Efficient Implementation of Modular Division by Input Bit Splitting

Danila Gorodecky

National Academy of Science of Belarus,

Minsk, Belarus

danila.gorodecky@gmail.com

Tiziano Villa

University of Verona,

Verona, Italy

tiziano.villa@univr.it

Motivation

– due to historical and algorithmic reasons RNS is not a common arithmetic approach and this topic has not studied proper and not common implemented in hardware computing;

– there are a numeric of a perspective applications such parallel computing, cryptography;

– there is no an “efficient” hardware approach of *X mod P* calculation (not synthesizable in state-of-art EDA tools)

– Digital filtering with finite impulse response (FIR-filtering);

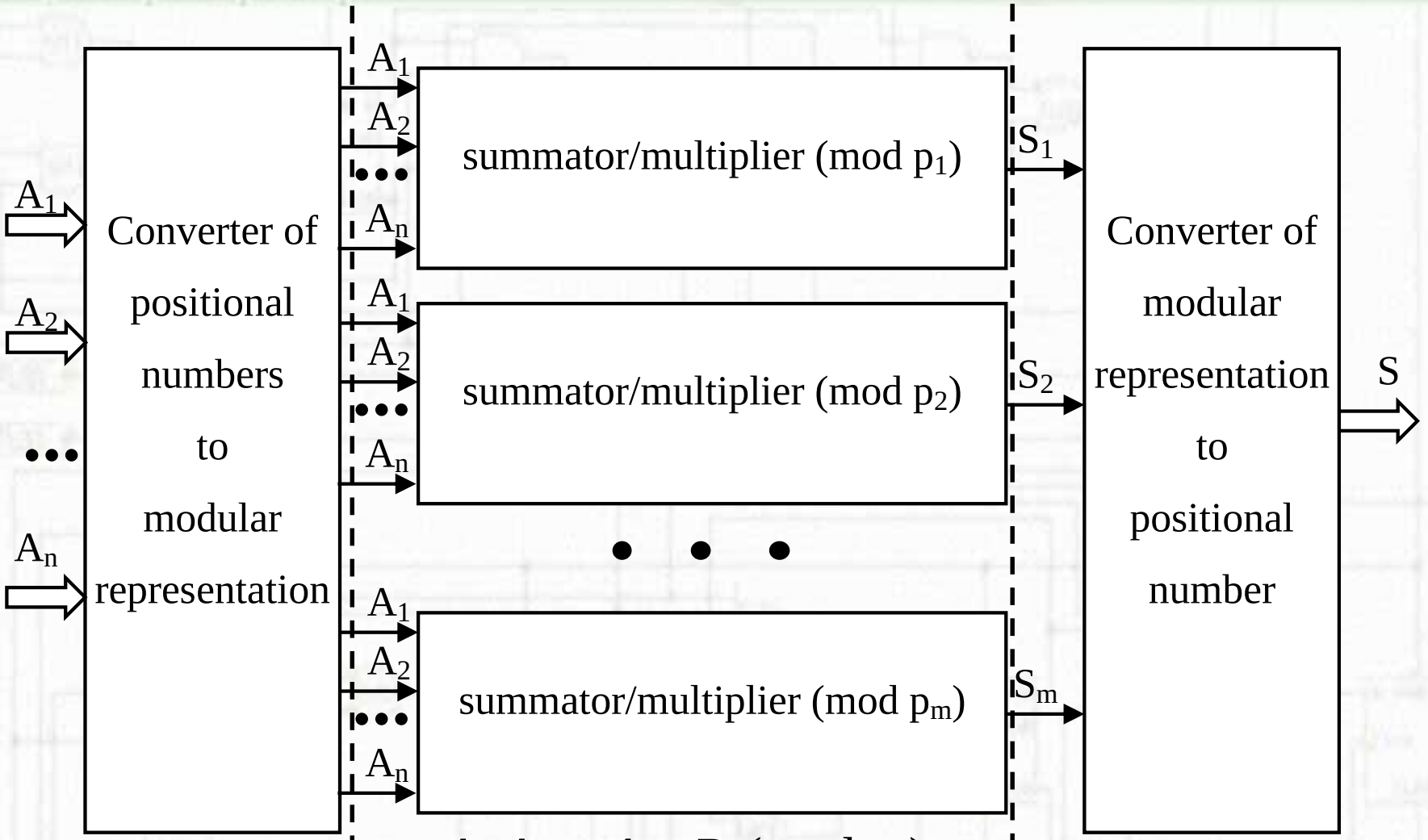
– Crypto system of Federal Reserve System of USA;

– Space flight control (Russia);

– Data transferring between Space satellites and Earth (Russia);

– Air Defense System (USA, Russia);

Common architecture of computation in RNS



$A_i \pmod{p_i}$

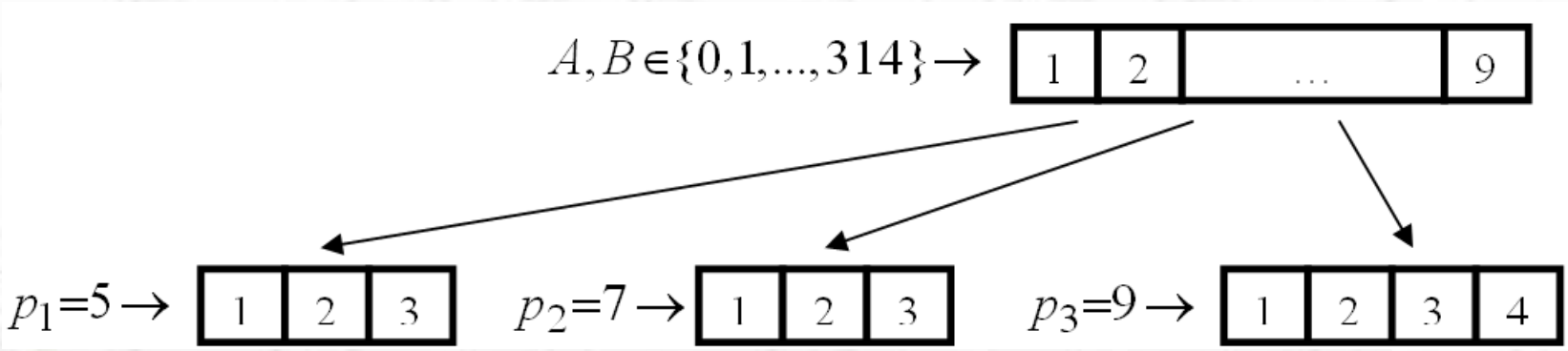
$A_1 \cdot A_2 \cdot \ldots \cdot A_n = R_i \pmod{p_i}$

$A_1 + A_2 + \ldots + A_n = S_i \pmod{p_i}$

$\sum_{i=1}^{m} S_i \quad or \quad \prod_{i=1}^{m} R_i = S \pmod{P}$

$P = \prod_{i=1}^{m} p_i$

Example of computation in RNS



$$A=\{A_1,A_2,A_3\}$$

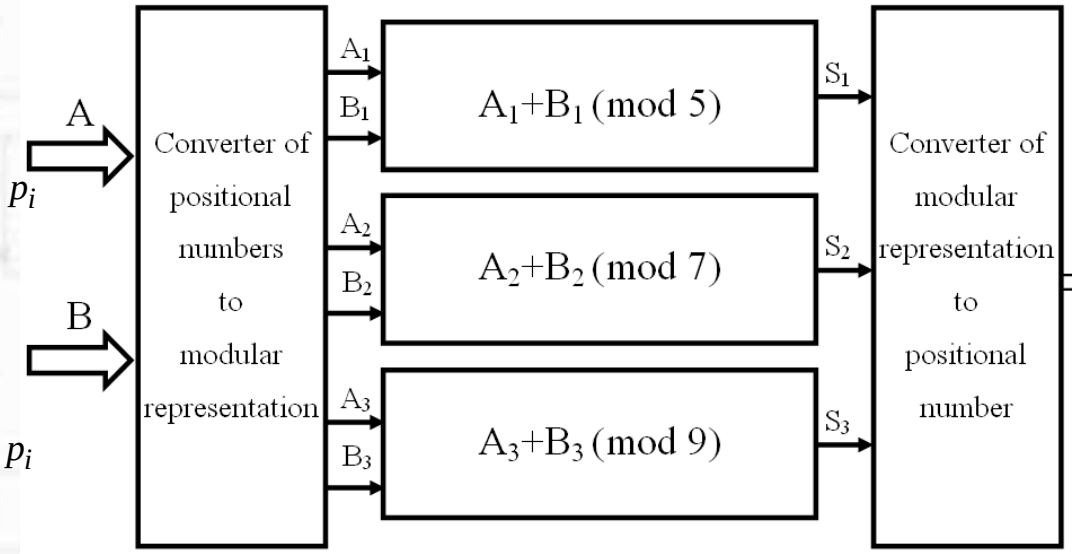
$$A_i \pmod{p_i} = A-\left[\frac{A}{p_i}\right]p_i$$

$$0\le A\le 314$$

$$B=\{B_1,B_2,B_3\}$$

$$B_i \pmod{p_i} = B-\left[\frac{B}{p_i}\right]p_i$$

$$0\le B\le 314$$



$$S=S_1\cdot Y_1+S_2\cdot Y_2+S_3\cdot Y_3-r\cdot P$$

$$\frac{Y_i}{p_i}=1 \pmod{p_i}$$

$$Y_i=\left(\frac{P}{p_i}\right)k_i$$

$$i=1,2,3 \qquad k_i=\overline{1,p_i} \qquad r=0,1,2,...$$

- *X (mod P)* realizations for an arbitrary *P* (forward conversion). Synthesis error in the most of all EDA tools;

- Transformation modular representation into position numbers (reverse conversion). It is a long critical pass or big hardware costs.

Approaches of X (mod P) design

– combination approaches for special moduli sets ($2^s \pm v$ (for $v = 1,3,5$), 2^s, and etc.) [1];

– based on periodic properties of $2^s (mod\ P)$ [1];

– based on modular exponentiation [1];

– shared hardware [1];

– "*Low and Chang*" for arbitrary moduli [1];

– scaled residue computation [1];

– pipelining for an arbitrary *P* [2];

– Computing *mod* without *mod* [3];

– etc.

[1] P.V.A. Mohan, "Residue Number System. Theory and applications", Springer International Publishing, 2016, 351 p.

[2] J.T. Butler and T. Sasao, "Fast hardware computation of x mod z"// 25th IEEE International Parallel and Distributed Processing Symposium Anchorage, Ak, USA, May 16-17, 2011, p. 289-292.

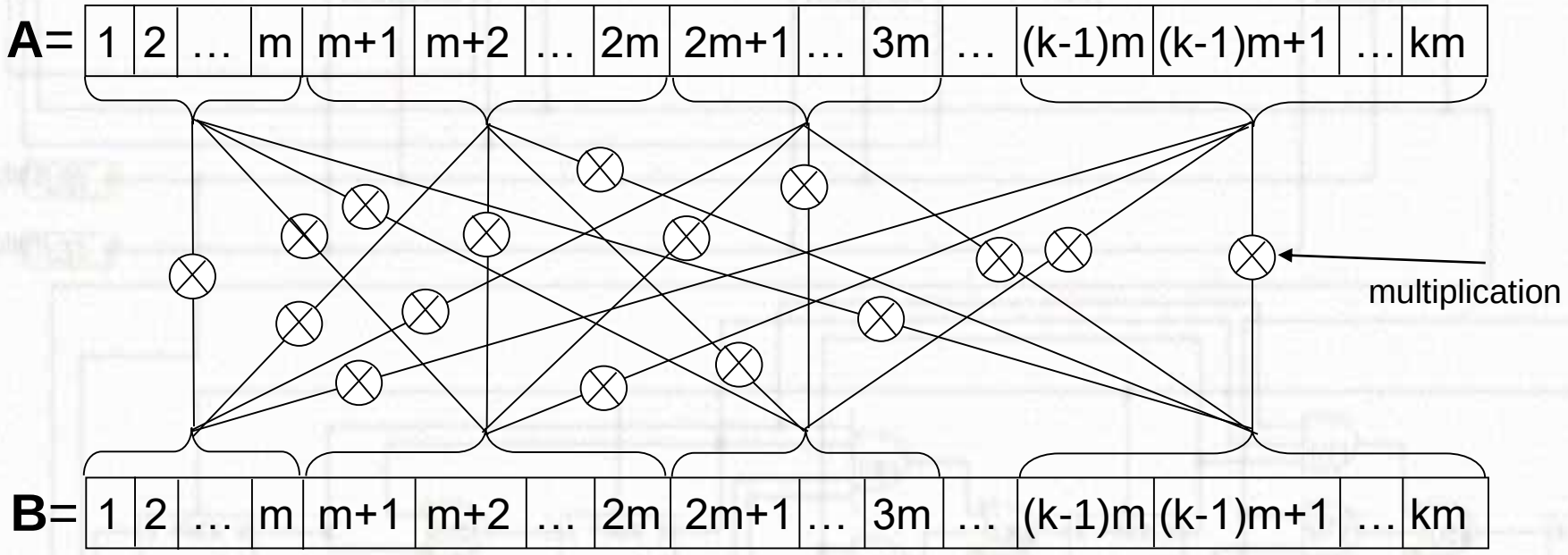
[3] Mark A. Will and Ryan K. L. Ko, "Computing Mod Without Mod"

- splitting input into small tuples (up to 12-bit);

- Boolean minimization (Disjunctive Normal Form, Reed-Muller expansion, Binary Decision Tree, Majority Graph)

Fourier transformation multiplication (FTM)

FTM splits up binary vectors into *m*-bit tuples and multiplies to each others:

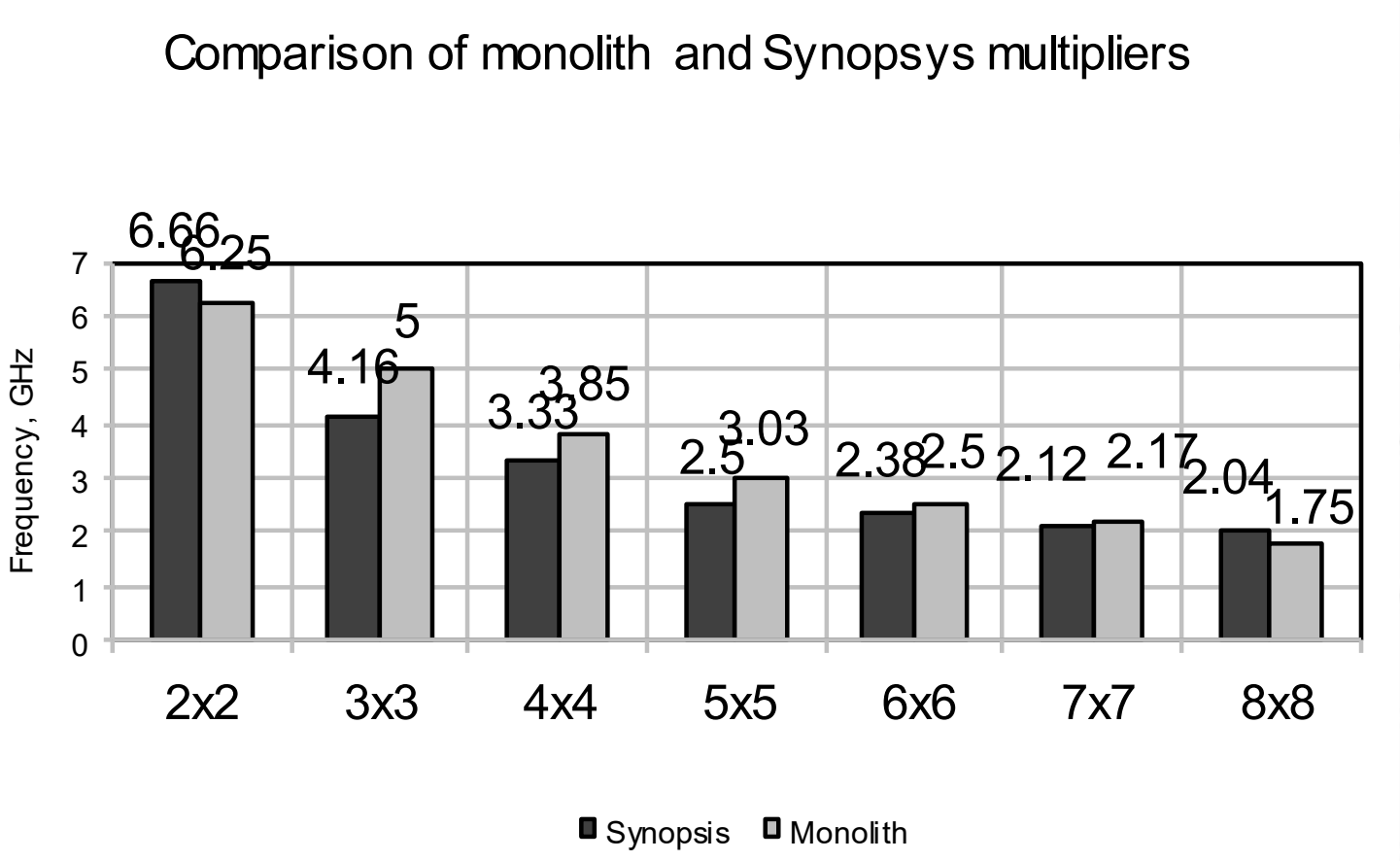


R= 1 2 … m m+1 m+2 … 2m … (k-1)m (k-1)m+1 … km … 2k(m-1) … 2km

There are k^2 of *2m*-bit operands in FTM $R=\sum_{i=1}^{k}\sum_{j=1}^{k} A_i B_j \cdot 2^{m\cdot(i+j-2)}$

It leads to $\lceil \log_2 m\cdot k \rceil$ -levels of adders tree

Multiplication in Synopsys in 28 nm technology



Example. $A \cdot B = R$, where A and B are 14-bits operands.

$$A = (A_4, A_3, A_2, A_1) \qquad B = (B_4, B_3, B_2, B_1)$$

$$A_1 = (a_4, a_3, a_2, a_1) \quad A_2 = (a_8, a_7, a_6, a_5) \qquad B_1 = (b_4, b_3, b_2, b_1) \quad B_2 = (b_8, b_7, b_6, b_5)$$

$$A_3 = (a_{12}, a_{11}, a_{10}, a_9) \quad A_4 = (a_{14}, a_{13}) \qquad B_3 = (b_{12}, b_{11}, b_{10}, b_9) \quad B_4 = (b_{14}, b_{13})$$

Multiplication in **regular** arithmetic, where R is 28 bit vector:

$$\begin{aligned} R = {} & A_1 \cdot B_1 + A_1 \cdot B_2 \cdot 2^4 + A_1 \cdot B_3 \cdot 2^8 + A_1 \cdot B_4 \cdot 2^{12} + \\ & + A_2 \cdot B_1 \cdot 2^4 + A_2 \cdot B_2 \cdot 2^8 + A_2 \cdot B_3 \cdot 2^{12} + A_2 \cdot B_4 \cdot 2^{16} + \\ & + A_3 \cdot B_1 \cdot 2^8 + A_3 \cdot B_2 \cdot 2^{12} + A_3 \cdot B_3 \cdot 2^{16} + A_3 \cdot B_4 \cdot 2^{20} + \\ & + A_4 \cdot B_1 \cdot 2^{12} + A_4 \cdot B_2 \cdot 2^{16} + A_4 \cdot B_3 \cdot 2^{20} + A_4 \cdot B_4 \cdot 2^{24}. \end{aligned}$$

Adder-tree levels reduction

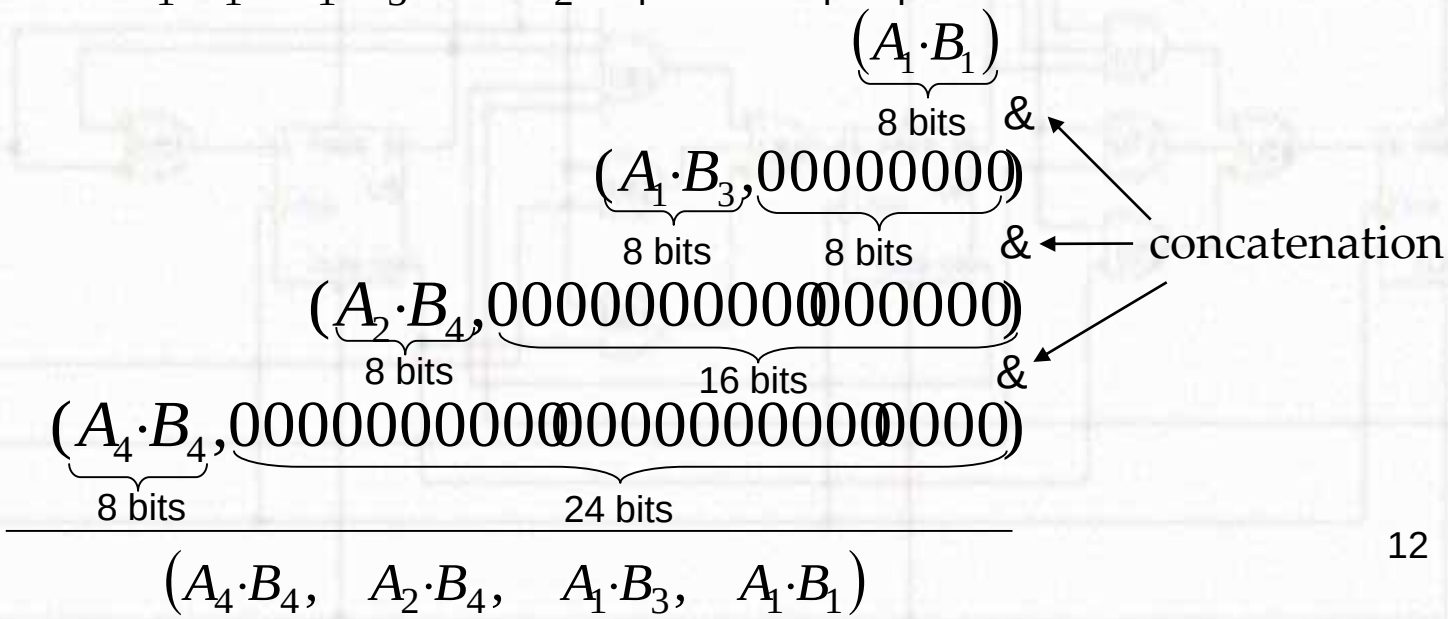
FTM needs $\lceil \log_2 k \cdot m \rceil$ levels in adder-tree, k – number of m-bits sub vectors.

We propose techniques to upgrade architectures of monolith based multipliers and as a consequence to minimize speed of calculation.

The principle of adder-tree optimization concludes in concatenating of detached results of monolith multiplication.

For instance, for $k=4$ and $m=4$, concatenating of the following operands can be joint into one vector:

$$R=\ldots A_1 \cdot B_1 + A_1 \cdot B_3 \cdot 2^8 + A_2 \cdot B_4 \cdot 2^{16} + A_4 \cdot B_4 \cdot 2^{24} \ldots.$$



Efficient architecture

Example. $A \cdot B = R$, where A and B are 14-bits operands and R is 28 bits.

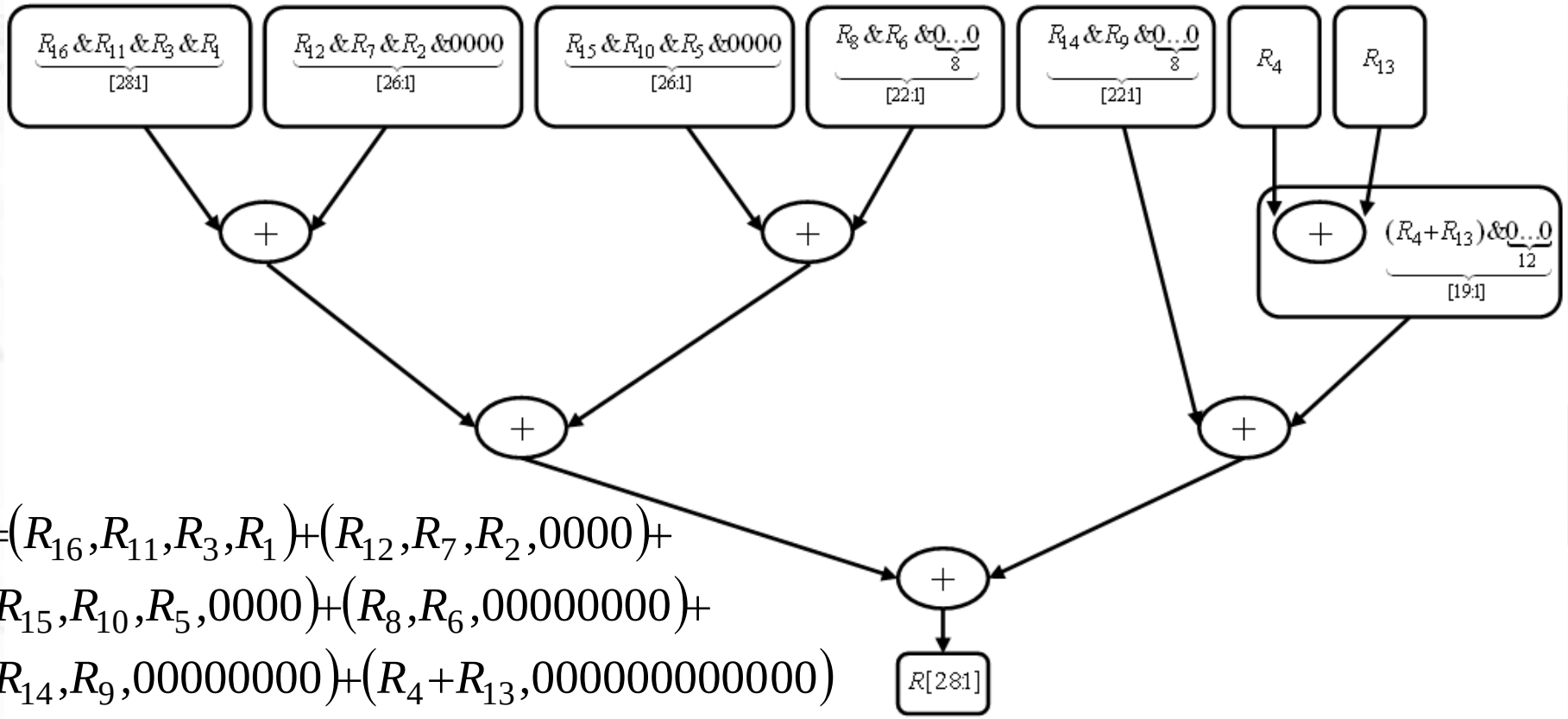
$$A = (A_4, A_3, A_2, A_1) \qquad B = (B_4, B_3, B_2, B_1)$$

$$A_1 \cdot B_1 = R_1 \quad A_2 \cdot B_1 = R_5 \quad A_3 \cdot B_1 = R_9 \quad A_4 \cdot B_1 = R_{13}$$

$$A_1 \cdot B_2 = R_2 \quad A_2 \cdot B_2 = R_6 \quad A_3 \cdot B_2 = R_{10} \quad A_4 \cdot B_2 = R_{14}$$

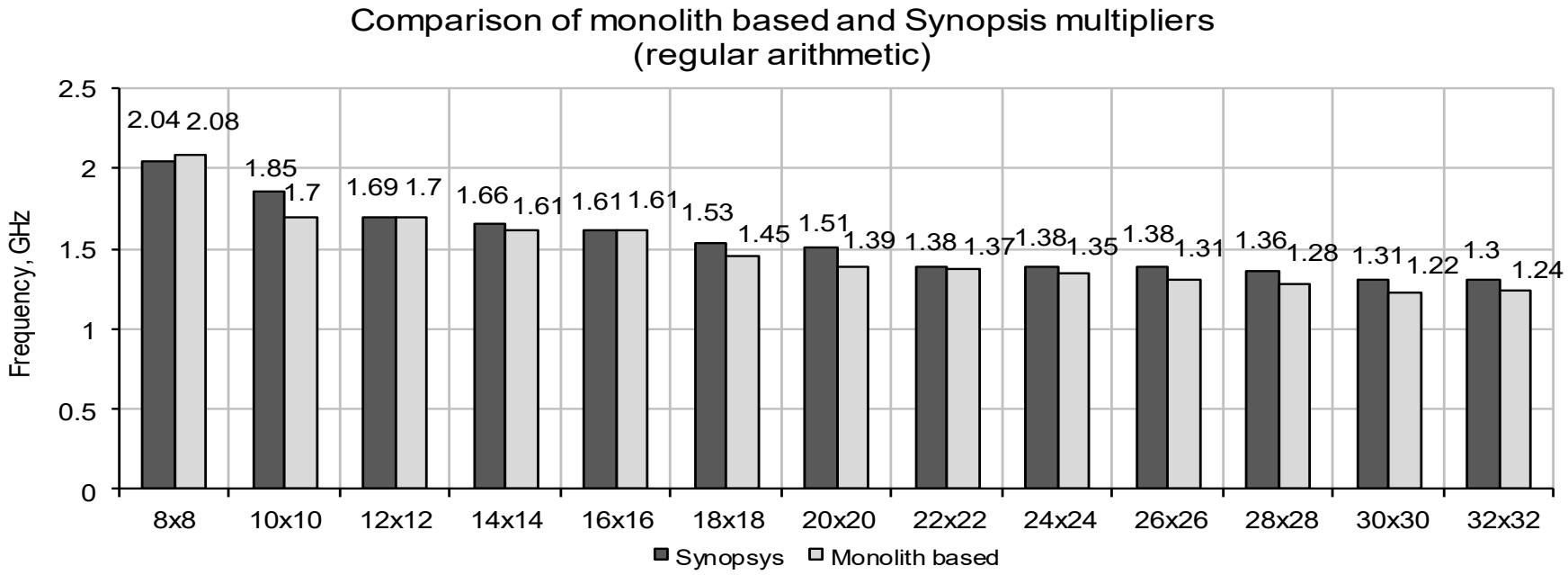
$$A_1 \cdot B_3 = R_3 \quad A_2 \cdot B_3 = R_7 \quad A_3 \cdot B_3 = R_{11} \quad A_4 \cdot B_3 = R_{15}$$

$$A_1 \cdot B_4 = R_4 \quad A_2 \cdot B_4 = R_8 \quad A_3 \cdot B_4 = R_{12} \quad A_4 \cdot B_4 = R_{16}$$



$$R = (R_{16}, R_{11}, R_3, R_1) + (R_{12}, R_7, R_2, 0000) +$$
$$+ (R_{15}, R_{10}, R_5, 0000) + (R_8, R_6, 00000000) +$$
$$+ (R_{14}, R_9, 00000000) + (R_4 + R_{13}, 000000000000)$$

Multiplication in Synopsys in 28 nm technology



Fourier transformation for *X (mod P):*

$$X\ (mod\ P) =$$
$$(x_\delta, \dots, x_1)\ (mod\ P) +$$
$$+2^\delta (mod\ P) \cdot (x_{2\delta}, \dots, x_{\delta+1}) +$$
$$+2^{2\delta} (mod\ P) \cdot (x_{3\delta}, \dots, x_{2\delta+1}) +$$
$$+2^{3\delta} (mod\ P) \cdot (x_{4\delta}, \dots, x_{3\delta+1}) + \dots$$

$$X = (x_{k\delta}, \dots, x_{(k-1)}, \dots, x_1)$$

$$\delta =]\log_2 P[$$

$$9 \cdot X (\text{mod } 13) = R$$

$X = (x_4, x_3, x_2, x_1)$ $R = (r_4, r_3, r_2, r_1)$

$x_4x_3x_2x_1$	$r_4r_3r_2r_1$	$x_4x_3x_2x_1$	$r_4r_3r_2r_1$
0 0 0 1	0 0 0 0	0 1 0 0	1 0 0 0
0 0 0 0	1 0 0 1	1 0 1 -	1 0 0 0
0 0 1 0	0 1 0 1	0 0 0 1	1 0 0 0
0 0 1 1	0 0 0 1	0 1 1 1	1 0 0 0
0 1 0 0	1 0 1 0	0 1 0 1	0 1 0 0
0 1 0 1	0 1 1 0	- 0 1 0	0 1 0 0
0 1 1 0	0 0 1 0	1 - 0 0	0 1 0 0
0 1 1 1	1 0 1 1	1 0 0 -	0 0 1 0
1 0 0 0	0 1 1 1	0 1 - -	0 0 1 0
1 0 0 1	0 0 1 1	- 0 0 1	0 0 0 1
1 0 1 0	1 1 0 0	1 0 0 -	0 0 0 1
1 0 1 1	1 0 0 0	0 - 1 1	0 0 0 1
1 1 0 0	0 1 0 0	0 0 1 -	0 0 0 1

Espresso (Berkeley, USA)

or

ELS (Minsk, Belarus)

- DNF minimization (two-level minimization);

- Binary Decision Diagram (DBB) minimization (multi-level minimization);

- minimization in class of Reed-Muller expansions;

- etc.

Verilog for [100:1] (mod 997)

```
module x_100_mod_997(
input [100:1] X,
output [10:1] R );

wire [22:1] R_temp_1;
wire [15:1] R_temp_2;
wire [11:1] R_temp_3;
reg [10:1]  R_temp;
```

R_temp_1 < 3566179 (22-bit number)

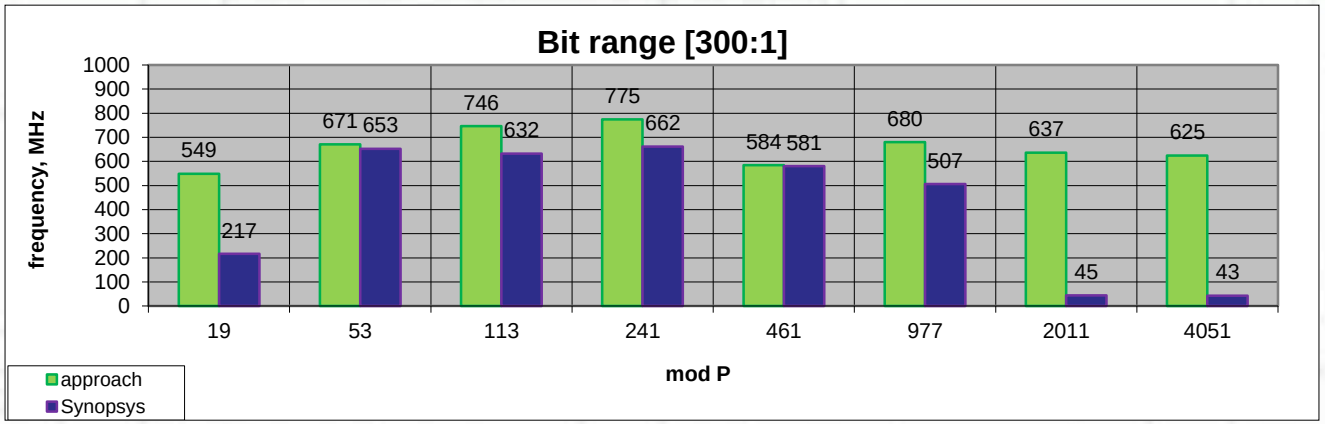
R_temp_2 < 30831 (15-bit number)

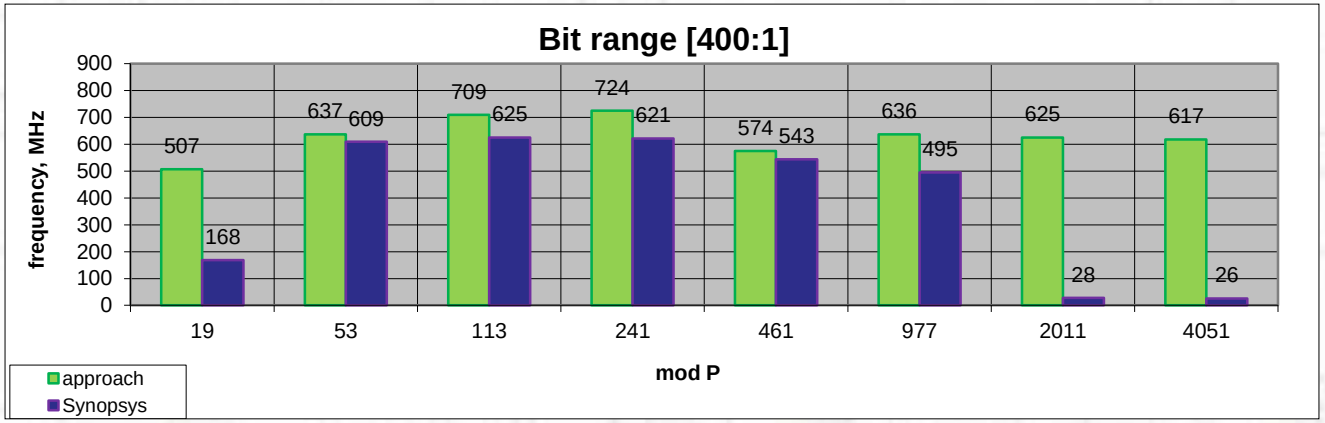
R_temp_3 < 1833 (11-bit number)

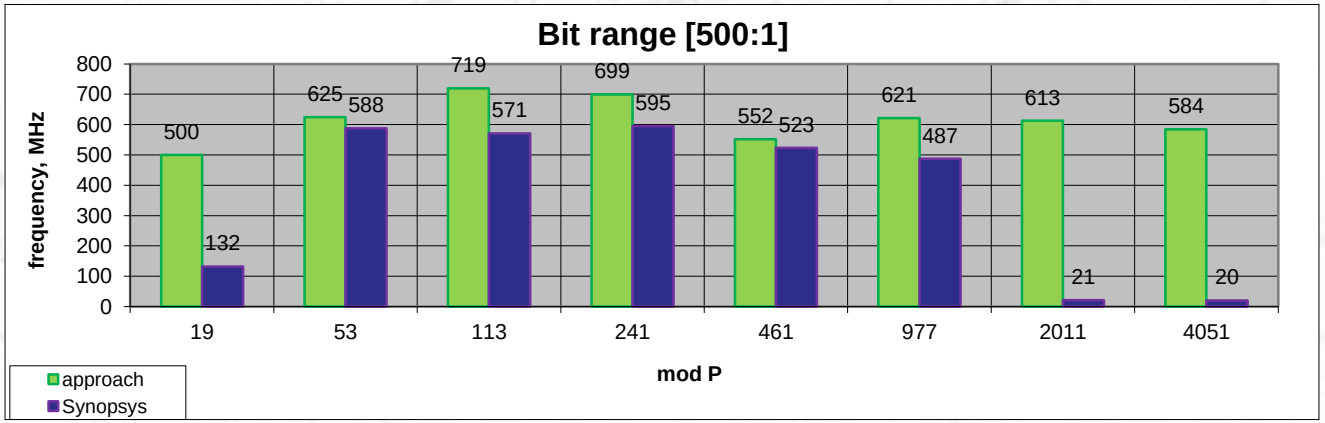
```
assign R_temp_1 = X [ 10 : 1 ]  + X [ 20 : 11 ] * 5'b11011 + X [ 30 : 21 ] * 10'b1011011001 +
X [ 40 : 31 ] * 10'b1011100100 + X [ 50 : 41 ] * 6'b101000 + X [ 60 : 51 ] * 7'b1010011 + X [
70 : 61 ] * 8'b11110111 + X [ 80 : 71 ] * 10'b1010101111 + X [ 90 : 81 ] * 10'b1001011011 + X
[ 100 : 91 ] * 9'b101001001 ;
assign R_temp_2 = R_temp_1 [ 10 : 1 ]  + R_temp_1 [ 20 : 11 ] * 5'b11011 + R_temp_1 [ 22
: 21 ] * 10'b1011011001 ;
assign R_temp_3 = R_temp_2 [ 10 : 1 ]  + R_temp_2 [ 15 : 11 ] * 5'b11011 ;

always @(R_temp_3)
begin  if (R_temp_3 >= 10'b1111100101)
R_temp <= R_temp_3 - 10'b1111100101;
else    R_temp <= R_temp_3;
endassign R = R_temp;
endmodule
```

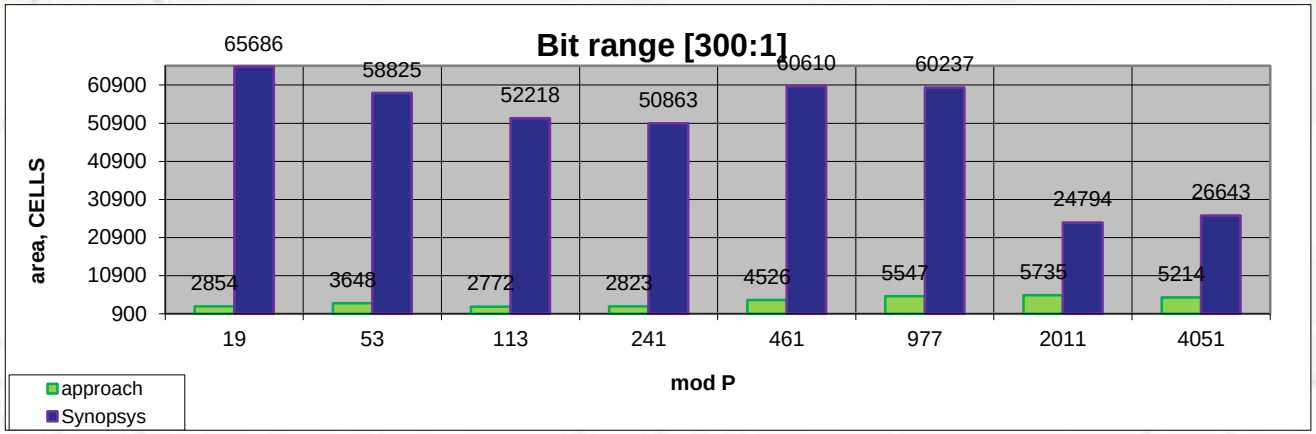
Performance in Synopsys in 28 nm for X (mod P)

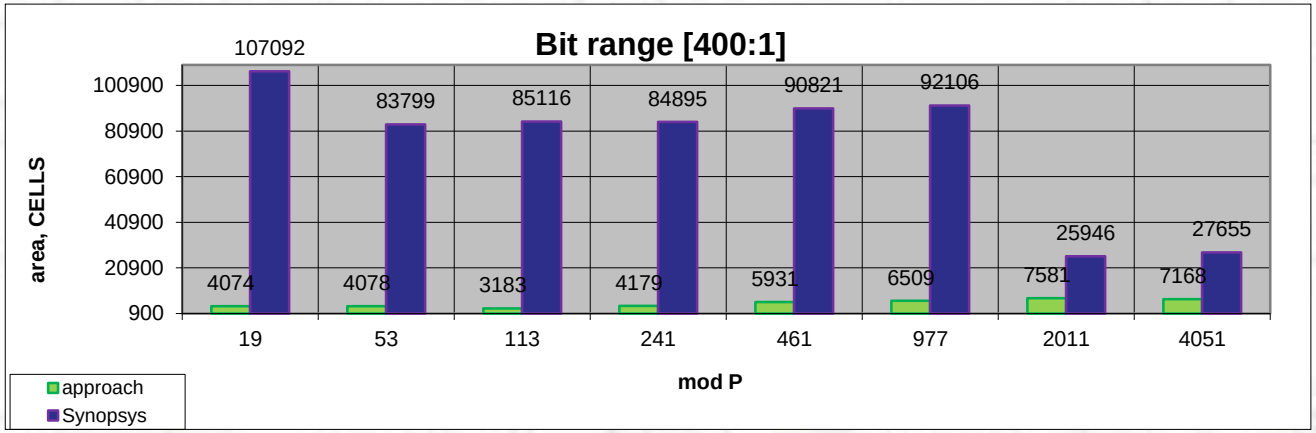


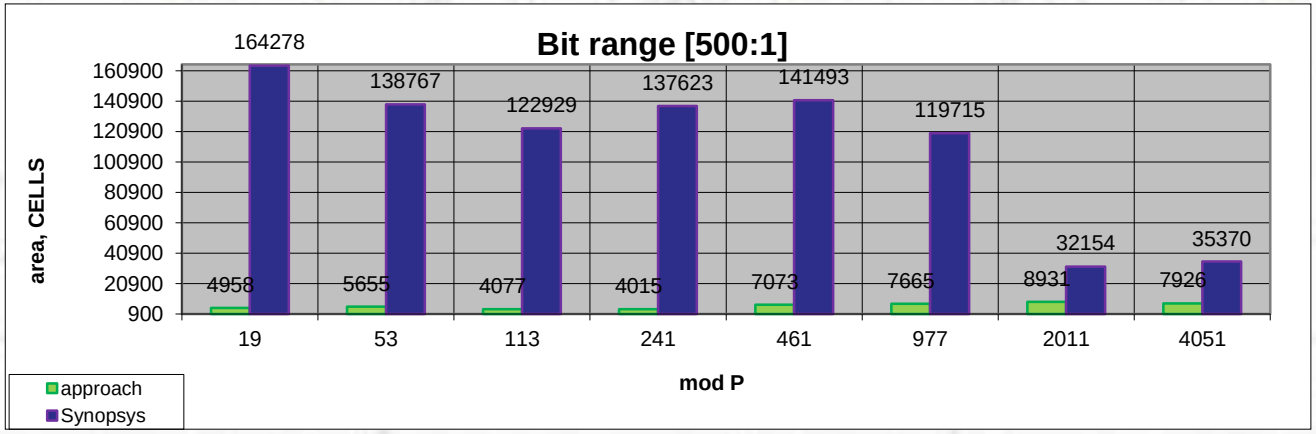




Area in Synopsys in 28 nm technology for X (mod P)







Conclusions

The proposed hardware realization of X (mod P)

- is faster up to 30 times (comparing with Synopsys)

- occupies area up to 34 times smaller (comparing with Synopsys analogues).

An upper bound 500 bit input *X* is the biggest for which Synopsys could synthesis a circuit.

Synthesis of modulus function for 600 bit input *X* was corrupted after nine days of synthesis in Synopsys, but it takes only 20 minutes to synthesize circuits with the proposed technique.

Further research

– “Input splitting” approach implement to reverse conversion (from RNS to positional numbers);

– “Input splitting” approach implement to floating point for half, single, and double precision calculations for RISC-processors;

– create EDA for design RNS units