Functioning Hardware from Functional Specifications

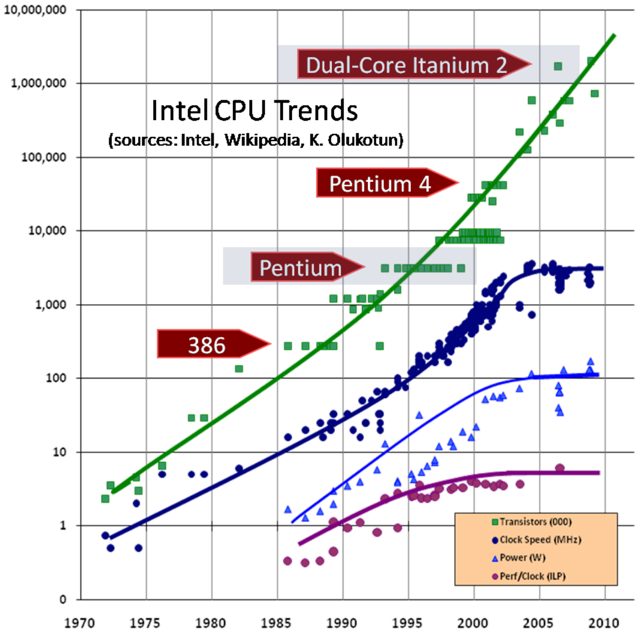
Stephen A. Edwards

Columbia University

SYNCHRON, Dagstuhl, Germany, November 2013

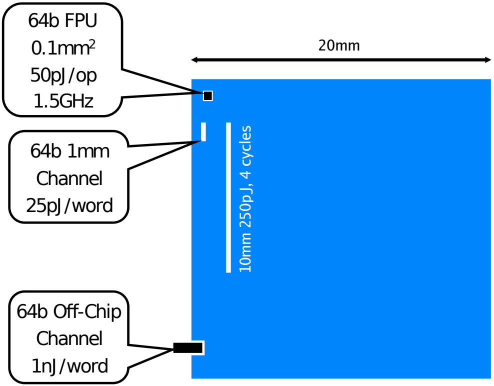
$$(\lambda x.?) f = \text{FPGA}$$

Where's my 10 GHz processor?



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

Dally: Calculation is Cheap; Communication is Costly



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

Performance = Parallelism

Efficiency = Locality

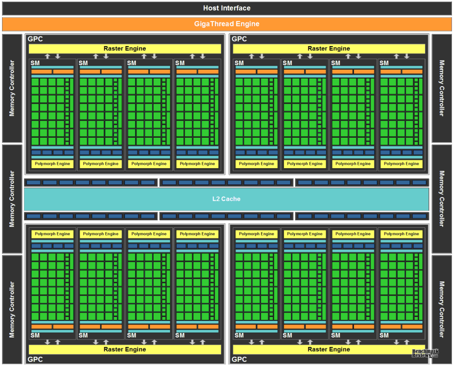
Bill Dally's 2009 DAC Keynote, *The End of Denial Architecture*

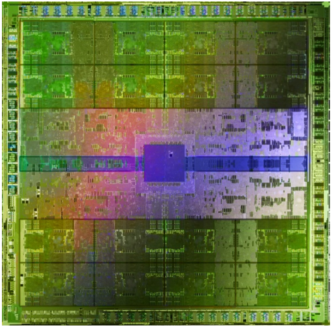
Parallelism for Performance and Locality for Efficiency

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

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

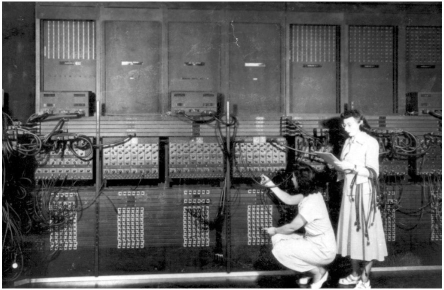
Massive On-Chip Parallelism is Here

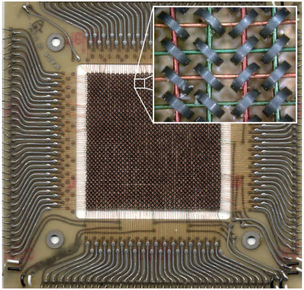




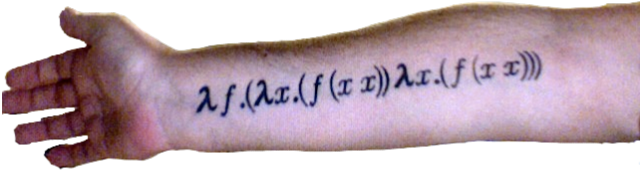
NVIDIA GeForce GTX-400/GF100/Fermi:
3 billion transistors, 512 CUDA cores, 16 geometry units, 64 texture units, 48 render output units, 384-bit GDDR5

The Future is Wires and Memory





Functional Programs to FPGAs

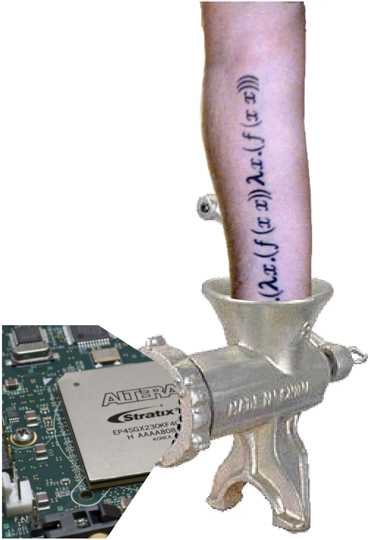


Functional Programs to FPGAs

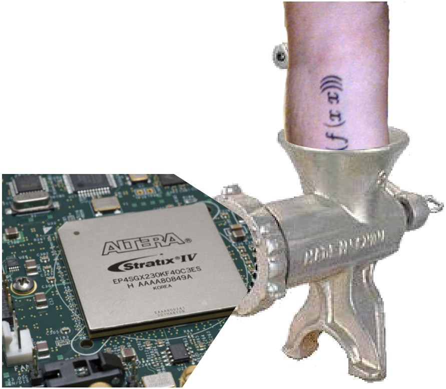
Functional Programs to FPGAs

Functional Programs to FPGAs

Functional Programs to FPGAs

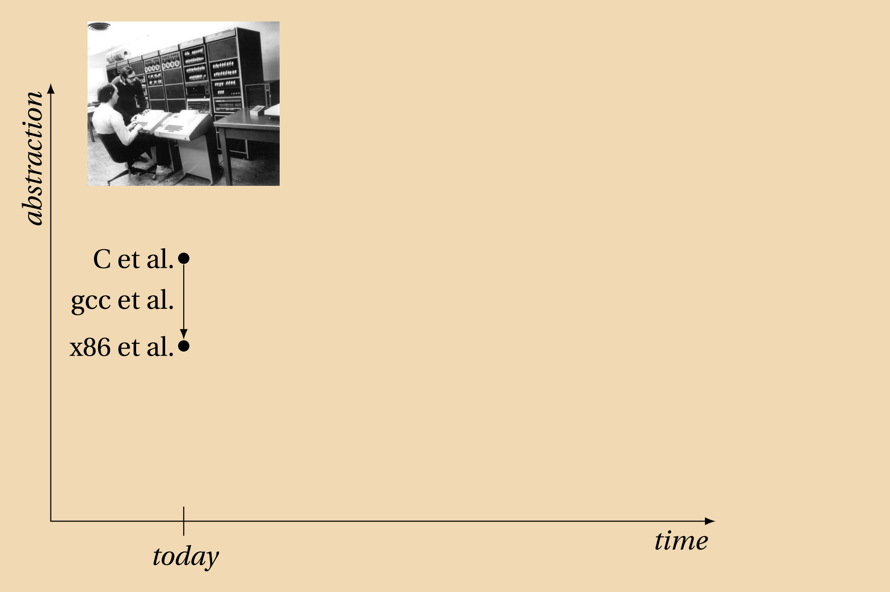


Functional Programs to FPGAs

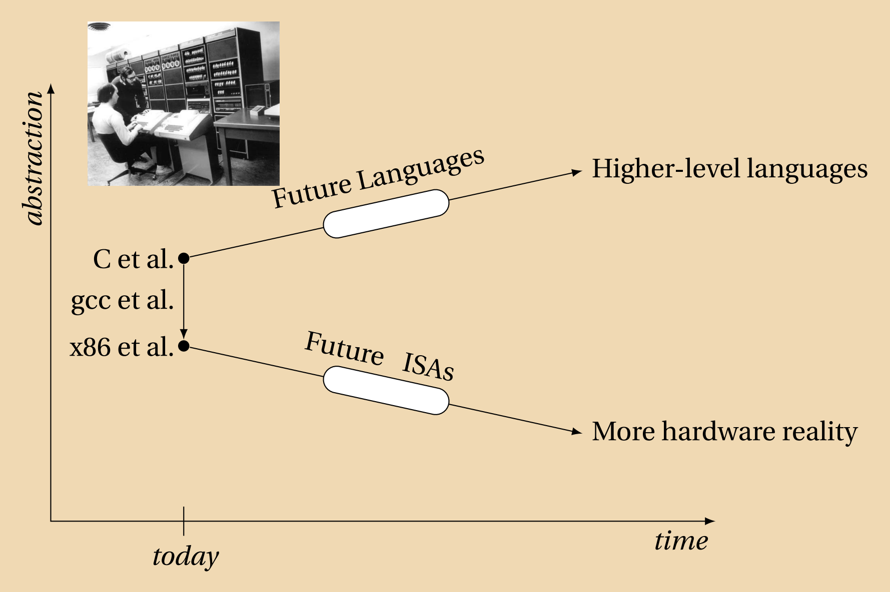


Functional Programs to FPGAs

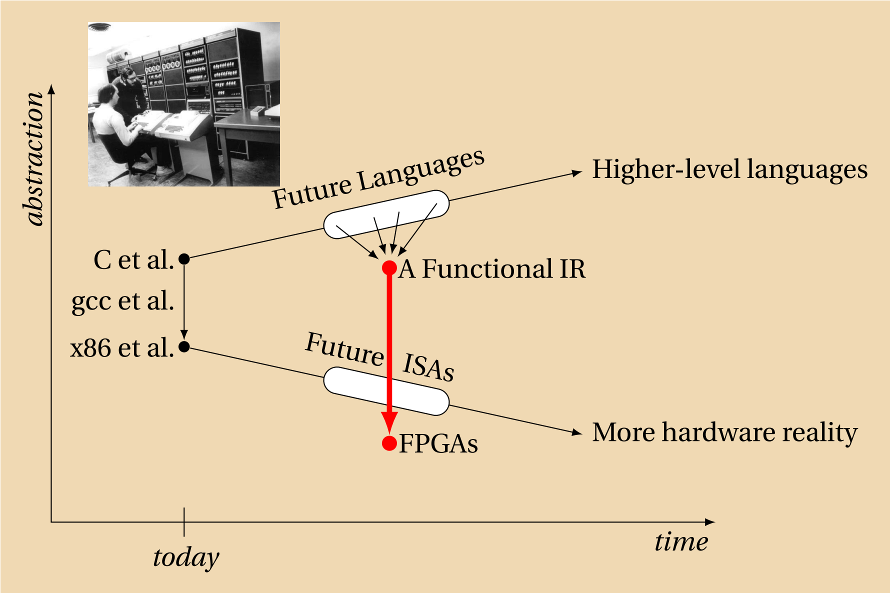
A Little More Detail



A Little More Detail

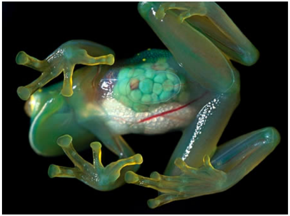


A Little More Detail



Why Functional Specifications?

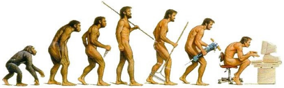
- Referential transparency/side-effect freedom make formal reasoning about programs vastly easier



- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.

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

Why FPGAs?



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

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

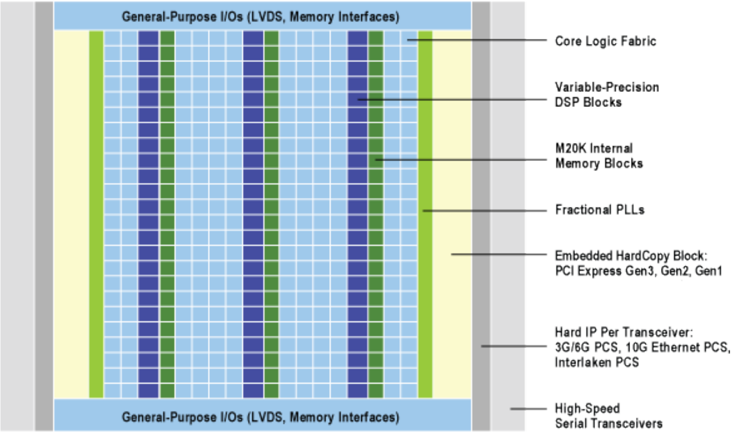
Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

A Recent High-End FPGA: Altera's Stratix V

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

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

300000 6-input LUTs; 28 nm feature size



The Practical Question

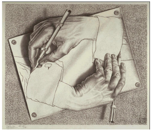
How do we synthesize hardware from pure functional languages for FPGAs?

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

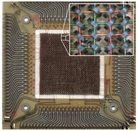
To Implement Real Algorithms in Hardware, We Need

- Structured, recursive data types

- Recursion to handle recursive data types



- Memories



- Memory Hierarchy

The Type System: Algebraic Data Types

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

$\mathit{type} ::= \mathit{Type}$		Named type/primitive
	$\mid$ $\mathit{Constr}\ \mathit{Type}^* \mid \cdots \mid \mathit{Constr}\ \mathit{Type}^*$	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*		Named type/primitive
	$\mid$ *Constr Type** $\mid \cdots \mid$ *Constr Type**	Tagged union

Examples:

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

```
data Bintree = Leaf  Int                  -- Binary tree w/ integer leaves
             | Branch Bintree Bintree
```

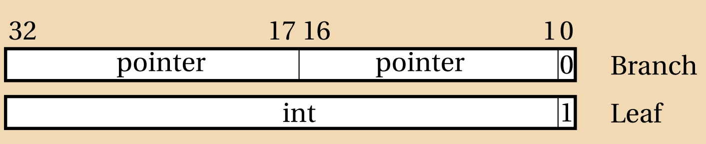
```
data Expr = Literal Int                   -- Arithmetic expression
          | Var String
          | Binop Expr Op Expr

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

Algebraic Datatypes in Hardware: Lists

$$
\begin{aligned}
\textbf{data}\ \text{IntList}\ &=\ \text{Cons}\ \textbf{Int}\ \text{IntList} \\
&\mid\ \text{Nil}
\end{aligned}
$$

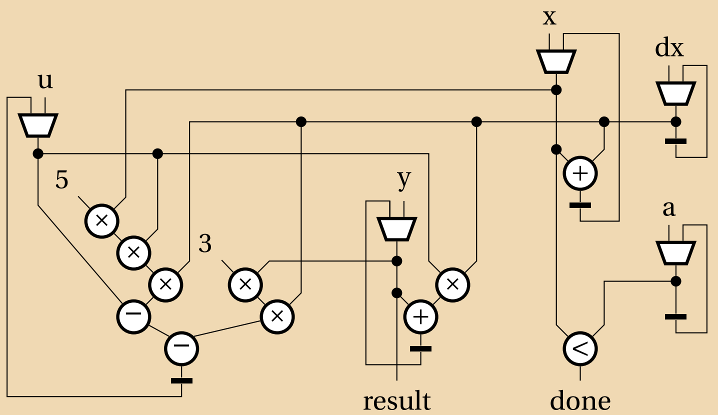
Datatypes in Hardware: Binary Trees

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

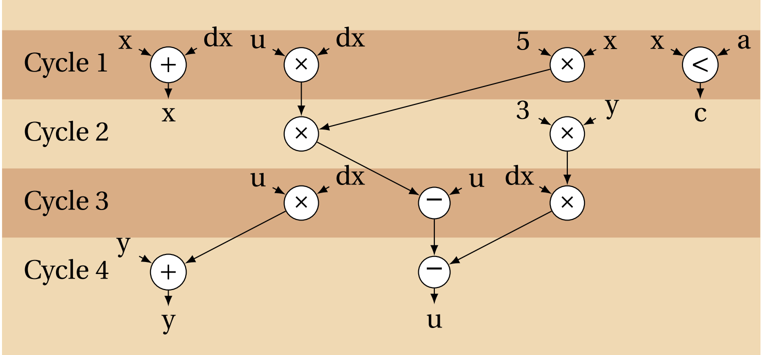


High-Level Synthesis in a Functional Setting

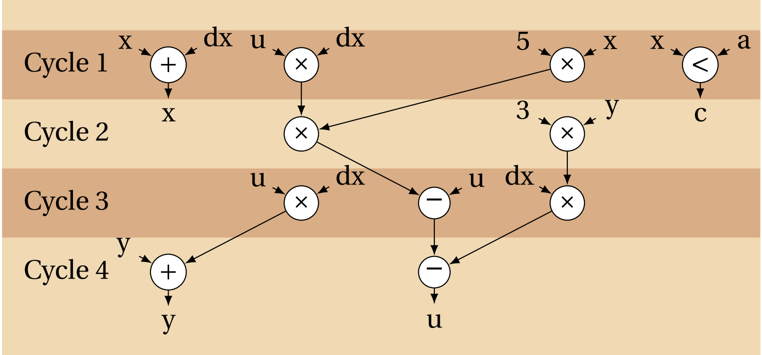
```
diffeq a dx x u y =
  if x < a then
    diffeq a dx (x + dx) (u − 5∗x∗u∗dx − 3∗y∗dx) (y + u∗dx)
  else y
```



Scheduling

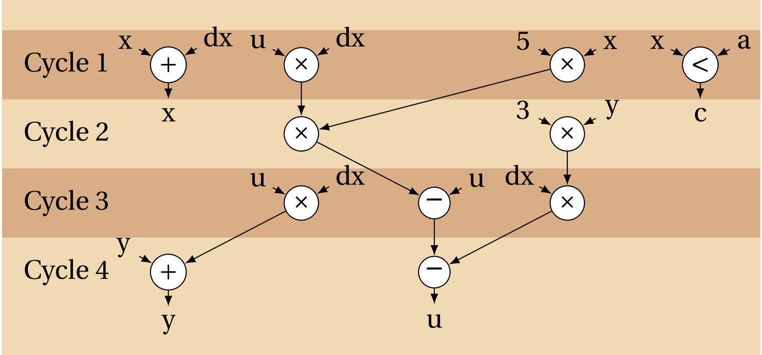


Scheduling




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

Scheduling




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

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

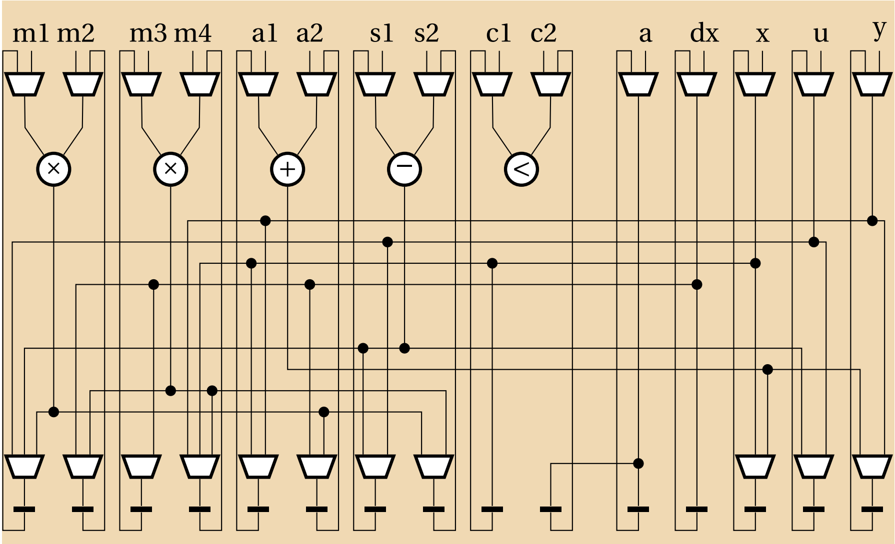
In the last code block, "if", "not", "then" and "else" are printed in bold.

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

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

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

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

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

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

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

In Hardware



Removing Recursion: The Fib Example

$$
\begin{aligned}
\text{fib } n \quad &= \textbf{case } n \textbf{ of} \\
&\quad 1 \quad \to 1 \\
&\quad 2 \quad \to 1 \\
&\quad n \quad \to \text{fib } (n-1) + \text{fib } (n-2)
\end{aligned}
$$

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)
```

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
```

Representing Continuations with a Type

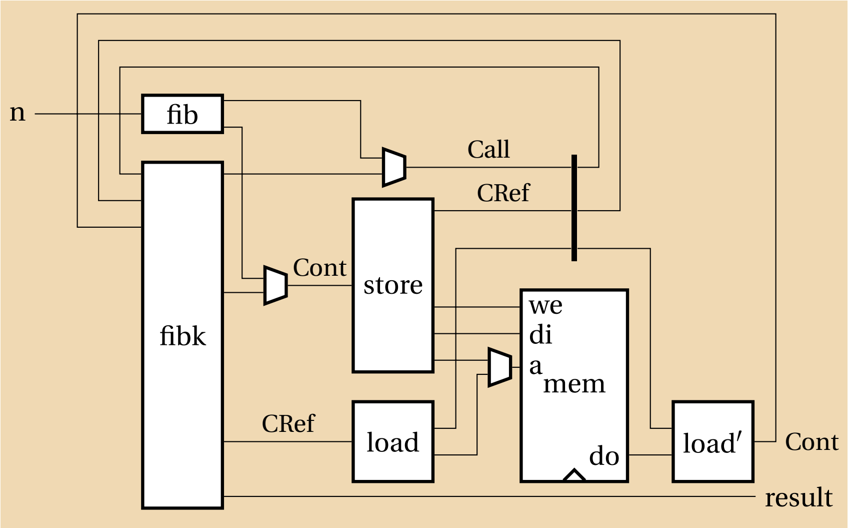
```
data Cont = K0 | K1 Int Cont | K2 Int Cont

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
```

Merging 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)
```

Adding Explicit Memory Operations

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

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

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


n
fib
fibk
Call
CRef
store
Cont
we
di
a
mem
do
CRef
load
load′
Cont
result

Duplication for Performance

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

Duplication for Performance

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

After duplicating functions:

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

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

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

Here, *fib'* and *fib"* may run in parallel.

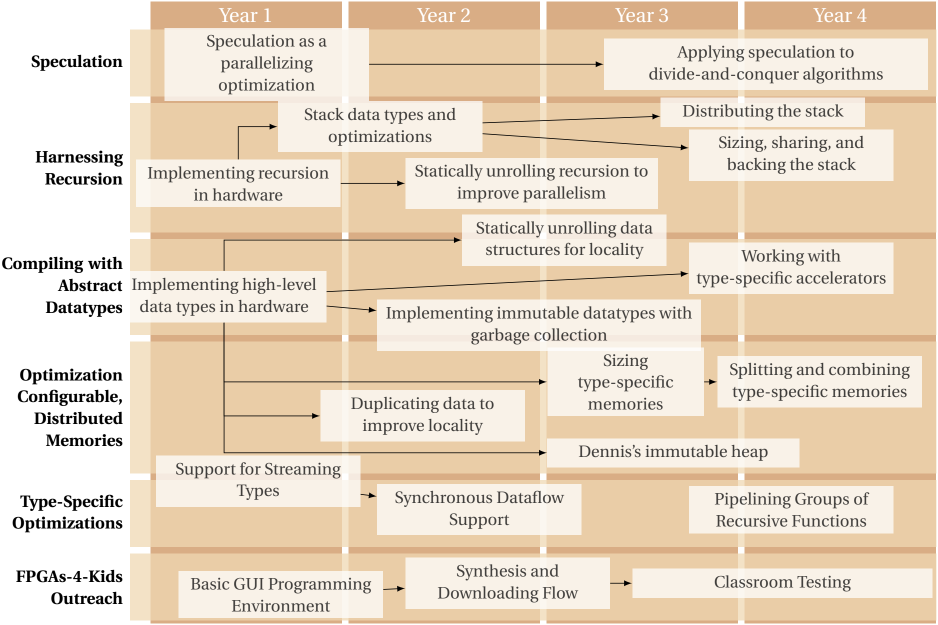
Unrolling Recursive Data Structures

Original Huffman tree type:

$\textbf{data}\ \text{Htree} = \text{Branch Htree HTree} \mid \text{Leaf}\ \textbf{Char}$

Unrolled Huffman tree type:

$\textbf{data}\ \text{Htree} = \text{Branch Htree}'\ \text{HTree}' \mid \text{Leaf}\ \textbf{Char}$
$\textbf{data}\ \text{Htree}' = \text{Branch}'\ \text{Htree}''\ \text{HTree}'' \mid \text{Leaf}'\ \textbf{Char}$
$\textbf{data}\ \text{Htree}'' = \text{Branch}''\ \text{Htree HTree} \mid \text{Leaf}''\ \textbf{Char}$



Year 1
Year 2
Year 3
Year 4
Speculation
Speculation as a parallelizing optimization
Applying speculation to divide-and-conquer algorithms
Harnessing Recursion
Stack data types and optimizations
Distributing the stack
Sizing, sharing, and backing the stack
Implementing recursion in hardware
Statically unrolling recursion to improve parallelism
Compiling with Abstract Datatypes
Statically unrolling data structures for locality
Working with type-specific accelerators
Implementing high-level data types in hardware
Implementing immutable datatypes with garbage collection
Optimization Configurable, Distributed Memories
Sizing type-specific memories
Splitting and combining type-specific memories
Duplicating data to improve locality
Dennis's immutable heap
Type-Specific Optimizations
Support for Streaming Types
Synchronous Dataflow Support
Pipelining Groups of Recursive Functions
FPGAs-4-Kids Outreach
Basic GUI Programming Environment
Synthesis and Downloading Flow
Classroom Testing