Symbolic Modeling of a Universal Reconfigurable Logic Gate and its Applications to Circuit Synthesis

Paul Tarau[1] Brenda Luderman[2]

University of North Texas[1]

Texas Instruments Inc.[2]

RACS'2012, System Software I, Wed, Oct 24, 9:50am-11:55am

Outline

- motivation: designs diverging radically from the von Neumann architecture, questioning the strong separation of processing and memory elements, are becoming more and more relevant
- $\Rightarrow$ unify the combinational and memory component as a single, uniformly structured, reconfigurable building block
- we describe a simple reconfigurable logic gate that emulates conjunction, implication and a 1-bit memory cell
- we devise a mechanism for synthesizing fine grained circuits that overlap multiple logic functions
- an exact synthesizer for small circuits: a literate Haskell program, code at `http://logic.csci.unt.edu/tarau/research/2012/fsyn.hs`

Our Reconfigurable Circuit Element

- an *If-Then-Else* (ITE) gate
- a *1-bit memory element*
- some surprising properties can be 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

Emulating Conjunction and Implication

The following identities hold:

Proposition

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

- 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
- it also provides at time `t+1` the `Else` input of the `ITE` gate

The one-step Transformation of the Circuit

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

next :: M → O → O → (M,O)

next (M 0) (O x) (O y)=(M r,O r) where
  r = conj x y   -- conjunction becomes implication
next (M 1) (O x) (O y)=(M r,O r) where
  r = impl x y   -- implication becomes conjunction
```

Modeling the Evolution of the Sequential Circuit

`run` is a function taking as inputs two lists of bits and returning a list of bits while threading the state of the memory between transitions

```
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

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

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 Truth Table of the Transition

- the transition from `m` to `m'` under the effect of inputs `x` and `y`
- 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 Dual use as Combinational and Sequential Circuit

- transition from `m=0` (conjunction) occurs exactly when `x=1`, `y=1` i.e. the conjunction succeeds,
- transition from `m=1` (implication) occurs exactly when the implication fails for x=1, y=0

morph into an implication at the next tick

```
toImpl x = run (M x) [1] [1]
```

morph into a conjunction at the next tick

```
toAnd x = run (M x) [1] [0]
```

- besides morphing into conjunction or implication gates, our circuit is also usable as a 1-bit memory cell

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

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

Characterizing the Behavior of the Circuit

Proposition

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`*.*

Definition

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.*

A possible Reconfiguration Protocol

- at time $t = 2k$ a signal is sent to program the desired combinational function using the `toWrite` operation
- at time $t = 2k + 1$ the combinational function (conjunction or implication) is executed
- by iterating over these steps for $k = 0, 1, \ldots$, multiple circuits can share silicon in exchange for running at about half the speed of a dedicated circuit

The Bitvector Representation of Boolean Variables

Proposition (Knuth)

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 $x_1 = 5 = [0, 1, 0, 1]$.

Generating Formulas as Trees with `unfold`

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

- we generate trees with library operations marking internal nodes of type `F` and primary inputs marking the leaves of type `V`
- generating all trees is specialization of an `unfold` operation

```
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]
```

Evaluating Candidate Formulas by Specializing `fold`

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

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

>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)
```

Assembling the Circuit Synthesizer

- we first 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`

Using the Synthesizer

a minimal circuit for the `2` variable boolean function with truth table `6=[0,1,1,0]` (`xor`) containing only the operator `nand`=[0]

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

other examples:

```
> 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)))"
```

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.

Sharing Combinational Logic based our `ITE+1Bit` Circuit

Let $C_0, C1, \ldots C_k, ..$ a set of combinational circuits (of size at most `n`).

- 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

Efficient Sharing of Combinational Logic

the resulting architecture is efficient in two ways:

- its gates are shared by the $C_0, C1, \ldots C_k, ..$ 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

Discussion

- conjunction and implication can be seen as a Galois connection where the two mappings are described by $F(x) = a \wedge x$ and $G(y) = a \Rightarrow y$
- implication is the upper adjoint of conjunction
- interesting duality between $\wedge$, $\Rightarrow$ on one hand, and `Half-XOR` (defined as $\neg x \wedge y$) and disjunction, on the other hand
- a dual of our ITE+1Bit can obtained by linking the `Then` output of the `ITE` gate to a 1-bit memory cell
- $\Rightarrow$ these algebraic properties might be relevant for circuit synthesis and minimization tools

Conclusion

- we have designed 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
- we have described an exact synthesis algorithm that provides a library of optimal small circuits using conjunction and implication
- 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

Acknowledgment: Paul Tarau's work has been supported by NSF research grant 1018172.