MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 514 March 1979

Design of LISP-Based Processors
or, SCHEME: A Dielectric LISP
or, Finite Memories Considered Harmful
or, LAMBDA: The Ultimate Opcode

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Abstract:

We present a design for a class of computers whose "instruction sets" are based on LISP. LISP, like traditional stored-program machine languages and unlike most high-level languages, conceptually stores programs and data in the same way and explicitly allows programs to be manipulated as data. LISP is therefore a suitable language around which to design a stored-program computer architecture. LISP differs from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of integers or bit fields of fixed size. The record structures can be organized into trees or graphs. An instruction set can be designed for programs expressed as such trees. A processor can interpret these trees in a recursive fashion, and provide automatic storage management for the record structures.

We describe here the basic ideas behind the architecture, and for concreteness give a specific instruction set (on which variations are certainly possible). We also discuss the similarities and differences between these ideas and those of traditional architectures.

A prototype VLSI microprocessor has been designed and fabricated for testing. It is a small-scale version of the ideas presented here, containing a sufficiently complete instruction interpreter to execute small programs, and a rudimentary storage allocator. We intend to design and fabricate a full-scale VLSI version of this architecture in 1979.

Keywords: microprocessors, LISP, SCHEME, large scale integration, integrated circuits, VLSI, list structure, garbage collection, storage management

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This work was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

* Fannie and John Hertz Fellow

** Esther and Harold E. Edgerton Associate Professor of Electrical Engineering

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1979

Introduction

An idea which has increasingly gained attention is that computer architectures should reflect specific language structures to be supported. This is an old idea; one can see features in the machines of the 1960's intended to support COBOL, FORTRAN, ALGOL, and PL/I. More recently research has been conducted into architectures to support string or array processing as in SNOBOL or APL.

An older and by now well-accepted idea is that of the stored-program computer. In such a computer the program and the data reside in the same memory; that is, the program is itself data which can be manipulated as any other data by the processor. It is this idea which allows the implementation of such powerful and incestuous software as program editors, compilers, interpreters, linking loaders, debugging systems, etc.

One of the great failings of most high-level lenguages is that they have abandoned this idea. It is extremely difficult, for example, for a PL/I (PASCAL, FORTRAN, COBOL ...) program to manipulate PL/I (PASCAL, FORTRAN, COBOL ...) programs.

On the other hand, many of these high-level languages have introduced other powerful ideas not present in standard machine languages. Among these are (1) recursively defined, nested data structures; and (2) the use of functional composition to allow programs to contain expressions as well as (or instead of) statements. The LISP language in fact has both of these features. It is unusual among high-level languages in that it also explicitly supports the stored-program idea: LISP programs are represented in a standardized way as recursively defined, nested LISP data structures. By contrast with some APL implementations, for example, which allow programs to be represented as arrays of characters, LISP also reflects the structure of program expressions in the structure of the data which represents the program. (An array of APL characters must be parsed to determine the logical structure of the APL expressions represented by the array. Similar remarks apply to SNOBOL statements represented as SNOBOL strings.)

It is for this reason that LISP is often referred to as a "high-level machine language". As with standard stored-program machine languages, programs and data are made of the same stuff. In a standard machine, however, the "stuff" is a homogeneous, linear (ordered) vector of fixed-size bit fields; a program is represented as an ordered sequence of bit fields (instructions) within the overall vector. In LISP, the "stuff" is a heterogeneous, unordered set of records linked to form lists, trees, and graphs; a program is represented as a tree (a "parse tree" or "expression tree") of linked records (a subset of the overall set of records). Standard machines usually exploit the linear nature of the "stuff" through such mechanisms as indexing by additive offset and linearly advancing program counters. A computer based on LISP can similarly exploit tree structures. The counterpart of indexing is component selection; the counterpart of linear instruction execution is evaluation of expressions by recursive tree-walk.

Just as the "linear vector" stored-program-computer model leads to a variety of specific architectures, so with the "linked record" model. For concreteness we present here one specific architecture based on the linked record model which has actually been constructed.

List Structure and Programs

One of the central ideas of the LISP language is that storage management should be completely invisible to the programmer, so that he need not concern himself with the issues involved. LISP is an object-oriented language, rather than a value-oriented language. The LISP programmer does not think of variables as the objects of interest, bins in which values can be held. Instead, each data item is itself an object, which can be examined and modified, and which has an identity independent of the variable(s) used to name it.

In this section we discuss LISP data structures at the conceptual level; the precise form of LISP data objects is not of concern here. Later we will discuss specific representations within the machine. LISP data is collectively referred to as "S-expressions" ("S" for "symbolic"). For our purposes we will need only the special cases of S-expressions called atoms and lists. An atom is an "indivisible" data object, which we denote by writing a string of letters and digits; if only digits are used, then the atom is considered to be a number. Many special characters such as "-", "+", "@", and "*", are considered to be letters; we will see below that it is not necessary to specially reserve them for use as operator symbols. A list is a (possibly empty) sequence of LISP data objects, notated by (recursively) notating the objects in order, between a set of parentheses and separated by blank space. A list of the atoms "FOO", "43", and "BAR" would be written "(FOO 43 BAR)". Notice that the definition of a list is recursive. For example,

```
(DEFINE SECOND (LAMBDA (X) (CAR (CDR X))))
```

is a list of three things: the atomic symbol DEFINE, the atomic symbol SECOND, and another list of three things LAMBDA, (X), and (CAR (CDR X)).

A convenient way use lists to represent algebraic expressions is to use "Cambridge Polish" notation, essentially a parenthesized version of prefix Polish notation. Numeric constants are encoded as numeric atoms; variables are encoded as non-numeric atoms (which henceforth we will call symbols); and procedure invocations (function calls) are encoded as lists, where the first element of the list represents the procedure and the rest represent the arguments. For example, the algebraic expression "a*b+c*d" can be represented as "(+ (* a b) (* c d))". Notice that LISP does not need the usual precedence rules concerning whether multiplication or addition is performed first; the parentheses (or rather, the structure of the lists) explicitly define the order. Also, all procedure invocations have a uniform syntax, no matter how many arguments are involved. Infix, superscript, and subscript notations are not used; thus the expression "$J_p(x^2+1)$" would be written "(J p (+ (↑ x 2) 1))".

To encode a conditional expression "if p then x else y" we write:

```
(IF p x y)
```

Expressions are made into procedures (functions) by the use of Church's lambda-notation. For example,

```
(LAMBDA (X Y) (+ (* 3 Y) X))
```

evaluates to a function of two arguments X and Y which computes 3*Y+X. The list of variables names after the LAMBDA indicates how the variables names in the expression are to be matched positionally to supplied arguments when the function is applied.

We can also encode recursive LISP programs as list data. For example, to compute N factorial (N!):

```
(DEFINE FACTORIAL
        (LAMBDA (N)
                (IF (= N 0) 1
                    (* N (FACTORIAL (- N 1))))))
```

Suppose that we now want to write a LISP program which will take such a data structure and perform some useful operation on it, such as determining the value of an algebraic expression represented as a list structure. We need some procedures for categorizing, decomposing, and constructing LISP data.

The predicate ATOM, when applied to a LISP datum, produces true when given an atom and false otherwise. The empty list ("()") is considered to be an atom. The predicate NULL is true of only the empty list; its argument need not be a list, but may be any LISP datum. The predicate NUMBERP is true of numbers and false of symbols and lists. The predicate EQ, when applied to two symbols, is true if the two atomic symbols are identical. It is false when applied to two distinct symbols, or to a symbol and any other datum.

The decomposition operators for lists are traditionally called CAR and CDR for historical reasons. CAR extracts the first element of a list, while CDR produces a list containing all elements but the first. Because compositions of CAR and CDR are commonly used in LISP, an abbreviation is provided: all the C's and R's in the middle can be squeezed out. For example, "(CDR (CDR (CAR (CDR X))))" can be written as "(CDDADR X)".

The construction operator CONS, given any datum and a list, produces a new list whose car is the datum and whose cdr is the given list; that is, CONS adds a new element to the front of a list. The operator LIST can take any number of arguments (a special feature), and produces a list of its arguments.

Notice that CONS (and LIST) conceptually create new data structures. As far as the LISP programmer is concerned, new data objects are available in endless supply. They can be conveniently called forth to serve some immediate purpose and discarded when they are no longer of use. While creation is

explicit, discarding is not; a data object simply disappears into limbo when the program throws away all references (direct or indirect) to that object.

The immense freedom this gives the programmer may be seen by an example taken from current experience. A sort of error message familar to most programmers is "too many nested DO loops" or "more than 200 declared arrays" or "symbol table overflow". Such messages typically arise within compilers or assemblers which were written in languages requiring data tables to be pre-allocated to some fixed length. The author of a compiler, for example, might well guess, "No one will ever use more than, say, ten nested DO loops; I'll double that for good measure, and make the nested-DO-loop-table 20 long." Inevitably, someone eventually finds some reason to write 21 nested DO loops, and finds that the compiler overflows its fixed table and issues an error message (or, worse yet, doesn't issue an error message!). On the other hand, had the compiler writer made the table 100 long or 1000 long, most of the time most of the memory space devoted to that table would be wasted.

A compiler written in LISP would be much more likely to keep a linked list of records describing each DO loop. Such a list could be grown at any time by creating a new record on demand and adding it to the list. In this way as many or as few records as needed could be accommodated.

Now one could certainly write a compiler in any language and provide such dynamic storage management with enough programming. The point is that LISP provides automatic storage management from the outset and encourages its use (in much the same way that FORTRAN provides floating-point numbers and encourages their use, even though the particular processor on which a FORTRAN program runs may or may not have floating-point hardware).

Using CAR, CDR, and CONS, we can now write some interesting programs in LISP to deal with LISP data. For example, we can write a program APPEND, which given two lists produces their concatenation as a new list:

```
(DEFINE APPEND
        (LAMBDA (X Y)
                (IF (NULL X) Y
                    (CONS (CAR X) (APPEND (CDR X) Y)))))
```

Because LISP programs are represented as LISP data structures, there is a difficulty with representing constants. For example, suppose we want to determine whether or not the value of the variable X is the symbol "FOO". We might try writing:

```
(EQ X FOO)
```

This doesn't work. The occurrence of "FOO" does not refer to the symbol FOO as a constant; it is treated as a variable, just as "X" is.

The essential problem is that we want to be able to write any LISP datum as a constant in a program, but some data objects must be used to represent other things, such as variables and procedure invocations. To solve

this problem we invent a new notation: (QUOTE d) in a program represents the constant datum d. Thus we can write our test as "(EQ X (QUOTE FOO))". Similarly,

```
(APPEND X (LIST Y Z))
```

constructs a list from the values of Y and Z, and appends the result to the value of X, while

```
(APPEND X (QUOTE (LIST Y Z)))
```

appends to the value of X the constant list "(LIST Y Z)". Because the QUOTE construction is used so frequently in LISP, we use an abbreviated notation: "'FOO" ("FOO" with a preceding quote-mark) is equivalent to "(QUOTE FOO)". This is only a notational convenience; the two notations denote the same list.

A LISP Interpreter

Here is one possible interpreter for the LISP dialect we have described, written in that dialect (this fact makes this interpreter meta-circular — it can interpret itself):

```
(DEFINE EVAL
        (LAMBDA (EXP ENV)
                (IF (ATOM EXP)
                    (IF (NUMBERP EXP) EXP (VALUE EXP ENV))
                    (IF (EQ (CAR EXP) 'QUOTE)
                        (CADR EXP)
                        (IF (EQ (CAR EXP) 'LAMBDA)
                            (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV)
                            (IF (EQ (CAR EXP) 'IF)
                                (IF (EVAL (CADR EXP) ENV)
                                    (EVAL (CADDR EXP) ENV)
                                    (EVAL (CADDDR EXP) ENV))
                                (APPLY (EVAL (CAR EXP) ENV)
                                       (EVLIS (CDR EXP) ENV)))))))))

(DEFINE APPLY
        (LAMBDA (FUN ARGS)
                (IF (PRIMOP FUN) (PRIMOP-APPLY FUN ARGS)
                    (IF (EQ (CAR FUN) '&PROCEDURE)
                        (EVAL (CADDR FUN)
                              (BIND (CADR FUN) ARGS (CADDDR FUN)))
                        (ERROR)))))

(DEFINE EVLIS
        (LAMBDA (ARGLIST ENV)
                (IF (NULL ARGLIST) '()
                    (CONS (EVAL (CAR ARGLIST) ENV)
                          (EVLIS (CDR ARGLIST) ENV)))))
```

The evaluator is divided into two conceptual components: EVAL and APPLY. EVAL classifies expressions and directs their evaluation. Simple expressions (such as constants and variables) can be evaluated directly. For the complex case of procedure invocations (technically called "combinations"), EVAL looks up the procedure definition, recursively evaluates the arguments (using EVLIS), and then calls APPLY. APPLY classifies procedures and directs their application. Simple procedures (primitive operators) are applied directly. For the complex case of user-defined procedures, APPLY uses BIND (see below) to add to the lexical environment, a kind of symbol table, of the procedure, by associating the formal parameters from the procedure definition with the actual argument values provided by EVAL. The body of the procedure definition is then passed to EVAL, along with the environment just constructed, which is used to determine the values of variables occurring in the body.

In more detail, EVAL is a case analysis on the structure of the S-expression EXP. If it is an atom, there are two subcases. Numeric atoms evaluate to themselves. Atomic symbols, however, encode variables; the value associated with that symbol is extracted from the environment ENV using the function VALUE (see below).

If the expression to be evaluated is not atomic, then it may be a QUOTE form, a LAMBDA form, an IF form, or a combination. For a QUOTE form, EVAL extracts the S-expression constant using CADR. LAMBDA forms evaluate to procedure objects (here represented as lists whose cars are the atom "&PROCEDURE") containing the lexical environment and the "text" of the procedure definition. For an IF form, the predicate part is recursively evaluated; depending on whether the result is <u>true</u> or <u>false</u>, the consequent or alternative is selected for evaluation. For combinations, the procedure is obtained, the arguments evaluated (using EVLIS), and APPLY called as described earlier.

EVLIS is a simple recursive function which calls EVAL on successive arguments in ARGLIST and produces a list of the values in order.

APPLY distinguishes two kinds of procedures: primitive and user-defined. For now we avoid describing the precise implementation of primitive procedures by assuming the existence of a predicate PRIMOP which is true only of primitive procedures, and a function PRIMOP-APPLY which deals with the application of such primitive procedures. We consider primitive procedures to be a kind of atomic S-expression other than numbers and atomic symbols; we define no particular written notation for them here. However, primitive procedures are not to be confused with the atomic symbols used as their names. The actual procedure involved in the combination (CAR X) is not the atomic symbol CAR, but rather some bizarre object (the <u>value</u> of the atomic symbol CAR) which is meaningful only to PRIMOP-APPLY.

The interpreter uses several utility procedures for maintaining environments. An environment is represented as a list of buckets; each bucket is a list whose car is a list of names and whose cdr is a list of corresponding values. (Note that this representation is <u>not</u> the same as the

"a-list" representation traditionally used in LISP interpreters.) If a variable name occurs in more than one bucket, the most recently added such bucket has priority; in this way new symbol definitions added to the front of the list can supersede old ones. The code for manipulating environments is below.

```
(DEFINE BIND
        (LAMBDA (VARS ARGS ENV)
                (IF (= (LENGTH VARS) (LENGTH ARGS))
                    (CONS (CONS VARS ARGS) ENV)
                    (ERROR))))

(DEFINE VALUE
        (LAMBDA (NAME ENV)
                (VALUE1 NAME (LOOKUP NAME ENV))))

(DEFINE VALUE1
        (LAMBDA (NAME SLOT)
                (IF (EQ SLOT '&UNBOUND) (ERROR)
                    (CAR SLOT))))

(DEFINE LOOKUP
        (LAMBDA (NAME ENV)
                (IF (NULL ENV) '&UNBOUND
                    (LOOKUP1 NAME (CAAR ENV) (CDAR ENV) ENV))))

(DEFINE LOOKUP1
        (LAMBDA (NAME VARS VALS ENV)
                (IF (NULL VARS) (LOOKUP NAME (CDR ENV))
                    (IF (EQ NAME (CAR VARS)) VALS
                        (LOOKUP1 NAME (CDR VARS) (CDR VALS) ENV)))))
```

BIND takes a list of names, a list of values, and a symbol table, and produces a new symbol table which is the old one augmented by an extra bucket containing the new set of associations. (It also performs a useful error check — LENGTH returns the length of a list.)

VALUE is essentially an interface to LOOKUP. The check for &UNBOUND catches incorrect references to undefined variables.

LOOKUP takes a name and a symbol table, and returns that portion of a bucket whose car is the associated value.

State-Machine Implementation

The LISP interpreter we have presented is recursive. It implicitly relies on a hidden control mechanism which retains the state information which must be saved for each recursive invocation. Here we make this control information explicit. Below we present an interpreter in the form of a state machine controller. The controller manipulates a small set of registers, and

also issues commands to a list memory system. The recursion control information which is typically kept on a stack will be maintained in the linked-list memory.

This evaluator, written in LISP, has five global variables which are used to simulate the registers of a machine. EXP is used to hold the expression or parts of the expression under evaluation. ENV is used to hold the pointer to the environment structure which is the context of evaluation of the current expression. VAL is used to hold the value developed in evaluation of expressions. It is set whenever a primitive operator is invoked, or whenever a variable is evaluated, a quoted expression is evaluated, or a lambda expression is evaluated. ARGS is used to hold the list of evaluated arguments (the "actual parameters") being accumulated for a combination. Finally, CLINK is the pointer to the top of the list structure which is the control stack. (It is called "CLINK" for historical reasons stemming from CONNIVER [McDermott 1974] and "spaghetti stacks" [Bobrow 1973].)

The style of coding here depends on "tail-recursion" (although the current implementations of MacLISP are not really tail-recursive); that is, iterative loops are implemented as patterns of function calls.

EVAL-DISPATCH is the procedure which dispatches on the type of an expression — implementing the action of EVAL. When EVAL-DISPATCH is called, EXP contains an expression to be evaluated, ENV contains the environment for the evaluation, and the top element of CLINK is a "return address", i.e. the name of a function to call when the value has been determined and placed in VAL.

```
(DEFUN EVAL-DISPATCH ()
       (COND ((ATOM EXP)                                ;If an atomic expression:
              (COND ((NUMBERP EXP)                      ; numbers evaluate
                     (SETQ VAL EXP)                     ; to themselves
                     (POPJ-RETURN))                     ; (i.e. are "self-quoting"),
                    (T                                  ; but symbols must be looked
                     (SETQ VAL (VALUE EXP ENV))         ; up in the environment.
                     (POPJ-RETURN))))
             ((EQ (CAR EXP) 'QUOTE)                     ;If a QUOTE expression
              (SETQ VAL (CADR EXP))                     ; extract the quoted constant
              (POPJ-RETURN))                            ; and return it.
             ((EQ (CAR EXP) 'LAMBDA)                    ;If a LAMBDA expression
              (SETQ VAL (CADR EXP))                     ; get the formal parameters,
              (SETQ EXP (CADDR EXP))                    ; get the body,
              (SETQ VAL (LIST '&PROCEDURE VAL EXP ENV)) ; and construct a closure
              (POPJ-RETURN))                            ; which includes ENV.
             ((EQ (CAR EXP) 'IF)                        ;If a conditional,
              (SETQ CLINK (CONS ENV CLINK))             ; save the environment
              (SETQ CLINK (CONS EXP CLINK))             ; save the expression,
              (SETQ CLINK (CONS 'EVIF-DECIDE CLINK))    ; set up a return address,
              (SETQ EXP (CADR EXP))                     ; then extract  the predicate
              (EVAL-DISPATCH))                          ; and evaluate it.
```

```
((NULL (CDR EXP))                          ;If a call with no arguments,
 (SETQ CLINK (CONS 'APPLY-NO-ARGS CLINK))  ; set up a return address,
 (SETQ EXP (CAR EXP))                      ; get the function position
 (EVAL-DISPATCH))                          ; and evaluate it.
(T                                         ;Otherwise,
 (SETQ CLINK (CONS ENV CLINK))             ; save ENV,
 (SETQ CLINK (CONS EXP CLINK))             ; save EXP,
 (SETQ CLINK (CONS 'EVARGS CLINK))         ; set up return address,
 (SETQ EXP (CAR EXP))                      ; get the function position
 (EVAL-DISPATCH))))                        ; and evaluate it.
```

When the process evolved by the evaluator has finished the evaluation of a subexpression, it must continue executing the rest of the expression. The place in the evaluator to continue executing was pushed onto CLINK when the evaluation of the subexpression was begun. This return address is now at the top of the CLINK, where it can be popped off and called:

```
(DEFUN POPJ-RETURN ()                      ;Return to caller:
       (SETQ EXP (CAR CLINK))              ; Save return address in EXP,
       (SETQ CLINK (CDR CLINK))            ;  and pop it off CLINK.
       (FUNCALL EXP))                      ; Transfer control.
```

After the predicate part of a conditional is evaluated, the process comes back to here to look at VAL to see whether the consequent or the alternative branch is to be taken. One of these is selected and made the EXP to be further evaluated.

```
(DEFUN EVIF-DECIDE ()
       (SETQ EXP (CAR CLINK))              ;Restore expression
       (SETQ CLINK (CDR CLINK))            ; and pop it off.
       (SETQ ENV (CAR CLINK))              ;Restore ENV
       (SETQ CLINK (CDR CLINK))            ; and pop it off.
       (COND (VAL                          ;If predicate was true,
              (SETQ EXP (CADDR EXP)))      ; extract consequent.
             (T                            ;Otherwise
              (SETQ EXP (CADDDR EXP))))    ; extract alternative.
       (EVAL-DISPATCH))                    ;In either case, evaluate it.
```

The following procedures are the states the evaluator must go through to evaluate the arguments to procedures before applying them. There is a special-case check in EVAL-DISPATCH for functions with no arguments. In this case, it is not necessary to save the state of the evaluator when evaluating the function position because there are no further arguments to evaluate. One may just apply the procedure which comes back in VAL. This is a case of "evlis tail-recursion" (see [Wand 1977]). We will see this idea again in EVARGS1 where we have a special-case check for evaluation of the last argument.

```
(DEFUN APPLY-NO-ARGS ()
       (SETQ ARGS NIL)                     ;Set up null argument list
       (SAPPLY))                           ; and apply function in VAL.
```

General argument evaluations come to EVARGS. This segment of the evaluator incorporates some cleverness in that it checks for the special case of the last argument in a combination. However, for the sake of clarity and uniformity we did not try to remove all unnecessary pushing and popping. There are many cleverer ways to write this code, as we will see later. The following procedure is the initialization of the argument evaluation loop.

```
(DEFUN EVARGS ()
       (SETQ EXP (CAR CLINK))                    ;Restore EXP
       (SETQ CLINK (CDR CLINK))                  ; and pop it off.
       (SETQ ENV (CAR CLINK))                    ;Restore ENV,
       (SETQ CLINK (CDR CLINK))                  ; and pop it.
       (SETQ CLINK (CONS VAL CLINK))             ;Save function.
       (SETQ EXP (CDR EXP))                      ;Get rid of function part.
       (SETQ ARGS NIL)                           ;Initialize argument list.
       (EVARGS1))                                ;Evaluate arguments.
```

This is the top of the argument evaluation loop.

```
(DEFUN EVARGS1 ()
       (COND ((NULL (CDR EXP))                   ;Is this the last argument?
              (SETQ CLINK (CONS ARGS CLINK))     ;If so, save argument list,
              (SETQ CLINK (CONS 'LAST-ARG CLINK)) ; set up return address,
              (SETQ EXP (CAR EXP))               ; set up last argument,
              (EVAL-DISPATCH))                   ; and evaluate it.
             (T                                  ;Otherwise,
              (SETQ CLINK (CONS ENV CLINK))      ; save ENV,
              (SETQ CLINK (CONS EXP CLINK))      ; save EXP,
              (SETQ CLINK (CONS ARGS CLINK))     ; save argument list,
              (SETQ CLINK (CONS 'EVARGS2 CLINK)) ; set up return address,
              (SETQ EXP (CAR EXP))               ; set up next argument,
              (EVAL-DISPATCH))))                 ; and evaluate it.
```

This is the place where we end up after each argument is evaluated. The evaluated argument is accumulated into ARGS.

```
(DEFUN EVARGS2 ()
       (SETQ ARGS (CAR CLINK))                   ;Restore argument list,
       (SETQ CLINK (CDR CLINK))                  ; and pop it off.
       (SETQ EXP (CAR CLINK))                    ;Restore EXP,
       (SETQ CLINK (CDR CLINK))                  ; and pop it off.
       (SETQ ENV (CAR CLINK))                    ;Restore ENV,
       (SETQ CLINK (CDR CLINK))                  ; and pop it off.
       (SETQ ARGS (CONS VAL ARGS))               ;Add value to argument list.
       (SETQ EXP (CDR EXP))                      ;Flush form just evaluated.
       (EVARGS1))                                ;Go evaluate next argument.
```

When the last argument has been evaluated we come back here. The value is accumulated onto the ARGS and the function is restored from the stack. The whole mess is then shipped to SAPPLY for application.

```
(DEFUN LAST-ARG ()
       (SETQ ARGS (CAR CLINK))                ;Restore argument list,
       (SETQ CLINK (CDR CLINK))               ; and pop it off.
       (SETQ ARGS (CONS VAL ARGS))            ;Add last value to it.
       (SETQ VAL (CAR CLINK))                 ;Retrieve function,
       (SETQ CLINK (CDR CLINK))               ; and pop it off.
       (SAPPLY))                              ;Apply function to arguments.
```

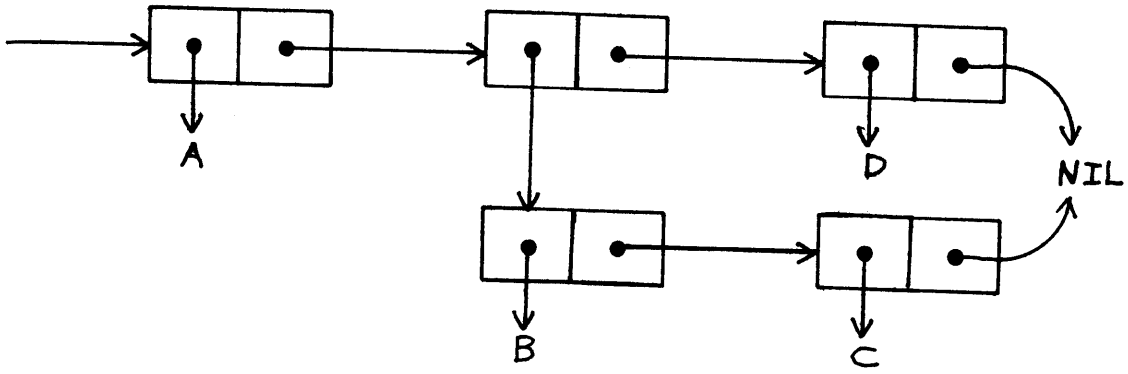
SAPPLY is the state machine analog of APPLY. This procedure checks out what kind of procedure is to be applied. If it is primitive, the appropriate magic occurs. If it is a procedural closure, we evaluate the body of the closed procedure in an environment constructed by binding the formal parameters of the closed procedure to the actual parameters (in ARGS) in the environment carried in the closure.

```
(DEFUN SAPPLY ()                              ;Apply function in VAL to ARGS.
       (COND ((PRIMOP? VAL)                   ;If a primitive procedure,
              (SETQ VAL (PRIMOP-APPLY VAL ARGS))  ; do it!
              (POPJ-RETURN))                  ; then return value to caller.
             ((EQ (CAR VAL) '&PROCEDURE)      ;If a defined procedure,
              (SETQ ENV                       ; set up its environment
                    (BIND (CADR VAL)          ; by binding the formals
                          ARGS                ; to the actuals
                          (CADDDR VAL)))      ; in the closure environment
              (SETQ EXP (CADDR VAL))          ; then get the procedure body
              (EVAL-DISPATCH))                ; and evaluate it.
             (T (ERROR))))                    ;Otherwise, error.
```

In this state-machine code we have avoided functional composition. Each statement is an assignment or a conditional. (We have used the usual LISP COND conditional form, rather than IF, for reasons of convenience. This interpreter is not meta-circular. Instead, it is working MacLISP code which implements a non-MacLISP version of LISP.) An assignment can contain at most one call to a storage management procedure such as CONS or CAR (we allow calls to e.g. CADDR, realizing that (SETQ X (CADDR Y)) can be considered an abbreviation for the sequence (SETQ X (CDR Y)), (SETQ X (CDR X)), (SETQ X (CAR X))). Also, VALUE and BIND can be considered here to be complex storage operations (defined essentially as before).

Representing List Data

Lists are normally represented by records each of which contains two pointers to other records. One pointer is the car, and the other is the cdr. In this way a list (A (B C) D) can be visualized by the following diagram:



Atoms are represented as records of other types.

The exact representation of a pointer is not of interest here. All we really care about is that if we give the pointer to the memory system, it can return the contents of the record pointed to. (In particular, there is nothing at this level requiring the parts of a record to be "contiguous". Later we will discuss ways to represent LISP data within standard linear memories.)

In our particular architecture, we find it convenient to associate with each pointer a type field describing the nature of the record pointed to. This type field can be exploited for other purposes as well; in particular, we shall use it to encode "opcodes" and "return addresses". We will say that the type field is a part of the pointer, and that the other part of the pointer (that which identifies another record) is the address part. The list shown above, with type fields added, looks like this:

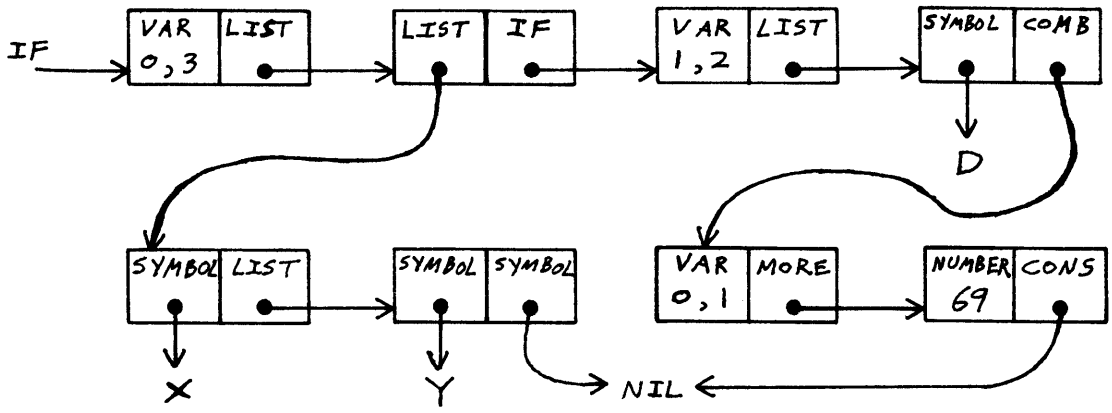
One efficiency problem with the version of the LISP interpreter given above is that the repeated consecutive tests for atoms, LAMBDA, IF, etc. take time. Conceptually what these tests are doing is a dispatch on the syntactic category of the expression. Each expression is distinguished by a special symbol in the car position — except for atoms and procedure calls. The evaluator could be made faster and simpler if it could dispatch in a more uniform way.

Another efficiency problem is that LOOKUP must search for the values of variables. Because our dialect of LISP is lexically scoped like ALGOL, we can arrange for variable references to specify "n levels back, and then j over" in

much the same way used by the ALGOL "display" technique, eliminating the search.

To allow these efficiencies we systematically alter the representation of programs. They will still be represented as trees of list records, but we encode the syntactic categories in the type fields of pointers to the expressions. EVAL can then simply dispatch on this type code. For a pointer whose type is "variable reference", we use the address part as "immediate data" indicating the values of n and j for the environment lookup. We draw a piece of program in this way:

```
(IF A '(X Y) (IF C 'D (CONS E 69)))
```



Because variable references and other constructs have separate types, lists and symbols can be "self-evaluating" in the same way as numbers. Also, we assume that CONS is a "built-in" primitive operator, and encode that operator in the type field at the end of the list representing the call to CONS. The encoding of IF forms has been changed to reduce space requirements; the car of a conditional is the predicate, and the cdr is a cons cell whose car is the consequent and whose cdr is the alternative.

We ought perhaps to define a printed representation for all these new data types. We do not do this here, however. We assume that in practice one will write LISP code in the usual style, and a simple "compiler" program will transform it into the typed-pointer representation.

To describe the evaluator for this new representation, we introduce a construct TYPE-DISPATCH which is not part of the LISP language, but which we use to indicate the underlying mechanism. We also use a primitive operator TYPED-CONS, which creates a new list-like cell with a specified type. The new evaluator is very much like the old one: EVCOMB is sort of like EVLIS combined with the entry point to APPLY.

```
(DEFINE EVAL
        (LAMBDA (EXP ENV)
                (TYPE-DISPATCH EXP
                               ("NUMBER" EXP)
                               ("SYMBOL" EXP)
                               ("LIST" EXP)
                               ("VARIABLE" (DISPLAY-LOOKUP EXP ENV))
                               ("PROCEDURE" (TYPED-CONS "CLOSURE" (CDR EXP) ENV))
                               ("IF" (IF (EVAL (CAR EXP) ENV)
                                         (EVAL (CADR EXP) ENV)
                                         (EVAL (CDDR EXP) ENV)))
                               ("COMBINATION"  (EVCOMB (CDR EXP)
                                                       ENV
                                                       (CONS (EVAL (CAR EXP) ENV)
                                                             '())))
                               (OTHERWISE (ERROR)))))

(DEFINE EVCOMB
        (LAMBDA (EXP ENV ARGS)
                (TYPE-DISPATCH EXP
                               ("MORE-ARGS" (EVCOMB (CDR EXP)
                                                    ENV
                                                    (CONS (EVAL (CAR EXP) ENV)
                                                          ARGS)))
                               (OTHERWISE (APPLY EXP ARGS)))))

(DEFINE APPLY
        (LAMBDA (FUN ARGS)
                (TYPE-DISPATCH FUN
                               ("FUNCALL" (EVAL (CDAAR ARGS)
                                                (DISPLAY-BIND (CDR ARGS)
                                                              (CDAR ARGS))))
                               ("CONS" (CONS (CADR ARGS) (CAR ARGS)))
                               ("CAR" (CAAR ARGS))
                               ("CDR" (CDAR ARGS))
                               ("ATOM" (ATOM (CAR ARGS)))
                               ...
                               (OTHERWISE (ERROR)))))
```

When a non-"MORE-ARGS" type code is seen in EVCOMB, it indicates that a primitive operation is to be performed on the argument values. EVCOMB then calls APPLY to perform this operation. (As shown here, APPLY needlessly duplicates the dispatching operation in EVCOMB; we have done this to exhibit the similarity of this interpreter to the previous one. Later we will remove this duplication.) One of these primitive operations, "FUNCALL", is used to invoke user-defined procedures (closures). (The type codes used to indicate primitive operations may overlap those used to distinguish syntactic categories, because they are used in different contexts. Compare this to the way in which the same bits in an instruction can be used for different purposes depending on the opcode; for example, in the PDP-11 the same bits of an instruction word can be a register number, part of a branch offset, or

condition code bits.)

Combining these Ideas

The state machine implementation of a LISP interpreter can be combined with the typed pointer dispatch idea to form a very efficient interpreter for LISP which can be easily implemented directly in hardware. We now present such an interpreter, written in a statement-oriented language to emphasize that we are describing a hardware interpreter. As before, the controller manipulates a small set of registers, and also issues commands to a list memory system. The recursion-control information is, as before, stored in a push-down control list maintained in linked-list memory. Type fields in the cdr pointers of the control list will be used to retain "return addresses" within the state machine; in this way return addresses do not require any extra conses in the CLINK. (Compare this with the previous state-machine interpreter, which used separate tokens in the CLINK as return addresses.) This is possible because the set of return addresses is small.

```
BEGIN "EVALUATOR"
        DECLARE REGISTERS
                  EXP           !GENERALLY HOLDS EXPRESSION BEING EVALUATED
                  ENV           !HOLDS CURRENT ENVIRONMENT
                  VAL           !RESULT OF EVALUATION; ALSO SCRATCH
                  ARGS          !ACCUMULATES EVALUATED ARGUMENTS OF A COMBINATION
                  CLINK         !"CONTROL LINK": RECURSION CONTROL STACK
        EVAL:   TYPE-DISPATCH ON EXP INTO
                        "NUMBER": GOTO SELF
                        "SYMBOL": GOTO SELF
                        "LIST": GOTO SELF
                        "VARIABLE": GOTO LOOKUP
                        "PROCEDURE": GOTO PROC
                        "IF": GOTO IF1
                        "COMBINATION": GOTO EVCOMB
                HCTAPSID-EPYT
        SELF:   VAL := EXP; GOTO RETURN
        PROC:   VAL := TYPED-CONS("CLOSURE", EXP, ENV); GOTO RETURN
        IF1:    VAL := CDR(EXP)
                CLINK := CONS(ENV, CLINK)
                CLINK := TYPED-CONS("IF2", VAL, CLINK)
                EXP := CAR(EXP); GOTO EVAL
        !RECURSIVE EVALUATION OF PREDICATE RETURNS HERE
        IF2:    EXP := CAR(CLINK)
                CLINK := CDR(CLINK)
                ENV := CAR(CLINK)
                CLINK := CDR(CLINK)
                IF NULL(VAL)
                   THEN EXP := CDR(EXP); GOTO EVAL
                   ELSE EXP := CAR(EXP); GOTO EVAL
                FI
```

```
        EVCOMB: ARGS := '()
        EVCOM1: TYPE-DISPATCH ON EXP INTO
                        "COMBINATION": GOTO EVCOM2
                        "FUNCALL": GOTO CALL
                        "CONS": GOTO CONS
                        "CAR": GOTO CAR
                        "CDR": GOTO CDR
                        ...
                HCTAPSID-EPYT
        EVCOM2: CLINK := CONS(ENV, CLINK)
                CLINK := CONS(ARGS, CLINK)
                VAL := CDR(EXP)
                CLINK := TYPED-CONS("EVCOM3", VAL, CLINK)
                EXP := CAR(EXP); GOTO EVAL
        !RECURSIVE EVALUATION OF ARGUMENT RETURNS HERE
        EVCOM3: EXP := CAR(CLINK)               !UNWIND STACK
                CLINK := CDR(CLINK)
                ARGS := CAR(CLINK)
                CLINK := CDR(CLINK)
                ENV := CAR(CLINK)
                CLINK := CDR(CLINK)
                ARGS := CONS(VAL, ARGS); GOTO EVCOM1
        CALL:   ARGS := CDR(ARGS)               !N.B. VAL = CAR(ARGS)
                EXP := CAR(VAL)
                VAL := CDR(VAL)
                ENV := CONS(ARGS, VAL); GOTO EVAL
        CONS:   ARGS := CDR(ARGS)               !I.E. ARGS := CADR(ARGS)
                ARGS := CAR(ARGS)               !(ALREADY HAD VAL := CAR(ARGS), IN EFFECT)
                VAL := CONS(ARGS, VAL); GOTO RETURN
        CAR:    VAL := CAR(VAL); GOTO RETURN
        CDR:    VAL := CDR(VAL); GOTO RETURN
        ...
        RETURN: TYPE-DISPATCH ON CLINK INTO
                        "IF2": GOTO IF2
                        "EVCOM3": GOTO EVCOM3
                HCTAPSID-EPYT                   !SESOL ARTSKJID
END "EVALUATOR"
```

In this state-machine code we have avoided functional composition rigorously. Each statement is an assignment or a dispatch operation (IF-THEN-ELSE being a kind of dispatch). As assignment can contain at most one call to a simple storage management procedure such as CONS or CAR. Each statement goes to another statement (to the one textually following, if no GOTO clause is present).

We have omitted the details of the LOOKUP operation (it gets the value from the environment and then goes to RETURN). We have, however, shown DISPLAY-BIND (beginning at CALL). These are not done as subroutines (as they were in the previous state-machine interpreter); they are coded "in-line" as state-machine code.

Recursive evaluation of subexpressions is handled by using an explicit stack. When for an IF or a COMBINATION a recursive evaluation is needed, any required registers (e.g. ENV) are consed onto the control structure CLINK. The last cons onto CLINK uses the type code to encode the "return address" (IF2 or EVCOM3) within the state machine. (These return address codes may be the same codes used as "opcodes" or "primitive operator codes" — this is a third, distinct context in which type bits are used for some funny purpose unrelated to the type of the data.) The expression to be recursively evaluated is put into EXP, and then state EVAL is entered. When the evaluation finishes, the code at RETURN decodes the type field of CLINK and resumes execution of the caller, which retrieves the saved information from CLINK and carries on. Thus CLINK, though implemented as linked records, behaves as a stack.

This is in fact how we have implemented a LISP evaluator in the form of a VLSI microprocessor. There are five registers on a common bus (the E bus). The state machine is in the form of a read-only memory plus a "mitro-PC" which encodes the current state. At each transition the EVAL state machine can read one register onto the E bus, load one or more other registers from the E bus, request some storage operation to occur, and enter some new state (possibly computed by dispatching on bits obtained from the E bus). Only one operand can be passed at a time to the storage manager (via the bus), and so an operation such as CAR is actually managed as two operations:

(1) pass operand to storage manager and request CAR;
(2) retrieve result of storage operation.

Similarly, CONS is managed as three operations:

(1) pass the cdr part to storage manager;
(2) pass the car part, and request CONS;
(3) retrieve result.

Often operations can be "bummed out"; for example, after requesting a CAR, the result need not be retrieved if it is to be used immediately as one operand of a CONS. In this case (CONS (CAR X) C) takes only three transactions, not five.

Storage Management

A complete LISP system, as implied in the previous section, is conveniently divided into two parts: (1) a storage system, which provides an operator for the creation of new data objects and also other operators (such as pointer traversal) on those objects; and (2) a program interpreter (EVAL), which executes programs expressed as data structures within the storage system. (Note that this memory/processor division characterizes the usual von Neumann architecture also. The differences occur in the nature of the processor and the memory system.)

Most hardware memory systems which are currently available commercially are not organized as sets of linked lists, but rather as the usual linearly-indexed vectors. (More precisely, commercially available RAMs are organized as Boolean N-cubes indexed by bit vectors. The usual practice is to impose a total ordering on the memory cells by ordering their addresses lexicographically, and then to exploit this total ordering by using indexing

hardware typically containing an addition unit (or, more rarely, a subtraction unit, as on the IBM 7094).)

Commercially available memories are, moreover, available only in finite sizes (more's the pity). Now the free and wasteful throw-away use of data objects would cause no problem if infinite memory were available, but within a finite memory it is an ecological disaster. In order to make such memories useable to our processor we must interpose between EVAL and the storage system a storage manager which makes a finite vector memory appear to the evaluation mechanism to be an infinite linked-record memory. This would seem impossible, and it is; the catch is that at no time may more records be active than will fit into the finite memory actually provided. The memory is "apparently infinite" in the sense that an indefinitely large number of new records can be "created" using the CONS operator. The storage manager recycles discarded records in order to create new ones in a manner completely invisible to the evaluator.

The storage manager therefore consists of routines which implement the operations CAR, CDR, CONS, etc. in terms of the vector memory, plus a garbage collector which deals with the finiteness of the memory by locating records which have been discarded and making them available to the CONS routine for recycling.

The method we use for implementing CAR, CDR, and CONS is the usual one of using two consecutive words of memory to hold a list cell, the first being the cdr and the second the car, where each word of memory can hold a type field and an address field. The address part of a pointer is in turn the address within the linear memory of the record pointed to. (This may seem obvious, but remember that until now we have been noncommittal about the precise representation of pointers, as until this point all that was necessary was that the memory system associate records with pointers by any convenient means whatsoever. The evaluator is completely unconcerned with the format or meaning of addresses; it merely accepts them from the memory system and eventually gives them back later to retrieve record components. One may think of an address as a capability for accessing a record using certain defined operations.)

Many techniques for garbage collection are well-documented in the literature [McCarthy 1962] [Minsky 1963] [Hart 1964] [Saunders 1964] [Schorr 1967] [Conrad 1974] [Baker 1978] [Morris 1978], and will not be discussed here. Suffice it to say here that, in the prototype processor we have designed, the storage manager is implemented as a second state machine. It also has a small set of registers on a second bus (the G bus). The storage manager runs continuously, performing services for the evaluator. When the storage manager has completed a request, it then advances the evaluator to its next state, and dispatches on the new request from the evaluator. The storage manager can connect the E bus and G bus together in order to retrieve an operand or return a result (which, if either, is to be done is determined by the request protocol). The storage manager can also read from or write into the off-chip memory.

(In fact, in the prototype processor, the storage manager includes no garbage collector. The prototype was one project of a "project set" including some two dozen separate circuits, all of which had to be fit onto a single chip together. This imposed severe area limitations which restricted the address size to eight bits, and required the elimination of the microcode for the garbage collector. We anticipate no obstacles to including a garbage collector in a full-sized single-chip processor. The complexity of a simple garbage collector is comparable to that of the evaluator shown above.)

Physical Layout of the Prototype Processor

The evaluator and the storage manager are each implemented in the same way as an individual processor. Each processor has a state-machine controller and a set of registers. On each clock cycle the state-machine outputs control signals for the registers and also makes a transition to a new state.

The contents of any register is a pointer, containing an address field (8 bits in the prototype) and a type field (3 bits in the prototype). The registers of a processor are connected by a common bus (E bus in the evaluator, G bus in the storage manager). Signals from the controller can read at most one register onto the bus, and load one or more other registers from the bus. One register in each controller has associated incrementation logic; the controller can cause the contents of that register, with 1 added to its address part, to be read onto the bus. The controller can also force certain constant values onto the bus rather than reading a register.

The processors can communicate with each other by causing the E and G busses to be connected. The address and type parts of the busses can be connected separately. (Typically the E bus might have its address part driven from the G bus and its type part driven by a constant supplied by the evaluator controller.) The G bus can also be connected to the address/data lines for the off-chip memory system. The storage-manager controller produces additional signals (ADR and WRITE) to control the external memory. In a similar manner, the evaluator controller produces signals which control the storage manager. (Remember that from the point of view of the evaluator, the storage manager <u>is</u> the memory interface!)

Each controller effectively has an extra "state register" which may be thought of as its "micro-PC". At each step the next state is computed by combining its current state with external signals in the following manner. Each "microinstruction" has a field explicitly specifying the next desired state, as well as bits specifying possible modifications of that state. If specified, external signals are logically OR'd into the desired state number. In the prototype evaluator these external signals are: (1) the type bits from the E bus; (2) a bit which is 1 iff the E bus type field is zero and a bit which is 1 iff the E bus address is zero. In the storage manager these signals are: (1) the four control bits from the evaluator controller; (2) a bit which is 1 iff the G bus address is zero. This is the way in which dispatching is achieved.

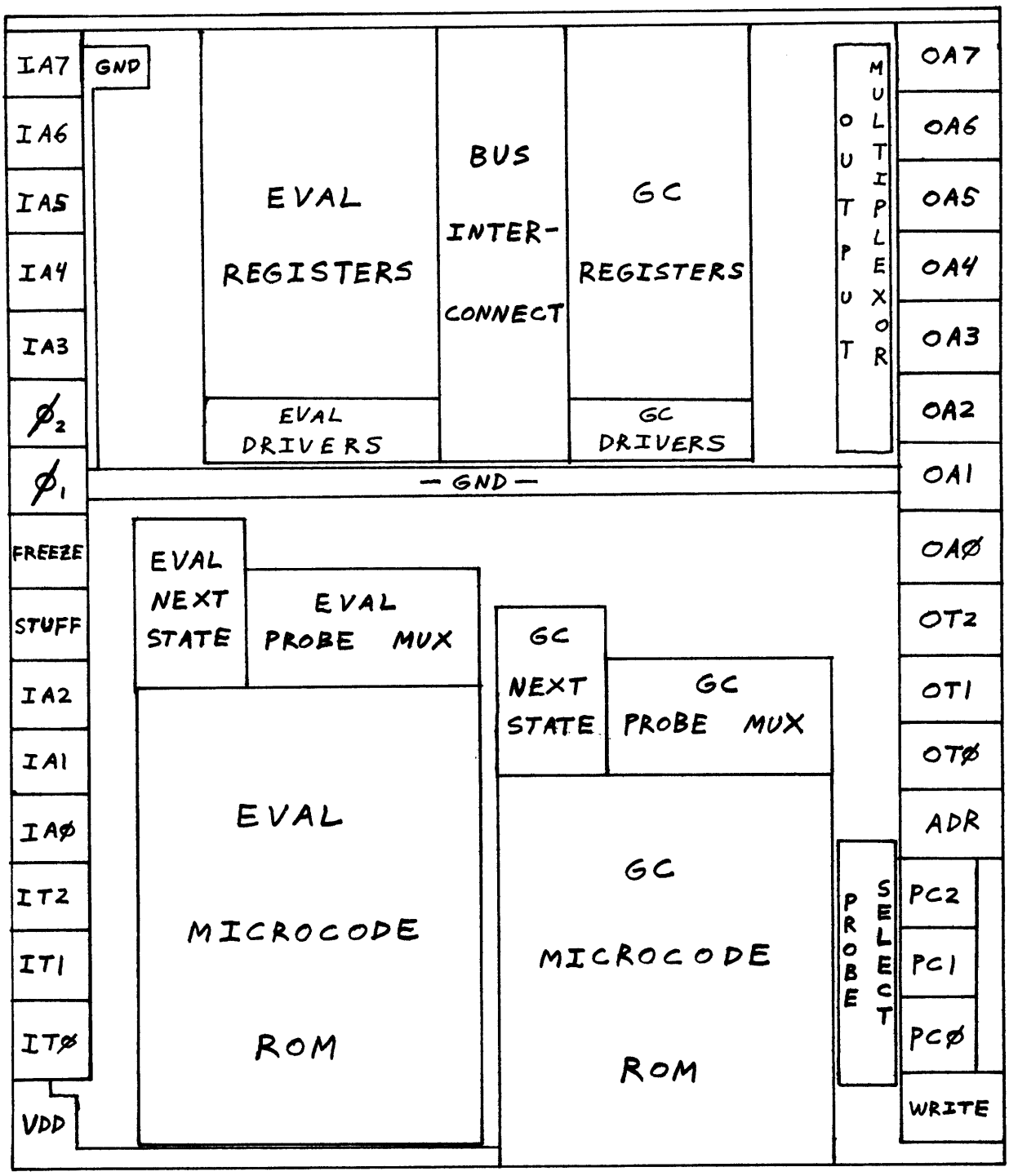
Once this new state is computed, it is passed through a three-way selector before entering the state register. The other two inputs to the selector are the current state and the data lines from the external memory system. In this way the selector control can "freeze" a controller in its current state by recirculating it, or jam an externally supplied state into the state register (both useful for debugging operations). The "freeze" mechanism is used by the storage manager to suspend the evaluator until it is ready to process the next request. In the same way, the external memory can suspend the storage manager by asserting the external FREEZE signal, thereby causing a "wait state".

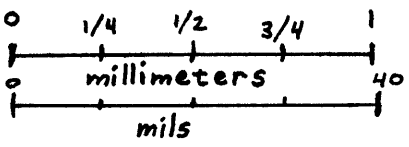
(The FREEZE signal is provided as a separate control because the dynamic logic techniques usual in NMOS were used; if one stopped the processor simply by stopping the clock, the register contents would dissipate. The clocks must keep cycling in order to "refresh" the registers. The state recirculation control allows the machine to be logically stopped despite the fact that data is still circulating internally. We discovered that this technique imposed constraints on other parts of the design: the incrementation logic is the best example. It was originally intended to design an incrementing counter register, which would increment its contents in place during the recirculation of a clock cycle in which an "increment" signal was asserted. If this had been done, however, and the processor were frozen during an instruction which asserted this signal, the counter would continue to count while the processor was stopped! This could have been patched by having the FREEZE signal override the increment signal, but it was deemed simpler to adopt a design strategy in which nothing at the microcode level called for any data to be read, modified, and stored back into the same place. Thus in the actual design one must read data through modification logic and then onto the bus, to be stored in a different register; then if this operation is repeated many times because of the FREEZE signal it makes no difference.)

Each state-machine controller consists of a read-only memory (implemented as a programmed-logic-array), two half-registers (clocked inverters, one at each input and one at each output), and some random logic (e.g. for computing the next state). The controllers are driven by externally supplied two-phase non-overlapping clock signals; on phase 1 the registers are clocked and the next state is computed, and on phase 2 the next-state signals appear and are latched.

All of the signals from the two controllers (62 = 34+28 in the prototype) are multiplexed onto twelve probe lines by six unary probe-control signals. (These signals are derived from three binary-encoded off-chip signals.) When a probe-control signal is asserted, the memory output pads (11 data pads plus the ADR signal in the prototype) are disconnected from the G bus and connected to the twelve probe lines. In this way the chip can be frozen and then all controller outputs verified (by cycling the probe-control signals through all six states). Also recall that the controller states can be jammed into the state registers from the memory input pads. This should allow the controller microcode to be tested completely without depending on the registers and busses working.

The following diagram shows the physical layout of the prototype chip. The two controllers are side by side, with the evaluator on the left and the storage manager on the right. Above each controller is the next-state logic and probe multiplexor for that controller. Above those are the register arrays, with the busses running horizontally through them. The bus connections are in the center. The input pads are on the left edge, and the output pads on the right edge. The input pads are bussed through the evaluator's register array parallel to the E bus lines, so that they can connect to the G bus. (Unfortunately, there was no time to design tri-state pads for this project.)





Physical layout of prototype LISP processor

3.96 mm x 3.38 mm

156 mils x 133 mils

Discussion

A perhaps mildly astonishing feature of this computer is that it contains no arithmetic-logic unit (ALU). More precisely, it does have arithmetic and logical capabilities, but the arithmetic units can only add 1, and the logical units can only test for zero. (Logicians know that this suffices to build a "three-counter machine", which is known to be as universal (and as convenient!) as a Turing Machine. However, our LISP architecture is also universal, and considerably more convenient.)

LISP itself is so simple that the interpreter needs no arithmetic to run interesting programs (such as computing symbolic derivatives and integrals, or pattern matching). All the LISP interpreter has to do is shuffle pointers to and from memory, and occasionally dispatch on the type of a pointer. The incrementation logic is included on the chip for two reasons. In the evaluator it is used for counting down a list when looking up lexical variables in the environment; this is not really necessary, for there are alternative environment representation strategies. In the storage manager incrementation is necessary (and, in the prototype, sufficient) for imposing a total ordering on the external memory, so as to be able to enumerate all possible addresses. The only reason for adding 1 is to get to the next memory address. (One might note that the arithmetic properties of general two-argument addition are not exploited here. Any bijective mapping from the set of external memory addresses onto itself (i.e. a permutation function) would work just fine (but the permutation should contain only one cycle if memory is not to be wasted!). For example, subtracting 1 instead of adding, or Gray-code incrementation, would do.)

This is not to say that real LISP programs do not ever use arithmetic. It is just that the LISP interpreter itself does not require binary arithmetic of the usual sort (but it does require CONS, CAR, and CDR, which in a formal sense indeed form a kind of "number system" [Levin 1974], where CONS corresponds to "add 1" and both CAR and CDR to "subtract 1" — in this view, the purpose of the storage manager is to interface between two kinds of arithmetic, namely "LISP arithmetic" and Peano arithmetic). This architecture is intended to use devices which are addressed as memory, in the same manner used by the PDP-11, for example. We envision having a set of devices on the external memory bus which do arithmetic. One would then write operands into specific "memory locations" and then read arithmetic results from others. Such devices could be very complex processors in themselves, such as specialized array or string processors. In this way the LISP computer could serve as a convenient controller for other processors, for one thing LISP does well is to provide recursive control and environment handling without much prejudice (or expertise!) as to the data being operated upon.

Expanding on this idea, one could arrange for additional signals to the external memory system from the storage manager, such as "this data item is needed (or not needed)", which would enable external processors to do their own storage management cooperatively with the LISP processor. One might imagine, for example, an APL machine which provided tremendous array processing power, controlled by a LISP interpreter specifying which operations to perform. The APL machine could manage its own array storage, using a

relatively simple storage manager cued by "mark" signals from the LISP storage manager.

The possibility of additional processors aside, this architecture exhibits an interesting layered approach to machine design. One can draw boundaries at various places such that everything above the boundary is a processor which treats everything below the boundary as a memory system with certain operations. If the boundary is drawn between the evaluator and the storage manager, then everything below the boundary together constitutes a list-structure memory system. If it is drawn between the storage manager and the external memory, then everything below the boundary is the external memory. Supposing the external memory to be a cached virtual memory system, then we could draw boundaries between the cache and main memory, or between main memory and disks, and the same observation would hold. At the other end of the scale, a complex data base management system could be written in LISP, and then the entire LISP chip (plus some software, perhaps in an external ROM) would constitute a memory system for a data base query language interpreter. In this manner we have a layered series of processors, each of which provides a more sophisticated memory system to the processor above it in terms of the less sophisticated memory system below it.

Another way to say this is that we have a hierarchy of data abstractions, each implemented in terms of a more primitive one. Thus the storage manager makes a finite, linear memory look "infinite" and tree-structured. A cache system makes a large, slow memory plus a small, fast memory look like a large, fast memory.

Yet another way to view this is as a hierarchy of interpreters running in virtual machine. Each layer implements a virtual machine within which the next processor up operates.

It is important to note that we may choose any boundary and then build everything below it in hardware and everything above it in software. Our LISP system is actually quite similar to those before it, except that we have pulled the hardware boundary much higher. One can also put different layers on different chips (as with the LISP chip and its memory). We choose to put the evaluator and the storage manager on the same chip only because (a) they fit, and (b) in the planned full-scale version, the storage manager would need too many pins as a separate chip.

Each of the layers in this architecture has much the same organization: it is divided into a controller ("state machine") and a data base ("registers"). There is a reason for this. Each layer implements a memory system, and so has state; this state is contained in the data base (which may be simply a small set of references into the next memory system down). Each layer also accepts commands from the layer above it, and transforms them into commands for the layer below it; this is the task of the controller.

We have already mentioned some of the analogies between a LISP-based processor and a traditional processor. Corresponding to indexing there is

component selection; corresponding to a linearly advancing program counter there is recursive tree-walk of expressions. Another analogy we might draw is to view the instruction set as consisting of variable-length instructions (whose pieces are joined by pointers rather than being arranged in sequential memory locations). Each instruction (variable reference, call to CONS, call to use function, etc.) takes a number of operands. We may loosely say that there are two addressing modes in this architecture, one being immediate data (as in a variable reference), and the other being a recursive evaluation. In the latter case, merely referring to an operand automatically calls for the execution of an entire routine to compute it!

Project History

In January 1978 one of us (Sussman) attended a course given at MIT by Charles Botchek about the problems of integrated circuit design. There he saw pictures of processors such as 8080's which showed that half of the chip area was devoted to arithmetic and logical operations and associated data paths. On the basis of our previous work on LISP and SCHEME [Sussman 1975] [Steele 1976a] [Steele 1976b] [Steele 1977] [Steele 1978a] [Steele 1978b] it occurred to him that LISP was sufficiently simple that almost all the operations performed in a LISP interpreter are dispatches and register shuffles, and require almost no arithmetic. He concluded that if you could get rid of the ALU in a microprocessor, there would be plenty of room for a garbage collector, and one could thus get an entire LISP system onto a chip. He also realized that typed pointers could be treated as instructions, with the types treated as "opcodes" to be dispatched on by a state machine. (The idea of typed pointers came from many previous implementations of LISP-like languages, such as MUDDLE [Galley 1975], ECL [Wegbreit 1974], and the LISP Machine [Greenblatt 1974]. However, none of these uses the types as opcodes in the evaluator. This idea stemmed from an aborted experiment in nonstandard LISP compiler design which we performed in 1976.)

"THEY LAUGHED WHEN I SAT DOWN AT THE PIANO...
but when I started to play!—"
— John Caples [Caples 1925]

Jon Allen thought building such a processor was a fine idea, but everyone else laughed. The other of us (Steele) laughed loudest, but promised to help work on it. In February 1978 we wrote down a state machine specification for a LISP evaluator and put it on the shelf.

In the summer of 1978 Sussman wrote a LISP interpreter based on the state machine specification. It worked.

In the fall of 1978 Lynn Conway came to MIT from Xerox PARC as a visiting professor to teach a subject (i.e. course) on VLSI design which she developed with Carver Mead of Caltech. Sussman suggested that Steele take the course "because it would be good for him" (and also because he couldn't sit in himself because of his own teaching duties). Steele decided that it might be interesting. So why not?

The course dealt with the structured design of NMOS circuits. As part of the course each student was to prepare a small project, either individually or collaboratively. (This turned out to be a great success. Some two dozen projects were submitted, and nearly all were fit together onto a single 7 mm x 10 mm project chip for fabrication by an outside semiconductor manufacturer and eventual testing by the students.)

Now Steele remembered that Sussman had claimed that a LISP processor on a chip would be simple. A scaled-down version seemed appropriate to design for a class project. Early estimates indicated that the project would occupy 2.7 mm x 3.7 mm, which would be a little large but acceptable. (The average student project was a little under 2 mm x 2 mm.) The LISP processor prototype project would have a highly regular structure, based on programmed logic array cells provided in a library as part of the course, and on a simple register cell which could be replicated. Hence the project looked feasible. Steele began the design on November 1, 1978.

The various register cells and other regular components took about a week to design. Another week was spent writing some support software in LISP, including a microassembler for the microcode PLAs; software to produce iterated structures automatically, and rotate and scale them; and an attempt to write a logic simulator (which was "completed", but never debugged, and was abandoned after three days).

The last three weeks were spent doing random interconnect of PLA's to registers and registers to pads. The main obstacle was that there was no design support software for the course other than some plotting routines. All projects had to be manually digitized and the numbers typed into computer files by keyboard (the digitization language was the Caltech Intermediate Format (CIF)). This was rather time-consuming for all the students involved.

In all the design, layout, manual digitization, and computer data entry for this project took one person (Steele) five weeks of full-time work spanning five and one-half weeks (with Thanksgiving off). This does not include the design of the precise instruction set to be used, which was done in the last week of October (and later changed!). (The typical student project also took five weeks, but presumably with somewhat less than full-time effort.)

During this time some changes to the design were made to keep the area down, for as the work progressed the parts inexorably grew by 20 microns here and 10 microns there. The number of address bits was chopped from ten to eight. A piece of logic to compare two addresses for equality (to implement the LISP EQ operation) was scrapped (this logic was to provide an additional dispatch bit to the evaluator in the same group as the E-bus-type-zero bit and the E-bus-address-zero bit). The input pad cell provided in the library had to be redesigned to save 102 microns on width. The WRITE pad was connected to the bottom of the PLA because there was no room to route it to the top, which changed the clock phase on which the WRITE signal rose, which was compensated for by rewriting the microcode on the day the project was due (December 6, 1978). Despite these changes, the area nevertheless increased. The final

design occupied 3.378 mm x 3.960 mm.

The prototype processor layout file was merged with the files for the other students' projects, and the project chip was sent out for fabrication. Samples were packaged in 40-pin DIPs and in the students' hands by mid-January 1979. As of March 1979, several (more than three) of the nineteen projects on the chip had been tested and found to work.

We intend to implement a full-scale version of a LISP processor in 1979, using essentially the same design strategies. The primary changes will be the introduction of a full garbage collector and an increase in the address space and number of types. We have tentatively chosen a 41-bit word, with 31 bits of address, 5 bits of type, 3 bits of "cdr code", and 2 bits for the garbage collector.

Conclusions

We have presented a general design for and a specific example of a new class of hardware processors. This model is "classical" in that it exhibits the stored-program, program-as-data idea, as well as the processor/memory dichotomy which leads to the so-called "von Neumann bottleneck" [Backus 1978]. It differs from the usual stored-program computer in organizing its memory differently, and in using an instruction set based on this memory organization. Where the usual computer treats memory as a linear vector and executes a linear instruction stream, the architecture we present treats memory as linked records, and executes a tree-shaped program by recursive expression evaluation.

The processor described here is not to be confused with the "LISP Machine" designed and built at MIT by Greenblatt and Knight [Greenblatt 1974 [Knight 1974] [LISP Machine 1977] [Weinreb 1978]. The current generation of LISP Machine is built of standard TTL logic, and its hardware is organized as a very general-purpose microprogrammed processor of the traditional kind. It has a powerful arithmetic-logic unit and a large writable control store. Almost none of the hardware is specifically designed to handle LISP code; it is the microcode which customizes it for LISP. Finally, the LISP Machine executes a compiled order code which is of the linearly-advancing-PC type; the instruction set deals with a powerful stack machine. Thus the LISP Machine may be thought of as a hybrid architecture that takes advantage of linear vector storage organization and stack organization as well as linked-list organization. In contrast, the class of processors we present here is organized purely around linked records, especially in that the instruction set is embedded in that organization. The LISP Machine is a well-engineered machine for general-purpose production use, and so uses a variety of storage-management techniques as appropriate. The processor described here is instead intended as an illustration of the abstracted essence of a single technique, with as little additional context or irrelevant detail as possible.

We have designed and fabricated a prototype LISP-based processor. The

actual hardware design and layout was done by Steele as a term project for a course on VLSI given at MIT by Lynn Conway in Fall 1978. The prototype processor has a small but complete expression evaluator, and an incomplete storage manager (everything but the garbage collector). We plan to design and fabricate by the end of 1979 a full-scale VLSI processor having a complete garbage collector, perhaps more built-in primitive operations, and a more complex storage representation (involving "CDR-coding" [Hansen 1969] [Greenblatt 1974]) for increased bit-efficiency and speed.

A final philosophical thought: it may be worth considering kinds of "stuff" other than vectors and linked records to use for representing data. For example, in LISP we generally organize the records only into trees rather than general graphs. Other storage organizations should also be explored. The crucial idea, however, is that the instruction set should then be fit into the new storage structure in some natural and interesting way, thereby representing programs in terms of the data structures. Continuing the one example, we might look for an evaluation mechanism on general graphs rather than on trees, or on whatever other storage structure we choose. Finally, the instruction set, besides being represented in terms of the data structures, must include means for manipulating those structures. Just as the usual computer has ADD and AND; just as the LISP architecture presented here must supply CAR, CDR, and CONS; so a graph architecture must provide graph manipulation primitives, etc. Following this paradigm we may discover yet other interesting architectures and interpretation mechanisms.

Acknowledgements

We are very grateful to Lynn Conway for coming to MIT, teaching the techniques for NMOS design, and providing an opportunity for us to try our ideas as part of the course project chip. The text used for the course was written by Carver Mead and Lynn Conway [Mead 1978]. Additional material was written by Bob Hon and Carlo Sequin [Hon 1978]. It should be mentioned that the course enabled a large number of students to try interesting and imaginative LSI designs as part of the project chip. This paper describes only one project of the set, but many of these student projects may have useful application in the future.

Paul Penfield and Jon Allen made all this possible by organizing the LSI design project at MIT and arranging for Charles Botchek and Lynn Conway to teach.

Charles Botchek provided our first introduction to the subject and started our wheels spinning.

The course and project chip were executed with the cooperation, generosity, and active help of the Xerox Palo Alto Research Center [Xerox PARC] (which provided software and design support), Micromask (which generated the masks), and Hewlett-Packard (which fabricated the wafers at their Deer Creek facility).

Dick Lyon and Alan Bell of Xerox PARC performed plots of the projects and assembled the projects into the final mask specifications. They were of particular direct aid to Steele in debugging his project.

Glen Miranker and William Henke maintained the plotting software used at MIT to produce intermediate plots of student projects during the design cycle, and were helpful in making modifications to the software to accommodate this project.

Dmitri Antoniadis of MIT packaged and bonded the chips in 40-pin DIPs. Prof. Antoniadis was also a source of good advice concerning device physics and fabrication.

Peter Deutsch and Fernando Corbato were kind enough to hand-carry project plots from California to Boston to help meet the project deadline.

Tom Knight and Jack Holloway provided useful suggestions and sound engineering advice, as usual. (In particular, Knight helped Steele to design a smaller pad to reduce the area of the project, and Holloway suggested the probe multiplexor technique for testing internal signals.)

Peter Deutsch suggested the first subtitle of this paper, for which we gleefully thank him.

Guy Steele's graduate studies during 1978-1979 are supported by a Fannie and John Hertz Fellowship. In the spring of 1978 they were supported by a National Science Foundation Graduate Fellowship.

A condensed version of this report, less the appendix, appeared as [Steele 1979].

APPENDIX

Prototype LISP Processor Technical Specifications

The November 1978 prototype LISP processor bears the working name SIMPLE (Small Integrated Micro-Processor for Lisp Expressions). Here we present complete technical specifications, schematic circuit diagrams, and microcode. Bear in mind that this is only a prototype intended to test the ideas involved, and does not constitute a complete working processor. It is expected, however, that if it works at all, it should be able to execute some small but interesting complete LISP programs.

External Pin Specifications

SIMPLE is expected to be connected to a memory system providing 256 words of 11 bits. Each word is divided into three type bits T0-T2 and eight address bits A0-A7. SIMPLE communicates with the outside world via thirty-three pins:

(11) IT0-IT2, IA0-IA7 (input) [Input Type, Input Address]
Input data from the memory system.

(11) OT0-OT2, OA0-OA7 (output) [Output Type, Output Address]
Output data to the memory system. Addresses and write data are multiplexed on OA0-OA7 according to the ADR and WRITE pins. (The output pads actually contain this information only if the probe controls are zero — see below.)

(2) ADR, WRITE (output)
When ADR is high, the outputs OA0-OA7 contain an address for the memory system; in this case OT0-OT2 are irrelevant. When WRITE is high, the outputs OT0-OT2, OA0-OA7 contain write data for the memory system. In either case, the outputs are maintained for a short time after the control signal goes from high to low to permit proper latching of the data. When both lines are low, the memory system is expected to be presenting to the input pins IT0-IT2, IA0-IA7 the memory data for the address last latched. SIMPLE never raises both ADR and WRITE simultaneously. There is no handshake procedure; the memory is assumed to be able to respond within the clock cycle time used, or to be able to use the FREEZE signal if necessary.

(3) PC0-PC2 (input) [Probe Control]
These signals are for testing only, and are normally tied to ground. The output pins OT0-OT2, OA0-OA7, and ADR normally contain addresses or write data and the ADR signal for the memory system. If the probe controls PC0-PC2 are non-zero, then various signals internal to the chip are gated onto these output pins instead.

(1) STUFF (input)
This signal is for testing only, and is normally tied to ground. If this signal is high, the state bits which serve as the "micro-PC's" for the internal controllers are forcibly loaded from the input pins IT0-IT2, IA0-IA2. The same six bits are used to load both micro-PC's. In this way the controllers can be forced into any given state, and the resulting output signals probed. (This facility is also used to initialize the chip; for this purpose the input pins IT0-IT2, IA0-IA2 should all be zero.)

(1) FREEZE (input)
This signal is for testing only, and is normally tied to ground. If this signal is high, the controllers recirculate in the same state instead of advancing. This signal defers to STUFF.

(2) PHI1, PHI2 (input)
Two-phase non-overlapping clock signals.

(2) VDD, GND (input)
Power supply lines.

(33) pins total.

Instruction Set

The "instruction set" processed by SIMPLE is actually a modified version of SCHEME, a dialect of LISP. The tree-like expressions are essentially LISP S-expressions which constitute a slightly "compiled" version of the usual LISP code. The main effect of this compilation is to pre-calculate the positions of variables in the environment so that variable references do not require search. This in turn simplifies the structure of the environment and of procedures.

When the chip is initialized, it takes a given expression (how it is given is described below), and uses a null environment to evaluate it. If that evaluation ever terminates, the result of the evaluation is stored in memory and the chip halts, with the evaluator looping in a dead state. The chip also halts if it runs out of memory (i.e. after consing the last of the 256 words), with the storage manager looping in a dead state.

When the chip is asked to evaluate an expression, it examines the 3-bit type field. This provides eight "op codes":

0 = constant list	4 = procedure
1 = constant symbol	5 = conditional (if-then-else)
2 = variable reference	6 = procedure call
3 = constant closure	7 = quoted constant

The address part of the word has different purposes depending on the type. For type 2, it is the negative (two's complement) of the position in the

environment of the variable to be referenced, with the first element of the environment being number 1 (hence referenced as -1). For types 0, 1, 3, 5, and 6, the address points to the first of two consecutive words; the first is the cdr, and the second the car. For types 4 and 7, the address points to a single word (a record containing a single pointer), referred to as the "cdr" for compatibility with the previous case.

The evaluation of a type 0 (list), 1 (symbol), or 3 (closure) object simply results in that object; such objects are "self-evaluating". (Notice that symbols are not the same as variables here; this usage has been "compiled out". The only purpose for symbols here is that they are atomic, as opposed to lists, which are not.)

The evaluation of type 7 (quote) returns the cdr of the object. In this way any object whatsoever, not just a list, symbol, or closure, can be included as a constant datum in a program.

The evaluation of type 2 (variable reference) chains (CDRs) down the environment one less than the negative of the number in the address part of the expression pointer. It then takes the car, and returns the value so found.

The evaluation of type 4 (procedure) results in a pointer to a newly-allocated word pair. This pointer has type 3 (closure). The car of the pair contains the cdr of the procedure; this is the code body of the procedure. The cdr of the pair contains the current environment (the environment within which the procedure object is being evaluated). In this way the code and the environment are bound up together (as a closuree, or "FUNARG") for later application.

A conditional (type 5) points to a two-word cell, the cdr of which points to another two-word cell. The car of the conditional object is a predicate expression (IF), the cadr is a consequent expression (THEN), and the cddr is an alternative expression (ELSE). The predicate expression is evaluated first; depending on whether the result is non-NIL or NIL, then the consequent or alternative is evaluated, throwing away the other one, to produce the value of the conditional.

A procedure call (type 6) is the most complicated of the lot. It is a list of indefinite length, chained together by cdr pointers. Each cdr pointer except the last MUST have type 0 (list). The last cdr pointer should have a zero address and a NON-zero type. This last type specifies the operation to be performed. In CDRing down the list, SIMPLE evaluates each of the expressions in the car, saving the resulting values. These values are available as arguments to the operation to be performed. The operations available are:

0 = <more arguments>	4 = ATOM
1 = CAR	5 = PROGN
2 = CDR	6 = LIST
3 = CONS	7 = FUNCALL

For operations CAR, CDR, and ATOM there should be one argument; for CONS, two. No checking is performed for this. For PROGN, LIST, and FUNCALL there may be any non-zero number of arguments.

CAR, CDR, CONS, ATOM, and LIST are the standard LISP primitive operations. LIST is actually a REVERSE-LIST, because it produces a list of the arguments in reverse order; this matters only if the calculations of the arguments have side effects which could interfere with each other. PROGN is a standard LISP primitive which evaluates any number of arguments and returns only the last one. This is useful only when side effects are used. It was included in the prototype primarily to replace EQ when it was removed from the design, because PROGN was fortunately so trivial that it required no extra microcode (it shares a word with the POPJ code).

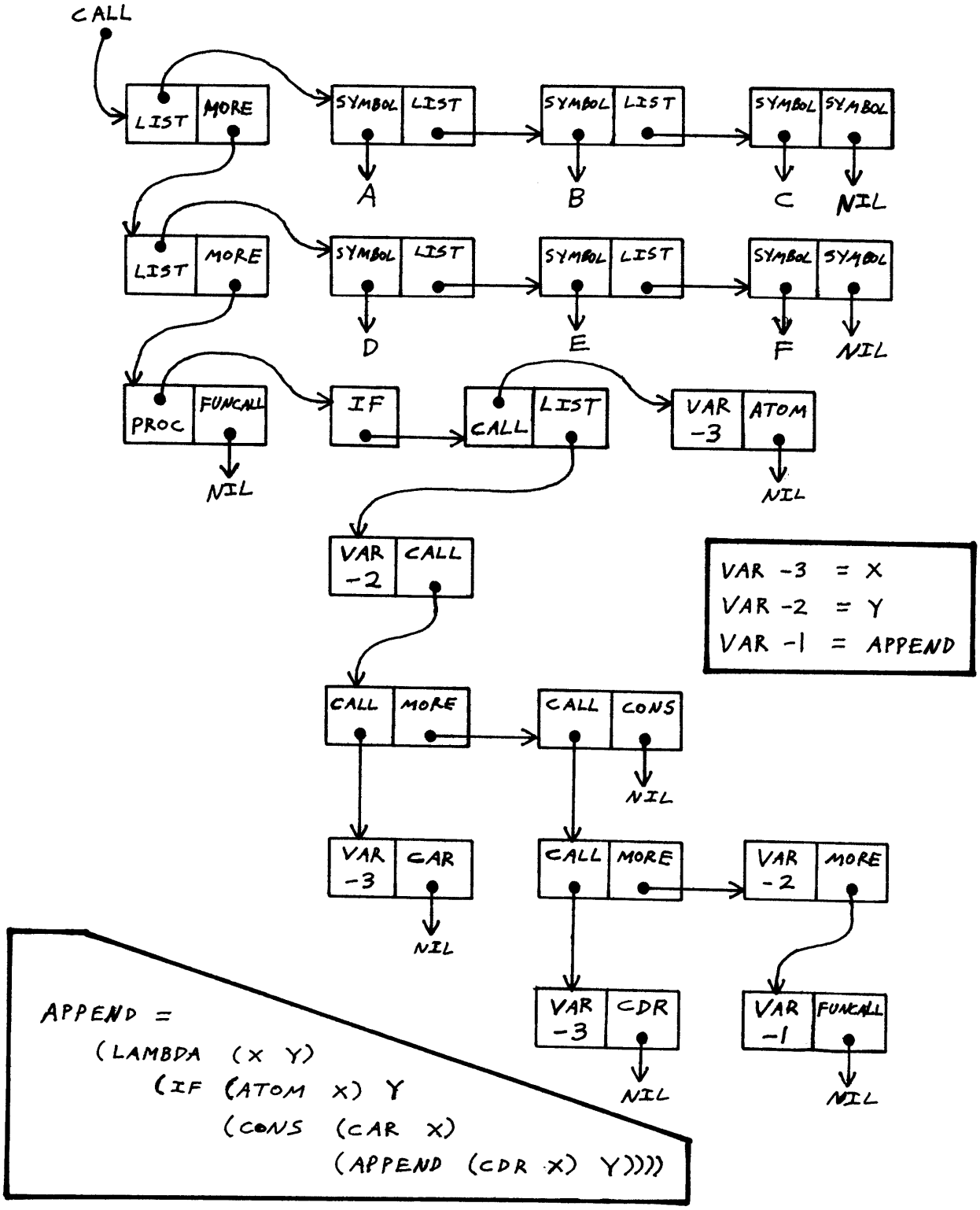
FUNCALL is the operation which calls a user procedure (actually a closure). The last argument (not the first!) must be a closure (this is not checked for!), which is applied to the preceding arguments. (No checking is performed to see whether the correct number of arguments has been supplied!) The body (car) of the closure is evaluated in an environment produced by tacking (NCONCing) all the arguments onto the front of the environment (cdr) of the closure. In this way "lexical scoping" is achieved as in SCHEME or ALGOL. (Because successive sets of variables are tacked together using NCONC rather than being consed onto a display, the environment in the prototype processor takes the form of a simple list of values rather than a list of buckets of values. This is done so that a variable reference can be simply "n back" rather than "n back and j over".) Notice that the closure itself is added to the environment along with the other arguments. In this way the procedure can refer to itself recursively.

As an example, an expression calling for the evaluation of the expression

```
(APPEND '(A B C) '(D E F))
```

including the definition of APPEND itself, is shown in an accompanying diagram.

SIMPLE EXPRESSION FOR (APPEND '(A B C) '(D E F))



Initialization

The chip is initialized by holding the input pins IT0-IT2, IA0-IA2 low while raising STUFF, and stepping the clock for a few cycles. Then STUFF should be lowered. Whenever FREEZE is also low during a clock cycle, the chip will run.

The initial contents of memory should be as follows:

Location	Contents	Remarks
0	0|000	Cdr part of NIL (points to NIL)
1	0|000	Car part of NIL (points to NIL)
2	x|xxx	Cdr part of T
3	x|xxx	Car part of T
4	0|nnn	Beginning of free storage
5	exp	Expression to be evaluated
6	x|xxx	Reserved for result of evaluation

The notation "t|a" means a pointer with type "t" and address "aaa".

NIL must be at location 0, and T at location 2. These are used as returned values by the built-in predicate ATOM. Notice that NIL is considered to be a list. Location 5 should contain the expression to be evaluated. Additional cells in the expression may occupy other memory words above location 6 as appropriate. Location 4 should point at the last word used in the initial expression. Words after the one pointed to by location 4 are assumed to be free for allocation by the "garbage collector". (Actually, SIMPLE has no garbage collector. When storage has been once allocated, it drops dead.) The actual free storage pointer is not maintained in location 4; location 4 is in general not changed. It is only used to initialize the free storage pointer internal to the chip. Location 6 is reserved for the result of the evaluation; if evaluation of the expression in location 5 ever terminates, the resulting pointer is written into location 6, and the processor halts.

Register/Logic Level Description

Internally SIMPLE is organized into two parts. One part, EVAL, is concerned with the evaluation algorithm. The other part is concerned with storage management, and is called GC. One way to think about this division is that GC supplies a set of "subroutines" which are used by EVAL to deal with the memory system. In this way EVAL can deal with the details of the evaluation of the code in terms of a linked-list memory system. GC implements this memory system in terms of the usual "linear-array" memory system.

Each of the two parts, EVAL and GC, is itself divided into two parts: registers and controller. The registers provide storage for type/pointer words, and are connected by a common bus in each part. Each controller is a finite-state machine implemented as a PLA, plus some random logic. Each PLA is organized as a micro-code ROM, addressed by a "micro-PC" and yielding a set

of control signals, including register controls and a new micro-PC indicating the next state.

EVAL has five registers, called V (Value), X (Expression), N (Environment), L (List of arguments), and C (Control stack). Each can hold an eleven-bit word of three type bits T0-T2 and eight address bits A0-A7. (These bits are therefore referred to as VT0, VT1, XA0, NA6, etc.) They are connected together by a common bus called the E bus. Any of these registers can be loaded from the bus (according to the signals LDV, LDX, LDN, LDL, LDC) or read from the bus (according to the signals RDV, RDX, RDN, RDL, RDC). In addition, register X has incrementation logic associated with it. The signal RDX+ causes the contents of X, plus 1, to be read onto the E bus. (This incrementer constitutes the "arithmetic" portion of EVAL's "ALU".) The type bits do not participate in the incrementation, only the address bits. (If the address bits of X are all ones, then incrementing X reads an all-zero address part, but does not read an incremented type.)

GC has three-and-a-half registers, called P, Q, R, and S. They are connected by a common bus called the G bus. They can be loaded from the G bus (according to the signals LDP, LDQ, LDR, LDS) and read back onto the G bus (according to the signals RDP, RDQ, RDR). Register S cannot be read back onto the G bus. The output pads O are ordinarily driven from the contents of register S; thus register S serves as the latch for the output pads. Register P, like register X, has associated incrementation logic; the contents of P, plus 1, can be read onto the G bus according to the signal RDP+. Register P is also special (and unlike X) in that its type bits are always zero. Reading register P always forces zeroes onto GT0-GT2, and loading P always discards the type bits.

Here is a complete list of the signals produced by the two controllers of EVAL and GC. (Unfortunately, the hyphen character "-" is used both as a break character within a signal name and to indicate a series of signal names ending in consecutive digits. The two cases can be distinguished by whether or not a digit appears before the hyphen.)

EVAL	Signal name	Remarks
6	ENSTATE0-ENSTATE5	Next state
6	EOSTATE0-EOSTATE5	Old (current) state
2	ET-DISP, EAZ-ETZ-DISP	Dispatch selectors
5	LDV, LDX, LDN, LDL, LDC	Register load controls
6	RDV, RDX, RDX+, RDN, RDL, RDC	Register read controls
4	GCOP0-GCOP3	GC OPeration request
3	LIT0-LIT2	Three-bit literal value
2	EA-LIT, ET-LIT	Literal read controls
34 signals total		

GC	Signal name	Remarks
6	GNSTATE0-GNSTATE5	Next state
6	GOSTATE0-GOSTATE5	Old (current) state
2	GCOP-DISP, GA-ZERO-DISP	Dispatch selectors
4	LDP, LDQ, LDR, LDS	Register load controls
4	RDP, RDP+, RDQ, RDR	Register read controls
3	CONN-I, CONN-EA, CONN-ET	Internal bus connections
1	STEP-EVAL	Advance EVAL controller
2	ADR, WRITE	Memory controls
28	signals total	

Actually, some signals are required only in their inverted (active low) state; rather than have additional random inverters, the PLA programming is inverted so that the signal emerges from the PLA in the active low sense. Thus these actual signals emerge from the PLA's: -ENSTATE3, -ENSTATE4, -ENSTATE5, -GNSTATE2, -GNSTATE3, -GNSTATE4, -GNSTATE5, and -STEP-EVAL. (This is a third use of the hyphen in signal names: a leading hyphen means an "active low" or "inverted" signal.)

The two controllers are clocked in parallel. At each step a new state is latched for each controller. This new state can come from one of three sources: the old state (OSTATE), the next state (NSTATE), or the input pads (I). It comes from I if STUFF is high. It comes from OSTATE if STUFF is low and FREEZE is high. Otherwise, it comes from NSTATE.

The preceding contains two inaccuracies. One is that the EVAL controller uses not FREEZE, but rather FREEZE-EVAL, which is a signal computed as the OR of FREEZE and -STEP-EVAL. Thus the EVAL controller actually uses OSTATE if either FREEZE is high OR if STEP-EVAL is not asserted by the GC controller (i.e. if -STEP-EVAL is asserted); in this way GC can control the advance of EVAL.

The other inaccuracy is that each controller has two dispatch control lines which can cause NSTATE to be modified before entering the FREEZE/STUFF selector. If either one is asserted (it should never occur that both are asserted), then certain control signals are OR'd with the low bits of NSTATE to produce the new state (if so selected by STUFF and FREEZE). This facility provides for conditional dispatching in the "microcode" for the controllers: if the microword provides zero-bits in the low bits of NSTATE, then the next microinstruction is selected according to the control signals specified. Some of the control signals may be effectively masked by specifying one-bits in NSTATE corresponding to those signals. The control signals which may be dispatched on, and their corresponding dispatch selectors, are:

Controller	Selector	Signals selected to OR with NSTATE	
EVAL	ET-DISP	ET0-ET2	(8-way dispatch)
EVAL	EAZ-ETZ-DISP	EA-ZERO, ET-ZERO	(4-way dispatch)
GC	GCOP-DISP	GCOP0-GCOP3	(16-way dispatch)
GC	GA-ZERO-DISP	GA-ZERO	(2-way dispatch)

EA-ZERO is asserted if the address bits on the E bus (EA0-EA7) are all

zero. ET-ZERO similarly is asserted if the type bits on the E bus (ET0-ET2) are all zero. (There was originally to be a third signal in this group called EQV, asserted if the address on the E bus was equal to the address in the V register. This was eliminated late in the design to conserve area. However, it left an after-effect: the 4-way dispatch on EA-ZERO and ET-ZERO affects bits 3 and 4 of NSTATE, not bits 4 and 5, breaking the intended general design rule that dispatching affects the low n bits of NSTATE.) GA-ZERO is asserted if all the address bits on the G bus are zero.

The EVAL controller can read certain constants onto the E bus, rather than reading the contents of a register. This is done by asserting EA-LIT or ET-LIT. These respectively read the address (0,0,0,0,LIT0,LIT1,LIT2,0) onto EA0-EA7, and LIT0-LIT2 onto ET0-ET2. In this way any constant type can be specified, and a small range of even addresses (0, 2, 4, 6, 10, 12, 14, 16 octal — as it turned out only the first four are used).

The GC controller can connect the busses in certain ways. The signal CONN-I connects the input pads I to the G bus; the input pads thus serve as a "read-only register" to the G bus in much the same way that the S register (to which the output pads are normally tied) serves as a "write-only register". The signal CONN-EA connects the address bits of the E and G busses together; similarly CONN-ET independently connects the type bits of the E and G busses. When they are so tied, only one of EVAL and GC should be attempting to read something onto its bus; the other may then load one or more registers from this source. Thus this bus connection facility provides for bidirectional communication between the two sets of registers; only one direction may be used per EVAL step, however. The reason for being able to specify the address and type connections separately is that frequently GC will supply an address to the E bus from the G bus, while simultaneously EVAL will supply type bits to the E bus using ET-LIT.

All the signals emerging from the PLA's pass through a structure called the probe multiplexor. Under the control of the external signals PC0-PC2 these signals can be gated to the output pads OT0-OT2, OA0-OA7, and ADR for external testing purposes.

The signals PC0-PC2 are decoded from binary to 1-of-7, producing the mutually exclusive signals PCA, PCB, PCC, PCD, PCE, PCF, and -PROBE. -PROBE is normally asserted (PC0-PC2 = 000), which allows the output pads to function normally. The other signals gate internal signals to the pads as follows:

Pad	PCA=001	PCB=010	PCC=011	PCD=101	PCE=110	PCF=111	Probe bus
ADR	GOSTATE5	-GNSTATE5	LDR	RDQ	---	---	GP6
OT0	GOSTATE4	-GNSTATE4	LDQ	LDP	---	---	GP5
OT1	GOSTATE3	-GNSTATE3	RDP	RDP+	---	---	GP4
OT2	GOSTATE2	-GNSTATE2	CONN-ET	CONN-EA	---	---	GP3
OA0	GOSTATE1	GNSTATE1	CONN-I	-STEP-EVAL	ADR	WRITE	GP2
OA1	GOSTATE0	GNSTATE0	GA-ZERO-DISP	GCOP-DISP	LDS	RDR	GP1
OA2	EOSTATE5	-ENSTATE5	LDC	RDL	---	---	EP6
OA3	EOSTATE4	-ENSTATE4	LDL	RDN	GCOP3	GCOP2	EP5
OA4	EOSTATE3	-ENSTATE3	LDN	RDV	GCOP1	GCOP0	EP4
OA5	EOSTATE2	ENSTATE2	LDV	RDX+	LIT2	LIT1	EP3
OA6	EOSTATE1	ENSTATE1	RDX	LDX	LIT0	ET-LIT	EP2
OA7	EOSTATE0	ENSTATE0	EAZ-ETZ-DISP	ET-DISP	EA-LIT	RDC	EP1

When PC0-PC2 = 100, then the output pads are not driven from any source. Also, entries "---" in the table indicate an output which is not driven. (It was originally intended that the numbering of the probed OSTATE and NSTATE bits should follow the ordering of the OT and OA bits. In the last-minute haste of the design effort, the probe lines were accidentally hooked up backwards. This is "only" a matter of elegance — the probe multiplexor will still do its job.)

Signals which must control more than a few gates (e.g. register controls) are actually used to drive superbuffers. The signal is first NANDed with PHI1 and then used to control an inverting superbuffer. The superbuffer allows the long control lines passing through the register array to be driven faster; while this may not be necessary in the prototype, it will probably be a good idea in a larger version with 40-bit registers. The gating by PHI1 was intended to permit the pre-charging of the bus lines during PHI2 if desired; but later it was decided to omit pre-charging from the prototype.

SIMPLE Microcode

Here we present the microcode for the two controllers. Remember, in reading it, that the two machines are clocked in parallel, but EVAL advances only when stepped by the STEP-EVAL signal from GC (rather, when not inhibited by -STEP-EVAL).

The basic protocol is that GC is in a loop, constantly performing GCOP-DISP according to the operation GCOP requested by the current EVAL microcode word. When GC has performed the requested operation, it steps EVAL and then returns to its loop.

There are basically five kinds of operation EVAL can request: NOP, CAR/CDR, CONS/XCONS, RPLACD, and load/store Q.

NOP means "do nothing"; it is used when EVAL just wants to shuffle things on the E bus. GC takes two cycles to perform a NOP: one to dispatch on the GCOP, and one to do STEP-EVAL. (The synchrony of the two controllers prevents these two operations from being simultaneous. There are ways to

avoid this problem, which should be used in a full-scale version. In the prototype we wanted to avoid complicated timing problems.) Thus EVAL proceeds at at most half the speed of GC.

CAR/CDR operations request GC to perform a CAR or CDR operation on memory. Many variants of this are provided to optimize data shuffling between the E and G busses. A general convention is that after a CAR/CDR operation the result of the operation is left in Q, and the original operand is left in R. The basic CAR and CDR operations take their operand from the E bus; the result is left in Q. The CDRQ, CARQ, CARR variants take their operands from the indicated register (Q or R), and return the result to the E bus (and also put it in Q). The CDRRX, CARRX, CDRQX variants take their operands from the indicated register, put the result in Q, and make no attempt to use the E bus. In this way EVAL can request GC to do something useful while simultaneously doing something else with the E bus.

The CONS operation accepts a car pointer from the E bus, takes the contents of Q to be the cdr pointer, and then allocates a new two-word cell containing that car and cdr. A pointer to the result, with type 0 (list), is left in Q.

The RPLACDR operation alters the cdr of the cell pointed to by R to be the pointer passed from the E bus.

The load/store Q operations allow EVAL to access the Q register. RDQ requests GC to pass the contents of Q back to the E bus. RDQA asks for just the address bits; EVAL typically supplies the type bits from a literal using ET-LIT. LDQ stuffs the E bus contents into Q. (Note that the names RDQ and LDQ are meant to be suggestive of the standard register control signals; but in this context they are <u>not</u> such signals, but rather particular values for the GCOP field which request GC to apply its register control signals of the same name.)

RDQCDRRX is a combination of RDQ and CDRRX useful in odd circumstances; that is, a CDRRX is performed, and then the <u>old</u> contents of Q are passed back to the E bus. This operation breaks the Q-and-R convention: Q is indeed set to the result of the CDRRX, but R is used to contain the old value of Q, and not the operand of the CDRRX.

The reason there are such complex variants on the CAR/CDR operations has to do, strangely enough, with geometrical layout constraints. The original design for the prototype had only three GCOP control lines, and thus eight possible requests (NOP, CAR, CDR, CONS, XCONS, RDQ, RDQA, and one unused spare). With this design EVAL required about 80 words of microcode and GC only about 30 words. This imbalance would have made the EVAL PLA much too large, and the project would have had an awkward shape. Thus it was decided to look for common operation sequences in EVAL and make them into single GC operations, thus shrinking EVAL and expanding GC by adding extra "subroutines". Making EVAL just a little larger than GC allowed a better balance and an overall rectangular shape.

The initialization protocol is a bit peculiar. EVAL begins at state INIT, and GC at state GC; these states are both state zero. The first thing EVAL does is put a pointer in register C with type 2 (which in this context is a "return address" code). This 2 is also gated into the address bits, thus providing a 4 there. The 4 is irrelevant to the value placed in C, but is needed by GC, which will connect the E and G busses on that cycle. This 4 is used to fetch the initial free storage pointer, and is also placed in Q so that EVAL can later request CARQ to get the expression to be evaluated from location 5. EVAL also initializes register N to contain NIL (a zero type/zero address pointer). This is all rather kludgy, but the multiple uses of certain magic numbers allows the initialization to occupy only two microwords in each PLA.

Multiple magic numbers are also used in the implementation of ATOM. The symbol T must be at location 2 in the memory because the same literal 1 is used to generate both the type (1 = SYMBOL) and the address (twice 1 = 2) when generating a result of T. Similarly, a 0 type and 0 address is used for NIL. This unfortunately results in (ATOM (ATOM <any list>)) = NIL, despite the fact that in LISP NIL is defined to be an atom. This is a defect in the design of the prototype. In preceding diagrams we have shown NIL as a symbol rather than as a list. Ideally it should be treated as a special object which is both a list and an atom, depending on context.

Register C always contains a control stack for the recursive evaluation. Quantities which need to be saved are consed onto this stack. The cdr pointers which chain the stack cells together are usually of type 0 (list), but sometimes have other type codes which encode "return addresses" within the EVAL microcode. At the state labelled POPJ, the EVAL controller dispatches on the type code of what is in C. This specifies what to do next on a return from a "recursive call" to EVAL. The type 2 ("TOPLEVEL") pointer which is initially placed in C specifies that EVAL should "drop dead", as the expression evaluation has been completed, after storing the result of the evaluation into memory location 6.

The microcode is written in a "LISPy" form, as a list of four things. The first thing is the symbol *UCODE, which indicates that this is microcode. The second thing is the name of the controller (EVAL or GC) for which this is the microcode.

The third thing is a list beginning with the symbol FIELDS. This is a declaration of the names and sizes of all fields of the microword. If a declaration is a symbol (e.g. LDV), then it is the name of a single bit. If it is a list, then the car of the list is the name of the field. The cadr of the list may be a number, which is the width of the field in bits; this is optionally followed by NSTATE or OSTATE (which indicate special treatment for those fields by the microassembler). If the cadr is not a number, then some number of items will follow the field name. The number of such items must be a power of two, and these items are names (or lists of names) for the possible binary values of the field. For example, in the EVAL microcode, the declaration of LIT indicates that it has 8 values (hence is $\log_2 3 = 8$ bits wide). The names LIST, EV2, and NIL each indicate the value 0 in the context

of the LIT field; CLOSURE and RESULT each indicate the value 3 in that context; IF indicates the value 5; and so on.

The fourth thing is a list beginning with the symbol CODE; following this symbol is the microcode proper, which is a series of items. A symbol is a tag denoting the address of the following microinstruction (example: INIT or POPJ1). A list is a set of signals forming a single microcode word. In general, a signal like LDC is asserted iff its name LDC is present in the microinstruction. Multi-bit fields are specified by a list of the field name and the value (example: (LIT IF2) specifies the value 1 for the LIT field, meaning LIT0=0, LIT1=0, LIT2=1).

Fields tagged in the FIELDS declaration as NSTATE and OSTATE fields are handled specially by the microassembler. OSTATE is just the address of the current state. NSTATE may be explicitly specified by (GOTO <tag>), or implicitly specified by the rule that omitting a (GOTO <tag>) means that the next state is the textually following microinstruction. (This does not imply that the microinstructions have consecutive "addresses", but only that one has the address of another in its NSTATE field.) The microassembler fills in NSTATE and OSTATE fields automatically after it has assigned addresses to all the instructions.

The address of a microinstruction may be constrained by an "=" specifier (this idea is borrowed from the microassembler used by DEC for KL10 microcode, largely because the dispatching technique was borrowed from that used by the KL10, which Steele has had some experience microprogramming). This address-alignment facility is used primarily for aligning dispatch tables. A specification such as "(= % 1 * *)" means that the addresses of the next four (2 to the power <number of *'s>) microinstructions are constrained to end in 100, 101, 110, 111 in that order. Thus an explicit 1 (or 0, but this is never used in practice) forces an address bit to be 1 (0), while a * indicates that either may be used, and all combinations are used in lexicographic order for textually successive instructions. After the 2-to-the-power-<number-of-*'s> instructions, one writes "(= %)" to mark the end of the table; this is used for error-checking by the microassembler. The character "%" may be any single character; it is used to make sure the two "=" specifications match, and is also used in the microcode assembly listing to show where the dispatch table was placed. As an example, the specification (= + 1 * 1) at ATOM1 causes the next instruction to have an address ending in binary 101, and the one textually following that at an address ending in 111.

If a number follows the "=" rather than a one-character symbol (for example, "(= 0)"), then the address of the next microinstruction is forced to be that number. In this case no matching "=" construct follows the instruction whose location was forced. In the listings that follow, this is used to force the first instruction of each controller to be at location 0.

```
(*UCODE EVAL
        (FIELDS (ENSTATE 6 NSTATE)
                (EOSTATE 6 OSTATE)
                (LIT (LIST EV2 NIL)
                     (SYMBOL IF2 T)
                     (VARIABLE TOPLEVEL)
                     (CLOSURE RESULT)
                     (PROCEDURE)
                     (IF)
                     (COMBINATION)
                     (QUOTE))
                (GCOP NOP CDR CAR CDRQ CARQ CARR CDRRX CARRX CDRQX
                      CONS XCONS RPLACDR LDQ RDQ RDQA RDQCDRRX)
                LDV LDX LDN LDL LDC RDV RDX RDX+ RDN RDL RDC
                ET-DISP EAZ-ETZ-EQV-DISP EA-LIT ET-LIT)
        (CODE INIT
         (= 0)
              (ET-LIT EA-LIT (LIT TOPLEVEL) LDC)        ;TOPLEVEL = 2!!!
              (EA-LIT ET-LIT (LIT NIL) LDN (GOTO INIT1))
         EVAL
         (= @ * * *)
              (RDX LDV (GOTO POPJ))                              ;LIST
              (RDX LDV (GOTO POPJ))                              ;SYMBOL
              ((GCOP LDQ) RDN (GOTO VAR1))                       ;VARIABLE
              (RDX LDV (GOTO POPJ))                              ;CLOSURE
              ((GCOP CDR) RDX (GOTO PROC1))                      ;PROCEDURE
              ((GCOP CDR) RDX (GOTO IF1))                        ;IF
              (EA-LIT ET-LIT (LIT NIL) LDL (GOTO EVARGS))        ;COMBINATION
              ((GCOP CDR) RDX (GOTO STOREV))                     ;QUOTE
         (= @)
         VAR1
              (RDX+ LDV EAZ-ETZ-EQV-DISP)
         (= # * 1 1)                                    ;EA ZERO DISP
              ((GCOP CDRQX) RDV LDX (GOTO VAR1))        ;NONZERO
              ((GCOP CARQ) LDV (GOTO POPJ))             ;ZERO
         (= #)
         POPJ1
         (= $ 1 * *)                                    ;ONLY ET1-ET2 RELEVANT
              ((GCOP CONS) RDV (GOTO EV3))              ;EV2
              ((GCOP RDQCDRRX) LDN (GOTO IF3))          ;IF2
              ((GCOP CDR) EA-LIT ET-LIT
                          (LIT RESULT) (GOTO STORE6))   ;TOP LEVEL (RESULT = LOCATION 6)
         DEAD ((GOTO DEAD))                             ;??? SHOULDN'T HAPPEN
         (= $)
         IF1
              ((GCOP XCONS) RDC)
              ((GCOP CONS) RDN)
              ((GCOP RDQA) ET-LIT (LIT IF2) LDC (GOTO IF1A))
         IF3
              ((GCOP CDRQ) LDC)
              ((GCOP CARRX) RDV EAZ-ETZ-EQV-DISP (GOTO IF4))
```

```
IF4
(= % * 1 1)                                    ;EA ZERO DISP
INIT1
     ((GCOP CARQ) LDX ET-DISP (GOTO EVAL))     ;NONZERO
     ((GCOP CDRQ) LDX ET-DISP (GOTO EVAL))     ;ZERO
(= %)
EVARGS
(= & * * *)
     ((GCOP CDR) RDX (GOTO EV1))               ;MORE
     ((GCOP CAR) RDV (GOTO STOREV))            ;CAR
     ((GCOP CDR) RDV (GOTO STOREV))            ;CDR
     ((GCOP CDR) RDL (GOTO CONS1))             ;CONS
     (RDV EAZ-ETZ-EQV-DISP (GOTO ATOM1))       ;ATOM
POPJ ((GCOP CAR) RDC ET-DISP (GOTO POPJ1))     ;PROGN
     (RDL LDV (GOTO POPJ))                     ;LIST
     ((GCOP CDR) RDV (GOTO FUN1))              ;FUNCALL
(= &)
CONS1
     ((GCOP CARQ))
     ((GCOP XCONS) RDV)
STOREV
     ((GCOP RDQ) LDV (GOTO POPJ))              ;PUT Q INTO V, POPJ
ATOM1
(= + 1 * 1)                                    ;ET ZERO DISP
     (EA-LIT ET-LIT (LIT T) LDV (GOTO POPJ))   ;ATOM
     (EA-LIT ET-LIT (LIT NIL) LDV (GOTO POPJ)) ;LIST
(= +)
FUN1
     ((GCOP RDQ) LDN)
     ((GCOP CDR) RDL EAZ-ETZ-EQV-DISP)
FUN2
(= \ * 1 1)                                              ;EA ZERO DISP
     ((GCOP CDRQ) EAZ-ETZ-EQV-DISP (GOTO FUN2))          ;NONZERO
     ((GCOP RPLACDR) RDN (GOTO FUN3))                    ;ZERO
(= \)
FUN3
     (RDL LDN)
     ((GCOP CAR) RDV (GOTO FUN4))
EV1
     ((GCOP XCONS) RDC)
     ((GCOP CONS) RDN)
     ((GCOP CONS) RDL)
     ((GCOP RDQA) ET-LIT (LIT EV2) LDC)
IF1A
     ((GCOP CAR) RDX)
FUN4
     ((GCOP RDQ) LDX ET-DISP (GOTO EVAL))
```

```
        EV3
             ((GCOP RDQ) LDL)
             ((GCOP CDR) RDC)
             ((GCOP CARQ) LDN)
             ((GCOP CDRRX))
             ((GCOP CDRQ) LDC)
             ((GCOP CARR) LDX ET-DISP (GOTO EVARGS))
        PROC1
             ((GCOP XCONS) RDN)
             ((GCOP RDQA) ET-LIT (LIT CLOSURE) LDV (GOTO POPJ))
        STORE6
             ((GCOP RPLACDR) RDV (GOTO DEAD))
))      ;END OF *UCODE EVAL


(*UCODE GC
        (FIELDS (GNSTATE 6 NSTATE)
                (GOSTATE 6 OSTATE)
                STEP-EVAL CONN-EA CONN-ET CONN-I
                LDP LDQ LDR LDS RDP RDP+ RDQ RDR
                ADR WRITE GCOP-DISP GA-ZERO-DISP)
        (CODE GC
           (= 0)
              (CONN-EA LDS LDQ ADR)                    ;EVAL SUPPLIES 4
              (CONN-I LDP STEP-EVAL (GOTO LOOP))       ;READ INITIAL FREE PTR
           LOOP
              (GCOP-DISP)
           (= @ * * * *)
              (STEP-EVAL (GOTO LOOP))                          ;NOP
              (CONN-EA CONN-ET LDS LDR ADR (GOTO CDR1))        ;CDR
              (RDP LDQ (GOTO CAR1))                            ;CAR
              (RDQ LDS ADR (GOTO CDRQ1))                       ;CDRQ
           CARQ0 (RDP LDR (GOTO CARQ1))                        ;CARQ
              (RDR LDQ (GOTO CARQ0))                           ;CARR
              (RDR LDS ADR (GOTO CDR1))                        ;CDRRX
              (RDP LDQ (GOTO CARRX1))                          ;CARRX
              (RDQ LDS LDR ADR (GOTO CDR1))                    ;CDRQX
              (RDP+ LDS LDR ADR GA-ZERO-DISP (GOTO CONS1))     ;CONS
              (RDP+ LDS LDR ADR GA-ZERO-DISP (GOTO XCONS1))    ;XCONS
              (RDR LDS ADR (GOTO RPLACDR1))                    ;RPLACDR
              (CONN-EA CONN-ET LDQ STEP-EVAL (GOTO LOOP))      ;LDQ
              (CONN-EA CONN-ET RDQ STEP-EVAL (GOTO LOOP))      ;RDQ
              (CONN-EA RDQ STEP-EVAL (GOTO LOOP))              ;RDQA
              (RDR LDS ADR (GOTO RDQCDRRX1))                   ;RDQCDRRX
           (= @)
```

```
        CAR1
           (CONN-EA CONN-ET LDP LDR)
        CAR2
           (RDP+ LDS)
           (RDQ LDP)
        CDR1
        CAR3
           (CONN-I LDQ STEP-EVAL (GOTO LOOP))
        CARQ1
           (RDQ LDP)
           (RDP+ LDS)
           (RDR LDP)
        CDRQ1
           (RDQ LDR)
           (CONN-I CONN-EA CONN-ET LDQ STEP-EVAL (GOTO LOOP))
        CARRX1
           (RDR LDP (GOTO CAR2))
        RPLACDR1
           (CONN-EA CONN-ET LDS LDQ)                ;LEAVE GAP BETWEEN ADR AND WRITE
           (WRITE STEP-EVAL (GOTO LOOP))
        XCONS1
        (= $ *)
           (RDR LDP (GOTO XCONS2))
        GCDEAD ((GOTO GCDEAD))                      ;FREE PTR WRAPPED AROUND
        (= $)
        XCONS2
           (RDQ LDR)
           (CONN-EA CONN-ET LDS (GOTO CONS3))
        CONS1
        (= % *)
           (RDR LDP (GOTO CONS2))
           ((GOTO GCDEAD))
        (= %)
        CONS2
           (CONN-EA CONN-ET LDR)
           (RDQ LDS)
        CONS3
           (RDP LDQ WRITE)
           (RDP+ LDS ADR GA-ZERO-DISP)
        (= & *)
           (RDR LDS (GOTO CONS4))               ;LEAVE GAP BETWEEN WRITE AND ADR
           ((GOTO GCDEAD))
        (= &)
        CONS4
           (WRITE RDP+ LDR)
           (RDR LDP STEP-EVAL (GOTO LOOP))
        RDQCDRRX1
           (RDQ LDR)
           (CONN-I LDQ)
           (RDR CONN-EA CONN-ET STEP-EVAL (GOTO LOOP))
))      ;END OF *UCODE GC
```

A remnant of the planned logic for the EQ operation survives in that in the actual EVAL microcode the name "EAZ-ETZ-EQV-DISP" was still used instead of the more correct "EAZ-ETZ-DISP".

It may be noticed that this code has been tightly bummed to share instructions among several different paths of code (for example at IF1A and FUN4). This is no accident. The microassembler looks for assembled microwords which have the same value (except for OSTATE fields) and flags them in the assembly listing precisely so that such instructions may be merged if desired.

The assembly listing actually used to produce the PLA programming for the prototype is reproduced here. The listing is designed to be readable both by people (and so it is arranged in columns) and by LISP (and so it is parenthesized). All numbers in the listing are octal. The listing for each program is a single list, beginning with the symbol UCODE and the name of the program (EVAL or GC). Then is the maximum number of words of microcode memory needed to contain the program; this is actually two to the power <size of the NSTATE field>. After this is a comment specifying the width of the microword.

Next comes the definitions of all fields, as assigned by the microassembler. (These definitions have nothing to do with the order of the signals emerging from the PLA. They simply indicate where in each assembled microword the microassembler has placed the value for each field. The software which produces the PLA programming from the assembly listing permutes the bits in an arbitrarily specified fashion to suit the layout. It also automatically inverts the programming for such bits as -STEP-EVAL. In this way the written microcode can mention these bits in the positive sense rather than the negative (active low) sense.)

Each field definition has the word FIELD, a numeric value with a 1 bit in every bit position of the field and 0 bits elsewhere, and a 4-list. The 4-list contains the name of the field; the type of the field (BIT, NSTATE, OSTATE, or the list of symbolic values from the declaration); the width of the field; and the position of the field in the assembled microword, measured from the right.

If a field is more than one bit wide, then the assembler automatically defines name for each of the bits of the field, by methodically appending decimal numbers to the field name, and numbering the bits of the field from left to right.

Following the field definitions are the assembled instructions, in address order (which in general will not be the same as the textual order of the instructions in the input program). For each instruction is listed the address; the assembled microword value; a single character if the instruction is part of a dispatch table, or a "!" if the instructions location was forced by "(= <number>)", or a blank otherwise; if the instruction had a tag or tags, then that tag or a list of the tags, followed by a colon; the symbolic instruction; and an arrow "=>" followed by the address in the NSTATE field.

Following the assembled instructions is a remark indicating the actual number of assembled microwords. This need not be greater than the last address used (though it is in the actual cases shown here), because the microassembler may leave gaps in the address space to accommodate dispatch tables.

The comments before and after the listings are timestamp information generated by the microassembler.

```
;THIS IS THE ASSEMBLED MICROCODE FOR ((DSK SCHIP) USIMPL /22)
;It is 6 days, 15 hours, and 3 minutes past the new moon.
;The sun is 41*56'59" east of south, 13*5'50" above the horizon.
;That means it is now 8:36 AM on Wednesday, December 6, 1978.

(UCODE EVAL
(100 WORDS)
(REMARK MICROWORDS ARE 42 (OCTAL) BITS WIDE)
(FIELD 100000000000 (ET-LIT BIT 1 41))
(FIELD 040000000000 (EA-LIT BIT 1 40))
(FIELD 020000000000 (EAZ-ETZ-EQV-DISP BIT 1 37))
(FIELD 010000000000 (ET-DISP BIT 1 36))
(FIELD 004000000000 (RDC BIT 1 35))
(FIELD 002000000000 (RDL BIT 1 34))
(FIELD 001000000000 (RDN BIT 1 33))
(FIELD 000400000000 (RDX+ BIT 1 32))
(FIELD 000200000000 (RDX BIT 1 31))
(FIELD 000100000000 (RDV BIT 1 30))
(FIELD 000040000000 (LDC BIT 1 27))
(FIELD 000020000000 (LDL BIT 1 26))
(FIELD 000010000000 (LDN BIT 1 25))
(FIELD 000004000000 (LDX BIT 1 24))
(FIELD 000002000000 (LDV BIT 1 23))
(FIELD 000001700000 (GCOP (NOP CDR CAR CDRQ CARQ CARR CDRRX CARRX CDRQX CONS XCONS RPLACDR
LDQ RDQ RDQA RDQCDRRX) 4 17))
(FIELD 000001000000 (GCOP0 BIT 1 22))
(FIELD 000000400000 (GCOP1 BIT 1 21))
(FIELD 000000200000 (GCOP2 BIT 1 20))
(FIELD 000000100000 (GCOP3 BIT 1 17))
(FIELD 000000070000 (LIT ((LIST EV2 NIL) (SYMBOL IF2 T) (VARIABLE TOPLEVEL) (CLOSURE RESULT)
(PROCEDURE) (IF) (COMBINATION) (QUOTE)) 3 14))
(FIELD 000000040000 (LIT0 BIT 1 16))
(FIELD 000000020000 (LIT1 BIT 1 15))
(FIELD 000000010000 (LIT2 BIT 1 14))
(FIELD 000000007700 (EOSTATE OSTATE 6 6))
(FIELD 000000004000 (EOSTATE0 BIT 1 13))
(FIELD 000000002000 (EOSTATE1 BIT 1 12))
(FIELD 000000001000 (EOSTATE2 BIT 1 11))
(FIELD 000000000400 (EOSTATE3 BIT 1 10))
(FIELD 000000000200 (EOSTATE4 BIT 1 7))
(FIELD 000000000100 (EOSTATE5 BIT 1 6))
```

```
(FIELD 000000000077 (ENSTATE NSTATE 6 0))
(FIELD 000000000040 (ENSTATE0 BIT 1 5))
(FIELD 000000000020 (ENSTATE1 BIT 1 4))
(FIELD 000000000010 (ENSTATE2 BIT 1 3))
(FIELD 000000000004 (ENSTATE3 BIT 1 2))
(FIELD 000000000002 (ENSTATE4 BIT 1 1))
(FIELD 000000000001 (ENSTATE5 BIT 1 0))
(00 140040020001 ! INIT :       (ET-LIT EA-LIT (LIT TOPLEVEL) LDC) => 01)
(01 140010000133                (EA-LIT ET-LIT (LIT NIL) LDN (GOTO INIT1)) => 33)
(02 020402000223   VAR1 :       (RDX+ LDV EAZ-ETZ-EQV-DISP) => 23)
(03 004001200320   IF1 :        ((GCOP XCONS) RDC) => 20)
(04 000101100464 $ POPJ1 :      ((GCOP CONS) RDV (GOTO EV3)) => 64)
(05 000011700522 $              ((GCOP RDQCDRRX) LDN (GOTO IF3)) => 22)
(06 140000130671 $              ((GCOP CDR) EA-LIT ET-LIT (LIT RESULT) (GOTO STORE6)) => 71)
(07 000000000707 $ DEAD :       ((GOTO DEAD)) => 07)
(10 000202001045 @ EVAL :       (RDX LDV (GOTO POPJ)) => 45)
(11 000202001145 @              (RDX LDV (GOTO POPJ)) => 45)
(12 001001401202 @              ((GCOP LDQ) RDN (GOTO VAR1)) => 02)
(13 000202001345 @              (RDX LDV (GOTO POPJ)) => 45)
(14 000200101453 @              ((GCOP CDR) RDX (GOTO PROC1)) => 53)
(15 000200101503 @              ((GCOP CDR) RDX (GOTO IF1)) => 03)
(16 140020001640 @              (EA-LIT ET-LIT (LIT NIL) LDL (GOTO EVARGS)) => 40)
(17 000200101730 @              ((GCOP CDR) RDX (GOTO STOREV)) => 30)
(20 001001102021                ((GCOP CONS) RDN) => 21)
(21 100041612161                ((GCOP RDQA) ET-LIT (LIT IF2) LDC (GOTO IF1A)) => 61)
(22 000040302224   IF3 :        ((GCOP CDRQ) LDC) => 24)
(23 000105002302 #              ((GCOP CDRQX) RDV LDX (GOTO VAR1)) => 02)
(24 020100702433                ((GCOP CARRX) RDV EAZ-ETZ-EQV-DISP (GOTO IF4)) => 33)
(25 000000402526   CONS1 :      ((GCOP CARQ)) => 26)
(26 000101202630                ((GCOP XCONS) RDV) => 30)
(27 000002402745 #              ((GCOP CARQ) LDV (GOTO POPJ)) => 45)
(30 000003503045   STOREV :     ((GCOP RDQ) LDV (GOTO POPJ)) => 45)
(31 000011503132   FUN1 :       ((GCOP RDQ) LDN) => 32)
(32 022000103263                ((GCOP CDR) RDL EAZ-ETZ-EQV-DISP) => 63)
(33 010004403310 % (INIT1 IF4): ((GCOP CARQ) LDX ET-DISP (GOTO EVAL)) => 10)
(34 002010003435   FUN3 :       (RDL LDN) => 35)
(35 000100203562                ((GCOP CAR) RDV (GOTO FUN4)) => 62)
(36 004001203654   EV1 :        ((GCOP XCONS) RDC) => 54)
(37 010004303710 %              ((GCOP CDRQ) LDX ET-DISP (GOTO EVAL)) => 10)
(40 000200104036 & EVARGS :     ((GCOP CDR) RDX (GOTO EV1)) => 36)
(41 000100204130 &              ((GCOP CAR) RDV (GOTO STOREV)) => 30)
(42 000100104230 &              ((GCOP CDR) RDV (GOTO STOREV)) => 30)
(43 002000104325 &              ((GCOP CDR) RDL (GOTO CONS1)) => 25)
(44 020100004455 &              (RDV EAZ-ETZ-EQV-DISP (GOTO ATOM1)) => 55)
(45 014000204504 & POPJ :       ((GCOP CAR) RDC ET-DISP (GOTO POPJ1)) => 04)
(46 002002004645 &              (RDL LDV (GOTO POPJ)) => 45)
(47 000100104731 &              ((GCOP CDR) RDV (GOTO FUN1)) => 31)
(50 000000605051                ((GCOP CDRRX)) => 51)
(51 000040305152                ((GCOP CDRQ) LDC) => 52)
(52 010004505240                ((GCOP CARR) LDX ET-DISP (GOTO EVARGS)) => 40)
(53 001001205370   PROC1 :      ((GCOP XCONS) RDN) => 70)
```

```
(54 001001105456                ((GCOP CONS) RDN) => 56)
(55 140002015545 + ATOM1 :      (EA-LIT ET-LIT (LIT T) LDV (GOTO POPJ)) => 45)
(56 002001105660                ((GCOP CONS) RDL) => 60)
(57 140002005745 +              (EA-LIT ET-LIT (LIT NIL) LDV (GOTO POPJ)) => 45)
(60 100041606061                ((GCOP RDQA) ET-LIT (LIT EV2) LDC) => 61)
(61 000200206162   IF1A :       ((GCOP CAR) RDX) => 62)
(62 010005506210   FUN4 :       ((GCOP RDQ) LDX ET-DISP (GOTO EVAL)) => 10)
(63 020000306363 \ FUN2 :       ((GCOP CDRQ) EAZ-ETZ-EQV-DISP (GOTO FUN2)) => 63)
(64 000021506465   EV3 :        ((GCOP RDQ) LDL) => 65)
(65 004000106566                ((GCOP CDR) RDC) => 66)
(66 000010406650                ((GCOP CARQ) LDN) => 50)
(67 001001306734 \              ((GCOP RPLACDR) RDN (GOTO FUN3)) => 34)
(70 100003637045                ((GCOP RDQA) ET-LIT (LIT CLOSURE) LDV (GOTO POPJ)) => 45)
(71 000101307107   STORE6 :     ((GCOP RPLACDR) RDV (GOTO DEAD)) => 07)
(REMARK NUMBER OF INSTRUCTIONS = 72 (OCTAL))
)       ;END OF UCODE EVAL

(UCODE GC
(100 WORDS)
(REMARK MICROWORDS ARE 34 (OCTAL) BITS WIDE)
(FIELD 1000000000 (GA-ZERO-DISP BIT 1 33))
(FIELD 0400000000 (GCOP-DISP BIT 1 32))
(FIELD 0200000000 (WRITE BIT 1 31))
(FIELD 0100000000 (ADR BIT 1 30))
(FIELD 0040000000 (RDR BIT 1 27))
(FIELD 0020000000 (RDQ BIT 1 26))
(FIELD 0010000000 (RDP+ BIT 1 25))
(FIELD 0004000000 (RDP BIT 1 24))
(FIELD 0002000000 (LDS BIT 1 23))
(FIELD 0001000000 (LDR BIT 1 22))
(FIELD 0000400000 (LDQ BIT 1 21))
(FIELD 0000200000 (LDP BIT 1 20))
(FIELD 0000100000 (CONN-I BIT 1 17))
(FIELD 0000040000 (CONN-ET BIT 1 16))
(FIELD 0000020000 (CONN-EA BIT 1 15))
(FIELD 0000010000 (STEP-EVAL BIT 1 14))
(FIELD 0000007700 (GOSTATE OSTATE 6 6))
(FIELD 0000004000 (GOSTATE0 BIT 1 13))
(FIELD 0000002000 (GOSTATE1 BIT 1 12))
(FIELD 0000001000 (GOSTATE2 BIT 1 11))
(FIELD 0000000400 (GOSTATE3 BIT 1 10))
(FIELD 0000000200 (GOSTATE4 BIT 1 7))
(FIELD 0000000100 (GOSTATE5 BIT 1 6))
(FIELD 0000000077 (GNSTATE NSTATE 6 0))
(FIELD 0000000040 (GNSTATE0 BIT 1 5))
(FIELD 0000000020 (GNSTATE1 BIT 1 4))
(FIELD 0000000010 (GNSTATE2 BIT 1 3))
(FIELD 0000000004 (GNSTATE3 BIT 1 2))
(FIELD 0000000002 (GNSTATE4 BIT 1 1))
(FIELD 0000000001 (GNSTATE5 BIT 1 0))
(00 0102420001 ! GC :           (CONN-EA LDS LDQ ADR) => 01)
```

```
(01 0000310102                (CONN-I LDP STEP-EVAL (GOTO LOOP)) => 02)
(02 0400000220   LOOP :       (GCOP-DISP) => 20)
(03 0001260304   CAR1 :       (CONN-EA CONN-ET LDP LDR) => 04)
(04 0012000405   CAR2 :       (RDP+ LDS) => 05)
(05 0020200506                (RDQ LDP) => 06)
(06 0000510602   (CAR3 CDR1): (CONN-I LDQ STEP-EVAL (GOTO LOOP)) => 02)
(07 0020200710   CARQ1 :      (RDQ LDP) => 10)
(10 0012001011                (RDP+ LDS) => 11)
(11 0040201112                (RDR LDP) => 12)
(12 0021001213   CDRQ1 :      (RDQ LDR) => 13)
(13 0000571302                (CONN-I CONN-EA CONN-ET LDQ STEP-EVAL (GOTO LOOP)) => 02)
(14 0040201404   CARRX1 :     (RDR LDP (GOTO CAR2)) => 04)
(15 0002461516   RPLACDR1 :   (CONN-EA CONN-ET LDS LDQ) => 16)
(16 0200011602                (WRITE STEP-EVAL (GOTO LOOP)) => 02)
(17 0021001742   XCONS2 :     (RDQ LDR) => 42)
(20 0000012002 @              (STEP-EVAL (GOTO LOOP)) => 02)
(21 0103062106 @              (CONN-EA CONN-ET LDS LDR ADR (GOTO CDR1)) => 06)
(22 0004402203 @              (RDP LDQ (GOTO CAR1)) => 03)
(23 0122002312 @              (RDQ LDS ADR (GOTO CDRQ1)) => 12)
(24 0005002407 @ CARQ0 :      (RDP LDR (GOTO CARQ1)) => 07)
(25 0040402524 @              (RDR LDQ (GOTO CARQ0)) => 24)
(26 0142002606 @              (RDR LDS ADR (GOTO CDR1)) => 06)
(27 0004402714 @              (RDP LDQ (GOTO CARRX1)) => 14)
(30 0123003006 @              (RDQ LDS LDR ADR (GOTO CDR1)) => 06)
(31 1113003144 @              (RDP+ LDS LDR ADR GA-ZERO-DISP (GOTO CONS1)) => 44)
(32 1113003240 @              (RDP+ LDS LDR ADR GA-ZERO-DISP (GOTO XCONS1)) => 40)
(33 0142003315 @              (RDR LDS ADR (GOTO RPLACDR1)) => 15)
(34 0000473402 @              (CONN-EA CONN-ET LDQ STEP-EVAL (GOTO LOOP)) => 02)
(35 0020073502 @              (CONN-EA CONN-ET RDQ STEP-EVAL (GOTO LOOP)) => 02)
(36 0020033602 @              (CONN-EA RDQ STEP-EVAL (GOTO LOOP)) => 02)
(37 0142003755 @              (RDR LDS ADR (GOTO RDQCDRRX1)) => 55)
(40 0040204017 $ XCONS1 :     (RDR LDP (GOTO XCONS2)) => 17)
(41 0000004141 $ GCDEAD :     ((GOTO GCDEAD)) => 41)
(42 0002064247                (CONN-EA CONN-ET LDS (GOTO CONS3)) => 47)
(43 0001064346   CONS2 :      (CONN-EA CONN-ET LDR) => 46)
(44 0040204443 % CONS1 :      (RDR LDP (GOTO CONS2)) => 43)
(45 0000004541 %              ((GOTO GCDEAD)) => 41)
(46 0022004647                (RDQ LDS) => 47)
(47 0204404750   CONS3 :      (RDP LDQ WRITE) => 50)
(50 1112005052                (RDP+ LDS ADR GA-ZERO-DISP) => 52)
(51 0211005154   CONS4 :      (WRITE RDP+ LDR) => 54)
(52 0042005251 &              (RDR LDS (GOTO CONS4)) => 51)
(53 0000005341 &              ((GOTO GCDEAD)) => 41)
(54 0040215402                (RDR LDP STEP-EVAL (GOTO LOOP)) => 02)
(55 0021005556   RDQCDRRX1 :  (RDQ LDR) => 56)
(56 0000505657                (CONN-I LDQ) => 57)
(57 0040075702                (RDR CONN-EA CONN-ET STEP-EVAL (GOTO LOOP)) => 02)
(REMARK NUMBER OF INSTRUCTIONS = 60 (OCTAL))
)       ;END OF UCODE GC
```

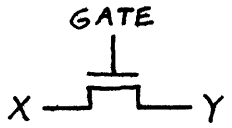
```
;It is 6 days, 15 hours, 4 minutes, and 2 seconds past the new moon.
;The sun is 41*44'49" east of south, 13*13'18" above the horizon.
;That means it is now 8:37 AM on Wednesday, December 6, 1978.
```

Logic-Level Circuit Diagrams

We conclude by giving a complete set of logic-level circuit diagrams for the SIMPLE prototype processor. The geometry of these diagrams approximately reflects the physical layout. It should be noted particularly that, while the WRITE signal emerges from the PLA at the top, clocked by PHI2, so that it can enter the probe multiplexor with the other signals, it also emerges from the bottom of the PLA before being clocked by PHI2, and goes directly to the WRITE output pad. Hence this signal appears on the output pad half a clock cycle earlier than might otherwise be expected. The timing diagram should make the external memory signals clear.

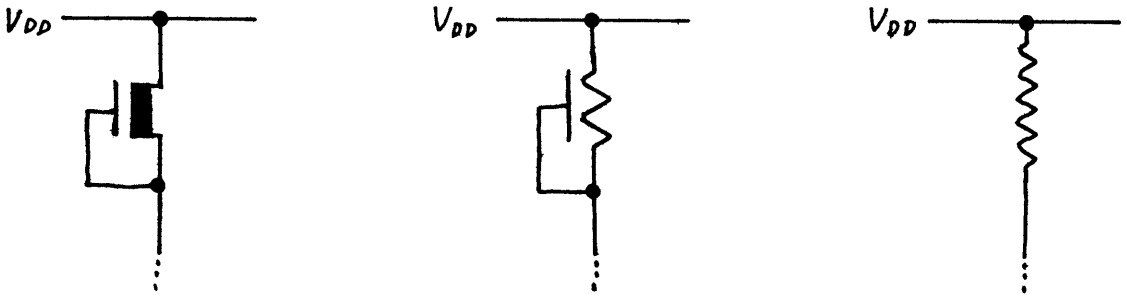
The circuits were designed to occupy minimum area with almost no thought given to speed. The register cell is a modification of that used in the OM2 data chip [Mead 1978] [Johannsen 1978]. Many of the other structures are based on ideas discussed in [Mead 1978] and [Hon 1978]. In particular, the output pads and PLA structures were taken from a library described in [Hon 1978].

For those readers not familiar with the symbology employed, here is a brief (and approximate!) explanation. The symbol



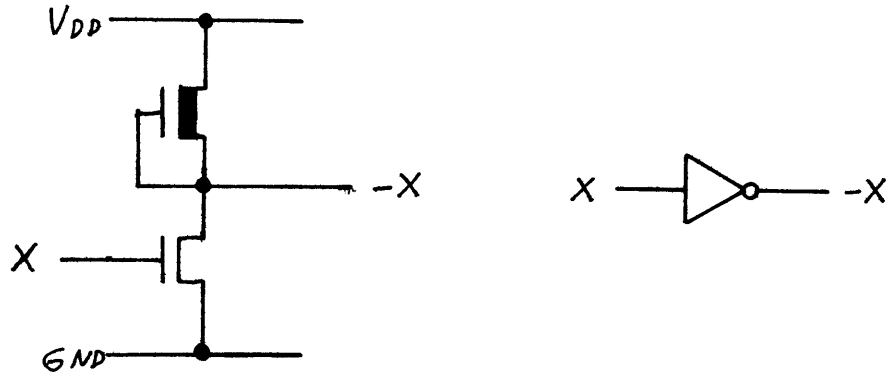
represents a pass (enhancement-mode) transistor. When the gate voltage is high (VDD), then the two arms X and Y are effectively connected; when it is low (GND), X and Y are effectively disconnected.

A transistor symbol with a filled-in body represents a transistor treated with an ion implant process so that it is always on (depletion mode). In SIMPLE this is always used in a certain configuration to get the effect of a resistor:

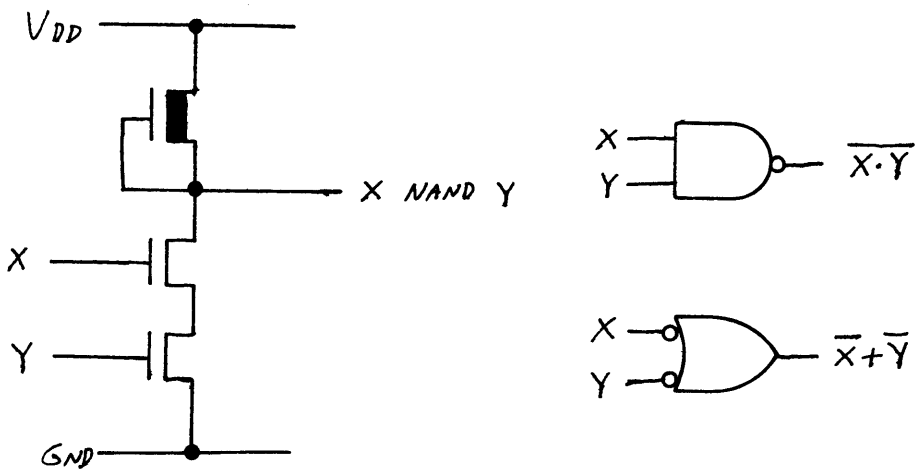


The first two symbols above are used interchangeably; the intent of either is to represent a resistor attached to VDD, as shown by the third symbol.

Such depletion-mode transistors are used to build logic gates. For example, this circuit is an inverter. If the input X is high (near VDD), the output will be low (close to GND), and vice versa. (The enhancement-mode transistor, when on, has a much lower resistance than the depletion-mode transistor. Hence when X is high the two transistors form a voltage divider which brings the output close enough to (but not actually at) GND.) This circuit is represented by the triangular logic symbol shown (which elides the essential but uninteresting details of the connections to VDD and GND).

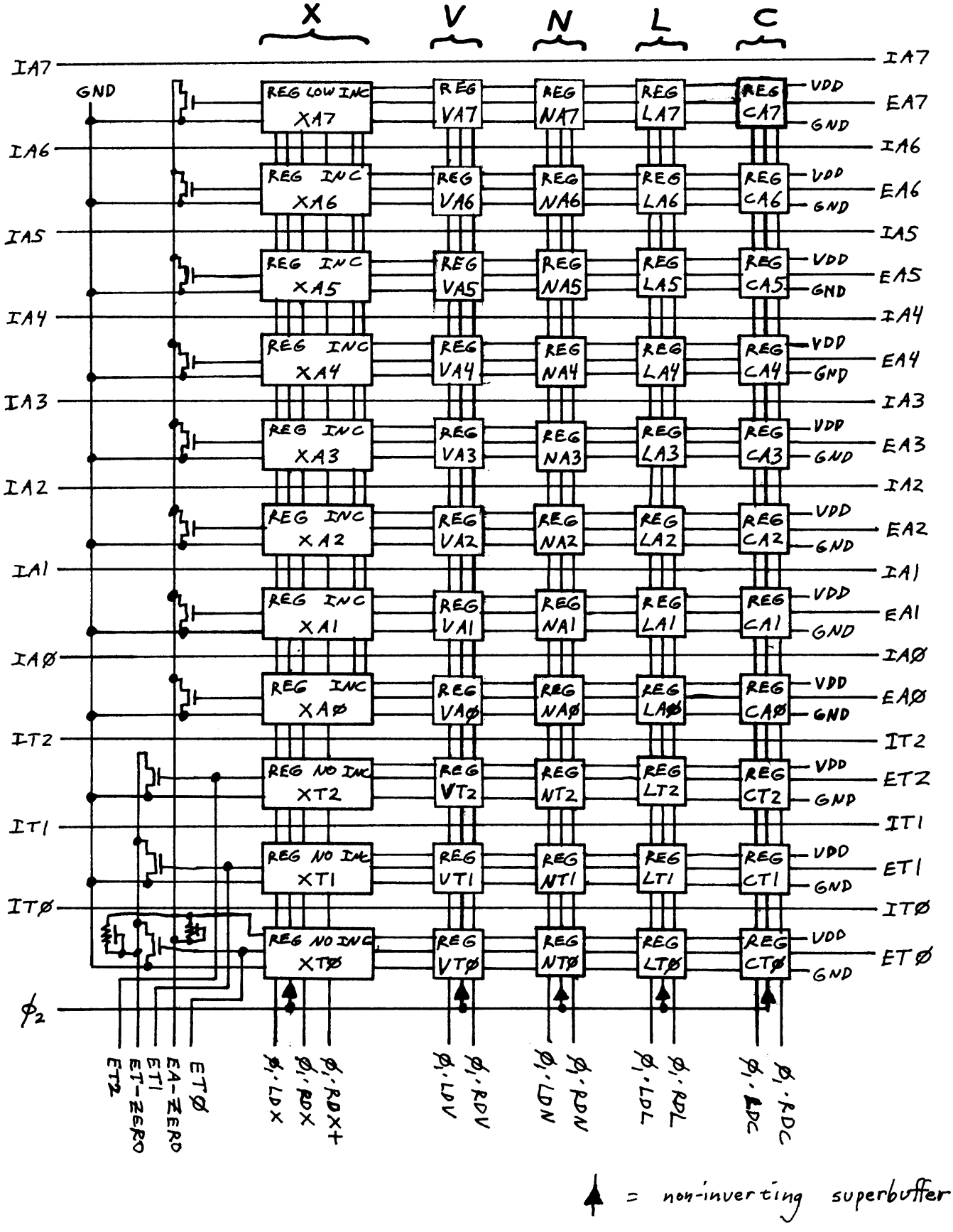


An extension of this idea allows one to construct a NAND gate easily, a circuit whose output is high iff not both inputs are high. This circuit is represented by either of two logic symbols depending on context (to emphasize one or the other of the two notations which are equivalent by DeMorgan's Law).



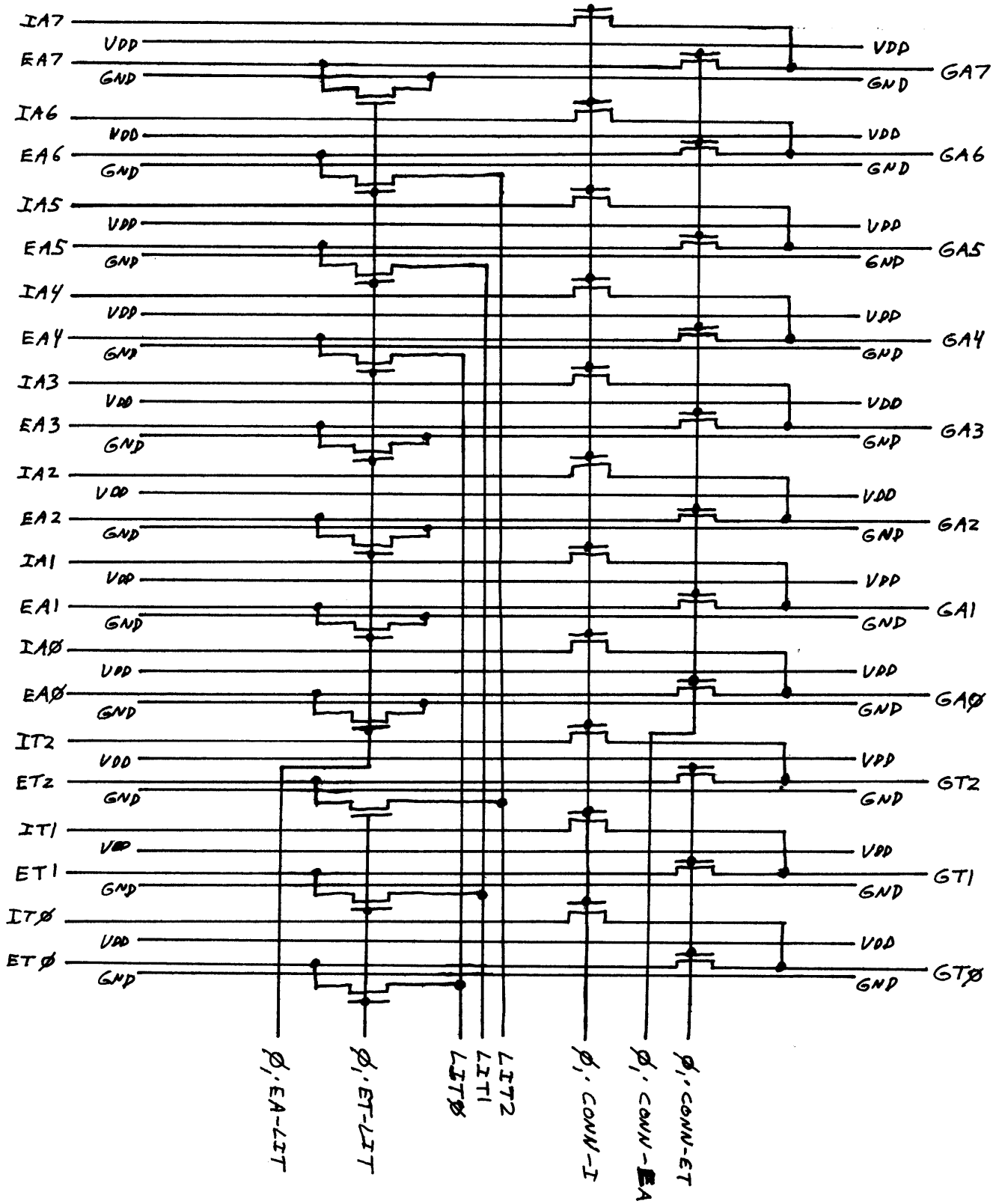
Space constraints do not permit us to exhibit complete geometrical layout diagrams. However, following the logic-level diagrams is the geometrical layout of a single register cell.

EVAL REGISTERS

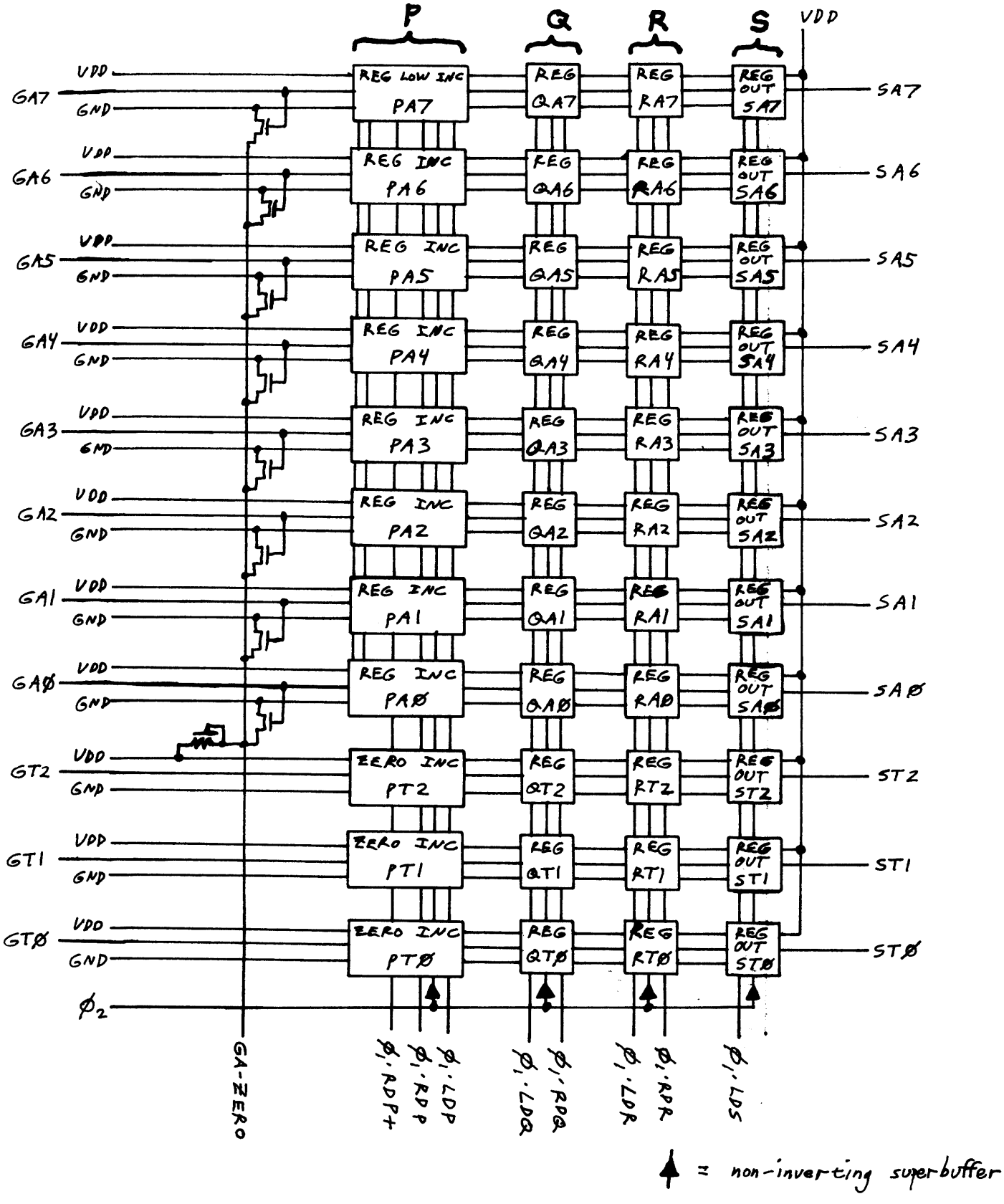


▲ = non-inverting superbuffer

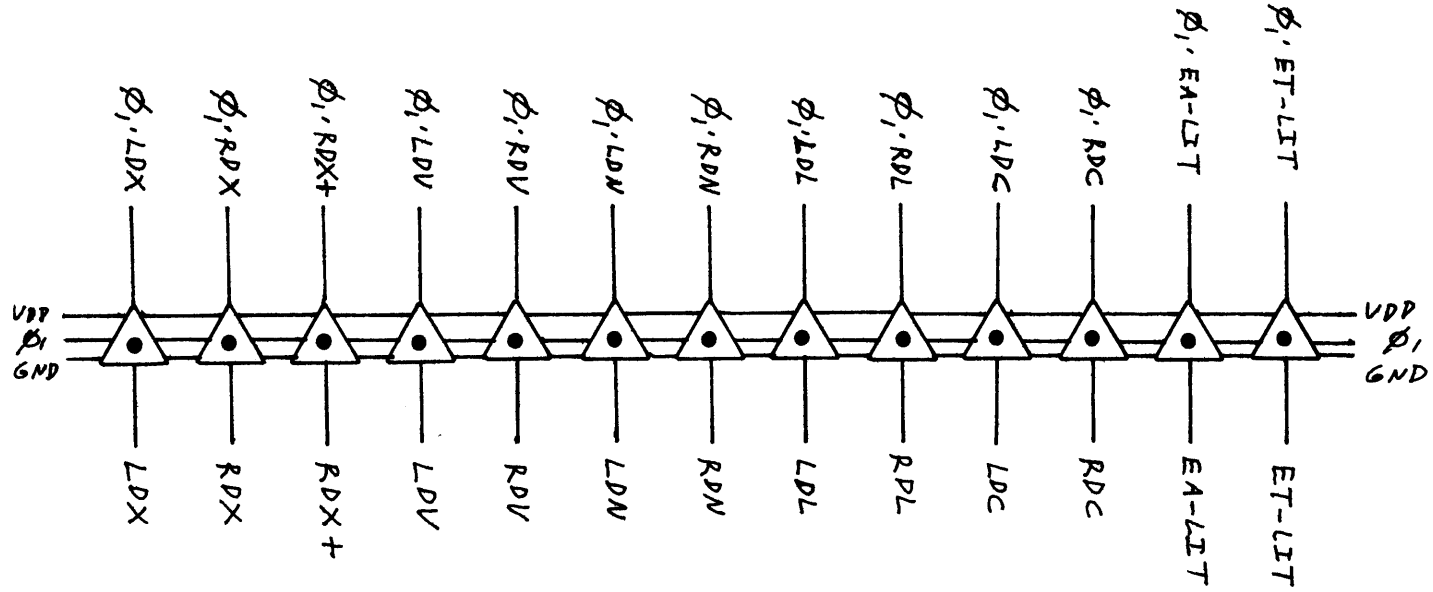
BUS INTERCONNECT



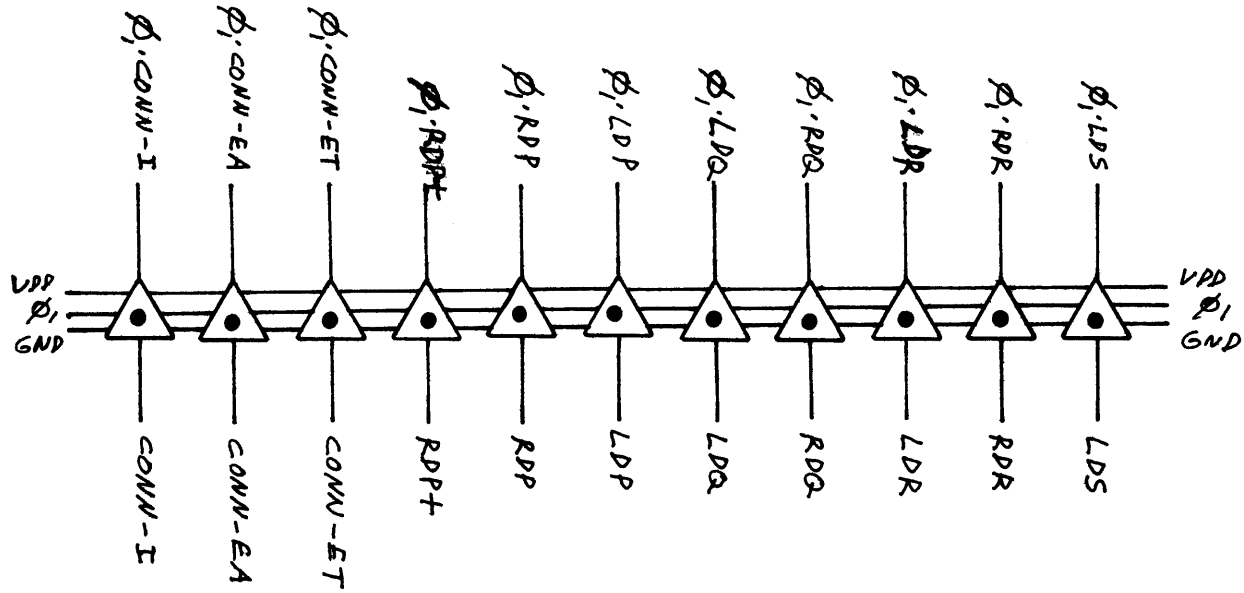
GC REGISTERS

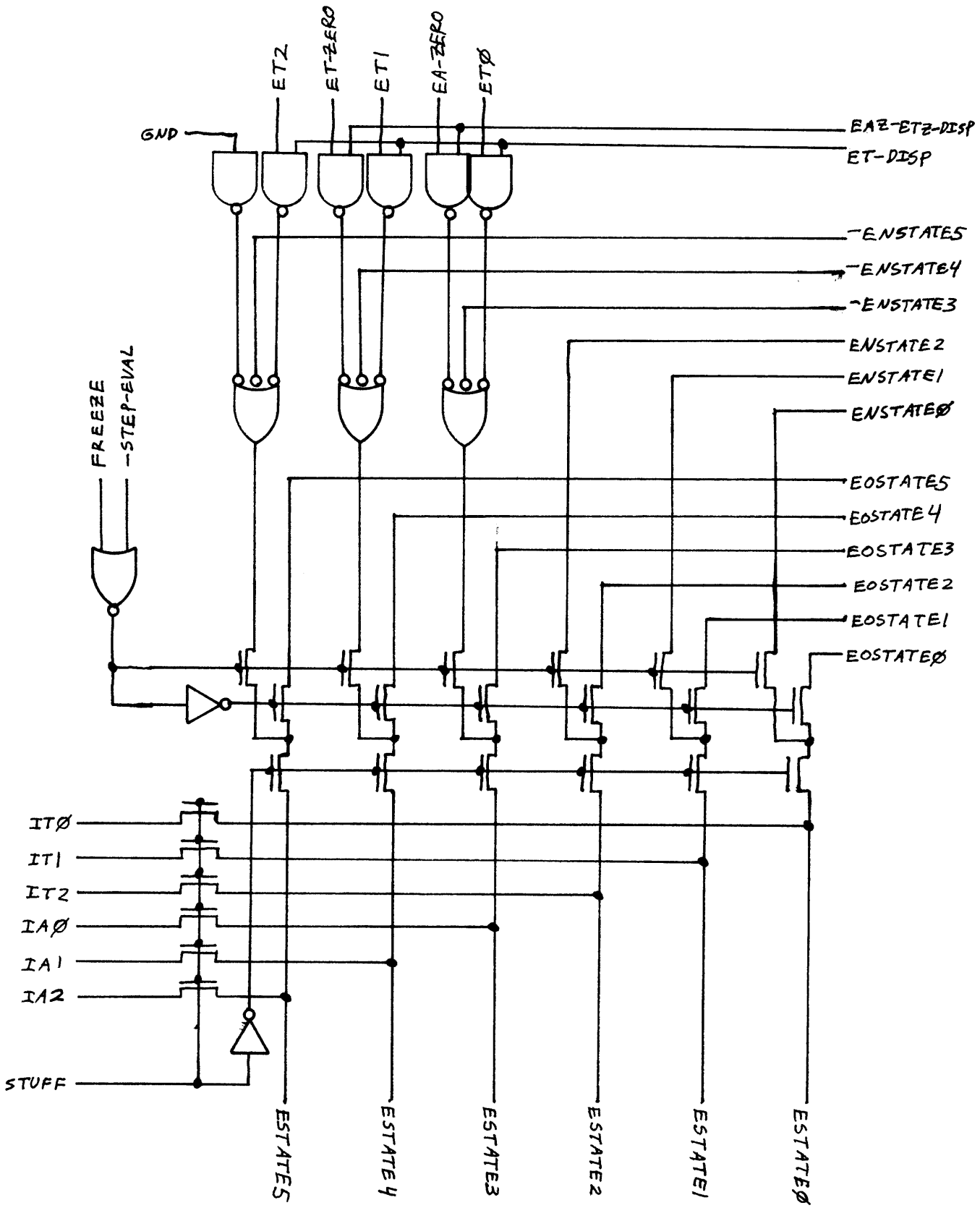


EVAL DRIVERS

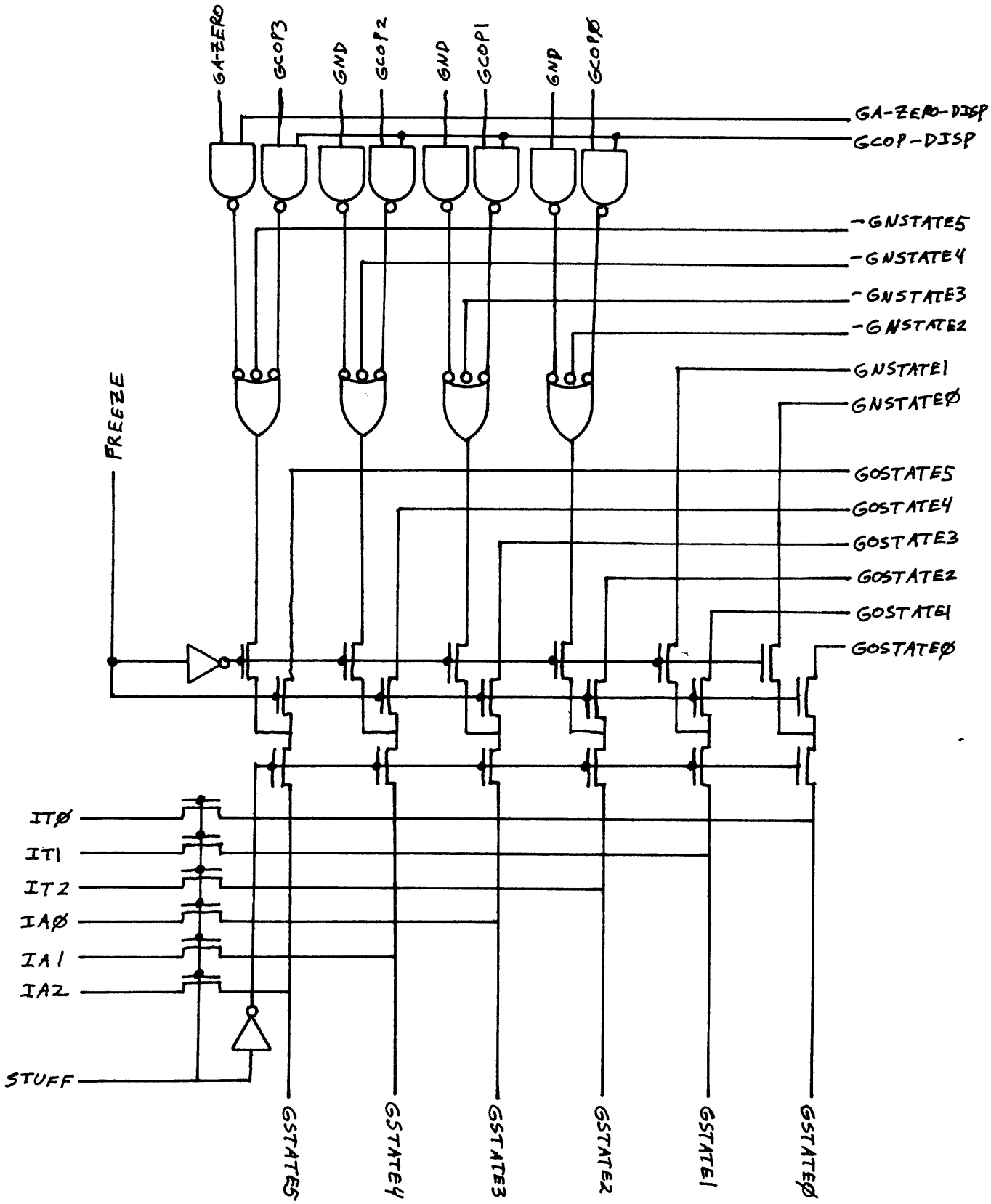


GC DRIVERS



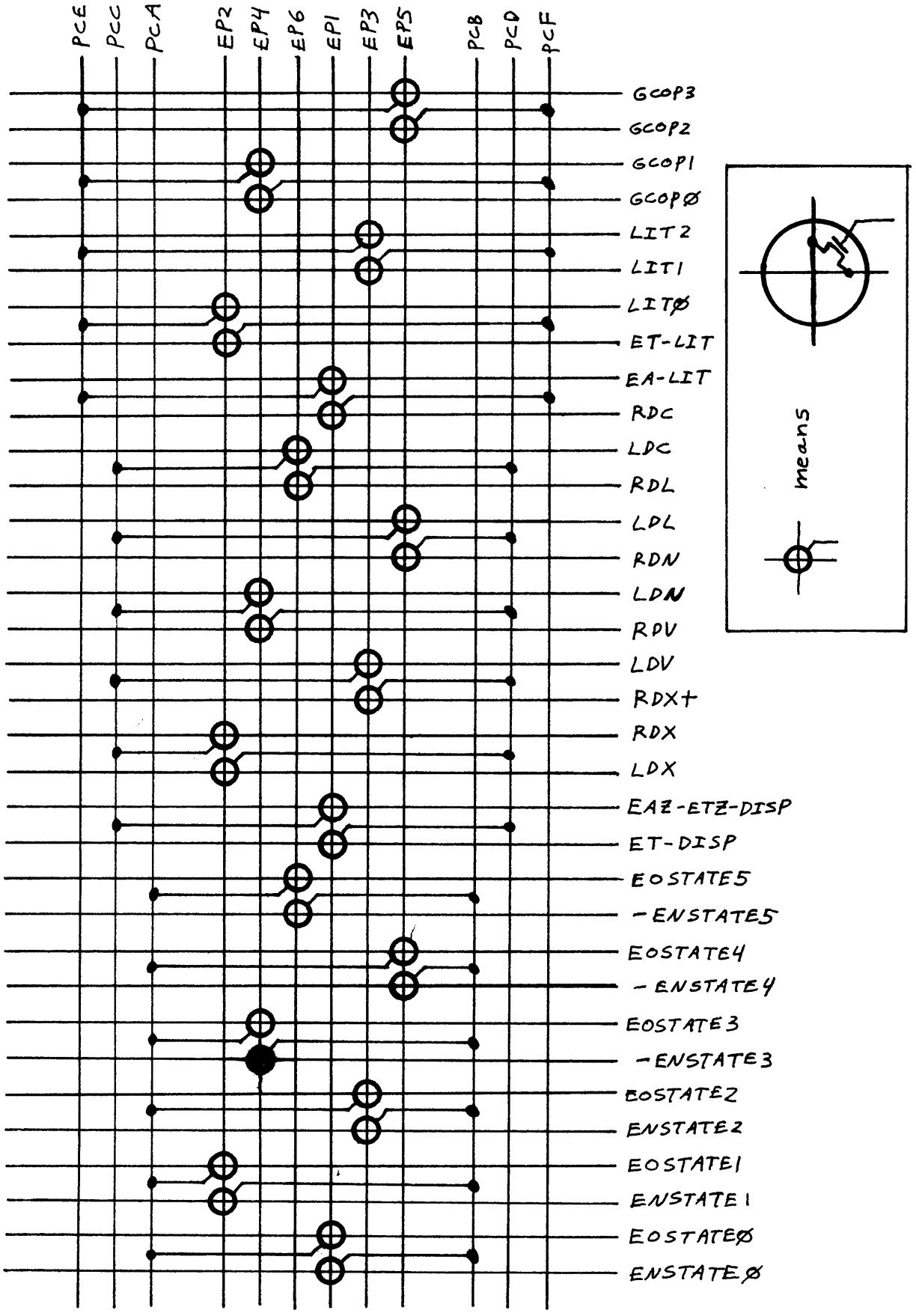


EVAL NEXT STATE

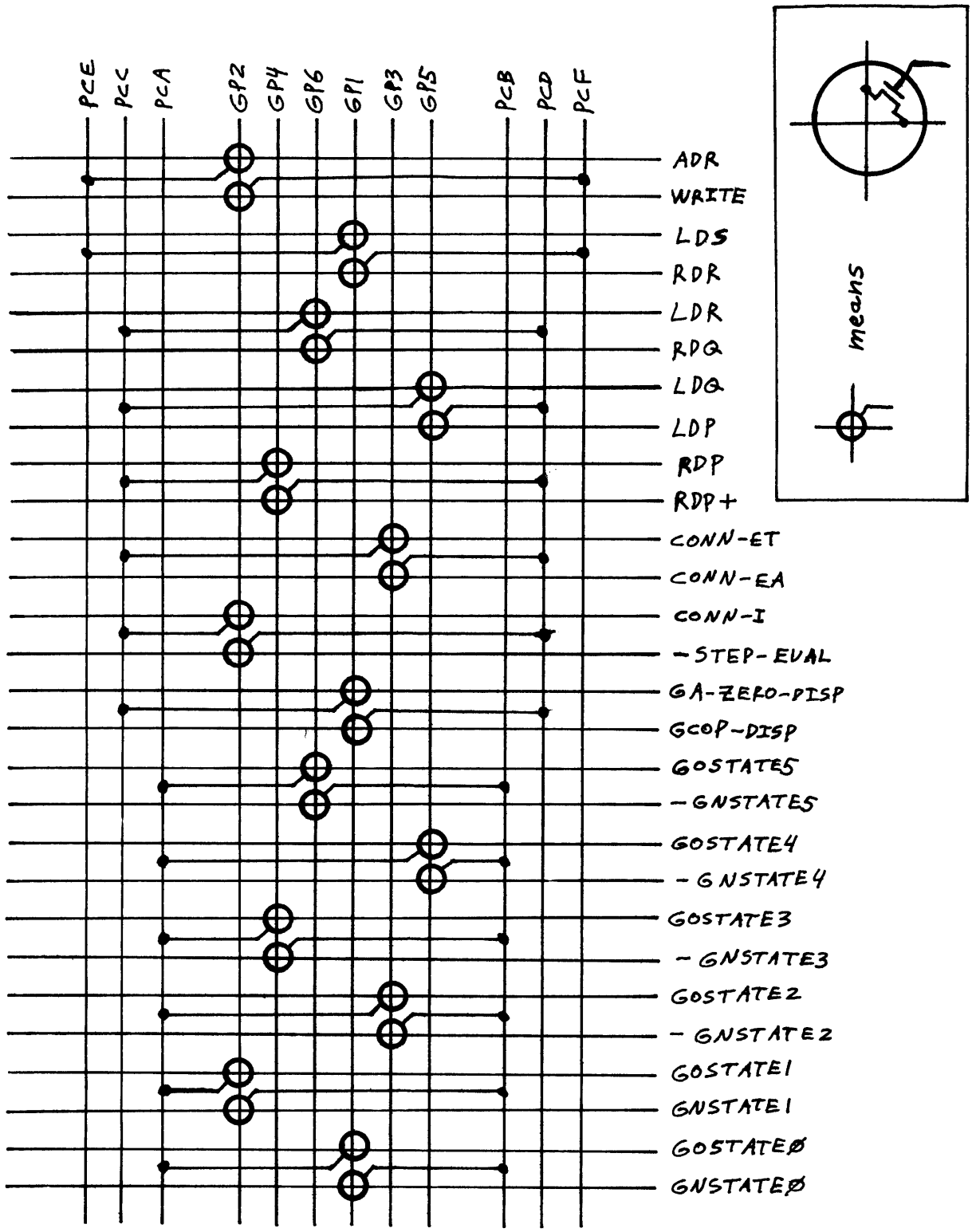


GC NEXT STATE

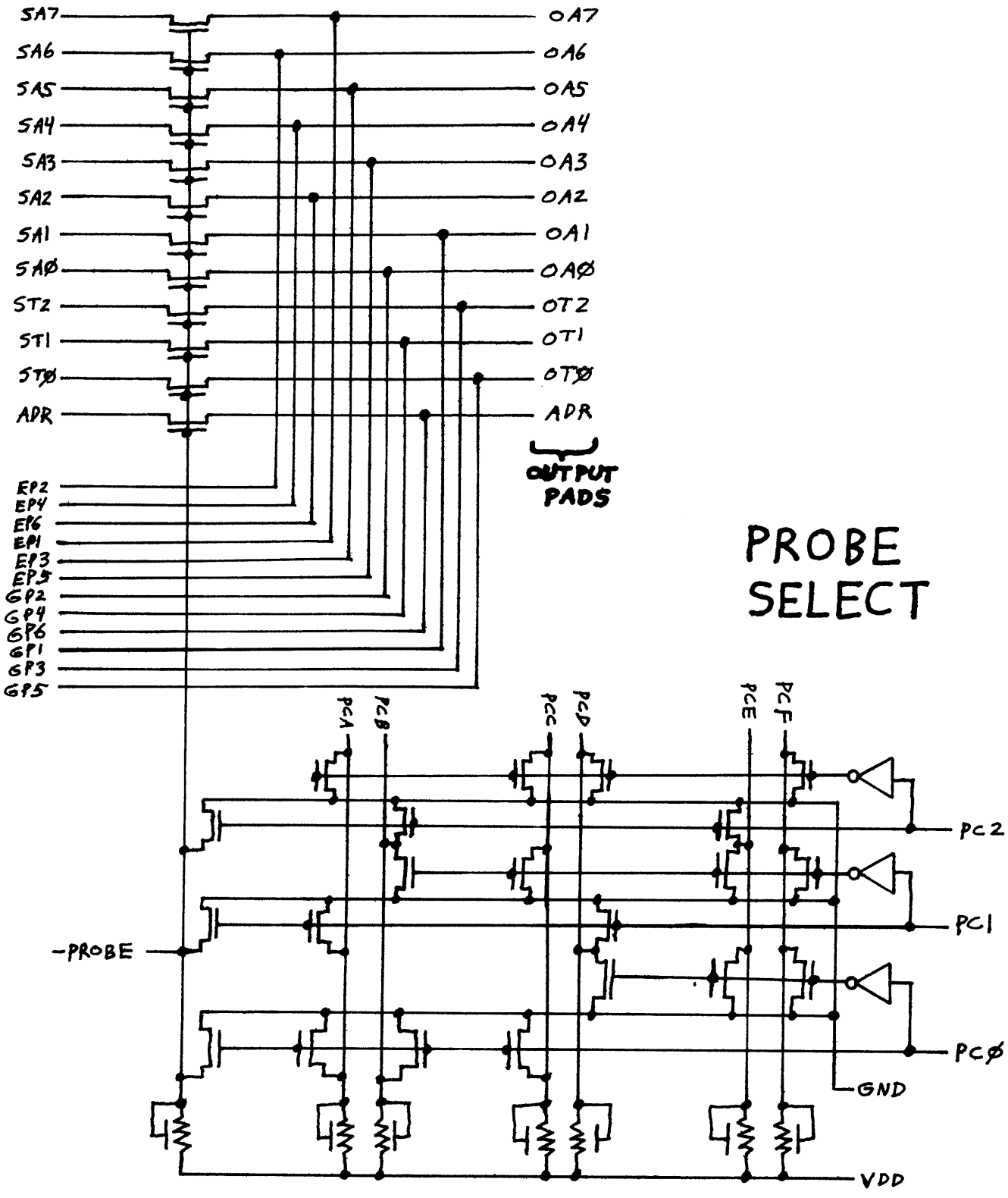
EVAL PROBE MUX



GC PROBE MUX

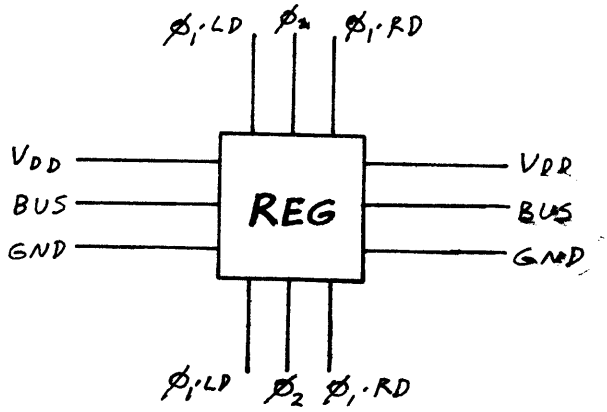

PCE
PCC
PCA
GP2
GP4
GP6
GP1
GP3
GP5
PCB
PCD
PCF
ADR
WRITE
LDS
RDR
LDR
RDQ
LDQ
LDP
RDP
RDP+
CONN-ET
CONN-EA
CONN-I
-STEP-EVAL
GA-ZERO-DISP
GCOP-DISP
GOSTATE5
-GNSTATE5
GOSTATE4
-GNSTATE4
GOSTATE3
-GNSTATE3
GOSTATE2
-GNSTATE2
GOSTATE1
GNSTATE1
GOSTATEØ
GNSTATEØ
means

OUTPUT MULTIPLEXOR

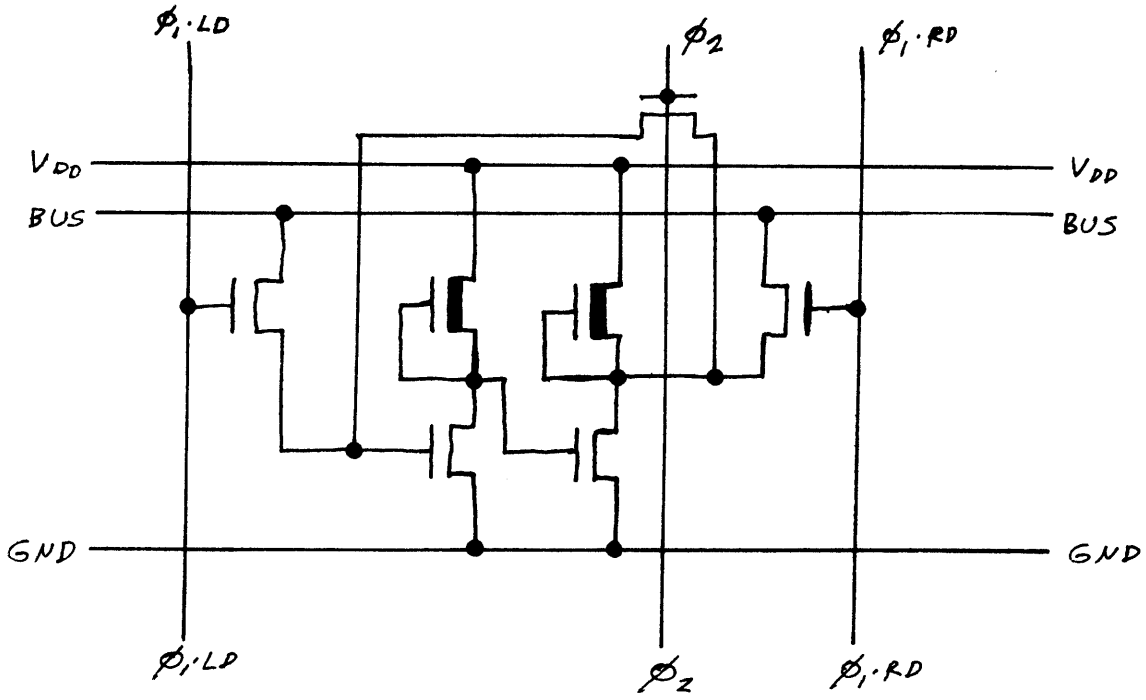


REGISTER CELL

Symbol



Circuit

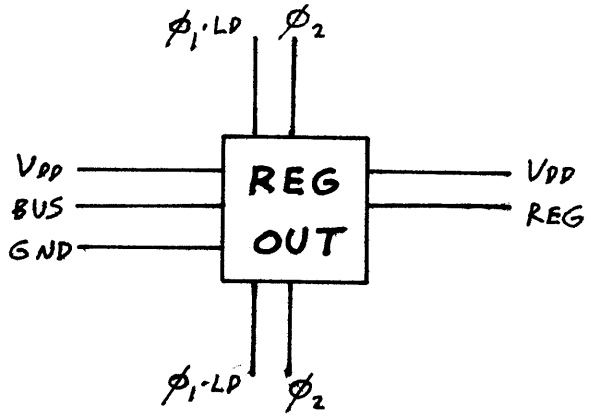


Description

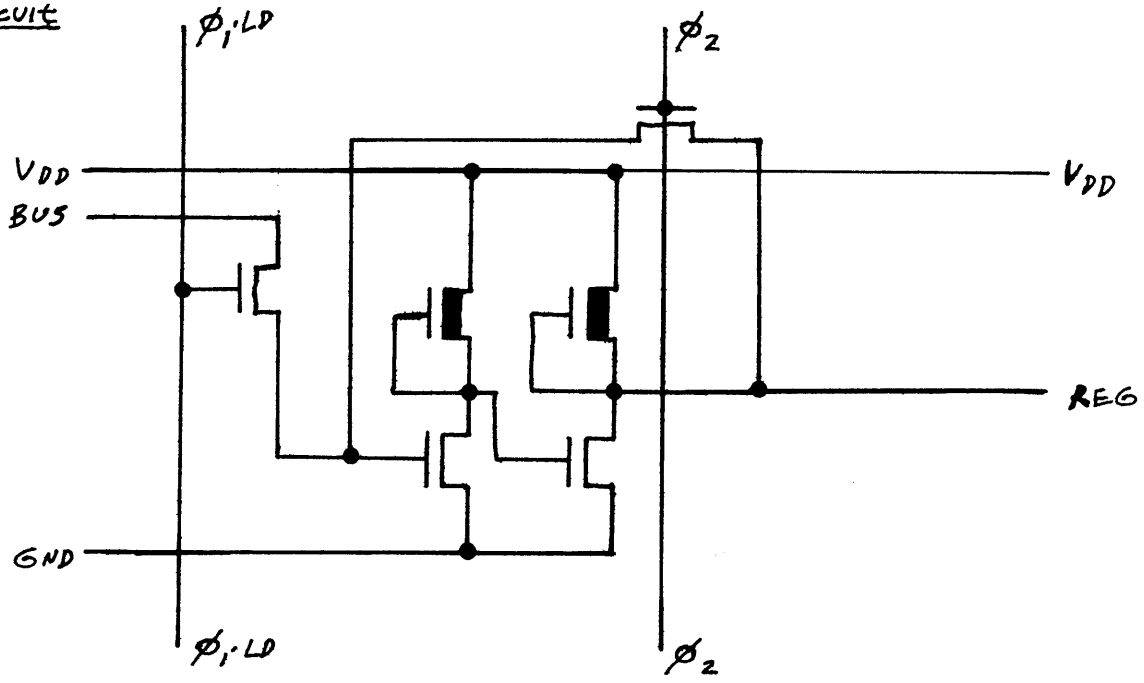
This is a one-bit register cell. The signal LD during ϕ_1 loads the cell from the bus. The signal RD during ϕ_1 drives the bus from the cell. The cell is refreshed during ϕ_2.

OUTPUT REGISTER

Symbol



Circuit

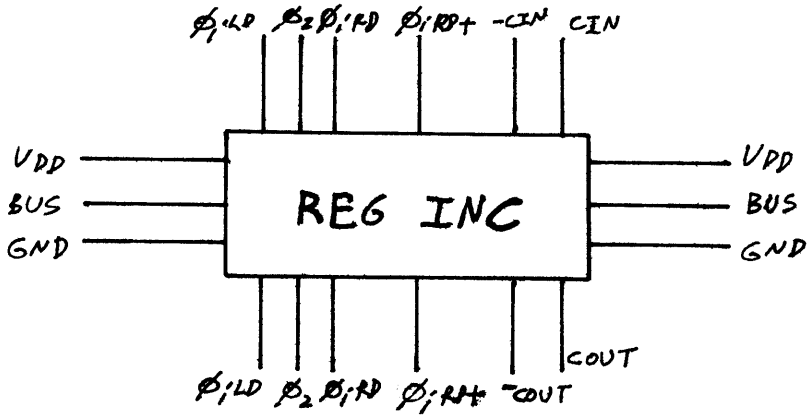


Description

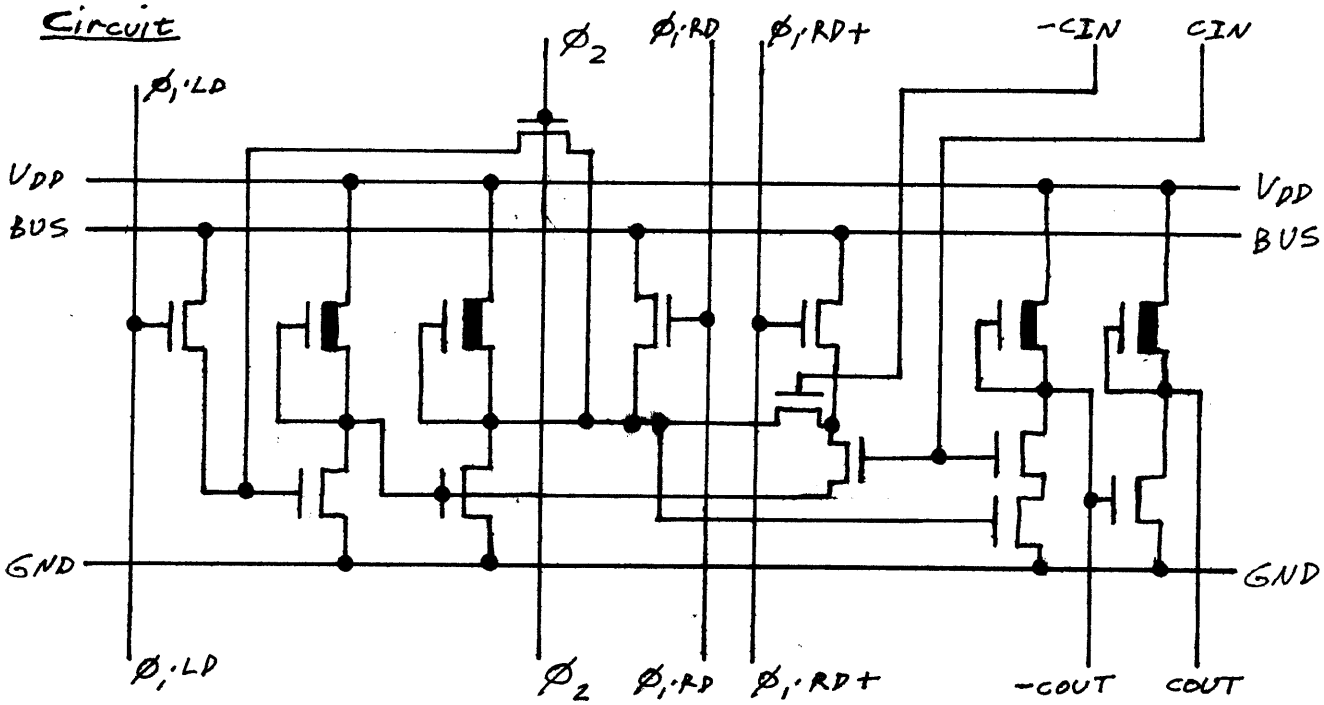
A "write-only" register cell. The register contents are continuously available. This is intended for use in latching data for output pads.

REGISTER WITH INCREMENTER

Symbol



Circuit

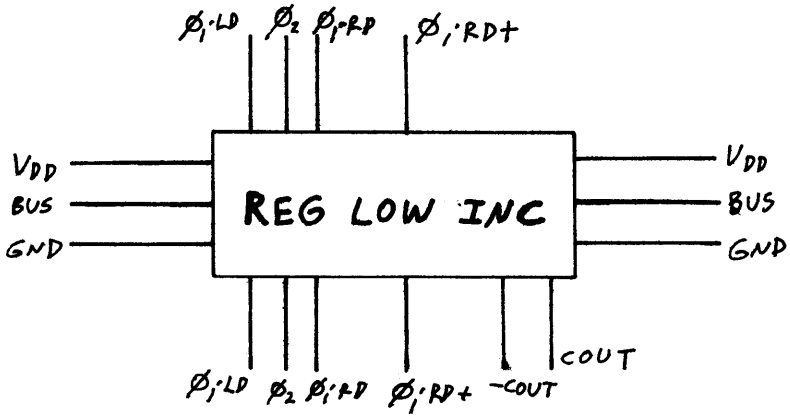


Description

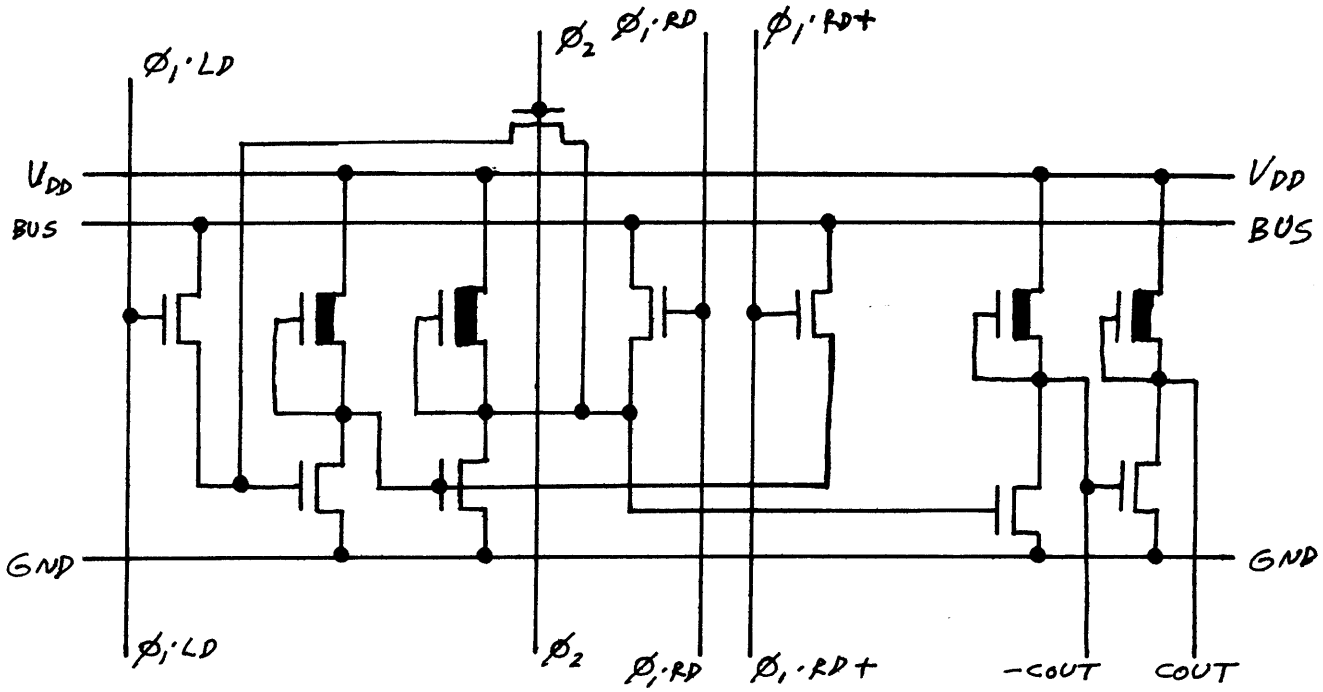
A register cell with incrementation logic. The signal RD+ will read the contents of the cell plus CIN onto the bus, generating COUT. The carry chain was designed for minimal area and is slow.

REGISTER WITH INCREMENTER — LOW BIT

Symbol



Circuit

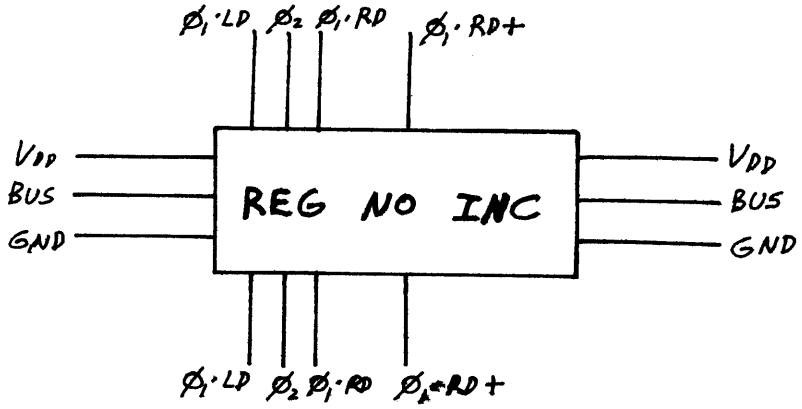


Description

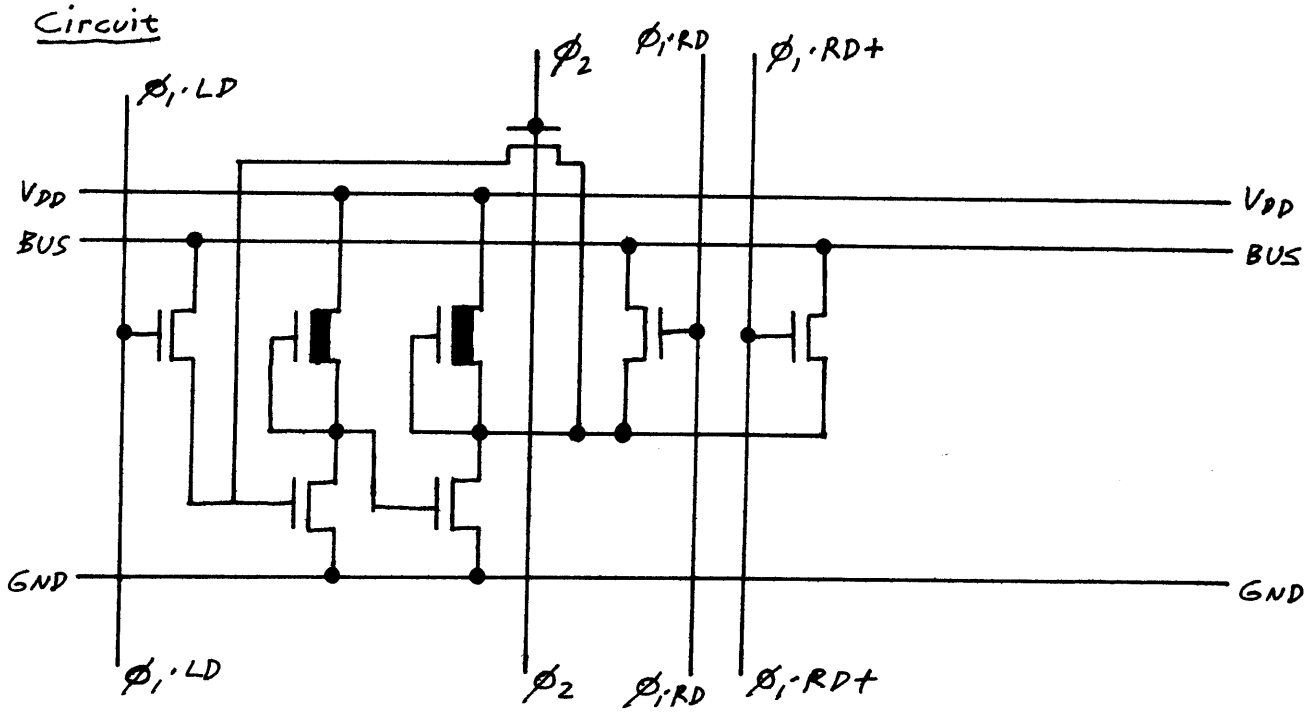
This cell is like REG INC but with CIN assumed to be forced to 1. It is therefore suitable for the low bit of a field to be incremented.

REG INC (WITHOUT THE INC)

Symbol



Circuit

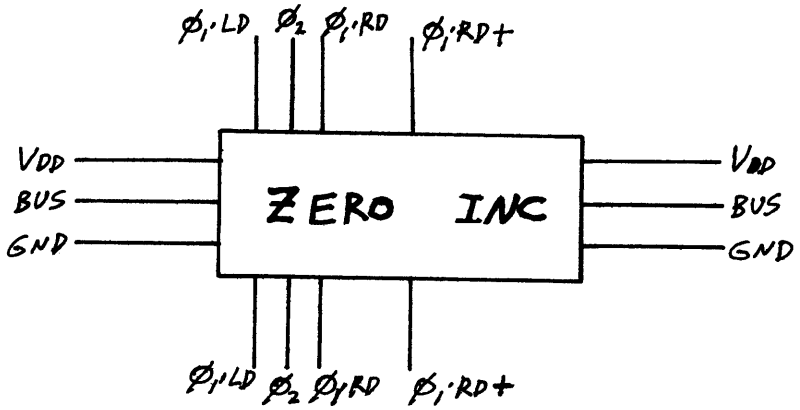


Description

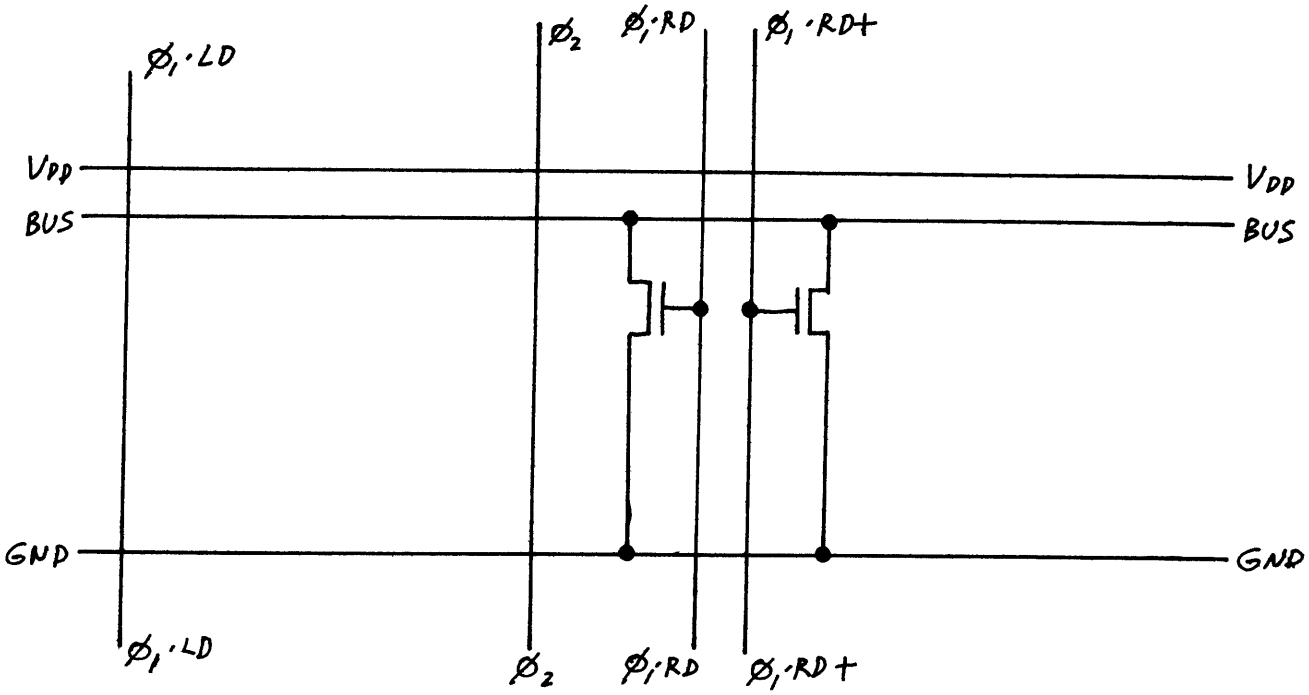
This is just an ordinary register cell designed to be used with REG INC. Either $\phi_1 \cdot RD$ or $\phi_1 \cdot RD+$ will read the (unincremented!) contents onto the bus.

ZERO REGISTER

Symbol



Circuit

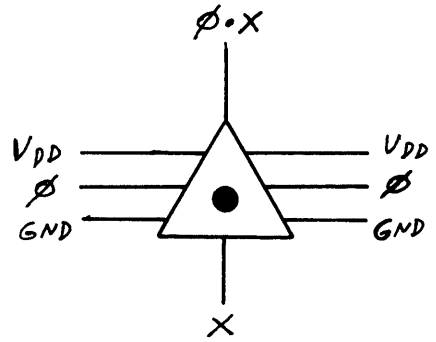


Description

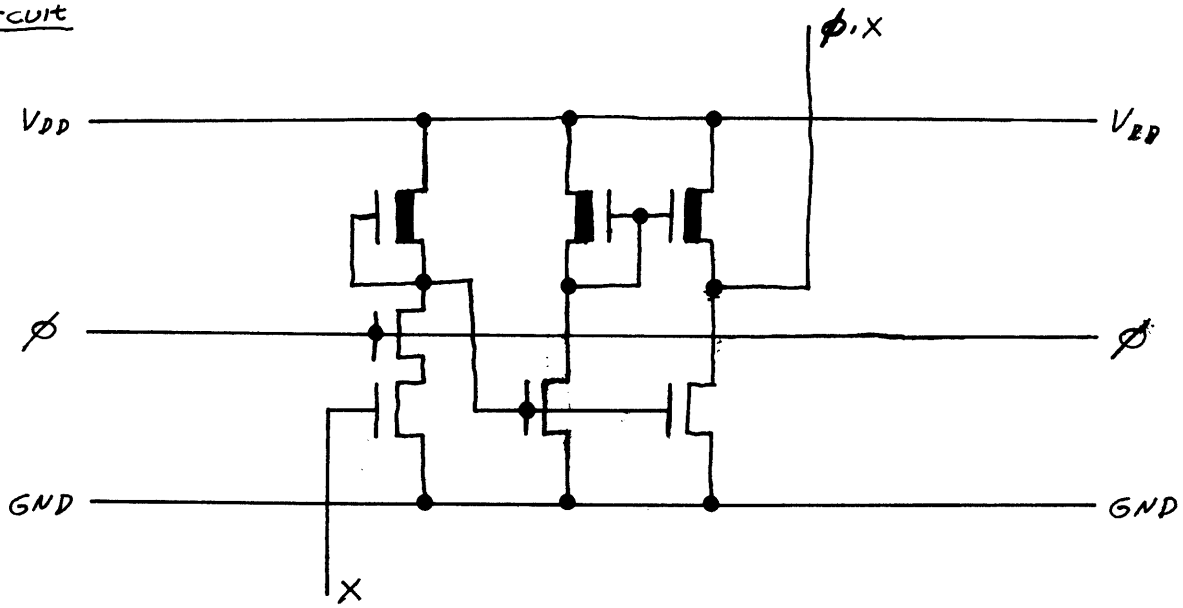
This is a dummy register cell for use with REG INC. It throws away bits when loaded and always supplies a zero when read.

CLOCKED DRIVER

Symbol

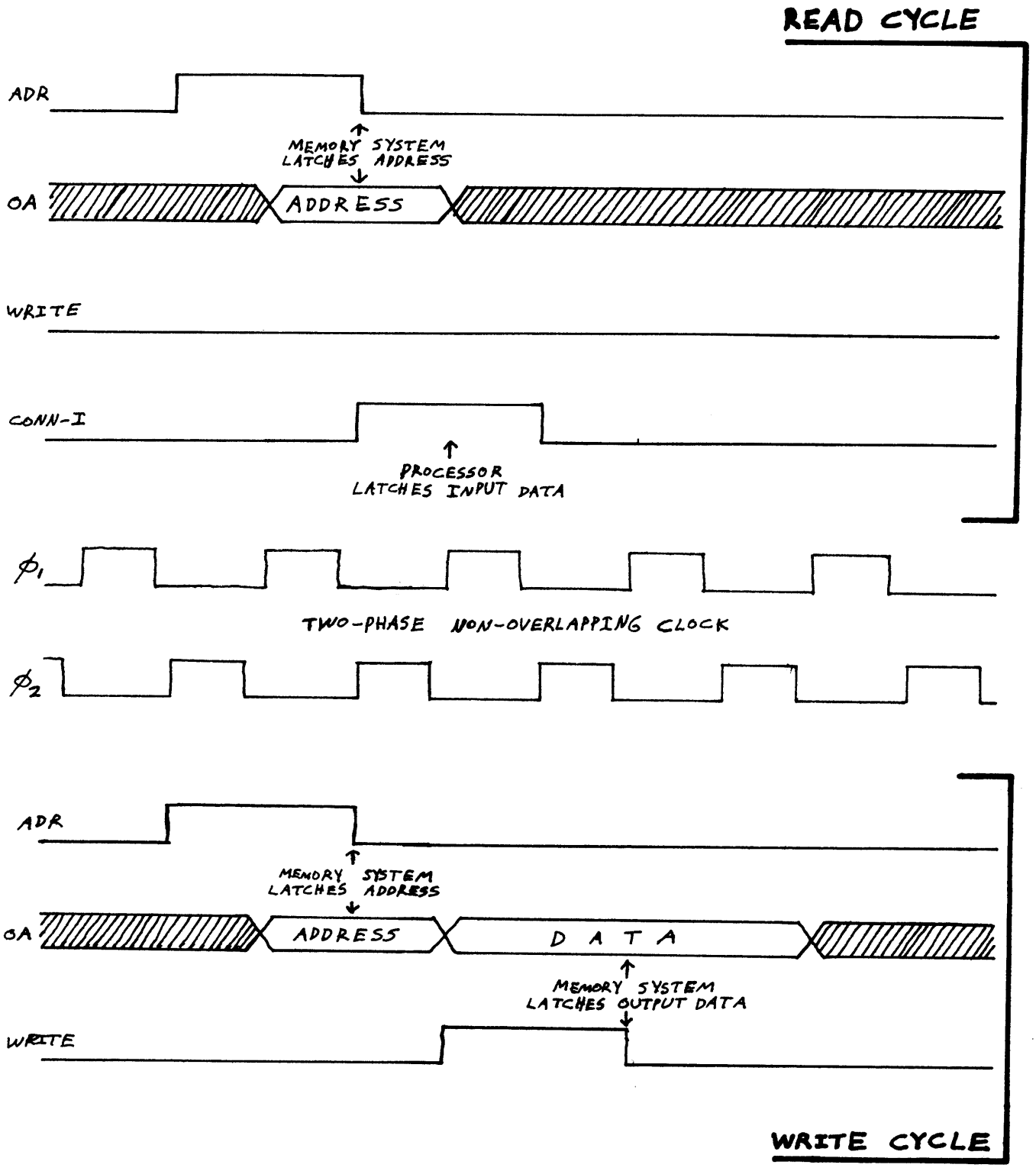


Circuit



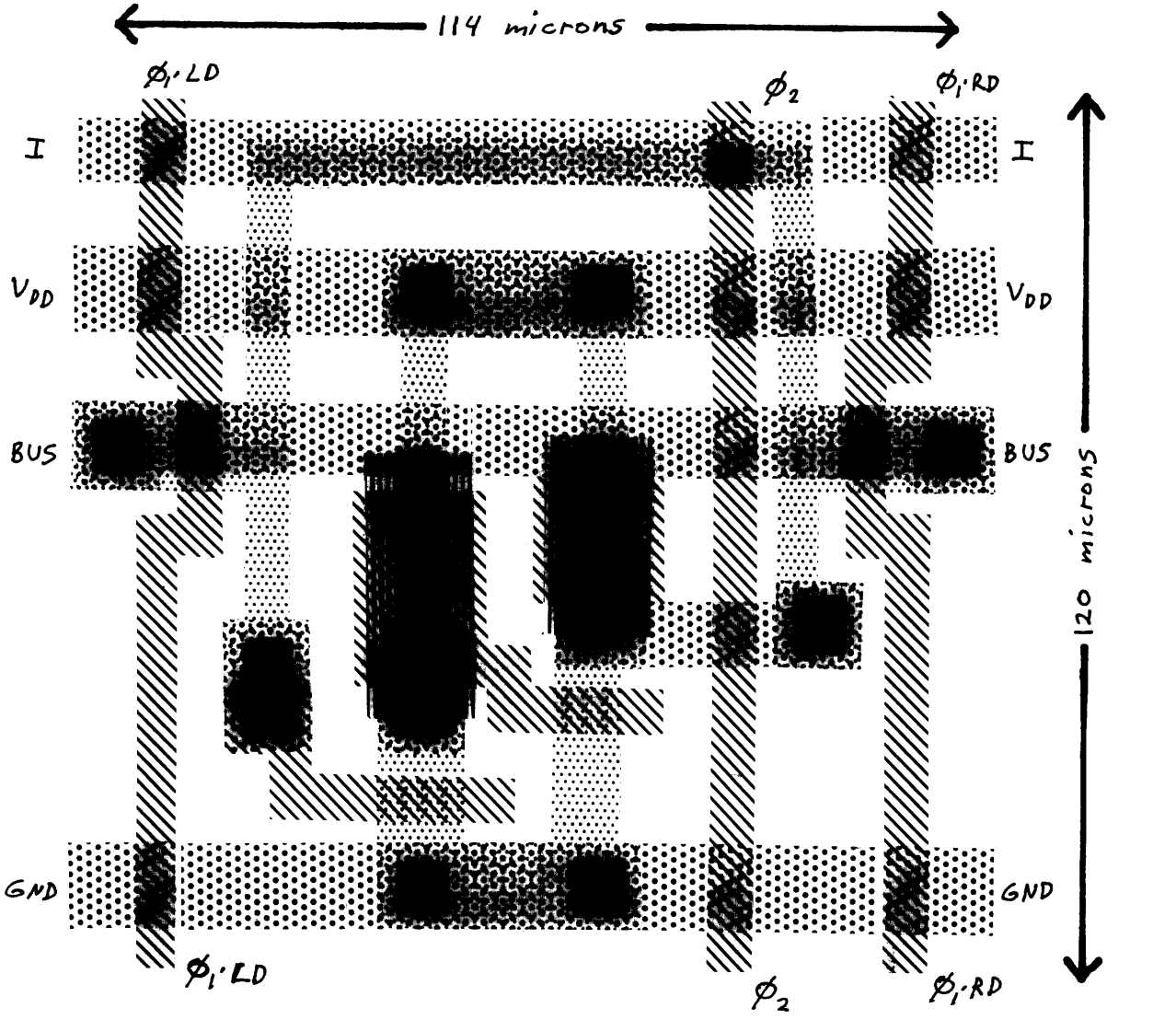
Description

This is just a NAND gate plus an inverting super-buffer. It is intended to drive long signal lines, during ϕ_1 only (so that during ϕ_2 busses may be precharged, for example).



MEMORY CYCLE TIMING

REGISTER CELL LAYOUT



Minimum line width for poly and diffusion is 6 microns.
Minimum line width for metal is 9 microns.
When the cell is replicated horizontally, bus contacts for adjacent cells are superposed.

References

[Backus 1978]
Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Comm. ACM 21, 8 (August 1978), 613-641.

[Baker 1978]
Baker, Henry B., Jr. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.

[Berkeley 1964]
Berkeley, Edmund C., and Bobrow, Daniel G. (Eds.) The Programming Language LISP: Its Operation and Applications. Information International, Inc. (Cambridge, 1964).

[Bobrow 1973]
Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." Comm. ACM 16, 10 (October 1973) pp. 591-603.

[Caples 1925]
Caples, John. "They Laughed When I Sat Down At the Piano But When I Started to Play!—" Advertisement for the U.S. School of Music. Reprinted in Rowsome, Frank Jr. They Laughed When I Sat Down: An informal history of advertising in words and pictures. Bonanza Books (New York, 1959), page 153.

[Conrad 1974]
Conrad, William R. A compactifying garbage collector for ECL's non-homogeneous heap. Technical Report 2-74. Center for Research in Computing Technology, Harvard U. (Cambridge, February 1974).

[Galley 1975]
Galley, S.W. and Pfister, Greg. The MDL Language. Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).

[Greenblatt 1974]
Greenblatt, Richard. The LISP Machine. Artificial Intelligence Working Paper 79, MIT (Cambridge, November 1974).

[Hansen 1969]
Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation." Comm. ACM 12, 9 (September 1969), 499-507.

[Hart 1964]
Hart, Timothy P., and Evans, Thomas G. "Notes on implementing LISP for the M-460 computer." In Berkeley and Bobrow, The Programming Language LISP, 191-203.

[Hon 1978]
Hon, Robert, and Sequin, Carlo. A Guide to LSI Implementation. Xerox PARC (Palo Alto, September 1978).

[Johannsen 1978]
Johannsen, David L. "Our Machine: A Microcoded LSI Processor." Proceedings of MICRO-11 11th Annual Microprogramming Workshop (November 1978). SIGMICRO Newsletter 9, 4 (December 1978), 1-7.

[Knight 1974]
Knight, Tom. The CONS Microprocessor. Artificial Intelligence Working Paper 80, MIT (Cambridge, November 1974).

[Levin 1974]
Levin, Michael. Mathematical Logic for Computer Scientists. MIT Project MAC TR-131 (Cambridge, June 1974).

[LISP Machine 1977]
The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. LISP Machine Progress Report. AI Memo 444. MIT AI Lab (Cambridge, August 1977).

[McCarthy 1962]
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[McDermott 1974]
McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER Reference Manual. AI Memo 295a. MIT AI Lab (Cambridge, January 1974).

[Mead 1978]
Mead, Carver A., and Conway, Lynn A. Introduction to VLSI Systems (draft). (1978). To be published in 1979.

[Minsky 1963]
Minsky, M. L. A LISP garbage collector using serial secondary storage. Artificial Intelligence Memo No. 58 (revised), MIT (Cambridge, December 1963).

[Morris 1978]
Morris, F. Lockwood. "A Time- and Space-Efficient Garbage Compaction Algorithm." Comm. ACM 21, 8 (August 1978), 662-665.

[Saunders 1964]
Saunders, Robert A. "The LISP system for the Q-32 computer." In Berkeley and Bobrow, The Programming Language LISP, 220-231.

[Schorr 1967]
Schorr, H., and Waite, W. M. "An efficient machine-independent procedure for garbage collection in various list structures." Comm. ACM 10, 8 (August 1967), 501-506.

[Steele 1976a]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Steele 1976b]
Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Steele 1977]
Steele, Guy Lewis Jr. Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto. S.M. thesis. MIT (Cambridge, May 1977). operations.

[Steele 1978a]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME: A Dialect of LISP. MIT AI Memo 452 (Cambridge, January 1978).

[Steele 1978b]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). MIT AI Memo 453 (Cambridge, January 1978).

[Steele 1979]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. "Storage Management in a LISP-Based Processor." Proc. Caltech Conference on Very Large Scale Integration (Pasadena, January 1979).

[Sussman 1975]
Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).

[Wand 1977]
Wand, Mitchell. Continuation-Based Program Transformation Strategies Technical Report 61. Computer Science Department, Indiana University (Bloomington, March 1977).

[Wegbreit 1974]
Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 23-74. Center for Research in Computing Technology, Harvard U. (Cambridge, December 1974).

[Weinreb 1978]
Weinreb, Daniel, and Moon, David. LISP Machine Manual (Preliminary Version). MIT AI Lab (Cambridge, November 1978).