CASL - A Language for Automating the Implementation of Computer Architectures *

by

Gregory F. Maxey and Elliott I. Organick

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

UUCS - 79 - 111

*This research is sponsored in part by the Burroughs Corporation.

CASL - A Language for Automating the Implementation of Computer Architectures*

by

Gregory F. Maxey and Elliott I. Organick

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

The Computer Architecture Specification Language (CASL), described in this paper, is intended for use by computer architects as a Design Automation tool for experimenting with new architectures. CASL is a state machine description language especially useful for describing digital systems at the "register transfer" level and designed to meet the needs of the computer architect as a design and documentation medium.

A machine described in CASL may be decomposed into cooperating Modules, each representing an asynchronous finite state machine. Each Module consists of an Abstractions, Structure, and Procedure section. An architect may use the Abstractions section to define his own data representations and primitive operations. The Structure section describes structural elements (combinatorial and sequential hardware "building blocks") and connections (explicit specifications of each data path). The Procedure section is a textual (nearly ALGOL-like) representation of the state transition graph and the sets of control signals issued concurrently in each state to drive the structure. Statements in each state are collateral, rather than sequential, and CASE structures are used for conditional selection of control signals to be issued by a state. A macro facility is provided to encode groups of control signals.

Our current effort centers on compiling CASL into microcoded interpreters for the Burroughs B1700/B1800. We foresee research aimed at compiling CASL into certain VLSI hardware implementations.

*This research is sponsored in part by the Burroughs Corporation.

1. Introduction

Our long range goal is the development of a system that accepts a description of a digital machine's architecture and implements it (i.e., its representation) automatically. No constraints are placed on the nature of the implementation. It may be soft, like a microprogram, or hard, like a VLSI chip.

If our efforts are successful, computer architects should be able to experiment with new architectural features as easily as programmers now experiment with programming language features. In principle, the architect can evaluate a design by "exercising" the machine's executable microcoded representation or, if implemented in VLSI, the machine itself.

As in software development where short turn-around time is always a key factor, the potential benefits of the planned system will be realized only if its use greatly reduces the time and effort required to implement, test, and refine the machine design. To achieve this, it is crucial that the source language of this system is a convenient and natural means of conveying the architect's view of the structure and behavior of his machine.

Certain earlier efforts which led to the design of computer description languages are the points of departure for the language design reported in this paper. These earlier efforts (and we make no claim at having a comprehensive view) have appeared to us as a bottom-up progression whose steps focus on successively higher system levels, reflecting the increasing complexity of computer systems. Chu's CDL (2,3), perhaps the earliest of the "register transfer" level languages, is aimed primarily at the logic designer's point of view, while higher in the progression, Bell and Newell's ISPS and PMS languages (4,5,6) cater more to the view of the system architect. In the latter, however, certain parts of the system may be described more formally than are other parts (5). Our objective is to provide a language in which architectural-level formal specification of a complete system organization is feasible and convenient.

We see two essential design principles for a useful architecture description language.

1. Provide the user
   - (a) a language whose primitive representational concepts are as close as possible to those with which the architect originally formulated the machine architecture. The architect should not have a sense of "translating" the machine design into a relatively "foreign" language.
   - (b) constructs that provide expressive power for describing appropriate detail.
   - (c) logical segmentation of the job of machine description, so that anyone using the language is guided to concentrate on one manageable aspect of the machine at a time. (e.g., the separation of a machine description into structural and procedural parts.)
   - (d) constructs that allow the control of complexity through abstraction mechanisms (modularization and suppression of detail).
   - (e) a language that is as natural for the documentation of architectural ideas as it is for input to a language processor.

2. Assure that the characteristics of the language will not frustrate automatic implementation, by
   (a) omitting constructs that might be used in combination with certain other language constructs to express ill-defined systems, leading to impossible or unpredictable situations (e.g. description of an intentional "race condition").
   (b) minimizing the amount of inference required of the language processor. The language should not permit description of structure or behavior that is syntactically correct but architecturally vague, since this could require the implementation system to be "intelligent" enough to complete the machine design.

Since a good language for describing computer architecture is essential to the success of the automatic implementation system, the design of the Computer Architecture Specification Language (CASL) is the first major focus of our research, and the primary subject of this paper.

1.1 Overview of the paper

We present the main concepts of CASL in Section 2, and we show the logical separation of computer descriptions into ABSTRACTIONS (Section 2.1), STRUCTURE (Section 2.2), and PROCEDURE (Section 2.3). Section 3 introduces the CASL Module, which is useful not only for modularization and creation of levels of abstraction in a machine design, but also as a mechanism for specifying additional concurrency. In Section 4, we explain how Input/Output is expressed in CASL. Section 5 is an example of a piece of digital hardware described in CASL and shows particular instances of many of the features discussed in Section 2. Finally, Sections 6 and 7 report on our current research, future research plans, and conclusions we have been able to draw from the work so far.

2. The Computer Architecture Specification Language

CASL is a "register transfer" level language that, like ISPS, is intended for system architects. However, CASL allows one to specify more architectural detail than instruction set description languages like ISPS. CASL's universe includes such components as registers, stacks, memories, data transformation elements, and control elements, along with cables to interconnect them, packaging to partition them into modules, and even simple I/O devices to communicate with them.

2.1 The ABSTRACTIONS section

Since data representations and primitive operations used in new computer architectures may vary widely, CASL lets the architect describe his machine in terms of its own primitives. CASL adapts to the architect's ideas. The architect tailors CASL to his machine by making various definitions in the ABSTRACTIONS section. CASL has no "native" character set, no "native" integer representation, and not even a "native" kind of bit, (although defaults such as '0' and '1' for bit values, etc., are provided).

Examples of CASL flexibility:

1. One can specify the number of possible values of a "bit", and then name and order these values. For example, a "bit" for use in three-valued machine logic would be specified as having three ordered possible values with one-character names such as '0', '1', and '2', or 'i', 'j', and 'k'.

2. Nearly any integer arithmetic may be built in, e.g., 3's complement arithmetic, by providing the parameters that describe that arithmetic. Thereafter, constants and arithmetic operations may be specified as components in that arithmetic.
3. A machine design may use some specified non-standard set of character codes.
4. One can invent new logical operations (as may be needed, for instance, with multivalued logic) and define symbols to represent them.

2.2 The STRUCTURE section

As in some other computer hardware description languages, there is an important distinction between description of the structural and procedural aspects of a computer. However, the distinction is more pronounced in CASL. The STRUCTURE section of a CASL description completely describes the data-handling structure of the computer. This includes not only data storage elements, but also data transformation elements and data paths. The PROCEDURE section which follows is concerned only with expressing the actions of the controller which activates the already-specified structure.

Elements

The STRUCTURE section is divided into two subsections: ELEMENTS and CONNECTIONS. The ELEMENTS subsection specifies structural elements like registers (including subfields), stacks, random access memories, and "storage buses". In CASL, "operators" which transform data (combinational logic) are also specified as structural elements.

The structural elements available in CASL are a set of pre-selected types of parts whose pre-defined behavior is that of real pieces of hardware. The characteristics of, for example, a random access memory are pre-defined except for the word size, number of words, width of the memory address input, and integer representation of memory addresses. The pre-defined properties of CASL structural elements include "electrical" details such as relative timing (e.g., registers are characterized as being composed of Master/Slave flip-flops, although their exact technology such as TTL or MOS is never specified). In this way, the system architect using CASL may safely ignore low-level details such as "race" conditions and fan-out constraints. We observe that the pre-defined nature of structural elements applies to operators to the extent of defining their "electrical" properties, without placing any prior constraints on what arithmetic and logical functions individual operators may be specified to perform in a CASL machine description.

Connections

CASL emphasizes the importance of providing explicit and formal specification of all data path connections between pairs of structural elements. In the CONNECTIONS subsection, one not only specifies the existence of all data paths, but also any special features that certain data paths may have. For example, if sign-extension is needed in data transfers from a 12-bit register to a 16-bit register, it is usually performed in the data path by connecting the highest-order line of the path to the five higher-order inputs of the 16-bit register. Of course, CASL doesn't require the architect to give a wire-by-wire statement of the special connections of this data path; in this case it is sufficient to indicate that the data carried by this path have a particular arithmetic type.

The CONNECTIONS subsection is an important feature because:

(1) It leads to clearer thinking for the user; he specifies the connections explicitly in a special section devoted only to that purpose, rather than implicitly in the PROCEDURE section.

(2) It is the best place to specify interconnections that have special properties.

(3) It lets the user treat the connections as the important part of the machine design that it is (especially with growing significance in VLSI of interconnections relative to gates), and

(4) The language processor can check that the statements in the PROCEDURE section are consistent with the information supplied in the CONNECTIONS subsection.

2.3 PROCEDURE section

The control mechanism of the computer is described in the PROCEDURE section as a finite state machine (FSM), or as a collection of related state machines. Statements in the PROCEDURE section fall into four categories:

(1) ENABLE statements, which enable actions in the data-handling structure of this Module,

(2) Statements for controlling FSM's in subordinate Modules, if any,

(3) Conditional statements, and

(4) NEXT-STATE statements, which specify the state the FSM will enter for the next "state time".

CASL's FSM "timing" protocols are modeled after those of the Algorighms State Machine (ASM) Charts invented by Clare and Osborne (7,8). Indeed, we may regard the PROCEDURE section as a textual equivalent of an ASM chart. For a particular state time of the FSM, all of the actions to be performed are enabled concurrently. Hence the statements in a state description are collateral rather than sequential. Sequential dependencies are expressed in terms of state sequencing in the FSM.

Absence of explicit clocking

In CASL, state transitions are asynchronous. There are no built-in clocks, because it is easier to impose synchronous operation on a naturally asynchronous system than to try to make a naturally synchronous system cope with the demands of asynchronous operation. (We all recognize, of course, that "synchronous" hardware is really asynchronous until we artificially define regular time frames using clocks). The true significance of this asynchronous operation does not become apparent until we have multiple FSM's active concurrently.

3. Modules

A computer or other digital system may be described as a hierarchical network of CASL Modules. There are several (usually interrelated) reasons for describing a system in this way.

1. Modularity permits the functional decomposition of large systems. Breaking up the description into Modules can make the design easier to think about (and therefore easier to create and debug).

2. Descriptions can often be made more compact through the use of modularity. There is a direct analogy to conventional programming languages, where certain frequently duplicated pieces of code are taken out-of-line and made into procedures.
3. The CASL Module provides an additional mechanism for specifying concurrency. Often the lack of restrictions in the sequencing of actions is best described by multiple FSM's operating concurrently. Each FSM is associated with its own Module.
4. CASL Modules can describe how the hardware of a system is actually divided into physical modules (e.g., a computer system that has physically independent CPU's that communicate).

While modularity, per se, is not a unique feature among computer description languages, this architecturally faithful approach to modularity is unusual in a language above the logic design level. Languages that we have seen at the system architecture level use procedures or functions (as in conventional programming languages) to approach the issue of modularity. CASL's architecturally realistic Modules make it useful for accurate description of actual hardware modularization.

As always in CASL, the characteristics of the input and output ports of Modules, along with the timing protocols of CASL FSM's, assure that there will be no low-level timing problems with inter-Module communications.

3.1 Description

In Section 2, we described the division of a machine description into

ABSTRACTIONS, STRUCTURE, and PROCEDURE sections, without mentioning Modules. In a multi-Modular description, each Module is a complete machine description. Each Module has its own STRUCTURE section and usually also has its own PROCEDURE section. Thus there can be more than one FSM in operation at the same time, introducing "global" concurrency in CASL, as distinct from the "local" concurrency within the states of individual FSM's. (Each Module may also have its own ABSTRACTIONS section.)

Module descriptions are never nested within other Module descriptions. Also, except for its input and output ports, a Module description is completely independent of where it will be instantiated. An important implication of this is that a Module's FSM cannot activate any structural elements of other Modules.

3.2 Instantiation

A Module is instantiated as a structural element within the description of another Module. It can be instantiated many times and in many different environments throughout a computer description. The Module instance is just an ordinary structural element to the Module that contains it; the containing Module has no access to any of the structural elements inside the Module instance. All communication with the Module instance is through its input and output ports.

3.3 Activation

There are statements that can be used in the PROCEDURE section of a Module to activate, and examine the status of, the FSM of any Module that is instantiated within it. There is also a statement that allows an FSM to stop itself.

Alternatively, a Module can be instantiated in such a way that its FSM will be activated automatically as soon as the FSM of the containing Module is activated, using no statements in the PROCEDURE section.

4. Expressing Input/Output

One problem with computer description languages has been that they do not usually provide any means for expressing I/O. Implementations generated from computer descriptions have either been unsuitable for running realistic programs, or require the use of an auxiliary, implementation-dependent, language for this purpose (e.g. TRW's PL/1-like SMITE language supplemented by assembly language to do I/O (9)). CASL not only breaks this tradition by providing I/O facilities, but does so with a notation that is consistent with the rest of the language. Thus, we do not insert ALGOL-like READ and WRITE statements, disrupting the computer description. Instead, the PERIPHERAL_DEVICE structural element may be specified as a unit record device, a tape drive, or an addressable device like a disk, drum or bubble memory. Data can be transferred to or from the device by enabling the data paths that connect the device to the rest of the computer's structure.

The PERIPHERAL_DEVICE specification statement is not intended as a general I/O device description facility, and so has limited descriptive capability. There are many specific models of I/O devices that cannot be described precisely. Therefore, it is perhaps more correct to say that this statement specifies certain properties of an interface that has an I/O device plugged into it. In other words, one can specify a disk drive, for example, even if one cannot always directly specify a particular type of disk drive.

5. Illustrative example of a CASL machine description

Rather than describe a toy computer, we have chosen to describe in this report an actual LSI circuit, the Signetics 8X02 Control Store Sequencer (Figure 1). While the 8X02 illustration does have the advantages that it is straight forward to understand, and it is real, there is also a slight disadvantage. CASL is really intended for describing computer architecture, not necessarily general digital hardware; there are a few aspects the 8X02 circuit that are slightly outside of the intended scope of CASL. Therefore, our illustration should sharpen our understanding of CASL's expressibility limits. In a subsequent paper, we plan to supply an example CASL description of a machine design for which CASL is more directly appropriate.

The example supplied here includes a Module description that is a complete 8X02 (Figure 2), and excerpts from another module where an 8X02 is instantiated. We consider the 8X02 description first. The Module heading in Figure 2 gives the type name of the Module ('SIGNETICS_8X02') which will be used in other modules when specifying instances of this Module. The heading also includes the names and widths of this Module's input and output ports. Note that the inputs do not include V_{cc} and GND, since these have to do with electrical rather than logical specification. Also, we do not include an enable ($\overline{EN}$) input. We explain later why this line is not treated as an input port.

In the ABSTRACTIONS section we have used all three of the optional subsections. In the DATA_REPRESENTATIONS subsection we define an arithmetic type, named 'UBIN', unsigned binary arithmetic, which is how addresses are represented in the 8X02. Next, in the OPERATION_DEFINITIONS subsection we define the symbol '$+' to mean addition in UBIN arithmetic. The statements in the SYMBOLIC_DEFINITIONS subsection merely give names to some constants.

In the ELEMENTS subsection (of the STRUCTURE section) there are just three elements.

	CASL structural element	Signetics block(s) of Figure 1b
(a)	addr_reg	ADDRESS REGISTER
(b)	addr_stack	STACK REGISTER FILE / STACK POINTER (2)
(c)	incr	+1, +2 LOGIC

Typing addr_stack as a stack 10 bits wide and 4 deep allows us to subsume the two blocks of the Signetics diagram. Also note that our description of the incr operator includes names for its input and output ports ('current_addr' and 'next_addr' respectively, each 10 bits wide.) The two operations which incr can perform are named 'by_1' and 'by_2', each accompanied by its definition. Note the use of '$+' unsigned binary addition; and the constants '1' and '2' which are each typed as unsigned binary values.

In the CONNECTIONS subsection, each data path is stated. Since each one is of the same length, ten bits, as indicated in Figure 1b, the CASL paths in this example do not need to be typed or otherwise qualified (as might be the case where truncation or sign extensions occur on a path.)

The PROCEDURE section begins with the CONTROL_COMBINATIONS subsection. Here we define the macro 'BUMP_ADDR_REG', which, when used in the STATES subsection, will be expanded into the three statements shown.

In the STATES subsection we define the states named 'AWAIT_CLK_LOW', 'AWAIT_CLK_HIGH', 'EXECUTE', and 'GATE_OUTPUT' and their interrelationship (alternatively expressed in the graph of Figure 3).

AWAIT_CLK_LOW is the starting state for the four-state FSM. The sequence of the two states, AWAIT_CLK_LOW followed by AWAIT_CLK_HIGH, detects the level transitions on the clk input port that correspond to the 8X02 triggering on the rising edge of the clock pulse. Each of these states contains a single statement which selects the state for the next state time. The choice of which state is the next is performed by a decode expression, which is similar to a case statement except that it takes the place of a value. The value selected by these decode expressions depends on the data found on the clk input port.

In the EXECUTE state a case statement is used to decode the opcode that is input to the input port named 'ac'. Note the use of the opcode mnemonics defined in the ABSTRACTIONS section and used here as selectors in the case statement. Note also the examples of ENABLE statements. For example,

```
ENABLE addr_reg TO incr.current_addr;
```

enables the data pathway (previously defined in the CONNECTIONS subsection) that runs from addr_reg to the current_addr port of the operator incr The next ENABLE statement,

```
ENABLE incr.by_2;
```

enables the operator incr to perform the by_2 operation.

It is not necessary to include separate ENABLE statments that push or pop the stack. This is because statements that enable a data pathway to or from addr_stack also cause pushing or popping of the stack (also, the stack can be popped without gating the data to any destination).

After completion of the EXECUTE state, the FSM goes into the state named GATE_OUTPUT. This seemingly extra state is necessary because the value that is gated to the input of addr_reg in the EXECUTE state is not instantly available at the output of the register, as some time is needed by the register's flip-flops to assume their new states. The new value appears on the register's outputs only at the very end of the state time of the EXECUTE state.

The other part of the example (Figures 4 and 5) sketches a Module that contains an instance of the 8X02 (marked by the arrow in the left margin of Figure 5). The containing Module's STRUCTURE section shows the specification of some element types that were not demonstrated in Figure 2, the 8X02 module. For example, we see the use of a subregister (part of the register named 'control_word'), a random access memory, and of course the 8X02 Module as a structural element. The STRUCTURE shows how the 8X02 might be connected into a containing system.

Returning to the specification of the 8X02 itself we can point out some of the more subtle differences between the block diagram description (Figure 1) and our CASL description (Figure 2).

1. <u>Absence of multiplexers in CASL STRUCTURE section</u>. In Figure 1b there are two multiplexer blocks, yet none is explicitly represented in the CASL description. In CASL, the fact that more than one data path leads to the same register as a destination implies the existence of a multiplexer at the gate level (assuming that tri-state hardware is not used); but multiplexers are not given as structural elements. We do not wish to require the architect to specify (explicitly) the codes to be sent to multiplexers to select among their input. This detail is to be considered part of the controller rather than part of the data-handling structure. This information is stated in control terms in the PROCEDURE section, and it is not that difficult for the language processor to translate it from this form to codes suitable for controlling multiplexers (and to provide appropriate multiplexers whereever multiple data paths have the same destination).

2. No distinction of tri-state logic. The output of the 8X02 is specified (Figure 1a) as tri-state logic. The $\overline{\text{EN}}$ input line (Figure 1b) is used to select whether the output is allowed to be connected to a 0,1 level or is in a high impedance state. Although there is no specification facility in CASL to differentiate between tri-state outputs and ordinary two-state outputs, the definition of CASL data paths is that they affect their data destinations only when they are enabled. Thus, instead of taking the view that there are implicit multiplexers wherever needed, as we did above, it is equally valid to say that there are no multiplexers anywhere, and every data path is tri-state (held in high-impedance unless it is enabled). Therefore, no $\overline{\text{EN}}$ input is shown in the STRUCTURE section of our CASL description, because the question of enabling or not enabling a data path is a control issue rather than a data-handling structure issue.

3. Clocking. Dealing with the 8X02's explicit clk input (Figure 1b) is somewhat below the intended descriptive level of CASL. The 8X02 actually contains no FSM. It is composed of storage elements and operators, activated by a transition on the clk input. CASL is not concerned with low-level description that would be required to specify the details of clocking of structural elements. We simulated this detail by adding an FSM to watch the clock and activate the structure on a rising clock transition. State transitions in our FSM take place asynchronously, and we observe the clk input to determine which state to select as the next one. An FSM description is insensitive to the uniformity of the intervals and dur-

ations of clock pulses. Our attitude, therefore, is simply to assume that the FSM's asynchronous state transitions occur fast enough so that no signals sent to the FSM on the *clk* input port are ever "missed". (In actual clocked electronics there is a corresponding requirement that the response of the circuitry determines the speed of the clock and the pulse duration.)

6. Current Use and Future Efforts

The design of CASL is currently frozen so it will be stable until the first automatic implementation system can be completed. This will permit us to get some experience using CASL, and will give us a good idea of the relative strengths and weaknesses of the language.

The opportunity to use CASL has been limited so far to those involved in its design, and that use only as a notation since the automatic implementation is not yet complete. In the Spring of 1979 students at the University of Utah will begin to experiment with CASL as a documentation medium for their hardware designs of machines to solve the "Eight Queens" problem (11).

Work is proceeding on a CASL compiler that will convert computer descriptions into microcoded emulators for the Burroughs B1700/B1800. It would be very gratifying if future research involving CASL could move in the direction of producing actual VLSI circuitry from CASL descriptions.

Future work on CASL design will probably include attempts to improve its extensibility. The pre-selected set of structural elements is sometimes cumbersome when a machine designer has in mind a structural element that is both a storage element and a transformation element, like a counter or shift register, and uses it in many places in his machine. Some combination of the existing constructs for storage element specification and operator specification should yield a construct that will allow extensions to the set of CASL structural elements.

CASL is, and needs to be, very flexible. However, it is obvious that there are limits to CASL's ability to adapt to the primitives of various machine designs. Some future effort is likely to be devoted to extending these limits. In particular, it is difficult (or impossible) to provide a descriptive scheme that parameterizes every possible form of arithmetic. Not only have we chosen to omit real arithmetic from the present arithmetic description statement, but we have also restricted the facility to certain classes of arithmetic (e.g., modular arithmetic can not be described, at present). Additional development in this area would probably be very useful.

There are other new features that may prove useful. We may want a means for representing elapsed time, for such diverse purposes as performance evaluation and providing a means for an I/O interrupt to occur at a "reasonable" time after I/O has been initiated. We may also wish to generalize the PERIPHERAL_DEVICE to describe, in detail, the special characteristics of actual devices (e.g., disk track sizes).

Finally, long range future plans for CASL (other than generation of VLSI circuitry that we mentioned) must be concerned with

(1) Verification of CASL computer descriptions,

(2) Performance studies of CASL-described architectures, and

(3) The possible use of CASL as a "base" upon which to build an auto-

mated system that actually performs the architectural design of machines from very-high-level specifications. We have yet to determine how this relates to the language design efforts begun by Teichroew and others (ISDOS Problem Statement Language (PSL) (10)). Perhaps that language might serve as a point of departure for a language useful in an automated design and implementation system.

7. Conclusions

At the present time, CASL appears to meet most of our needs well. It seems to be a natural medium for computer description. While the language is not very terse, we think it is easy to use; so computer descriptions can be written rapidly. The syntax of CASL looks familiar enough that it is relatively easy to read. At the same time, CASL is suitable for processing by a compiler.

There are unusual (and even unique) concepts and features in CASL that represent a step forward in architectural-level description language:

1. CASL provides complete representation of data-handling structure, including operators and hardware modules, rather than just storage elements.
2. CASL provides explicit description of all data pathways.
3. The FSM representation of control expresses both concurrency and sequentiality in a natural way.
4. Modularity is treated in an architecturally faithful way, and is

recognized as the mechanism for specifying "global" concurrency.

5. CASL is a protean language that adapts to the data representation and data transformation primitives of each machine design.
6. CASL has an I/O specification facility that is consistent with the rest of the language.

The successful design effort on CASL has taken us a long way toward our goal. It paves the way for the completion of the automatic implementation system.

In this paper we have presented the concepts embodied in CASL, rather than the details of CASL syntax. The complete details of the language can be found in the dissertation on this research (1).

8. References

1. Maxey, G.F., "The Computer Architecture Specification Language" (tentative title) Ph.D. Dissertation, University of Utah, (work in progress).

2. Chu, Y.,"Computer Organization and Microprogramming", Prentice-Hall, 1972

3. Chu, Y., "An ALGOL-like Computer Design Language", Comm. of the ACM, Oct. 1965, pp. 607-615.

4. Barbacci, M., Barnes, G., Cattell, R., Siewiorek, D., "The Symbolic Manipulation of Computer Descriptions; The ISPS Computer Description Language." Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, March, 1978.

5. Barbacci, M.R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Transactions on Computers, C-24 #2, Feb. 1975, pp 137-150.

6. Bell, C.G., Mudge, J.C., and McNamara, J.E.,"Computer Engineering: A DEC View of Hardware Systems Design", Digital Press, Digital Equipment Corp., 1978.

7. Clare, C.R.,"Designing Logic Systems Using State Machines," McGraw-Hill, 1973

8. Clare, C.R., and Osborne, T.E., "Design of Small Machines (The Algorithmic State Machine)", in AFIPS Workshop on the Influence of Programming Languages on Computer Systems Architecture, May 1971, pp 46-62.

9. Press, B.,"SMITE Language Specification", TRW, Redondo Beach, Cal., 1975.

10. Koch, R.F., Krohn, M.J., McGrew, P.W., and Sibley, E.H., "PSL Version 2 Release 1: A PSL Language Primer", ISDOS Working Paper #33, ISDOS Research Project, Department of Industrial Engineering, University of Michigan, August 1970, (Revised May 1971).

11. Wirth, N., "Algorithms + Data Structures = Programs", Prentice-Hall, Englewood Cliffs, N.J., 1976.

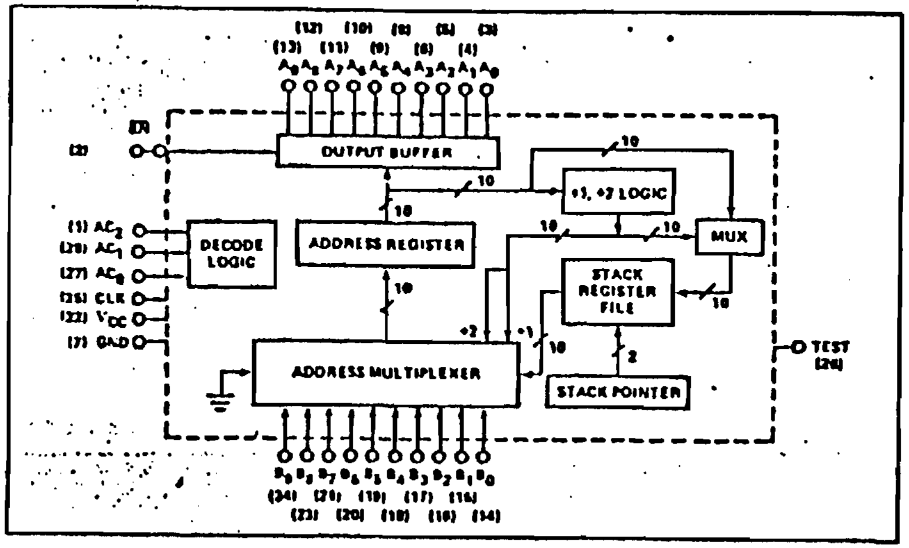
OBJECTIVE SPECIFICATION

FEATURES

- Low-power Schottky process
- 50ns cycle time (TYP)
- 1024 microinstruction addressability
- N-way branch
- 4-level stack register file (LIFO type)
- Automatic push/pop stack operation
- "Test & skip" operation on test input line
- 3-bit command code
- → Tri-state buffered outputs
- Auto-reset to address 0 during power-up
- Conditional branching, pop stack, & push stack
- → Positive edge trigger (low-to-high transition)

(a)

BLOCK DIAGRAM



(b)

Figure 1

FUNCTIONAL DESCRIPTION

The following is a description of each of the eight Next Address Control Functions (AC_2 - AC_0)

MNEMONIC	FUNCTION DESCRIPTION
TSK	AC_{2-0}-000: TEST & SKIP Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address=Current Address +1; Stack Pointer unchanged If test is TRUE (HIGH): Next Address=Current Address +2 (i.e. Skip next microinstruction); Stack Pointer unchanged
INC	AC_{2-0}-001: INCREMENT Next Adress=Current Address + 1 Stack Pointer unchanged
BLT	AC_{2-0}=-010: BRANCH TO LOOP IF TEST CONDITION TRUE. Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address=Current Address+1; Stack Pointer decremented by 1 If test is TRUE (HIGH): Next Address=Address from Stack Register File (POP); Stack Pointer decremented by 1
POP	AC_{2-0}=011: POP STACK Next Address=Address from Stack Register File (POP) Stack Pointer decremented by 1
BSR	AC_{2-0}= 100: BRANCH TO SUBROUTINE IF TEST CONDITION TRUE. Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address=Current Address +1; Stack Pointer unchanged If test is TRUE (HIGH): Next Address=Branch Address Input (B_{0-9}); Stack Pointer incremented by 1; PUSH (write) Current Address +1→Stack Register File
PLP	AC_{2-0}= 101: PUSH FOR LOOPING Next Address=Current Address +1 Stack Pointer incremented by 1 PUSH (write) Current Address→Stack Register File
BRT	AC_{2-0}= 110: BRANCH ON TEST CONDITION TRUE Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address=Current Address +1; Stack Pointer unchanged If test is TRUE (HIGH): Next Address=Branch Address Input (B_{0-9}); Stack Pointer unchanged
RST	AC_{2-0}= 111: RESET TO ZERO Next Address=0 Stack Pointer unchanged

(c)

```
**********************************************************************
*
*  This CASL module is a description of the Signetics 8XØ2 control
*  store sequencer (shown in Figure 1).
*
*

MODULE
    type(signetics_8XØ2)
          input (a̅c̅(3 bits), b(1Ø bits), test(1 bit), clk(1 bit))
          output (a(1Ø bits)).


ABSTRACTIONS:

    DATA_REPRESENTATIONS:
       ubin : arithmetic (base(2), unsigned).

    OPERATION_DEFINITIONS:
       $+ : '+'ubin.

    SYMBOL_DEFINITIONS:
*   Op-code mnemonics.
       tsk  : 'ØØØ'bit.
       inc  : 'ØØ1'bit.
       blt  : 'Ø1Ø'bit.
       popj : 'Ø11'bit.
       bsr  : '1ØØ'bit.
       plp  : '1Ø1'bit.
       brt  : '11Ø'bit.
       rst  : '111'bit.
*   Names for CLK levels.
       low  : 'Ø'bit.
       high : '1'bit.


STRUCTURE:

    ELEMENTS:

       addr_reg register (1Ø bits).

       addr_stack stack (1Ø bits by 4).

       incr operator
          input (current_addr(1Ø bits))
          output (next_addr(1Ø bits))

          by_1 (next_addr <- current_addr $+ '1'ubin)

          by_2 (next_addr <- current_addr $+ '2'ubin).


    CONNECTIONS:

       addr_reg to  incr.current_addr, addr_stack, a.
       b to addr_reg.
       addr_stack to addr_req.
       incr.next_addr to  addr_reg, addr_stack.
```

```
PROCEDURE:

    CONTROL_COMBINATIONS:

bump_addr_reg
       enable addr_reg to incr.current_addr;
       enable incr.by_1;
       enable incr.next_addr to addr_reg.


    STATES:

await_clk_low
       next_state is  decode (clk, (low  : await_clk_high,
                                    else : await_clk_low)).

await_clk_high
       next_state is  decode (clk, (high : execute,
                                    else : await_clk_high)).

execute
       next_state is gate_output;
       case ac of
*  "Test and Skip (if test input true)" op.
          tsk: do;
                 enable addr_reg to incr.current_addr;
                 case test of
                   'l'bit : do;
                               enable incr.by_2;
                            end;
                    else  : do;
                               enable incr.by_1;
                            end;
                 enable incr.next_addr to addr_reg;
              end;
*  "Increment" op.
          inc: do;
                 bump_addr_reg;
              end;
*  "Branch to Loop if Test input true" op.
          blt: do;
                 case test of
                    'l'bit : do;
                                enable addr_stack to addr_reg;
                             end;
                     else  : do;
                                enable pop(addr_stack);
                                bump_addr_reg;
                             end;
              end;
*  "Pop stack" op.
          popj: do;
                  enable addr_stack to addr_reg;
               end;
```

Figure 2 (continued).

```
*   "Branch to Subroutine if test input true" op.
          bsr: do;
                 enable addr_reg to incr.current_addr;
                 enable incr.by_1;
                 case test of
                    '1'bit : do;
                               enable incr.next_addr to addr_stack;
                               enable b to addr_reg;
                            end;
                     else   : do;
                               enable incr.next_addr to addr_reg;
                            end;
              end;
*   "Push for Looping" op.
          plp: do;
                 enable addr_reg to addr_stack;
                 bump_addr_reg;
              end;
*   "Branch if Test input true" op.
          brt: do;
                 case test of
                    '1'bit : do;
                               enable b to addr_reg;
                            end;
                     else   : do;
                               bump_addr_reg;
                            end;
              end;
*   "Reset address register to zero" op.
          rst: do;
                 enable 'Ø'ubin to addr_reg;
              end;
*   The "else" part of this case will never be used.
         else: .


gate_output
       enable addr_reg to a;
       next_state is await_clk_low.
```

Figure 2 (continued).

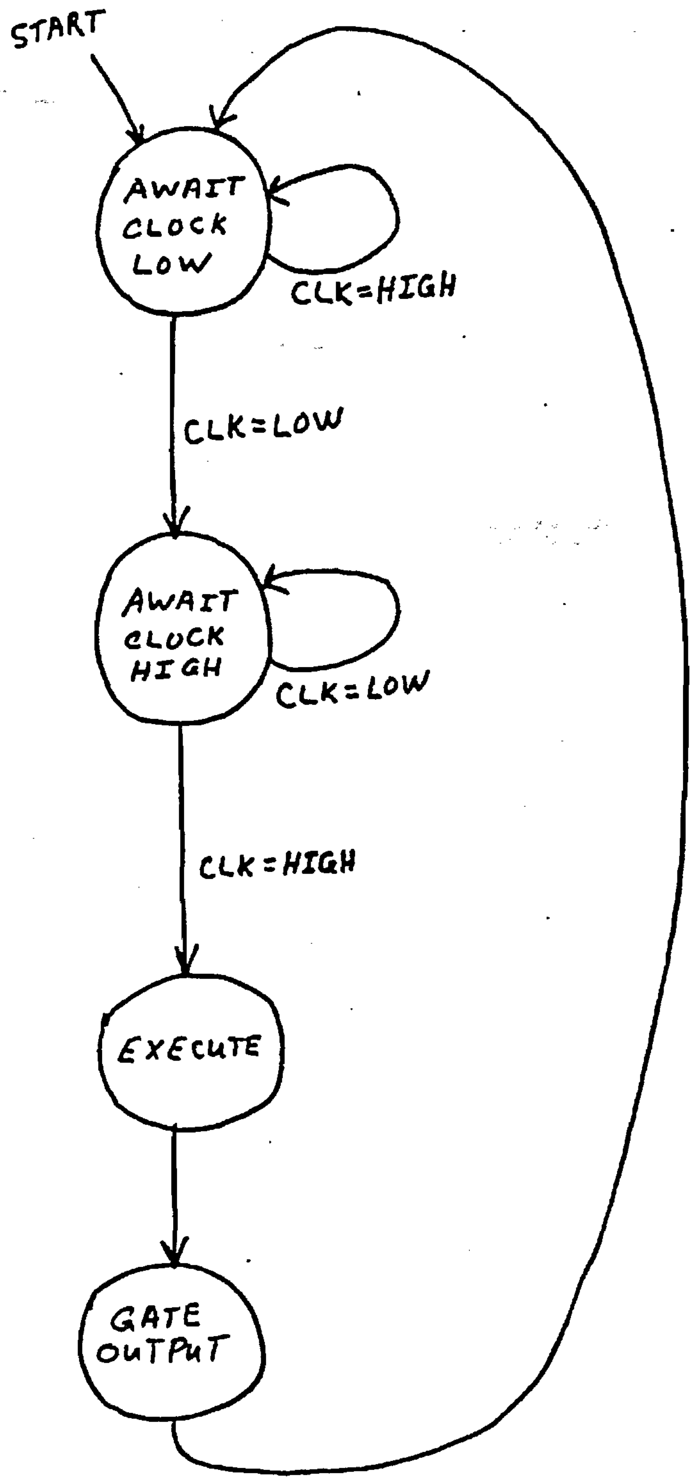


Figure 3. State Transition Diagram of the 4-state FSM

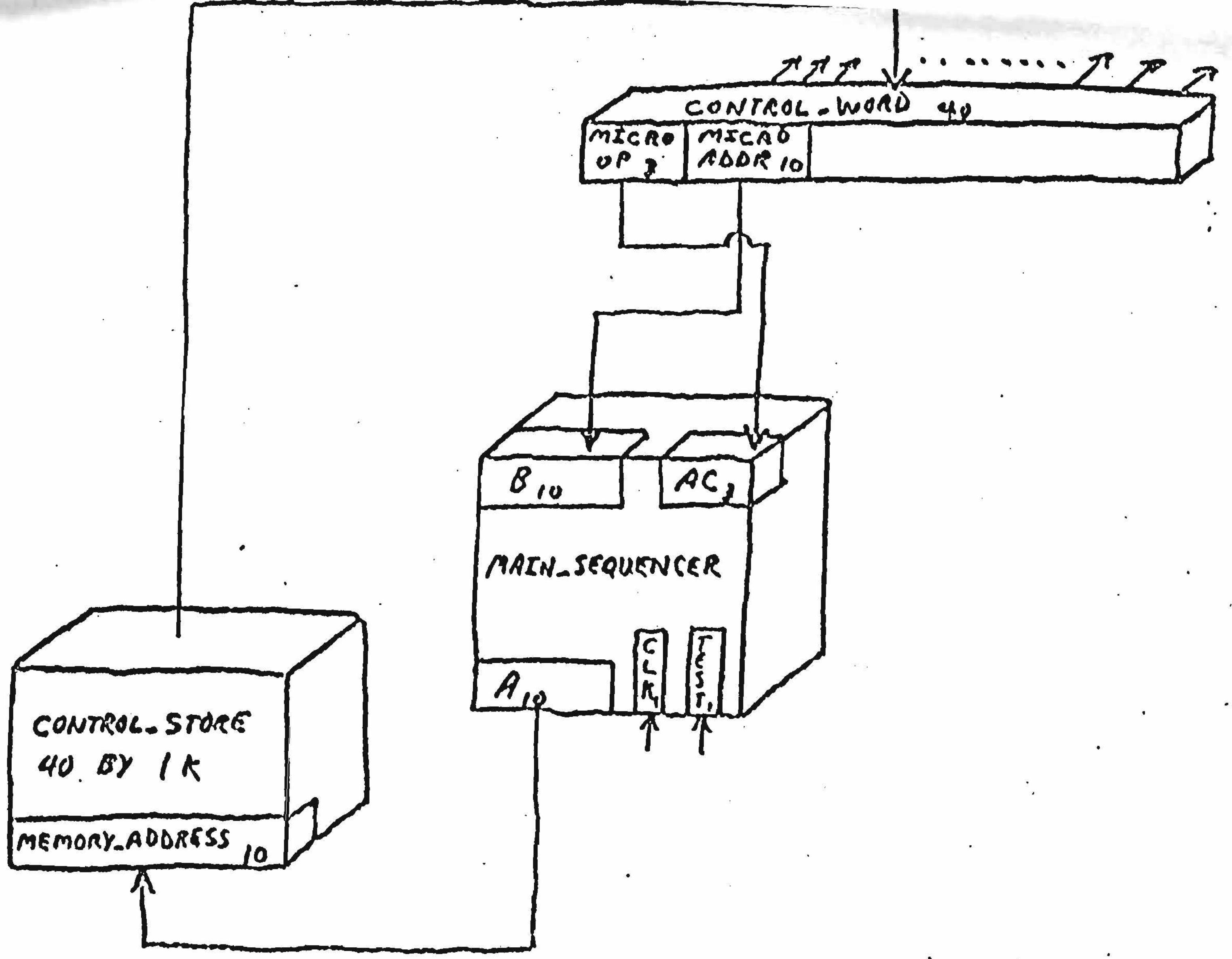

CONTROL_WORD 40
MICRO OP 3
MICRO ADDR 10
B 10
AC 3
MAIN_SEQUENCER
A 10
CLK 1
TEST 1
CONTROL_STORE
40 BY 1K
MEMORY_ADDRESS 10

Figure 4.

```
**************************************************************************
*
*   The following excerpts are from a CASL module that is a description
*   of a machine (shown in Figure 4) that contains a Signetics 8XØ2
*   Control Store Sequencer.
*
*

                 .
                 .
                 .

STRUCTURE:

    ELEMENTS:

                 .
                 .
                 .

       control_word register (4Ø bits).

          micro_op : control_word (bit Ø, 3 bits).
          micro_addr : control_word (bit 3, 1Ø bits).
          control_bits : control_word (bit 3, 37 bits).
          control_bits_valid : micro_op (bit Ø, 1 bit).

       control_store memory (4Ø bits by 1k),
          memory_address (1Ø bits) binary_addr.

       main_sequencer module  type(signetics_8XØ2)
          input (ac(3 bits), b(1Ø bits), test(1 bit), clk(1 bit))
          output (a(1Ø bits)).

                 .
                 .
                 .

    CONNECTIONS:

                 .
                 .
                 .

       main_sequencer.a to control_store.memory_address.
       control_store to control_word.
       micro_op to  main_sequencer.ac.
       micro_addr to  main_sequencer.b.

                 .
                 .
                 .
```

Figure 5.