A Comparison of Register Transfer Languages for Describing Computers and Digital Systems.

M.R. Barbacci
Department of Computer Science
Carnegie-Mellon University

March 1973

This Research was supported by the National Science Foundation under grant no. GJ 32758X.

I would like to thank Professors Gordon Bell, Allen Newell, and Daniel Siewiorek for their comments and suggestions during the preparation of this report.

Abstract

Different notations have been proposed over the years to describe Register Transfer (RT) systems. They have met with varying degrees of success and to provide a direct comparison of them is a difficult task. One of the reasons for this is the different views of the RT level of design held by the proponents of the languages.

The approach used this paper is to provide a common ground, or kernel, for the entities the languages try to describe. The starting point is, since the RT level is an abstraction of the Switching Circuit level, the ability to describe systems at the lower level. Then the concepts introduced by the RT level of design, namely, the selective control of the activities, is presented. At this point, a clasification of the languages into Procedural and Non-procedural is provided.

The next step in the study deals with the ability of the languages to denote the asynchronous control modules as defined in [M07,M11]. The next step deals with their usefulness to describe complex systems (e.g., Instruction Set Processors) and the way a description is organized to provide different levels of detail. A few remarks are made about their implementation for simulation and automatic design systems. Finally, the languages are compared against a set of requirements for writing behavioral and structural descriptions of digital systems.

Table of Contents

Tables

INTRODUCTION

Although the RT level is a perfectly valid level of design, it has not been fully defined and understood (it is not a problem of youthfulness, the level was recognized as such in the early 50's). As a consequence we do not have a proven and accepted (complete?) set of primitive elements. Also, there is no accepted design form/style, and usually informal flow-chart and data path diagrams are used. The result is that a designer, working at the RT level must ocasionally descend one level (to the gate level) to describe precisely a particular piece of hardware. Contrast this with the gate level of design, with the traditionally accepted set of primitives (AND, OR, FLIP-FLOP etc.) where it is indeed rare for a designer to have to describe a gate in terms of diodes or transistors (the Circuit Level).

RT languages are similar to most programming languages since both carry out register assignments. The special nature of hardware suggests that it could be useful to at least have a special notation, even though programming languages such as FORTRAN can be used for this purpose. The need for such a notation is due to the influence that it has upon the designer, simultaneously limiting his intellect (a system may be more easily described in one language than in another) and enhancing it (a notation provides a formal mechanism to generate and transmit knowledge). The main motivation, is that: "Complexity diminishes and clarity increases to a marked degree if algorithms are described in a language in which appropriate control structures are primitive or easily expressible" [R06].

The fundamental properties of hardware systems dictate the properties an RT language must possess. In hardware systems many activities occur concurrently. Thus it is important to have a natural way of describing parallelism. Other characteristics of hardware systems are the non-recursive nature of their operation, and timing issues that must be considered. Since machines are designed from logically disjoint functional units*, a way of describing interactions between units, both as subroutines (hierarchical relationships) and as coroutines (symmetric relationships), is required. The behavior of the system is described by sequences of actions or activities, (where control flows along selected paths - an abstraction), and mechanisms required to guarantee the harmonious cooperation (synchronization) between activities.

Control operations in the language should provide only the information necessary to understand the behavior of the system and should imply the actual implementation as little as possible. The efficiency of a RT language depends upon the generality of the control operations, i.e. whether a given operation exists as a primitive in an RT set or is described as a composition of more primitive operations.

---

* Even when the primitives were gates and flip-flops, the complexity of a computer made it impossible to view the machine as a large network of components, instead, designers would think in terms of busses, shift registers, adders etc.

1.0 -- THE DESIGN LEVELS

Five major levels exist in the digital systems hierarchy [R02], for which we are interested in formally defining, using notations, and computer recognized languages. They are:

PMS level (System level).- The top level of description, evaluates the gross properties of the computer system. Its elements are processors, memories, switches, peripheral units etc, and the parameters are costs, memory capacities, information flow rates, power etc.

Programming level.- The basic components are the interpretation cycle, the machine instructions, and operations (which are defined at the RT level). The behavior of the processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules. Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends solely on the initial conditions and the particular program.

Register Transfer level (Functional level).- Data flow and control operate in discrete steps. A combination of switching circuits is used to form registers, register-transfers and other data operations. The elements (registers) are combined (transformed) according to some rule and then stored (transfered) into another register. The rules of transformation can be almost anything, from simple transfers to complex logical and arithmetic expressions.

Switching circuit level (Sequential and Combinational sublevels).- The system structure is given by a collection of gates and flip-flops, and the behavior by a set of boolean equations. Timing is carried out at a finer degree than at the preceding level, a time unit being usually on the order of a gate delay.

Circuit level.- Gates are described as some interconnection of diodes, transistors, resistors, etc. according to electrical circuit laws. Most of the discrete properties of the previous two levels are lost, and timing is carried out at a finer degree, where transient behavior is usually a very important consideration.

Part of the characterization of a System Level, (as described above) is the existance of a notation for representing the system, that is, the components, modes of combination and laws of behavior. The three intermediate hardware levels: Programming, Register Transfer and Switching can all be described using a single language. In fact, conventional programming languages have been used. The issue is, however, how much they must be changed to reflect parallelism, timing and the structure of the object being represented.

2.0 -- THE LANGUAGES

Four languages were selected on the basis of their characteristics as representative of a class of languages. They are:

1) CDL [L06,L07], one of the most successful, it has been in use for several years and has been implemented for simulation and design automation purposes [S12]. CDL is an excellent representative of the class of non-procedural languages (the distinction is given below).

2) APDL [L08], a representative of the class of Algol+ languages [L08,L12,L19,L25], i.e. languages based on Algol with extensions to handle timing and register variables.

3) APL [L16,L17], selected because of the richness of the set of data operators and its facility to handle arrays. A major drawback is the lack of facilities to describe parallel activities. It is used in the ALERT System [S05,S06] developed at IBM as a front end for their Design and Logic Automation Systems [S04,S18].

4) ISP [R02,L04], developed initially to describe the primitives of the programming level of design, handles concurrency and sequencing of activities in a simple fashion and provides an adequate set of data and control operators.

Non-procedural languages [L01,L05,L06,L09,L22,L24] attach no meaning to the lexicographical ordering of statements describing the operation of the system. Statements are associated with some sort of "label" describing the condition for execution (activation) of the operation described by the statement(s). They can be written as a table, where each entry consists of a label (activation conditions) and a set of activities.

Procedural languages impose an explicit ordering in the activities, given by the sequential ordering of the statements. The activation of activities is conditioned by the completion of the preceding ones. Concurrent activities are described in "time blocks" and the description is then a list of these blocks.

3.0 -- CHARACTERIZATION OF THE RT LEVEL

The RT level is a generalization of the Switching Circuit level. The structural elements are arrays of identical subsystems belonging to the Switching level, i.e., registers are made of flip-flops and gates, driven by clocks (or clock-like signals). The behavior is described by transformations (functions) and transfers between registers. The key element that sets this level apart from the lower level is the appearance of control (the ability to perform these transformations and transfers in a selective way) as an explicit entity. Section 3.1 describes the primitive components at the Switching Circuit level. The generalization of these components at the RT level is presented in section 3.2.

3.1 -- CARRIERS AND OPERATORS.-

Operators are entities that produce information by transformation of bit patterns to which a meaning has been assigned. These bit patterns reside in *carriers*, which are the entities used in storing and transmitting the information to and from the operators.

Operators take their input from carriers, perform their function on the data and present their outputs onto carriers. At the Switching Circuit level, operators range in complexity from simple transfer paths between carriers, to logical gates, to combinational networks.

Carriers can be divided into two types, according to the latency of the information stored in them. Thus, we have memory components where the information stored in them is kept over periods of time; and we have memoryless carriers, where the information is of a more transient nature. An example of the latter is wires coming out of a combinational network.

It would not be possible to make a clear distinction between these two types of carriers on the basis of their function in transmitting or storing information. There are many cases in which the distiction would be rather foggy. For instance, a bus is used to transmit information i.e. a carrier, but it also keeps it (a memory function) for a period of time adequate for the information to be collected at the receiving end. On the other hand, a set of flip-flops connected as a shift register provides both a storage and transmission function.

Among the physical operators and carriers we have the individual gates (AND, OR, NOT, EXOR, EQUIV, NAND, NOR, etc) and flip-flops of different types (RS, JK, D, etc.). These operators and carriers can be combined to form combinational and sequential networks.

At the Switching level, notations are based on Boolean equations or Logic diagrams and there seems to be general acceptance of the industrial or military standards currently in use. The use of logic equations allows a more concise representation of the system but does not provide for physical configuration descriptions. Logic diagrams provide a pictorial representation and allow the description of elements required by the circuits (physical restrictions) but not by the logic.

3.2 -- ARRAYS.-

The first specialization introduced by the RT level is the generalization of the operators and carriers, i.e. arrays of (identical) operators performing simultaneously on the information held in arrays of (identical) carriers. At this point, carriers become hierarchically organized information structures, in which each level consists of a number of subcarriers, all identically organized. This decomposition eventually yields elementary carriers that can not be decomposed further (bit carriers).

This abstraction allows the designer to reduce the complexity of his task by using logic equations to describe the network (Logic diagrams are less useful since the objective is the reduction of detail, specially when logically irrelevant). This implies that operators and operands are now actually arrays of identical entities. Subcarriers are referred to by some subscripting or array notation (similar to those in most programming languages). A concatenation operator is used to represent compound carriers. RT languages usually have the following elements:

- *Registers* are vectors of components whose values can be characters from an arbitrary alphabet. They are declared by giving the name of the register, the dimensions, and the size of the alphabet (defaulted to 2 in most cases, i.e. bits). Dimensions are given in brackets or parenthesis, as a list of subscripts or element names. Abbreviated lists are indicated by bound-pairs using some *range* operator, for instance, 1:5 stands for the list 1,2,3,4,5.

- *Subregisters* are part of registers and are usually declared with their own name and subscript specifications.

- *Compound registers*, formed by concatenation of several registers and subregisters can be declared. The declaration usually provides a name and subscript specifications.

- *Arrays* of registers are declared by providing a name, dimensions, and the specification of the individual registers (element names and base).

Table I -- Carrier declarations

	CDL	APDL	APL	ISP
Base	binary	general	general (1)	general
Register structure declaration	list of names or bound-pairs (4)	bound-pair	number of elements (2)	list of names or bound-pairs (4)
Register array dimensions	one dimension (3)	multiple dimensions (Algol)	multiple dimensions (6)	multiple dimensions
Subregisters declaration	mnemonic subscripts (5)	general (one dimension)	no	general
Compound registers declaration	general (one dimension)	general (one dimension)	no	general
Memoryless carriers	one dimensional	no	no (7)	multidimensional
Special carriers	LIGHT,SWITCH DECODER			

(1) Binary in ALERT
(2) Indexing is from 0 to N-1 or 1 to N depending on system parameter.
(3) Requires explicit use of memory address register.
(4) Numbers represent element names and not relative positions (as subscripts) along a given dimension.
(5) Mnemonic subscripts can be used to represent ranges of subscripts. Does not allow independent naming of subregisters.
(6) Up to two dimensions in ALERT.
(7) Macros are used for this purpose in ALERT.

Carriers are accessed through their names and the specification of the subcomponents by some subscripting mechanism. There are several modes of specifying this subscripting, depending on the number of dimensions and the number of subcomponents accessed. The following are typical RT language properties:

- Single characters (for instance, bits) are accessed by providing a full set of subscripts, one for each dimension up to the individual character.

- Complete registers are accessed by their names alone (defaulting the list of individual component names and bound pairs). Register arrays require specification of the subscripts up to the individual register.

- Partial registers are specified by providing a list of component names and bound pairs, in the order in which access is desired (not necessarily the order they occupy in the register).

- Multiple registers can be specified by lists of component names and bound pairs along the dimensions of the register array.

Table II -- Carrier accesses

	CDL	APDL	APL	ISP
Single character	Y	Y	Y	Y
Full register (1)	Y	Y	Y	Y
Scattered register	Y	Y	Y (2)	Y
Scattered array	N	N	Y (2)	Y
Expression subscripts	N (3)	Y	Y (4)	Y

(1) Total register access by name alone.
(2) Only for constant subscripts in ALERT.
(3) Always requires use of the Memory Address Register used in the declaration.
(4) Only for single bit access in ALERT.

The primitive operators are classified into logical, arithmetic vector, and special. The symbols used to represent the operators are either special characters or combinations of letters (names).

Expressions are formed by combination of constants, operators, and operands. Rules of precedence are usually as in Algol, with the notable exception of APL (right to left precedence). The value of the expression represents the signals at the output of the combinational network that performs the operation.

New operators, beyond those of the Switching level, are introduced such as a *transfer* (←) operator to describe the loading of information in a register. Several complex operations are now taken as primitives, for instance, shift and rotation.

Table III -- Primitive operators

	CDL	APDL	APL	ISP
Logical	¬ ∨ ∧ ⊕ ≡	¬ ∨ ∧ ⊕ (1)	¬ ∨ ∧	¬ ∨ ∧ ⊕ ≡ (1)
Arithmetic	add sub	+ −	+ − x ÷ * |	+ − * ÷
Vector	shift rotate		take,drop, rotate, (2)	
Special operators	countup countdn	exchange		:= (3)
Concatenation	−	,	,	□
Transfer	←	←	← (4)	←

(1) Since the registers can have elements in any base, the logical operators have been generalized in APDL and ISP, to handle non binary values.
(2) Operators that modify dimensions of variables are excluded in ALERT.
(3) Expressions can be followed by a modifier, providing more information about the meaning and interpretation of the operands and operators. A modifier consists of a data type specification or an operation mode enclosed in curly brackets, e.g.:

B+C{1's complement}

(4) Special operators, ←SET and ←RESET, are used in ALERT to select the SET and RESET inputs to the flip-flops.

3.3 -- CONDITIONAL ACTIONS.-

At the switching circuit level the systems are considered as networks of elements performing their activities continuously. All components are considered equally relevant to the specification of the system state. When the system is of more than trivial size, the difficulty in comprehending the behavior increases, and new abstractions are introduced.

In RT level descriptions, the elements are assumed to be going through periods of activity and rest. Clearly, the physical elements are always active. The abstraction merely consists in specifying, at different times, the components were "interesting" activities are taking place. For instance, storing new information in a carrier or changing the inputs to an operator.

This abstraction is reflected in the description by the use of conditional operators that specify the subsystems that are to be considered "active" (in this new, abstract sense), and also the conditions for this transition of activity state to take place.

Conditional statements are used to select actions under a condition generated by a test network. The network is described by a Boolean or relational expression. Single bit tests are usually simplified by using the bit itself as a Boolean expression. This can be extended to an empty/non-empty test for a whole register.

Conditional statements, in most languages, can be nested to any depth (In APL, conditional statements are implemented by testing and branching).

Table IV -- Conditional operators

	CDL	APDL	APL	ISP
Relational operators	= ≠	= ≠ < > ≤ ≥	= ≠ < > ≤ ≥	= ≠ < > ≤ ≥
Conditional statements	IF THEN ELSE	IF THEN ELSE	→ (1)	(⇒) (2)

(1) The branch operator → is used to select a label or line number. The right hand side evaluates to a vector and the branching is to the line number given by the first element of the vector. The next line is taken if the vector is empty. The ALERT (simplified) syntax is: IF (boolean expression) GO TO label.
(2) Corresponds to the IF ... THEN ...; case of Algol:
(boolean expression ⇒ actions)

4.0 -- CONCURRENT AND SEQUENTIAL ACTIONS

At the switching circuit level, logic equations describe the nature of the input signals to the individual flip-flops, including the timing and control pulses.

In RT level descriptions, the basic building blocks are the ***simple actions***, for instance, the transfer to a register of the contents of another register or the value encoded in the output lines of a combinational network. Simple actions can be executed in parallel if there is no conflict in the use of resources (e.g., simultaneous transfers to the same register, or simultaneous use of a bus to transmit different pieces of information).

An accurate description of a hardware system must specify the time required to perform the operations, usually in terms of a basic cycle time or time unit. ***Time blocks*** are used to group actions that are performed concurrently (taking a certain number of time units to complete). A succession of time blocks may be used to define sequential operations. The selective activation of components, introduced in the previous section, allows the representation of the timing and control signals as the condition for the activation of the transfer operation (represented as an assignment statement). The data (logically relevant) part of the original equations forms the expression whose "value" is transferred to the register.

At the RT level, concurrent activities are described by allowing them to be activated simultaneously (i.e. under the same conditions). Synchronous or asynchronous systems differ mostly in the way the activation conditions are specified, i.e. whether it is a signal coming from a central clock or a signal produced at the completion of the preceding activities or a combination of both*.

The concept of selective activation of elements in the system introduces the first classification of RT languages into Procedural and Non-Procedural Languages.

- - - - - - - - - - - - - - - - - - -

* Clocks are special units used to generate sequences of pulses that can be used to step the actions of the system. Clock variables when used in expressions are considered as Boolean variables whose value alternates between 0 and 1 automatically, according to some frequency.
Synchronous systems operate in synchronism with the clock pulses, and the length of the time blocks is usually the period of the clock.
Asynchronous systems do not usually require a clock. Each time block can take different time intervals, depending on the operations performed.

4.1 -- SEQUENCING IN NON-PROCEDURAL LANGUAGES.-

Non-procedural RT languages use some special ***control variables*** to form a ***label*** describing the conditions for execution of a time block. Sequencing is performed by modifying the control variables used in the labels, hence enabling or inhibiting the activation of the time blocks. The modification of these control variables can be made explicit, as part of the operations performed in the time block or can be generated by an independent entity of the system, working concurrently, for instance a clock or a finite state automaton.

Some languages, like CASSANDRE [L01] and DDL [L09], make use of finite state machines as the controlling entities for the operations. They use ***state registers*** to store information about the ***state*** of the system. Special operators are used to describe the sequencing of operations by testing and modifying the value i.e. the state held in the state registers.

Table V -- Sequencing in non-procedural languages

	CDL	CASSANDRE [L01]	DDL [L09]
Time block labels	control expression	state value	state value
Sequencing	assignments to control variables	assignments to state registers	assignments to state registers
Clock variables	Y (1)	Y (2)	Y

(1) Used with other variables in control expressions.
(2) Used to step the operations inside a block defined by a state label, thus providing a hierarchy of time blocks.

4.2 -- SEQUENCING IN PROCEDURAL LANGUAGES.-

In ***procedural*** RT languages, systems are described as lists of statements representing the actions. Parallel actions are grouped into time blocks and sequential actions are described as lists of time blocks. Sometimes this definition can be used recursively (by using some type of brackets), to build complex time blocks (a group of concurrent sequences)

Conditional activities can be of two types, depending on the interpretation of the test. In some cases, the test is continuously performed and the actions are initiated every time the condition becomes true, i.e., there is a monitoring function being performed concurrently with the rest of the activities. In other cases, the test is performed once, and the actions are executed or skipped depending on the result.

Time blocks are described as a list of conditional and unconditional actions. All non-monitoring statements are executed once and in the order in which they are written (unless modified by branches). Monitoring statements are assumed to be permanently active and performing the test. They cease to be active when the end of the time block is reached (the scope of a monitoring statement is that of the time block in which it occurs).

Delay operators are used to hold up the execution of a sequence for a number of time units. Delays are similar to an empty time block that takes the appropriate number of time units.

Since APL does not allow the description of concurrent activities, ALERT provides several conventions to that effect:

- Concurrency of simple actions can be determined automatically by the system. The algorithms used will try to group the sequence of simple actions into the minimun number of time blocks. For each sequence of time blocks obtained in this fashion, a ***sequence counter*** is provided to step the machine through the time blocks. Sequencing is obtained by stepping the counter, by forcing a value into it, or inhibiting the counting when waiting for a condition to become true.

- At a higher level, concurrency is obtained by dividing the description in ***microprograms***, capable of operating concurrently. Each microprogram specifies whether the system is to provide the grouping of simple actions or not. In the latter case the statements are assumed to be active simultaneously (as in non procedural languages, with the enabling and disabling of actions explicitly described in the microprogram).

Table VI -- Sequencing in procedural languages

	APDL	APL	ISP
Sequential actions	Y (1)	Y	Y (2)
Concurrent actions	Y (3)	N (4)	Y (5)
Monitor statements	IF EVER.... THEN....;		(wait....⇒....)
Recursive description of concurrent and sequential actions	N (6)	N	Y (7)
GO TO operator	Y	Y	N
Delay operator	wait		wait
Clock variables	PULSE (8)		

(1) List of time blocks separated by ";".
(2) List of actions using the term "**next**" as delimiter.
(3) Concurrent actions are described in time blocks of the form:

n TIME BEGIN . . . . . END

to describe a statement or group of statements requiring n cycle times.
(4) ALERT provides some conventions to describe concurrent activities.
(5) In ISP concurrency of actions is assumed by default. Concurrent actions are described as a list of statements using the ";" as delimiter.
(6) Time blocks can not be nested. IF EVER statements are taken as declarations, local to an Algol block and can be nested to any depth.
(7) Parenthesis are used to group statements and to indicate the scope of conditional activities. Complex sequential and concurrent activities are described using ";", "**next**", "(" and ")" in a recursive way.
(8) Pulses are boolean variables that are automatically reset to 0 one cycle time after they are set to 1.

5.0 -- ASYNCHRONOUS CONTROL PRIMITIVES AND THEIR REPRESENTATION

The Computation Structures Group at MIT is involved in the investigation of formal descriptions of computer systems. In particular the definition of asynchronous control structures [M07], generating signals to direct the actions of data operators in a data flow structure.

Nine types of control modules were identified [M11] as sufficient to implement simple control structures. The modules are interconnected by direct links that are capable of transmitting ***ready*** and ***acknowledge*** signals used to indicate a request for some operation or the completion of a requested action, respectively. The reasons behind the selection of this particular set of modules, their representation in terms of Petri–nets, and other formal considerations do not belong in this paper, and our concern will lie on the presence or absence of similar control function as primitives in the RT languages. The Nine modules are:

– The ***source*** module generates a ready signal each time an acknowledge signal is received.

– The ***sink*** module responds with an acknowledge each time a ready signal is received.

– The ***sequence*** module causes ready/acknowledge cycles to occur on each of its output links, one at a time, for each ready signal received on the input link.

– The ***wye*** module is used to permit several actions to proceed concurrently by generating simultaneous ready/acknowledge cycles on its output links.

– The ***junction*** module causes an operation to wait for several independent events to occur. It waits until a ready signal has been received on each of its input links before generating a ready/acknowledge cycle on its output link.

– The ***trigger*** module implements the basic control mechanism for an operator in a chain of operators that processes data in pipeline fashion. It generates a ready/acknowledge cycle (the stage operations) when a ready signal is received, provided that the next stage is idle (the acknowldge signal of the current stage is used as the ready signal for the next stage).

– The ***decision*** module permits actions by a control structure to be affected by external conditions. When a ready signal is received, it performs a test, and depending on the result a ready/acknowledge cycle is generated on one of its output links.

–The ***union*** module causes a ready/acknowledge cycle to occur on its output link for a ready signal on any of its input links.

– The *arbiter* module causes a ready/acknowledge cycle on one of the output links for each ready/ackowledge cycle in the corresponding input links. It provides an interlocking mechanism so that only one of a set of activities can be executed at a time.

Table VII -- Asynchronous control primitives

	CDL	APDL	APL	ISP
Source	-----	PULSE variables	-----	-----
Sink	-----	-----	-----	-----
Sequence	(4)	;	lines	next
Wye	control labels	TIME BLOCK IF EVER stmts.	-----	;
Junction	(1)	END (Algol)	-----	next
Trigger	(1)	(2)	(2)	(2)
Decision	IF THEN ELSE DECODER	IF THEN ELSE SWITCH	... ⇒ ...	(⇒)
Union	(3)	Procedure call	Function call	Process call
Arbiter	-----	-----	-----	-----

(1) Conjunction of signals as control expression (label).
(2) Conjunction of signals as condition in conditional statements.
(3) Disjunction of signals as control expression (label).
(4) Output of decoder/counter network as control expression (label)

6.0 -- BEHAVIORAL AND STRUCTURAL DESCRIPTIONS

Several levels of detail can be used in a system description. They range from ***behavioral*** descriptions (in which the properties of the system are specified in terms of the input/output relationship between variables - a black box approach), to ***structural*** descriptions (where the system is described in terms of the real hardware components and their interconnections). Intermediate, ***functional***, descriptions represent the system in terms of the actual components and their functional relationship or algorithm.

Behavioral descriptions are closer to conventional programs in most programming languages. Complex expressions and operators are allowed for simplicity in the description, e.g., the use of arithmetic expressions as subscripts. Variables and operators do not necessarily have a hardware counterpart and timing details are usually ignored i.e. operations are assumed to take no time since the intention is only to show the algorithm at a gross level.

Functional descriptions are closer to the real hardware. They describe the system as an algorithm in terms of the real registers and components of the machine. The operators may or may not be hardware primitives and expressions can be of a complicated nature. Timing and concurrency are taken into account.

Structural descriptions represent the system in terms of the hardware components. Operators have physical counterparts (i.e. they are primitive) and the descriptions tend to give more detail than the other two levels. Being closer to the physical implementation, timing is described in terms of clock pulses or event completed signals.

Non-procedural languages tend to impose more restrictions on the user. The sequence of operations (the algorithm) must be described by providing the timing and the conditions to execute the operations. This kind of detail is irrelevant if the designer only wants to describe the algorithm in terms of input/output sequences, without any consideration to the actual clock pulses or state register values.

Procedural languages are better suited for behavioral descriptions. The algorithm is described as a sequence of steps (as in conventional programming languages) and the details of control can be ignored. At the lower level, where these details (required by non-procedural languages) are needed, procedural languages are capable of describing the system using conditional statements. The conditions are the labels used in non-procedural languages. In fact, at the lower levels, if necessary, a procedural description can appear as a table of (concurrent) primitive conditional actions, much like a non-procedural description.

6.1 -- INSTRUCTION SET PROCESSORS.-

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (usually in the central processor). Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and the particular program.

This behavioral level describes, roughly speaking, what the programmer sees, namely, the *architecture* of the machine.

Computer architectures are usually described in terms of the following relatively fixed format:

Memory.- Physical components which hold information encoded in data.

> Primary-memory.- Contains program and its data.
>
> Processor-state.- Registers accessible to the program - i.e. general registers and program location counter.
>
> Console-state.- Lights and switches enabling communication with the processor.
>
> Input/Output-state.- Controller registers accessible to the program.

Data-Types.- Described in terms of registers which could carry information.

Data-Operations.- Defining operations that can be carried out in terms of data-types.

Instruction-Format.- Specific instances of data-types.

Addressing-Scheme.- Defining how instructions and data are accessed.

Interpreter.- The mechanism of the processor which fetches, decodes, and executes the instructions.

Instruction-Set.- Definition of the particular instructions that the processor executes.

The structural description, on the other hand, corresponds to the *machine organization*, i.e., the particular combination of registers, busses, combinational networks, and control (whether microprogramming or sequential).

The selection of an architecture is usually the first step in the design process, and it is followed by the selection of a machine organization. This process, however, is not a top down set of decisions. The architecture influences the machine organization by impossing a set of requirements (a particular instruction set) and the organization, mainly for technological reasons, influences the architecture of the machine. The result is usually that a given computer architecture can be implemented on a set of machine organizations, and a given organization accepts several architectures.

6.2 -- ORGANIZATION OF THE DESCRIPTION.-

CDL.- The description consists of a list of declaration statements followed by a list of execution statements. There is no provision for partition of the description in blocks of related statements (reflecting a particular organization or hierarchy of activities).

APDL.- The description is organized like an Algol program i.e. a set of blocks, each with its own declarations and statements. Blocks can be nested to any depth, providing a simple scheme to organize a description in a hierarchical fashion.

APL.- The programs are sequences of statements (there are no declarations in the language). Large programs can be divided in segments or functions. ALERT descriptions are organized in microprograms. A description consists of a list of declarations followed by a list of microprograms.

ISP.- Descriptions follow the block structure of Algol programs. A description of a machine consists of a list of declarations followed by the processes (subunits) and action sequences. Descriptions can be named and used as independant processes or as part of larger units.

6.3 -- EXAMPLE: A MICROPROGRAM-CONTROLLED COMPUTER

A small microprogrammed computer [L07] will be described in CDL (the original description) and in ISP (at different levels).

The processor has a main memory M with capacity of 32K, 24 bit words, and a control memory of 1K, 24 bit words.

The microprograms reside in the control memory and the sequencing of microinstructions is explicitly given in an address field (the first 10 bits) in each microword. This address points to the next microinstruction to be executed.

The microwords contain, besides the address of the successor, several one bit fields, each representing a specific micro-order. The micro-orders in a microword are executed in synchronism with a three phase clock.

Instructions are fetched and executed using two control memory words. The first one implements the instruction fetch and the second implements the instruction proper i.e., the data operations. For this simple machine each instruction requires only one microword. An instruction cycle of the machine, takes 6 clock pulses (3 for each control memory word). The address of the first control memory word (the fetch sequence) is a constant (9). The address of the second (the instruction sequence) is given by the operation code.

Sequence name and address of microwords in control memory

	9	0	1	2	3	4	5	6	7	8
Field	FETCH	ADD	SUB	JOM	STO	JMP	SHR	CLS	CLA	STP
F(0-9)	0	9	9	9	9	9	9	9	9	9
F(10)	1	1	1	0	0	0	0	0	1	0
F(11)	1	0	0	0	0	0	0	0	0	0
F(12)	1	0	0	0	0	0	0	0	0	0
F(13)	0	1	1	1	1	1	1	1	1	1
F(14)	0	0	0	0	1	0	0	0	0	0
F(15)	0	1	0	0	0	0	0	1	0	0
F(16)	0	0	1	0	0	0	0	0	0	0
F(17)	0	0	0	0	0	1	0	0	0	0
F(18)	0	0	0	1	0	0	0	0	0	0
F(19)	0	0	0	0	0	0	1	0	0	0
F(20)	0	0	0	0	0	0	0	1	0	0
F(21)	0	0	0	0	0	0	0	0	1	0
F(22)	0	0	0	0	0	0	0	0	0	1

CDL−1 Structural Description

```
Register,          R(0−23),          buffer register for memory M
                   A(0−23),          arithmetic register
                   C(0−14),          address register for memory M
                   D(0−14),          program register
                   F(0−23),          buffer register for memory CM
                   H(0−9),           address register for memory CM
                   G,                start-stop control register

Subregister,       R(op) = R(0−5), op-code part of register R
                   R(addr) = R(9−23),          address part of register R
                   F(addr) = F(0−9),           address part of register F

Memory,            M(C) = M(0−32767,0−23),  main memory
                   CM(H) = CM(0−1023,0−23), control memory

Switch,            Power(ON),
                   Start(ON),
                   Stop(ON),

Clock,             P(1−3),           three phase clock

/Power(ON)/        G←0, F←0, H←0, C←0, D←0, R←0,
/Start(ON)/        G←1, F(12)←1, F(0−11,13−22)←0,
/Stop(ON)/         G←0,

/F(10)*P(1)/       R←M(C),           memory fetch
/F(11)*P(2)/       H←R(op),          address of microword (instruction sequence)
                   D←Countup D,      increment program counter
/F(12)*P(3)/       IF (G)
                   THEN (F←CM(H), C←R(addr)) control memory fetch
                   ELSE (H←9, C←0, D←0, R←0)          clear registers
/F(13)*P(2)/       H←F(addr),        address of microword (fetch sequence)
/F(13)*P(3)/       C←D, F←CM(H),     control memory fetch
/F(14)*P(1)/       R←A,              store accumulator (2 cycles)
/F(14)*P(2)/       M(C)←R,
/F(15)*P(2)/       A←A add R,
/F(16)*P(2)/       A←A sub R,
/F(17)*P(1)/       D←R(addr),
/F(18)*P(1)/       IF (A(0)) THEN (D←R(addr)),
/F(19)*P(1)/       A←shr A,
/F(20)*P(1)/       A←cil A,
/F(21)*P(1)/       A←0,
/F(22)*P(1)/       G←0,
```

ISP-1 Structural Description

```
R<0:23>
A<0:23>
C<0:14>
D<0:14>
F<0:23>
H<0:9>
G

R.op<0:5> := R<0:5>
R.addr<0:15> := R<9:23>
F.addr<0:9> := F<0:9>

M[0:32767]<0:23>
CM[0:1023]<0:23>

Power
Start
Stop

P<1:3>

P.loop := (P←1;next P←2;next P←4;next P.loop)

(Power ⇒   G←0;F←0;H←9;C←0;D←0;R←0)
(Start ⇒   G←1;F<12>←1;F<0:11,13:22>←0)
(Stop ⇒    G←0)

(F<10>∧P<1>⇒ R←M[C]);
(F<11>∧P<2>⇒ H←R.op; D←D+1)

(F<12>∧P<3>⇒ (G⇒ F←CM[H];C←R.addr);
              (¬G⇒ H←0;C←0;D←0;R←0) );

(F<13>∧P<2>⇒ H←F.addr);
(F<13>∧P<3>⇒ F←CM[H];C←D);
(F<14>∧P<1>⇒ R←A);
(F<14>∧P<2>⇒ M[C]←R);
(F<15>∧P<2>⇒ A←A+R);
(F<16>∧P<2>⇒ A←A−R);
(F<17>∧P<1>⇒ D←R.addr);
(F<18>∧P<1>⇒ (A<0>⇒   D←R.addr));
(F<19>∧P<1>⇒ A←0□A<0:22>);
(F<20>∧P<1>⇒ A←A<1:23>□A<0>);
(F<21>∧P<1>⇒ A←0);
(F<22>∧P<1>⇒ G←0)
```

ISP−2 Functional Description

The exact sequencing of operations is described without timing details.

```
R<0:23>
A<0:23>
C<0:14>
D<0:14>
F<0:23>
H<0:9>
G

R.op<0:5> := R<0:5>
R.addr<0:15> := R<9:23>
F.addr<0:9> := F<0:9>

M[0:32767]<0:23>
CM[0:1023]<0:23>

Power
Start
Stop

(Power ⇒   G←0;F←0;H←9;C←0;D←0;R←0)
(Start ⇒   G←1;F<12>←1;F<0:11,13:22>←0)
(Stop ⇒    G←0)

(H=9 ⇒     R←M[C];next H←R.op;DD+1;next
           (G⇒ F←CM[H];C←R.addr);
           (¬G⇒ H←0;C←0;D←0;R←0) )

(H=0 ⇒     R←M[C];next A←A+R;H←F.addr;next C←D;F←CM[H])

(H=1 ⇒     R←M[C];nextA←A−R;H←F.addr;next C←D;F←CM[H])

(H=2 ⇒     (A<0>⇒ D←R.addr);next H←F.addr;next C←D;F←CM[H])

(H=3 ⇒     R←A;nextM[C]←R; H←F.addr;next C←D;F←CM[H])

(H=4 ⇒     D←R.addr;next H←F.addr;next C←D;F←CM[H])

(H=5 ⇒     A←0□A<0:22>;next H←F.addr;next C←D;F←CM[H])

(H=6 ⇒     A←A<1:23>□A<0>;next H←F.addr;next C←D;F←CM[H])

(H=7 ⇒     R←M[C];A←0;next A←A+R; H←F.addr;next C←D;F←CM[H])

(H=8 ⇒     G←0;next H←F.addr;next C←D;F←CM[H])
```

ISP-3 Behavioral Description

No details about timing and control are provided, only the algorithmic description of the effect of each instruction upon the memory components of the system.

```
R<0:23>
A<0:23>
D<0:14>
G

R.op<0:5> := R<0:5>
R.addr<0:15> := R<9:23>

M[0:32767]<0:23>

Power
Start
Stop

(Power ⇒    G←0;D←0)
(Start ⇒    G←1))
(Stop ⇒     G←0)

INTERPRETER := (G⇒R←M[D];D←D+1;next EXECUTE;next INTERPRETER)

EXECUTE := (
        (R.op=0⇒ A←A+M[R.addr]);
        (R.op=1⇒ A←A−M[R.addr]);
        (R.op=2⇒ (A < 0⇒D←R.addr));
        (R.op=3⇒ M[R.addr]←A);
        (R.op=4⇒ D←R.addr);
        (R.op=5⇒ A←A÷2 {LOGICAL});
        (R.op=6⇒ A←A*2 {ROTATE});
        (R.op=7⇒ A←M[R.addr]);
        (R.op=8⇒ G←0)
               )
```

7.0 -- SIMULATION AND DESIGN AUTOMATION

Several languages have been implemented in Design Automation Systems in an attempt to facilitate the design and implementation of complex digital systems. The main areas of application are:

- Simulation and Analysis.- Prior to the actual implementation of the system its characteristics are evaluated using a digital systems simulator. The information required by the simulator and the techniques used depend upon the level of design. This has been the most succesful application of computers as design aids, as witnessed by the proliferation of material dealing with automated analysis and simulation of digital systems [S01-S21].

- Synthesis.- The implementation of the physical machine is a process of translation from a symbolic representation to a physical representation, with the constraint that both representations define the same behavior (algorithm) and that the physical implementation satisfies the constraints associated with the elements (e.g. fan-in, fan-out, etc) and constraints imposed by the designer (cost, speed, etc). This application has been relatively succesful at the lower levels of design (combinational and sequential switching circuits).

As the complexity and size of the systems increases, modular components have become a necessity for the designer. It is no longer economic (in terms of design and development time) to design at the gate level. Several sets of modules are available [M01-M11]. They reflect in hardware (the physical representation) what the RT languages are for the symbolic representation of systems, namely, reduction of the complexity of the design task by abstraction and elimination of redundant details.

At this level, automatic design programs have yet to be implemented. There are no techniques for analysis or synthesis* of modular digital systems, but this situation is bound to change as more research is done in the area [R01].

For a survey on the Automated Design and Analysis of Digital Systems at the lower level of design, see [S02].

- -

* Simulation of modular systems has been more succesful, perhaps because of the closeness of the RT level description to conventional programming. This allows simple transliteration of the RT description into executable programs, providing cheap and fast simulation (although in many cases, RT languages are compiled directly). Most simulation techniques used at the lower level are applicable to register transfer systems.

ISP−3 Behavioral Description

No details about timing and control are provided, only the algorithmic description of the effect of each instruction upon the memory components of the system.

```
R<0:23>
A<0:23>
D<0:14>
G

R.op<0:5> := R<0:5>
R.addr<0:15> := R<9:23>

M[0:32767]<0:23>

Power
Start
Stop

(Power ⇒    G←0;D←0)
(Start ⇒    G←1))
(Stop ⇒     G←0)

INTERPRETER := (G⇒R←M[D];D←D+1;next EXECUTE;next INTERPRETER)

EXECUTE := (
        (R.op=0⇒ A←A+M[R.addr]);
        (R.op=1⇒ A←A−M[R.addr]);
        (R.op=2⇒ (A < 0⇒D←R.addr));
        (R.op=3⇒ M[R.addr]←A);
        (R.op=4⇒ D←R.addr);
        (R.op=5⇒ A←A÷2 {LOGICAL});
        (R.op=6⇒ A←A*2 {ROTATE});
        (R.op=7⇒ A←M[R.addr]);
        (R.op=8⇒ G←0)
              )
```

7.0 -- SIMULATION AND DESIGN AUTOMATION

Several languages have been implemented in Design Automation Systems in an attempt to facilitate the design and implementation of complex digital systems. The main areas of application are:

- Simulation and Analysis.- Prior to the actual implementation of the system its characteristics are evaluated using a digital systems simulator. The information required by the simulator and the techniques used depend upon the level of design. This has been the most succesful application of computers as design aids, as witnessed by the proliferation of material dealing with automated analysis and simulation of digital systems [S01-S21].

- Synthesis.- The implementation of the physical machine is a process of translation from a symbolic representation to a physical representation, with the constraint that both representations define the same behavior (algorithm) and that the physical implementation satisfies the constraints associated with the elements (e.g. fan-in, fan-out, etc) and constraints imposed by the designer (cost, speed, etc). This application has been relatively succesful at the lower levels of design (combinational and sequential switching circuits).

As the complexity and size of the systems increases, modular components have become a necessity for the designer. It is no longer economic (in terms of design and development time) to design at the gate level. Several sets of modules are available [M01-M11]. They reflect in hardware (the physical representation) what the RT languages are for the symbolic representation of systems, namely, reduction of the complexity of the design task by abstraction and elimination of redundant details.

At this level, automatic design programs have yet to be implemented. There are no techniques for analysis or synthesis* of modular digital systems, but this situation is bound to change as more research is done in the area [R01].

For a survey on the Automated Design and Analysis of Digital Systems at the lower level of design, see [S02].

- -

* Simulation of modular systems has been more succesful, perhaps because of the closeness of the RT level description to conventional programming. This allows simple transliteration of the RT description into executable programs, providing cheap and fast simulation (although in many cases, RT languages are compiled directly). Most simulation techniques used at the lower level are applicable to register transfer systems.

CDL.– A simulator and Boolean translator has been implemented for CDL [S12]. During the execution of the simulator, the contents of chosen registers, memory words, and positions of switches can be displayed at every clock cycle. The execution of the simulator runs in a loop called label cycle, during which:

a) If a manual switch operation occurs, the micro–statements of the corresponding switch statement are carried out.

b) All labels are evaluated, the activated labels i.e. the labels having the value 1, are located.

c) The micro statements of the activated labels are carried out in two steps. First, all values to be stored in various registers and memories are evaluated and collected. Then the collected values are stored.

d) It is checked whether the simulation should be terminated. If not, the process starts from a) again.

The translation into Boolean equations consists of four phases. During the first phase, the design is scanned and a micro–statement table is generated. The second phase generates a truth table for each micro–statement. During the third phase, Boolean product terms are generated from the truth tables. The fourth phase sorts and combines the product terms into a list of Boolean input equations for each device.

Since the meaning of the inputs to the storage devices depend on the type of device (flip–flop), the output of the Boolean translator must be processed later with given flip–flop definitions.

APDL.– An APDL system exists, in which descriptions are compiled into executable code*. This code, combined with a set of run time routines, can be executed to simulate the system described. Since APDL has no simulation command language, all reading of data, computing of statistics, and printing of results must be included as part of the description.

A special procedure "T" (Time) is called, at run time, at the beginning of each simulated cycle time. This procedure increases the cycle time counter, resets the PULSE variables that were true during the previous cycle time, and evaluates the IF EVER conditions (executing the proper statements if the condition is satisfied). At the end of the procedure, control is returned to the object code program and the non–monitoring statements of the time block are executed.

– – – – – – – – – – – – – – – – – – –

* Actually, it translates into a subset of Algol, thus providing exportability, but at the cost of executing parallel statements serially which may produce undesirable side effects.

The implementation program produces a hardware specification list consisting of three items:

a) A list of hardware components

b) A list of data paths between the hardware elements.

c) An SFD-Algol [L19] description of a finite state controller that will sequence the data transfers via the data paths listed in b).

The hardware specification is not complete since it does not provide the information on the particular types of elements to be used i.e. they are somewhat idealized but have feasible realizations.

APL.- The ALERT implementation does not provide a simulation capability. The system is used as a front end for the IBM Design and Logic Automation Systems [S04,S18]. The description is processed in 8 steps that provide details needed to complete the logic design.

1) Translation.- Transforms the description into an internal format, performing syntax analysis.

2) Selection decoding.- Provides array accessing logic.

3) Macro generation.- Introduces predefined combinational networks.

4) Sequence analysis.- Control and sequencing requirements are determined.

5) Flip-flop identification.- Non-declared variables are implemented as flip-flops whenever a memory capability is required.

6) Control provision.- Introduces the sequence control counters.

7) Consolidation.- Eliminates redundant logic blocks, rearranges the connections, and associated elements are tied together.

8) Expansion.- Arrays of elements are replaced by individual devices and connections.

8.0 -- LANGUAGE REQUIREMENTS

What properties are desired in a language for writing behavioral and structural descriptions of a digital system?

RT languages are similar to most programming languages since they both carry out register assignments. The parallel nature of hardware suggests that it could be useful to at least have a special notation, even though programming languages such as FORTRAN can be used for this purpose.

We can divide the set of properties in two classes: one of them consists of the requirements for a scientific notation used in a design process; and the other has to deal directly with the objects we are devising, namely digital systems, particularly computers.

GENERAL PROPERTIES

A) Readability.– The notation is going to be used as a conveyor of information, not only between man and machines but among humans, who do not all have the same experience or involvement in the design. A description in this notation should be precise, concise and elegant (considerations of typography, character sets, formats, the way operations such as array accesses are described, etc.). It should be usable as the ultimate source of information about the object. Information should be extracted from the context rather than by syntax which clouds the description (e.g. register declarations, keywords etc.).

CDL.– Fair, expressions are simple but the flow of control requires more detail than necessary for a behavioral description. A small character set imposes restrictions, such as the use of "–" as both a range operator and a register concatenation operator.

APDL.– Fair, Algol block structure facilitates partitioning of the description. Requires a large number of reserved keywords.

APL.– Good, but requires a large character set. Few reserved keywords (system parameters). Compact encoding of algorithms ("one–liners") requires the development of some reading skills.

ISP.– Good, very few reserved keywords. A large character set with simple transliterations. Block structure allows the partitioning of the description.

B) Familiarity.– Primitive concepts in the language should be named and used in a way consistent with general practice.

CDL.– Good.

APDL.– Good.

APL.– Fair, uses a non standard precedence.

ISP.– Good.

C) Generality.–The notation should describe the elements occurring in the universe of interest, and at several levels of detail (in a hierarchical way), by suppressing repetitive or unnecessary detail.

CDL.– Fair; does not allow the suppression of unnecessary detail, for instance, the use of the Memory Address Register in memory access operations.

APDL.– Good; allows descriptions at different levels of detail. The language has all the power of Algol as an algorithmic language.

APL.– Good; A powerful set of operators can be used to describe algorithms at different levels.

ISP.– Excellent; systems can be described at any level of detail, from the programming level down to the switching circuits.

D) Simplicity.– There should be few primitive concepts, and they should be used consistently throughout the description, avoiding special cases of more general concepts, or things that are of relative importance or that imply a specific implementation.

CDL.– Good, there are few types of statements. Uses a few elements that are special cases of more general concepts i.e. DECODER (combinational network), SWITCH, LIGHT (registers).

APDL.– Poor, too many types of registers and register arrays which are special cases of the general concept of information carrier.

APL.– Excellent, very few concepts which are used consistently throughout the language.

ISP.– Excellent, one type of carrier declaration allows the definition of registers, memories, and combinational networks. Actions and action sequences have block structure, allowing description of complex activities in a recursive fashion.

SPECIFIC PROPERTIES

A) Extensibility.– The language should be able to extend gracefully by defining constructs in terms of elements already in the language. This also gives the capability to describe machines in a hierarchical fashion.

CDL.– Poor, does not allow the description of new operators (except combinational networks) or procedures.

APDL.– Good, restricted to the use of Algol procedures. Does not allow the declaration of combinational networks.

APL.– Good, procedures can have up to two parameters. Macro substitution was added in the ALERT system.

ISP.– Excellent, procedures and operators can be defined by the user. Expression modifiers can be used to provide information about the data–types. Names can be substituted by abbreviations (alias).

B) Fidelity.– The organization of the description should reflect the organization of the machine, making the intentions of the designers transparent to the users, humans or simulation/production automation processes.

CDL.– Good, the description reflects the organization of the machine at the RT level, but not at the programming level.

APDL.– Good.

APL.– Poor, lacks block structure and parallelism.

ISP.– Excellent, descriptions can be partitioned and organized in different ways to reflect the machine organization at different levels.

C) Timing and Concurrency.– Machines are essentially parallel and this implies that concurrency should be the rule rather than the exception.

CDL.– Excellent.

APDL.– Good, new statement types allow the description of concurrent activities. Time blocks can not be nested.

APL.– Poor, does not allow the description of concurrent activities.

ISP.– Excellent, concurrency of actions is assumed by default.

D) Syntactically simple (writable).– The notation is a tool for designers and descriptions should be written by them and not (necessarily) by programmers.

CDL.– Excellent.

APDL.– Good, Algol is a richer language.

APL.– Excellent.

ISP.— Excellent.

E) Hardware independence.— The notation should be relatively independent of any hardware technology, machine organization, timing mode, design procedure or simulation/ production techniques. This is somewhat in conflict with the fidelity property.

CDL.— Good, the language is best suited for synchronous systems.

APDL.— Good, the language favors synchronous systems.

APL.— Excellent. ALERT is, however, oriented towards a specific technology.

ISP.— Excellent.

F) Separability.— The notation should be able to express the dichotomy between data and control. It should express the structure and behavior of the data flow, which implies the behavior of the control part. Also separability should permit the function of a primitive (e.g. an AND gate) to be described in an independent fashion.

CDL.— Excellent, true in all non—procedural languages.

APDL.— Fair, Algol statements mix data and control operators. Separation is provided by the compiler.

APL.— Poor, in ALERT the control can be automatically provided by the design programs.

ISP.— Good, specialized control components like clocks and pulses could be added to the language.

The conclusion that may be extracted from the preceding comparison is that, except for minor improvements, the languages do all that could be desired. This is, however, not true. Designers that use these languages tend to write the description in terms of very simple constructs, because that is all that the languages provide. Similarly, and to complete the circular argument, languages are designed with simple elements because that is all the users need.

The situation will change in the future, as the level at which we design our hardware raises. For instance, all the present RT languages are more or less suitable to describe the primitive components of the programming level of design, that is, they are capable of describing instruction set processors for machine languages as we know them, i.e. relatively fixed instruction formats, static machine organizations, static interpretation of operands and operators etc. This is not enough to describe future machines, say, that interpret directly a high level language. The RT languages are not capable of handling dynamic interpretation of names or instructions.

REFERENCES

LANGUAGES

[L01] Anceau, F., Liddell, P., Mermet, J., and Payand, C.: "CASSANDRE: A Language to describe Digital Systems, Applications to Logic Design". Third International Symposium on Computer and Information Science (COINS-69), Miami, December 1969.

[L02] Baray, M.B. and Su, S.Y.H.: "A Digital System Modeling Philosophy and Design Language". 8th Annual Design Automation Workshop, Atlantic City, New Jersey, June 1971, pp. 1-22.

[L03] Barbacci, M.R., Bell, C.G., and Newell, A.: "ISP: A Language to describe Instruction Sets and other Register Transfer Systems". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 219-222.

[L04] Bell, C.G. and Newell, A.: "The PMS and ISP Descriptive Systems for Computer Structures". SJCC 1970, pp. 351-374.

[L05] Burnett, G.J.: "A Design Language for Digital Systems". M.S. Thesis, EE department, MIT, 1965.

[L06] Chu, Y.: "An Algol-like Computer Design Language". CACM, Vol. 8, October 1965, pp. 607-615.

[L07] Chu, Y.: "Introducing the Computer Design Language". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 215-218.

[L08] Darringer, J.A.: "The Description, Simulation, and Automatic Implementation of Digital Computer Processors". PhD Thesis, EE Department, CMU, May 1969.

[L09] Duley, J.R.: "DDL - A Digital Design Language". PhD Thesis, EE Department, University of Wisconsin, June 1970.

[L10] Falkoff, A.D., Iverson, K.E., and Sussenguth, E.H.: "Formal description of System/360". IBM Systems Journal, Vol. 3, pp. 198-262, 1964.

[L11] Falkoff, A.D.: "Formal description of processess - The first step in Design Automation". IBM Research Note NC-510, June 1965.

[L12] Giese, A.: "HARGOL - A Hardware Oriented Algol Language". A/S Regnecentralen, Copenhagen, Denmark, Feb. 1969.

[L13] Gorman, D.F. and Anderson, J.P.: "A Logic Design Translator". FJCC 1962, pp.251–261.

[L14] Gorman, D.F.: "A System Descriptive Language and its Uses". PhD Thesis, EE Department, University of Pennsylvania, April 1968.

[L15] Gorn, S., Ingerman, P.Z., and Crozier, J.B.: "On the Automatic Construction of Micro–flowcharts". CACM, Vol. 2, No. 10, October 1959, pp. 27–31.

[L16] Iverson, K.E.: "A Programming Language". Wiley, 1962.

[L17] Iverson, K.E.: "A Common Language for Hardware, Software, and applications". FJCC 1962, pp. 121–129.

[L18] Metze, G. and Seshu, S.: "A proposal for a Computer Compiler". SJCC 1966, pp. 253–263.

[L19] Parnas, D.L.: "System Function Description Algol". PhD Thesis, EE Department, CMU, Feb. 1965.

[L20] Proctor, R.M.: "A Logic Design Translator experiment demonstrating relationships of Language to Systems and Logic Design". IEEE–TEC, Vol. EC–13, August 1964, pp. 422–430.

[L21] Schlaeppi, H.P.: "A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)". IEEE–TEC, Vol. EC–13, August 1964, pp. 439–448.

[L22] Schorr, H.: "Computer Aided Digital System Design and Analysis Using a Register Transfer Language". IEEE–TEC, Vol. EC–13, December 1964, pp. 730–737.

[L23] Srinivasan, C.V.: "An Introduction to CDL1, A Computer Description Language". AD661591.

[L24] Stabler, E.P.: "System Description Languages", IEEE–TC, Vol. C–19, December 1970, pp. 1160–1173.

[L25] Wilber, J.A.: "A Language for Describing Digital Computers". M.S. Thesis, Report No. 197, Department of Computer Science, University of Illinois, Feb. 1966.

SIMULATION AND DESIGN AUTOMATION SYSTEMS

[S01] Balducci, E.G., Davis, W.E., and Persels, C.G.: "Automatic Logic Implementation". Proceedings of the ACM National Conference 1968, pp. 223-240.

[S02] Breuer, M.A.: "Recent Developments in the Automated Design and Analysis of Digital Systems". Proceedings of the IEEE, Vol. 60, No. 1, January 1972, pp. 12-27.

[S03] Breuer, M.A. (Ed):"Design Automation of Digital Systems: Theory and Techniques". Vol. 1, Prentice Hall 1972.

[S04] Case, P.W. et al: "Solid Logic Design Automation". IBM Journal of Research and Development, Vol. 8, April 1964, pp. 127-140.

[S05] Friedman, T.D. and Yang, S.: "Quality of Designs from an Automatic Logic Design Generator". IBM Research Report RC-2068, April 1968.

[S06] Friedman, T.D. and Yang, S.: "Methods used in an Automatic Logic Design Generator (ALERT)". IEEE-TC, Vol. C-18, No. 7, July 1969, pp. 593-614.

[S07] Guskin, J.R. and Dingwall, T.J.: "The Discrete, Logical Design, Simulation System". CONCOMP Memorandum 26, University of Michigan, April 1970.

[S08] Jacoby, K. and LaLiberte, A.R.: "Using a Computer to Design a Computer". Computers and Automation, April 1966, pp. 36-39.

[S09] Koomok, M., Case, P.W., and Graff, H.H.: "The Recording, Checking, and Printing of Logic diagrams". Proceedings of the EJCC, December 1958, pp 108-118.

[S10] Leiner, A.L., Weiberger, A., Coleman, C., and Loberman, H.: "Using Digital Computers in the Design and Maintenance of new Computers". IEEE-TEC, Vol. EC-10, December 1961, pp. 680-690.

[S11] Lewin, D.W. and Waters, M.C.: "Computer Aids to Logic Systems Design". The Computer Bulletin, November 969, pp. 382-388.

[S12] Mesztenyi, C.K.: "Computer Design language. Simulation and Boolean Translation". Technical Report 68-72, Computer Science center, University of maryland, June 1968.

[S13] Parnas, D.L. and Darringer, J.A.: "SODAS and a Methodology for Systems Design". FJCC 1967.

[S14] Parnas, D.L.: "More on Simulation Languages and Design Methodology for Computer Systems". SJCC 1969, pp. 739–743.

[S15] Potash, H.: "A Digital Control Design System". PhD Thesis, Report No. 69–21, School of Engineering, UCLA, May 1969.

[S16] Rocket, F.A.: "A Systematic Method for Computer Simplification of Logic Diagrams". IRE International Convention, March 1961.

[S17] Rosenthal, C.W.: "Computing Machine Aids to a development project". IRE–TEC, Vol. EC–10, September 1961, pp. 400–406.

[S18] Roth, J.P.: "Systematic Design of Automata". FJCC 1965, pp. 1093–1100.

[S19] Rozenberg, D.P. and Savage, R.L.: "A Proposal for the Computer Design Process based on Multi–level Simulation". IFIP Congress 1971.

[S20] Scheff, B.H.: "A Machine Design Aids System for Digital Designers". Computer Design, October 1969, pp. 76–81.

[S21] Ulrich, E.G.: "Selective Path Simulation of Synchronous and Asynchronous Digital Networks". Report X6–1918/030, North American Aviation/Autonetics, Anaheim, California, August 1966.

SIMULATION AND DESIGN AUTOMATION SYSTEMS

[S01] Balducci, E.G., Davis, W.E., and Persels, C.G.: "Automatic Logic Implementation". Proceedings of the ACM National Conference 1968, pp. 223-240.

[S02] Breuer, M.A.: "Recent Developments in the Automated Design and Analysis of Digital Systems". Proceedings of the IEEE, Vol. 60, No. 1, January 1972, pp. 12-27.

[S03] Breuer, M.A. (Ed):"Design Automation of Digital Systems: Theory and Techniques". Vol. 1, Prentice Hall 1972.

[S04] Case, P.W. et al: "Solid Logic Design Automation". IBM Journal of Research and Development, Vol. 8, April 1964, pp. 127-140.

[S05] Friedman, T.D. and Yang, S.: "Quality of Designs from an Automatic Logic Design Generator". IBM Research Report RC-2068, April 1968.

[S06] Friedman, T.D. and Yang, S.: "Methods used in an Automatic Logic Design Generator (ALERT)". IEEE-TC, Vol. C-18, No. 7, July 1969, pp. 593-614.

[S07] Guskin, J.R. and Dingwall, T.J.: "The Discrete, Logical Design, Simulation System". CONCOMP Memorandum 26, University of Michigan, April 1970.

[S08] Jacoby, K. and LaLiberte, A.R.: "Using a Computer to Design a Computer". Computers and Automation, April 1966, pp. 36-39.

[S09] Koomok, M., Case, P.W., and Graff, H.H.: "The Recording, Checking, and Printing of Logic diagrams". Proceedings of the EJCC, December 1958, pp 108-118.

[S10] Leiner, A.L., Weiberger, A., Coleman, C., and Loberman, H.: "Using Digital Computers in the Design and Maintenance of new Computers". IEEE-TEC, Vol. EC-10, December 1961, pp. 680-690.

[S11] Lewin, D.W. and Waters, M.C.: "Computer Aids to Logic Systems Design". The Computer Bulletin, November 969, pp. 382-388.

[S12] Mesztenyi, C.K.: "Computer Design language. Simulation and Boolean Translation". Technical Report 68-72, Computer Science center, University of maryland, June 1968.

[S13] Parnas, D.L. and Darringer, J.A.: "SODAS and a Methodology for Systems Design". FJCC 1967.

[S14] Parnas, D.L.: "More on Simulation Languages and Design Methodology for Computer Systems". SJCC 1969, pp. 739-743.

[S15] Potash, H.: "A Digital Control Design System". PhD Thesis, Report No. 69-21, School of Engineering, UCLA, May 1969.

[S16] Rocket, F.A.: "A Systematic Method for Computer Simplification of Logic Diagrams". IRE International Convention, March 1961.

[S17] Rosenthal, C.W.: "Computing Machine Aids to a development project". IRE-TEC, Vol. EC-10, September 1961, pp. 400-406.

[S18] Roth, J.P.: "Systematic Design of Automata". FJCC 1965, pp. 1093-1100.

[S19] Rozenberg, D.P. and Savage, R.L.: "A Proposal for the Computer Design Process based on Multi-level Simulation". IFIP Congress 1971.

[S20] Scheff, B.H.: "A Machine Design Aids System for Digital Designers". Computer Design, October 1969, pp. 76-81.

[S21] Ulrich, E.G.: "Selective Path Simulation of Synchronous and Asynchronous Digital Networks". Report X6-1918/030, North American Aviation/Autonetics, Anaheim, California, August 1966.

MODULES

[M01] Bell, C.G. and Grason, J.: The Register Transfer Module Design Concept". Computer Design, Vol. 10, No. 5, May 1971, pp. 87-94.

[M02] Bell, C.G. Eggert, J.L., Grason, J., and Williams, P.: "The Description and Use of Register-Transfer Modules (RTM's)". IEEE-TC, Vol. C-21, No. 5, May 1972, pp. 495-500.

[M03] Bell, C.G., Grason, J., and Newell, A.: "Designing Computers and Digital Systems". Digital Press, Digital Equipment Corporation, 1972.

[M04] Bell, C.G., Grason, J., and Siewiorek, D.P.: "Register Transfer Modules (RTMs) for understanding Digital Systems Design". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 305-308.

[M05] Clark, W.A.: "Macromodular Computer Systems". (an introduction to a set of 6 papers on the subject), SJCC 1967, pp. 335-401.

[M06] Clark, W.A. and Molnar C.E.: "The promise of Macromodular Systems". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 309-312.

[M07] Dennis, J.B.: "Modular, Asynchronous Control Structures for a High Performance Processor". Conference Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, pp. 55-80.

[M08] Digital Equipment Corporation: "PDP16 Computer Design Handbook", 1971.

[M09] Ellis, R.A.: "Modular Computer Systems". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 301-302.

[M10] Ellis, R.A. and Franklin, M.A.: "High level Logic Modules: A qualitative comparison". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 313-316.

[M11] Patil, S.S. and Dennis, J.B.: "The Description and Realization of Digital Systems". IEEE Computer Conference, COMPCON 72, San Francisco, September 1972, pp. 223-226.

OTHER REFERENCES

[R01] Bell, C.G., Jordan, A.G., and Traub, J.F.: "Register Transfer (RT) level: Components, Representation and Design techniques". Proposal from the Electrical Engineering and Computer Science Departments, CMU, to the National Science Foundation (Computer Systems Design Program), CMU proposal No. 1582, October 1971.

[R02] Bell, C.G. and Newell, A.: "Computer Structures: Readings and Examples". Mc-Graw Hill Book Company, New York, 1971.

[R03] Chu, Y.: "Introduction to Computer Organization". Prentice-Hall Inc., 1970.

[R04] Chu, Y.: "Computer Organization and Microprogramming". Prentice-Hall Inc., 1972.

[R05] Dennis, J.B., MIT Project MAC Progress Report VII, July 1969-July 1970, pp. 11-41.

[R06] Fisher, D.A.: "Control Structures for Programming Languages". PhD Thesis, Computer Science Department, CMU, May 1970.

[R07] Van der Poel, W.L.: "SERA 69, a new hypothetical machine for educational purposes". IFIP World Conference on Computer Education, 1970.