), we have written an indexed load instruction.

```

immediate_range immKey 0 240 16

operation KeyLSet {in AR *addr, in immKey offset}
                {out VAddr, in MemDataIn128, out cKeyL}
{
    assign VAddr = addr + offset;
    wire [127:0] key = MemDataIn128;
    assign cKeyL = key;
}

```

When writing load and store operations in TIE, the “`*addr`” syntax alerts the C compiler’s type checking that this AR operand is holding a pointer type. The `offset` operand is defined as an immediate range, which can only take values of 0 through 240 in steps of 16. This immediate value is added to the address to calculate the effective load address, and the 16-byte granularity of the offset is because of the 128-bit (16-byte) data being loaded. The C/C++ compiler (and the assembler) will produce a compile-time error if this immediate value does not conform to the `immKey` range definition. The 16-bit step size of the `offset` immediate operand allows this immediate field in the `KeyLSet` instruction to be encoded using only 4 bits.

8. Optimization: Instruction Combining

In the previous sections, AES transformations and created individual TIE instructions to perform these transformations were discussed. With the exception of the `ROUNDKEY` instruction, each took a single cycle to execute. We may increase the encryption performance by merging the individual TIE instructions into a combined TIE instruction and issue the single instruction for each encryption round. Our clock frequency must be able to accommodate all levels of logic in a single cycle, of course.

Recall that the encryption rounds use all the transformations with the exception of round 0 and round 10. In round 0, only the RoundKey transformation is performed. Round 10 does not use the MixColumns transformation. In the last section we saw that the encrypted twiddle value T will be XORed before the first round and after the last round. Therefore, three separate instructions are required to support all encryption rounds.

The three TIE instructions created are:

- ◆ `XTS.AESF` (“AES First”) instruction that XOR’s the plain text block with the T value, then performs the RoundKey transformation.
- ◆ `XTS.AESR` (“AES Repeat”) instruction that performs all transformations, but does nothing with the T value.
- ◆ `XTS.AESL` (“AES Last”) instruction that performs all transformations except the MixColumns transformation, then XOR’s the result with the T value, before storing the cipher text block.

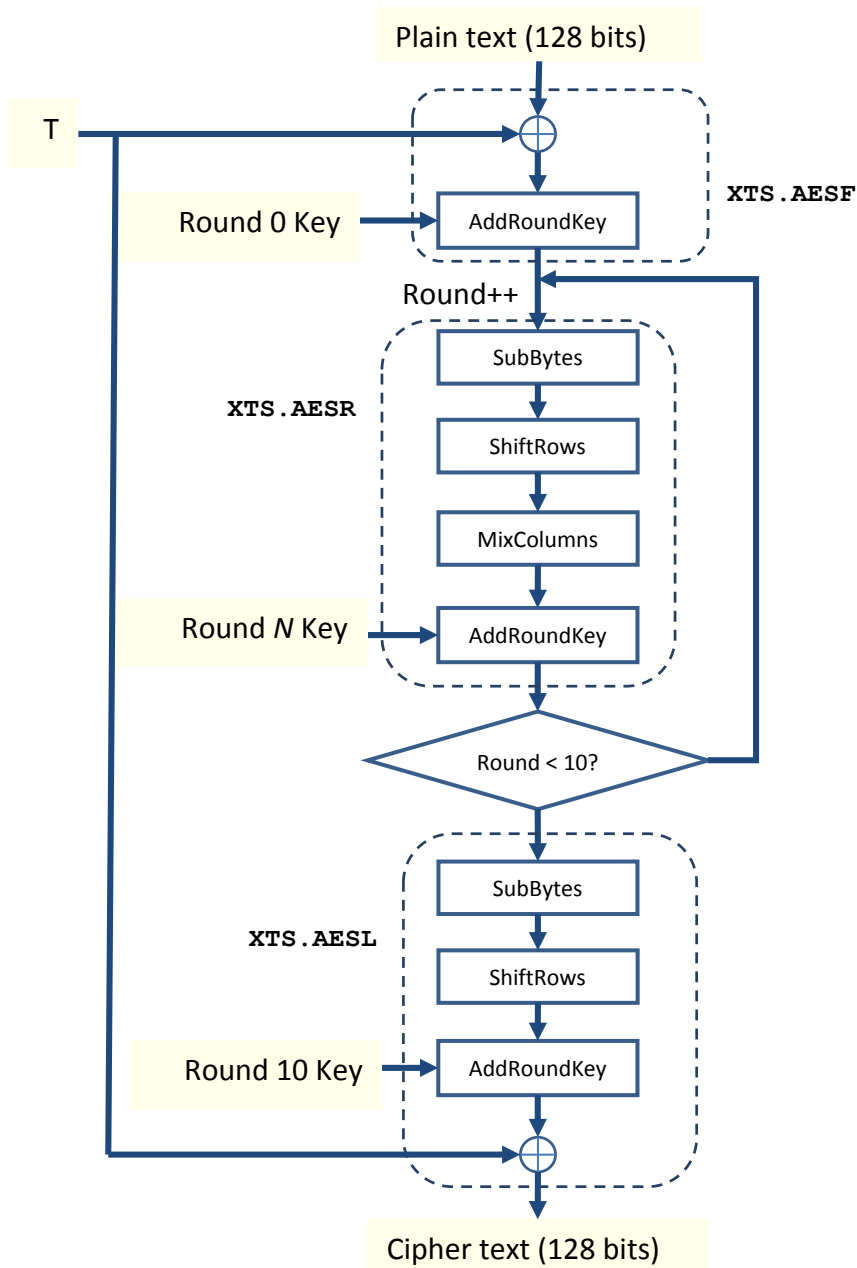


FIGURE 8: COMBINED XTS-AES INSTRUCTIONS

It would be somewhat wasteful to create separate logic for each instruction, because they share a great deal of logic. In TIE, we can use the same logic to implement multiple operations through the use of the TIE semantic description. Because XTS.AESF and XTS.AESL perform a subset of the operation found in the XTS.AESR instructions, logic can be shared amongst the instructions, and the various transformations needed for each round can be switched in and out of the data path based on which round we are executing. Data flow through the semantic block is selectively routed based upon the instruction issued. This is shown in the following abridged TIE code.

```
// pseudo code describing the shared semantic, implementing
// the three operations for XTS-AES 128-bit encryption
```



```

// Read the example workspace TIE source for actual implementation

operation XTS.AESF {inout AR *ars, inout aes aesrr}
    {out VAddr, in MemDataIn128, in cKeyL}
operation XTS.AESR {inout AR *ars, inout aes aesrr}
    {out VAddr, in MemDataIn128}
operation XTS.AESL {inout AR *ars, inout aes aesrr}
    {out VAddr, in MemDataIn128, in cKeyL}

semantic aes_encode {XTS.AESF, XTS.AESR, XTS.AESL}
{

//0. Split aesrr into bytes
;for ($i=0;$i<16;$i++) {
; $hibit = 127 - $i*8;
; $lobit = $hibit - 7;
wire [7:0] in_`$i` = aesrr[`$hibit`:`$lobit`];
;}

//1. LD 128bit key from key schedule in memory and auto-increment
assign VAddr=ars;
wire [127:0] Key = MemDataIn128;

// auto-increment address register to point to next round key
assign ars = ars+16;

//2. Subbytes transformation
;for ($i=0;$i<16;$i++) {
wire [7:0] subout_`$i` = sub_table[in_`$i`];
;}

//3. Shiftrow transformation
wire [7:0] shiftout_0 = subout_0;
wire [7:0] shiftout_4 = subout_4;
wire [7:0] shiftout_8 = subout_8;
wire [7:0] shiftout_12 = subout_12;
wire [7:0] shiftout_1 = subout_5;
wire [7:0] shiftout_5 = subout_9;
// ...
// <snip> several more shiftout byte assignments
wire [7:0] shiftout_15 = subout_11;

// collect bytes into 128-bit wire for later use
wire [127:0] shiftout = {
;for ($i=0;$i<15;$i++) {
shiftout_`$i`,
;}
shiftout_15};

```

```

//4. Mixcolumn transformation (XTS.AESR only)

// calculate x2 and x3 arrays
;for ($i=0;$i<16;$i++) {
wire [7:0] x2_`$i`a = {shiftout_`$i`[6:0],1'b0};
// ...
// <snip> more x2, x3 terms calculated from shiftout bytes

// calculate new column from previous column and x2, x3 arrays
;for ($i=0;$i<16;$i=$i+4) {
wire [7:0] mixout_`$i` = x2_`$i` ^ x3_`$i+1` ^
    shiftout_`$i+2` ^ shiftout_`$i+3`;
// ...
// <snip more mixout_* bytes calculated from x2, x3, shiftout bytes
;}

// combine bytes into 128 bit wire
wire [127:0] mixout = {
;for ($i=0;$i<15;$i++) {
    mixout_`$i`,
;}
    mixout_15};

//5. Multiple Roundkey transformations, for each different instruction
wire [127:0] rF = cKeyL ^ aesrr;
wire [127:0] rL = shiftout ^ cKeyL;
wire [127:0] rR = mixout;

// Data path routing per instruction (multiplex by opcode)
wire [127:0] aesr_out = TIEsel(
                                XTS.AESF, rF,
                                XTS.AESR, rR,
                                XTS.AESL, rL
                                );

// now finally add the Key and store result to aes register
assign aesrr = aesr_out ^ Key;

} // end of encode semantic

```

The `aes_encode` semantic block contains the TIE expressions for all four AES encryption transformations described in earlier sections. (However, the actual TIE for each transformation is omitted in the abridged TIE description). Logic from the `SUBBYTES`, `SHIFTRW`, and `MIXCOLUMN` instructions are daisy chained. A slightly different

ROUNDKEY-like function is created for each of the three instructions. During the execution of the `XTS.AESF` instruction, the ROUNDKEY logic is fed with the `aesr` register operand, bypassing the logic for the other transforms. When the `XTS.AESL` instruction is executed, the MIXCOLUMN logic is bypassed. During execution of the `XTS.AESR` instruction, none of the logic is bypassed. The encrypted twiddle value `T` is included in the ROUNDKEY function of both `XTS.AESF` and `XTS.AESL`, because it is added only in the first and last encryption rounds. Finally, the opcode of the operation being executed is used to select which of the three ROUNDKEY results are appropriate for the operation. In the TIE source, the name of the operation can be used on the right side of an assignment, implying a wire that is true only when the named operation is being executed. This is how several operations, three in this case, can share a TIE semantic.

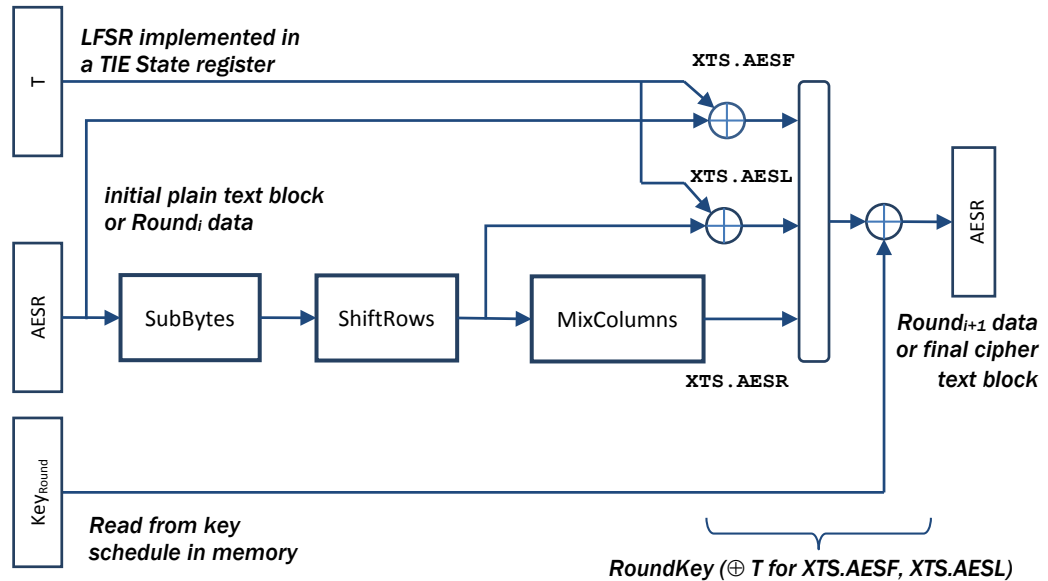


FIGURE 9: BLOCK DIAGRAM OF THE AES_ENCODE TIE SEMANTIC

The code example below shows how the AES encryption code is simplified when using the combined instructions.

```
encrypt128(aes *p_in, aes *p_out, aes *p_key) {
// encrypted twiddle value T already set up in cKeyL TIE register
// key schedule already set up starting at p_key address in memory

// load plain text block into AES register
aes my_state = *p_in;

// T is an implicit input operand to AESF
XTS_AESF(p_key, my_state); //encryption round 0

// p_key pointer is auto-incremented by each XTS_AES* operation
// to step through the key schedule in memory
XTS_AESR(p_key, my_state); //encryption round 1 - 9
XTS_AESR(p_key, my_state);
```

```
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);
XTS_AESR(p_key, my_state);

// T is an implicit input operand to AESL
XTS_AESL(p_key, my_state); //encryption round 10

// store cipher text block
*p_out = my_state;

// Rotate LFSR, ready for next block
  tupdateE();
}
```

Each encrypt instruction must be scheduled to execute in two cycles to support the loading of the key schedule from memory. As we will see in the next section, the AES engine executing the code above is capable of encrypting a block in about 25 cycles per block.

9. Integrating XTS Extensions into AES

Now we have all the ingredients for the XTS-AES algorithm.

- ◆ The AES register file, for handling 128-bit data types
- ◆ An efficient basic AES-ECB implementation
- ◆ An instruction for performing the key expansion step in one cycle per key
- ◆ A 128-bit TIE register for storing the encrypted twiddle value T, and a way to load it
- ◆ A single-cycle LFSR rotation of the T register (move to next in pseudorandom sequence)

With these elements, we can write the pseudo code for the whole XTS-AES algorithm shown in Figure 6 as follows:

```
// 1. perform key expansion on key2, prior to twiddle encryption
char zeros[16] __attribute__((aligned(16))) =
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
keyLSet( &zeros[0], 0);

aes_encrypt_key128( (char*) key2, (aes*) key2_sched );

// 2. Encrypt Logical Block Address (LBA) using key2 schedule
aes_encrypt( (aes*) my_LBA, (aes*) my_T, (aes*) key2_sched);
// result is at my_T in memory; load it into the T register
keyLSet( my_T, 0);

// 3. Create key1 schedule
aes_encrypt_key128( (char*) key1, (aes*) key1_sched );

// 4. Iterate over all plain text blocks
for() { // each plain text block

    aes_encrypt((aes*)p_plain, (aes*)p_cipher, (aes*)key1_sched);
    p_plain++; p_cipher++;

    // 5. Rotate T LFSR
    tupdateE();
}
}
```

In the first step (1), the key schedule for key2 is generated and stored in memory with the `aes_encrypt_key128()` routine. Note that the `cKeyL` register, which holds the encrypted twiddle value T during ordinary XTS-AES passes, must be zero so that the T value does not affect basic AES-ECB encryption. Once the key schedule is generated, the logical block address is encrypted and stored at the location `(*my_T)` in memory (step 2). The encrypted T value is then loaded into the `cKeyL` register using `keyLSet()`. The next step (3) is to generate the key schedule for key1, the basic AES encryption key for the block. That key is used by each iteration of the loop in Step 4, which encrypts one block of plain text into cipher text. The last step (5) within the loop is to rotate the T register through the LFSR logic, pseudo randomly creating a new T value for the next iteration.

10. Timing and Clock Period Considerations

Xtensa processors may be configured with a 5- or 7-stage instruction pipeline. The TIE source for the XTS-AES functions described in this Application Note are only appropriate for a processor with a 5-stage instruction pipeline, though a similar TIE design can be created for a 7-stage pipeline. Tensilica names the five instruction pipeline stages I (Instruction fetch), R (Register file reference), E (Execution), M (Memory), and W (Write-back). Every instruction passes through these five stages, which are fully pipelined so there are generally five different instructions in the five stages. In the Instruction Fetch I-stage, the program counter is used to fetch the next instruction from instruction memory. Register files are de-referenced during the R-stage, and the operands are routed to the inputs of logic implementing different operations. Typically, ALU functions occur in the E-stage of the pipeline, or, in the case of data loads or stores, the effective address of the memory operation is calculated in the E-stage. Data memory loads occur in the M-stage of the pipeline, and finally the W-stage is when the operation's outputs are written to register files, or stored in data memory. Any instruction that has not yet made it to the W-stage may be killed, and the instruction pipeline flushed, for a number of reasons including cache miss, jump taken, and exceptions. Conversely, those instructions that have made it to W-stage are "committed" and will not be killed under any circumstances.

Operations designed and described in TIE do not have to adhere strictly to the purpose of each pipeline stage. The E-stage is the earliest that custom operations may place custom logic, since the I-stage and R-stage are virtually the same for every instruction that uses register operations. But the operation does not have to place logic in the E-stage, or in just one stage of the pipeline. Indeed it is possible to create operations that need several cycles to execute, writing their results to register files or memory later than the W-stage. The

specification of which logic will be performed during each pipeline stage is called the operation's *schedule*.

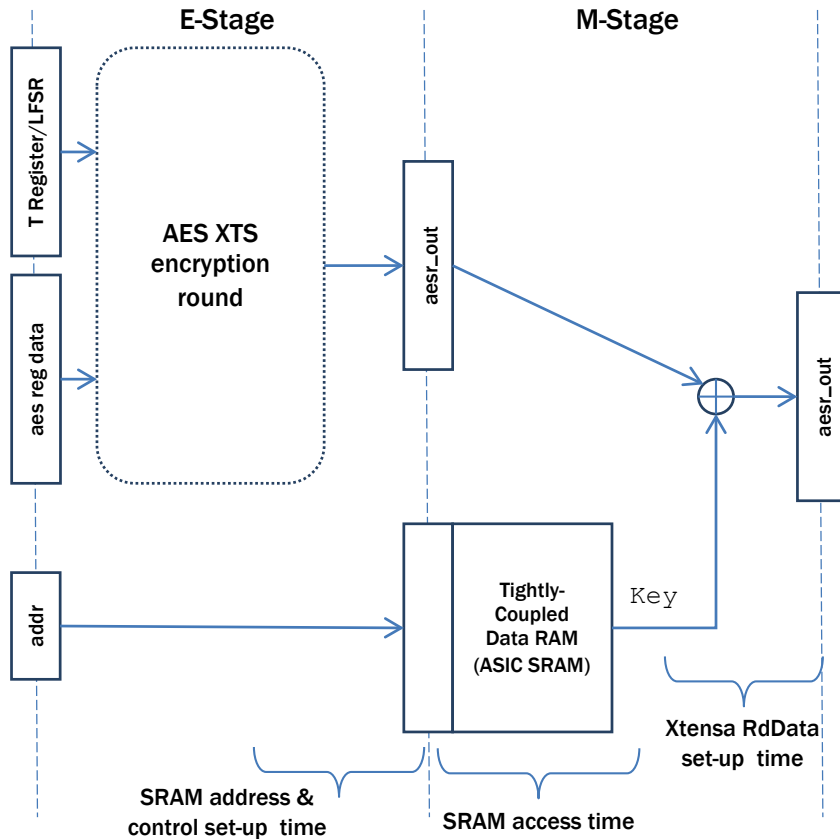


FIGURE 10: DEFAULT AES OPERATION SCHEDULE

The default schedule for of the `aes_encrypt` semantic makes it an E-stage operation, except for the use of `MemDataIn128`. Because `MemDataIn128` is not available until M-stage (“use Mstage”) it forces the update of the output AES register to occur at the end of M-stage (“def Mstage”). This default schedule, therefore, creates a two-cycle operation as shown in Figure 10 – the reading of the AES register input operand and the writing of AES register output are two cycles apart. The iterative process of XTS-AES encryption, with its 10 rounds, has operand interlocks on the AES register. A new XTS.AES* instruction cannot start every cycle due to this interlock; new instructions can start only every other cycle. This results in about 26 total cycles needed to encrypt one 128-bit block, as shown in this pipeline diagram in Figure 11.

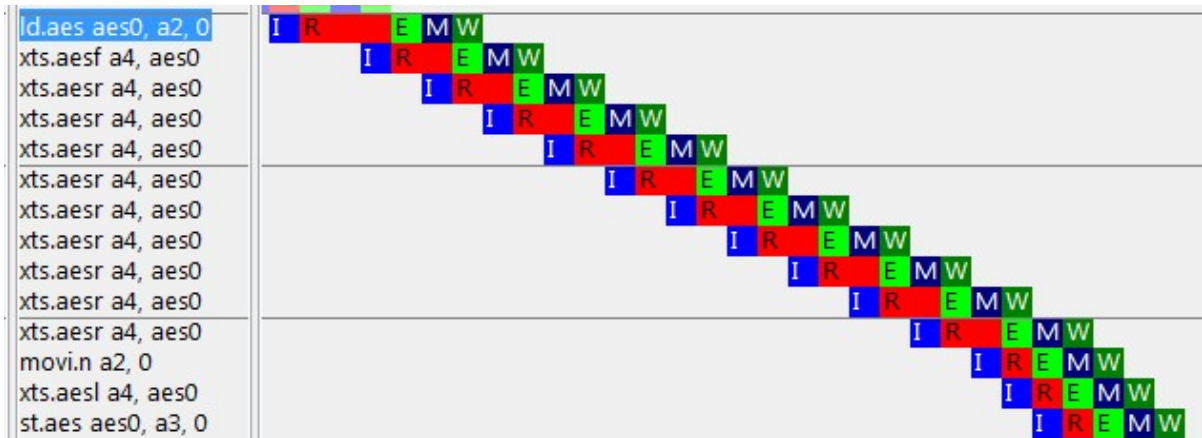


FIGURE 11: PIPELINE DIAGRAM OF AES ENCRYPTION, DEFAULT OPERATION SCHEDULE

An improved pipeline schedule might be realized if the clock period can accommodate all of the XTS-AES logic within the M-stage of the pipeline. Normally, Tensilica urges caution when using MemDataIn load data during the M-stage, because the read data path from local data memory is often the longest path in the Xtensa/memory subsystem. Adding any levels of logic to the MemDataIn path can lower maximum frequencies. But as the figure shows, we are already adding a level of XOR gates to the read data paths given the default schedule. These XOR gates are the very last logic level before writing the result into the AES register, and if our cycle time can accommodate this level of XOR, then maybe we can create a better schedule.

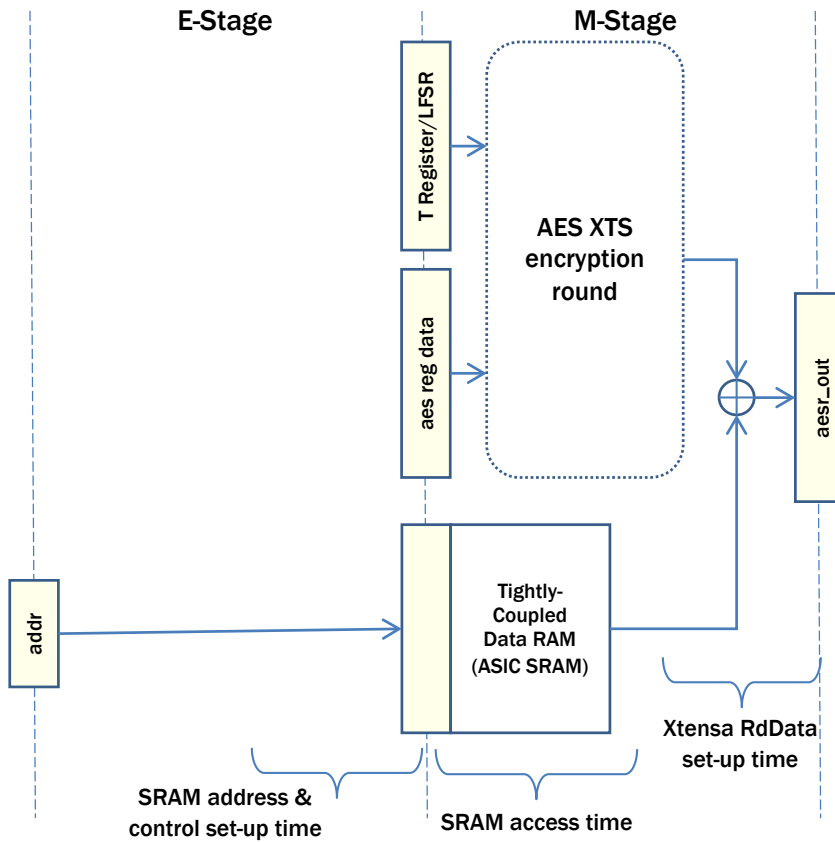


FIGURE 12: IMPROVED AES OPERATION SCHEDULE

A schedule that reads and writes the AES register every cycle, instead of every other cycle, could reduce the cycles required for the encryption, at the possible cost of a slightly longer clock period. The logic of this improved schedule is depicted in Figure 12. Whether this actually takes less absolute time is a matter for investigation in a real system with a particular ASIC technology target.

The improved schedule is depicted in Figure 13.

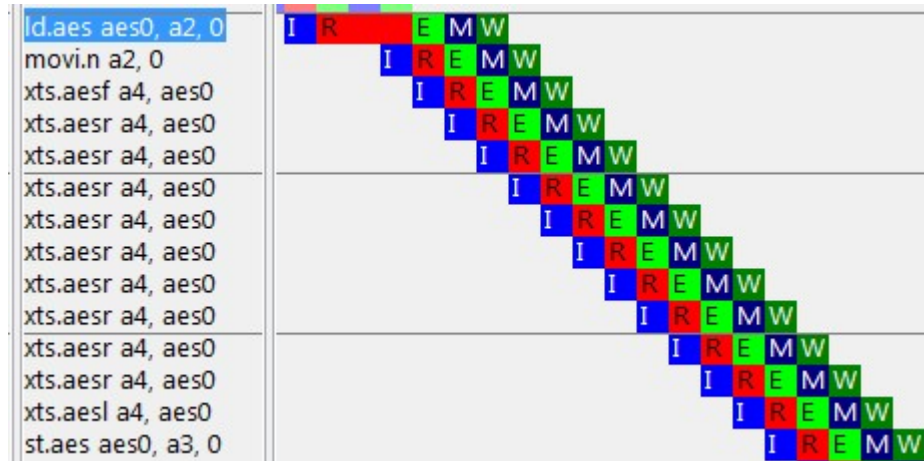


FIGURE 13: PIPELINE DIAGRAM OF AES ENCRYPTION, IMPROVED OPERATION SCHEDULE

10.1 AES Decryption

The previous sections covered TIE acceleration to support XTS-AES encryption. Due to the high degree of symmetry in the AES cipher, the decryption process is quite similar. In decryption, most transformations are the inverse of the transforms used in encryption. Other than the AES register file and its load/store/move instructions, most of the encryption logic cannot be directly leveraged in decryption. However, the same techniques covered can be applied to develop a new set of instructions to support decryption and achieve throughput equivalent to encryption. The AES test bench described in the next section includes both AES encryption and decryption.

11. Conclusion

The Xtensa processor offers significant advantages in almost any application. The Xtensa architecture combines a powerful general-purpose instruction set design with a unique configuration and extension process. These are used together to handle general-purpose processing as well as the most compute intensive requirements in system design today, including security algorithms.

In this application note, we looked at the AES cipher transformations and created TIE instructions to accelerate the transformations. We also made use of 128-bit data loads and stores utilizing a 128-bit local data memory interface and C data types to support them. We used a 128-bit TIE state register to store the encrypted twiddle factor T and used a custom operation to perform the LFSR shift-update of T in a single cycle.

In the following section, the example Xtensa Xplorer workspace is described. This workspace, contains the TIE and C source for the XTS-AES TIE design described in this Application Note. After attaching the TIE to an Xtensa processor, Xplorer gives TIE Compiler-generated estimates of the gate counts for each TIE element. Performance profiling can be used to obtain detailed cycle counts of the C-language reference code, and the same algorithms implemented using the TIE design.

TABLE 1: SUMMARY OF THE BENEFITS OF XTS-AES EXTENSIONS

Algorithm Feature	C-Language Reference Cycles	TIE Design Cycles	TIE Gate Estimate (#'s of NAND2)
AES Register File Storage +load, store, move ops	-	-	5,931 +641
T-Register Storage + T-register load operation	-	-	3,748 +317
Key Expansion† (for 10-round encoding)	313	48	7,187
Encrypt 128-bit block†	794	25	11,535
Decrypt 128-bit block†	798	36	13,900
T-LFSR Shift	38	1	19
Total			47,526 gates‡

Table notes:

†Cycle counts for encryption/decryption/key expansion include all subroutine overhead including call/return

‡Total gate estimate includes approximately 3,800 gate equivalents for operation decode, operand multiplexing, etc.

We can see in the table that encryption and decryption are speeded up by tremendously by a relatively modest increase in gates.

The hardware design challenge of adding instructions to a processor was greatly simplified by the TIE compiler. The TIE compiler takes a text description of custom operations and automatically extends the instruction set of the Xtensa processor. These instructions look and act like any other core instruction in that they are correct for any stalls, interrupts, or exceptions that may occur in a pipelined processor. The TIE compiler also extends the software development tools so that designers can quickly test and benchmark code using the TIE instructions without having to write any assembly code.

An XTS-AES engine based on the Xtensa processor can provide performance that rivals most hardware solutions, but retains the ease of design and flexibility found in software based solutions.

Example XTS-AES Test Bench

A workspace for Xtensa Xplorer integrated development environment (IDE) demonstrating these techniques is generally distributed with this application note. The file to import into Xplorer is called `xts_aes_example.xws`. This file is a working example of XTS-AES encryption and decryption as described, but the actual working source code differs from the pseudo code in the preceding sections. There are two basic reasons for this difference.

First, the example performs all of the 128-bit encryption, decryption, and key-handling operations described, with two 128-bit keys that are two halves of a 256-bit key. But it also has operations for 256-bit key expansion, or two halves of a 512-bit key. Some of the test vectors included in the example workspace have 512-bit keys, and in this case the 512-bit key is treated as two 256-bit keys, and only the lower 128 bits of each expanded 256-bit half key is actually used in the example.

The second way the pseudo code is simplified compared to the example is the fully functional test bench, with the supporting data structures and procedure call structure. This example has been derived from previous work by Brian Gladman of Worcester UK, and it has been adapted to verify, and demonstrate our example implementation of XTS-AES.

11.1 Unpacking Example and Attaching TIE to a Processor

To unpack the example, follow these steps:

1. Save the `xts_aes_example.xws` workspace file from this distribution to your local disk.
2. Inside of Xtensa Xplorer IDE, select File from the main menu, then select Import.
3. When the import wizard opens, expand the Xtensa Xplorer folder if necessary, and then select Import Xtensa Xplorer Workspace. Click Next.
4. Enter the location of the `xts_aes_example.xws` file on your local disk, or click Browse and use the file browser to locate it. After the file name is correct, click Next.
5. Now click Select All and Next for each additional screen that appears. Do not press Finish until or unless Next is not available. In other words, be sure to import all Projects, Launches, and TIE files present in the `.xws` file.

The `xts_aes_example.xws` workspace does not contain a processor configuration, it is a “source only” workspace. The TIE source from the workspace can be attached to any little-endian processor configuration with a 128-bit load/store data width. If you do not have a processor configuration to use with the example, you can use one of the example processor configurations included with the Xplorer samples. The configuration called `xpres_sample_cfg` can be installed in your workspace by following these instructions:

1. From Xplorer’s main menu, select Help and then Browse Samples....
2. Locate and select the `xtt_sample2.xws` workspace on the list of sample workspaces. Click OK.
3. On the first dialog, just click Next (do not import the SAD project).
4. On the next dialog, select `xpres_sample_cfg`, then click Finish. This will install the processor configuration in your workspace.

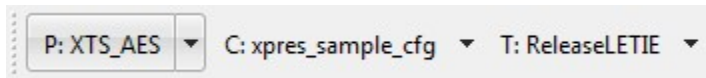
Whether you use the sample processor configuration or your own, you must attach the TIE source for the XTS-AES operations to the configuration and compile it before you can run the example. To build the TIE example:

1. Locate the processor configuration in Xplorer's System Overview window, right-click the configuration and select Attach TIE and TDB files to configuration from the pop-up sub-menu.
2. The dialog box that opens will list the TIE files available in your workspace. Locate aes_xts.tie and select it by checking the box next to it. Click Finish.
3. Now right-click on the configuration again in the System Overview, and select Compile TDK for configuration. The TIE will compile in a few moments. You should only get one message, a warning that the Tie Compiler has automatically created load, store, and more operations for the AES register file. If you get errors, check to be sure you have followed all instructions above, and that your processor configuration has 128-bit load/store data width.

11.2 Running the Example XTS-AES Application

To build the example application, use the Active Set Selectors in Xplorer. Set the project **P** selector to **XTS_AES**, and set the core **C** selector to your processor, after attaching and building the TIE source. The example workspace provides several build targets, to demonstrate the algorithm with and without using TIE operations. Set the target **T** selector to **ReleaseLETIE** to build the application using the TIE operations, and set it to **ReleaseLE** to run the C reference version.

For example, when using the example xpres_sample_cfg processor configuration, and building for the TIE example, the Xplorer working set will look like this:



To build the software for the C reference, set **T:ReleaseLE**.

Once the software is built, use the run_xts launch that is included with the workspace to run, debug, and profile the application. This launch will correctly set the command-line options to run the example on the test vector files found in the testvals folder of the XTS_AES software project. As it is distributed, there are 7 files of test vectors, with a total of several thousand block encryptions and decryptions. It takes several minutes to run on ISS in cycle-accurate mode.