# Automatic Formal Synthesis of Hardware from Higher Order Logic

Mike Gordon [a] Juliano Iyoda [a] Konrad Slind [b] Scott Owens [b]

[a] *University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, UK*

[b] *University of Utah, School of Computing, 50 South Central Campus Drive, Salt Lake City, Utah UT84112, USA*

**Abstract**

A compiler is described that translates recursive function definitions in higher order logic to clocked synchronous hardware. Circuits are synthesised by formal proof mechanised in the HOL4 system. The logic terms representing hardware can be directly translated to Verilog HDL, simulated and then input to standard design automation tools. The theorem proving scripts that perform compilation are extensible. A simple example is adding rewrites for peephole optimisation, but all the theorem-proving infrastructure in HOL4 is available for tuning the compilation. The synthesisable subset can also be extended. For example, the core system can only compile tail-recursive function definitions, but a separate tool `linRec` is being developed to transform linear recursions to tail recursions, thereby extending the synthesisable subset to include linear recursion.

## 1 Introduction

Our goal is to create correct-by-construction hardware directly from mathematical specifications. The 'synthesisable subset' of logic is not intended to be fixed, but to grow as we do case studies. Currently, the compiler can automatically generate hardware to implement tail recursive function definitions. A typical example is the following iterative multiply-and-accumulate function:

```
MultIter(m,n,acc) =
 if m = 0 then (0,n,acc) else MultIter(m-1,n,n+acc)
```

Since `MultIter(m,n,acc) = (0,n,(m×n)+acc)`, a multiplier is defined by:

```
Mult(m,n) = SND(SND(MultIter(m,n,1)))
```

where `SND(SND(x,y,z))` evaluates to `z`, so `Mult(m,n) = m×n`. Using this multiplier one could then define the factorial function by:

```
FACT n = if n = 0 then 1 else Mult(n, FACT(n-1))
```

This isn't tail-recursive, so isn't synthesisable, however a separate tool `linRec` (see Section 4) can automatically generate a synthesisable definition:

*This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

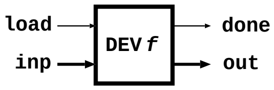
```
FactIter(n,acc) =
 if n = 0 then (n,acc) else FactIter(n-1,Mult(n,acc)))

Fact n = SND(FactIter (n,1))
```

`linRec` automatically proves `FACT = Fact`.

## 2 Representation of functions as circuits

The compiler translates a function $f$, defined in HOL, into a device `DEV` $f$ that computes $f$ via a four-phase handshake circuit on signals `load`, `inp`, `done` and `out`. These signals are a request line, a data input bus, an acknowledge line and a data output bus, respectively.




The exact behavior of such a handshaking device is specified in the HOL definition of the predicate `DEV`, which is given in the Appendix. This specification says roughly that if a value $v$ is input on `inp` when a request is made on `load` then eventually $f(v)$ will be output on `out`, and when this occurs is signalled on `done`. Here's a more detailed description: at the start of a transaction (say at time $t$) the device must be outputting `T` on `done` (to indicate it is ready) and the environment must be asserting `F` on `load`, i.e. in a state such that a positive edge on `load` can be generated. A transaction is initiated by asserting (at time $t{+}1$) the value `T` on `load`, i.e. `load` has a positive edge at time $t{+}1$. This causes the device to read the value, $v$ say, being input on `inp` (at time $t{+}1$) and to set `done` to `F`. The device then becomes insensitive to inputs until `T` is next asserted on `done`, at which time the computed value $f(v)$ will be output on `out`.

A synchronous circuit clocked on a clock signal `clk` conforms to the handshake protocol if it quarantees that the higher order logic formula

`DEV` $f$ `(load at clk,inp at clk,done at clk,out at clk)`

is true. The signals `load`, `inp`, `done`, `out` are modelled as functions mapping time to values, and the `at`-operator projects a signal to the sequence of values occurring at rising edges of the clock `clk`. More precisely $\sigma$ `at clk` is the signal that has the value at time $t$ that the signal $\sigma$ has at the $t^{\text{th}}$ rising edge of signal `clk`. The notation "$\sigma$`@clk`" is sometimes used instead of "$\sigma$ `at clk`". The formal theory of temporal projection (which is sometimes called 'temporal abstraction') is covered in detail in Melham's monograph [8].

An actual circuit is represented as a conjunction of terms, each representing a component instance. Internal wires are existentially-quantified. This is a standard modelling of hardware in higher order logic, and is also described in detail in Melham's book (ibid).

The result of compiling the definition of `MultIer` given earlier is the following theorem:

```
⊢ InfRise clk
   ==>
   (∃ v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21 v22
       v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34 v35 v36 v37 v38 v39 v40 v41 v42
       v43 v44 v45 v46 v47 v48 v49 v50 v51 v52 v53 v54 v55 v56 v57.
      DtypeT(clk,load,v21) ∧ NOT(v21,v20) ∧ AND(v20,load,v19) ∧ Dtype(clk,done,v18) ∧
      AND(v19,v18,v17) ∧ OR(v17,v16,v11) ∧ DtypeT(clk,v15,v23) ∧ NOT(v23,v22) ∧
      AND(v22,v15,v16) ∧ MUX(v16,v14,inp1,v3) ∧ MUX(v16,v13,inp2,v2) ∧
      MUX(v16,v12,inp3,v1) ∧ DtypeT(clk,v11,v26) ∧ NOT(v26,v25) ∧ AND(v25,v11,v24) ∧
      MUX(v24,v3,v27,v10) ∧ Dtype(clk,v10,v27) ∧ DtypeT(clk,v11,v30) ∧ NOT(v30,v29) ∧
      AND(v29,v11,v28) ∧ MUX(v28,v2,v31,v9) ∧ Dtype(clk,v9,v31) ∧
      DtypeT(clk,v11,v34) ∧ NOT(v34,v33) ∧ AND(v33,v11,v32) ∧ MUX(v32,v1,v35,v8) ∧
      Dtype(clk,v8,v35) ∧ DtypeT(clk,v11,v39) ∧ NOT(v39,v38) ∧ AND(v38,v11,v37) ∧
      NOT(v37,v7) ∧ CONSTANT 0 v40 ∧ EQ32(v3,v40,v36) ∧ Dtype(clk,v36,v6) ∧
      DtypeT(clk,v7,v44) ∧ NOT(v44,v43) ∧ AND(v43,v7,v42) ∧ AND(v42,v6,v5) ∧
      NOT(v6,v41) ∧ AND(v41,v42,v4) ∧ DtypeT(clk,v5,v48) ∧ NOT(v48,v47) ∧
      AND(v47,v5,v46) ∧ NOT(v46,v0) ∧ CONSTANT 0 v45 ∧ Dtype(clk,v45,out1) ∧
      Dtype(clk,v9,out2) ∧ Dtype(clk,v8,out3) ∧ DtypeT(clk,v4,v53) ∧ NOT(v53,v52) ∧
      AND(v52,v4,v51) ∧ NOT(v51,v15) ∧ CONSTANT 1 v54 ∧ SUB32(v10,v54,v50) ∧
      ADD32(v9,v8,v49) ∧ Dtype(clk,v50,v14) ∧ Dtype(clk,v9,v13) ∧ Dtype(clk,v49,v12) ∧
      Dtype(clk,v15,v56) ∧ AND(v15,v56,v55) ∧ AND(v0,v7,v57) ∧ AND(v57,v55,done))
   ==>
   DEV MultIter
       (load at clk, (inp1<>inp2<>inp3) at clk, done at clk, (out1<>out2<>out3) at clk)
```

This theorem has the form:

⊢ `InfRise clk ==>` *circuit* `==>` *device specification*

The term `InfRise clk` asserts that the signal `clk` has an infinite number of rising edges. This is a standard precondition for temporal abstraction (ibid) and is needed because of the use of the `at`-operator in the device specification.

The term *circuit* is the standard representation of the synthesised circuit in higher order logic. The components are described in Section 2.1. Circuits in this form are the lowest level of formal representation we generate. However they are easily converted to HDL and then simulated or input to other tools. We have written a 'pretty-printer' that generates Verilog HDL and have used several simulators and the Quartus II FPGA synthesis tool to run examples (including `MultIter` and `Fact`) on FPGAs.

The term *device specification* uses the HOL predicate `DEV` described above to specify that `MultIter` is computed using a four-phase handshake. Our compiler defaults to using 32-bit words. The input and output of `MultIter` are thus triples of 32-bit words, which are represented by terms `inp1<>inp2<>inp3` and `out1<>out2<>out3` where `inp1`, `inp2`, `inp3`, `out1`, `out2`, `out3` are 32-bit words and `<>` denotes word concatenation.

### *2.1 Components*

The compiler generates circuits using components from a predefined library, which can be changed to correspond to the targeted technology (the default target is Altera FPGAs synthesised using Quartus II).

The components used to implement `MultIter` are `NOT`, `AND`, `OR` (logic gates), `EQ32` (32-bit equality test), `MUX` (multiplexer), `DtypeT` (Boolean D-type register that powers up into an initial state storing the value `T`), `Dtype` (D-type register with unspecified initial state), `CONSTANT` (read-only register with a predefined value), ADD32 (32-bit adder) and 32-bit SUB32 (32-bit subtractor). Each of these components is defined in a standard style in higher order logic. For example, NOT is defined by:

$$\texttt{NOT(inp,out) = }\forall\texttt{t. out(t) = }\neg\texttt{inp(t)}$$

NOT is typical of all the combinational components (i.e. conponentns that can be implemented directly with logic gates without using registers). The two sequential components, Dtype and DtypeT, are registers that are triggered on the rising edge (posedge) of a clock and their definitions use the predicte Rise defined by:

$$\texttt{Rise}\ s\ t\ =\ \neg s(t)\ \wedge\ s(t{+}1)$$

and then Dtype and DtypeT are defined by:

$$\begin{array}{ll}\texttt{Dtype}\ \ (clk,d,q) = & \forall t.\ q(t{+}1) = \texttt{if Rise}\ clk\ t\ \texttt{then}\ d\ t\ \texttt{else}\ q\ t\\ \texttt{DtypeT}(clk,d,q) = & (q\ 0 = \texttt{T})\ \wedge\ \texttt{Dtype}(clk,d,q)\end{array}$$

These models are standard and are described in Melham's monograph (ibid).

## 3 How the compiler works

The compiler implements functions $f$ where $f : \sigma_1 \times \cdots \times \sigma_m \to \tau_1 \times \cdots \times \tau_n$ and $\sigma_1, \ldots, \sigma_m, \tau_1, \ldots, \tau_n$ are the types of values that can be carried on busses (e.g. $n$-bit words). The starting point of compilation is the definition of such a function $f$ by an equation of the form: $f(x_1, \ldots, x_n) = e$, where any recursive calls of $f$ in $e$ must be tail-recursive. Invoking our compiler on such a definition (if necessary with a user-supplied measure function to aid proof of termination) will first define $f$ in higher order logic (using TFL [14]) and then prove a theorem:

```
|- InfRise clk
     ==> circuit_f
     ==> DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

where *inputs* is inp1<>···<>inpm, *outputs* is out1<>···<>outn (with the type of inp$i$ matching $\sigma_i$ and the type of out$j$ matching $\tau_j$) and $circuit_f$ is a HOL term representing a circuit with inputs clk, load, inp1, ..., inpm and outputs done, out1, ..., outn.

The first step (**Step 1**) in compiling $f(x_1, \ldots, x_n) = e$ encodes $e$ as an applicative expression, $e_c$ say, built from the operators Seq (compute in sequence), Par (compute in parallel), Ite (if-then-else) and Rec (recursion), defined by:

$$\begin{array}{ll}\texttt{Seq}\ f_1\ f_2 & = \lambda x.\ f_2(f_1\ x)\\ \texttt{Par}\ f_1\ f_2 & = \lambda x.\ (f_1\ x,\ f_2\ x)\\ \texttt{Ite}\ f_1\ f_2\ f_3 & = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f_3\ x\\ \texttt{Rec}\ f_1\ f_2\ f_3 & = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ \texttt{Rec}\ f_1\ f_2\ f_3\ (f_3\ x)\end{array}$$

The encoding into an applicative expression built out of Seq, Par, Ite and Rec is performed by a proof script and results in a theorem $\vdash (\lambda(x_1, \ldots, x_n).\ e)\ =\ e_c$, and hence $\vdash f = e_c$. The algorithm used is straightforward and is not described here. As an example, the proof script deduces from:

$$\vdash \texttt{FactIter}(n, acc) \ = \\ \quad \texttt{if } n = 0 \texttt{ then } (n, acc) \texttt{ else } \texttt{FactIter}(n-1, n{\times}acc)$$

the theorem:

$$\begin{array}{l} \vdash \texttt{FactIter} \ = \\ \quad \texttt{Rec (Seq (Par } (\lambda(n,acc).\ n)\ (\lambda(n,acc).\ 0))\ (=)) \\ \qquad \texttt{(Par } (\lambda(n,acc).\ n)\ (\lambda(n,acc).\ acc)) \\ \qquad \texttt{(Par (Seq (Par } (\lambda(n,acc).\ n)\ (\lambda(n,acc).\ 1))\ (-)) \\ \qquad\quad \texttt{(Seq (Par } (\lambda(n,acc).\ n)\ (\lambda(n,acc).\ acc))\ (\times))) \end{array}$$

The second step (**Step 2**) is to replace the combinators `Seq`, `Par`, `Ite` and `Rec` with corresponding circuit constructors `SEQ`, `PAR`, `ITE` and `REC` that compose handshaking devices (see the Appendix for their definitions). The key property of these constructors are the following theorems that enable us to compositionally deduce theorems of the form $\vdash \mathit{Imp}_C \Longrightarrow \texttt{DEV}\ f$, where $\mathit{Imp}_C$ is a term constructed using the circuit constructors, and hence is a handshaking device (the long implication symbol $\Longrightarrow$ denotes implication lifted to functions – i.e. $f \Longrightarrow g \ = \ \forall x.\ f(x) \Rightarrow g(x)$):

$$\vdash \texttt{DEV}\ f \ \Longrightarrow \ \texttt{DEV}\ f$$

$$\begin{array}{l} \vdash (P_1 \Longrightarrow \texttt{DEV}\ f_1) \wedge (P_2 \Longrightarrow \texttt{DEV}\ f_2) \\ \quad \Rightarrow (\texttt{SEQ}\ P_1\ P_2 \Longrightarrow \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2)) \end{array}$$

$$\begin{array}{l} \vdash (P_1 \Longrightarrow \texttt{DEV}\ f_1) \wedge (P_2 \Longrightarrow \texttt{DEV}\ f_2) \\ \quad \Rightarrow (\texttt{PAR}\ P_1\ P_2 \Longrightarrow \texttt{DEV}\ (\texttt{Par}\ f_1\ f_2)) \end{array}$$

$$\begin{array}{l} \vdash (P_1 \Longrightarrow \texttt{DEV}\ f_1) \wedge (P_2 \Longrightarrow \texttt{DEV}\ f_2) \wedge (P_3 \Longrightarrow \texttt{DEV}\ f_3) \\ \quad \Rightarrow (\texttt{ITE}\ P_1\ P_2\ P_3 \Longrightarrow \texttt{DEV}\ (\texttt{Ite}\ f_1\ f_2\ f_3)) \end{array}$$

$$\begin{array}{l} \vdash \texttt{Total}(f_1, f_2, f_3) \\ \quad \Rightarrow (P_1 \Longrightarrow \texttt{DEV}\ f_1) \wedge (P_2 \Longrightarrow \texttt{DEV}\ f_2) \wedge (P_3 \Longrightarrow \texttt{DEV}\ f_3) \\ \quad \Rightarrow (\texttt{REC}\ P_1\ P_2\ P_3 \Longrightarrow \texttt{DEV}\ (\texttt{Rec}\ f_1\ f_2\ f_3)) \end{array}$$

where $\texttt{Total}(f_1, f_2, f_3)$ is a predicate ensuring termination.

If $e_c$ is an expression built using `Seq`,`Par`, `Ite` and `Rec`, then by instantiating the predicate variables $P_1$, $P_2$ and $P_3$, these theorems enable an expression $e_C$ to be built from circuit constructors `SEQ`, `PAR`, `ITE` and `REC` such that $\vdash e_C \Longrightarrow \texttt{DEV}\ e_c$. From Step 1 we have $\vdash f = e_c$, hence $\vdash e_C \Longrightarrow \texttt{DEV}\ f$

A function $f$ which is combinational can be packaged as a handshaking device using a constructor `ATM`, which creates a simple handshake interface and satisfies the refinement theorem:

$$\vdash \texttt{ATM}\ f \ \Longrightarrow \ \texttt{DEV}\ f$$

The circuit constructor `ATM` is defined with the other constructors in the Appendix. To avoid a proliferation of internal handshakes, when the proof script that constructs $e_C$ from $e_c$ is implementing Seq $f_1$ $f_2$, it checks to see whether $f_1$ or $f_2$ are compositions of combinational functions and if so introduces PRECEDE or FOLLOW instead of SEQ, using the theorems:

$$\vdash\ (P \Longrightarrow \texttt{DEV}\ f_2)\ \Rightarrow\ (\texttt{PRECEDE}\ f_1\ P \Longrightarrow \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2))$$

$$\vdash\ (P \Longrightarrow \texttt{DEV}\ f_1)\ \Rightarrow\ (\texttt{FOLLOW}\ P\ f_2 \Longrightarrow \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2))$$

PRECEDE $f$ $d$ processes inputs with $f$ before sending them to $d$ and FOLLOW $d$ $f$ processes outputs of $d$ with $f$. The definitions are:

$$\begin{array}{l} \texttt{PRECEDE}\ f\ d\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) = \\ \quad \exists v.\ \texttt{COMB}\ f\ (\mathit{inp}, v)\ \wedge\ d(\mathit{load}, v, \mathit{done}, \mathit{out}) \end{array}$$

$$\begin{array}{l} \texttt{FOLLOW}\ d\ f\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) = \\ \quad \exists v.\ d(\mathit{load}, \mathit{inp}, \mathit{done}, v)\ \wedge\ \texttt{COMB}\ f\ (v, \mathit{out}) \end{array}$$

COMB $f$ $(v_1, v_2)$ drives $v_2$ with $f(v_1)$, i.e. COMB $f$ $(v_1, v_2) = \forall t.\ v_2\ t = f(v_1\ t)$. SEQ $d_1$ $d_2$ introduces a handshake between the executions of $d_1$ and $d_2$, but PRECEDE $f$ $d$ and FOLLOW $d$ $f$ just 'wire' f before or after $d$, respectively, without introducing a handshake. Replacing SEQ by PRECEDE or FOLLOW is an example of a 'peephole' optimisation.

Step 2 results in a theorem $\vdash e_C \Longrightarrow$ DEV $f$ where $e_C$ is an expression built out of the circuit constructors ATM, SEQ, PAR, ITE, REC, PRECEDE and FOLLOW.

The third step (**Step 3**) is to rewrite with the definitions of these constructors (see their definitions in the Appendix) to get a circuit built out of standard kinds of gates (AND, OR, NOT and MUX), a generic combinational component COMB $g$ (where $g$ will be a function represented as a HOL $\lambda$-expression) and Dtype registers.

The next phase of compilation converts terms of the form COMB $g$ $(\mathit{inp}, \mathit{out})$ into circuits built only out of components that it is assumed can be directly realised in hardware. Such components currently include Boolean functions (e.g. $\wedge$, $\vee$ and $\neg$), multiplexers and simple operations on $n$-bit words (e.g. versions of $+$, $-$ and $<$, various shifts etc.). A special purpose proof rule uses a recursive algorithm to synthesise combinational circuits. For example:

$$\begin{array}{l} \vdash\ \texttt{COMB}\ (\lambda(m,n).\ (m < n,\ \ m{+}1))\ (\mathit{inp}1\texttt{<>}\mathit{inp}2, \mathit{out}1\texttt{<>}\mathit{out}2)\ = \\ \qquad \exists v0.\ \texttt{COMB}\ (<)\ (\mathit{inp}1\texttt{<>}\mathit{inp}2, \mathit{out}1)\ \wedge\ \texttt{CONSTANT}\ 1\ v0\ \wedge \\ \qquad\qquad \texttt{COMB}\ (+)\ (\mathit{inp}1\texttt{<>}v0, \mathit{out}2) \end{array}$$

where <> is bus concatenation, CONSTANT 1 $v0$ drives $v0$ high continuously, and COMB $<$ and COMB $+$ are assumed given components (if they were not given, then they could be implemented explicitly, but one has to stop somewhere).

The circuit resulting at the end of Step 3 uses unclocked abstract registers DEL, DELT and DFF that were chosen for convenience in defining ATM, SEQ, PAR, ITE and REC (see the Appendix). The register DFF is easily defined in terms of DEL, DELT and some combinational logic (details omitted).

The fourth step (**Step 4**) introduces a clock (with default name `clk`) and performs an automatic temporal abstraction as described in Melham's book [8] using the theorems:

$$\vdash \ \mathtt{InfRise}\ clk \ \Rightarrow \ \forall d\ q.\ \mathtt{Dtype}(clk, d, q) \ \Rightarrow \ \mathtt{DEL}(d\ \mathtt{at}\ clk,\ q\ \mathtt{at}\ clk)$$
$$\vdash \ \mathtt{InfRise}\ clk \ \Rightarrow \ \forall d\ q.\ \mathtt{DtypeT}(clk, d, q) \ \Rightarrow \ \mathtt{DELT}(d\ \mathtt{at}\ clk,\ q\ \mathtt{at}\ clk)$$

By instantiating *load*, *inp*, *done* and *out* in the theorem obtained by Step 3 to *load* `at` *clk*, *inp* `at` *clk*, *done* `at` *clk* and *out* `at` *clk*, respectively, and then performing some deductions using the above theorems and the monotonicity of existential quantification and conjunction with respect to implication, we obtain a theorem:

```
|- InfRise clk ==>
   circuit implementing f ==>
   DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

## 4 Additional tools: `linRec`

The 'synthesisable subset' of HOL is the subset that can be automatically compiled to circuits. Currently this only includes tail-recursive function definitions. We anticipate compiling higher level specifications by using proof tools that translate into the synthsisable subset. Such tools are envisioned as 'third party' add-ons developed for particular applications. As a preliminary experiment we are implementing a tool `linRec` to translate linear recursions to tail-recursions. This would enable, for example, the automatic generation of `MultIter` and `FactIter` from the more natural definitions:

```
Mult(m,n) = if m = 0w then 0w else m+Mult(m-1w,n)
Fact n    = if n = 0w then 1w else n*Fact(n-1)
```

A prototype implementation of `linRec` exists. It uses the following definition of linear and tail recursive recursion schemes:

```
linRec(x)    = if a(x) then b(x) else c (linRec(d x)) (e x)
tailRec(x,u) = if a(x) then c (b x) u else tailRec(d x, c (e x) u)
```

A linear recursion is matched with the definition of `linRec` to find values of `a`, `b`, `c`, `d`, `e` and then converted to a tail recursion by instantiating the theorem:

```
∀R a b c d e.
   WF R
   ∧ (∀x. ¬(a x) ==> R (d x) x)
   ∧ (∀p q r. c p (c q r) = c (c p q) r)
   ==>
   ∀x u. c (linRec a b c d e x) u = tailRec a b c d e (x,u)
```

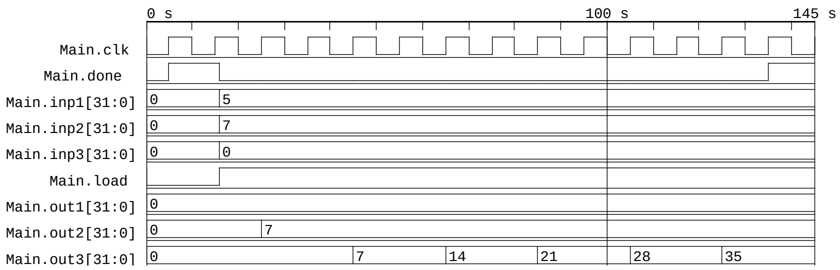
where `WF R` means that `R` is well-founded. Heuristics are used to choose an appropriate witness for `R`.

# 5 Current State and Future work

The compiler described here has been through several versions and now works robustly on all the examples we have tried.

We have written a 'pretty-printer' that converts circuit terms to Verilog, so that they can be simulated and input to other tools. There were initially difficulties when we first experimented with Verilog simulation. Our formal model represents bits as Booleans (`T`, `F`), but the Verilog simulation model is multivalued (`1`, `0`, `x`, `z` etc.), so our formal model does not predict the Verilog simulation behavior in which registers are initialised to `x`. As a result, Verilog simulation was generating undefined `x`-values instead of the outputs predicted by our proofs. The behaviour of most real hardware does not correspond to Verilog simulation because in reality registers initialise to a definite value, which is `0` for the Altera FPGAs we are using. By making our Verilog model of `Dtype` initialise its state to `0` we were able to successfully simulate all our examples. Our investigation of this issue was complicated by a bug in the Verilog simulation test harness: `load` was being asserted before `done` became `T`, violating the precondition of the handshake protocol, so even after we understood the initialisation problem, simulation was giving inexplicable results. However, once we fixed the testbench, everything worked. All our examples now execute correctly both under simulation and on an Altera Excalibur FPGA board.

If we simulate our implementation of `MultIter` with inputs $(5, 7, 0)$ using a standard Verilog simulator (http://www.icarus.com) and view the result with a waveform viewer (http://home.nc.rr.com/gtkwave), the result is:




`load` is asserted at time 15; `done` is `T` then, but immediately drops to `F` in response to `load` being asserted. At the time when `load` is asserted the values 5, 7 and 0 are put on lines `inp1`, `inp2` and `inp3`, respectively. At time 135 `done` rises to `T` again, and by then the values on `out1`, `out2` and `out3` are 0, 7 and 35, respectively, thus `Mult32Iter(5,7,0) = (0,7,35)`, which is correct.

In the immediate future we plan to complete a substantial example, being done at the University of Utah, to use our compiler to implement the Advanced Encryption Standard (AES) [11] algorithm for private-key encryption. This specifies a multi-round algorithm with primitive computations based on finite field operations. Starting from an existing formalization of AES [15], we have generated netlists and circuits for the major components of an encryption (and decryption) round. Although out work on AES is incomplete, our current progress confirms the viability of our synthesis methodology. The AES formalization includes a proof of functional correctness for the algorithm: specifically, encryption and decryption are inverse functions. Deriving the hardware from the proven specification using logical inference assures us that the hardware encrypter is the inverse of the hardware decrypter. Many of the AES specifications are not tail recursive, but formally deriving (and verifying) tail-recursive versions was straightforward. To automate such proofs for future work we developed the `linRec` tool (Section 4).

At present all data-refinement (e.g. from numbers or enumerated types to words) must be done manually, by proof in higher order logic. The HOL4 system has some 'boolification' facilities that automatically translate higher level data-types into bit-strings, and we hope to develop 'third-party' tools based on these that can be used for automatic data-refinement with the compiler.

We want to investigate using the compiler to generate test-bench monitors that can run in parallel simulation with designs that are not correct by construction. Thus our hardware can act as a "golden" reference against which to test other implementations.

The work described here is part of a project to create hardware/software combinations by proof. We hope to investigate the option of creating software for ARM processors and linking it to hardware created by our compiler (possibly packaged as an ARM co-processor). Our emphasis is likely to be on cryptographic hardware and software, because there is a clear need for high assurance of correct implementation in this domain.

## 6 Related work

Previous approaches to combine theorem provers and formal synthesis established an analogy between the goal-directed proof technique and an interactive design process. In LAMBDA, the user starts from the behavioural specification and builds the circuit incrementally by adding primitive hardware components which automatically simplify the goal [4]. Hanna *et al.* [5] introduce several *techniques* (functions) that simplify the current goal into simpler subgoals. Techniques are adaptations to hardware design of *tactics* in LCF.

Alternative approaches synthesise circuits by applying semantic-preserving transformations to their specifications. For instance, the Digital Design Derivation (DDD) transforms finite-state machines specified in terms of tail-recursive lambda abstractions into hierarchical Boolean systems [6]. Lava and Hydra are both hardware description languages embedded in Haskell whose programs consist of definitions of gates and their connections (netlists) [1,10]. While Lava interfaces with external theorem provers to verify its circuits, Hydra designers can synthesise them via formal equational reasoning (using definitions and lemmas from functional programming). The functional languages $\mu$FP and Ruby adopt similar principles in hardware design [7,13]. The circuits are defined in terms of primitive functions over Booleans, numbers and lists, and higher-order functions, the *combining forms*, which compose hardware blocks in different structures. Their mathematical properties provide a calculational style in design exploration.

These approaches deal with an interactive synthesis at the gate or state-machine level of abstraction only. Moreover, the synthesis and the proof of correctness require a substantial user guidance. Gropius and SAFL are two related works that address these issues.

Gropius is a hardware description language defined as a subset of HOL [2,3]. Its algorithmic level provides control structures like if-then-else, sequential composition and while loop. The atomic commands are DFGs (data flow graphs) represented by lambda abstractions. The compiler initially combines every while loop into a single one at the outermost level of the program:

`PROGRAM` *out_default* (`LOCVAR` *vars* (`WHILE` *c* (`PARTIALIZE` *b*)))

The body *b* of the `WHILE` loop is an acyclic DFG. The list *out_default* provides initial values for the output variables. The term `LOCVAR` declares the local variables *vars* and `PARTIALIZE` converts a non-recursive (terminating) DFG into a potentially non-terminating command. The compiler then synthesises a handshaking interface which encapsulates this program. Each of these hardware blocks are now regarded as primitive blocks or *processes* at the system level. Processes are connected via communication units (*k-processes*) which implement delay, synchronisation, duplication, splitting and joining of a process output data (actually there are 10 different k-processes [2]). Although the synthesis produces the proof of correctness of each process and k-process, the correctness of the top-level system is not generated. The reason for that is mainly because the top-level interface of a network of processes and k-processes does not match the handshaking interface pattern.

Our compilation method is partly inspired by SAFL (Statically Allocated Functional Language) [9], especially the ideas in Richard Sharp's PhD thesis [12]. SAFL is a first-order functional language whose programs consist of a sequence of tail-recursive function definitions. Its high-level of abstraction allows the exploitation of powerful program analyses and optimisations not available in traditional synthesis systems. However, the synthesis is not based on the correct-by-construction principles and the compiler has not been verified.

The novelty of our approach is the compilation of functional programs by composing especially designed and pre-verified circuit constructors. As each of these circuit constructors has the key property of implementing a device that computes precisely their corresponding combinators, the verification and the compilation of functional programs can be done automatically.

## 7 acknowledgements

David Greaves gave us advice on the hardware implementation of handshake protocols and also helped us understand the results of simulating circuits produced by our compiler. Simon Moore and Robert Mullins lent us an Escalibur FPGA board on which we are running compiled hardware and they helped us with the Quartus II design software that we are using to drive the board.


## References


[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.

[2] Christian Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*, pages 11–20, Braunschweig, Germany, 1999. Shaker-Verlag.

[3] Christian Blumenröhr and Dirk Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In *Proceedings of the Digital System Design Workshop at the Euromicro 98 Conference, Västeras, Sweden*, pages 34–37, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1998.

[4] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.

[5] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 153–170. North-Holland, 1989.

[6] Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990.

[7] Geraint Jones and Mary Sheeran. Circuit design in Ruby. Lecture notes on Ruby from a summer school in Lyngby, Denmark., September 1990.

[8] Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.

[9] Alan Mycroft and Richard Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 236–251, Genova, Italy, April 2001. Springer-Verlag. LNCS Vol. 2031.

[10] John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2002.

[11] United States National Institute of Standards and Technology. Advanced Encryption Standard. 2001.

[12] Richard Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2002.

[13] Mary Sheeran. muFP, A language for VLSI design. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 104–112. ACM, ACM, August 1984.

[14] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 381–398, Turku, Finland, August 1996. Springer-Verlag.

[15] Konrad Slind. A verification of Rijndael in HOL. In V. A Carreno, C. A. Munoz, and S. Tahar, editors, *Supplementary Proceedings of TPHOLs 2002*, number CP-2002-211736 in NASA Conference Proceedings, August 2002.

## APPENDIX: formal specifications in higher order logic

The specification of the four-phase handshake protocol is represented by the definition of the predicate `DEV`, which uses auxiliary predicates `Posedge` and `HoldF`. A positive edge of a signal is defined as the transition of its value from low to high or, in our case, from `F` to `T`. The term `HoldF` $(t_1, t_2)$ $s$ says that a signal $s$ holds a low value `F` during a half-open interval starting at $t_1$ to just before $t_2$. The formal definitions are:

$$\vdash \texttt{Posedge}\ s\ t \ = \ \texttt{if}\ t{=}0\ \texttt{then F else}\ (\neg\, s(t{-}1)\ \wedge\ s\ t\ )$$

$$\vdash \texttt{HoldF}\ (t_1, t_2)\ s = \ \forall t.\ t_1 \leq t < t_2 \ \Rightarrow\ \neg(s\ t)$$

The behaviour of the handshaking device computing a function $f$ is described by the term `DEV` $f$ $(load, inp, done, out)$ where:

$$\begin{array}{l}
\vdash \texttt{DEV}\ f\ (load, inp, done, out) = \\
\quad (\forall t.\ done\ t\ \wedge\ \texttt{Posedge}\ load\ (t{+}1) \\
\qquad \Rightarrow \\
\qquad \exists t'.\ t' > t{+}1\ \wedge\ \texttt{HoldF}\ (t{+}1, t')\ done\ \wedge \\
\qquad\quad done\ t'\ \wedge\ (out\ t' = f(inp\ (t{+}1))))\ \wedge \\
\quad (\forall t.\ done\ t\ \wedge\ \neg(\texttt{Posedge}\ load\ (t{+}1))\ \Rightarrow\ done\ (t{+}1))\ \wedge \\
\quad (\forall t.\ \neg(done\ t)\ \Rightarrow\ \exists t'.\ t' > t\ \wedge done\ t')
\end{array}$$

The first conjunct in the right-hand side specifies that if the device is available and a positive edge occurs on *load*, there exists a time $t'$ in future when *done* signals its termination and the output is produced. The value of the output at time $t'$ is the result of applying $f$ to the value of the input at time $t{+}1$. The signal *done* holds the value F during the computation. The second conjunct specifies the situation where no call is made on *load* and the device simply remains idle. Finally, the last conjunct states that if the device is busy, it will eventually finish its computation and become idle.

## The circuit constructors

The following primitive components are used by the circuit constructors.

$$\vdash \texttt{AND}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)$$
$$\vdash \texttt{OR}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \vee\ in_2\ t)$$
$$\vdash \texttt{NOT}\ (inp, out)\ = \forall t.\ out\ t = \neg(inp\ t)$$
$$\vdash \texttt{MUX}(sw, in_1, in_2, out)\ =\ \forall t.\ out\ t = \texttt{if}\ sw\ t\ \texttt{then}\ in_1\ t\ \texttt{else}\ in_2\ t$$
$$\vdash \texttt{COMB}\ f\ (inp, out)\ = \forall t.\ out\ t = f(inp\ t)$$
$$\vdash \texttt{DEL}\ (inp, out)\ = \forall t.\ out(t{+}1) = inp\ t$$
$$\vdash \texttt{DELT}\ (inp, out)\ = (out\ 0 = \texttt{T})\ \wedge\ \forall t.\ out(t{+}1) = inp\ t$$
$$\vdash \texttt{DFF}(d, sel, q)\ =\ \forall t.\ q(t{+}1) = \texttt{if Posedge}\ sel\ (t{+}1)\ \texttt{then}\ d(t{+}1)\ \texttt{else}\ q\ t$$
$$\vdash \texttt{POSEDGE}(inp, out) = \exists c_0\ c_1.\ \texttt{DELT}(inp, c_0) \wedge \texttt{NOT}(c_0, c_1) \wedge \texttt{AND}(c_1, inp, out)$$

Atomic handshaking devices.

$$\begin{aligned}\vdash\ & \texttt{ATM}\ f\ (load, inp, done, out) = \\ & \exists c_0\ c_1.\ \texttt{POSEDGE}(load, c_0)\ \wedge\ \texttt{NOT}(c_0, done)\ \wedge \\ & \qquad \texttt{COMB}\ f\ (inp, c_1)\ \wedge\ \texttt{DEL}(c_1, out)\end{aligned}$$

Sequential composition of handshaking devices.

$$\begin{aligned}\vdash\ & \texttt{SEQ}\ f\ g\ (load, inp, done, out) = \\ & \exists c_0\ c_1\ c_2\ c_3\ data. \\ & \quad \texttt{NOT}(c_2, c_3)\ \wedge\ \texttt{OR}(c_3, load, c_0)\ \wedge\ f(c_0, inp, c_1, data)\ \wedge \\ & \quad g(c_1, data, c_2, out)\ \wedge\ \texttt{AND}(c_1, c_2, done)\end{aligned}$$

Parallel composition of handshaking devices.

$$\begin{aligned}\vdash\ & \texttt{PAR}\ f\ g\ (load, inp, done, out) = \\ & \exists c_0\ c_1\ start\ done_1\ done_2\ data_1\ data_2\ out_1\ out_2. \\ & \quad \texttt{POSEDGE}(load, c_0)\ \wedge\ \texttt{DEL}(done, c_1)\ \wedge\ \texttt{AND}(c_0, c_1, start)\ \wedge \\ & \quad f(start, inp, done_1, data_1)\ \wedge\ g(start, inp, done_2, data_2)\ \wedge \\ & \quad \texttt{DFF}(data_1, done_1, out_1)\ \wedge\ \texttt{DFF}(data_2, done_2, out_2)\ \wedge \\ & \quad \texttt{AND}(done_1, done_2, done)\ \wedge\ (out = \lambda t.\ (out_1\ t, out_2\ t))\end{aligned}$$

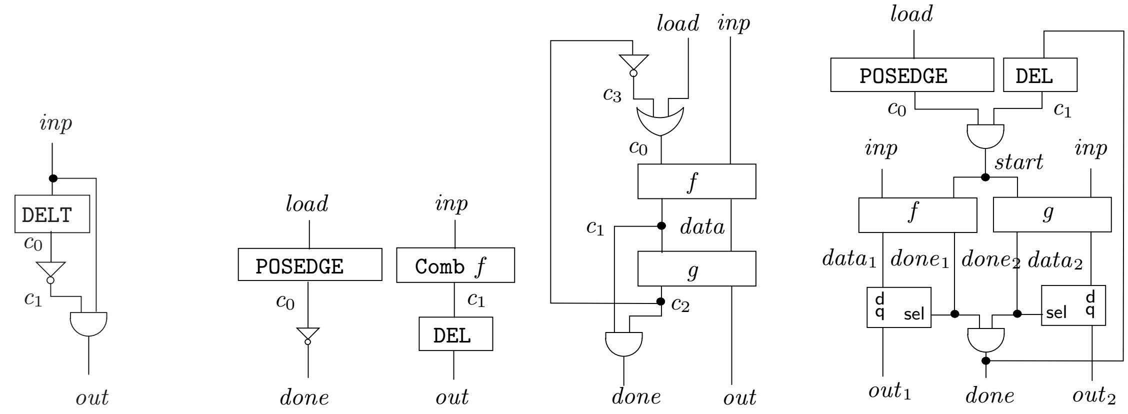
Conditional composition of handshaking devices.

$$
\begin{array}{l}
\vdash \texttt{ITE}\ e\ f\ g\ (load, inp, done, out) = \\
\quad \exists c_0\ c_1\ c_2\ start\ start'\ done\_e\ data\_e\ q\ not\_e\ data\_f\ data\_g\ sel \\
\quad\ \ done\_f\ done\_g\ start\_f\ start\_g. \\
\quad\quad \texttt{POSEDGE}(load, c_0)\ \wedge\ \texttt{DEL}(done, c_1)\ \wedge\ \texttt{AND}(c_0, c_1, start)\ \wedge \\
\quad\quad e(start, inp, done\_e, data\_e)\ \wedge\ \texttt{POSEDGE}(done\_e, start')\ \wedge \\
\quad\quad \texttt{DFF}(data\_e, done\_e, sel)\ \wedge\ \texttt{DFF}(inp, start, q)\ \wedge \\
\quad\quad \texttt{AND}(start', data\_e, start\_f)\ \wedge\ \texttt{NOT}(data\_e, not\_e)\ \wedge \\
\quad\quad \texttt{AND}(start', not\_e, start\_g)\ \wedge\ f(start\_f, q, done\_f, data\_f)\ \wedge \\
\quad\quad g(start\_g, q, done\_g, data\_g)\ \wedge\ \texttt{MUX}(sel, data\_f, data\_g, out)\ \wedge \\
\quad\quad \texttt{AND}(done\_e, done\_f, c_2)\ \wedge\ \texttt{AND}(c_2, done\_g, done)
\end{array}
$$

Tail recursion constructor.

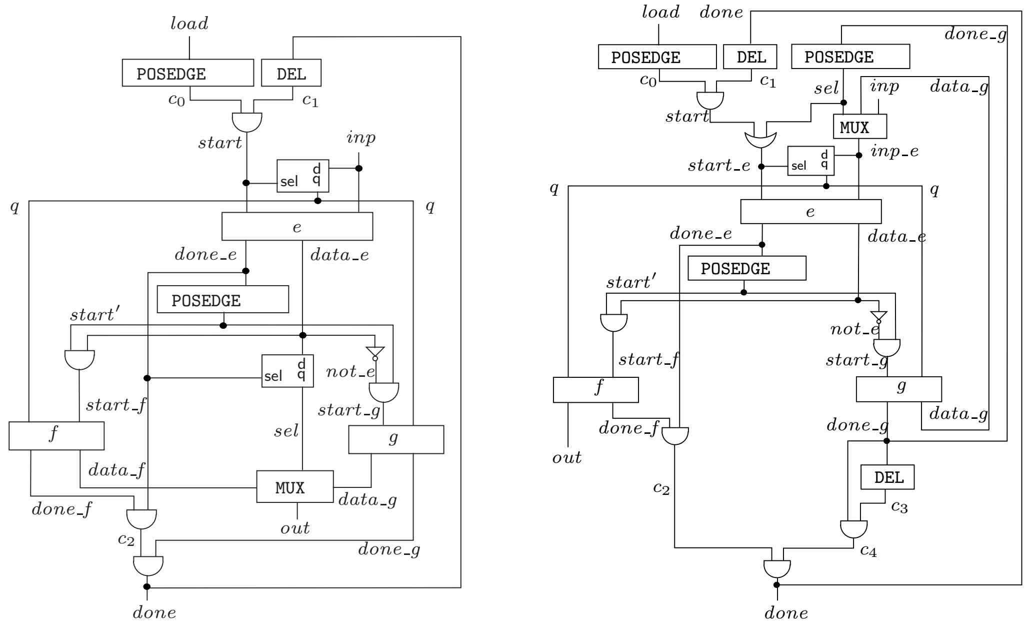
$$
\begin{array}{l}
\vdash \texttt{REC}\ e\ f\ g\ (load, inp, done, out) = \\
\quad \exists done\_g\ data\_g\ start\_e\ q\ done\_e\ data\_e\ start\_f\ start\_g\ inp\_e\ done\_f \\
\quad\ \ c_0\ c_1\ c_2\ c_3\ c_4\ start\ sel\ start'\ not\_e. \\
\quad\quad \texttt{POSEDGE}(load, c_0)\ \wedge\ \texttt{DEL}(done, c_1)\ \wedge\ \texttt{AND}(c_0, c_1, start)\ \wedge \\
\quad\quad \texttt{OR}(start, sel, start\_e)\ \wedge\ \texttt{POSEDGE}(done\_g, sel)\ \wedge \\
\quad\quad \texttt{MUX}(sel, data\_g, inp, inp\_e)\ \wedge\ \texttt{DFF}(inp\_e, start\_e, q)\ \wedge \\
\quad\quad e(start\_e, inp\_e, done\_e, data\_e) \wedge \texttt{POSEDGE}(done\_e, start')\ \wedge \\
\quad\quad \texttt{AND}(start', data\_e, start\_f)\ \wedge\ \texttt{NOT}(data\_e, not\_e)\ \wedge \\
\quad\quad \texttt{AND}(not\_e, start', start\_g)\ \wedge\ f(start\_f, q, done\_f, out)\ \wedge \\
\quad\quad g(start\_g, q, done\_g, data\_g)\ \wedge\ \texttt{DEL}(done\_g, c_3)\ \wedge \\
\quad\quad \texttt{AND}(done\_g, c_3, c_4)\ \wedge\ \texttt{AND}(done\_f, done\_e, c_2)\ \wedge\ \texttt{AND}(c_2, c_4, done)
\end{array}
$$

Circuit diagrams of the circuit constructors are shown on the following page.




(a) POSEDGE (b) ATM $f$ (c) SEQ $f$ $g$ (d) PAR $f$ $g$

Fig. 1. Implementation of composite devices.




(a) IF $e$ $f$ $g$ (b) REC $e$ $f$ $g$

Fig. 2. The conditional and the recursive constructors.