LAST REVISED June 2, 2005

A Proof-Producing Hardware Compiler for a Subset of Higher Order Logic

Mike Gordon, Juliano Iyoda
University of Cambridge
Computer Laboratory
William Gates Building
JJ Thomson Avenue
Cambridge CB3 0FD, UK

Scott Owens, Konrad Slind
University of Utah
School of Computing
50 South Central Campus Drive
Salt Lake City
Utah UT84112, USA

(*authors listed in alphabetical order*)

Abstract. Higher order logic (HOL) is a modelling language suitable for specifying behaviour at many levels of abstraction. We describe a compiler from a 'synthesisable subset' of HOL function definitions to correct-by-construction clocked synchronous hardware. The compiler works by theorem proving in the HOL4 system and goes through several phases, each deductively refining the specification to a more concrete form, until a representation that corresponds to hardware is deduced. It also produces a proof that the generated hardware implements the HOL functions constituting the specification. Synthesised designs can be translated to Verilog HDL, simulated and then input to standard design automation tools. Users can modify the theorem proving scripts that perform compilation. A simple example is adding rewrites for peephole optimisation, but all the theorem-proving infrastructure in HOL4 is available for tuning the compilation. Users can also extend the synthesisable subset. For example, the core system can only compile tail-recursions, but a 'third-party' tool `linRec` is being developed to automatically generate tail recursive definitions to implement linear recursions, thereby extending the synthesisable subset of HOL to include linear recursion.

1 Introduction

In the HOL4 proof system for higher order logic, a function f satisfying an equation $f(x) = e$, which may be recursive, is defined by executing:

`Define` ‘$f(x) = e$‘

We describe an extension to `Define`, called `cirDefine`, that generates hardware implementations for a 'synthesisable subset' of higher order logic. Executing:

`cirDefine` ‘$f(x) = e$‘

first defines f (i.e. invokes `Define`) and then automatically generates an implementation of f as a circuit suitable for executing in hardware.

The synthesised circuit is generated by proof and is thus correct by construction. A correctness theorem is proved by `cirDefine` for each circuit generated.

Our system is implemented in HOL4, but the ideas could be realised in other programmable proof systems.

In the next section we walk through a pedagogical example to illustrate the flow from higher order logic to Verilog HDL. We then outline how the proof-producing compiler works. It is a specialised theorem prover and we describe how deductions are used to generate hardware, and the sense in which synthesised circuits implement higher order logic functions. Next we discuss related work. Finally we outline future plans. An appendix contains technical details, including the formal description of the constructors used to build circuits and the four-phase handshake protocol that they implement.

2 Compiling higher order logic definitions to circuits

Let's dive in: `cirDefine` is an extension of HOL4's standard command `Define` for defining functions. Before using `cirDefine` one needs to load appropriate modules and start a new theory in which to store definitions and theorems. Assume this is done, and also assume that the `word32` library [6] is loaded, so that arithmetic operations like `+` and `-` default to 32-bit versions. Numerals like `0w` and `1w` denote the appropriate 32-bit word values, and `w2n:word32->num` converts a 32-bit word to a natural number. The HOL4 top level is Standard ML (SML); consider the following declaration:

```
- val (Mult32Iter_def,Mult32Iter_ind,Mult32Iter_cir) =
   cirDefine
    ‘(Mult32Iter(m,n,acc) = if m = 0w then (0w,n,acc) else Mult32Iter(m-1w,n,n+acc))
     measuring (w2n o FST)‘;
```

The right hand side of the declaration applies the SML function `cirDefine` to an argument of the form ‘*equation* `measuring` *measure-function*‘. The term *equation* is a recursive definition of a function `Mult32Iter` that takes a triple of 32-words and returns another triple of the same type. The measure function[1] `w2n o FST` maps a triple of 32-bit words to the natural number denoted by the first member of the triple (it is used to show termination).

The result of `cirDefine` is a triple of theorems in higher order logic. The first component of the triple, which is bound to `Mult32Iter_def` in the declaration above, is the theorem resulting from applying the HOL4 function definition tool (`Define`), using the measure function to aid proof of termination. A side effect of this definition is to define `Mult32Iter` as a constant and prove the appropriate 'definitional' theorem, which is the theorem returned. Thus the first output from the input shown above would be:

```
> val Mult32Iter_def =
   |- Mult32Iter(m,n,acc) = (if m = 0w then (0w,n,acc) else Mult32Iter(m-1w,n,n+acc))
```

The second output is a theorem that is bound to `Mult32Iter_ind`. This is a custom induction principle for the constant `Mult32Iter` that can be used for proofs about it. We show this for completeness, but we do not discuss any proofs.

```
val Mult32Iter_ind =
 |- ∀P. (∀m n acc. (¬(m = 0w) ==> P(m-1w,n,n+acc)) ==> P(m,n,acc)) ==> ∀v v1 v2. P(v,v1,v2)
```

[1] Measure functions are not always necessary as they can be inferred using heuristics by the termination prover. We expect future releases of the compiler to figure out the measure function automatically for simple recursions like the one here.

The final component of the triple of theorems returned by `cirDefine` is the result of compiling the definition of `Mult32Iter` to a circuit. We show this now, and then follow it by some explanation.

```
val Mult32Iter_cir =
  |- 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 0w 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 0w 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 1w 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 Mult32Iter
         (load at clk, (inp1<>inp2<>inp3) at clk, done at clk, (out1<>out2<>out3) at clk)
```

This theorem, which is bound to the ML name `Mult32Iter_cir`, has the form:

`|- InfRise clk ==>` *circuit* `==>` *device specification*

The variable `clk` represents the clock and is modelled by a function from time (natural numbers) to clock values (Booleans). The term `InfRise clk` asserts that `clock` has an infinite number of rising edges. This is a standard precondition for temporal abstraction [10] and is needed because of the use of the `at`-operator (explained below) in the device specification.

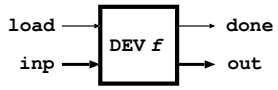
The *circuit* is a standard representation as a conjunction of component instances with internal lines existentially quantified (ibid). The components used here are described in Section 2.1. Circuits in this form are the lowest level of formal representation we generate. However they are easily converted 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 `Mult32Iter`) on FPGAs.

The *device specification* uses a HOL predicate `DEV` that specifies how a HOL function f is computed using a four-phase handshake. The general form is:

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

where a term of the form σ `at clk` denotes the signal consisting of the sequence of values of σ at successive rising edges of `clk`. This is a standard operation of

temporal abstraction (sometimes called clock projection and denoted by σ`@clk`). Temporal abstraction (projection) converts a signal σ representing the behavior of a wire (or bundle of wires) at the clocked circuit level to a signal σ `at clk` representing the behavior at the 'cycle level' – i.e. to an abstracted signal in which successive values represent the values during successive stable states of the circuit. The predicate `DEV` relates the values of signals at the abstracted level. A term `DEV` f `(load,inp,done,out)` specifies a handshaking device computing f where the signals `load`, `inp`, `done` and `out` are the handshake request line, the data input bus, the handshake acknowledge line and data output bus, respectively.

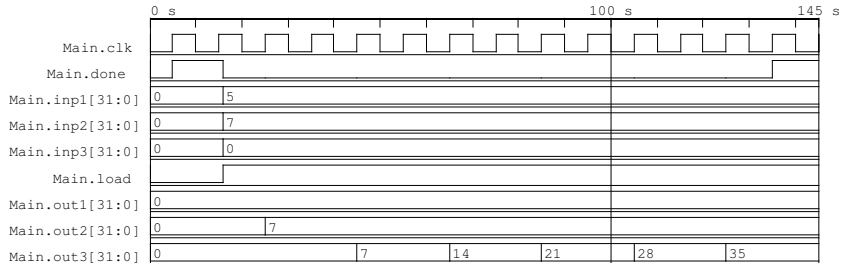


The behavior of such a handshaking device is formalised in the HOL definition of `DEV`, which says that if a value v is input on `inp` when a request is made on `load` then eventually $f(v)$ will be output on `out`, and when this occurs `T` is signalled on `done`. A formal specification of the handshake protocol is given in the Appendix. 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 (say time $t' > t$+1) the value $f(v)$ computed will be output on `out`. In the implementation generated by our compiler, `load`, `inp`, `done` and `out` are only sampled on rising edges of a clock `clk`, hence the behavior is specified by:

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

In `Mult32Iter_cir`, the lines `inp` and `out` carry triples of 32-bit words, which are represented by `inp1<>inp2<>inp3` and `out1<>out2<>out3` where `inp1`, `inp2`, `inp3`, `out1`, `out2`, `out3` are 32-bit busses and `<>` denotes word concatenation.

If we simulate our implementation of `Mult32Iter` with inputs $(5, 7, 0)$ using the Icarus Verilog simulator (http://www.icarus.com) and view the result with the GTKWave waveform viewer (http://home.nc.rr.com/gtkwave), the result is:



`load` is asserted at time 15 and `done` is `T` then, but `done` immediately drops to `F` in response to `load` being asserted. At the same time as `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.

2.1 Primitive 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 in `Mult32Iter_cir` 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: `0w` or `1w` are used in `Mult32Iter_cir`), `ADD32` (32-bit adder) and 32-bit `SUB32` (32-bit subtractor). Each of these components is defined in a standard style (ibid) in higher order logic. For example, `NOT` is defined by:

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

and the corresponding Verilog module definition that the compiler generates is

```
// Verilog module implementing HOL unary operator
// $~ :bool -> bool
//
// Automatically generated definition of NOT
module NOT (inp,out);
 parameter inpsize = 0;
 parameter outsize = 0;
 input   [inpsize:0] inp;
 output  [outsize:0] out;

 assign out = ! inp;

endmodule
```

An instance of a `NOT`-gate occurring in `Mult32Iter_cir` is `NOT(v51,v15)`, which is 'pretty-printed' as a module instance with unique name `NOT_12`:

```
/* NOT ((v51 :num -> bool),(v15 :num -> bool)) */
NOT        NOT_12 (v51,v15);
  defparam NOT_12.inpsize = 0;
  defparam NOT_12.outsize = 0;
```

Notice that comments are automatically generated in the Verilog showing the corresponding HOL source. This is so that manual inspection can be used to check that the Verilog is correct (a formal check is impossible, as there is no formal semantics of Verilog).

`NOT` is typical of all the combinational components. 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 `Rise` defined by:

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

and then `Dtype` and `DtypeT` are defined by:

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

which are coded in Verilog as:

```
// Positive edge triggered Dtype register
// Dtype(clk,d,q) = !t. q(t+1) = if Rise clk t then d t else q t
module Dtype (clk,d,q);
 parameter size = 31;
 input clk;
 input  [size:0] d;
 output [size:0] q;
 reg    [size:0] q;

 initial q = 0;

 always @(posedge clk) q <= d;

endmodule

// Boolean positive edge triggered flip-flop starting in state 1
// DtypeT(clk,d,q) = (q 0 = T) /\ Dtype(clk,d,q)
module DtypeT (clk,d,q);
 input clk,d;
 output q;
 reg q;

 initial q = 1;

 always @(posedge clk) q <= d;

endmodule
```

The reason for `initial q = 0` in the definition of the `Dtype` module is explained in Section 7. Since our proofs are valid for any initial value of `q`, the Verilog module `Dtype` is a valid implementation of the model in higher order logic.

Terms `Dtype(clk,v50,v14)` and `DtypeT(clk,v4,v53)` in `Mult32Iter_cir` generate named instances of these modules:

```
/* Dtype ((clk :num -> bool),(v50 :num -> word32),(v14 :num -> word32)) */
Dtype      Dtype_8 (clk,v50,v14);
  defparam Dtype_8.size = 31;

/* DtypeT ((clk :num -> bool),(v4 :num -> bool),(v53 :num -> bool)) */
DtypeT     DtypeT_8 (clk,v4,v53);
```

The automatically generated comments show the HOL source, to aid checking that the Verilog is correct.

2.2 Compiled components

After compiling `Mult32Iter`, its implementation is added to the library of components, so one can use it in subsequent compilations. For example, a multiplier using `Mult32Iter` could be defined by:

```
(*****************************************************************************)
(* Create an implementation of a multiplier from Mult32Iter                  *)
(*****************************************************************************)
val (Mult32,_,Mult32_cir) =
 cirDefine
  `Mult32(m,n) = SND(SND(Mult32Iter(m,n,0w)))`;
```

where `SND(SND(m,n,acc))` evaluates to `acc`. The compiler finds hardware implementing `Mult32Iter` in the component library and uses that to generate an implementation of `Mult32` (abbreviated to *Mult32Circuit* below):

```
|- InfRise clk
    ==> Mult32Circuit
    ==> DEV Mult32 (load at clk, (inp1<>inp2) at clk, done at clk, out at clk)
```

`Mult32` is then added to the component library and can be used in subsequent compilations, for example:

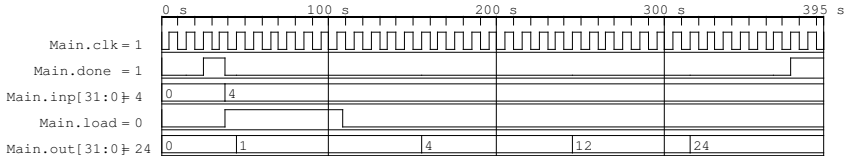
```
(*****************************************************************************)
(* Implement iterative function as a step to implementing factorial          *)
(*****************************************************************************)
val (Fact32Iter,Fact32Iter_ind,Fact32Iter_cir) =
 cirDefine
  `(Fact32Iter(n,acc) =
     if n = 0w then (n,acc) else Fact32Iter(n-1w, Mult32(n,acc)))
   measuring (w2n o FST)`;

(*****************************************************************************)
(* Implement a function Fact32 to compute SND(Fact32Iter (n,1))              *)
(*****************************************************************************)
val (Fact32,_,Fact32_cir) =
 cirDefine
  `Fact32 n = SND(Fact32Iter (n,1w))`;
```

This generates a circuit *Fact32Circuit* to compute the factorial function:

```
|- InfRise clk
    ==> Fact32Circuit
    ==> DEV Fact32 (load at clk, inp at clk, done at clk, out at clk)
```

The example waveform below shows that if 4 is input then 24 (i.e. 4!) is being output on `Main.out` when `Main.done` next goes high.



The 32-bit registers will overflow if one attempts to compute the factorial of n where $n > 12$. One can prove in HOL that:

$\vdash$ `∀n. (FACT n < 2^32) ==> (FACT n = w2n(Fact32(n2w n)))`

`FACT` is the factorial function on natural numbers and the value of `n2w n` is the 32-bit word representing `n` **mod** `32`. We have downloaded the Verilog version of *Fact32Circuit* onto an FPGA using Quartus II and verified that the factorial function is computed for `n` $\leq$ `12`, and that the expected wrap-around values are computed for `n > 12`.

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. Applying `cirDefine` to such a definition (if necessary with a measure function to aid proof of termination) will first define f in higher order logic (using TFL [16]) 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* will be `inp1<>···<>inpm`, *outputs* will be `out1<>···<>outn` (with the type of `inp`i matching σ_i and the type of `out`j matching τ_j) and $circuit_f$ will be 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:

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

the theorem:

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

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 $\vdash Imp_C \Longrightarrow \texttt{DEV}\ f$, where 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 suitably instantiating the predicate variables P_1, P_2 and P_3, these theorems allow us to

construct an expression e_C 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 (i.e. can be implemented directly with logic gates without using registers) 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 $\texttt{Seq}\ f_1\ f_2$, 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:

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

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

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

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

$\texttt{COMB}\ f\ (v_1, v_2)$ drives v_2 with $f(v_1)$, i.e. $\texttt{COMB}\ f\ (v_1, v_2) = \forall t.\ v_2\ t = f(v_1\ t)$. The construction $\texttt{SEQ}\ d_1\ d_2$ introduces a handshake between the executions of d_1 and d_2, but $\texttt{PRECEDE}\ f\ d$ and $\texttt{FOLLOW}\ d\ f$ just 'wire' f before or after d, respectively, without introducing a handshake.

The result of Step 2 is a theorem $\vdash e_C \Longrightarrow \texttt{DEV}\ f$ 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 $\texttt{COMB}\ g$ (where g will be a function represented as a HOL λ-expression) and Dtype registers.

The next phase of compilation converts terms of the form $\texttt{COMB}\ g\ (inp, out)$ 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. $\wedge$, $\vee$ 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 straightforward recursive algorithm to synthesise combinational circuits. For example:

$$\begin{aligned}\vdash\ \texttt{COMB}\ (\lambda(m,n).\ (m < n,\ \ m{+}1))\ (inp1\texttt{<>}inp2, out1\texttt{<>}out2)\ \ =\\ \exists v0.\ \texttt{COMB}\ (<)\ (inp1\texttt{<>}inp2, out1)\ \wedge\ \texttt{CONSTANT}\ 1\ v0\ \wedge\\ \texttt{COMB}\ (+)\ (inp1\texttt{<>}v0, out2)\end{aligned}$$

where <> is bus concatenation, $\texttt{CONSTANT}\ 1\ v0$ drives $v0$ high continuously, and $\texttt{COMB}\ <$ and $\texttt{COMB}\ +$ 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 [10] using the theorems:

$\vdash$ `InfRise` *clk* $\Rightarrow$ $\forall d\ q.$ `Dtype`(clk, d, q) $\Rightarrow$ `DEL`$(d$ `at` *clk*, q `at` *clk*$)$

$\vdash$ `InfRise` *clk* $\Rightarrow$ $\forall d\ q.$ `DtypeT`(clk, d, q) $\Rightarrow$ `DELT`$(d$ `at` *clk*, q `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_f ==>
   DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

4 Third party 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 `Mult32Iter` and `Fact32Iter` from the more natural definitions:

```
Mult32(m,n) = if m = 0w then 0w else m+Mult32(m-1w,n)
Fact32 n    = if n = 0w then 1w else n*Fact32(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 against 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 Case studies

As part of a project to verify an ARM processor [5], a high-level model of the multiplication algorithm used by some ARM implementations was created in higher order logic. This is a Booth multiplier and we are using Fox's existing specification as an example for testing our compiler.

A more substantial example, being done at the University of Utah, is implementing the Advanced Encryption Standard (AES) [13] algorithm for private-key encryption. This specifies a multi-round algorithm with primitive computations based on finite field operations. 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. An encryption round performs the following transformations on a 4-by-4 matrix of input bytes:

1. application of *sbox*, an invertible function from bytes to bytes, to each byte;
2. a cyclical shift of each row;
3. multiplication of each column by a fixed degree 3 polynomial, with coefficients in the 256 element finite field, GF(2^8);
4. adding a key to the matrix with exclusive OR.

We are exploring various options for generating components either as separate handshaking designs or expanding them into combinational logic. We have also explored converting our high-level recursive specification of multiplication into a table lookup. The resulting verified tables can then be stored into a RAM or ROM device. For synthesizing the tables directly into hardware, we have automated the definition of a function on bytes as a balanced `if` expression, branching on each successive bit of its input.

```
0xB ** x = if WORD_BIT 7 x then
             if WORD_BIT 6 x then
               ...
                       if WORD_BIT 0 then 0xA3 else 0xA8
               ...
           else
             if WORD_BIT 6 x then
               ...
```

Our experience so far is positive: compiling implementations by deduction provides a secure and flexible framework for creating and optimising designs.

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.* [7] 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 [8]. Lava and Hydra are both hardware description languages embedded in Haskell whose programs consist of definitions of gates and their connections (netlists) [1, 12]. 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 μFP and Ruby adopt similar principles in hardware design [9, 15]. 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) [11], especially the ideas in Richard Sharp's PhD thesis [14]. 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 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. There were, however, some initial 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.

In the immediate future we plan to continue and complete the case studies described in Section 5.

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 bigger 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.

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. Online at: http://www.ubka.uni-karlsruhe.de/cgi-bin/psgunzip/1998/informatik/37/37.pdf.
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. Anthony C. J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, The Computer Laboratory, University of Cambridge, England, November 2002.
6. Anthony C. J. Fox. HOL n-bit word Library, February 2004. Documentation available with the HOL4 system (http://hol.sourceforge.net/).
7. 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.
8. Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990. Available on the Internet at: http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR323.
9. Geraint Jones and Mary Sheeran. Circuit design in Ruby. Lecture notes on Ruby from a summer school in Lyngby, Denmark., September 1990. Online at: http://www.cs.chalmers.se/~ms/papers.html.
10. Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.
11. 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.
12. 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.
13. United States National Institute of Standards and Technology. Advanced Encryption Standard. Web: http://csrc.nist.gov/encryption/aes/, 2001.
14. Richard Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2002.
15. 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.

16. 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.

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:

$$\vdash \texttt{Posedge}\ s\ t \quad = \texttt{if}\ t{=}0\ \texttt{then F else}\ (\neg\, s(t{-}1)\ \wedge\ s\ t\ )$$
$$\vdash \texttt{HoldF}\ (t_1, t_2)\ s = \forall t.\ t_1 \leq t < t_2\ \Rightarrow\ \neg(s\ t)$$

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.

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

Sequential composition of handshaking devices.

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

Parallel composition of handshaking devices.

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

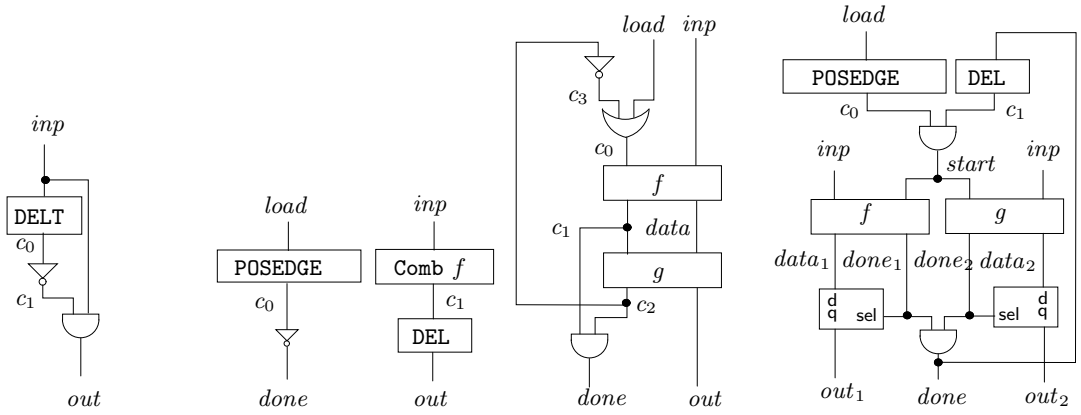
Conditional composition of handshaking devices.

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

Tail recursion constructor.

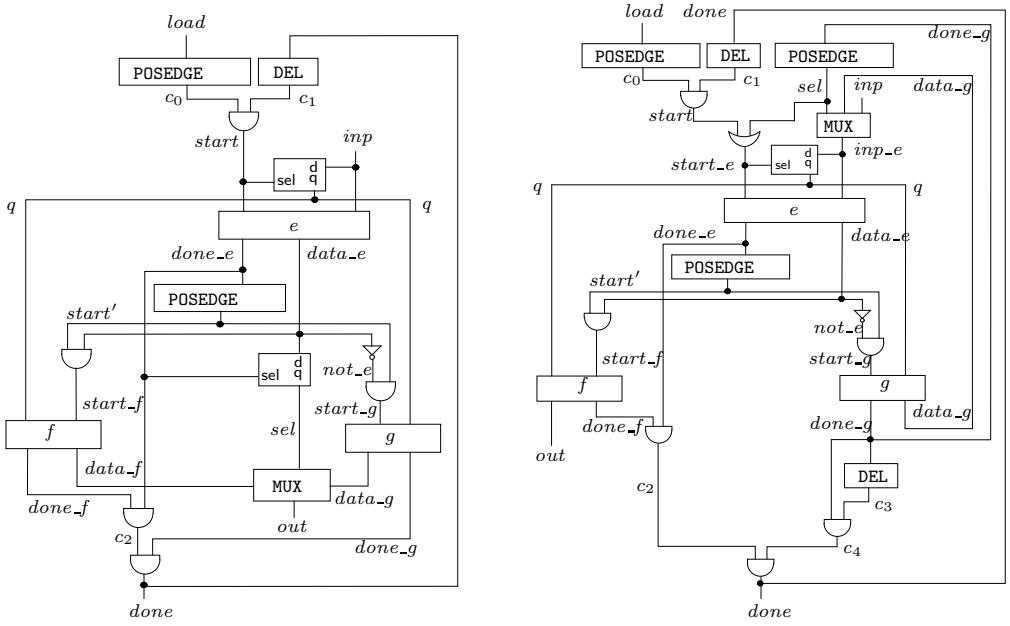
$$\vdash \texttt{REC}\ e\ f\ g\ (load, inp, done, out) =$$
$$\exists done\_g\ data\_g\ start\_e\ q\ done\_e\ data\_e\ start\_f\ start\_g\ inp\_e\ done\_f$$
$$c_0\ c_1\ c_2\ c_3\ c_4\ start\ sel\ start'\ not\_e.$$
$$\texttt{POSEDGE}(load, c_0)\ \wedge\ \texttt{DEL}(done, c_1)\ \wedge\ \texttt{AND}(c_0,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

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.