SPECIFICATION AND VLSI DESIGN

Graham Birtwistle, Jeff Joyce, Breen Liblong (+), Tom Melham (++), and Rick Schediwy,

Computer Science Department, University of Calgary,
Calgary, Alberta, Canada T2N 1N4.

(+) now at Silicart, Montreal, Canada.
(++) now at Cambridge University, U.K.

ABSTRACT

We describe research into specification-based VLSI design underway at the University of Calgary. Our long term research goals are directed towards building a specification-based design environment (EDICT) to support an iterative, hierarchic design methodology. Our current research has three aspects: the SHIFT high level design capture format (completed); gaining experience in verifying large designs (underway); and building a specification library. In this paper we describe work in progress on two large proofs. The first is for the elimination unit of a local area network device, for which the proof is well underway. The second project concerns the specification driven design of Landin's SECD machine and is just beginning. To set the context for this work on verification, we start by giving partial descriptions of EDICT and SHIFT to show how they use specifications.

INTRODUCTION

In EDICT [2] we picture a design under construction as a tree, every node of which has a specified functional behaviour. If we compose together the behaviours of a node's immediate offspring, we can check at once to see whether or not their composition agrees with their parent's specification. If so we have a correct refinement of the design; if not we know at the earliest possible stage that we are wrong. By deductive argument, we can show the correctness or otherwise of a complete design (within the limits of the proof model - we work at the register transfer level and do not include electrical effects). Specifications are expected to play an increasingly important role in VLSI design, especially for military, industrial control and medical applications where the guarantee of design soundness is crucial. They also assist the process of splitting designs amongst teams in such a way that the function of the whole chip need not be revealed in its entirety to any sub-contracted party.

The output from EDICT is in a high level VLSI design exchange format called SHIFT [15, 16] which captures and retains the structure of the design as well as its low level detail. The SHIFT description of a design follows the behavioural hierachy exactly. A SHIFT description consists of several *views* (e.g. behaviour, constraints, electrical properties, geometry, structure, ...). These views have been chosen to enable SHIFT to interface easily to current (and future) lower level tools such as SPICE, MOSSIM, timing verifiers and plotters. These special purpose tools can be used to fill out those views in a SHIFT definition that are

not derivable from the behaviour. For example, the electrical-properties view can be filled out via SPICE for leaf cells, and higher level simulators can be used to check timing constraints (e.g. port to port signal timings). Other standard packages will provide plots and foundry tapes etc. SHIFT supplies composition tools for all views so that large designs can correctly constructed automatically from validated library elements. SHIFT is also used as the design data base language for validated designs. Like EDIF [3, 4] to which it bears a very close resemblance, SHIFT may also be used to port designs between workstations, to different fabrication lines, and to put designs in the public domain.

The latter sections of this paper outline progress on two substantial register transfer level proofs of correctness. The first proof will demonstrate the correctness of a design for the elimination unit of a local area network box. The second is concerned with an SECD machine [9, 13]. Both proofs are being effected mechanically using proof assistants. We worked with LCF_LSM [5] for the elimination unit proof which was started in 1983, but swapped to HOL [6] for the SECD specification work.

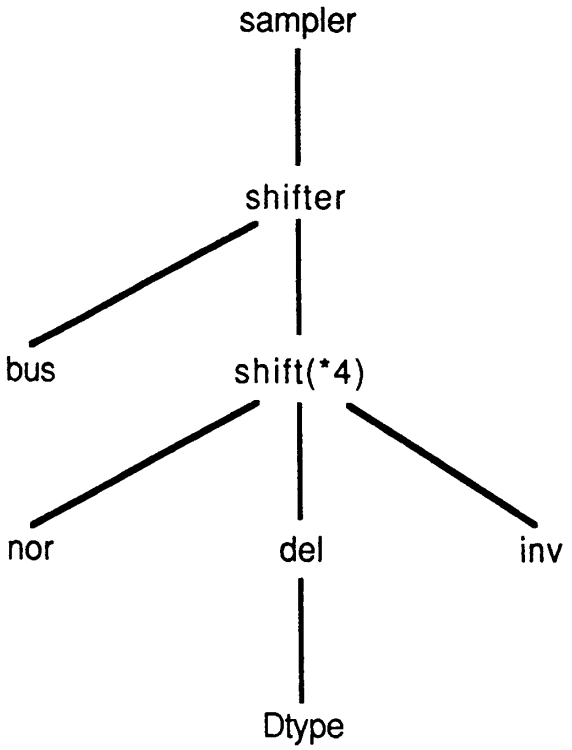

Figure 1 Sampler hierarchy

OVERVIEW OF EDICT

The key elements in EDICT [2] are support for a design methodology, the use of proof techniques to ensure that designs meet their specification, and SHIFT.

One method of controlling the complexity of VLSI designs is to reduce the design space by imposing a design methodology. We have chosen to work with hierarchies where a design is constructed as a sequence of levels each lower one a proven consistent refinement of the one above. Because today's design becomes tomorrow's building block, we also arrange to collect validated designs in a library from whence they can be offered, perhaps semi-automatically, to future designers. In this way, we are accomodating and expanding the standard cell approach. Within the EDICT system we create large cells automatically by composition, and will be looking into the possibilities for system-polished designs (as with TOPOLOGIZER [12]).

Designs in progress are represented in EDICT by a tree of nodes. A simple example presented in [8] considers the implementation of a sampler which inputs a serial bit stream and outputs a 4-bit word. The output value at time t is #0000 if the sample line is low; if the sample line is high, it is the concatenation of the input bits at times t-3 through t. We can decompose the sampler first into a bus and a shifter, then the shifter into four slices, each slice into a delay an inverter and a nor gate, and finally implement the delay unit by a D-type flip-flop. This design decomposition is shown in figure 1.

If we design from the top down, the single node at level 1 represents the complete design, its offspring at level 2 the floor plan elements, and so on. Initially, we will have the specification of the design's behaviour (which expresses output signals in terms of current and previous values on its input lines), together with some performance expectations and area limitations expressed as constraints. Our first task is to partition the design into a set of components whose behaviour when connected together conforms to the specification of the design at level 1, and will satisfy its constraints. We specify the behaviour of each floor plan element in turn and write down its local constraints, if any. Then we compose the behaviours of the floor plan elements together and deduce the behaviour of this first design refinement. If the aggregate behaviour can be shown to be equivalent to that of the design at level 1, then we have a valid implementation (subject to the constraints being satisfied) and can recursively apply the same technique to each floor plan element in turn. If the specifications do not match, then we know at the earliest possible moment that we are on the wrong track.

As a design develops, the nodes become simpler and simpler until they can be implemented at once by a leaf cell editor or already exist as library elements. End nodes in the design tree are called *leaf cells* (e.g. bus, not, nor, D-type above) and intermediate nodes are called *composition cells* (e.g. shifter, slice, del above). Leaf cells are fully developed and, besides their inherited behaviours and ports, will have synthesized attributes such as structure, layout, schematics, and known power consumption, delay, and area.

New leaf cells will be hooked up to SPICE, MOSSIM, timing verifiers, design rule checkers and layout tools etc for complete validation and to ensure that they satisfy their constraints. When templates for all classes of leaf cell instances are known in full, the dialog

with the user is suspended, and an automatic design composer takes over. The composer has built in composition rules for all template attributes. It composes the structure and layout of offspring and fills in the attributes of their parent. It will do the same for power requirements, delay, and area. We now work our way back up the design tree applying composition rules to fill out the composition cell templates and to check that they satisfy their constraints. A cell whether it is a leaf cell or a composition cell is not complete until all its inherited and synthesized attributes have been filled out and its local and imposed constraints satisfied. Once a template is complete and verified, it is included in the SHIFT library and will be available to future designers.

The automatic composer back end effects the compositions remaining (other than the behaviours). If a design fails to meet its constraints, then EDICT can help at the next iteration. Templates from the previous iteration should contain better estimates for critical values than were available on the previous unsatisfactory design attempt. These can be taken into account and may lead to global optimisations, and floor plan alternatives or architectural choices being investigated early on in the next iteration.

Thus the overall structure of EDICT is an interactive design capture front end and and automatic composer. The front end constructs the design tree by requesting specifications from the designer, composes specifications and assists in the arduous task of checking whether or not two specifications are the same. The front end fills in only the specification portion of the design template. The composer extracts completed templates from the library and composes them to complete the design.

THE SHIFT DESIGN CAPTURE LANGUAGE

SHIFT is described in [15, 16]. A prototype for flexible NMOS and CMOS cells was implemented in 1984 and has been enhanced since. SHIFT will be used in EDICT to capture the various descriptions of a circuit in a consistent manner, and support the development of the design through incremental refinement. SHIFT retains the structure and hierarchy put into a design, supports composition, and serves as a library format.

SHIFT bears a strong resemblance to EDIF [3, 4] (Electronic Design Interface Format) which is being developed as a language for design exchange between designers and as a foundry interface language, but not as a design data base language. SHIFT and EDIF were developed quite independently.

SHIFT is embedded in LISP and the syntax retains the same flavour. A design is specified by leaf cells and composition cells. All cells are rectangular with a boundary on which ports are placed. Cells are composed by abutment in horizontal and vertical directions. The composed cells are stretched to connect adjacent ports.

A leaf cell is specified by a name, a list of ports, geometry, structure, electrical properties, a minimum bounding box, behaviour, and a set of constraints.

(**defcell** *cell_name*
(ports.....)
(geometry.....)
(structure.....)
(electrical properties....)
(mbb.....)
(behaviour.....)
(constraints.....)
)

Ports are specified along the north, south, east and west walls of the cell, and it is only through abutment of these external ports that instances of cells are connected in any of the views. This view is derived from the behaviour and constraints fields, and is used in floorplanning, placement, and routing stages.

The *geometrical* view is specified as a series of geometric patterns on distinct layers corresponding to the fabrication masks. A range of operators is provided for specifying and transforming polygons, boxes, wires, and contacts. Leaf cells are made stretchable by specifying constraints between the ports. Any geometric coordinates specified relative to the ports are determined when the cell is instantiated. The actual connection of layers through the abutting ports of the components of a composition cell are checked separately by a geometric design rule checker.

The *structural* view is specified as a collection of components and a netlist. Components are named instances of transistors, resistors and capacitors. Connections are then specified between the gate, source and drains of the transistors, the terminals of the resistors and capacitors, and the ports. We will be investigating how to derive this view from the behavioural view automatically.

The *electrical properties* field supplies additional information about the components, such as relative strengths of the transistors, relative sizes of the capacitances specified as functions dependent on the port values in order to facilitate performance simulation, pin to pin timings and power estimates. SPICE is used for leaf cells. We will be using algorithms similar to those of Lin [17] and Trimberger [25] respectively to compose electrical properties and optimise delays and power. We are also incorporating restoring logic checks following Mead and Rem's algorithms [24].

The minimum bounding box, *mbb*, is derived from the geometry view and is used in floorplanning, placement and routing.

Constraints may be specified in the design of a composition cell for any view. For example, the physical dimensions of the resulting identity cell could be specified through constraints on the composition cell's ports that must be met in the instantiation of the component cells. Similarly we could specify that the list of connections from the composition of the three component cells must match some structural specification of the count cell. Finally a behavioural constraint could be that some specified behaviour of the composition cell must match the behaviour derived from the the component cells.

Composition cell definitions require two fields - the composition itself plus any additional constraints that may be made. For example, a 4 by 4 array of SHIFT register cells can be constructed from a single SHIFT cell by

```
(defcomp SHIFT4ROW (> (rep 4 SHIFT)) nil)
(defcomp SHIFT4by4 (^ (rep 4 SHIFT4ROW)) nil)
```

where > and ^ are the *abut east* and *abut north* operators and 4 the repetition factor. No extra constraints are specified. If empty, other views are filled out automatically by composition If not empty they may be considered as specifications to be met. The composition rule specifies the patterns for the masks from the patterns of its constituents. It also specifies the structure as two sub-graphs being joined together at adjacent ports. The behavioural description is given by the composition of the behaviours of the components, where abutting ports are identified. And so on. Nothing extra in the specification in any of the views is introduced by the composition rule that is not derivable from the component specifications.

The *behavioural* view will be specified in HOL [6]. One current research topic is to find out just how much can be deduced from the specification alone. For example, it is quite straightforward to generate hierarchical simulators automatically from typed specifications (these will be used to give confidence in the correctness of specifications as design work progresses) and it is also straightforward to produce first cuts at floorplans automatically from the wiring (just using the typed ports) in the manner of Nixon [22].

TOWARDS THE VERIFICATION OF THE ELIMINATION UNIT

The flooding sink is a local area network proposed, designed, and analysed at the University of Calgary by a team led by Brian Unger [23]. The design permits specially tailored nodes to be linked together in an arbitrary fashion and is suitable for use as a real time local area network requiring a very high bandwidth.

Attached to each node is a host through which users enter and receive messages. Each node contains an arbitrary number of connections to other nodes numbered [0..n]. A connection between nodes consists of a full duplex link. One link (which we number 0 for textual convenience) is reserved for communication with the host.

Messages circulating in the network contain a source address, a serial number, and a destination address each of length 8 bits. The 16 bit field 'source address + serial number' is taken as the message identifier. When a host sends a message, a copy is routed into its node and from there broadcast along each of the output lines [1..n], but not back to the host. Messages received are buffered separately for each input port. In a broadcast system, the same message may reach a particular node several times via different routes. Incoming messages are thus first checked to see if they have been received before. If so, they are discarded. If not, they are added to the list of messages received and will be rejected if received again later. The analysis given in [23] shows that overall system performance depends heavily the ability to accept new messages and reject repeats quickly. The *elimination unit* is a hardware reponse to that problem and it is with its implementation that we are concerned.

The elimination unit polls each input buffer in turn, looking for a request from the buffer to check an incoming 16-bit message ID. When a request is found, the elimination unit checks the message ID against a list of the last 256 identifiers to reach this node by using the message ID as an index into a 64K by one-bit memory, *lookup*. This memory returns true if the identifier is one of the last 256 received or false if it is not. If true is returned, the message is not to be broadcast via the output ports and the eliminator responds to the request with keep=false. If false is returned by the memory, the eliminator responds keep=true so that the message is kept for broadcasting to the other network nodes. In this case, the eliminator updates *lookup* by setting the bit indexed by the incoming ID to true. In order to ensure that *lookup* contains only 256 bits which are set, the elimination unit has a 256 long FIFO in which the last 256 ID's to arrive are stored in order. When *lookup* is updated, the bit indexed by the oldest ID in the FIFO is set to false, so that *lookup* will contain only 256 set bits. The FIFO is implemented using a 256 by 16 bit memory, *lastids*, and a pointer, *oldest*, which is used to point to the oldest ID in the queue. By using two memories as outlined above, the elimination unit does not have to sequentially search a list of previously seen IDs and therefore is able to respond very quickly to look up requests. Pseudo-code for the elimination algorithm is given below:

ELIMINATION UNIT:

```
for k := 0 to n do
  if buffer[k] requests an ID look up then
    if lookup[new_ID] then signal keep=false else
      {! set look up memory right;
       lookup[new_ID] := true;
       lookup[lastids[oldest]] := false;
       ! add new ID to FIFO;
       lastids[oldest] := new_id;
       oldest := oldest+1 mod 256;
       keep := true;
      }
    wait until buffer[k] drops request;
repeat;
```

A prototype of the elimination unit was designed and breadboarded by Rick Schediwy and the proof undertaken by Tom Melham at the University of Calgary. Implemented in bipolar, the prototype consists of 37 SSI, MSI and LSI chips ranging in complexity from 4-input NAND gates to AM2910's. The proof has proceeded from the bottom-up. At the time of writing, full proofs have been completed for all the floor plan elements and what remains is the matching of their composition to that of the high level specification. Details of the proof of this device are beyond the scope of this paper. Even in its present incomplete state it covers 9000 lines of LCF-LSM. Instead of giving details of the proof at all levels, we will follow down one branch of the full proof tree from the top level to the bottom.

At the top level we view the elimination unit as a device with three ports and six states (see figure 2).

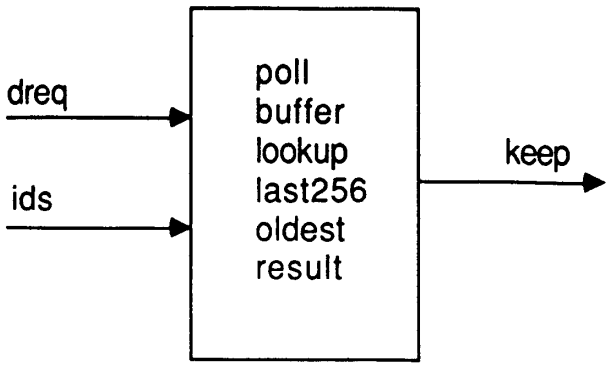

Figure 2 Top level view of the elimination unit

The ports are named *dreq*, *ids*, and *keep*. Requests to look up identifiers are received on the 8-bit bus, *dreq*, which is polled by the elimination unit. The input port, *ids*, is a list of 8 16-bit busses from which the incoming message IDs are read. The output line, *keep*, is a one-bit flag which indicates that an identifier has been seen previously (*keep*=false) or has not (*keep*=true).

The elimination unit has six state variables: *poll*, *buffer*, *lookup*, *lastids*, *oldest*, and *result*. The variables, *lookup* and *lastids*, are the 64K by one-bit look up memory and the 256 by 16-bit FIFO, respectively and the state variable, *oldest*, is the 8-bit FIFO pointer. The three-bit register *buffer* stores the number of the input buffer currently being polled. The result of the ID look up is stored in the one-bit state, *result*. The flag, *poll*, indicates which of two states that the device is in. When *poll*=true, the elimination unit is at the beginning of a polling cycle, looking for a request from the input buffer whose number is given by *buffer*. In this state, the keep output has the value true. When *poll*=false, the result of the ID look up is on the output and the elimination unit is waiting for the requesting input buffer to drop its look up request.

The specification of the elimination unit in LCF_LSM is:

```
ELIM(poll, buffer, lookup, last256, oldest, result)
  ==
dev {dreq, ids, keep}.
{keep = (request and not poll -> not result | T)};
ELIM( (request -> F | T),
      (request -> buffer | INC3 buffer)
      (request and poll and (not res) -> STORE16 id (#1) (STORE16 oldid (#0) lookup) | lookup),
      (request and poll and (not res) -> STORE8 oldest id last256 | last256),
      (request and poll and (not res) -> INC8 oldest | oldest)),
      (request ->(poll -> res | result) | F))
    )
```

where *request* is the input request from the current input buffer (formally an abbreviation for *REQUEST buffer dreq* which returns the value of *dreq[buffer]*, the 8-bit request bus indexed by the current input buffer), *ids* returns the identifier belonging to the input buffer currently being polled from the list of input identifiers (formally *EL (VAL3 buffer) ids*), and *res* returns the lookup memory indexed by the incoming identifier (formally this value is given by *BOOL1(FETCH16 lookup id)*.

This specification states that the device ELIM, with the six state variables mentioned above, has the behaviour given by the 'dev' expression on the right hand side of the equation. The first part of the dev expression is just a list of the device's external ports: *dreq*, *ids* and *keep*. The second part of the dev expression gives an equation for the output line, *keep*. The output equation states that if the device is not in the polling state (i.e. **not** *poll*) and the incoming request for ID look up is still active then the output will be given by the negation of the stored result of the look up, *result*. Otherwise, the *keep* output will have the value true.

The third part of the dev expression gives the 'next state' expression for each of the state variables. The first of these gives the next state for *poll*. If there is a look up request then the next state of *poll* will be false, otherwise *poll* will become true. Thus, whenever there is no incoming look up request, the elimination unit goes into polling mode and, whenever the incoming request is active, the elimination unit goes into output mode. The next state expression for *buffer* specifies that, when there is no request, *buffer* is incremented MOD 8 (by INC3) so that the elimination unit will go on to poll the next input buffer. Otherwise, *buffer* remains the same. The next states of *lookup*, *ids* and *oldest* all depend on the expression *request* **and** *poll* **and** (**not** *res*). When this expression is true, the device is in polling mode, there is an identifier look up request and the incoming identifier has not been seen before. In this case, the three states *lookup*, *ids* and *oldest* are all updated to reflect the fact that the new identifier has now been seen. When there is no request, the next state of *result* is the value false. Otherwise, the next state of *result* depends on the value of *poll*. When *poll* is true, the next state of *result* is the result of the identifier look up, *res*. When the elimination unit is not in polling mode (*poll*=false) the value stored in *result* stays constant.

The elimination unit is implemented using eight components START, IDBUS, RAMA, RAMB, COUNTERS, RESULT, DRIVERS and UNCNTL wired together as shown in figure 3. The LCF_LSM specification of the implementation is:

```
ELIM_IMP(start,device,lookup,last256,count,lastinc,buffer,result,uinstr,reg,adder,stack,sp,ucode)
    ==
  [| START start
   | IDBUS rn[en=det]
   | RAMA lookup ]
   | RAMB last256
   | COUNTER(count, lastinc)
   | RESULT(buffer, result)
   | DRIVERS
   | UCNTL(ucode, uinstr, adder, sp, stack, device)
  |]
  hide {adata,addr,ainit,awe,bsel,bwe,clear,drivers,init,nffl,nffh,rado,ramb,reply,req,run}
```

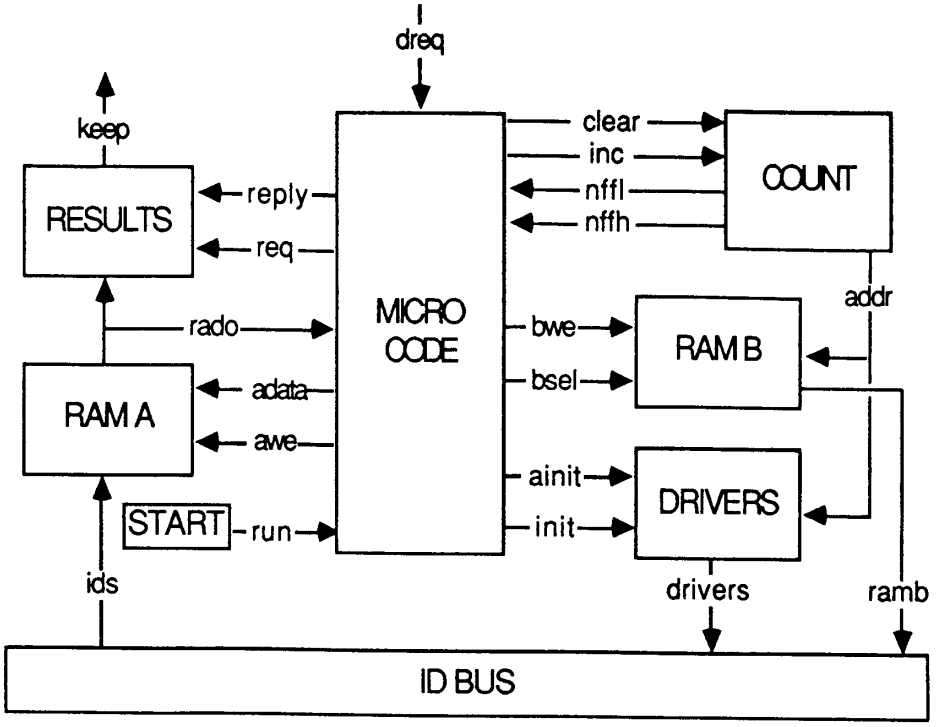

Figure 3 Elimination unit floor plan

which lists the composing elements, how they are wired together including any wire renamings, and also lists the internal (hidden) wires.

START is used for proof purposes to simulate initialization from power-up. When its state goes from false to true, the initialization sequence will begin. When its state remains constant at true, the device is in regular operating mode. At this time, we have not worked on the correctness of the initialization cycle. IDBUS is the 16-bit bus for incoming identifiers which serves two purposes. It is used 1) during the initialization cycle to address RAMA from the counters, and 2) to address RAMA from the contents of RAMB when clearing the oldest ID bit in lookup. RAMA is the 64K by one-bit look up memory (a 64K by one-bit RAM, with state *lookup*. RAMB is the 256-item FIFO memory, with state *last256*. COUNTER is a 16 bit counter which serves two function. During the initialization sequence, it generates addresses for clearing RAMA and RAMB. Otherwise, the LOW 8 bits are used as the pointer into RAMB, (the FIFO pointer *oldest*). COUNTER has two outputs, nffl and nffh which indicate when the low and high bytes (respectively) have the value #11111111. This data is used by the controller during initialization. RESULT stores the look up result and generates the *keep* output. One of its states, *result*, is the result state of the specification. DRIVERS is used to control what is put onto the IDBUS from internal sources, and is discussed further

below. UCNTL is the controller for the whole device; it uses microcode and an AM2910 microcontroller to generate control signals.

The proof is quite large (several thousand lines) and even its specification is too large to give in this paper. The specifications and proofs of all these top level components is now complete. What remains is the final step of showing that they compose to the original specification.

In what follows we trace the implementation of DRIVERS. DRIVERS controls what is put onto the id bus from the counters. The line *drivers* goes to the bus, the line *addr* comes from COUNTER. The specification of DRIVERS is:

```
DRIVERS
  ==
dev {ainit, init, addr, drivers}.
{drivers = (init -> FLOAT16 | (ainit -> MK_TRI16 addr
                | MK_TRI16(MERGE_BYTES(#00000000, HIGH_BYTE(addr))))};
DRIVERS
```

DRIVERS has three inputs ainit, init, addr and one output line drivers. When init is true, DRIVERS is disabled and the value put onto the bus is a 16-bit floating value *FLOAT16*. When init is false, DRIVERS is enabled and is being used to put addresses onto the bus in order to address RAMA and RAMB for initialization. When ainit is true, we are addressing RAMA and the address *addr* is put directly onto the bus. MK_TRI16 is an explicit type conversion which changes the value of addr from a 16-bit type which only includes 16-bit integers to a type which also has the floating value. When ainit is false, we are addressing RAMB using the high byte of the counter address, addr. The low byte output consists of a padding of 0's.

DRIVERS was implemented as the composition of seven primitive units.

```
DRIVERS_IMP
  ==
[| ZERO4          rn[out=zero4] |
 | SPLIT_BYTE     rn[low_nibble=low4; high_nibble=high4; bytein=low]
 | SPLIT_WORD     rn[low_byte=low; high_byte=high; wordin=addr]
 | TRI_MUX        rn[ina=zero4; inb=low4; select=ainit; enable=init; outtri=nibble1]
 | TRI_MUX        rn[ina=zero4; inb=high4; select=ainit; enable=init; outtri=nibble2]
 | DRIVER8        rn[input=high; output=byte; enable=init]
 | MERGE_TRI      rn[word=drivers]
|]
hide {zero4: word4, low4: word4, high4: word4, nibble1: tri_word4,
      nibble2: tri_word4, byte: tri_word8, low: word8, high: word8}
```

ZERO4, SPLIT_BYTE, SPLIT_WORD, TRI_MUX, DRIVER8, and MERGE_TRI are given basic building blocks, and have to be defined as axioms in LCF_LSM. The implementation is depicted in figure 4.

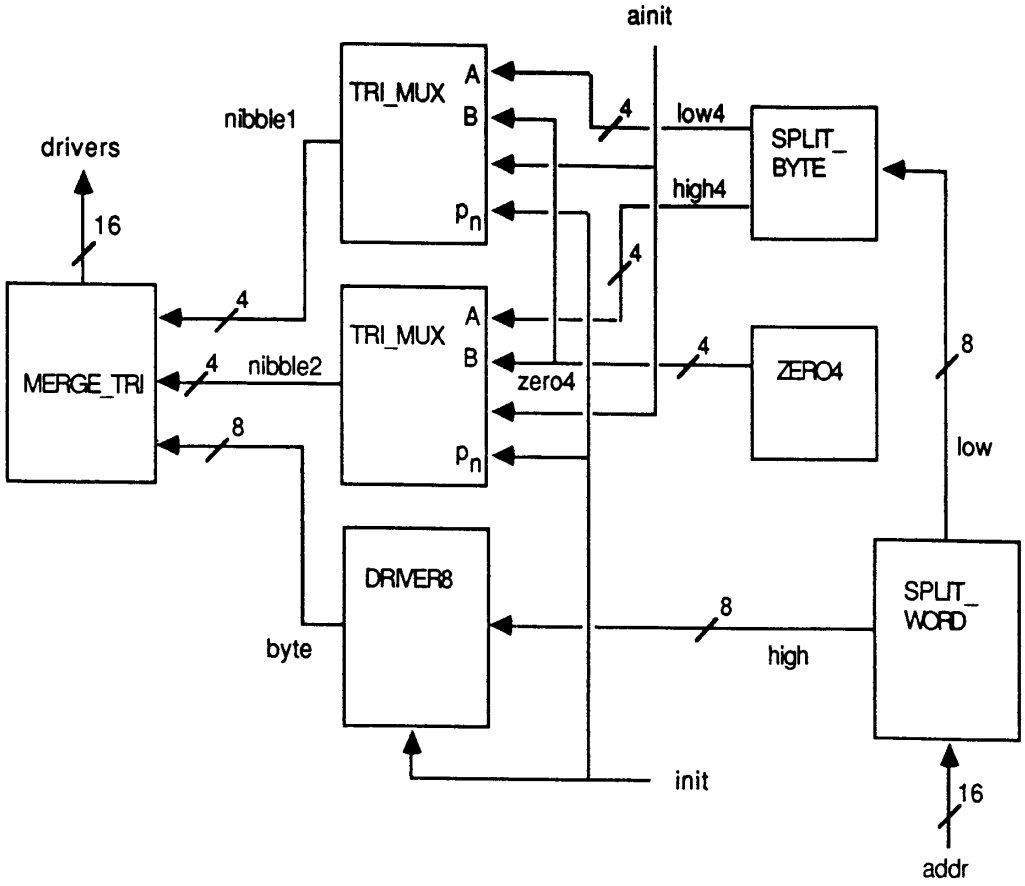

Figure 4 DRIVERS implementation

ZERO4 always outputs the value #0000. Its specification is:

ZERO4 == dev {out}. {out = #0000}; ZERO4

This subterfuge represents a real device which is actually wired to ground. SPLIT_BYTE splits an 8-bit byte into 2 4-bit nibbles. This is achieved in the real device by wiring, but we need to be careful because we have to do the type-conversion. Its specification is:

```
SPLIT_BYTE
  ==
dev {bytein, high_nibble, low_nibble}.
{high_nibble = HIGH_NIBBLE(bytein), low_nibble  = LOW_NIBBLE(bytein)};
SPLIT_BYTE
```

SPLIT_WORD splits a word into low and high bytes. Again we need explicit type conversion. It is implemented as wiring in the real device.

```
SPLIT_WORD
  ==
dev {wordin, high_byte, low_byte}.
{high_byte = HIGH_BYTE(wordin), low_byte  = LOW_BYTE(wordin)};
SPLIT_WORD
```

MERGE_TRI merges two 4-bit tri-state words and one tri-state byte into a 16-bit tri-state word. It is only wiring in the real device.

```
MERGE_TRI
  ==
dev {nibble1, nibble2, byte, word}.
{word = (nibble1=FLOAT4 and nibble2=FLOAT4 and byte=FLOAT8) -> FLOAT16
              | MERGE4_4_8(nibble1, nibble2, byte)};
MERGE_TRI
```

TRI_MUX outputs 4-bit floating when disabled, otherwise selects one of two 4-bit tri-state inputs. It is implemented as a 74257 in the real device.

```
TRI_MUX
  ==
dev {ina, inb, select, enable, outtri}.
{outtri = (enable -> FLOAT4 | (select -> MK_TRI4 inb | MK_TRI4 ina))};
TRI_MUX
```

DRIVER8 is an 8-bit tri-state driver. When enabled, it outputs its input. Otherwise it outputs FLOAT8. It is a 74244 driver in the real device.

```
DRIVER8
  ==
dev {enable, input, output}.
{output = (enable -> FLOAT8 | MK_TRI8 input)};
DRIVER8
```

The DRIVERS statement of correctness is straightforward, but tedious because of type conversion boxes. Because the circuit is purely combinational, it is easy to show that *DRIVERS == DRIVERS_IMP*. (DRIVERS has been since proven correct automatically using a version of BARROW's VERIFY system [1].)

We stop here, having reached the level of TTL chips actually used in the design, a level that corresponds in EDICT to making use of validated library elements. We have verified all the floor plan elements in similar fashion. Working with a bipolar implementation of the design gave rise to a certain lack of elegance in the lowest level specifications, because the 8-bit functions we wanted were not available to us and we had to work with 4-bit chips. In

VLSI implementations, cells at this (and higher) levels can be parameterised [14] and the resulting specifications are correspondingly clearer.

THE SECD MACHINE

The SECD machine is an architecture for executing programs written in a purely functional dialect of Lisp. The SECD machine was originally described by Landin in 1963 [13]. A version of this machine, described by Henderson [9], is the basis of our study [10].

We have designed a register-transfer level implementation of the abstract machine described by Henderson. Our design includes an register-transfer level implementation of garbage collection. We have also produced a formal specification of this implementation in the LCF_LSM hardware specification language.

Our design work began with a hardware interpreter written in a high-level programming language. This simulation software was used to investigate a possible SECD architecture down to the micro-code level. The simulation gave us insight into Landin's original SECD proposal and a complete and detailed understanding of its intended mode of operation. The simulation model of the architecture was refined precisely along the lines of the envisaged hardware decomposition. Detailed timings and critical path highlightings which emerged from simulation runs, were fed back into the next design iteration cycle. Most importantly, the simulation gave us a much clearer understanding of what the top level and floor plan specifications should contain and what they should look like.

This first-approximation freely employed high-level programming language constructs such as complex data structures and recursion. For instance, a recursive tree traversal was used in the implementation of garbage collection. The next step was to refine this high-level simulation into a register-transfer level simulation of the SECD machine. This involved eliminating the rather liberal use of high-level programming constructs and using very simple control constructs in their place. For instance, the recursive tree traversal became a non-recursive traversal. Similarly, complex data structures were replaced with a hardware oriented representation of data. In the process of refining the high-level simulation into a register-transfer level simulation, the actual architecture of the implementation gradually evolved. This was really a matter of realizing, for example, when an additional register was required in the course of translating some high-level code in the simulation into register-transfer instructions.

The effort to produce a formal specification of the implementation in parallel with development of the register-transfer level simulation was extremely worthwhile. The formal specification was used to capture the current state of the register-transfer level design. Furthermore, the formal specification was responsible for consistency in the design. For example, the formal specification enforced assumptions that the designer had 'in the back of his head' such as whether a particular signal was a 14-bit or a 32-bit value. Formal specification also encourage the designer to observe functionality in the design, for example, avoiding using a single register for two unrelated functions.

The last step was to translate the register-transfer level simulation into microcode

which was then assembled into a binary image. This binary image was then used to control the lowest level simulation. This lowest level simulation actually read a microinstruction out of a (simulated) ROM, decoded it, and interpreted the control signals. The purpose of this simulation was simply to verify that the microcode has been accurately derived from the register-transfer level simulation. The simulation also provided exact (simulated) execution times.

We are currently working on a target-level specification of the SECD machine. The target-level specification of the SECD machine will closely resemble the formal description given by Henderson. Henderson describes the SECD machine in terms of a state-to-state transition for each one of the twenty-one SECD machine instructions where the state of the machine is given by the contents of the four principle registers: s, e, c and d. An sample of one of these transitions is shown below.

$$(a\ b.s)\ e\ (ADD.c)\ d \rightarrow (b+a.s)\ e\ c\ d$$

This target-level specification will be used to formally state that the implementation correctly implements the SECD machine. Proving this statement of correctness will formally verify the implementation.

The specification and verification of the SECD will involve several different types of abstractions including structural, temporal and data abstractions. We have had experience with both structural and temporal abstraction in the specification and verification of a general-purpose 16-bit microcoded computer [7, 11]. However, we expect that data abstraction will be a difficult task requiring the use of fixed-points. The required data abstraction will relate (potentially infinite) S-expressions in the target-level specification to linked lists implemented in finite RAM. The final result will be be a substantial example of hardware specification and verification.

REMARKS

Our long term aim is to build a specification based VLSI CAD system. In the short term, work on verification at Calgary has concentrated on gaining experience with large proofs. For this work, the register transfer level is the most appropriate abstraction level. More detailed mathematical models can be found in [18, 20, 21], and [19] describes initial work on an abstraction function relating timing level signals and register transfer level signals.

Implementing the elimination unit in off-the-shelf bipolar was awkward at the lowest level where we had to work with the building blocks available. These were not always what we wanted, e.g. we had to work in 4-bit nibbles when we really wanted 8-bit components. The bottom level specifications were correspondingly cluttered. We feel that this problem is due to the implementation medium, not the verification technique, for parameterised components are easy to handle in VLSI. The elimination unit proof has also taught us to be wary of attempting bottom up proofs - a major headache has been (and still is) devising a top level specification to suit the floorplan. Finally, attempting the verification of a design after its implementation was instructive. Our feelings are that implementation ad hocery begets untidy

proofs and that neat specifications beget clean hardware implementations. This confirms our prejudice towards specification driven design.

ACKNOWLEDGEMENTS

The authors would like to thank Geoff Wyvill who painted the figures for us, JADE colleagues at Calgary for their help, encouragement, and general software support, and the other members of the VLSI team at Calgary for providing an stimulating milieu. We would also like to thank John Gray and Mike Gordon who have been particularly helpful not only in getting us going in verification but in sustaining us thereafter. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] H.Barrow. *Proving the correctness of digital hardware designs.* VLSI Design, July 1984.

[2] G.Birtwistle *et al EDICT an environment for the design of integrated circuits.* University of Calgary Research Report 84/155/13, 1984.

[3] J.D.Crawford. *EDIF: a mechanism for the exchange of design information.* CICC, 1984, 446-449.

[4] Electronic Design Interchange Format Specification. Version 1.0. 1985.

[5] M.Gordon. *LCF_LSM.* Technical Report 41, Computer Laboratory, University of Cambridge, 1983.

[6] M.Gordon. *HOL a machine oriented version of higher order logic.* Draft Report, Computing Laboratory, University of Cambridge, May 1985.

[7] M.Gordon. *Proving a computer correct.* Technical Report 42, Computer Laboratory, University of Cambridge, 1983.

[8] M.Gordon. *Formal methods for hardware verification.* University of Cambridge Draft Report, 1984.

[9] P.Henderson *Functional programming.* Prentice Hall 1985, pages 100-150.

[10] J.Joyce. *The SECD machine.* University of Calgary Research Report, 1985.

[11] J.Joyce, G.Birtwistle, and M.Gordon. *Proving a Computer Correct in Higher Order Logic.* University of Calgary Research Report 85/208/21.

[12] P.W.Kollaritsch and N.H.E.Weste. *TOPOLOGIZER: an expert system translator of transistor connectivity to symbolic cell layout.* IEEE Journal of Solid State Circuits, SC-20(3), 799-804, 1985.

[13] P.Landin. *On the mechanical evaluation of expressions.* Computer Journal 1963.

[14] Lattice Logic. *Designing with gate arrays.* Edinburgh. 1981.

[15] B.Liblong. *SHIFT a Structured Hierarchic Intermediate Form for VLSI Design.* Masters Thesis, University of Calgary, 1984.

[16] B.Liblong and G.Birtwistle. *A VLSI Design System Based Upon a High Level Interrmediate Form.* Canadian Conference on VLSI, Waterloo, 1983, 150-153.

[17] T-M.Lin. *A hierarchical timing simulation model for digital integrated circuits and systems.* PhD Thesis, Caltech 1984. Available as 5133:TR:84.

[18] G.Milne. *CIRCAL and the representation of communication, concurrency, and time.* Toplas, 7(2), April 1985, 270-298.

[19] T.Melham. *A signal abstraction function.* Draft memo, Computing Laboratory, Cambridge University. 1985.

[20] B.Moszkowski. *A temporal logic for multi-level reasoning about digital hardware.* IEEE Computer, February 1985, 10-19.

[21] B.Moszkowski. *Executing temporal logic programs.* Technical Report 55, Computing Laboratory, Cambridge University. 1984.

[22] I.M.Nixon *IF an idiomatic floorplanner.* University of Edinburgh Research Report CSR-170-84, 1984.

[23] T.Patten, N.Hutchinson, and B.Unger. *The flooding sink: a new approach to local area networking.* University of Calgary Research Report no. 83/124/13. May 1983.

[24] M.Rem, and C.Mead. *A notation for designing restoring logic circuitry.* Proc Second Caltech Conference on VLSI. Pasadena 1981, 399-412.

[25] S.Trimberger. *Automated performance optimisation for custom integrated circuits.* PhD Thesis. Caltech 1981.