

Compiling Parallel Algorithms to Memory Systems

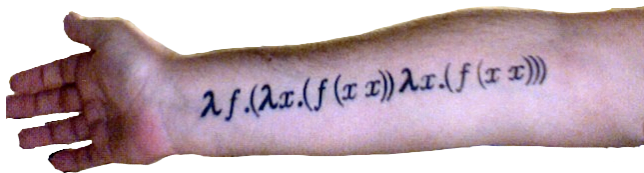
Stephen A. Edwards

Columbia University

ESLsyn Keynote, June 2nd, 2012

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

Functional Programs to FPGAs



Functional Programs to FPGAs



Functional Programs to FPGAs



Functional Programs to FPGAs



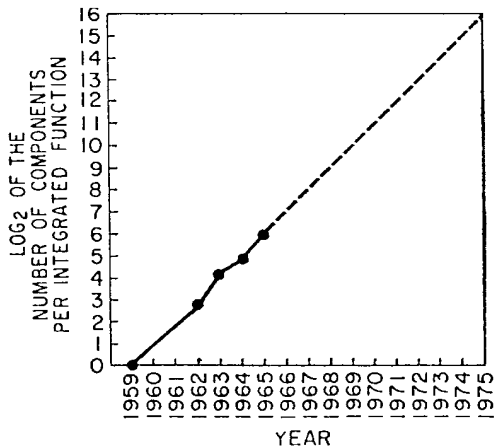
Functional Programs to FPGAs



Functional Programs to FPGAs



Moore's Law: Lots of Cheap Transistors...

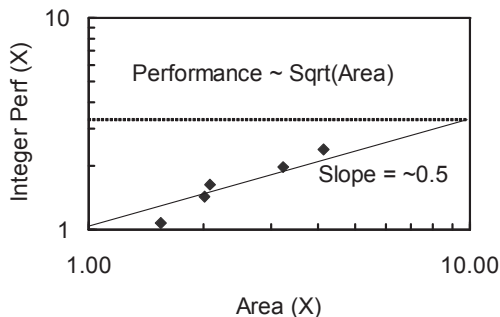


“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.”

Closer to every 24 months

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

Pollack's Rule: ...Give Diminishing Returns for Processors

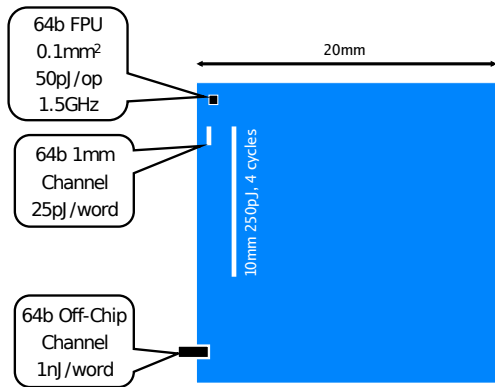


Single-core processor performance follows the **square root** of area.

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

Fred J. Pollack, MICRO 1999 keynote. Graph from Borkar, DAC 2007

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.

Bacon et al.'s Liquid Metal

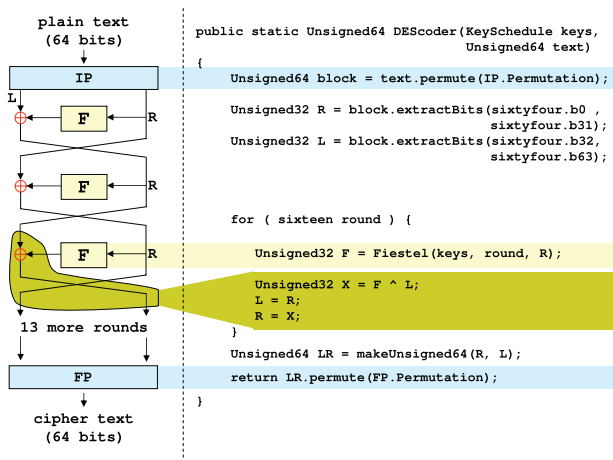


Fig. 2. Block level diagram of DES and Lime code snippet

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

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

Goldstein et al.'s Phoenix

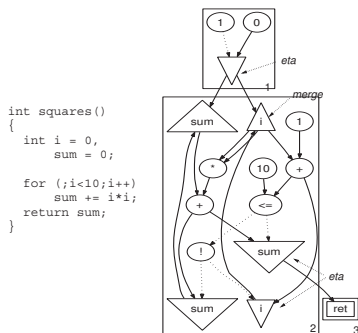


Figure 3: C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)

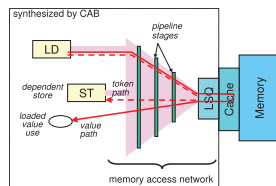


Figure 8: Memory access network and implementation of the value and token forwarding network. The LOAD produces a data value consumed by the oval node. The STORE node may depend on the load (i.e., we have a token edge between the LOAD and the STORE, shown as a dashed line). The token travels to the root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

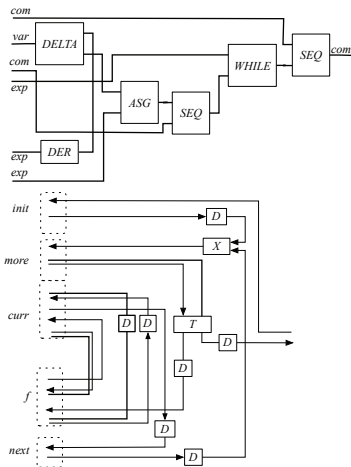


Figure 1. In-place map schematic and implementation

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

Arvind, Hoe et al.'s Bluespec

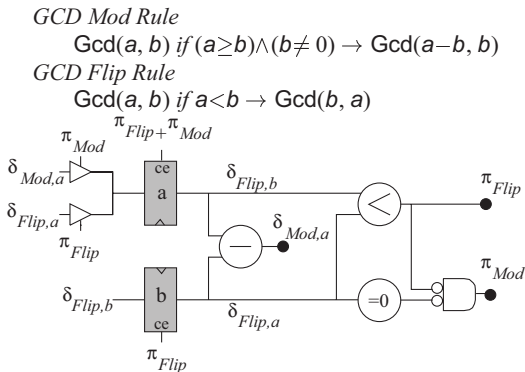


Figure 1.3 Circuit for computing $\text{Gcd}(a, b)$ from Example 1.

Guarded commands and functions to synchronous logic

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
```

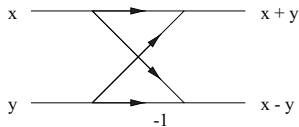


Figure 9: A butterfly

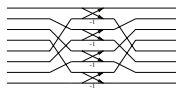


Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures

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

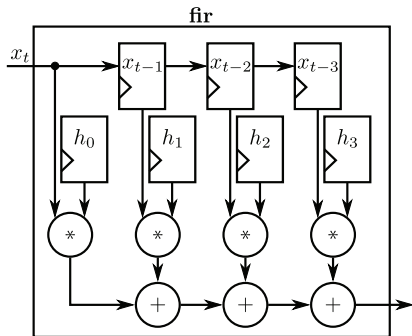


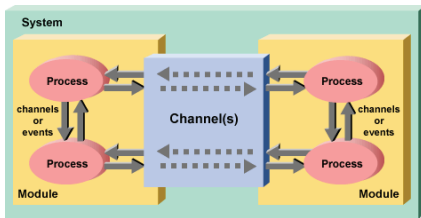
Fig. 6. 4-taps FIR Filter

More operational Haskell specifications of regular structures

AutoESL (Xilinx, was Cong's xPilot)

◆ SSDM (System-level Synthesis Data Model)

- Hierarchical netlist of concurrent processes and communication channels

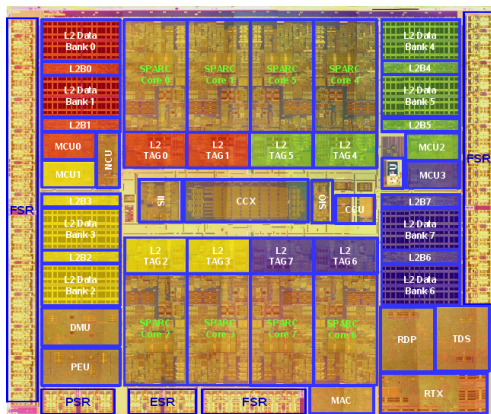


- Each leaf process contains a sequential program which is represented by an extended LLVM IR with hardware-specific semantics
 - Port / IO interfaces, bit-vector manipulations, cycle-level notations

SystemC input; classical high-level synthesis for processes

Jason Cong, presentation at ISARS 2005

Optimization of Parallel “Programs” Enables Chip Design



Sun's UltraSPARC T2

The “Niagara 2”

8 cores; 64 threads

Built 2007, 1.6 GHz, 65 nm

Released open-source as
the OpenSPARC T2

www.opensparc.net

454 000 lines of synthesizable Verilog → 503 000 000 transistors

A mix of Boolean logic and structure

The Lesson of Logic Synthesis: the Enabling Technology

How do you compile and optimize a digital logic circuit?

$$f_1 = abcd + abce + \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}c + cdf + abc\overline{d}\overline{e} + \overline{a}\overline{b}\overline{c}\overline{d}\overline{f}$$
$$f_2 = bdg + \overline{b}d\overline{f}g + \overline{b}\overline{d}g + b\overline{d}eg$$

$$f_1 = c(x + \overline{a}) + a\overline{c}\overline{x}$$

$$f_2 = gx$$

$$x = d(b + f) + \overline{d}(\overline{b} + e)$$

The Lesson of Logic Synthesis: the Enabling Technology

How do you compile and optimize a digital logic circuit?

Use a simple, formal model and automate it.

$$\begin{aligned}f_1 &= abcd + abce + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}\bar{d} + \bar{a}c + cdf + abc\bar{d}\bar{e} + a\bar{b}\bar{c}\bar{d}\bar{f} \\f_2 &= bdg + \bar{b}d\bar{f}g + \bar{b}\bar{d}g + b\bar{d}eg\end{aligned}$$

Minimize

$$\begin{aligned}f_1 &= bcd + bce + \bar{b}\bar{d} + \bar{a}c + cdf + abc\bar{d}\bar{e} + a\bar{b}\bar{c}\bar{d}\bar{f} \\f_2 &= bdg + d\bar{f}g + \bar{b}\bar{d}g + \bar{d}eg\end{aligned}$$

Factor

$$\begin{aligned}f_1 &= c(b(d + e) + \bar{b}(\bar{d} + f) + \bar{a}) + a\bar{c}(b\bar{d}\bar{e} + \bar{b}\bar{d}\bar{f}) \\f_2 &= g(d(b + f) + \bar{d}(\bar{b} + e))\end{aligned}$$

Decompose

$$\begin{aligned}f_1 &= c(x + \bar{a}) + a\bar{c}\bar{x} \\f_2 &= gx \\x &= d(b + f) + \bar{d}(\bar{b} + e)\end{aligned}$$

High-Level Synthesis: Adding Time Meant Scheduling

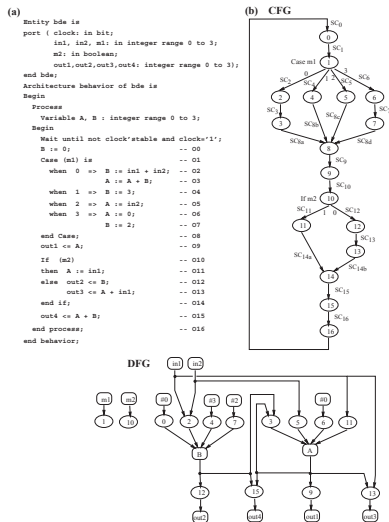


Figure 2: (a) VHDL description; (b) Separate control and data-flow graphs

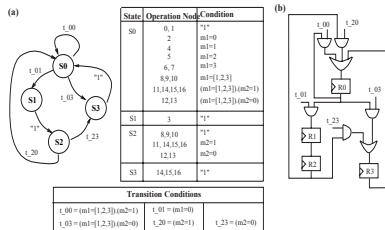


Figure 3: (a) FSM for scheduled CFG in Figure 2(b), (b) Hardware implementation of FSM using one-hot encoding

Bergamaschi, *Behavioral Network Graph*, DAC 1999.

The High-Level Synthesis Lessons

Don't Start From C

“The so-called high-level specifications in reality grew out of the need for simulation and were often little more than an input language to make a discrete event simulator reproduce a specific behavior.”

Gupta and Brewer, *High-Level Synthesis: A Retrospective*, 2008.

Don't Forget Memory

Goldstein et al.'s Phoenix synthesized asynchronous hardware from ANSI C. Required heroic work [CGO 2003] to recover any parallelism.

Our Approach



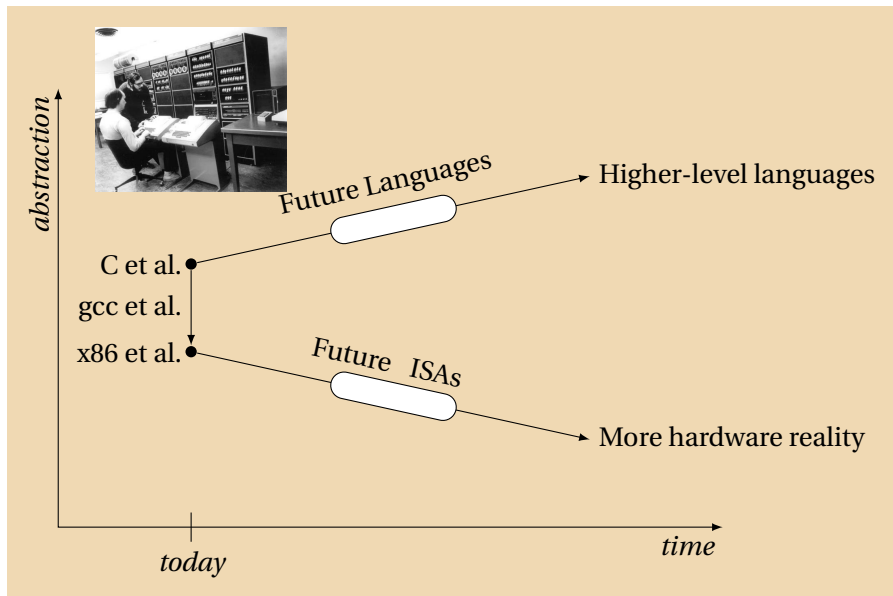
abstraction

C et al. ●
gcc et al. ↓
x86 et al. ●

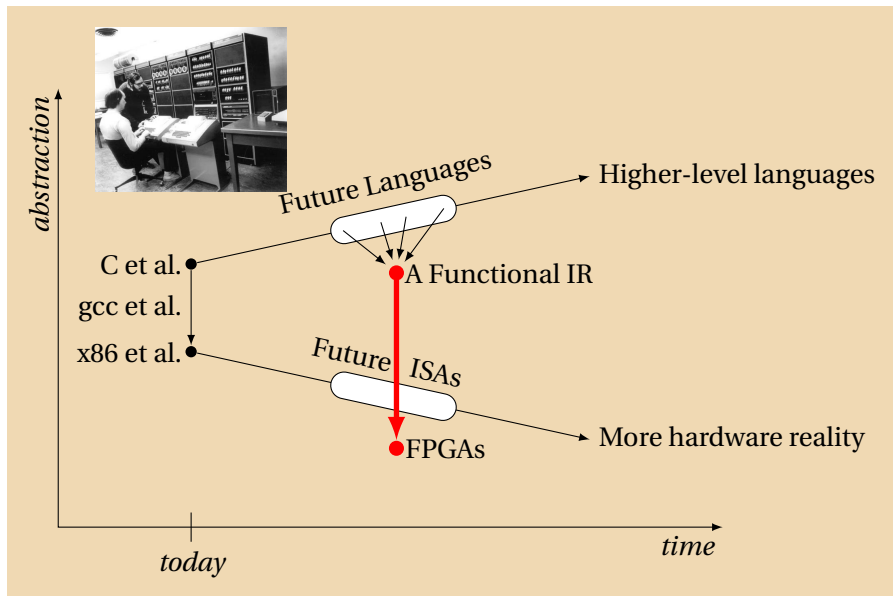
today

time

Our Approach



Our Approach



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?



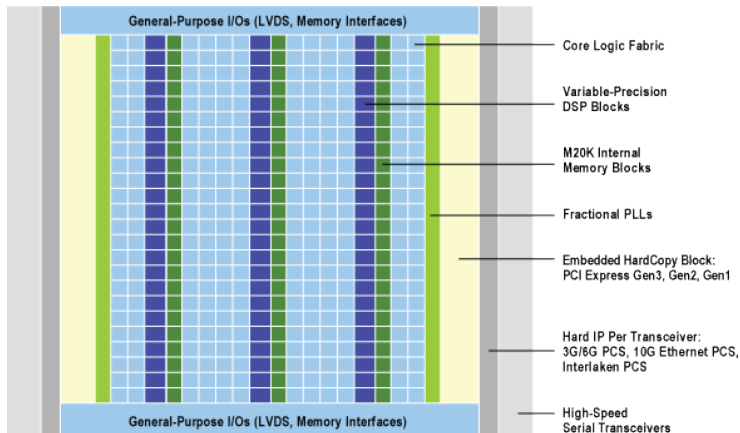
Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

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

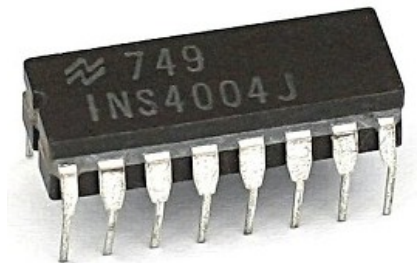
2500 dual-ported 2.5KB 600 MHz memory blocks; 6 Mb total

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

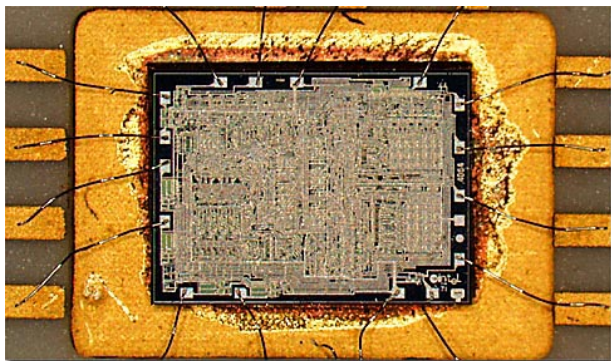
300000 6-input LUTs; 28 nm feature size



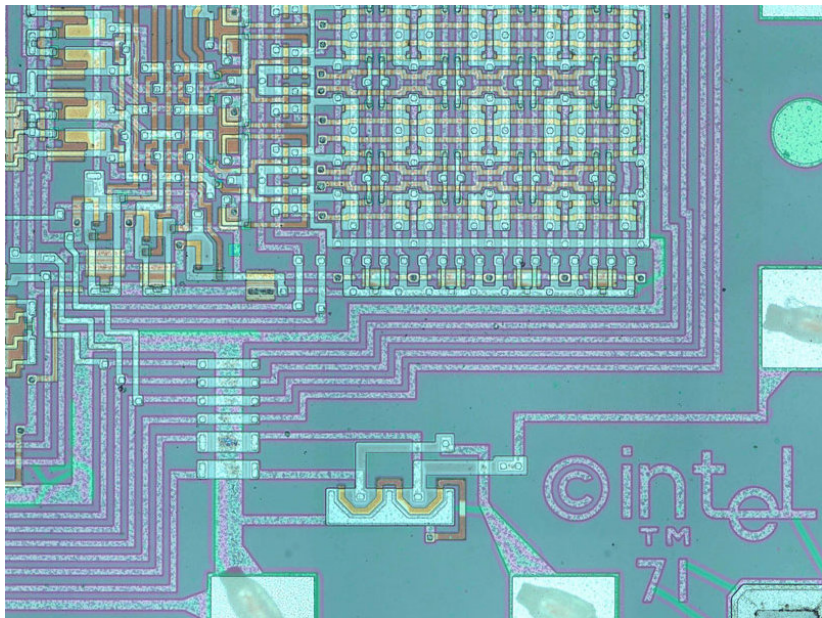
Let's Talk Details



Let's Talk Details



Let's Talk Details



Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler's “Core” representation

*expr ::= name var**

Function call

Includes primitive arithmetic operators and type constructors

Non-tail-recursive calls generally inlined to improve parallelism;
Mycroft and Sharp's [IWLS 2000] propose sharing policies

True recursion transformed to tail recursion with a stack

Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler's “Core” representation

$expr ::= name\ var^*$

| **let** ($var = expr$)⁺ **in** $expr$

Function call

Parallel evaluation

Parallelism and sequencing:

let $v_1 = e_1$

$v_2 = e_2$

$v_3 = e_3$ **in** e

e_1

e_2

e_3

} evaluated in parallel, then e

Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler's "Core" representation

$expr ::= name\ var^*$	Function call
let ($var = expr$) ⁺ in $expr$	Parallel evaluation
case var of ($pat \rightarrow expr$) ⁺	Multiway conditional
$pat ::= literal$	Exact match
$_$	Default
$Constr.$ ($var \mid literal \mid _$) [*]	Match a tagged union

Evaluate and return one of the expressions based on the pattern

Our Starting Point: A Functional IR

Inspired by the Glasgow Haskell Compiler's "Core" representation

$expr ::= name\ var^*$	Function call
$\mathbf{let}\ (var = expr)^+ \mathbf{in}\ expr$	Parallel evaluation
$\mathbf{case}\ var \mathbf{of}\ (pat \rightarrow expr)^+$	Multiway conditional
var	Variable reference
$literal$	Literal value
$pat ::= literal$	Exact match
$_$	Default
$Constr.\ (var \mid literal \mid _)^*$	Match a tagged union

The Type System: Tagged Unions

Types are primitive (Boolean, Integer, etc.) or tagged unions:

$type ::= Type$	Named type/primitive
$ Constr\ Type^* \dots Constr\ Type^*$	Tagged union

Subsume C structs, unions, and enums

Comparable power to C++ objects with virtual methods

Sometimes called “algebraic data types”: sums of products

The Type System: Tagged Unions

Types are primitive (Boolean, Integer, etc.) or tagged unions:

```
type ::= Type                                Named type/primitive  
        | Constr Type* | ... | Constr Type*    Tagged union
```

Examples:

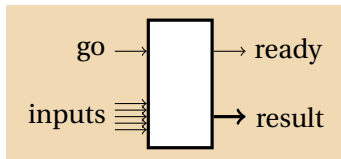
```
data Intlist = Nil                                -- Linked list of integers  
              | Cons Int Intlist
```

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

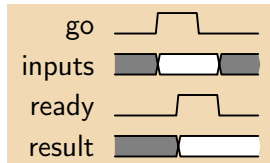
```
data Expr = Literal Int                            -- Arithmetic expression  
           | Var String  
           | Binop Expr Op Expr
```

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

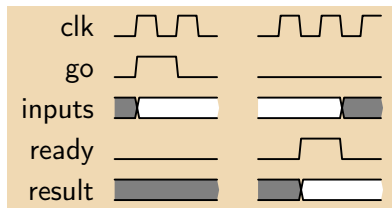

Syntax-Directed Translation of Expressions to Hardware



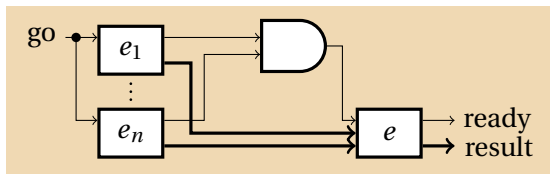
Combinational functions:



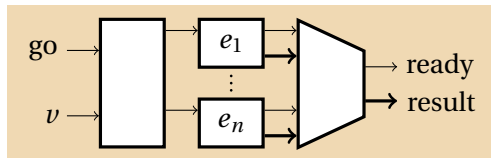
Sequential functions:



Translating Let and Case



Let makes all new variables available to its body.



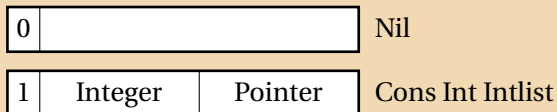
Case invokes one of its sub-expressions, then synchronizes.

Representing Recursive Algebraic Data Types

Consider a list of integers:

```
data Intlist = Nil  
           | Cons Int Intlist
```

An obvious representation:



- ▶ Usual byte-alignment unnecessary & wasteful in hardware
- ▶ Naturally stored & managed in a custom integer-list memory
- ▶ Width of pointer can depend on integer-list memory size

Removing Recursion: Recursive Fibonacci Example

Starting point: a dumb way to compute Fibonacci numbers

$$fib\ 1 = 1$$

$$fib\ 2 = 1$$

$$fib\ n = fib\ (n-1) + fib\ (n-2)$$

Removing Recursion: Recursive Fibonacci

Reformatting

$$\begin{aligned} fib\ 1 &= 1 \\ fib\ 2 &= 1 \\ fib\ n &= fib\ (n-1) + \\ &\quad fib\ (n-2) \end{aligned}$$

Removing Recursion: Continuation-Passing Style

In continuation-passing style (the “and then?” transformation):

<i>fib1</i> 1 <i>c</i>	=	<i>c</i> 1	
<i>fib1</i> 2 <i>c</i>	=	<i>c</i> 1	
<i>fib1</i> <i>n</i> <i>c</i>	=	<i>fib1</i> (<i>n</i> -1)	-- Calls made sequential
	(\ <i>n1</i> ->	<i>fib1</i> (<i>n</i> -2)	-- Intermediates named
	(\ <i>n2</i> ->	<i>c</i> (<i>n1</i> + <i>n2</i>)))	-- Add scheduled last
<i>fib</i> <i>n</i>	=	<i>fib1</i> <i>n</i> (\ <i>x</i> -> <i>x</i>)	-- Wrapper

Removing Recursion: Naming Functions

Naming functions; converting unbound variables to arguments:

$fib1\ 1\ c$	$=$	$c\ 1$	
$fib1\ 2\ c$	$=$	$c\ 1$	
$fib1\ n\ c$	$=$	$fib1\ (n-1)\ (fib2\ n\ c)$	-- Unbound variables passed
$fib2\ n\ c\ n1$	$=$	$fib1\ (n-2)\ (fib3\ n1\ c)$	-- Lambdas named
$fib3\ n1\ c\ n2$	$=$	$c\ (n1 + n2)$	
$fib\ n$	$=$	$fib1\ n\ fib0$	
$fib0\ n$	$=$	n	-- Identity function named

Removing Recursion: True Recursion to Tail Recursion

Introducing a stack; merging functions

```
f      (Fib1 1 c)    = f (Cont c 1)           -- Single function
f      (Fib1 2 c)    = f (Cont c 1)           -- Continuation the stack
f      (Fib1 n c)    = f (Fib1 (n-1) (Fib2 n c))
f (Cont (Fib2 n c) n1) = f (Fib1 (n-2) (Fib3 n1 c))
f (Cont (Fib3 n1 c) n2) = f (Cont c (n1 + n2))
f      (Fib n)      = f (Fib1 n Fib0)
f (Cont Fib0 n)     = n
```

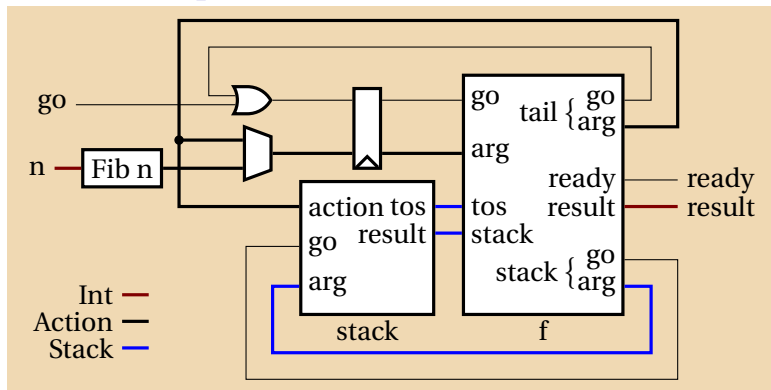
-- Continuations (references to the lambda expressions)

```
data Stack = Fib2 Int Stack -- fib2 n c
           | Fib3 Int Stack -- fib3 n1 c
           | Fib0           -- identity function (bottom of stack)
```

-- Invoke a named function or a continuation

```
data Action = Fib Int      -- fib n (outside call)
            | Fib1 Int Stack -- fib1 n c (recursive call)
            | Cont Stack Int -- c (...) (invoke continuation)
```


Fibonacci Datapath



$f \quad (Fib1 \ 1 \ c) \quad = f \ (Cont \ c \ 1)$
 $f \quad (Fib1 \ 2 \ c) \quad = f \ (Cont \ c \ 1)$
 $f \quad (Fib1 \ n \ c) \quad = f \ (Fib1 \ (n-1) \ (Fib2 \ n \ c))$
 $f \ (Cont \ (Fib2 \ n \ c) \ n1) = f \ (Fib1 \ (n-2) \ (Fib3 \ n1 \ c))$
 $f \ (Cont \ (Fib3 \ n1 \ c) \ n2) = f \ (Cont \ c \ (n1 + n2))$
 $f \quad (Fib \ n) \quad = f \ (Fib1 \ n \ Fib0)$
 $f \ (Cont \ Fib0 \ n) \quad = n$

data $Stack = Fib2 \ Int \ Stack$
 | $Fib3 \ Int \ Stack$
 | $Fib0$

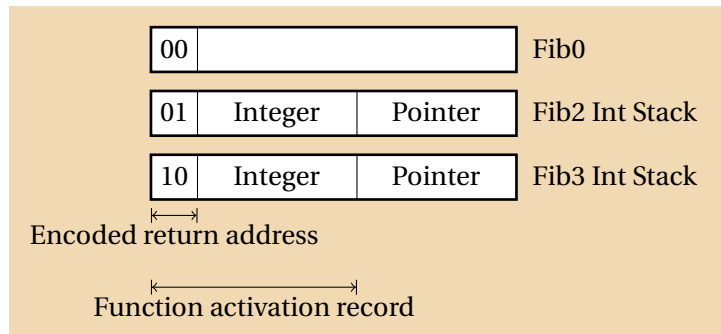
data $Action = Fib \ Int$
 | $Fib1 \ Int \ Stack$
 | $Cont \ Stack \ Int$

Implementing the Stack in Hardware

This uses a list-like stack data type:

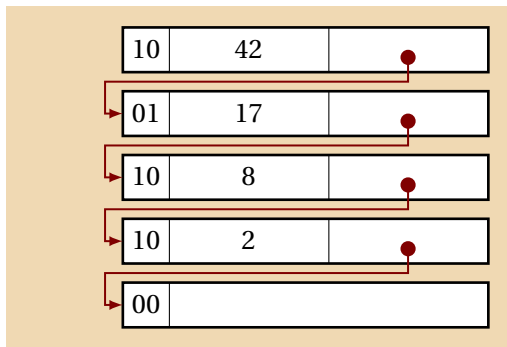
```
data Stack = Fib2 Int Stack  
          | Fib3 Int Stack  
          | Fib0
```

A naïve, but correct, way to implement it in hardware:



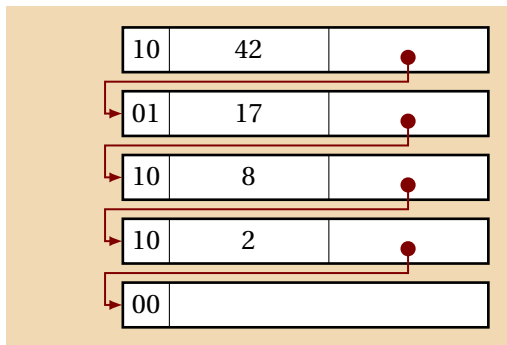
Specializing Data Types: Recovering a Classical Stack

Fib3 42 (Fib2 17 (Fib3 8 (Fib3 2 Fib0)))



Specializing Data Types: Recovering a Classical Stack

Fib3 42 (Fib2 17 (Fib3 8 (Fib3 2 Fib0)))



The only “pop” operation discards the previous top-of-stack

$$f (Cont (Fib3 n1 c) n2) = f (Cont c (n1 + n2))$$

so this code will never generate a tree.

Sequential memory allocation is safe.

Specializing Data Types: Recovering a Classical Stack

Fib3 42 (Fib2 17 (Fib3 8 (Fib3 2 Fib0)))

4:	10	42	3
3:	01	17	2
2:	10	8	1
1:	10	2	0
0:	00		

Sequential memory allocation makes “next” pointers predictable...

Specializing Data Types: Recovering a Classical Stack

Fib3 42 (Fib2 17 (Fib3 8 (Fib3 2 Fib0)))

4:

10	42
----	----

3:

01	17
----	----

2:

10	8
----	---

1:

10	2
----	---

0:

00	
----	--

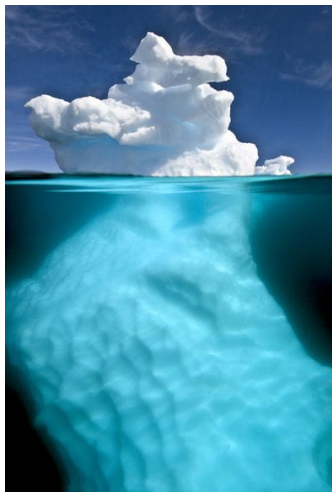
...so there is no need to store them.

Constructor (Fib0) always returns 0.

Constructors (Fib2/3 n s) writes (Fib2/3 n) at $s + 1$ and returns $s + 1$.

Reading 0 returns Fib0; reading s returns (Fib2/3 n $s - 1$).

Specializing Data Types



Stacks are the tip of the iceberg

Synthesizing custom memory systems for specific types is a key goal of this project

Shape Analysis relevant here

This is a simple case; a simple, mathematical IR enables such clever optimizations.

Imagine trying to do this in C.

Unrolling Code for Better Parallelism

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

fib (*n*-1) and *fib* (*n*-2) are functionally independent.

Yet because they share *fib*, they are performed sequentially.

Unrolling Code for Better Parallelism

```
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)
```

By unrolling the recursion once, *fib*' and *fib*'' run in parallel.

A further improvement: balance the work done by *fib*' and *fib*''

Unrolling Types for Better Locality

```
data Stack = Fib2 Int Stack  
          | Fib3 Int Stack  
          | Fib0
```

Each Stack object naturally represents a single activation record

Unrolling Types for Better Locality

```
data Stack = Fib2 Int Stack'  
          | Fib3 Int Stack'  
          | Fib0
```

```
data Stack' = Fib2 Int Stack''  
          | Fib3 Int Stack''  
          | Fib0
```

```
data Stack'' = Fib2 Int Stack'''  
          | Fib3 Int Stack'''  
          | Fib0
```

```
data Stack''' = Fib2 Int Stack  
          | Fib3 Int Stack  
          | Fib0
```

A similar unrolling amounts to packing records that can be processed in parallel

Abstract data types enables this

Imagine trying to do this safely in a C compiler

Example: Huffman Decoder in Haskell

```
data HTree = Branch HTree HTree
           | Leaf Char
```

```
decode :: HTree -> [Bool] -> [Char] -- Huffman tree & bitstream to symbols
```

```
decode table str = decoder table str
```

where

```
decoder (Leaf s) i = s : (decoder table i) -- Identified symbol; start again
```

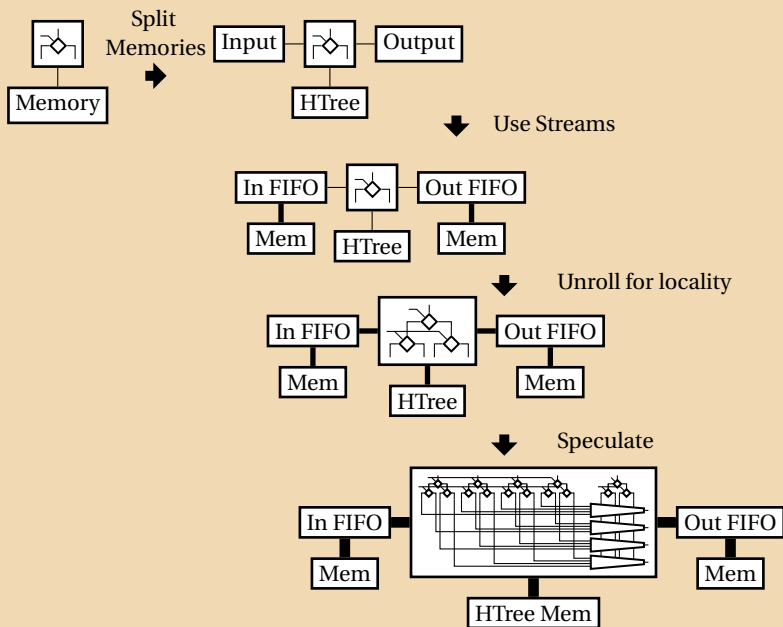
```
decoder _ [] = []
```

```
decoder (Branch f _) (False:xs) = decoder f xs -- 0: follow left branch
```

```
decoder (Branch _ t) (True:xs) = decoder t xs -- 1: follow right branch
```

Three data types: Input bitstream, output character stream, and Huffman tree

Optimizations



Target Applications

- ▶ “Data-parallel irregular applications [that] manipulate large pointer-based data structures like graphs”

[Pingali et al.’s Galois project]

- ▶ Datatype accelerators

Hash tables, Balanced trees, Heaps

- ▶ Application-domain accelerators

Relational databases, Cryptography, Data compression

- ▶ *Non-scientific computing: the stuff that’s hard for vector units and GPGPUs*

Acknowledgements

Project started while at MSR Cambridge



Satnam Singh (now at Google)

See his keynote tomorrow



Simon Peyton Jones (MSR)



Martha Kim (Columbia)

