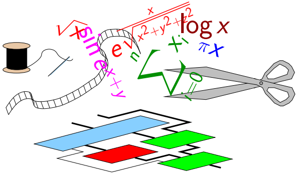
Application-specific arithmetic with FloPoCo

Florent de Dinechin



Outline

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

→ 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

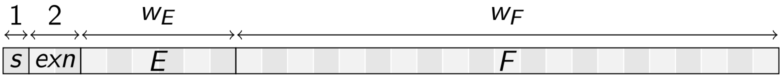
A complete single-precision FPU in a single VHDL file:

```
./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



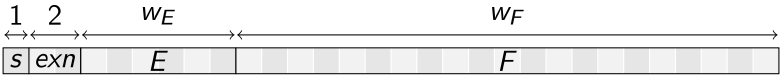
FloPoCo floating-point format

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



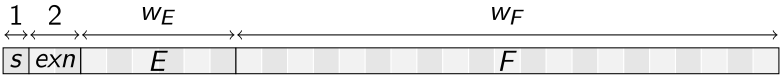
FloPoCo floating-point format

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

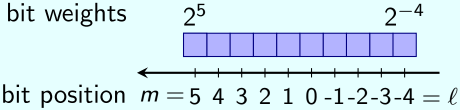
Number formats in FloPoCo

- The previous floating-point format
- A few operators for IEEE floating-point format
- Posits soon
- Logarithm Number System (LNS) in older versions
- One Obscure Branch contains decimal arithmetic
- Residue Number System (RNS) and other modular arithmetic should come some day

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

Fixed-point format

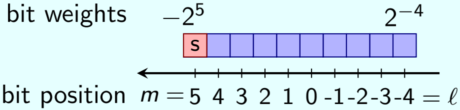
Parameters for an unsigned (positive) fixed-point format



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

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

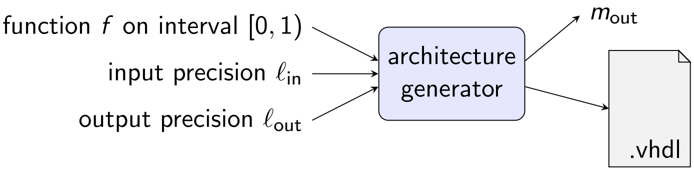
Parameters for a fixed-point format in two's complement



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

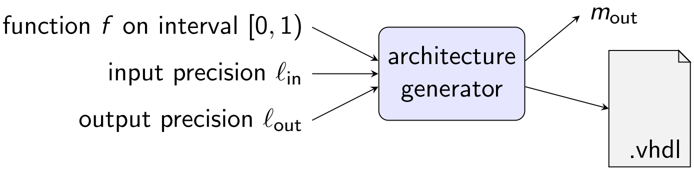
Integers have $\ell = 0, m > 0$.

Typical interface to a FloPoCo operator




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

Typical interface to a FloPoCo operator

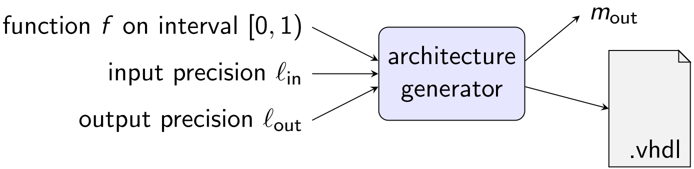



```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24
    msbOut=3 d=3
```

Output precision ℓ_{out} also specifies the accuracy of the architecture

Difference between computed value and $f(x)$ never larger than $2^{\ell_{\mathrm{out}}}$

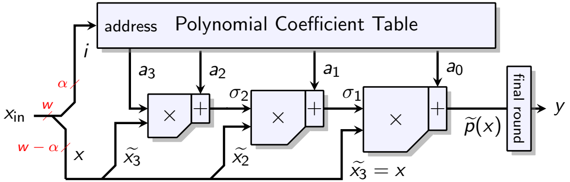
Typical interface to a FloPoCo operator




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

Output precision ℓ_{out} also specifies the accuracy of the architecture

Difference between computed value and $f(x)$ never larger than $2^{\ell_{out}}$



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
```

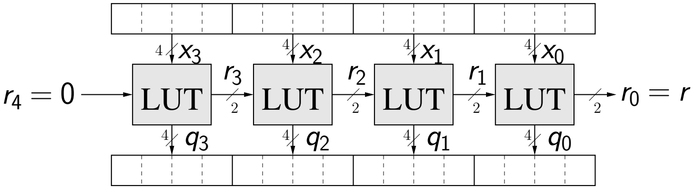
```
flopoco target=Virtex6 IntConstDiv wIn=16 d=3
```

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
```

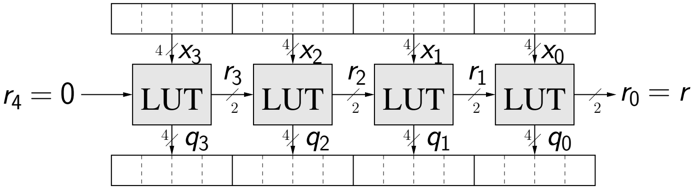


Also match the architecture to the target FPGA

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

Parenthesis: minimalist interfaces

In the previous example (an integer divider by 3) we didn't specify output size: FloPoCo computes it, too.

Parenthesis: minimalist interfaces

In the previous example (an integer divider by 3) we didn't specify output size: FloPoCo computes it, too.

More importantly,

When `lsbOut` is given, it also specifies the accuracy of the operator

Compute just right!

- No need to compute more accurately than $2^{\texttt{lsbOut}}$,
  we couldn't output it
- No sense in computing less accurately than $2^{\texttt{lsbOut}}$,
  we don't want to output garbage bits

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

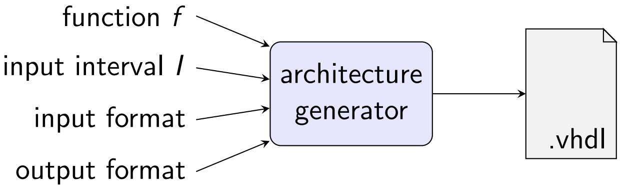
Not by simulation of the generated architecture!

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

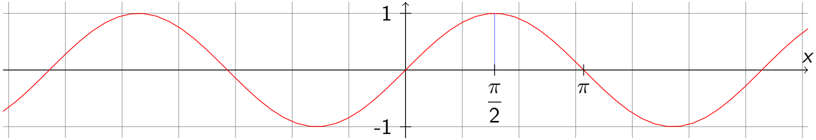
- `fp2bin 9 36 3.1415926`
- `bin2fp 9 36 010100000000100100100001111110110100110100010011`

Example: fixed-point functions

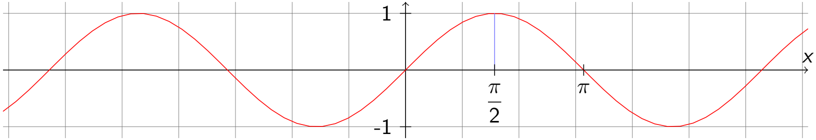
Generic generator of fixed-point functions



The sine function



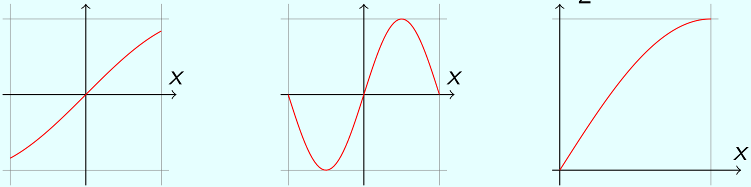
The sine function



Input format is in fixed point

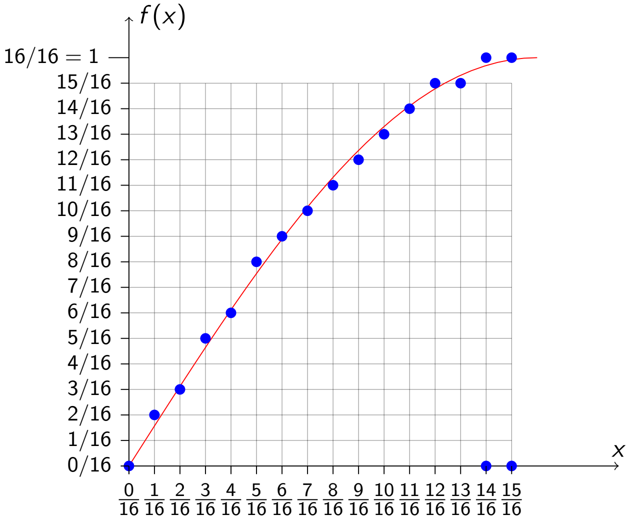
Arbitrary choice in FloPoCo: the input domain will be $[0, 1)$ or $[-1, 1)$.

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

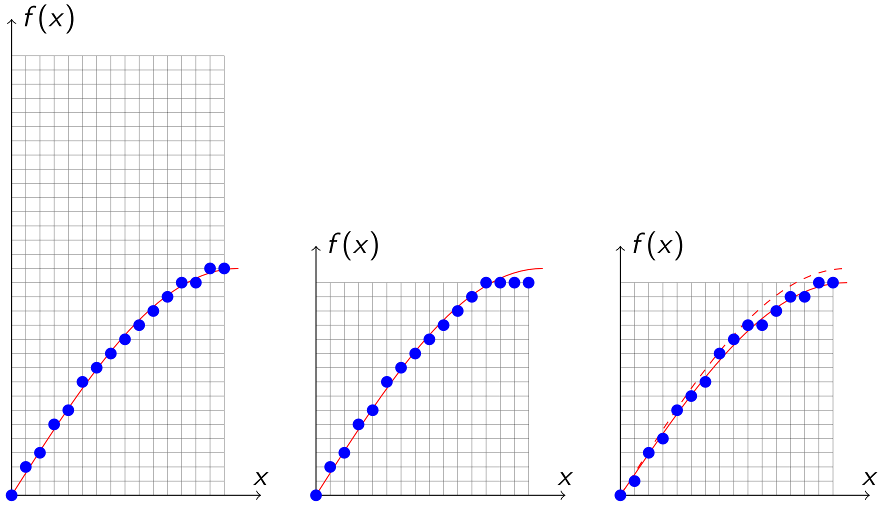


Discretization issues

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



Possible fixes for corner-case discretization issues

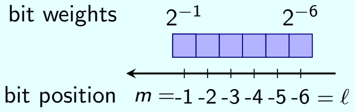


Using 1 bit more　　　saturating　　　Scaling by $\frac{15}{16}$

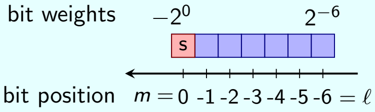
FixFunctionByTable

```
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.)

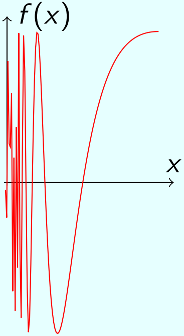
FixFunctionByTable, fixed

```
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

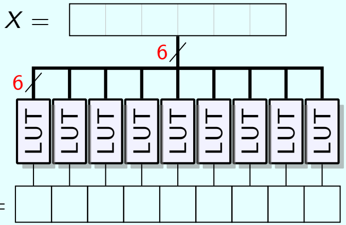
The previous example was a 16-bit in, 16-bit out.
(you just added 64 KLOC to your project)

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.

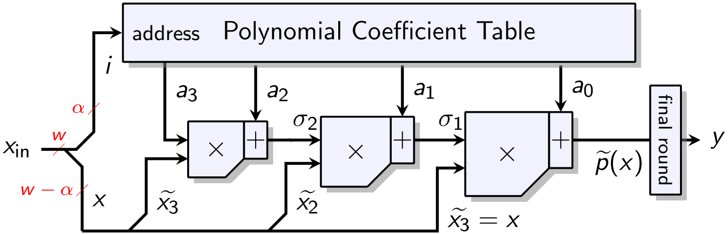


- A 20 Kb dual-port BlockRAM can hold two tables of $2^{10} \times 10$ bits.

When plain tables won't scale

... This is where FloPoCo can do clever stuff.

- The multipartite table + additions method: `FixFunctionByMultipartiteTable`
  - rule of thumb: cost grows as $2^{p/2} \times p$ instead of $2^p \times p$
  - but only works for functions that are **continuous**, **derivable**, and even **monotonic** on the domain.
- 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**



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 $0 \le x_i < 2^\alpha$

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

  α bits (←→)

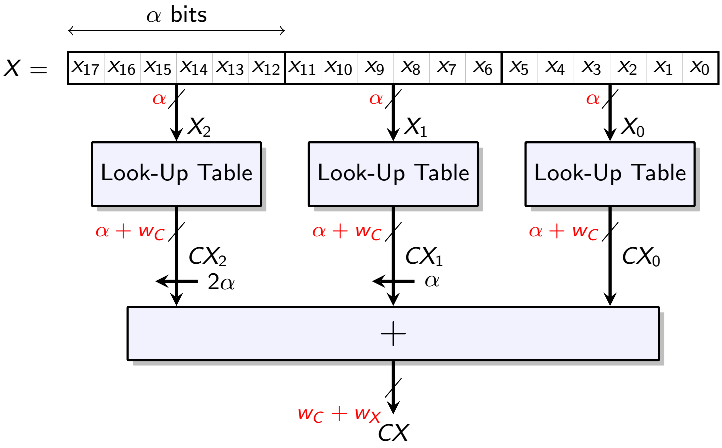
  $X =$

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

- 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?)

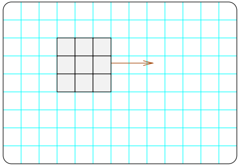
Extremely efficient for some constants such as 17.

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

Floating-point multiplication by a rational constant

Motivation

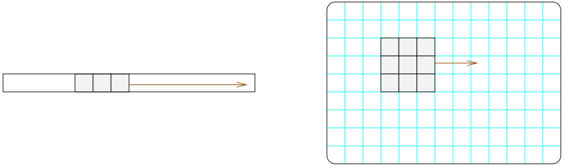
divisions by 3 and by 9 in stencil applications



Floating-point multiplication by a rational constant

Motivation

divisions by 3 and by 9 in stencil applications



$1/3 = 0.01010101010101010101010101010\cdots$

$1/9 = 0.000111000111000111000111000111\cdots$

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)

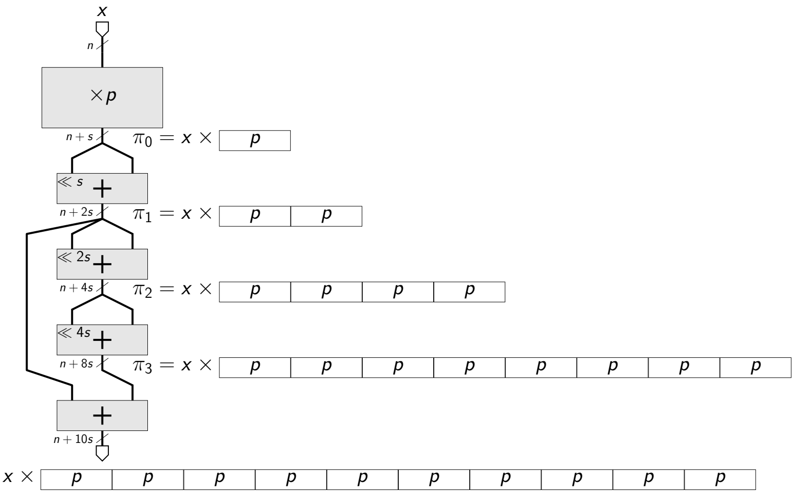
Then

- 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

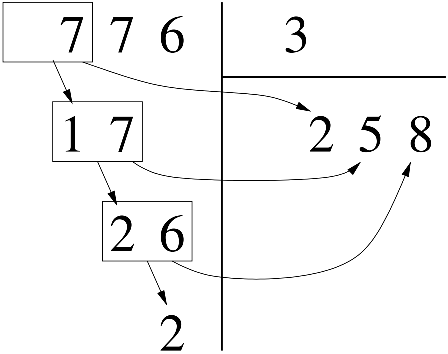
(The precisions chosen here are those of the IEEE754-2008 formats)

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

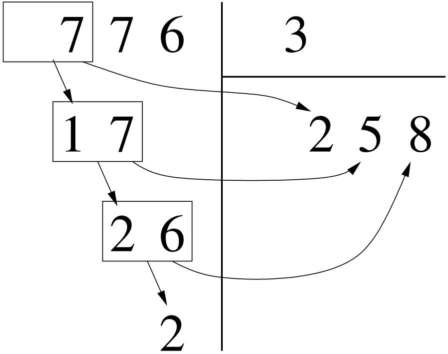
And now for something completely different

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

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

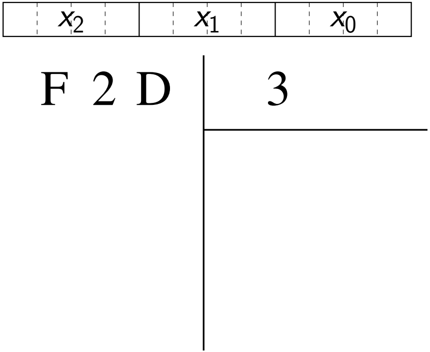
(split of the bits of x into chunks of α bits)

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 $\alpha = 4$



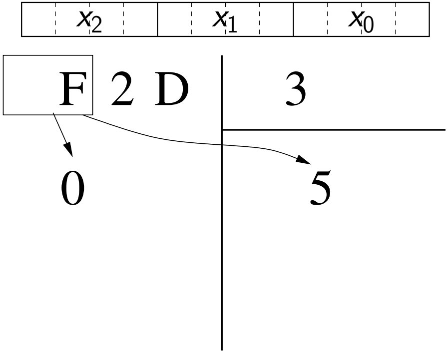
The same, but in binary-friendly radix

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 $\alpha = 4$



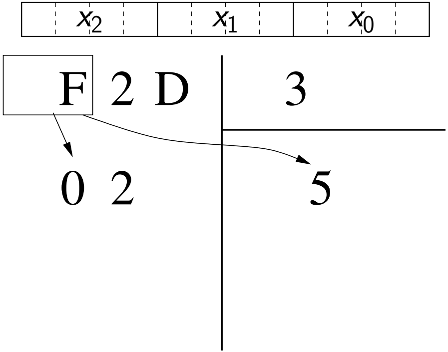
The same, but in binary-friendly radix

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 $\alpha = 4$



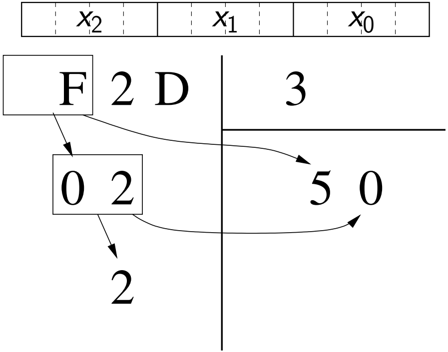
The same, but in binary-friendly radix

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 $\alpha = 4$

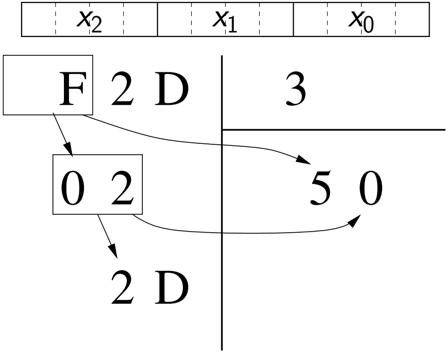


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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



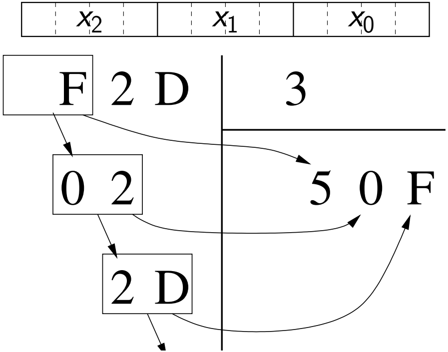
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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

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



And now for some mathematical obfuscation

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

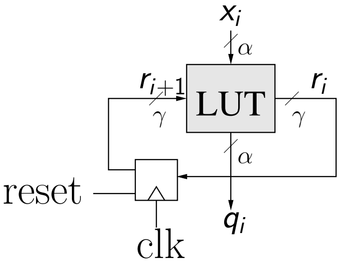
- consumes α bits of x, and a remainder of size $\gamma = \lceil \log_2 d \rceil$
- produces α bits of q, and a remainder of size γ
- implemented as a table with $\alpha + \gamma$ bits in, $\alpha + \gamma$ bits out

At this point nobody wants to see the proof

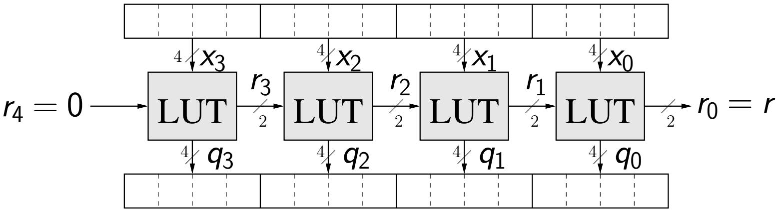
(if you're convinced the decimal version works...)

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

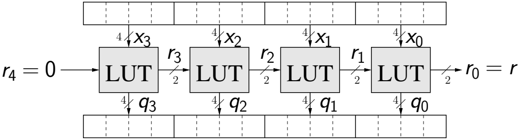
Sequential implementation



Unrolled implementation



Logic-based version

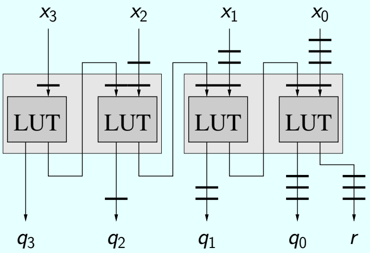


For instance, assuming a 6-input LUTs (e.g. LUT6)

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

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

Synthesis results on Virtex-5 for combinatorial Euclidean division

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

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

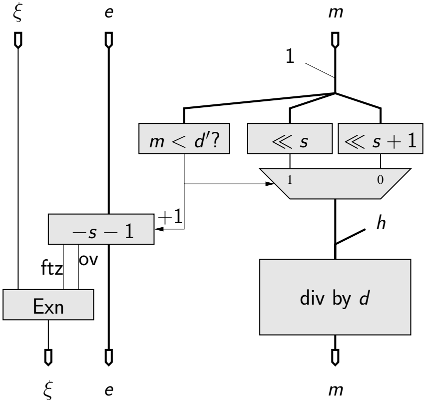
Logic optimizer even finds something to chew: *don't care* lines in the tables.

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

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

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

Floating-point version is cheap, too



- pre-normalisation and pre-rounding:

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

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

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

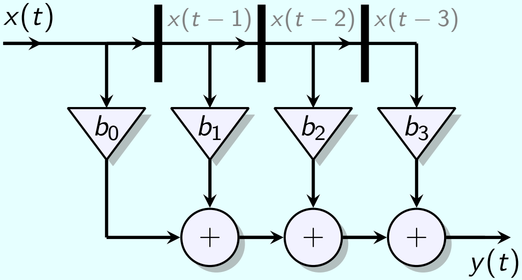
- the b_i are potentially real numbers (or almost: Matlab numbers)
- the $x(t)$ and $y(t)$ are discrete, fixed-point, low-precision signals

(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

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

$$
\begin{array}{rr}
b_0 = & .00001001111111010001010101101\ldots \\
b_1 = & .00101110110001000101001110000\ldots \\
b_2 = & .11000001011011010001001100101\ldots \\
b_3 = & .00110101000001001110111001111\ldots \\
\\
b_0x_0 & \mathtt{xxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_1x_1 & \mathtt{xxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_2x_2 & \mathtt{xxxxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
+b_3x_3 & \mathtt{xxxxxxxxxxxxxxxxxxxxxxxxxxx}\ldots \\
\\
y = & \mathtt{yyyyyyyyyyyyyyyyyyyyyyyyyyyyyy}\ldots
\end{array}
$$

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

FIR filters, arithmetic view

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

$b_0 =$ .000010011111110100010101
$b_1 =$ .001011101100010001010011
$b_2 =$ .110000010110110100010011
$b_3 =$ .001101010000010011101110

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

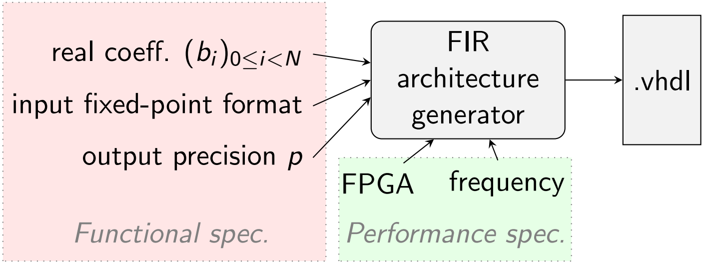
$$y(t) = \sum_{i=0}^{N-1} b_i x(t-i)$$

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

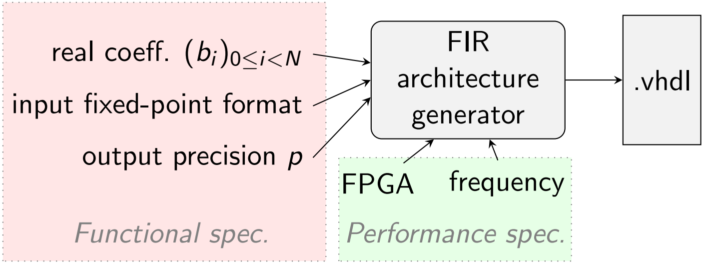
```
 b0x0        xxxxxxxxxxxxxxxxxxxx|xxx|
+b1x1      xxxxxxxxxxxxxxxxxxxxxx|xxx|
+b2x2    xxxxxxxxxxxxxxxxxxxxxxxx|xxx|
+b3x3      xxxxxxxxxxxxxxxxxxxxxx|xxx|
    =   zzzzzzzzzzzzzzzzzzzzzzzzz|zzz|
  y =   yyyyyyyyyyyyyyyyyyyyyyyyy|   |
                             2^-p  2^-p-g
```

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

Really a matter of interface

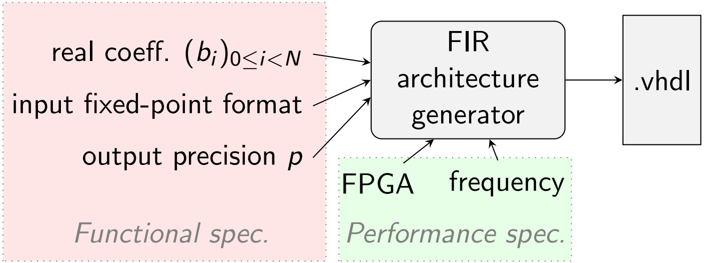


Really a matter of interface



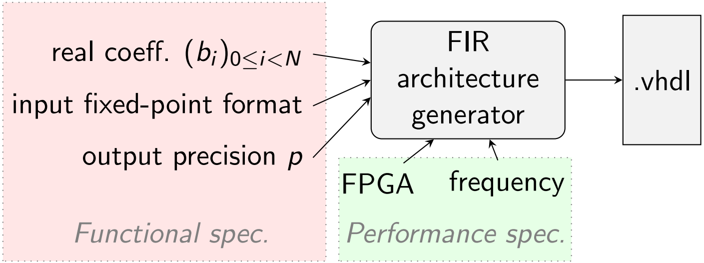
- Output precision defines accuracy of the architecture

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



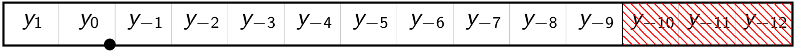
- Output precision defines accuracy of the architecture
- Accuracy defines the optimal precisions to be used internally

No point in computing more, no point in computing less

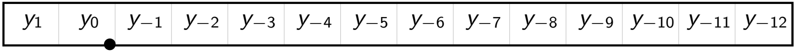
Example of the accuracy/cost tradeoff

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

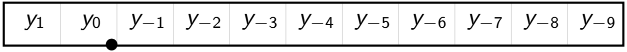
Naive, $p = 12$ 5.9 ns, 444 LUT $\overline{\epsilon} > 2^{-9}$



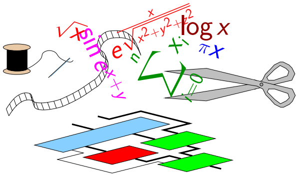
Proposed, $p = 12$ 4.4 ns, 564 LUT $\overline{\epsilon} < 2^{-12}$



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

$$
\begin{array}{rcl}
a_0 & = & .0000100111111101000101010101101\ldots \\
a_1 & = & .0010111011000100010100111000 0\ldots \\
a_2 & = & .1100000101101101000100110010 1\ldots \\
a_3 & = & .0011010100000100111011100111 1\ldots
\end{array}
$$

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

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

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

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

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

Compute Just Right: Determining the LSB

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

$a_0 x_0$ xxxxxxxxxxxxxxxxxxxxxxxxx ...
$+a_1 x_1$ xxxxxxxxxxxxxxxxxxxxxxxxxxx ...
$+a_2 x_2$ xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ...
$+a_3 x_3$ xxxxxxxxxxxxxxxxxxxxxxxxxxx ...

$y =$ yyyyyyyyyyyyyyyyyyyyyyyyy

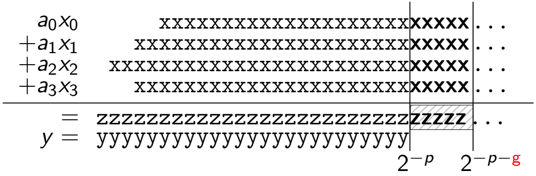
2^{-p}

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: $\varepsilon_{mult} < 2^{-p-1}$

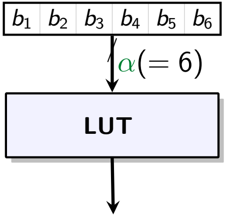
- sum error: $\varepsilon_{y_{total}} = \sum_{i=0}^{N} \varepsilon_{mult} + \varepsilon_{final\_rounding} < N \cdot 2^{-p-g-1} + 2^{-p-1}$

Need for larger intermediary precision

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

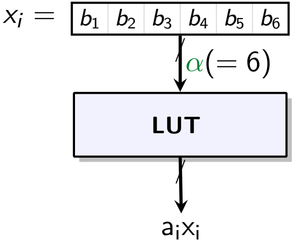
$$\Longrightarrow g = \lceil \log_2(N) \rceil$$

Perfect constant multipliers in an FPGA



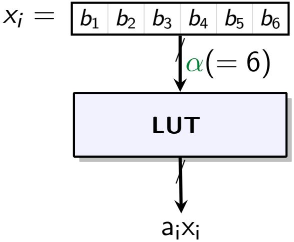
- basic FPGA computing element: look-up table (**LUT**)

Perfect constant multipliers in an FPGA



- basic FPGA computing element: look-up table (**LUT**)
- tabulate all the 2^α values of $a_i x_i$
- … 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 $a_i x_i$
- ... 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}						d_{i2}						d_{i3}			

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

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

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}			

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

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

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}			

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

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

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}			

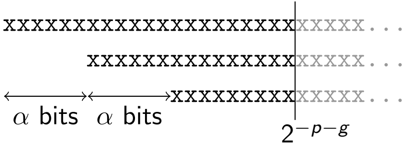
$$x_i = \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\}$$

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

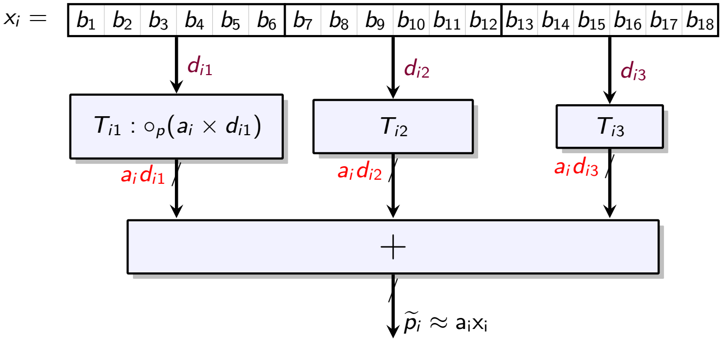
Each $a_i d_{ik}$ tabulated, 1 LUT/output bit

How many output bits?

$$\begin{aligned} \mathrm{a_i x_i} &= a_i d_{i1} \\ &+ 2^{-\alpha} a_i d_{i2} \\ &+ 2^{-2\alpha} a_i d_{i3} \end{aligned}$$



KCM multipliers by real constants



Summing it all up

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

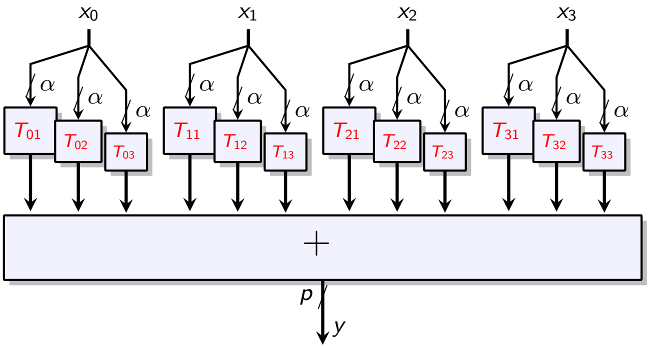
Summing it all up

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

Summing it all up

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

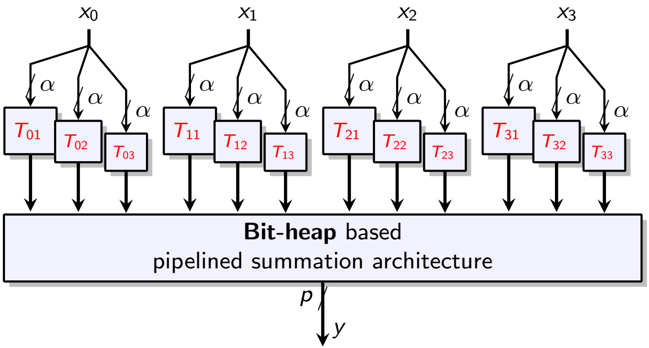
- each $a_i d_{ik}$ is a perfect multiplier
- therefore $g = \lceil \log_2(N \cdot n) \rceil$



Summing it all up

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

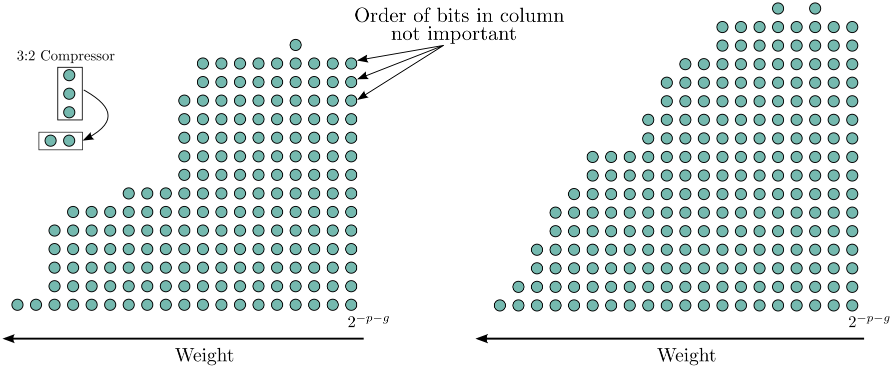
- each $a_i d_{ik}$ is a perfect multiplier
- therefore $g = \lceil \log_2(N \cdot n) \rceil$



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

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...

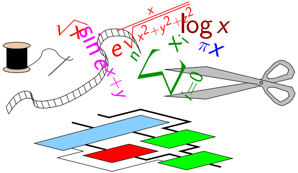
- (the easy half)

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 have contributed to FloPoCo:
S. Banescu, L. Besème, N. Bonfante,
M. Christ, N. Brunie, S. Collange, 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.



http://flopoco.org/

Example: floating-point exponential

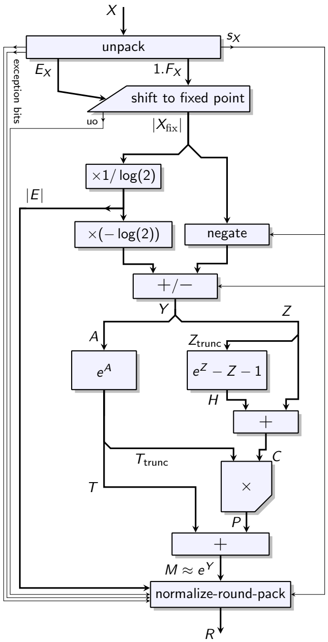
First, a math proficiency test

Three identities to remember from our happy school days

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

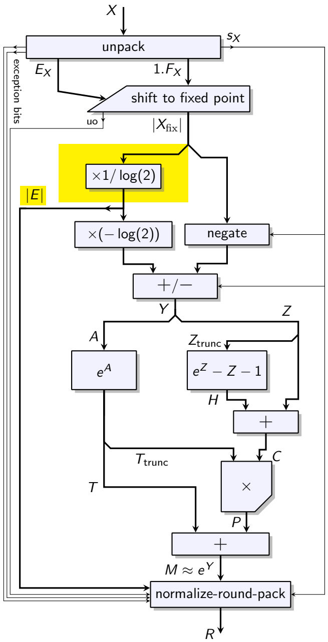
$$e^{A+B} = e^A \times e^B \quad (2)$$

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



We want to obtain e^X as

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

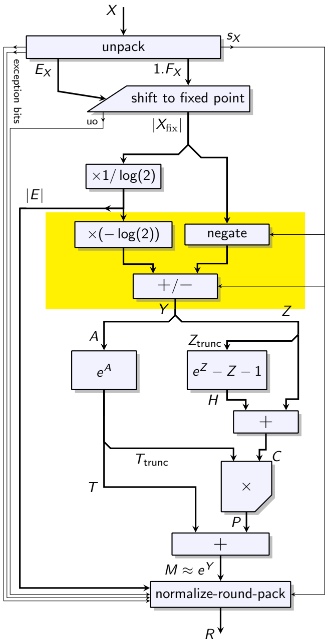


We want to obtain e^X as

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

Compute

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



We want to obtain e^X as

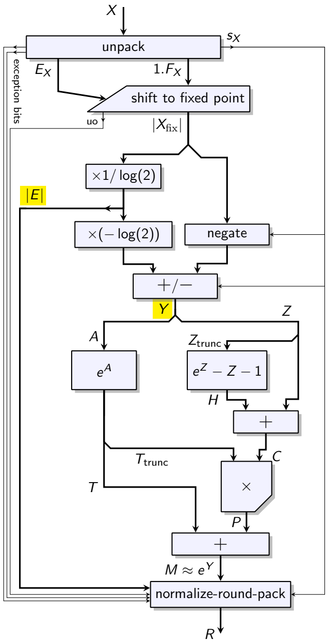
$$e^X = 2^E \cdot 1.F$$

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

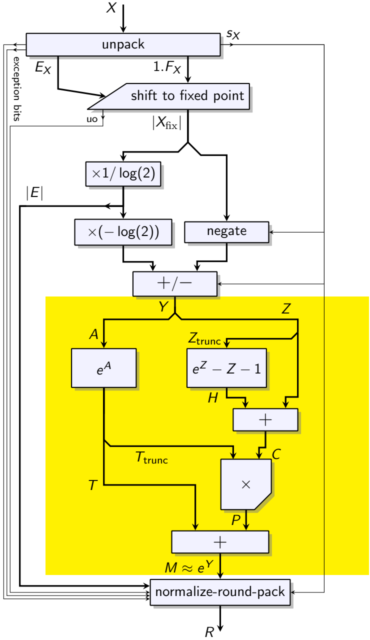
$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$

then

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

Now

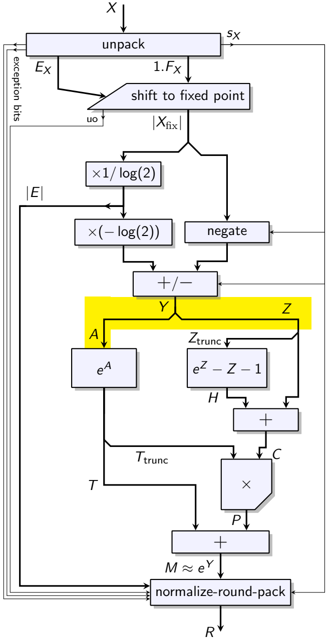
$$\begin{aligned} e^X &= e^{E \log 2 + Y} \\ &= e^{E \log 2} \cdot e^Y \\ &= 2^E \cdot e^Y \end{aligned}$$



We want to obtain e^X as

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

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

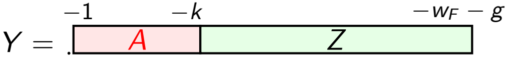


We want to obtain e^X as

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

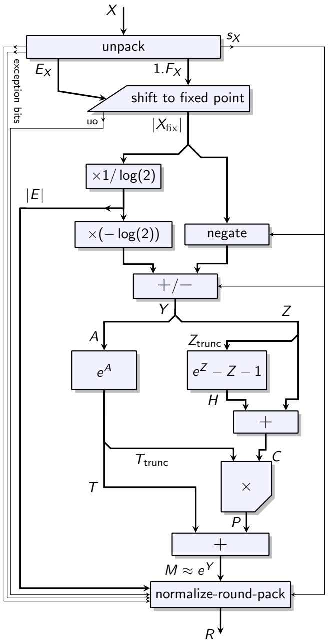
Now we have to compute e^Y with $Y \in (-1/2, 1/2)$.

Split Y:



i.e. write

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

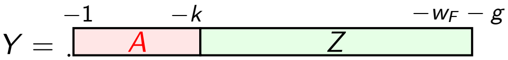


We want to obtain e^X as

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

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

Split Y:

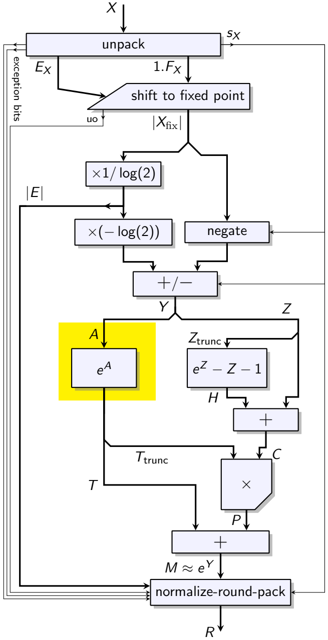


i.e. write

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

so

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

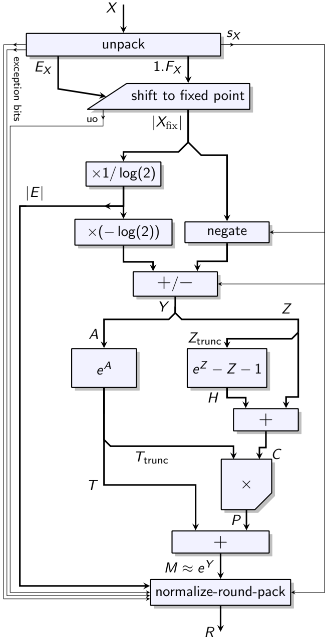


We want to obtain e^X as

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

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

Tabulate e^A in a ROM



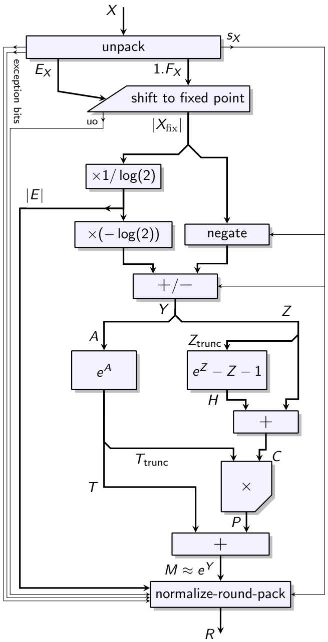
We want to obtain e^X as

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

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

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

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



We want to obtain e^X as

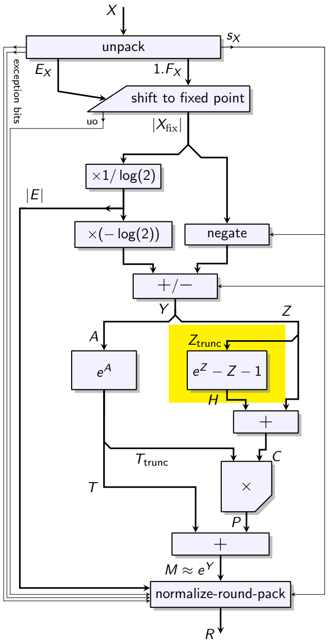
$$e^X = 2^E \cdot e^Y$$

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

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

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

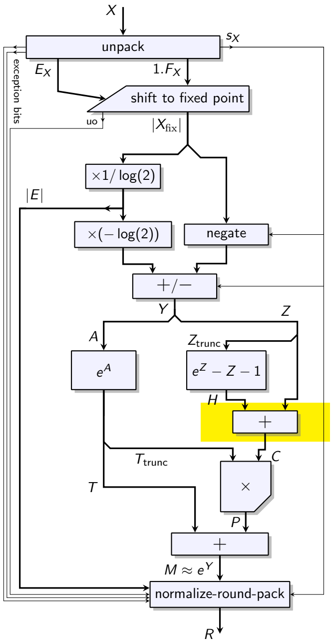
$$e^Y = e^A \times e^Z$$

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

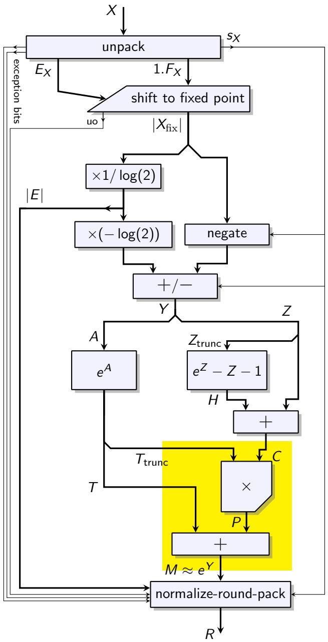
$$e^Z \approx 1 + Z + Z^2/2$$

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

Evaluate $e^Z - Z - 1$ somehow

(out of Z truncated to its higher bits only)

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



We want to obtain e^X as

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

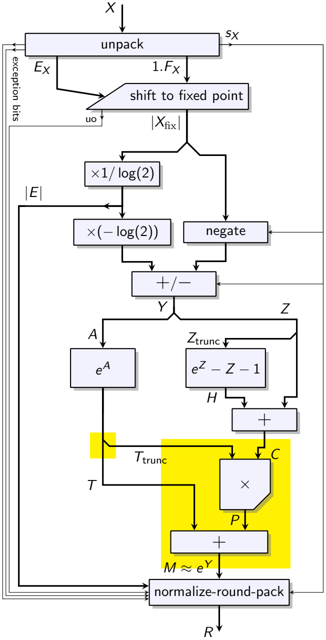
$$e^Y = e^A \times e^Z$$

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

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

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

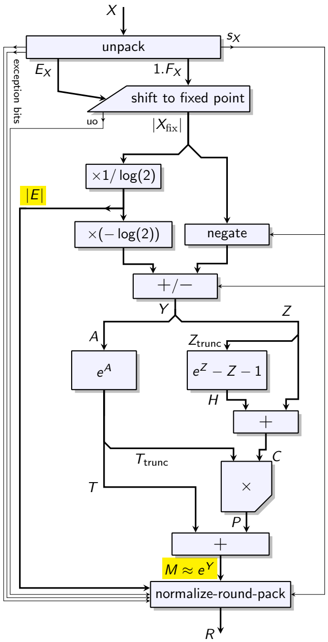
Also notice that

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

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

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

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

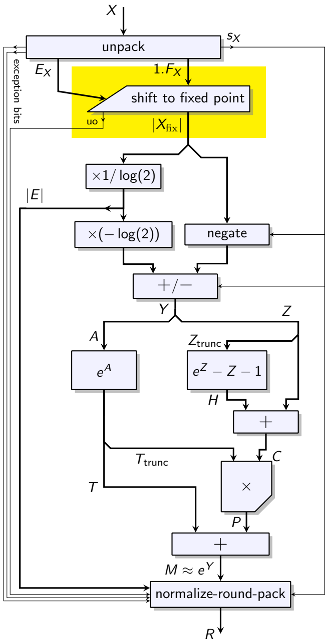


We want to obtain e^X as

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

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

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

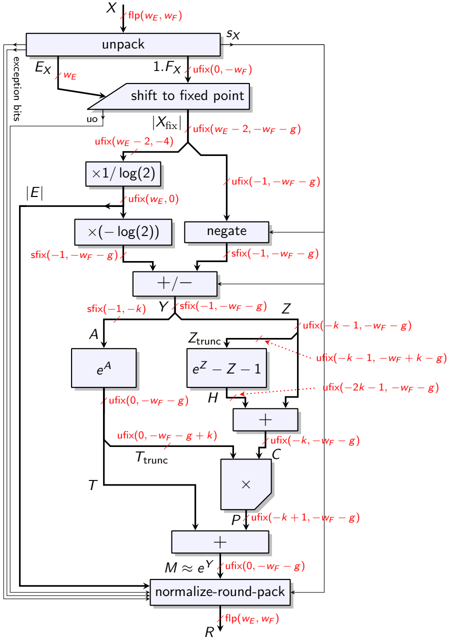


We want to obtain e^X as

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

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

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



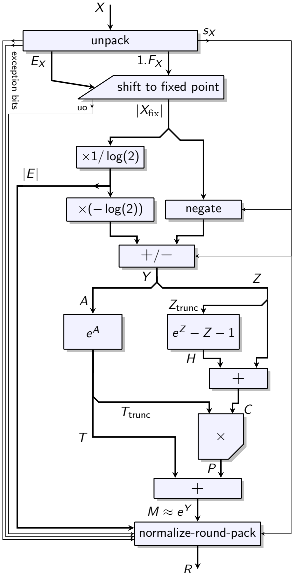
We want to obtain e^X as

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

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

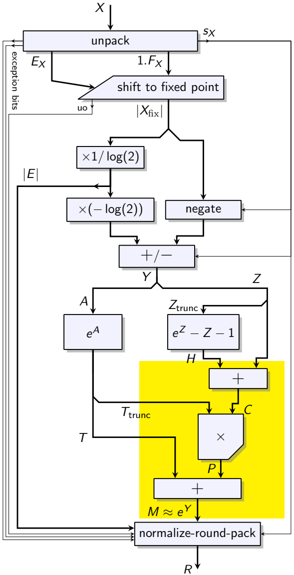
And that's it, we have E and e^Y
(using only *fixed-point* computations)

Single-precision magic



Modern FPGAs also have

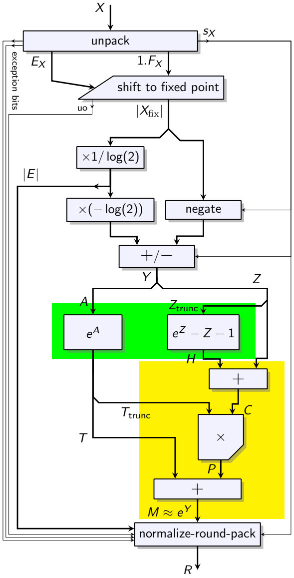
Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders

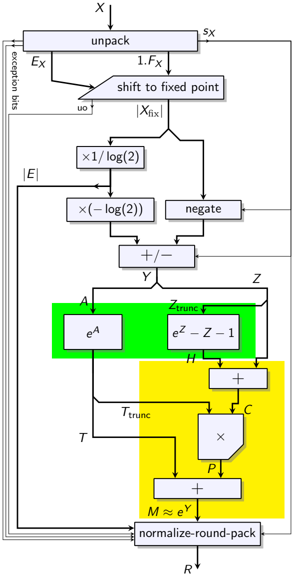
Single-precision magic



Modern FPGAs also have

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

Single-precision magic



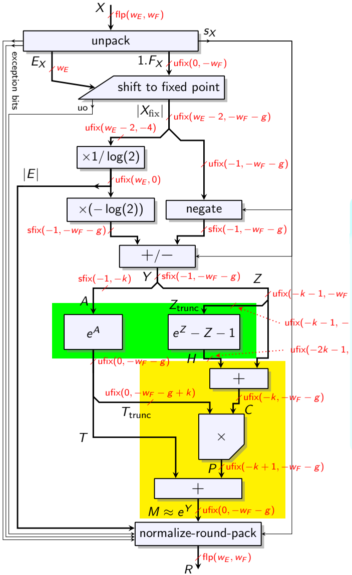
Modern FPGAs also have

- 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)



Modern FPGAs also have

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

Single-precision accurate exponential on Xilinx

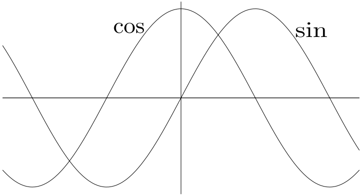
- one block RAM (0.1% of the chip)
- one DSP block (0.1%)
- $<$ 400 LUTs (0.1%, $\approx$ one FP adder)

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)

Example: fixed-point sine/cosine

Introduction



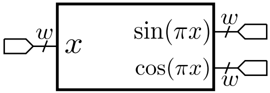
- Sine and cosine functions
  - fundamental in signal processing and signal processing applications like FFT, modulation/demodulation, frequency synthesizers, ...
- **How** to compute them ? In this work:
  1. the classical CORDIC algorithm, based on additions and shifts
  2. a method based on tables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

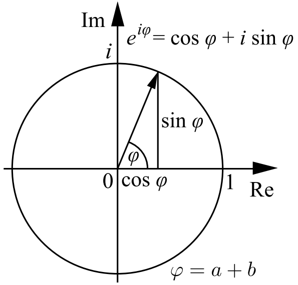
$$e^{i(a+b)} = e^{ia} \times \ e^{ib}$$

- From complex to real

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

- Decompose a rotation in smaller sub-rotations

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

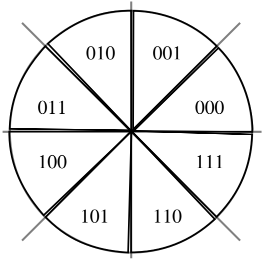


Argument Reduction

- 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:



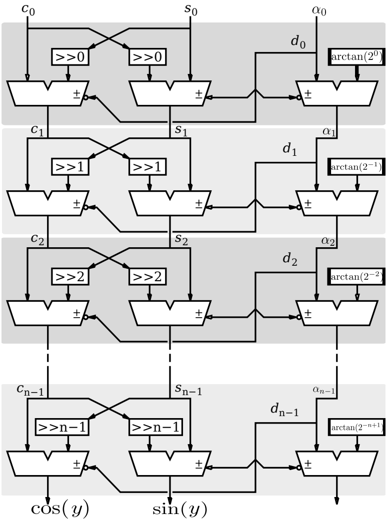
sqo	Reconstruction
000	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = \cos(\pi y') \end{cases}$
001	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = \sin(\pi y') \end{cases}$
010	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = -\sin(\pi y') \end{cases}$
011	$\begin{cases}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

CORDIC Architecture

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

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

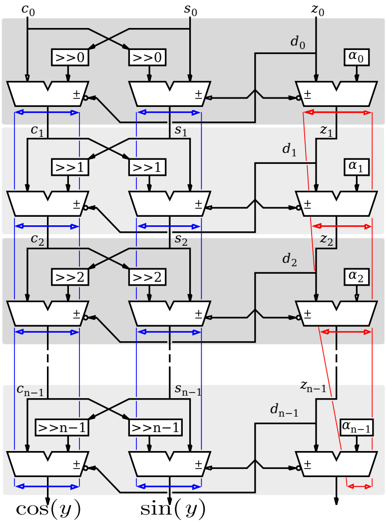
$$\begin{cases} c_{n\to\inf} = \cos(y) \\ s_{n\to\inf} = \sin(y) \\ \alpha_{n\to\inf} = 0 \end{cases}$$



CORDIC Improvements

Reduced α-Datapath

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



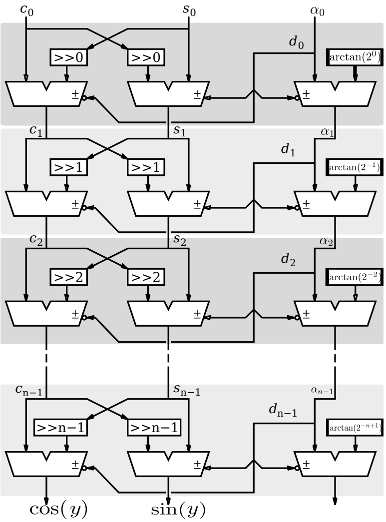
Reduced Iterations

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

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

- half the iterations replaced by

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



CORDIC Improvements

Reduced Iterations

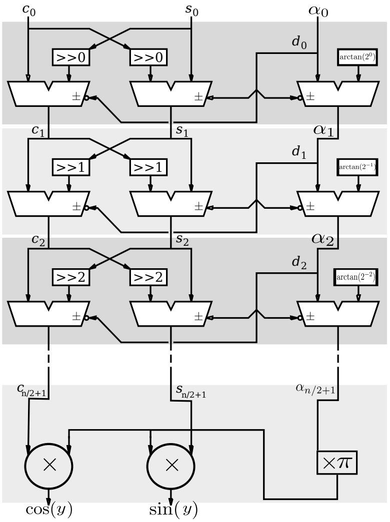
- stop iterations when they can be replaced by a single rotation, with enough accuracy

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

- half the iterations replaced by

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

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



CORDIC Error Analysis

Goal: last-bit accuracy of the result

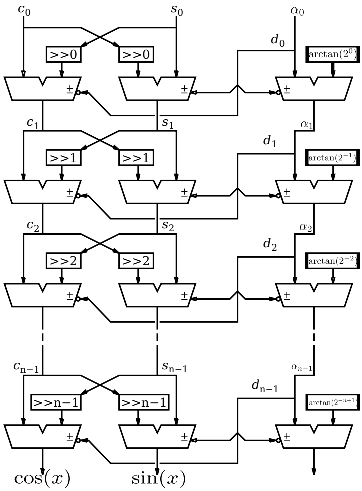
- the result is within 1**ulp** of the mathematical result
- **ulp** = weight of least significant bit

Intermediate precision

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

Error budget: total of 1**ulp**

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

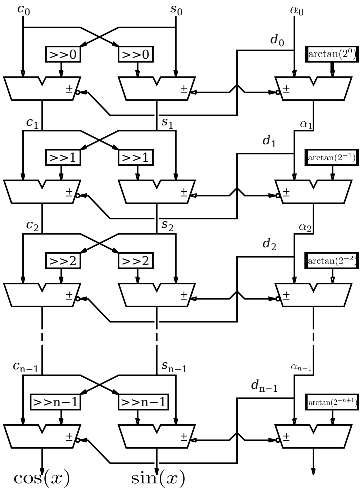


CORDIC Error Analysis (1)

Analysis: method error (ε_{method})

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

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



CORDIC Error Analysis (2)

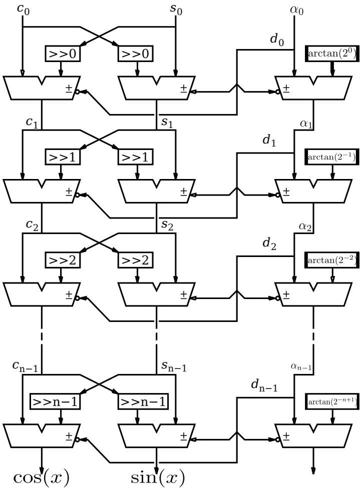
Analysis: rounding errors (ε_{round})

on the α datapath

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

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

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



CORDIC Error Analysis (2)

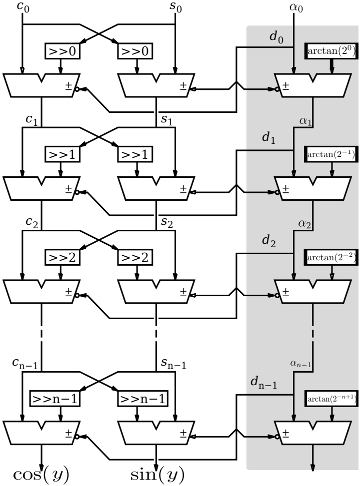
Analysis: rounding errors (ε_{round})

on the α datapath

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

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

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



CORDIC Error Analysis (2)

Analysis: rounding errors (ε_{round})

on the α datapath

- correct rounding of $arctan(2^{-i})$

  error bounded by 2^{-w-g-1}
- total error on the α-datapath:

  $nb\_iter \times 2^{-w-g-1}$

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

- for each shift operation, error bounded by 2^{-w-g}
- total error larger than on the α-datapath
- must be smaller than 2^{-w-2}:

  $\varepsilon \times 2^{-w-g} < 2^{-w-2}$
- this gives g
- $\varepsilon_{method} + \varepsilon_{round} < 2^{-w-1}$

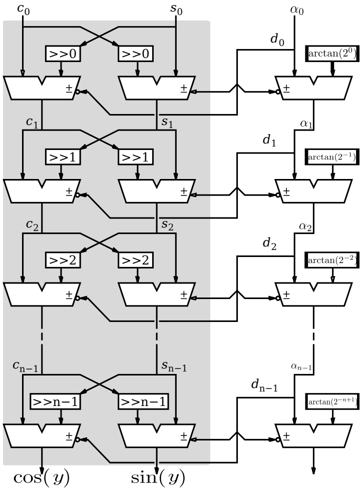


Table- and DSP-based method

Algorithm

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

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

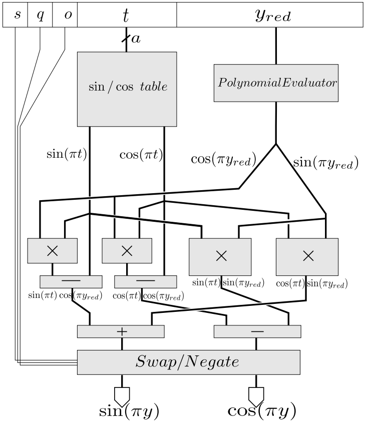


Table- and DSP-based method

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

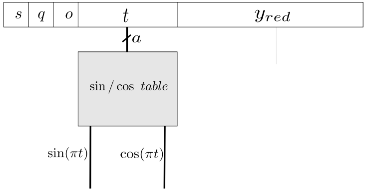
$$\begin{cases} \sin(\pi(t + y_{red})) = \sin(\pi t)\cos(\pi y_{red}) + \cos(\pi t)\sin(\pi y_{red}) \\ \cos(\pi(t + y_{red})) = \cos(\pi t)\cos(\pi y_{red}) - \sin(\pi t)\sin(\pi y_{red}) \end{cases}$$

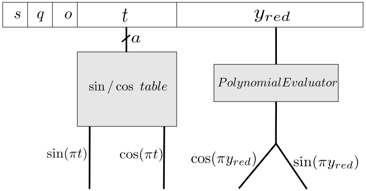
Table- and DSP-based method

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

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





Algorithm

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

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

Table- and DSP-based method

Algorithm

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

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

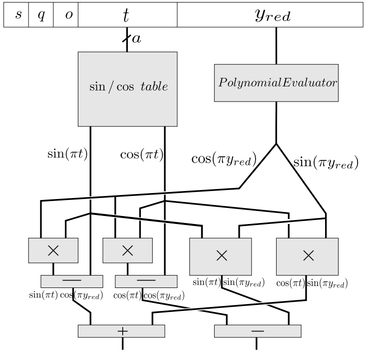


Table- and DSP-based method

Algorithm

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

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

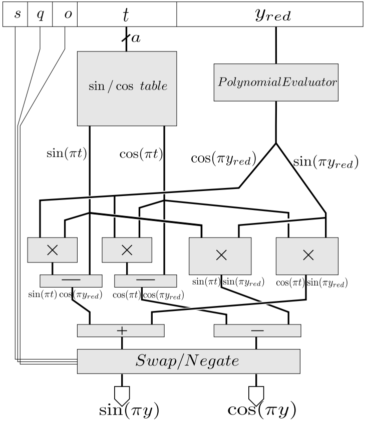


Table- and DSP-based method: Details

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

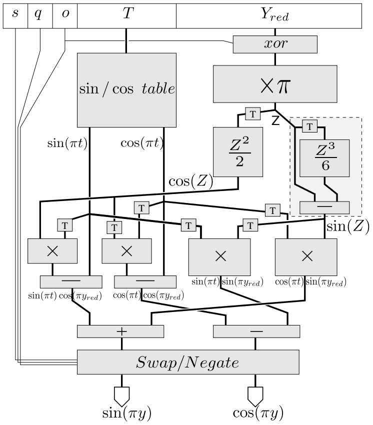
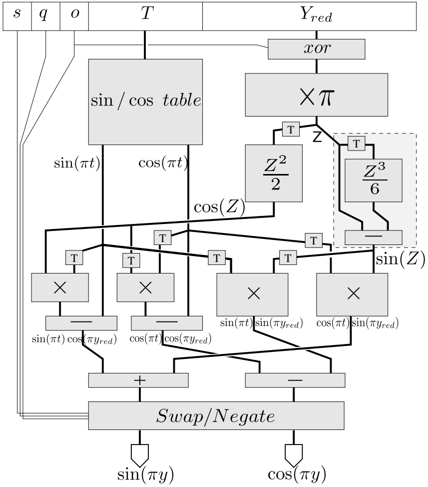


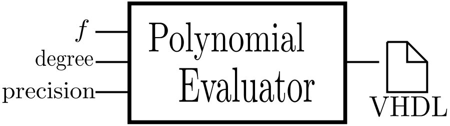
Table- and DSP-based method: Error Analysis

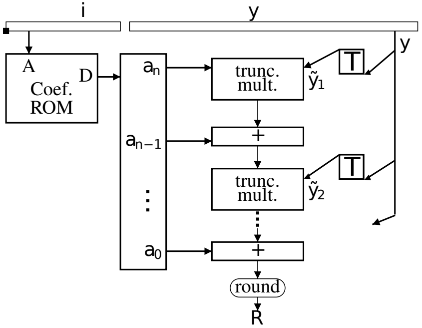
Error Analysis

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



Polynomial-based method





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

Results – 16−bit Precision

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

Synthesis Results on Virtex5 FPGA, Using ISE 12.1

Results – Highest Frequency

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

Results – Options for $\frac{Z^3}{6}$

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

Conclusions

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

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

Synthesis results for logic-only implementations

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

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:

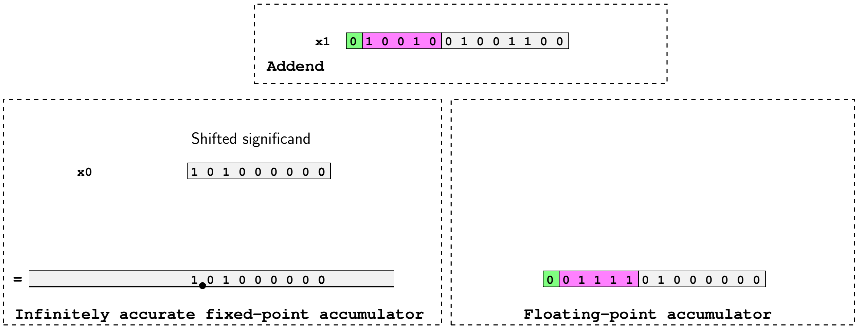
$$x = (-1)^S \times 1.f \times 2^e$$

where:

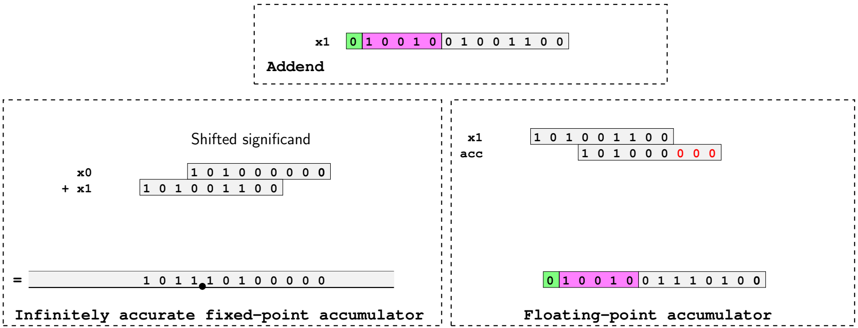
- S - the **sign** of x
- f - the **fraction** of x.
- e - the **exponent** of x

Graphical representation:

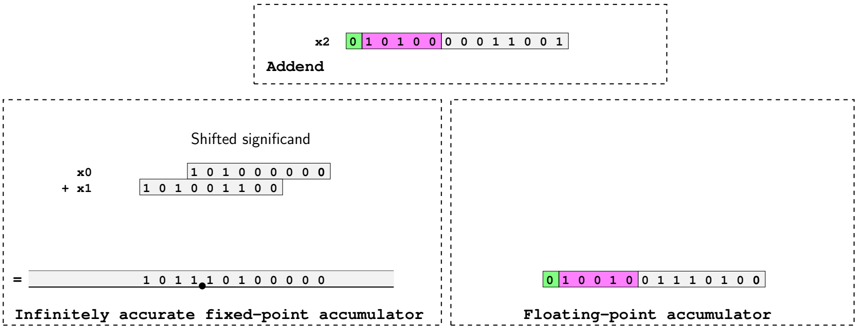
Accumulation



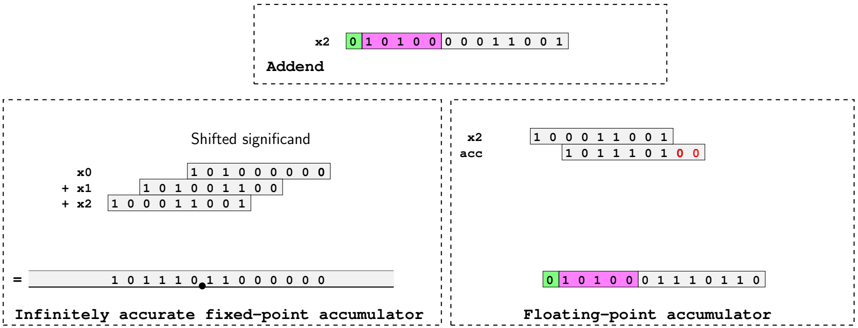
Accumulation



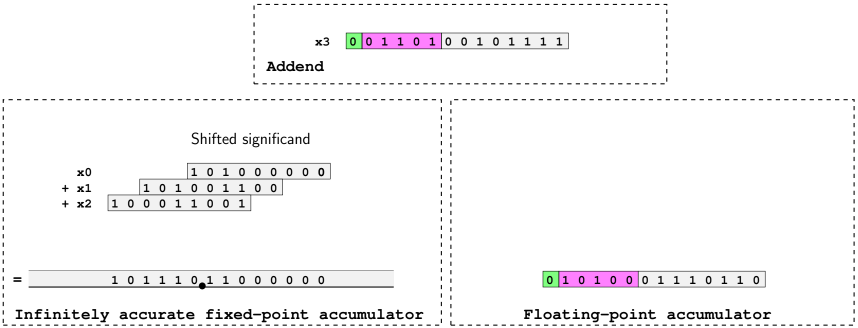
Accumulation



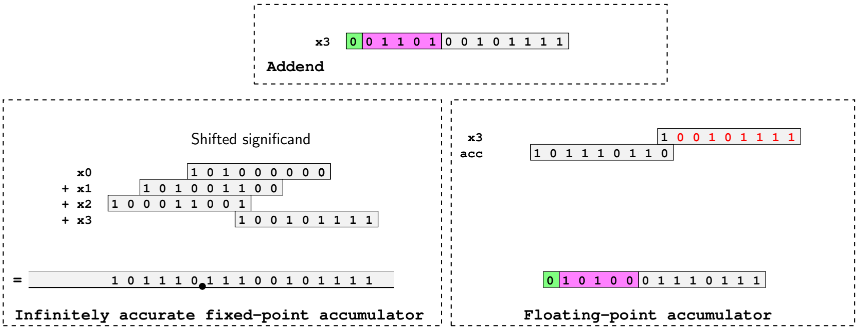
Accumulation



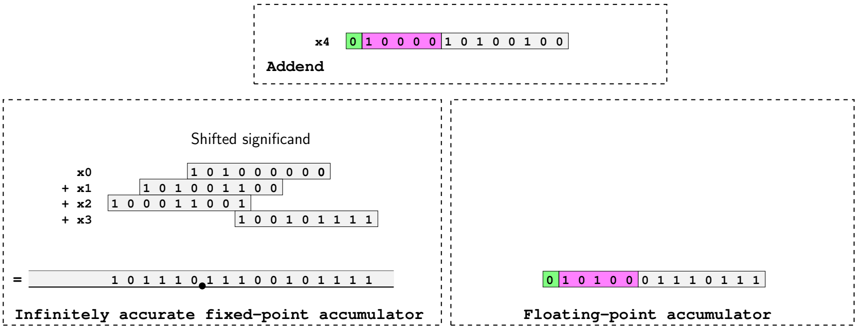
Accumulation



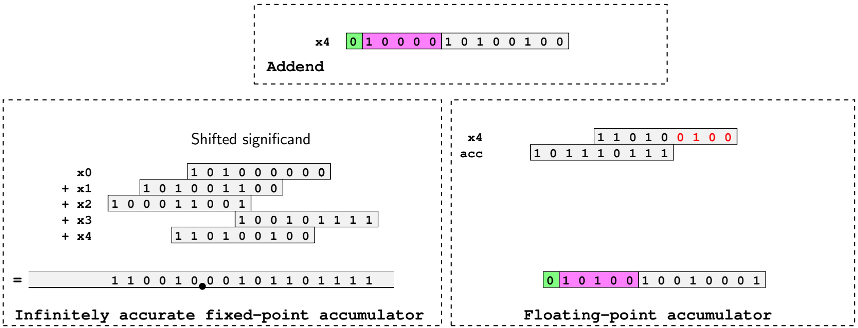
Accumulation



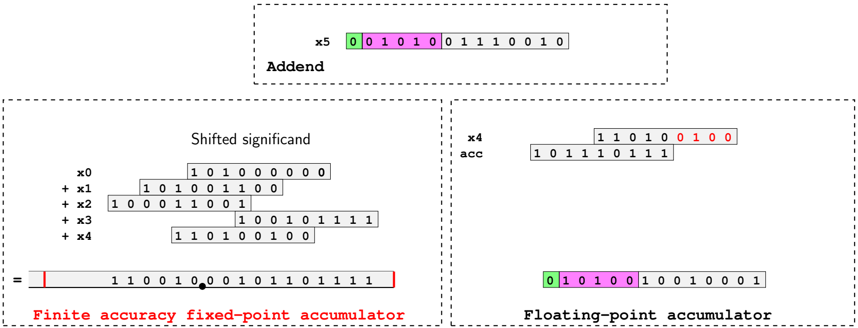
Accumulation



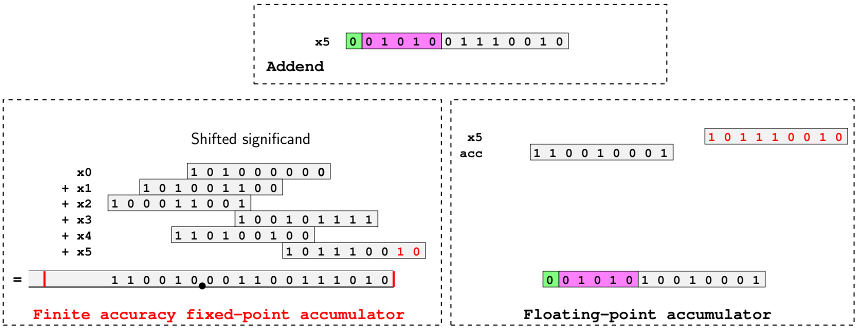
Accumulation



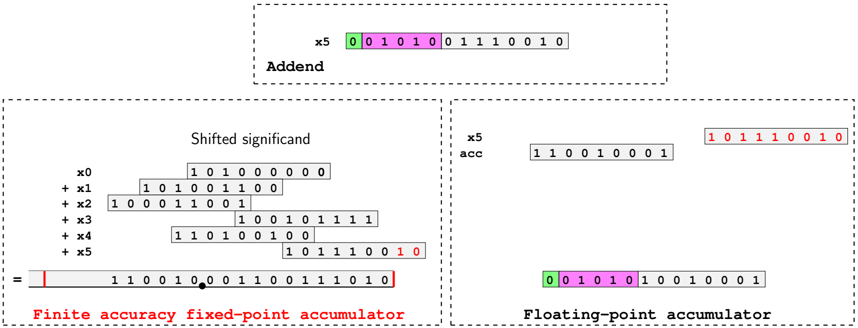
Accumulation



Accumulation



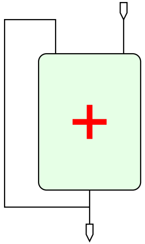
Accumulation



Accuracy:

Exact Result	=	50.2017822265625
FP Acc	=	50.125
Fixed-Point Acc	=	50.20166015625

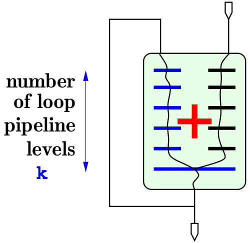
Closer look



Accumulator based on combinatorial floating-point adder

- very low frequency
- must pipeline for larger frequency

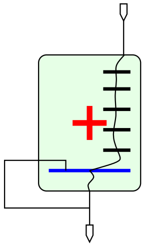
Closer look



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

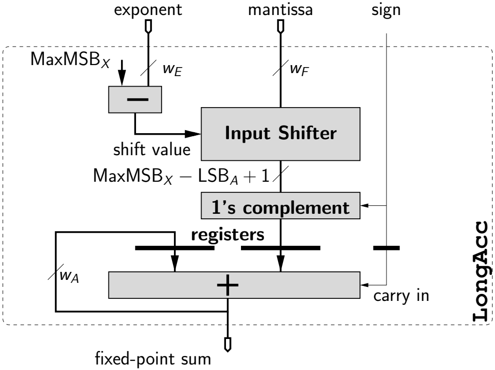
Closer look



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

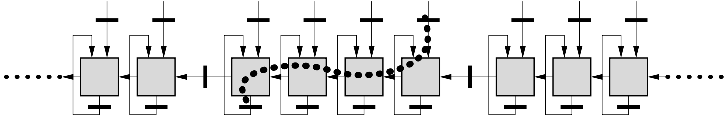
The accumulator should run at a target frequency

- 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

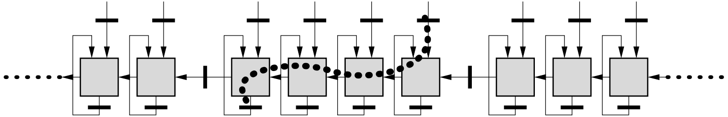
- 64-bit addition works at 220MHz on Xilinx Virtex4 FPGA due to fast-carry chains
- still not enough ?
- use *partial carry-save representation*
  - cut large carry-propagation into chunks of k bits
  - critical path $=$ k-bit addition
  - small cost: $\lfloor width_{accumulator}/k \rfloor$ registers



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

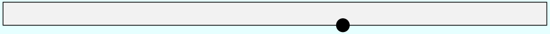
We advocate:

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

Ensuring that:

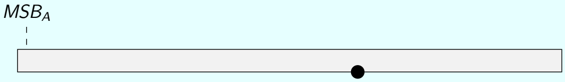
1. accumulator significand never needs to be shifted
2. it never overflows
3. provides a **result as accurate as the application requires**

Accumulator Parameters



The designer must provide values for these parameters.

Accumulator Parameters

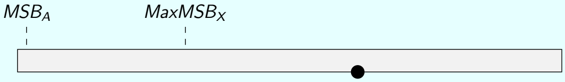


MSB_A the weight of the MSB of the accumulator

- must to be larger than max. expected result

The designer must provide values for these parameters.

Accumulator Parameters



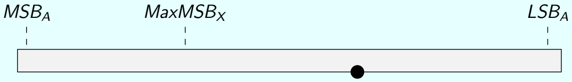
MSB_A the weight of the MSB of the accumulator

- 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:

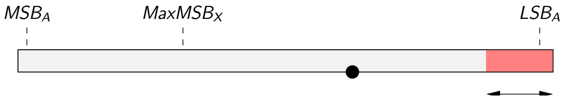
- **software profiling** + safety margins
- **rough error analysis** + safety margins

How to chose the parameters using the rough error analysis ?

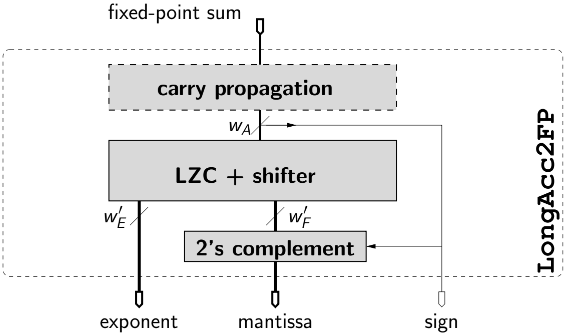
MSB_A
- know an actual maximum + 10 bits safety margin
- consider the number of terms to sum

$MaxMSB_X$
- 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

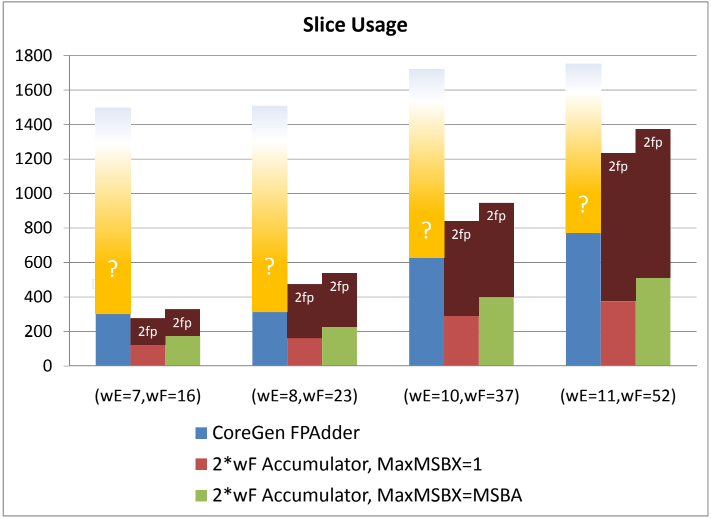


Post-normalization unit, or not

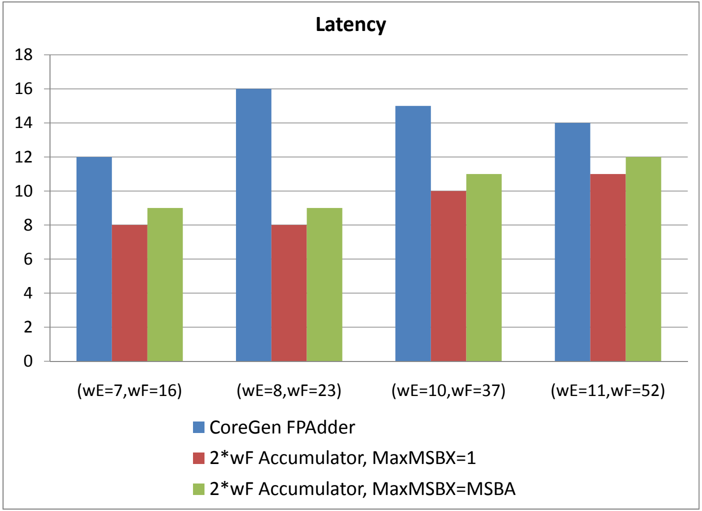


- converts fixed-point accumulator format to floating-point
- pipelined unit may be shared by several accumulators
- less useful:
  - many applications do not need the running sum
  - better to do conversion in software, use FPGA to accelerate the computation

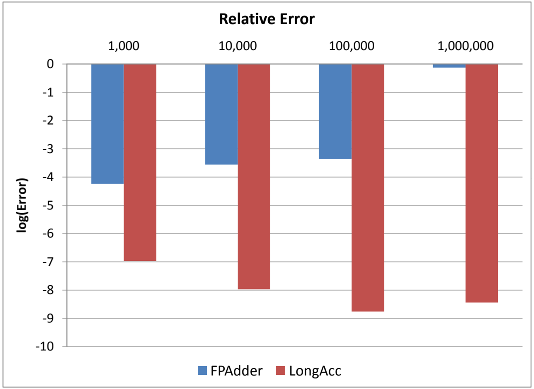
Performance results



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

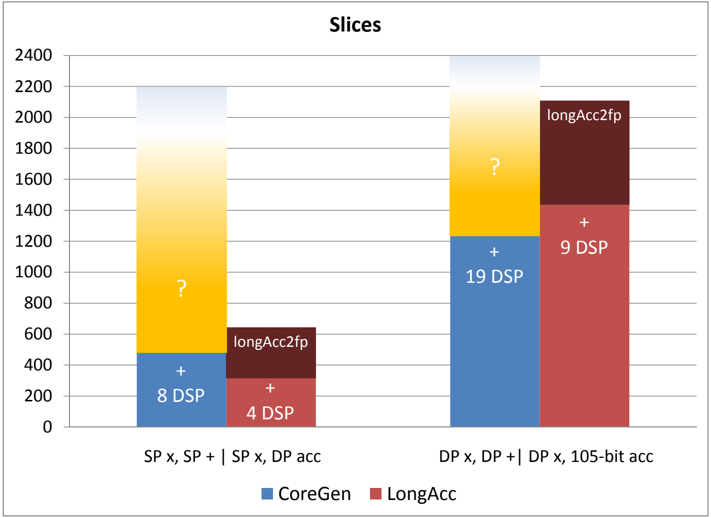
Ideea

Accumulate **exact** results of all multiplications

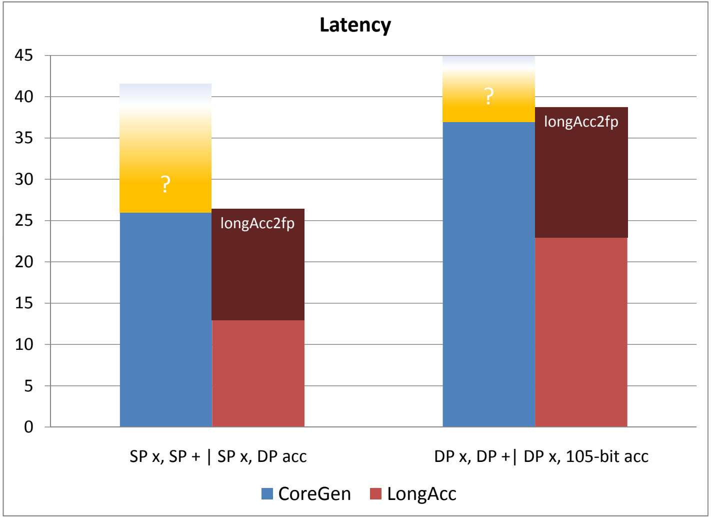
1. Use exact multipliers:
   - return all the bits of the exact product
   - contain no rounding logic
   - are cheaper to build
2. Feed the accumulator with exact multiplication results

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

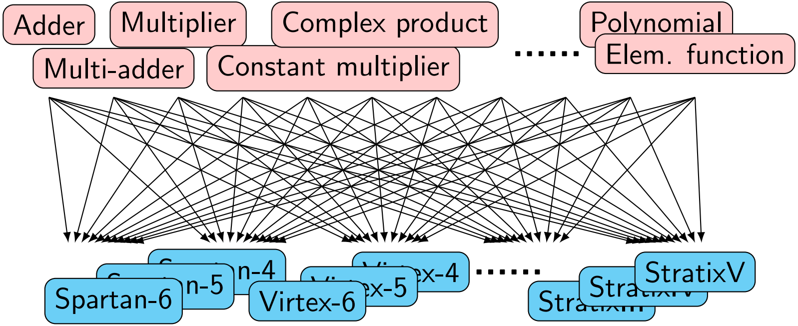
FPGA arithmetic the way it should be:

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

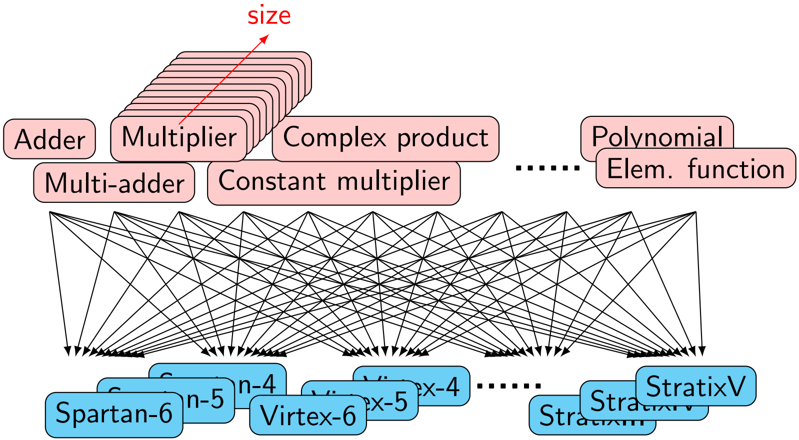
How to ensure **performance** across all this range?

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

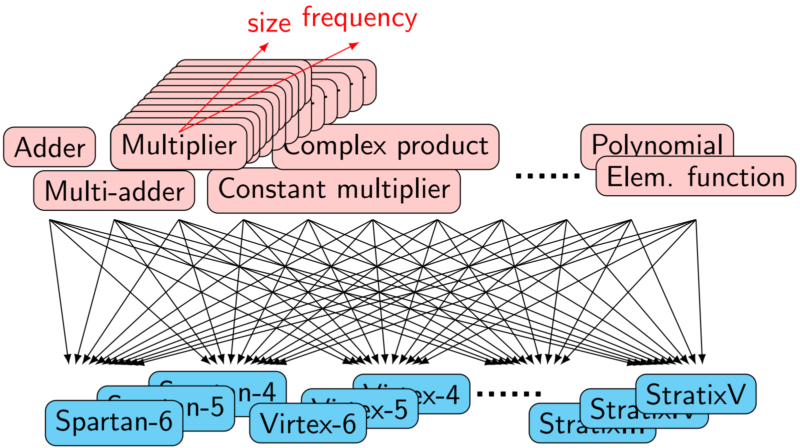
Portable versus optimized



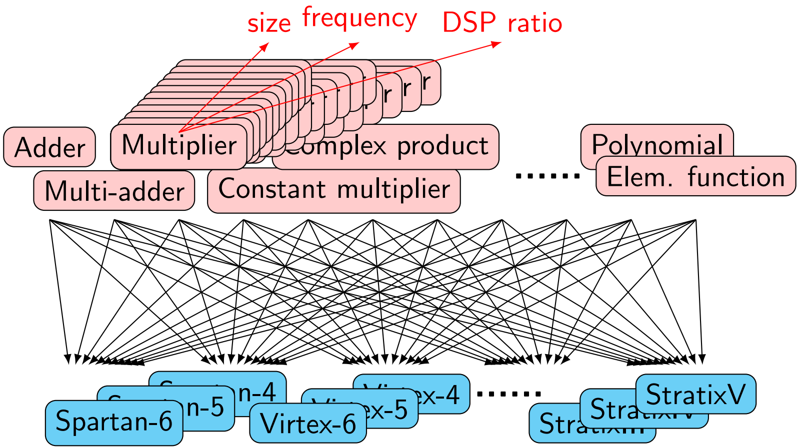
Portable versus optimized



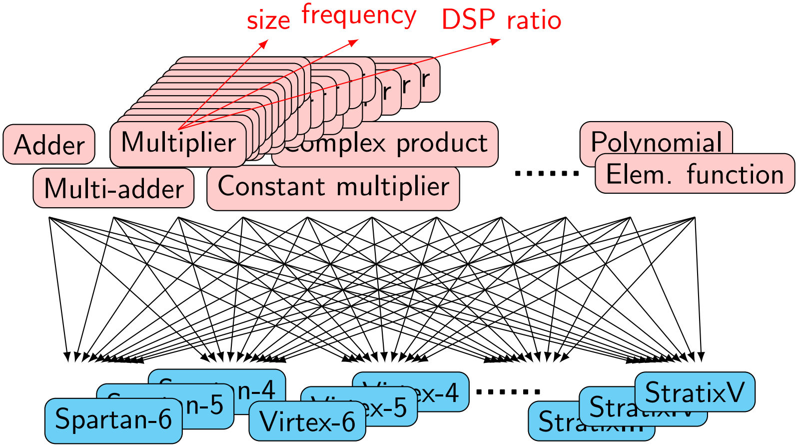
Portable versus optimized



Portable versus optimized

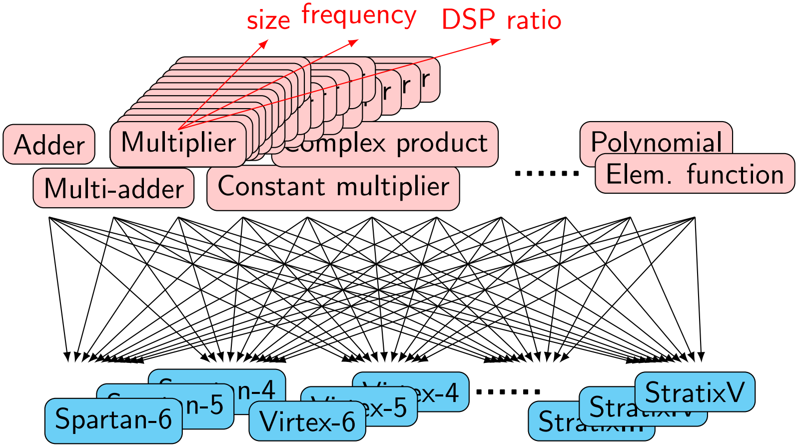


Portable versus optimized



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

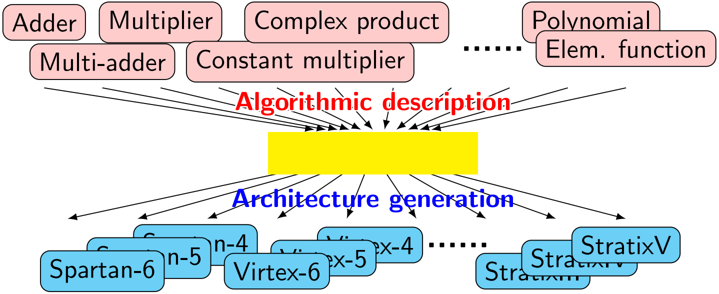
Portable versus optimized



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

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

Reducing the combinatorics

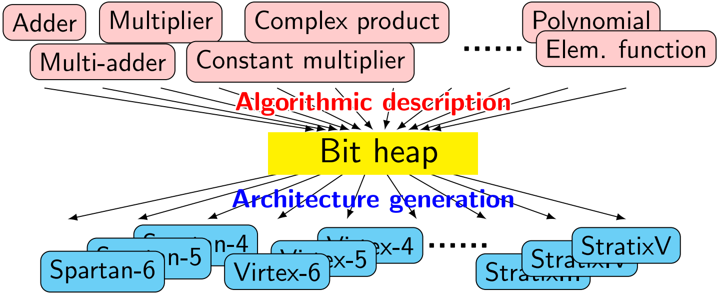


What is a bit heap?

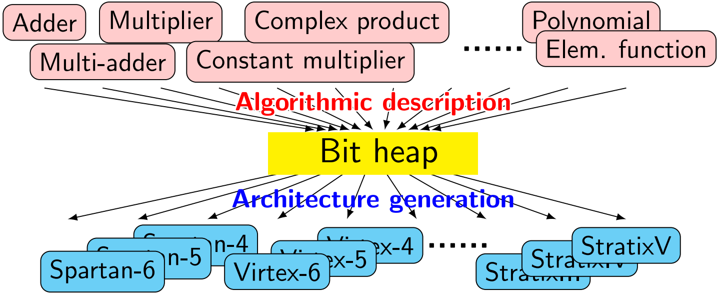
- A **data-structure**
  - capturing bit-level descriptions of a wide class of operators
  - exposing bit-level parallelism and optimization opportunities
- An associated **architecture generator**

which can be optimized for each target

Reducing the combinatorics



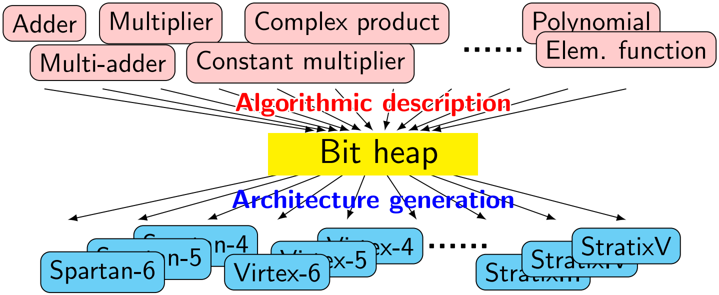
Reducing the combinatorics



What is a bit heap?

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

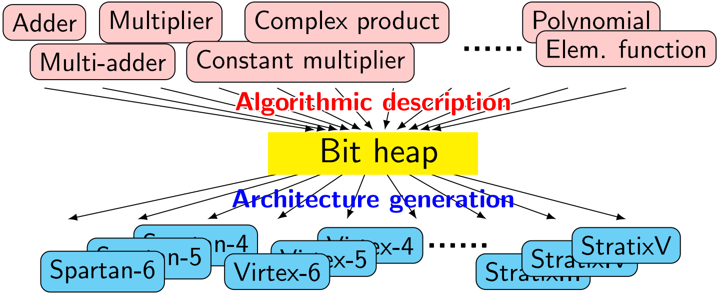
Reducing the combinatorics



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

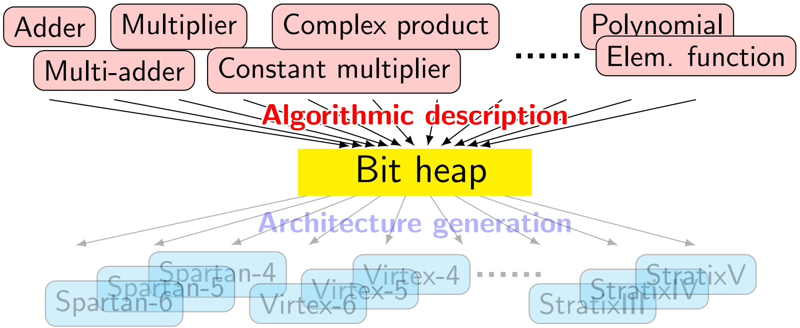


What is a bit heap?

- 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



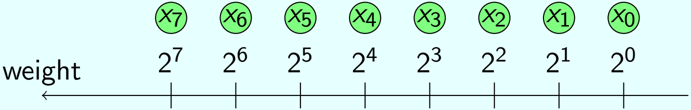
Weighted bits

- Integers or real numbers represented in binary fixed-point

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

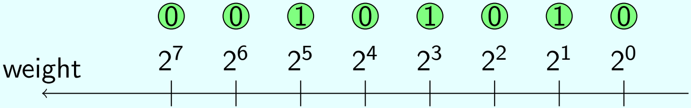
- 2^i : "weight" $\Longrightarrow$ X "sum of weighted bits"

Representation as a **dot diagrams**

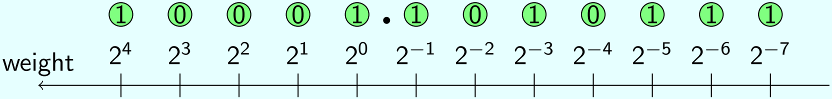


Integer or fixed-point

Example: 42 written in binary



Example: 17.42 written in binary



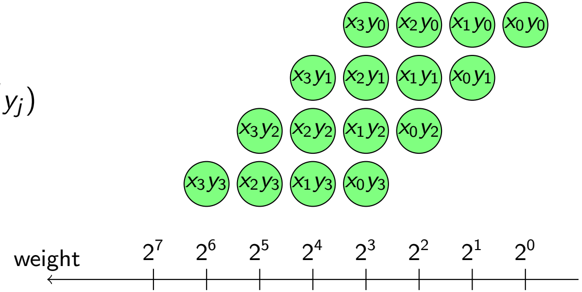
The historical bit heap

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

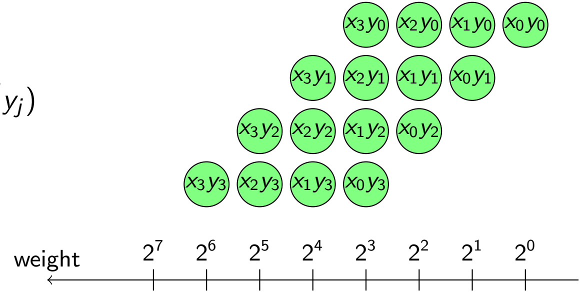
The historical bit heap

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

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



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



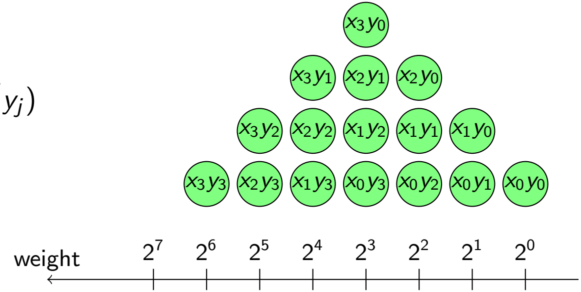
A multiplier is an architecture that computes this sum.

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.

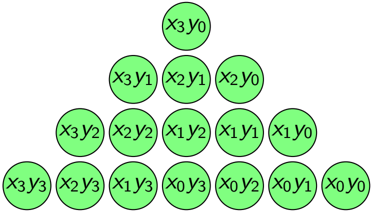
Historical motivation for bit heaps

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

(freedom thanks to addition associativity and commutativity)

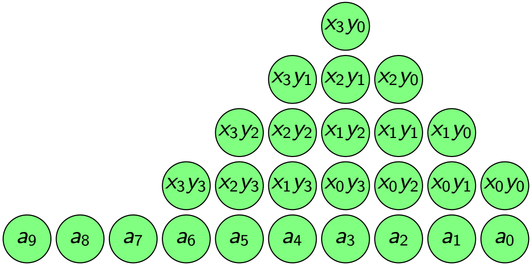
Beyond product

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



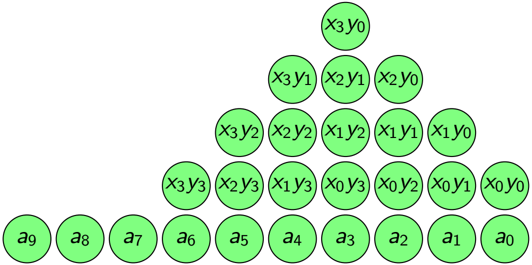
Beyond product

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



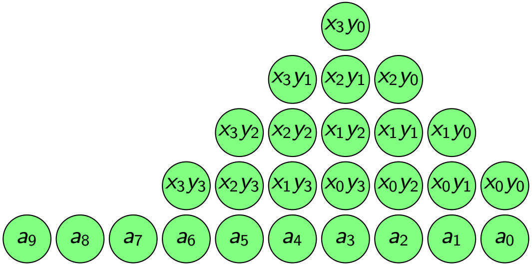
Beyond product

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



Beyond product

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



When generating an architecture

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

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

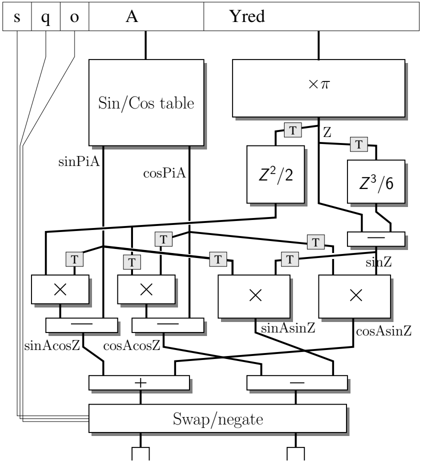
… where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

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

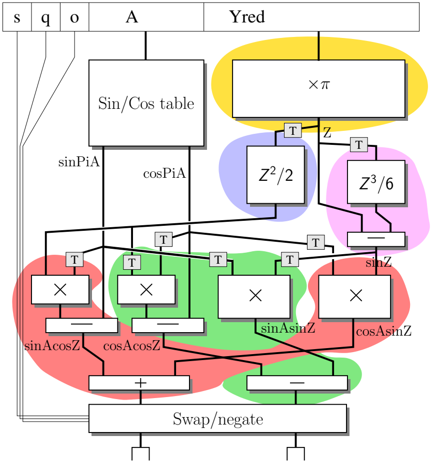
When you have a good hammer, you see nails everywhere

A sine/cosine architecture (HEART 2013)



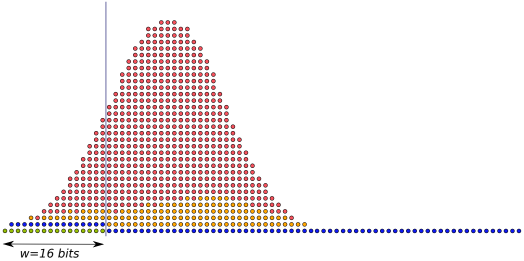
When you have a good hammer, you see nails everywhere

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

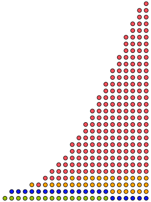


A bit heap for $Z - Z^3/6$ in the previous architecture

Full bit heap



Bit heap truncated just right



The constant vector

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

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

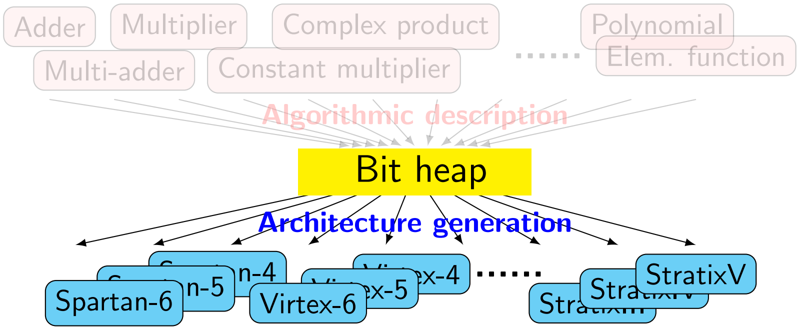
To replicate bit s from weight p to weight q

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

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

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

Generating an architecture



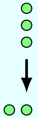
Architecture computing the value of a bit heap

Elementary case 1: the compressor

A compressor replaces a column of bits

by its sum written in binary (on fewer bits)

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

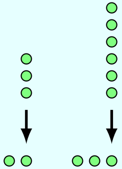


Architecture computing the value of a bit heap

Elementary case 1: the compressor

A compressor replaces a column of bits

by its sum written in binary (on fewer bits)



- archetype: the *full adder* is a *3 to 2* compressor
- on a recent FPGA: a *6 to 3* compressor

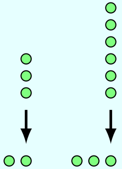
  tabulated in 3 6-input LUTs.
- survey and refs in the FPL 2013 paper, see also papers by M. Kumm.

Architecture computing the value of a bit heap

Elementary case 1: the compressor

A compressor replaces a column of bits

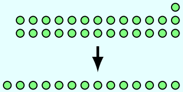
by its sum written in binary (on fewer bits)



- archetype: the *full adder* is a *3 to 2* compressor
- on a recent FPGA: a *6 to 3* compressor

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

Elementary case 2: the adder



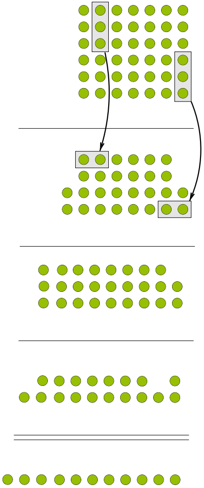
An adder replaces two *n*-bit lines, and a carry

by a line of $n+1$ bits

Architecture computing the value of a bit heap

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

   Stop when bit heap height equal to two

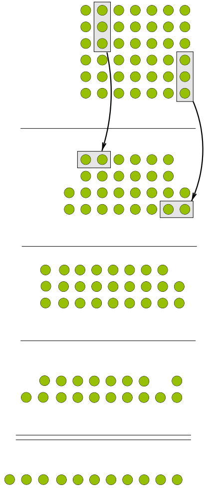


Architecture computing the value of a bit heap

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

   Stop when bit heap height equal to two

2. Final fast addition
   - add the remaining two lines

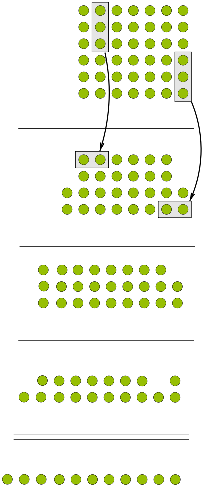


Architecture computing the value of a bit heap

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

   Stop when bit heap height equal to two

2. Final fast addition
   - add the remaining two lines



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

Bit heaps and DSP blocks

Elementary case: the DSP block?

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

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

Really different architectures here

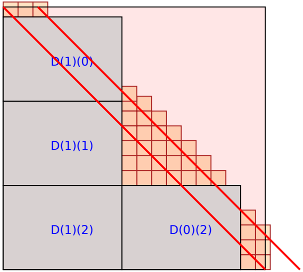
Elementary case: the DSP block?

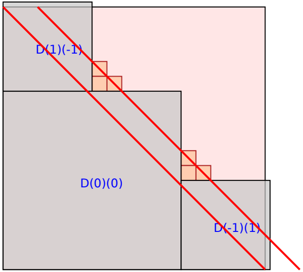
- Xilinx DSP blocks compute $A + XY$ (48+18x25)
- Altera DSP blocks compute XY (36x36)

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

Really different architectures here

Exemple: 53-bit truncated multiplier

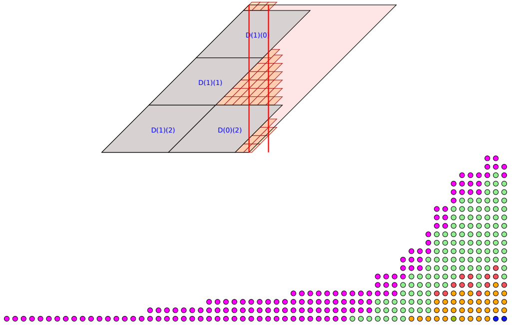


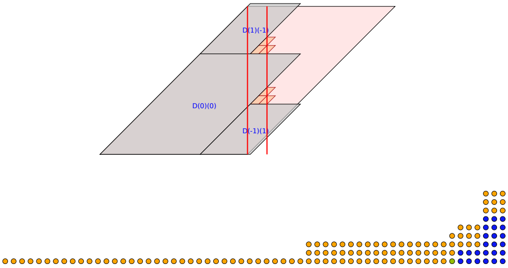


Reconciling bit heaps and DSP blocks

Instanciating DSP blocks is part of the compression

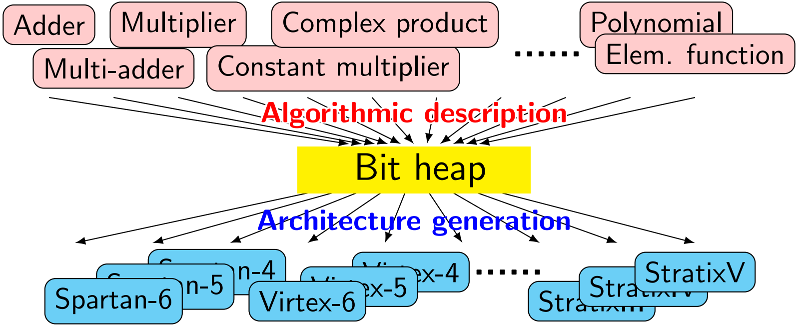
- merge operands from various sources in a DSP
- unused DSP adders may remove random bits from the heap





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)

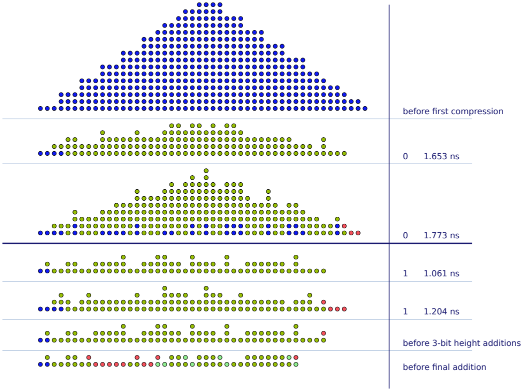
Benefits in terms of performance

... thanks to operator fusion

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

Progress in the BitHeap class benefits to many operators

Generate VHDL, test bench, and nice clickable SVG graphics



Future work, from short-term to hopeless

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

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

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