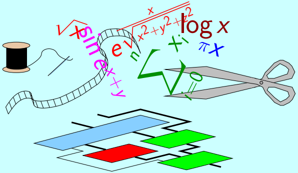
Computing Just Right: Application-Specific Arithmetic with FloPoCo



Florent de Dinechin

S. Banescu, L. Besème, N. Bonfante, N. Brunie, M. Christ, S. Collange, O. Desrentes, J. Detrey, P. Echeverría, F. Ferrandi, L. 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, A. Volkova.

citi lab

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Inria informatiques mathématiques

UNIVERSITÉ DE LYON

Outline

Bibliography

You can find articles on all these subjects on the web page of FloPoCo.
And a book to be published soon by Springer:
Application-Specific Arithmetic, by Florent de Dinechin and Martin Kumm.

Intro: arithmetic operators

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)
    - (even *DSP filters* are defined by a transfer function)

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)
    - (even *DSP filters* are defined by a transfer function)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))
  - Let's use the same approach for fixed-point operators, and non-standard ones

→ 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)
    - (even *DSP filters* are defined by a transfer function)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))
  - Let's use the same approach for fixed-point operators, and non-standard ones

→ 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)
    - (even *DSP filters* are defined by a transfer function)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))
  - Let's use the same approach for fixed-point operators, and non-standard ones

$\rightarrow$ 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...

FloPoCo, the user point of view

Here should come a demo

FloPoCo is freely available from

`http://flopoco.org/`

- Stable version 4.1.2: more operators
- git master version (will be 5.0): cleaner code, fewer operators
  - used in these slides (mostly)
  - several interface differences

Command line syntax

- a sequence of operator specifications
- each with many parameters
  - operator parameters (mandatory and optional)
  - global optional parameters: target frequency, target hardware, ...
- Output: synthesizable VHDL.

First something classical

A single precision floating-point adder

(8-bit exponent and 23-bit mantissa)

```
./flopoco FPAdd wE=8 wF=23
```

```
Final report:
|---Entity FPAdder_8_23_uid2_RightShifter
|---Entity IntAdder_27_f400_uid7
|---Entity LZCShifter_28_to_28_counting_32_uid14
|---Entity IntAdder_34_f400_uid17
Entity FPAdder_8_23_uid2
Output file: flopoco.vhdl
```

To probe further:

- `./flopoco FPAdd wE=11 wF=51`
  double precision
- `./flopoco FPAdd wE=9 wF=36`
  just right for you

Actually there are two variants

To get a larger but shorter-latency architectural variant:

```
./flopoco FPAdd wE=8 wF=23 dualpath=true
```

Here, `dualpath` is an optional performance option.
(different VHDL, same function)

Classical floating-point, continued

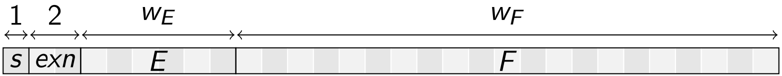
A complete single-precision FPU in a single VHDL file:

```
./flopoco FPAdd wE=8 wF=23 FPMult wE=8 wF=23 FPDiv wE=8 wF=23 FPSqrt wE=8
    wF=23
```

```
Final report:
|---Entity FPAdder_8_23_uid2_RightShifter
|---Entity IntAdder_27_f400_uid7
|---Entity LZCShifter_28_to_28_counting_32_uid14
|---Entity IntAdder_34_f400_uid17
Entity FPAdder_8_23_uid2
Entity Compressor_2_2
Entity Compressor_3_2
|   |---Entity IntAdder_49_f400_uid39
|---Entity IntMultiplier_UsingDSP_24_24_48_unsigned_uid26
|---Entity IntAdder_33_f400_uid47
Entity FPMultiplier_8_23_8_23_8_23_uid24
Entity FPDiv_8_23
Entity FPSqrt_8_23
Output file: flopoco.vhdl
```

Damn lies

It was not a classical single-precision FPU



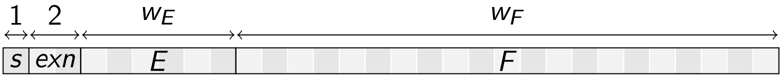
FloPoCo floating-point format

Inspired and compatible with IEEE-754, except that

- exponent size w_E and mantissa size w_F can take arbitrary values

Damn lies

It was not a classical single-precision FPU



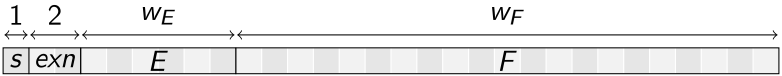
FloPoCo floating-point format

Inspired and compatible with IEEE-754, except that

- exponent size w_E and mantissa size w_F can take arbitrary values
- 0, ∞ and NaN flagged in 2 explicit *exception bits*: *exn*
  - not as special exponent values
  - (as a consequence, two more exponent values available in FloPoCo)

Damn lies

It was not a classical single-precision FPU



FloPoCo floating-point format

Inspired and compatible with IEEE-754, except that

- exponent size w_E and mantissa size w_F can take arbitrary values
- 0, ∞ and NaN flagged in 2 explicit *exception bits*: *exn*
  - not as special exponent values
  - (as a consequence, two more exponent values available in FloPoCo)
- subnormal numbers are not supported
  - Adding 1 more exponent bit provides them all, and is much more area-efficient
  - However we lose `a-b==0` $\iff$ `a==b`
    - HLS compiler writers, beware!
- Conversions operators from/to IEEE floating point available

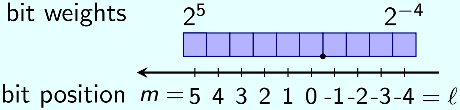
Number formats in FloPoCo

- Integers and fixed-point numbers
- The previous floating-point format
- A few operators for IEEE floating-point format
- A few operators for posits
- Logarithm Number System (LNS) in older versions
- One Obscure Branch contains decimal arithmetic
- no Residue Number System (RNS) and other modular arithmetic – waiting for them

... Plus good old binary fixed-point (integer) for quite a few operators

Fixed-point format

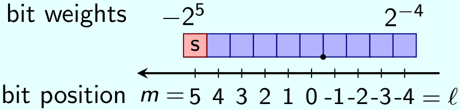
Parameters for an unsigned (positive) fixed-point format



$$X = \sum_{i=\ell}^{m} 2^i x_i$$

- m is the Most Significant Bit position, and determines the **range**
- ℓ is the Least Significant Bit position, and determines the **precision**

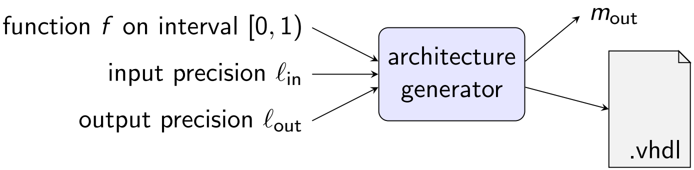
Parameters for a fixed-point format in two's complement



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

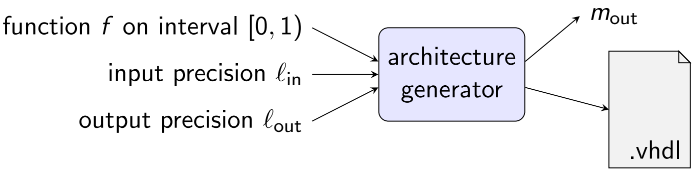
Integers have $\ell = 0, m > 0$.

Typical interface to a fixed-point FloPoCo operator

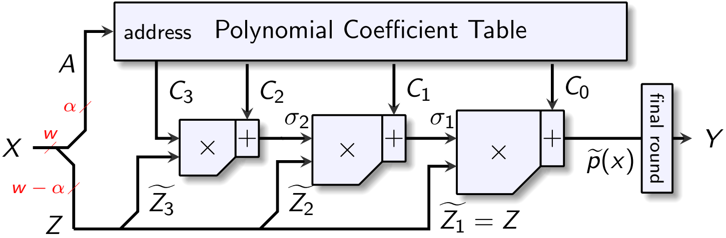



```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24
    msbOut=3 d=3
```

Typical interface to a fixed-point FloPoCo operator




```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24
    msbOut=3 d=3
```



Computing Just Right makes interfaces simpler

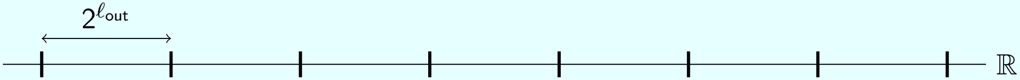
Never output bits that do not hold useful information:

Output precision (ℓ_{out}) specifies operator accuracy

- No need to compute more accurately than $2^{\ell_{\text{out}}}$: we couldn't output it
- No sense in computing less accurately than $2^{\ell_{\text{out}}}$:

  we don't want to output garbage bits

Correct rounding (IEEE-754): the best we can do with machine numbers



Computing Just Right makes interfaces simpler

Never output bits that do not hold useful information:

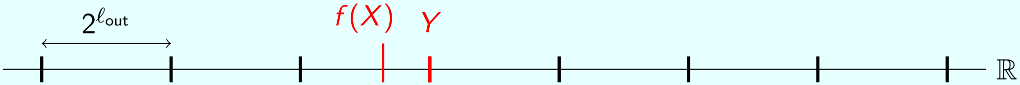
Output precision (ℓ_{out}) specifies operator accuracy

- No need to compute more accurately than $2^{\ell_{out}}$: we couldn't output it
- No sense in computing less accurately than $2^{\ell_{out}}$:

  we don't want to output garbage bits

Correct rounding (IEEE-754): the best we can do with machine numbers

Operator specification: return the number Y closest to the exact result $f(X)$



Computing Just Right makes interfaces simpler

Never output bits that do not hold useful information:

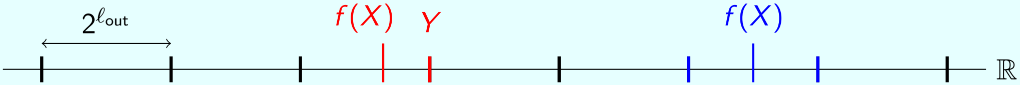
Output precision (ℓ_{out}) specifies operator accuracy

- No need to compute more accurately than $2^{\ell_{\text{out}}}$: we couldn't output it
- No sense in computing less accurately than $2^{\ell_{\text{out}}}$:

  we don't want to output garbage bits

Correct rounding (IEEE-754): the best we can do with machine numbers

Operator specification: return the number Y closest to the exact result $f(X)$

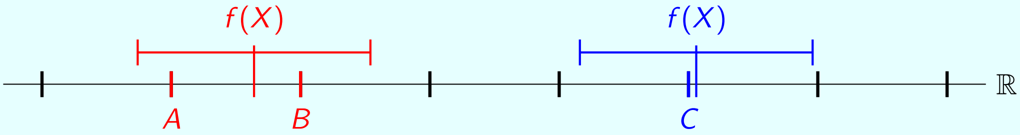


The difference between the computed value Y and $f(X)$ will be at most $2^{\ell_{\text{out}}-1}$.

It would be too simple, people would complain

Sometimes correct rounding is to expensive to implement, or just impossible to guarantee...

Faithful rounding: the next best thing



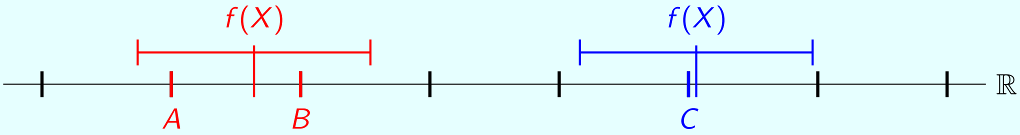
Two equivalent specifications:

- The output Y of the operator may be one of the two numbers surrounding $f(X)$. When $f(X)$ is a machine number, then $Y = f(X)$.
- The difference between the output value Y and $f(x)$ is strictly smaller than $2^{\ell_{\text{out}}}$.

It would be too simple, people would complain

Sometimes correct rounding is to expensive to implement, or just impossible to guarantee...

Faithful rounding: the next best thing



Two equivalent specifications:

- The output Y of the operator may be one of the two numbers surrounding $f(X)$. When $f(X)$ is a machine number, then $Y = f(X)$.
- The difference between the output value Y and $f(x)$ is strictly smaller than $2^{\ell_{out}}$.

Slightly less accurate than correct rounding, but still:

if you add one bit to the output, you double the accuracy.

Binary for theoretical physicists

- $2^{10} \approx 10^3$ (kBytes are actually 1024 bytes).
- Another point of view : $10\log_{10}(2) \approx 3$
- In other words, 1 bit $\approx$ 3 dB

I don't count signal/noise ratio in dB, I count accuracy in bits.

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

... but better because *compositional*

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

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

... but better because *compositional*

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

Remark: automatic pipeline framework improved from version 4 to (future) version 5, but all the operators need to be ported.

Examples of pipeline

```
./flopoco frequency=400 FPAdd wE=8 wF=23
```

```
Final report:
|---Entity FPAdder_8_23_uid2_RightShifter
|      Pipeline depth = 1
|---Entity IntAdder_27_f400_uid7
|      Pipeline depth = 1
|---Entity LZCShifter_28_to_28_counting_32_uid14
|      Pipeline depth = 4
|---Entity IntAdder_34_f400_uid17
|      Pipeline depth = 1
Entity FPAdder_8_23_uid2
   Pipeline depth = 9
```

```
./flopoco frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 4
```

Of course the frequency depends on the target FPGA

```
./flopoco target=Zynq7000 frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 5
```

```
./flopoco target=VirtexUltrascalePlus frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 1
```

Altera and Xilinx targets supported in the stable branch (at various levels of accuracy, in various versions): Spartan3, Zynq7000, Virtex4, Virtex5, Virtex6, Kintex7, VirtexUltrascalePlus, StratixII, StratixIII, StratixIV, StratixV, CycloneII, CycloneIII, CycloneIV, CycloneV.

Frequency-directed pipelining in practice

We do our best but we know it's hopeless

The actual frequency obtained will depend on the whole application (placement, routing pressure etc)...

- best-effort philosophy,
- aiming to be accurate to 10% for an operator synthesized alone
- asking a higher frequency provides a deeper pipeline

Frequency-directed pipelining in practice

We do our best but we know it's hopeless

The actual frequency obtained will depend on the whole application (placement, routing pressure etc)...

- best-effort philosophy,
- aiming to be accurate to 10% for an operator synthesized alone
- asking a higher frequency provides a deeper pipeline

And a big TODO: VLSI targets.

Also match the architecture to the target FPGA

Compare the VHDL produced with FloPoCo 4.1.2 for

```
flopoco target=Virtex4 IntConstDiv wIn=16 d=3
```

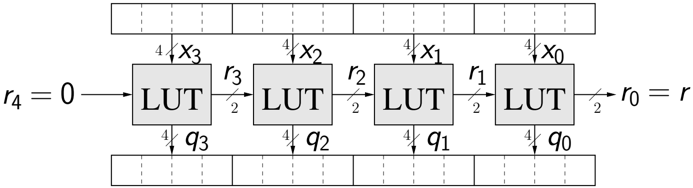
```
flopoco target=Virtex6 IntConstDiv wIn=16 d=3
```

Also match the architecture to the target FPGA

Compare the VHDL produced with FloPoCo 4.1.2 for

```
flopoco target=Virtex4 IntConstDiv wIn=16 d=3
```

```
flopoco target=Virtex6 IntConstDiv wIn=16 d=3
```

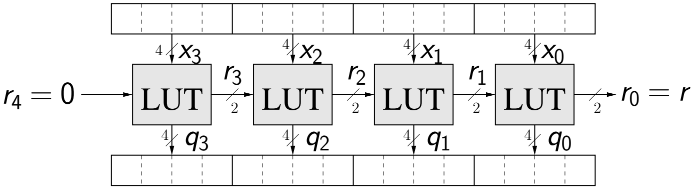


Also match the architecture to the target FPGA

Compare the VHDL produced with FloPoCo 4.1.2 for

```
flopoco target=Virtex4 IntConstDiv wIn=16 d=3
```

```
flopoco target=Virtex6 IntConstDiv wIn=16 d=3
```



Architecture specificities

- LUTs
- DSP blocks
- memory blocks

Non-standard operators

- Correctly rounded divider by 3:

```
flopoco FPConstDiv wE=8 wF=23 d=3
```

- Floating-point exponential:

```
flopoco FPExp wE=8 wF=23
```

- Multiplication of a 32-bit signed integer by the constant 1234567 (two algorithms, your mileage may vary):

```
flopoco IntIntKCM
```

```
flopoco IntConstMult
```

Full list in the documentation, or by typing just

```
flopoco
```

Sorry for the sometimes incomplete or inconsistent interface.

Don't trust us

`TestBench` generates a test bench for the operator preceding it on the command line

- `flopoco FPExp wE=8 wF=23 TestBench n=10000`
  generates 10000 random tests
- `flopoco IntConstDiv wIn=16 d=3 TestBench`
  generates an exhaustive test

Don't trust us

`TestBench` generates a test bench for the operator preceding it on the command line

- `flopoco FPExp wE=8 wF=23 TestBench n=10000`
  generates 10000 random tests
- `flopoco IntConstDiv wIn=16 d=3 TestBench`
  generates an exhaustive test

Specification-based test bench generation

Not by simulation of the generated architecture!

Don't trust us

`TestBench` generates a test bench for the operator preceding it on the command line

- `flopoco FPExp wE=8 wF=23 TestBench n=10000`
  generates 10000 random tests
- `flopoco IntConstDiv wIn=16 d=3 TestBench`
  generates an exhaustive test

Specification-based test bench generation

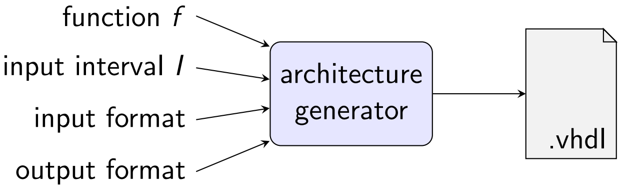
Not by simulation of the generated architecture!

Helper functions for encoding/decoding FP format, if you want to check the testbench...

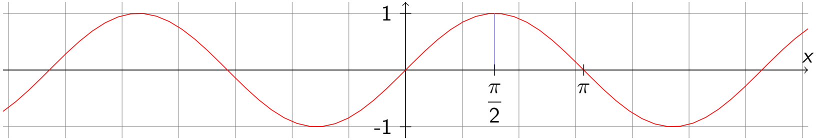
- `fp2bin 9 36 3.1415926`
- `bin2fp 9 36 010100000000100100100001111110110100110100010011`

Example: fixed-point functions

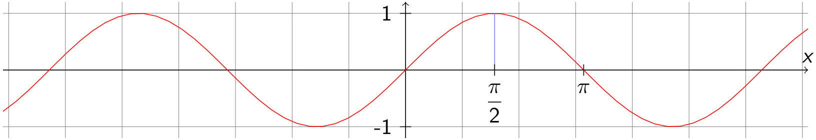
Generic generator of fixed-point functions



The sine function



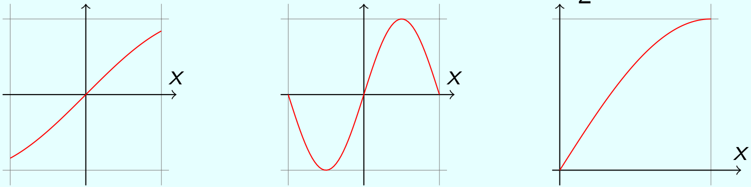
The sine function



Input format is in fixed point

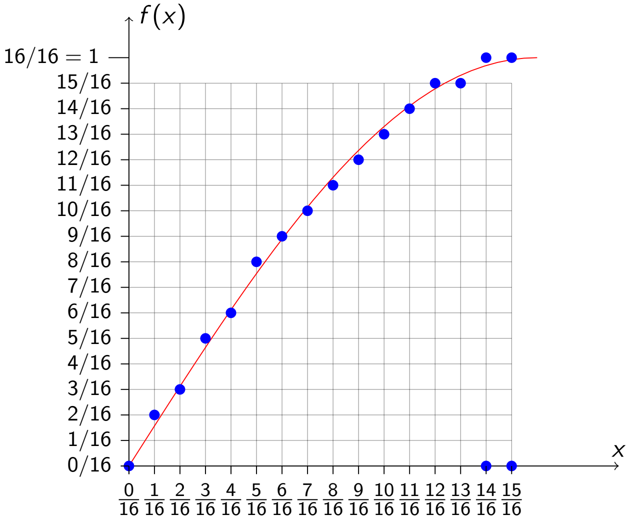
Arbitrary choice in FloPoCo: the input domain will be $[0,1)$ or $[-1,1)$.

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

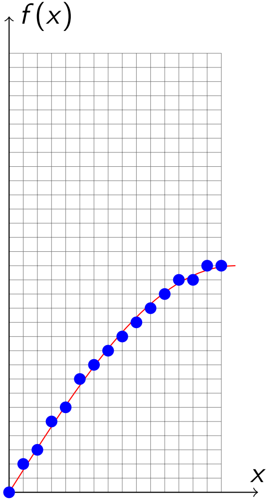


Discretization issues

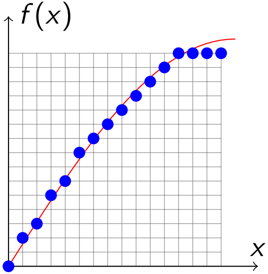
Inputs and outputs in $[0, 1)$ (4-bit fixed-point) :



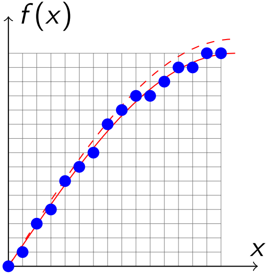
Possible fixes for corner-case discretization issues



Using 1 bit more



saturating

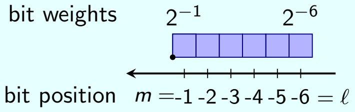


Scaling by $\frac{15}{16}$

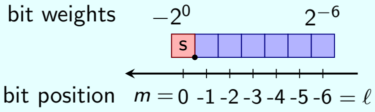
FixFunctionByTable

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

Input



Output



Go check in the VHDL which solution is used...
(Hint: remember that msbOut is computed.)

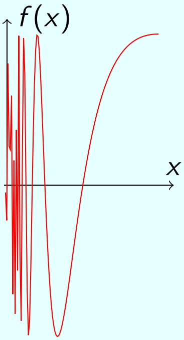
FixFunctionByTable, fixed

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

Go check the VHDL...

Tables can hold functions that are arbitrarily ugly

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




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

Tables scaling

The previous example was a 16-bit in, 16-bit out.

Tables scaling

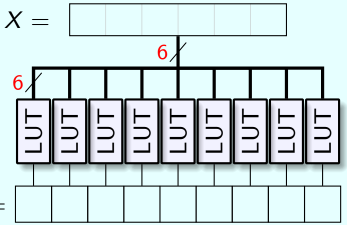
The previous example was a 16-bit in, 16-bit out.
(you just added 64 KLOC to your project)

Tables scaling

The previous example was a 16-bit in, 16-bit out.
(you just added 64 KLOC to your project)

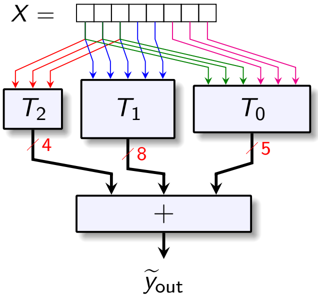
Practical sizes

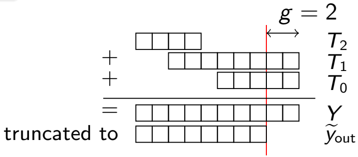
- The generated VHDL: $2^{-\texttt{lsbIn}}$ lines of `lsbOut` bits each
- LUT cost: $2^{-\texttt{lsbIn}-6} \times \texttt{lsbOut}$
- A table of $2^6 \times 6$ bits costs exactly 6 LUTs.

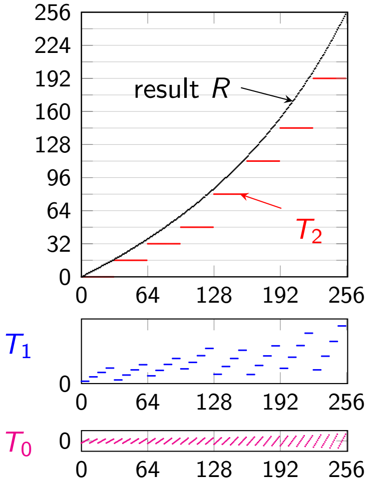


- A 20 Kb dual-port BlockRAM can hold two tables of $2^{10} \times 10$ bits.

FixFunctionByMultipartiteTable for 12 to 24 bits

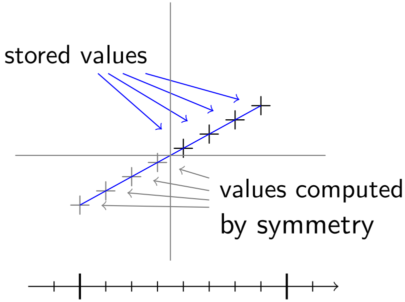


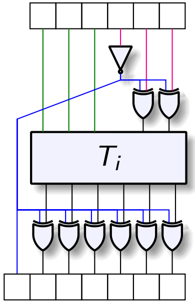




- rule of thumb: cost grows as $2^{p/2} \times p$ instead of $2^p \times p$
- but requires the function to be **continuous**, **derivable**, and even **monotonic**

One more trick: symmetry



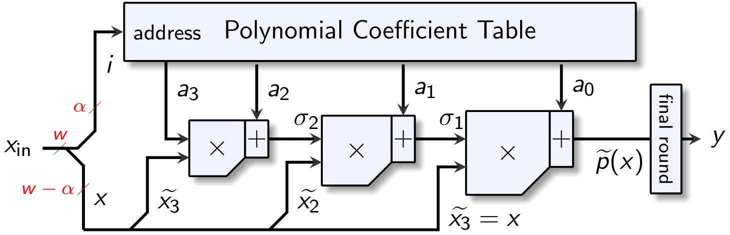


We exploit symmetry to trade one table input bit for two rows of XOR gates...

And above 16 bits...

A generic piecewise polynomial approximation method: `FixFunctionByPiecewisePoly`

- requires higher-order derivability, but scales to 64 bits.
- One more parameter: the *degree* of the polynomials, trades-off **memory** and **multipliers**



All these function evaluation methods have the same interface, you can swap one for another.

Example: multiplication and division by constants

Multiplication by a constant, first method

FPGA-specific LUT-based methods

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

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

  α bits

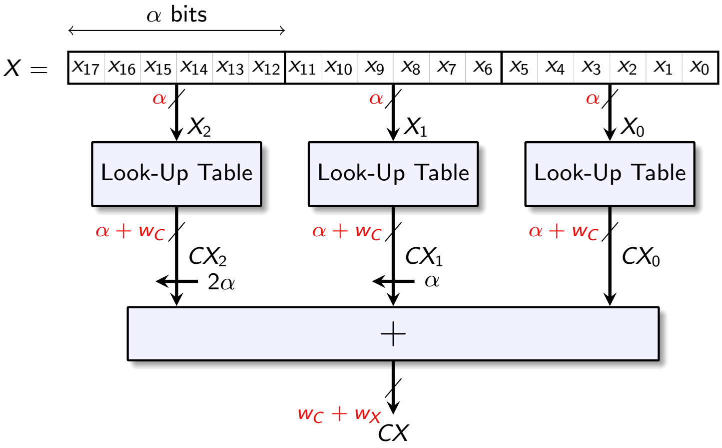
x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

  $X =$ (above)

- 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.

An architecture for 6-input LUTs



Multiplication by a constant, second method

Shift-and-add methods for integer constants

- $17x \quad = \quad 16x + x \quad = \quad (x \ll 4) + x$
- $15x \quad = \quad 16x - x$ (Booth recoding)
- $7697x \quad = \quad 15x \ll 9 \; + \; 17x$ (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, second method

Shift-and-add methods for integer constants

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

(demo?)

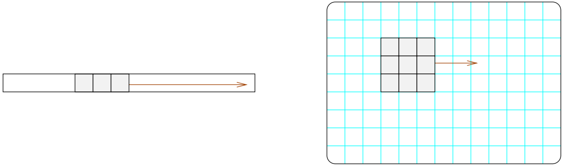
Extremely efficient for some constants such as 17.

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

Floating-point multiplication by a rational constant

Motivation

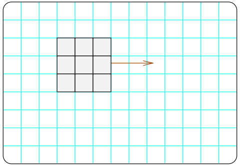
divisions by 3 and by 9 in stencil applications



Floating-point multiplication by a rational constant

Motivation

divisions by 3 and by 9 in stencil applications



$$1/3 = 0.0101010101010101010101010\cdots$$
$$1/9 = 0.000111000111000111000111\cdots$$

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)

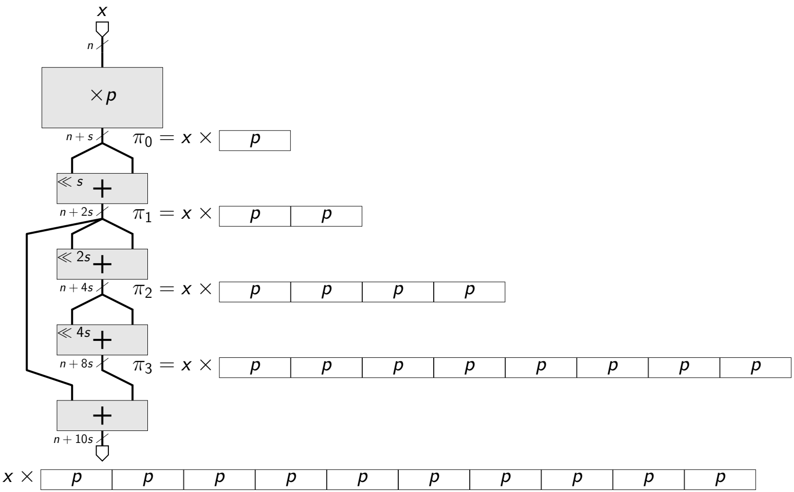
Then

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

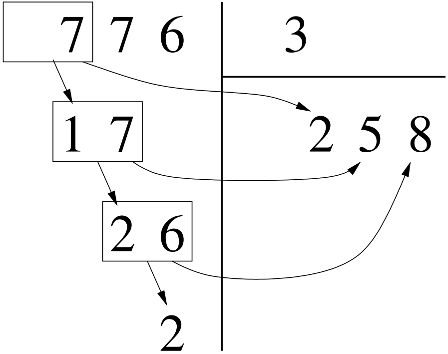
(The precisions chosen here are those of the IEEE754-2008 formats)

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

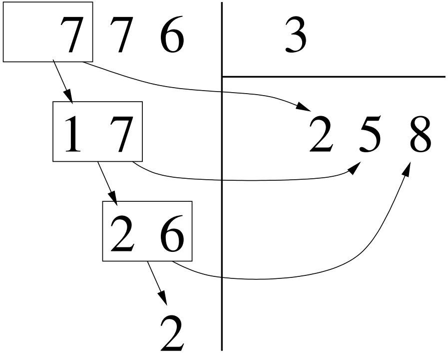
And now for something completely different

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

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

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

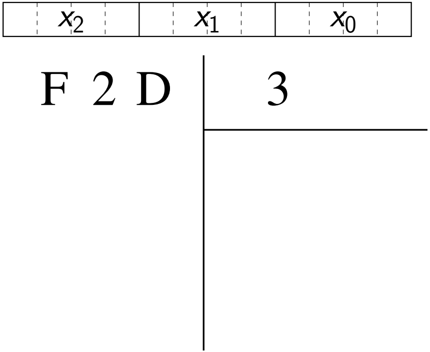
The same, but in binary-friendly radix

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 $\alpha = 4$



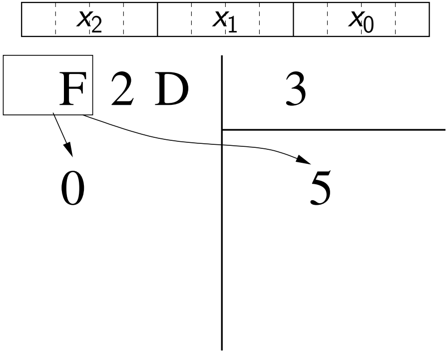
The same, but in binary-friendly radix

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 $\alpha = 4$

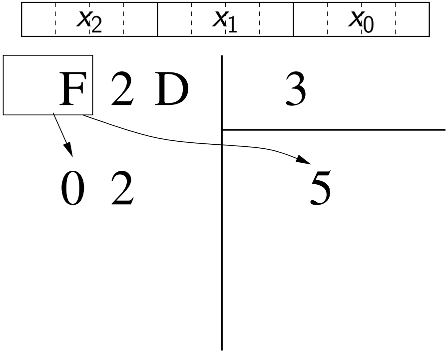


The same, but in binary-friendly radix

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 $\alpha = 4$



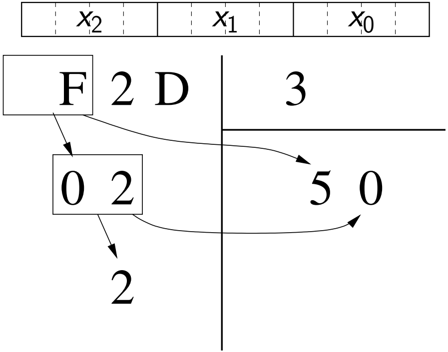
The same, but in binary-friendly radix

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 $\alpha = 4$

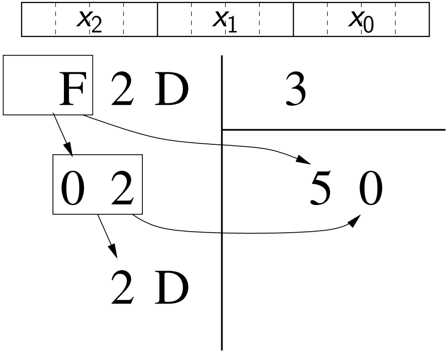


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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



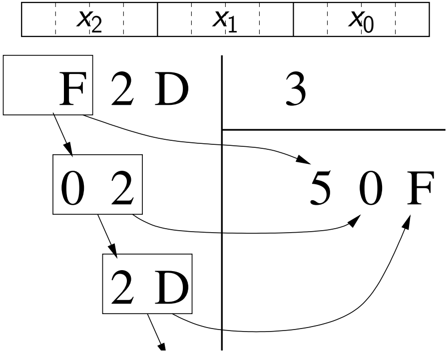
The same, but in binary-friendly radix

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 $\textsc{ConstantDiv}(x,\, d)$
$r_k \leftarrow 0$
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 $\textsc{ConstantDiv}(x,\ d)$
$r_k \leftarrow 0$
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

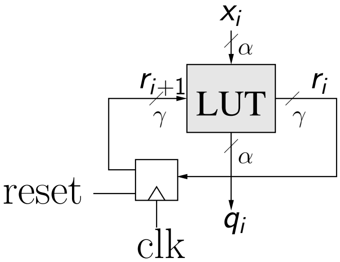
- consumes α bits of x, and a remainder of size $\gamma = \lceil \log_2 d \rceil$
- produces α bits of q, and a remainder of size γ
- implemented as a table with $\alpha + \gamma$ bits in, $\alpha + \gamma$ bits out

At this point nobody wants to see the proof

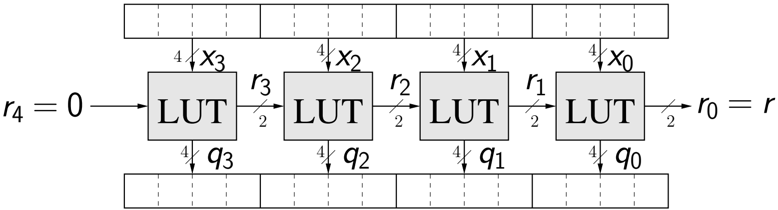
(if you're convinced the decimal version works...)

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

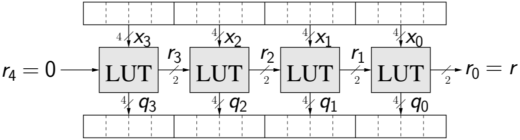
Sequential implementation



Unrolled implementation



Logic-based version

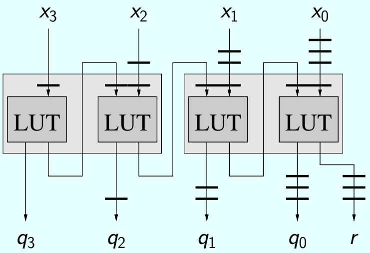


For instance, assuming a 6-input LUTs (e.g. LUT6)

- 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	LUT6	$n = 32$ bits (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	LUT6	$n = 64$ bits (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	LUT6 ($n = 32$ bits)	(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	LUT6 ($n = 64$ bits)	(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

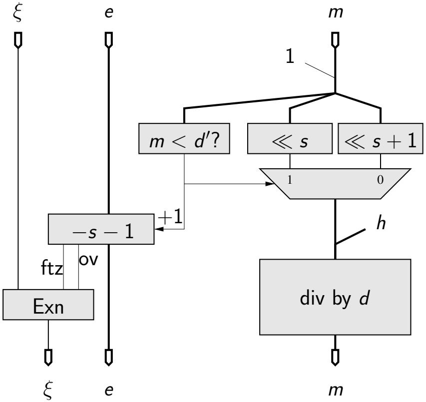
Logic optimizer even finds something to chew: *don't care* lines in the tables.

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:

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

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: FIR filters

Finite Impulse Response filters

$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

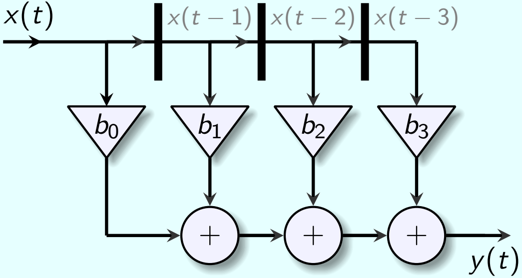
- the b_i are potentially real numbers (or almost: Matlab numbers)
- the $x(t)$ and $y(t)$ are discrete, fixed-point, low-precision signals

(the lower, the cheaper)

FIR filters, architectural view (abstract)

$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

Abtract architecture



FIR filters, arithmetic view

$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

$$\begin{array}{rl}
b_0 = & .0000100111111010001010101101\ldots \\
b_1 = & .0010111011000100010100111000\ldots \\
b_2 = & .1100000101101101000100110010\ldots \\
b_3 = & .0011010100000100111011100111\ldots \\
 & \\
b_0x_0 & \texttt{xxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_1x_1 & \texttt{xxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_2x_2 & \texttt{xxxxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_3x_3 & \texttt{xxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
 & \\
y = & \texttt{yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy}\ldots
\end{array}$$

The b_i are reals, therefore the exact result y may be an irrational.

FIR filters, arithmetic view

$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

$$\begin{aligned}
b_0 &= .00001001111111010001010 1\\
b_1 &= .00101110110001000101001 1\\
b_2 &= .11000001011011010001001 1\\
b_3 &= .00110101000001001110111 0
\end{aligned}$$

$b_0 x_0$	xxxxxxxxxxxxxxxxxxxx
$+b_1 x_1$	xxxxxxxxxxxxxxxxxxxxxx
$+b_2 x_2$	xxxxxxxxxxxxxxxxxxxxxxxx
$+b_3 x_3$	xxxxxxxxxxxxxxxxxxxxxx
$y =$	yyyyyyyyyyyyyyyyyyyyyyyyy
	2^{-p}

Naive approach: round the b_i and the products to the target precision.

FIR filters, arithmetic view

$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

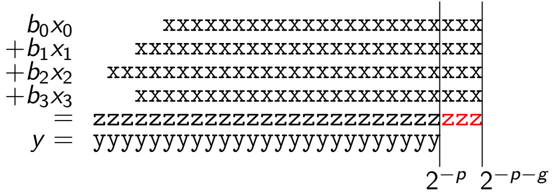
```
  b0 =  .000010011111110100010101
  b1 =  .001011101100010001010011
  b2 =  .110000010110110100010011
  b3 =  .001101010000010011101110

  b0x0        xxxxxxxxxxxxxxxxxxxx|
 +b1x1      xxxxxxxxxxxxxxxxxxxxxx|
 +b2x2    xxxxxxxxxxxxxxxxxxxxxxxx|
 +b3x3      xxxxxxxxxxxxxxxxxxxxxx|

   y =   yyyyyyyyyyyyyyyyyyyyyyyyy|
                                 2^-p
```

... but the accumulation of rounding errors makes the result inaccurate

FIR filters, arithmetic view

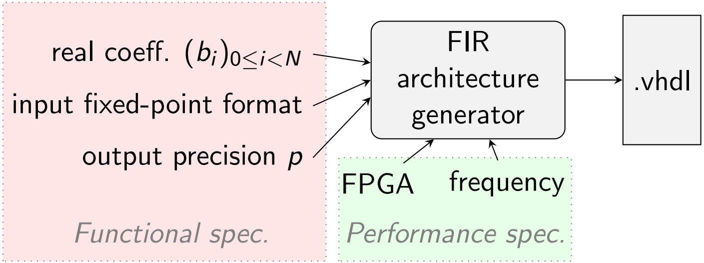
$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

$b_0 =$ .00001001111111010001010101101...
$b_1 =$ .00101110110001000101001110000...
$b_2 =$ .11000001011011010001001100101...
$b_3 =$ .00110101000001001110111001111...

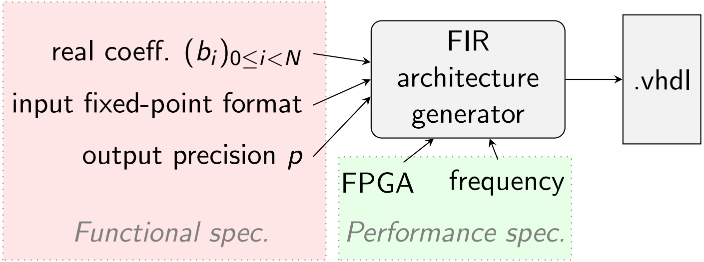


Proposed approach: last-bit-accurate architecture
with respect to the exact result

Really a matter of interface

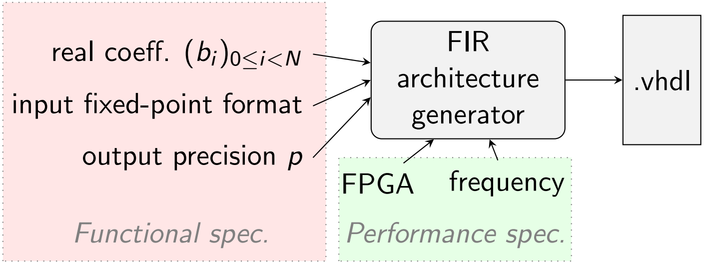


Really a matter of interface



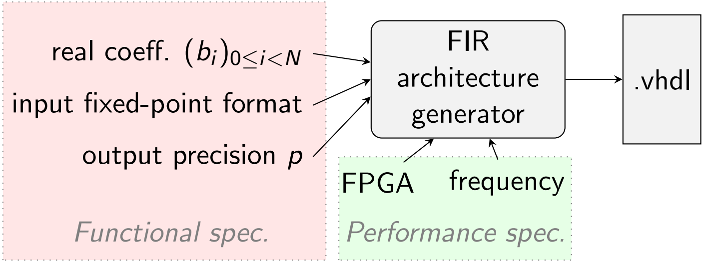
- Output precision defines accuracy of the architecture

Really a matter of interface



- Output precision defines accuracy of the architecture
- Accuracy defines the optimal precisions to be used internally

Really a matter of interface



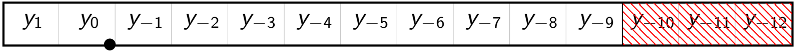
- Output precision defines accuracy of the architecture
- Accuracy defines the optimal precisions to be used internally

No point in computing more, no point in computing less

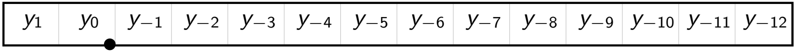
Example of the accuracy/cost tradeoff

8-tap, 12 bit Root-Raised Cosine FIR filters

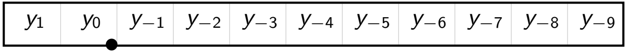
Naive, $p = 12$ 5.9 ns, 444 LUT $\overline{\epsilon} > 2^{-9}$



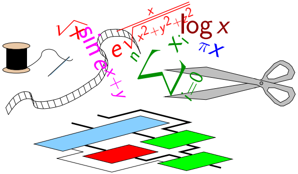
Proposed, $p = 12$ 4.4 ns, 564 LUT $\overline{\epsilon} < 2^{-12}$



Proposed, $p = 9$ 4.12 ns, 380 LUT $\overline{\epsilon} < 2^{-9}$



Demo



- Coefficients entered as math. formulae
- FPGA-specific optimizations
- Frequency-directed pipeline
- Test-driven design

... and all the other operators

Compute Just Right: Determining msb_o

$$
\begin{array}{rl}
a_0 = & .000010011111101000101010101101\ldots \\
a_1 = & .001011101100010001010011100000\ldots \\
a_2 = & .110000010110110100010011001 01\ldots \\
a_3 = & .001101010000010011101110011111\ldots
\end{array}
$$

```
  a0x0          xxxxxxxxxxxxxxxxxxxxxxxxx...
 +a1x1        xxxxxxxxxxxxxxxxxxxxxxxxxxx...
 +a2x2      xxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
 +a3x3        xxxxxxxxxxxxxxxxxxxxxxxxxxx...
--------------------------------------------
 y    =     yyyyyyyyyyyyyyyyyyyyyyyyyyyyyy...
```

The MSB of $a_i x_i$

- x_i bounded (fixed-point number)
- a_i known

$$msb_{a_i x_i} = \lceil \log_2(|a_i| val_{max}(x_i)) \rceil$$

The MSB of the sum

- $a_i x_i$ bounded

$$msb_o = msb_y = \lceil \log_2(\sum_{i=0}^{N-1} |a_i| val_{max}(x_i)) \rceil$$

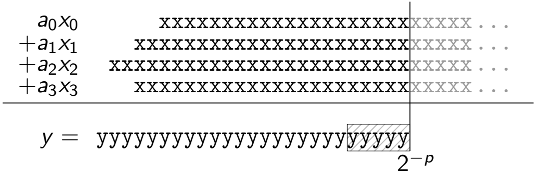
Compute Just Right: Determining the LSB

$a_0 =$ .00001001111111010001010101101 . . .
$a_1 =$ .00101110110001000101001110000 . . .
$a_2 =$ .11000001011011010001001100101 . . .
$a_3 =$ .00110101000001001110111001111 . . .

```
 a0x0        xxxxxxxxxxxxxxxxxxxx|xxxxx ...
+a1x1      xxxxxxxxxxxxxxxxxxxxxx|xxxxx ...
+a2x2    xxxxxxxxxxxxxxxxxxxxxxxx|xxxxx ...
+a3x3      xxxxxxxxxxxxxxxxxxxxxx|xxxxx ...
---------------------------------------------
  y =   yyyyyyyyyyyyyyyyyyyyyyyyy|
                                2^-p
```

Supose we use perfect multipliers: $\varepsilon_{mult} < 2^{-p-1}$

$$a_0 = .00001001111111010001010101101\ldots$$
$$a_1 = .00101110110001000101001110000\ldots$$
$$a_2 = .11000001011011010001001100101\ldots$$
$$a_3 = .00110101000001001110111001111\ldots$$

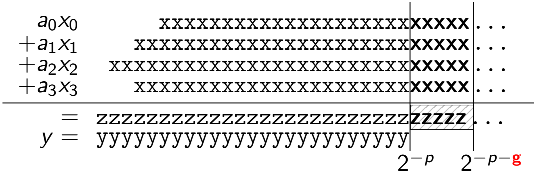


Supose we use perfect multipliers: $\varepsilon_{mult} < 2^{-p-1}$

- sum error: $\varepsilon_y = \sum_{i=0}^{N} \varepsilon_{mult} < N \cdot 2^{-p-1}$

Compute Just Right: Determining the LSB

$$
\begin{aligned}
a_0 &= .00001001111111010001010101101\ldots\\
a_1 &= .00101110110001000101001110000\ldots\\
a_2 &= .11000001011011010001001100101\ldots\\
a_3 &= .00110101000001001110111001111\ldots
\end{aligned}
$$



Supose we use perfect multipliers: $\varepsilon_{mult} < 2^{-p-1}$

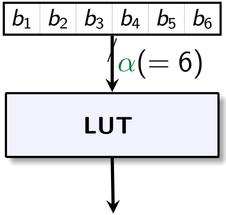
- sum error: $\varepsilon_{y_{total}} = \sum_{i=0}^{N} \varepsilon_{mult} + \varepsilon_{final\_rounding} < N \cdot 2^{-p-g-1} + 2^{-p-1}$

Need for larger intermediary precision

- **g guard bits**
- such that errors accumulate in the guard bits

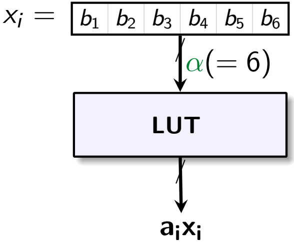
$$\Longrightarrow g = \lceil \log_2(N) \rceil$$

Perfect constant multipliers in an FPGA



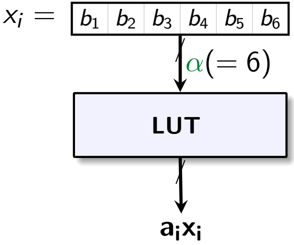
- basic FPGA computing element: look-up table (**LUT**)

Perfect constant multipliers in an FPGA



- basic FPGA computing element: look-up table (**LUT**)
- tabulate all the 2^α values of $\mathbf{a_i x_i}$
- ... correctly rounded to the output precision

Perfect constant multipliers in an FPGA



- basic FPGA computing element: look-up table (**LUT**)
- tabulate all the 2^α values of $\mathbf{a_i x_i}$
- ... correctly rounded to the output precision
- perfect fit for small sizes:

  α-input LUT + α-bit input $\Longrightarrow$ 1 LUT/output bit
- but doesn't scale:

  2 LUT/output bit for $(\alpha + 1)$-bit inputs,...

  2^k LUT/output bit for $(\alpha + k)$-bit inputs

KCM multipliers by real constants

$x_i =$

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}	b_{17}	b_{18}

d_{i1} d_{i2} d_{i3}

$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\}$$

KCM multipliers by real constants

$x_i =$	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}	b_{17}	b_{18}

d_{i1} $\quad d_{i2}$ $\quad d_{i3}$

$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\}$$

$$\Longrightarrow \mathbf{a_i x_i} = \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

KCM multipliers by real constants

$x_i =$

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}	b_{17}	b_{18}

d_{i1} d_{i2} d_{i3}

$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\}$$

$$\Longrightarrow \mathbf{a_i x_i} = \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

Each $a_i d_{ik}$ tabulated, 1 LUT/output bit

KCM multipliers by real constants

$x_i =$	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}	b_{17}	b_{18}
			d_{i1}						d_{i2}						d_{i3}			

$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^\alpha - 1\}$$

$$\Longrightarrow \mathbf{a_i x_i} = \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

Each $a_i d_{ik}$ tabulated, 1 LUT/output bit

How many output bits?

KCM multipliers by real constants

$x_i =$ | b_1 | b_2 | b_3 | b_4 | b_5 | b_6 | b_7 | b_8 | b_9 | b_{10} | b_{11} | b_{12} | b_{13} | b_{14} | b_{15} | b_{16} | b_{17} | b_{18} |

d_{i1} d_{i2} d_{i3}

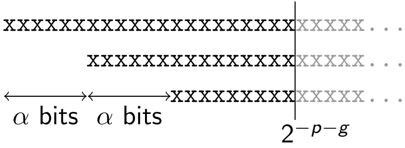
$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\}$$

$$\Longrightarrow \mathbf{a_i x_i} = \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

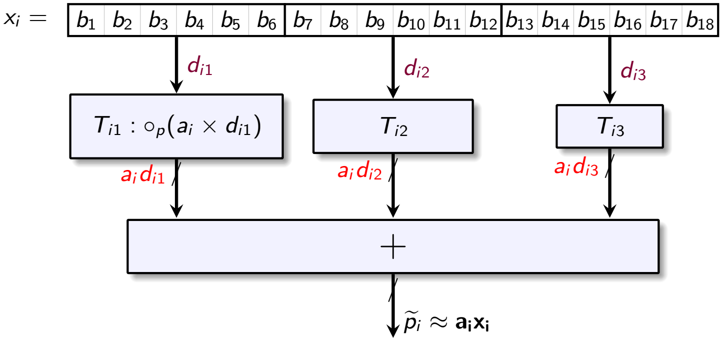
Each $a_i d_{ik}$ tabulated, 1 LUT/output bit

How many output bits?

$$\mathbf{a_i x_i} = a_i d_{i1} + 2^{-\alpha} a_i d_{i2} + 2^{-2\alpha} a_i d_{i3}$$



KCM multipliers by real constants



Summing it all up

$$y = \sum_{i=0}^{N-1} \mathbf{a_i x_i}$$

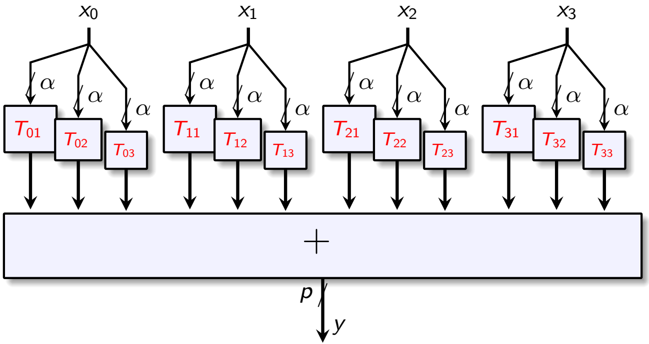
Summing it all up

$$y = \sum_{i=0}^{N-1} \mathbf{a_i x_i} \quad = \sum_{i=0}^{N-1} \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

Summing it all up

$$y = \sum_{i=0}^{N-1} \mathbf{a_i x_i} \quad = \sum_{i=0}^{N-1} \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

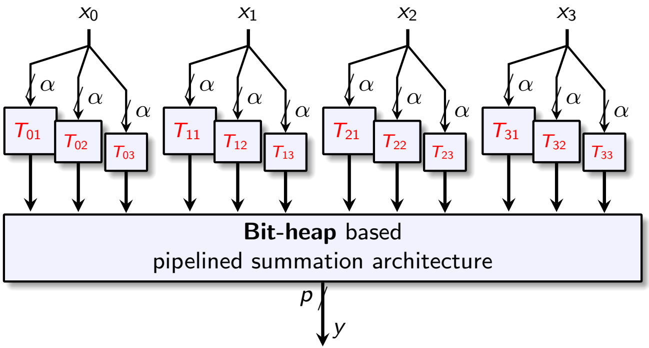
- each $a_i d_{ik}$ is a perfect multiplier
- therefore $g = \lceil \log_2(N \cdot n) \rceil$



Summing it all up

$$y = \sum_{i=0}^{N-1} \mathbf{a_i x_i} \quad = \sum_{i=0}^{N-1} \sum_{k=1}^{n} 2^{-k\alpha} a_i d_{ik}$$

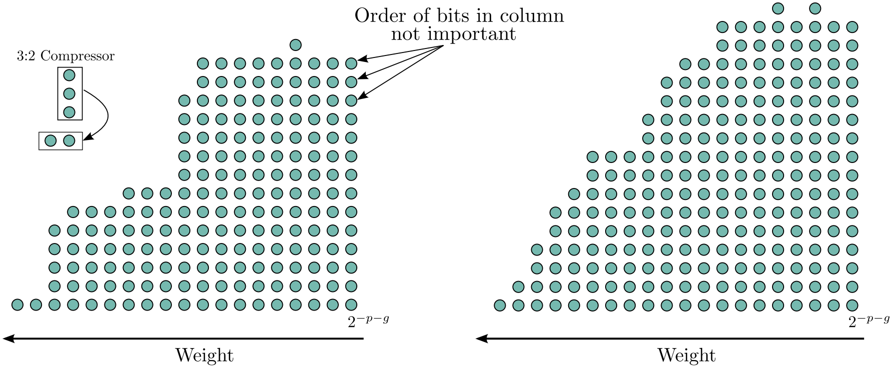
- each $a_i d_{ik}$ is a perfect multiplier
- therefore $g = \lceil \log_2(N \cdot n) \rceil$



Summing it all up

Bit-heaps (generalization of **bit arrays**) in FloPoCo
(see FPL 2013 article)

- 8-tap, 12-bit FIR filters



Half-Sine **Root-Raised Cosine**

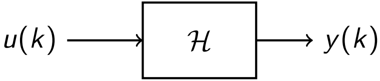
Work in progress

- Extension to IIRs done last year (with Paris VI and ENS-Lyon)
  - infinite accumulation of rounding errors: how many guard bits?
  - link with a trusted library computing the worst-case peak gain of a filter
- Address the combinatorics of filter realizations (with Paris VI)
- Filter approximation from frequency response (with ENS-Lyon)
  - Remez with an arithmetic focus

Example: IIR filters

Once upon a time in the green pastures of pure mathematics

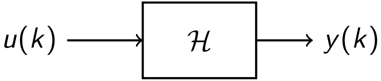
... there lived a handsome filter named $\mathcal{H}$



[1] This is a fairy tale, everybody knows Matlab does not compute with real numbers.

Once upon a time in the green pastures of pure mathematics

… there lived a handsome filter named $\mathcal{H}$



$\mathcal{H}$ was linear and time-invariant.
He was born in the distant Frequency Domain from a frequency specification, which the Matlab fairies had transformed into a transfer function:

$$\mathcal{H}(z) = \frac{\displaystyle\sum_{i=0}^{n_b} b_i z^{-i}}{1 + \displaystyle\sum_{i=1}^{n_a} a_i z^{-i}}, \quad \forall z \in \mathbb{C}.$$

whose coefficients (a_i) and (b_i) were real numbers[1].

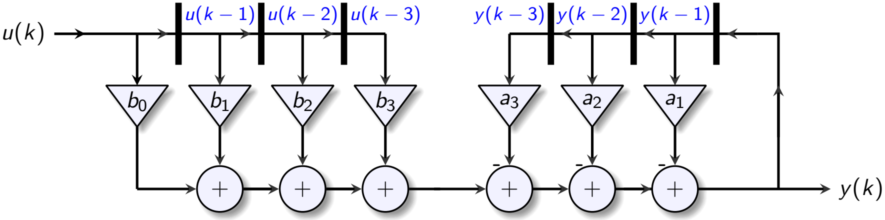
[1] This is a fairy tale, everybody knows Matlab does not compute with real numbers.

And so $\mathcal{H}$ converged beautifully

using its evaluation formula in the time domain

$$y(k) = \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i y(k-i)$$

as long as $\mathcal{H}$ remained safely linear and its poles safely within the unit circle.

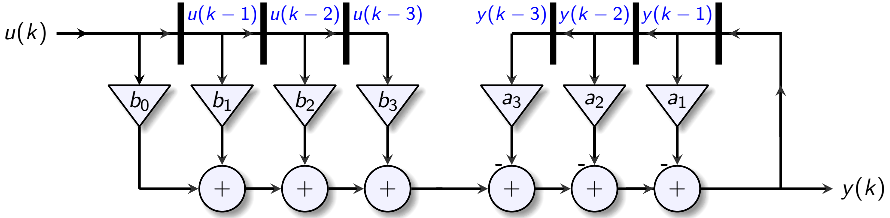


And so $\mathcal{H}$ converged beautifully

using its evaluation formula in the time domain

$$y(k) = \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i y(k-i)$$

as long as $\mathcal{H}$ remained safely linear and its poles safely within the unit circle.

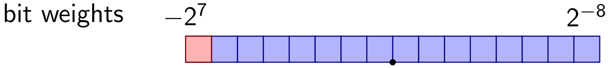


But the fairies had warned $\mathcal{H}$:

Don't let your poles come close to the unit circle! And above all, remain linear!

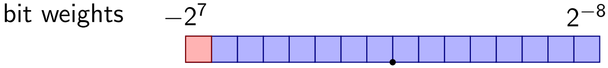
But one day, $\mathcal{H}$ decided to travel far from home

Our hero decided to visit the land of Digital Circuits, a rough and arid country where ony binary fixed-point numbers could live.



But one day, $\mathcal{H}$ decided to travel far from home

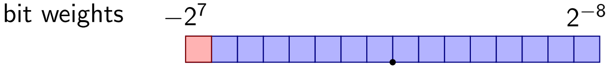
Our hero decided to visit the land of Digital Circuits, a rough and arid country where ony binary fixed-point numbers could live.



$\mathcal{H}$ thought he could feed on them, for a fixed-point number is also a real number.

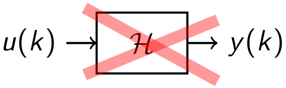
But one day, $\mathcal{H}$ decided to travel far from home

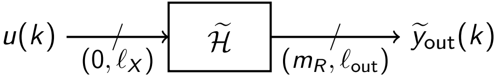
Our hero decided to visit the land of Digital Circuits, a rough and arid country where ony binary fixed-point numbers could live.



$\mathcal{H}$ thought he could feed on them, for a fixed-point number is also a real number.

But $\mathcal{H}$ also had to round his outputs, and this transformed him into a vile monster with a tilde.



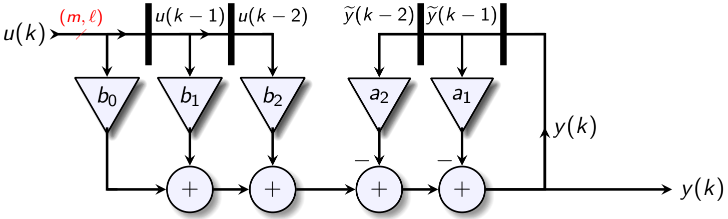


Why fixed-point numbers are toxic for LTI filters

To become a Digital Circuit, an LTI filter had to be cursed with time-domain rounding

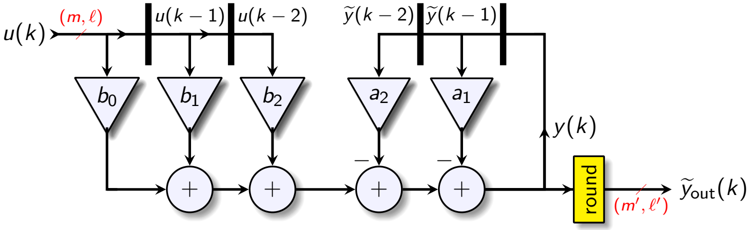
Why fixed-point numbers are toxic for LTI filters

To become a Digital Circuit, an LTI filter had to be cursed with time-domain rounding



Why fixed-point numbers are toxic for LTI filters

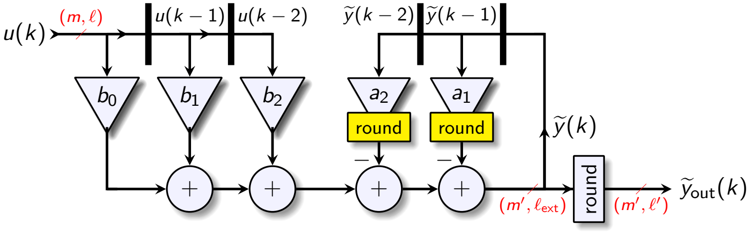
To become a Digital Circuit, an LTI filter had to be cursed with time-domain rounding



- on its output

Why fixed-point numbers are toxic for LTI filters

To become a Digital Circuit, an LTI filter had to be cursed with time-domain rounding

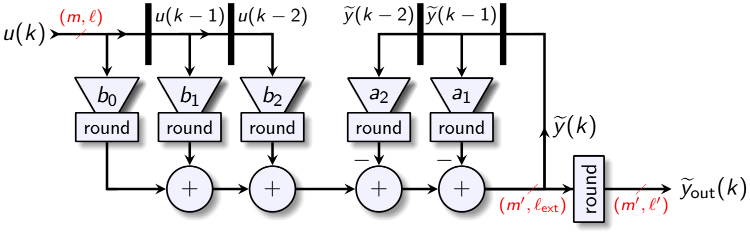


- on its output
- and on its feedback loops if it was recursive

for without rounding, a product has more bits than each of its arguments.

Why fixed-point numbers are toxic for LTI filters

To become a Digital Circuit, an LTI filter had to be cursed with time-domain rounding



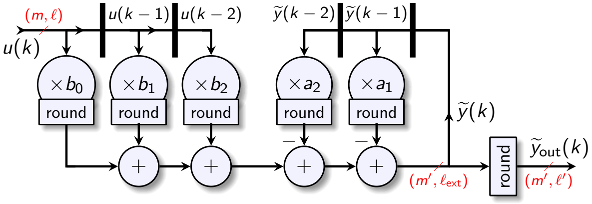
- on its output
- and on its feedback loops if it was recursive

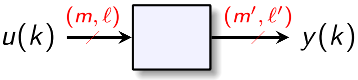
for without rounding, a product has more bits than each of its arguments.

For performance, it was not uncommon to see time-domain rouding warts

all over the innards of a circuit...

In the land of Digital Circuits, $\mathcal{H}$ forgot the fairies' advice!

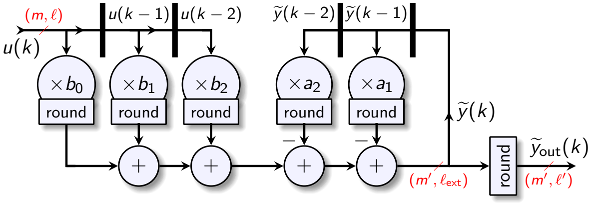


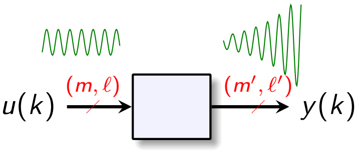


Time-domain rounding is not linear, and for this reason

$\mathcal{H}$ was cut from his transfer function heritage!

In the land of Digital Circuits, $\mathcal{H}$ forgot the fairies' advice!



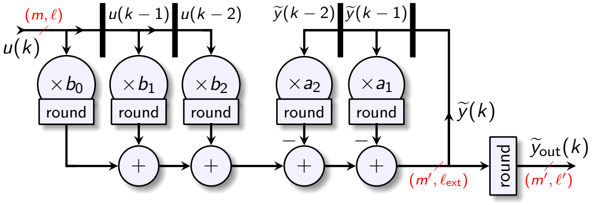


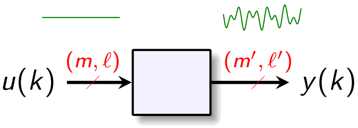
Time-domain rounding is not linear, and for this reason

$\mathcal{H}$ was cut from his transfer function heritage!

Sometimes he would become an unstable digital circuit, in a way difficult to predict.

In the land of Digital Circuits, $\mathcal{H}$ forgot the fairies' advice!





Time-domain rounding is not linear, and for this reason

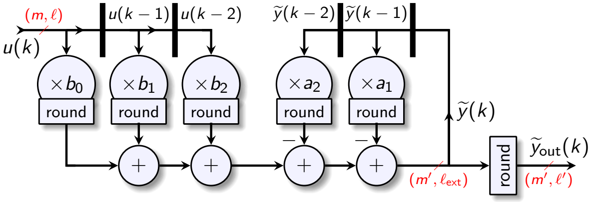
$\mathcal{H}$ was cut from his transfer function heritage!

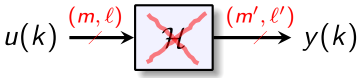
Sometimes he would become an unstable digital circuit, in a way difficult to predict.

Sometimes he would even become a zombie with limit cycle oscillations,

howling in the night even when fed with a null signal.

In the land of Digital Circuits, $\mathcal{H}$ forgot the fairies' advice!





Time-domain rounding is not linear, and for this reason

$\mathcal{H}$ was cut from his transfer function heritage!

Sometimes he would become an unstable digital circuit, in a way difficult to predict.

Sometimes he would even become a zombie with limit cycle oscillations,

howling in the night even when fed with a null signal.

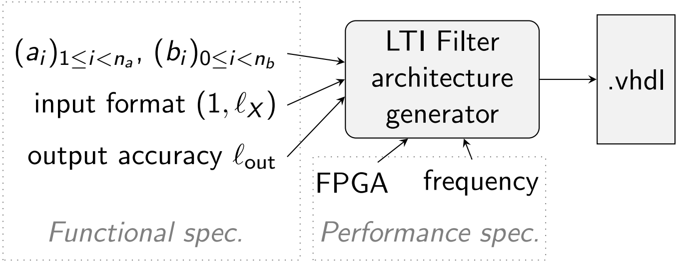
So $\mathcal{H}$ was crying, alone and forgotten

So $\mathcal{H}$ was crying, alone and forgotten

... when the good old witch FloPoCo heard his complaint.
Looking at him, she said:
you're not that evil, you are just poorly specified.

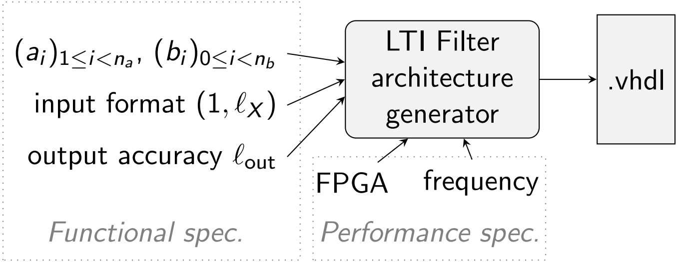
So $\mathcal{H}$ was crying, alone and forgotten

... when the good old witch FloPoCo heard his complaint.
Looking at him, she said:
you're not that evil, you are just poorly specified.
And in a whip of her magical TikZ cursor, she designed him a new interface:



So $\mathcal{H}$ was crying, alone and forgotten

... when the good old witch FloPoCo heard his complaint.
Looking at him, she said:
you're not that evil, you are just poorly specified.
And in a whip of her magical TikZ cursor, she designed him a new interface:

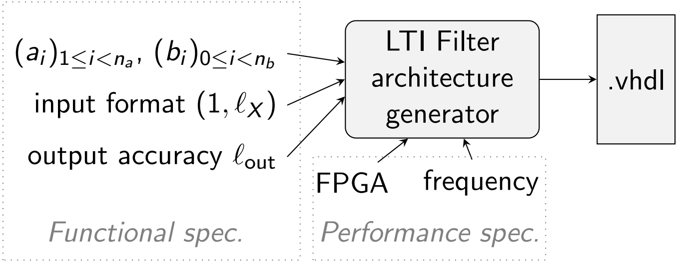


These a_i and b_i were reals! The very real coefficients!

$\mathcal{H}$ suddenly felt much lighter.

So $\mathcal{H}$ was crying, alone and forgotten

... when the good old witch FloPoCo heard his complaint.
Looking at him, she said:
you're not that evil, you are just poorly specified.
And in a whip of her magical TikZ cursor, she designed him a new interface:



These a_i and b_i were reals! The very real coefficients!

$\mathcal{H}$ suddenly felt much lighter.

But... but...

But you forgot to provide me a m_R, cried $\mathcal{H}$

No, said FloPoCo, for I have, somewhere in my library, a spell that can compute it out of your coefficients.

But you forgot to provide me a m_R, cried $\mathcal{H}$

No, said FloPoCo, for I have, somewhere in my library, a spell that can compute it out of your coefficients. (wait a moment, where is it? It was written by poor princess Anastasia Volkova during her captivity in the caves of the mighty sorcerers Lauter and Hilaire...)

But you forgot to provide me a m_R, cried $\mathcal{H}$

No, said FloPoCo, for I have, somewhere in my library, a spell that can compute it out of your coefficients. (wait a moment, where is it? It was written by poor princess Anastasia Volkova during her captivity in the caves of the mighty sorcerers Lauter and Hilaire...)
(I hope it still compiles...)

But you forgot to provide me a m_R, cried $\mathcal{H}$

No, said FloPoCo, for I have, somewhere in my library, a spell that can compute it out of your coefficients. (wait a moment, where is it? It was written by poor princess Anastasia Volkova during her captivity in the caves of the mighty sorcerers Lauter and Hilaire...)
(I hope it still compiles...) Ha, here you go:

Definition: *Worst-Case Peak Gain* $\langle\langle\mathcal{H}\rangle\rangle$ of a filter $\mathcal{H}$

$$\langle\langle\mathcal{H}\rangle\rangle = \max_{||u||_\infty = 1} ||y||_\infty$$

where $||u||_\infty$ is defined as $||u||_\infty = \max_k |u(k)|$.

Then of course,

$$m_R = \lceil \log_2 \langle\langle\mathcal{H}\rangle\rangle \rceil \ .$$

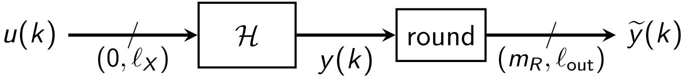
But how will this save me from diverging? cried $\mathcal{H}$

- Remember: you are $\mathcal{H}$, answered the good witch

  and $\mathcal{H}$ doesn't diverge in the pure mathematical world

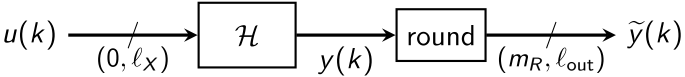
But how will this save me from diverging? cried $\mathcal{H}$

- Remember: you are $\mathcal{H}$, answered the good witch
  and $\mathcal{H}$ doesn't diverge in the pure mathematical world
- Let me cast this spell on your architecture:
  $\widetilde{\mathcal{H}}$ shall return a result that is that of $\mathcal{H}$, rounded only once.
- Then, your alter ego $\widetilde{\mathcal{H}}$ won't diverge.



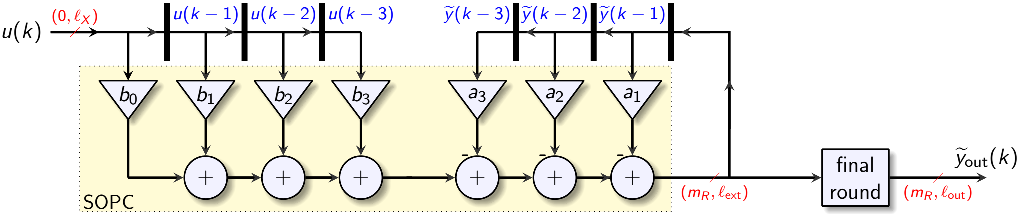
But how will this save me from diverging? cried $\mathcal{H}$

- Remember: you are $\mathcal{H}$, answered the good witch
  and $\mathcal{H}$ doesn't diverge in the pure mathematical world
- Let me cast this spell on your architecture:
  $\widetilde{\mathcal{H}}$ shall return a result that is that of $\mathcal{H}$, rounded only once.
- Then, your alter ego $\widetilde{\mathcal{H}}$ won't diverge.

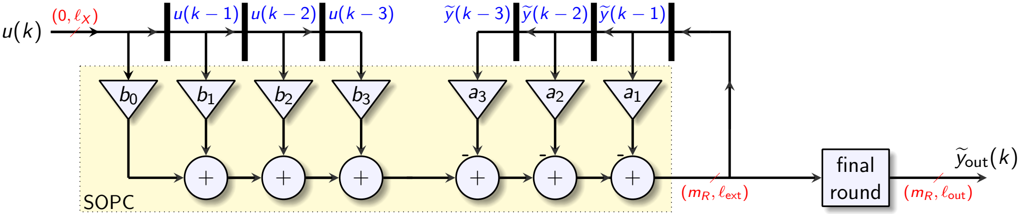


And FloPoCo invoked his two most crafted gremlins, Istoan and de Dinechin, to code this spell, with the help of Princess Anastasia who had managed to escape her tormentors.

Actual architecture



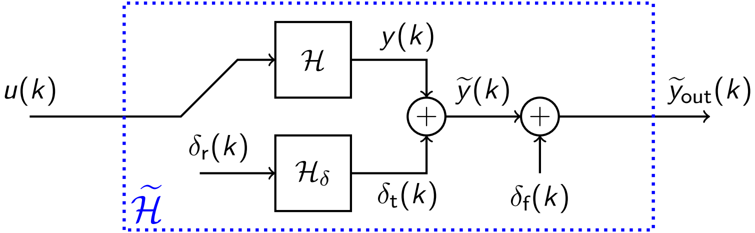
Actual architecture



Another point of view:
When $\ell_{\text{ext}} \to -\infty$ (which means: as the internal accuracy increase),
at some point the computation shall become accurate enough for $\widetilde{\mathcal{H}}$ to converge.

Amplification of errors on the feedback loop

Here should come 3 pages of runes which end in the following figure:



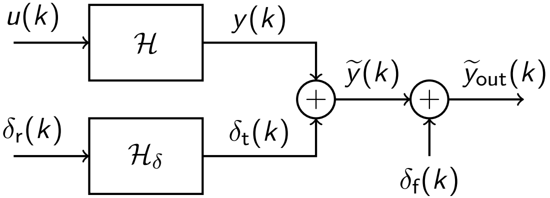
- δ_{r} is the sum of all rounding errors

$$\delta_{\text{r}}(k) = \widetilde{y}(k) - \left(\sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \right)$$

- $\mathcal{H}_\delta$ is the virtual filter that captures the error amplification on the feedback loop:

$$\overline{\delta}_{\text{t}} = \langle\!\langle \mathcal{H}_\delta \rangle\!\rangle \, \overline{\delta}_{\text{r}} \ .$$

Errors are captured, let us chain them in the basement



Rounding errors depend on the architecture

Example: An architecture optimized for LUT-based FPGAs:

- Split an input x into D chunks of α bits (e.g. $\alpha = 4$: hexadecimal).

$$x = \sum_{k=1}^{D} 2^{-k\alpha} d_k \quad \text{where} \quad d_k \in \{0, ..., 2^{\alpha} - 1\}$$

- Then cx becomes

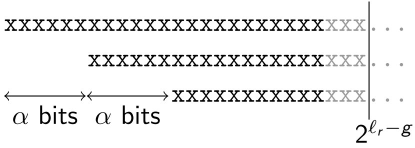
$$cx = \sum_{k=1}^{D} 2^{-k\alpha} c d_k$$

- Tabulate each cd_k sub-product in an α-input table indexed by d_k

$\widetilde{t_1} \approx cd_1$

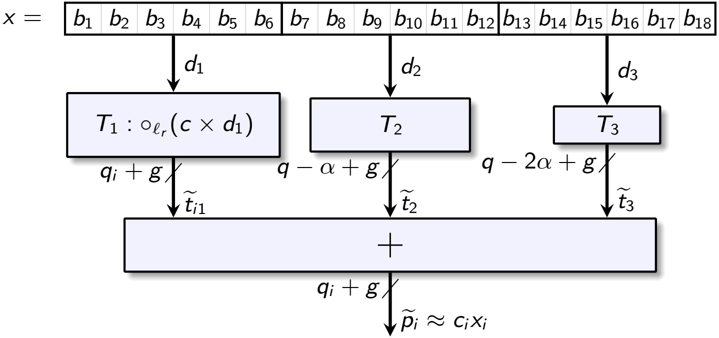
$\widetilde{t_2} \approx cd_2$

$\widetilde{t_3} \approx cd_3$

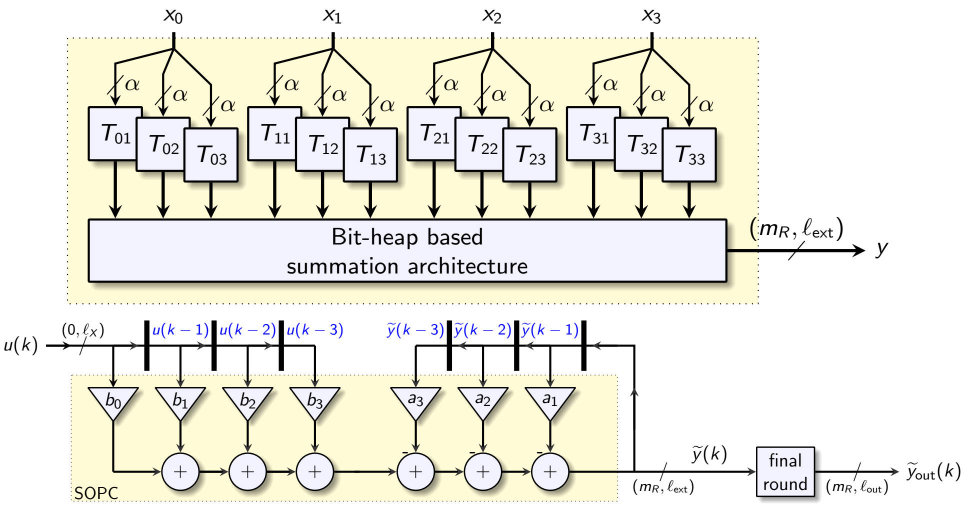


- Remark: c is a real number here, no need to quantize it!

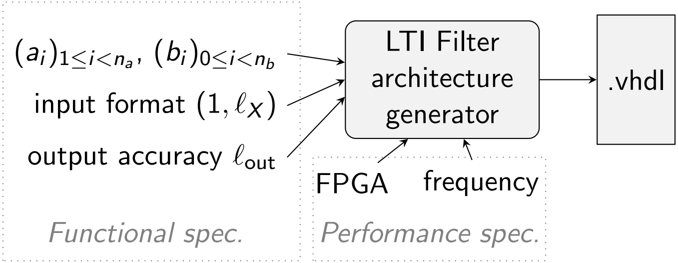
A LUT-based architecture



The error is proportional to 2^{-g}, so can made as small as needed by increasing g.


x_0
x_1
x_2
x_3
α
T_{01}
T_{02}
T_{03}
T_{11}
T_{12}
T_{13}
T_{21}
T_{22}
T_{23}
T_{31}
T_{32}
T_{33}
Bit-heap based
summation architecture
(m_R, ℓ_{ext})
y
$u(k)$
$(0, \ell_x)$
$u(k-1)$
$u(k-2)$
$u(k-3)$
$\tilde{y}(k-3)$
$\tilde{y}(k-2)$
$\tilde{y}(k-1)$
b_0
b_1
b_2
b_3
a_3
a_2
a_1
SOPC
$\tilde{y}(k)$
(m_R, ℓ_{ext})
final
round
(m_R, ℓ_{out})
$\tilde{y}_{out}(k)$

You're not evil, you're just poorly specified



- a_i and b_i: real numbers
  - high-precision numbers from Matlab
  - mathematical formulae such as `sin(3*pi/8)`
- ℓ_X and ℓ_{out}: integers denoting the weight of the least significant bits of the input and of the result.

Computing just right (TM)

ℓ_{out} specifies output precision, but also output accuracy.

A demo?

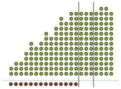
- A small Butterworth filter

```
./flopoco generateFigures=1 FixIIR
coeffb="0x1.7bdf4656ab602p-9:0x1.1ce774c100882p-7:0x1.1ce774c100882p-7:0x1.7bdf4656ab602p-9"
coeffa="-0x1.2fe25628eb285p+1:0x1.edea40cd1955ep+0:-0x1.106c2ec3d0af8p-1"
lsbIn=-12 lsbOut=-12
TestBench n=10000
```

- a radar filter submitted to Thibault a few years ago, with poles really close to 1

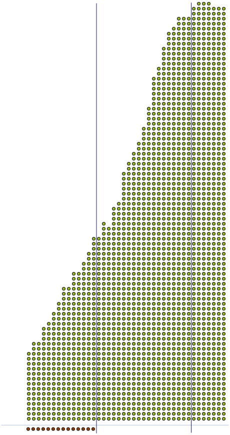
```
./flopoco generateFigures=1 FixIIR
coeffb="0x1.89ff611d6f472p-13:-0x1.2778afe6e1ac0p-11:0x1.89f1af73859fap-12:
0x1.89f1af73859fap-12:-0x1.2778afe6e1ac0p-11:0x1.89ff611d6f472p-13"
coeffa="-0x1.3f4f52485fe49p+2:0x1.3e9f8e35c8ca8p+3:-0x1.3df0b27610157p+3:
0x1.3d42bdb9d2329p+2:-0x1.fa89178710a2bp-1"
lsbIn=-12 lsbOut=-12
TestBench n=10000
```

Bit heaps for some 12-bit Butterworth filters



Order 4, $g = 4$

because $-\log_2 \langle\langle \mathcal{H}_\delta \rangle\rangle = 3$



Order 20, $g = 7$

because $-\log_2 \langle\langle \mathcal{H}_\delta \rangle\rangle = 19$

Sometimes I wonder if this is the right arithmetic for this problem.

Everybody lived happily ever after...

- A point of view on filter design that is universal
  - don't compute useless bits: output format specifies output accuracy
  - complete error analysis (coefficient quantization + architectural rounding errors)
  - error amplification captured by a safe implementation of the WCPG

Everybody lived happily ever after...

- A point of view on filter design that is universal
  - don't compute useless bits: output format specifies output accuracy
  - complete error analysis (coefficient quantization + architectural rounding errors)
  - error amplification captured by a safe implementation of the WCPG

- Also a magical cure for a few other filter diseases
  - If you input a 0 signal, the output converges to 0 (± 1 unit in the last place)

Everybody lived happily ever after...

- A point of view on filter design that is universal
  - don't compute useless bits: output format specifies output accuracy
  - complete error analysis (coefficient quantization + architectural rounding errors)
  - error amplification captured by a safe implementation of the WCPG
- Also a magical cure for a few other filter diseases
  - If you input a 0 signal, the output converges to 0 ($\pm$ 1 unit in the last place)
- A finely tuned implementation that uses FPGA-specific arithmetic

... And they had a lot of children

This is just a basic block on the way to more interesting filter structures.

- implementation space: state space, SIF
- clean rule of the game: enables comparison of functionally equivalent architectures

(to be continued)

- Try me in FloPoCo v. 4.1.3 or later
- Read more on HAL or in IEEETC
  Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study

Later, they visited the Frequency Domain Fairies

... with two presents to help them design circuits that obey a frequency specification:

Definitive Curse 1

A digital circuit $\mathcal{C}$ is said to be faithful to a stable LTI filter $\mathcal{H}$
iff the numerical difference between the fixed-point output $\widetilde{y}_{\text{out}}(k)$ of $\mathcal{C}$
and the exact result $y(k)$ of $\mathcal{H}$
does not exceed one unit in the last place of $\widetilde{y}_{\text{out}}(k)$.

Definitive Curse 2

A Digital Circuit $\mathcal{C}$ is said to be faithful to a frequency specification
iff there exists a stable LTI filter $\mathcal{H}$ such that
1/ $\mathcal{H}$ respects the frequency specification, and
2/ $\mathcal{C}$ is faithful to $\mathcal{H}$.

Example: Multimodal sound synthesis (WIP)

Big picture

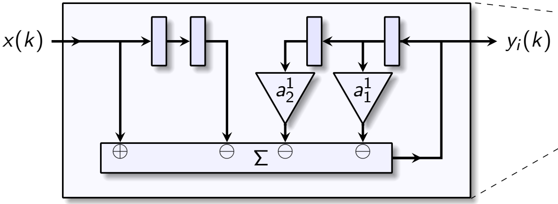
The part that I don't understand:

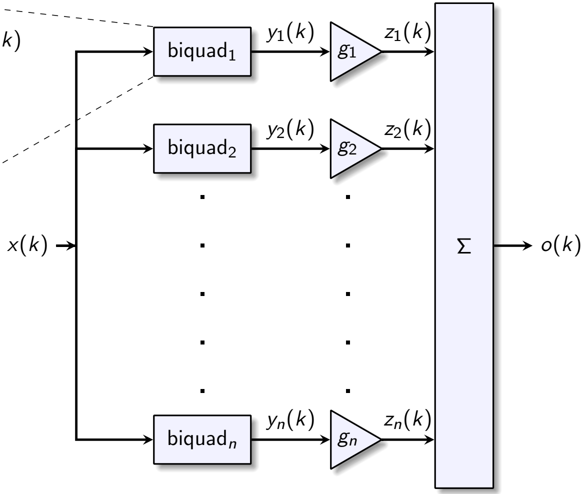
- finite element decomposition of a noisy object (e.g. a bell)
- physics simulation to get its resonnant frequencies with their attenuations

The part that I more or less understand:

- build a biquad filter for each frequency
- sum them all together to simulate the sound of the noisy object

The big picture in picture





$$
\begin{aligned}
b_0 &= 1 \\
b_1 &= 0 \\
b_2 &= -1 \\
\omega &= 2\pi f/s \\
r &= 0.001^{1/(s\times t_{60})} \\
a_1 &= -2r\cos(\omega) \\
a_2 &= r^2
\end{aligned}
$$

https://ccrma.stanford.edu/~jos/filters/Decay_Time_Q_Periods.html

Resonating filters with slow decay time have high WCPGs

From the bell model in the FAUST distribution:

i	a_1	a_2	$\langle\langle\mathcal{H}\rangle\rangle_i$	$\langle\langle\mathcal{H}_\delta\rangle\rangle_i$
0	-1.99510896	0.999985754	1.79e5	1.29e6
1	-1.99504113	0.999985695	1.79e5	1.27e6
2	-1.98264325	0.999980509	1.31e5	4.98e5
...	...	...	...	...
25	-1.85236752	0.999858916	1.81e4	2.40e4
...	...	...	...	...
47	-1.42887342	0.7367661	9.78	8.59
48	-0.351596594	0.0449641831	2.71	1.45

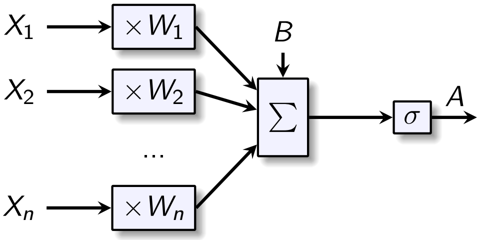
- For a bell actioned with a hammer, do we need to consider WCPG?
- ... and for a violin string?
- Audible zombies when using low precisions.

And the work in progress is

- To build a FloPoCo operator that builds the hardware for all this.

Example: Low-precision logarithmic neuron

Perceptron artificial neuron model



X: input vector, **W**: weight vector, B: bias, σ: activation function
Output A of the neuron defined by:

$$A = \sigma(\mathbf{W} \cdot \mathbf{X} + B)$$

What precision should a neuron use?

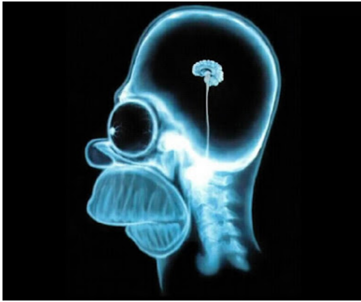
Current consensus

- 8-bit integers are good enough for weights and activations
  - (if higher precision is used for internal computations)
- 1-bit representations (binary and ternary networks)
  - require more layer and specific training
  - entail loss of application-level accuracy

Is there some space in between?

- data on 3 to 6 bits ?
- (incidentally, this would be a very good match to LUT-based FPGAs)

Proposed approach: use ad-hoc **logarithmic formats**



K. Usher, *"The Dwarf in the Dirt"*,

Bones, 2009

Instead of encoding a real value X, encode its logarithm.

Unfortunately $\log(X)$ is only defined for $X > 0$. To represent $X \in \mathbb{R}$, we will need

- a sign bit s_x for the sign of X,
- $L_X \approx \log(|X|)$ encoded in some signed fixed point format
  - itself signed: $L_X \geq 0 \iff X \geq 1$
- a "is-zero" bit z_x (or a special encoding of $X = 0$ in one of the values of L_X)

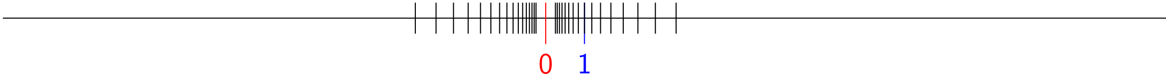
$^{\lg}X = (s_X, z_X, L_X) =$

s_X	z_X	x_2	x_1	x_0	x_{-1}	x_{-2}	
		2^2	2^1	2^0	2^{-1}	2^{-2}	binary weight

"A kind of floating-point where you only have the exponent, and it is fractional"

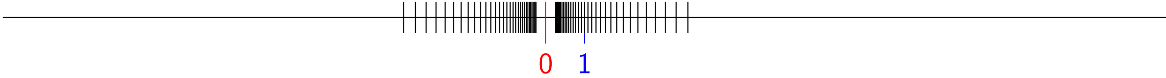
The MSB and LSB of a LNS representation

$(m, \ell) = (2, -2) \qquad L_X = \boxed{x_2}\boxed{x_1}\boxed{x_0}\bullet\boxed{x_{-1}}\boxed{x_{-2}}$



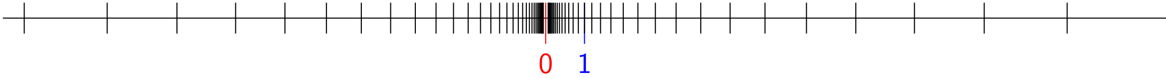
Adding one bit to the LSB doubles the number of representable values, with the same range.

$(m, \ell) = (2, -3) \qquad L_X = \boxed{x_2}\boxed{x_1}\boxed{x_0}\bullet\boxed{x_{-1}}\boxed{x_{-2}}\boxed{x_{-3}}$



Adding one bit to the MSB increases the range, and also reduces the gap around zero.

$(m, \ell) = (3, -2) \qquad L_X = \boxed{x_3}\boxed{x_2}\boxed{x_1}\boxed{x_0}\bullet\boxed{x_{-1}}\boxed{x_{-2}}$



LNS arithmetic

- Multiplication turns into addition

$$\log(X \times W) = \log(X) + \log(W)$$

  - And it is exact! (fixed-point addition may overflow, but no rounding)

LNS arithmetic

- Multiplication turns into addition

$$\log(X \times W) = \log(X) + \log(W)$$

  - And it is exact! (fixed-point addition may overflow, but no rounding)
- Division and square root similarly cheap (no use here)

LNS arithmetic

- Multiplication turns into addition

$$\log(X \times W) = \log(X) + \log(W)$$

  - And it is exact! (fixed-point addition may overflow, but no rounding)
- Division and square root similarly cheap (no use here)
- Addition turns into a nightmare

$$\log(X+Y) \quad = \quad \log\left(X \times \left(1 + \frac{Y}{X}\right)\right) \quad = \quad \log(X) + \log\left(1 + b^{\log(Y)-\log(X)}\right)$$

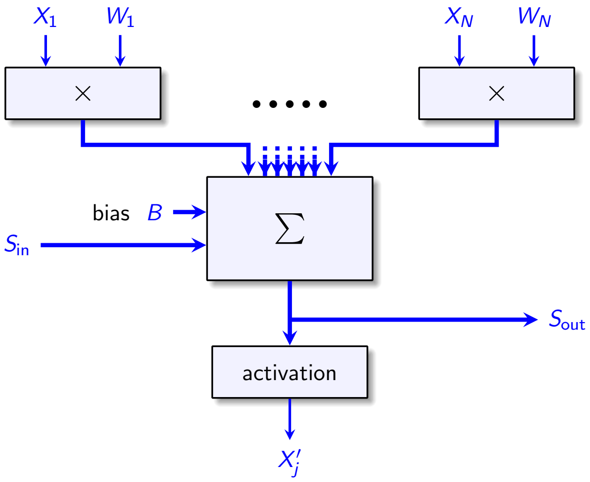
  - one subtraction to compute $Z = \log(Y) - \log(X)$
  - evaluation of the ugly function $\log\left(1 + b^{Z}\right)$
  - another addition

Reference ad-hoc linear domain implementation

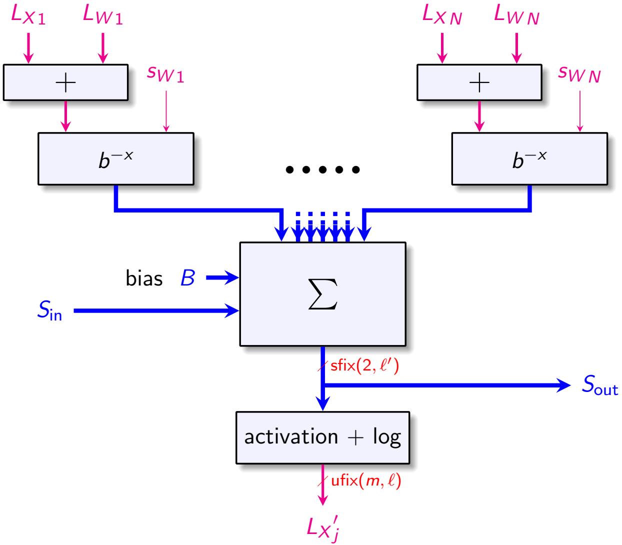
We unroll a neuron:

$$B + \sum_i (W_i \times X_i)$$

(it unlocks some optimizations in the $\boxed{\sum}$ box)



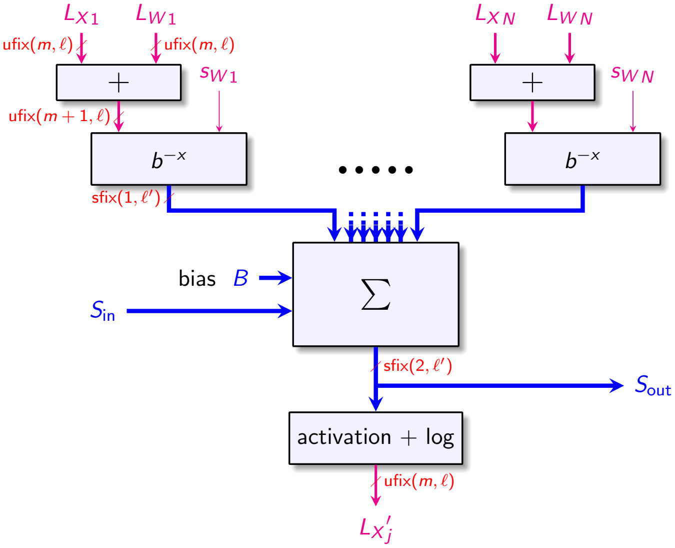
Proposed design



logarithmic data

linear data

Proposed design



logarithmic data

linear data

Key ideas

- Leverage LNS to replace $\times$ by $+$
- Dodge the complexity of accumulating in LNS
- Parametric design to experiment with application-level accuracy
- Merge linear to log conversion with activation function table
- Leverage FPGA LUT architecture to tabulate ugly functions

Cost of tabulating b^{-x} (or any function) in an FPGA

The FPGA basic logic element:
an α-input Look-Up Table

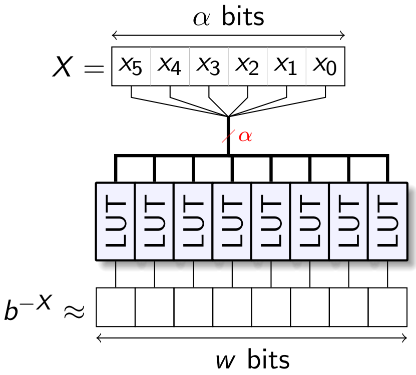
- universal logic gate
  (any truth table of α bits)
- $\alpha \in \{4, 5, 6\}$ these days,
  depending on vendor and generation

Cost of tabulating b^{-x} (or any function) in an FPGA

The FPGA basic logic element:
an α-input Look-Up Table

- universal logic gate
  (any truth table of α bits)
- $\alpha \in \{4, 5, 6\}$ these days,
  depending on vendor and generation

Therefore, a table
of α in bits and w out bits
costs w FPGA LUTs:



Input size α is FPGA soft spot! For input sizes larger than α, cost grows exponentially.

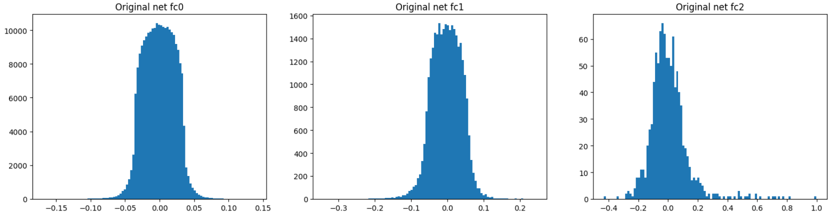
Other advantages of plain tabulation

- As accurate as your output format allows
  - no approximation error
  - one single rounding error
- Output size can be larger than input size
  - cost grows only linearly with output size
  - this is what enables accurate summation
- It works for any function

  - No reason why 2 should be the best b
  - Activation: Gaussian ReLU or sigmoid for the same cost
- Oh, and it is simple to program and use.

Only condition: keep our data format really, really small!

Weight distribution observations

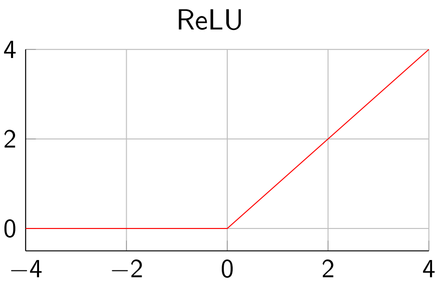


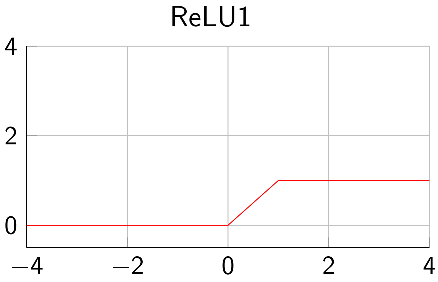
- Looks like a normal distribution
- $|W| \leq 1$
- $|X| \leq 1$?

Every bit matters, in particular sign bits

- If $|X| \leq 1 \implies \log(|X|) \leq 0$
  -
  - We can decide that $L_X = \lfloor -\log(|X|) \rceil$ instead of $L_X = \lfloor \log(|X|) \rceil$
  - encoding L_X as an unsigned fixed-point number effectively saves 1 bit !
- How to ensure $X \leq 1$ and $W \leq 1$?
  - For the weights: it is OK without retraining (saturate the few large values to 1)
  - For the activations: just use ReLU1 (or any function that maxes at 1)

Every bit matters, and activation functions may help





Now $X \leq 1$!

Also $X \geq 0$, we can drop s_X as well

Now ${}^{\lg}X = (z_X, L_X)$ and ${}^{\lg}W = (s_W, z_W, L_W)$

Every bit matters, in particular zero bits

Now ${}^{\lg}X = (z_X, L_X)$ and ${}^{\lg}W = (s_W, z_W, L_W)$
What happens if we drop those bits z_X and z_W ?

- 0 is no longer representable

It should be very very bad, as zero is the most common value
both for weights and activations.

Yet another trick

Let us call Z the largest possible value of L_X
(which corresponds to the smallest representable value of $X = 2^{-L_X}$).
If Z is rounded to 0 by the $\boxed{b^{-X}}$ block, then the same holds for $Z + L_W$, $\forall L_W$
Then Z effectively represents a zero activation.
(the same holds for weights)

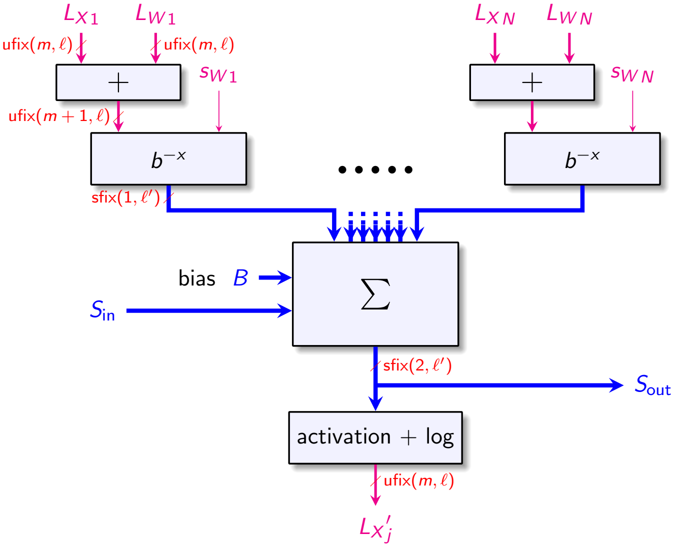
So we do not care that we cannot represent zero, and we can drop both zero bits.

Summary of “every bit matters”: ${}^{\mathrm{lg}}X$ and ${}^{\mathrm{lg}}W$

Logarithmic representation for inputs and product (here for MSB $m = 2$ and LSB $\ell = -1$)

binary weight	2^3	2^2	2^1	2^0	2^{-1}	
${}^{\mathrm{lg}}W = -\lfloor \log_b \lvert W \rvert \rceil =$		w_2	w_1	w_0	w_{-1}	s_W
${}^{\mathrm{lg}}X = -\lfloor \log_b X \rceil =$		x_2	x_1	x_0	x_{-1}	
${}^{\mathrm{lg}}P = -\lfloor \log_b \lvert P \rvert \rceil =$	p_3	p_2	p_1	p_0	p_{-1}	s_W

Back to the design



Simulation setup

- pytorch to evaluate the classification accuracy
- FloPoCo to describe the architecture
- Vivado to synthesize our design and evaluate the area

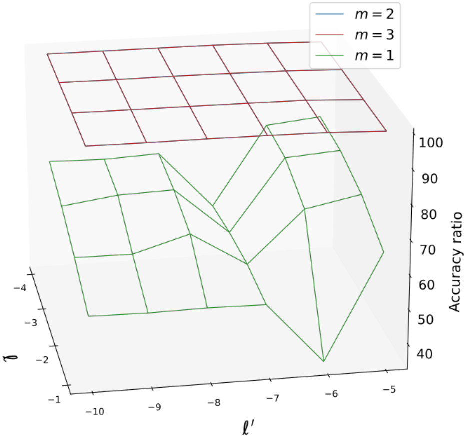
Exhaustive exploration of the design space for MNIST,

then targetted experiments on a larger CIFAR10.

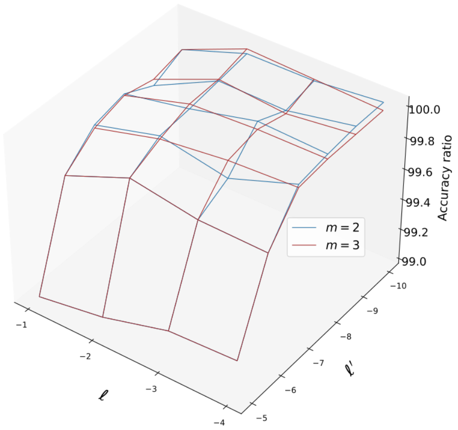
Accuracy experiments on MNIST

- Training standard (784, 300, 100, 10) MLP in full precision with `pytorch`: 98.03% accuracy on test set
- Conversion to LNS and evaluate accuracy on the test set again

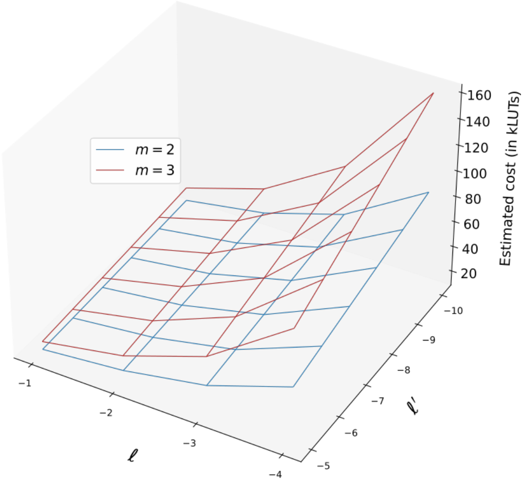
Accuracy experiments on MNIST



Accuracy experiments on MNIST



Cost of the MNIST architecture



As expected, exponential in ℓ, linear in ℓ'

CIFAR10

Take a pre-trained network, and convert it to LNS
VGG-like network layer architecture:

layer index	layer type
(1)	$LNSConv(3, 128) + ReLU1()$
(2)	$LNSConv(128, 128) + ReLU1()$
(3)	$MaxPool2d(2, 2)$
(4)	$LNSConv(128, 256) + ReLU1()$
(5)	$LNSConv(256, 256) + ReLU1()$
(6)	$MaxPool2d(2, 2)$
(7)	$LNSConv(256, 512) + ReLU1()$
(8)	$LNSConv(512, 512) + ReLU1()$
(9)	$MaxPool2d(2, 2)$
(10)	$LNSConv(512, 1024) + ReLU1()$
(11)	$MaxPool2d(2, 2)$

Results are similar for CIFAR 10

Accuracy and synthesis results for parallel neurons

benchmark	parameters $(m, \ell), (1, \ell')$	accuracy ratio	LUT cost	latency
MNIST	(2, -1), (1, -6)	99.6	12491	10.3ns
MNIST	(2, -1), (1, -7)	99.8	13790	10.9ns
MNIST	6-bit linear	99.9	36658	10.2ns
CIFAR10	6-bit linear	96.9	51910	13.0ns
CIFAR10	(3, -1), (1, -10)	97.5	30632	12.8ns
CIFAR10	**(2, -2), (1, -10)***	98.5	**28652**	**12.4ns**
CIFAR10	8-bit linear	**99.8**	83522	13.4ns

* L_X on 5 bits, L_W on 6 bits, L_P on 7 bits, summation of 12-bit terms.

Conclusion

Very small logarithmic encoding works for the weights and activations :

- more accurate than standard linear quantization with identical bit-width
- smaller on FPGA than standard linear implementation of similar accuracy

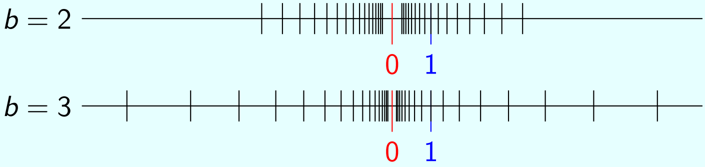
All this was without any form of retraining.

Retraining can only improve accuracy and/or save a few more bits.

All this was in base 2

There is absolutely no reason to think that it is the best base.

Another base, another range/accuracy trade-off for the same format.



Example: floating-point exponential

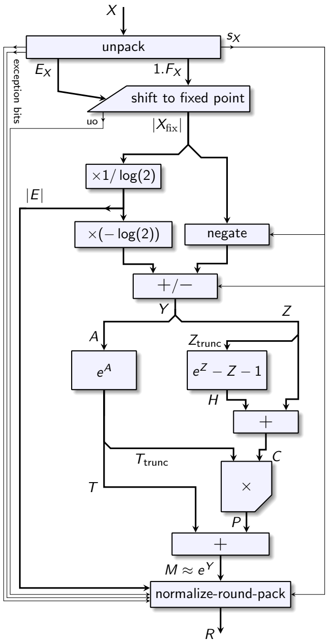
First, a math proficiency test

Three identities to remember from our happy school days

$$2^X = e^{X \log(2)} \tag{1}$$

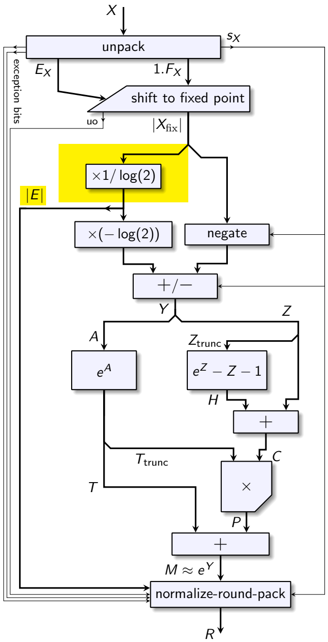
$$e^{A+B} = e^A \times e^B \tag{2}$$

$$e^Z \approx 1 + Z + \frac{Z^2}{2} \quad \text{if } Z \text{ is small} \tag{3}$$



We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

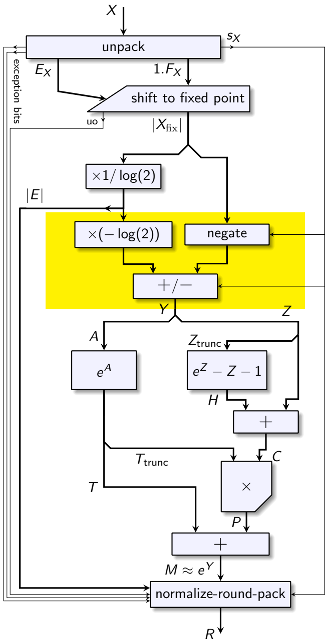


We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

Compute

$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$



We want to obtain e^X as

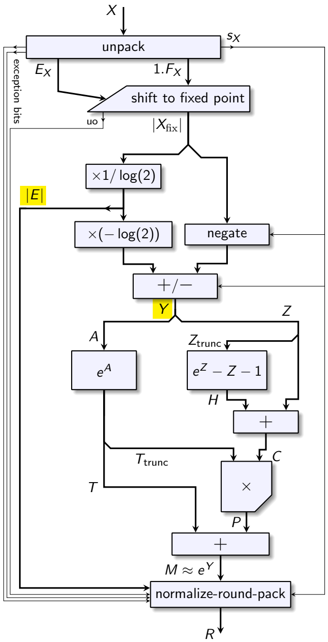
$$e^X = 2^E \cdot 1.F$$

Compute

$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$

then

$$Y \approx X - E \times \log 2.$$



We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

Compute

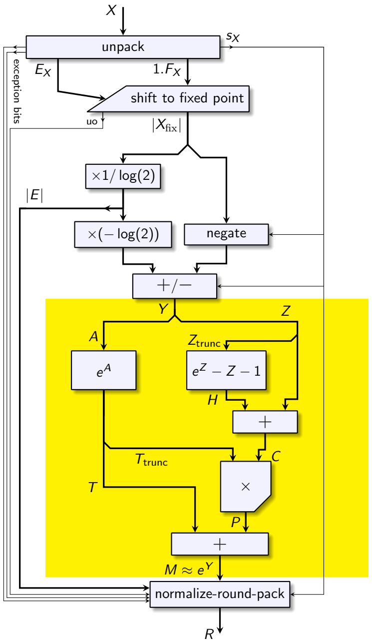
$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$

then

$$Y \approx X - E \times \log 2.$$

Now

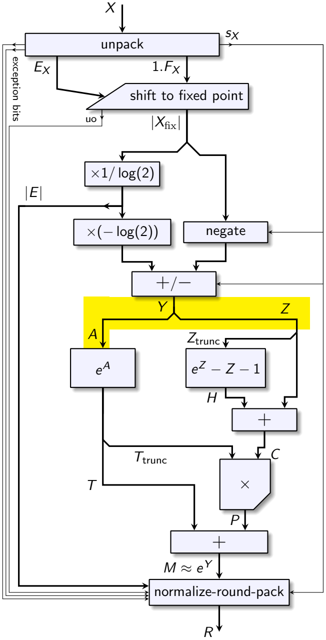
$$\begin{aligned} e^X &= e^{E \log 2 + Y} \\ &= e^{E \log 2} \cdot e^Y \\ &= 2^E \cdot e^Y \end{aligned}$$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

Now we have to compute e^Y
with $Y \in (-1/2, 1/2)$.



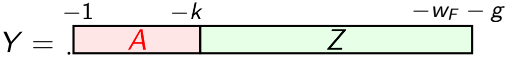
We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

Now we have to compute e^Y

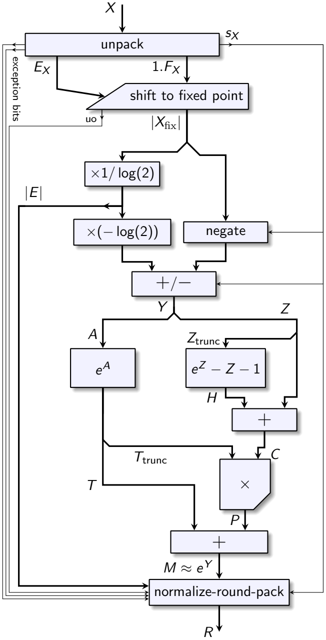
with $Y \in (-1/2, 1/2)$.

Split Y:



i.e. write

$$Y = A + Z \qquad \text{with} \quad Z < 2^{-k}$$

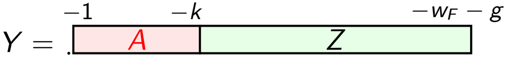


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

Now we have to compute e^Y with $Y \in (-1/2, 1/2)$.

Split Y:

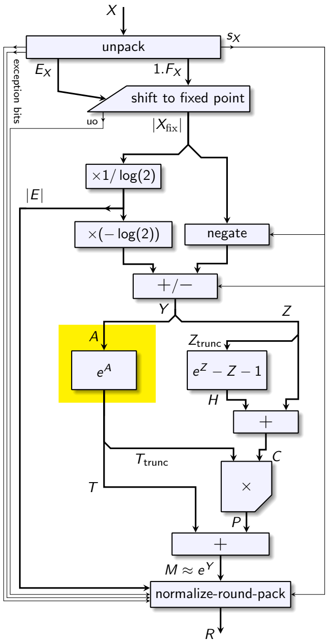


i.e. write

$$Y = A + Z \qquad \text{with} \quad Z < 2^{-k}$$

so

$$e^Y = e^A \times e^Z$$

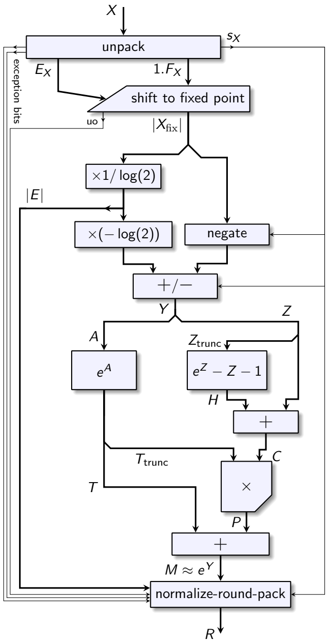


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Tabulate e^A in a ROM



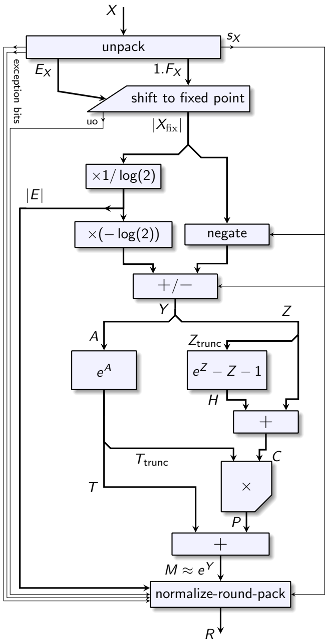
We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$



We want to obtain e^X as

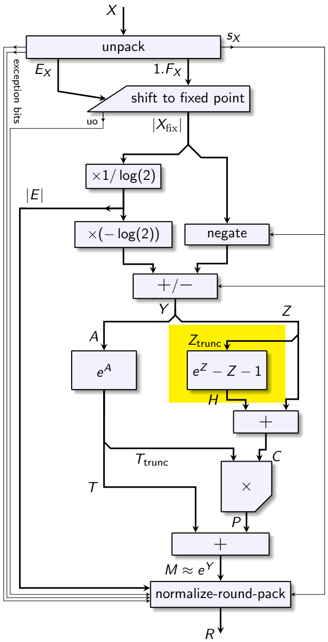
$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

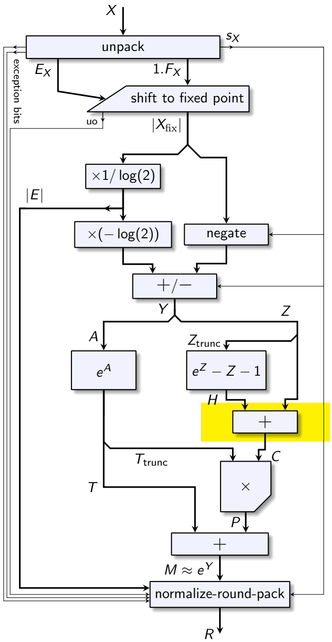
$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$

Evaluate $e^Z - Z - 1$ somewhow
(out of Z truncated to its higher bits only)



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

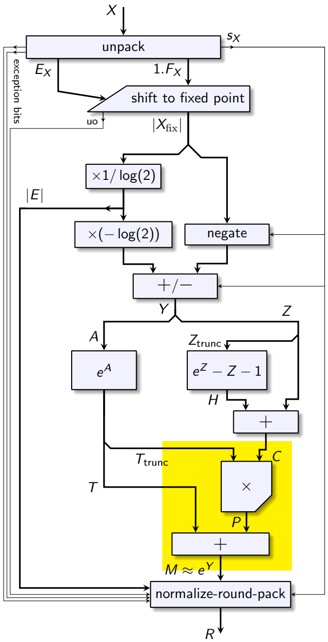
$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$

Evaluate $e^Z - Z - 1$ somewhow

(out of Z truncated to its higher bits only)

then add Z to obtain $e^Z - 1$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

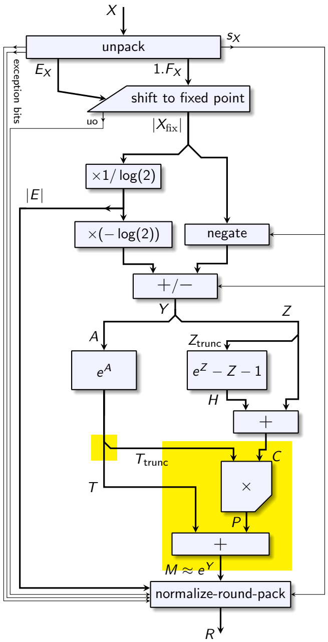
$$e^Y = e^A \times e^Z$$

Also notice that

$$e^Z = 1.\overbrace{000...000}^{k-1 \text{ zeroes}}zzzz$$

Evaluate $e^A \times e^Z$ as

$$e^A + e^A \times (e^Z - 1)$$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

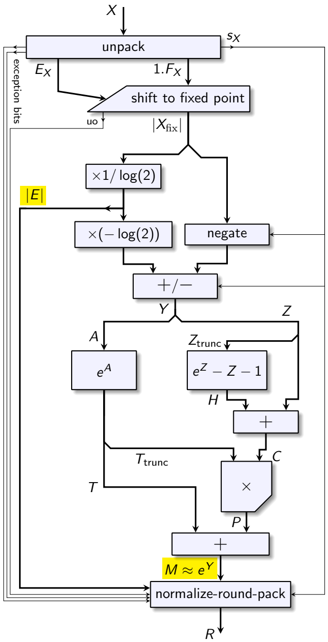
Also notice that

$$e^Z = 1.\overbrace{000...000}^{k-1 \text{ zeroes}}zzzz$$

Evaluate $e^A \times e^Z$ as

$$e^A + e^A \times (e^Z - 1)$$

(before the product, truncate e^A to precision of $e^Z - 1$)

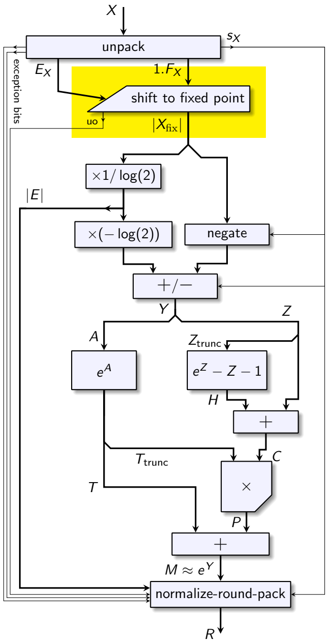


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

And that's it, we have E and e^Y

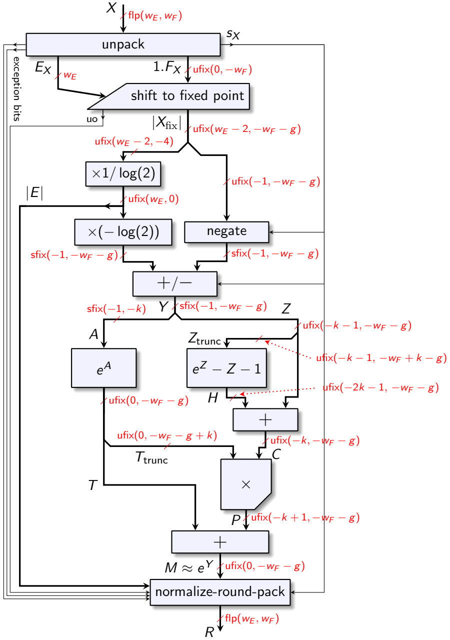


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

And that's it, we have E and e^Y
(using only *fixed-point* computations)



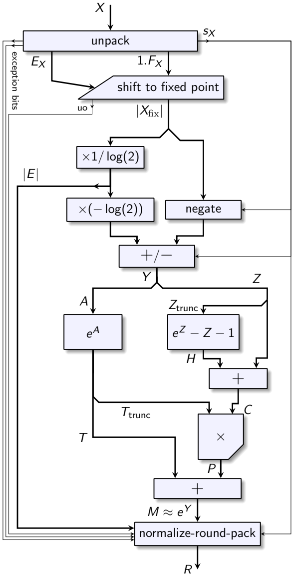
We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

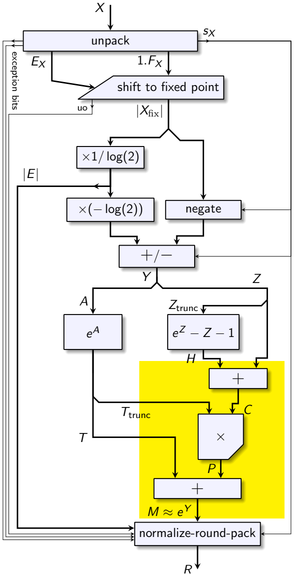
And that's it, we have E and e^Y (using only *fixed-point* computations)

Single-precision magic



Modern FPGAs also have

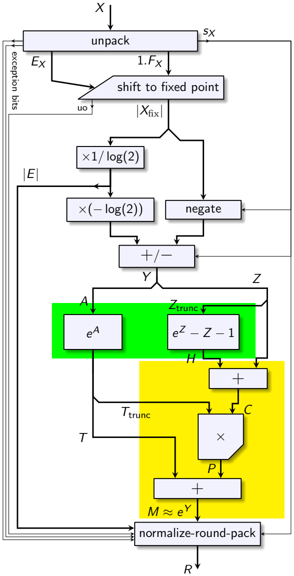
Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders

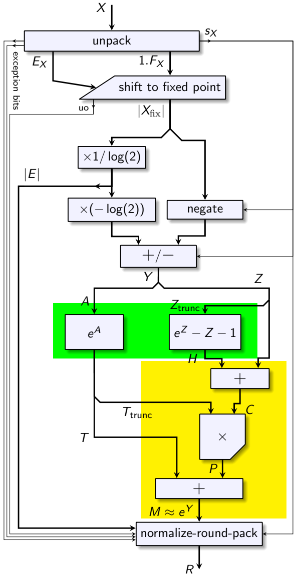
Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

Single-precision magic



Modern FPGAs also have

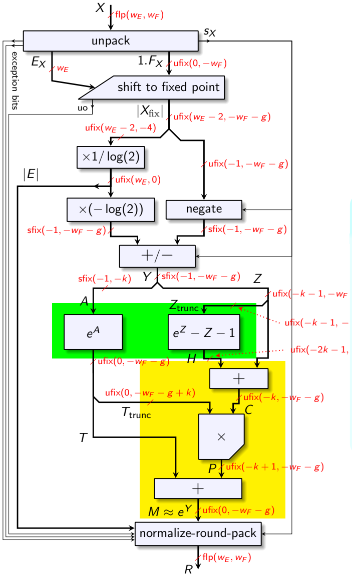
- small multipliers with pre-adders and post-adders
- … and dual-ported small memories

Single-precision accurate exponential on Xilinx

- one block RAM (0.1% of the chip)
- one DSP block (0.1%)
- $<$ 400 LUTs (0.1%, $\approx$ one FP adder)

to compute one exponential per cycle at 500MHz
($\sim$ one AVX512 core trashing on its 16 FP32 lanes)

Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders
- … and dual-ported small memories

Single-precision accurate exponential on Xilinx

- one block RAM (0.1% of the chip)
- one DSP block (0.1%)
- $<$ 400 LUTs (0.1%, $\approx$ one FP adder)

to compute one exponential per cycle at 500MHz
($\sim$ one AVX512 core trashing on its 16 FP32 lanes)

For one specific value only of the architectural parameter k!
(over-parameterization is cool)

Error analysis for dummies (and other proof assistants)

Computing just right

"Error analysis" used to be the kind of things you do to ensure the operator works.

Computing just right

"Error analysis" used to be the kind of things you do to ensure the operator works.
This is sooooo nineties.
Here, error analysis is for **optimization**.
(the fact that the operators work is an appreciable bonus)

Error analysis method in my early papers: handwaving

The typical error analysis used to look like this:

- "This term contributes at most 1 ulp (unit in the last place) to the overall error"
- "This operation contributes at most one half-ulp to the error"
- ...
- "Altogether we have 6 ulps of error"
- "so if we add $\lceil \log_2(6) \rceil$ bits to all the datapath, it should be accurate enough.

And then I saw the light

G. Melquiond, the creator of Gappa (the proof assistant for the rest of us)

An error is a difference between a less accurate value and a more accurate value.

And then I saw the light

G. Melquiond, the creator of Gappa (the proof assistant for the rest of us)

An error is a difference between a less accurate value and a more accurate value.

For instance, to bound some error, first write it $\delta_{AC} = A - C$, then

And then I saw the light

G. Melquiond, the creator of Gappa (the proof assistant for the rest of us)

An error is a difference between a less accurate value and a more accurate value.

For instance, to bound some error, first write it $\delta_{AC} = A - C$, then

- look for some intermediate value B (more accurate than A but less accurate than C)

And then I saw the light

G. Melquiond, the creator of Gappa (the proof assistant for the rest of us)

An error is a difference between a less accurate value and a more accurate value.

For instance, to bound some error, first write it $\delta_{AC} = A - C$, then

- look for some intermediate value B (more accurate than A but less accurate than C)
- write

$$\begin{array}{lrcccl} & A - C & = & A - B & + & B - C \\ \text{or} & \delta_{AC} & = & \delta_{AB} & + & \delta_{BC} \end{array} \qquad (4)$$

And then I saw the light

G. Melquiond, the creator of Gappa (the proof assistant for the rest of us)

An error is a difference between a less accurate value and a more accurate value.

For instance, to bound some error, first write it $\delta_{AC} = A - C$, then

- look for some intermediate value B (more accurate than A but less accurate than C)
- write

$$A - C = A - B + B - C \tag{4}$$

or

$$\delta_{AC} = \delta_{AB} + \delta_{BC}$$

- By triangular inequality,

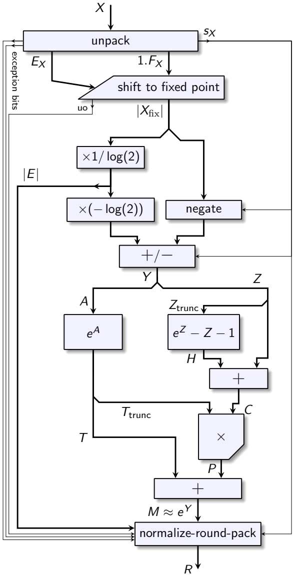
$$|\delta_{AC}| \leq |\delta_{AB}| + |\delta_{BC}|$$

- Therefore, the error bounds (noted $\overline{\delta} = \max|\delta|$) verify

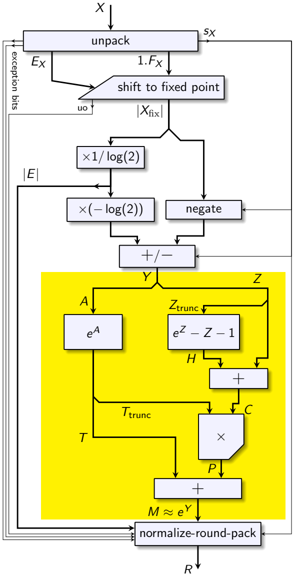
$$\overline{\delta}_{AC} = \overline{\delta}_{AB} + \overline{\delta}_{BC} \tag{5}$$

A divide-and-conquer method, to use when approximations and rounding errors pile up...

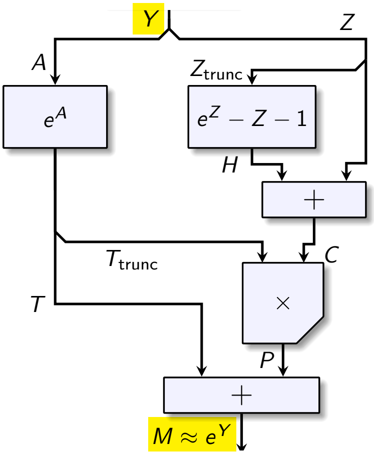
A big mess of rounding errors piled over approximation



A big mess of rounding errors piled over approximation

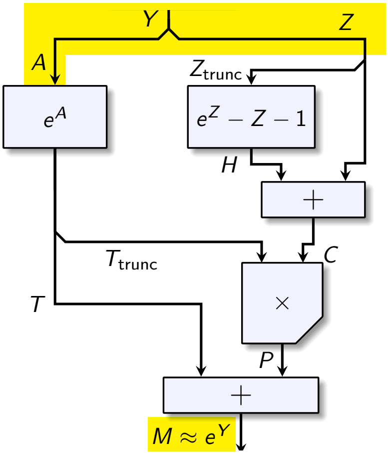


A big mess of rounding errors piled over approximation



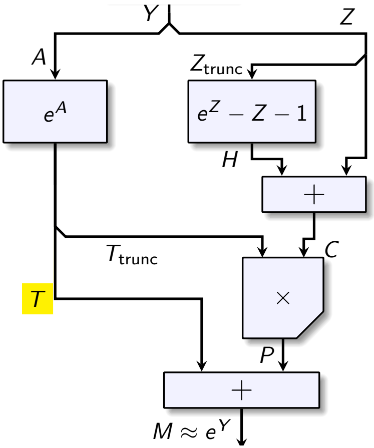
$$\delta_{\text{total}} = M - e^Y$$

A big mess of rounding errors piled over approximation



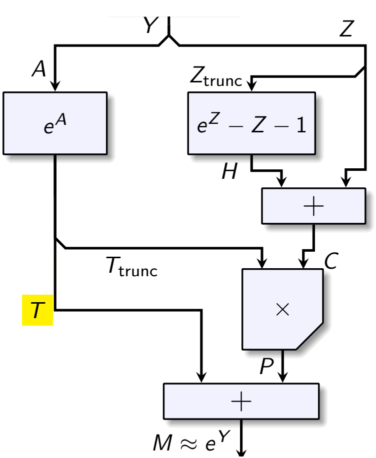
$$\begin{aligned}\delta_{\text{total}} &= M - e^Y \\ &= M - e^A e^Z \qquad \text{since } Y = A + Z \text{ exactly}\end{aligned}$$

A big mess of rounding errors piled over approximation



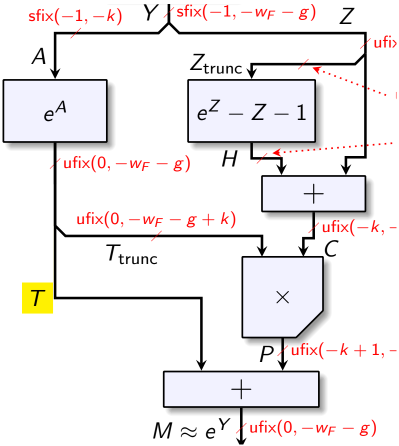
$$
\begin{aligned}
\delta_{\text{total}} &= M - e^Y \\
&= M - e^A e^Z \qquad \text{since } Y = A + Z \text{ exactly} \\
&= M - Te^Z \quad + \quad Te^Z - e^A e^Z
\end{aligned}
$$

A big mess of rounding errors piled over approximation



$$\delta_{total} = M - e^Y$$
$$= M - e^A e^Z \quad \text{since } Y = A + Z \text{ exactly}$$
$$= M - Te^Z + \underbrace{Te^Z - e^A e^Z}_{= (T - e^A)e^Z = \delta_T}$$

A big mess of rounding errors piled over approximation



$$\begin{aligned}
\delta_{\text{total}} &= M - e^Y \\
&= M - e^A e^Z \qquad \text{since } Y = A + Z \text{ exactly} \\
&= M - Te^Z \quad + \quad \underbrace{Te^Z - e^A e^Z}
\end{aligned}$$

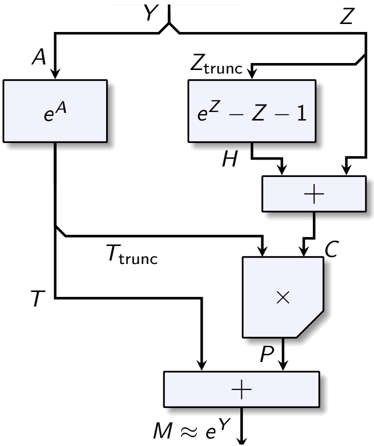
$$\underbrace{Te^Z - e^A e^Z} = (T - e^A)e^Z = \delta_T$$

Now we can bound this first source of error:

$$\begin{aligned}
|\delta_T| &= |T - e^A| \cdot |e^Z| \\
&< 2^{-w_F - g} \cdot (1 + 2^{-k+1}) \qquad (6)
\end{aligned}$$

Keep it parametric!

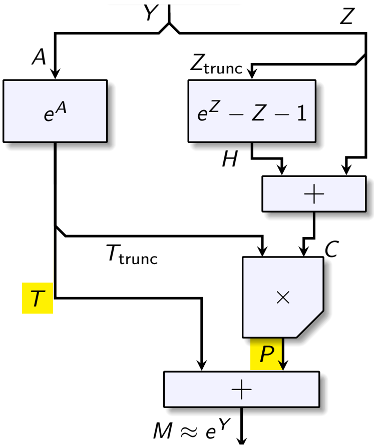
That was the first step



$$\delta_{\text{total}} = M - Te^Z + \delta_T$$

Where can we go from here?

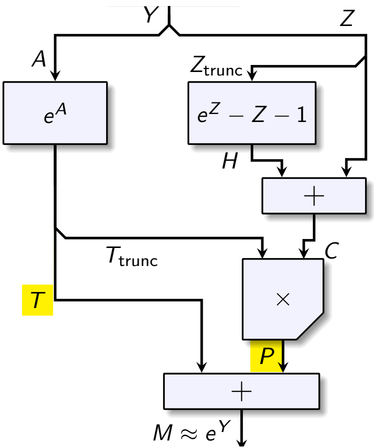
That was the first step



$$\delta_{total} = M - Te^Z + \delta_T$$

Where can we go from here?
Last addition is exact (that's fixed-point for you) so $M = T + P$, hence :

$$\begin{aligned} M - Te^Z &= T + P - Te^Z \\ &= T + P - T_{trunc}C + T_{trunc}C - Te^Z \end{aligned}$$

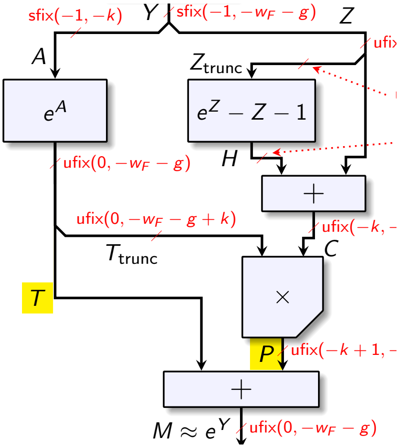


$$\delta_{total} = M - Te^Z + \delta_T$$

Where can we go from here?
Last addition is exact (that's fixed-point for you) so $M = T + P$, hence :

$$\begin{aligned} M - Te^Z &= T + P - Te^Z \\ &= T + \underbrace{P - T_{trunc}C}_{\delta_P} + T_{trunc}C - Te^Z \end{aligned}$$

That was the first step



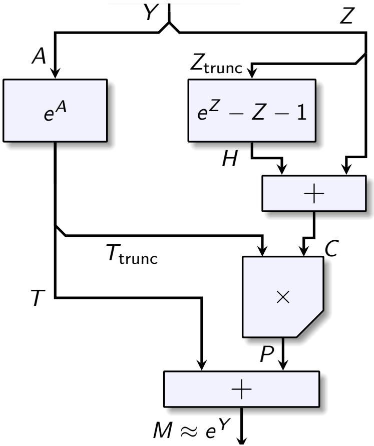
$$\delta_{\text{total}} = M - Te^Z + \delta_T$$

Where can we go from here?
Last addition is exact (that's fixed-point for you) so $M = T + P$, hence :

$$\begin{aligned} M - Te^Z &= T + P - Te^Z \\ &= T + \underbrace{P - T_{\text{trunc}} C}_{= \delta_P} + T_{\text{trunc}} C - Te^Z \end{aligned}$$

The bound on δ_P depends on the technology used for the multiplier (at most $\overline{\delta}_P = 2^{-w_F - g}$)
anyway it is under control

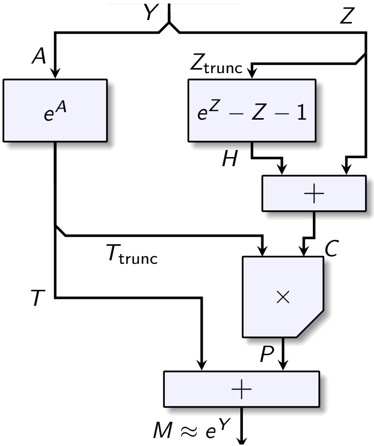
More of the same



$$\delta_{\text{total}} = T + T_{\text{trunc}} C - Te^Z + \delta_T + \delta_P$$

Where can we go from here?

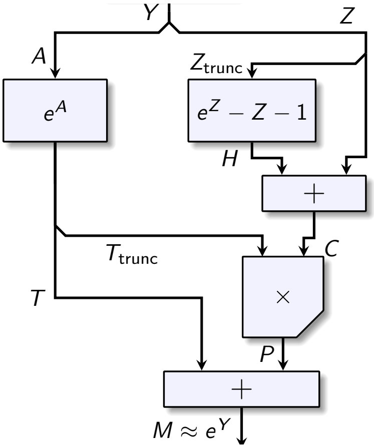
More of the same



$$\delta_{\text{total}} = T + T_{\text{trunc}}C - Te^Z \quad + \delta_T + \delta_P$$

Where can we go from here?
This T_{trunc} is annoying, so let's get it out of the way

$$T + T_{\text{trunc}}C - Te^Z = T + \underbrace{T_{\text{trunc}}C - TC}_{\substack{\| \\ (T_{\text{trunc}} - T)C \\ \| \\ \delta_{T_{\text{trunc}}}}} + TC - Te^Z$$



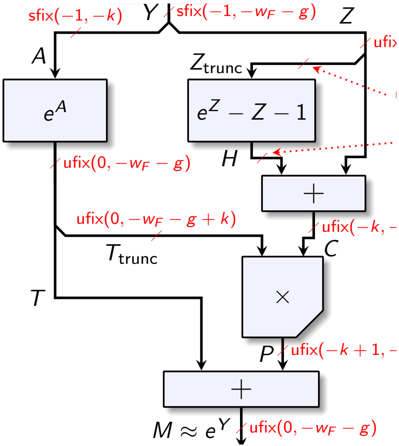
$$\delta_{total} = T + T_{trunc}C - Te^Z \quad + \delta_T + \delta_P$$

Where can we go from here?
This T_{trunc} is annoying, so let's get it out of the way

$$T + T_{trunc}C - Te^Z = T + \underbrace{T_{trunc}C - TC}_{\substack{\| \\ (T_{trunc} - T)C \\ \| \\ \delta_{T_{trunc}}}} + TC - Te^Z$$

$T_{trunc} - T < 2^{-w_F - g + k}$, then we need a bound on C;
$C \approx e^Z - 1$ so Taylor is our friend again

More of the same



$$\delta_{\text{total}} = T + T_{\text{trunc}}C - Te^Z \quad + \delta_T + \delta_P$$

Where can we go from here?

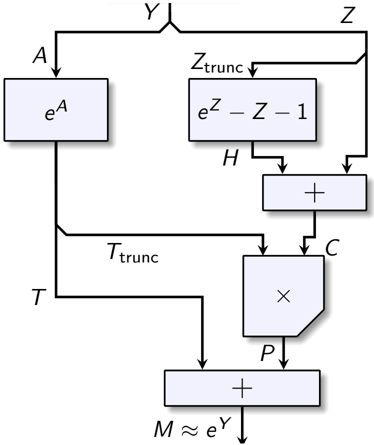
This T_{trunc} is annoying, so let's get it out of the way

$$T + T_{\text{trunc}}C - Te^Z = T + \underbrace{T_{\text{trunc}}C - {\color{blue}TC}}_{} + {\color{blue}TC} - Te^Z$$

$$\| \\ (T_{\text{trunc}} - T)C \\ \| \\ \delta_{T_{\text{trunc}}}$$

$T_{\text{trunc}} - T < 2^{-w_F-g+k}$, then we need a bound on C;
$C \approx e^Z - 1$ so Taylor is our friend again

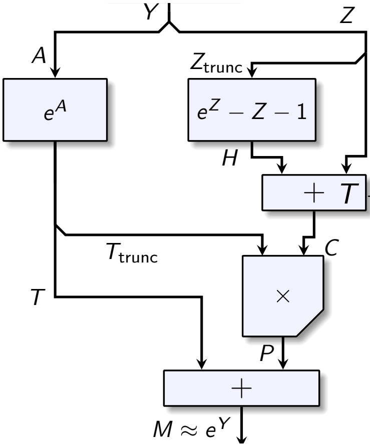
You'll get your lunch only after I get to an approximation error



$$\delta_{\text{total}} = T + TC - Te^Z + \delta_T + \delta_P + \delta_{T_{\text{trunc}}}$$

Where can we go from here?

You'll get your lunch only after I get to an approximation error



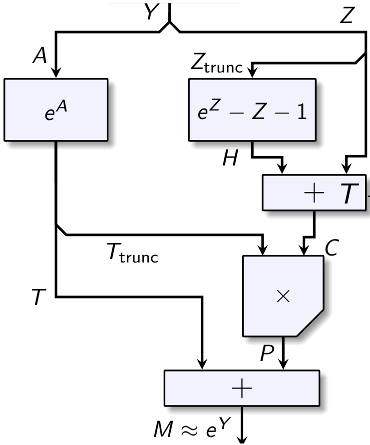
$$\delta_{\mathrm{total}} = T + TC - Te^Z \quad + \delta_T + \delta_P + \delta_{T_{\mathrm{trunc}}}$$

Where can we go from here?

$C = H + Z$ exactly (fixed-point additions are exact)

$$\begin{aligned}
T + TC - Te^Z &= T \cdot (1 + H + Z - e^Z) \\
&= T \cdot (H - h(Z)) \quad \text{with} \quad h(Z) = e^Z - Z - 1 \\
&= T \cdot (H {\color{blue}- h(Z_{\mathrm{trunc}}) + h(Z_{\mathrm{trunc}})} - h(Z)) \\
&= \underbrace{T \cdot (H - h(Z_{\mathrm{trunc}}))}_{\delta_H} + \underbrace{T \cdot (h(Z_{\mathrm{trunc}}) - h(Z))}_{\delta_{Z_{\mathrm{trunc}}}}
\end{aligned}$$

You'll get your lunch only after I get to an approximation error



$$\delta_{\text{total}} = T + TC - Te^Z \quad + \delta_T + \delta_P + \delta_{T_{\text{trunc}}}$$

Where can we go from here?

$C = H + Z$ exactly (fixed-point additions are exact)

$$\begin{aligned}
T + TC - Te^Z &= T \cdot (1 + H + Z - e^Z) \\
&= T \cdot (H - h(Z)) \quad \text{with} \quad h(Z) = e^Z - Z - 1 \\
&= T \cdot (H - h(Z_{\text{trunc}}) + h(Z_{\text{trunc}}) - h(Z)) \\
&= \underbrace{T \cdot (H - h(Z_{\text{trunc}}))}_{\delta_H} + \underbrace{T \cdot (h(Z_{\text{trunc}}) - h(Z))}_{\delta_{Z_{\text{trunc}}}}
\end{aligned}$$

δ_H includes the approximation error $H - h(Z_{\text{trunc}})$

Finally, scientific precision sabotaging

$$\delta_{\text{total}} \quad = \quad \delta_T + \delta_P + \delta_{T_{\text{trunc}}} + \delta_H + \delta_{Z_{\text{trunc}}}$$

hence

$$\overline{\delta}_{\text{total}} \quad = \quad \overline{\delta}_T + \overline{\delta}_P + \overline{\delta}_{T_{\text{trunc}}} + \overline{\delta}_H + \overline{\delta}_{Z_{\text{trunc}}}$$

- If any of these terms is much smaller than the others, useless bits are being computed
- I'll hack at the hardware to make this error worse!
  - by moving a parameter up or down,
  - maybe adding a truncation somewhere...

Finally, scientific precision sabotaging

$$\delta_{\text{total}} = \delta_T + \delta_P + \delta_{T_{\text{trunc}}} + \delta_H + \delta_{Z_{\text{trunc}}}$$

hence

$$\overline{\delta}_{\text{total}} = \overline{\delta}_T + \overline{\delta}_P + \overline{\delta}_{T_{\text{trunc}}} + \overline{\delta}_H + \overline{\delta}_{Z_{\text{trunc}}}$$

- If any of these terms is much smaller than the others, useless bits are being computed
- I'll hack at the hardware to make this error worse!
  - by moving a parameter up or down,
  - maybe adding a truncation somewhere...

Oh, yes, I will also make sure that $\overline{\delta}_{\text{total}}$ is small enough to guarantee last-bit accuracy.

Take away messages

- Error analysis for performance, not only for accuracy
- Straightforward engineering based on additions and multiplications
- Strict and accurate worst-case analysis (amenable to formal proof)
- Perfectly captures how an early rounding error is amplified in the algorithm

And for you floating-point people, there exists a relative-error version

If A approximates B and B approximates C, then

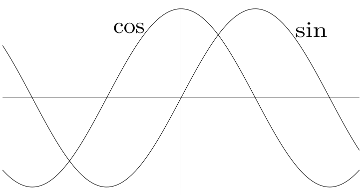
$$\frac{A-C}{C} = \frac{A-B}{B} + \frac{B-C}{C} + \frac{A-B}{B} \times \frac{B-C}{C} \qquad (7)$$

or

$$\varepsilon_{AC} = \varepsilon_{AB} + \varepsilon_{BC} + \varepsilon_{AB} \cdot \varepsilon_{BC}$$

Example: fixed-point sine/cosine

Introduction



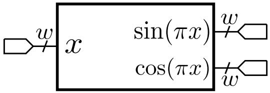
- Sine and cosine functions
  - fundamental in signal processing and signal processing applications like FFT, modulation/demodulation, frequency synthesizers, ...
- **How** to compute them ? In this work:
  1. the classical CORDIC algorithm, based on additions and shifts
  2. a method based on tables 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

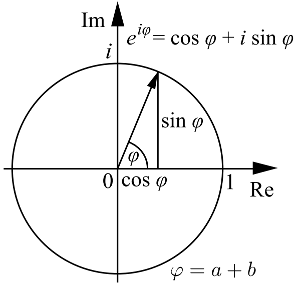
Textbook Stuff

- 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

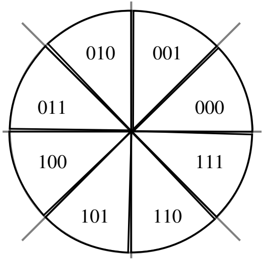
$$\begin{cases} \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b) \\ \cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b) \end{cases}$$

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:



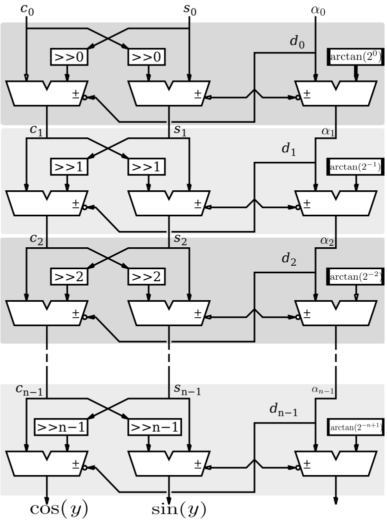
sqo	Reconstruction
000	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = \cos(\pi y') \end{cases}$
001	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = \sin(\pi y') \end{cases}$
010	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = -\sin(\pi y') \end{cases}$
011	$\begin{cases}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

CORDIC Architecture

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

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

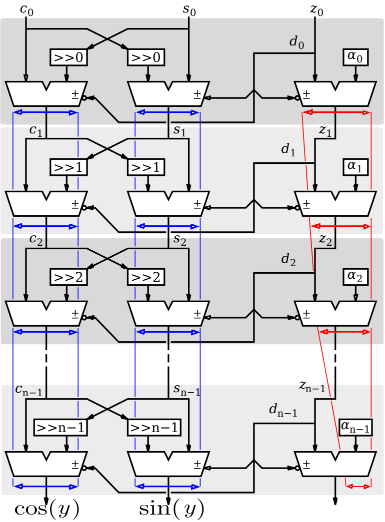
$$\begin{cases} c_{n\to\inf} &= \cos(y) \\ s_{n\to\inf} &= \sin(y) \\ \alpha_{n\to\inf} &= 0 \end{cases}$$



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

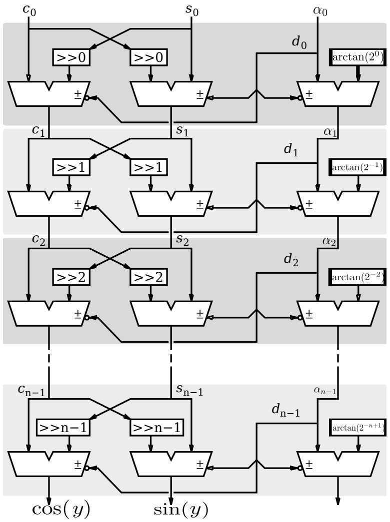
Reduced Iterations

- 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

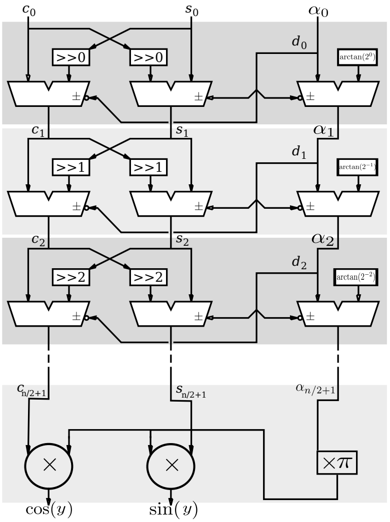
- stop iterations when they can be replaced by a single rotation, with enough accuracy

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

- 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

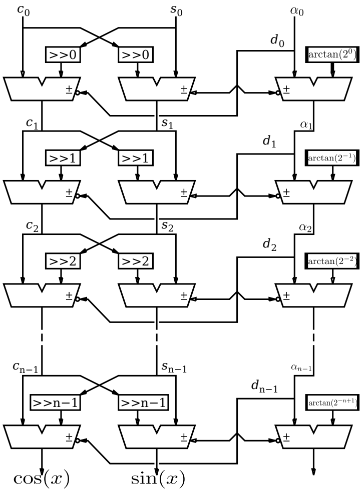
- the result is within 1**ulp** of the mathematical result
- **ulp** = weight of least significant bit

Intermediate precision

- approximations and roundings
  → 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

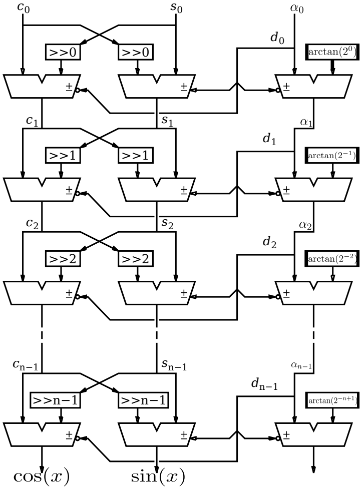


CORDIC Error Analysis (1)

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)

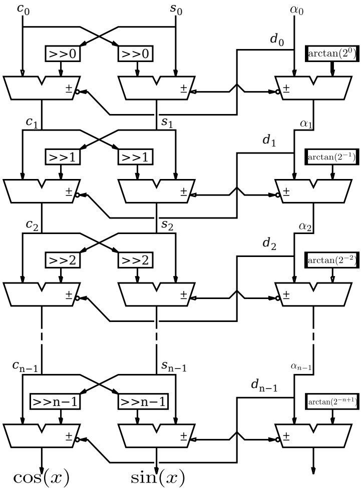
Analysis: rounding errors (ε_{round})

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)

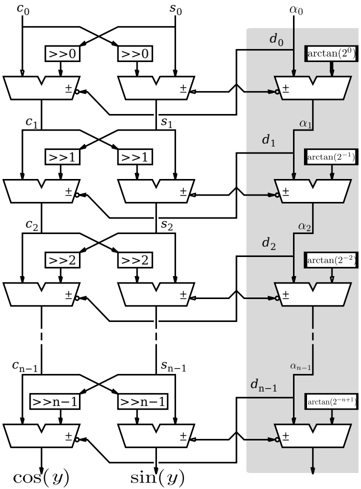
Analysis: rounding errors (ε_{round})

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
- $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$

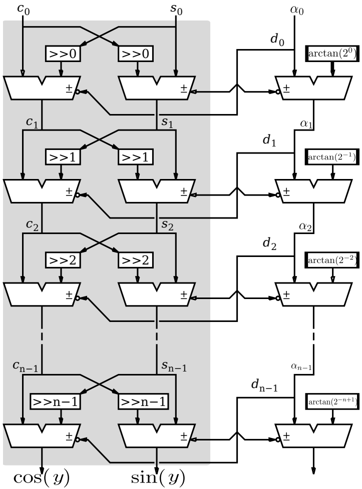


Table- and DSP-based method

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

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

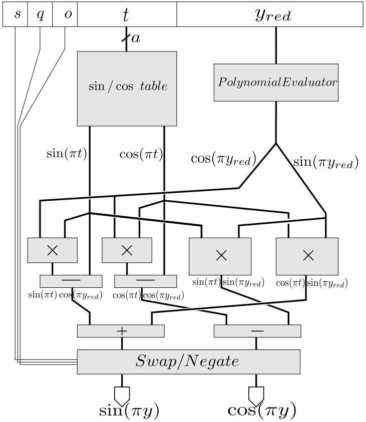


Table- and DSP-based method

s	q	o	t	y_{red}

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

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

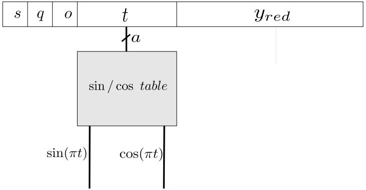


Table- and DSP-based method

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

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

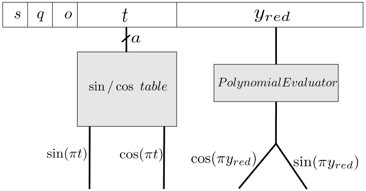


Table- and DSP-based method

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

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

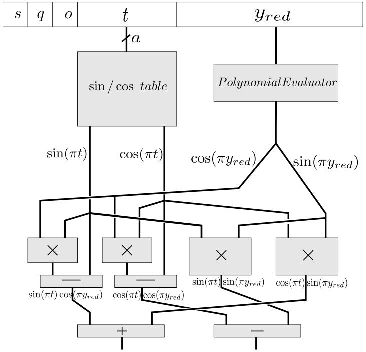


Table- and DSP-based method

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

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

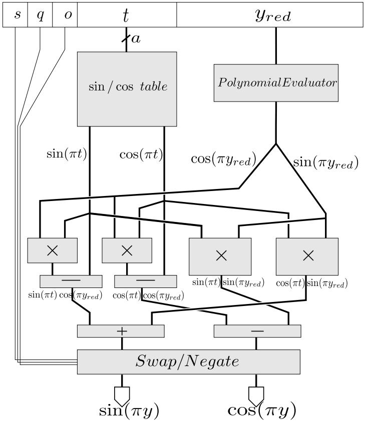
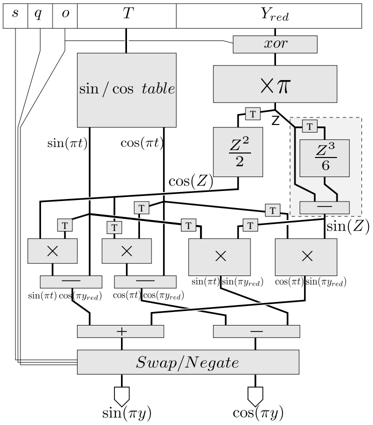


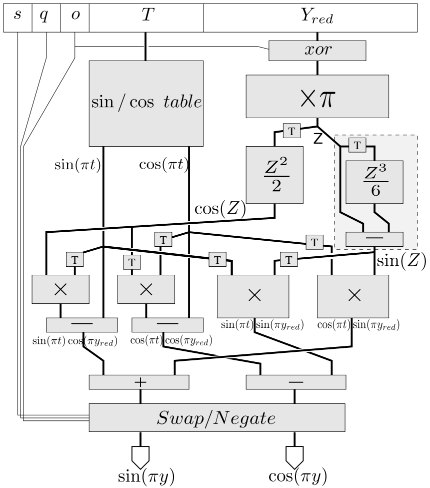
Table- and DSP-based method: Details

- 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)

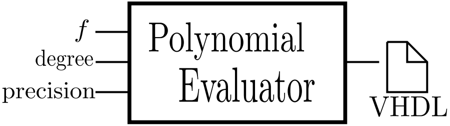


Error Analysis

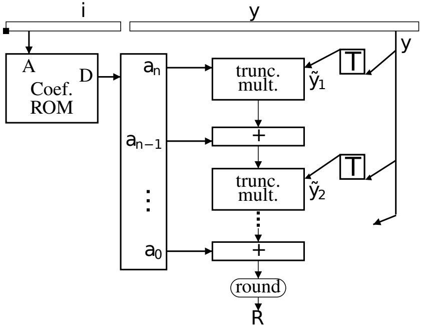
- $\frac{1}{2}$**ulp** lost per table
- 1**ulp** per truncation and truncated multiplier/squarer
- 1**ulp** for computing $\frac{1}{4} - y_{red}$ (as $\neg y_{red}$)
- total of 15**ulp**, independent of the input width
- $\rightarrow$ gives **g=4**



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?

Example: floating-point sums and sums of products

Floating-point accumulation

Summing a large number of **floating-point** *terms fast and accurately*

Crucial for:

- **Scientific computations:**
  - dot-product, matrix-vector product, matrix-matrix product
  - numerical integration
- **Financial simulations:**
  - Monte-Carlo simulations
- ...

Floating-Point(FP) numbers

normalized binary FP number:

$$x = (-1)^S \times 1.f \times 2^e$$

where:

- S - the **sign** of x
- f - the **fraction** of x.
- e - the **exponent** of x

Floating-Point(FP) numbers

normalized binary FP number:

$$x = (-1)^S \times 1.f \times 2^e$$

where:

S - the **sign** of x

f - the **fraction** of x.

e - the **exponent** of x

- e gives the dynamic range
  - IEEE-754 FP **double precision**, e_{min}=-1022 and $e_{max} = 1023$

Floating-Point(FP) numbers

normalized binary FP number:

$$x = (-1)^S \times 1.f \times 2^e$$

where:

S - the **sign** of x

f - the **fraction** of x.

e - the **exponent** of x

- e gives the dynamic range
  - IEEE-754 FP **double precision**, e_{min}=-1022 and $e_{max} = 1023$
- number of bits of f gives the **precision** p
  - IEEE-754 FP **double precision**, p=52

Floating-Point(FP) numbers

normalized binary FP number:

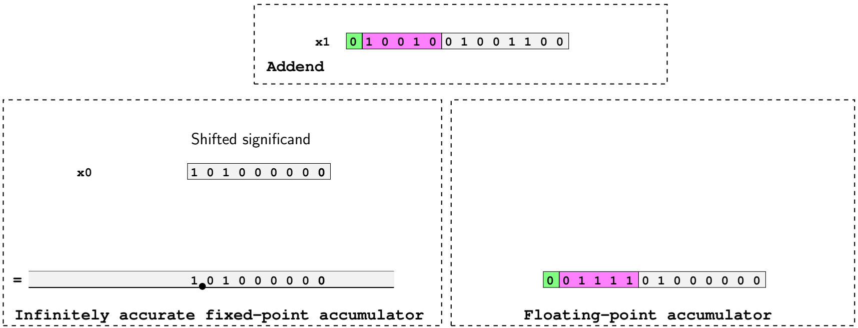
$$x = (-1)^S \times 1.f \times 2^e$$

where:

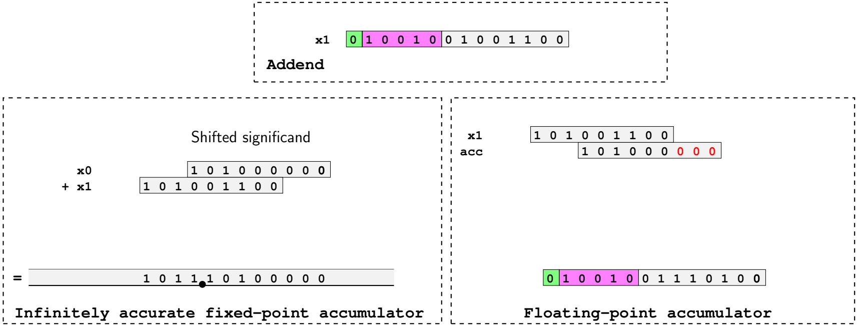
- S - the **sign** of x
- f - the **fraction** of x.
- e - the **exponent** of x

Graphical representation:

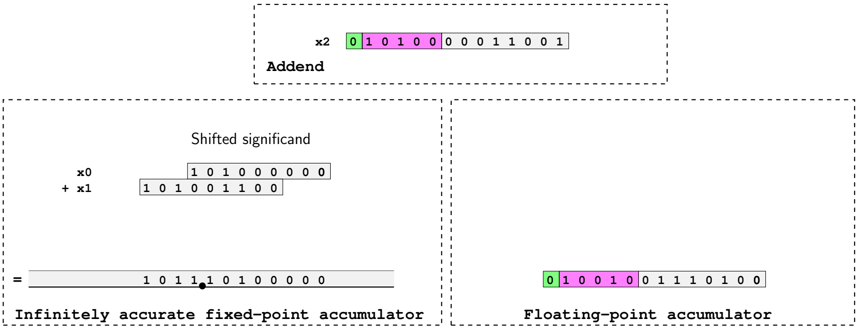
Accumulation



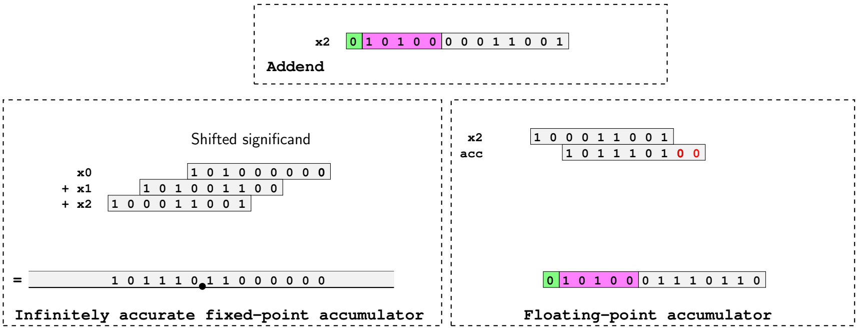
Accumulation



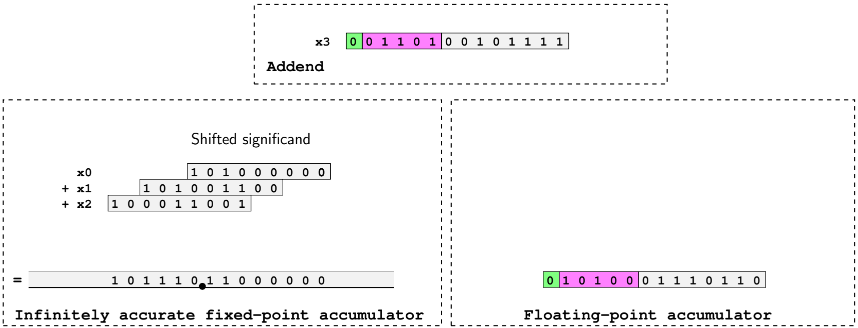
Accumulation



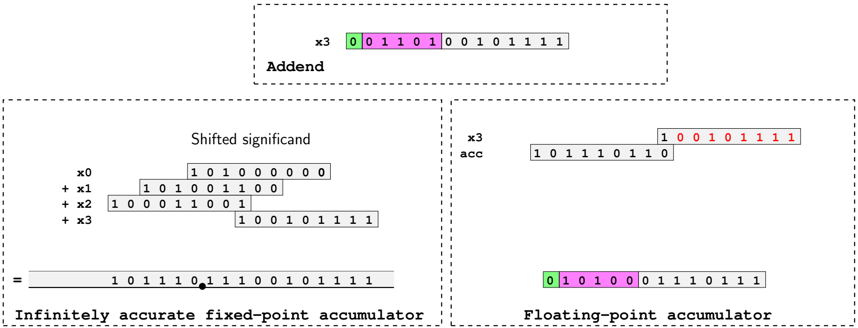
Accumulation



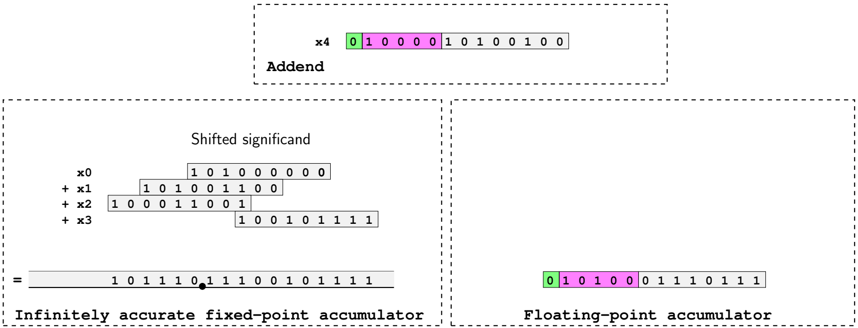
Accumulation



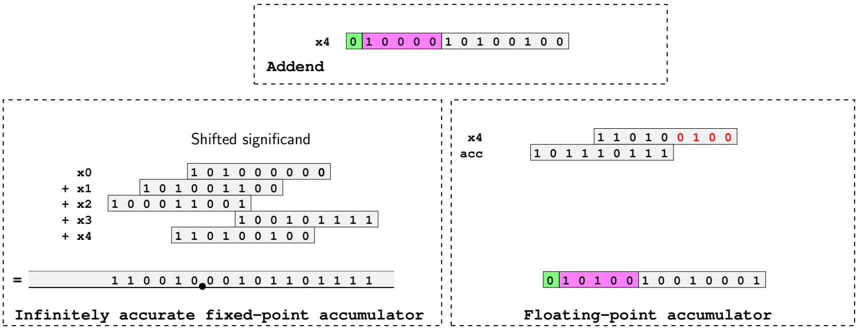
Accumulation



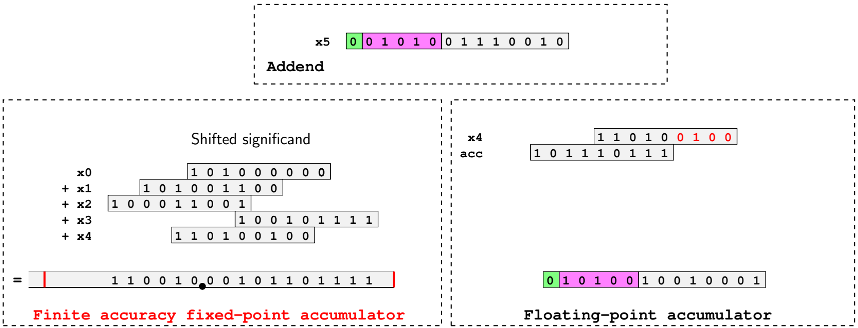
Accumulation



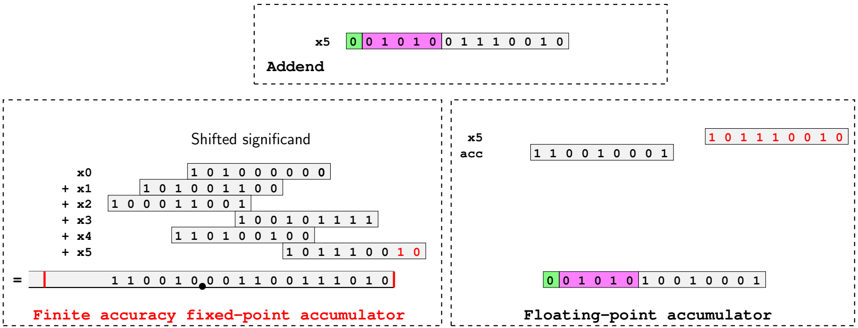
Accumulation



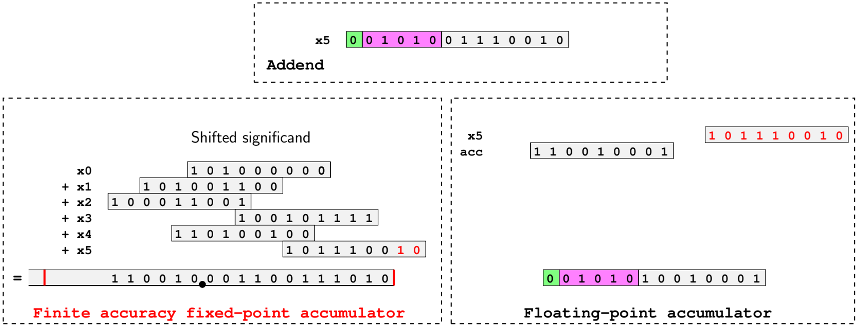
Accumulation



Accumulation



Accumulation



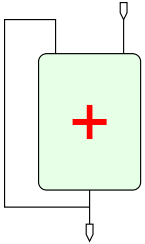
Accuracy:

Exact Result = 50.2017822265625

FP Acc = 50.125

Fixed-Point Acc = 50.20166015625

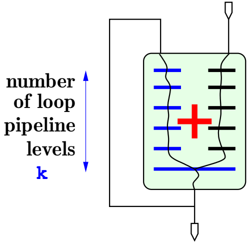
Closer look



Accumulator based on combinatorial floating-point adder

- very low frequency
- must pipeline for larger frequency

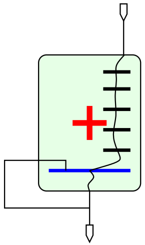
Closer look



Accumulator based on pipelined floating-point adder

- loop's critical path contains 2 shifters
- shifters are deeply pipelined
- produces k accumulation results
- these results have to be added somehow
  - adder tree
  - multiplexing mechanism on accumulation loop

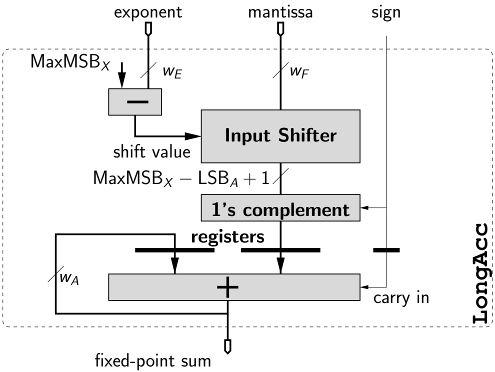
Closer look



Accumulator based on proposed long accumulator

- no shifts on the loop's critical path
- returns the result of the accumulation in fixed point
- the alignment shifter pipeline depth does not concern the result

Accumulator Architecture



- the sum is kept as a **large fixed-point number**
- one **alignment shift** (size depends on $MaxMSB_X$ and LSB_A)
- the loop's **critical path** contains a **fixed-point addition**
- fixed-point addition is fast on current FPGAs

Fast Accumulator Design

The accumulator should run at a target frequency

Fast Accumulator Design

The accumulator should run at a target frequency

- 64-bit addition works at 220MHz on Xilinx Virtex4 FPGA due to fast-carry chains

Fast Accumulator Design

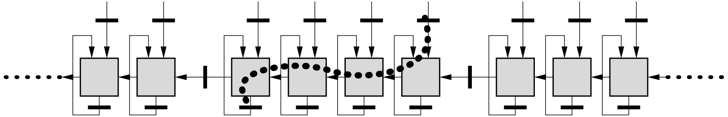
The accumulator should run at a target frequency

- 64-bit addition works at 220MHz on Xilinx Virtex4 FPGA due to fast-carry chains
- still not enough ?

Fast Accumulator Design

The accumulator should run at a target frequency

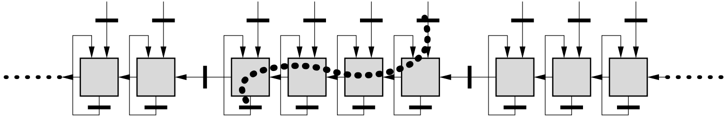
- 64-bit addition works at 220MHz on Xilinx Virtex4 FPGA due to fast-carry chains
- still not enough ?
- use *partial carry-save representation*
  - cut large carry-propagation into chunks of k bits
  - critical path $=$ k-bit addition
  - small cost: $\lfloor width_{accumulator} / k \rfloor$ registers



Fast Accumulator Design

The accumulator should run at a target frequency

- 64-bit addition works at 220MHz on Xilinx Virtex4 FPGA due to fast-carry chains
- still not enough ?
- use *partial carry-save representation*
  - cut large carry-propagation into chunks of k bits
  - critical path $=$ k-bit addition
  - small cost: $\lfloor width_{accumulator}/k \rfloor$ registers



- shifters can be arbitrarily pipelined for a given frequency

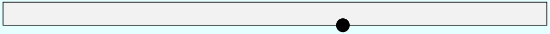
We advocate:

An **application tailored** fixed-point accumulator
for **floating-point inputs**

Ensuring that:

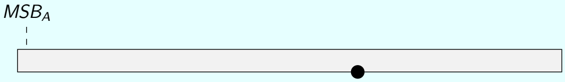
1. accumulator significand never needs to be shifted
2. it never overflows
3. provides a **result as accurate as the application requires**

Accumulator Parameters



The designer must provide values for these parameters.

Accumulator Parameters

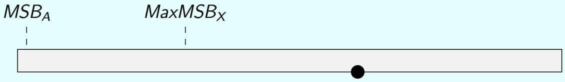


MSB_A the weight of the MSB of the accumulator

- must to be larger than max. expected result

The designer must provide values for these parameters.

Accumulator Parameters



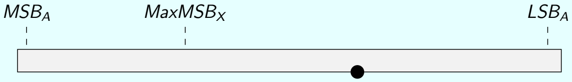
MSB_A the weight of the MSB of the accumulator

- must to be larger than max. expected result

$MaxMSB_X$ the max. weight of the MSB of the summand

The designer must provide values for these parameters.

Accumulator Parameters



MSB_A the weight of the MSB of the accumulator

- must to be larger than max. expected result

$MaxMSB_X$ the max. weight of the MSB of the summand

LSB_A weight of the LSB of the accumulator

- determines the final accumulation accuracy

The designer must provide values for these parameters.

Application Tailored

Application dictates parameter values

Application Tailored

Application dictates parameter values

Two possibilities:

- **software profiling** + safety margins
- **rough error analysis** + safety margins

Application Tailored

Application dictates parameter values

Two possibilities:

- **software profiling** + safety margins
- **rough error analysis** + safety margins

How to chose the parameters using the rough error analysis ?

Application Tailored

Application dictates parameter values

Two possibilities:

- **software profiling** + safety margins
- **rough error analysis** + safety margins

How to chose the parameters using the rough error analysis ?

MSB_A
- know an actual maximum + 10 bits safety margin
- consider the number of terms to sum

Application Tailored

Application dictates parameter values

Two possibilities:

- **software profiling** + safety margins
- **rough error analysis** + safety margins

How to chose the parameters using the rough error analysis ?

MSB_A
- know an actual maximum + 10 bits safety margin
- consider the number of terms to sum

$MaxMSB_X$
- exploit input properties + safety margin
- worst case: $MaxMSB_X = MSB_A$

Application Tailored

Application dictates parameter values

Two possibilities:

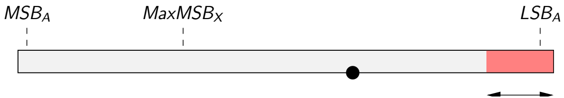
- **software profiling** + safety margins
- **rough error analysis** + safety margins

How to chose the parameters using the rough error analysis ?

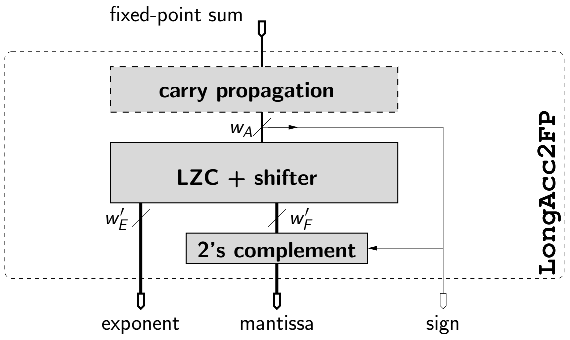
MSB_A
- know an actual maximum + 10 bits safety margin
- consider the number of terms to sum

$MaxMSB_X$
- exploit input properties + safety margin
- worst case: $MaxMSB_X = MSB_A$

LSB_A **precision** vs. **performance**
- consider the desired final precision
- sum n terms, at most $\log_2 n$ bits are invalid

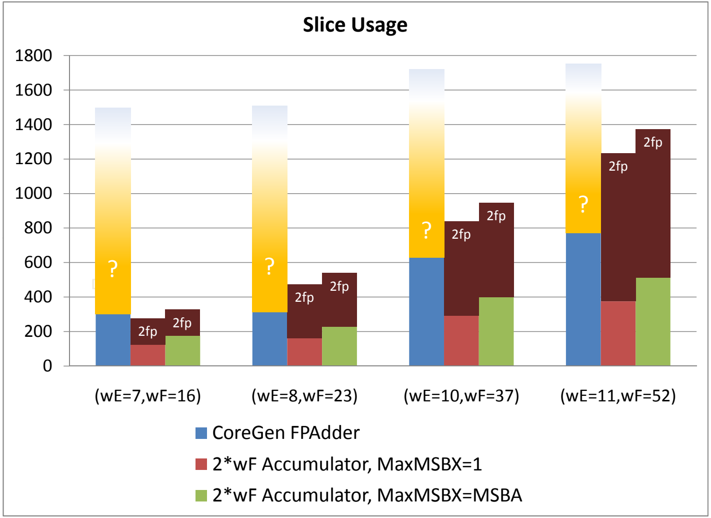


Post-normalization unit, or not

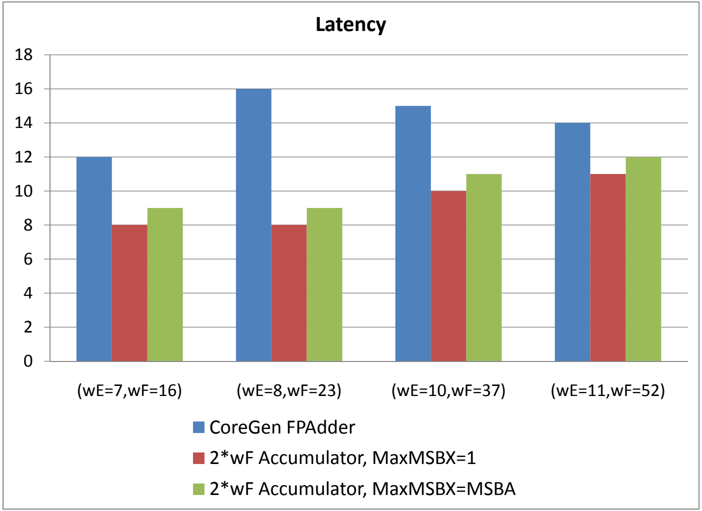


- converts fixed-point accumulator format to floating-point
- pipelined unit may be shared by several accumulators
- less useful:
  - many applications do not need the running sum
  - better to do conversion in software, use FPGA to accelerate the computation

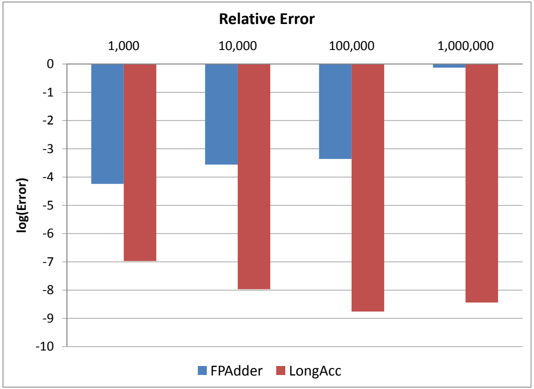
Performance results



Performance results



Relative error results



Accumulation of FP($w_E = 7, w_F = 16$) in unif. [0,1]

- LongAcc ($MSB_A = 20$, $LSB_A = -11$)

Accurate Sum-of-Products

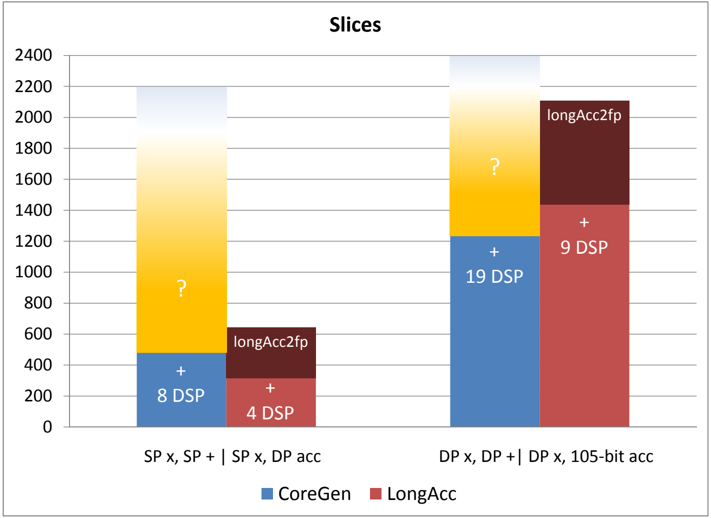
Ideea

Accumulate **exact** results of all multiplications

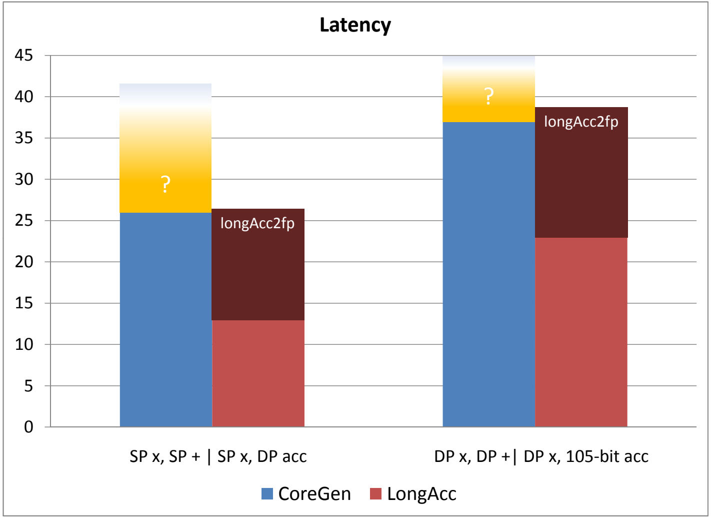
1. Use exact multipliers:
   - return all the bits of the exact product
   - contain no rounding logic
   - are cheaper to build
2. Feed the accumulator with exact multiplication results

Cost: Input shifter of accumulator is twice as large

Operator Performance



Operator Performance



The universal bit heap

Introduction and motivation

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

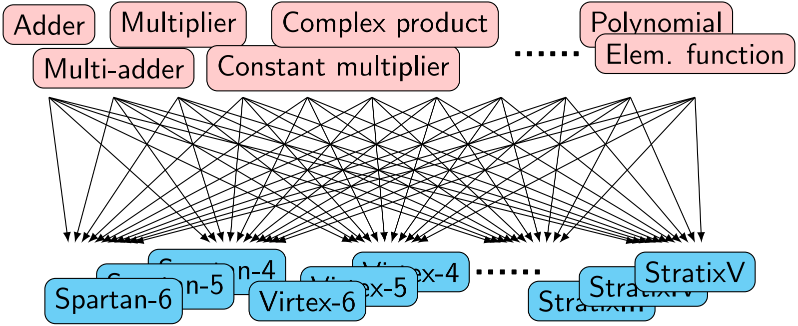
FPGA arithmetic the way it should be:

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

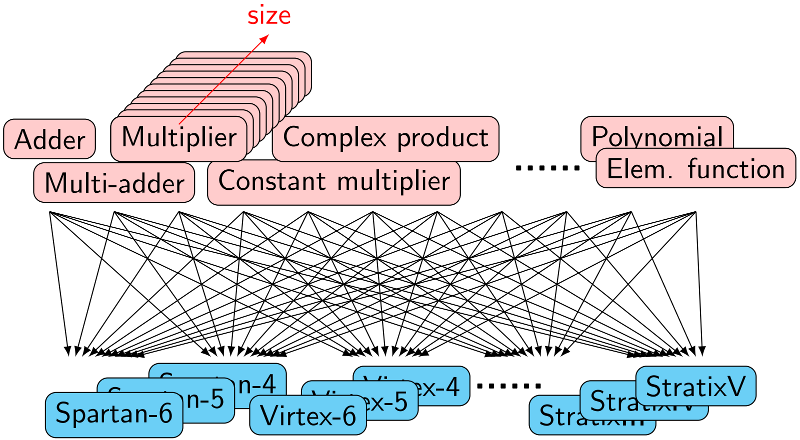
How to ensure **performance** across all this range?

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

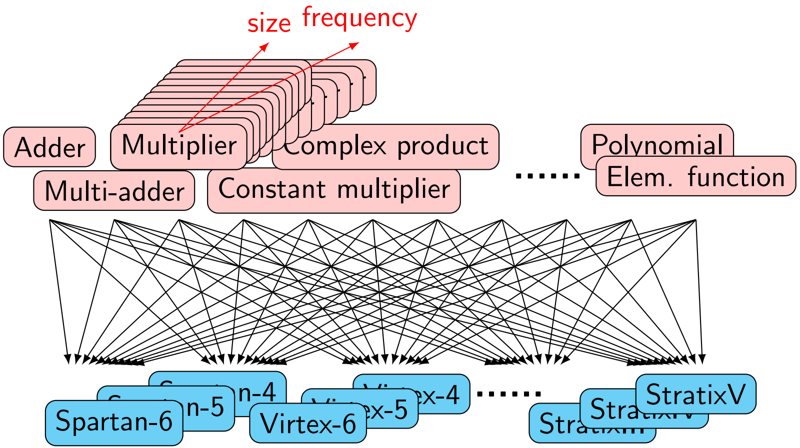
Portable versus optimized



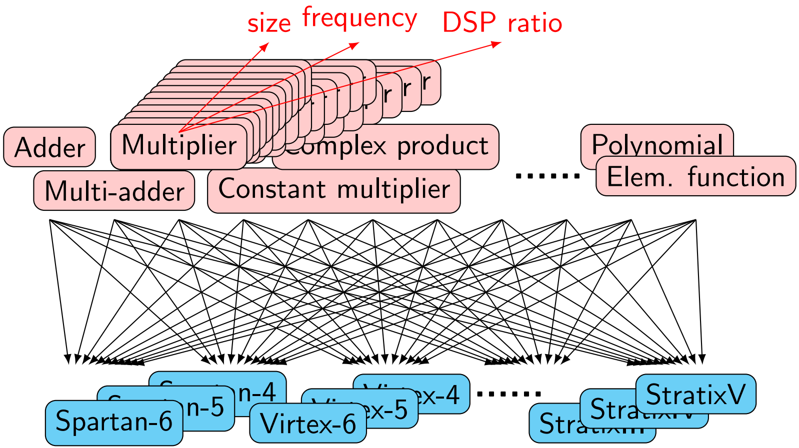
Portable versus optimized



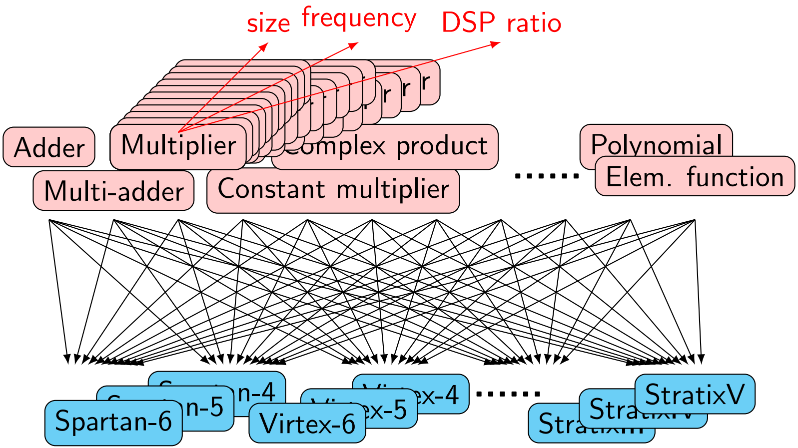
Portable versus optimized



Portable versus optimized

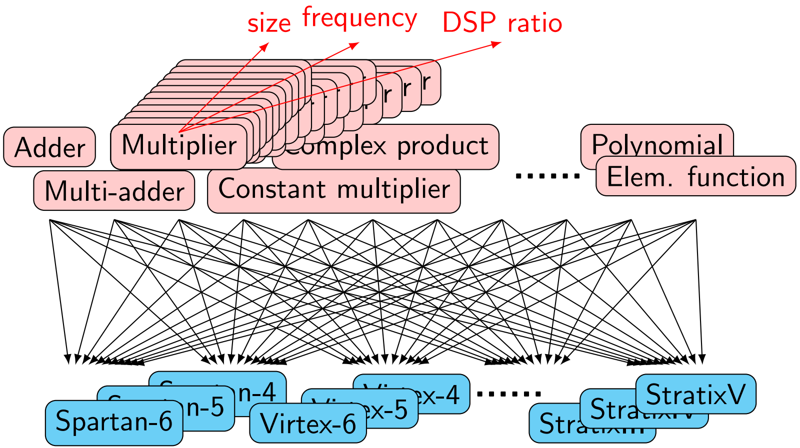


Portable versus optimized



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

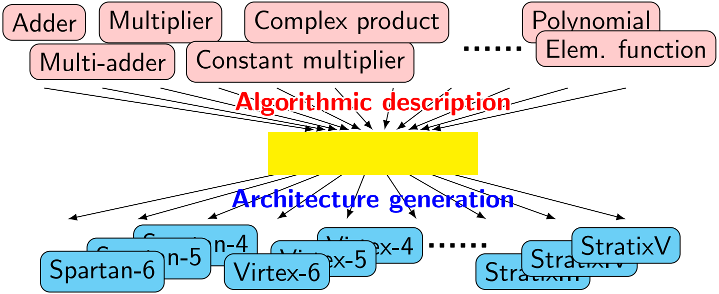
Portable versus optimized



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

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

Reducing the combinatorics

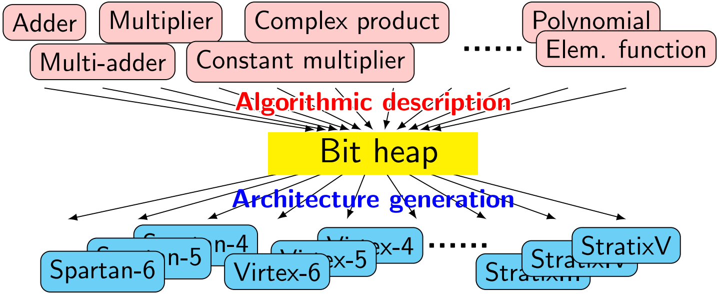


What is a bit heap?

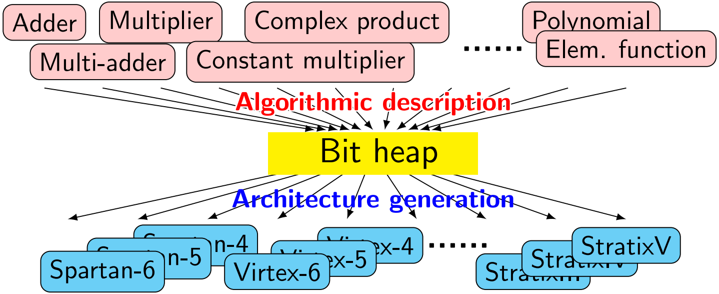
- A **data-structure**
  - capturing bit-level descriptions of a wide class of operators
  - exposing bit-level parallelism and optimization opportunities
- An associated **architecture generator**

which can be optimized for each target

Reducing the combinatorics



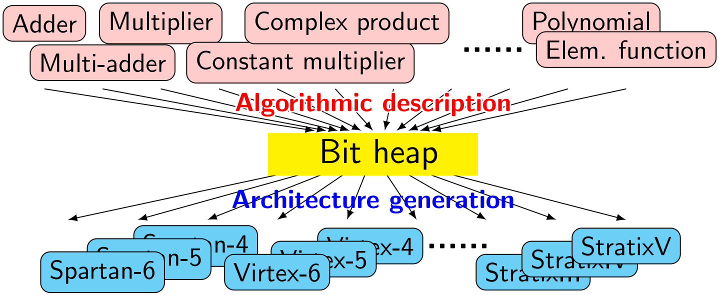
Reducing the combinatorics



What is a bit heap?

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

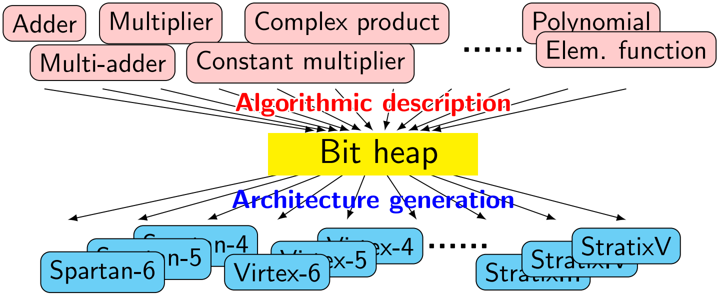
Reducing the combinatorics



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

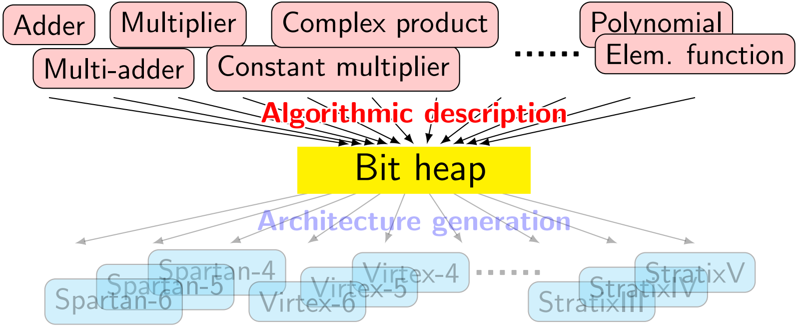


What is a bit heap?

- 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



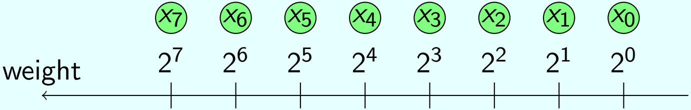
Weighted bits

- 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

	0	0	1	0	1	0	1	0
weight ←	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Example: 17.42 written in binary

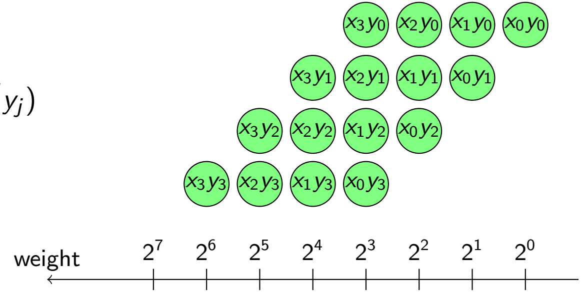
	1	0	0	0	1 .	1	0	1	0	1	1	1
weight ←	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

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

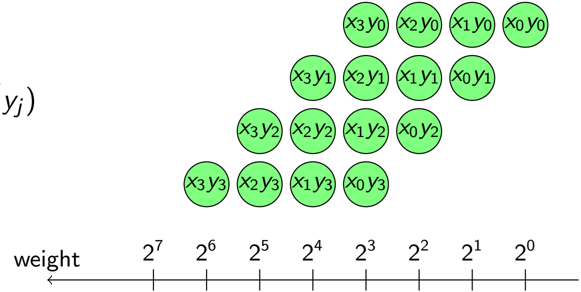
$$XY = \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right)$$
$$= \sum_{i,j} 2^{i+j} x_i y_j$$



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.

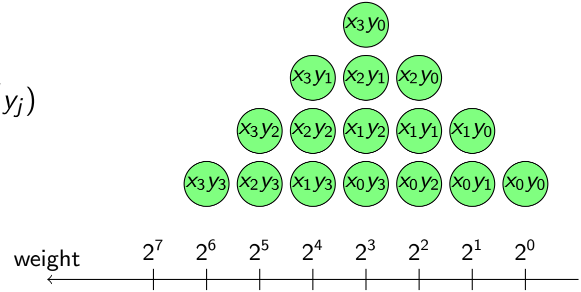
Historical motivation for bit heaps

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

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.

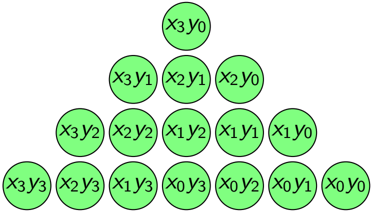
Historical motivation for bit heaps

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

(freedom thanks to addition associativity and commutativity)

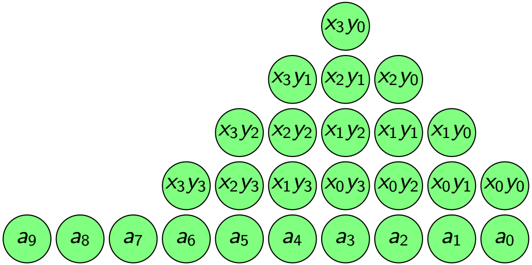
Beyond product

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



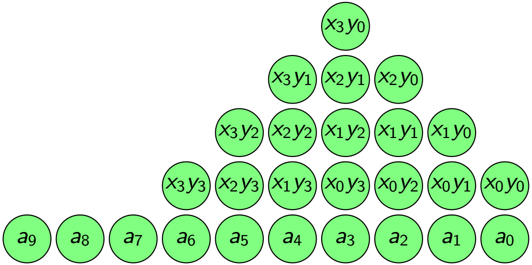
Beyond product

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



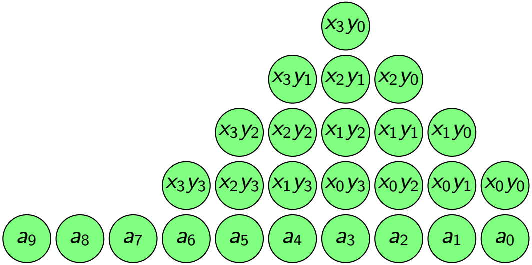
Beyond product

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



Beyond product

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



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 (and solved by ILP)

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

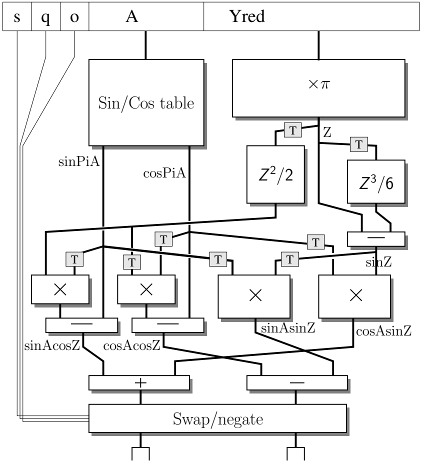
... where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

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

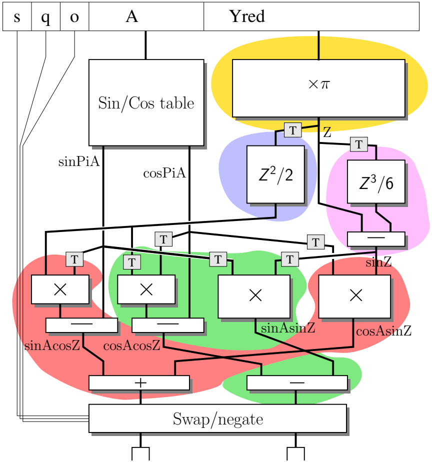
When you have a good hammer, you see nails everywhere

A sine/cosine architecture (HEART 2013)

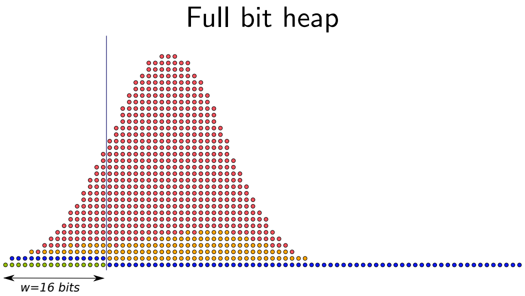


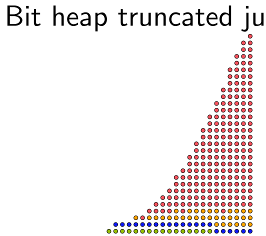
When you have a good hammer, you see nails everywhere

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



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





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)

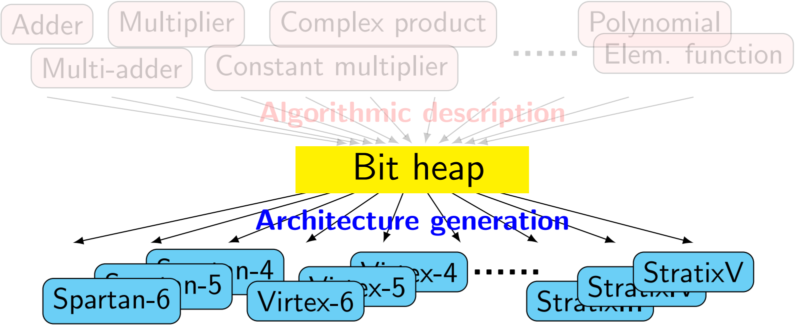
To replicate bit s from weight p to weight q

- 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



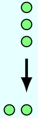
Architecture computing the value of a bit heap

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

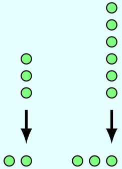


Architecture computing the value of a bit heap

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

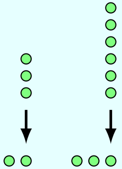
  tabulated in 3 6-input LUTs.
- survey and refs in the FPL 2013 paper, see also papers by M. Kumm.

Architecture computing the value of a bit heap

Elementary case 1: the compressor

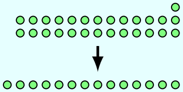
A compressor replaces a column of bits

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



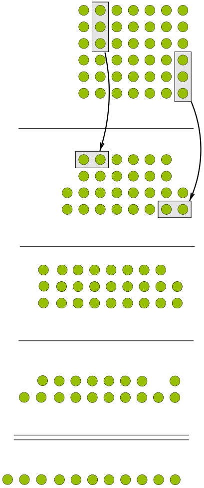
An adder replaces two n-bit lines, and a carry

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

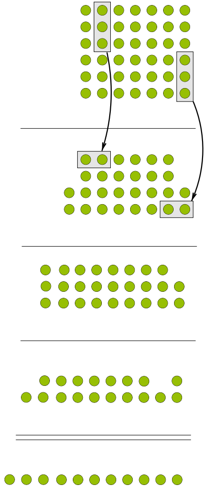


Architecture computing the value of a bit heap

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

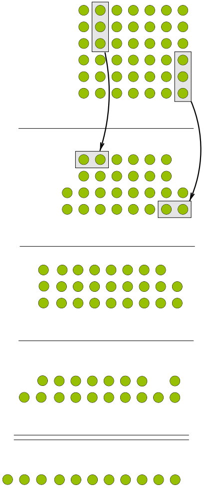


Architecture computing the value of a bit heap

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

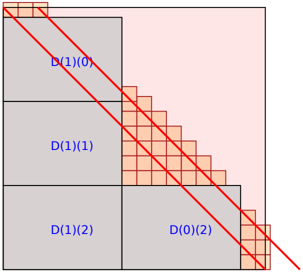
Elementary case: the DSP block?

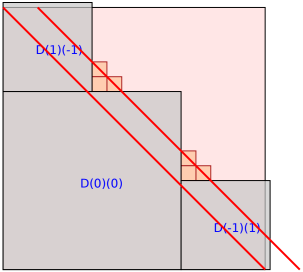
- Xilinx DSP blocks compute $\mathbf{A} + \mathbf{XY}$ (48+18x25)
- Altera DSP blocks compute $\mathbf{XY}$ (36x36)

or $\mathbf{AB} \pm \mathbf{CD}$ (18x18+18x18) or ...

Really different architectures here

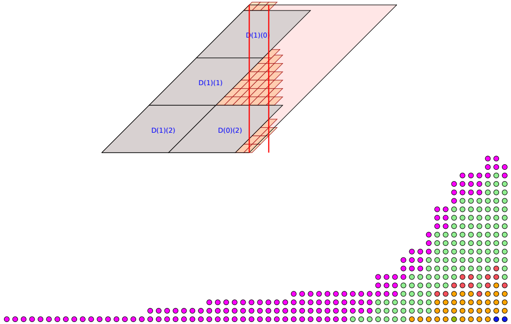
Exemple: 53-bit truncated multiplier

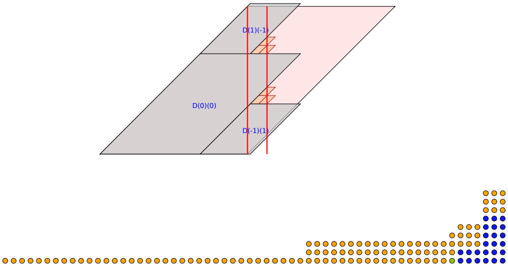




Instanciating DSP blocks is part of the compression

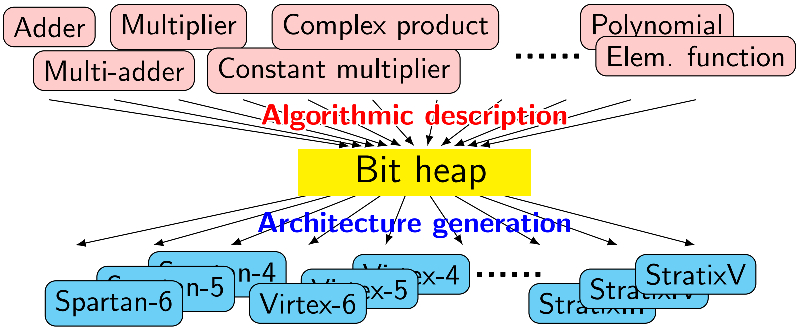
- merge operands from various sources in a DSP
- unused DSP adders may remove random bits from the heap





Stratix IV

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

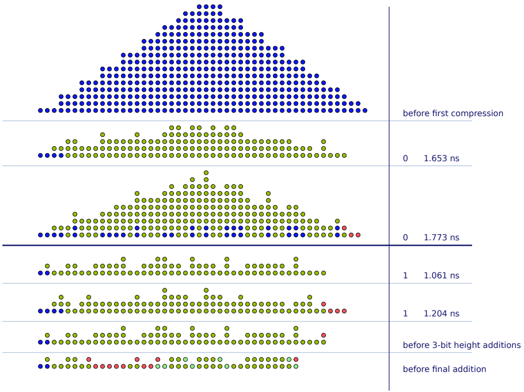
- Reduction of FloPoCo code size
- Fewer obscure bugs hidden in obscure operators
- (I didn't say fewer bugs)

Benefits in terms of performance thanks to operator fusion

- Already a few examples
  - complex product, KCM multipliers, FIR and IIR filters, ...
- Still work in progress
  - M. Kumm replaced initial heuristics with ILP-based optimal algorithms
  - 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)

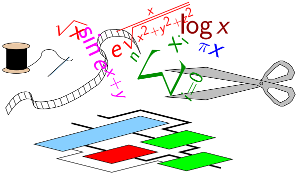
The difficult half is: how to use them?

- What precision is required at what point of a computation

Thanks for your attention

The following people contributed to FloPoCo:

S. Banescu, L. Besème, N. Bonfante, N. Brunie,
M. Christ, S. Collange, O. Desrentes, J. Detrey,
P. Echeverría, F. Ferrandi, L. 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, A. Volkova.



http://flopoco.org/