

Haskell-to-Hardware: The FHW Project

Stephen A. Edwards

Columbia University

Octopi Workshop
Chalmers University of Technology
Gothenburg, Sweden
December 2018

Motivation: Specialized Accelerators and Dark Silicon

Related Work

FHW: Functional Programs to Hardware

Algebraic Data Types in Hardware

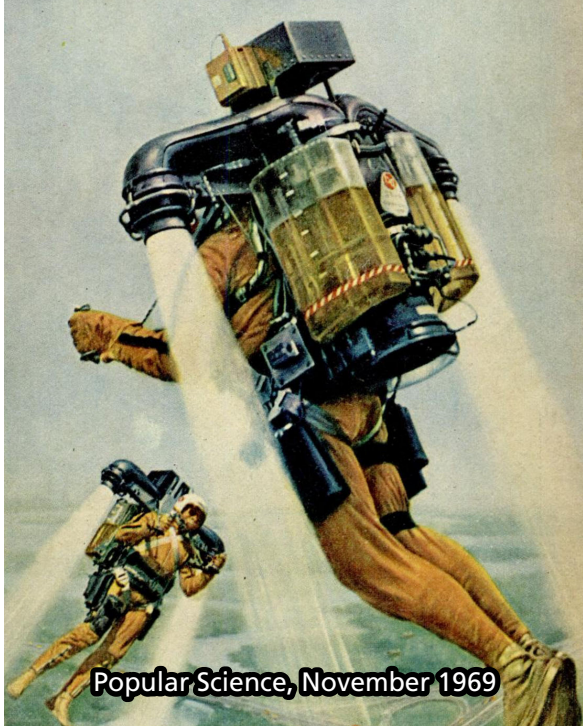
Implementing Recursion in Hardware

Functional IR to Dataflow

Dataflow to Hardware

Synthesizing Parallel Memory Systems

Conclusions

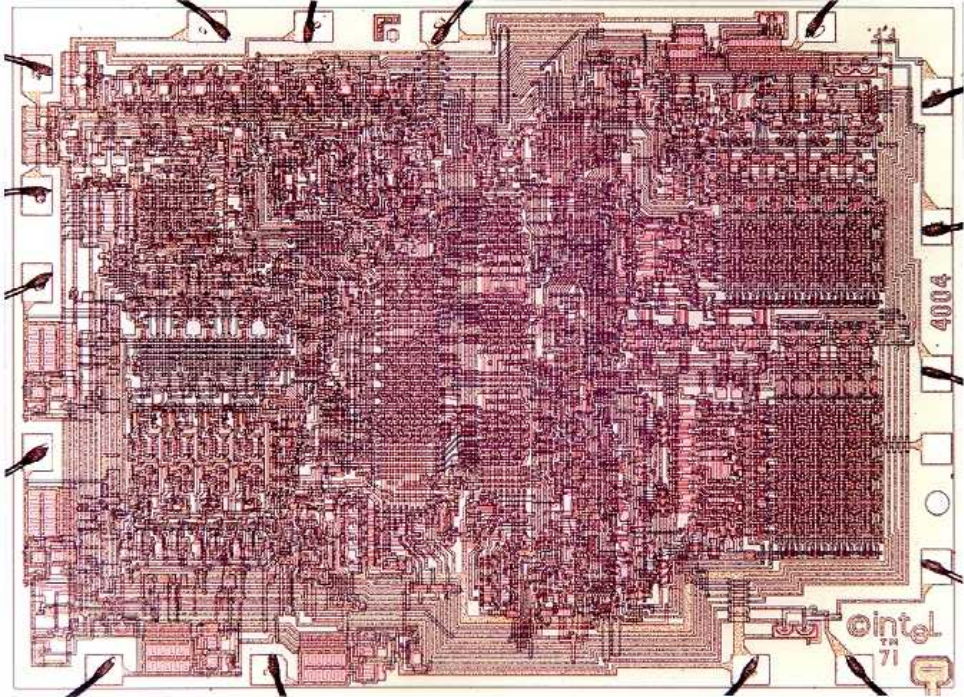


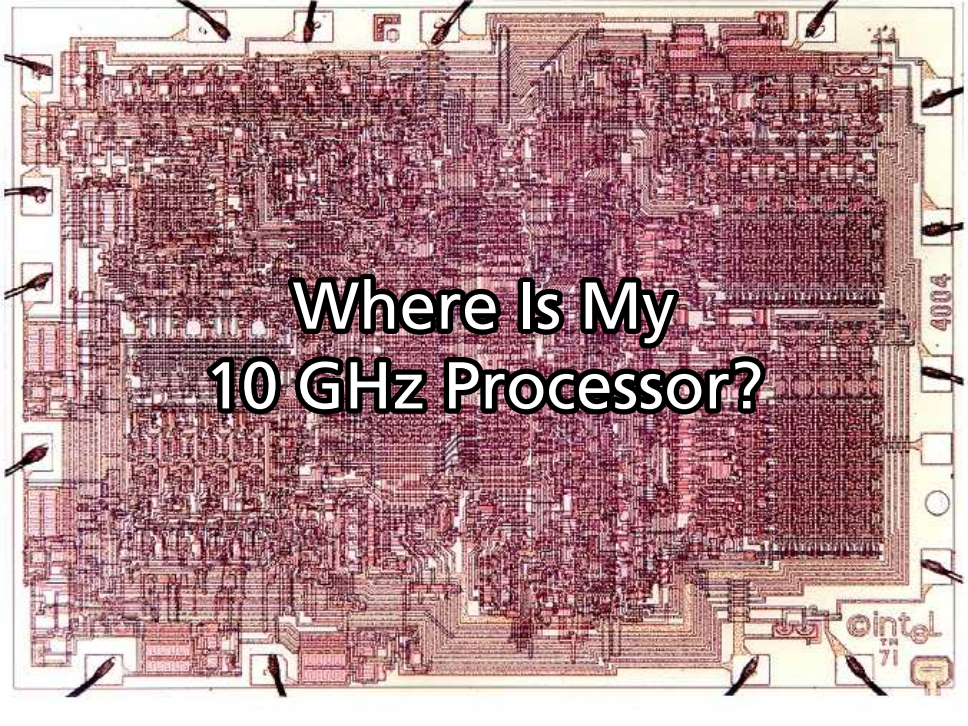
Popular Science, November 1969



Where Is My Jetpack?

Popular Science, November 1969



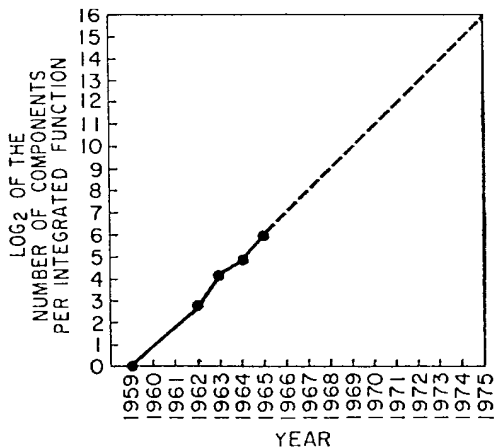


**Where Is My
10 GHz Processor?**

Intel
TM
71

4004

Moore's Law

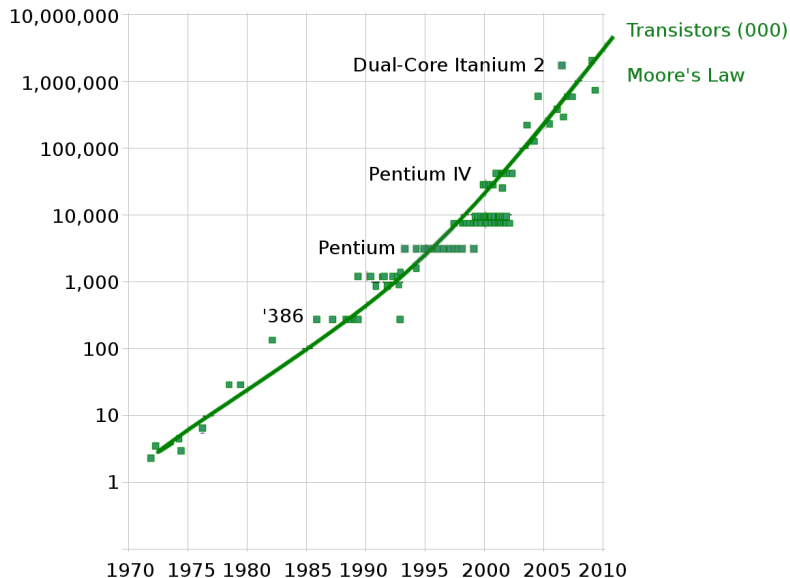


“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.”

Closer to every 24 months

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

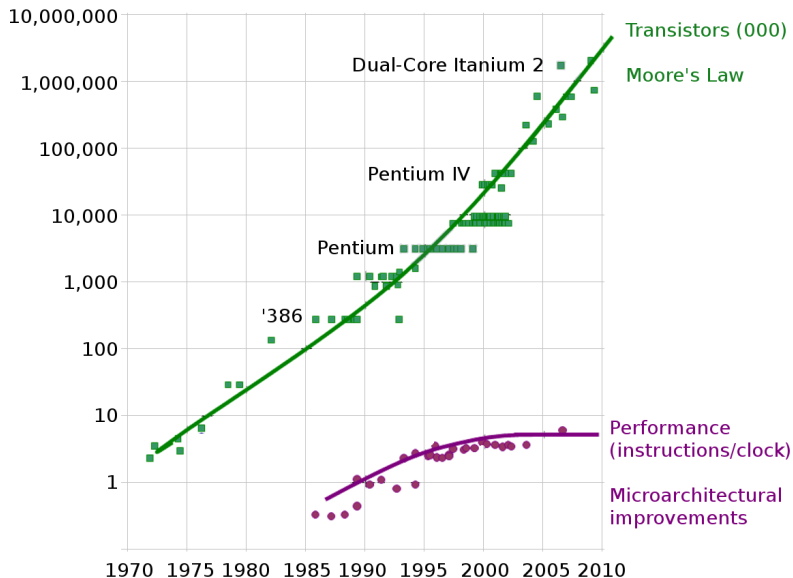
Intel CPU Trends



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

Data: Intel, Wikipedia, K. Olukotun

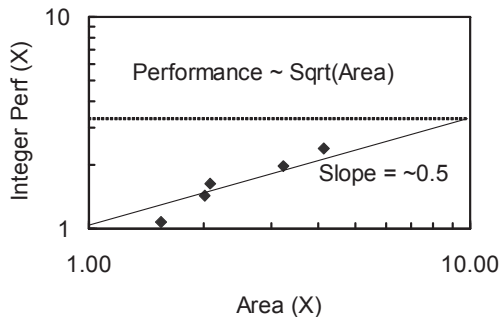
Intel CPU Trends



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

Data: Intel, Wikipedia, K. Olukotun

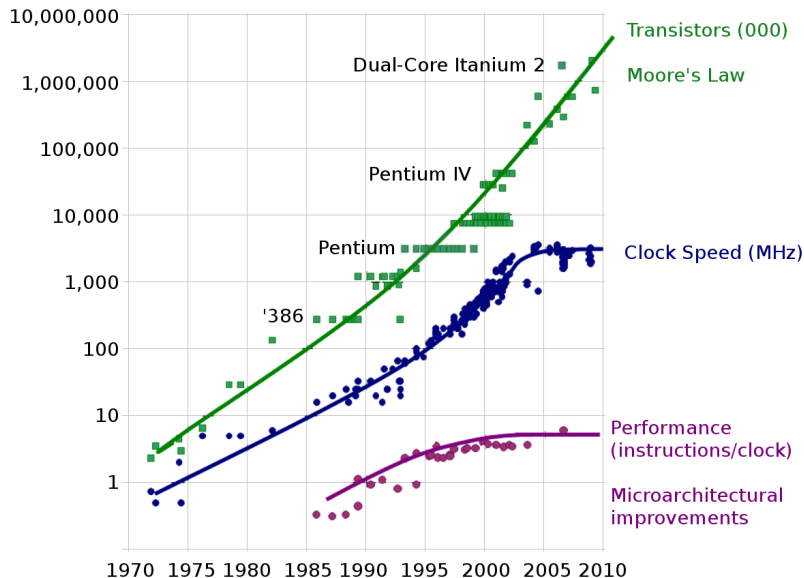
Pollack's Rule: Diminishing Returns for Processors



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

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

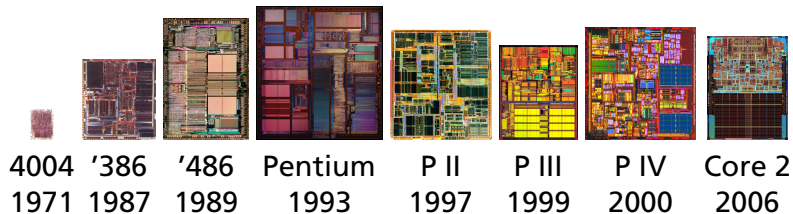
Intel CPU Trends



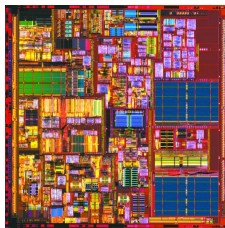
Sutter, *The Free Lunch is Over*, DDJ 2005.

Data: Intel, Wikipedia, K. Olukotun

Intel Processors to Scale



What Happened in 2005?

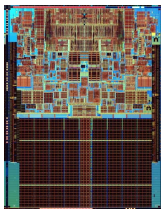


Pentium 4

2000

1 core

Transistors: 42 M

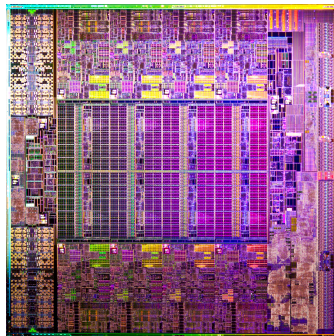


Core 2 Duo

2006

2 cores

291 M



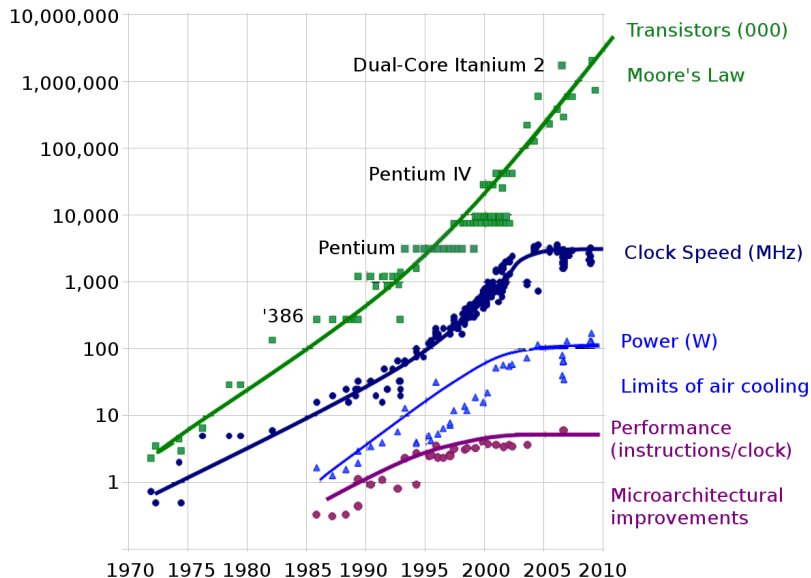
Xeon E5

2012

8 cores

2.3 G

Intel CPU Trends



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

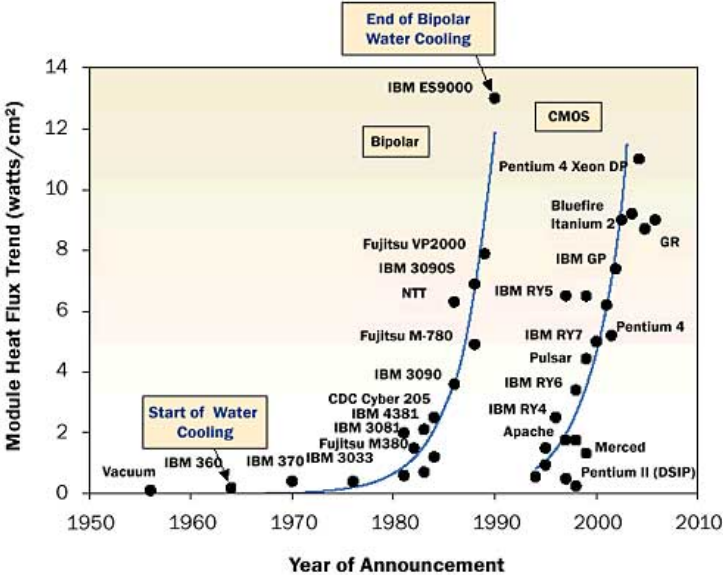
Data: Intel, Wikipedia, K. Olukotun

The Cray-2: Immersed in Fluorinert



1985 ECL 150 kW

Heat Flux in IBM Mainframes: A Familiar Trend



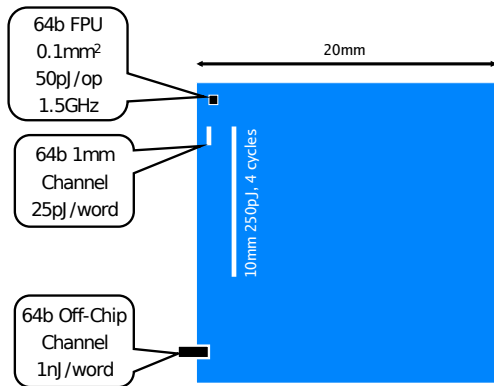
Schmidt. *Liquid Cooling is Back*. Electronics Cooling. August 2005.

Liquid Cooled Apple Power Mac G5



2004 CMOS 1.2 kW

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*

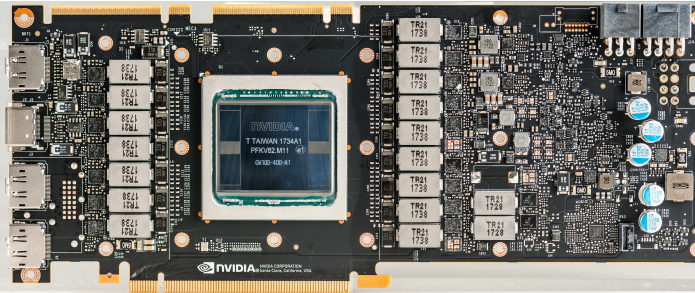
Parallelism for Performance; Locality for Efficiency



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 Titan V



The NVIDIA Titan V/Volta GV100-400 (Dec 2017)

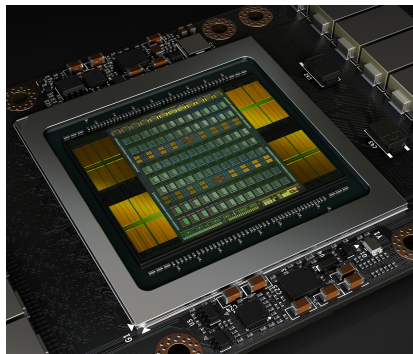
Speed	110 TFLOP/s
Frequency	1200 MHz
Power	250 W
Process	12 nm
Transistors	21.1 G
Area	815 mm ²
CUDA Cores	5120

Memory

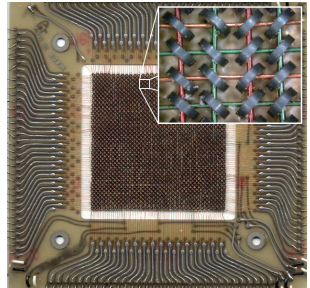
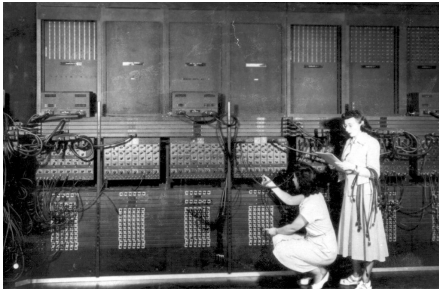
Size	12 GB
Bus width	3072 bits
Clock	850 MHz
Bandwidth	652.8 Gb/s

Price

\$3000



The Future is Wires and Memory



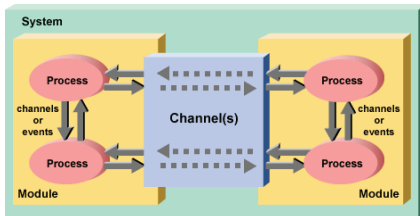
Dark Silicon



Related Work

Xilinx's Vivado (Was xPilot, AutoESL)

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

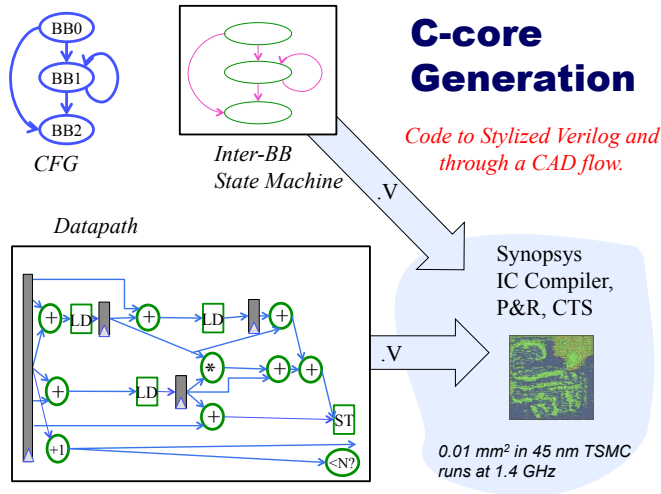


- Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
 - Port / IO interfaces, bit-vector manipulations, cycle-level notations

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

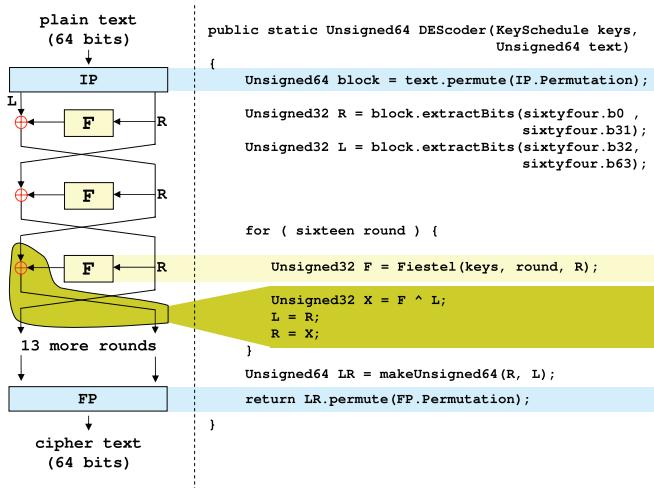


Fig. 2. Block level diagram of DES and Lime code snippet

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

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

Goldstein et al.'s Phoenix

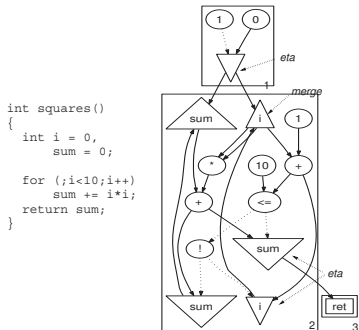


Figure 3: C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)

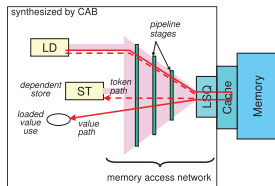


Figure 8: Memory access network and implementation of the value and token forwarding network. The LOAD produces a data value consumed by the oval node. The STORE node may depend on the load (i.e., we have a token edge between the LOAD and the STORE, shown as a dashed line). The token travels to the root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

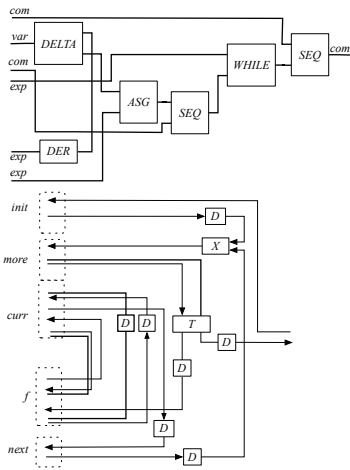


Figure 1. In-place map schematic and implementation

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

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

GCD Flip Rule

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

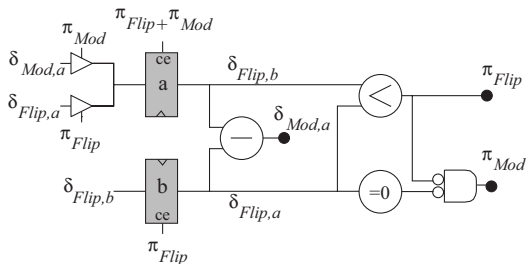


Figure 1.3 Circuit for computing $\text{Gcd}(a, b)$ from Example 1.

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
```

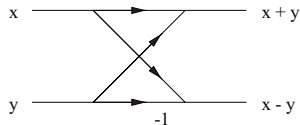


Figure 9: A butterfly

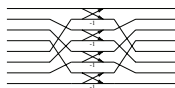


Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures

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

Kuper et al.'s Clash

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

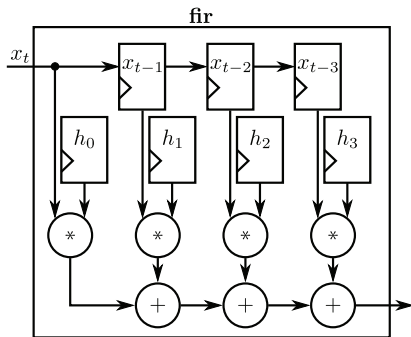


Fig. 6. 4-taps FIR Filter


More operational Haskell specifications of regular structures

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

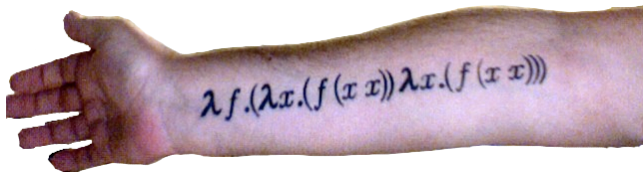
FHW: Functional Programs to Hardware

Deterministic Concurrency: A Fool's Errand?

What Models of Computation Provide Deterministic 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



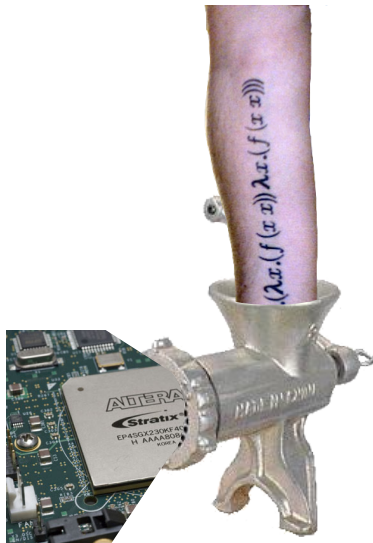
Our Project: Functional Programs to Hardware



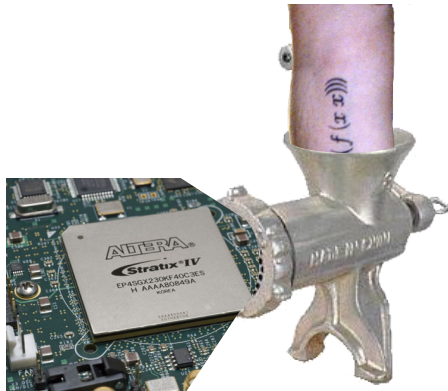
Our Project: Functional Programs to Hardware



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



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?



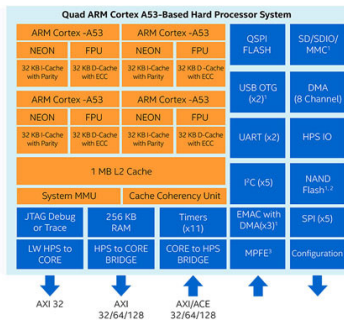
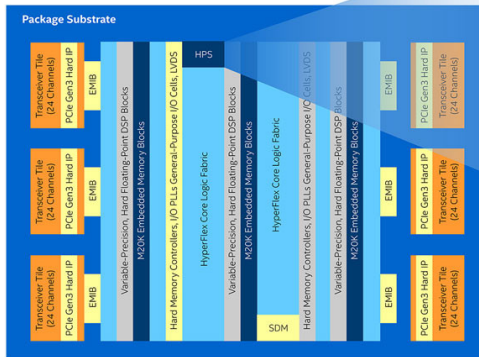
Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

A High-End FPGA: Intel/Altera Stratix 10

6847 dual-ported 2.5KB memory blocks; 16 MB total

3960 36-bit 500 MHz DSP blocks (MAC-oriented datapaths)

2M 6-input LUTs; 14 nm feature size



To Implement Real Algorithms, We Need

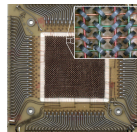
Structured, recursive data types



Recursion to handle recursive data types



Memories



Memory Hierarchy



Algebraic Data Types in Hardware

Bit vectors with tag bits
Recursive types use pointers

The Type System: Algebraic Data Types

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

```
type ::= Type
         | Constr Type* | ... | Constr Type*
```

Primitive 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
```

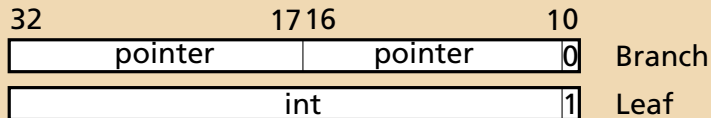
Algebraic Datatypes in Hardware: Lists

```
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
```

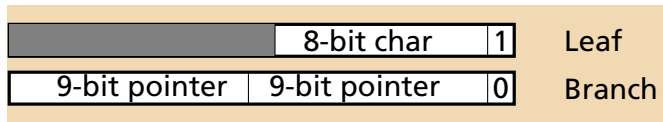
```
bit (False:xs) (Branch l _) = bit xs l           -- 0: left
bit (True:xs)  (Branch _ r) = bit xs r           -- 1: right
bit x          (Leaf c)      = c : bit x table    -- leaf
bit []         _             = []                 -- done
```

Three data types:

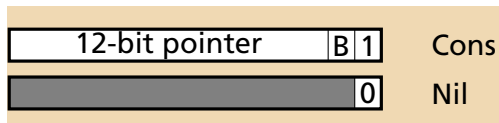
Input bitstream	[Bool] (list of Booleans)
Output character stream	[Char] (list of Characters)
Huffman tree	HTree

Encoding the Types

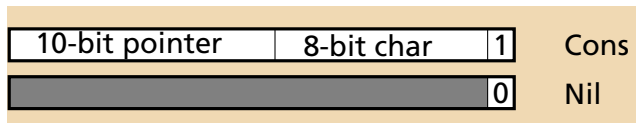
Huffman tree nodes: (19 bits)



Boolean input stream: (14 bits)



Character output stream: (19 bits)



Implementing Recursion in Hardware

Transform to continuation-passing style
Algebraic type for continuations/activation records
Tail-recursion only

Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware Synthesis from a Recursive Functional Language. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 83-93, Amsterdam, The Netherlands, October 2015.

What Do We Do With Recursion?

Compile it into tail recursion with explicit stacks

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

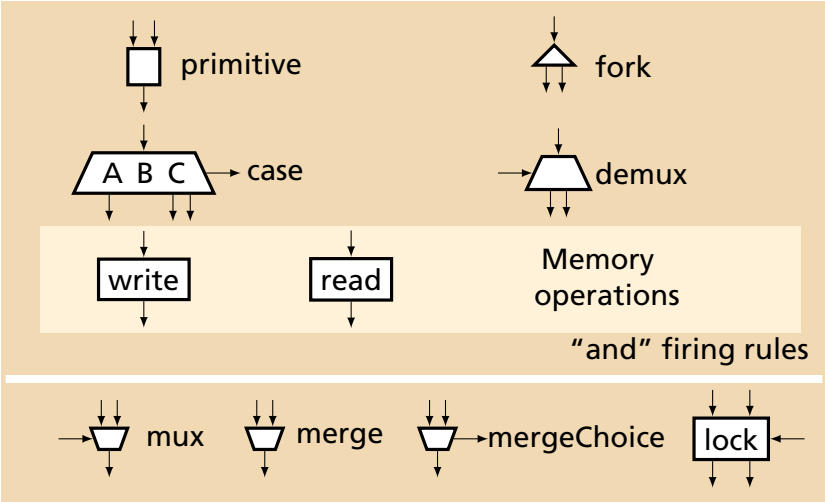
fib n = fibk (Fibk n (write K0))1

Functional IR to Dataflow

Function calls receive arguments and route results back
Non-strict functions + tail calls enable pipeline parallelism

Richard Townsend, Martha A. Kim, and Stephen A. Edwards.
From Functional Programs to Pipelined Dataflow Circuits. In
Proceedings of Compiler Construction (CC), pages 76-86, Austin,
Texas, February 2017.

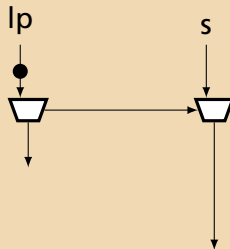
Dataflow Node Library



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)
```

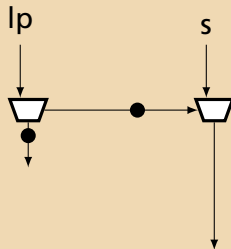


Functional to Dataflow

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 function: body starts evaluating lp before s is available



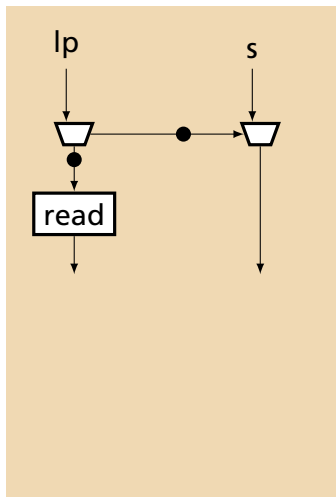
Functional to Dataflow

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)
```

Read: pointer → data

Write: data → pointer



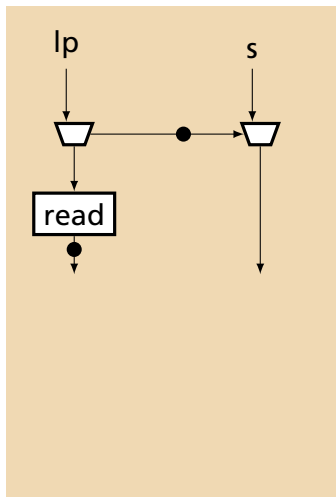
Functional to Dataflow

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)
```

Read: pointer → data

Write: data → pointer

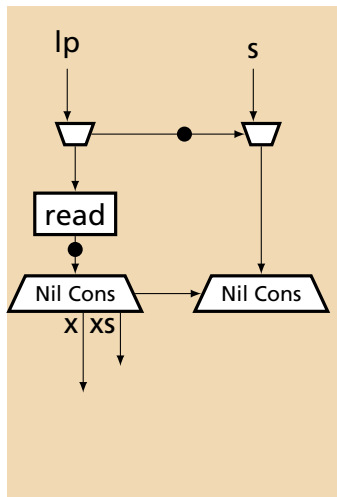


Functional to Dataflow

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)
```

Pattern matching with a decomposition mux

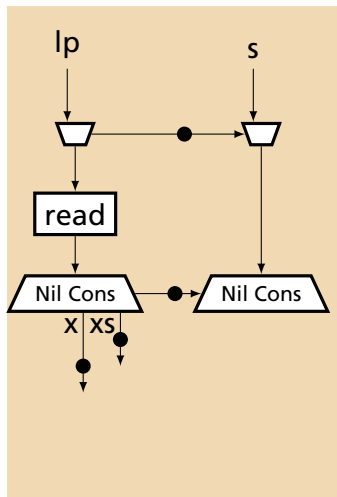


Functional to Dataflow

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)
```

Pattern matching with a decomposition mux

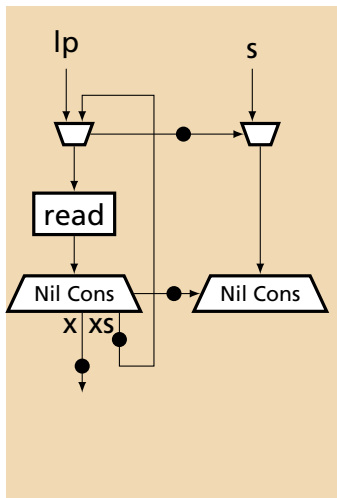


Functional to Dataflow

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)
```

Tail recursion: physical loop

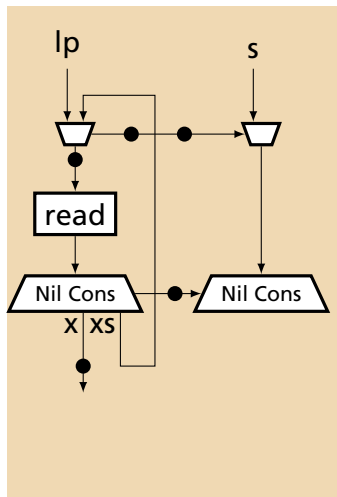


Functional to Dataflow

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-strictness enables
pipeline parallelism: second
list element is read before
first processed

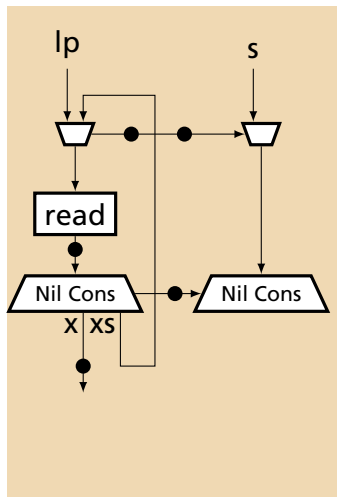


Functional to Dataflow

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)
```

Buffer sizes affect pipeline depth

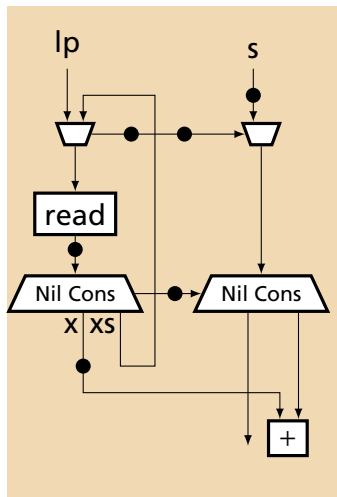


Functional to Dataflow

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)
```

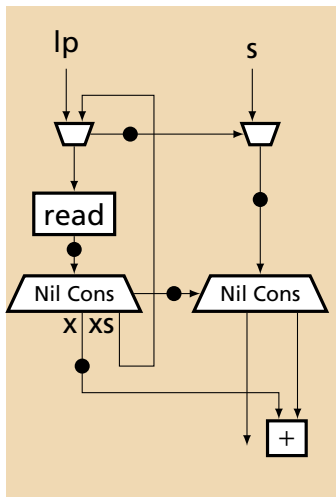
s arrives: can start computing sum



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

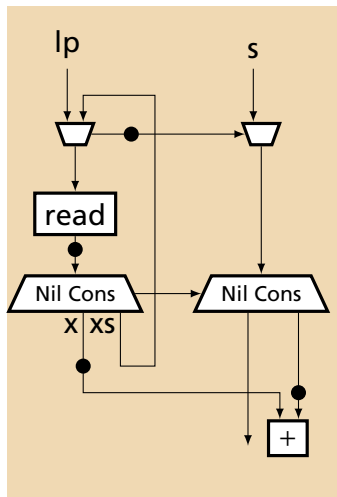
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

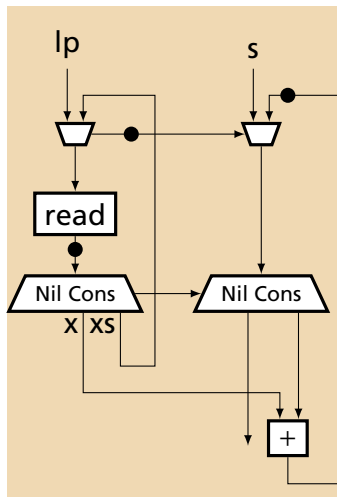
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

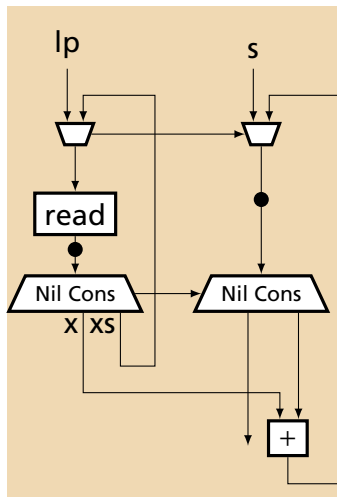
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

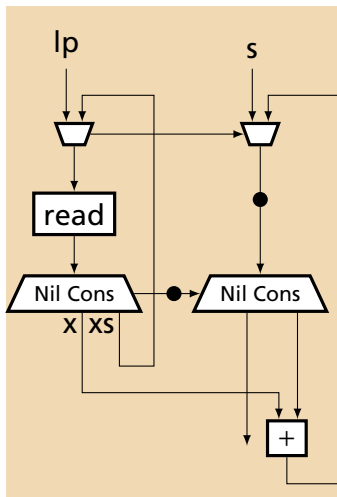
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

Sum a list using an accumulator and tail-recursion

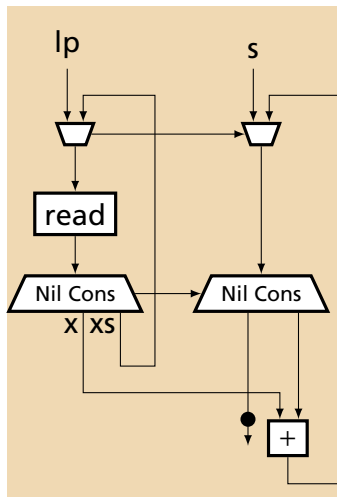
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



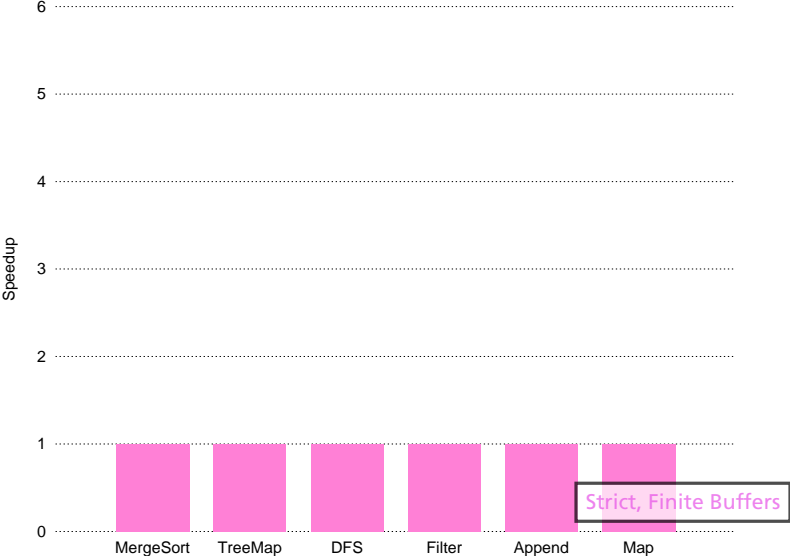
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

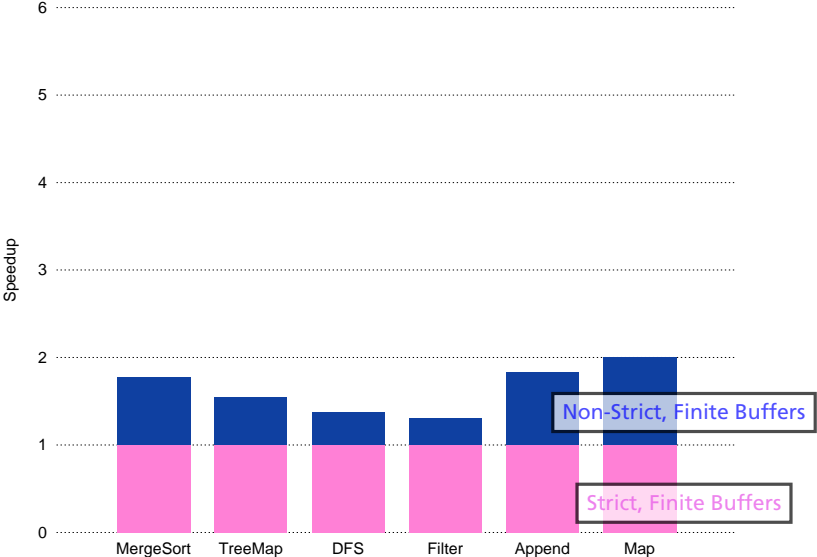
```
sum lp s =  
  case read lp of  
    Nil      → s  
    Cons x xs → sum xs (s + x)
```



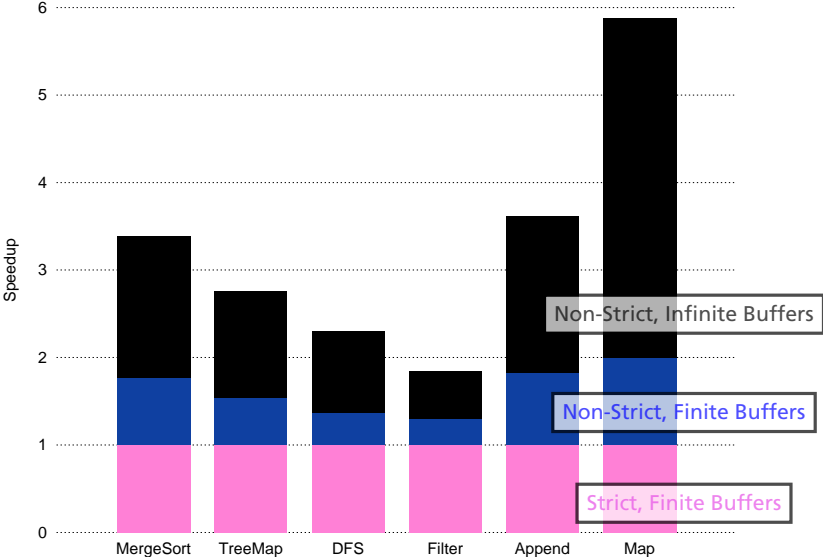
Non-Strict Functions Run Faster



Non-Strict Functions Run Faster



Non-Strict Functions Run Faster



Dataflow to Hardware

Valid-ready handshake protocol

Data and control buffers break combinational cycles

Compositionality from prohibiting paths from ready to valid

Fork outputs are non-strict

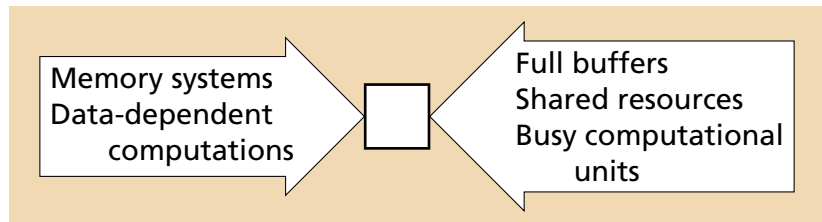
Nondeterministic merge with choice output

Stephen A. Edwards, Richard Townsend, and Martha A. Kim.

Compositional Dataflow Circuits. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 175–184, Vienna, Austria, September 2017.

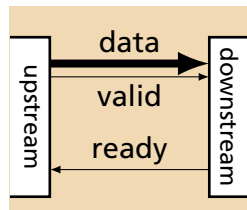
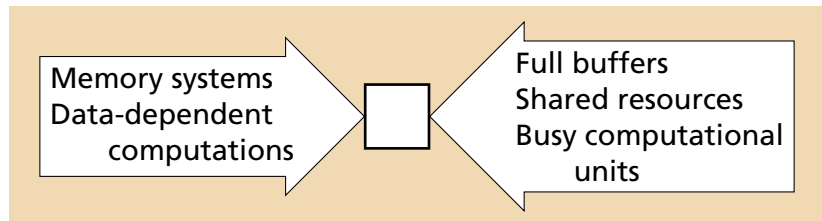
Patience Through Handshaking

Want *patient* blocks to handle delays from



Patience Through Handshaking

Want *patient* blocks to handle delays from



valid	ready	Meaning
1	1	Token transferred
1	0	Token valid; held
0	–	No token to transfer

Latency-insensitive Design (Carloni et al.)

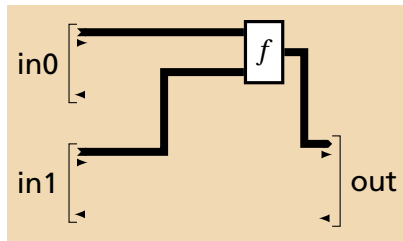
Elastic Circuits (Cortadella et al.)

FIFOs with backpressure

Combinational Function Block

Strict/Unit Rate:

All input tokens required to produce an output



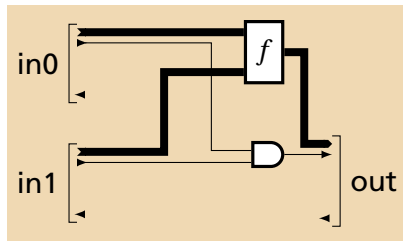
Datapath

Combinational function ignores flow control

Combinational Function Block

Strict/Unit Rate:

All input tokens required to produce an output



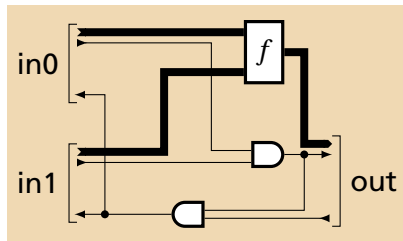
Valid network

Output valid if both inputs are valid

Combinational Function Block

Strict/Unit Rate:

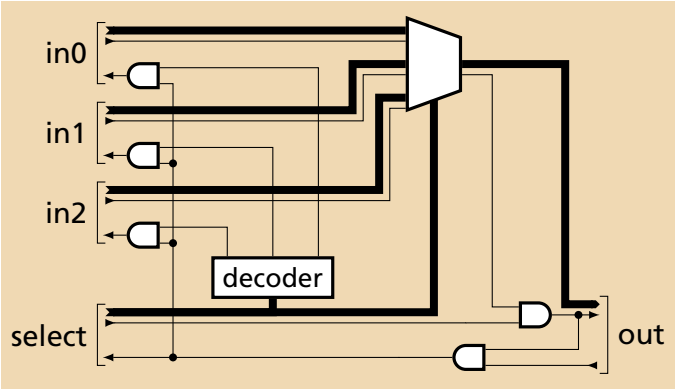
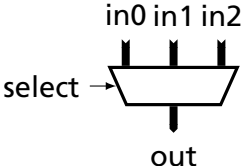
All input tokens required to produce an output



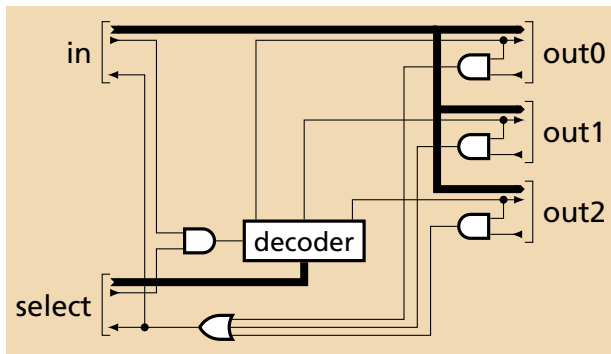
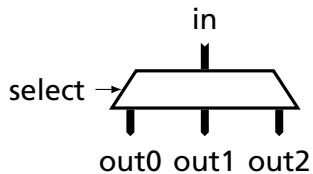
Ready network

Input tokens consumed if output token is consumed
(output is valid and ready)

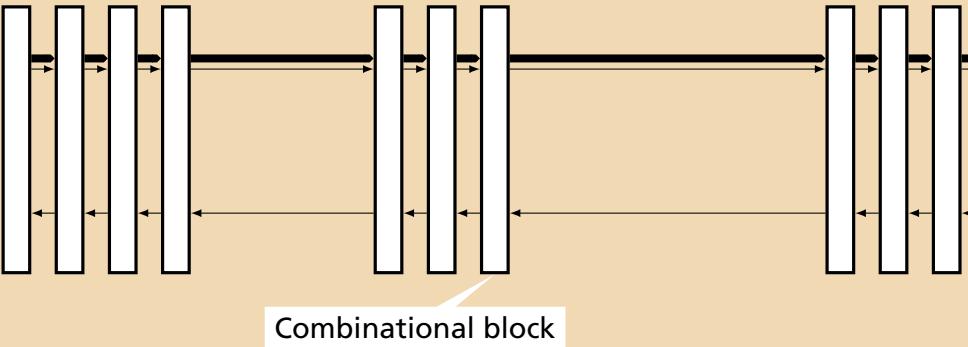
Multiplexer Block



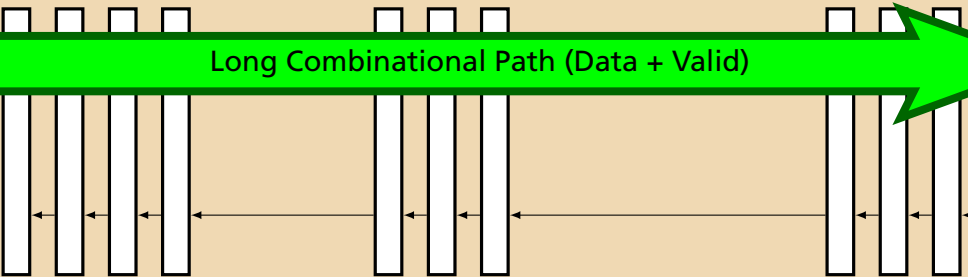
Demultiplexer Block



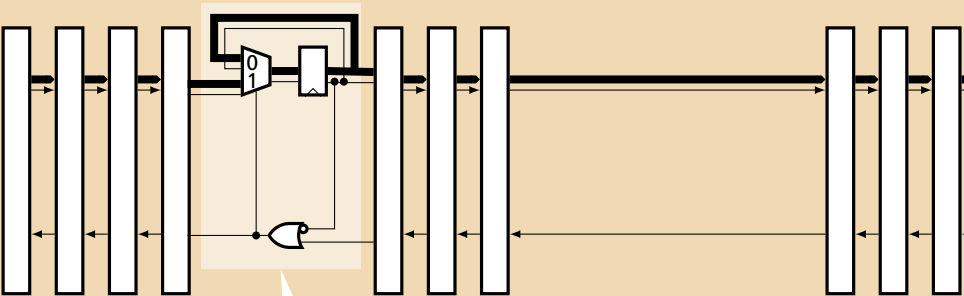
Buffering a Linear Pipeline



Buffering a Linear Pipeline

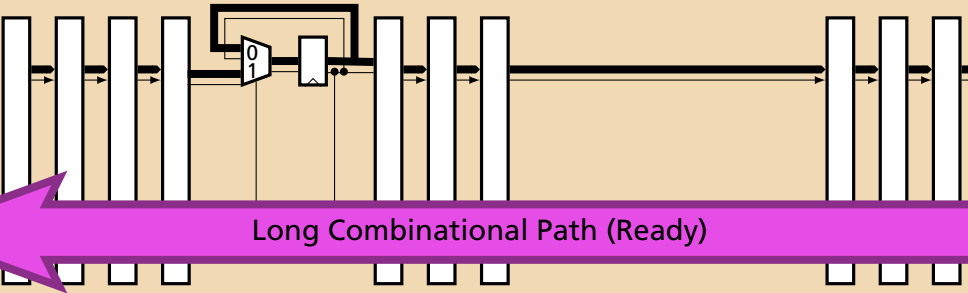


Buffering a Linear Pipeline

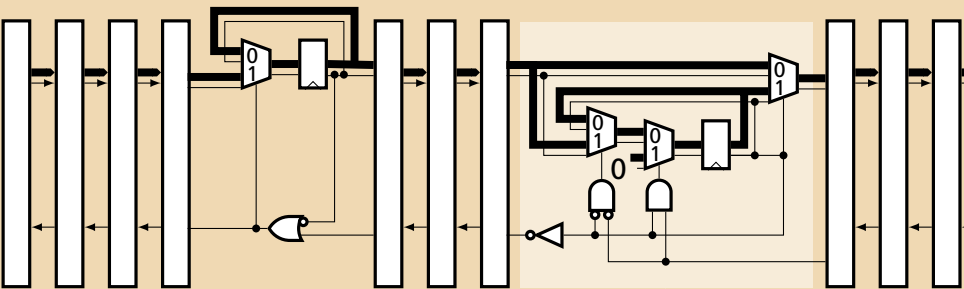


Data buffer:
Pipeline register
with valid, enable

Buffering a Linear Pipeline



Buffering a Linear Pipeline

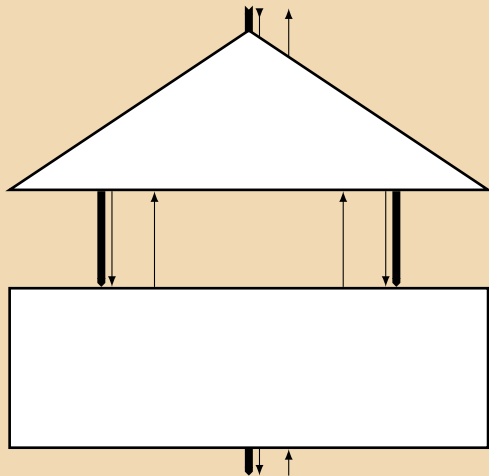


Control Buffer:
Register diverts token when
downstream suddenly stops

Cao et al. MEMOCODE 2015

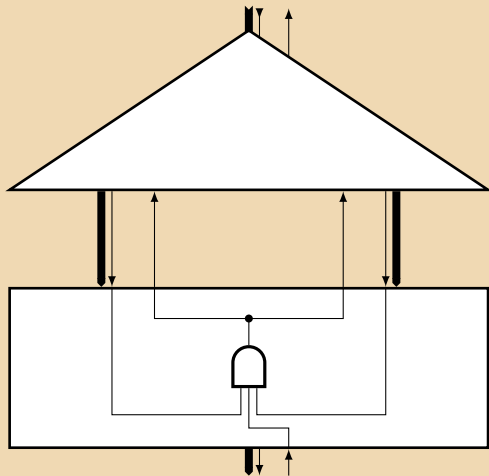
Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)

The Problem with Fork



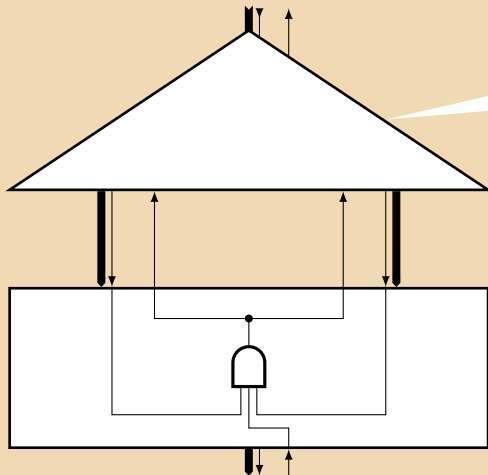
Combinational Block:
inputs ready when
both valid &
output ready

The Problem with Fork



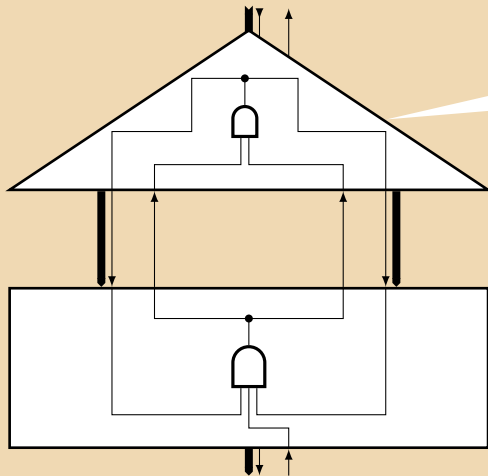
Combinational Block:
inputs ready when
both valid &
output ready

The Problem with Fork



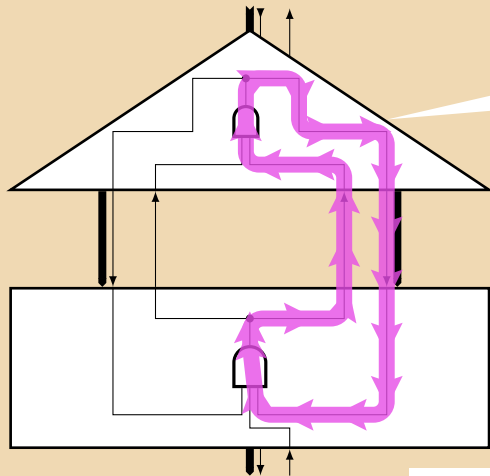
Fork:
outputs valid only
when all are ready

The Problem with Fork



Fork:
outputs valid only
when all are ready

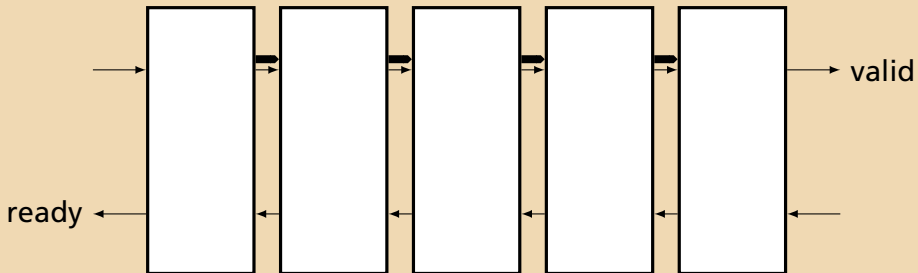
The Problem with Fork



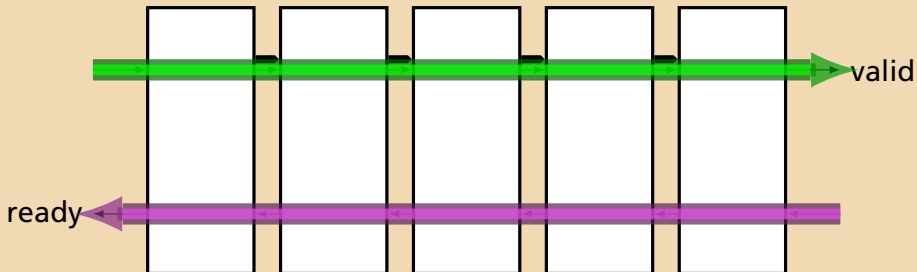
Fork:
outputs valid only
when all are ready

Oops: Combinational Cycle
This is *not* compositional

The Solution to Combinational Loops

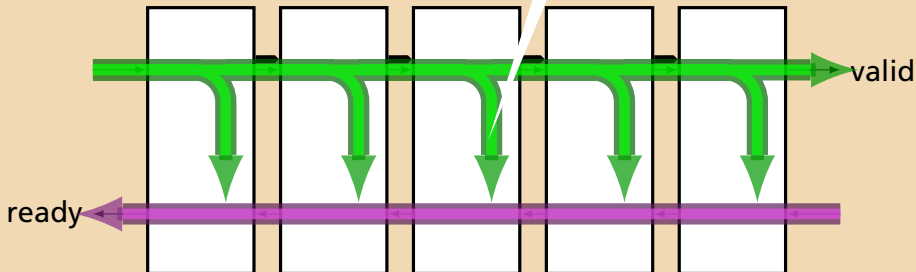


The Solution to Combinational Loops



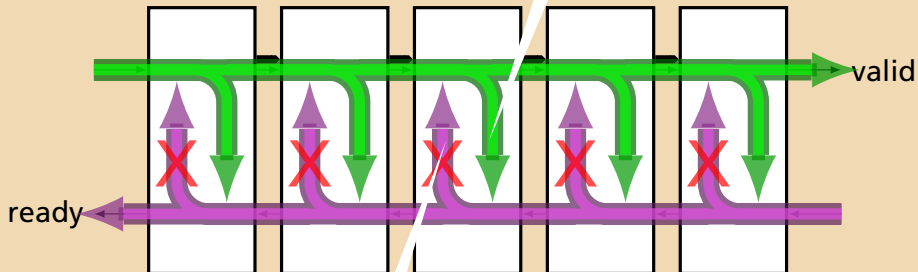
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready



The Solution to Combinational Loops

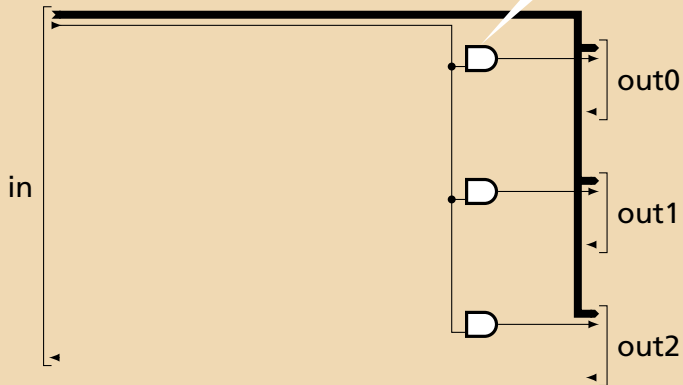
Allowed: Combinational paths from valid to ready



Prohibited: Combinational paths from ready to valid

The Solution to Fork: A Little State

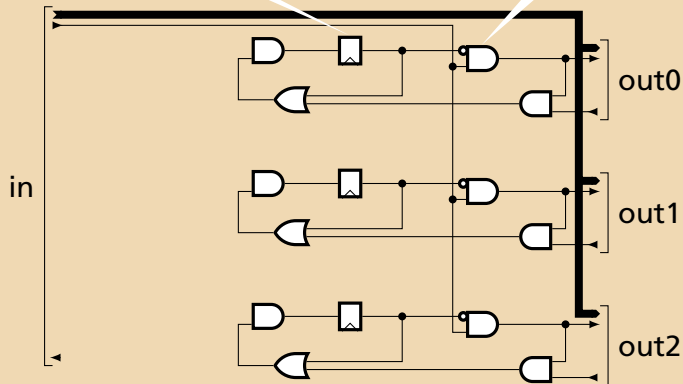
Valid out ignores ready of other outputs



The Solution to Fork: A Little State

Flip-flop set after token sent suppresses duplicates

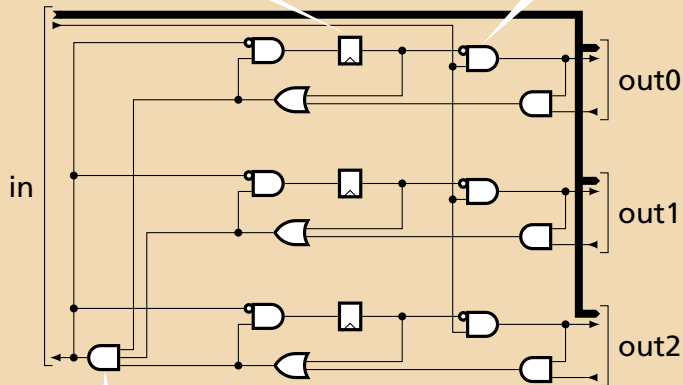
Valid out ignores ready of other outputs



The Solution to Fork: A Little State

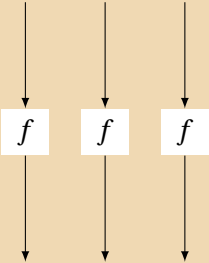
Flip-flop set after token sent suppresses duplicates

Valid out ignores ready of other outputs

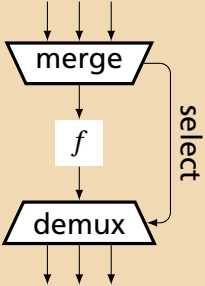


Input consumed once one token sent on every output

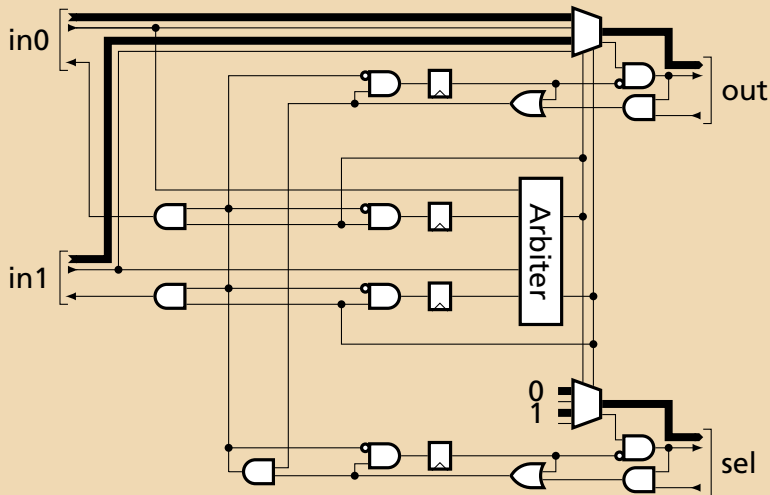
Nondeterministic Merge



Share with merge/demux



Two-Way Nondeterministic Merge Block w/ Select

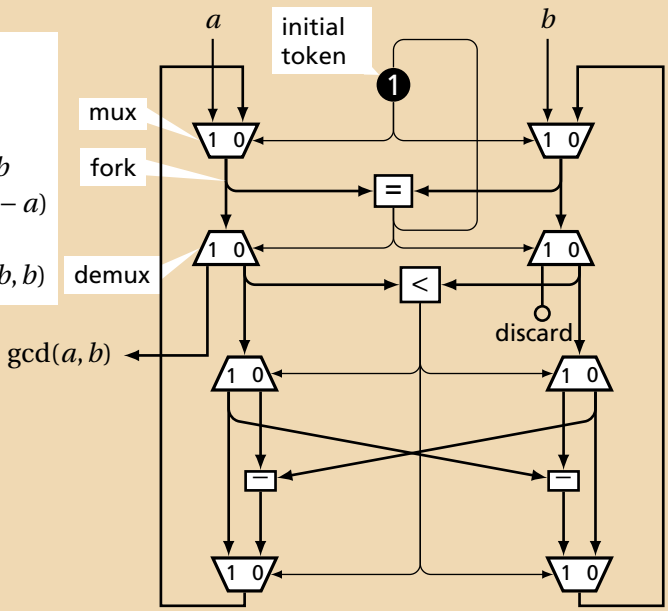


"Two-way fork with multiplexed output selected by an arbiter"

```

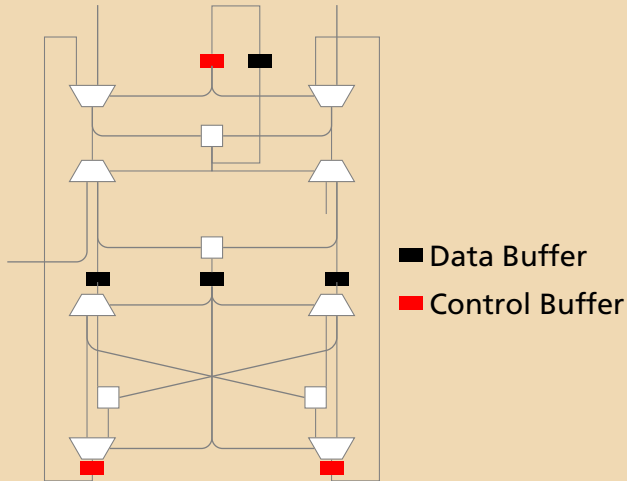
gcd(a, b) =
  if a = b
    a
  else if a < b
    gcd(a, b - a)
  else
    gcd(a - b, b)

```



Best Buffering for GCD (Manually Obtained)

Each loop has one of each buffer



Synthesizing Parallel Memory Systems

Duplicate the recursive task

Assign separate cache partitions to parallel tasks

Works best with balanced workload

Richard Townsend, Martha A. Kim, and Stephen A. Edwards.
Synthesizing Parallel Hardware Implementations of
Divide-and-Conquer Algorithms. Submitted to *Proceedings of the
Design Automation Conference (DAC)*, 2019

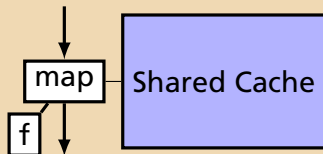
Divide-and-Conquer Functions: Inherently Parallel

`map :: Tree → Tree`

`map t = case t of`

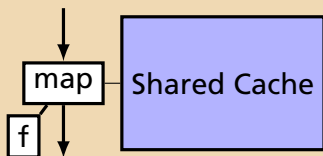
`Leaf → t`

`Node l x r → Node (map l) (f x) (map r)`



Transformations to Enable Parallelism

map t = ...



Transformations to Enable Parallelism

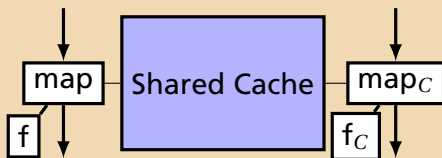
$\text{map } t = \dots$

$\text{map}_C :: \text{Tree} \rightarrow \text{Tree}$

$\text{map}_C t = \text{case } t \text{ of}$

Leaf $\rightarrow t$

Node $l \ x \ r \rightarrow \text{Node } (\text{map}_C l) \ (f_C x) \ (\text{map}_C r)$



Transformations to Enable Parallelism

$\text{map } t = \dots$

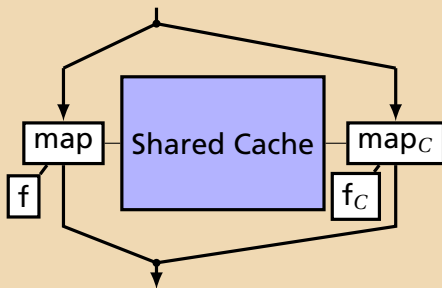
$\text{map}_C :: \text{Tree} \rightarrow \text{Tree}$

$\text{map}_C t = \dots$

$\text{map}_S t = \text{case } t \text{ of}$

Leaf $\rightarrow t$

Node $l \ x \ r \rightarrow \text{Node } (\text{map } l) \ (f \ x) \ (\text{map}_C r)$



Memory Partitioning to Exploit the Parallelism

$\text{map } t = \dots$

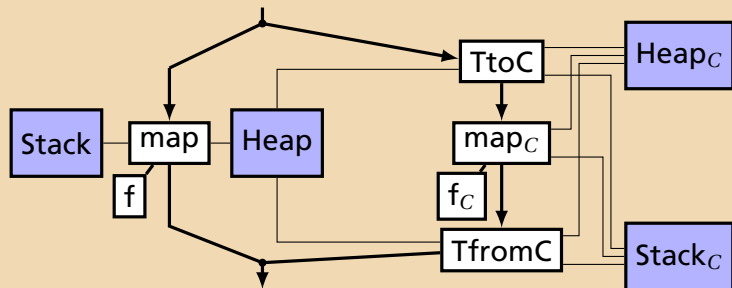
$\text{map}_C :: \text{Tree}_C \rightarrow \text{Tree}_C$

$\text{map}_C \text{ tp} = \dots$

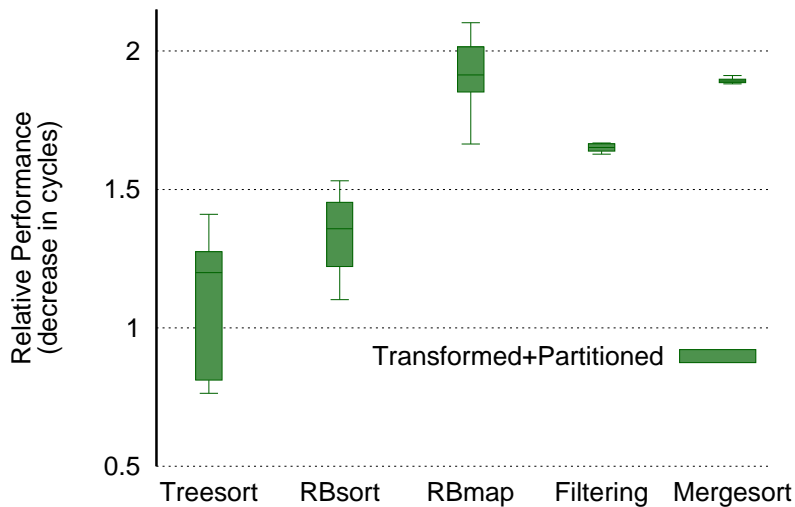
$\text{map}_S \text{ tp} = \text{case } t \text{ of}$

Leaf $\rightarrow t$

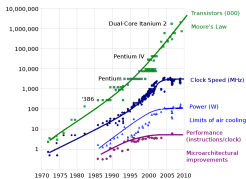
Node $l \ x \ r \rightarrow \text{Node } (\text{map } l) \ (f \ x) \ (\text{TfromC } (\text{map}_C \ (\text{TtoC } r)))$



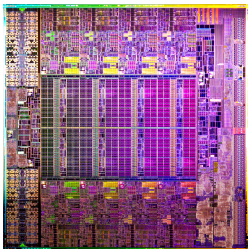
Doing This Increases Performance



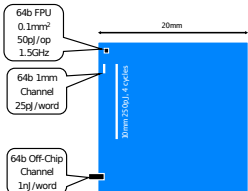
Conclusions



Moore's Law is alive and well



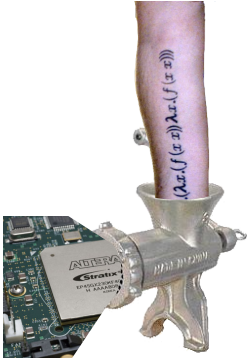
But we hit a power wall in 2005. Massive parallelism is now mandatory



Communication is the culprit



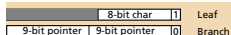
Dark Silicon with special-purpose accelerators is the future



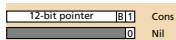
Our Project: Haskell-to-Hardware

Encoding the Types

Huffman tree nodes: (19 bits)



Boolean input stream: (14 bits)



Character output stream: (19 bits)



Implement algebraic data types as bit vectors with tags and pointers

Represent Continuations with a Type

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

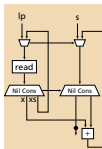
```
fibk n k = case (n,k) of
  (1, k) -> kk k 1
  (2, k) -> kk k 1
  (n, k) -> fibk (n-1) (K1 n k)
kk k a = case (k, a) of
  ((K1 n k), n1) -> fibk (n-2) (K2 n1 k)
  ((K2 n1 k), n2) -> kk k (n1 + n2)
  (K0, x) -> x
fib n = fibk n K0
```

Implement recursive functions with tail recursion and explicit types for activation records/continuations

Functional to Dataflow

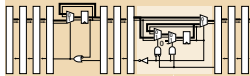
Sum a list using an accumulator and tail-recursion

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



Implement the functional IR with a dataflow network. Non-strict tail-recursive functions express pipeline parallelism

Buffering a Linear Pipeline



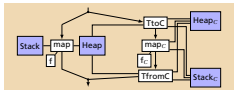
Control Buffer:
Register diverts token when
downstream suddenly stops

Cao et al. MEMOCODE 2015
Inspired by Carlson's Latency Immensive Design (e.g., MEMOCODE 2007)

Implement dataflow in hardware with compositional blocks with handshaking

Memory Partitioning to Exploit the Parallelism

```
map t = ...  
mapc :: TreeC → TreeC  
mapc tp = ...  
mapc tp = case t of  
  Leaf    → t  
  Node l x r → Node (map l) (f x) (TfromC (mapc (TtoC r)))
```



Duplicate tasks and partition caches to speed recursive algorithms