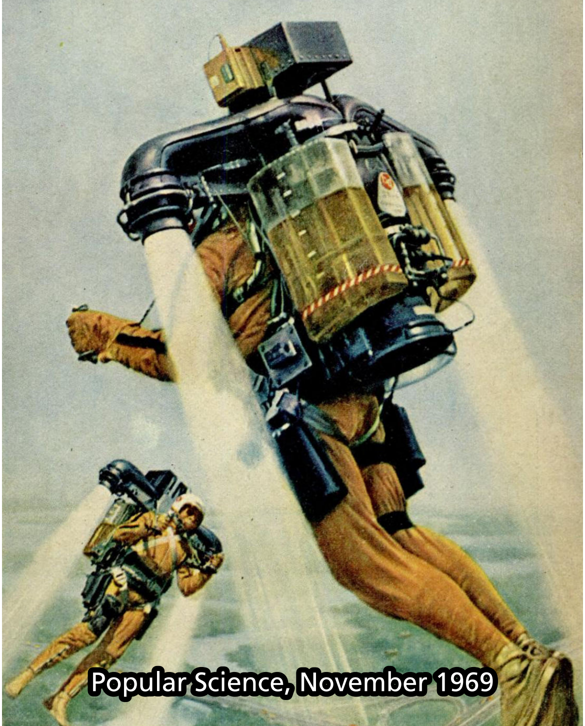
Haskell to Hardware and Other Dreams

Stephen A. Edwards

Richard Townsend   Martha A. Kim
Lianne Lairmore   Kuangya Zhai

Columbia University

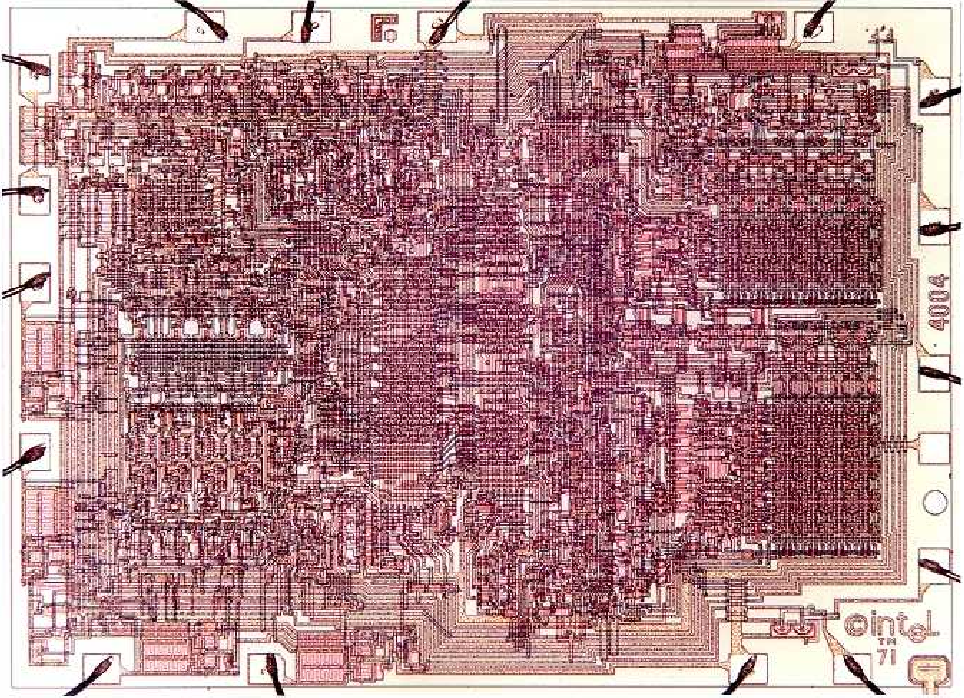
Synchron, Bamberg, Germany, December 7, 2016

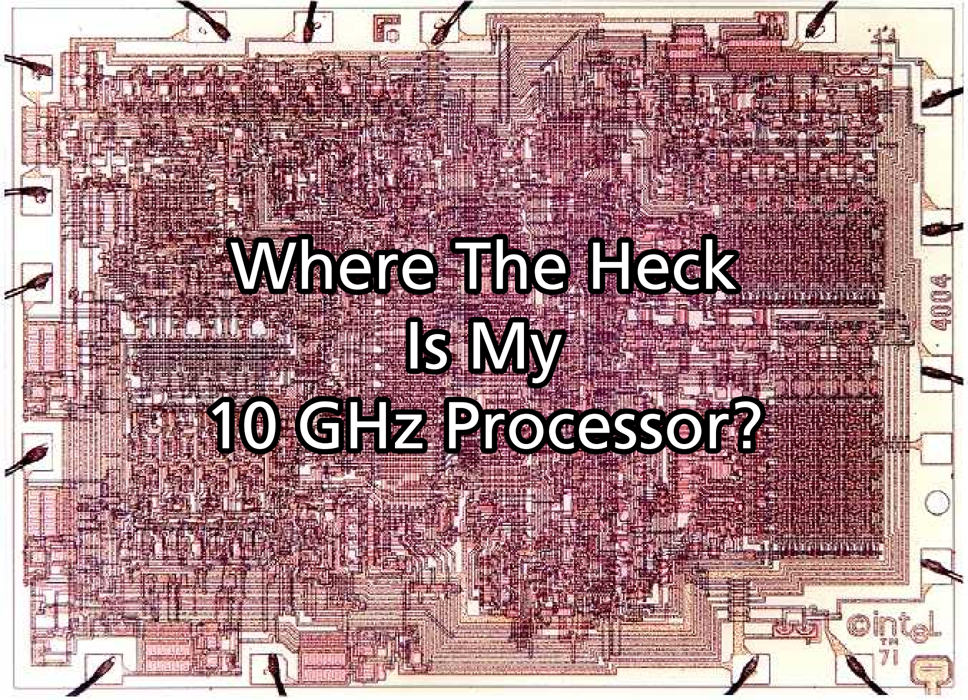


Popular Science, November 1969

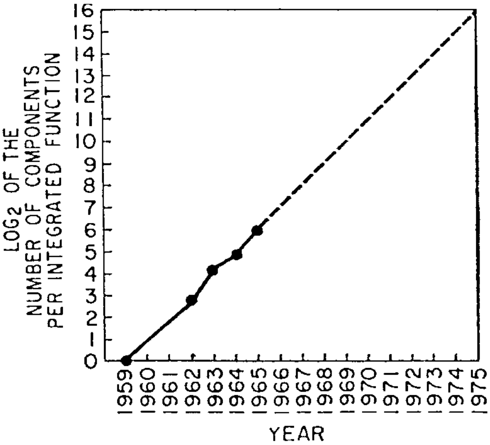
Where Is My Jetpack?

Popular Science, November 1969


4004
©intel
TM
71


Where The Heck
Is My
10 GHz Processor?
4004
intel
71

Moore's Law

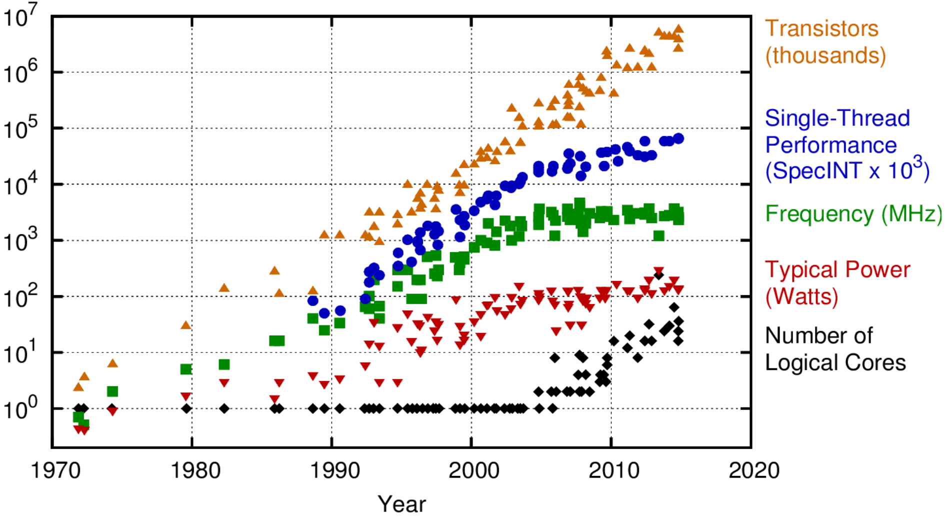


"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."

Closer to every 24 months

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

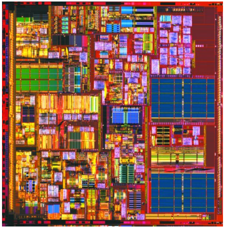
Four Decades of Microprocessors Later...



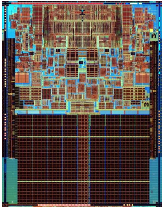
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Source: `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`

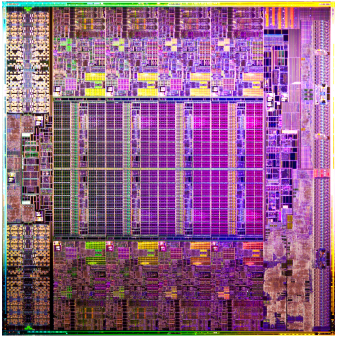
What Happened in 2005?



Pentium 4
2000
1 core
Transistors: 42 M

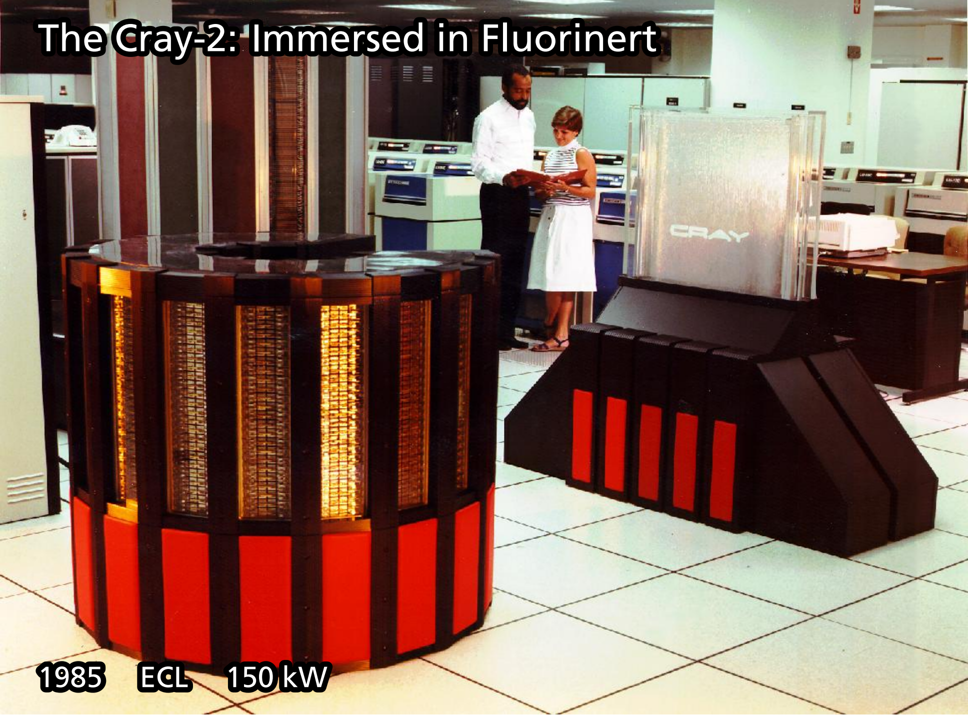


Core 2 Duo
2006
2 cores
291 M

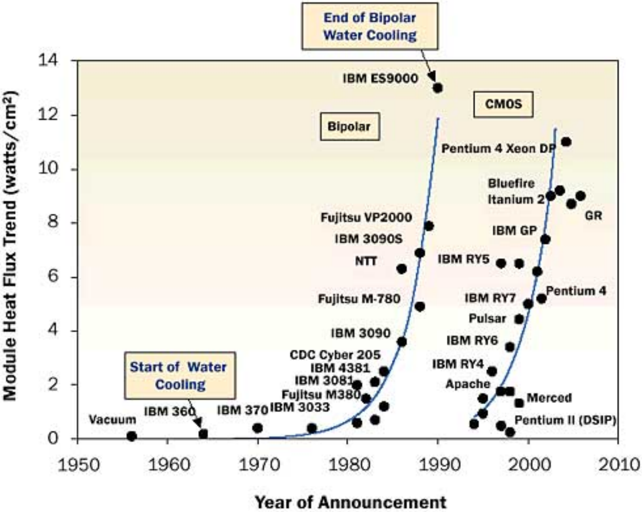


Xeon E5
2012
8 cores
2.3 G

The Cray-2: Immersed in Fluorinert



Heat Flux in IBM Mainframes: A Familiar Trend

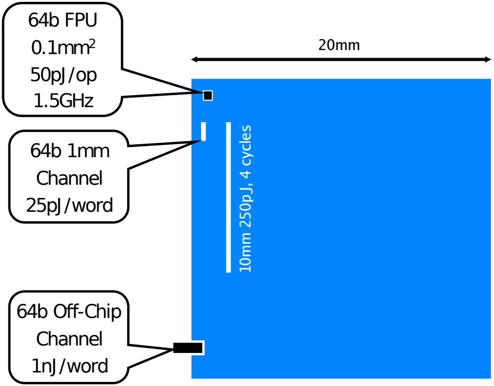


Schmidt. *Liquid Cooling is Back*. Electronics Cooling. August 2005.

Liquid Cooled Apple Power Mac G5

2004 CMOS 1.2 kW

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”

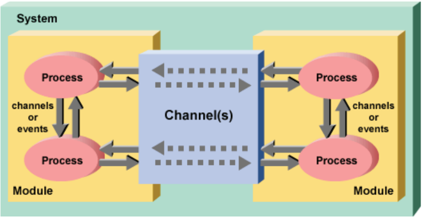
We need
different programming paradigms
and
different architectures
on which to run them.

Dark Silicon

Related Work

Xilinx's Vivado (Was xPilot, AutoESL)

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

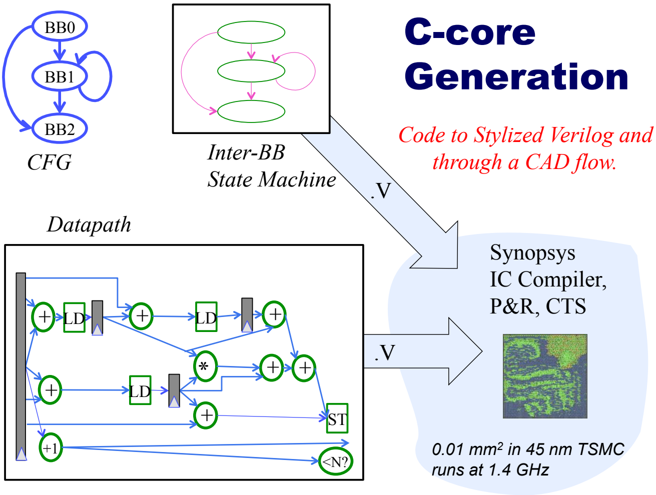


  - Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
    - Port / IO interfaces, bit-vector manipulations, cycle-level notations

SystemC input; classical high-level synthesis for processes

Jason Cong et al. ISARS 2005

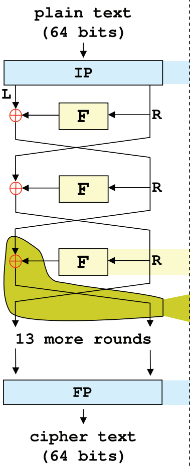
Taylor and Swanson's Conservation Cores



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

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

Bacon et al.'s Liquid Metal




```
public static Unsigned64 DEScoder(KeySchedule keys,
                                  Unsigned64 text)
{
    Unsigned64 block = text.permute(IP.Permutation);

    Unsigned32 R = block.extractBits(sixtyfour.b0 ,
                                     sixtyfour.b31);
    Unsigned32 L = block.extractBits(sixtyfour.b32,
                                     sixtyfour.b63);


    for ( sixteen round ) {

        Unsigned32 F = Fiestel(keys, round, R);

        Unsigned32 X = F ^ L;
        L = R;
        R = X;
    }
    Unsigned64 LR = makeUnsigned64(R, L);
    return LR.permute(FP.Permutation);
}
```

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;
}
```

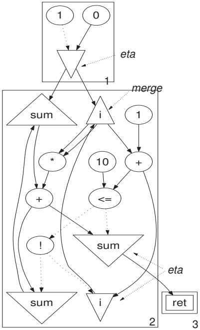


Figure 3: *C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)*

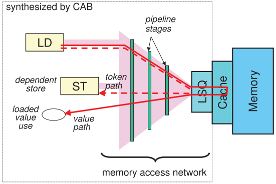


Figure 8: *Memory access network and implementation of the value and token forwarding network. The* LOAD *produces a data value consumed by the oval node. The* STORE *node may depend on the load (i.e., we have a token edge between the* LOAD *and the* STORE*, shown as a dashed line). The token travels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

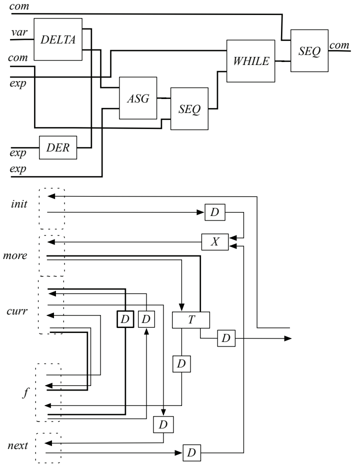


Figure 1. In-place map schematic and implementation

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

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

GCD Flip Rule

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

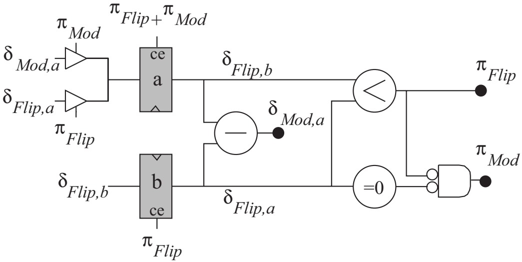


Figure 1.3 Circuit for computing $\text{Gcd}(a, b)$ from Example 1.

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
```

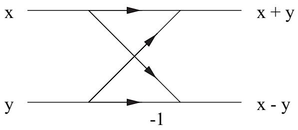


Figure 9: A butterfly

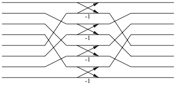


Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures

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

Kuper et al.'s CλaSH

$$\begin{array}{l} \mathit{fir}\ (\mathit{State}\ (\mathit{xs}, \mathit{hs}))\ x = \\ \quad (\mathit{State}\ (\mathit{shiftInto}\ x\ \mathit{xs}, \mathit{hs}), (x \triangleright \mathit{xs}) \bullet \mathit{hs}) \end{array}$$

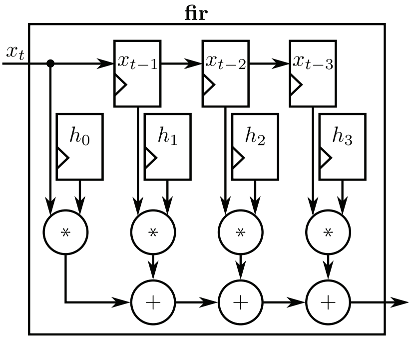


Fig. 6. 4-taps FIR Filter

More operational Haskell specifications of regular structures

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

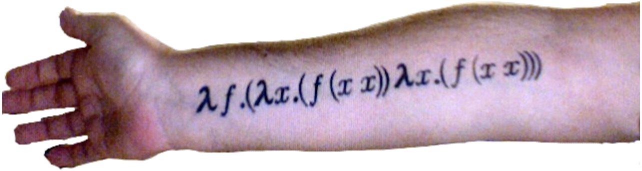
My Crusade

Deterministic Concurrency: A Fool's Errand?

What Models of Computation Provide Determinstic Concurrency?

Synchrony	The Columbia Esterel Compiler 2001–2006
Kahn Networks	SHIM The SHIM Model/Language 2006–2010
The Lambda Calculus	This Project 2010–

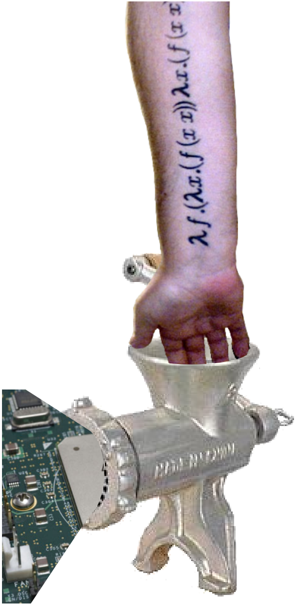
Our Project: Functional Programs to Hardware



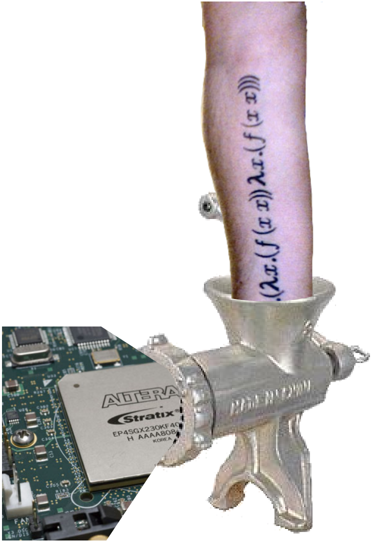
Our Project: Functional Programs to Hardware

Our Project: Functional Programs to Hardware

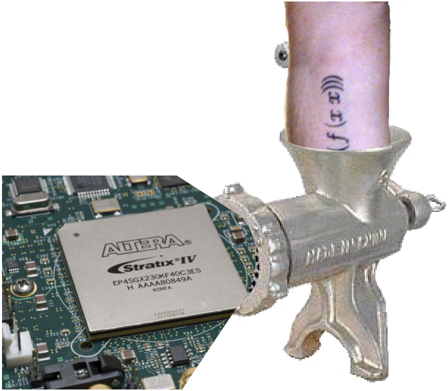
Our Project: Functional Programs to Hardware



Our Project: Functional Programs to Hardware



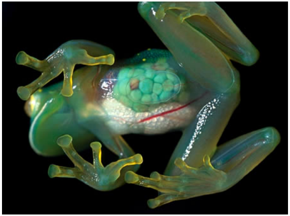
Our Project: Functional Programs to Hardware



Our Project: Functional Programs to Hardware

Why Functional?

- Referential transparency simplifies formal reasoning about programs



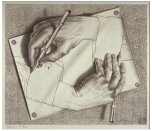
- Inherently concurrent and deterministic
  (Thank Church and Rosser)

- Immutable data makes it vastly easier to reason about memory in the presence of concurrency

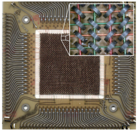
To Implement Real Algorithms, We Need

Structured, recursive data types

Recursion to handle recursive data types



Memories



Memory Hierarchy

Structured, Recursive Data Types

Algebraic Data Types

In modern functional languages: ML, OCaml, Haskell, ...

An algebraic type is a sum of product types

Basic example: List of integers

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

Constructing a list:

```
Cons 42 (Cons 17 (Cons 2 (Cons 1 Nil)))
```

Summing the elements of a list:

```
sum li = case li of
  Nil         → 0
  Cons x xs → x + sum xs
```

An Interpreter in One Slide

Abstract syntax tree data type:

```
data Expr = Lit   Int
          | Plus  Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
```

Recursive evaluation function:

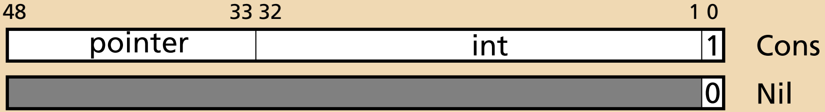
```
eval e = case e of
   Lit x         → x
   Plus e1 e2    → eval e1 + eval e2
   Minus e1 e2   → eval e1 − eval e2
   Times e1 e2   → eval e1 * eval e2
```

```
eval (Plus (Lit 42) (Times (Lit 2) (Lit 50)))
```

gives $42 + 2 \times 50 = 142$

Algebraic Datatypes in Hardware: Lists

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



Recursion to Handle Recursive Data Types

What Do We Do With Recursion?

Compile it into tail recursion with explicit stacks

[Zhai et al., CODES+ISSS 2015]

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

[Proceedings of the ACM Annual Conference, 1972]

Really clever idea:

Sophisticated language ideas such as recursion and higher-order functions can be implemented using simpler mechanisms (e.g., tail recursion) by rewriting.

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

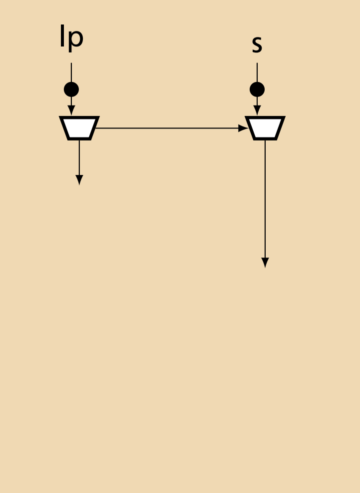
```
read   ::  CRef → Cont
write  ::  Cont → CRef
data Cont = K0 | K1 Int CRef | K2 Int CRef
data Call = Fibk Int CRef | KK Cont Int

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

    (KK (K1 n  k)  n1) → fibk (Fibk (n−2) (write (K2 n1 k)))
    (KK (K2 n1 k) n2) → fibk (KK (read k) (n1 + n2))
    (KK K0        x )  → x
fib n            =       fibk (Fibk n (write K0))1
```

Simplified Functional to Dataflow

Functional to Dataflow

Sum a list using an accumulator and tail-recursion

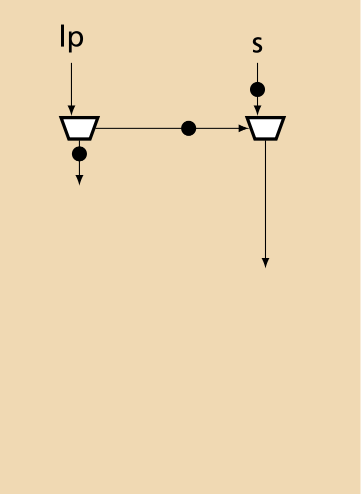
```
sum lp s =
 case read lp of
   Nil        →  s
   Cons x xs  →  sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

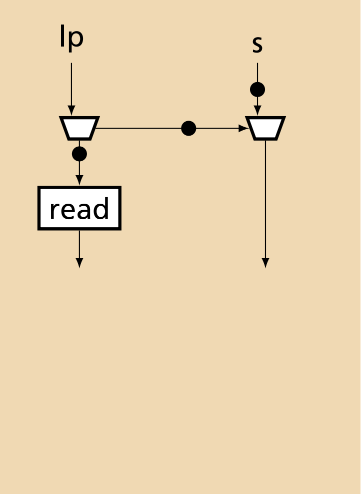
```
sum lp s =
 case read lp of
   Nil       → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

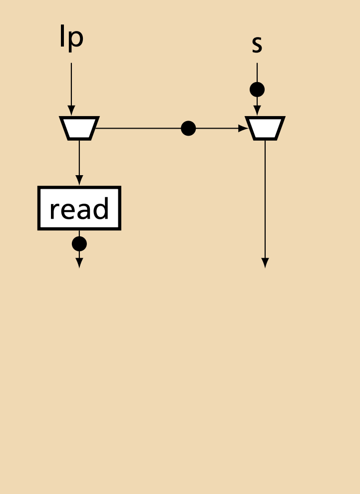
sum lp s =
 case read lp **of**
  Nil → s
  Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

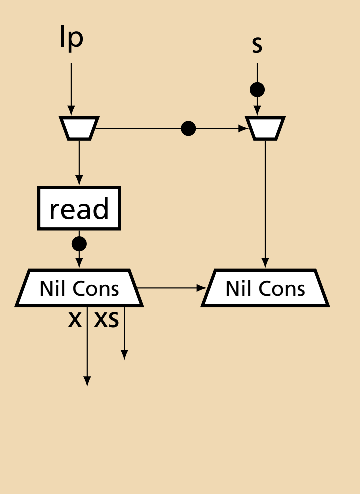
sum lp s =
 case read lp **of**
  Nil → s
  Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

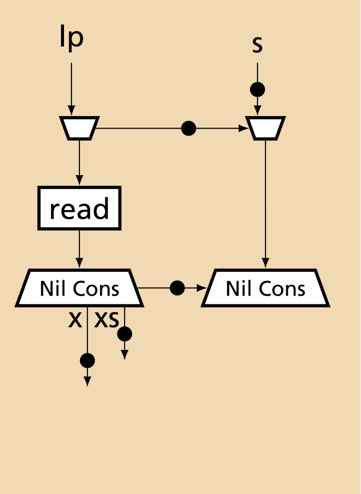
```
sum lp s =
 case read lp of
    Nil        → s
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

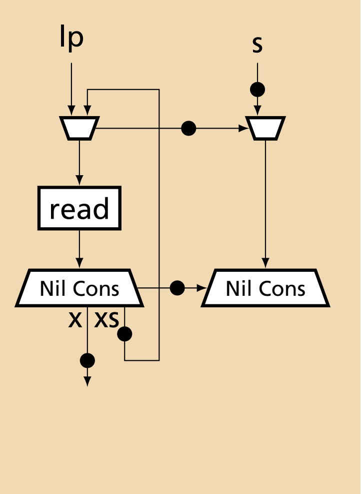
```
sum lp s =
 case read lp of
   Nil       → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

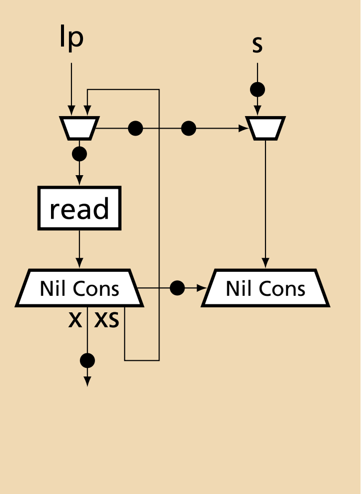
sum lp s =
 case read lp **of**
  Nil → s
  Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

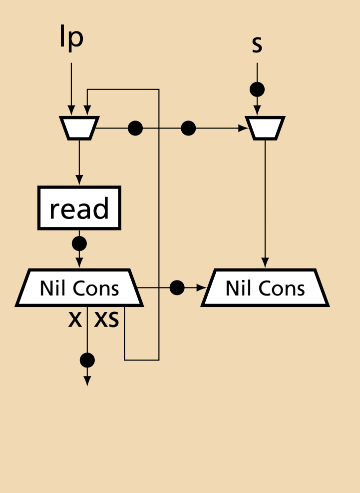
```
sum lp s =
 case read lp of
   Nil       → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

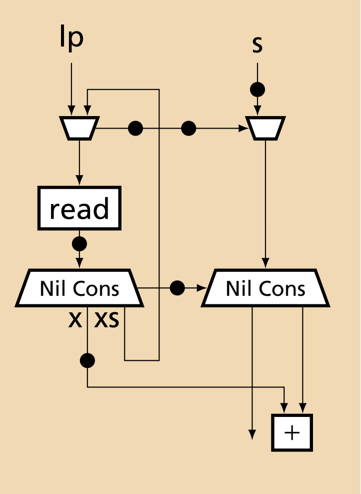
sum lp s =
 case read lp **of**
  Nil → s
  Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

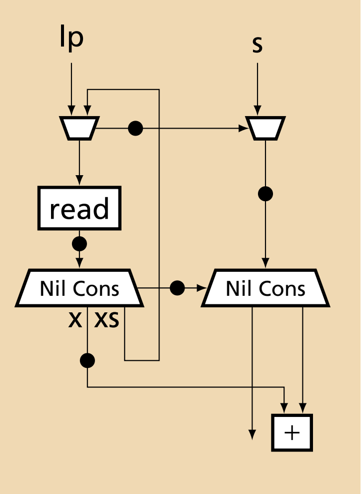
```
sum lp s =
  case read lp of
    Nil        → s
    Cons x xs  → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

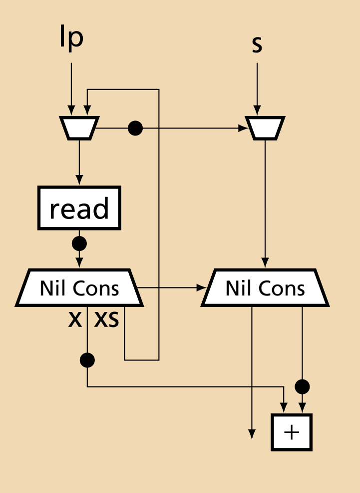
```
sum lp s =
  case read lp of
    Nil      → s
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

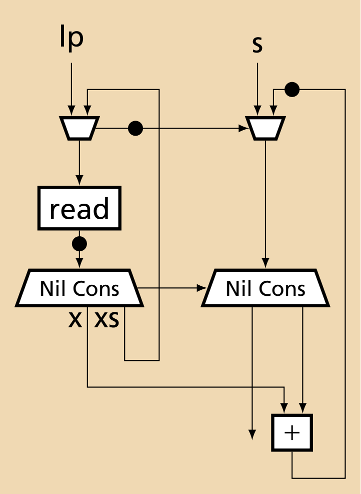
```
sum lp s =
  case read lp of
    Nil       → s
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

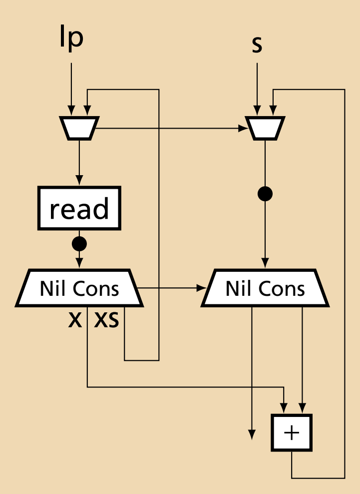
sum lp s =
  case read lp **of**
    Nil → s
    Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

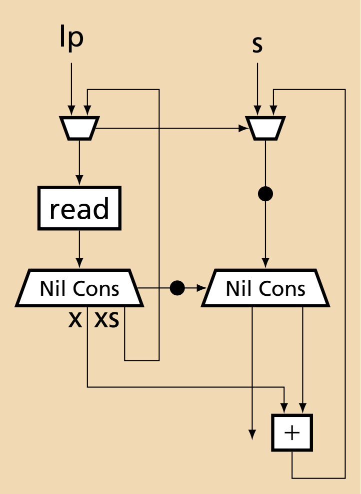
sum lp s =
  case read lp **of**
    Nil → s
    Cons x xs → **sum** xs (s + x)



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

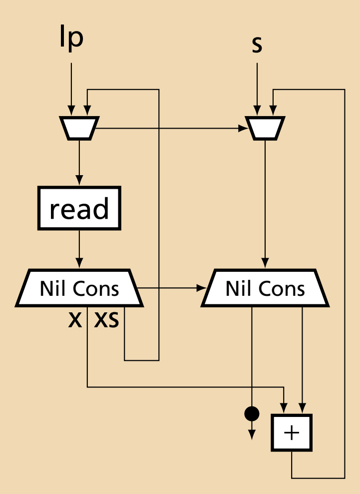
```
sum lp s =
  case read lp of
    Nil        → s
    Cons x xs → sum xs (s + x)
```



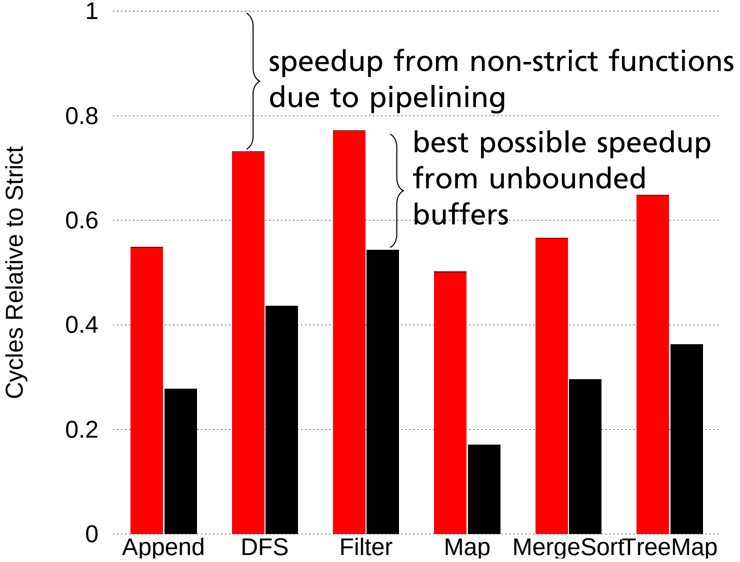
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

```
sum lp s =
  case read lp of
    Nil       → s
    Cons x xs → sum xs (s + x)
```

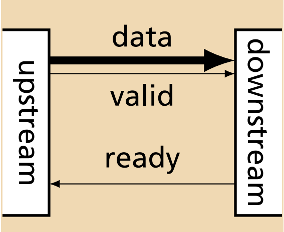


Non-strict functions enables pipelining

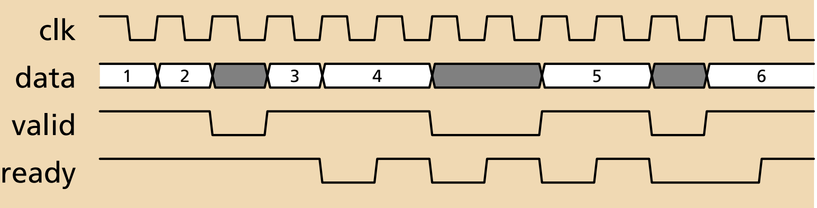


Dataflow to Hardware

A Latency-Insensitive Protocol

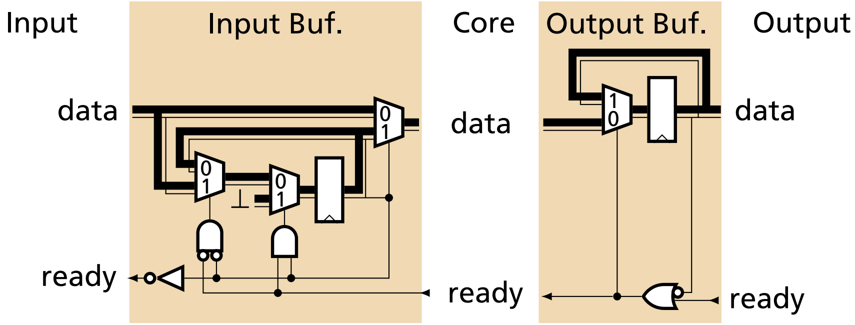


valid	ready	action
0	–	No token
1	1	Token Transfer
1	0	Token held upstream



Inspired by Carloni et al.
[Cao et al., Memocode 2015]

Input and Output Buffers



Combinational paths broken:

Input buffer breaks *ready* path

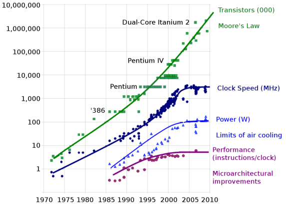
Output buffer breaks *data/valid* path

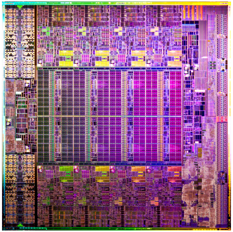
Larger Systems Run Just As Fast

Splitters	Token Bits	F_{max} MHz	Area Resources ALMs	%	Registers
2	32	167	189	1	414
2	64	157	350	1	798
2	128	152	672	2	1573
32	128	137	10821	26	25536
64	128	140	21704	52	51168
4	64	158	700	2	1621
8	64	145	1409	3	3261
16	64	147	2826	7	6559
32	64	144	5682	14	13148
64	64	138	11404	27	26414
128	64	140	22914	55	53087

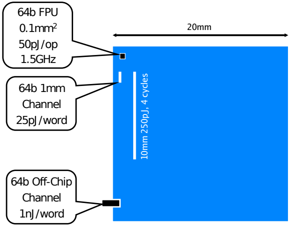
Synthesis results on an Altera Cyclone V. 166 MHz target clock rate.

- Moore’s Law is alive and well





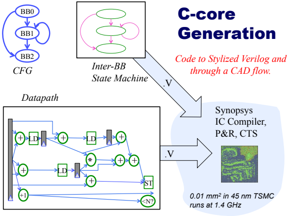
- But we hit a power wall in 2005. Massive parallelism now mandatory



- Communication is the culprit

- Dark Silicon is the future: faster transistors; most must remain off

- Custom accelerators are the future; many approaches



- Our project: A Pure Functional Language to FPGAs

- Algebraic Data Types in Hardware

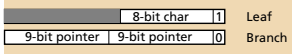
- Removing recursion

- Functional to dataflow

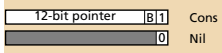
- Dataflow to hardware

Encoding the Types

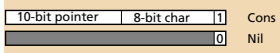
Huffman tree nodes: (19 bits)



Boolean input stream: (14 bits)



Character output stream: (19 bits)



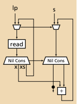
Add Explicit Memory Operations

```
read  ::  CRef → Cont
write ::  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 (read k) 1)
    (Fibk      2  k) → fibk (KK (read k) 1)
    (Fibk      n  k) → fibk (Fibk (n−1) (write (K1 n  k)))

    (KK (K1 n  k) n1) → fibk (Fibk (n−2) (write (K2 n1 k)))
    (KK (K2 n1 k) n2) → fibk (KK (read k) (n1 + n2))
    (KK K0        x ) → x
fib n            =      fibk (Fibk n (write K0))1
```

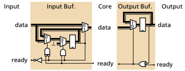
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

```
sum lp s =
 case read lp of
   Nil       → s
   Cons x xs → sum xs (s + x)
```



Input and Output Buffers



Combinational paths broken:

Input buffer breaks *ready* path

Output buffer breaks *data/valid* path