Circuit Morphing: Declarative Modeling of Reconfigurable Combinational Logic

Paul Tarau[1] and Brenda Luderman[2]

[1] Department of Computer Science and Engineering
University of North Texas
Denton, Texas
tarau@cs.unt.edu
[2] ACES CAD
Lewisville, Texas, USA
brenda.luderman@gmail.com

Abstract. Using a simple reconfigurable logic gate that combines an ITE gate and a 1-bit memory cell we devise a mechanism for synthesizing fine grained circuits that overlap multiple logic functions.
A declarative model of the approach, including an exact synthesizer for small circuits, is provided as a literate Haskell program (code available at `http://logic.csci.unt.edu/tarau/research/2009/fsyn.hs`).
Potential applications include new processor architectures supporting fine grained massively parallel execution.
Keywords: *reconfigurable combinational logic, circuit modeling in Haskell, exact combinational logic synthesis*

1 Introduction

As processor clock frequency and power consumption limitations are harder and harder to circumvent, designs diverging more radically from the von Neumann architecture, questioning the strong separation of processing and memory elements, are becoming relevant as foundations for the next generation hardware [1, 2].

In particular, various *reconfigurable* or even *evolvable* hardware models promising scalable distribution of parallel processing tasks have emerged [3–5].

Typical reconfigurable hardware implementations separate the memory and combinational component through the use of *look-up tables* (LUTs) where arbitrary logic functions are performed by looking up their values in fast memory [6] grouped usually in blocks of SRAM. As a result, they are also subject to the von Neumann bottleneck, face complex routing issues, and typically perform orders of magnitude slower, for identical logic functions, than ASIC designs, often reversing the speed-up provided by mapping specialized algorithms to dedicated hardware.

This suggest exploring unconventional reconfigurable hardware designs closely integrating the memory elements used to select configurations with the combinational logic providing the actual functions.

This paper models, using a Haskell program, the behavior of a reconfigurable computational element combining two of the simplest and most widely used circuits: an if-then-else gate (usually called a MUX in circuit design) and a 1-bit memory element. We first show some surprising properties derived from the fact that the Else branch, connected to a memory cell keeps the value of the previous output:

- the circuit emulates conjunction and implication gates
- the circuit can be configured using its 2 input wires to select one or the other of the gates
- the circuit can be used as a 1-bit memory to be read or written using the same 2 input wires

Next, we build a simple, special purpose exact synthesizer that generates optimal small circuits built with our gate. This enables efficient alternative configurations sharing the same gates through a simple reconfiguration algorithm alternating programming and function steps. Finally we explore efficient transistor-level implementations for our circuit - a key element for practical uses.

The paper is organized as follows:

Section 2 introduces a simple reconfigurable gate usable both for combinational logic and as a 1-bit memory element. Section 3 describes an exact combinational circuit synthesizer. Section 4 introduces the two-step reconfiguration algorithm and gives examples of circuits using the technique. Section 5 describes transistor-level implementations of the our reconfigurable gate. Sections 6 and 7 discuss related work, future work and conclusions. The code in the paper, embedded in a literate programming LaTeX file, is entirely self contained and has been tested under `GHC 6.4.10`. It uses a few standard libraries:

```
import Data.List
import Data.Bits
import Data.Array
```

2 A Simple Reconfigurable Circuit

We start with a circuit representing the if-then-else function (called ITE subsequently), defined as follows:

```
if_then_else 0 _ z = z
if_then_else 1 y _ = y
```

resulting in

```
> [([x,y,z],if_then_else x y z)|
      x←[0,1],y←[0,1],z←[0,1]]
  [([0,0,0],0),
   ([0,0,1],1),
   ([0,1,0],0),
   ([0,1,1],1),
```

```
   ([1,0,0],0),
   ([1,0,1],0),
   ([1,1,0],1),
   ([1,1,1],1)]
```

Similarly, we will also define conjunction and implication:

```
conj 1 1 = 1
conj _ _ = 0

impl 1 0 = 0
impl _ _ = 1
```

The following identities holds:

Proposition 1
conj x y = if_then_else x y 0
impl x y = if_then_else x y 1

The truth tables for both equivalences prove the proposition:

```
> [([x,y],if_then_else x y 0,conj x y)| x←[0,1],y←[0,1]]
[([0,0],0,0),
 ([0,1],0,0),
 ([1,0],0,0),
 ([1,1],1,1)]
> [([x,y],if_then_else x y 1,impl x y)| x←[0,1],y←[0,1]]
[([0,0],1,1),
 ([0,1],1,1),
 ([1,0],0,0),
 ([1,1],1,1)]
```

The ability to emulate the two, arguably most natural logic operations (conjunction and implication), suggests adding a 1-bit memory to be used as control signal that reconfigures a circuit to behave as either a conjunction or an implication. Given that the library consisting in these two gates and functions 0,1 is *universal* (see, for instance, [7]), in combination with a circuit synthesizer, we will derive the ability to *morph* a set of such gates into arbitrary boolean functions.

Given that the `Else` *input of the* `ITE` *gate controls the reconfiguration of the circuit, it makes sense to attach to it a link to a memory cell that stores the result of the evaluation at time* `t` *of the function and provides at time* `t+1` *the* `Else` *input of the* `ITE` *gate.*

We model this in Haskell using two wrappers (`M` and `O` that indicate, respectively, the use of a bit either to write to a memory cell or as the output value of the circuit.

```
newtype M=M Int deriving (Eq,Ord,Show,Read)
newtype O=O Int deriving (Eq,Ord,Show,Read)
```

The transitions are now modeled with the function `next`, mapping a signal from the previous state of the memory and two inputs `x` and `y` to a pair consisting in the new state of the memory and the output value `r`.

```
next :: M→O→O→(M,O)
next (M 0) (O x) (O y)=(M r,O r) where r=conj x y   -- conj becomes ⇒
next (M 1) (O x) (O y)=(M r,O r) where r=impl x y  -- ⇒ becomes conj
```

We can now model the evolution of the resulting sequential circuit as a function taking as inputs two lists of bits and returning a list of bits while threading the state of the memory between transitions, as follows:

```
run :: M → [Int] → [Int] → (M, [Int])
run m is js = runWith m is js [] where
  runWith :: M → [Int] → [Int] → [Int] → (M, [Int])
  runWith (M m) [] [] os = (M m,os)
  runWith (M m) (i:is) (j:js) os = ((M m'),(o:os')) where
    (M n,O o)=next (M m) (O i) (O j)
    ((M m'),os')=runWith (M n) is js os
```

Running the simulation with successive inputs at times t_0, t_0+1, t_0+2, t_0+3 gives a list of 4 outputs:

```
> run (M 0) [1,1,0,1] [0,1,0,1]
(M 1,[0,1,1,1])
```

Proposition 2 *If the circuit implements* conjunction *at time* `t` *and it succeeds, then it will implement* implication *at time* `t+1`. *If the circuit implements* implication *at time* `t` *and it fails, then it will implement conjunction at time* `t+1`. *Otherwise, the circuit will implement at time* `t+1` *the same gate as at time* `t`.

The following truth table shows the transition from `m` to `m'` under the effect of inputs `x` and `y` (as well as the output `z`).

```
> [((x,y),M m,run (M m) [x] [y])|x←[0,1],y←[0,1],m←[0,1]]
-- x,y,   m    m' z --
[((0,0),M 0,(M 0,[0])),
 ((0,0),M 1,(M 1,[1])),
 ((0,1),M 0,(M 0,[0])),
 ((0,1),M 1,(M 1,[1])),
 ((1,0),M 0,(M 0,[0])),
 ((1,0),M 1,(M 0,[0])),
 ((1,1),M 0,(M 1,[1])),
 ((1,1),M 1,(M 1,[1]))]
```

The proposition is proven by observing that transition from `m=0` (conjunction) occurs exactly when `x=1, y=1` i.e. the conjunction succeeds, and transition from `m=1` (implication) occurs exactly when the implication fails for x=1, y=0. We can describe this property by defining:

```
-- morph into a conjunction at the next tick
toAnd x = run (M x) [1] [0]

-- morph into an implication at the next tick
toImpl x = run (M x) [1] [1]
```

We will now show that besides morphing into conjunction or implication gates, our circuit is also usable as a 1-bit memory cell. We define:

```
toRead x = run (M x) [0] [0]

toWrite m x = run (M m) [1] [x]
```

The following holds:

Proposition 3 *If the circuit holds bit* `m` *at time* `t`*, then sending at time* `t` *the signals* `x=0,y=0` *reads* `m` *nondestructively i.e the state of the 1-bit memory will be the same at time* `t+1` *and the output signal will be* `m`*. Independently of the state of the 1-bit memory at time* `t`*, sending the signal* `x=1,y=m` *results in the memory holding bit* `m` *at time* `t+1`*.*

The first part of the proposition is proven by observing that:

```
> toRead 0
(M 0,[0])
> toRead 1
(M 1,[1])
```

The second part of the proposition is proven by observing that:

```
> toWrite 0 0
(M 0,[0])
> toWrite 0 1
(M 1,[1])
> toWrite 1 0
(M 0,[0])
> toWrite 1 1
(M 1,[1])
```

i.e. that the second argument controls the state of the 1-bit memory independently of the previous state of the memory given by the first argument of the function `toWrite`.

Definition 1 *We will call* `ITE+1Bit gate` *the reconfigurable circuit consisting in an if-then-else gate and a 1-bit memory cell connected to its* `Else` *wire.*

Its ability to work as both combinational and memory cells suggests a two step mechanism to implement arbitrary logic.

- at time $t = 2k$ a signal is sent to program the desired function using the `toWrite` operation
- at time $t = 2k + 1$ the combinational function (conjunction or implication) is executed

Clearly, by iterating over these steps for $k = 0, 1, \ldots$, multiple circuits[3] can share silicon in exchange for running at about half the speed of a dedicated circuit. This

[3] Assumed having about the same size in terms of conjunction and implication gates.

suggests using an exact synthesizer for building a library of such circuits in an optimal way, i.e. by minimizing the total number of conjunction and implication gates for each circuit.

We will prototype here a simple exact synthesizer written in Haskell along the lines of [7, 8].

3 Exact Combinational Circuit Synthesis

To make the paper self-contained we start by reviewing a mechanism for fast boolean evaluation, following [9] and [8].

3.1 Evaluation of Boolean Functions with Bitvector Operations

The boolean evaluation mechanism uses integer encodings of 2^n bits for each boolean variable $x_0, \ldots, x_{n-1}$. Bitvector operations are used to evaluate all value combinations at once.

Proposition 4 *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:*

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

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`.

```
allOnes nvars = 2^2^nvars - 1
```

Next we define a function providing the integer representation of the `k-th` boolean variable (out of `n`).

```
var_n n k = var_mn (allOnes n) n k

var_mn mask n k = mask `div` (2^(2^(n-k-1))+1)
```

We have used in `var_n` an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of [9]. 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.

3.2 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).

```
encode_var m n k | k==n = m
encode_var m n k | k==n+1 = 0
encode_var m n k = 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:

```
init_inputs n =
  0:m:(map (encode_var m n) [0..n-1]) where
  m=allOnes n


>init_inputs 3
  [0,15,3,5]
>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.

```
decode_var nvars v | v==(allOnes nvars) = nvars
decode_var nvars 0 = nvars+1
decode_var nvars v = head
  [k|k←[0..nvars-1],(encode_var m nvars k)==v]
    where m=allOnes nvars


>map (decode_var 2) (init_inputs 2)
  [3,2,0,1]
>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.

```
bindings 0 us = [[]]
bindings n us =
  [zs|ys←bindings (n-1) us,zs←map (:ys) us]
```

```
>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.

```
generateVarMap occs vs  =
  map (listArray (0,occs-1)) (bindings occs vs)


>generateVarMap 2 [3,5]
  [array (0,1) [(0,3),(1,3)],
   array (0,1) [(0,5),(1,3)],
   array (0,1) [(0,3),(1,5)],
   array (0,1) [(0,5),(1,5)]]
```

3.3 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`.

```
data T a = V a | F a (T a) (T a) deriving (Show, Eq)
```

Generating all trees is a variant of an `unfold` operation (an *anamorphism*).

```
generateT lib n = unfoldT lib n 0


unfoldT _ 1 k = [V k]
unfoldT lib n k = [F op l r |
  i←[1..n-1],
  l ← unfoldT lib i k,
  r ← unfoldT lib (n-i) (k+i),
  op←lib]
```

For later use, we will also define the dual `fold` operation (a *catamorphism*) parameterized by a function `f` describing action on the leaves and a function `g` describing action on the internal nodes.

```
foldT _ g (V i) = g i
foldT f g (F i l r) =
  f i (foldT f g l) (foldT f g r)
```

This catamorphism 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:

```
fsize t = foldT f g t where
   g _ = 0
   f _ l r = 1+l+r
```

A first use of `foldT` will be to decode the constants and variables occurring in the result:

```
decodeV nvars is i = V (decode_var nvars (is!i))

decodeF i x y = F i x y

decodeResult nvars (leafDAG,varMap,_) =
  foldT decodeF (decodeV nvars varMap) leafDAG
```

The following example shows the action of the decoder:

```
>decodeV 2 (array (0,1) [(0,5),(1,3)]) 0
  V 1
>decodeV 2 (array (0,1) [(0,5),(1,3)]) 1
  V 0
>decodeResult 2 ((F 1 (V 0) (V 1)),
            (array (0,1) [(0,5),(1,3)]), 4)
  F 1 (V 1) (V 0)
```

We can also use `foldT` to generate a human readable string representation of the result (using the `opname` function given in Appendix):

```
showT nvars t = foldT f g t where
  g i =
     if i<nvars
       then "x"++(show i)
       else show (nvars+1-i)
  f i l r =(opname i)++"("++l++","++r++")"

> showT 2 (F 4 (V 0) (F 1 (V 1) (V 0)))
  "xor(x0,nor(x1,x0))"
```

3.4 Assembling the Circuit Synthesizer

Definition 2 *A Leaf-DAG generalizes 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`:

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

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

```
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).

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

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

The following two functions provide a human readable output:

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

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]`.

```
> syn [0] 2 6
  "6:nand(nand(x0,nand(x1,1)),nand(x1,nand(x0,1)))"
```

The following examples show circuits synthesized, in terms of a few different libraries, for the 3 argument function `if-the-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.

```
> syn symops 3 83
  "83:nor(nor(x2,x0),nor(x1,nor(x0,0)))"
>syn asymops 3 83
  "83:impl(impl(x2,x0),less(x1,impl(x0,0)))"
```

We refer to the `Appendix` for a few details, related to the bitvector operations on various boolean functions used in the libraries, as well as a few tests.

4 The Reconfiguration Algorithm

We can specialize our exact synthesizer to the library consisting of implication and conjunction with

`rsyn nvars tt = syn impl_and nvars tt`

working as follows

```
> rsyn 2 6
"6:impl(impl(x0,x1),and(x1,impl(x0,0)))"
*> rsyn 2 9
"9:and(impl(x0,x1),impl(x1,x0))"
```

Clearly the two circuits could share the same gates provided that they can be morphed selectively into implications or conjunctions.

This suggests the following algorithm, for sharing multiple combinational circuits using a library consisting solely of our `ITE+1Bit` *reconfigurable circuit.*

Let $C_0, C1, ...C_k, ..$ a set of combinational circuits of size at most `n` each.

- using an exact synthesizer if `n` is small, or conventional minimization techniques otherwise, express each C_k in terms of primary inputs, constant functions 0,1 and a library consisting of `conjunction` and `implication` gates
- encode the configuration of each C_k as a bitvector with `0` indicating a `conjunction` and `1` indicting an `implication` gate and store it in an `n`-bit memory built with our `ITE+1Bit` gates used in memory mode - let's call this the *controller*
- connect each output bit `m` of the controller to the inputs of the corresponding combinational element (knowing that `x=1,y=m` configures it either as a conjunction if m=0 or implication if m=1)
- iterate as follows:
  - at time 2*k write the stored circuit C_k to program the n combinational elements selectively as conjunctions or implications
  - at time 2*k+1 run the combinational circuit providing the function defined by C_k

Note that the resulting architecture is efficient in two ways:

- its gates are shared by the $C_0, C1, ...C_k, ..$ distinct boolean functions
- no additional wires are needed for the reconfiguration steps at times `2k` as this process is achieved using the same 2 input wires for each gate that are also used for their combinational functions

5 Building the ITE+1Bit Circuit

We will now turn to use the exact synthesizer defined in section 3 to provide optimal designs for the combined ITE and 1-bit circuit.

It is known that `nand` and `nor` gates can be built without using pass-transistor logic as 4-transistor gates. To obtain an optimal ITE (a 3 variable function with its truth table encoded as 83), we can run the synthesizer as follows:

```
> syn [0] 3 83
"83:nand(nand(x1,x0),nand(x2,nand(x0,1)))"

> syn [1] 3 83
"83:nor(nor(x2,x0),nor(x1,nor(x0,0)))"
```

Observing that `nand(x0,1)` and `nor(x0,0)` are equivalent to logical negation (and can therefore be implemented with 2 transistor inverters), we obtain in each case 3*4+2=14 transistor designs. Fig. 1 shows a `14T` nand-based implementation.

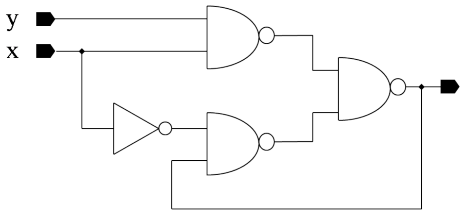

Fig. 1: 14T circuit for ITE+1Bit gate

However, by using Pass Transistor Logic (PTL) a design built directly around a `2T` MUX is possible, as shown in Fig 2.

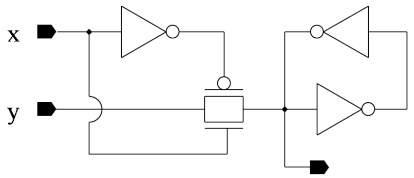

Fig. 2: 8T circuit for ITE+1Bit gate

This shows that it is likely that the ITE+1Bit circuit is can be competitively implemented in conventional ASIC designs and as such it could provide an interesting alternative to LUT-based FPGA architectures.

6 Related work

We refer to [10] for general information on circuit design. [6, 2] provide overviews of trends and achievements in reconfigurable hardware.

The use of functional programming as a hardware description tool goes back as early as [11]. Tools like Hydra, Lava and Wired [12] have have shown that

various design concepts can be can be elegantly embedded [13–15] in Haskell. However, we are not aware of modeling of reconfigurable circuits and combinatorial search based exact synthesis algorithms as described in this paper.

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 [9, 16–21]. Synthesis of reconfigurable logic is covered in [22].

The use of cyclical circuits for combinational logic has been subject of recent work in [23], although in this case the focus is simplification of combinational circuits rather than encapsulation of state as in our ITE+1Bit circuit.

7 Conclusion and Future Work

We have described a surprisingly simple computational element requiring as little as 8 transistors in PTL, that, in combination with an exact circuit synthesizer provides an optimal universal building block for reconfigurable logic. Future work will focus on synthesis of a practical library of circuits based on it and detailed empirical study through simulation with SPICE of its scalability and electrical behavior. More work is also needed on the implementation of the reconfiguration algorithm discussed in the paper and its possible integration with Haskell based circuit and layout design tools like Wired. Further down the road, beyond the next decade, the advent of alternative circuit implementations might involve radical departures from traditional CMOS processes, ranging from optical and quantum computing to biological or molecular techniques [24]. In some of these fields, manufacturability is likely to limit the variety of gate-level building blocks. This also implies that the resulting libraries might have to use as few as possible gates, possibly involving unconventional, yet unknown processes. In this case, architectures built around a simple universal computational element like our ITE+1Bit circuit could be of significant interest.

References

1. Becker, J., Hartenstein, R.: Configware and morphware going mainstream. Journal of Systems Architecture **49**(4-6) (2003) 127–142
2. Hartenstein, R.: A decade of reconfigurable computing: a visionary retrospective. In: DATE '01: Proceedings of the conference on Design, automation and test in Europe, Piscataway, NJ, USA, IEEE Press (2001) 642–649
3. Sekanina, L.: Evolutionary design of gate-level polymorphic digital circuits. In Rothlauf, F., Branke, J., Cagnoni, S., Corne, D.W., Drechsler, R., Jin, Y., Machado, P., Marchiori, E., Romero, J., Smith, G.D., Squillero, G., eds.: EvoWorkshops. Volume 3449 of Lecture Notes in Computer Science., Springer (2005) 185–194
4. Lee, M., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F., Filho, E., Alves, V.: Design and implementation of the MorphoSys reconfigurable computing processor. The Journal of VLSI Signal Processing **24**(2) (2000) 147–164
5. Zhang, Q., Chamberlain, R., Indeck, R., West, B., White, J.: Massively parallel data mining using reconfigurable hardware: Approximate string matching. In: Proc. of Workshop on Massively Parallel Processing. (2004)

6. Compton, K., Hauck, S.: An introduction to reconfigurable computing. IEEE Computer, Apr (2000)
7. Tarau, P., Luderman, B.: Exact combinational logic synthesis and non-standard circuit design. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA, ACM (2008) 179–188
8. Tarau, P., Luderman, B.: A Logic Programming Framework for Combinational Circuit Synthesis. In: 23rd International Conference on Logic Programming (ICLP), LNCS 4670, Porto, Portugal, Springer (September 2007) 180–194
9. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
10. Rabaey, J., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits: A Design Perspective. Prentice Hall (2003)
11. O'Donnell, J.: Hardware description with recursion equations In Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications, pages 363–382. NorthHolland, April 1987.
12. Axelsson, E.: Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction PhD Thesis, Chalmers, 2008.
13. Claessen, K., Sheeran, M., Singh, S.: Functional hardware description in Lava. In: The Fun of Programming. Cornerstones of Computing. Palgrave (2003) 151–176
14. Claessen, K., Pace, G.: Embedded hardware description languages: Exploring the design space. Hardware Design and Functional Languages (HFL07), Braga, Portugal (2007)
15. Claessen, K., Seger, C., Sheeran, M., Shriver, E., Swierstra, W.: High-level architectural modelling for early estimation of power and performance. In: Proc. of Hardware Description using Functional Languages (HFL). (March 2009)
16. Shannon, C.E.: Claude Elwood Shannon: collected papers. IEEE Press, Piscataway, NJ, USA (1993)
17. Oettinger, A.G., Aiken, H.H.: Retiring computer pioneer. Commun. ACM **5**(6) (1962) 298–299
18. Davies, D.W.: Switching functions of three variables. Trans. Inst. Radio Engineers **6**(4) (1957) 265–275
19. Culliney, J.N., Young, M.H., Nakagawa, T., Muroga, S.: Results of the Synthesis of Optimal Networks of AND and OR Gates for Four-Variable Switching Functions. IEEE Trans. Computers **28**(1) (1979) 76–85
20. Lai, H.C., Muroga, S.: Logic Networks with a Minimum Number of NOR(NAND) Gates for Parity Functions of Variables. IEEE Trans. Computers **36**(2) (1987) 157–166
21. Drechsler, R., Gunther, W.: Exact Circuit Synthesis. In: International Workshop on Logic Synthesis. (1998)
22. Mishchenko, A., Sasao, T.: Encoding of Boolean functions and its application to LUT cascade synthesis. In: International Workshop on Logic Synthesis. (2002)
23. Riedel, M.: Cyclic Combinational Circuits. In: Ph.D. Dissertation, Caltech. (2004)
24. Mira, J., Álvarez, J.R., eds.: Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005, Las Palmas, Canary Islands, Spain, June 15-18, 2005, Proceedings, Part II. In Mira, J., Álvarez, J.R., eds.: IWINAC (2). Volume 3562 of Lecture Notes in Computer Science., Springer (2005)

Appendix

We list here some auxiliary functions making the paper paper fully self contained.

Bitvector Boolean Operation Definitions

```
type Nat=Integer

nand_ :: Nat→Nat→Nat→Nat
nor_ :: Nat→Nat→Nat→Nat
impl_ :: Nat→Nat→Nat→Nat
less_ :: Nat→Nat→Nat→Nat
and_ :: Nat→Nat→Nat→Nat

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

```
-- operation codes
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))

-- operation names
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]
```

Tests for the Circuit Synthesizer

```
t0 = findFirstGood symops 3 8 71
t1 = syn asymops 3 71
t2 = mapM_ print (synall asymops 2)
```