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
Where Is My Jetpack?

Popular Science, November 1969
Where Is My 10 GHz Processor?
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

Intel CPU Trends

Intel CPU Trends

Pollack’s Rule: Diminishing Returns for Processors

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

It takes $4 \times$ the transistors to give $2 \times$ the performance.

Intel CPU Trends

Intel Processors to Scale

- 4004 (1971)
- 8086 (1982)
- 8088 (1979)
- Pentium (1993)
- P II (1997)
- P III (1999)
- P IV (2000)
- Core 2 (2006)
What Happened in 2005?

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Cores</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>1</td>
<td>42 M</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>2</td>
<td>291 M</td>
</tr>
<tr>
<td>Xeon E5</td>
<td>2012</td>
<td>8</td>
<td>2.3 G</td>
</tr>
</tbody>
</table>
Intel CPU Trends

The Cray-2: Immersed in Fluorinert
Heat Flux in IBM Mainframes: A Familiar Trend

Liquid Cooled Apple Power Mac G5

2004  CMOS  1.2 kW
“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)

<table>
<thead>
<tr>
<th>Category</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>110 TFLOP/s</td>
</tr>
<tr>
<td>Frequency</td>
<td>1200 MHz</td>
</tr>
<tr>
<td>Power</td>
<td>250 W</td>
</tr>
<tr>
<td>Process</td>
<td>12 nm</td>
</tr>
<tr>
<td>Transistors</td>
<td>21.1 G</td>
</tr>
<tr>
<td>Area</td>
<td>815 mm²</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>5120</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Category</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>12 GB</td>
</tr>
<tr>
<td>Bus width</td>
<td>3072 bits</td>
</tr>
<tr>
<td>Clock</td>
<td>850 MHz</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>652.8 Gb/s</td>
</tr>
</tbody>
</table>

### Price

<table>
<thead>
<tr>
<th>Category</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>$3000</td>
</tr>
</tbody>
</table>
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
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

```c
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
Ghica et al.’s Geometry of Synthesis

Figure 1. In-place map schematic and implementation

Algol-like imperative language to handshake circuits

In this section we demonstrate how a circuit that performs communication over an I2C bus can be expressed using the Kiwi library. The motivation for tackling such an example arises from the fact that the typical coding style for such circuits involves hand coding state machines using nested case statements in VHDL (or equivalent features in Verilog). In particular, the sequencing of operations can be implemented as follows:

```csharp
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

Arvind, Hoe, et al.‘s Bluespec

**GCD Mod Rule**

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

**GCD Flip Rule**

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

*Figure 1.3*  Circuit for computing 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

```haskell
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 C$\lambda$aSH

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

More operational Haskell specifications of regular structures
Baaij, Kooijman, Kuper, Boeijink, and Gerards. C$\lambda$ash, DSD 2010
FHW: Functional Programs to Hardware
Deterministic Concurrency: A Fool’s Errand?

What Models of Computation Provide Deterministic Concurrency?

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchrony</td>
<td>The Columbia Esterel Compiler</td>
<td>2001–2006</td>
</tr>
<tr>
<td>Kahn Networks</td>
<td>The SHIM Model/Language</td>
<td>2006–2010</td>
</tr>
<tr>
<td>The Lambda Calculus</td>
<td>This Project</td>
<td>2010–</td>
</tr>
</tbody>
</table>
Our Project: Functional Programs to Hardware

\lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x)))
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:

\[
\text{type ::= Type} \\
\mid \text{Constr Type}^* \mid \cdots \mid \text{Constr Type}^*
\]

- 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:

\[
type ::= Type \quad \text{Primitive type}
\]
\[
| Constr\ Type^\ast \ | \cdots \ | Constr\ Type^\ast \quad \text{Tagged union}
\]

Examples:

\textbf{data} \text{ Intlist } = \text{ Nil } -- Linked list of integers
\quad | \text{ Cons } \text{ Int } \text{ Intlist }

\textbf{data} \text{ Bintree } = \text{ Leaf } \text{ Int } -- Binary tree of integers
\quad | \text{ Branch } \text{ Bintree } \text{ Bintree }

\textbf{data} \text{ Expr } = \text{ Literal } \text{ Int } -- Arithmetic expression
\quad | \text{ Var } \text{ String }
\quad | \text{ Binop } \text{ Expr } \text{ Op } \text{ Expr }

\textbf{data} \text{ Op } = \text{ Add } | \text{ Sub } | \text{ Mult } | \text{ Div }
Algebraic Datatypes in Hardware: Lists

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

![Diagram showing IntList structure with pointers and integers]
Datatypes in Hardware: Binary Trees

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

```plaintext
<table>
<thead>
<tr>
<th>32</th>
<th>17 16</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>pointer</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

Branch

Leaf
Example: Huffman Decoder in Haskell

```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)

<table>
<thead>
<tr>
<th>Leaf: 8-bit char</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch: 9-bit pointer</td>
<td>9-bit pointer</td>
</tr>
</tbody>
</table>

### Boolean input stream: (14 bits)

<table>
<thead>
<tr>
<th>Cons: 12-bit pointer</th>
<th>B</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil: 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Character output stream: (19 bits)

<table>
<thead>
<tr>
<th>Cons: 10-bit pointer</th>
<th>8-bit char</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil: 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing Recursion in Hardware

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

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

\[
\text{fib } n \quad = \quad \text{case } n \text{ of}
\]

\[
1 \quad \rightarrow \quad 1
\]

\[
2 \quad \rightarrow \quad 1
\]

\[
n \quad \rightarrow \quad \text{fib } (n-1) + \text{fib } (n-2)
\]
Transform to Continuation-Passing Style

\[
\text{fibk } n \ k = \ \text{case } n \ \text{of}
\]
\[
\begin{array}{ccl}
1 & \rightarrow & k \ 1 \\
2 & \rightarrow & k \ 1 \\
n & \rightarrow & \text{fibk } (n-1) (\lambda n1 \rightarrow \\
& & \text{fibk } (n-2) (\lambda n2 \rightarrow \\
& & k \ (n1 + n2)))
\end{array}
\]

\[
\text{fib } n = \ \text{fibk } n \ (\lambda x \rightarrow x)
\]
fibk \ n \ k \ = \ \textbf{case} \ n \ \textbf{of}
\begin{align*}
1 & \rightarrow k \ 1 \\
2 & \rightarrow k \ 1 \\
n & \rightarrow \ fibk \ (n-1) \ (k1 \ n \ k)
\end{align*}

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

```haskell
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
```
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*

Dataflow Node Library

- primitive
- fork
- case
- demux
- write
- read
- Memory operations
- "and" firing rules
- mux
- merge
- mergeChoice
- lock
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum \ } lp \ s = \\
\text{case \ read \ } lp \ \text{of} \\
\quad \text{Nil} \quad \rightarrow \ s \\
\quad \text{Cons} \ x \ xs \rightarrow \ \text{sum} \ xs \ (s + x)
\]
Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \\
\quad \text{case } \text{read } \text{lp} \text{ of } \\
\quad \quad \text{Nil} \quad \rightarrow \quad s \\
\quad \quad \text{Cons } x \text{ xs} \rightarrow \quad \text{sum } \text{xs} \ (s + x)
\]

Non-strict function: body starts evaluating \text{lp} before \text{s} is available
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \\
\text{case } \text{read } lp \text{ of} \\
\quad \text{Nil} \to s \\
\quad \text{Cons } x \ x s \to \text{sum } x s \ (s + x)
\]

Read: pointer → data
Write: data → pointer
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \\
\begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons } x \ xs & \rightarrow \text{sum } xs \ (s + x) 
\end{cases}
\]

Read: pointer → data
Write: data → pointer
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp\ s = \begin{cases} 
\text{Nil} & \rightarrow s \\
\text{Cons } x\ xs & \rightarrow \text{sum } xs \ (s + x)
\end{cases}
\]

Pattern matching with a decomposition mux
Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp\space s = \\
\text{case } \text{read } lp\space \text{of} \\
\quad \text{Nil} \quad \rightarrow \quad s \\
\quad \text{Cons } x\space xs \rightarrow \quad \text{sum } xs\ (s + x)
\]

Pattern matching with a decomposition mux
**Functional to Dataflow**

Sum a list using an accumulator and tail-recursion

\[
\text{sum } \text{lp } s = \\
\text{case } \text{read } \text{lp} \text{ of} \\
\quad \text{Nil } \rightarrow s \\
\quad \text{Cons } x \, \text{xs} \rightarrow \text{sum } \text{xs} \, (s + x)
\]

Tail recursion: physical loop
Sum a list using an accumulator and tail-recursion

\[
\text{\textbf{sum \ lp \ s =}} \\
\text{\hspace{1cm} \textbf{case \ read \ lp \ of}} \\
\text{\hspace{2cm} Nil \hspace{0.5cm} \rightarrow \hspace{0.5cm} s} \\
\text{\hspace{2cm} Cons \hspace{0.2cm} x \hspace{0.2cm} xs \rightarrow \textbf{sum} \hspace{0.2cm} xs \hspace{0.2cm} (s + x)}
\]

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

\[
\text{sum } lp \ s = \ \begin{cases} 
\text{Nil} & \rightarrow \ s \\
\text{Cons } x \ \text{xs} & \rightarrow \ \text{sum } \text{xs} (s + x)
\end{cases}
\]

Buffer sizes affect pipeline depth
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \begin{cases} 
\text{read } lp \rightarrow \ s \\
\text{Cons } x \ \text{xs} \rightarrow \ \text{sum } \text{xs} \ (s + x)
\end{cases}
\]

s arrives: can start computing sum
Sum a list using an accumulator and tail-recursion

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

![Diagram of the sum function](image-url)
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

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

[Diagram of the dataflow representation of the code]

[Diagram explanation: Read the list, process each element by summing the current accumulator with the new element, and recursively sum the rest of the list.]

Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[ \text{sum } lp \ s = \begin{cases} \text{Nil} & \to \ s \\ \text{Cons } x \ xs & \to \ \text{sum } xs \ (s + x) \end{cases} \]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum } lp \ s = \\
\quad \begin{cases} \\
\quad \text{Nil} & \rightarrow \ s \\
\quad \text{Cons } x \ xxs & \rightarrow \ \text{sum } xxs (s + x) \\
\end{cases}
\]
Functional to Dataflow

Sum a list using an accumulator and tail-recursion

\[
\text{sum \ } lp \ s = \begin{cases} 
\text{Nil} & \rightarrow \ s \\
\text{Cons} \ x \ xs & \rightarrow \ \text{sum} \ xs \ (s + x)
\end{cases}
\]
Non-Strict Functions Run Faster

- Speedup
  - Strict, Finite Buffers
  - Non-Strict, Finite Buffers
  - Non-Strict, Infinite Buffers

Bar chart showing the speedup for different functions: MergeSort, TreeMap, DFS, Filter, Append, Map.
Non-Strict Functions Run Faster

- MergeSort
- TreeMap
- DFS
- Filter
- Append
- Map

Comparison between:
- Non-Strict, Finite Buffers
- Strict, Finite Buffers

Chart showing speedup for different functions under various conditions.
Non-Strict Functions Run Faster

<table>
<thead>
<tr>
<th>Function</th>
<th>Non-Strict, Infinite Buffers</th>
<th>Non-Strict, Finite Buffers</th>
<th>Strict, Finite Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeSort</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TreeMap</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DFS</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Filter</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Append</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Map</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
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

Patience Through Handshaking

Want *patient* blocks to handle delays from

- Memory systems
- Data-dependent computations

versus

- Full buffers
- Shared resources
- Busy computational units
Patience Through Handshaking

Want *patient* blocks to handle delays from

<table>
<thead>
<tr>
<th>valid</th>
<th>ready</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Token transferred</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Token valid; held</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>No token to transfer</td>
</tr>
</tbody>
</table>

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

The diagram illustrates a multiplexer block with inputs in0, in1, and in2, and a select line. The block outputs to an AND gate, with the decoder selecting which input is passed through to the output (out).
Buffering a Linear Pipeline

Combinational block

Data buffer: Pipeline register with valid, enable

Control Buffer: Register diverts token when downstream suddenly stops

Long Combinational Path (Data + Valid)

Long Combinational Path (Ready)

Cao et al. MEMOCODE 2015

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

Long Combinational Path (Data + Valid)

Cao et al. MEMOCODE 2015

Inspired by Carloni's Latency Insensitive Design (e.g., MEMOCODE 2007)
Data buffer: Pipeline register with valid, enable
Buffering a Linear Pipeline

Combinational block
Data buffer: Pipeline register with valid, enable

Control Buffer: Register diverts token when downstream suddenly stops

Long Combinational Path (Data + Valid)
Long Combinational Path (Ready)

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

Fork: outputs valid only when all are ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

Combinational Block: inputs ready when both valid & output ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

Combinational Block:
inputs ready when both valid & output ready

Fork:
outputs valid only when all are ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

Combinational Block:
inputs ready when both valid & output ready

Fork:
outputs valid only when all are ready

Oops: Combinational Cycle
This is not compositional
The Problem with Fork

Combinational Block:
inputs ready when both valid & output ready

Fork:
outputs valid only when all are ready

Oops: Combinational Cycle
This is *not* compositional
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
The Solution to Combinational Loops

- **Allowed**: Combinational paths from valid to ready
- **Prohibited**: Combinational paths from ready to valid
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
The Solution to Combinational Loops

Allowed: Combinational paths from valid to ready

Prohibited: Combinational paths from ready to valid
Valid out ignores ready of other outputs

The Solution to Fork: A Little State

Valid out ignores ready of other outputs

Flip-flop set after token sent suppresses duplicates

Input consumed once one token sent on every output
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
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

merge

demux

select
Two-Way Nondeterministic Merge Block w/ Select

“Two-way fork with multiplexed output selected by an arbiter”
\[
gcd(a, b) = 
\begin{cases} 
a & \text{if } a = b \\
gcd(a, b - a) & \text{else if } a < b \\
gcd(a - b, b) & \text{else} 
\end{cases}
\]
Best Buffering for GCD (Manually Obtained)

Each loop has one of each buffer

- **Data Buffer**
- **Control Buffer**
Synthesizing Parallel Memory Systems

Duplicate the recursive task
Assign separate cache partitions to parallel tasks
Works best with balanced workload

Divide-and-Conquer Functions: Inherently Parallel

\[ \text{map} :: \text{Tree} \rightarrow \text{Tree} \]
\[ \text{map} \ t = \text{case} \ t \ \text{of} \]
\[ \quad \text{Leaf} \rightarrow t \]
\[ \quad \text{Node} \ l \ x \ r \rightarrow \text{Node} \ (\text{map} \ l) \ (f \ x) \ (\text{map} \ r) \]
Transformations to Enable Parallelism

map \ t = \ldots

\text{map}

\text{Shared Cache}

f
Transformations to Enable Parallelism

map \( t = \ldots \)

\[
\text{map}_C :: \text{Tree} \rightarrow \text{Tree}
\]

\[
\text{map}_C \ t = \text{case } t \text{ of}
\]

\[
\quad \text{Leaf} \rightarrow t
\]

\[
\quad \text{Node} \ l \ x \ r \rightarrow \text{Node} \ (\text{map}_C \ l) \ (f_C \ x) \ (\text{map}_C \ r)
\]
Transformations to Enable Parallelism

\[ \text{map } t = \ldots \]
\[ \text{map}_C :: \text{Tree} \rightarrow \text{Tree} \]
\[ \text{map}_C t = \ldots \]
\[ \text{map}_S t = \text{case } t \text{ of} \]
\[ \quad \text{Leaf} \rightarrow t \]
\[ \quad \text{Node } l \times x \times r \rightarrow \text{Node } (\text{map } l) (f \times x) (\text{map}_C r) \]
Memory Partitioning to Exploit the Parallelism

\[
\begin{align*}
\text{map } t &= \ldots \\
\text{map}_C :: \text{Tree}_C &\rightarrow \text{Tree}_C \\
\text{map}_C \ tp &= \ldots \\
\text{map}_S \ tp &= \text{case } t \text{ of} \\
&\quad \text{Leaf} \rightarrow t \\
&\quad \text{Node} \ l \ x \ r \rightarrow \text{Node} \ (\text{map} \ l) \ (f \ x) \ (\text{TfromC} \ (\text{map}_C \ (\text{TtoC} \ r)))
\end{align*}
\]

Diagram:
- Stack
- Stack
- Heap
- Heap
- f
- f
- TtoC
- TtoC
- TfromC
- TfromC
- Heap_C
- Stack_C
Doing This Increases Performance

Relative Performance
(decrease in cycles) Transformed+Partitioned

Treesort | RBsort | RBmap | Filtering | Mergesort

0.5
1
1.5
2
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
Implement algebraic data types as bit vectors with tags and pointers

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

Implement dataflow in hardware with compositional blocks with handshaking

Memory Partitioning to Exploit the Parallelism

Duplicate tasks and partition caches to speed recursive algorithms