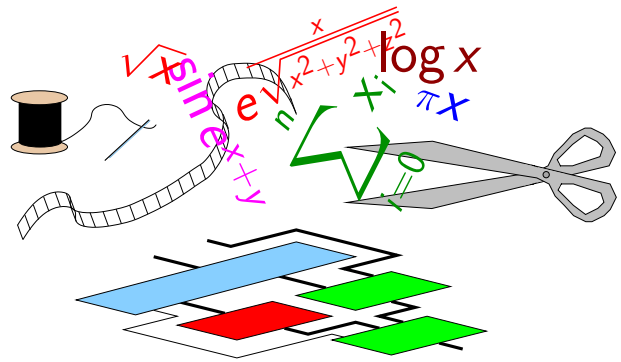
Computing just right: Application-specific arithmetic

Florent de Dinechin



citi lab

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Inria informatiques mathématiques

UNIVERSITÉ DE LYON

Outline

Introduction and motivation

Dark silicon

In current tech, you can no longer
use 100% of the transistors 100% of the time
without destroying your chip.

"Dark silicon" is the percentage that must be off at a given time
... expected to represent 50% of the area of an high-end processor around 2020

Dark silicon

In current tech, you can no longer
use 100% of the transistors 100% of the time
without destroying your chip.

"Dark silicon" is the percentage that must be off at a given time
... expected to represent 50% of the area of an high-end processor around 2020

One way out the dark silicon apocalypse (M.B. Taylor, 2012)

Hardware implementations of rare (but useful) operations:

- when used, dramatically reduce the energy per operation (compared to a software implementation that would take many more cycles)
- when unused, serve as radiator for the used parts

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).
- *Should a processor include elementary functions ?*
  Yes (Paul&Wilson, 1976), No since the transition to RISC

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).
- *Should a processor include elementary functions ?*
  Yes (Paul&Wilson, 1976), No since the transition to RISC
  Yes again soon as useful dark silicon? (e.g. SpiNNaker-2)

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).
- *Should a processor include elementary functions ?*
  Yes (Paul&Wilson, 1976), No since the transition to RISC
  Yes again soon as useful dark silicon? (e.g. SpiNNaker-2)
- *Should a processor include decimal hardware?*
  Yes say IBM, No say Intel
  (motivation is weak IMHO, but if it is useful dark silicon...)

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).
- *Should a processor include elementary functions ?*
  Yes (Paul&Wilson, 1976), No since the transition to RISC
  Yes again soon as useful dark silicon? (e.g. SpiNNaker-2)
- *Should a processor include decimal hardware?*
  Yes say IBM, No say Intel
  (motivation is weak IMHO, but if it is useful dark silicon...)
- *Should a processor include a multiplier by* log(2)*?*
  No of course.

Useful operators that make sense in a processor

(define "rare but useful")

- *Should a processor include a divider and square root?*
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA-based FPUs (first IBM then HP then Intel)
  Then Yes again these days, because it reduces energy (D. Lutz).
- *Should a processor include elementary functions ?*
  Yes (Paul&Wilson, 1976), No since the transition to RISC
  Yes again soon as useful dark silicon? (e.g. SpiNNaker-2)
- *Should a processor include decimal hardware?*
  Yes say IBM, No say Intel
  (motivation is weak IMHO, but if it is useful dark silicon...)
- *Should a processor include a multiplier by* log(2)*?*
  No of course.
  ... except as part of a (more useful) exponential operator

Specific arithmetic hidden in coarser dark silicon functions

(fingerprint recognition, AI accelerators, etc.)

Second motivation: computing with FPGAs

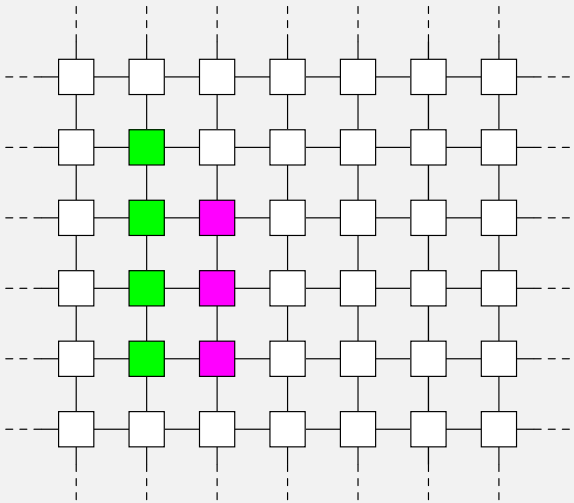
Former title of this presentation was:

The arithmetic operators you will never see in a microprocessor (and how to build them)

But I had to change it...

Basic FPGA structure

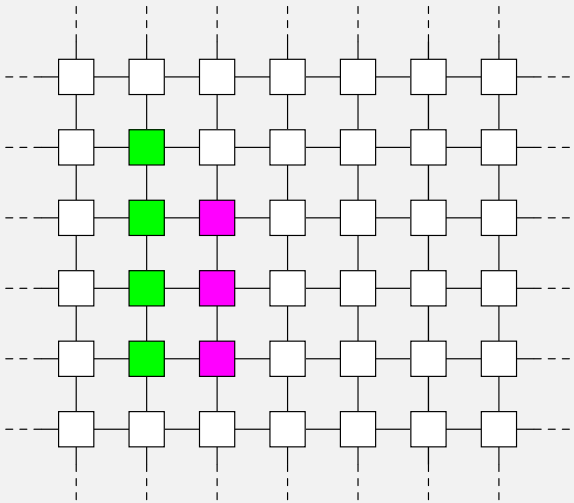
Overal view



- Cell: configurable logic blocks
  - ... a small configurable automaton

Basic FPGA structure

Overal view

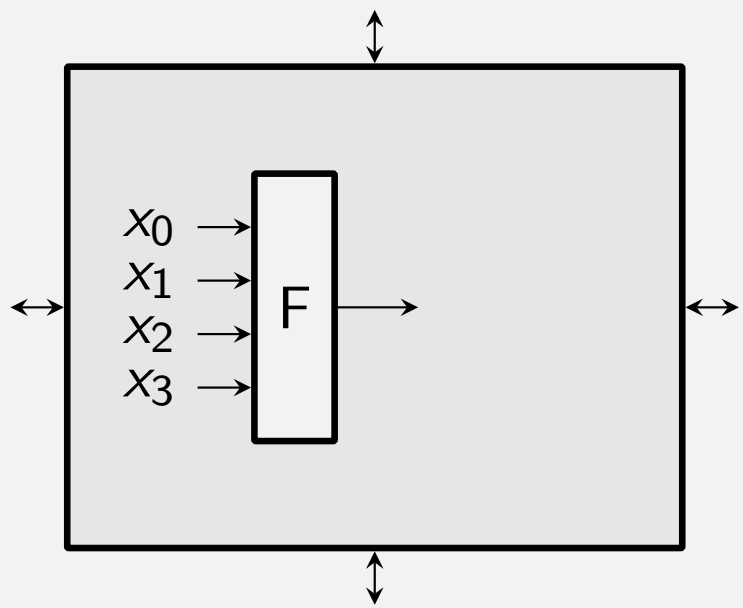


- Cell: configurable logic blocks
  - ... a small configurable automaton

- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

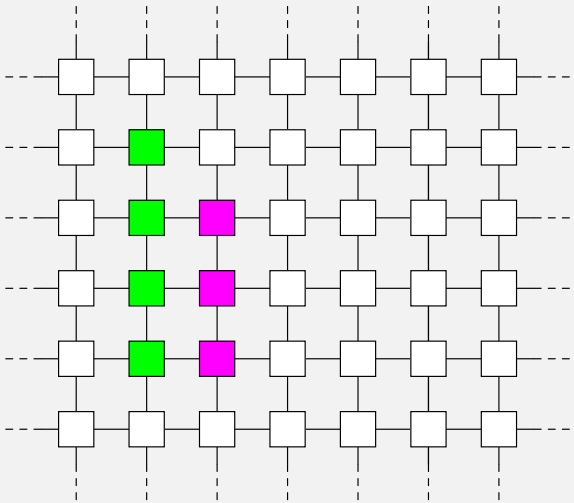
  filled with an arbitrary truth table

Content of one block



Basic FPGA structure

Overal view

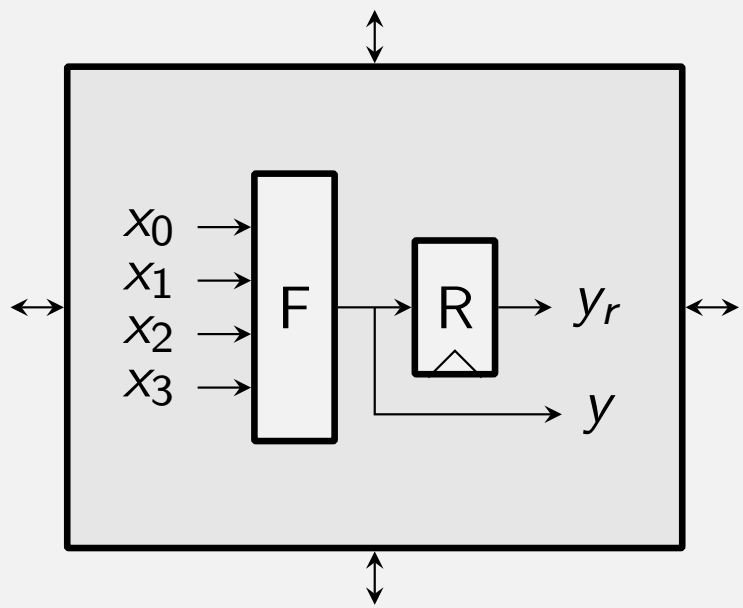


- Cell: configurable logic blocks
  - ... a small configurable automaton

- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

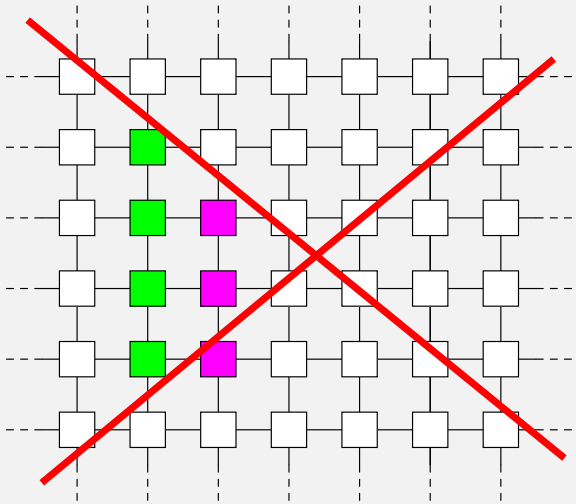
  filled with an arbitrary truth table
- Memory: 1-bit register

Content of one block



Basic FPGA structure

Overal view

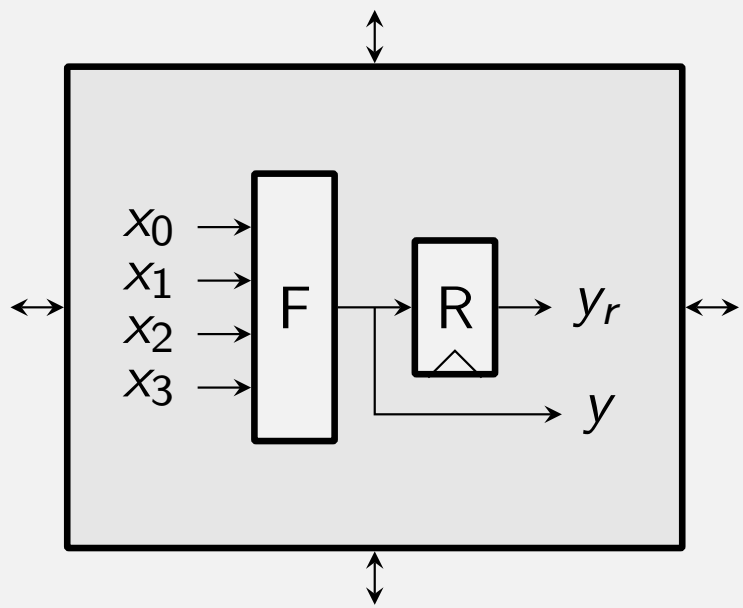


- Cell: configurable logic blocks
  - ... a small configurable automaton
- Configurable routing instead of the local routing

- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

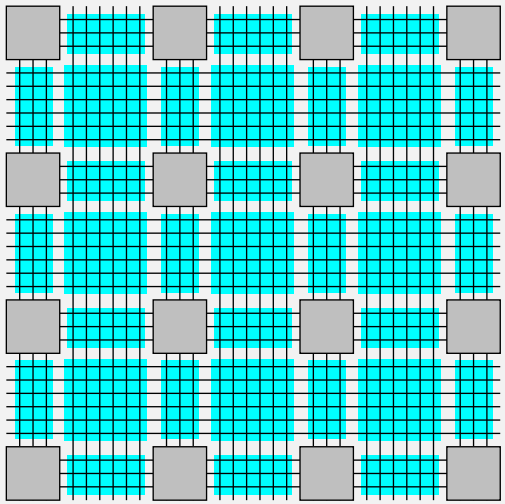
  filled with an arbitrary truth table
- Memory: 1-bit register

Content of one block



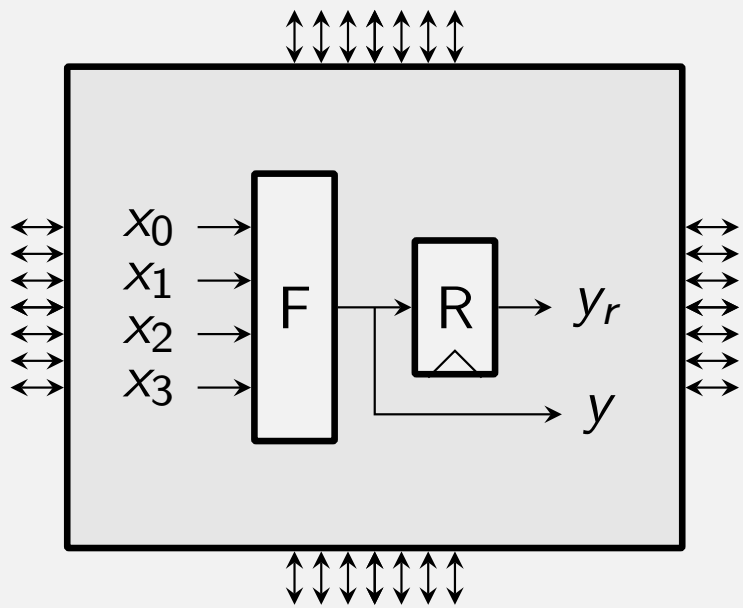
Basic FPGA structure

Overal view



- Cell: configurable logic blocks
  - ... a small configurable automaton
- Configurable routing instead of the local routing
  - need random access here

Content of one block



- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

  filled with an arbitrary truth table
- Memory: 1-bit register

In blue, *switch boxes* to connect crossing lines

Two moments in the life of an FPGA

Configuration time (a few ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

a program == a lot of configuration bits

Two moments in the life of an FPGA

Configuration time (a few ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

a program == a lot of configuration bits

Run time (forever if needed)

- Data is processed by each LUT according to its truth table
- Data moves from LUT to LUT along the (static) connexions
- The FPGA behaves as a circuit of gates

Two moments in the life of an FPGA

Configuration time (a few ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

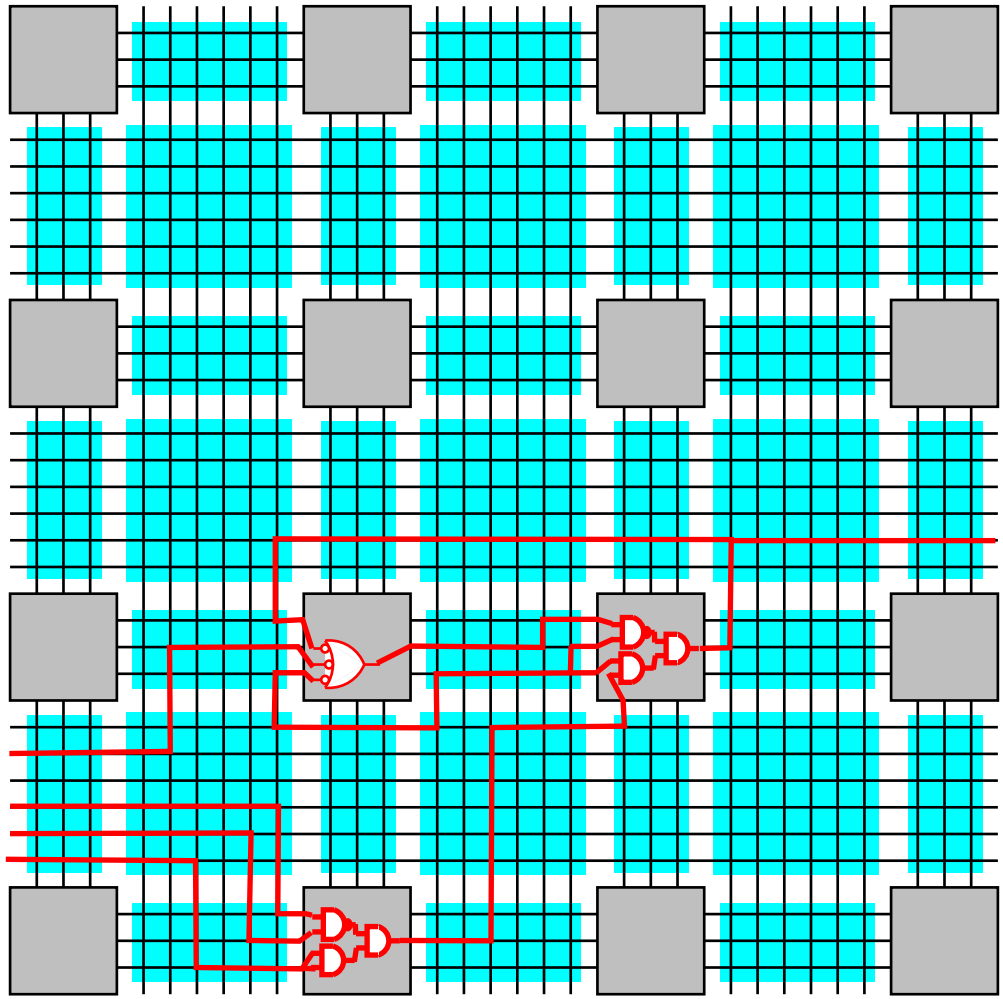
a program == a lot of configuration bits

Run time (forever if needed)

- Data is processed by each LUT according to its truth table
- Data moves from LUT to LUT along the (static) connexions
- The FPGA behaves as a circuit of gates

The programming model of FPGAs is the digital circuit.

A configured FPGA



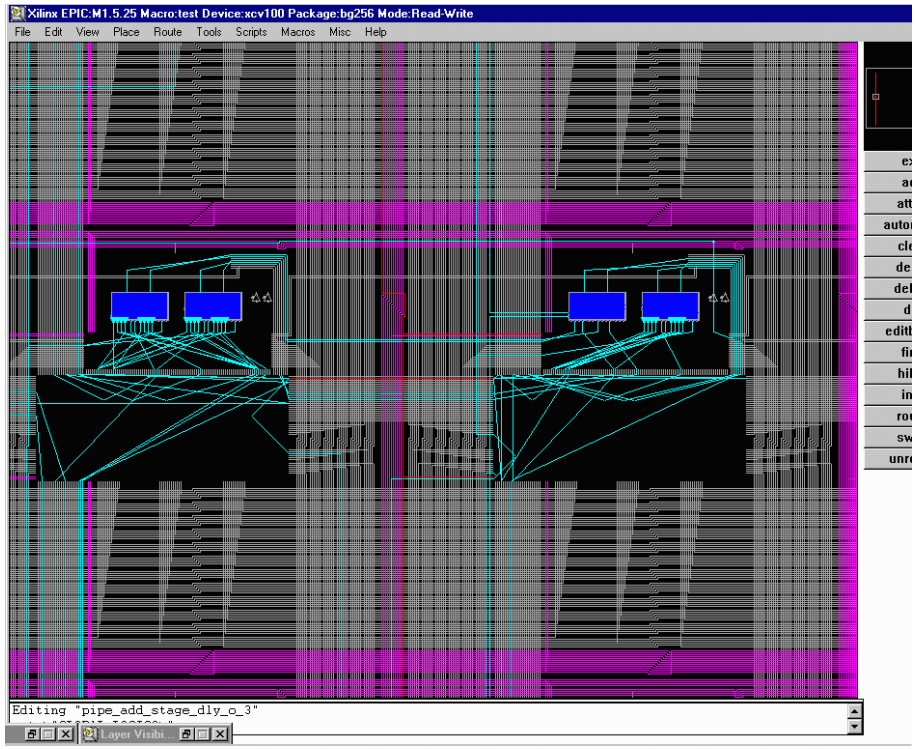
Also known as **reconfigurable circuits**

used for **reconfigurable computing**

Compared to ASIC, 1/10th the speed

Why?

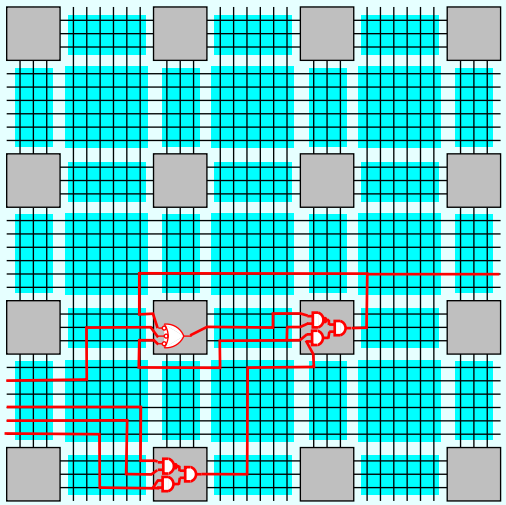
- Most of the silicon is dedicated to programmable routing



- Cost in area, but also delay: many transistors on each wire
- "Customers buy logic, but they pay for routing" (Langhammer)
- And it gets worse (Rent's law)

Rent's law?

Yet another experimental law



In a circuit of diameter n,
the number of wires crossing a diameter
is proportional to n^r with $1 < r < 2$.

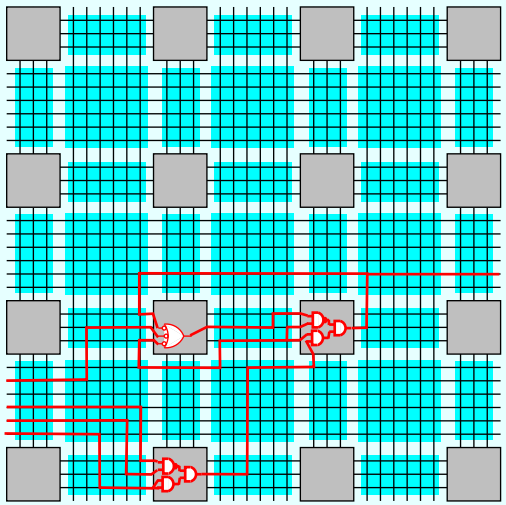
- more than proportional to n, the diameter,
- note quite proportional to the area n^2 of each half-circuit.

The value of r (Rent's exponent) depends of the class of circuit.

FPGAs are designed for worst-case circuits, hence r close to 2...

Rent's law?

Yet another experimental law



In a circuit of diameter n,
the number of wires crossing a diameter
is proportional to n^r with $1 < r < 2$.

- more than proportional to n, the diameter,
- note quite proportional to the area n^2 of each half-circuit.

The value of r (Rent's exponent) depends of the class of circuit.

FPGAs are designed for worst-case circuits, hence r close to 2...

Replace "circuit" with "city", and "wires" with "citizen commuters", and you have the explanation of the Hopeless Universal Trafic Jam in expanding cities.

Useful operators that make sense in an FPGA

- *Elementary functions ?*
  Yes iff your application needs it
- *Divider or square root?*
  Yes iff your application needs it
- *Decimal hardware?*
  Yes iff your application needs it
- *A multiplier by* $\log(2)$*?*
  Yes iff your application needs it

In FPGAs, useful means: useful to one application.

Useful operators that make sense in an FPGA

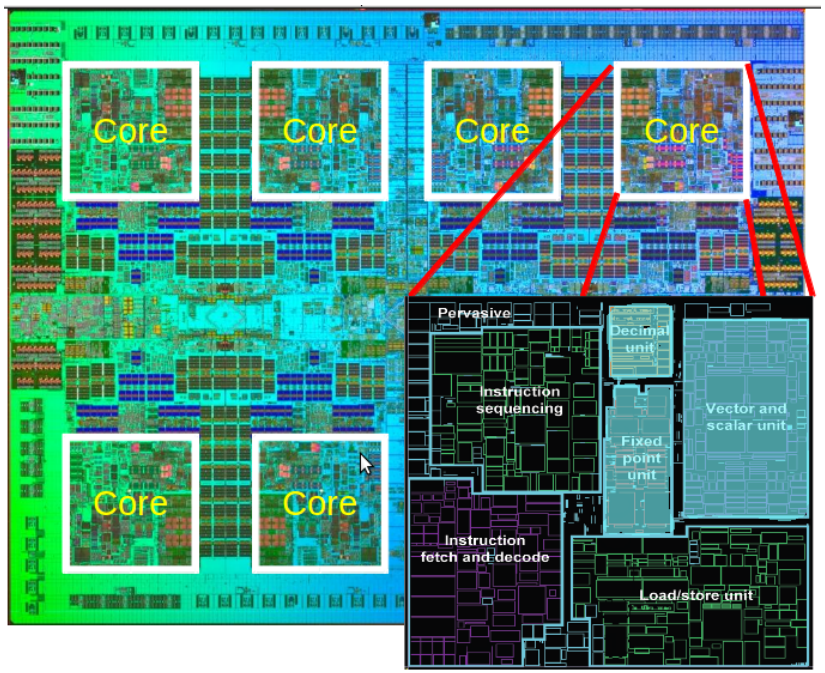
- Specialized operators: constant multipliers, squarers, ...
- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($\frac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Floating-point sums, dot products, sums of squares
- Complex arithmetic
- LNS arithmetic
- Decimal arithmetic
- Interval arithmetic
- ...

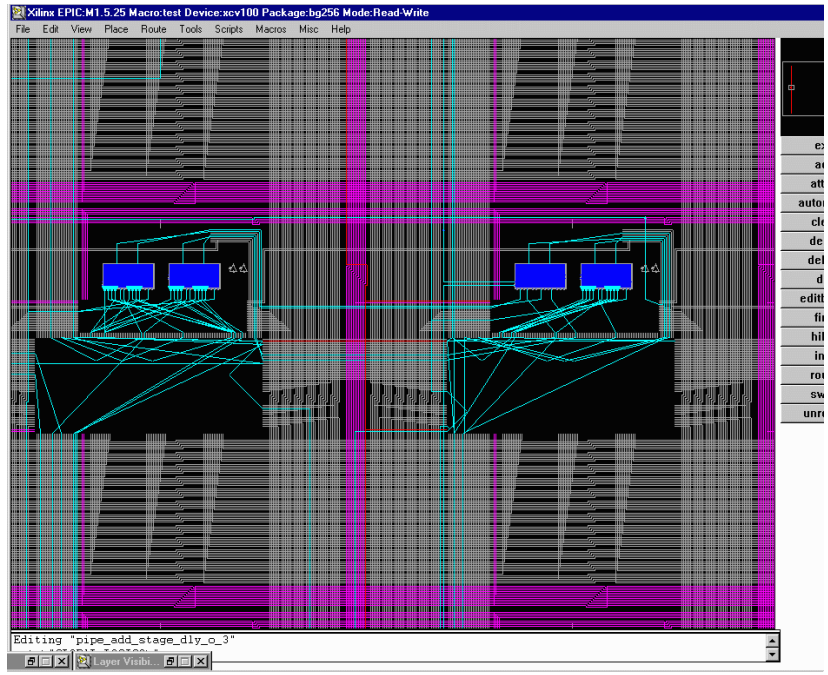
Useful operators that make sense in an FPGA

- Specialized operators: constant multipliers, squarers, ...
- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($\frac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Floating-point sums, dot products, sums of squares
- Complex arithmetic
- LNS arithmetic
- Decimal arithmetic
- Interval arithmetic
- ...
- Oh yes, basic operations, too.

Two different ways of wasting silicon

Here are two programmable chips.





Which is best for (insert your computation here) ?

Are FPGAs any good at floating-point?

Long ago (1995), people ported the basic operations: $+, -, \times$

- Versus the highly optimized FPU in the processor,
- each operator **10x slower** in an FPGA

This is the inavoidable overhead of programmability.

Are FPGAs any good at floating-point?

Long ago (1995), people ported the basic operations: $+, -, \times$

- Versus the highly optimized FPU in the processor,
- each operator **10x slower** in an FPGA

This is the inavoidable overhead of programmability.

If you lose according to a metric, change the metric.

Peak figures for double-precision floating-point exponential

- Pentium core: 20 cycles / DPExp @ 4GHz: **200 MDPExp/s**
- FPExp in FPGA: 1 DPExp/cycle @ 400MHz: **400 MDPExp/s**
- Chip vs chip: 6 Pentium cores vs 150 FPExp/FPGA
- Power consumption also better
- Single precision data better

(Intel MKL vector libm, vs FPExp in FloPoCo version 2.0.0)

Dura Amdahl lex, sed lex

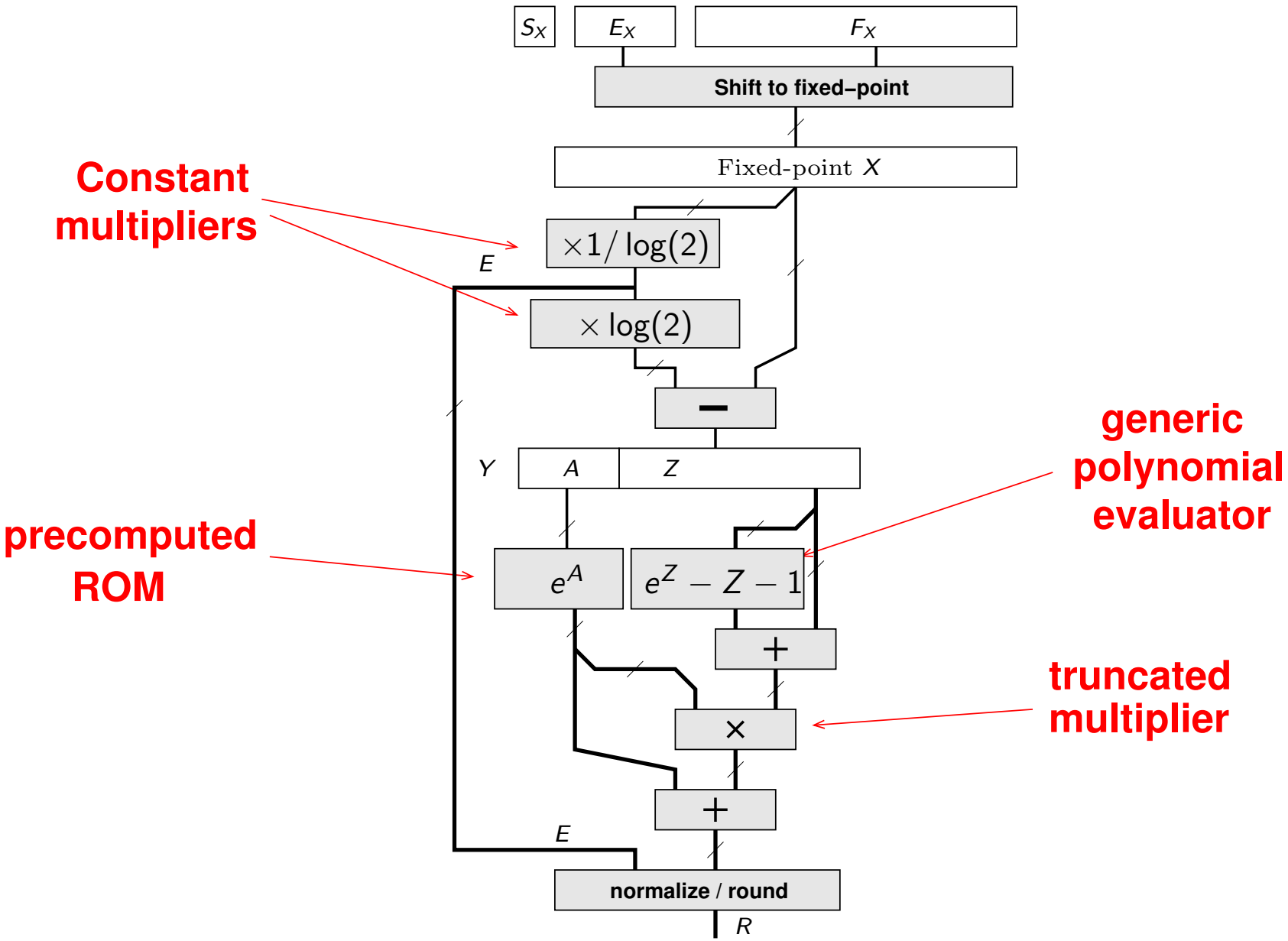
SPICE Model-Evaluation, cut from Kapre and DeHon (FPL 2009)

Table 2. Verilog-AMS Compiler Output

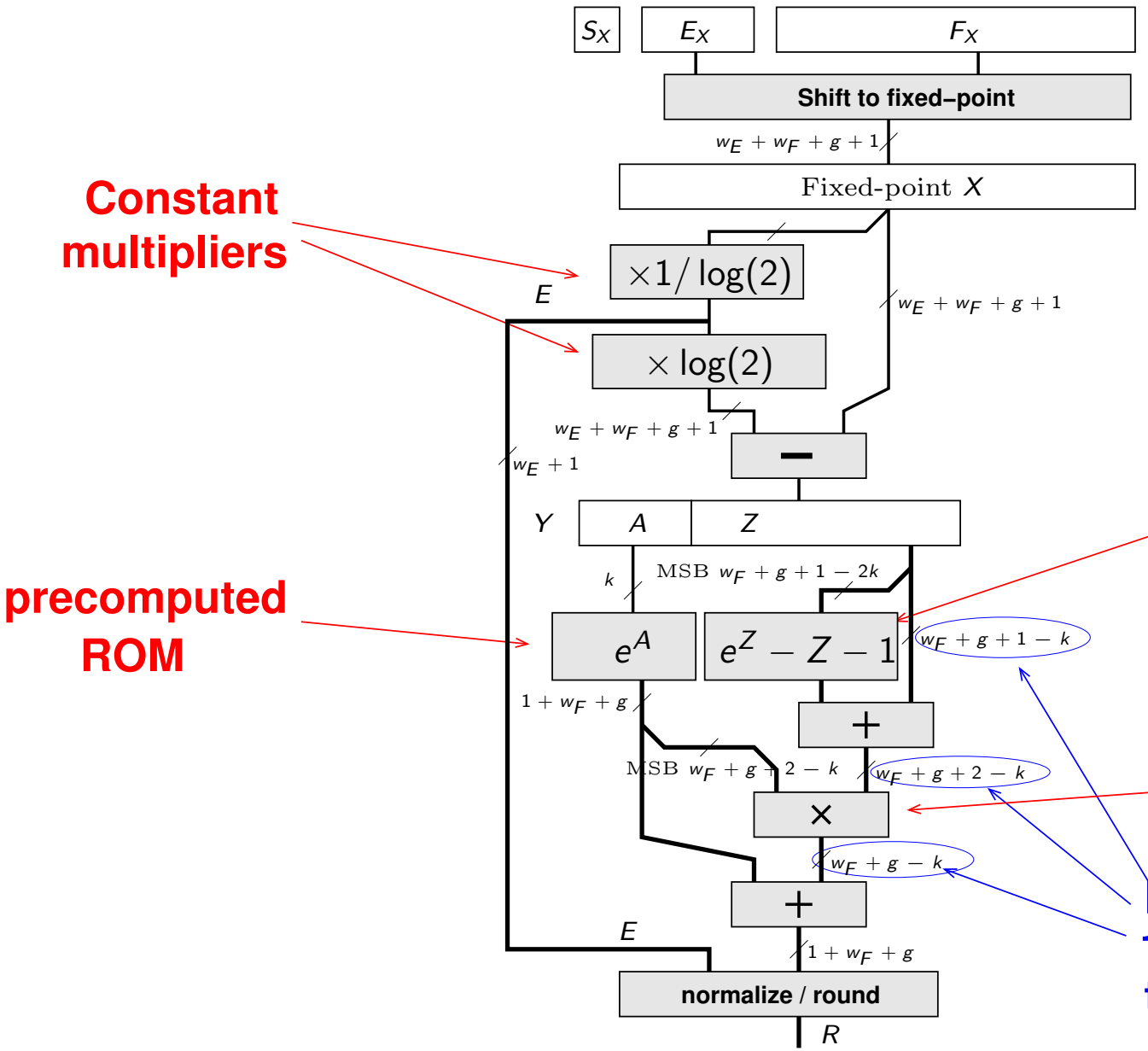
Models	Instruction Distribution					
	Add	**Mult.**	**Div.**	**Sqrt.**	**Exp.**	**Log**
`bjt`	22	30	17	0	2	0
`diode`	7	5	4	0	1	2
`hbt`	112	57	51	0	23	18
`jfet`	13	31	2	0	2	0
`mos1`	24	36	7	1	0	0
`vbic`	36	43	18	1	10	4

(who is "rare but useful" here?)

Not your Pentium's exponential



Not your Pentium's exponential



Constant multipliers

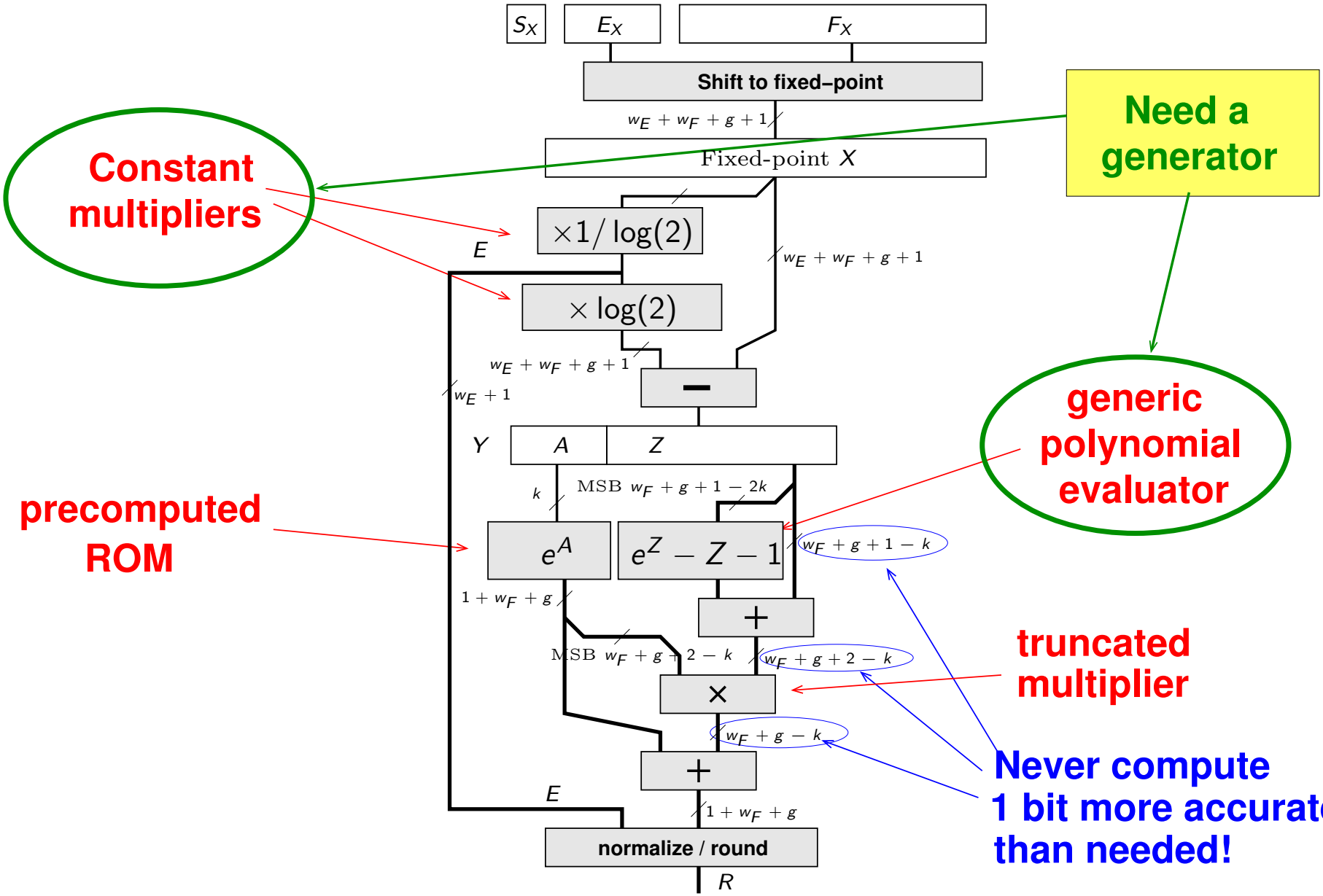
precomputed ROM

generic polynomial evaluator

truncated multiplier

Never compute 1 bit more accurately than needed!

Not your Pentium's exponential



The FloPoCo project: Not your neighbour's FPU

- A generator framework
  - written in C++, outputting VHDL
  - open and extensible
- Goal: provide application-specific arithmetic operators
  - open-ended list
  - all operators fully parameterized
  - flexible pipeline for all operators

The FloPoCo project: Not your neighbour's FPU

- A generator framework
  - written in C++, outputting VHDL
  - open and extensible
- Goal: provide application-specific arithmetic operators
  - open-ended list
  - all operators fully parameterized
  - flexible pipeline for all operators
- Approach: **computing just right**
  - Interface: never output bits that are not numerically meaningful
  - Inside: never compute bits that are not useful to the final result

Optimizing operators in context

What's nice with arithmetic operators

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)

What's nice with arithmetic operators

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))

→ Clean mathematic definition, even for floating-point arithmetic

What's nice with arithmetic operators

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))

→ Clean mathematic definition, even for floating-point arithmetic

An operator, as a *circuit*...

... is a direct acyclic graph (DAG):

- easy to build and pipeline
- easy to test against its mathematical specification

What's nice with arithmetic operators

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))

→ Clean mathematic definition, even for floating-point arithmetic

An operator, as a *circuit*...

... is a direct acyclic graph (DAG):

- easy to build and pipeline
- easy to test against its mathematical specification

And also, operators are small, no FPGA I/O problem, etc...

Sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

Sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required

Sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless

Sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - $(x^2 + y^2) + z^2$: asymmetrical

Sum of squares

$$x^2 + y^2 + z^2$$

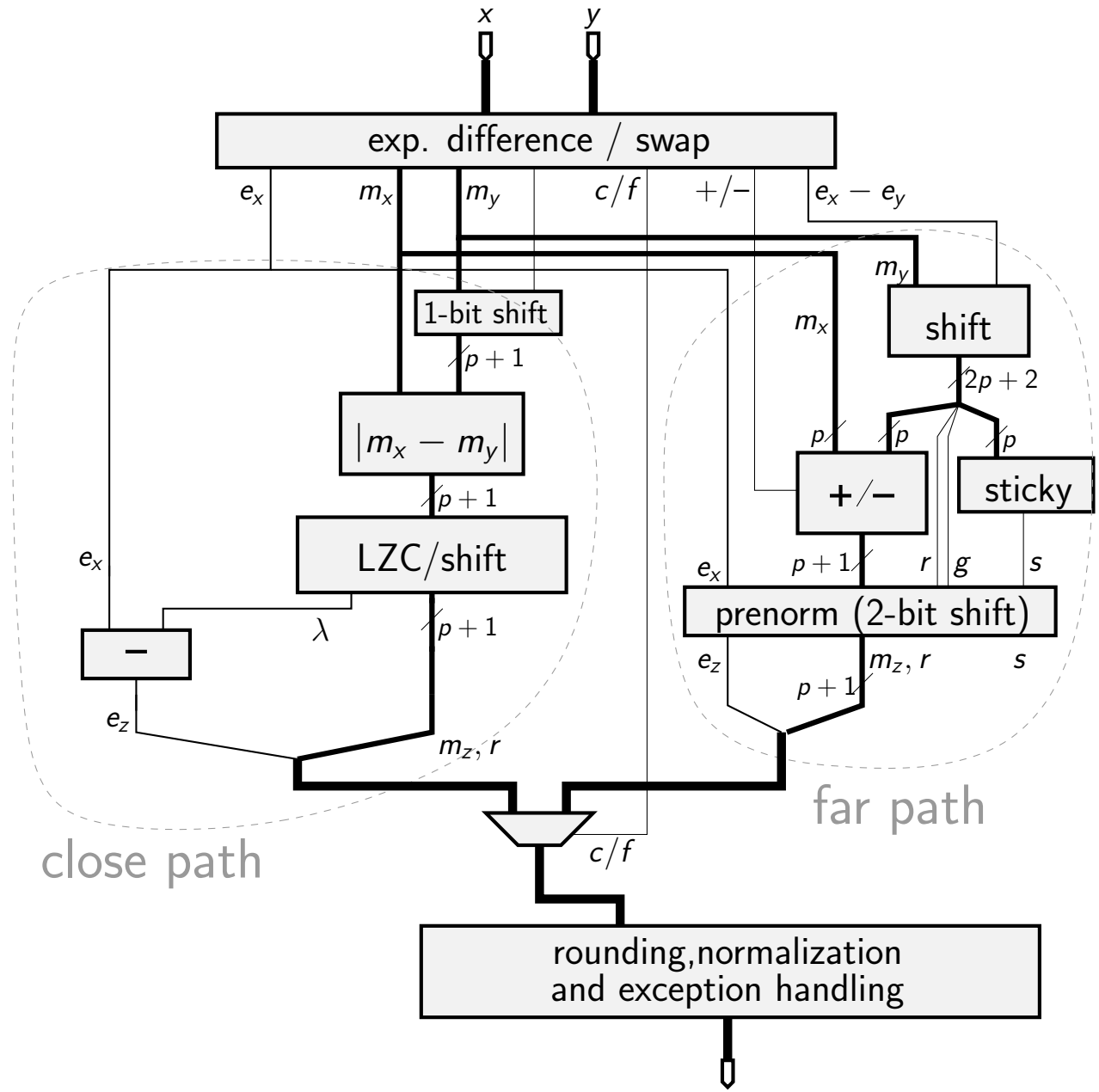
(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - $(x^2 + y^2) + z^2$: asymmetrical

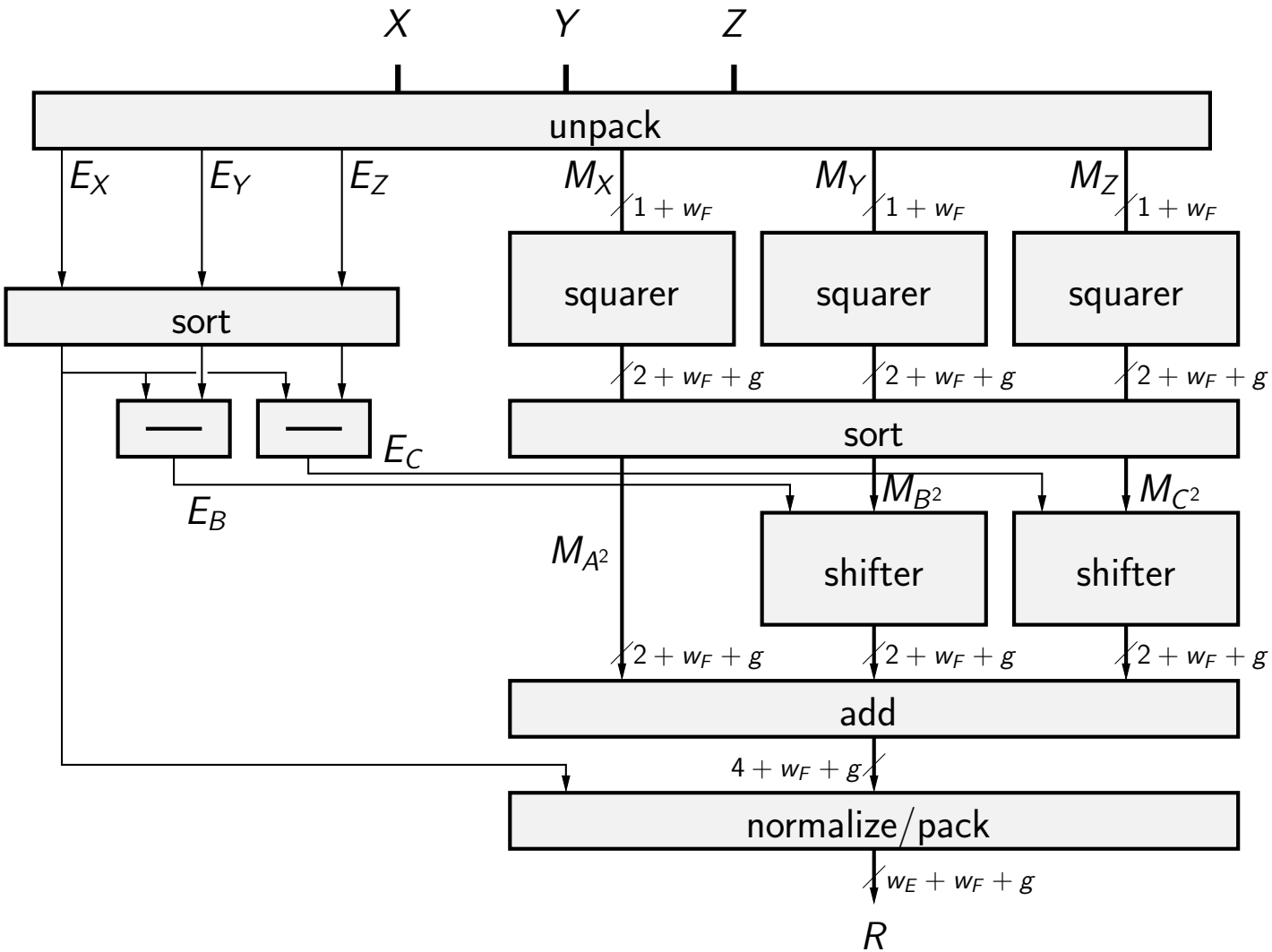
The FloPoCo Recipe

- provide the floating-point interface
- build a fixed-point architecture
- ensure a clear accuracy specification

A floating-point adder



Optimization opportunities



Optimization opportunities (2)

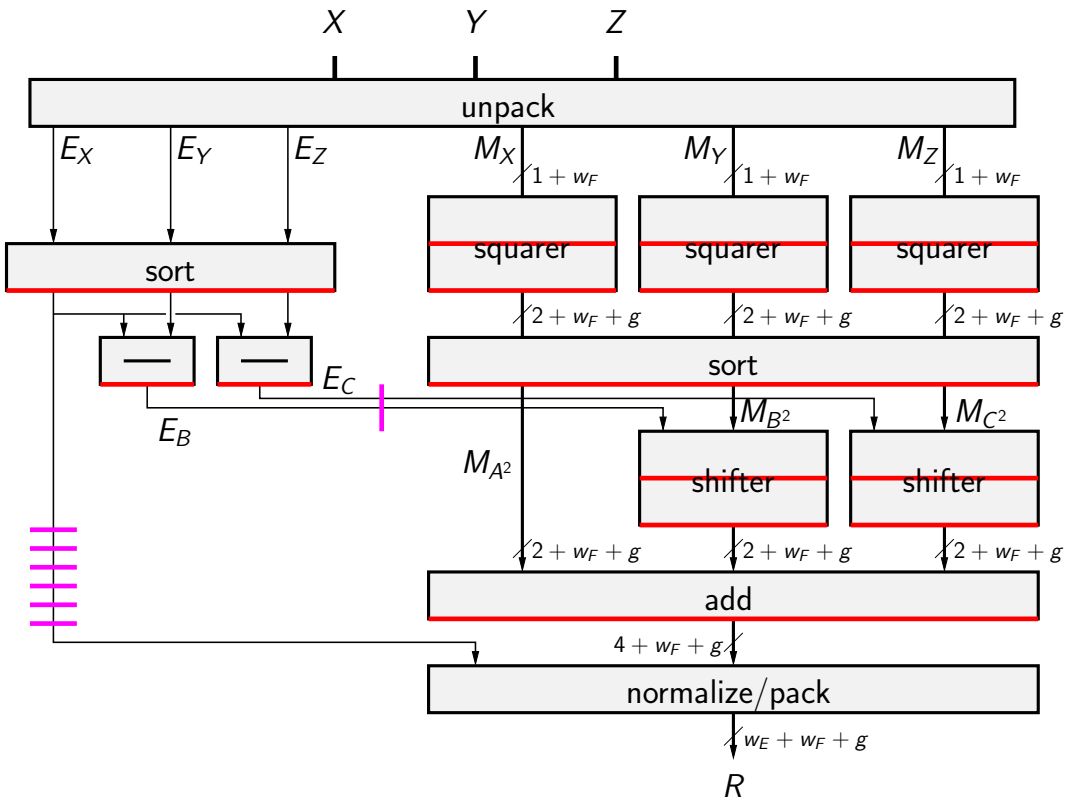
A few results for floating-point sum-of-squares on Virtex4:
(*classic:* assembly of classical FP adders and multipliers,
custom: the architecture on previous slide)

Simple Precision	area	performance
LogiCore classic	1282 slices, 20 DSP	43 cycles @ 353 MHz
FloPoCo classic	1188 slices, 12 DSP	29 cycles @ 289 MHz
FloPoCo custom	453 slices, 9 DSP	11 cycles @ 368 MHz

Double Precision	area	performance
FloPoCo classic	4480 slices, 27 DSP	46 cycles @ 276 MHz
FloPoCo custom	1845 slices, 18 DSP	16 cycles @ 362 MHz

- all performance metrics improved, FLOP/s/area more than doubled
- Plus: custom operator more accurate, and symmetrical

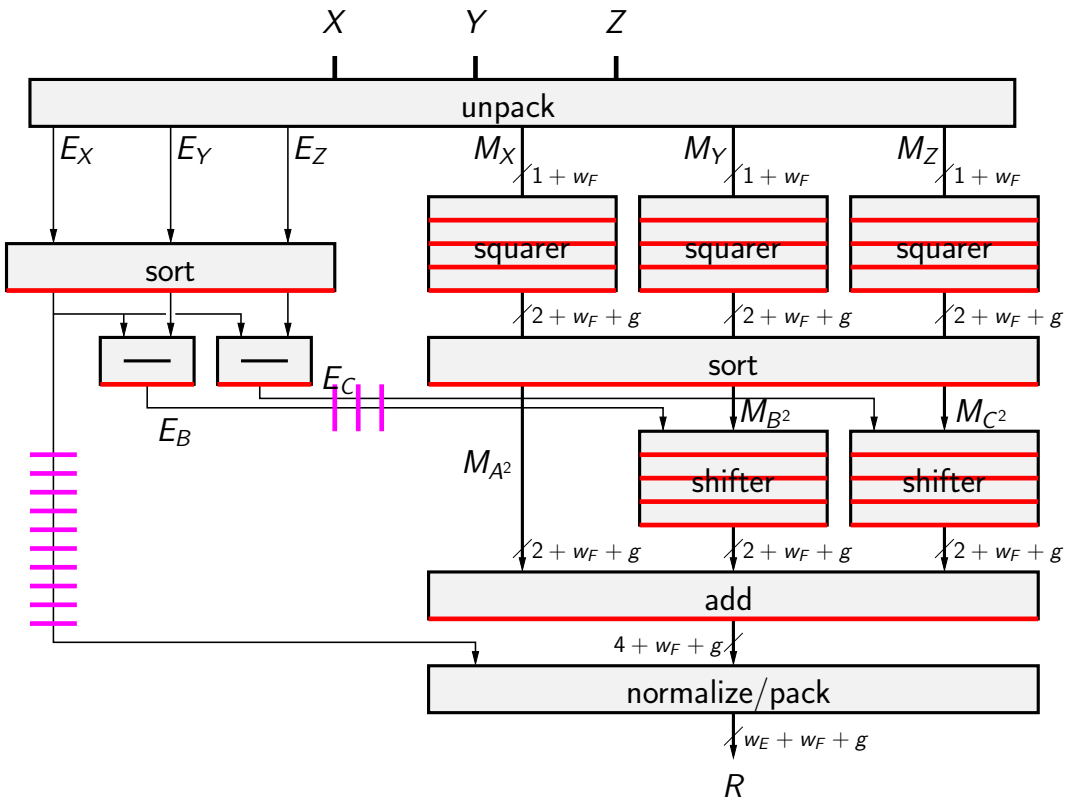
Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption

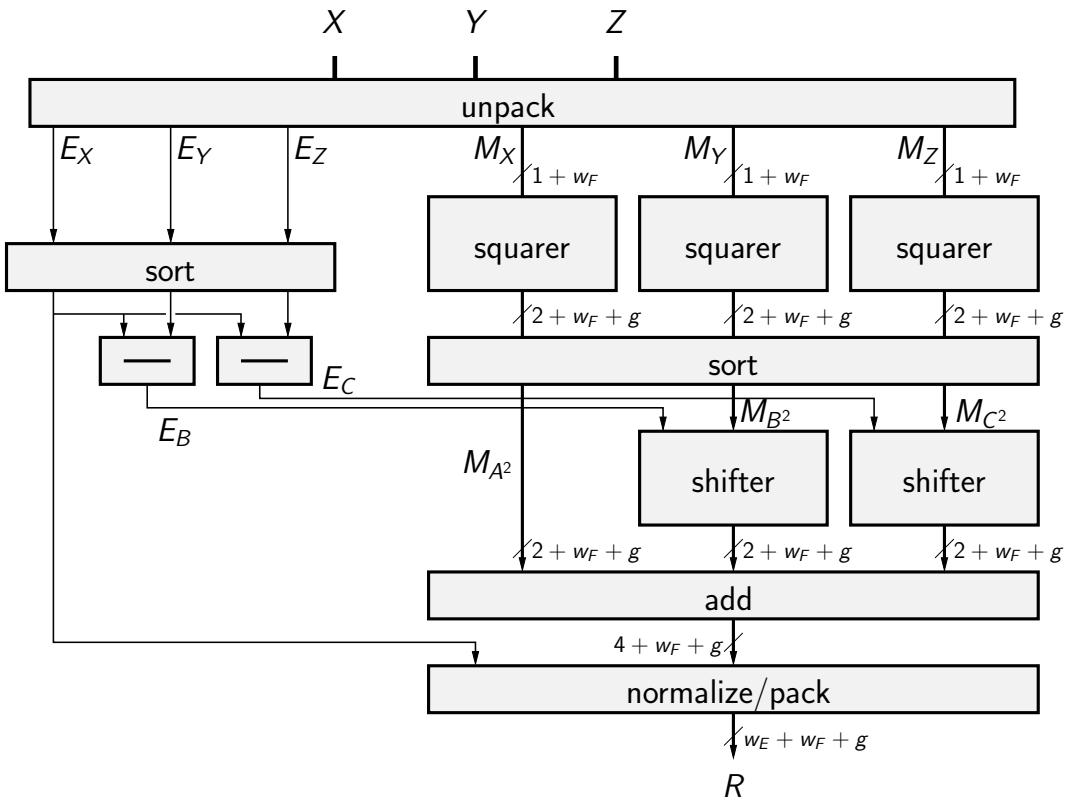
Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)

Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)
- Combinatorial

Frequency-directed pipelining

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Frequency-directed pipelining

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

Frequency-directed pipelining

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

Better because *compositional*

When you assemble components working at frequency f, you obtain a component working at frequency f.

Here should come a demo

FloPoCo is not a library, but a *generator* of operators written in C++.

- Command line syntax: a sequence of operator specifications
- Options: target frequency, target hardware, ...
- Output: synthesizable VHDL.

FloPoCo is open-source and freely available from

`http://flopoco.gforge.inria.fr/`

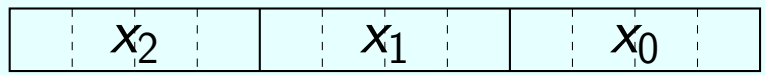
Example: Multiplication by rational constants

Multiplication by a constant, 1

FPGA-specific LUT-based methods

- Write x in radix 2^α: $x = \sum_{i=0}^{n} 2^{\alpha i} x_i \quad \text{with} \quad 0 \leq x_i < 2^\alpha$

  Ex: good old hexadecimal is $\alpha = 4$:



- then $Cx = \sum_{i=0}^{n} 2^{\alpha i} (Cx_i)$
- and tabulate the products Cx_i in α-input LUTs
- (also works if C is a real number like, say, $1/\log(2)$)

Extremely efficient for small n (input size) on LUT-based FPGAs.

Multiplication by a constant, 2

Shift-and-add methods for integer constants

- $17x = 16x + x = (x \ll 4) + x$
- $15x = 16x - x$ (Booth recoding)
- $7697x = 15x \ll 9 + 17x$ (open problem here)
- very good recent ILP-based heuristics
- In FPGAs, take into account the size of each addition

(demo?)

Extremely efficient for some constants such as 17.

Multiplication by a constant, 2

Shift-and-add methods for integer constants

- $17x = 16x + x = (x \ll 4) + x$
- $15x = 16x - x$ (Booth recoding)
- $7697x = 15x \ll 9 + 17x$ (open problem here)
- very good recent ILP-based heuristics
- In FPGAs, take into account the size of each addition

(demo?)

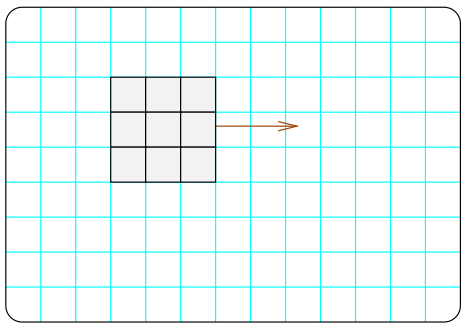
Extremely efficient for some constants such as 17.

FloPoCo offers both methods (and the exponential uses both).

Floating-point multiplication by a rational constant

Motivation

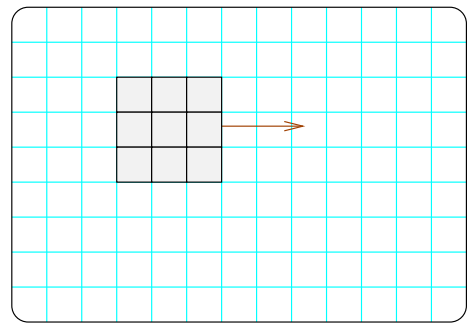
divisions by 3 and by 9 in stencil applications



Floating-point multiplication by a rational constant

Motivation

divisions by 3 and by 9 in stencil applications



$$1/3 = 0.01010101010101010101010101010101\cdots$$
$$1/9 = 0.00011100011100011100011100011100\cdots$$

Two specificities

- The binary representation of the constant is periodic
  $\longrightarrow$ specific optimisation of the shift-and-add approach
- Precision required for correct rounding

Computing periodicity

A lemma adapted from 19th century number theory

Let a/b be an irreductible rational such that

- $a < b$
- 2 divides neither a nor b (powers of two are a matter of exponent)

Then

- a/b has a purely periodic binary representation
- The period size s is the multiplicative order of 2 modulo b
  - (the smallest integer such that $2^s \mod b = 1$)
- The periodic pattern is the integer $p = \lfloor 2^s a/b \rfloor$

Computing periodicity

A lemma adapted from 19th century number theory

Let a/b be an irreductible rational such that

- $a < b$
- 2 divides neither a nor b (powers of two are a matter of exponent)

Then

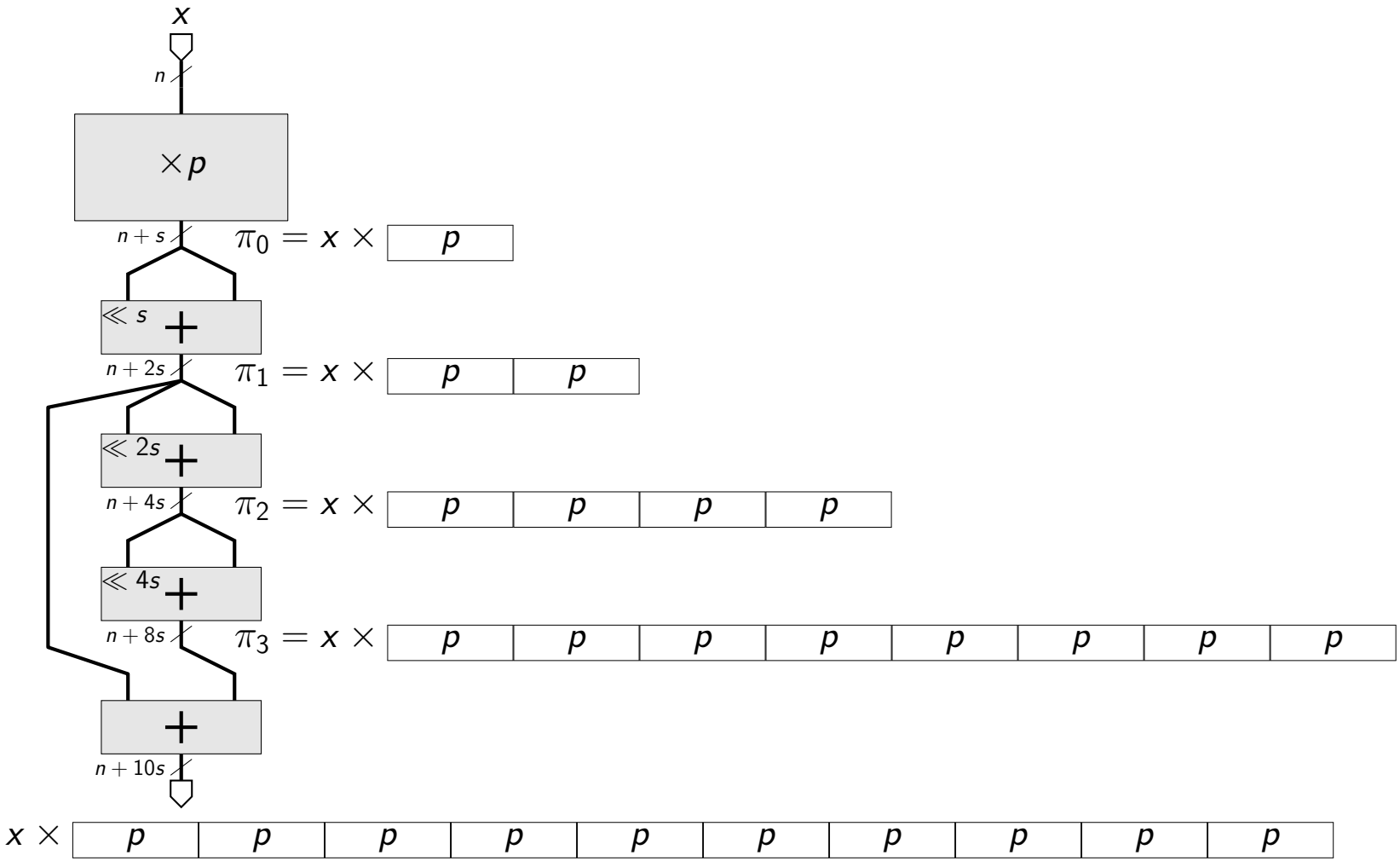
- a/b has a purely periodic binary representation
- The period size s is the multiplicative order of 2 modulo b
  - (the smallest integer such that $2^s \mod b = 1$)
- The periodic pattern is the integer $p = \lfloor 2^s a/b \rfloor$

Example: $1/9$

- $b = 9$; period size is $s = 6$ because $2^6 \mod 9 = 1$.
- The periodic pattern is $\lfloor 1 \times 2^6/9 \rfloor = 7$, which we write on 6 bits 000111, and we obtain that

$$1/9 = 0.(000111_2)^\infty \ .$$

Optimal architecture for precision p_c



Correct rounding of a floating-point x by a rational a/b

A lemma adapted from the exclusion lemma of FP division

- Correct rounding on n bits needs $n + 1 + \lceil \log_2 b \rceil$ bits of the constant

In practice, it is for free if b is small.

This work was motivated by divisions by 3 and by 9

constant	p	This work		previous SotA		depth
		p_c	#FA	p_c	#FA	
$\mathbf{1/3}$	24	32	118	27	190	4
	53	64	317	56	368	5
$p = 01_2$	113	128	792	116	1026	6
$\mathbf{1/9}$	24	30	132	29	131	5
	53	60	356	58	408	6
$p = 000111_2$	113	120	885	118	1116	7

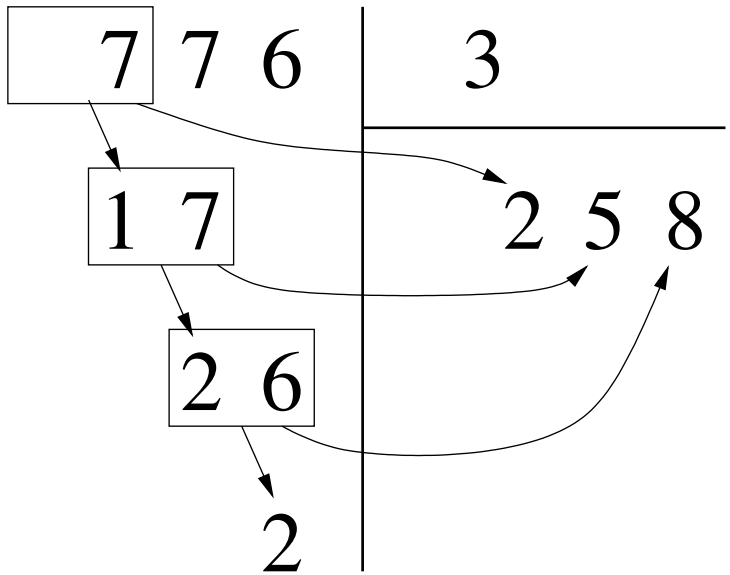
(The precisions chosen here are those of the IEEE754-2008 formats)

... But the FloPoCo code manages arbitrary a/b (including $a > b$).

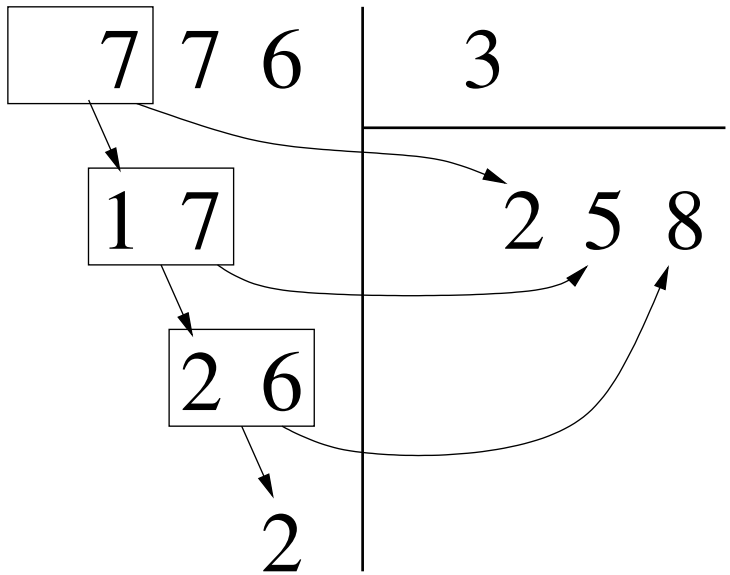
And now for something completely different

Instead of specializing multiplication, let us try and specialize division.

Anybody here remembers how we compute divisions?



Anybody here remembers how we compute divisions?



- iteration body: Euclidean division of a 2-digit decimal number by 3
- The first digit is a remainder from previous iteration:
  its value is 0, 1 or 2
- Possible implementation as a look-up table that, for each value from 00 to 29, gives the quotient and the remainder of its division by 3.

The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i \qquad \text{(split of the bits of } x \text{ into chunks of } \alpha \text{ bits)}$$

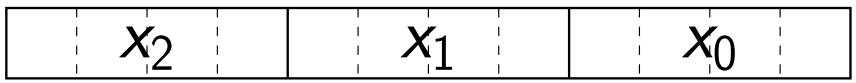
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$

(split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$



F 2 D | 3

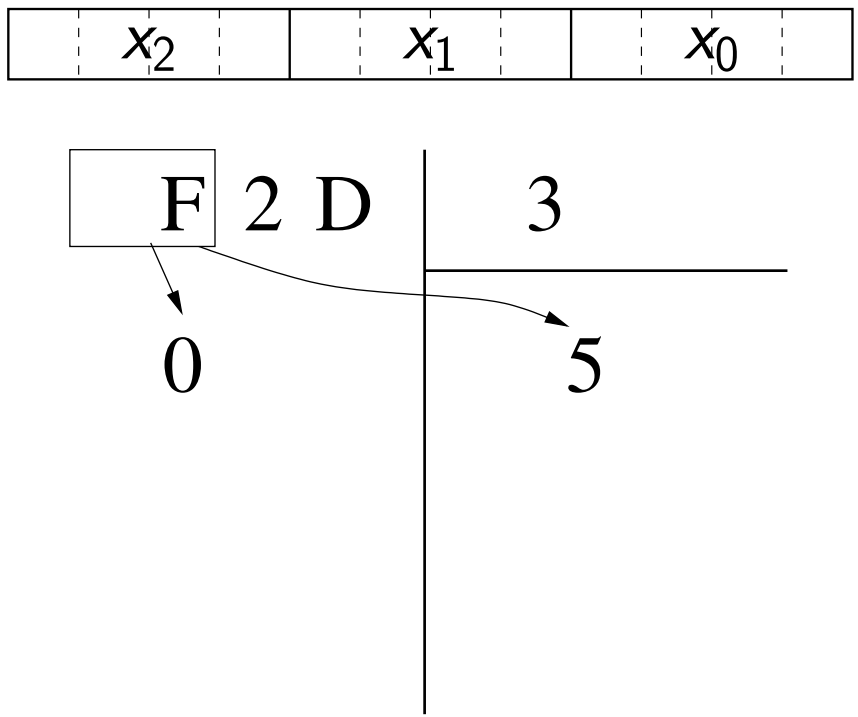
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$

(split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$

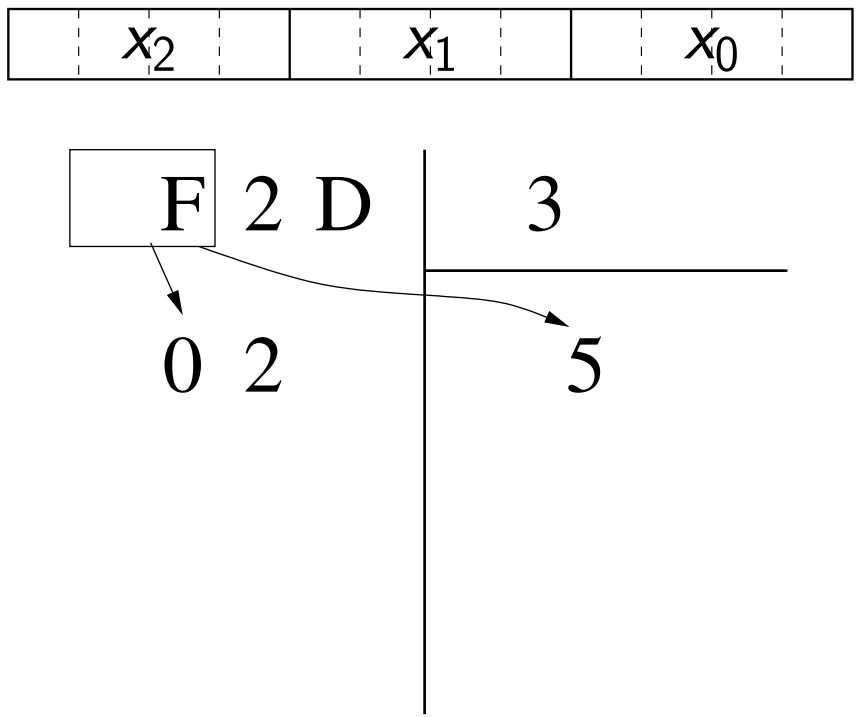


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$ (split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$

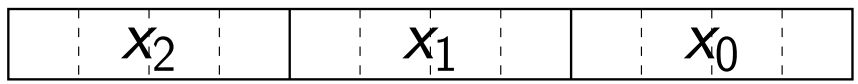


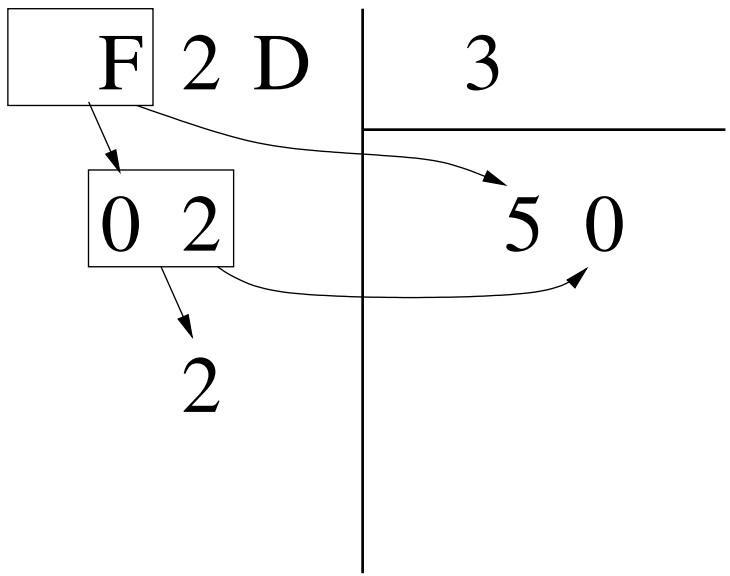
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$ (split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$



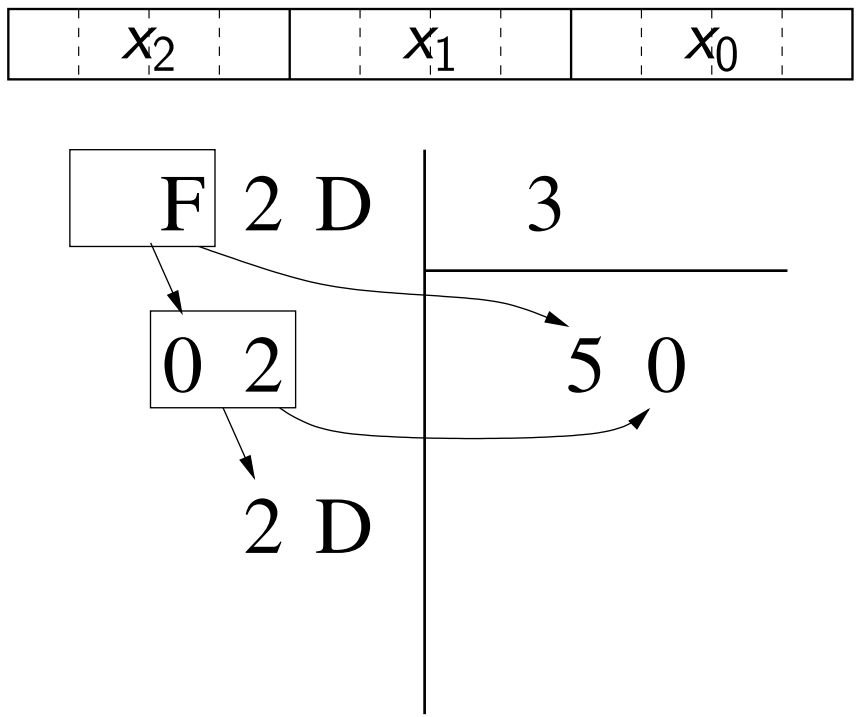


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$ (split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$

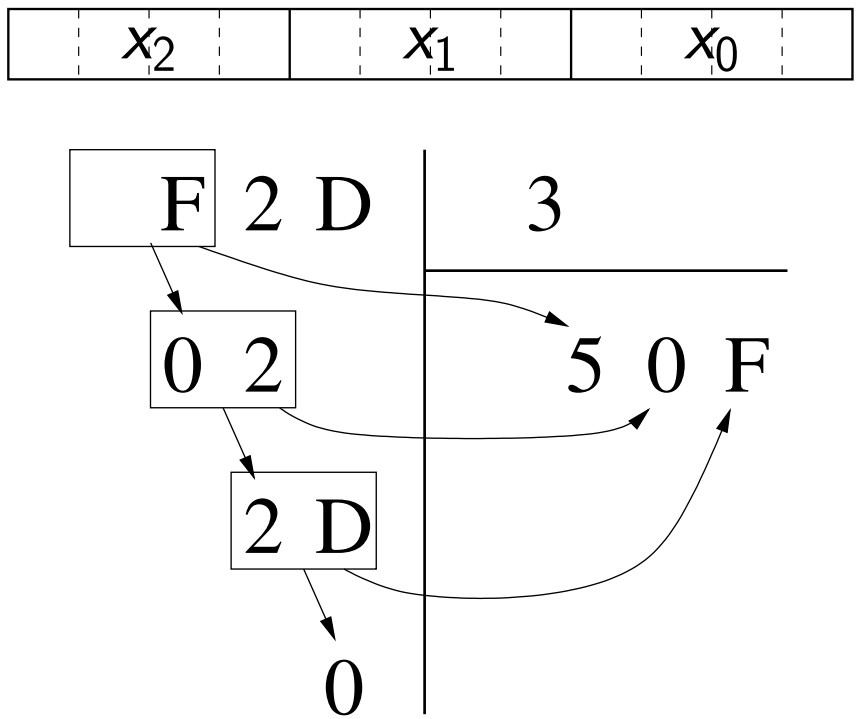


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i$$ (split of the bits of x into chunks of α bits)

Example: good old hexadecimal is $\alpha = 4$



And now for some mathematical obfuscation

procedure ConstantDiv(x, d)

$r_k \leftarrow 0$

for $i = k - 1$ **down to** 0 **do**

$y_i \leftarrow x_i + 2^\alpha r_{i+1}$ (this $+$ is a concatenation)

$(q_i, r_i) \leftarrow (\lfloor y_i/d \rfloor,\ y_i \mod d)$ (read from a table)

end for

return $q = \sum_{i=0}^{k} q_i . 2^{-\alpha i},\ r_0$

end procedure

And now for some mathematical obfuscation

procedure ConstantDiv(x, d)
  $r_k \leftarrow 0$
  for $i = k-1$ **down to** 0 **do**
    $y_i \leftarrow x_i + 2^\alpha r_{i+1}$ (this $+$ is a concatenation)
    $(q_i, r_i) \leftarrow (\lfloor y_i/d \rfloor,\ y_i \mod d)$ (read from a table)
  end for
  return $q = \sum_{i=0}^{k} q_i . 2^{-\alpha i},\ r_0$
end procedure

Each iteration

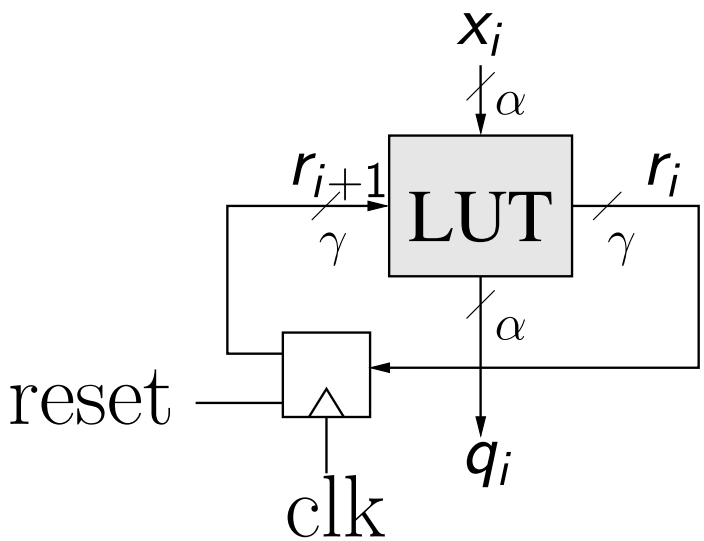
- consumes α bits of x, and a remainder of size $\gamma = \lceil \log_2 d \rceil$
- produces α bits of q, and a remainder of size γ
- implemented as a table with $\alpha + \gamma$ bits in, $\alpha + \gamma$ bits out

At this point nobody wants to see the proof

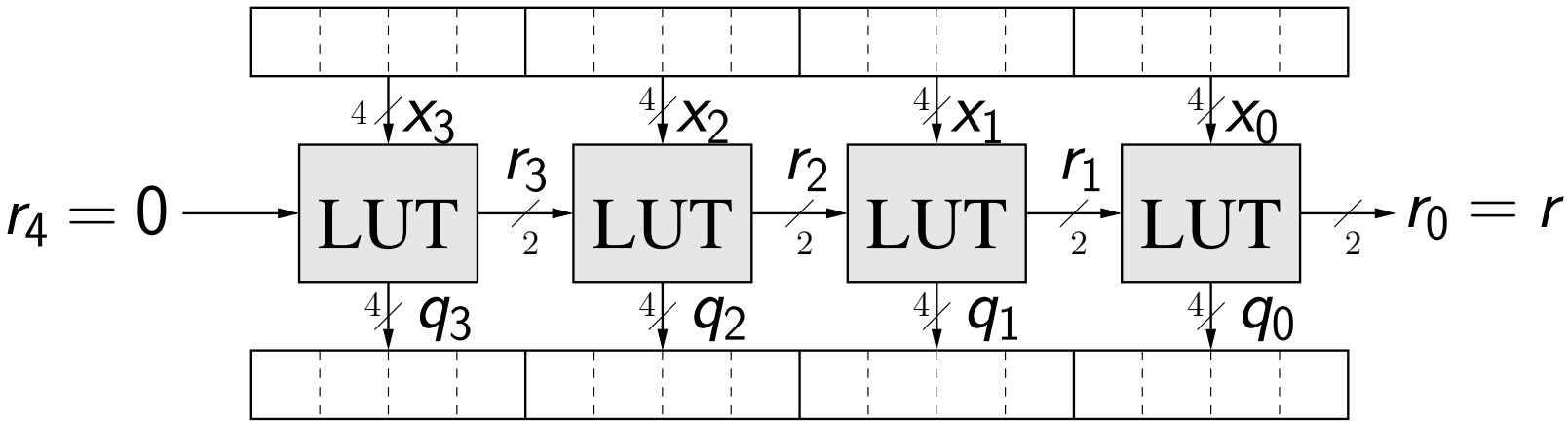
(if you're convinced the decimal version works...)

- prove that we indeed compute the Euclidean division
- prove that the result is indeed a radix-2^α number

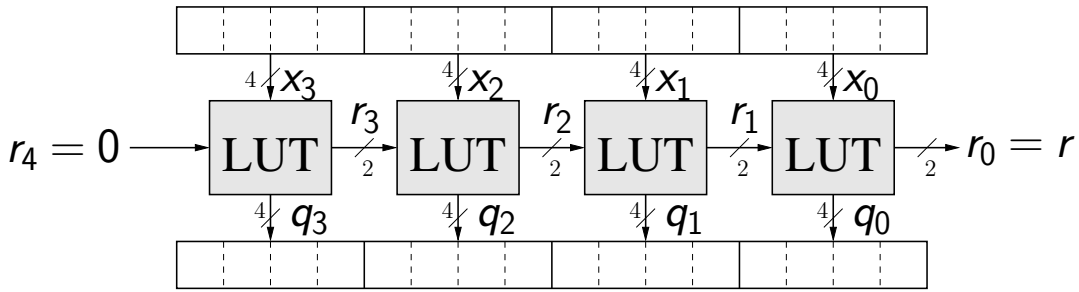
Sequential implementation



Unrolled implementation



Logic-based version

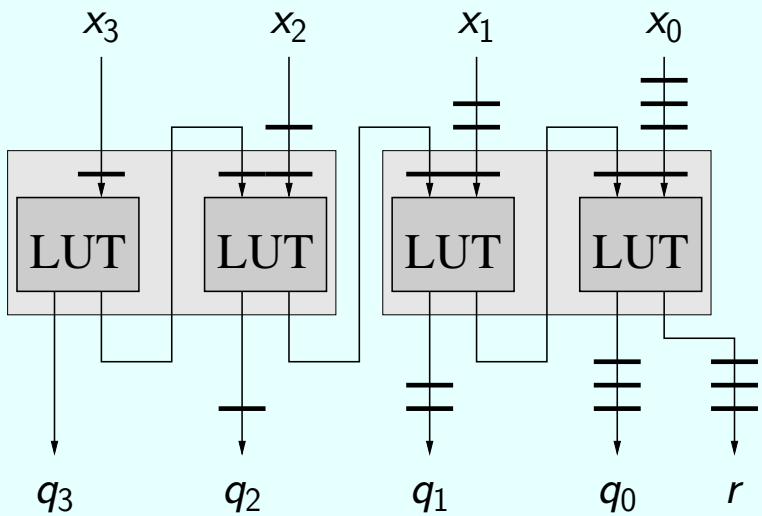


For instance, assuming a 6-input LUTs (e.g. LUT6)

- A 6-bit in, 6-bit out consumes 6 LUT6
- Size of remainder is $\gamma = \log_2 d$
- If $d < 2^5$, very efficient architecture: $\alpha = 6 - \gamma$
- The smaller d, the better
- Easy to pipeline (one register behind each LUT)

Dual-port RAM-based version?

For larger d?



(not really studied, waiting for the demand)

Synthesis results on Virtex-5 for combinatorial Euclidean division

constant	$n = 32$ bits LUT6	(predicted)	latency
$d = 3$ ($\alpha = 4$)	47	(6*8=48)	7.14ns
$d = 5$ ($\alpha = 3$)	60	(6*11=66)	6.79ns
$d = 7$ ($\alpha = 3$)	60	(6*11=66)	7.30ns

constant	$n = 64$ bits LUT6	(predicted)	latency
$d = 3$ ($\alpha = 4$)	95	(6*16=96)	14.8ns
$d = 5$ ($\alpha = 3$)	125	(6*22=132)	13.8ns
$d = 7$ ($\alpha = 3$)	125	(6*22=132)	15.0ns

Synthesis results on Virtex-5 for combinatorial Euclidean division

constant	$n = 32$ bits LUT6	(predicted)	latency
$d = 3$ ($\alpha = 4$)	47	(6*8=48)	7.14ns
$d = 5$ ($\alpha = 3$)	60	(6*11=66)	6.79ns
$d = 7$ ($\alpha = 3$)	60	(6*11=66)	7.30ns

constant	$n = 64$ bits LUT6	(predicted)	latency
$d = 3$ ($\alpha = 4$)	95	(6*16=96)	14.8ns
$d = 5$ ($\alpha = 3$)	125	(6*22=132)	13.8ns
$d = 7$ ($\alpha = 3$)	125	(6*22=132)	15.0ns

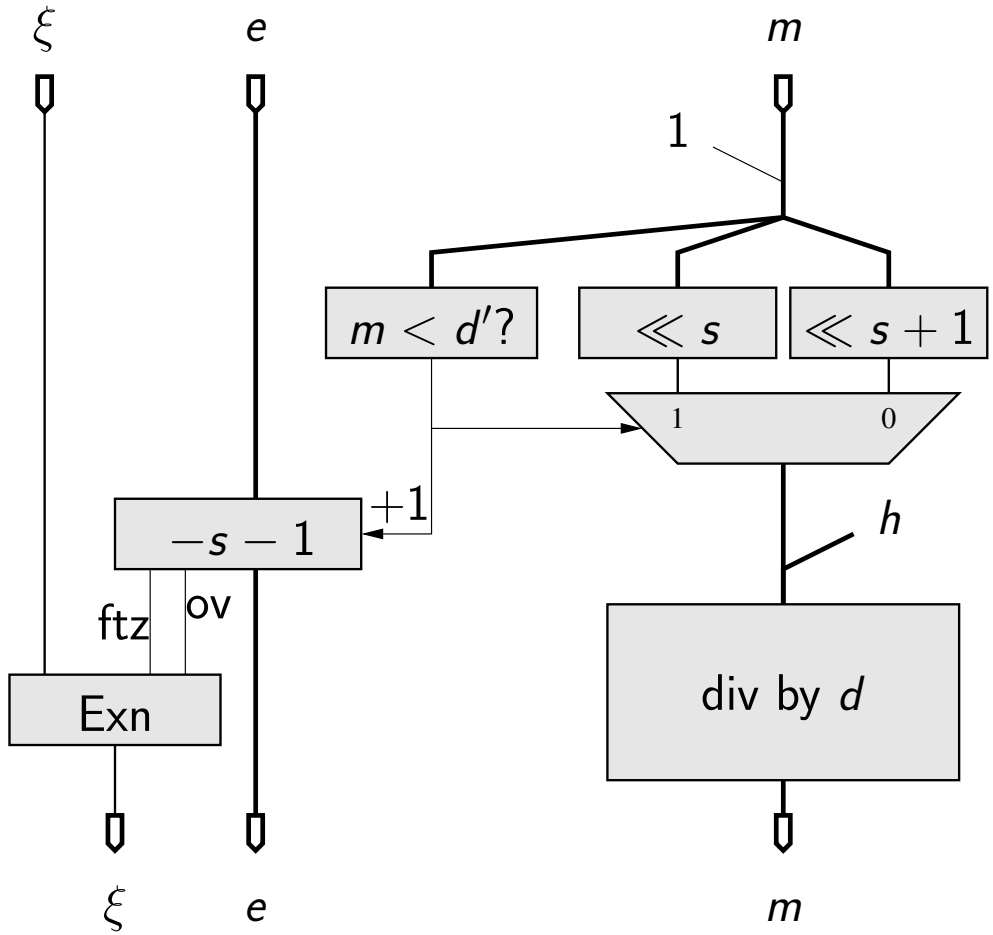
Logic optimizer even finds something to chew: *don't care* lines in the tables.

Synthesis results on Virtex-5 for pipelined Euclidean division by 3

$n = 32$ bits	
FF + LUT6	performance
33 Reg + 47 LUT	1 cycle @ 230 MHz
58 Reg + 62 LUT	2 cycles @ 410 MHz
68 Reg + 72 LUT	3 cycles @ 527 MHz

$n = 64$ bits	
FF + LUT6	performance
122 Reg + 112 LUT	2 cycles @217 MHz
168 Reg + 198 LUT	5 cycles @ 410 MHz
172 Reg + 188 LUT	7 cycles @ 527 MHz

Floating-point version is cheap, too



- pre-normalisation and pre-rounding:

$$\circ\left(\frac{2^{s+\epsilon}m}{d}\right) = \left\lfloor \frac{2^{s+\epsilon}m}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{2^{s+\epsilon}m + d/2}{d} \right\rfloor$$

Synthesis results on Virtex-5 for pipelined floating-point division by 3

single precision

FF + LUT6	performance
35 Reg + 69 LUT	1 cycle @ 217 MHz
105 Reg + 83 LUT	3 cycles @ 411 MHz
standard correctly rounded divider	
1122 Reg + 945 LUT	17 cycles @ 290 MHz

double precision

FF + LUT6	performance
122 Reg + 166 LUT	2 cycles @ 217 MHz
245 Reg + 250 LUT	6 cycles @ 410 MHz
using shift-and-add	
282 Reg + 470 LUT	5 cycles @ 307 MHz

Was it worth to spend so much time on division by 3?

Was it worth to spend so much time on division by 3?

(this slide intentionally left blank)

Was it worth to spend so much time on division by 3?

(this slide intentionally left blank)

(three years later, Ugurdag et al spent more time on a parallel version)

My personal record

Two weeks from the first intuition of the algorithm
to complete pipelined FloPoCo implementation + paper submission.

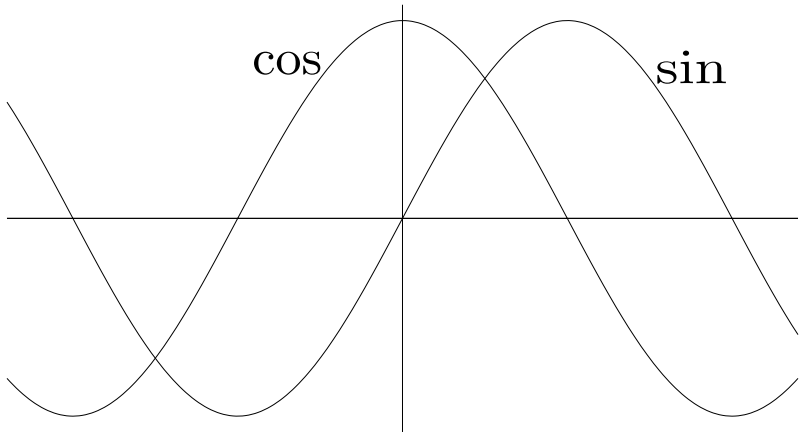
Implementation time

- 10 minutes to obtain a testbench generator
- 1/2 day for the integer Euclidean division
- 20 mn for its flexible pipeline
- 1/2 day for the FP divider by 3
- and again 20 mn

This was advertising for the FloPoCo framework.

Example: Sin/Cos

Introduction



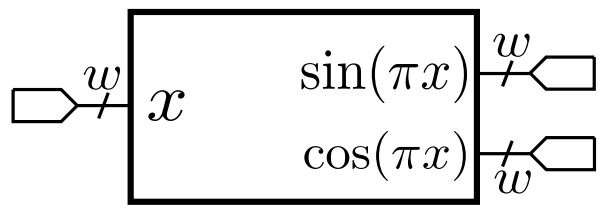
- **Why** compute the trigonometric functions sine and cosine?
  - fundamental in signal processing and signal processing applications like FFT, modulation/demodulation, frequency synthesizers, ...
- **How** to compute them ? In this work:
  1. the classical CORDIC algorithm, based on additions and shifts
  2. a method based on tables and multipliers, suited for modern FPGAs
  3. a generic polynomial approximation

  Which is best on FPGAs?
- **What is the cost of w bits of sine and cosine?**

Which method is best on FPGAs?

A fair comparison of methods computing **sine** and **cosine**:

- **same specification** (the best possible one)
  - Fixed-point inputs and outputs compute $\sin(\pi x)$ and $\cos(\pi x)$ for $x \in [-1, 1)$
  - **Faithful rounding**: **all** the produced **bits are useful**, no wasted resources



- **same effort** (the best possible one)
  - open-source implementations in FloPoCo
  - state-of-the-art?

Computing just one, or both?

- some applications need both sine and cosine (e.g. rotation)
- some methods compute both

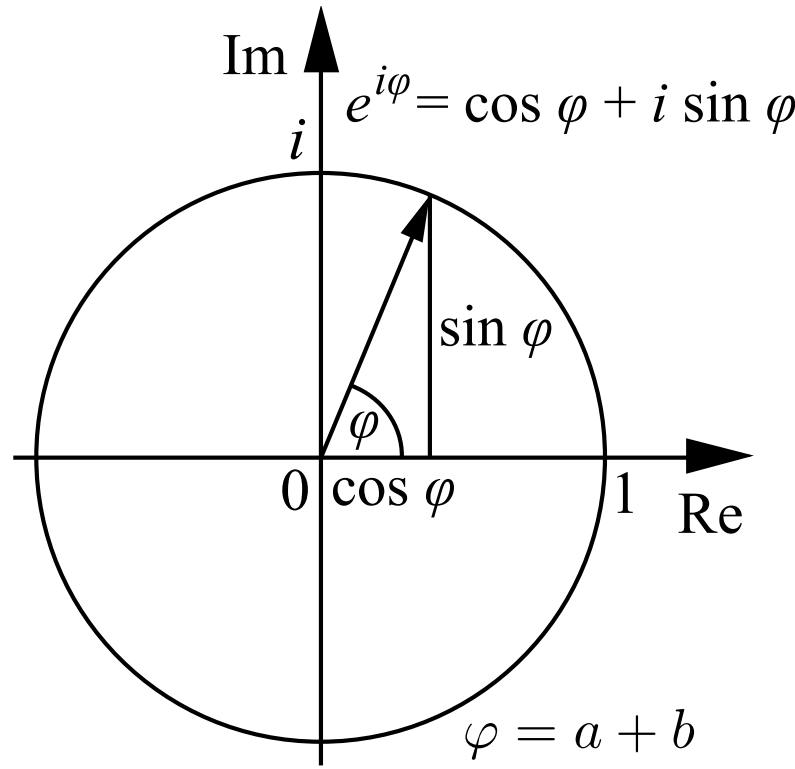
Textbook Stuff

- Decomposition of the exponential in two exponentials

$$e^{i(a+b)} = e^{ia} \times e^{ib}$$

- From complex to real

$$e^{i\varphi} = \cos(\varphi) + i\sin(\varphi)$$



- Decompose a rotation in smaller sub-rotations

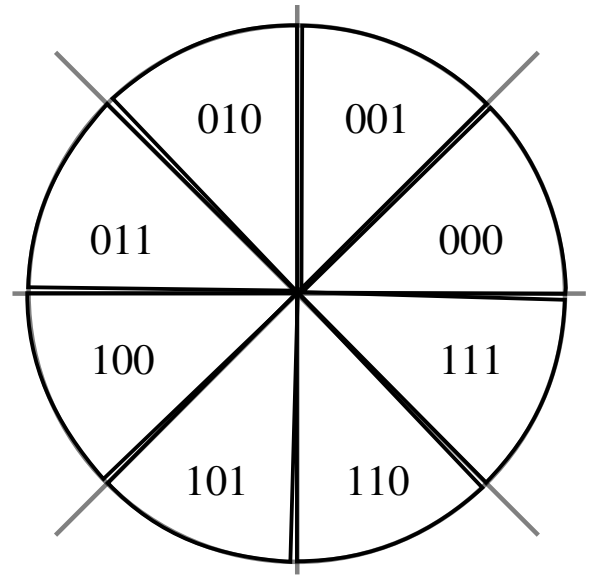
$$\begin{cases} \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b) \\ \cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b) \end{cases}$$

Argument Reduction

- based on the 3 MSBs of the input angle x
  - s - **s**ign
  - q - **q**uadrant
  - o - **o**ctant
- remaining argument $y \in [0, 1/4)$

$$y' = \begin{cases} \frac{1}{4} - y \text{ if } o = 1 \\ y \text{ otherwise.} \end{cases}$$

- compute $\cos(\pi y')$ and $\sin(\pi y')$
- reconstruction:



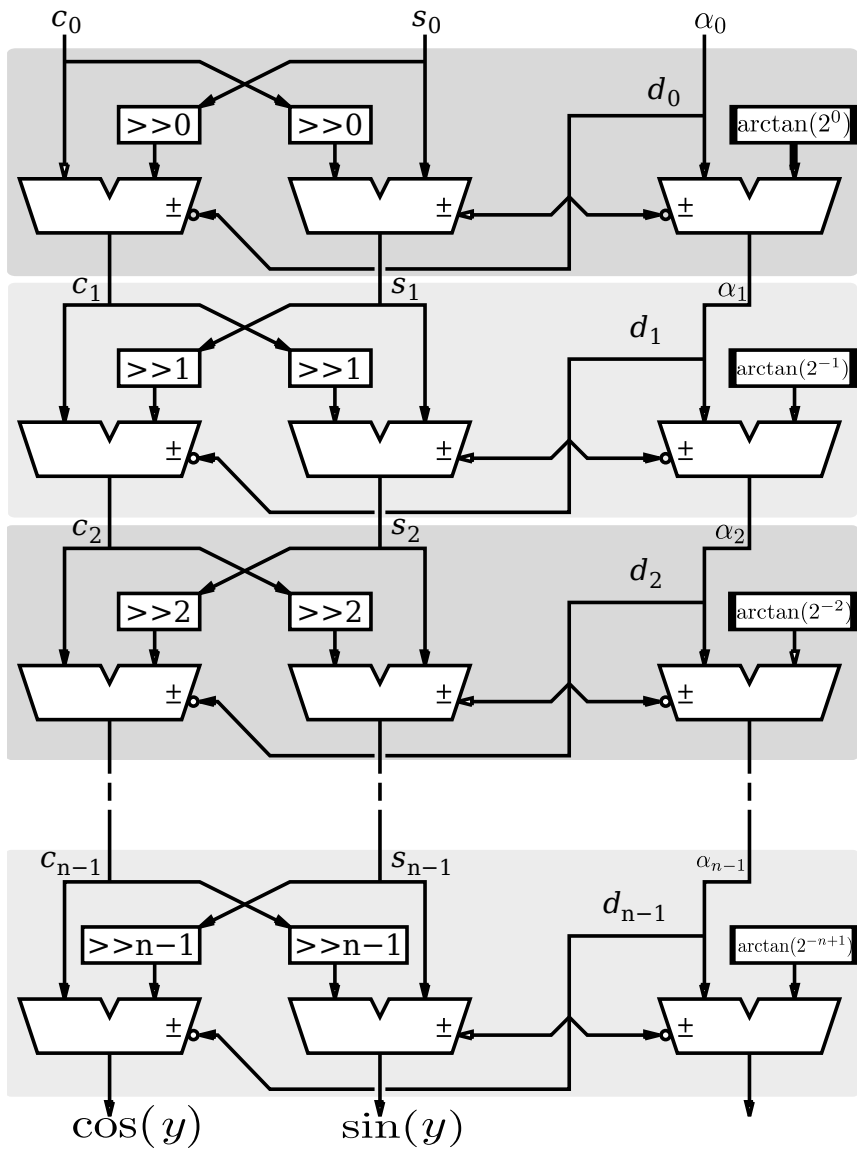
sqo	Reconstruction
000	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = \cos(\pi y') \end{cases}$
001	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = \sin(\pi y') \end{cases}$
010	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = -\sin(\pi y') \end{cases}$
011	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = -\cos(\pi y') \end{cases}$

CORDIC Architecture

$$\begin{cases} c_0 &= \frac{1}{\prod_{i=1}^{n}\sqrt{1+2^{-i}}} \\ s_0 &= 0 \\ \alpha_0 &= y \quad \text{(the reduced argument)} \end{cases}$$

$$\begin{cases} d_i &= +1 \text{ if } \alpha_i > 0, \text{ otherwise } -1 \\ c_{i+1} &= c_i - 2^{-i} d_i s_i \\ s_{i+1} &= s_i + 2^{-i} d_i c_i \\ \alpha_{i+1} &= \alpha_i - d_i \arctan(2^{-i}) \end{cases}$$

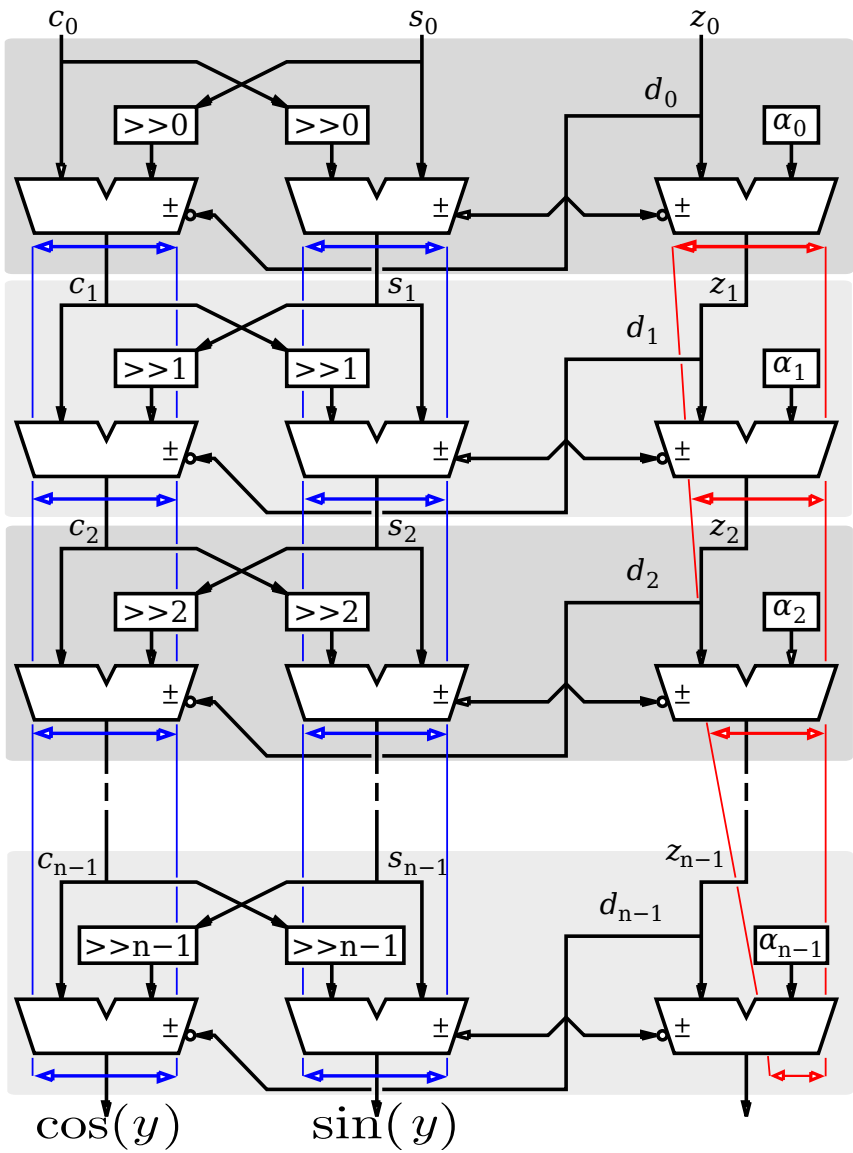
$$\begin{cases} c_{n\to\inf} &= \cos(y) \\ s_{n\to\inf} &= \sin(y) \\ \alpha_{n\to\inf} &= 0 \end{cases}$$



CORDIC Improvements

Reduced α-Datapath

- $\alpha_i < 2^{-i}$
- decrement the α-datapath by 1 bit per iteration
- benefits
  - saves space
  - saves latency



CORDIC Improvements

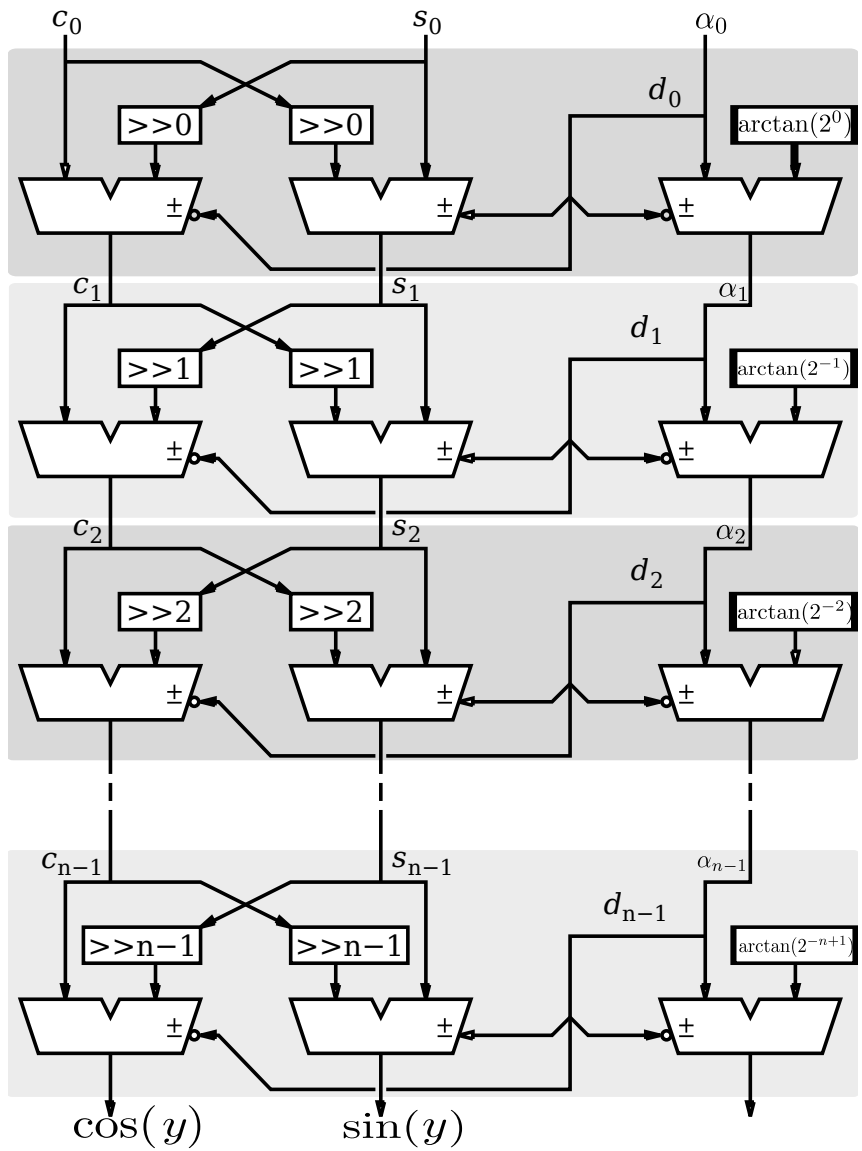
Reduced Iterations

- stop iterations when they can be replaced by a single rotation, with enough accuracy

$$\begin{cases} \sin(\alpha) \simeq \alpha \\ \cos(\alpha) \simeq 1 \end{cases}$$

- half the iterations replaced by

$$\begin{cases} x_{i+1} = x_i + \alpha \cdot y_i \\ y_{i+1} = y_i - \alpha \cdot x_i \end{cases}$$



CORDIC Improvements

Reduced Iterations

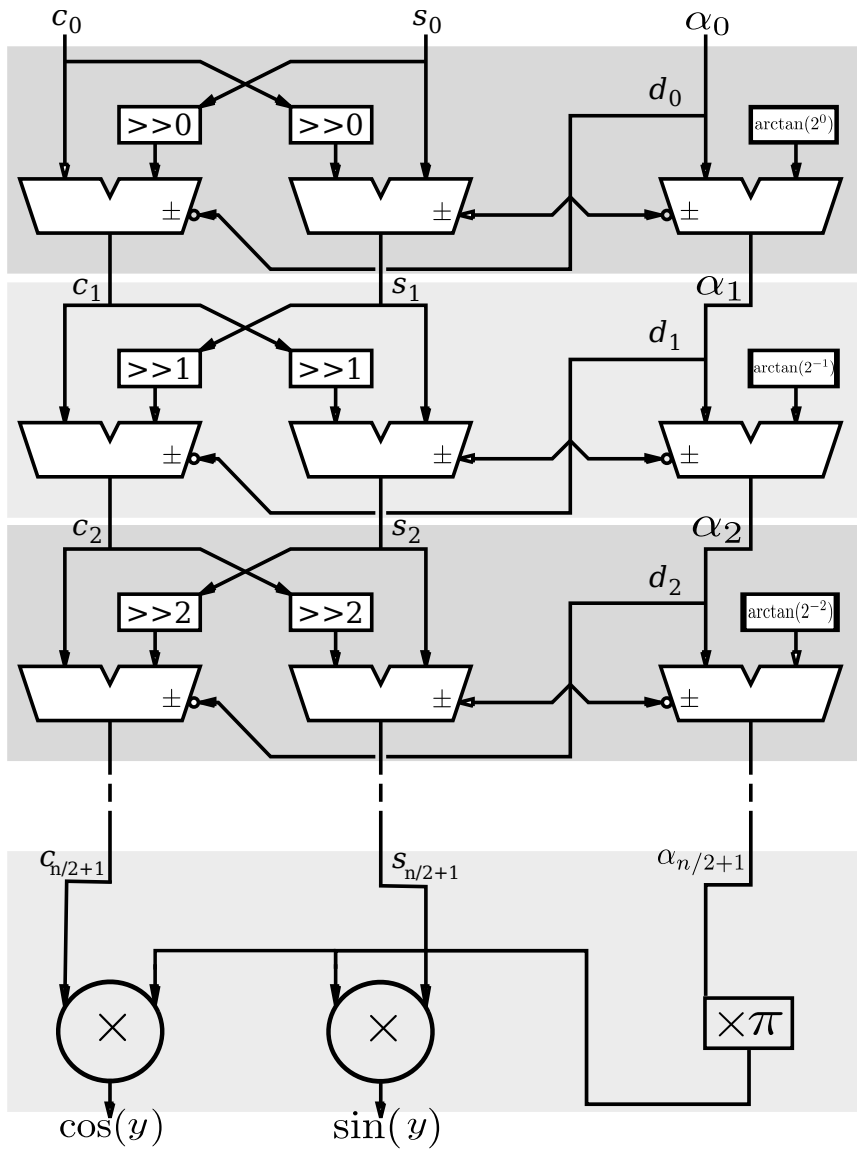
- stop iterations when they can be replaced by a single rotation, with enough accuracy

$$\begin{cases} \sin(\alpha) \simeq \alpha \\ \cos(\alpha) \simeq 1 \end{cases}$$

- half the iterations replaced by

$$\begin{cases} x_{i+1} = x_i + \alpha \cdot y_i \\ y_{i+1} = y_i - \alpha \cdot x_i \end{cases}$$

- only 2 multiplications
  - 2 DSPs for up to 32 bits
  - truncated multiplications for larger sizes



CORDIC Error Analysis

Goal: last-bit accuracy of the result

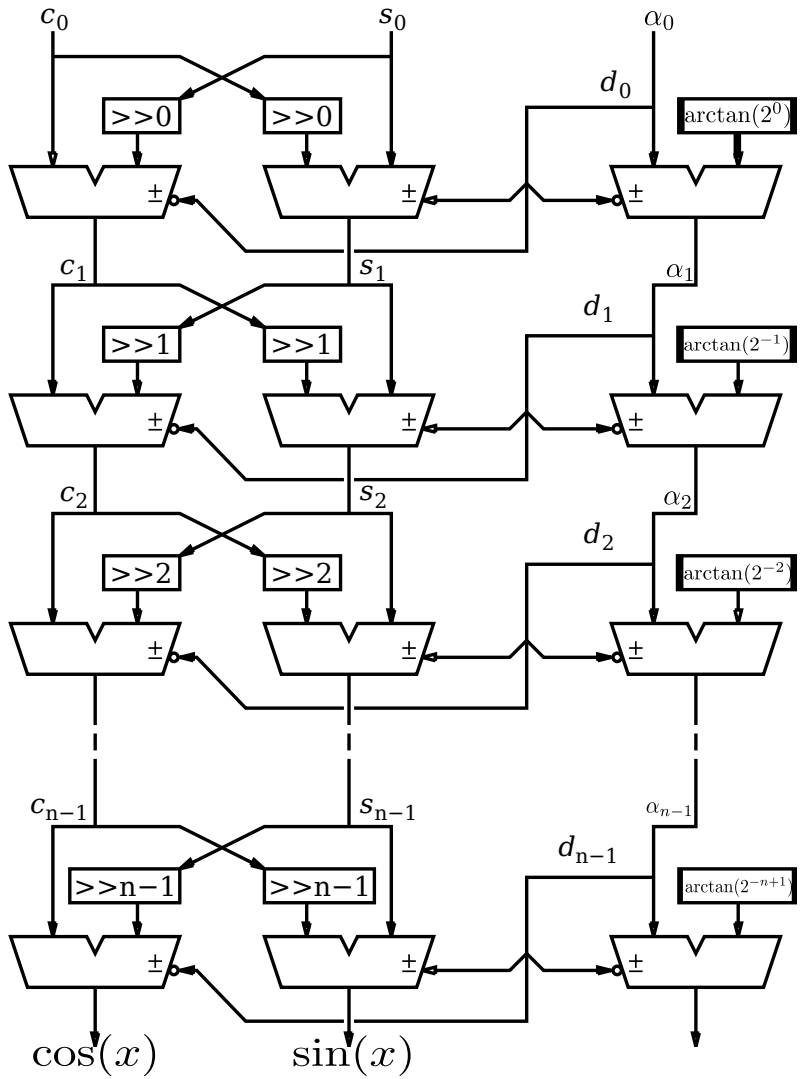
- the result is within 1**ulp** of the mathematical result
- **ulp** = weight of least significant bit

Intermediate precision

- approximations and roundings $\rightarrow$ computations on **w+g** bits internally
- guard bits **g**

Error budget: total of 1**ulp**

- $\frac{1}{2}$**ulp** for the final rounding error
- $\frac{1}{4}$**ulp** for the method error
- $\frac{1}{4}$**ulp** for the rounding errors

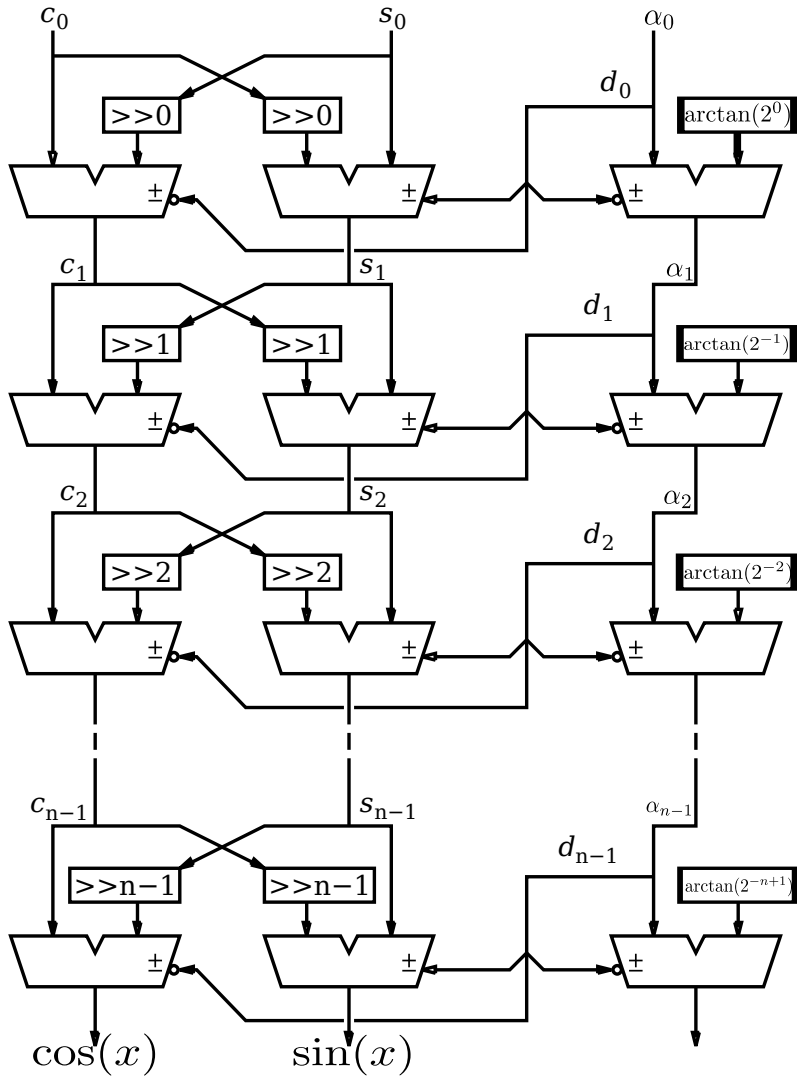


CORDIC Error Analysis (1)

Analysis: method error (ε_{method})

- ε_{method} of the order of the value of α_{final}
- α_{final} can be bounded numerically

$\rightarrow$ number of iterations:
smallest number for which
$\varepsilon_{method} < 2^{-w-2}$



CORDIC Error Analysis (2)

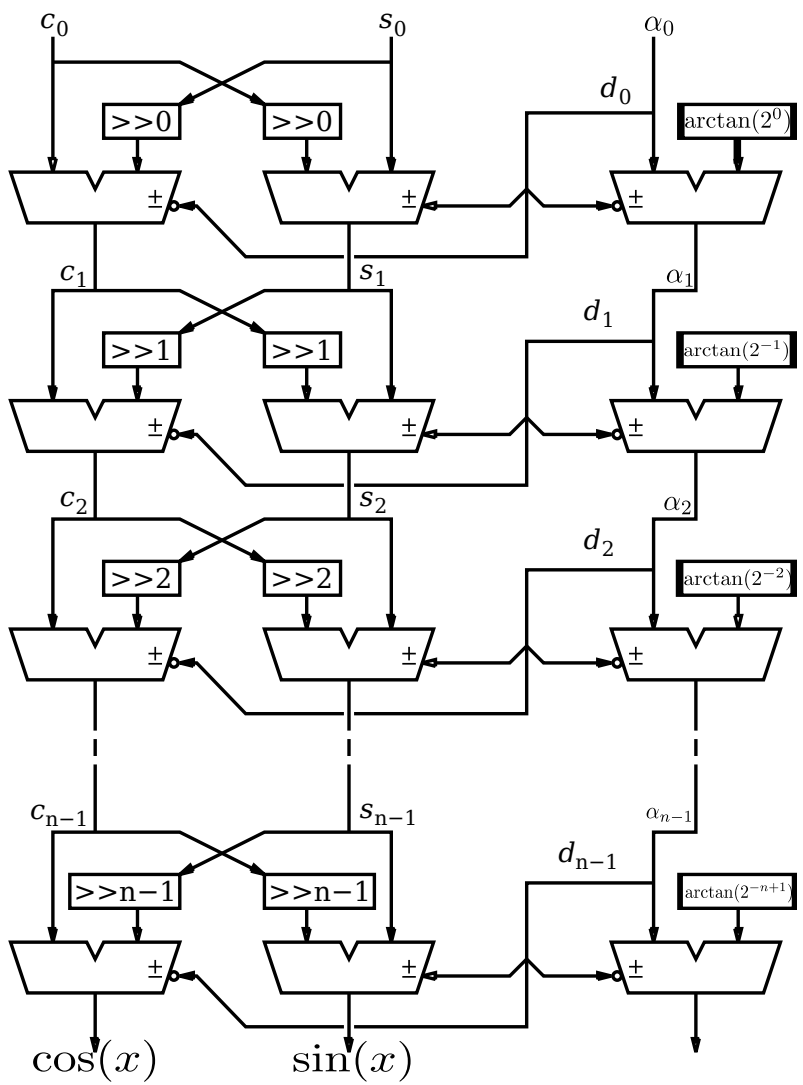
Analysis: rounding errors (ε_{round})

on the α datapath

- correct rounding of $arctan(2^{-i})$
  error bounded by 2^{-w-g-1}
- total error on the α-datapath:
  $nb\_iter \times 2^{-w-g-1}$

on the sin() **and** cos() **datapath**

- for each shift operation, error bounded by 2^{-w-g}
- total error larger than on the α-datapath
- must be smaller than 2^{-w-2}:
  $\varepsilon \times 2^{-w-g} < 2^{-w-2}$
- this gives g
- $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$



CORDIC Error Analysis (2)

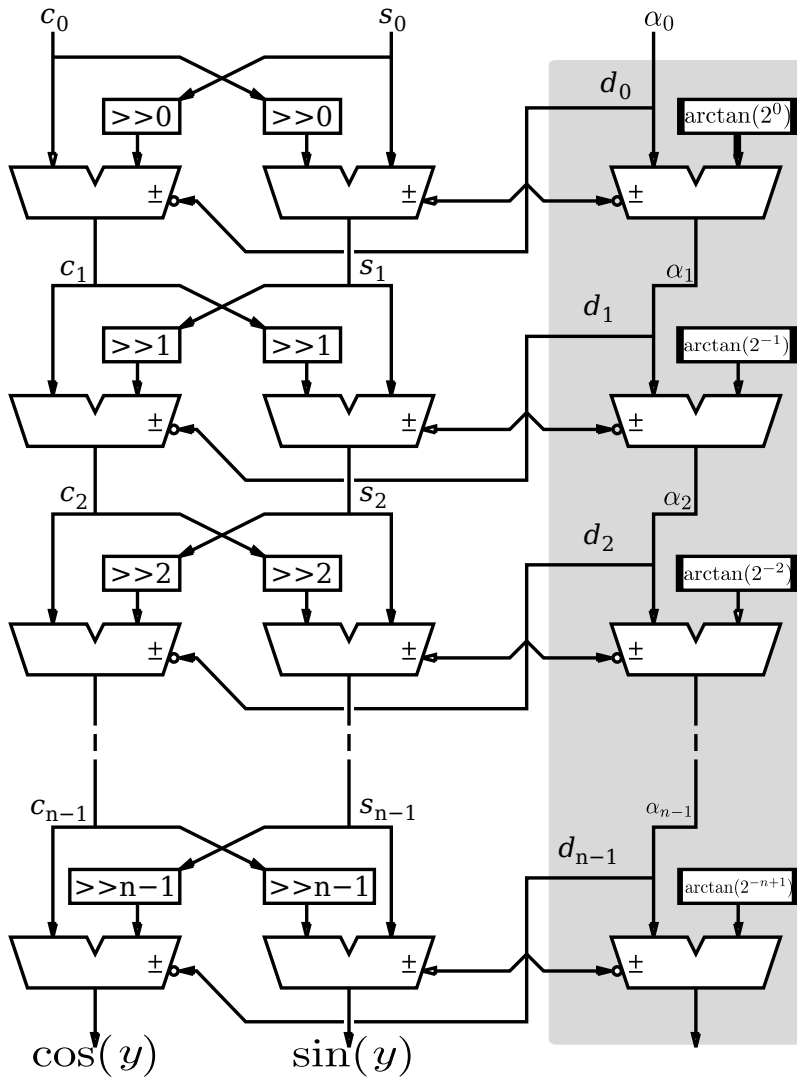
Analysis: rounding errors (ε_{round})

on the α datapath

- correct rounding of $arctan(2^{-i})$
  error bounded by 2^{-w-g-1}
- total error on the α-datapath:
  $nb\_iter \times 2^{-w-g-1}$

on the sin() **and** cos() **datapath**

- for each shift operation, error bounded by 2^{-w-g}
- total error larger than on the α-datapath
- must be smaller than 2^{-w-2}:
  $\varepsilon \times 2^{-w-g} < 2^{-w-2}$
- this gives g
- $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$



CORDIC Error Analysis (2)

Analysis: rounding errors (ε_{round})

on the α datapath

- correct rounding of $arctan(2^{-i})$
  error bounded by 2^{-w-g-1}
- total error on the α-datapath:
  $nb\_iter \times 2^{-w-g-1}$

on the sin() **and** cos() **datapath**

- for each shift operation, error bounded by 2^{-w-g}
- total error larger than on the α-datapath
- must be smaller than 2^{-w-2}:
  $\varepsilon \times 2^{-w-g} < 2^{-w-2}$
- this gives g
- $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$

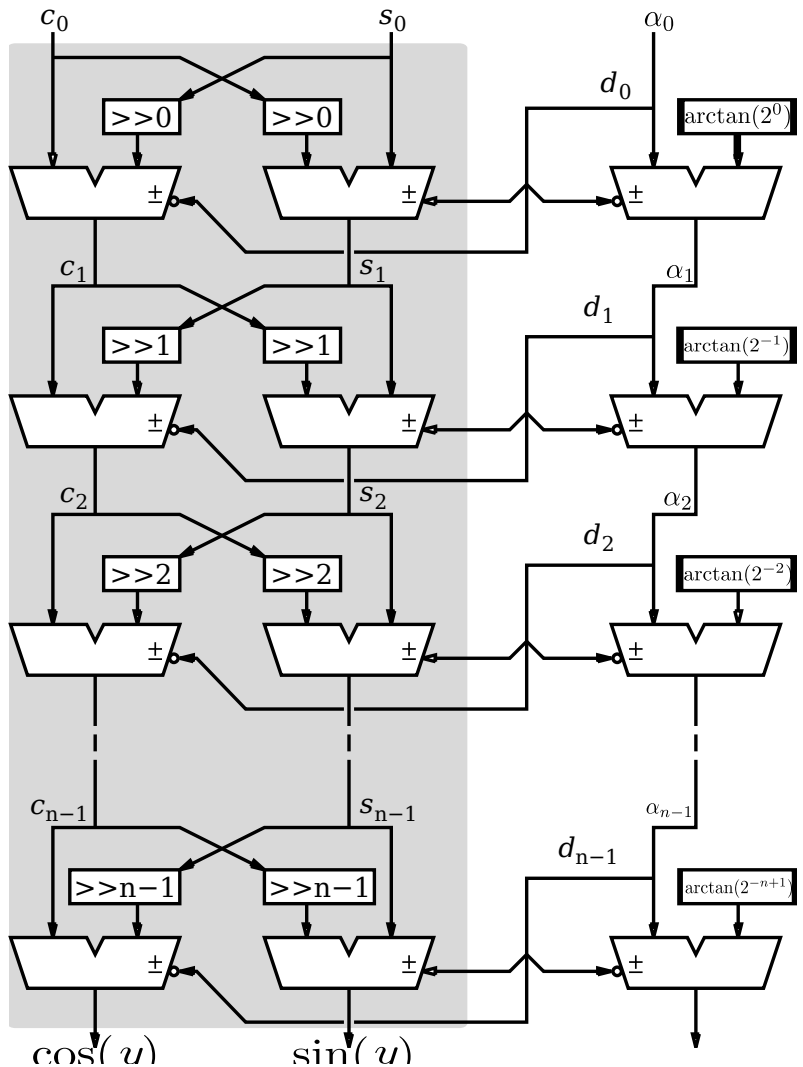
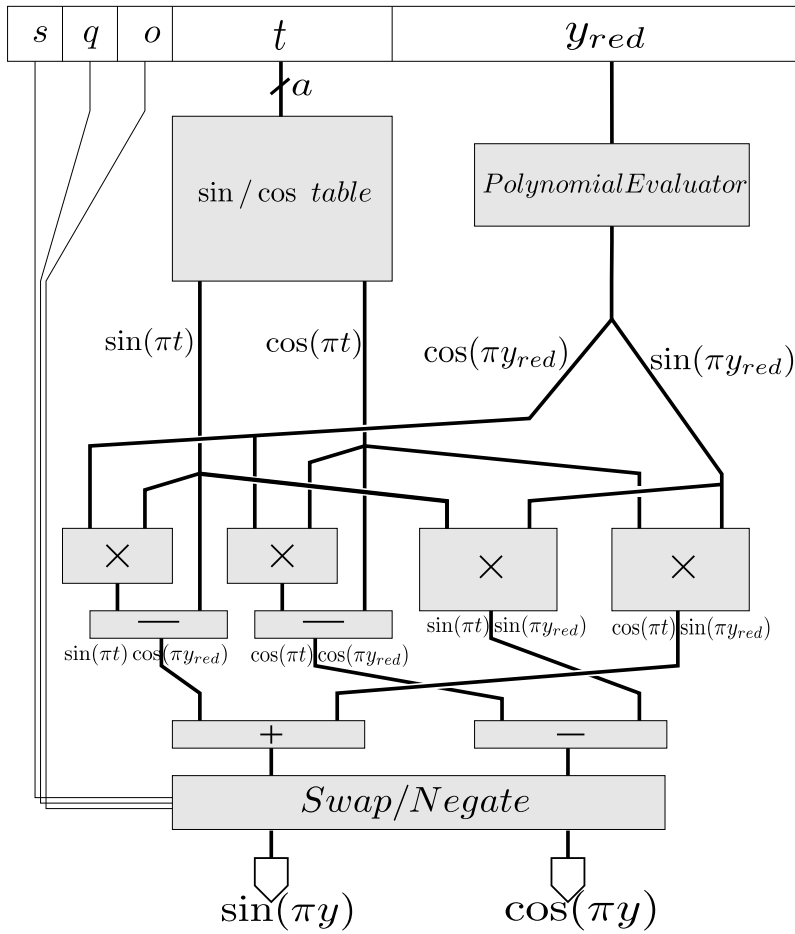


Table- and DSP-based method

Algorithm

- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using



$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method

Algorithm

s	q	o	t	y_{red}

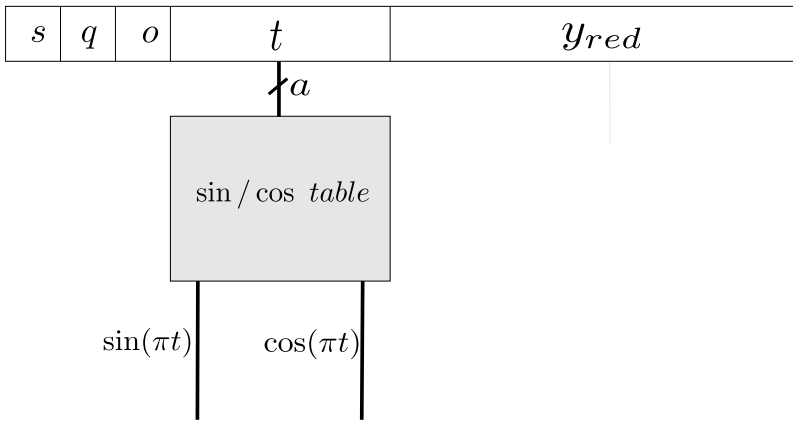
- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using

$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method

Algorithm

- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using

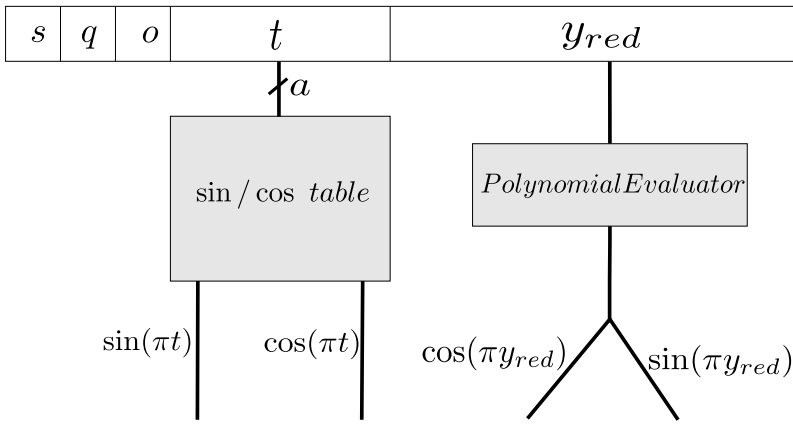


$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method

Algorithm

- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using

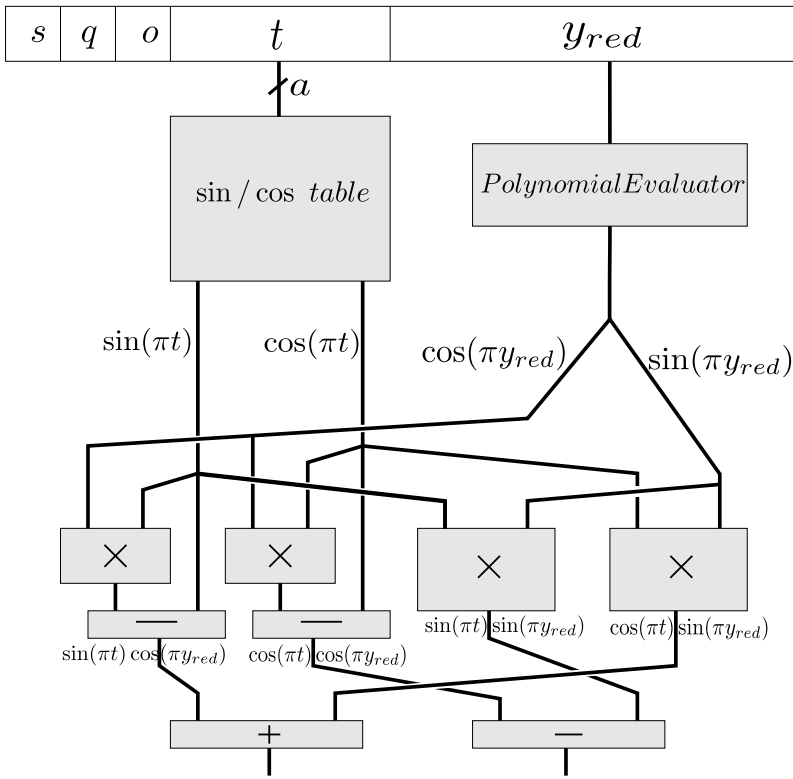


$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method

Algorithm

- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using

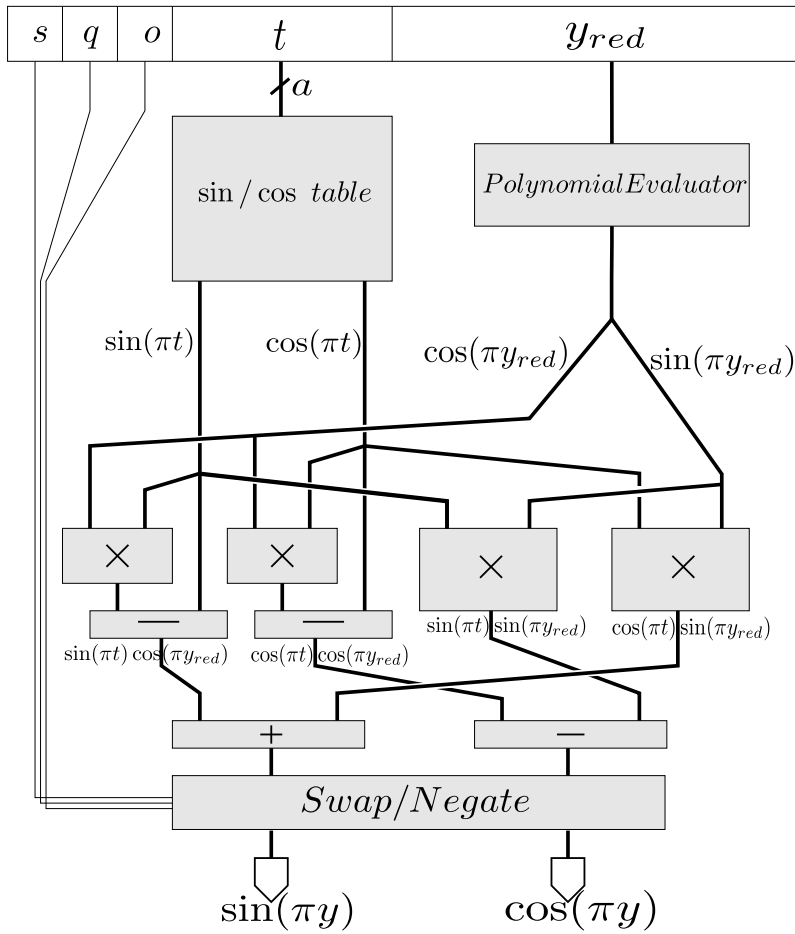


$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method

Algorithm

- angle split:
  y (*the reduced angle*) $= t + y_{red}$
  - t on a bits
  - y_{red} such that $y_{red} < 2^{-(a+2)}$
- store $\sin(\pi t)$ and $\cos(\pi t)$ in tables
- evaluate $\sin(\pi y_{red})$ and $\cos(\pi y_{red})$ using a Taylor polynomial approximation
  - need to compute first $z = y_{red} \times \pi$
  - $\sin(z) \approx z - z^3/6$
  - $\cos(z) \approx 1 - z^2/2$
- reconstruct the values of $\sin(\pi y)$ and $\cos(\pi y)$ using



$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

Table- and DSP-based method: Details

- approximating $y' = \frac{1}{4} - y_{red}$ as $\neg y_{red}$
- choose a such that $\frac{z^4}{24} \leq 2^{-w-g}$
  - so that a degree-3 Taylor polynomial may be used
  - means that $4(a+2) - 2 \geq w + g$
- truncated multiplications
- constant multiplication by π
- $z^2/2$
  - computed using a squarer
- $z^3/6$
  - read from a table for small precisions
  - computed with a dedicated architecture for larger precisions (based on a bit heap and divider by 3, see paper)

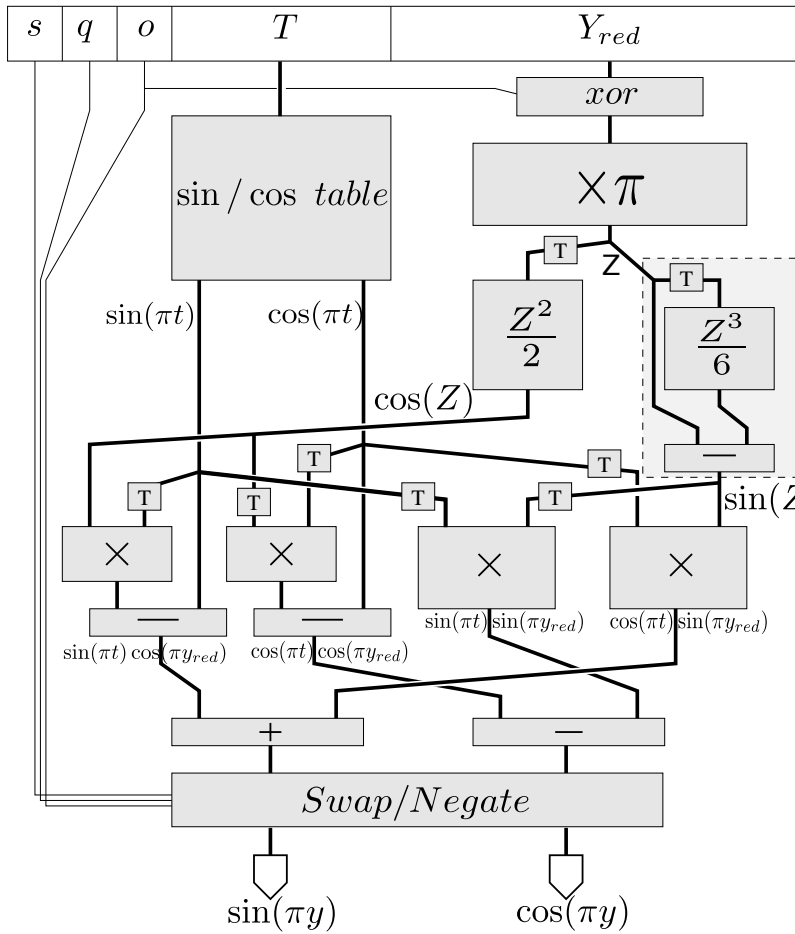
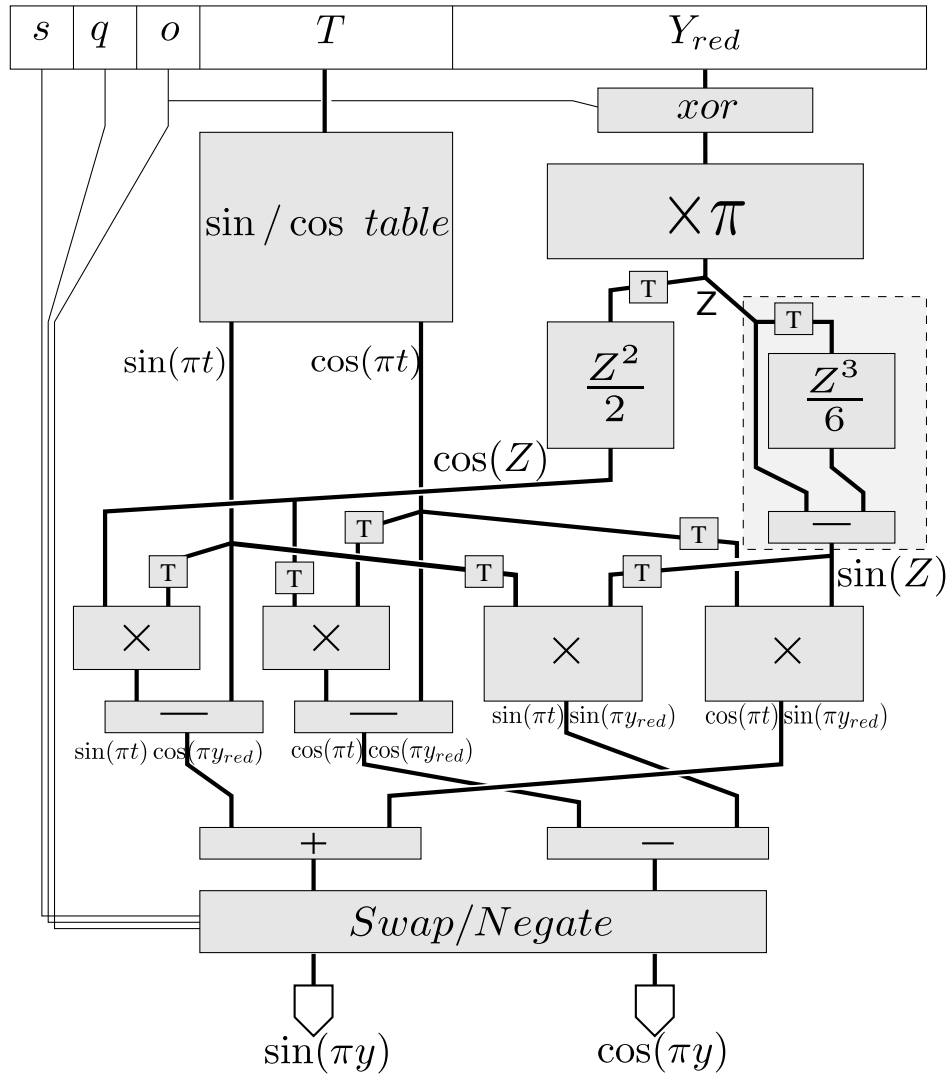


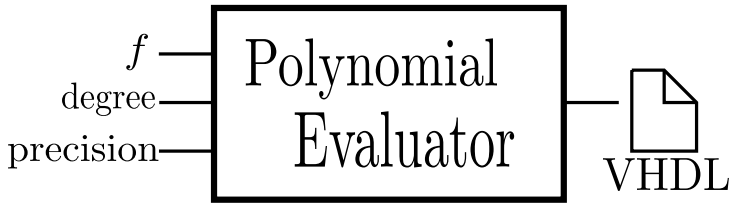
Table- and DSP-based method: Error Analysis

Error Analysis

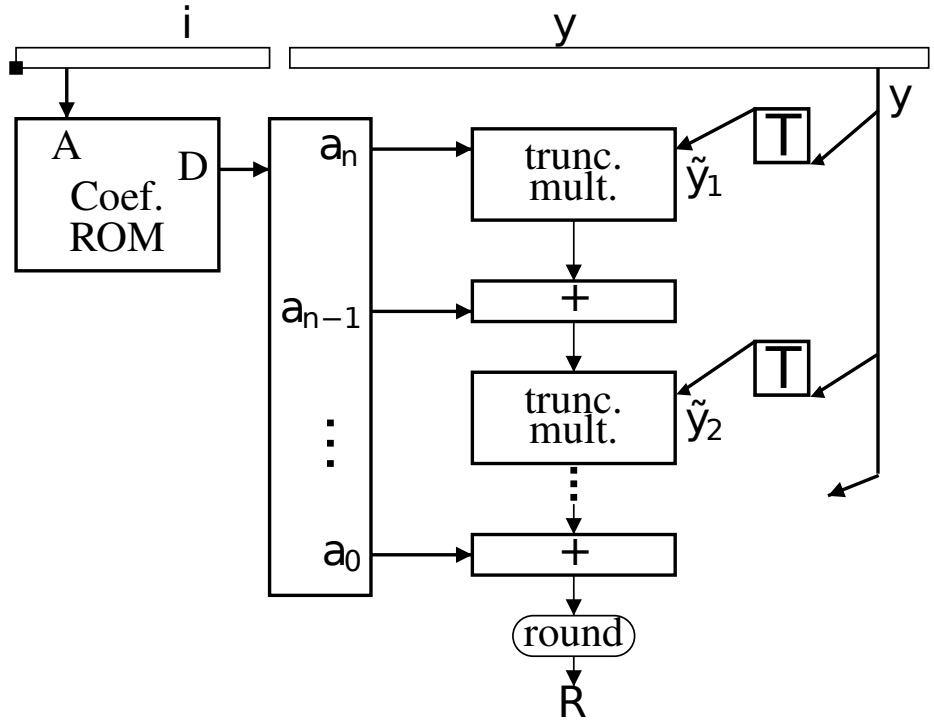
- $\frac{1}{2}$**ulp** lost per table
- 1**ulp** per truncation and truncated multiplier/squarer
- 1**ulp** for computing $\frac{1}{4} - y_{red}$ (as $\neg y_{red}$)
- total of 15**ulp**, independent of the input width
- $\rightarrow$ gives **g=4**



Polynomial-based method



- using existing software (more details in the reference)
- based on polynomial approximation
- computes only one of the functions, depending on an input



Results – 16–bit Precision

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
CORDIC	18	478	969 + 1131	0	0
CORDIC	14	277	776 + 1086	0	0
CORDIC	7	194	418 + 1099	0	0
CORDIC	3	97	262 + 1221	0	0
Red. CORDIC	16	273	657 + 761	0	2
Red. CORDIC	13	368	625 + 719	0	2
Red. CORDIC	7	238	327 + 695	0	2
Red. CORDIC	4	238	106 + 713	0	2
SinAndCos	4	298	107 + 297	0	5
SinAndCos	3	114	168 + 650	0	2
SinOrCos (d=2)	9	251	136 + 183	1	2
SinOrCos (d=2)	5	115.3	87 + 164	1	2

Synthesis Results on Virtex5 FPGA, Using ISE 12.1

Results – Highest Frequency

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 16 bits					
CORDIC	18	478	969 + 1131	0	0
Red. CORDIC	13	368	625 + 719	0	2
SinAndCos	4	298	107 + 297	0	5
SinOrCos (d=2)	9	251	136 + 183	1	2
precision = 24 bits					
CORDIC	28	439.9	1996 + 2144	0	0
Red. CORDIC	20	273.4	1401 + 1446	0	4
SinAndCos	5	262	197 + 441	3	7
SinOrCos (d=2)	9	251	202 + 279	2	2
precision = 32 bits					
CORDIC	37	403.5	3495 + 3591	0	0
Red. CORDIC	24	256.8	2160 + 2234	0	4
SinAndCos	10	253	535 + 789	3	9
SinOrCos (d=3)	14	251	444 + 536	4	5
precision = 40 bits					
CORDIC	45	375	5070 + 5289	0	0
Red. CORDIC	37	252	3695 + 3768	0	8
SinAndCos (bit heap)	11	266	895 + 1644	3	12
SinAndCos (table $z^3/6$)	8	232	500 + 949	4	12
SinOrCos (d=3)	15	251	628 +725	4	8
precision = 48 bits					
SinAndCos (bit heap)	13	232	1322 + 2369	12	17
SinOrCos	15	250	734 + 879	17	10

Results – Options for $\frac{Z^3}{6}$

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 40 bits					
CORDIC	45	375	5070 + 5289	0	0
CORDIC	25	149	2948 + 5245	0	0
Red. CORDIC	37	252	3695 + 3768	0	8
Red. CORDIC	9	123	931 + 3339	0	8
SinAndCos (bit heap)	11	266	895 + 1644	3	12
SinAndCos (table $z^3/6$)	8	232	500 + 949	4	12
SinAndCos (bit heap)	4	154	612 + 2826	0	12
SinAndCos (table $z^3/6$)	4	156	395 + 2268	2	12
SinOrCos (d=3)	15	251	628 +725	4	8
SinOrCos (d=3)	9	132	376 +675	4	8
precision = 48 bits					
SinAndCos (bit heap)	13	232	1322 + 2369	12	17
SinAndCos (bit heap)	6	132	972 + 2133	12	17
SinOrCos	15	250	734 + 879	17	10
SinOrCos	9	124	431 + 823	17	10

Conclusions

- A wide range of open-source accurate implementations
  - CORDIC implementation on par with vendor-provided solutions
  - some tuning still needed on DSP-based methods
- SinAndCos method overall best
- Little point in using unrolled CORDIC for FPGAs

Approach	latency	area
CORDIC 16 bits	30.3 ns	1034 LUTs
SinAndCos 16 bits	15.0 ns	1211 LUTs
CORDIC 24 bits	44.6 ns	2079 LUTs
SinAndCos 24 bits	17.0 ns	2183 LUTs
CORDIC 32 bits	62.1 ns	3513 LUTs
SinAndCos 32 bits	19.4 ns	3539 LUTs

Synthesis results for logic-only implementations

What is the cost of computing w bits of sine/cosine?

The universal bit heap

Introduction and motivation

So much VHDL to write, so few slaves to write it

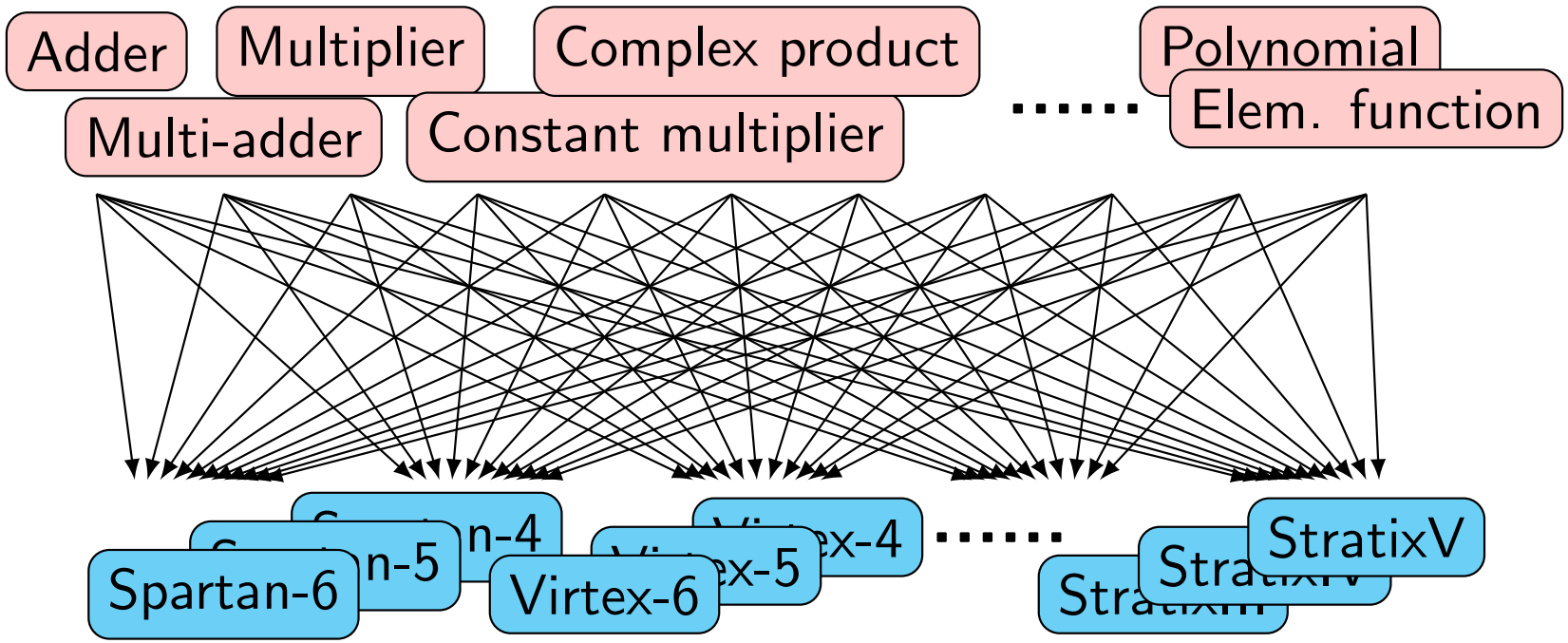
FPGA arithmetic the way it should be:

- An infinite number of application-specific operators
- Each heavily parameterized (bit-size, performance, etc)
- Portable to any FPGA, and even ASIC

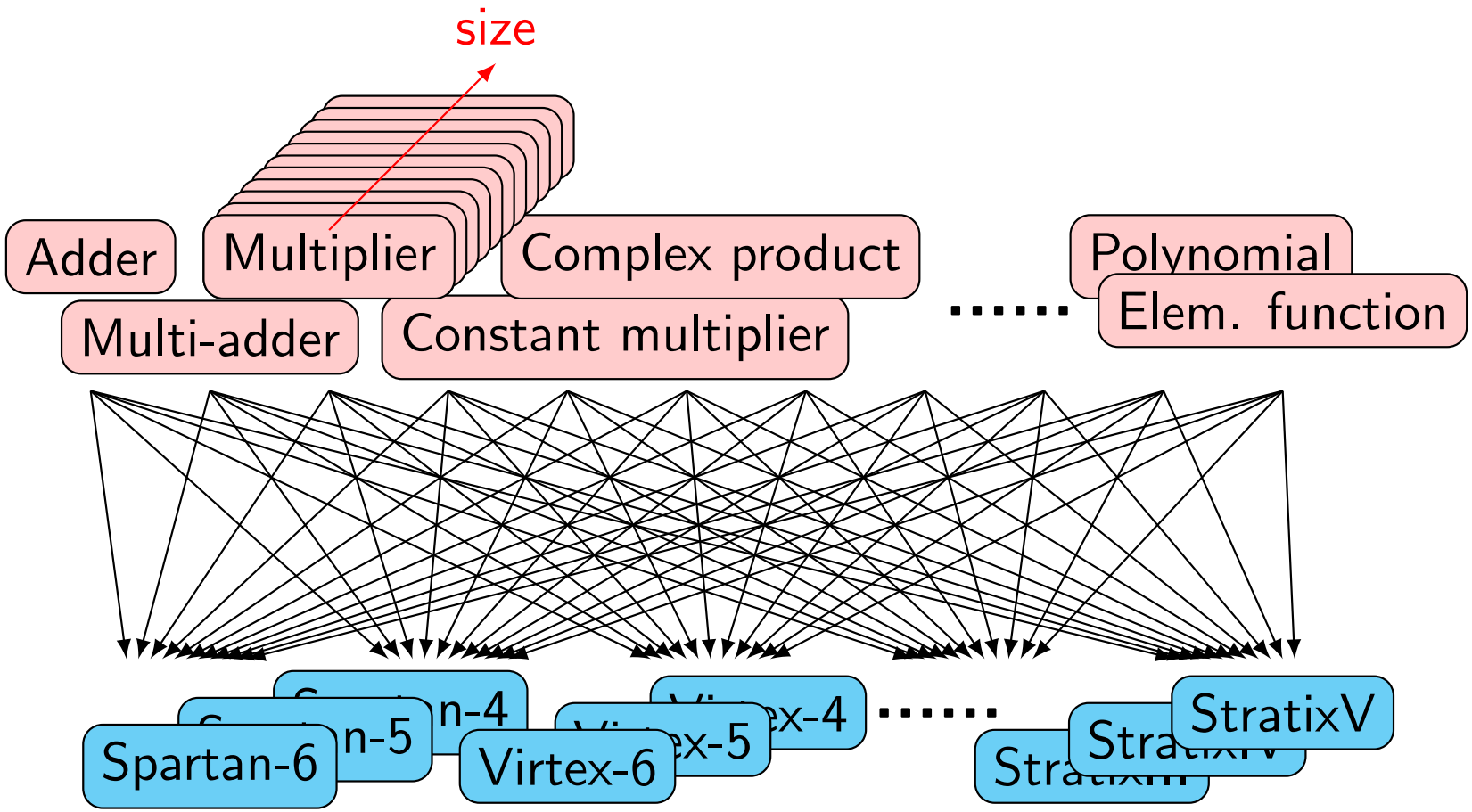
How to ensure **performance** across all this range?

- object-oriented abstraction of vendor-specific features
- ... not enough

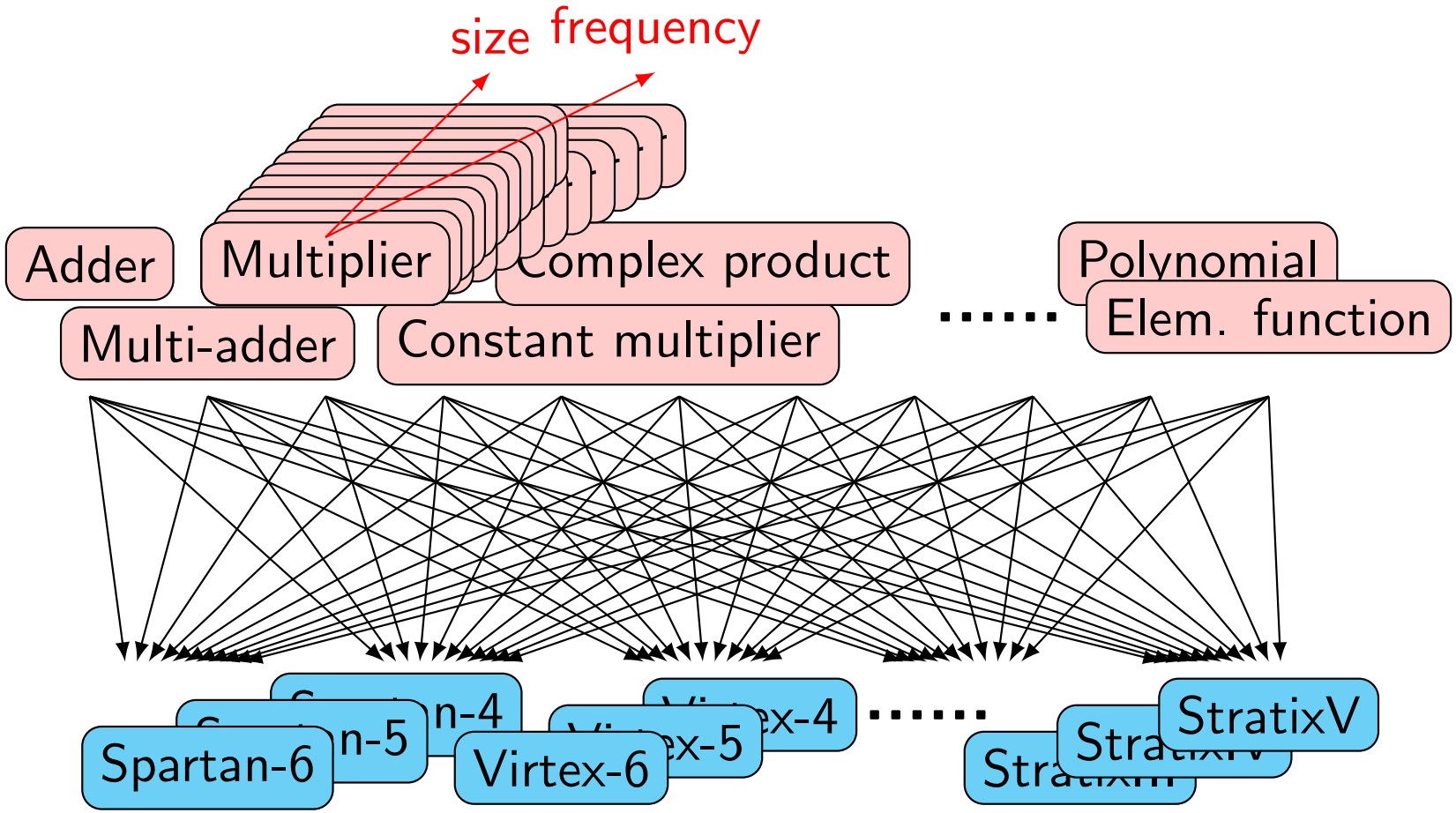
Portable versus optimized



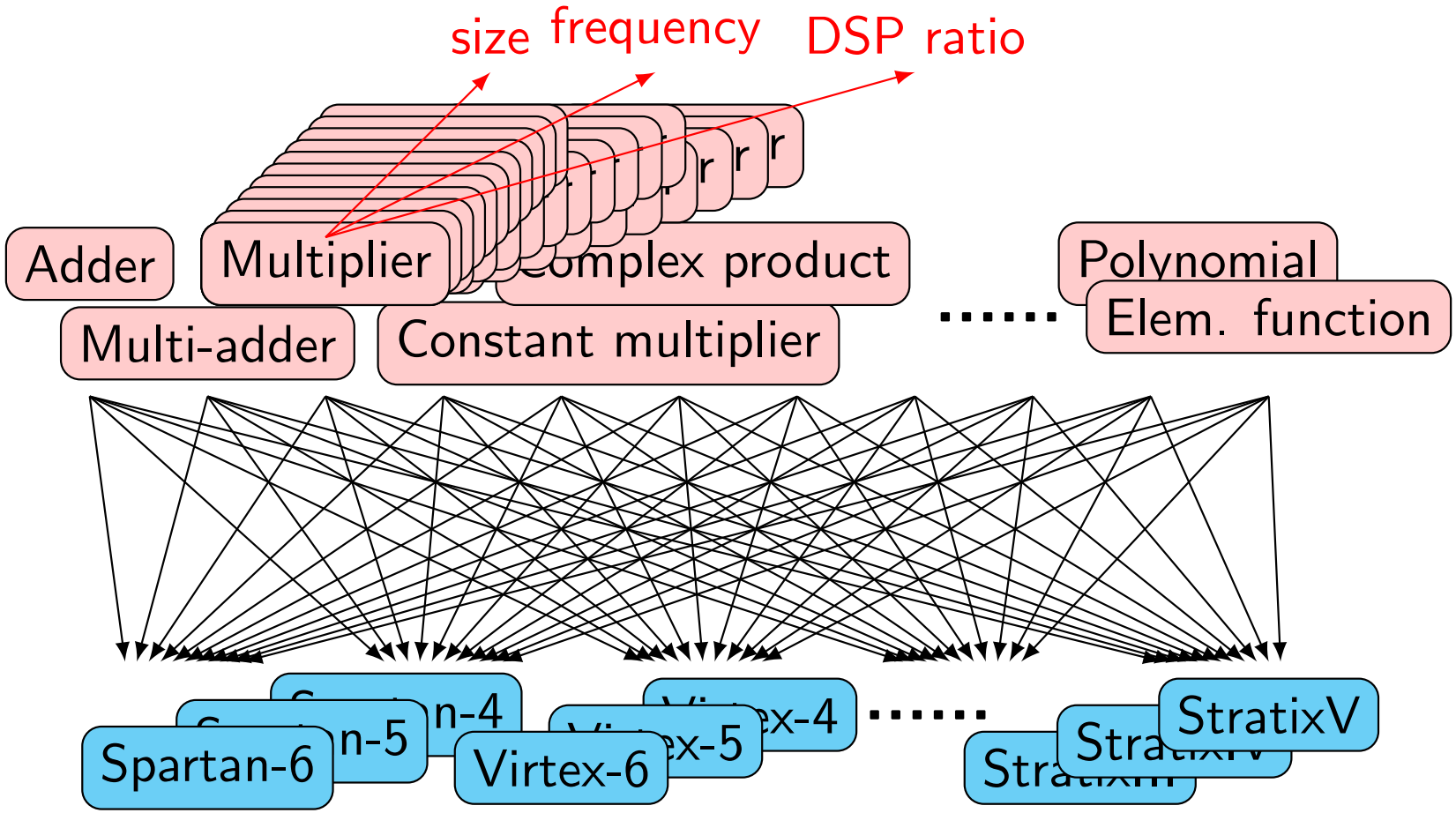
Portable versus optimized



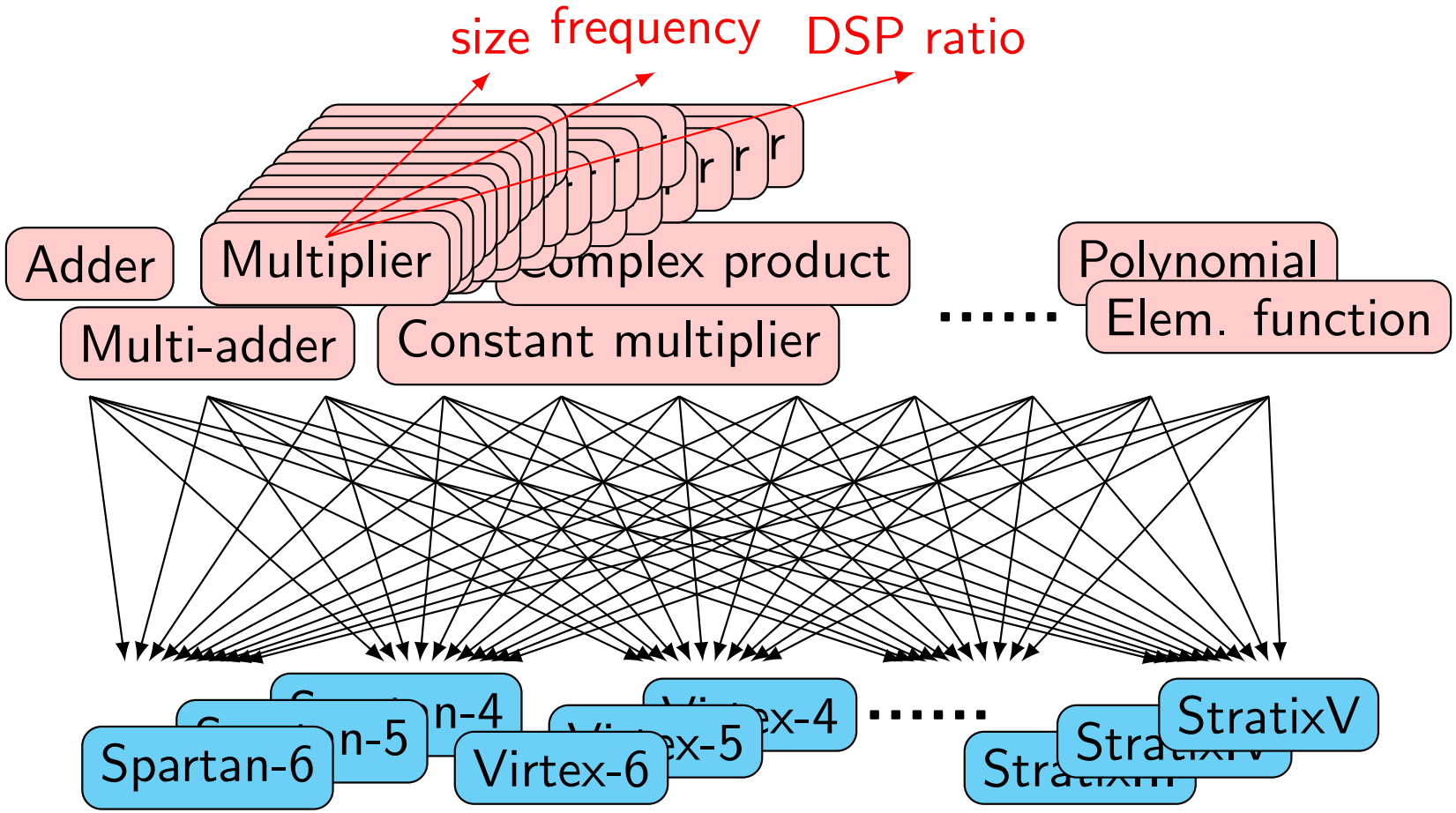
Portable versus optimized



Portable versus optimized

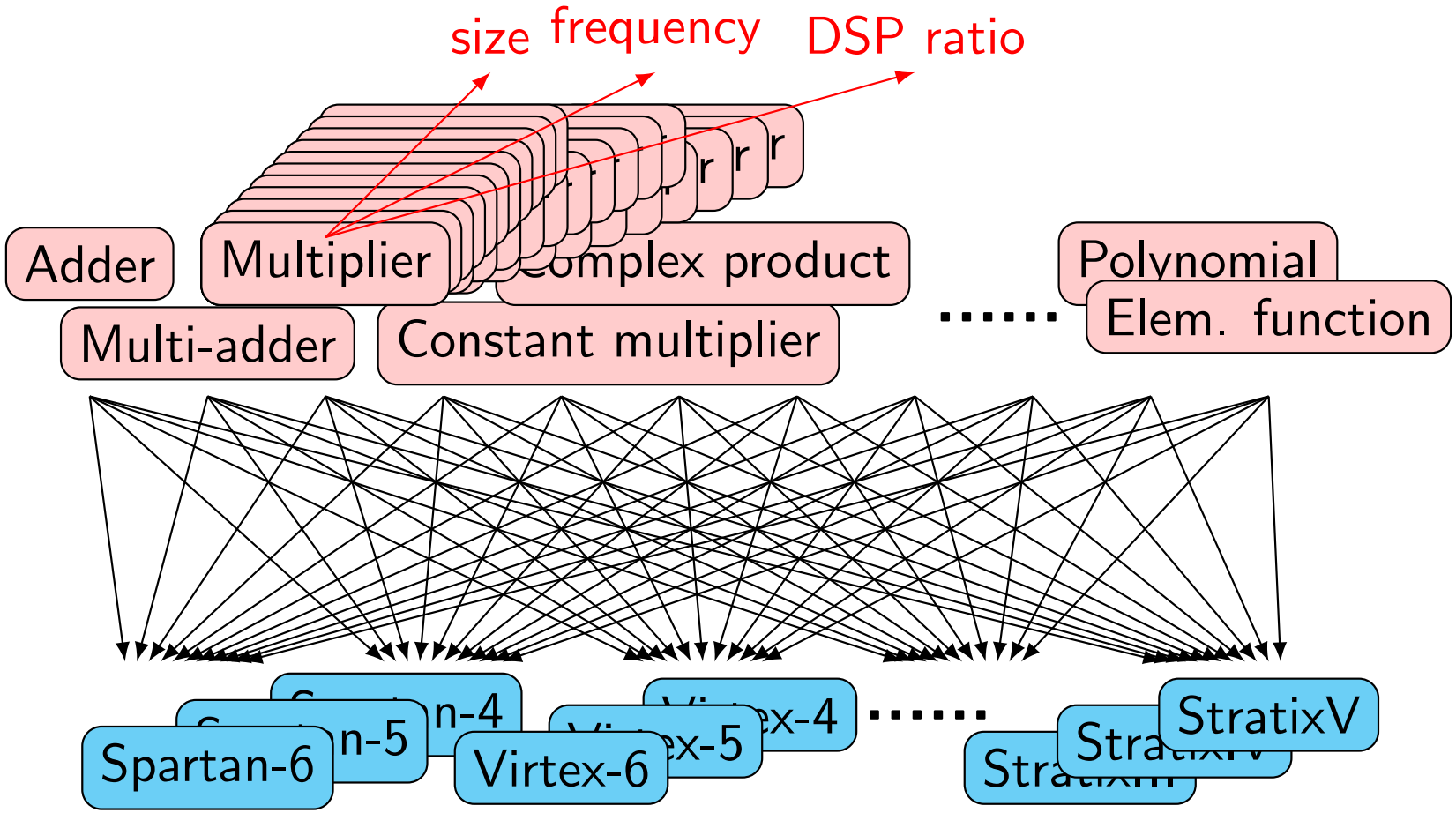


Portable versus optimized



I know how to optimize by hand each operator on each target

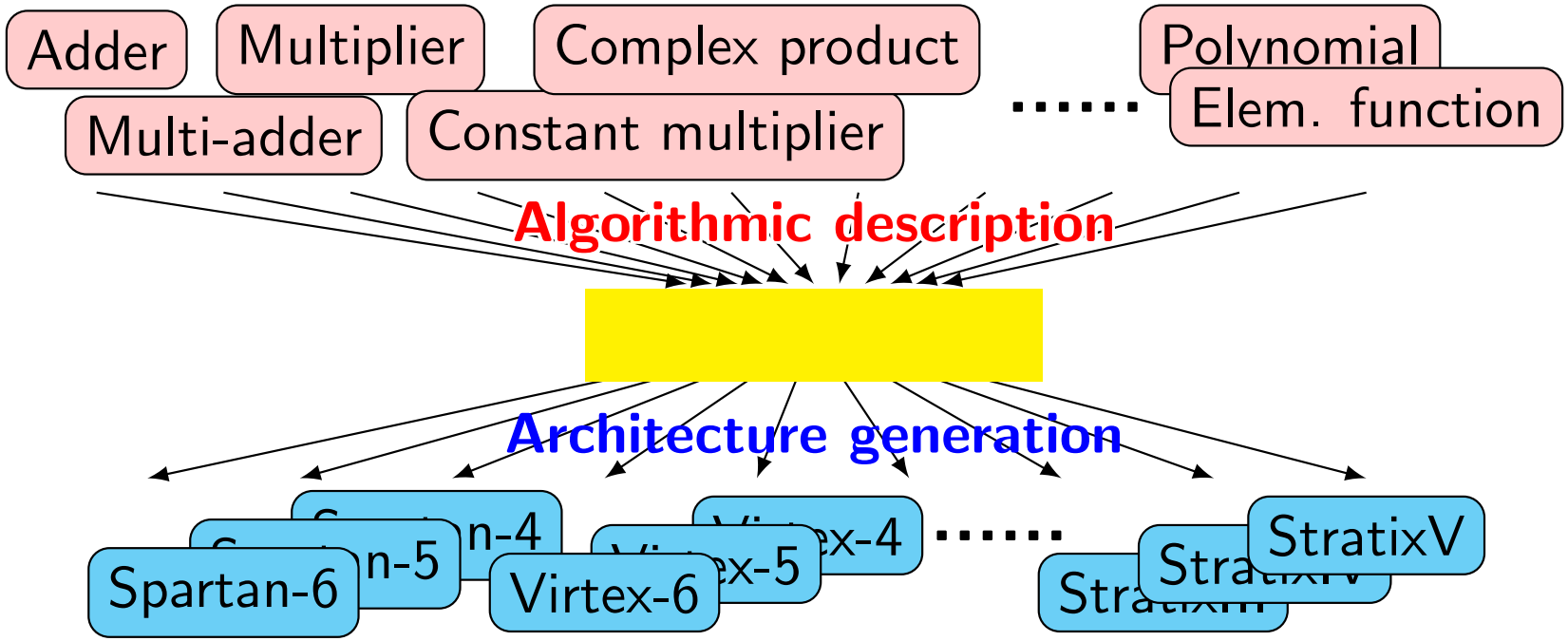
Portable versus optimized



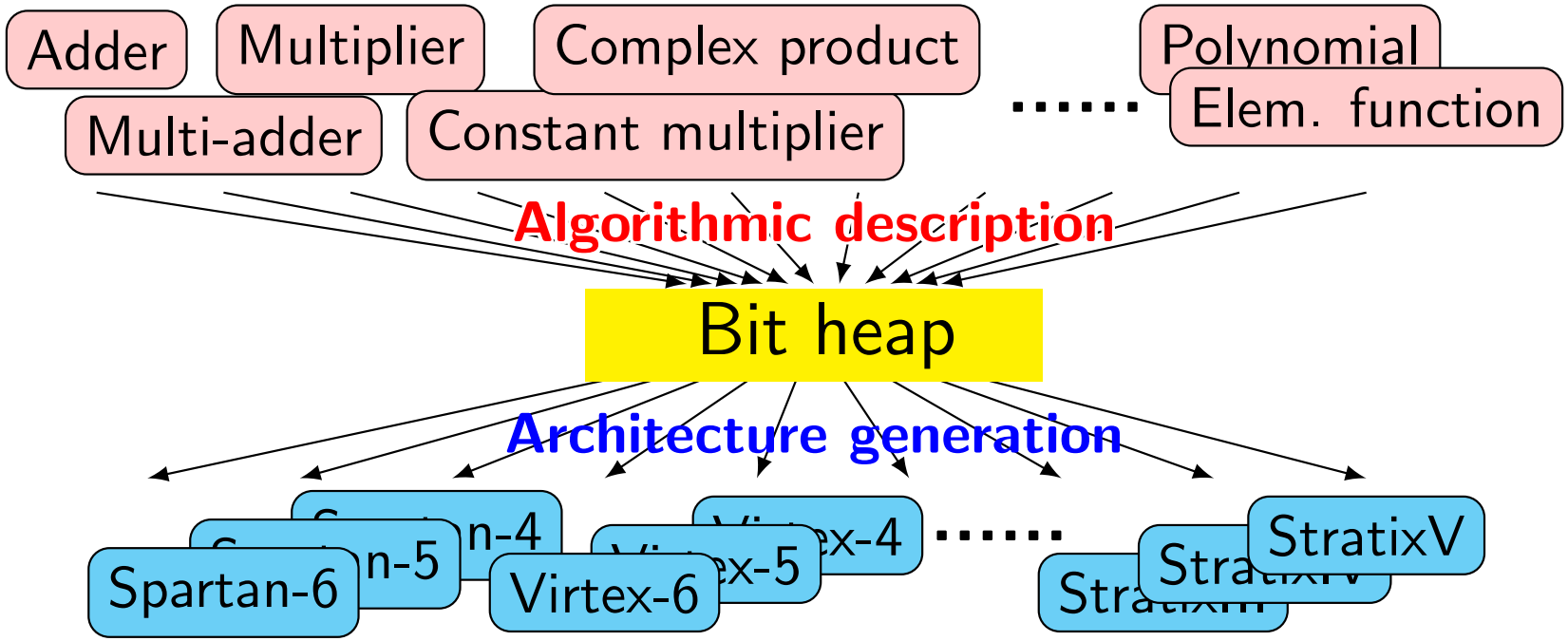
I know how to optimize by hand each operator on each target

... But I don't want to do it.

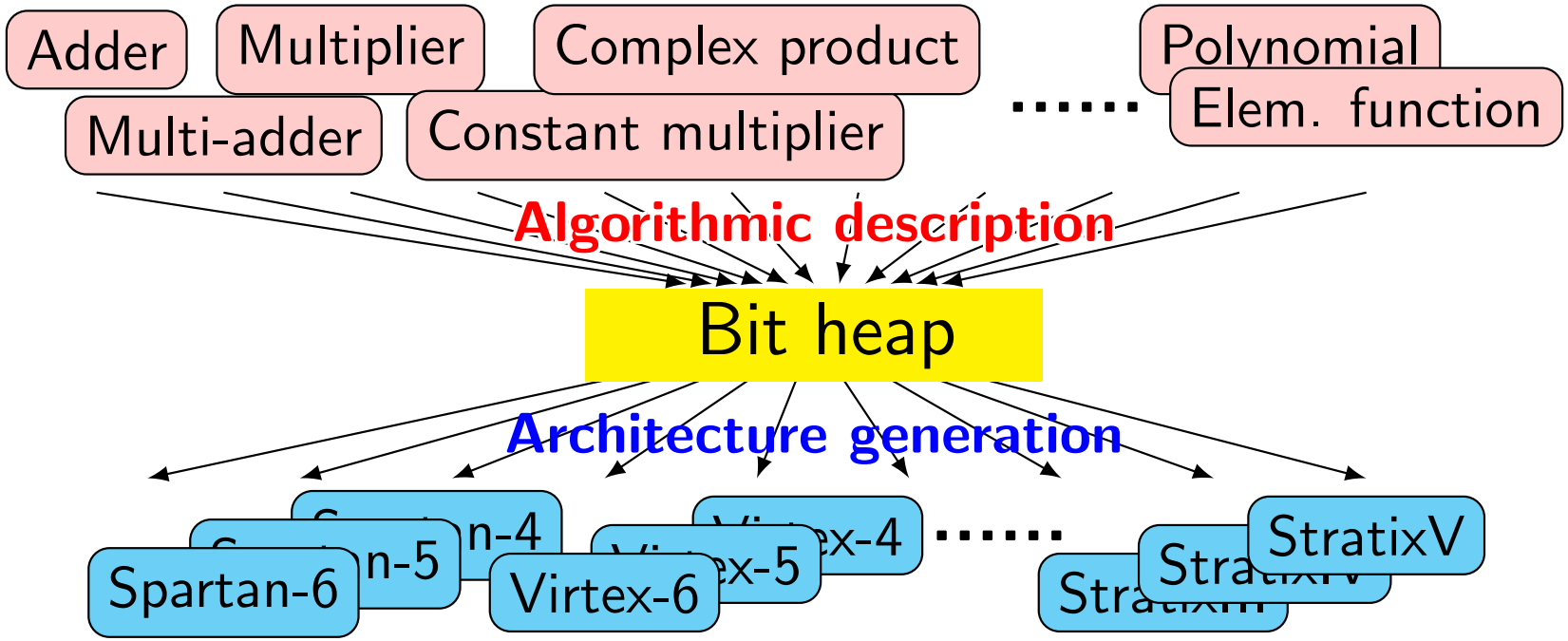
Reducing the combinatorics



Reducing the combinatorics



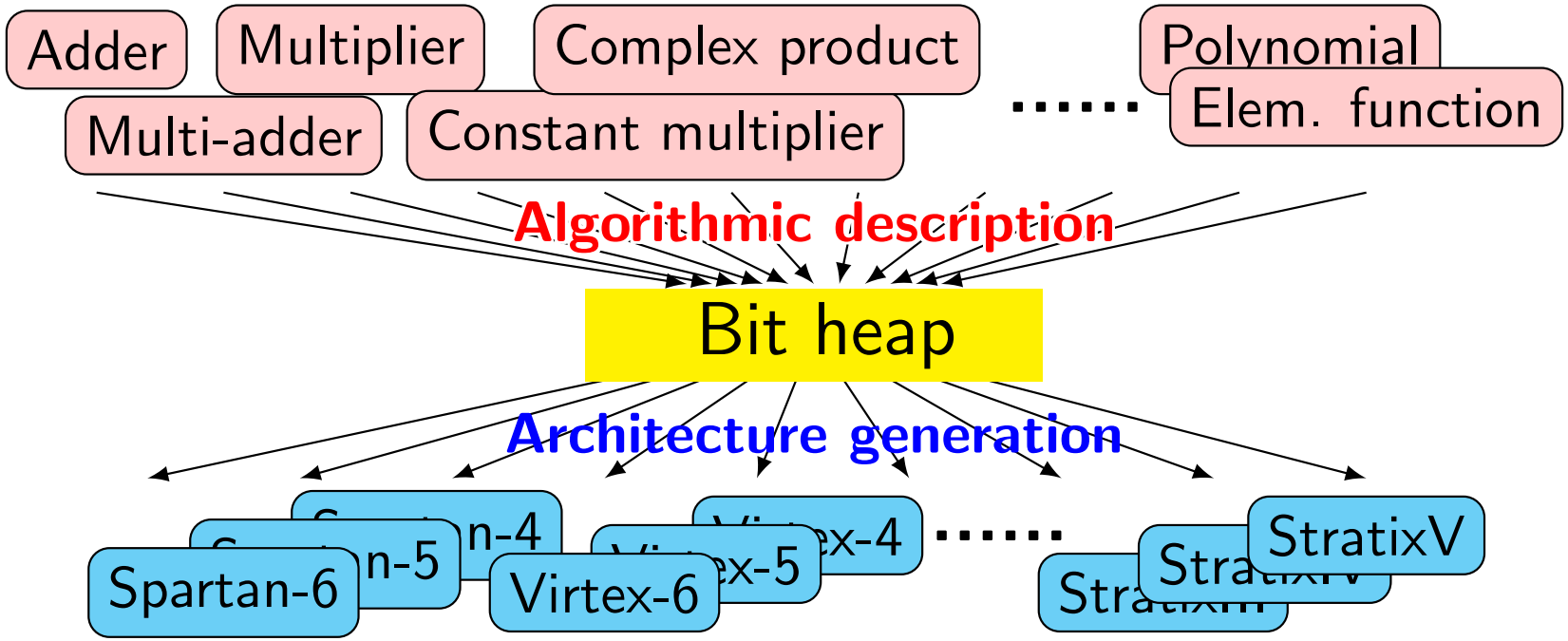
Reducing the combinatorics



What is a bit heap?

- A **data-structure**
  - capturing bit-level descriptions of a wide class of operators

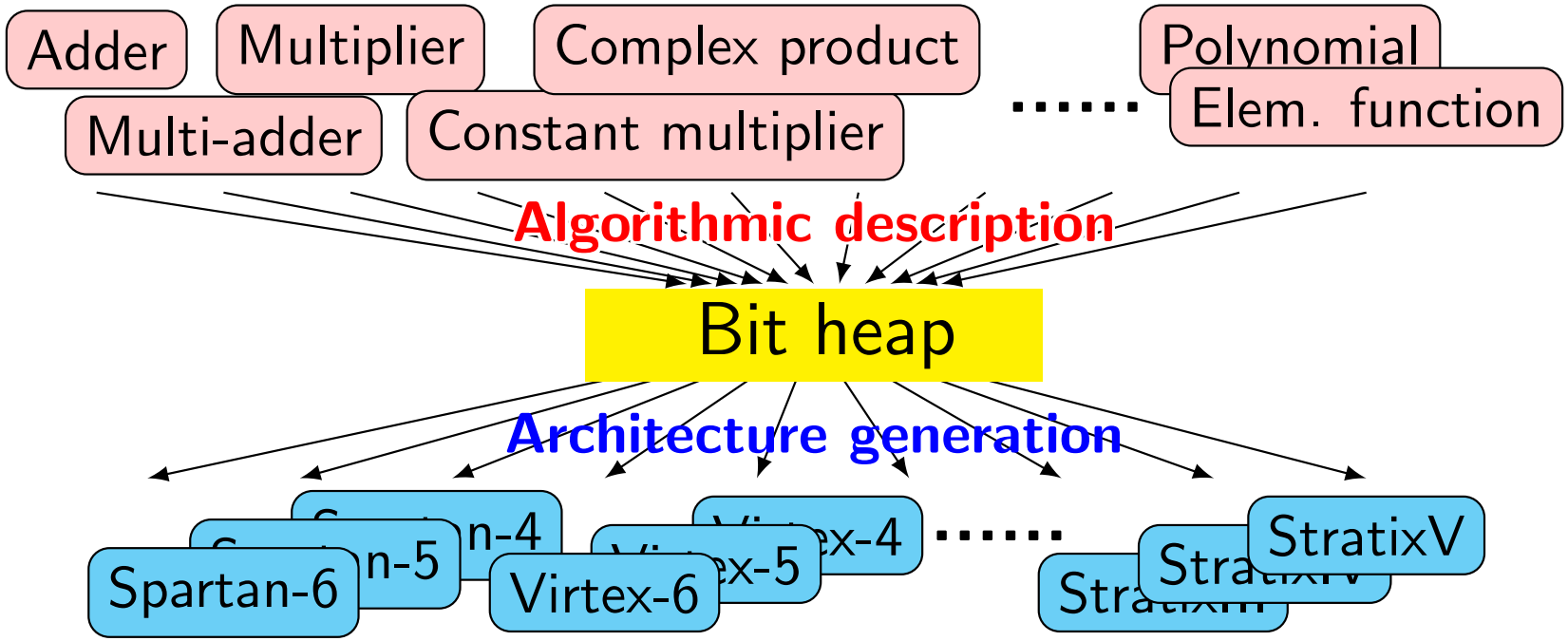
Reducing the combinatorics



What is a bit heap?

- A **data-structure**
  - capturing bit-level descriptions of a wide class of operators
  - exposing bit-level parallelism and optimization opportunities

Reducing the combinatorics

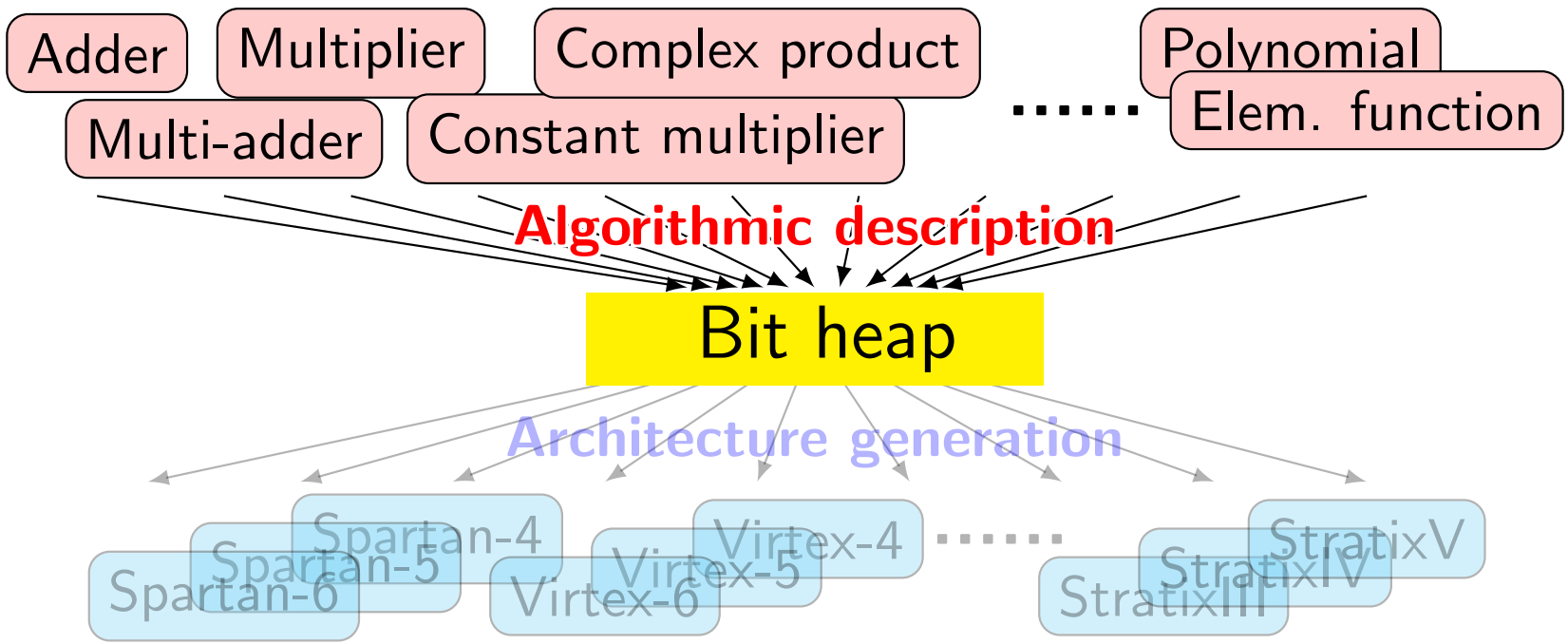


What is a bit heap?

- A **data-structure**
  - capturing bit-level descriptions of a wide class of operators
  - exposing bit-level parallelism and optimization opportunities
- An associated **architecture generator**

  which can be optimized for each target

Operations as bit heaps



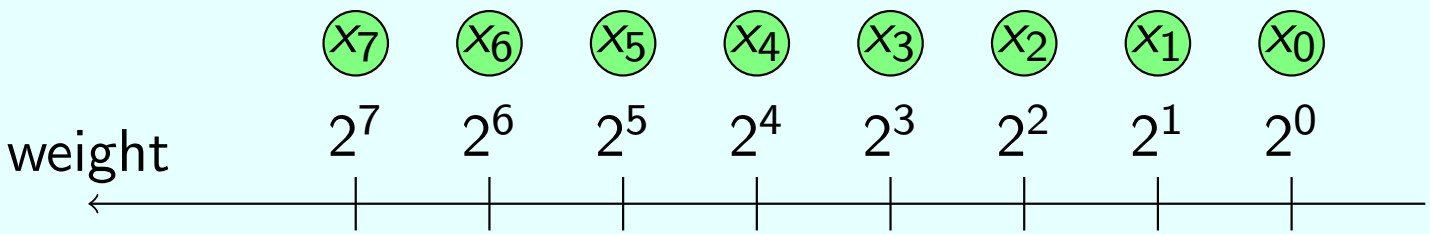
Weighted bits

- Integers or real numbers represented in binary fixed-point

$$X = \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i$$

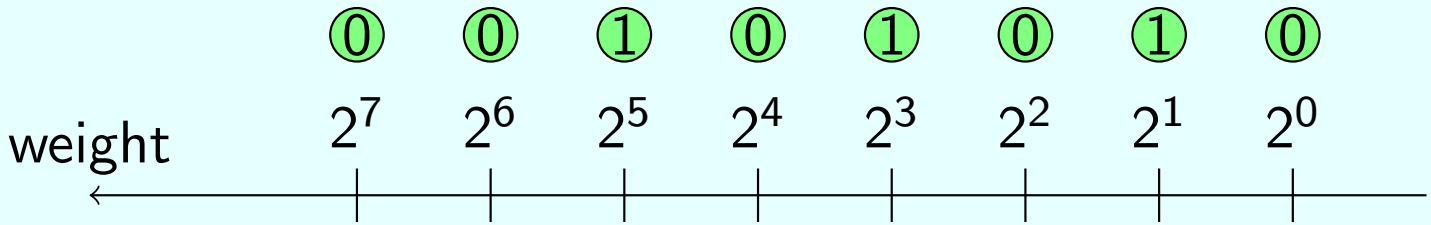
- 2^i : "weight" $\Longrightarrow$ X "sum of weighted bits"

Representation as a **dot diagrams**

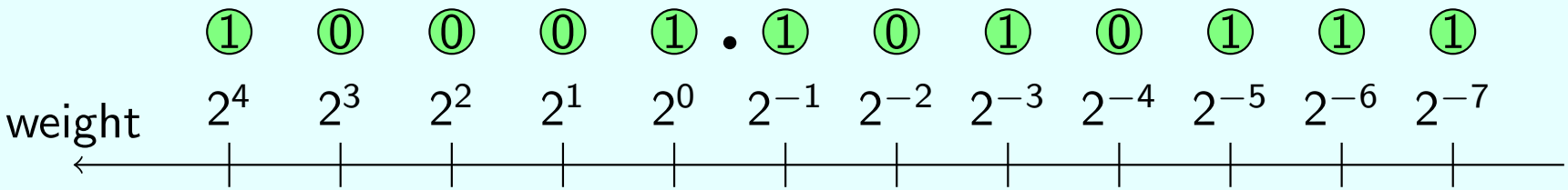


Integer or fixed-point

Example: 42 written in binary



Example: 17.42 written in binary

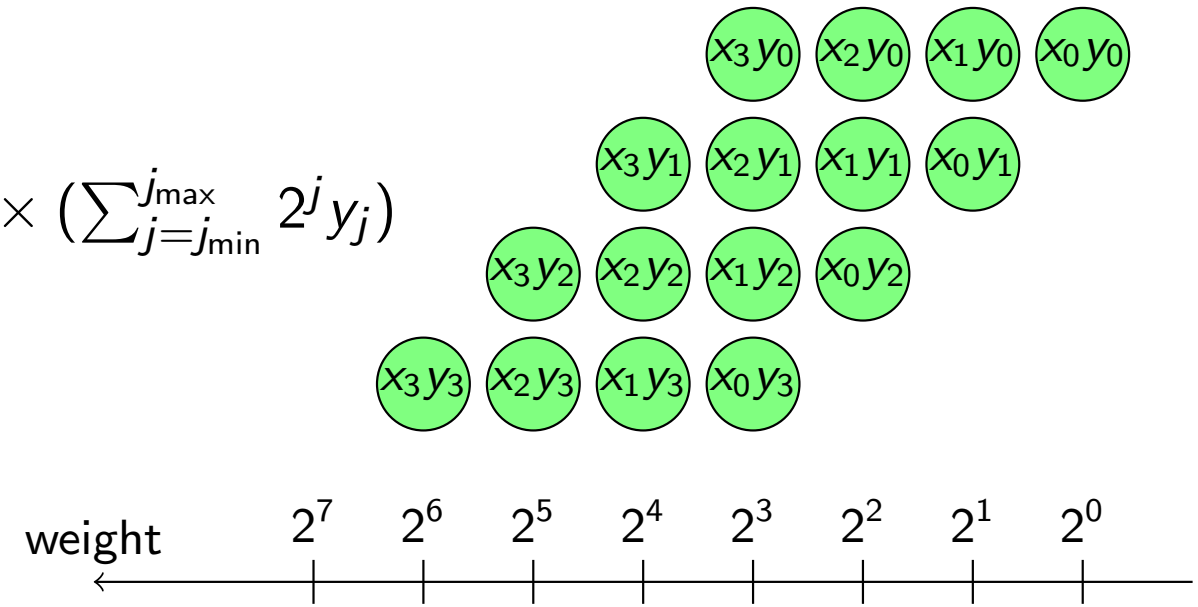


The historical bit heap

$$
\begin{aligned}
XY &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right) \\
&= \sum_{i,j} 2^{i+j} x_i y_j
\end{aligned}
$$

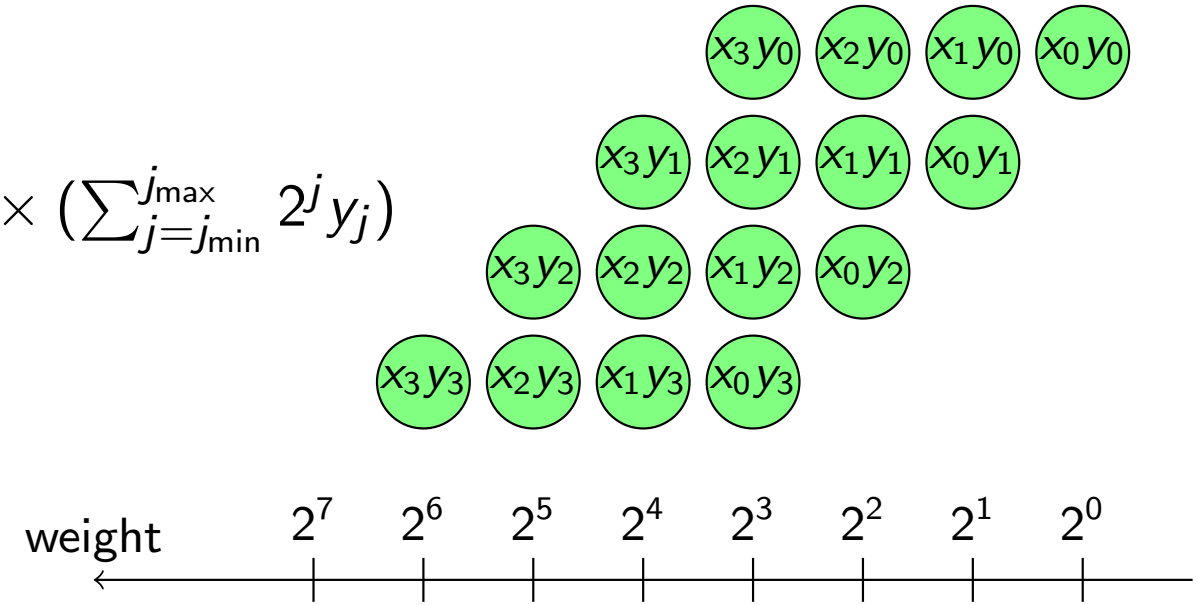
The historical bit heap

$$
\begin{aligned}
XY &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right) \\
&= \sum_{i,j} 2^{i+j} x_i y_j
\end{aligned}
$$



The historical bit heap

$$\begin{aligned} XY &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right) \\ &= \sum_{i,j} 2^{i+j} x_i y_j \end{aligned}$$



A multiplier is an architecture that computes this sum.

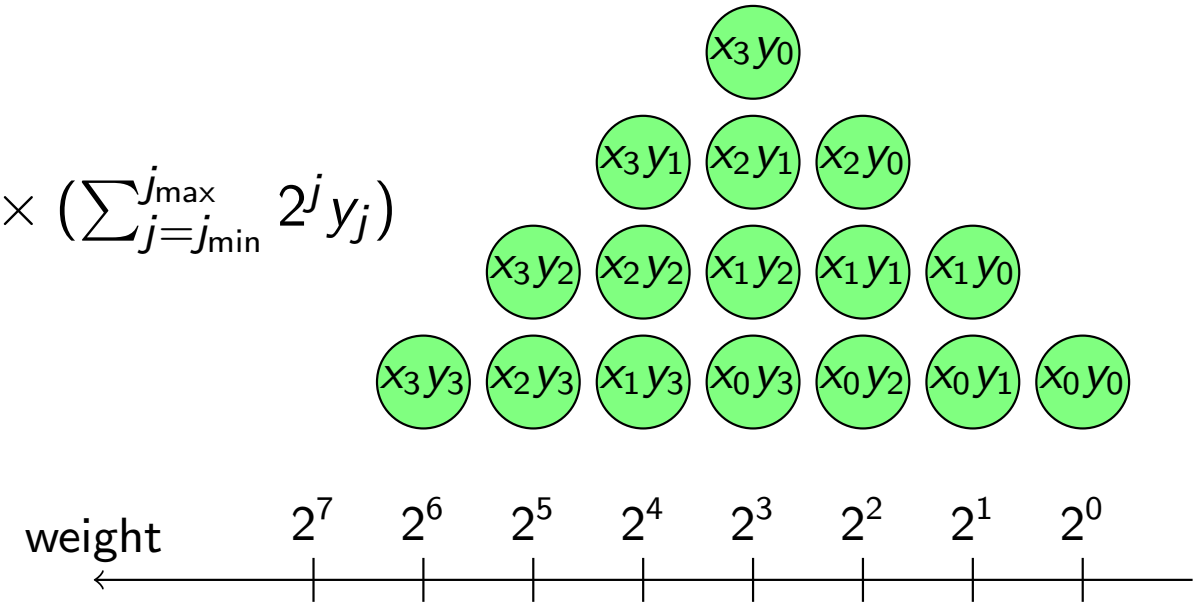
Historical motivation for bit heaps

$\sum_{i,j} 2^{i+j} x_i y_j$ **expresses the bit-level parallelism of the problem**

The historical bit heap

$$XY = \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right)$$

$$= \sum_{i,j} 2^{i+j} x_i y_j$$



A multiplier is an architecture that computes this sum.

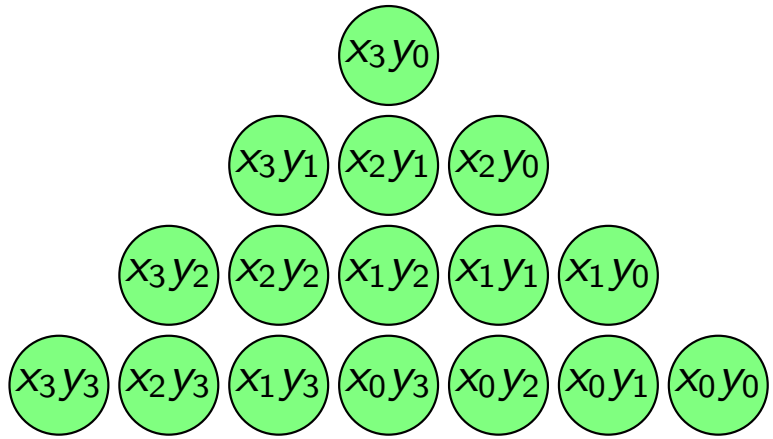
Historical motivation for bit heaps

$\sum_{i,j} 2^{i+j} x_i y_j$ **expresses the bit-level parallelism of the problem**

(freedom thanks to addition associativity and commutativity)

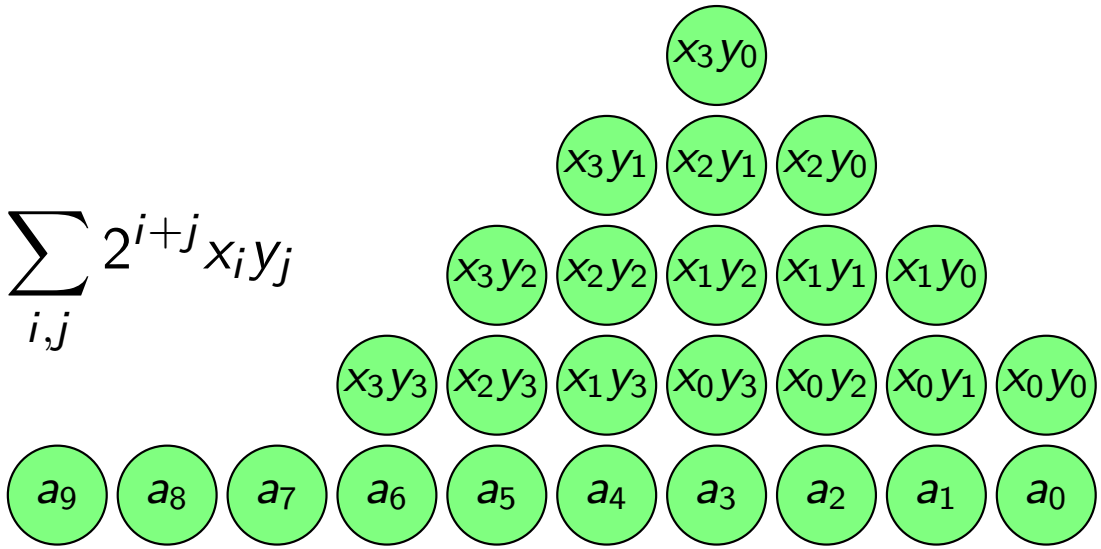
Beyond product

$$XY = \sum_{i,j} 2^{i+j} x_i y_j$$



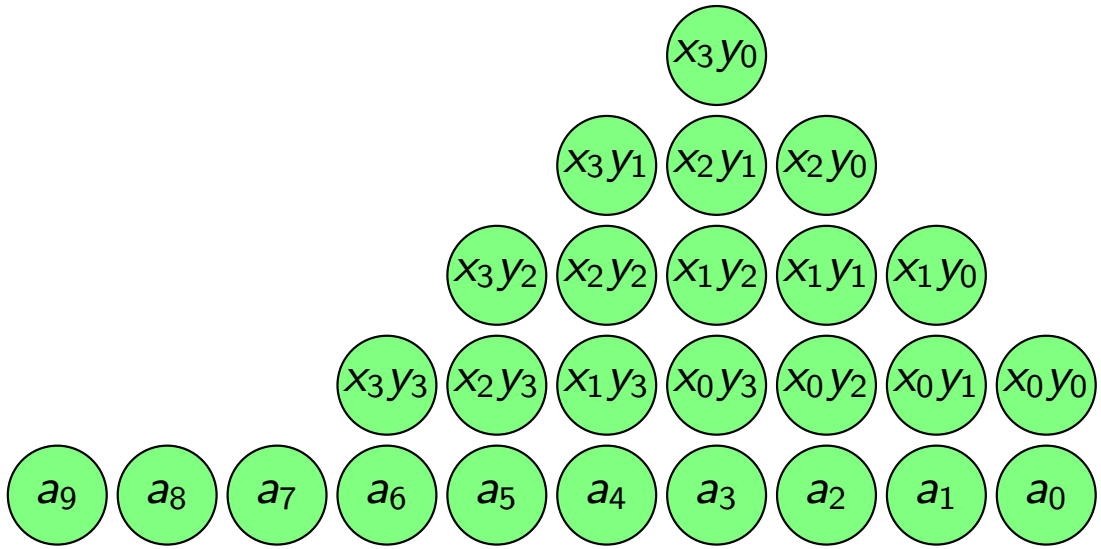
Beyond product

$$A + XY = \sum_i 2^i a_i + \sum_{i,j} 2^{i+j} x_i y_j$$



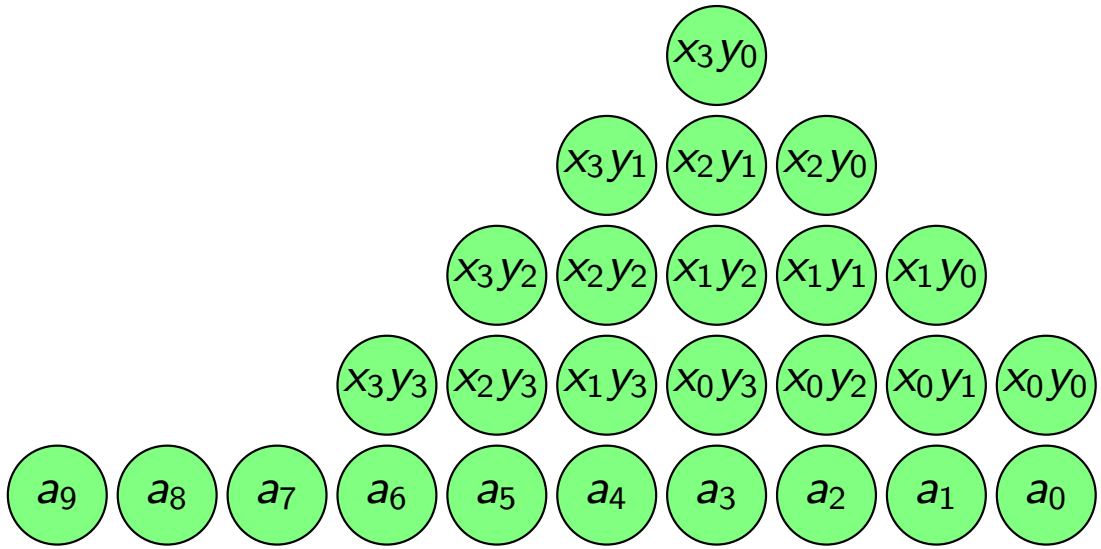
Beyond product

$$A + XY = \sum_{w,h} 2^w b_{w,h}$$



Beyond product

$$A + XY = \sum_{w,h} 2^w b_{w,h}$$



When generating an architecture

consider **only one big sum of weighted bits**

- get rid of artificial sequentiality
  inside operators, and between operators
- focus on true timing information
  e.g. critical path delay of each weighted bit
- A global optimization instead of several local ones

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Any polynomial of multiple variables is a bit heap

... where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Any polynomial of multiple variables is a bit heap

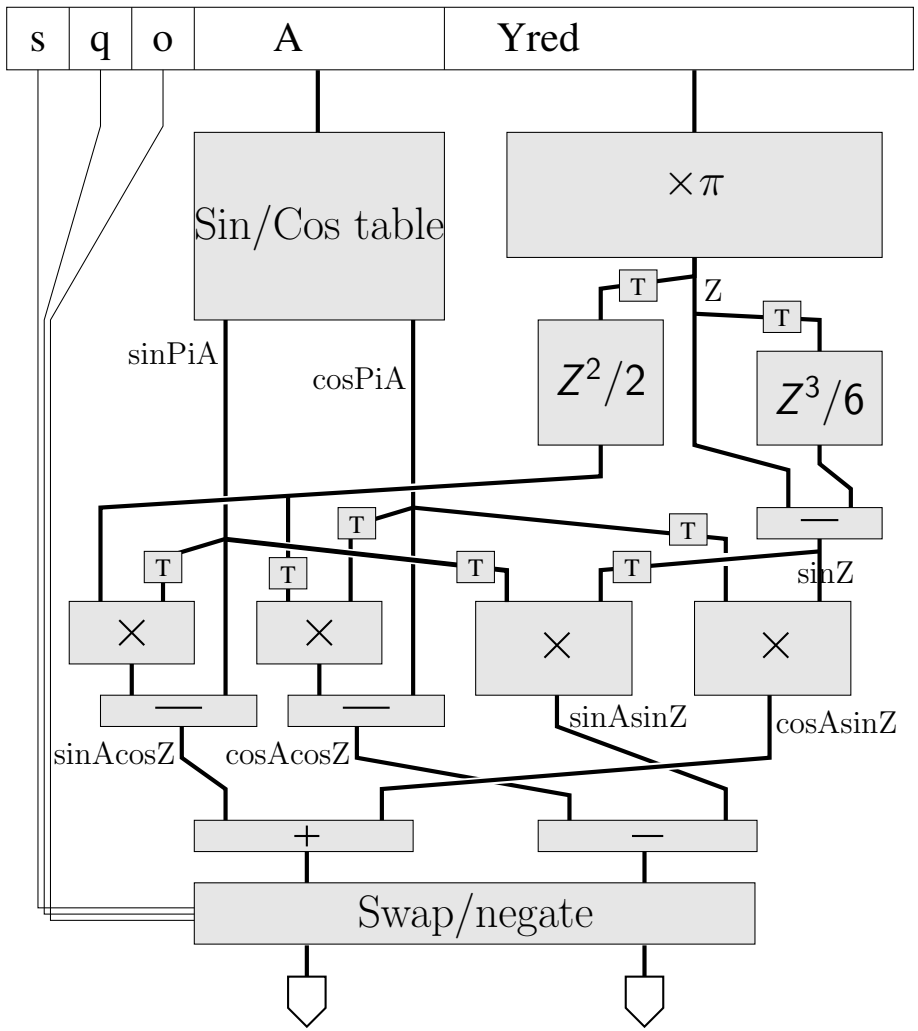
... where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

And then more

- A huge class of function may be *approximated* by polynomials
- The $b_{w,h}$ may be read from arbitrary look-up tables
- An operator may include several bit heaps

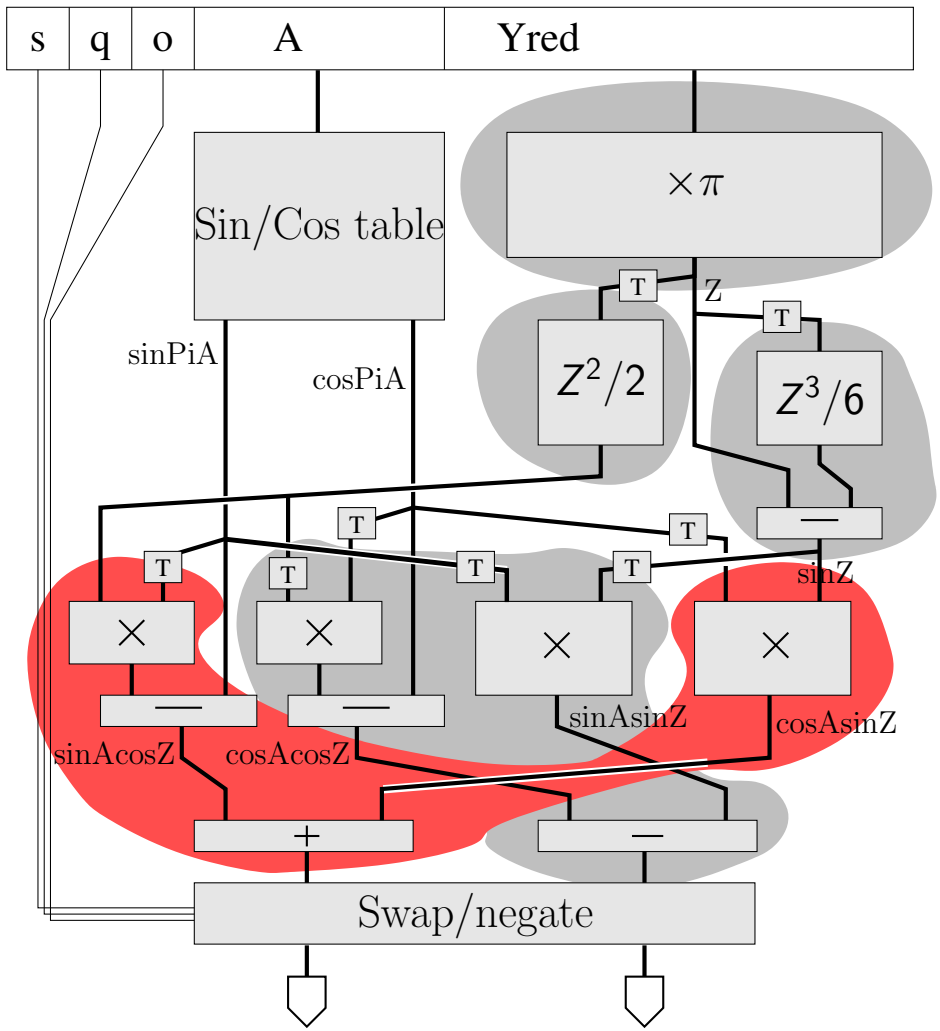
When you have a good hammer, you see nails everywhere

A sine/cosine architecture (HEART 2013)



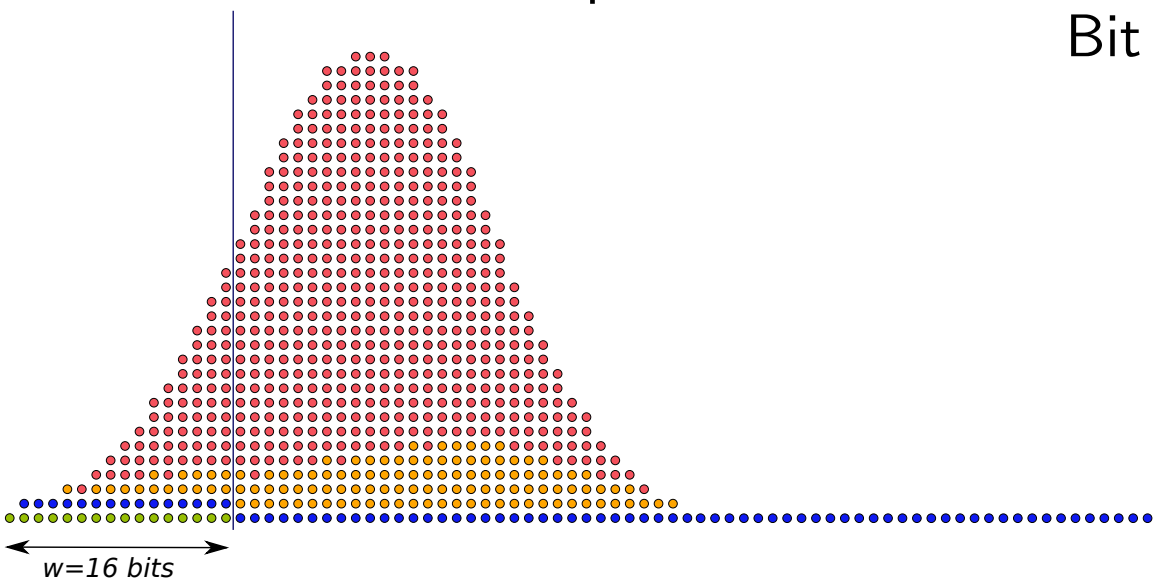
When you have a good hammer, you see nails everywhere

A sine/cosine architecture (HEART 2013) with 5 bit heaps

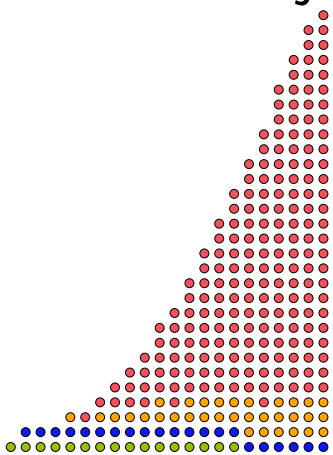


A bit heap for $Z - Z^3/6$ in the previous architecture

Full bit heap



Bit heap truncated just right



The constant vector

Quite often you need to add a constant to a bit heap:

- Rounding bit
- Constant coefficient
- Sign extension for two's complement (generalizating a classical multiplier trick)

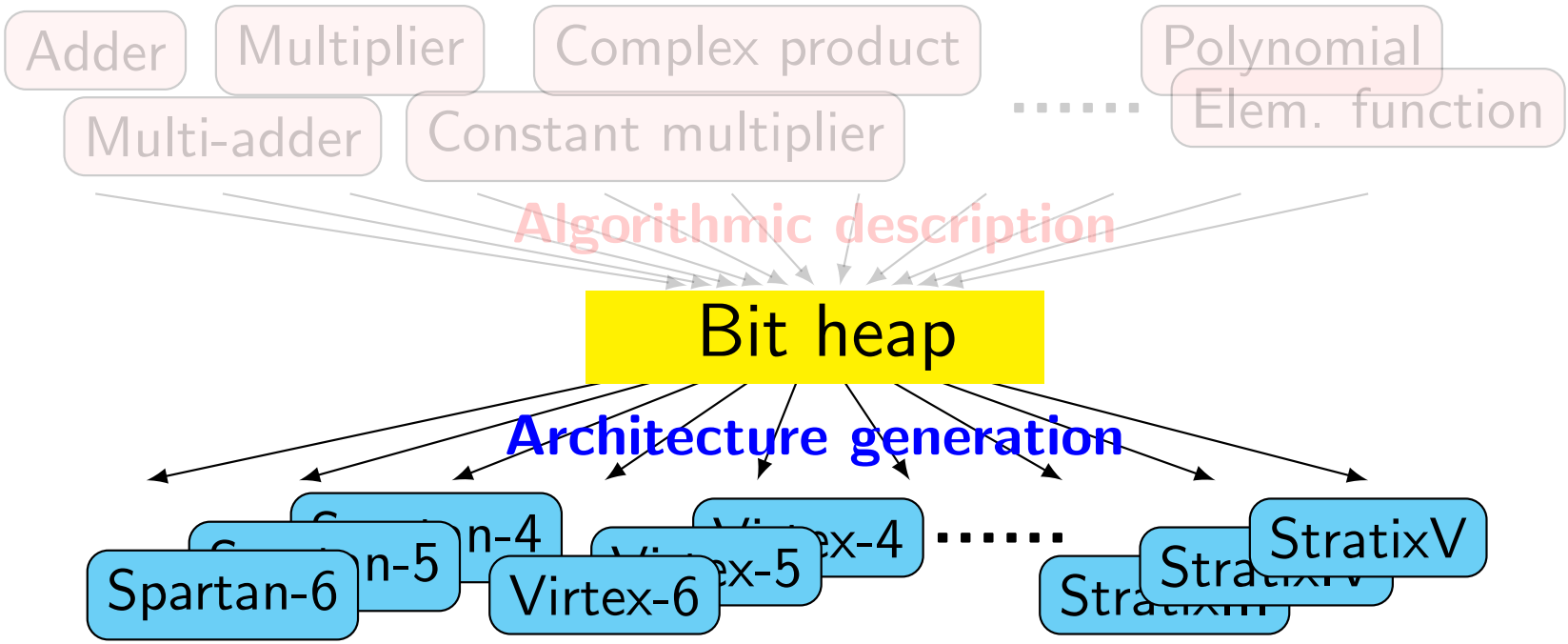
To replicate bit s from weight p to weight q

- add $\overline{s}$ at weight p.
- then add $2^q - 2^p$ to the constant bit vector
  (a string of 1's stretching from bit p to bit q)

This performs the sign extension both when $s = 0$ and $s = 1$.

All these constants may be pre-added, and only their sum added to the bit heap.
Managing signed number costs at most one line in the bit heap.

Generating an architecture



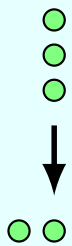
Architecture computing the value of a bit heap

Elementary case 1: the compressor

A compressor replaces a column of bits

by its sum written in binary (on fewer bits)

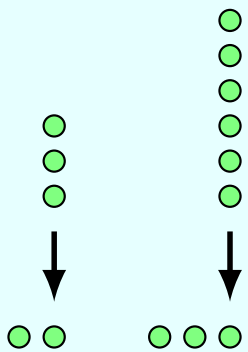
- archetype: the *full adder* is a *3 to 2* compressor



Architecture computing the value of a bit heap

Elementary case 1: the compressor

A compressor replaces a column of bits
by its sum written in binary (on fewer bits)

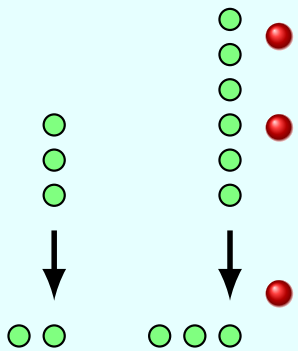


- archetype: the *full adder* is a *3 to 2* compressor
- on a recent FPGA: a *6 to 3* compressor
  tabulated in 3 6-input LUTs.
- survey and refs in the FPL 2013 paper, see also papers by M. Kumm.

Architecture computing the value of a bit heap

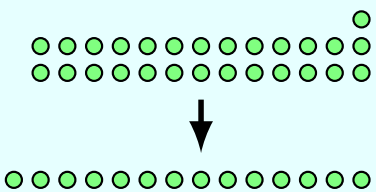
Elementary case 1: the compressor

A compressor replaces a column of bits
by its sum written in binary (on fewer bits)



- archetype: the *full adder* is a *3 to 2* compressor
- on a recent FPGA: a *6 to 3* compressor
  tabulated in 3 6-input LUTs.
- survey and refs in the FPL 2013 paper, see also papers by M. Kumm.

Elementary case 2: the adder

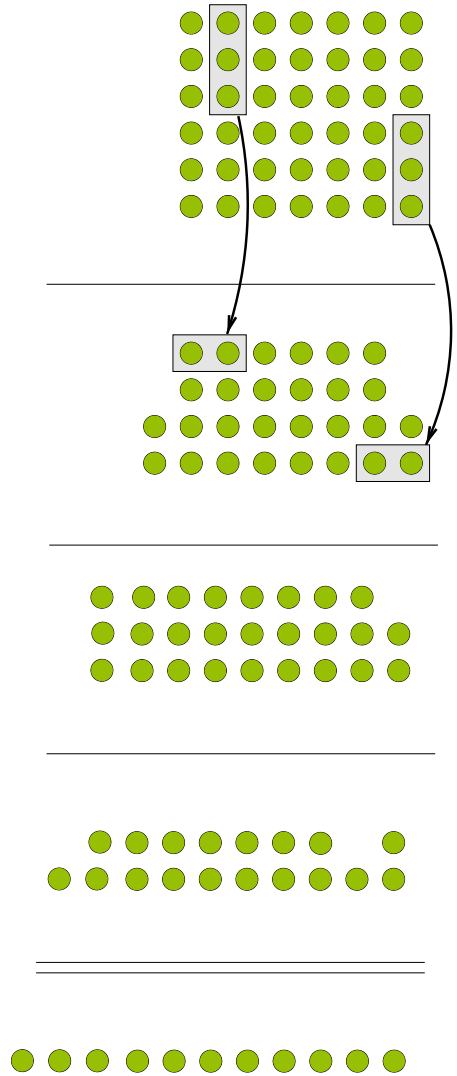


An adder replaces two n-bit lines, and a carry
by a line of $n+1$ bits

Architecture computing the value of a bit heap

1. Compression
   - Tile the bit heap with compressors
     - use as many compressors in parallel as possible
     - this produces a new, smaller bit heap
     - ... in one LUT delay
   - Start again on the compressed bit heap

   Stop when bit heap height equal to two

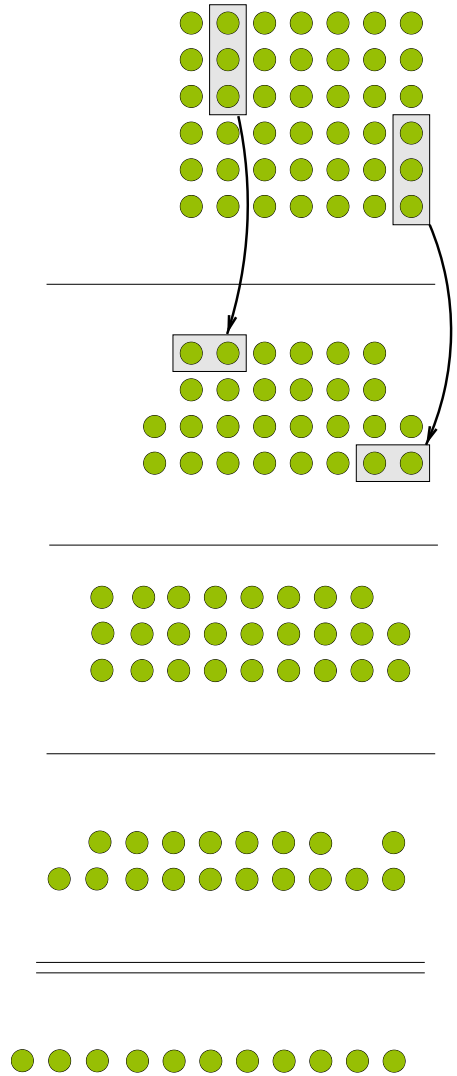


Architecture computing the value of a bit heap

1. Compression
   - Tile the bit heap with compressors
     - use as many compressors in parallel as possible
     - this produces a new, smaller bit heap
     - ... in one LUT delay
   - Start again on the compressed bit heap

   Stop when bit heap height equal to two

2. Final fast addition
   - add the remaining two lines

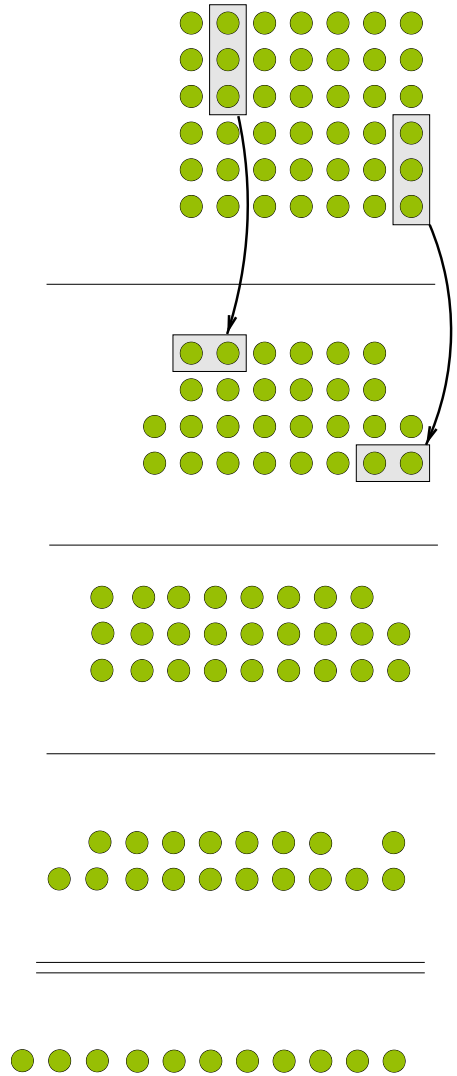


Architecture computing the value of a bit heap

1. Compression
   - Tile the bit heap with compressors
     - use as many compressors in parallel as possible
     - this produces a new, smaller bit heap
     - ... in one LUT delay
   - Start again on the compressed bit heap

   Stop when bit heap height equal to two

2. Final fast addition
   - add the remaining two lines



Both steps can be done in $\log n$ time and $n \log n$ area

Bit heaps and DSP blocks

Elementary case: the DSP block?

- Xilinx DSP blocks compute $\mathbf{A} + \mathbf{XY}$ (48+18x25)
- Altera DSP blocks compute $\mathbf{XY}$ (36x36)
  or $\mathbf{AB} \pm \mathbf{CD}$ (18x18+18x18) or ...

Really different architectures here

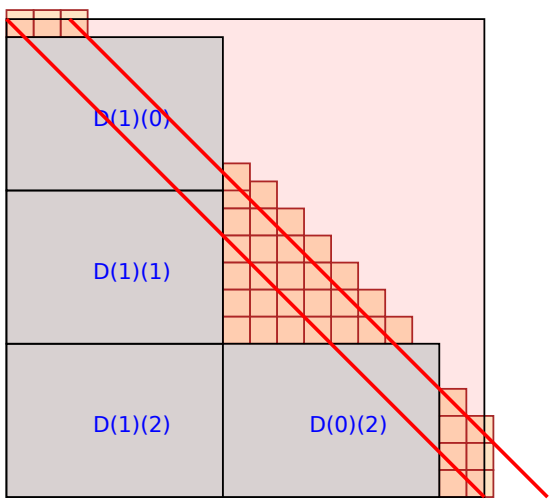
Bit heaps and DSP blocks

Elementary case: the DSP block?

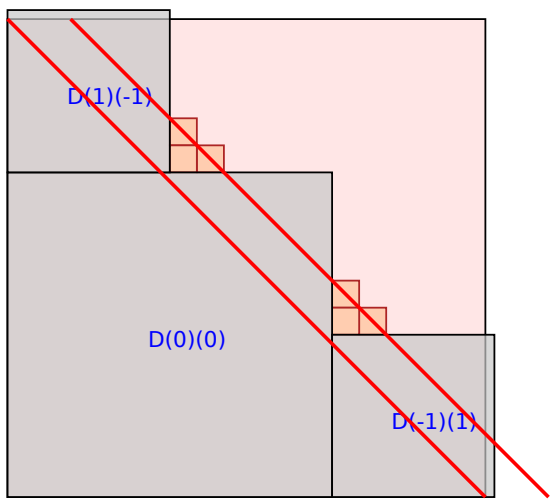
- Xilinx DSP blocks compute $\mathbf{A} + \mathbf{XY}$ (48+18x25)
- Altera DSP blocks compute $\mathbf{XY}$ (36x36)
  or $\mathbf{AB} \pm \mathbf{CD}$ (18x18+18x18) or ...

Really different architectures here

Exemple: 53-bit truncated multiplier



Virtex-5

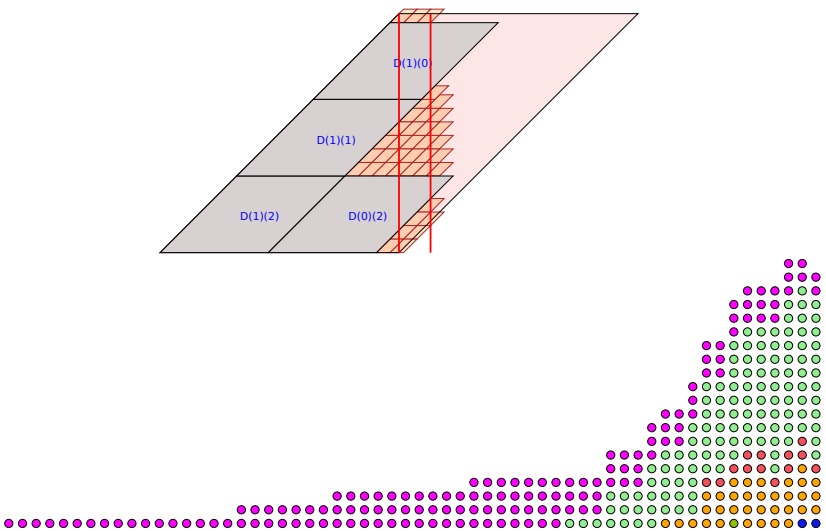


Stratix IV

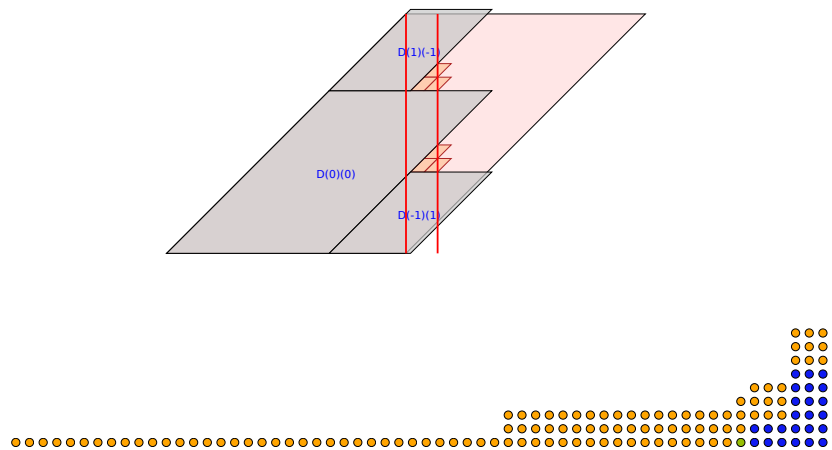
Reconciling bit heaps and DSP blocks

Instanciating DSP blocks is part of the compression

- merge operands from various sources in a DSP
- unused DSP adders may remove random bits from the heap



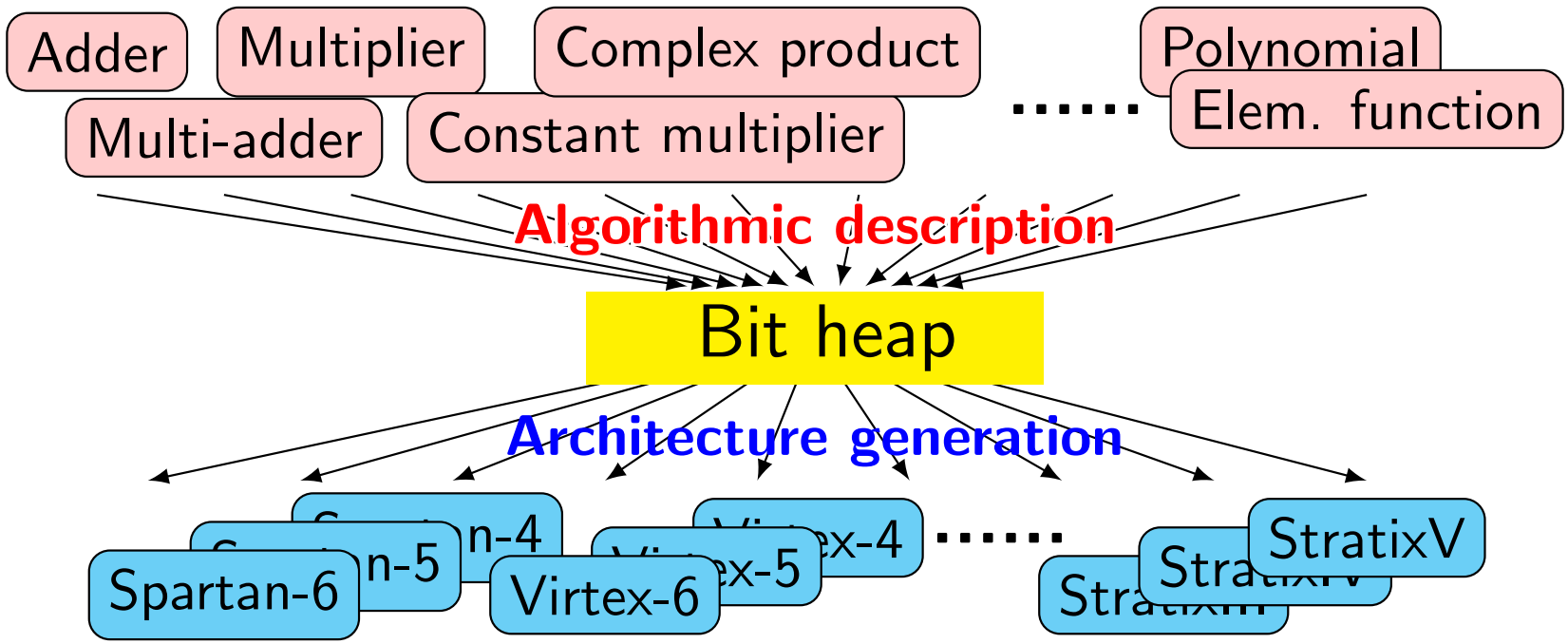
Virtex-5



Stratix IV

Many more details in the paper.

Current status



So, does it work?

Benefits in terms of software engineering

- Reduction of FloPoCo code size
- Fewer obscure bugs hidden in obscure operators
- (I didn't say fewer bugs)

So, does it work?

Benefits in terms of software engineering

- Reduction of FloPoCo code size
- Fewer obscure bugs hidden in obscure operators
- (I didn't say fewer bugs)

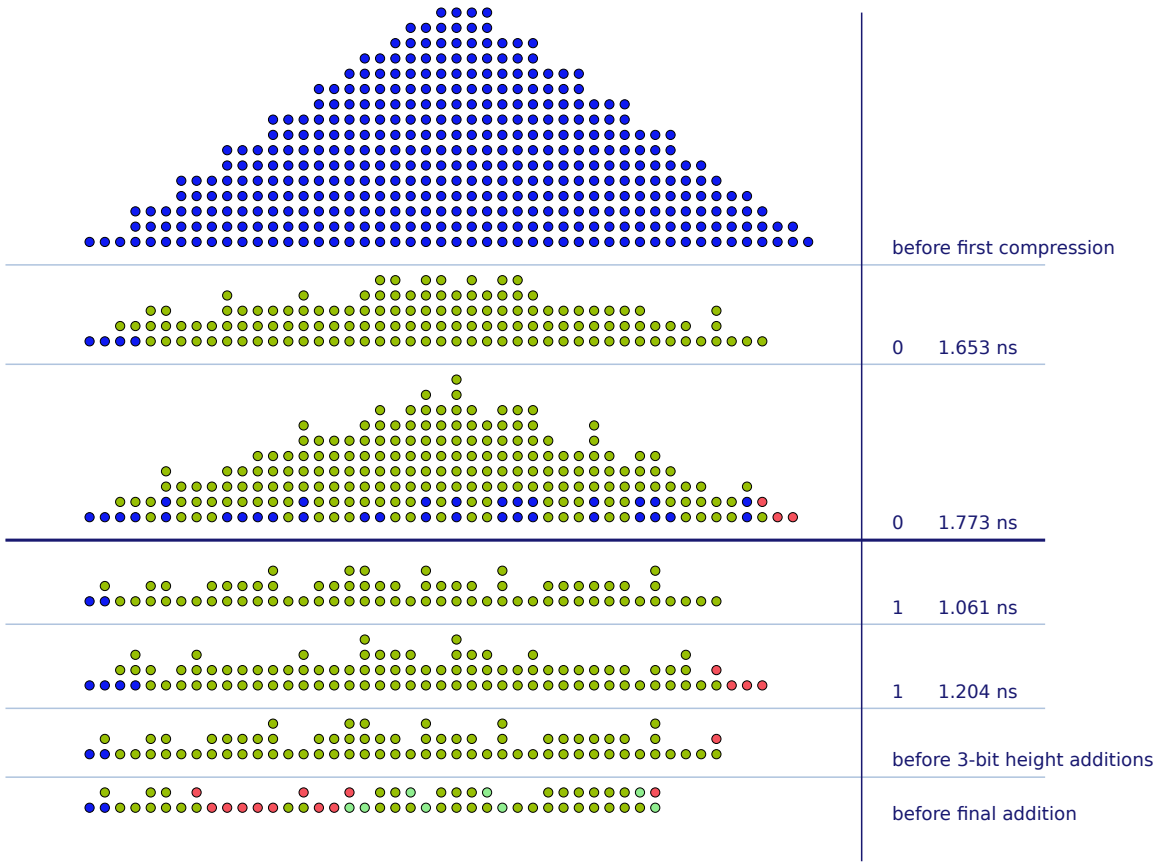
Benefits in terms of performance

... thanks to operator fusion

- Already a few examples
  - complex product
  - cosine transforms
- Still work in progress
  - improve compression heuristics
  - fuse in all the integer adder variants
  - rework the polynomial evaluator

Progress in the BitHeap class benefits to many operators

Generate VHDL, test bench, and nice clickable SVG graphics



Future work, from short-term to hopeless

- Adapt all the remaining operators to take advantage of bit heaps
- Improve the compression heuristics

  done, thanks to Martin Kumm
- Automate some of the algebraic optimisations done by hand so far
- Answer open questions like:

  How many bits must flip to compute 16 bits of sin(x)?

Conclusion

Computing just right

In a processor

the choice is between

- an integer SUV, or
- a floating-point SUV.

Computing just right

In a processor

the choice is between

- an integer SUV, or
- a floating-point SUV.

In an FPGA

- If all I need is a bicycle, I have the possibility to build a bicycle
- (and I'm usually faster to destination)

Computing just right

In a processor

the choice is between

- an integer SUV, or
- a floating-point SUV.

In an FPGA

- If all I need is a bicycle, I have the possibility to build a bicycle
- (and I'm usually faster to destination)

Save routing! Save power! Don't move useless bits around!

Busy until retirement (1)

An almost virgin land

Most of the arithmetic literature addresses the construction of SUVs.

Busy until retirement (2)

Designing the flexible parameters was only half of the problem...

- (the easy half)

The difficult half is: how to use them?

- What precision is required at what point of a computation

Meanwhile, in the real world

A very nice paper at Arith 2018 by Lutz and Bruguera:

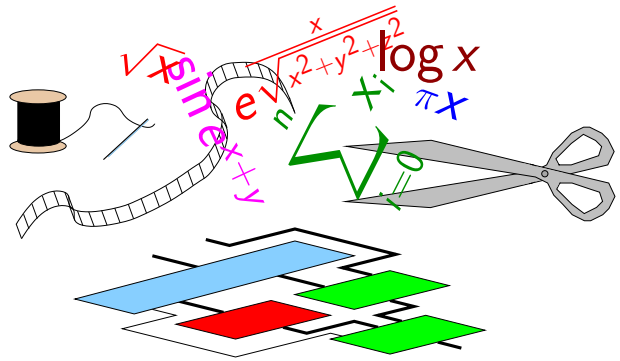
- radix-64 divider architecture in future ARM processors
- Massive speculation: replicating hardware that computes many results in parallel, most of which will be thrown out
- in order to reduce latency (whatever the hardware cost)
- ... and this is a low-power processor!

Almost, but not quite, everything but Computing Just Right

Any cycle gain
allowing us to switch off earlier this huge superscalar core
actually saves energy

Thanks for your attention

The following people have contributed to FloPoCo:
S. Banescu, Louis Beseme, Nicolas Bonfante, Maxime Christ, N. Brunie, S. Collange, J. Detrey, P. Echeverría, F. Ferrandi, Luc Forget, M. Grad, K. Illyes, M. Istoan, M. Joldes, J. Kappauf, C. Klein, M. Kleinlein, M. Kumm, D. Mastrandrea, K. Moeller, B. Pasca, B. Popa, X. Pujol, G. Sergent, D. Thomas, R. Tudoran, A. Vasquez.



http://flopoco.gforge.inria.fr/

Introduction and motivation

Optimizing operators in context

Example: Multiplication by rational constants

Example: Sin/Cos

The universal bit heap

Conclusion