Automatic Formal Synthesis of Hardware from Higher Order Logic

Mike Gordon\textsuperscript{a}  Juliano Iyoda\textsuperscript{a}  Konrad Slind\textsuperscript{b}  Scott Owens\textsuperscript{b}

\textsuperscript{a} University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, UK
\textsuperscript{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:

\[
\text{MultIter}(m,n,acc) =
\begin{cases} 
(0,n,acc) & \text{if } m = 0 \\
\text{MultIter}(m-1,n,n+acc) & \text{else}
\end{cases}
\]

Since \(\text{MultIter}(m,n,acc) = (0,n,(m \times n) + acc)\), a multiplier is defined by:

\[
\text{Mult}(m,n) = \text{SND}(\text{SND}(\text{MultIter}(m,n,1)))
\]

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

\[
\text{FACT} n = \begin{cases} 
1 & \text{if } n = 0 \\
\text{Mult}(n, \text{FACT}(n-1)) & \text{else}
\end{cases}
\]

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

\textit{This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs}
\[
\text{FactIter}(n, acc) = \\
\quad \text{if } n = 0 \text{ then } (n, acc) \text{ else } \text{FactIter}(n-1, \text{Mult}(n, acc))
\]

\[
\text{Fact } n = \text{SND(FactIter } (n,1))
\]

\text{linRec automatically proves } FACT = \text{Fact.}

2 \text{ Representation of functions as circuits}

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

![handshake_circuit](image)

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

A synchronous circuit clocked on a clock signal \( \text{clk} \) conforms to the handshake protocol if it guarantees that the higher order logic formula

\[
\text{DEV } f ( \text{load at clk, inp at clk, done at clk, out at clk})
\]

is true. The signals \( \text{load, inp, done, out} \) are modelled as functions mapping time to values, and the \( \text{at}-\text{operator} \) projects a signal to the sequence of values occurring at rising edges of the clock \( \text{clk} \). More precisely \( \sigma \text{ at clk} \) is the signal that has the value at time \( t \) that the signal \( \sigma \) has at the \( t^{th} \) rising edge of signal \( \text{clk} \). The notation “\( \sigma \text{ at clk} \)” is sometimes used instead of “\( \sigma \text{ at clk} \)”.

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 \( \text{MultIter} \) given earlier is the following theorem:
This theorem has the form:

\[ \vdash \text{InfRise clk} \quad (\exists v_0 \ldots v_{53}) \quad \text{circuit} \quad \Rightarrow \quad \text{device specification} \]

The term \text{InfRise clk} asserts that the signal \text{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 \text{at}-operator in the device specification.

The term \text{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 \text{MultIter} and \text{Fact}) on FPGAs.

The term \text{device specification} uses the HOL predicate \text{DEV} described above to specify that \text{MultIter} is computed using a four-phase handshake. Our compiler defaults to using 32-bit words. The input and output of \text{MultIter} are thus triples of 32-bit words, which are represented by terms \text{inp1<>inp2<>inp3} and \text{out1<>out2<>out3} where \text{inp1, inp2, inp3, out1, out2, out3} are 32-bit words and \text{<>} 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 \text{MultIter} are \text{NOT}, \text{AND}, \text{OR} (logic gates), \text{EQ32} (32-bit equality test), \text{MUX} (multiplexer), \text{DtypeT} (Boolean D-type register that powers up into an initial state storing the value T), \text{Dtype} (D-type register with unspecified initial state), \text{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:

\[
\text{NOT}(\text{inp}, \text{out}) = \forall t. \text{out}(t) = \neg \text{inp}(t)
\]

NOT is typical of all the combinational components (i.e. components 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 predicate \(\text{Rise}\) defined by:

\[
\text{Rise}\ st = \neg s(t) \land s(t+1)
\]

and then Dtype and DtypeT are defined by:

\[
\text{Dtype}\ (\text{clk}, d, q) = \forall t. q(t+1) = \text{if} \ \text{Rise} \ \text{clk}\ t \ \text{then} \ d\ t \ \text{else} \ q\ t \\
\text{DtypeT}\ (\text{clk}, d, q) = (q\ 0 = T) \land \text{Dtype}(\text{clk}, d, q)
\]

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

3 How the compiler works

The compiler implements functions \(f\) where \(f : \sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n\) 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:

\[\vdash \text{InfRise}\ \text{clk} \Rightarrow \text{circuit}_f \Rightarrow \text{DEV}\ f\ (\text{load}\ \text{at}\ \text{clk, inputs}\ \text{at}\ \text{clk, done}\ \text{at}\ \text{clk, outputs}\ \text{at}\ \text{clk})\]

where inputs is \(\text{inp}_1<\cdots<\text{inp}_m\), outputs is \(\text{out}_1<\cdots<\text{out}_n\) (with the type of \(\text{inp}_i\) matching \(\sigma_i\) and the type of \(\text{out}_j\) matching \(\tau_j\)) and \(\text{circuit}_f\) is a HOL term representing a circuit with inputs \(\text{clk, load, inp}_1, \ldots, \text{inp}_m\) and outputs \(\text{done, out}_1, \ldots, \text{out}_n\).

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 \(\text{Seq}\) (compute in sequence), \(\text{Par}\) (compute in parallel), \(\text{Ite}\) (if-then-else) and \(\text{Rec}\) (recursion), defined by:

\[
\begin{align*}
\text{Seq}\ f_1\ f_2 &= \lambda x.\ f_2(f_1\ x) \\
\text{Par}\ f_1\ f_2 &= \lambda x.\ (f_1\ x, f_2\ x) \\
\text{Ite}\ f_1\ f_2\ f_3 &= \lambda x.\ \text{if}\ f_1\ x\ \text{then}\ f_2\ x\ \text{else}\ f_3\ x \\
\text{Rec}\ f_1\ f_2\ f_3 &= \lambda x.\ \text{if}\ f_1\ x\ \text{then}\ f_2\ x\ \text{else}\ \text{Rec}\ f_1\ f_2\ f_3\ (f_3\ x)
\end{align*}
\]

The encoding into an applicative expression built out of \(\text{Seq}, \text{Par}, \text{Ite}\) and \(\text{Rec}\) 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:

$\vdash \text{FactIter}(n, acc) = \begin{cases} n = 0 & \text{then } (n, acc) \text{ else } \text{FactIter}(n - 1, n \times acc) \end{cases}$

the theorem:

$\vdash \text{FactIter} = \text{Rec} \left( \text{Seq} \left( \text{Par} \left( \lambda(n, acc). n \right) (\lambda(n, acc). 0) \right) (=) \right) \left( \text{Par} \left( \lambda(n, acc). n \right) (\lambda(n, acc). acc) \right) \left( \text{Par} \left( \text{Seq} \left( \text{Par} \left( \lambda(n, acc). n \right) (\lambda(n, acc). 1) \right) (-) \right) \right) \left( \text{Seq} \left( \text{Par} \left( \lambda(n, acc). n \right) (\lambda(n, acc). acc) \right) (\times) \right) \right)$

The second step (Step 2) is to replace the combinators \text{Seq}, \text{Par}, \text{Ite} and \text{Rec} with corresponding circuit constructors \text{SEQ}, \text{PAR}, \text{ITE} and \text{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 $\vdash \text{Imp}_C \implies \text{DEV} f$, where $\text{Imp}_C$ is a term constructed using the circuit constructors, and hence is a handshaking device (the long implication symbol $\implies$ denotes implication lifted to functions – i.e. $f \implies g = \forall x. f(x) \implies g(x)$):

$\vdash \text{DEV } f \implies \text{DEV } f$

$\vdash (P_1 \implies \text{DEV } f_1) \land (P_2 \implies \text{DEV } f_2)$

$\implies (\text{SEQ } P_1 P_2 \implies \text{DEV } (\text{Seq } f_1 f_2))$

$\vdash (P_1 \implies \text{DEV } f_1) \land (P_2 \implies \text{DEV } f_2)$

$\implies (\text{PAR } P_1 P_2 \implies \text{DEV } (\text{Par } f_1 f_2))$

$\vdash (P_1 \implies \text{DEV } f_1) \land (P_2 \implies \text{DEV } f_2) \land (P_3 \implies \text{DEV } f_3)$

$\implies (\text{ITE } P_1 P_2 P_3 \implies \text{DEV } (\text{Ite } f_1 f_2 f_3))$

$\vdash \text{Total}(f_1, f_2, f_3)$

$\implies (P_1 \implies \text{DEV } f_1) \land (P_2 \implies \text{DEV } f_2) \land (P_3 \implies \text{DEV } f_3)$

$\implies (\text{REC } P_1 P_2 P_3 \implies \text{DEV } (\text{Rec } f_1 f_2 f_3))$

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

If $e_c$ is an expression built using \text{Seq}, \text{Par}, \text{Ite} and \text{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 \text{SEQ}, \text{PAR}, \text{ITE} and \text{REC} such that $\vdash e_C \implies \text{DEV } e_c$. From Step 1 we have $\vdash f = e_c$, hence $\vdash e_C \implies \text{DEV } f$

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

$\vdash \text{ATM } f \implies \text{DEV } f$

The circuit constructor \text{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 \(\text{Seq} f_1 f_2\), it checks to see whether \(f_1\) or \(f_2\) are compositions of combinational functions and if so introduces \(\text{PRECEDE}\) or \(\text{FOLLOW}\) instead of \(\text{SEQ}\), using the theorems:

\[
\vdash (P \implies \text{DEV} f_2) \implies (\text{PRECEDE} f_1 P \implies \text{DEV} (\text{Seq} f_1 f_2))
\]

\[
\vdash (P \implies \text{DEV} f_1) \implies (\text{FOLLOW} P f_2 \implies \text{DEV} (\text{Seq} f_1 f_2))
\]

\(\text{PRECEDE}\) processes inputs with \(f\) before sending them to \(d\) and \(\text{FOLLOW}\) processes outputs of \(d\) with \(f\). The definitions are:

\[
\text{PRECEDE}\ f\ d\ (\text{load, inp, done, out}) = \exists v. \text{COMB}\ f\ (\text{inp, v}) \land d(\text{load, v, done, out})
\]

\[
\text{FOLLOW}\ d\ f\ (\text{load, inp, done, out}) = \exists v. d(\text{load, inp, done, v}) \land \text{COMB}\ f\ (v, \text{out})
\]

\(\text{COMB}\ f\ (v_1, v_2)\) drives \(v_2\) with \(f(v_1)\), i.e. \(\text{COMB}\ f\ (v_1, v_2) = \forall t. v_2 t = f(v_1 t)\).

\(\text{SEQ}\ d_1 d_2\) introduces a handshake between the executions of \(d_1\) and \(d_2\), but \(\text{PRECEDE}\ f\ d\) and \(\text{FOLLOW}\ d\ f\) just 'wire' \(f\) before or after \(d\), respectively, without introducing a handshake. Replacing \(\text{SEQ}\) by \(\text{PRECEDE}\) or \(\text{FOLLOW}\) is an example of a 'peephole' optimisation.

Step 2 results in a theorem \(\vdash e_C \implies \text{DEV}\ f\) where \(e_C\) is an expression built out of the circuit constructors \(\text{ATM, SEQ, PAR, ITE, REC, PRECEDE}\) and \(\text{FOLLOW}\).

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 (\(\text{AND, OR, NOT}\) and \(\text{MUX}\)), a generic combinational component \(\text{COMB}\ g\) (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 \(\text{COMB}\ g\ (\text{inp, out})\) 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. \(\land, \lor\) and \(\neg\)), multiplexers and simple operations on \(n\)-bit words (e.g. versions of \(+, -, \text{and} <\), various shifts etc.). A special purpose proof rule uses a recursive algorithm to synthesise combinational circuits. For example:

\[
\vdash \text{COMB}\ (\lambda (m, n). (m < n, m+1))\ (\text{inp}1<>\text{inp}2, \text{out}1<>\text{out}2) = \\
\exists v0. \text{COMB}\ (<)\ (\text{inp}1<>\text{inp}2, \text{out}1) \land \text{CONSTANT}\ 1\ v0 \land \\
\text{COMB}\ (+)\ (\text{inp}1<>v0, \text{out}2)
\]

where <> is bus concatenation, \(\text{CONSTANT}\ 1\ v0\) drives \(v0\) high continuously, and \(\text{COMB}\ <\) and \(\text{COMB}\ +\) 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 \(\text{DEL, DELT}\) and \(\text{DFF}\) that were chosen for convenience in defining \(\text{ATM, SEQ, PAR, ITE}\) and \(\text{REC}\) (see the Appendix). The register \(\text{DFF}\) is easily defined in terms of \(\text{DEL, DELT}\) 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:

\[
\begin{align*}
\vdash \text{InfRise} \, clk & \Rightarrow \forall d \, q. \, \text{Dtype}(clk,d,q) \Rightarrow \text{DEL}(d \, \text{at} \, clk, \, q \, \text{at} \, clk) \\
\vdash \text{InfRise} \, clk & \Rightarrow \forall d \, q. \, \text{Dtype}(clk,d,q) \Rightarrow \text{DELT}(d \, \text{at} \, clk, \, q \, \text{at} \, clk)
\end{align*}
\]

By instantiating \( load \), \( inp \), \( done \) and \( out \) in the theorem obtained by Step 3 to \( load \, \text{at} \, clk \), \( inp \, \text{at} \, clk \), \( done \, \text{at} \, clk \) and \( out \, \text{at} \, clk \), 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:

\[
\vdash \text{InfRise} \, clk \Rightarrow \forall d \, q. \, \text{Dtype}(clk,d,q) \Rightarrow \text{DEL}(d \, \text{at} \, clk, \, q \, \text{at} \, clk)
\]

4 Additional tools: \texttt{linRec}

The ‘synthesisisable 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 synthesisable subset. Such tools are envisioned as ‘third party’ add-ons developed for particular applications. As a preliminary experiment we are implementing a tool \texttt{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:

\[
\begin{align*}
\text{Mult}(m,n) & = \text{if } m = 0w \text{ then } 0w \text{ else } m + \text{Mult}(m-1w,n) \\
\text{Fact} \, n & = \text{if } n = 0w \text{ then } 1w \text{ else } n \times \text{Fact}(n-1)
\end{align*}
\]

A prototype implementation of \texttt{linRec} exists. It uses the following definition of linear and tail recursive recursion schemes:

\[
\begin{align*}
\text{linRec}(x) & = \text{if } a(x) \text{ then } b(x) \text{ else } c \left( \text{linRec}(d \, x) \right) \left( e \, x \right) \\
\text{tailRec}(x,u) & = \text{if } a(x) \text{ then } c \left( b \, x \right) u \text{ else } \text{tailRec}(d \, x, c \left( e \, x \right) u)
\end{align*}
\]

A linear recursion is matched with the definition of \texttt{linRec} to find values of \( a \), \( b \), \( c \), \( d \), \( e \) and then converted to a tail recursion by instantiating the theorem:

\[
\forall R \, a \, b \, c \, d \, e.
\begin{align*}
\text{WF} \, R \\
\land \left( \forall x. \, \neg(a \, x) \Rightarrow R \left( d \, x \right) \, x \right) \\
\land \left( \forall p \, q \, r. \, c \, p \, \left( c \, q \, r \right) = c \, \left( c \, p \, q \right) \, r \right) \\
\Rightarrow \forall x \, u. \, c \left( \text{linRec} \, a \, b \, c \, d \, e \, x \right) \, u = \text{tailRec} \, a \, b \, c \, d \, e \left( x, u \right)
\end{align*}
\]

where \( \text{WF} \, R \) means that \( R \) is well-founded. Heuristics are used to choose an appropriate witness for \( R \).
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 \( \text{Mult32Iter}(5, 7, 0) = (0, 7, 35) \), 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 en-
cryption (and decryption) round. Although our 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 sub-goals. 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:

```plaintext
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


APPENDIX: formal specifications in higher order logic

The specification of the four-phase handshake protocol is represented by the definition of the predicate \( \text{DEV} \), which uses auxiliary predicates \( \text{Posedge} \) and \( \text{HoldF} \). 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 \( \text{HoldF} (t_1, t_2) \) 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:

\[
\begin{align*}
\vdash \text{Posedge} \quad t \quad = \quad & \text{if } t=0 \text{ then } F \text{ else } (\neg s(t-1) \land s \ t) \\
\vdash \text{HoldF} (t_1, t_2) \quad s \quad = \quad & \forall t. \ t_1 \leq t < t_2 \Rightarrow \neg (s \ t)
\end{align*}
\]

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

\[
\begin{align*}
\vdash \text{DEV} \ f \ (\text{load, inp, done, out}) = \\
& (\forall t. \ \text{done} \ t \land \text{Posedge} \ \text{load} \ (t+1) \\
\Rightarrow \\
& \exists t'. \ t' > t + 1 \land \text{HoldF} \ (t+1, t') \ \text{done} \land \text{done} \ t' \land (\text{out} \ t' = f(\text{inp} \ (t+1))) \land \\
& (\forall t. \ \text{done} \ t \land \neg (\text{Posedge} \ \text{load} \ (t+1)) \Rightarrow \text{done} \ (t+1)) \land \\
& (\forall t. \ \neg (\text{done} \ t) \Rightarrow \exists t'. \ t' > t \land \text{done} \ t')
\end{align*}
\]

The first conjunct in the right-hand side specifies that if the device is available and a positive edge occurs on \( \text{load} \), there exists a time \( t' \) in future when \( \text{done} \) 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 \( \textit{done} \) holds the value \( F \) during the computation. The second conjunct specifies the situation where no call is made on \( \textit{load} \) 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.

\[ \begin{align*}
\vdash & \text{AND} \ (in_1, in_2, out) = \forall t. \ out \ t = (in_1 \ t \ \land \ in_2 \ t) \\
\vdash & \text{OR} \ (in_1, in_2, out) = \forall t. \ out \ t = (in_1 \ t \ \lor \ in_2 \ t) \\
\vdash & \text{NOT} \ (inp, out) = \forall t. \ out \ t = \neg (inp \ t) \\
\vdash & \text{MUX} (sw, in_1, in_2, out) = \forall t. \ out \ t = \text{if } sw \ t \ \text{then } in_1 \ t \ \text{else } in_2 \ t \\
\vdash & \text{COMB} f \ (inp, out) = \forall t. \ out \ t = f(inp \ t) \\
\vdash & \text{DEL} \ (inp, out) = \forall t. \ out(t+1) = inp \ t \\
\vdash & \text{DELT} \ (inp, out) = (out \ 0 = T) \ \land \ \forall t. \ out(t+1) = inp \ t \\
\vdash & \text{DFF}(d, sel, q) = \forall t. \ q(t+1) = \text{if } \text{Posedge} \ sel \ (t+1) \ \text{then } d(t+1) \ \text{else } q \ t \\
\vdash & \text{POSEEDGE}(inp, out) = \exists c_0 \ c_1. \ \text{DELT}(inp, c_0) \ \land \ \text{NOT}(c_0, c_1) \ \land \ \text{AND}(c_1, inp, out)
\end{align*} \]

Atomic handshaking devices.

\[ \begin{align*}
\vdash & \text{ATM} f \ (load, inp, done, out) = \\
& \exists c_0 \ c_1. \ \text{POSEEDGE}(load, c_0) \ \land \ \text{NOT}(c_0, done) \ \land \\
& \text{COMB} f \ (inp, c_1) \ \land \ \text{DEL}(c_1, out)
\end{align*} \]

Sequential composition of handshaking devices.

\[ \begin{align*}
\vdash & \text{SEQ} f \ g \ (load, inp, done, out) = \\
& \exists c_0 \ c_1 \ c_2 \ c_3 \ data. \\
& \text{NOT}(c_2, c_3) \ \land \ \text{OR}(c_3, load, c_0) \ \land \ f(c_0, inp, c_1, data) \ \land \\
& g(c_1, data, c_2, out) \ \land \ \text{AND}(c_1, c_2, done)
\end{align*} \]

Parallel composition of handshaking devices.

\[ \begin{align*}
\vdash & \text{PAR} f \ g \ (load, inp, done, out) = \\
& \exists c_0 \ c_1 \ \text{start} \ \text{done}_1 \ \text{done}_2 \ \text{data}_1 \ \text{data}_2 \ \text{out}_1 \ \text{out}_2. \\
& \text{POSEEDGE}(load, c_0) \ \land \ \text{DEL}(\text{done}_1, c_1) \ \land \ \text{AND}(c_0, c_1, \text{start}) \ \land \\
& f(\text{start}, inp, \text{done}_1, data_1) \ \land \ g(\text{start}, inp, \text{done}_2, data_2) \ \land \\
& \text{DFF}(data_1, \text{done}_1, \text{out}_1) \ \land \ \text{DFF}(data_2, \text{done}_2, \text{out}_2) \ \land \\
& \text{AND}(\text{done}_1, \text{done}_2, done) \ \land \ (\text{out} = \lambda t. \ (out_1 \ t, out_2 \ t))
\end{align*} \]
Conditional composition of handshaking devices.

\[ \text{ITE } e \ f \ g \ (\text{load}, \text{inp}, \text{done}, \text{out}) = \]
\[ \exists c_0 \ c_1 \ c_2 \ \text{start} \ \text{start}' \ \text{done}_e \ \text{data}_e \ q \ \text{not}_e \ \text{data}_f \ \text{data}_g \ \text{sel} \]
\[ \text{done}_f \ \text{done}_g \ \text{start}_f \ \text{start}_g. \]
\[ \text{POSEDGE}(\text{load}, c_0) \ \land \ \text{DEL}(\text{done}, c_1) \ \land \ \text{AND}(c_0, c_1, \text{start}) \ \land \]
\[ e(\text{start}, \text{inp}, \text{done}_e, \text{data}_e) \ \land \ \text{POSEDGE}(\text{done}_e, \text{start}') \ \land \]
\[ \text{DFF}(\text{data}_e, \text{done}_e, \text{sel}) \ \land \ \text{DFF}(\text{inp}, \text{start}_e, q) \ \land \]
\[ \text{AND}(\text{start}', \text{data}_e, \text{start}_f) \ \land \ \text{NOT}(\text{data}_e, \text{not}_e) \ \land \]
\[ \text{AND}(\text{start}', \text{not}_e, \text{start}_g) \ \land \ f(\text{start}_f, q, \text{done}_f, \text{data}_f) \ \land \]
\[ g(\text{start}_g, q, \text{done}_g, \text{data}_g) \ \land \ \text{MUX}(\text{sel}, \text{data}_f, \text{data}_g, \text{out}) \ \land \]
\[ \text{AND}(\text{done}_e, \text{done}_f, c_2) \ \land \ \text{AND}(c_2, \text{done}_g, \text{done}) \]

Tail recursion constructor.

\[ \text{REC } e \ f \ g \ (\text{load}, \text{inp}, \text{done}, \text{out}) = \]
\[ \exists \text{done}_g \ \text{data}_g \ \text{start}_e \ q \ \text{done}_e \ \text{data}_e \ \text{start}_f \ \text{start}_g \ \text{inp}_e \ \text{done}_f \]
\[ c_0 \ c_1 \ c_2 \ c_3 \ c_4 \ \text{start} \ \text{sel} \ \text{start}' \ \text{not}_e. \]
\[ \text{POSEDGE}(\text{load}, c_0) \ \land \ \text{DEL}(\text{done}, c_1) \ \land \ \text{AND}(c_0, c_1, \text{start}) \ \land \]
\[ \text{OR}(\text{start}, \text{sel}, \text{start}_e) \ \land \ \text{POSEDGE}(\text{done}_g, \text{sel}) \ \land \]
\[ \text{MUX}(\text{sel}, \text{data}_g, \text{inp}_e, \text{start}_e, q) \ \land \]
\[ e(\text{start}_e, \text{inp}_e, \text{done}_e, \text{data}_e) \ \land \ \text{POSEDGE}(\text{done}_e, \text{start}') \ \land \]
\[ \text{AND}(\text{start}', \text{data}_e, \text{start}_f) \ \land \ \text{NOT}(\text{data}_e, \text{not}_e) \ \land \]
\[ \text{AND}(\text{not}_e, \text{start}', \text{start}_g) \ \land \ f(\text{start}_f, q, \text{done}_f, \text{out}) \ \land \]
\[ g(\text{start}_g, q, \text{done}_g, \text{data}_g) \ \land \ \text{DEL}(\text{done}_g, c_3) \ \land \]
\[ \text{AND}(\text{done}_g, c_3, c_4) \ \land \ \text{AND}(\text{done}_f, \text{done}_e, c_2) \ \land \ \text{AND}(c_2, c_4, \text{done}) \]

Circuit diagrams of the circuit constructors are shown on the following page.
Fig. 1. Implementation of composite devices.

Fig. 2. The conditional and the recursive constructors.