High-level Synthesis from Functional Languages

Stephen A. Edwards

Columbia University

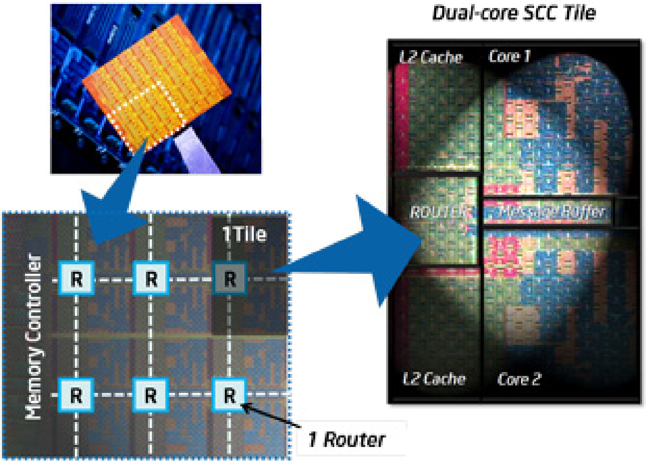
Fall 2010

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

Parallelism is the Big Question

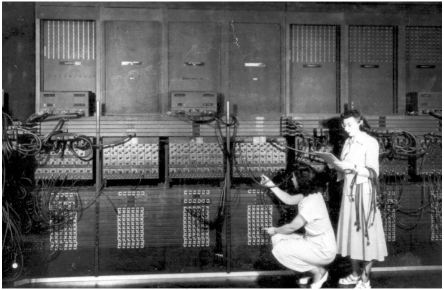
Parallelism is the Big Question

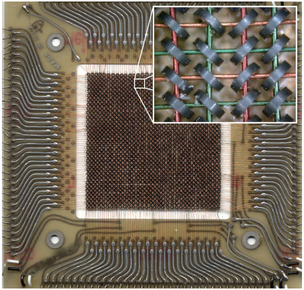
Massive On-Chip Parallelism is Inevitable



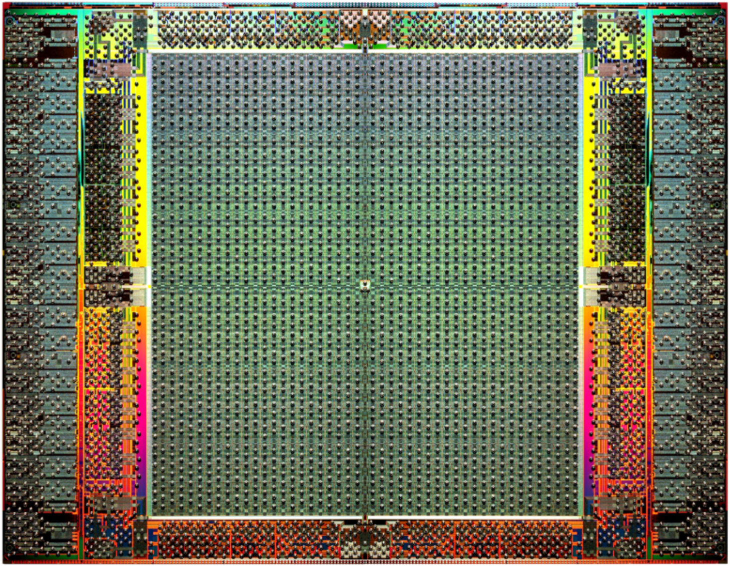
Intel's 48-core "Single Chip Cloud Computer"

The Future is Wires and Memory



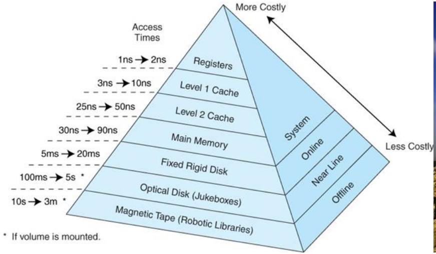


The Future is Already Here



Altera Stratix IV FPGA

The Memory Hierarchy is the Interesting Part



The Big Question

How Do Algorithms Manipulate Data?

My Hypothesis

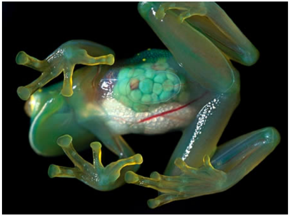
How Do Algorithms Manipulate Data?

We will only be able to answer this in very disciplined languages.

E.g., pure functional languages with immutable data structures

Why Functional Specifications?

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



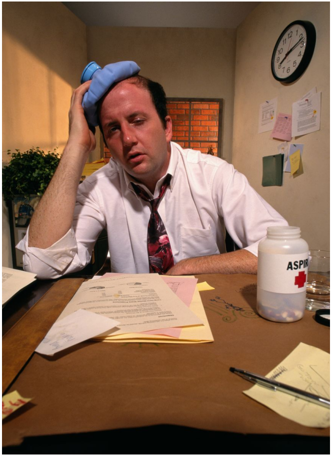
- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.

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

Multiprocessor Memory is a Headache

- Cache Coherency
- Write buffers
- Sequential Memory Consistency
- Memory barriers
- Data Races
- Atomic operations

Immutable data structures simplify these



I Don't Think We Want Laziness

Laziness has certain semantic advantages, but the bookkeeping is probably not worth it

Approach

- 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.

The Big Question′

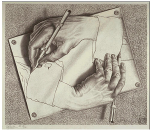
How do we synthesize hardware
from pure functional languages
for FPGAs?

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

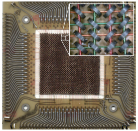
To Implement Real Algorithms in Hardware, We Need

Structured, recursive data types

Recursion to handle recursive data types



Memories



Memory Hierarchy

Example: Huffman Decoder in Haskell

```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

One Way to Encode the Types

Huffman tree nodes: (19 bits)

0	8-bit character	(unused)	Leaf Char
1	9-bit tree ptr.	9-bit tree ptr.	Branch Tree Tree

Boolean input stream: (10 bits)

0	(unused)		Nil
1	bit	8-bit tail pointer	Cons Bool List

Character output stream: (19 bits)

0	(unused)		Nil
1	8-bit character	10-bit tail pointer	Cons Char List

Intermediate Representation Desiderata

Mathematical formalism convenient for performing "parallelizing" transformations, a.k.a. parallel design patterns

- Pipeline
- Speculation
- Multiple workers
- Map-reduce

Intermediate Representation: Recursive "Islands"

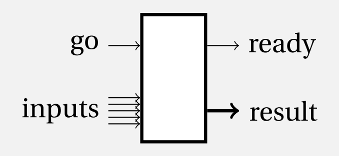
program	::= *island**	
island	::= **island** *name arg** = *expr state**	Group of states w/ stack
state	::= *label arg** = *expr*	Arguments & expression
expr	::= *name var**	Apply a function
	| **let** (*var* = *expr*)$^+$ **in** *expr*	Parallel evaluation
	| **case** *var* **of** (*pattern* -> *expr*)$^+$	Multiway conditional
	| *var*	
	| *literal*	
	| **recurse** *label var** (*var**)	Explicit continuation
	| **return** *var*	
	| **goto** *label var**	Branch to another state
pattern	::= *name var** | *literal* | _	Constructor/literal/def.

Huffman as a Recursive Island

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

decode :: HTree -> [Bool] -> [Char]

decode table str = decoder table str
  where
    decoder (Leaf s) i =
        s : (decoder table i)
    decoder _ [] = []
    decoder (Branch f _) (False:xs) =
        decoder f xs
    decoder (Branch _ t) (True:xs) =
        decoder t xs
```

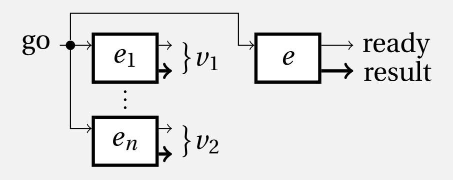
```
island decoder treep ip =
    let r = dec treep treep ip in return r

island dec treep statep ip =
  let i = fetchi ip
      state = fetcht statep in
  case state of
  Leaf a -> recurse s1 a (treep treep ip)
  Branch f t ->
    case i of
    Nil -> let np = Nil in return np
    Cons x xsp ->
      case x of
      True -> goto dec treep t xsp
      False -> goto dec treep f xsp

s1 a rp = let rrp = Cons a rp
          in return rrp
```

The Basic Translation Template

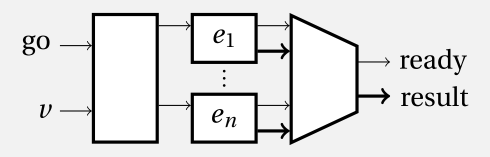


Strobe-based interface: *go* indicates inputs are valid; *ready* pulses once when result is valid.

Translating Let and Case

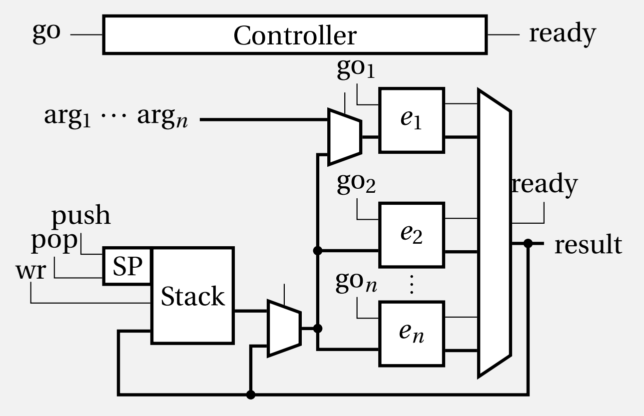


Let makes new values available to an expression.



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

Translating an Island



Each island consists of expressions for each state, its own stack, and a controller that manages the stack and invokes the states.

Constructors and Memory

A constructor is a function that stores data in memory.

$$\text{constructor } \alpha :: \alpha \rightarrow \text{Ptr } \alpha$$

Memory access functions turn pointers into data.

$$\text{fetch } \alpha :: \text{Ptr } \alpha \rightarrow \alpha$$

Memory stores return an address, not take one as an argument

Constructor is responsible for memory management.

By default, each data type gets its own memory.

Duplication for Performance

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

Duplication for Performance

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

After duplicating functions:

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

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

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

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

Unrolling Recursive Data Structures

Like a “blocking factor,” but more general. Idea is to create larger memory blocks that can be operated on in parallel.

Original Huffman tree type:

data *Htree* = *Branch Htree HTree* | *Leaf* **Char**

Unrolled Huffman tree type:

data *Htree* = *Branch Htree' HTree'* | *Leaf* **Char**
data *Htree'* = *Branch' Htree'' HTree''* | *Leaf'* **Char**
data *Htree''* = *Branch'' Htree HTree* | *Leaf''* **Char**

Recursive instances must be pointers; others can be explicit.

Functions must be similarly modified to work with the new types.

Acknowledgements

Project started while at MSR Cambridge

Satman Singh

Simon Peyton Jones