TECHNICAL REPORT NO. 290

Derivation of an SECD Machine: Experience with a Transformational Approach to Synthesis

by

Robert M. Wehrmeister

September, 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Derivation of an SECD Machine: Experience with a Transformational Approach to Synthesis*

Robert M. Wehrmeister
VLSI Manager
Computer Science Department
Indiana University
Bloomington, Indiana

ABSTRACT

This project represents a continuing effort to investigate hardware design in an algebraic framework. A transformation system for digital design derivation (DDD) is used to derive the control and architecture of a computer based on the SECD machine of Landin[10] and Henderson[3]. DDD is an editor used to apply a sequence of correctness preserving transformations to a high level design description. It allows the designer to conceptualize, refine and debug the design at a high level, while maintaining confidence in the low level descriptions.

Several criteria were used to choose a suitable design. It had to be well established in the literature. It had to be simple enough to be prototyped using field programmable hardware. It had to present new problems for the transformation system and the derivation oriented approach. The SECD machine challenges DDD because it contains two cooperating processes: the CPU and the garbage collector (GC), the processes share the same memory object, and the memory object is viewed differently by each process. This paper describes the transformation process used to implement the SECD machine and discusses the exposed capabilities and limitations of DDD.

*Research reported herein was supported, in part, by the National Science Foundation under grants numbered MIP 87-07067 and DCR 85-21497.

1. INTRODUCTION

This project represents a continuing effort to investigate hardware design in an algebraic framework. A transformation system for digital design derivation (DDD) is used to derive the control and architecture of a computer based on the SECD machine of Landin[10] and Henderson[3]. DDD is an editor used to apply a sequence of correctness preserving transformations to a high level design description. It allows the designer to conceptualize, refine and debug the design at a high level, while maintaining confidence in the low level descriptions.

Several criteria were used to choose a suitable design. It had to be well established in the literature. It had to be simple enough to be prototyped using field programmable hardware. It had to present new problems for the transformation system and the derivation oriented approach. The SECD machine challenges DDD because it contains two cooperating processes: the CPU and the garbage collector (GC), the processes share the same memory object, and the memory object is viewed differently by each process. This paper describes the transformation process used to implement the SECD machine and discusses the exposed capabilities and limitations of DDD.

This paper does not describe the SECD or DDD in great detail. It does describe how DDD was used to transform the SECD machine into a hardware implementation. For more details about DDD and the transformational approach to synthesis consult [2,4,5,6,7,8,9]. For more details about SECD consult Landin [10] or Henderson [3].

2. SECD

The SECD machine was first described by Landin as an abstract machine for the mechanical evaluation of expressions[10]. Henderson later gave a concrete description of the SECD machine and a Lisp compiler for it [3]. Figure 1 contains the initial specification of the SECD machine. This is a direct translation of Henderson's description into the Scheme programming language[13]. The machine consists of four registers, each of which represent a list structure. They are Stack (s), Environment (e), Control (c) and Dump (d).

There are several differences between the machine described by Henderson and the machine implemented here. The multiply and divide instructions were omitted for the sake of simplicity. Several instructions and types were added to facilitate the implementation of I/O and a Read-Evaluate-Print loop. The type **char** was added to allow for the implementation of symbols as lists of characters. The instructions **SL** (symbol->list), **LS** (list->symbol) and **CI** (char->integer) were added to allow for the manipulation and creation of symbols. The instructions **RECH** (readchar) and **WRCH** (writechar) were added for I/O support. The instructions **NUM**, **SYM**, and **PAIR**, were added so that the various types could be distinguished for printing. The instruction **EXEC** was added so that code that was generated by a compiler could be executed. The instruction **POP** was added so that a sequence of expressions could be executed with only the value of the last expression remaining on the stack. The instruction **SET** was added to facilitate the implementation of the Read-Evaluate-Print loop.

```
(define secd
  (lambda (exp)
    (letrec (
      (exec
        (lambda (s e c d)
          (case (car c)
            (RTN    (exec (cons (car s)(car d))(cadr d)(caddr d)(cdddr d)))
            (DUM    (exec s (cons () e) (cdr c) d))
            (AP     (exec nil (cons (cadr s)(cdar s))
                       (caar s)(cons (cddr s)(cons e (cons (cdr c) d)))))
            (RAP    (exec nil
                       (set-car! (cdar s)(cadr s))
                       (caar s)
                       (cons (cddr s)(cons (cdr e)(cons (cdr c) d)))))
            (SEL    (exec (cdr s)
                       e
                       (if (car s)(cadr c)(caddr c))
                       (cons (cdddr c) d)))
            (JOIN   (exec s e (car d) (cdr d)))
            (CAR    (exec (cons (caar s)(cdr s)) e (cdr c) d))
            (CDR    (exec (cons (cdar s)(cdr s)) e (cdr c) d))
            (CONS   (exec (cons (cons (car s)(cadr s))(cddr s)) e (cdr c) d))
            (LD     (exec (cons (locate (cadr c) e) s) e (cddr c) d))
            (LDC    (exec (cons (cadr c) s) e (cddr c) d))
            (LDF    (exec (cons (cons (cadr c) e) s) e (cddr c) d))
            (ATOM   (exec (cons (atom? (car s))(cdr s)) e (cdr c) d))
            (EQ     (exec (cons (eq? (car s)(cadr s))(cddr s)) e (cdr c) d))
            (LEQ    (exec (cons (<=? (car s)(cadr s))(cddr s)) e (cdr c) d))
            (ADD    (exec (cons (+ (car s)(cadr s))(cddr s)) e (cdr c) d))
            (SUB    (exec (cons (- (car s)(cadr s))(cddr s)) e (cdr c) d))
            (SL     (exec (cons (sym-list (car s))(cdr s)) e (cdr c) d))
            (LS     (exec (cons (list-sym (car s))(cdr s)) e (cdr c) d))
            (CI     (exec (cons (char2int (car s))(cdr s)) e (cdr c) d))
            (RECH   (exec (cons (readchar) s) e (cdr c) d))
            (WRCH   (writechar (car s))(exec s e (cdr c) d))
            (NUM    (exec (cons (num? (car s))(cdr s)) e (cdr c) d))
            (SYM    (exec (cons (sym? (car s))(cdr s)) e (cdr c) d))
            (PAIR   (exec (cons (pair? (car s))(cdr s)) e (cdr c) d))
            (EXEC   (exec (cons (cons (car s) e)(cdr s)) e (cdr c) d))
            (POP    (exec (cdr s) e (cdr c) d))
            (SET    (exec (cons (set (cadr c) e (car s)) (cdr s)) e (cddr c) d))
            (STOP   (exec s e c d))
            (else   (exec s e c d)))))
      (index (lambda (n s)
            (if (zero? n) (car s) (index (sub1 n) (cdr s)))))
      (locate (lambda (i e)
            (index (cdr i) (index (car i) e))))
      (set (lambda (i e v)
            (setnth (cdr i) v (index (car i) e))))
      (setnth (lambda(n v s)
            (if (zero? n) (setcar! s v) (setnth (sub1 n) v (cdr s))))))
    (exec nil nil exp nil))))
```

Figure 1. Initial Specification of the SECD Machine.

3. SECD MACHINE IMPLEMENTATION

Initially, the designer has only a sketchy view of the architecture. As the design progresses the structure becomes clearer until the design is finalized. DDD supports this top-down design methodology, allowing design decisions to be postponed as long as possible and allowing the exploration of many design options. The design of the SECD machine went through several iterations. For the sake of clarity, only the final version of the SECD architecture is described. Then the process by which it is derived is explained.

The SECD machine contains two cooperating processes, the CPU and the garbage collector (GC). The two processes communicate through shared signals and the shared memory object. Their implementations are derived separately but in close conjunction.

The CPU consists of six parts, the data path unit, instruction generator, state generator, memory unit, arithmetic/logic unit (ALU) and the serial interface. Of these, the data path unit, instruction generator and state generator are derived automatically from the high level specification. The serial interface, ALU and memory unit are all implemented by hand. Figure 2 is the block diagram of the CPU.

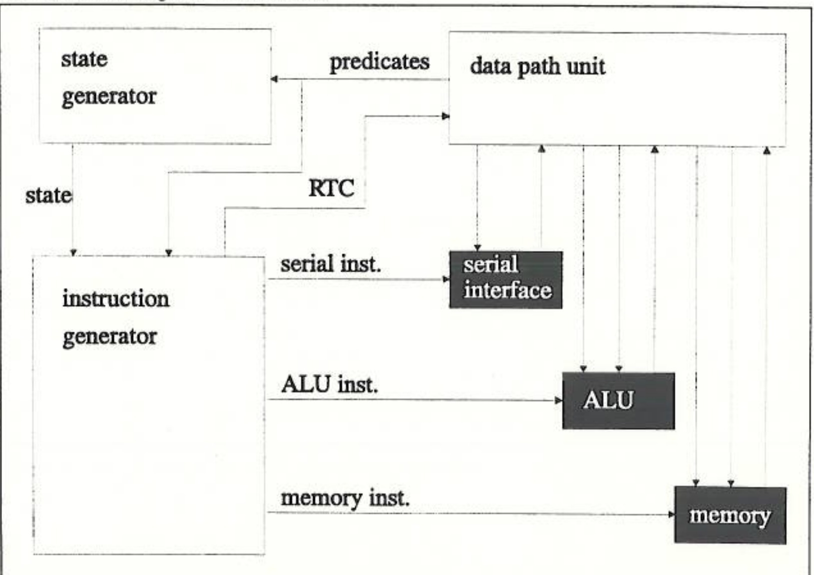


Figure 2. Structure of the SECD CPU.

The data path unit consists of the machine registers and their interconnections. It is within the data path unit that the register transfers occur. A register can take on the value of another register, an input, a constant or some function of these. The data path unit takes as an input the Register Transfer Code (RTC) which controls what each register does in each state. The RTC can be thought of as an instruction to the data path unit. The data path unit can also have combinational outputs which are controlled in the same manner as the registers but their values are not latched.

The instruction generator is a combinational circuit which takes as inputs the current state and a set of external predicates. It produces as outputs the instructions for the serial interface, ALU and memory unit, as well as the RTC.

The State generator provides the next state function. It calculates next state based on the current state and external predicates.

3.1 Memory

The CPU operates on a list based memory. The memory consists of list cells, comprised of two memory words, the **car** and **cdr** (head and tail.) Each memory word contains two fields, **tag** and **value**. The **tag** field of a memory word, describes the type of object that is contained in the **value** field. Included in the memory unit is a pointer to the next available list cell (**avail**.) This structure is shown in Figure 3. The various types of objects used by the CPU are shown in Table I.

Table I. Tag values for the SECD machine.

TAG	VALUE
Pointer	Address of a List Cell
Constant	True or Nil
Symbol	Address of a List Cell
Number	Two's Complement Integer
Character	Single ASCII Character
Forward Pointer	Address of a List Cell Used only by garb. coll.

The physical memory is word addressable: only one of the **car** or **cdr** fields can be accessed at a time. The abstract memory is cell addressable: the **value** field of pointers contain the addresses of cells. This difference in views of memory is resolved in the memory unit by arranging physical memory so that the **car** fields are in the even memory locations and the **cdr** fields are in the odd memory locations. The memory unit controls the low order bit of the physical address according to the memory operation and forms the higher bits of the address from the **value** field of the cell. This **cdr bit** in the physical address will be set to 1 when the operation is on the **cdr** field and 0 when on the **car** field. The instructions for the memory unit are listed in table II.

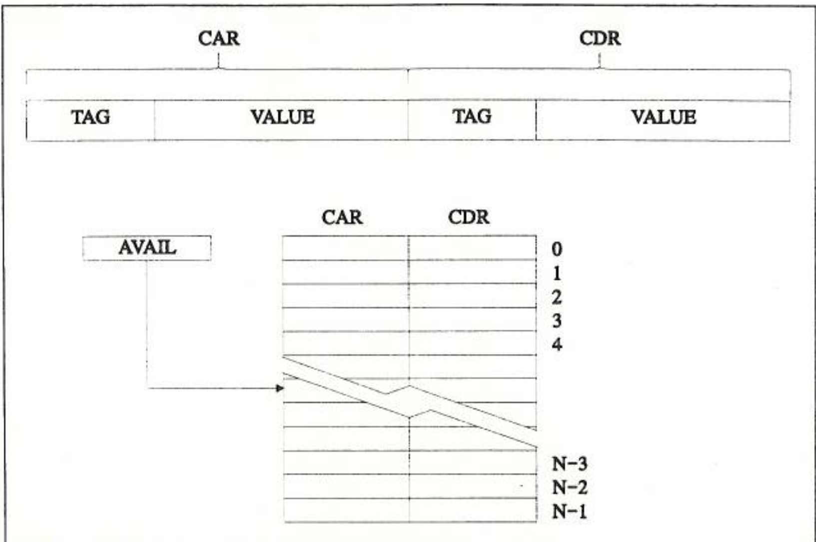


Figure 3. The Memory Structure.

Table II. Memory instructions for the SECD CPU.

Instruction	Address	CDR bit	Data	Buffer	Avail
Car	x	0	?	read(x*2)	Avail
Cdr	x	1	?	read(x*2 + 1)	Avail
Setcar!	x	0	y	write(x*2, y)	Avail
Setcdr!	x	1	y	write(x*2 + 1, y)	Avail
Alloc	?	?	?	Avail	Avail + 1
Meminit	?	?	?	?	read(0)

3.2 ALU

The ALU for the CPU performs the operations listed in table III. There are three kinds of operations. Those that have numeric arguments and produce a numeric result, those that have numeric arguments and produce a boolean (constant) result, and those that take arbitrary arguments and produce a boolean result.

Table III. ALU instructions for the SECD CPU.

Instruction	Inputs	Return Value	Type
Add	x y	x + y	Number
Sub	x y	x - y	Number
Sub1	x -	x - 1	Number
Eq	x y	x == y	Constant
Leq	x y	x <= y	Constant
Atom?	x -	atom? x	Constant
Number?	x -	number? x	Constant
Symbol?	x -	symbol? x	Constant
Pair?	x -	pair? x	Constant

There are two parts to the ALU, the value ALU and the tag ALU. It is implemented with standard ALU components (74ls181) and one Altera EP1800 PLD [1]. This PLD translates the instruction coming from the CPU into the proper control signals for the 74ls181, manipulates the tag bits and returns the result of the 74ls181 or a constant value.

3.3 Serial Interface

The serial interface has two instructions, **s-in** and **s-out**. When given the instruction **s-in**, the serial interface will return on its output the character currently on the input device. When given **s-out**, it will send the character on its input to the output device. The serial interface is implemented with a standard UART with the appropriate line drivers. All control signals come directly from the CPU

4. GARBAGE COLLECTOR

For this implementation of the SECD machine the **stop-and-copy** garbage collection algorithm was used. When there is not enough space for the machine to continue, the CPU stops and gives control to the GC, which reclaims the memory by coping the accessible data from one memory space to another. The GC also initializes the memory upon power-up or reset.

The structure of the GC is very similar to that of the CPU. It has six parts, the data path unit, state generator, memory unit, two counters and a read-only-memory (ROM). Of these, the data path unit and the state generator are derived directly from the high level specification. The counters, memory unit and ROM are all implemented by hand. Figure 4 is the block diagram of the GC.

The state generator serves the same function as in the CPU, with the addition of generating the RTC for the data path unit. The instructions for the memory unit, and counters are produced in the data path unit, so there is no need for a separate instruction generator.

The counters in the GC serve as pointers into the memory space. Each counter takes as inputs, an instruction and a value, and produces one output: the current value of the counter. The instructions are **load**, **inc** and **hold**.

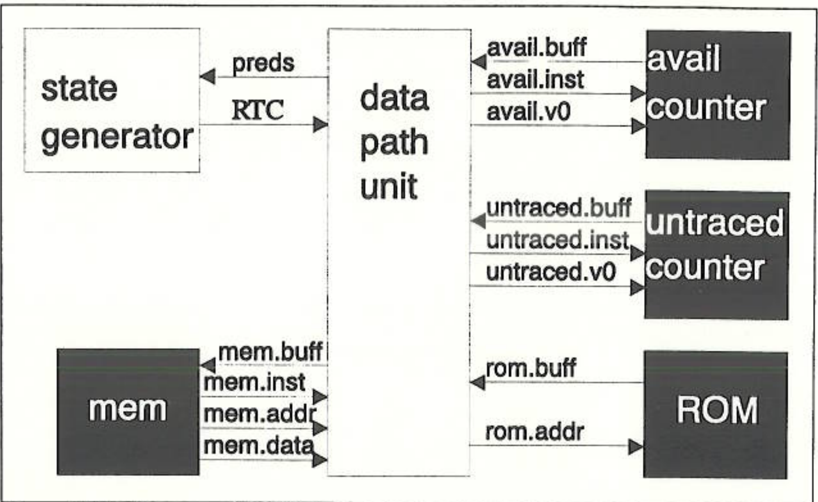


Figure 4. The Structure of the Garbage Collector.

The ROM is used to store the initial memory image. Upon power-up or reset, the CPU gives control to the GC. The GC copies the contents of the ROM to RAM and returns control to the CPU. The ROM is implemented with two 32K x 8 bit EPROMs (27128).

5. SHARED MEMORY

The GC shares the memory unit with the CPU, but it has a different view of it. The GC addresses the memory by words with the instructions **mem-read** and **mem-write**. The low order bit of the physical address is provided directly from the GC and not from within the memory unit. In addition, the GC manages two copies of the memory space. The collection process copies the heap from the **old** to the **new** memory space then reverses the roles of the two memory spaces. When the CPU has control of the memory it can only access the **new** memory space. These two spaces are implemented in one physical memory by assigning a bit of the physical address (space bit), under the control of the memory unit, to signify which memory space is to be used. Also, a new instruction port and a new instruction are added to manage the two memory spaces. The new instruction port is used to specify which memory space the instruction (on the other port) is to be applied. The instructions for this port are **new_addr** and **old_addr**. The new instruction **mem-switch** reverses the roles of the two memory spaces.

These opposing views of memory are resolved by layering two interfaces on top of the physical memory as illustrated in Figure 5. The CPU presents the memory instruction, address and data to the level 2 memory interface. The level 2 interface translates these to the level

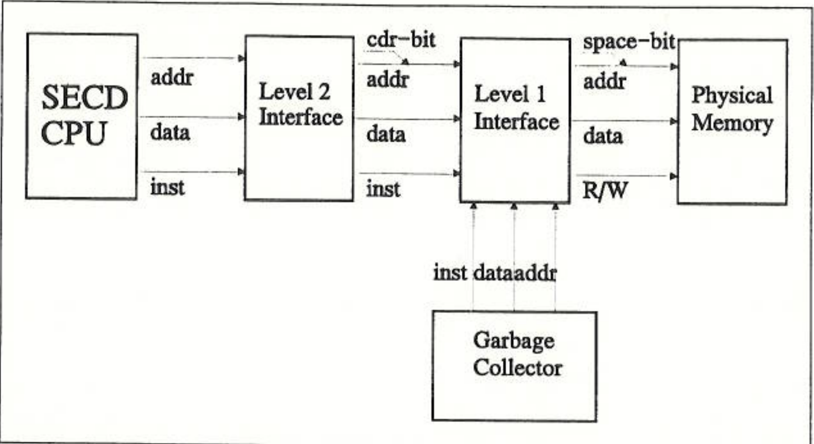


Figure 5. The Memory Interface.

1 format adding the appropriate value for the **cdr-bit**. The level 1 interface translates the inputs into the physical memory format adding the appropriate value for the memory space bit. When the GC has control of the memory, it presents the memory instruction, address and data directly to the level 1 interface, bypassing the level 2 interface altogether.

The memory unit was implemented using four 32K x 8 bit static RAM chips (TC55257), giving a total of 32K, 16 bit words (16K, 32 bit cells) for each of the two memory spaces. An additional 32K x 1 bit RAM was needed for the GC to mark words as collected. This was implemented with one 32K x 8 bit static RAM chip (TC55257). Two Altera EP1800 PLDs were used to implement the two views of memory and to maintain the **avail** pointer and the mapping of the two memory spaces.

6. DDD SOURCE SPECIFICATION

The code in Figure 1 represents the abstract SECD machine as described by Henderson[3]. There are several things which are not explicitly stated in this specification. The concept of state, the abstractions of memory, arithmetic units and other such things have to be made more explicit. The first step toward this goal is to formulate a specification in which each function call represents a state transition and the formal parameters represent the abstract registers of the machine. Figure 6 is an outline of the form required by DDD and Figure 7 is a partial listing of the CPU in this form. This form will be referred to as the source specification. The complete listing of the CPU is seen in Appendix A.


```
(define machine
  (letrec (
        (state-1 (lambda (reg1 reg2 reg3 ...)
            (next-state reg1' reg2' reg3' ...)))
        (state-2 (lambda (reg1 reg2 reg3 ...)
            (next-state reg1' reg2' reg3' ...)))
        .
        .
        .
        (state-n (lambda(reg1 reg2 reg3 ...)
            (next-state reg1' reg2' reg3' ...))))
      (lambda()(initial-state reg1.i reg2.i reg3.i ...))))
```

Figure 6. The DDD Source Specification Template.

```
(define secd
  (letrec (
        .
        .
        .
        (fetch (lambda (s e c d mem i j)
            (exec s e c d mem (car mem c) j)))
        (exec (lambda (s e c d mem i j)
            (case i
              (RTN     (rtn1 s e c d mem (car mem s) j))
              (DUM (dum1 s e (cdr mem c) d mem i j))
              (AP      (ap1 s e c d mem i (car mem s)))
              .
              .
              .
              )))
        .
        .
        .
        (ap5  (lambda (s e c d mem i j)
            (ap6 s e c d mem i (car mem j))))
        (ap6  (lambda (s e c d mem i j)
            (ap7 s i c d (setcar! mem i j) i e)))
        .
        .
        .
        )
      (lambda ()(idle 0 0 0 0 0 0 0 0 0 0))))
```

Figure 7. Partial Listing of the SECD CPU Source Specification. See Appendix A for a complete listing.

The transition from the abstract specification of Figure 1 to source specification of Figure 7 is not at all straight forward. The designer has to make some architectural decisions. For instance, the register set, the number of ALUs, and the memory interface have to be defined. Naturally, these decisions can be postponed until later or modified as the design progresses.

The register set of the CPU includes the s, e, c, and d, as well as the **mem**, **i**, and **j** registers. The additional registers **i** and **j** are accumulators which were deemed necessary to store intermediate results. The **mem** register represents the abstraction of memory.

The architecture of the machine imposes certain constraints on the implementation. With one memory and one ALU in the CPU, only one memory operation and/or one ALU operation can take place in any given state. The data path unit is general enough to allow multiple register transfers at a time. Other constraints may be added as the design progresses based on the technology used for implementation.

To meet these constraints the code is serialized, adding new registers as needed to store intermediate results. It was during this phase of the design that the **i** and **j** registers were introduced. The serialization was done entirely by hand. As an example of the process consider the SECD instruction **ldc**. The original code is:

```
(exec
   (lambda (s e c d)
      (case ...
         (LDC (exec (cons (cadr c) s) e (cddr c) d)) ...)))
```

expanding macros it becomes:

```
(exec
   (lambda (s e c d)
      (case ...
         (LDC (exec (cons (car (cdr c)) s) e (cdr (cdr c)) d)) ...)))
```

It now needs to be serialized so that there is at most one memory operation per state. It is clear how to do this for the **car** and **cdr** operations, but the **cons** operation does not correspond directly to any of the memory operations. It is actually a sequence of three

operations. First, a new cell is allocated (**alloc**) and then the **car** and **cdr** fields are set (**setcar!** and **setcdr!**.) The expansion for this instruction becomes:

```
(exec
   (lambda (s e c d mem i j)
      (case ...
         (LDC (ldc1 s e c d mem (cdr mem c) j)) ...)))

(ldc1
   (lambda (s e c d mem i j)
      (ldc2 s e (cdr mem i) d mem i j)))
(ldc2
   (lambda (s e c d mem i j)
      (ldc3 s e c d mem (car mem i) j)))
(ldc3
   (lambda (s e c d mem i j)
      (ldc4 s e c d mem i (alloc mem))))
(ldc4
   (lambda (s e c d mem i j)
      (ldc5 s e c d (setcar mem j i) i j)))
(ldc5
   (lambda (s e c d mem i j)
      (fetch j e c d (setcdr mem j s) i j)))
```

The source specification is executable as a Scheme program. This, along with the code for the abstractions of the memory, ALU and serial interface, as well as any other functions that are to be implemented as part of the data path unit make for a software simulator for the eventual machine. Because it is executable, it is easy to debug, gather statistics about and fine tune the specification and since it reflects in great detail the execution of the derived hardware, it is also useful as a tool when debugging the hardware.

A Lisp compiler for the SECD Machine was developed in Scheme, so that Lisp expressions could be compiled and executed in this environment. The same compiler was also used to create EPROMs for the initial memory image of the actual machine.

Lisp code for the TAK benchmark was compiled and executed on both the software and hardware versions of the machine. This was used to gather various statistics about the operation of the machine. The code for TAK is given in Appendix E.

Lisp code was developed for a Read-Evaluate-Print loop. This consists of a reader which parses s-expressions entered through the serial interface, a compiler which compiles the parsed expression and a printer which formats and outputs through the serial interface, the result of executing the compiled expression. The code for the Read-Evaluate-Print loop is given in Appendix E.

7. SECD CPU DERIVATION

Transformations are made to the specification by the application of Scheme functions. The complete derivation requires many individual transformations. I chose to group these transformations into script files. Each script performs some major step of the process. I also developed "makefile" so that after a change to the specification or a script file, I simply had to enter "make" to execute the proper script files. All the script files for the CPU derivation are in Appendix B and for the GC derivation Appendix D. Throughout the description of the derivations there are references to the page number of the script file associated with the transformation being discussed.

The first step in transforming the CPU (and in most other designs) is the separation of control and architecture. DDD transforms the source specification into a **selector** and a set of **stream equations** (one for each register,) representing the control and architecture respectively. Also the representation of state is transformed from a function call to a register (Appendix B1).

The **selector** is a function whose body is a skeleton of the original specification. Each transfer of control and predicate being replaced by a parameter to the **selector**. The parameters to the **selector** function are the **state** signal, a list of predicates and a list of values. A **stream equation** for a register is the application of the selector to the **state** signal, the predicates and the list of values that that register takes on for each transfer of control. The **selector** can be thought of schematically as a multiplexor, with its control coming from the **state** signals and predicates and the inputs being the various values that can be loaded into the register. Each stream equation is an instance of the multiplexor with its output feeding a register.

The stream equations at this point still contain operations on abstract objects such as the memory unit, serial interface and ALU. Abstract objects can be thought of as external devices which communicate with the system. Their interface must be specified, but their inner workings need not be specified. Typically, those functions which can easily be implemented with "off the shelf" hardware or that are too complicated to be implemented within the data path unit are extracted as abstract objects. DDD factors out the abstract objects, leaving only the connections necessary for them to be installed. The implementation of the object is then left to other means.

Operations on abstract objects are usually expressed as follows:

```
(operation object p0 p1 ...)
```

Where **operation** is one of a set of operations allowable for this type of object, **object** is the object being operated on and **p0**, **p1**, ... are the arguments to the abstract object. DDD factors these expressions out of the stream equations and generates stream equations representing the instruction and argument ports. See [5,6] for more details on factorizations.

For example, a read operation on memory looks like:

```
(j         <== (select state preds ... (car mem j) ... ))
```


This is factored by DDD to:

```
(j          <== (select state preds ... mem.buff ... ))
(mem.inst   <== (select state preds ... car      ... ))
(mem.addr   <== (select state preds ... j        ... ))
(mem.data   <== (select state preds ... ?        ... ))
```

There are now separate entries for all the connections to the memory object. **Mem.inst** being the instruction, **mem.addr** and **mem.data** the two input arguments and **mem.buff** the output of the memory object. The **mem.data** value is ? indicating a *don't-care* value, since a value for **mem.data** is not needed during a read operation.

A write operation,

```
(mem   <== (select state preds ... (setcar! mem i j)  ... ))
```

occurs in the **mem** stream equation, implying that the result is a new memory object as opposed to a data value. The write operation is not side-effecting the memory object, but generating a new object, identical to the original with the possible exception of the contents of the location written. Of course memory is implemented in the traditional way. The above expression is factored to:

```
(mem.inst    <== (select state preds ... setcar! ... ))
(mem.addr    <== (select state preds ... i       ... ))
(mem.data    <== (select state preds ... j       ... ))
```

The ALU is also an example of an abstract object. Although it is possible to implement each operation as a separate abstract object or as part of the data path unit, I elected to implement all of them with an "off-the-shelf" ALU. A typical ALU operation looks like:

```
(j          <== (select state preds ... (plus i j)  ... ))
```

This is factored by DDD to:

```
(j          <== (select state preds ... alu      ... ))
(alu.inst   <== (select state preds ... plus     ... ))
(alu.v0     <== (select state preds ... i        ... ))
(alu.v1     <== (select state preds ... j        ... ))
```

Where **alu** is the result of the ALU operation, **alu.inst** is the instruction signal and **alu.v0** and **alu.v1** are the input signals to the ALU.

This process of factoring out the abstract objects is done for each abstract object in the specification (Appendix B2). When done, the only expressions left in the stream equations are constants (eg. **car**, **cdr**, **setcar**), register or input values (eg. s, e, **mem.buff**), or simple expressions which can be implemented internal to the data path unit. These simple expressions might be bit field manipulations, maskings, conditionals or simple arithmetic operations. The

complexity of the operations allowable in the data path unit were determined by the target technology.

The stream equations at this point are suitable for implementation as a data path unit. There are, however, some things that can be done to simplify the data path unit. Some of the stream equations do not depend or depend very little on the other equations. These are probably more efficiently implemented separately. In the CPU this was done for the instruction equations (**mem.inst, ser.inst, alu.inst**) generated by the factorization of the abstract objects and the state equation (Appendix B3). These equations are independent of other signals. In general, determining how to group the signals is a hard problem. The designer relies on his expertise and experimentation to make these decisions.

The equations for the ALU inputs (**alu.v0, alu.v1**) each only require the value of one register, **i** and **j** respectively. Since the ALU inputs are combinational, they can be implemented by taking the signals directly from the outputs of the **i** and **j** registers. Therefore these equations were removed from the description (Appendix B3). It was no coincidence that this occurred. After several iterations of the derivation process, the source specification was modified by hand so that the two ALU inputs are always the **i** and **j** registers. In some cases this transformation required that new intermediate states be introduced.

Each stream equation is a separate instance of the selector, each separately decoding the state and predicates. This decoding can be factored out of all the stream equations and implemented by a single device. This requires that each selector be transformed to receive an encoding of the state and predicates as its control. This is likely to be much smaller since the product of the number of states and the number of predicates is usually much larger than the number of unique register transfer combinations.

7.1 Register Transfer Table

A Register Transfer Table (RTT) is constructed to visualize the stream equations and the data path unit (Appendix B3). The RTT represents the set of all register operations used by the data path unit and is generated by extracting the list of inputs from each stream equations into the columns of a table. Table IV is a partial listing of the RTT for the CPU before partitioning as discussed above.

The row number of an entry, representing the encoding of the state and predicates, is called the Register Transfer Code (RTC). When the data path unit is presented with an RTC the register transfers in the corresponding row of the RTT are performed. For instance, if an RTC of 3 is presented to the data path unit above, the serial instruction (**ser.inst**) and ALU instruction (**alu.inst**) get the constant value **noop**, the memory instruction (**mem.inst**) the constant value **car**, the memory address (**mem.addr**) the value of the **c** register and the **i** register the value of the memory buffer (**mem.buff**). All other registers either hold their current value or get an unknown (*don't-care*) value.

Since the RTC is encoded from the state and predicates, the rows of the RTT can be rearranged in any way, as long as the encoding function is modified to reflect the change. Redundant entries are removed and the table is ordered to facilitate implementation.

Table IV Partial RTT for SECD

RTC	ser. inst	ser. v0	mem. inst	mem. addr	s	e	c	d	mem. data	i	j	alu. inst	alu. v0	alu. v1
0	s-in	?	noop	?	s	e	c	d	?	ser. buff	j	noop	?	?
1	s-out	j	noop	?	s	e	c	d	?	i	j	noop	?	?
2	noop	?	alloc	?	s	e	c	d	?	mem. buff	j	noop	?	?
3	noop	?	car	c	s	e	c	d	?	mem. buff	j	noop	?	?
4	noop	?	setcar!	i	s	e	c	d	s	i	j	noop	?	?
5	noop	?	noop	?	j	e	c	d	?	i	j	noop	?	?
6	noop	?	noop	?	s	e	i	d	?	i	j	noop	?	?
7	noop	?	car	i	s	e	c	mem. buff	?	i	j	noop	?	?
8	noop	?	noop	?	s	e	c	d	?	alu	j	add	i	j

After several iterations of the derivation process, it was evident that the resulting specification was too complicated for the target technology. The target is Altera EP1800 PLDs and the target specification is boolean equations. The Altera PLDs have the property that each macro-cell (register) has a limit of eight or-terms. If more are needed they are cascaded, consuming more macro-cells. In order to reduce the number of macro-cells needed, the source specification was serialized further more. To understand the goals of this process, it is necessary to describe more of the derivation process. Therefore, this topic will be described later. After the manipulations to the RTT are complete, new stream equations are generated that reflect these changes along with a new encoding function (Appendix B4).

7.2 Projection and Representation

As they stand, each stream equation represents a full register or output signal. To develop boolean equations, stream equations must be generated for each bit. Each stream equation is projected onto bits, so that one equation representing an n-bit signal is expanded to n equations each representing a single bit of the signal (Appendix B5).

To accomplish this bitslicing, representation decisions must be made. The size of each register and the bit patterns for the constants and tag fields are decided upon. Also the meaning of all the remaining functions in the stream equations are defined in terms of bits. With this information it is possible to derive the boolean equations needed to implement the hardware.

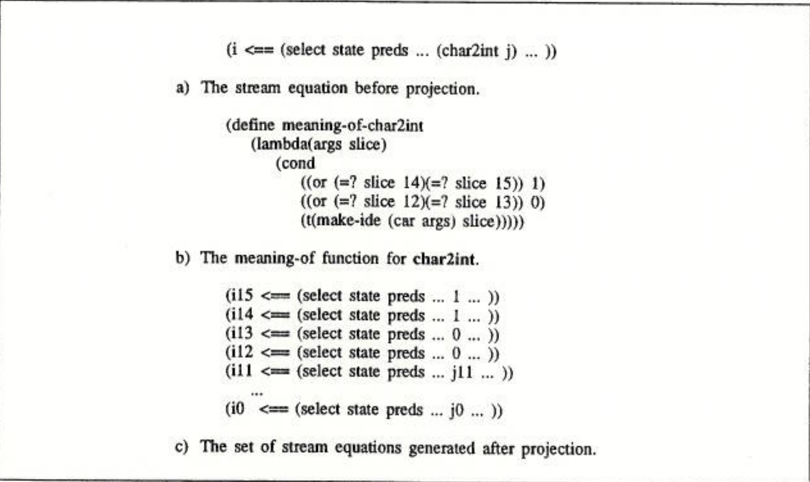
The representation information for each signal is given in a table as shown in Table V. Each column represents one bit of each signal. Each row represents the projection of one signal onto each bitslice.

The meanings of the remaining functions are defined with Scheme functions that take two arguments: **args** - the list of arguments to the function and **slice** - the number of the slice for which it is to be evaluated. As an example, the function **char2int** converts character types

Table V. Representation table for the SECD CPU.

signal	bitslice 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mem.addr	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x
s	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
e	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
c	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
d	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
mem.data	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
i	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
j	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
do_gc	-	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-
donesecd	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

x : signal exists in this bitslice
- : signal does not exists in this bitslice



```
(i <== (select state preds ... (char2int j) ... ))
```

a) The stream equation before projection.

```
(define meaning-of-char2int
   (lambda(args slice)
      (cond
         ((or (=? slice 14)(=? slice 15)) 1)
         ((or (=? slice 12)(=? slice 13)) 0)
         (t(make-ide (car args) slice)))))
```

b) The meaning-of function for **char2int**.

```
(i15 <== (select state preds ... 1 ... ))
(i14 <== (select state preds ... 1 ... ))
(i13 <== (select state preds ... 0 ... ))
(i12 <== (select state preds ... 0 ... ))
(i11 <== (select state preds ... j11 ... ))
...
(i0  <== (select state preds ... j0 ... ))
```

c) The set of stream equations generated after projection.

Figure 8. The Projection of **(char2int j)** onto Bitslices.

to integer types by changing the value of the tag bits of its argument. The function **meaning-of-char2int** is then used to evaluate the function for each slice. Figure 8 shows how the expression **(char2int j)** is projected onto bitslices.

The derivation of boolean equations from the stream equations is now straight forward. The equation for a register is derived directly from the corresponding column in the RTT by or-

ing together all the items in the column and-ed with their respective RTC. For example, the equation for the **i** register taken from the partial RTT above is:

```
i.d  = (RTC-0 * s-b)
     + (RTC-1 * i)
     + (RTC-2 * m-b)
     + (RTC-3 * m-b)
     + (RTC-4 * i)
     + (RTC-5 * i)
     + (RTC-6 * i)
     + (RTC-7 * i)
     + (RTC-8 * alu);
```

The value **i.d** represents the value **i** will take on upon the next state transition and is therefore input to a D-type register.

A different boolean equation can be generated if a Toggle register is used. When using Toggle registers, the equation formed represents a boolean indicating whether the current value is to change (toggle) or else remain the same. The equation for a column of the RTT in this case is composed of the terms:

RTC-i * (d ⊕ r)

where:
RTC-i is the Register Transfer Code for that row
d is the value to be loaded into the register
r is the current value of the register
⊕ is the exclusive-or function

The equation for the **i** register is now:

```
i.t  = (RTC-0 * (s-b ⊕ i))
     + (RTC-1 * (i ⊕ i))
     + (RTC-2 * (m-b ⊕ i))
     + (RTC-3 * (m-b ⊕ i))
     + (RTC-4 * (i ⊕ i))
     + (RTC-5 * (i ⊕ i))
     + (RTC-6 * (i ⊕ i))
     + (RTC-7 * (i ⊕ i))
     + (RTC-8 * (alu ⊕ i));
```

The terms which have the sub-term $(i \oplus i)$ can be eliminated since this sub-term is always zero. The equation for **i.t** is then:

$$
\begin{aligned}
\text{i.t} \; &= (\text{RTC-1} * (\text{s-b} \oplus \text{i})) \\
&+ (\text{RTC-3} * (\text{m-b} \oplus \text{i})) \\
&+ (\text{RTC-4} * (\text{m-b} \oplus \text{i})) \\
&+ (\text{RTC-9} * (\text{alu} \oplus \text{i}));
\end{aligned}
$$

This is now eight or-terms (two for each exclusive-or), one better than **i.d**. In general, the toggle form will be better if the register tends to hold its value and the D form will be better when it tends to change often. For further discussion of this technique, see Winkel [15,16]. DDD generates both forms of the equations for each signal and uses the best one in each case (Appendix B6, B7). This turned out to be the toggle type for all signals except **mem.addr**, **mem.data**, **i**, **do_gc** and **donesecd**. For combinational signals, the D form must be used.

7.3 Serialization

The serialization mentioned above was done to reduce the size of the resulting boolean equations. For any given column in the RTT the rows can be rearranged so that unique values are grouped together. This can allow the boolean equations to be reduced by introducing *don't-care* values into the RTC. Furthermore, buffer rows can be added to fill out the groupings so they will start and finish on power of two boundaries. Each column cannot be rearranged independently. The ordering of one column will affect the ordering of other columns.

The goal of the serialization is to minimize the interdependence between columns by limiting the number of transfers allowed in each state. This is clearly a speed (parallel transfers), size tradeoff. The CPU specification was serialized with the added constraint that at most one register to register transfer and either one memory operation, ALU operation or serial operation be allowed per state.

The resulting RTT was then sorted (by hand) according to the following procedure:

1) Find the column that has the largest number of items of a single value (other than itself).

2) Arrange the rows of the table into two groups: one with rows that contain the item in the column and the other with rows that do not.

3) Perform this procedure on each group (if it has more than one row) ignoring the columns that have already been grouped.

The initial unconstrained version was 161 states. The serial version was expanded to 204 states. The total size of all equations generated to implement the RTT for the unconstrained version was 2221 or-terms and for the serial version 1012 or-terms.

7.4 Live/Dead Analyses

The source specification was analyzed for dead registers. A register is considered dead in the current state, if in all possible next states, it is not used and is either set or dead. The analyses starts at a state in which the liveness of the register in question is known, typically the initial state. If the register is dead in the current state, then it is dead in each state along the execution path up to but not including the state in which it is next set. If the register is live, it is live in each state along the execution path up to but not including the last state in which it is used before it is next set. It is dead from this state up to but not including the state in which it is set. The analyses then continues from the state where the register is set. If a branch is encountered, each path is followed. For a register to be dead in the state of the branch it must meet the criteria for each branch.

All registers are given the *don't-care* value in each state in which they are dead. This allows the *don't-care* values to be propagated into the boolean equations and used during minimization. It is tempting to merge rows of the RTT which are identical except for occurrences of *don't-cares*. This will counter the serialization that was done. Rows are eliminated if they are strictly more general than an already existing row.

The live/dead analysis on the serialized CPU specification led to a set of boolean equations for the data path unit with a total of 1012 or-terms. Without the live/dead analysis the resulting boolean equations had a total of 1060 or-terms.

Table VI gives a summary of the effects of the various optimizations. The final version has a 51% improvement in size with a cost of 31% more cycles based on the TAK benchmark.

All equations are reduced using espresso [14] , split into groups that will fit onto one chip and transduced to the Altera ADF format [1] (Appendix B8-B9). The grouping of the equations is a trial and error process. To determine if a particular grouping fits onto one chip, the equations are compiled with the Altera Aplus compiler [1]. If it fails to fit, another grouping can be tried, or the Altera specification can be modified in an attempt to find a fit. The final version of the CPU data path unit fit onto four Altera EP1800 PLDs, with four bitslices per part.

7.5 Control

The state generator is implemented by a partial evaluation of the **state** stream equation, creating a function that generates the next state based on the current state and predicates. This partial evaluation can be done because the **state** signal takes on only constant values that do not require "run time" evaluation. DDD also creates a mapping of state names to bit patterns. Boolean equations are generated from the next state function for each bit of the state register. This required eight bits for the 204 states of the CPU. Each equation is reduced using espresso and transduced to the Altera ADF format. The state generator was implemented with one Altera EP1800 PLD (Appendix B16-B19).

There are several things the designer can do to try to reduce the size of the state generator. The mapping of state names to bit patterns can be chosen to simplify the resulting boolean

Table VI. Summary of the effects of various design parameters.

version of SECD	total number of states	Number of OR-terms (unmerged total/largest merged total/largest)				number of cycles to run TAK
		with live-dead anal. unsorted	with live-dead anal. sorted	without live-dead anal. unsorted	without live-dead anal. sorted	
parallel	161	2160/33 2071/27		2301/34 2206/31		13137101
serial	204	1376/22 1546/21	1012/12	1531/24 1471/22	1060/12	17144484

31% increase in number of cycles (for TAK)
51% reduction in size (total number of OR-terms)

equations. This was not attempted for the CPU. There are programs available which attempt this optimization.

The state generator can be implemented with a counter, so that the state register is either incremented or loaded with a new value. This requires an additional signal to decide whether to increment or load, but possibly simplifies the equations for the state bits since they will have *don't-care* values when the register increments. In the case of the CPU, this technique resulted in a larger implementation of the state generator. There may exist a bit assignment for the states so that this technique would result in a smaller implementation.

The state generator can also be built with a micro-controller. The automatic generation of the micro-code from the selector is quite straight forward. The resulting hardware will typically be slower and may require more area, but if the hardware is being developed on a micro-code development system, such as the Logic Engine [11,12] with all the software and hardware support already present, this may be the quickest path to a working system. The micro-controller can also be used to supply the instruction signals for the abstract objects.

7.6 Instructions

The instruction generators (including the RTC generator) are very similar to the state generator. Each generator is constructed by a partial evaluation of its stream equation. In the case of the instruction generators, the mapping of instruction names to bit patterns must be done by the designer in accordance with the object being instructed. The mapping for the RTC generator was created by DDD after the RTT was sorted. Boolean equations are generated from each instruction equation. Each boolean equation is reduced using espresso and transduced to the Altera ADF format. All the instruction generators and the RTC generator were implemented with one Altera EP1800 PLD (Appendix B11-B15).

8. GARBAGE COLLECTOR DERIVATION

The derivation of the GC is very similar to the CPU derivation. The design is simple enough so that many of the optimization done in the CPU derivation were avoided. The first step is the generation of the selector and the stream equations. This was done in the same way as in the CPU derivation (Appendix D1).

The abstract objects are factored out. The memory unit and the ROM are factored in much the same way as the memory unit for the CPU. The **rom.inst** and **rom.addr** signals generated from the factorization, are removed since the **rom.inst** is always **read** and the **rom.addr** is always the value of the signal **untraced** (Appendix D2).

Part of the function of the level 1 memory interface is to modify the memory address so that the appropriate memory space is accessed, based on the operations **old_addr** and **new_addr**. This is implemented as part of the memory unit, but for convenience, I chose to factor these operations as a separate ALU. This factorization creates two signals, the instruction and the input to this address ALU. Since all memory operations require that the memory address be qualified with either a **new_addr** or **old_addr** operation, the memory address signal is in all cases the result of the address ALU as shown in Figure 9. The memory address signal is removed from the data path unit and the output of the address ALU is wired directly to the memory address port of the memory unit as illustrated in Figure 10 (Appendix D2, D3).

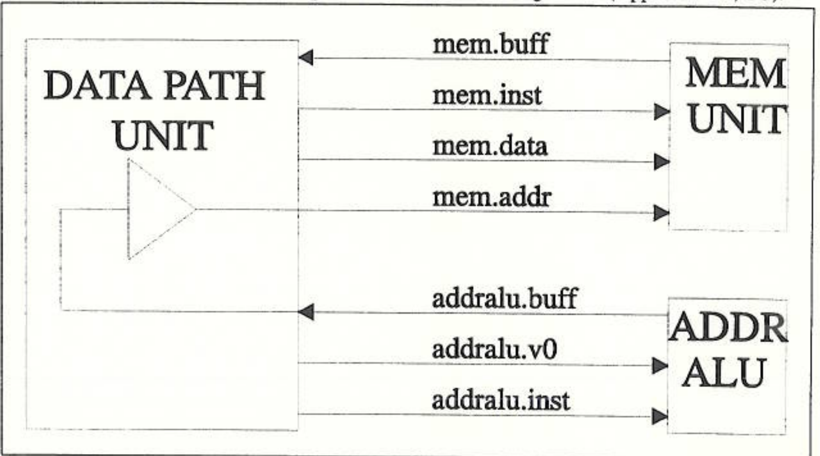


Figure 9. The Address ALU Abstraction.

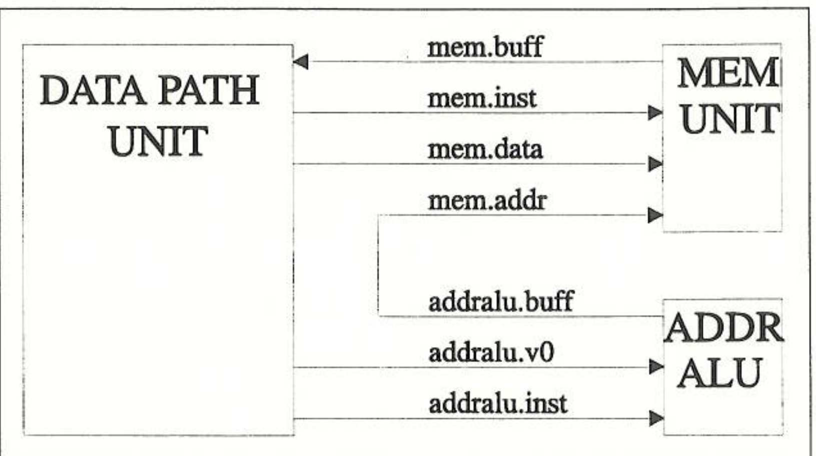


Figure 10. The Address ALU after Optimization.

Two of the registers (**avail** and **untraced**) of the GC are incremented in several places. This operation was originally implemented as a combinational incrementer and factored as an ALU. The incrementer was built using two Altera EP600 PLDs. After gathering statistics on this hardware it was determined that the carry chain of the incrementer was limiting the maximum clock rate of the machine.

The two registers that used this incrementer were then abstracted as separate counters (Appendix D2). These abstract objects have three instructions: **load**, **increment**, and **hold**, and can easily be implemented with "off-the-shelf" counters (74as869.) The process of modifying the scripts, executing the transformations and rewiring the hardware for this modification took a matter of hours. The result was a significant increase in speed of the system.

Since the GC is much smaller and simpler than the CPU, not much effort was needed to come up with a feasible design. Live/dead analysis was done, but there was no need to serialize the specification or to sort the RTT. The instruction stream equations were not removed from the data path unit.

The projection of the stream equations onto bitslices was done in the same way as in the CPU derivation (Appendix D5). The representation table is shown below in table VII.

The signals **untraced.v0** and **avail.v1** are the input values used during **load** operations for the two counters. Although the counters are 16 bits, these signals are only 1 bit each. The only values ever loaded into the **avail** counter are 0 and 2. The only values for the **untraced**

counter are 0 and 1. Since in both cases the values only differ in one bit the other bits can be wired to a constant.

Table VII. Representation table for the garbage collector.

signal	bitslice 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mem.inst	x	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
mem.data	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
header	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-
data	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
untraced.inst	-	-	x	x	-	-	-	-	-	-	-	-	-	-	-	-	-
untraced.v0	-	-	-	-	x	-	-	-	-	-	-	-	-	-	-	-	-
avail.inst	-	-	-	-	-	x	x	-	-	-	-	-	-	-	-	-	-
avail.v0	-	-	-	-	-	-	-	-	x	-	-	-	-	-	-	-	-
gcdone	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
addralu.inst	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
mem.addr	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

x - signal exists in this bitslice
\- - signal does not exists in this bitslice

Due to the differences in views of memory, there are several places in the GC where memory addresses have to be converted from one view to the other. This amounts to a shift operation. A right-shift converts from the CPU view (cell addressable) to the GC view (word addressable) and a left-shift converts from the GC view to the CPU view. These shift operations could easily be implemented as an ALU and factored out of the data path unit, but since it is such a simple operation it can just as easily be implemented inside the data path unit. In order to do this the meaning-of functions are developed for each operation. The **meaning-of-shiftright** function zeros the tag bits and the highest order value bit and uses the next higher order bit for each of the remaining value bits. The projection of (**shiftright avail**) onto bitslices is shown in Figure 11.

The stream equations are projected onto bitslices, boolean equations are generated, reduced, and transduced to Altera ADF format. Each bitslice was implemented using one Altera EP320 or EP600 PLD (Appendix D5-D11).

The state and RTC generators were implemented in the same manner as for the CPU and fit into one Altera EP320 PLD (Appendix D12-D19).

9. PROCESS INTERFACE

The two processes of the SECD machine, the CPU and the GC, are derived separately, but interact closely. The interface between them must be specified in a careful fashion in both processes. The two processes interact in such a way that when one is doing useful work the other is idle. The interface must guarantee that the two process remain in this relationship.

```
(mem.data <== (select state preds ... (shiftright avail) ... ))
```

a) The stream equation before projection.

```
(define meaning-of-shiftright
   (lambda(args slice)
      (cond
         ((or (=? slice 14)(=? slice 15)(=? slice 16)) 0)
         (t (make-ide (car args) (1+ slice))))))
```

b) The meaning-of function for **shiftright**.

```
(mem.data16 <== (select state preds ... 0 ... ))
(mem.data15 <== (select state preds ... 0 ... ))
(mem.data14 <== (select state preds ... 0 ... ))
(mem.data13 <== (select state preds ... avail14 ... ))
(mem.data12 <== (select state preds ... avail13 ... ))
...
(mem.data1 <== (select state preds ... avail2 ... ))
(mem.data0 <== (select state preds ... avail1 ... ))
```

c) The set of stream equations generated after projection.

Figure 11. The Projection of (**shiftright avail**) onto Bitslices.

The execution model of the SECD machine is that a check is made at the beginning of each instruction cycle to determine if there are enough memory cells to execute the next instruction. If so, the instruction is executed. If not, the CPU prepares for garbage collection and signals the GC to collect. The CPU then waits for a signal from the GC that it is done. When the signal is received it recovers from the garbage collection and continues at the start of the execution cycle. Similarly, the GC initially waits for a signal from the CPU for it to start collecting. When this signal is received, it starts the collection of the heap. When done, it signals the CPU and resumes waiting. The timing of this interface is critical. Figure 12 illustrates how this interface was expressed in the source specification.

In state **fetch** of the CPU the predicate **need_2_gc** is queried. This predicate is true when the **avail** pointer reaches or surpasses a certain critical value such that there would not be enough memory cells available for the worst case instruction to execute. This can be done because for every instruction, memory usage is constant. The alternative is to check before each allocation to see if there is a memory cell available. The advantage of the first method is that there is only one place where the CPU has to prepare for, wait for and recover from the garbage collection. If the check were done before each allocation, the code for this would have to be repeated in-line or defined as a subroutine which would then require the implementation of a continuation pointer. The disadvantage is that if the instruction to be executed is not the worst case instruction it is possible that the **need_2_gc** predicate will be true when in fact there is enough memory for the instruction to execute. The first method was chosen because it was the simplest to implement.


```
(define secd
   (letrec (
         .
         .
         .
         (fetch
            (lambda (s e c d mem i j do_gc sio)
               (if (need_2_gc)
                  (init_gc1 s e c d mem (alloc mem) j do_gc sio)
                  (exec s e c d mem (car mem c) j do_gc sio))))
         .
         .
         .
         (init_gc12
            (lambda (s e c d mem i j do_gc donesecd sio)
               (wait_gc s e c d (setcdr! mem j (const nil)) i j (bit true) donesecd sio)))
         (wait_gc
            (lambda (s e c d mem i j do_gc sio)
               (if gcdone
                  (recover_gc1 (car mem (pointer 1)) e c d mem i j (bit nil) sio)
                  (wait_gc s e c d mem i j do_gc sio))))
         .
         .
         .)))

(define secdgc
      (letrec (
         (idlegc
            (lambda (mem header data untraced avail gcdone rom)
               (if do_gc
                  (next-obj mem (mem-read mem (old_addr (abs 0)))
                            data (abs 1) (abs 2) (const nil) rom)
                  (idlegc mem header data untraced avail (const true) rom))))
         .
         .
         .
         (restore
            (lambda (mem header data untraced avail gcdone rom)
               (idlegc (mem-write! mem (old_addr (abs 0)) (/ avail 2))
                       header data untraced avail (const true) rom)))
         .
         .
         .)))
```

Figure 12. The Specification of the Process Interface.

The CPU must prepare for garbage collection before it can signal the GC to begin. This involves creating a list of the values of the registers that must be followed by the GC. The first cell of this list is a dedicated memory location that is used for this purpose only. This is the root of the garbage collection. The rest of the cells of the list are allocated in the normal manner. The number of cells needed to build this list is a constant and must be accounted for by the **need_2_gc** predicate. After garbage collection is complete the CPU must

recover. This involves initializing the registers from the new list of register values starting at the same dedicated memory location and initializing the **avail** pointer of the memory unit.

If **need_2_gc** is true the CPU prepares for garbage collection. If not, it continues with the execution of the next instruction. In state **init_gc12**, preparation for garbage collection is complete, and the signal **do_gc** is set true. This signals the GC to start collecting. The CPU then enters the state **wait_gc** and waits for the GC to signal its completion. The GC, having been in the state **idlegc**, queries the predicate **do_gc**. If false, it stays in the state **idlegc**. If true, it sets the signal **gcdone** false and begins garbage collection. The CPU waits in the state **wait_gc**, checking for the signal **gcdone** to go true. When the GC is done, it sets the signal **gcdone** true and returns to the state **idlegc**. This signals the CPU to leave the state **wait_gc** and to recover from garbage collection. The timing of these events is shown in Figure 13.

The signal **do_gc** in Figure 13 is a clocked signal while **gcdone** is combinational. This is necessary to ensure that **gcdone** is false upon entry into state **wait_gc**. If not the CPU would immediately leave the state **wait_gc**, before garbage collection was done.

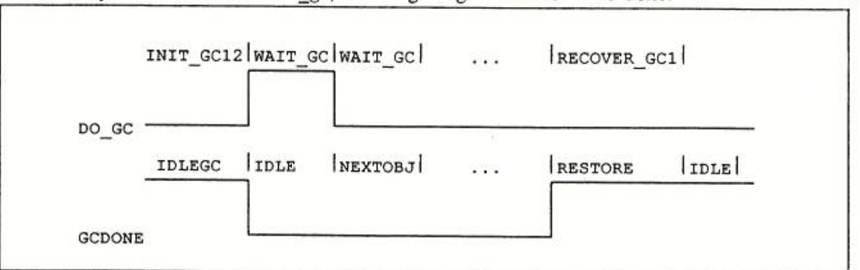


Figure 13. Timing of the Process Interface Events.

With this interface the two processes, the CPU and GC, are able to run as separate, but cooperating machines. Unfortunately, the specification of the process interface can not be simulated. Instead the GC was simulated as a function called from the CPU. This is the one case in which the execution of the specification did not reflect the execution of the resulting hardware.

10. PROBLEMS

There were no errors in the parts of the SECD machine that were generated automatically by DDD. The errors that occurred were all due to human intervention. There were several wiring errors and the implementation of the abstract objects typically went through several iterations before they were correct. The implementation of the process interface was initially wrong and had to be modified.

11. OBSERVATIONS

In my opinion, the use of abstraction in the specification is a very powerful tool. The operations on the memory object were chosen not based on the operation of any particular physical memory, but on an abstraction of memory that best suited the description of the process. This abstraction allows me to conceptualize the memory object of the CPU differently than that of the GC, when in fact they are the same memory object. Also, it allows me to add functionality to the memory unit in addition to that of the physical memory. If I had been restricted to the physical view of memory, the specifications of both processes would have been cluttered with low level details. The operations of the ALU are also abstract. They do not correspond to the operations of any "off-the-shelf" ALU. This freed me from having to chose the exact part to be used at the initiation of the design and keeps the specification clean of implementation details.

The development environment of DDD proved to be very productive. Since the source specification is executable and reflects the behavior of the eventual machine, it provides a simulation and debugging tool. Once the script files defining the sequence of desired transformations is developed, changes in the source specification or the derivation can be reflected in the low level hardware description by simply executing these script files. On a VAX 8800, it takes about 40 minutes to run all the script files for the CPU derivation. Because it is so easy to regenerate the boolean equations, I found myself exploring many different design paths using the size of the boolean equations as a metric. For instance, I explored several different ways of sorting the RTT and serializing the source specification, before I was satisfied with the results. The ease of simulation and derivation of low level specification encourages the designer to experiment with various design options before any hardware is built.

There were several instances in the SECD machine derivation in which transformations were done by hand. Most notably, the sorting of the RTT and the Live/Dead analysis. The addition of support for these operations to DDD would improve its utility.

The specification of the process interface caused the most problems. Since the simulation environment does not allow the processes to be executed concurrently, the GC was run as a subroutine of the CPU. This worked for the SECD machine due to the nature of the interaction of the two processes. It however, did not allow for the modeling of the interface between the two processes. This was the one instance in which the execution of the specification, did not reflect the execution of the hardware. This situation could be improved if DDD provided an environment in which the stream equations could be executed directly.

REFERENCES

[1] Altera Corporation, Altera Programmable Logic User System User Guide (Version 4.0), Altera Corporation, Santa Clara, 1985.

[2] Bose, Bhaskar, DDD - A Transformation System for Deriving Digital Design, Indiana University Department of Computer Science Technical Report (in progress).

[3] Henderson, Peter, Functional Programming, Application and Implementation, (Prentice-Hall, Englewood Cliffs, 1980)

[4] Johnson, Steven D. and Bose, Bhaskar, A System for Digital Design Derivation, Indiana University Department of Computer Science Technical Report No. 289, (August, 1989).

[5] Johnson, Steven D., Hardware Specification, Verification and Synthesis: Mathematical Aspects, Indiana University Department of Computer Science Technical Report No. 279, (June 1989) to appear in: Leeser, M and Browne, G. (eds.), Proceedings of the Cornell Mathematical Sciences Institute Workshop, Ithica, July 1989).

[6] Johnson, Steven D., Bose, Bhaskar and Boyer, C. David, A Tactical Framework for Digital Design, in: Birtwistle, G and Subrahmanyam, P.A., (eds.), VLSI Specification, Verification and Synthesis (Kluwer Academic Publishers, Boston, 1988) pp. 349-383.

[7] Johnson, Steven D., Digital Design in a Functional Calculus, in: Milne, G. and Subrahmanyam, P.A. (eds.) Formal Aspects of VLSI Design (North-Holland, Amsterdam, 1986) pp. 153-178.

[8] Johnson, Steven D., Applicative Programming and Digital Design, Proc. Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1984), pp. 218-227.

[9] Johnson, Steven D., Synthesis of Digital Designs from Recursion Equations (The MIT Press, Cambridge, 1984)

[10] Landin, P.J., The Mechanical Evaluation of Expressions, Computer Journal, 6 (4), 308-320.

[11] Prosser, Franklin P. and Winkel, David, The Art of Digital Design, second ed. (Prentice-Hall, Englewood Cliffs, 1987)

[12] Prosser, Franklin P., and Winkel, David E., The Logic Engine Development System-Support for Microprogrammed Bit-Slice Development, Proc. Micro 16, 84-91.

[13] Rees, Jonathan and Clinger, William C., (eds.), Revised Report on the Algorithmic Language Scheme, Indiana University Computer Science Department Technical Report No. 174 (1986)

[14] Scott, Walter S., Mayo, Robert N., Hamachi, Gordon and Ousterhout, John K., (eds.), 1986 VLSI Tools, Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California at Berkeley, (1985)

[15] Winkel, David, The Use of PALS in CPU Design, Indiana University Department of Computer Science Technical Report No. 204 (1986)

[16] Winkel, David, What Next for PAL-Devices - The Second Generation Challenge, Indiana University Department of Computer Science Technical Report No. 188 (1986)

APPENDICES

APPENDIX A The Entire CPU source specification

APPENDIX B The Script files for the CPU derivation

APPENDIX C The Entire GC source specification

APPENDIX D The Script files for the GC derivation

APPENDIX E The Lisp code for TAK and the Read-Evaluate-Print-Loop


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;    CPU SOURCE SPECIFICATION.                                     ;;
;;                                                                  ;;
;;    For derivation, this file is complete.                        ;;
;;                                                                  ;;
;;    For execution, this file requires four other files:           ;;
;;         parse.ss    - contains all the help functions            ;;
;;         compiler.ss - contains an LISPKIT Lisp compiler          ;;
;;         secdgc.ss   - contains the garbage collector             ;;
;;                       for the SECD Machine.                      ;;
;;         case.ss     - defines the case statement                 ;;
;;                                                                  ;;
;;     There are two places in the code where the derivation        ;;
;;     and execution code differ.  Both versions are included,      ;;
;;     so certain parts of the code need to be commented out        ;;
;;     or uncommented depending on whether a derivation or          ;;
;;     simulation is being done. These differences are clearly      ;;
;;     marked and explained in the code.                            ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(define secd
  (letrec (
       ;
       ;Set up initial values of registers
       ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      (idle
        (lambda (s e c d mem i j do_gc donesecd sio)
          (if dosecd
            (fetch
              (const nil)
              (const nil)
              (pointer 1)               ;leave room for register file
              (const nil)
              *mem*
              ?
              ?
              (bit nil)
              (bit nil)
              sio)
            (idle ? ? ? ? mem ? ? (bit nil)(bit true) sio))))
       ;
       ;Fetch next instruction
       ;~~~~~~~~~~~~~~~~~~~~~~
      (fetch
        (lambda (s e c d mem i j do_gc donesecd sio)
          (if (need_2_gc)
            (init_gcl s e c d mem (alloc* mem) ? (bit nil)(bit nil) sio)
            (exec s e c d mem (car* mem c) ? (bit nil)(bit nil) sio))))
```

```
;
;Prepare for garbage collection
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(init_gc1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc2 s e c d (setcar!* mem (pointer 0) i) i ? (bit nil)
                      (bit nil) sio)))

(init_gc2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc3 ? e c d (setcar!* mem i s) i ? (bit nil) (bit nil) sio)))

(init_gc3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc4 ? e c d mem i (alloc* mem) (bit nil) (bit nil) sio)))

(init_gc4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc5 ? e c d (setcdr!* mem i j) ? j (bit nil) (bit nil) sio)))

(init_gc5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc6 ? ? c d (setcar!* mem j e) ? j (bit nil) (bit nil) sio)))

(init_gc6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc7 ? ? c d mem (alloc* mem) j (bit nil) (bit nil) sio)))

(init_gc7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc8 ? ? c d (setcdr!* mem j i) i ? (bit nil) (bit nil) sio)))

(init_gc8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc9 ? ? ? d (setcar!* mem i c) i ? (bit nil) (bit nil) sio)))

(init_gc9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc10 ? ? ? d mem i (alloc* mem) (bit nil) (bit nil) sio)))

(init_gc10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc11 ? ? ? d (setcdr!* mem i j) ? j (bit nil) (bit nil) sio)))

(init_gc11
  (lambda (s e c d mem i j do_gc donesecd sio)
    (init_gc12 ? ? ? d (setcar!* mem j d) ? j (bit nil) (bit nil) sio)))

(init_gc12
  (lambda (s e c d mem i j do_gc donesecd sio)
    (wait_gc ? ? ? d (setcdr!* mem j (const nil)) ? ? (bit true)
             (bit nil) sio)))
```



```
;
;Wait on garbage collection
;~~~~~~~~~~~~~~~~~~~~~~~~~~
;***********************************************
;This version of wait_gc is for derivation.   ;
;It does a busy wait on the garbage collector.;
;***********************************************
;
(wait_gc
  (lambda (s e c d mem i j do_gc donesecd sio)
    (if gc_done?
      (recover_gc1 (car* mem (pointer 1)) ? ? d mem ? ? (bit nil)
                   (bit nil) sio)
      (wait_gc ? ? ? d mem ? ? (bit nil) (bit nil) sio))))
;************************************************
;This version of wait_gc is for execution .     ;
;It calls the garbage collector as a subroutine.;
;************************************************
(wait_gc
  (lambda (s e c d mem i j do_gc donesecd sio)
    (writeln "garbage collecting av = " *avail*)
    (gc_go)
    (writeln "done av = " *avail*)
    (recover_gc1 (car* mem (pointer 1)) ? ? d mem ? ? (bit nil) (bit nil)
               sio)))
;
;Recover from garbage collection
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(recover_gc1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc2 s ? ? d mem (cdr* mem (pointer 1)) ? (bit nil) (bit nil)
               sio)))

(recover_gc2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc3 s (car* mem i) ? d mem i ? (bit nil) (bit nil) sio)))

(recover_gc3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc4 s e ? d mem ? (cdr* mem i) (bit nil) (bit nil) sio)))

(recover_gc4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc5 s e (car* mem j) d mem ? j (bit nil) (bit nil) sio)))

(recover_gc5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc6 s e c d mem (cdr* mem j) ? (bit nil) (bit nil) sio)))

(recover_gc6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (recover_gc7 s e c (car* mem i) mem ? ? (bit nil) (bit nil) sio)))

(recover_gc7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e c d (mem-init mem (pointer 0)) ? ? (bit nil) (bit nil)
          sio)))
```

```
;
;Decode Instruction
;~~~~~~~~~~~~~~~~~~
(exec
  (lambda (s e c d mem i j do_gc donesecd sio)
    (case i
      (RTN-t   (rtnl ? ? ? d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (DUM-t   (duml s e ? d mem ? (cdr* mem c)(bit nil)(bit nil) sio))
      (AP-t    (apl s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (RAP-t   (rapl s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (SEL-t   (sell s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (JOIN-t  (joinl s e ? d mem (car* mem d) ? (bit nil)(bit nil) sio))
      (CAR-t   (carl s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (CDR-t   (cdrl s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (CONS-t  (consl s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (LD-t    (ldl s e c d mem (cdr* mem c) ? (bit nil)(bit nil) sio))
      (LDC-t   (ldcl s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio))
      (LDF-t   (ldfl s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio))
      (ATOM-t  (testl s e c d mem i (car* mem s)(bit nil)(bit nil) sio))
      (NUM-t   (testl s e c d mem i (car* mem s)(bit nil)(bit nil) sio))
      (SYM-t   (testl s e c d mem i (car* mem s)(bit nil)(bit nil) sio))
      (PAIR-t  (testl s e c d mem i (car* mem s)(bit nil)(bit nil) sio))
      (EQ-t    (eql s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (LEQ-t   (leql s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (ADD-t   (add_1 s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (SUB-t   (sub_1 s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (EXEC-t  (exl s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (SL-t    (sll s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (LS-t    (lsl s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      (CI-t    (cil s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (RECH-t  (rechl s e c d mem ? (alloc* mem)(bit nil)(bit nil) sio))
      (WRCH-t  (wrchl s e c d mem ? (car* mem s)(bit nil)(bit nil) sio))
      (SET-t   (setl s e c d mem (cdr* mem c) ? (bit nil)(bit nil) sio))
      (POP-t   (popl ? e c d mem (cdr* mem s) ? (bit nil)(bit nil) sio))
      (STOP-t  (done s e c d mem ? ? (bit nil)(bit nil) sio)))))
;
;Instruction Code
;~~~~~~~~~~~~~~~~
(done
  (lambda (s e c d mem i j do_gc donesecd sio)
;****************************************************
;Comment out the following line for execution       ;
;Uncomment the following line for derivation        ;
;****************************************************
    (idle ? ? ? ?  mem ? ? (bit nil)(bit true) sio)
;****************************************************
;Comment out the following two lines for derivation;
;Uncomment the following two lines for execution   ;
;****************************************************
;   (writeln "done")
;   (car* mem s)
    ))

(rtnl
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn2 ? ? ? d mem i (alloc* mem)  (bit nil)(bit nil) sio)))
```



```
(rtn2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn3 j ? ? d mem i ? (bit nil)(bit nil) sio)))

(rtn3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn4 s ? ? d mem i (car* mem d)  (bit nil)(bit nil) sio)))

(rtn4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn5 s ? ? d (setcar!* mem s i) ? j (bit nil)(bit nil) sio)))

(rtn5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn6 s ? ? d (setcdr!* mem s j) ? ? (bit nil)(bit nil) sio)))

(rtn6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn7 s ? ? d mem ? (cdr* mem d)  (bit nil)(bit nil) sio)))

(rtn7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn8 s ? ? d mem (car* mem j) j  (bit nil)(bit nil) sio)))

(rtn8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn9 s ? ? d mem i (cdr* mem j)  (bit nil)(bit nil) sio)))

(rtn9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn10 s i ? d mem ? j (bit nil)(bit nil) sio)))

(rtn10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn11 s e ? d mem (car* mem j) j  (bit nil)(bit nil) sio)))

(rtn11
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn12 s e i d mem ? j (bit nil)(bit nil) sio)))

(rtn12
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn13 s e c d mem (cdr* mem j) ?  (bit nil)(bit nil) sio)))

(rtn13
  (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e c i mem ? ? (bit nil)(bit nil) sio)))

(dum1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (dum2 s ? ? d mem e j  (bit nil)(bit nil) sio)))

(dum2
  (lambda(s e c d mem i j do_gc donesecd sio)
    (dum3  s ? j d mem i ? (bit nil)(bit nil) sio)))
```

```
(dum3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (dum4 s ? c d mem i (alloc* mem)  (bit nil)(bit nil) sio)))

(dum4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (dum5 s j c d mem i j (bit nil)(bit nil) sio)))

(dum5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e c d (setcdr!* mem j i) ? ? (bit nil)(bit nil) sio)))

(ap1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap2 s e c d mem ? (cdr* mem j)  (bit nil)(bit nil) sio)))

(ap2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap3 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))

(ap3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap4 s e c d (setcdr!* mem i j) i ? (bit nil)(bit nil) sio)))

(ap4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap5 s e c d mem i (cdr* mem s)  (bit nil)(bit nil) sio)))

(ap5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap6 s e c d mem i (car* mem j)(bit nil)(bit nil) sio)))

(ap6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap7 s e c d (setcar!* mem i j) i ? (bit nil)(bit nil) sio)))

(ap7
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap8 s ? c d mem i e (bit nil)(bit nil) sio)))

(ap8
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap9 s i c d mem ? j (bit nil)(bit nil) sio)))

(ap9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap10 s e ? d mem (cdr* mem c) j   (bit nil)(bit nil) sio)))

(ap10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap11 s e j d mem i ?  (bit nil)(bit nil) sio)))

(ap11
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap12 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))
```



```
(ap12
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap13 s e c d (setcdr!* mem j d) i j (bit nil)(bit nil) sio)))

(ap13
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap14 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(ap14
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap15 s e c d mem (alloc* mem) j  (bit nil)(bit nil) sio)))

(ap15
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap16 s e c d (setcdr!* mem i j) i ? (bit nil)(bit nil) sio)))

(ap16
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap17 s e ? d (setcar!* mem i c) i ? (bit nil)(bit nil) sio)))

(ap17
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap18 s e ? d mem  i (alloc* mem)(bit nil)(bit nil) sio)))

(ap18
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap19 s e ? d (setcdr!* mem j i) ? j (bit nil)(bit nil) sio)))

(ap19
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap20 s e ? d mem (cdr* mem s) j  (bit nil)(bit nil) sio)))

(ap20
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap21 s e ? d mem (cdr* mem i) j  (bit nil)(bit nil) sio)))

(ap21
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap22 s e ? d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(ap22
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap23 s e ? j mem ? ? (bit nil)(bit nil) sio)))

(ap23
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap24 ? e ? d mem (car* mem s) ? (bit nil)(bit nil) sio)))

(ap24
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap25 ? e ? d mem (car* mem i) ? (bit nil)(bit nil) sio)))

(ap25
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ap26 ? e i d mem ? ?  (bit nil)(bit nil) sio)))
```

```
(ap26
  (lambda(s e c d mem i j do_gc donesecd sio)
    (fetch (const nil) e c d mem ? ? (bit nil)(bit nil) sio)))

(rap1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rap2 s e c d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(rap2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rap3 s e c d mem i (cdr* mem s)(bit nil)(bit nil) sio)))

(rap3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rap4 s e c d mem i (car* mem j)(bit nil)(bit nil) sio)))

(rap4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (rap5 s e c d (setcar!* mem i j) i ? (bit nil)(bit nil) sio)))

(rap5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ap8 s ? c d mem i (cdr* mem e)(bit nil)(bit nil) sio)))

(sel1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel2 s e c d mem i (cdr* mem c)(bit nil)(bit nil) sio)))

(sel2
  (lambda(s e c d mem i j do_gc donesecd sio)
    (if (true?* i)
      (sel3 s e c d mem ? j (bit nil)(bit nil) sio)
      (sel3 s e c d mem ? (cdr* mem j)(bit nil)(bit nil) sio))))

(sel3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel4 s e c d mem ? (car* mem j)(bit nil)(bit nil) sio)))

(sel4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel5 s e ? d mem (cdr* mem c) j (bit nil)(bit nil) sio)))

(sel5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel6 s e j d mem i ? (bit nil)(bit nil) sio)))

(sel6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel7 s e c d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(sel7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel8 s e c d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(sel8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel9 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))
```



```
(sel9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sel10 s e c d (setcar!* mem j i) ? j  (bit nil)(bit nil) sio)))

(sel10
  (lambda(s e c d mem i j do_gc donesecd sio)
    (sel11 s e c d (setcdr!* mem j d) ? j (bit nil)(bit nil) sio)))

(sel11
  (lambda(s e c d mem i j do_gc donesecd sio)
    (sel12 s e c j mem ? ? (bit nil)(bit nil) sio)))

(sel12
  (lambda(s e c d mem i j do_gc donesecd sio)
    (sel13 ? e c d mem (cdr* mem s) ? (bit nil)(bit nil) sio)))

(sel13
  (lambda(s e c d mem i j do_gc donesecd sio)
    (fetch i e c d mem ? ? (bit nil)(bit nil) sio)))

(join1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (join2 s e i d mem ? ? (bit nil)(bit nil) sio)))

(join2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (join3 s e c d mem (cdr* mem d) ? (bit nil)(bit nil) sio)))

(join3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e c i mem ? ? (bit nil)(bit nil) sio)))

(car1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car2 s e c d mem (car* mem i) ? (bit nil)(bit nil) sio)))

(car2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car3 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(car3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car4 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(car4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car5 ? e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(car5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car6 ? e c d (setcdr!* mem j i) ? j (bit nil)(bit nil) sio)))

(car6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 j e c d mem ? ? (bit nil)(bit nil) sio)))
```

```
(cdr1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (car2 s e c d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(cons1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (cons2 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))

(cons2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons3 s e c d (setcar!* mem i j) i ? (bit nil)(bit nil) sio)))

(cons3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons4 s e c d mem i (cdr* mem s)(bit nil)(bit nil) sio)))

(cons4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons5 s e c d mem i (car* mem j)(bit nil)(bit nil) sio)))

(cons5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons6 s e c d (setcdr!* mem i j) i ? (bit nil)(bit nil) sio)))

(cons6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons7 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(cons7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons8 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(cons8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons9 ? e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(cons9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons10 j e c d mem i j (bit nil)(bit nil) sio)))

(cons10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (cons11 s e c d mem (cdr* mem i) j (bit nil)(bit nil) sio)))

(cons11
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcdr!* mem j i) ? ? (bit nil)(bit nil) sio)))

(ld1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld2 s e c d mem i e (bit nil)(bit nil) sio)))

(ld2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))
```



```
(ld3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld4 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(ld4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (ld6 s e c d mem ? (car* mem j)(bit nil)(bit nil) sio)
      (ld5 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))

(ld5
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ld4 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))

(ld6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld7 s e c d mem (cdr* mem c) j (bit nil)(bit nil) sio)))

(ld7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld8 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(ld8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld9 s e c d mem (cdr* mem i) j (bit nil)(bit nil) sio)))

(ld9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (ld11 s e c d mem (car* mem j) ? (bit nil)(bit nil) sio)
      (ld10 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))

(ld10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld9 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))

(ld11
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld12 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(ld12
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld13 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(ld13
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld14 ? e c d mem s j (bit nil)(bit nil) sio)))

(ld14
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld15 j e c d mem i j (bit nil)(bit nil) sio)))

(ld15
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ld16 s e c d (setcdr!* mem j i) ? ? (bit nil)(bit nil) sio)))
```

```
(1d16
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1d17 s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio)))

(1d17
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next2 s e ? d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(1dc1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1dc2 s e i d mem i ? (bit nil)(bit nil) sio)))

(1dc2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1dc3 s e c d mem (car* mem i) ? (bit nil)(bit nil) sio)))

(1dc3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1dc4 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(1dc4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1dc5 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(1dc5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1dc6 ? e c d (setcdr!* mem j s) ? j (bit nil)(bit nil) sio)))

(1dc6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 j e c d mem ? ? (bit nil)(bit nil) sio)))

(1df1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df2 s e i d mem i ? (bit nil)(bit nil) sio)))

(1df2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df3 s e c d mem (car* mem i) ? (bit nil)(bit nil) sio)))

(1df3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df4 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(1df4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df5 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(1df5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df6 s e c d (setcdr!* mem j e) ? j (bit nil)(bit nil) sio)))

(1df6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df7 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))
```



```
(1df7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (1df8 s e c d (setcar!* mem i j) i ? (bit nil)(bit nil) sio)))

(1df8
  (lambda(s e c d mem i j do_gc donesecd sio)
    (1df9 ? e c d (setcdr!* mem i s) i ? (bit nil)(bit nil) sio)))

(1df
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next1 i e c d mem ? ? (bit nil)(bit nil) sio)))

(ex1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex2 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(ex2
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex3 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))

(ex3
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex4 s e c d (setcdr!* mem j e) ? j (bit nil)(bit nil) sio)))

(ex4
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex5 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))

(ex5
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex6 s e c d (setcar!* mem i j) i ? (bit nil)(bit nil) sio)))

(ex6
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex7 ? e c d mem i (cdr* mem s)(bit nil)(bit nil) sio)))

(ex7
  (lambda(s e c d mem i j do_gc donesecd sio)
    (ex8 ? e c d (setcdr!* mem i j) i ? (bit nil)(bit nil) sio)))

(ex8
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next1 i e c d mem ? ? (bit nil)(bit nil) sio)))

(test1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (test2 s e c d mem (test?* i j) ? (bit nil)(bit nil) sio)))

(test2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (test3 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))

(test3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (test4 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))
```

```
(test4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (test5 ? e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(test5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (test6 ? e c d (setcdr!* mem j i) ? j (bit nil)(bit nil) sio)))

(test6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 j e c d mem ? ? (bit nil)(bit nil) sio)))

(eq1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq2 s e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(eq2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(eq3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq4 s e c d mem (eq?* i j) ? (bit nil)(bit nil) sio)))

(eq4
  (lambda(s e c d mem i j do_gc donesecd sio)
    (eq5 ? e c d mem i (cdr* mem s)(bit nil)(bit nil) sio)))

(eq5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq6 ? e c d mem i (cdr* mem j)(bit nil)(bit nil) sio)))

(eq6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq7 i e c d mem ? j (bit nil)(bit nil) sio)))

(eq7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq8 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))

(eq8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq9 ? e c d (setcar!* mem i s) i j (bit nil)(bit nil) sio)))

(eq9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq10 i e c d mem i j (bit nil)(bit nil) sio)))

(eq10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcdr!* mem i j) ? ? (bit nil)(bit nil) sio)))

(leq1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (leq2 s e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))
```



```
(leq2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (leq3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(leq3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (eq4 s e c d mem (leq?* i j) ? (bit nil)(bit nil) sio)))

(add_1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_2 s e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(add_2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(add_3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_4 s e c d mem (add* i j) ? (bit nil)(bit nil) sio)))

(add_4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_5 ? e c d mem i (cdr* mem s)(bit nil)(bit nil) sio)))

(add_5
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_6 i e c d mem ? j (bit nil)(bit nil) sio)))

(add_6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_7 s e c d mem ? (cdr* mem j)(bit nil)(bit nil) sio)))

(add_7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_8 s e c d mem (alloc* mem) j (bit nil)(bit nil) sio)))

(add_8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_9 ? e c d (setcar!* mem i s) i j (bit nil)(bit nil) sio)))

(add_9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_10 i e c d mem i j (bit nil)(bit nil) sio)))

(add_10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcdr!* mem i j) ? ? (bit nil)(bit nil) sio)))

(sub_1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sub_2 s e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))

(sub_2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (sub_3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))
```

```
(sub_3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (add_4 s e c d mem (sub* i j) ? (bit nil)(bit nil) sio)))

(sl1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcar!* mem s (sym-list i)) ? ? (bit nil)(bit nil)
           sio)))

(ls1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcar!* mem s (list-sym i)) ? ? (bit nil)(bit nil)
           sio)))

(ci1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (ci2 s e c d mem ? (char2int j)(bit nil)(bit nil) sio)))

(ci2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next1 s e c d (setcar!* mem s j) ? ? (bit nil)(bit nil) sio)))

(set1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set2 s e c d mem i e (bit nil)(bit nil) sio)))

(set2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set3 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(set3
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set4 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(set4
  (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (set6 s e c d mem ? (car* mem j)(bit nil)(bit nil) sio)
      (set5 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))

(set5
  (lambda(s e c d mem i j do_gc donesecd sio)
    (set4 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))

(set6
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set7 s e c d mem (cdr* mem c) j (bit nil)(bit nil) sio)))

(set7
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set8 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))

(set8
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set9 s e c d mem (cdr* mem i) j (bit nil)(bit nil) sio)))
```



```
(set9
  (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (set11 s e c d mem (car* mem s) j (bit nil)(bit nil) sio)
      (set10 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))

(set10
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set9 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))

(set11
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set12 s e c d (setcar!* mem j i) ? ? (bit nil)(bit nil) sio)))

(set12
  (lambda (s e c d mem i j do_gc donesecd sio)
    (set13 s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio)))

(set13
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next2 s e ? d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))

(rech1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (if (char-rdy)
      (rech2 s e c d mem (s-in sio) j (bit nil)(bit nil) sio)
      (rech1 s e c d mem ? j (bit nil)(bit nil) sio))))

(rech2
  (lambda(s e c d mem i j do_gc donesecd sio)
    (rech3 s e c d (setcar!* mem j (char i)) ? j (bit nil)(bit nil)
           sio)))

(rech3
  (lambda(s e c d mem i j do_gc donesecd sio)
    (rech4 ? e c d (setcdr!* mem j s) ? j (bit nil)(bit nil) sio)))

(rech4
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next1 j e c d mem ? ? (bit nil)(bit nil) sio)))

(wrch1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (if (rdy-send)
      (next1 s e c d mem ? ? (bit nil)(bit nil)(s-out sio j))
      (wrch1 s e c d mem ? j (bit nil)(bit nil) sio))))

(pop1
  (lambda(s e c d mem i j do_gc donesecd sio)
    (next1 i e c d mem ? ? (bit nil)(bit nil) sio)))

(next1
  (lambda (s e c d mem i j do_gc donesecd sio)
    (next2 s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio)))
```

```
(next2
  (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e i d mem ? ? (bit nil)(bit nil) sio))))

(lambda()(idle 0 0 0 0 0 0 0 0 0 0))))
```


PENDIX B   The Script Files for the CPU Derivation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                                                              ;;
       This is the script file for generating the control     ;;
       and architecture descriptions from the source          ;;
       specification.                                         ;;
                                                              ;;
       Inputs: Source specification                           ;;
               'secdx0'                                       ;;
                                                              ;;
       Outputs: Stream equations (architecture)               ;;
                'x3'                                          ;;
                                                              ;;
                Select (control)                              ;;
                'xsel'                                        ;;
                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 LOAD DDD
 ~~~~~~~~
load "/u/rmw/secd/xsystem/src/loadxsystem.ss")

define x1
 (iterative->singleloop (read-from-file "/u/rmw/secd/xsystem/input/secdx0")))
define x2 (singleloop->architecture x1))

 DEFINE CONTROL
 ~~~~~~~~~~~~~~
define xsel (singleloop->control x1))

 DEFINE STREAM EQUATIONS
 ~~~~~~~~~~~~~~~~~~~~~~~
define x3 (architecture->streamequations x2))

 OUTPUT
 ~~~~~~
write-to-file x3 "x3")
write-to-file xsel "xsel")
exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                       ;;
;;      This script file is for factoring out the abstractions           ;;
;;      of memory, alu and serial interface.  It also                    ;;
;;      separates the instruction signals and state from the             ;;
;;      rest of the description and removes any redundant                ;;
;;      signals.                                                         ;;
;;                                                                       ;;
;;                                                                       ;;
;;      Inputs: stream equations from 'scrl'                             ;;
;;              'x3'                                                     ;;
;;                                                                       ;;
;;      Outputs: Register Transfer Table                                 ;;
;;               'table' 'pal.opt'                                       ;;
;;                                                                       ;;
;;               Instruction stream equations                            ;;
;;               'mem.inst' 'ser.inst' 'alu.inst'                        ;;
;;                                                                       ;;
;;               State stream equation                                   ;;
;;               'state'                                                 ;;
;;                                                                       ;;
;;               Stream Equations                                        ;;
;;               'x7'                                                    ;;
;;                                                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/dcare.ss")
(load "/u/rmw/secd/xsystem/src/sioabs.ss")
(load "predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; FACTOR OUT THE MEMORY ABSTRACTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define x4 (make-abstract-mem (read-from-file "x3") 'mem))
(writeln "done mem abs")
;
; FACTOR OUT THE ALU ABSTRACTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define alu-op-set '(test?* atom?* eq?* leq?* add* sub* sub1* sub1))
(define x5 (make-abstract-alu x4))
(writeln "done alu abs")
;
; FACTOR OUT THE SERIAL INTERFACE ABSTRACTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define x6 (make-abstract-sio x5 'sio))
(writeln "done sio abs")
```



```
REMOVE INSTRUCTION, REDUNDANT AND STATE SIGNALS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
define x7
(remove-eqn 'sio-addr    ; same as 'j'
(remove-eqn 'm-i         ; from instruction generator
(remove-eqn 's-i         ; from instruction generator
(remove-eqn 'alu-v0      ; same as 'i'
(remove-eqn 'alu-v1      ; same as 'j'
(remove-eqn 'alu-i       ; from instruction generator
(remove-eqn 'state x6)))))))

FORMAT STREAM EQUATIONS AS A REGISTER TRANSFER TABLE
OUTPUT IS "pal.opt" and "table"
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
make-paltable x7 "pal.opt" "table" "pal.all")

OUTPUT

write-to-file (extract-eqn 'm-i x6) "mem.inst")
write-to-file (extract-eqn 's-i x6) "ser.inst")
write-to-file (extract-eqn 'alu-i x6) "alu.inst")
write-to-file  (extract-eqn 'state x6) "state")
write-to-file x7 "x7")

exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;;      This script file creates a new set of stream equations ;;
;;      from the sorted RTT. It also creates the mapping file  ;;
;;      for the RTC.                                            ;;
;;                                                                 ;;
;;      Inputs: Sorted Register Transfer Table                   ;;
;;              'palset.sort'                                    ;;
;;                                                                 ;;
;;              Original Register Transfer Table                 ;;
;;              'pal.all'                                        ;;
;;                                                                 ;;
;;      Outputs: New stream equations                            ;;
;;               'secd.con'                                      ;;
;;                                                                 ;;
;;               RTT mapping file                                ;;
;;               'pal.map'                                       ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/palmap.ss")
(load "predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE NEW STREAM EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define !palset! (read-from-file "palset.sort"))
(write-to-file (create-optimized-circuit-description !palset!) "secd.con")
;
; CREATE MAPPING FILE
; ~~~~~~~~~~~~~~~~~~~
(write-to-file
 (mkpalmap (read-from-file "pal.all")
           !palset!)
 "pal.map")

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                  ;;
;       This script file projects the stream equations onto       ;;
;       bitslices according to the bitmap file, creating a        ;;
;       new set of stream equations for each slice.               ;;
;                                                                  ;;
;       Inputs: bitmap file                                        ;;
;               'bitslices'                                        ;;
;                                                                  ;;
;               stream equations                                   ;;
;               'secd.con'                                         ;;
;                                                                  ;;
;       Outputs: Projected stream equations                        ;;
;                'bit0.con' - 'bit15.con'                          ;;
;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 LOAD DDD
 ~~~~~~~~
load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
load "/u/rmw/secd/xsystem/src/bitslice.ss")
load "/u/rmw/secd/xsystem/run/secd/pal/predinfo")
load "/u/rmw/secd/xsystem/src/global.ss")

 PROJECT STREAM EQUATIONS ONTO BITSLICES
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
define bit-map (transpose (read-from-file "bitslices")))
define eqns (read-from-file "secd.con"))
con->bitslices eqns bit-map)

exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;;      This script file generates espresso truth table format    ;;
;;      files for each signal in all slices in the D-register     ;;
;;      form.                                                      ;;
;;                                                                 ;;
;;      Inputs: Projected stream equations                         ;;
;;              'bit0.con' - 'bit15.con'                           ;;
;;                                                                 ;;
;;      Outputs: Espresso Files                                    ;;
;;               '*.-bit*'                                         ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/espresso.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/equation.ss")
(load "../predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE ESPRESSO FILES
; ~~~~~~~~~~~~~~~~~~~~~
(con->eqn 'bit0)
(con->eqn 'bit1)
(con->eqn 'bit2)
(con->eqn 'bit3)
(con->eqn 'bit4)
(con->eqn 'bit5)
(con->eqn 'bit6)
(con->eqn 'bit7)
(con->eqn 'bit8)
(con->eqn 'bit9)
(con->eqn 'bit10)
(con->eqn 'bit11)
(con->eqn 'bit12)
(con->eqn 'bit13)
(con->eqn 'bit14)
(con->eqn 'bit15)

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;;      This script file generates espresso truth table format    ;;
;;      files for each signal in all slices in the Toggle         ;;
;;      register form.                                             ;;
;;                                                                 ;;
;;      Inputs: Projected stream equations                         ;;
;;              'bit0.con' - 'bit15.con'                           ;;
;;                                                                 ;;
;;      Outputs: Espresso Files                                    ;;
;;               '*.-bit*'                                         ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xoresp.ss")          ; Toggle register form
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/equation.ss")
(load "../predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE ESPRESSO FILES
; ~~~~~~~~~~~~~~~~~~~~~
(con->eqn 'bit0)
(con->eqn 'bit1)
(con->eqn 'bit2)
(con->eqn 'bit3)
(con->eqn 'bit4)
(con->eqn 'bit5)
(con->eqn 'bit6)
(con->eqn 'bit7)
(con->eqn 'bit8)
(con->eqn 'bit9)
(con->eqn 'bit10)
(con->eqn 'bit11)
(con->eqn 'bit12)
(con->eqn 'bit13)
(con->eqn 'bit14)
(con->eqn 'bit15)

(exit)
```

```
####################################################################
##                                                                ##
##      This csh script file runs espresso on each of the         ##
##      espresso files. The outputs from all the espresso         ##
##      runs are grouped into four files.                         ##
##                                                                ##
##      Inputs: Espresso Files                                    ##
##              '*.-bit*'                                         ##
##                                                                ##
##      Outputs: Grouped espresso files                           ##
##              'bg0.esp' bg1.esp' 'bg2.esp' 'bg3.esp'            ##
##                                                                ##
####################################################################
#! /bin/csh -f
/bin/rm -f bg*.esp
foreach i (*bit[0-3])                   # group bit slices 0 - 3
echo $i                                 # together.
espresso $i >> bg0.esp
end
foreach i (*bit[4-7])                   # group bit slices 4 - 7
echo $i                                 # together.
espresso $i >> bg1.esp
end
foreach i (*bit[8-9] *bit1[0-1])        # group bit slices 8 - 11
echo $i                                 # together.
espresso $i >> bg2.esp
end
foreach i (*bit1[2-5])                  # group bit slices 12 - 15
echo $i                                 # together.
espresso $i >> bg3.esp
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;      This script file converts the grouped espresso files        ;;
;;      to boolean equations.                                       ;;
;;                                                                  ;;
;;      Inputs: Grouped espresso files                              ;;
;;              'bg0.esp' bg1.esp' 'bg2.esp' 'bg3.esp'              ;;
;;                                                                  ;;
;;      Outputs: Grouped boolean files                              ;;
;;              'bg0.eqn' bg1.eqn' 'bg2.eqn' 'bg3.eqn'              ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/run/secd/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~
(continue 'bg0)
(continue 'bg1)
(continue 'bg2)
(continue 'bg3)

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;      This script file converts the grouped boolean files          ;;
;;      to Altera ADF format.                                         ;;
;;                                                                    ;;
;;      Inputs: Grouped espresso files                                ;;
;;              'bg0.eqn' bg1.eqn' 'bg2.eqn' 'bg3.eqn'                ;;
;;                                                                    ;;
;;      Outputs: Grouped boolean files                                ;;
;;               'bg0.adf' bg1.adf' 'bg2.adf' 'bg3.adf'               ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; CREATE ALTERA ADF FILES
; ~~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera '(s0 e0 c0 d0 i0 j0        ; specify which signals are registered
                s1 e1 c1 d1 i1 j1
                s2 e2 c2 d2 i2 j2
                s3 e3 c3 d3 i3 j3)
              (split-eqns (read-from-file "bg0.eqn") 8)
              "bg0.adf")

(eqns->altera '(s4 e4 c4 d4 i4 j4
                s5 e5 c5 d5 i5 j5
                s6 e6 c6 d6 i6 j6
                s7 e7 c7 d7 i7 j7)
              (split-eqns (read-from-file "bg1.eqn") 8)
              "bg1.adf")

(eqns->altera '(s8 e8 c8 d8 i8 j8
                s9 e9 c9 d9 i9 j9
                s10 e10 c10 d10 i10 j10
                s11 e11 c11 d11 i11 j11)
              (split-eqns (read-from-file "bg2.eqn") 8)
              "bg2.adf")

(eqns->altera '(s12 e12 c12 d12 i12 j12
                s13 e13 c13 d13 i13 j13
                s14 e14 c14 d14 i14 j14 do_gc14
                s15 e15 c15 d15 i15 j15 donesecd15)
              (split-eqns (read-from-file "bg3.eqn") 8)
              "bg3.adf")

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;      This script file generates the instruction generators        ;;
;;      for the memory unit, serial unit, and the ALU along          ;;
;;      with the generator for the Register Transfer Code.           ;;
;;                                                                    ;;
;;      Inputs: Stream equations                                      ;;
;;              'mem.inst' 'ser.inst' 'alu.inst'                      ;;
;;                                                                    ;;
;;              Mapping files                                         ;;
;;              'mem.map' 'serial.map' 'alu.map' 'pal.map'            ;;
;;                                                                    ;;
;;              Select (control)                                      ;;
;;              'xsel'                                                ;;
;;                                                                    ;;
;;      Outputs: Generators                                           ;;
;;               'mem.generator'                                      ;;
;;               'serial.generator'                                   ;;
;;               'alu.generator'                                      ;;
;;               'pal.generator'                                      ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/state.ss")
(load "/u/rmw/secd/xsystem/src/sel2eqn.ss")
(load "/u/rmw/secd/xsystem/run/secd/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; LOAD THE STREAM EQUATIONS AND MAPPING FILES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define mem.inst (read-from-file "mem.inst"))
(define mem.map (read-from-file "mem.map"))
(define serial.inst (read-from-file "ser.inst"))
(define serial.map (read-from-file "serial.map"))
(define alu.inst (read-from-file "alu.inst"))
(define alu.map (read-from-file "alu.map"))
(define pal.map (read-from-file "pal.map"))
(define xsel  (read-from-file "xsel"))
;
; CREATE THE INSTRUCTION GENERATORS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(write-to-file (apply-select-to-eqn mem.inst xsel mem.map) "mem.generator")
(write-to-file
 (apply-select-to-eqn serial.inst xsel serial.map) "serial.generator")
(write-to-file (apply-select-to-eqn alu.inst xsel alu.map) "alu.generator")
(write-to-file (remap xsel pal.map) "pal.generator")

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;       This script file generates the boolean equations from        ;;
;;       the generators and formats them as truth tables,             ;;
;;       creating espresso files for each bit of each                 ;;
;;       instruction.                                                 ;;
;;                                                                    ;;
;;       Inputs: Generators                                           ;;
;;               'mem.generator'                                      ;;
;;               'serial.generator'                                   ;;
;;               'alu.generator'                                      ;;
;;               'pal.generator'                                      ;;
;;                                                                    ;;
;;               Mapping files                                        ;;
;;               'mem.map' 'serial.map' 'alu.map' 'pal.map'           ;;
;;               'state.map' 'case.map'                               ;;
;;                                                                    ;;
;;       Outputs: Espresso files                                      ;;
;;                'm5.inesp' 'm6.inesp' 'm7.inesp'                    ;;
;;                'a5.inesp' 'a6.inesp' 'a7.inesp'                    ;;
;;                'sr6.inesp' 'sr7.inesp'                             ;;
;;                'pa2.inesp' 'pa3.inesp' 'pa4.inesp'                 ;;
;;                'pa5.inesp' 'pa6.inesp' 'pa7.inesp'                 ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xsel2bool.ss")
(load "/u/rmw/secd/xsystem/src/eqn2esp.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/mpla.ss")
;
; LOAD THE GENERATORS AND MAPPING FILES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define mem.gen (read-from-file "mem.generator"))
(define mem.map (read-from-file "mem.map"))
(define ser.gen (read-from-file "serial.generator"))
(define ser.map (read-from-file "serial.map"))
(define alu.gen (read-from-file "alu.generator"))
(define alu.map (read-from-file "alu.map"))
(define pal.gen (read-from-file "pal.generator"))
(define pal.map (read-from-file "pal.map"))
(define state.map (read-from-file "state.map"))
(define case.map (read-from-file "case.map"))
;
; GENERATE BOOLEAN EQUATIONS FOR EACH GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define mem.bool (xsel2bool mem.gen 'v 'm))
(define ser.bool (xsel2bool ser.gen 'v 'sr))
(define alu.bool (xsel2bool alu.gen 'v 'a))
(define pal.bool (xsel2bool pal.gen 'v 'pa))
```



```
;
; FIX THE CASE IN WHICH THE EQUATION IS NULL
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define fix (lambda(bool)
  (cond
    ((null? bool)())
    ((null? (cadar bool))(fix (cdr bool)))
    (t(cons (car bool)(fix (cdr bool)))))))

(define mem.bool (fix mem.bool))
(define ser.bool (fix ser.bool))
(define alu.bool (fix alu.bool))
(define pal.bool (fix pal.bool))
;
; CREATE ESPRESSO FILES FOR EACH BIT OF EACH INSTRUCTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; DECODE INSTRUCTIONS, CASE STATEMENT ITEMS AND STATE NAMES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(booleqns->espresso
 (statenames->code
  (statenames->code
   (statenames->code mem.bool mem.map (map car mem.bool))
   case.map (s-maker (fit-eqn (length case.map)) 'p3.))
  state.map '(st0 st1 st2 st3 st4 st5 st6 st7))
 '.inesp)

(booleqns->espresso
 (statenames->code
  (statenames->code
   (statenames->code ser.bool ser.map (map car ser.bool))
   case.map (s-maker (fit-eqn (length case.map)) 'p3.))
  state.map '(st0 st1 st2 st3 st4 st5 st6 st7))
 '.inesp)

(booleqns->espresso
 (statenames->code
  (statenames->code
   (statenames->code alu.bool alu.map (map car alu.bool))
   case.map (s-maker (fit-eqn (length case.map)) 'p3.))
  state.map '(st0 st1 st2 st3 st4 st5 st6 st7))
 '.inesp)

(booleqns->espresso
 (statenames->code
  (statenames->code
   (statenames->code pal.bool pal.map (map car pal.bool))
   case.map (s-maker (fit-eqn (length case.map)) 'p3.))
  state.map '(st0 st1 st2 st3 st4 st5 st6 st7))
 '.inesp)

(exit)
```

```
#####################################################################
##                                                                 ##
##      This csh script file runs espresso on each of the          ##
##      espresso files. The outputs from all the espresso          ##
##      runs are grouped into four files.                          ##
##                                                                 ##
##      Inputs: Espresso Files                                     ##
##              '*.inesp'                                          ##
##                                                                 ##
##      Outputs: Grouped espresso files                            ##
##               'inst.esp' 'mem.esp'                              ##
##                                                                 ##
#####################################################################
#! /bin/csh -f
/bin/rm -f inst.esp mem.esp
foreach i (m*.inesp)                # group all memory instruction
echo $i                             # equations together.
espresso $i >> mem.esp
end
foreach i ([asp]*.inesp)            # group all alu, serial and RTC
echo $i                             # equations together.
espresso $i >> inst.esp
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;      This script file generates the boolean equations            ;;
;;      for the grouped espresso files. The equations are           ;;
;;      formatted as Altera ADF files.                              ;;
;;                                                                  ;;
;;      Inputs: Grouped espresso files                              ;;
;;              'inst.esp' 'mem.esp'                                ;;
;;                                                                  ;;
;;      Outputs: Altera ADF files                                   ;;
;;               'inst.adf' 'mem.adf'                               ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; GENERATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~
; CREATE TEMP. FILES (inst.eqn mem.eqn)
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(continue 'inst)
(continue 'mem)
(define eqns (split-eqns (read-from-file "inst.eqn") 8))
(define memeqns (split-eqns (read-from-file "mem.eqn") 8))
;
; CREATE ALTERA ADF FILES
; ~~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera () eqns "inst.adf")

(eqns->altera () memeqns "mem.adf")

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;       This script file creates the state generator and the        ;;
;;       state map from the state stream equation and control        ;;
;;                                                                    ;;
;;       Inputs: State stream equation                                ;;
;;               'state'                                              ;;
;;                                                                    ;;
;;               Control                                              ;;
;;               'xsel'                                               ;;
;;                                                                    ;;
;;       Outputs: State generator                                     ;;
;;                'state.generator'                                   ;;
;;                                                                    ;;
;;                State map                                           ;;
;;                'state.map'                                         ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/state.ss")
(load "/u/rmw/secd/xsystem/run/secd/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; LOAD STATE STREAM EQUATION AND CONTROL
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define state (read-from-file "state"))
(define xsel  (read-from-file "xsel"))
;
; GENERATE STATE MAP AND STATE GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(compose-state-generator state xsel)

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;       This script file generates the boolean equations from       ;;
;;       the state generator and formats them as truth tables,       ;;
;;       creating espresso files for each bit of state.              ;;
;;                                                                    ;;
;;       Inputs: State generators                                     ;;
;;               'state.generator'                                    ;;
;;                                                                    ;;
;;               Mapping files                                        ;;
;;               'state.map' 'case.map'                               ;;
;;                                                                    ;;
;;       Outputs: Espresso files                                      ;;
;;               'st0.inesp' 'st1.inesp' 'st2.inesp''st3.inesp';;
;;               'st4.inesp' 'st5.inesp' 'st6.inesp''st7.inesp';;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xsel2bool.ss")
(load "/u/rmw/secd/xsystem/src/eqn2esp.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/mpla.ss")
;
; LOAD THE STATE GENERATOR AND MAPPING FILES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define sg (read-from-file "state.generator"))
(define sm (read-from-file "state.map"))
(define cm (read-from-file "case.map"))
;
; GENERATE BOOLEAN EQUATIONS FOR THE STATE GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;(define sb (stategenerator->bool sg sm))        ; Used for loader version
(define sb (xsel2bool sg 's 'st))                ; Used for calculated version
;
; CREATE ESPRESSO FILES FOR EACH BIT OF STATE
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; DECODE CASE STATEMENT ITEMS AND STATE NAMES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(booleqns->espresso
 (statenames->code
  (statenames->code sb sm (map car sb))
  cm
  (s-maker (fit-eqn (length cm)) 'p3.))
 '.inesp)

(exit)
```

```
########################################################################
##                                                                    ##
##      This csh script file runs espresso on each of the             ##
##      espresso files. The outputs from all the espresso             ##
##      runs are grouped into one file.                               ##
##                                                                    ##
##      Inputs: Espresso Files                                        ##
##              'st[0-7].inesp'                                       ##
##                                                                    ##
##      Outputs: Grouped espresso file                                ##
##              'state.esp'                                           ##
##                                                                    ##
########################################################################
#! /bin/csh -f
/bin/rm -f state.esp
foreach i (*.inesp)                     # group all state bits
echo $i                                 # together
espresso $i >> state.esp
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;      This script file generates the boolean equations              ;;
;;      for the state espresso files. The equations are               ;;
;;      formatted as Altera ADF files.                                ;;
;;                                                                    ;;
;;      Inputs: Grouped espresso file                                 ;;
;;              'state.esp'                                           ;;
;;                                                                    ;;
;;      Outputs: Altera ADF files                                     ;;
;;              'state.adf'                                           ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; GENERATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~
; CREATE TEMP. FILE (state.eqn)
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(continue 'state)
(define eqns (split-eqns (read-from-file "state.eqn") 8))
;
; CREATE ALTERA ADF FILE
; ~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera '(st0 st1 st2 st3
                st4 st5 st6 st7)         ; specify which signals are registered
              eqns
              "state.adf")

(exit)
```

APPENDIX C The Entire GC Source Specification

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;      GARBAGE COLLECTOR SOURCE SPECIFICATION.                       ;;
;;                                                                    ;;
;;      This process preforms two functions for the SECD              ;;
;;      machine.  It copies an initial heap from ROM to RAM           ;;
;;      on reset and garbage collects (stop-and-copy) the             ;;
;;      heap when a request is made.                                  ;;
;;                                                                    ;;
;;      For derivation, this file is complete.                        ;;
;;                                                                    ;;
;;      For execution, this file requires one other file.             ;;
;;           parse.ss - contains all the help functions               ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
(define secdgc
  (letrec (
        ;
        ; Set up initial values of registers
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (idle
          (lambda (mem header data untraced avail gcdone rom)
            (if (and reset do_gc)          ; copy initial heap upon reset
              (copy mem ? ? (load (val 0)) (load (val 0)) (const nil)
                    (mem-write! rom 0 0))   ; hack
              (if do_gc                    ; start a garbage collection
                (next-obj mem (mem-read mem (old_addr (abs 0))) ? (load val 1))
                          (load (val 2)) (const nil) rom)
                (idle mem ? ? ? ? (const true) rom)))))
        ;
        ; Copy initial heap from ROM to RAM.
        ; Untraced points into ROM and avail points into RAM.
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (copy
          (lambda (mem header data untraced avail gcdone rom)
            (copy2 (mem-write! mem (new_addr avail) (mem-read rom untraced))
                   ? ? untraced (inc ?) (const nil) rom)))
        (copy2
          (lambda (mem header data untraced avail gcdone rom)
            (if (=? avail 0)
              (idle (switch_mem mem) ? ? ? ? (const nil) rom)
              (copy mem ? ? (inc ?) avail (const nil) rom))))
        ;
        ; Garbage Collector
        ; ~~~~~~~~~~~~~~~~~
        ; header   - Holds current untraced value.
        ; data     - Holds valueat location header.
        ; untraced - Points into new memory.
        ;            Pointers located  below untraced
        ;            point to locations  in new memory.
        ;            Pointers located  above untraced
        ;            point to locations in old memory.
        ; avail    - Points to next available location
        ;            available in new memory.
```



```
        ;
        ; Restore is the last state of the garbage collector
        ; before returning to idle. Here the new value of avail
        ; is stored in memory location 0.
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (restore
          (lambda (mem header data untraced avail gcdone rom)
            (idle (mem-write! mem (old_addr (abs 0)) (/ avail 2)) ? ? ? ?
                  (const nil) rom)))
        ;
        ; Driver is the top of the garbage collection loop.
        ; If avail is equal to untraced, garbage collection is
        ; complete and the memory spaces are exchanged.
        ; Otherwise the next object is examined.
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (driver
          (lambda (mem header data untraced avail gcdone rom)
            (if (= untraced avail)
              (restore (switch_mem mem) ? ? ? avail (const nil) rom)
              (next-obj mem (mem-read mem (new_addr untraced)) ? untraced avail
                        (const nil) rom))))
        ; Next-obj is the entry point of the garbage collection.
        ; If the current object is a pointer, get the data it
        ; points to, otherwise it does not need to be followed
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (next-obj
          (lambda (mem header data untraced avail gcdone rom)
            (if (spointer?* header)
              (obj-type mem header
                        (mem-read mem (old_addr (* 2 (pntprt header))))
                        untraced avail (const nil) rom)
              (driver mem ? ? (inc ?) avail (const nil) rom))))
        ; If the data object is not a forward pointer, it must
        ; be a 'cons' cell. Copy the 'car' field from old memory
        ; to new memory.  If the data is a forward pointer, update
        ; to point to proper location in the new memory.
        ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        (obj-type
          (lambda (mem header data untraced avail gcdone rom)
            (if (sforward?* data)
              (driver (mem-write! mem (new_addr untraced) (combine header data))
                      ? ? (inc ?) avail (const nil) rom)
              (pairl (mem-write! mem (new_addr avail) data)
                     header ? untraced avail (const nil) rom))))
```

```
  ; Leave a forward pointer from old memory to new memory
  ; in the 'car' position in case there are other pointers
  ; to this cell.
  ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (pair1
   (lambda (mem header data untraced avail gcdone rom)
     (pair2 (mem-write! mem (old_addr (* 2 (pntprt header)))
            (forward (/ avail 2)))
            header ? untraced avail (const nil) rom)))
  ; Update pointer in new memory at untraced to point to
  ; new memory copy of data.
  ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (pair2
   (lambda (mem header data untraced avail gcdone rom)
     (pair3 (mem-write! mem (new_addr untraced)
            (combine header (/ avail 2)))
            header ? untraced (inc ?) (const nil) rom)))
  ; Copy 'cdr' of pair from old to new memory.
  ; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (pair3
   (lambda (mem header data untraced avail gcdone rom)
     (pair4 mem ?
            (mem-read mem (old_addr (cdr* (* 2 (pntprt header)))))
            (inc ?) avail (const nil) rom)))

 (pair4
   (lambda (mem header data untraced avail gcdone rom)
     (driver (mem-write! mem (new_addr avail) data)
             ? ? untraced (inc ?) (const nil) rom))))
(lambda()(idle 0 0 0 0 0 0 0 0))))
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;;        This is the script file for generating the control     ;;
;;        and architecture descriptions from the source          ;;
;;        specification.                                         ;;
;;                                                                ;;
;;        Inputs: Source specification                           ;;
;;                'newgc0'                                       ;;
;;                                                                ;;
;;        Outputs: Stream equations (architecture)               ;;
;;                 'x3'                                          ;;
;;                                                                ;;
;;                 Select (control)                              ;;
;;                 'xsel'                                        ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
;
(define x1
  (iterative->singleloop
    (read-from-file "/u/rmw/secd/xsystem/input/newgcx0")))
(define x2 (singleloop->architecture x1))
;
; DEFINE CONTROL
; ~~~~~~~~~~~~~~
(define xsel (singleloop->control x1))
;
; DEFINE STREAM EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~
(define x3 (architecture->streamequations x2))
;
; OUTPUT
; ~~~~~~
(write-to-file x3 "x3")
(write-to-file xsel "xsel")
(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;        This script file is for factoring out the abstractions;;
;;        of memory, alu and serial interface.  It also          ;;
;;        separates the instruction signals and state from the   ;;
;;        rest of the description and removes any redundant      ;;
;;        signals.                                               ;;
;;                                                               ;;
;;                                                               ;;
;;        Inputs: stream equations from 'scrl'                   ;;
;;                'x3'                                           ;;
;;                                                               ;;
;;        Outputs: Register Transfer Table                       ;;
;;                 'table' 'pal.opt'                             ;;
;;                                                               ;;
;;                 State stream equation                         ;;
;;                 'state'                                       ;;
;;                                                               ;;
;;                 Stream Equations                              ;;
;;                 'x6'                                          ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/dcare.ss")
(load "/u/rmw/secd/xsystem/src/sioabs.ss")
(load "predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; FACTOR OUT THE MEMORY ABSTRACTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define x4 (make-abstract-mem (read-from-file "x3") 'mem))
(define x4 (remove-eqn 'rom-data
           (remove-eqn 'r-a    ; same as untraced
           (remove-eqn 'r-i    ; always read
             (make-abstract-mem x4 'rom)))))
(writeln "done mem abs")
;
; FACTOR OUT THE COUNTER ABSTRACTIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define x5 (make-abstract-signal-equation 'avail
           (make-abstract-signal-equation 'untraced x4)))
;
; FACTOR OUT THE LEVEL 2 INTERFACE ABSTRACTION
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define alu-op-set '(old_addr new_addr))
(define x5 (make-abstract-alu x5))
```



```
;
; REWIRE OUTPUT OF L2 INTERFACE TO MEMORY
; RENAME INPUTS TO COUNTERS~~~~~~~~~~~~~
; ~~~~~~~~~~~~~~~~~~~~~~~~~
(define x5 (rename-eqn 'alu-i 'm-i2
           (rename-eqn 'alu-v0 'm-a
           (rename-eqn 'avail-v0 'av-val
           (rename-eqn 'untraced-v0 'untr-val
           (remove-eqn 'm-a x5))))))
(writeln "done alu abs")
;
; REMOVE STATE SIGNAL
; ~~~~~~~~~~~~~~~~~~
(define x6
 (remove-eqn 'state x5))
;
; FORMAT STREAM EQUATIONS AS A REGISTER TRANSFER TABLE
; OUTPUT IS "pal.opt" and "table"
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(make-paltable x6 "pal.opt" "table" "pal.all")
;
; OUTPUT
;
(write-to-file  (extract-eqn 'state x5) "state")
(write-to-file x6 "x6")
(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;;        This script file creates a new set of stream equations;;
;;        from the sorted RTT. It also creates the mapping file ;;
;;        for the RTC.                                               ;;
;;                                                                   ;;
;;        Inputs: Sorted Register Transfer Table                     ;;
;;                'palset.sort'                                      ;;
;;                                                                   ;;
;;                Original Register Transfer Table                   ;;
;;                'pal.all'                                          ;;
;;                                                                   ;;
;;        Outputs: New stream equations                              ;;
;;                 'gc.con'                                          ;;
;;                                                                   ;;
;;                 RTT mapping file                                  ;;
;;                 'pal.map'                                         ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/palmap.ss")
(load "predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE NEW STREAM EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define !palset! (read-from-file "palset.sort"))
(write-to-file (create-optimized-circuit-description !palset!) "gc.con")
;
; CREATE MAPPING FILE
; ~~~~~~~~~~~~~~~~~~~
(write-to-file
 (mkpalmap (read-from-file "pal.all")
           !palset!)
 "pal.map")

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;;        This script file projects the stream equations onto        ;;
;;        bitslices according to the bitmap file, creating a         ;;
;;        new set of stream equations for each slice.                ;;
;;                                                                   ;;
;;        Inputs: bitmap file                                        ;;
;;                'bitslices'                                        ;;
;;                                                                   ;;
;;                stream equations                                   ;;
;;                'gc.con'                                           ;;
;;                                                                   ;;
;;        Outputs: Projected stream equations                        ;;
;;                 'bit0.con' - 'bit16.con'                          ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "bitslice.ss")
(load "/u/rmw/secd/xsystem/run/newgc/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; PROJECT STREAM EQUATIONS ONTO BITSLICES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define bit-map (transpose (read-from-file "bitslices")))
(define eqns (read-from-file "gc.con"))
(con->bitslices eqns bit-map)

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;        This script file generates espresso truth table format;;
;;        files for each signal in all slices in the D-register ;;
;;        form.                                                ;;
;;                                                             ;;
;;        Inputs: Projected stream equations                   ;;
;;                'bit0.con' - 'bit16.con'                     ;;
;;                                                             ;;
;;        Outputs: Espresso Files                              ;;
;;                 '*.-bit*'                                   ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/espresso.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/equation.ss")
(load "../predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE ESPRESSO FILES
; ~~~~~~~~~~~~~~~~~~~~
(con->eqn 'bit0)
(con->eqn 'bit1)
(con->eqn 'bit2)
(con->eqn 'bit3)
(con->eqn 'bit4)
(con->eqn 'bit5)
(con->eqn 'bit6)
(con->eqn 'bit7)
(con->eqn 'bit8)
(con->eqn 'bit9)
(con->eqn 'bit10)
(con->eqn 'bit11)
(con->eqn 'bit12)
(con->eqn 'bit13)
(con->eqn 'bit14)
(con->eqn 'bit15)
(con->eqn 'bit16)

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;        This script file generates espresso truth table format;;
;;        files for each signal in all slices in the Toggle    ;;
;;        register form.                                       ;;
;;                                                             ;;
;;        Inputs: Projected stream equations                   ;;
;;                'bit0.con' - 'bit15.con'                     ;;
;;                                                             ;;
;;        Outputs: Espresso Files                              ;;
;;                 '*.-bit*'                                   ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xoresp.ss")        ; Toggle register form
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/equation.ss")
(load "../predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE ESPRESSO FILES
; ~~~~~~~~~~~~~~~~~~~~
(con->eqn 'bit0)
(con->eqn 'bit1)
(con->eqn 'bit2)
(con->eqn 'bit3)
(con->eqn 'bit4)
(con->eqn 'bit5)
(con->eqn 'bit6)
(con->eqn 'bit7)
(con->eqn 'bit8)
(con->eqn 'bit9)
(con->eqn 'bit10)
(con->eqn 'bit11)
(con->eqn 'bit12)
(con->eqn 'bit13)
(con->eqn 'bit14)
(con->eqn 'bit15)
(con->eqn 'bit16)

(exit)
```

```
#####################################################################
##                                                                 ##
##       This csh script file runs espresso on each of the         ##
##       espresso files. The outputs from all the espresso         ##
##       runs are grouped into seventeen files.                    ##
##                                                                 ##
##       Inputs: Espresso Files                                    ##
##               '*.-bit*'                                         ##
##                                                                 ##
##       Outputs: Grouped espresso files                           ##
##                'bit[0-16].esp'                                  ##
##                                                                 ##
#####################################################################
#! /bin/csh -f
foreach i (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
  /bin/rm -f bit$i.esp
  foreach j (*bit$i)                 # group each bitslice
   echo $j
   espresso $j >> bit$i.esp
  end
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;       This script file converts the grouped espresso files      ;;
;;       to boolean equations.                                      ;;
;;                                                                  ;;
;;       Inputs: Grouped espresso files                             ;;
;;               'bit[0-16].esp'                                    ;;
;;                                                                  ;;
;;       Outputs: Grouped boolean files                             ;;
;;                'bit[0-16].eqn'                                   ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/run/secd/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; CREATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~
(continue 'bit0)
(continue 'bit1)
(continue 'bit2)
(continue 'bit3)
(continue 'bit4)
(continue 'bit5)
(continue 'bit6)
(continue 'bit7)
(continue 'bit8)
(continue 'bit9)
(continue 'bit10)
(continue 'bit11)
(continue 'bit12)
(continue 'bit13)
(continue 'bit14)
(continue 'bit15)
(continue 'bit16)

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;        This script file converts the grouped boolean files        ;;
;;        to Altera ADF format.                                       ;;
;;                                                                    ;;
;;        Inputs: Grouped espresso files                              ;;
;;                'bit[0-16].eqn'                                     ;;
;;                                                                    ;;
;;        Outputs: Grouped boolean files                              ;;
;;                'bit[0-16].adf'                                     ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; CREATE ALTERA ADF FILES
; ~~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera '(header0 data0)        ; specify which signals are registered
                (read-from-file "bit0.eqn") "bit0.adf")

(eqns->altera '(header1 data1)
                (read-from-file "bit1.eqn") "bit1.adf")

(eqns->altera '(header2 data2)
                (read-from-file "bit2.eqn") "bit2.adf")

(eqns->altera '(header3 data3)
                (read-from-file "bit3.eqn") "bit3.adf")

(eqns->altera '(header4 data4)
                (read-from-file "bit4.eqn") "bit4.adf")

(eqns->altera '(header5 data5)
                (read-from-file "bit5.eqn") "bit5.adf")

(eqns->altera '(header6 data6)
                (read-from-file "bit6.eqn") "bit6.adf")

(eqns->altera '(header7 data7)
                (read-from-file "bit7.eqn") "bit7.adf")

(eqns->altera '(header8 data8)
                (read-from-file "bit8.eqn") "bit8.adf")

(eqns->altera '(header9 data9)
                (read-from-file "bit9.eqn") "bit9.adf")

(eqns->altera '(header10 data10)
                (read-from-file "bit10.eqn") "bit10.adf")

(eqns->altera '(header11 data11)
                (read-from-file "bit11.eqn") "bit11.adf")
```



```
(eqns->altera '(header12 data12)
                (read-from-file "bit12.eqn") "bit12.adf")

(eqns->altera '(header13 data13)
                (read-from-file "bit13.eqn") "bit13.adf")

(eqns->altera '(header14 data14)
                (read-from-file "bit14.eqn") "bit14.adf")

(eqns->altera '(header15 data15)
                (read-from-file "bit15.eqn") "bit15.adf")

(eqns->altera '(data16)
                (read-from-file "bit16.eqn") "bit16.adf")

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;        This script file generates the Register Transfer Code ;;
;;        generator.                                                  ;;
;;                                                                    ;;
;;        Inputs: Mapping files                                       ;;
;;                'pal.map'                                           ;;
;;                                                                    ;;
;;                Select (control)                                    ;;
;;                'xsel'                                              ;;
;;                                                                    ;;
;;        Outputs: RTC generator                                      ;;
;;                 'pal.generator'                                    ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/state.ss")
(load "/u/rmw/secd/xsystem/src/sel2eqn.ss")
(load "/u/rmw/secd/xsystem/run/newgc/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; LOAD CONTROL AND THE RTC  MAPPING FILE
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define xsel  (read-from-file "xsel"))
(define pal.map (read-from-file "pal.map"))
;
; CREATE THE RTC GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~
(write-to-file (remap xsel pal.map) "pal.generator")

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;        This script file generates the boolean equations from ;;
;;        the RTC generator and formats them as truth tables,   ;;
;;        creating espresso files for each bit of the RTC.      ;;
;;                                                                    ;;
;;        Inputs: Generators                                          ;;
;;                'pal.generator'                                     ;;
;;                                                                    ;;
;;                Mapping files                                       ;;
;;                'pal.map' 'state.map'                               ;;
;;                                                                    ;;
;;        Outputs: Espresso files                                     ;;
;;                 'pal.inesp' 'pa2.inesp'                            ;;
;;                 'pa3.inesp' 'pa4.inesp'                            ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xsel2bool.ss")
(load "/u/rmw/secd/xsystem/src/eqn2esp.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/mpla.ss")
;
; LOAD THE RTC GENERATOR AND MAPPING FILES
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define pal.gen (read-from-file "pal.generator"))
(define pal.map (read-from-file "pal.map"))
(define state.map (read-from-file "state.map"))
;
; GENERATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~
(define pal.bool (xsel2bool pal.gen 'v 'pa))
;
; FIX THE CASE IN WHICH THE EQUATION IS NULL
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define fix (lambda(bool)
  (cond
    ((null? bool)())
    ((null? (cadar bool))(fix (cdr bool)))
    (t(cons (car bool)(fix (cdr bool)))))))

(define pal.bool (fix pal.bool))
;
; CREATE ESPRESSO FILES FOR EACH BIT OF THE RTC
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; DECODE STATE NAMES
; ~~~~~~~~~~~~~~~~~~
(booleqns->espresso
 (statenames->code
   (statenames->code pal.bool pal.map (map car pal.bool))
  state.map '(st1 st2 st3 st4))
 '.inesp)

(exit)
```

```
#####################################################################
##                                                                 ##
##      This csh script file runs espresso on each of the          ##
##      espresso files. The outputs from all the espresso          ##
##      runs are grouped into one file.                            ##
##                                                                 ##
##      Inputs: Espresso Files                                     ##
##              '*.inesp'                                          ##
##                                                                 ##
##      Outputs: Grouped espresso file                             ##
##               'pal.esp'                                         ##
##                                                                 ##
#####################################################################
#! /bin/csh -f
/bin/rm -f pal.esp
foreach i (*.inesp)                    # group all RTC equtions
echo $i                                # together.
espresso $i >> pal.esp
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;;      This script file generates the boolean equations for      ;;
;;      the RTC espresso files. The equations are formatted       ;;
;;      as Altera ADF files.                                       ;;
;;                                                                 ;;
;;      Inputs: RTC espresso file                                  ;;
;;              'pal.esp'                                          ;;
;;                                                                 ;;
;;      Outputs: Altera ADF file                                   ;;
;;               'pal.adf'                                         ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; GENERATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~
; CREATE TEMP. FILE (pal.eqn)
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
(continue 'pal)
(define eqns (split-eqns (read-from-file "pal.eqn") 8))
;
; CREATE ALTERA ADF FILE
; ~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera ()
              eqns
              "pal.adf")

(exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;        This script file creates the state generator and the        ;;
;;        state map from the state stream equation and control        ;;
;;                                                                    ;;
;;        Inputs: State stream equation                               ;;
;;                'state'                                             ;;
;;                                                                    ;;
;;                Control                                             ;;
;;                'xsel'                                              ;;
;;                                                                    ;;
;;        Outputs: State generator                                    ;;
;;                 'state.generator'                                  ;;
;;                                                                    ;;
;;                 State map                                          ;;
;;                 'state.map'                                        ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/state.ss")
(load "/u/rmw/secd/xsystem/run/newgc/pal/predinfo")
(load "/u/rmw/secd/xsystem/src/global.ss")
;
; LOAD STATE STREAM EQUATION AND CONTROL
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define state (read-from-file "state"))
(define xsel  (read-from-file "xsel"))
;
; GENERATE STATE MAP AND STATE GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(compose-state-generator state xsel)

(exit)
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;        This script file generates the boolean equations from ;;
;;        the state generator and formats them as truth tables, ;;
;;        creating espresso files for each bit of state.             ;;
;;                                                                    ;;
;;        Inputs: State generators                                    ;;
;;                'state.generator'                                   ;;
;;                                                                    ;;
;;                Mapping files                                       ;;
;;                'state.map'                                         ;;
;;                                                                    ;;
;;        Outputs: Espresso files                                     ;;
;;                 'st0.inesp' 'st1.inesp' 'st2.inesp'                ;;
;;                 'st3.inesp' 'st4.inesp'                            ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/xsel2bool.ss")
(load "/u/rmw/secd/xsystem/src/eqn2esp.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/mpla.ss")
;
; LOAD THE STATE GENERATOR AND MAPPING FILE
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define sg (read-from-file "state.generator"))
(define sm (read-from-file "state.map"))
;
; GENERATE BOOLEAN EQUATIONS FOR THE STATE GENERATOR
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define sb (xsel2bool sg 's 'st))
;
; CREATE ESPRESSO FILES FOR EACH BIT OF STATE
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; DECODE STATE NAMES
; ~~~~~~~~~~~~~~~~~~
(booleqns->espresso
  (statenames->code sb sm (map car sb))
 '.inesp)

(exit)
```

```
##################################################################
##                                                              ##
##      This csh script file runs espresso on each of the      ##
##      espresso files. The outputs from all the espresso      ##
##      runs are grouped into one file.                        ##
##                                                              ##
##      Inputs: Espresso Files                                 ##
##              'st[0-4].inesp'                                ##
##                                                              ##
##      Outputs: Grouped espresso file                         ##
##              'state.esp'                                    ##
##                                                              ##
##################################################################
#! /bin/csh -f
/bin/rm -f state.esp
foreach i (*.inesp)                 # group all state bits
echo $i                             # together
espresso $i >> state.esp
end
```



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;      This script file generates the boolean equations          ;;
;;      for the state espresso files. The equations are           ;;
;;      formatted as Altera ADF files.                            ;;
;;                                                                  ;;
;;      Inputs: Grouped espresso file                             ;;
;;              'state.esp'                                       ;;
;;                                                                  ;;
;;      Outputs: Altera ADF files                                 ;;
;;              'state.adf'                                       ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LOAD DDD
; ~~~~~~~~
(load "/u/rmw/secd/xsystem/src/loadxsystem.ss")
(load "/u/rmw/secd/xsystem/src/con-eqn.ss")
(load "/u/rmw/secd/xsystem/src/eqnesp.ss")
(load "/u/rmw/secd/xsystem/src/esp-eqn.ss")
(load "/u/rmw/secd/xsystem/src/altera.ss")
(load "/u/rmw/secd/xsystem/src/split.ss")
;
; GENERATE BOOLEAN EQUATIONS
; ~~~~~~~~~~~~~~~~~~~~~~~~~~
; CREATE TEMP. FILE (state.eqn)
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(continue 'state)
(define eqns (split-eqns (read-from-file "state.eqn") 8))
;
; CREATE ALTERA ADF FILE
; ~~~~~~~~~~~~~~~~~~~~~~
(eqns->altera '(st1 st2 st3 st4)        ; specify which signals are registered
              eqns
              "state.adf")

(exit)
```

APPENDIX E The Lisp Code for TAK and the Read-Evaluate-Print-Loop

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                        ;;
;;      Lisp code for TAK.                                                ;;
;;      includes functions:                                               ;;
;;            tak - to calculate tak                                      ;;
;;            loop - to repeatedly invoke (tak '18 '12 '6)                ;;
;;            write-interger - to display result                          ;;
;;                                                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define tak
   '(letrec (loop)

       (loop lambda()
          (begin
             (write-integer (tak '18 '12 '6))
             (writechar '#\newline)
             (writechar '#\return)
             (loop)))

       (tak lambda(x y z)
          (if (leq x y)
             z
             (tak
                (tak (sub x '1) y z)
                (tak (sub y '1) z x)
                (tak (sub z '1) x y))))

       (write-integer
          let
             (lambda (n)
                (let
                   (if (eq v '0)
                      (writechar '#\0)
                      (let
                         (begin
                            (if (eq q '0) '()(write-integer q))
                            (writechar (list-ref numerals r)))
                         (q . (quot v '10)) (r . (rema v '10))))
                   (v . (if (leq n '0)
                           (begin
                              (writechar '#\-)
                              (sub '0 n))
                           n))))
             (numerals . '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)))

       (quot
          lambda(n m)
             (if (leq n m)
                (if (eq n m)
                   (add '1 (quot (sub n m) m))
                   '0)
                (add '1 (quot (sub n m) m))))
```



```
(rema
   lambda(n m)
      (if (leq n m)
         (if (eq n m)
            (rema (sub n m) m)
            n)
         (rema (sub n m) m)))

(list-ref
   lambda(l n)
      (if (eq n '0)
         (car l)
         (list-ref (cdr l)(sub n '1))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;      Lisp code for Read-Evaluate-Print-Loop                        ;;
;;      includes functions:                                           ;;
;;           loop - to repeatly read, compile and execute an          ;;
;;                  expression.                                       ;;
;;           write - to write the result of the execution.            ;;
;;           read - to read an s-expression.                          ;;
;;           comp - to compile the expression.                        ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define repl
  '(letrec (loop)
      (loop
        lambda()
          (begin
            (writechar '#\return)
            (writechar '#\:)
            (writechar '#\:)
            (writechar '#\space)
            (write (exec (comp (read) '() '(5))))
            (loop)))

      (write
        lambda (x)
          (if (number x)(write-integer x)
            (if (eq x '())
              (begin
                (writechar '#\( )
                (writechar '#\) ))
                (if (eq x 't) (writechar '#\t)
                  (if (symbol x)(write-symbol (sym-list x))
                    (if (pair x)
                      (if (eq (car x) 'quote)
                        (begin
                          (writechar '#\')
                          (write (car (cdr x))))
                        (begin
                          (writechar '#\( )
                          (write (car x))
                          (write-list (cdr x))))
                          (writechar x)))))))
```



```
(write-list
  lambda(ls)
    (if (eq '() ls) (writechar '#\) )
      (if (pair ls)
        (begin
          (writechar '#\space)
          (write (car ls))
          (write-list (cdr ls)))
        (begin
          (writechar '#\space)
          (writechar '#\.)
          (writechar '#\space)
          (write ls)
          (writechar '#\))))))

(write-integer
  lambda (n)
    (let
      (if (eq v '0)
        (writechar '#\0)
        (let
          (begin
            (if (eq q '0) '()(write-integer q))
            (writechar (list-ref numerals r)))
          (q . (quot v '10)) (r . (rema v '10))))
      (v . (if (leq n '0)
             (begin
               (writechar '#\-)
               (sub '0 n))
             n))))

(numerals . '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9))

(quot
  lambda(n m)
    (if (leq n m)
      (if (eq n m)
        (add '1 (quot (sub n m) m))
        '0)
      (add '1 (quot (sub n m) m))))

(rema
  lambda(n m)
    (if (leq n m)
      (if (eq n m)
        (rema (sub n m) m)
        n)
      (rema (sub n m) m)))

(list-ref
  lambda(l n)
    (if (eq n '0)
      (car l)
      (list-ref (cdr l)(sub n '1))))
```

```
(write-symbol
   lambda(l)
      (if (eq '() l) '()
          (begin
             (writechar (car l))
             (write-symbol (cdr l)))))

(read-char-type
   lambda (c)
      (list-ref char-table (char-int c)))

(char-table . '(
            cn cn cn cn          ;nul soh stx etx
            cn cn cn cn          ;eot enq ack bel
            cn wh wh cn          ;bs  ht  lf  vt
            wh wh cn cn          ;ff  cr  s0  s1
            cn cn cn cn          ;dle dc1 dc2 dc3
            cn cn cn cn          ;dc4 nak syn etb
            cn cn cn cn          ;can em  sub esc
            cn cn cn cn          ;fs  gs  rs  us
            wh al dq ha          ;sp  !   "   #      "
            al al al qu          ;$   %   &   '
            lp rp al pl          ;(   )   *   +
            ca mi al al          ;,   -   .   /
            di di di di          ;0   1   2   3
            di di di di          ;4   5   6   7
            di di al ct          ;8   9   :   ;
            al al al al          ;<   =   >   ?
            al al al al          ;@   A   B   C
            al al al al          ;D   E   F   G
            al al al al          ;H   I   J   K
            al al al al          ;L   M   N   O
            al al al al          ;P   Q   R   S
            al al al al          ;T   U   V   W
            al al al lb          ;X   Y   Z   (
            al rb al al          ;\   )   ^   _
            bq al al al          ;`   a   b   c
            al al al al          ;d   e   f   g
            al al al al          ;h   i   j   k
            al al al al          ;l   m   n   o
            al al al al          ;p   q   r   s
            al al al al          ;t   u   v   w
            al al al al          ;x   y   z   {
            al al al cn          ;|   }   ~   del
            cn))                 ;codes 128-255
```



```
(read
   lambda ()
      (let
         (let
            (if (eq char-type 'wh) (read)
            (if (eq char-type 'al) (read-symbol (cons c '()))
            (if (eq char-type 'di) (read-number (cons c '()) '())
            (if (eq char-type 'lp) (read-list)
            (if (eq char-type 'pl)
               (let
                  (let (if (eq char-type 'di)
                           (read-number (cons c '()) '())
                           (begin
                              (unread-char c)
                              (read-symbol (cons '#\+ '())))))
                       (char-type . (read-char-type c)))
                  (c . (read-char)))
            (if (eq char-type 'mi)
               (let
                  (let (if (eq char-type 'di)
                           (read-number (cons c '()) 't)
                           (begin
                              (unread-char c)
                              (read-symbol (cons '#\- '())))))
                       (char-type . (read-char-type c)))
                  (c . (read-char)))
            (if (eq char-type 'qu) (cons 'quote (cons (read) '()))
               (write char-type))))))))
            (char-type . (read-char-type c)))
         (c . (read-char))))

(read-symbol
   lambda (ls)
      (let
         (let
            (if (eq char-type 'al)
               (read-symbol (cons c ls))
               (begin
                  (unread-char c)
                  (list-sym (store (reverse ls) stringstore))))
            (char-type . (read-char-type c)))
         (c . (read-char))))

(read-number
   lambda (ls neg)
      (let
         (let
            (if (eq char-type 'di)
                (read-number (cons c ls) neg)
                (begin
                   (unread-char c)
                   (let
                      (if neg (sub '0 val) val)
                      (val . (num-val ls)))))
            (char-type . (read-char-type c)))
         (c . (read-char))))
```

```
(num-val
  lambda(ls)
    (if (eq '() ls)
        '0
        (add (sub (char-int (car ls)) zero)
             (mull (num-val (cdr ls)) '10))))

(zero . (char-int '#\0))

(list-number
  lambda(ls)
    (if (eq '() ls)
        '0
        (add (sub (char-int (car ls)) (char-int '#\0))
             (mull (list-number (cdr ls)) '10))))

(read-list
  lambda ()
    (let
       (if (eq 'wh (read-char-type c)) (read-list)
       (if (eq c '#\) ) '()
       (if (eq c '#\.)
           (let
              (if (eq '() rest) (writechar '#\w)
              (if (cdr rest) (writechar '#\w)
                  (car rest)))
              (rest . (read-list)))
           (begin
              (unread-char c)
              (let (cons next (read-list))
                   (next . (read))))))
     (c . (read-char))))

(read-buffer . '() )

(unread-char
  lambda(c)
    (set! read-buffer (cons c read-buffer)))

(read-char
  lambda()
    (if (eq '() read-buffer)
        (readchar)
        (let
           (begin
              (set! read-buffer (cdr read-buffer))
              ret)
           (ret . (car read-buffer)))))

(reverse
  lambda(l)
    (if (eq '() l)
        '()
        (append (reverse (cdr l))(car l))))
```



```
(append
  lambda(l a)
    (if (eq '() l)
        (cons a '())
        (cons (car l)(append (cdr l) a))))

(mull
  lambda(n m)
    (if (eq n '0)
        '0
        (add m (mull (sub n '1) m))))

(comp
  lambda (E N C)
    (if (atom E)(cons (quote 1) (cons (location E N) c))
    (if (eq (car E) (quote quote))
        (cons (quote 2) (cons (car (cdr E)) C))
    (if (eq (car E) (quote add))
        (comp (car (cdr E)) N
          (comp (car (cdr (cdr E))) N (cons (quote 15) C)))
    (if (eq (car E) (quote sub))
        (comp (car (cdr E)) N
          (comp (car (cdr (cdr E))) N (cons (quote 16) C)))
    (if (eq (car E) (quote leq))
        (comp (car (cdr E)) N
          (comp (car (cdr (cdr E))) N (cons (quote 20) C)))
    (if (eq (car E) (quote eq))
        (comp (car (cdr E)) N
          (comp (car (cdr (cdr E))) N (cons (quote 14) C)))
    (if (eq (car E) (quote car))
        (comp (car (cdr E)) N (cons (quote 10) C))
    (if (eq (car E) (quote cdr))
        (comp (car (cdr E)) N (cons (quote 11) C))
    (if (eq (car E) (quote atom))
        (comp (car (cdr E)) N (cons (quote 12) C))
    (if (eq (car E) (quote cons))
        (comp (car (cdr (cdr E))) N
          (comp (car (cdr E)) N (cons (quote 13) C)))
    (if (eq (car E) (quote if))
        (let
           (comp (car (cdr E)) N
              (cons (quote 8) (cons thenpt (cons elsept C))))
           (thenpt comp (car (cdr (cdr E))) N (quote (9)))
           (elsept comp (car (cdr (cdr (cdr E)))) N (quote (9))))
    (if (eq (car E) (quote lambda))
        (let
           (cons (quote 3) (cons body C))
           (body comp (car (cdr (cdr E)))
              (cons (car (cdr E)) N) (quote (5))))
    (if (eq (car E) (quote let))
        (let
           (let
              (complis Args N (cons (quote 3)
                  (cons body (cons (quote 4) C))))
              (body comp (car (cdr E)) M (quote (5))))
           (M cons (vars (cdr (cdr E))) N)
           (Args exprs (cdr (cdr E))))
```

```
      (if (eq (car E) (quote letrec))
          (let
             (let
                (cons (quote 6)
                   (complis Args M (cons (quote 3)
                      (cons body (cons (quote 7) C)))))
                (body comp (car (cdr E)) M (quote (5))))
             (M cons (vars (cdr (cdr E))) N)
             (Args exprs (cdr (cdr E))))
          (complis (cdr E) N
             (comp (car E) N (cons (quote 4) C))))))))))))))))))

(complis
   lambda (E N C)
      (if (eq E (quote nil))
          (cons (quote 2) (cons (quote nil) C))
          (complis (cdr E) N (comp (car E) N (cons (quote 13) C)))))

(location
   lambda (E N)
      (if (member E (car N))
          (cons (quote 0) (posn E (car N)))
          (incar (location E (cdr N)))))

(member
   lambda (E N)
      (if (eq N (quote nil))
          (quote f)
          (if (eq E (car N)) (quote t) (member E (cdr N)))))

(posn
   lambda (E N)
      (if (eq E (car N))
          (quote 0)
          (add (quote 1) (posn E (cdr N)))))

(incar
   lambda (L)
      (cons (add (quote 1) (car L)) (cdr L)))

(vars
   lambda (D)
      (if (eq D (quote nil))
          (quote nil)
          (cons (car (car D)) (vars (cdr D)))))

(exprs
   lambda (D)
      (if (eq D (quote nil))
          (quote nil)
          (cons (cdr (car D)) (exprs (cdr D)))))
```



```
(store
   lambda(l ss)
      (if (eq '() ss)
          (begin
             (set! stringstore (cons l stringstore))
             l)
          (if (stringequal? l (car ss))
              (car ss)
              (store l (cdr ss)))))

(stringstore . (immed 2462))

(stringequal?
   lambda(l1 l2)
      (if (eq '() l1)
          (if (eq '() l2)
              '#t
              '())
          (if (eq (car l1)(car l2))
              (stringequal? (cdr l1)(cdr l2))
              '()))))

(copy
   lambda(l)
      (if (eq '() l)
          '()
          (cons (car l)(copy (cdr l)))))))
```