Exact Combinational Circuit Synthesis in Haskell

Paul Tarau
Department of Computer Science and Engineering
University of North Texas

Abstract. An exact synthesizer for small circuits, is provided as a literate Haskell program.

Keywords: functional programming and hardware design, symbolic circuit modeling, exact combinational logic synthesis

1 Introduction

We need the following Haskell modules:

```haskell
module Syn where
import Data.List
import Data.Bits
import Data.Array
```

We start with some auxiliary functions that we will use.

Bitvector Boolean Operation Definitions

```haskell
type N = Integer
nand_ :: N -> N -> N -> N
nor_ :: N -> N -> N -> N
impl_ :: N -> N -> N -> N
less_ :: N -> N -> N -> N
and_ :: N -> N -> N -> N
nand_ mask x y = mask .&. (complement (x .&. y))
nor_ mask x y = mask .&. (complement (x .| y))
impl_ mask x y = (mask .&. (complement x)) .| y
less_ _ x y = x .&. (complement y)
and_ _ x y = x .&. y
```

Boolean Operation Encodings and Names

```haskell
opcode m 0 = nand_ m
opcode m 1 = nor_ m
opcode m 2 = impl_ m
```
opcode m 3 = less m
opcode _ 4 = xor
opcode m 5 = and m
opcode _ n = error ("unexpected opcode:"++(show n))

opname 0 = "nand"
opname 1 = "nor"
opname 2 = "impl"
opname 3 = "less"
opname 4 = "xor"
opname 5 = "and"
opname n = error ("no such opcode:"++(show n))

A Few Interesting Libraries

symops = [0,1]
asymops = [2,3]
impl_and = [2,5]

Before looking at the actual code, one can try it out through a few simple tests.

Tests for the Circuit Synthesizer

t0 = findFirstGood symops 3 8 71
t1 = syn asymops 3 71
t2 = mapM_ print (synall asymops 2)
t3 = syn symops 3 83
t4 = syn asymops 3 83
t5 = syn [0..4] 3 83 -- ite with all ops
   -- x xor y xor z -- cpu intensive
t6 = syn asymops 3 105

2 Exact Combinational Circuit Synthesis

We start by reviewing a mechanism for fast boolean evaluation, following [1] and [2].

2.1 Evaluation of Boolean Functions with Bitvector Operations

The boolean evaluation mechanism uses integer encodings of $2^n$ bits for each boolean variable $x_0, \ldots, x_{n-1}$. In a way reminding of qubits in quantum computing, bitvector operations are used to evaluate all value combinations at once.

Proposition 1 (Knuth,[1]) Let $x_k$ be a variable for $0 \leq k < n$ where $n$ is the number of distinct variables in a boolean expression. Then column $k$ of the truth table represents, as a bitstring, the natural number:

$$x_k = (2^{2^n} - 1)/(2^{2^{n-1}} + 1)$$

(1)
For instance, if $n = 2$, the formula computes $x_0 = 3 = [0, 0, 1, 1]$ and $x_1 = 5 = [0, 1, 0, 1]$.

The following functions, working with arbitrary length bitstrings are used to evaluate the $[0..n-1]$ variables $x_k$ with formula 1 and map the constant 1 to the bitstring of length $2^n$, 111...1. The constant 1 is provided by the function allOnes.

\[
\text{allOnes nvars} = 2^{2^nvars} - 1
\]

Next we define a function providing the (arbitrary size) Integer representation of the $k$-th boolean variable (out of $n$).

\[
\text{var_n n k} = \text{var_mn (allOnes n) n k}
\]

\[
\text{var_mn mask n k} = \text{mask \ 'div' \ (2^{(2^{n-k-1})})+1}
\]

We have used in \text{var_n} an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of [1]. Intuitively, it is based on the idea that one can look at $n$ variables as bitstring representations of the $n$ columns of the truth table.

Variables representing such bitstring-truth tables (seen as projection functions) can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 is represented as 0 while the constant 1 is represented as $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

We will now use these variable encodings for combinational circuit synthesis, known to be intractable for anything beyond a few input variables. Clearly, a speed-up by a factor proportional to the machine’s wordsize matters in this case.

### 2.2 Encoding the Primary Inputs

First, let us extend the encoding to cover constants 1 and 0, that we will represent as “variables” $n$ and $n+1$ and encode as vectors of $n$ zeros or $n$ ones (i.e. $2^{2^n} - 1$, passed as the precomputed parameter $m$ to avoid costly recomputation).

\[
\text{encode_var m n k} | \text{ k=n = m}
\]

\[
\text{encode_var m n k} | \text{ k=n+1 = 0}
\]

\[
\text{encode_var m n k} = \text{var_mn m n k}
\]

Next we can precompute all the inputs knowing the number $n$ of primary inputs for the circuit we want to synthesize:

\[
\text{init_inputs n} = \text{0:m:(map (encode_var m n) [0..n-1]) where m=allOnes n}
\]

\[
> \text{init_inputs 3}
[0, 15, 3, 5]
\]

\[
> \text{init_inputs 3}
[0, 255, 15, 51, 85]
\]
Given that inputs have all distinct encodings, we can decode them back - this function will be needed after the circuit is found.

\[
\text{decode_var nvars } v \mid v = (\text{allOnes nvars}) = \text{nvars}
\]

\[
\text{decode_var nvars } 0 = \text{nvars+1}
\]

\[
\text{decode_var nvars } v = \text{head}
\]

\[
[k | k \leftarrow [0..\text{nvars}-1], (\text{encode_var } m \text{ nvars } k) = v]
\]

where \( m = \text{allOnes nvars} \)

\[
\text{map (decode_var 2) (init_inputs 2)}
\]

\[
[3,2,0,1]
\]

\[
\text{map (decode_var 3) (init_inputs 3)}
\]

\[
[4,3,0,1,2]
\]

We can now connect the inputs to their future occurrences as leaves in the DAG representing the circuit. This means simply finding all the functions from the set of input variables to the set of their occurrences, represented as a list (with possibly repeated) values.

\[
\text{bindings } 0 \text{ us } = [[]]
\]

\[
\text{bindings } n \text{ us } =
\]

\[
[zs | ys \leftarrow \text{bindings } (n-1) \text{ us}, zs \leftarrow \text{map (\text{\_})} \text{ us}]
\]

\[
> \text{bindings } 2 \ [0,3,5]
\]

\[
[[0,0],[3,0],[5,0],[0,3],[3,3],\ [5,3],[0,5],[3,5],[5,5]]
\]

For fast lookup, we place the precomputed value combinations in a list of arrays.

\[
\text{generateVarMap } \text{occs vs } =
\]

\[
\text{map (listArray } (0,\text{occs}-1) \text{) (bindings } \text{occs vs)}
\]

\[
> \text{generateVarMap } 2 \ [3,5]
\]

\[
[\text{array } (0,1) \ [(0,3),(1,3)], \text{ array } (0,1) \ [(0,5),(1,3)],\ [5,3],[0,5],[3,5],[5,5], \text{ array } (0,1) \ [(0,3),(1,5)], \text{ array } (0,1) \ [(0,5),(1,5)]]
\]

2.3 The Folds and the Unfolds

We now are ready to generate trees with library operations marking internal nodes of type \( F \) and primary inputs marking the leaves of type \( V \).

\[
\text{data } T \ a = V \ a \mid F \ a (T \ a) \ (T \ a) \text{ deriving (Show, Eq)}
\]

Generating all trees is a variant of an \textbf{unfold} operation.

\[
\text{generateT lib n } = \text{unfoldT lib n 0}
\]

\[
\text{unfoldT } _n 1 \ k = [V \ k]
\]

\[
\text{unfoldT lib n k } = [F \ \text{op l r} \mid \ i \leftarrow [1..n-1],\ l \leftarrow \text{unfoldT lib i k},\ r \leftarrow \text{unfoldT lib (n-i) (k+i)},\ \text{op} \leftarrow \text{lib}]
\]
For later use, we will also define the dual fold operation parameterized by a function \( f \) describing action on the leaves and a function \( g \) describing action on the internal nodes.

\[
\begin{align*}
\text{foldT}_g \cdot \text{g} \cdot (V \cdot i) &= \text{g} \cdot i \\
\text{foldT}_f \cdot \text{g} \cdot (F \cdot i \cdot l \cdot r) &= \\
&= f \cdot i \cdot (\text{foldT}_f \cdot \text{g} \cdot l) \cdot (\text{foldT}_f \cdot \text{g} \cdot r)
\end{align*}
\]

The \text{foldT} operation will be used later in the synthesis process - for things like boolean evaluation. A simpler use would be to compute the size of a formula as follows:

\[
\text{fsize} \cdot t = \text{foldT}_f \cdot \text{g} \cdot t \quad \text{where}
\begin{align*}
\text{g} \cdot i &= 0 \\
\text{f} \cdot l \cdot r &= 1 + l + r
\end{align*}
\]

We will use \text{foldT} to decode the constants and variables occurring in the result:

\[
\begin{align*}
\text{decodeV} \cdot nvars \cdot \text{i} \cdot = \text{V} \cdot (\text{decode_var} \cdot nvars \cdot (\text{is!i})) \\
\text{decodeF} \cdot i \cdot x \cdot y &= F \cdot i \cdot x \cdot y \\
\text{decodeResult} \cdot nvars \cdot (\text{leafDAG},\text{varMap},_)&= \\
&= \text{foldT} \cdot \text{decodeF} \cdot (\text{decodeV} \cdot nvars \cdot \text{varMap}) \cdot \text{leafDAG}
\end{align*}
\]

The following example shows the action of the decoder:

\[
\begin{align*}
> & \text{decodeV} 2 \cdot (\text{array} \cdot (0,1) \cdot [(0,5),(1,3)]) \cdot 0 \\
& \quad \text{V} \cdot 1 \\
> & \text{decodeV} 2 \cdot (\text{array} \cdot (0,1) \cdot [(0,5),(1,3)]) \cdot 1 \\
& \quad \text{V} \cdot 0 \\
> & \text{decodeResult} 2 \cdot ((F \cdot 1 \cdot (\text{V} \cdot 0) \cdot (\text{V} \cdot 1)), \\
& \quad (\text{array} \cdot (0,1) \cdot [(0,5),(1,3)]), 4) \\
& \quad F \cdot 1 \cdot (\text{V} \cdot 1) \cdot (\text{V} \cdot 0)
\end{align*}
\]

We can also use \text{foldT} to generate a human readable string representation of the result (using the \text{opname} function):

\[
\begin{align*}
\text{showT} \cdot nvars \cdot t &= \text{foldT}_f \cdot \text{g} \cdot t \quad \text{where}
\begin{align*}
\text{g} \cdot i &= \\
&= \text{if} \ i < \text{nvars} \\
&\quad \text{then} \ "x"++(\text{show} \ i) \\
&\quad \text{else} \ \text{show} \ (\text{nvars}+1-i) \\
\text{f} \cdot i \cdot l \cdot r &= \text{(opname} \ i)+"("++l++","++r++")"
\end{align*}
\end{align*}
\]

\[
\begin{align*}
> & \text{showT} 2 \cdot (F \cdot 4 \cdot (\text{V} \cdot 0) \cdot (F \cdot 1 \cdot (\text{V} \cdot 1) \cdot (\text{V} \ 0))) \\
& \quad "xor(x0,nor(x1,x0))"
\end{align*}
\]

### 2.4 Assembling the Circuit Synthesizer

**Definition 1** A Leaf-DAG is obtained from an ordered tree by fusing together equal leaves.

Leaf equality in our case means sharing a primary input variable or a constant.

In the next function we build candidate Leaf-DAGs by combining two generators: the inputs-to-occurrences generator `generateVarMap` and the expression tree generator `generateT`. Then we compute their bitstring value with a foldT based boolean formula evaluator. The function is parameterized by a library of logic gates `lib`, the number of primary inputs `nvars` and the maximum number of leaves it can use `maxleaves`:

```haskell
buildAndEvalLeafDAG lib nvars maxleaves =
  [(leafDAG, varMap, eval varMap leafDAG) |
    k ← [1..maxleaves],
    varMap ← generateVarMap k vs,
    leafDAG ← generateT lib k
  ] where
    mask = allOnes nvars
    vs = init_inputs nvars
    eval varMap leafDAG = foldT (opcode mask) (varMap!) leafDAG
```

We are now ready to test if the candidate matches the specification given by the truth table of `n` variables `ttn`.

```haskell
findFirstGood lib nvars maxleaves ttn =
  head [r |
    r ← buildAndEvalLeafDAG lib nvars maxleaves,
    testspec ttn r
  ] where testspec spec (_,_,v) = spec === v
```

> findFirstGood [1] 2 8 1

`(F 1 (F 1 (V 0) (V 1)) (F 1 (V 2) (V 3)),
array (0,3) [(0,5),(1,0),(2,3),(3,0)],1)
```

The final steps of the circuit synthesizer consist in converting to a human readable form the successful first candidate (guaranteed to be minimal as they have been generated ordered by increasing number of nodes).

```haskell
synthesize_from lib nvars maxleaves ttn =
  decodeResult nvars candidate where
    candidate = findFirstGood lib nvars maxleaves ttn
```

```haskell
synthesize_with lib nvars ttn =
  synthesize_from lib nvars (allOnes nvars) ttn
```

The following two functions provide a human readable output:

```haskell
syn lib nvars ttn = (show ttn) ++ ":" ++
  (showT nvars (synthesize_with lib nvars ttn))
```

```haskell
synall lib nvars = map (syn lib nvars) [0..(allOnes nvars)]
```

The next example shows a minimal circuit for the 2 variable boolean function with truth table 6 (xor) in terms of the library with opcodes in [0] i.e. containing only the operator nand. Note that codes for functions represent their truth tables i.e. 6 stands for [0,1,1,0].
The following examples show circuits synthesized, in terms of a few different libraries, for the 3 argument function if-then-else (corresponding to truth table 83 i.e. [0,1,0,1,0,0,1,1]). As this function is the building block of boolean circuit representations like Binary Decision Diagrams, having perfect minimal circuits for it in terms of a given library has clearly practical value. The reader might notice that it is quite unlikely to come up intuitively with some of these synthesized circuits.

3 Related work

We refer to [3] for general information on circuit design.

The use of functional programming as a hardware description tool goes back as early as [4]. Tools like Hydra, Lava and Wired [5] have shown that various design concepts can be can be elegantly embedded in Haskell [6-8].

Exact circuit synthesis has been a recurring topic of interest in circuit design, complexity theory, boolean logic, combinatorics and graph theory for more than half a century [1, 9-14]. In [15, 2] a Prolog-based exact synthesis algorithm is used to compare expressiveness of various gate libraries in combinational logic. In [16] a similar algorithm is used for reversible circuits. Synthesis of reconfigurable logic is covered in [17].

References

7. Claessen, K., Pace, G.: Embedded hardware description languages: Exploring the design space. Hardware Design and Functional Languages (HFL'07), Braga, Portugal (2007)


