Vulnerability Assessment of an IHP ECC Implementation (technical report)

Estuardo Alpirez Bock, Zoya Dyka
System dept.,
IHP
Frankfurt(Oder), Germany

Abstract— *Mathematically, cryptographic approaches are secure. This means that the time an attacker needs for finding the secret by brute forcing these approaches is about the time of the existence of our world. Practically, an algorithm implemented in hardware is a device that generates a lot of additional data during calculation. Its power consumption, electromagnetic radiation etc. can be measured, saved and analysed for the key extraction. Such attacks – the side channel analysis attacks (SCA attacks) – are significant threats when applying cryptographic algorithms. By taking the issue of physical attacks into consideration when implementing a cryptographic algorithm, it is possible to design an implementation that is resilient – at least to a certain extend – against side channel analyses. In this report, we give implementation details of the IHP accelerator for the elliptic curve point multiplication. We analysed the implemented algorithm flow and its power consumption using simulated power traces for the 130nm CMOS IHP technology. We made a horizontal power analysis attack using the difference of means test with the goal of finding potential SCA leakage sources, i.e. finding the operations in the algorithmic flow that are responsible for the correct extraction of the cryptographic key.*

Keywords — difference of means test, elliptic curve cryptography, power analysis, side channel analysis

I. Introduction

This report presents the technical details of the results obtained from a vulnerability assessment made on an Elliptic Curve Cryptography (ECC) implementation from IHP [1]. This vulnerability assessment was made within the framework of the project for Tamper-Resistant ICs for Critical Infrastructures (German: Manipulationssichere Schaltkreise für kritische Infrastrukturen - MaSch) [2]. MaSch develops work and research on the secure implementation of cryptographic algorithms. Standardized cryptographic algorithms guarantee security from a mathematical point of view as long as the value of the used key is kept secret. Nevertheless the implementation and execution of these algorithms on ICs lead to measurable physical parameters, such as power consumption and execution time of the operations performed. The information provided by the measurements of such parameters can help an attacker reach his goal: finding out the value of the secret key. By taking these aspects into consideration when implementing a crypto-algorithm, it is possible to design an implementation that is strong against side channel analyses (SCA).

This technical report presents the results and details of the assessment. It first describes the implementation's characteristics. The system's most relevant technical aspects regarding this assessment are then described in three subsections, providing information about the system's architecture, the system's most relevant signal for the control of its operations and a description of the implemented Montgomery algorithm.

Then, primary observations that lead to the implementation's vulnerabilities are listed before concluding this report with suggestions on how to improve this design regarding its protection against side channel attacks.

II. Implementation Details

The IHP hardware accelerator for the Elliptic Curve Cryptography (ECC) is an implementation of the Montgomery algorithm for elliptic curves point multiplication *kP*. The following points describe further characteristics of the implementation:

- The algorithm is implemented using standard NIST elliptic curves *B-233* [3]
- The implementation is optimized for IHP 130nm technology.
- The points of the elliptic curves (EC) have been represented by using Lopez-Dahab projective coordinates [4].
- The $GF(2^{233})$ element's multiplication has been implemented using the iterative *4-segment Karatsuba* multiplication method [5] and needs 9 clock cycles for calculating one product.
- The analysed version of the ECC design is a further developed version of the one described in [6].
- Countermeasures against differential power analysis (DPA) have not been implemented in this design. A *difference of means* attack was successfully realized by means of the TAMPRES-Project [7].

The task of this vulnerability assessment was to find out the possible causes of the successful realization of the *difference of means* attack. The analysis has been carried out based on simulated values of the power consumption of the ECC design while executing the EC point multiplications *kP* (in this report referred to as *kP*-operation).

III. Difference of Means Test of the IHP ECC Design

We performed the *difference of means* test for two simulated cases:

- ✓ Case 1: for the IHP *kP*-design processed the EC point $P_1=(x_1,y_1)$ as the input data using the scalar k_1
- ✓ Case 2: for the IHP *kP*-design processed the EC point $P_1=(x_1,y_1)$ as the input data using the scalar k_2

The values of the data and the scalars in hexadecimal notation are:

x_1=181 856adc1e 7df13784 91fa736f 2d02e8ac f1b9425e b2b061ff 0e9e8246
y_1=89 fed47b79 6480499c baa86d8e b39457c4 9d5bf345 a0757e46 e2582de6
k_1=93 919255fd 4359f4c2 b67dea45 6ef70a54 5a9c44d4 6f7f409f 96cb52cc
k_2=cd ea65f6dd 7a75b8b5 133a70d1 f27a4d95 06ecfb6a 50ea526e b3d426ed

Simulation results of the power consumption of the IHP ECC design for the investigated cases are obtained using the Synopsis Tools PrimeTime [8] and saved in an "out"-file. This file consists of the power consumption not only for the whole design but also for each of its components, i.e. for each block of the ECC design. The *difference of means* test was performed for the following blocks separately:

1 – whole ECC design (ecc)
2 – block multiplier (mult)
4 – block ALU
5 – register x_1 (X1)
6 – register z_1 (Z1)
7 – register x_2 (X2)
8 – register z_2 (Z2)

Additionally, the *difference of means* test was performed for the sum of the power consumptions of the registers X1, Z1, X2, Z2 and the block ALU. The analysis of the results of the difference-of-means test for these blocks can be useful to find the block(s) that is the SCA leakage source.
Each simulated power trace of the investigated IHP ECC design can be separated in parts corresponding to the kP - algorithm:

- The initialization part
  This part corresponds to the initialization phase of the Montgomery kP-algorithm, the conversion of affine EC point coordinates (*x*,*y*) to projective coordinates (X,Y,Z), including the processing of the most significant bit of the scalar *k*. This part takes 8 clock cycles.
- The part of the processing of all remaining bits of the scalar (i.e. the key) *k*, excluding its most significant bit.
  Both scalars that we use – k_1 and k_2 – are 232 bit long. It means this part consists of 232-1=231 slots[1]. The duration of each slot is the same –57 clock cycles always. Each clock cycle is presented by one power value only. This means each slot consists of 57 points. So, the *kP*-operation needs (232-1)·57=13167 clock cycles to be calculated.
- The last part of the *kP* trace
  This part corresponds to the conversion of the multiplication result $kP=(X, Y, Z)$ back to affine coordinate and takes 431 clock cycles.

For the *difference of means* test only the part of the *kP*-operation that corresponds to the processing of the scalar was chosen, i.e. the part with 231 slots. The difference-of-means test was performed as follows:

1. We partitioned the investigated part of the power trace into 231 slots, 57 clock cycles each. We observed each slot as a curve that consists of 57 points. Fig. 1 shows the first 8 slots of the simulated PT for case 1. Fig. 2 shows the slots from Fig. 1 clockwise, i.e. in the same coordinates.

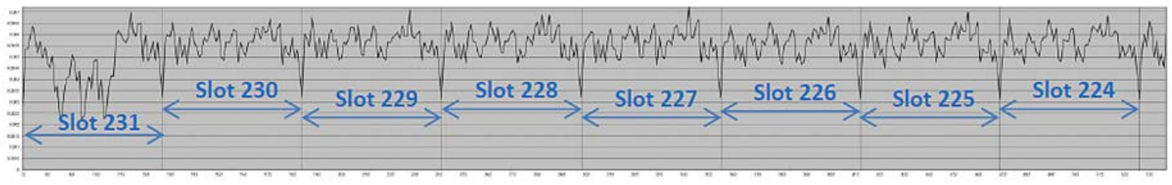


Fig. 1. First 8 slots of the simulated Power Trace for the case 1: the IHP *kP*-design processed the EC point $P_1=(x_1,y_1)$ as the input data using the scalar k_1.

[1] Each slot corresponds a processing of only one bit of the scalar *k*.

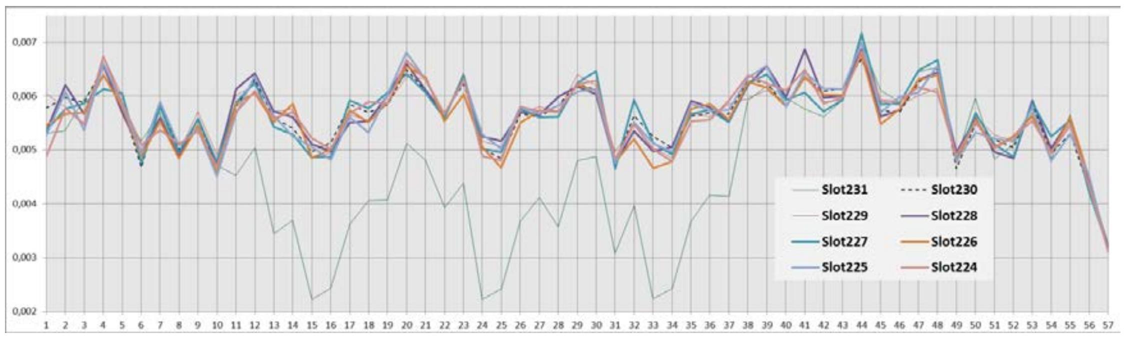


Fig. 2. All slots from Fig. 1 shown clockwise, in the same coordinates

2. We obtained the mean curve of all these 57 points long curves, i.e. we calculated the mean curve for 231 slots.
3. We compared the power value of the 1st point of the means curve with the power value of the 1st point of each slots, from slot 0, that corresponds to the processing of the key bit number 0, up to the slot 231, that corresponds to the processing of the key bit number 231. If the power value of the mean curve is higher than the value of the current slot, we assumed, it corresponds to the key bit value '1', otherwise to '0'. Thus, we obtained the first key candidate.
4. We repeated step 3 for all other 56 points of the mean curve and obtained the remaining 56 key candidates.

Usually the variance is calculated to obtain information about SCA leakage sources. We calculated it for each point of the mean curve as follows:

$$variance = \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)$$

Here n=231 is the number of slots, $\overline{x}$ is the average value of all values x_i.

Using this formula we calculated values of the variance for all 57 points of the mean curve. Fig. 3 shows all calculated variances as a 57 points long curve.

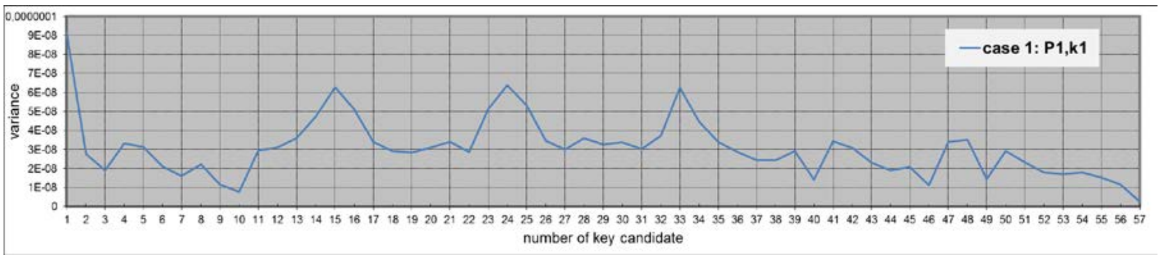


Fig. 3. All calculated variances as a 57 points long curve for case 1

The peaks on the variance curve are usually defined as a potential SCA leakage sources. Applying this approach we have at least 4 SCA leakage sources: they are the operations performed at clock cycle 1, 15, 24 and 33 of slots.

The variance curve for case 2, i.e. for the same ECC design processing the same input data but using the other key, i.e. scalar k_2, is shown in Fig. 4.

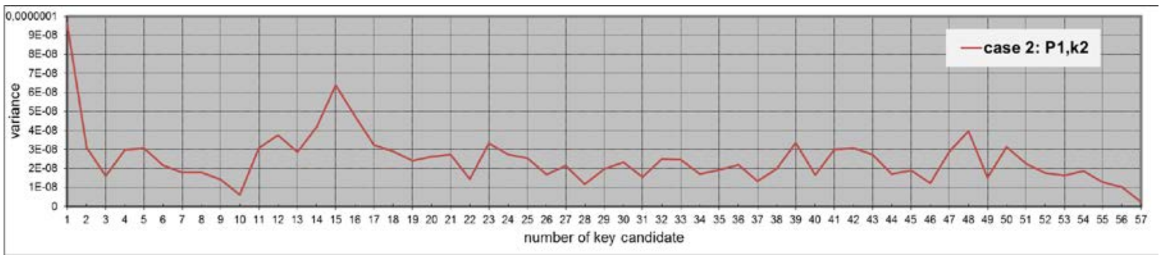


Fig. 4. All calculated variances as a 57 points long curve for case 2

The peaks on the variance curve in Fig. 4 are at clock cycle 1 and 15.

Because the variance curve cannot be used to define clearly all SCA leakage sources of the investigated designs, we compared each of the 57 key candidates with the key value that was really processed. We calculated the relative correctness for each key candidate as *number_of_correct_extracted_bits/232*100%*. We presented this relative correctness of the extraction of the key for each of the 57 key candidates as a curve that consists of 57 points, see red curve in Fig. 5 for the case 1 and the black curve for the case 2. From the security point of view the ideal case is if the correctness of the key extraction is 50% for all key candidates. The green curve in Fig. 5 corresponds to this case.

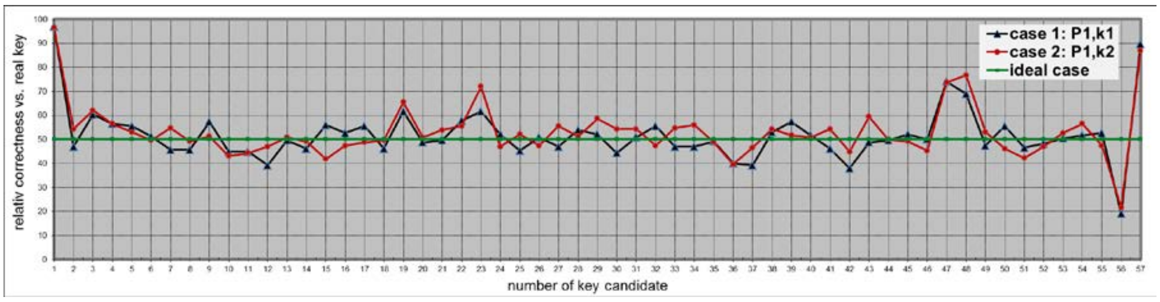


Fig. 5. Relative correctness of the extraction of the key for each of the 57 key candidates as a curve: the red curve corresponds to the case 1 and the black curve to the case 2; the green curve shows the ideal case

The correlation between the red and the black curves in Fig. 5 is much higher than the correlation between variance curves for both cases: in both investigated cases the correctness of the key extraction for the key candidates 1, 47, 48 and 57 is extremely high. For example, the correctness of the key candidate 1 is about 97 % in both cases.
Also the correctness of the key candidates 19 and 23 is high.

Contrary, the correctness of the key candidate 56 is extremely low, only about 20 %. It means that our basic assumption for the search of the key candidate was wrong. But it means that for example for the 56th point of each slot the "inverted" assumption is correct: if the power value of the 56th point of the mean curve is higher than the value of the current slot, it corresponds to the key bit value '0', otherwise to '1'. The new key candidate, that we obtained using this new assumption, is equal to the inverted "old" key candidate. The correctness of this new key candidate number 56 is 100-20=80 %.
The relative correctness of the extraction of the key for each of the 57 key candidates using the inverted assumption in comparison to the "old" assumption is shown in Fig. 6.

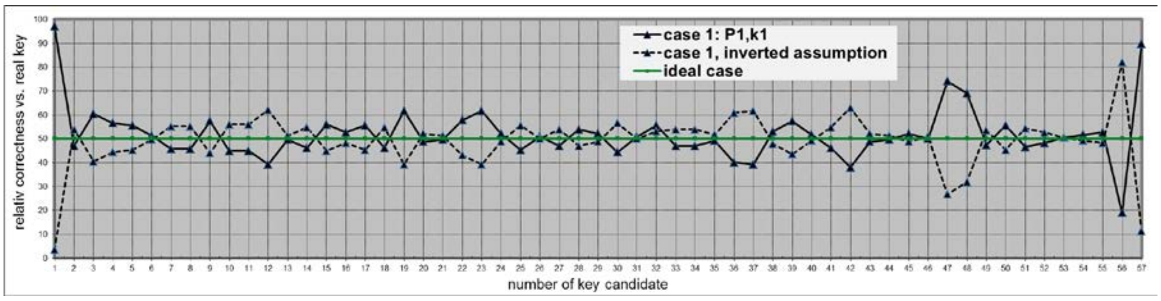


Fig. 6. Relative correctness of the extraction of the key for each of the 57 key candidates using the inverted assumption in comparison to Fig. 5.

Using the curve of relative correctness of the key candidates we can summarize: 5 of 57 calculated key candidates are extracted with relative correctness from 70% up to 97% and next 3 key candidates – number 19, 23 and 36 are extracted with relative correctness of about 60%.
This means, the IHP implementation of the *kP* operation:

- can be successfully attacked using *difference of means* test by a horizontal attack
- has many SCA leakage points: at least at clock numbers 1, 19, 23, 36, 47, 48, 56 and 57.

This kind of analysis of the results of the *difference of means* test provides much more correct defined SCA leakage sources in comparison to the calculation of the variance.

The *difference of means* test using the PTs of blocks of the ECC design shows much more possible sources of the SCA leakage. Fig. 7 shows the relative correctness of the extraction of the key for each of the 57 key candidates for most blocks of the ECC design separately. The register X1, X2, Z1, Z2 and the block ALU are most unsecure blocks. The activity of the block multiplier can be assessed as significant SCA leakage source at clock 56 only.

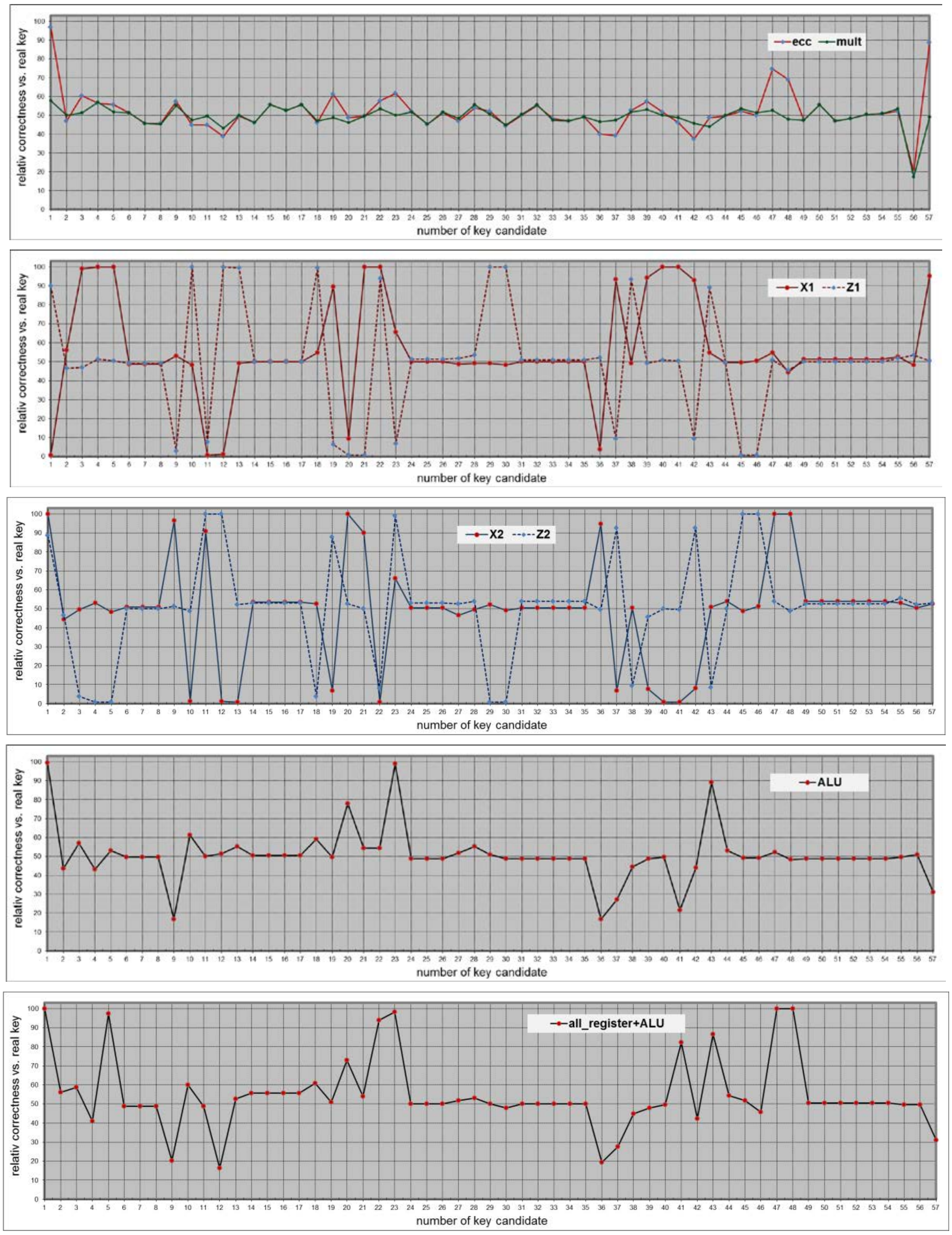


Fig. 7. Relative correctness of the extraction of the key for each of the 57 key candidates using PTs of whole ecc design and of its blocks separately.

IV. Technical Description

This section describes the ECC design properties that have been identified through the analysis of the source code of the *kP* implementation, its functionality and its power consumption. The properties being described are the most relevant ones regarding the vulnerability assessment made on this report.

A. *System Architecture*

The analysed IHP ECC design was implemented with the hardware description language VHDL (Very High Speed Integrated Circuit Hardware Description Language) and consists of 17 files describing, e.g., registers and mathematical operators. It was intended with this assessment to analyse the implementation details of this design through observation of its source code and through simulation of its functionalities by means of a test bench, which included the key k and the coordinates of the elliptic curve point $P = (x,y)$ as input parameters. The inputs k, x and y can be up to 233-bit long binary numbers. The following hexadecimal values have been assigned to these inputs:

k = 2cc
x = 181 856adc1e 7df13784 91fa736f 2d02e8ac f1b9425e b2b061ff 0e9e8246
y = 89 fed47b79 6480499c baa86d8e b39457c4 9d5bf345 a0757e46 e2582de6

The key k is only 10 bit long, thus the simulation time is reduced. The simulated traces for the power consumption of the entire design and of its single functional units (FUs) were generated with the software PrimeTime from Synopsis [8].

This subsection presents the system architecture of the implementation, represented by the block diagram shown in Fig. 8, which shows all entities that take part in the execution of the *kP*-operation and how these are connected with each other.

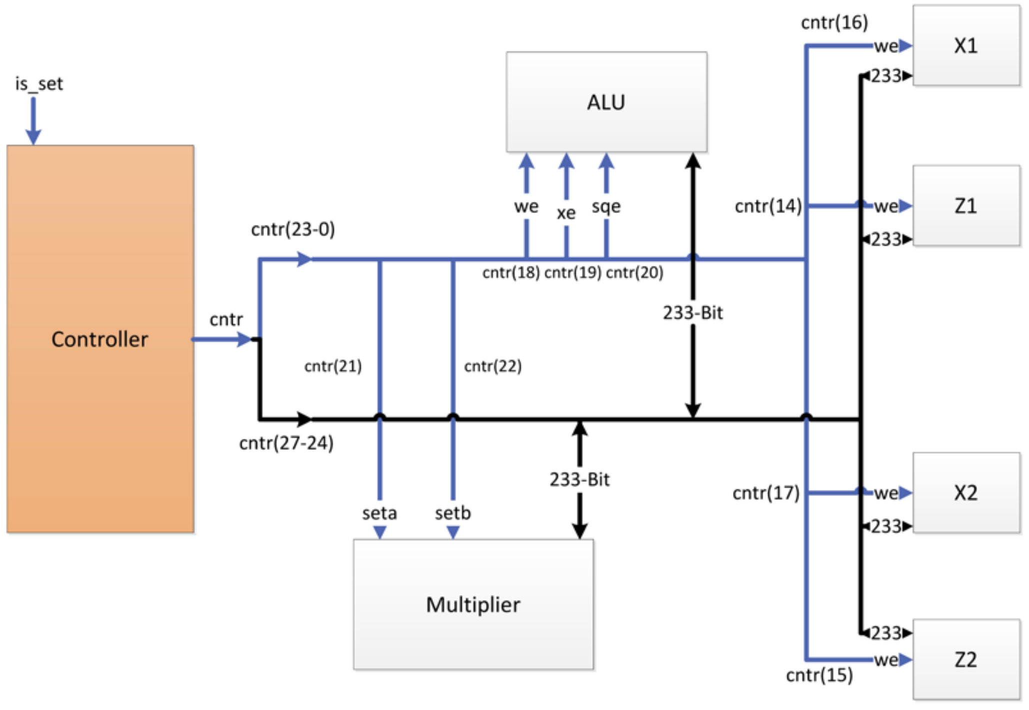


Fig. 8. Structure of the IHP ECC design.
The entities shown in this diagram execute the *kP* -operation and are connected with each other through a bus channel. The FUs X1, Z1, X2 and Z2 are 233-bit long registers. The FUs Multiplier (denoted as 'mult' in section II) and ALU perform mathematical operations in *GF*(2^{233}).

- **Controller**: This FU is described as a 32-bit register. The Controller manages the work of all other FUs. The bits 0-27 of this register manage the access of all other FUs to the bus. This way, the process of the *kP*-operation is controlled. Fig. 8 shows the single bits of the *cntr* signal next to their corresponding inputs on the other FUs. The input signal *is_set* determines if a key bit with value '1' or with value '0' should be processed. Depending on this and on its current internal status, the value of the output signal *cntr* is set. A detailed description of *cntr* is presented in the next subsection of this section

- **Bus channel**: This entity is responsible for the data exchange between all other FUs in the design. The value on its input signal *cntr*(27-24), bits 27 to 24 of *cntr*, determines the FU who's output data is to be written to the bus. The bus transfers this data to the inputs of all other FUs immediately. The Controller decides which of these FUs should save this data in their internal register. All data inputs and outputs are connected this way through the bus.
  Additionally during the execution of the *kP*-operation the bus reads once the output data from the registers *x* and *b* of the ECC design. These two registers are not shown in Fig. 8.

- **external registers X1, Z1, X2, Z2**: These entities are 233 bit long storage registers. If their input signal we has the value '1', the data from the bus will be saved in the register. These registers are used to latch values during the execution of the Montgomery *kP* algorithm.

- **ALU**: The arithmetic logic unit (ALU) has a 233-bit long data input and a 233 bit long data output. The results of its operations are saved in an internal 233 bit long register. All values saved on this register are automatically sent to the ALU's output. The ALU is responsible for two arithmetic operations: addition and squaring of the elements of $GF(2^{233})$.
  When the ALU's input signal we has the value '1', the data on its input (that is, the data from the bus) is saved in the internal register. When the signal *xe* has the value '1' the data on its input is added (XORed) to the data saved in the internal register. This means that the addition of two elements of $GF(2^{233})$, that is the bitwise XOR of two big binary numbers, needs two clock cycles to be executed. When the ALU's input signal *sqe* has the value '1', the data on its input is squared and saved in its internal register.

- **Multiplier**: When the input signal seta has the value '1', the Multiplier saves the data from the bus as the first operand in one internal register. The second operand to be multiplied with the first one is saved from the bus in another internal register one or two clock cycles later when the signal *setb* has the value '1'. Once both operands have been saved, the actual multiplication process begins and is finished after 9 clock cycles. This means that the Multiplier needs in total 11 clock cycles in order to complete a multiplication: 2 cycles for saving both operands and 9 cycles for the execution of the operation.

B. *Control Signal*

The block diagram in Fig. 8 in the previous subsection shows how the FUs in the design are connected with each other. This diagram shows that the Controller's output signal *cntr* serves as an input signal for all other FUs, controlling the bus access to their inputs and outputs. This subsection describes the configuration of this signal and the function of its single bits.

The signal *cntr* is a 32-bit long signal, which has been implemented as the output signal of the FU Controller. Depending on its internal state, the Controller sets the value of this signal and sends it as separated bits to all other functional units of the ECC design. Fig. 9 illustrates the configuration of the *cntr* signal, portraying its single bits and the corresponding names of the FU's inputs to which they are directed.

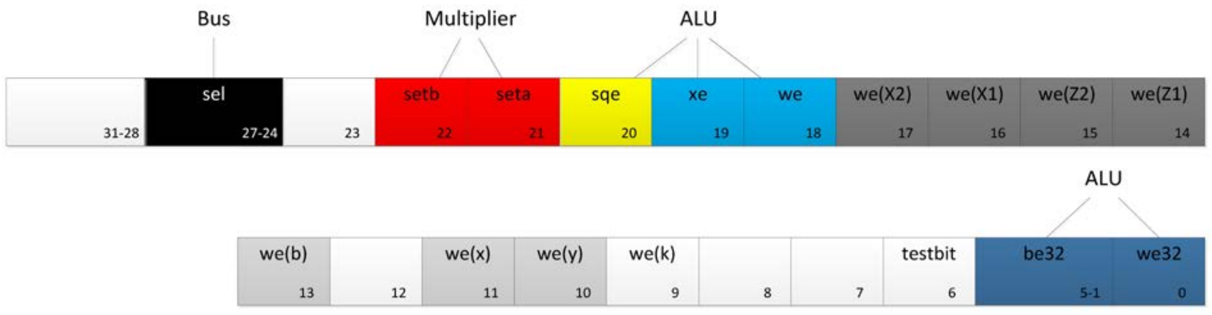


Fig. 9. Configuration of the signal *cntr*.

The bits 31-28, 23, 12, 8 and 7, represented in white boxes in Fig. 9, remain unused by the Controller for the analysed process of the *kP*-operation. This subsection presents a detailed description of the functionality of the other bits and bit groups of this signal.

- **Bits 27 to 24**: This group of bits is the input data of the bus. The value of this bit word indicates which FU's data output should be written to the bus. The values represented as decimal numbers in TABLE I. determine the bus access to the output data of the corresponding FUs. The names given on the table for the signals are the names used on the design's source code file ecc_233.vhd. These are the names used for the port mapped signals between the functional units in the design.

TABLE I. DESCRIPTION OF THE POSSIBLE INPUT VALUES FOR THE BUS

signal *sel*, used value	**signal name**	**description**
0	*o_ext_reg_r_out*	output of external regiser is written to the bus
1	*o_b_r_out*	output of register b is written to the bus
2	*o_z1_r_out*	output of register Z1 is written to the bus
3	*o_z2_r_out*	output of register Z2 is written to the bus
4	*o_x1_r_out*	output of register X1 is written to the bus
5	*o_x2_r_out*	output of register X2 is written to the bus
6	*o_multiply_result*	output of Multiplier is written to the bus
7	*o_alu_r_out*	output of ALU is written to the bus
8	*o_x_r_out*	output of register x is written to the bus
9	*o_y_r_out*	output of register y is written to the bus

The bus receives four bits from *cntr*, bit 27 to bit 24, as the input signal *sel*. The binary number represented by this 4-bit word determines the access control of the bus. This means that depending on the value of *sel*, the output from one FU of the design is written to the bus as input data for other FUs.

- **Bits 22 and 21**: These bits are sent to the inputs *seta* and *setb* of the FU Multiplier respectively. If any of these bits has the value '1', the Multiplier saves the value from the bus in one of its internal registers and uses it as one of the multiplicands for the next multiplication. The multiplication starts as soon as both values have been set.
- **Bits 20, 19, 18**: These bits are sent to the inputs *sqe* (square enable), *xe* (xor enable), and *we* (word enable) of the functional unit ALU respectively. The value on these inputs determines the operation to be executed by the ALU with the data from the bus on the current clock cycle: This way input data of the ALU can either be squared (*sqe*) and saved in the ALU's internal register, only saved (*we*) in the internal register, or added (*xe*) to the value already saved in the internal register. Everything saved in the ALU's internal register is automatically placed on the ALU's output.
- **Bits 17, 16, 15, 14**: These bits are in charge of controlling the functionality of the registers X2, X1, Z2 and Z1 respectively. For each bit with value '1', the corresponding register saves its input data at the current clock cycle. This way values can be latched within the process.
- **Bits 13, 11, 10, 9**: These bits are in charge of controlling the functionality of the registers *b*, *x*, *y* and *k* respectively. For each bit with value '1', the corresponding register saves the actual value located on its input data at this clock cycle.
  These bits are only used outside the Montgomery algorithm's loops[2] for specific purposes. The value of bit 13 is never set to '1'. Bits 11 and 10 are used only once each at the end of the process to set the calculated values of the coordinates *x* and *y* to the registers *x* and y. Bit 9 is used regularly in the finalization phase after all loops in the algorithm have been processed.
- **Bit 6**: This bit is in charge of controlling the functionality of the test bit register. When this bit has the value '1', the test bit register saves the actual value located on its input data at this clock cycle.
- **Bits 5 to 1 and bit 0**: These bits are sent to two ALU inputs, *be32* and *we32* respectively. When *we32* has the value '1', a 32 bit long value standing on the data input *r_in32* of the ALU can be saved as part of the 233 bit long internal register of the ALU. The value on *be32* specifies which 32 bits of the internal register should be overwritten by this new value.

[2] Loop is another term for referring to a slot.

The value of bit 0 is set to ‘1’ only once in the initialization phase in order to set the value of the ALU’s internal register to 1.

The following subsection describes the calculation processes of the *kP* -operation, which is controlled by the *cntr* signal.

C. *Montgomery Algorithm Implementation*

The biggest amount of time needed for the performance of the *kP*-operation with this design is spent in the inner loop of the Montgomery algorithm. The number of times this loop is performed depends on the value, or rather the length, of the key *k* [6]. The initialization and finalization phases are each performed only once. This is why the most relevant processes for this assessment have been the inner loops. Being dependable from k, these processes play a crucial role in the performance of a side channel attack on this implementation. This subsection describes the IHP implementation of Montgomery algorithm for the *kP*-operation, that is the sequence of the operations performed in the 2 possible cases for the inner loop.

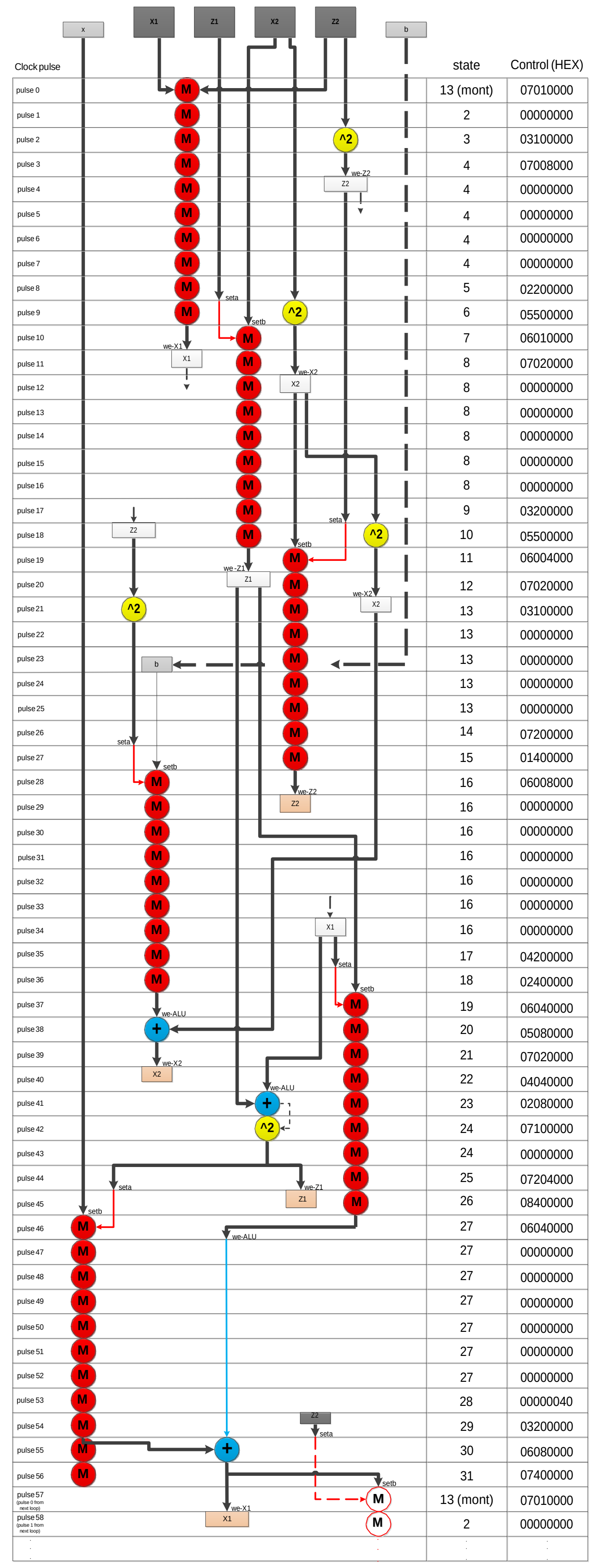
As it is described above, the Montgomery algorithm consists of 3 steps: the initialization, the inner loops (for the cases $k_i=0$ and the $k_i=1$), and the finalization. These 3 steps are implemented in the VHDL file controller_9_FPGA.vhd under the names ***mont***, ***montk1*** (for $k_i=1$), ***montk0*** (for $k_i=0$) and ***montpost*** respectively. Two variables - the *state* and the *control* - organize the calculation of the *kP* product. The variable *state* is responsible for the running step of the Montgomery *kP* algorithm, i.e. it can start the programs for ***mont***, ***montk1***, ***montk0*** or ***montpost*** and it can set the value of the variable *control*. The value of the *control* activates the FUs of the ECC design and the data exchange between them. Two flow charts titled ***montk1*** and ***montk0*** describe the operation flow for both cases of the inner loop (see Fig. 10 and Fig. 11). Here is a short description of the structure of these flow charts:

- On the left side of the diagrams the clock cycles are listed. Each loop consists of 57 cycles.
- The column named ‘*state*’ lists the states of the variable program during the loop.
  Each flow chart describes the complete process operated by one program, either ***montk0*** or ***montk1***. The *state* described on the cycle 0 of each chart (*state 13* (***mont***) for every case) is part of the program ***mont***. The program ***mont*** is the initialization program and it is always entered on its *state 13* for the first cycle of each loop. During this clock cycle, the first multiplication process for a loop starts and the Controller also decides, if a loop for a key bit with ‘1’ or for a bit with ‘0’ should be entered: if the Controller’s input signal *is_set* has the value ‘1’, *state 2* of ***montk1*** is entered on the cycle 1 of the loop. If *is_set* has the value ‘0’, *state 2* of ***montk0*** is entered on the cycle 1 of the loop.
- On the right side of the flow chart the values of the Controller’s *cntr* signal are listed. They are given as hexadecimal numbers, as described in the VHDL file.
- The actions denoted as “we-” represent a *word enable* operation performed by the indicated register.
- The red units with the label ‘M’ represent the activity of the Multiplier. Units displayed with a solid red color represent the current activity of the Multiplier. Units displayed with a transparent red color (cycles 9, 55 and 56 for both programs) represent a non-operating state of the Multiplier. The result obtained from the last operation is saved and can be read. Each loop contains 6 multiplications. The non-filled units with the label ‘M’, placed on the cycles 57 and 58 of each loop represent multiplications which belong to the following loop.
- The yellow units with the label ‘ˆ2’ represent the squaring operation of an element of $GF(2^{233})$ performed by the ALU. Each loop contains 5 squaring operations. Each squaring operation needs one clock cycle.
- The blue units with the label ‘+’ represent the addition of two elements of $GF(2^{233})$ performed by the ALU. This operation needs two clock cycles for the calculation of the sum: at the first cycle the first operand is saved in the inner register of the ALU (when its *word enable* signal is set to ‘1’). At the second clock cycle the second operand is directly added to the first if its *xor enable* signal is set to ‘1’. There are 3 additions performed on each loop.
- All registers X1, Z1, X2, Z2 are shown in grey color when they are being used as input parameters on the beginning of the loop. They are white colored when being used as latches and beige colored when they store the final values (output values) of the loop.
- The programs ***montk0*** and ***montk1*** have been implemented in the same way.

In the Montgomery algorithm, both loops require exact the same number and the same sequence of operations to be executed. They differ only in the use of the registers (X1 and X2; Z1 and Z2) as input and output parameters [4]. In the IHP implementation ***montk1*** and ***montk0*** differ as follows:

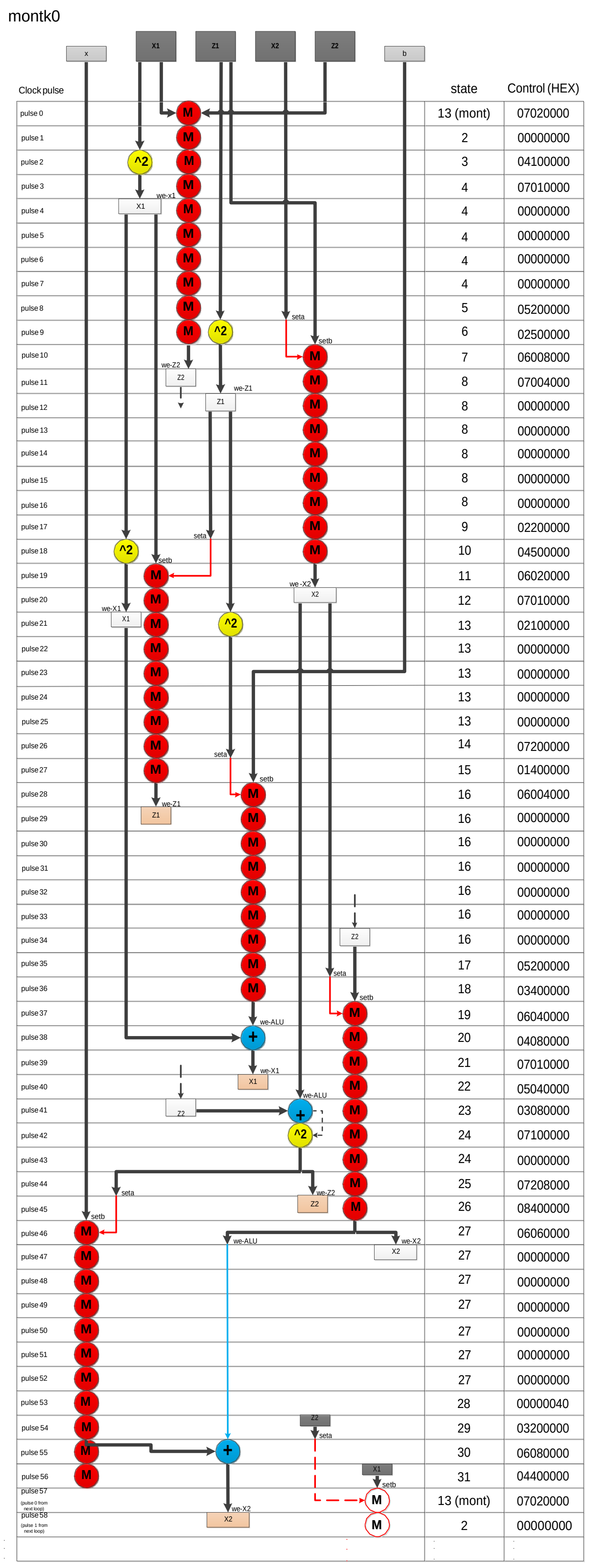
- The sequence of operations of ***montk0*** differs from the sequence of operations of ***montk1***: on the clock cycle 2 of ***montk0*** a square operation with the value of X1 is performed. The corresponding operation for this clock cycle in ***montk****1* would be a square operation with the value of X2. Nevertheless the operation performed at this clock cycle in ***montk1*** is a square operation with the value of Z2. In the clock cycle 9 of ***montk0*** the value of Z1 is squared, while in cycle 9 of ***montk1*** the value of X2 is squared.
- On cycle 46 of ***montk0*** the *we*- operation is performed by the ALU and another one by the register X2. This means, two *we*- operations are performed on this cycle for ***montk0***. On cycle 46 of ***montk1*** in contrast, only one *we*- operation is performed: *we-ALU*. The performance of the operation *we-X2* is not needed since the value of the register X2 is overwritten again in the clock cycle 0 of the following loop, being this the last operation performed by ***montk0***.
  Between the clock cycles 46 and 57 (clock cycle 0 for the next loop), the value of X2 is never read. It is believed that the performed we-X2-operation has been implemented by mistake.
- On cycle 56, i.e. in the last clock cycle in the loop, a *setb* operation is performed for setting the second multiplicand for the multiplication for the next loop. In ***montk0*** the value of the multiplicand is obtained from register X1; in ***montk1*** this value is obtained from the functional unit ALU.

montk1



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2	00000000
. . .	. . .	. . .

Fig. 10. Oparation flow for the program ***montk1***.



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2	00000000

Fig. 11. Oparation flow for the program ***montk0***.

V. Observations

The flow charts shown in the previous section describe the inner loops of the Montgomery algorithm by depicting every operation being executed in each clock cycle. An analysis of these flow charts has been a key part for the realization of this security assessment. Figures in Appendix A show the complete flow diagram for the *kP*-operation with the use of a small key *k=2cc*. To illustrate the leakage sources, the power consumption is shown in the same diagram as traces for each FU of ECC designs.

This section lists the leakage sources that were detected through this assessment. These leakage sources are explained based on observations made on the implementation's characteristics that have been listed in the previous sections and in the simulated power traces of the FUs of the ECC design.

1. Easy identification of the boundaries between the loops.
   Periodically repeated forms of the power trace simplify the detection of how long (how many clock cycles) individual key bits need to be processed. Power Traces for the analysed design show two of such characteristic forms: one being a "plateau" in the curve and the other one being a significant dip. Both of these characteristics appear right after each other in all power traces for each bit processing, also for each loop. Fig. 12 shows these characteristic changes of the power trace.

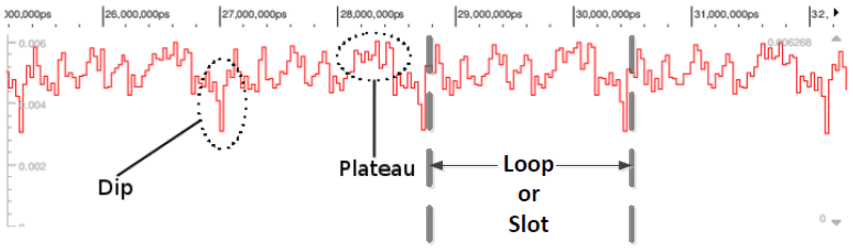


Fig. 12. Power Trace of the ECC design during the performance of the *kP*-operation. The "dips" in the trace's curve are caused due to a very low power consumption of the design during one clock cycle. A few clock cycles before each "dip" a "plateau" can be observed. Each loop (or slot) of the *kP*-algorithm takes 57 clock cycles, i. e. the time between two dips is 57 clock cycles.

The shown in Fig. 12 part of simulated Power Traces illustrates the power consumption of the design for the entire process. This power consumption is strongly caused by the periodic activities performed by the IC. The following conclusions have been made related to both of these anomalies through analysing the flow charts for the inner loops of the Montgomery algorithm:

- ✓ **Plateau**: This object is seen between the clock cycles 38 and 42 (5 cycles in total). Three operations are performed by the ALU during this time: two additions and one squaring. This amount of operations is only performed in this short period of time between these clock cycles and it demands constant activity from the ALU during this period, thus leading to a continuous high power consumption from this FU. Such an amount of activities in a similar time interval is not performed in any other period during the loop.
- ✓ **Dip**: This anomaly is seen at the end of each loop. The reason for the dip is the fact, that consuming most energy of all FUs during the last two clock cycles of each slot (i.e. clock cycle 55 and 56) the unit Multiplier is not active. After cycle 54, the last multiplication in the loop has been completed and its result can be read. The Multiplier stays in its non-active state for the following two clock cycles because it needs to wait for its next input parameters. The Multiplier's energy consumption reaches a value close to zero during cycle 56 when none of its gates are switched. This implies a big decrease of the energy consumption of the entire design at this clock cycle.

2. Unbalanced implementation of the ECC design.
   The processing of each bit of the key depends on the value of the processed key bit. (see section IV-C).

3. The length of a Power Trace depends on the size of the key.

   By using a key of small length, the power trace of the *kP*-operation has a short size (in time). This could let an attacker know about the size of the key and thus providing him information about the implemented algorithm or letting him know that the performance of a brute force attack is possible in case of a key with a short length. Fig. 13 displays the entire initialization process for the *kP*-operation. Some activities can be seen on the beginning and then, depending on the length of the key, no operations are performed for a certain time. Operations are performed again as soon as the processing for the first key bit starts.

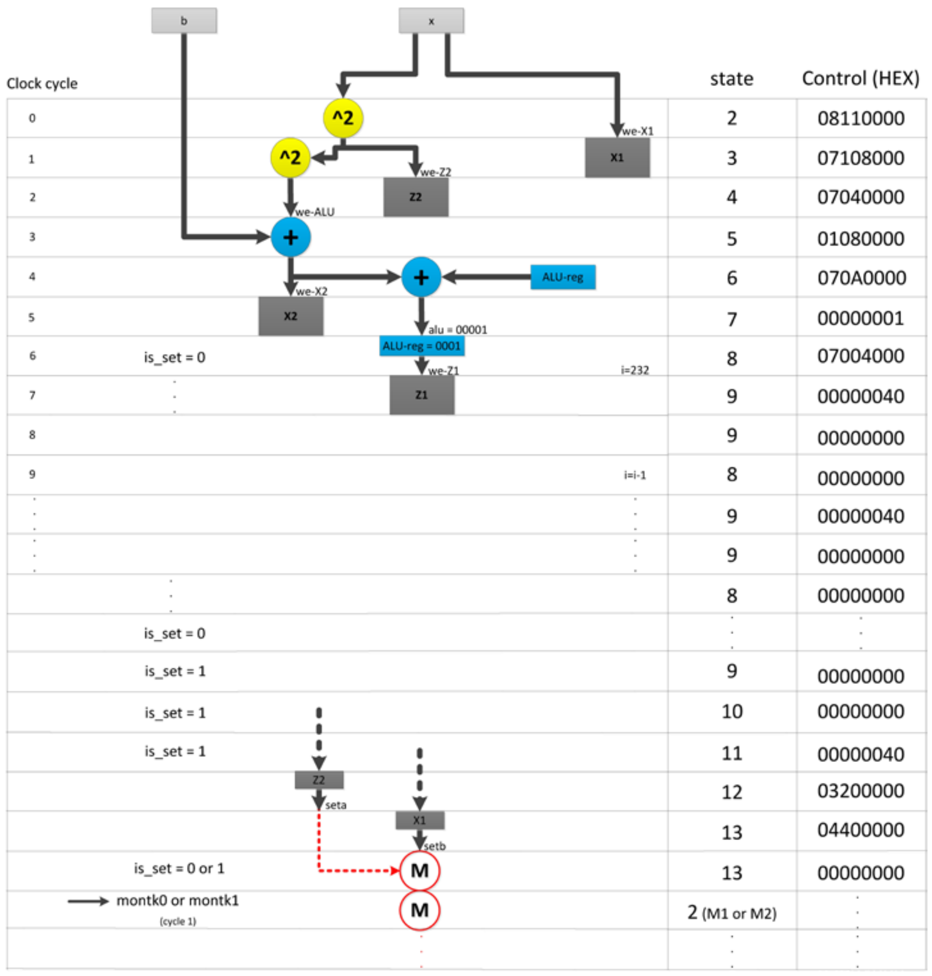


Fig. 13. Flow chart of the initialization part of the Montgomery *kP*-algorithm.

VI. CONCLUSION

The structure of the IHP ECC design and its functionalities are described in this report. The observations made in this report helped identify the reasons that caused this system’s vulnerabilities to side channel attacks. The design’s functionality and its power consumption are displayed as a diagram. This helps to understand, which operations can be potential leakage sources for successful side channel analysis attacks. By taking these properties into consideration when implementing crypto-algorithms, more resistant implementations can be developed.

ACKNOWLEDGEMENTS

The work presented in this report has been partially funded by the Ministry of Sciences, Research and Cultural Affairs (MWFK) from resources of the European Social Fund (ESF) and of the state Brandenburg under grant number 1389053.

REFERENCES

[1] IHP - http://www.ihp-microelectronics.com/en/start.html

[2] IHP - Research - Project MASCH http://www.ihp-microelectronics.com/ de/forschung/drahtlose-systeme-und-anwendungen/projekte/masch.html

[3] NIST Computer Security Division - Digital Signature Standard (DSS), FIPS 186-3, http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[4] J. Lopez, R. Dahab - Fast multiplication on elliptic curves over GF (2m) without precomputation, Proceedings of the First International Workshop Cryptographic Hardware and Embedded Systems, Springer-Verlag, 1999

[5] Z. Dyka, P. Langend¨orfer - Area Efficient Hardware Implementation of Elliptic Curve Cryptography by Iteratively Applying Karatsuba's method, Proceedings of De- sign Automation and Test in Europe (DATE) Conference, IEEE Society Press, 2005

[6] S. Peter - Evaluation of Design Alternatives for Flexible Elliptic Curve Hardware Accelerators-Diplomarbeit, BTU Cottbus, 2006

[7] IHP - Forschung - Projekt TAMPRES http://www.ihp-microelectronics.com/ de/forschung/drahtlose-systeme-und-anwendungen/projekte/tampres.html

[8] Synopsis - PrimeTime http://www.synopsys.com/Tools/Implementation/ SignOff/Pages/PrimeTime.aspx

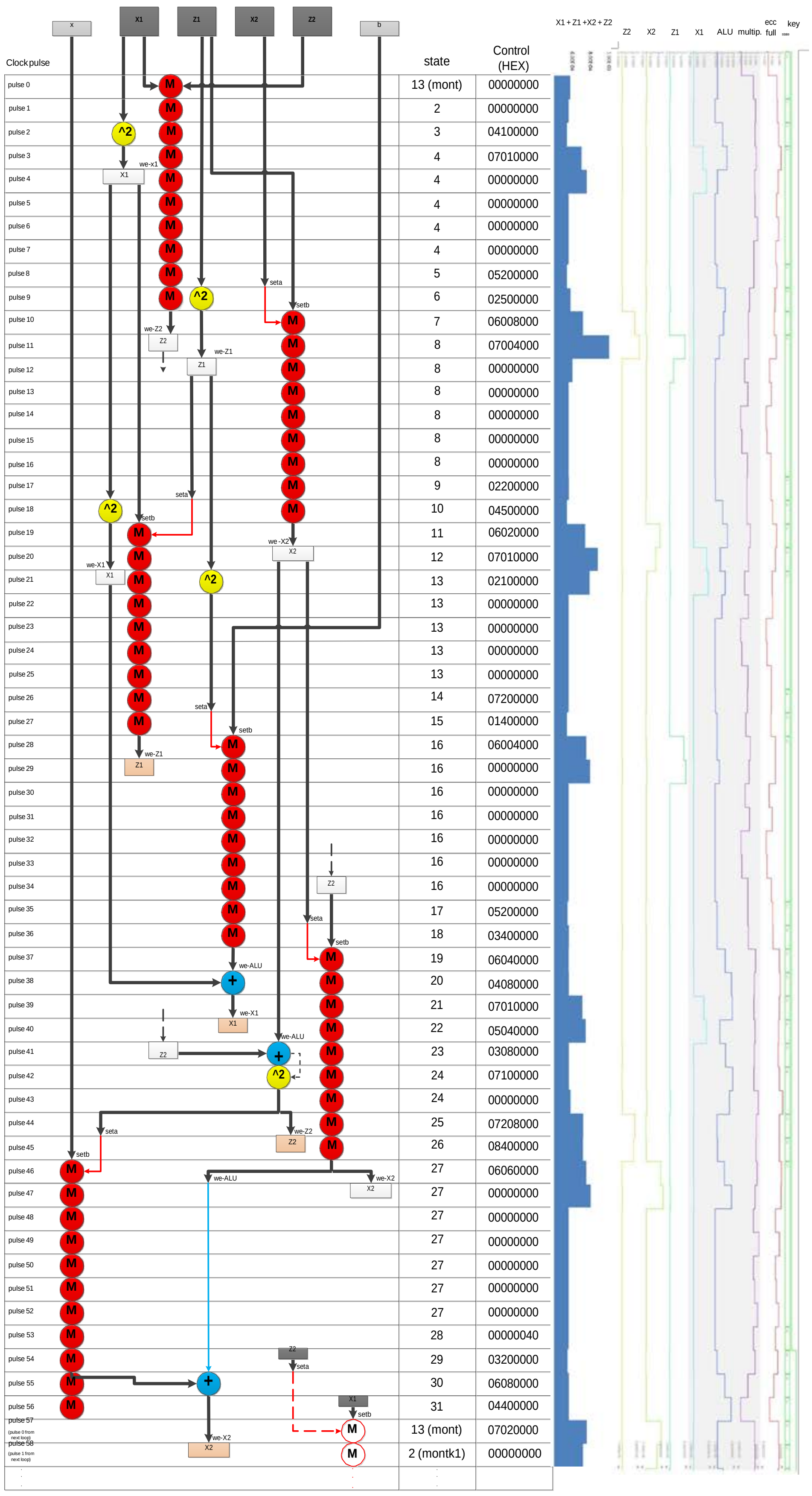
APPENDIX A: FLOW DIAGRAMS WITH POWER TRACES

The Fig. 14-Fig. 22 show the flow diagram for all slots of the *kP*-operation using the EC point $P = (x_1, y_1)$ (see section III) and the scalar $k = 2cc$ in hexadecimal, i.e. $k=k_9k_8k_7k_6k_5k_4k_3k_2k_1k_0=1011001100$ in binary.

The most significant bit of the scalar the (the bit k_9) is processed in the initialization part of the Montgomery *kP* algorithm and is not shown here. To illustrate the leakage sources, the power consumption of the entire ECC design and of its analysed single entities is shown in the same diagram as traces. This way, it is possible to observe how much power is consumed by each entity during the determined clock cycles.

The power consumption of the following entities is shown on the right side of each diagram:

- Complete ECC design
- Multiplier
- ALU
- register X1
- register Z1
- register X2
- register Z2
- The sum of the registers X1, Z1, X2, Z2



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	00000000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
. .	. .	

Fig. 14. Processing of the $k_8=0$ (i.e. the 8th bit of the scalar $k = 2cc$).

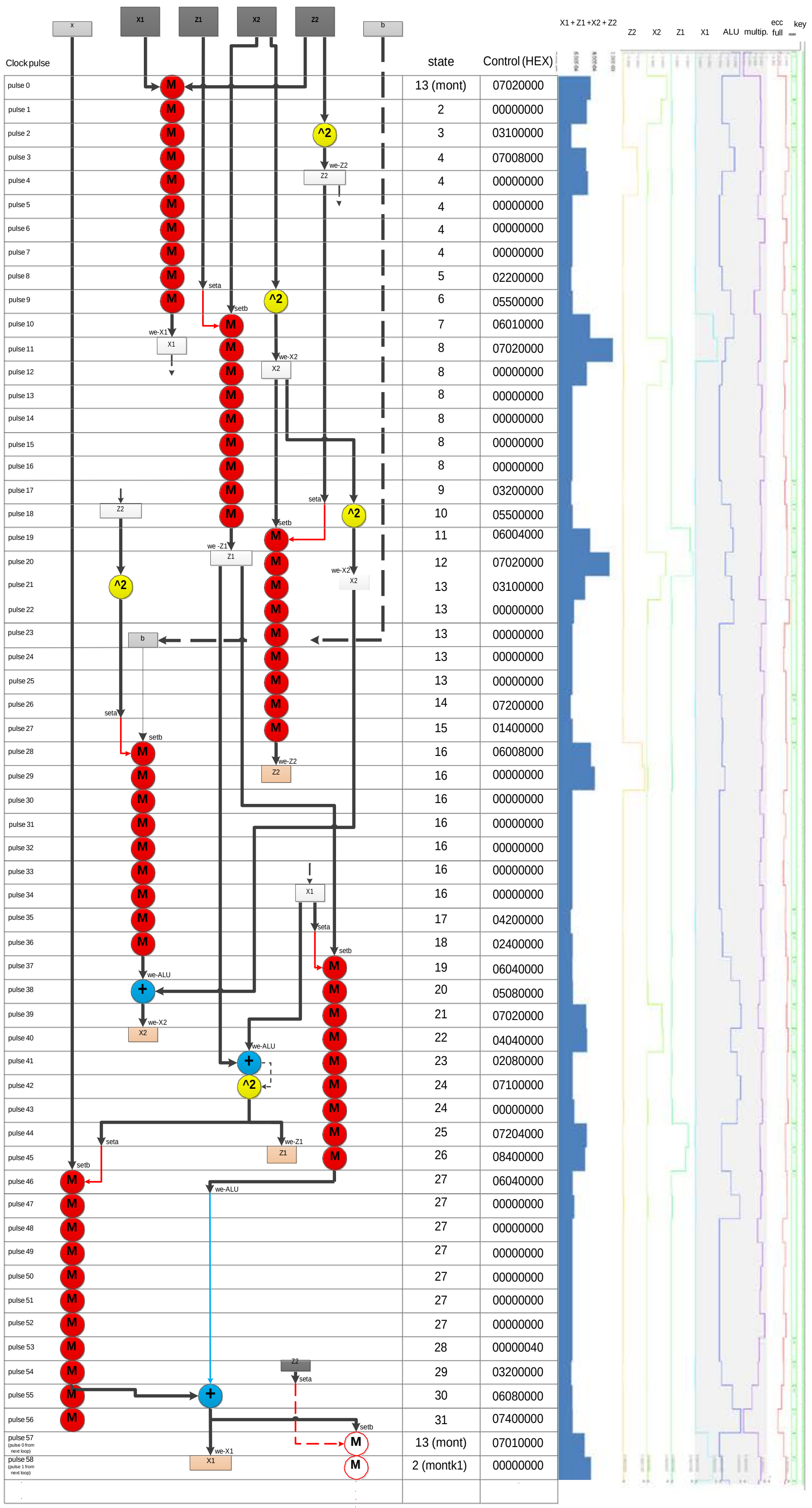
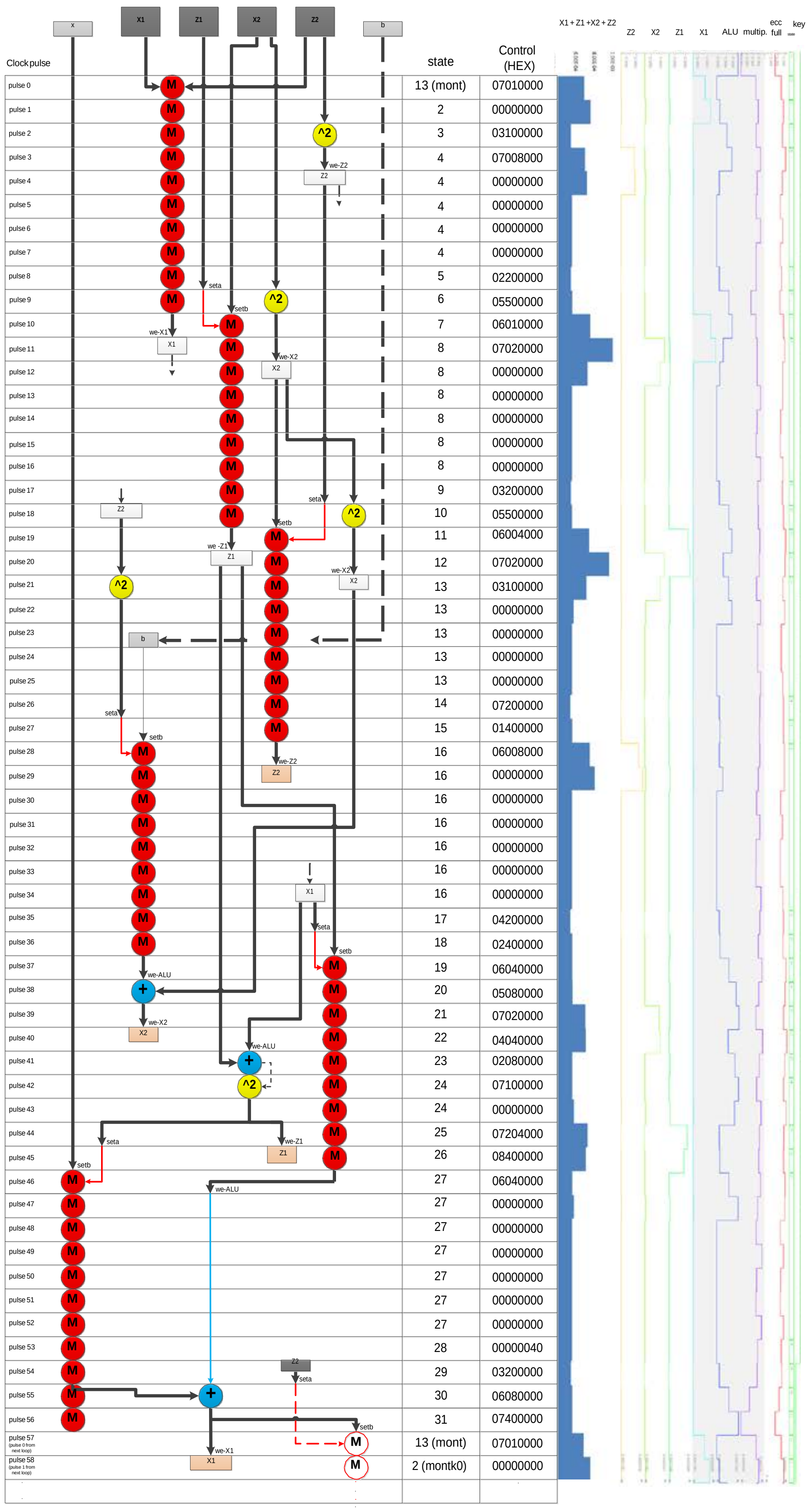
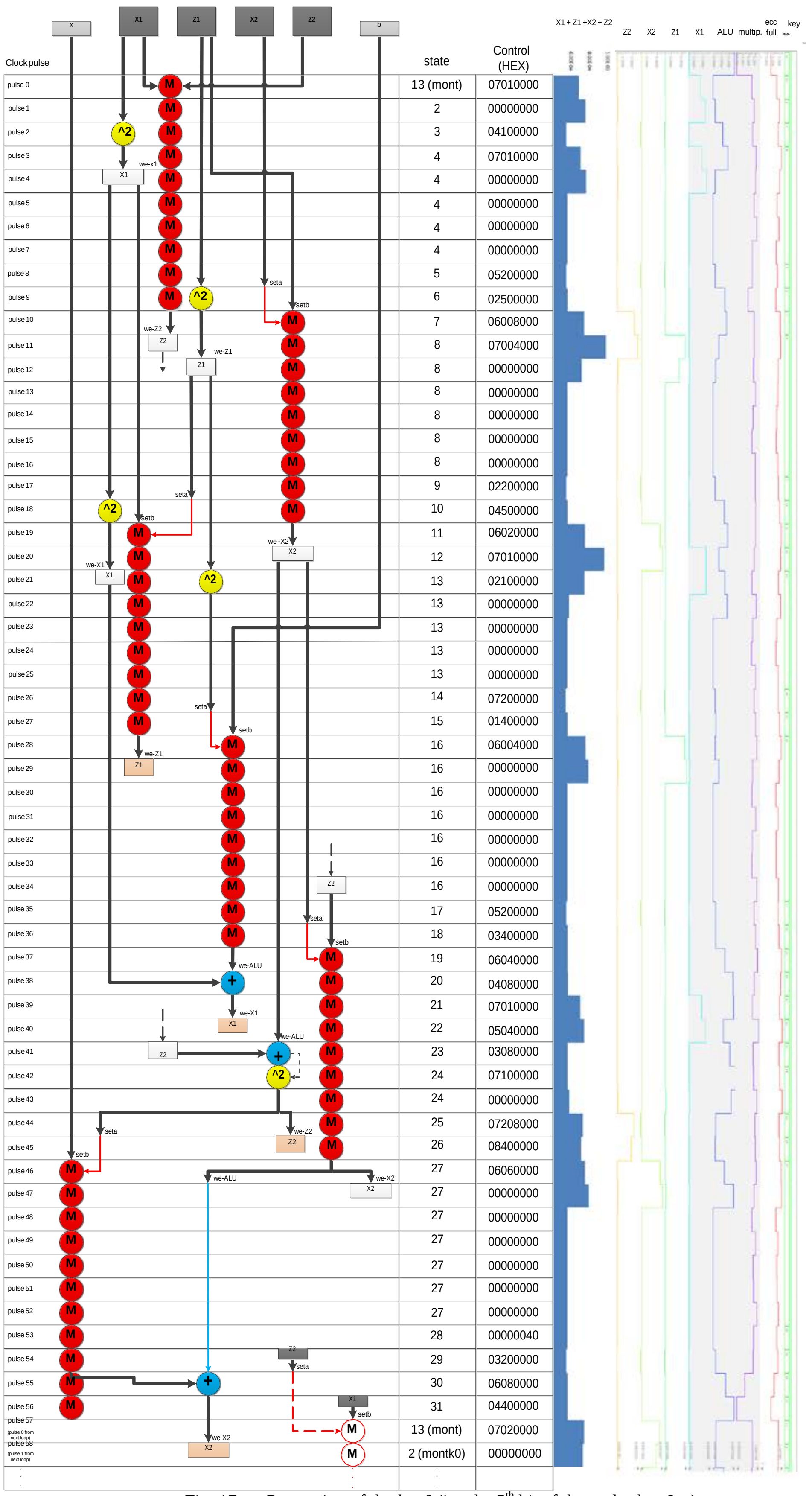


Fig. 15. Processing of the $k_7=1$ (i.e. the 7th bit of the scalar $k = 2cc$).



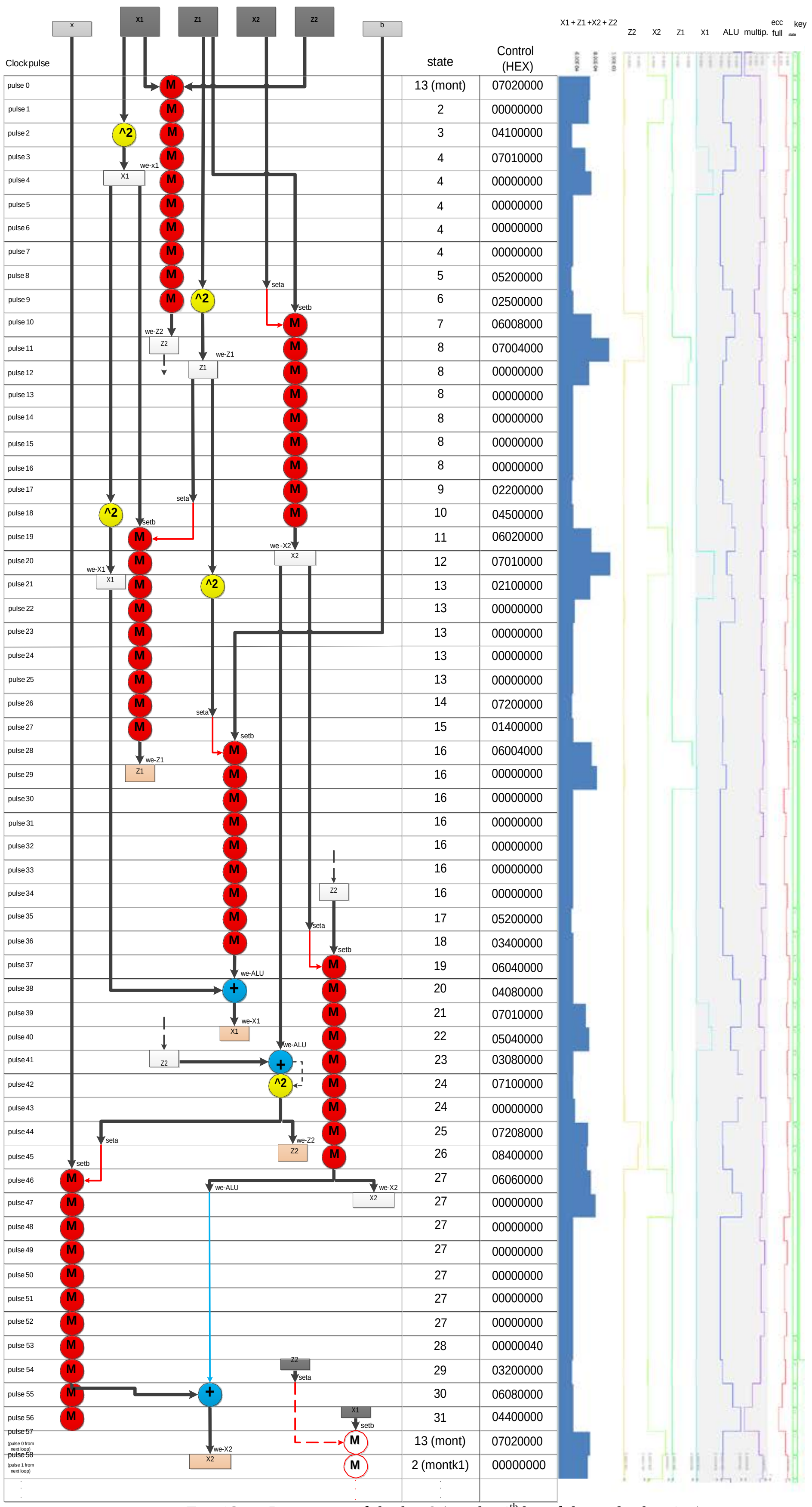
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000

Fig. 16. Processing of the $k_6=1$ (i.e. the 6th bit of the scalar $k = 2cc$)..



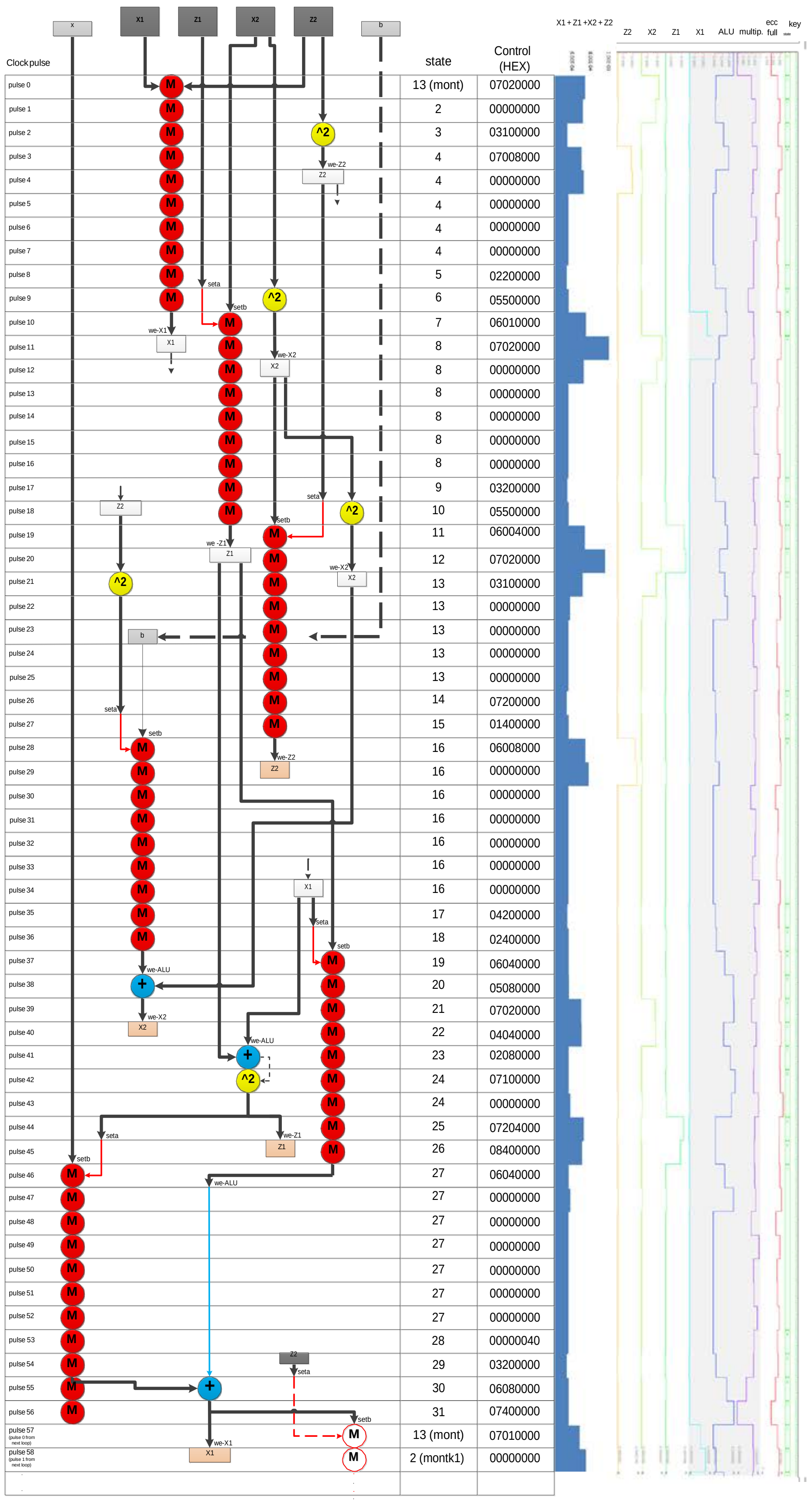
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
. . .	. . .	

Fig. 17. Processing of the $k_5=0$ (i.e. the 5th bit of the scalar $k = 2cc$).



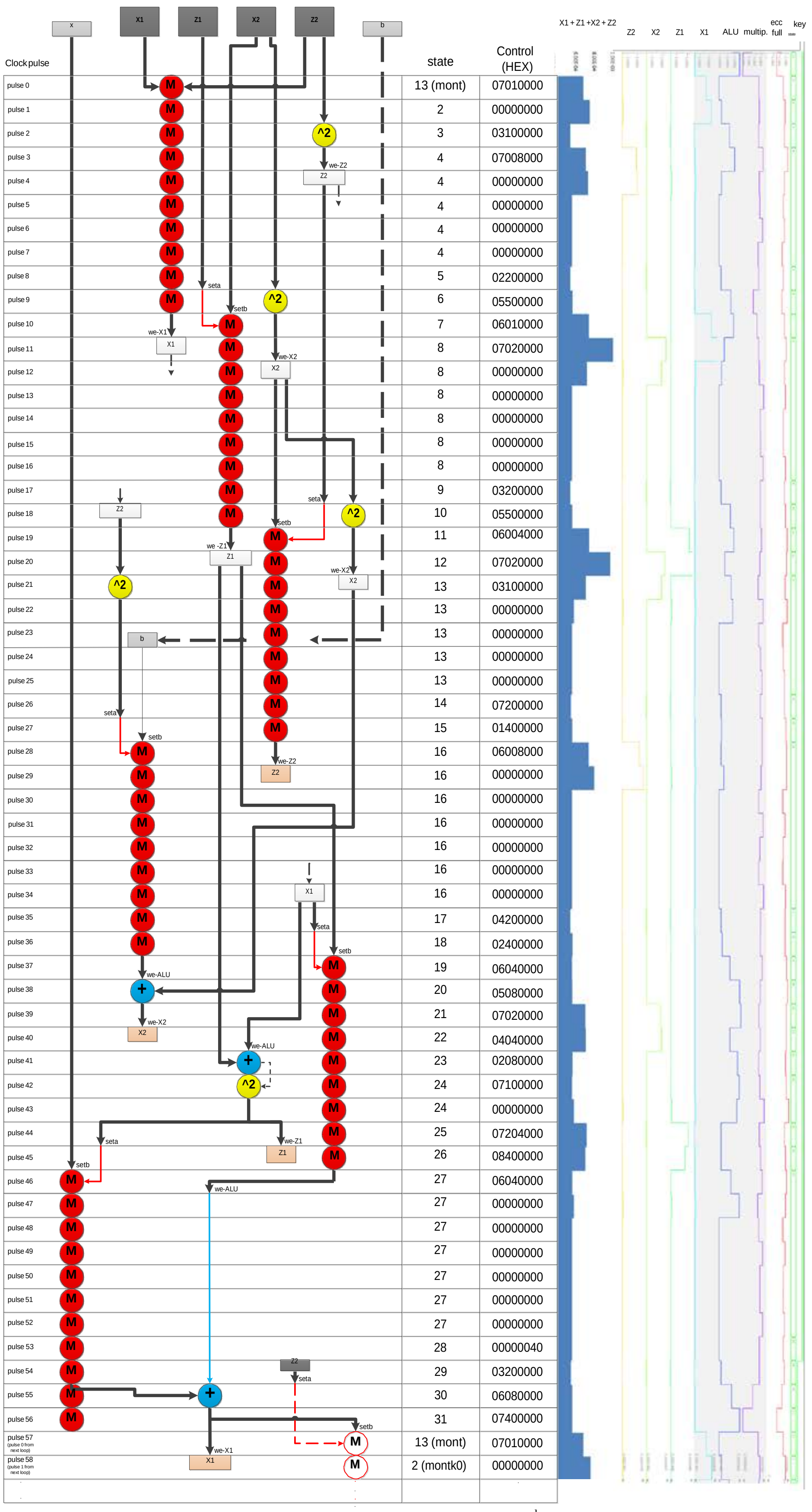
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
.	.	.

Fig. 18. Processing of the $k_4=0$ (i.e. the 4th bit of the scalar $k = 2cc$).



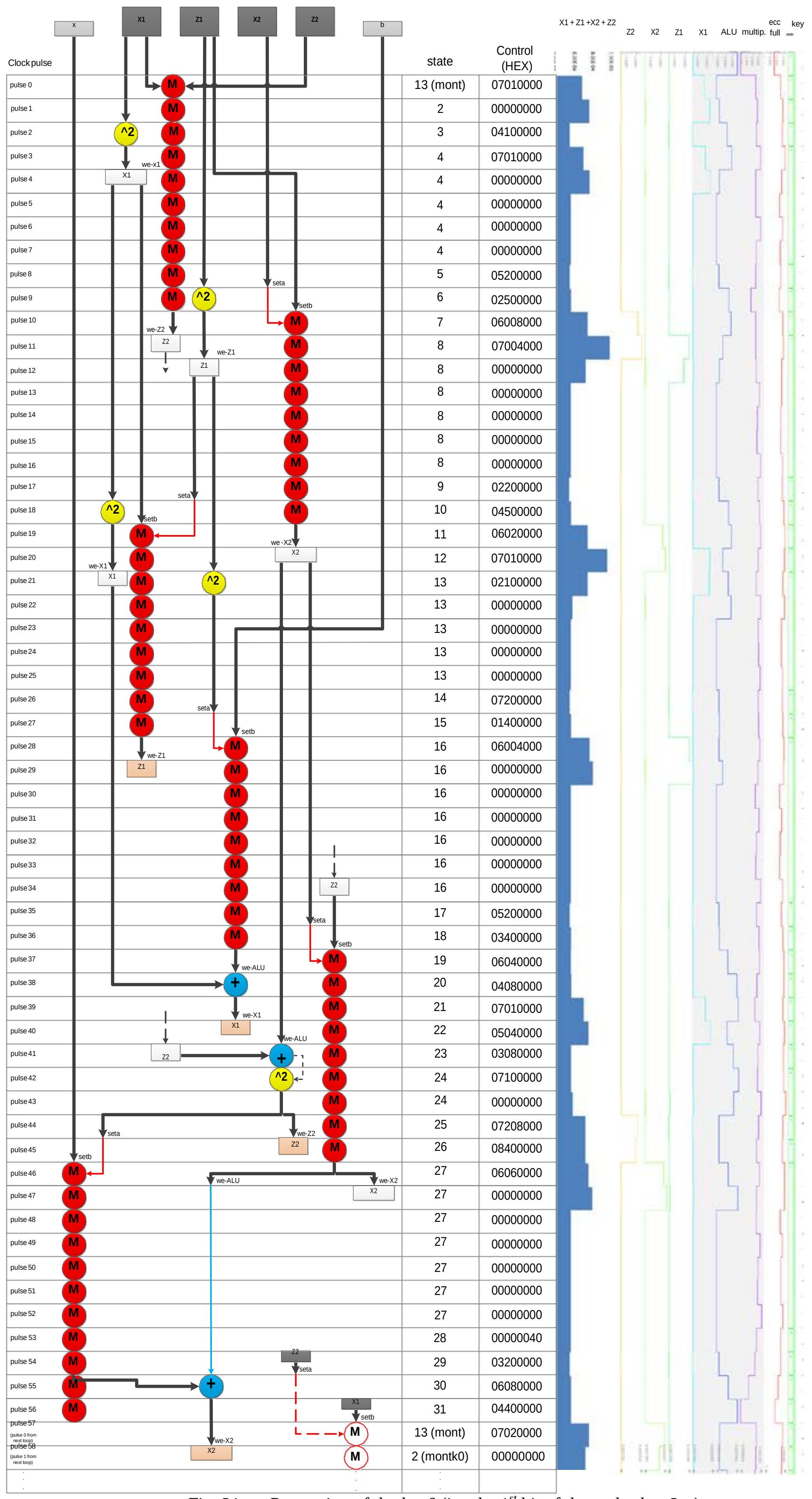
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000

Fig. 19. Processing of the $k_3=1$ (i.e. the 3th bit of the scalar $k = 2cc$).



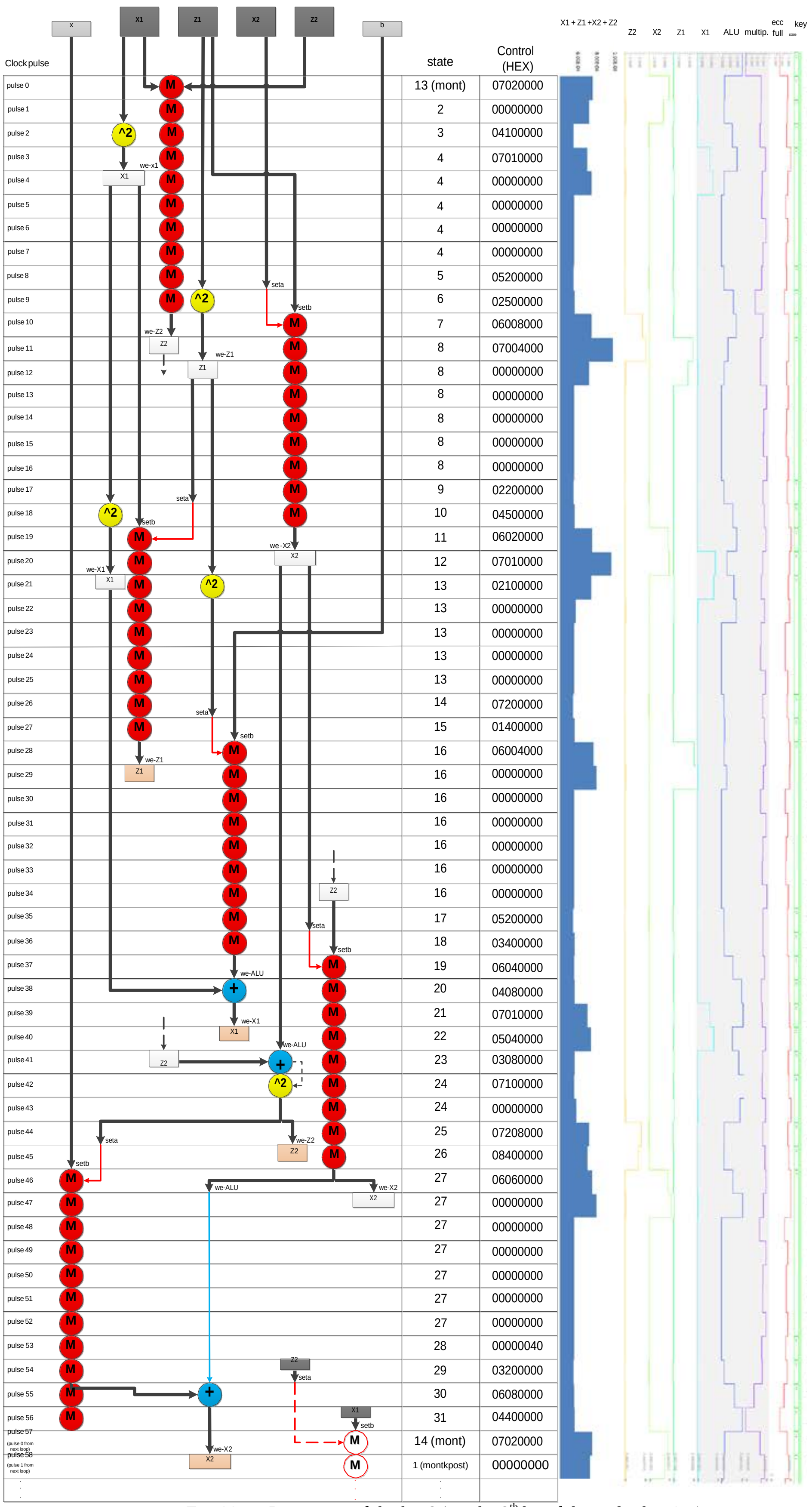
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000

Fig. 20. Processing of the $k_2=1$ (i.e. the 2nd bit of the scalar $k = 2cc$).



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
.	.	.

Fig. 21. Processing of the $k_1=0$ (i.e. the 1st bit of the scalar $k = 2cc$).



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	14 (mont)	07020000
pulse 58 (pulse 1 from next loop)	1 (montkpost)	00000000
. . .	. . .	

Fig. 22. Processing of the $k_0=0$ (i.e. the 0th bit of the scalar $k = 2cc$).