

Saclay 2022

Computing with FPGAs ?

Florent de Dinechin



Outline

Introduction: the war of the programming models

FPGA architectures

Programming FPGAs

A few FPGA success stories

Conclusion

Introduction: the war of the programming models

Introduction: the war of the programming models

FPGA architectures

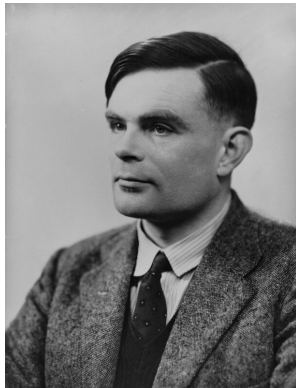
Programming FPGAs

A few FPGA success stories

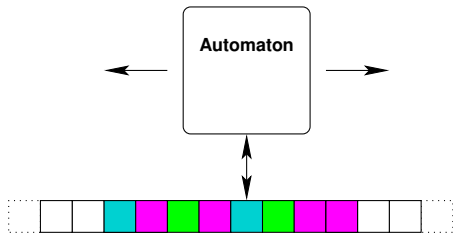
Conclusion

Alan Turing and John von Neumann

... *the twin gods of the computing pantheon* (A. C. Clarke in 2001: *A Space Odyssey*)

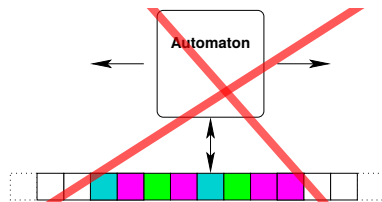
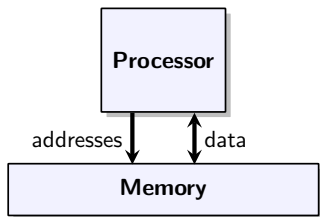


The Turing Machine: a good idea, but a commercial failure



- Defines universality
- Finite automaton infinite memory
- local access to the memory

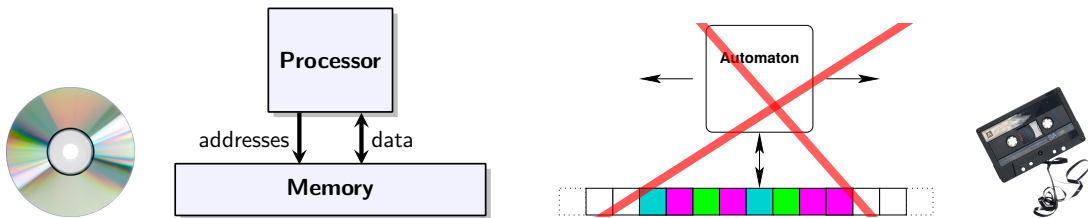
The von Neumann Machine that ruled



- Universal as well
- Same finite automaton (processor), same infinite¹ memory
- Turing-killer feature: **random** access to the memory

¹From there on, infinite means: some power of two, e.g. 2^{32} so large that I can't count that far.

The von Neumann Machine that ruled



- Universal as well
- Same finite automaton (processor), same infinite¹ memory
- Turing-killer feature: **random** access to the memory
 - so much more efficient (no tape rewinding)
 - so much easier to exploit (program here, data there, etc)

¹From there on, infinite means: some power of two, e.g. 2^{32} so large that I can't count that far.

When reality kicks back

A law of nature

You can't move data

faster than the speed of light

If the memory is infinite,
some bits will be physically distant
and will therefore be accessed slowly.

When reality kicks back

A law of nature

You can't move data

faster than the speed of light

If the memory is infinite,
some bits will be physically distant
and will therefore be accessed slowly.

Random access in constant (fast) time
is but a **dream**
and the von Neumann model
is but a model...

When reality kicks back

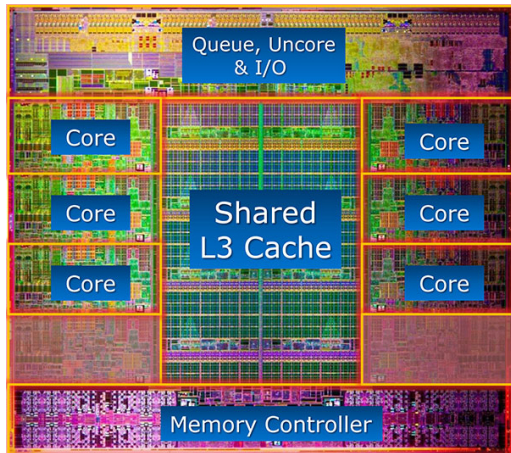
A law of nature

You can't move data

faster than the speed of light

If the memory is infinite,
some bits will be physically distant
and will therefore be accessed slowly.

Random access in constant (fast) time
is but a **dream**
and the von Neumann model
is but a model...

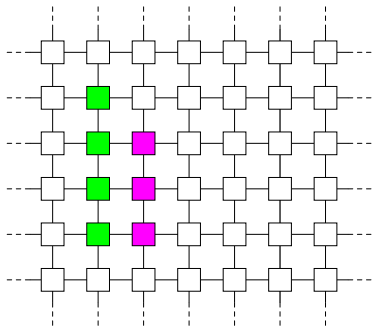


One half of your processor is there to keep this dream alive!

Meanwhile, von Neumann had a better idea

Cellular automaton: a spatial/parallel version of Turing machine.

(most famous instance: Conway's **Game of Life**)



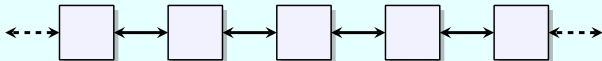
- an infinite number of automata
- all working in parallel
- with next-neighbour (local) communications
- universal all the same
(youtube "game of life in game of life")

Let us build this! (with a little help of Moore's law)

Yes but... next-neighbour communications suck!

The firing squad synchronization problem

Synchronize n cells, using next-neighbour communication only



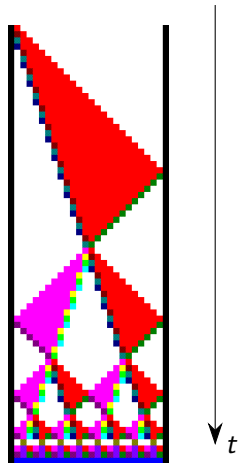
- right: $3n$ steps, using 15 states
- best known: $2n - 2$ steps, 6 states



Jacques Mazoyer.

A six-state minimal time solution to the firing squad synchronization problem. *Theoretical Computer Science*, 50(2):183–238, 1987.

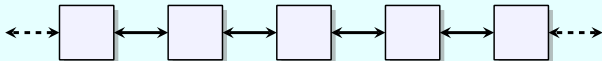
2D version: youtube “game of life in game of life”



Yes but... next-neighbour communications suck!

The firing squad synchronization problem

Synchronize n cells, using next-neighbour communication only



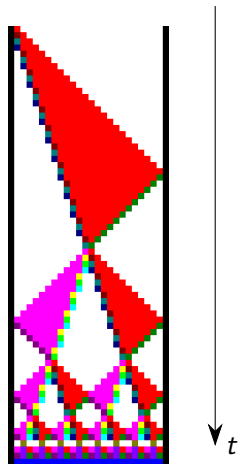
- right: $3n$ steps, using 15 states
- best known: $2n - 2$ steps, 6 states



Jacques Mazoyer.

A six-state minimal time solution to the firing squad synchronization problem. *Theoretical Computer Science*, 50(2):183–238, 1987.

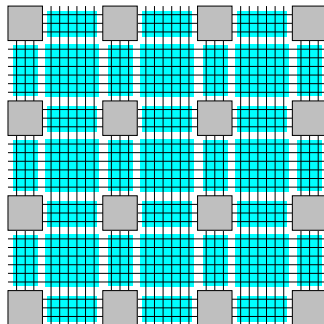
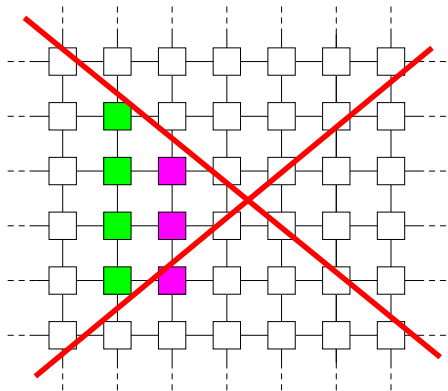
2D version: [youtube "game of life in game of life"](#)



... in real life the captain simply shouts "Fire!" (a **global** communication)

Field-Programmable Gate Arrays

FPGAs are to 2D cellular automata what von Neuman machines are to Turing machines:
something useful in practice.



FPGA architectures

Introduction: the war of the programming models

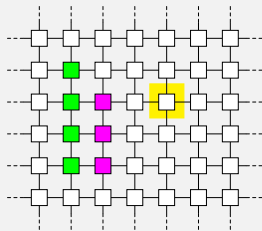
FPGA architectures

Programming FPGAs

A few FPGA success stories

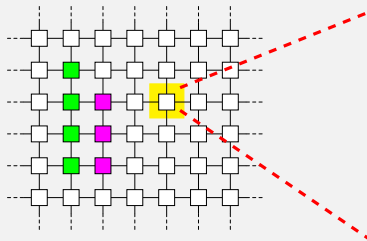
Conclusion

Overview



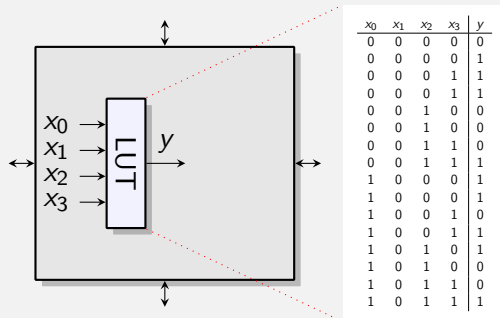
- Configurable logic cell
 - ... all what you need to build an automaton

Overview



- Configurable logic cell
 - ... all what you need to build an automaton

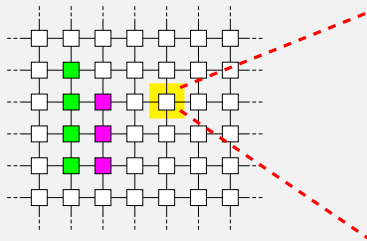
Inside a cell



- A Look-Up Table (LUT)
 - 4 inputs (for some value of 4),
 - 1 output
- ... may be filled with **any truth table**

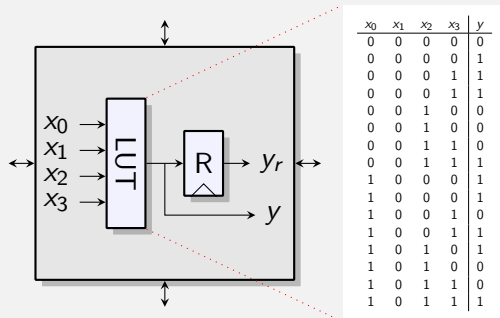
Basic FPGA structure

Overview



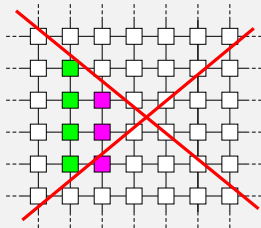
- Configurable logic cell
 - ... all what you need to build an automaton

Inside a cell



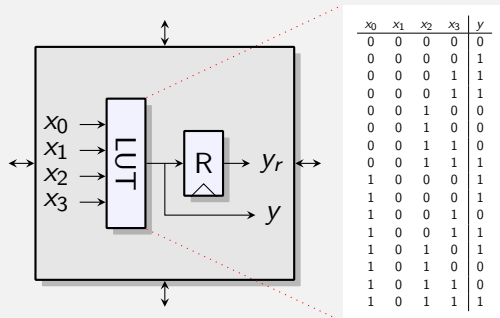
- A Look-Up Table (LUT)
 - 4 inputs (for some value of 4),
 - 1 output... may be filled with **any truth table**
- 1 bit of run-time memory R

Overview



- Configurable logic cell
 - ... all what you need to build an automaton
- Random access to distant cells

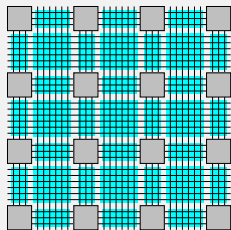
Inside a cell



- A Look-Up Table (LUT)
 - 4 inputs (for some value of 4),
 - 1 output... may be filled with **any truth table**
- 1 bit of run-time memory R

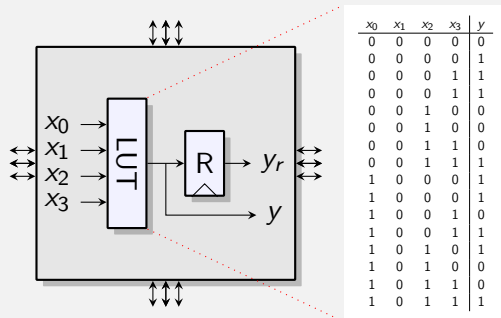
Basic FPGA structure

Overview



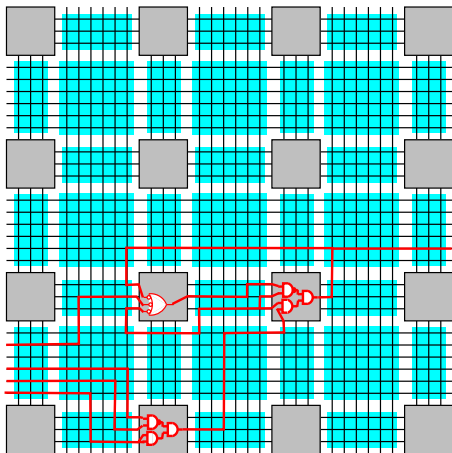
- Configurable logic cell
 - ... all what you need to build an automaton
- Random access to distant cells
 - routing channels
 - configurable switch boxes

Inside a cell



- A Look-Up Table (LUT)
 - 4 inputs (for some value of 4),
 - 1 output... may be filled with any truth table
- 1 bit of run-time memory R

A configured FPGA



Also known as **reconfigurable circuits**, used for **reconfigurable computing**

Two moments in the life of an FPGA

Configuration time (many ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

an FPGA program == a lot of configuration bits

Two moments in the life of an FPGA

Configuration time (many ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

an FPGA program == a lot of configuration bits

Run time (forever if needed)

- Data is processed by each LUT according to its truth table
- Data moves from LUT to LUT along the (static) connexions
- The FPGA behaves as a circuit of gates

Two moments in the life of an FPGA

Configuration time (many ms)

- the LUTs are filled with truth tables
- the switching state (on/off) of each switch in each switch boxes is defined

an FPGA program == a lot of configuration bits

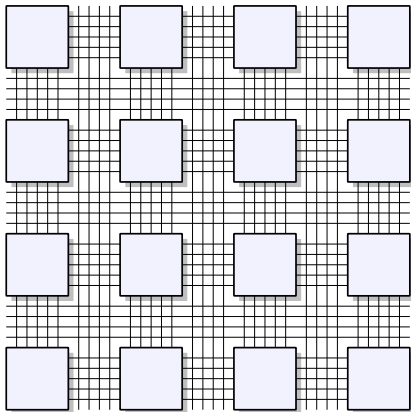
Run time (forever if needed)

- Data is processed by each LUT according to its truth table
- Data moves from LUT to LUT along the (static) connexions
- The FPGA behaves as a circuit of gates

The programming model of FPGAs is the digital circuit

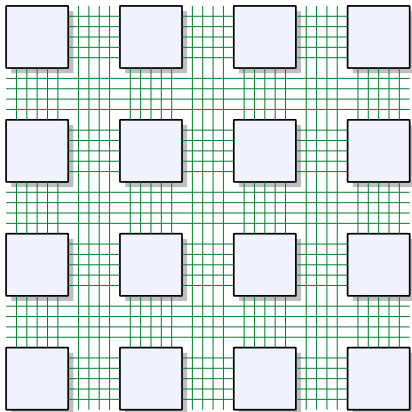
When reality kicks back

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



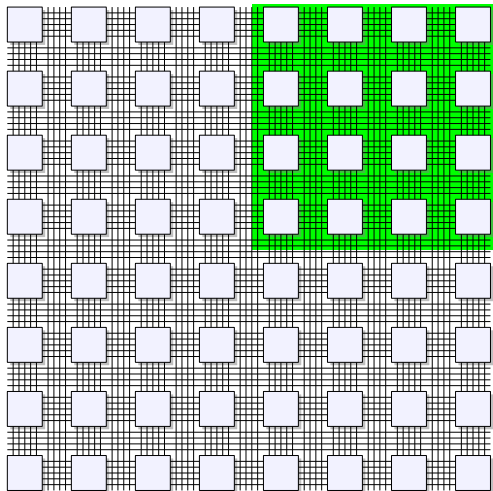
When reality kicks back

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



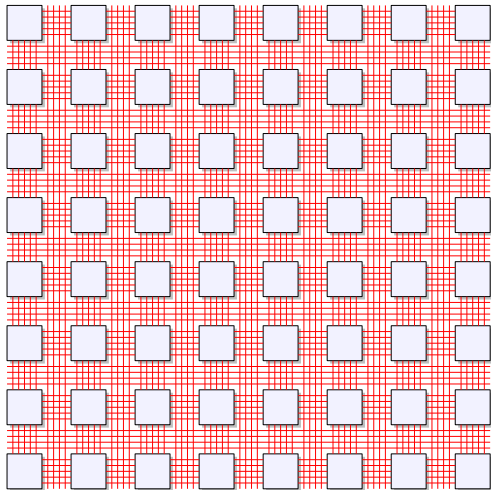
When reality kicks back

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



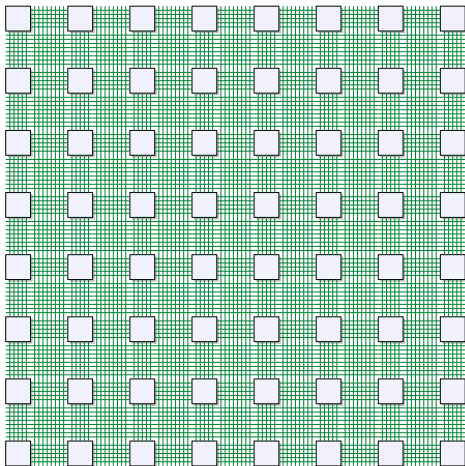
When reality kicks back

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



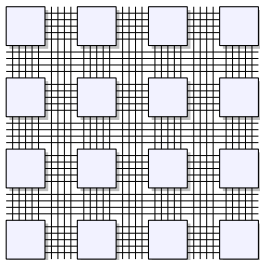
When reality kicks back

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



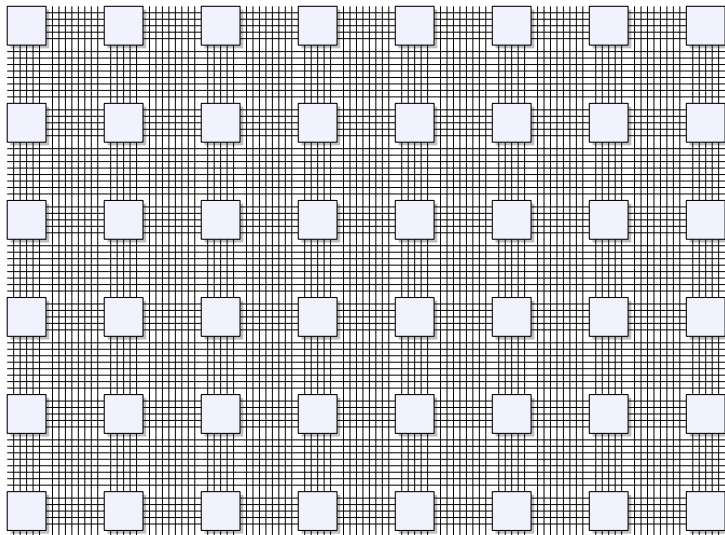
When reality kicks back

1990:



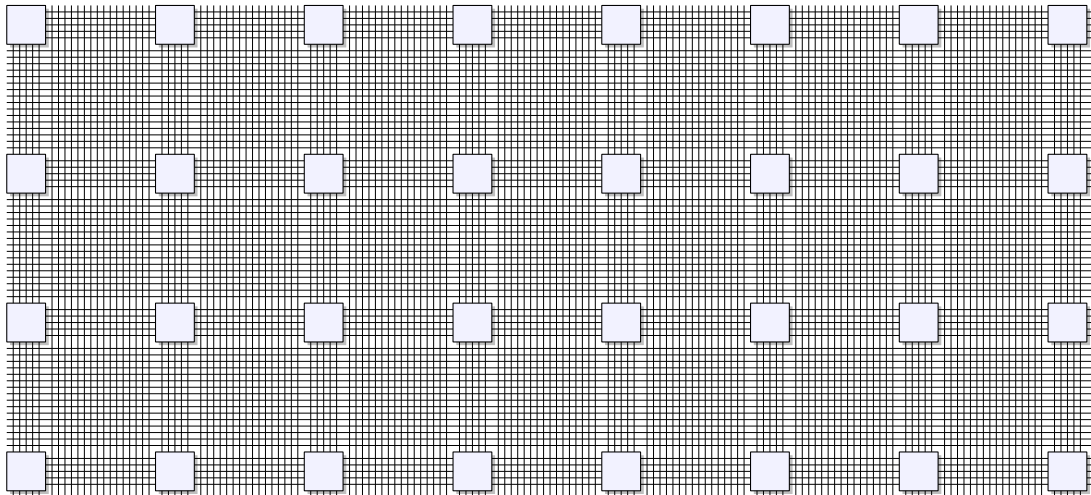
When reality kicks back

1994:



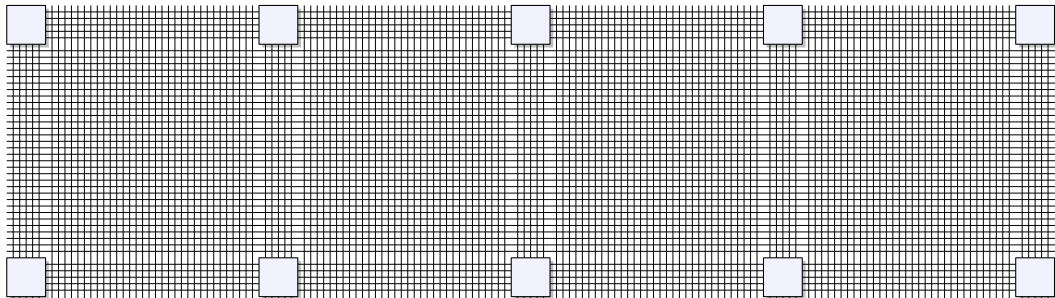
When reality kicks back

1996:



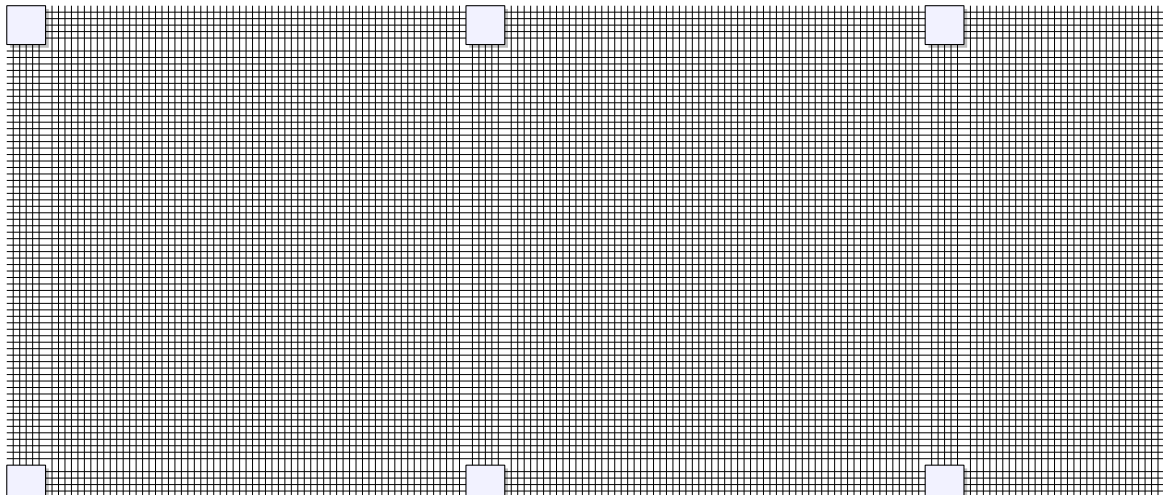
When reality kicks back

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 → 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.

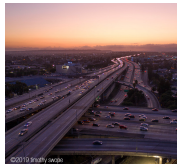
Two examples of this process at work:

- **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)

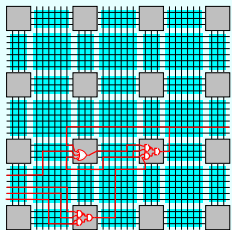
- **Lyon**

- in the 70s: planned destruction of a Renaissance area
to make space for a highway
- conservatism, opposition to progress → project cancellation
- now: a car-free area, and UNESCO world heritage



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

Yet another experimental law



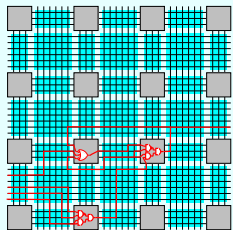
*In a circuit of diameter n (gates),
the number of wires crossing a diameter
is proportional to n^r with $1 < r < 2$.*

- more than proportional to n , the diameter,
- note 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...

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,
- note 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.

In cities:

Build highways of various widths

Build busses, metro, tramway

Relocalize the economy

And for you, the user:

Use bicycles instead of SUVs

Transposed to FPGA:

heterogeneous routing

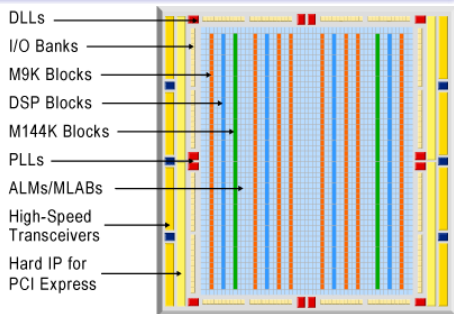
increase compute granularity

relocalize computations

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 (≈ 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 50 mm^2 FPGA...
- ... and the configured FPGA will run at 1/10th the frequency of the ASIC
 - there are transistors on all the wires!

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.

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

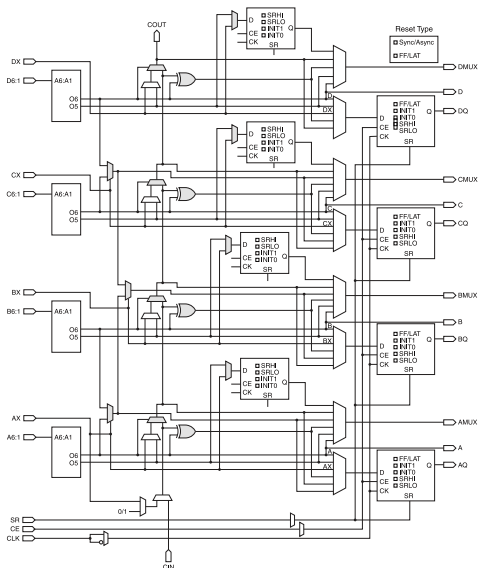
It will probably *not* accelerate your scientific code.

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

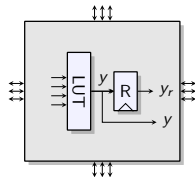


Shih-Lien L. Lu, Peter Yiannacouras, Taeweon Suh, Rolf Kassa, Michael Konow.
An FPGA-Based Pentium[®] in a Complete Desktop System
In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.

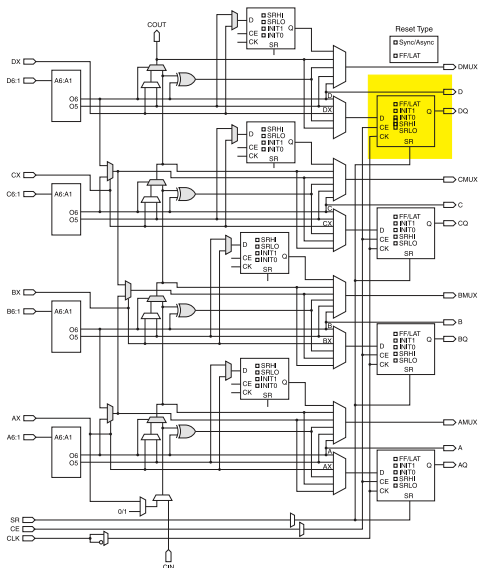
WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



A "slice" (Virtex7)

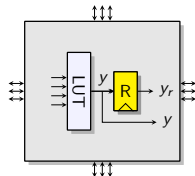


WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block

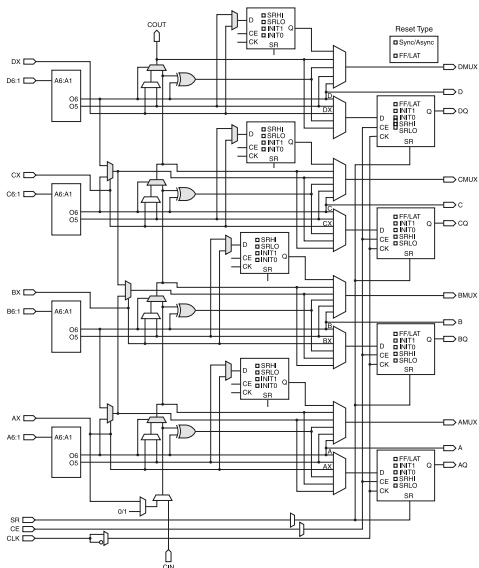


A "slice" (Virtex7)

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

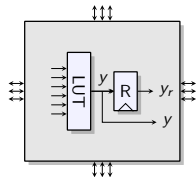


WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



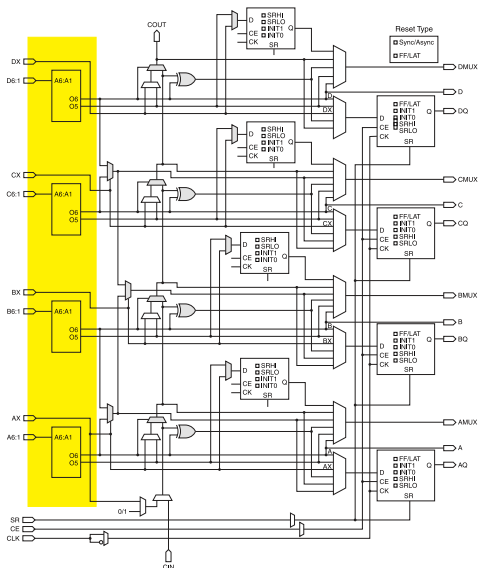
A “slice” (Virtex7)

- See the LUT?
- ... the register?
- Granularity increasing
 - 6-input LUTs



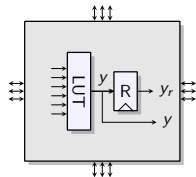
(and counting)

WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



A “slice” (Virtex7)

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

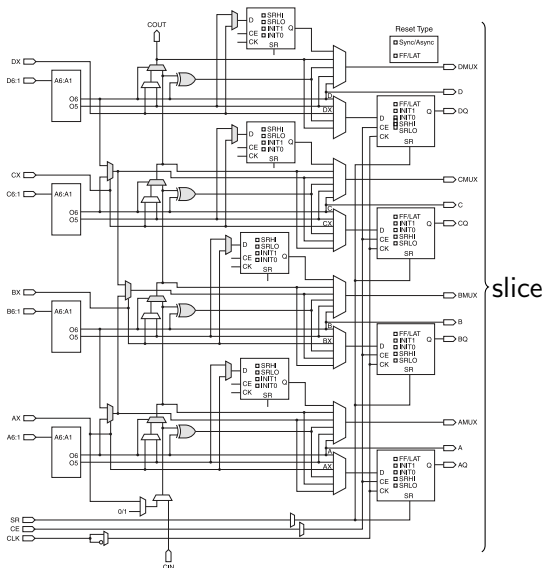


• Granularity increasing

- 6-input LUTs
- 4 LUT/slice

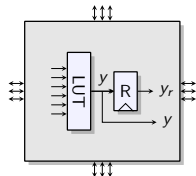
(and counting)
(and counting)

WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



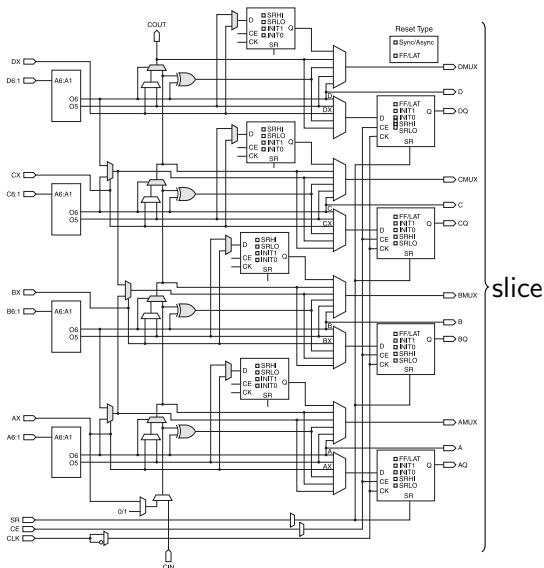
A "slice" (Virtex7)

- See the LUT?
- ... the register?
- Granularity increasing
 - 6-input LUTs
 - 4 LUT/slice
 - 2 slices/CLB



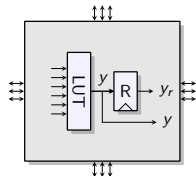
(and counting)
(and counting)

WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



A "slice" (Virtex7)

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

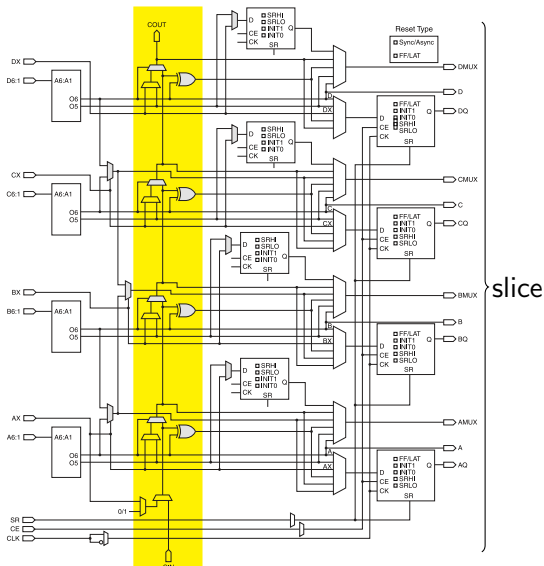


• Granularity increasing

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

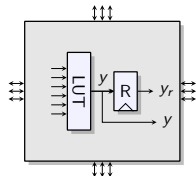
All this keeps routing local

WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



A “slice” (Virtex7)

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



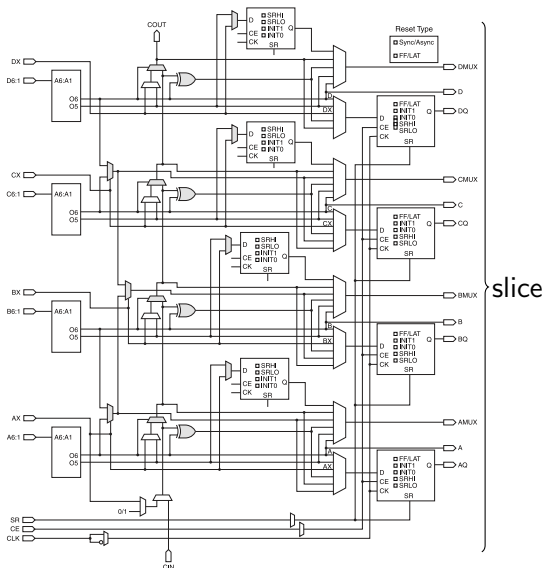
• Granularity increasing

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

All this keeps routing local

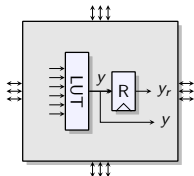
- Support of frequent ops
 - addition: **carry logic** (skips the slow routing)

WRKB: The real ~~Xilinx~~ AMD Configurable Logic Block



A "slice" (Virtex7)

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

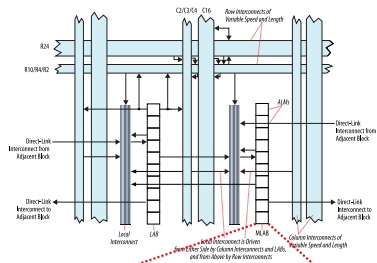


- Granularity increasing
 - 6-input LUTs (and counting)
 - 4 LUT/slice (and counting)
 - 2 slices/CLB
 - Ratio reg/LUT still equal to 1

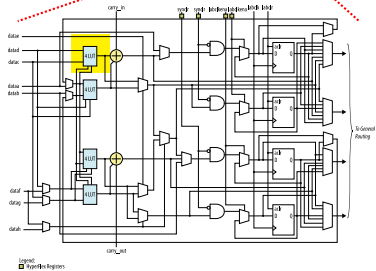
All this keeps routing local

- Support of frequent ops
 - addition: **carry logic** (skips the slow routing)
 - shift registers (SRL)
 - etc...

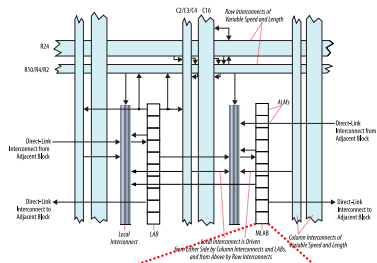
WRKB: The real ~~Altera~~ Intel Logic Array Block



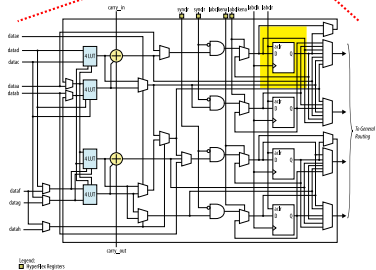
- You still see the LUTs (4 inputs/LUT)



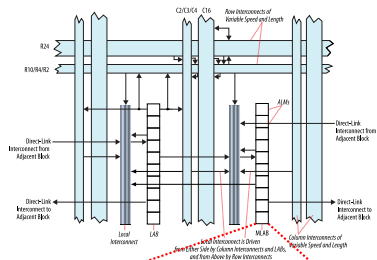
WRKB: The real ~~Altera~~ Intel Logic Array Block



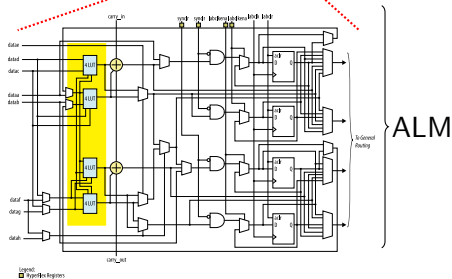
- You still see the LUTs (4 inputs/LUT)
- and the registers



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

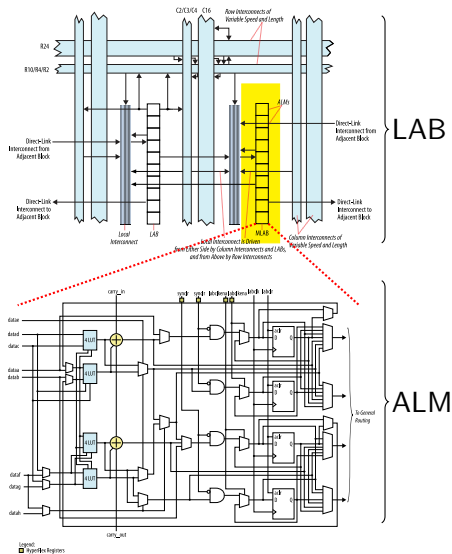


- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
 - 4 LUT/ALM (adaptive logic module)



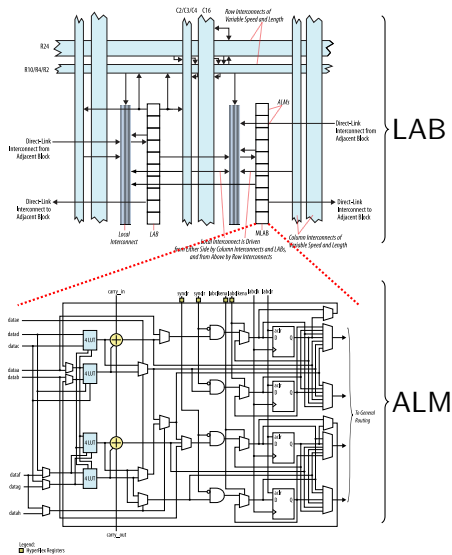
ALM

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



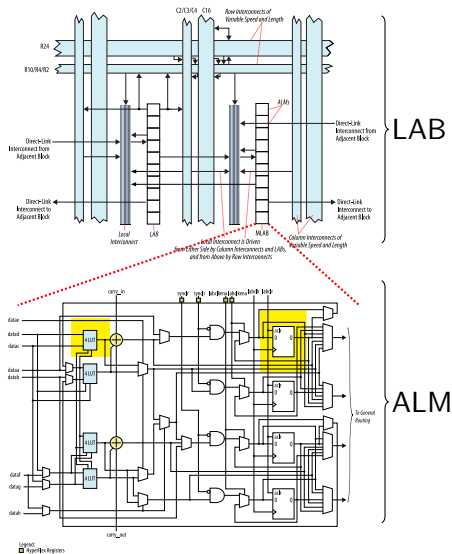
- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
 - 4 LUT/ALM (adaptive logic module)
 - 10 ALM/LAB (logic array block)

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



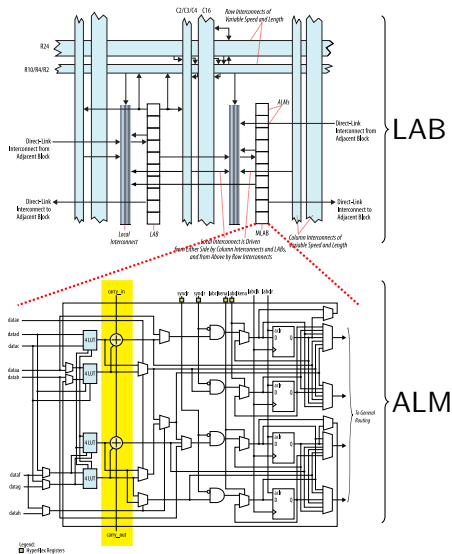
- You still see the LUTs (4 inputs/LUT)
 - and the registers
 - Granularity increasing
 - 4 LUT/ALM (adaptive logic module)
 - 10 ALM/LAB (logic array block)
- (and here you see the routing!)

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



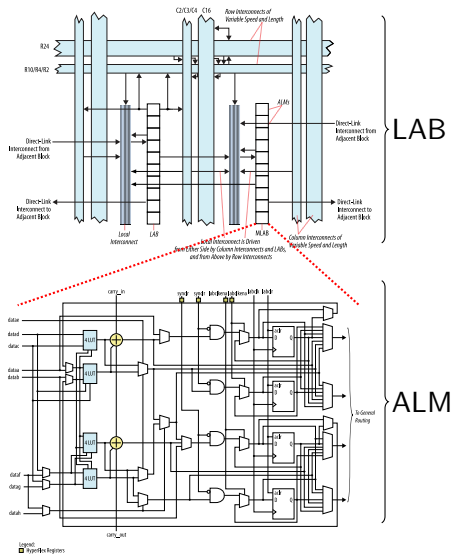
- You still see the LUTs (4 inputs/LUT)
- and the registers
- Granularity increasing
 - 4 LUT/ALM (adaptive logic module)
 - 10 ALM/LAB (logic array block)
 (and here you see the routing!)
- ratio LUT/reg still 1

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.

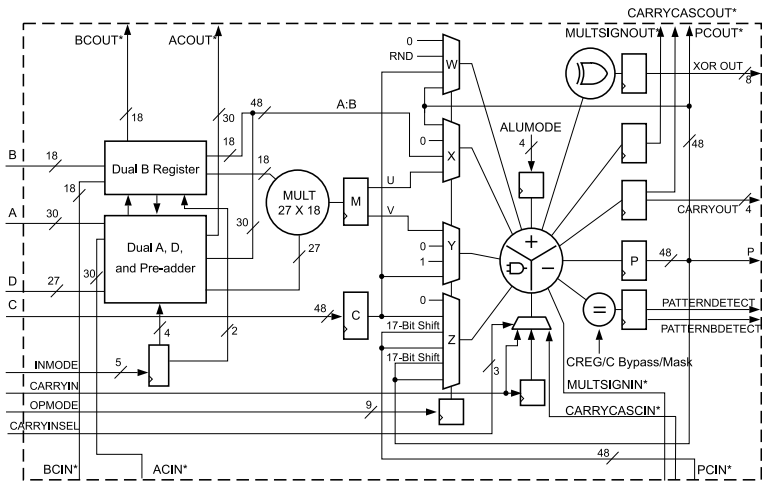
WRKB: The real ~~Altera~~ Intel Logic Array Block



- 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.

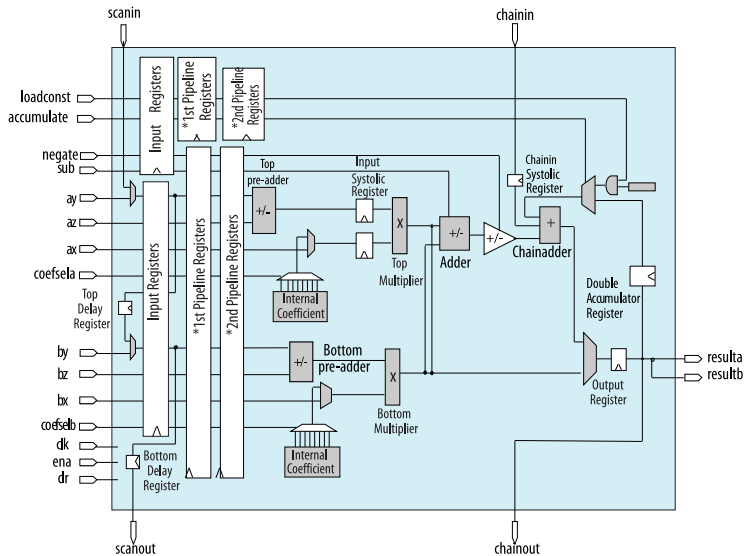
Two teams solving the same problems with mutual patent avoidance

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)

Programming FPGAs

Introduction: the war of the programming models

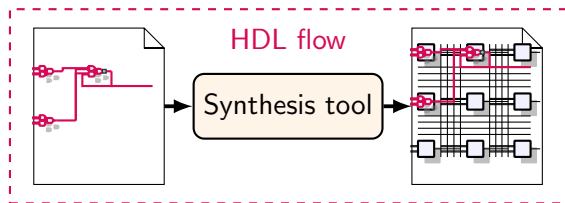
FPGA architectures

Programming FPGAs

A few FPGA success stories

Conclusion

Real Programmers needed

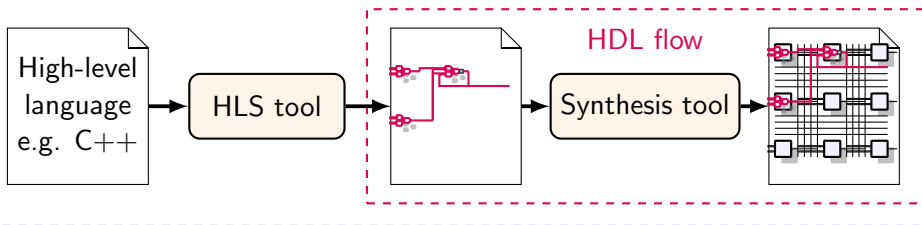


- You don't program, you design a circuit
 - in fancy Hardware Description Languages (HDL) such as VHDL or Verilog
 - with compilers called "synthesis tools" that can take hours:

1-week compilation time for a large FPGA...

Real Programmers needed

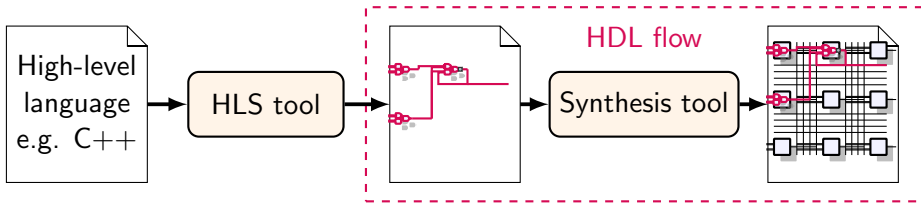
High-level synthesis (HLS) flow



- You don't program, you design a circuit
 - in fancy Hardware Description Languages (HDL) such as VHDL or Verilog
 - with compilers called "synthesis tools" that can take hours:
1-week compilation time for a large FPGA...
- Since the 2010, FPGA programming in C/C++ for the rest of us
 - higher productivity, but... no escaping the synthesis tools backend.

Real Programmers needed

High-level synthesis (HLS) flow



- You don't program, you design a circuit
 - in fancy Hardware Description Languages (HDL) such as VHDL or Verilog
 - with compilers called "synthesis tools" that can take hours:
 - **1-week compilation time for a large FPGA...**
- Since the 2010, FPGA programming in C/C++ for the rest of us
 - higher productivity, but... no escaping the synthesis tools backend.

Haha, C++ programming is called "high level" ...

If you have ever touched Cuda or OpenCL, you know what comes next

From a state of the art matrix-matrix product:



Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler.

Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis.

In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 244–254, 2020.

```
1 for (i = 0; i < M; i++)
2     for (j = 0; j < N; j++)
3         for (k = 0; k < K; k++)
4             C[i, j] = C[i, j] + A[i, k]*B[k, j];
```

Listing 1: Classical MMM algorithm.

```

1 // Memory tiles  $m$ 
2 for ( $i_m = 1; i_m \leq n; i_m = i_m + x_{tot}$ )
3   for ( $j_m = 1; j_m \leq m; j_m = j_m + y_{tot}$ )
4     for ( $k = 1; k \leq k; k = k + 1$ ) // Full dimension  $k$ 
5 // [Sequential] Block tiles  $b$  in memory tile
6   for ( $i_b = i_m; i_b \leq i_m + x_{tot}; i_m = i_m + x_t x_c x_p$ )
7     for ( $j_b = j_m; j_b \leq j_m + y_{tot}; j_m = j_m + y_t y_p y_c$ )
8 // [Sequential] Compute tiles  $t$  in block tile
9   for ( $i_t = i_b; i_t \leq i_b + x_t x_p x_c; i_b = i_b + x_c x_p$ )
10  for ( $j_t = j_b; j_t \leq j_b + y_t y_p y_c; j_t = j_t + y_c y_p$ )
11 // [Parallel] Processing elements  $p$  in compute tile
12   forall ( $i_p = i_t; i_p \leq i_t + x_p x_c; i_t = i_t + x_c$ )
13     forall ( $j_p = j_t; j_p \leq j_t + y_p y_c; j_p = j_p + y_c$ )
14 // [Parallel] Compute units  $c$  in processing element
15   forall ( $i_c = i_p; i_c \leq i_p + x_c; i_c = i_c + 1$ )
16     forall ( $j_c = j_p; j_c \leq j_p + y_c; j_c = j_c + 1$ )
17      $C(i_c, j_c) = C(i_c, j_c) + A(i_c, k) \cdot B(k, j_c)$ 

```

Listing 2: Pseudocode of the tiled MMM algorithm.


```

3
Compute.cpp
1 // ap_intrinsic::constexpr(kInnerTiles) m0 - 1;
2 // ap_intrinsic::constexpr(kComputeTilesH) m2 - 1;
3 // ap_intrinsic::constexpr(kInnerTiles) m1 - 1;
4 unsigned inner = 1;
5 constexpr auto inner;
6 for (unsigned i = 1; i < writebacktimes; ++i)
7     pragma omp parallel {
8         if (inner < kComputeTilesH * kInnerTiles)
9             cout << inner << " * kInnerTiles + allIn1();
10         if (m1 < kComputeTilesH - 1) {
11             m1 = 1;
12             if (m2 < kComputeTilesH - 1) {
13                 m2 = 1;
14                 ++m2;
15             }
16             ++m1;
17         }
18         if (inner < kComputeTilesH - 1)
19             cout << inner << " * m1";
20     }
21     if (inner < writebacktimes - 1)
22         inner = 1;
23         ++inner;
24     }
25 }
26 // non-templated implementation below
27 // WriteC_M0
28 for (unsigned m1 = 1; m1 < kInnerTiles; ++m1)
29     // WriteC_M1
30     for (unsigned m2 = 1; m2 < kComputeTilesH; ++m2) {
31         // WriteC_M2
32         for (unsigned m3 = 1; m3 < kInnerTiles; ++m3)
33             pragma omp parallel {
34                 cout << inner << " * kInnerTiles + allIn1();
35             }
36     }
37     // Forward other tiles of C
38     // =====
39     // We have to use operators, so first tile contains B= tiles, and the
40     // last tile contains 1 tile.
41     if (inner < kComputeTilesH - 1)
42         // ForwardC_M0s:
43         for (unsigned w = 1; w < kComputeTilesH - inner; ++w)
44             // ForwardC_M1:
45             for (unsigned m1 = 1; m1 < kComputeTilesH; ++m1)
46                 // ForwardC_M2:
47                 for (unsigned m2 = 1; m2 < kInnerTiles; ++m2)
48                     pragma omp parallel {
49                         cout << inner << " * m1";
50                     }
51     }
52     // =====
53 }
54 }

```

```

1
Memory.cpp
1 // Author: Johannes W. Fink, MIT License
2 // Copyright This software is copyrighted under the MIT License.
3
4 #include "Memory.h"
5 #include "constexpr"
6 using namespace std;
7 #ifdef MK_TRANSPOSED
8
9 unsigned inner(const unsigned m1, const unsigned m2, const unsigned m3,
10             const unsigned k1, const unsigned k2, const unsigned size_X,
11             const unsigned size_Y, const unsigned size_Z) {
12     pragma omp parallel {
13         const auto index =
14             [m1 * kOuterTilesH + m2 * kInnerTilesH + m3] * sizeMemory[size_Y] +
15             [k1 * kInnerTilesH + k2 * kMemoryWidth + k3];
16         // assert(index < size_X * sizeMemory[size_Z]);
17         return index;
18     }
19 }
20 #else // MK_TRANSPOSED
21
22 unsigned inner(const unsigned k, const unsigned m1, const unsigned m2,
23             const unsigned size_X, const unsigned size_Y,
24             const unsigned size_Z) {
25     pragma omp parallel {
26         const auto index =
27             k * sizeMemory[size_Y] + m1 * kOuterTilesHMemory + m2;
28         // assert(index < size_X * sizeMemory[size_Z]);
29         return index;
30     }
31 }
32 #endif // MK_TRANSPOSED
33
34 unsigned inner(const unsigned k, const unsigned m1, const unsigned m2,
35             const unsigned size_X, const unsigned size_Y,
36             const unsigned size_Z) {
37     pragma omp parallel {
38         const auto index =
39             k * sizeMemory[size_Y] + m1 * kOuterTilesHMemory + m2;
40         // assert(index < size_X * sizeMemory[size_Z]);
41         return index;
42     }
43 }
44
45 unsigned inner(const unsigned m1, const unsigned m2, const unsigned m3,
46             const unsigned size_X, const unsigned size_Y,
47             const unsigned size_Z) {
48     pragma omp parallel {
49         const auto index = [m1 * kOuterTilesH + m2] * sizeMemory[size_Y] +
50             [m3 * kOuterTilesHMemory + m1];
51         // assert(index < size_X * sizeMemory[size_Z]);
52         return index;
53     }
54 }
55 #ifdef MK_TRANSPOSED
56
57 void _writeBack(const MemoryPack<T> & mem) {
58     size_t memSize = mpiGetTranspMemWidth(), memSizeX = mem.size(),
59             memSizeY = mem.size(), memSizeZ = mem.size();
60     const unsigned size_X, const unsigned size_Y,
61             const unsigned size_Z, const unsigned size_X,
62             const unsigned size_Y, const unsigned size_Z) {
63     pragma omp parallel {
64         auto pack = [memSizeX, memSizeY, memSizeZ, size_X, size_Y, size_Z];
65     }
66     // =====
67     for (unsigned w = 1; w < kMemoryWidth; ++w)
68         mpiPut[k] * kMemoryWidth + w, memSizePack[w];
69     }
70 }
71
72 template < unsigned innerMem>
73 void _writeBack(const MemoryPack<T> & mem) {
74     size_t memSize = mpiGetTranspMemWidth(), memSizeX = mem.size(),
75             memSizeY = mem.size(), memSizeZ = mem.size();
76     const unsigned size_X, const unsigned size_Y,
77             const unsigned size_Z, const unsigned size_X) {
78     pragma omp parallel {
79         for (unsigned m1 = 1; m1 < kInnerTilesH; ++m1)
80             // =====

```

Actual listing (2)


```

Memory.cpp 4
391 MemoryackM_ memoryack;
392 ConvertWidthMemory;
393 for (unsigned j = 0; j < MemoryackdM / kComputeTileSizeM; ++j) {
394     #pragma HLS PIPELINE II=0
395     #pragma HLS LOOP_UNROLL
396     if (j % 4)
397         memoryack = wide.Pop();
398     ComputePackM_computePack;
399 ConvertWidthCompute;
400 for (unsigned w = 0; w < kComputeTileSizeM; ++w) {
401     #pragma HLS UNROLL
402     computePackw = memoryack[j * kComputeTileSizeM + w];
403     narrow.Push(computePack);
404 }
405 #else
406 narrow.Push(wide.Pop());
407 #endif
408 }
409 #endif // HW_TRANSPOSED -- true
410
411 void Read(MemoryackM_ const& memory, StreamComputePackM_ to epipe,
412          const unsigned size_A, const unsigned size_B,
413          const unsigned size_C) {
414     assert(!toA || !toB || !toC || (toA && toB && toC) == 0);
415     QueueTileMsize_A size_A; kQueueTileSizeMemory;
416     MemoryackM::kwidth m_TotalReassemblysize_A, size_B, size_C;
417
418     ReadQueueTileM;
419     for (unsigned m = 0; m < queueTileMsize_A; ++m) {
420         ReadQueueTileM;
421         for (unsigned k = 0; k < size_B; ++k) {
422             ReadQueueTileM;
423             for (unsigned s = 0; s < kQueueTileSizeMemory; ++s) {
424                 #pragma HLS PIPELINE II=0
425                 #pragma HLS LOOP_UNROLL
426                 pipe.Push(memory[instrMk.m, m, s, size_A, size_B, size_C]);
427             }
428         }
429     }
430 }
431
432 void ConvertWidth(StreamComputePackM_ to wide, StreamComputePackM_ to narrow,
433                  const unsigned size_A, const unsigned size_B,
434                  const unsigned size_C) {
435     assert(!kMemorywidthM / kComputeTileSizeM);
436     MemoryackM::kwidth m_TotalReassemblysize_A, size_B, size_C;
437     assert(!toA || !toB || !toC || (toA && toB && toC) == 0);
438     kMemorywidthM / kComputeTileSizeM; kQueueTileSizeM;
439     TotalReassemblysize_A, size_B, size_C;
440
441     ConvertWidthQueue;
442     for (unsigned j = 0; j < TotalReassemblysize_A, size_B, size_C / kMemorywidthM; ++j) {
443         MemoryackM_ memoryack;
444         ConvertWidthMemory;
445         for (unsigned l = 0; l < kMemorywidthM / kComputeTileSizeM; ++l) {
446             #pragma HLS PIPELINE II=0
447             #pragma HLS LOOP_UNROLL
448             if (j % 4)
449                 memoryack = wide.Pop();
450             ComputePackM_computePack;
451             ConvertWidthCompute;
452             for (unsigned w = 0; w < kComputeTileSizeM; ++w) {
453                 #pragma HLS UNROLL
454                 computePackw = memoryack[j * kComputeTileSizeM + w];
455                 narrow.Push(computePack);
456             }
457         }
458     }
459 }

```

```

Memory.cpp 5
391 |
392 void ConvertWidth(StreamComputePackM_ to narrow, StreamComputePackM_ to wide,
393                  const unsigned size_A, const unsigned size_B,
394                  const unsigned size_C) {
395     assert(!kMemorywidthM / kComputeTileSizeM);
396     // assert(!size_A || size_B || MemoryackM::kwidth)
397     // kMemorywidthM / kComputeTileSizeM; kQueueTileSizeM; kQueueTileSizeM;
398     // size_A, size_B;
399
400     ConvertWidthQueue;
401     for (unsigned j = 0; j < queueTileMsize_A; kQueueTileSizeM; ++j) {
402         ConvertWidthM;
403         for (unsigned l = 0; l < kQueueTileSizeM; kQueueTileSizeM; ++l) {
404             #ifdef HW_CONVERT_M
405                 ConvertWidthMemory;
406                 MemoryackM_ memoryack;
407                 for (unsigned k = 0; k < kMemorywidthM / kComputePackM::kwidth; ++k) {
408                     #pragma HLS PIPELINE II=0
409                     #pragma HLS LOOP_UNROLL
410                     const auto computePack = wide.Pop();
411                     ConvertWidthCompute;
412                     for (unsigned w = 0; w < kComputePackM::kwidth; ++w) {
413                         #pragma HLS UNROLL
414                         computePackM::kwidth + w;
415                         memoryack[j * kQueueTileSizeM + w];
416                         if (j % 4)
417                             memoryack = computePack[w];
418                     }
419                     if (j % 4)
420                         narrow.Push(memoryack);
421                 }
422             #else
423                 narrow.Push(wide.Pop());
424             #endif
425         }
426     }
427
428     void write(StreamComputePackM_ to epipe, MemoryackM_ memory) {
429         const unsigned size_A, const unsigned size_B,
430         const unsigned size_C) {
431             // assert(!toA || !toB || !toC || (toA && toB && toC) == 0);
432             // kQueueTileSizeMemory; kQueueTileSizeM; kQueueTileSizeM;
433             // kQueueTileSizeMemory; MemoryackM::kwidth == size_A, size_B;
434
435             writeQueueTileM;
436             for (unsigned m = 0; m < QueueTileMsize_A; ++m) {
437                 writeQueueTileM;
438                 for (unsigned k = 0; k < QueueTileMsize_B; ++k) {
439                     writeQueueTileM;
440                     for (unsigned s = 0; s < kQueueTileSizeMemory; ++s) {
441                         #pragma HLS PIPELINE II=0
442                         #pragma HLS LOOP_UNROLL
443                         const auto val = pipe.Pop();
444                         if (toA || toB || toC)
445                             m_TotalReassemblysize_A, m, s, size_A, size_B, size_C;
446                     }
447                 }
448             }
449         }
450     }
451
452     #ifdef HW_WITHESD
453     static const char* flushLine() {
454         return "FlushLine" << m << ", " << k << ", " << s << "\n";
455     }
456     for (unsigned m = 0; m < QueueTileMsize_A; ++m) {
457         for (unsigned k = 0; k < QueueTileMsize_B; ++k) {
458             flushLine();
459         }
460     }
461     #endif
462 }
463
464 #ifdef HW_CONVERT_M
465 void read(StreamComputePackM_ to narrowMemory, StreamComputePackM_ to wideMem,
466          const unsigned size_A, const unsigned size_B,
467          const unsigned size_C) {
468     #else
469     void read(StreamComputePackM_ to narrowMemory, StreamComputePackM_ to wideMem,
470             const unsigned size_A, const unsigned size_B,

```

Actual listing (4)

Linpack performance of a large FPGA

For 32-bit floating-point:

- 409 GFlop/s (10 GFlop/J),
- (other works report 800 GFlop/s)
- The processor of my laptop: about 100 GFlop/s (3 GFlop/J)
- Top-end GPUs are above 1000 GFlop/s peak

Linpack performance of a large FPGA

For 32-bit floating-point:

- 409 GFlop/s (10 GFlop/J),
- (other works report 800 GFlop/s)
- The processor of my laptop: about 100 GFlop/s (3 GFlop/J)
- Top-end GPUs are above 1000 GFlop/s peak

Not bad considering that the FPGA runs below 500MHz, but...
you are a bit disappointed, aren't you ?

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

Ideed, processors and GPUs are engineered to be good at FP32 and FP64 Linpack Flops.

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

Ideed, processors and GPUs are engineered to be good at FP32 and FP64 Linpack Flops.

Peak marketing lies for double-precision floating-point exponential

(Intel MKL vector libm, versus FloPoCo FPExp)

- Processor core: 20 cycles / DPExp @ 4GHz: **200 MDPExp/s**
- FPExp in FPGA: 1 DPExp/cycle @ 400MHz: **400 MDPExp/s**
- Chip vs chip: 20 cores/processor vs 200 FPExp/FPGA: **FPGA 20 times better**
- Power consumption improvement even higher
- Single precision data also better




Florent de Dinechin and Bogdan Pasca.

Floating-point exponential functions for DSP-enabled FPGAs.

In Field Programmable Technologies, 2010.

SPICE Model Evaluation

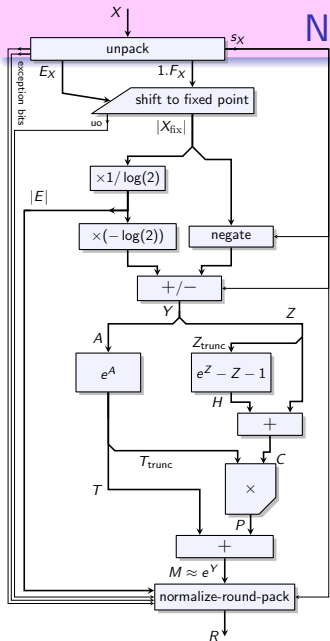
Models	Instruction Distribution					
	Add	Mult.	Div.	Sqrt.	Exp.	Log
bjt	22	30	17	0	2	0
diode	7	5	4	0	1	2
hbt	112	57	51	0	23	18
jfet	13	31	2	0	2	0
mos1	24	36	7	1	0	0
vbic	36	43	18	1	10	4

-  Nachiket Kapre and Andre DeHon
 Accelerating SPICE Model-Evaluation using FPGAs.
In Field-Programmable Custom Computing Machines, 2009.

Not your processor's exponential

Why is it 2x faster when it should be 10x slower ?

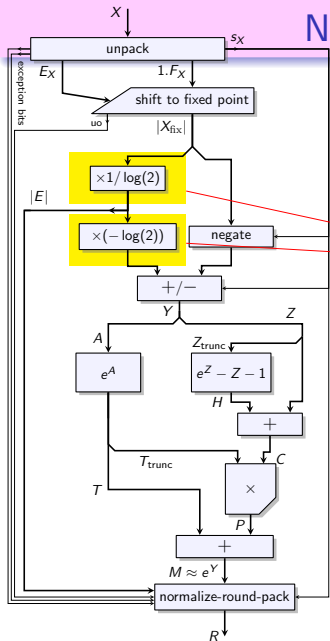
- data-flow architecture (no von Neuman tax)



Not your processor's exponential

Why is it 2x faster when it should be 10x slower ?

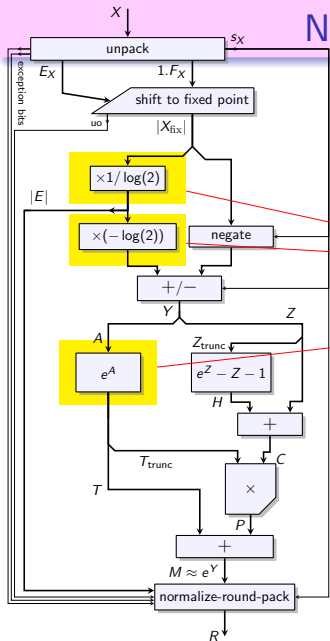
- data-flow architecture (no von Neuman tax)
- application-specific constant multipliers



Not your processor's exponential

Why is it 2x faster when it should be 10x slower ?

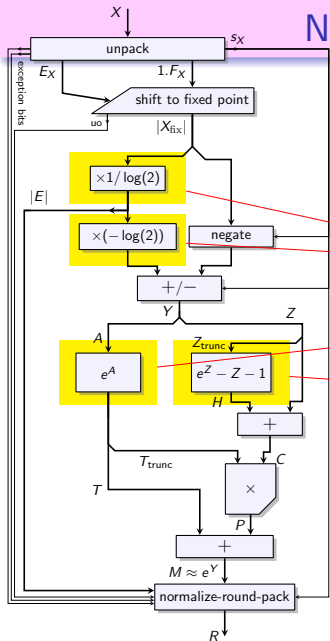
- data-flow architecture (no von Neuman tax)
- application-specific constant multipliers
- tables of pre-computed values



Not your processor's exponential

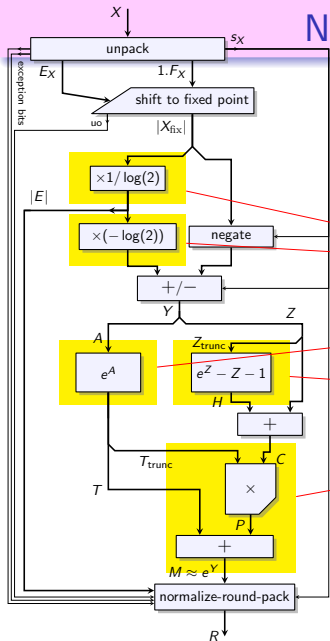
Why is it 2x faster when it should be 10x slower ?

- data-flow architecture (no von Neuman tax)
- application-specific constant multipliers
- tables of pre-computed values
- arbitrary function evaluator



Not your processor's exponential

Why is it 2x faster when it should be 10x slower ?

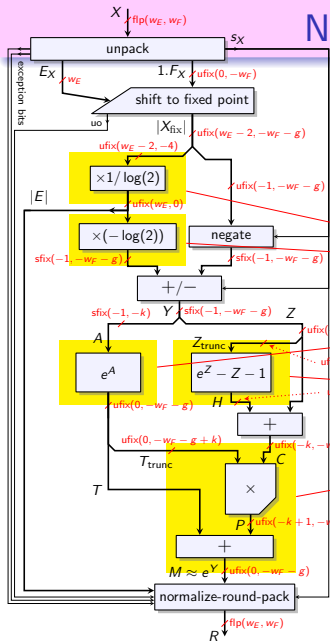


- data-flow architecture (no von Neuman tax)
- application-specific constant multipliers
- tables of pre-computed values
- arbitrary function evaluator
- matching DSP blocks

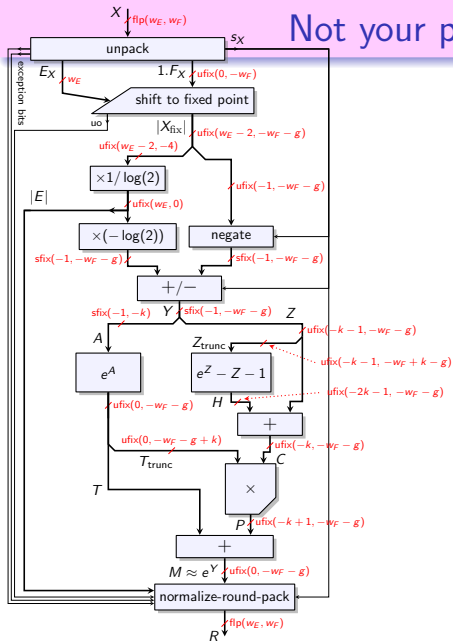
Not your processor's exponential

Why is it 2x faster when it should be 10x slower ?

- data-flow architecture (no von Neuman tax)
- application-specific constant multipliers
- tables of pre-computed values
- arbitrary function evaluator
- matching DSP blocks
- only compute useful bits

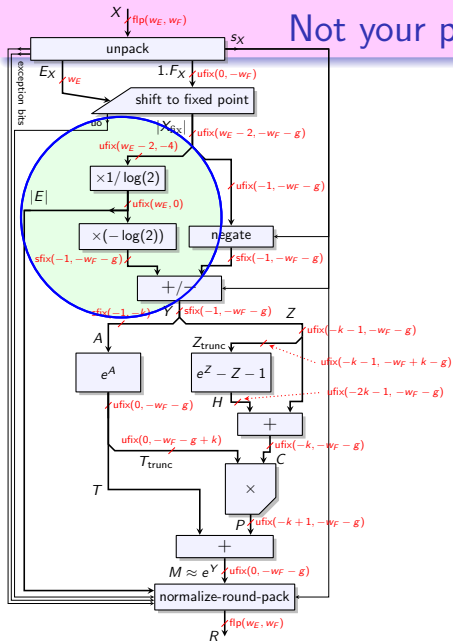


Not your processor's multipliers, either



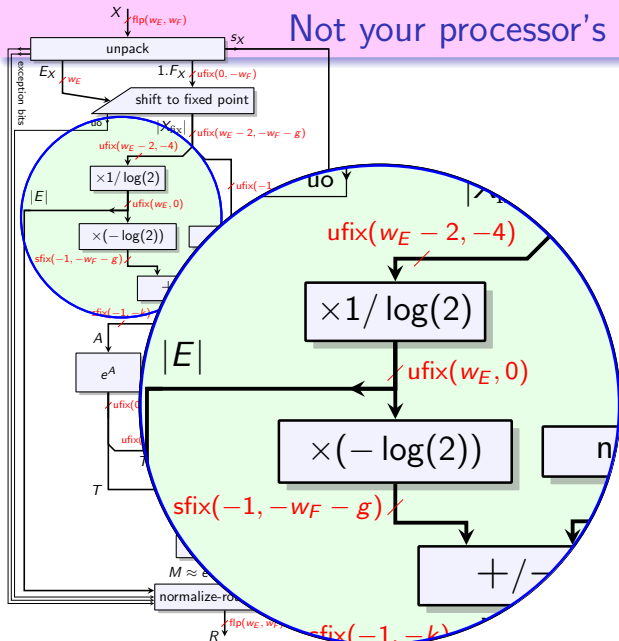
- only compute useful bits

Not your processor's multipliers, either



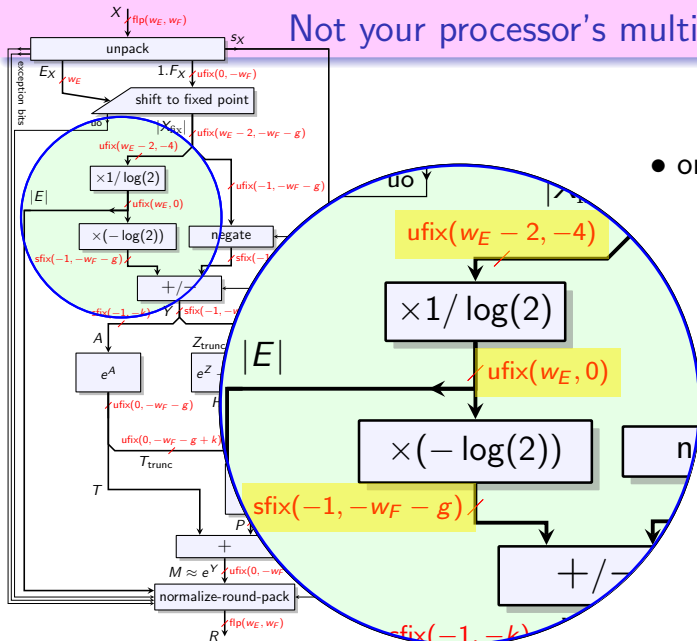
- only compute useful bits

Not your processor's multipliers, either



- only compute useful bits

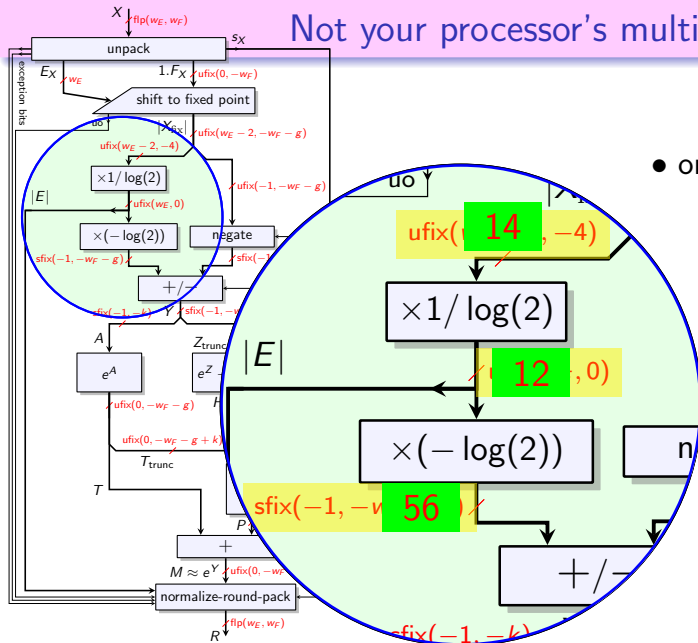
Not your processor's multipliers, either



- only compute useful bits

– depending on
exponent size w_E
and mantissa size w_F

Not your processor's multipliers, either



- only compute useful bits

- depending on
exponent size $w_E = 11$
and mantissa size $w_F = 52$

- compare to **FP64**
12 bits: too small
56 bits: too large

A few FPGA success stories

Introduction: the war of the programming models

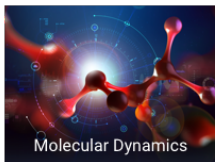
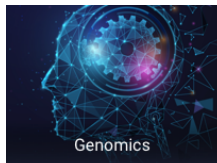
FPGA architectures

Programming FPGAs

A few FPGA success stories

Conclusion

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

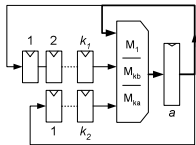
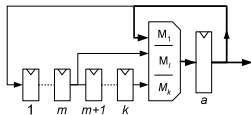


©Xilinx

- 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.

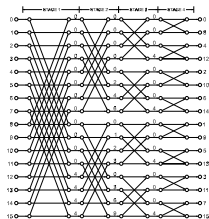
A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation
In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2009.

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

 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.

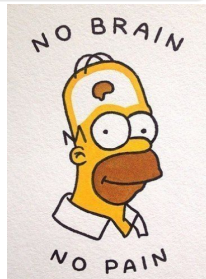
So many papers these days, just pick one extreme:

 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



TigerGraph claims $100\times$ acceleration of Cosine Similarity in one FPGA+HBM, compared to a dual 24-core Xeon server.

- memory-intensive kernels on GBs of data
- the expensive cache hierarchy of the processor is no good here

Conclusion

Introduction: the war of the programming models

FPGA architectures

Programming FPGAs

A few FPGA success stories

Conclusion

OK, I'm convinced, I want to try to program an FPGA, where do I start?

- Amazon, Microsoft, OVH,... all offer FPGA resources on the cloud
- ... sometimes with pre-packaged FPGA-ware.
- Intel and Xilinx sell accelerator cards
- also many entry-level boards to play with
- HLS definitely more welcoming than Hardware Description Languages
 - don't believe that you'll get good results without understanding the FPGA architecture
 - don't believe that you'll get good results without tweaking

A quote from the random generation paper:

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

A quote from the random generation paper:

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:

$\forall X$ 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 your computation

When do we get an FPGA in every PC ?

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)

When do we get an FPGA in every PC ?

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)

Dinechin's Theorem: we'll wait forever.

When do we get an FPGA in every PC ?

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)

Dinechin's Theorem: we'll wait forever.

Proof: Assume that such an application pops up.

- either it is indeed widely useful
 - then next year's processor 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 then it means that it was not a killer app.

When do we get an FPGA in every PC ?

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)

Dinechin's Theorem: we'll wait forever.

Proof: Assume that such an application pops up.

- either it is indeed widely useful
 - then next year's processor 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 then it means that it was not a killer app.

The killer feature of FPGAs is their fine-grain programmability.

When do we get an FPGA in every PC ?

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)

Dinechin's Theorem: we'll wait forever.

Proof: Assume that such an application pops up.

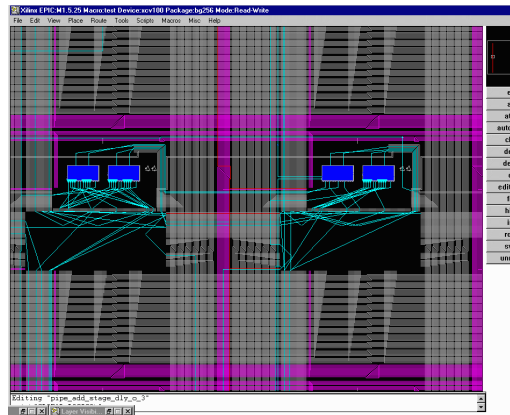
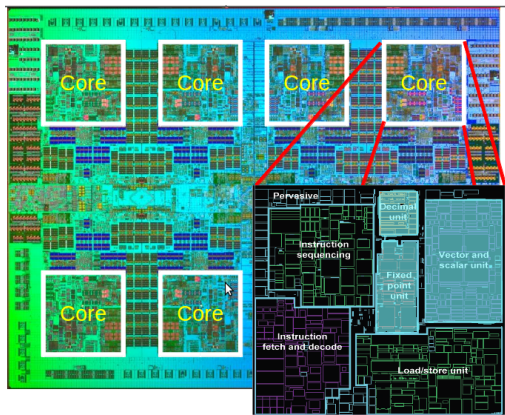
- either it is indeed widely useful
 - then next year's processor will do it in hardware 10x faster than the FPGA,
 - so it won't be an *FPGA* killer app next year...
- or the FPGA remains competitive next year
 - but then it means that it was not a killer app.

The killer feature of FPGAs is their fine-grain programmability.

Constraints in classical programming become degrees of freedom in FPGAs

Back to the war of the programming models

Here are two ~~programmable chips~~ pathetic ways of wasting silicon.



Which is best for (insert your computation here) ?