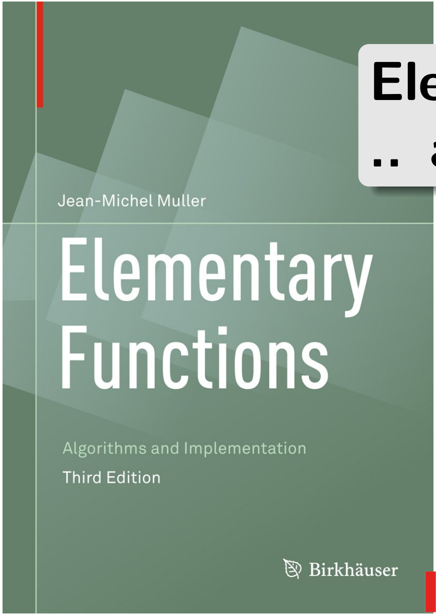


RAIM 2025

Elementary functions (or not) .. and implementations

Florent de Dinechin

citi lab

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Inria

UNIVERSITÉ DE LYON

Introduction

A panorama of elementary function implementation problems

Software	Hardware

A panorama of elementary function implementation problems

Software

arbitrary precision

floating point

Hardware

A panorama of elementary function implementation problems

Software

arbitrary precision

floating point

Hardware

A panorama of elementary function implementation problems

Software | Hardware

arbitrary precision | floating point

arb | mpfr | vector | CR

… | … | … | …

A panorama of elementary function implementation problems

Software | Hardware

Software			
arbitrary precision		floating point	
arb	mpfr	vector	CR
...	...	...	...

A panorama of elementary function implementation problems

Software

arbitrary precision | floating point

arb | mpfr | vector | CR

··· | ··· | ··· | ···

Hardware

function-specific methods | generic methods

A panorama of elementary function implementation problems

Software

arbitrary precision

arb

mpfr

floating point

vector

CR

Hardware

function-specific methods

addition based

CORDIC

BKM

multiplication based

Newton Raphson

generic methods

A panorama of elementary function implementation problems

- Software
  - arbitrary precision
    - arb
      - ...
    - mpfr
      - ...
  - floating point
    - vector
      - ...
    - CR
      - ...
- Hardware
  - function-specific methods
    - addition based
      - CORDIC
      - BKM
      - ... ...
    - multiplication based
      - Newton Raphson
      - ... ...
  - generic methods

A panorama of elementary function implementation problems

Software

Hardware

generic methods

A panorama of elementary function implementation problems

Software

Hardware

generic methods

tables

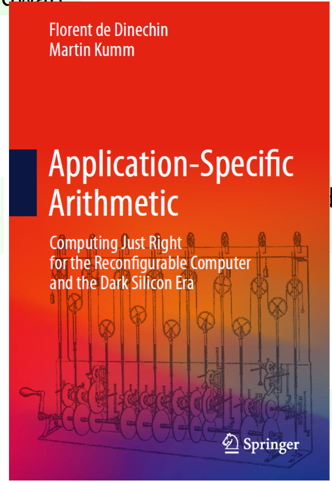
tables and additions

polynomial based

and some new stuff

A panorama of elementary function implementation problems

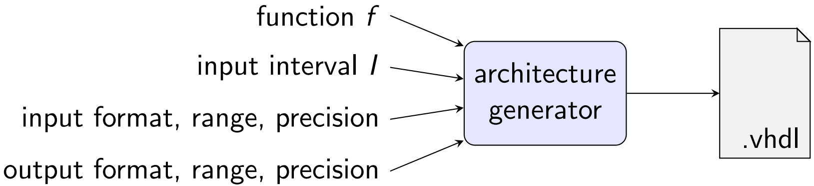
Software | Hardware



Generic Generators

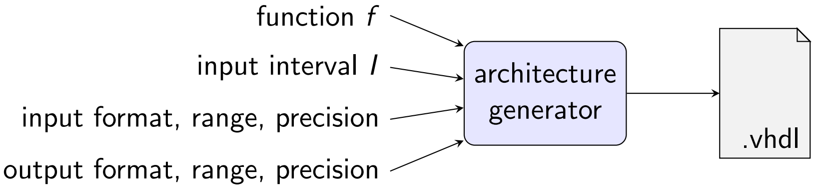
Janitoring generic generators is a genuine chore

Canonical/minimal interface ?



Janitoring generic generators is a genuine chore

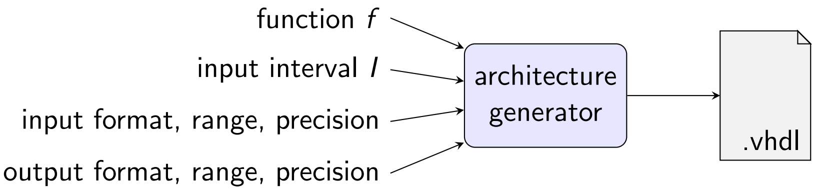
Canonical/minimal interface ?



- Early attempts would use a drop-down menu of possible functions

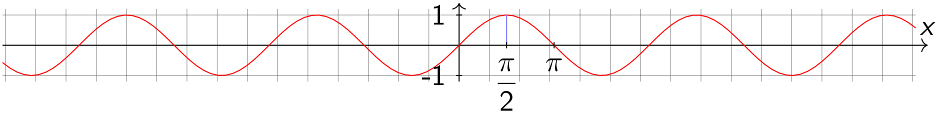
Janitoring generic generators is a genuine chore

Canonical/minimal interface ?

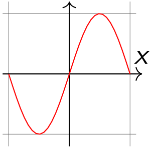


- Early attempts would use a drop-down menu of possible functions
- And then came Sollya:
  - arbitrary functions can be provided as expressions
  - ... which Sollya will evaluate accurately to arbitrary precision

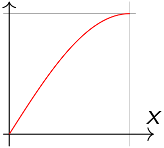
Encoding input ranges as scalings in the function

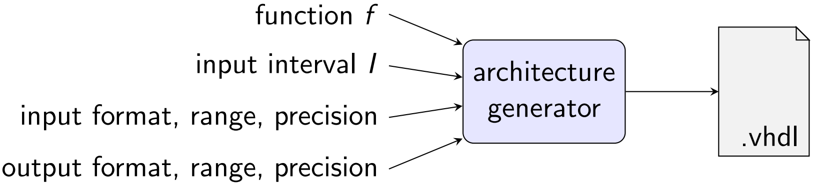


$\sin(\pi x)$ on $[-1,1]$:

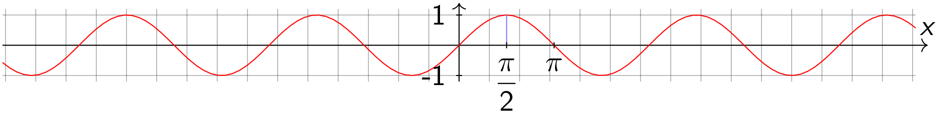


$\sin(\frac{\pi}{2}x)$ on $[0,1)$:

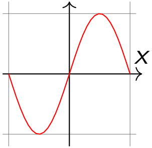




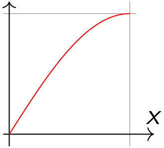
Encoding input ranges as scalings in the function

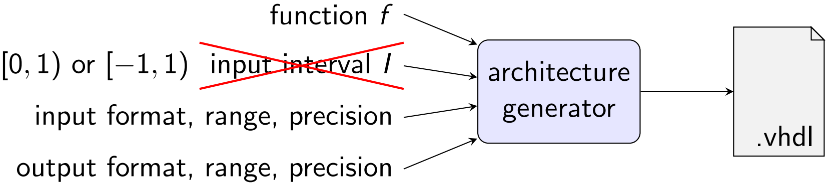


$\sin(\pi x)$ on $[-1, 1)$:

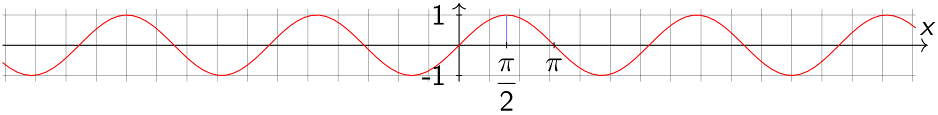


$\sin(\frac{\pi}{2}x)$ on $[0, 1)$:

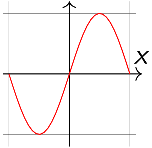




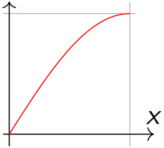
Encoding input ranges as scalings in the function

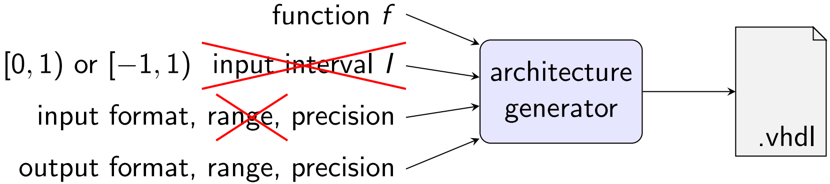


$\sin(\pi x)$ on $[-1,1)$:

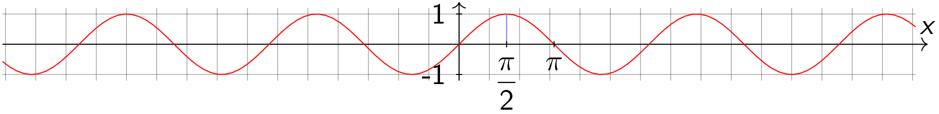


$\sin(\frac{\pi}{2}x)$ on $[0,1)$:

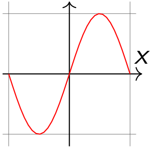




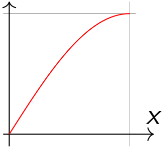
Encoding input ranges as scalings in the function

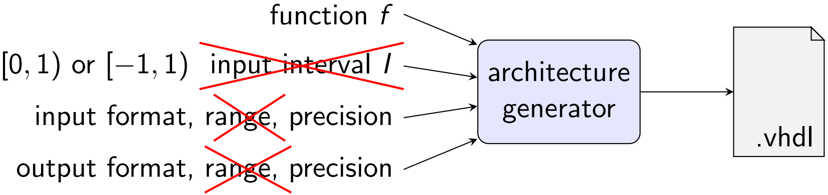


$\sin(\pi x)$ on $[-1, 1)$:



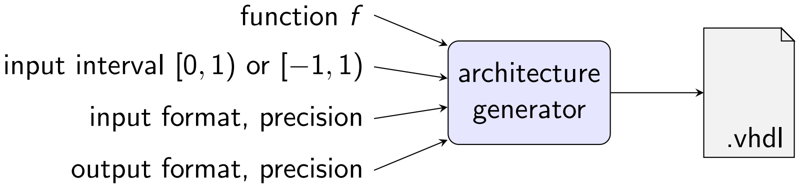
$\sin(\frac{\pi}{2}x)$ on $[0, 1)$:



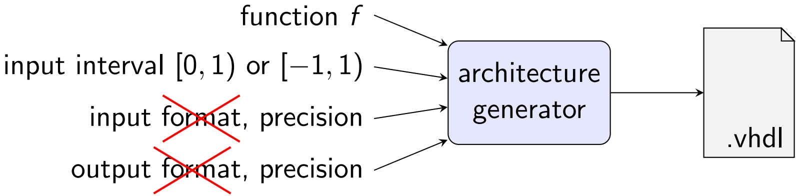


Oh, and output range can be computed...

Reduce everything to fixed-point

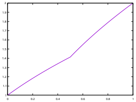


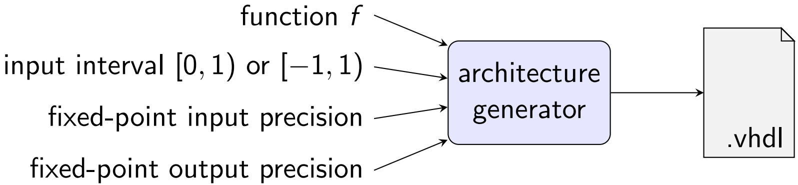
Reduce everything to fixed-point



Reduce everything to fixed-point

- For a generator for the function g over posits, just input $f = \text{encode}(g(\text{decode}(x)))$
- More seriously, this is one of the reasons for range reductions

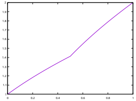
$f(x) = \exp(x) - x - 1, \qquad f(x) = \log(1+x)/x \qquad$ and composite functions 



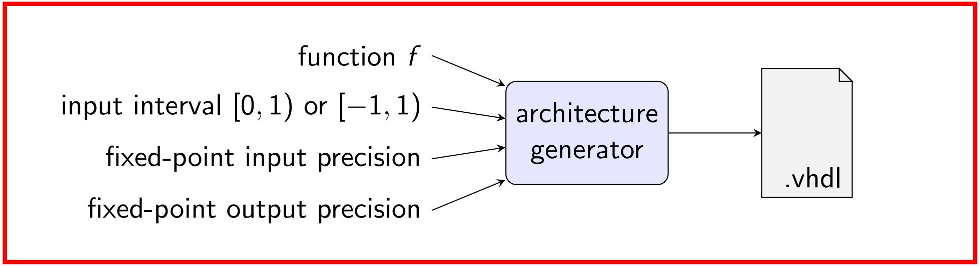
Reduce everything to fixed-point

- For a generator for the function g over posits, just input $f = \text{encode}(g(\text{decode}(x)))$
- More seriously, this is one of the reasons for range reductions

$f(x) = \exp(x) - x - 1, \qquad f(x) = \log(1+x)/x \qquad$ and composite functions

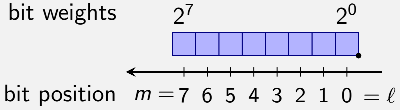


Canonical minimal interface



Fixed-point for dummies (1)

Unsigned integers



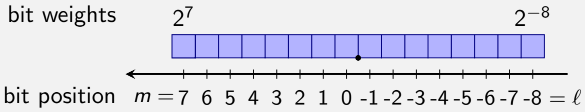
- m position of the most significant bit, defines the range of representable numbers
- least significant bit at position $\ell = 0$
- position i corresponds to weight 2^i of the bit:

$$\text{Encoded value is } X = \sum_{i=\ell}^{m} 2^i x_i$$

- $00 \cdots 00$ encodes 0
- $11 \cdots 11$ encodes $2^{m+1} - 1$

Fixed-point for dummies (2)

Unsigned fixed point: just relax ℓ



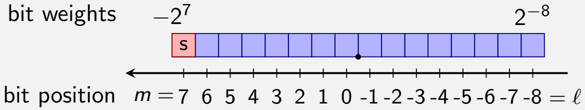
- least significant bit at arbitrary position ℓ
- ℓ defines the resolution / quantization step / unit in the last place: 2^ℓ
- MSB position m still defines the range
- position i still corresponds to weight 2^i of the bit:

$$\text{Encoded value is still } X = \sum_{i=\ell}^{m} 2^i x_i \qquad \text{(an integer scaled by } 2^\ell\text{)}$$

- $00\cdots00$ encodes 0, $\qquad 11\cdots11$ encodes $2^{m+1} - 2^\ell$

Fixed-point for dummies (3)

Signed fixed point: just change the weight of one bit



- ℓ still defines the resolution / quantization step / unit in the last place 2^ℓ
- MSB position m still defines the range
- ... but the MSB has a negative weight -2^m:

$$\text{Encoded value is } X = -2^m x_m + \sum_{i=\ell}^{m-1} 2^i x_i$$

- $10\cdots00$ encodes -2^m, $00\cdots00$ encodes 0, $01\cdots11$ encodes $2^m - 2^\ell$

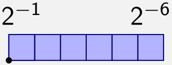
And finally my input intervals

"Unsigned" $[0, 1)$ is actually $[0, 1 - 2^{\ell}]$

Example: $\ell = -6$

- ulp $u = 2^{-6}$
- 6 bits total

bit weights

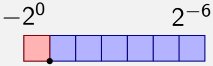


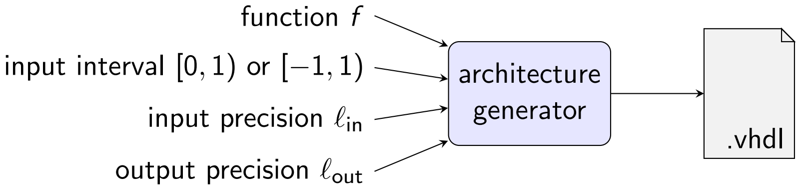
"Signed" $[-1, 1)$ is actually $[-1, 1 - 2^{\ell}]$

Example: $\ell = -6$

- ulp $u = 2^{-6}$
- 7 bits total

bit weights

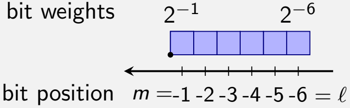


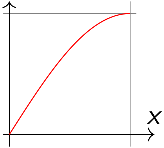


My first function generator: FixFunctionByTable

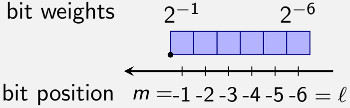
```
flopoco FixFunctionByTable f="sin(pi/2*x)" signedIn=0 lsbIn=-6 lsbOut=-6
```

Input





Output should be

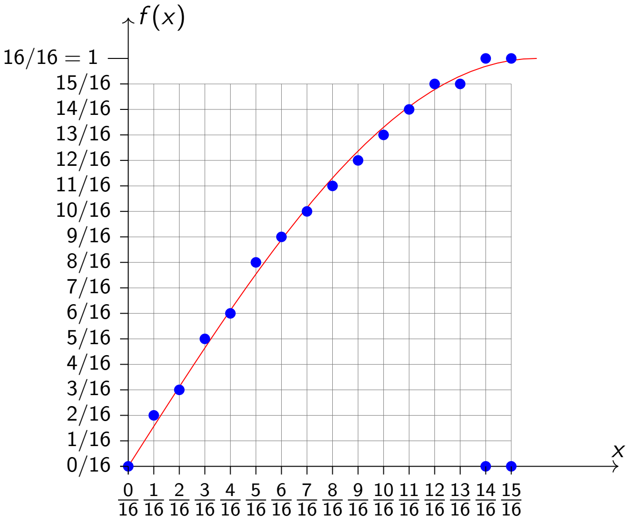


m_{out} is computed

Let us check the VHDL...

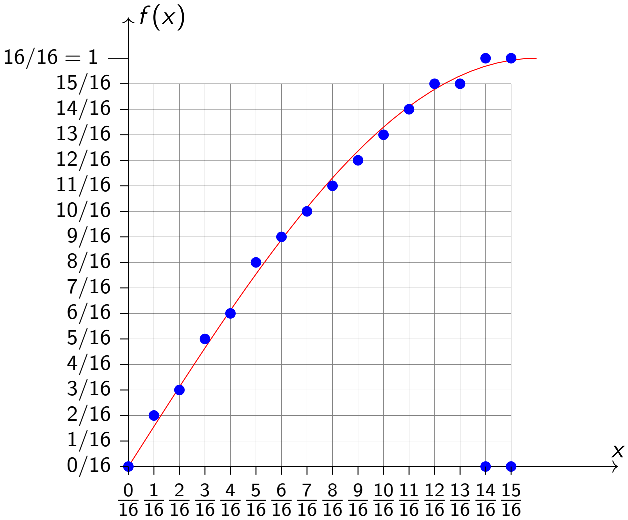
?!? Why do I have one more output bit than input bit ?

1 is excluded from the interval [0, 1)

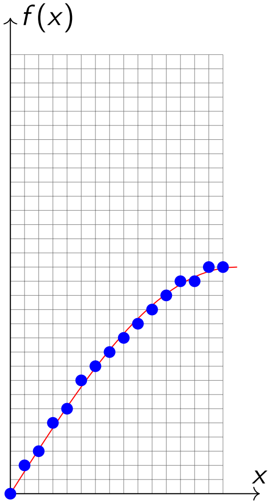


?!? Why do I have one more output bit than input bit ?

1 is excluded from the interval [0, 1)

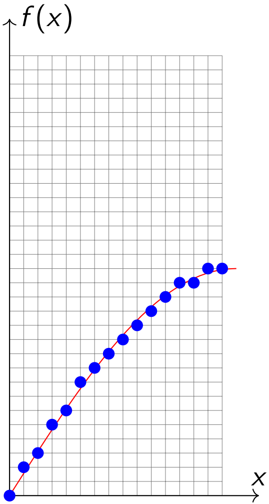


… so FloPoCo added one more bit

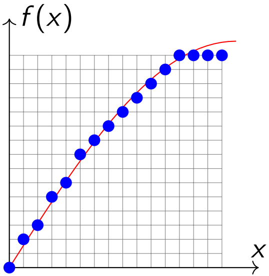


The expression parser to the rescue (again)

Waste of output space

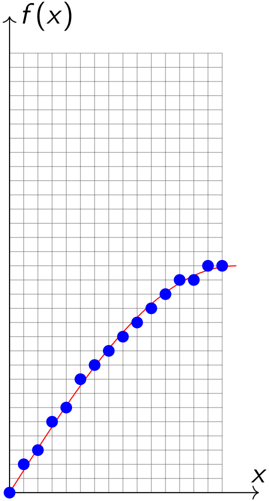


Saturation
(will not sound nice)

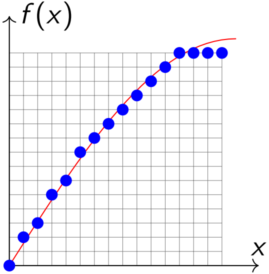


The expression parser to the rescue (again)

Waste of output space

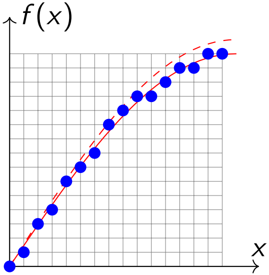


Saturation
(will not sound nice)



Output scaling by $\frac{15}{16}$:

$$f(x) = \frac{15}{16}\sin(\frac{\pi}{2}x)$$



FixFunctionByTable, fixed

```
flopoco FixFunctionByTable f="63/64*sin(pi/2*x)" signedIn=0 lsbIn=-6 lsbOut=-6
```

Thanks to the Sollya expression parser again.

... also provides the cryptic but useful version: `f="(1-1b-6)*sin(pi/2*x)"`

where (1-1b-6) reads $(1 - 2^{-6})$

FixFunctionByTable, fixed

```
flopoco FixFunctionByTable f="63/64*sin(pi/2*x)" signedIn=0 lsbIn=-6 lsbOut=-6
```

Thanks to the Sollya expression parser again.

... also provides the cryptic but useful version: `f="(1-1b-6)*sin(pi/2*x)"`

where (1-1b-6) reads $(1 - 2^{-6})$

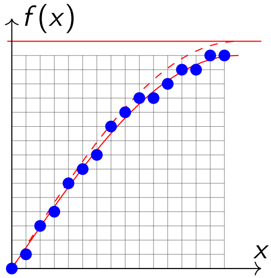
A generic generator should be faithful

faithful: $\overline{|\epsilon|} < u$; correctly rounded: $\overline{|\epsilon|} \leq u/2$;

- marketing: every output bit counts
- interface: no "accuracy" input needed:

  output precision specifies it

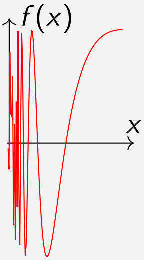
and here: the output will never reach 1.



Sollya helps a lot to achieve this, too.

Tables can hold functions that are arbitrarily ugly

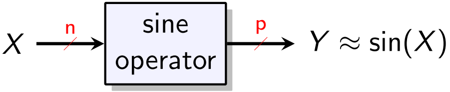
$\sin(\frac{\pi}{2x})$ on $[0, 1)$




```
flopoco FixFunctionByTable f="(1-1b-16)*sin(pi/2/x)" signedIn=0 lsbIn=-16
    lsbOut=-16
```

Thanks to Sollya magic, we get correctly rounded values to the output format.
For what it is worth.

Hardware cost of plain tables



2^n entries of p bits each, so $2^n \times p$ bits

- very good for really small precisions
- for larger precisions, cost grows exponentially in n

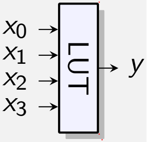
address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

2^n

p

Hardware cost of plain tables on FPGAs

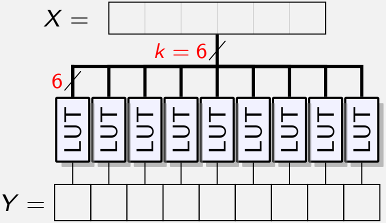
FPGAs are LUT-based



x_0	x_1	x_2	x_3	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Practical sizes on FPGAs with k-input LUTs

- A table of $2^k \times p$ bits costs exactly p LUTs.



- In general: LUT cost of a $2^n \times p$ table is $2^{n-k} \times p$

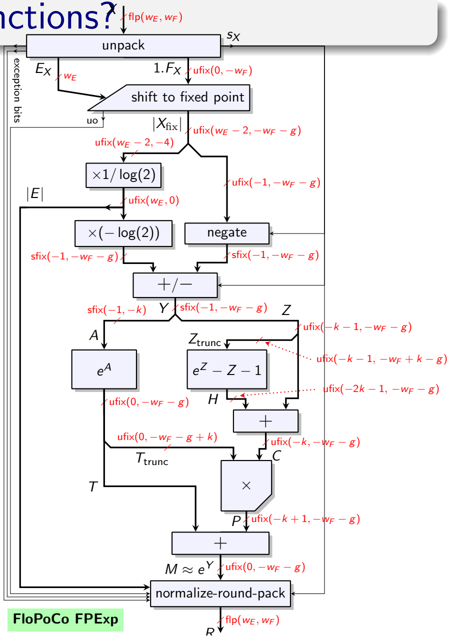
- A 20 Kb dual-port BlockRAM can hold two tables of $2^{10} \times 10$ bits.

Accurate precision-contracting functions

Precision contracting functions?

This means: **more input bits than output bits**.
Why would anybody want this?

Precision contracting functions?

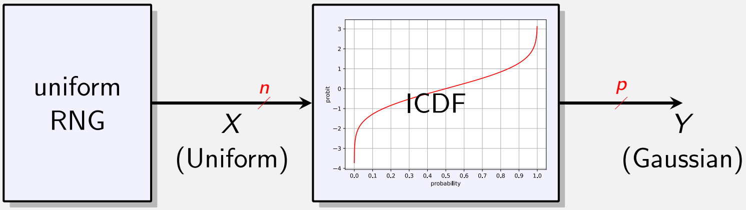


FloPoCo FPExp

This means: **more input bits than output bits**.
Why would anybody want this?

They are everywhere if you are paranoid enough

Pseudo random number generation for non-uniform distributions



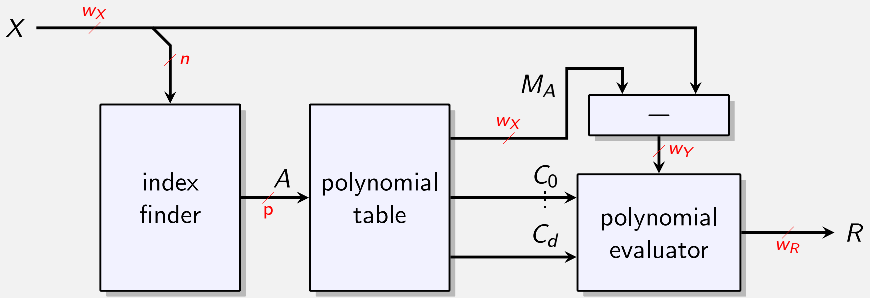
ICDF: Inverse Cumulative Distribution Function
(there is one for each distribution)

A lot of resolution in X needed to capture the tail.

Faithful is OK, but rounding the input to p bits won't do.

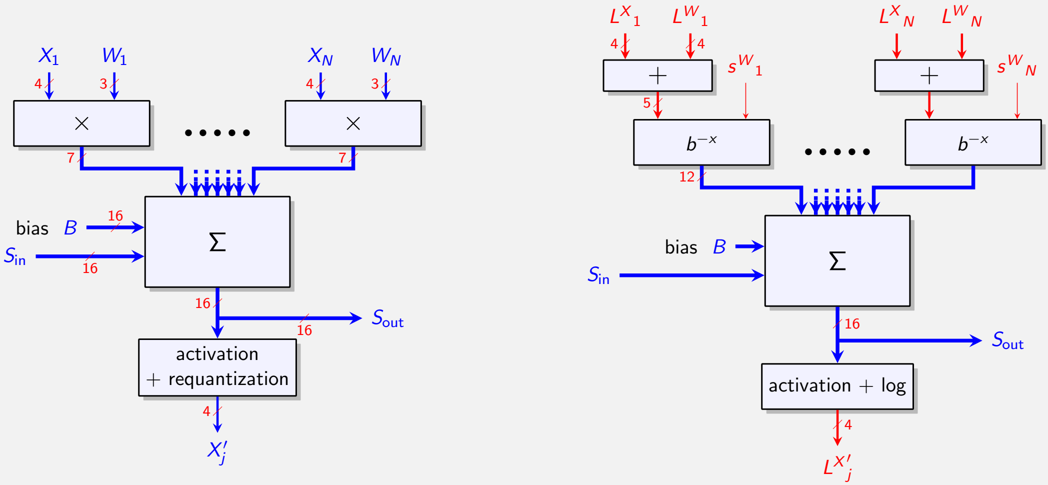
They are everywhere if you are paranoid enough

Finding the interval for a non-uniform polynomial evaluation



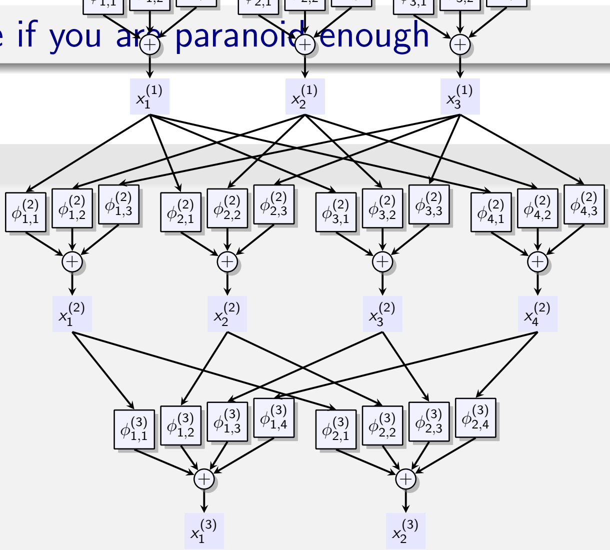
They are everywhere if you are paranoid enough

Deeply quantized neural networks



They are everywhere if you are paranoid enough

… or Kolmogorov-Arnold Networks



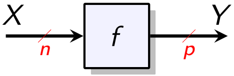
They are everywhere if you are paranoid enough

... and then more

- Direct Digital Synthesis without spurious frequencies
- High Dynamic Range imaging
- ...

Logic synthesis is your friend (or not)

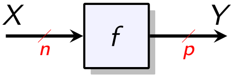
We know in principle what a table is going to cost:



- Any boolean function can be described as a table of $2^n \times p$ bits.
- Address-decoding logic (mux trees) whose area is also in $2^n \times p$.
- Your constants may vary.

Logic synthesis is your friend (or not)

We know in principle what a table is going to cost:



- Any boolean function can be described as a table of $2^n \times p$ bits.
- Address-decoding logic (mux trees) whose area is also in $2^n \times p$.
- Your constants may vary.

And then the tools do their magic

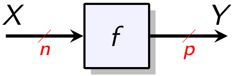
Oscar Gustafsson and Kenny Johansson.
An Empirical Study on Standard Cell Synthesis of Elementary Function Lookup Tables.
In: Asilomar Conference on Signals, Circuits and Systems. IEEE, 2008, pp. 1810–1813

$$\text{Area is proportional to } 2^{0.65\min(n,p)} \times 2^{0.19|n-p|} \quad .$$

Admire that the formula is symmetrical in input and output sizes...

Logic synthesis is your friend (or not)

We know in principle what a table is going to cost:



- Any boolean function can be described as a table of $2^n \times p$ bits.
- Address-decoding logic (mux trees) whose area is also in $2^n \times p$.
- Your constants may vary.

And then the tools do their magic

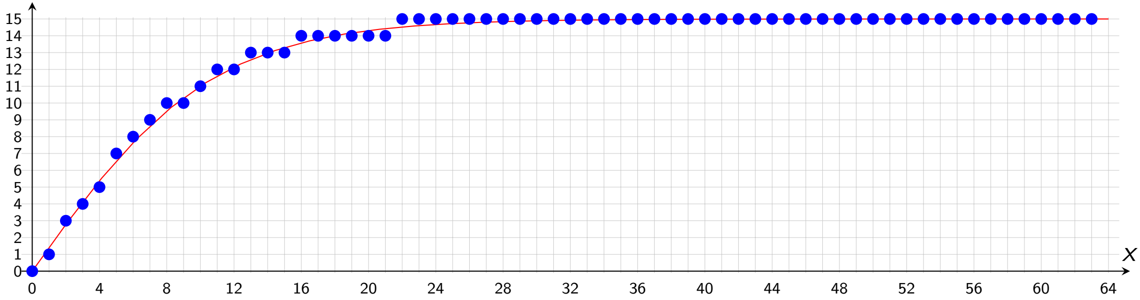
Oscar Gustafsson and Kenny Johansson.
An Empirical Study on Standard Cell Synthesis of Elementary Function Lookup Tables.
In: Asilomar Conference on Signals, Circuits and Systems. IEEE, 2008, pp. 1810–1813

$$\text{Area is proportional to } 2^{0.65\min(n,p)} \times 2^{0.19|n-p|} \quad .$$

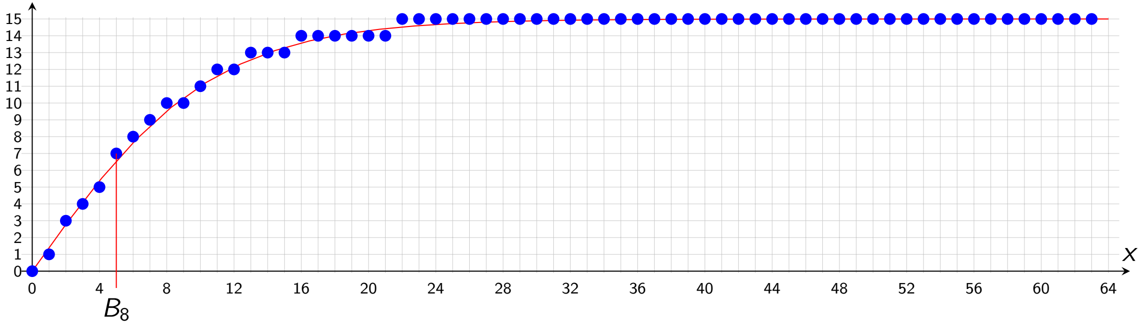
Admire that the formula is symmetrical in input and output sizes...

Can we engineer this symmetry? Can we build architectures in $2^p \times n$ when $n \gg p$?

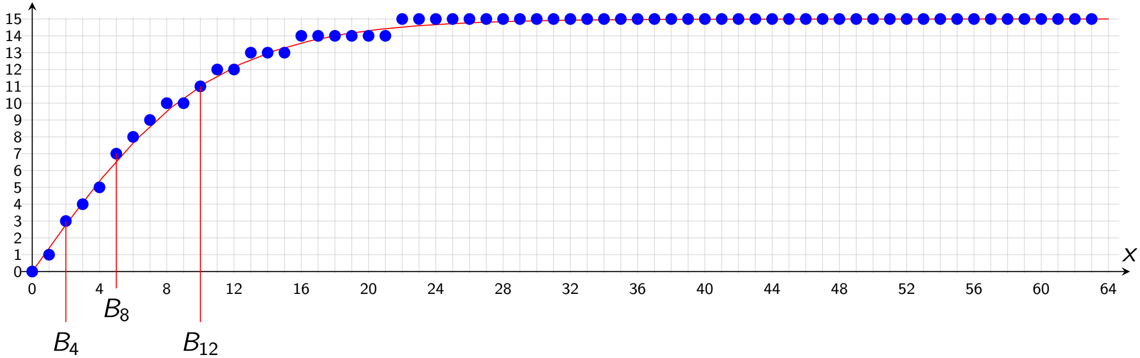
Core idea: a binary search in the inverse table



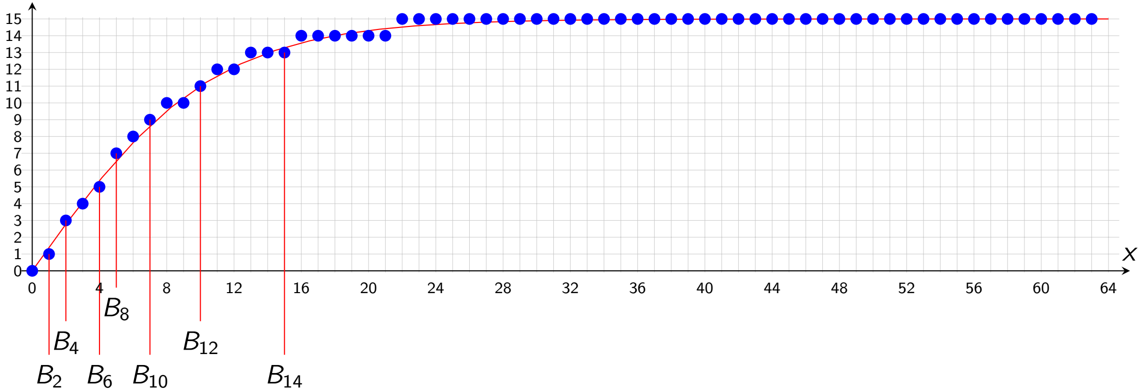
Core idea: a binary search in the inverse table



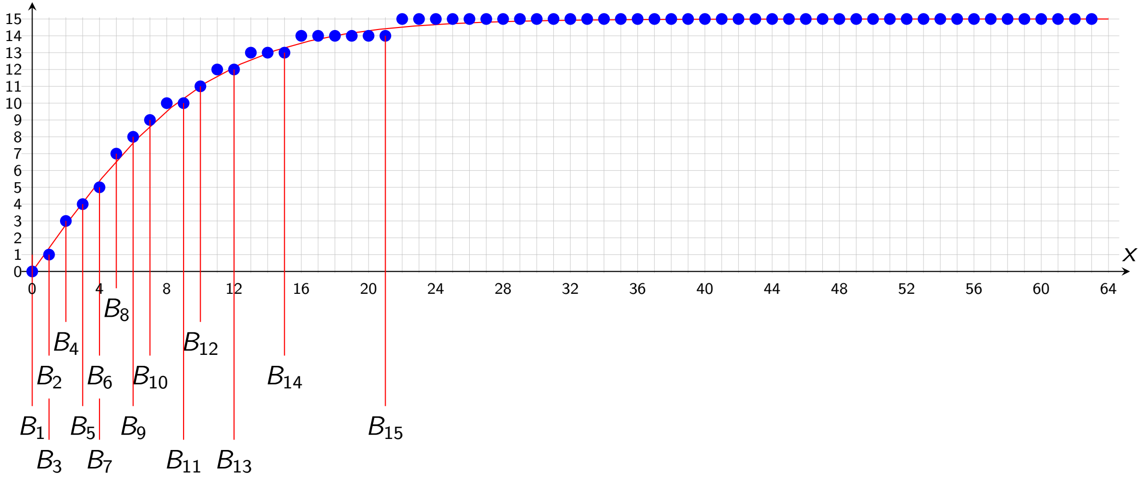
Core idea: a binary search in the inverse table



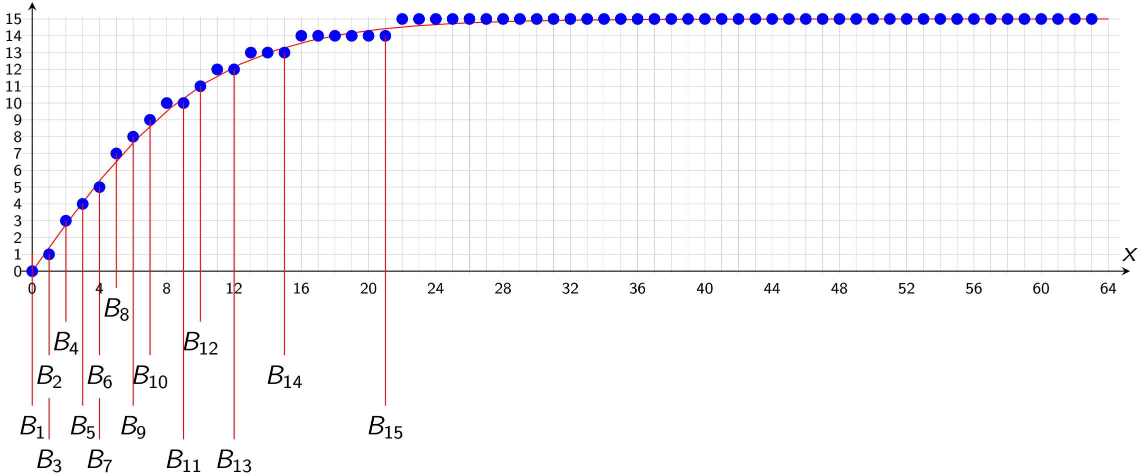
Core idea: a binary search in the inverse table



Core idea: a binary search in the inverse table



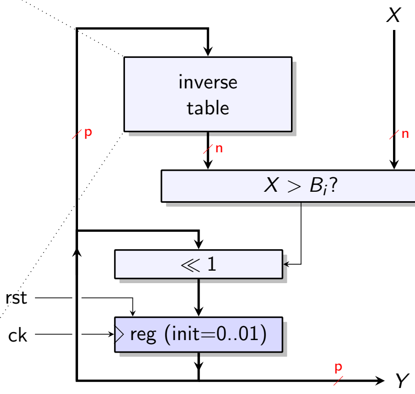
Core idea: a binary search in the inverse table



Of course it only works if the function is monotonic...

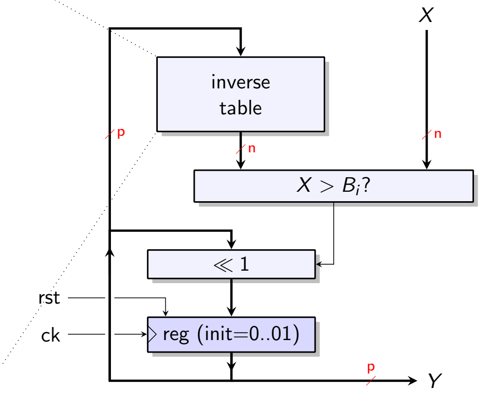
Binary search using a state machine and the inverse table

address	data
0000	—
0001	B_8
0010	B_4
0011	B_{12}
0100	B_2
0101	B_6
0110	B_{10}
0111	B_{14}
1000	B_1
1001	B_3
1010	B_5
1011	B_7
1100	B_9
1101	B_{11}
1110	B_{13}
1111	B_{15}



Binary search using a state machine and the inverse table

address	data
0000	—
0001	B_8
0010	B_4
0011	B_{12}
0100	B_2
0101	B_6
0110	B_{10}
0111	B_{14}
1000	B_1
1001	B_3
1010	B_5
1011	B_7
1100	B_9
1101	B_{11}
1110	B_{13}
1111	B_{15}



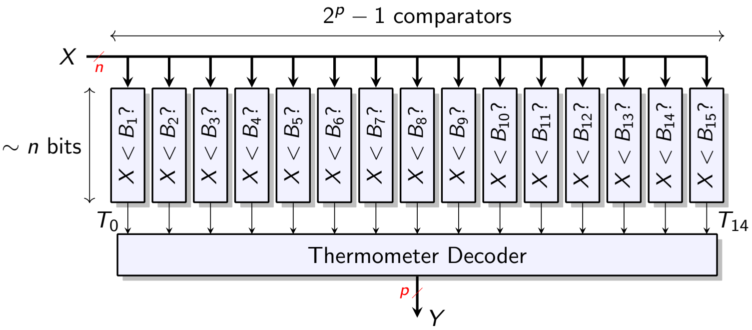
Time:

- p iterations,
- each with the latency of a n-bit comparison

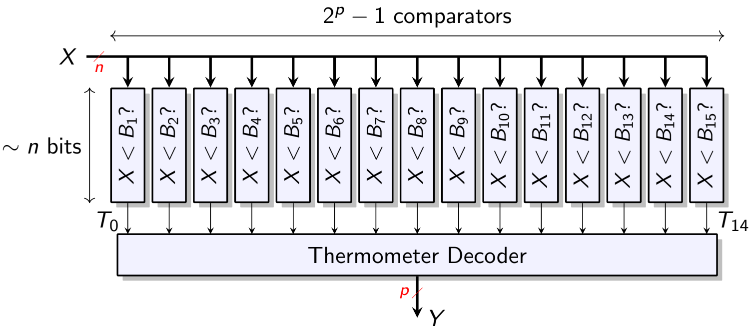
Area:

- one comparator of n bits,
- one shift register of size p bits,
- one table of size $2^p n$

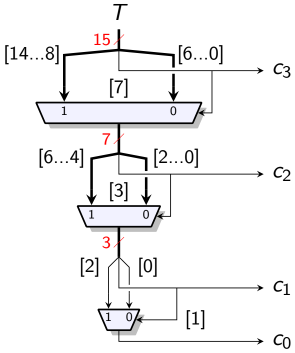
Binary search in parallel hardware



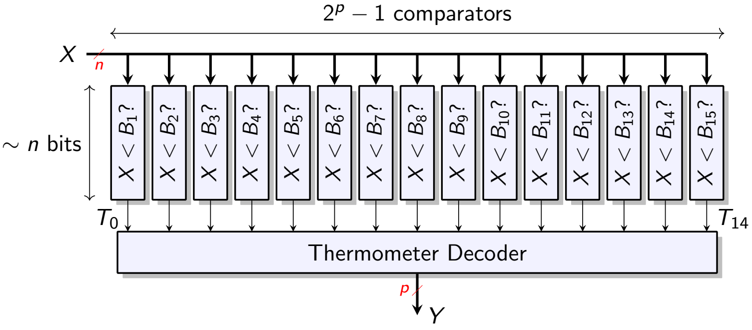
Binary search in parallel hardware



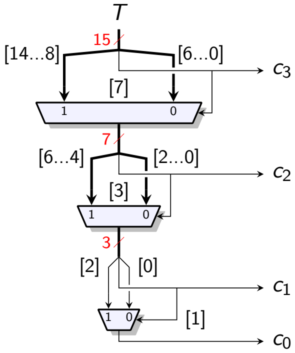
Thermometer decoder: counts the ones in a thermometer code.



Binary search in parallel hardware



Thermometer decoder: counts the ones in a thermometer code.



Time:

- one constant comparison on n bits, and p mux2 delays,

Area:

- $2^p - 1$ comparators of n bits (so $\sim 2^p n$, see next slide)
- $2^p - p$ mux2 in the decoder.

By the way, what is the cost of a comparator?

- Naive idea: subtract, then take the sign

By the way, what is the cost of a comparator?

- Naive idea: subtract, then take the sign
- Recursive comparison tree:
  - split X into X_H (high bits) and X_L (low bits). Same for Y.
  - Then it is a lexicographic comparison (lc):

X lc Y	$X_H < Y_H$	$X_H = Y_H$	$X_H > Y_H$
$X_L < Y_L$	$X < Y$	$X < Y$	$X > Y$
$X_L = Y_L$	$X < Y$	$X = Y$	$X > Y$
$X_L > Y_L$	$X < Y$	$X > Y$	$X > Y$

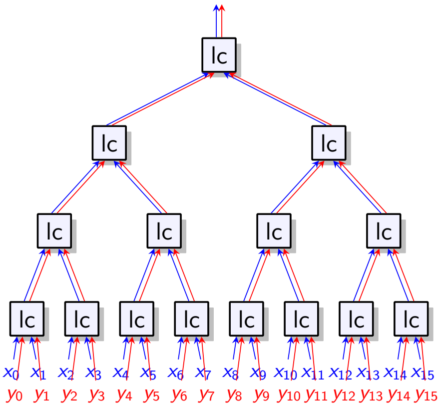
By the way, what is the cost of a comparator?

- Naive idea: subtract, then take the sign
- Recursive comparison tree:
  - split X into X_H (high bits) and X_L (low bits). Same for Y.
  - Then it is a lexicographic comparison (lc):

X lc Y	$X_H < Y_H$	$X_H = Y_H$	$X_H > Y_H$
$X_L < Y_L$	$X < Y$	$X < Y$	$X > Y$
$X_L = Y_L$	$X < Y$	$X = Y$	$X > Y$
$X_L > Y_L$	$X < Y$	$X > Y$	$X > Y$

  - split X_H and X_L recursively
  - The good binary encoding for this table is the one we have for free at the leaves:

case	$x_i < y_i$	$x_i = y_i$	$x_i > y_i$
encoding	01	00 or 11	10



In summary, for a comparator of n-bit numbers

time is $\sim \log_2 n$, area is $\sim n$

This talk would not be complete without some FPGA hacking

FPGA architecture for dummies

- the **basic gate** is any 5-input truth table
- ... complemented by ripple-carry addition is comparatively 30x faster than in VLSI

This talk would not be complete without some FPGA hacking

FPGA architecture for dummies

- the **basic gate** is any 5-input truth table
- ... complemented by ripple-carry addition is comparatively 30x faster than in VLSI

So

- back to the subtractor solution then
- but $n/2$ LUTs are enough:
  - split X and Y into 2-bit chunks X_i and Y_i
  - each LUT compress one X_i and one Y_i into just two bits that compare the same
  - then these two bits are subtracted using the fast-carry logic

This talk would not be complete without some FPGA hacking

FPGA architecture for dummies

- the **basic gate** is any 5-input truth table
- ... complemented by ripple-carry addition is comparatively 30x faster than in VLSI

So

- back to the subtractor solution then
- but $n/2$ LUTs are enough:
  - split X and Y into 2-bit chunks X_i and Y_i
  - each LUT compress one X_i and one Y_i into just two bits that compare the same
  - then these two bits are subtracted using the fast-carry logic

And if you are comparing to a constant

- on VLSI the tree simplifies itself
- on FPGAs X may be split into 5-bit chunks, so area is at most $\lceil n/5 \rceil$ LUTs.

No results yet

Executive summary

Area should be in $2^p \times n$ instead of $2^n \times p$ for the naive table.
Success.

- There is some trade-off space between the sequential and the parallel architecture.
- What if the function is not monotonic?

(joint work with Pierrick Joseph and Martin Kumm)

Oh and by the way

All this should work just the same for the small **floating-point** machine learning formats:

- FP numbers are ordered as their binary representation (almost)
- The only difference is the placement of the B_i on the function plot.

Oh and by the way

All this should work just the same for the small **floating-point** machine learning formats:

- FP numbers are ordered as their binary representation (almost)
- The only difference is the placement of the B_i on the function plot.

Tensor operators: sum of products in some fixed-point, then conversion back to float, then some function.

Generic optimization techniques for generic function generators

Generic optimization techiques

	ILP (CPLEX, Gurobi)	**CP** (ORTools, Choco)	**SAT** (ORTools, Gecode)
Variables	Integers / Binary	Finite domains	Boolean only
Constraints	Linear expressions	General: logical, global constraints	Logical clauses (CNF)
Objective	Optimization (min / max)	Satisfaction, optimization, enumeration	
Modeling	Laborious for combinatorial problems	Highly expressive (logic, scheduling)	Requires translation to Boolean logic
Solvers	Often copyright protected	Always license-free (maintained by community)	
Strengths	Powerful for numerical optimization	Very flexible for complex constraints	Extremely fast on Boolean problems
Limitations	Curse of linearization and numerical instabilities	Badly handles arithmetic constraints	Poorly handles counting

Generic optimization techiques

Exhaustive enumeration (a.k.a. brute force), or
mathematical programming:

- Integer linear programming (and variants thereof)
- SAT
- **SMT** (SAT modulo theory)
- CP

Generic optimization techiques

Exhaustive enumeration (a.k.a. brute force), or
mathematical programming:

- Integer linear programming (and variants thereof)
- SAT
- **SMT** (SAT modulo theory)
- CP

Motivations

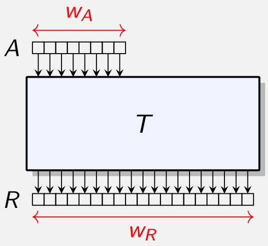
- Problem is NP-complete
- Replace tinkering and heuristics will well-founded optimization
- Describe the problem, not the algorithm that solves it
- Integrate approximation errors and rounding errors

Brute force

Lossless Differential Table Compression

A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...

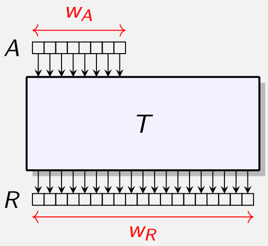


Size $2^{w_A} \times w_R$ bits

Lossless Differential Table Compression

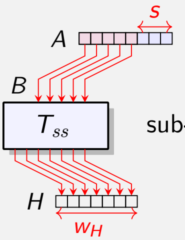
A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...



Size $2^{w_A} \times w_R$ bits

... can sometimes be compressed



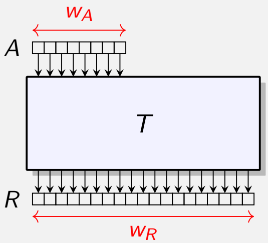
sub-sampling table

Size $2^{w_A - s} \times w_H$ bits

Lossless Differential Table Compression

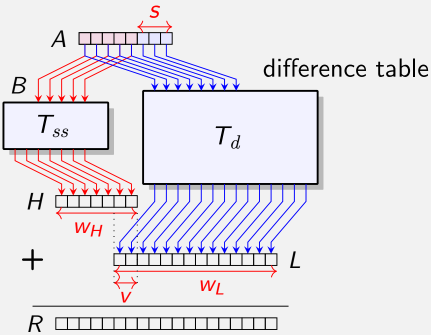
A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...



Size $2^{w_A} \times w_R$ bits

... can sometimes be compressed

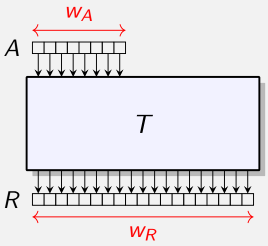


Size $2^{w_A - s} \times w_H + 2^{w_A} \times w_L$ bits

Lossless Differential Table Compression

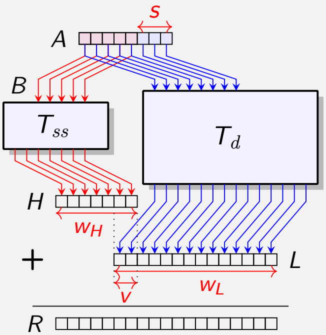
A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...



Size $2^{w_A} \times w_R$ bits

... can sometimes be compressed



Size $2^{w_A - s} \times w_H + 2^{w_A} \times w_L$ bits

No approximation! This is a lossless compression.

A candidate for the most inelegant algorithm award

Algorithm: Generic LDTC optimization

function `optimizeLDTC`(T, w_A, w_R)
  bestVector $\leftarrow (0, w_R, 0)$; `// no compression`

  forall (s, w_H, w_L); `// enumerate all the possible parameter values`
  do

  end forall
  return *bestVector*

A candidate for the most inelegant algorithm award

Algorithm: Generic LDTC optimization

function `optimizeLDTC`(T, w_A, w_R)

  bestVector $\leftarrow (0, w_R, 0)$; `// no compression`

  forall (s, w_H, w_L); `// enumerate all the possible parameter values`

   do

  end forall

  return *bestVector*

A candidate for the most inelegant algorithm award

Algorithm: Generic LDTC optimization

function `optimizeLDTC`(T, w_A, w_R)

  $bestVector \leftarrow (0, w_R, 0)$; `// no compression`

  $bestCost \leftarrow \mathrm{cost}(w_A, 0, w_R, 0)$;

  forall (s, w_H, w_L); `// enumerate all the possible parameter values`

  do

    $cost \leftarrow \mathrm{cost}(w_A, s, w_H, w_L)$; `// cost can be #bits or FPGA cost`

  end forall

  return $bestVector$

A candidate for the most inelegant algorithm award

Algorithm: Generic LDTC optimization

```
function optimizeLDTC(T, w_A, w_R)
    bestVector ← (0, w_R, 0) ;                       // no compression
    bestCost ← cost(w_A, 0, w_R, 0) ;
    forall (s, w_H, w_L);    // enumerate all the possible parameter values
     do
        cost ← cost(w_A, s, w_H, w_L);           // cost can be #bits or FPGA cost
        if cost < bestCost then
            if isValid(T, w_A, w_R, s, w_H, w_L) then
                bestCost ← cost;
                bestVector ← (s, w_H, w_L);
            end if
        end if
    end forall
    return bestVector
```

The `isValid` function is also brute force

Algorithm: Is a parameter vector valid?

function `isValid`$(T, w_A, w_R, s, w_H, w_L)$

- **for** $B \in (0, 1, \ldots, 2^{w_A - s} - 1)$; `// loop on slices`
  do
  - $S \leftarrow \{T[j]\}_{j \in \{B \cdot 2^s \ldots (B+1) \cdot 2^s - 1\}}$; `// slice`
  - $M \leftarrow \max(S)$; `// max on slice`
  - $m \leftarrow \min(S)$; `// min on slice`
  - $mask \leftarrow 2^{w_R - w_H} - 1$;
  - $H \leftarrow m - (m \,\&\, mask)$; `//` w_H `upper bits of` m
  - $M_{\text{low}} \leftarrow M - H$; `// max diff value on this slice`
  - **if** $M_{low} \geq 2^{w_L}$ **then**
    - **return** *false* ; `// this slice won't fit: exit with false`

    end if

  end for
- **return** *true*

Shameless but effective

Loop nest of depth 4 but...

- a lot of trivial branch cutting
- simple int64 computations inside
- all loop indices are small bit sizes (< 16)

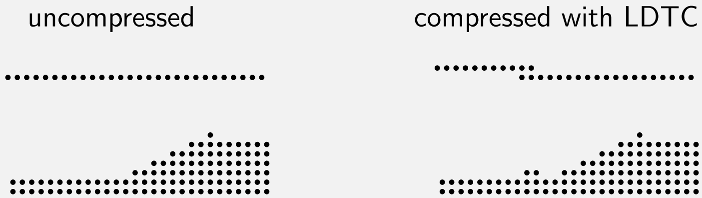
In short: it works when tables make sense.

Savings

-20% to -60% bits in tables from various existing FloPoCo operators.

What about cost of addition?

Probably negates the compression... except if the table is one input to a bit heap (compressor tree).

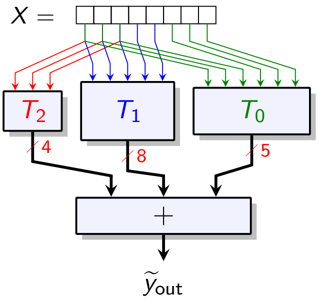


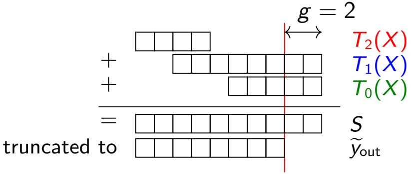
Dot-diagrams for the 24-bit multipartite implementation of $\sin(\frac{\pi}{4}x)$

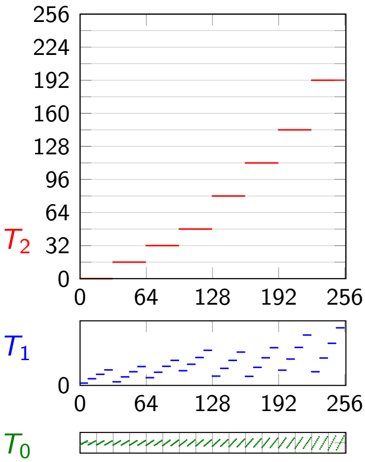
ILP

Multipartite tables: Orégane showed this yesterday

My first hardware arithmetic paper





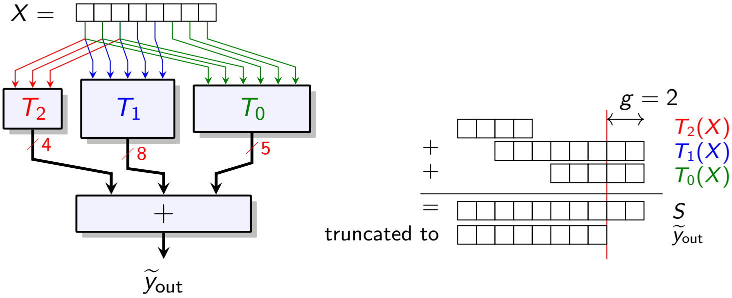


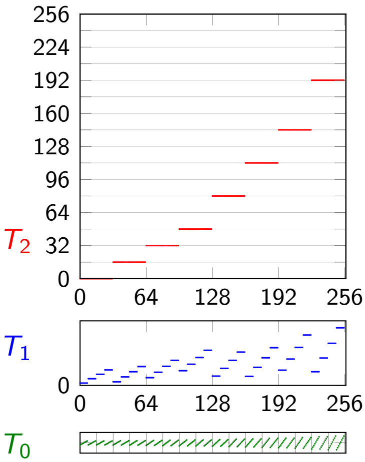
Multipartite tables: Orégane showed this yesterday

My first hardware arithmetic paper

with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



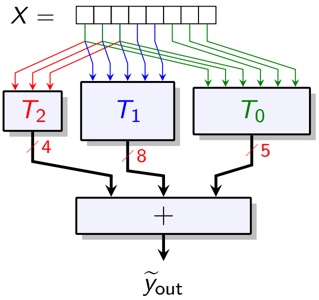


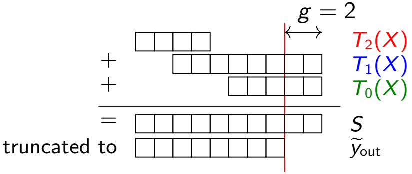
Multipartite tables: Orégane showed this yesterday

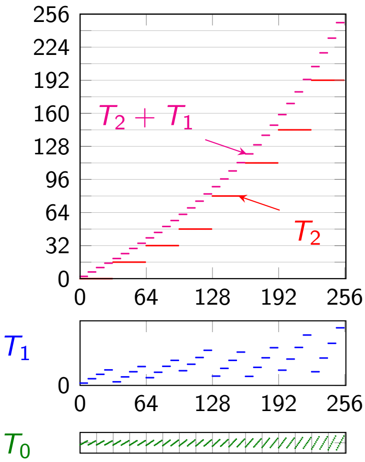
My first hardware arithmetic paper

with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*





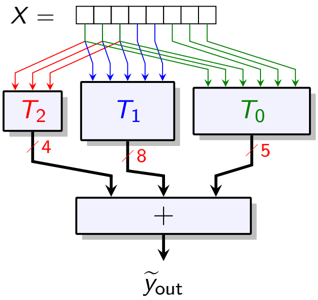


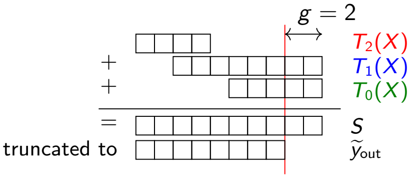
Multipartite tables: Orégane showed this yesterday

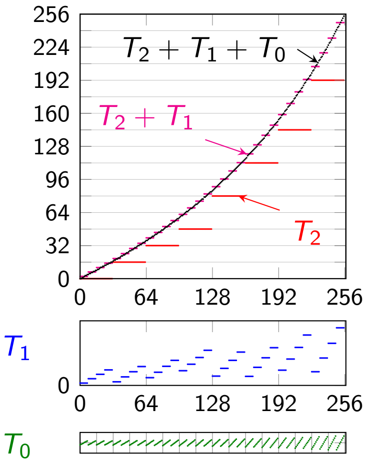
My first hardware arithmetic paper

with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*





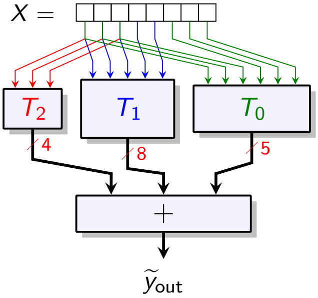


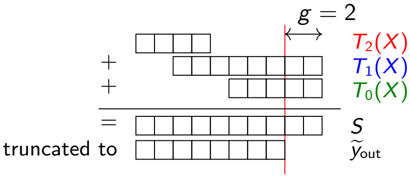
Multipartite tables: Orégane showed this yesterday

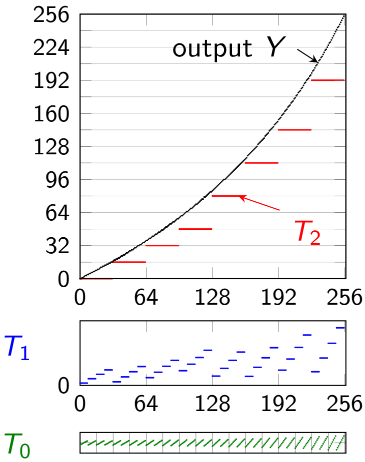
My first hardware arithmetic paper

with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*





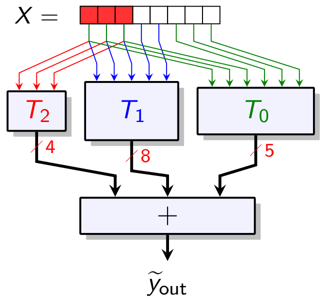


Multipartite tables: Orégane showed this yesterday

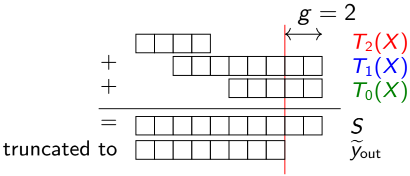
My first hardware arithmetic paper

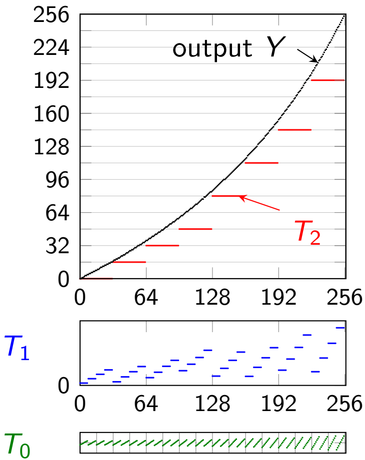
with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



$2^3 \cdot 4$



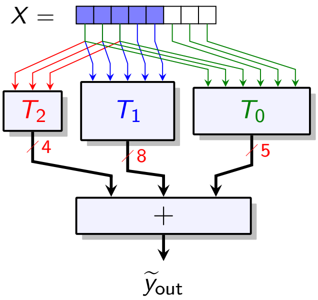


Multipartite tables: Orégane showed this yesterday

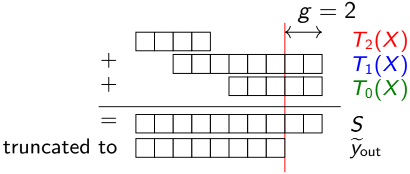
My first hardware arithmetic paper

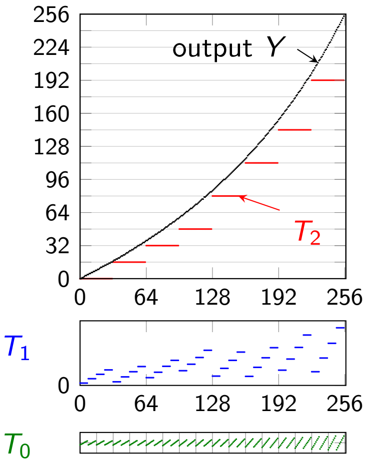
with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



$2^3 \cdot 4 + 2^5 \cdot 8$



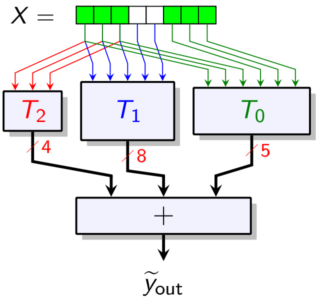


Multipartite tables: Orégane showed this yesterday

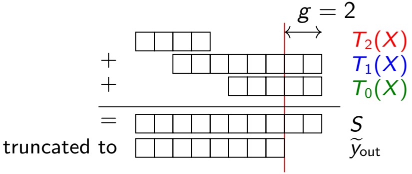
My first hardware arithmetic paper

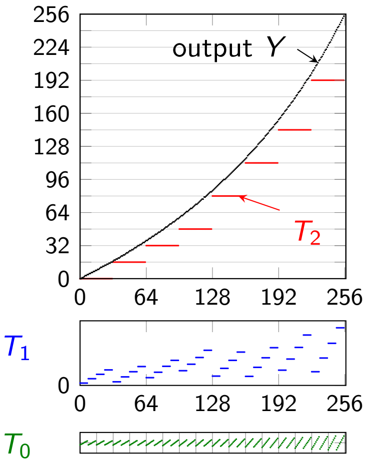
with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5$



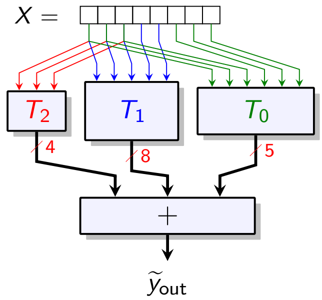


Multipartite tables: Orégane showed this yesterday

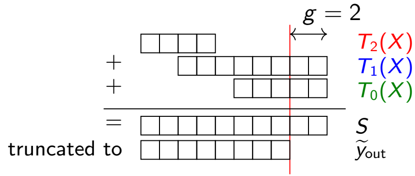
My first hardware arithmetic paper

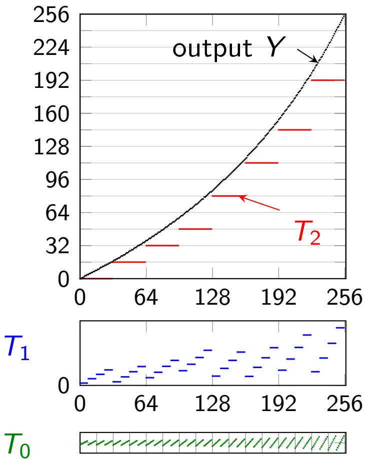
with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



$$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5 < 2^8 \cdot 8$$



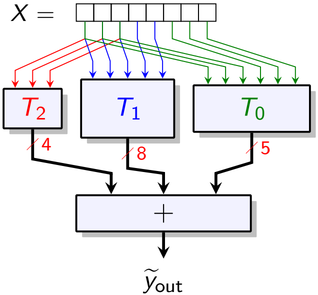


Multipartite tables: Orégane showed this yesterday

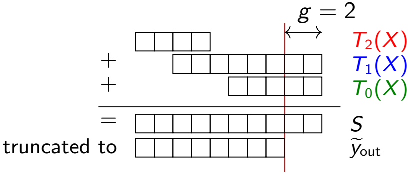
My first hardware arithmetic paper

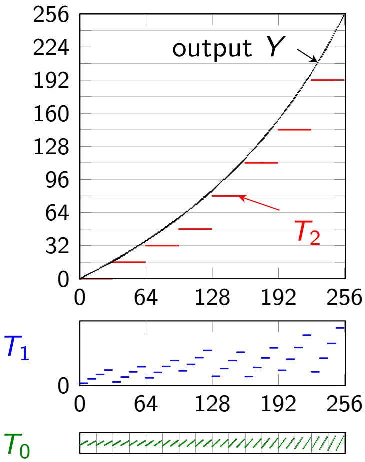
with A. Tisserand, in Jean-Michel's team !

Abstract: *Jean Michel published a matematical heuristic to find a multipartite architecture. We replace it with a brute force approach that works better.*



$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5 < 2^8 \cdot 8$





Then this 2022 FPT paper with Orégane: use ILP to remove the last traces of maths.

The idea came from this 2017 paper

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS 1

Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding

Davide De Caro, *Senior Member, IEEE*, Ettore Napoli, *Senior Member, IEEE*, Darjn Esposito, Gerardo Castellano, Nicola Petra, *Member, IEEE*, and Antonio G. M. Strollo, *Senior Member, IEEE*

***Abstract*— Piecewise polynomial interpolation is a well-established technique for hardware function evaluation. The paper describes a novel technique to minimize polynomial coefficients wordlength with the aim of obtaining either exact or faithful rounding at a reduced hardware cost. The standard approaches employed in literature subdivide the design of piecewise-polynomial interpolators into three steps (coefficients calculation, coefficients quantization and arithmetic hardware optimization) and estimate conservatively the overall approximation error as the sum of the error components arising in each step. The proposed technique, using Integer Linear Programming (ILP), optimizes the polynomial coefficients taking into account all error components simultaneously. This gives two advantages. Firstly, we can obtain exactly rounded approximations; secondly, for faithfully rounded interpolators, we avoid any overdesign due to pessimistic assumptions on error components, optimizing in this way the resulting hardware. The proposed ILP based algorithm requires an acceptable CPU time (from few seconds to tens of minutes) and is suited for approximations up to, maximum, 24 input bits. The results compare favorably with previously published data. We present synthesis results in 28 nm and 90 nm CMOS technologies, to further assess the effectiveness of the proposed approach.**

***Index Terms*— Arithmetic circuits, exact rounding, faithful rounding, hardware functions evaluation, polynomial approximation, truncated multiplier, VLSI systems.**

polynomial and rational approximations and table-based methods. Table-based methods can be further subdivided [11] into compute-bound methods, table-bound methods [12]–[13] and in-between methods [11], [14]–[23]. In between methods, also known as piecewise-polynomial approximations, are widely used, being a good compromise between LUT size and arithmetic circuitry complexity.

Let us consider the problem of evaluating a function $f(x)$ over an interval $[a, b]$ with a given precision requirement. The evaluation of $f(x)$ typically consists of three phases [10]: i) range reduction of the input to a predetermined interval $[a, b]$, ii) function approximation over the reduced range, iii) range reconstruction, expanding the result back to the original range. In this paper, we will focus on phase ii) above, investigating piecewise linear and piecewise-quadratic approximation of mathematical functions such as sine, cosine, exponential, logarithm, reciprocal, square root which are continuous with continuous derivatives in the interval of interest.

In a piecewise-polynomial approximation the interval $[a, b]$ is subdivided in T equal-length segments $[a_j, a_{j+1}]$ and a low-degree polynomial is employed in each segment to approximate the desired function. Polynomial coeffi

The idea came from this 2017 paper

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS 1

Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding

Davide De Caro, *Senior Member, IEEE*, Ettore Napoli, *Senior Member, IEEE*, Darjn Esposito,
Gerardo Castellano, Nicola Petra, *Member, IEEE*, and Antonio G. M. Strollo, *Senior Member, IEEE*

Overview

- Uniform piecewise splitting of the input interval in 2^α subintervals
- Degree 1 or 2 polynomial, evaluated in developed form
- Truncated multipliers and squarers
- 2^α ILP instances with $\sim 2^{n-\alpha}$ constraints each
- ... inside a few other loops
- Correct rounding! Our previous heuristics only addressed faithful rounding.

Scales further than Oregane's ILP... because many small ILPs instead of a big one.

approximation, truncated multiplier, VLSI systems.

SAT and SMT

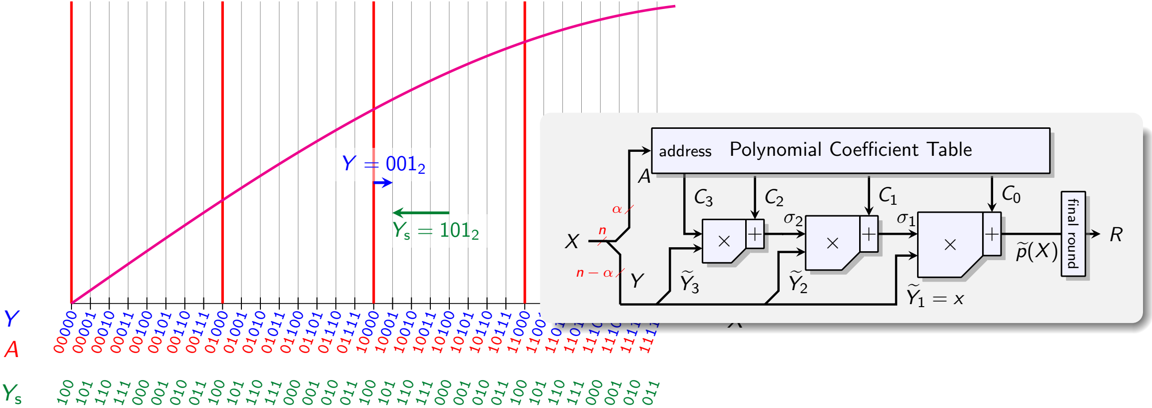
This slide unfortunately left blank

We end up doing: **space enumeration + optimization**.

Mathematical programming should be:
Space enumeration + satisfaction vs **Optimization**.

All these methods only give you the optimal in the space you explore.

Uniform piecewise segmentation of the input domain



Should the reduced argument be Y (unsigned) or Y_s (signed)?

Should the reduced argument be Y (unsigned) or Y_s (signed)?

- Range reduction with signed reduced arguments entails smaller polynomial coefficients:

Range reduction to unsigned:
$f_{u,10}(Y) = \exp(\frac{10}{16} + 2^{-4}Y)$
with $Y \in [0, 1]$

$$
\begin{aligned}
C_0 &= 1.1101111001000101011 \\
C_1 &= 0.0001110111100100010 \\
C_2 &= 0.0000000011101111011 \\
C_3 &= 0.0000000000000100111
\end{aligned}
$$

Range reduction to signed:
$f_{s,10}(Y_s) = \exp(\frac{10.5}{16} + 2^{-5}Y_s)$
with $Y_s \in [-1, 1]$

$$
\begin{aligned}
C_0 &= 1.1110110101110100000 \\
C_1 &= 0.0000111101101011101 \\
C_2 &= 0.0000000000111101101 \\
C_3 &= 0.0000000000000000101
\end{aligned}
$$

Here $f_{u,10}$ and $f_{s,10}$ cover the same subinterval of $\exp(x)$ and both polynomials are accurate to 10^{-6}.

Should the reduced argument be Y (unsigned) or Y_s (signed)?

- Range reduction with signed reduced arguments entails smaller polynomial coefficients:

Range reduction to unsigned:
$f_{u,10}(Y) = \exp(\frac{10}{16} + 2^{-4}Y)$
with $Y \in [0,1]$

$$\begin{aligned} C_0 &= 1.1101111001000101011 \\ C_1 &= 0.0001110111100100010 \\ C_2 &= 0.0000000011101111011 \\ C_3 &= 0.0000000000000100111 \end{aligned}$$

Range reduction to signed:
$f_{s,10}(Y_s) = \exp(\frac{10.5}{16} + 2^{-5}Y_s)$
with $Y_s \in [-1,1]$

$$\begin{aligned} C_0 &= 1.1110110101110100000 \\ C_1 &= 0.0000111101101011101 \\ C_2 &= 0.0000000000111101101 \\ C_3 &= 0.0000000000000000101 \end{aligned}$$

Here $f_{u,10}$ and $f_{s,10}$ cover the same subinterval of $\exp(x)$
and both polynomials are accurate to 10^{-6}.

- in developed form, symmetry may save one input bit to each term (e.g. $Y^2 = |Y|^2$).

Should the reduced argument be Y (unsigned) or Y_s (signed)?

- Range reduction with signed reduced arguments entails smaller polynomial coefficients:

Range reduction to unsigned:
$f_{u,10}(Y) = \exp(\frac{10}{16} + 2^{-4}Y)$
with $Y \in [0,1]$

$$
\begin{aligned}
C_0 &= 1.1101111001000101011 \\
C_1 &= 0.0001110111100100010 \\
C_2 &= 0.0000000011101111011 \\
C_3 &= 0.0000000000000100111
\end{aligned}
$$

Range reduction to signed:
$f_{s,10}(Y_s) = \exp(\frac{10.5}{16} + 2^{-5}Y_s)$
with $Y_s \in [-1,1]$

$$
\begin{aligned}
C_0 &= 1.1110110101110100000 \\
C_1 &= 0.0000111101101011101 \\
C_2 &= 0.0000000000111101101 \\
C_3 &= 0.0000000000000000101
\end{aligned}
$$

Here $f_{u,10}$ and $f_{s,10}$ cover the same subinterval of $\exp(x)$
and both polynomials are accurate to 10^{-6}.

- in developed form, symmetry may save one input bit to each term (e.g. $Y^2 = |Y|^2$).

(and by the way, for the functions above we are happy to have the Sollya parser again)

Conclusion

I'm busy until retirement with interesting questions!

A huge thanks to Jean-Michel for introducing me to Computer Arithmetic.