CSEE4840 Project Report

Integer Vector Homomorphic Encryption Accelerator

Lanxiang Hu, Liqin Zhang, Enze Chen
Department of {Electrical Engineering, Computer Science}
Columbia University
New York, NY
{lh3116, lz2809, ec3576}@columbia.edu

CONTENTS

I Introduction 3

II System Block Diagram 3

III Theory 4
- III-A **Motivation** 4
- III-B **Formulation** 4
- III-C **Encryption and Key Switching** 4
- III-D **Decryption** 5
- III-E **Encrypted Domain Operations** 5
  - III-E1 **Addition** 5
  - III-E2 **Linear Transform** 5
  - III-E3 **Weighted Inner Product** 6
  - III-E4 **Polynomial** 6
  - III-E5 **Examples** 6

IV Design 7
- IV-A **Software** 7
  - IV-A1 **Matrix Library** 7
  - IV-A2 **Client Functions** 7
  - IV-A3 **Server Functions** 7
- IV-B **Hardware** 7
  - IV-B1 **Client-side Accelerator** 7
  - IV-B2 **Server-side Accelerator** 8
- IV-C **Hardware/Software Interface** 9
  - IV-C1 **Client-side Kernel** 9
  - IV-C2 **Server-side Kernel** 9
  - IV-C3 **Avalon Bus** 10

V Resource Budgets 10

VI Hardware Simulation 10
- VI-A **Vector Addition** 10
- VI-B **Linear Transformation** 11
- VI-C **Weighted Inner Product** 14
- VI-D **Bit Representation of Vector and Matrix** 17
- VI-E **Random and Noise Matrix Generation** 23

VII Implementation 25
- VII-A **Software: User Library** 25
  - VII-A1 **`mat.h`** 25
  - VII-A2 **`client_functions.c`** 32
  - VII-A3 **`server_functions.c`** 32
- VII-B **Software: Device Drivers and Kernel Code** 32

VII-B1 `key_switching.c/.h` . . . 32
VII-B2 `encrypted_domain.c/.h` . . . 38
VII-C **Software: Integer Vector Homomorphic Scheme Demonstration** . . . 49
VII-C1 `client_server.c` . . . 49
VII-D **Hardware: Key Switching Unit** . . . 53
VII-D1 **Top-level: `key_switching.sv`** . . . 53
VII-D2 `bit_repr_vector.sv` . . . 56
VII-D3 `bit_repr_matrix.sv` . . . 58
VII-D4 `get_random_matrix.sv` . . . 61
VII-D5 `get_noise_matrix.sv` . . . 64
VII-E **Hardware: Encrypted-Domain Computational Unit** . . . 66
VII-E1 **Top-level: `encrypted_domain.sv`** . . . 66
VII-E2 `vector_addition.sv` . . . 72
VII-E3 `linear_transform.svn` . . . 73
VII-E4 `weighted_inner_product.sv` . . . 74

References 78

I. Introduction

In the past few decades, it has witnessed evolution in cryptographic techniques and growing numbers of applications. In Zhou and Wornell's work [1], the fully homomorphic encryption scheme they proposed encrypts integer vectors to deliberately generate malleable ciphertext and to allow computation of arbitrary polynomials in the encrypted domain. In this scheme, specific integer vector operations including addition, linear transformation and weighted inner products are supported. Building upon that and by taking combinations of these primitive operations, arbitrary polynomial can be effectively computed with high accuracy.

This fully homomorphic encryption scheme is useful for applications in cloud computation, when one be interested in learning low dimensional representations of the stored encrypted data without exposing either data plaintext or operation plaintext to the server.

Moreover, in the dawn of the DL explosion for smartphones and embedded devices, it's possible for the devices deployed with DL models to be interested in accessing cloud data and make inferences on them. However, many DL-based models deployed on embedded devices are not well-protected, a research in 2019 showed that out of 218 DL-based Android apps, only less than 20% of them use encryption [2]. Homomorphic encryption, on the other hand, can resolve this problem easily by the DL models to meallable ciphertexts.

II. System Block Diagram

In this work, we present the Integer Vector Homomorphic Encryption scheme on embedded devices as a prototype for encrypting vector-valued functions. We will use softwares run by the processor as client and server in the encryption scheme, and hardwares implemented on FPGA as the accelerators. The system block diagram can then be drawn as shown in Fig. 1.

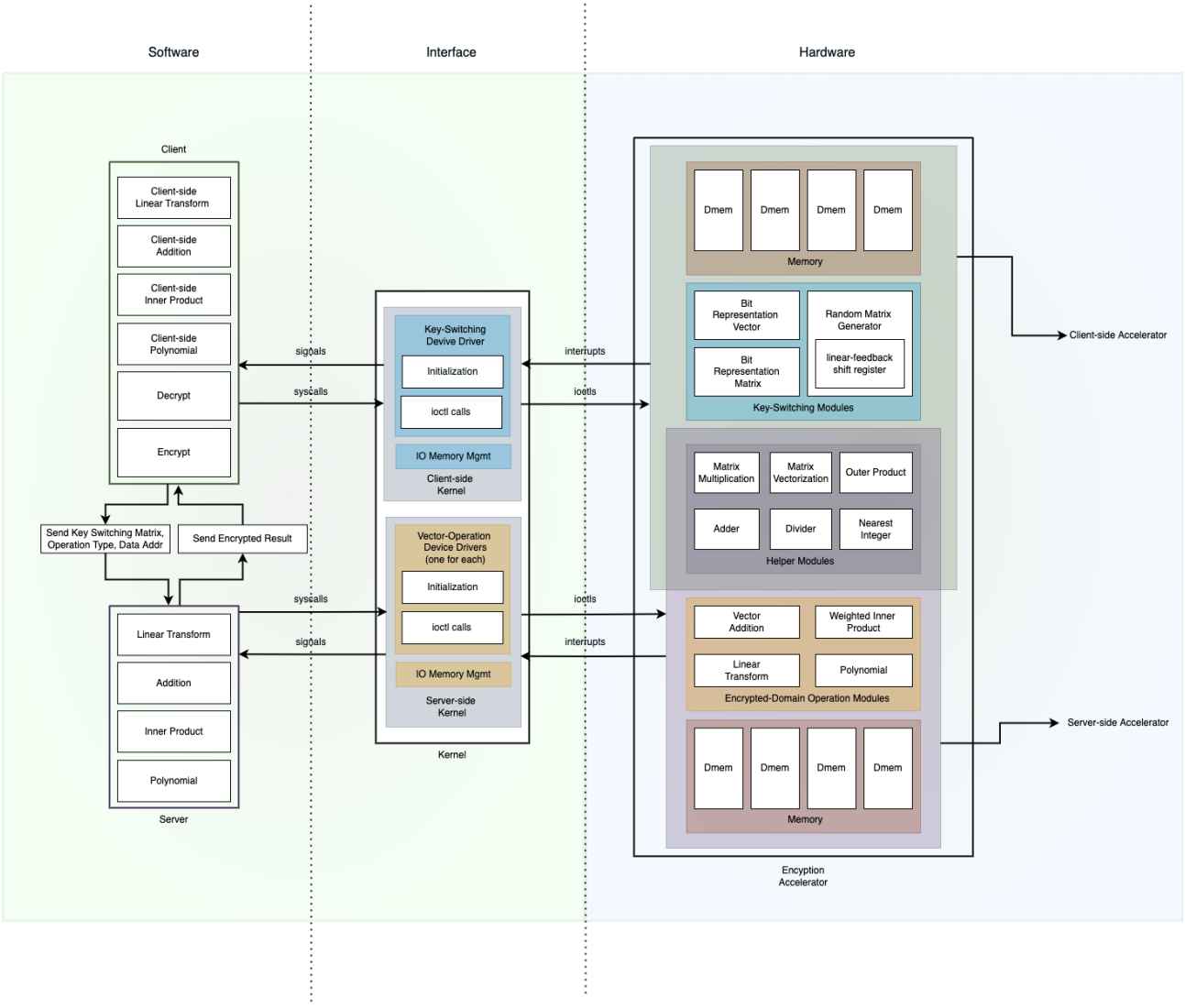

Figure 1. Overview of the Software/Hardware Architecture Design

In this architecture, whenever the client decides to perform a linear operation on the encrypted integer vectors stored in the server, the key-switching accelerator will compute the key-switching matrix that corresponds to the specific operation and the client will send the matrix along with desired operation to the server. On the other hand, once received these information from the client, the server carries out vectorized calculations in the custom accelerator over the encrypted domain, and returns encrypted computational results to the client.

III. Theory

A. *Motivation*

Consider the following scenario. Assume there is a client whose data is stored in a cloud server and the data is encrypted. The client decides to a make hidden query on the data without letting the cloud server learn the nature of the data or anything about the query.

To achieve this goal, fully homomorphic encryption is adopted to make ciphertext meallable. According to the homomorphic encryption scheme proposed by Zhou and Wornell, both plaintext and ciphertext considered in this case are integer-valued vectors with a integer-valued matrix as the secret key. Mathematically, we can formulate the problem as follows.

B. *Formulation*

Let the plaintext be $\mathbf{x} \in \mathbb{Z}^m$ and the corresponding ciphertext be $\mathbf{x} \in \mathbb{Z}^n$ with a large scalar w, a secret key $S \in \mathbb{Z}^{m\times n}$, and an error term $\mathbf{e} \in \left\{ \mathbf{e} \in \mathbb{Z}^m \middle| e_i < \frac{w}{2} \right\}$ such that

$$S\mathbf{c} = w\mathbf{x} + \mathbf{e} \tag{1}$$

To keep the error term small while applying multiple linear operations in the encrypted domain, we want to assume $\|S\| \ll w$.

Moreover, consider an arbitrary linear operator $A \in \mathbb{R}^{n\times m}$, to keep the result as integer vector and the scheme self-consistent, the following operations along with their notations will be used throughout this presentation and will be implemented in the accelerators.

Definition B1 For scalar $a \in \mathbb{R}$, define $\lfloor a \rceil$ to round a to the nearest integer.

Definition B2 For vector $\mathbf{a} \in \mathbb{R}^n$, define $\lfloor \mathbf{a} \rceil$ to round each entry a_i in $\mathbf{a}$ to the nearest integer.

Definition B3 For vector $\mathbf{a} \in \mathbb{R}^n$, define $|\mathbf{a}| := \max_i\{|a_i|\}$.

Definition B4 For matrix $A \in \mathbb{R}^{n\times m}$, define $|A| := \max_{ij}\{|A_{ij}|\}$.

Definition B5 For matrix $A \in \mathbb{R}^{n\times m}$, define $\mathrm{vec}\,(A) := \left[\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T\right]^T$ as a vector concatenating all a_i where $\mathbf{a}_i$ is the i^{th} column of A.

C. *Encryption and Key Switching*

To encrypt $\mathbf{x}$, let $w\mathbf{I}$ be the original secret key. Consider a key-switching operation that can change a secret-key-ciphertext pair to another with a new secret key while keeping the original plaintext encrypted. Without loss of generality, let the plaintext currently be encrypted as ciphertext $\mathbf{c} \in \mathbb{Z}^n$, we want to device a new secret-key-ciphertext pair $\left(S', \mathbf{c}'\right)$ such that

$$S'\mathbf{c}' = S\mathbf{c} \tag{2}$$

The first step is to convert S and $\mathbf{c}$ into intermediate bit representation S^* and $\mathbf{c}^*$ with $S^*\mathbf{c}^* = S\mathbf{c}$. The bit representation follows $|\mathbf{c}^*| := \max\left\{|c_i|\right\} = 1$ to prevent the transformed error term $\mathbf{e}'$ from growing too large to preserve correctness while rounding to the nearest integer.

First of all, pick a scalar ℓ that satisfies $2^\ell > |\mathbf{c}|$. Assume $c_i = b_{i0} + b_{i1}2 + \cdots + b_{i(\ell-1)}2^{\ell-1}$. We can then rewrite $\mathbf{c}$ in its bit representation following the rule: $\mathbf{b}_i = \left[b_{i(\ell-1)}, \ldots, b_{i1}, b_{i0}\right]^T$ with $b_{ik} \in \{-1, 0, 1\}, k \in \{\ell-1, \ldots, 0\}$. And this gives Eq. 3.

$$\mathbf{c}^* = \left[\mathbf{b}_1^T, \ldots, \mathbf{b}_n^T\right]^T \tag{3}$$

Similarly, we can make a bit-representation of the secret key S to acquire a new key S^* with Eq. 4.

$$S^*_{ij} = \left[2^{\ell-1}S_{ij}, \ldots, 2S_{ij}, S_{ij}\right] \tag{4}$$

And it can be demonstrated [3, 4] that this technique preserves $S^*\mathbf{c}^* = S\mathbf{c}$.

Beyond that, the second step is to convert the bit vector representation into a new secret-key-ciphertext pair. Consider a random Gaussian noise matrix $E \in \mathbb{Z}^{m\times n\ell}, E_{ij} \sim_{i.i.d.} \mathcal{N}\left(0, \sigma_E^2\right)$ for some σ_E, key-swiching matrix $M \in \mathbb{Z}^{n'\times n\ell}$ along with the new key S' such that

$$S'M = S^* + E \tag{5}$$

Consider keys only with the form $S' = [I, T]$, as a identity matrix concatenated horizontally with some matrix T, whose choice is not critical for our purposes. Now we can calculate

$$M = \begin{bmatrix} S^* - TA + E \\ A \end{bmatrix} \tag{6}$$

where A is another random Gaussian matrix $K \in \mathbb{Z}^{(n'-m)\times n\ell}$, $K_{ij} \sim_{i.i.d.} \mathcal{N}\left(0, \sigma_K^2\right)$ for some σ_K. Define

$$\mathbf{c}' = M\mathbf{c}^* \tag{7}$$

And it allows us to calculate

$$S'\mathbf{c}' = S^*\mathbf{c}^* + \mathbf{e}' \tag{8}$$

where $\mathbf{e}' = E\mathbf{c}^*$ is the new error term.

D. *Decryption*

With the encryption scheme specified above, given that we know the secret key S, large scalar w, notice that neareast integer rounding allows us to recover the plaintext by taking

$$\mathbf{x} = \left\lfloor \frac{S\mathbf{c}}{w} \right\rceil \tag{9}$$

according to the linear relation in Eq. 1 and the fact that the error term is taken from the set $\left\{ \mathbf{e} \in \mathbb{Z}^{\mathbb{m}} \middle| e_i < \frac{w}{2} \right\}$ with constrained error.

E. *Encrypted Domain Operations*

To dive into the mathematical algorithm in the operations, first we need to define several variables (or registers in hardware design):

Definition E1 $\mathbf{c_1}$, $\mathbf{c_2}$ are two ciphertexts in the big data stored in the server.

Definition E2 S is the secret key for encryption. To be mentioned, all the ciphertexts are encrypted with the same secret key, and the key only depends on the operation we choose.

Definition E3 M is the key-switch matrix that contains the information of the operation as well as the switched secret key.

Definition E4 $\mathbf{x_1}$, $\mathbf{x_2}$ are the corresponding plaintexts of ciphertexts $\mathbf{c_1}$, $\mathbf{c_2}$. Usually the cipher-plain pairs are predone and the client knows the address of each ciphertexts, so he/she just need to tell which 2 addresses are used.

After that, we can illustrate the algorithms for carrying out each operation in the encrypted domain as follows.

*1) **Addition**:* It is obvious that

$$S(\mathbf{c_1} + \mathbf{c_2}) = w(\mathbf{x_1} + \mathbf{x_2}) + (\mathbf{e_1} + \mathbf{e_2}) \tag{10}$$

so the addition of the ciphertexts in the encrypted domain simply follows

$$\mathbf{c}' = \mathbf{c_1} + \mathbf{c_2} \tag{11}$$

Notice that $\mathbf{e_1}, \mathbf{e_1}$ are devised such that the error is contained within $|\mathbf{e_1}| + |\mathbf{e_2}| \leq w$.

*2) **Linear Transform**:* Given a linear transformation $G \in \mathbb{Z}^{m'\times m}$, the encrypted result is

$$(GS)\,\mathbf{c} = wG\mathbf{x} + G\mathbf{e} \tag{12}$$

When we consider GS as a secret key, then the equation above is an encryption of plaintext $G\mathbf{x}$. Thus, the client need to create the key-switch matrix $M \in \mathbb{Z}^{(m'+1)\times m'\ell}$ to switch the key from GS to $S' \in \mathbb{Z}^{m'\times(m'+1)}$. After getting M and S', the client can send M to the server, and the cloud server simply computes

$$\mathbf{c}' = M\mathbf{c} \tag{13}$$

as the encrypted result for the client to decrypt.

*3) **Weighted Inner Product**:* Consider some plaintext $\mathbf{x}_1, \mathbf{x}_2$, their corresponding ciphertexts $\mathbf{c}_1, \mathbf{c}_2$ and a matrix H with information about weights of the inner products we want to take. The inner product of our interest takes the form

$$h = \mathbf{x}_1^T H \mathbf{x}_2 \tag{14}$$

Now, we need one mathematical side note to proceed. Consider the following Lemma.

Lemma E1 For any arbitrary vectors $\mathbf{x}, \mathbf{y}$ and matrix M with appropriate dimensions, its inner product follows

$$\mathbf{x}^T H \mathbf{y} = \text{vec}\,(M)^T \,\text{vec}\left(\mathbf{x}\mathbf{y}^T\right) \tag{15}$$

See [1] and [3] for proof of this Lemma.

In order to compute Eq. 14 in the encrypted domain, we can leverage Lemma E1 to derive the proposition specified below.

Proposition E1 Consider secret key $S = \text{vec}\left(S_1^T H S_2\right)^T$ and ciphertext $\mathbf{c} = \left\lfloor \frac{\text{vec}\left(\mathbf{c}_1 \mathbf{c}_2^T\right)}{w} \right\rceil$ corresponding to the plaintext of the inner product $\mathbf{x}_1^T H \mathbf{x}_2$. And for some error term e independent of $\mathbf{e}_1$ and $\mathbf{e}_2$ for $\mathbf{c}_1$ and $\mathbf{c}_2$ they satisfy the following condition,

$$\text{vec}\left(S_1^T H S_2\right)^T \left\lfloor \frac{\text{vec}\left(\mathbf{c}_1 \mathbf{c}_2^T\right)}{w} \right\rceil = w \mathbf{x}_1 H \mathbf{x}_2 + e \tag{16}$$

See [3] and [5] for details about the proof of this Proposition.

Notice that instead of a key switching matrix here, the operator to be applied to $\left\lfloor \frac{\text{vec}\left(\mathbf{c}_1 \mathbf{c}_2^T\right)}{w} \right\rceil$ in calculating the weighted inner product is a row vector $\text{vec}\,(S_1 H S_2)^T$. Because of the vectorization operation, the width of operator $\text{vec}\,(S_1 H S_2)^T$ after key switching is n^2.

To this end, we can leverage this property of the row vector, and concatenate m' such operators, each of which corresponds to a weight matrix $H_j, j \in \{1, \ldots, m'\}$, together to be a key-switching matrix S' so that m' such weighted inner products can be carried out simultaneously.

Namely, we now have

$$S' \left\lfloor \frac{\text{vec}\left(\mathbf{c}_1 \mathbf{c}_2^T\right)}{w} \right\rceil = w\mathbf{p} + \mathbf{e} \tag{17}$$

where vector $\mathbf{p}$ contains m' entries of weighted inner products, each of which corresponds to an inner product $\mathbf{x}_1^T H_j \mathbf{x}_2$. We can then apply the key-switching algorithm as specificed in Eq. 3 to Eq. 8 to S' and obtain a corresponding key switching matrix $M \in \mathbb{Z}^{n^2 \times (m'+1)}$ along with a new secret key $S'' \in \mathbb{Z}^{m' \times (m'+1)}$. The final ciphertext is therefore

$$\mathbf{c}'' = M \left\lfloor \frac{\text{vec}\left(\mathbf{c}_1 \mathbf{c}_2^T\right)}{w} \right\rceil \tag{18}$$

*4) **Polynomial**:* Building upon the three aforementioned operations, we can now synthesize polynomial operations with weighted inner products introduced above to calculate multiple arbitrary degree polynomials in parallel. All we need is to expand the x and S' and thereby account for constant terms in polynomials. Let the modified input vector $\mathbf{x}_p = [1, x_1, x_2, ..., x_n]^T$. The new ciphertext then becomes $\mathbf{c}' := [w, c_1, ..., c_n]^T$. We also need to extend the secret key S because we simply added a constant factor 1 to $\mathbf{x}$:

$$S' := \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix} \tag{19}$$

Thus, given any inner product weight matrices $\{H_j\}$, we can calculate its key-switch pair M so that each $\mathbf{x}_p^T H_j \mathbf{x}_p$ can be calculated in accordance with Eq. 16, or more compactly with Eq. 17 to deal with all $\{H_j\}$ in parallel. The steps follow the ones introduced in the *Weighted Inner Product* section. For degree 2 polynomials, one inner product simply does the computation. For higher-degree polynomials, notice that higher-order polynomials can be calculated based on lower-order polynomials.

*5) **Examples**:*

- **Key Switching Example** Consider the case $\ell = 3$, ciphertext $\mathbf{c} = [1, -2]$, and

$$S = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{20}$$

The corresponding bit representation of $\mathbf{c}$ and S are therefore

$$c^* = [0, 0, 1, 0, -1, 0] \tag{21}$$

$$S^* = \begin{bmatrix} 4 & 2 & 1 & 8 & 4 & 2 \\ 12 & 6 & 3 & 16 & 8 & 4 \end{bmatrix} \tag{22}$$

- **Polynomial/Weighted Inner Product Example** To calculate the polynomial $f(\mathbf{x}) = x_2^2 - 4x_1x_3$, we can express it as an weighted inner product $\mathbf{x'}^T H \mathbf{x'}$, and

$$H = \begin{bmatrix} 0 & 0 & -2 \\ 0 & 1 & 0 \\ -2 & 0 & 0 \end{bmatrix} \tag{23}$$

IV. Design

A. Software

Software serves as both the client and the server in this project, each of which takes different responsibilities in the process of encryption and decryption. In order to accomplish most of the operations in C, we also implemented our own matrix operation library. First we will start from the matrix library source code.

*1) **Matrix Library**:* We defined a new data struct data type called `Mat`, which includes the row number and col number of the matrix, and also a pointer to the double typed data entries. To help debugging the most of the matrix operations, we have a `showmat` helper function to print the matrix elements in place. We also supported a lot of handy matrix operations, such as creating an identity matrix, a matrix filled with random numbers, handler for data entries, etc. In order to support more advanced matrix operations that will be used in our homomorphic encryption scheme, we included several integer vector arithmetics, such as sum and minus of two integer vectors, scalar multiple of a matrix and one scalar value, inner product of two matrix, find the transpose and norm of the matrix, and we also supported computing the inverse of the matrix, which can be docomposed into steps of finding the determinant of matrix. All these mentioned operations are implemented in our `mat.h` header file, which can be used to implement all the client-server operations. See the Implementation section for more details.

*2) **Client Functions**:* The client-side software serves as the APIs that are exposed to user as function calls to initiate general homomorphic encryption, and key switching operation that generates public keys for one of the four operations discussed above. Note that key switching is not needed for vector addition. Once the client-side functions are called, homomorphic encryption and key switching for each of the primary operation will be done in the key switching matrix accelerator by making a syscall and invoking the corresponding device driver.

- **Client Side Key Switching Matrix:**
  Here we have a set of functions to implement the key switching read and write with the accelerator hardware through ioctl calls. For example, we have 3 reading functions, including reading width, reading length and reading data for the key switching arguments. Similarly, we have 4 writing functions, where we used four control signals to separate writing operations for data, width, length and operation. Notice that a library of client-side syscalls is also needed for client user program to talk to the kernel.
- **Client Side Encryption:**
  The client side encryption scheme involves telling hardware to evoke get random matrix module, which initializes the memory space for storing a random matrix. After the random data is generated by the software, it will call hardware to inform a fully loaded function. Encryption scheme also involves the process of multiplying a random generated number and applied on an error term, which guarantees its randomness.
- **Client Side Decryption:**
  The client side decryption scheme takes the two input matrix from server, one is the secret key and the other one is the ciphertext, and then multiply two matrix to get the

The function specified above only works for degree-2 polynomial as an example demonstrate the workflow. To implement arbitrary degree-d polynomial, more sophisticated functions are needed to factorize polynomials into a sequence of weighted inner product operations.

*3) **Server Functions**:* The server-side software serves as the data center where user information are stored. It takes pre-configured and encrypted data and performs encrypted domain operations in the server-side hardware to accelerator computations. All computations can be decomposed into a combination of vector additions, linear transformations and weighted inner products. See the Implementation section for details.

The function specified above only works for degree-2 polynomial. To generalize the function that computes arbitrary degree-d polynomial, more sophisticated functions are needed to factorize polynomials into a sequence of weighted inner product operations.

B. Hardware

*1) **Client-side Accelerator**:* The client-side accelerator is mostly responsible for key-switching operations to get the correct key-switching matrices for calculating linear transformation, weighted inner product and polynomial. After the key switching

completes, a new Secret key S' will be derived and retained by the client to decrypt encrypted result once the server has done computation, and a key-switching matrix M will also be generated and sent to the server to carry out different computations.

- **Key-switching accelerator**: The key-switching accelerator is the only device driver in the client-side hardware. It composes of two major sub-modules, one module for getting bit-representation of a vector corresponding to Eq. 3, one module for getting bit-representation of a matrix corresponding to Eq. 4, one module for getting random Gaussian matrix and one module for getting a noise matrix who elements are 4-bit integers in order to calculate Eq. 6. See the Implementation section for details. Note that for vectors and matrices exceeding the size capacity of the accelerators, the kernel is responsible for breaking the query into reasonable sizes that fits the accelerators (maximum row and column number as 256).
  Notice that more parallelism can be acquired with this design if we wrap each of the module presented above with a higher-level module that instantiates multiple instances simultaneous, one for every row vector. This way, we can achieve linear speedup proportional to the number of instances initiated.

*2) **Server-side Accelerator**:* Since the operations over the encrypted domain have four different types, each of which involves different number and size of inputs, we want to use four different accelerators, each of which serves as a distinct device driver in the kernel. Also, in order to be robust, each device driver needs to have sufficiently large memory in order to handle with large key-switching matrices in various tests. This memory block will be implemented as DMEM and will be discussed in more details in section VI.

Note that each device driver corresponding to each of the four operations takes different number of ciphertexts and perform different computations (as mentioned in the *Encrypted Domain Operation* section).

In other words, four modules are needed including addition accelerator, linear transformation accelerator, weighted inner product accelerator and polynomial accelerator respectively.

To improve computational efficiency (by exploiting mainly the parallel computing power of the board and DRAM in FPGA) and try to make our system capable of handling bigger sized data, we convert our input data to flows with certain "batch size" and use single-cycle-processing-like scheme to deal with it. In our implementation, the batch size is defined as 16, meaning that we can handle 16 elements of each input at a time. Following this, we created specific arithmetic logic unit modules.

- **Addition Accelerator**:
  The vector addition accelerator takes the simple form of performing a element-wise addition of two vectors. In the pseudocode presented below, the vector addition accelerator adds two 16-element vectors together at one cycle. Queries about ciphertexts stored at `c_1_addr` and `c_2_addr` will be read from the database, and each element will be sent to one port of the module.
  We create `on` signal in the input to control the status of system, meaning that the system will only work when `on` signal is 1. To instruct the validation of output, we create `write` signal. It will be 1 if and only if the result has been calculated. See the Implementation section for details.
- **Linear Transformation Accelerator**:
  The linear transformation accelerator simply takes a matrix multiplication of the key-switching matrix received from the client and apply it as a linear operator to ciphertext $\mathbf{c}$ stored at address `c_addr`. Queries about ciphertext stored at `c_addr` will be read from the database and each 16-element row of the key-switching matrix M will be sent to the accelerator from the server each cycle, with each of these elements takes one input port of the module.
  In our implementation, the hardware takes 16 elements of a row of a linear operator and 16 elements of a vector at each clock cycle. This way, an inner product between the two can be computed in one cycle and fed back into the top-level module for the encrypted domain operations, where all inner products will be collected and stored in a vector. See the Implementation section for more details.
- **Weighted Inner Product Accelerator**:
  Recall Eq. 18, for this example pseudocode implementation of the weighted inner product module, consider ciphertext as a 4-element vector $\mathbf{c}$. 4-element vector $\mathbf{c}$ is chosen because matrix vectorization after outer production yields $4^2 = 16$ element output.
  In our implementation, we divide this operation into 3 stages. In stage 1, ciphertexts $\mathbf{c}_1, \mathbf{c}_2$ stored will be sent to the accelerator from the top-level module along with the scalar w in Eq. 18 in the first cycle. Each 16-element row of the key-switching matrix M at one cycle. In stage 2, we vectorize the outer products of the two inputs from stage 1. In stage 3, we do linear transformation of the output, using the same theory as in **Linear Transformation**. See the Implementation section for more details.
- **Polynomial Accelerator**:
  Note that calculating polynomial is essentially a glorified weighted inner product according to our scheme. Therefore, having let the kernel figure out how to perform a sequence of weighted inner products to obtain a polynomial of degree d, we can potentially customize an accelerator for a polynomial of, say degree 2, that supports inner product for two ciphertexts $\mathbf{c}_1, \mathbf{c}_2$ as 16-element vectors. The only difference is that W is now derived from a secret key of the form Eq. 19. Therefore, as long as we modify the weighted inner product module presented above to a slightly larger size that fits the key-switching matrix corresponding to $S' \to S''$ in Eq. 18, the accelerator can do its job.

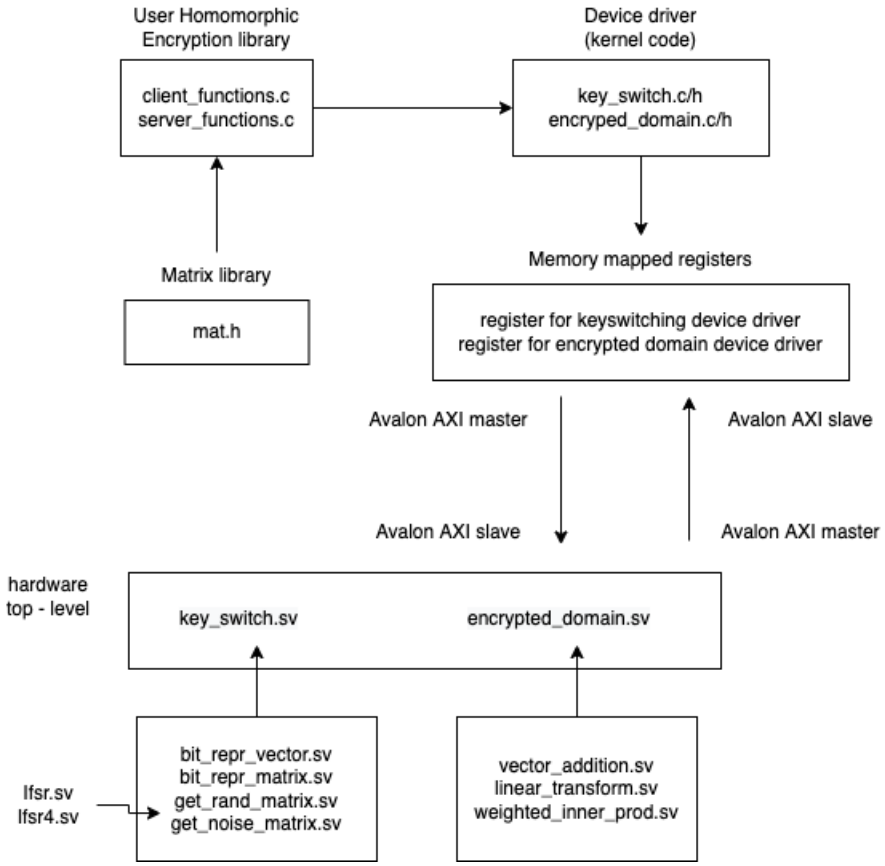

Figure 2. System Block Diagram for Hardware Software Interface

In our design, we action of scheduling and arranging accumulative weighted inner product is done in the kernel. Only individual weighted inner products are performed in hardware.

C. *Hardware/Software Interface*

1) ***Client-side Kernel****:* For the client-side kernel, the only device driver can be accessed is the key-switching accelerator. Recall from Eq. 4 to Eq. 6, for each key-switching operation, in order to compute S^*, M and S', the module needs to take matrix S as input, along with two random Gaussian matrices that can be generated by the hardware itself (see the *Client-side accelerator* section).

In this design, we plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths (height, number of rows). Namely, from Eq. 5, it follows $n' \leq 256, n\ell \leq 16$.

At the lowest level based on our hardware design, the key-switching operation is done by taking element-wise inputs and store them into multiple DMEM blocks, each of which as a row vector. However, notice that this operation can be further parallelized if we modify the `bit_repr_matrix` module such that processes each row vector at a time. Alternatively, it can also be done by instantiating multiple instances of `bit_repr_vector` to convert each row of S to its bit representation at one cycle.

With this architecture, there will be 16 32-bit-wide input ports for the key-switching accelerator to deliver one row of the secret key S at a time and achieve a significant speedup.

2) ***Server-side Kernel****:* For the client-side kernel, the device drivers that can be accessed include addition accelerator, linear transformation accelerator, weighted inner product accelerator and polynomial accelerator.

In this design, we plan to support vector operations with up to 16 elements at a time in one cycle. Each element in a vector is taken as a 32-bit signed integer in agreement with the scheme. Specifically, that means the addition accelerator can calculate the sum of two 16-element vectors $\mathbf{c}_1, \mathbf{c}_2$ in one cycle. The linear transformation accelerator can compute the inner product of one 16-element row vector W_i with one 16-element column vector $\mathbf{c}$ and completes the desired linear transformation in m cycles given $W \in \mathbb{Z}^{m \times 16}$. The weighted inner product accelerator and the polynomial accelerator take in 16-element row vector of W each cycle along with $\mathbf{c}_1, \mathbf{c}_2, w$ at the first cycle, and they generate the output all at once after some cycles of operations to compute intermediate steps.

As discussed in the *Server-side accelerator* section, one thing to notice is that polynomial accelerator is essentially the same as the weighted inner product accelerator with a slightly different key-switching matrix and ciphertext. Therefore, it's important for the kernel and the polynomial accelerator device driver to handle one weighted inner product operation at a time. The order at which weighted inner products will be carried out and the exact steps are handled by the server-side software as discussed in *Server-side software* section.

3) ***Avalon Bus****:* For each device driver, it keeps a set of registers that can talk to the FPGA via Avalon memory mapped interface. When the Linux kernel writes to FPGA, the kernel serves as the master while the FPGA serves as the slave through the Avalon bridge. And vice versa for the case where the FPGA writes to the kernel at specific addresses where the data is stored in registers and can be retrieved at a later time.

See Figure 2 for detailed hardware-software communication scheme.

V. Resource Budgets

For the client-side accelerator, notice that we plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths. Since each element is a 32-bit integer, each row in the key-switching matrix takes a DRAM of size 16×32, so the maximum size for a DRAM that stores each row is $M_i = 512$ bits. For a key-switching matrix with $m \leq 256$ rows, it takes up to 256 such DRAM blocks to store all rows. This costs $256 \times 512 = 131072$ bits $= 16384$B $= 16$kB. Let each of this memory module be call Dmem in the system block diagram as shown in Fig. 1. We can create 8 such DRAM modules as client-side accelerator's cache, and it takes $16\text{kB} \times 8 = 128\text{kB}$ of memory on FPGA.

For the server-side accelerator, we as well plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths. This is the same as the client-side accelerator because the key-switching matrix serves as the public key and need to be used by the server to carry out linear transformation and polynomial computations. We want to create 16 such DRAM modules as server-side accelerator's cache for not only key-switching matrix but also results from outer product and vector addition, so it takes $16\text{kB} \times 16 = 256\text{kB}$ of memory on FPGA.

Some registers might also be needed to store real-time vector inputs, results including nearest integer vector division, matrix vectorization and linear transformation results while doing linear transformation. Assume they take up to 256kB of memory on FPGA.

Combine all of them together, we need around 640kB of memory on FPGA. From The Cyclone® V FPGA core architecture's specifications, it states that the FPGA comprises of up to 12 Mb of embedded memory arranged as 10 Kb (M10K) blocks. Namely, we have more than 1 MB of memory avaliable on FPGA.

Therefore, we can assume that there is sufficient memory on FPGA for all computations required.

VI. Hardware Simulation

A. *Vector Addition*

Figure 3 show the simulation of addition. In this experiment, we did the addition of 2 inputs with same values. When the **on** is on, the whole system starts working, and stops working as the **on** is off. The **write** will tell the user when is the time to collect data. By the way, with the **on** is on, the user can pour all the input data, and wait to collect data multiple times. The output will not influence the operations of input.

Test code is here:

```

module test();

logic CLOCK_50;  // 50 MHz Clock input
logic reset;
logic on;

logic [31:0] c_1_1, c_1_2, c_1_3, c_1_4, c_1_5, c_1_6, c_1_7, c_1_8, c_1_9, c_1_10, c_1_11,
    c_1_12, c_1_13, c_1_14, c_1_15, c_1_16;
logic write;
logic [31:0] c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_10, c_11, c_12, c_13, c_14, c_15,
    c_16;

initial begin
        #5 begin
            reset = 1; on = 0;
            c_1_1 = 32'b0;
            c_1_2 = 32'b0;
            c_1_3 = 32'b0;
            c_1_4 = 32'b0;
            c_1_5 = 32'b0;
            c_1_6 = 32'b0;
```

```
            c_1_7 = 32'b0;
            c_1_8 = 32'b0;
            c_1_9 = 32'b0;
            c_1_10 = 32'b0;
            c_1_11 = 32'b0;
            c_1_12 = 32'b0;
            c_1_13 = 32'b0;
            c_1_14 = 32'b0;
            c_1_15 = 32'b0;
            c_1_16 = 32'b0;
            end
        #10 reset = 0;
        #10 begin
            on = 1;
            c_1_1 = 32'd1;
            c_1_2 = 32'd2;
            c_1_3 = 32'd3;
            c_1_4 = 32'd4;
            c_1_5 = 32'd5;
            c_1_6 = 32'd6;
            c_1_7 = 32'd7;
            c_1_8 = 32'd8;
            c_1_9 = 32'd9;
            c_1_10 = 32'd10;
            c_1_11 = 32'd11;
            c_1_12 = 32'd12;
            c_1_13 = 32'd13;
            c_1_14 = 32'd14;
            c_1_15 = 32'd15;
            c_1_16 = 32'd16;
            end
        #10 on = 0;
end

initial begin
        #5 CLOCK_50 = 0;
        forever begin
                #5 CLOCK_50 = ~ CLOCK_50; end
end

vector_addition add16(.clk(CLOCK_50), .reset(reset), .on(on),
    .c_1_1(c_1_1), .c_1_2(c_1_2), .c_1_3(c_1_3), .c_1_4(c_1_4), .c_1_5(c_1_5), .c_1_6(c_1_6),
    .c_1_7(c_1_7), .c_1_8(c_1_8), .c_1_9(c_1_9), .c_1_10(c_1_10), .c_1_11(c_1_11),
    .c_1_12(c_1_12), .c_1_13(c_1_13), .c_1_14(c_1_14), .c_1_15(c_1_15), .c_1_16(c_1_16),
    .c_2_1(c_1_1), .c_2_2(c_1_2), .c_2_3(c_1_3), .c_2_4(c_1_4), .c_2_5(c_1_5), .c_2_6(c_1_6),
    .c_2_7(c_1_7), .c_2_8(c_1_8), .c_2_9(c_1_9), .c_2_10(c_1_10), .c_2_11(c_1_11),
    .c_2_12(c_1_12), .c_2_13(c_1_13), .c_2_14(c_1_14), .c_2_15(c_1_15), .c_2_16(c_1_16),
    .write(write),
    .c_1(c_1), .c_2(c_2), .c_3(c_3), .c_4(c_4), .c_5(c_5), .c_6(c_6),
    .c_7(c_7), .c_8(c_8), .c_9(c_9), .c_10(c_10), .c_11(c_11),
    .c_12(c_12), .c_13(c_13), .c_14(c_14), .c_15(c_15), .c_16(c_16)
    );

endmodule
```

B. Linear Transformation

Figure 4 show the simulation of linear transformation. When the **on** is on, the whole system starts working, and stops working as the **on** is off. The **write** will tell the user when is the time to collect data, and it is controlled by the value of **count**. By the way, with the **on** is on, the user can pour all the input data, and wait to collect data multiple times. The output will not influence the operations of input, just like in this experiment, we did 2 batch-sized calculations.

Test code is here:

```
module test();

logic CLOCK_50;  // 50 MHz Clock input
logic reset;
logic on;
```

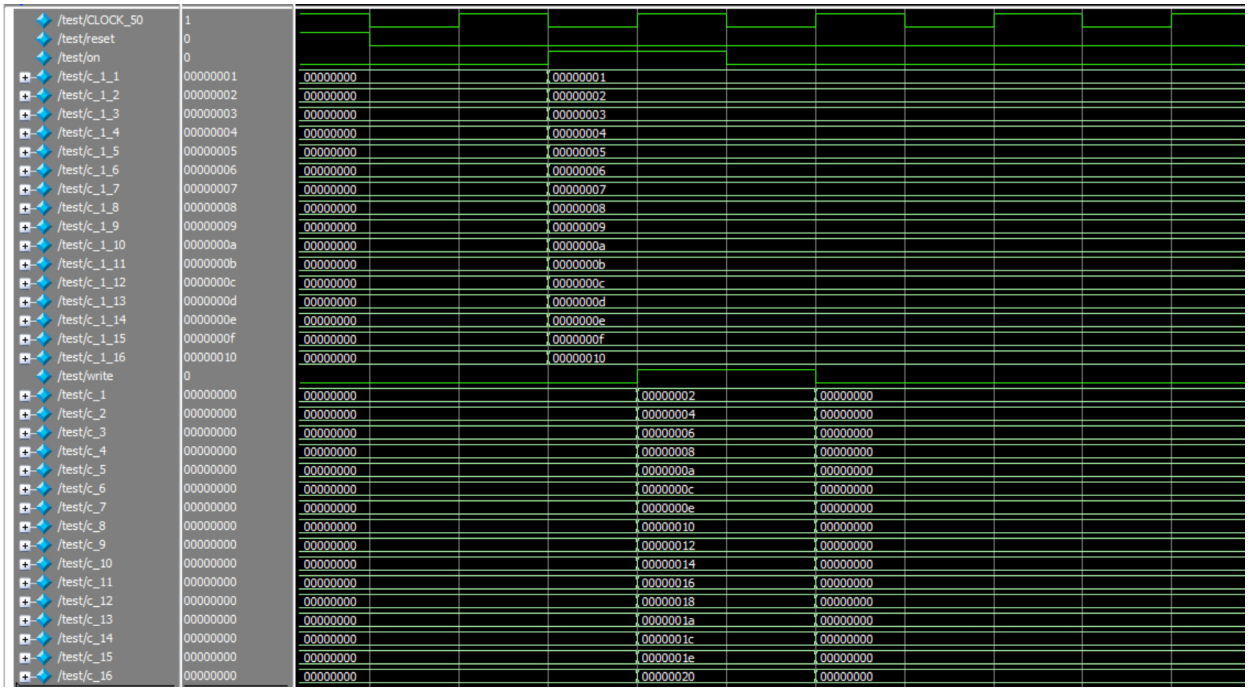

Figure 3. Simulation of Vector Addition

```
6
7  logic [31:0] c_1_1, c_1_2, c_1_3, c_1_4, c_1_5, c_1_6, c_1_7, c_1_8, c_1_9, c_1_10, c_1_11,
       c_1_12, c_1_13, c_1_14, c_1_15, c_1_16;
8  logic [31:0] m_1_1, m_1_2, m_1_3, m_1_4, m_1_5, m_1_6, m_1_7, m_1_8, m_1_9, m_1_10, m_1_11,
       m_1_12, m_1_13, m_1_14, m_1_15, m_1_16;
9
10 logic write;
11 logic [31:0] y;
12
13 initial begin
14         #5 begin
15             reset = 1; on = 0;
16             c_1_1 = 32'b0;
17             c_1_2 = 32'b0;
18             c_1_3 = 32'b0;
19             c_1_4 = 32'b0;
20             c_1_5 = 32'b0;
21             c_1_6 = 32'b0;
22             c_1_7 = 32'b0;
23             c_1_8 = 32'b0;
24             c_1_9 = 32'b0;
25             c_1_10 = 32'b0;
26             c_1_11 = 32'b0;
27             c_1_12 = 32'b0;
28             c_1_13 = 32'b0;
29             c_1_14 = 32'b0;
30             c_1_15 = 32'b0;
31             c_1_16 = 32'b0;
32
33             m_1_1 = 32'b0;
34             m_1_2 = 32'b0;
35             m_1_3 = 32'b0;
36             m_1_4 = 32'b0;
37             m_1_5 = 32'b0;
38             m_1_6 = 32'b0;
39             m_1_7 = 32'b0;
40             m_1_8 = 32'b0;
41             m_1_9 = 32'b0;
42             m_1_10 = 32'b0;
```

```
            m_1_11 = 32'b0;
            m_1_12 = 32'b0;
            m_1_13 = 32'b0;
            m_1_14 = 32'b0;
            m_1_15 = 32'b0;
            m_1_16 = 32'b0;
            end
        #10 reset = 0;
        #10 begin // count1, count2,...,count16
            on = 1;
            c_1_1 = 32'd1;
            c_1_2 = 32'd2;
            c_1_3 = 32'd3;
            c_1_4 = 32'd4;
            c_1_5 = 32'd5;
            c_1_6 = 32'd6;
            c_1_7 = 32'd7;
            c_1_8 = 32'd8;
            c_1_9 = 32'd9;
            c_1_10 = 32'd10;
            c_1_11 = 32'd11;
            c_1_12 = 32'd12;
            c_1_13 = 32'd13;
            c_1_14 = 32'd14;
            c_1_15 = 32'd15;
            c_1_16 = 32'd16;

            m_1_1 = 32'd1;
            m_1_2 = 32'd2;
            m_1_3 = 32'd3;
            m_1_4 = 32'd4;
            m_1_5 = 32'd5;
            m_1_6 = 32'd6;
            m_1_7 = 32'd7;
            m_1_8 = 32'd8;
            m_1_9 = 32'd9;
            m_1_10 = 32'd10;
            m_1_11 = 32'd11;
            m_1_12 = 32'd12;
            m_1_13 = 32'd13;
            m_1_14 = 32'd14;
            m_1_15 = 32'd15;
            m_1_16 = 32'd16;
            end
        //#170 on = 0;
end



initial begin
        #5 CLOCK_50 = 0;
        forever begin
                #5 CLOCK_50 = ~ CLOCK_50; end

end

linear_transform linear_trans0 (.clk(CLOCK_50), .reset(reset), .on(on),
    .M_1(m_1_1), .M_2(m_1_2), .M_3(m_1_3), .M_4(m_1_4), .M_5(m_1_5), .M_6(m_1_6),
    .M_7(m_1_7), .M_8(m_1_8), .M_9(m_1_9), .M_10(m_1_10), .M_11(m_1_11),
    .M_12(m_1_12), .M_13(m_1_13), .M_14(m_1_14), .M_15(m_1_15), .M_16(m_1_16),
    .c_1(c_1_1), .c_2(c_1_2), .c_3(c_1_3), .c_4(c_1_4), .c_5(c_1_5), .c_6(c_1_6),
    .c_7(c_1_7), .c_8(c_1_8), .c_9(c_1_9), .c_10(c_1_10), .c_11(c_1_11),
    .c_12(c_1_12), .c_13(c_1_13), .c_14(c_1_14), .c_15(c_1_15), .c_16(c_1_16),
    .write(write),
    .y(y)
    );

endmodule
```

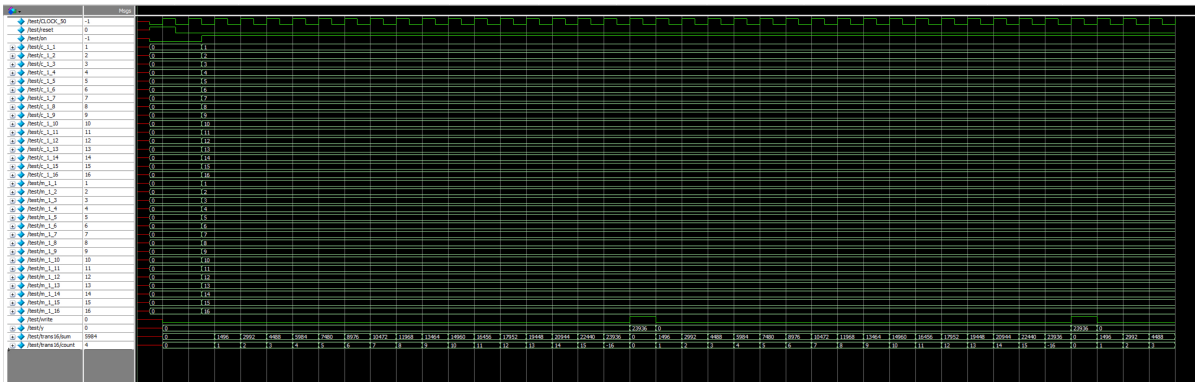

Figure 4. Simulation of Linear Transformation

C. *Weighted Inner Product*

Figure 5 shows the simulation of weight inner product. When the **start** changes to 1, the module kicks off. Since the module is a Moore machine that only depends on its own computational state once inputs are loaded, we consider three internal signals. The **gen_enable** is used to load input data. And the **vec_enable** and the **read_enable** are used during the stages. We use **M_on** to instruct the output data.

The code is here:

```
#include <iostream>
#include <iomanip>
#include "Vweighted_inner_product.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

// encrypted data
int c_1[2] = {0x1, 0xfffffffe};    // 0-1

int c_2[2] = {0x5, 0xfffffffd};    // 0-1

// encrypted linear operator
int M[4][4] = { {0x1, 0x2, 0x3, 0x4},    // 0-7
                              {0xfffffffe, 0xfffffffd, 0xfffffffc, 0xfffffffb},    // 8-15
                {0x3, 0x6, 0x9, 0xc},    // 16-23
                {0xfffffffc, 0xfffffffb, 0x9, 0xc}    // 24-31
               };

// reminder: vectorized result
int x[4] = {0x5, 0xfffffffd, 0xfffffff6, 0x6};     // 0-4

int w = 1;

// expected result
int a[4] = {0xfffffff9, 0x9, 0xffffffeb, 0xffffffe9};     // 0-4

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  // Treat the argument on the command-line as the place to start
  int width, length, ell;
  if (argc > 1 && argv[1][0] != '+') {
    width = atoi(argv[1]);
  } else {
    width = 2; // Default, anything <= 4 works
  }
```

```
Vweighted_inner_product * dut = new Vweighted_inner_product;  // Instantiate a new module

// Enable dumping a VCD file

Verilated::traceEverOn(true);
VerilatedVcdC * tfp = new VerilatedVcdC;
dut->trace(tfp, 99);
tfp->open("weighted_inner_product.vcd");

dut->clk = 0;
dut->start = 0;
dut->width = width;

int time;
std::cout << "width: " << width;
std::cout << std::endl;

int r = 0;
int i = 0;
int err = 0;

int start_stamp = 100;
// after outer product and vectorization completes
int comp_stamp = start_stamp + width * width * 20 + width * width * 20 + 10* 20;
std::cout << "comp_stamp: " << comp_stamp;
std::cout << std::endl;

for (time = 0 ; time < 10000 ; time += 10) {
  dut->clk = ((time % 20) >= 10) ? 1 : 0; // Simulate a 50 MHz clock
  if (time == 20) dut->reset = 1; // Handle "reset" on for four cycles
  if (time == 100) {
    dut->reset = 0;
    dut->start = 1;   // Put "start" on
  }

  // take inputs
  if (time >= start_stamp && time <= start_stamp + 20 && dut->clk == 1) {
      dut->w = w;

      dut->c_1_1 = c_1[0];
      dut->c_1_2 = c_1[1];
      dut->c_1_3 = 0;
      dut->c_1_4 = 0;

      std::cout << "vector 1 input received: " << (int) c_1[0];
      std::cout << std::endl;
      std::cout << "vector 1 input received: " << (int) c_1[1];
      std::cout << std::endl;
      std::cout << "vector 1 input received: " << 0;
      std::cout << std::endl;
      std::cout << "vector 1 input received: " << 0;
      std::cout << std::endl;

      dut->c_2_1 = c_2[0];
      dut->c_2_2 = c_2[1];
      dut->c_2_3 = 0;
      dut->c_2_4 = 0;
```

```
        std::cout << "vector 2 input received: " << (int) c_2[0];
        std::cout << std::endl;
        std::cout << "vector 2 input received: " << (int) c_2[1];
        std::cout << std::endl;
        std::cout << "vector 2 input received: " << 0;
        std::cout << std::endl;
        std::cout << "vector 2 input received: " << 0;
        std::cout << std::endl;
    }

    if (time == start_stamp + 20) {
        dut->start = 0;
    }

    // take matrix input
    if (time >= comp_stamp && time <= comp_stamp + 20 * width * width && dut->clk == 1) {
        dut->M_on = 1;
        dut->M_1 = M[r][0];
        dut->M_2 = M[r][1];
        dut->M_3 = M[r][2];
        dut->M_4 = M[r][3];
        dut->M_5 = 0;           // unused
        dut->M_6 = 0;
        dut->M_7 = 0;
        dut->M_8 = 0;
        dut->M_9 = 0;
        dut->M_10 = 0;
        dut->M_11 = 0;
        dut->M_12 = 0;
        dut->M_13 = 0;
        dut->M_14 = 0;
        dut->M_15 = 0;
        dut->M_16 = 0;

        std::cout << "matrix input received: " << M[r][0];
        std::cout << std::endl;
        std::cout << "matrix input received: " << M[r][1];
        std::cout << std::endl;
        std::cout << "matrix input received: " << M[r][2];
        std::cout << std::endl;
        std::cout << "matrix input received: " << M[r][3];
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
        std::cout << std::endl;
        std::cout << "matrix input received: " << 0;
```

```
            std::cout << std::endl;
            std::cout << "matrix input received: " << 0;
            std::cout << std::endl;
            std::cout << "matrix input received: " << 0;
            std::cout << std::endl;
            std::cout << "matrix input received: " << 0;
            std::cout << std::endl;
            std::cout << "matrix input received: " << 0;
            std::cout << std::endl;

            r+= 1;
        }

        if (time == comp_stamp + 20 * width * width + 20) {
                dut->M_on = 0;
        }


        dut->eval();      // Run the simulation for a cycle
        tfp->dump(time); // Write the VCD file for this cycle

        // compare outputs for (length) rows
        // for each row, compare outputs for (width * ell) cycles
        if (time >= comp_stamp && dut->done && dut->clk == 1) {
            std::cout << "vector output received: " << (int) dut->y;
            std::cout << std::endl;
            if (dut->y == a[i])
                std::cout << " OK";
            else {
                std::cout << " INCORRECT expected " << std::setfill('0') << (int) a[i];
                err += 1;
            }
            std::cout << std::endl;

            i += 1;
        }

        if (i == width * width) {
            break;
        }

    }

    tfp->close(); // Stop dumping the VCD file
    delete tfp;

    dut->final(); // Stop the simulation
    delete dut;

    return 0;
}
```

D. Bit Representation of Vector and Matrix

For the two modules `bit_repr_vector.sv` and `bit_repr_matrix.sv`, we use one signal testbench `key_switching.cpp` that uses Verilator to test whether the computational results of the two modules match with expectations. The code is attached here:

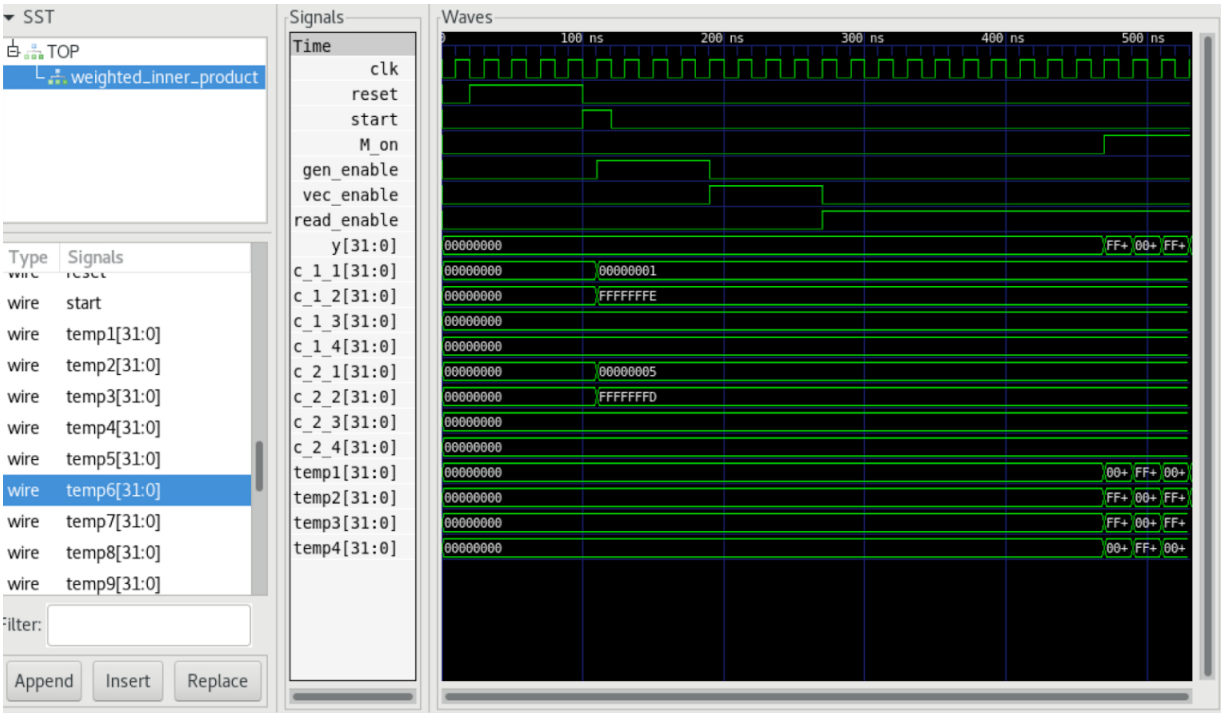

Figure 5. Simulation of Linear Transformation

```
#include <iostream>
#include <iomanip>
#include "Vkey_switching.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

// ciphertext
int c[] = { 0x1, 0x2, 0x3, 0x4,    // 0-3
                              0xffffffff, 0xfffffffe, 0xfffffffc, 0xfffffff8}; // 4-7

// bit-repr_ciphertext
int c_star[] = { 0x0, 0x0, 0x0, 0x0,       // 0-3
                 0x1, 0x0, 0x0, 0x0,                  // 4-7
                 0x1, 0x0, 0x0, 0x0,                  // 8-11
                 0x0, 0x1, 0x1, 0x0,                  // 12-15
                 0x0, 0x1, 0x0, 0x0,                  // 16-19
                 0x0, 0x0, 0x0, 0x0,                  // 20-23
                 0xffffffff, 0x0, 0x0, 0x0,           // 24-27
                 0xffffffff, 0x0, 0x0, 0x0,           // 28-31
                 0xffffffff, 0x0, 0x0, 0x0,           // 32-35
                 0xffffffff, 0x0, 0x0, 0x0};          // 36-39

// secret key
int S[2][8] = { {0x6, 0x5, 0x0, 0x3, 0x9, 0x3, 0x3, 0x6},    // 0-7
                              {0x9, 0x7, 0x6, 0x8, 0x2, 0x0, 0x6, 0x1}}; // 8-15

// l = 4
int S_star[2][32] = { {0x30, 0x18, 0xc, 0x6,    // 0-3
                0x28, 0x14, 0xa, 0x5,                 // 4-7
                0x0, 0x0, 0x0, 0x0,                   // 7-11
                0x18, 0xc, 0x6, 0x3,                  // 12-15
                0x48, 0x24, 0x12, 0x9,                 // 16-19
```

```
                0x18, 0xc, 0x6, 0x3,                  // 20-23
                0x18, 0xc, 0x6, 0x3,                  // 24-27
                0x30, 0x18, 0xc, 0x6},                // 28-31
                          {0x48, 0x24, 0x12, 0x9,                 // 0-3
                0x38, 0x1c, 0xe, 0x7,                 // 4-7
                0x30, 0x18, 0xc, 0x6,                 // 7-11
                0x40, 0x20, 0x10, 0x8,                // 12-15
                0x10, 0x8, 0x4, 0x2,                  // 16-19
                0x0, 0x0, 0x0, 0x0,                   // 20-23
                0x30, 0x18, 0xc, 0x6,                 // 24-27
                0x8, 0x4, 0x2, 0x1}};                 // 28-31


int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  // Treat the argument on the command-line as the place to start
  int width_v, ell_v;
  int width_m, length_m, ell_m;
  if (argc > 1 && argv[1][0] != '+') {
    width_v = atoi(argv[1]);
    ell_v = atoi(argv[2]);
    width_m = atoi(argv[3]);
    length_m = atoi(argv[4]);
    ell_m = atoi(argv[5]);
  } else {
    width_v = 8; // Default
    ell_v = 5;  // Default
    width_m = 8; // Default
    length_m = 2; // Default
    ell_m = 4;  // Default
  }

  Vkey_switching * dut = new Vkey_switching;  // Instantiate a new module

  // Enable dumping a VCD file
  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99);
  tfp->open("key_switching.vcd");

  dut->clk = 0;

  int time;
  std::cout << "width_v: " << width_v;
  std::cout << std::endl;
  std::cout << "ell_v: " << ell_v;
  std::cout << std::endl;
  std::cout << "width_m: " << width_m;
  std::cout << std::endl;
  std::cout << "length_m: " << length_m;
  std::cout << std::endl;
  std::cout << "ell_m: " << ell_m;
  std::cout << std::endl;

  // bit_repr_vector operation test
  int i = 0;
  int j = 0;
```

```
int err = 0;
int load_stamp = 100;
int start_stamp = 160;
// let run (width * ell) cycles for computations
// extra 11 cycles also needed
int read_stamp = start_stamp + width_v * ell_v * 20 + 17 * 20;
for (time = 0 ; time < 10000 ; time += 10) {
  dut->clk = ((time % 20) >= 10) ? 1 : 0; // Simulate a 50 MHz clock
  if (time == 20) dut->reset = 1; // Handle "reset" on for four cycles
  if (time == 100) {
    dut->reset = 0;
    dut->write = 1;           // Put "write" on
    dut->chipselect = 1;
  }

  // load operation type, width, ell
  // load operation type
  if (time >= load_stamp && time <= load_stamp + 20 && dut->clk == 1) {
      dut->address = 0;
      // bit_repr_vector
      dut->writedata = 0;
      std::cout <<"operation type received: " << (int) dut->writedata;
      std::cout << std::endl;
  }

  // load width
  if (time >= load_stamp + 20 && time <= load_stamp + 40 && dut->clk == 1) {
      dut->address = 1;
      dut->writedata = width_v;
      std::cout <<"width received: " << (int) dut->writedata;
      std::cout << std::endl;
  }

  // load ell
  // all loaded
  if (time >= load_stamp + 40 && time <= load_stamp + 60 && dut->clk == 1) {
      dut->address = 3;
      dut->all_loaded = 1;
      dut->writedata = ell_v;
      std::cout << "ell received: " << (int) dut->writedata;
      std::cout << std::endl;
  }

  // taking inputs for (width) cycles
  if (time >= start_stamp && time <= start_stamp + width_v * 20 && dut->clk == 1) {
      dut->address = 4;
      dut->writedata = c[i];
      std::cout << i << "-th input received: " << (int) c[i];
      std::cout << std::endl;
      i += 1;
  }

  dut->eval();      // Run the simulation for a cycle
  tfp->dump(time); // Write the VCD file for this cycle

  // compare outputs for (width * ell) cycles
  if (time >= read_stamp && time <= read_stamp + width_v * ell_v * 20 && dut->clk == 1) {
      std::cout << ' ' <<  std::setfill('0') << std::setw(0) << (int) dut->DATA_OUT;
```

```
        if (dut->DATA_OUT == c_star[j])
            std::cout << " OK";
        else {
            std::cout << " INCORRECT expected " << std::setfill('0') << (int) c_star[j];
            err += 1;
        }
        std::cout << std::endl;
        j += 1;
    }

    if (dut->DONE) {
        dut->all_loaded = 0;
        break;
    }
}

int run1_time = time + 10;

// bit_repr_matrix operation test
i = 0;
int r = 0;
err = 0;
load_stamp = run1_time + 100;
start_stamp = run1_time + 180;
// let run ((width * ell) * length) cycles for computations
// extra 16 cycles are needed in addition for computations
read_stamp = start_stamp + length_m * width_m * ell_m * 20 + 18 * 20;

for (time = run1_time; time < 10000 ; time += 10) {
    dut->clk = ((time % 20) >= 10) ? 1 : 0; // Simulate a 50 MHz clock
    if (time == run1_time + 20) dut->reset = 1; // Handle "reset" on for four cycles
    if (time == run1_time + 100) {
      dut->reset = 0;
      dut->write = 1;           // Put "write" on
      dut->chipselect = 1;
    }

    // load operation type, width, length, ell
    // load operation type
    if (time >= load_stamp && time <= load_stamp + 20 && dut->clk == 1) {
        dut->address = 0;
        // bit_repr_matrix
        dut->writedata = 1;
        std::cout <<"operation type received: " << (int) dut->writedata;
        std::cout << std::endl;
    }

    // load width
    if (time >= load_stamp + 20 && time <= load_stamp + 40 && dut->clk == 1) {
        dut->address = 1;
        dut->writedata = width_m;
        std::cout <<"width received: " << (int) dut->writedata;
        std::cout << std::endl;
    }

    // load length
    if (time >= load_stamp + 40 && time <= load_stamp + 60 && dut->clk == 1) {
        dut->address = 2;
```

```
    dut->writedata = length_m;
    std::cout <<"length received: " << (int) dut->writedata;
    std::cout << std::endl;
}

// load ell
if (time >= load_stamp + 60 && time <= load_stamp + 80 && dut->clk == 1) {
    dut->address = 3;
    dut->writedata = ell_m;
    std::cout << "ell received: " << (int) dut->writedata;
    std::cout << std::endl;
}

// take inputs for (length) rows
// for each row, take inputs for (width) cycles
if (time >= start_stamp && time <= start_stamp + width_m * length_m * 20 && dut->clk == 1)
    dut->address = 4;
    dut->all_loaded = 1;
    dut->writedata = S[r][i];
    std::cout << "input received: " << (int) S[r][i];
    std::cout << std::endl;
    i += 1;
    if (i == width_m) {
        i = 0;
        r += 1;
        if (r == length_m) {
            r = 0;
        }
    }
}

dut->eval();      // Run the simulation for a cycle
tfp->dump(time); // Write the VCD file for this cycle

// compare outputs for (length) rows
// for each row, compare outputs for (width * ell) cycles
if (time >= read_stamp && time <= read_stamp + width_m * length_m * ell_m * 20 && dut->clk
    std::cout << ' ' <<  std::setfill('0') << std::setw(0) << (int) dut->DATA_OUT;
    if (dut->DATA_OUT == S_star[r][i])
        std::cout << " OK";
    else {
        std::cout << " INCORRECT expected " << std::setfill('0') << (int) S_star[r][i];
        err += 1;
    }
    std::cout << std::endl;
    i += 1;
    if (i == width_m * ell_m) {
        i = 0;
        r += 1;
        if (r == length_m) {
            r = 0;
        }
    }
}

if (time >= read_stamp && dut->DONE) {
    dut->all_loaded = 0;
    break;
```

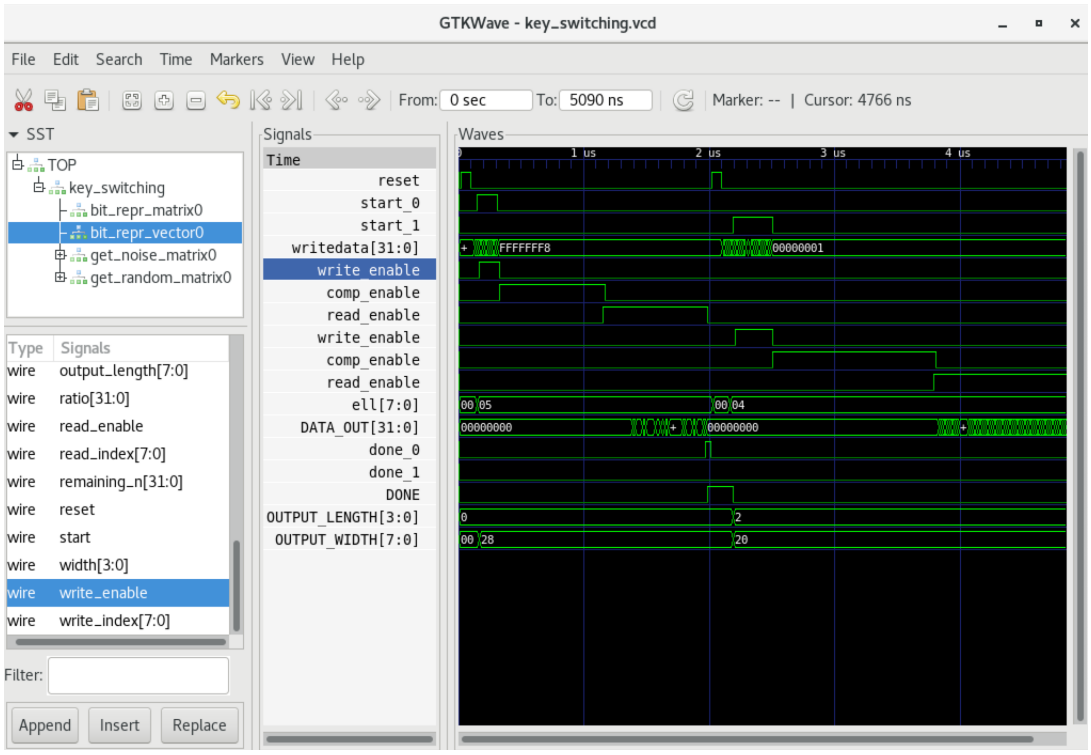

Figure 6. Simulation of Key-Switching Operations

```
    }
  }

  tfp->close(); // Stop dumping the VCD file
  delete tfp;

  dut->final(); // Stop the simulation
  delete dut;

  return 0;
}
```

And the resulted waveforms are shown in Figure 6, where the outputs agree with the ground truths.

E. Random and Noise Matrix Generation

In this project, random matrix of up to 8 by 8 in size with 16-bit integer entries can be generated with `get_random_matrix.sv`. The underlying mechanism is a pseudo-random number generator implemented with `lfsr.sv` that takes 16-bit seeds and generates 16-bit random outputs. Another version of this module with entries of smaller magnitudes (4-bit integers), is also implemented with `lfsr4.sv` in order to realize fast noise matrix generation.

A matlab script has been written in order to generate expected outputs with a given set of seeds for `lfsr.sv`. And the matlab outputs are compared with the one spitted out by these two modules to verify correctness.

The test code is:

```
1  `timescale 1ns/1ps
2  `define SD #0.010
3  `define HALF_CLOCK_PERIOD #0.90
4  `define QSIM_OUT_FN "./data/qsim.out"
5  `define MATLAB_OUT_FN "./data/lfsr.results"
6
7  module testbench();
8
9          logic clk;
10         logic resetn;
11         logic [15:0] seed;
```

```
12
13          integer lfsr_out_matlab;
14          integer lfsr_out_qsim;
15
16          logic [15:0] lfsr_out;
17
18          integer i;
19          integer ret_write;
20          integer ret_read;
21          integer qsim_out_file;
22          integer matlab_out_file;
23
24          integer error_count = 0;
25
26          lfsr lfsr_0 ( .clk(clk), .resetn(resetn), .seed(seed), .lfsr_out(lfsr_out) );
27
28          always begin
29                  `HALF_CLOCK_PERIOD;
30                  clk = ~clk;
31          end
32
33          initial begin
34                  // File IO
35                  qsim_out_file = $fopen(`QSIM_OUT_FN,"w");
36                  if (!qsim_out_file) begin
37                          $display("Couldn't␣create␣the␣output␣file.");
38                          $finish;
39                  end
40
41                  matlab_out_file = $fopen(`MATLAB_OUT_FN,"r");
42                  if (!matlab_out_file) begin
43                          $display("Couldn't␣open␣the␣Matlab␣file.");
44                          $finish;
45                  end
46
47                  // register setup
48                  clk = 0;
49                  resetn = 0;
50                  seed = 16'd1;
51                  @(posedge clk);
52
53                  @(negedge clk);   // release resetn
54                  resetn = 1;
55
56                  @(posedge clk);   // start the first cycle
57                  for (i=0 ; i<256; i=i+1) begin
58                          // compare w/ the results from Matlab sim
59                          ret_read = $fscanf(matlab_out_file, "%d", lfsr_out_matlab);
60                          lfsr_out_qsim = lfsr_out;
61
62                          $fwrite(qsim_out_file, "%0d\n", lfsr_out_qsim);
63                          if (lfsr_out_qsim != lfsr_out_matlab) begin
64                                  error_count = error_count + 1;
65                          end
66
67                          @(posedge clk);  // next cycle
68                  end
69
70                  // Any mismatch b/w rtl and matlab sims?
71                  if (error_count > 0) begin
72                          $display("The␣results␣DO␣NOT␣match␣with␣those␣from␣Matlab␣:(␣");
73                  end
74                  else begin
75                          $display("The␣results␣DO␣match␣with␣those␣from␣Matlab␣:)␣");
76                  end
77
78                  // finishing this testbench
79                  $fclose(qsim_out_file);
80                  $fclose(matlab_out_file);
```

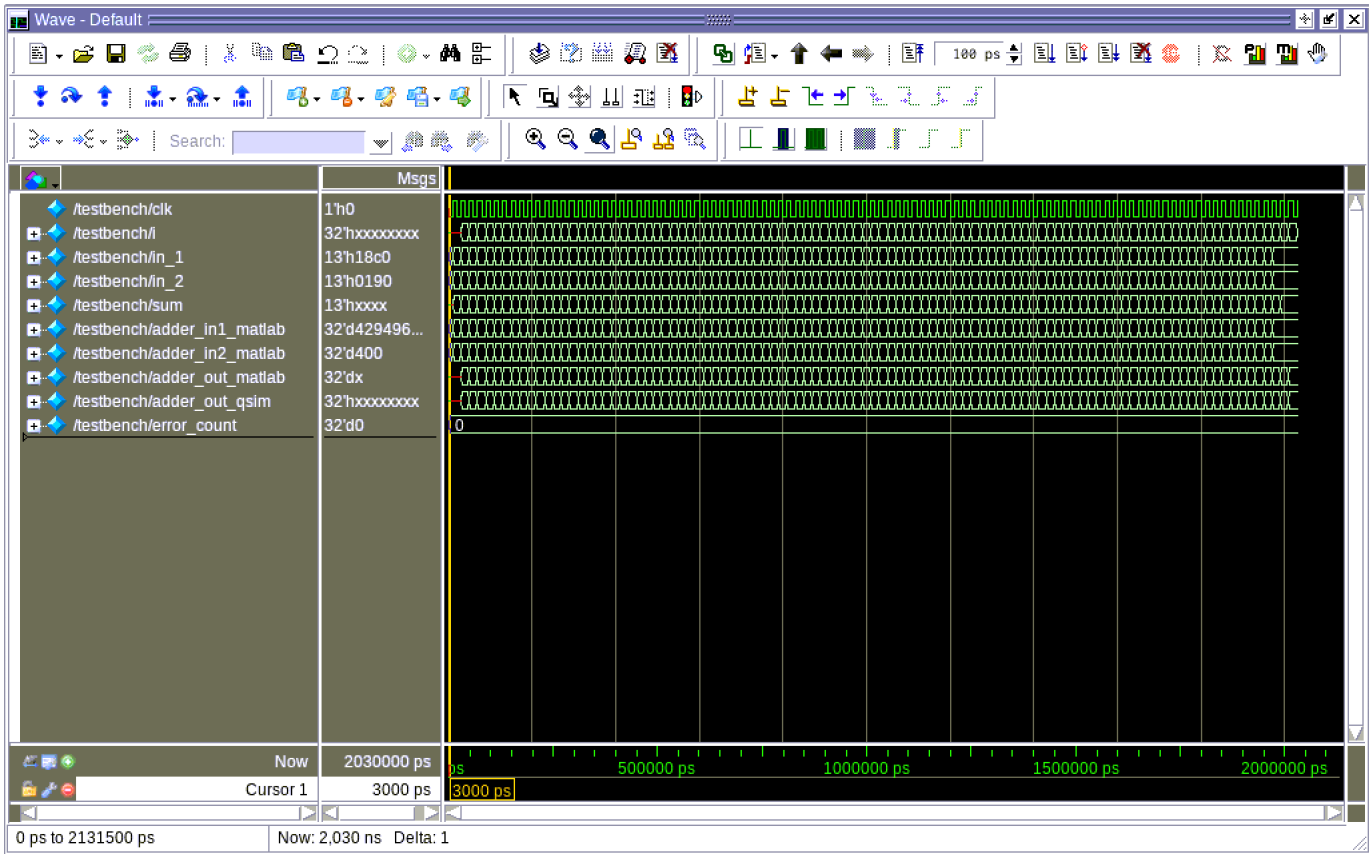

Figure 7. Simulation of LFSR and Random Number Generation

```
81                  $finish;
82          end
83
84  endmodule // testbench
```

And the resulted waveforms are shown in Figure 7, where the error count is shown to be 0.

VII. Implementation

A. *Software: User Library*

1) ***mat.h****:* Code for `mat.h` shown below.

```
//
// Created by Graves Zhang on 5/7/22.
//

#ifndef UNTITLED_MAT_H
#define UNTITLED_MAT_H

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>

struct Mat{
    double* entries;
    int row;
    int col;
};
```

```
typedef struct Mat Mat;

void showmat(Mat* A){
    if(A->row>0&&A->col>0){
        int k=0;
        printf("[");
        for(int i=1;i<=A->row;i++){
            for (int j=1;j<=A->col;j++){
                if(j<A->col){
                    printf("%f\t",A->entries[k++]);
                }else{
                    printf("%f",A->entries[k++]);
                }
            }
            if(i<A->row){
                printf("\n");
            }else{
                printf("]\n");
            }
        }
        printf("\n");
    }else{
        printf("[]");
    }
}


Mat* newmat(int r,int c,double d){
    Mat* M=(Mat*)malloc(sizeof(Mat));
    M->row=r;M->col=c;
    M->entries=(double*)malloc(sizeof(double)*r*c);
    int k=0;
    for(int i=1;i<=M->row;i++){
        for(int j=1;j<=M->col;j++){
            M->entries[k++]=d;
        }
    }
    return M;
}

void freemat(Mat* A){
    free(A->entries);
    free(A);
}

Mat* eye(int n){
    Mat* I=newmat(n,n,0);
    for(int i=1;i<=n;i++){
        I->entries[(i-1)*n+i-1]=1;
    }
    return I;
}
Mat* zeros(int r,int c){
    Mat* Z=newmat(r,c,0);
    return Z;
}
Mat* ones(int r,int c){
    Mat* O=newmat(r,c,1);
```

```
    return O;
}
Mat* randm(int r,int c,double l,double u){
    Mat* R=newmat(r,c,1);
    int k=0;
    for(int i=1;i<=r;i++){
        for(int j=1;j<=c;j++){
            double r=((double)rand())/((double)RAND_MAX);
            R->entries[k++]=l+(u-l)*r;
        }
    }
    return R;
}
double get(Mat* M,int r,int c){
    double d=M->entries[(r-1)*M->col+c-1];
    return d;
}
void set(Mat* M,int r,int c,double d){
    M->entries[(r-1)*M->col+c-1]=d;
}

Mat* scalermultiply(Mat* M,double c){
    Mat* B=newmat(M->row,M->col,0);
    int k=0;
    for(int i=0;i<M->row;i++){
        for(int j=0;j<M->col;j++){
            B->entries[k]=M->entries[k]*c;
            k+=1;
        }
    }
    return B;
}
Mat* sum(Mat* A,Mat* B){
    int r=A->row;
    int c=A->col;
    Mat* C=newmat(r,c,0);
    int k=0;
    for(int i=0;i<r;i++){
        for(int j=0;j<c;j++){
            C->entries[k]=A->entries[k]+B->entries[k];
            k+=1;
        }
    }
    return C;
}
Mat* minus(Mat* A,Mat* B){
    int r=A->row;
    int c=A->col;
    Mat* C=newmat(r,c,0);
    int k=0;
    for(int i=0;i<r;i++){
        for(int j=0;j<c;j++){
            C->entries[k]=A->entries[k]-B->entries[k];
            k+=1;
        }
    }
    return C;
}
```

```
Mat* multiply(Mat* A,Mat* B){
    int r1=A->row;
    int r2=B->row;
    int c1=A->col;
    int c2=B->col;
    if (r1==1&&c1==1){
        Mat* C=scalermultiply(B,A->entries[0]);
        return C;
    }else if (r2==1&&c2==1){
        Mat* C=scalermultiply(A,B->entries[0]);
        return C;
    }
    Mat* C=newmat(r1,c2,0);
    for(int i=1;i<=r1;i++){
        for(int j=1;j<=c2;j++){
            double de=0;
            for(int k=1;k<=r2;k++){
                de+=A->entries[(i-1)*A->col+k-1]*B->entries[(k-1)*B->col+j-1];
            }
            C->entries[(i-1)*C->col+j-1]=de;
        }
    }
    return C;
}
Mat* removerow(Mat* A,int r){
    Mat* B=newmat(A->row-1,A->col,0);
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            if(i!=r){
                B->entries[k]=A->entries[(i-1)*A->col+j-1];
                k+=1;
            }
        }
    }
    return B;
}
Mat* removecol(Mat* A,int c){
    Mat* B=newmat(A->row,A->col-1,0);
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            if(j!=c){
                B->entries[k]=A->entries[(i-1)*A->col+j-1];
                k+=1;
            }
        }
    }
    return B;
}
Mat* transpose(Mat* A){
    Mat* B=newmat(A->col,A->row,0);
    int k=0;
    for(int i=1;i<=A->col;i++){
        for(int j=1;j<=A->row;j++){
            B->entries[k]=A->entries[(j-1)*A->row+i-1];
            k+=1;
```

```
        }
    }
    return B;
}
double det(Mat* M){
    int r=M->row;
    int c=M->col;
    if(r==1&&c==1){
        double d=M->entries[0];
        return d;
    }
    Mat* M1=removerow(M,1);
    Mat* M2=newmat(M->row-1,M->col-1,0);
    double d=0, si=+1;
    for(int j=1;j<=M->col;j++){
        double c=M->entries[j-1];
        removecol2(M1,M2,j);
        d+=si*det(M2)*c;
        si*=-1;
    }
    freemat(M1);
    freemat(M2);
    return d;
}

Mat* adjoint(Mat* A){
    Mat* B=newmat(A->row,A->col,0);
    Mat* A1=newmat(A->row-1,A->col,0);
    Mat* A2=newmat(A->row-1,A->col-1,0);
    for(int i=1;i<=A->row;i++){
        removerow2(A,A1,i);
        for(int j=1;j<=A->col;j++){
            removecol2(A1,A2,j);
            double si=pow(-1,(double)(i+j));
            B->entries[(i-1)*B->col+j-1]=det(A2)*si;
        }
    }
    Mat* C=transpose(B);
    freemat(A1);
    freemat(A2);
    freemat(B);
    return C;
}

Mat* inverse(Mat* A){
    Mat* B=adjoint(A);
    double de=det(A);
    Mat* C=scalermultiply(B,1/de);
    freemat(B);
    return C;
}

Mat* copyvalue(Mat* A){
    Mat* B=newmat(A->row,A->col,0);
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            B->entries[k]=A->entries[k];
```

```
            k++;
        }
    }
    return B;
}
Mat* hconcat(Mat* A,Mat* B){
    Mat* C=newmat(A->row,A->col+B->col,0);
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            C->entries[k]=A->entries[(i-1)*A->col+j-1];
            k++;
        }
        for(int j=1;j<=B->col;j++){
            C->entries[k]=B->entries[(i-1)*B->col+j-1];
            k++;
        }
    }
    return C;
}
Mat* vconcat(Mat* A,Mat* B){
    Mat* C=newmat(A->row+B->row,A->col,0);
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            C->entries[k]=A->entries[(i-1)*A->col+j-1];
            k++;
        }
    }
    for(int i=1;i<=B->row;i++){
        for(int j=1;j<=B->col;j++){
            C->entries[k]=B->entries[(i-1)*B->col+j-1];
            k++;
        }
    }
    return C;
}
double norm(Mat* A){
    double d=0;
    int k=0;
    for(int i=1;i<=A->row;i++){
        for(int j=1;j<=A->col;j++){
            printf("computing norm... \n");
            d+=A->entries[k]*A->entries[k];
            k++;
        }
    }
    d=sqrt(d);
    return d;
}
Mat* null(Mat *A){
    Mat* RM=rowechelon(A);
    int k=RM->row;
    for(int i=RM->row;i>=1;i--){
        bool flag=false;
        for(int j=1;j<=RM->col;j++){
            if(RM->entries[(i-1)*RM->col+j-1]!=0){
                flag=true;
```

```
                break;
            }
        }
        if(flag){
            k=i;
            break;
        }
    }
    Mat* RRM=submat(RM,1,k,1,RM->col);
    freemat(RM);
    int nn=RRM->col-RRM->row;
    if(nn==0){
        Mat* N=newmat(0,0,0);
        return N;
    }
    Mat* R1=submat(RRM,1,RRM->row,1,RRM->row);
    Mat* R2=submat(RRM,1,RRM->row,1+RRM->row,RRM->col);
    freemat(RRM);
    Mat* I=eye(nn);
    Mat* T1=multiply(R2,I);
    freemat(R2);
    Mat* R3=scalermultiply(T1,-1);
    freemat(T1);
    Mat* T2=triinverse(R1);
    freemat(R1);
    Mat* X=multiply(T2,R3);
    freemat(T2);
    freemat(R3);
    Mat* N=vconcat(X,I);
    freemat(I);
    freemat(X);
    for(int j=1;j<=N->col;j++){
        double de=0;
        for(int i=1;i<=N->row;i++){
            de+=N->entries[(i-1)*N->col+j-1]*N->entries[(i-1)*N->col+j-1];
        }
        de=sqrt(de);
        for(int i=1;i<=N->row;i++){
            N->entries[(i-1)*N->col+j-1]/=de;
        }

    }
    return N;
}

double innermultiply(Mat* a,Mat* b){
    double d=0;
    int n=a->row;
    if(a->col>n){
        n=a->col;
    }
    for(int i=1;i<=n;i++){
        d+=a->entries[i-1]*b->entries[i-1];
    }
    return d;
}

#endif//UNTITLED_MAT_H
```

2) ***client_functions.c****:* Code for `client_functions.c` shown below.

3) ***server_functions.c****:* Code for `server_functions.c` shown below.

B. Software: Device Drivers and Kernel Code

1) ***key_switching.c/.h****:* Code for `key_switching.c/.h` shown below.

```
/* * Device driver for the VGA key-switchiing accelerator
 *
 * A Platform device implemented using the misc subsystem

 * Lanxiang Hu
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod key_switching.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree key_switching.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "key_switching.h"

#define DRIVER_NAME "key_switching"

/* Device registers */
#define OP_TYPE(x) ((x))
#define WIDTH(x) ((x)+4)
#define LENGTH(x) ((x)+8)
#define ELL(x) ((x)+12)
#define INPUT(x) ((x)+16)
#define OUTPUT(x) ((x)+20)
#define OUTPUT_LENGTH(x) ((x)+24)
#define OUTPUT_WIDTH(x) ((x)+28)
#define OUTPUT_DONE(x) ((x)+32)

/*
 * Information about our device
 */
```

```
struct key_switching_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
                key_switching_loading_t load_info;
                key_switching_status_t status_info;
                key_switching_output_t output_info;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_op_type(key_switching_loading_t load_info)
{
        iowrite8(load_info->input, OP_TYPE(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_width(key_switching_loading_t load_info)
{
        iowrite8(load_info->input, WIDTH(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_length(key_switching_loading_t load_info)
{
        iowrite8(load_info->input, LENGTH(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_ell(key_switching_loading_t load_info)
{
        iowrite8(load_info->input, ELL(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_input(key_switching_loading_t load_info)
{
        iowrite32(load_info->input, INPUT(dev.virtbase) );
        dev.load_info = *load_info;
}

static void set_reset(key_switching_status_t status_info)
{
        iowrite8(status_info->reset, RESET(dev.virtbase) );
        dev.status_info = *status_info;
}

static void set_all_loaded(key_switching_status_t status_info)
{
        iowrite8(status_info->all_loaded, LOADED(dev.virtbase) );
        dev.status_info = *status_info;
}

static void read_output(key_switching_output_t output_info)
{
        ioread32(OUTPUT(dev.virtbase) );
        output_info = dev.output_info;
```

```
}

static void read_length(key_switching_output_t output_info)
{
        ioread8(OUTPUT_LENGTH(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_width(key_switching_output_t output_info)
{
        ioread8(OUTPUT_WIDTH(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_done(key_switching_output_t output_info)
{
        ioread8(OUTPUT_DONE(dev.virtbase) );
        output_info = dev.output_info;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long key_switching_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        key_switching_arg_t ksa;

        switch (cmd) {
        case KEY_SWITCHING_WRITE:
                if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                write_input(&ksa.load_info);

        case KEY_SWITCHING_WRITE_OP_TYPE:
                if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                write_op_type(&ksa.load_info);
                break;

        case KEY_SWITCHING_WRITE_WIDTH:
                if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                write_width(&ksa.load_info);
                break;

        case KEY_SWITCHING_WRITE_LENGTH:
                if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                write_length(&ksa.load_info);
                break;

        case KEY_SWITCHING_WRITE_ELL:
```

```
                if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                write_ell(&ksa.load_info);
                break;

        case KEY_SWITCHING_READ:
                if (copy_to_user((key_switching_arg_t *) arg, &ska,
                                 sizeof(key_switching_arg_t)))
                        return -EACCES;
                read_output(&ksa.output_info);
                break;

        case KEY_SWITCHING_READ_WIDTH:
                if (copy_to_user((key_switching_arg_t *) arg, &ska,
                                 sizeof(key_switching_arg_t)))
                        return -EACCES;
                read_width(&ksa.output_info);
                break;

        case KEY_SWITCHING_READ_LENGTH:
                if (copy_to_user((key_switching_arg_t *) arg, &ska,
                                 sizeof(key_switching_arg_t)))
                        return -EACCES;
                read_length(&ksa.output_info);
                break;

        case KEY_SWITCHING_READ_DONE:
                if (copy_to_user((key_switching_arg_t *) arg, &ska,
                                 sizeof(key_switching_arg_t)))
                        return -EACCES;
                read_done(&ksa.output_info);
                break;

        case KEY_SWITCHING_WRITE_RESET:
                  if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                set_reset(&ksa.status_info);
                break;

        case KEY_SWITCHING_WRITE_LOADED:
                  if (copy_from_user(&ksa, (key_switching_arg_t *) arg,
                                   sizeof(key_switching_arg_t)))
                        return -EACCES;
                set_all_loaded(&ksa.status_info);
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations key_switching_fops = {
        .owner                = THIS_MODULE,
```

```
        .unlocked_ioctl = key_switching_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice key_switching_misc_device = {
        .minor                = MISC_DYNAMIC_MINOR,
        .name                 = DRIVER_NAME,
        .fops                 = &key_switching_fops,
};

/*
 * Initialization code: get resources (registers) and have the accelerator ready
 */
static int __init key_switching_probe(struct platform_device *pdev)
{
        key_switching_status_t load_reset = {0x1, 0x0, 0x0};
        int ret;

        /* Register ourselves as a misc device: creates /dev/key_switching */
        ret = misc_register(&key_switching_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* reset */
    loading_input(&load_reset);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&key_switching_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int key_switching_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
```

```
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&key_switching_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id key_switching_of_match[] = {
        { .compatible = "csee4840-sonny-hu,key_switching_1" },
        {},
};
MODULE_DEVICE_TABLE(of, key_switching_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver key_switching_driver = {
        .driver        = {
                .name         = DRIVER_NAME,
                .owner         = THIS_MODULE,
                .of_match_table = of_match_ptr(key_switching_of_match),
        },
        .remove        = __exit_p(key_switching_remove),
};

/* Called when the module is loaded: set things up */
static int __init key_switching_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&key_switching_driver, key_switching_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit key_switching_exit(void)
{
        platform_driver_unregister(&key_switching_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(key_switching_init);
module_exit(key_switching_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Lanxiang Hu");
MODULE_DESCRIPTION("key-switchiing accelerator");

#ifndef _KEY_SWITCHING_H
#define _KEY_SWITCHING_H

#include <linux/ioctl.h>

#define WIDTH_LIMIT 256
#define LENGTH_LIMIT 8
#define L_LIMIT 32

typedef struct {
  unsigned char op_type;
  unsigned char width;
  unsigned char length;
```

```
  unsigned char ell;
  int input;
} key_switching_loading_t;

typedef struct {
  unsigned char reset;
  unsigned char done;
  unsigned char all_loaded;
} key_switching_status_t;

typedef struct {
  int output;
  char output_length;
  char output_width;
  char done;
} key_switching_output_t;

typedef struct {
  key_switching_loading_t load_info;
  key_switching_status_t status_info;
  key_switching_output_t output_info;
} key_switching_arg_t;

#define KEY_SWITCHING_MAGIC 'q'

/* ioctls and their arguments */
#define KEY_SWITCHING_WRITE _IOW(KEY_SWITCHING_MAGIC, 1, key_switching_arg_t *)
#define KEY_SWITCHING_WRITE_OP_TYPE _IOW(KEY_SWITCHING_MAGIC, 2, key_switching_arg_t *)
#define KEY_SWITCHING_WRITE_WIDTH _IOW(KEY_SWITCHING_MAGIC, 3, key_switching_arg_t *)
#define KEY_SWITCHING_WRITE_LENGTH _IOW(KEY_SWITCHING_MAGIC, 4, key_switching_arg_t *)
#define KEY_SWITCHING_WRITE_ELL _IOW(KEY_SWITCHING_MAGIC, 5, key_switching_arg_t *)
#define KEY_SWITCHING_READ  _IOR(KEY_SWITCHING_MAGIC, 6, key_switching_arg_t *)
#define KEY_SWITCHING_READ_WIDTH _IOR(ENCRYPTED_DOMAIN_MAGIC, 7, encrypted_domain_arg_t *)
#define KEY_SWITCHING_READ_LENGTH _IOR(ENCRYPTED_DOMAIN_MAGIC, 8, encrypted_domain_arg_t *)
#define KEY_SWITCHING_READ_DONE _IOR(ENCRYPTED_DOMAIN_MAGIC, 9, encrypted_domain_arg_t *)
#define KEY_SWITCHING_WRITE_RESET _IOW(ENCRYPTED_DOMAIN_MAGIC, 10, encrypted_domain_arg_t *)
#define KEY_SWITCHING_WRITE_LOADED _IOW(ENCRYPTED_DOMAIN_MAGIC, 11, encrypted_domain_arg_t *)

#endif
```

2) ***encrypted_domain.c/.h****:* Code for `encrypted_domain.c/.h` shown below.

```
/* * Device driver for the VGA encrypted-domain accelerator
 *
 * A Platform device implemented using the misc subsystem

 * Lanxiang Hu
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod encrypted_domain.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree encrypted_domain.c
```

```
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "encrypted_domain.h"

#define DRIVER_NAME "encrypted_domain"

/* Device registers */
#define OP_TYPE(x) ((x))
#define DATA_TYPE(x) ((x+4))
#define WIDTH(x) ((x)+8)
#define LENGTH(x) ((x)+12)
#define INPUT(x) ((x)+16)
#define OUTPUT_0(x) ((x)+20)
#define OUTPUT_1(x) ((x)+24)
#define OUTPUT_2(x) ((x)+28)
#define OUTPUT_3(x) ((x)+32)
#define OUTPUT_4(x) ((x)+36)
#define OUTPUT_5(x) ((x)+40)
#define OUTPUT_6(x) ((x)+44)
#define OUTPUT_8(x) ((x)+48)
#define OUTPUT_9(x) ((x)+52)
#define OUTPUT_10(x) ((x)+56)
#define OUTPUT_11(x) ((x)+60)
#define OUTPUT_12(x) ((x)+64)
#define OUTPUT_13(x) ((x)+68)
#define OUTPUT_14(x) ((x)+72)
#define OUTPUT_15(x) ((x)+76)
#define OUTPUT_LENGTH(x) ((x)+80)
#define OUTPUT_WIDTH(x) ((x)+84)
#define OUTPUT_DONE(x) ((x)+88)
#define OUTPUT_RESET(x) ((x)+92)
#define OUTPUT_LOADED(x) ((x)+96)
#define W(x) ((x)+100)


/*
 * Information about our device
 */
struct encrypted_domain_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
                encrypted_domain_loading_t load_info;
                encrypted_domain_status_t status_info;
                encrypted_domain_output_t output_info;
} dev;
```

```
/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_op_type(encrypted_domain_loading_t load_info)
{
        iowrite8(load_info->input, OP_TYPE(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_op_type(encrypted_domain_loading_t load_info)
{
        iowrite8(load_info->data_type, DATA_TYPE(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_width(encrypted_domain_loading_t load_info)
{
        iowrite8(load_info->input, WIDTH(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_length(encrypted_domain_loading_t load_info)
{
        iowrite8(load_info->input, LENGTH(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_ell(encrypted_domain_loading_t load_info)
{
        iowrite8(load_info->input, ELL(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_w(encrypted_domain_loading_t load_info)
{
        iowrite32(load_info->w, INPUT(dev.virtbase) );
        dev.load_info = *load_info;
}

static void write_input(encrypted_domain_loading_t load_info)
{
        iowrite32(load_info->input, INPUT(dev.virtbase) );
        dev.load_info = *load_info;
}

static void set_reset(encrypted_domain_status_t status_info)
{
        iowrite8(status_info->reset, RESET(dev.virtbase) );
        dev.status_info = *status_info;
}

static void set_all_loaded(encrypted_domain_status_t status_info)
{
        iowrite8(status_info->all_loaded, RESET(dev.virtbase) );
        dev.status_info = *status_info;
}
```

```
static void read_output_0(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_0(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_1(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_1(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_2(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_2(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_3(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_3(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_4(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_4(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_5(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_5(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_6(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_6(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_7(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_7(dev.virtbase) );
        output_info = dev.output_info;
}


static void read_output_8(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_8(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_9(encrypted_domain_output_t output_info)
{
```

```
        ioread32(OUTPUT_9(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_10(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_10(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_11(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_11(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_12(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_12(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_13(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_13(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_14(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_14(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_output_15(encrypted_domain_output_t output_info)
{
        ioread32(OUTPUT_15(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_length(encrypted_domain_output_t output_info)
{
        ioread8(OUTPUT_LENGTH(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_width(encrypted_domain_output_t output_info)
{
        ioread8(OUTPUT_WIDTH(dev.virtbase) );
        output_info = dev.output_info;
}

static void read_done(encrypted_domain_output_t output_info)
{
        ioread8(OUTPUT_DONE(dev.virtbase) );
        output_info = dev.output_info;
}
```

```
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long encrypted_domain_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        encrypted_domain_arg_t ksa;

        switch (cmd) {
        case ENCRYPTED_DOMAIN_WRITE:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                   sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                loading_input(&ksa.load_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_OP_TYPE:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                   sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                write_op_type(&ksa.load_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_DATA_TYPE:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                   sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                write_data_type(&ksa.load_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_WIDTH:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                   sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                write_width(&ksa.load_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_LENGTH:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                   sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                write_length(&ksa.load_info);
                break;


        case ENCRYPTED_DOMAIN_READ_0:
                if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                                 sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                read_output_0(&ksa.output_info);
                break;

        case ENCRYPTED_DOMAIN_READ_1:
                if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                                 sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                read_output_1(&ksa.output_info);
```

```
        break;

case ENCRYPTED_DOMAIN_READ_2:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_2(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_3:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_3(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_4:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_4(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_5:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_5(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_6:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_6(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_7:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_7(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_8:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_8(&ska.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_9:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                        sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_9(&ska.output_info);
        break;
```

```
case ENCRYPTED_DOMAIN_READ_10:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_10(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_11:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_11(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_12:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_12(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_13:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_13(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_14:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_14(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_15:
        if (copy_to_user((encrypted_domain_arg_t *) arg, &ska,
                         sizeof(encrypted_domain_arg_t)))
                return -EACCES;
        read_output_15(&ksa.output_info);
        break;

        case ENCRYPTED_DOMAIN_READ_WIDTH:
        if (copy_to_user((key_switching_arg_t *) arg, &ska,
                         sizeof(key_switching_arg_t)))
                return -EACCES;
        read_width(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_LENGTH:
        if (copy_to_user((key_switching_arg_t *) arg, &ska,
                         sizeof(key_switching_arg_t)))
                return -EACCES;
        read_length(&ksa.output_info);
        break;

case ENCRYPTED_DOMAIN_READ_DONE:
        if (copy_to_user((key_switching_arg_t *) arg, &ska,
```

```
                                sizeof(key_switching_arg_t)))
                        return -EACCES;
                read_done(&ksa.output_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_RESET:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                set_reset(&ksa.status_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_LOADED:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                set_all_loaded(&ksa.status_info);
                break;

        case ENCRYPTED_DOMAIN_WRITE_W:
                if (copy_from_user(&ksa, (encrypted_domain_arg_t *) arg,
                                sizeof(encrypted_domain_arg_t)))
                        return -EACCES;
                write_w(&ksa.status_info);
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations encrypted_domain_fops = {
        .owner                = THIS_MODULE,
        .unlocked_ioctl = encrypted_domain_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice encrypted_domain_misc_device = {
        .minor                = MISC_DYNAMIC_MINOR,
        .name                = DRIVER_NAME,
        .fops                = &encrypted_domain_fops,
};

/*
 * Initialization code: get resources (registers) and have the accelerator ready
 */
static int __init encrypted_domain_probe(struct platform_device *pdev)
{
        encrypted_domain_status_t load_reset = {0x1, 0x0, 0x0};
        int ret;

        /* Register ourselves as a misc device: creates /dev/encrypted_domain */
        ret = misc_register(&encrypted_domain_misc_device);

        /* Get the address of our registers from the device tree */
```

```
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* reset */
    loading_input(&load_reset);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&encrypted_domain_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int encrypted_domain_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&encrypted_domain_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id encrypted_domain_of_match[] = {
        { .compatible = "csee4840-sonny-hu,encrypted_domain_1" },
        {},
};
MODULE_DEVICE_TABLE(of, encrypted_domain_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver encrypted_domain_driver = {
        .driver        = {
                .name         = DRIVER_NAME,
                .owner         = THIS_MODULE,
                .of_match_table = of_match_ptr(encrypted_domain_of_match),
        },
        .remove        = __exit_p(encrypted_domain_remove),
```

```
};

/* Called when the module is loaded: set things up */
static int __init encrypted_domain_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&encrypted_domain_driver, encrypted_domain_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit encrypted_domain_exit(void)
{
        platform_driver_unregister(&encrypted_domain_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(encrypted_domain_init);
module_exit(encrypted_domain_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Lanxiang Hu");
MODULE_DESCRIPTION("encrypted-domain accelerator");



#ifndef _ENCRYPTED_DOMAIN_H
#define _ENCRYPTED_DOMAIN_H

#include <linux/ioctl.h>

#define WIDTH_LIMIT 16
#define LENGTH_LIMIT 16

typedef struct {
  unsigned char op_type;
  unsigned char data_type;
  unsigned char width;
  unsigned char length;
  int input;
  int w;
} encrypted_domain_loading_t;

typedef struct {
  unsigned char reset;
  unsigned char done;
  unsigned char all_loaded;
} encrypted_domain_status_t;

typedef struct {
  int output_0;
  int output_1;
  int output_2;
  int output_3;
  int output_4;
  int output_5;
  int output_6;
  int output_7;
  int output_8;
```

```
  int output_9;
  int output_10;
  int output_11;
  int output_12;
  int output_13;
  int output_14;
  int output_15;
  int output;
  char output_length;
  char output_width;
  char done;
} encrypted_domain_output_t;

typedef struct {
  encrypted_domain_loading_t load_info;
  encrypted_domain_status_t status_info;
  encrypted_domain_output_t output_info;
} encrypted_domain_arg_t;

#define ENCRYPTED_DOMAIN_MAGIC 'q'

/* ioctls and their arguments */
#define ENCRYPTED_DOMAIN_WRITE _IOW(ENCRYPTED_DOMAIN_MAGIC, 1, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_WRITE_OP_TYPE _IOW(ENCRYPTED_DOMAIN_MAGIC, 2, ENCRYPTED_DOMAIN_arg_t
#define ENCRYPTED_DOMAIN_WRITE_DATA_TYPE _IOW(ENCRYPTED_DOMAIN_MAGIC, 3, ENCRYPTED_DOMAIN_arg_
#define ENCRYPTED_DOMAIN_WRITE_WIDTH _IOW(ENCRYPTED_DOMAIN_MAGIC, 4, ENCRYPTED_DOMAIN_arg_t *)
#define ENCRYPTED_DOMAIN_WRITE_LENGTH _IOW(ENCRYPTED_DOMAIN_MAGIC, 5, ENCRYPTED_DOMAIN_arg_t *
#define ENCRYPTED_DOMAIN_READ_0  _IOR(ENCRYPTED_DOMAIN_MAGIC, 6, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_1  _IOR(ENCRYPTED_DOMAIN_MAGIC, 7, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_2  _IOR(ENCRYPTED_DOMAIN_MAGIC, 8, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_3  _IOR(ENCRYPTED_DOMAIN_MAGIC, 9, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_4  _IOR(ENCRYPTED_DOMAIN_MAGIC, 10, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_5  _IOR(ENCRYPTED_DOMAIN_MAGIC, 11, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_6  _IOR(ENCRYPTED_DOMAIN_MAGIC, 12, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_7  _IOR(ENCRYPTED_DOMAIN_MAGIC, 13, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_8  _IOR(ENCRYPTED_DOMAIN_MAGIC, 14, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_9  _IOR(ENCRYPTED_DOMAIN_MAGIC, 15, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_10  _IOR(ENCRYPTED_DOMAIN_MAGIC, 16, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_11  _IOR(ENCRYPTED_DOMAIN_MAGIC, 17, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_12  _IOR(ENCRYPTED_DOMAIN_MAGIC, 18, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_13  _IOR(ENCRYPTED_DOMAIN_MAGIC, 19, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_14  _IOR(ENCRYPTED_DOMAIN_MAGIC, 20, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_15  _IOR(ENCRYPTED_DOMAIN_MAGIC, 21, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_WIDTH _IOR(ENCRYPTED_DOMAIN_MAGIC, 22, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_READ_LENGTH _IOR(ENCRYPTED_DOMAIN_MAGIC, 23, encrypted_domain_arg_t *
#define ENCRYPTED_DOMAIN_READ_DONE _IOR(ENCRYPTED_DOMAIN_MAGIC, 24, encrypted_domain_arg_t *)
#define ENCRYPTED_DOMAIN_WRITE_RESET _IOW(ENCRYPTED_DOMAIN_MAGIC, 25, encrypted_domain_arg_t *
#define ENCRYPTED_DOMAIN_WRITE_LOADED _IOW(ENCRYPTED_DOMAIN_MAGIC, 26, encrypted_domain_arg_t
#define ENCRYPTED_DOMAIN_WRITE_W _IOW(ENCRYPTED_DOMAIN_MAGIC, 27, encrypted_domain_arg_t *)

#endif
```

C. *Software: Integer Vector Homomorphic Scheme Demonstration*

1) ***client_server.c****:* Code for `client_server.c` shown below.

```
/*
 * Userspace program that communicates with the VHE accelerator
 * through ioctls
```

```
 *
 * Acknowledgement (modified from):
 * Stephen A. Edwards
 * Columbia University
 * Lanxiang Hu, Liqin Zhang
 * May, 2022
 *
 */

#include <stdio.h>
#include "switching_matrix.h"
#include "encrypted_domain.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <stdio.h>

#include "limits.h"

#include "./library/mat.h"
#include "./client_functions.c"
#include "./server_functions.c"
#define M_1 4
#define N_1 4
#define M_2 2
#define N_2 2
#define L_1 4
#define L_2 8

int switching_matrix_fd;
int encrypted_domain_fd;

int weight;

// ciphertext
int c[] = { 0x1, 0x2, 0x3, 0x4,   // 0-3
                            0xffffffff, 0xfffffffe, 0xfffffffc, 0xfffffff8}; // 4-7

// bit-repr_ciphertext
int c_star[] = { 0x0, 0x0, 0x0, 0x0,      // 0-3
                 0x1, 0x0, 0x0, 0x0,                // 4-7
                 0x1, 0x0, 0x0, 0x0,                // 8-11
                 0x0, 0x1, 0x1, 0x0,                // 12-15
                 0x0, 0x1, 0x0, 0x0,                // 16-19
                 0x0, 0x0, 0x0, 0x0,                // 20-23
                 0xffffffff, 0x0, 0x0, 0x0,         // 24-27
                 0xffffffff, 0x0, 0x0, 0x0,         // 28-31
                 0xffffffff, 0x0, 0x0, 0x0,         // 32-35
                 0xffffffff, 0x0, 0x0, 0x0};        // 36-39

// secret key
int S[2][8] = { {0x6, 0x5, 0x0, 0x3, 0x9, 0x3, 0x3, 0x6},   // 0-7
                           {0x9, 0x7, 0x6, 0x8, 0x2, 0x0, 0x6, 0x1}}; // 8-15
```

```
// l = 4
int S_star[2][32] = { {0x30, 0x18, 0xc, 0x6,   // 0-3
                0x28, 0x14, 0xa, 0x5,                // 4-7
                0x0, 0x0, 0x0, 0x0,                  // 7-11
                0x18, 0xc, 0x6, 0x3,                 // 12-15
                0x48, 0x24, 0x12, 0x9,                // 16-19
                0x18, 0xc, 0x6, 0x3,                 // 20-23
                0x18, 0xc, 0x6, 0x3,                 // 24-27
                0x30, 0x18, 0xc, 0x6},               // 28-31
                          {0x48, 0x24, 0x12, 0x9,                // 0-3
                0x38, 0x1c, 0xe, 0x7,                // 4-7
                0x30, 0x18, 0xc, 0x6,                // 7-11
                0x40, 0x20, 0x10, 0x8,               // 12-15
                0x10, 0x8, 0x4, 0x2,                 // 16-19
                0x0, 0x0, 0x0, 0x0,                  // 20-23
                0x30, 0x18, 0xc, 0x6,                // 24-27
                0x8, 0x4, 0x2, 0x1}};                // 28-31

// encrypted data
int c_1[16] = {0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8,   // 0-7
                           0xffffffff, 0xfffffffe, 0xfffffffd, 0xfffffffc, 0xfffffffb, 0xffff

int c_2[16] = {0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,   // 0-7
                           0x1, 0x2, 0x3, 0x4, 0xffffffff, 0xfffffffe, 0xfffffffd, 0xfffffffc

// expected result
int c_out[16] = {0x3, 0x5, 0x7, 0x9, 0xb, 0xd, 0xf, 0x11,   // 0-7
                           0x0, 0x0, 0x0, 0x0, 0xfffffffa, 0xfffffff8, 0xfffffff6, 0xfffffff4

// encrypted data
int c[8] = {0x1, 0x2, 0x3, 0x4, 0xffffffff, 0xfffffffe, 0xfffffffd, 0xfffffffc};   // 0-7

// encrypted linear operator
int M[4][8] = { {0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8},   // 0-7
                           {0xfffffffe, 0xfffffffd, 0xfffffffc, 0xfffffffb, 0xfffffffa, 0xfff
                {0x3, 0x6, 0x9, 0xc, 0xfffffffd, 0xfffffffa, 0xfffffff7, 0xfffffff4},   // 16-
                {0xfffffffe, 0xfffffffd, 0xfffffffc, 0xfffffffb, 0x6, 0x7, 0x8, 0x9}   // 24-3
               };

// result
int a[4] = {0xffffffd8, 0x28, 0xb4, 0xffffff88};    // 0-4

// encrypted data
int c_1_out[2] = {0x1, 0xfffffffe};   // 0-1

int c_2_out[2] = {0x5, 0xfffffffd};   // 0-1

// encrypted linear operator
int M_wi[4][4] = { {0x1, 0x2, 0x3, 0x4},   // 0-7
                           {0xfffffffe, 0xfffffffd, 0xfffffffc, 0xfffffffb},   // 8-15
                {0x3, 0x6, 0x9, 0xc},   // 16-23
                {0xfffffffc, 0xfffffffb, 0x9, 0xc}   // 24-31
               };

// reminder: vectorized result
int x[4] = {0x5, 0xfffffffd, 0xfffffff6, 0x6};    // 0-4

int w = 1;
```

```
// expected result
int a_out[4] = {0xfffffff9, 0x9, 0xffffffeb, 0xffffffe9};    // 0-4


int width_v, ell_v;
int width_m, length_m, ell_m;

int width_vd;
int width_l, length_l;
int width_w, length_w;

width_v = 8; // Default
ell_v = 5;
width_m = 8;
length_m = 2;
ell_m = 4;

width_vd = 16; // Default
width_l = 8;
length_l = 4;
width_w = 2;
length_w = 4;

int main() {
    key_switching_arg_t ksa;
    encrypted_domain_arg_t eda;

    printf("VHE Userspace program started\n");

    static const char filename[] = "/dev/key_switching";
    printf("key_switching device driver started\n");

    if ( (key_switching_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

    static const char filename[] = "/dev/encrypted_domain";

    printf("encrypted_domain device driver started\n");

    if ( (encrypted_domain_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

    srand(time(0));
    weight = rand() % 10;

    Mat *I = eye(M);
    Mat *S = scalermultiply(I, weight);
    Mat *x_1 = get_random_matrix(M, 1, INT_MIN>>26, INT_MAX>>26); // initialized variable rang
    Mat *x_2 = get_random_matrix(M, 1, INT_MIN>>26, INT_MAX>>26); // initialized variable rang
    printf("********** Client: Plaintext 1 ************\n");
    showmat(x_1);
    printf("********** Client: Plaintext 2 ************\n");
    showmat(x_2);
```

```
    Mat *c_1 = encryption(S, x_1); // ciphertext
    Mat *c_2 = encryption(S, x_2); // ciphertext
    printf("********** Client -> Server: Encrypted Text 1 ************\n");
    showmat(c_1);
    printf("********** Client -> Server: Encrypted Text 2 ************\n");
    showmat(c_2);
    printf("********** Server: Encrypted Addition ************\n");
    Mat *s_1 = serverAddition(c_1, c_2);
    showmat(s_1);
    printf("********** Server->Client: Decrypted Addition ************\n");
    Mat *addi = serverAddition(x_1, x_2);
    showmat(addi);
    printf("************** Linear Transformation ***************\n");
    printf("********** Clinet: Linear Transformation Key Switching Matrix ************\n");
    Mat *G = get_random_matrix(M, N, INT_MIN>>24, INT_MAX>>24);
    Mat *KSM = clientLinearTransform(G, S, c_1);
    showmat(KSM);
    printf("********** Server: Encrypted Linear Transform, (M_l, c_1) ************\n");
    Mat *s_2 = serverLinearTransform(KSM, c_1);
    showmat(s_2);
    printf("********** Server->Client: Decrypted Linear Transform ************\n");
    Mat *M_l = get_random_matrix(M+5, 1, INT_MIN>>24, INT_MAX>>24); // initialized variable ra
    Mat *delin = serverLinearTransform(M_l, x_1);
    showmat(delin);
    printf("********** Server: Encrypted Weighted Inner Product ************\n");
    Mat *M_i = get_random_matrix(M, M, INT_MIN>>24, INT_MAX>>24); // initialized variable rang
    Mat *s_3 = serverInnerProduct(c_1, transpose(c_2), M_i);
    showmat(s_3);
    printf("********** Server->Client: Decrypted Weighted Inner Product ************\n");
    Mat *dewin = serverInnerProduct(x_1, transpose(x_2), M_i);
    showmat(dewin);
    printf("********** Client: Decrypted Plaintext 1 ************\n");
    Mat *xx_1 = decryption(S, c_1);
    showmat(xx_1);
    printf("********** Client: Decrypted Plaintext 2 ************\n");
    Mat *xx_2 = decryption(S, c_2);
    showmat(xx_2);

}
```

D. *Hardware: Key Switching Unit*

*1) **Top-level: `key_switching.sv`**:* Code for `key_switching.sv` shown below.

```
// CSEE 4840 Project: Choose and Run One of the Three Key-Switching Operations
// Avalon memory-mapped peripheral that generates accelerate key_switchiing operations
//
// Spring 2022
//
// By: Lanxiang Hu
// Uni: lh3116

module key_switching (input logic     clk,          // 50MHz clock
              input logic             reset,
                input logic signed [31:0]      writedata,
                input logic           write,
                input logic                   chipselect,
                input logic [3:0]       address,

            input logic             all_loaded,

            output logic signed [31:0]     DATA_OUT,
```

```
19
20              output logic [3:0]      OUTPUT_LENGTH, // at most 8
21              output logic [7:0]      OUTPUT_WIDTH,  // at most 256
22              output logic            DONE);
23
24          logic [3:0] operation;
25          logic start_0;
26          logic start_1;
27          logic start_2;
28          logic start_3;
29          logic [3:0]  width; // n <= 8
30          logic [3:0]  length; // n <= 8
31          logic [31:0] in;     // field for c_i and S_ij
32          logic [7:0]  ell;
33          logic [7:0]  output_length_0; // at most 256, same as vector 'width'
34          logic [3:0]  output_length_1;
35          logic [7:0]  output_width_1;  // at most 256
36          logic signed [31:0] data_out_0;
37          logic signed [31:0] data_out_1;
38          logic signed [31:0] data_out_2;
39          logic signed [31:0] data_out_3;
40          logic done_0;
41          logic done_1;
42          logic done_2;
43          logic done_3;
44
45          const logic [3:0] load_op_type = 4'h0;
46          const logic [3:0] load_width = 4'h1;
47          const logic [3:0] load_length = 4'h2;
48          const logic [3:0] load_ell = 4'h3;
49          const logic [3:0] load_input = 4'h4;
50
51          const logic [3:0] bit_repr_vector = 4'h0;
52          const logic [3:0] bit_repr_matrix = 4'h1;
53          const logic [3:0] get_random_matrix = 4'h2;
54          const logic [3:0] get_noise_matrix = 4'h3;
55
56          // instantiate the three modules
57          bit_repr_vector bit_repr_vector0(.clk(clk),
58                           .reset(reset),
59                           .start(start_0),
60                           .width(width),
61                           .c_i(in),
62                           .ell(ell),
63
64                           .output_length(output_length_0),
65                           .data_out(data_out_0),    // prev memory or S_star_ij
66                           .done(done_0));
67
68          bit_repr_matrix bit_repr_matrix0(.clk(clk),
69                           .reset(reset),
70                           .start(start_1),
71                           .width(width), // n <= 8
72                           .length(length), // n <= 8
73                           .S_ij(in),
74                           .ell(ell),
75
76                           .output_length(output_length_1),      // at most 8
77                           .output_width(output_width_1),        // at most 256
78                           .data_out(data_out_1),
79                           .done(done_1));
80
81          get_random_matrix get_random_matrix0(.clk(clk),
82                           .reset(reset),
83                           .start(start_2),
84                           .length(length), // n <= 8
85                           .width(width),  // n <= 8
86
87                           .data_out(data_out_2),
```

```
                    .done(done_2));

    get_noise_matrix get_noise_matrix0(.clk(clk),
                    .reset(reset),
                    .start(start_3),
                    .length(length), // n <= 8
                    .width(width),  // n <= 8

                    .data_out(data_out_3),
                    .done(done_3));

    always_ff @(posedge clk) begin
          if (reset) begin
                operation <= 0;
                width <= 0;
                length <= 0;
                in <= 0;
                ell <= 0;
                done_0 <= 0;
                done_1 <= 0;
                done_2 <= 0;
                done_3 <= 0;

          end else if (chipselect && write) begin
                case (address)
                      load_op_type : operation <= writedata[3:0];
                      load_width : width <= writedata[3:0];
                      load_length : length <= writedata[3:0];
                      load_ell : ell <= writedata[7:0];
                      load_input : in <= writedata;
                endcase
          end
    end

    logic [3:0]  counter_0;
    logic [6:0]  counter_1;
    logic [6:0]  counter_2;
    logic [6:0]  counter_3;
    // workflow: load operation type, width, length, ell, inputs sequentially
    // after been properly loaded, "all_loaded" turns true
    always_ff @(posedge clk) begin
          // computation stage
          if (all_loaded) begin
                case (operation)
                      bit_repr_vector : begin
                            if (counter_0 < width) begin
                                  start_0 <= 1;
                                  counter_0 <= counter_0 + 1;
                            end else begin
                                  start_0 <= 0;
                            end
                            DONE <= done_0;
                            DATA_OUT <= data_out_0;
                            OUTPUT_WIDTH <= output_length_0;
                      end
                      bit_repr_matrix : begin
                            if (counter_1 < width * length) begin
                                  start_1 <= 1;
                                  counter_1 <= counter_1 + 1;
                            end else begin
                                  start_1 <= 0;
                            end
                            DONE <= done_1;
                            DATA_OUT <= data_out_1;
                            OUTPUT_LENGTH <= output_length_1;
                            OUTPUT_WIDTH <= output_width_1;
                      end
                      get_random_matrix : begin
                            if (counter_2 < width * length) begin
```

```
                                    start_2 <= 1;
                                    counter_2 <= counter_2 + 1;
                              end else begin
                                    start_2 <= 0;
                              end
                              DONE <= done_2;
                              DATA_OUT <= data_out_2;
                        end
                        get_random_matrix : begin
                              if (counter_3 < width * length) begin
                                    start_3 <= 1;
                                    counter_3 <= counter_3 + 1;
                              end else begin
                                    start_3 <= 0;
                              end
                              DONE <= done_3;
                              DATA_OUT <= data_out_3;
                        end
                  endcase
            end else begin
                  counter_0 <= 0;
                  counter_1 <= 0;
                  counter_2 <= 0;
                  counter_3 <= 0;

                  start_0 <= 0;
                  start_1 <= 0;
                  start_2 <= 0;
                  start_3 <= 0;
            end
      end


endmodule
```

2) ***bit_repr_vector.sv****:* Code for `bit_repr_vector.sv` shown below.

```
module bit_repr_vector(input logic   clk,
    input logic  reset,
    input logic start,
    input logic [3:0]  width, // n <= 8
    input logic signed [31:0] c_i,
    input logic [7:0] ell, // l <= 32

    output logic [7:0]    output_length, // at most 256
    output logic signed [31:0]   data_out,
    output logic done);

    logic write_enable;
    logic read_enable;
    logic comp_enable;
    logic [7:0] write_index;
    logic [7:0] comp_index;
    logic [7:0] read_index;

    integer i;
    // initialize a DMEM to store input vector.
    logic signed [31:0] input_mem [7:0]; // n <= 8
    always_ff @(posedge clk) begin
        if (reset) begin
            for(i = 0; i <= 7; i = i+1)begin
                input_mem[i] <= 0;
            end
            write_enable <= 0;
            write_index <= 0;

        end else if (start || write_enable) begin
            if (write_index[3:0] < width) begin
                write_enable <= 1;
                input_mem[write_index[2:0]] <= c_i;
```

```
34                  write_index <= write_index + 1;
35
36              end else if (write_index[3:0] == width) begin
37                  comp_enable <= 1;
38                  write_enable <= 0;
39                  write_index <= 0;
40              end
41          end else if (read_enable) begin
42              // set computational flag to false
43              comp_enable <= 0;
44          end
45      end
46
47      assign output_length = width * ell;
48
49      logic signed [31:0] n; // corresponding to n-th element in the input vector
50      logic signed [31:0] remaining_n;
51      logic [7:0] comp_input_index;
52      logic [7:0] expo_index;
53      logic [31:0] bin_factor;
54      logic signed [31:0] ratio;
55      // initialize a DMEM to store output vector.
56      logic signed [31:0] output_mem [255:0]; // n <= 256
57
58      // do computations when computational flag is true
59      always_ff @(posedge clk) begin
60          if (comp_enable) begin
61              if (comp_index < output_length) begin
62
63                  // perform bit-representation conversion for vector c
64                  // find the element to be converted to binary representation
65                  n <= input_mem[comp_input_index[2:0]];
66
67                  bin_factor <= 2**(ell-expo_index-1);
68                  // update the remining value for each input after subtraction
69
70                  if (expo_index == 0) ratio <= input_mem[comp_input_index[2:0]] / 2**(ell-
                        expo_index-1);
71                  else ratio <= remaining_n / 2**(ell-expo_index-1);
72                  if (expo_index == 0) remaining_n <= input_mem[comp_input_index[2:0]];
73                  if ((remaining_n / 2**(ell-expo_index-1)) == 1 && expo_index != 0) begin
74                      remaining_n <= remaining_n - 2**(ell-expo_index-1);
75                  end else if ((remaining_n / 2**(ell-expo_index-1)) == -1 && expo_index != 0)
                        begin
76                      remaining_n <= remaining_n + 2**(ell-expo_index-1);
77                  end
78
79                  if (expo_index == ell - 1) expo_index <= 0;
80                  else expo_index <= expo_index + 1;
81
82                  output_mem[comp_index] <= ratio;
83
84                  comp_index <= comp_index + 1;
85                  comp_input_index <= (comp_index + 1) / ell;
86
87              end else if (comp_index == output_length) begin
88                  comp_index <= 0;
89                  comp_input_index <= 0;
90                  // once finished, set read_enable to true
91                  read_enable <= 1;
92              end
93          end else if (done) begin
94              // once finished, set read_enable to true
95              read_enable <= 0;
96          end
97      end
98
99      // read each element every clock cycle
100     always_ff @(posedge clk) begin
```

```
        if (read_enable) begin
            if (read_index < output_length) begin
                // spitting out one element each cycle with appropriate index
                data_out <= output_mem[read_index];
                read_index <= read_index + 1;

            end else if (read_index == output_length) begin
                read_index <= 0;
                done <= 1;
            end
        end else begin
            done <= 0;
        end

    end
endmodule
```

3) ***bit_repr_matrix.sv****:* Code for `bit_repr_matrix.sv` shown below.

```
module bit_repr_matrix(input logic   clk,
    input logic  reset,
    input logic start,
    input logic [3:0]  width, // n <= 8
    input logic [3:0]  length, // n <= 8
    input logic signed [31:0] S_ij,
    input logic [7:0] ell, // l <= 32

    output logic [3:0]    output_length, // at most length
    output logic [7:0]    output_width, // at most 256
    output logic signed [31:0]   data_out, // prev memory or S_star_ij
    output logic done);

    logic write_enable;
    logic read_enable;
    logic comp_enable;
    logic [3:0] write_index;
    logic [7:0] comp_index;
    logic [7:0] read_index;
    logic [3:0] loaded_row_num;
    logic [3:0] comp_row_num;
    logic [3:0] spitting_row_num;

        integer i;
    // initialize a DMEM to store input vector.
    // maximum number of 8 rows are supported.
    logic signed [31:0] input_mem0 [7:0]; // n <= 8
    logic signed [31:0] input_mem1 [7:0];
    logic signed [31:0] input_mem2 [7:0];
    logic signed [31:0] input_mem3 [7:0];
    logic signed [31:0] input_mem4 [7:0];
    logic signed [31:0] input_mem5 [7:0];
    logic signed [31:0] input_mem6 [7:0];
    logic signed [31:0] input_mem7 [7:0];
    always_ff @(posedge clk) begin
        if (reset) begin
            for(i = 0; i <= 7; i = i+1)begin
                input_mem0[i] <= 0;
                input_mem1[i] <= 0;
                input_mem2[i] <= 0;
                input_mem3[i] <= 0;
                input_mem4[i] <= 0;
                input_mem5[i] <= 0;
                input_mem6[i] <= 0;
                input_mem7[i] <= 0;
            end
            write_enable <= 0;
            write_index <= 0;
            loaded_row_num <= 0;

        end else if (start || write_enable) begin
```

```
            if (write_index[3:0] < width) begin
                write_enable <= 1;

                case (loaded_row_num)
                    4'b0000: begin
                        input_mem0[write_index[2:0]] <= S_ij;
                    end
                    4'b0001: begin
                        input_mem1[write_index[2:0]] <= S_ij;
                    end
                    4'b0010: begin
                        input_mem2[write_index[2:0]] <= S_ij;
                    end
                    4'b0011: begin
                        input_mem3[write_index[2:0]] <= S_ij;
                    end
                    4'b0100: begin
                        input_mem4[write_index[2:0]] <= S_ij;
                    end
                    4'b0101: begin
                        input_mem5[write_index[2:0]] <= S_ij;
                    end
                    4'b0110: begin
                        input_mem6[write_index[2:0]] <= S_ij;
                    end
                    4'b0111: begin
                        input_mem7[write_index[2:0]] <= S_ij;
                    end
                    default: begin
                        input_mem0[write_index[2:0]] <= 0;
                    end
                endcase

                write_index <= write_index + 1;
                if (write_index + 1 == width) begin
                    write_index <= 0;
                    loaded_row_num <= loaded_row_num + 1;
                    if (loaded_row_num + 1 == length) begin
                        comp_enable <= 1;
                        write_enable <= 0;
                        loaded_row_num <= 0;
                        write_index <= 0;
                    end
                end
            end
        end else if (read_enable) begin
            // set computational flag to false
            comp_enable <= 0;
        end
    end

    assign output_width = width * ell;
    assign output_length = length;

    logic signed [31:0] n; // corresponding to n-th element in the input vector
    logic [7:0] comp_input_index;
    logic [7:0] expo_index;

    // initialize a DMEM to store output vector.
    // maximum number of 8 rows are supported.
    logic signed [31:0] output_mem0 [255:0]; // n <= 256
    logic signed [31:0] output_mem1 [255:0];
    logic signed [31:0] output_mem2 [255:0];
    logic signed [31:0] output_mem3 [255:0];
    logic signed [31:0] output_mem4 [255:0];
    logic signed [31:0] output_mem5 [255:0];
    logic signed [31:0] output_mem6 [255:0];
    logic signed [31:0] output_mem7 [255:0];

```

```
    // do computations when computational flag is true
    always_ff @(posedge clk) begin
        if (comp_enable) begin
            if (comp_index < output_width) begin

                case (comp_row_num)
                    4'b0000: begin
                        n = input_mem0[comp_input_index[2:0]];
                        output_mem0[comp_index] <= 2**(ell-expo_index-1) * n;
                    end

                    4'b0001: begin
                        n = input_mem1[comp_input_index[2:0]];
                        output_mem1[comp_index] <= 2**(ell-expo_index-1) * n;
                    end

                    4'b0010: begin
                        n = input_mem2[comp_input_index[2:0]];
                        output_mem2[comp_index] <= 2**(ell-expo_index-1) * n;
                    end

                    4'b0011: begin
                        n = input_mem3[comp_input_index[2:0]];
                        output_mem3[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                    4'b0100: begin
                        n = input_mem4[comp_input_index[2:0]];
                        output_mem4[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                    4'b0101: begin
                        n = input_mem5[comp_input_index[2:0]];
                        output_mem5[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                    4'b0110: begin
                        n = input_mem6[comp_input_index[2:0]];
                        output_mem6[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                    4'b0111: begin
                        n = input_mem7[comp_input_index[2:0]];
                        output_mem7[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                    default: begin
                        n = input_mem0[comp_input_index[2:0]];
                        output_mem0[comp_index] <= 2**(ell-expo_index-1) * n;
                    end
                endcase

                comp_index <= comp_index + 1;
                comp_input_index <= (comp_index + 1) / ell;

                if (expo_index == ell - 1) expo_index <= 0;
                else expo_index <= expo_index + 1;

                if (comp_index + 1 == output_width) begin
                    comp_index <= 0;
                    comp_input_index <= 0;
                    expo_index <= 0;
                    comp_row_num <= comp_row_num + 1;
                    if (comp_row_num + 1 == length) begin
                        comp_input_index <= 0;
                        expo_index <= 0;
                        comp_index <= 0;
                        comp_row_num <= 0;
                        // once finished, set read_enable to true
                        read_enable <= 1;
                    end
                end
            end
        end else if (done) begin
```

```
            read_enable <= 0;
        end
    end

    // read element in each row every clock cycle
    always_ff @(posedge clk) begin
        if (read_enable) begin
            if (read_index < output_width) begin

                // spitting out one element each cycle with appropriate index
                case (spitting_row_num)
                    4'b0000: begin
                        data_out <= output_mem0[read_index];
                    end

                    4'b0001: begin
                        data_out <= output_mem1[read_index];
                    end

                    4'b0010: begin
                        data_out <= output_mem2[read_index];
                    end

                    4'b0011: begin
                        data_out <= output_mem3[read_index];
                    end
                    4'b0100: begin
                        data_out <= output_mem4[read_index];
                    end
                    4'b0101: begin
                        data_out <= output_mem5[read_index];
                    end
                    4'b0110: begin
                        data_out <= output_mem6[read_index];
                    end
                    4'b0111: begin
                        data_out <= output_mem7[read_index];
                    end
                    default: begin
                        data_out <= output_mem0[read_index];
                    end
                endcase
                read_index <= read_index + 1;
                if (read_index + 1 == output_width) begin
                    read_index <= 0;
                    spitting_row_num <= spitting_row_num + 1;
                    if (spitting_row_num + 1 == output_length) begin
                        spitting_row_num <= 0;
                        read_index <= 0;
                        done <= 1;
                    end
                end
            end
        end
    end

endmodule
```

4) ***get_random_matrix.sv***: Code for `get_random_matrix.sv` shown below.

```
module get_random_matrix(input logic   clk,
    input logic  reset,
    input logic  start,
    input logic [3:0] length, // n <= 8
    input logic [3:0] width,  // n <= 8

    output logic signed [31:0] data_out,
    output logic done);

    // Notice that a random matrix generated has a maximium size of 8 by 8
```

```
    // To get larger random matices, concatenation needed on the software side

    logic gen_enable;
    logic read_enable;
    logic [3:0] gen_index;
    logic [3:0] read_index;
    logic [3:0] gen_row_num;
    logic [3:0] spitting_row_num;


    // seeds below can be modified
    logic [15:0] seed_0 = 16'd1;
    logic [15:0] seed_1 = 16'd2;
    logic [15:0] seed_2 = 16'd3;
    logic [15:0] seed_3 = 16'd4;
    logic [15:0] seed_4 = 16'd5;
    logic [15:0] seed_5 = 16'd6;
    logic [15:0] seed_6 = 16'd7;
    logic [15:0] seed_7 = 16'd8;
    // seed above can be modified

    logic signed [15:0] lfsr_out_0;
    logic signed [15:0] lfsr_out_1;
    logic signed [15:0] lfsr_out_2;
    logic signed [15:0] lfsr_out_3;
    logic signed [15:0] lfsr_out_4;
    logic signed [15:0] lfsr_out_5;
    logic signed [15:0] lfsr_out_6;
    logic signed [15:0] lfsr_out_7;

    // generate multiple LFSR instances to create a Gaussian random variable at each cycle
    // 8 16-bit LFSR
    lfsr lfsr_0( .clk(clk), .resetn(reset), .seed(seed_0), .lfsr_out(lfsr_out_0) );
    lfsr lfsr_1( .clk(clk), .resetn(reset), .seed(seed_1), .lfsr_out(lfsr_out_1) );
    lfsr lfsr_2( .clk(clk), .resetn(reset), .seed(seed_2), .lfsr_out(lfsr_out_2) );
    lfsr lfsr_3( .clk(clk), .resetn(reset), .seed(seed_3), .lfsr_out(lfsr_out_3) );
    lfsr lfsr_4( .clk(clk), .resetn(reset), .seed(seed_4), .lfsr_out(lfsr_out_4) );
    lfsr lfsr_5( .clk(clk), .resetn(reset), .seed(seed_5), .lfsr_out(lfsr_out_5) );
    lfsr lfsr_6( .clk(clk), .resetn(reset), .seed(seed_6), .lfsr_out(lfsr_out_6) );
    lfsr lfsr_7( .clk(clk), .resetn(reset), .seed(seed_7), .lfsr_out(lfsr_out_7) );

    // initialize a DMEM to store input vector.
    // maximum number of 8 rows are supported.
    logic signed [31:0] output_mem0 [7:0]; // n <= 8
    logic signed [31:0] output_mem1 [7:0];
    logic signed [31:0] output_mem2 [7:0];
    logic signed [31:0] output_mem3 [7:0];
    logic signed [31:0] output_mem4 [7:0];
    logic signed [31:0] output_mem5 [7:0];
    logic signed [31:0] output_mem6 [7:0];
    logic signed [31:0] output_mem7 [7:0];

    integer i;
    always_ff @(posedge clk) begin
        if (reset) begin
                for(i = 0; i <= 7; i = i+1)begin
                output_mem0[i] <= 0;
                output_mem1[i] <= 0;
                output_mem2[i] <= 0;
                output_mem3[i] <= 0;
                output_mem4[i] <= 0;
                output_mem5[i] <= 0;
                output_mem6[i] <= 0;
                output_mem7[i] <= 0;
            end
            gen_enable <= 0;
            read_enable <= 0;
            gen_index <= 0;
            gen_row_num <= 0;
```

```
80
81          end else if (start || gen_enable) begin
82              if (gen_index < width) begin
83                  gen_enable <= 1;
84
85                  case (gen_row_num)
86                      4'b0000: output_mem0[gen_index[2:0]][15:0] <= lfsr_out_0;
87
88                      4'b0001: output_mem1[gen_index[2:0]][15:0] <= lfsr_out_1;
89
90                      4'b0010: output_mem2[gen_index[2:0]][15:0] <= lfsr_out_2;
91
92                      4'b0011: output_mem3[gen_index[2:0]][15:0] <= lfsr_out_3;
93
94                      4'b0100: output_mem4[gen_index[2:0]][15:0] <= lfsr_out_4;
95
96                      4'b0101: output_mem5[gen_index[2:0]][15:0] <= lfsr_out_5;
97
98                      4'b0110: output_mem6[gen_index[2:0]][15:0] <= lfsr_out_6;
99
100                     4'b0111: output_mem7[gen_index[2:0]][15:0] <= lfsr_out_7;
101
102                     default: output_mem0[gen_index[2:0]][15:0] <= lfsr_out_0;
103                 endcase
104
105                 gen_index <= gen_index + 1;
106
107                 if (gen_index + 1 == width) begin
108                     gen_index <= 0;
109                     gen_row_num <= gen_row_num + 1;
110
111                     if (gen_row_num + 1 == length) begin
112                         gen_index <= 0;
113                         // once finished, set read_enable to true
114                         read_enable <= 1;
115                         // set computational flag to false
116                         gen_enable <= 0;
117                     end
118                 end
119             end
120         end else if (done) begin
121             read_enable <= 0;
122         end
123     end
124
125     // read element in each row every clock cycle
126     always_ff @(posedge clk) begin
127         if (read_enable) begin
128             if (read_index < length) begin
129
130                 // spitting out one element each cycle with appropriate index
131                 case (spitting_row_num)
132                     4'b0000: data_out <= output_mem0[read_index[2:0]];
133
134                     4'b0001: data_out <= output_mem1[read_index[2:0]];
135
136                     4'b0010: data_out <= output_mem2[read_index[2:0]];
137
138                     4'b0011: data_out <= output_mem3[read_index[2:0]];
139
140                     4'b0100: data_out <= output_mem4[read_index[2:0]];
141
142                     4'b0101: data_out <= output_mem5[read_index[2:0]];
143
144                     4'b0110: data_out <= output_mem6[read_index[2:0]];
145
146                     4'b0111: data_out <= output_mem7[read_index[2:0]];
147
148                     default: data_out <= output_mem0[read_index[2:0]];
```

```
                endcase
                read_index <= read_index + 1;
                if (read_index + 1 == width) begin
                    read_index <= 0;
                    spitting_row_num <= spitting_row_num + 1;
                    if (spitting_row_num + 1 == length) begin
                        read_index <= 0;
                        spitting_row_num <= 0;
                        done <= 1;
                    end
                end
            end
        end else begin
            done <= 0;
        end
    end

endmodule
```

5) ***get_noise_matrix.sv***: Code for `get_noise_matrix.sv` shown below.

```
module get_noise_matrix(input logic   clk,
    input logic  reset,
    input logic  start,
    input logic [3:0] length, // n <= 8
    input logic [3:0] width,  // n <= 8

    output logic signed [31:0] data_out,
    output logic done);

    // Notice that a random matrix generated has a maximium size of 8 by 8
    // To get larger random matices, concatenation needed on the software side

    logic gen_enable;
    logic read_enable;
    logic [3:0] gen_index;
    logic [3:0] read_index;
    logic [3:0] gen_row_num;
    logic [3:0] spitting_row_num;


    // seeds below can be modified
    logic [3:0] seed_0 = 4'd1;
    logic [3:0] seed_1 = 4'd2;
    logic [3:0] seed_2 = 4'd3;
    logic [3:0] seed_3 = 4'd4;
    logic [3:0] seed_4 = 4'd5;
    logic [3:0] seed_5 = 4'd6;
    logic [3:0] seed_6 = 4'd7;
    logic [3:0] seed_7 = 4'd8;
    // seed above can be modified

    logic signed [3:0] lfsr_out_0;
    logic signed [3:0] lfsr_out_1;
    logic signed [3:0] lfsr_out_2;
    logic signed [3:0] lfsr_out_3;
    logic signed [3:0] lfsr_out_4;
    logic signed [3:0] lfsr_out_5;
    logic signed [3:0] lfsr_out_6;
    logic signed [3:0] lfsr_out_7;

    // generate multiple LFSR instances to create a Gaussian random variable at each cycle
    // 8 4-bit LFSR
    lfsr4 lfsr_0( .clk(clk), .resetn(reset), .seed(seed_0), .lfsr_out(lfsr_out_0) );
    lfsr4 lfsr_1( .clk(clk), .resetn(reset), .seed(seed_1), .lfsr_out(lfsr_out_1) );
    lfsr4 lfsr_2( .clk(clk), .resetn(reset), .seed(seed_2), .lfsr_out(lfsr_out_2) );
    lfsr4 lfsr_3( .clk(clk), .resetn(reset), .seed(seed_3), .lfsr_out(lfsr_out_3) );
    lfsr4 lfsr_4( .clk(clk), .resetn(reset), .seed(seed_4), .lfsr_out(lfsr_out_4) );
    lfsr4 lfsr_5( .clk(clk), .resetn(reset), .seed(seed_5), .lfsr_out(lfsr_out_5) );
    lfsr4 lfsr_6( .clk(clk), .resetn(reset), .seed(seed_6), .lfsr_out(lfsr_out_6) );
```

```
    lfsr4 lfsr_7( .clk(clk), .resetn(reset), .seed(seed_7), .lfsr_out(lfsr_out_7) );

    // initialize a DMEM to store input vector.
    // maximum number of 8 rows are supported.
    logic signed [31:0] output_mem0 [7:0]; // n <= 8
    logic signed [31:0] output_mem1 [7:0];
    logic signed [31:0] output_mem2 [7:0];
    logic signed [31:0] output_mem3 [7:0];
    logic signed [31:0] output_mem4 [7:0];
    logic signed [31:0] output_mem5 [7:0];
    logic signed [31:0] output_mem6 [7:0];
    logic signed [31:0] output_mem7 [7:0];

    integer i;
    always_ff @(posedge clk) begin
        if (reset) begin
                for(i = 0; i <= 7; i = i+1)begin
                output_mem0[i] <= 0;
                output_mem1[i] <= 0;
                output_mem2[i] <= 0;
                output_mem3[i] <= 0;
                output_mem4[i] <= 0;
                output_mem5[i] <= 0;
                output_mem6[i] <= 0;
                output_mem7[i] <= 0;
            end
            gen_enable <= 0;
            read_enable <= 0;
            gen_index <= 0;
            gen_row_num <= 0;

        end else if (start || gen_enable) begin
            if (gen_index < width) begin
                gen_enable <= 1;

                case (gen_row_num)
                    4'b0000: output_mem0[gen_index[2:0]][3:0] <= lfsr_out_0;

                    4'b0001: output_mem1[gen_index[2:0]][3:0] <= lfsr_out_1;

                    4'b0010: output_mem2[gen_index[2:0]][3:0] <= lfsr_out_2;

                    4'b0011: output_mem3[gen_index[2:0]][3:0] <= lfsr_out_3;

                    4'b0100: output_mem4[gen_index[2:0]][3:0] <= lfsr_out_4;

                    4'b0101: output_mem5[gen_index[2:0]][3:0] <= lfsr_out_5;

                    4'b0110: output_mem6[gen_index[2:0]][3:0] <= lfsr_out_6;

                    4'b0111: output_mem7[gen_index[2:0]][3:0] <= lfsr_out_7;

                    default: output_mem0[gen_index[2:0]][3:0] <= lfsr_out_0;
                endcase

                gen_index <= gen_index + 1;

                if (gen_index + 1 == width) begin
                    gen_index <= 0;
                    gen_row_num <= gen_row_num + 1;

                    if (gen_row_num + 1 == length) begin
                        gen_index <= 0;
                        // once finished, set read_enable to true
                        read_enable <= 1;
                        // set computational flag to false
                        gen_enable <= 0;
                    end
                end
```

```
            end
        end  else if (done) begin
            read_enable <= 0;
        end
    end

    // read element in each row every clock cycle
    always_ff @(posedge clk) begin
        if (read_enable) begin
            if (read_index < length) begin

                // spitting out one element each cycle with appropriate index
                case (spitting_row_num)
                    4'b0000: data_out <= output_mem0[read_index[2:0]];

                    4'b0001: data_out <= output_mem1[read_index[2:0]];

                    4'b0010: data_out <= output_mem2[read_index[2:0]];

                    4'b0011: data_out <= output_mem3[read_index[2:0]];

                    4'b0100: data_out <= output_mem4[read_index[2:0]];

                    4'b0101: data_out <= output_mem5[read_index[2:0]];

                    4'b0110: data_out <= output_mem6[read_index[2:0]];

                    4'b0111: data_out <= output_mem7[read_index[2:0]];

                    default: data_out <= output_mem0[read_index[2:0]];
                endcase
                read_index <= read_index + 1;
                if (read_index + 1 == width) begin
                    read_index <= 0;
                    spitting_row_num <= spitting_row_num + 1;
                    if (spitting_row_num + 1 == length) begin
                        read_index <= 0;
                        spitting_row_num <= 0;
                        done <= 1;
                    end
                end
            end
        end else begin
            done <= 0;
        end
    end

endmodule
```

E. Hardware: Encrypted-Domain Computational Unit

*1) **Top-level: `encrypted_domain.sv`:*** Code for `encrypted_domain.sv` shown below.

```
// CSEE 4840 Project: Choose and Run One of the Three Encrypted-Domain Operations
// Avalon memory-mapped peripheral that generates accelerate encrypted domain operations
//
// Spring 2022
//
// By: Lanxiang Hu
// Uni: lh3116

module encrypted_domain (input logic    clk,          // 50MHz clock
            input logic                         reset,
                input logic signed [31:0]       writedata,
                input logic                     write,
                input logic                             chipselect,
                input logic [3:0]               address,

        input logic                   all_loaded,
```

```
17
18          output logic signed [31:0]     DATA_OUT_0,
19          output logic signed [31:0]     DATA_OUT_1,
20          output logic signed [31:0]     DATA_OUT_2,
21          output logic signed [31:0]     DATA_OUT_3,
22          output logic signed [31:0]     DATA_OUT_4,
23          output logic signed [31:0]     DATA_OUT_5,
24          output logic signed [31:0]     DATA_OUT_6,
25          output logic signed [31:0]     DATA_OUT_7,
26          output logic signed [31:0]     DATA_OUT_8,
27          output logic signed [31:0]     DATA_OUT_9,
28          output logic signed [31:0]     DATA_OUT_10,
29          output logic signed [31:0]     DATA_OUT_11,
30          output logic signed [31:0]     DATA_OUT_12,
31          output logic signed [31:0]     DATA_OUT_13,
32          output logic signed [31:0]     DATA_OUT_14,
33          output logic signed [31:0]     DATA_OUT_15,
34
35          output logic [4:0]      OUTPUT_LENGTH, // at most 16
36          output logic [4:0]      OUTPUT_WIDTH,  // at most 16
37          output logic            DONE);
38
39        logic [3:0] operation;
40        logic [3:0] data_type;
41        logic start_0;
42        logic start_1;
43        logic start_2;
44        logic [4:0]  width; // n <= 16, for incoming matrix or vector
45        logic [4:0]  length; // n <= 16, for incoming matrix
46
47        logic [31:0] data_out_1;
48        logic [31:0] data_out_2;
49        logic done_0;
50        logic done_1;
51        logic done_2;
52
53        const logic [3:0] load_op_type = 4'h0;
54        const logic [3:0] load_data_type = 4'h1;
55        const logic [3:0] load_width = 4'h2;
56        const logic [3:0] load_length = 4'h3;
57        const logic [3:0] load_input = 4'h4;
58
59        const logic [3:0] vector_addition = 4'h0;
60        const logic [3:0] linear_transform = 4'h1;
61        const logic [3:0] weighted_inner_product = 4'h2;
62
63        const logic [3:0] load_c_1 = 4'h0;
64        const logic [3:0] load_c_2 = 4'h1;
65        const logic [3:0] load_M = 4'h2;
66        const logic [3:0] load_c = 4'h3;
67        const logic [3:0] load_c_1_out = 4'h4;
68        const logic [3:0] load_c_2_out = 4'h5;
69        const logic [3:0] load_w = 4'h6;
70
71        logic signed [31:0] c_1 [15:0];
72        logic signed [31:0] c_2 [15:0];
73        logic signed [31:0] c_out [15:0];
74        // instantiate the three modules
75        vector_addition vector_addition0 (
76          .clk(clk),
77          .reset(reset),
78          .start(start_0),
79          .c_1_1(c_1[0]),
80          .c_1_2(c_1[1]),
81          .c_1_3(c_1[2]),
82          .c_1_4(c_1[3]),
83          .c_1_5(c_1[4]),
84          .c_1_6(c_1[5]),
85          .c_1_7(c_1[6]),
```

```
86          .c_1_8(c_1[7]),
87          .c_1_9(c_1[8]),
88          .c_1_10(c_1[9]),
89          .c_1_11(c_1[10]),
90          .c_1_12(c_1[11]),
91          .c_1_13(c_1[12]),
92          .c_1_14(c_1[13]),
93          .c_1_15(c_1[14]),
94          .c_1_16(c_1[15]),
95
96          .c_2_1(c_2[0]),
97          .c_2_2(c_2[1]),
98          .c_2_3(c_2[2]),
99          .c_2_4(c_2[3]),
100         .c_2_5(c_2[4]),
101         .c_2_6(c_2[5]),
102         .c_2_7(c_2[6]),
103         .c_2_8(c_2[7]),
104         .c_2_9(c_2[8]),
105         .c_2_10(c_2[9]),
106         .c_2_11(c_2[10]),
107         .c_2_12(c_2[11]),
108         .c_2_13(c_2[12]),
109         .c_2_14(c_2[13]),
110         .c_2_15(c_2[14]),
111         .c_2_16(c_2[15]),
112
113         .done(done_0),
114         .c_1(c_out[0]),
115         .c_2(c_out[1]),
116         .c_3(c_out[2]),
117         .c_4(c_out[3]),
118         .c_5(c_out[4]),
119         .c_6(c_out[5]),
120         .c_7(c_out[6]),
121         .c_8(c_out[7]),
122         .c_9(c_out[8]),
123         .c_10(c_out[9]),
124         .c_11(c_out[10]),
125         .c_12(c_out[11]),
126         .c_13(c_out[12]),
127         .c_14(c_out[13]),
128         .c_15(c_out[14]),
129         .c_16(c_out[15]));
130
131
132       logic signed [31:0] M [15:0];
133       logic signed [31:0] c [15:0];
134       linear_transform linear_transform0 (
135         .clk(clk),
136         .reset(reset),
137         .on(start_1),
138         .M_1(M[0]),
139         .M_2(M[1]),
140         .M_3(M[2]),
141         .M_4(M[3]),
142         .M_5(M[4]),
143         .M_6(M[5]),
144         .M_7(M[6]),
145         .M_8(M[7]),
146         .M_9(M[8]),
147         .M_10(M[9]),
148         .M_11(M[10]),
149         .M_12(M[11]),
150         .M_13(M[12]),
151         .M_14(M[13]),
152         .M_15(M[14]),
153         .M_16(M[15]),
154
```

```
155         .c_1(c[0]),
156         .c_2(c[1]),
157         .c_3(c[2]),
158         .c_4(c[3]),
159         .c_5(c[4]),
160         .c_6(c[5]),
161         .c_7(c[6]),
162         .c_8(c[7]),
163         .c_9(c[8]),
164         .c_10(c[9]),
165         .c_11(c[10]),
166         .c_12(c[11]),
167         .c_13(c[12]),
168         .c_14(c[13]),
169         .c_15(c[14]),
170         .c_16(c[15]),
171 
172         .write(done_1),
173         .y(data_out_1));
174 
175       logic signed [31:0] w;
176       logic M_on;
177       logic signed [31:0] c_1_out [15:0];
178       logic signed [31:0] c_2_out [15:0];
179       weighted_inner_product weighted_inner_product0 (
180         .clk(clk),
181         .reset(reset),
182         .start(start_2),
183         .width(width[2:0]),         // n <= 4
184         .w(w),
185         .M_on(M_on),
186         .M_1(M[0]),
187         .M_2(M[1]),
188         .M_3(M[2]),
189         .M_4(M[3]),
190         .M_5(M[4]),
191         .M_6(M[5]),
192         .M_7(M[6]),
193         .M_8(M[7]),
194         .M_9(M[8]),
195         .M_10(M[9]),
196         .M_11(M[10]),
197         .M_12(M[11]),
198         .M_13(M[12]),
199         .M_14(M[13]),
200         .M_15(M[14]),
201         .M_16(M[15]),
202 
203         .c_1_1(c_1_out[0]),
204         .c_1_2(c_1_out[1]),
205         .c_1_3(c_1_out[2]),
206         .c_1_4(c_1_out[3]),
207 
208 
209         .c_2_1(c_2_out[0]),
210         .c_2_2(c_2_out[1]),
211         .c_2_3(c_2_out[2]),
212         .c_2_4(c_2_out[3]),
213 
214         .done(done_2),
215         .y(data_out_2));
216 
217       integer i;
218       logic [3:0] load_counter;
219       always_ff @(posedge clk) begin
220           if (reset) begin
221               operation <= 0;
222               data_type <= 0;
223 
```

```
            width <= 0; // n <= 16, for incoming matrix or vector
            length <= 0; // n <= 16, for incoming matrix

            data_out_1 <= 0;
            data_out_2 <= 0;
            done_1 <= 0;
            done_2 <= 0;

            w <= 0;

            for(i = 0; i <= 15; i = i+1)begin
                c_1[i] <= 0;
                c_2[i] <= 0;
                c_out[i] <= 0;
                M[i] <= 0;
                c[i] <= 0;
                c_1_out[i] <= 0;
                c_2_out[i] <= 0;
            end
        // 'all_loaded' is to be controled from external
        end else if (chipselect && write && !all_loaded) begin
            // loading stage
            case (address)
                load_op_type : begin
                    operation <= writedata[3:0];
                    // new operation starts, zero out counters
                    load_counter <= 0;
                end
                load_data_type : begin
                    data_type <= writedata[3:0];
                    // new operation starts, zero out counters
                    load_counter <= 0;
                end
                load_width : begin
                    width <= writedata[4:0];
                    // new operation starts, zero out counters
                    load_counter <= 0;
                end
                load_length : begin
                    length <= writedata[4:0];
                    // new operation starts, zero out counters
                    load_counter <= 0;
                end
                load_input : begin
                    case (data_type)
                        load_c_1 : c_1[load_counter] <= writedata;
                        load_c_2 : c_2[load_counter] <= writedata;
                        load_M : M[load_counter] <= writedata;
                        load_c : c[load_counter] <= writedata;
                        load_c_1_out : c_1_out[load_counter] <= writedata;
                        load_c_2_out : c_2_out[load_counter] <= writedata;
                        load_w: w <= writedata;
                    endcase
                    load_counter <= load_counter + 1;
                end
            endcase
        end else if (all_loaded) begin
            // new operation starts, zero out counters
            load_counter <= 0;
        end
    end

    logic [8:0]  counter_2;
    logic [3:0] read_index;
    // workflow: load operation type, data type, width, length, inputs sequentially
    // after been properly loaded, "all_loaded" turns true
    always_ff @(posedge clk) begin
        // computation stage
        if (all_loaded) begin
```

```
293             case (operation)
294                 vector_addition : begin
295                     start_0 <= 1;
296                     // no loading, zero out counters
297                     counter_2 <= 0;
298                     DONE <= done_0;
299 
300                     DATA_OUT_0 <= c_out[0];
301                     DATA_OUT_1 <= c_out[1];
302                     DATA_OUT_2 <= c_out[2];
303                     DATA_OUT_3 <= c_out[3];
304                     DATA_OUT_4 <= c_out[4];
305                     DATA_OUT_5 <= c_out[5];
306                     DATA_OUT_6 <= c_out[6];
307                     DATA_OUT_7 <= c_out[7];
308                     DATA_OUT_8 <= c_out[8];
309                     DATA_OUT_9 <= c_out[9];
310                     DATA_OUT_10 <= c_out[10];
311                     DATA_OUT_11 <= c_out[11];
312                     DATA_OUT_12 <= c_out[12];
313                     DATA_OUT_13 <= c_out[13];
314                     DATA_OUT_14 <= c_out[14];
315                     DATA_OUT_15 <= c_out[15];
316 
317                     OUTPUT_WIDTH <= width;
318                 end
319                 linear_transform : begin
320                     start_1 <= 1;
321                     // no loading, zero out counters
322                     counter_2 <= 0;
323 
324                     DONE <= done_1;
325                     DATA_OUT_0 <= data_out_1;
326                     OUTPUT_WIDTH <= length;
327                 end
328                 weighted_inner_product : begin
329                     if (counter_2 == 0) begin
330                         start_2 <= 1;
331 
332                     end else if (counter_2 == 1) begin
333                         start_2 <= 0;
334                     // M_on turns on
335                     end else if (counter_2 == 2 * width * width + 10) begin
336                         M_on <= 1;
337                     end else if (counter_2 == 3 * width * width + 10) begin
338                         M_on <= 0;
339                         // no loading, zero out counters
340                         counter_2 <= 0;
341                     end
342                     counter_2 <= counter_2 + 1;
343                     DONE <= done_2;
344                     DATA_OUT_0 <= data_out_2;
345                     OUTPUT_WIDTH <= length;
346                 end
347             endcase
348         end else begin
349             // turns off all modules when 'all_loaded' false
350             start_0 <= 0;
351             start_1 <= 0;
352             start_2 <= 0;
353             DONE <= 0;
354 
355             counter_2 <= 0;
356             DATA_OUT_0 <= 0;
357             DATA_OUT_1 <= 0;
358             DATA_OUT_2 <= 0;
359             DATA_OUT_3 <= 0;
360             DATA_OUT_4 <= 0;
361             DATA_OUT_5 <= 0;
```

```
            DATA_OUT_6 <= 0;
            DATA_OUT_7 <= 0;
            DATA_OUT_8 <= 0;
            DATA_OUT_9 <= 0;
            DATA_OUT_10 <= 0;
            DATA_OUT_11 <= 0;
            DATA_OUT_12 <= 0;
            DATA_OUT_13 <= 0;
            DATA_OUT_14 <= 0;
            DATA_OUT_15 <= 0;
        end
    end


endmodule
```

2) ***vector_addition.sv****:* Code for `vector_addition.sv` shown below.

```

module vector_addition (
    input logic clk,
    input logic reset,
    input logic start,
    input logic signed [31:0] c_1_1,
    input logic signed [31:0] c_1_2,
    input logic signed [31:0] c_1_3,
    input logic signed [31:0] c_1_4,
    input logic signed [31:0] c_1_5,
    input logic signed [31:0] c_1_6,
    input logic signed [31:0] c_1_7,
    input logic signed [31:0] c_1_8,
    input logic signed [31:0] c_1_9,
    input logic signed [31:0] c_1_10,
    input logic signed [31:0] c_1_11,
    input logic signed [31:0] c_1_12,
    input logic signed [31:0] c_1_13,
    input logic signed [31:0] c_1_14,
    input logic signed [31:0] c_1_15,
    input logic signed [31:0] c_1_16,

    input logic signed [31:0] c_2_1,
    input logic signed [31:0] c_2_2,
    input logic signed [31:0] c_2_3,
    input logic signed [31:0] c_2_4,
    input logic signed [31:0] c_2_5,
    input logic signed [31:0] c_2_6,
    input logic signed [31:0] c_2_7,
    input logic signed [31:0] c_2_8,
    input logic signed [31:0] c_2_9,
    input logic signed [31:0] c_2_10,
    input logic signed [31:0] c_2_11,
    input logic signed [31:0] c_2_12,
    input logic signed [31:0] c_2_13,
    input logic signed [31:0] c_2_14,
    input logic signed [31:0] c_2_15,
    input logic signed [31:0] c_2_16,

    output logic done,
    output logic signed [31:0] c_1,
    output logic signed [31:0] c_2,
    output logic signed [31:0] c_3,
    output logic signed [31:0] c_4,
    output logic signed [31:0] c_5,
    output logic signed [31:0] c_6,
    output logic signed [31:0] c_7,
    output logic signed [31:0] c_8,
    output logic signed [31:0] c_9,
    output logic signed [31:0] c_10,
    output logic signed [31:0] c_11,
    output logic signed [31:0] c_12,
```

```
    output logic signed [31:0] c_13,
    output logic signed [31:0] c_14,
    output logic signed [31:0] c_15,
    output logic signed [31:0] c_16);

adder adder_1 (clk, reset, start, c_1_1, c_2_1, c_1, done);
adder adder_2 (clk, reset, start, c_1_2, c_2_2, c_2, done);
adder adder_3 (clk, reset, start, c_1_3, c_2_3, c_3, done);
adder adder_4 (clk, reset, start, c_1_4, c_2_4, c_4, done);
adder adder_5 (clk, reset, start, c_1_5, c_2_5, c_5, done);
adder adder_6 (clk, reset, start, c_1_6, c_2_6, c_6, done);
adder adder_7 (clk, reset, start, c_1_7, c_2_7, c_7, done);
adder adder_8 (clk, reset, start, c_1_8, c_2_8, c_8, done);
adder adder_9 (clk, reset, start, c_1_9, c_2_9, c_9, done);
adder adder_10 (clk, reset, start, c_1_10, c_2_10, c_10, done);
adder adder_11 (clk, reset, start, c_1_11, c_2_11, c_11, done);
adder adder_12 (clk, reset, start, c_1_12, c_2_12, c_12, done);
adder adder_13 (clk, reset, start, c_1_13, c_2_13, c_13, done);
adder adder_14 (clk, reset, start, c_1_14, c_2_14, c_14, done);
adder adder_15 (clk, reset, start, c_1_15, c_2_15, c_15, done);
adder adder_16 (clk, reset, start, c_1_16, c_2_16, c_16, done);

endmodule


module adder(
    input logic clk,
    input logic reset,
    input logic on,
    input logic signed [31:0] c1,
    input logic signed [31:0] c2,

    output logic signed [31:0] c,
    output logic done
);

    always @(posedge clk) begin
        if(reset) begin
            done <= 0;
            c <= 32'b0;
        end

        else if (on) begin
            done <= 1;
            c <= c1 + c2;
        end

        else begin
            done <= 0;
            c <= 32'b0;
        end
    end

endmodule
```

3) ***linear_transform.svn****:* Code for `linear_transform.sv` shown below.

```

module linear_transform (
    input logic clk,
    input logic reset,
    input logic on,
    input logic signed [31:0] M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_10, M_11, M_12,
        M_13, M_14, M_15, M_16,
    input logic signed [31:0] c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_10, c_11, c_12,
        c_13, c_14, c_15, c_16,

    output logic write,
    output logic signed [31:0] y
);
```

```

logic signed [31:0] temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8, temp9, temp10,
    temp11, temp12, temp13, temp14, temp15, temp16;


assign temp1 = M_1 * c_1;
assign temp2 = M_2 * c_2;
assign temp3 = M_3 * c_3;
assign temp4 = M_4 * c_4;
assign temp5 = M_5 * c_5;
assign temp6 = M_6 * c_6;
assign temp7 = M_7 * c_7;
assign temp8 = M_8 * c_8;
assign temp9 = M_9 * c_9;
assign temp10 = M_10 * c_10;
assign temp11 = M_11 * c_11;
assign temp12 = M_12 * c_12;
assign temp13 = M_13 * c_13;
assign temp14 = M_14 * c_14;
assign temp15 = M_15 * c_15;
assign temp16 = M_16 * c_16;

always @(posedge clk) begin
    if (reset) begin
        write <= 0;
        y <= 32'b 0;
    end

    else if (on) begin
            write <= 1;
            y <= temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8 + temp9 + temp10
                 + temp11 + temp12 + temp13 + temp14 + temp15 + temp16;
    end

    else begin
        write <= 0;
        y <= 32'b 0;
    end
end



endmodule
```

4) ***weighted_inner_product.sv****:* Code for `weighted_inner_product.sv` shown below.

```

module weighted_inner_product (
    input logic clk,
    input logic reset,
    input logic start,
    input logic [2:0] width,        // n <= 4
    input logic signed [31:0] w,
    input logic M_on,
    input logic signed [31:0] M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_10, M_11, M_12,
        M_13, M_14, M_15, M_16,
    input logic signed [31:0] c_1_1,
    input logic signed [31:0] c_1_2,
    input logic signed [31:0] c_1_3,
    input logic signed [31:0] c_1_4,


    input logic signed [31:0] c_2_1,
    input logic signed [31:0] c_2_2,
    input logic signed [31:0] c_2_3,
    input logic signed [31:0] c_2_4,

    output logic done,
    output logic signed [31:0] y
);
```

```
24
25      logic signed [31:0] vec_c [15:0];
26      logic signed [31:0] c_1 [3:0];
27      logic signed [31:0] c_2 [3:0];
28      logic signed [31:0] temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8, temp9, temp10,
            temp11, temp12, temp13, temp14, temp15, temp16;
29
30      logic gen_enable;
31      logic vec_enable;
32      logic read_enable;
33      logic [2:0] gen_index;
34      logic [2:0] vec_index;
35      logic [2:0] gen_row_num;
36      logic [2:0] vec_row_num;
37
38      integer i;
39      // initialize a DMEM to store outer product.
40      // maximum number of 16 rows are supported.
41      logic signed [31:0] outer_mem0 [3:0]; // n <= 16
42      logic signed [31:0] outer_mem1 [3:0];
43      logic signed [31:0] outer_mem2 [3:0];
44      logic signed [31:0] outer_mem3 [3:0];
45
46      always_ff @(posedge clk) begin
47          if (reset) begin
48              gen_enable <= 0;
49              for(i = 0; i <= 3; i = i+1) begin
50                  c_1[i] <= 0;
51                  c_2[i] <= 0;
52              end
53
54          end else if (start) begin
55              c_1[0] <= c_1_1;        // load everything in one cycle
56              c_1[1] <= c_1_2;
57              c_1[2] <= c_1_3;
58              c_1[3] <= c_1_4;
59              c_2[0] <= c_2_1;
60              c_2[1] <= c_2_2;
61              c_2[2] <= c_2_3;
62              c_2[3] <= c_2_4;
63
64              gen_enable <= 1;
65          end
66
67          if (vec_enable) begin
68              gen_enable <= 0;
69          end
70      end
71
72      // generate one entry in outter product matrix at once
73      always_ff @(posedge clk) begin
74          if (gen_enable) begin
75              if (gen_index < width) begin
76
77                  case (gen_row_num)
78                      3'b000: begin
79                          outer_mem0[gen_index[1:0]] <= c_2[gen_index[1:0]]*c_1[gen_row_num[1:0]]
                                ;
80                      end
81                      3'b001: begin
82                          outer_mem1[gen_index[1:0]] <= c_2[gen_index[1:0]]*c_1[gen_row_num[1:0]]
                                ;
83                      end
84                      3'b010: begin
85                          outer_mem2[gen_index[1:0]] <= c_2[gen_index[1:0]]*c_1[gen_row_num[1:0]]
                                ;
86                      end
87                      3'b011: begin
```

```
                        outer_mem3[gen_index[1:0]] <= c_2[gen_index[1:0]]*c_1[gen_row_num[1:0]]
                            ;
                    end
                    default: begin
                        outer_mem0[gen_index[1:0]] <= 0;
                    end
                endcase

                gen_index <= gen_index + 1;
                if (gen_index + 1 == width) begin
                    gen_index <= 0;
                    gen_row_num <= gen_row_num + 1;
                    if (gen_row_num + 1 == width) begin
                        vec_enable <= 1;
                        gen_row_num <= 0;
                        gen_index <= 0;
                    end
                end
            end else begin
                gen_row_num <= 0;
                gen_index <= 0;
            end
        end else if (read_enable) begin
            vec_enable <= 0;
            for(i = 0; i <= 3; i = i+1) begin

                outer_mem0[i] <= 0;
                outer_mem1[i] <= 0;
                outer_mem2[i] <= 0;
                outer_mem3[i] <= 0;

            end
        end
    end

    integer j;
    always_ff @(posedge clk) begin
        if (vec_enable) begin
            // do vectorization
            if (vec_index < width) begin
                case (vec_row_num)
                    3'h0: vec_c[j] <= outer_mem0[vec_index[1:0]] / w;
                    3'h1: vec_c[j] <= outer_mem1[vec_index[1:0]] / w;
                    3'h2: vec_c[j] <= outer_mem2[vec_index[1:0]] / w;
                    3'h3: vec_c[j] <= outer_mem3[vec_index[1:0]] / w;
                    default: vec_c[0] <= 0;
                endcase

                vec_index <= vec_index + 1;
                j += 1;
                if (vec_index + 1 == width) begin
                    vec_index <= 0;
                    vec_row_num <= vec_row_num + 1;
                    if (vec_row_num + 1 == width) begin
                        read_enable <= 1;
                        vec_row_num <= 0;
                        vec_index <= 0;
                    end
                end
            end
        end
    end

    assign temp1 = M_1 * vec_c[0];
    assign temp2 = M_2 * vec_c[1];
    assign temp3 = M_3 * vec_c[2];
    assign temp4 = M_4 * vec_c[3];
    assign temp5 = M_5 * vec_c[4];
    assign temp6 = M_6 * vec_c[5];
```

```
156     assign temp7 = M_7 * vec_c[6];
157     assign temp8 = M_8 * vec_c[7];
158     assign temp9 = M_9 * vec_c[8];
159     assign temp10 = M_10 * vec_c[9];
160     assign temp11 = M_11 * vec_c[10];
161     assign temp12 = M_12 * vec_c[11];
162     assign temp13 = M_13 * vec_c[12];
163     assign temp14 = M_14 * vec_c[13];
164     assign temp15 = M_15 * vec_c[14];
165     assign temp16 = M_16 * vec_c[15];
166 
167     always @(posedge clk) begin
168         if (read_enable && M_on) begin
169             done <= 1;
170             y <= temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8 + temp9 + temp10
                     + temp11 + temp12 + temp13 + temp14 + temp15 + temp16;
171         end
172 
173         else begin
174             done <= 0;
175             y <= 32'b 0;
176         end
177     end
178 
179 endmodule
```

REFERENCES

[1] Zhou, Hongchao and Gregory W. Wornell. “Efficient homomorphic encryption on integer vectors and its applications.” 2014 Information Theory and Applications Workshop (ITA) (2014): 1-9.

[2] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In The World Wide Web Conference (WWW ’19). Association for Computing Machinery, New York, NY, USA, 2125–2136.

[3] Z. Brakerski and V. Vaikuntanathan, ”Efficient Fully Homomorphic Encryption from (Standard) LWE,” 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science, 2011, pp. 97-106, doi: 10.1109/FOCS.2011.12.

[4] Z. Brakerski, C. Gentry, and S. Halevi, “Packed ciphertexts in LWE- based homomorphic encryption,” in Public-Key Cryptography - PKC, (LNCS) vol. 7778, pp. 1–13, 2013.

[5] Yu, A. et al. “Efficient Integer Vector Homomorphic Encryption.” (2015).

[6] Yang, Zhaoxiong et al. “FPGA-Based Hardware Accelerator of Homomorphic Encryption for Efficient Federated Learning.” ArXiv abs/2007.10560 (2020): n. pag.