5IF Tronc Commun Scientifique

Computing with circuits

Florent de Dinechin

Outline

The kind of stuff you get in keynote talks in hardware conferences;
Then a philosophical introduction to my own little problems:

Moore's Law and the end of it

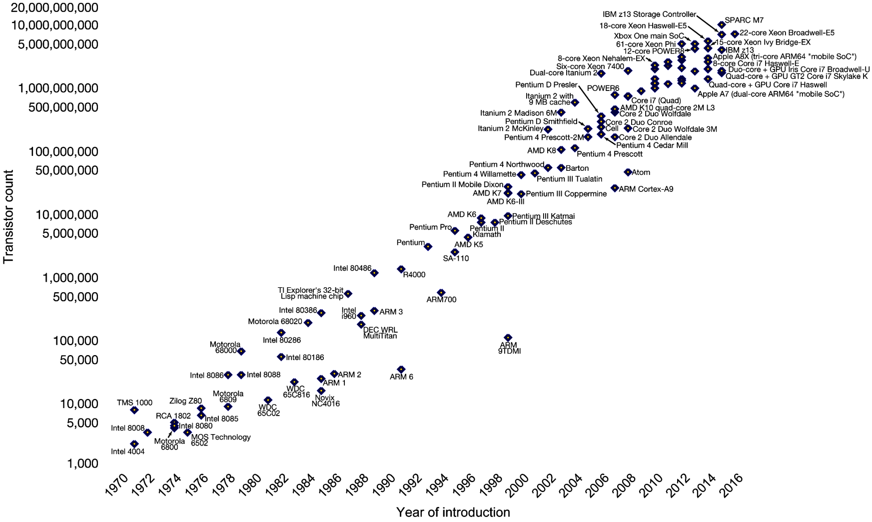
Moore's law

From observations in a 1965 paper by Gordon Moore (Intel)

The number of transistors we can pack on an economically viable chip doubles every two years

- Mostly a self-fulfilling prophecy
  - If it stops being true, a huge part of the economy collapses
- Mostly thanks to the ability to etch *smaller transistors*
  - $\sqrt{2}$ times smaller every other year
- plus, up to the 70s, improvements in chip area
  - current plateau at 1 cm^2
  - ... for "economically viable"

From 2004 on: more transistors produced in the world than grains of rice, and cheaper



Licence CC, Source: Wikipedia

Dennard scaling

From a 1974 paper by Robert Dennard (IBM)

Smaller transistors run faster and consume less

In details,

- frequency follows Moore's law
- computing power follows Moore's law
- power dissipated in a transistor follows inverse Moore's law
  - factor $\sqrt{2}$ on both voltage and current

And overall:

- chip-level dissipated power mostly constant

Dennard scaling

From a 1974 paper by Robert Dennard (IBM)

Smaller transistors run faster and consume less

In details,

- frequency follows Moore's law
- computing power follows Moore's law
- power dissipated in a transistor follows inverse Moore's law
  - factor $\sqrt{2}$ on both voltage and current

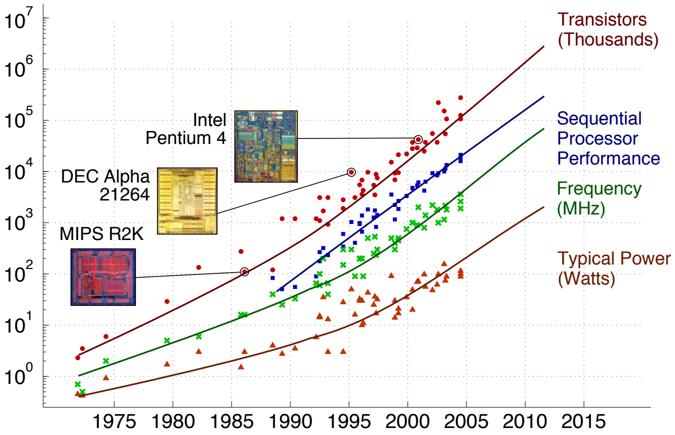
And overall:

- chip-level dissipated power mostly constant

Contrary to Moore's law, Dennard scaling stopped in 2004.

The good old times of Dennard scaling

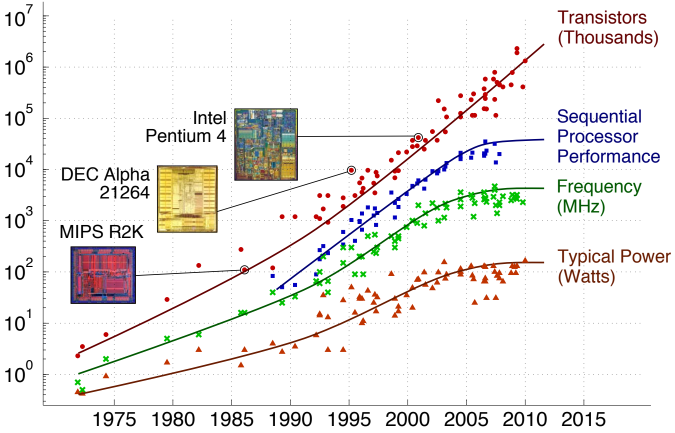
Exponential Scaling for Processor Computation



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

The end of Dennard scaling

Trend 2: Power Constrains Single-Processor Scaling



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Why the end of Dennard scaling

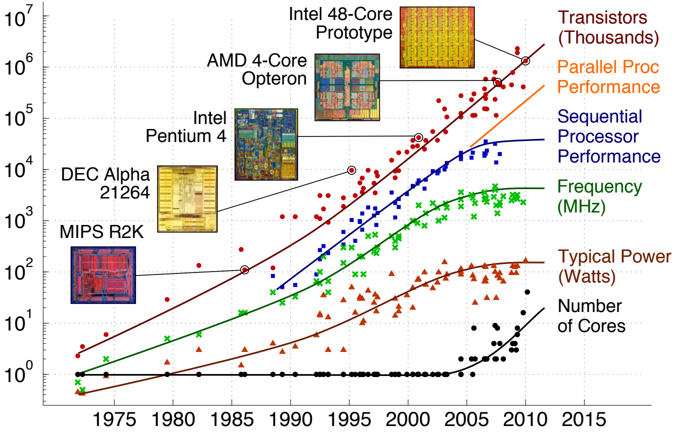
We can build faster circuits, the problem is that they melt down

Practical power dissipation limit: $100 W/cm^2$
10x your cooking pan, comparable to the rods of a nuclear power plant

In the previous slide, the line that imposes the trend is the power.
Remark: 3D integration helps Moore, but annoys Dennard even more.

The current solution to the end of Dennard scaling

Trend 2: Multicore Performance Scaling



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

The problem with the current solution to the end of Dennard scaling

The great depression

- Edward Lee: The Problem With Threads, 2006
- David Patterson: The Trouble With Multicore, 2010

Homework: go read them.

Reality shouldn't constrain our formalisms

Reality shouldn't constrain our formalisms

The end of Moore

- Size of an atom?
  The mesh size in silicon crystal is about 0.5nm (1nm=10^{-9}m).

Reality shouldn't constrain our formalisms

The end of Moore

- Size of an atom?
  The mesh size in silicon crystal is about 0.5nm (1nm=10^{-9}m).
- Current technology is marketed as 14nm
  This corresponds to 30 atoms wide.

Reality shouldn't constrain our formalisms

The end of Moore

- Size of an atom?
  The mesh size in silicon crystal is about 0.5nm (1nm=10^{-9}m).
- Current technology is marketed as 14nm
  This corresponds to 30 atoms wide.
- Corresponding oxide layer is about two atoms high,
  and won't get much thinner.

Reality shouldn't constrain our formalisms

The end of Moore

- Size of an atom?
  The mesh size in silicon crystal is about 0.5nm (1nm=10^{-9}m).
- Current technology is marketed as 14nm
  This corresponds to 30 atoms wide.
- Corresponding oxide layer is about two atoms high,
  and won't get much thinner.

The end of Dennard

- Corresponding oxide layer is about two atoms high.
  $\longrightarrow$ quantum tunelling $\longrightarrow$ power waste

Reality shouldn't constrain our formalisms

The end of Moore

- Size of an atom?
  The mesh size in silicon crystal is about 0.5nm (1nm=10^{-9}m).
- Current technology is marketed as 14nm
  This corresponds to 30 atoms wide.
- Corresponding oxide layer is about two atoms high,
  and won't get much thinner.

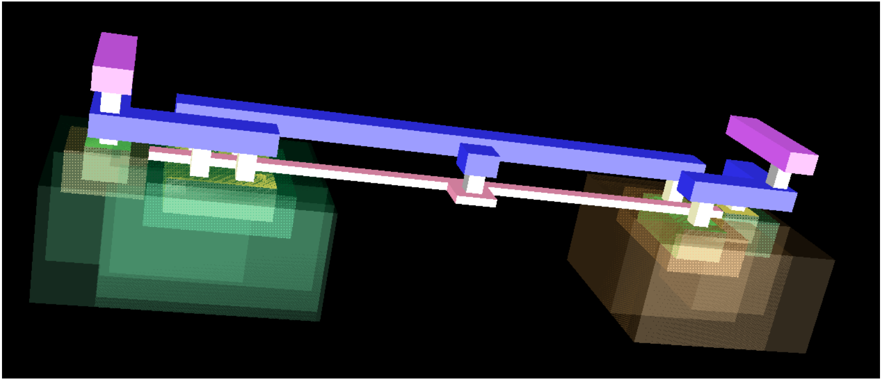
The end of Dennard

- Corresponding oxide layer is about two atoms high.
  $\longrightarrow$ quantum tunelling $\longrightarrow$ power waste
- Transistor threshold voltage got down from 5V to 1V,
  and won't go much lower

Oxide layer?

The following picture is advertising for the Electric CAD software

http://www.staticfreesoft.com/



(more interactive advertising if the beamer allows)

Reality shouldn't constraint our formalisms

Other limits

- Speed of light?

Reality shouldn't constraint our formalisms

Other limits

- Speed of light? $3 \cdot 10^8$ m/s.

Reality shouldn't constraint our formalisms

Other limits

- Speed of light? $3 \cdot 10^8$ m/s.
- At the speed of light, a 3GHz signal
  travels no further than 10 cm in a period
- Homework: cross this with atom size, and get a limit frequency

It's the economy, stupid

The economic cost of a self-fulfilling prophecy

Each new foundry is twice as expensive as the previous one
(or: the cost of a new foundry also follows Moore's law)

- Why?
  - Build billions of reliable objects, each 3 atoms high, 50 atoms wide requires a pretty good vacuum cleaner
  - Lithographic process used light, then UV, now almost X rays...
- So foundries are teaming up to share the costs
- At 22nm, we were down to 5 foundries
- ... at some point there will be no competitor left to merge with.

It's the energy, stupid

Back to physics:

Computing consumes energy

- Each bit flipped entails a transfer of electrons from the ⊖ to the ⊕ through some resistors
- Currently, switching 1 bit costs $10^{-18} J$ (1 attoJoule)

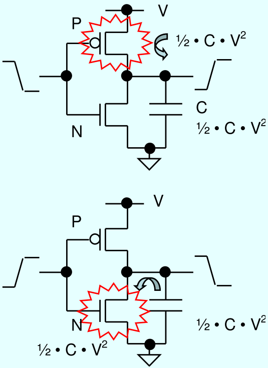


Figure from *Energy per Instruction Trends in Intel Microprocessors* by Grochowski and Annavaramx

It's the energy, stupid (2)

Moving bits consumes energy

- Switching 1 bit costs $10^{-18}J$,
- Moving 1 bit costs $10^{-12}J/cm$ ($1pJ/cm$)

(Really the same drawing, but with a larger C)

Doing nothing consumes energy

These days, roughly 1/3rd of power is leaked (quantum tunelling, etc).

It's the energy, stupid (3): the macro view

Approximate power in 28nm processor (adapted from Bill Dally)

- One 64-bit floating-point Fused Multiply-Add: **50 pJ**
  - This includes switching and moving around inside the FMA
- Access to a 1Kx256-bit on-chip SRAM: **50 pJ**

It's the energy, stupid (3): the macro view

Approximate power in 28nm processor (adapted from Bill Dally)

- One 64-bit floating-point Fused Multiply-Add: **50 pJ**
  - This includes switching and moving around inside the FMA
- Access to a 1Kx256-bit on-chip SRAM: **50 pJ**

- Moving 64 bits 1mm on-chip: **6 pJ**
- Moving 64 bits 1cm on-chip: **64 pJ**
  - Remark: there are several tens of km of wires inside your Core i7

It's the energy, stupid (3): the macro view

Approximate power in 28nm processor (adapted from Bill Dally)

- One 64-bit floating-point Fused Multiply-Add: **50 pJ**
  - This includes switching and moving around inside the FMA
- Access to a 1Kx256-bit on-chip SRAM: **50 pJ**

- Moving 64 bits 1mm on-chip: **6 pJ**
- Moving 64 bits 1cm on-chip: **64 pJ**
  - Remark: there are several tens of km of wires inside your Core i7

- Reading 64 bits from external DRAM: **4000 pJ**
  - due to the C of macro wires (between chips on your main board)

Hence the current trends in VLSI circuits

Exposed here very well by Christopher Batten:

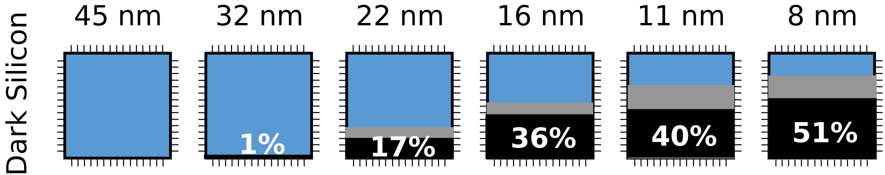
`https://web.csl.cornell.edu/engrg1060/handouts/engrg1060-ece-lecture.pdf`

The dark silicon apocalypse

Dark silicon?

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

“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, serve as radiator for the used parts

Since they are rare, nobody bothered to study them before...

And besides and en vrac

More data from various conference presentations

- In a 1MBit SRAM (a cache), 10^{-25} fault per bit per cycle (and worsening)
  - solved by hardware CRC in SRAM architectures,
  - but similar fault rates affect any circuit...
- Flash memory displacing spinning disks
- Non-volatile RAM soon to displace flash memory
  - micro-controler with non-volatile registers now on the market
  - A complete game changer in OS design
- Digital radio: able to manage charges with a resolution of 200 electrons

Trends in embedded systems

(an old slide written in 2013) Tilera versus Kalray versus Platform2012

Homework: go googling their respective datasheets, and place bets.

- Things in common
  - Massively multicore (64-256)
  - Modern 64-bit VLIW cores
  - Homogeneous network-on-chip
- Things that differ
  - Local memory: scratchpad (Kalray), L1+L2 caches (Tilera), core-specific (P2012)
    How to avoid memory starvation?
  - Homogeneous nodes (Kalray, Tilera) versus heterogeneous nodes (P2012)
  - Approaches to clock domains and power domains (probably, did not check really)
  - Floating-point (Kalray) versus integer-only
  - Only two of them made in Grenoble (Kalray, P2012)
- 2019 update: only Kalray still alive

Why care about embedded systems?

- This is where 90% of the research will need you
- This is where Europe is (still) active

Computing with circuits

Examples of computations

- Sorting n numbers
- Computing the product of an $n \times n$ matrix by a vector
- Computing the exponential of a double-precision floating-point number
- Generating pseudo random numbers with a Gaussian distribution

Examples of computations

- Sorting n numbers
- Computing the product of an $n \times n$ matrix by a vector
- Computing the exponential of a double-precision floating-point number
- Generating pseudo random numbers with a Gaussian distribution

Compiling a computation

- implementing a computation

Examples of computations

- Sorting n numbers
- Computing the product of an $n \times n$ matrix by a vector
- Computing the exponential of a double-precision floating-point number
- Generating pseudo random numbers with a Gaussian distribution

Compiling a computation

- implementing a computation
- evaluating the quality of the implementation ($\approx$ complexity)

Examples of computations

- Sorting n numbers
- Computing the product of an $n \times n$ matrix by a vector
- Computing the exponential of a double-precision floating-point number
- Generating pseudo random numbers with a Gaussian distribution

Compiling a computation

- implementing a computation
- evaluating the quality of the implementation ($\approx$ complexity)
- because we want to optimize the implementation

Warning: we're going to change the computing model

At this point of your career, you should have pretty clear ideas on

- implementation on a PC, including sequential complexity
- complexity notions in non-practical models

Before compiling to circuits, we need to deconstruct some of your software heritage.

Practical complexity

Just like the real thing, but

- constants are important
- the actual machine is important

Practical complexity

Just like the real thing, but

- constants are important
- the actual machine is important

Between “asymptotic complexity on PRAM” and “`time` + `gnuplot`’’

Practical complexity

Just like the real thing, but

- constants are important
- the actual machine is important

Between “asymptotic complexity on PRAM” and “`time` + `gnuplot`''

PRAM (Parallel RAM) is a very nice model built against the laws of physics...

Practical complexity

Just like the real thing, but

- constants are important
- the actual machine is important

Between "asymptotic complexity on PRAM" and "`time` + `gnuplot`''

PRAM (Parallel RAM) is a very nice model built against the laws of physics...

The level we need for an optimizing compiler to take decisions

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting n integers, each represented on 16 bits ?

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting *n* integers, each represented on 16 bits ?
    - bucket sorting maybe ?

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting *n* integers, each represented on 16 bits ?
  - bucket sorting maybe ?
- How many boolean operations does it take to sort *n* 16-bit integers?

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting n integers, each represented on 16 bits ?
  - bucket sorting maybe ?
- How many boolean operations does it take to sort n 16-bit integers?
  - I don't know exactly, but there is an answer
  - (need to refine the problem formulation a bit more, still. Where?)

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting n integers, each represented on 16 bits ?
  - bucket sorting maybe ?
- How many boolean operations does it take to sort n 16-bit integers?
  - I don't know exactly, but there is an answer
  - (need to refine the problem formulation a bit more, still. Where?)
- How many boolean operations does it take to sort n arbitrarily large integers?

For a circuit, typing is important

complexity + sorting algorithms = oh no, not again ?

- What do you think of the asymptotic complexity of sorting n integers, each represented on 16 bits ?
  - bucket sorting maybe ?
- How many boolean operations does it take to sort n 16-bit integers?
  - I don't know exactly, but there is an answer
  - (need to refine the problem formulation a bit more, still. Where?)
- How many boolean operations does it take to sort n arbitrarily large integers?
  - I don't know exactly, but I know it depends on their coding
  - (need to refine the problem formulation even more)

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms?

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but...

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but... floating-point addition is not associative.

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but... floating-point addition is not associative.
Strassen or blocking *return different results*.

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but... floating-point addition is not associative.
Strassen or blocking *return different results*.
Then, returning the null matrix also answers the problem, and faster.

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but... floating-point addition is not associative.
Strassen or blocking *return different results*.
Then, returning the null matrix also answers the problem, and faster.

Quoting Kahan (Turing Award)

The fast drives out the slow even if the fast is wrong.

For optimizing and parallelizing compilers, typing is important, too

What can be dumber than sorting algorithms? Matrix product maybe?

- Strassen, then Coppersmith/Winograd for reducing asymptotic complexity
- Blocking for reducing practical complexity (cache inefficiencies)

Very nice on the reals, but... floating-point addition is not associative.
Strassen or blocking *return different results*.
Then, returning the null matrix also answers the problem, and faster.

Quoting Kahan (Turing Award)

The fast drives out the slow even if the fast is wrong.

The good question is still open

What is the complexity (in FP operations) of computing the product of two $n \times n$ FP matrices with 3 correct decimal digits in each element of the result?

Why this is a compiler issue, too

- The compiler should respect the semantic of the language
- C imposes a strict sequential evaluation order
- Many optimizations are illegal in C.
- Solutions:
  - program in Fortran :)
  - hide the problem behind under-specified libraries (BLAS)
  - or improve the language

The good language is still to design

- declarative + precision specification
- what to compute, not how to do it

For a circuit, typing is important, 2

Computing the exponential of a floating-point number.

Lindemann theorem

if z is rational and $z \neq 0$ then e^z is transcendental.

What is the complexity of computing its binary representation?

For a circuit, typing is important, 2

Computing the exponential of a floating-point number.

Lindemann theorem

if z is rational and $z \neq 0$ then e^z is transcendental.
What is the complexity of computing its binary representation?

- Let's compute an approximation, then.

For a circuit, typing is important, 2

Computing the exponential of a floating-point number.

Lindemann theorem

if z is rational and $z \neq 0$ then e^z is transcendental.
What is the complexity of computing its binary representation?

- Let's compute an approximation, then.

How many floating-point operations does it take to compute the floating-point number nearest to the exponential of a floating-point number?

For a circuit, typing is important, 2

Computing the exponential of a floating-point number.

Lindemann theorem

if z is rational and $z \neq 0$ then e^z is transcendental.
What is the complexity of computing its binary representation?

- Let's compute an approximation, then.

How many floating-point operations does it take to compute the floating-point number nearest to the exponential of a floating-point number?
How many operations does it take to compute the floating-point number nearest to the exponential of a floating-point number?

For a circuit, typing is important, 2

Computing the exponential of a floating-point number.

Lindemann theorem

if z is rational and $z \neq 0$ then e^z is transcendental.
What is the complexity of computing its binary representation?

- Let's compute an approximation, then.

How many floating-point operations does it take to compute the floating-point number nearest to the exponential of a floating-point number?
How many binary operations does it take to compute the floating-point number nearest to the exponential of a floating-point number?

Don't forget to type the complexity, too!

Practical version

How many cycles does it take to compute exp if I have two parallel *fused multiply-and-add* operators pipelined in 5 cycles each?

Practical version

How many cycles does it take to compute `exp` if I have two parallel *fused multiply-and-add* operators pipelined in 5 cycles each? How much memory does it consume?

Practical version

How many cycles does it take to compute exp if I have two parallel *fused multiply-and-add* operators pipelined in 5 cycles each? How much memory does it consume?

See the Itanium books by HP's Markstein or Intel's Cornea et al.
In both cases, mathematical library team very close to the compiler team.

Time versus space

Naive matrix product takes n^3 operations.

My FPGAs are massively parallel. These n^3 operations can be computed on n^2 operators in n time.

Time versus space

Naive matrix product takes n^3 operations.

My FPGAs are massively parallel. These n^3 operations can be computed on n^2 operators in n time.

Let's vote

Strassen brings down the operation count for matrix multiplication to $2^{2.808}$. Does it reduce

- the time?
- the number of needed operators?
- both?

The future is parallel, isn't it?

The past was parallel, too (we have had a *Laboratoire de l'Informatique et du Parallélisme* in Lyon since the 90s)

The future is parallel, isn't it?

The past was parallel, too (we have had a *Laboratoire de l'Informatique et du Parallélisme* in Lyon since the 90s)

Three problems:

- Technological problem 1: cost of communication wrt cost of computation

The future is parallel, isn't it?

The past was parallel, too (we have had a *Laboratoire de l'Informatique et du Parallélisme* in Lyon since the 90s)
Three problems:

- Technological problem 1: cost of communication wrt cost of computation
- Technological problem 2: *programming*
  - parallel programming *vs* automatic parallelization
  - none of which works well in all cases

The future is parallel, isn't it?

The past was parallel, too (we have had a *Laboratoire de l'Informatique et du Parallélisme* in Lyon since the 90s)

Three problems:

- Technological problem 1: cost of communication wrt cost of computation
- Technological problem 2: *programming*
  - parallel programming *vs* automatic parallelization
  - none of which works well in all cases
- Semantic problem: wild parallelism $\Longrightarrow$ non-determinism
  - linked to the language problem, of course

The future is parallel, isn't it?

The past was parallel, too (we have had a *Laboratoire de l'Informatique et du Parallélisme* in Lyon since the 90s)

Three problems:

- Technological problem 1: cost of communication wrt cost of computation
- Technological problem 2: *programming*
  - parallel programming *vs* automatic parallelization
  - none of which works well in all cases
- Semantic problem: wild parallelism $\Longrightarrow$ non-determinism
  - linked to the language problem, of course

See *the great depression*.

Processor makers' current marketing

Your old bicyle takes tou from Terreaux to La Doua in 20 minutes?
Buy our new dodecacycle, and you will travel in less than 2 minutes!

Still, there is no escaping parallelism

I call that a heavy trend

- current top supercomputer built out of 260-core chips
- current phone processors with 8 cores
- Kalray processor has 288 cores
- current GPUs with 400+ cores
- current FPGAs with 3000+ multipliers

Hardware description languages

Classical design flow

- Circuit described in the VHDL or Verilog languages
- Compilation in several steps
  - Logic optimization
  - Technology mapping: implement the logic as a graph of basic components
    - CMOS VLSI: nand, nor, flip-flop, ...
    - FPGA: 6-input LUT, 18x18 multiplier, ...
  - Place the components to minimize wasted space and total length of wires (NP-complete)
  - Route the wires between the components (also NP-complete)
- Place and route may take weeks or months...

The VHDL language in two slides (1)

- Entities (= black boxes), ports, instances, signals (= wires)
  - just like when we draw architectures

The VHDL language in two slides (1)

- Entities (= black boxes), ports, instances, signals (= wires)
  - just like when we draw architectures
- Intrinsically parallel
  - in a circuit, all the gates operate in parallel
  - consequence: the order of statements in the code is often irrelevant
  - `A <= B xor C;` means: connect output of `B xor C` to `A`
    (this describes an infinite number of xor operations)

The VHDL language in two slides (1)

- Entities (= black boxes), ports, instances, signals (= wires)
  - just like when we draw architectures
- Intrinsically parallel
  - in a circuit, all the gates operate in parallel
  - consequence: the order of statements in the code is often irrelevant
  - `A <= B xor C;` means: connect output of `B xor C` to `A`
    (this describes an infinite number of xor operations)
- Two approaches to describing circuits (to be used together)
  - structural: connect boxes with wires
    - to be used for hierarchical description of complex circuits
  - behavioural: describes *what the circuit does*, not how it is built
    - to be used to describe the lower-level (smaller) boxes
    - describe semantics, leave to the compiler the technology-dependent plumbing of gates/LUT

Semantic of a circuit? Event-driven simulation

.

- an event (t, s, v) is a transition of signal s to value v at instant t. v may be 0, 1, Z (high impedance), and a few others
- the semantic of a circuit is: how it reacts to events on its inputs.

How to simulate a circuit (event-driven simulator)

- maintain a list of events (t, s), sorted by t:
  next event to happen is first of list
- while(list not empty) {
  remove first event;
  propagate it through the components that have it at input;
  insert the resulting events in the list;
  }

Not completely deterministic if several events happen at the same time.

The VHDL language in two slides (2)

- The semantic of `A <= B xor C;` is:
  each time an event arrives to `B` or to `C`, propagate it through the `xor` to generate an event on `A`
- again, such statements may be written in any order: the order of events is given by the graph of the circuit
- more accurate (when needed): `A <= B xor C after 10 ns;`
- Behaviourial VHDL: describe your circuit as processes that react to events. Such processes may be described in an imperative language.

A gentle introduction to FPGAs?

FPGAs

- FPGAs are mass-produced VLSI chips designed to emulate arbitrary logic circuits
- Mostly used for rapid prototyping
  - Simulate/debug a circuit at 1/10 the speed
  - instead of 1/100000 for software simulation

FPGAs

- FPGAs are mass-produced VLSI chips designed to emulate arbitrary logic circuits
- Mostly used for rapid prototyping
  - Simulate/debug a circuit at 1/10 the speed
  - instead of 1/100000 for software simulation

- Other applications:
  - Small series (cheaper than designing a chip)

FPGAs

- FPGAs are mass-produced VLSI chips designed to emulate arbitrary logic circuits
- Mostly used for rapid prototyping
  - Simulate/debug a circuit at 1/10 the speed
  - instead of 1/100000 for software simulation

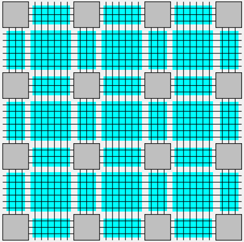
- Other applications:
  - Small series (cheaper than designing a chip)
  - Market locking (Cisco)
    - faster to market than founding a chip
    - product out before standard is finalized, then upgraded on the field

FPGAs

- FPGAs are mass-produced VLSI chips designed to emulate arbitrary logic circuits
- Mostly used for rapid prototyping
  - Simulate/debug a circuit at 1/10 the speed
  - instead of 1/100000 for software simulation
- Other applications:
  - Small series (cheaper than designing a chip)
  - Market locking (Cisco)
    - faster to market than founding a chip
    - product out before standard is finalized, then upgraded on the field
- Can we use these chips as programmable co-processors?

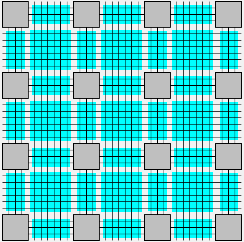
Basic FPGA structure

Overal view



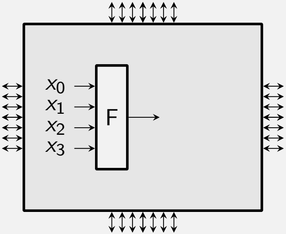
Basic FPGA structure

Overal view



- Cell: configurable logic blocks

Content of one block

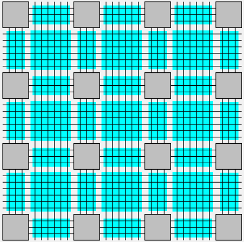


- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

filled with an arbitrary truth table

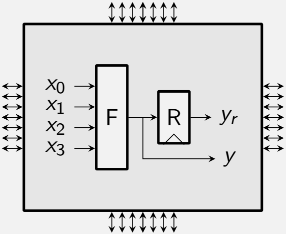
Basic FPGA structure

Overal view



- Cell: configurable logic blocks

Content of one block



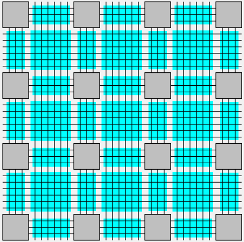
- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

  filled with an arbitrary truth table

- Memory: 1-bit register

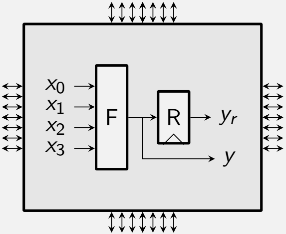
Basic FPGA structure

Overal view



- Cell: configurable logic blocks
- Configurable routing
  - need random access here

Content of one block



- Logic: Look-Up Table F
  - 4 inputs,
  - 1 output

  filled with an arbitrary truth table
- Memory: 1-bit register

Two moments in the life of an FPGA

Configuration time (a few ms)

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

a program == a lot of configuration bits

Two moments in the life of an FPGA

Configuration time (a few ms)

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

a 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 (a few ms)

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

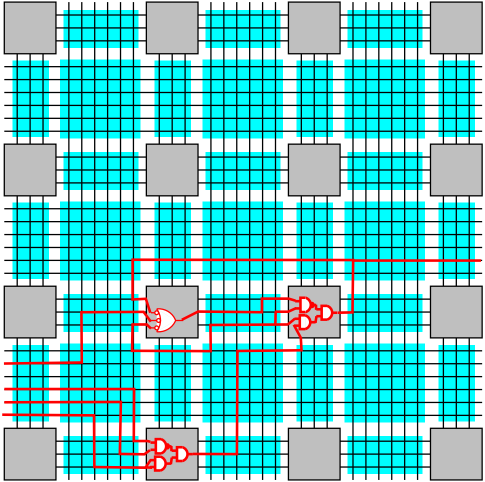
a program == a lot of configuration bits

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.

A configured FPGA



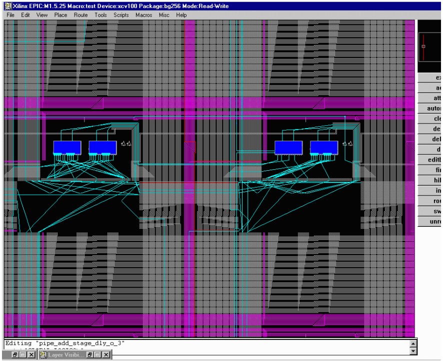
Also known as **reconfigurable circuits**

used for **reconfigurable computing**

Compared to ASIC, 1/10th the speed

Why?

- Most of the silicon is dedicated to programmable routing



- Cost in area, but also delay: many transistors on each wire
- “Customers buy logic, but they pay for routing” (Langhammer)
- And it gets worse (Rent’s law)

Rent's law?

Yet another experimental law

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

- more than proportional to n, the diameter itself,
- 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, etc.

Replace "circuit" with "city" and "wires" with "citizens each morning" and you have the explanation of the Hopeless Universal Trafic Jam in expanding cities.

Programming FPGAs

Current mainstream design-flow is hardware-like:

You don't write a program, you design a circuit

Programming FPGAs

Current mainstream design-flow is hardware-like:

You don't write a program, you design a circuit

Design tools are inherited from those of VLSI circuits

Programming FPGAs

Current mainstream design-flow is hardware-like:

You don't write a program, you design a circuit

Design tools are inherited from those of VLSI circuits

This is the main challenge. Who wants 1-week compilation time?

Current generation of FPGAs

- Programmable cells got coarser
  - currently, clusters of 6-input LUT
  - up to 500,000 of them
  - tens of millions of equivalent logic gates

Current generation of FPGAs

- Programmable cells got coarser
  - currently, clusters of 6-input LUT
  - up to 500,000 of them
  - tens of millions of equivalent logic gates
- up to several tens of Mb of embedded memories
  - configurable in all sorts of ways
  - massive intra-FPGA bandwidth

Current generation of FPGAs

- Programmable cells got coarser
  - currently, clusters of 6-input LUT
  - up to 500,000 of them
  - tens of millions of equivalent logic gates
- up to several tens of Mb of embedded memories
  - configurable in all sorts of ways
  - massive intra-FPGA bandwidth
- up to several thousand DSP blocks
  - MAC unit (typically 18x18+40 bits, **integer**)
  - Also flexible (may be split into 9x9 multipliers, etc)
- and many other features irrelevant to this talk
  - High-speed configurable I/Os: 1500 pages of documentation

Summary: *Finest Programmable Granularity Around*

- Programmable chips, but programmed in VHDL

Summary: *Finest Programmable Granularity Around*

- Programmable chips, but programmed in VHDL
- Efficient for integer addition and integer multiplication

Summary: *Finest Programmable Granularity Around*

- Programmable chips, but programmed in VHDL
- Efficient for integer addition and integer multiplication
- Efficient for implementing tables

Summary: *Finest Programmable Granularity Around*

- Programmable chips, but programmed in VHDL
- Efficient for integer addition and integer multiplication
- Efficient for implementing tables
- All this at the granularity of the bit
  - 17-bit adder
  - multiplier of 7-bit numbers by 56-bit numbers
  - table of 2^7 entries of 17 bits each
  - ...

Summary: *Finest Programmable Granularity Around*

- Programmable chips, but programmed in VHDL
- Efficient for integer addition and integer multiplication
- Efficient for implementing tables
- All this at the granularity of the bit
  - 17-bit adder
  - multiplier of 7-bit numbers by 56-bit numbers
  - table of 2^7 entries of 17 bits each
  - ...
- And of course glue logic

Conclusion

Computing at large

Think parallel, but then think circuit.

The “no killer app for FPGAs” theorem

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

The “no killer app for FPGAs” theorem

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

Theorem: we’ll wait forever.

The "no killer app for FPGAs" theorem

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

Theorem: we'll wait forever.

Proof: When such an application pops up,

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

The “no killer app for FPGAs” theorem

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

Theorem: we'll wait forever.

Proof: When such an application pops up,

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

The killer feature of FPGAs is flexibility

To exploit it, we need to design better tools...