1 BACKGROUND

We are designing and specifying, verifying, and building a substantial design in silicon as part of a project entitled "Specification driven design". The objectives of the project are:

1. to gain experience with the methodology;
2. to implement prototype CAD tools which will produce gate array and standard cell chips from HOL specifications, and to design transformation algorithms from specifications in HOL to standard VLSI tools (multi-level simulators, Mossim, floor-planners, etc);
3. establish a chrestomathy of cell and sub-system specifications; and
4. produce a viable chip.

Our long term goals are aimed at a specification based CAD system which will work mainly through specifications, support design iteration, and facilitate the construction of a provenly consistent specification tree from which custom chips can be laid out automatically. Our current work is a first step in this direction.

Initially we chose to work with Landin's SECD architecture because it is well-described and well-studied and reasonably attractive for eager functional languages. Our starting points were the Lispkit compiler and SECD architecture documented in Henderson's 1980 Prentice Hall text. Initial work was completed in 1985 by JJ, who first built the Lispkit compiler and SECD interpreter described by Henderson, and then refined the interpreter down to the bus access level. Then a microcode was specified, and model was further refined to interpret microcode instructions. These software models not proven correct, but gave consistent results when running a wide range of test programs, the largest of which was the compilation of the Lispkit compiler which occupies page 340 of Henderson (the last right parenthesis is missing!). Our prototype SECD machine has no I/O instructions. Its intended mode of operation is to poll until there is a task to carry out. When the task code has been placed in its memory by an outside agency, the machine will cease polling and run the program. When the computation is over, the machine signals completion, and returns to polling mode.

Several programs have been run on all the software versions of the SECD machine. Instruction frequencies for a compilation of the Lispkit compiler are:

LD	=	12839	LDC	=	7828	LDF	=	1080
AP	=	2041	RTN	=	2298	DUM	=	257
RAP	=	257	SEL	=	5456	JOIN	=	5456
CAR	=	5011	CDR	=	1423	ATOM	=	502
CONS	=	6771	EQ	=	4698	ADD	=	226
SUB	=	0	MUL	=	0	DIV	=	0
REM	=	0	LEQ	=	0	STO	=	1

Declarations are introduced in Lispkit only via **let** and **letrec** instructions. In

let x1 = e1, x2 = e2, ..., xn = en **in** E the several new quantities (the x's) being defined are initialised to the values of their associated e's before E is evaluated. Letrec is similar except that the e expressions may refer to the x's being defined. In this case, Henderson constrains the e's to be lambda expressions. The evaluation of an e associates its closure with the corresponding x.

Each time a let or a letrec is executed, a new list of x definitions is created and added to the environment. On exiting from the let or letrec, the definition list is discarded. x definitions are referenced as a dotted pair (m.n) which is interpreted as "go back m levels of list in the environment and access the nth definition in that list". Compilation of the Lispkit compiler turns out to be a fairly Lispkit typical program - it averages 0.21 for m and 0.55 for n. Further 80% of all references are to the current let or letrec block, and 8% of all references are to "main". Thus the run-time look-up organisation matches this program very well, but it may be worth while adding in a permanent pointer to the environment list of main at which 8% of accesses are directed.

JJ completed informal specification work on SECD (the top level and the microcode) before leaving to enter the PhD program at Cambridge. The informal hierarchical specification of SECD in Mossim is complete down to the transistor level. Its subsystems have been checked out individually, and we have run a short but tricky Lispkit program (3 mutually recursive functions) through the complete Mossim description.

2 TOP-LEVEL DESCRIPTION

The SECD machine was devised by Landin in order to explicate the operational semantics of programming languages. The machine may be characterised by four status registers, Store, Environment, Control, and Dump.

1. A stack **S** hold the expression under evaluation. S is flushed each time a function is called or exited.

2. **E** points to a stack of currently accessible definitions. E is reset on function entry and exit.

3. The control register **C** references the next instruction to be executed (code is generated in the form of a list). C is usually incremented past the current SECD instruction, but not in the case of function calls or IFs.

4. **D** saves the state of the machine (the current S, E, and C values) on function entry, and the code 'rejoin' point upon THEN or ELSE entry, and restores them on exit.

Henderson specifies the operation of the machine in terms of (S, E, C, D) transitions. We explain a few of them adopting a notational device which highlights the current instruction and uses •.c in the control slot to represent the current instruction.

- **LDC x** pushes the in-line constant x on top of the stack.

$$s\ e\ \bullet.c\ d \rightarrow x.s\ e\ c\ d$$

- **LD (m.n)** examines offset *n* within environment *m* back and copies its current value onto the stack.

  s e •.c d → (lookup m n).s e c d

- Non-recursive function calls, e.g. $(\lambda$xy.E)ab use LDF, AP, and RTN. First we place the evaluated parameters (a and b) on top of S. Then the called environment e' and the function entry point c' are placed on S by LDF c' where

- **LDF c'**

  s e •.c d → (c'.e).s e c d

  Finally the function is entered via a call on AP(ply) which constructs a closure from the top-of-stack entries.

- **AP**

  (c'.e' v.s) e •.c d → nil (v.e') c' (s e c.d)

  Notice that AP saves the calling state on D, flushes S, extends the environment of the function by (in our case) copying initial values for x and y, and sets C to the entry point. Function exit is effected by planting RTN which saves the state stored on D.

- **RTN**

  x e' •.c (s e c.d) → (x.s) e c d

- Recursive calls (**letrec**) are rather more tricky. The evaluation of the parameters is delayed until space for all the recursive definitions has been allotted. This is implemented using *replcar* to construct a circular lookup list. A special instruction DUM is needed to 'get things going' on entry.

- **DUM**

  s e •.c d → s (nil.e) c d

- **RAP**

  (c'.e' v.s) (nil.e) •.c d → nil (replcar v e') c' (s e c.d)

  The remaining transitions are obvious.

2.1 From spec to layout

One of the research lines we are following is to see how much of the architecture can be inferred from the specification. Consider **AP**. Rearranging it line by line (as suggested by Henderson), we get

Stack	**Before**	**After**
S:	(c'.e') v.s	nil
E:	e	v.e'
C:	**AP**.c	c'
D:	d	(s e c.d)

A little pattern matching enables us to derive

```
D  :=  cons (tl tl s , (cons e , (cons tl c , d)));;
C  :=  hd hd s;;
E  :=  cons ( hd tl s , tl hd s );;
S  :=  nil;;
```

subject to observing the precedence rules d < c, e < s to prevent overwriting. Thus a high level SECD interpreter can be obtained merely by transforming the specifications. Continuing on in this way, we observe that hd, tl, and cons are frequent operations and will build-in special units to perform them. Assuming a single bus structure, we now move down one level, and derive the following for the E and S assignments:

```
e := cons(hd tl s, tl hd s);
 bus (x2 := s);
 hd (x2);
 tl (x2);
 bus (x1 := s);
 tl (x1);
 hd (x1);
 Consx1x2;
 bus (e := mar);

s := nil;
 bus (s := nil);
```

from which x1, x2, and nil suggest themselves as extra registers and at one level down, we then derive mechanically

```
   E:
   rs     wx2   % bus (x2 := s)
  rx2    wmar   % hd (x2)
 rcar     wx2
  rx2    wmar   % tl (x2)
rmem      wx2
   rs     wx1   % bus (x1 := s)
  rx1    wmar   % tl (x1);
rmem      wx1
  rx1    wmar   % hd (x1);
 rcar     wx1
call (Consx1x2,$)
rmar       we   % bus (e := mar);

   S:
 rnil      ws   % bus (s := nil)
```

3 THE FLOOR PLAN ELEMENTS

The SECD chip can be broken up into 4 functional units: the control unit, the shift registers, the datapath, and the pad frame. We introduce them one at a time.

- 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: asynchronous reset and interrupt button signals, status flags, and a nine bit opcode (the code of the current machine instruction). Outputs include read and write signals for registers within the datapath and also for memory.
- The *shift register block* provides a (rudimentary) means of entering test vectors and examining the state of the chip. In the absence of "designed testability", this gives us a passable ability to test the chip in operation, and furthermore, permits independent testing of the datapath and control unit components. Most signals passing between the control unit and datapath are routed through the shift registers. These include read and write signals, alu signals, and test flags. For observability and controllability, it also routes some signals which are functionally internal to the control unit (specifically the 5 select signals from the PLA and the mpc register contents). Thus all these signals may be read and/or altered. In normal operation, signals pass through uninterrupted, with no added clock cycles required. To access or alter any value, the system clock is halted, and the shift register controls configured for the desired operation (shift in or out). Then, pulse the independent shift register clocks while either reading the output or providing the desired test input. A similar dual non overlapping clock strategy as is used for the system clock.
- The *datapath unit* executes simple micro-operations one at a time when 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, hd, tl, the arithmetic operations of addition, subtraction, decrementation, and setting status flags. Each SECD machine instruction is implemented by a number of micro read/write cycles each of which places a register value on the bus, and stores it somewhere else. The control unit initiates the next desired operation by setting the appropriate signal lines (one or more) high;; values to be placed onto the bus are controlled by signals beginning with 'r', signals beginning with 'w' control where bus values are to be stored. Besides read and write lines and the arithmetic operator select lines, the only other input control line is the clock (phiB). Outputs include the select flags, the memory address register (for the off chip RAM).

- The chip is framed by a set of bidirectional *I/O pads* which connect the chip to the outside world. The bidirectionality constraint was imposed by the limited number of pads allowed on the intended fabrication process.

3.1 TIMING AND STATE REGISTERS

A two-phase non-overlapping clocking scheme is used (PhiA, PhiB). All latches (ie. registers) are 'level triggered', so the state changes as the clock signal rises, but the latched state will be the value of the input when the clock signal falls. Static logic is used throughout the chip.

The control unit uses two registers, MPC and NEXTMPC, with some combinational logic in between. NEXTMPC loads on PhiB with the address of the next microcode instruction. This address transfers to the MPC register during PhiA. Embedded in the micro-code controller is a 4 deep stack used to implement subroutine calls in the microcode. This stack uses the same pattern of paired registers, with the next contents of each stack register loading during PhiB and transferring to the stack register during PhiA.

The choice of two-phase clocking arises from the need to not only buffer the next state from the present state (achieved by using paired registers), but also to prevent random writes to the data path registers during state transition. The latter concern is met by AND'ing all write signals in the datapath with a clock (PhiB) which does not overlap the clock which controls the change of state (PhiA). The use of PhiB to clock the buffering NEXTMPC register is possible since the inputs from the datapath (status flags) which are determined by the contents of datapath registers written during PhiB, will settle well before the end of PhiB. While this does restrict the maximum clock rate, the strategy has been verified both by simulation and fabrication of a previously implemented microprocessor.

3.2 DESIGN OF THE CONTROL UNIT

The control unit implementation is divided into seven functional blocks: a ROM, 3 decoders, a PLA, and a small datapath. The ROM contains a microcode program for the control unit. The three decoders produce discrete read, write, and alu signals from the encoded ROM output. The PLA determines which of four possible next addresses to select for the micro pc. Finally, a micro pc datapath (mpc-dp) contains the registers (for both MPC and NEXTMPC, as well as the stack registers) and logic to implement the selection of the next microinstruction.

Microcode Design

The microcode is designed to implement the top level state transitions that define the SECD machine. A sequence of register transfers, memory fetches and writes, and alu operations were defined separately for each machine instruction. A jumptable indexed by 'opcode' is used to enter the microcode segment appropriate for each machine instruction. The jump table is located at the microcode addresses that correspond to the numerical values of the machine instructions (1 through 23). At the end of each segment, an unconditional jump returns control to a "ready" position, to begin the cycle for the next machine instruction. A subroutine mechanism is used to permit sharing of common blocks of code among several different machine instruction code segments. Microcoded garbage collection also uses this subroutine mechanism. The maximum depth of subroutine calls is four.

The microcode instructions required several distinct components, namely:

- *test signal* – to determine how to choose the next microinstruction.
- *address field* – the address to use when the next microinstruction is not sequential
- *read signals* – to determine which register contents are placed on the bus.
- *write signals* – to determine where to store the bus value. alu signals - to select the appropriate alu operation.

In most of the microcode, the next instruction to be executed is usually located in the following memory location. Thus the next microinstruction address is usually obtained by incrementing the current micro pc. Additionally, there are 8 conditional jump instructions (one for each of the 7 condition flags and the button input), an unconditional jump, a subroutine call (just an unconditional jump concurrent with pushing the incremented current micro pc on the stack as the return address), a return (pop the micro pc from the top of the stack), and a jump opcode instruction, that implements the jump table for the set of machine instructions.

In total, there are 13 possible means of determining the next micro pc. This information is coded into 4 bits in the microinstruction. The address field requires 9 bits. The 12 alu operations (excluding no-op) require 4 bits, and 23 read signals and 17 write signals (both excluding no-op's) require 5 bits each. Thus the total microinstruction word length is 27 bits. Various methods of optimized encoding were examined, but further encoding of the microcode did not produce sufficient saving in space to warrant the added complexity.

Microcode ROM

In designing the ROM, several possible optimisations were considered in attempts to exploit microcode characteristics. The relative sparseness of the address and alu signal fields (about one in four microinstructions use these fields) suggested a separate ROM for these fields. (The random distribution of these microinstructions in the microcode produces a structure conceptually closer to a PLA in organisation.) However significant savings in area would accrue only by sharing the decode units, and a full 9 x 400 row decoder proved too large in one dimension for reasonable operation and for the predicted chip dimensions. This motivated a single interleaved ROM design with column decoding.

The required 9 x 400 x 27 ROM is implemented with a 7 x 100 row decoder and a 2-bit column decode. This configuration results in a nearly square unit which accorded well with other constraints. The decoder is a fully complementary CMOS design, but the 'OR' plane is implemented in a pseudo NMOS style, using pullup transistors for each column, and only ntran devices. The output of the ROM is actually inverted, and thus banks of inverters serve to both get the logic level right and buffer the output.

The Micro PC Datapath (mpc-dp)

This small datapath loads the next microinstruction address into the MPC rgister. It consists of the MPC and NEXTMPC registers, the stack mechanism implemented by the use of 8 more registers and several transmission gates, an incrementation circuit and four sets of transmission gates to gate each of the possible next instruction addresses into NEXTMPC. The datapath is nine bits wide, and also contains a row of random logic, to produce and buffer the required control signals.

The mpc-dp must to compute a new value for the NEXTMPC register prior to the fall of PhiB, and similarly for each 'next' register in the four deep stack. In the case of the NEXTMPC, this is calculated by selecting one of four possible values: the 'A' address field output from the ROM, the current value of the MPC imcremented by 1, the opcode signal supplied by the data path, or the value on top of the stack.

The new values for the 'next' stack registers are also selected from one of several values, including the lower and higher stack register contents, the current register contents, and for the top of stack register, the current MPC increment by one (the return address from a subroutine call at the current MPC address). To reduce the number of transmission gates required, the 'next' registers are clocked with a modified clock signal, so that they only change state when the current instruction is a subroutine call or a return. Thus only two possible inputs need be gated to each register. This raises a possible inconsistency with the previously discussed clocking strategy, in that the inputs to select the 'next' register contents are not required to settle until some time during PhiB. However, the signals for push and pop are not a function of the datapath status flags, and thus will settle BEFORE PhiB. Therefore the clock signal for the stack 'next' registers is PhiB $\wedge$ (PUSH $\vee$ POP).

Decoders

All 3 signal decoders use the same design as the decode plane of the ROM. It is a full complementary CMOS design, and in each case an encoded signal from the ROM is decoded to the required number of discrete signals for use by the datapath. The number of signals is required to be a multiple of 4 (in the automated decoder generator), hence there are unused outputs in 2 of the 3 decoders. The read decoder produces 23 signals, the write decoder produces 17, and the alu decoder 12.

Select PLA

The select PLA determines which of the 4 possible next address fields will be selected. Inputs include the 4-bit test field output by the ROM, 7 condition flags and the interrupt button. The output includes enable signals for the 4 transmission gates used to gate input to the NEXTMPC register, namely selA, selOp, selNxt, and pop. These correspond to the ROM address field, the opcode, the current MPC + 1, and the top of stack. Additionally, a push signal is used for a subroutine call micro instruction, although the NEXTMPC is loaded with the address field from the ROM, while the MPC + 1 address (return address) is pushed. Four of the signals (all except selNxt) are produced as direct outputs of the 12 x 12 x 4 PLA. The remaining signal is the NOR of those 4. Additionally, this unit contains the logic to AND the write memory signal with PhiB, for use by the datapath to control the bidirectional I/O pads.

Accessible Signals

The view presented of the control unit has ignored the interconnection between the control unit and the shift registers. Several inputs, outputs and internal signals are in fact routed through the bank of shift registers to permit examination and alteration of these values for test purposes. For example:

- all read, write and alu signals originating in the control unit (with the exception of wmem).
- the condition flags originating in the datapath.
- the contents of the MPC are routed through the shift registers in advance of reaching the ROM input.
- the 5 signals from the PLA are routed through the shift registers before reaching the mpc-dp.

3.3 DESIGN OF THE SHIFTERS

The shift register cell consists of a pair of latches and 2 MUX's. The latches are similar in operation to the control unit registers, with the first latching on sr-PhiA and the second latching on sr-PhiB. The input signal from the chip and the contents of the previous cell are MUX'ed by the shift control signal to the first latch input. The output of the second latch and the input signal from the chip are MUX'ed by the test control signal as the cell output. Thus the latches permit the following modes:

1. In normal chip operation, the test control signal is not asserted, so signals pass unaltered through the shift registers.
2. When reading the present state of the machine, with the system clock stopped, the shift and test control signals are asserted. Pulsing the shift register clock will produce a stream of bit values on the shift register output pin.

3. When entering a test vector, again the system clock is stopped, the shift and test control signals are asserted, and the test vector bit sequence is presented in sequence to the shiftregister input pin, as the shift register clock is pulsed. Once the test vector loading is complete, the system clock may again be operated, and the test control signal deasserted after one cycle.

The sr-shiftcell used in the actual chip layout actually contains 2 of the shift register cells described above. Additionally, there are 9 pass through lines located in a block within the group of shift registers. These pass throughs are for the low 9 bits of the arg register of the datapath, which form the opcode input to the control unit.

3.4 DESIGN OF THE DATAPATH UNIT.

The datapath is organised as a number of functional blocks communicating via a bidirectional bus. The functions of the individual units and their interplay were fixed after extensive simulations at various architectural levels (instruction, datapath, and microcode). Normally datapath operations are controlled through the read and write signals sent from the control unit, but when the chip is operating in test-mode control signals are routed via the shifters.

SECD words (and the bus) are 32 bits wide. Bits 31 and 30 are reserved for use by the control unit during garbage collection and may only be set by the control unit. Bits 29 and 28 are tag bits used to identify the type of the word occupying bits 27 through 0 as a cons'ed symbol, or an atom (either a symbol or a numeric item). If the type is cons, then the lower 28 bits are treated as two 14 bit HD and TL pointers. If the type is atomic, then the lower order bits contain either a symbol identifier or an integer value. A symbol is identified by a (unique) look-up index. An integer key is used, since the only operation allowed on symbols is comparison for equality. Numbers are represented in 2's complement format.

The functional sub-units of the datapath are: alu, clearunit, consunit, flagsunit, regs-14-hd, regs-14-misc, regs-14-y2, regs-32-arg, regs-32-bufs, which are not detailed. Arithmetic operations affect only the bottom 28 bits. Pointer operations use 14 bits (either the HD or the TL field) and map onto bus locations 0 to 13. Other operations affect even smaller fields, perhaps only specific bits. The datapath unit was constructed using a bit slice approach.

3.5 IO PADS

Between the datapath and the pads sits rmem – 32 busgates that allow the input values fromn the bidirectional pads to be passed onto the bus when the rmem signal is high. The bidirectional pads are write-enabled and default to act as inputs. Rmem prevents the bidirectional io pads from always trying to write values onto the datapath bus.

SUMMARY

We have succeeded in getting a chip fabricated in custom CMOS. In parallel we have developed a formal specification and verification of our design (from the gate level and upwards). This work will be used as further develop a translator which takes a HOL proof to a net list for input to gate array tools.

ACKNOWLEDGEMENTS

We gratefully acknowledge support for our research from the Natural Sciences and Engineering Research Council of Canada and from the Canadian Microelectronics Corporation. We are also grateful to MOSIS who built the chip for their speedy, efficient and courteous service.