The Scheme Machine *

Robert G. Burger

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
burger@cs.indiana.edu

Indiana University Computer Science Department
Technical Report #413

August 1994

Abstract

This paper describes the design and implementation of the Scheme Machine, a symbolic computer derived from an abstract Scheme interpreter. The derivation is performed in several transformation passes. First, the interpreter is factored into a compiler and an abstract CPU. Next, the CPU specification is refined so that it can be used with the Digital Design Derivation system. Finally, the DDD system assists in the transformation into hardware. The resulting CPU, implemented in field programmable gate arrays and PALs, is interfaced to a garbage-collecting heap to form a complete Scheme system.

1 Introduction

This report describes the design and implementation of the Scheme Machine CPU. In chapter 12 of *Essentials of Programming Languages* [2], a simple CPS Scheme interpreter is transformed into a compiler and a virtual machine with very specialized, Scheme-specific instructions. The Scheme Machine implements a slight variation of this virtual machine.

This project started in the spring of 1993 with some informal meetings held by Steve Johnson. Peter Weingartner and I learned about the Digital Design Derivation (DDD) system [1] and began to transform the software CPU I designed in the C511 course into a form acceptable to DDD. In the summer of 1993 I worked on the Scheme Machine for C890 credit under Steve Johnson. Kamlesh Rath, Esen Tuna, and Willie Hunt helped me learn more about the DDD system and how to design hardware simple enough to fit on the Actel field-programmable gate arrays. I finished the majority of the design in the summer. In the fall of 1993 I worked on the implementation for C690 credit under Steve Johnson. I was able to fit the main datapath and some of the control hardware on a single large Actel chip. Two smaller Actel chips were used for the next-state and continuation

*The research reported herein was supported in part by a National Science Foundation Graduate Research Fellowship.

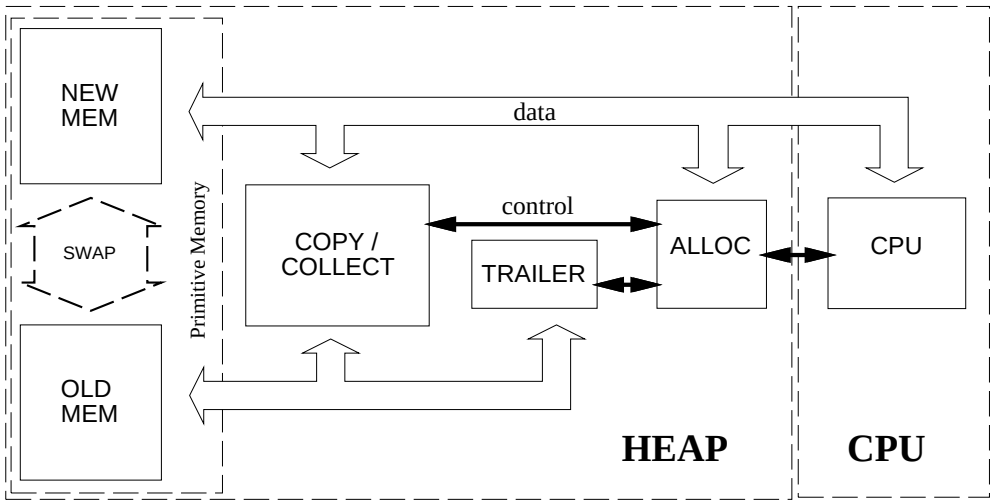

Figure 1: High-level architecture

hardware. The remaining control hardware fit onto 10 PALs. The hardware garbage-collecting heap had already been built, so I simply interfaced with it. There was a subtle bug in the heap that caused the garbage collector to lock up sporadically. Early in 1994 Willie Hunt corrected the problem.

The Scheme Machine consists of the CPU and heap management hardware on a Logic Engine circuit board. The heap consists of two semi-spaces each containing 1,048,576 32-bit words. Heap management is divided into three tasks: allocation, garbage collection, and invalidation. The only interface to the CPU is through the allocator. The garbage collector uses a simple stop-and-copy approach. The invalidation task is performed by the "trailer," a process that goes through the old semi-space and writes innocuous immediate values at the same time the CPU accesses the new space. Because of the trailer process, the CPU need not initialize newly allocated structures because they are guaranteed to contain no pointers. Figure 1 shows the high-level relationships between the components.

2 Interpreter

We begin with a simple CPS Scheme interpreter augmented with an interrupt handler and an error handler. Every time *eval-exp* is invoked, the machine will check for an interrupt. (In this interpreter, the external interrupt is stored in the global variable **interrupt**.) If there is an interrupt, the interrupt handler (stored in global variable **interrupt-handler**) is called with no arguments. The continuation simply throws away the result from the interrupt handler and continues execution at the point the interrupt occurred. The flag **interrupts-enabled** is used to enable and disable interrupts. It is disabled upon entry to the interrupt handler.

The environment is represented as a linked structure of Scheme vectors. *env* is a vector representing the current lexical environment. The first element is a vector representing the next enclosing environment. The remaining elements are pairs whose car is the variable's name and whose cdr is

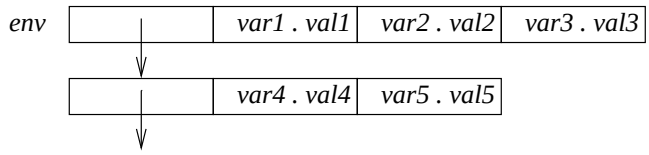

Figure 2: Environment structure

the variable's value. Figure 2 shows this relationship.

The arguments to a procedure are stored in a Scheme vector. *eval-rands* evaluates the arguments in order and stores them in the *args* vector. Figure 3 shows the augmented interpreter.

There are three types of procedures: closures, primitives, and callcc. Closures are generated by the proc clause of *eval-exp*. Primitives are the basic Scheme routines provided by the interpreter. Callcc is used to represent *call/cc* continuations. Figure 4 shows the function *apply-proc*, which processes these procedures.

The interpreter provides very minimal error detection. If an error is found, an error handler (stored in global variable **error-handler**) is invoked.

```
(define eval-exp
  (lambda (exp env k)
    (if (and *interrupt* *interrupts-enabled*)
        (begin
          (set! *interrupts-enabled* #f)
          (apply-proc *interrupt-handler*
                      (vector)
                      (lambda (v)
                        (eval-exp exp env k))))
        (variant-case exp
          [lit (datum) (k datum)]
          [varref (var) (k (lookup var env))]
          [varassign (var exp)
            (eval-exp exp env (lambda (v) (k (assign! var v env))))]
          [if (test-exp then-exp else-exp)
            (eval-exp test-exp env
                      (lambda (v)
                        (if v
                            (eval-exp then-exp env k)
                            (eval-exp else-exp env k))))]
          [begin (exp1 exp2)
            (eval-exp exp1 env
                      (lambda (v)
                        (eval-exp exp2 env k)))]
          [proc (formals body)
            (k (make-closure formals body env))]
          [app (rator rands)
            (eval-exp rator env
                      (lambda (op)
                        (eval-rands rands env (make-vector (length rands)) 0
                                    (lambda (args)
                                      (apply-proc op args k)))))]
          [else (printf "Illegal expression ~s~%" exp)
                (apply-proc *error-handler* (vector opcode-error) k)])))))

(define eval-rands
  (lambda (rands env args i k)
    (if (null? rands)
        (k args)
        (eval-exp (car rands) env
          (lambda (v)
            (vector-set! args i v)
            (eval-rands (cdr rands) env args (add1 i) k))))))
```

Figure 3: CPS interpreter with interrupt and error handling

```
(define apply-proc
  (lambda (op args k)
    (variant-case op
      [closure (formals body env)
        (eval-exp body (extend-env env formals args) k)]
      [primitive (f)
        (case f
          [call/cc (apply-proc (vector-ref args 0)
                               (vector (make-callcc k))
                               k)]
          [abort (printf "~%Scheme Interpreter Aborting!~%")
                 (void)]
          [interrupts-enabled
            (if (zero? (vector-length args))
              (k *interrupts-enabled*)
              (begin
                (set! *interrupts-enabled* (vector-ref args 0))
                (k (void))))]
          [interrupt-handler
            (if (zero? (vector-length args))
              (k *interrupt-handler*)
              (begin
                (set! *interrupt-handler* (vector-ref args 0))
                (k (void))))]
          [error-handler
            (if (zero? (vector-length args))
              (k *error-handler*)
              (begin
                (set! *error-handler* (vector-ref args 0))
                (k (void))))]
          [else (k (apply (eval f) (vector->list args)))])]
      [callcc (k)
        (k (vector-ref args 0))]
      [else (printf "Illegal procedure ~s~%" op)
            (apply-proc *error-handler* (vector procedure-error) k)])))
```

Figure 4: *apply-proc*

3 Compiler and Abstract Machine Derivation

Using the techniques shown in chapter 12 of *Essentials of Programming Languages*, the interpreter was transformed into a compiler and an abstract machine. The initial transformation results in code that is tree structured. Figures 5 and 6 show the initial compiler and abstract machine, respectively.

The next step was to convert the tree-structured code into linear code. Figure 7 gives the linearization procedure. The change in code gives us a new abstract machine. It executes the code using six registers: *acc*, *val**, *lex-env*, *k*, *code*, and *pc*. The accumulator, *acc*, holds the result of the current expression. The values register, *val**, holds a vector containing the arguments to a procedure. The environment register, *lex-env*, holds the vector representing the current lexical environment. The lexical environment no longer carries the variable names since the compiler generates indices for variables. The continuation register, *k*, holds the current continuation. The code register, *code*, holds the current code vector. The program counter, *pc*, holds the current index into the code vector for the current instruction. Figure 8 shows the new abstract machine.

At this point the continuation register *k* holds a Scheme closure. We next change this representation to records so that we can eventually represent continuations in hardware. (We have already represented machine closures as records.) We also note that the accumulator, *acc*, is dead in the `args-inst` and `push-inst` cases, so we modify these cases to destroy the accumulator in hopes that the resulting hardware will be smaller. Figure 9 gives the revised abstract machine that represents continuations using records.

```
(define compile-exp
  (lambda (tail-pos? exp symbol-table next-code)
    (variant-case exp
      [lit (datum) (make-lit-action datum next-code)]
      [varref (var) (make-varref-action (lexical-addr symbol-table var) next-code)]
      [varassign (var exp)
        (compile-exp #f exp symbol-table
                     (make-varassign-action (lexical-addr symbol-table var) next-code))]
      [if (test-exp then-exp else-exp)
       (compile-exp #f test-exp symbol-table
                    (make-if-action (compile-exp tail-pos? then-exp symbol-table next-code)
                                    (compile-exp tail-pos? else-exp symbol-table next-code)))]
      [begin (exp1 exp2) (compile-exp #f exp1 symbol-table
                                      (compile-exp tail-pos? exp2 symbol-table next-code))]
      [proc (formals body)
       (make-proc-action formals (compile-exp #t body (cons formals symbol-table) restore-code)
                         next-code)]
      [app (rator rands)
       (let ([app-code
              (compile-rands rands symbol-table (compile-exp #f rator symbol-table invoke-code))])
         (if tail-pos? app-code (make-save-action app-code next-code)))])))

(define compile-rands
  (letrec
      ([help (lambda (rands pos symbol-table next-code)
               (if (null? rands)
                   next-code
                   (compile-exp #f (car rands) symbol-table
                                (make-push-action pos
                                  (help (cdr rands) (add1 pos) symbol-table next-code)))))])
    (lambda (rands symbol-table next-code)
      (make-args-action (length rands) (help rands 0 symbol-table next-code)))))

(define restore-code
  (make-restore-action))

(define invoke-code
  (make-invoke-action))
```

Figure 5: Compiler

```
(define apply-action
  (lambda (action acc val* lex-env k)
    (if (and *interrupt* *interrupts-enabled*)
        (begin
          (set! *interrupts-enabled* #f)
          (apply-proc *interrupt-handler* (vector)
            (lambda (v)
              (apply-action action acc val* lex-env k))))
        (variant-case action
          [lit-action (datum code)
           (apply-action code datum val* lex-env k)]
          [varref-action (lex-addr code)
           (apply-action code (lookup-lex-env lex-env lex-addr) val* lex-env k)]
          [varassign-action (lex-addr code)
           (assign-lex-env! lex-env lex-addr acc)
           (apply-action code (void) val* lex-env k)]
          [if-action (then-code else-code)
           (if (true-value? acc)
               (apply-action then-code acc val* lex-env k)
               (apply-action else-code acc val* lex-env k))]
          [proc-action (formals code-of-body code)
           (apply-action code (make-closure lex-env code-of-body) val* lex-env k)]
          [restore-action ()
           (k acc)]
          [invoke-action ()
           (apply-proc acc val* k)]
          [save-action (code1 code2)
           (apply-action code1 (void) '() lex-env
                         (lambda (v)
                           (apply-action code2 v val* lex-env k)))]
          [args-action (size code)
           (apply-action code acc (make-vector size '()) lex-env k)]
          [push-action (pos code)
           (vector-set! val* pos acc)
           (apply-action code acc val* lex-env k)])))))
```

Figure 6: Abstract machine

```
(define linearize
  (letrec
      ([help
        (lambda (code loc)
          (variant-case code
            [lit-action (datum code)
              (cons (make-lit-inst datum)
                    (help code (add1 loc)))]
            [varref-action (lex-addr code)
              (cons (make-varref-inst lex-addr)
                    (help code (add1 loc)))]
            [varassign-action (lex-addr code)
              (cons (make-varassign-inst lex-addr)
                    (help code (add1 loc)))]
            [if-action (then-code else-code)
              (let* ([then (help then-code (add1 loc))]
                     [else-loc (+ 1 loc (length then))])
                (append (cons (make-if-inst else-loc) then)
                        (help else-code else-loc)))]
            [proc-action (formals code-of-body code)
              (let* ([rest (help code (add1 loc))]
                     [loc-of-body (+ 1 loc (length rest))])
                (append (cons (make-proc-inst loc-of-body) rest)
                        (help code-of-body loc-of-body)))]
            [restore-action ()
              (list (make-restore-inst))]
            [invoke-action ()
              (list (make-invoke-inst))]
            [save-action (code1 code2)
              (let* ([first (help code1 (add1 loc))]
                     [next-loc (+ 1 loc (length first))])
                (append (cons (make-save-inst next-loc) first)
                        (help code2 next-loc)))]
            [args-action (size code)
              (cons (make-args-inst size)
                    (help code (add1 loc)))]
            [push-action (pos code)
              (cons (make-push-inst pos)
                    (help code (add1 loc)))]))])
    (lambda (code)
      (list->vector (help code 0)))))
```

Figure 7: Code linearization procedure

```
(define elc
  (lambda (acc val* lex-env k code pc)
    (if (and *interrupt* *interrupts-enabled*)
        (begin
          (set! *interrupts-enabled* #f)
          (apply-proc *interrupt-handler* #() (lambda (v) (elc acc val* lex-env k code pc))))
        (variant-case (vector-ref code pc)
          [lit-inst (datum) (elc datum val* lex-env k code (add1 pc))]
          [varref-inst (lex-addr)
           (let ([acc (lookup-lex-env lex-env lex-addr)])
             (if (eq? acc '*)
                 (apply-proc *error-handler* (vector "Unbound variable" '*) k)
                 (elc acc val* lex-env k code (add1 pc))))]
          [varassign-inst (lex-addr)
           (assign-lex-env! lex-env lex-addr acc)
           (elc (void) val* lex-env k code (add1 pc))]
          [if-inst (else-loc)
           (if (true-value? acc)
               (elc acc val* lex-env k code (add1 pc))
               (elc acc val* lex-env k code else-loc))]
          [proc-inst (loc-of-body)
           (elc (make-closure lex-env code loc-of-body) val* lex-env k code (add1 pc))]
          [restore-inst () (k acc)]
          [invoke-inst () (apply-proc acc val* k)]
          [save-inst (next-loc)
           (elc (void) '() lex-env (lambda (v) (elc v val* lex-env k code next-loc)) code (add1 pc))]
          [args-inst (size) (elc acc (make-vector size '()) lex-env k code (add1 pc))]
          [push-inst (pos)
           (vector-set! val* pos acc)
           (elc acc val* lex-env k code (add1 pc))]))))

(define apply-proc
  (lambda (proc args k)
    (variant-case proc
      [closure (lex-env code pc) (elc (void) '() (extend-lex-env args lex-env) k code pc)]
      [primitive (f) (k (apply f (vector->list args)))]
      [callcc (k) (k (vector-ref args 0))]
      [else (apply-proc *error-handler* (vector "Invalid procedure" proc) k)])))
```

Figure 8: Initial abstract machine for linearized code

```
(define elc
  (lambda (acc val* lex-env k code pc)
    (if (and *interrupt* *interrupts-enabled*)
        (begin
          (set! *interrupts-enabled* #f)
          (apply-proc *interrupt-handler* #() (make-int-cont acc val* lex-env k code pc)))
        (variant-case (vector-ref code pc)
          [lit-inst (datum) (elc datum val* lex-env k code (add1 pc))]
          [varref-inst (lex-addr)
            (let ([acc (lookup-lex-env lex-env lex-addr)])
              (if (eq? acc '*undefined*)
                  (apply-proc *error-handler* (vector "Unbound variable" '*) k)
                  (elc acc val* lex-env k code (add1 pc))))]
          [varassign-inst (lex-addr)
            (assign-lex-env! lex-env lex-addr acc)
            (elc (void) val* lex-env k code (add1 pc))]
          [if-inst (else-loc)
            (if (true-value? acc)
                (elc acc val* lex-env k code (add1 pc))
                (elc acc val* lex-env k code else-loc))]
          [proc-inst (loc-of-body)
           (elc (make-closure lex-env code loc-of-body) val* lex-env k code (add1 pc))]
          [restore-inst () (apply-cont k acc)]
          [invoke-inst () (apply-proc acc val* k)]
          [save-inst (next-loc)
            (elc (void) '() lex-env (make-save-cont val* lex-env k code next-loc) code (add1 pc))]
          [args-inst (size) (elc (void) (make-vector size '()) lex-env k code (add1 pc))]
          [push-inst (pos)
            (vector-set! val* pos acc)
            (elc (void) val* lex-env k code (add1 pc))]))))

(define apply-cont
  (lambda (cont v)
    (variant-case cont
      [save-cont (val* lex-env k code pc) (elc v val* lex-env k code pc)]
      [int-cont (acc val* lex-env k code pc) (elc acc val* lex-env k code pc)]
      [done-cont () v])))

(define apply-proc
  (lambda (proc args k)
    (variant-case proc
      [closure (lex-env code pc) (elc (void) '() (extend-lex-env args lex-env) k code pc)]
      [primitive (f) (apply-cont k (apply f (vector->list args)))]
      [callcc (k) (apply-cont k (vector-ref args 0))]
      [else (apply-proc *error-handler* (vector "Invalid procedure" proc) k)])))
```

Figure 9: Revised abstract machine for linearized code

Datum	*Type*	*Subtype*	*Value*
fixnum n	imm	fixnum	n in 2's complement
char c	imm	char	ASCII code of c
#t	imm	true	?
#f	imm	false	?
void	imm	void	?
undefined	imm	undefined	?
eof-object	imm	eof-object	?
()	imm	null	?
$(a\ .\ b)$	fixed	pair	$[a\ b]$
$\#(v_1\ \ldots\ v_n)$	vector	vec	[(fixnum n) $v_1\ \ldots\ v_n$]
"$c_1 \ldots c_n$"	vector	string	[(fixnum n) (char c_1) ... (char c_n)]
symbol $s_1 \ldots s_n$	vector	symbol	[(fixnum n) (char s_1) ... (char s_n)]
primitive n	imm	primitive	n
closure	fixed	closure	[*lex-env code pc*]
callcc	fixed	callcc	[k]
done-cont	imm	done-cont	?
save-cont	vector	save-cont	[(fixnum 5) *val* lex-env k code pc*]
int-cont	vector	int-cont	[(fixnum 6) *acc val* lex-env k code pc*]

Figure 10: Data representation

4 Making Heap Usage Explicit

The previous model uses Scheme's run-time heap and garbage collector as its memory model. This dependence must be made explicit so that the interface can be adapted to the hardware heap and garbage collector. The first step was to define the specific data representations for the various record structures used by the machine.

4.1 Data Representation

All machine data are represented as 32-bit *citations*. A citation's upper 8 bits determine the object type. The remaining 24 bits hold either an address to the data or an immediate value, depending on the type. In the software model of the heap, the 8-bit object type is classified by a 3-bit type and a 5-bit subtype. Figure 10 lists the object representations. A value of ? indicates a "don't-care" value. When the value field is an address, the contents of the cells pointed to by that address are indicated sequentially in brackets.

4.2 Primitives

The design philosophy for the Scheme Machine's primitives was to find a minimal set of operations from which most other Scheme primitives could be built. Low-level I/O is provided by the primitives *@read-port* and *@write-port*, which read and write 32-bit citations from a possible 2^{24} ports. Low-level memory operations are provided by the primitives *@read-obj*, *@write-obj*, and *@gc*. The

@execute primitive is used to execute a piece of code generated at run-time.

The addition and subtraction primitives check for arithmetic overflow. The error handler is invoked upon overflow with the truncated result as the first argument. Except for arithmetic overflow, the primitives provide no error checking. It is up to the compiler to insert necessary run-time error checks. The CPU will catch the following errors:

- arithmetic overflow during an arithmetic operation
- referencing a variable that contains the "unbound" value
- calling a non-procedure

The complete list of machine primitives is given in Appendix A.

4.3 Heap Allocation and Garbage Collection

The next step was to modify the machine to use citations rather than Scheme values. This modification entailed making allocations and garbage collections explicit. A software model of the hardware heap was used to test this change. In order to perform garbage collection correctly, every call to the allocator needed to include a list of all live citations so that a root set could be provided to the stop-and-copy garbage collector. The calls also had to include a continuation procedure to be called when the allocation (and possibly a garbage collection) completes. The continuation is a procedure (**lambda** (*new-citation . roots*) ...) that takes the new allocation and the (possibly modified) values of the root set.

At the same time we converted to citations, we converted the machine into one large **letrec** expression that incorporated the separate helper functions, *e.g.*, *lookup-lex-env*, *assign-lex-env!*, *extend-lex-env*, *apply-cont*, and *apply-proc*. With the DDD framework in mind, we added four temporary registers to the design to assist us in coding the helper functions. In this way, we could eventually standardize all the states of the machine to take as arguments the entire register set.

A great advantage of using the executable software model during this process was that we could always run the machine to test its behavior. This capability greatly reduced the time needed to debug the design, since we would typically catch bugs before they would propagate further into the design.

After we converted to citations and explicit allocation and garbage collection, we could use the software model for the heap. After this stage was debugged, we simply switched interface functions so that we could run the software CPU off the hardware heap.

Here is the protocol (from the CPU perspective) for performing a heap allocation:

1. Put the tag and size on the bus, and assert the *alloc* signal to request an allocation.
2. Continue to put the tag and size on the bus and assert *alloc* while waiting for the allocator to reply with either *gc-needed?* or *alloc-done?*. If a garbage collection is needed, do the following:
    a. Save the current state (registers) on the heap (while still asserting *alloc*).
    b. Unassert *alloc* to signal the start of garbage collection.
    c. Wait until the allocator asserts *alloc-ready?*. When it does, reload the current state from the heap.

d. Put the tag and size on the bus again, and assert *alloc*.

e. Continue to put the tag and size on the bus and assert *alloc* while waiting for the allocator to reply with either *gc-needed?* or *alloc-done?*. If a garbage collection is needed, signal an error because there is no more memory.

3. Assert *alloc* and *read-avail*, and read the new citation's address from the bus.

4. Unassert *alloc* and wait for *alloc-ready?*.

5 Using DDD—Design I

At this point the machine was nearly in a form acceptable to DDD. Our interrupt, memory, and port operations still used assignments and **begin** expressions. We abstracted the interrupts-enabled flag, the memory, and the ports as three new registers. With these as registers, assignment began a simple state change. We also needed to serialize the operations on the memory, since the physical memory cannot do more than one read or write per machine state.

Now the hardware design issues began to drive our work. For example, we wanted to use one or maybe two ALUs, so we serialized all the arithmetic operations, which resulted in a large increase in the number of states. Each of the machine registers is loaded from a multiplexer that gets inputs from all possible sources of data for the register. Our target technology was Actel-2 gate arrays, which have 4-to-1 multiplexers as a basic unit. Consequently, our goal was to have registers that needed at most 4 inputs. With two modules we could provide 7 inputs; with 3, 10; with 4, 13; and with 5, 16. The naïve design had large numbers of inputs to each register, which resulted in a huge number of modules just for the register file, not including what would be needed to implement the controller.

After tweaking the design for about a month, we decided it would be wiser to redesign the machine from the last specification phase before we made heap usage explicit.

6 Using DDD-Design II

Extremely familiar with the entire Scheme Machine specification, I designed a minimal datapath that would support all the operations needed by the machine. I eliminated some decoding circuitry by choosing to encode the allocator's continuations as machine states. In this way, invoking the continuation was simply a transfer to the state given in the continuation register. (Note that the allocator's continuation register is *separate* from the interpreter's continuation register, *k*. The allocator's continuation register gives the continuation after a heap allocation, for example.) ([3] describes the use of continuations in hardware design with the Scheme Machine as the primary example.) Furthermore, I encoded the instructions as machine states so that the execute phase could simply transfer to the state indicated by the instruction. Later this aided in debugging the hardware since I could give an instruction to go to any machine state. The primitives were also encoded by the states.

One can see that the *val**, *lex-env*, and *code* registers are always vectors. Consequently, I could implement these registers with 24 bits instead of 32, since the tag would always be constant. Furthermore, *pc* is always a fixnum, so it needs only 24 bits as well.

Address	*Data*
0	[imm fixnum 9] (length of this vector)
1	*acc*
2	[vector vec *val**]
3	[vector vec *lex-env*]
4	*k*
5	[vector vec *code*]
6	[imm fixnum *pc*]
7	**error-handler**
8	**interrupt-handler**
9	[imm fixnum 0] (zero vector)

Figure 11: Fixed heap locations

Realizing that many of the operations involve simple arithmetic address computations, I added a 24-bit address register, *addr*, and a simple 24-bit adder. The program counter, *pc*, also was also given a 24-bit adder. A single 24-bit ALU was allocated to the accumulator, *acc*.

To aid in memory allocations, I added an 8-bit tag register, *tag*, to hold the requested type of allocation. I added an 8-bit allocation continuation register, *cont*, to tell the allocator which state to go to when the allocation completed. The address register, *addr*, would contain the size of the requested allocation.

I designed a single 32-bit bidirectional bus to connect the Scheme Machine to the outside world. Using time-division multiplexing already provided by the heap hardware, I could alternatively place and read addresses and data to and from the bus. All memory and port operations share the same bus.

A simple interrupt mechanism is provided via an external signal, **interrupt**, and the internal interrupts-enabled flip-flop, *ie*. In addition, one can halt the machine by asserting the *halt* signal. It causes the Scheme Machine to complete the current instruction, dump its registers to the heap, and halt. The machine can be restarted by asserting the *go* signal. When starting, the Scheme Machine loads its registers from the heap and then begins instruction execution.

I fixed the first several absolute heap addresses to be used for register saves during startup and halting, garbage collection, and complex operations that need a temporary storage location for a register or two. The error handler and interrupt handler were placed at absolute heap locations as well. The hardware heap expects a vector to start at location zero. Figure 11 gives the layout of the vector. The zero at location 9 is used to generate an empty vector. Placing 9 in *val**, for example, makes this register contain an empty vector. This technique is a slight hack, but it is needed to clear *val** to the empty vector without performing an allocation.

Whereas the previous design was driven first by the specification, the second design was driven by the datapath. Using the previous specification as a guide, I hand-coded the new specification to match the datapath I desired. The coding process led to minor refinements of the datapath. The resulting design was small enough to be implemented on 3 Actel FPGAs and 10 PALs.

Figure 12 shows a high-level view of the Scheme Machine datapath.

6.1 Scheme Machine Specification

The specification was written for the DDD system. In order to remove some of the tedious syntax of state transitions where most of the registers do not change, I developed some macros that simplified the description of state transitions.

The **$regs** macro is used to specify the common set of registers at every state. (The printed specification in Appendix B has this macro expanded.)

```
(extend-syntax ($regs)
  ((_ e1 e2 ...)
   (with ((r *regs*))
     (lambda r e1 e2 ...))))

(define *regs*
  '(acc val* lex-env k code pc tag addr ie port mem cont))
```

The **$next** macro is used to specify a state transition, where the registers that change are specified in a let-style notation. The global variable **reg-defaults** is used to specify the architecture for retaining the current value of a register. (The printed specification has this macro displayed as "⇒".)

```
(extend-syntax ($next)
  ((_ state (reg exp) ...)
   (with (((e1 ...) (fill-regs '((reg . exp) ...))))
     (state e1 ...))))

(define fill-regs
  (lambda (l)
    (for-each (lambda (pair)
                (if (not (memq (car pair) *regs*))
                    (error 'fill-regs "unknown register ~s" (car pair))))
              l)
    (let ([l (append l (map cons *regs* *reg-defaults*))])
      (map (lambda (reg) (cdr (assq reg l))) *regs*))))

(define *reg-defaults*
  '((make-cite (obj.tag acc) (alu-out (alu a+0 (obj.ptr acc) ? ff)))
    val* lex-env k code (add pc (c24 0) ff) tag (add addr (c24 0) ff)
    ie port mem cont))
```

The **assign** macro is simply a synonym for **let** as far as the software specification is concerned. When this macro is expanded to produce the input to DDD, however, it produces a fresh state. **assign** is used to specify another state whose body is the body of the **assign** clause. The register assignment clauses determine the transition to the new state. Figure 13 shows a portion of a machine description before and after expansion of the **assign** macro.

The states that perform the heap allocation and garbage collection protocol were designed by hand. The garbage collection primitive, *@gc*, simply requests an allocation of a vector too large to fit in the physical memory, which will trigger a garbage collection. When the collection finishes,

control transfers to state p-gc-1, which reloads the registers from the heap and then applies the current continuation, k. Figure 14 gives a high-level ASM chart for the allocation protocol.

The complete Scheme Machine specification is given in Appendix B.

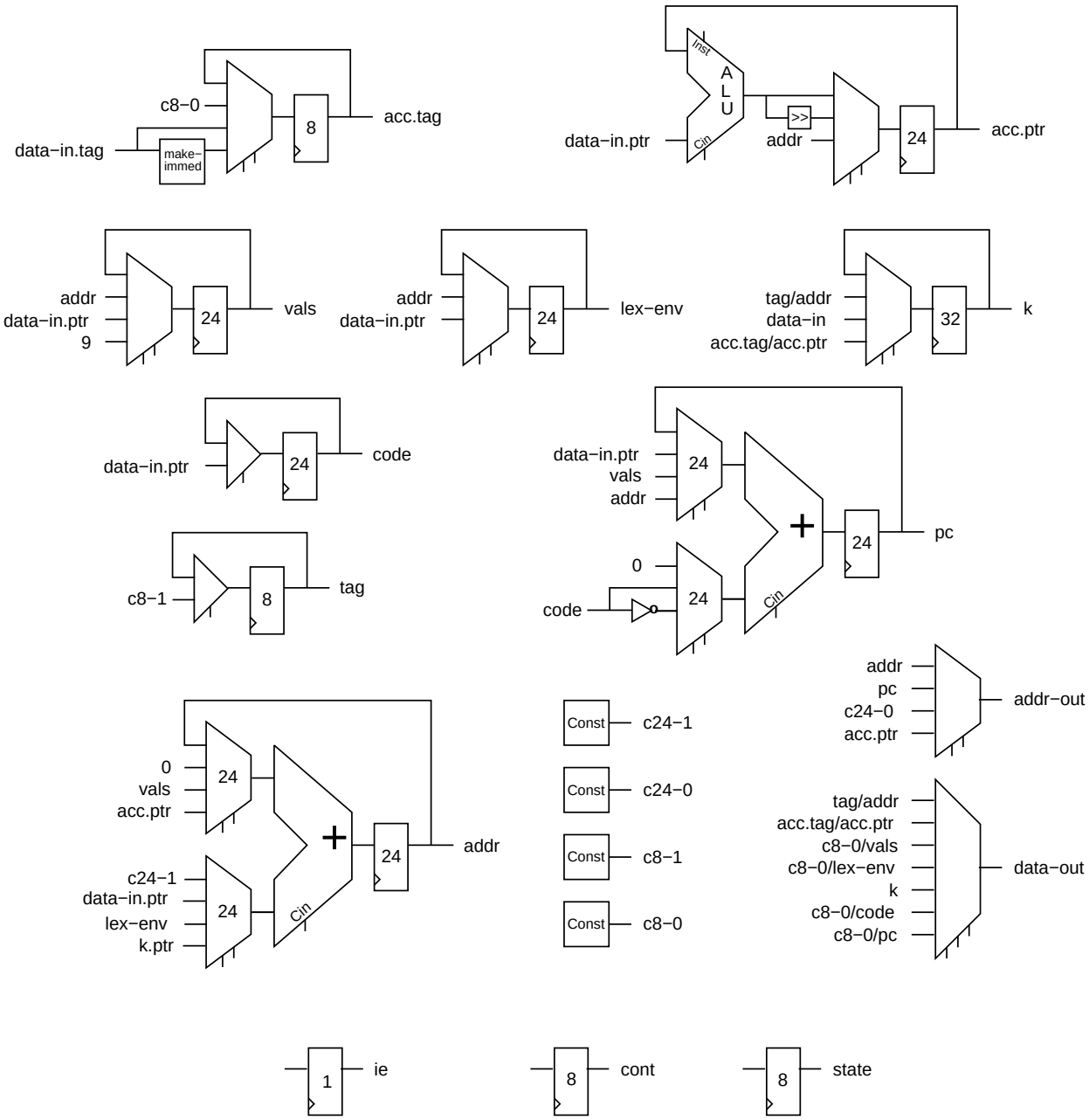

Figure 12: Scheme Machine Datapath

Before:

```
[state-1 (lambda (r1 r2)
           (assign ((r1 (+ r1 r2)))
                   (state-2 r1 0)))]
[state-2 (lambda (r1 r2)
           (state-1 r1 (+ r1 r2)))]
```

After:

```
[state-1 (lambda (r1 r2)
           (state-3 (+ r1 r2) r2))]
[state-2 (lambda (r1 r2)
           (state-1 r1 (+ r1 r2)))]
[state-3 (lambda (r1 r2)
           (state-2 r1 0))]
```

Figure 13: Expansion of **assign** macro

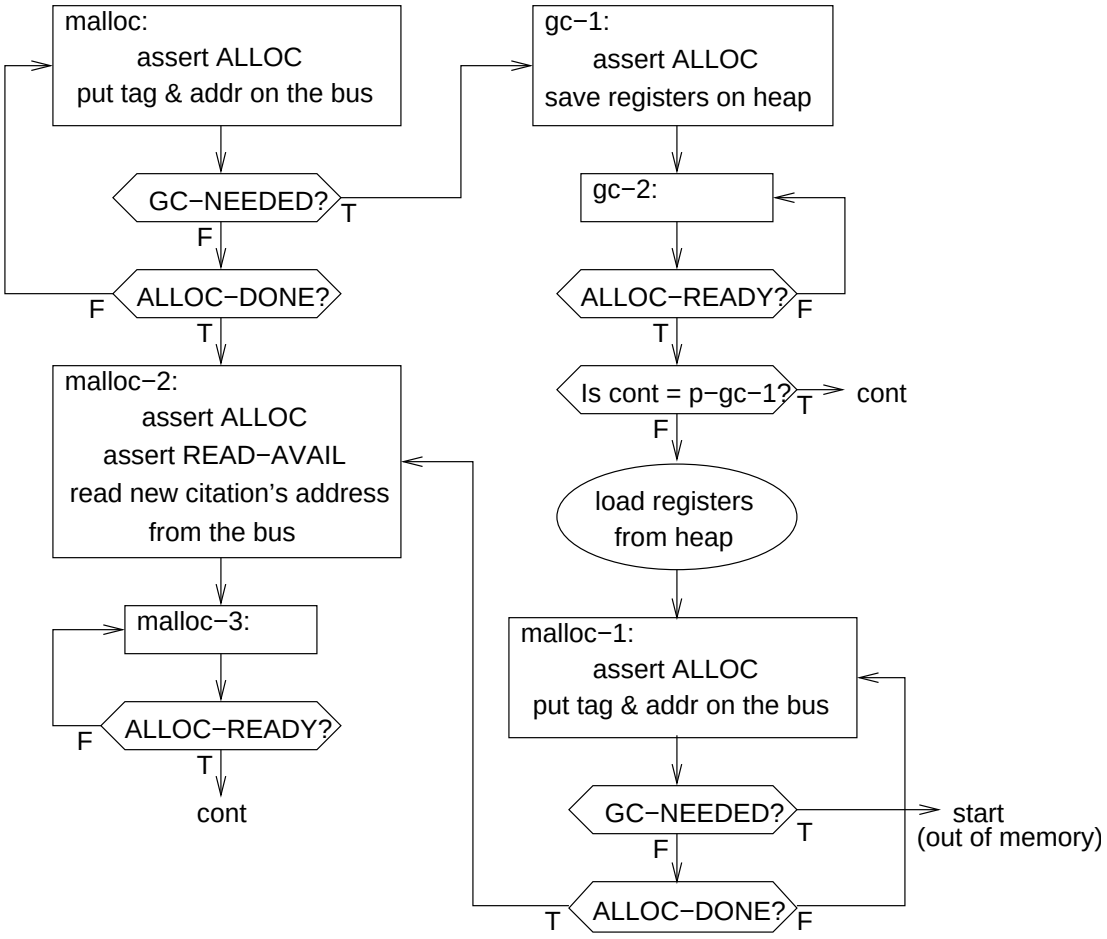

Figure 14: Allocation protocol

6.2 Generating the Stream Equations

The transformations from the Scheme Machine specification to an appropriate system of stream equations occurs in the following steps (the files can be found in /u/ddd/SM/deriv):

1. The macro-expanded Scheme Machine specification, machine.ss, is converted into a sequential system of streams (seq.ss) and a selector for these streams (seqsel.ss) by the script seq.s (see Appendix C.1). In addition, it renames the start state to a-start so that it will be assigned code 0, the default reset state.

2. The ALU and the adders for *pc* and *addr* are factored from the sequential system by the script alu.s (see Appendix C.2). In order to facilitate the handling of the tag and address parts of the accumulator, acc is split into acc.tag and acc.ptr.

3. The heap allocator is factored by the script alloc.s (see Appendix C.3).

4. The memory and ports are factored by the script mem.s (see Appendix C.4). The memory and port address buses are merged into ADDR-OUT, and the memory, port, and allocator data buses are merged into DATA-IN and DATA-OUT.

5. Because they can be implemented more efficiently as a group rather than as individual constants, the 8-bit, 24-bit, and 32-bit constants are factored out by the script const.s (see Appendix C.5). The script uses a special program that attempts to find the minimal number of constant boxes. Unfortunately, this optimization problem is NP-complete, so it uses a heuristic that worked well for this particular machine.

6.3 Generating Boolean Equations for the Predicates

The *predicates* are the test parts of **if** expressions in the DDD specification. Unfortunately DDD did not provide a function to generate boolean equations for the predicates. Consequently, the equations were coded by hand. Because the transformation system provided no support for the predicates, these hand-coded equations were checked (by hand) several times, and later a simulated bit-level version of the Scheme Machine was executed on some small programs.

The predicates were split into two groups: trivial and non-trivial. The trivial predicates were either single bits or a simple sum of products of a couple bits. They can be found in trivial-preds.eqn. These predicates will not be generated as separate signals; instead, they will be "inlined" into the remaining equations using SIS. The remaining predicates, preds.eqn, were minimized with Espresso. P36 was generated bit-wise for ease of specification in sum-of-products form. The script fold-preds combines these back into the single signal, P36.

6.4 Bitslicing the Datapath

The script sel.s (see Appendix C.6) splits the single sequential system for the Scheme Machine into separate sequential systems for each of the components of the data path. The multiplexor selector equations are extracted by DDD at the same time. These equations are minimized by Espresso and written to SEL.eqn.

6.5 Generating the Boolean Equations for the Control Signals

The control signals consist of the signals to the memory, allocator, ports, ALU, adders, interrupts-enabled flip-flop, state generator, and *cont* register. They are extracted, minimized by Espresso, and written to GENERATORS.eqn by the next.s script (see Appendix C.7).

6.6 Generating the Actel Modules

The file actel.ss contains the definitions that map bit streams to Actel 4-to-1 multiplexors. The layout chosen is based on the number of inputs. Figure 15 shows the layout for streams with two to ten inputs, inclusively. The binary numbers by the output of each block indicate the sequence of control signals needed to select the inputs in numeric order.

The script actel.s (see Appendix C.8) generates the Actel module descriptions for the datapath, mods.actel. The multiplexors are run through a simple optimizer that looks for constant input patterns; this is especially beneficial for the constant streams, many of which simply collapse into a wire. The optimizer also combines identical gates, which is useful for the 7-input case above. Appendix D gives the source code for the optimizer.

The Actel-2 24-bit ALU and adders were designed by Willie Hunt, and Shyam Pullela verified them. alu.actel contains the ALU, adder-pc.actel contains the adder for *pc*, and adder-addr.actel contains the adder for *addr*.

I hand-coded the Actel-2 modules that generate the zero flag for the ALU and invert the *code* register for input into the *pc* adder. These modules are in comp.actel. I also hand-coded the modules for the memory interface in mem.actel.

Through a process of trial and error, I partitioned the boolean equations for the control and predicates into four groups: those to be implemented on the main Actel-2 chip main.blif, those for the first of the smaller Actel-1 chips rest1.blif, those for the second Actel-1 chip rest2.blif, and those to be implemented in PALs pals.blif. The script pal.s does the partitioning.

SIS has a built-in command, act-map, that maps boolean equations to Actel-1 modules in a fairly efficient manner. The modules for the two smaller Actel-1 chips were generated in this way to rest1.actel and rest2.actel.

I designed a "genlib" for SIS that describes the Actel-2 modules. Because the SIS map command is designed toward nand and nor operations rather than multiplexors, I listed the various possible nand, nor, and xor gates that can be built from a single Actel-2 module. The resulting library, actel.genlib, worked fairly well in SIS. I used this library to generate the Actel-2 modules from main.blif to main.actel.

The module descriptions are processed by a simple translator that provided compatibility with WorkView software on the PC. act2wv.ss contains the Scheme source. The *.actel descriptions are converted to *.act descriptions.

I combined the modules for the main chip, mods.act, main.act, comp.act, mem.act, adder-addr.act, adder-pc.act, and alu.act into sm.act.

Because of physical fan-out restrictions, some buffers needed to be added to the descriptions for the chips. The file fanout.ss (see Appendix E) contains the Scheme source that adds buffers in a hierarchical fashion.

The Actel modules were converted into WorkView format by a program designed by Willie Hunt, act2wv.ss. Because it was written for the previous release of WorkView, I needed to add a

new header that changed with the new release, and I needed to shorten some of the names. The script vlupdate performs this process.

The I/O pin descriptions were written by hand in sm.pin, rest1.pin, and rest2.pin. They were converted into WorkView form by pin2wv.ss.

The resulting WorkView files were used on the PC both for simulation and for burning the chips. In order to get the main chip to route, we needed to place the output pins according to the intuition Willie Hunt developed when helping Bhaskar Bose route the FM9001 chip.

6.7 Simulation

The DDD specification was executable throughout the design process, which was extremely valuable in debugging the design. Almost all of the bugs in the design were found at this stage.

After I generated the Actel module descriptions for the entire design, I wrote an Actel simulator generator, /u/ddd/SM/deriv/simulator/act2c.ss (see Appendix F). It topologically sorts the circuit (and notifies the user of any asynchronous cycles) and then generates a C program that simulates the circuit. Using this simulator, I was able to simluate the Scheme Machine at the bit level. Although it ran about 1000 times slower, I was able to test small programs.

I also tooled the simulator to write a set of ViewSim test vectors to be used on the PC to simulate behavior at the Actel module level. As I executed programs at the C simulator level, the simulator generated test vectors for ViewSim, which I then ran on the Actel simulator. This stage of simulation uncovered errors in the representation file, sm.rep, usually just typos. I also found errors in actel.genlib and the mux-generation software.

References

[1] Bhaskar Bose. DDD—A transformation system for Digital Design Derivation. Technical Report 331, Indiana University, Computer Science Department, May 1991.

[2] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1992.

[3] M. Esen Tuna, Steven D. Johnson, and Robert G. Burger. Continuations in hardware-software codesign. In *Proceedings of the International Conference on Computer Design (ICCD)*. IEEE, October 1994. To appear. Also published as Tech Report # 409, Computer Science Department, Indiana University.

A Scheme Machine Primitives

Primitive	*Result*
(*@fixnum? x*)	#t if x is a fixnum, #f otherwise
(*@positive? x*)	#t if ptr(x) > 0, #f otherwise
(*@negative? x*)	#t if ptr(x) < 0, #f otherwise
(*@zero? x*)	#t if ptr(x) $= 0$, #f otherwise
(*@odd? x*)	#t if ptr(x) is odd, #f otherwise
(*@even? x*)	#t if ptr(x) is even, #f otherwise
(*@eq a b*)	#t if ptr(a) $=$ ptr(b), #f otherwise
(*@ne a b*)	#t if ptr(a) $\neq$ ptr(b), #f otherwise
(*@gt a b*)	#t if ptr(a) $>$ ptr(b), #f otherwise
(*@ge a b*)	#t if ptr(a) $\geq$ ptr(b), #f otherwise
(*@lt a b*)	#t if ptr(a) $<$ ptr(b), #f otherwise
(*@le a b*)	#t if ptr(a) $\leq$ ptr(b), #f otherwise
(*@add a b*)	[imm(tag(a)) ptr(a)+ptr(b)], where imm(t) $= t \wedge$ #b0110000
(*@sub a b*)	[imm(tag(a)) ptr(a)–ptr(b)]
(*@add1 x*)	[imm(tag(x)) ptr(x)+1]
(*@sub1 x*)	[imm(tag(x)) ptr(x)−1]
(*@asr x*)	[imm(tag(x)) ptr(x) >> 1], where >> is arithmetic right shift
(*@boolean? x*)	#t if x is #t or #f, #f otherwise
(*@not x*)	#t if $x \neq$ #t, #f otherwise
(*@bitand a b*)	[imm(tag(a)) ptr(a)$\wedge$ptr(b)]
(*@bitor a b*)	[imm(tag(a)) ptr(a)$\vee$ptr(b)]
(*@bitxor a b*)	[imm(tag(a)) ptr(a)$\oplus$ptr(b)]
(*@char? x*)	#t if x is a char, #f otherwise
(*@read-char p*)	citation from input port ptr(p)
(*@write-char p c*)	void; write c to output port ptr(p)
(*@eof-object? x*)	#t if x is the eof-object, #f otherwise
(*@null? x*)	#t if x is (), #f otherwise
(*@pair? x*)	#t if x is a pair, #f otherwise
(*@cons a b*)	(a . b)
(*@vector? x*)	#t if x is a vector, #f otherwise
(*@make-vector l*)	vector of length ptr(l) filled with unknown immediate values
(*@string? x*)	#t if x is a string, #f otherwise
(*@make-string l*)	string of length ptr(l), filled with unknown characters
(*@symbol? x*)	#t if x is a symbol, #f otherwise
(*@string–>symbol s*)	[string ptr(s)] Note that this is just a type cast
(*@symbol–>string s*)	[symbol ptr(s)] Note that this is just a type cast
(*@procedure? x*)	#t if x is a closure, callcc, or primitive, #f otherwise
(*@eq? a b*)	#t if tag(a) $=$ tag(b) and ptr(a) $=$ ptr(b), #f otherwise
(*@apply proc argv*)	applies the procedure *proc* to the argument *vector argv* and returns the result
(*@abort*)	void; halts the machine
(*@call/cc proc*)	calls the procedure *proc* with one argument, the current continuation.
(*@void*)	void

(*@read-obj x offset*)	object at heap location ptr(x)+ptr(*offset*)
(*@write-obj x offset val*)	void; writes object *val* to heap location ptr(x)+ptr(*offset*)
(*@gc*)	void; forces a garbage collection
(*@argc*)	[fixnum n], where n is the vector length of *lex-env*.
(*@execute code env*)	executes the code vector *code* with initial environment *env* and returns the result
(*@interrupts-enabled?*)	#t if interrupts are enabled, #f otherwise
(*@interrupts-enabled x*)	#f if x is false, #t otherwise; the interrupts-enabled flip-flop is set or reset accordingly.
(*@interrupt-handler*)	the current interrupt handler
(*@set-interrupt-handler proc*)	void; the interrupt handler is set to *proc*, which should be a procedure (**lambda** () ...) that's called when an interrupt occurs.
(*@error-handler*)	the current error handler
(*@set-error-handler proc*)	void; the error handler is set to *proc*, which should be a procedure (**lambda** (x y) ...), where x is the error number and y is the value of *acc* at the time of error.

B Scheme Machine DDD Specification

```
(define scheme-machine
  (λ (go interrupt halt)
    (letrec
        ([start ; Wait for GO signal
          ; Pre: none
          (λ (acc val* lex-env k code pc tag addr ie port mem cont)
           (if (go)
               (⇒ load (addr (add (c24 0) (c24 0) tt)) (cont fetch))
               (⇒ start)))]

         [load ; Load all registers from the heap
          ; Pre: addr valid, cont = state to go after loading registers
          (λ (acc val* lex-env k code pc tag addr ie port mem cont)
           (⇒ load-1 (addr (add addr (c24 0) tt)) (acc (alu-memrd mem addr))))]

         [load-1 ; Load all but acc from heap
          ; Pre: addr valid, cont = next-state
          (λ (acc val* lex-env k code pc tag addr ie port mem cont)
           (assign ((val* (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
            (assign ((lex-env (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
             (assign ((k (memrd mem addr)) (addr (add addr (c24 0) tt)))
              (assign ((code (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
               (⇒ cont (pc (add (obj.ptr (memrd mem addr)) code tt)))))))))]

         [save ; Save all registers on the heap
          ; Pre: addr valid, cont = state to go after dumping registers
          (λ (acc val* lex-env k code pc tag addr ie port mem cont)
```

```
  (⇒ save-1 (mem (memwr mem addr acc)) (addr (add addr (c24 0) tt))))]

[save-1 ; Save all registers except acc on the heap
 ; Pre: addr valid, cont = next-state
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((mem (memwr mem addr (hcite <vector> <vec> val*)))
           (addr (add addr (c24 0) tt)))
   (assign ((mem (memwr mem addr (hcite <vector> <vec> lex-env)))
            (addr (add addr (c24 0) tt)))
    (assign ((mem (memwr mem addr k)) (addr (add addr (c24 0) tt)))
     (assign ((mem (memwr mem addr (hcite <vector> <vec> code)))
              (addr (add addr (c24 0) tt)) (pc (add pc (complement code) ff)))
      (⇒ cont (mem (memwr mem addr (hcite <imm> <fixnum> pc)))))))))]

[fetch ; Reads opcode into cont register, bumps PC, and goes to opcode
 ; Pre: valid PC
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (if (halt)
      (⇒ save (addr (add (c24 0) (c24 0) tt)) (cont start))
      (if (and (interrupt) ie)
          (⇒ malloc (tag (c8 (make-tag <vector> <int-cont>)))
             (addr (add (c24 0) (c24 5) tt)) (cont interrupt-1))
          (assign ((cont (convert (memrd mem pc))) (pc (add pc (c24 0) tt)))
            (⇒ cont))))))]

[lit-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ fetch (acc (alu-memrd mem pc)) (pc (add pc (c24 0) tt))))]

[varref-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ lookup (pc (add pc (c24 0) tt)) (cont varref-inst-1)
      (addr (add (c24 0) lex-env tt)) (acc (alu-memrd mem pc))))]

[varref-inst-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (if (same-type? m <imm> <undefined>)
        (⇒ error (addr (add (c24 0) (c24 unbound-error) ff)) (acc (alu-void)))
        (⇒ fetch (acc (hcite (obj.tag m) (alu-out (alu b+0 ? (obj.ptr m) ff))))))))]

[varassign-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((mem (memwr mem (c24 acc-loc) acc)))
    (⇒ lookup (pc (add pc (c24 0) tt)) (cont varassign-inst-1)
        (addr (add (c24 0) lex-env tt)) (acc (alu-memrd mem pc)))))]

[varassign-inst-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem (c24 acc-loc))))
    (⇒ fetch (mem (memwr mem addr acc)) (acc (alu-void)))))]
```

```
[if-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (if (true-value? acc)
      (⇒ fetch (pc (add pc (c24 0) tt)))
      (⇒ fetch (pc (add (obj.ptr (memrd mem pc)) code tt)))))]

[proc-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <fixed> <closure>))) (cont proc-inst-1)))]

[proc-inst-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((lex-env addr) (addr (add addr (c24 0) tt))
           (mem (memwr mem addr (hcite <vector> <vec> lex-env))))
   (assign ((mem (memwr mem addr (hcite <vector> <vec> code)))
            (addr (add addr (c24 0) tt)))
    (assign ((acc (alu-memrd mem pc)) (pc (add pc (c24 0) tt)))
     (assign ((mem (memwr mem addr acc))
              (addr (add (c24 0) lex-env ff)))
      (⇒ fetch (acc (hcite <fixed> <closure> addr))
           (lex-env (obj.ptr (memrd mem addr)))))))))]

[save-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <save-cont>)))
      (addr (add (c24 0) (c24 5) ff)) (cont save-inst-1)))]

[save-inst-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite <vector> <save-cont> addr)) (addr (add addr (c24 0) tt)))
   (assign ((mem (memwr mem addr (hcite <vector> <vec> val*)))
            (val* (c24 zero-loc)) (addr (add addr (c24 0) tt)))
    (assign ((mem (memwr mem addr (hcite <vector> <vec> lex-env)))
             (addr (add addr (c24 0) tt)))
     (assign ((mem (memwr mem addr k)) (k acc) (addr (add addr (c24 0) tt)))
      (assign ((mem (memwr mem addr (hcite <vector> <vec> code)))
               (addr (add addr (c24 0) tt)))
       (assign ((acc (alu-memrd mem pc)) (pc (add pc (c24 0) tt)))
        (⇒ fetch (mem (memwr mem addr acc)) (acc (alu-void)))))))))))]

[args-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <vec>)))
      (addr (add (c24 0) (obj.ptr (memrd mem pc)) ff))
      (pc (add pc (c24 0) tt)) (acc (alu-void)) (cont args-inst-1)))]

[args-inst-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ fetch (val* addr)))]

[push-inst
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
```

```
  (assign ((addr (add val* (obj.ptr (memrd mem pc)) tt)) (pc (add pc (c24 0) tt)))
    (⇒ fetch (mem (memwr mem addr acc)) (acc (alu-void)))))]

[interrupt-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ save (addr (add addr (c24 0) tt)) (cont interrupt-2)))]

[interrupt-2
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem (c24 int-loc))) (addr (add addr (c24 −6) ff)))
    (⇒ apply-proc (ie ff) (k (hcite tag addr)))))]

[lookup
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((r (alu a-0 (obj.ptr acc) ? ff)))
    (if (n-flag r)
        (⇒ cont (pc (add pc (c24 0) tt)) (acc (hcite (obj.tag acc) (alu-out r)))
            (addr (add addr (obj.ptr (memrd mem pc)) ff)))
        (⇒ lookup (acc (hcite (obj.tag acc) (alu-out r)))
            (addr (add (c24 0) (obj.ptr (memrd mem addr)) tt)))))))]

[extend-env
 ; addr → new rib
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  ; [pc-loc] ← pc, pc ← rib+1
  (assign ((mem (memwr mem (c24 pc-loc) (hcite <imm> <fixnum> pc)))
           (pc (add addr (c24 0) tt)))
    ; [rib+1] ← lex-env, lex-env ← rib, pc ← rib, addr ← rib+1
    (assign ((mem (memwr mem pc (hcite <vector> <vec> lex-env)))
             (lex-env addr) (pc (add addr (c24 0) ff)) (addr (add addr (c24 0) tt)))
      ; [k-loc] ← k
      (assign ((mem (memwr mem (c24 k-loc) k)))
        ; acc ← [rib], addr ← rib+2, pc ← val*+1
        (⇒ extend-env-1 (acc (alu-memrd mem pc))
            (addr (add addr (c24 0) tt)) (pc (add val* (c24 0) tt))))))))]

[extend-env-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((r (alu a-0 (obj.ptr acc) ? ff)))
    (if (n-flag r)
        (assign ((acc (hcite (obj.tag acc) (alu-out r)))
                 (val* (c24 zero-loc)) (k (memrd mem (c24 k-loc))))
          (⇒ fetch (pc (add (obj.ptr (memrd mem (c24 pc-loc))) (c24 0) ff))))
        (assign ((k (memrd mem pc)) (pc (add pc (c24 0) tt))
                 (acc (hcite (obj.tag acc) (alu-out r))))
          (⇒ extend-env-1 (mem (memwr mem addr k)) (addr (add addr (c24 0) tt))))))))]

[apply-proc
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (if (same-type? acc <fixed> <closure>)
      (assign ((addr (add (obj.ptr acc) (c24 0) ff)))
       (assign ((lex-env (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
```

```
        (assign ((code (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
         (assign ((pc (add (obj.ptr (memrd mem addr)) code tt))
                  (addr (add val* (c24 0) ff)))
           (⇒ malloc (cont extend-env) (tag (c8 (make-tag <vector> <vec>)))
                (addr (add (c24 0) (obj.ptr (memrd mem addr)) tt)))))))
      (if (same-type? acc <imm> <primitive>)
          (assign ((cont (convert acc)) (addr (add val* (c24 0) tt)))
            (⇒ cont))
          (if (same-type? acc <fixed> <callcc>)
              (⇒ callcc (addr (add val* (c24 0) tt)) (k (memrd mem (obj.ptr acc))))
              (⇒ error (addr (add (c24 0) (c24 procedure-error) ff)))))))]

[callcc
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (alu-memrd mem addr))))]

[apply-cont
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (if (same-type? k <vector> <int-cont>)
      (⇒ load (addr (add (c24 0) (obj.ptr k) tt)) (cont fetch))
      (if (same-type? k <vector> <save-cont>)
          (⇒ load-1 (addr (add (c24 0) (obj.ptr k) tt)) (cont fetch))
          ; Else it's a done-cont
          (⇒ save (addr (add (c24 0) (c24 0) tt)) (cont start)))))]

[malloc ; Allocate
 ; Pre: tag+addr specify what to allocate,
 ; Post: addr contains the new address
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((a (alloc tt ff (make-cite tag addr))))
    (if (gc-needed? a)
        (⇒ gc-1)
        (if (alloc-done? a)
            (⇒ malloc-2)
            (⇒ malloc)))))]

[malloc-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((a (alloc tt ff (make-cite tag addr))))
    (if (gc-needed? a)
        (⇒ start (mem (panic mem)))
        (if (alloc-done? a)
            (⇒ malloc-2)
            (⇒ malloc-1)))))]

[malloc-2
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((a (alloc tt tt ?)))
    (⇒ malloc-3 (addr (add (c24 0) (mem-avail a) ff)))))]

[malloc-3
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
```

```
  (let ((a (alloc ff ff ?)))
    (if (alloc-ready? a)
        (⇒ cont)
        (⇒ malloc-3)))))]

[gc-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((a (alloc tt ff ?)))
   (assign ((mem (memwr mem (c24 acc-loc) acc)))
    (let ((a (alloc tt ff ?)))
     (assign ((mem (memwr mem (c24 val*-loc) (hcite <vector> <vec> val*))))
      (let ((a (alloc tt ff ?)))
       (assign ((mem (memwr mem (c24 lex-env-loc) (hcite <vector> <vec> lex-env))))
        (let ((a (alloc tt ff ?)))
         (assign ((mem (memwr mem (c24 k-loc) k)))
          (let ((a (alloc tt ff ?)))
           (assign ((mem (memwr mem (c24 code-loc) (hcite <vector> <vec> code)))
                    (pc (add pc (complement code) ff)))
            (⇒ gc-2
               (mem (memwr mem (c24 pc-loc) (hcite <imm> <fixnum> pc))))))))))))))))]

[gc-2
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((a (alloc ff ff ?)))
    (if (alloc-ready? a)
        (if (eqv? cont p-gc-1)
            (⇒ cont)
            (assign ((acc (alu-memrd mem (c24 acc-loc))))
             (assign ((val* (obj.ptr (memrd mem (c24 val*-loc)))))
              (assign ((lex-env (obj.ptr (memrd mem (c24 lex-env-loc)))))
               (assign ((k (memrd mem (c24 k-loc))))
                (assign ((code (obj.ptr (memrd mem (c24 code-loc)))))
                 (⇒ malloc-1 (pc (add (obj.ptr (memrd mem (c24 pc-loc))) code tt)))))))))
        (⇒ gc-2))))]

[error
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <vec>))) (addr (add (c24 0) (c24 2) ff))
     (cont error-1) (pc (add addr (c24 0) ff))))]

[error-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((addr (add addr (c24 0) tt)) (val* addr))
   (assign ((mem (memwr mem addr (hcite <imm> <fixnum> pc)))
            (addr (add addr (c24 0) tt)))
    (assign ((mem (memwr mem addr acc)))
     (⇒ apply-proc (acc (alu-memrd mem (c24 err-loc))))))))]

[p-fixnum?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <imm> <fixnum>) () ())))]
```

```
[p-positive?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b+0 ? (obj.ptr m) ff)))
      (alu-if (not (or (n-flag r) (z-flag r))) () ())))))]

[p-negative?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b+0 ? (obj.ptr m) ff)))
      (alu-if (n-flag r) () ())))))]

[p-zero?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b+0 ? (obj.ptr m) ff)))
      (alu-if (z-flag r) () ())))))]

[p-odd?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (c-odd? m) () ())))]

[p-even?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (not (c-odd? m)) () ())))]

[p-eq
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu a-b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (z-flag r) () ())))))]

[p-ne
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu a-b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (not (z-flag r)) () ())))))]

[p-gt
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu b-a (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (xor (n-flag r) (v-flag r)) () ())))))]

[p-ge
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu a-b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (not (xor (n-flag r) (v-flag r))) () ())))))]
```

```
[p-lt
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu a-b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (xor (n-flag r) (v-flag r)) () ()))))]

[p-le
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((r (alu b-a (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (alu-if (not (xor (n-flag r) (v-flag r))) () ()))))]

[p-add
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite (make-imm (memrd mem addr))
                       (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff))))
           (addr (add addr (c24 0) tt)))
    (let ((r (alu a+b (obj.ptr acc) (obj.ptr (memrd mem addr)) ff)))
      (if (v-flag r)
          (⇒ error (addr (add (c24 0) (c24 overflow-error) ff))
             (acc (hcite (obj.tag acc) (alu-out r))))
          (⇒ apply-cont (acc (hcite (obj.tag acc) (alu-out r)))))))))]

[p-sub
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite (make-imm (memrd mem addr))
                       (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff))))
           (addr (add addr (c24 0) tt)))
    (let ((r (alu a-b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (if (v-flag r)
          (⇒ error (addr (add (c24 0) (c24 overflow-error) ff))
             (acc (hcite (obj.tag acc) (alu-out r))))
          (⇒ apply-cont (acc (hcite (obj.tag acc) (alu-out r)))))))))]

[p-add1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b+0 ? (obj.ptr m) tt)))
      (if (v-flag r)
          (⇒ error (addr (add (c24 0) (c24 overflow-error) ff))
             (acc (hcite (make-imm m) (alu-out r))))
          (⇒ apply-cont (acc (hcite (make-imm m) (alu-out r)))))))))]

[p-sub1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b-0 ? (obj.ptr m) ff)))
      (if (v-flag r)
          (⇒ error (addr (add (c24 0) (c24 overflow-error) ff))
             (acc (hcite (make-imm m) (alu-out r))))
          (⇒ apply-cont (acc (hcite (make-imm m) (alu-out r)))))))))]
```

```
[p-asr
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (let ((r (alu b+0 ? (obj.ptr m) ff)))
      (⇒ apply-cont (acc (hcite (make-imm m) (asr (alu-out r))))))))]

[p-boolean?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (or (same-type? m <imm> <true>) (same-type? m <imm> <false>)) () ())))]

[p-not
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <imm> <false>) () ())))]

[p-bitand
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite (make-imm (memrd mem addr))
                       (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff))))
           (addr (add addr (c24 0) tt)))
    (let ((r (alu a&b (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (⇒ apply-cont (acc (hcite (obj.tag acc) (alu-out r)))))))]

[p-bitor
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite (make-imm (memrd mem addr))
                       (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff))))
           (addr (add addr (c24 0) tt)))
    (let ((r (alu aorb (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (⇒ apply-cont (acc (hcite (obj.tag acc) (alu-out r)))))))]

[p-bitxor
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (hcite (make-imm (memrd mem addr))
                       (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff))))
           (addr (add addr (c24 0) tt)))
    (let ((r (alu axorb (obj.ptr acc) (obj.ptr (memrd mem addr)) tt)))
      (⇒ apply-cont (acc (hcite (obj.tag acc) (alu-out r)))))))]

[p-char?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <imm> <char>) () ())))]

[p-read-char
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((pc (add (obj.ptr (memrd mem addr)) (c24 0) ff))
           (addr (add addr (c24 0) tt)))
    ; PC = port number
    (let ((p (pread port pc)))
      (⇒ apply-cont (acc (hcite (obj.tag p) (alu-out (alu b+0 ? (obj.ptr p) ff))))))))]
```

```
[p-write-char
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((pc (add (obj.ptr (memrd mem addr)) (c24 0) ff)) (addr (add addr (c24 0) tt)))
   ; PC = port number, ACC = char
   (assign ((acc (alu-memrd mem addr)))
     (⇒ apply-cont (port (pwrite port pc acc)) (acc (alu-void))))))]

[p-eof-object?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <imm> <eof-object>) () ()))))]

[p-null?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <imm> <null>) () ()))))]

[p-pair?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <fixed> <pair>) () ()))))]

[p-cons
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <fixed> <pair>)))
      (pc (add addr (c24 0) ff)) (cont p-cons-1)))]

[p-cons-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem pc)) (pc (add pc (c24 0) tt)) (val* addr))
   (assign ((mem (memwr mem addr acc)) (addr (add addr (c24 0) tt)))
    (assign ((acc (alu-memrd mem pc)))
     (assign ((mem (memwr mem addr acc)) (addr (add val* (c24 0) ff)))
       (⇒ apply-cont (acc (hcite <fixed> <pair> addr))))))))]

[p-vector?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <vector> <vec>) () ()))))]

[p-make-vector
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <vec>))) (cont p-make-vector-1)
      (addr (add (c24 0) (obj.ptr (memrd mem addr)) ff))))]

[p-make-vector-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (hcite <vector> <vec> addr))))]

[p-string?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
```

```
    (zero-if (same-type? m <vector> <string>) () ()))))]

[p-make-string
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <string>))) (cont p-make-string-1)
      (addr (add (c24 0) (obj.ptr (memrd mem addr)) ff))))]

[p-make-string-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (hcite <vector> <string> addr))))]

[p-symbol?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (same-type? m <vector> <symbol>) () ())))]

[p-string->symbol
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (hcite <vector> <symbol>
                            (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff)))))))]

[p-symbol->string
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (hcite <vector> <string>
                            (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff)))))))]

[p-procedure?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (let ((m (memrd mem addr)))
    (zero-if (or (same-type? m <imm> <primitive>) (same-type? m <fixed> <closure>)
                 (same-type? m <fixed> <callcc>)) () ())))]

[p-eq?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (let ((m (memrd mem addr)))
      (let ((r (alu a-b (obj.ptr acc) (obj.ptr m) tt)))
        (alu-if (and (same-type? acc (obj.tag.type m) (obj.tag.sub m)) (z-flag r)) () ())))))]

[p-apply
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((acc (alu-memrd mem addr)) (addr (add addr (c24 0) tt)))
    (⇒ apply-proc (val* (obj.ptr (memrd mem addr))))))]

[p-abort
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ save (acc (alu-void)) (addr (add (c24 0) (c24 0) tt)) (cont start)))]

[p-call/cc
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <vec>))) (addr (add (c24 0) (c24 0) tt))
      (acc (alu-memrd mem addr)) (cont p-call/cc-1)))]
```

```
[p-call/cc-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <fixed> <callcc>))) (val* addr)
      (pc (add addr (c24 0) tt)) (cont p-call/cc-2)))]

[p-call/cc-2
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((mem (memwr mem addr k)))
    (⇒ apply-proc (mem (memwr mem pc (hcite tag addr))))))]

[p-void
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ apply-cont (acc (alu-void))))]

[p-read-obj
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((addr (add (c24 0) (obj.ptr (memrd mem addr)) ff)) (pc (add addr (c24 0) tt)))
   ; ADDR → object, PC → offset
   (assign ((addr (add addr (obj.ptr (memrd mem pc)) ff)))
     (⇒ apply-cont (acc (alu-memrd mem addr))))))]

[p-write-obj
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((addr (add (c24 0) (obj.ptr (memrd mem addr)) ff)) (pc (add addr (c24 0) tt)))
   ; ADDR → object, PC → offset
   (assign ((addr (add addr (obj.ptr (memrd mem pc)) ff)) (pc (add pc (c24 0) tt)))
    (assign ((acc (alu-memrd mem pc)))
      (⇒ apply-cont (acc (alu-void)) (mem (memwr mem addr acc)))))))]

[p-gc
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ malloc (tag (c8 (make-tag <vector> <vec>)))
      (addr (add (c24 0) (c24 #x7fffff) ff)) (cont p-gc-1)))]

[p-gc-1
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (⇒ load-1 (addr (add (c24 0) (c24 val∗-loc) ff)) (acc (alu-void)) (cont apply-cont)))]

[p-argc
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((addr (add (c24 0) lex-env ff)))
    (⇒ apply-cont (acc (hcite (obj.tag (memrd mem addr))
                              (alu-out (alu b-0 ? (obj.ptr (memrd mem addr)) ff)))))))]

[p-execute
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
  (assign ((code (obj.ptr (memrd mem addr))) (addr (add addr (c24 0) tt)))
    (assign ((lex-env (obj.ptr (memrd mem addr))) (addr (add (c24 0) (c24 0) ff)))
      (⇒ fetch (pc (add addr code tt)) (val* (c24 zero-loc)) (acc (alu-void))))))]

[p-interrupts-enabled?
 (λ (acc val* lex-env k code pc tag addr ie port mem cont)
```

```
      (zero-if ie () ()))]

    [p-interrupts-enabled
     (λ (acc val* lex-env k code pc tag addr ie port mem cont)
      (let ((m (memrd mem addr)))
        (zero-if (true-value? m) ((ie tt)) ((ie ff)))))]

    [p-interrupt-handler
     (λ (acc val* lex-env k code pc tag addr ie port mem cont)
      (⇒ apply-cont (acc (alu-memrd mem (c24 int-loc)))))]

    [p-set-interrupt-handler
     (λ (acc val* lex-env k code pc tag addr ie port mem cont)
      (assign ((acc (alu-memrd mem addr)))
        (⇒ apply-cont (mem (memwr mem (c24 int-loc) acc)) (acc (alu-void)))))]

    [p-error-handler
     (λ (acc val* lex-env k code pc tag addr ie port mem cont)
      (⇒ apply-cont (acc (alu-memrd mem (c24 err-loc)))))]

    [p-set-error-handler
     (λ (acc val* lex-env k code pc tag addr ie port mem cont)
      (assign ((acc (alu-memrd mem addr)))
        (⇒ apply-cont (mem (memwr mem (c24 err-loc) acc)) (acc (alu-void)))))])
  (start ? ? ? ? ? ? ? ? ? ? ? ? ?))))

(extend-syntax (zero-if)
  ((_ pred ((reg1 exp1) ...) ((reg2 exp2) ...))
   (if pred
       (⇒ apply-cont (acc (hcite <imm> <true> (alu-out (alu clr ? ? ff))))
          (reg1 exp1) ...)
       (⇒ apply-cont (acc (hcite <imm> <false> (alu-out (alu clr ? ? ff))))
          (reg2 exp2) ...))))

(extend-syntax (alu-if)
  ((_ pred ((reg1 exp1) ...) ((reg2 exp2) ...))
   (if pred
       (⇒ apply-cont (acc (hcite <imm> <true> (alu-out r))) (reg1 exp1) ...)
       (⇒ apply-cont (acc (hcite <imm> <false> (alu-out r))) (reg2 exp2) ...))))

(extend-syntax (alu-void)
  ((_)
   (hcite <imm> <void> (alu-out (alu clr ? ? ff)))))

(extend-syntax (alu-memrd)
  ((_ mem addr)
   (hcite (obj.tag (memrd mem addr)) (alu-out (alu b+0 ? (obj.ptr (memrd mem addr)) ff)))))
```

C DDD Scripts

C.1 seq.s

```
(load "/u/bose/DDD/V1/ddd.so")
```

```
(load "add.ss")
(ddd-file-overwrite #t)

(define scmac (read-file "machine.ss"))
(let ([SEQ (itrsys->seqsys scmac)])
  (set! SEQSYS_1 SEQ)
  (set! SELECT_1 (seqsys.select SEQSYS_1))
  (set! SYSTEM_1 (seqsys.system SEQSYS_1))
  (set! BASE_1   (seqsys.base   SEQSYS_1))

  (set! SYSTEM_1 (rename-stream 'VAL* 'VALS SYSTEM_1))
```

Rename the start state so that it's first

```
  (set! SYSTEM_1 (subst* 'START 'A-START SYSTEM_1))
  (set! SELECT_1 (subst* 'start 'a-start SELECT_1))

  (write-file SELECT_1 "seqsel.ss")
  (write-file SYSTEM_1 "seq.ss")
  (system->rtt SYSTEM_1 "seq.rtt")

  (exit))

(exit 1)
```

C.2 alu.s

```
(load "/u/bose/DDD/V1/ddd.so")
(load "fm/f.ss")
(load "add.ss")

(define SYSTEM  (read-file "seq.ss"))

(let* ([SYS SYSTEM]
```

Abstract ALU

```
       [SYS (car (factor-general 'ALU 'ABS-ALU '(ALU) SYS))]
       [SYS (remove-stream 'ALU SYS)]
       [SYS (remove-output 'ALU SYS)]
       [SYS (remove-stream 'ALU-INST SYS)]
       [SYS (remove-input 'ALU-INST SYS)]
       [SYS (remove-output 'ALU-INST SYS)]
       [SYS (rename-stream 'ALU-V0 'ALU-F SYS)]
       [SYS (rename-stream 'ALU-V1 'ALU-A SYS)]
       [SYS (rename-stream 'ALU-V2 'ALU-B SYS)]
       [SYS (rename-stream 'ALU-V3 'ALU-CIN SYS)]
```

R is also the ALU

```
       [SYS (if (eqv? 'ALU (constant-streqn? 'R SYS))
                (remove-stream 'R SYS)
                (error 'alu.s "streqn R is not just ALU"))]
       [SYS (remove-stream 'R SYS)]
```

```
    [SYS (remove-input 'R SYS)]
    [SYS (remove-output 'R SYS)]
    [SYS (subst* 'R 'ALU SYS)]
```

Break ACC into ACC.TAG & ACC.PTR

```
    [SYS (car (rename-function 'MAKE-CITE 'CONCAT-ACC 'ACC SYS))]
    [SYS (car (factor-general 'ACC-FRAG 'ABS-ACC-FRAG '(CONCAT-ACC) SYS))]
    [SYS (remove-stream 'ACC-FRAG SYS)]
    [SYS (remove-input 'ACC-FRAG SYS)]
    [SYS (remove-output 'ACC-FRAG SYS)]
    [SYS (remove-stream 'ACC-FRAG-INST SYS)]
    [SYS (remove-input 'ACC-FRAG-INST SYS)]
    [SYS (remove-output 'ACC-FRAG-INST SYS)]
    [SYS (rename-stream 'ACC-FRAG-V0 'ACC.TAG SYS)]
    [SYS (rename-stream 'ACC-FRAG-V1 'ACC.PTR SYS)]
    [SYS (make-reg-stream 'ACC.TAG SYS)]
    [SYS (make-reg-stream 'ACC.PTR SYS)]
    [SYS (if (eqv? 'ACC-FRAG (constant-streqn? 'ACC SYS))
             (remove-stream 'ACC SYS)
             (error 'alu.s "ACC didn't fragment into ACC.TAG & ACC.PTR"))]
    [SYS (remove-input 'ACC SYS)]
    [SYS (remove-output 'ACC SYS)]
    [SYS (subst* 'ACC '(MAKE-CITE ACC.TAG ACC.PTR) SYS)]
```

Abstract ADDERs

```
    [SYS (car (rename-function 'ADD 'ADD-ADDR 'ADDR SYS))]
    [SYS (car (factor-general 'ADDER-ADDR 'ABS-ADD-ADDR '(ADD-ADDR) SYS))]
    [SYS (remove-stream 'ADDER-ADDR SYS)]
    [SYS (remove-output 'ADDER-ADDR SYS)]
    [SYS (remove-stream 'ADDER-ADDR-INST SYS)]
    [SYS (remove-input 'ADDER-ADDR-INST SYS)]
    [SYS (remove-output 'ADDER-ADDR-INST SYS)]
    [SYS (rename-stream 'ADDER-ADDR-V0 'ADDER-ADDR-A SYS)]
    [SYS (rename-stream 'ADDER-ADDR-V1 'ADDER-ADDR-B SYS)]
    [SYS (rename-stream 'ADDER-ADDR-V2 'ADDER-ADDR-CIN SYS)]

    [SYS (car (rename-function 'ADD 'ADD-PC 'PC SYS))]
    [SYS (car (factor-general 'ADDER-PC 'ABS-ADD-PC '(ADD-PC) SYS))]
    [SYS (remove-stream 'ADDER-PC SYS)]
    [SYS (remove-output 'ADDER-PC SYS)]
    [SYS (remove-stream 'ADDER-PC-INST SYS)]
    [SYS (remove-input 'ADDER-PC-INST SYS)]
    [SYS (remove-output 'ADDER-PC-INST SYS)]
    [SYS (rename-stream 'ADDER-PC-V0 'ADDER-PC-A SYS)]
    [SYS (rename-stream 'ADDER-PC-V1 'ADDER-PC-B SYS)]
    [SYS (rename-stream 'ADDER-PC-V2 'ADDER-PC-CIN SYS)])

(set! SYSTEM SYS)

(delete-file "alu.rtt")
(system->rtt SYSTEM "alu.rtt")
```

```
  (delete-file "alu.ss")
  (write-file SYSTEM "alu.ss")
  (exit))

(exit 1)
```

C.3 alloc.s

```
(load "/u/bose/DDD/V1/ddd.so")
(load "add.ss")

(define SYSTEM  (read-file "alu.ss"))
```

Abstract ALLOC

```
(let* ([sys SYSTEM]
       [sys (car (factor-general 'ALLOC 'ABS-ALLOC '(ALLOC) sys))]
       [sys (remove-stream 'ALLOC sys)]
       [sys (remove-output 'ALLOC sys)]
       [sys (remove-stream 'ALLOC-INST sys)]
       [sys (remove-input 'ALLOC-INST sys)]
       [sys (remove-output 'ALLOC-INST sys)]
       [sys (rename-stream 'ALLOC-V0 'ALLOC-ALLOC sys)]
       [sys (rename-stream 'ALLOC-V1 'READ-AVAIL sys)]
       [sys (rename-stream 'ALLOC-V2 'TAG-SIZE sys)]
```

A is short for ALLOC

```
       [sys (if (eqv? 'ALLOC (constant-streqn? 'A sys))
                (remove-stream 'A sys)
                (error 'alloc.s "streqn A is not just ALLOC"))]
       [sys (remove-input 'A sys)]
       [sys (remove-output 'A sys)]
       [sys (subst* 'A 'ALLOC sys)])
  (set! SYSTEM sys)

  (delete-file "alloc.rtt")
  (system->rtt SYSTEM "alloc.rtt")

  (delete-file "alloc.ss")
  (write-file SYSTEM "alloc.ss")
  (exit)
  )

(exit 1)
```

C.4 mem.s

```
(load "/u/bose/DDD/V1/ddd.so")
(load "add.ss")

(define SYSTEM (read-file "alloc.ss"))
```

(**let**$*$ ([SYS $SYSTEM$]

Abstract PORT

[SYS (*car* (*factor-streqn* 'PORT-OUT 'ABS-PORT 'PORT SYS))]
[SYS (*remove-stream* 'PORT-V0 SYS)]
[SYS (*remove-input* 'PORT-V0 SYS)]
[SYS (*remove-stream* 'PORT-PROBE-0-V0 SYS)]
[SYS (*remove-input* 'PORT-PROBE-0-V0 SYS)]
[SYS (*merge-streams* 'PORT-INST 'PORT-PROBE-0-INST 'PORT-INST SYS)]
[SYS (*merge-streams* 'PORT-V1 'PORT-PROBE-0-V1 'PORT-ADDR SYS)]
[SYS (*rename-stream* 'PORT-V2 'PORT-DATA SYS)]
[SYS (*subst*$*$ 'PORT-OUT-0 'PORT-OUT SYS)]
[SYS (*subst*$*$ 'PORT-OUT0 'PORT-OUT SYS)]
[SYS (*remove-stream* 'PORT-OUT SYS)]
[SYS (*remove-input* 'PORT-OUT SYS)]

P is just PORT-OUT

[SYS (**if** (*eqv?* 'PORT-OUT (*constant-streqn?* 'P SYS))
(*remove-stream* 'P SYS)
(*error* 'mem.s "streqn P is not just PORT-OUT")))]
[SYS (*remove-input* 'P SYS)]
[SYS (*remove-output* 'P SYS)]
[SYS (*subst*$*$ 'P 'PORT-OUT SYS)]

Abstract MEM

[SYS (*car* (*factor-streqn* 'MEM-OUT 'ABS-MEM 'MEM SYS))]
[SYS (*remove-stream* 'MEM-V0 SYS)]
[SYS (*remove-input* 'MEM-V0 SYS)]
[SYS (*remove-output* 'MEM-V0 SYS)]
[SYS (*remove-stream* 'MEM-PROBE-0-V0 SYS)]
[SYS (*remove-input* 'MEM-PROBE-0-V0 SYS)]
[SYS (*remove-output* 'MEM-PROBE-0-V0 SYS)]
[SYS (*merge-streams* 'MEM-INST 'MEM-PROBE-0-INST 'MEM-INST SYS)]
[SYS (*merge-streams* 'MEM-V1 'MEM-PROBE-0-V1 'MEM-ADDR SYS)]
[SYS (*rename-stream* 'MEM-V2 'MEM-DATA SYS)]
[SYS (*subst*$*$ 'MEM-OUT-0 'MEM-OUT SYS)]
[SYS (*subst*$*$ 'MEM-OUT0 'MEM-OUT SYS)]
[SYS (*remove-stream* 'MEM-OUT SYS)]

M is just MEM-OUT

[SYS (**if** (*eqv?* 'MEM-OUT (*constant-streqn?* 'M SYS))
(*remove-stream* 'M SYS)
(*error* 'mem.s "streqn M is not just MEM-OUT")))]
[SYS (*remove-input* 'M SYS)]
[SYS (*remove-output* 'M SYS)]
[SYS (*subst*$*$ 'M 'MEM-OUT SYS)]

Combine PORT + MEM + TAG-SIZE buses

[SYS (*merge-streams* 'MEM-ADDR 'PORT-ADDR 'ADDR-OUT SYS)]
[SYS (*merge-streams* 'MEM-DATA 'PORT-DATA 'DATA-OUT SYS)]
[SYS (*subst*$*$ 'MEM-OUT 'DATA-IN SYS)]
[SYS (*subst*$*$ 'PORT-OUT 'DATA-IN SYS)]

```
        [SYS (merge-streams 'TAG-SIZE 'DATA-OUT 'DATA-OUT SYS)])
  (set! SYSTEM SYS)

  (delete-file "mem.rtt")
  (system->rtt SYSTEM "mem.rtt")

  (delete-file "mem.ss")
  (write-file SYSTEM "mem.ss")

  (exit))

(exit 1)
```

C.5 const.s

```
(load "/u/bose/DDD/V1/ddd.so")
```

This routine goes through a given system and factors out all constants. It will only factor out constants used more than once in a given streqn. Furthermore, no streqn will contain more than one constant box when the routine finishes. Constants are represented by a function, whose name is given. Unfortunately, factor-const may not produce the minimal number of constant boxes. This particular optimization problem is equivalent to graph coloring and is thus NP-complete.

```
(define factor-const
  (lambda (name system)
    (let ([streqns (system.streqns system)]
          [name-prefix (string-append (symbol->string name) "-")])
      (let loop
           ([poss (target-streqns name streqns)]
            [db '()]
            [n 0]
            [sys system])
        (if (null? poss)
            ; All done – time to factor 'em out!
            (let factor
                 ([lst db]
                  [sys sys])
              (if (null? lst)
                  sys
                  (let* ([entry (car lst)]
                         [name (car entry)]
                         [values (cdr entry)]
                         [sys (factor-const2 name values sys)])
                    (factor (cdr lst) sys))))

            (let ([streqn (car poss)]
                  [rest (cdr poss)])
              (let ([flst (streqn->function-list name streqn)]
                    [streqn-name (streqn.name streqn)])
                (let merge
                     ([pre db]
                      [post '()])
```

```
                    (if (null? pre)
                        ; Not mergeable – so we create another entry
                        (let ([new-n (add1 n)]
                              [new-name (string->symbol
                                         (string-append name-prefix
                                                        (number->string n)))])
                          (let ([sys (car (rename-function
                                           name
                                           new-name
                                           streqn-name
                                           sys))]
                                [new (cons new-name flst)])
                            (loop rest (cons new post) new-n sys)))

                        ; Test for mergeability
                        (let* ([entry (car pre)]
                               [new-name (car entry)]
                               [flst2 (cdr entry)]
                               [new-flst (merge-function-lists flst flst2)])
                          (if new-flst
                              ; They merged successfully
                              (let ([new-entry (cons new-name new-flst)]
                                    [sys (car (rename-function
                                               name
                                               new-name
                                               streqn-name
                                               sys))])
                                (diagnostic "Merged" streqn-name "with" new-name)
                                (loop rest
                                      (cons new-entry (append post (cdr pre)))
                                      n
                                      sys))
                              ; No merge
                              (merge (cdr pre) (cons entry post)))))))))))))

(define factor-const2
  (letrec ([rm (lambda (name lst)
                 (if (null? lst)
                     '()
                     (let ([head (car lst)]
                           [tail (cdr lst)])
                       (if (eq? head name)
                           head
                           (if (pair? head)
                               (cons (rm name head)
                                     (rm name tail))
                               (cons head
                                     (rm name tail)))))))])

    (lambda (name values system)
      (diagnostic "factoring constant: " name)
      (let ([streams (rm name (system.streams system))])
```

```
        (build-system (system.name system)
                      (system.inputs system)
                      (append streams (list (list name values)))
                      (system.outputs system))))))

(define show-const
  (lambda (name system)
    ; Find the streqns that have more than 1 unique constant
    (let loop ([poss (target-streqns name (system.streqns system))]
               [db '()])
      (if (null? poss)
          ; Now compute the compatibility chart
          (if (null? db)
              ; No constants found!
              (printf "No optimization of ~s needed~n" name)

              (let loop1 ([base db])
                (if (null? base)
                    ; Done!
                    (void)
                    (let* ([entry (car base)]
                           [flst (cdr entry)])
                      (printf "~s: " (car entry))
                      (let loop2 ([lst (cdr base)])
                        (if (null? lst)
                            (newline)
                            (let* ([entry2 (car lst)]
                                   [flst2 (cdr entry2)]
                                   [merged (merge-function-lists flst flst2)])
                              (if (merge-function-lists flst flst2)
                                  (printf " ~s " (car entry2))
                                  (printf "!~s " (car entry2)))
                              (loop2 (cdr lst)))))
                      (loop1 (cdr base))))))

          (let* ([streqn (car poss)]
                 [flst (streqn->function-list name streqn)]
                 [entry (cons (streqn.name streqn) flst)])
            (loop (cdr poss) (cons entry db)))))))

(define SYSTEM (read-file "mem.ss"))
```

Abstract Constants

```
(printf "C32 chart:~n")
(show-const 'C32 SYSTEM)
(printf "C24 chart:~n")
(show-const 'C24 SYSTEM)
(printf "C8 chart:~n")
(show-const 'C8 SYSTEM)

(let* ([SYS SYSTEM]
       [SYS (factor-const 'C32 SYS)]
```

```
        [SYS (factor-const 'C24 SYS)]
        [SYS (factor-const 'C8 SYS)]
        [SYS (make-unique-?&NOP SYS)]
        )
    (set! SYSTEM SYS)

    (delete-file "const.rtt")
    (system->rtt SYSTEM "const.rtt")

    (delete-file "const.ss")
    (write-file SYSTEM "const.ss")
    (exit))

(exit 1)
```

C.6 sel.s

```
(case-sensitive #t)
(load "sm.rep")
(load "pal.ss")
(ddd-file-overwrite #t)

(define SYSTEM
  (subst* 'C24-1-? '(C24 0)
    (subst* 'C24-0-? '(C24 1)
      (subst* 'C8-1-? '(C8 (MAKE-TAG <VECTOR> <VEC>))
        (subst* 'C8-0-? '(C8 (MAKE-TAG <VECTOR> <VEC>))
          (read-file "const.ss"))))))

(define SELECT (read-file "seqsel.ss"))

(load "sel2.ss")
(load "actel.ss")

(define single-row->bit-seqsys
  (lambda (name sys sel rep.map)
    (let* ([seq (optimize-seqsys sel (single-row->system name sys))]
           [sel (seqsys.select seq)]
           [sys (seqsys.system seq)]
           [bits (cdr (project-streqns rep.map (system.streams sys) 32))]
           [sys
            (build-system
              (system.name sys)
              (system.inputs sys)
              (cons (system.status sys) bits)
              (system.outputs sys))])
      (optimize-seqsys sel sys))))

(define generate-sel-map
  (lambda (seq name)
    (let ([streqns (system.streqns (seqsys.system seq))])
```

```
    (car (streqn->map/actel (car streqns) name)))))

(define ACC.TAG.SEQ (single-row->bit-seqsys 'ACC.TAG SYSTEM SELECT sm.map))
(define ACC.TAG.sel (seqsys.select ACC.TAG.SEQ))
(define ACC.TAG.sel.beqn
  (sel+map->booleqns ACC.TAG.sel (generate-sel-map ACC.TAG.SEQ 'ACC.TAG)))

(define ACC.PTR.SEQ (single-row->bit-seqsys 'ACC.PTR SYSTEM SELECT sm.map))
(define ACC.PTR.sel (seqsys.select ACC.PTR.SEQ))
(define ACC.PTR.sel.beqn
  (sel+map->booleqns ACC.PTR.sel (generate-sel-map ACC.PTR.SEQ 'ACC.PTR)))

(define VALS.SEQ (single-row->bit-seqsys 'VALS SYSTEM SELECT sm.map))
(define VALS.sel (seqsys.select VALS.SEQ))
(define VALS.sel.beqn
  (sel+map->booleqns VALS.sel (generate-sel-map VALS.SEQ 'VALS)))

(define LEX-ENV.SEQ (single-row->bit-seqsys 'LEX-ENV SYSTEM SELECT sm.map))
(define LEX-ENV.sel (seqsys.select LEX-ENV.SEQ))
(define LEX-ENV.sel.beqn
  (sel+map->booleqns LEX-ENV.sel (generate-sel-map LEX-ENV.SEQ 'LEX-ENV)))

(define K.SEQ (single-row->bit-seqsys 'K SYSTEM SELECT sm.map))
(define K.sel (seqsys.select K.SEQ))
(define K.sel.beqn
  (sel+map->booleqns K.sel (generate-sel-map K.SEQ 'K)))

(define CODE.SEQ (single-row->bit-seqsys 'CODE SYSTEM SELECT sm.map))
(define CODE.sel (seqsys.select CODE.SEQ))
(define CODE.sel.beqn
  (sel+map->booleqns CODE.sel (generate-sel-map CODE.SEQ 'CODE)))

(define ADDER-PC-A.SEQ (single-row->bit-seqsys 'ADDER-PC-A SYSTEM SELECT sm.map))
(define ADDER-PC-A.sel (seqsys.select ADDER-PC-A.SEQ))
(define ADDER-PC-A.sel.beqn
  (sel+map->booleqns ADDER-PC-A.sel (generate-sel-map ADDER-PC-A.SEQ 'ADDER-PC-A)))

(define ADDER-PC-B.SEQ (single-row->bit-seqsys 'ADDER-PC-B SYSTEM SELECT sm.map))
(define ADDER-PC-B.sel (seqsys.select ADDER-PC-B.SEQ))
(define ADDER-PC-B.sel.beqn
  (sel+map->booleqns ADDER-PC-B.sel (generate-sel-map ADDER-PC-B.SEQ 'ADDER-PC-B)))

(define TAG.SEQ (single-row->bit-seqsys 'TAG SYSTEM SELECT sm.map))
(define TAG.sel (seqsys.select TAG.SEQ))
(define TAG.sel.beqn
  (sel+map->booleqns TAG.sel (generate-sel-map TAG.SEQ 'TAG)))

(define ADDER-ADDR-A.SEQ (single-row->bit-seqsys 'ADDER-ADDR-A SYSTEM SELECT sm.map))
(define ADDER-ADDR-A.sel (seqsys.select ADDER-ADDR-A.SEQ))
(define ADDER-ADDR-A.sel.beqn
  (sel+map->booleqns ADDER-ADDR-A.sel
                     (generate-sel-map ADDER-ADDR-A.SEQ 'ADDER-ADDR-A)))
```

```
(define ADDER-ADDR-B.SEQ (single-row->bit-seqsys 'ADDER-ADDR-B SYSTEM SELECT sm.map))
(define ADDER-ADDR-B.sel (seqsys.select ADDER-ADDR-B.SEQ))
(define ADDER-ADDR-B.sel.beqn
  (sel+map->booleqns ADDER-ADDR-B.sel
                     (generate-sel-map ADDER-ADDR-B.SEQ 'ADDER-ADDR-B)))

(define ADDR-OUT.SEQ (single-row->bit-seqsys 'ADDR-OUT SYSTEM SELECT sm.map))
(define ADDR-OUT.sel (seqsys.select ADDR-OUT.SEQ))
(define ADDR-OUT.sel.beqn
  (sel+map->booleqns ADDR-OUT.sel (generate-sel-map ADDR-OUT.SEQ 'ADDR-OUT)))

(define DATA-OUT.SEQ (single-row->bit-seqsys 'DATA-OUT SYSTEM SELECT sm.map))
(define DATA-OUT.sel (seqsys.select DATA-OUT.SEQ))
(define DATA-OUT.sel.beqn
  (sel+map->booleqns DATA-OUT.sel (generate-sel-map DATA-OUT.SEQ 'DATA-OUT)))

(define C24-1.SEQ (single-row->bit-seqsys 'C24-1 SYSTEM SELECT sm.map))
(define C24-1.sel (seqsys.select C24-1.SEQ))
(define C24-1.sel.beqn
  (sel+map->booleqns C24-1.sel (generate-sel-map C24-1.SEQ 'C24-1)))

(define C24-0.SEQ (single-row->bit-seqsys 'C24-0 SYSTEM SELECT sm.map))
(define C24-0.sel (seqsys.select C24-0.SEQ))
(define C24-0.sel.beqn
  (sel+map->booleqns C24-0.sel (generate-sel-map C24-0.SEQ 'C24-0)))

(define C8-1.SEQ (single-row->bit-seqsys 'C8-1 SYSTEM SELECT sm.map))
(define C8-1.sel (seqsys.select C8-1.SEQ))
(define C8-1.sel.beqn
  (sel+map->booleqns C8-1.sel (generate-sel-map C8-1.SEQ 'C8-1)))

(define C8-0.SEQ (single-row->bit-seqsys 'C8-0 SYSTEM SELECT sm.map))
(define C8-0.sel (seqsys.select C8-0.SEQ))
(define C8-0.sel.beqn
  (sel+map->booleqns C8-0.sel (generate-sel-map C8-0.SEQ 'C8-0)))

(define SEL.min
  (espresso (append
              ACC.TAG.sel.beqn
              ACC.PTR.sel.beqn
              VALS.sel.beqn
              LEX-ENV.sel.beqn
              K.sel.beqn
              CODE.sel.beqn
              ADDER-PC-A.sel.beqn
              ADDER-PC-B.sel.beqn
              TAG.sel.beqn
              ADDER-ADDR-A.sel.beqn
              ADDER-ADDR-B.sel.beqn
              ADDR-OUT.sel.beqn
              DATA-OUT.sel.beqn
              C24-1.sel.beqn
```

```
                C24-0.sel.beqn
                C8-1.sel.beqn
                C8-0.sel.beqn
                )
               'booleqns 'SEL))
(system "rm *.SEL SEL.sch")
(booleqns->eqn SEL.min '() 'static "SEL.eqn")
```

Write the equations for the datapath

```
(define write-bit
  (lambda (seq filename)
    (write-file (system.streqns (seqsys.system seq))
                filename)))

(let ([cd (current-directory)])
  (current-directory "bit")
  (write-bit ACC.TAG.SEQ "acc.tag.bit")
  (write-bit ACC.PTR.SEQ "acc.ptr.bit")
  (write-bit VALS.SEQ "vals.bit")
  (write-bit LEX-ENV.SEQ "lex-env.bit")
  (write-bit K.SEQ "k.bit")
  (write-bit CODE.SEQ "code.bit")
  (write-bit ADDER-PC-A.SEQ "adder-pc-a.bit")
  (write-bit ADDER-PC-B.SEQ "adder-pc-b.bit")
  (write-bit TAG.SEQ "tag.bit")
  (write-bit ADDER-ADDR-A.SEQ "adder-addr-a.bit")
  (write-bit ADDER-ADDR-B.SEQ "adder-addr-b.bit")
  (write-bit ADDR-OUT.SEQ "addr-out.bit")
  (write-bit DATA-OUT.SEQ "data-out.bit")
  (write-bit C24-1.SEQ "c24-1.bit")
  (write-bit C24-0.SEQ "c24-0.bit")
  (write-bit C8-1.SEQ "c8-1.bit")
  (write-bit C8-0.SEQ "c8-0.bit")
  (current-directory cd))
```

C.7 next.s

```
(load "sm.rep")
(load "pal.ss")
(ddd-file-overwrite #t)

(define SYSTEM (read-file "const.ss"))
(define SELECT (read-file "seqsel.ss"))

(define STATE (extract-stream 'STATE SYSTEM))
(define CONT (extract-stream 'CONT SYSTEM))
(define MEM-INST (extract-stream 'MEM-INST SYSTEM))
(define PORT-INST (extract-stream 'PORT-INST SYSTEM))
(define ALU-F (extract-stream 'ALU-F SYSTEM))
(define ALU-CIN (extract-stream 'ALU-CIN SYSTEM))
(define ADDER-ADDR-CIN (extract-stream 'ADDER-ADDR-CIN SYSTEM))
```

```
(define ADDER-PC-CIN (extract-stream 'ADDER-PC-CIN SYSTEM))
(define ALLOC-ALLOC (extract-stream 'ALLOC-ALLOC SYSTEM))
(define READ-AVAIL (extract-stream 'READ-AVAIL SYSTEM))
(define IE (extract-stream 'IE SYSTEM))

(define NEXT-STATE (optimize-sel (partial-eval STATE SELECT)))
(define NEXT-CONT (optimize-sel (partial-eval CONT SELECT)))
(define NEXT-MEM-INST (optimize-sel (partial-eval MEM-INST SELECT)))
(define NEXT-PORT-INST (optimize-sel (partial-eval PORT-INST SELECT)))
(define NEXT-ALU-F (optimize-sel (partial-eval ALU-F SELECT)))
(define NEXT-ALU-CIN (optimize-sel (partial-eval ALU-CIN SELECT)))
(define NEXT-ADDER-ADDR-CIN (optimize-sel (partial-eval ADDER-ADDR-CIN SELECT)))
(define NEXT-ADDER-PC-CIN (optimize-sel (partial-eval ADDER-PC-CIN SELECT)))
(define NEXT-IE (optimize-sel (partial-eval IE SELECT)))
(define NEXT-ALLOC-ALLOC (optimize-sel (partial-eval ALLOC-ALLOC SELECT)))
(define NEXT-READ-AVAIL (optimize-sel (partial-eval READ-AVAIL SELECT)))

(define NEXT-STATE.BIN (project-sel state.map NEXT-STATE))
(define NEXT-CONT.BIN (project-sel state.map NEXT-CONT))
(define NEXT-MEM-INST.BIN (project-sel mem.map NEXT-MEM-INST))
(define NEXT-PORT-INST.BIN (project-sel port.map NEXT-PORT-INST))
(define NEXT-ALU-F.BIN (project-sel alu-f.map NEXT-ALU-F))
(define NEXT-ALU-CIN.BIN (project-sel 1bit.map NEXT-ALU-CIN))
(define NEXT-ADDER-ADDR-CIN.BIN (project-sel 1bit.map NEXT-ADDER-ADDR-CIN))
(define NEXT-ADDER-PC-CIN.BIN (project-sel 1bit.map NEXT-ADDER-PC-CIN))
(define NEXT-IE.BIN (project-sel 1bit.map NEXT-IE))
(define NEXT-ALLOC-ALLOC.BIN (project-sel 1bit.map NEXT-ALLOC-ALLOC))
(define NEXT-READ-AVAIL.BIN (project-sel 1bit.map NEXT-READ-AVAIL))

(define bin->eqn
  (lambda (bin)
    (symbolic->bit (map sel->booleqns bin) case.map)))

(define NEXT-STATE.EQN (bin->eqn NEXT-STATE.BIN))
(define NEXT-CONT.EQN (bin->eqn NEXT-CONT.BIN))
(define NEXT-MEM-INST.EQN (bin->eqn NEXT-MEM-INST.BIN))
(define NEXT-PORT-INST.EQN (bin->eqn NEXT-PORT-INST.BIN))
(define NEXT-ALU-F.EQN (bin->eqn NEXT-ALU-F.BIN))
(define NEXT-ALU-CIN.EQN (bin->eqn NEXT-ALU-CIN.BIN))
(define NEXT-ADDER-ADDR-CIN.EQN (bin->eqn NEXT-ADDER-ADDR-CIN.BIN))
(define NEXT-ADDER-PC-CIN.EQN (bin->eqn NEXT-ADDER-PC-CIN.BIN))
(define NEXT-IE.EQN (bin->eqn NEXT-IE.BIN))
(define NEXT-ALLOC-ALLOC.EQN (bin->eqn NEXT-ALLOC-ALLOC.BIN))
(define NEXT-READ-AVAIL.EQN (bin->eqn NEXT-READ-AVAIL.BIN))
```

espresso is used for its side-effect here!! It generates GENERATORS.min, which the DDD Scheme monster reads and converts to GENERATORS.sch

```
(define GENERATORS.min
  (espresso (append NEXT-STATE.EQN NEXT-CONT.EQN NEXT-MEM-INST.EQN
                    NEXT-PORT-INST.EQN NEXT-ALU-F.EQN NEXT-ALU-CIN.EQN
                    NEXT-ADDER-ADDR-CIN.EQN NEXT-ADDER-PC-CIN.EQN
                    NEXT-IE.EQN
```

```
                    NEXT-ALLOC-ALLOC.EQN NEXT-READ-AVAIL.EQN)
            'booleqns 'GENERATORS))

    (write-file (apply append (map sch-¿pal (read-file "GENERATORS.sch"))) "GENERATORS.pal")

(system "rm *.GENERATORS GENERATORS.sch")
(booleqns->eqn GENERATORS.min
               (append
                (eval-slice 'STATE sm.map)
                (eval-slice 'CONT sm.map)
                (eval-slice 'IE sm.map))
               'static
               "GENERATORS.eqn")

(booleqns->eqn trivial-preds.bool '() 'static "trivial-preds.eqn")

(define PREDS.min
  (espresso preds.bool 'booleqns 'PREDS))
(system "rm *.PREDS PREDS.sch PREDS.min")
(booleqns->eqn PREDS.min '() 'static "preds.eqn")
```

C.8 actel.s

```
(if (not (case-sensitive))
    (load "ddd/ddd.so"))
(ddd-file-overwrite #t)
(load "add.ss")
(load "optmux.ss")
(load "actel.ss")

(define actel-it
  (lambda (filename name)
    (let* ([streqns (read-file filename)]
           [streqn->actel (cdr (streqn->map/actel (car streqns) name))])
      (diagnostic "Generating Actel description for:" name)
      (optimize-actel (apply append (map streqn->actel streqns))))))

(define actel-it-noopt
  (lambda (filename name)
    (let* ([streqns (read-file filename)]
           [streqn->actel (cdr (streqn->map/actel (car streqns) name))])
      (diagnostic "Generating Actel description for:" name)
      (apply append (map streqn->actel streqns)))))

(define make-actel
    (lambda ()
      (let ([cd (current-directory)])
        (current-directory "bit")
        (set! actel
              (optimize-actel
               (append
                (actel-it-noopt "c24-1.bit" 'C24-1)
```

```
            (actel-it-noopt "c24-0.bit" 'C24-0)
            (actel-it-noopt "c8-1.bit" 'C8-1)
            (actel-it-noopt "c8-0.bit" 'C8-0)
            (actel-it "acc.tag.bit" 'ACC.TAG)
            (actel-it "acc.ptr.bit" 'ACC.PTR)
            (actel-it "vals.bit" 'VALS)
            (actel-it "lex-env.bit" 'LEX-ENV)
            (actel-it "k.bit" 'K)
            (actel-it "code.bit" 'CODE)
            (actel-it "adder-pc-a.bit" 'ADDER-PC-A)
            (actel-it "adder-pc-b.bit" 'ADDER-PC-B)
            (actel-it "tag.bit" 'TAG)
            (actel-it "adder-addr-a.bit" 'ADDER-ADDR-A)
            (actel-it "adder-addr-b.bit" 'ADDER-ADDR-B)
            (actel-it "addr-out.bit" 'ADDR-OUT)
            (actel-it "data-out.bit" 'DATA-OUT)
            )))
      (current-directory cd))
    (if (not (open-output "mods.actel"))
        (error 'make-actel "couldn't open mods.actel"))
    (let ([p (ddd-out-port)])
      (fprintf p "(inputs)~n(outputs)~n")
      (for-each
       (lambda (mod)
         (display mod p)
         (newline p))
       actel)
      (close-output))
    ))
```

D Actel Multiplexor Optimizer

```
(define match?
  (lambda (lst1 lst2)
    (and
     (eq? (length lst1) (length lst2))
     (andmap
      (lambda (a b)
        (or (eq? a b)
            (eq? a '?)
            (eq? b '?)))
      lst1 lst2))))

(define module.name (lambda (l) (car (reverse l))))

(define nop (lambda (des changed) des))

(define replace-with
  (lambda (old-name new-name)
    (letrec ([help (lambda (des)
                     (if (null? des)
```

```
                        '()
                        (let ([head (car des)]
                              [tail (cdr des)])
                          (if (eq? old-name (module.name head))
                              (help tail)
                              (cons head (help tail))))))])
          (lambda (des changed)
            (diagnostic "folding" old-name "to" new-name)
            (changed)
            (subst* old-name new-name (help des))))))

(define optimize-nand
  (let ([header (lambda (lst) (reverse (cdr (reverse lst))))])
    (lambda (nand lst)
      (let ([hdr (header nand)]
            [name (module.name nand)])
        (let loop ([lst lst])
          (if (null? lst)
              nop
              (let ([head (car lst)])
                (if (equal? hdr (header head))
                    (replace-with (module.name head) name)
                    (loop (cdr lst)))))))))))

(define optimize-mx4
  (let ([low (lambda (s) (string->symbol
                          (string-append (symbol->string s) ".L")))]
        [non-? (lambda (l)
                 (cond
                  [(not (eq? (car l) '?)) (car l)]
                  [(not (eq? (cadr l) '?)) (cadr l)]
                  [(not (eq? (caddr l) '?)) (caddr l)]
                  [else (cadddr l)]))])
    (lambda (mx4 lst)
      (let ([f (subrange 1 4 mx4)]
            [s1 (list-ref mx4 5)]
            [s0 (list-ref mx4 6)]
            [name (list-ref mx4 7)]
            [hdr (subrange 0 6 mx4)])
        (cond
         [(match? f '(0 0 0 0)) (replace-with name '0)]
         [(match? f '(1 1 1 1)) (replace-with name '1)]
         [(match? f '(0 0 1 1)) (replace-with name s1)]
         [(match? f '(0 1 0 1)) (replace-with name s0)]
         [(match? f (make-list 4 (non-? f))) (replace-with name (non-? f))]
         [else
          (let loop ([lst lst])
            (if (null? lst)
                nop
                (let ([head (car lst)])
                  (if (and (eq? (car hdr) (car head))
                           (equal? hdr (subrange 0 6 head)))
```

```
                    (replace-with (module.name head) name)
                    (loop (cdr lst))))))])))))

(define optimize-actel
  (lambda (des)
    (let* ([changed? #f]
           [changed (lambda () (set! changed? #t))])
      (let loop ([lst des]
                 [des des])
        (if changed?
            (optimize-actel des)
            (if (null? lst)
                des
                (let* ([head (car lst)]
                       [tail (cdr lst)]
                       [action (case (car head)
                                 [mx4ddd (optimize-mx4 head tail)]
                                 [(NAND2 NAND2A NAND3 NAND3A)
                                  (optimize-nand head tail)]
                                 [else nop])])
                  (loop tail (action des changed)))))))))
```

E Actel Fanout Utility

```
(define fanout
  (lambda (mods)
    (let* ([counts (get-counts mods)]
           [bufs (get-new counts)]
           [lst (get-transformer counts)]
           [converter (build-analyzer
                       (lambda (l)
                         (map
                          (lambda (i)
                            ((cdr (assq i lst))))
                          l)))])
      (append (map converter mods) bufs))))

(define get-counts
  (lambda (mods)
    (let* ([lst '()]
           [analyzer (build-analyzer
                      (lambda (l)
                        (for-each
                         (lambda (i)
                           (let ([r (assq i lst)])
                             (if r
                                 (set-cdr! r (add1 (cdr r)))
                                 (set! lst (append! lst (list (cons i 1))))))))
                         l)
                        l))])
      (for-each analyzer mods)
```

```
        lst)))

(define build-analyzer
  (lambda (f)
    (lambda (mod)
      (case (car mod)
        [(outddd triddd mx4ddd dfm6addd cm8ddd cm8addd dfddd)
         `(,(car mod) ,@(f (rdc (cdr mod))) ,(rac mod))]
        [(ioddd)
         (let ([l (f (list (cadr mod) (cadddr mod)))])
           `(ioddd ,(car l) ,(caddr mod) ,(cadr l) ,(rac mod)))]
        [(inddd clkddd) mod]
        [else
         (error 'build-analyzer "unrecognized module ~s" mod)]))))

(define get-transformer
  (lambda (counts)
    (map
     (lambda (r)
       (let ([name (car r)]
             [count (cdr r)])
         (cons name
               (let ([t (build-t name count)]
                     [cnt 0])
                 (lambda ()
                   (set! cnt (add1 cnt))
                   (t cnt))))))
     counts)))

(define make-buf
  (lambda (name . endings)
    (printf "Adding ~s buffers for ~s~n" (length endings) name)
    (map
     (lambda (ending)
       `(mx4ddd ,name GND GND GND GND GND ,(make-ide name ending)))
     endings)))

(define build-t
  (lambda (name count)
    (cond
     [(memq name '(Vdd GND clk)) (lambda (i) name)]
     [(< count 11) (lambda (i) name)]
     [(< count 20)
      (lambda (i)
        (cond
         [(< i 10) name]
         [else (make-ide name "a")]))]

     [(< count 29)
      (lambda (i)
        (cond
         [(< i 9) name]
```

```
    [(< i 19) (make-ide name "a")]
    [else (make-ide name "b")]))]

[(< count 38)
 (lambda (i)
   (cond
    [(< i 8) name]
    [(< i 18) (make-ide name "a")]
    [(< i 28) (make-ide name "b")]
    [else (make-ide name "c")]))]

[(< count 47)
 (lambda (i)
   (cond
    [(< i 7) name]
    [(< i 17) (make-ide name "a")]
    [(< i 27) (make-ide name "b")]
    [(< i 37) (make-ide name "c")]
    [else (make-ide name "d")]))]

[(< count 56)
 (lambda (i)
   (cond
    [(< i 6) name]
    [(< i 16) (make-ide name "a")]
    [(< i 26) (make-ide name "b")]
    [(< i 36) (make-ide name "c")]
    [(< i 46) (make-ide name "d")]
    [else (make-ide name "e")]))]

[(< count 65)
 (lambda (i)
   (cond
    [(< i 5) name]
    [(< i 15) (make-ide name "a")]
    [(< i 25) (make-ide name "b")]
    [(< i 35) (make-ide name "c")]
    [(< i 45) (make-ide name "d")]
    [(< i 55) (make-ide name "e")]
    [else (make-ide name "f")]))]

[(< count 74)
 (lambda (i)
   (cond
    [(< i 4) name]
    [(< i 14) (make-ide name "a")]
    [(< i 24) (make-ide name "b")]
    [(< i 34) (make-ide name "c")]
    [(< i 44) (make-ide name "d")]
    [(< i 54) (make-ide name "e")]
    [(< i 64) (make-ide name "f")]
    [else (make-ide name "g")]))]
```

```
        [(< count 83)
         (lambda (i)
           (cond
            [(< i 3) name]
            [(< i 13) (make-ide name "a")]
            [(< i 23) (make-ide name "b")]
            [(< i 33) (make-ide name "c")]
            [(< i 43) (make-ide name "d")]
            [(< i 53) (make-ide name "e")]
            [(< i 63) (make-ide name "f")]
            [(< i 73) (make-ide name "g")]
            [else (make-ide name "h")]))]

        [(< count 92)
         (lambda (i)
           (cond
            [(< i 2) name]
            [(< i 12) (make-ide name "a")]
            [(< i 22) (make-ide name "b")]
            [(< i 32) (make-ide name "c")]
            [(< i 42) (make-ide name "d")]
            [(< i 52) (make-ide name "e")]
            [(< i 62) (make-ide name "f")]
            [(< i 72) (make-ide name "g")]
            [(< i 82) (make-ide name "h")]
            [else (make-ide name "i")]))]

        [else
         (if (>= count 101)
             (printf "~s has ~s fanouts!~n" name count))
         (lambda (i)
           (cond
            [(< i 11) (make-ide name "a")]
            [(< i 21) (make-ide name "b")]
            [(< i 31) (make-ide name "c")]
            [(< i 41) (make-ide name "d")]
            [(< i 51) (make-ide name "e")]
            [(< i 61) (make-ide name "f")]
            [(< i 71) (make-ide name "g")]
            [(< i 81) (make-ide name "h")]
            [(< i 91) (make-ide name "i")]
            [else (make-ide name "j")]))]))))

(define get-new
  (lambda (counts)
    (apply append
           (map
            (lambda (r)
              (let ([name (car r)]
                    [count (cdr r)])
                (cond
                 [(memq name '(Vdd GND clk)) '()]
```

```
                [(< count 11) '()]
                [(< count 20) (make-buf name 'a)]
                [(< count 29) (make-buf name 'a 'b)]
                [(< count 38) (make-buf name 'a 'b 'c)]
                [(< count 47) (make-buf name 'a 'b 'c 'd)]
                [(< count 56) (make-buf name 'a 'b 'c 'd 'e)]
                [(< count 65) (make-buf name 'a 'b 'c 'd 'e 'f)]
                [(< count 74) (make-buf name 'a 'b 'c 'd 'e 'f 'g)]
                [(< count 83) (make-buf name 'a 'b 'c 'd 'e 'f 'g 'h)]
                [(< count 92) (make-buf name 'a 'b 'c 'd 'e 'f 'g 'h 'i)]
                [else (make-buf name 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j)])))
          counts))))

(define rf
  (lambda (filename)
    (let ([p (open-input-file filename)])
      (let ([l (let loop ([e (read p)])
                 (if (eof-object? e)
                     '()
                     (cons e (loop (read p)))))])
        (close-input-port p)
        l))))

(define wf
  (lambda (lst filename)
    (delete-file filename)
    (call-with-output-file filename
      (lambda (p)
        (for-each
         (lambda (e)
           (display e p)
           (newline p))
         lst)))))
```

F Simulator Generator

```
(define sm
  (lambda ()
    (act2c "sm.c" "cat.act" "../comp.act" "../main.act"
           "../adder-addr.act" "../adder-pc.act" "../alu.act")
    (printf "Wrote sm.c~n")))

(define act2c
  (lambda (filename . names)
    (let* ([p (open-output filename)]
           [mods (apply append (map
                                (lambda (name) (convert (read-file* name)))
                                names))]
           [io (get-i/o mods)]
           [i (car io)]
           [o (cadr io)]
```

```
            [d (caddr io)]
            [q (cadddr io)]
            [inputs (union i d)]
            [outputs (union o q)]
            [vars (union inputs outputs)]
            [ext-inputs (difference inputs outputs)]
            [roots (difference o i)]
            )
          (set! *i* i)
          (set! *o* o)
          (set! *d* d)
          (set! *q* q)
          (set! *inputs* inputs)
          (set! *outputs* outputs)
          (set! *vars* vars)
          (set! *ext-inputs* ext-inputs)
          (set! *roots* roots)
          (printf "Writing the C file...~n")
          (fprintf p "#include \"sim.h\"~n~n")
          (fprintf p "void main()~n{~n")
          (for-each
           (lambda (var)
             (if (and (memq var d) (not (memq var ext-inputs)))
                 (fprintf p "  static unsigned char ~s = 0;~n" var)
                 (fprintf p "  static unsigned char ~s;~n" var)))
           vars)
          (fprintf p "~n  for (;;)~n  {~n")
          (for-each (lambda (i) (fprintf p "    input(~s);~n" i)) ext-inputs)
          (fprintf p "    newline();~n~n")
          (for-each (lambda (d q) (fprintf p "    ~s = ~s;~n" q d)) d q)
          (newline p)

          (print-dfs p mods roots o)

          (fprintf p "  }~n~n  exit(0);~n}~n")
          (close-output-port p))))

(define print-dfs
  (lambda (p mods roots o)
    (let loop ([l mods])
      (if (not (null? l))
          (let ([mod (car l)])
            (if (and (pair? mod) (memq (rac mod) roots))
                (begin
                  (set-car! l #f)
                  (dfs p mod mods o (list (rac mod)))
                  (newline p)))
            (loop (cdr l)))))
    ; Collect everything that was missed (this would be an asynch cycle)
    (let ([remnant
           (let loop ([l mods])
             (if (null? l)
```

```
                '()
                (if (and (car l) (not (eqv? (caar l) 'dfddd)))
                    (cons (car l) (loop (cdr l)))
                    (loop (cdr l)))))])
          (if (not (null? remnant))
              (error 'dfs "asynchronous cycle(s) found ~s"
                     (map rac remnant))))))

(define dfs
  (lambda (p mod mods o seen)
    (let ([inputs (rdc (cdr mod))]
          [output (rac mod)])
      (for-each
       (lambda (i)
         (cond
          [(memq i seen)
           (error 'dfs "asynchronous loop found: ~s"
                  (memq i (reverse seen)))]
          [(memq i o)
           (let loop ([l mods])
             (if (not (null? l))
                 (let ([m (car l)])
                   (if (and (pair? m) (eqv? (rac m) i))
                       (begin
                         (set-car! l #f)
                         (dfs p m mods o (cons i seen)))
                       (loop (cdr l))))))]))
       (apply union (map list inputs)))
      (fprintf p "      ~s = ~s(~s" output (car mod) (car inputs))
      (for-each (lambda (i) (fprintf p ", ~s" i)) (cdr inputs))
      (fprintf p ");~n"))))

(define get-i/o
  (letrec ([insert (lambda (val set)
                     (if (or (integer? val) (memq val set))
                         set
                         (cons val set)))]
           [insert* (lambda (vals set)
                      (if (null? vals)
                          set
                          (insert* (cdr vals) (insert (car vals) set))))]
           [insert-nodup (lambda (val set)
                           (if (memq val set)
                               (error 'get-i/o "duplicate name ~s" val)
                               (cons val set)))]
           [help (lambda (mods i o d q)
                   (if (null? mods)
                       (list i o d q)
                       (let* ([mod (car mods)]
                              [rest (cdr mods)]
                              [name (car mod)])
                         (if (eqv? name 'dfddd)
```

```
                        (help rest i o
                              (insert-nodup (cadr mod) d)
                              (insert-nodup (caddr mod) q))
                        (help rest
                              (insert* (rdc (cdr mod)) i)
                              (insert-nodup (rac mod) o)
                              d q))))))])
      (lambda (mods)
        (printf "Determining the inputs & outputs...~n")
        (help mods '() '() '() '())))))

(define convert
  (letrec ([name-cvt! (lambda (s)
                        (let loop ([i 0] [len (string-length s)])
                          (if (>= i len)
                              s
                              (let ([c (string-ref s i)])
                                (if (memq c '(#\. #\-))
                                    (string-set! s i #\_))
                                (loop (add1 i) len)))))]
           [fix-names (lambda (l)
                        (map
                         (lambda (a)
                           (if (symbol? a)
                               (string->symbol (name-cvt! (symbol->string a)))
                               a))
                         l))]
           [help (lambda (mods)
                   (if (null? mods)
                       '()
                       (let* ([mod (fix-names (car mods))]
                              [rest (cdr mods)]
                              [name (car mod)])
                         (case name
                           [dfddd
                            (cons (cons 'dfddd (cdddr mod))
                                  (help rest))]
                           [dfm6addd
                            (let* ([name (rac mod)]
                                   [new-name (make-ide "new_" name)])
                              (cons (cons 'mx4ddd (snoc (rdc (cdddr mod))
                                                       new-name))
                                    (cons (list 'dfddd new-name name)
                                          (help rest))))]
                           [else
                            (cons mod (help rest))])))))])
    (lambda (mods)
      (printf "Massaging the Actel description...~n")
      (help (substlst* '(GND gnd Vdd vdd) '(0 0 1 1) mods)))))
```

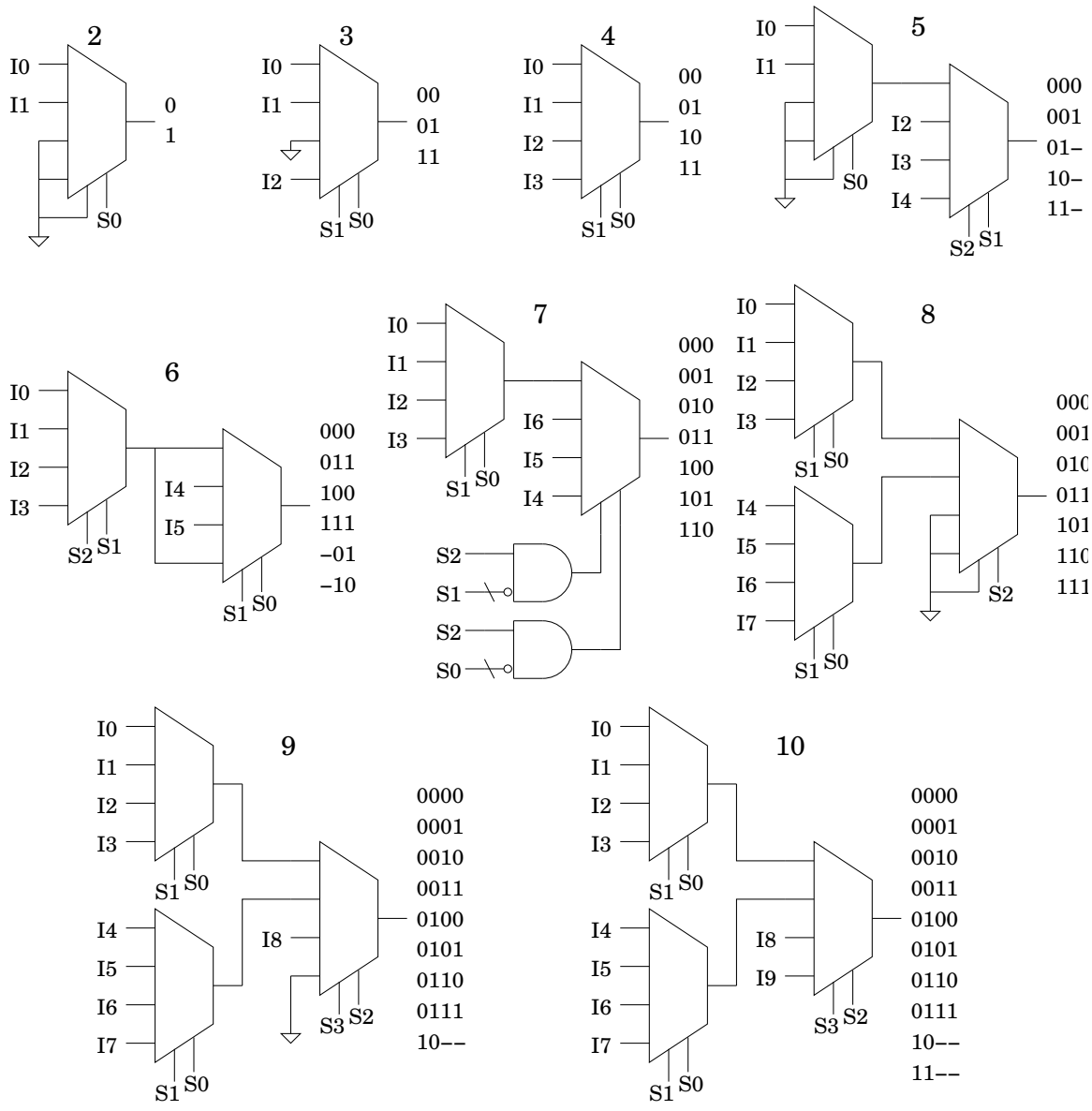

Figure 15: Actel MUX layout