Flexible Hardware Design at Low Levels of Abstraction

Emil Axelsson

Hardware Description and Verification

May 2008
Acknowledgment

This project received Intel-custom funding through the Semiconductor Research Corporation
Why low-level?

- **Related question**: Why is some software written in C?  
  *(but difference between high- and low-level is much greater in hardware)*

- **Ideal**:
  Software-like code → magic compiler → chip masks

```haskell
gadget a b = case a of
  2 -> thing (b+10)
  3 -> thing (b+20)
  _ -> fixNumber a
```
Why low-level?

- Related question: Why is some software written in C? (but difference between high- and low-level is much greater in hardware)

- Ideal:
  Software-like code → magic compiler → chip masks

```haskell
gadget a b = case a of
  2 -> thing (b+10)
  3 -> thing (b+20)
_  -> fixNumber a
```
Why low-level?

Reality:

“Ascii schematic” → chain of synthesis tools → chip masks
Why low-level?

- Reality:
  “Ascii schematic” → chain of synthesis tools → chip masks

- Reiterate to improve timing/power/area/etc.
  - Very costly / time-consuming

- Each fabrication costs ≈ $1,000,000
Failing abstraction

- Realistic flow cannot avoid low-level awareness
- Paradox
  - Modern designs require higher abstraction level
  - ...but...
  - Modern chip technologies make abstraction harder
    - Main problem: Routing wires are dominant in signal delays and power consumption
- Controlling the wires is key to the performance!
Gate vs. wire delay under scaling
Physical design level

- Certain high-performance components (e.g. arithmetic) need to be designed at even lower level

- Physical level:
  - A set of connected standard cells (implemented gates)
  - Absolute or relative positions of cells (placement)
  - Shape of connecting wires (routing)
Physical design level

- Design by interfacing to physical CAD tools
  - Call automatic tools for certain tasks (mainly routing)

- Often done through scripting code
  - Tedious
  - Hard to explore design space
  - Limited design reuse

- Aim of this work:
  *Raise the abstraction level of physical design!*
Two ways to raise abstraction

- Automatic synthesis
  + Powerful abstraction
  - May not be optimal for e.g. high-performance arithmetic
  - Opaque (hard to control the result)
  - Unstable (heuristics-based)

- Language-based techniques (higher-order functions, recursion, etc.)
  + Transparent, stable
  - Still quite low-level
  - Somewhat limited to regular circuits
Two ways to raise abstraction

- Automatic synthesis
  + Powerful abstraction
  - May not be optimal for e.g. high-performance arithmetic
  - Opaque (hard to control the result)
  - Unstable (heuristics-based)

- Language-based techniques (higher-order functions, recursion, etc.)
  + Transparent, stable
  - Still quite low-level
  - Somewhat limited to regular circuits

Our approach
Lava

- Gate-level hardware description in Haskell
- *Parameterized module generators*: Haskell programs that generate circuits
  - Can be smart, e.g. optimize for speed in a given environment
- Basic placement expressed through combinators
- Used successfully to generate high-performance FPGA cores
Wired: Extension to Lava

- Finer control over geometry
- More accurate performance models
  - Feedback from timing/power analysis enables self-optimizing generators
- Wire-awareness (unique for Wired)
  - Performance analysis based on wire length estimates
  - Control routing through “guides” (experimental)
- ...

Monads in Haskell

- Haskell functions are pure
- Side-effects can be “simulated” using monads [8]

```haskell
add a = do
  m <- get
  put (m+a)
  return (m+a)
```

Syntactic sugar, expands to a pure program with explicit state passing

```haskell
*Main> print $ evalState (add 1) 10
11
```

Monads can also be used to model e.g. IO, exceptions, non-determinism etc.
Monad combinators

Haskell has a general and well-understood combinator library for monadic programs

mapAdd = mapM add [10,10,10,10]

*Main> print $ evalState mapAdd 0
[10,20,30,40]

*Main> print $ evalState ((add =>> add) 1) 10
22
Example: Parallel prefix

Given inputs \( x_1, x_2, \ldots, x_n \)

compute

\[ y_1 = x_1 \]

\[ y_2 = x_1 \circ x_2 \]

\[ \ldots \]

\[ y_n = x_1 \circ x_2 \circ \ldots \circ x_n \]

for \( \circ \), an associative (but not necessarily commutative) operator
Parallel prefix

- Very central component in microprocessors
- Most common use: Computing carries in fast adders
- Trying different operators:
  - **Addition**: prefix (+) [1,2,3,4]
Parallel prefix

- Very central component in microprocessors
- Most common use: Computing carries in fast adders
- Trying different operators:
  - **Addition**: prefix (+) \([1,2,3,4]\)
    
    \[= [1, 1+2, 1+2+3, 1+2+3+4] = [1,3,6,10]\]
Parallel prefix

- Very central component in microprocessors
- Most common use: Computing carries in fast adders
- Trying different operators:
  - **Addition:** prefix (+) [1, 2, 3, 4]
    \[
    = [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10]
    \]
  - **Boolean OR:** prefix (||) [F, F, F, T, F, T, T, F]
Parallel prefix

- Very central component in microprocessors
- Most common use: Computing carries in fast adders
- Trying different operators:
  - **Addition**: prefix (+) \([1,2,3,4]\)
    
    \[= [1, 1+2, 1+2+3, 1+2+3+4] = [1,3,6,10]\]

  - **Boolean OR**: prefix (||) \([F,F,F,T,F,T,T,F]\)
    
    \[= [F,F,F,T,T,T,T,T]\]
Parallel prefix

Implementation choices (relying on associativity):

prefix \( (\circ) [x_1, x_2, x_3, x_4] = [y_1, y_2, y_3, y_4] \)

- **Serial**: \( y_4 = ((x_1 \circ x_2) \circ x_3) \circ x_4 \)
- **Parallel**: \( y_4 = (x_1 \circ x_2) \circ (x_3 \circ x_4) \)
- **Sharing**: \( y_4 = y_3 \circ x_4 \)
There are many of them...

Sklansky

Brent-Kung

Ladner-Fischer
Parallel prefix: Sklansky

\[
\text{sklansky op [a] = return [a]}
\]
\[
\text{sklansky op as = do}
\]
\[
\hspace{1cm} \text{let } k \quad = \text{length as `div` 2}
\]
\[
\hspace{1cm} (ls,rs) = \text{splitAt k as'}
\]
\[
ls' \leftarrow \text{sklansky op ls}
\]
\[
rs' \leftarrow \text{sklansky op rs}
\]
\[
rs'' \leftarrow \text{sequence [op (last ls', r) | r \leftarrow rs']}
\]
\[
\text{return (ls' ++ rs'')}
\]
sklansky \text{op} \ [a] = \textit{space cellWidth} \ [a]

sklansky \text{op} \text{as} = \textit{downwards 1} \$ \text{do}

\begin{align*}
\text{let } k &= \text{length as `\div` 2} \\
(ls,rs) &= \text{splitAt } k \text{ as'}
\end{align*}

\begin{align*}
(ls',rs') &\leftarrow \textit{rightwards 0} \$ \text{liftM2 (,)} \\
&(sklansky \text{op } ls) \\
&(sklansky \text{op } rs)
\end{align*}

rs'' \leftarrow \textit{rightwards 0} \$ \\
\text{sequence} \ [\text{op } \text{(last ls', r)} \mid r \leftarrow \text{rs'}] \\
\text{return } (ls' ++ rs'')
Sklansky with placement

Simple postscript allows interactive development of placement
Refinement: Add routing guides

bus = rightwards 0 . mapM bus1
    where
        bus1 = space 2750 =>> guide 3 500 =>> space 1250

sklanskyIO op = downwards 0
    $ inputList 16 "in"
    >>= bus
    >>= space 1000
    >>= sklansky op
    >>= space 1000
    >>= bus
    >>= output "out"

Reusing standard (monadic)
Haskell combinators
(nothing Wired-specific)
Sklansky with guides
sklansky op [a] = space cellWidthD [a]

sklansky op as = downwards 1 $ do

    bus as
    let k = length as `div` 2
            (ls,rs) = splitAt k as

            (ls',rs') <- rightwards 0 $ liftM2 (,)
            (sklansky op ls)
            (sklansky op rs)

            rs'' <- rightwards 0 $
                sequence [op (last ls', r) | r <- rs']
            bus (ls' ++ rs'')
Sklansky with guides
Experiment: Compaction

\[ \text{sklansky op [a]} = \text{space cellWidthD [a]} \]

\[ \text{sklansky op [a]} = \text{return [a]} \]

Buses were compacted separately
Export to CAD tool (Cadence Soc Encounter)

Auto-routed in Encounter

Exchanged using DEF file format

Odd rows flipped to share power rails

Simple change in recursive call:

```
sklansky (flipY.op) ls
```
Fast, low-power prefix networks

- Mary Sheeran has developed circuit generators in Lava that search for fast, low-power parallel prefix networks.

- Initially, crude performance models:
  - Delay: Logical depth
  - Power: Number of operators

- Still good results

- Now using Wired to improve accuracy:
  - Static timing/power analysis using models from cell library
prefix \( \mathbf{f} \mathbf{p} = \text{memo pm} \)

where

\[
\begin{align*}
\text{pm} (\text{
},w) &= \text{perhaps id'} (\text{
},w) \\
\text{pm} ([\text{i}],w) &= \text{perhaps id'} ([\text{i}],w) \\
\text{pm} (\text{is},w) &| 2^\text{(maxd(is,w))} < \text{length is} = \text{Fail} \\
\text{pm} (\text{is},w) &= (\text{bestOn is} \ \mathbf{f} \ . \ \text{dropFail}) \\
&| \ \text{wrpC} \ \text{ds} \ (\text{prefix} \ \mathbf{f} \ \mathbf{p}) \ (\text{prefix} \ \mathbf{p} \ \mathbf{p}) \\
&| \ \text{ds} \leftarrow \text{igen} \ldots \\
\end{align*}
\]

where

\[
\begin{align*}
\text{wrpC} \ \text{ds} \ \text{p1} \ \text{p2} &= \\
\text{wrp} \ \text{ds} \ (\text{perhaps id'} \ \text{c}) \ (\text{p1} \ \text{c1}) \ (\text{p2} \ \text{c2}) \\
\end{align*}
\]
Minimal change to search algorithm

prefix \( fp \) = memo pm

where

\[
\begin{align*}
\text{pm} \ ([],w) & = \text{perhaps id'} \ ([],w) \\
\text{pm} \ ([i],w) & = \text{perhaps id'} \ ([i],w) \\
\text{pm} \ (is,w) | 2^{(\text{maxd}(is,w))} < \text{length is} & = \text{Fail} \\
\text{pm} \ (is,w) & = (\text{bestOn is} \ f \ . \ \text{dropFail}) \\
& \quad [ \ \text{wrpC} \ ds \ (\text{prefix} \ \text{fp}) \ (\text{prefix} \ pp) \\
& \quad \quad | \ ds \leftarrow \text{igen} \ldots \ ] \\
\text{where} \\
\text{wrpC} \ ds \ p1 \ p2 & = \\
& \quad \text{wrp} \ ds \ (\text{perhaps id'} \ c1) \ (p2 \ c2)
\end{align*}
\]

Plug in cost functions that analyze the placed network through Wired
85 bits, depth 8
**Design exploration**

- 85 inputs, depth 8, varying allowed fanout

<table>
<thead>
<tr>
<th>Fanout</th>
<th>Delay [ns]</th>
<th>Power [mW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.646</td>
<td>15.2</td>
</tr>
<tr>
<td>8</td>
<td>0.628</td>
<td>15.7</td>
</tr>
<tr>
<td>9</td>
<td>0.624</td>
<td>15.9</td>
</tr>
<tr>
<td>10</td>
<td>0.620</td>
<td>16.1</td>
</tr>
</tbody>
</table>

- At 128 bits, minimum depth is slower than going one deeper (crude delay model fails)

- Accurate model consistent with timing report from Encounter
Wire-aware hardware design methods needed

Wired offers flexible hardware design at low levels of abstraction
  - Sklansky
    - At Intel: 1000 lines of scripting code (Perl)
    - In Wired: <50 lines (though fewer details)
  - Layout-/wire-aware design exploration