Model Checking Embedded Synchronous Hardware Compilers

Gordon Pace
University of Malta
(joint work with Koen Claessen)

Theory Days, Voore, Estonia
September 2006

Embedded Languages

- A meta-language for free
  - Generation
  - Analysis
  - Manipulation
  - Semantics
- Tools for free

Regular Expressions in Haskell

```
data RegExp = Input Char
            | Plus RegExp
            | RegExp :+: RegExp
            | RegExp :>: RegExp
```

When embedding a language as a data structure, nice datatypes are *very* desirable.

Manipulation and Generation

```
any :: [RegExp] -> RegExp
any [e] = e
any (e:es) = e :+: any es

getString :: [Char] -> RegExp
getString [c] = Input c
getString (c:cs) = Input c :>: getString cs

permutations :: [Char] -> RegExp
permutations cs = any [ getString cs' | cs' <- perms cs ]
```

So what?

Well, let me first say something about Lava...

Lava in a Nutshell

- It is an embedded language
- It is embedded in Haskell
- It is a synchronous hardware description language

Combinational circuits

```
majority :: (Sig, Sig, Sig) -> Sig
majority (x,y,z) = or2(xy, or2(yz, xz))
  where
    xy = and2(x,y)
    yz = and2(y,z)
    xz = and2(x,z)
```

Combinational circuits (2)

```
majority :: (Sig, Sig, Sig) -> Sig
majority (x,y,z) = or2(xy, or2(yz, xz))
  where
    xy = and2(x,y)
    yz = and2(y,z)
    xz = and2(x,z)
```

```
Main> simulate majority (high, low, low)
low
```

Combinational circuits (3)

```
majority :: (Sig, Sig, Sig) -> Sig
majority (x,y,z) = or2(xy, or2(yz, xz))
  where
    xy = and2(x,y)
    yz = and2(y,z)
    xz = and2(x,z)
```

```
Main> writeVhdl "majority2" majority
Writing to file "majority2.vhd" ... Done.
```

Generic Combinational circuits

```
exactlyZero :: [Sig] -> Sig
exactlyZero sigs = inv (orl sigs)

exactlyOne :: [Sig] -> Sig
exactlyOne [] = low
exactlyOne (x:xs) =  mux(x, (one, zero))
  where
    one  = exactlyOne xs
    zero = exactlyZero xs

twoOrMore sigs = nor2(exactlyZero sigs, exactlyOne sigs)
```

Parametrized combinational circuits

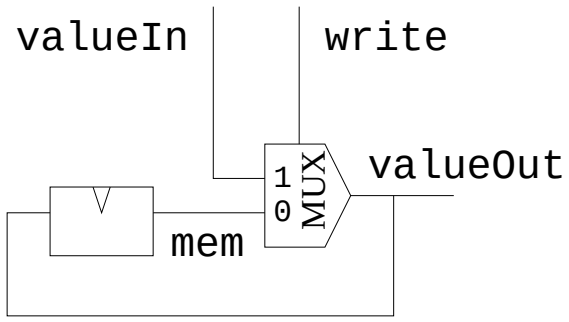
```
exactly :: Int -> [Sig] -> Sig
exactly 0 sigs = inv (orl sigs)
exactly n []   = low
exactly n (s:sigs) = mux (s, (nLeft, n_1Left))
  where
    nLeft   = exactly n sigs
    n_1Left = exactly (n-1) sigs

twoOrMore sigs = nor2(exactly 0 sigs, exactly 1 sigs)
```

```
Main> simulate (exactly 1) [low, low, high, low]
high
```

Delays in Lava...

delay init value

```
register (write, valueIn) = valueOut
  where
    valueOut = mux (write, (memory, valueIn))
    memory = delay low valueOut
```



Sequential simulation

```
always :: Sig -> Sig
always sig = now
  where
    before = delay high now
    now    = and2(before, sig)
```

Sequential simulation

```
always :: Sig -> Sig
always sig = now
  where
    before = delay high now
    now    = and2(before, sig)
```

```
Main> simulateSeq always [high, high, high, low, high]
[high,high,high,low,low]
```

Verification

```
compareMajorities (x,y,z) = ok
  where
    ok = majority (x,y,z) <==> twoOrMore [x,y,z]
```

Verification

```
compareMajorities (x,y,z) = ok
  where
    ok = majority (x,y,z) <==> twoOrMore [x,y,z]
```

```
Main> vis compareMajorities
Vis: ... (t=0.0)
Valid.
```

Verification (2)

```
another (w,x,y,z) = ok
  where
    ok = exactly 2 [w,x,y,z] ==> majority (x,y,z)
```

Verification (2)

```
another (w,x,y,z) = ok
  where
    ok = exactly 2 [w,x,y,z] ==> majority (x,y,z)
```

```
Main> verify another
Proving: ... Falsifiable.
(high,high,low,low)
```

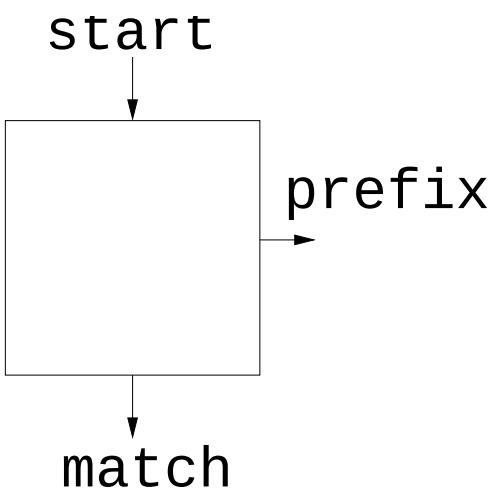
Compiling Embedded Languages

- A hardware compiler is just another parametrised circuit.
- We can use Lava to describe one ...
- and reason about the programs generated ...
- and about the compiler itself.

Compiling Embedded Languages

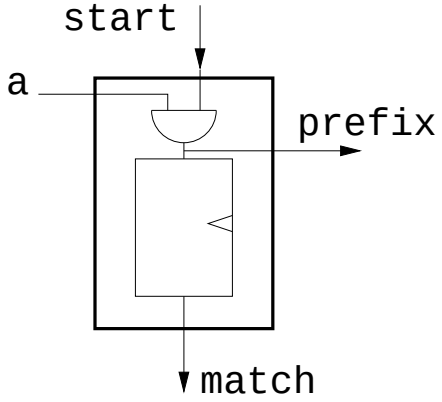
Circuits parametrized by programs ...

```
regexp :: RegExp -> Circuit_RegExp
```

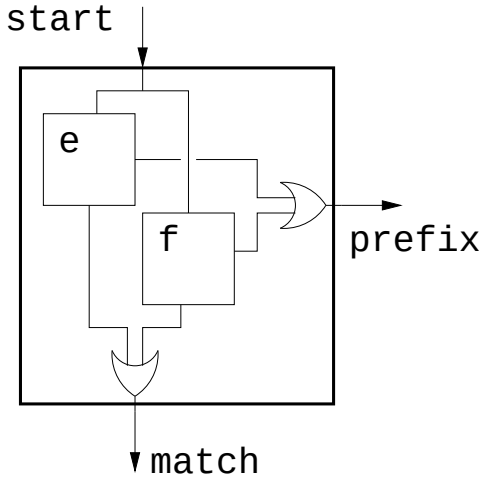



```
type Circuit_RegExp :: Sig -> (Sig, Sig)
```

Compiling Embedded Languages



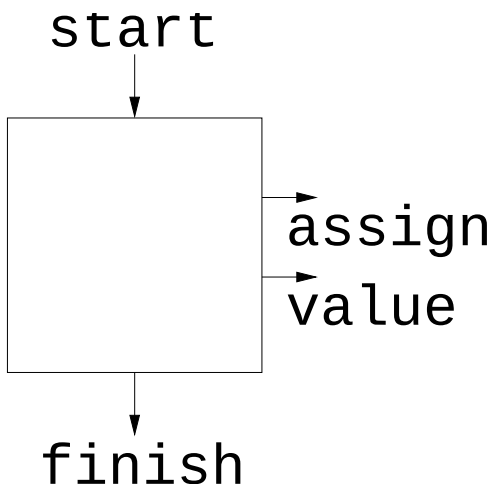

```
regexp (Input x) start = (prefix, match)
  where
    prefix = and2(start, match)
    match  = delay low prefix
```




```
regexp (e :+: f) start = (prefix, match)
  where
    (prefix_e, match_e) = regexp e start
    (prefix_f, match_f) = regexp f start
    prefix = or2(prefix_e, prefix_f)
    match  = or2(match_e, match_f)
```

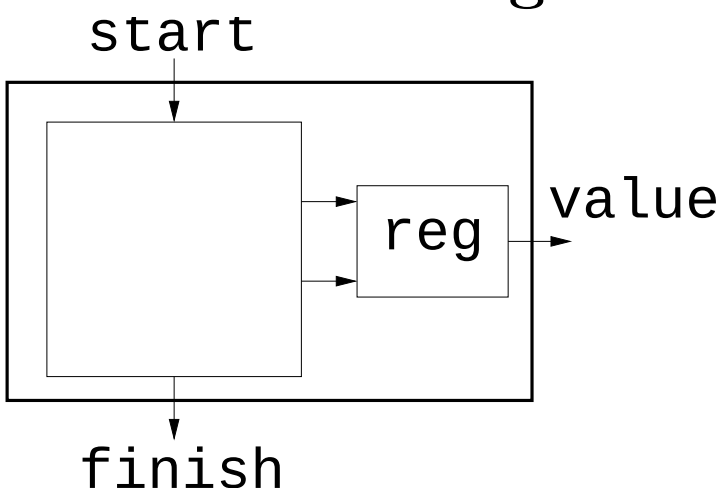
A more complex language

```
data Flash =
    Assign Sig
  | Flash :|| Flash
  | ...
```




```
type flashCircuit = Sig -> (Sig, (Sig,Sig))

flash :: Flash -> Circuit
```

And then use a register ...

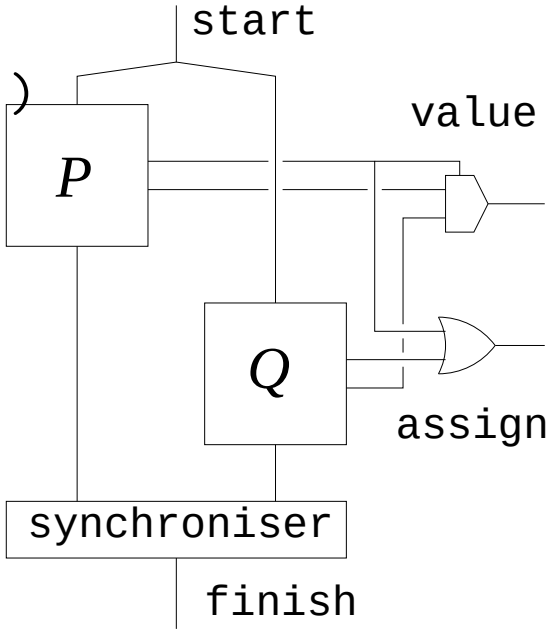



```
compile prg start = (finish, value)
  where
    (finish, (assign, newvalue)) = flash prg start
    value = register (assign, newvalue)
```

Compiling Flash

```
flash (Assign val) start = (finish, (assign, newvalue))
  where
    finish = delay low start
    assign = start
    newvalue = val
```

```
flash (p :|| q) start = (fin, (wr, val))
  where
    (f_p, (w_p, v_p)) = flash p start
    (f_q, (w_q, v_q)) = flash q start
    wr  = or2(w_p, w_q)
    val = mux(w_p, (v_p, v_q))
    fin = synchronise (f_p, f_q)
```



Reasoning about Embedded Languages (1)

Implementation-Observer Verification

By using observers, we can reason about compiled circuits directly.

Reasoning about Embedded Languages (1)

Implementation-Observer Verification

```
(program `satisfies` observer) (start, ins) = ok
  where
    outs = compile (program ins) start
    ok   = observer (start, ins, outs)
```

Reasoning about Embedded Languages (1)

Implementation-Observer Verification

```
(program `satisfies` observer) (start, ins) = ok
  where
    outs = compile (program ins) start
    ok   = observer (start, ins, outs)

majorityFlash (x,y,z) =
  While high (
    While x (
      Assign (y <|> z)
    ) :>
    Assign (y <&> z)
  )
```

Reasoning about Embedded Languages (1)

Implementation-Observer Verification

```
(program `satisfies` observer) (start, ins) = ok
  where
    outs = compile (program ins) start
    ok   = observer (start, ins, outs)

majority' (s, xyz, (f, val)) = ok
  where
    ok = always (s <==> delay high low)
           ==> (majority xyz <==> val)
```

Reasoning about Embedded Languages (1)

Implementation-Observer Verification

```
(program `satisfies` observer) (start, ins) = ok
  where
    outs = compile (program ins) start
    ok   = observer (start, ins, outs)
```

```
Main> lesar (majorityFlash `satisfies` majority')
Lesar: ... (t=0.0)
Valid.
```

Reasoning about Embedded Languages (2)

Verification for Compilation

- Add an observer wire to the compiled circuit which checks compiler semantic preconditions.
- We can model-check that the result really works.
- And by ignoring the observer wire, we get the original compiled circuit.

Reasoning about Embedded Languages (2)

Verification for Compilation

```
flash (p :|| q) start = (fin, (wr, val))
  where
    (f_p, (w_p, v_p))       = flash p start
    (f_q, (w_q, v_q))       = flash q start

    wr  = or2(w_p, w_q)
    val = mux(w_p, (v_p, v_q))

    fin = synchronise (f_p, f_q)
```

Reasoning about Embedded Languages (2)

Verification for Compilation

```
flash (p :|| q) start = (fin, (wr, val), err)
  where
    (f_p, (w_p, v_p), e_p) = flash p start
    (f_q, (w_q, v_q), e_q) = flash q start

    wr  = or2(w_p, w_q)
    val = mux(w_p, (v_p, v_q))

    fin = synchronise (f_p, f_q)

    both = andl[w_p, v_p <=/=> v_q, w_q]
    err = orl[e_p, both, e_q]
```

Reasoning about Embedded Languages (2)

Verification for Compilation

```
noClash prg = inv err
  where
    start = delay high low
    (_,_,err) = flash prg start

up = Assign high                    down = Assign low
clock1 = While high (up   :> down)
clock3 = While high (up   :> Delay :> Delay :>
                    down :> Delay :> Delay)
```

```
Main> lesar (noClash (clock1 :|| clock3))
Lesar: ... (t=0.1)
Valid.
```

Reasoning about Embedded Languages (2)

Verification for Compilation

- Testing methods for functional programs (eg see Quickcheck) can be applied on huge programs
- `error` wires (and components that are used to calculate it) do not appear in the final circuit once verified
- Various constructs have error wires, eg:
  - Loop bodies take time
  - Shared components are only syntactically shared
  - Dependencies in languages with potential combinational cycles (`verify (constructive circuit)`)
  - Data overflow

Reasoning about Embedded Languages (3)

Syntactic Reasoning and Verification of Compilation

Verification of *algebraic laws* is useful so as to:

- increase confidence in the compilation
- allow us to reason (mechanically) syntactically (eg to improve compiled circuit efficiency)

Reasoning about Embedded Languages

Syntactic Reasoning and Verification of Compilation

```
regexp (e :+: f) start = (prefix, match)
  where
    (prefix_e, match_e) = regexp e start
    (prefix_f, match_f) = regexp f start
    prefix = or2(prefix_e, prefix_f)
    match  = or2(match_e, match_f)

plusCircuit (s, p, m) (s1,p1,m1) (s2,p2,m2) = ok
  where
    ok = andl [ s1 <==> s
              , s2 <==> s
              , p  <==> or2(p1,p2)
              , m  <==> or2(m1,m2)
              ]
```

Reasoning about Embedded Languages (3)

Syntactic Reasoning and Verification of Compilation

```
plusCommutative (c,c',c1,c2) = ok
  where
    ok = always (
           andl [ plusCircuit c  c1 c2
                , plusCircuit c' c2 c1
                , sameStart c c'
                ]
         ) ==>  sameMatch c c'
```

```
Main> verify plusCommutative
Proving: ... (t=0.2) Valid.
```

Reasoning about Embedded Languages (4)

Syntactic Reasoning and Verification of Compilation

But this approach does not always work ...

- Flash programs produce one finish for every start recieved.
- $\forall$ `P . P :|| skip` $\equiv$ `P`

We need *language invariants* and *environment conditions* and combine them together using **structural** and **temporal** induction.

This is indispensable when we *combine languages* and *optimise* compilation.

Reasoning about Embedded Languages (4)

Flash Invariant and Environment condition

Environment condition: I, the environment, hereby solemnly declare that I will always wait for the second party (also known as *the program*) to reciprocate with a finish for every start that I provide, before I proceed to give another start. I will thus adhere to $\int s \leq \int f + 1$.

The invariant: I, the program, hereby solemnly declare that I will never produce a finish unless the environment has previously provided me with a start. I also guarantee, never to provide more than one finish for each start that I receive. I will thus adhere to $\int f \leq \int s$.

Reasoning about Embedded Languages (4)

Regular Expressions Invariant

```
invariant (start, prefix, match) =
                  match ==> delay low prefix
```

In general, the property will be of the form:

```
property interface =
   always(environment interface) ==> invariant interface
```

Reasoning about Embedded Languages (4)

Regular Expressions Invariant

```
structuralInduction0 invariant constructor result =
  always (
    result <==> constructor
  ) ==> invariant result

structuralInduction1 invariant constructor (result, subexp1) =
  always (
    and2 ( result <==> constructor subexp1
         , invariant subexp1
         )
  ) ==> invariant result

etc
```

Reasoning about Embedded Languages

Regular Expressions Invariant

```
proveREInvariant invar =
  do
    verify (structuralInduction0 invar inputCircuit)
    verify (structuralInduction2 invar nondetCircuit)
    verify (structuralInduction2 invar seqCircuit)
    verify (structuralInduction1 invar repCircuit)
```

Discussion

- There's nothing new here, we're just making a point about how useful embedded languages can be for verification and language design/compilation.
- The new observation is that for safety properties, we have a finite model, and can thus 'easily' model-check it.
- Combining languages can be a hazardous business. Take precautions!

Current and Future Work

- Data paths complicate matters considerably, and we are currently investigating the use of this technique with Esterel.
- Correctness of compilation with respect to operational semantics
- Fitting better into a framework for combining synchronous languages