Compiling Parallel Algorithms to Memory Systems

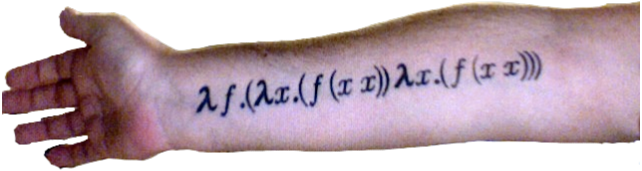
Stephen A. Edwards

Columbia University

RAWFP Workshop, May 29, 2012

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

Functional Programs to FPGAs

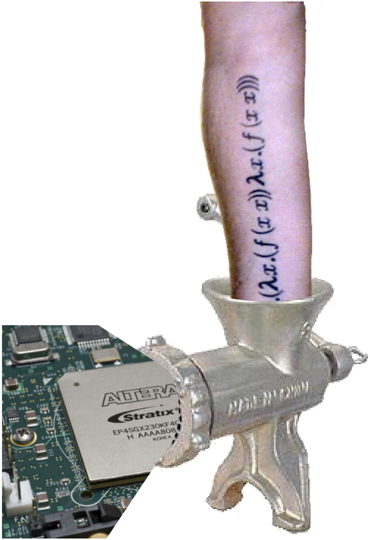


Functional Programs to FPGAs

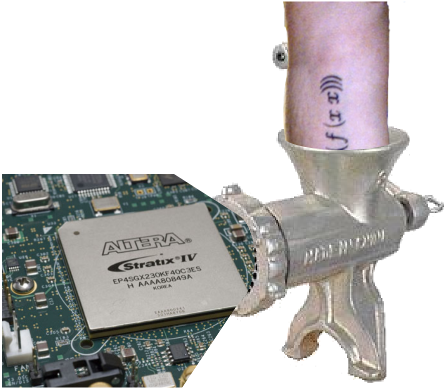
Functional Programs to FPGAs

Functional Programs to FPGAs

Functional Programs to FPGAs

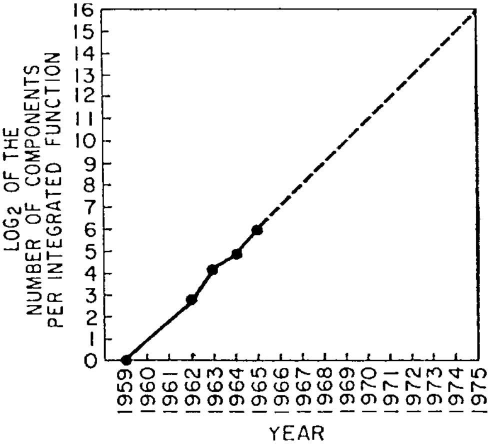


Functional Programs to FPGAs



Functional Programs to FPGAs

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

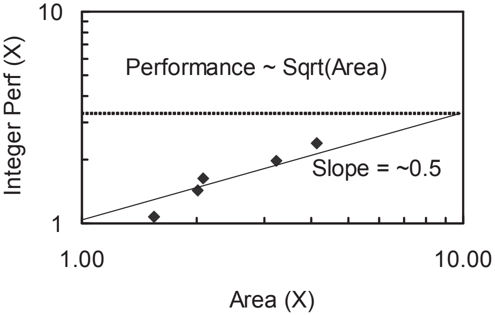


Gordon Moore, *Cramming More Components onto Integrated Circuits*, Electronics, 38(8) April 19, 1965.

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

Closer to every 24 months

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



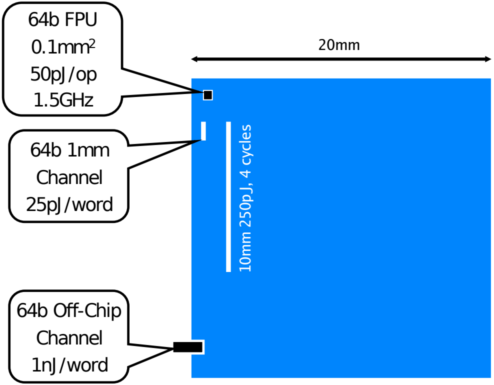
Single-threaded processor performance grows with the square root of area.

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

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

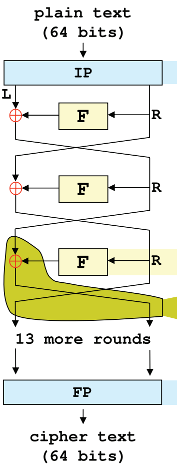
Bill Dally's 2009 DAC Keynote,
The End of Denial Architecture

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




```
public static Unsigned64 DEScoder(KeySchedule keys,
                                  Unsigned64 text)
{
    Unsigned64 block = text.permute(IP.Permutation);

    Unsigned32 R = block.extractBits(sixtyfour.b0 ,
                                     sixtyfour.b31);
    Unsigned32 L = block.extractBits(sixtyfour.b32,
                                     sixtyfour.b63);


    for ( sixteen round ) {

        Unsigned32 F = Fiestel(keys, round, R);

        Unsigned32 X = F ^ L;
        L = R;
        R = X;
    }
    Unsigned64 LR = makeUnsigned64(R, L);

    return LR.permute(FP.Permutation);
}
```

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

```
int squares()
{
  int i = 0,
      sum = 0;

  for (;i<10;i++)
      sum += i*i;
  return sum;
}
```

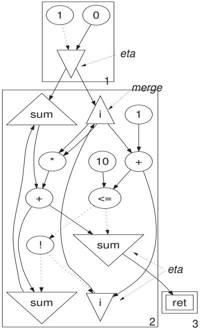


Figure 3: *C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)*

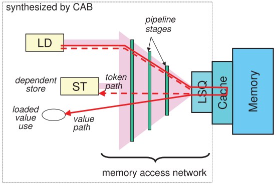


Figure 8: *Memory access network and implementation of the value and token forwarding network. The* LOAD *produces a data value consumed by the oval node. The* STORE *node may depend on the load (i.e., we have a token edge between the* LOAD *and the* STORE*, shown as a dashed line). The token travels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

C to asynchronous logic, monolithic memory

Budiu, Venkataramani, Chelcea and Goldstein, *Spatial Computation*, ASPLOS 2004.

Ghica et al.'s Geometry of Synthesis

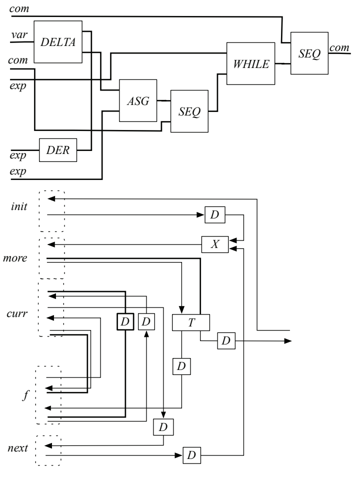


Figure 1. In-place map schematic and implementation

Algol-like imperative language to handshake circuits

Ghica, Smith, and Singh. *Geometry of Synthesis IV*, ICFP 2011

Greaves and Singh's Kiwi

```
public static void SendDeviceID()
{ int deviceID = 0x76;
  for (int i = 7; i > 0; i--)
  { scl = false;
    sda_out = (deviceID & 64) != 0;
    Kiwi.Pause(); // Set it i-th bit of the device ID
    scl = true; Kiwi.Pause(); // Pulse SCL
    scl = false; deviceID = deviceID << 1;
    Kiwi.Pause();
  }
}
```

C# with a concurrency library to FPGAs

Greaves and Singh. *Kiwi*, FCCM 2008

Arvind, Hoe, et al.'s Bluespec

GCD Mod Rule

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

GCD Flip Rule

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

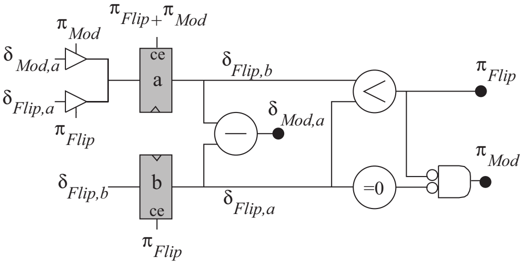


Figure 1.3 Circuit for computing $\mathsf{Gcd}(a, b)$ from Example 1.

Guarded commands and functions to synchronous logic

Hoe and Arvind, *Term Rewriting*, VLSI 1999

Sheeran et al.'s Lava

```
bfly :: CmplxArithmetic m
     => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
  do o1 <- csubtract (i1, i2)
     o2 <- cplus (i1, i2)
     return [o1, o2]

bflys :: CmplxArithmetic m
      => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
  riffle >-> raised n two bfly >-> unriffle
```

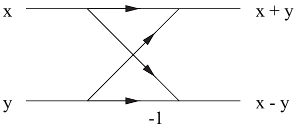


Figure 9: A butterfly

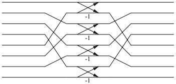


Figure 10: A butterfly stage of size 8 expressed with riffling

Functional specifications of regular structures

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

Kuper et al.'s CλaSH

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

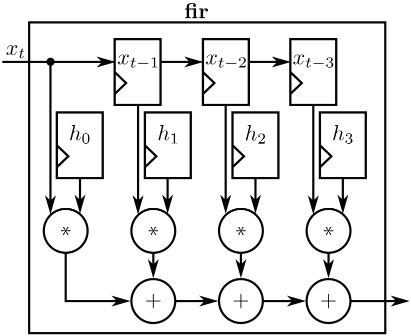


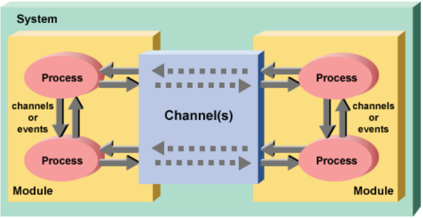
Fig. 6. 4-taps FIR Filter

More operational Haskell specifications of regular structures

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

AutoESL (Xilinx, was Cong's xPilot)

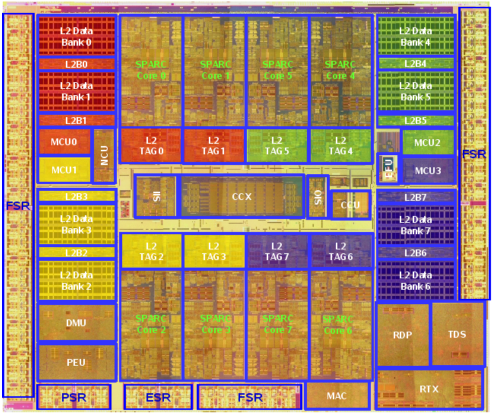
- ***SSDM*** **(System-level Synthesis Data Model)**
  - Hierarchical netlist of concurrent processes and communication channels



  - 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 et al. [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?

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

$$f_1 = c(x + \overline{a}) + a\overline{c}\overline{x}$$
$$f_2 = gx$$
$$x = d(b + f) + \overline{d}(\overline{b} + e)$$

After Brayton et al.'s class on Multi-Level Logic Synthesis

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.

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

Minimize

$$f_1 = bcd + bce + \overline{bd} + \bar{a}c + cdf + ab\bar{c}\bar{d}\bar{e} + a\bar{b}\bar{c}d\bar{f}$$
$$f_2 = bdg + dfg + \overline{bd}g + \bar{d}eg$$

Factor

$$f_1 = c\big(b(d+e) + \bar{b}(\bar{d}+f) + \bar{a}\big) + a\bar{c}(b\bar{d}\bar{e} + \bar{b}d\bar{f})$$
$$f_2 = g\big(d(b+f) + \bar{d}(\bar{b}+e)\big)$$

Decompose

$$f_1 = c(x+\bar{a}) + a\overline{cx}$$
$$f_2 = gx$$
$$x = d(b+f) + \bar{d}(\bar{b}+e)$$

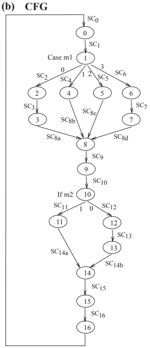
After Brayton et al.'s class on Multi-Level Logic Synthesis

High-Level Synthesis: Adding Time Meant Scheduling

(a)

```
Entity bde is
port ( clock: in bit;
       in1, in2, m1: in integer range 0 to 3;
       m2: in boolean;
       out1,out2,out3,out4: integer range 0 to 3);
end bde;
Architecture behavior of bde is
Begin
  Process
    Variable A, B : integer range 0 to 3;
  Begin
    Wait until not clock'stable and clock='1';
    B := 0;                                  -- O0
    Case (m1) is                             -- O1
      when  0  =>  B := in1 + in2;           -- O2
                   A := A + B;               -- O3
      when  1  =>  B := 3;                   -- O4
      when  2  =>  A := in2;                 -- O5
      when  3  =>  A := 0;                   -- O6
                   B := 2;                   -- O7
    end Case;                                -- O8
    out1 <= A;                               -- O9
    If  (m2)                                 -- O10
    then  A := in1;                          -- O11
    else  out2 <= B;                         -- O12
          out3 <= A + in1;                   -- O13
    end if;                                  -- O14
    out4 <= A + B;                           -- O15
  end process;                               -- O16
end behavior;
```

(b) CFG



DFG

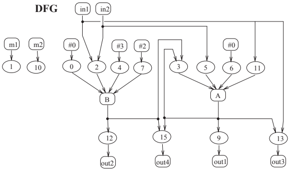
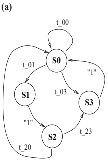


Figure 2: (a) VHDL description; (b) Separate control and data-flow graphs

(a)



State	Operation Node	Condition
S0	0, 1	"1"
	2	m1=0
	4	m1=1
	5	m1=2
	6, 7	m1=3
	8,9,10	m1=[1,2,3]
	11,14,15,16	(m1=[1,2,3]).(m2=1)
	12,13	(m1=[1,2,3]).(m2=0)
S1	3	"1"
S2	8,9,10	"1"
	11, 14,15,16	m2=1
	12,13	m2=0
S3	14,15,16	"1"

Transition Conditions		
t_00 = (m1=[1,2,3]).(m2=1)	t_01 = (m1=0)	
t_03 = (m1=[1,2,3]).(m2=0)	t_20 = (m2=1)	t_23 = (m2=0)

(b)

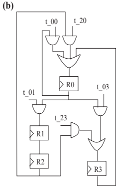


Figure 3: (a) FSM for scheduled CFG in Figure 2(b), (b) Hardware implementation of FSM using one-hot encoding

Bergamaschi, *Behavioral Network Graph*, DAC 1999.

The High-Level Synthesis Lessons

Don't Start From C

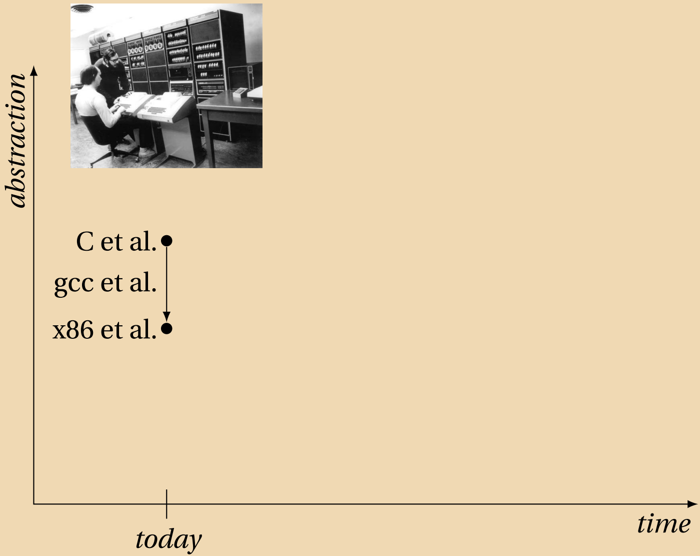
> *"The so-called high-level specifications in reality grew out of the need for simulation and were often little more than an input language to make a discrete event simulator reproduce a specific behavior."*

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

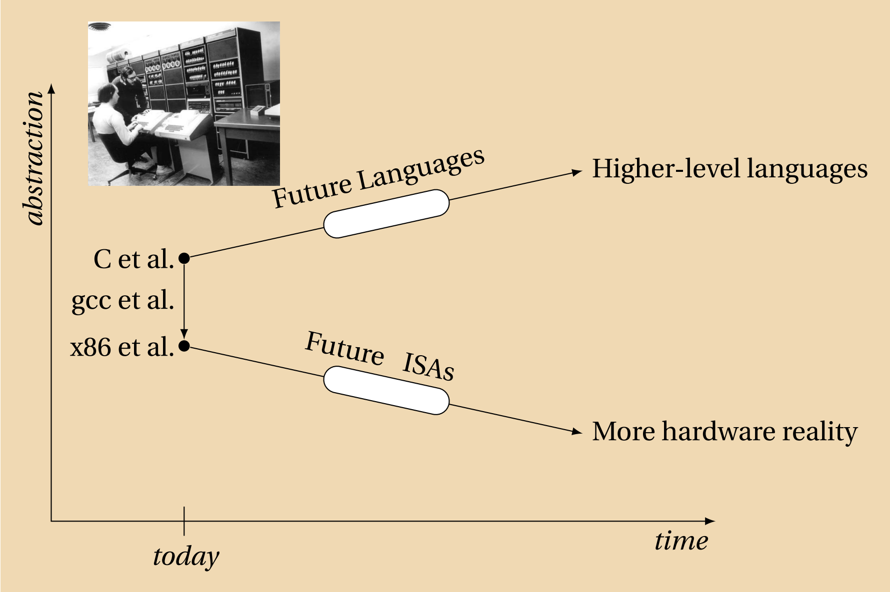
Don't Forget Memory

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

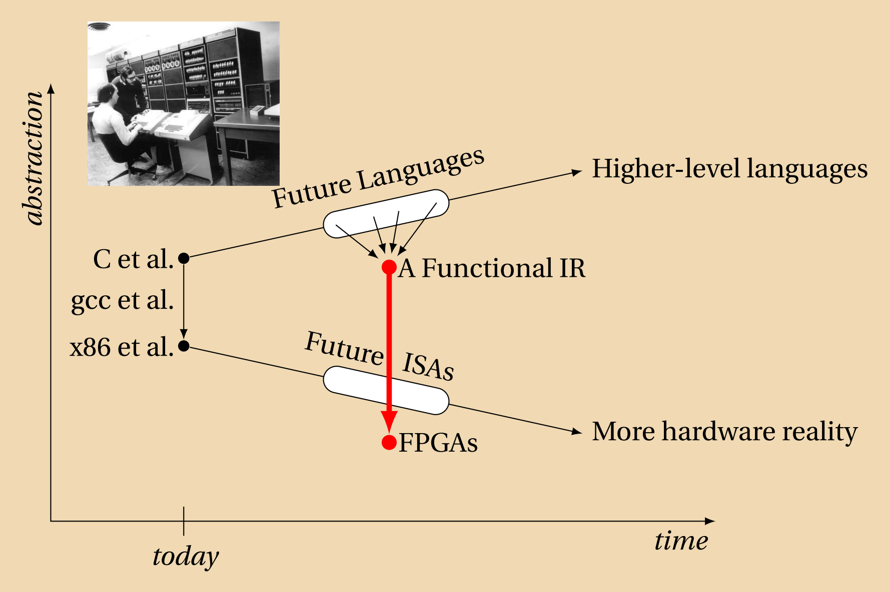
Our Approach



Our Approach

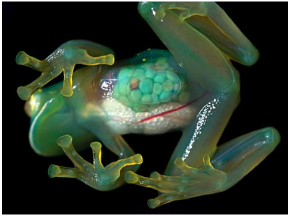


Our Approach



Why Functional Specifications?

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

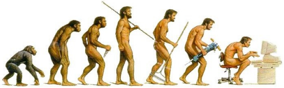


- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.

- 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?

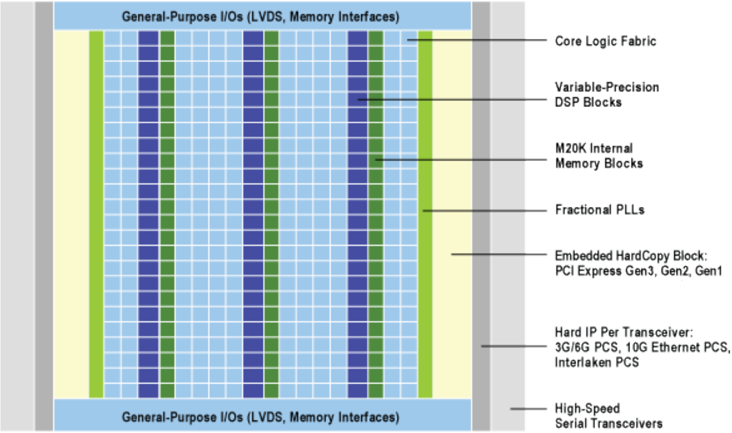
Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

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

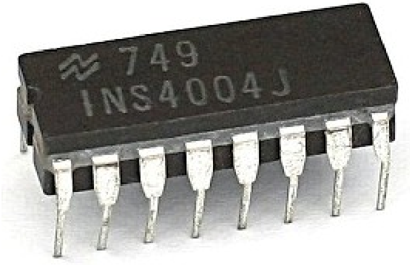
2500 dual-ported 2.5KB 600 MHz memory blocks; 6 Mb total

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

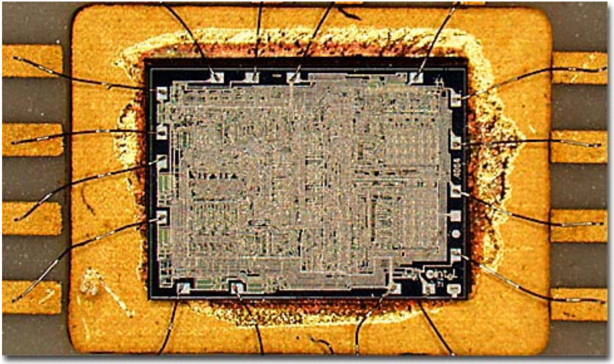
300000 6-input LUTs; 28 nm feature size



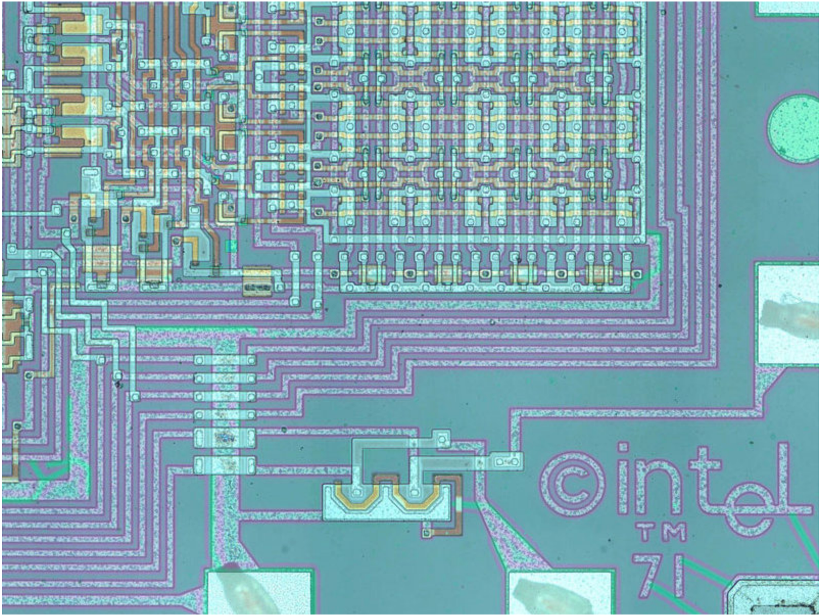
Let's Talk Details



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 [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

$expr ::= name\ var^*$	Function call
$\mid\ \mathbf{let}\ (var = expr)^+\ \mathbf{in}\ expr$	Parallel evaluation

Parallelism and sequencing:

$$\begin{aligned} \mathbf{let}\ & v_1 = e_1 \\ & v_2 = e_2 \\ & v_3 = e_3\ \mathbf{in}\ e \end{aligned} \qquad \left.\begin{matrix} e_1 \\ e_2 \\ e_3 \end{matrix}\right\} \text{evaluated in parallel, then } e$$

Our Starting Point: A Functional IR

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

$\mathit{expr} ::= \mathit{name}\ \mathit{var}^*$	Function call
$\mid\ \mathbf{let}\ (\mathit{var} = \mathit{expr})^+\ \mathbf{in}\ \mathit{expr}$	Parallel evaluation
$\mid\ \mathbf{case}\ \mathit{var}\ \mathbf{of}\ (\mathit{pat}\ \texttt{->}\ \mathit{expr})^+$	Multiway conditional
$\mathit{pat} ::= \mathit{literal}$	Exact match
$\mid\ \_$	Default
$\mid\ \mathit{Constr.}\ (\mathit{var} \mid \mathit{literal} \mid \_)^*$	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

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

The Type System: Tagged Unions

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

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

Subsume C structs, unions, and enums

Comparable power to C++ objects with virtual methods

The Type System: Tagged Unions

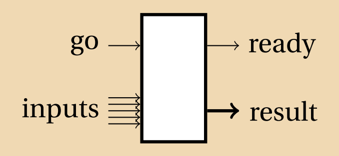
Types are primitive (Boolean, Integer, etc.) or tagged unions:

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

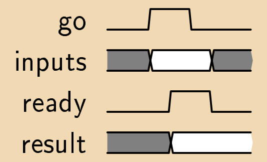
Examples:

```
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
```

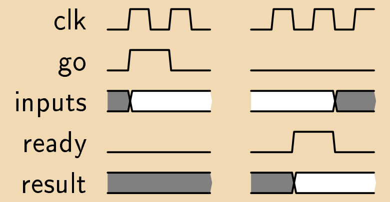
Syntax-Directed Translation of Expressions to HW



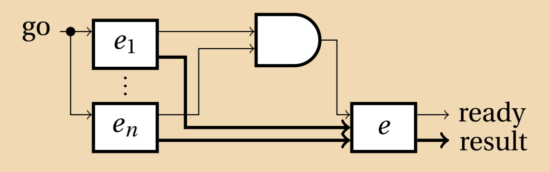
Combinational functions:



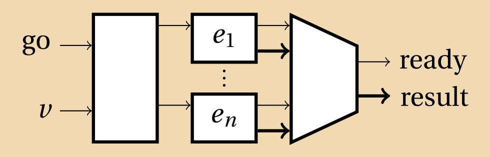
Sequential functions:



Translating Let and Case



Let makes all new variables available to its body.



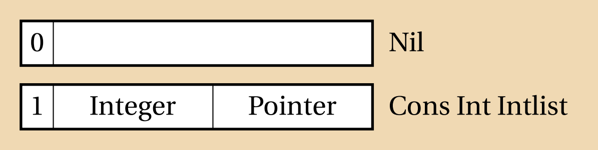
Case invokes one of its sub-expressions, then synchronizes.

Representing Abstract Data Types

Consider an integer list:

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

$$\mathit{fib}\ 1 = 1$$
$$\mathit{fib}\ 2 = 1$$
$$\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$$

Removing Recursion: Recursive Fibonacci

Reformatting

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

Removing Recursion: Continuation-Passing Style

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

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

Removing Recursion: Naming Functions

Naming functions; converting unbound variables to arguments:

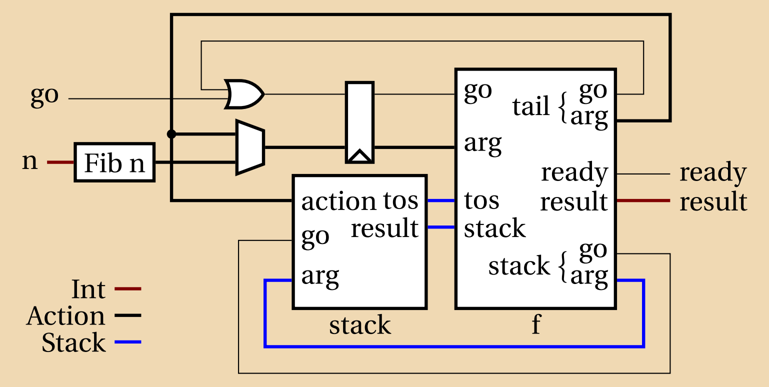
```
fib1 1  c      =      c 1
fib1 2  c      =      c 1
fib1 n  c      =  fib1 (n−1) (fib2 n  c)  −− Unbound variables passed
fib2 n  c n1   =  fib1 (n−2) (fib3 n1 c)  −− Lambdas named
fib3 n1 c n2   =      c (n1 + n2)
fib  n         =  fib1 n fib0
fib0 n         = n                        −− Identity 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)

          -- Named functions and call 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



$$
\begin{aligned}
&f\ (Fib1\ 1\ \ c) &&= f\ (Cont\ c\ 1)\\
&f\ (Fib1\ 2\ \ c) &&= f\ (Cont\ c\ 1)\\
&f\ (Fib1\ n\ \ c) &&= f\ (Fib1\ (n-1)\ (Fib2\ n\ c))\\
&f\ (Cont\ (Fib2\ n\ \ c)\ \ n1) &&= f\ (Fib1\ (n-2)\ (Fib3\ n1\ c))\\
&f\ (Cont\ (Fib3\ n1\ c)\ n2) &&= f\ (Cont\ c\ (n1+n2))\\
&f\ (Fib\ \ n) &&= f\ (Fib1\ n\ Fib0)\\
&f\ (Cont\ \ Fib0\ n) &&= n
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{data}\ Stack &= Fib2\ \mathbf{Int}\ Stack\\
&\mid\ Fib3\ \mathbf{Int}\ Stack\\
&\mid\ Fib0
\end{aligned}
$$

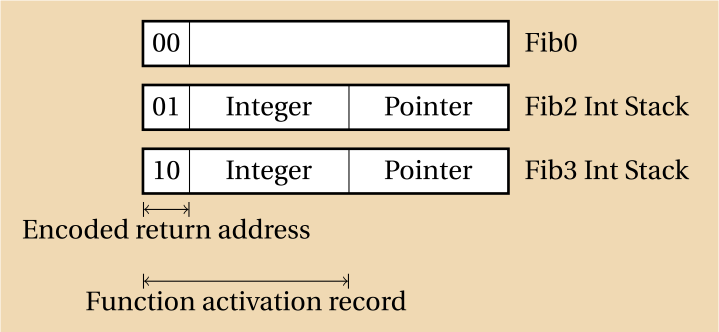
$$
\begin{aligned}
\mathbf{data}\ Action &= Fib\ \mathbf{Int}\\
&\mid\ Fib1\ \mathbf{Int}\ Stack\\
&\mid\ Cont\ Stack\ \mathbf{Int}
\end{aligned}
$$

Implementing the Stack in Hardware

This uses a stack data type that looks like a kind of list:

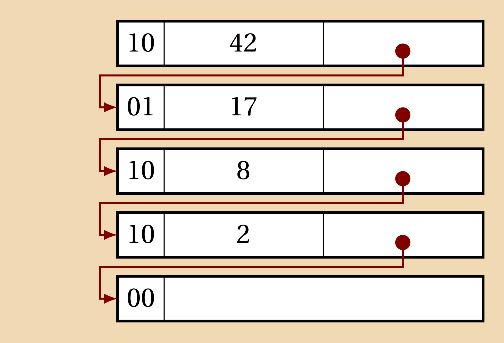
$$\textbf{data}\ \mathit{Stack} = \mathit{Fib2}\ \ \textbf{Int}\ \ \mathit{Stack}$$
$$\quad\quad\quad\quad |\ \mathit{Fib3}\ \ \textbf{Int}\ \ \mathit{Stack}$$
$$\quad\quad\quad\quad |\ \mathit{Fib0}$$

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



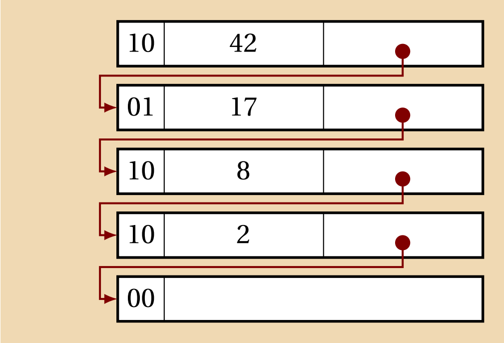
Specializing Data Types: Recovering a Classical Stack

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

$$
\begin{aligned}
&fib\ 0 = 0\\
&fib\ 1 = 1\\
&fib\ n = fib\ (n{-}1) + fib\ (n{-}2)
\end{aligned}
$$

$fib\ (n{-}1)$ 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.

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

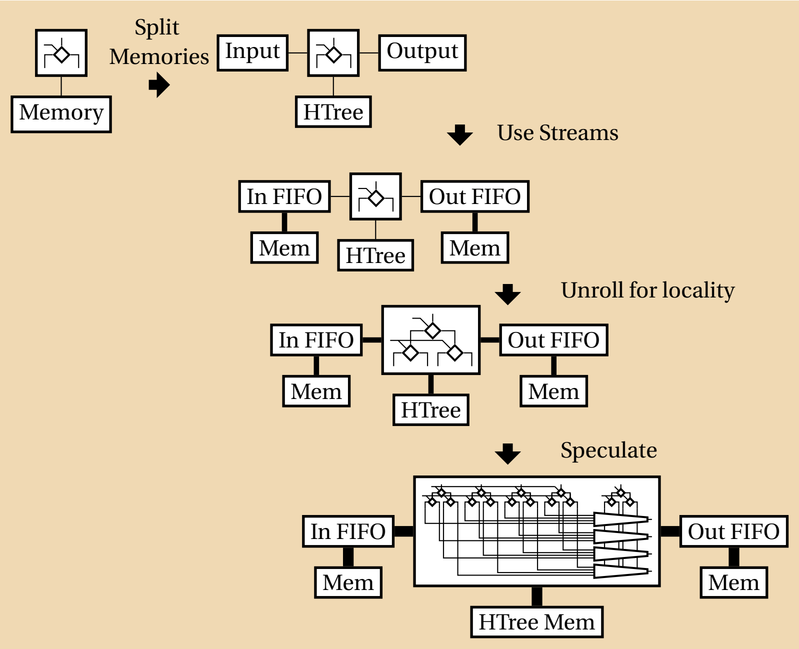
Imagine trying to do this safely in a C compiler

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

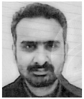
Optimizations



Acknowledgements

Project started while at MSR Cambridge

Satnam Singh (now at Google)



Simon Peyton Jones (MSR)

Martha Kim (Columbia)