# Verifying SECD in HOL

G. Birtwistle and B. Graham[1]

## Abstract

This paper describes some of the work done at Calgary on the design of an SECD chip and its verification using the Cambridge HOL proof assistant. The chip is a physical realization of Henderson's variant of Landin's abstract architecture to execute the lambda calculus. The machine uses closures and includes explicit machine instructions to assist recursion. The complete proof, which goes from an abstract specification down to the transistor level, is far too involved to be covered in a single paper. In this paper, we discuss the SECD architecture and design and trace through a portion of the proof of correctness of one sequence of the microcode.

[1]Addr.: Department of Computer Science, University of Calgary, 2500 University Drive, Calgary, Alberta, Canada T2N 1N4



## Contents

# 1 Introduction

Work on SECD started in 1985. The aim of the project was to investigate the specification driven design methodology (design for verifiability) by designing a reasonably substantial chip and verifying it with the HOL proof assistant [10]. We chose SECD because it was well known and well understood and there was much work to build upon — see for example, [16] and [22]. We chose HOL because it was there and had the best track record in hardware verification. Besides the proof, we have also: (i) designed, fabricated and are testing a version the chip, (ii) constructed a rig and associated software (including a compiler) so that we can compile Lispkit[2] programs and download them to run on SECD, (iii) completed a (hand) proof that the abstract SECD machine executes Lispkit programs correctly.

The SECD chip arose within an ongoing research effort by the VLSI and verification group at the University of Calgary. The chip was used as a vehicle to explore the use of specification to drive design synthesis. The methodology entails elaborating a design hierarchically as a tree of nodes and formally specifying the behaviour at each node. Verifying that the composition of behaviours of a node's children agrees with the node's specification assures a correct design. By deductive argument, we can show the correctness or otherwise of a complete design. Thus the design and verification process is the object of study, and the chip is only a byproduct providing the team with hands-on experience with a nontrivial design. This intention gave rise to design criteria (make it simple, testable, useful, but above all correct) that affected decisions taken throughout the design.

The perceived need for higher levels of reliability of computer and control systems has driven the exploration of formal verification methods. Software and hardware verification efforts have generally been undertaken independently. A common notation for all levels of hardware design as well as the software running on the systems is advantageous when attempting to verify a complete system. The HOL notation used in our work is sufficiently expressive to achieve this objective.

The difficulty of verifying program correctness for imperative languages has motivated the choice of functional programming languages for our subject system. The strong mathematical basis of functional languages makes them more amenable to formal correctness study than imperative languages. Thus the choice of an architecture that supports functional programming was made with the intention that a completely verifiable system could be produced. SECD was chosen as a well understood and well documented architecture, and we had available a compiler for LispKit down to SECD code. We began with the

[2]Lispkit is a pure functional Lisp language subset, see Henderson's text [16].

assumption that the compiler can be shown to be correct, and that SECD executing compiled LispKit programs correctly implements the high level language.

The complete proof of SECD in HOL is far too large to be presented in a single paper. In other accounts, we have given the correctness proof of the translation schema from LispKit down to SECD [24], described the rig for running LispKit programs on SECD [26], documented the LispKit to SECD compiler [23], described the abstraction mechanisms between the various design levels of SECD [11] and explained the chip design [2], [12], [13], [14]. In the rest of this paper we sketch the HOL notation and give an introduction to carrying out proofs with HOL, describe the abstract SECD machine and explain its formal specification, sketch the design of the SECD chip and carry through a detailed look at a section of the proof of correctness of the SECD microcode. In a companion paper in this volume, Lars Rossen gives an account of another HOL based project — expressing Mary Sheeran's Ruby in HOL.

# 2 Introducing HOL

The verification work in the SECD project was carried out in the HOL system designed and written by Mike Gordon at the University of Cambridge. HOL implements a version of Church's typed higher order logic [3]. The main bonus of working with a higher order logic is that they permit clear, readable, and succinct specifications. Higher order features are particularly useful when it comes to joining sub-systems together and for describing architectures and abstractions. The downside is that with increased expressive power, less can be automated. In fact the HOL system is a proof-checker or proof-assistant and not an automated theorem prover. You have to supply to HOL the detailed working of what you think is a proof of the theorem at hand and HOL checks things out step by step. Well documented examples of HOL can be found in [4], [5], [6], [7], [8], [18] and [25].

The HOL logic is a conventional higher order logic. The HOL logic is interfaced to the ML language in which proof strategies can be encoded. ML has a very strong type discipline which is used to ensure that the only way to create a theorem is to prove it: they cannot be faked. Term and theorem are types in the implementation. The basic axioms are theorems and the only means of producing new theorems is through inference rules. Theorems are prefixed with a turnstile $\vdash$. The set of terms in basic HOL is defined by:

1. the bool constants $T$ (true) and $F$ (false) are terms
2. variables are terms



3. if $R$ and $S$ are terms, then so are $\sim R$ (negation), $R \wedge S$ (conjunction), $R \vee S$ (disjunction), $R$ ==> $S$ (implication), $R$ <=> $S$ (equivalence), and *( R )*.
4. if $x$ is a variable and *body* is a term, then *! x . body* (! stands for $\forall$), *? x . body* (? stands for $\exists$), *@ x . body* (@ stands for the select operator)[3], and *\ x . body* (\ stands for $\lambda$) are terms.
5. a function application of the form $R$ $S$ where $R$ and $S$ are terms (the rator and rand respectively) is a term
6. a conditional *b => R | S* where *b*, $R$ and $S$ are terms is a term. Read this as *if b then R else S*.
7. a local declaration *let x = expr in body* where *x* is a variable and *expr* and *body* are terms is a term. The local declaration is taken as a sugared version of *( \ x . body) expr*

The HOL system has a theory (library) facility where definitions and theorems can be stored. The HOL system comes equipped with several built-in theories. These include theories for numbers (0, 1, 2, 3, ...), lists, and pairs. The theories include the expected operators, e.g.: *+*, *-*, ***, ..., *=*[4], *<*, ..., *CONS, NIL, HD, TL, FST, SND*. HOL comments are enclosed in % signs.

All terms in HOL are typed and the type of a term is usually inferred by the system. The type of a constant is manifest. The type of a variable is defined by its usage, e.g. in $a \wedge b = 5$, *a* is of type *bool* and *b* is of type *num*. The type of a function from *** to **** is written ** -> ***, the type of a list whose members are all of type *** is ** list*, and the type of a pair is written ** # ***, where *** is the type of its first component and **** the type of its second component.

## 2.1 Conducting proofs in HOL

The HOL system is constructed from 5 axioms and 8 inference rules. New theorems may be proved from the axioms and previous theorems by applying inference rules. This method of proof, forward proof, suffers from two handicaps: (i) it is usually quite hard to know where to start, and (ii) there is a lot of bookkeeping associated with large proofs. HOL supports forward proof, but also has facilities for backward proof. In a backward proof, we start by supplying what we want to prove as a *goal*. Goals may be rewritten or split into simpler parts by *tactics*. Tactics may be thought of as inverse inference rules. A tactic splits a goal into subgoals but we are assured that if we can prove each

[3] The select operator is a higher order version of Hilbert's $\epsilon$-operator.
[4] The operator '=' is polymorphic

of the subgoals then there is a companion inference rule to that tactic which will prove the parent goal from these proofs. Suppose tactic $T$ takes goal $G$ to subgoals $g1$ and $g2$. Then there will be an inference rule $R$ which takes theorems $\vdash g1$ and $\vdash g2$ to the theorem $\vdash G$. For example, there is an inference rule *IMP_ANTISYM_RULE* which constructs the theorem $\vdash a = b$ from the theorems $\vdash a \Longrightarrow b$ and $\vdash b \Longrightarrow a$. Corresponding to *IMP_ANTISYM_RULE* there is a tactic *EQ_TAC* which when applied to the goal $a = b$ produces two subgoals $a \Longrightarrow b$ and $b \Longrightarrow a$. The idea behind a backward proof in HOL is that we recursively apply tactics to decompose a goal into a tree of subgoals until all the leaves are simple enough to prove. Then the HOL system will take our manually generated pattern of tactics and subgoals and transform it into a forward proof. HOL supplies several functions to manipulate goals. *g = set_goal* is used to initialise a goal stack to the next theorem to be proved; *e = expand* takes a tactic and applies it to the current goal; *b = backup* backs up the stack (throws away the last application of a tactic) and is very useful during proof explorations. If a tactic splits a goal into more than one goal (as with *EQ_TAC*) the subgoals are tackled one at a time. When one subgoal has been proved, the next to be proved is automatically popped.

## 2.2 Prove $\vdash(\exists x\ .\ (x = t) \wedge A\ x) = A\ t$

As an example, here is the proof in HOL of a primitive version of a theorem which is used to remove hidden wires in hardware proofs. We start by setting the goal and then split it into two implications using *EQ_TAC*.

```
#g "(? x:bool . (x = t) /\ A x) = A t";;
"(?x. (x = t) /\ A x) = A t"

#e(EQ_TAC);;
OK..
2 subgoals
"A t ==> (?x. (x = t) /\ A x)"

"(?x. (x = t) /\ A x) ==> A t"
```

It is easier to work with the implicands on the assumption list. We want this to happen for both subgoals so we back up and apply the compound tactic *EQ_TAC THEN STRIP_TAC* to the original goal.



```
#b();;
"(?x. (x = t) /\ A x) = A t"

#e(EQ_TAC THEN STRIP_TAC);;
OK..
2 subgoals
"?x. (x = t) /\ A x"
    [ "A t" ]

"A t"
    [ "x = t" ]
    [ "A x" ]
```

The pertinent assumptions are listed under each subgoal. We work on the subgoals one at a time. The textually lower one takes precedence. We can use the two assumptions to attack the goal. A simple-minded method is to discharge $A\ x$ and then rewrite what we have with the assumption $x = t$.

```
#e(UNDISCH_TAC "(A:bool->bool) x");;
OK..
"A x ==> A t"
    [ "x = t" ]


#e(ASM_REWRITE_TAC []);;
OK..
goal proved
. |- A x ==> A t
.. |- A t

Previous subproof:
"?x. (x = t) /\ A x"
    [ "A t" ]
```

*ASM_REWRITE_TAC [ ]* uses terms in the assumption list to perform rewrites, together with a few rules that are applied automatically. Extra theorems may be included explicitly in its argument list. The other subgoal is now popped. In this case we choose a suitable value for $x$ (if we choose an inappropriate value, the goal will be rendered unprovable, but the system remains sound) and the proof is completed with a simple rewrite from the assumption list.

```
#e(EXISTS_TAC "t:bool");;
OK..
"(t = t) /\ A t"
    [ "A t" ]

#e(ASM_REWRITE_TAC []);;
OK..
goal proved
. |- (t = t) /\ A t
. |- ?x. (x = t) /\ A x
|- (?x. (x = t) /\ A x) = A t

Previous subproof:
goal proved
```

The session is completed by storing away the theorem in the current theory under the name *EXISTS_ELIM*. *prove_thm* is a built-in theorem which takes a save name (in single back quotes), a goal (in double quotes), and a tactic from which it generates a forward proof of the theorem.

```
#let EXISTS_ELIM = prove_thm
  (`EXISTS_ELIM`,
   " ( ? x . (x = t) /\ ((A:bool->bool) x)) = A t",
   EQ_TAC THEN STRIP_TAC
   THENL
     [ UNDISCH_TAC "(A:bool->bool) x"
     ;
       EXISTS_TAC "t:bool"
     ]
   THEN ASM_REWRITE_TAC []
 );;
```

Note that when a tactic splits a goal into several subgoals we supply suitable tactic for each of the subgoals in a list following a THENL. When each subgoal tactic has a common tail (here *ASM_REWRITE_TAC [ ]*) we can merge these subgoal branches.

## 2.3 Specifying hardware in HOL

**Specification.** A specification treats a device as a black box. Typically, it states what can be observed on each output line in terms of current and previous



inputs. We express the signal value on line $p$ at time $t$ by $p\ t$, i.e. we take $p$ as a function from $num \rightarrow bool$. Using this notation, here is the specification of a combinational exclusive or gate

```
! a b out . xor2 a b out = (! t . out t = ~(a t <=> b t))
```

which is read as "the signals on lines *a*, *b*, and *out* are constrained by an xor2 gate to be in the relation that *out* at time *t* is high iff the signal values on *a* and *b* at time t differ".

The simplest sequential circuit is probably the unit delay which which simply delays the incoming signal by one time unit. The specification

```
! i out . del i out = (! t . out(t+1) = i t)
```

states that for all values of $t$ (t = 0, 1, 2, 3, ...), *out(t+1)*, the signal value on port *out* at time *t+1* equals that on the input line *i* at time *t*. The initial output value of this device is not defined. A variant on the theme is a clocked delay where the input is read only when its clock line goes high, otherwise the output remains the same (state is introduced). The specification of this device is

```
! i ck out . del2 i ck out = (! t . out(t+1) = (ck t => i t | out t))
```

We now specify the behaviour of a synchronous clocked register. From clock tick to clock tick, the n–bit register may reset to zero, parallel load, or remain as it is. The register has an explicit clock line and remains unchanged unless this clock line goes high. The 3 control signals are handled by a pair of control wires *cntl*. We use the pair notation $(c_0, c_1)$ to denote the value on the control line, and assign the following codes: *RESET = (T, T), LOAD = (T, F), NOP = (F, _)*. We adopt the notation *out t k* to represent the value on the wire $out_k$ at time *t*. It makes sense to bundle the input wires and the output wires together as buses. If we define a function *low k = F* we can use the principal of extensionality to remove the indices *k* from the specification which then reads:

```
! n ck cntl i out . nReg n ck cntl i out
   =  ! t . out (t+1)  = ck t /\ cntl t = RESET => low
                       | ck t /\ cntl t = LOAD  => i t
                       | out t
```

**Implementation.** Implementations are much easier to define: we list the components and how they are wired together. As an example here is the HOL definition of the implementation of an xor2 gate constructed from 4 nand2 gates.

```
! a b out . xor2_imp a b out
   = ? r q p . nand2 a b p /\ nand2 a p q
             /\ nand2 p b r /\ nand2 q r out
```

The only difficulty comes in handling the internal wires. What we need is a notation that ensures that hidden lines are defined (and can be typed) and yet are not visible outside the definition. We also need a logical inference rule which allows us to remove all occurrences of hidden lines when we unfold implementation definitions. This is where stronger versions of the rule *EXISTS_ELIM* are used. When unfolding an implementation, we first rewrite with the definitions of the parts (nand2 in the example above). This puts the implementation definition in standard form, conjuncts of the form ***output wire = expression*** with all hidden lines existentially quantified. These hidden lines can then be (pruned) eliminated by applications of *EXISTS_ELIM*. Here is the way the definition of the xor2 implementation is unfolded and pruned:

```
xor2_imp a b out

= ? r q p . nand2 a b p /\ nand2 a p q
          /\ nand2 p b r /\ nand2 q r out

= ? r q p . p t = ~(a t /\ b t)  /\  q t   = ~(a t /\ p t)
          /\ r t = ~(p t /\ b t)  /\  out t = ~(q t /\ r t)

= ? r q . q t = ~(a t /\ (~(a t /\ b t)))
          /\ r t = ~((~(a t /\ b t)) /\ b t)
          /\ out t = ~(q t /\ r t)

=  out t = ~((~(a t /\ (~(a t /\ b t))))
             /\ (~((~(a t /\ b t)) /\ b t)))
```

We are left with an expression solely in terms of input and output values and ripe for simplification by the normal rules of logic.

To be a little more precise with HOL terminology, *UNFOLD*ing substitutes the expression defining the value of a signal for that signal throughout the rest of a term. *PRUNE*ing removes the quantified variable when its only remaining occurrence in the term is the left hand side of an equation.



# 3 The abstract SECD machine

Henderson's SECD machine has three load functions (one each for variables, constants, and in-line functions), *AP* to enter a non-recursive function call, *DUM* and *RAP* to enter recursive function calls, *RTN* for function exit, *SEL* and *JOIN* to support conditional branching, 4 built-in list operators, 7 comparison or arithmetic operations, and a halt instruction. We characterize the behaviour of an operating abstract SECD machine by giving its state at the end of an instruction. The state is a 4-tuple listing the contents of the 4 stacks that give it its name, *(S, E, C, D)*.

| | | | |
|---|---|---|---|
| LD (m.n): | s e •.c d | ⇒ | (x.s) e c d<br>where x = locate((m.n),e) |
| LDC x: | s e •.c d | ⇒ | (x.s) e c d |
| LDF c': | s e •.c d | ⇒ | ((c'.e).s) e c d |
| AP: | ((c'.e') v.s) e •.c d | ⇒ | nil (v.e') c' (s e c.d) |
| DUM: | s e •.c d | ⇒ | s (nil.e) c d |
| RAP: | ((c'.e') v.s) (nil.e) •.c d | ⇒ | nil e" c' (s e c.d)<br>where e" = replcar(v, e') |
| RTN: | (x) e' (•) (s e c.d) | ⇒ | (x.s) e c d |
| SEL cT cF: | (T.s) e •.c d | ⇒ | s e cT (c.d) |
| | (F.s) e •.c d | ⇒ | s e cF (c.d) |
| JOIN: | s e (•) (c.d) | ⇒ | s e c d |
| CAR: | ((a.b).s) e •.c d | ⇒ | (a.s) e c d |
| ADD: | (a b.s) e •.c d | ⇒ | ((b + a).s) e c d |
| STOP: | s e (•) d | ⇒ | s e (•) d<br>Stop. Result is s. |

CAR *is typical of the unary operations;* ADD *is typical of the binary operations*

Table 1: SECD code transitions

Broadly speaking, expressions are evaluated on *S*, *E* is the environment (a list of lists — each dynamically nested function introduces a new level of nomenclature, and several items may be defined at each level), and the SECD code is held in *C*. The old machine state *(S, E, C)* is pushed onto *D* when a new function is entered, and restored therefrom on return. The operation of the abstract SECD can therefore be specified by a transition table which shows how each instruction affects the state 4–tuple. For example under *AP* in table 1 we find *((c'.e') v.s) AP.c d ⇒ nil (v.e') c' (s e c.d)*. The compiler sees to it that prior to executing an *AP* instruction, we have placed the argument list *v* on the stack *s*, and also a function closure (the function code *c'* and its environment *e'* as a dotted pair on *s* above *v*). The effect of executing *AP* is to: set the stack *S* to empty; set a new environment by augmenting the defining environment for

the function with the arguments to this call *(v.e')*; set the code pointer to *c'*, the function body; and to save *s e c* on the dump.

# 4 Designing the SECD

In this section we discuss transforming the abstract machine description into a microelectronic device. First we describe the chip interface, next the interpreter which was derived from the specification by pattern matching; then the register transfer level which was derived from the interpreter after basic architectural questions had been resolved. In the final subsection, we state the purposes of the 4 major floor plan elements in the SECD design.

## 4.1 The chip interface

Realising the SECD machine as a working system required that it be able to accept a task, compute a result, and return it. The interface to permit a user to pose a problem and the machine to return a result was the first major design decision. Two major options were considered: a co-processor role and a stand-alone system[5].

Using the SECD as a co-processor to another system would permit i/o to be handled by the other system rather than the SECD chip. For instance, using SECD as a co-processor for a SUN workstation would see the SUN able to read in an S-expression, set up a memory image for the problem, signal the SECD to begin computation, receive a signal back on completion of the calculation, and print out the S-expression solution. This has the advantage of simplifying the tasks the SECD must perform.

The second option considered was that of a stand-alone system. This would require incorporating primitive read and write operations into the definition of the machine. Ideally, an operating system for such a system would be written in the higher level LispKit language, but its pure functional nature provides recursion as the sole means to achieve the infinite 'while' loop required, but since every recursion uses up system resources, this would exhaust memory. A possible solution would be to hand modify the machine code to create the desired loop.

An extension of the stand-alone system concept is the multi-processor SECD system. Such a system was envisioned as having multiple (perhaps 100) SECD

[5] This discussion summarizes work by Jeff Joyce on system configuration, reported originally in [17].



chips operating concurrently on shared memory. An operating system kernel would assign S-expressions to processors for evaluation. Each processor would be assigned exclusive use of memory blocks in a sort of virtual memory system, with blocks being assigned and garbage collected by the kernel.

The co-processor option was chosen as most appropriate to the scope of the project. If desired, a stand-alone approach could be used for a second iteration of the design.

Representation of S-expressions (the "stuff" of programs and data in the SECD machine) was not determined at this level, but it was known that they would be stored within a finite memory, and this necessitated garbage collection. In practice, a simple *mark and sweep* garbage collector was used. Thus, the SECD system consisted of a microprocessor operating upon an S-expression image in a RAM. Major phases in the operation of the system are:

- load problem into RAM (done by main processor)
- send *start* signal to SECD system
- run
- stop and signal completion to main processor
- return result (again, main processor task)

A four state finite state machine model was used to express the desired operating interface. Several ideas were incorporated including the addition of a *reset* input to permit a deterministic startup of the machine, an *idle* state for waiting until a problem is loaded, a *top_of_cycle* state as the top of a fetch/execute cycle, two *error* states, and a *button* input to control transitions from *error* and *idle* states. This view is summarized by Figure 1.

## 4.2 The SECD interpreter

We designed SECD with ease of verification in mind. Since the total verification effort was known to be large, we settled on a simple architecture and a simple clocking strategy. We still underestimated the total verification effort by a large margin.

Perhaps the most interesting feature of the design effort was realising how much progress could be made by transforming the initial specification of SECD once a few architectural decisions had been made. Henderson [16] pointed out that it is possible to derive an interpreter for the SECD machine by noting how individual instructions update the *(S, E, C, D)* state 4-tuple as they are executed. We show how this is done for the *LDF* instruction, an instruction of medium complexity whose verification we will be tracking in the sequel.

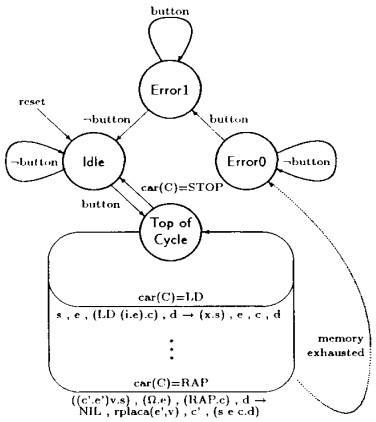



Figure 1: Top Level Finite State Machine View of SECD

$$s\ e\ (\mathbf{LDF}\ f).c\ d \Rightarrow ((f.e).s)\ e\ c\ d$$

A new computation is entered when an **LDF** instruction is carried out, The new computation builds a function closure *(c'.e)* on top of $S$ finding the body of the function in-line, pairing it with the current environment $e$, and pushing the new pair onto $S$. The instruction $LDF\ f$ is then peeled off the code sequence. $E$ and $D$ are not affected by the instruction. By pattern matching, $LDF$ = *car(car C)*, $f$ = *car(cdr C)*, and the next instruction is located at *cdr(cdr C)*.

```
S := cons(cons(car(cdr C), E), S);
E := E;
C := cdr(cdr C);
D := D;
```

In general we must take some care about the order in which we update the fields



of the state 4-tuple. Here it is important to evaluate $S$ before we evaluate $C$.

### 4.3 The register level view

We now refine this top level view into an architecture. For ease of implementation and verification, we settled on a single bus architecture and an external RAM memory $M$. The SECD stacks are registers pointing to addresses in the memory. The memory is initially chained together and referenced by another register *FREE*. Access to the external memory $M$ is via a register *MAR*. A *read* operation places *M[MAR]* on the bus; a *write* operation stores the value on the bus in *M[MAR]*. We decided to go for 32-bit words in this implementation. With type tags taking 2 bits and garbage collection flags another 2 bits per word, that leaves space for either a 28-bit two's complement number or a cons'ed word containing two 14-bit addresses. This is a sufficient address space for running the LispKit compiler on the SECD machine itself.

We now refine the interpreter level instructions down towards the register-register level. Since *car*, *cdr*, and *cons* operations are very common we anticipate special hardware for them. At this stage we mechanically rewrite the operations for $LDF$ into smaller steps, each step being of the form $R$ := *rhs;* where $R$ is a register and *rhs* is either a register, or a single *car*, *cdr* or *cons* operation on a register. Much of the time we are assembling sub-expressions in order to cons them. We simplify matters by allotting two general purpose temporary registers called *X1* and *X2*. When we are carrying out a *cons* operation, we put the second argument in *X2* and the first argument in *X1* (since we have a functional language, their order of evaluation is immaterial).

```
S := cons(cons(car(cdr C), E), S)
X2  :=  E;
X1  :=  cdr S;
X1  :=  car X1;
X1  :=  cons X1 X2;
X2  :=  S;
S   :=  cons X1 X2;
C := cdr(cdr C)
C   :=  cdr C;
C   :=  cdr C;
```

In the main, each register-register operation is split into a read onto the bus and a write from the bus at the bus level. Each register $X$ has two control signals associated with it, systematically called *rX* and *wX*. When *rX* goes high, the contents of $X$ are read onto the bus. The controller sees to it that only one register may be read onto the bus at a time and for simplicity, only

one register may be written from the bus at a time. When *wX* goes high, the contents of the bus are written into *X*: all the bits of the bus if *X* is a 32-bit register; if *X* is a 14-bit register, the top 18-bits of the bus are ignored except for the special *CAR* register which is wired to accept bits 14-27 of the bus. Bread and butter microcode can be generated directly from the register-register level description as soon as we supply a template for each class of assignment in the register-register level description.

- Copying the contents of register *X* to register *Y* is quite simply

```
rX      wY
```

Movement involving *cdr*, *car* or *cons* require access to the external memory. The first two cases are have simple and short templates; *cons* is rather more involved and is implemented as a micro-code subroutine.

- To carry out the request *Y := cdr X* we place the contents of *X* in *MAR*, read *M[MAR]* onto the bus (all 32 bits), and then store its *cdr* field in Y.

```
rX      wmar
rmem    wY
```

- To carry out the request *Y := car X* we place the contents of *X* in *MAR*, read *M[MAR]* onto the bus (all 32 bits), then filter out the *car* bits via the *CAR* register before passing the result to Y.

```
rX      wmar
rmem    wCAR
rCAR    wY
```

- The much used *cons* operation requires more thought. A call on *cons* requires the freeing of a fresh cell in memory (perhaps initiating a garbage collection) and the passing of two addresses to memory to fill out its fields. We have chosen to restrict *cons* to operating upon two specific registers which we call *X1* and *X2*. Our *consX1X2* operation locates a free cell in memory, fills out its two address fields from *X1* and *X2*, sets its type bits to indicate a cons'ed word, zeroes its garbage collection bits, writes it to the memory, and returns its address. Calls to *consX1X2* occur so frequently in the code and are of such complexity that we inserted a subroutine facility into the micro-code.

```
if (null rfree) then GCBegin()
rfree    wmar
rmem     wfree
rcons    wmem
```



At the end of this analysis we have seen the need for two auxiliary registers and micro-code routine for consing their contents and placing the address of the new cell on the bus. Figure 2 summarizes the register-register level view of the chip design. The control part of the design is shown as a classic finite state machine, with a state register (the *MPC*), a microcode *ROM*, and a *DECODE* component. Several programs from Gabriel [9] and Henderson [16] were run as simulations at the Mossim level. The largest test was the compilation of the LispKit compiler from Henderson.

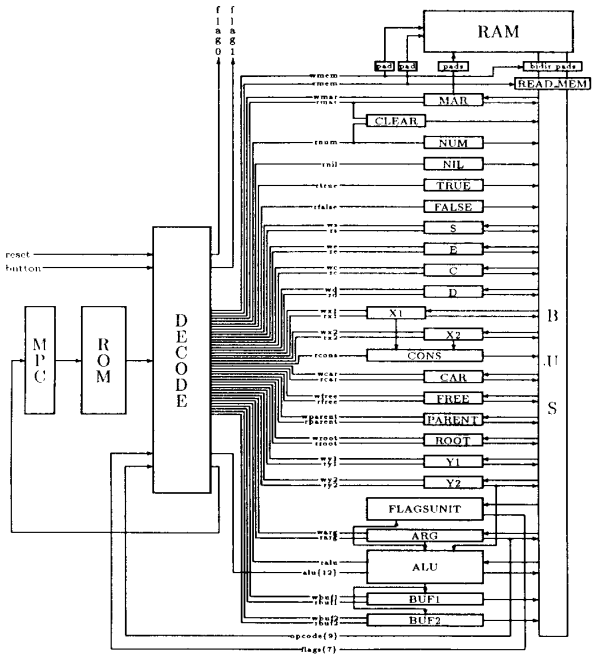



Figure 2: Register Transfer Level View of SECD Machine



## 4.4 SECD layout

The SECD chip has 4 major functional components: the **control unit**, the **datapath**, the **shift registers** and the **pad frame** as depicted in Figure 3.

1. The *control unit* interprets SECD machine instructions, breaking them up into a stream of micro-instructions to be effected one at a time by the *datapath* unit. It is conceived as a finite state machine whose state is held by a micro program counter (MPC) register which always refers to the current micro-control step. Inputs to the *control unit* include: external *reset* and *button* signals, status *flags*, and a 9-bit *opcode* (the code of the current machine instruction). Outputs include *read* and *write* signals for registers within the *datapath* and also for memory, *alu* control signals, the bi-directional pad control signal *bidir*, and the two state flags *flag0* and *flag1*.

2. The *shift registers* provide a means of entering test vectors and examining the state of the chip, giving us a passable ability to test the chip in operation. They also permit independent testing of the *datapath* and *control unit* components. Most signals passing between the *control unit* and the *datapath* are routed through the *shift registers*. These include *read* and *write* signals, *alu* signals, and status *flags*. For observability and controllability, it also routes some signals which are functionally internal to the *control unit* (specifically the 5 select signals, and the *mpc* register contents). Thus all these signals may be read and/or altered, though in normal operation, signals pass through uninterrupted. The *shift registers* use a clock input independent from the system clock.

3. The *datapath* executes simple micro-operations signalled by the *control unit*. The *datapath* unit is built as an ensemble of devices – registers, the arithmetic unit, and memory – communicating via a common *bus*. The operations performed by the *datapath* unit include: copying the value of a selected register onto a *bus*, storing a value from the *bus* into a selected register, the list manipulating functions cons, car, cdr, the arithmetic operations of addition, subtraction, decrementation, and setting status flags. Besides read and write lines and the arithmetic operator select lines, the only other input control line is the clock ($\Phi_A$, which clocks the writing of registers). Outputs include the status *flags*, the memory address register (for the off chip RAM), the lower 9 bits of the *arg* register (the *opcode*) and the 32 bit *bus*.

4. The chip is framed by a set of *I/O pads* which connect the chip to the outside world. Bidirectional i/o pads connect the data bus to the external memory. The default mode for the pads is input, switching to output

mode only when writing to memory. A set of busgates isolates the input from the datapath bus unless a read memory operation is signalled. A 14 bit *mar* exported off chip from the datapath selects the memory location. The SECD chip is packaged in a 64-pin DIP. External (off chip) signals fall within three groups: signals used for normal operation of SECD, pwr/gnd, and SECD testing signals.

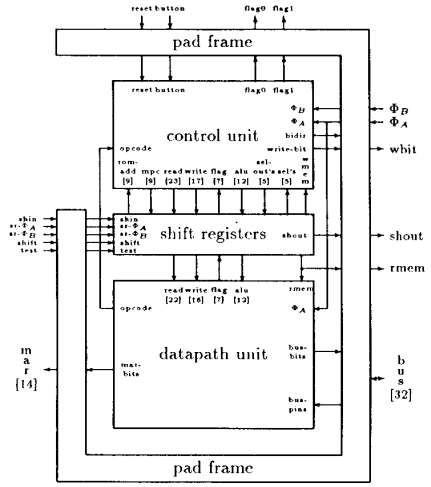



Figure 3: SECD Chip Major Subcomponents



# 5 SECD Formal Specification

Three levels of description of the SECD chip have been specified in HOL: the **low level** consists of primitive gates (AND, OR, latch, etc.) and regular structures (ROM's, PLA's), a **register transfer level** (rtl) model, consisting of multi bit registers and functional components such as ALU's, etc. and the **top level** specification, a representation of the abstract SECD architecture defined earlier. We will focus on the rtl and top levels, limiting discussion of the low level to a few key elements that impact the sequel. We adopt a convention of subscripting the explicit time parameters to indicate the granularity of time associated with each level, low level (fine) grain time $t_f$, rtl (medium) $t_f$, and top level (coarse) $t_c$.

## 5.1 The Low Level Definition

The low level definition uses simple gates as base components, and bears a one-to-one relation to the layout of gates in the SECD chip. Specifying the memory elements (latches) must be understood in relation to the granularity of time associated with this level of description. We use the coarsest granularity of time that captures the behaviour of the clocks: using 4 points per clock cycle. Constraints on clock behaviour require that the separate system and shift register clock signals are not cycled simultaneously, and that when started, the clocks always complete a full cycle *i.e.* the system clock cycle is never interrupted by a series of shift register clock cycles, and vice-versa. All latches on the SECD chip are level-triggered. Additionally, we require the *reset* input be asserted when the system clock starts cycling, so that we have a deterministic state on startup.

The standard design regimen for 2-phase non-overlapping clocking, that of linking memory devices clocked on opposite clock phases with combinational logic, was violated in one part of the SECD chip design. Two registers clocked on the same clock phase are wired in series, with the output of the first feeding the input of the second through strictly combinational logic. This odd circuit feature will degrade chip performance, but at sufficiently slow clock rates will still function properly, since the output of level triggered latches change at the start of the clock pulse, and no circular feedback path exists. This single piece of questionable design affected the entire chip specification, and added an obligation to prove the impossibility of circular feedback. A somewhat nonstandard definition of the primitive latch was required to express this behaviour appropriately, expressing present state as a function of the clock signal at the same moment, instead of the previous moment. Notice the difference between this definition and that of *del2* on page 10.

```
latch clk inp state =
  !t. state t = (clk t) => inp t | state (PRE t)
```

## 5.2 Register Transfer Level

The Register Transfer Level (rtl) model of the chip has a similar component hierarchy as the low level view, but does not have the same depth. It has 3 major components: the control unit, the datapath, and the pad frame (*CU*, *DP*, and *rt_PF* respectively). The *rt_PF* definition is the proven behaviour of the padframe component of the lower level. The *DP* is the conjunction of the same set of components as the low level view, but the behaviour of each component replaced the primitive definitions. The *CU* differed most from the lower level. The layout hierarchy was replaced by one that better reflected its functionality, using three components for its definition: a *state register*, a microcode *ROM*, and a *decode* section. Conjoining the definitions of these 3 parts gives a typical finite state machine behaviour, defining the next state and current output values as a function of the current state and current input values. Notably absent from this view is the shift register component. Under normal operation, these are intended to be transparent to the operation of the chip, and thus constraining their controls appropriately enables us to abstract it out of this level of description completely.

The external *memory* is represented only at the register transfer level, and the conjunction of the chip description and a simple memory with *Fetch* and *Store* commands constitutes the system at this level.

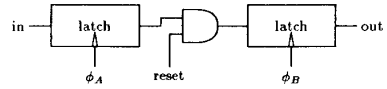
### 5.2.1 Temporal representation

The time granularity at this level corresponds to clock cycles. Selecting which point of the cycle to use for this time grain was complicated by the outputs of registers clocked on different clock phases feeding into a single register (*i.e.* this is the case when two registers clocked on the same phase are wired in series). The part of the clock cycle over which the output of the register is stable is determined by the clock phase used to clock the register. In order to have a uniform temporal abstraction for the time parameter for each input type, we selected the time that the first clock phase is asserted as the point at which the rtl state is defined.

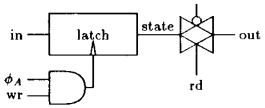
Two different types of registers are used. The control unit uses pairs of latches clocked on opposite clock phases, while the datapath registers clock write operations on $\phi_B$ to prevent transitory spikes on state transitions from



causing unwanted writes. The low level circuit and the rtl specification for each follow.




```
MPC9 clk in out =
  (out 0 = #000000000) ∧
  (!t_m. out (t_m+1) = clk t_m => in t_m | state t_m)
```




```
reg in out clk wr rd state =
(!t_m. state t_m =
       clk t_m => (wr t_m => in t_m | state (PRE t_m))
                | state (PRE t_m)) ∧
(!t_m. rd t_m ==> (out t_m = state t_m))
```

In both definitions, the clock signal *clk* is an abstraction of the two distinct clock phases ($\phi_A$ and $\phi_B$), and is asserted when the system clock cycles. Note that the *mpc* register specification does not include the *reset* input explicitly. However, constraints on the *reset* signal force an initialisation at power up, and this constraint and the signal appear as a determined value in the register at time $t_m = 0$. The typical datapath register has a gated output, and is clocked by both the system clock phase and the specific write signal. The clock signal was included here to permit eventual extension of the verification to the test mode of chip operation. The difference in the definition of the registers and the choice of point on the time cycle can be seen from the timing diagram in Figure 4.

The *reset* signal is held low at the start of the clock operation. The mpc register content at $t_m+1$ is a function of the inputs at time $t_m$. If the *arg* register (a typical datapath register) content at time $t_m$ feeds into the first *mpc* latch at the same time, this relation holds, as it also does if the *mpc* content itself feeds back into this latch. Sampling one point earlier would have required distinct temporal abstractions for the signals arising from registers clocked on different clock phases.

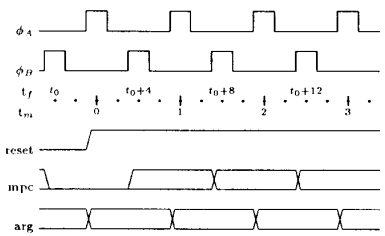



Figure 4: Relating low and rtl times

### 5.2.2 The control unit

The state register component for the control unit includes not only the *mpc* register, but a 4-deep *stack* for microcode subroutine calls as well. The definition of these registers is altered slightly, with no provision for resetting. The clock signal fed to these latches is the same system clock, and the load signal is the OR of the *pop* and *push* control signals, so latching only occurs when a stack command is issued.

```
Stack_register9 Clocked load in_sig out_sig =
 !t_m. out_sig (t_m+1) =
       Clocked t_m => (load t_m => in_sig t_m | out_sig t_m)
                    | out_sig t_m
```

### 5.2.3 The datapath

The datapath consists of the conjunction of the components shown in Figure 2. The bus is expressed as the wiring together of the components. All registers are instances of the previously shown definition. There are 3 elements that are worth a closer examination: the *FLAGSUNIT*, the *ALU*, and the *READ_MEM* component.

The *READ_MEM* component is a ***busgate*** (a set of transmission gates), which connect the bus to the bidirectional i/o pad output. This is required since



the pads default to input operation, but only when a read memory instruction is asserted can the input be permitted to drive the bus. A ***busgate*** is defined as follows:

```
busgate in_val rd out =
      !t. (rd t) ==> (out t = in_val t)
```

The *ALU* can perform 9 distinct operations. Six of these are used exclusively in garbage collection: ***replcar***, ***replcdr***, ***setbit31***, ***setbit30***, ***resetbit31***, and ***resetbit30***. The remaining operations are the arithmetic operations: ***add***, ***sub***, and ***dec***. The ***mul***, ***div***, and ***rem*** operations were not implemented, and selecting any of these defaults to the ***dec*** operation. The output of the *ALU* is gated onto the bus controlled by the ***ralu*** signal. If no ***alu*** operation is selected, the value computed is not significant, and thus we define this component by the use of implication, typically:

```
add t ==>
  (alu t = ADD28 (atom_bits(arg t)) (atom_bits(bus t)))
```

(The *ADD28* function is the specification for the operation performed by the low level component, and matches what is specified at the top level.) Just as the behaviour for no operation is of no concern, so the output when more than one operation is selected should be undefined. This is expressed in the specification by the use of a ***one_asserted*** predicate which implies the desired behaviour. This predicate states that if one of the signals is asserted, the remainder are not.

Finally, 6 status flags emerge from the *FLAGSUNIT* : ***atomflag***, ***bit30flag***, ***bit31flag***, ***zeroflag***, ***eqflag***, and ***leqflag***. The ***bit*** flags are used by the garbage collector. The ***atomflag*** indicates if the value on the bus is an atom record, as opposed to a cons cell record. The unary flag operations operate on the bus value, the binary use the content of the ***arg*** register in addition.

### 5.2.4 The padframe

The padframe definition includes input, output, and bidirectional pads. The input and output pads are defined by stating the equality of the pairs of nodes connected by the pads. The bidirectional pads are defined by:

```
(bidir t => (mem_bits t = bus_pins t)   % read memory %
          | (bus_pins t = bus_bits t)) % write memory %
```

where *bus_pins* are the off-chip side, *bus_bits* are the bus nodes, and *mem_bits* are the nodes linking the pads to the input of the *READ_MEM* unit described earlier.

### 5.3 The Top Level Specification

At the most abstract level, the SECD machine is defined in terms of transformations to S-expressions in the 4 stacks, as shown previously in Table 1. A formal specification of the top level behaviour is ideally defined in terms of transformations to an S-expression data type that closely resembles this elegant definition. The closer the resemblance the better we are assured that the HOL specification is equivalent.

The method used by SECD for implementing recursive function definitions as closures with a circular environment component (see [16]) raises the complexity of the data representation problem considerably. Such circular S-expression lists, created by a destructive operation, cannot be mapped to a simple recursive data type. Further, structure is shared by S-expressions, particularly the environment component of closures. Mutually recursive function closures each reference the same environment, which is also in the $E$ stack. When a destructive replace operation is performed (by executing a RAP instruction) to create the circular list structure, the change affects the common component of all closures simultaneously.

Thus, a much lower level of representation has been chosen to describe the top level specification. Rather than directly defining transformations on structures of an S-expression data type, we define an abstract memory type which can contain representations of S-expressions. Further, we define a set of primitive operations upon the memory which correspond to the operations on S-expressions, namely *cons*, *car*, *cdr*, *atom*, *replcar*, *dec*, *eq*, *leq*, *add*, and *sub*. The 4 state registers contain values that reference the appropriate S-expression representation. Finally, an additional *free* register containing a value to access the free list structure is needed to define the *cons* operation. The state of the machine is then defined by a tuple:

$$(S, E, C, D, Free, memory, FSM\_state)$$

where the FSM_state is one of the 4 major states of the top level finite state machine view of the machine (as in Figure 1).

The abstract memory type $\mu$ is basically a function: $\mu = \delta \rightarrow (\delta \times \delta \cup \alpha)$ where $\delta$ is the domain of the function and $\alpha$ is the set of atoms: $\alpha = integers \cup symbols$. The set of symbols includes the symbolic constants: *T*, *F*, and *NIL*. The domain of the function is chosen to be the type of 14 bit words (*word14*, a



14 bit instance of the general *wordn* type of fixed width bus values), matching the type that is used by the implementation definition. We extend the definition of memory to incorporate garbage collection features, by adding *mark* and *field* bits to each cell:

$$\mu = \delta \rightarrow ((bool \times bool) \times (\delta \times \delta \cup \alpha))$$

Additionally, we include the *replcdr*, *setf*, and *setm* operators used by the garbage collector, as well as a *Garbage_collect* function, which is left undefined for the first proof attempt. Extractor functions *mark*, *field*, *Int_of*, and *Atom_of* are provided for the values returned by the $\mu$ function. The relevant built-in functions and their types are summarized in Table 2.

| Operation | Type |
|---|---|
| Car, Cdr | $(\delta \times \mu \times \delta) \rightarrow (\delta)$ |
| CAR, CDR | $(\delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$ |
| Cons, Cons_tr | $(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$ |
| EQ, LEQ | $(\delta \times \delta \times \mu \times \delta) \rightarrow bool$ |
| ADD, SUB, MUL, DIV, REM | $(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$ |
| Replcar, Replcdr | $(\delta \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$ |
| setm, setf | $(bool \times \delta \times \mu \times \delta) \rightarrow (\delta \times \mu \times \delta)$ |
| mark, field | $\delta \rightarrow \mu \rightarrow bool$ |
| Int_of | $(\delta \times \mu \times \delta) \rightarrow integer$ |
| Atom_of | $(\delta \times \mu \times \delta) \rightarrow \alpha$ |
| Atom | $(\delta \times \mu \times \delta) \rightarrow bool$ |
| Is_int | $(\delta \times \mu \times \delta) \rightarrow bool$ |
| Is_TRUE | $(\delta \times \mu \times \delta) \rightarrow bool$ |
| Garbage_collect | $(\mu \times \delta) \rightarrow (\mu \times \delta)$ |

Table 2: Primitive Operations on Abstract Memory Data Type

As seen in the table, many of the functions return triples, consisting of a memory cell, a memory, and a second cell which represents a free list pointer. Operations such as *Car*, *Cdr*, *EQ*, *LEQ*, etc. do not alter memory, while *Cons*, *ADD*, *SUB*, *setm*, *setf*, *Replcar*, *Replcdr* do alter one cell in the memory, and thus must return the new memory. In order to permit composition of the primitive memory operations, we provide the *CAR* and *CDR* functions which return the unaltered memory and free pointer. For example, to access an argument to a *LD* command, we can write the following:

$$let\ m = Int\_of(CAR(CAR(CDR(c, MEM, free)))).$$

In Table 3 we provide the top level transition specification for the *LDF* instruction. Following this, in Table 4 the set of 21 such transitions are used to define the next state of the machine for each instruction, as well as the top level specification for the SECD. The top level function, *SYS_spec*, closely resembles the top level finite state machine of Figure 1, with 4 states, 2 possible

```
LDF_trans (s:δ,e:δ,c:δ,d:δ,free:δ,MEM:μ) =
  let cell_mem_free =
    M_Cons_tr(s, M_Cons_tr(e, M_CAR(M_CDR(c,MEM,free))))
  in
  (cell_of cell_mem_free,                          % s %
   e,                                              % e %
   M_Cdr(M_CDR(c, mem_free_of cell_mem_free)), % c %
   d,                                              % d %
   free_of cell_mem_free,                          % free %
   mem_of cell_mem_free,                           % memory %
   top_of_cycle)                                   % state %
```

Table 3: Transition for LDF Instruction

transitions from 3 of them, and 18 transitions from the *top_of_cycle* state, one for each (implemented) machine instruction. The system must be assured of starting out in the *idle* state. The given types of all system parameters have the subscript "$_c$". This identifies them as signals sampled at a *coarse* grain of time.

The type $\mu$ was defined in HOL as the abstract data type *mfscrp_mem*, and the type $\alpha$ was defined as the type *atom*. Both types have associated *REP* and *ABS* functions to map between the abstract and representing types. For example, the term *"REP_mfscrp_mem(memory:(word14,atom)mfscrp_mem)"* has the type:

$$word14 \rightarrow ((bool \times bool) \times (word14 \times word14 \cup atom)).$$



```
NEXT (s:δ,e:δ,c:δ,d:δ,free:δ,MEM:μ) =
 let instr = M_int_of(M_CAR(c,MEM,free))
 in
 ((instr = LD)   =>   (LD_trans   (s,e,c,d,free,MEM)) |
  (instr = LDC)  =>   (LDC_trans  (s,e,c,d,free,MEM)) |
  (instr = LDF)  =>   (LDF_trans  (s,e,c,d,free,MEM)) |
    ...
  (instr = LEQ)  =>   (LEQ_trans  (s,e,c,d,free,MEM)) |
% (instr = STOP) %    (STOP_trans (s,e,c,d,free,MEM)))
```

```
SYS_spec (s:δ_c) (e:δ_c) (c:δ_c) (d:δ_c)
         (free:δ_c) (MEM:μ_c)
         (button:bool_c) (state:state_c) =
 (state 0_c = idle) ∧
 ! t_c.
  ((s(t_c+1), e(t_c+1), c(t_c+1), d(t_c+1),
    free(t_c+1), MEM(t_c+1), state(t_c+1)) =
   ((state t_c = idle)
    => (button_pin t_c
       => (M_Cdr(M_CAR(NUM_addr, MEM t_c, free t_c)),
           NIL_addr,
           M_Car(M_CDR(NUM_addr, MEM t_c, free t_c)),
           NIL_addr, NIL_addr, MEM t_c, top_of_cycle)
        | (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c, idle))
   |(state t_c = error0)
    => (button_pin t_c
       => (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c, error1)
        | (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c, error0))
   |(state t_c = error1)
    => (button_pin t_c
       => (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c, error1)
        | (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c, idle))
   |% (state t_c = top_of_cycle) %
    (NEXT (s t_c, e t_c, c t_c, d t_c, free t_c, MEM t_c)))
```

Table 4: Top Level Specification

### 5.3.1 Memory Abstraction

Abstracting from the implementation memory to the abstract memory type maintains the mark and field bits unchanged, and maps the 28 bit field to the appropriate *cons*, *integer*, or *symbol* record based on the record type bits. The memory abstraction function is defined in two steps: first a function maps records in the range of the implementation memory into the range of the representative type for abstract memories, and then *ABS_mfsexp_mem* is applied to this function composed with the implementation memory:

```
Mem_Range_Abs:word32->((bool#bool)#((word14#word14)+atom))w =
 ((mark_bit w),(field_bit w)),
  ((is_symbol w) => INR(Symb(Val(Bits28(atom_bits w))))
  |(is_number w) => INR(Int(iVal(Bits28(atom_bits w))))
                  | INL(car_bits w, cdr_bits w))

mem_abs (M:word14->word32) = ABS_mfsexp_mem(Mem_Range_Abs o M)
```

The Mem_Range_Abs function is total, and the unused record type (#01) gets mapped to the *cons* type record of the abstract memory. This ensures that the result of composing it with an implementation memory always returns an object in the representative type of the abstract memory, and thus applying *REP_mfsexp_mem* to *mem_abs M* at some value *v* will be *Mem_Range_Abs (M v)*. It further ensures the desirable property that every non-atomic record is a *cons* type record.

### 5.3.2 Temporal Abstraction

The coarsest grain of time used to describe the system corresponds to the points when the system is in major states of the top-level FSM (*Idle*, *Error0*, *Error1*, and *Top_of_Cycle*). We map from this coarse granularity to the medium grain points of time when specific microcode addresses are in the *mpc* register. The mapping is not a linear function, since the number of cycles needed to execute any machine instruction varies, and can vary between executions of the same instruction. The latter differences arise due to garbage collection calls during instruction execution, as well as varying search distances required to load values from the environment. The definition of the required mapping functions is well described in [21].

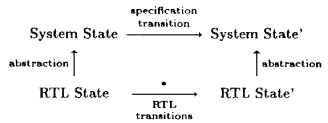


# 6 Verification of the SECD Design

Within the accuracy of the model of the lowest level of description, and subject to given constraints, a proof of correctness will show that a lower level description ensures the behaviour of the higher level of description. Parameters to the higher level description are abstractions of the lower level ones, with regard to temporal granularity and data type. Thus, we can view each level description as a specification, and also as an implementation of the higher level specification. Using this terminology, the goal of a proof generally looks like:

constraints ⊃
implementation (state) (inputs) (outputs) ⊃
specification (abs o state) (abs o inputs) (abs o outputs)

For the register transfer to top level specification proof, the abstraction must map between two granularities of time, as well as different data types, particularly for the memories. The register transfer definition includes a memory function with simple Fetch and Store operations only, while the top level uses an abstract memory data type. The task of the verification is to show that the sequence of operations performed at the register transfer level commutes with the specification transition at the abstracted top level.




A complete verification of the SECD system from lowest level description to top level specification requires correctness theorems relating the low and register transfer levels, and one relating the register transfer and top levels. The first of these will not be discussed in this paper.

This section of the paper addresses the constraints on the scope of the proof, the form of the correctness goal, and the approach used in achieving the proof. Each major step in the proof is outlined, using the LDF transition as the running example.

## 6.1 Constraints

Four major constraints limit the scope of proof:

**clock_constraint** We restrict the proof to normal operating mode (*i.e.* the shift registers are not operating).

```
clock_constraint SYS_Clocked = ! t_m. SYS_Clocked t_m
```

**reserved_words_constraint** There are 3 reserved words in memory that contain the symbolic constants for *NIL*, *T*, and *F*.

```
reserved_words_constraint mpc memory =
 !t_m. (state_abs (mpc t_m) = top_of_cycle) ==>
  ((memory t_m NIL_addr =
    bus32_symb_append #00000000000000000000000000000000) ∧
   (memory t_m T_addr   =
    bus32_symb_append #00000000000000000000000000000001) ∧
   (memory t_m F_addr   =
    bus32_symb_append #00000000000000000000000000000010))
```

**well_formed_free_list** Informally, we want the free list to be a *cdr* linked list, containing only *cons* cells, aside from the last cell which is *NIL*. Only cells that are not used in any data structure that is part of the computation (*i.e.* cells accessible from the *s*, *e*, *c*, or *d* registers), and cells that are not reserved words should appear in the free list. Finally, cells should only appear in the list at most once. For the formal specification, we limit the constraint to the maximum number of *cons* operations in any SECD instruction: the AP instruction performs the operation 4 times. The requirement that the cells be *cons* records ensures the symbolic constants are not in the list. The function *nth* applies its 2nd argument *n* times to the last argument.

*free_list_constraint*, derived from this and the previous constraint, simply asserts that the first 4 pointers of the free list are not *NIL_addr*, and is used in the microprogramming level proofs.

```
well_formed_free_list mpc free memory =
 ! t_m. (state_abs (mpc t_m) = top_of_cycle) ==>
 ! n. (n < 4) ==>
   ((is_cons (nth n ((memory t_m) o cdr_bits)
                    (memory t_m(free t_m)))) ∧
    (~(nth n(cdr_bits o (memory t_m))(free t_m) = NUM_addr)) ∧
    !n'. (n' < n) ==>
     ~(nth n (cdr_bits o (memory t_m))(free t_m) =
       nth n'(cdr_bits o (memory t_m))(free t_m)))
```



**valid_program_constraint** The state of the machine must pattern match with the left side of the abstract machine transition for one of the 18 implemented machine instructions, or have a problem loaded in memory when the machine is directed to start computation. The constraint gives a quite detailed specification of the type and in some cases the value of each record in memory involved in the pattern match. A less detailed constraint, limiting the possible machine instructions and requiring the arguments to the LD instruction represent non-negative values, was sufficient for the proof of liveness of the machine. It was defined independently and proved to be implied by the more detailed constraint shown here.

This valid program constraint must be assured by the correctness of the compilation process. As with the *free_list_constraint* we restrict the constraint to times after the machine is initialised.

```
valid_program_constraint memory mpc button_pin s e c d =
 !t_m.
   (((state_abs(mpc t_m) = idle) ∧ button_pin t_m) ==>
     (is_cons(memory t_m NUM_addr)) ∧
     (is_cons(memory t_m(car_bits(memory t_m NUM_addr))))) ∧
   ((state_abs(mpc t_m) = top_of_cycle) ==>
     let head_c = memory t_m(c t_m)
     in
     ((is_cons head_c) ∧
      let instr' = memory t_m(car_bits head_c)
      and next_c = cdr_bits head_c
      in
      ((is_number instr') ∧
       let instr = atom_bits instr'
       in
        (  ...
         ((instr = ~LDC_instr28) ∧
          (is_cons(memory t_m next_c)) ∧
          (is_atom(memory t_m(car_bits(memory t_m next_c))))
         ) ∨
         ((instr = ~LDF_instr28) ∧
          (is_cons(memory t_m next_c))
         ) ∨
          ...
         ((instr = ~STOP_instr28) ∧
          (is_cons(memory t_m(s t_m))) ∧
          (next_c = NIL_addr))))))
```

## 6.2 Structure of the proof

The goal for this level of the proof is shown in Figure 5. Most of the signals are simply abstracted from the medium to the coarse time granularity. Additionally, the memory and state are abstracted to the appropriate data type as well. The validity of the temporal abstraction of the one input signal which is an argument to the top level specification (*i.e.* the *button_pin* input) should be carefully considered. We can show that this input affects the next state of the machine at the medium time granularity in only 3 places in the microcode, and these 3 places correspond to points in coarse grain time, since they are *major states*.

```
"((clock_constraint SYS_Clocked)                    ∧
  (reserved_words_constraint mpc memory)            ∧
  (well_formed_free_list mpc free memory)           ∧
  (valid_program_constraint memory mpc button-pin s e c d))   ==>
 (SYS memory SYS_Clocked
      mpc s0  s1  s2  s3
      button_pin
      flag0_pin   flag1_pin  write_bit_pin  rmem_pin
      bus_pins    mar_pins
      s  e  c  d  free
      x1  x2  y1  y2  car  root  parent
      buf1    buf2    arg)                        ==>
 SYS_spec  ((mem_abs o memory)  when (is_major_state mpc))
           (s                   when (is_major_state mpc))
           (e                   when (is_major_state mpc))
           (c                   when (is_major_state mpc))
           (d                   when (is_major_state mpc))
           (free                when (is_major_state mpc))
           (button_pin          when (is_major_state mpc))
           ((state_abs o mpc)   when (is_major_state mpc))"
```

Figure 5: RTL ⊃ top level goal

Although the lower (rtl) level definition of the system for this proof consists of the wiring together of high-level components defined behaviourally, it is not feasible to undertake the proof of the goal in Figure 5 directly. The size of terms generated alone makes this impossible to manage. The proof is instead undertaken in several stages.

First, a simplified, flattened specification for the register transfer level definition was obtained. Second, this specification was used to derive theorems for the



change of state effected by each of the 400 microcode instructions. These theorems represent the "Phase level" of description of Anceau [1]. Third, a sort of microcode level simulation used the 400 theorems to step through the microcode to produce theorems that summarized the behaviour of instruction sequences. This corresponds to the "microprogramming level" of Anceau. Fourth, these theorems were then used to prove a liveness property for the system. And last, we proved that the state transitions resulting from the register transfer level definition correspond to those defined at the top level. A description of each stage follows.

### 6.2.1 Unwinding the System Definition

The register transfer level of description consists of a conjunction of behavioural descriptions of major components. For example, the control unit consists of a *state register* (with 5 fields), a *ROM* and a *decode* unit. A specification for the conjunction of these three subcomponents is not appreciably simpler than the conjunction of the component behaviours individually. It is only when the whole system is assembled that simplifications become possible. A good example is the definition of the *ALU* subcomponent of the datapath. Its behavioural definition is constrained by a one-assertedness property on the control lines. This constraint is eliminated in the whole system because it is proved that the control unit assures this property of the control signals to the *ALU*.

Thus the normal approach, that of first defining a specification for each level of the hierarchy and then using each of these in turn to prove the next level of specification, was abandoned. Instead we simplified each level by UNWINDing existentially quantified variables where possible. This was carried out up to the top of the hierarchy where more substantial simplifications could be made. The process of creating these proofs was relatively straightforward, complicated only by the relatively large size of terms involved. However, simple rewrites could exhaust the Lisp namestack, so that the proof process had to be managed extremely closely (generally using quite low level tactics and rules).

Theorems for simplifying the control unit and datapath were undertaken first. These were achieved by rewriting with all subcomponent definitions, and UNWINDing and PRUNEing existentially quantified variables (hidden wires). The *one_asserted* constraint in the datapath was moved to the outermost level of the definition. Also, as all subcomponents have a quantified time parameter, the distinct quantified variables were replaced by a single quantified variable, encompassing the entire definition.

A theorem showing that the alu control outputs from the control unit have a *one_asserted* property was proved. The intensive computation required to prove inequality of two values of a specified wordn type made it desirable to

prove an exhaustive set of theorems for all possible values of each subfield of the microcode ROM output. Rather than repeatedly proving the cases for each of the 400 ROM addresses, this required 68 theorems all told. These theorems were of the form:

```
Alu_base_0001 =
⊢ (Alu_field(ROM_fun(mpc t_m)) = #0001) ==>
   ((Alu_field(ROM_fun(mpc t_m)) = #0001) = T) ∧
   ((Alu_field(ROM_fun(mpc t_m)) = #0010) = F) ∧
    ...
   ((Alu_field(ROM_fun(mpc t_m)) = #1011) = F) ∧
   ((Alu_field(ROM_fun(mpc t_m)) = #1100) = F)
```

In addition, theorems for the value of the Inc9 function, used to calculate the next mpc address in sequential microcode sequences, for each of 400 addresses were proved.

```
⊢ Inc9 #000000000 = #000000001
```

At the top of the hierarchy, the static RAM definition of memory was added in, and the CU and DP simplifications were applied to create one flattened expression for the SYS implementation. The *one_asserted* constraint fell out. All internal read and write control lines were eliminated. The only remaining existentially quantified variables, *bus_bits*, *mem_bits*, and the *alu*, were quantified over all conjuncts. The most important step involved moving the time parameter outside the existentially quantified internal lines, and replacing these time varying lines with static values. The equivalence of this transformation was proved, and specialised conversions designed. This key step permits evaluating the entire expression when constraining some value at a given time $t_m$; in fact we evaluated the simplified expression for the system description under the constraint $mpc\ t_m = x$, where $x$ is one of the 400 microcode addresses. From this stage onward, equivalence was no longer attempted, so the final form of the theorem has the clock_constraint and the *SYS* definition[6] as assumptions, and the conclusion is the simplified expression for the system, with the 3 existentially quantified variables. Also the conjunct giving original value of the mpc (*i.e.* at time $t_m = 0$) was dropped from the expression. Representative samples of each type of conjunct of the theorem are shown in Figure 6.

[6] We will abbreviate this constraint as *SYS_imp* in the following material.



```
Base_thm =
["clock_constraint SYS_Clocked"; ^SYS_imp]
⊢ ? bus_bits_t mem_bits_t alu_t.
      (mpc(SUC t_m) =
       (((Test_field(ROM_fun(mpc t_m)) = #0001)   ∨
         (Test_field(ROM_fun(mpc t_m)) = #0011) ∧
          field_bit bus_bits_t                    ∨
         ...
         (Test_field(ROM_fun(mpc t_m)) = #1011)) =>
        A_field(ROM_fun(mpc t_m)) |
        ((Test_field(ROM_fun(mpc t_m)) = #1100) =>
         s0 t_m |
         ((Test_field(ROM_fun(mpc t_m)) = #0010) =>
          Opcode arg t_m | Inc9(mpc t_m))))) ∧
       ...
      (memory(SUC t) =
       ((Write_field(ROM_fun(mpc t)) = #00001) =>
        Store14(mar_pins t)(bus_pins t)(memory t) |
        memory t)) ∧
      ((Read_field(ROM_fun(mpc t_m)) = #00010) ==>
       (bus_bits_t = mem_bits_t)) ∧
      ...
      ((Alu_field(ROM_fun(mpc t)) = #0001) ==>
       (alu_t = DEC28(atom_bits(arg t)))) ∧
      ...
      (~(Write_field(ROM_fun(mpc t_m)) = #00001) ∧
       (Read_field(ROM_fun(mpc t_m)) = #00010) ==>
       (bus_pins t_m = memory t_m(mar_pins t_m))) ∧
      ...
      (s t_m = ((Write_field(ROM_fun(mpc t_m)) = #00110) =>
                cdr_bits bus_bits_t | s(PRE t_m))) ∧
      ...
      (flag1_pin t_m = (mpc t_m = #000101011) ∨
                       (mpc t_m = #000011000))
```

Figure 6: Base_thm: the rtl definition simplified

### 6.2.2 Phase level: effect of each microinstruction

The simplified expression for the system was next utilised to produce theorems for the state of the system at each microcode address. A sample theorem for the system when the value #001100001 is in the *mpc* register is shown in figure 7. This is the first instruction of the microcode sequence for the LDF instruction,

and effects a transfer of the content of the *e* register to the *x2* register. The two dots represent the clock constraint and the system implementation assumptions. The large number (400) of such theorems required that they be generated with-

```
SYS_lemma_97 =
 .. ⊢ (mpc t = #001100001) ==>
       (mpc(SUC t) = #001100010) ∧
       (s0(SUC t) = s0 t) ∧
       (s1(SUC t) = s1 t) ∧
       (s2(SUC t) = s2 t) ∧
       (s3(SUC t) = s3 t) ∧
       (memory(SUC t) = memory t) ∧
       (x2 t = e(PRE t)) ∧
       (rmem_pin t = F) ∧
       (buf1 t = buf1(PRE t)) ∧
       (buf2 t = buf2(PRE t)) ∧
       (mar_pins t = mar_pins(PRE t)) ∧
       (s t = s(PRE t)) ∧
       (e t = e(PRE t)) ∧
       (c t = c(PRE t)) ∧
       (d t = d(PRE t)) ∧
       (free t = free(PRE t)) ∧
       (x1 t = x1(PRE t)) ∧
       (car t = car(PRE t)) ∧
       (arg t = arg(PRE t)) ∧
       (parent t = parent(PRE t)) ∧
       (root t = root(PRE t)) ∧
       (y1 t = y1(PRE t)) ∧
       (y2 t = y2(PRE t)) ∧
       (write_bit_pin t = T) ∧
       (flag0_pin t = F) ∧
       (flag1_pin t = F)
```

Figure 7: Theorem for microcode instruction at address 97

out user intervention, and as efficiently as possible. A single proof function was designed for this purpose, and while efficiency was important in its design, the required generality was a limiting factor. Generating the set of theorems took approximately 36 hours running on a dedicated Sun 3/60 workstation with 16 megabyte memory. The theorems were generated in groups that corresponded to sets of SECD instruction code sequences or subroutines, to save search times for theorems at the next proof level.

The proof function works as follows:



1. Fetch the appropriate microcode lemma for the given address, and specialise it with the specific mpc value. This lemma has the form:
   . ⊢ ROM_fun(mpc $t_m$) = #000000000000000000110001000
   The dot to the left of the turnstile symbol represents the assumption *mpc* $t_m$ = *#001100001*, in this and all the following theorems.

2. Substitute for the 27 bit constant with the 27 bit Bus, then for each field, apply the appropriate field selector function, rewrite with the definition of the function, and return the Bus form of the field value to the wordn constant form. This theorem is then resolved with the CU_proofs theorem for that field value of constant, and split into conjuncts, giving a set of theorems each having the form:
   . ⊢ (Test_field(ROM_fun(mpc $t_m$)) = #0101) = F
   This is done for each required value for the Test, Read, Write, and Alu fields of the 27 bit value.

3. The relevant Inc9 theorem is retrieved from Inc9_proofs and specialised:
   . ⊢ Inc9(mpc $t_m$) = #001100010.

4. An Afield theorem is obtained similarly to the other field theorems:
   . ⊢ A_field(ROM_fun(mpc $t_m$)) = #000000000.

5. The flag output expressions are evaluated with the mpc value substituted in, to generate a theorem of the form:
   . ⊢ (mpc $t_m$ = #000010110) ∨(mpc $t_m$ = #000011000) = F

6. The set of all the proceeding theorems is assembled as a list, and substituted into the Base_thm, using a template, and a series of rewrites reduces constant boolean expressions and removes eliminated existentially quantified variables, followed by a pruning of any remaining existentially quantified variables.

7. If any existentially quantified variables remain, they are eliminated by a specialised rule that uses theorems about the existence of fields of an existentially quantified variable:
   car_bits_thm = ⊢ ! y. ? x. car_bits x = y
   cdr_bits_thm = ⊢ ! y. ? x. cdr_bits x = y

### 6.2.3 Microprogramming level: symbolic execution

Just as every one of the 400 lemmas about the rtl description give us the next state in terms of the present state, we next want to combine these results to give the state after a number of steps in terms of the present state.

The definition for *SYS_spec* describes a 4 state fsm, with 2 transitions from each of 3 states controlled by the button input, and 18 possible transitions, one

for each machine instruction code, from the 4th state. Further, 4 instruction sequences have a branch conditional upon some function of the state, and one (the sequence for the *LD* instruction) contains 2 loops. Thus there are minimally 28 paths to consider. Additionally, several instruction sequences call subroutines, and subroutine calls are nested. Under the *free_list_constraint*, the subroutines have a deterministic execution time.

The proof needed to show the state of the machine when it next reached a major state. This was easily achieved by a combination of *MATCH_MP*[7] with the conjunct giving the value of *mpc(SUC $t_m$)*, and rewriting with all the other clauses of the *SYS_lemma_** theorems. Additionally, it was necessary to prove whether the system was in a major state at each step. A proof function was designed to produce such a pair of theorems, and a higher level function could call it recursively to produce a pair of theorems covering a sequence of steps. The subroutine correctness proofs were produced using these functions, plus another higher level function that could use a preproved segment proof pair (required when a subroutine call was involved).

The sequence for a machine execution, in this case the *LDF* instruction, is summarized in the two theorems in Figure 8. There are 5 assumptions, including:

1. "clock_constraint SYS_Clocked"
2. SYS_imp
3. "free_list_constraint mpc free"
4. "mpc t = #00010101"
5. "opcode_bits(memory t(car_bits(memory t(c t)))) = #000000011"

The rather complex expression for the state of the machine becomes more comprehensible when only the values abstracted to the top level are considered. The *mpc* content corresponds to the *top_of_cycle* state. The *memory* has had 2 cells altered: the first cell has pointers to the code argument to the *LDF* instruction, and the current environment *e*. The second cell has pointers to the previous one, and the original stack *s*. The *s* register is updated to point to the latter cell. The *c* register now points to the rest of the control list after the *LDF* instruction and its argument. Notice that the *cdr* operation is performed on different memories: the first on the original memory, and the second after the memory is updated with the 2 rewritten cells. The *free* register has similarly been updated to the the 3rd cell in the original free list. The content of the other registers is either unchanged or irrelevant to the abstracted state. The expression for the updated memory appears several times, so has been extracted to simplify reading.

[7] *MATCH_MP* is a HOL variant of Modus Ponens.



```
LDF_state =
..... ⊢  (mpc(t_m+26) = #0001010111) ∧
         (s0(t_m+26) = s0 t_m) ∧ (s1(t_m+26) = s1 t_m) ∧
         (s2(t_m+26) = s2 t_m) ∧ (s3(t_m+26) = #000000000) ∧
         (memory(t_m+26) = ^memory_exp) ∧
         (mar_pins(t_m+26) =
          cdr_bits(^memory_exp(cdr_bits(^memory_exp(c t_m))))) ∧
         (rmem_pin(t_m+26) = F) ∧
         (buf1(t_m+26) = buf1 t_m) ∧
         (buf2(t_m+26) = buf2 t_m) ∧
         (s(t_m+26) = cdr_bits(memory t_m(free t_m))) ∧
         (e(t_m+26) = e t_m) ∧
         (c(t_m+26) =
          cdr_bits(^memory_exp(cdr_bits(^memory_exp(c t_m))))) ∧
         (d(t_m+26) = d t_m) ∧
         (free(t_m+26) =
          cdr_bits(^memory_exp(cdr_bits(memory t_m(free t_m))))) ∧
         (x1(t_m+26) = free t_m) ∧ (x2(t_m+26) = s t_m) ∧
         (car(t_m+26) =
          car_bits(memory t_m(cdr_bits(memory t_m(c t_m))))) ∧
         (arg(t_m+26) = memory t_m(car_bits(memory t_m(c t_m)))) ∧
         (parent(t_m+26) = parent t_m) ∧ (root(t_m+26) = root t_m) ∧
         (y1(t_m+26) = y1 t_m) ∧ (y2(t_m+26) = y2 t_m) ∧
         (write_bit_pin(t_m+26) = T) ∧
         (flag0_pin(t_m+26) = F) ∧ (flag1_pin(t_m+26) = T)

where memory_exp =
 "Store14
   (cdr_bits(memory t_m(free t_m)))
   (bus32_cons_append #00 RT_CONS(free t_m)(s t_m))
   (Store14
    (free t_m)
    (bus32_cons_append
       #00 RT_CONS
       (car_bits(memory t_m(cdr_bits(memory t_m(c t_m)))))
       (e t_m))
    (memory t_m))"

LDF_Next = ..... ⊢ Next t_m(t_m+26)(is_major_state mpc)
```

Figure 8: Microprogramming level theorems for LDF instruction

### 6.2.4 Liveness

If the SECD system initially reaches a major state, and every time it is in a major state all possible paths eventually return it to a major state, then the temporal abstraction function from the coarse granularity of time to the medium granularity is total. This important property is essential for the last part of the proof.

```
⊢ Next t1 t2 f = (t1 < t2) ∧ (f t2)  ∧
                 ! t.(t1 < t) ∧ (t < t2) ==> ~ f t

⊢ (IsTimeOf      0  f t = f t ∧ !t'.(t'<t) ==> ~ f t') ∧
  (IsTimeOf (SUC n) f t = ? t'. IsTimeOf n f t' ∧ Next t' t f)

⊢ TimeOf f n = @t.IsTimeOf n f t

⊢ when (s:num->*) (p:num->bool) = λn.s (TimeOf p n)

⊢ Inf f = ! t.? t'.(t<t') ∧ (f t')
```

Figure 9: Temporal Abstraction Function Definitions

The *state_abs* function is well-defined only when one of 4 values is in the *mpc* register, and these are precisely the 4 values for which the temporal abstraction predicate *is_major_state mpc* is true. Unfortunately, one cannot prove that the abstraction predicate holds at the time abstracted to by the predicate, *i.e. (is_major_state mpc) when (is_major_state mpc)*$|_{t_c}$, or in general *(P when P') t* (which is definitionally equivalent to *P(TimeOf P' t)*). This limitation arises from the use of the Select operator *@* in defining the temporal abstraction function TimeOf[8], given above. We overcome this obstacle by by proving a "liveness" property for the predicate used for the abstraction (*is_major_state mpc*). Liveness as defined by *Inf* states that the predicate is true infinitely often. The proof of this property is simplified by using the theorem *Inf_thm*:

```
⊢ ! f.(? t'm. 0 < t'm ∧ f t'm) ∧
      (! tm. f tm ==> (? t'm. tm < t'm ∧ f t'm)) ==> Inf f
```

[8]From the defining axiom for the Select operator SELECT_AX: ⊢∀(P:* -> bool) (x:*). P x ==> P($@ P), it is necessary to show that P holds at some value x in order to derive that P($@ P) holds. See [18] for further description.



which limits the proof requirement to showing that a major state is reached initially, and every time the machine is in a major state it will reach a major state again in the future. This limits the number of starting points to four, instead of every possible machine state (consisting of 400+ values that can be in the *mpc*). The microprogramming level theorems defining the *Next* times the temporal abstraction function holds are used for each branch of the proof.

### 6.2.5 Relating the Levels over Abstraction

The top goal splits into two parts: the state of the mpc when the machine first reaches a major state, and the state of the machine when it is next in a major state, in terms of its state in the previous major state (*i.e.* at time $t_c$). The first goal is quite simply solved by using SYS_lemma_0 to show that at time $SUC\ 0_m$ we reach a major state, and that at all times before that (namely time $t_m = 0$), the system is not in a major state. The required theorem has the form:

```
⊢ clock_constraint SYS_Clocked ==>
  ^SYS_imp  ==>
  (((state_abs o mpc) when (is_major_state mpc))0c = idle)
```

```
car_cdr_mem_abs_lemma =
⊢ is_cons(memory v) ==>
  !x. (M_Car(v,mem_abs memory, x) = car_bits (memory v)) ∧
      (M_Cdr(v,mem_abs memory, x) = cdr_bits (memory v))

number_mem_abs_lemma =
⊢ is_number(memory v) ==>
  !x. M_int_of(v,mem_abs memory, x) =
      iVal(Bits28(atom_bits (memory v)))

opcode_mem_abs_lemma =
⊢ is_cons(memory v) ==>
  is_number(memory(car_bits(memory v))) ==>
  !x. M_int_of(M_CAR(v,mem_abs memory,x)) =
      iVal(Bits28(atom_bits(memory(car_bits(memory v)))))

cons_mem_abs_lemma =
⊢ is_cons (memory free) ==>
  !v w.
   M_Cons(v,w,mem_abs memory,free) =
   (free,
    mem_abs(Store14 free (bus32_cons_append #00 RT_CONS v w) memory),
    cdr_bits (memory free))

Store14_cons_mem_abs_lemma =
⊢ (~(free = (cdr_bits(memory free))))       ==>
  is_cons (memory (cdr_bits (memory free))) ==>
  !v w z.
   M_Cons(v,w,
          mem_abs(Store14 free z memory),
  cdr_bits(memory free)) =
   (cdr_bits(memory free),
    mem_abs(Store14 (cdr_bits(memory free))
                    (bus32_cons_append #00 RT_CONS v w)
                    (Store14 free z memory)),
    cdr_bits (Store14 free z memory (cdr_bits (memory free))))
```

Figure 10: Abstract Memory Theorems



We may split the second goal into a set of subgoals, corresponding to each transition defined in the SYS_spec. The transitions are determined by the state at time $t_c$, which is a function of the *mpc* at *TimeOf(is_major_state mpc)*$t_c$. Using the liveness property described previously, with the theorem TimeOf_TRUE:

$$\text{TimeOf\_TRUE} = \vdash \forall f.\ Inf\ f\ ==>\ (\forall n.\ f\ (TimeOf\ f\ n)),$$

to obtain the result that *is_major_state mpc* is true at all points of the coarser granularity of time. With this result, the state abstraction function is well defined, and the *mpc* has one of four values at all points in the coarser granularity of time.

The specification for each transition defines the new values for each of the *s*, *e*, *c*, *d*, and *free* registers and the abstract memory in terms of abstract memory operations on the previous values of the registers and abstract memory. A set of theorems relating abstract memory operations to rtl level operations is given in Figure 10. The theorem *opcode_mem_abs_lemma* combines the two previous theorem results for the machine instruction code value evaluation. The last theorem, *Store14_cons_mem_abs_lemma*, is used when two *M_Cons* operations are performed on a memory.

Theorems for the correctness of each top level transition are typified by the LDF transition theorem in Figure 11. The last 4 assumptions in the theorem match the applicable portion of *valid_program_constraint* for the LDF instruction branch. The major steps in proving this theorem include:

- deriving the value of the 9 bits of the 28 bit instruction code from the bottom assumption, for resolving with the less specific assumption of *LDF_state* and *LDF_Next*,
- rewriting the right side of the goal with the definition of *NEXT*,
- deriving the integer value of the 28 bit instruction code from the bottom assumption, and proving inequality of this value with all lesser values in order to reduce the right side of the goal to the *LDF_trans* branch,
- using *LDF_Next* to prove that *(TimeOf(is_major_state mpc)(t+1)) = (TimeOf(is_major_state mpc)t)+26*
- rewriting the left side of the goal with *LDF_state* and the right side with the definition of *LDF_trans*
- using the abstract memory theorems of Figure 10 resolved with the assumptions about record types and the *free_list_constraint* to rewrite the right side of the goal, producing expressions for the next state in the same terminology as the left side.

```
[ SYS_imp
; "clock_constraint SYS_Clocked"
; "reserved_words_constraint mpc memory"
; "well_formed_free_list mpc free memory"
; "mpc(TimeOf(is_major_state mpc)t) = #000101011"
; "is_cons(memory(TimeOf(is_major_state mpc)t)
          (c'(TimeOf(is_major_state mpc)t)))"
; "is_number(memory(TimeOf(is_major_state mpc)t)
            (car_bits(memory(TimeOf(is_major_state mpc)t)
                     (c'(TimeOf(is_major_state mpc)t)))))"
; "is_cons(memory(TimeOf(is_major_state mpc)t)
          (cdr_bits(memory(TimeOf(is_major_state mpc)t)
                   (c'(TimeOf(is_major_state mpc)t)))))"
; "atom_bits(memory(TimeOf(is_major_state mpc)t)
            (car_bits(memory(TimeOf(is_major_state mpc)t)
                     (c'(TimeOf(is_major_state mpc)t))))) =
  #00000000000000000000000000011" ]

 ⊢ s(TimeOf(is_major_state mpc)(SUC t)),
   e(TimeOf(is_major_state mpc)(SUC t)),
   c(TimeOf(is_major_state mpc)(SUC t)),
   d(TimeOf(is_major_state mpc)(SUC t)),
   free(TimeOf(is_major_state mpc)(SUC t)),
   mem_abs(memory(TimeOf(is_major_state mpc)(SUC t))),
   state_abs(mpc(TimeOf(is_major_state mpc)(SUC t))) =
   NEXT
   (s(TimeOf(is_major_state mpc)t),
    e(TimeOf(is_major_state mpc)t),
    c(TimeOf(is_major_state mpc)t),
    d(TimeOf(is_major_state mpc)t),
    free(TimeOf(is_major_state mpc)t),
    mem_abs(memory(TimeOf(is_major_state mpc)t))
```

Figure 11: The Correctness result for the LDF instruction



- rewriting with the triplet selector function definitions (*cell_of*, *mem_of*, *free_of*, *mem_free_of*, etc.) used in the definition of LDF_trans.

Splitting the top goal (Figure5) on the state, and for *top_of_cycle* state, on the instruction codes permitted by the *valid_program_constraint*, reduces the problem to cases solved by similar theorems for each top level transition.

# 7 What Has Been Proved

If the goal of verifying hardware designs is to provide assurance that the designed (as opposed to fabricated) circuit will provide a defined functionality, the significance of the SECD verification effort must be judged in this overall context. The proof described has relates the register transfer level definition to the top level specification. A proof relating the low and register transfer levels is also being undertaken, but was not described herein. Taken together, these proofs only partially achieve the goal.

The top level specification lacks a definition for garbage collection. The proof has been constrained to cases where this is not required, but the issue of the bound in computation imposed by limited memory size has not been addressed. More generally, we have not given any assurance that the top level specification is consistent with the abstract SECD definition we started with.

The constraints (other than the *free_list_constraint*) represent reasonable limitations on the operation of the system. The *valid_program_constraint* will in the ideal case be guaranteed by a proven compilation process. As a start in this direction, a hand proof of the correctness of the compilation process of a high level language to SECD code has been completed. The rtl model (as also the low level model) is an abstraction of the lower level circuit. Although we believe that the model is correct, this belief is not supported (neither is it contradicted!) by this work.

The contributions made by this work include the use of a multi-bit data type in a large proof, the specification of a complex state relation using an abstract memory data type to represent complex S-expressions, and the achievement of a large proof result in the HOL system. Through the course of this project, we have learned that for a trained HOL user, designing proofs in the system, while at times exceedingly tedious, was not in itself that difficult. Managing the proof: the size of expressions, complexity of transitions, and number of cases, required great care. Most challenging, however, was deriving the logical representation of the machine and its specification, and formally defining the constraints.

## Acknowledgements

We wish to acknowledge the insight into representing LISP data structures gained from conversations with Ian Mason ([19] and [20]). Inder Dhingra and Tom Melham gave us good advice and much help in discussions on the top level specification. We gratefully acknowledge the efforts of the SECD chip building team: Mark Brinsmead, Jeff Joyce (who began the development of the SECD in early 1985), Mary Keefe, Wallace Kroeker (Alberta Microelectronic Centre), Breen Liblong (Alberta Microelectronic Centre), Rick Schediwy, Konrad Slind, Glen Stone, Walter Vollmerhaus, Mark Williams, and Simon Williams.

The research described above was supported by Strategic, Operating, and Equipment Grants from the Natural Sciences and Engineering Research Council of Canada and The Canadian Microelectronics Corporation. The Strategic Grant was also supported by The Alberta Microelectronic Centre and LSI Canada Inc. The SECD verification effort was also supported by The Communication Research Establishment, Ottawa. We are also grateful to MOSIS (who built two versions of the chip) for their speedy, efficient and courteous service.

## References

[1] François Anceau. *The Architecture of Microprocessors.* Addison-Wesley Publishing Company, 1986.

[2] G. Birtwistle, B. Graham, J. Joyce, S. Williams, M. Brinsmead, M. Keefe, W. Kroeker, B. Liblong, and W. Vollmerhaus. The SECD Machine on a Chip. In *Int. Conf. on CAD and CG*, Beijing, 1989.

[3] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] A. J. Cohn. A Proof of Correctness of the VIPER Microprocessors: The First Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Norwell, Massachusetts, 1988. Kluwer.

[5] A. J. Cohn. A Proof of Correctness of the VIPER Microprocessors: The Second Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91, New York, 1989. Springer Verlag.

[6] A. J. Cohn and M. J. C. Gordon. A Mechanized Proof of Correctness of a Simple Counter. Technical Report 94, Computing Laboratory, University of Cambridge, 1986.



[7] I. S. Dhingra. Formal Validation of an Integrated Circuit Design Style. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321, Norwell, Massachusetts, 1988. Kluwer.

[8] I. S. Dhingra. Formal Validation of an Integrated Circuit Design Style. PhD thesis, University of Cambridge Computer Laboratory, 1988.

[9] P. Gabriel. *Performance and Evaluation of LISP Systems.* MIT Press, Boston, 1985.

[10] M. J. C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, Computing Laboratory, University of Cambridge, 1985.

[11] B. Graham and G. Birtwistle. Formalising the Design of an SECD chip. In *Proceedings of the Cornell Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, New York, 1989. Springer Verlag.

[12] B. Graham, S. Williams, G. Birtwistle, J. Joyce, and B. Liblong. The Mossim Specification of the SECD DESIGN. Research report 89/341/03, Computer Science Department, University of Calgary, 1989.

[13] B. Graham, S. Williams, and G. Stone. Operating Specification for the SECD Chip. Research report 89/353/15, Computer Science Department, University of Calgary, 1989.

[14] Brian Graham. SECD: Design Issues. Research Report 89/369/31, Computer Science Department, University of Calgary, 1989.

[15] Brian Graham. Dealing with the Choice Operator in HOL88. Research Report 90/382/06, Computer Science Department, University of Calgary, 1990.

[16] P. Henderson. *Functional programming; applications and implementation.* Prentice Hall, London, 1980.

[17] J. Joyce. The SECD Machine, A Study in Advanced Architectures. Unpublished report of CPSC 603 course work at the University of Calgary.

[18] J. Joyce. Formal Verification and Implementation of a Microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Norwell, Massachusetts, 1988. Kluwer.

[19] I. A. Mason. *The Semantics of Destructive Lisp.* Center for the Study of Languange and Information, Stanford, 1986.

[20] I. A. Mason. Verification of Programs that Destructively Manipulate Data. *Science of Computer Programming*, 10:177–210, 1988.

[21] T. F. Melham. Abstraction Mechanisms for Hardware Verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Norwell, Massachusetts, 1988. Kluwer.

[22] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[23] T. Simpson, G. Birtwistle, B. Graham, and M. J. Hermann. A Compiler for LispKit Targetted at Henderson's SECD Machine. Research Report 89/339/01, Computer Science Department, University of Calgary, 1989.

[24] T. Simpson, B. Graham, and G. Birtwistle. From Lispkit Code to SECD Chip: Some Steps on the Way to a Verified System. In *Proceedings of the Third Banff Higher Order Workshop*, 1989. submitted for publication.

[25] The HOL System. Tutorial. Technical Report, SRI International Cambridge Research Center, under contract to DSTO Australia, Cambridge, England, 1989.

[26] M. Williams. SECD Controller Board Implementation. Technical Report #89/359/21, Computer Science Department, University of Calgary, Calgary, 1989.