FPGA Implementation of RSA Public-Key Cryptographic Coprocessor

Mohamed Khalil Hani
khalil@suria.fke.utm.my

Tan Siang Lin
tanlin@pl.jaring.my

Nasir Shaikh-Husin
nasir_s_h@yahoo.com

MiCE Department,
Faculty of Electrical Engineering
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia.

Abstract: The hardware implementation of the RSA algorithm for public-key cryptography is presented. The algorithm is dependent on the computation of modular exponentials. Critical to this computation is a fast implementation of modular multiplications. A high-performance systolic array architecture for modular multiplication based on the algorithm of P.L. Montgomery is proposed. The design is targeted for implementation in reconfigurable logic, which can yield custom-hardware performance yet maintains all the flexibility of software-based systems. Reconfigurable computing allows the designer to respond, in the prototyping stage, to flaws discovered in implementation or to changes in standards or data formats. We report the issues involved in the preliminary design of the prototype to be fabricated in Altera FLEX10KE series FPGA mounted on a PCI card.

Keywords: RSA algorithm, Montgomery algorithm, systolic array architecture, FPGA.

I. INTRODUCTION

Cryptography is the art of using mathematics to address the issue of information security. The secure transfer and storage of information in the electronic realm has today become critical as the digital world becomes more and more dependent on e-mail, secure telephony, mobile internet, e-commerce, e-banking and so on. Cryptographic systems can provide the objectives of information security: confidentiality, user authentication, data origin authentication, data integrity and non-repudiation [11]. In contrast to symmetric-key cryptosystems, public-key cryptosystems are capable of fulfilling all of these objectives. However, in order to be fast enough and feasibly practical in the applications mentioned above, public-key schemes have to be implemented in hardware. Hardware implementations also provide for ease of installation as well as security from tampering.

Among the existing algorithms for public-key cryptography, the Rivest-Shamir-Adleman (RSA) algorithm is the best known. Its security lies in the difficulty of the factorizing large integers [1]. In RSA, a longer key size means better security. Improvements in the factorization algorithm may inadvertently require that the size of the key be continually and appropriately recommended. The flexibility to change key length or modify the embedded algorithm to respond to design flaws or changes in standards or data formats, requires hardware reconfigurability. Reconfigurable hardware applies to a device that can be configured, at run-time, to implement a function as a hardware circuit. Commercially available reconfigurable devices include Field Programmable Gate Arrays (FPGA) and Complex Programmable Logic Devices (CPLD).

The basic operation in RSA algorithm is modular exponentiation on large integers, and this operation requires a long computation time. The square and multiply method [7] is the most popular and effective algorithm for computing modular exponentiation. The technique reduces the problem to a series of modular multiplications and squaring steps. Consequently, it is critical that the modular multiplication operation is fast, and Montgomery [2] has proposed a fast method for multiplying two integers modulo M, while avoiding division by M. The idea is transform the integers to M-residues and compute the multiplication with these M-residues. It is then transformed back to the normal representation [5]. Walter [4] then proposed a systolic array architecture for high-speed hardware implementation of the Montgomery algorithm. This architecture gives a throughput of one digit per clock cycle and a latency of $2m+2$, where m is the number of digit in the multiplicand A. However, this architecture needs several millions of gates for the typical input of 512 bits.

An interesting choice is to implement only one row of the architecture to perform single message encryption. This would be realizable in a single IC using today's FPGA technology. In this paper, a systolic array architecture, suitable for reconfigurable logic implementation, of the Montgomery modular multiplication is proposed. This will become the core module in a proposed RSA coprocessor that is implemented in a reconfigurable logic device, the FPGA. This coprocessor will off-load computing-intensive cryptographic operations off a general-purpose processor, such that a high performance system can be obtained. To achieve the required speed-up in the RSA operation a PCI bus interface is employed. The prototype is fabricated in Altera FLEX10KE series FPGA mounted on a PCI card in a Pentium PC.

II. RSA CRYPTOGRAPHY

In RSA, to encrypt a message M to its cipher text C, we perform $C = X^E \bmod M$ using the public key E. To restore the message, $X = C^D \bmod M$ is performed, where D is the private key. In general, $E = \sum_{i=0}^{n-1} e_i \cdot 2^i$, $e_i \in \{0, 1\}$ denotes a big integer that consists of n bit in radix-2, e_i is the ith digit.

Modular exponentiation is performed using the square and multiply method as given below. Here, the exponent E is treated bit by bit. Note that the two lines in step 2a are modular multiplications under the same modulus, and they have same multiplier Z_i. Since they are independent of each other, they can be executed in parallel. The loop runs for n cycles where n is the number of bit in E. If higher radix is used in E, then the number of digit to represent E and hence the number of iteration is reduced. The drawback of this speedup is that 2^k–2 multiple of X have to be precomputed and stored. k is the number of bit used to represent one digit.

```
Algorithm 1: ModExp(X, E, M)
compute $P = X^E \bmod M$,
$E = \sum_{i=0}^{n-1} e_i \cdot 2^i, e_i \in \{0, 1\}$
1.  $P_0 = 1, Z_0 = X$
2.  For $i = 0$ to $n$-1Loop
2a.     $P_{temp} = P_i \cdot Z_i \bmod M$
        $Z_{i+1} = Z_i^2 \bmod M$
2b.     If $e_i = 1$ Then $P_{i+1} = P_{temp}$ Else $P_{i+1} = P_i$
3.  End For
```

The speed of this algorithm relies on the speed of modular multiplication in step 2a. There are many algorithms that can be used to perform fast modular multiplication in hardware. An overview of the techniques involved can be found in [3].

III. MONTGOMERY'S ALGORITHM

Montgomery's algorithm [2], given in Algorithm 2 below, is a fast and effective method to calculate modular multiplications. Instead of computing $A \cdot B \bmod M$, it calculates $A \cdot B \cdot R^{-1} \bmod M$. $R > M$ and R is chosen to be a value of power of two so that the operations of mod R and div R is trivial. A precondition is that R must be relatively prime to M, but this is always true in the RSA algorithm because M is an odd number. R^{-1} is the inverse multiplicative of R modulo M. To describe Montgomery modular multiplication, we need an extra integer, N' that satisfy $R \cdot R^{-1} - N \cdot N' = 1$. N' can be calculated using the Extended Euclidean Algorithm. In step 2, the result satisfies $P < 2M$ provided that $A \cdot B < M \cdot R$.

```
Algorithm 2: MonMult(A, B, M)
compute $P = A \cdot B \cdot R^{-1} \bmod M$
1.  $Q = A \cdot B \cdot M' \bmod R$
2.  $P = (A \cdot B + Q \cdot M) / R$
3.  If $P \geq M$ Then Return $P - M$ Else Return $P$
```

To use *MonMult* in *ModExp*, the input operands A and B for *ModExp* are first transformed to the M-residue domain by performing $A \cdot R \bmod M$ and $B \cdot R \bmod M$ respectively. This is performed in step 1 in Algorithm 3 below, where $R^2 \bmod M$ is precomputed and unchanged throughout the whole cryptographic processing using the same key. Thus the product will carry an extra R modulo M. This product can be reused as the operand in the next Montgomery modular multiplication. The final product is transformed back to normal integer by eliminating the extra $R \bmod M$. Now, *ModExp* function becomes:

```
Algorithm 3: ModExp(X, E, M)
compute $P = X^E \bmod M$,
$E = \sum_{i=0}^{n-1} e_i \cdot 2^i, e_i \in \{0, 1\}$
1.  $P_0 = MonMult(1, R^2 \bmod M)$
    $Z_0 = MonMult(X, R^2 \bmod M)$
2.  For $i = 0$ to $n$-1
2a.     $P_{temp} = MonMult(P_i, Z_i)$
        $Z_{i+1} = MonMult(Z_i, Z_i)$
2b.     If $e_i = 1$ Then $P_{i+1} = P_{temp}$ Else $P_{i+1} = P_i$
3.  End For
4.  $P = MonMult(P_n, 1)$
```

Since the operands involved are very large and can vary from 512 to 2048 bits, multiprecision arithmetic is employed in *MonMult*. The operands M, A and B will each consist of m digits, with each digit occupying k bits. If the digit is in radix r, then Mod r will yield a digit value, while div r will be equivalent to a simple operation of a right shift of one digit. *MonMult* now becomes:

```
Algorithm 4: MonMult(A, B)
compute $P = A \cdot B \cdot r^{-m} \bmod M$,
$M = \sum_{i=0}^{m-1} m_i \cdot r^i, m_i \in \{0, 1, \ldots, r-1\}, r = 2^k$,
$B = \sum_{i=0}^{m-1} b_i \cdot r^i, b_i \in \{0, 1, \ldots, r-1\}$,
$A = \sum_{i=0}^{m-1} a_i \cdot r^i, a_i \in \{0, 1, \ldots, r-1\}$
1.  $P_0 = 0$
2.  For $i = 0$ to $m$-1 Loop
2a.     $q_i = (p_{i,0} + a_i \cdot b_0) m_0' \bmod r$
2b.     $P_{i+1} = (P_i + a_i \cdot B + q_i \cdot M) / r$
3.  End Loop
4.  If $P_m \geq M$ Then Return $P_m - M$ Else Return $P_m$
```

The notation $p_{i,0}$ denotes the 0th digit of ith integer P. In step 3, this algorithm produces $P \equiv A \cdot B \cdot r^{-(m+1)} \bmod M$ that satisfies $P < 2M$, provided that A, $B < 2M$. Eldridge and Walter in [3] pointed out that, the Montgomery's algorithm:

- reverses the order of treating the digit of the multiplicand A,
- performs a shift down instead of up on each iteration, and
- does an addition rather than subtraction.

The digit q_i calculated in step 2a determines the multiple of M to be added in the long addition in step 2b. This is the most important property of the Montgomery's method. In short, the classical modular multiplication algorithms compute the entire sum in order to decide whether a reduction needs to be performed [9].

IV. ALGORITHM OPTIMIZATION

The long subtraction in every *MonMult* is an expensive operation, and it can be avoided by leaving it out until the final step of the *ModExp*. So the intermediate result from the *MonMult* falls below $2M$. In radix-2, in order to reuse this value as the operands for the next *MonMult*, two extra iterations are needed by inserting $a_{m+1} = 0$ [4]. After $m+2$ iterations, $P \equiv A \cdot B \cdot 2^{-(m+2)} \bmod M$ is produced. In high-radix cases, only one more iteration is needed since the most significant digit of B has a maximum value of 1 instead of r-1. So, R is equal to $r^{(m+1)}$. Algorithm 5 below shows the radix-2 version of *MonMult* that accepts A, $B < 2M$ and produces result below $2M$.

```
Algorithm 5: MonMult(A, B)
compute $P = A \cdot B \cdot 2^{-(m+2)} \bmod M$,
$M = \sum_{i=0}^{m-1} m_i \cdot 2^i$, $B = \sum_{i=0}^{m} b_i \cdot 2^i$, $A = \sum_{i=0}^{m+1} a_i \cdot 2^i$,
$m_i, b_i, a_i \in \{0, 1\}$, $a_{m+1} = 0$,
1.  $P_0 = 0$
2.  For $i = 0$ to $m+1$ Loop
2a.     $q_i = (p_{i,0} + a_i \cdot b_0) m_0' \bmod 2$
2b.     $P_{i+1} = (P_i + a_i \cdot B + q_i \cdot M) / 2$
3.  End Loop
```

To simplify the determination of digit q_i, several techniques are used [3][10]. In radix-2, $m_0' = 1$, and thus $q_i = (p_{i,0} + a_i \cdot b_0) \bmod r$. In the higher radix case, a technique to avoid multiplication with m_0' is by transforming M into N, where $N = M \cdot m_0'$. Since $n_0 = -1$ and thus $n_0' = 1$. Now, $q_i = (p_{i,0} + a_i \cdot b_0) n_0' \bmod r = (p_{i,0} + a_i \cdot b_0) \bmod r$. Since N is one digit bigger than M, the final result has to subtract at most r-1 times of M to reduce to below M. Also, the acceptable operand size for the *MonMult* is now A, $B < 2N$. An extra iteration is needed to take into account the extra digit in A.

To further simplify it, we can shift the operand B up 1 digit, in order to make $b_0 = 0$. Now, $q_i = p_{i,0} \bmod r = p_{i,0}$. The technique of shifting B up one position is applied in both radix-2 and high-radix cases. The price for these simplifications is one more iteration of the loop. Thus, $a_{m+2} = 0$ is inserted. The algorithms below summarize the above discussion for the radix-2 (in Algorithm 6) and high-radix (in Algorithm 7) versions.

```
Algorithm 6: MonMult(A, B)
compute $P = A\cdot B\cdot 2^{-(m+2)} \bmod M$,
$M = \sum_{i=0}^{m-1} m_i\cdot 2^i,\ B = \sum_{i=0}^{m} b_i\cdot 2^i,\ A = \sum_{i=0}^{m+2} a_i\cdot 2^i$,
$m_i, b_i, a_i \in \{0, 1\},\ a_{m+1} = a_{m+2} = 0$,
1. $P_0 = 0$
2. For $i = 0$ to $m+2$ Loop
2a.     $q_i = p_{i,0}$
2b.     $P_{i+1} = (P_i + q_i\cdot M) / 2 + a_i\cdot B$
3. End Loop
```

```
Algorithm 7: MonMult(A, B)
compute $P = A\cdot B\cdot r^{-(m+2)} \bmod M$,
$M = \sum_{i=0}^{m-1} m_i\cdot r^i,\ B = \sum_{i=0}^{m} b_i\cdot r^i,\ A = \sum_{i=0}^{m+2} a_i\cdot r^i$,
$m_i, b_i, a_i \in \{0, 1, \ldots, r-1\},\ a_{m+1}, b_{m+1} \in \{0,1\},\ a_{m+2} = 0$,
1. $P_0 = 0$
2. For $i = 0$ to $m+2$ Loop
2a.     $q_i = p_{i,0}$
2b.     $P_{i+1} = (P_i + q_i\cdot M) / r + a_i\cdot B$
3. End Loop
```

Next, we discuss the mapping of above algorithm in the proposed systolic array architecture, which is based on Walter's described in [4].

V. SYSTOLIC ARRAY ARCHITECTURE

The complexity of Algorithm 6 or Algorithm 7 lies in the addition of three operands of $m+2$ bits for calculating P_{m+3}. Blum[6] points out that two different strategies have been pursued: redundant representation and systolic array. In redundant representation, the intermediate results are kept in redundant form. Conversion into binary is only performed at the end of every modular multiplication. In systolic array approach, there are m processing elements, each calculating 1 bit per clock cycle. Since the signal is distributed between adjacent processing elements, faster clock rate and thus higher bandwidth can be achieved. The cost is higher latency and more resources.

The systolic array proposed by Walter consists of $m+4$ columns by $m+3$ rows for the modular multiplication operation as described in Algorithm 6. Due to the limitation in space of this paper, we will now focus our discussion on the implementation of Algorithm 6 (radix-2 case) only. At the outset, note that $m+4$ columns (i.e. 4 bits more than the number of bits of M) are needed because the intermediate result has an upper bound of $10M$ before div 2. Each row performs an iteration of the loop, and the columns compute successive values for a single bit position. The rows and columns are pipelined so that the data flow from the upper right cell to the lower left cell and each cell take a cycle to process. Since there are $m+3$ iterations to go through, the latency is thus $2(m+3)$ cycles. If all the $m+3$ rows are implemented, the throughput is one Montgomery modular multiplication per clock cycle and it can encrypt $m+3$ different messages simultaneously. Since we are going to implement just one row, the throughput will be one Montgomery modular multiplication per $m+3$ clock cycles. This architecture is able to perform two Montgomery modular multiplications simultaneously or single message at a time.

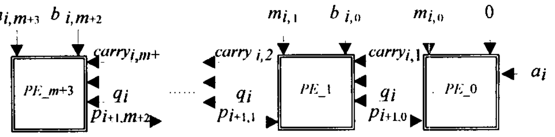

Fig. 1. One row of modular multiplication cells

Fig. 1 shows the systolic array of one row. The typical cell performs a single bit calculation of the operation:

$P_{i+1} = (P_i + q_i\cdot M) / r + a_i\cdot B$, i.e.

$p_{i+1,j-1} + 2\cdot carry_{i,j+1} = p_{i,j} + a_i\cdot b_{i,j-1} + q_i\cdot m_{i,j} + carry_{i,j}$.

Each cell generates a carry. These carries are bounded by 2 and thus 2 bits are needed, while the rest of the signals connected to each cell are 1 bit in length. The leftmost cell is much simpler due to the fact that $m_{i,m+3}$, $m_{i,m+2}$, $m_{i,m+1}$, $m_{i,m}$ and $b_{i,m+2}$ are all zero. The signal $carry_{i,m+3}$ is therefore bounded by 1. This row performs $m+3$ iterations of the looping by feeding back the previous P_{i+1}, i.e. feed the bit $p_{i,j}$ to the right cells as $p_{i+1,j-1}$. q_i is calculated in the rightmost processing element by simply taking the value of previous $p_{i,0}$. Both q_i and a_i are pumped through the processing elements.

The signal flow for Algorithm 6 in this architecture is described as follows. Initially, all the registers that store P are reset, i.e. $P_0 = 0$. At clock cycle 1, the first iteration starts: q_0 is assigned to $p_{0,0} = 0$, while $m_{0,0}$, 0 and the first a_0 are fed to PE_0 to generate $carry_{0,1}$ and $p_{1,-1}$. This bit p from PE_0 is always equal to zero and it is discarded. In clock cycle 2, $m_{0,1}$ and $b_{0,0}$ are fed to PE_1 to generate $carry_{0,2}$ and $p_{1,0}$. At the same time, $m_{1,0}$, 0 and the second a_0 are fed to PE_0. The $p_{1,0}$ from PE_1 is used to calculate q_1 of second iteration in clock cycle 3. After $2(m+3)$ clock cycles, the first bit of the result of first modular multiplication is generated at PE_1. At the next clock cycle, PE_2 generates the second bit. At the same time, the first bit of the second modular multiplication also appears out of PE_1. As soon as the first bit result of a modular multiplication is generated, the next modular multiplication can commence. Both modular multiplications in step 1 or step 2a in Algorithm 3 are thus run in parallel using this one row architecture.

Inspection of Algorithm 3 shows that the modular squaring result is reused as B in both the modular squaring and modular multiplication of next iteration. Also, both modular squaring and modular multiplication results are reused as As in next iteration. Thus a register B is needed to store the squaring result, and two RAMs are needed to store the variables P and Z. These RAMs feed the one row architecture with a_i by alternatively selecting the bit of P

and Z from the least significant bit to the most significant bit. Initially, register B is filled with the precomputed value of R^2 mod M while P and Z are filled with 1 and X, respectively, in order to perform step 1. Before executing step 4, the value 1 is fed to the register B. A long subtraction has to be performed finally before we get the result of the modular exponentiation.

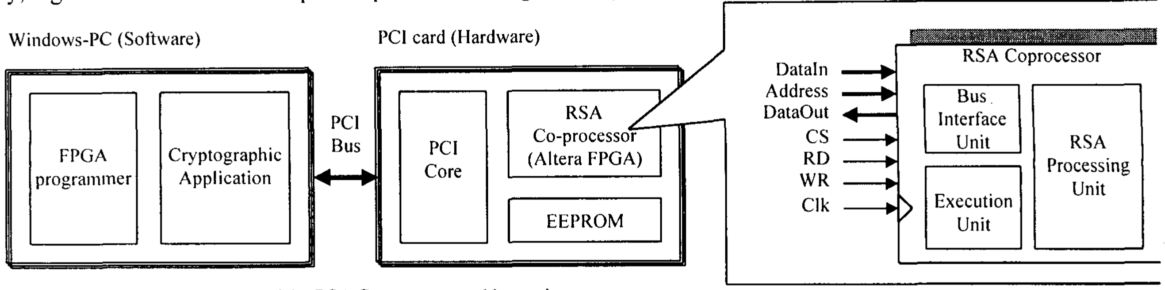

Fig. 2. Top-level block diagram of the RSA Coprocessor and its environment

VI. FPGA IMPLEMENTATION

In this section, we present some of the features of the Altera FLEX10KE that makes it particularly suitable to implement the systolic array architecture. This device consists of four types of reconfigurable elements, the logic array blocks (LAB), embedded array blocks (EAB), the I/O elements (IOE) and the routing resource [8]. Each LAB contains eight logic elements (LE) and a local interconnect. An LE consists of a look-up-table (LUT) that can perform any Boolean functions of 4-bit input. Its 1 bit output can either passed out directly or registered in a flop-flop. The EAB is a 2048 bits RAM block that can be configured in size from 2048 x 1 to 256 x 8. This is appropriate for storing the key in RSA algorithm, which is typically 512 to 2048 bits in length. The routing resource connects the LAB, EAB and IOE into a network. This routing provides predictable performance because it is a series of continuous horizontal and vertical routing channels that traverse the device. Another feature of the LAB is the ability to fast perform arithmetic function via the fast carry chain between LEs, thus avoiding the long carry propagation delay as found in the adder.

Fig. 2 shows a functional block diagram of the proposed RSA coprocessor, and its interface to the PC environment. The modular exponentiation using the one row systolic array architecture described above is the main module in the RSA processing unit. For preprocessing and post processing purposes in the RSA algorithm, this unit also contains other modules that perform Extended Euclidean algorithm, and long integer subtraction, multiplication and division. Several RAM blocks are employed using the EAB to store various intermediate results. The PCI core acts as the PCI controller that interface the system PCI bus and the backend design.

For both cost and speed evaluation, the gate count and gate delay model in [4] for VLSI implementation is not suitable in the FLEX10KE device. The timing information in our design is obtained from the timing analyzer of Altera MAX+plusII software. It is calculated based on the timing model of the building blocks of the FLEX10KE such as the LE's delay, the flip-flop setup/hold time and the routing path's delay.

The design in this project is described in VHDL using VHDLmg (VHDL module generator), an in-house developed EDA tool [12]. Synopsis FPGA Express is used as the logic synthesis tool, while Altera MAX+plusII is the compilation tool.

VII. CONCLUSION

We have presented the development of a modular multiplication algorithm suitably mapped to a systolic array architecture targeted for implementation in a reconfigurable logic device. A high-speed implementation of this algorithm is critical to fast modular exponentiation operation required in a RSA cryptographic coprocessor. The proposed implementation in hardware reconfigurable logic also provides the necessary flexibility required to deal with changes in standards and responds to design flaws in prototyping.

Acknowledgement: This work is supported by a fellowship grant from Intel Technology Sdn. Bhd. Penang.

REFERENCE

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signature and Public-Key Cryptosystems," Commun. ACM, vol. 21, pp 120-126, 1978.

[2] P. L. Montgomery, "Modular Multiplication Without Trial Division," Mathemat. of Computat., vol 44, pp 512-521, April 1985.

[3] S. E. Eldridge and C. D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," IEEE Trans. Comput., vol. 42, no. 6, pp 693-699, June 1993.

[4] C. D. Walter, "Systolic Modular Multiplication," IEEE Trans. Comput., vol 42, no. 3, pp 376-378, Mar. 1993.

[5] C. K. Koc, T. Acar, and B. S. Kaliski, Jr, "Analyzing and Comparing Montgomery Multiplication Algorithms," IEEE Micro., vol. 16, no. 3, pp 26-33, June 1996.

[6] T. Blum and C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware," 14th IEEE symposium on Comput. Arith., pp 70-77, 1999.

[7] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 2nd edition, Reading, MA: Addison-Wesley, 1981.

[8] Altera Inc., San Jose, CA, Device Data Book 1999.

[9] M. Shand, J. Vuillemin. "Fast Implementation of RSA Cryptography," In Proceedings 11th IEEE Symposium on Computer Arithmetic, pp 252-259, 1993.

[10] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," In Proceedings 12th Symposium on Computer Arithmetic, pp 193-199, 1995.

[11] B. Schneier. Applied Cryptography: Protocols, Algorithm and Source Code in C, 2nd Edition, New York: John Wiley & Sons, Inc., 1996.

[12] Mohamed Khalil, Koay Kah Hoe, " VHDL Module Generator: A Rapid-prototyping Design Entry Tool for Digital ASICs," Jurnal Teknologi, 31(D)1999, Univ. Teknologi Malaysia, Dec.1999, pp.45-61.

BIOGRAPHY

Mohamed Khalil Hani obtained his B.Eng. in Communication from University of Tasmania, Australia in 1978, M. Eng. in Computer Engineering from Florida Atlantic Univ., Boca Raton in 1985, and Ph.D. in Digital System and Computer Engineering from Washington State University, Pullman in 1992. He is currently the Vice Dean at the Faculty of Electrical Engineering, Universiti Teknologi Malaysia, Skudai. His research interests include digital system design and computer architecture, VHDL and FPGA hardware implementations, CAD or digital design automation, artificial intelligence with focus on hardware implementations of neural networks, fuzzy logic and expert systems, and emergent technologies in reconfigurable computing, embedded multimedia and cryptography.

Nasir Shaikh-Husin is a Lecturer at Faculty of Electrical Engineering, Universiti Teknologi Malaysia. He is presently also head of VLSI CAD Lab. Nasir obtained his B. Eng. (Electrical) degree from Lakehead University, Canada in 1985 and M.Sc. (Microelectronics) in 1987 from University of Durham, United Kingdom. His research interests are applications of neural networks for solving optimization problems and IC design, especially related to hardware implementation of neural networks.

Tan Siang Lin obtained his first degree with second class upper in Electrical Engineering from Universiti Teknologi Malaysia in 1999. Currently, he is pursuing his master degree in Electronic Engineering at the same institution. His research interest includes digital hardware design and computer engineering.