Functioning Hardware from Functional Specifications

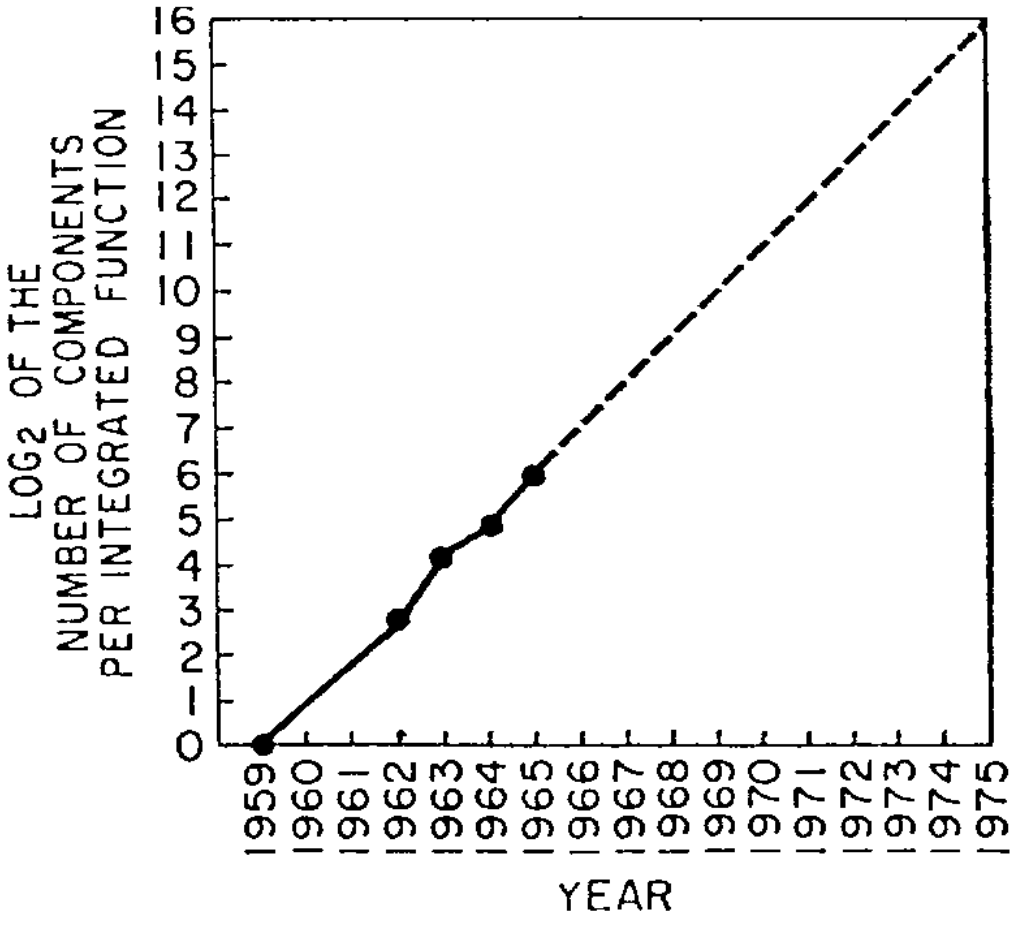
Stephen A. Edwards

Columbia University

ACSD Keynote, Tunis, June 26, 2014

Where’s my 10 GHz processor?

Moore's Law: Transistors Shrink and Get Cheaper

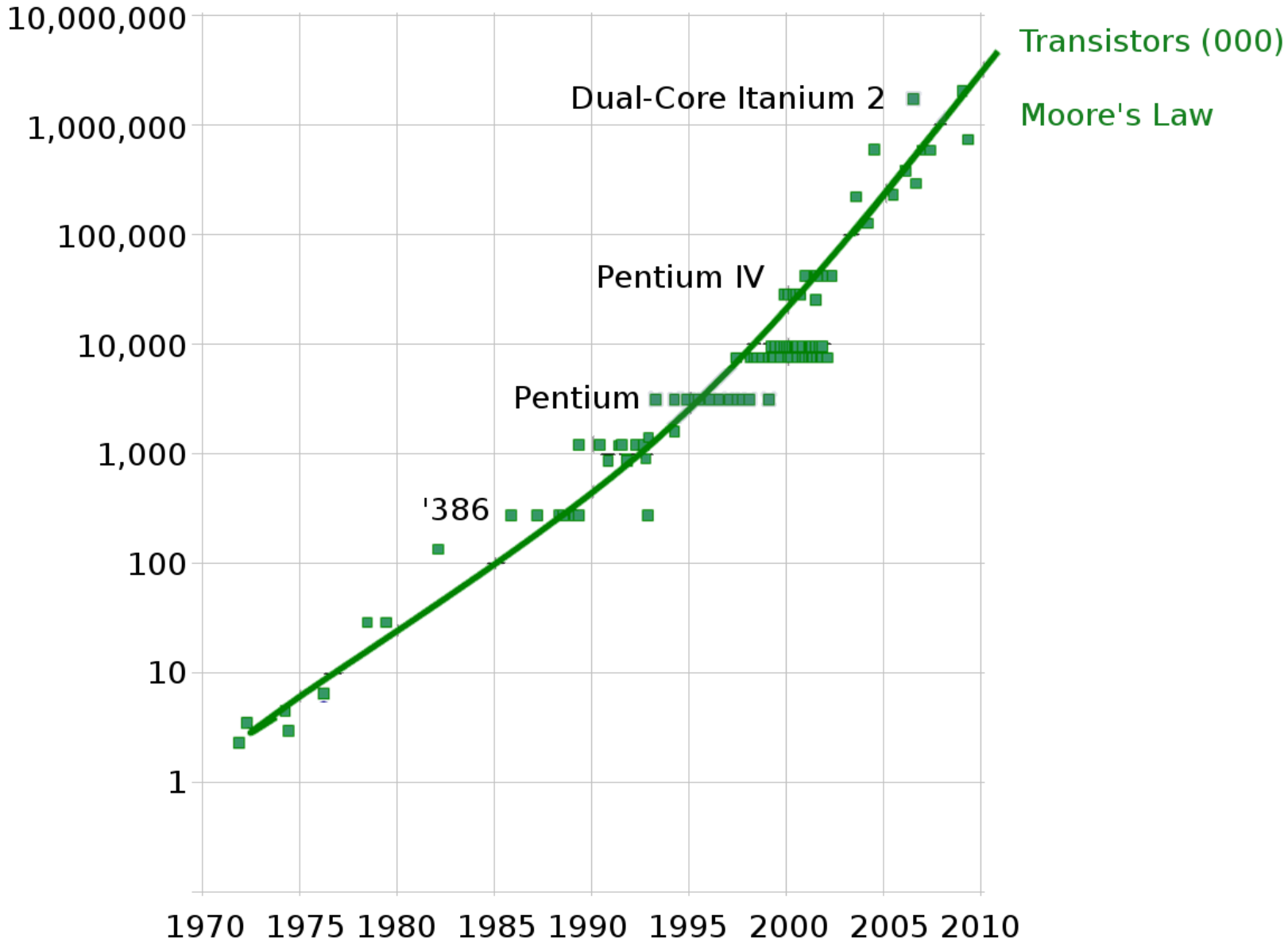


"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."

Closer to every 24 months

Gordon Moore, *Cramming More Components onto Integrated Circuits*, Electronics, 38(8) April 19, 1965.

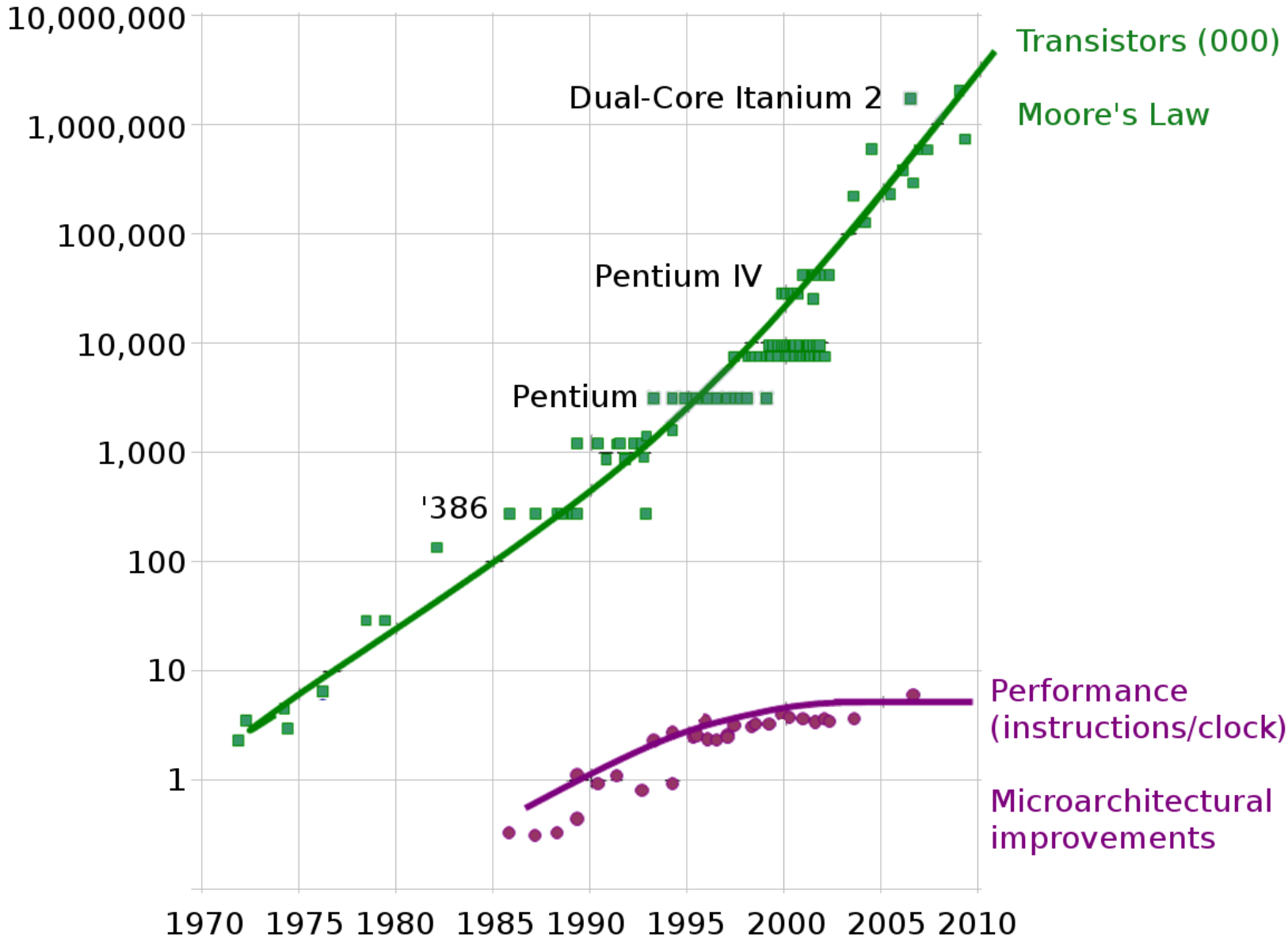
Intel CPU Trends



Sutter, *The Free Lunch is Over*, DDJ 2005.

Data: Intel, Wikipedia, K. Olukotun

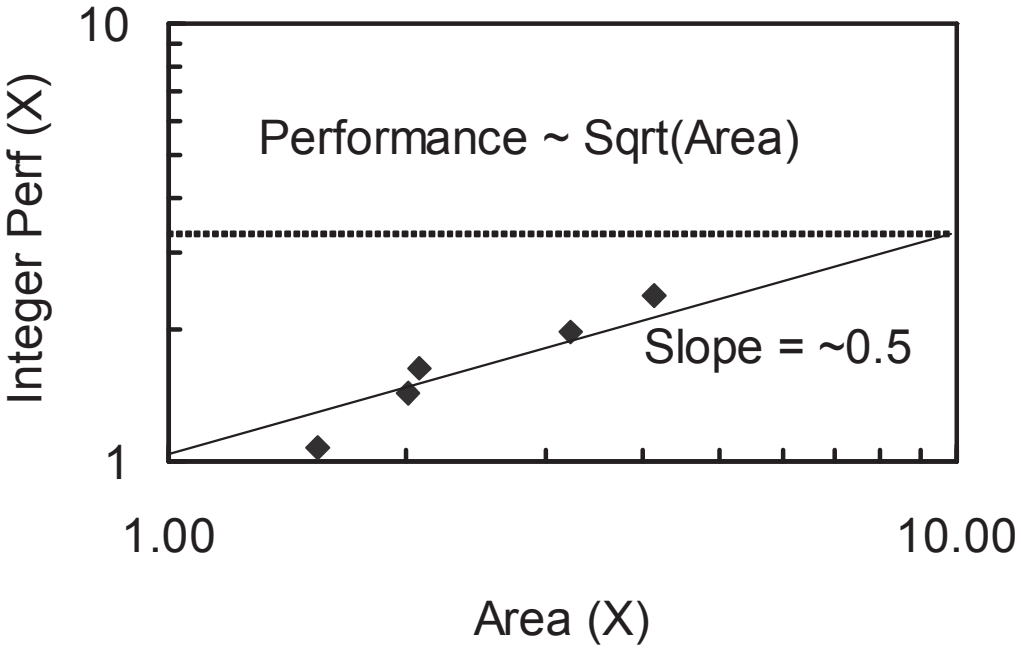
Intel CPU Trends



Sutter, *The Free Lunch is Over*, DDJ 2005.

Data: Intel, Wikipedia, K. Olukotun

Pollack's Rule: Diminishing Returns for Processors



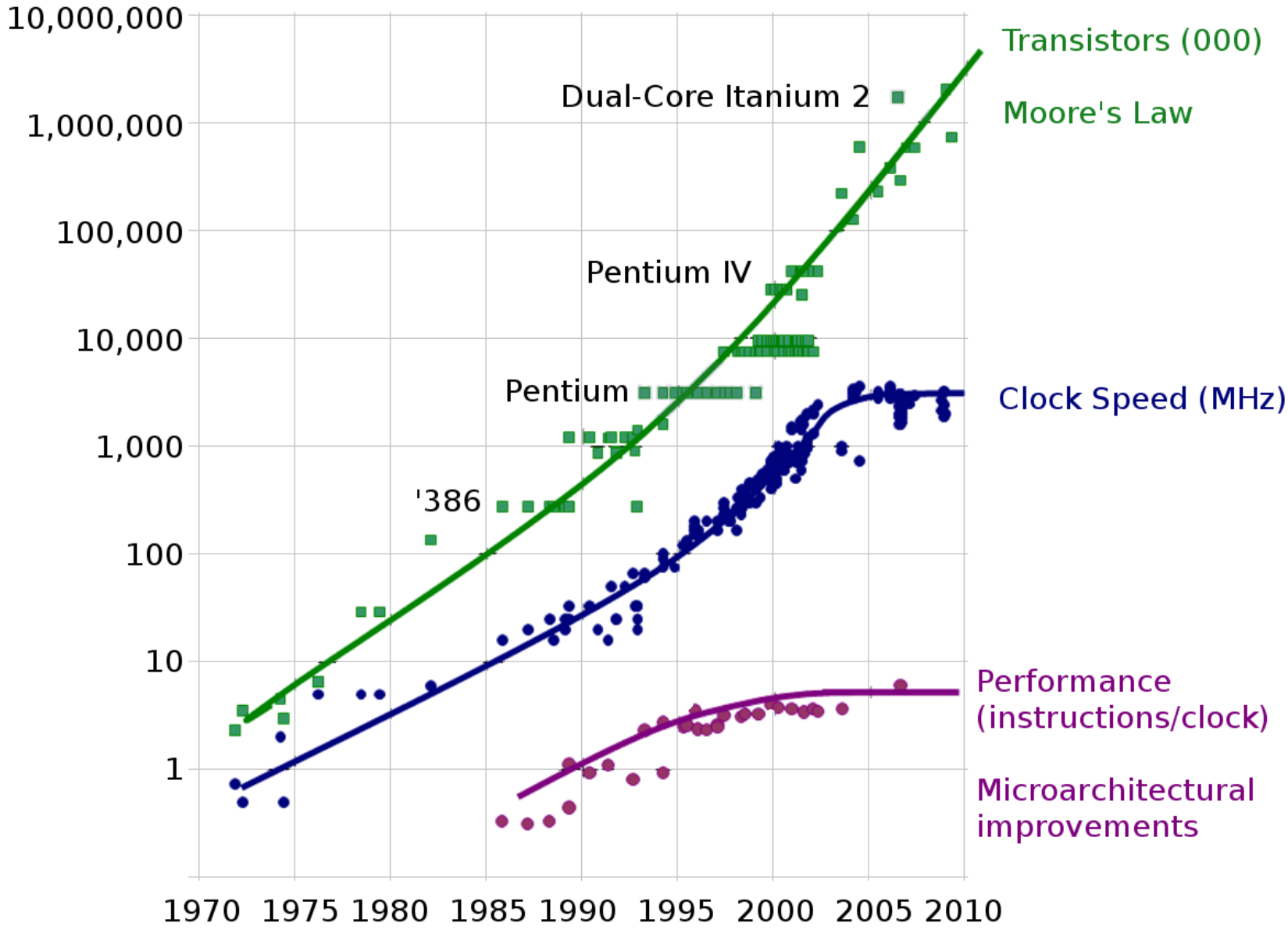
Single-threaded processor performance grows with the square root of area.

It takes
4× the transistors to give
2× the performance.

Fred J. Pollack, MICRO 1999 keynote.

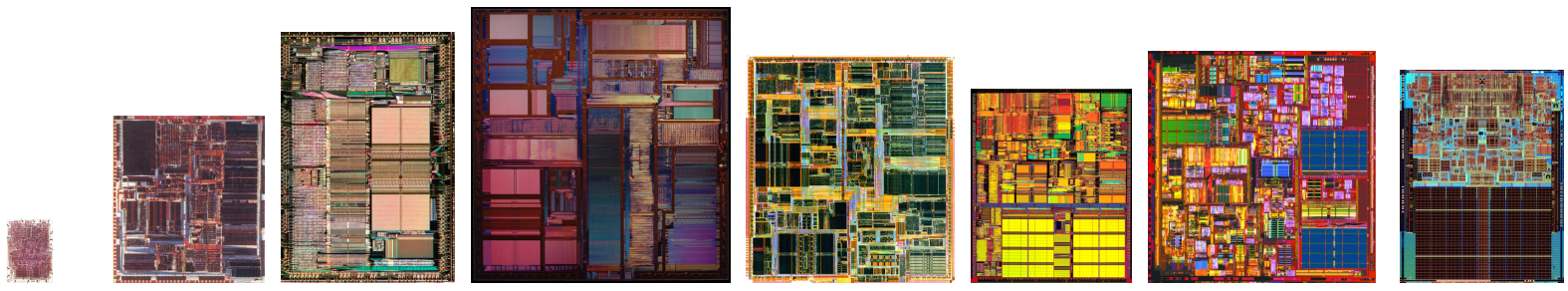
Graph from Borkar, DAC 2007.

Intel CPU Trends



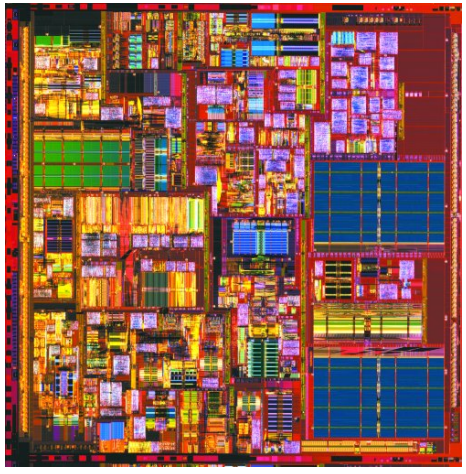
Sutter, *The Free Lunch is Over*, DDJ 2005. Data: Intel, Wikipedia, K. Olukotun

Intel Processors to Scale

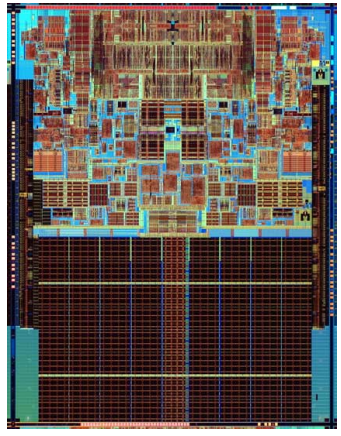


4004	’386	’486	Pentium	P II	P III	P IV	Core 2
1971	1987	1989	1993	1997	1999	2000	2006

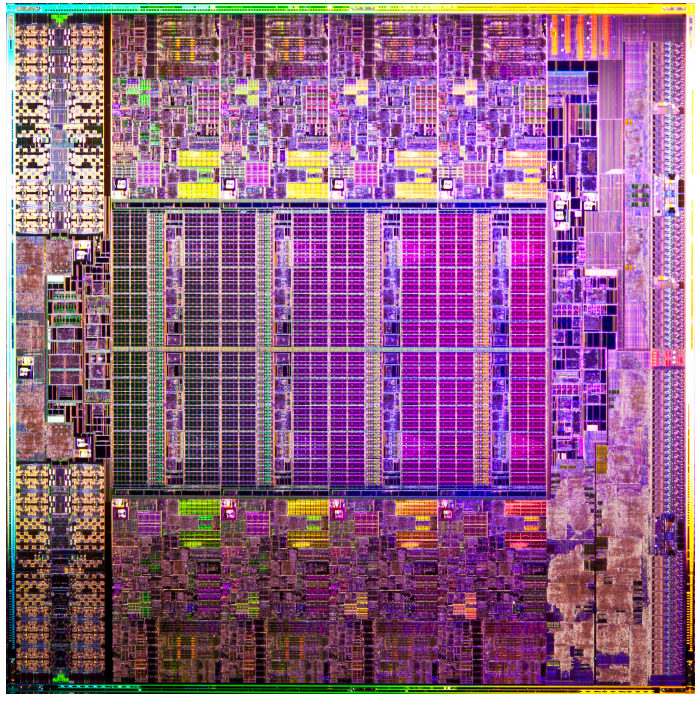
What Happened in 2005?



Pentium 4
2000
1 core
Transistors: 42 M

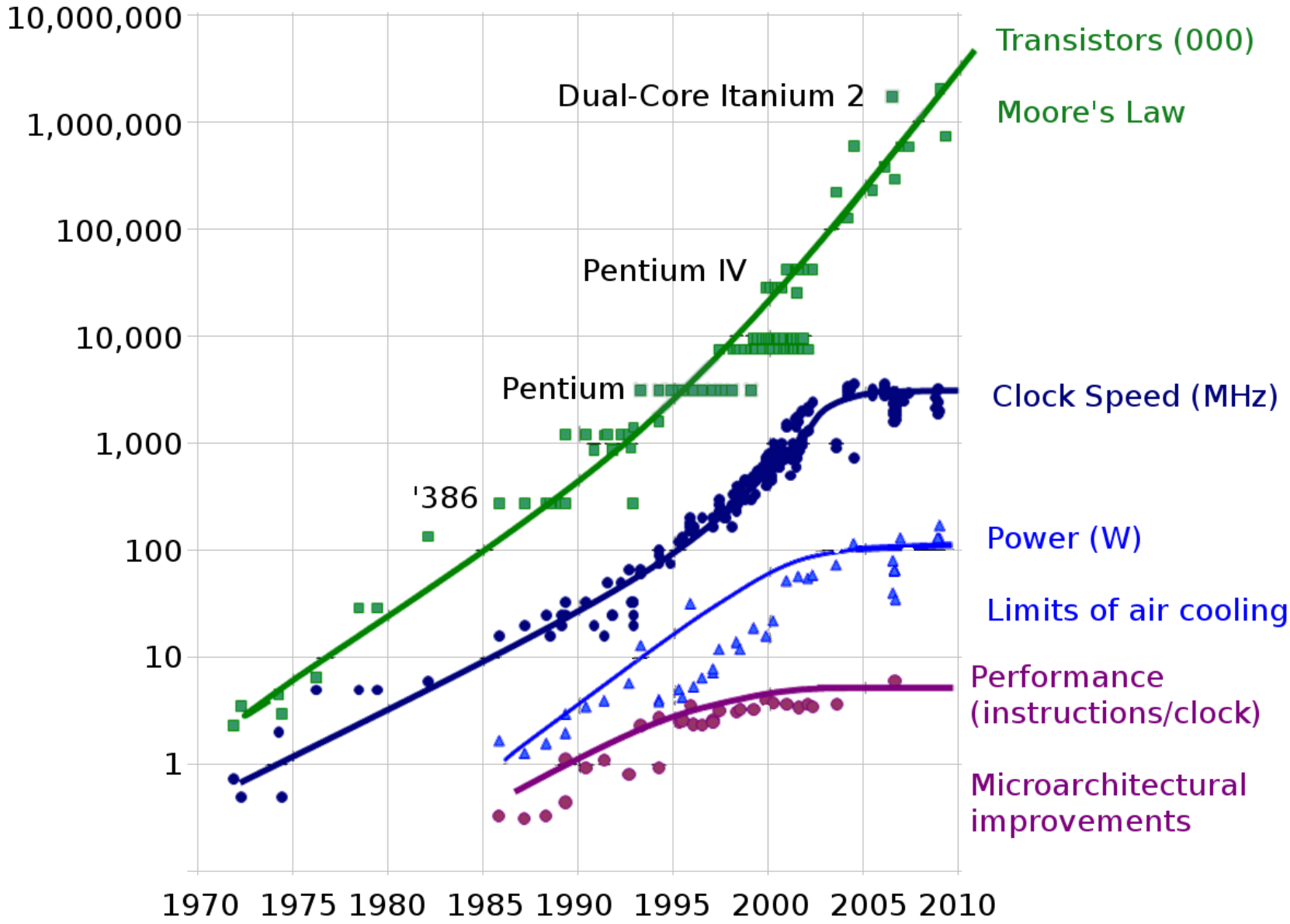


Core 2 Duo
2006
2 cores
291 M



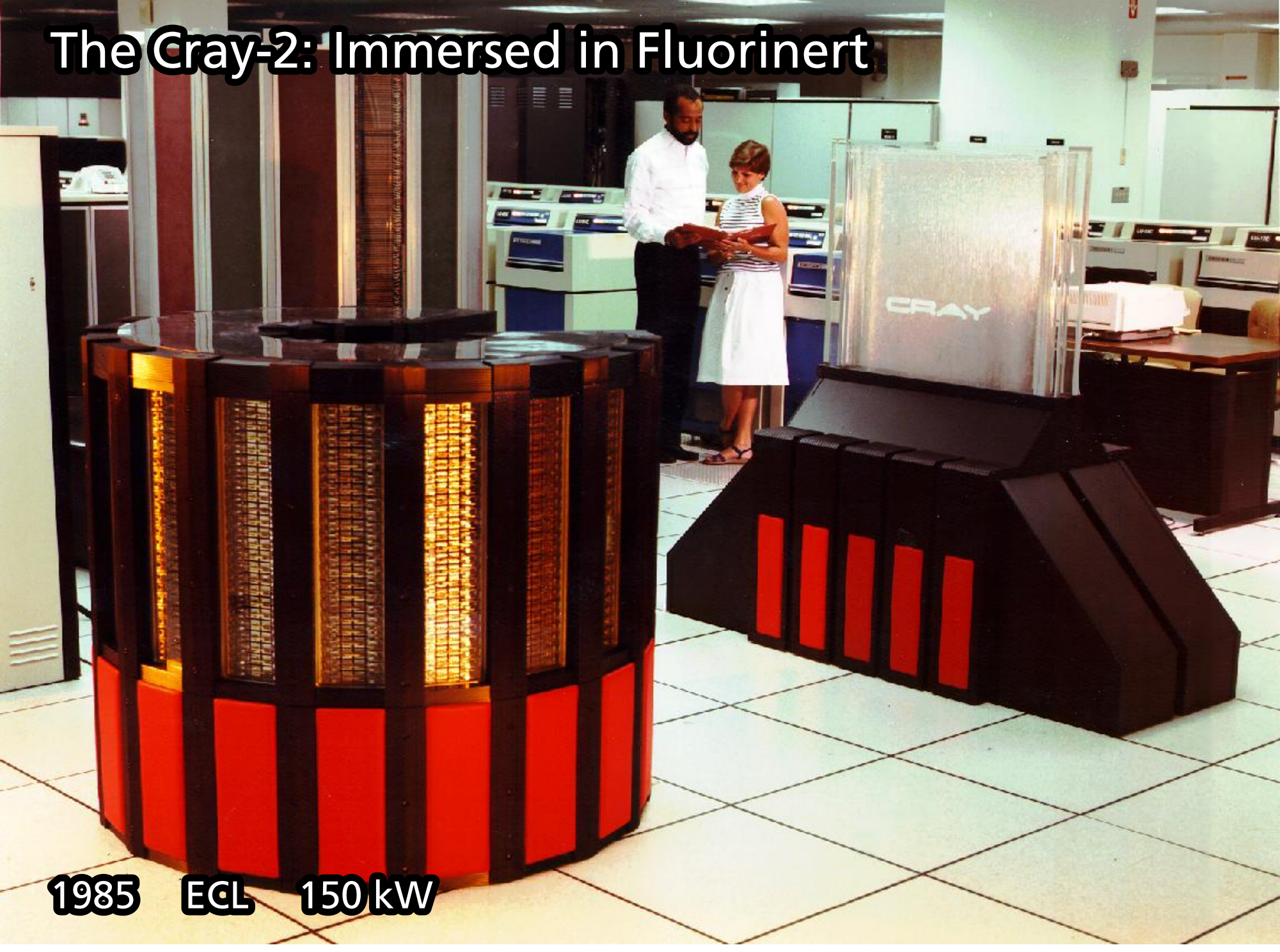
Xeon E5
2012
8 cores
2.3 G

Intel CPU Trends

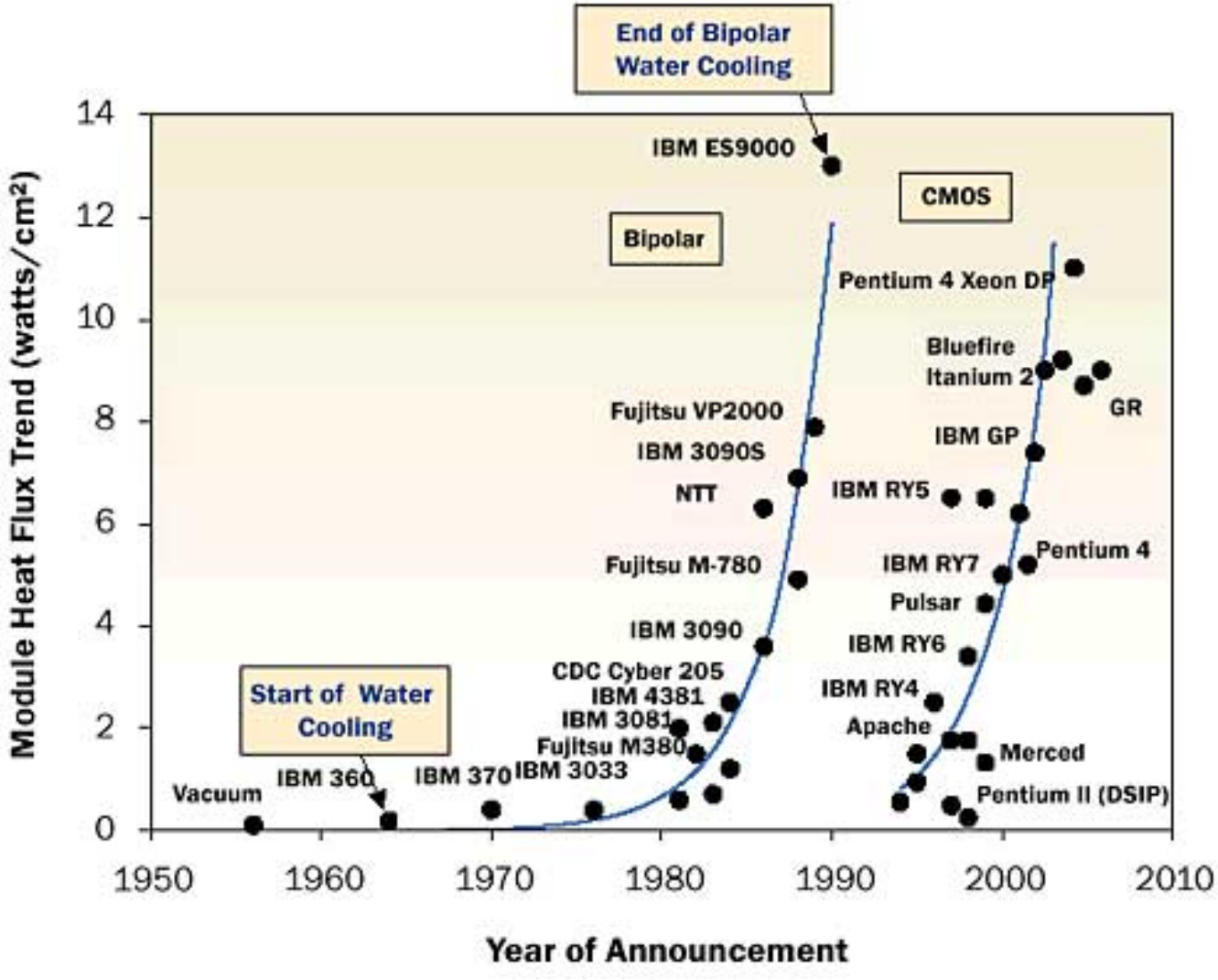


Sutter, *The Free Lunch is Over*, DDJ 2005.

Data: Intel, Wikipedia, K. Olukotun


The Cray-2: Immersed in Fluorinert
CRAY
1985 ECL 150 kW

Heat Flux in IBM Mainframes: A Familiar Trend



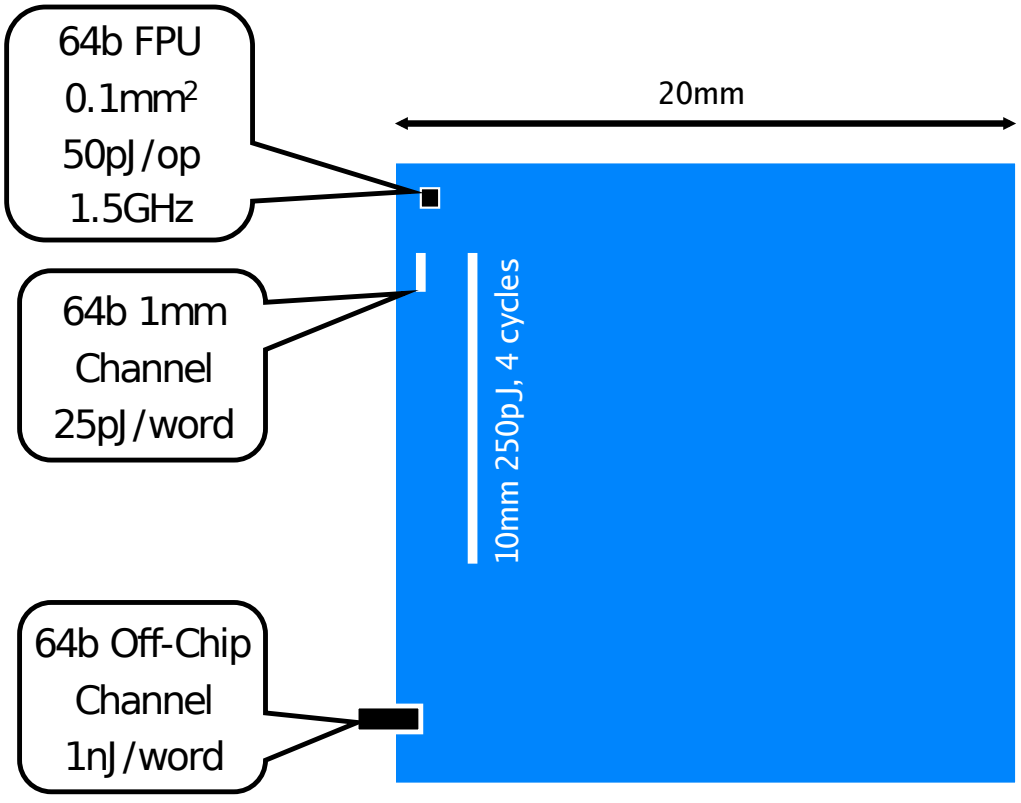
Schmidt. *Liquid Cooling is Back*. Electronics Cooling. August 2005.

Liquid Cooled Apple Power Mac G5

2004 CMOS 1.2 kW

Where’s all that power going?
What can we do about it?

Dally: Calculation Cheap; Communication Costly



"Chips are power limited and most power is spent moving data

Performance = Parallelism

Efficiency = Locality

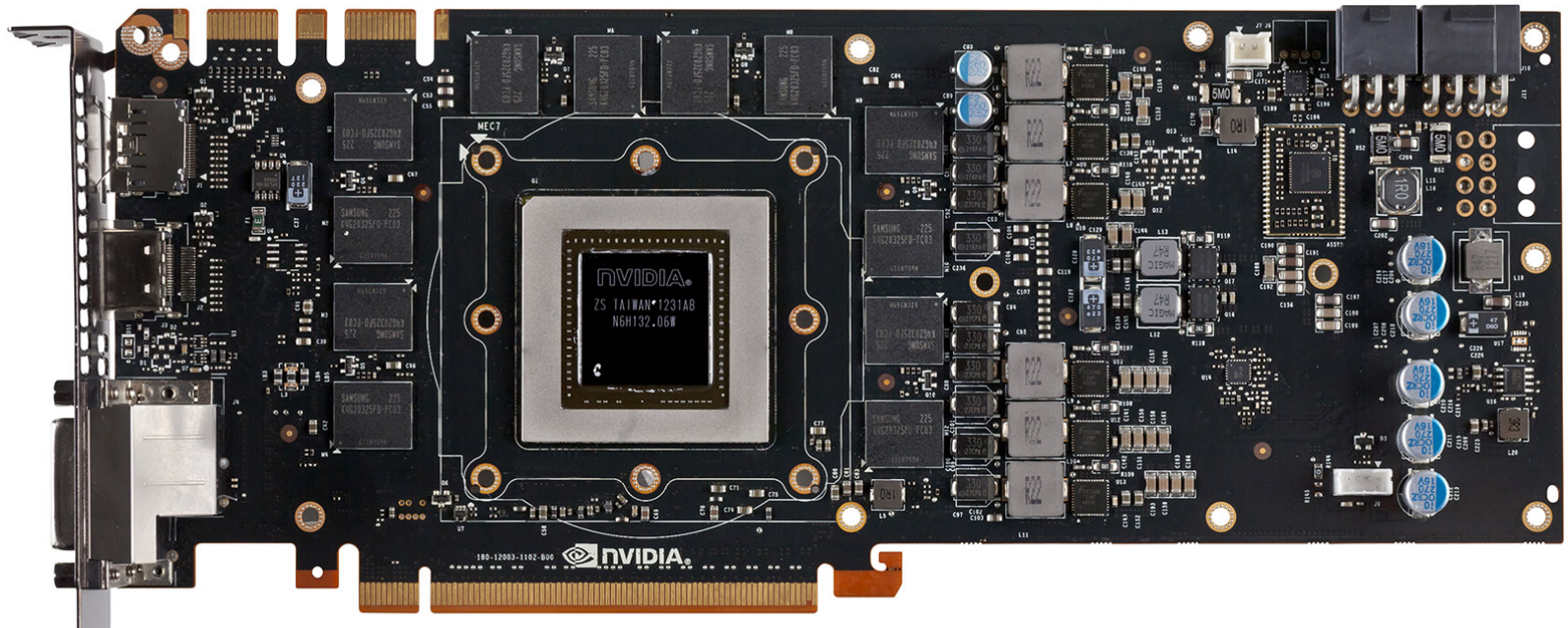
Bill Dally's 2009 DAC Keynote, *The End of Denial Architecture*

Parallelism for Performance; Locality for Efficiency

Dally: "Single-thread processors are in denial about these two facts"

We need
different programming paradigms
and
different architectures
on which to run them.

Massive On-Chip Parallelism: The NVIDIA GTX Titan



The NVIDIA GTX Titan/GK110 GPU

Speed	4.5	TFLOP/s
Frequency	876	MHz
Power	250	W
Transistors	7	G
Area	561	mm^2
Cores	2688	

Memory

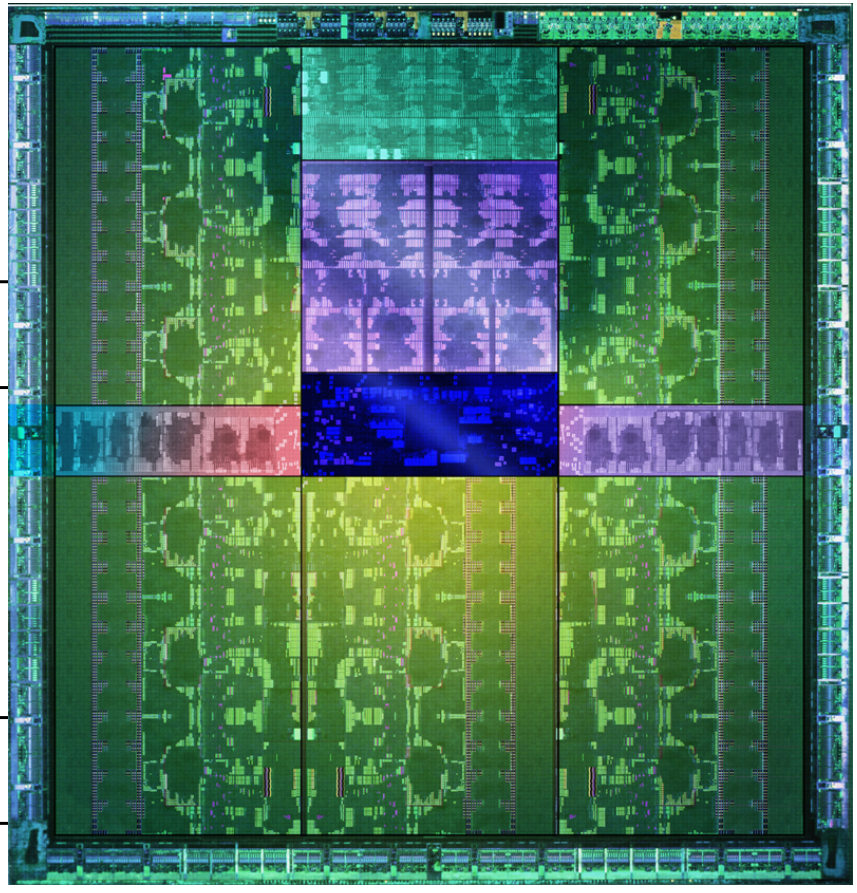
Size	6	Gb
Bus width	384	bits
Clock	1.5	GHz
Bandwidth	288	Gb/s

Price

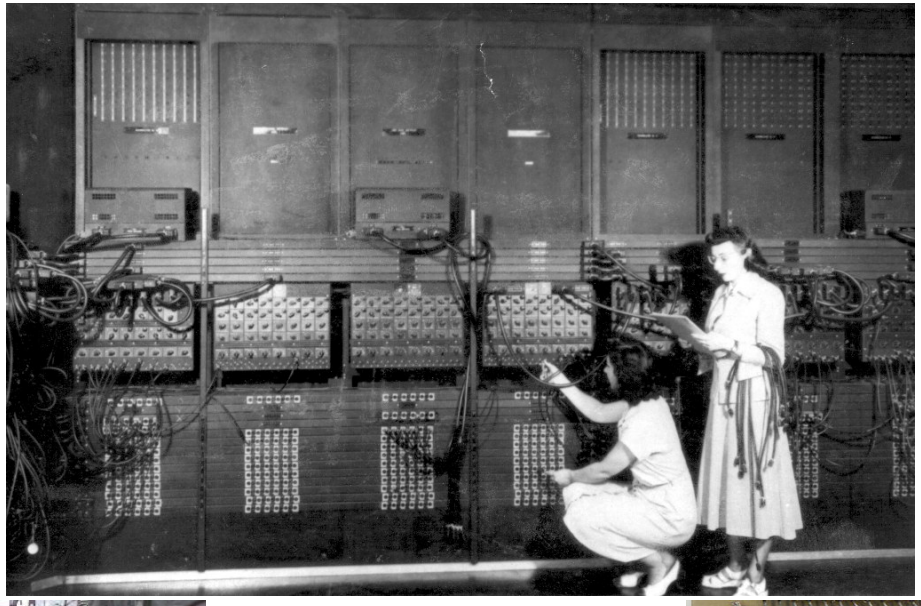
$1000

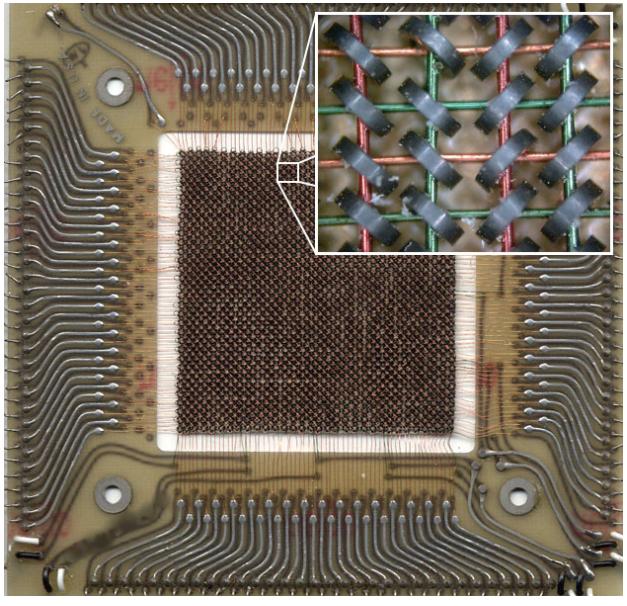
€730

TND 1660



The Future is Wires and Memory



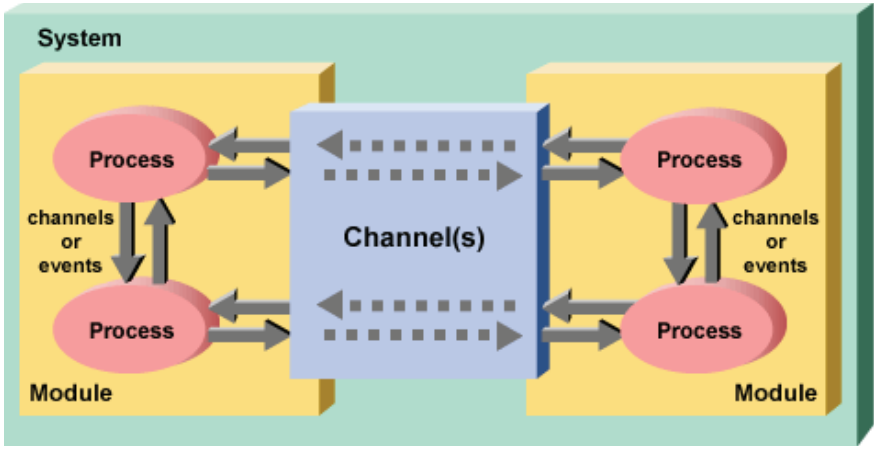


How best to use all those transistors?


Dark Silicon

Xilinx's Vivado (Was xPilot, AutoESL)

- *SSDM* (System-level Synthesis Data Model)
  - Hierarchical netlist of concurrent processes and communication channels

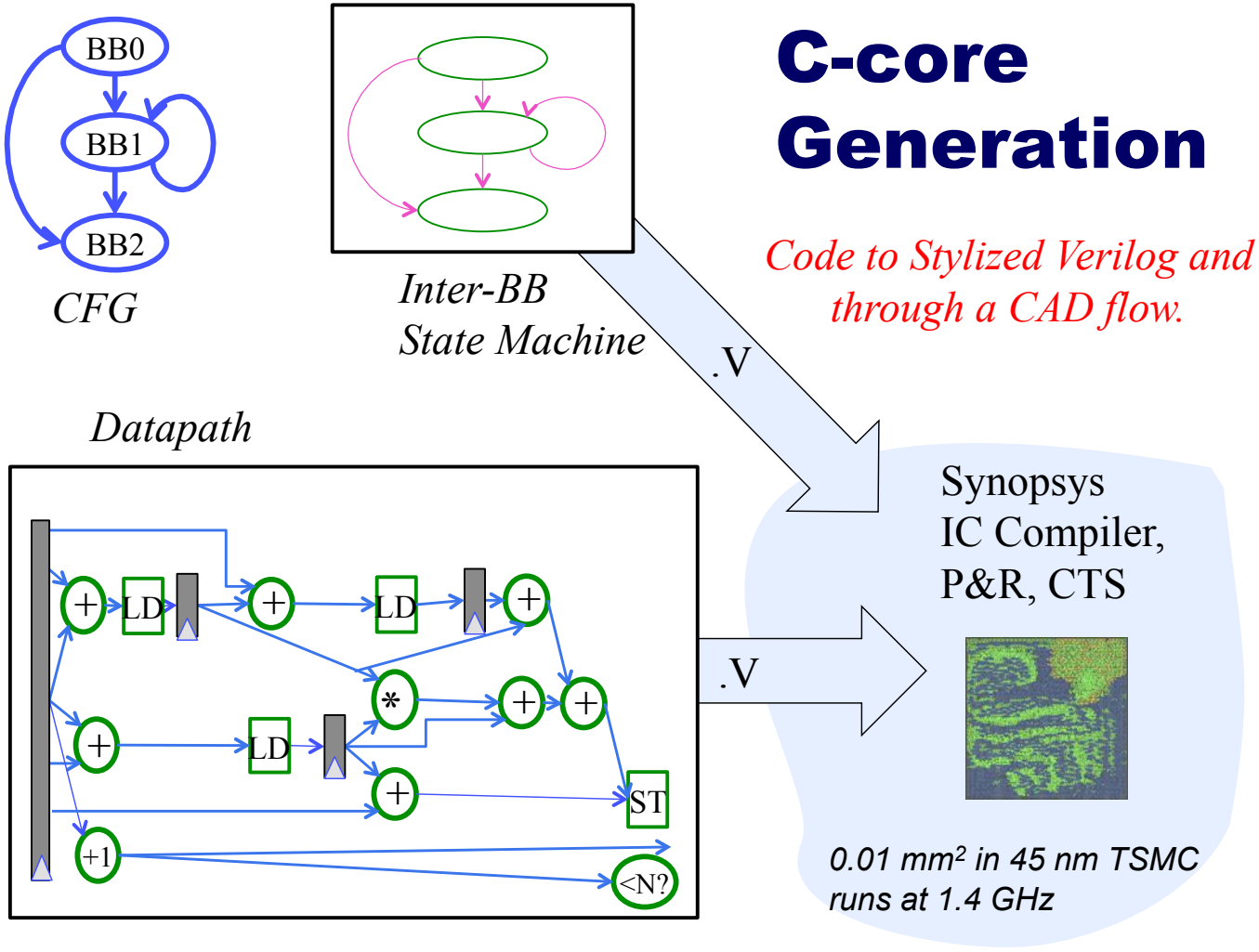


  - Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
    - Port / IO interfaces, bit-vector manipulations, cycle-level notations

SystemC input; classical high-level synthesis for processes

Jason Cong et al. ISARS 2005

Taylor and Swanson's Conservation Cores



Custom datapaths, controllers for loop kernels; uses existing memory hierarchy

Swanson, Taylor, et al. *Conservation Cores*. ASPLOS 2010.

Bacon et al.'s Liquid Metal

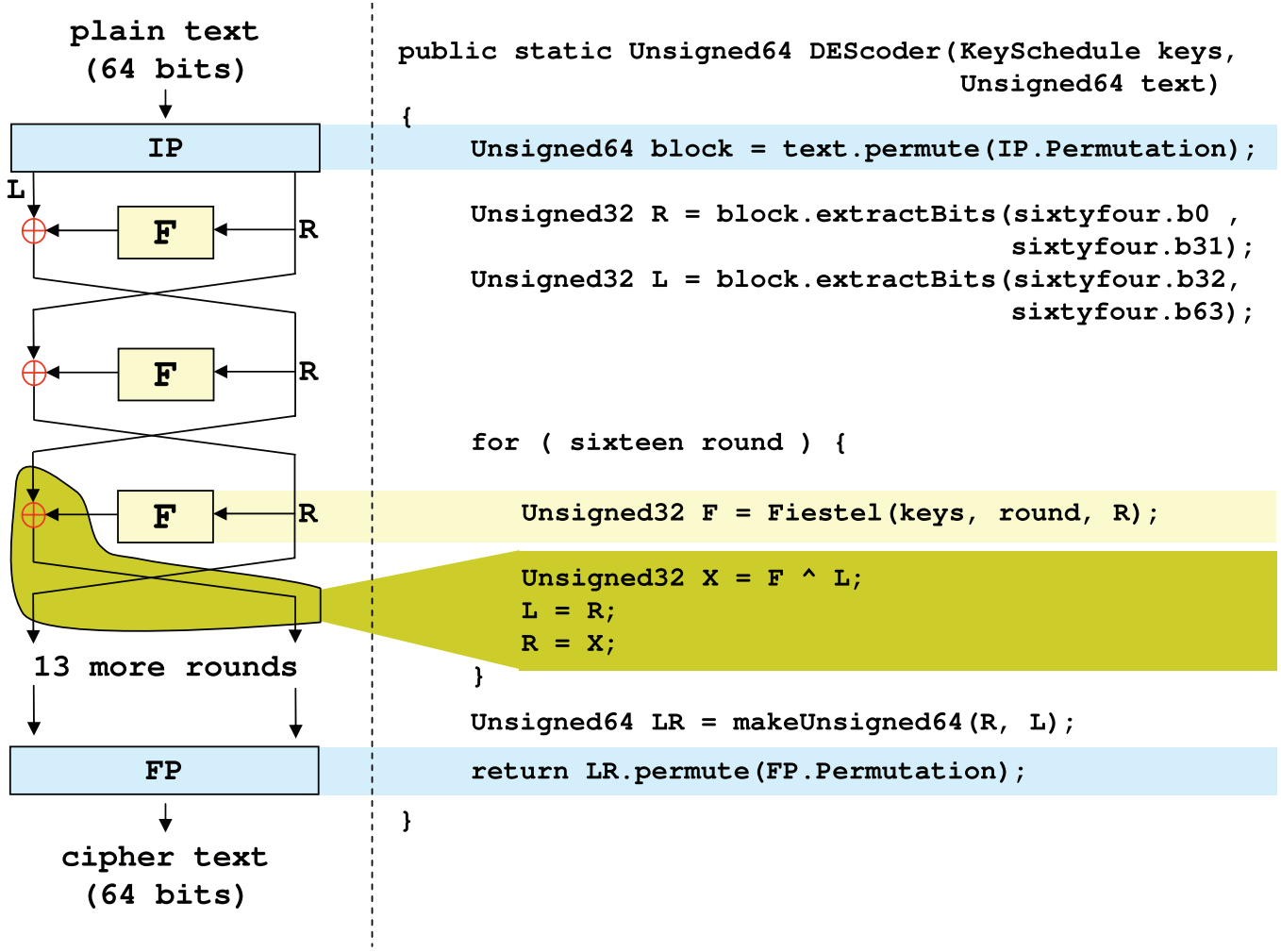


Fig. 2. Block level diagram of DES and Lime code snippet

JITting Lime (Java-like, side-effect-free, streaming) to FPGAs

Huang, Hormati, Bacon, and Rabbah, *Liquid Metal*, ECOOP 2008.

Goldstein et al.'s Phoenix

```
int squares()
{
  int i = 0,
      sum = 0;

  for (;i<10;i++)
      sum += i*i;
  return sum;
}
```

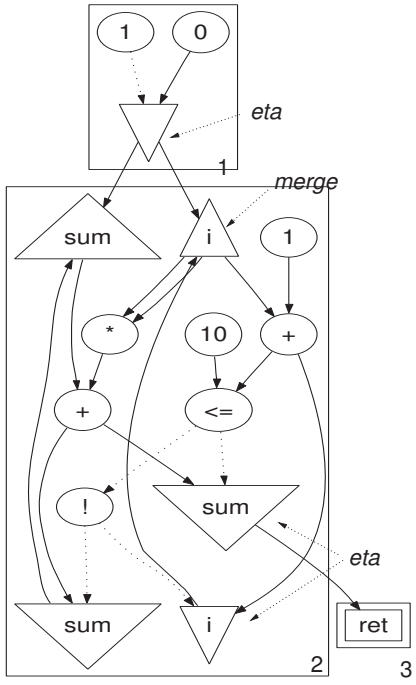


Figure 3: *C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)*

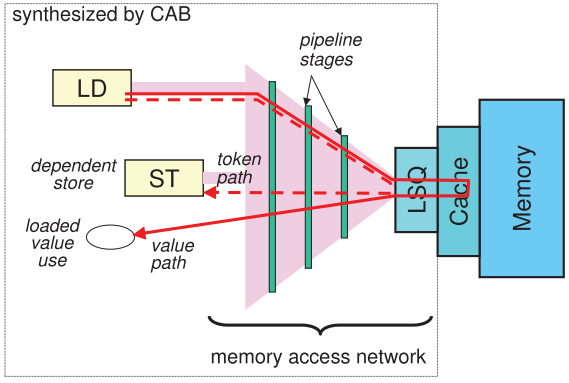


Figure 8: *Memory access network and implementation of the value and token forwarding network. The* LOAD *produces a data value consumed by the oval node. The* STORE *node may depend on the load (i.e., we have a token edge between the* LOAD *and the* STORE*, shown as a dashed line). The token travels to the root of the tree, which is a load-store queue (LSQ).*

C to asynchronous logic, monolithic memory

Budiu, Venkataramani, Chelcea and Goldstein, *Spatial Computation*, ASPLOS 2004.

Ghica et al.'s Geometry of Synthesis

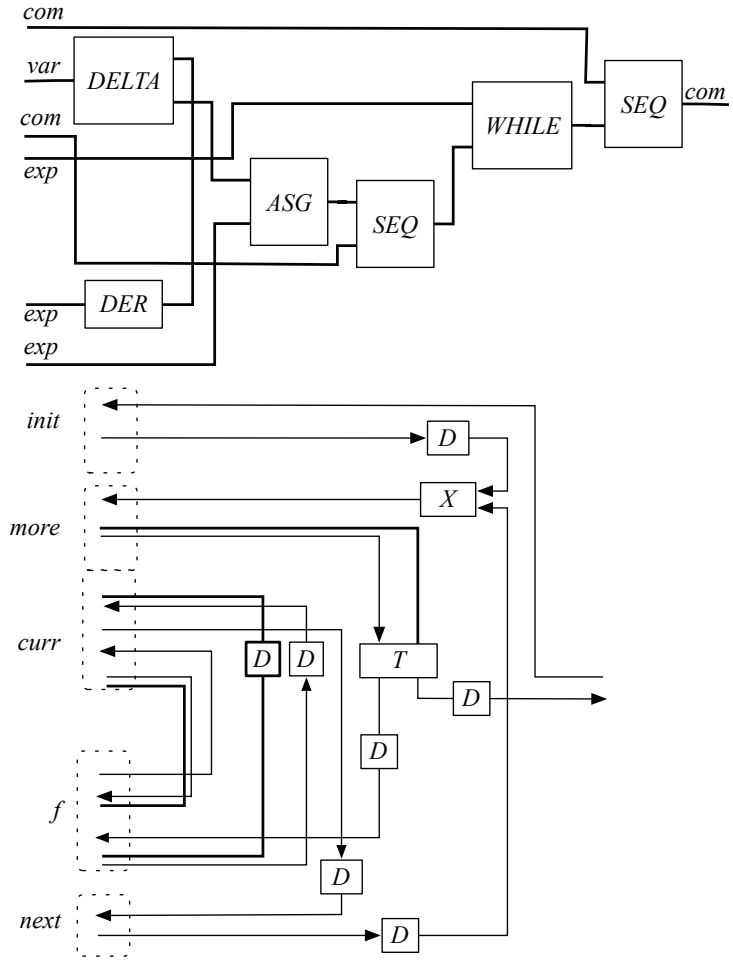


Figure 1. In-place map schematic and implementation

Algol-like imperative language to handshake circuits

Ghica, Smith, and Singh. *Geometry of Synthesis IV*, ICFP 2011

Greaves and Singh's Kiwi

```
public static void SendDeviceID()
{ int deviceID = 0x76;
  for (int i = 7; i > 0; i--)
  { scl = false;
    sda_out = (deviceID & 64) != 0;
    Kiwi.Pause(); // Set it i-th bit of the device ID
    scl = true; Kiwi.Pause(); // Pulse SCL
    scl = false; deviceID = deviceID << 1;
    Kiwi.Pause();
  }
}
```

C# with a concurrency library to FPGAs

Greaves and Singh. *Kiwi*, FCCM 2008

Arvind, Hoe, et al.'s Bluespec

GCD Mod Rule

$\mathsf{Gcd}(a, b)$ *if* $(a \geq b) \wedge (b \neq 0) \rightarrow \mathsf{Gcd}(a-b, b)$

GCD Flip Rule

$\mathsf{Gcd}(a, b)$ *if* $a < b \rightarrow \mathsf{Gcd}(b, a)$

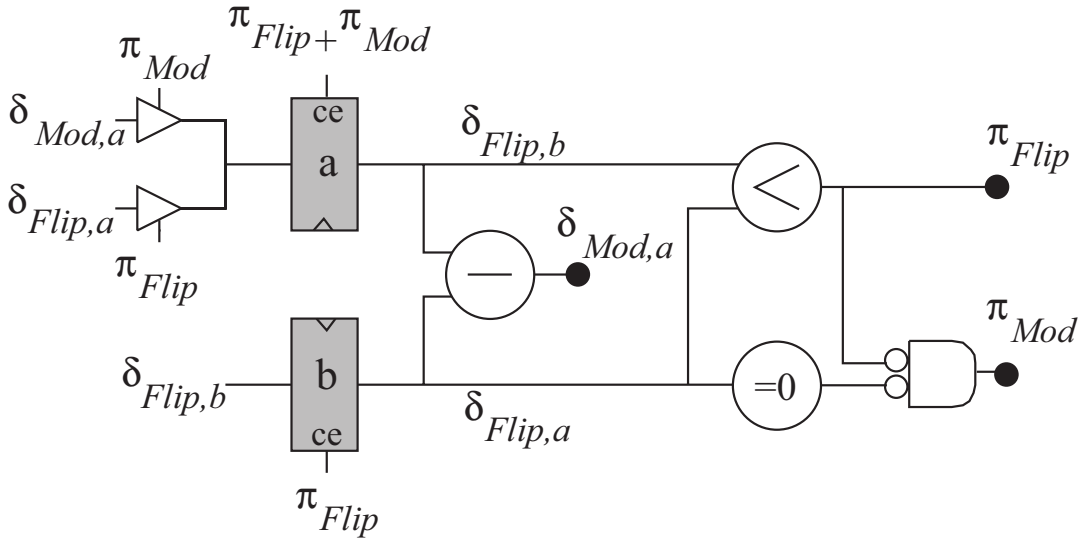


Figure 1.3 Circuit for computing $\mathsf{Gcd}(a, b)$ from Example 1.

Guarded commands and functions to synchronous logic

Hoe and Arvind, *Term Rewriting*, VLSI 1999

Sheeran et al.'s Lava

```
bfly :: CmplxArithmetic m
     => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
  do o1 <- csubtract (i1, i2)
     o2 <- cplus (i1, i2)
     return [o1, o2]

bflys :: CmplxArithmetic m
      => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
  riffle >-> raised n two bfly >-> unriffle
```

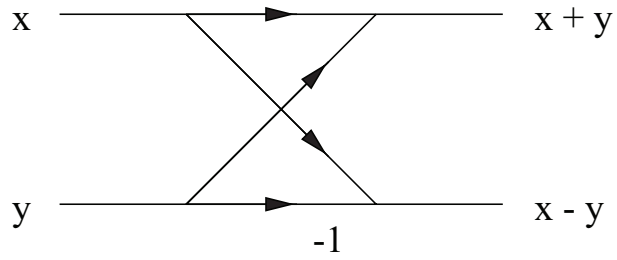


Figure 9: A butterfly

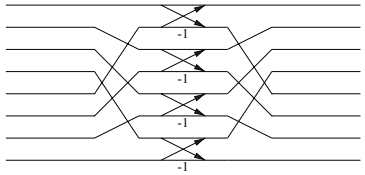


Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures

Bjesse, Claessen, Sheeran, and Singh. *Lava*, ICFP 1998

Kuper et al.'s CλaSH

$$fir\ (State\ (xs, hs))\ x =$$
$$(State\ (shiftInto\ x\ xs, hs), (x \triangleright xs) \bullet hs)$$

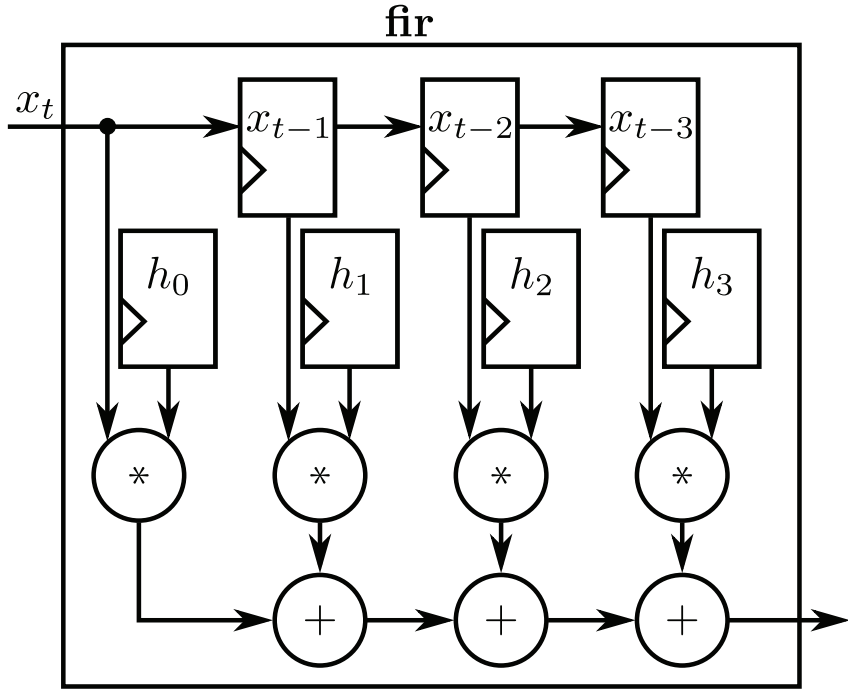


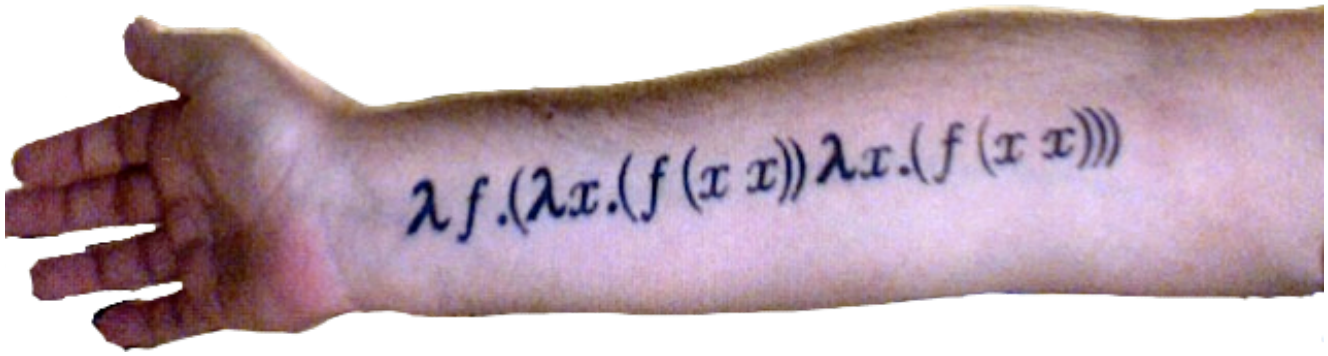
Fig. 6. 4-taps FIR Filter

More operational Haskell specifications of regular structures

Baaij, Kooijman, Kuper, Boeijink, and Gerards. *CλaSH*, DSD 2010

What am I doing about it?

Functional Programs to FPGAs

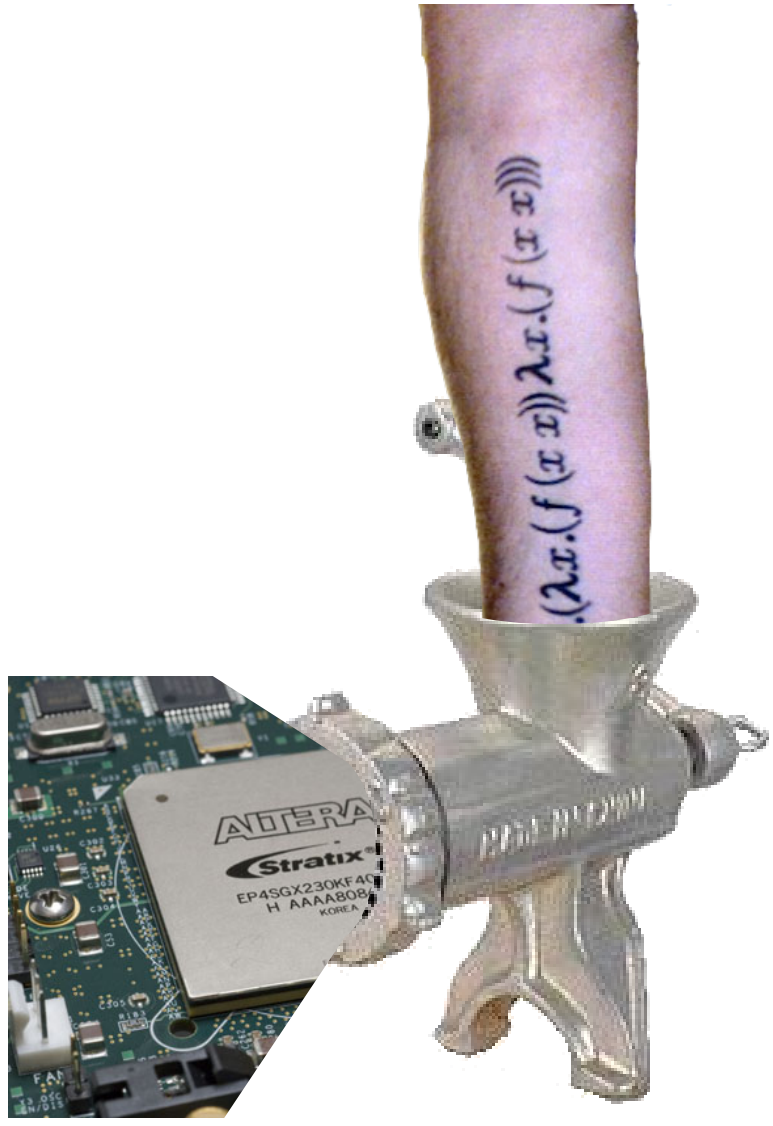


Functional Programs to FPGAs

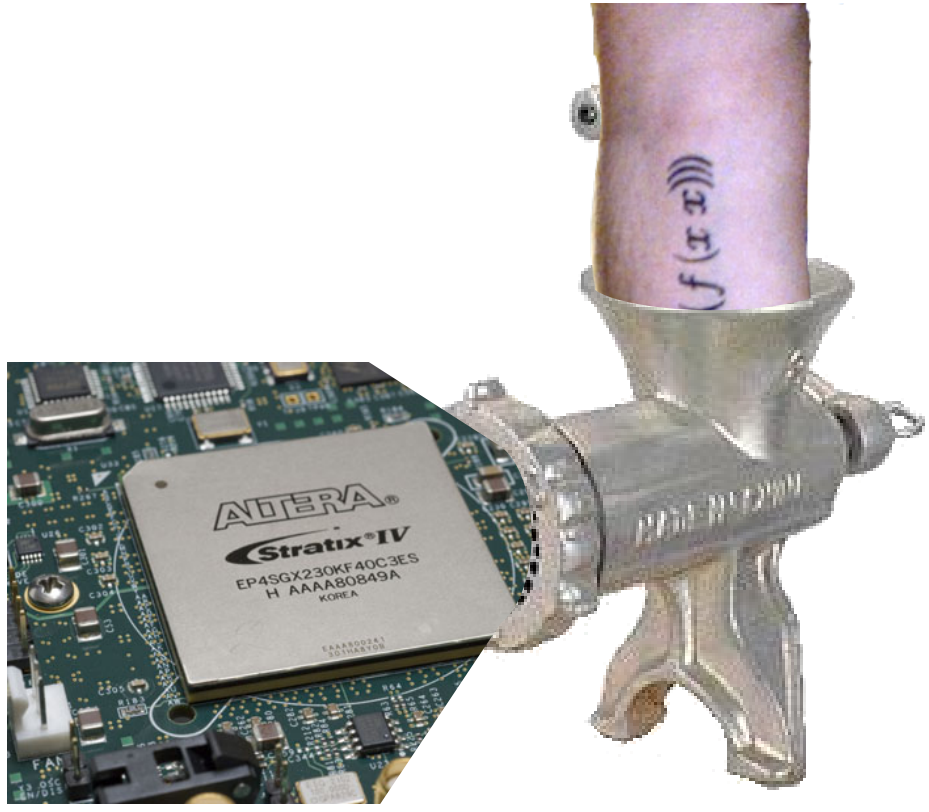
Functional Programs to FPGAs

Functional Programs to FPGAs

Functional Programs to FPGAs



Functional Programs to FPGAs



Functional Programs to FPGAs

More Motivation

More Motivation

abstraction

Future Languages

Higher-level languages

C et al.

gcc et al.

x86 et al.

Future ISAs

More hardware reality

today

time

More Motivation

abstraction

Future Languages

Higher-level languages

C et al.

gcc et al.

x86 et al.

A Functional IR

Future ISAs

FPGAs

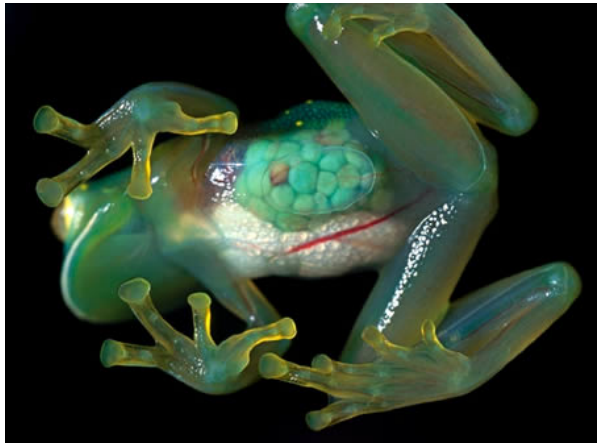
More hardware reality

today

time

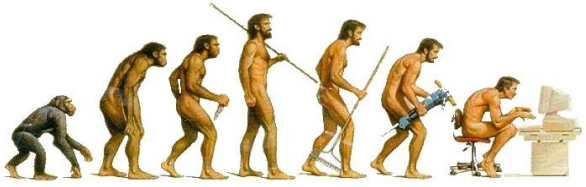
Why Functional Specifications?

- Referential transparency/side-effect freedom make formal reasoning about programs vastly easier
- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.
- Immutable data structures makes it vastly easier to reason about memory in the presence of concurrency



Why FPGAs?

- We do not know the structure of future memory systems
  Homogeneous/Heterogeneous?
  Levels of Hierarchy?
  Communication Mechanisms?



- We do not know the architecture of future multi-cores
  Programmable in Assembly/C?
  Single- or multi-threaded?

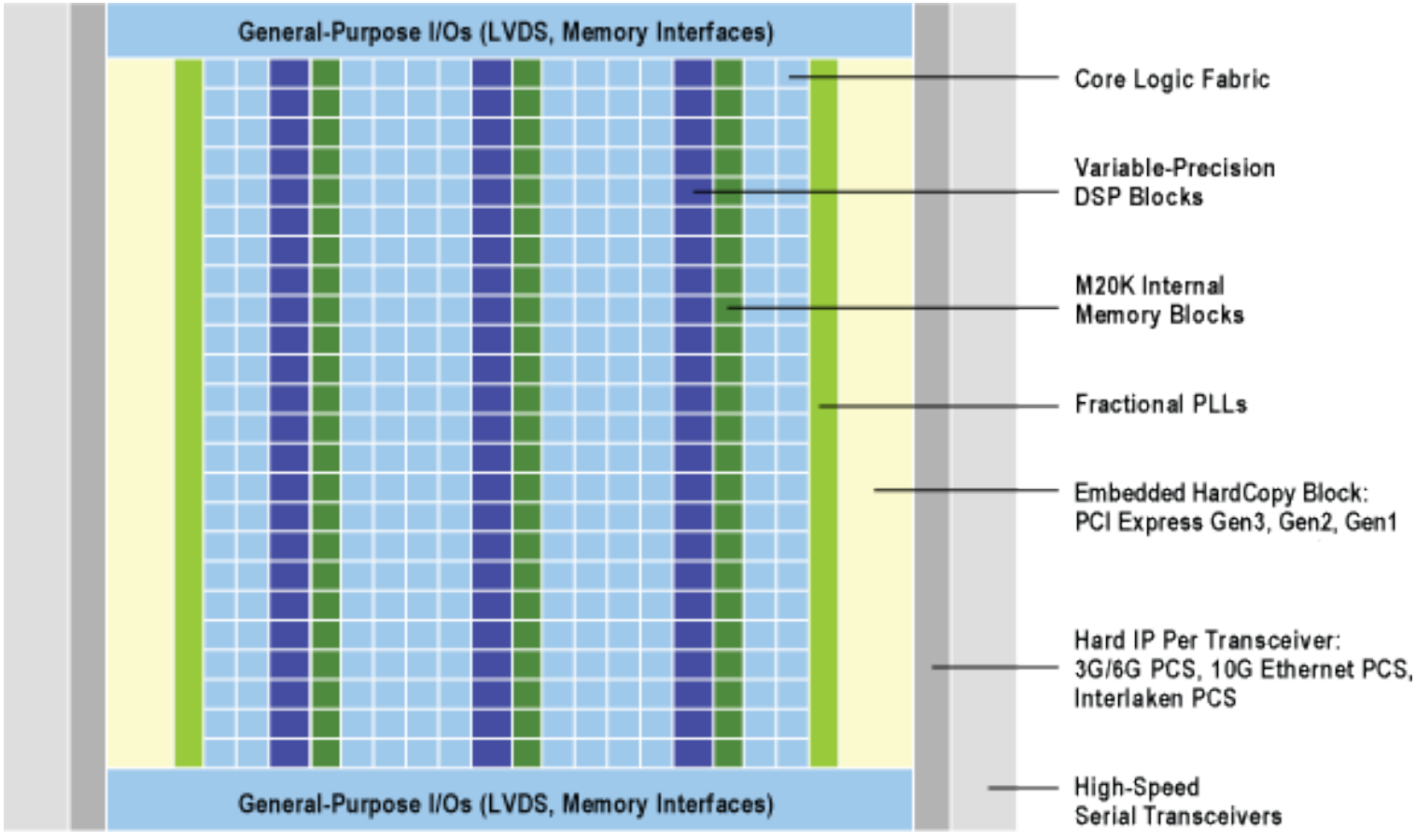
Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

A High-End FPGA: Altera's Stratix V

2500 dual-ported 2.5KB 600 MHz memory blocks; 6 Mb total

350 36-bit 500 MHz DSP blocks (MAC-oriented datapaths)

300000 6-input LUTs; 28 nm feature size



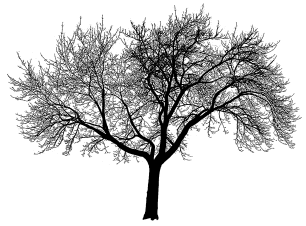
The Practical Question

How do we synthesize hardware from pure functional languages for FPGAs?

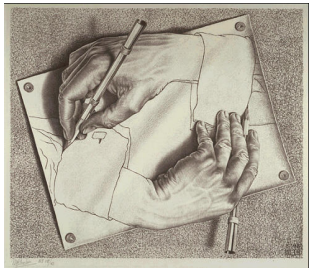
Control and datapath are easy; the memory system is interesting.

To Implement Real Algorithms, We Need

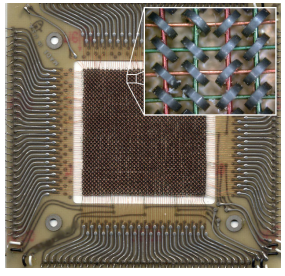
Structured, recursive data types



Recursion to handle recursive data types



Memories



Memory Hierarchy

The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

type ::= *Type*	Primitive type
| *Constr Type** | $\cdots$ | *Constr Type**	Tagged union

Subsume C structs, unions, and enums

Comparable power to C++ objects with virtual methods

"Algebraic" because they are sum-of-product types.

The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

```
type ::= Type                                          Primitive type
       | Constr Type* | ··· | Constr Type*             Tagged union
```

Examples:

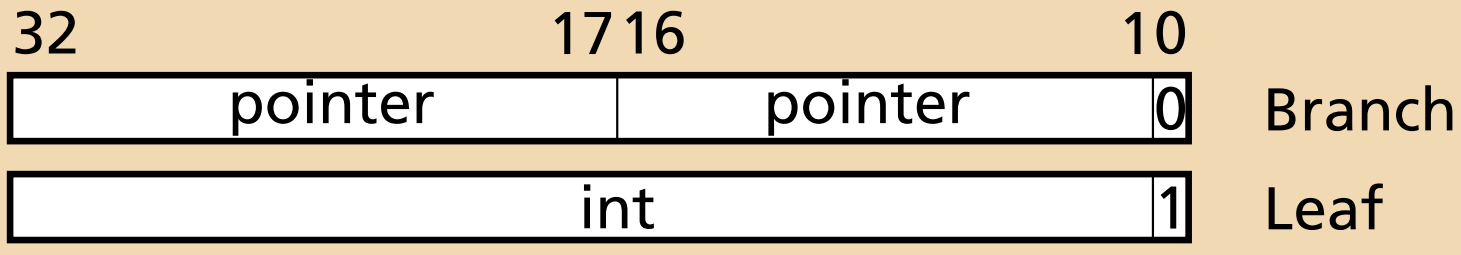
```
data Intlist = Nil                  -- Linked list of integers
             | Cons Int Intlist

data Bintree = Leaf Int             -- Binary tree of integers
             | Branch Bintree Bintree

data Expr = Literal Int             -- Arithmetic expression
          | Var String
          | Binop Expr Op Expr

data Op = Add | Sub | Mult | Div
```

Algebraic Datatypes in Hardware: Lists

```
data IntList  = Cons Int IntList
              | Nil
```

Datatypes in Hardware: Binary Trees

```
data IntTree = Branch IntTree IntTree
             |  Leaf Int
```



Example: Huffman Decoder in Haskell

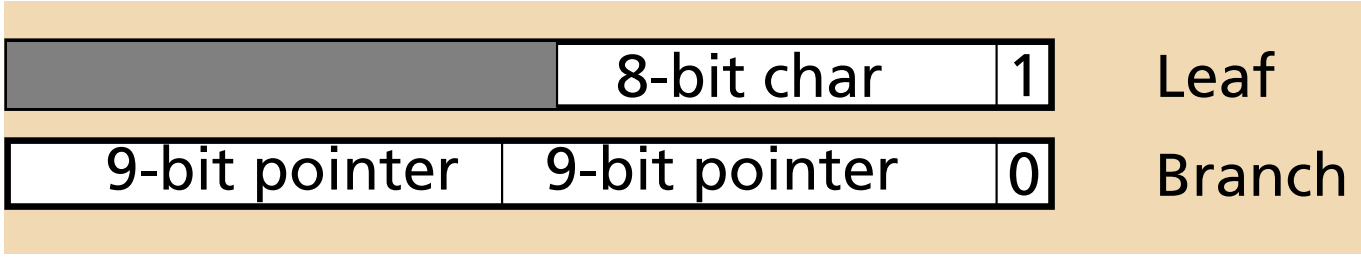
```
data HTree = Branch HTree HTree
           | Leaf Char

decode :: HTree → [Bool] → [Char]

decode table str = bit str table
  where
    bit (False:xs) (Branch l _) = bit xs l          -- 0: left
    bit (True:xs)  (Branch _ r) = bit xs r          -- 1: right
    bit x          (Leaf c)     = c : bit x table   -- leaf
    bit []         _            = []                -- done
```

Three data types:

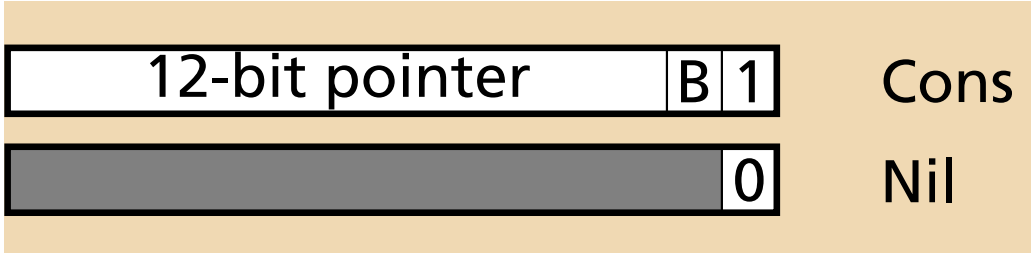
Input bitstream	[**Bool**] (list of Booleans)
Output character stream	[**Char**] (list of Characters)
Huffman tree	HTree

Encoding the Types

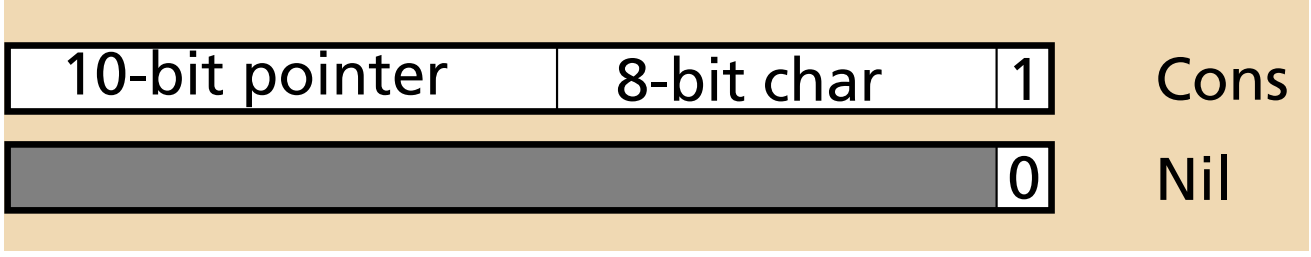
Huffman tree nodes: (19 bits)

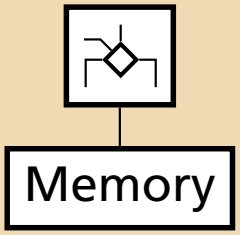


Boolean input stream: (14 bits)

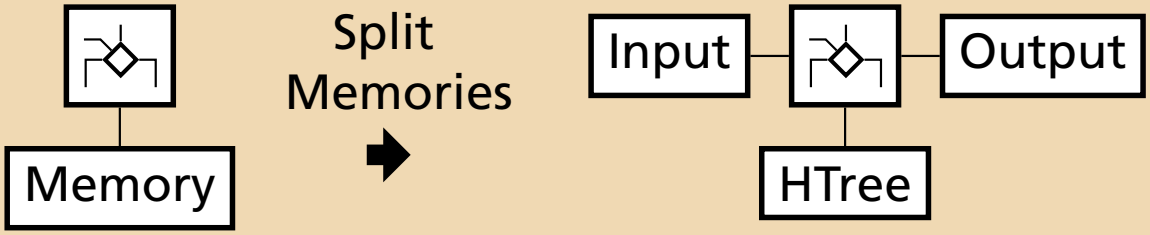


Character output stream: (19 bits)



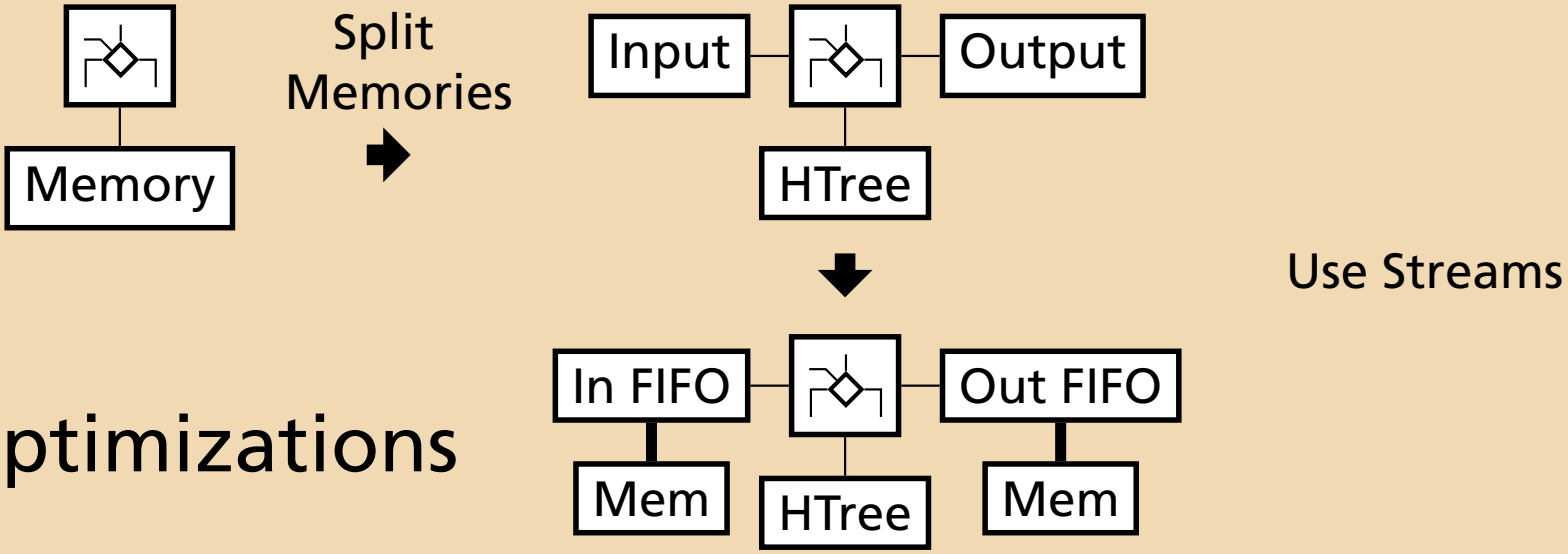


Optimizations

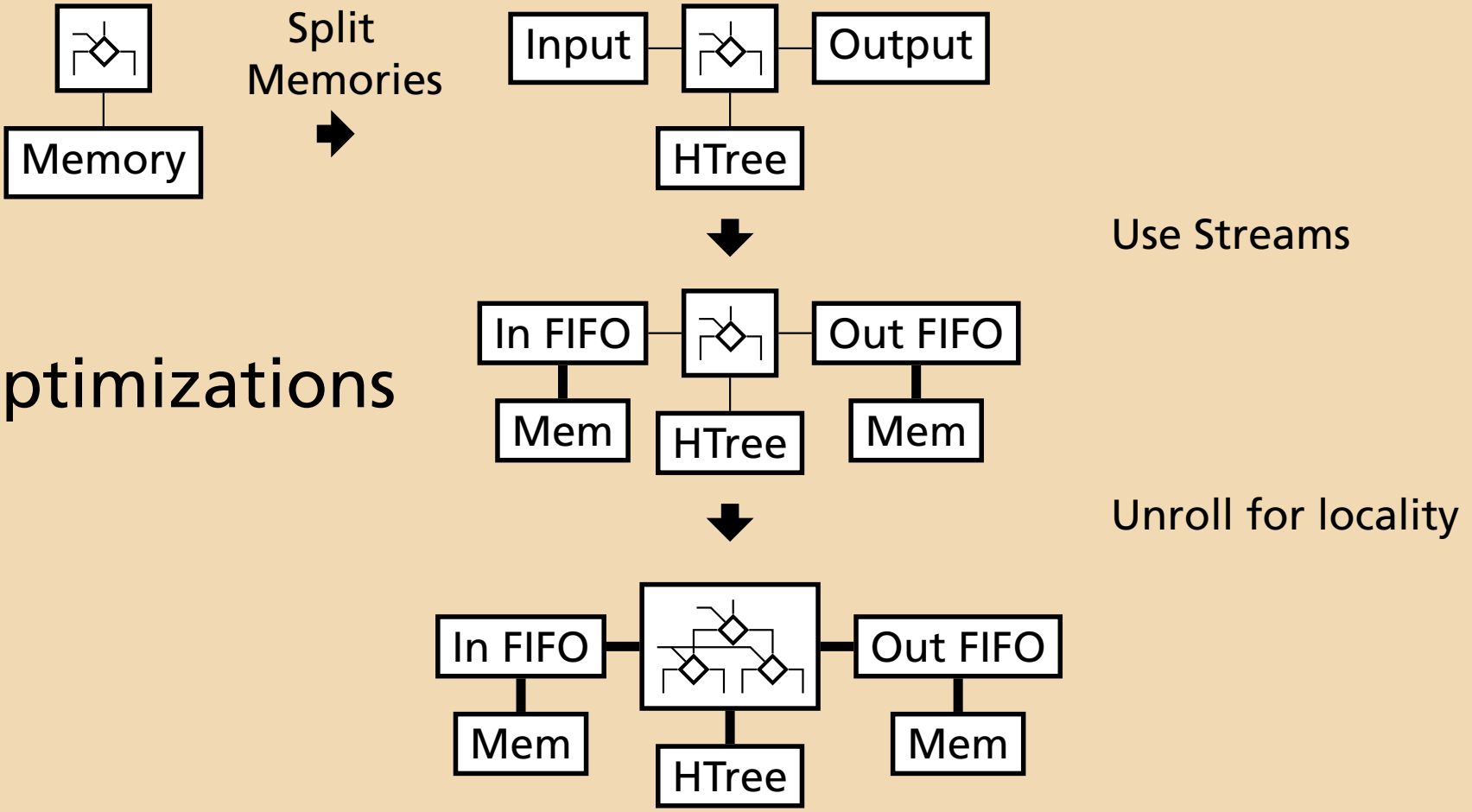


Optimizations

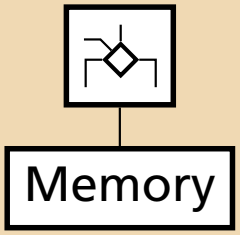
Optimizations



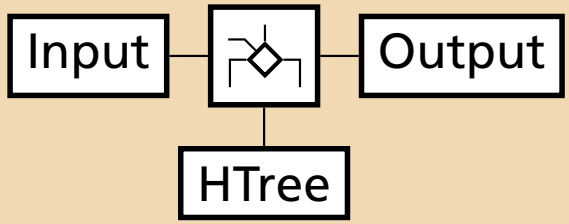
Optimizations



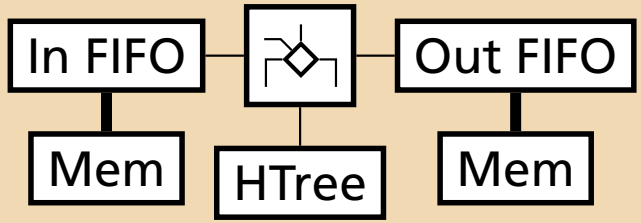
Optimizations



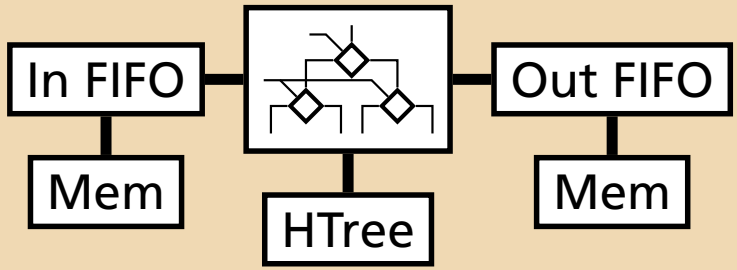
Split Memories



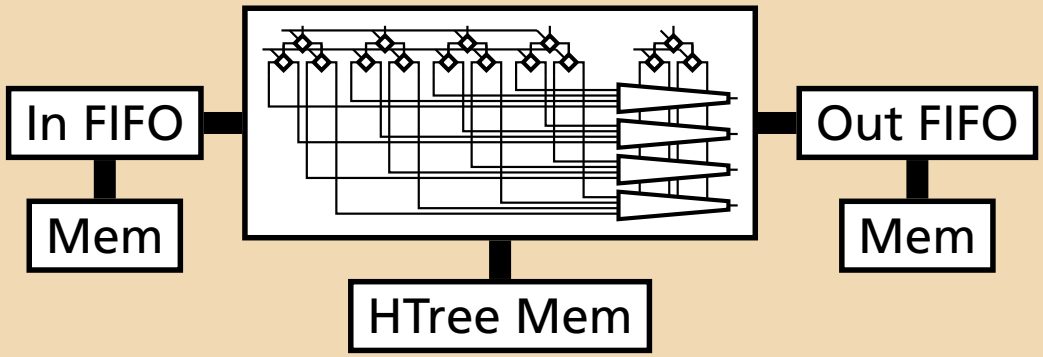
Use Streams



Unroll for locality



Speculate

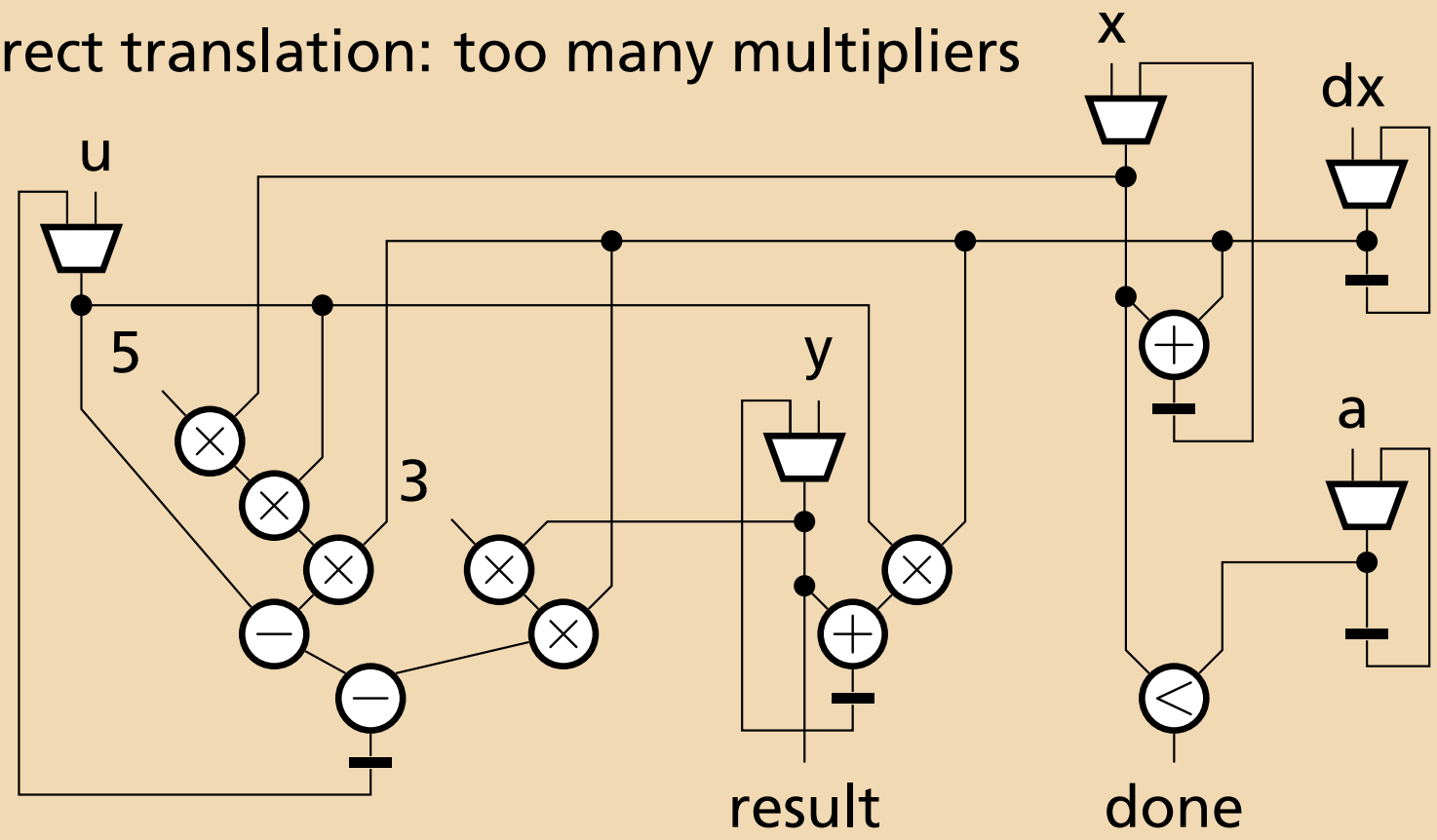


Hardware Synthesis: Semantics-preserving steps to a low-level dialect

High-Level Synthesis in a Functional Setting

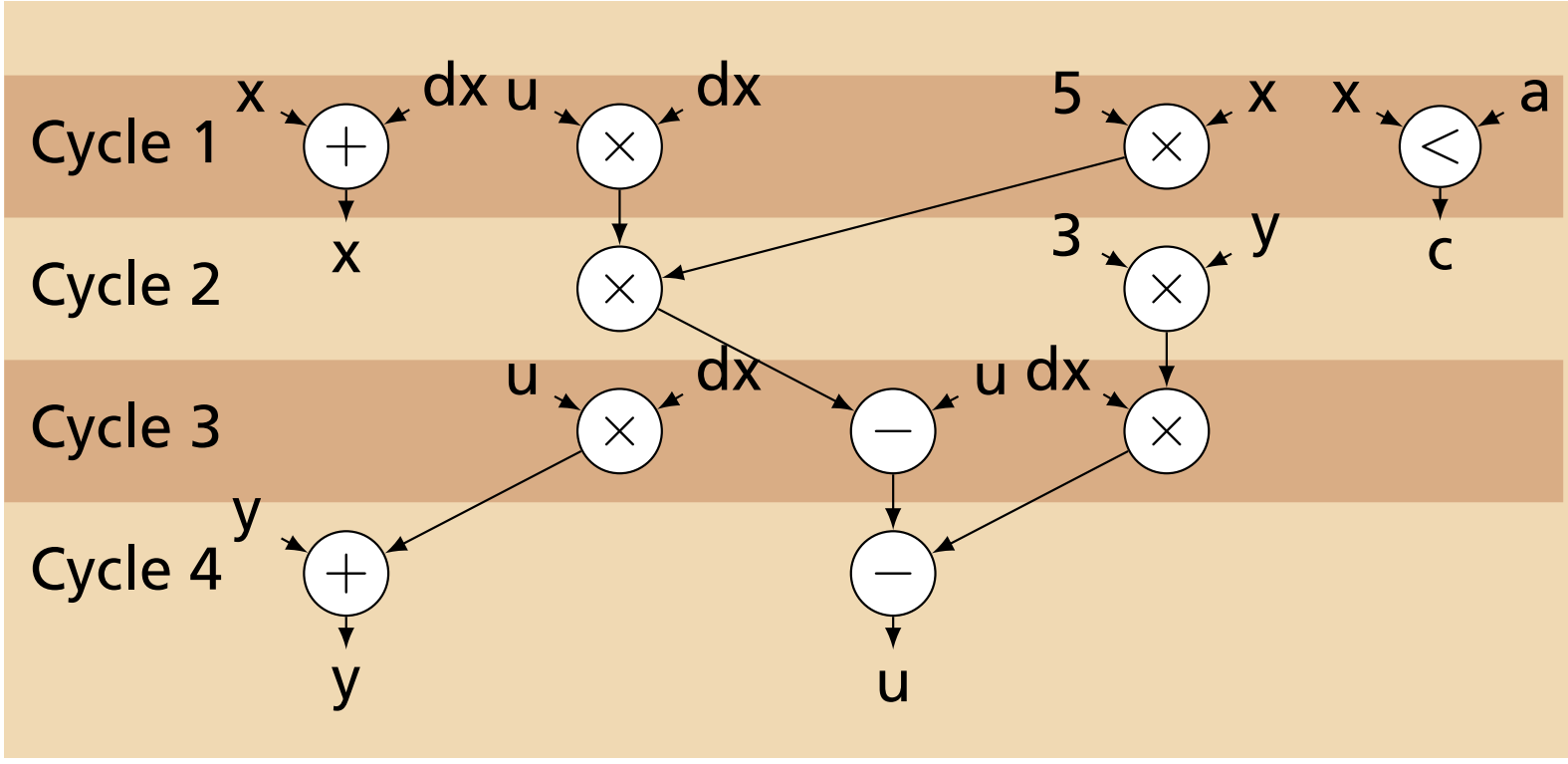
```
diffeq a dx x u y =
  if  x < a then
    diffeq a dx (x + dx) (u − 5∗x∗u∗dx − 3∗y∗dx) (y + u∗dx)
  else y
```

Direct translation: too many multipliers



Scheduling: Time-multiplex Two Multiplers

```
diffeq a dx x u y =
  if  x < a then
    diffeq a dx (x + dx) (u − 5∗x∗u∗dx − 3∗y∗dx) (y + u∗dx)
  else y
```



Scheduling: Mapping to a Datapath

```
diffeq a dx x u y =
  if x < a then
    diffeq a dx (x + dx) (u − 5∗x∗u∗dx − 3∗y∗dx) (y + u∗dx)
  else y
```

Introduce a function representing the datapath

```
dpath m1 m2 m3 m4 a1 a2 s1 s2 c1 c2 k =
   k (m1 ∗ m2) (m3 ∗ m4) (a1 + a2) (s1 − s2) (c1 < c2)
```

and re-express the function with the datapath

```
diffeq a dx x u y =
  dpath u dx  5  x  x dx 0  0  x a (λpa pb x _ c → if not c then y else
  dpath pa pb 3  y  0 0  0  0  0 0 (λpa pb _ _ _ →
  dpath u dx  dx pb 0 0  u  pa 0 0 (λpa pb _ d _ →
  dpath 0 0   0  0  y pa d  pb 0 0 (λ_ _  s  d _ → diffeq a dx x d s ))))
```

Name Lambda Expressions (Continuations)

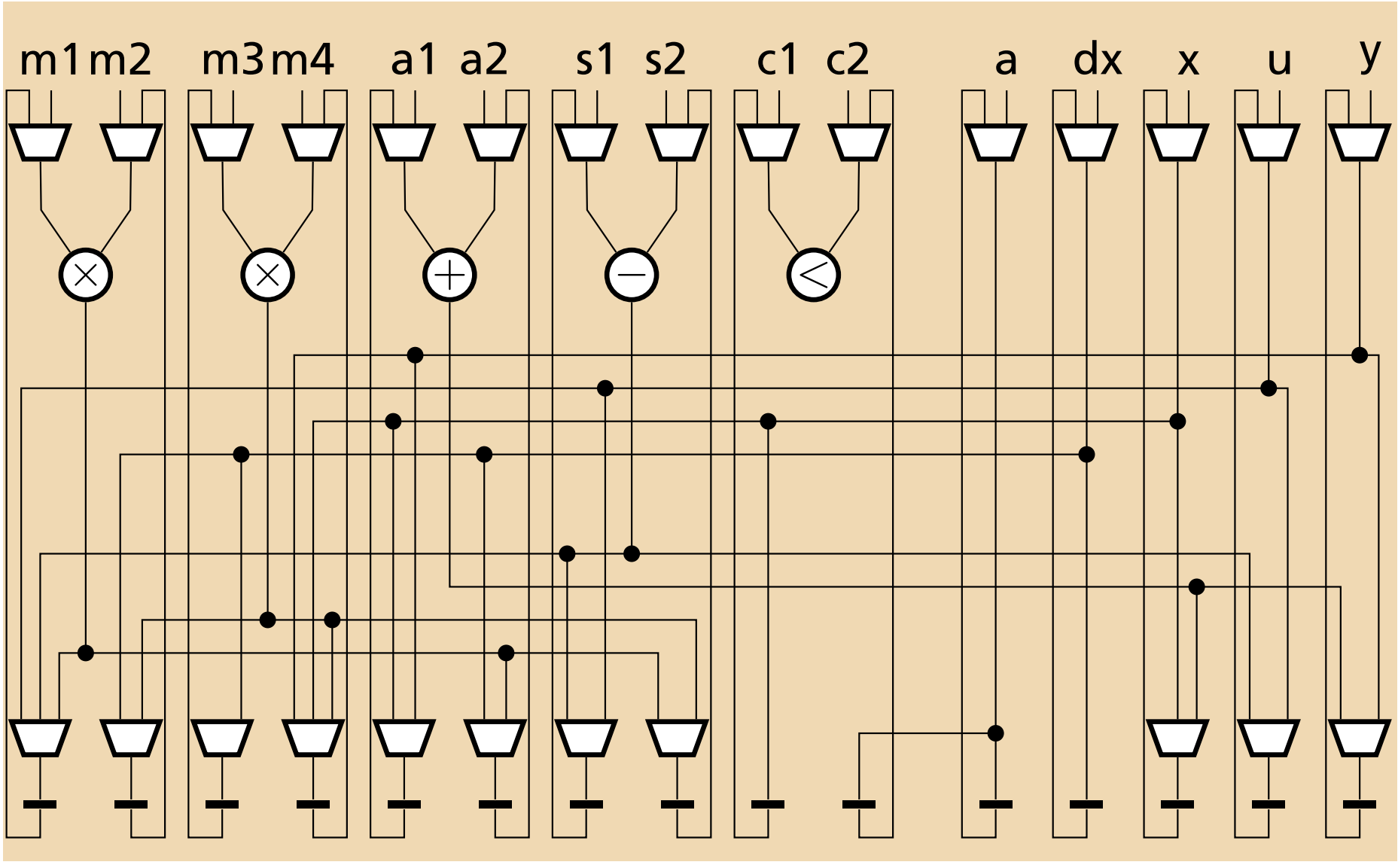
```
diffeq a dx x u y =
  if x < a then
    diffeq a dx (x + dx) (u − 5∗x∗u∗dx − 3∗y∗dx) (y + u∗dx)
  else y
```

```
k0 a dx x       _  _   s d _ =
   dpath d dx  5  x    x dx 0  0  x a (k1 a dx    d s)
k1 a dx   u y  pa pb s _ c =
   if not c then y else
   dpath pa pb 3  y    0 0  0  0  0 0 (k2 a dx s  u y)
k2 a dx x u y  pa pb _ _ _ =
   dpath u dx  dx pb  0 0  u  pa 0 0 (k3 a dx x    y)
k3 a dx x   y  pa pb _ d _ =
   dpath 0 0    0  0   y pa d  pb 0 0 (k0 a dx x     )

diffeq a dx x u y = k0 a dx x 0 0 y u False
```

Encode Continuations as a Type

```
data Cont = K0 Int Int Int
          | K1 Int Int       Int  Int
          | K2 Int Int  Int  Int  Int
          | K3 Int Int  Int       Int

dpath m1 m2 m3 m4 a1 a2 s1 s2 c1 c2 k =
   kk k (m1 * m2) (m3 * m4) (a1 + a2) (s1 - s2) (c1 < c2)

kk k m1 m2 a s c = case (k, m1, m2, a, s,  c)  of
 (K0 a dx x      ,_ ,_ ,s,d,_) →
      dpath d dx 5  x   x dx 0 0   x a (K1 a dx    d s)
 (K1 a dx    u y,pa,pb,s,_,c) → if  not c then y else
      dpath pa pb 3 y   0 0   0 0   0 0 (K2 a dx s u y)
 (K2 a dx x u y,pa,pb,_,_,_) →
      dpath u dx dx pb 0 0  u pa 0 0 (K3 a dx x    y)
 (K3 a dx x    y,pa,pb,_,d,_) →
      dpath 0 0   0   0   y pa d pb 0 0 (K0 a dx x      )

diffeq a dx x u y = kk (K0 a dx x) 0 0 y u False
```

Syntax-Directed Translation to Hardware



Removing Recursion: The Fib Example

```
fib  n          = case n of
                    1      → 1
                    2      → 1
                    n      → fib  (n−1) + fib (n−2)
```

Transform to Continuation-Passing Style

```
fibk  n k       = case n of
                    1     → k 1
                    2     → k 1
                    n     → fibk (n−1) (λn1 →

                                fibk (n−2) (λn2 →
                                k (n1 + n2)))

fib     n       =           fibk n (λx → x)
```

Name Lambda Expressions (Lambda Lifting)

```
fibk  n  k      = case n of
                    1      → k 1
                    2      → k 1
                    n      → fibk (n−1) (k1 n k)

k1    n  k n1 =              fibk (n−2) (k2 n1 k)
k2    n1 k n2 =              k (n1 + n2)
k0          x =              x
fib   n       =              fibk n k0
```

Represent Continuations with a Type

```
data Cont = K0 | K1 Int Cont | K2 Int Cont


fibk n k        = case (n,k) of
                    (1, k) → kk k 1
                    (2, k) → kk k 1
                    (n, k) → fibk (n−1) (K1 n k)
kk k a          = case (k, a) of
        ((K1 n k), n1) → fibk (n−2) (K2 n1 k)
        ((K2 n1 k), n2) → kk k (n1 + n2)
        (K0,        x ) → x
fib n           =          fibk n K0
```

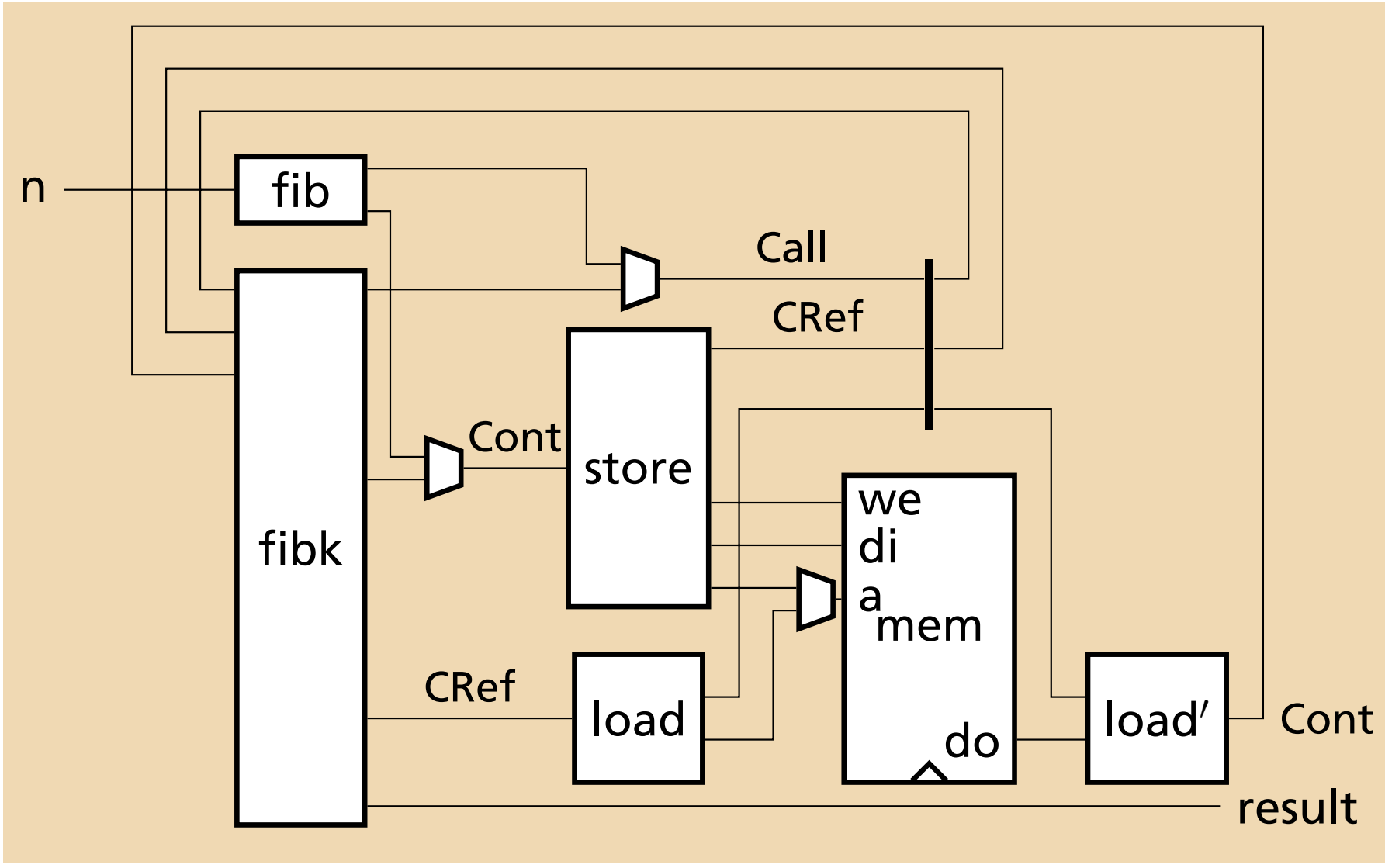
Merge Functions

```
data Cont = K0 | K1 Int Cont | K2 Int Cont
data Call = Fibk Int  Cont | KK Cont Int

fibk  z          = case z of
     (Fibk       1   k) → fibk  (KK k 1)
     (Fibk       2   k) → fibk  (KK k 1)
     (Fibk       n   k) → fibk  (Fibk (n−1) (K1 n  k))

     (KK (K1 n  k)  n1) → fibk  (Fibk (n−2) (K2 n1 k))
     (KK (K2 n1 k) n2) → fibk  (KK k (n1 + n2))
     (KK K0         x ) → x
fib  n           =          fibk  (Fibk n K0)
```

Add Explicit Memory Operations

```
load  :: CRef → Cont
store :: Cont → CRef
data Cont = K0 | K1 Int CRef | K2 Int CRef
data Call = Fibk Int CRef | KK Cont Int

fibk z           = case z of
    (Fibk    1  k) → fibk (KK (load k) 1)
    (Fibk    2  k) → fibk (KK (load k) 1)
    (Fibk    n  k) → fibk (Fibk (n−1) (store (K1 n  k)))

    (KK (K1 n  k) n1) → fibk (Fibk (n−2) (store (K2 n1 k)))
    (KK (K2 n1 k) n2) → fibk (KK (load k) (n1 + n2))
    (KK K0        x ) → x
fib n            =      fibk (Fibk n (store K0))
```

Syntax-Directed Translation to Hardware



Duplication Can Increase Parallelism

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

Duplication Can Increase Parallelism

```
fib  0 = 0
fib  1 = 1
fib  n = fib  (n−1) + fib (n−2)
```

After duplicating functions:

```
fib  0 = 0
fib  1 = 1
fib  n = fib′  (n−1) + fib′′  (n−2)

fib′  0 = 0
fib′  1 = 1
fib′  n = fib′  (n−1) + fib′  (n−2)

fib′′  0 = 0
fib′′  1 = 1
fib′′  n = fib′′  (n−1) + fib′′  (n−2)
```

Here, *fib′* and *fib″* may run in parallel.

Unrolling Recursive Data Structures

Original Huffman tree type:

```
data Htree = Branch Htree HTree | Leaf Char
```

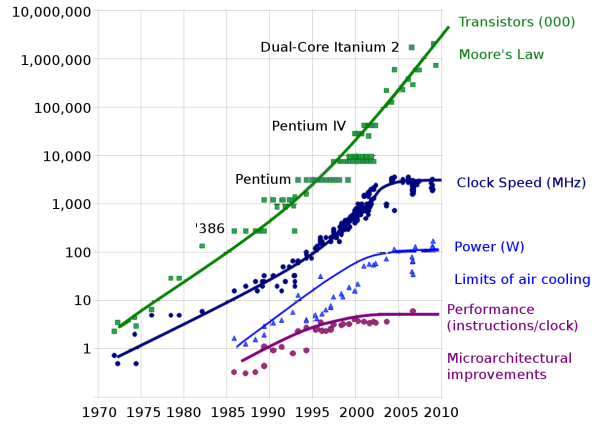
Unrolled Huffman tree type:

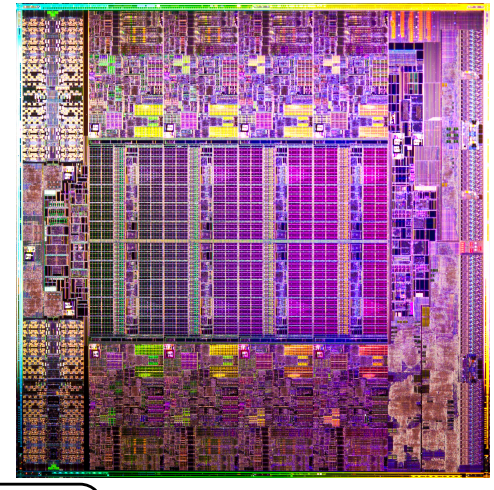
```
data Htree = Branch Htree′ HTree′ | Leaf Char
data Htree′ = Branch′ Htree′′ HTree′′ | Leaf′ Char
data Htree′′ = Branch′′ Htree HTree | Leaf′′ Char
```

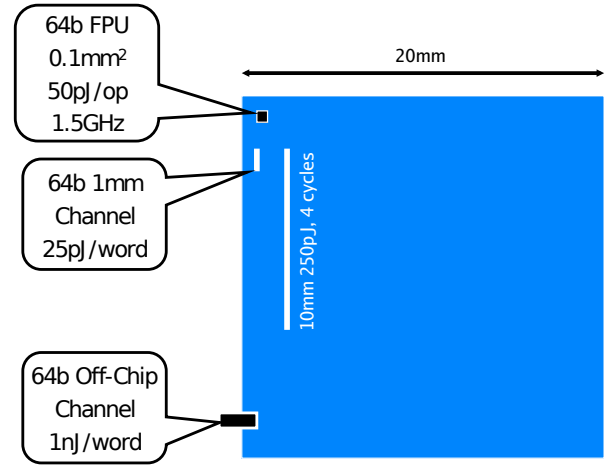
Increases locality: larger data blocks.

A type-aware cache line

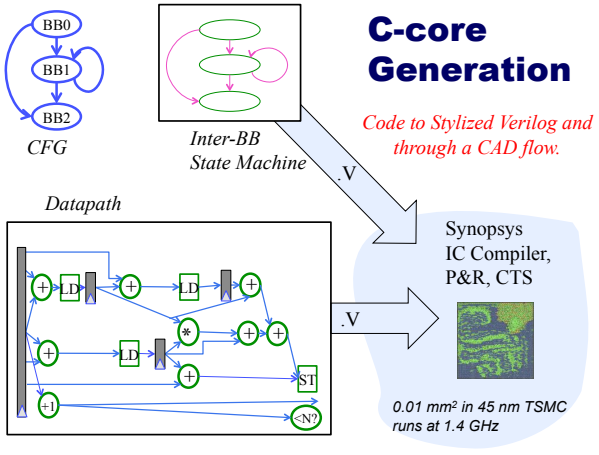
- Moore's Law is alive and well
- But we hit a power wall in 2005. Massive parallelism now mandatory
- Communication is the culprit

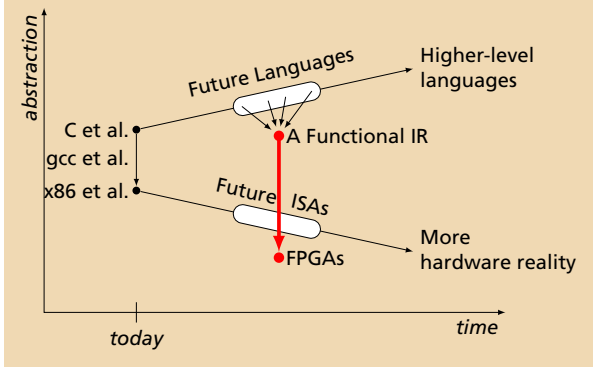






- Dark Silicon is the future: faster transistors; most must remain off
- Custom accelerators are the future; many approaches
- My project: A Pure Functional Language to FPGAs

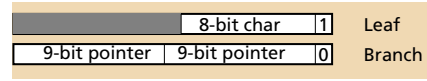




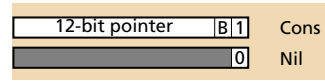
- Algebraic Data Types in Hardware
- Optimizations
- Removing recursion

Encoding the Types

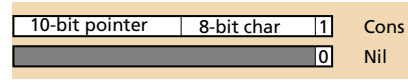
Huffman tree nodes: (19 bits)

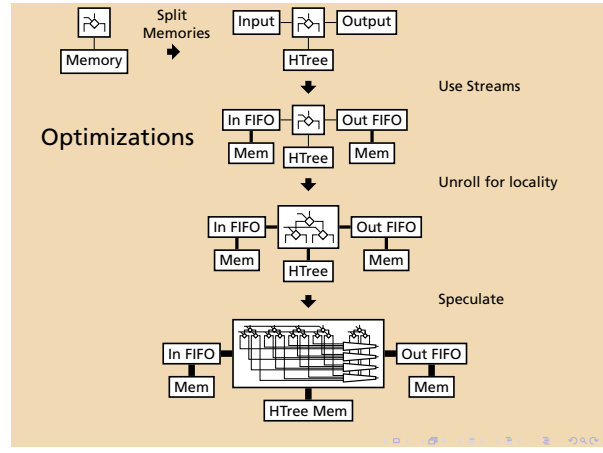


Boolean input stream: (14 bits)



Character output stream: (19 bits)





Syntax-Directed Translation to Hardware