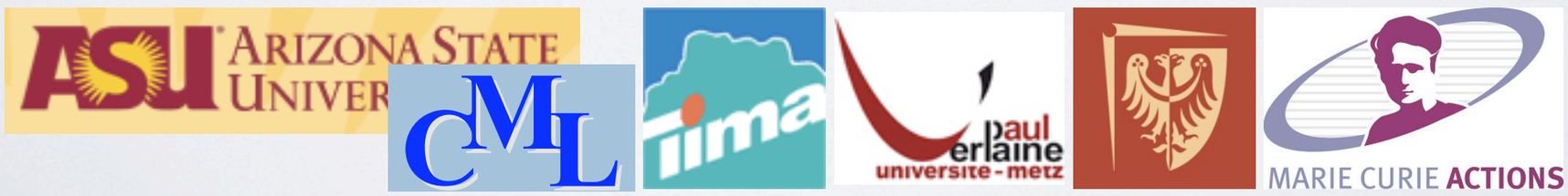
EXPLOITING RESIDUE NUMBER SYSTEM FOR POWER-EFFICIENT DIGITAL SIGNAL PROCESSING IN EMBEDDED PROCESSORS

Rooju Chokshi[1], Krzysztof S. Berezowski[2,3], Aviral Shrivastava[2], Stanisław J. Piestrak[4]

[1]Microsoft Corporation, Redmond, WA, USA
[2]CSE Dept, Arizona State University, Tempe, AZ, USA
[3]TIMA Laboratory, 38031 Grenoble, France
[4]LICM, University of Metz, 57070 Metz, France



FOR THE IMPATIENT

Standard Embedded Processor (ARM)

+

Residue Number System (2^n+1, 2^k, 2^n-1)

+

Compiler-aware Architectural Design
(exposition of RNS advantages and overheads)

+

Compilation Techniques
(appropriate instruction selection and scheduling)

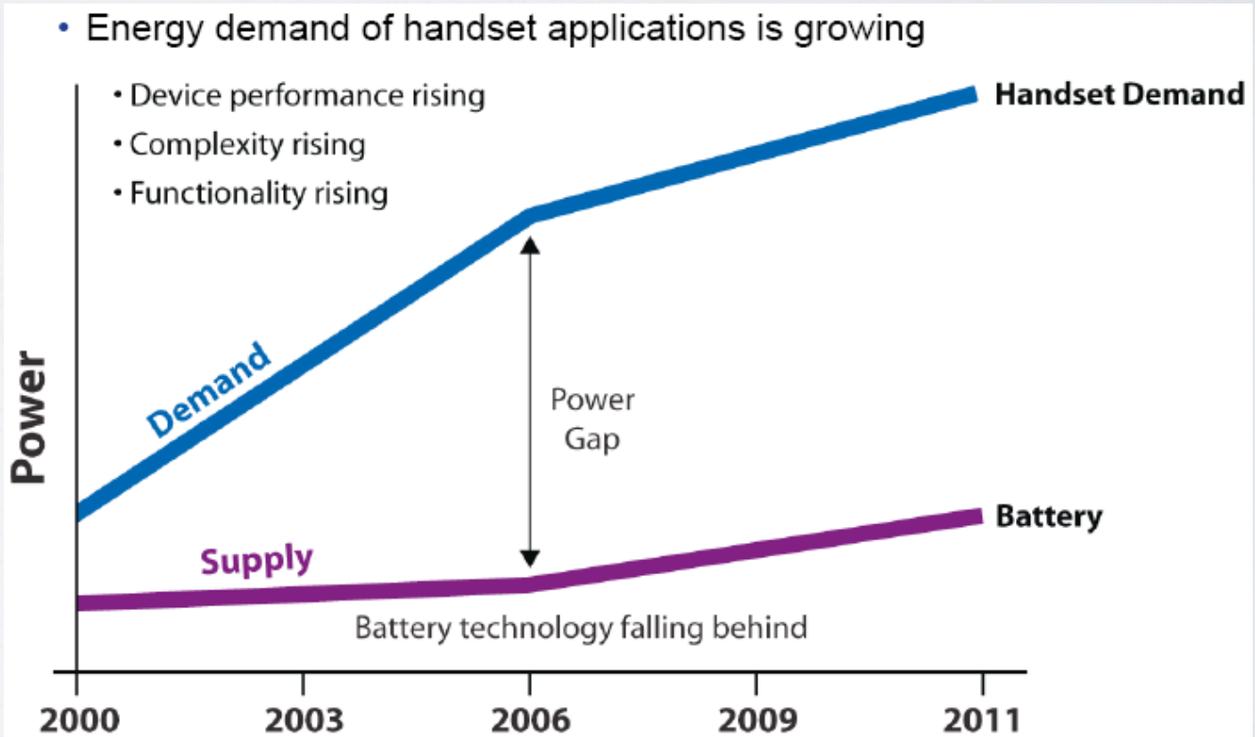
=

Significant improvements in power-efficiency of DSP-intensive embedded applications

MOTIVATION

- digital signal processing (DSP) dominates workload of portable embedded processors (communicat. & multimedia)
- portable devices quickly become amazingly feature-rich
- **power efficiency** is insufficient - battery demands determine volume, shape factor, weight, charging frequency, lifetime...

personal experience:
iPod Touch (fully charged)
allows for ~**45 mins**
of a **Skype conversation**
what about a video call?
and the gap increases...



Source: Battery Technology & Power Management Conference, Vancouver, 18 August, 2005. Stuart Robinson, Strategy Analytics, Ltd.

closing the gap requires **breaking the current trends**

RNS? WHY BOTHER?

non-positional number system where a value

$$X \in [0, M)$$

is represented as a **set** of residues

$$\{x_1, ..., x_l : \forall_{i=1}^{l} x_i = |X|_{P_i}\}$$

modulo respective *co-prime* integers (*moduli*)

$$P = \{P_1, ..., P_l\}$$

as a result $\circ \in \{+, -, \times\}$ on residues **are inherently parallel:**

$$X \circ Y = \{|x_1 \circ y_1|_{P_1}, ..., |x_l \circ y_l|_{P_l}\}$$

where $X = \{x_1, ..., x_l\} \quad Y = \{x_1, ..., x_l\}$

THERE IS NO FREE LUNCH...

- **no** support for **division** nor **magnitude comparison**
- interaction with 2's complement system requires **conversions:**
  - **forward** (computation of residue)
  - **reverse** - Chinese Remainder Theorem (CRT):

$$X = x_1 + P_1 \cdot \left| m_1(x_2 - x_1) + \sum_{i=2}^{l-1} m_i \left(\prod_{j=2}^{i} P_j \right)(x_{i+1} - x_i) \right|_{\prod_{j=2}^{l} P_j}$$

where

$$|m_1 P_1|_{P_2 P_3 \ldots P_l} \equiv 1, \ldots, |m_{l-1} P_1 P_2 \ldots P_{l-1}|_{P_l} \equiv 1$$

hardware (under some assumptions):
both forward and reverse ≡ **multi-operand modulo addition**

QUICK EXAMPLE

$$X = 13, Y = 22 \qquad P_1 = 8, P_2 = 9, P_3 = 7$$

$$x_1 = |13|_8 = 5, \quad x_2 = |13|_9 = 4, \quad x_3 = |13|_7 = 6$$

$$y_1 = |22|_8 = 6, \quad y_2 = |22|_9 = 4, \quad y_3 = |22|_7 = 1$$

$$x_1 + y_1 = |5 + 6|_8 = 3$$
$$x_2 + y_2 = |4 + 4|_9 = 8$$
$$x_3 + y_3 = |6 + 1|_7 = 0$$

$$x_1 \cdot y_1 = |5 \cdot 6|_8 = 6$$
$$x_2 \cdot y_2 = |4 \cdot 4|_9 = 7$$
$$x_3 \cdot y_3 = |6 \cdot 1|_7 = 6$$

$$X = z_1 + P_1 \cdot |m_1(z_2 - z_1) + m_2 P_2(z_3 - z_2)|_{P_2 P_3}$$

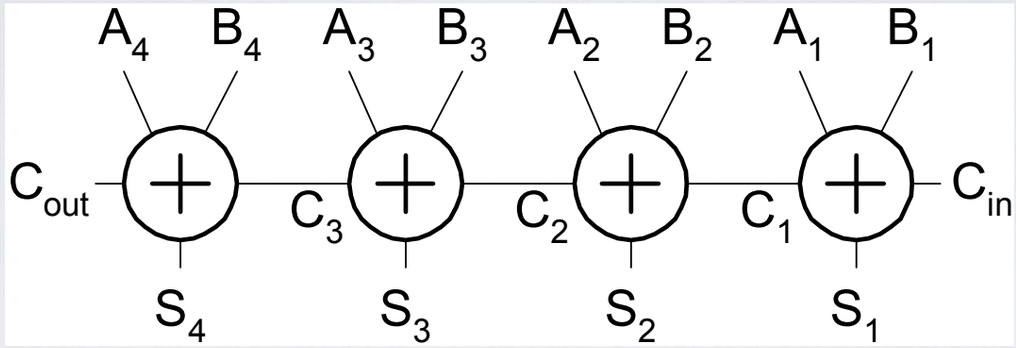
$$m_1 = 8, \quad m_2 = 4$$

$$X + Y = 3 + 8 \cdot |8 \cdot 5 + 4 \cdot 9 \cdot (-8)|_{63} = 3 + 8 \cdot |-248|_{63} = 35$$

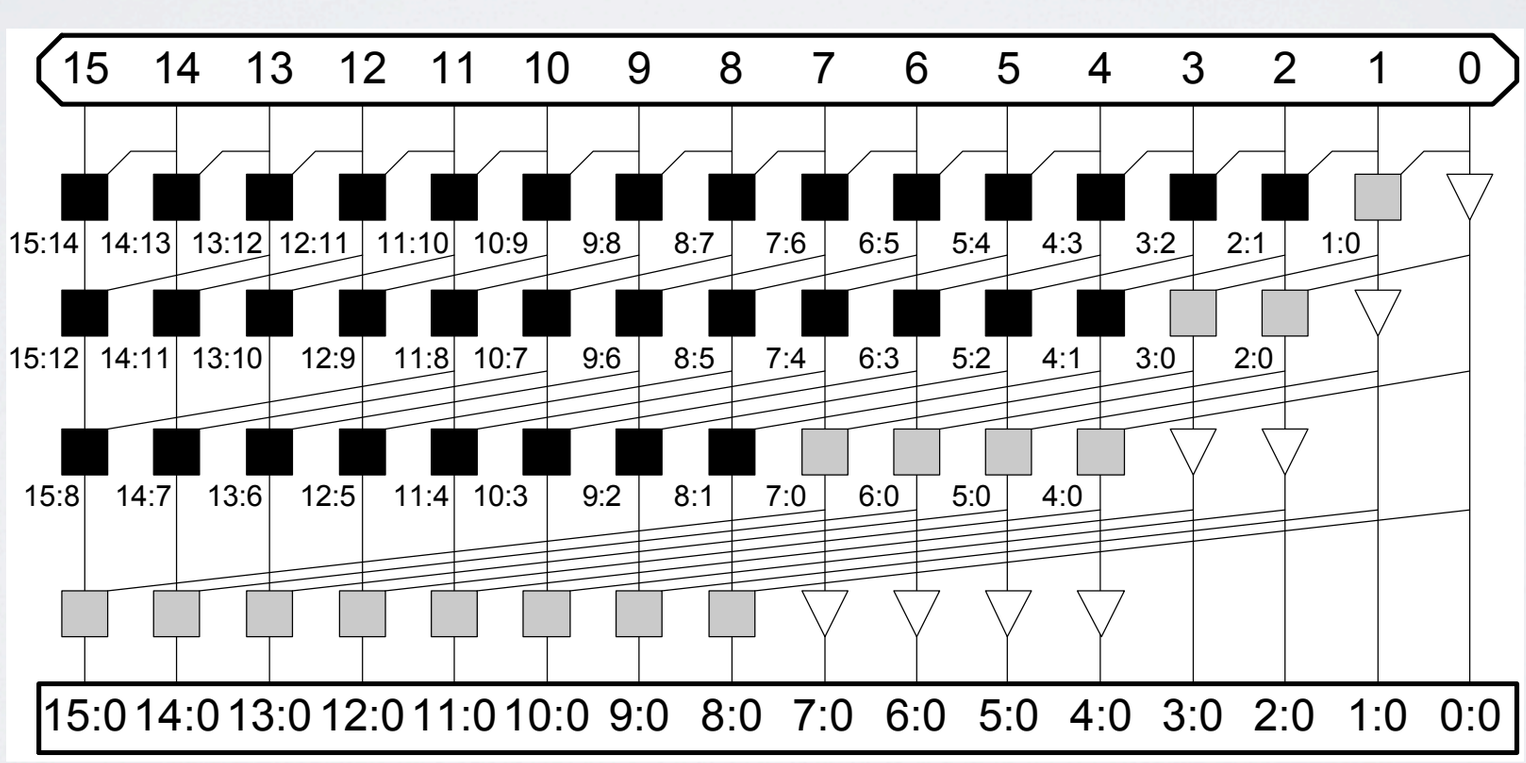
$$X \cdot Y = 6 + 8 \cdot |8 \cdot 1 + 4 \cdot 9 \cdot (-1)|_{63} = 6 + 8 \cdot |-28|_{63} = 286$$

IMPACT ON HARDWARE (+)

carry propagation is either **power-efficient but slow...**



or **fast but hardware intensive.**



RNS partitions the DP - **reduces propagation chain width**

IMPACT ON HARDWARE (*)

Sum of squares is less than square of sum!

				y_5	y_4	y_3	y_2	y_1	y_0		
				x_5	x_4	x_3	x_2	x_1	x_0		
				x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0		
			x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0			
		x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0				
				x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0		
			x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0			
		x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0				
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

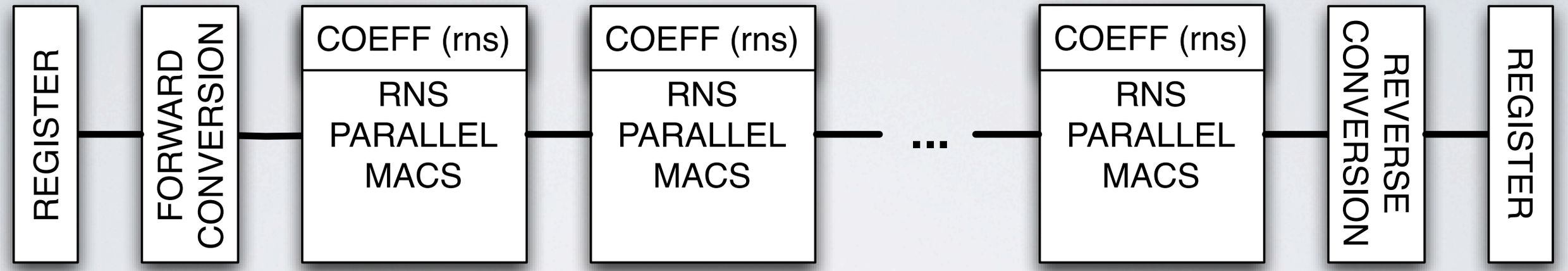
number of partial products:

- 32x32 integer: 1024 PP to reduce
- 4095,4096,4097 **RNS**: 121+66+144 = **331 PP to reduce**!

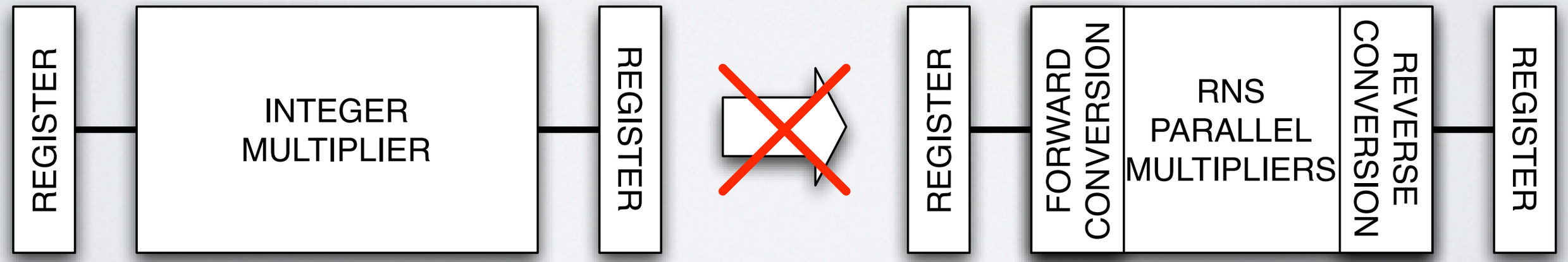
both size AND depth of the carry-save addition (CSA) **reduces**

HOW TO REAP THE BENEFITS?

- **in custom logic** it is easy to **dilute the overheads in benefits**
  a lot of efficient RNS computations happens before conversion

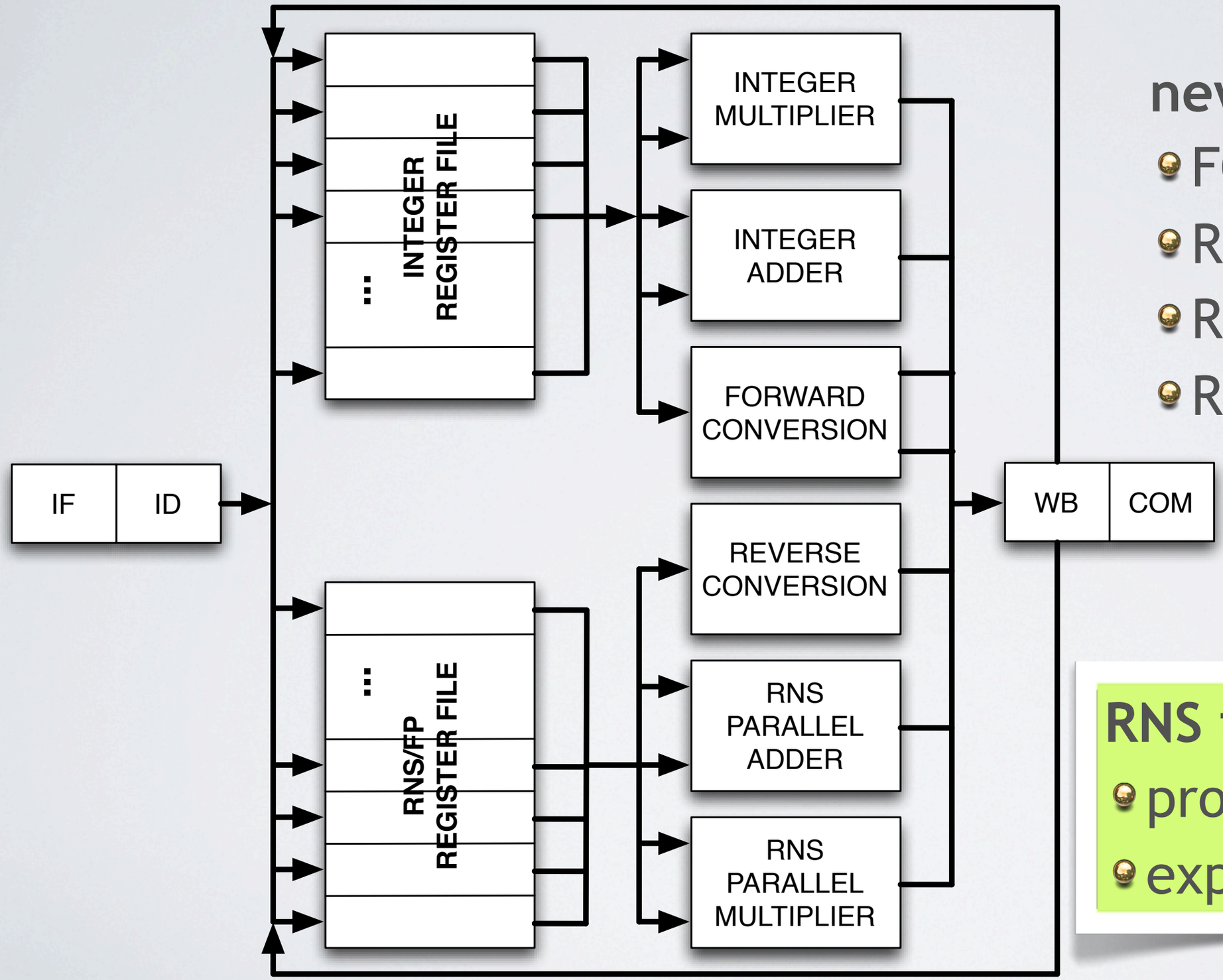


- **in a programmable processor** it is easy to make
  the overheads overshade the benefits...



but the **compiler manages the pipeline**, so let it know!

SYNERGISTIC ARCHITECTURE



new instructions:

- FC
- RC
- RADD
- RMULT

RNS trade-offs:

- promoted to ISA ➜
- exposed to compiler

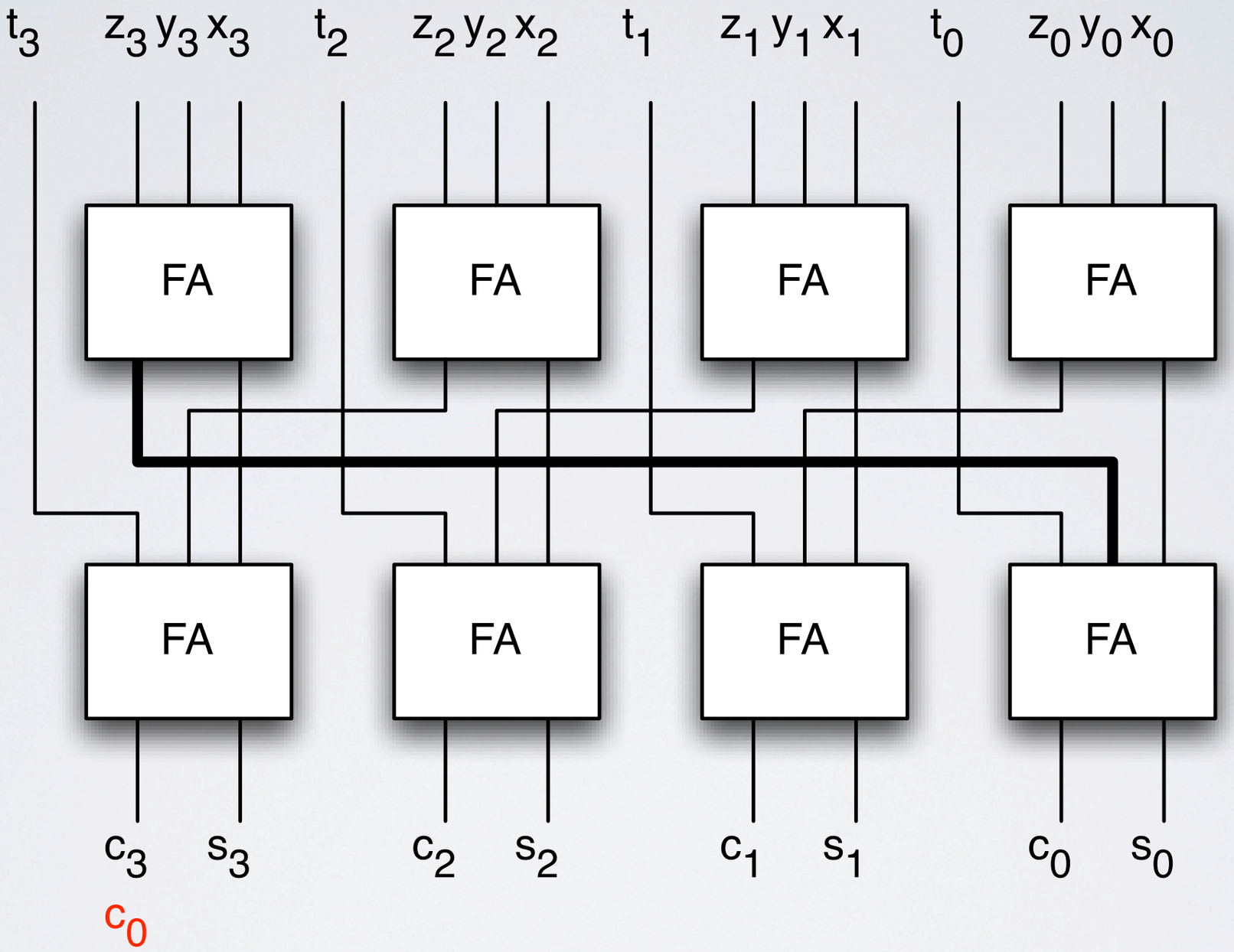
HOW TO DESIGN HARDWARE

- some odd numbers P (moduli) enjoy **periodicity property**, i.e. the series of residues of the $\left|2^i\right|_P$ series is periodic
- for **certain moduli** (certain odd numbers), it happens so that these series of **residues are consecutive powers of 2** (at least their absolute values)
- particularly, the series $\left|2^i\right|_P$ when P is 2^n+1 or 2^n-1 is periodic with respect to *n*:

2^i	2^0	2^1	2^2	2^3	$\dots$	2^{n-1}	2^n	$\dots$	2^{2n-1}	2^{2n}	$\dots$
$\left\|2^i\right\|_{2^n-1}$	2^0	2^1	2^2	2^3	$\dots$	2^{n-1}	2^0	$\dots$	2^{n-1}	2^0	$\dots$
$\left\|2^i\right\|_{2^n+1}$	2^0	2^1	2^2	2^3	$\dots$	2^{n-1}	-2^0	$\dots$	-2^{n-1}	2^0	$\dots$

promotes the usage of **n-bit wide end-around-carry CSA** for the implementation of most of the RNS hardware

END-AROUND CARRY CSA



since: $-\sum_{i=0}^{k} 2^i b_i = \sum_{i=0}^{k} 2^i \bar{b}_i - \sum_{i=0}^{k} 2^i$ for neg. weights invert & correct!

PERIODICITY IMPLICATIONS

essentially EAC-CSA produces a two vectors C and S such that to compute a residue in the channel we need to compute

$$r = |2C + S|_{\mathsf{P}}$$

reducing 2C+S to r is costly - takes 2-operand modulo adders (usually some parallel-prefix adder with EAC embedded)

obviously, by definition, *2C+S* is congruent to *r*.
we can compute on congruences as easily as on residues
and due to EAC in periodic moduli **DP-width never grows**!

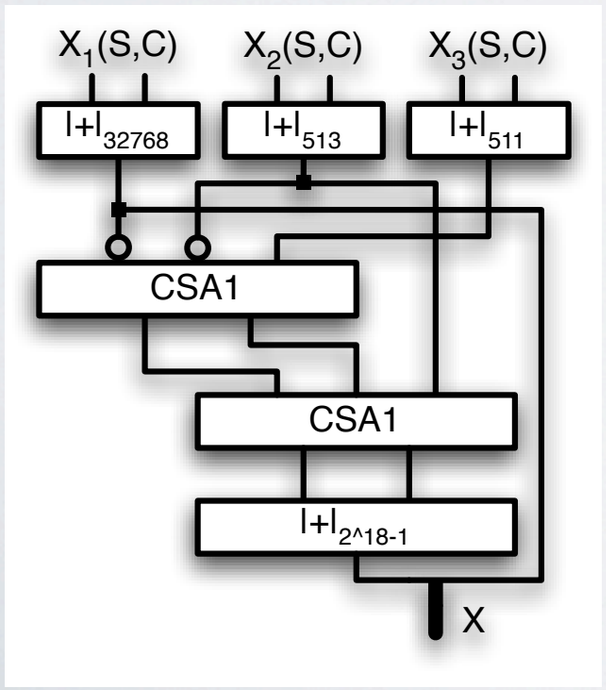
use pair (S,C) for internal representation
work with congruences using EAC-CSA trees

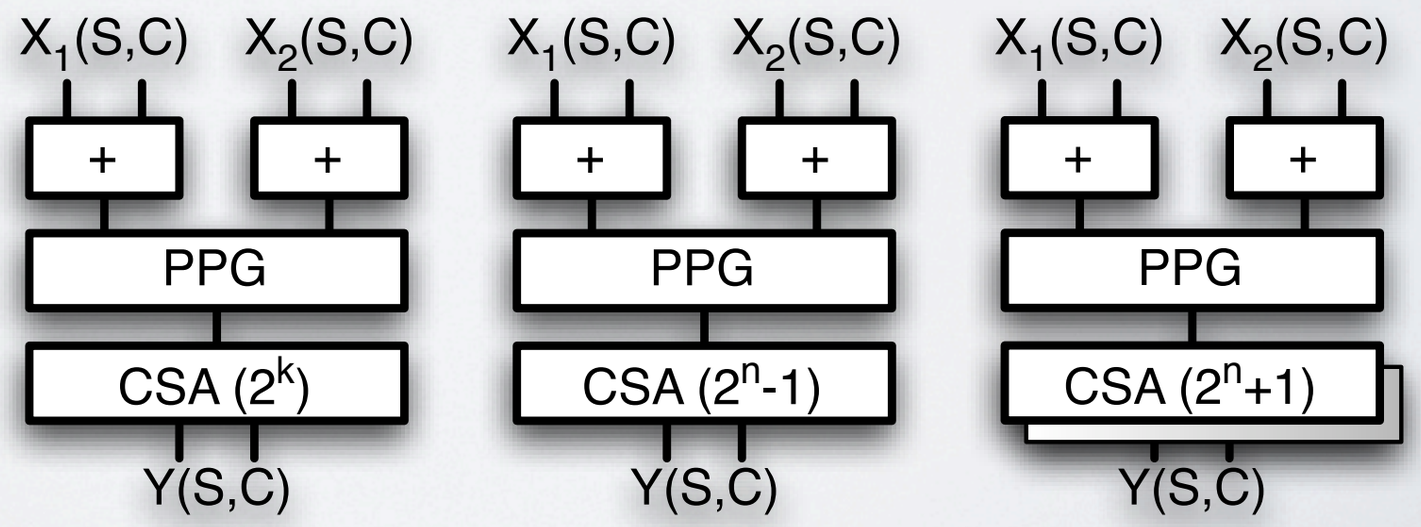
IMPLICATIONS TO HARDWARE

- adders/forward converters (FC) are just standard CSA-EAC
  - 2 layers for two-operand ADD or single forward converter
  - **4 layers** for **three-operand ADD** or **double forward converter**

depth of EAC-CSA is independent of DP-path width

- in multiplier and reverse converter initial reduction is done but again (in the multiplier) we can work with congruences





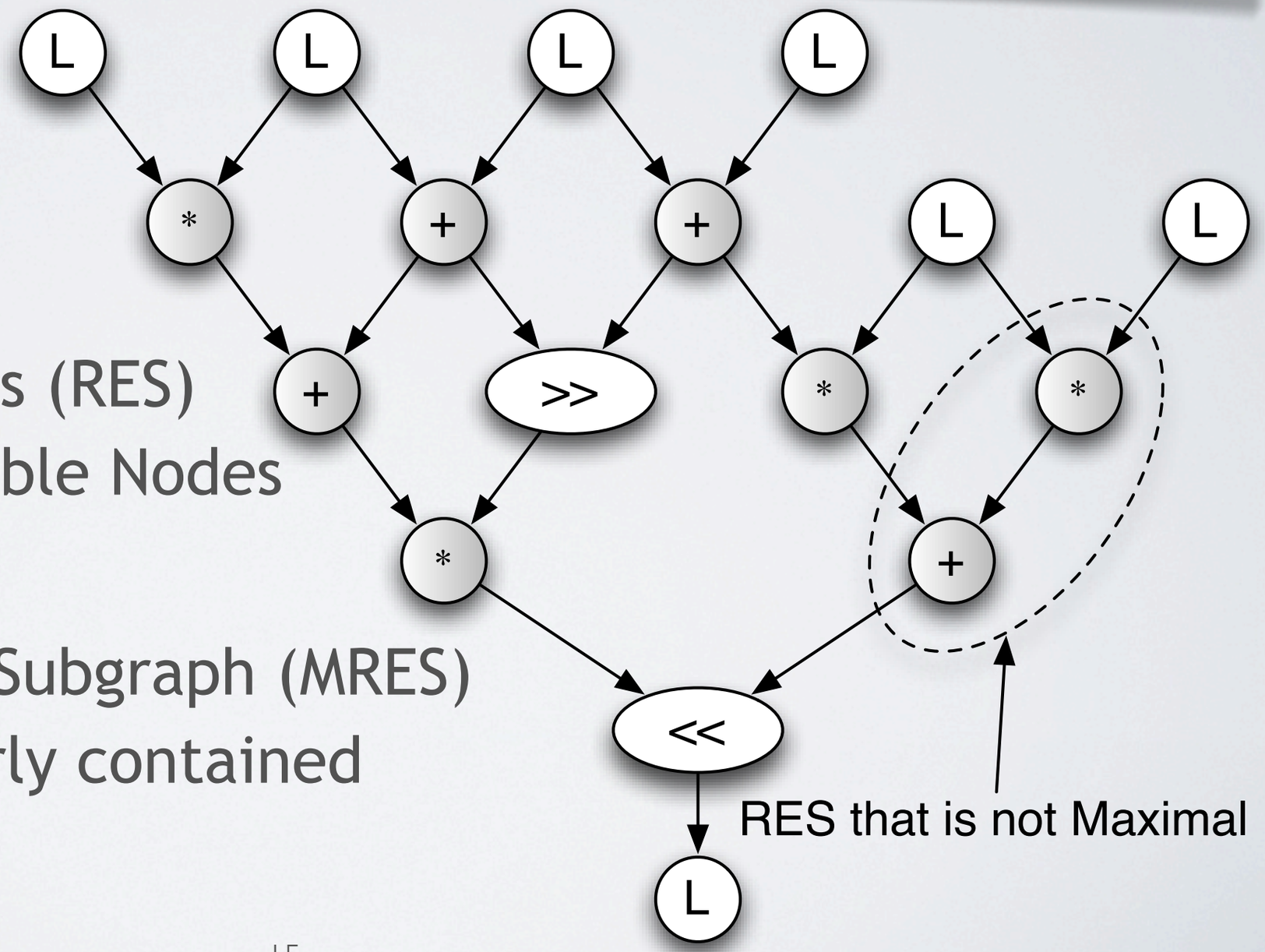
COMPILATION

discover **subgraphs of Data-Flow Graph (DFG)**
that can be **profitably mapped to RNS operators**

RNS Eligible Nodes
(+, -, *)

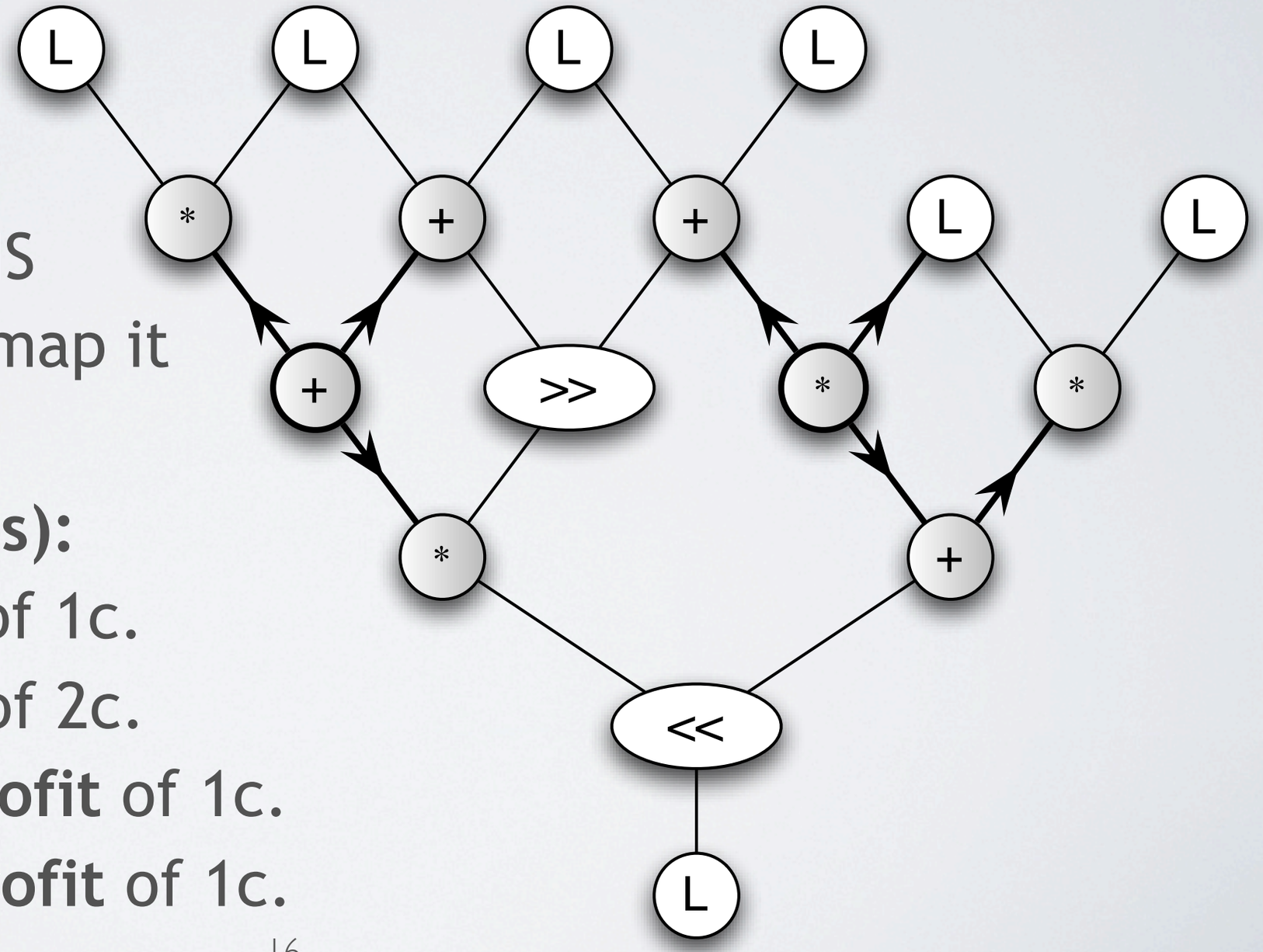
RNS Eligible Subgraphs (RES)
contain only RNS Eligible Nodes

Maximal RNS Eligible Subgraph (MRES)
RES that is not properly contained
in any other RES



FINDING MRES

starting from any *RNS Eligible Node* expand on undirected graph through **breadth-first search** till no more nodes can be included

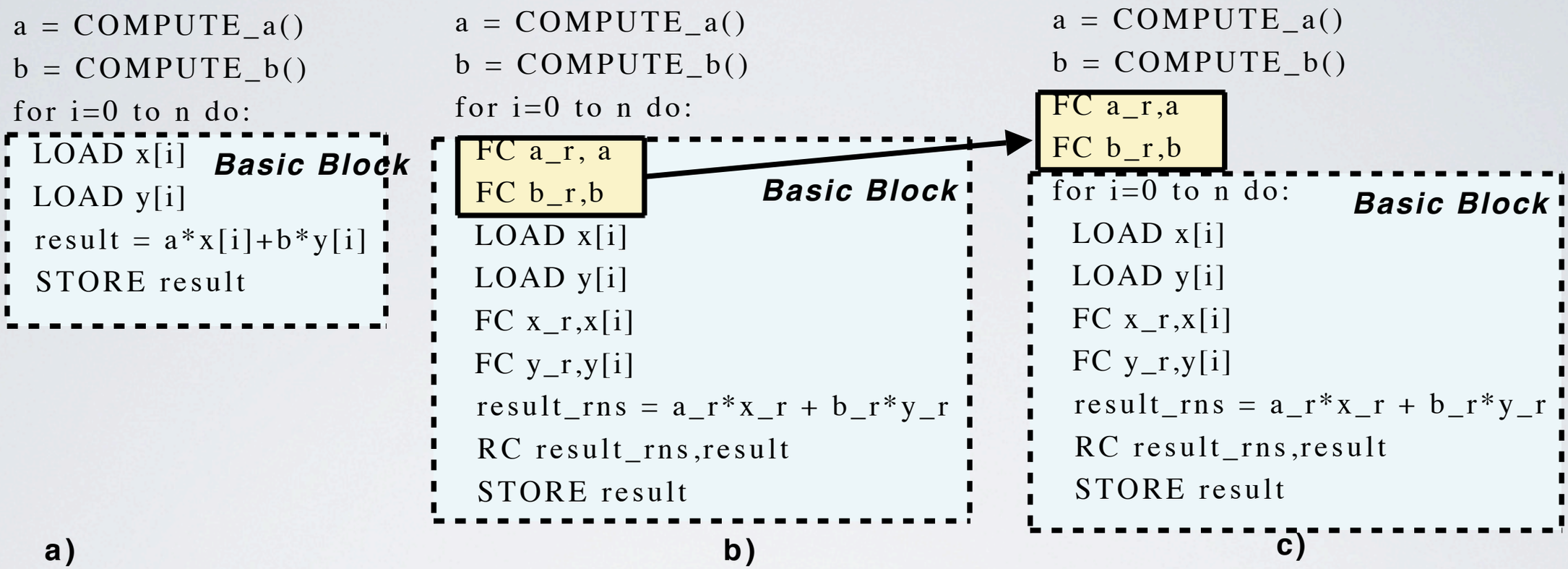


If mapping MRES to RNS turns out profitable - map it

Profit measure (cycles):

- 2 **FC** - an **overhead** of 1c.
- 1 **RC** - an **overhead** of 2c.
- 3-operand **RADD** - **profit** of 1c.
- 2-operand **RMUL** - **profit** of 1c.

SCHEDULING FC IN LOOPS



without scheduling

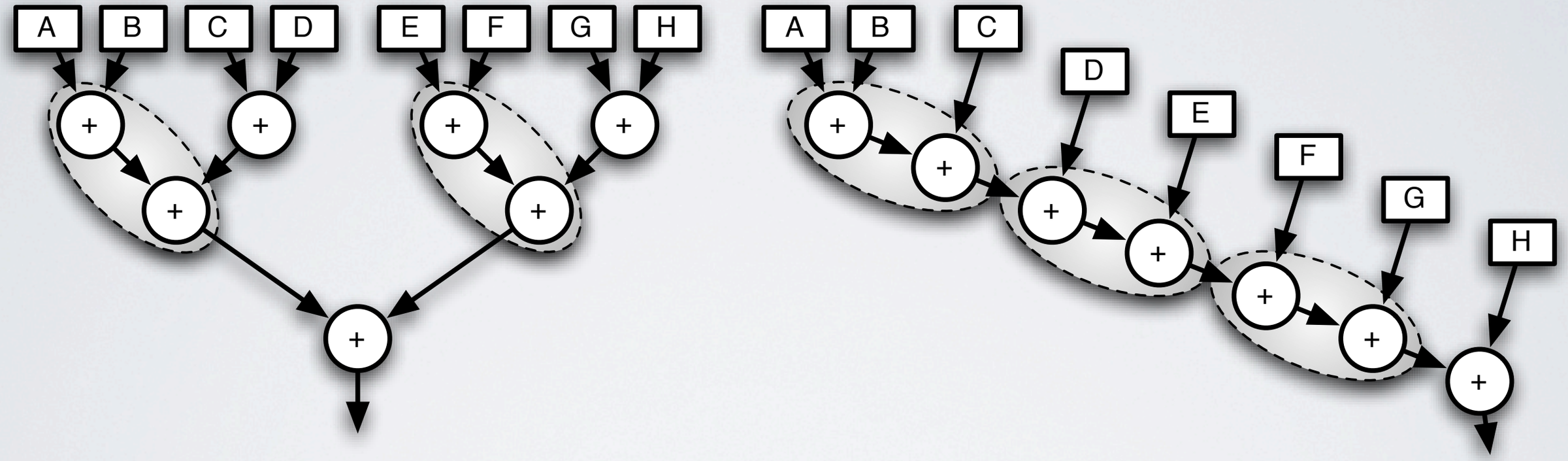
with scheduling

Move FC out of the loop only if:

- register is not being written in the loop
- register is being written only in the same MRES as the FC

ADDITION PAIRING

fast RNS hardware allowed us to implement **3-operand addition**

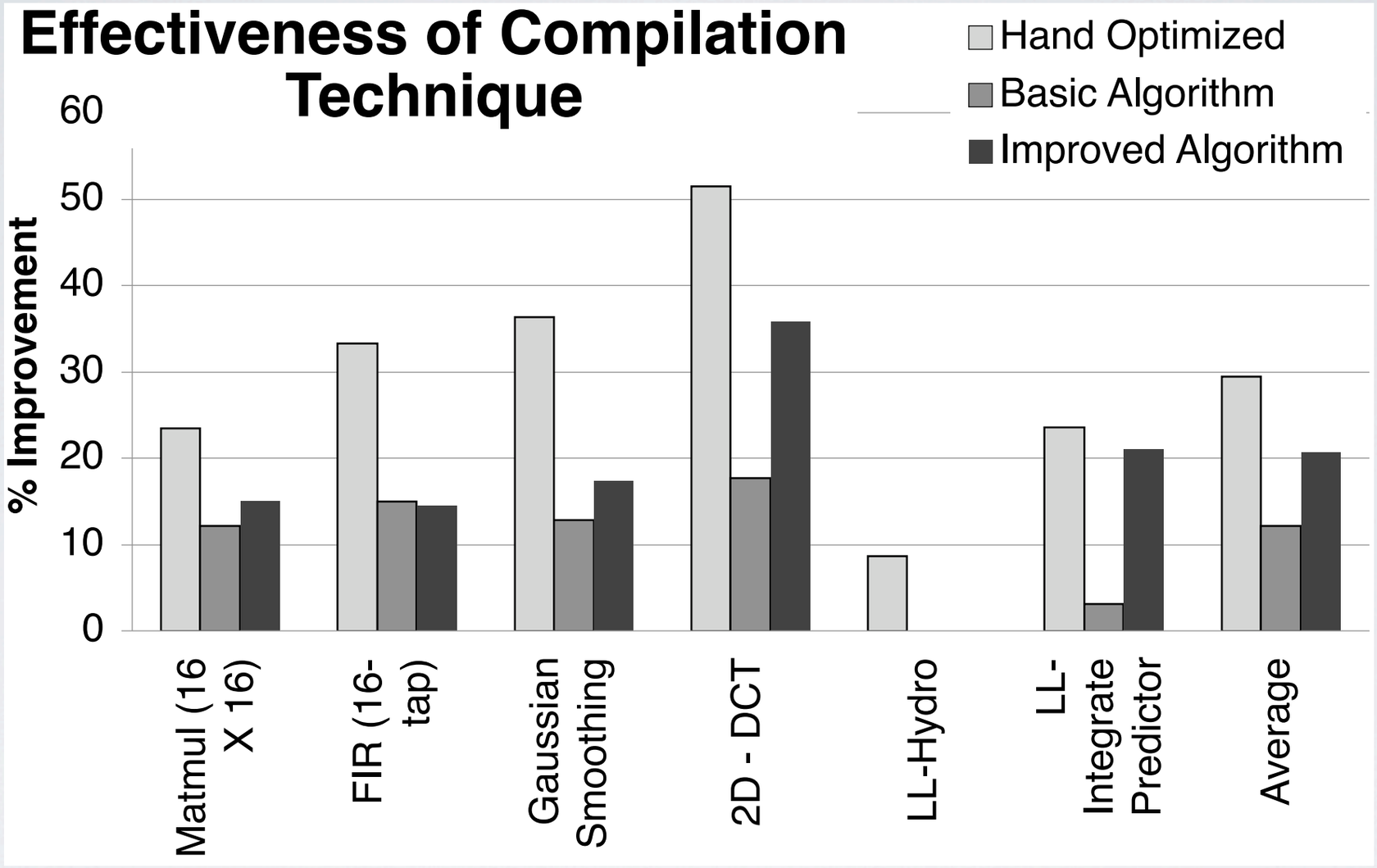


in general case ternary tree of additions should be built
for a single set of RNS resources linearizing is enough

EXPERIMENTAL SETUP

- hardware components were designed in RTL Verilog synthesized using Cadence RC Compiler and OSU 0.18um lib
- SimpleScalar ARM was extended to support RNS instructions RNS values are stored in floating point registers
- GCC was extended to implement the ISA extension and the compilation techniques
- a collection of DSP and image processing kernels were run:
  - without extension
  - with extension - hand-optimized code
  - with extension - basic technique (instruction selection only)
  - with extension - extended technique (selection & scheduling)
  - performance (no. cycles) and power numbers were collected

RESULTS (PERFORMANCE)

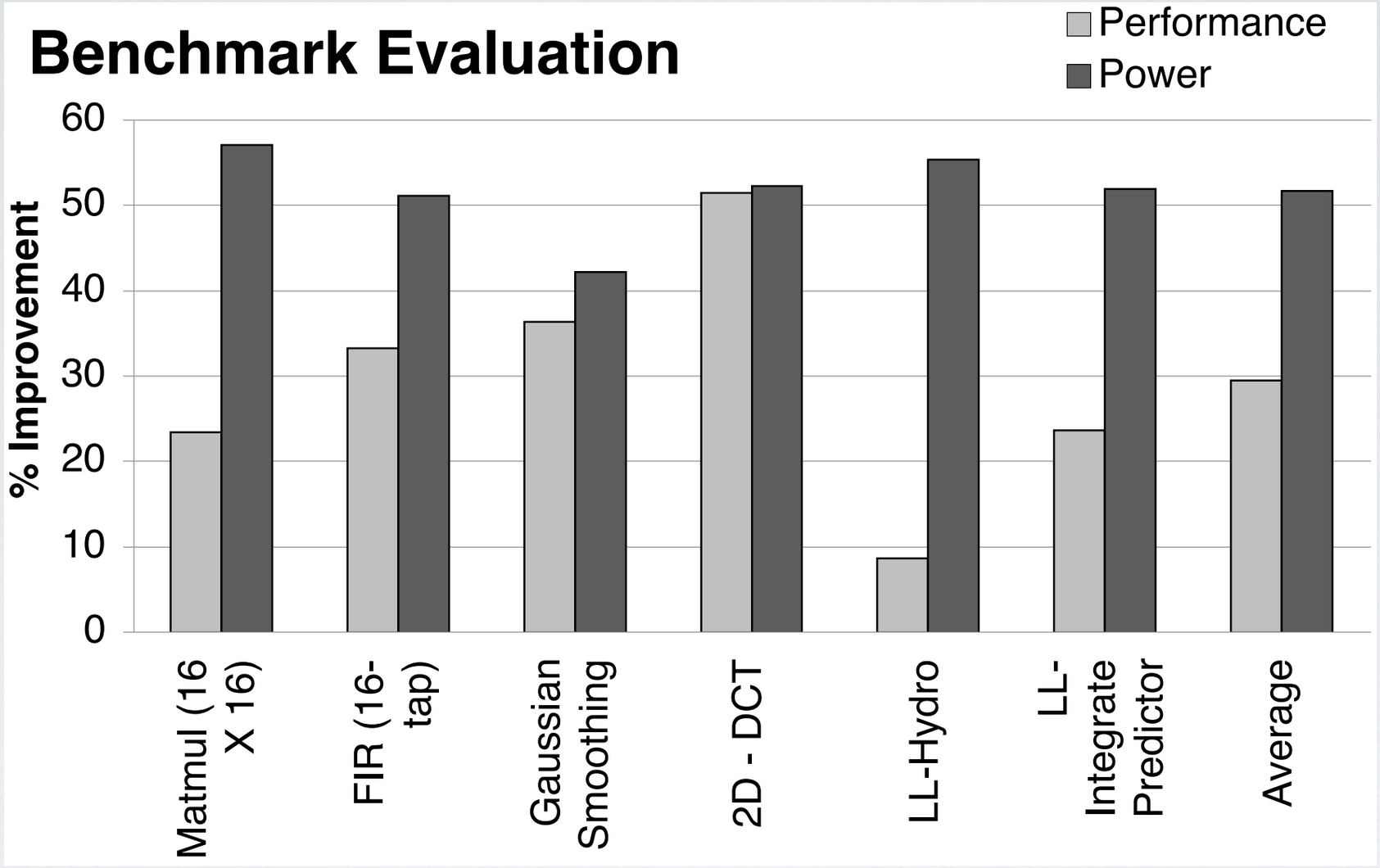


instruction selection: +12% performance on average

selection & scheduling: +20% performance on average

manual optimizations: +30% performance on average

RESULTS (HAND-OPT)

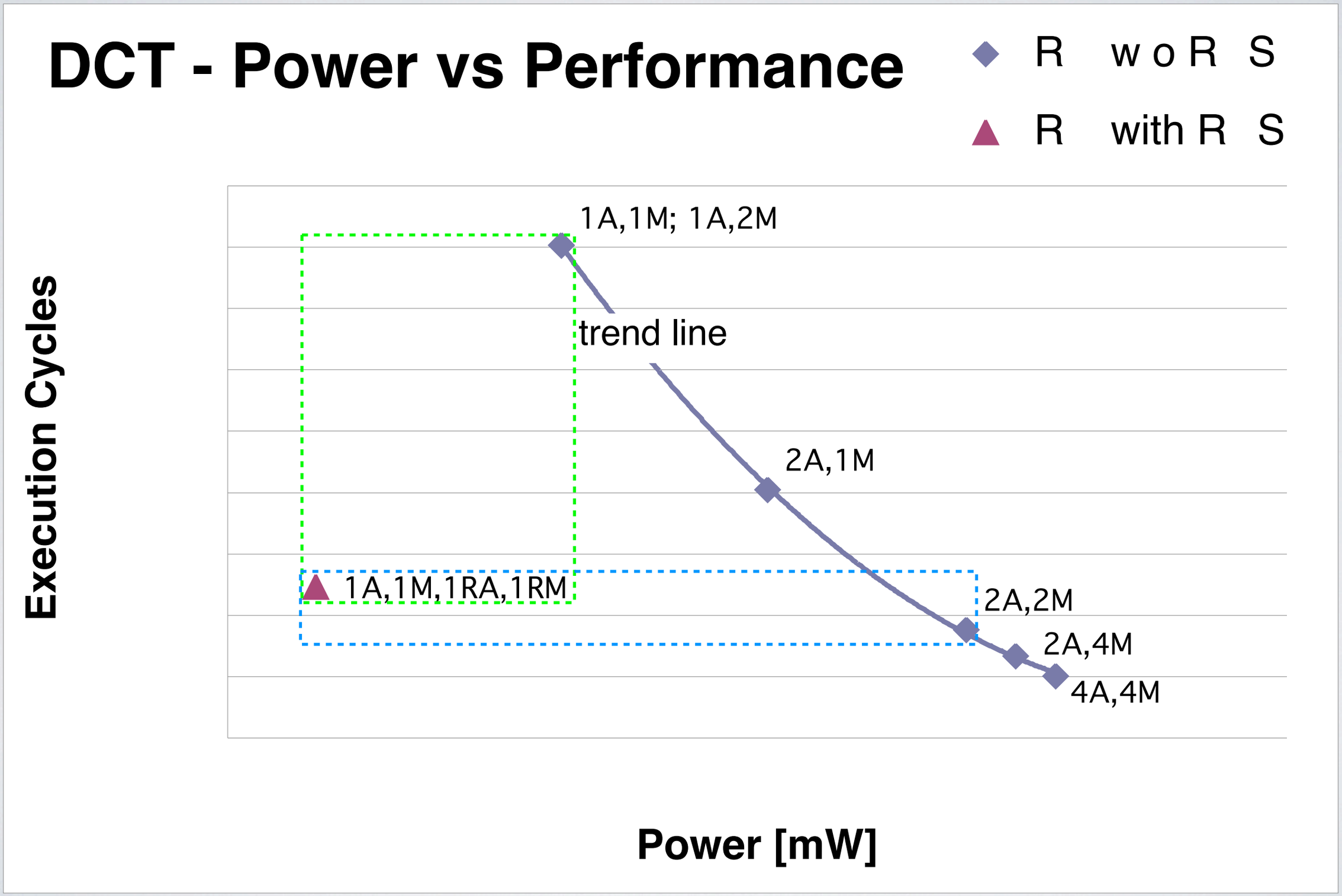


hand-optimized code:

+30% average performance

-57% power on average

DESIGN SPACE EXPLORATION



CONCLUSIONS

- promoting RNS idiosyncrasies to the ISA level
  is an efficient way to use RNS in the programmable CPU
  ➜ effectively creates custom RNS pipeline using software
- Compiler was able to extract as much as
  - **21% improvement in performance** (30% hand-opt), and
  - **52% improvement in power dissipation** (57% hand-opt)
- if you plan to add **second resource set** to your embedded CPU
  consider RNS over regular Integer set.
- if you plan to use only single resource set...
  think again ➜ add RNS. **saves power and boosts performance**

interestingly, unlike in application-specific hardware
in a CPU its **forward conversions that define the bottleneck**

FUTURE WORK

- more aggressive ISA optimizations:
  - integrating loads/stores and conversions
  - support for arithmetic shifts
  - sign detection for magnitude comparison
  - moving conversions into memory interface?
- compilation techniques for super- and hyper-block level
- code annotation for direct interaction between programmer and compiler

EXPLOITING RESIDUE NUMBER SYSTEM FOR POWER-EFFICIENT DIGITAL SIGNAL PROCESSING IN EMBEDDED PROCESSORS

Thank you for your attention!

Rooju Chokshi[1], Krzysztof S. Berezowski[2,3], Aviral Shrivastava[2], Stanisław J. Piestrak[4]

[1]Microsoft Corporation, Redmond, WA, USA
[2]CSE Dept, Arizona State University, Tempe, AZ, USA
[3]TIMA Laboratory, 38031 Grenoble, France
[4]LICM, University of Metz, 57070 Metz, France