FAQ for Proof Producing Synthesis in HOL

Konrad Slind, Scott Owens, Juliano Iyoda, Mike Gordon
[Project web page: `http://www.cl.cam.ac.uk/~mjcg/dev/`]

1 What is proof producing synthesis?

Proof producing synthesis compiles a source specification (see 2) to an implementation and generates a theorem certifying that the implementation is correct. The specification is expressed in higher order logic.

2 What is the synthesisable subset of HOL?

The compiler automatically translates functions $f : \sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n$, where the argument ($\sigma_i$) and result ($\tau_j$) types are words. It can translate any tail recursive definition of such a function as long as the sub-functions used in the definition are in the library of primitive or previously defined functions. Formal refinement into this subset is by proof in the HOL4 system (13, 14, 31, 30, 34 have more discussion and examples).

3 Why not verify synthesis functions?

Synthesis functions would need to be coded inside higher order logic if they were to be proved correct. This would be impractical as the compiler uses many HOL4 system tools to automatically infer circuits – it would not be feasible to represent these tools (a substantial chunk of the HOL4 theorem proving infrastructure) in higher order logic.

4 Is proof producing synthesis really theorem-proving?

The compiler that synthesises circuits is a derived proof rule in the HOL4 system which is implemented by rewriting and a variety of custom proof strategies. It is a special purpose automatic theorem prover for proving correctness certifying theorems (see 12).

5 Is proof producing synthesis the same as formal synthesis?

Proof producing synthesis is a kind of formal synthesis [14] in which the synthesised circuit is not only formally inferred from the specification, but, in addition, a certifying theorem is produced (see 38 also).

6 Are there benefits of formal synthesis besides assurance?

Formal synthesis by theorem proving ensures that circuits are correct by construction. Users can safely tinker with the proof scripts used by the compiler, confident that they cannot produce incorrect implementations. Users familiar with the underlying HOL4 theorem proving infrastructure can easily experiment with application-specific extensions or optimisations. An example of an optimisation is combinational inlining (see 26). An example of an extension is `let`-expressions (see 30). Safe extensibility is thus a benefit.

7 Why use proof producing synthesis for crypto hardware?

Implementations of cryptographic algorithms are evaluated to a high standard of assurance such as Common Criteria Evaluation Assurance Level 7 (EAL7) [4, 5.9]. Formal methods are an established technique in this area. Proof producing synthesis provides a new way of certifying that cryptographic hardware implements high level specifications.

8 Formal synthesis is an old idea, so why is it still interesting?

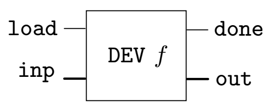
In the past it has been hard to justify the high cost of formal synthesis – the additional confidence of correctness it produces has not been considered worth the expense. However, we think there are niche applications (see 7) where the approach could be cost effective, because it may make it easier to achieve required levels of assurance. Also, we think that safe extensibility (see 6) is a feature that is worth exploring more.

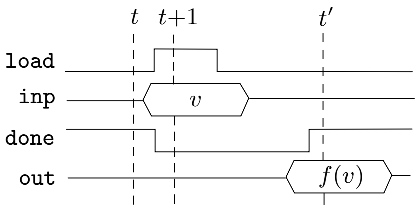
9 Is proof producing synthesis automatic?

The synthesis of clocked synchronous circuits from tail recursive definitions of functions mapping words to words is fully automatic. Currently users must manually refine specifications that use general recursion schemes to tail recursive form (there is a tool in development to automatically compile linear recursion to tail recursion). Data refinement from functions operating on types other than words must also be done manually.

10 What is the hardware realisation of a HOL function?

A function f defined in higher order logic is realised by a device `DEV` f that computes f via a four-phase handshake circuit on signals `load`, `inp`, `done` and `out`.





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, 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`, when the computed value $f(v)$ will be output on `out`. See 33 for an example.

11 What is the formal specification of a handshake device?

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, i.e. from F to T. The formula 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:

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

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

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

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.

12 What is the form of a correctness certifying theorem?

Synthesising a circuit implementing $f : \sigma_1 \times \cdots \times \sigma_m \to \tau_1 \times \cdots \times \tau_n$ (see 2) proves:

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

$\texttt{CIR}_f$ is a formula representing a circuit containing variables `clk`, `load`, `inp1`, ..., `inpm` representing inputs and variables `done`, `out1`, ..., `outn` representing outputs. The type of `inp`i matches σ_i and the type of `out`j matches τ_j. `InfRise clk` asserts that clock `clk` has infinitely many rising edges. See 31 and 34 for examples. The term *inputs* stands for `inp1<>`$\cdots$`<>inpm` which is the concatenation of the variables `inp1`, ..., `inpm` using the word concatenation operator `<>` and *outputs* is `out1<>`$\cdots$`<>outn` representing the concatenation of the output variables. A term s `at clk` is the temporal projection of signal s at rising edges of `clk` (see 15, 21).

13 Are specifications using high level datatypes synthesisable?

During synthesis the compiler generates circuits that use polymorphic registers and combinational components (i.e. components having inputs and outputs of arbitrary types). However, the lower level phases instantiate all types to words (currently represented as lists of bits). Thus we can generate circuits with wires carrying abstract values (e.g. numbers), but these cannot be refined to a form that can be input to FPGA tools. Our intention is that users will derive Boolean level specifications inside higher order logic using data-refinement methods. Automating this is a possible future direction.

14 Are there tools to translate into the synthesisable subset?

There is an experimental proof producing tool called `linRec` that translates linear recursions to tail recursions. For example it translates:

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

to:

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

15 What hardware components are used in circuits?

The compiler generates circuits built from a user-specifiable library of combinational components, e.g. `AND`, `OR`, `NOT`, `MUX`, `ADD` (the default library is chosen for use with the Quartus II FPGA software). Synthesised circuits may also contain constants (`CONSTANT`), edge-triggered D-type registers with unspecified initial state (`Dtype`) and Dtypes that power up into an initial state storing the value T (`DtypeT`). Constants and the registers are specified in higher order logic by:

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

where Rise $s\ t$ means signal s has a rising edge starting at time t:

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

Both `Dtype` and `DtypeT` are implemented in Verilog by instantiating a single generic register module `dtype` that is parametised on its size and initial stored value (see 32).

16 How much do you rely on untrustworthy FPGA tools?

The 'sign-off' from logic to EDA tools occurs at the clocked synchronous RTL level (see 15). A circuit $\texttt{CIR}_f$ (see 12) is translated to Verilog using a pretty-printer written in ML, and this Verilog is then fed to FPGA tools (e.g. Quartus II). FPGA implementations thus rely on the Verilog pretty-printer and the subsequent industrial tools. Higher assurance could be gained by taking the proof producing synthesis to a lower level (e.g. to an FPGA netlist language).

17 Can users control how specifications are synthesised?

The architecture of synthesised circuits reflects the input specification, so can be tuned by adjusting the higher order logic source. For example, using `let`-expressions (see 30) prevents logic blocks from being duplicated. There are also user-settable parameters: for example, modules can be inserted directly as combinational logic (i.e. without an enclosing handshake interface) if they are declared "combinational" (see 26, 28).

18 How efficient is proof producing synthesis?

Because synthesisers invokes a theorem prover it is relatively slow. Simple one-line examples take a few seconds on a standard workstation, bigger examples take several minutes.

19 How fast are synthesised circuits?

Some simple experiments comparing synthesised and hand coded circuits suggest that performance is not too bad, but we do not have solid evidence. However, the user has some control over the amount of computation per clock cycle via a facility to declare functions to be inlined as combinational logic, rather that via a handshake (see 28). We think the approach will support the creation of optimised implementations (necessary for crypto applications), but so far the emphasis has been on proof of concept.

20 How large are synthesised circuits?

Some of the bigger examples we have synthesised did not at first fit onto the FPGAs we are using. Whole program compaction (see 28) and use of `let` (see 30) solved the problem for these examples, but we still worry that the circuits are too big.

21 How does the hardware compiler work?

The operation of the compiler can be decomposed into four phases.

1. Translate $\forall x_1 \ldots x_n.\ f(x_1, \ldots, x_n) = e$ to an equivalent equation $f = \mathcal{E}$, where the expression, $\mathcal{E}$ is built from combinators Seq (compute in sequence), Par (compute in parallel), Ite (if-then-else) and Rec (recursion).
2. Replace the combinators Seq, Par, Ite and Rec with corresponding circuit constructors SEQ, PAR, ITE and REC to create a circuit term and a theorem that this implements DEV f.
3. Replace circuit constructors with cycle-level implementations and prove a theorem that the resulting design implements DEV f.
4. Introduce a clock and perform temporal projection [10] from cycle level to a clocked RTL circuit and prove a theorem that the resulting RTL circuit implements DEV f.

22 What are the combinators Seq, Par, Ite and Rec?

Seq, Par, Ite and Rec are used to build the combinatory expression $\mathcal{E}$ that is generated when translating $\forall x_1 \ldots x_n.\ f(x_1, \ldots, x_n) = e$ to $f = \mathcal{E}$. They are defined by:

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

For example:

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

is translated to:

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

23 What components do circuit constructors use?

The circuit constructors are built using the following components, which are represented at an unclocked cycle level of abstraction.

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

24 What are the circuit constructors SEQ, PAR, ITE and REC?

SEQ, PAR, ITE and REC are circuit constructors that implement Seq, Par, Ite and Rec, respectively (see 22). They construct circuits that combine delay elements with combinational logic. The delay elements are refined to clocked synchronous registers.

The circuit constructors are defined in higher order logic below. The components they use are defined in 15 and schematic diagrams of the implementations are in 29.

Sequential composition of handshaking devices.

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

Parallel composition of handshaking devices.

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

Conditional composition of handshaking devices.

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

Tail recursion constructor.

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

25 How are combinational circuits represented?

A function f can be packaged as a handshaking device with constructor ATM:

$$
\begin{array}{l}
\vdash \ \mathsf{ATM}\ f\ (load, inp, done, out) = \\
\quad \exists c_0\ c_1.\ \mathsf{POSEDGE}(load, c_0) \wedge \mathsf{NOT}(c_0, done) \wedge \\
\quad\quad\quad \mathsf{COMB}\ f\ (inp, c_1) \wedge \mathsf{DEL}(c_1, out)
\end{array}
$$

This creates a simple handshake interface that computes f and satisfies the refinement theorem: $\vdash\ \forall f.\ \mathsf{ATM}\ f \implies \mathsf{DEV}\ f$.

Formulas of the form COMB g (inp, out) are compiled into circuits built only using components in user-supplied library of predefined circuits. The default library currently includes 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:

$$
\begin{array}{l}
\vdash \ \mathsf{COMB}\ (\lambda(m, n).\ (m < n,\ \ m + 1))\ (inp_1\texttt{<>}inp_2, out_1\texttt{<>}out_2)\ = \\
\quad \exists v_0.\ \mathsf{COMB}\ (<)\ (inp_1\texttt{<>}inp_2, out_1)\ \wedge\ \mathsf{CONSTANT}\ 1\ v_0\ \wedge \\
\quad\quad \mathsf{COMB}\ (+)\ (inp_1\texttt{<>}v_0, out_2)
\end{array}
$$

where <> is bus concatenation, CONSTANT 1 v_0 drives v_0 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).

26 How is an explosion of internal handshakes avoided?

When processing Seq f_1 f_2 (see 27), the compiler 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:

$$
\begin{array}{l}
\vdash\ (P \implies \mathsf{DEV}\ f_2)\ \Rightarrow\ (\mathsf{PRECEDE}\ f_1\ P \implies \mathsf{DEV}\ (\mathsf{Seq}\ f_1\ f_2)) \\
\vdash\ (P \implies \mathsf{DEV}\ f_1)\ \Rightarrow\ (\mathsf{FOLLOW}\ P\ f_2 \implies \mathsf{DEV}\ (\mathsf{Seq}\ f_1\ f_2))
\end{array}
$$

where 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:

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

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 without introducing a handshake.

27 How are the circuit constructors introduced?

The following theorems enable the compiler to compositionally deduce theorems of the form $\vdash Imp \Longrightarrow$ DEV f, where Imp is a formula constructed using the circuit constructors. The long arrow symbol $\Longrightarrow$ denotes implication lifted to functions:
$f \Longrightarrow g \ = \ \forall \textit{load inp done out}.\ f(\textit{load}, \textit{inp}, \textit{done}, \textit{out}) \Rightarrow g(\textit{load}, \textit{inp}, \textit{done}, \textit{out})$.

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

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

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

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

The predicate Total is defined so that Total(f_1, f_2, f_3) ensures termination.

If $\mathcal{E}$ 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 a logic formula $\mathcal{F}$ to be built from circuit constructors SEQ, PAR, ITE and REC such that $\vdash \mathcal{F} \Longrightarrow$ DEV $\mathcal{E}$. We have $\vdash f = \mathcal{E}$ (see 21, phase 1), hence $\vdash \mathcal{F} \Longrightarrow$ DEV f. This is the basic idea, but see also 26, 28.

28 What optimisation does the compiler perform?

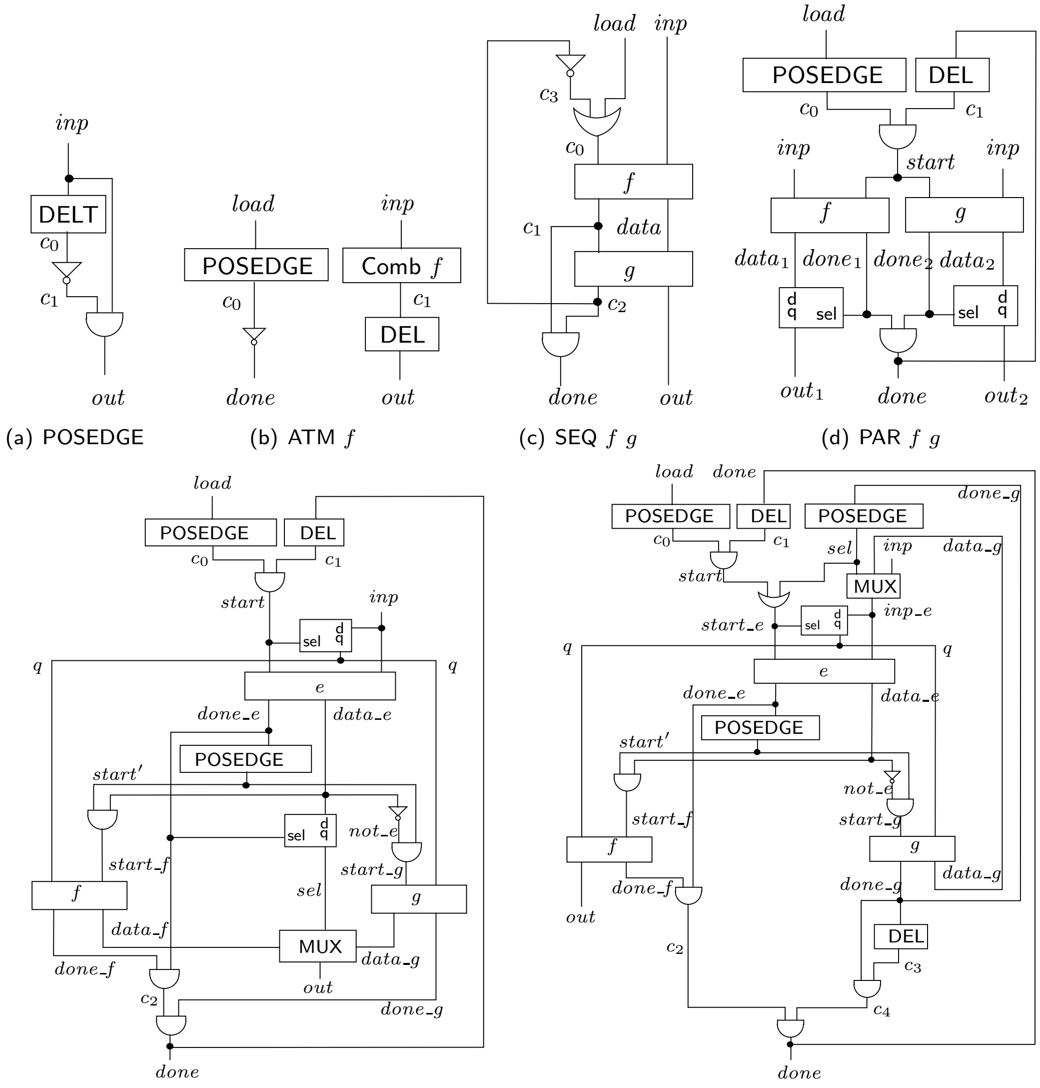
There are currently two main optimisations used in synthesis: reducing handshakes between implementations of functions (see 26) and whole program compaction.

The normal synthesis of Seq f_1 f_2 (see 22) is to SEQ d_1 d_2 (see 24), where the circuit combinator SEQ puts a handshake between the device d_1 implementing f_1 and the device d_2 implementing f_2. Users may declare f_1 to be 'combinational' and then a combinational logic block c_1 implementing f_1 is synthesised and put in series before d_2. Similarly f_2 can be declared combinational and then combinational logic c_2 will be put in series after d_1. Only components that can be realised using known combinational logic blocks can be declared combinational. Using this mechanism, all the computation from the arguments of a function to its recursive call can be synthesised as combinational logic, so there is just a single handshake to manage the iteration.

This optimisation is restricted to the scope of a single function. However, before applying this method, we can inline the function calls to produce a system defined by a single function (the function 'main'). The whole program compaction eliminates unnecessary handshake circuits that could have been generated to implement the function calls.

In practise these techniques can generate long logic paths (i.e. slow clocks), so some pipelining via internal handshakes can be appropriate.

29 What do the circuit constructor implementations look like?



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

(e) IF e f g (f) REC e f g

30 How do `let`-expressions work?

A `let`-expression has the form `let` v `=` e_1 `in` e_2 where v is a "varstruct" (variable structure) which is either a single variable or, recursively, a non-empty tuple of varstructs (e.g. `(x,(m,n),y)`).

If e_1 is combinational, then a `let`-expression is synthesised into a circuit consisting of e_1 driving wires corresponding to v that are inputs to the circuit corresponding to e_2.

If e_1 is not combinational, the `let`-expression is compiled using:

$\vdash\ \forall f_1\ f_2.\ (\lambda x.\ \texttt{let}\ v\ =\ f_1\ x\ \texttt{in}\ f_2\ (x,v))\ =\ \mathsf{Seq}\ (\mathsf{Par}\ (\lambda x.\ x)\ f_1)\ f_2$

For example, suppose H and J are defined by:

```
H x = x+1w
J x = let y = H x in y + y + y
```

where 1w is the 32-bit word denoting 1 and + is 32-bit addition.

If H is not declared combinational, then J compiles to:

```
|- 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.
      DtypeT (clk,load,v10) ∧ NOT (v10,v9) ∧ AND (v9,load,v8) ∧ Dtype (clk,done,v7) ∧
      AND (v8,v7,v6) ∧ DtypeT (clk,v6,v13) ∧ NOT (v13,v12) ∧ AND (v12,v6,v11) ∧
      NOT (v11,v5) ∧ Dtype (clk,inp,v3) ∧ DtypeT (clk,v6,v17) ∧ NOT (v17,v16) ∧
      AND (v16,v6,v15) ∧ NOT (v15,v4) ∧ CONSTANT 1w v18 ∧ ADD32 (inp,v18,v14) ∧
      Dtype (clk,v14,v2) ∧ DtypeT (clk,v5,v21) ∧ NOT (v21,v20) ∧ AND (v20,v5,v19) ∧
      MUX (v19,v3,v22,v1) ∧ Dtype (clk,v1,v22) ∧ DtypeT (clk,v4,v25) ∧ NOT (v25,v24) ∧
      AND (v24,v4,v23) ∧ MUX (v23,v2,v26,v0) ∧ Dtype (clk,v0,v26) ∧ AND (v5,v4,done) ∧
      ADD32 (v0,v0,v27) ∧ ADD32 (v27,v0,out)) ==>
   DEV J (load at clk,inp at clk,done at clk,out at clk)
```

but if H is declared to be combinational, then J compiles to:

```
|- InfRise clk ==>
   (∃v0 v1 v2 v3 v4 v5 v6.
      DtypeT (clk,load,v3) ∧ NOT (v3,v2) ∧ AND (v2,load,v1) ∧ NOT (v1,done) ∧
      CONSTANT 1w v5 ∧ ADD32 (inp,v5,v4) ∧ ADD32 (v4,v4,v6) ∧ ADD32 (v6,v4,v0) ∧
      Dtype (clk,v0,out)) ==>
   DEV J (load at clk,inp at clk,done at clk,out at clk)
```

31 What is a simple example?

A simple example is iterative accumulator-style multiplication on 32-bit words:

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

where 0w, 1w are 32-bit numbers and +, - are 32-bit operations.

This specification compiles to:

```
|- 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)
```

See 12 and 15 for explanations and 32 for the Verilog that is extracted from this circuit.

32 What is the generated Verilog like?

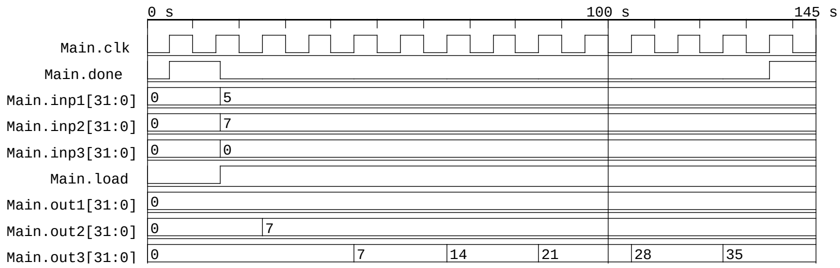
The iterative accumulator-style multiplication device (see 31) generates the following Verilog.

```
module dtype (clk,d,q);
 parameter size = 31; parameter value = 1;
 input clk; input [size:0] d; output [size:0] q; reg [size:0] q = value;

 always @(posedge clk) q <= d;

endmodule

module Mult32Iter (clk,load,inp1,inp2,inp3,done,out1,out2,out3);
 input clk,load; input [31:0] inp1; input [31:0] inp2; input [31:0] inp3;
 output done; output [31:0] out1; output [31:0] out2; output [31:0] out3;
 wire clk,done; wire [0:0] v0; wire [31:0] v1; wire [31:0] v2; wire [31:0] v3; wire [0:0] v4;
 wire [0:0] v5; wire [0:0] v6; wire [0:0] v7; wire [31:0] v8; wire [31:0] v9; wire [31:0] v10;
 wire [0:0] v11; wire [31:0] v12; wire [31:0] v13; wire [31:0] v14; wire [0:0] v15;
 wire [0:0] v16; wire [0:0] v17; wire [0:0] v18; wire [0:0] v19; wire [0:0] v20; wire [0:0] v21;
 wire [0:0] v22; wire [0:0] v23; wire [0:0] v24; wire [0:0] v25; wire [0:0] v26; wire [31:0] v27;
 wire [0:0] v28; wire [0:0] v29; wire [0:0] v30; wire [31:0] v31; wire [0:0] v32; wire [0:0] v33;
 wire [0:0] v34; wire [31:0] v35; wire [0:0] v36; wire [0:0] v37; wire [0:0] v38; wire [0:0] v39;
 wire [31:0] v40; wire [0:0] v41; wire [0:0] v42; wire [0:0] v43; wire [0:0] v44; wire [31:0] v45;
 wire [0:0] v46; wire [0:0] v47; wire [0:0] v48; wire [31:0] v49; wire [31:0] v50; wire [0:0] v51;
 wire [0:0] v52; wire [0:0] v53; wire [31:0] v54; wire [0:0] v55; wire [0:0] v56; wire [0:0] v57;

 dtype   dtype_0 (clk,load,v21);   defparam dtype_0.size = 0;
 assign v20 = ~ v21;
 assign v19 = v20 && load;
 dtype   dtype_1 (clk,done,v18);   defparam dtype_1.size = 0;
 assign v17 = v19 && v18;
 assign v11 = v17 || v16;
 dtype   dtype_2 (clk,v15,v23);    defparam dtype_2.size = 0;
 assign v22 = ~ v23;
 assign v16 = v22 && v15;
 assign v3 = v16 ? v14 : inp1;
 assign v2 = v16 ? v13 : inp2;
 assign v1 = v16 ? v12 : inp3;
 dtype   dtype_3 (clk,v11,v26);    defparam dtype_3.size = 0;
 assign v25 = ~ v26;
 assign v24 = v25 && v11;
 assign v10 = v24 ? v3 : v27;
 dtype   dtype_4 (clk,v10,v27);    defparam dtype_4.size = 31;
 dtype   dtype_5 (clk,v11,v30);    defparam dtype_5.size = 0;
 assign v29 = ~ v30;
 assign v28 = v29 && v11;
 assign v9 = v28 ? v2 : v31;
 dtype   dtype_6 (clk,v9,v31);     defparam dtype_6.size = 31;
 dtype   dtype_7 (clk,v11,v34);    defparam dtype_7.size = 0;
 assign v33 = ~ v34;
 assign v32 = v33 && v11;
 assign v8 = v32 ? v1 : v35;
 dtype   dtype_8 (clk,v8,v35);     defparam dtype_8.size = 31;
 dtype   dtype_9 (clk,v11,v39);    defparam dtype_9.size = 0;
 assign v38 = ~ v39;
 assign v37 = v38 && v11;
 assign v7 = ~ v37;
 assign v40 = 0;
 assign v36 = v3 == v40;
 dtype   dtype_10 (clk,v36,v6);    defparam dtype_10.size = 0;
 dtype   dtype_11 (clk,v7,v44);    defparam dtype_11.size = 0;
 assign v43 = ~ v44;
 assign v42 = v43 && v7;
 assign v5 = v42 && v6;
 assign v41 = ~ v6;
 assign v4 = v41 && v42;
 dtype   dtype_12 (clk,v5,v48);    defparam dtype_12.size = 0;
 assign v47 = ~ v48;
 assign v46 = v47 && v5;
 assign v0 = ~ v46;
 assign v45 = 0;
 dtype   dtype_13 (clk,v45,out1);  defparam dtype_13.size = 31;
 dtype   dtype_14 (clk,v9,out2);   defparam dtype_14.size = 31;
 dtype   dtype_15 (clk,v8,out3);   defparam dtype_15.size = 31;
 dtype   dtype_16 (clk,v4,v53);    defparam dtype_16.size = 0;
 assign v52 = ~ v53;
 assign v51 = v52 && v4;
 assign v15 = ~ v51;
 assign v54 = 1;
 assign v50 = v10 - v54;
 assign v49 = v9 + v8;
 dtype   dtype_17 (clk,v50,v14);   defparam dtype_17.size = 31;
 dtype   dtype_18 (clk,v9,v13);    defparam dtype_18.size = 31;
 dtype   dtype_19 (clk,v49,v12);   defparam dtype_19.size = 31;
 dtype   dtype_20 (clk,v15,v56);   defparam dtype_20.size = 0;
 assign v55 = v15 && v56;
 assign v57 = v0 && v7;
 assign done = v57 && v55;

endmodule
```

To conserve space, all comments and many line breaks have been removed from the preceding Verilog. Each Verilog statement is printed with a comment showing the HOL source to aid visual checking (e.g. for Common Criteria EAL7 certification evaluators [4, 5.9]). We are still tinkering with the Verilog: for example, experiments show that Quartus II configures Altera FPGAs to initialise faster with registers as instances of a separate module (`dtype` above) than with inlined behavioral statements `always @(posedge clk) q <= d`, where `q` is initialised with declaration `reg q = 1`.

33 What simulation tools have you used?

We currently use Icarus Verilog (`http://www.icarus.com`) for simulation and then view waveforms with GTKWave (`http://home.nc.rr.com/gtkwave`). These tools are both public domain. If we simulate the `Mult32Iter` example (see 31) with inputs $(5, 7, 0)$, then the resulting waveform 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.

34 What is a bigger example?

An example drawn from cryptography is the TEA block cipher [17]. The encryption algorithm is described by the following HOL definitions (all variables are 32-bit words):

```
TEAEncrypt (keys,txt) = FST(Rounds (32,(txt,keys,0)))

Rounds (n,s) = if n=0 then s else Rounds(n-1, Round s)

Round ((y,z),(k0,k1,k2,k3),s) =
  let s' = s + DELTA in
  let t = y + ShiftXor(z,s',k0,k1)
  in
     ((t, z + ShiftXor(t,s',k2,k3)), (k0,k1,k2,k3), s')

ShiftXor (x,s,k0,k1) = ((x << 4) + k0) # (x + s) # ((x >> 5) + k1)

DELTA = 0x9e3779b9w
```

There is a corresponding `TEADecrypt` function (omitted). In HOL-4 we can prove functional correctness:

```
|- ∀plaintext keys. TEADecrypt(keys,TEAEncrypt(keys,plaintext)) = plaintext
```

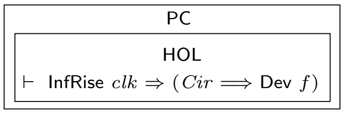
The compiler generates the following netlist for `TEAEncrypt` and also one for `TEADecrypt`:

```
|- 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 v58 v59 v60 v61 v62 v63 v64 v65 v66 v67 v68 v69
     v70 v71 v72 v73 v74 v75 v76 v77 v78 v79 v80 v81 v82 v83 v84 v85 v86
     v87 v88 v89 v90 v91 v92 v93 v94 v95 v96 v97 v98 v99 v100 v101 v102
     v103 v104 v105 v106 v107 v108 v109 v110 v111 v112 v113 v114 v115 v116
     v117 v118 v119 v120 v121 v122 v123 v124 v125 v126 v127 v128 v129 v130
     v131 v132 v133 v134 v135 v136 v137 v138 v139 v140 v141 v142 v143 v144
     v145 v146 v147 v148 v149 v150 v151 v152 v153 v154 v155 v156 v157 v158
     v159 v160 v161 v162 v163 v164 v165 v166 v167 v168 v169 v170 v171 v172
     v173 v174 v175 v176 v177 v178 v179 v180 v181 v182 v183 v184 v185 v186
     v187 v188 v189 v190 v191 v192 v193 v194 v195 v196 v197 v198 v199 v200
     v201 v202 v203 v204 v205 v206 v207 v208 v209 v210 v211 v212 v213 v214
     v215 v216 v217 v218 v219 v220 v221 v222 v223 v224 v225 v226 v227 v228
     v229 v230 v231 v232 v233 v234 v235 v236 v237 v238 v239 v240 v241 v242
     v243 v244 v245 v246 v247 v248 v249 v250 v251 v252 v253 v254 v255 v256
     v257 v258 v259 v260 v261 v262 v263 v264 v265 v266 v267 v268 v269 v270
     v271 v272 v273 v274 v275 v276 v277 v278 v279 v280 v281 v282 v283 v284
     v285 v286 v287 v288 v289 v290 v291 v292 v293 v294 v295 v296 v297 v298
     v299 v300 v301 v302 v303 v304 v305 v306 v307 v308 v309 v310 v311 v312
     v313 v314 v315 v316 v317 v318 v319 v320 v321 v322 v323 v324 v325 v326
     v327 v328 v329 v330 v331 v332 v333 v334 v335 v336 v337 v338 v339 v340
     v341 v342 v343 v344 v345 v346 v347 v348 v349 v350 v351 v352 v353 v354.
       CONSTANT 32w v6 ∧ CONSTANT 0w v5 ∧ DtypeT (clk,load,v43) ∧
       NOT (v43,v42) ∧ AND (v42,load,v41) ∧ Dtype (clk,done,v40) ∧
       AND (v41,v40,v39) ∧ OR (v39,v38,v28) ∧ DtypeT (clk,v37,v45) ∧
       NOT (v45,v44) ∧ AND (v44,v37,v38) ∧ MUX (v38,v36,v6,v15) ∧
       MUX (v38,v35,inp2,v14) ∧ MUX (v38,v34,inp3,v13) ∧ MUX (v38,v33,inp11,v12) ∧
       MUX (v38,v32,inp12,v11) ∧ MUX (v38,v31,inp13,v10) ∧ MUX (v38,v30,inp14,v9) ∧
       MUX (v38,v29,v5,v8) ∧ DtypeT (clk,v28,v48) ∧ NOT (v48,v47) ∧
       AND (v47,v28,v46) ∧ MUX (v46,v15,v49,v27) ∧ Dtype (clk,v27,v49) ∧
       DtypeT (clk,v28,v52) ∧ NOT (v52,v51) ∧ AND (v51,v28,v50) ∧
       MUX (v50,v14,v53,v26) ∧ Dtype (clk,v26,v53) ∧ DtypeT (clk,v28,v56) ∧
       NOT (v56,v55) ∧ AND (v55,v28,v54) ∧ MUX (v54,v13,v57,v25) ∧
       Dtype (clk,v25,v57) ∧ DtypeT (clk,v28,v60) ∧ NOT (v60,v59) ∧
       AND (v59,v28,v58) ∧ MUX (v58,v12,v61,v24) ∧ Dtype (clk,v24,v61) ∧
       DtypeT (clk,v28,v64) ∧ NOT (v64,v63) ∧ AND (v63,v28,v62) ∧
       MUX (v62,v11,v65,v23) ∧ Dtype (clk,v23,v65) ∧ DtypeT (clk,v28,v68) ∧
       NOT (v68,v67) ∧ AND (v67,v28,v66) ∧ MUX (v66,v10,v69,v22) ∧
       Dtype (clk,v22,v69) ∧ DtypeT (clk,v28,v72) ∧ NOT (v72,v71) ∧
       AND (v71,v28,v70) ∧ MUX (v70,v9,v73,v21) ∧ Dtype (clk,v21,v73) ∧
       DtypeT (clk,v28,v76) ∧ NOT (v76,v75) ∧ AND (v75,v28,v74) ∧
       MUX (v74,v8,v77,v20) ∧ Dtype (clk,v20,v77) ∧ DtypeT (clk,v28,v81) ∧
       NOT (v81,v80) ∧ AND (v80,v28,v79) ∧ NOT (v79,v19) ∧ CONSTANT 0w v82 ∧
       EQ32 (v15,v82,v78) ∧ Dtype (clk,v78,v18) ∧ DtypeT (clk,v19,v86) ∧
       NOT (v86,v85) ∧ AND (v85,v19,v84) ∧ AND (v84,v18,v17) ∧ NOT (v18,v83) ∧
       AND (v83,v84,v16) ∧ DtypeT (clk,v17,v89) ∧ NOT (v89,v88) ∧ AND (v88,v17,v87) ∧
       NOT (v87,v7) ∧ Dtype (clk,v26,out1) ∧ Dtype (clk,v25,out2) ∧ Dtype (clk,v24,v4) ∧
       Dtype (clk,v23,v3) ∧ Dtype (clk,v22,v2) ∧ Dtype (clk,v21,v1) ∧
       Dtype (clk,v20,v0) ∧ DtypeT (clk,v16,v104) ∧ NOT (v104,v103) ∧
       AND (v103,v16,v102) ∧  Dtype (clk,v37,v101) ∧ AND (v102,v101,v100) ∧
       DtypeT (clk,v100,v108) ∧ NOT (v108,v107) ∧ AND (v107,v100,v106) ∧
       NOT (v106,v99) ∧ CONSTANT 1w v109 ∧ SUB32 (v27,v109,v105) ∧
       Dtype (clk,v105,v97) ∧ DtypeT (clk,v100,v123) ∧ NOT (v123,v122) ∧
       AND (v122,v100,v121) ∧ Dtype (clk,v98,v120) ∧ AND (v121,v120,v119) ∧
       DtypeT (clk,v119,v132) ∧ NOT (v132,v131) ∧ AND (v131,v119,v130) ∧
       Dtype (clk,v118,v129) ∧ AND (v130,v129,v128) ∧ DtypeT (clk,v128,v143) ∧
       NOT (v143,v142) ∧ AND (v142,v128,v141) ∧ Dtype (clk,v127,v140) ∧
       AND (v141,v140,v139) ∧ DtypeT (clk,v139,v146) ∧ NOT (v146,v145) ∧
       AND (v145,v139,v144) ∧ NOT (v144,v138) ∧ Dtype (clk,v26,v136) ∧
       CONSTANT 2654435769w v148 ∧ ADD32 (v20,v148,v147) ∧ DtypeT (clk,v139,v152) ∧
       NOT (v152,v151) ∧ AND (v151,v139,v150) ∧ NOT (v150,v137) ∧
       CONSTANT 4 v158 ∧ LSL32 (v25,v158,v157) ∧ ADD32 (v157,v24,v156) ∧
       ADD32 (v25,v147,v155) ∧ XOR32 (v156,v155,v154) ∧ CONSTANT 5 v160 ∧
       ASR32 (v25,v160,v159) ∧ ADD32 (v159,v23,v153) ∧ XOR32 (v154,v153,v149) ∧
       Dtype (clk,v149,v135) ∧ DtypeT (clk,v138,v163) ∧ NOT (v163,v162) ∧
       AND (v162,v138,v161) ∧ MUX (v161,v136,v164,v134) ∧ Dtype (clk,v134,v164) ∧
       DtypeT (clk,v137,v167) ∧ NOT (v167,v166) ∧ AND (v166,v137,v165) ∧
       MUX (v165,v135,v168,v133) ∧ Dtype (clk,v133,v168) ∧ AND (v138,v137,v127) ∧
       ADD32 (v134,v133,v125) ∧ DtypeT (clk,v128,v179) ∧ NOT (v179,v178) ∧
       AND (v178,v128,v177) ∧ Dtype (clk,v126,v176) ∧ AND (v177,v176,v175) ∧
       DtypeT (clk,v175,v182) ∧ NOT (v182,v181) ∧ AND (v181,v175,v180) ∧
       NOT (v180,v174) ∧ Dtype (clk,v25,v172) ∧ NOT (v187,v190) ∧
       OR (v190,v175,v189) ∧ DtypeT (clk,v189,v201) ∧ NOT (v201,v200) ∧
```

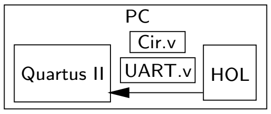
```
    AND (v200,v189,v199) ∧ Dtype (clk,v188,v198) ∧ AND (v199,v198,v197) ∧
    DtypeT (clk,v197,v212) ∧ NOT (v212,v211) ∧ AND (v211,v197,v210) ∧
    Dtype (clk,v196,v209) ∧ AND (v210,v209,v208) ∧ DtypeT (clk,v208,v215) ∧
    NOT (v215,v214) ∧ AND (v214,v208,v213) ∧ NOT (v213,v207) ∧
    Dtype (clk,v26,v205) ∧ CONSTANT 2654435769w v217 ∧ ADD32 (v20,v217,v216) ∧
    DtypeT (clk,v208,v221) ∧ NOT (v221,v220) ∧ AND (v220,v208,v219) ∧
    NOT (v219,v206) ∧ CONSTANT 4 v227 ∧ LSL32 (v25,v227,v226) ∧
    ADD32 (v226,v24,v225) ∧ ADD32 (v25,v216,v224) ∧ XOR32 (v225,v224,v223) ∧
    CONSTANT 5 v229 ∧ ASR32 (v25,v229,v228) ∧ ADD32 (v228,v23,v222) ∧
    XOR32 (v223,v222,v218) ∧ Dtype (clk,v218,v204) ∧ DtypeT (clk,v207,v232) ∧
    NOT (v232,v231) ∧ AND (v231,v207,v230) ∧ MUX (v230,v205,v233,v203) ∧
    Dtype (clk,v203,v233) ∧ DtypeT (clk,v206,v236) ∧ NOT (v236,v235) ∧
    AND (v235,v206,v234) ∧ MUX (v234,v204,v237,v202) ∧ Dtype (clk,v202,v237) ∧
    AND (v207,v206,v196) ∧ ADD32 (v203,v202,v194) ∧ DtypeT (clk,v197,v241) ∧
    NOT (v241,v240) ∧ AND (v240,v197,v239) ∧ NOT (v239,v195) ∧
    CONSTANT 2654435769w v242 ∧ ADD32 (v20,v242,v238) ∧ Dtype (clk,v238,v193) ∧
    Dtype (clk,v22,v192) ∧ Dtype (clk,v21,v191) ∧ DtypeT (clk,v196,v245) ∧
    NOT (v245,v244) ∧ AND (v244,v196,v243) ∧ MUX (v243,v194,v246,v186) ∧
    Dtype (clk,v186,v246) ∧ DtypeT (clk,v195,v249) ∧ NOT (v249,v248) ∧
    AND (v248,v195,v247) ∧ MUX (v247,v193,v250,v185) ∧ Dtype (clk,v185,v250) ∧
    DtypeT (clk,v195,v253) ∧ NOT (v253,v252) ∧ AND (v252,v195,v251) ∧
    MUX (v251,v192,v254,v184) ∧ Dtype (clk,v184,v254) ∧ DtypeT (clk,v195,v257) ∧
    NOT (v257,v256) ∧ AND (v256,v195,v255) ∧ MUX (v255,v191,v258,v183) ∧
    Dtype (clk,v183,v258) ∧ AND (v196,v195,v188) ∧ DtypeT (clk,v188,v262) ∧
    NOT (v262,v261) ∧ AND (v261,v188,v260) ∧ NOT (v260,v187) ∧
    CONSTANT 4 v268 ∧ LSL32 (v186,v268,v267) ∧ ADD32 (v267,v184,v266) ∧
    ADD32 (v186,v185,v265) ∧ XOR32 (v266,v265,v264) ∧ CONSTANT 5 v270 ∧
    ASR32 (v186,v270,v269) ∧ ADD32 (v269,v183,v263) ∧ XOR32 (v264,v263,v259) ∧
    Dtype (clk,v259,v171) ∧ AND (v188,v187,v173) ∧ DtypeT (clk,v174,v273) ∧
    NOT (v273,v272) ∧ AND (v272,v174,v271) ∧ MUX (v271,v172,v274,v170) ∧
    Dtype (clk,v170,v274) ∧ DtypeT (clk,v173,v277) ∧ NOT (v277,v276) ∧
    AND (v276,v173,v275) ∧ MUX (v275,v171,v278,v169) ∧ Dtype (clk,v169,v278) ∧
    AND (v174,v173,v126) ∧ ADD32 (v170,v169,v124) ∧ DtypeT (clk,v127,v281) ∧
    NOT (v281,v280) ∧ AND (v280,v127,v279) ∧ MUX (v279,v125,v282,v116) ∧
    Dtype (clk,v116,v282) ∧ DtypeT (clk,v126,v285) ∧ NOT (v285,v284) ∧
    AND (v284,v126,v283) ∧ MUX (v283,v124,v286,v115) ∧ Dtype (clk,v115,v286) ∧
    AND (v127,v126,v118) ∧ DtypeT (clk,v119,v290) ∧ NOT (v290,v289) ∧
    AND (v289,v119,v288) ∧ NOT (v288,v117) ∧ CONSTANT 2654435769w v291 ∧
    ADD32 (v20,v291,v287) ∧ Dtype (clk,v24,v114) ∧ Dtype (clk,v23,v113) ∧
    Dtype (clk,v22,v112) ∧ Dtype (clk,v21,v111) ∧ Dtype (clk,v287,v110) ∧
    DtypeT (clk,v118,v294) ∧ NOT (v294,v293) ∧ AND (v293,v118,v292) ∧
    MUX (v292,v116,v295,v96) ∧ Dtype (clk,v96,v295) ∧ DtypeT (clk,v118,v298) ∧
    NOT (v298,v297) ∧ AND (v297,v118,v296) ∧ MUX (v296,v115,v299,v95) ∧
    Dtype (clk,v95,v299) ∧ DtypeT (clk,v117,v302) ∧ NOT (v302,v301) ∧
    AND (v301,v117,v300) ∧ MUX (v300,v114,v303,v94) ∧ Dtype (clk,v94,v303) ∧
    DtypeT (clk,v117,v306) ∧ NOT (v306,v305) ∧ AND (v305,v117,v304) ∧
    MUX (v304,v113,v307,v93) ∧ Dtype (clk,v93,v307) ∧ DtypeT (clk,v117,v310) ∧
    NOT (v310,v309) ∧ AND (v309,v117,v308) ∧ MUX (v308,v112,v311,v92) ∧
    Dtype (clk,v92,v311) ∧ DtypeT (clk,v117,v314) ∧ NOT (v314,v313) ∧
    AND (v313,v117,v312) ∧ MUX (v312,v111,v315,v91) ∧ Dtype (clk,v91,v315) ∧
    DtypeT (clk,v117,v318) ∧ NOT (v318,v317) ∧ AND (v317,v117,v316) ∧
    MUX (v316,v110,v319,v90) ∧ Dtype (clk,v90,v319) ∧ AND (v118,v117,v98) ∧
    DtypeT (clk,v99,v322) ∧ NOT (v322,v321) ∧ AND (v321,v99,v320) ∧
    MUX (v320,v97,v323,v36) ∧ Dtype (clk,v36,v323) ∧ DtypeT (clk,v98,v326) ∧
    NOT (v326,v325) ∧ AND (v325,v98,v324) ∧ MUX (v324,v96,v327,v35) ∧
    Dtype (clk,v35,v327) ∧ DtypeT (clk,v98,v330) ∧ NOT (v330,v329) ∧
    AND (v329,v98,v328) ∧ MUX (v328,v95,v331,v34) ∧ Dtype (clk,v34,v331) ∧
    DtypeT (clk,v98,v334) ∧ NOT (v334,v333) ∧ AND (v333,v98,v332) ∧
    MUX (v332,v94,v335,v33) ∧ Dtype (clk,v33,v335) ∧ DtypeT (clk,v98,v338) ∧
    NOT (v338,v337) ∧ AND (v337,v98,v336) ∧ MUX (v336,v93,v339,v32) ∧
    Dtype (clk,v32,v339) ∧ DtypeT (clk,v98,v342) ∧ NOT (v342,v341) ∧
    AND (v341,v98,v340) ∧ MUX (v340,v92,v343,v31) ∧ Dtype (clk,v31,v343) ∧
    DtypeT (clk,v98,v346) ∧ NOT (v346,v345) ∧ AND (v345,v98,v344) ∧
    MUX (v344,v91,v347,v30) ∧ Dtype (clk,v30,v347) ∧ DtypeT (clk,v98,v350) ∧
    NOT (v350,v349) ∧ AND (v349,v98,v348) ∧ MUX (v348,v90,v351,v29) ∧
    Dtype (clk,v29,v351) ∧ AND (v99,v98,v37) ∧ Dtype (clk,v37,v353) ∧
    AND (v37,v353,v352) ∧ AND (v7,v19,v354) ∧ AND (v354,v352,done))
==>
 DEV TEAEncrypt
   (load at clk,
    ((inp11 <> inp12 <> inp13 <> inp14) <> inp2 <> inp3) at clk,
    done at clk,(out1 <> out2) at clk) : thm
```

35 How are HOL designs downloaded to an FPGA?

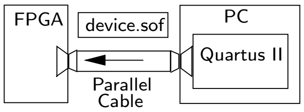
There are four steps to download our circuits to an FPGA.



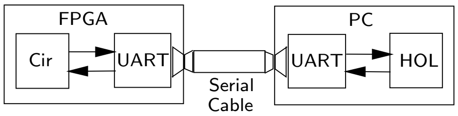
(g) Proof producing synthesis.



(h) Verilog compilation.



(i) Download the object file.



(j) Run the circuit.

Proof Producing Synthesis. The initial step is concerned with the production of the theorem: $\vdash$ InfRise $clk \Rightarrow (Cir \Longrightarrow$ Dev $f)$.

Verilog compilation. A pretty-printer translates the circuit Cir into the Verilog file `Cir.v`. No formal verification is applied to this translation as Verilog has no formal semantics. The primitive components — operators like AND, OR, MUX — are mapped to Verilog modules. The file `UART.v` contains a Verilog implementation of an interface that connects a serial cable to the circuit (this interface has not being formally verified). Both files are sent to Quartus II for compilation.

Download the object file. Quartus II translates the Verilog files into the object file `device.sof`, which is downloaded to the FPGA via the parallel cable.

Run the circuit. HOL is connected to the serial cable by a UART program previously coded in C. The circuit is triggered interactively via an automatically generated function defined in HOL which communicates with the UART program.

All the four steps can be carried out from the HOL system provided that the pretty-printer is able to map every primitive combinational operator to Verilog.

36 What are the plans for the future?

We hope to use the compiler to generate various kinds of cryptographic hardware. We expect more aggressive compaction may be needed to fit bigger examples (e.g. AES) onto the FPGAs we are using.

Eventually, it is hoped to provide wrapper circuitry to enable synthesised HOL functions to be invoked from ARM code as hardware co-processors. The FPGA board we are using (Altera Excalibur) has an ARM processor on it.

37 Is the compiler freely available?

The compiler is distributed with the HOL-4 system (`http://hol.sourceforge.net/`) in the directory `examples/dev`.

38 What related work is there?

There is considerable previous work on using functional programming to specify and design hardware. Examples include μFP (Sheeran [15]), Ruby (Jones & Sheeran [9, 8]), Hydra (O'Donnell [12]), Lava (Bjesse et al [1]), DDD (Johnson & Bose [7]), LAMBDA (Finn et al [5]), Gropius (Blumenröhr [2]), DUAL-EVAL (Brock & Hunt [3]) and SAFL (Mycroft & Sharp [11]).

Our work was initially inspired by SAFL, a hardware compiler for a language based on a first-order subset of ML, though we use a subset of higher order logic rather than a separate special-purpose design specification language. The general way we employ serial/parallel combinators for compositional translation has similarities to compilers for Handel (Page [13]) and SAFL. However, many details differ in our approach: in particular, our compiler is proof-producing.

The paper "Formal Synthesis in Circuit Design – A Classification and Survey" [14] provides an excellent overview. In terms of the classification in that paper, our approach is *formal synthesis* by *transformational derivation* in a *general purpose calculus*.

The way we realise HOL functions by handshaking devices is reminiscent of some self-timed design methods [6, 16], though we produce clocked synchronous circuits.

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 Excalibur FPGA board on which we are running compiled hardware at Cambridge, and they helped us with the Quartus II design software that we are using to drive the board. Ken Larsen used his dynlib library to write an ML version of our original C interface to the serial port (this is used to communicate with the Excalibur board, see 35).

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] Bishop Brock and Warren A. Hunt Jr. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the fm9001 microprocessor. *Formal Methods in System Design*, 11(1):71–104, 1997.

[4] Common Criteria for Information Security Evaluation, 2004. Part 3: Security Assurance Requirements, `http://niap.nist.gov/cc-scheme/cc_docs/cc_v22_part3.pdf`.

[5] 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.

[6] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, 1995.

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

[8] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, North-Holland, 1990.

[9] G. Jones and M. Sheeran. Relations and refinement in circuit design. In C. Morgan, editor, *BCS FACS Workshop on Refinement*. Springer-Verlag, 1991.

[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 synthesis using SAFL and application to processor design. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Livingston, Scotland, September 2001. Springer Verlag. Invited Talk. LNCS Vol. 2144.

[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] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996. `citeseer.ist.psu.edu/page96constructing.html`.

[14] R. Kumar, C. Blumenroehr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 294–299, Palo Alto, CA, USA, 1996. Springer Verlag.

[15] Mary Sheeran. μFP, A Language for VLSI Design. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 104–112. ACM Press, Austin, Texas, 1984.

[16] Kees van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*. Cambridge University Press, New York, NY, USA, 1993.

[17] David Wheeler and Roger Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop*, volume 1008 of *LNCS*, pages 363–366. Springer Verlag, 1999.