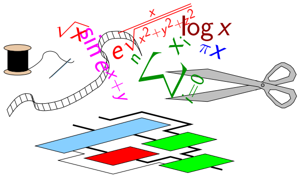
Arithmetic for FPGAs
(all the operators you will never see in a processor)

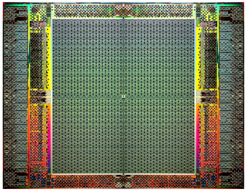
Florent de Dinechin

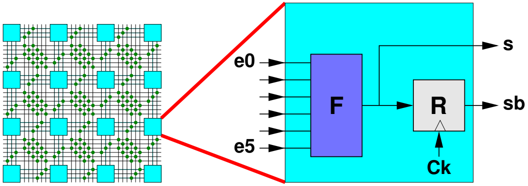


Outline

Introduction: FPGAs for computing?

Finest Programmable Granularity Around



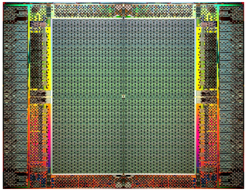


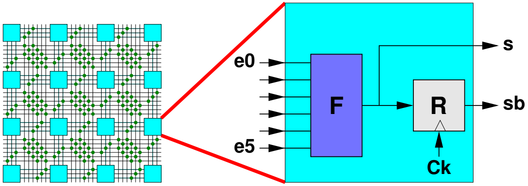
Field-Programmable Gate Array (FPGA)

Mass-produced chips, universally programmable, but:

- programming model: the circuit
- granularity: the bit

Finest Programmable Granularity Around





Field-Programmable Gate Array (FPGA)

Mass-produced chips, universally programmable, but:

- programming model: the circuit
- granularity: the bit

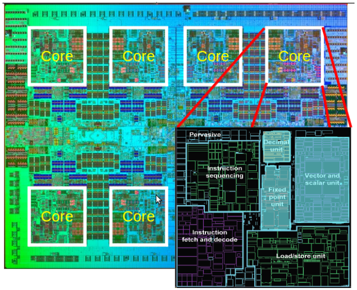
Applications

- Rapid prototyping of VLSI circuits
- Small series, where a dedicated chip would be too expensive
- Universal computing accelerators?

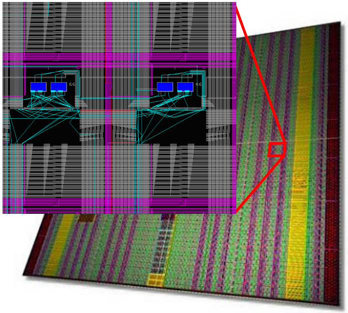
Two different ways of wasting silicon

Here are two universally programmable chips.

processeur IBM Power 7



FPGA Xilinx Virtex-4



Who's best for (insert your computation here) ?

Are FPGAs any good at floating-point?

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

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

This is the inavoidable overhead of programmability.

Are FPGAs any good at floating-point?

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

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

This is the inavoidable overhead of programmability.

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

Peak marketing lies for double-precision floating-point exponential:

- AVX core: 40 cycles / 4 DPExp @ 4GHz: **400 MDPExp/s**
- FPExp in FPGA: 1 DPExp/cycle @ 500MHz: **500 MDPExp/s**
- Chip vs chip: 8 Pentium cores vs 200 FPExp/FPGA
- **Energy/DPExp** also much better on FPGA
- Single precision comparison even better for FPGA

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

Dura Amdahl lex, sed lex

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

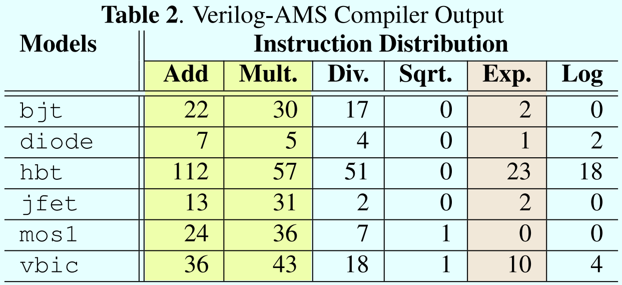
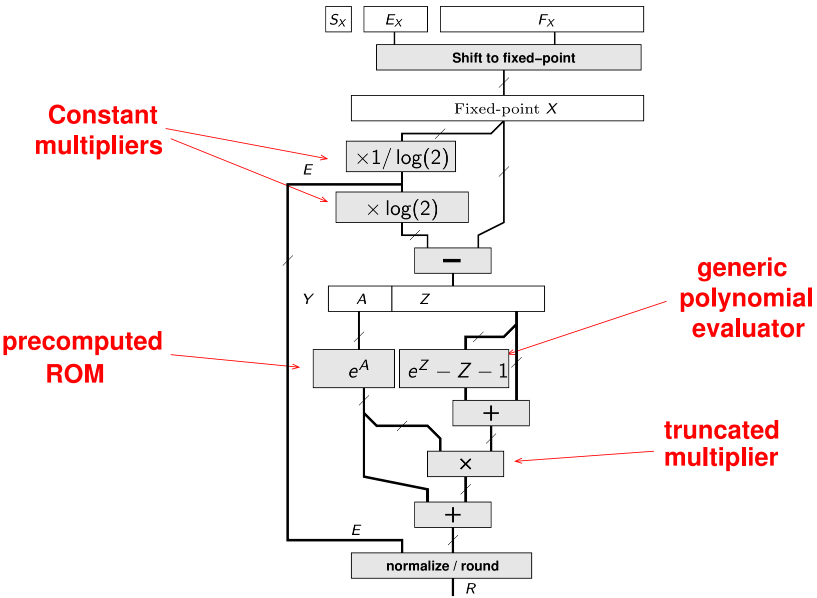


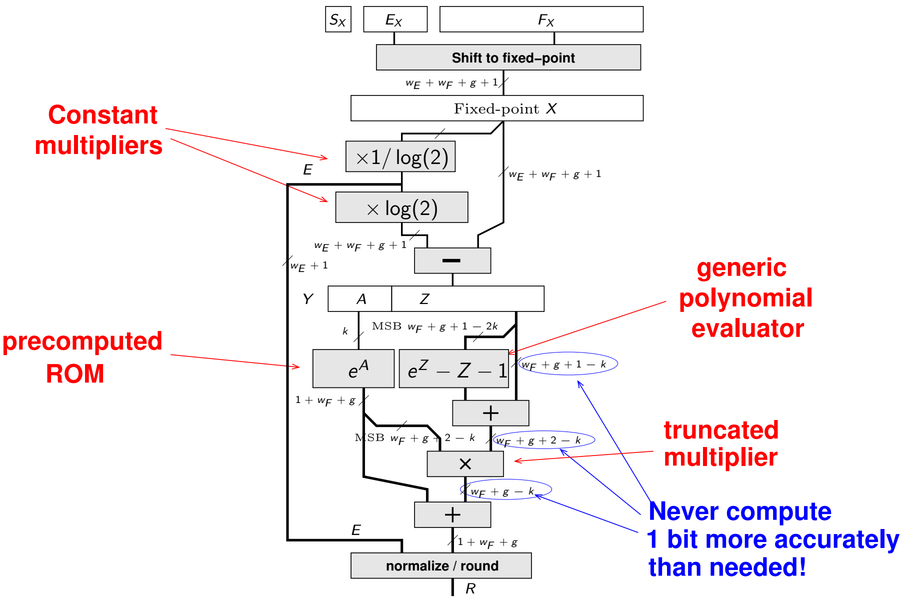
Table 2. Verilog-AMS Compiler Output

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

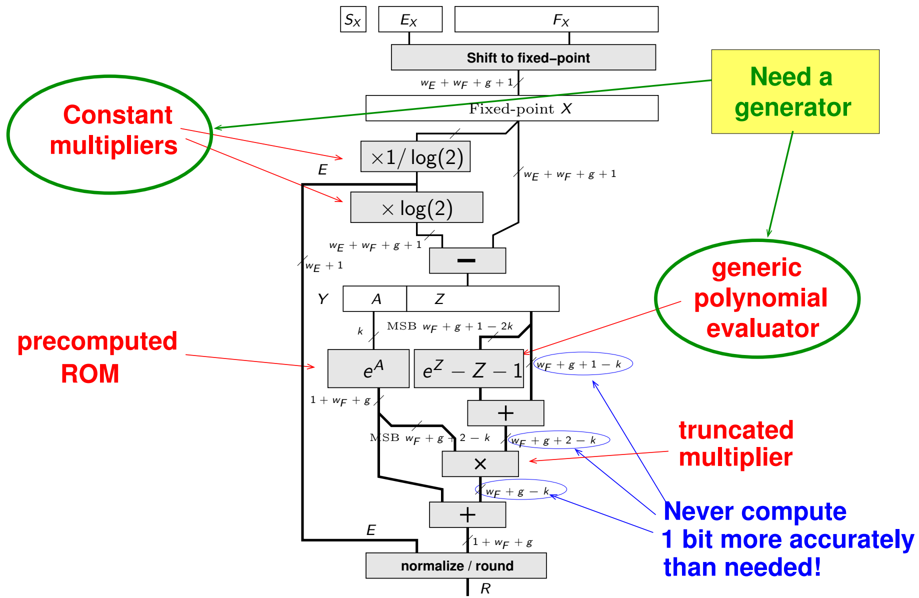
Custom arithmetic (not your Pentium's)



Custom arithmetic (not your Pentium's)



Custom arithmetic (not your Pentium's)



Useful operators that make sense in a processor

- Should a processor include hardware elementary functions ?
  Yes (Paul&Wilson, 1976), No since the transition to RISC

Useful operators that make sense in a processor

- Should a processor include hardware elementary functions ?
  Yes (Paul&Wilson, 1976), No since the transition to RISC

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

Useful operators that make sense in a processor

- Should a processor include hardware elementary functions ?
  Yes (Paul&Wilson, 1976), No since the transition to RISC

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

- Should a processor include decimal hardware?
  Yes say IBM, No say Intel

Useful operators that make sense in a processor

- Should a processor include hardware elementary functions ?
  Yes (Paul&Wilson, 1976), No since the transition to RISC
- Should a processor include a divider and square root?
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA (IBM then HP then Intel)
- Should a processor include decimal hardware?
  Yes say IBM, No say Intel
- Should a processor include a multiplier by log(2)?
  No of course.

Useful operators that make sense in an FPGA or ASIC

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

In FPGAs, useful means: useful to one application.

Enough work to keep me busy to retirement

Arithmetic operators useful to at least one application:

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

What do we call arithmetic operators?

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

What do we call arithmetic operators?

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

→ Clean mathematical definition (even for floating-point arithmetic)

What do we call arithmetic operators?

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

→ Clean mathematical definition (even for floating-point arithmetic)

The operator as a *circuit*...

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

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

The FloPoCo project

Floating Point Cores, but not only

Initial goal: FPGA arithmetic the way it should be

that is: open-ended, unlike processor arithmetic

- an open-ended list of custom operators
- open-ended data formats: all operators fully parameterized
- open-ended performance trade-off: flexible pipeline

General philosophy: *computing just right*

Floating Point Cores, but not only

Initial goal: FPGA arithmetic the way it should be

that is: open-ended, unlike processor arithmetic

- an open-ended list of custom operators
- open-ended data formats: all operators fully parameterized
- open-ended performance trade-off: flexible pipeline

General philosophy: *computing just right*

Beyond the plan

- the FloPoCo framework was successfully used to design the FPU of the Kalray processor
- FloPoCo provides the floating-point back-end to the PandA project (politecnico de Milano)

Here should come a demo

FloPoCo is freely available from

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

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

Don't trust this operator, it was written by an underpaid computer

FloPoCo is already able to generate an infinite number of operators.
We haven't tested them all.

Don't trust this operator, it was written by an underpaid computer

FloPoCo is already able to generate an infinite number of operators.
We haven't tested them all.
Two operators, TestBench and TestBenchFile, generate test benchs for the operator preceding them on the command line

- `flopoco FPExp 8 23 TestBenchFile 10000`
  generates 10000 random tests
- `flopoco IntConstDiv 16 3 -1 TestBenchFile -2`
  generates an exhaustive test

Don't trust this operator, it was written by an underpaid computer

FloPoCo is already able to generate an infinite number of operators.
We haven't tested them all.
Two operators, TestBench and TestBenchFile, generate test benchs for the operator preceding them on the command line

- `flopoco FPExp 8 23 TestBenchFile 10000`
  generates 10000 random tests
- `flopoco IntConstDiv 16 3 -1 TestBenchFile -2`
  generates an exhaustive test

Specification-based test bench generation

Not by simulation of the generated architecture!

Don't trust this operator, it was written by an underpaid computer

FloPoCo is already able to generate an infinite number of operators.
We haven't tested them all.
Two operators, TestBench and TestBenchFile, generate test benchs for the operator preceding them on the command line

- `flopoco FPExp 8 23 TestBenchFile 10000`
  generates 10000 random tests
- `flopoco IntConstDiv 16 3 -1 TestBenchFile -2`
  generates an exhaustive test

Specification-based test bench generation

Not by simulation of the generated architecture!

Operator-specific random generator overloading

- FPExp: bias towards the small interval on which it is defined
- FPAdder: bias towards catastrophic cancellation

One example of operator fusion

Floating-point sum of squares

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

(not a toy example but a useful building block)

Floating-point sum of squares

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

(not a toy example but a useful building block)

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

Floating-point sum of squares

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

(not a toy example but a useful building block)

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

Floating-point sum of squares

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

(not a toy example but a useful building block)

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

Floating-point sum of squares

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

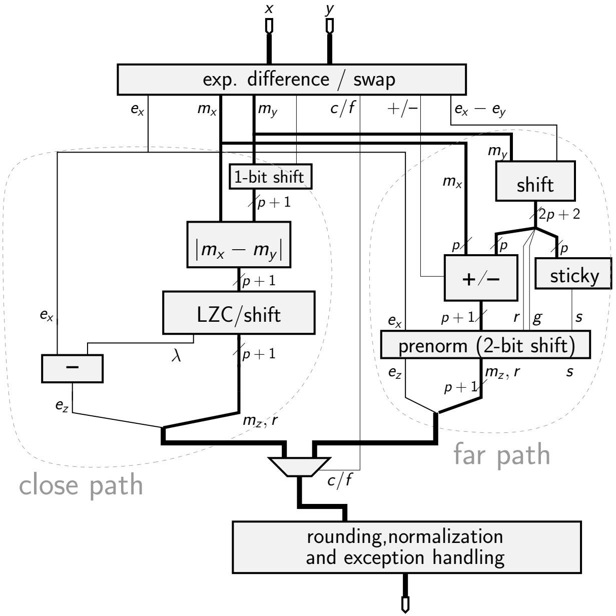
(not a toy example but a useful building block)

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

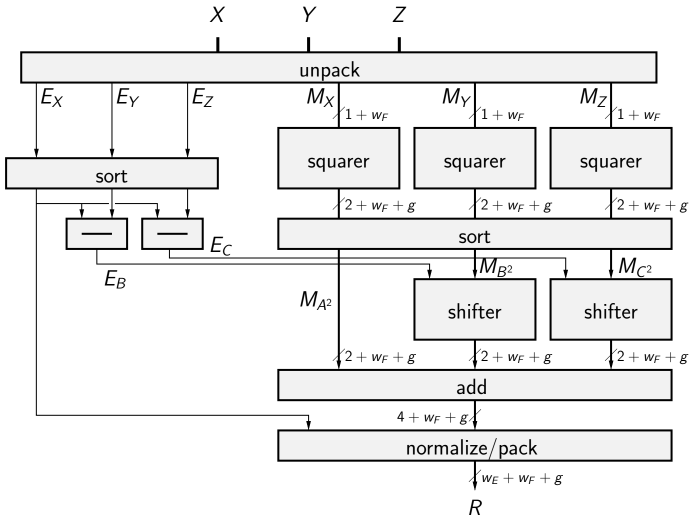
The FloPoCo Recipe

- Floating-point interface for convenience
- Clear accuracy specification for computing just right
- Fixed-point internal architecture for efficiency

A floating-point adder



A fixed-point architecture



The benefits of custom computing

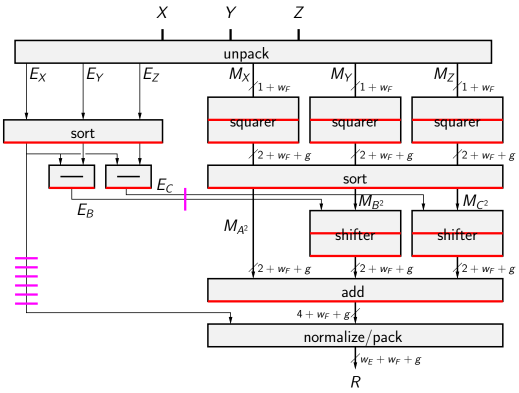
A few results for floating-point sum-of-squares on Virtex4:

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

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

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

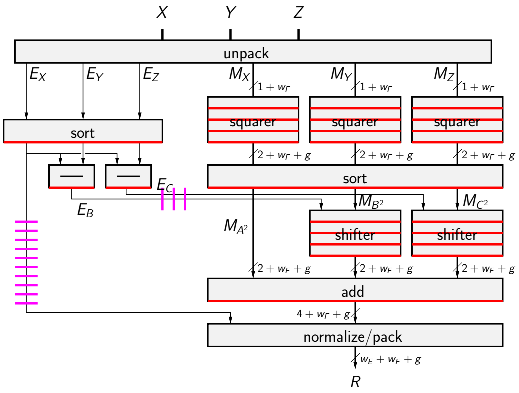
Custom also means: custom pipeline



One operator does not fit all

- Low frequency, low resource consumption

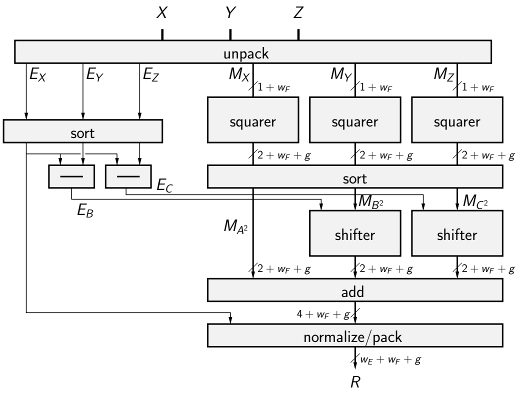
Custom also means: custom pipeline



One operator does not fit all

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

Custom also means: custom pipeline



One operator does not fit all

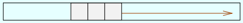
- Low frequency, low resource consumption
- Faster but larger (more registers)
- Combinatorial

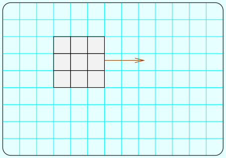
One example of operator specialization

Division by small integer constants

Motivation

divisions by 3 and by 9 in stencil applications (Jacobi)

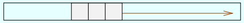


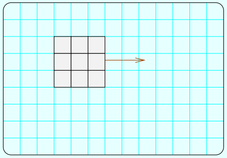


Division by small integer constants

Motivation

divisions by 3 and by 9 in stencil applications (Jacobi)





Also

- fancy address generation
- division by 10 for decimal/binary conversion,
- exponent processing in floating-point cubic root,
- coefficients 1/6 and 1/24 in sine/cosine Taylor formula
- ...

Implementation options

Floating-point, single precision, Virtex 5

- as a divider, with one input tied to 3:
  1122 Reg + 945 LUT; 17 cycles @ 290 MHz

Implementation options

Floating-point, single precision, Virtex 5

- as a divider, with one input tied to 3:
  1122 Reg + 945 LUT; 17 cycles @ 290 MHz
- as an FP multiplier with one input tied to 1/3:
  88 Reg + 130 LUT; 2 DSP blocks; 3 cycles @ 500MHz

Implementation options

Floating-point, single precision, Virtex 5

- as a divider, with one input tied to 3:
  1122 Reg + 945 LUT; 17 cycles @ 290 MHz
- as an FP multiplier with one input tied to $1/3$:
  88 Reg + 130 LUT; 2 DSP blocks; 3 cycles @ 500MHz
- as an FP "multiplier by a constant":
  149 Reg + 318 LUT; 4 cycles @ 439MHz

Implementation options

Floating-point, single precision, Virtex 5

- as a divider, with one input tied to 3:
  1122 Reg + 945 LUT; 17 cycles @ 290 MHz
- as an FP multiplier with one input tied to 1/3:
  88 Reg + 130 LUT; 2 DSP blocks; 3 cycles @ 500MHz
- as an FP "multiplier by a constant":
  149 Reg + 318 LUT; 4 cycles @ 439MHz
- the same, but exploiting the periodicity of the constant
  $(1/3 = 0.01010101010101010\ldots)$
  107 Reg + 197 LUT; 4 cycles @ 422MHz

Implementation options

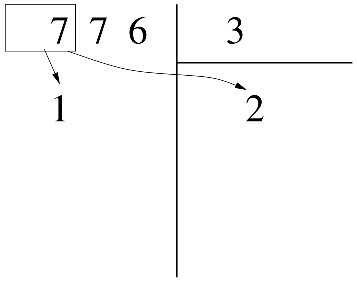
Floating-point, single precision, Virtex 5

- as a divider, with one input tied to 3:
  1122 Reg + 945 LUT; 17 cycles @ 290 MHz
- as an FP multiplier with one input tied to 1/3:
  88 Reg + 130 LUT; 2 DSP blocks; 3 cycles @ 500MHz
- as an FP "multiplier by a constant":
  149 Reg + 318 LUT; 4 cycles @ 439MHz
- the same, but exploiting the periodicity of the constant
  $(1/3 = 0.01010101010101010...)$
  107 Reg + 197 LUT; 4 cycles @ 422MHz
- this work:
  105 Reg + 83 LUT ; 3 cycles @ 411 MHz
  ... and correctly rounded (equivalent to using a divider)

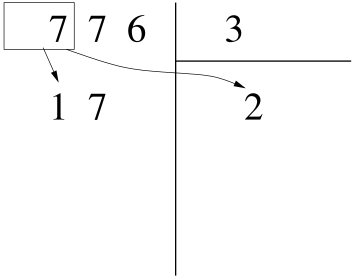
Anybody here remembers how we compute divisions?

$$\begin{array}{r|l} 7\;7\;6 & \quad 3 \\ \cline{2-2} & \\ & \\ & \end{array}$$

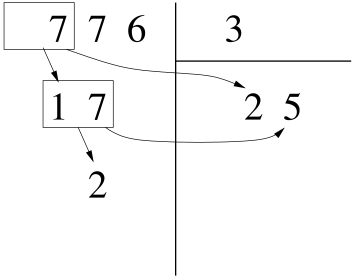
Anybody here remembers how we compute divisions?



Anybody here remembers how we compute divisions?

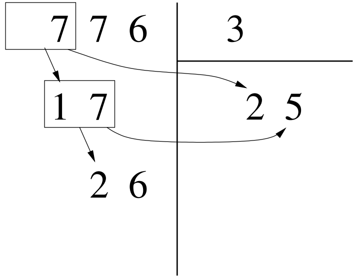


Anybody here remembers how we compute divisions?



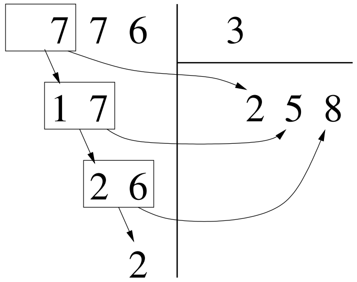
- iteration body: Euclidean division of a 2-digit decimal number by 3
- The first digit is a remainder from previous iteration: its value is 0, 1 or 2
- Possible implementation as a look-up table.

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.

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.

The same, but in binary-friendly radix

Writing an integer x in radix 2^α

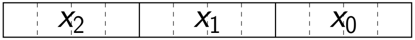
$$x = \sum_{i=0}^{n} 2^{\alpha i} x_i \qquad \text{(split of the bits of } x \text{ into chunks of } \alpha \text{ bits)}$$

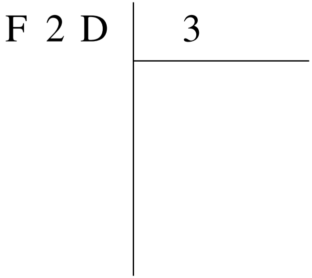
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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



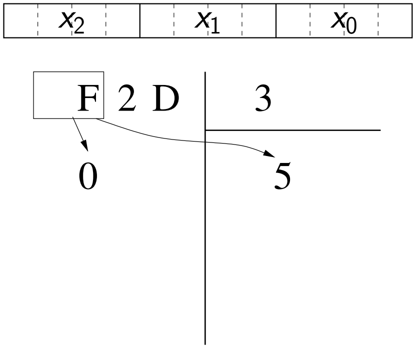


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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

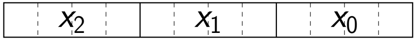


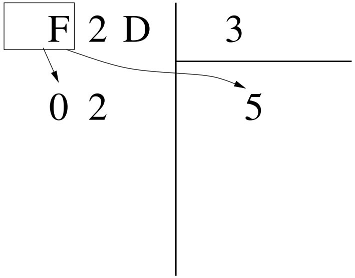
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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



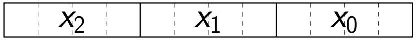


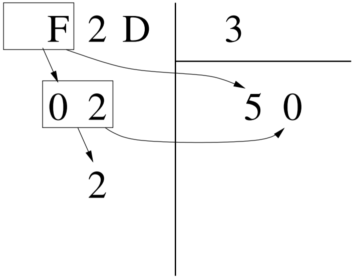
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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



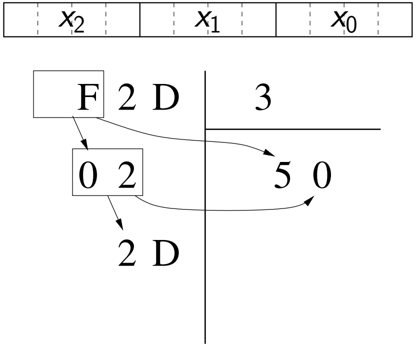


The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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

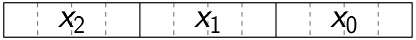


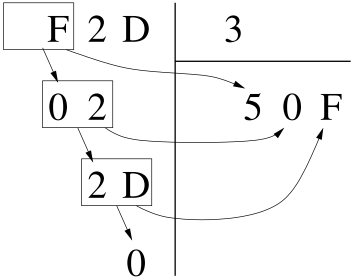
The same, but in binary-friendly radix

Writing an integer x in radix 2^α

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

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





And now for some mathematical obfuscation

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

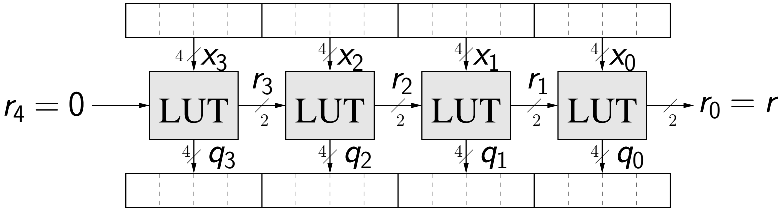
And now for some mathematical obfuscation

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

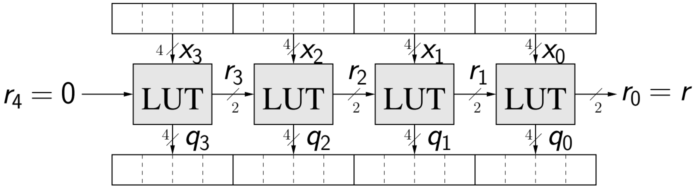
Each iteration

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

Unrolled implementation

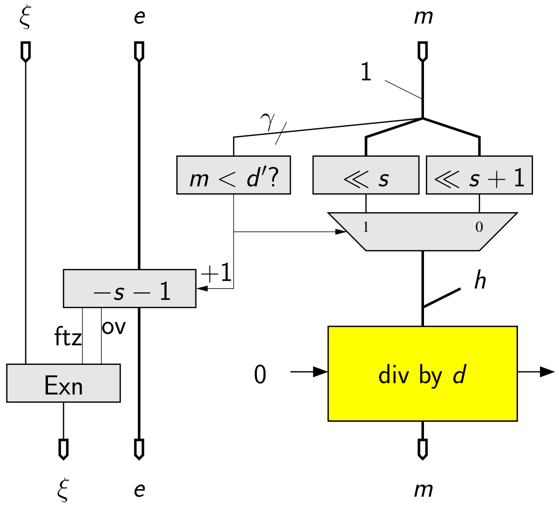


Choice of parameters for a logic-based implementation

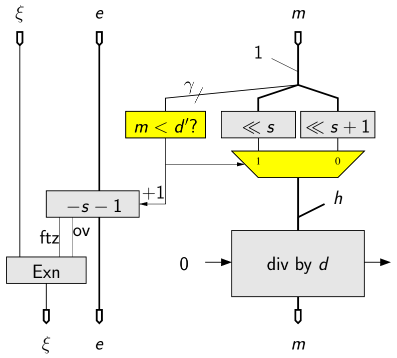


- FPGA logic is LUT-based (Example: LUT6 is a 6-input LUT)
  - A 6-bit in, 6-bit out LUT consumes 6 LUT6
  - Easy to pipeline (one register behind each LUT)
- Optimal α s.t. $\alpha + \gamma = 6$
- Efficient for small constants only (need small γ)
  Notice that 24 is actually 3...

Floating-point has low overhead

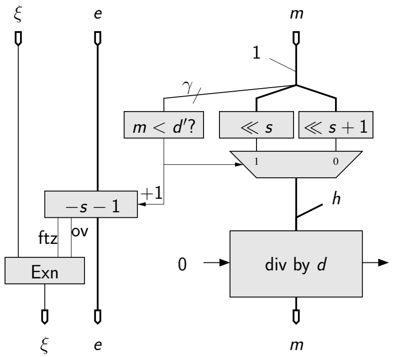


Floating-point has low overhead



- normalisation: small comparison, big mux;

Floating-point has low overhead



- normalisation: small comparison, big mux;
- rounding for free! $\circ(z/d) = \lfloor z/d + \frac{1}{2} \rfloor$

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

and this $+$ is again a concatenation (h on the picture)

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
200 Reg + 214 LUT	4 cycles @ 336 MHz
using shift-and-add	
282 Reg + 470 LUT	5 cycles @ 307 MHz

Conclusion

Was it worth to waste your precious time on division by 3?

Conclusion

Was it worth to waste your precious time on division by 3?

(this slide intentionally left blank)

My personal record

Two weeks from the first blackboard description 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

My personal record

Two weeks from the first blackboard description 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

This was advertising for the FloPoCo framework.

One example of open-ended operator

Just one slide

A polynomial evaluator for arbitrary functions

Example:

```
flopoco FunctionEvaluator "(sin(x*Pi/2))^ 2" 32 32 4
```

- The string is a *mathematical* function
- 32-bit in, 32-bit out
- Last-bit accurate (all returned bits hold useful information)
- 4 is the degree of the polynomial, allows to express a memory/multiplier trade-off
- Works for the set of functions for which it works

Another one is HOTBM.
Still work in progress...

Conclusion

My current crusade

The evil (at least on FPGAs)

"A fast implementation of single-precision floating-point exponential (but accurate to 2^{-8} only)"

Do you see why it is wrong?

My current crusade

The evil (at least on FPGAs)

"A fast implementation of single-precision floating-point exponential (but accurate to 2^{-8} only)"

Do you see why it is wrong?

A line I shall have in each of my talks until the world is saved

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

My current crusade

The evil (at least on FPGAs)

"A fast implementation of single-precision floating-point exponential (but accurate to 2^{-8} only)"

Do you see why it is wrong?

A line I shall have in each of my talks until the world is saved

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

Or maybe this one

Do you really need to compute this bit?

The “no killer app” theorem

For 20 years, the FPGA community has been waiting for the “killer application”.
(The widely useful application on which the FPGA is so much better)

The “no killer app” theorem

For 20 years, the FPGA community has been waiting for the “killer application”.
(The widely useful application on which the FPGA is so much better)

Theorem: we’ll wait forever.

The “no killer app” theorem

For 20 years, the FPGA community has been waiting for the “killer application”.
(The widely useful application on which the FPGA is so much better)

Theorem: we’ll wait forever.

Proof: When such an application pops up,

- either it is indeed widely useful, and next year’s Pentium will do it in hardware 10x faster than the FPGA, so it won’t be an *FPGA* killer app next year,
- or the FPGA remains competitive next year, but it means that it was not a killer app.

The “no killer app” theorem

For 20 years, the FPGA community has been waiting for the “killer application”.
(The widely useful application on which the FPGA is so much better)

Theorem: we’ll wait forever.

Proof: When such an application pops up,

- either it is indeed widely useful, and next year’s Pentium will do it in hardware 10x faster than the FPGA, so it won’t be an *FPGA* killer app next year,
- or the FPGA remains competitive next year, but it means that it was not a killer app.

The killer feature of FPGAs is flexibility

To exploit it, we do need infinitely many arithmetic operators.

Computing just right

In a Pentium

the choice is between

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

Computing just right

In a Pentium

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 Pentium

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!

An almost virgin land

Most of the arithmetic literature addresses the construction of SUVs.

So when do we have an FPGA in every PC?

When they become as easy to program as processors?

(now that's a challenge)

So when do we have an FPGA in every PC?

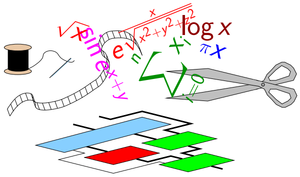
When they become as easy to program as processors?

(now that's a challenge)

(or do we quietly wait for processors to become as messy to program as FPGAs?)

Thanks for your attention

The following people have contributed to FloPoCo:
S. Banescu, N. Brunie, S. Collange, J. Detrey,
P. Echeverría, F. Ferrandi, M. Grad, K. Illyes,
M. Iștoan, M. Joldeș, C. Klein, D. Mastrandrea,
B. Pașca, B. Popa, X. Pujol, D. Thomas,
R. Tudoran, A. Vasquez.



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