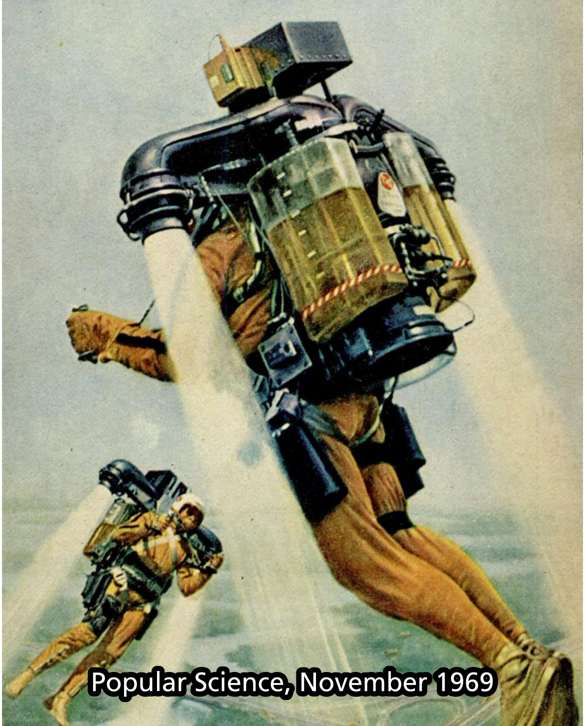
Haskell to Hardware and Other Dreams

Stephen A. Edwards

Columbia University

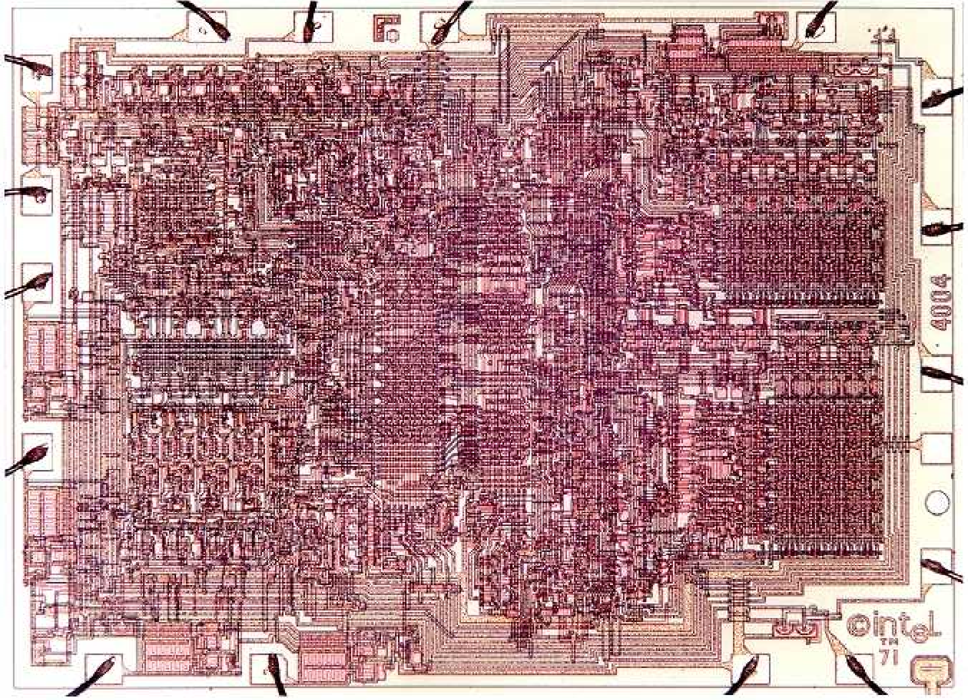
June 12, 2018

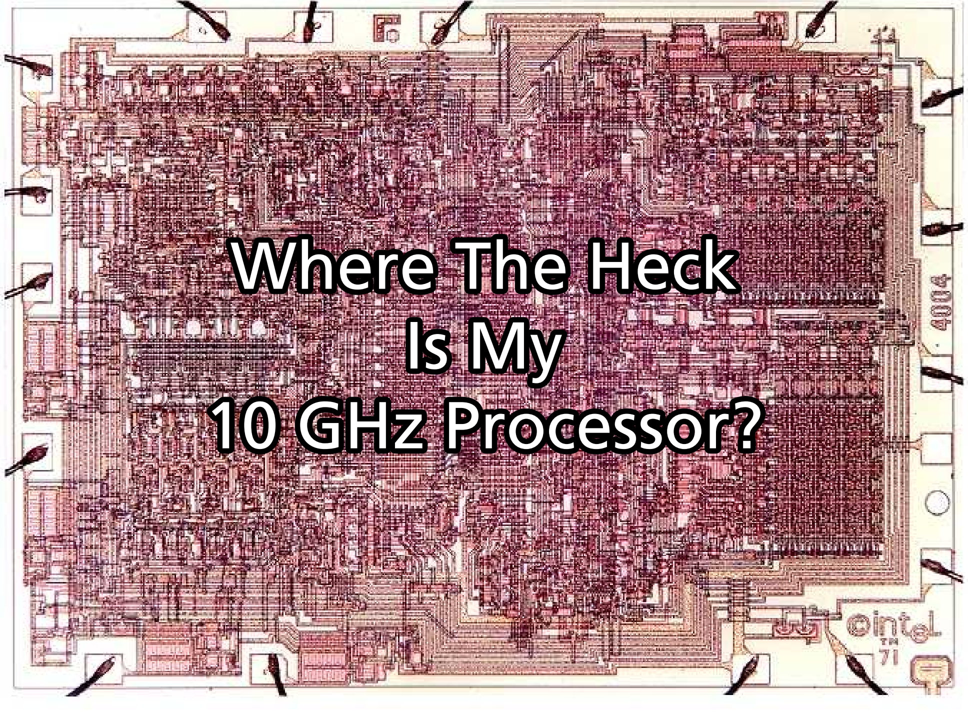


Popular Science, November 1969

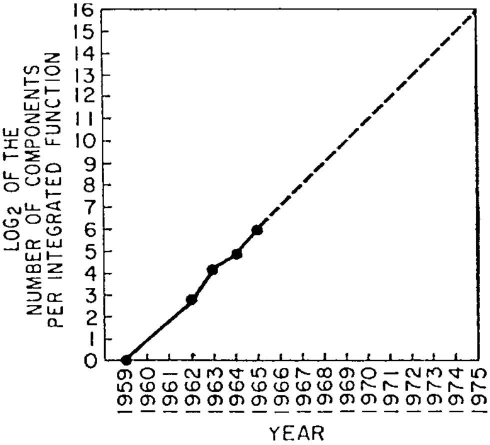
Where Is My Jetpack?

Popular Science, November 1969


4004
©intel


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

Moore's Law

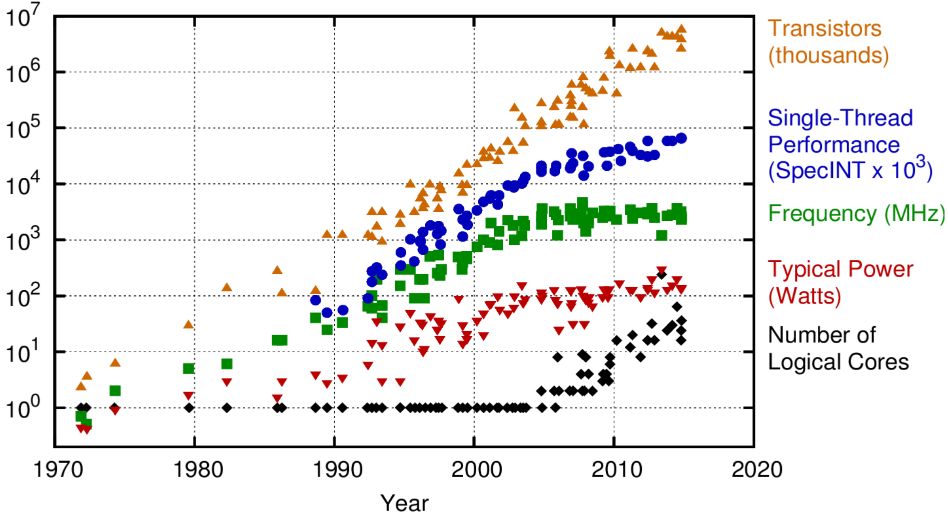


"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."

Closer to every 24 months

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

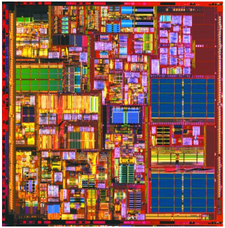
Four Decades of Microprocessors Later...

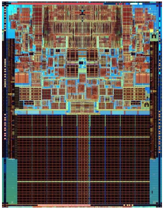


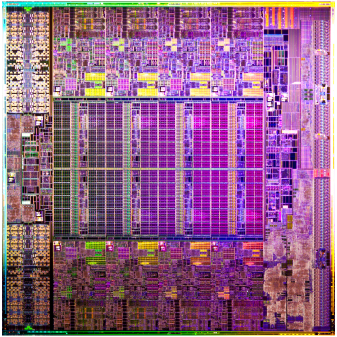
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

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

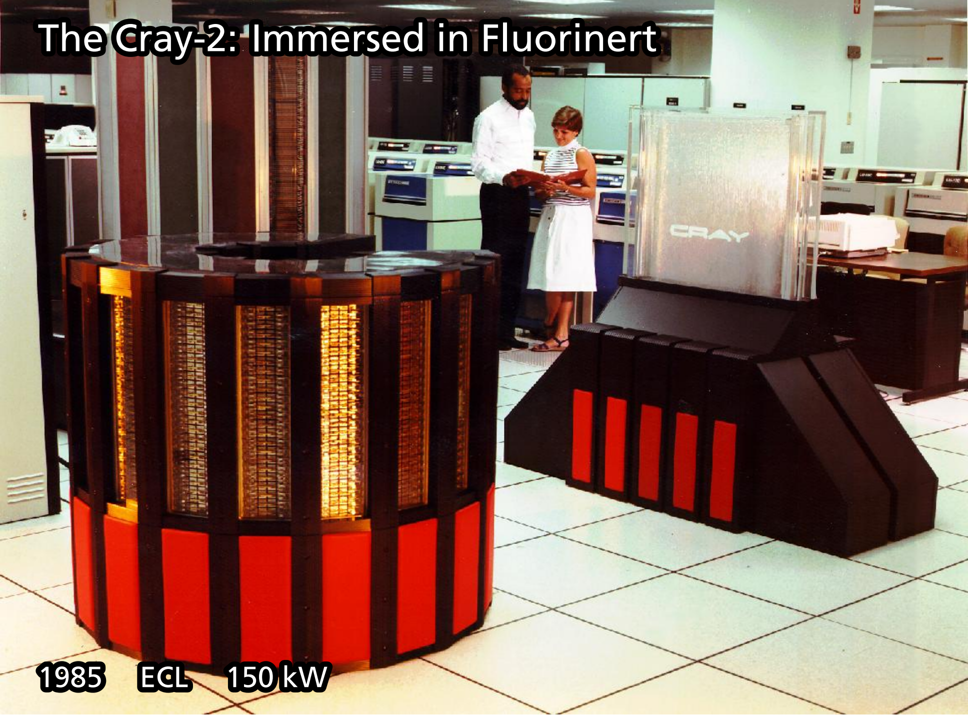
What Happened in 2005?



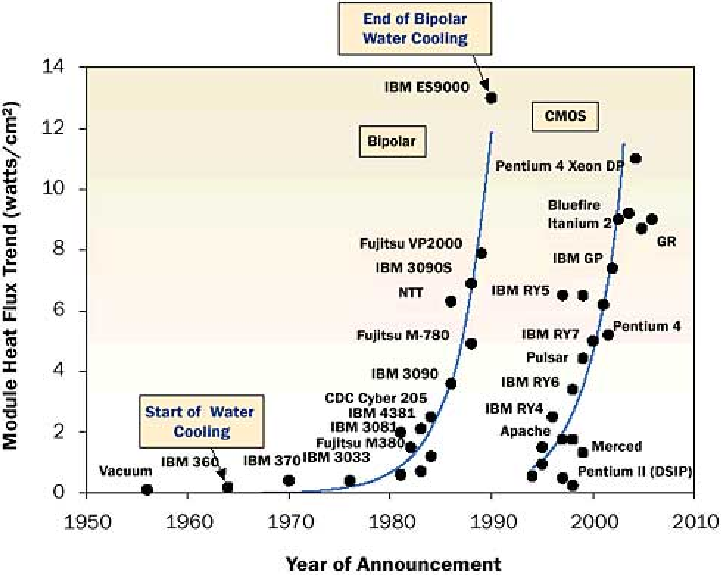




	Pentium 4	Core 2 Duo	Xeon E5
	2000	2006	2012
	1 core	2 cores	8 cores
Transistors:	42 M	291 M	2.3 G


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

Heat Flux in IBM Mainframes: A Familiar Trend

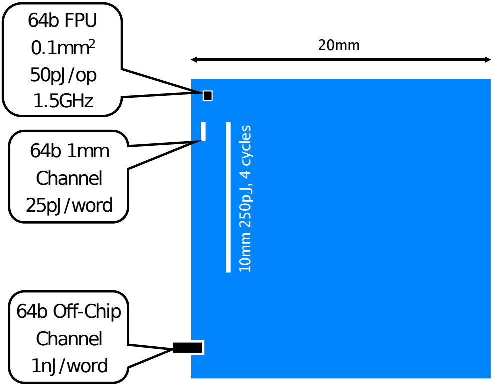


Schmidt. *Liquid Cooling is Back*. Electronics Cooling. August 2005.

Liquid Cooled Apple Power Mac G5

2004 CMOS 1.2 kW

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.

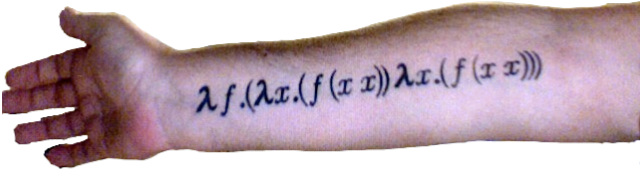
Dark Silicon

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–

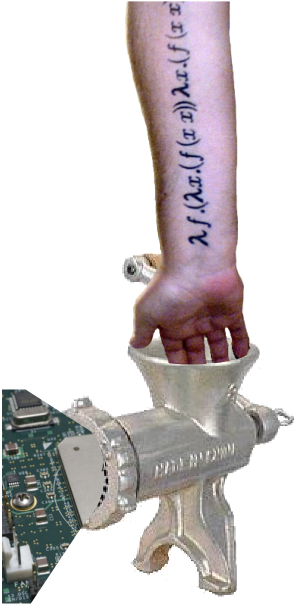
Our Project: Functional Programs to Hardware



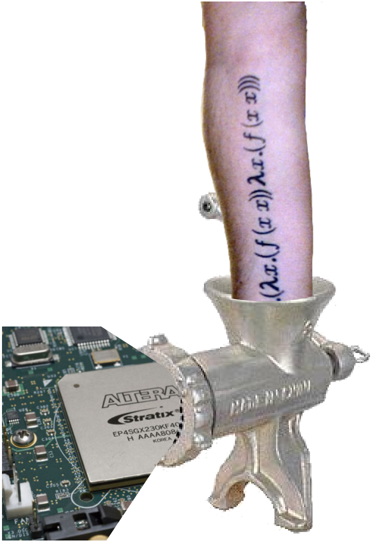
Our Project: Functional Programs to Hardware

Our Project: Functional Programs to Hardware

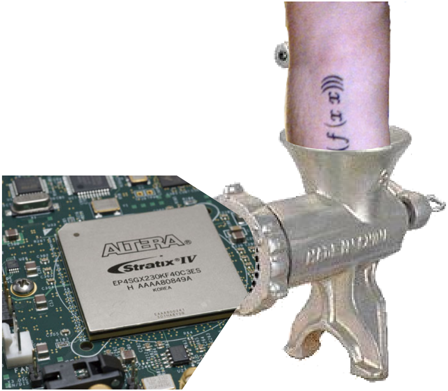
Our Project: Functional Programs to Hardware



Our Project: Functional Programs to Hardware



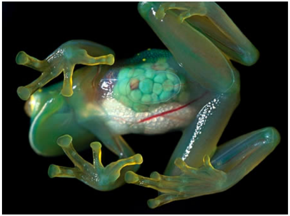
Our Project: Functional Programs to Hardware



Our Project: Functional Programs to Hardware

Why Functional?

- Referential transparency simplifies formal reasoning about programs



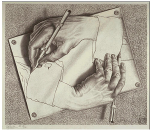
- Inherently concurrent and deterministic
(Thank Church and Rosser)

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

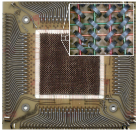
To Implement Real Algorithms, We Need

Structured, recursive data types

Recursion to handle recursive data types



Memories



Memory Hierarchy

Recursion

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

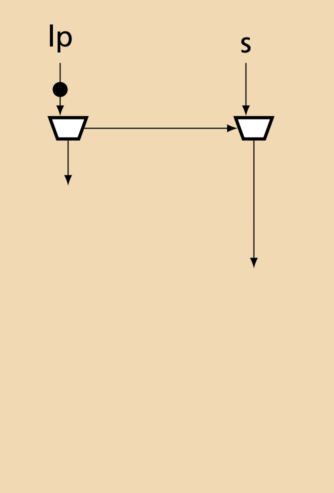
```
read   ::  CRef → Cont
write  ::  Cont → CRef
data Cont = K0 | K1 Int CRef | K2 Int CRef
data Call = Fibk Int CRef | KK Cont Int

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
```

Functional IR to Dataflow

Functional to Dataflow

[Townsend et al., CC 2017]
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)
```

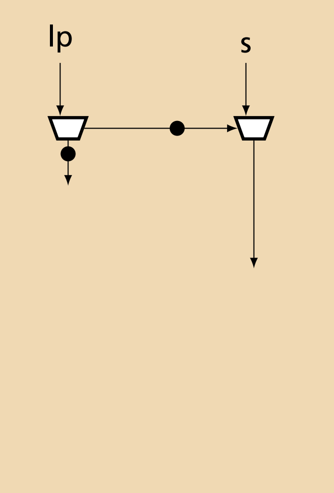


Functional to Dataflow

[Townsend et al., CC 2017]
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

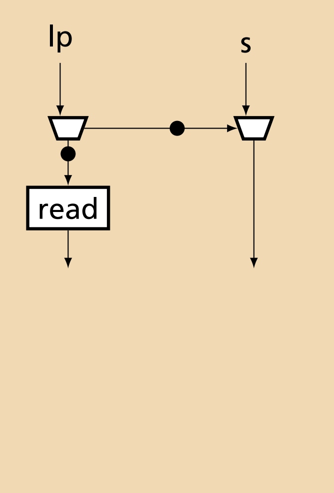


Functional to Dataflow

[Townsend et al., CC 2017]
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

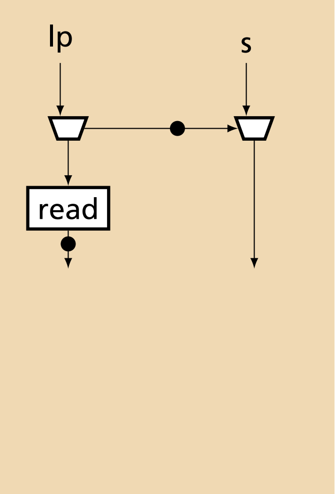


Functional to Dataflow

[Townsend et al., CC 2017]
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

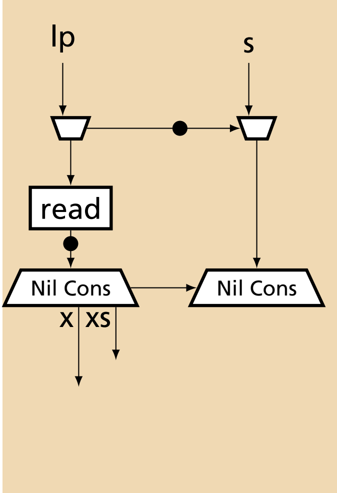


Functional to Dataflow

[Townsend et al., CC 2017]
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

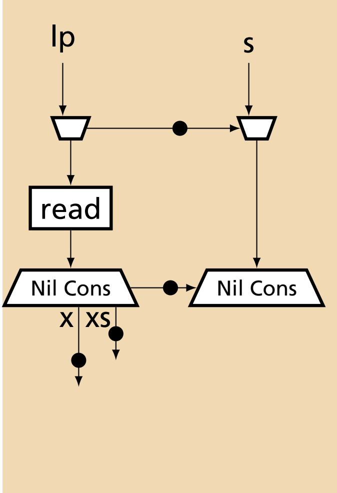


Functional to Dataflow

[Townsend et al., CC 2017]
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

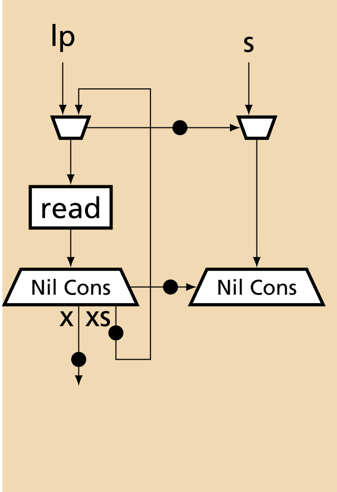


Functional to Dataflow

[Townsend et al., CC 2017]
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

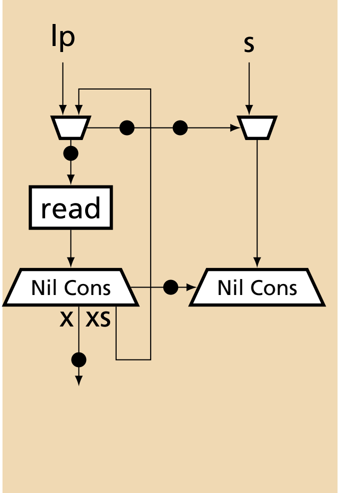


Functional to Dataflow

[Townsend et al., CC 2017]
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

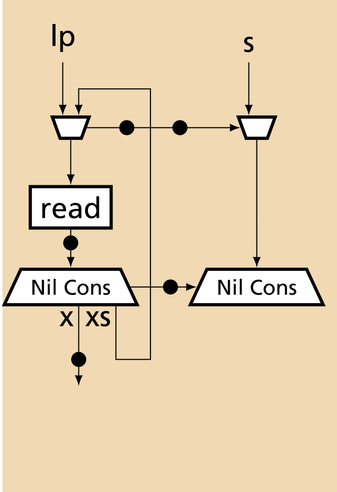


Functional to Dataflow

[Townsend et al., CC 2017]
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

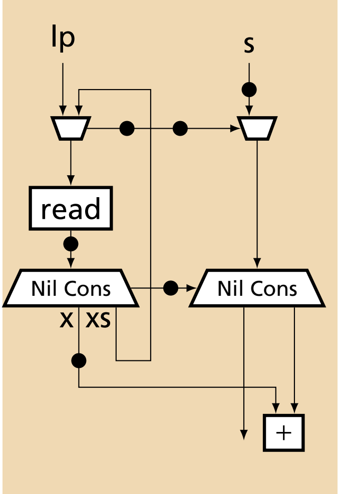


Functional to Dataflow

[Townsend et al., CC 2017]
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)
```

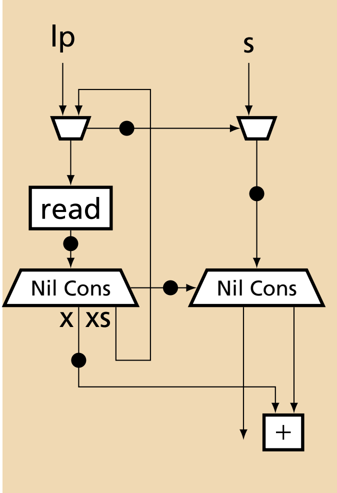
s arrives: can start computing sum



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

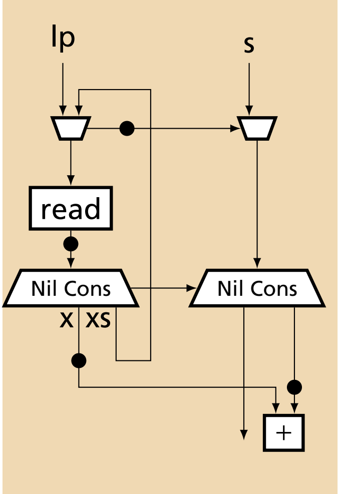
```
sum lp s =
 case read lp of
   Nil        → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

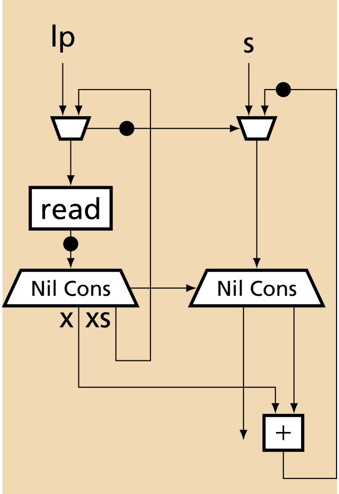
```
sum lp s =
  case read lp of
    Nil       → s
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

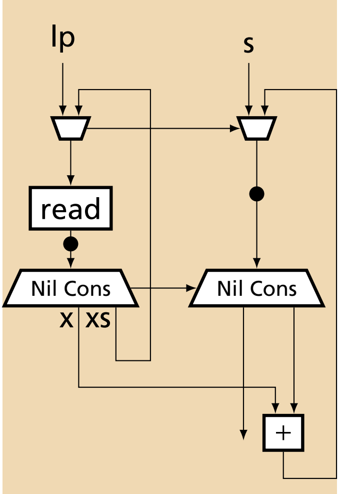
```
sum lp s =
  case read lp of
    Nil        → s
    Cons x xs → sum xs (s + x)
```



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

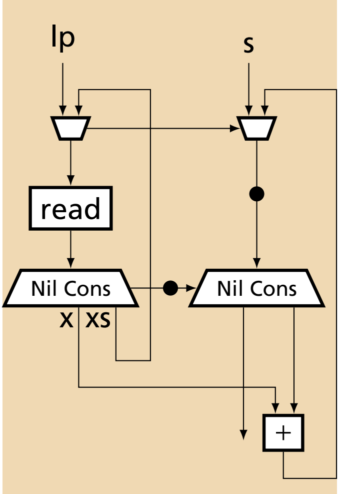
```
sum lp s =
 case read lp of
   Nil        → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

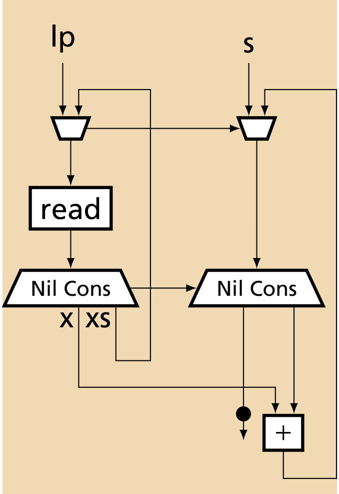
```
sum lp s =
 case read lp of
   Nil       → s
   Cons x xs → sum xs (s + x)
```



Functional to Dataflow

[Townsend et al., CC 2017]
Sum a list using an accumulator and tail-recursion

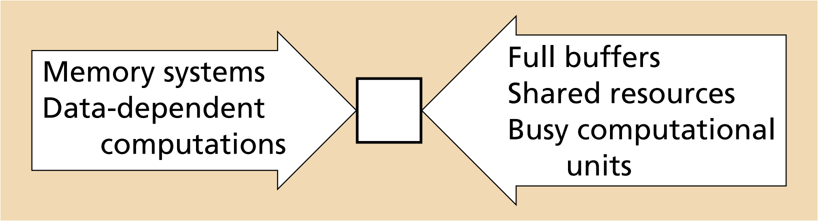
```
sum lp s =
 case read lp of
   Nil        → s
   Cons x xs → sum xs (s + x)
```



Dataflow to Hardware

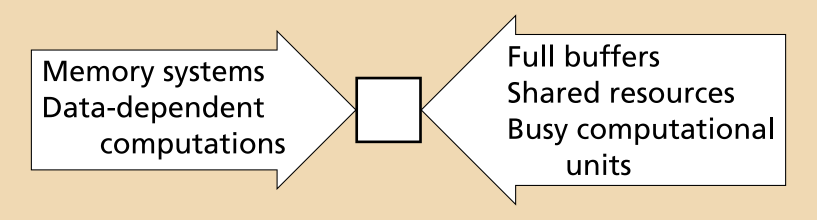
Patience Through Handshaking

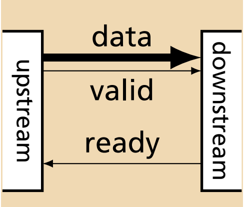
Want *patient* blocks to handle delays from



Patience Through Handshaking

Want *patient* blocks to handle delays from





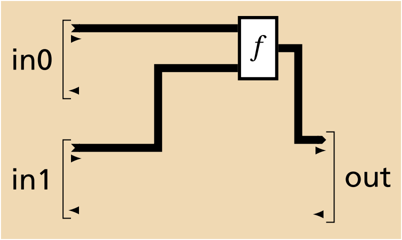
valid	ready	Meaning
1	1	Token transferred
1	0	Token valid; held
0	–	No token to transfer

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



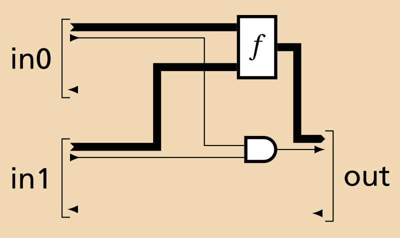
Datapath

Combinational function ignores flow control

Combinational Function Block

Strict/Unit Rate:

All input tokens required to produce an output



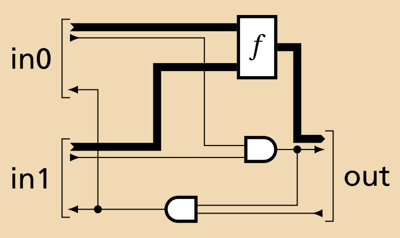
Valid network

Output valid if both inputs are valid

Combinational Function Block

Strict/Unit Rate:

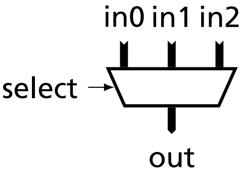
All input tokens required to produce an output

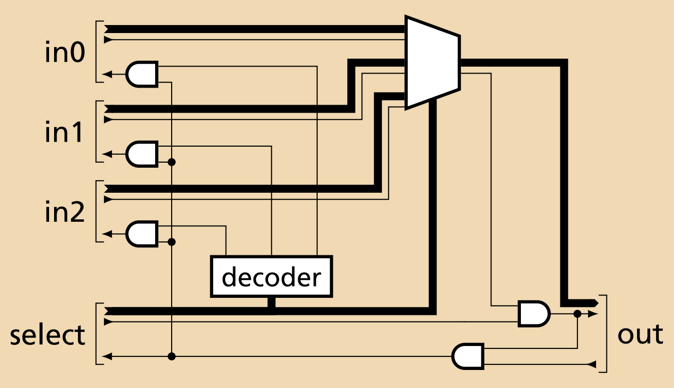


Ready network

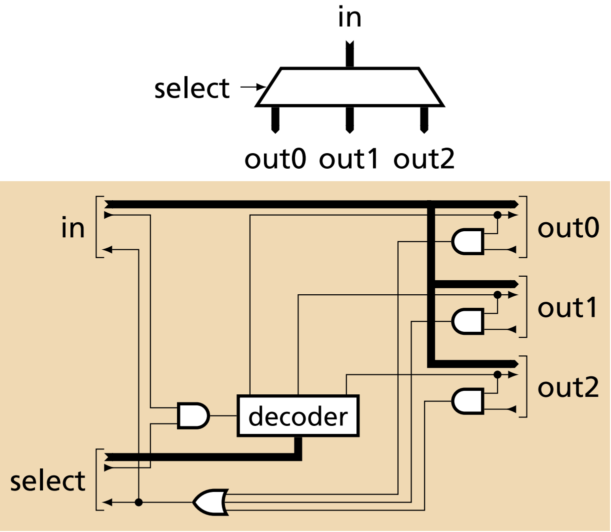
Input tokens consumed if output token is consumed
(output is valid and ready)

Multiplexer Block

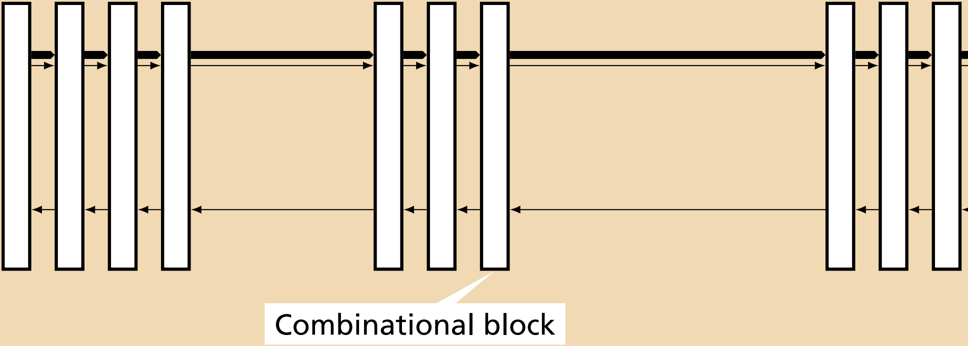




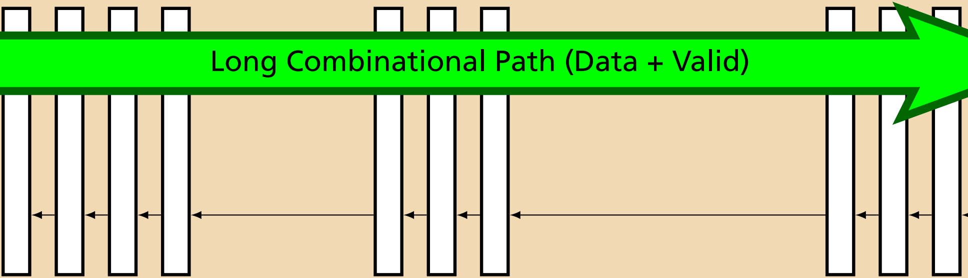
Demultiplexer Block



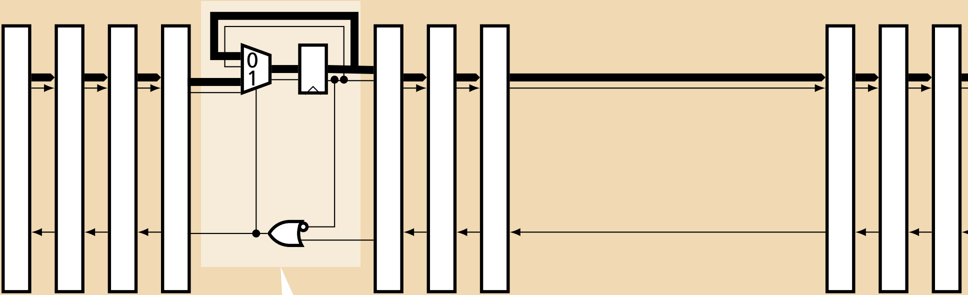
Buffering a Linear Pipeline



Buffering a Linear Pipeline

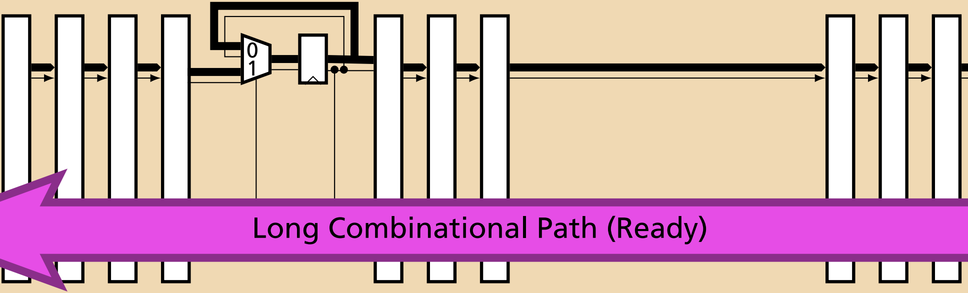


Buffering a Linear Pipeline

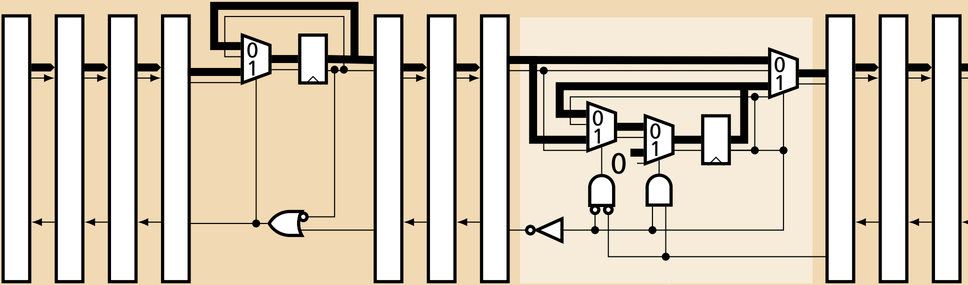


Data buffer:
Pipeline register
with valid, enable

Buffering a Linear Pipeline



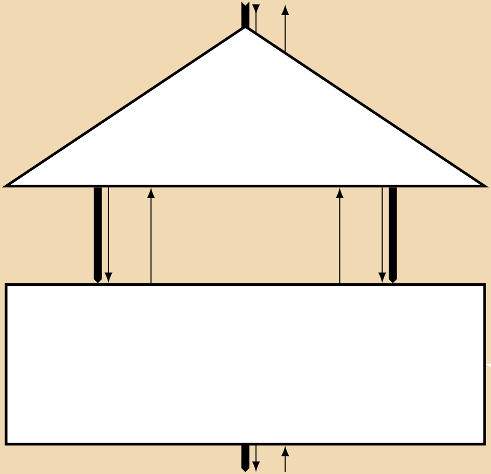
Buffering a Linear Pipeline



Control Buffer:
Register diverts token when downstream suddenly stops

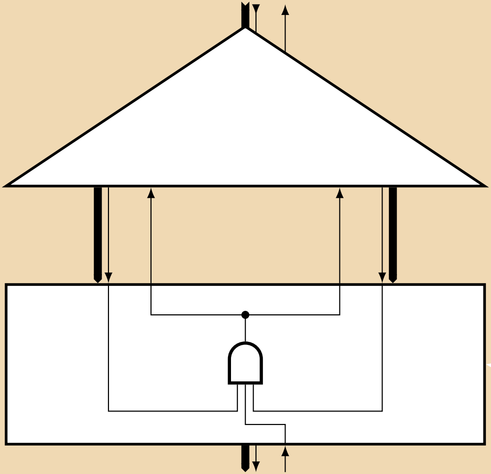
Cao et al. MEMOCODE 2015
Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)

The Problem with Fork



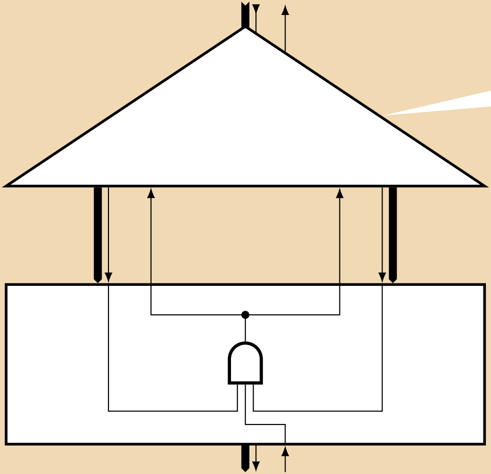
Combinational Block:
inputs ready when
both valid &
output ready

The Problem with Fork



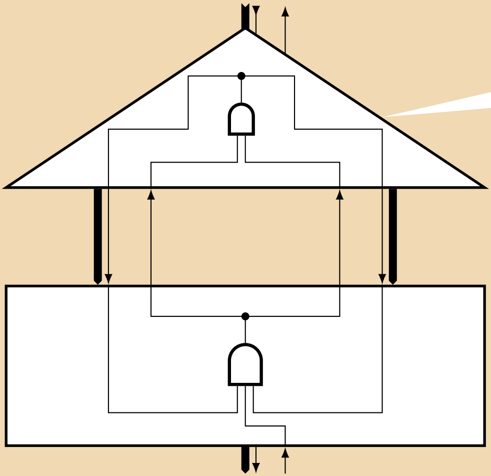
Combinational Block:
inputs ready when
both valid &
output ready

The Problem with Fork



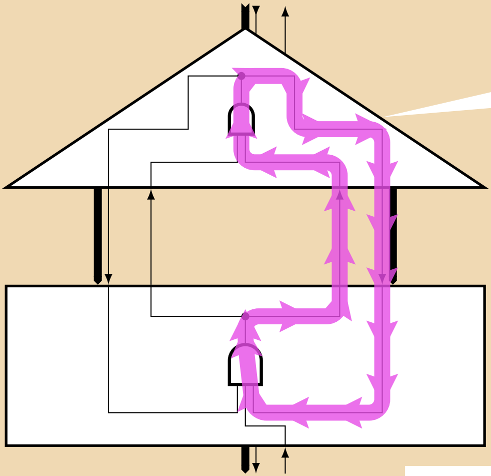
Fork:
outputs valid only
when all are ready

The Problem with Fork



Fork:
outputs valid only
when all are ready

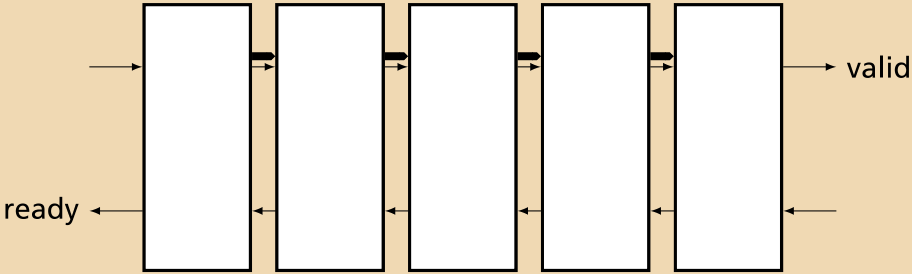
The Problem with Fork



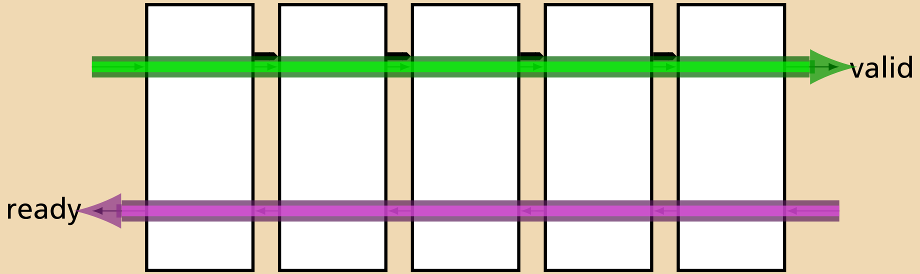
Fork:
outputs valid only
when all are ready

Oops: Combinational Cycle
This is *not* compositional

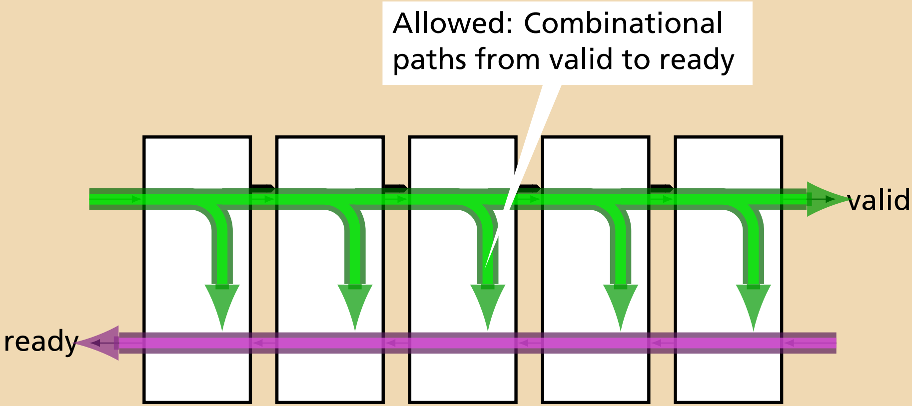
The Solution to Combinational Loops



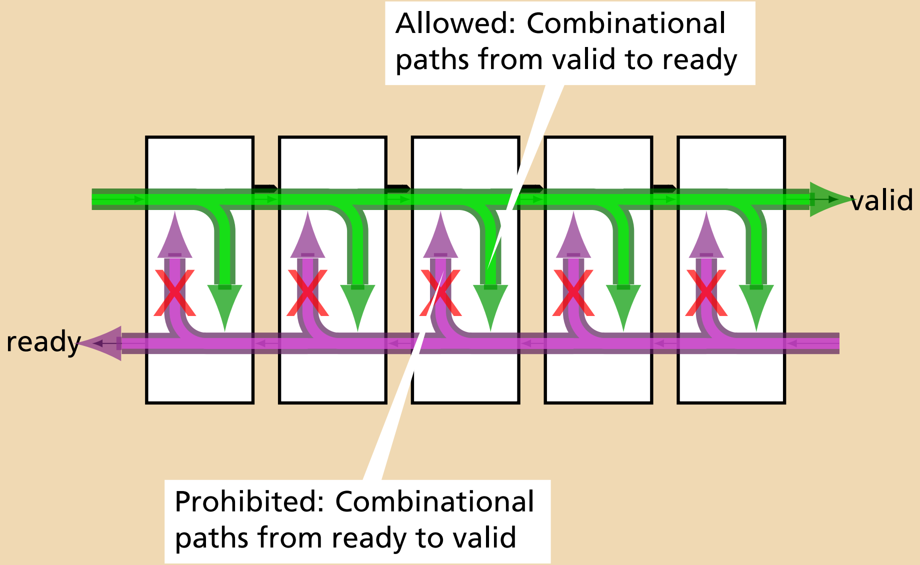
The Solution to Combinational Loops



The Solution to Combinational Loops

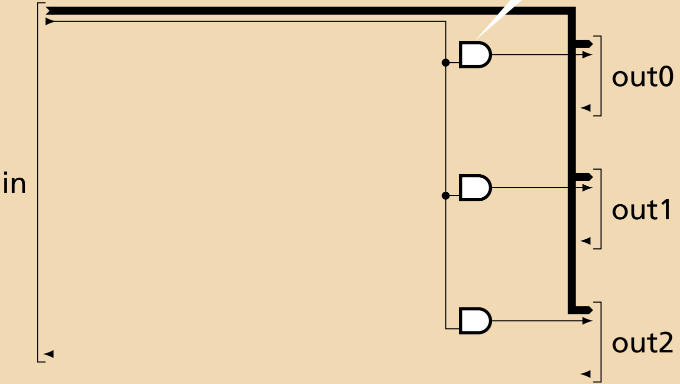


The Solution to Combinational Loops



The Solution to Fork: A Little State

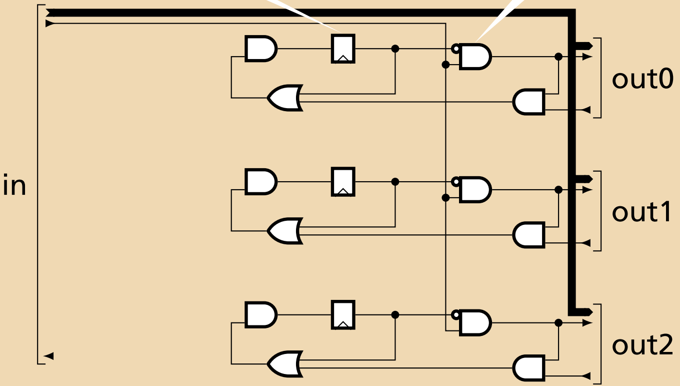
Valid out ignores ready of other outputs



The Solution to Fork: A Little State

Flip-flop set after token sent suppresses duplicates

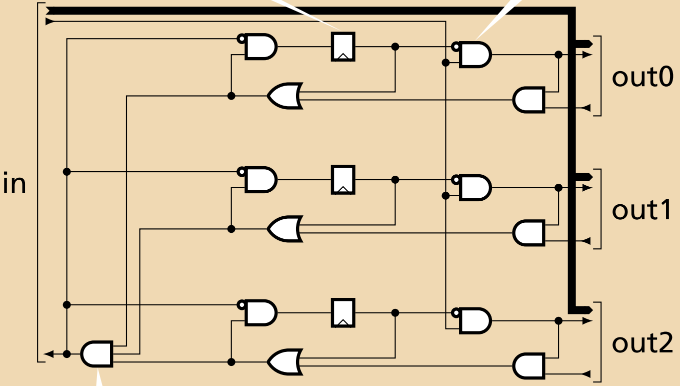
Valid out ignores ready of other outputs



The Solution to Fork: A Little State

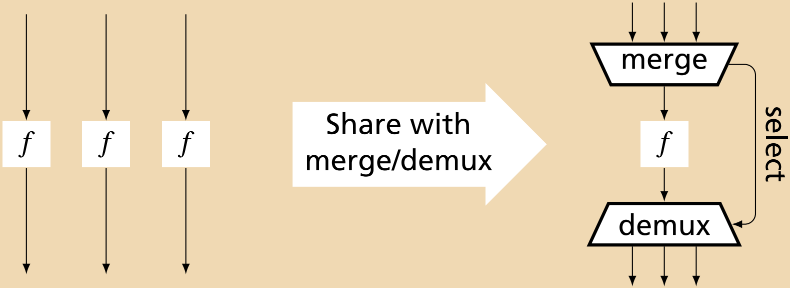
Flip-flop set after token sent suppresses duplicates

Valid out ignores ready of other outputs

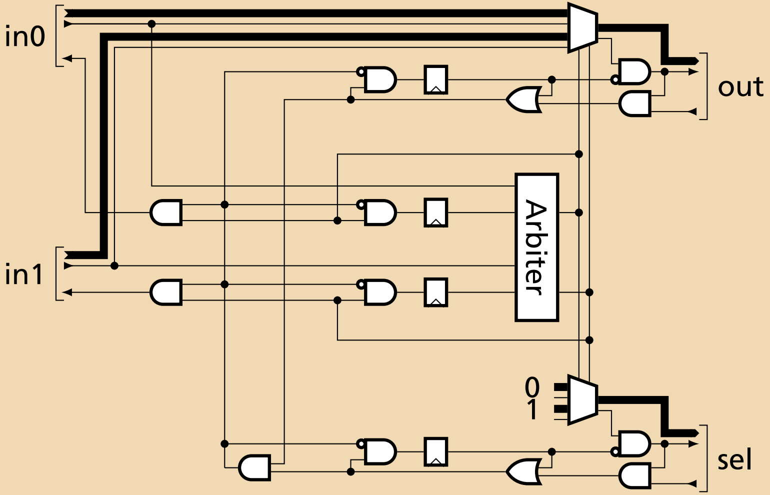


Input consumed once one token sent on every output

Nondeterministic Merge

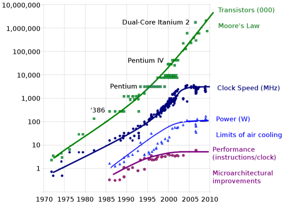


Two-Way Nondeterministic Merge Block w/ Select

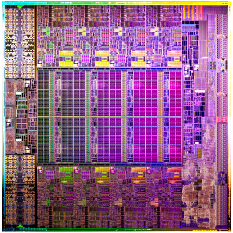


"Two-way fork with multiplexed output selected by an arbiter"

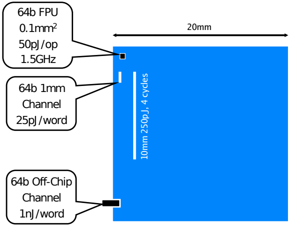
- Moore’s Law is alive and well



- 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

- Our project: A Pure Functional Language to FPGAs

- Removing recursion
- Functional to dataflow
- Dataflow to hardware

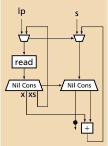
Add Explicit Memory Operations

```
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
```

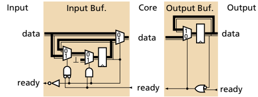
Functional to Dataflow

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