From Recursive Functions to Real FPGAs

Stephen A. Edwards

Columbia University

CCPC, 4 March 2012

\((\lambda x.?) f = \text{FPGA}\)
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
...and it’s Already Here

Altera Stratix IV FPGA
What We are Doing About It

C et al.
gcc et al.
x86 et al.

Future Languages
Future ISAs
More hardware reality
A Functional IR
FPGAs
What We are Doing About It

- C et al.
- gcc et al.
- x86 et al.

Future Languages
- Higher-level languages

Future ISAs
- More hardware reality

abstraction
today

time
What We are Doing About It

Future Languages

C et al.
gcc et al.
x86 et al.

A Functional IR

Future ISAs

FPGAs

More hardware reality

Today

time

Higher-level languages

abstraction
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.
The Memory Hierarchy is the Interesting Part
Multiprocessor Memory is a Headache

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

Immutable data structures simplify these
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
**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
Planned Optimizations

- Split Memories
- Use Streams
- Unroll for locality
- Speculate
One Way to Encode the Types

Huffman tree nodes: (19 bits)

<table>
<thead>
<tr>
<th></th>
<th>8-bit character</th>
<th>(unused)</th>
<th>Leaf Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9-bit tree ptr.</td>
<td>9-bit tree ptr.</td>
<td>Branch Tree Tree</td>
</tr>
</tbody>
</table>

Boolean input stream: (10 bits)

<table>
<thead>
<tr>
<th></th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bit</td>
<td>Cons Bool List</td>
</tr>
<tr>
<td></td>
<td>8-bit tail pointer</td>
<td></td>
</tr>
</tbody>
</table>

Character output stream: (19 bits)

<table>
<thead>
<tr>
<th></th>
<th>(unused)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8-bit character</td>
<td>Cons Char List</td>
</tr>
<tr>
<td></td>
<td>10-bit tail pointer</td>
<td></td>
</tr>
</tbody>
</table>
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”

\[
\text{program} ::= \text{island}^*
\]

\[
\text{island} ::= \text{island} \text{name} \text{arg}^* = \text{expr} \text{state}^*
\]

\[
\text{state} ::= \text{label} \text{arg}^* = \text{expr}
\]

\[
\text{expr} ::= \text{name} \text{var}^* \\
| \text{let} \ (\text{var} = \text{expr})^+ \ \text{in} \ \text{expr} \\
| \text{case} \ \text{var} \ \text{of} \ (\text{pattern} \rightarrow \text{expr})^+ \\
| \text{var} \\
| \text{literal} \\
| \text{recurse} \ \text{label} \ \text{var}^* \ (\text{var}^*) \\
| \text{return} \ \text{var} \\
| \text{goto} \ \text{label} \ \text{var}^*
\]

\[
\text{pattern} ::= \text{name} \text{var}^* \ | \ \text{literal} \ | \ _
\]

Group of states w/ stack
Arguments & expression
Apply a function
Parallel evaluation
Multiway conditional
Explicit continuation
Branch to another state
Constructor/literal/def.
Huffman as a Recursive Island

**data** 
\[HTree = \text{Branch } HTree \ HTree \mid \text{Leaf } \text{Char}\]

\[decode :: HTree \to [\text{Bool}] \to [\text{Char}]\]

\[\text{decode table str } = \text{decoder table str}\]

\[\text{where}\]
\[\text{decoder } (\text{Leaf } s) \ i = s : (\text{decoder table } i)\]
\[\text{decoder } _ \ [] = []\]
\[\text{decoder } (\text{Branch } f \ _) \ (\text{False}:xs) = \text{decoder } f \ xs\]
\[\text{decoder } (\text{Branch } _ \ t) \ (\text{True}:xs) = \text{decoder } t \ xs\]

**island** 
\[\text{decoder treep ip } = \]
\[\text{let } r = \text{dec treep treep } ip \ \text{in return } r\]

**island** 
\[\text{dec treep statep } ip = \]
\[\text{let } i = \text{fetchi } ip \]
\[\text{state } = \text{fetcht } \text{statep } \text{in}\]

\[\text{case state of}\]
\[\text{Leaf } a \to \text{recurse } s1 \ a \ (\text{treep treep } \text{ip})\]
\[\text{Branch } f \ t \to\]
\[\text{case } i \ \text{of}\]
\[\text{Nil } \to \text{let } np = \text{Nil } \text{in return } np\]
\[\text{Cons } x \ xsp \to\]
\[\text{case } x \ \text{of}\]
\[\text{True } \to \text{goto } \text{dec treep } t \ xsp\]
\[\text{False } \to \text{goto } \text{dec treep } f \ xsp\]

\[s1 \ a \ rp = \text{let } rrp = \text{Cons } a \ rp \]
\[\text{in return } rrp\]
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.
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

\[\begin{align*}
\text{fib} &\ 0 = 0 \\
\text{fib} &\ 1 = 1 \\
\text{fib} &\ n = \text{fib} (n-1) + \text{fib} (n-2)
\end{align*}\]
Duplication for Performance

\[ \text{fib} \ 0 = 0 \]
\[ \text{fib} \ 1 = 1 \]
\[ \text{fib} \ n = \text{fib} \ (n-1) + \text{fib} \ (n-2) \]

After duplicating functions:

\[ \text{fib} \ 0 = 0 \]
\[ \text{fib} \ 1 = 1 \]
\[ \text{fib} \ n = \text{fib}' \ (n-1) + \text{fib}'' \ (n-2) \]

\[ \text{fib}' \ 0 = 0 \]
\[ \text{fib}' \ 1 = 1 \]
\[ \text{fib}' \ n = \text{fib}' \ (n-1) + \text{fib}' \ (n-2) \]

\[ \text{fib}'' \ 0 = 0 \]
\[ \text{fib}'' \ 1 = 1 \]
\[ \text{fib}'' \ n = \text{fib}'' \ (n-1) + \text{fib}'' \ (n-2) \]

Here, \text{fib}' and \text{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 (now at Google)

Simon Peyton Jones (MSR)

Martha Kim (Columbia)