1 Introduction

The SECD chip was designed as the focus of a study of specification-driven VLSI design. It is a physical realization of Landin's abstract architecture ([Lan64]), as described in [Hen80]. A detailed, hierarchical definition of the actual SECD chip written in CDL (an interface language for an implementation of the Mossim Simulator by R. Bryant [Bry80]) can be found in [GWB+89], while a description of the abstract architecture can be found in [HBGS89]. An informal operating specification of the chip is described in [GWS89].

This document discusses issues that arose in the specification and design of the chip. We begin with a description of the wider project in which it was conceived, and follow with a listing of the various levels at which the chip was considered. Each of these levels and the design issues associated with it is presented in turn. Three levels of formal specification are used to describe the chip in HOL. The representation of time, clocks, and data types, as well as the abstraction mechanisms to relate different levels of these, are described. We conclude with some comments about the interaction between verification and design.

1.1 Project Context

The SECD chip arose within an ongoing research effort by the VLSI 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 process is the object of study, and the chip is a byproduct only of that process, providing the team with hands-on experience with a nontrivial design. This intention gave rise to design criteria 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 (Higher Order Logic — see [Gor85]) 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, with an available compiler for a pure functional Lisp language subset (Henderson's Lispkit language). We begin with the assumption that the compiler is or can be shown to be correct, and that SECD executing compiled LispKit programs correctly implements the high level language. A hand proof of the latter, along the lines of [Plo75], has been completed [SBG89] and will be formalised in HOL in the future.

The project context described gave rise to design criteria that affected decisions taken throughout the design of the SECD chip.

- The most important criterion was that a *correct, working device* be produced. Correctness is the primary focus of the specification-driven process. On the other hand, an ostensibly "correct" design for a device that fails to work, raises questions about the methodology and assumptions.

- The next criterion was *simplicity*. Simplicity was necessary on two counts: to ensure that verification could cope with what promised to be the most complex microprocessor verification attempted to date, and secondly, to improve the likelihood of meeting the first criterion.
- *Testability* of the design was considered essential. In the event of malfunction, determination of the source of the error requires examining the state of the machine extensively. Furthermore, reliance on a correct output to test problems does little to assure design correctness. Rather, each step of the computation should be accessible for checking.
- Lastly, *utility* should be considered. It was preferable that the design could be given meaningfully sized tasks, rather than be considered a toy device, incapable of all but the most trivial tasks. A task particularly relevant to this design would be compiling LispKit programs, for running on the system.

Equal in importance to the selection of criteria is the explicit statement of items that will not be given priority. *Speed* was specifically eliminated as a determining criterion, both in terms of clocking rate, and optimality of the operation sequences, insofar as they could conflict with the simplicity criterion. The design philosophy that emerged from these constraints leads to a design that could be characterized as "dumb, but correct".

1.2 Levels of the Design

Throughout the design process, the chip and system are viewed at many different levels in a hierarchical fashion that sees increasing level of detail as we proceed. We now proceed to describe the major levels that were considered.

Abstract machine is the high level definition of the machine characterized by the contents of 4 stacks. The machine is defined in [Hen80] by giving a state transition for each machine instruction, in terms of the contents of the stacks.

Abstract System level views the SECD machine as a batch mode co-processor to another processor (SUN workstation has been chosen). A simulation for this level is characterized by the use of high level programming constructs, and introduces 'read' and 'print' routines to represent the workstation task of uploading and downloading problems to the SECD memory.

Top level FSM (finite state machine) describes the control of the system in terms of major states and transitions. These states are *Idle, Error1, Error2,* and *Top of Cycle*, introducing the ideas of initialization and completion of task, and error conditions. The transitions given for the abstract machine are the set of cycles from the *Top of Cycle* state.

Abstract RTL (register transfer level) view includes all the required registers and combinational logic devices in the datapath and a memory, and a simulation controller implemented using simple control mechanisms such as 'while', 'case', and 'if ... then ... else'. Records are represented as 32 bit words, and the bit assignments are determined. In-place, non-recursive garbage collection is introduced. Higher level routines represent bus transfer functions and combinational logic devices to build cons records and extract car and cdr fields from them.

Concrete RTL develops a full microcode from the simulation model used in the Abstract register transfer level. Subroutine implementation details are included, as well as the implementation of flow control. The controller uses a microcode sequence assembled to a binary image for the microcode ROM. The controller can be viewed at this level as the full finite state machine for the SECD chip.

Mossim defines the design down to the transistor level using CDL. It is used for simulation of the design components.

Layout maintains a (nearly) one-to-one correspondence with the Mossim model. The layout was defined using the Electric layout software. The design is hierarchically structured, and different concerns arise at different levels in the hierarchy.

Several programs from Gabriel and Henderson were run on all simulation models below the Mossim level. The largest test was the compilation of the LispKit compiler from Henderson. A LispKit program with 3 mutually recursive functions was used to test the Mossim models of the controlunit and datapath.

Three levels are formalised in HOL definitions, corresponding roughly to the mossim/layout, concrete register transfer, and the abstract system levels. The lowest level defines the chip in terms of primitive gates and transistors, with a granularity of time that captures the clock signal characteristics, and data types based on boolean valued signals. The RTL definition uses the clock cycle granularity of time, and more complex data types. The system level defines the behaviour of the system in terms of the state of memory and register contents on the chip. It uses an abstract time corresponding to machine instruction execution cycles, and defines an abstract memory type with high level operations such as *cons*, *car*, and *cdr*. The HOL definitions may be considered formalized versions of the earlier description levels, which interact with them, constraining the design in a variety of ways.

2 The Abstract Machine & System Levels

The top level definition of the SECD machine given in [Hen80] defines a set of machine transitions, one for each of the 21 SECD machine instructions, in terms of contents of the 4 stacks. This concise specification hardly begins to define how a machine would operate, but once the data representation is fixed, it can be used to determine the data structure transformations required for each instruction execution.

INITIAL STATE					TRANSFORMED STATE			
S	E	C	D		S	E	C	D
s	e	(LD (m.n).c)	d	→	(x.s)	e	c	d
					where x = locate ((m.n),e)			
s	e	(LDC x.c)	d	→	(x.s)	e	c	d
(a b.s)	e	(OP.c)	d	→	(b op a.s)	e	c	d
where (OP,op) ∈ {(EQ,=),(LEQ,≤),(ADD,+),(SUB,−),(MUL,×),(DIV,/),(REM,rem) }								
((a.b).s)	e	(CAR.c)	d	→	(a.s)	e	c	d
((a.b).s)	e	(CDR.c)	d	→	(b.s)	e	c	d
(a.s)	e	(ATOM.c)	d	→	(t.s)	e	c	d
					where t = (a is an atom)			
(a b.s)	e	(CONS.c)	d	→	((a.b).s)	e	c	d
(x.s)	e	(SEL ct cf.c)	d	→	s	e	cx	(c.d)
					where cx = if (x=T) then ct else cf			
s	e	(JOIN)	(c.d)	→	s	e	c	d
s	e	(LDF c'.c)	d	→	((c'.e).s)	e	c	d
((c'.e')v.s)	e	(AP.c)	d	→	NIL	(v.e')	c'	(s e c.d)
(x)	e'	(RTN)	(s e c.d)	→	(x.s)	e	c	d
s	e	(DUM .c)	d	→	s	(Ω.e)	c	d
((c'.e')v.s)	(Ω.e)	(RAP.c)	d	→	NIL	rplaca(e',v)	c'	(s e c.d)
s	e	(STOP)	d	→	s	e	(STOP)	d

Table 1: Abstract Machine Level Transitions

This level was modelled by a interpreter written in Franz Lisp ([HBGS89]), which, together with a LispKit compiler ([SBGH89]), was used to gain a better understanding of the way high level constructs in the source language (LispKit) were implemented by the compiled machine instructions. Error checking on operation arguments was performed, and run-time statistics on instruction counts and environment accesses were recorded. This stage was simply a learning exercise, and did not seek to flesh out the implementation design. Thus, the interpreter did not attempt to define data structures or elemental machine operations, but rather relied entirely upon the Franz Lisp data structure (S-expressions or lists) representation, as well as the *cons*, *car*, and *cdr* operations and the 5 arithmetic operations that SECD uses. Furthermore, fundamental implementation concerns were ignored, with recursive functions used to locate values in the environment list, and resource management (*i.e.* garbage collection of records) completely omitted.

2.1 Abstract System: the First Refinement

Realizing the SECD machine as a working system required that it be able to accept a task, compute a result, and return it. The abstract machine definition required that problems be in the form of a function to be applied to a list of arguments. 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[2].

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 does not permit coding of an infinite 'while' loop the execution of which will not exhaust system resources. Recursion is the only means to achieve an infinite sequence, and each recursion uses up some 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 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) is not determined at this level, but it is known that they will be stored within a finite memory, and this necessitates garbage collection. A simple *mark and sweep* garbage collector is used. Thus, the SECD system will consist 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 simulation at this level (written in "C") used external *read* and *print* routines to model the 'load problem' and 'return result' tasks of the system. These functions prepared a memory image within a finite memory data structure and extracted an S-expression from a memory image respectively. The SECD system was modelled making free use of high level language constructs including complex data types to represent data records, and recursive tree traversal in the garbage collector *mark* routine.

To prevent overwriting, we established a precedence (a partial ordering) on registers for each machine instruction when designing a control sequence for the simulation. Consider the transition for the AP instruction as an example of the generation of a microcode sequence. The abstract machine transition is given by:

$$((c'.e')v.s) \quad e \quad (AP.c) \quad d \quad \rightarrow \quad NIL \quad (v.e') \quad c' \quad (s\ e\ c.d)$$

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

We observe the following precedence on registers:

$$d \prec \begin{matrix} e \\ c \end{matrix} \prec s$$

. The control sequence then takes the following form.

```
d = (cons (cdr (cdr s))
          (cons e
                (cons (cdr c) d)))
c = (car (car s))
e = (cons (car (cdr s))
          (cdr (car s)))
s = NIL
```

The introduction of garbage collection introduces the "memory exhausted" error into this level simulation. Further errors, including arithmetic over/underflow, validity of operands for machine operations, and correct program compilation, remain possible, but are not incorporated yet.

3 The Top FSM Level

The previous abstract view of the SECD system was concerned primarily with manipulating the S-expressions in the 4 stacks. The finite state machine view concerns the development of a controller for driving the manipulations. The top level FSM has only four states, and the earlier view of the machine's state as being represented by the contents of the 4 main stacks (S, E, C, and D) is incorporated as annotations in transitions where these stack contents change (see Figure 1).

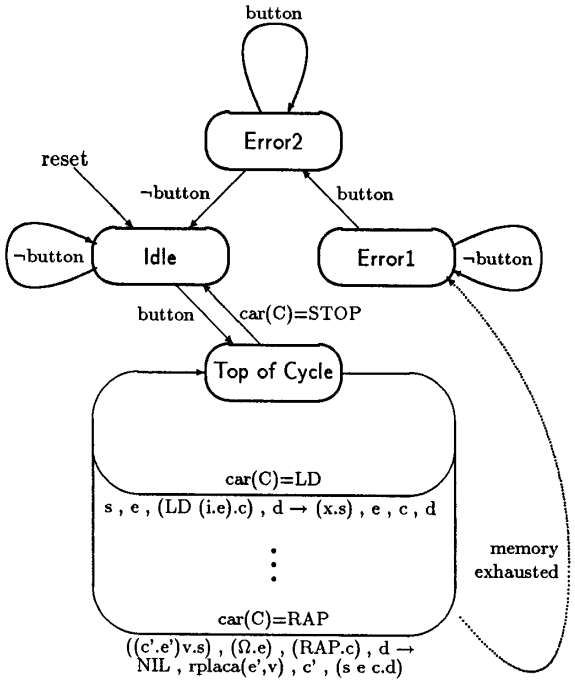


Figure 1: Top Level Finite State Machine View of SECD

This view introduces several ideas concerning the operation of the machine.

- An external *reset* input has been included to permit a deterministic startup of the machine. In simulations to this point, we have always begun with the same sequence of operations, and modelling the controller as a finite state machine similarly requires selection of the startup state. The assumption of this initial state is most simply implemented by the *reset* input, along with a constraint that the reset is asserted in the initial clock cycle.
- An *idle* state has been introduced, permitting the chip to be turned on but not computing until a problem has been loaded, and also permitting the completion of the computation to be recognizable. Earlier models saw the machine performing single computations only. This level introduces the idea of repeated executions.
- A *Top of Cycle* state corresponds to the state the machine is in at the start of execution of any SECD machine instruction. There are 21 transitions leading from this state, one for each machine instruction, forming an instruction fetch/execute style of loop. The abstract machine transition for the STOP instruction is clarified by transferring to the idle state, instead of looping infinitely.

- Two *Error* states are shown. The diagram shows an error condition arising from the exhaustion of memory only. This is a necessary error condition, consistent with the previous level view. Further errors are not indicated, actually reflecting decisions made at a lower level.
- The introduction of *control state* suggests the existence of state values as outputs, particularly as a means of signalling the completion of a task.
- An external input labelled *button* controls the state transitions from the *Error* and *Idle* states. The use of separate inputs could have eliminated the need for the second error state at this level[3], but a concern over the number of pins available for inputs mandated the single signal.
- There is no data input or output facility indicated. The machine will operate in a sort of "batch" mode on a problem in memory. It is also able to perform successive computations.

This level is not independent of the abstract system or abstract register transfer views of the SECD. Instead, it is a particular view of the system that is useful in formalising the behaviour of the system. Nor is a simulation model directly related to this level. Instead, we see the next level simulation incorporating the notion of major state by adding a state value at suitable points in the control sequence.

[3]Requiring distinct values on incoming transitions than those for outgoing transitions from the state make the signal timing less critical.

4 The Abstract Register Transfer Level

The actual architecture of the SECD chip developed with the definition of the Register Transfer Level view. The SECD machine at this stage is seen as a set of registers, combinational logic units, a memory, and a bus linking the components. The state transformations are effected by shifting values between registers and memory, using combinational logic to perform such functions as "cons", "car", and "cdr". The sequence of operations required is the model for the controller, which at this stage still retains some higher level control structures such as "if ... then ... else", "case", and "while".

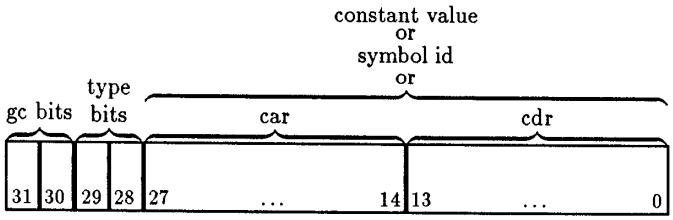
The S-expression data type, which is the "stuff" of SECD machine programs and data, is composed of three types of objects: numbers, symbols, and cons records. A simple mark and sweep garbage collector required the use of 2 bits in each record. Two additional bits indicate the "type" each record contains.

Numbers were permitted to range over integers (rather than the natural numbers), consistent with the definition of SECD.

Symbols represent atomic values which can only be tested for equality with each other. Thus, a distinct symbol identification number is a suitable representation. The "meaning" of the symbol (or its written form) is of concern only on input and output operations, and hence assignment and interpretation can be handled entirely outside of the SECD chip (by the compiler, since new symbols cannot be created in the course of executing programs). Three symbolic constants (Nil, True, and False) are required by the SECD chip, and were "built-in" at this stage.

Cons records represent a pairing operation of S-expressions. In typical Lisp fashion, these are implemented by pairs of pointers to other cells. The size of pointer determines the maximum size of memory which can be addressed, and hence the maximum problem size that the machine can compute. As a minimum, it was felt that the SECD machine should be able to run the Lispkit compiler on Lispkit programs, and this required approximately 2^{12} words. This set a lower bound of a 28 bit word $((2 \times 12) + 4)$. The availability of memories in multiples of 8 bits made 32 bit words an appealing choice.

The resulting word configuration is as follows:



The S, E, C, and D stacks are implemented as 14 bit registers that will contain pointers to S-expressions in memory. The free list, used to allocate unused cells as required by the computation process, is similarly implemented by a register holding a pointer to the free list in memory. Further registers were added as their need was determined. Working registers *x1* and *x2* were added, to permit computation of intermediate results as arguments to a *cons* operation. A memory address register *(mar)* was added to select memory locations. A 32 bit **arg** register was added to hold integer or symbol arguments for alu operations, and generally for holding 32 bit records read from memory, including the machine instruction codes. The output of the **alu** is connected to two 32 bit buffer registers. The 32 bit **buf1** register is necessary since the integer and symbol inputs to **alu** operations come from the **arg** register and the **bus**, and the 32 bit output must be written

to some other register. The second buffer, **buf2** is used only by the *mark* routine of the garbage collector, to prevent loss of an arithmetic result being held in **buf1**.

With each addition to the hardware, a functionality or role was determined, and thereafter this functionality was respected. Non-transparent uses of components was avoided, with the hopeful expectation that this would make the verification task more manageable. The clearest indication of this approach is the provision of separate registers *(root, parent, y1, and y2)* for use by the garbage collector.

The description of this level is contained in a simulation characterized by data records consisting of 32 bits, 14 bit pointers, a fixed set of registers, and the modelling of memory, control and combinational logic elements by high level routines. A sample of the code at this level implements the same operations as shown for the abstract system level simulation (for the AP instruction).

```
                                        ! x2 := (cons (cdr c) d)
bus(x2=d);
bus(x1=c);
bus(mar=x1);
bus(x1=cdrvalue(memory[mar]);
consx1x2();                             ! resulting cell address is in mar
bus(x2=mar)

                                        ! x2 = (cons e x2)
bus(x1=e);
consx1x2();
bus(x2=mar);

                                        ! d = (cons(cdr(cdr s)) x2)
bus(x1=s);
bus(mar=x1);
bus(x1=cdrvalue(memory[mar]);
bus(mar=x1);
bus(x1=cdrvalue(memory[mar]);
consx1x2();
bus(d=mar);

                                        ! e = (cons (car(cdr s)) (cdr(car s)))
bus(x2=s);
bus(mar=x2);
bus(x2=carvalue(memory[mar]);
bus(mar=x2);
bus(x2=cdrvalue(memory[mar]);
bus(x1=x);
bus(mar=x1);
bus(x1=cdrvalue(memory[mar]);
bus(mar=x1);
bus(x1=carvalue(memory[mar]);
consx1x2();
bus(e=mar);

                                        ! c = (car(car s))
bus(c=s);
bus(mar=c);
bus(c=carvalue(memory[mar]);
bus(mar=c);
```

```
bus(c=carvalue(memory[mar]);
                                        ! s = NIL
bus(s=NIL);
```

The instruction sequence in the simulation was systematically derived from the previous abstract system level simulation. The bus function used for all data transfers models a single bus architecture, which was selected for simplicity. The 3 data structure operations are translated as follows:

```
car(z):       bus(x1 = z);
              bus(mar = x1);
              bus(x1 = carvalue(memory[mar]));
cdr(z):       bus(x1 = z);
              bus(mar = x1);
              bus(x1 = cdrvalue(memory[mar]));
cons(z1,z2):  bus(x2 = z2);
              bus(x1 = z1);
              consx1x2();
```

Five combinational logic elements are indicated. These are the ALU, the flagsunit, the consunit, and the carvalue and cdrvalue units. The latter three implement the primitive operations on records. The first two require further elaboration.

The ALU primary function is the computation of values for the arithmetic SECD machine operations: ADD, SUB, MUL, DIV, and REM. Additional operations are required for the garbage collector, including setting and clearing of the mark and field bits, and the desctructive *replcar* and *replcdr* operations used for the in-place traversal of the data structures in memory. These operations were masked previously by recursive functions implementing the garbage collector. Lastly, there is a *decrement* operation, used in looking up values in the environment. It is also used in the "sweep" phase of garbage collecting to step through the memory address space. The high address value is built-in as a fixed value to provide a starting point for the sweep. The two uses have distinct data type arguments, the first uses integers, while the second is applied to addresses. Thus, the 14 bit addresses must be padded out with 0's to make 28 bit integers. For this purpose, a constant register *(the clearunit)* will load 0's onto the upper 14 bits of the bus when required.

The flagsunit will return the boolean result of predicates used both for the control of the *if ... then ... else* and the *while* structures, as well as computing the SECD machine operations EQ and LEQ.

5 The Concrete Register Transfer Level

Refining the abstract register transfer level view concentrates on transforming the control sequence into a series of microcode instructions. The resulting sequence is then compiled into a binary image used to define a microcode ROM.

A microcode sequence is generated from the abstract register transfer level model by translating each of the higher level functions into an instruction sequence as follows:

```
bus(z = w)                          → rw wz
bus(z = carvalue(memory[mar]))      → rmem wcar ; rcar wz
bus(z = cdrvalue(memory[mar]))      → rmem wz
consx1x2()                          → call(Consx1x2,$)
```

A simple transfer of values on the bus becomes simultaneous read and write signals to the appropriate registers. The car operation requires that the word be fetched from memory, and its car field be accessed by writing it first to the car register, and thence transferring it on the bus to the desired register. It is assumed that the cdr field can be written directly from the memory to the selected register. Finally, the cons operation is called from so many different locations that it is treated as a subroutine call, with the following microcode location as the second parameter to enable a return from the subroutine on completion.

```
112, rd     wx2   (jump 113)                ! bus(x2=d);
113, rc     wx1   (jump 114)                ! bus(x1=c);
114, rx1    wmar  (jump 115)                ! bus(mar=x1);
115, rmem   wx1   (jump 116)                ! bus(x1=cdrvalue(memory[mar]);
116,              (call ("Consx1x2", 117))  ! consx1x2();
117, rmar   wx2   (jump 118)                ! bus(x2=mar);
118, re     wx1   (jump 119)                ! bus(x1=e);
119,              (call ("Consx1x2", 120))  ! consx1x2();
120, rmar   wx2   (jump 121)                ! bus(x2=mar);
121, rs     wx1   (jump 122)                ! bus(x1=s);
122, rx1    wmar  (jump 123)                ! bus(mar=x1);
123, rmem   wx1   (jump 124)                ! bus(x1=cdrvalue(memory[mar]);
124, rx1    wmar  (jump 125)                ! bus(mar=x1);
125, rmem   wx1   (jump 126)                ! bus(x1=cdrvalue(memory[mar]);
126,              (call ("Consx1x2", 127))  ! consx1x2();
127, rmar   wd    (jump 128)                ! bus(d=mar);
128, rs     wx2   (jump 129)                ! bus(x2=s);
129, rx2    wmar  (jump 130)                ! bus(mar=x2);
130, rmem   wcar  (jump 131)
131, rcar   wx2   (jump 132)                ! bus(x2=carvalue(memory[mar]);
132, rx2    wmar  (jump 133)                ! bus(mar=x2);
133, rmem   wx2   (jump 134)                ! bus(x2=cdrvalue(memory[mar]);
134, rx     wx1   (jump 135)                ! bus(x1=x);
135, rx1    wmar  (jump 136)                ! bus(mar=x1);
136, rmem   wx1   (jump 137)                ! bus(x1=cdrvalue(memory[mar]);
137, rx1    wmar  (jump 138)                ! bus(mar=x1);
138, rmem   wcar  (jump 139)
139, rcar   wx1   (jump 140)                ! bus(x1=carvalue(memory[mar]);
140,              (call ("Consx1x2", 141))  ! consx1x2();
141, rmar   we    (jump 142)                ! bus(e=mar);
142, rs     wc    (jump 143)                ! bus(c=s);
143, rc     wmar  (jump 144)                ! bus(mar=c);
144, rmem   wcar
145, rcar   wc    (jump 146)                ! bus(c=carvalue(memory[mar]);
146, rc     wmar  (jump 147)                ! bus(mar=c);
147, rmem   wcar  (jump 148)
148, rcar   wc    (jump 149)                ! bus(c=carvalue(memory[mar]);
149, rnil   ws    (jump 150)
150,              (jump "top_of_cycle")     ! bus(s=NIL)
```

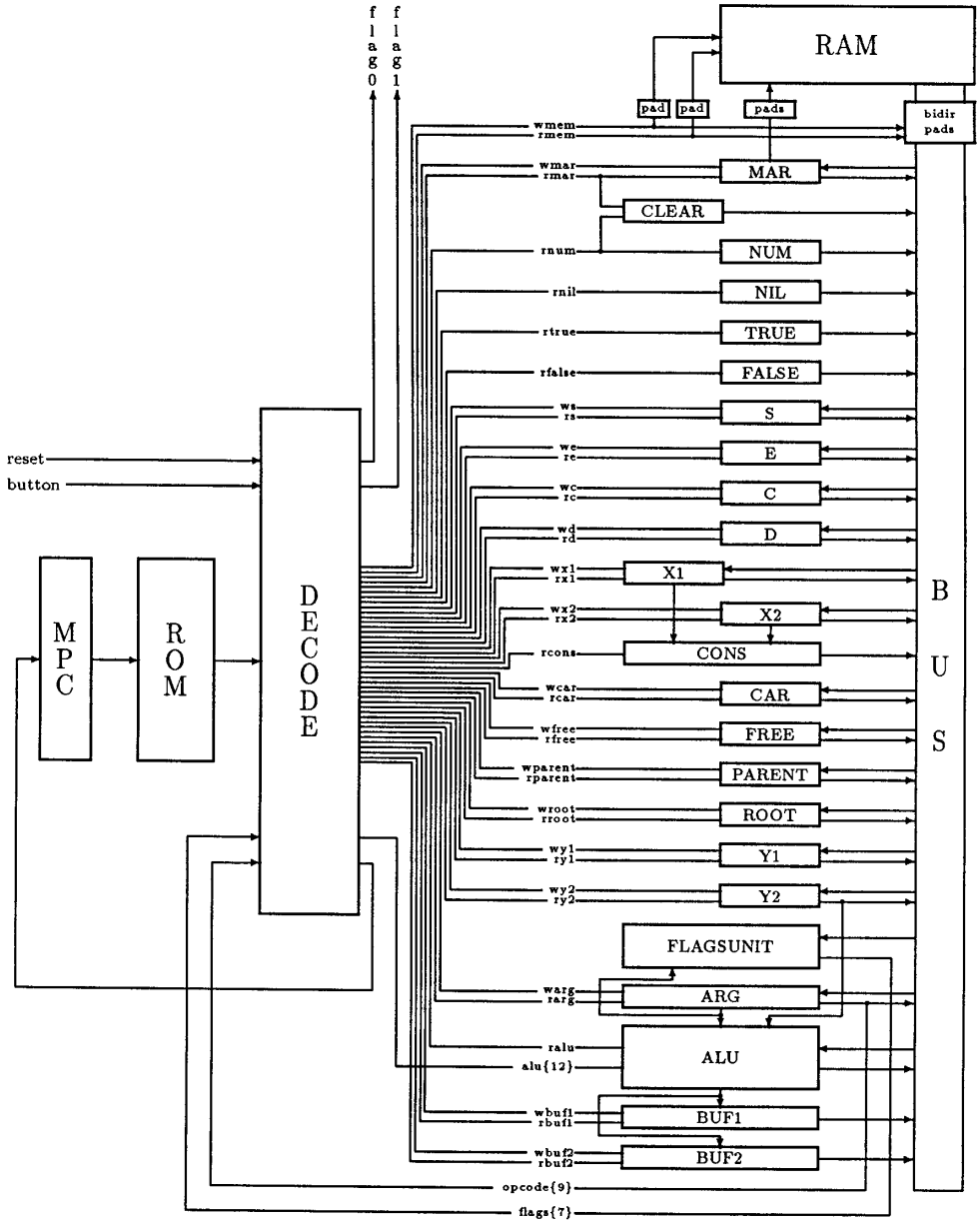


Figure 2: Register Transfer Level View of SECD Machine

The set of datapath register (and memory) control signals is now established, as are the 12 ALU operations, given in the previous level. Control mechanisms for the microcode consist of 5 types: unconditional jumps, conditional jumps, subroutine calls, subroutine returns, and a jump table that uses the current machine instruction value. Of these, the conditional jumps and subroutine calls both require 2 addresses, while the others require a single address argument. The subroutine mechanism required a microaddress stack. A total of 8 conditional jump instructions were required, with conditions consisting of:

- the value of the *button* input
- is the argument an *atom* record
- the EQ operation applied to 2 arguments
- the LEQ operation applied to 2 arguments
- is record equal to the symbolic constant "NIL"
- is the record equal to the symbolic constant "TRUE"
- is the *mark* bit set
- is the *field* bit set

These values will be the flagsunit outputs. Additionally, the datapath will also send the current machine instruction value to the controller. The controller has thus 12 distinct ways of computing the next microinstruction address to select.

This evolution was suitable for mechanical rewriting, ignoring optimizations that would be obvious to the reader. The final microcode evolved through several successive refinements, aimed at reducing the ROM size. The major optimizations include:

- A peephole optimization run eliminated unnecessary bus transfers using the working x1 or x2 registers, when the value could be transferred directly to the required register in one operation.
- An external reset input was added, as indicated in the top level FSM view.
- An additional reserved memory location was added, in which a pointer to the problem in memory would be installed for the use of the SECD chip in initializing its state registers. Upon completion of the program, the location is used to return a pointer to the solution. This replaced a bank of switch inputs planned to provide a problem location input.
- Unconditional jump instructions and subroutine calls were combined with the previous instruction in sequentially executed code.
- An additional level of subroutining was added to share common code sequences in arithmetic and logical SECD machine instruction sequences.
- A jump table used to implement the "case" instruction at the head of the instruction execute cycle was relocated in the microcode so that the actual instruction value could be used as the microcode address, rather than having to add it as an offset from the first jump table address.
- An early version of the microcode had anticipated an on-chip RAM, which defaulted to read. While the output was gated to the bus, the car field was gated directly to the car unit, which was simply a combinational device. Thus, carvalue(memory[mar]) did not require a cycle to read from memory. The change involved making the carunit a register, and adding a cycle to load it from memory where required.
- The value of binary flags was used on the cycle following the read from memory of one of their arguments. The implied latching arrangement was eliminated in favor of reading the argument in the same cycle in which the flag value was used.
- Again, an early version had an error condition for an invalid machine instruction code (just the specific code 0). This error would be a compiler error only, and was eliminated from on chip testing.

- The status flags for unary operations changed from using the value on the bus to using the contents of the ARG register, just as the binary **alu** operations.
- An odd timing regimen implying the use of paired (master-slave) latches in the datapath (the contents of ARG was decrimented and written back to ARG on the same microcycle) was eliminated, and a buffer registers added for ALU output, as mentioned earlier.
- Slight revision to the ordering of conditional jump instructions was made so that the following microinstruction was always the default next address. This permitted the elimination of this default address field as an explicit value in each microcode instruction, since the only other instruction using two addresses was the subroutine call, and the return address parameter to a subroutine call was always the immediately following address as well. Thus, at most one address field was required in each microinstruction. The unconditional jump instruction was divided into remote jumps and jumps to the following microinstruction, with the latter not needing a specified argument. This increases to 13 the number of ways of choosing the next **mpc** contents.

The final version of the microcode for the AP instructions is as follows:

```
L("AP");
        rd      ; wx2                   ; (inc ()) ;
        rc      ; wmar                  ; (inc ()) ;
        rmem    ; wx1                   ; (call ("Consx1x2")) ;
        rmar    ; wx2                   ; (inc ()) ;
        re      ; wx1                   ; (call ("Consx1x2")) ;
        rmar    ; wx2                   ; (inc ()) ;
        rs      ; wmar                  ; (inc ()) ;
        rmem    ; wx1                   ; (inc ()) ;
        rx1     ; wmar                  ; (inc ()) ;
        rmem    ; wx1                   ; (call ("Consx1x2")) ;
        rmar    ; wd                    ; (inc ()) ;
        rs      ; wmar                  ; (inc ()) ;
        rmem    ; wcar                  ; (inc ()) ;
        rcar    ; wmar                  ; (inc ()) ;
        rmem    ; wx2                   ; (inc ()) ;
        rs      ; wmar                  ; (inc ()) ;
        rmem    ; wx1                   ; (inc ()) ;
        rx1     ; wmar                  ; (inc ()) ;
        rmem    ; wcar                  ; (inc ()) ;
        rcar    ; wx1                   ; (call ("Consx1x2")) ;
        rmar    ; we                    ; (inc ()) ;
        rs      ; wmar                  ; (inc ()) ;
        rmem    ; wcar                  ; (inc ()) ;
        rcar    ; wmar                  ; (inc ()) ;
        rmem    ; wcar                  ; (inc ()) ;
        rcar    ; wc                    ; (inc ()) ;
        rnil    ; ws                    ; (jump ("top_of_cycle")) ;
```

This modelling of the chip implicitly assumes an external RAM, since an outside agency is expected to download problems and upload results, and there is no provision in the model for handing control of the memory to the external agency. External RAM is consistent with the simplicity criterion, and focussed our effort on the microprocessor design, rather than the distinct concerns of RAM design. The RAM is treated as just another, though addressable, register, with read and write signals controlling it. It was expected that the RAM would default to a read operation, and the rmem control line would control its gating onto the bus.

This implementation of the controller is a classical finite state machine design. We view the controller as a finite state machine, with the state held in the **mpc** register (this is extended to include the microcode stack registers as well), changing with each microcode instruction executed.

6 The Mossim and Layout Levels

The next level will model the SECD down to transistors. The simulation used is an implementation of Randall Bryant's Mossim simulator, written in C by Jeff Joyce, with a Common Lisp interface, called CDL, written by Breen Liblong. The complete CDL definition of the SECD chip is available in [GWB+89]. The layout design was completed using the 'Electric' system ([Rub87]).

The mossim level model was used to capture component design for simulation, and also guided the actual layout. Layout decisions determined the mossim definition, and the mossim model was used as a suitable form to define the components then implemented in the layout.

A full custom design was selected in preference to gate array or semi-custom using a standard cell library. Our team had the expertise to undertake full custom design, and it also provided a better learning experience than the other options. Further, non-custom designs suffered from constraints, including the number of gates in gate array, and the availability of suitable cell libraries. Full custom fabrication was available through MOSIS, with a clear and concise set of scalable design rules.

Concerns about the limited likelihood of fabricating a working chip on the first try and the ability to determine causes of failure led to the decision to introduce some degree of testability into the design. Previous views of the SECD divided it into two major functional components: the controller and the data path. As described earlier, these two parts were developed somewhat independently, and it was felt that they should be independently testable. If a flaw occurred in one component, testing of the other component would still be possible. To meet this objective, it was decided to add a bank of shift registers between the two components, which could be used to trap all, or most, signals passing between them[4].

6.1 Timing and Clocking

Implementing the controller as a finite state machine requires buffering between current and next states. This is achieved by the use of a two-phase non-overlapping clocking scheme and paired registers, along the design style described in [MC80]. The state register in the control unit is the **mpc** register, but in a more general sense, the values on the 4-deep microcode subroutine stack are also part of the state. In the following discussion, references to the **mpc** register can be applied similarly to the stack registers.

Level-triggered registers were selected:

- for space/transistor count efficiency.
- level triggered latches are in keeping with the view of circuits presented in [MC80] as a system of opening and closing valves.
- previous experience on Tamarack2[5].

In some sense, level triggered latches can be viewed as falling edge triggered, although the state is lost at the start of the clock pulse. The **mpc** register is actually the second of the pair of state registers, the first is labelled **nextmpc**. **Nextmpc** is clocked on ϕ_A and **mpc** on ϕ_B. The control unit state is considered to change on ϕ_B.

Particular attention was paid to possible *race* conditions. One example was the possibility of generating transient "write" signals from the ROM when the value **mpc** is changing. The solution was to delay latching of the datapath registers until after the **mpc** is latched (and the

[4] Detailed listings of signals in the shift registers are given in [GWS89]

[5] a second implementation of the Tamarack chip [Joy88], an implementation of Gordon's toy computer [Gor83]

value propagated). Use of the inverse clock signal ($\overline{\phi_B}$) was still subject to race conditions[6], so the ϕ_A phase was used. The clocking scheme requires that inputs to registers latching on ϕ_A be stable prior to the end of the ϕ_A pulse. The overall view of the chip now sees the control unit as changing state on ϕ_B and the datapath changing on ϕ_A.

The first layout iteration, and the mossim simulation, really did not properly account for the operation of the external memory. It was expected that memory outputs would float unless the rmem signal was asserted. Capacitance induced delays of signal switching and power disapation caused by both memory and SECD devices driving external lines simultaneously for some overlapping interval were seen as potential problems later. Thus, a more considered memory interface timing was developed (described in detail in [GWS89]), and the new control signal logic added in a design revision.

6.2 Floorplanning

Mosis offered a 3μ double metal p-well CMOS process, in dies that permitted maximum project sizes of:

2.3 × 3.4 mm
4.6 × 6.8 mm
6.9 × 6.8 mm
7.9 × 9.2 mm

Initial estimates indicated we would require the largest size, and the size was a fixed constraint. A lengthy delay between the completion of the layout (late 1986) and the chip fabrication (fall 1988) permitted us to use the 2μ process introduced by Mosis in the interim. The use of scalable design rules was vital in enabling us to take advantage of the new technology *without any redesign.*

The major functional components, namely the datapath and controller, from the register transfer level were maintained as major floorplan elements. The shift register block was located between these two, and all were surrounded by a padframe.

6.3 Design Guidelines

The project team had already completed a microprocessor layout (the Tamarack2) and thus had some experience in layout. The design would use a cell library, with guidelines rigidly controlling the cell designs. Power and ground rails occurred at top and bottom of cells in metal-2, and bit slices were arranged so one rail were shared between two adjacent slices. Data generally flowed horizontally through the cells in metal-1, while control signals and clock lines run vertically in polysilicon. The use of metal-2 was restricted within cells to the rails, so that it could be freely used for horizontal interconnect running over the cells without any design rule violations.

Cell height was selected based on the example of past work, and by building sample cells using different heights. An optimal value of 86λ was selected. Each cell was to be self-sufficient, so all required well/substrate contacts were internal. Port locations and boundary clearances were standardized, and multiple instances of ports was encouraged to ease cell composition.

All library cells were defined and exhaustively simulated in mossim. Layout cells used the same root name for ports as the mossim definition, and a one-to-one correspondence between the two definitions was attempted through all levels in the design hierarchy. The XOR cell was an exception, since mossim could not correctly model the 6 transistor design actually used.

[6] One could devise a scenario where the $\overline{\phi_B}$ signal overlaps the start of the ϕ_B pulse, and thus random write signals may be generated.

6.4 Controlunit

Several key decisions directly affected the design of the controlunit:

- the encoding of the microcode
- the interface between the controlunit and datapath.
- the choice of signals trapped by shift registers.

A simple view of the ROM has a fully horizontal microcode, with discrete outputs for each read, write, and alu control signal (23, 17, and 12 signals respectively). Further, the address field required by the goto, subroutine call, and conditional jump instructions was 9 bits (since the microcode length was approximately 400 instructions). Lastly, the method of selection of the next microinstruction has 13 possibilities. A fully horizontal microcode ROM would be 9 $\times$ 400 $\times$ 74. With a square pitch of 12.5λ for the ROM layout, in the 3μ process, we had a ROM size of approximately 2.06 $\times$ 7.5 mm, excluding routing to and from, and buffering of inputs and outputs. While this size might fit on the chip, it was felt that we could reduce it considerably, and in the process also reduce line capacitance and thus provide a higher probability of reliable operation.

Microcode characteristics were examined in the search for an encoding scheme. The mutually exclusive assertion of individual read, write, alu, and test signals during any cycle suggested these signals could be encoded. The number of distinct combinations of control signals in a microinstruction numbered approximately 120, while the number of distinct combinations of read and write signals numbered approximately 86. Further, the address and alu fields are sparse in the microcode; alu instructions appeared 17 times in total, while the address field was used approximately 118 times. The simple encoding of read, write, alu control, and test fields to microinstruction fields was selected, since it was a natural way of breaking up the signals, permitting easier examination for error detection, and would be a simple encoding to verify later on. This reduced the microinstruction word length to 27 bits.

Physical arrangements for the ROM layout were also considered. The sparse address field, and the correspondence between the use of this field and the test field selecting other than the next instruction, suggested using 2 ROM devices, one for the read, write, and alu fields, and the other for the address and test fields. The two devices would be $9 \times 400 \times 14$ and $9 \times 115 \times 13$ respectively. The alu fields could be generated directly from the address decoder outputs, and reordering the decoder outputs (to something conceptually closer to a PLA structure than a ROM) could enable sharing of a single decoder between the 2 devices. This scheme was abandoned, largely because of its complexity, and the implicitly inconsistent treatment of outputs, and the line lengths resulting from the need for the full 9 $\times$ 400 decoder. Also, the savings generated by the simple encoding already brought the ROM into an acceptable size. Reduction of the length of internal lines in the ROM was achieved by the transformation to a $7 \times 100 \times 104$ ROM, with 2 bit column decode. This produced a nearly square device, and considerable flexibility in the control unit layout. The ROM layout was generated automatically from a bit pattern produced from the microcode simulation of the previous level. The decoder component is a fully complementary CMOS device, while the 'OR' is implemented in a pseudo NMOS design, using pullup transistors in each column, and n-type devices exclusively in the plane.

Since deciding to have the ROM output encoded signals, decoders were required. It was possible to decode fully within the controlunit, or permit the datapath to decode. The decoders were included in the controlunit because they did not easily fit with the bitslice dominated layout approach of the datapath. Further, if the decoded signals were routed through the shift registers, more flexibility control was available in debug operation mode. The same automated ROM/PLA generator was used to produce the layout of all 3 decoders.

The 13 alternative methods of selecting the next microcode instruction select from 4 possible values only:

- the address following the current one in the microcode,
- the address supplied as a field in the microcoed (for historical reasons called the A address),
- the SECD machine instruction code (opcode), and
- the top of the microcode subroutine stack value.

A 4 × 1 MUX gates these values to nextmpc register. The mux control signals are generated by a *test* field of the microcode, in combination with the value of the flag and button inputs. The logic is implemented in a PLA, again generated automatically.

A bit-slice approach is used for the **mpc** and **nextmpc** registers, the microcode stack, and the logic to implement the selection of the next microinstruction. The required control signals for this are generated from a single row of random logic. Other signals, such as the control for the bidirectional i/o pads, were generated from random logic that was located with the PLA device.

6.5 Datapath

While *symbol* and *cons* record representation was fixed at an earlier stage, the representation of integers was left until the design of the datapath. The use of the **alu** to decrement addresses as well as integers required that the mapping of 14 bit addresses to 28 bit integers (accomplished by clearing the upper 14 bits) should be consistent with the representation of integers. While both 2's complement and sign magnitude conformed to this constraint, the use of 2's complement representation produced a simpler **alu** implementation.

The datapath was designed around a 32 bit bus, connecting all the registers and combinational logic devices. Most registers are simply 14 bits, connected to the *cdr* field of the bus, aside from the alu output **buffer** registers and the **arg** register which are all 32 bits. The **car** register inputs are connected to the upper 14 bit address field of the bus, while its outputs are connected to the lower (*cdr*) field. The **x1** register inputs and outputs both connect to the *cdr* field of the bus, but the output additionally connects to the *car* field inputs of the **consunit**. The **clearunit** sets the upper (*car*) field of the bus to zeros when the **mar** or **num** registers are read, because these are the sources of addresses that are decremented by the **alu**. This operation effectively maps a 14 bit address to a 28 bit integer. The **alu** was simplified by dropping the 3 most complex (in terms of area) operations: mul, div, and rem. The opcodes were not eliminated however, and the implementation let them default to the dec operation.

Registers are grouped into subcomponents: **regs-14-misc**, **regs-14-car**, **regs-14-y2**, **regs-32-arg**, and **regs-32-bufs**. No error checking of type bits is implemented for any operations in the datapath. Logical **alu** operations maintain the (unaffected) bits, while the arithmetic operations produce 32 bit output with the type bits set to integer, and both mark and field bits cleared. Similarly, the **consunit** outputs a 32 bit value with record bits set to *cons* and mark and field bits cleared.

Once the padframe was designed, it was found necessary to add one additional unit to the datapath. This **read-mem** unit allows the input values from the bidirectional pads to be passed onto the bus only when the rmem signal is high. This is necessary since the bidirectional pads were designed as write-enabled, and default to input mode. The busgates prevent the pads from writing onto the bus when not reading from memory.

6.6 Shift Registers

The shift register block is a simple device used in the previous Tamarack2 design. It uses separate controls and clocks[7], and is used to take a "snapshot" of the state of the chip, or to enter a vector for

[7] The use of distinct clocks for system and shift registers was chosen to simplify the logic design and improve the probability of obtaining working subcomponents on the chip by minimizing operational dependencies.

testing. Every signal that was considered reasonably useful for testing was routed through the shift registers. This included all read, write (except the write memory signal which was initially expected to be exported directly), and alu signals, the status flags from the datapath, and additionally, the mpc contents (the lines between the **mpc** register and the ROM), and the control signals for the 4×1 MUX feeding the **nextmpc** register, and controlling the microcode stack. In total, 72 bits are trapped by the shift registers. The only value passing between these components that was not trapped was the machine instruction code. Passthrough lines were provided in the shift registers for this signal. Additionally, the control signal for the bidirectional pads, which was added at a late stage in the design, was not trapped. Unfortunately, this made examining datapath register contents more difficult, as described in [GWS89].

6.7 Padframe

The original die size and package had a limit of 64 pins. Bidirectional pins (32) were inevitable for the bus connection. The MAR output required 14 pins, system clocks 2 pins, control inputs 2 pins, state output signals 2 pins, shift register controls, input, and output a total of 4 pins. 2 pins were used for the separate shift register clocks, instead of using one pin for a clock control input. Lastly, 2 power and 2 ground pins are used. A pair of power and ground pins (called *dirty power* and *dirty ground*) drive the pads only, to reduce noise on the supply lines to the chip, while the other pair drives the rest of the chip. In the final layout, the distribution of the pads around the chip perimeter was constrained by the number of bonding fingers along each cavity edge of the package, and a maximum 45 degree angle of bonding wires.

Aside from the power and ground pads, there are input, output and bidirectional pads. Simple input and output pads are on little interest, but the design of the bidirectional pads required more effort. When used in output mode, the pad must increase the drive strength of the signal. A step-up buffer is used for this, but it cannot drive on input mode. Thus the circuitry turns off both n and p-transistors by providing 0 and 5 volt gate inputs respectively. The designs were simulated using SPICE, and switching times in the 20 nanosecond range were achieved using a load capacitance of 50 pf. This speed was quite acceptable, given the constraints stated earlier.

7 Formalizing the SECD Representation

This section concerns formalizing the implementation and specification of the SECD machine in HOL. We consider 3 levels of definition: the lowest level (implementation) definition, the intermediate (register transfer) level, and the top (abstract system) level. The discussion focuses on the representation of time, clocking and data types, and how they are used to specify the system behaviour. Abstraction mechanisms between levels of representation are described.

7.1 Time

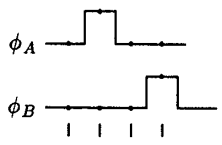
The three levels each have a distinct granularity of time. The implementation level uses the finest grain, while the register transfer level grain is that of the clock cycle. The top level granularity relates to the time to execute individual SECD machine instructions.

7.1.1 Lowest Level

The finest granularity of time represented must permit the capture of the essential behaviour of every signal. Specifically, describing the signal using a finer sampling of time should not detect patterns that are not expressed at the chosen granularity of time. The choice of a fully static design determines that the essential behaviours of generated signals will be describable in terms of their settled values, assuming a clock rate that permits them to settle. The finest granularity needed is that which captures the behaviour of the clock signals. Further, we constrain all input signals appropriately to validate their abstraction to these discrete points in time.

The grain of time described bears no direct relation to real time, but instead corresponds to the changing of values on clock lines. Thus, if the clock is stopped for any interval, there are no points of fine grain time in that interval.

We use a 2 phase non-overlapping clock. This is abstracted to two boolean signals (referred to as phases), ϕ_A and ϕ_B, with values defined at 4 points of fine grain time per clock cycle: at 2 points one signal is asserted, alternating with points when neither is asserted:



We have used a second pair of clock phase lines for clocking the **shift register** unit. The intended mode of operation will not permit cycling of both clocks simultaneously. This unusual clock arrangement obscures the simple relation of clocks to the granularity of time just described. The interesting point is that intervals of the finest grain of time must correspond to intervals for either system of clocks. We express the desired clocking behaviour in terms of predicates applied to both sets of clock phase lines, requiring that complete cycles will be executed when any clock is asserted. Further, advances in time correspond to cycling one of the clocks. This description of two independent clock pairs provides a logical expression of the system we have designed.

The behaviour of the 4 clock lines during any given clock cycle can be described using the following predicates:

$\forall$ (t_0:ftime) (f:fsig).
CycleA t_0 f $\equiv$ ($\neg$f (t_0)) $\wedge$ (f ($t_0 + 1$)) $\wedge$ ($\neg$f ($t_0 + 2$)) $\wedge$ ($\neg$f ($t_0 + 3$))

$\forall$ (t_0:ftime) (f:fsig).

CycleB t_0 f $\equiv$ ($\neg$f (t_0)) $\wedge$ ($\neg$f ($t_0 + 1$)) $\wedge$ ($\neg$f ($t_0 + 2$)) $\wedge$ (f ($t_0 + 3$))

$\forall$ (t_0:ftime) (f:fsig).
noCycle t_0 f $\equiv$ ($\neg$f (t_0)) $\wedge$ ($\neg$f ($t_0 + 1$)) $\wedge$ ($\neg$f ($t_0 + 2$)) $\wedge$ ($\neg$f ($t_0 + 3$))

$\forall$ ϕ_A ϕ_B:fsig (t:ftime).
CYCLE_f ϕ_A ϕ_B t $\equiv$ CycleA t ϕ_A $\wedge$ CycleB t ϕ_B

$\forall$ ϕ_A ϕ_B:fsig (t:ftime).
no_CYCLE_f ϕ_A ϕ_B t $\equiv$ noCycle t ϕ_A $\wedge$ noCycle t ϕ_B

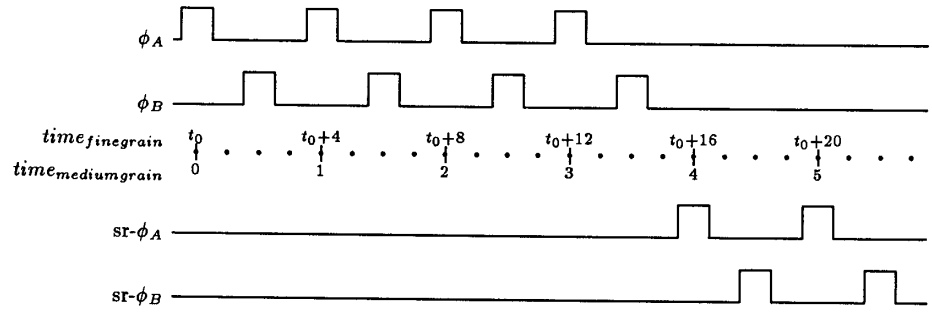
$\forall$ ϕ_{A_1} ϕ_{B_1} ϕ_{A_2} ϕ_{B_2}:fsig (t:ftime).
CLOCK_CYCLE_f ϕ_{A_1} ϕ_{B_1} ϕ_{A_2} ϕ_{B_2} t $\equiv$ CYCLE_f ϕ_{A_1} ϕ_{B_1} t $\wedge$ no_CYCLE_f ϕ_{A_2} ϕ_{B_2} t

In each clock cycle interval, we shall require that the following holds:

CLOCK_CYCLE_f ϕ_A ϕ_B sr-ϕ_A sr-ϕ_B t $\vee$ **CLOCK_CYCLE_f** sr-ϕ_A sr-ϕ_B ϕ_A ϕ_B t

7.1.2 Register Transfer Level

The time grain for the Register Transfer level corresponds to 4 intervals of finest grain time, or precisely one clock cycle (of either system or shift register clock). For the system clock, the cycle consists of the 4 points beginning with the point when ϕ_A is asserted. (A similar pattern applies to the shift register clock.) We map points in medium grain time to the first of these points.



The existence of discrete clock phase lines is hidden entirely at this level. Normally, a register transfer view abstracts away the clock entirely, but the existence of two distinct clocks requires that we maintain signals at this level to indicate whether the system or shift register clock advances in any given clock cycle. Thus, registers must retain an actual clock input, rather than treating the clock as inherent in the time parameter. The correctness goal for the system will be constrained to the case where only the system clock advances.

7.1.3 Top Level

The most coarse 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, Error1, Error2,* 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 as was the mapping from medium to fine grain time, 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 method of defining such mapping functions is well described in [Mel88].

7.2 Data Types

7.2.1 Lowest Level

We represent the clock phase lines as functions from the fine grain of time to boolean values. Boolean values are used to model all signals, aside from pseudo-nmos regular structures where the modelling of pullup devices requires a tristate logic. The validity of this abstraction relies upon several assumptions, including the ability of all signals to settle within one time interval and that devices output acceptably strong-valued signals. Floating values of outputs are modeled by the use of implication for defining behaviour. This gives rise to a proof obligation to show that circuit nodes that may potentially be driven from multiple sources are only driven by at most one source at any point in time. The problem is most readily observed for the single internal bus. Since many values are gated onto the bus, the value of the bus is expressed by the use of implication. If two devices are simultaneously driving opposite values onto the bus, the situation arises in the abstracted circuit model where "T = F". This problem is constrained if we prove independently that no two devices ever drive the bus simultaneously.

Combinational logic devices are modelled as instantaneous, consistent with the assumption that time intervals are sufficiently long to permit all signals to settle. Memory devices (registers or latches), can be defined simply by giving the output at (fine grain) time $(t+1)$ in terms of input and clock signal at time (t).

latch wrt in out ≡
    ∀ t:time. out (t + 1) = wrt t ⇒ in t | out t

This type of definition is appropriate where the circuit consists of sequences of memory elements clocked on alternating clock phases and separated by combinational logic. The level-triggered latch devices used will actually produce the new output at the start of the clock cycle, rather than at the end, although the value is latched at the falling edge of the clock. An optimization in the fetch instruction microcode makes use of this observation, by utilizing the output of one latch (the **arg** register) as an input of another (the **nextmpc** register), with both clocked on the same clock phase. This has required a redefinition of the latch.

latch wrt in out ≡
    ∀ t:time. out (t + 1) = wrt (t + 1) ⇒ in (t + 1) | out t

This redefinition has the effect of shifting the output signal by one point in fine grain time: the new output appears one interval earlier that with the previous definition.

7.3 Register Transfer Level

Signals abstracted to this level are really a sampling of the signals described at the finer grain. Notice that signals hold the same value through 4 points in fine grain time starting at the point when the register giving rise to the signal is latched. This holds for the **mpc** contents, from the point when ϕ_B is asserted, and is a property we can prove for most outputs, given the inputs behave and we clock as described, but other signals are off phase. Thus two sorts of stability are identified, which we shall label ϕ_A-stability and ϕ_B-stability, determined by the clocking of the memory device *producing* the signal.

It is clear that an output with ϕ_B-stability is suitable as in input to a device clocked on ϕ_A, and vice-versa. Further, with the definition of the latch behaviour above, devices clocked on the

same clock line may feed each other *provided that there is no circular feedback path*[8]. Datapath registers are clocked on a signal dependent upon both ϕ_A and the write signal for the register, and thus will not change every clock cycle. Clearly then, such a datapath register which is not clocked in a cycle can feed any device latching on ϕ. These complete the set of constraint relations that arise between signals within the SECD chip at this level.

An external *reset* input has been provided to permit a deterministic startup of the chip state. Asserting the *reset* input during a pulse of ϕ_B will force the **mpc** to 0 (the *idle* state). We require the reset line to be ϕ_A-stable, and asserted at time t_0, and never reasserted at any later cycle. Further, the system clock ϕ_B must be asserted at the same time, in order for the reset to occur. The button input will be constrained to ϕ_B-stability, since it is an input to the **nextmpc** register latching on ϕ_A. ϕ_B-stability of the **mpc** and **flag** outputs is readily derived from clocking constraints. ϕ_A-stability of the datapath registers and memory is derivable from the clocking constraints, the ϕ_B-stability of the **mpc**, and the resulting ϕ_B-stability of the register control signals.

The behaviour of the system at this level is expressed in the composition of major components, whose behavioural specification is proven from the implementation level. The control unit is defined by 3 blocks: the **mpc** register and the 4 deep microcode stack, a **ROM**, and a **DECODE** component, consisting entirely of combinational logic. The datapath consists of registers and combinational logic devices (the **alu**, **consunit**, **flagsunit**, and an array of transmission gates isolating the bus from memory output when not reading a memory location), with the bus modelled as the wiring together of these devices. The behaviour of the shift registers when disabled[9] is proved from the implementation, and we constrain the shift register controls and clock accordingly. Additionally, the logic of the padframe (the bidirectional pads particularly) is specified in terms of a mixture of 'wordn' and discrete signals, once again proven from the implementation.

7.3.1 Top Level

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

The method of implementing recursive function definitions as closures with a circular environment component raises the complexity of the data representation problem considerably. Such circular S-expression lists, created by destructive operations, cannot be mapped to a simple recursive type. Further, structure is shared by S-expressions, particularly the environment component of closures. In defining mutually recursive functions originating within a LETREC in LispKit, each function closure references the same environment, which is also in the E "stack". When a destructive replace operation is performed to create the circular list structure, the change affects all those components simultaneously.

Thus, a much lower level of representation has been chosen to describe the top level specification. Rather than directly defining transformations to S-expression data type structures, 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, sub, mul, div,* and *rem.* The 4 state

[8] As described, the arg register is the only component feeding another register clocked on the same phase, and the output of this second register, the nextmpc register, is an input to only one memory device, which is latched on ϕ_B. Thus no circular feedback path exists.

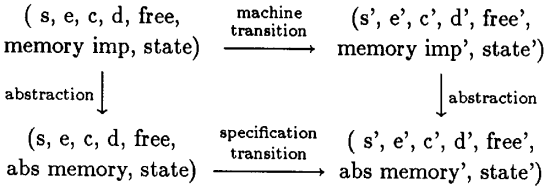
[9] Shift registers are disabled when not being clocked and when the input values are passed directly through to output.

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.

The implementation definition includes a memory function with simple read and write operations only. The task of the verification is to show that the sequence of operations performed on the real memory commutes with the specification transition of the abstract memory.



The abstract memory type μ is basically a function :

$$\mu = \delta \rightarrow (\delta^2 \cup \alpha)$$

where δ is the domain of the function and α 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, matching the type 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: $\mu = \delta \rightarrow ((bool \times bool) \times (\delta^2 \cup \alpha))$

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 μ function. The relevant built-in functions and their types are summarized in Table 2. 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.

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 *AP* 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.

We wish to verify the behaviour of the system only under very constrained conditions, representing the actual operating conditions of SECD. The major constraints include the following:

- Clock behaviour as described.

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

```
AP_trans (s:δ,e:δ,c:δ,d:δ,free:δ,MEM:μ) ≡
        let cell_mem_free = Cons(e, Cons_tr(d,CDR(c,MEM,free))) in
        let d_mem_free = Cons_tr( cell_of cell_mem_free,
                                  CDR(CDR(s,mem_free_of cell_mem_free))) in
        let e_mem_free = Cons (Car (CDR (s,mem_free_of d_mem_free)),
                               CDR (CAR (s,mem_free_of d_mem_free))) in
        (LK_NIL,                                  % S %
         cell_of e_mem_free,                      % E %
         Car (CAR (s,mem_free_of e_mem_free)),    % C %
         cell_of d_mem_free,                      % D %
         free_of e_mem_free,                      % free %
         mem_of e_mem_free,                       % memory %
         top_of_cycle)                            % FSM state %
```

Table 3: Transition for AP Instruction

- The reset input is asserted at the start of machine operation and is never subsequently asserted.
- Input signal stability as mentioned earlier.
- The shift registers are disabled and the shift register clock does not advance.
- The free list pointer is never NIL. (i.e. No garbage collection is required.) This constraint will be eliminated in our next verification attemp.
- The control list represents a valid program. This constraint concerns the form of the control list, limits the instruction codes to the 21 machine instructions, and requires the appropriate argument and environment structure for the individual instructions. For example, the arguments to the LD instruction must reference a position within the environment list.

```
NEXT (s:δ,e:δ,c:δ,d:δ,free:δ,MEM:μ) ≡
      let instr = Int_of (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 = AP)    ⇒ (AP_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 (MEM:μ_csig)
         (s:δ_cvec) (e:δ_cvec) (c:δ_cvec) (d:δ_cvec)
         (free:δ_cvec)
         (reset:csig) (button:csig)
         (state:state_csig)
≡ ∀ t:ctime.
  ((s (t+1), e (t+1), c (t+1), d (t+1), free (t+1), MEM (t+1), state (t+1)) =
   (reset t ⇒ (s t, e t, c t, d t, free t, MEM t, idle) |
   (state t = idle) ⇒
    (button t ⇒ (Cdr (CAR (LK_NUM, MEM t, free t)),
                  LK_NIL,
                  Car (CDR (LK_NUM, MEM t, free t)),
                  LK_NIL, LK_NIL, MEM t, top_of_cycle) |
                 (s t, e t, c t, d t, free t, MEM t, idle)) |
   (state t = error0) ⇒
    (button t ⇒ (s t, e t, c t, d t, free t, MEM t, error1) |
                 (s t, e t, c t, d t, free t, MEM t, error0)) |
   (state t = error1) ⇒
    (button t ⇒(s t, e t, c t, d t, free t, MEM t, error1) |
               (s t, e t, c t, d t, free t, MEM t, idle)) |
   (state = top_of_cycle) ⇒
    (NEXT (s t, e t, c t, d t, free t, MEM t))))
```

Table 4: Top Level Specification

Acknowledgements

This work is supported by the Department of National Defence, the Natural Sciences and Engineering Research Council of Canada, the Alberta Microelectronic Centre in Calgary, and the Canadian Microelectronics Corporation. The team that designed and specified the SECD chip in 1986 consisted of Graham Birtwistle, Mark Brinsmead, Brian Graham, Jeff Joyce, Mary Keefe, Wallace Kroeker, Breen Liblong, Walter Vollmerhaus, and Simon Williams. Jeff Joyce began the development of the SECD in early 1985, implementing an interpreter based on Henderson's book[Hen80], and subsequently designed a RTL implementation. The design team was led by Wallace Kroeker in 1986. Rick Schediwy supplied needed expertise throughout the layout stage.

References

[Bry80] R. Bryant. An Algorithm for MOS Logic Simulation. *LAMBDA*, 1980.

[Gor83] M. J. C. Gordon. Proving a Computer Correct using the LCF_LSM Hardware Description Language. Technical report 42, Computing Laboratory, University of Cambridge, September 1983.

[Gor85] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical report 68, Computing Laboratory, University of Cambridge, 1985.

[GWB+89] 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.

[GWS89] 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.

[HBGS89] M. J. Hermann, G. Birtwistle, B. Graham, and T. Simpson. The Architecture of Henderson's SECD Machine. Research report 89/340/02, Computer Science Department, University of Calgary, 1989.

[Hen80] Peter Henderson. *Functional Programming Application and Implementation.* Prentice/Hall International, 1980.

[Joy] J. Joyce. The secd machine, a study in advanced architectures. Unpublished report of cpsc 603 course work at the University of Calgary.

[Joy88] 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.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Company, 1980.

[Mel88] 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.

[Plo75] G. D. Plotkin. Call–by–name, call–by–value, and the lambda calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[Rub87] S.M. Rubin. *Computer Aids to VLSI Design.* Addison-Wesley, Reading, MA, 1987.

[SBG89] Todd Simpson, Graham Birtwistle, and Brian Graham. Verifying Systems: Lispkit and the SECD. to be published in proceedings of the third annual Banff Workshop on Hardware Verification, 1989.

[SBGH89] 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.