1969

A design oriented digital design language

David Michael Rouse

Department:

A DESIGN ORIENTED DIGITAL DESIGN LANGUAGE

BY

DAVID MICHAEL ROUSE,

A

THESIS

submitted to the faculty of

THE UNIVERSITY OF MISSOURI-ROLLA

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Rolla, Missouri

1969

Approved by

James H. Tracy (advisor)

L. A. Eggenberger

Charles Hatfield

ABSTRACT

A digital design language is presented here which is more consistent with the design sequence of digital computers than existing languages. An ideal design sequence is first investigated and the following desirable design langauge characteristics obtained. A good design oriented language must be: 1) multi-level, 2) capable of expressing ideas easily, 3) easily understood, 4) machine acceptable, 5) modular and, 6) capable of showing timing and control. It should also be: 1) independent of technology, 2) unrestricted to any particular structural feature such as serial processes, synchronous processes, etc., 3) concise, 4) precise, and 5) non-ambiguous.

With regard to these features, the language presented here has a marked improvement over most of the other languages in that it is 1) multi-leveled, 2) modular, 3) capable of showing timing and control clearly, 4) unrestricted to any particular structural features, and 5) is easily understandable.

A flow chart based language is used to make the language more easily understood since it separates the control and operation variables into more appropriate and distinct categories. Multi-level specification is used not only to make the design more readily understood, but also as a means

of making the design language more consistent with the design procedure. This language is very versatile in representing all types of designs from completely synchronous to completely asynchronous in either serial or parallel operation.

Since this language is closely related to, and enhances flow table representation and can be used to express asynchronous operations, it is of significant value in bridging the now existing gap between digital system design and asynchronous sequential switching theory.

The multi-level structuring of the language makes simulation and fault diagnosis easier on both the logic level and the functional level. This is due to the partitioning techniques of the language.

The initial phase of the design of a large digital computer is presented using this language to show how it makes larger systems more easily understandable and to show its consistency with the design procedure.

ACKNOWLEDGEMENTS

The author wishes to express his appreciation to Dr. James H. Tracey for his careful guidance throughout the entire project.

The author also wishes to thank Dr. Steven A. Szygenda and Richard L. Christensen for stimulating discussions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

At present computer design is started by a mental conception, transformed to a narrative type of description, carried on in a pseudo-isolationist atmosphere by sets of disjoint sequences and finished by experience, ingenuity, trial and error and a lot of perseverance spread out over an unnecessarily long period of time. The great fault and burden of this sequence is not having a suitable means of expressing and communicating design ideas regardless of the phase of the design. It is for this reason that a more design oriented language is desired.

The features of a design oriented language can be seen in the ideal design sequence as is presented by Breuer[9]. Breuer divides the sequence of digital design into three areas: preconstructional analysis, design and implementation, and software. These are subdivided pictorially as indicated in Fig. 1. From this drawing it is inferred that one phase is consistent and carried out from the preceding phase. Thus, it is desirable to have a language which can be used in the initial phase of the design and also be built upon as the design progresses.

To have a design language which can be used in the initial design phase and also as the design progresses means

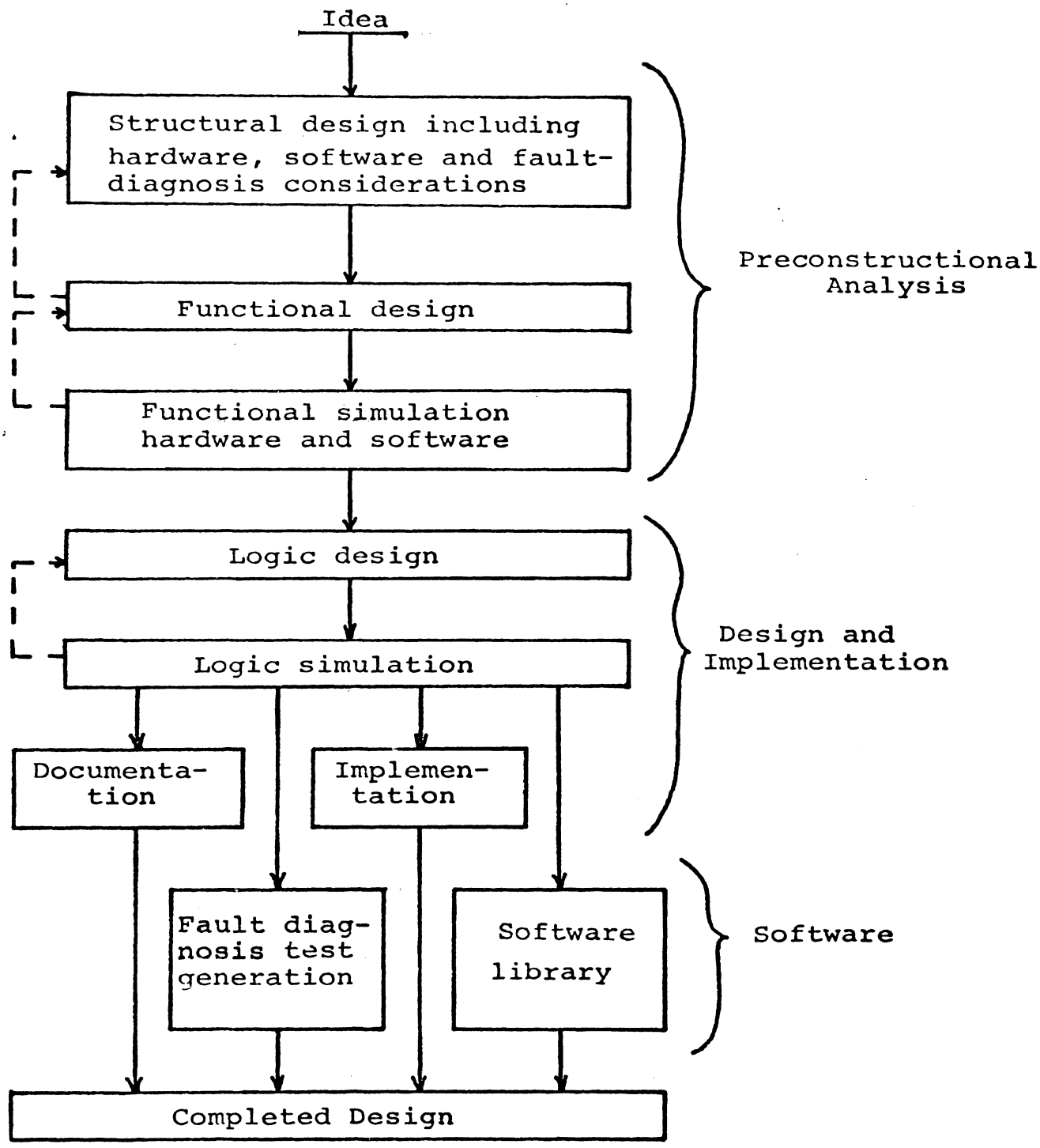


Fig. 1 Digital Design Procedure

that the language must be capable of expressing the structural phase of the design as well as the functional phase without loss of clarity of structure. To do this a high-level language would be ideal for expressing the structural phase and for functional design, a low-level description which is compatible with the high-level description is needed. These two descriptions along with the logic design layout should be capable of giving a complete documentation of how the system is to operate.

It is almost inevitable that the language will meet with automated design procedures. For this reason the language should either be acceptable as input to the computer or easily converted to something that is acceptable. This input could either be graphical or character strings. Although it is quite desirable to have a design language which can also be used as a programming language, there are other requirements which have higher priority such as understanding on all levels.

A design must go through both functional and logical simulation. Therefore one would expect the language to be such that functional and logical simulation would be enhanced by the language, if not a direct take-off from the language. Due to the increasing size of digital systems, logical simulation of complete systems is totally unrealistic under present logical simulation techniques. Therefore, a language which would express more clearly the necessary

requirements for a functionally controlled logic simulation would be of extreme importance in the design of large systems.

A high-level language which easily depicts modular structures would be useful in fault diagnosis since modular diagnosis would effectively reduce the size of the unit being diagnosed.

To make the design task easier,the design language should be such that the conversion from functional design to logical design is easily obtained. Yet the language must not be so close to hardware that changing technology would soon make the language obsolete.

In summary, a good design oriented language must be: (1) multi-level, (2) capable of expressing ideas easily, (3) easily understood, (4) machine acceptable, (5) modular and, (6) capable of showing timing and control. It should also be: (1) independent of technology, (2) unrestricted to any particular structural features such as serial processes, synchronous processes, etc., 3) concise, 4) precise, and 5) non-ambiguous.

A review of the existing languages will now be presented. It is not the intention to present the existing languages so that the reader will be proficient in their use, but to summarize the desirable and undesirable features of the language with respect to the characteristics listed in the previous paragraph.

An example problem of a design of a small digital computer will be used to help depict the language and to show some of its characteristics[13]. The example problem as presented in its original form will be slightly modified to make it easily expressible with the language being used.

II. REVIEW OF EXISTING LANGUAGES

A. Computer Design Language (C.D.L.)

Computer Design Language is an ALGOL-like language developed by Y. Chu for computer design and documentation[7]. Fig. 2 gives the specification of the example computer in Computer Design Language. Computer Design Language uses ALGOL-like statements to declare registers, control signals, memories, etc. Then it specifies all data transfers as well as the signals used to control these data transfers. One can see from Fig. 2 that the control variables are easily noticed since they are separated from the transfer. Because of this the timing is easily expressed if it is of a synchronous nature. Since there is no restriction that the control variables be mutually exclusive, parallel operations can be expressed. But this is often hard to follow since one must check every statement condition to see which is true. Since the reader has no hint as to the order or sequence in which they become true, following through a design often becomes a very tedious and time consuming task for large machines. Because there is no explicit indications of the timing or control sequence, it is not immediately obvious to the reader.

```
REGISTER,  R(0-10), F(0-3), A(0-10), C(0-5), G(0), D(0-5)
SUBREGISTER, R(OP) = R(0-2), R(ADDR) = R(3-10), F(I) = F(1-3)
MEMORY,  M(C) = M(0-77, 0-10)
DECODER,  K(0-17) = F
SWITCH,  POWER (ON,OFF), START (ON,OFF)
CLOCK,  P

START*P   :  if POWER = ON then F←17, G←0
      P   :  if START = ON then G←1
K(17)*P   :  if G=0 then (C←0, D←0)
             if G=1 then (F←6)
K(06)*P   :  R←M(C), D←D count +1
             if G=0 then (F←17)
             if G=1 then (F←12)
K(12)*P   :  F(I)←R(OP), C←R(ADDR), F(0)←0
K(00)*P   :  R←M(C), F←10
K(10)*P   :  A←A add R, F←13
K(01)*P   :  R←M(C), F←11
K(11)*P   :  A←A sub R, F←13
K(04)*P   :  D←R (ADDR), F←13
K(02)*P   :  if A(0)=0 then (F←13
             if A(0)=1 then (D←R(ADDR), F←13)
K(03)*P   :  M(C)←A, F←13
K(05)*P   :  if (C(3)=1) then (F←17, G←0)
             if (C(2)=1) then (F←16)
             if (C(1)=1) then (F←15)
             if (C(0)=1) then (F←14)
K(13)*P   :  C←D, F←6
K(14)*P   :  A←1 shr A, F←13
K(15)*P   :  A←1 cirl A, F←13
K(16)*P   :  A←0, F←13
```

Fig. 2 An Example of Computer Design Language

The language is fairly concise, precise and non-ambiguous on the statement level.

Computer Design Language also has the ability to express high-level actions like hardware subroutines with the use of the "do" statement. This is the way that system structure is expressed.

A desirable feature of Computer Design Language is that it is about the right distance from hardware. It is independent of technology to a great extent but yet is close enough to hardware so that there is an easy conversion from Computer Design Language to logic design. Chu is experimenting with a translator that will accept as inputs a machine description in Computer Design Language and will translate this description into a set of Boolean Equations.

In comparing Computer Design Language with the desirable features of a design oriented language, it can be seen that Computer Design Language has many of these features such as preciseness, conciseness and non-ambignity on the statement level. It also can be used to express high level actions, show timing and control, and is capable of expressing parallelism. For these reasons Computer Design Language is a good language at the statement level. But Computer Design Language lacks the ability to easily express the sequence of high-level actions. Also, it is hard for one to express his ideas in a high-level form and then take this into a low-level form.

It is also hard to understand the timing and control because there is no sequence indication of actions on either the high-level or the low-level. This seems to be one of the major drawbacks in trying to communicate a large system in this language. The timing is also fairly restricted to the use of synchronous actions.

B. Register Transfer Language

The Register Transfer Language is specified by T.C. Bartees and I.S. Reed which can be used for symbolic computer design[6]. A modified version of this language is presented by H. Schorr[10,11]. The example problem using the Register Transfer Language by Schorr is presented in Fig. 3. Although the form is quite different from the Computer Design Language, since the Computer Design Language uses ALGOL-like statements, similar control expressions and register expressions exist in both. Like C.D.L., R.T.L. is capable of showing timing and control on a statement level but it does not show it explicitly or such that it is not obscured by the action specification. Thus, it does not show explicitly the timing or control sequence. For instance, in the example of Fig. 3, if t_5 and k_3 were both true, and t_7 became true, one would have to search through each control expression until he found the expression that was true. In this light, it is easy to imagine someone who was unfamiliar with a large design trying to figure out what was happening. It would be a long and tedious, if not impossible, task.

This also points to the fact that Register Transfer Language is essentially a one-level and not a structured modular language with the only hierarchy structure being

in the mind of the user as he arranges his statement. Similarly a lack of clear and easy expression of other than serial synchronous control exists. This might become more of a handicap as systems become larger and more complex.

At the loss of a little conciseness and preciseness, such as is enjoyed by Iverson notation, this language is more readily understood on the statement level. This seems to be a desirable feature in that when one is first introduced to a system, he looks for basic operations that are taking place and is not as concerned with the more detailed features for the moment.

It can be seen that the preciseness of the over-all language has not suffered. Schorr shows the feasibility of an automated translator for both analysis and synthesis between a Register Transfer Language specification and Boolean equations for the system. This fact indicates that Register Transfer Language is relatively close to hardware even though the language is not dependent upon technology to any great extent. Thus, it would not be easily expandable as larger and more complex functions are developed. As can be seen from Fig. 3, Register Transfer Language has a few unique symbols for some of the more common functional operations. For example, Register Transfer Language would express a right shift as R(A)→L(A); 0→A(0). Whereas Computer Design Language would use A←1 shr A. This also indicates a little confusion that might exist at first

glance as to whether R(A) is a right shift of A or a variable register element of register R.

Because of the form and the simplicity of symbols, Register Transfer Language is desirable in that it would also be easily acceptable as an input language to an automated design sequence.

REGISTERS, R[0-10], F[0-3], A[0-10], C[0-5], G[0], D[0-5]
SUBREGISTERS OP[R] = R[0-2], ADDR[R] = R[3-10], I[F] = F[1-3]

$\|t_1 * PO\|$	:		$1 \rightarrow t_2$
$\|t_1 * \overline{PO}\|$	:		$1 \rightarrow t_1$
$\|t_2 * \overline{G}\|$	:	C←0; D←0	$1 \rightarrow t_2$
$\|t_2 * G\|$	:		$1 \rightarrow t_3$
$\|t_3\|$	:	M<C> → R; D+1→D	$1 \rightarrow t_4$
$\|t_4 * \overline{G}\|$	:		$1 \rightarrow t_2$
$\|t_4 * G\|$	:	OP(R)→I(F); 0→HALT	$1 \rightarrow t_5$
$\|t_5 * (k_0 + k_1)\|$	:	M<C> → R	$1 \rightarrow t_6$
$\|t_5 * k_2 * A(0)\|$	:	ADDR→D	$1 \rightarrow t_7$
$\|t_5 * k_2 * \overline{A(0)}\|$	:		$1 \rightarrow t_7$
$\|t_5 * k_3\|$	:	M<C> →A	$1 \rightarrow t_7$
$\|t_5 * k_4\|$	:	ADDR→D	$1 \rightarrow t_7$
$\|t_5 * k_5 * C(0)\|$	:	R(A) → L(A), 0→A(0)	$1 \rightarrow t_7$
$\|t_5 * k_5 (C(1)\|$	:	λ(A)→A	$1 \rightarrow t_7$
$\|t_5 * k_5 * C(2)\|$	:	0→A	$1 \rightarrow t_7$
$\|t_5 * k_5 * C(3)\|$	:	0→G; 1→HALT	$1 \rightarrow t_7$
$\|t_6 * k_0\|$	:	A+R→A	$1 \rightarrow t_7$
$\|t_6 * k_1\|$	:	A-R→A	$1 \rightarrow t_7$
$\|t_7\|$	:	D→C	$1 \rightarrow t_8$
$\|t_8 * HALT\|$	:		$1 \rightarrow t_2$
$\|t_8 * \overline{HALT}\|$	:		$1 \rightarrow t_3$

Fig. 3 An Example of Register Transfer Language

C. Digital System Design Language

J.R. Duley and D.L. Dietmeyer describe a language called the Digital Design Language (DDL)[5] which would be placed somewhere between Computer Design Language and Iverson Notation with respect to conciseness and preciseness. But Digital Design Language has some other desirable features that neither Computer Design Language nor Iverson Notation has. The example problem expressed in Digital Design Language in Fig. 4 shows that one such desirable feature is the multi-level notation as indicated by the formating of each statement with respect to one another. The timing and control indication is very similar to that of Register Transfer Language. But because of the multi-level structure, it is a little more complicated than in Register Transfer Language. The different form of expressing decoders makes a little difference in the actual control expression of Digital Design Language and Register Transfer Language.

A typical transfer of control expression in Digital Design Language would be $\Rightarrow$ JMP($\rightarrow$J2, $\Rightarrow$P3). This indicates that the next segment <SEG> would be JMP and that J2 is the first state to be executed in that segment. P3 would specify the next statement to be executed in the segment transferred from after JMP has finished.

A very strict point of this language is the specification of modular structures and their interconnection. This is a very desirable feature for some types of design.

Although Digital Design Language emphasizes what variables correspond to what type of element or operation in hardware, the language seems to be as independent of technology as any of the other languages.

Digital Design Language is capable of expressing parallel and asynchronous operations.

Although Digital Design Language does express the timing and control variables, it is hard to understand the sequence of timing and control on either a high level or a low level. The reason is that one must search through the variables to see when the next state is set and then find where the next state actions are specified. Although the notation in Digital Design Language has been simplified by the use of a multi-level structure over that of Computer Design Language, it still handicaps the language with respect to ease of understanding and the ease of formulating one's ideas with the use of a language.

It was found that in order to express a design in Digital Design Language, the designer had to have most of the system's details worked out in his mind before he could use Digital Design Language in a progressive manner.

Just as in Iverson Notation, the complex functional notation of Digital Design Language makes it hard for a de-

signer who is unfamiliar with the language to express his design effectively. For this same reason, Digital Design Language would not be as desirable for documentational purposes as some of the more universal languages.

In Fig. 4 one can see the different levels of structure by noting that mnemonics in < > are different distances from the left margin of the page. The farther it is from the left the lower the level of the language. For example, <AU> CPU and MEM indicates the CPU module and the memory module respectively. <TI> and <RE> indicate the timing and registers relating these two modules respectively. The <RE> <TE> and <OP> which are inset under <AU> CPU refer to the registers, terminals and operations in the CPU and relating the lower level submodules in the CPU. The <SEG> heading indicates the submodules contained in the <AU> module. For example, <SEG> ADD SUB would be a routine that one would transfer control to if an add or subtract instruction were being executed. Each statement within a <SEG> is identified by a control variable such as A1 which is the first statement in <SEG> ADD SUB. Transfer from one statement to another is accomplished by an arrow pointing to the label variable of the next statement. For example, statement A3 of <SEG> ADD SUB, →A4 indicates that the next statement to be executed is A4. A double arrow (⇒) indicate a transfer to a different <SEG> and a triple arrow (⇛) indicates transfer to a different <AU>. Thus, the <SEG> DECODE would be

a subroute which is an instruction fetch and interpretation cycle. Statement P4 decodes the instruction register and transfers control to the appropriate <SEG> which executes the instruction and returns control back to <SEG> DECODE. ⌈ F[1:3] ⌊2 ⇒ JMP(→J2 ⇒ P3) would indicate that bits 1, 2, and 3 of F are decoded and if the result is a two, then the transfer to statement J2 of <SEG> JMP and when JMP is finished executing control is then transferred back to statement P3 of <SEG> DECODE. A <SEG> is finished executing when it reaches a statement that contains a double arrow that does not point to any statement. Such as in J3 of <SEG> JMP.

```
<SY>  Example:
    <T1>  P(100E-9)
    <RE>      ST, POW, RP, R1, A1, C1
    <AU>  CPU :P:
       <RE> C[6], D[6], F[4], A[11], G, HALT
       <TE> ADD
       <OP> ADD1 (Y,Z) [0:10]
           <TE> YQ0:10], Z[0:10], C[0:10]
           <BO> ADD1 = Y ⊕ Z ⊕ (C[1:10] ∘ C[0]) · ADD
                C = Y:ZV(YVZ) · C[1:10] ∘ 0
       <SEG> DECODE
          <ST> PO:POW: C ← 0, D ← 0, →P1, |ST| G ← 1
               P1: |Ḡ| → PO; |RQ| ⇛ MEM(RD=1)
                  →P2; →P1, ↕¢ D .
               P2: RP: R ← R1 → P4
               P4: |Ḡ| → PO; F[1:3] ← R[0:2],
                 ΓF[1:3] |0:1 ⇒ ADDSUB (⇒P3)
                 |2⇒JMP (→J2,⇒P3) |3 ⇒STA (⇒P3)
                 |4⇒JMP (⇒P3) |5 ⇒MICR (→M1,⇒P3)
               P3: C ← D, |HALT| → PO; →P1.

       <SEG> ADDSUB
                A2: |F[3]| R ←  R, →A3; →A4
                A3:  ↕¢ R        →A4
                A4: A ← ADD1 (A,R), ⇒ .

          <ST>  A1: RP: ⇛ MEM(RD=1), →A11
                A11: RP: R ← R1, →A2

       <SEG>   JMP
          <ST> J1: |A(0)| → J2; →J3.
               J2: D ← R[3:10]  →J3.
               J3:   ⇒ .
```

Fig. 4 An Example of Digital System Design Language

```
    <SEG>     STA
        <ST> S1:  RP:⇛MEM(WR=1), ⇒.

    <SEG>     MICR
        <ST> M1: |C[0]|o→A, |C[1]| A ← A[1:10] o A[0]
                 |C[2]| A ← 0, |C[3]| G ← 0, HALT ← 1, ⇒.

<AU>  MEM:P:
     <EL>  MEMORY (MC[11]), RS, WS, AM[12], CM[6]
     <R>   A[6], C[11], R[11]
     <DE>  DL1(.7E-6), DL2(.6E-6)
     <BO>   A:AM, RD = RS, WR = WS
            C = CM, RP = RPM, R = RM
     <ST>   RPM: R5VWS: |WS| MC ← 0, → ME1., DEL1 = 1
            ME1: DEL1: |RS| RM ← MC  →ME2
            ME2: |RS| MC ← RM., |WS| MC ← AM., →ME3, DEL2=1
            ME3: DEL2: RPM ←1
```

Fig. 4 An Example of Digital System Design Language (continued)

D. Iverson Notation

The programming language developed by K.E. Iverson can be used as a design language as well as a programming language[1,2]. Iverson Notation uses a very complex set of symbols to be able to express operations in a very concise and precise manner. Although it has no multi-level expression as such, it does express its architecture by the use of "system programs" and "defined operations". System programs describe such operations as CPU interrupts, input/output channels, etc. which are more or less independent from and executed in parallel with the main program. Defined operations are similar to subroutines which are executed only when needed by the main program. Defined operations are used to describe such operations as memory access and instruction execution.

As indicated in the example problem in Fig. 5, this language is expressed as a series of statements which are executed sequentially unless a conditional statement transfers control to another statement. This can either be done in a semi-graphical manner by using arrows to lead to the next statement or numbering the statements and then listing the number of the next statement with the condition. Since only one statement is executed at a time parallel operation is difficult to express, it can also be seen that it is hard for the designer to specify the type or types of timing

control used since Iverson Notation does not explicitly show timing control.

A modular structuring effect can be obtained by statement arrangement on the part of the user.

There exists no multi-level expression as such and a large amount of detail must be included at the statement level, if one is to use the language as intended. Therefore it would be hard for a designer to easily express his ideas and to build on them as he follows through with the design. The language would also be hard to understand for the novice if it were to be used as a documentation language.

Iverson Notation has been used in a complete and formal description of IBM's System /360[3]. The acceptability of Iverson Notation as a computer input language is shown by the existance of a compiler called Alert[4] which will accept as inputs, a description of the desired computers architecture in Modified Iverson Notation. Then it will produce a set of Boolean equations to implement the desired architecture. The procedure used in doing this can be outlined as follows:

1. Express the desired architecture (including instructions format and repertorie, word length or marking convention memory size, and registers that are available to the programmer) formally in Iverson Notation.

2. Alert then a) determines the general layout, data paths, etc., b) provides selection logic to replace variable subscripts, c) replaces high order operations such as "add" and "subtract" with combinational logic, d) groups statements into minimum of groups and provides timing and control signals, e) eliminates duplicate gates, f) assigns flip-flops to variables that must retain their state and provide set-reset commands, g) simplifies vector and matrix interconnections, h) itemizes all interconnections.

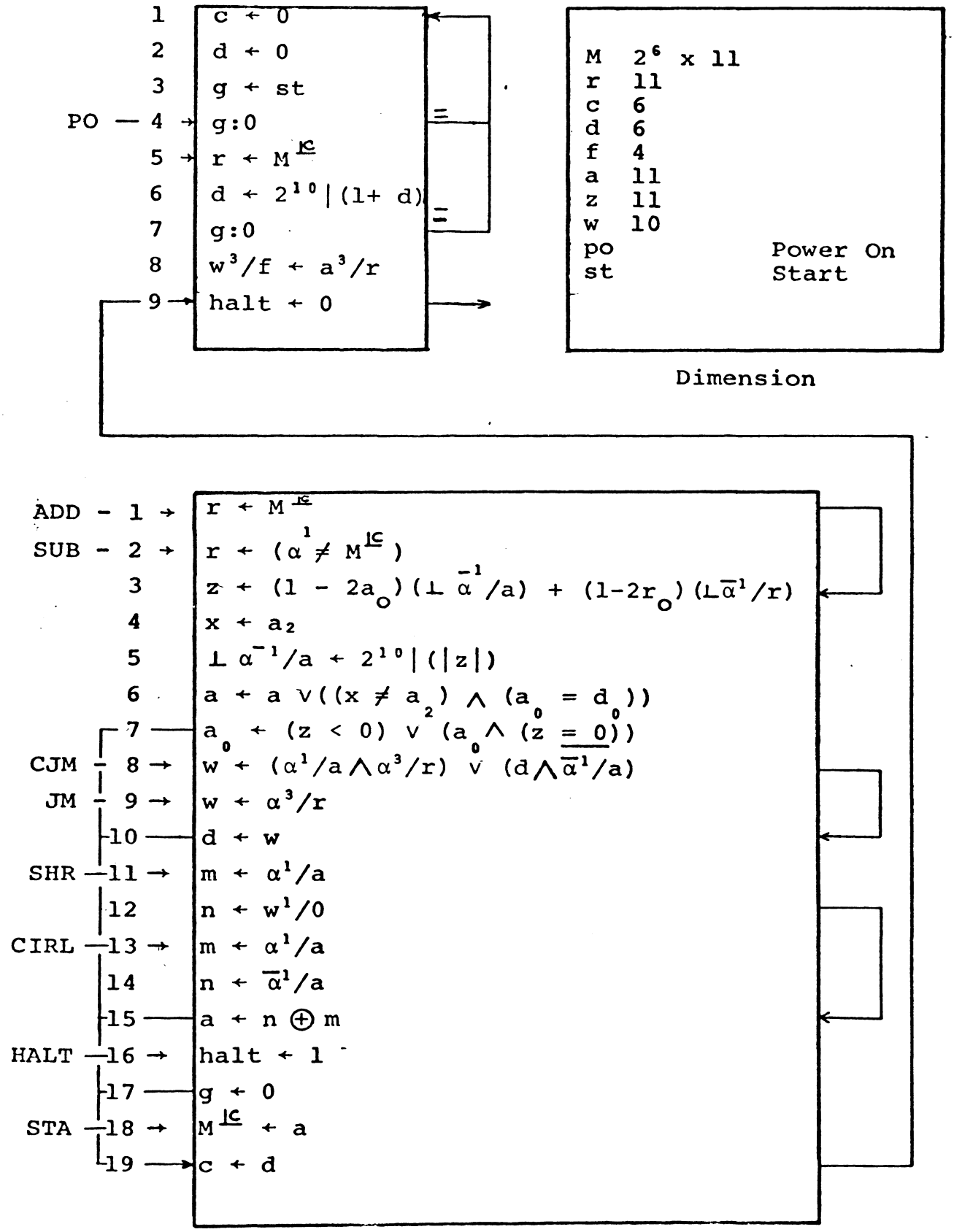


Fig. 5 Example of Iverson Notation

III. Comparison of Existing Languages

In the preceding review it was seen that each language had its own characteristic and desirable features. For example, Iverson Notation is a very precise and concise language with a highly symbolic functional representation which is also acceptable as computer input. Computer Design Language and Register Transfer Language are fairly low-level notation languages, but yet they are easily understood or grasped. Digital Design Language contains most of the desirable features of the others plus an ability in the language to represent multi-level or modular structures with the use of distinguishing format representations.

The one thing that all the languages seem to lack is a type of notation which can be used to express the initial design phase planning and also used as the design progresses by building upon that part of the system which has already been specified with the language. This means having a language which can be used throughout the design sequence without having to start all over again on each level of the design. This could be done by having a language which has a multi-level structure such that the designer can associate the levels of the design with the levels of the language. As a result this type of structure would also make the design more easily understood since it would put the design on different levels. It would also be desirable if the

language could be used to document the system when the designing is through. The documentation should not be so complicated, so large or symbolic, that it cannot be easily understood.

Also, many of the languages can express control or multi-level structure, but none of these seem to deal adequately with the problem of showing the control sequence and high-level control in a concise and understandable manner. In all instances one must "dig out" the high-level and control sequence from the other statement level actions.

For these reasons a language is presented here with the intention of satisfying as many as possible of the above mentioned characteristics and in doing so form a more design-oriented language.

IV. LANGUAGE DESCRIPTION

The actions in a digital system can be broken down into two basic components. First, there are the sequence control actions which guide the system through a particular sequence of actions depending upon the input conditions of the system. Second are the actions that this sequence controls. Fig. 8 is included in an attempt to give an overall picture of what our final goal will be before looking at the sequence control specification. From this a universal block can be found as shown in Fig. 6 which can be divided into 5 sub-parts as follows: union point, entrance conditions, action block, conditional branch point and branch condition, and parallel branch point.

A union point is an indication that control sequences of several blocks are converging into one control sequence in a particular block.

An entrance condition is a variable which stops the sequence flow until that variable becomes true. If one thinks of each block as a state, then the entrance condition is a variable which allows the machine to pass from the previous state to the next state according to the flow lines of the sequence control graph. The entrance condition variable can be any expression which can be reduced by evaluation to a logic 1 (true) or logic 0 (false). These variables

could be a result of operations in previous states or operations in states acting in parallel with this state. This condition could also be a sequence path from some other block. If this were the case, then the one sequence would have to finish before the other could begin. This would eliminate any possible inter-modular race conditions. For example, it would be desirable to make sure that the operand had been obtained and was in the proper registers before the instruction is executed if these two modules are not operating in series.

The action block is the actual specification of lower-level actions that are being controlled relative to other lower-level actions. These are sometimes referred to as modules, leaving the connotation that they are functional entities in themselves. As to whether these modules are actually self-contained or modular in the actual design is dependent upon the design requirements used in transferring from the design language to the design and not a direct result of the language.

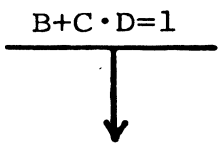
At this point, since one level of action is being represented by the intersequencing of lower-level actions, the multi-level effect becomes apparent. This is a very important effect in conveying the structure of the system in that first an overall specification can be stated in terms of large structures or modules which can, in turn, be specified

by smaller structures, etc. This permits a conveyance of ideas at the level most pertinent to the aspect being studied. Therefore, what is specified inside a block can vary from a macro-description of a particular module to a micro-description in the form of register transfers of very simple actions. The actual details of this micro-description will be discussed in a later section.

The conditional branch point is a point in the control sequence indicated by the diamond-shaped figure in which the control sequence can continue in different directions depending upon whether a particular expression is true or not. The expression is called the branch condition. For example, in Fig. 6 A,B,C, and D are the branch conditions. If the conditional branch point is a serial branch point, A,B,C and D must be mutually exclusive. Otherwise it is a conditional parallel branch point. More conditional branch paths can be obtained by including more diamond-shaped figures.

For simplicity, an unconditional parallel branch is indicated in Fig. 6 as two control sequences diverging from one point.

An entry point or starting point of a sequence control graph is indicated by a horizontal bar on top of a sequence control line as indicated below.



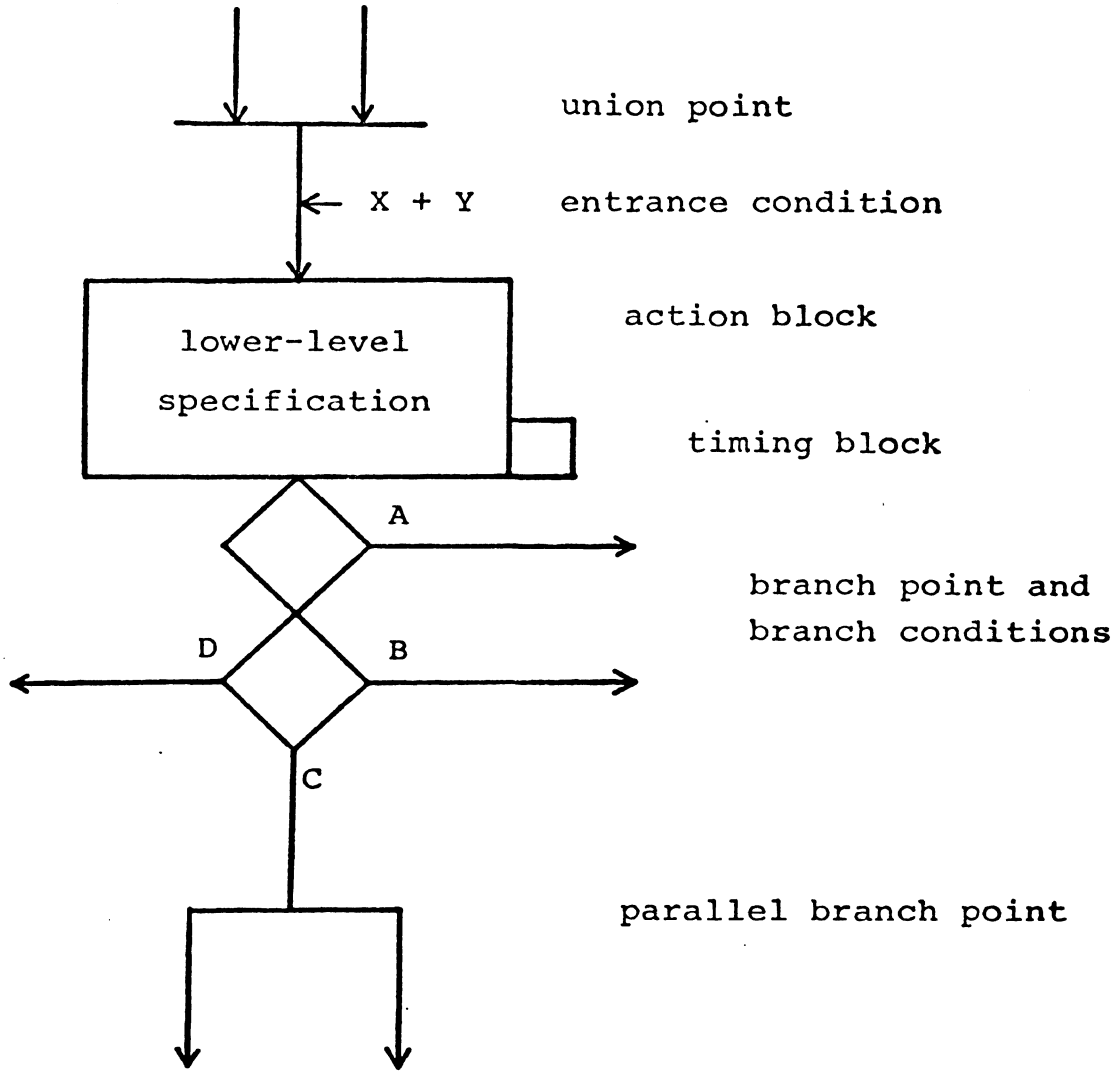
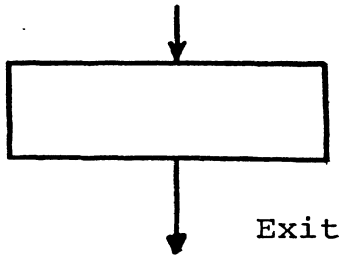


Fig. 6 A Universal Block

A relationship is listed above the horizontal bar. This relationship is the condition which must be true before a control sequence can start at this point. All entry points must be listed at the top of the page.

A sequence control line which does not point to any specific block as indicated below,



will be used to terminate a control sequence and generally indicates a completion of the particular sequence control graph which would return control back to a higher-level sequence control graph if one exists.

It can be seen that a sequence control graph specifies one or more sequences starting at an entry point, proceeding as indicated by the sequence control lines and stopping at a terminating point.

A circle with an identifying number placed in the center will be used to connect the same sequence control line of a sequence control graph on different pages. Fig. 12 shows how this can be used to allow the designer to continue a sequence control graph on another page.

Each block represents a period of time. This period of time will be referred to as a state and is the time interval in which the actions specified by the block occurs. Thus, if a block were named LDS then state LDS would refer to the

period of time represented by the block LDS. To specify a state time one must specify when it starts and when it ends or when it starts and how long it exists. For one to specify the state of the block precisely then he must specify the state of the block in one of these two ways. In the implementation phase of design one can see three different and distinct ways used to specify the state time. In synchronous timing a clock is used to generate the state by producing signals which indicate the beginning and end of the state. Delay elements and condition variables are used to form states by using the condition variable to indicate when the state is to start and the delay element to time how long the state is to exist. In a similar manner asynchronous states start when certain conditions become true and are timed by delay of the circuitry involved in the state. An asynchronous state is a state which is not of fixed length but one that can vary depending upon the circuit element delay of the state or on the particular operations being executed during the state. Since by specifying the beginning and the end of the state or the beginning and the length of the state one can specify the above mentioned types of timing then these two means will be used to specify the timing in this language.

To enable the language to specify these types of timing more clearly, a timing block will be attached to the lower right side of the block to identify which type of timing

will be used in that particular block. A "C" will be placed in the block to indicate that the state is a clocked state. Here it is assumed that there exists a periodic clocking signal with an effective clocking width of zero with respect to the rest of the circuitry. In actuality, the clocking signal may be of finite width but very short compared to the propagation time of the signals being clocked or it may correspond to the point of transition from one level to another. This clocking signal will be used as part of the entry condition signal and thus specifies the start of the state. The next clocking signal would then specify the end of the state. If a different clocking signal is to be used to determine the end of the state then this clocking signals mnemonic will also be indicated in the timing block as follows:"Variable:C" where the C indicates a clocked state. If two successive states have the same clocking signal to indicate the end of the first state and the beginning of the state, then the same pulse that ends the first state begins the next state. In other words one state immediately follows the next.

To indicate a state which is specified by a starting point and the length of time one of two types of timing blocks will be used, an A will be used to denote an asynchronous state in which the state time is determined by the operation speed of the circuit. A number in the timing block

will indicate the length of a state of fixed length. This will be referred to as a fixed time interval state. The fixed time interval state differs from an asynchronous state in that the length of a fixed time interval state is fixed regardless of the circuitry involved whereas the time of an asynchronous state would depend upon the particular circuitry used or the particular operations being executed during the state. The condition which indicates the beginning of the state is the entry condition of the block for both the asynchronous state and the fixed time interval state.

The transfers indicated in a clocked block are executed at the occurrence of the clock pulse which terminates the state. The transfers indicated in an asynchronous block are executed continuously during the state. The transfers of the fixed time interval block will be divided into two groups. These two groups will be separated by a horizontal line drawn through the block. Those transfers which occur continuously during the state will be listed above the horizontal line and transfers which occur at the end of the state will be listed below the horizontal line.

It is felt that this type of timing arrangement gives the designer a complete range of timing specification capability without tying the language down to any aspects of implementation. The designer can vary from a completely clocked system to a completely asynchronous system or any degree in between with ease and consistency of notation.

The language which will specify the actions being controlled by the control sequence will be discussed now. Table 1 gives a list of the symbols that will be permissible. The functional operators as listed in Table 1 is not a fixed or complete set but it is intended that these operators will be specified by the designer so that they are consistent with the functions that the designer wishes to use.

The numbers will be in hexadecimal when used as a constant in a register. This is to make the language more closely related to the actual implementation of the machine. The only exception would possibly be in variable names where a certain sequence of similar variables might be identified with consecutive decimal integers. For example, register A might contain three subregisters which would be identified as registers A1, A2, and A3.

Identifiers which are alpha-numeric character strings beginning with an alphabetic character are names given to the basic circuits such as registers, memory, adders, terminals, etc. The alphabetic character 0 will be distinguished from the numeric 0 by putting a slash through the alphabetic character Ø. Identifiers are also used for representing the output of these circuits. Thus, the identifiers ACD12 would be used to denote the numeric value of the register ACD12. Since ACD12 actually represents a set of 1's and 0's,(a series of outputs) then instead of actually naming an identifier to each output line or register cell, one

Classification	Symbols
1. digits	0 1 2 3 4 5 6 7 8 9
2. letters	A B C . . . Z a b c . . . z
3. values	true false
4. operators	
logical	- · + ⊙ ⊕
functional	add sub shl etc.
5. relations	= ≠ < > ≤ ≥
6. transfer	←
7. separators	, ; : ( ) [ ] { }
8. declarators	register subregister memory
	terminal constant operation

Table I Linguistics Symbols

identifier can represent the group of cells by first stating how many cells are in the particular register being represented by the identifier. For example, if ACD12 is a 16 bit register ACD12{16} would indicate this.

Subregisters (using register here to represent any of the above circuit elements) can be defined by stating which cells are used to form the subregisters. For example, if it were desirable to name the last five bits of register C as OP, then this would be expressed as OP = C{7-11} where OP has been defined as a five bit register (OP {5}). A variable element length can also be defined as follows: A = C{LEN}. Thus, A would be a subregister of C of length LEN beginning from the most significant side of C. Note that register lengths are specified in decimal. For example, if LEN = 101, then A = C{0-4} or the first five cells of C.

It is sometimes desirable to form one register from two or more registers. This is called concatenation. For example, to use the first five bits of A as the first five bits of X and the first seven bits of B as the last seven bits of X where X is defined as X {12}. This would be denoted as follows: X = {A{0-4}, B{0-6}}.

It is often useful in design to refer to one register in a group of registers or to address one register in a group of registers. For example a scratch pad memory called SP might contain 16 registers and to refer to the fourth,one would write SP[3], or if the register being referred to was

dependent upon the numerical value of another register, say C, then one would write SP [C]. Note that braces have been used to refer to actual elements of a register and brackets are used to denote one particular register in a set of registers. Thus a memory array (MEM), say 4,096 words, 16 bits long, would be specified as follows:

Memory: MEM[4096], MEM {16}

Another basic circuit element specifications is that of the decoder. The following notation will be used to specify full decoders:

DECODE: K[16] = F

Thus, the array K,one bit long and sixteen words wide, is the output of the full decoder of F. The numerical value of the code of F + 1 is equal to the number of the word of K, which is true. Thus, if F = 1010 then the output of word K[11] would be true and all others would be false.

In general, any combinational network can be declared by using the identifier

COMNET:

with the Boolean equation representation to the right of the identifier. For example, the "exclusive or" function would be specified as follows:

COMNET: FXØR = $A \cdot \overline{B} + \overline{A} \cdot B$

These identifiers will be used as part of the language to specify the basic circuit elements used in the language.

Values are used to verbally express that a condition or relation is a logical "1" for true and a logical "0" for false. For example, if the statement A+B=1 is satisfied then the statement is said to be true, or if A+B is to be true it is implied that A+B=1. In the first example, A+B=1, the equal sign can be replaced by any of the relation symbols. Another example would be, if the condition A add B≤1 is satisfied then the statement is said to be true.

From Table I it can be seen that the operation symbols have been broken up into three catagories, logical, functional and relational. Of the three catagories, relational has been discussed in the previous paragraph. The distinction between logical and functional is that logical operations represent operations which have a first-level associated representation in hardware, whereas functional operations are those which are formed by a series or group of low-level operations in hardware. The functional operations will be represented by a lower-case name which will be the name of the sub-system which executes this operation. For example, C ← A add B would indicate the addition of register A to register B mod 2^n where C is of length n and the results placed in C by a particular set of gates called add. It is intended that as higher level functions become of common use then appropriate functional operator names will be given to

them by the designer and thus build up the repertoire of high-level functional operators. Since this is totally dependent upon the system being designed, no attempt has been made to define all possible functional operators but it is left to the user to define the functional operators which would be suitable in his design. These functional operators would be specified as is done in Table II.

To specify the use of the operators, listed in Table I, Table II lists examples of the use of these operators and then gives an explanation of the operation.

By combining the use of the relational operators and the logical or functional operators, a conditional operation can be obtained. The relation condition will be understood to be on the left of the colon which separates the two and the functional or logical operation will be on the right. Thus, the expression A=B: C←D+E would indicate that the result of the logical "or" of D and E would be placed in C if and only if A were equal to B. So if the relation on the left is true, then the operation on the right is executed. For simplicity, relations of the form A+B=1 will be shortened to A+B:, where the =1 is implied. The colon was selected as the separator for its ease of reproduction, both manual and mechanical, and also for its lack of ambiguity. If no condition is required then the colon will be dropped and if the condition is the same as the preceding one, then the colon will be written but not the condition. For example:

Expression	Explanation
I Unary	
A) Logical	
+A	logical "or" of all bits of A
•A	logical "and" of all bits of A
⊕ A	logical "exclusive" or of all bits of A
⊙ A	logical "coincidence" of all bits of A
-A or $\overline{A}$	complement of each bit of A
B) Functional	
shl A	shift A one bit to the left and insert a 0 at the right end
cirl A	circulate A one bit toward left
etc.	
II Binary	
A) Logical	
A+B	logical "or" of corresponding bits of A and B
A•B	logical "and" of corresponding bits of A and B
A ⊕ B	logical "exclusive or" of corresponding bits of A and B
A ⊙ B	logical "coincidence" of corresponding bits of A and B
B) Relational	
A=B	contents of A equals the contents of B
A≠B	contents of A does not equal the contents of B
A<B	contents of A is algebraically less than that of B
A>B	contents of A is algebraically larger than that of B
A≤B	contents of A is algebraically less than or equal to the contents of B
A≥B	contents of A is algebraically larger than or equal to the contents of B
C) Functional	
A add B	add the contents of A to the contents of B using two's complement arithmetic
A sub B	subtract the contents of B from the contents of A using two's complement arithmetic
X shl A	shift A by X bits to the left replacing "0" on the right
etc.	the designer is free to specify any functional operator such that it meets his desired needs

Table II Explanation of Linguistic Symbols

```
A + B:  C ← D
A + B:  B ← A
```

would be the same as

```
A + B:  C ← D
     :  B ← A
```

If the complement of one condition is the condition of the next statement, then -: would be used to represent this. For example:

```
A + B = 1:  C ← D
A + B = 0:  C ← B
```

would be the same as

```
A + B:  C ← D
    -:  C ← B
```

This is the form of an if-then-else statement in some of the programming languages.

To show how these two concepts of control sequence specifications and controlled action specifications can be used together,consider Figures 7,8 and 9 as a complete specification of the example problem that was presented earlier in the language comparison section. Fig. 7 gives a description of the elements used in the design. Fig. 8 is the high-level sequence control graph of the machine. Since P is defined to be a clocked signal, then it is known that each state that has P as an entry condition is a clocked state.

Blocks S0, S1, S2 and S3 are all low-level blocks. Block S4 is a high-level block since it gives the name of another sequence control graph which specifies what is to happen when this state becomes true. Block S4 is the execute sequence block and its sequence control graph is Fig. 9.

One can determine the operation of this machine by following the control indicated in Fig. 8. If the machine is off and then turned on, it transfers a zero into G at the first clock pulse P. Now state S1 becomes true and upon the occurance of a clock pulse P, a zero is clocked into C and D and if ST is true a one is clocked to G. Control is still transferred back to state S1 until ST becomes true. If ST is true, then G is set to one and on the next clock pulse state S2 becomes active. At the occurence of the next clock pulse the contents of memory location C is transferred to register R and register D is incremented by one. If G is zero, then control goes back to state S1, but if G is one then control would go to state S3. At the occurance of the next clock pulse the contents of ØP would be transferred into I and zero into HALT. The next clock pulse would execute the appropriate state of the execute sequence of Fig. 9.

When the execute sequence has completed, the control would go to state S1 if $\overline{G}$ were a one or it would go to state S2 if $\overline{G}$ were a zero. Thus, the machine would continue to cycle through the different states of its sequence control graph.

This example shows the multi-level capability of the language as is depicted by state S4. It also shows how the timing and control sequence can be shown explicitly by the use of the state blocks and arrows to show the control sequence between blocks.

```
REGISTERS: R{11}, F{4}, A{11}, C{6}, G{1}, D{6}

SUBREGISTERS:  ØP = R{0-2},  ADDR = R{3-10},  I = F{1-3}

MEMORY:  M[32],  M{11}

DECODER:  K[16] = F

CLOCK  :  P

SWITCHES: ST
```

Fig. 7 An Example of Design Oriented Language, Part I

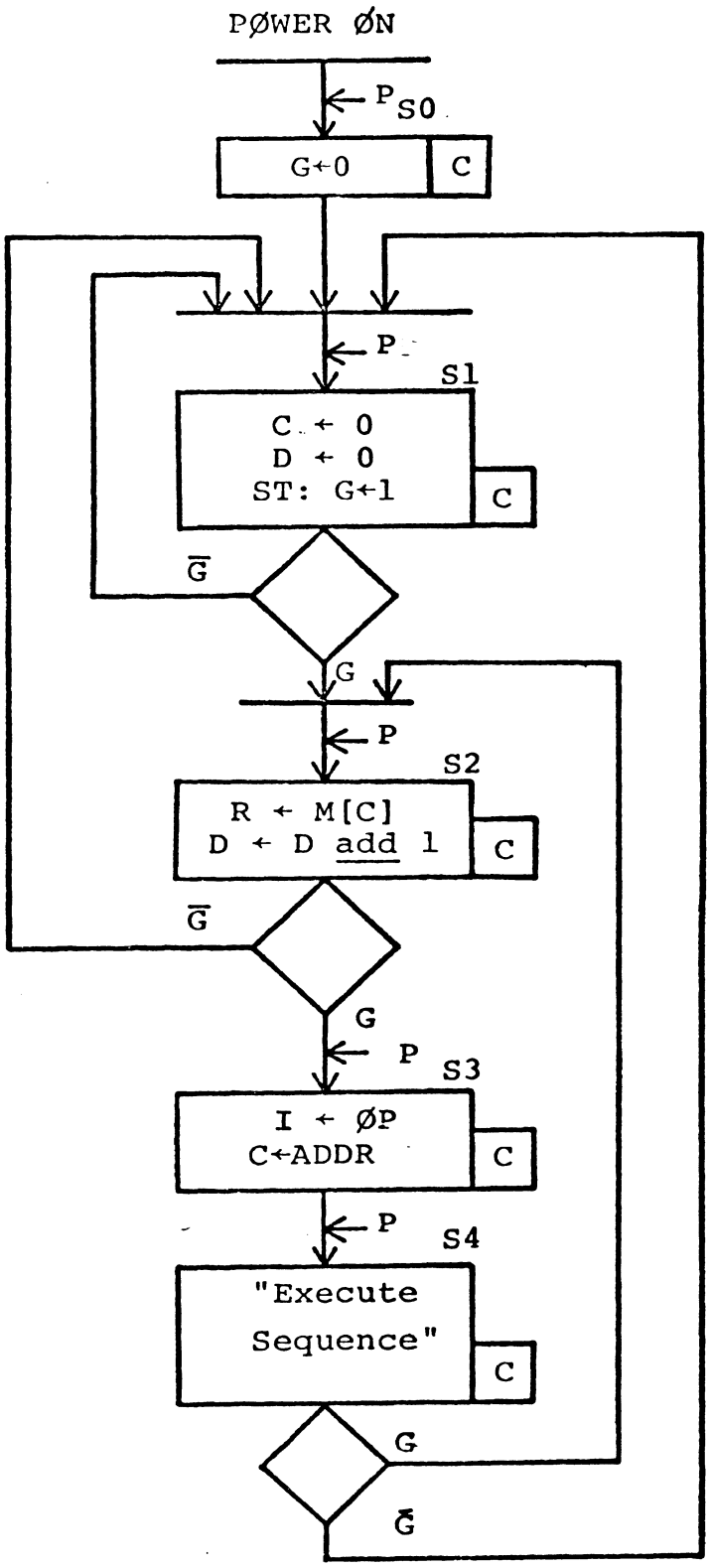


Fig. 8 An Example of Design Oriented Language Part II

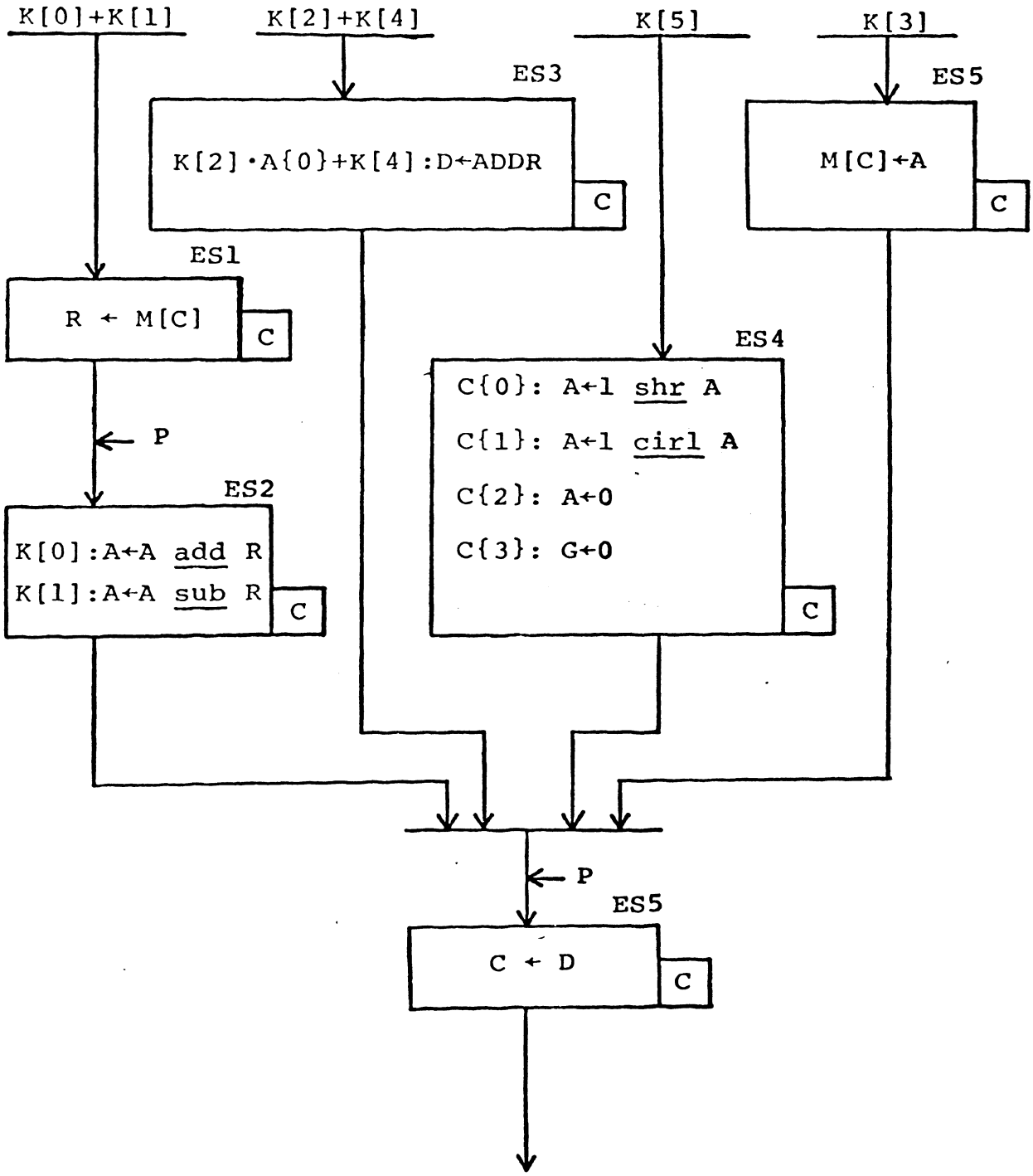
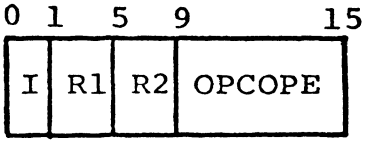


Fig. 9 An Example of Design Oriented Language Part III

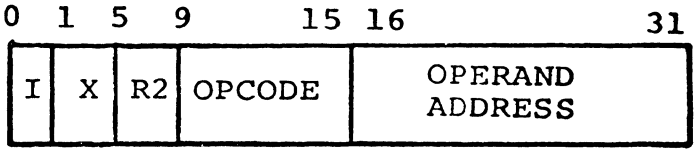
V. A DESIGN PROCEDURE

In an attempt to indicate the use of the design language being presented in this paper, one can consider the design sequence of a digital computer when the main design specifications are as follows:* The word length will be 32 bits long with instructions of lengths 16, 32, and 48. The 16 bit instruction will indicate the use of scratch pad memory at the first level of the operand fetch cycle. The 32 bit instruction will obtain one of its operands from magnetic core storage and the other from scratch pad memory. The extra half word of the 48 bit instruction will be used to extend the number of executable instructions. Thus, the instruction formats would be as indicated in Fig. 10.

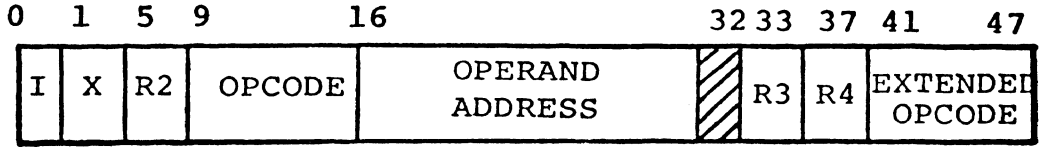
*These design specifications are part of the design specifications used in the design of the 7501C-4 arithmetic, logic and control unit being built by Collins Radio.



1. 16 bit instruction



2. 32 bit instruction



3. 48 bit instruction

Fig. 10 Instruction Field Specification

The R fields specify registers in the scratch pad memory. The R2 field is a direct memory reference. The R1 field along with the I bit specifies direct or indirect modes of addressing. In the 32 bit or 48 bit instructions the operand address field indicates the address in magnetic core storage of the operand. The I bit indicates indirect access and the X field indicates the register in the scratch pad that is used for indexing. The size of the scratch pad memory is 16 words and will be referred to as location $0\text{-}3F_{16}$. The size of the magnetic core storage memory is 2^{21} words or $IFFFFF_{16}$ words. Instruction look-ahead will be used to make use of parallel processing.

Considering the fact that it is possible to obtain two instructions in one memory cycle and that one word is going to have to be obtained in parallel with the one being executed, it can be seen that an acceptable register configuration for the first sequence control chart level would be as follows:

IBA {32}; Buffers the instruction word from memory

IBB {16}; Buffers an instruction when a full-word instruction is not aligned with a full-word boundary.

AR {16}; Holds either a half-word instruction or the operand address field of an instruction.

FB {16}; Contains the operation field of the next instruction to be executed.

F {16}; Contains the instruction currently being executed.

HWS : Indicates a half word boundary

BR : A branch instruction is being executed

BRC : There is a pending branch instruction

REMTJ : Indicates a 16-bit instruction is being executed.

EXMTJ : Indicates a 48-bit instruction is being executed.

MFMTB : Buffer that indicates a 32-bit instruction.

WALM : Wait on Arithmetic Logic Module

ESHW : A sum of variables which will be defined later.

BRI : Branch Instruction detect

ØPR : OPerand Required for present instruction.

ØPNR : OPerand Not Required for present instruction.

The formation of a high-level sequence control chart is a relatively simple task even though only the macro-features of the machine have been specified. For example, the sequence control chart of this machine can be roughly outlined for the moment as is indicated in Fig. 11. After initializing is done in state IR1, a parallel branch is indicated to allow the next instruction to be accessed by the NIAM module while the execution of the present instruction is

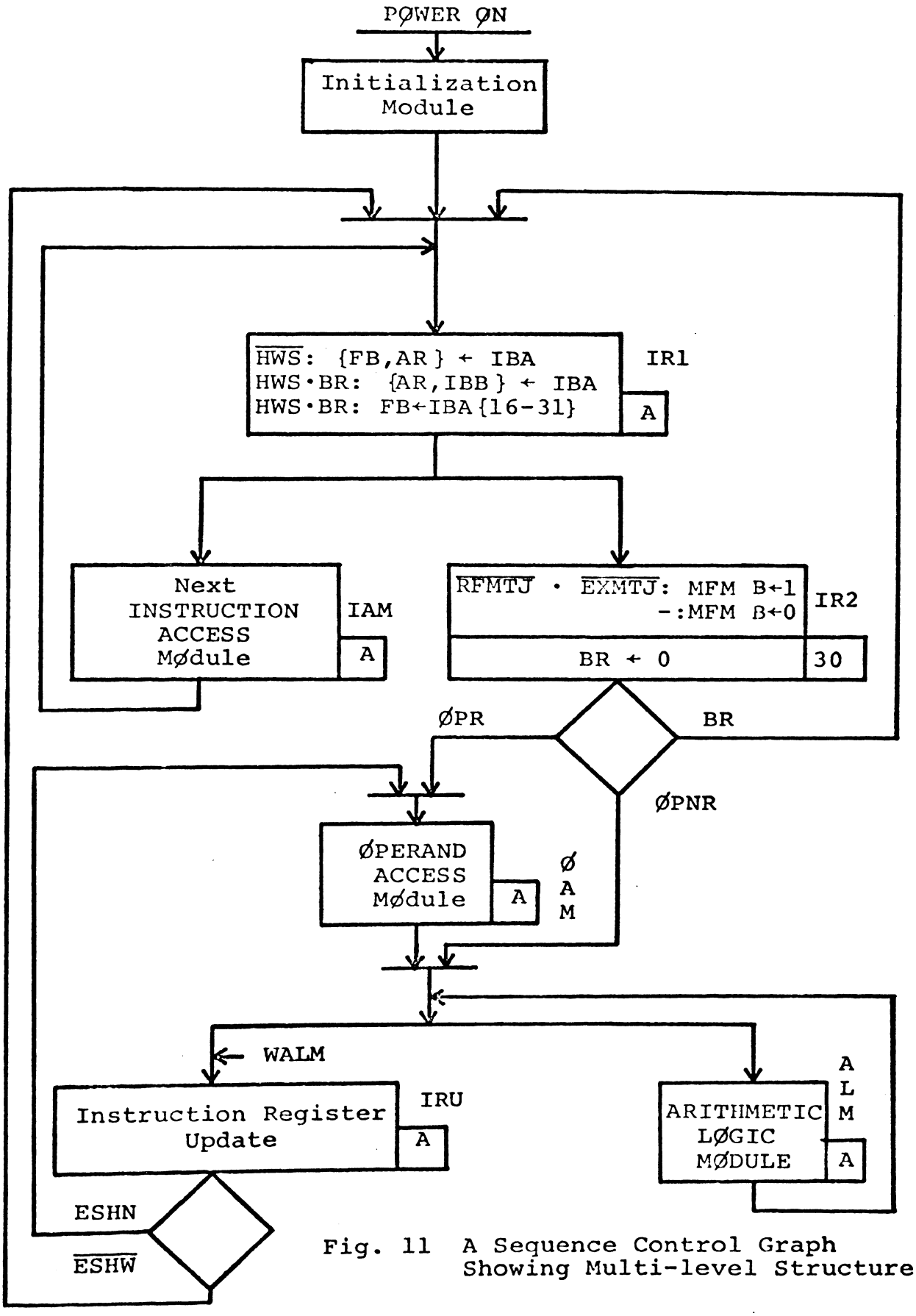


Fig. 11 A Sequence Control Graph Showing Multi-level Structure

occuring. The execution sequence starts in state IR2. In the execute sequence the operands are first obtained by the OAM module. Then if the last instruction has finished execution, the ALM is initiated to execute the present instruction in parallel with the IRU. The IRU is the instruction register update sequence. Since the interlocks and branch point conditions such as WALM, BRC, ØPR, ØPNR, and BR are dependent upon the instruction reportoire, and their exact specifications would not enhance the discussion of the design sequence, they will not be further specified.

The ability to indicate both high-level states and low-level states lets the designer put the modular actions in the proper perspective with the more important low-level operations. For example, block NIAM obtains the next instruction word to be placed in IBA, the block OAM obtains the operand of the instruction contained in FB and AR which will be executed next by the ALM. The NIAM, OAM, ALM, and IRU are all high-level blocks, whereas IR1 and IR2 are all low-level blocks. IR1, IR2, and IRU insure that the instruction registers are loaded at the proper time and with the proper values.

Looking at the more detailed sequence control graph of Fig. 12, one can see the loading sequence of IBB, AR, FB, and F for the different types of instructions. One also can note how they are arranged so that the proper values are in the proper registers when the execution modules are enabled. Fig. 13 gives the inter-register connection paths for IBB,

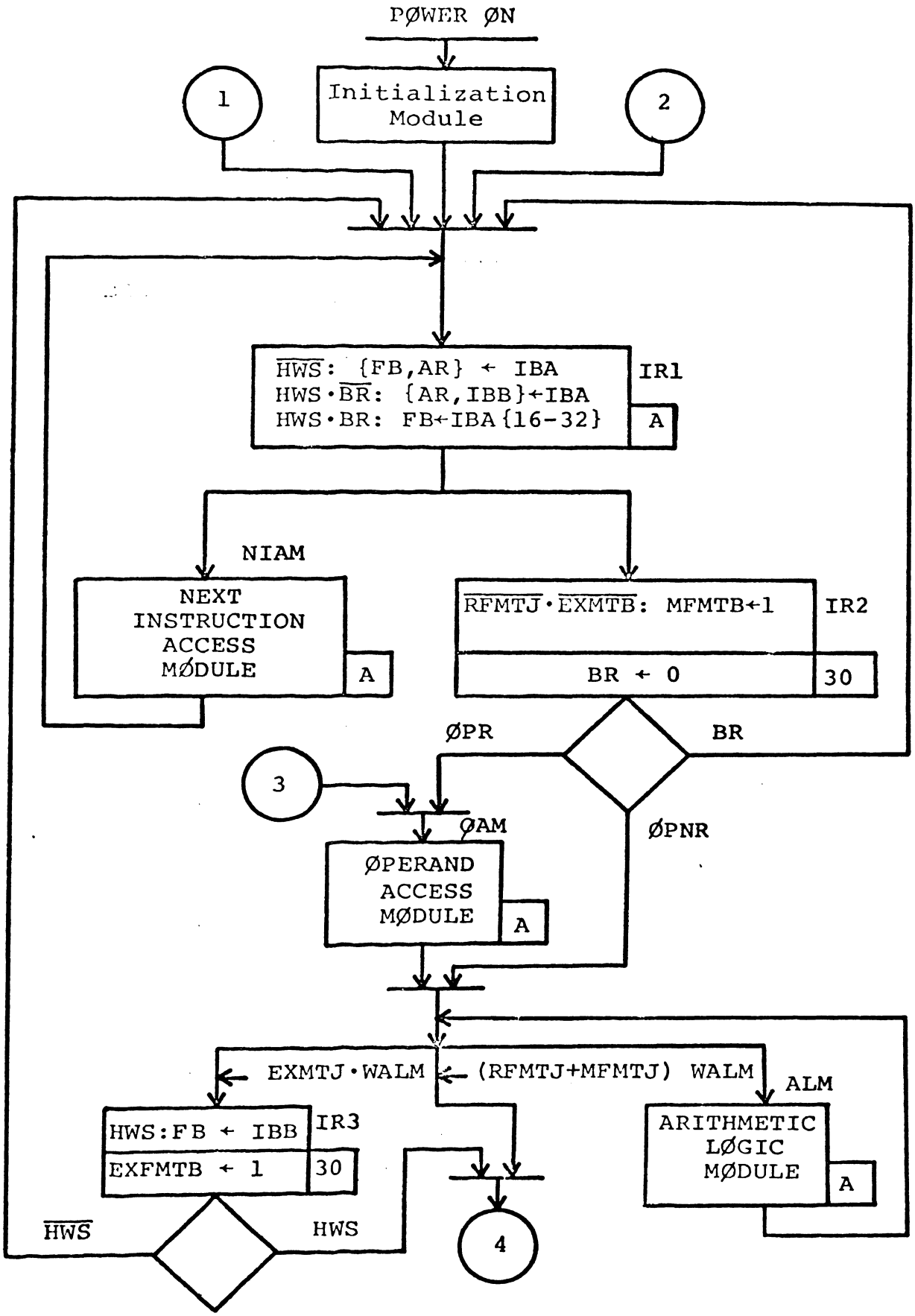


Fig. 12 Example Design's Sequence Control Graph

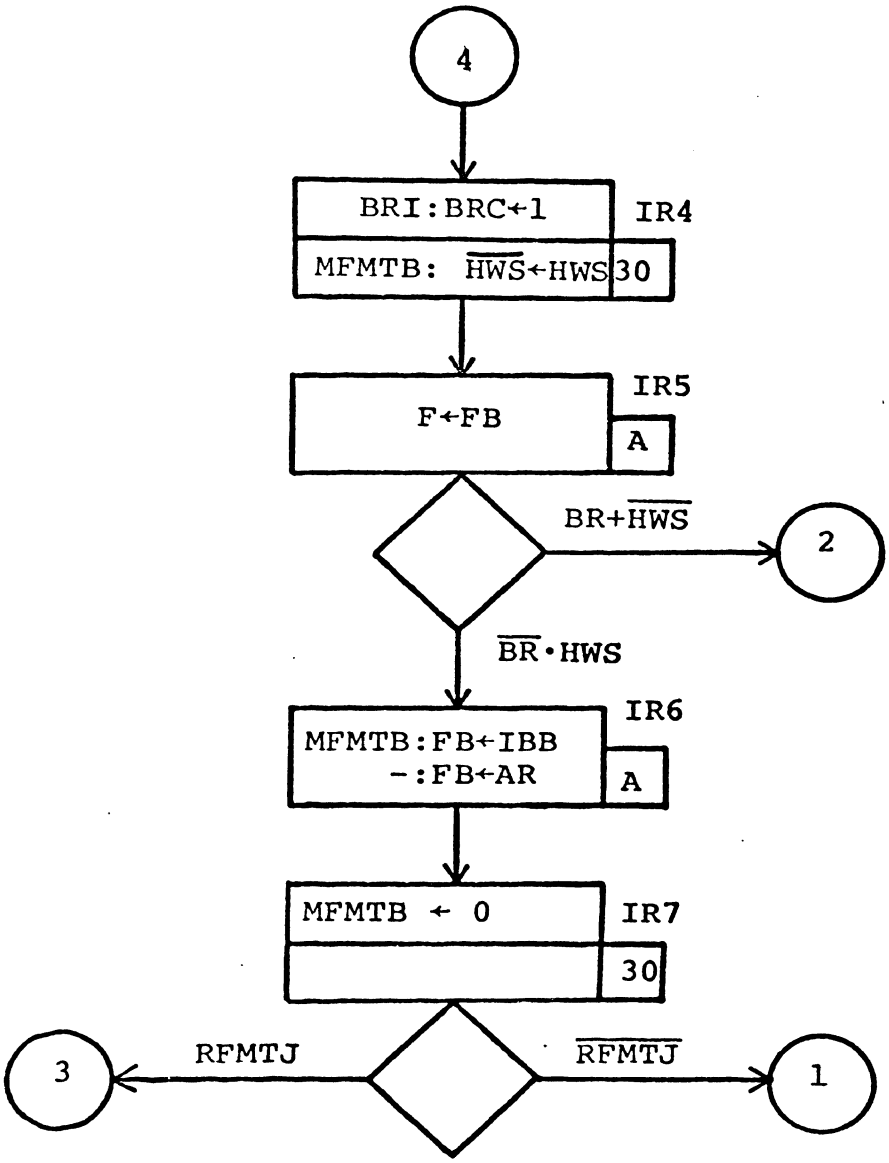


Fig. 12 Example Design's Sequence Control Graph (con't)

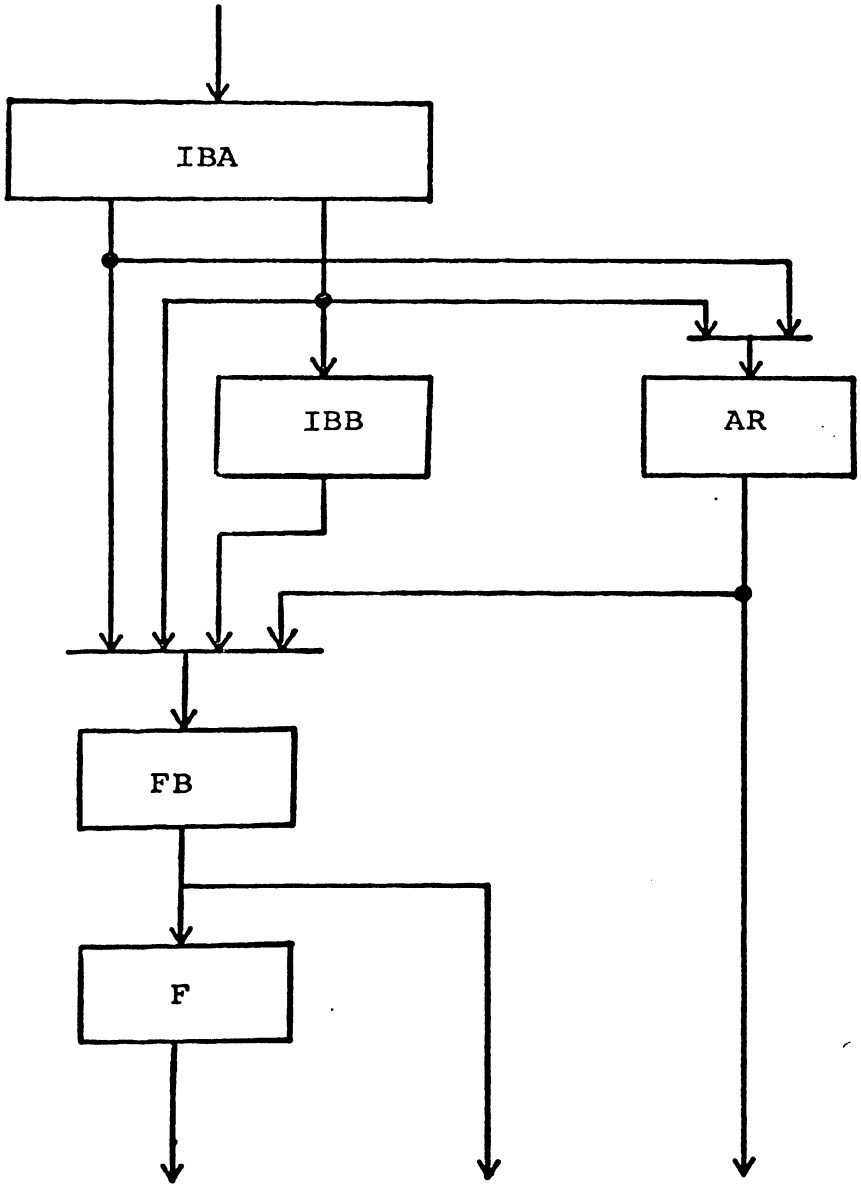


Fig. 13 Instruction Register Interconnection

AR, FB, F, and IBA to help indicate the loading sequence. After IBA has been unloaded in state IR1, state NIAM, operating in parallel with the instruction execution, obtains the next word and places it in IBA. The interlock requires that NIAM be through before the next cycle can begin. In the execution path the operand for the word in FB and its corresponding address field are obtained while the ALU is executing the instruction in F. When the OAM is finished, if the ALU is through, executing its instruction, the operand is transferred to the ALU and FB is transferred to F and the cycle starts over again. All three major units can be processing at the same time.

At this point, it can be seen that a description of this level can be very helpful in the design sequence. It is easy for a designer, who is working on one particular module, to grasp his job and the way it interfaces with the rest of the system without having to understand the details of the other modules. This level of description also helps make it clearer where the design boundaries fall.

The design sequence would be furthered,at this point,by taking each of the modules specified in the high-level sequence control graph and producing similiar sequence control graphs for the actions of these modules. This would be repeated until the lowest-level of specification was reached. Since the purpose of this example is to show the design se-

quence, the detail specification of the modules indicated in the high-level description will not be given here.

To use this language as a documentation of the machine, all that must be done is include the sequence control graphs of the completed system starting with the highest level and the element specification with each sequence control graph. This would form an easily understandable but yet precise description of the machine. The logic diagrams would be used to correlate what is being done and how it is being done in the hardware.

A desirable characteristic feature of each high-level state (or module) is that each is an entity in itself. It is set up very similar to a sequential machine in that it accepts as its inputs a relatively small number of stimuli and goes through a sequence of actions to produce the desired output. For example, the OAM accepts as its inputs the instruction field and the address field and produces as its output the desirable operands which are used as the inputs to the ALU. Thus, we have also reduced the interconnections between modules and thereby simplified the physical layout.

Fig. 14 shows the simplicity of the physical layout created by the natural partitioning of the language. The great simplicity is evident since in each module the physical layout has a corresponding module in the sequence control graph. Although this similarity does exist it is not a manditory result of the language but is totally in implementa-

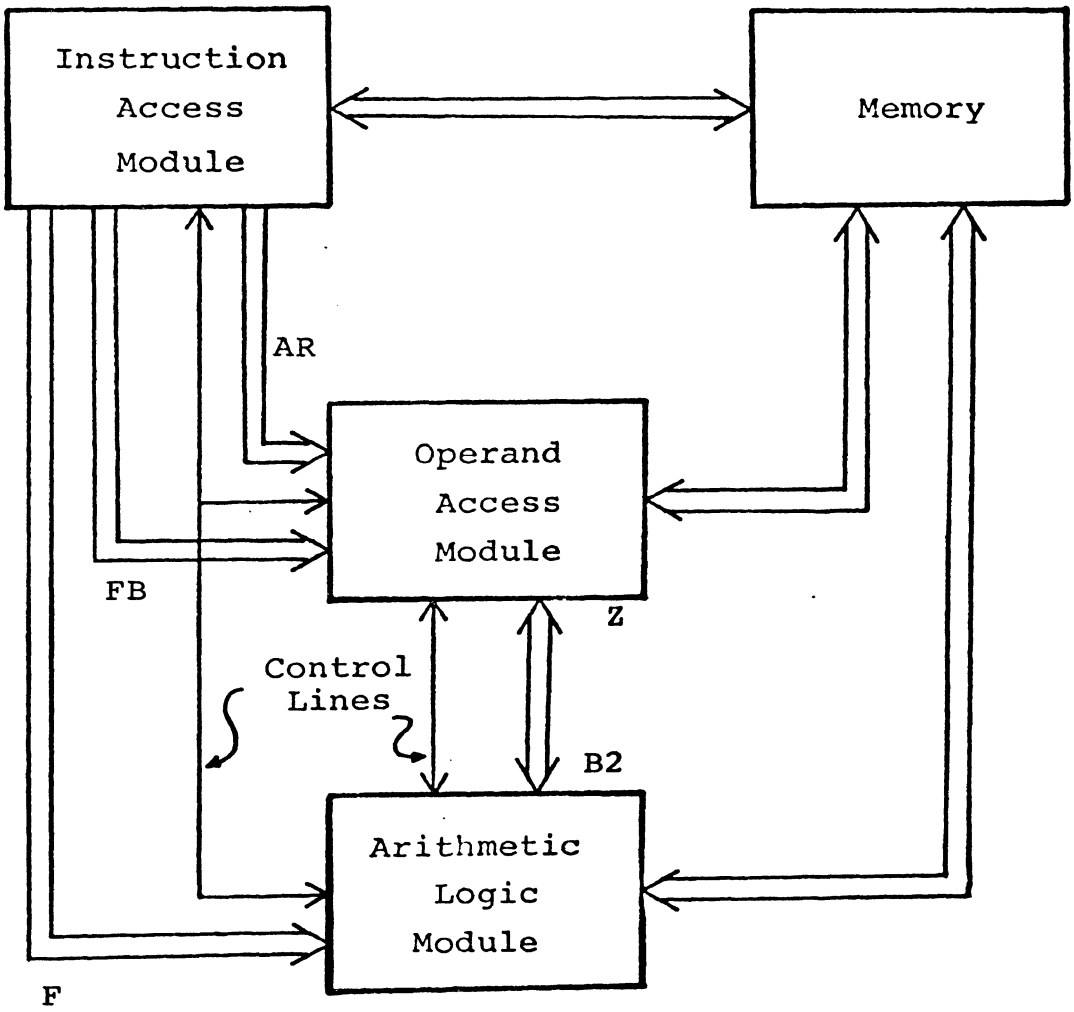


Fig. 14 System's Intermodular Connection

tion consideration. The language only provides a natural and easy mode of expressing such organization.

The modular multi-level structure of the language also simplifies the process of similation and fault diagnosis. Due to the modular structure, simulation could more easily be done in the initial stages of design on a functional level. Because of the lack of interdependency, the functional simulator could be used in the later stages of design as a controller for logic simulation on a modular basis. Fault diagnosis could also be applied on a modular level, thus reducing the size of the circuit being diagnosed.

VI. SUMMARY

The goal of this project was to define a design language based upon an ideal design sequence so that future designs would be a more formal and continuous process from start to finish. It was found that with the use of the multi-leveled aspect of this language the design sequence progresses fluently as the design is being formulated. A language of this form makes design communication easier between system architecture and logic design, and at the same time provides definite design boundaries between logic design groups.

Due to the ability of this language to express all types of control philosophies from serial to parallel and from asynchronous to synchronous, the designer is able to structure his system quite easily whether it be a large sophisticated machine or a small processer.

The modular, multi-level structure of the language permits the designer to use simulation in a more effective manner. This is done by using simulation concurrently with design. This simulation would be initially performed at the functional level. Then, as the logic design is being completed the functional simulator would be used as a

logic simulation controller. With this approach one need not have all the logic design completed before starting design verification through simulation.

Just as important, is the ability to express only a part of the total system at the gate level, since for large systems gate level simulation of the total system becomes impractical.

System documentation becomes an easier task because this language can be used as an organizer of the logic description. This would lead to a closer relationship between the more easily understood functional description and the more detailed logic description.

It is felt, that this language is a more design oriented language than existing languages. Hence, it can be used as an effective design tool to shorten and make more consistent the design cycle of digital systems.

BIBLIOGRAPHY

1. Iverson, K.E., (1962) "A Programming Language", New York, John Wiley & Sons.

2. Hellerman, H., (1967) "Digital Computer System Principals", New York, McGraw-Hill, p. 424.

3. Iverson, K.E., Falkoff, A.D., and Sussenguth, E.H., (1964) "A Formal Description of System /360", IBM Systems Journal, Vol. 3, p. 198-262.

4. Friedman, T.D., (1967) "Alert: A Program to Compile Logic Designs of New Computers", Digest of the First Annual IEEE Computer Conference, Chicago, p. 128-130.

5. Duley, J.R., and Dietmeyer, D.L., (September 1968) "A Digital System Design Language", IEEE Transactions on Computers, Vol. C-17, No. 9, p. 850-861.

6. Bartee, T.C., Lebow, I.L., and Reed, I.S., (1962) "Theory and Design of Digital Machines", New York, McGraw-Hill, p. 324.

7. Chu, Y., (1965) "An ALGOL-Like Computer Design Language", Communications of the A.C.M., Vol. 8, No. 10, p. 607-615.

8. McCurdy, B., and Chu, Y., (1967) "Boolean Translator of a Macro Logic Design", Digest of the First Annual IEEE Computer Conference, Chicago, p. 124-127.

9. Breuer, M.A., (December 1966) "General Survey of Design Automation of Digital Computers" Proceedings of the IEEE, Vol. 54, No. 12, p. 1708-1721.

10. H. Schorr, (December 1964) "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language", IEEE Transactions on Electronic Computers, Vol. EC-13, p. 730-737.

11. H. Schorr, (December 1962) "Towards the Automatic Analysis and Synthesis of Digital Systems", Ph.D. Dissertation, Department of Electrical Engineering, Princeton, University, New Jersey.

12. McCurdy, Bruce D., and Chu, Yaohan, (September 6-8, 1967) "Boolean Translation of a Macro Logic Design", Digest of the First Annual IEEE Computer Conference, p. 127.

VITA

David Michael Rouse was born on September 16, 1945 in Joplin, Missouri. He received a Bachelor of Science degree in Electrical Engineering from the University of Missouri at Rolla in June 1967. He has been enrolled in graduate school at the University of Missouri at Rolla since September, 1967. He has been on the staff of the Electrical Engineering Department since September, 1967.