01 Jan 1974

An Asynchronous Circuit Design Language (ACDL)

Gregory M. Bednar

James H. Tracey
Missouri University of Science and Technology

Recommended Citation

G. M. Bednar and J. H. Tracey, "An Asynchronous Circuit Design Language (ACDL)," *IEEE Transactions on Computers*, vol. C thru 23, no. 9, pp. 971 - 976, Institute of Electrical and Electronics Engineers, Jan 1974. The definitive version is available at https://doi.org/10.1109/T-C.1974.224064

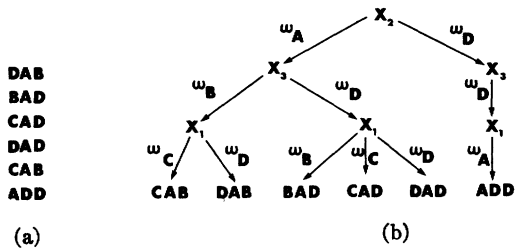

Fig. 1. (a) Set of words to be stored in DICTIONARY. (b) Diagnostic key of DICTIONARY.

lent to a trigram test d_{ijk} or as realizing simultaneously the digram test d_{ik} and d_{jk}. In fact, as the mth test of the diagnostic key is being performed, $m - 1$ digram tests are performed simultaneously.

Although there is reason to doubt the possible existence of efficient algorithms for generating optimal diagnostic keys (keys which minimize the expected number of tests to be performed) [1], efficient heuristics have been proposed [2], [4] for generating keys which approximate optimal keys.

Diagnostic keys are efficient with respect to time because after a series of tests has been successfully performed, one need not search for the word in DICTIONARY, since its existence will have been demonstrated as a side effect of the tests.

With respect to memory, the storage required for the digrams is obviated and the storage for DICTIONARY may be reduced (in spite of the need to store all the pointers), since the words may share portions of the diagnostic keys. A simple storage scheme consists of storing the pointers as stacks of 26 words (depending upon the machine and language being used, the pointers could be packed more efficiently). Updating under this storage scheme is particularly attractive, since one need only follow the path of a new word until the first missing pointer, and add the appropriate tests and pointers thereafter. The storage needed for the diagnostic key will surpass that of DICTIONARY as the words become more and more dissimilar, no longer sharing common paths of the key.

REFERENCES

[1] L. Hyafil and R. L. Rivest, "Graph partitioning and constructing optimal decision trees are polynomial complete problems," Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, Res. Rep. 33, Oct. 1973.
[2] A. Rescigno and G. A. Maccacaro, "The information content of biological classifications," in *Information Theory—A Symposium*, C. Cherry, Ed. London, England: Butterworth, 1960, pp. 437–446.
[3] E. M. Riseman and R. W. Ehrich, "Contextual word recognition using binary digrams," *IEEE Trans. Comput.*, vol. C-20, pp. 397–403, Apr. 1971.
[4] J. R. Slagle and R. C. T. Lee, "Application of game tree searching techniques to sequential pattern recognition," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 103–110, Feb. 1971.

An Asynchronous Circuit Design Language (ACDL)

GREGORY M. BEDNAR AND JAMES H. TRACEY

***Abstract*—This correspondence describes a special purpose Asynchronous Circuit Design Language (ACDL) for specifying the terminal behavior of asynchronous sequential circuits. The language is a valuable tool for formalizing and documenting asynchronous designs, as well as providing a user interface to a completely automated synthesis system. The language includes many special features which permit quick and precise specification of terminal behavior and is best suited for problems that are currently being described informally by word statements.**

Index Terms—**Asynchronous sequential circuits, design automation, design languages, sequential circuit synthesis, sequential circuit specifications, switching theory.**

Manuscript received January 11, 1973; revised October 1, 1973. This work was supported in part by the National Science Foundation Grant GK-20190.
G. M. Bednar is with the Systems Development Division, IBM, Rochester, Minn. 55901.
J. H. Tracey is with the Department of Electrical Engineering, University of Missouri–Rolla, Rolla, Mo. 65401.

I. INTRODUCTION

A number of synthesis procedures for fundamental-mode asynchronous sequential circuits now exist and many of these have been programmed for computer application[1]–[6]. Although considerable work has been directed toward improving the synthesis procedures, little has been done in interfacing the user to these procedures.

This paper presents an Asynchronous Circuit Design Language (ACDL) which provides a simple and concise means of describing the terminal behavior of asynchronous sequential circuits. As a circuit model, ACDL is more precise than word statements and more natural than flow tables. When an ACDL description is combined with a translator [7] and synthesis programs [4]–[6], the circuit design equations can be generated automatically. However, even without translation the language is a powerful tool for specifying, formalizing, and documenting a design.

Procedures exist for specifying asynchronous sequential circuits when the terminal characteristics of the machine are easily expressed in input/output sequences [6], [8]. However, only a small percentage of designs are suitable for this type of terminal description since for most circuits, the task of specifying all possible input-output sequences is not easy or straightforward.

ACDL is based on the observation that the complete terminal behavior of an asynchronous sequential circuit can be uniquely specified in terms of relatively few critical input-output events. Many designs originate in the form of critical events, and typically, a designer first converts these critical events into a word statement for later use in the manual construction of the primitive flow table. With ACDL the designer can proceed directly with the listing and formalization of the critical events without the need for specifying informal word statements and manually constructing flow tables. Further advantageous characteristics of ACDL are

1) it permits ease of expression;
2) it retains a formal structure;
3) it provides compact and descriptive specifications;
4) it permits automatic translation;
5) it provides readable documentation for a design.

A review of well-known digital design languages [9]–[14] reveals that they are intended for networks whose designs can best be described by functional operations and information transfers between basic hardware elements such as registers, switches, terminals, memory, etc. None of these languages were found to satisfy all of the above desired characteristics for specifying the terminal (input/output) behavior of an asynchronous sequential circuit. Specific drawbacks of these languages include the inability to assign transition values to variables, the inability to make proper declarations such as "input constraints" and the inability to list multiple independent sequence paths without introducing additional control variables or cluttering the listing with many "go to" type statements.

II. A DESCRIPTION OF ACDL

The following discussion is an informal description of the salient characteristics of ACDL. A complete, formal definition of the language is given in the Appendix. All symbols used in ACDL can be formed from characters available on standard computer keyboard devices.

A. Transition Value

In Boolean Algebra a symbol can stand for either a 0 value or 1 value. In addition to these level values, ACDL defines a more important type of value called the transition value. The possible transition values are '0 → 1' and '1 → 0', where '→' is the transition symbol and is read as "makes a transition to."

Circuit input variables will take on transition values as well as level values. Table I illustrates a shorthand notation that is defined

TABLE I
SHORTHAND NOTATION FOR BASIC TRANSITION EXPRESSIONS

Expression	Shorthand Notation
$X = 0 \rightarrow 1$	$X \rightarrow 1$
$X = 1 \rightarrow 0$	$X \rightarrow 0$
$(X = 0 \rightarrow 1) + (X = 1 \rightarrow 0)$	$X \rightarrow ?$

TABLE II
EXAMPLES OF TRANSITION EXPRESSIONS

Expression	Logical Meaning	Informal Meaning
$X1 \rightarrow 1$	$X1$ makes a transition from 0 to 1	$X1$ turns on $X1$ goes up etc.
$X1 \rightarrow$ and $X2 \rightarrow 0$	$X1$ makes a transition from 0 to 1 and $X2$ makes a transition from 1 to 0 simultaneously	$X1$ turns on at the same time $X2$ turns off
$X1 \rightarrow 1 + X2 \rightarrow 0$	$X1$ makes a transition from 0 to 1 or $X2$ makes a transition from 1 to 0	$X1$ is to turn on or $X2$ is to turn off
$X1 \rightarrow ?$ and $X2 \rightarrow ?$	$X1$ and $X2$ make a transition simultaneously, i.e., $X1$ and $X2$ go from 00 to 11 or 11 to 00 or 01 to 10 or 10 to 01	$X1$ and $X2$ change state simultaneously
$X1 \rightarrow 1$ WHILE $X2 = 1$	$X1$ makes a transition from 0 to 1 while $X2$ stays at 1, i.e., inputs $X1$, $X2$ go from state 01 to 11	$X1$ turns on while $X2$ is held on

TABLE III
EXAMPLES OF LABELS

Label Type	Examples
Standard	FIRST:, $31:, BEGIN-HERE:
Output	$Z00$:, $Z(00, 01/2, 11)$:, $Z110/3$:

for basic transition expressions to enable the designer to make quick and easy specifications. The notation '$X \rightarrow ?$' indicates the occurrence of a don't-care transition and essentially says that X makes a transition and "you don't care" if it is a '$0 \rightarrow 1$' or a '$1 \rightarrow 0$' transition. Further description of transition expressions is provided in Table II.

B. Statement Labels

In ACDL there are two different types of labels: standard labels and output labels. A standard label is essentially an identifier with the restriction that the first character of the label cannot be the letter "Z." The letter "Z" is reserved for the beginning character of output labels, and it is followed by the current output state of the circuit. Hence, the output label serves two major purposes; it indicates the next statement to occur in the design sequence while at the same time provides the designer with the present output state at that point in the sequence. The output state of the label is followed by a / and digit when it is necessary to distinguish a previous output label having the same output state. An output label may specify multiple output states where the next statement in the sequence is the same for each state. All labels are followed by a colon. Examples of standard and output labels are given in Table III.

C. Statements

The statements available in ACDL are briefly described in Table IV. Further explanation is provided as necessary.

1) Declare Statement: The input and output declarations indicate the number and names of the input and output variables required in the design. If no initial condition is explicitly shown for the variable, then an initial value of zero is assumed.

The constraints declaration (CONSTR:) indicates those input transitions that are not allowed. The input constraints are either shown explicitly with level and transition expressions, or for some of the more common occurring cases can be designated by special keywords. The keywords "Single Input Change" abbreviated SIC, assists the designer in problems which permit only single input changes. The keywords "All Unspecified Sequences" abbreviated AUS, restricts all sequences of input transitions which are not explicitly described in the design. This constraint is extremely useful where only a specific number of alternative sequences can occur. Level expression constraints, e.g., $X1 = 1$, restrict transitions to those input states which agree with the value of the expression. If there are no input transition constraints the keyword NONE is written.

The global declaration is used when there are certain transition conditions which arise frequently throughout the design and are independent of any particular I/O sequence. Instead of repeating the transition statement many times in the design specifications, the statements are listed once in a global declaration. The global list of transition statements have automatic linking designated, so that whenever a global transition occurs, an immediate branch is made to a specified point in the design sequence.

Examples of the design and declare statements are presented in Table V. In the declare statement, the initial conditions for the input variables are explicitly given (i.e., shown within parentheses) while for the output variable it is left to default to zero. The slash following the output change of the global declaration is the symbol used to designate automatic linking. This concept is further explained in Section V in conjunction with the list statement.

2) Transition Statement: The transition statement is used to show input transitions and may relate an input transition to an output change. Since the inputs to the circuit change at random, the input transitions specified by the transition expression are essentially test conditions for particular input transitions. An output change is an output variable(s) being assigned the value of a specified Boolean expression. The output change occurs only when the transition expression is satisfied. Examples of transition statements are shown in Table VI.

3) Link Statement: This statement provides a natural way to describe alternate behavior at a branch point in the design sequence.

TABLE IV
STATEMENTS OF ACDL

Statement	General Format Description	Use
Design	DESIGN optional accounting information of design #, designer's name, and date;	Begins ACDL Design
Declare	DECLARE INPUTS: input names with or without initial conditions CONSTR: keywords, level expressions or transition expression constraints OUTPUT: output names with or without initial conditions GLOBAL: list of automatic link transition statements;	Defines all input variables, output variables, constraint transitions and global transitions of the design.
Start	START;	Starting point for the design sequences. Invokes initial $I/0$ conditions and establishes initial state of machine.
Transition	transition expression; or transition expression$\Rightarrow$ output change with or without automatic linking specified;	Expresses the $I/0$ relationships of the critical input-output events of the design.
Link	LINK (test condition list) label list;	For branching in ACDL, where the test condition is an input level or transition test, causing a branch to the corresponding label.

Statement block	BEGIN;
	statement list
	END;
		For listing independent sequence paths resulting from a link or list statement.

List	LIST followed by a list of automatic link transition statements;	For branching when all test conditions lead directly to an output change.
Program end	END.	Designates the end of the sequence specifications and end of the design.
Comment	"any valid character string"	Adds clarification to the design.

TABLE V
EXAMPLES OF THE DESIGN AND DECLARE STATEMENTS

Statement Type	Example
Design	DESIGN 103, JOHN DOE, JAN 3, 1973;
Declare	DECLARE INPUTS: A(1), B(0) CONSTR: SIC, A = 1 & B = 1 OUTPUTS: Z BLOBAL: B $\rightarrow$ 0$\Rightarrow$ Z $\leftarrow$ 0/;

TABLE VI
EXAMPLES OF THE TRANSITION STATEMENT

Statement	Explanation
$A \rightarrow 1$;	Input A is to make a transition to 1
$A \rightarrow 0 \Rightarrow Z \leftarrow 1$;	Input A making a transition to 0 implies that output Z changes to 1 if not already 1
$(C \rightarrow 1) + (B \rightarrow 1) \Rightarrow G \leftarrow 1, R \leftarrow 0$;	Input C making a transition to 1 or input B making a transition to 1 causes output G to be replaced to 1 and R to be replaced to 0.

A test condition is either a transition expression, level relation, or the keyword ELSE (meaning all other unspecified conditions). Each test condition links the previous specified sequence to an alternate sequence path given by the label corresponding by position to the test condition.

Whenever an output change follows a test condition of the LINK, it is shown in the first statement of the new path. In this case the dummy term LINKTEST (abbrev. LK'T) is inserted as the transition expression for this transition statement. This implies that the LINK test condition causing the branch also causes the output change.

A special case of the link statement is the link unconditional written as: LINK label; . This case is primarily used for unconditionally branching to a previously specified statement.

4) Statement Block: The statement block provides a means of separating alternate sequence paths resulting from a link statement. The BEGIN and END statements act as separators which serve to segregate a block of statements from other statements. Statement blocks may be nested within other statement blocks. After a statement block has been completed, its sequence path continues with the next statement in the listing which does not belong to another statement block of the same nested level. Examples of the link statement and statement blocks are given in Table VII.

5) Automatic Linking and the List Statement: Automatic linking is designated by a slash, /, following the output change of a transition statement. This linking is accomplished by branching to the statement identified by an output label having the current output state. The designer must ensure that a unique and correct output label has been assigned to some statement. To distinguish between output states having the same value, but occurring at different points in the sequence, the designer follows the slash with a digit which must agree with the trailing digit of the correct output state label.

Automatic linking saves the designer having to explicitly specify a link statement and hence, improves the clarity of the specification

TABLE VII
EXAMPLE OF LINK STATEMENT AND STATEMENT BLOCKS

Statements	Explanation
LINK $(B \rightarrow 1, A \rightarrow 0)L1, L2$; L1: BEGIN; LK'T$\Rightarrow Z \leftarrow 1$; ⋮ END; L2: BEGIN; $C \rightarrow 1$; ⋮ END; $B \rightarrow 0$; ⋮	The 1st test condition transfers control to statement block $L1$ where the 1st statement says that the link test $B \rightarrow 1$ causes $Z \leftarrow 1$. After the 1st statement block is completed, control is transferred to the transition statement $B \rightarrow 0$. If none of the link statement test conditions are true for the current input transition, the sequence will not advance, but rather will remain at the link statement until a test condition becomes true for some later transition.

listing. It was the automatic linking feature which led to the development of the list statement. This statement is a special purpose link statement in which all test conditions lead directly to an output change. It does not specify a sequence of successively occurring transitions, but rather, it specifies a set of statements from which only one is selected to occur. The test conditions of all transition statements in the list are scanned concurrently, and only one test condition may be true at a time. When a test condition becomes true, its corresponding output change indicates the next statement in the design sequence via automatic linking. The transition statements within a LIST statement are separated by commas, while the end of the list is indicated by the LIST statement's semicolon. An example of the list statement is given in Table VIII.

6) Sequences: Statements in ACDL are interpreted in the sequential order in which they are specified, except when the physical sequence is interrupted by branching which results from explicit or automatic linking. The rules for interpreting sequences written in ACDL are

1) In a test condition of an ACDL statement, any undesignated input variables are considered as don't-cares in the specified input state transition.
2) The sequence will advance to the next statement for any input state transition which agrees with a test condition of the current statement.
3) The sequence remains quiescent (i.e., does not move) for any input state transition which does not agree with a test condition of the current statement.

Now that a description of the language has been presented, a simple and efficient technique for correctly specifying the operation of the sequential circuit will be described.

D. Critical Event

In studying the specification problem of asynchronous sequential circuits, it was found that only those sequences of input transitions which cause output changes are important in describing the basic operation of the circuit. Also it was observed that these sequences are relatively few compared to the total number specified in the flow table. These findings led to a specification technique for ACDL that provides a precise circuit description from which the complete synthesis can be carried out automatically.

Definition: A set of minimum length sequences of input states which cause the next output change and starts from the I/O state resulting from the previous output change is called a *critical event.*

A critical event may be an incompletely specified sequence, i.e., a sequence which contains don't-care variables in some states. In this case the critical event will actually represent more than one possible sequence resulting from the random changing of the don't-care variables. However, any intermediate states that are introduced by the don't-care variables will not affect the integrity of the critical event, i.e., these states neither cause an output change nor destroy any past history of the critical event.

For many asynchronous sequential circuits, the design conditions originate in the form of critical events, and it is from these that normally a word statement and flow table follow. ACDL, however, is a means to directly and formally specify the critical events. This is done by starting from the initial state of the circuit and listing the critical events which cause the first output changes. Continuing from these points in the sequences, subsequent critical events are listed until all possible output changes have been specified. The following design examples illustrate the use of this specification procedure.

Example 1: Design a circuit which has two inputs, OSC and BTN, and one output, Z. The input OSC is the output of a square wave oscillator, and BTN is a button which, when depressed, gates one and only one full width oscillator pulse to the output. An output pulse can occur only if the button depression overlaps the leading edge of an oscillator pulse. The inputs cannot change simultaneously [15].

The ACDL design is:

```
DESIGN 1: JOHN BROWN, AUG. 27, 1973;
DECLARE  INPUTS:   OSC, BTN
         CONSTR:   SIC
         OUTPUTS:  Z;
START;
BTN → 1;
OSC → 1 WHILE BTN = 1 ⇒ Z ← 1;
OSC → 0 ⇒ Z ← 0;
END.
```

TABLE VIII
AN EXAMPLE OF THE LIST STATEMENT

Example	Explanation
Z00: List X1 → 1⇒ Z1 ← 1/, X2 → 1⇒ Z2 ← 1/; ⋮ Z10: X2 → 1; ⋮ Z01: X1 → 1; ⋮	In the list statement either of the two listed input transitions can occur. If X1 → 1 then an automatic link is made to the statement having the output label Z10. Similarly, if X2 → 1 is true, then a branch will be made to Z01 upon completion of the output change Z2 ← 1.

There are usually many different ways to specify a design. For example, the critical events of Example 1 could have been described as:

```
     START;
L1:  OSC → 1 WHILE BTN = 1 ⇒ Z ← 1;
     OSC → 0 ⇒ Z ← 0;
     LINK (BTN = 0, ELSE) L1, L2
L2:  BTN → 0;
     END.
```

Example 2: Design a special asynchronous two-input coincidence detector in which the output turns on when both inputs are on concurrently, but turns off only when the input which rose first goes off. The inputs are not permitted to change simultaneously.

The ACDL design is:

```
DESIGN 2; "COINCIDENCE DETECTOR"
DECLARE  INPUTS:   A, B
         CONSTR:   SIC
         OUTPUTS:  Z;
START;
LIST
         A → 1 WHILE B = 1 ⇒ Z ← 1/1,
         B → 1 WHILE A = 1 ⇒ Z ← 1/2;
Z1/1:    B → 0 ⇒ Z ← 0/;
Z1/2:    A → 0 ⇒ Z ← 0;
Z0:      END.
```

Example 3: Design a combinational lock circuit which has two cipher signals ($X1$, $X2$) and a reset signal (R) as inputs and a single output (Z). The lock is to open ($Z = 1$) if there is a sequence of four consecutive changes of $X1$ while $X2$ remains on and is to stay open regardless of further changes of $X1$ and $X2$. If, however, $X2$ turns off, after two or three consecutive changes of $X1$ with $X2$ on, the lock will remain closed no matter what changes in $X1$ and $X2$ occur thereafter. Whenever the reset signal occurs, the lock closes (if not already closed) and the combination can be restarted. Only single input changes are permitted [16].

The ACDL design is:

```
DESIGN 3; "COMBINATION LOCK"
DECLARE  INPUTS:   X1, X2, R
         CONSTR:   SIC
         OUTPUTS:  Z
         GLOBAL:   R → 1 ⇒ Z ← 0/;
     START;
L2:  X1 → ? WHILE X2 = 1;
     LINK (X1 → ?, X2 → 0) L1, L2;
L1:  LINK (X1 → ?, X2 → 0) L3, Z0;
L3:  LINK (X1 → ?, X2 → 0) L4, Z0;
L4:  LK'T ⇒ Z ← 1;
Z0:  R → 0;
     END.
```

The primitive flow tables for Examples 1, 2, and 3 have 8, 7, and 22 rows, respectively. An algorithm to automatically translate ACDL designs to primitive flow tables was developed to provide a direct and simple mechanism for verifying the language and specification process. A detailed description of the ACDL translator and flow table algorithm is given in [7]. A system is also available to auto-

matically convert ACDL descriptions to the circuit design equations [4]–[7]. Further information on the availability of these programs' can be obtained from Dr. Tracey.

IV. CONCLUSION

ACDL is a compact, easy-to-use language for specifying the behavior of asynchronous sequential circuits which is more precise than word statements, and more natural and descriptive than flow tables. With ACDL, a design can be described in different ways by using the various features available in the language. In many problems, the critical event philosophy of listing only minimum sequences of input changes which cause output changes greatly reduces the amount of information needed for specification. This feature may enable the designer to handle some large problems (e.g., designs yielding 50–150 row flow tables) which would otherwise be too cumbersome for hand-construction of a flow table. However, as the number of primary inputs increase, the exponential rate of increase of input states and therefore, input sequences, causes problems to become inherently more complex.

ACDL is also a flexible system in terms of its problem versatility which includes designs for word statements, I/O sequences, switches flip-flops, counters, etc. However, it is best suited for describing those problems that originate as a small collection of critical events and are currently being described informally by word statements. Problems originating as a fixed set of I/O sequences are specified easily, but somewhat less naturally, than with present I/O sequence methods [6]–[7], since the I/O sequences have to be converted to the ACDL transition statements as opposed to a direct listing.

The ACDL system has been tested with many examples in an attempt to use every feature of the language and to verify the correctness of the specification process.

Presently, ACDL could be combined with existing synthesis programs to provide a completely automated synthesis system [4]–[7]. Future research will be examining the possibility of proceeding directly from an ACDL description to design or implementation equations without the classical intermediate steps. Also, ACDL provides an opening for further research in the area of language design for large asynchronous networks.

APPENDIX

FORMAL DEFINITION OF ACDL

To formalize the definition of ACDL the metalanguage Backus Normal Form (BNF) [17] is used which consists of the following symbols:

⟨*X*⟩	to be read as "the object named *X*;"
::=	to be read as "can be formed from;"
|	to be read as "or" (exclusive or).

The metalanguage construct takes on the following meaning: "the object named in the corner braces may be formed from the objects named or specified on the right." Concatenation of names or objects is implied by their juxtaposition in the construct.

The following ACDL definition is the accepted syntax of the current ACDL translator [7].

A. *Vocabulary of ACDL*

1) Symbols:

⟨letter⟩ ::= *A* | *B* | *C* | . . . | *Z* | # | — | @ | & | '
⟨binary digit⟩ ::= 0 | 1
⟨nonbinary digit⟩ ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨digit⟩ ::= ⟨binary digit⟩ | ⟨nonbinary digit⟩
⟨special character⟩ ::= " | / | ?
⟨separator⟩ ::= , | ; | : | . | (|)
⟨relation symbol⟩ ::= = | → | ⇒
⟨replacement op⟩ ::= ←
⟨logical op⟩ ::= ¬ | & | +

2) Constants and Identifiers:

⟨constant⟩ ::= ⟨number⟩ | ⟨level⟩ | ⟨transition⟩
⟨number⟩ ::= ⟨digit⟩ | ⟨number⟩ ⟨digit⟩
⟨level⟩ ::= ⟨binary digit⟩
⟨transition⟩ ::= ⟨long transition⟩ | ⟨short transition⟩
⟨long transition⟩ ::= 0 → 1 | 1 → 0
⟨short transition⟩ ::= →1 | →0 | →?
⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

3) Relations and Expressions:

⟨level relation⟩ ::= ⟨identifier⟩ = ⟨level⟩ | ⟨level relation⟩ & ⟨identifier⟩ = ⟨level⟩ | (⟨level relation⟩)
⟨transition relation⟩ ::= ⟨identifier⟩ = ⟨long transition⟩ | ⟨identifier⟩ ⟨short transition⟩ | ⟨transition relation⟩ & ⟨identifier⟩ = ⟨long transition⟩ | ⟨transition relation⟩ & ⟨identifier⟩ ⟨short transition⟩ | (⟨transition relation⟩)
⟨compound relation⟩ ::= ⟨transition relation⟩ WHILE ⟨level relation⟩ ⟨transition expression⟩ ::= ⟨transition relation⟩ | ⟨compound relation⟩ | ⟨transition expression⟩ + ⟨transition relation⟩ | ⟨transition expression⟩ + ⟨compound relation⟩ | ⟨dummy term⟩ | (⟨transition expression⟩)
⟨dummy term⟩ ::= LINKTEST | LK'T
⟨Boolean expression⟩ ::= ⟨logical factor⟩ | ⟨Boolean expression⟩ + ⟨logical factor⟩
⟨logical factor⟩ ::= ⟨logical term⟩ | ⟨logical factor⟩ & ⟨logical term⟩
⟨logical term⟩ ::= ⟨logical primary⟩ | ¬ ⟨logical primary⟩
⟨logical primary⟩ ::= ⟨level⟩ | ⟨identifier⟩ | (⟨Boolean expression⟩)

4) Statement Labels:

⟨label⟩ ::= ⟨standard label⟩: | ⟨output label⟩:
⟨standard label⟩ ::= ⟨letter (except Z)⟩ | ⟨standard label⟩ ⟨letter⟩ | ⟨standard label⟩ ⟨digit⟩
⟨output label⟩ ::= Z⟨output state⟩ | Z(⟨output state set⟩)
⟨output state set⟩ ::= ⟨output state⟩ | ⟨output state set⟩, ⟨output state⟩
⟨output state⟩ ::= ⟨output code⟩ | ⟨output code⟩/⟨number⟩
⟨output code⟩ ::= ⟨binary digit⟩ | ⟨output code⟩ ⟨binary digit⟩

B. *Statements*

1) Design Statement:

⟨design statement⟩ ::= DESIGN | DESIGN ⟨accounting information⟩
⟨accounting information⟩ ::= ⟨design number⟩ | ⟨design number⟩, ⟨designer's name⟩ | ⟨design number⟩, ⟨designer's name⟩, ⟨date⟩
⟨design number⟩ ::= ⟨number⟩
⟨designer's name⟩ ::= ⟨identifier⟩ | ⟨designer's name⟩ ⟨identifier⟩
⟨date⟩ ::= ⟨identifier⟩ ⟨number⟩, ⟨number⟩

2) Declare Statement:

⟨declare statement⟩ ::= DECLARE ⟨declaration type⟩ | ⟨declare statement⟩ ⟨declaration type⟩
⟨declaration type⟩ ::= ⟨input declaration⟩ | ⟨constraints declaration⟩ | ⟨output declaration⟩ | ⟨global declaration⟩
⟨input declaration⟩ ::= INPUTS: ⟨variable definition⟩ | ⟨input declaration⟩, ⟨variable definition⟩
⟨variable definition⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ ⟨initial condition⟩
⟨initial condition⟩ ::= (⟨level⟩)
⟨constraints declaration⟩ ::= CONSTR: ⟨constraints⟩
⟨constraints⟩ ::= NONE | AUS | SIC | ⟨transition expression⟩ | ⟨level relation⟩ | ⟨constraints⟩, ⟨transition expression⟩ | ⟨constraints⟩, ⟨level relation⟩
⟨output declaration⟩ ::= OUTPUTS: ⟨variable definition⟩ | ⟨output declaration⟩, ⟨variable definition⟩
⟨global declaration⟩ ::= GLOBAL: ⟨list⟩
⟨list⟩ ::= ⟨automatic link transition statement⟩ | ⟨list⟩, ⟨automatic link transition statement⟩

3) Start Statement:

⟨start statement⟩ ::= START

4) Transition Statement:

⟨transition statement⟩ ::= ⟨basic transition statement⟩ | ⟨automatic link transition statement⟩
⟨basic transition statement⟩ ::= ⟨transition expression⟩ | ⟨transition expression⟩ ⇒ ⟨output change⟩
⟨output change⟩ ::= ⟨identifier⟩ ← ⟨Boolean expression⟩ | ⟨output change⟩, ⟨identifier⟩ ← ⟨Boolean expression⟩
⟨automatic link transition statement⟩ ::= ⟨transition expression⟩ ⇒ ⟨output change⟩ ⟨autolink⟩
⟨auto link⟩ ::= / | /⟨number⟩

5) Link Statement:

⟨link statement⟩ ::= ⟨link conditional⟩ | ⟨link unconditional⟩
⟨link conditional⟩ ::= ⟨tests⟩ ⟨branch points⟩
⟨tests⟩ ::= (⟨test condition⟩ | ⟨tests⟩, ⟨test condition⟩

⟨test condition⟩ ::= ⟨transition expression⟩ | ⟨level relation⟩ | ELSE
⟨branch points⟩ ::=)⟨single label⟩ | ⟨branch points⟩, ⟨single label⟩
⟨single label⟩ ::= ⟨standard label⟩ | Z⟨output state⟩
⟨link unconditional⟩ ::= LINK ⟨single label⟩

6) *Statement Block:*
⟨statement block⟩ ::= ⟨beginning⟩ ⟨statement list⟩ ⟨ending⟩
⟨beginning⟩ ::= BEGIN; | ⟨label⟩ BEGIN;
⟨ending⟩ ::= END | ⟨label⟩ END

7) *List Statement:*
⟨list statement⟩ ::= LIST ⟨list⟩

8) *Comments:*
⟨comment⟩ ::= " ⟨almost anything⟩"
⟨almost anything⟩ ::= ⟨any string of valid system 360 characters which does not contain a "⟩

C. *Program Structure:*
⟨program⟩ ::= ⟨program head⟩ ⟨statement list⟩ ⟨ending⟩.
⟨program head⟩ ::= ⟨design statement⟩; ⟨declare statement⟩; ⟨start statement⟩;
⟨statement list⟩ ::= ⟨statement⟩ | ⟨statement list⟩ ⟨statement⟩
⟨statement⟩ ::= ⟨basic statements⟩ | ⟨statement block⟩;
⟨basic statement⟩ ::= ⟨transition statement⟩; | ⟨link statement⟩; | ⟨list statement⟩; | ⟨label⟩ ⟨basic statement⟩

REFERENCES

[1] E. J. McCluskey, *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill, 1965.
[2] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.
[3] Y. H. Chuang, "Transition logic circuits and a synthesis method," *IEEE Trans. Comput.*, vol. C-18, pp. 154–168, Feb. 1969.
[4] R. J. Smith, J. H. Tracey, W. L. Schoeffel, and G. K. Maki, "Automation in the design of asynchronous sequential circuits," in *1968 Spring Joint Computer Conf., AFIPS Conf. Proc.*, vol. 32, 1968, pp. 55–60.
[5] D. G. Raj-Karne, "A method for generating UTS assignments with an iterative state transition algorithm," Ph.D. dissertation, Univ. of Missouri-Rolla, Rolla, Mo., 1972.
[6] R. J. Smith, II, "Synthesis heuristics for large asynchronous sequential circuits," Ph.D. dissertation, Univ. Missouri–Rolla, Rolla, Mo., 1970.
[7] G. M. Bednar, "An asynchronous circuit design language system," Ph.D. dissertation, Univ. Missouri–Rolla, Rolla, Mo., 1972.
[8] R. A. Altman, "The computer aided generation of flow tables for asynchronous sequential circuits," M.S. thesis, Univ. of Missouri–Rolla, Rolla, Mo., 1968.
[9] T. C. Bartee, I. L. Lebow, and I. S. Reed, *Theory and Design of Digital Machines*. New York: McGraw-Hill, 1962, p. 324.
[10] H. Schorr, "Computer-aided digital system design and analysis using a register transfer language," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 730–737, Dec. 1964.
[11] Y. Chu, "An ALGOL-like Computer Design Language," in *Commun. Ass. Comput. Mach.* vol. 8, pp. 607–615, Oct. 1965.
[12] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.
[13] J. R. Duley and D. L. Dietmeyer, "A digital system design language (DDL)," *IEEE Trans. Comput.*, vol. C-17, pp. 850–861, Sept. 1968.
[14] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.
[15] G. A. Maley and J. Earle, *The Logic Design of Transistor Digital Computers*. Englewood Cliffs, N.J.: Prentice-Hall, 1963, p. 261.
[16] M. Krieger, *Basic Switching Circuit Theory*. New York: Macmillan, 1967, p. 169.
[17] W. M. McKeeman, J. J. Horning, and D. B. Wortman, *A Compiler Generator*. Englewood Cliffs, N. J.: Prentice-Hall, 1970.

Sequency Domain Design of Frequency Filters

A. E. KAHVECI AND E. L. HALL

***Abstract*—Each discrete transform has its advantages over other transforms. Fourier domain filters are easier to design than Walsh domain filters. But Walsh transforms can be computed much faster than Fourier transforms. This paper considers the problem of designing a Walsh domain filter, given a Fourier domain filter. Same procedure can be applied to other transforms. Given a Slant filter its equivalent Fourier filter or Walsh filter can be found.**

Index Terms—**Equivalent filtering, filter design, Fourier transforms, Karhunen–Loeve transforms, slant transforms, unitary transforms, Walsh transforms.**

Manuscript received June 30, 1972; revised January 31, 1973. This paper is a revised version of "Frequency Domain Design of Sequency Filters" which was published in the *1972 Walsh Functions Symposium Proceedings*.

The authors are with the Department of Electrical Engineering, University of Missouri at Columbia, Columbia, Mo.

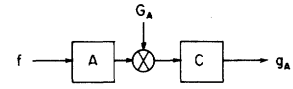

Fig. 1. Filtering with transforms A and C.

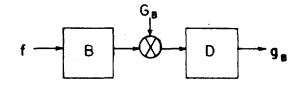

Fig. 2. Filtering with transforms B and D.

INTRODUCTION

Transform techniques are widely used in signal and image processing. It is also a very well-known fact that filtering operations are easier to design with some transforms than with others. As an example, filter design in the Fourier domain is much easier than in the Walsh domain because a large background of frequency domain design information is available. One may, therefore, make use of the superior properties of each transform by performing each operation in the transform domain which offers the greatest ease and the fastest time.

Fig. 1 is the block diagram of a generalized discrete filtering system using transforms A and C. Fig. 2 is the block diagram of a generalized filtering system using transforms B and D. The filtering result using transformations A and C may be expressed as $g_A = CG_AAf$ and using transformations B and D as $g_B = DG_BBf$. The only condition on A, B, C, and D is that they must be nonsingular with real or complex elements. Equivalent filtering would give $g_A = g_B$. Then $CG_AAf = DG_BBf$ and for a given G_A we can solve for $G_B = BCG_AAD$. For unitary transforms $C = A^{-1} = A^{*T}$ and $D = B^{-1} = B^{*T}$.

Equivalent filtering opens ways to doing filter design in one domain and computation in the other domain. A Karhunen–Loeve filter, therefore, may be designed in the Karhunen–Loeve domain and implemented in the Walsh domain or vice versa. If A is a complex unitary transform and B is a real unitary transform then $A^{-1}G_AA$ is required to be real for G_B to be realizable.

DESIGN OF SEQUENCY FILTER USING FREQUENCY INFORMATION

It is well known that the fast Walsh transform is computationally advantageous over the fast Fourier transform. However, Walsh transform filters are not as easy to design as simple frequency domain fast Fourier transform filters. To take advantage of both the simple design techniques of the fast Fourier transform and the decreased computation time of the Walsh transform the following problem is considered: given the discrete Fourier transform of a desired spatial filter, derive an equivalent Walsh sequency filter.

Pratt [2] has shown that discrete Weiner Filtering may be implemented by any unitary transformation, rather than just the Fourier transform.

The purpose of this paper is to describe methods for designing Walsh transform filters given the equi-spaced discrete frequency domain specifications. This solution allows one to use the fast Fourier transform for experimentation with the design of a digital filter for signal enhancement, then implement the filter with a faster computational method.

In the next section, equivalent Walsh transform filtering is described. Then computational problems are considered.

The block diagram of a Fourier filtering system is shown in Fig. 3. The block diagram of Walsh domain filtering is shown in Fig. 4. For a one-dimensional system, the input column vector, r, is first discrete Fourier (DFT) transformed, producing the DFT of s. Next, a point-by-point multiplication in the frequency domain is performed. Finally, the inverse Fourier transform of the filtered input