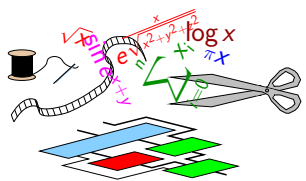
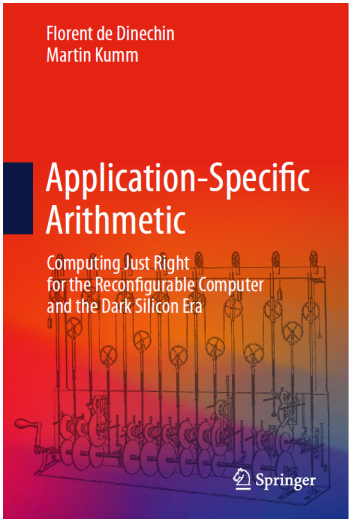
# FPGAs Computing Just Right: Application-Specific Arithmetic

**Florent de Dinechin**




citi lab
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
Inria informatique mathématiques
UNIVERSITÉ DE LYON

# Outline (complete with advertizing and all)




Anti-introduction: the arithmetic you want in a processor
What they didn't tell you about FPGA architectures
Some opportunities of hardware computing just right
A few FPGA success stories
Conclusion

# Anti-introduction: the arithmetic you want in a processor

# General-Purpose Processors and GPUs

The good arithmetic in a general-purpose processor is the most generally useful: **additions**, **multiplications**, and then?

- *Should a processor include a* **divider***? A* **square root***?*
- *Should a processor include* **elementary functions** *(exp, log sine/cosine)? Which?*
- *Should a processor include decimal hardware?*
- *Should a processor include an 8-bit tensor multiplier?*
- ...

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
       |----
       | ?
       |
       |
       |
```

## Just like decimal, but simpler

- find the next quotient digit

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
 011001 | 101
 -101   |-----
=0001   | 1
        |
        |
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | ----
 00101 | 1
       |
       |
       |
       |
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- start again, one digit to the right

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 1?
 00101 |
       |
       |
       |
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- start again, one digit to the right

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 11
 00101
 -101
=1101
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
- start again, one digit to the right

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 10?
 00101 |
 -101  |
=1101  |
 00101 |
       |
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 101
 00101 |
 -101  |
=1101  |
 00101 |
  -101 |
  =0000|
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

# Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 101
 00101 |
 -101  |
=1101  |
 00101 |
  -101 |
   000 |
```

## Just like decimal, but simpler

- find the next quotient digit — *it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

Light iteration (one subtraction and one test), but **one bit of the quotient per iteration**:
(more than) **53 cycles** for double-precision floating-point

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)

... and this divider should be a **fast** one, because of **Amdahl law**:

Although division is not frequent, *(...) a high latency divider can contribute an additional 0.50 CPI to a system executing SPECfp92*

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)

... and this divider should be a **fast** one, because of **Amdahl law**:

Although division is not frequent, *(...) a high latency divider can contribute an additional 0.50 CPI to a system executing SPECfp92*

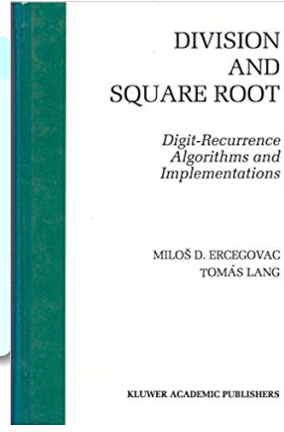
## Digit recurrence algorithms

Generalizations of the paper-and-pencil algorithm

- large radix: from $2^3$ to $2^6$
- fancy internal number systems to speedup
  - digit-by-number product
  - subtraction
  - finding the next quotient digit

Heavier iterations, giving one digit (2 to 5 bits) per iteration.

A lot of research, worth one full book (Ercegovac and Lang, 1994)



# Should a processor include a divider? (3)

Answer in 2000 is : **NO** (Markstein)

# Should a processor include a divider? (3)

Answer in 2000 is : **NO** (Markstein)

The Itanium: a brand new, expensive processor... without a divide instruction.

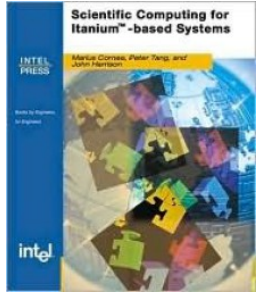
**Instead of a hardware divider,**

**a second FMA (fused multiply and add) is more generally useful**

... and can even be used to compute divisions:

## Multiplicative division algorithms

- several algorithms
  using a handful of multiplications
- the freedom of software:
  - quick and dirty, or accurate but slow
  - high throughput or short latency
  - ...
- and with a second FMA,
  BLAS and FFTs are 2x faster !


... and two more books.

# Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

# Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally wasteful** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - speculation all over the place:
    compute many options in parallel, then discard them all except one
- in a processor that is supposed to go in your smartphone?!?

# Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally wasteful** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - speculation all over the place:
    compute many options in parallel, then discard them all except one
- in a processor that is supposed to go in your smartphone?!?

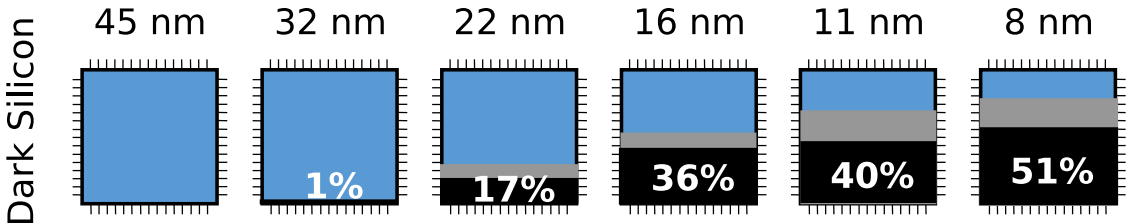
*We do this to reduce overal energy consumption!*
*There is this huge superscalar ARM core that consumes a lot,*
*we save energy if we can switch it off a few cycles earlier*

# A good example of dark silicon made useful

## Dark silicon?

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

We just can't dissipate the heat, and it gets worse with Moore's Law.
"Dark silicon" is the percentage that must be off at a given time




(picture from a 2013 HiPEAC keynote by Doug Burger)

# Pleasant times to be an architect

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

Hardware implementations of rare (but useful) operations:

- when used, dramatically reduce the energy per operation
  (compared to a software implementation that would take many more cycles)
- when unused (i.e. most of the time), serve as radiator for the parts in use

# Should a processor include elementary functions? (1)

Dura Amdahl lex, sed lex

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

| 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 |

Current performance of exp or log is 10 to 100 cycles,

to compare with 1 to 5 cycles for add and mult.

# Should a processor include elementary functions? (2)

Answer in 1976 is **YES** (Paul&Wilson)

# Should a processor include elementary functions? (2)

Answer in 1976 is **YES** (Paul&Wilson)
... and the initial x87 floating-point coprocessor was designed with a basic set of elementary functions

- implemented in microcode
- with some hardware assistance,
  in particular the 80-bit floating-point format.

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

## Table-based algorithms

- Moore's Law means cheap memory
- Fast algorithms thanks to huge (tens of Kbytes!) **tables of pre-computed values**
- Software beats micro-code, which cannot afford such tables

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

## Table-based algorithms

- Moore's Law means cheap memory
- Fast algorithms thanks to huge (tens of Kbytes!) **tables of pre-computed values**
- Software beats micro-code, which cannot afford such tables

None of the RISC processors designed in this period

even considers elementary functions support

# Should a processor include elementary functions? (4)

Answer in 2022 is... **sometimes**?

# Should a processor include elementary functions? (4)

Answer in 2022 is... **sometimes**?

- A few low-precision hardware functions in NVidia GPUs (Oberman & Siu 2005)
- The SpiNNaker-2 chip includes hardware exp and log (Mikaitis et al. 2018)
- Intel AVX-512 includes all sort of fancy floating-point instructions to speed up elementary function evaluation (Anderson et al. 2018)

# I won't answer the other questions here

- ✓ *Should a processor include a divider and square root?*
- ✓ *Should a processor include elementary functions (exp, log sine/cosine)?*
- *Should a processor include decimal hardware?*
- *Should a processor include an FFT operator?*
- *Should a processor include an AI accelerator?*
- ...
- *Should a processor include a divider by 3? A multiplier by* log(2) *?*

no, of course.

# At this point of the talk...

... everybody is wondering when I start talking about FPGAs.

# One nice thing with FPGAs

On FPGAs, there is a simpler answer to all these questions

| | |
|---|---|
| ✓ *divider? square root?* | Yes iff your application needs it |
| ✓ *elementary functions?* | Yes iff your application needs it |
| ✓ *FFT operator?* | Yes iff your application needs it |

# One nice thing with FPGAs

On FPGAs, there is a simpler answer to all these questions

| | |
|---|---|
| ✓ *divider? square root?* | Yes iff your application needs it |
| ✓ *elementary functions?* | Yes iff your application needs it |
| ✓ *FFT operator?* | Yes iff your application needs it |
| ✓ *multiplier by* $\log(2)$*? By* $\sin \frac{17\pi}{256}$*?* | Yes iff your application needs it |

# One nice thing with FPGAs

On FPGAs, there is a simpler answer to all these questions

- ✓ *divider? square root?* — Yes iff your application needs it
- ✓ *elementary functions?* — Yes iff your application needs it
- ✓ *FFT operator?* — Yes iff your application needs it
- ✓ *multiplier by* $\log(2)$*? By* $\sin \frac{17\pi}{256}$*?* — Yes iff your application needs it
- ...

In FPGAs, useful means: useful to one application.

# In an FPGA, you pay only for what you need

If your application is to simulate `jfet`,

| 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 |

... you want to build a floating-point unit with 13 adds, 31 mults, 2 divs, 2 exps, **and nothing more**.

# Conclusion so far

FPGA arithmetic $\neq$ arithmetic for CPUs or GPGPUs

## Application-specific arithmetic

All sorts of arithmetic operators that just **wouldn't make sense** in a processor can be useful in FPGAs.
This is what we are going to explore (and the object of the FloPoCo project).

# Conclusion so far

FPGA arithmetic $\neq$ arithmetic for CPUs or GPGPUs

## Application-specific arithmetic

All sorts of arithmetic operators that just **wouldn't make sense** in a processor can be useful in FPGAs.
This is what we are going to explore (and the object of the FloPoCo project).

This is a **qualitative** question, but there is a related **quantitative** question:

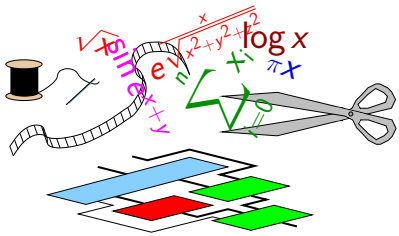
## Computing just right

In a processor, data is 8, 16, 32 or 64 bits (at best).
In an FPGA, data formats may be tightly fitted to the requirements of the application:

**if you need 17 bits, compute only 17 bits**

# Computing just right?

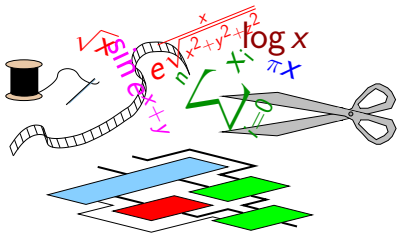
This is the pathetic logo of the FloPoCo project:




(the proper term is probably *allogory*)

# Computing just right?

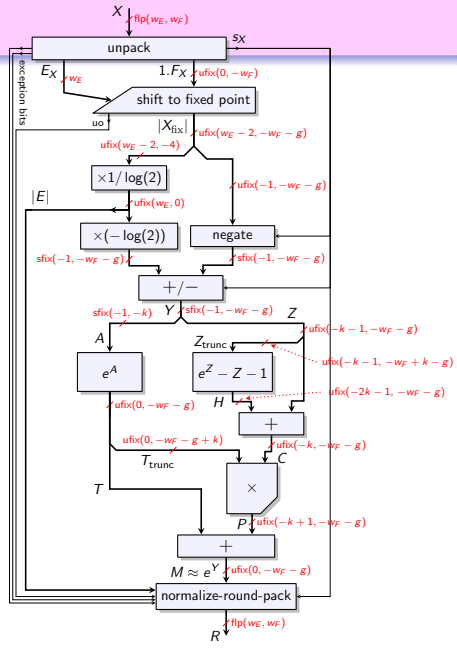
This is the pathetic logo of the FloPoCo project:



(the proper term is probably *allogory*)

This is the kind of thing FloPoCo does ⟶
It is a floating-point exponential operator
where each wire, each component is
**tailored to its context** with love and care.

(not a very good logo either)



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

## In software, if your result is correct, it is probably wasteful

Did you really need single precision (8 decimal digits of accuracy) in Angry birds considering that the trajectory was input using your fat fingers?

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

## In software, if your result is correct, it is probably wasteful

Did you really need single precision (8 decimal digits of accuracy) in Angry birds considering that the trajectory was input using your fat fingers?

## Plain common sense

- If the lower bits carry useless noise, you don't want to compute them...
- ... and you want even less to store them, transmit them, compute on them.

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

## In software, if your result is correct, it is probably wasteful

Did you really need single precision (8 decimal digits of accuracy) in Angry birds considering that the trajectory was input using your fat fingers?

## Plain common sense

- If the lower bits carry useless noise, you don't want to compute them...
- ... and you want even less to store them, transmit them, compute on them.

## Here we have more freedom when designing hardware

- In a circuit, we may choose, for each variable, how many bits are computed/stored/transmitted! $\longrightarrow$ **the opportunities**
- Overwhelming freedom! Help! $\longrightarrow$ **the challenges**

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

## In software, if your result is correct, it is probably wasteful

Did you really need single precision (8 decimal digits of accuracy) in Angry birds considering that the trajectory was input using your fat fingers?

## Plain common sense

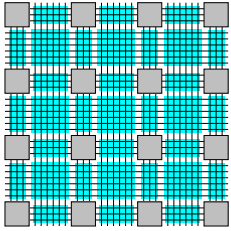
- If the lower bits carry useless noise, you don't want to compute them...
- ... and you want even less to store them, transmit them, compute on them.

## Here we have more freedom when designing hardware

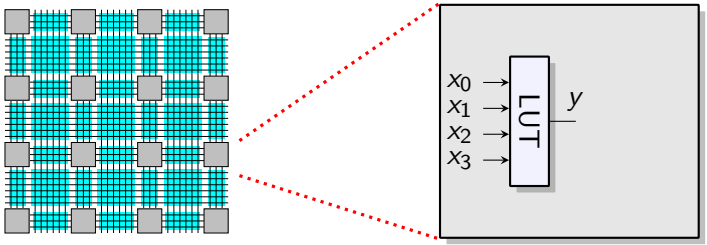
- In a circuit, we may choose, for each variable, how many bits are computed/stored/transmitted! $\longrightarrow$ **the opportunities**
- Overwhelming freedom! Help! $\longrightarrow$ **the challenges**

# What they didn't tell you about FPGA architectures

# Core FPGA structure

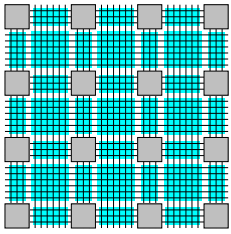


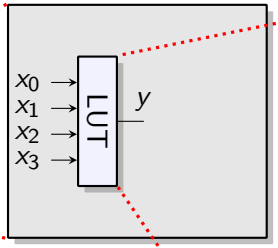
# Core FPGA structure




- to emulate an arbitrary logic function:
  a *Look-Up Table (LUT)*
  (arbitrary truth table)

# Core FPGA structure

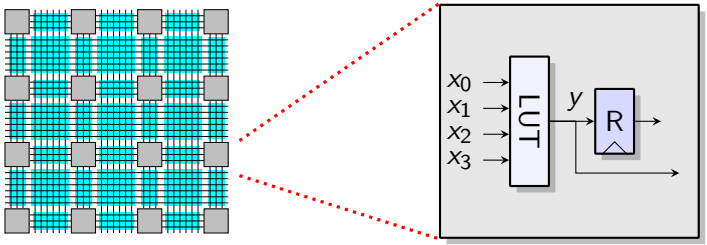





- to emulate an arbitrary logic function:
  a *Look-Up Table (LUT)*
  (arbitrary truth table)

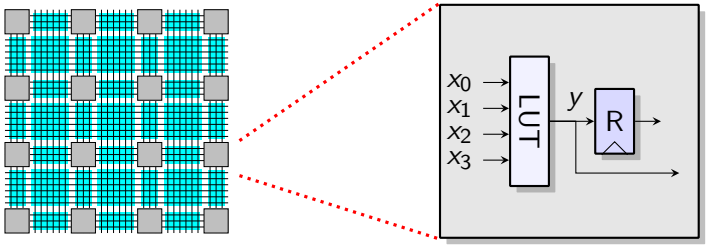
| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $y$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Core FPGA structure

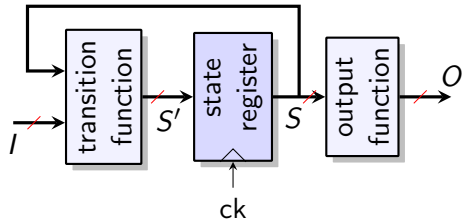



- to emulate an arbitrary logic function:
  a *Look-Up Table (LUT)*
  (arbitrary truth table)
- one flip-flop

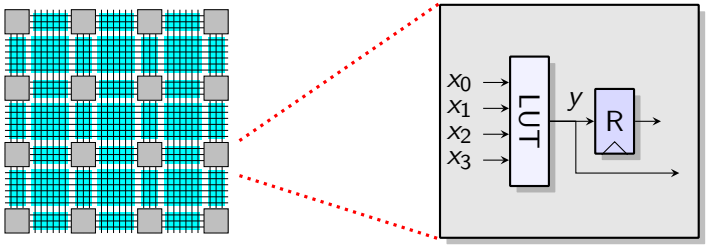
# Core FPGA structure



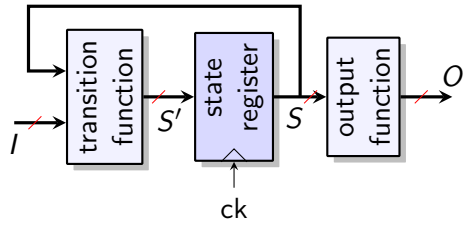

- By linking such cells
  we may emulate
  any **finite-state machine**
  hence any sequential circuit.



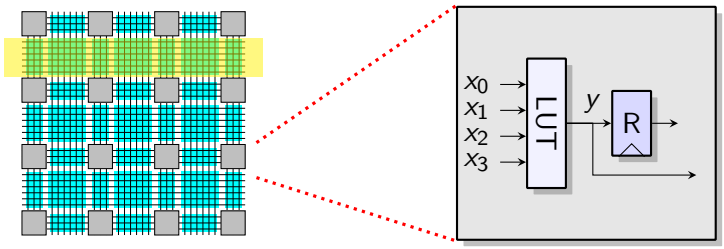
# Core FPGA structure



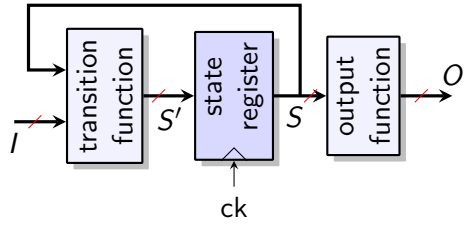

- By linking such cells
  we may emulate
  any **finite-state machine**
  hence any sequential circuit.
- How to link them?



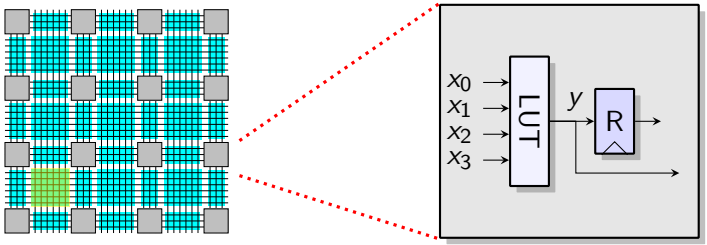
# Core FPGA structure



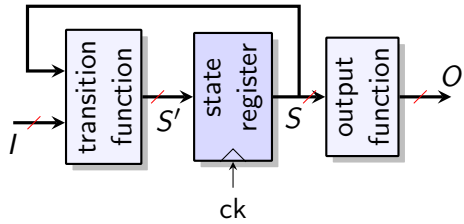

- By linking such cells
  we may emulate
  any **finite-state machine**
  hence any sequential circuit.
- How to link them?
  - *routing channels*



# Core FPGA structure

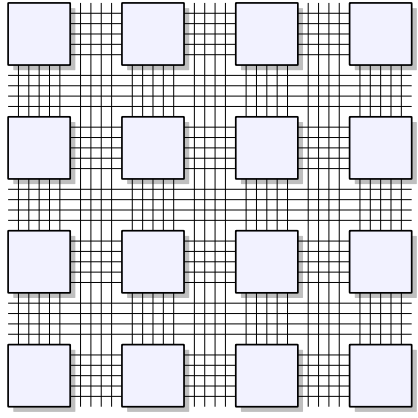



- By linking such cells
  we may emulate
  any **finite-state machine**
  hence any sequential circuit.
- How to link them?
  - *routing channels*
  - *switch boxes*



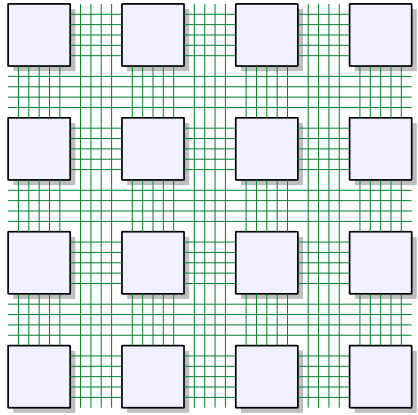
# When reality kicks back

How many wires do we need per routing channel for random access to distant cells?
1990:



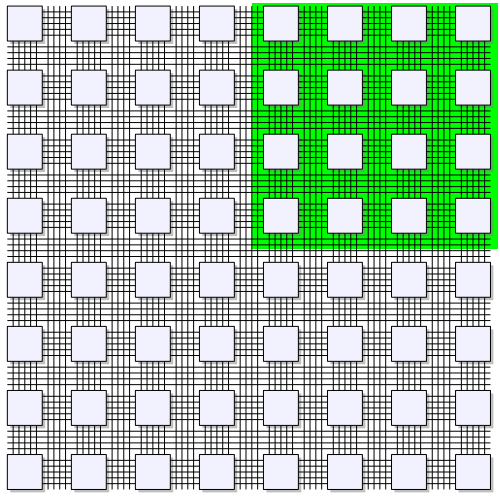
# When reality kicks back

How many wires do we need per routing channel for random access to distant cells?
1990:



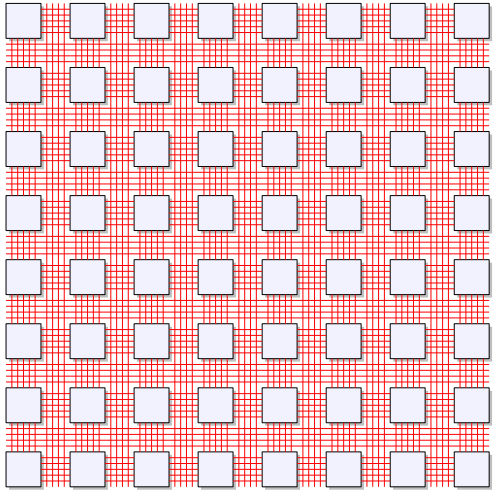
# When reality kicks back

How many wires do we need per routing channel for random access to distant cells?
1993:



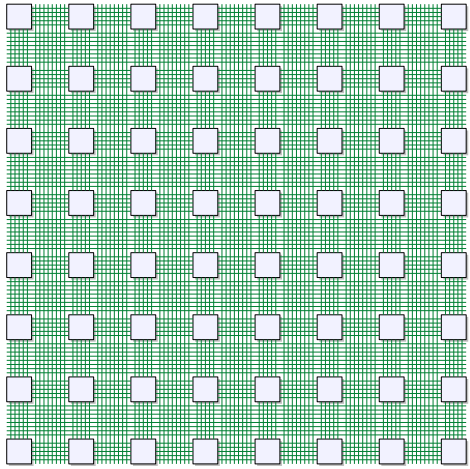
# When reality kicks back

How many wires do we need per routing channel for random access to distant cells?
1993:



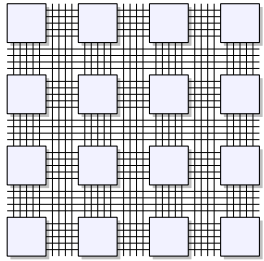
# When reality kicks back

How many wires do we need per routing channel for random access to distant cells?
1994:



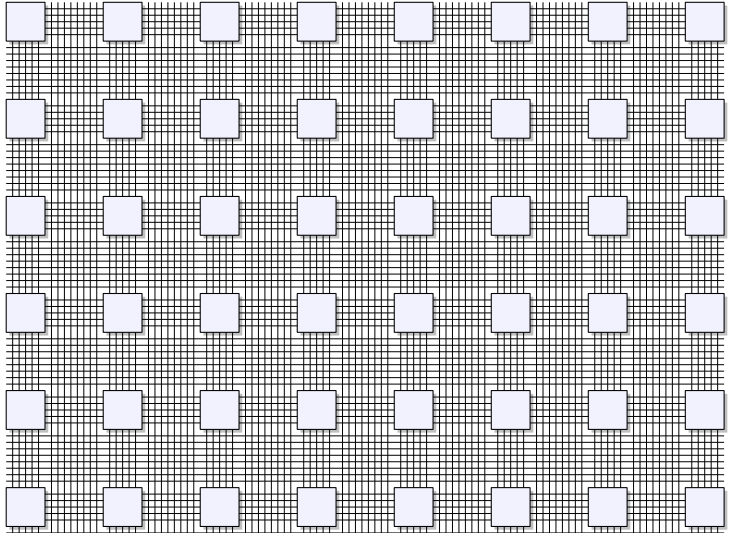
# When reality kicks back

1990:



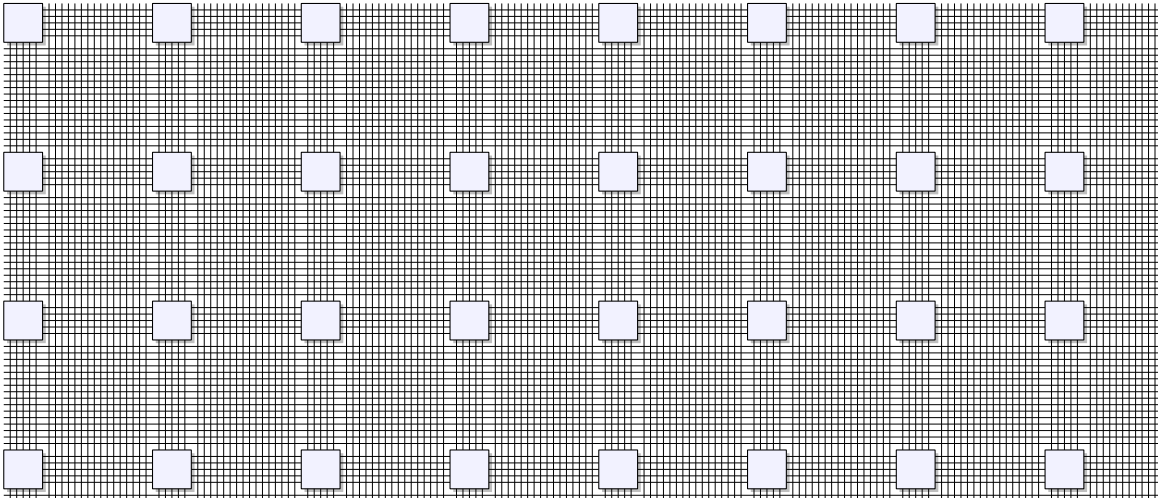
# When reality kicks back

1994:



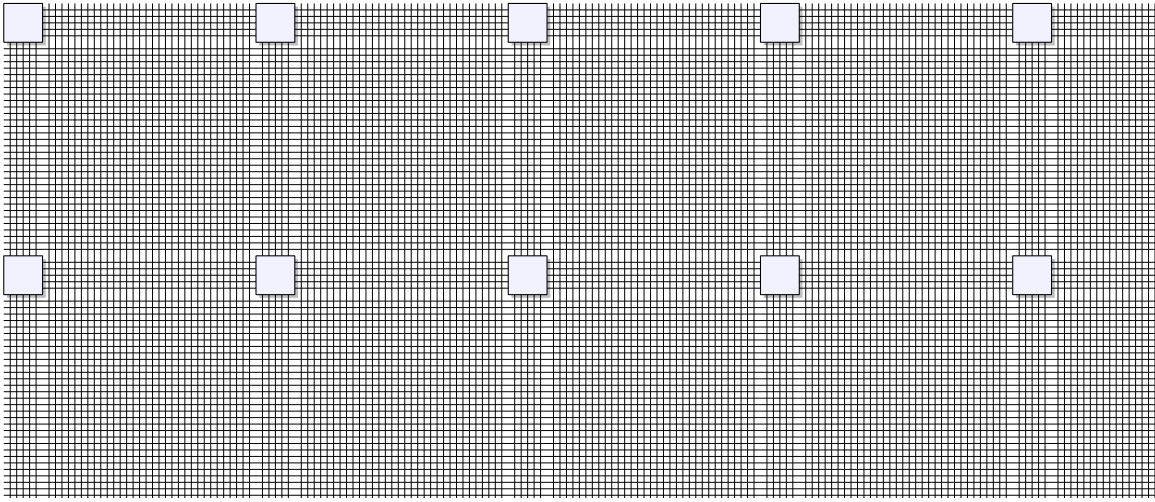
# When reality kicks back

1996:



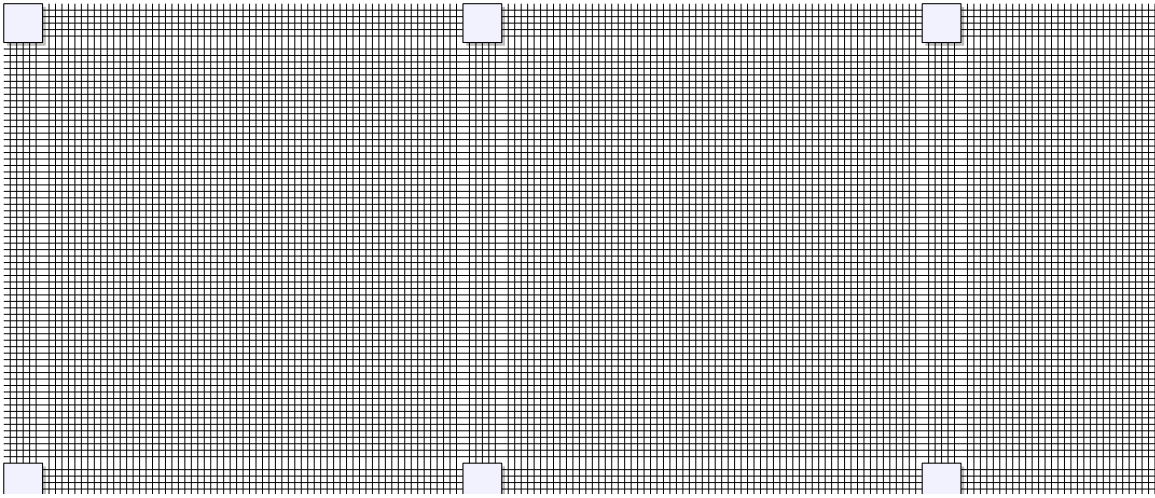
# When reality kicks back

1999:



# “Customers buy logic, but they pay for routing” (M. Langhammer)

2001 (after which I quit counting):



# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

- you plan a city, with roads sized to avoid traffic jams at the time

# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

- you plan a city, with roads sized to avoid traffic jams at the time
- then the city grows, and more people want to use the roads in the old center

# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

- you plan a city, with roads sized to avoid traffic jams at the time
- then the city grows, and more people want to use the roads in the old center
- so they get traffic-jammed.

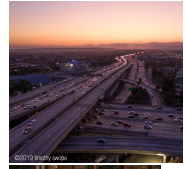
# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

- you plan a city, with roads sized to avoid traffic jams at the time
- then the city grows, and more people want to use the roads in the old center
- so they get traffic-jammed.

Two examples of this process at work:

- **Los Angeles**
  - since the 30s: one century of car-centered progress
  - ever-wider highways built in place of housing (see Roger Rabbit)
  - now: 2/3 of the area dedicated to cars (roads + parking lots)
- **Lyon**
  - in the 70s: planned destruction of a Renaissance area to make space for a highway
  - conservatism, opposition to progress $\longrightarrow$ project cancellation
  - now: a car-free area, and UNESCO world heritage



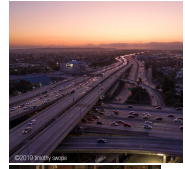
# The curse of Los Angeles

... or, the fatality of traffic jams in an expanding city.

- you plan a city, with roads sized to avoid traffic jams at the time
- then the city grows, and more people want to use the roads in the old center
- so they get traffic-jammed.

Two examples of this process at work:

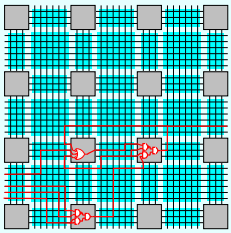
- **Los Angeles**
  - since the 30s: one century of car-centered progress
  - ever-wider highways built in place of housing (see Roger Rabbit)
  - now: 2/3 of the area dedicated to cars (roads + parking lots)
- **Lyon**
  - in the 70s: planned destruction of a Renaissance area to make space for a highway
  - conservatism, opposition to progress $\longrightarrow$ project cancellation
  - now: a car-free area, and UNESCO world heritage



The circuit variant of this curse is called Rent's law.

# Rent's law

## Yet another experimental law



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

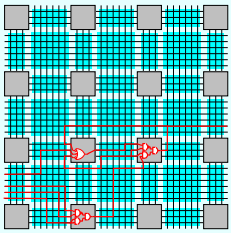
- more than proportional to $n$, the diameter,
- not quite proportional to the area $n^2$ of each half-circuit.

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

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

# Rent's law

## Yet another experimental law



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

- more than proportional to $n$, the diameter,
- not quite proportional to the area $n^2$ of each half-circuit.

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

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

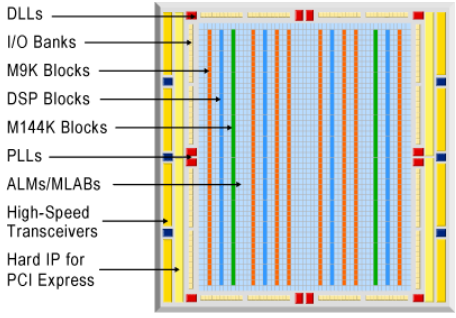
Our city planners should take crash courses in complexity theory.

# Addressing Rent's law

| **In cities:** | **Transposed to FPGA:** |
|---|---|
| Build highways of various widths | heterogeneous routing |
| Build busses, metro, tramway | increase compute granularity |
| Relocalize the economy | relocalize computations |
| | |
| *And for you, the user:* | |
| Use bicycles instead of SUVs | compute just right |

# Current FPGAs (with all these solutions)

- coarser cells, optimized for additions (up to 2,000,000 6-input LUT)
- small (24 bit) multipliers ("DSP blocks") (up to 3,000)
- small ($\approx$ 10 kBit) memories (up to 2,000)
- many independent clock networks & PLLs
- flexible input/outputs
- ...

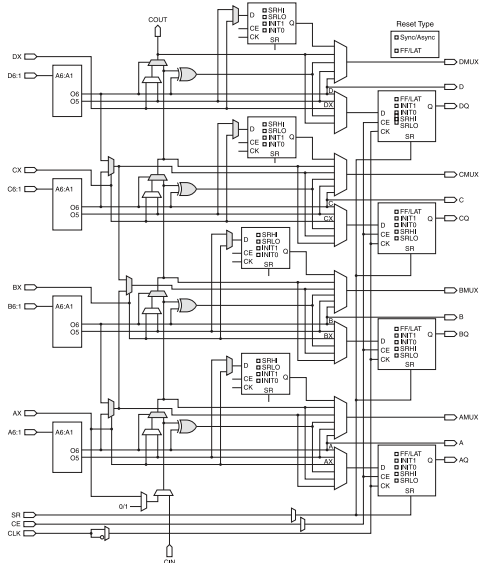



*the Altera/Intel stratix IV FPGA*

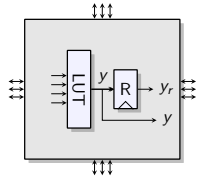
## ... and still, the price of routing

- A circuit that would fit in 1 mm$^2$ of ASIC silicon will only fit in a 50mm$^2$ FPGA...
- ... and the configured FPGA will run at 1/10th the frequency of the ASIC
  - there are transistors on all the wires!

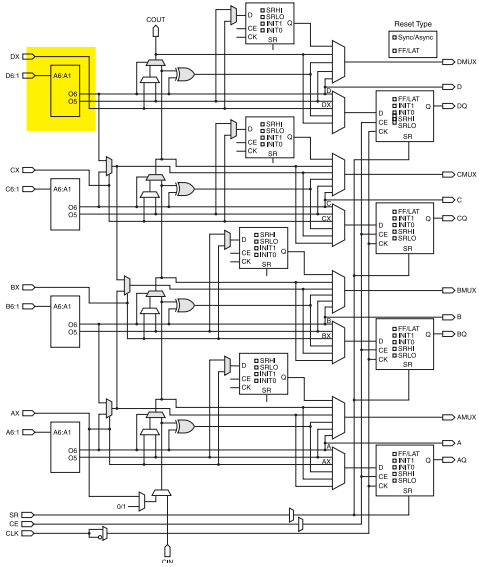
# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block




A “slice” (Virtex7)

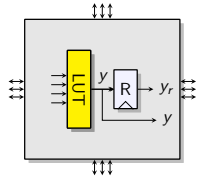


# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block

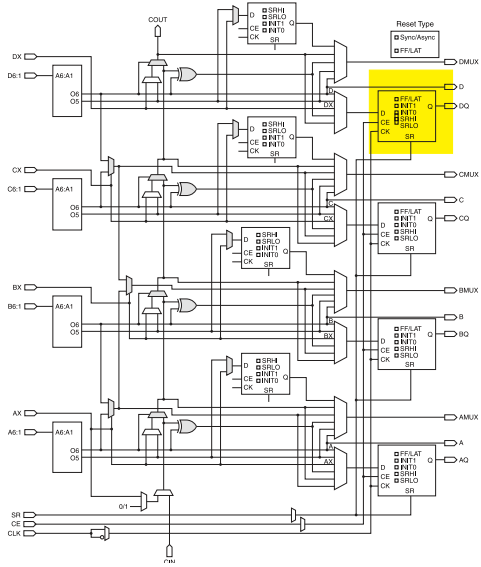



A "slice" (Virtex7)

- See the LUT?

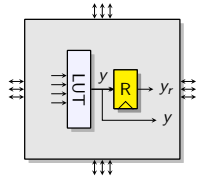


# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block

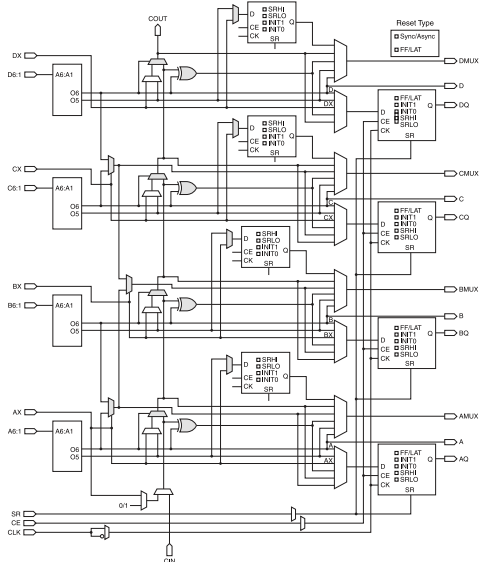



A "slice" (Virtex7)

- See the LUT?
- ... the register?

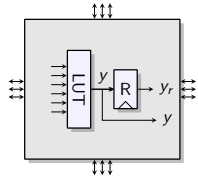


# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block



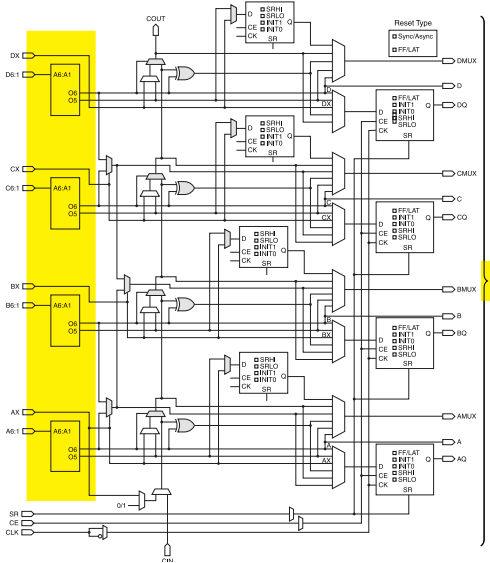

A "slice" (Virtex7)

- See the LUT?
- ... the register?



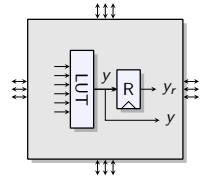

- Granularity increasing
  - 6-input LUTs (and counting)

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block



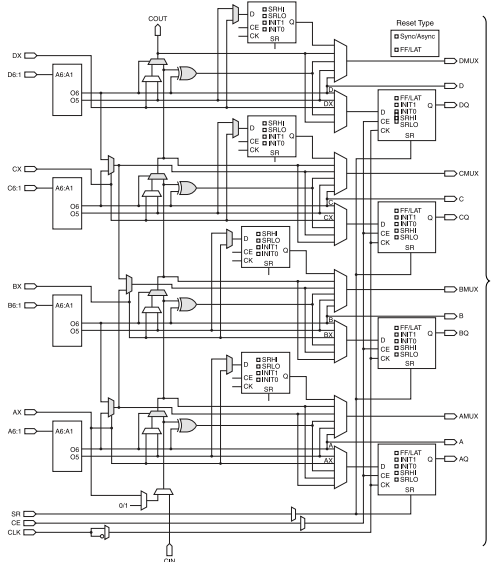

A "slice" (Virtex7)

- See the LUT?
- ... the register?



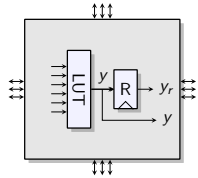

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block



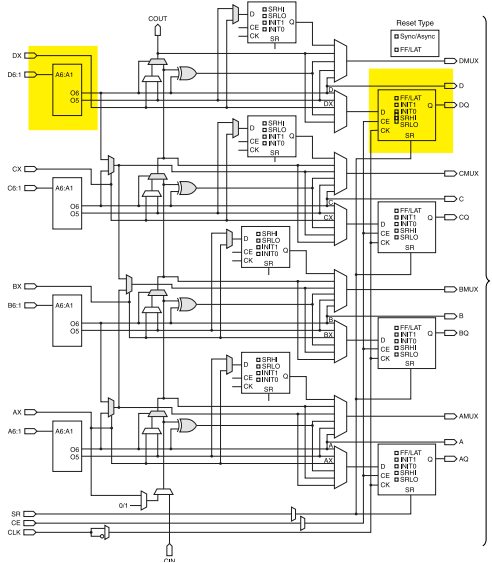

A "slice" (Virtex7)

- See the LUT?
- ... the register?



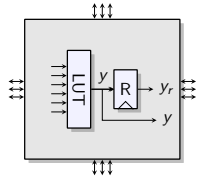

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)
  - 2 slices/CLB (and counting)

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block



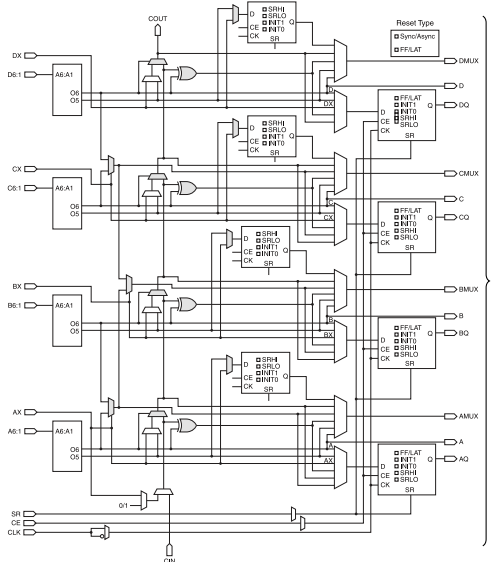

A "slice" (Virtex7)

- See the LUT?
- ... the register?



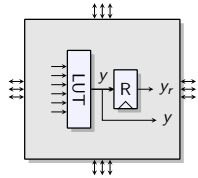

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)
  - 2 slices/CLB (and counting)
  - Ratio reg/LUT still equal to 1

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block




A "slice" (Virtex7)

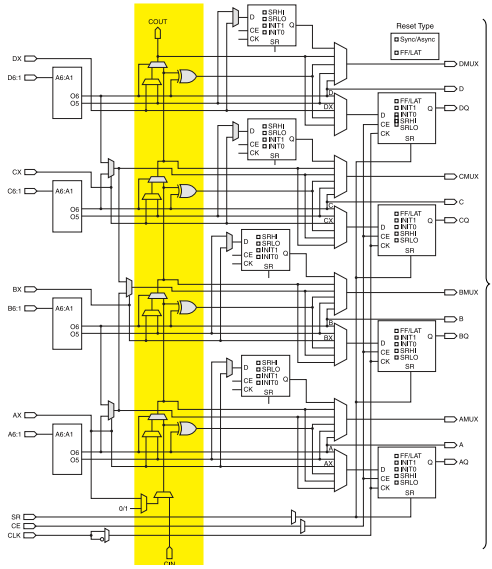
- See the LUT?
- ... the register?



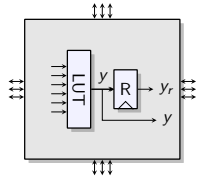

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)
  - 2 slices/CLB (and counting)
  - Ratio reg/LUT still equal to 1

All this keeps routing local

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block




A "slice" (Virtex7)



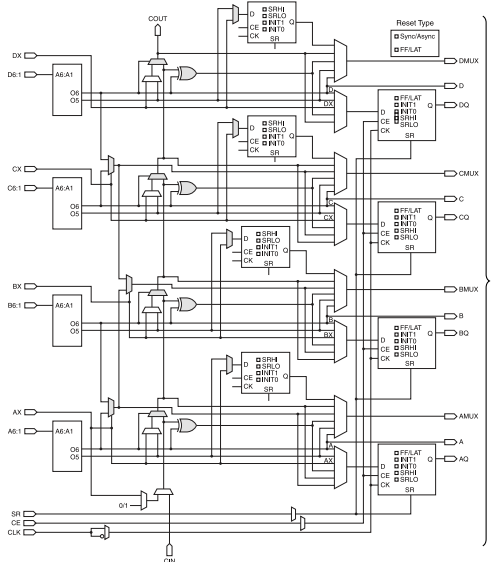

- See the LUT?
- ... the register?

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)
  - 2 slices/CLB (and counting)
  - Ratio reg/LUT still equal to 1

  All this keeps routing local
- Support of frequent operations
  - addition: **carry logic**

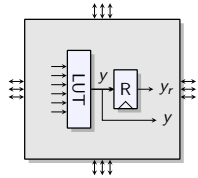
    (skips the slow routing)

# Back to Earth: The real ~~Xilinx~~ AMD Configurable Logic Block




A "slice" (Virtex7)

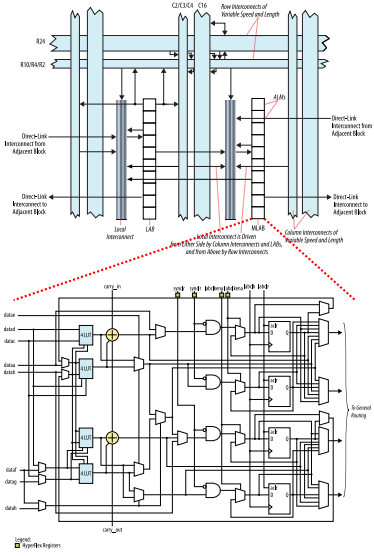
- See the LUT?
- ... the register?



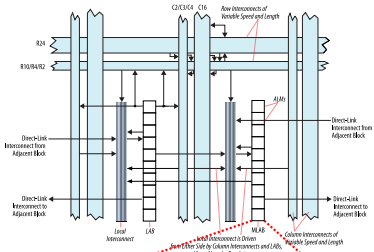

- Granularity increasing
  - 6-input LUTs (and counting)
  - 4 LUT/slice (and counting)
  - 2 slices/CLB (and counting)
  - Ratio reg/LUT still equal to 1

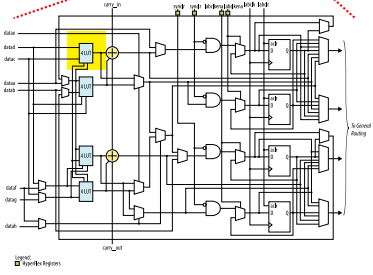
  All this keeps routing local
- Support of frequent operations
  - addition: **carry logic** (skips the slow routing)
  - shift registers (SRL)
  - etc...

# WRKB: The real ~~Altera~~ Intel Logic Array Block



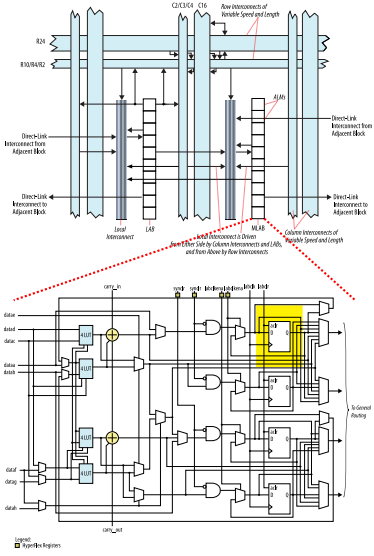
# WRKB: The real ~~Altera~~ Intel Logic Array Block



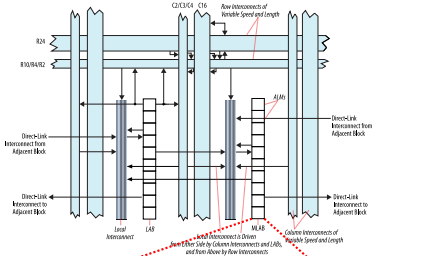

- You still see the LUTs (4 inputs/LUT)

# WRKB: The real ~~Altera~~ Intel Logic Array Block



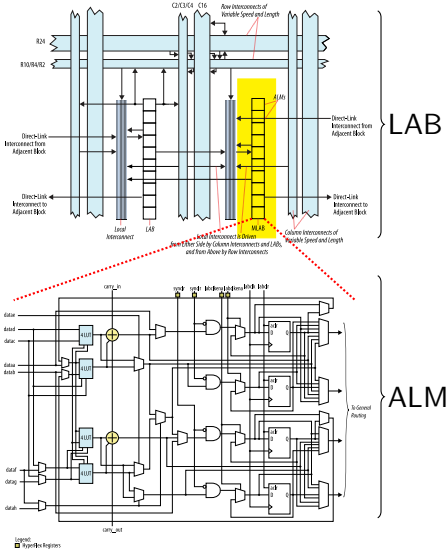

- You still see the LUTs (4 inputs/LUT)
- and the registers

# WRKB: The real ~~Altera~~ Intel Logic Array Block



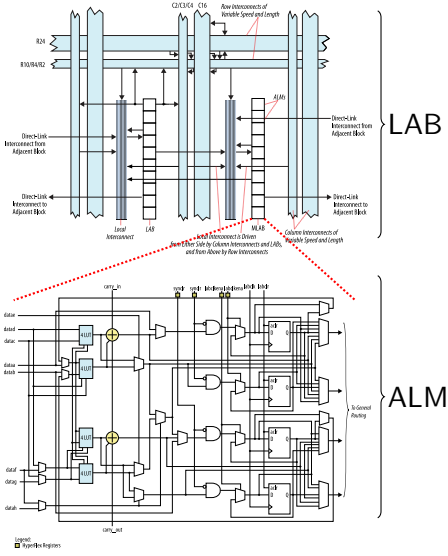

- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)

# WRKB: The real ~~Altera~~ Intel Logic Array Block




- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)
  - 10 ALM/LAB (logic array block)

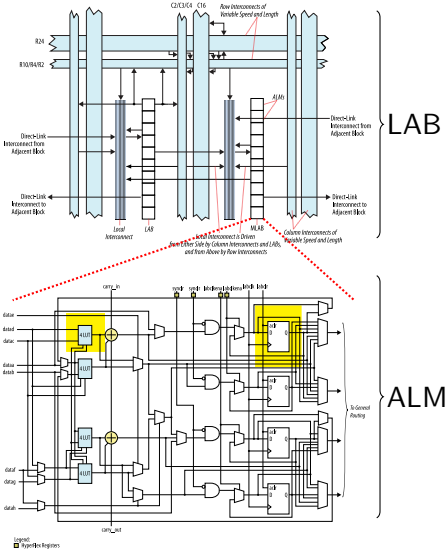
# WRKB: The real ~~Altera~~ Intel Logic Array Block




- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)
  - 10 ALM/LAB (logic array block)

(and here you see the routing!)

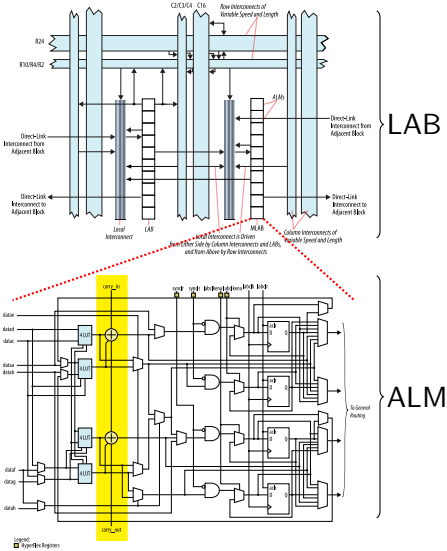
# WRKB: The real ~~Altera~~ Intel Logic Array Block




- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)
  - 10 ALM/LAB (logic array block)

  (and here you see the routing!)
- ratio LUT/reg still 1

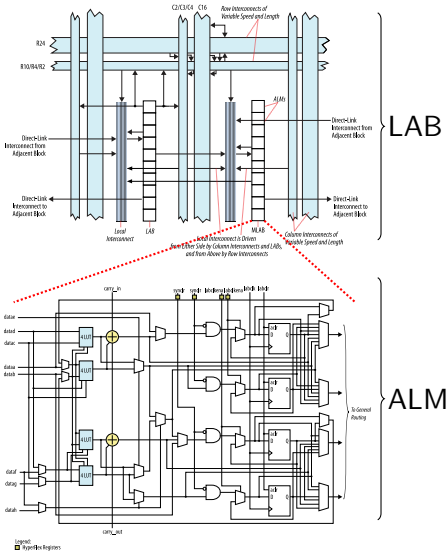
# WRKB: The real ~~Altera~~ Intel Logic Array Block




- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)
  - 10 ALM/LAB (logic array block)

  (and here you see the routing!)
- ratio LUT/reg still 1
- specific addition logic.

# WRKB: The real ~~Altera~~ Intel Logic Array Block

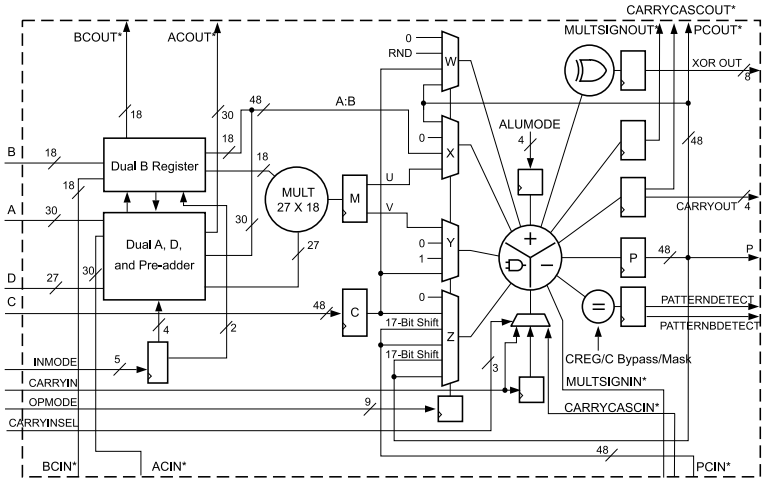



- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
  - 4 LUT/ALM (adaptive logic module)
  - 10 ALM/LAB (logic array block)

  (and here you see the routing!)
- ratio LUT/reg still 1
- specific addition logic.

Two teams solving the same problems with mutual patent avoidance

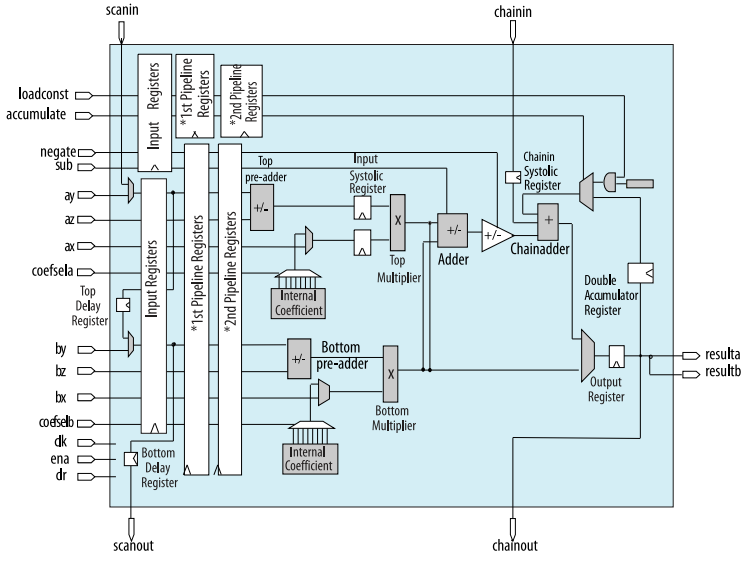
# The ~~Xilinx~~ AMD configurable DSP block




A multiplier with pre-adders and post-adders (for complex mult, symmetric FIR filters, etc.)

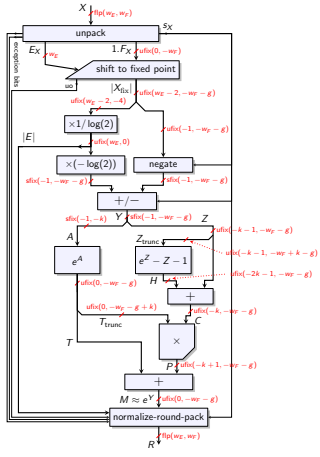
# The ~~Altera~~ Intel variable-precision DSP block

same comment
(with 2 multipliers
for complex arithmetic
etc.)



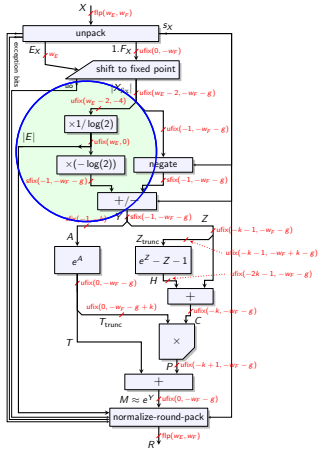
# Some opportunities of hardware computing just right

# Opportunity #1: Over-parameterization



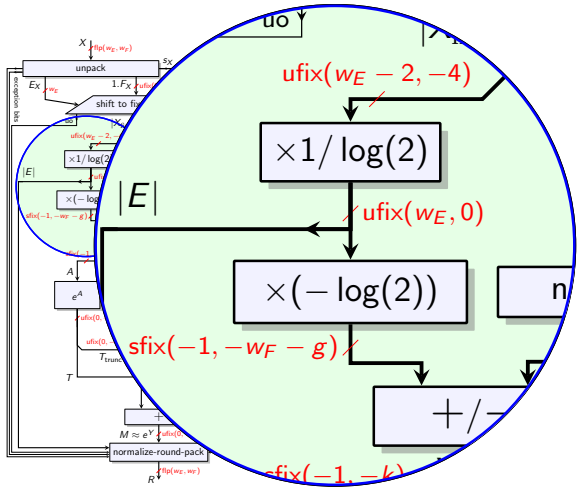

Example:

# Opportunity #1: Over-parameterization



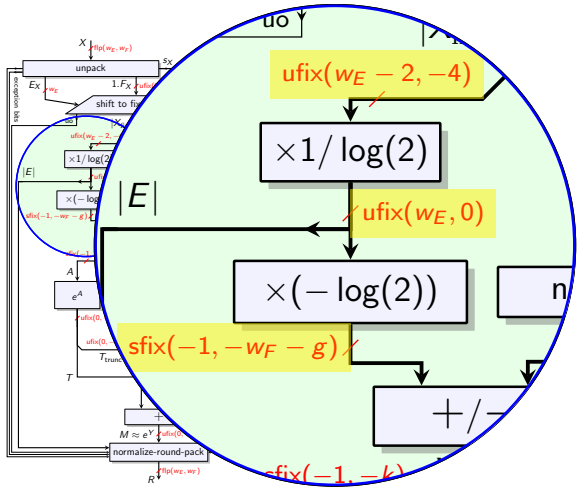

Example:

# Opportunity #1: Over-parameterization




Example:

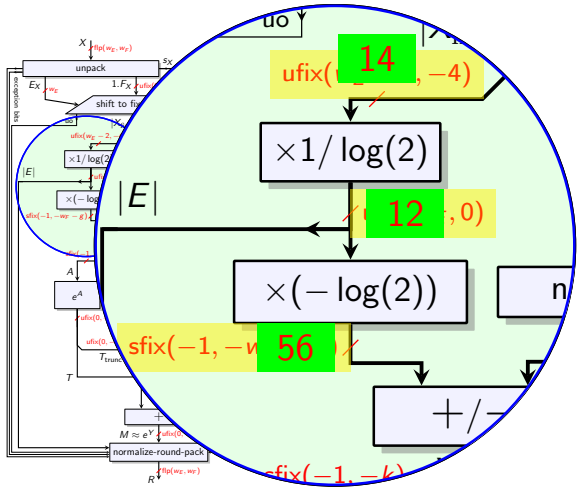
# Opportunity #1: Over-parameterization




Example:

**Multipliers of all shapes and sizes**

# Opportunity #1: Over-parameterization




Example:

**Multipliers of all shapes and sizes**

In a double-precision exponential,

- $w_E = 11$, $w_F = 52$,
- first multiplier 14-bits in, 12 bits out
- second multiplier 12-bits in, 56 bits out

... and truncated left and right

# Over-parameterization is cool

- OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program

# Over-parameterization is cool

- ⊖ OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program

- ⊕ Direct benefit to end-users: freedom of choice
  - People used to publish "An exponential architecture for single-precision", standard is now "A family of exponential architectures for each precision"
  - Application-specific optimal, future-proof, etc.

# Over-parameterization is cool

⊖ OK, there is a bit more work involved in designing a parametric operator
- To start with, it must be a hardware-generating program

⊕ Direct benefit to end-users: freedom of choice
- People used to publish "An exponential architecture for single-precision", standard is now "A family of exponential architectures for each precision"
- Application-specific optimal, future-proof, etc.

⊕ It actually simplifies design of composite operators (e.g. the exponential)
- No need to take any dramatic decision in the design phase:
  *You don't know how many bits on this wire make sense? Keep it open as a parameter.*
- Then estimate cost and accuracy as a function of the parameters
- Then find the optimal values of the parameters,
  e.g. using ILP or common sense (whichever gives the best results)

# Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*

# Opportunity #2: Operator specialization

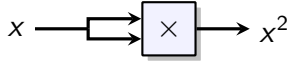
Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks

# Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks
- A squarer is a multiplier specialization

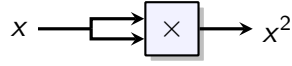



$$\begin{array}{rr} & 321 \\ \times & 321 \\ \hline & 321 \\ & 642 \\ & 963 \\ \hline & 103041 \end{array}$$

# Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks
- A squarer is a multiplier specialization

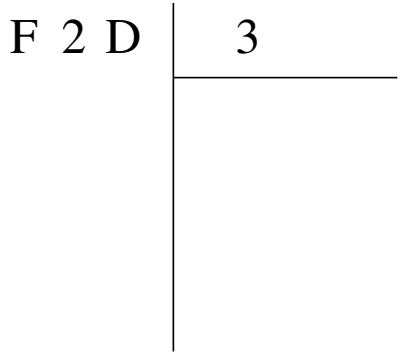



- Specialization of elementary functions to specific domains
- ...

$$\begin{array}{r} 321 \\ \times\ 321 \\ \hline 321 \\ 642 \\ 963 \\ \hline 103041 \end{array}$$

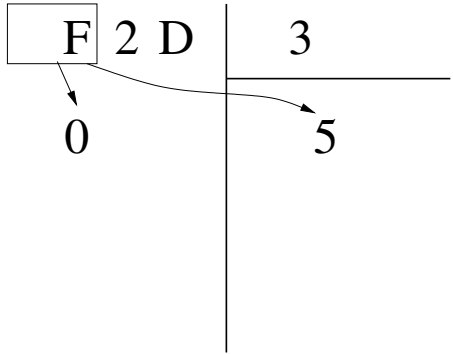
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



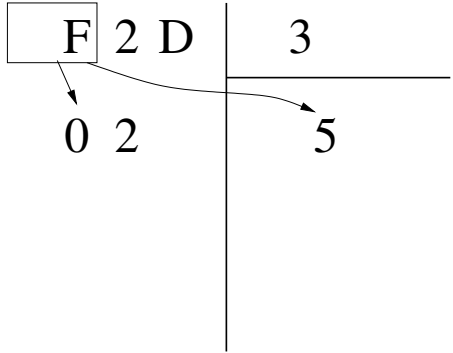
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



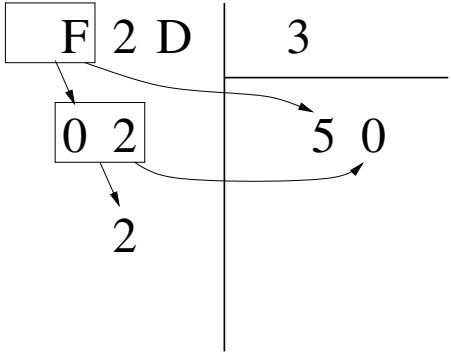
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



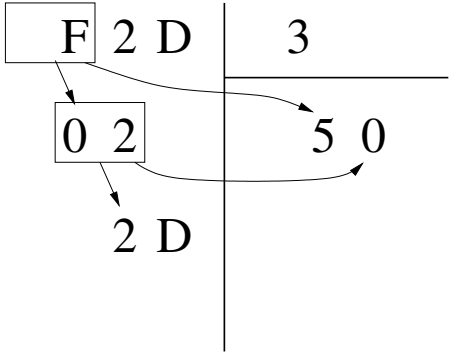
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



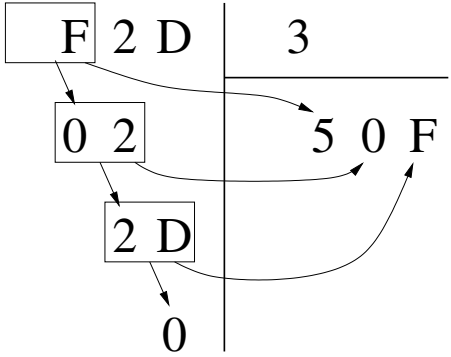
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

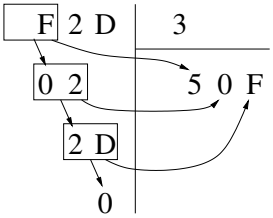


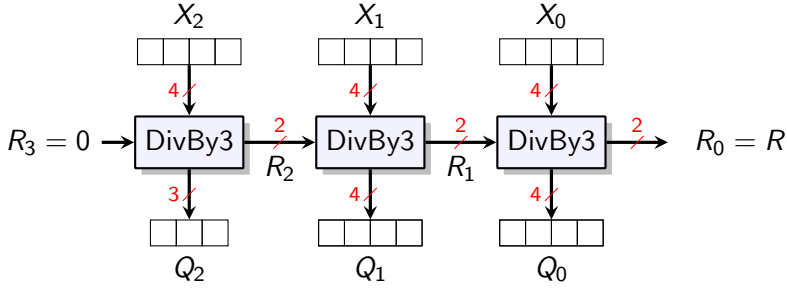
# Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

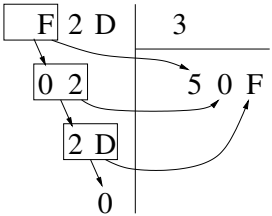


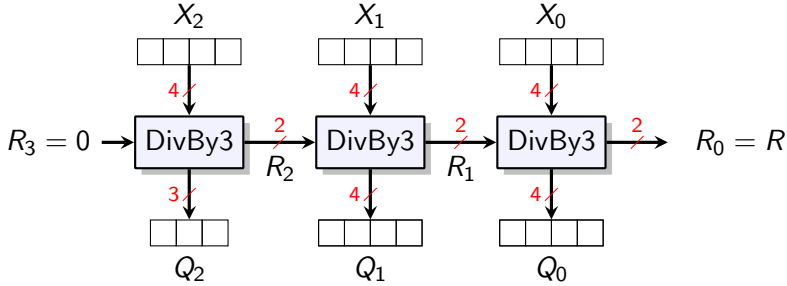
# Getting inspiration from the vexations of childhood



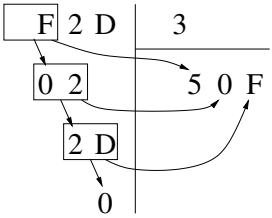
# Getting inspiration from the vexations of childhood

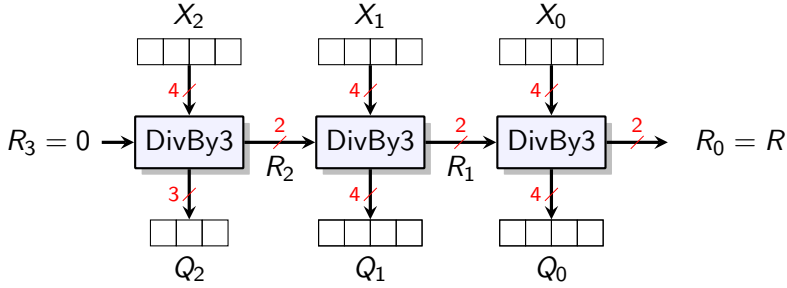



OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

# Getting inspiration from the vexations of childhood

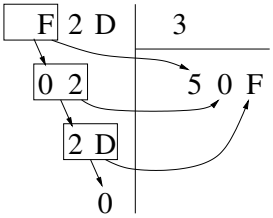


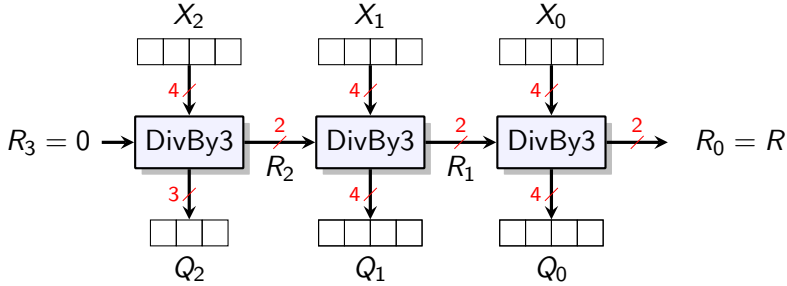




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

*Being unable to trust my reasoning, I learnt by heart the results of all the possible divisions*
(adapted from E. Ionesco)

# Getting inspiration from the vexations of childhood




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

*Being unable to trust my reasoning, I learnt by heart the results of all the possible divisions*
(adapted from E. Ionesco)

**If you're too lazy to compute, then tabulate**

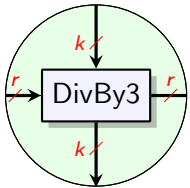
... here a table of $2^6$ entries of 6 bits each.

# What, my taxpayer money is wasted on studies of division by 3?

We did it for the fun of it, but it turns out to be useful for

- correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
- round-robin addressing with 3 banks of memory (need quotient and remainder)
- ...

# Opportunity #3: target-specific optimizations

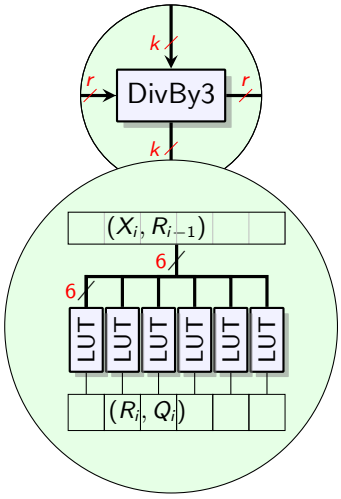



Generalizing hexadecimal to radix $2^k$

... or, how over-parameterization allows for adaptation

- to various values of 3, like $D = 5$, or 7, or 9

# Opportunity #3: target-specific optimizations




Generalizing hexadecimal to radix $2^k$

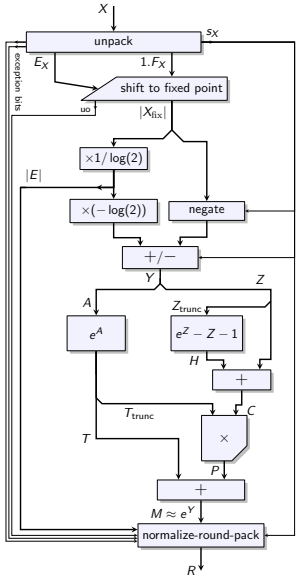
... or, how over-parameterization allows for adaptation

- to various values of 3, like $D = 5$, or 7, or 9
- to a given FPGA

**Perfect match to modern FPGAs**

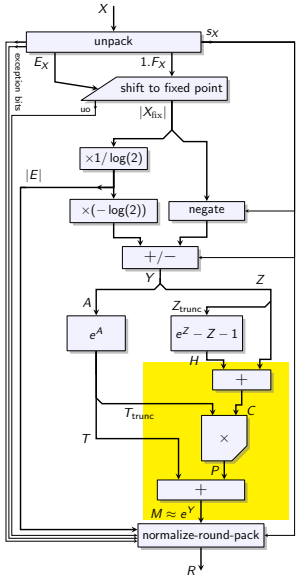
Unit of area: the LUT, with $\alpha$ input bits (here $\alpha = 6$)

# Opportunity #3: target-specific optimizations




Modern FPGAs also have

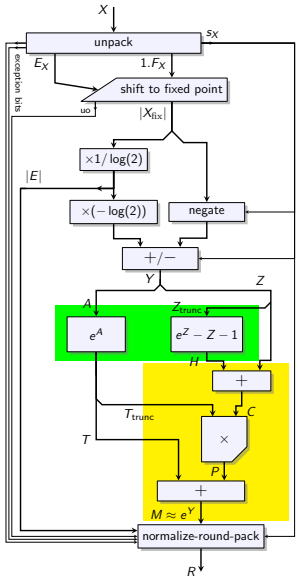
# Opportunity #3: target-specific optimizations




Modern FPGAs also have

- small multipliers with pre-adders and post-adders

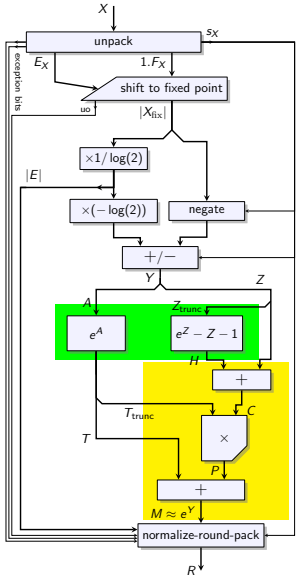
# Opportunity #3: target-specific optimizations




Modern FPGAs also have

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

# Opportunity #3: target-specific optimizations




Modern FPGAs also have

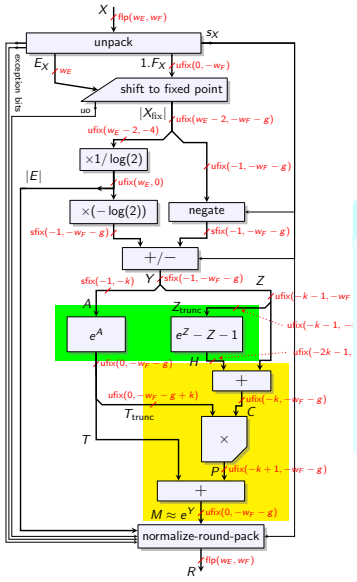
- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

**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)

# Opportunity #3: target-specific optimizations




Modern FPGAs also have

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

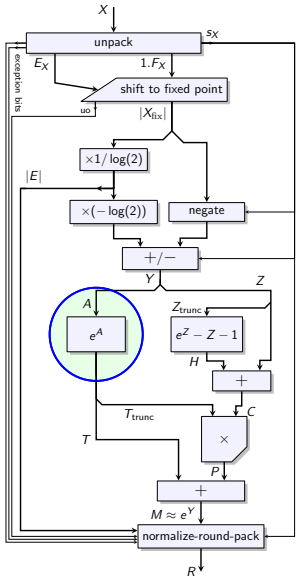
## Single-precision accurate exponential on Xilinx

- 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)*

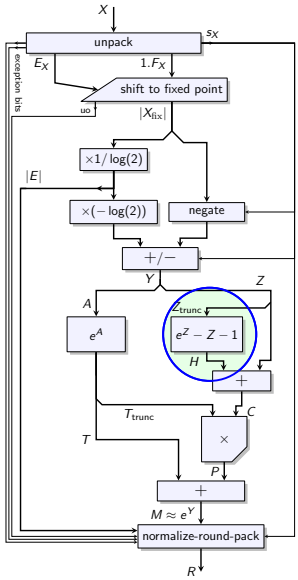
# Opportunity #4: Tabulation




*Being unable to trust my reasoning, I learnt by heart*
*the results of all the possible multiplications*
(E. Ionesco)

- ... and all the possible exponentials

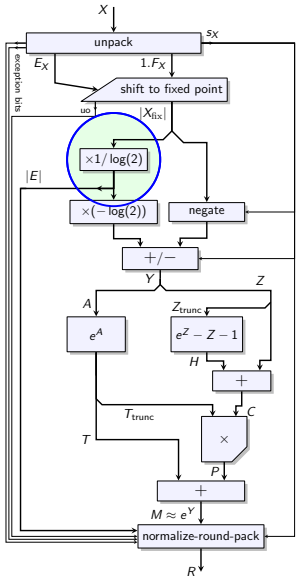
# Opportunity #4: Tabulation




*Being unable to trust my reasoning, I learnt by heart*
*the results of all the possible multiplications*
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$

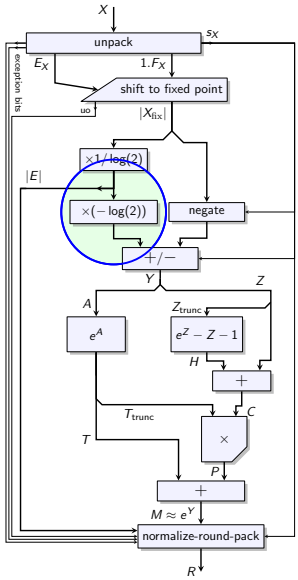
# Opportunity #4: Tabulation




*Being unable to trust my reasoning, I learnt by heart*
*the results of all the possible multiplications*
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

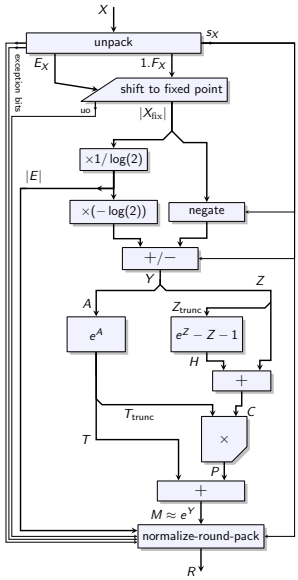
# Opportunity #4: Tabulation




*Being unable to trust my reasoning, I learnt by heart*
*the results of all the possible multiplications*
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

# Opportunity #4: Tabulation

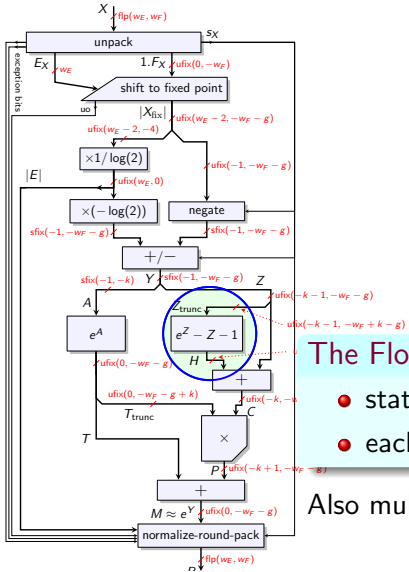


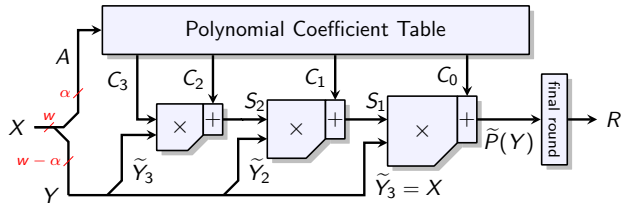

*Being unable to trust my reasoning, I learnt by heart*
*the results of all the possible multiplications*
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

Reading a tabulated value is very efficient
when the table is close to the consumer.

# Opportunity #5: Generic approximators (when tabulation won't scale)



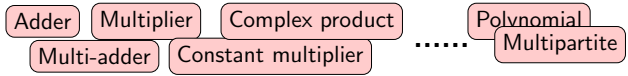




## The FloPoCo FixFunctionByPiecewisePoly operator

- state-of-the-art polynomial approximation
- each multiplier tailored with love and care

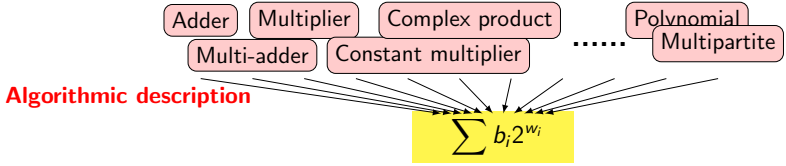
Also multipartite tables, filter approximators, and more to come.

# Opportunity #6: merged arithmetic in bit heaps



# Opportunity #6: merged arithmetic in bit heaps

One data-structure to rule them all...

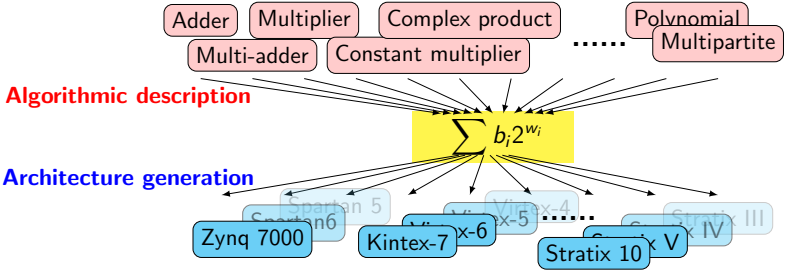



## The **sum of weighted bits** as a first-class arithmetic object

- A very wide class of operators: multi-valued polynomials, and more
- Captures the true bit-level parallelism, enables bit-level optimization opportunities

# Opportunity #6: merged arithmetic in bit heaps

One data-structure to rule them all... and in the hardware to bind them

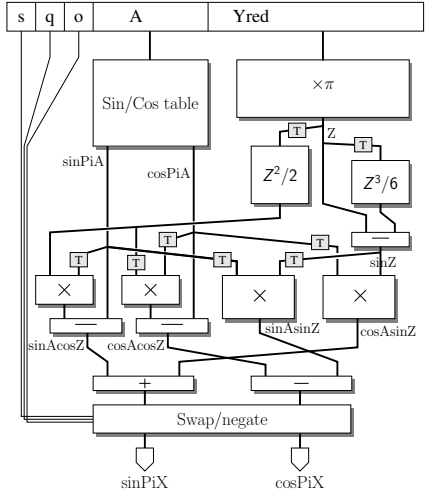



The **sum of weighted bits** as a first-class arithmetic object

- A very wide class of operators: multi-valued polynomials, and more
- Captures the true bit-level parallelism, enables bit-level optimization opportunities
- **Bit-array compressor trees** can be optimized for each target
  ... and optimally so for practical sizes, thanks to M. Kumm

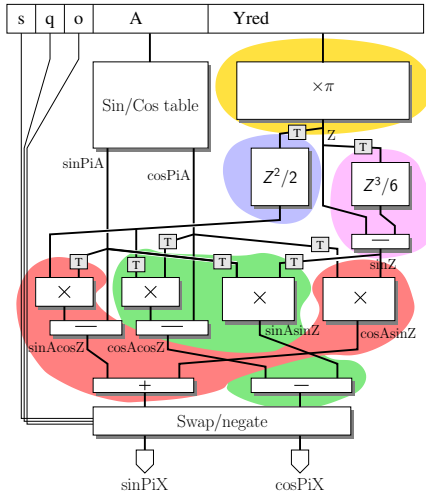
# When you have a good hammer, you see nails everywhere

A sine/cosine architecture (Iştoan, HEART 2013):

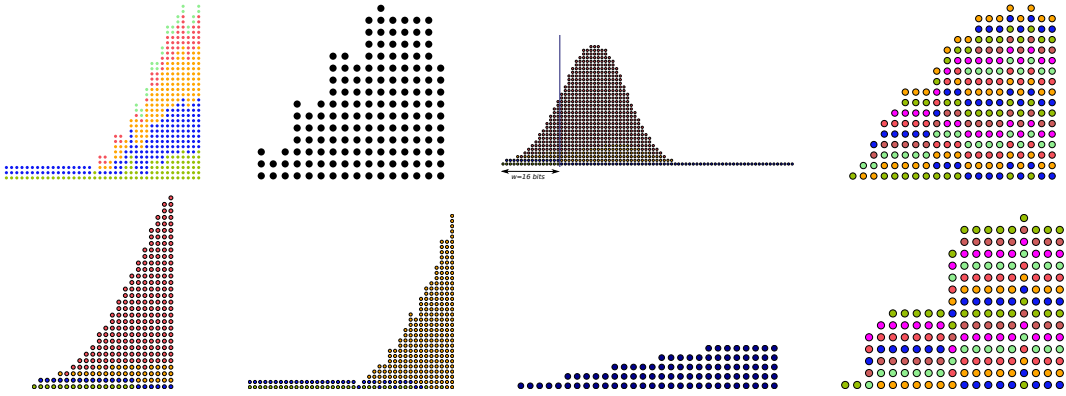


# When you have a good hammer, you see nails everywhere

A sine/cosine architecture (Iştoan, HEART 2013): 5 bit heaps



# Bit heaps for some operators and filters




Why are some people still insisting I should call these “bit arrays”?

# A few FPGA success stories

# Summary so far

How **not** to do scientific computing on FPGAs

- In 2007, an Intel team proudly demonstrates
  their 1994 flagship processor in a single FPGA
- and it runs at 25MHz (1/3rd of the 1994 frequency).
- It boots to Linux (terminal only) in 10mn

It will probably *not* accelerate your scientific code.

# Summary so far

How **not** to do scientific computing on FPGAs

- In 2007, an Intel team proudly demonstrates
  their 1994 flagship processor in a single FPGA
- and it runs at 25MHz (1/3rd of the 1994 frequency).
- It boots to Linux (terminal only) in 10mn

It will probably *not* accelerate your scientific code.

... jokes aside, this is a really nice paper.

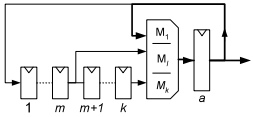
Shih-Lien L. Lu, Peter Yiannacouras, Taeweon Suh, Rolf Kassa, Michael Konow.
An FPGA-Based Pentium® in a Complete Desktop System
In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.

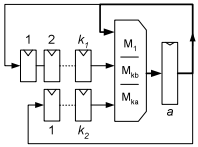
- A snapshot from Xilinx' HPC page (when it was still Xilinx):


©Xilinx

- You find the same keywords on Intel FPGA's pages
- and on Maxeler Technologies' (a company providing computation acceleration service)
- among others ...

# Monte Carlo simulation




- uniform random bits are cheap as chips on FPGAs
  - LFSRs are a CPU thing
  - generalize them to several parallel shift registers
  - several random bits in parallel from a single state
  - high-quality randomness if you get the math right

David B. Thomas and Wayne Luk.
FPGA-Optimised High-Quality Uniform Random Number Generators
In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2008.

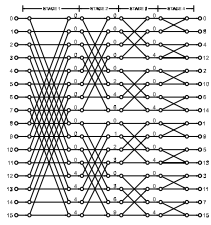
- FPGAs also much better at non-uniform (Gaussian etc) than CPUs or GPUs

David B. Thomas, Lee Howes, and Wayne Luk.
A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation
In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2009.

All this was developped for antiscientific computing (high-speed trading), but still...

# Signal processing

Mario Garrido, Konrad Möller, and Martin Kumm.
World's Fastest FFT Architectures: Breaking the Barrier of 100 GS/s.
*IEEE Transactions on Circuits and Systems I*, 66(4):1507–1516, 2019.



- Fully unrolled FFT (up to 256 points)
  - i.e. inputting 256 complex values per cycle, at 500 MHz
  - well above 10 TOp/s if you count all additions and multiplications
- 16-bit in/out, wider datapath inside
- Look, Ma: no multiplier !
  - each multiplier expanded as an adder graph (and optimally so)
  - ... leaving the 1800 DSP blocks free for other things.
- about 1/5th of LUT + registers of the target device (Virtex UltraScale 190)

*As previously, a good start is not to imitate the processor solution.*

# Artificial intelligence and machine learning

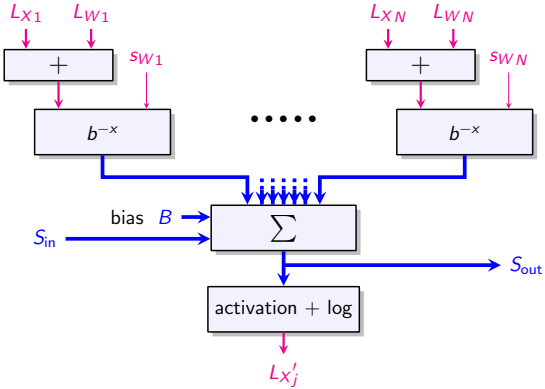
So many papers these days, here is one extreme:

Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot.
High-efficiency convolutional ternary neural networks with custom adder trees and weight compression.
*ACM Transactions on Reconfigurable Technologies and Systems*, 11(3), dec 2018.

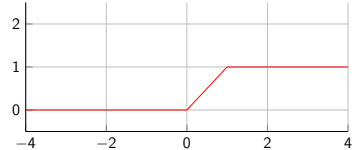
- Ternary logic: weight and activations $\in \{-1, 0, 1\}$
- Specific network retraining
- (and a few more layers to reach comparable accuracy)
- All the weights fit in the FPGA RAM blocks
- Embedded HPC on small FPGAs

# FPGA-oriented machine learning

Maxime Christ, Florent de Dinechin, and Frédéric Pétrot.
Low-precision logarithmic arithmetic for neural network accelerators
*Application-Specific Arrays and Processors, 2022.*



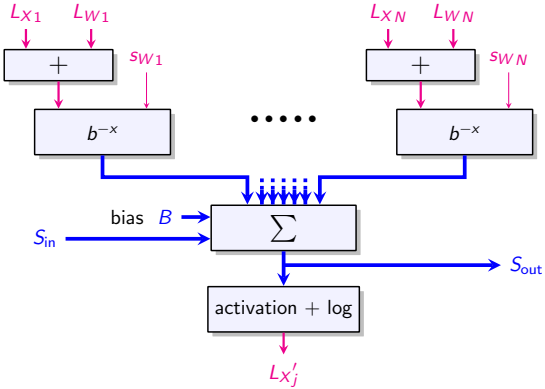

- Logarithmic data on 4 to 6 bits
- Tabulated $b^{-x}$ and activation+log
- Every bit counts:
  - ReLU1 saves 2 sign bits



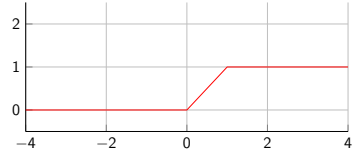

  - No need for special encoding of 0,

# FPGA-oriented machine learning

Maxime Christ, Florent de Dinechin, and Frédéric Pétrot.
Low-precision logarithmic arithmetic for neural network accelerators
*Application-Specific Arrays and Processors, 2022.*




- Logarithmic data on 4 to 6 bits
- Tabulated $b^{-x}$ and activation+log
- Every bit counts:
  - ReLU1 saves 2 sign bits




  - No need for special encoding of 0,

# Conclusion

A quote from the random generation paper (comparing CPU, GPU and FPGA):

The surprising result is that
each platform requires a different approach to random number generation

A quote from the random generation paper (comparing CPU, GPU and FPGA):

The ~~surprising~~ result is that
each platform requires a different approach to random number generation

I hope you appreciate now that it is not surprising.

A quote from the random generation paper (comparing CPU, GPU and FPGA):
$\forall X$ The ~~surprising~~ result is that
each platform requires a different approach to $X$

I hope you appreciate now that it is not surprising.

... and I invite you to suspect that this is true also for whatever you want to compute