Hardware Visualization of the Advanced Encryption Standard (AES) Algorithm

Mostafa I. Soliman and Ghada Y. Abozaid

Computer & System Section, Electrical Engineering Department, Faculty of Engineering at Aswan, South Valley University, Egypt

Abstract — Advanced Encryption Standard (AES) was issued as FIPS by NIST as a successor to data encryption standard (DES) algorithms. The applications of the AES are wide including any sensitive data requires cryptographic protection before communication or storage. Thus, many students are interested in learning how the AES is working. One of the inherent limitations of traditional methods for explaining/teaching the AES is their lack of visualization of dynamic behavior. This paper visualizes the dynamic behavior of the AES algorithm by simulating its hardware structure using HADES “Hamburg Design System”. HADES is an object-oriented and java language based WWW-enabled framework for interactive simulation of digital systems. Moreover, this paper shows how the AES pipeline can be implemented by hardware components of HADES. It visualizes how the data passes through different pipeline stages and shows the effect of each stage on the input data.

Index Terms — Advanced Encryption Standard (AES), cryptography, HADES, hardware visualization, pipelining.

I. INTRODUCTION

The Data Encryption Standard (DES) [1] had served as an important cryptographic algorithm for over two decades. However, the growth of computing power during that time had compromised the security of that algorithm. There was considerable evidence that it was time to replace DES with a new standard. In October 2000, the National Institute of Standards and Technology (NIST) selected the Rijndael algorithm [2] as the new encryption standard to replace the current DES algorithm. The Advanced Encryption Standard (AES) specifies the Rijndael algorithm. AES can process data blocks of 128 bits, using keys with lengths of 128, 192, and 256 bits. However, Rijndael was designed to handle additional block sizes and key lengths, which are not adopted in the AES.

According to NIST, AES is efficient, elegant, and secure. AES has the potential to maintain security well beyond twenty years due to the significantly larger key sizes than DES had [3]. Thus, AES was accepted as a FIPS (Federal Information Processing Standards) in November 2001 to protect electronic data [4]. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext, as shown in Fig. 1. The applications of the AES are wide; this standard can be used by agencies when an agency determines that sensitive information requires cryptographic protection. The AES algorithm may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc.

Since AES was accepted as a FIPS standard, there have been many different hardware implementations for ASIC and FPGA to improve its performance. Some ASIC implementations are found in references [5]-[7]. These references mainly focus on area efficient implementation of the AES algorithm using Sbox (byte substitution) optimizations. Reference [7] (the first ASIC implementation of the Rijndael on silicon) presented an AES encryption chip architecture and discussed the design optimizations. References [8]-[10] presented different pipelined implementations of the AES algorithm as well as the design decisions and the area optimizations that lead to a low area and high throughput AES encryption processor. FPGA implementations of the AES algorithm are found in references [11]-[14]. References [15]-[16] presented another approach to improve the performance of AES on 32-bit processors using instruction set extensions.

Learning how the AES is working is very attractive because of the practical importance of this standard algorithm. Visualization of dynamic behavior has proven being one of the key technologies for improving the learning/teaching further and for exploiting modern equipment. For example, RaVi (an educational tool for visualization of computer architecture) [17] visualizes complex system behavior which cannot be explained with the animation capabilities of Power-point and similar tools. RaVi included educational units for visualizing the dynamics of micro-programmed MIPS machines, pipelined MIPS-machines, dynamic out-of order instruction scheduling techniques and cache protocols. Visualizing the dynamic behavior of the AES algorithm presents an exciting opportunity to deeply understand how the algorithm works and to understand the hardware
structure of each step of the algorithm. None of the previously published researches present a visualization of the dynamic behavior of AES. This paper presents simple and regular hardware visualization for the AES. To explore the dynamic behavior of AES and the hardware implementation, HADES is used as a visualization tool to build the AES algorithm.

HADES, the “Hamburg Design System” [18], is a portable visual simulation Java-based visual simulation environment. While HADES can be used for any type of discrete event based simulation, the main focus is on the simulation of digital logic systems on gate-level up to system level, including efficient hardware-software co-simulation. It aims at providing interactive multimedia components capable of displaying dynamic behavior. With Hades, systems can be designed on a standard PC running the Java virtual machine. Unlike some other systems, where simulation models have to be written in a specialized internal language, all HADES simulation models are directly written in Java. This gives the designer full access to a modern object-oriented programming language with a rich class library.

This paper is organized as follows. The specification of the AES algorithm is presented in Section II. Section III describes how to use HADES components to build the main functions of the AES algorithm. Visualization of pipelined implementation for AES is presented in Section IV. Section V shows the results of testing our visualization tool with a numerical example. Finally Section VI concludes this paper.

### II. THE SPECIFICATION OF THE AES ALGORITHM

The AES algorithm is a symmetric block cipher that can convert data to an unintelligible form (encryption) and convert the data back into its original form (decryption). The input and output for the AES algorithm each consist of sequences of 128 bits. Each 128-bit is called a block, which can be represented as 16-byte; \( in_0 \) through \( in_{15} \) for input and \( out_0 \) through \( out_{15} \) for output. Moreover, the cipher key for the AES algorithm is a sequence of 128, 192 or 256 bits. Internally, the AES algorithm’s operations are performed on a two-dimensional (2-D) array of bytes called the State array, see Fig. 2. The State array consists of four rows of bytes, each containing \( Nb \) bytes, where \( Nb \) is the block length divided by 32 (the word size). At the start of encrypting/decrypting a block of data, the input (the 1-D array of bytes; \( in_0, in_1, \ldots in_{15} \)) is copied into the State array (2-D array of bytes) according to the scheme:

\[
\text{State}[\text{row}, \text{col}] = \text{in}[\text{row} + 4*\text{col}] \text{ for } 0 \leq \text{row} < 4 \text{ and } 0 \leq \text{col} < \text{Nb}
\]

Then all encryption/decryption operations are performed on State array. The final value of the State array is copied to the output (1-D array of bytes; \( out_0, out_1, \ldots out_{15} \)) as follows: \( out[\text{row} + 4*\text{col}] = \text{State}[\text{row}, \text{col}] \) for \( 0 \leq \text{row} < 4 \) and \( 0 \leq \text{col} < \text{Nb} \). The four bytes in each column of the State array form 32-bit words, where the row number (row) provides an index for the four bytes within each word.

For the AES algorithm, the length of the input block, the output block, and the State array is 128 bits. This is represented by \( Nb = 4 \), which reflects the number of 32-bit words (number of columns) in the State array. Besides, the length of the key, \( K \), is 128, 192, or 256 bits. The key

### Visualizing the AES Algorithm

![AES Algorithm Diagram](image)

**Fig. 2.** Processing AES algorithm.
length is represented by $N_k = 4, 6, \text{or } 8$, which reflects the number of 32-bit words (number of columns) in the key. For the AES algorithm, the 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 $N_r$, where $N_r$ equals 10, 12 or 14 when $N_k$ equal 4, 6, or 8, respectively.

For both encryption and decryption, the AES algorithm uses a round function that is composed of four different byte-oriented transformations as Fig. 3 shows:

1) byte substitution using a substitution table (Sbox),
2) shifting rows of the State array by different offsets (ShiftRows),
3) mixing the data within each column of the State array (MixColumn), and
4) adding (XORing) a round key to the State array.

III. HADES IMPLEMENTATION OF AES

AES algorithm is an iterated block cipher supporting a variable key length of 128, 192 or 256 bits. The algorithm consists of three distinct phases (see Fig. 3). In the first phase, an initial addition (XOR) is performed between the input data (plaintext) and the given key. A number of standard rounds are performed in the second phase, which represents the kernel of the algorithm and consumes most of the execution time. The number of these standard rounds depends on the key size; nine for 128-bits, eleven for 192-bits, or thirteen for 256-bits. Each standard round includes four fundamental algebraic function transformations on arrays of bytes. The third phase of the AES algorithm represents the final round of the algorithm, which is similar to the standard round, except that it does not have MixColumn operation. In this paper, 128-bit key length AES algorithm is visualized, which has 10 rounds (nine standard rounds plus a final round). The initial key is expanded to generate the round keys, each of size equal to block length (128-bit). Each of the ten rounds of the algorithm receives a new round key from the key schedule module (see Fig. 3). Decryption is performed by applying the inverse transformations of the round functions. The following sections show the implementations of functions needed for AES using HADES tool. After that these functions are connected together to construct the AES algorithm (see Section IV).

A. HADES Implementation of Sbox

Sbox is a non-linear substitution table used in several byte substitution transformations and in the key expansion routine to perform a one-for-one substitution of a byte value. It is the primary source of nonlinearity in the AES algorithm. It takes each byte in the 128-bit State and computes its multiplicative inverse in GF($2^8$), followed by a single stage of systematic bit mixing. The multiplicative inverse in GF($2^8$) is computed using the extended Euclidean algorithm [19], which is essentially the Euclidean algorithm for integers, applied to polynomials in GF($2^8$). The computational complexity of the Euclidean algorithm and the non-constant number of iterations required to compute the multiplicative inverse leads to use a lookup table for computing the multiplicative inverses. Otherwise, there is uncertainty regarding the number of clock cycles required to compute an inverse, as the number of steps through the Euclidean algorithm for a given input is dependent on the input byte. The average number is certainly greater than one clock cycle. Thus, a

![Fig. 3. Round function of AES.](image-url)
lookup table is the optimum software/hardware approach for implementing Sbox (substitution box).

Since there are only 256 representations of one byte, all the byte substitution results can be calculated (see Table I). Sbox is implemented as a look-up table. HADES implementation of a look-up table can be done using predefined memory component called ROM. Obviously, the size of the ROM component required to implement Sbox is $256 \times 8$ (256 bytes). Fig. 4 shows the ROM component (named Sbox) storing the content of Table I. Moreover, Fig. 4 shows an example of using Sbox; when applying 01H (01 hexadecimal value) on the input of the ROM component, it produces 7CH as the Sbox output.

B. HADES Implementation of ShiftRows

ShiftRows transformation function processes the State array by cyclically shifting the last three rows over different numbers of offsets. In other words, for rows number one, two, and three, the offsets used are one, two, and three bytes respectively, however, the first row is not shifted (see Fig. 5).

The hardware implementation of the ShiftRows transformation function does not need to perform any arithmetic or logic operations. However, wire connections are only needed (see Fig. 6). Since next transformation

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

Fig. 5. The effect of Shiftrows on the State array.

Fig. 6. HADES implementation of ShiftRows.

function (MixColumn) processes the columns of the State array, it is preferable to deal with columns in ShiftRows instead of rows. As Fig. 5 shows, each column of the input State array is distributed to all columns of the output State array.

Fig. 6 also shows an example of applying ShiftRows transformation function on input data \{00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d, 0e, 0f\} and produces output \{00, 05, 0a, 0f, 04, 09, 03, 08, 0d, 02, 07, 0c, 01, 06, 0b\}; all data are in hexadecimal.

C. HADES Implementation of MixColumns

MixColumns transformation function takes the columns of the State array and mixes their data (independently of one another) to produce new columns. The columns of the State array are considered as polynomials over GF(2^8) and multiplied modulo \((x^4 + 1)\) with a fixed polynomial \(a(x)\), given by \(a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}\). The MixColumns transformation for column \(c\) can be written as follows:

\[
\begin{align*}
\text{s}_0' &= (02\text{H} \cdot \text{s}_{0,c}) \oplus (03\text{H} \cdot \text{s}_{1,c}) \oplus (01\text{H} \cdot \text{s}_{2,c}) \oplus (01\text{H} \cdot \text{s}_{3,c}) \\
\text{s}_1' &= (01\text{H} \cdot \text{s}_{0,c}) \oplus (02\text{H} \cdot \text{s}_{1,c}) \oplus (03\text{H} \cdot \text{s}_{2,c}) \oplus (01\text{H} \cdot \text{s}_{3,c}) \\
\text{s}_2' &= (01\text{H} \cdot \text{s}_{0,c}) \oplus (01\text{H} \cdot \text{s}_{1,c}) \oplus (02\text{H} \cdot \text{s}_{2,c}) \oplus (03\text{H} \cdot \text{s}_{3,c}) \\
\text{s}_3' &= (03\text{H} \cdot \text{s}_{0,c}) \oplus (01\text{H} \cdot \text{s}_{1,c}) \oplus (01\text{H} \cdot \text{s}_{2,c}) \oplus (02\text{H} \cdot \text{s}_{3,c})
\end{align*}
\]

The above polynomials include three types of multiplication (01H\cdot s_{0,c}, 02H\cdot s_{0,c}, and 03H\cdot s_{0,c}). Obviously, any byte (binary polynomial \(b_7b_6b_5b_4b_3b_2b_1b_0\)) times 01H results in the same value (\(b_7b_6b_5b_4b_3b_2b_1b_0\)). Multiplying a byte by 02H can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with 1bH. If \(b_7 = 0\), after shifting the result \(b_7b_6b_5b_4b_3b_2b_1b_0\) is already in reduced form. Otherwise, the reduction is accomplished by XORing the shifted byte with the polynomial 1bH. This operation on bytes is denoted by xtime. Fig. 7 shows the hardware implementation of xtime using HADES components.

As Fig. 7 shows, the predefined HADES component NxXOR is used to perform bit-wise XOR between 8-bit polynomials. The Expander component is used in HADES to expand an input bus to its individual bits. As opposite to expander, MergeBits component merges the input bits to an n-bit bus. The shift left of input byte \(b_7b_6b_5b_4b_3b_2b_1b_0\) is implemented by applying a shifted input \(b_7b_6b_5b_4b_3b_2b_1b_0\) to the input of the MergeBits component, see Fig. 7. Then the most significant bit of the input byte \(b_7\) is ANDed with 1bH using N1AND component, which performs bit-wise AND between \{00011011\} and \{01101101\}. This is means the output of the N1AND is 1bH if \(b_7\) is one, otherwise 0H. NxXOR performs bit-wise XOR between the input byte shifted by one \(b_7b_6b_5b_4b_3b_2b_1b_0\) and the result of the

![Fig. 7. HADES implementation of xtime.](image)

![Fig. 8. HADES implementation of MixColumns.](image)

![Fig. 9. HADES implementation of KeyAddition for a column.](image)
N1\text{AND} (1bH or 00). As an example, when the input is d4H the outputs of the expander, MergeBits, N1\text{AND}, and N\times\text{XOR} components shown in Fig. 7 are d4H, a8H, 1bH, and b3H, respectively.

Multiplication by \{03\} can be implemented by applying \text{xtime} on the input byte and then adding (XORing) the result with the input byte. Fig. 8 shows the implementation of the MixColumn function using HADES components. It also gives an example, when input column of data is (d4H, b6H, 5dH, 30H) then the output after MixColumns is (04H, 66H, 81H, e5H).

D. HADES Implementation of KeyAddition

The addition in the AES algorithm is performed with the XOR operation. KeyAddition implemented in HADES using N\times\text{XOR} component, where four N\times\text{XOR} components are needed to perform a bit-wise XORing between a column in the \text{State} array and corresponding column of the round key (see Fig. 9). Thus, to implement a complete key addition, 16 N\times\text{XOR} components are needed.

E. HADES Implementation of the KeyExpansion

The AES algorithm takes the cipher key, \(K\), and performs the key expansion routine (KeyExpansion) to generate a key schedule. The KeyExpansion generates a total of \(N_b(N_r + 1)\) words: the algorithm requires an initial set of \(N_b\) words, and each of the \(N_r\) rounds requires \(N_b\) words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted \(w[i]\), where \(0 \leq i < N_b(N_r + 1)\). The expansion of the input key into the key schedule proceeds according to the following pseudo code.

\begin{verbatim}
Begin
  i = 0
  while(i < N_k)
    w[i]=word(key[4*i]:key[4*i+3])
    i = i+1
  end while
  i = N_k
  while(i < N_b*(N_r+1))
    t = w[i-1]
    if (i mod N_k = 0)
      t = SubWord(RotWord(t))XOR Rcon[i/N_k]
    end if
    w[i] = w[i-N_k] XOR t
    i = i + 1
  End while
End
\end{verbatim}

The main functions used in the key schedule module are Rcon, RotWord, and SubWord. Rcon[\(i\)] function is the round constant word array. It contains the values given by \([x^i]\{00\},\{00\},\{00\}\), with \(x\) being powers of \(x\) (\(x\) is denoted as \{02\}) in the field GF(\(2^8\)) (see Table II). RotWord is function used in the KeyExpansion routine to

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{Round} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
\text{Rcon} & 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1b & 36 \\
\hline
\end{tabular}
\caption{The CONTENT of the RCON[Round]}\label{table:RCON}
\end{table}

Fig. 10. The implementation of the first round in the KeyExpansion routine.
perform a cyclic permutation on a four-byte word. SubWord function takes a four-byte input word and applies Sbox transformation to each of the four bytes to produce an output word.

The hardware implementation of the first round of the KeyExpansion routine is shown in Fig. 10. The key round starts with RotWord function, which takes the last column of key array shifted cyclically by one byte. As mentioned previously the implementation of the cyclic shift does not require logic gates but wire connections. After cyclic shifting, SubWord function is applied on this last column of the key array. The resulted column is added (XORed) with the round constant (Rcon[round], see Table II). Only the most significant byte is involved in this operation, which uses NxXOR component. This result is added (XORed) with the first column of the key in round $i$ to produce the first column of the key in round $i+1$. The first column of the key $i+1$ is added to the second column of the key in round $i$ to produce the second column of the key in round $i+1$, and so forth. Adding two columns is easily implemented using NxXOR component.

To clarify these operations, Fig. 10 shows a numerical example, when the input key is \{2b 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c\}, the output after applying first expansion round is \{a0 fa fe 17 88 54 2c b1 23 a3 39 39 2a 6c 76 05\}, all data are in hexadecimal.

IV. VISUALIZATION OF AES ALGORITHM

The overall encryption algorithm can be implemented by connecting the main functions blocks explained in Section III. The rolled pipeline AES is implemented using HADES components. The main advantage of using rolled pipeline is the reduction on the required hardware, however, the execution time increases. The effects of the functions of the AES algorithm on the plaintext can be monitored each round. Both encryption algorithm and key schedule modules are shown in Fig. 11. Moreover, the input/output data is visible for each function block.

The algorithm starts from the left with the key addition transformation. Block of 128 $2 \times 1$ multiplexers is built using HADES components to choose between input plaintext and the feedback (State) from round output. Each step in the algorithm is placed and connected in order. Since the last round does not need MixColumn, 128 $2 \times 1$ multiplexes are used to select whether the output of the MixColumns (in case of rounds 1 through 9) or the output of ShiftRows (in case of the last round) to be added with key round. The outputs of the multiplexers are stored in 128-bit registers for synchronization.

The execution of the algorithm is controlled by a control unit. This control unit generates the control signals for multiplexer’s selectors and registers. The memory block is used to feed the AES hardware with the input data.

Fig. 11. The hardware implementation of the rolled AES algorithm on HADES platform.
This memory block can feed the hardware with a stream of 128-bit blocks of data. The content of the memory block can be easily changed, thus a student can see the encryption of any blocks of data.

V. TEST RESULTS

Many plaintexts were used to verify the hardware implementation of the AES algorithm. One of these plaintexts was that calculated in details in reference [4]. The specifications of this plaintext were as follow: input block = \{32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34\}, key = \{2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C\}, and 128-bit output = \{39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32\}, all are in hexadecimal. Rolled AES algorithm produces one 128-bit block of encrypted data every 11 cycles. One of the advantages of our hardware visualization is that the user can trace any change on the input data during the encryption process.

Fig. 12 shows the HADES waveform viewer, which displays the desired signals in the implementation. This waveform viewer clarifies the changes happened on the data (State array) at the end of each round. Besides, it shows the changes happened on the key during the key expansion process. Note that the signals labeled \(k_0, k_1, k_2\) and \(k_3\) constructs 128-bit key (four 32-bit words), while \(d_0, d_1, d_2\) and \(d_3\) are donated for 128-bit input data. The various values of key and data in each individual round are named \(k_{r0}, k_{r1}, k_{r2}, k_{r3}\) and \(d_{r0}, d_{r1}, d_{r2}, d_{r3}\), respectively, 32-bit each. The numbers (0 through 3) indicate the column number in the State array. All the signals values displayed in the waveform viewer in Fig. 12 are in hexadecimal format.

VI. CONCLUSION

Visualization of dynamic behavior has proven being one of the key technologies for improving the learning/teaching further and for exploiting modern equipment. This paper presents the visualization of the AES based on the hardware structure of the algorithm. The implementation of the pipelined AES architecture is visualized using HADES tool. It visualizes how the data passes through different pipeline stages and shows the effect of each stage on the input data. Thus, this tool can be used for explaining how the AES algorithm is working. HADES as a visualization tool can be used also to describe the hardware implementation of the AES algorithm. Moreover, the use of HADES enables the student to easily open any block of the implemented diagram to see its hardware contents and connections down to gate level.

ACKNOWLEDGMENT

The authors would like to express their gratitude to N. Hendrich for his technical assistance and helpful comments in solving some problems during using HADES.
REFERENCES


