Computer-aided system design*

by E. DAVID CROCKETT, DAVID H. COPP, J. W. FRANDEEN, and CLIFFORD A. ISBERG

Computer Synectics, Incorporated
Santa Clara, California

PETER BRYANT and W. E. DICKINSON

IBM ASDD Laboratory
Los Gatos, California

and

MICHAEL R. PAIGE

University of Illinois
Urbana, Illinois

INTRODUCTION

This paper describes the Computer-Aided System Design (CASD) system, a proposed collection of computer programs to aid in the design of computers and similar devices. CASD is a unified system for design, encompassing high-level description of digital devices, simulation of the device functions, automatic translation of the description to detailed hardware (or other) specifications, and complete record-keeping support. The entire system may be on-line, and most day-to-day use of the system would be in conversational mode.

Typically, the design of digital devices requires a long effort by several groups of people working on different aspects of the problem. The CASD system would make a central collection of all the design information available through terminals to anyone working on the job. With conversational access to a central file, many alternative designs can be quickly evaluated, proven standard design modules can be selected, and the latest version of the design can be automatically documented. The designer works only with high-level descriptions, which reduce the number of trivial errors and ensure the use of standard design techniques.

From October, 1968, through December, 1969, the authors participated in a study at the IBM Advanced Systems Development Laboratory in Los Gatos, California, which defined the proposed CASD system and looked into the problems of building the various component programs. Details of several prototype programs which were implemented are given elsewhere.[1] There are no present plans to continue work in this area. This paper is essentially a feasibility report, describing the overall system structure and the reasons for choosing it. It includes descriptions of the data forms in the system and of the component programs, discussions of the overall approach, and an example of a device described in the CASD design language.

THE SYSTEM IN GENERAL

The (proposed) Computer-Aided System Design (CASD) system is a collection of programs to aid the computer designer in his daily work, and to coordinate record-keeping and documentation. It offers the designer five major facilities:

High-level description

The designer describes his device in a high-level, functional language resembling PL/I, but tailored to his special needs. This is the only description he enters into the system, and the one to which all subsequent modifications, etc., refer.

* This work was performed at the IBM Advanced Systems Development Laboratory, Los Gatos, California.

High-level simulation

An interpretive simulator allows the designer to check out his design at a functional level, before it is committed to hardware. The simulation is interactive, allowing the designer to "watch" his design work and evaluate precisely design alternatives.

Translation to logic specifications

The high-level design, after testing by simulation, is automatically translated to detailed logic specifications. These specifications may take a variety of forms, such as (1) input to conventional Design Automation (DA) systems, or (2) microcode for an existing machine.

On-line, conversational updating

The designer makes design changes and does his general day-to-day work at a terminal, in a conversational mode. Batch facilities are also available.

Complete file maintenance and documentation

Extensive record-keeping is provided to keep track of different machines, different designs of machines, different versions of designs, results of simulation runs, and so forth. High-level documentation of designs (analogous to that produced at lower levels by today's design automation systems) is a natural by-product of the CASD organization.

The CASD system can thus be viewed as an extension to higher levels of current systems for design, in roughly the same way that compilers are functional extensions of assemblers to higher levels.

The general organization of the system is pictured in Figure 1. The designer describes his device in a *source design language*, which is translated by a compiler-like program called the *encoder* to an *internal form*. The internal form is the input both to the high-level simulator (called the *interpreter*) and to a series of *translators* (two are shown in Figure 1) which convert it to the appropriate form of logic specifications. Different series of translators give different kinds of final output (e.g., one series for DA input, another series for microcode). The entire system is on-line, operating under control of the CASD monitor, which handles communication to and from the terminals. The user interface programs handle the direct "talking" to the user and invoke the proper functional programs.

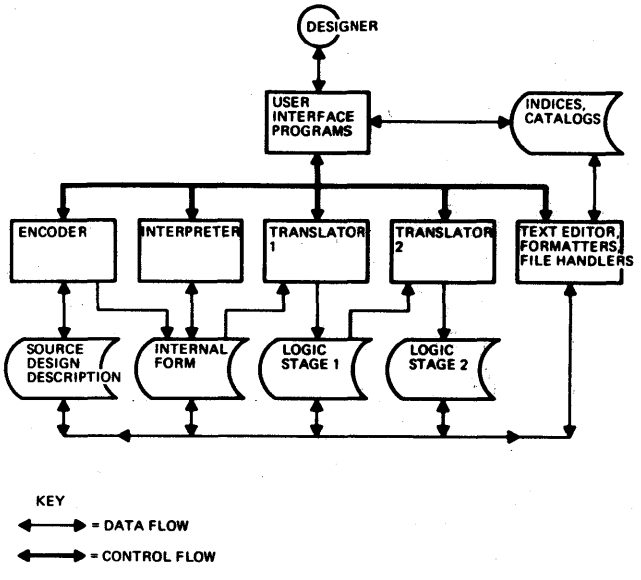

Figure 1—The CASD system

DATA FORMS IN THE CASD SYSTEM

Source design description

The CASD design language the designer uses is a variant of PL/I, stripped of features not needed for computer design and enriched with a few specialized features for such work. PL/I[2] and CASD's language[3] are described more fully elsewhere.

Procedures

The basic building block in a CASD description is the procedure. A procedure consists of: (1) declarations of the entities involved in the procedure, and (2) statements of what is to be done to these entities. A procedure is written as a PROCEDURE statement, followed by the declarations and statements, followed by a matching END statement, in the usual PL/I format:

```
PROC1:    PROCEDURE;

              declarations and statements

          END PROC1;
```

defines a procedure whose name is PROC1.

A procedure represents some logical module of the design, e.g., an adder. A complete design, in general, would have many such procedures, some nested within

others. The adder procedure, for example, may contain a half-adder as a subprocedure.

Data items

Each procedure operates on certain *data items*, such as registers or terminals. These items are defined by DECLARE statements, which have the general format:

DECLARE name attribute, attribute, . . .;

The name is used to refer to the item throughout the description. The attributes describe the item in more detail, and are of two types—logical and physical. Logical attributes describe the function of the item (it is bit storage, or a clock, say); physical attributes describe the form the item is to take in hardware (magnetic core, for example). Logical attributes influence the encoding, interpreting, and translating functions. Physical attributes, on the other hand, are ignored by the interpreter, giving a truly functional simulation.

Like any block-structured language, the CASD language has rules about local and global variables, and scope of names. These have been taken directly from the corresponding rules for PL/I.

Statements

The basic unit for describing what is to be done to the data items is the *expression*, defined as in PL/I but with some added Boolean operators, such as exclusive or (⅋), and some modifications to the bit string arithmetic.

The basic statement types for describing actions on data items are the assignment, WAIT, CALL, GO TO, IF, DO, and RETURN statements. These are basically as they are in PL/I, except as described below.

1. The assignment statement is extended to allow concatenated items to appear on the left-hand side. Thus:

   XREG || YREG:=ZREG;

   where XREG and YREG are 16 bits each and ZREG is 32 bits, means to put the high 16 bits of ZREG into XREG and the low 16 bits into YREG. In combination with the SUBSTR built-in function,[4] this assignment statement offers convenient ways to describe shifting and similar operations. The assignment symbol itself is the ALGOL ":=" rather than "=" as in PL/I.
2. The WAIT statement takes the form

   WAIT(expression);

   It thus differs from PL/I in that it allows one to specify a wait until an arbitrary expression is satisfied. This is useful for synchronizing tasks (see below).
3. The GO TO statement includes the facility of going to a label variable, and the label variable may be subscripted. This is useful for describing such operations as op-code decoding—for example: GO TO ROUTINE (OP).

Sequencing

The differences in motivation between CASD's language and PL/I are most evident in matters of sequence control and parallelism. PL/I, as a programming language, does not emphasize the use of parallelism. Programs are described and executed sequentially, which is not adequate for a design language.

The basic unit of work in CASD is the *node*. A node is a collection of actions which can be performed at the same time. For example, XREG:=YREG; and P:=Q; can be performed together if all the items involved are distinct. On the other hand, XREG:=YREG; ZREG:=XREG; cannot be performed (as written) at the same time, since the result of the first written operation is needed to do the second. The basic CASD rules are:

1. Operations are written as sequential statements.
2. However these operations are performed (sequentially or in parallel), the end results will be the same as the results of performing them sequentially.
3. Sequential statements will be combined into a single node (considered as being done in parallel) whenever this does not violate rule 2. That is, CASD assumes you mean parallel unless there's some "logical conflict."[5]

Of course, the designer may want to override rules 2 and 3. Another rule gives him one way to do this:

4. A labelled statement always begins a new node. Another way is by specifying parallelism explicitly. If the DO statement is written as DO CONCURRENTLY, all statements within the DO will be executed in parallel. Finally, the TASK option of the CALL statement makes it possible to set several tasks operating at once.

Preprocessor facilities

Some of the PL/I preprocessor facilities have been retained. These include the iterative %DO, which is particularly useful in describing repetitive operations, and the preprocessor assignment statement, useful for specifying word lengths, etc.

No defaults

Unlike PL/I, the CASD language follows the principle that nothing should be hidden from the designer. In particular, it has no default attributes, and everything must be declared. Similarly, it does not allow subscripted subscripts, subscripted parameters passed to subroutines, or anything else that might force the encoder to generate temporary registers not specified by the designer. Such restrictions might be relaxed in a later version, but we feel that until we have more experience with such systems, we had better hide as little as possible.

Internal form

Before the source description can be conveniently manipulated by other programs, it must be translated to an internal form. This form is designed to be convenient for both the translator programs and the interpreter. Compromises are necessary, of course—a computer program might be the most convenient form for simulation, but would be of no use at all to the translator.

The CASD internal form resembles the tabular structure used for intermediate results in compilers for programming languages. It consists of four kinds of tables: descriptors, expressions, statements and nodes.

The descriptor table records the nature of each item (taken from its DECLARE statement). The entries are organized according to the block structure of the source description and the scope-of-names rules of the language.

The expression table contains reverse Polish forms of all expressions in the source description, with names replaced by pointers to descriptors. Each expression appears only onee in the expression table, although it may appear often in the source description. In effect, the expression table lists the combinational logic the translator must generate.

The statement table consists of one entry for each statement in the source description, with expressions replaced by pointers to entries in the expression table, and a coded format for the rest of the statement (statement type plus parameters).

The node table tells which statements in the statement table belong in the same node, and the order in which various nodes should be executed.

The internal form has thus extracted three things from the source description—data items, actions to be taken on those items, and the timing of the actions—and recorded them in three separate tables—the descriptor, the statement, and the node tables. The expression table is added for convenience.

Simulation results

The high-level simulation involves three forms of data: values of the variables, control information, and run statistics.

Before a simulation run begins, the variables of the source design description (corresponding to registers, etc.) must be assigned initial values. One way to do this is with the INITIAL attribute in the DECLARE statement, which makes initialization of the variables at execution time a fundamental part of the description. Sometimes, though, the designer may want to test a special case, and simulate his design starting from some special set of initial values. CASD permits him to store one or more sets of initial values in his files; and for a given simulation run, to specify the set of initial values to be used. In this way, he can augment or override the INITIAL attribute.

At the end of a simulation run, the final values of the variables may be saved and used for print-outs, statistics gathering, or as initial values for the next simulation run. That is, a simulation run may continue where the last one left off.

The high-level, interpretive simulation in CASD is perhaps most useful because of its control options. As an interpreter, operating from a static, tabular description of the device, the CASD simulator can give the user unusually complete control over the running of the simulation. Through a terminal, he can at any time tell the system which variables to trace, how many nodes to interpret at a time, when to stop the simulation (e.g., stop if XREG ever gets bigger than 4 and display the results), and so forth. These control conditions may be saved just as the data values may be, and a simulation run may use either old or new control conditions.

Permanent records of a simulation also include summaries of run statistics (the number of subprocedure calls, number of waits, etc.).

Translator output

Different translators produce different kinds of output. Assembly-language level listings of mircocode might be needed for some lower-level systems, the coded equivalent of ALD sheets for others. Typically, output would include error and warning messages.

File structure

In an on-line, conversational system, it is particularly important that the working data be easily accessible to the user and the control language seem natural to him. CASD attempts to facilitate user control in two ways: through the user interface programs, and the structure of the data files.

The basic organizational unit in the CASD files is called the *design*. A design consists of all the data pertinent to the development of some given device. A design may have many *versions*, representing current alternatives or successive revisions. Each version has some or all of the basic forms of data associated with it: source description, internal form, simulation results, translator output, and so on.

Two catalogs, one for designs and one for versions, are the basic access points to CASD data. A typical entry in the design catalog (a *design record*) contains a list of pointers to the version descriptors for each version of every design in the system. The version descriptor contains pointers to each of the various forms of data for that version (source description, . . .) plus control information telling which set of translators has been applied to the design in this version, and so on.

These descriptors give the user interface programs efficient access to needed data. For example, if the user asks to translate a given design, the interface finds the version descriptor, and can then tell if the design has been encoded, and if not, inform the user and request the input parameters for encoding.

PROGRAMS IN THE CASD SYSTEM

CASD monitor and support programs

All the CASD component programs are under control of a monitor program, which provides the basic services for communicating with terminals and allocates system resources. In the prototype version[6] the environment was OS/360 MVT, and it was convenient to set up the monitor as a single job, attaching one subtask for each CASD terminal. The CASD files were all in one large data set, and access to them was controlled by service routines in the monitor. The monitor also controlled the allocation of CPU time to various CASD terminals within the overall CASD job. This approach makes it easier to manage the various interrelated data forms within the versions, and would probably work in environments other than OS/360 as well.

Besides the monitor and the data access routines, the support programs include a text-editing routine to use in editing the source description.

User interface programs

CASD system control is not specified in some general language. Rather, each CASD function has its own interface *program*, which has the complete facilities of the system available to it.

The design records and version descriptors give precisely the information needed by user interface programs. A typical user interface program might be one for encoding and simulating a source design description already in the CASD files. The version descriptor shows, for example, whether or not the source description has already been encoded. The interface may then give the user a message like "Last week you ran this design for 400 nodes. Should the results of that run be used as initial values for this run?" The point is that the conversation is natural to the task at hand. The tasks under consideration are well defined, and each natural combination of them has its own interface program.

Encoder

Since the CASD encoder is roughly the first half of a compiler, it may be built along pretty standard lines. Care must be taken only in providing some sort of conversational compilation facility. Conversational interaction is an important part of the CASD approach to design, and some sort of line-by-line feedback is required. Similarly, since modification is so common in design work, recompilation must be as efficient as possible. Incremental compilation—translating each source statement as far as possible on input, independently of other statements—is one answer. Then only those statements which have changed since the last compilation need be recompiled. The approach used in the CASD prototype is described elsewhere.[7,8]

Interpreter

The basic unit that the interpreter simulates is the node table, the various statements which comprise

the node are identified. These statments are then "executed" in two steps: First, all the expressions in the statements are evaluated; second, the results are stored. By this two-step procedure, the parallelism inherent in the definition of the node is correctly simulated.

The interpreter steps from node to node, as they appear in the node table, with several exceptions. One is the conditional branch, where some (usually just one) statment within the node must be evaluated or executed to determine what the next node should be. Another exception is when wait, halt, or trace conditions have been met. Such "values" as "stop if this item is referenced" may be stored with the item's descriptor in the internal form. If this kind of condition is encountered in a node, the interpreter takes the action indicated before going to the next node. Control conditions like these may be altered dynamically by the user, who may, when a "halt" condition is satisfied, not only observe the variables and their values, but alter the control conditions.

Translators

The translator used in the prototype system converts the internal form to a list structure of the machine logic. Techniques for translating from this to DA input or actual circuits for any given circuit family are straightforward. The elements of the list structure are: hash cells, part cells, subexpression cells, assignment cells, action cells, condition cells, and clock cells. Hash (as in "hash code") cells contain index entries and cross-references to the rest of the cells. Part cells contain all the information declared about each item; subexpression cells indicate how the various items are to be combined to form circuits. Assignment cells tell what data is to be transferred to where. Action cells and condition cells are lists of which actions (e.g., assignments) are to be taken and under which conditions. Clock cells contain labels and other information about sequencing. Most of the information in these cells comes fairly directly from the appropriate tables in the internal form, but the translator links the cells in a way that corresponds to the hardware that must be generated. For example, all assignments to a given register are linked together, and this might correspond (for a particular circuit family) to a single storage bus.

Essentially, the translator reduces the high-level description to a form which currently known procedures[9,10,11,12] can handle, by breaking up the information in the internal form and linking it up again in several different ways. Details of the various linking schemes and how they relate to the source description are given elsewhere.[13]

Other programs

The general structure of the CASD system is flexible enough to permit addition of other programs. A few possibilities have been considered.

One obvious drawback of interpretive simulation is the overhead. Simulation by compilation to machine code would be perhaps 50 or 100 times as fast. This is a significant difference on long runs, after the design is basically checked out (e.g., runs to get firm performance figures).

A generalized assembler program to prepare program input to the interpreter would allow larger quantities of software to be tested by "running" it on the machine being simulated.

Cost-estimating programs operating directly from the internal form would give quick-and-dirty estimates without going through the entire hardware translation process. Translation from the internal form to microcode is another possible extension.

COMMENTS

History

Others—most notably Gorman and Anderson,[14] Schorr,[15] Franke,[16] Duley and Dietmeyer,[17,18] Friedman and Yang,[20] and Metze and Seshu[21]—have described languages and systems for logic translation or simulation, and occasionally for both. Typically, in logic translation systems, the design is described in a special-purpose procedural language similar to programming languages. The description is usually at a lower level than in CASD and is translated to Boolean equations, or some similar form, by programs written for the purpose.

In most simulation systems, on the other hand, designs are described in some high-level, general-purpose language—either a general simulation language, or an existing programming language augmented with timing subroutines and the like. The description is translated by an existing compiler to a program which performs the simulation.

There is good reason for this difference. Until recently, no existing programming or simulation language was really adequate to describe logic, and no general-purpose simulation system was so deficient as to justify creating a special system for simulating computer

designs. But the advantages of integrating logic translation and simulation into the same system outweigh these factors, in our judgment.

Integration of the two functions is achieved in CASD by translating a single, high-level, special-purpose language to a common internal form, providing input to both logic translation programs and an interpretive simulator. The interpretive simulation is also a key point in making the system on-line.

Another innovation in CASD is the way in which descriptions incorporate timing. Timing is included rather explicitly in typical existing languages. At lower levels, every statement or action is accompanied by an indication of when it is to take place (at which clock pulse, say). At higher levels, actions are simply recorded sequentially, with some indication of how long they take and what resources they require. (Simulators operating from these descriptions usually construct "future events" lists, ordered by increasing time of occurrence, and simulate whichever event is on top of the list at the moment.)

Timing in the CASD descriptions is based on the use of asynchronous design as proposed by Metze and Seshu.[22] Multiple tasks are synchronized by using shared variables and referring to them with WAIT statements. This approach has several advantages. Asynchronous design at the functional level, as offered by the CASD system, allows reasonable hardware independence, since synchronizing conditions refer to elements of the functional design rather than to its physical implementation. (An asychronous description may, of course, be implemented in either synchronous or asynchronous logic circuits.) Perhaps most important, especially for an on-line system, is that the PL/I multitasking scheme, from which the CASD timing approach is derived, and techniques like DO CONCURRENTLY make it possible to describe timing relationships in a quick and natural manner.

Advantages of an on-line system

Conventional design work is slowed by turn-around time (in the model shop as well as in the computation center) and an elaborate hierarchy of system architects, engineers, and technicians. One result is that few alternatives are considered in designing a system, and fewer still are evaluated in any systematic way. The CASD system bypasses these limitations by putting the designer directly in touch with a design system by a terminal, having the system take over many of the bookkeeping functions of design, and giving him immediate feedback at each stage of the design process. Immediate feedback is important in:

a. Encoding, where descriptions are entered line by line, and syntax is checked immediately, allowing immediate correction and modification.
b. Simulation, in which the designer may "converse" with the system as his design is simulated. He may change control conditions as the simulation progresses, look at values of data items, and so forth.
c. Selection of different translation procedures based on the results of simulation, cost estimating programs, or other translations.

Except for (a), these could be done with a batch system, of course, but they are much more effective in an on-line environment. Suppose, for example, that a design for a computer is stored in the system, and it contains special hardware for floating point operations. The designer wants to know just what difference it would make if he eliminated this hardware and did all floating point operations with programmed subroutines. With the CASD text-editing programs, the designer would remove the description of the hardware for floating point, and change the floating point operation code descriptions to trap these operations to a specified location. He would re-encode the description and correct any errors. By simulating and translating both this new description and the old one, he would obtain precise figures on the exact difference in hardware and running time. An on-line system can reduce this complicated maneuver to a one-day job.

Advantages of an integrated system

Most of the advantages of integrating all aspects of design in a single system can be summed up in one word: control. Consider how important it is that the simulation model accurately reflect the hardware that is being built. Under the CASD system, this is automatic: the design description is the simulation model.

A necessary part of the design process is low-level checking of logic circuits both for logical correctness and for race and hazard conditions. In CASD, the system always uses proven methods. Besides reducing the necessary tests, this controlled logic synthesis ensures the use of standard techniques and building blocks. Different optimality criteria can be used and the results compared. For example, the different effects of restricting the logic to one chip type, or allowing more freedom, might be compared. Criteria such as these are often more important than minimizing the

total number of circuits; and under the CASD system, the correct criteria can be enforecd.

A good design must be reliable and allow ready diagnosis of problems that occur. The CASD controlled synthesis ensures that the resulting logic is diagnosable. Indeed, the required diagnostic tests can be produced as an integral part of the translation process by at least one method.[23] It is easy to see how translators could be made to produce either duplicated logic, triple-modular-redundant logic, or unduplicated logic (say) if the designer wants to compare their relative costs.

Finally, the advantages of a unified file system, providing documentation automatically, are fairly clear. Accurate, consistent, up-to-date documentation may be the most important single feature of the CASD system.

EXAMPLE

This section contains an example of a computer described in the CASD design language. The computer and the way it is described have been chosen to illustrate the features of the language, rather than for any intrinsic merit. The computer is a simple binary machine called SYSTEM/0. It contains 65,536 32-bit words of memory and 16 general-purpose and index registers called XREG(0) through XREG(15). XREG (0) always contains all zeros. It may be stored, tested, and shifted, but not altered.

The instructions of SYSTEM/0 are one word (32 bits) long. The first 8 bits contain the operation code. The next 8 bits contain two four-bit fields, the M (for modifier) and X (for index register) specifications. The last 16 bits are used for an address.

The following instructions are described in the following CASD description:

ST	M,X,ADDR	Store the contents of XREG(M) into memory location [ADDR+ contents of XREG(X)].
CLA	M,X,ADDR	Load the contents of memory location [ADDR+contents of XREG(X)] into XREG(M). M may not equal zero.
BC	M,X,ADDR	Branch to location [ADDR+ contents of XREG(X)] if and only if the contents of XREG (M) is zero. (Since XREG(0) is always zero, BC 0, X,ADDR is an unconditional branch.)
RR	M,X,ADDR	Rotate XREG(M) right [contents of XREG(X)+ADDR] places. The number of places to rotate is always assumed to be modulo 32.
BAL	M,X,ADDR	Branch and Link to location [ADDR+contents of XREG(X)] storing the return address (= next location) in the low-order 16 bits of XREG(M), setting the high-order 16 bits of XREG(M) to zero. M may not equal zero.
SIO	M,X,ADDR	Start an input-output operation on device number [ADDR + contents of XREG(X)]. The M field specifies which input-output operation is to be performed.

Figure 2 shows the data flow the designer might expect CASD to generate, after entering the functional description given in Figures 3 through 7. (The order of the figures is for illustration only. The designer need have only a shadowy outline of the data flow in mind at the time he prepares his functional description.)

Figures 3 through 7 are annotated to highlight interesting features of the CASD language. Also note that there are a few places where the designer did choose to dictate the data flow. For example, the only link to the XREG's is constrained to be through the Y register by specifying Y:=MSDATA; XREG(M):= Y; rather than just XREG(M):=MSDATA; . So, the designer can exercise as much or as little direct influence on the final data flow as he chooses.

ACKNOWLEDGMENTS

We wish to thank George T. Robinson and Dr. Eugene E. Lindstrom for their guidance and advice.

REFERENCES

1 E D CROCKETT et al
Computer-aided system design
Advanced Systems Development Division IBM Corporation Los Gatos California Technical Report ⁂16.198 1970
2 *IBM System/360 PL/I reference manual*
IBM Corporation White Plains New York Form C28-8201
3 CROCKETT Appendix A
4 IBM Corporation page 237
5 CROCKETT Appendix G
6 IBID Appendix H
7 IBID Appendices I, J, K

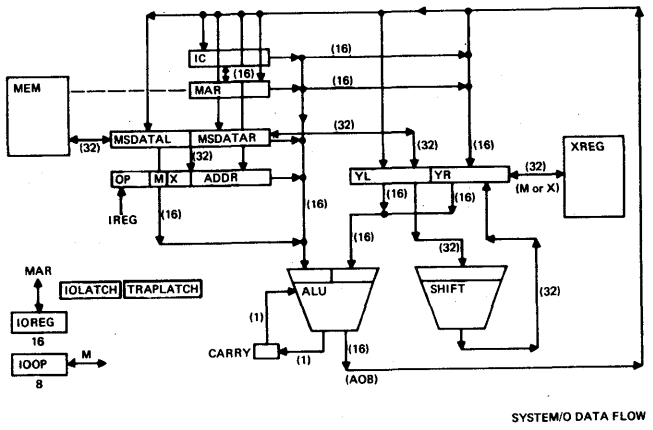

Figure 2—Flow of data in SYSTEM/0

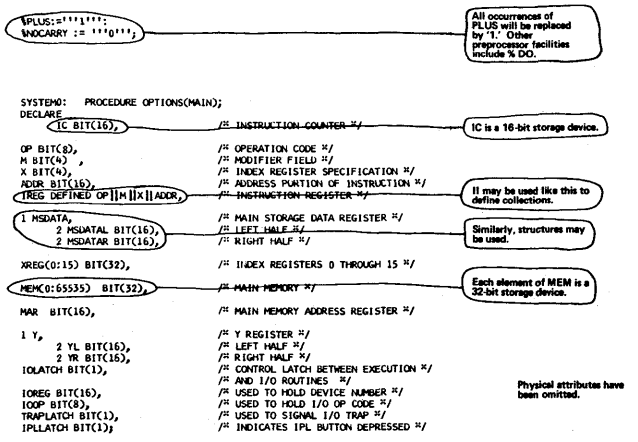

Figure 3—CASD description of SYSTEM/0, page 1

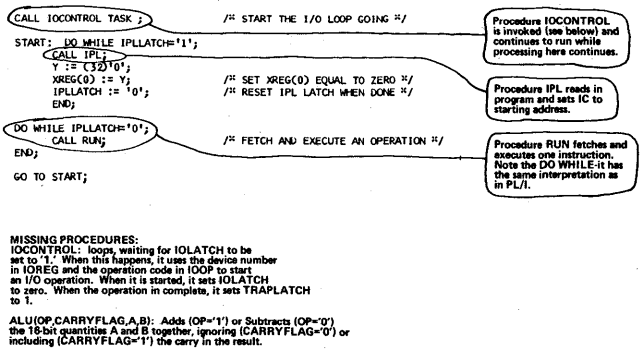

Figure 4—CASD description of SYSTEM/0, page 2

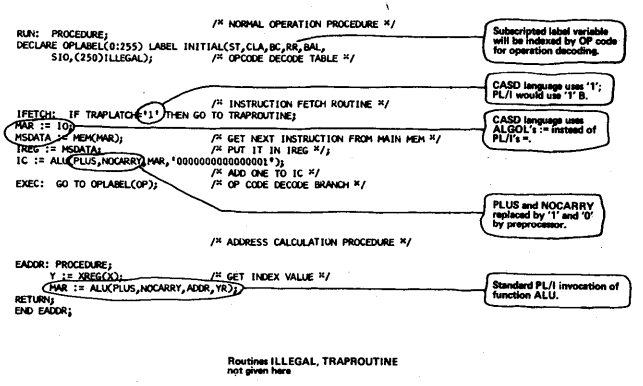

Figure 5—CASD description of SYSTEM/0, page 3

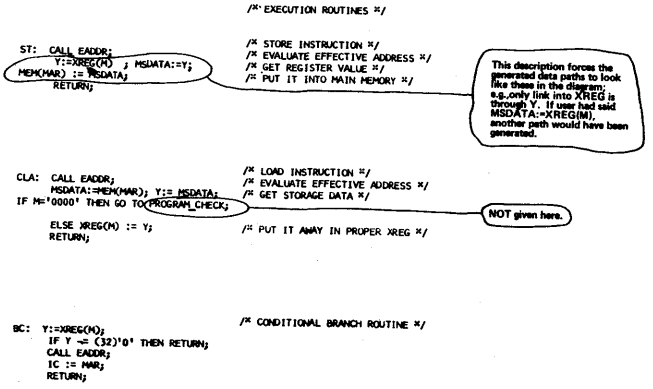

Figure 6—CASD description of SYSTEM/0, page 4

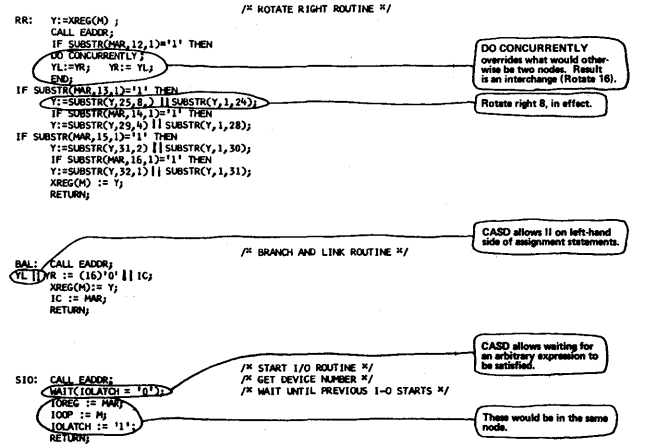

Figure 7—CASD description of SYSTEM/0, page 5

8 P BRYANT
A note on designing incremental compilers
Submitted to CACM August 1970
9 J R DULEY D L DIETMEYER
Translation of a DDL digital system specification to Boolean equations
IEEE Transactions Vol C-18 April 1969
10 T D FRIEDMAN
ALERT: a program to produce logic designs from preliminary machine descriptions
Research Division IBM Corporation Yorktown Heights New York Research Report No RC-1578 March 1966
11 T D FRIEDMAN S C YANG
Methods used in an automatic logic design generator (ALERT)
Research Division IBM Corporation Yorktown Heights New York Research Report No RC-2226 October 1968
12 D F GORMAN J P ANDERSON
A logic design translator
Proceedings AFIPS Fall Joint Computer Conference 1962
13 CROCKETT Appendix M
14 GORMAN and ANDERSON
15 H SCHORR
Computer-aided digital system design and analysis using a register-transfer language
IEEE Transactions Vol EC-13 December 1964
16 E A FRANKE
Computer-aided functional design of digital systems
IEEE Southwestern Conference Record April 1968
17 DULEY and DIETMEYER *op cit*
18 J R DULEY D L DIETMEYER
A digital system design language (DDL)
IEEE Transactions Vol C-17 September 1968
19 FRIEDMAN
20 FRIEDMAN and YANG
21 G METZE S SESHU
A proposal for a computer compiler
Proceedings AFIPS Spring Joint Computer Conference 1966
22 IBID
23 CROCKETT Appendix N