23

Reprinted from IEEE TRANSACTIONS
ON *COMPUTERS*
Volume C-21, Number 5, May, 1972
pp. 495-500

The Description and Use of Register-Transfer Modules (RTM's)®

C. GORDON BELL, SENIOR MEMBER, IEEE, J. L. EGGERT, MEMBER, IEEE, J. GRASON, MEMBER, IEEE, AND P. WILLIAMS

Abstract—This note describes a set of register-transfer modules (RTM's) that are used as a basis for digital systems design. RTM's allow digital systems to be specified in a flow chart form with complete construction (wiring) information, thus obviating combinational and sequential switching circuit theory based design. The modules make extensive use of integrated circuitry.

The note briefly describes the class of problems that RTM's can be used to solve, together with some of the module design decisions. The most important RTM's are described from the user's viewpoint, and two example designs are given.

Index Terms—Asynchronous logic, computer emulation, digital system design, flow charts, logical design, LSI, macro logic, register transfer.

INTRODUCTION

In the design of digital systems (e.g., computers) the problem formulation and the design solution are most likely carried out at a register-transfer concept level. Early and recent texts on logical and computer design discuss the register transfers as primitive components, e.g., Bartee *et al.* [1] and Chu [2]. Logical design simulators that use a register-transfer language have been written and there have been several attempts to carry out detailed sequential and combinational logic designs from register-transfer descriptions, e.g., Friedman and Yang [3]. Despite the acknowledgment that there are primitives based on register transfers, there is yet to emerge a common set of modules that are taken as primitive in the same way we think of various flip-flop types and NAND and NOR gates. However, Clark [4] at Washington University, St. Louis, Mo., has been developing and evaluating such a basic set of modules, called macromodules.

Register-transfer modules are our first attempt at providing a basic set of modules for high-level digital systems design. These modules have been implemented by the Digital Equipment Corporation (DEC). The design of RTM's has been influenced by many of the above approaches and disciplines, and by programming methods. This note presents the general problem RTM's are trying to solve, the factors constraining their design, a brief description of the more important modules from a user's point of view, and two examples of their use.

Several aspects of the RTM system are important.

1) Digital system design is carried out entirely in terms of the modules; combinational and sequential switching circuit design are not used. (The process is akin to programming a sequential computer.) Design time is significantly less than with conventional logical design.

2) The most abstract, and usually the only representation of a given design, has enough information for constructing the system. This representation is a standard flow chart to specify the control flow, coupled to a data part that holds the data and carries out data operations.

3) The register-transfer modules make extensive use of MSI circuitry and can use LSI circuitry to provide even lower cost modules.

MODULE DESIGN CONSIDERATIONS

The three problem classes for which the modules were designed are: special-purpose, computer-related, and educational digital systems. Although the initial motivation for the modules was for education, they were not designed solely for this purpose. The goals for educational use place too many constraints on the design. The main influence of the educational market has been to clarify the pedagogical nature; hence, the description of systems is made easy. The

Manuscript received February 19, 1971; revised May 11, 1971.
C. G. Bell is with Carnegie-Mellon University, Pittsburgh, Pa. 15213, and the Digital Equipment Corporation, Maynard, Mass. 01754.
J. L. Eggert is with the Digital Equipment Corporation, Maynard, Mass. 01754.
J. Grason is with Carnegie-Mellon University, Pittsburgh, Pa. 15213.
P. Williams is with the Digital Equipment Corporation, Maynard, Mass. 01754.
® Trademark of Digital Equipment Corporation, marketed under the name "PDP-16."

TABLE I
BASIC RT DESIGN DECISIONS

1. Logic: TTL (acceptable for speed and noise immunity; low cost).
2. Packaging: Printed circuit boards of 5" x $8\frac{1}{2}$" or $2\frac{1}{2}$" x $8\frac{1}{2}$" with 72 or 36 pins (DEC compatible).
3. Intermediate connection: pre-wired busses; wirewrap and push-on connections over wirewrap pins.
4. Logic interconnection rules: One kind of control signal and data bus. Very small number of rules compared to ic use.
5. Problem size: 4 ~ 100 control steps; 1 ~ 4 arithmetic registers; 16 ~ 100 variables; possibly read-only memory.
6. Word length: 8, 12 and 16-bit (present de facto standard - can be extended).
7. Universality and extendability: The modules aren't a panacea. There are provisions for escape to: regular integrated circuits, standard DEC modules, and DEC computers (and their components).
8. Selection of primitives: Basic register, bus interconnection structure and data representation were first determined. The operations which formed a complete set for the data representation were then specified. With this basic module set, designs were carried out for benchmark problems and design iteration occurred.
9. Notations: PMS and ISP of Bell and Newell [7].
10. Automatic (algorithmic) mapping of algorithm into hardware: The basic RT design archetype representation is a flow chart. The register transfer operations are expressed in the ISP language.
11. Parallelism and speed: Provision for multiple busses; the modules are asynchronous. (The application classes put relatively low weight on speed.) For teaching purposes parallelism is an important principle. (A decision to use a bus, and thereby limit parallelism to the number of busses was made for both cost and simplicity reasons.)

special-purpose digital systems are larger than 20 MSI circuits, but smaller than a stored-program computer (a typical RTM system would have 4~100 control states, 1~4 arithmetic units, and a small memory of 16~1000 words). Computer-related applications range from computer peripherals to the emulation of computers.

We make no attempt to show that the modules are an optimum set, according to an objective function. Because of the elementary nature of the control and data operations, the set is sufficient to construct digital systems. Table I shows the important design variables for RTM's, together with many of the constraints. Their design is described in [5].

THE RTM SYSTEM

The RTM system consists of about 20 different modules and a method of interconnecting modules via a common bus that carries data and timing interlock signals for the register transfers. Some of the modules (*DM*, *T*, and *M* types) connect to the bus in order to transfer data, and the remaining modules (*K* type) "control" when data are to be transferred. The module name types are based on the structure primitive types of Bell and Newell [6], [7].

A register-transfer language, ISP (instruction set processor [6], [7]), is used to define the register-transfer operations of the RTM's. Here we use only the parts of ISP that are commonly known by the digital systems engineer and are similar to a programming language (e.g., Fortran). The four main module types are as follows.

DM-Type (Data Operation Combined with Memory)

These modules are what we commonly think of as being a digital system (or at least the arithmetic unit). They are the register-transfer gating paths and combinational circuits for the simple arithmetic and logical functions—hence the *D* part (for data operations). The *D* part carries out the evaluation of the right-hand side of an arithmetic expression as in a programming language in which an integer value is computed prior to storing, e.g., $\leftarrow A+B$, $\leftarrow A-B$, $\leftarrow A\oplus B$, $\leftarrow A+1$. Thus, an expression "left-hand side←right-hand side" (e.g., $H\leftarrow C+D$) is used to indicate the integer value of the right-hand side being read and placed in the register on the left-hand side.

M-Type (Memory)

The *M* (memory) part is just the registers (e.g., *A*, *B*) that hold data between statements; these essentially correspond to the variables that are declared in a program. The operations on memory are usually reading ($\leftarrow M$) and writing ($M\leftarrow$). Types of *DM* and *M* modules are the general-purpose arithmetic unit, a single-transfer register, Boolean flags (1-bit registers), READ-WRITE memories, and READ-ONLY memories. The memories hold two's complement 8, 12, or 16-bit integers.

K-Type (Control)

The *K* modules are responsible for controlling the transfer of data among the various registers by appropriately evoking operations by *DM* and *M* types. The *K* modules are analogous to the control structure of a program. The *K* modules called *K*.evoke control the times when the various operations of the *DM*'s and *M*'s are evoked (executed). The *K*.branch modules are used to make decisions about which operations are to be evoked next. The *K*.subroutine modules are used to connect a sequence of operations together as a subroutine. *K*.serial-merge allows control flow to merge into a single control flow when any flow input is present. *K*.parallel-branch and *K*.parallel-merge modules synchronize control where there is more than one operation taking place at a time. Other control modules include clocks, delays, and manual start keys.

T-Type (Transducers)

These modules provide an interface to the environment outside RTM. These include the Teletype interface, analog/digital converters, lights, switches, and interfaces to computers. These modules also connect to the common data bus.

The details of the modules will be introduced by giving the four modules that are necessary for nontrivial digital systems: *K*.evoke, *DM.gpa*, *K*.branch, and *K*.bus.

K(Evoke)

K.evoke (*Ke*) is the basic module that evokes a function consisting of a data operation and a register transfer—in essence an arithmetic expression. When a *Ke* is evoked, it in turn evokes the function, consisting of the data operation followed by a register transfer, and when the function is complete, *Ke* evokes the next *K* in the control sequence. The diagram for *Ke* with its two inputs and two outputs is shown in Fig. 1. In terms of a finite state machine, *Ke* is a state with the ability to evoke an output action and then make a transition to another state. *K*.evoke is as follows.

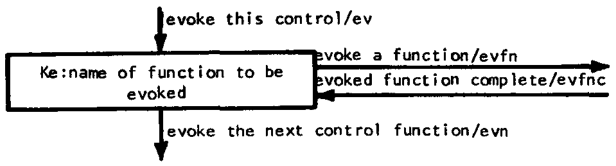

evoke this control/ev
evoke a function/evfn
evoked function complete/evfnc
Ke:name of function to be evoked
evoke the next control function/evn

Fig. 1. Diagram for the control module *K*.evoke.

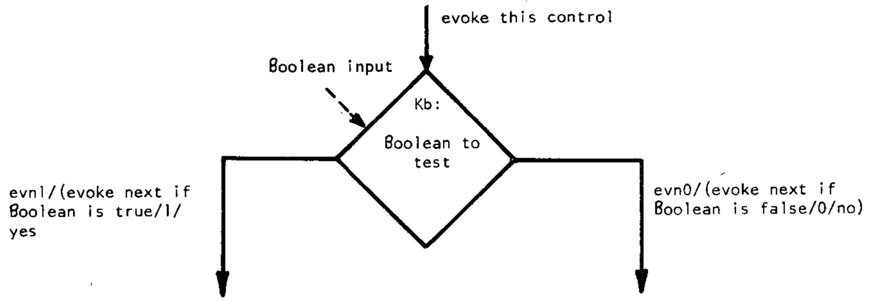
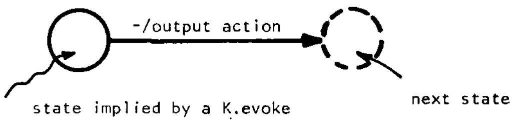
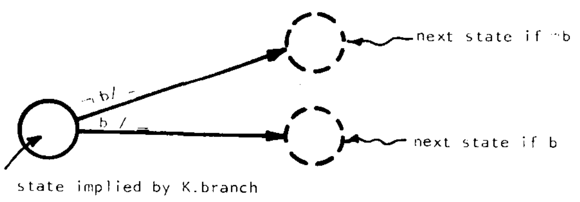

evoke this control
Boolean input
Kb:
Boolean to test
evn1/(evoke next if Boolean is true/1/yes
evn0/(evoke next if Boolean is false/0/no)

Fig. 2. Diagram for the control module *K*.branch.



K(*Branch*)

K.branch (*Kb*) provides for the routing of control flow based on the condition of a Boolean input. The diagram for *Kb* with its two inputs and two outputs is shown in Fig. 2. Each time a branch control is evoked, it in turn evokes either of the controls following it, depending on whether the Boolean input is true (*a* 1) or false (*a* 0). In terms of a finite state machine, *Kb* is a state with the capability of going to either of two next states, depending on a Boolean input. *K*.branch is as follows.



DM(*General Purpose Arithmetic*/*gpa*)

The *DM.gpa* allows arithmetic function results (data operations) that have been performed on its two registers *A* and *B* to be written into other registers (using the bus). Results can also be transferred (written) into *A* and *B* ($A\leftarrow$; $B\leftarrow$). The data operations are: $\leftarrow A$, $\leftarrow B$, $\leftarrow\neg A$, $\leftarrow\neg B$, $\leftarrow A+B$, $\leftarrow A-B$, $\leftarrow A-1$, $\leftarrow A+1$, $\leftarrow A\times 2$, $\leftarrow A\wedge B$, $\leftarrow A\vee B$, and $\leftarrow A\oplus B$. An input that evokes the function $\leftarrow$(Result)/2 can be combined with the previous function outputs to give $\leftarrow A/2$, $\leftarrow B/2$, $\leftarrow(A+B)/2$, etc. Two Boolean inputs, shift in $\langle 16, -1\rangle$, allow data to be shifted into the left- and right-hand bits on /2 and $\times 2$ operations, respectively. Bits of registers *A* and *B* are available as Boolean outputs.

K(*Bus Sense and Control Module*/*Bus*)

Each independent data bus in the system requires a centralized control module. It has a register, Bus, that always contains the result of the last register transfer that took place via the bus. *K*.bus carries out several functions: monitoring register-transfer operations; providing for single-step manual control for algorithm flow checkout by the user; providing for sense lights (indicators); providing for a word source of zero, i.e., $\leftarrow 0$; forming Boolean functions of the Bus register; power-on initialization; manual startup; and bus termination.

Design with RTM's

Digital systems engineers are concerned with formulating algorithms that, when executed by hardware, behave according to the solution of the original design problem. The solutions of digital systems design problems using programming, conventional logical design, and RTM design are all similar. The three design and implementation processes have the same goal: to construct a program for a machine, or a hardwired machine to execute the algorithm stated (or implied) in the problem. Thus, programming and digital systems engineering are concerned with interconnecting basic components or building blocks for executing algorithms; the building blocks are machine operations and logical design components, respectively. RTM's are a basic set of components for constructing hardware algorithms. That is, they are the components for digital systems design.

The design protocol using RTM's is very much akin to that of designing a program. The designer takes a natural language statement of the problem and carries out the conversion to a process description that acts on a set of data variables (and any temporary data variables). An RTM design has two parts: 1) the explicitly declared data variables and the implied data operations that are attached to these variables; and 2) the control part, a finite state machine, that accepts inputs and evokes the various operations on the data part. The control part is shown as a combined flow chart-wiring diagram.

Two examples show how this design is carried out. The schematic for the first example, an algorithm to sum integers, shows all wires and modules, and the schematic for the second example, a small stored program computer, shows the control flow and the data part, but excludes the connections between the control and data parts.

Example: Sum of Integers to N

A small system to sum the integers to N ($S \leftarrow 0+1+2+\cdots+N$) can be built that uses the four aforementioned modules: *DM.gpa*, *K*.bus, *K*.evoke, and *K*.branch together with a switch register to enter N, and a manual start control module to start the system. The data and control parts together are given in the RTM wiring diagram (Fig. 3); the data part is shown on the right and the control part on the left. The final result S and the variable N are held in a general-purpose arithmetic module *DM.gpa*. N is held in the switch register T initially. The control sequence is initiated by a *K*.manual-start (a human presses a key). Instead of counting to N, we start with N and count down to zero while tallying the sum S. The first control step reads T to register N, ($N \leftarrow T$). The second step initializes the sum S, ($S \leftarrow 0$). The inner loop consists of the three functions: $S \leftarrow S+N$; $N \leftarrow N-1$; and a test for $N=0$.

Example: A Small Stored Program Computer Design Using RTM's

Fig. 4 shows an RTM diagram for a small stored program computer that was initially constructed as an application experiment to demonstrate the feasibility of the modules and to investigate systems problems. The process of specifying the machine took approximately two hours (with three people). The computer was wired and, aside from minor system/circuit problems (for which the experiment was designed), the computer operated essentially when power was applied, since there were no logic errors. The computer was designed for an actual application that had about 300 constants, 600 control steps, and about 16 variables. (An alternative approach would have been to hardwire the 600 control steps directly, thereby operating faster, but being more expensive and less flexible.) The computer has only 24 evoke and 16 branch controls. (By way of comparison, RTM interpreters to emulate the PDP-8 and the Data General NOVA computers require about 90 evoke and branch control modules, 2 *DM.gpa*'s, and core memory.) Since the price ratio of a single hardwired control to a single READ-ONLY memory control word is approximately 10:1, and since the overhead price of a 1000-word READ-ONLY memory is about 100 controls, it was cheaper in the above application to use RTM's to first build an interpreter, commonly called a stored program digital computer, and then let the computer program execute the control steps.

The data part of the machine is shown in the upper right of Fig. 4. Three *DM*-type RTM modules hold the processor state and temporary registers. The processor state, that part of memory accessible and controlled by the program, includes: A, the accumulator; P the program counter; and L, a register used to hold subroutine return addresses (links). The temporary registers needed in the interpretation of the instructions are: i, instruction holding register; and B, used for binary operations on A (e.g., ADD, AND). Also connected to the RTM bus are the READ-ONLY and READ-WRITE memories and the Teletype, as well as a special input/output register interface to the remainder of the system.

The method of defining the interpreter can be seen from the RTM diagram (Fig. 4). The control part consists of three subparts: the START and CONTINUE keys that are used to initialize the processor to start at location 0, and to restart the processor; the instruction fetch; and the instruction execution. The instruction fetch consists of picking up the instruction from the memory word addressed by the program counter P and incrementing P to point to the next instruction. The instruction is placed in the i register, which has been specially wired to allow decoding of the three most significant bits. Individual bits of i can be tested for the OPERATE (OPR) instruction, and the address field $i\langle 10:0\rangle$ can be read.

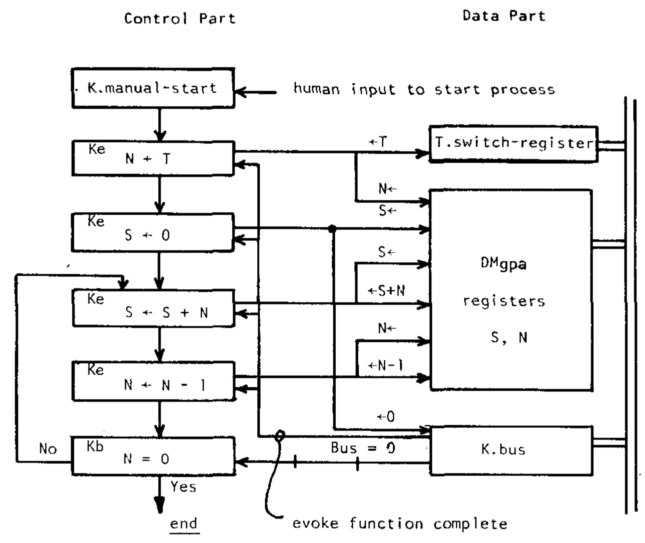
After the instruction is fetched and placed in i, $Ke(MA \leftarrow i\langle 10:0\rangle)$ is evoked to address data referenced by the instruction. Immediately following this evoke operation, an eight-way *K*.branch allows control to move to the one path corresponding to the operation code of the instruction being interpreted, that is, the instruction is decoded, and control is transferred to execute it. After the execution of the appropriate instruction, control returns to fetch the next instruction. For example, executing the ADD (two's complement add) instruction consists of loading the data from memory into the temporary register, B (i.e., $B \leftarrow MB$) and then adding B to the accumulator, A (i.e., $A \leftarrow A+B$).

For the OPERATE instruction, which does not reference memory, each bit of the address part of the instruction specifies an operation to be carried out on the accumulator (test for $-$ or 0, clear, complement, add one, shift right or left, or return from the subroutine). Each bit is tested in sequence, and if a one, the corresponding operation is carried out. If the instruction code with $op=6$ is given, the computer halts; pressing CONTINUE restarts it.

The instruction set is shown to be straightforward and simple. Subroutine return addresses are stored in a link register L. Thus to call subroutines at a depth of more than one level requires the called subroutine to save the link register in a temporary location. But there is no way of storing this register; thus it is difficult to call a subroutine and pass parameter information (e.g., the location of tables). Since the computer requires a few minor changes to allow nested subroutines and parameter passing, the reader is invited to make the appropriate incremental changes.

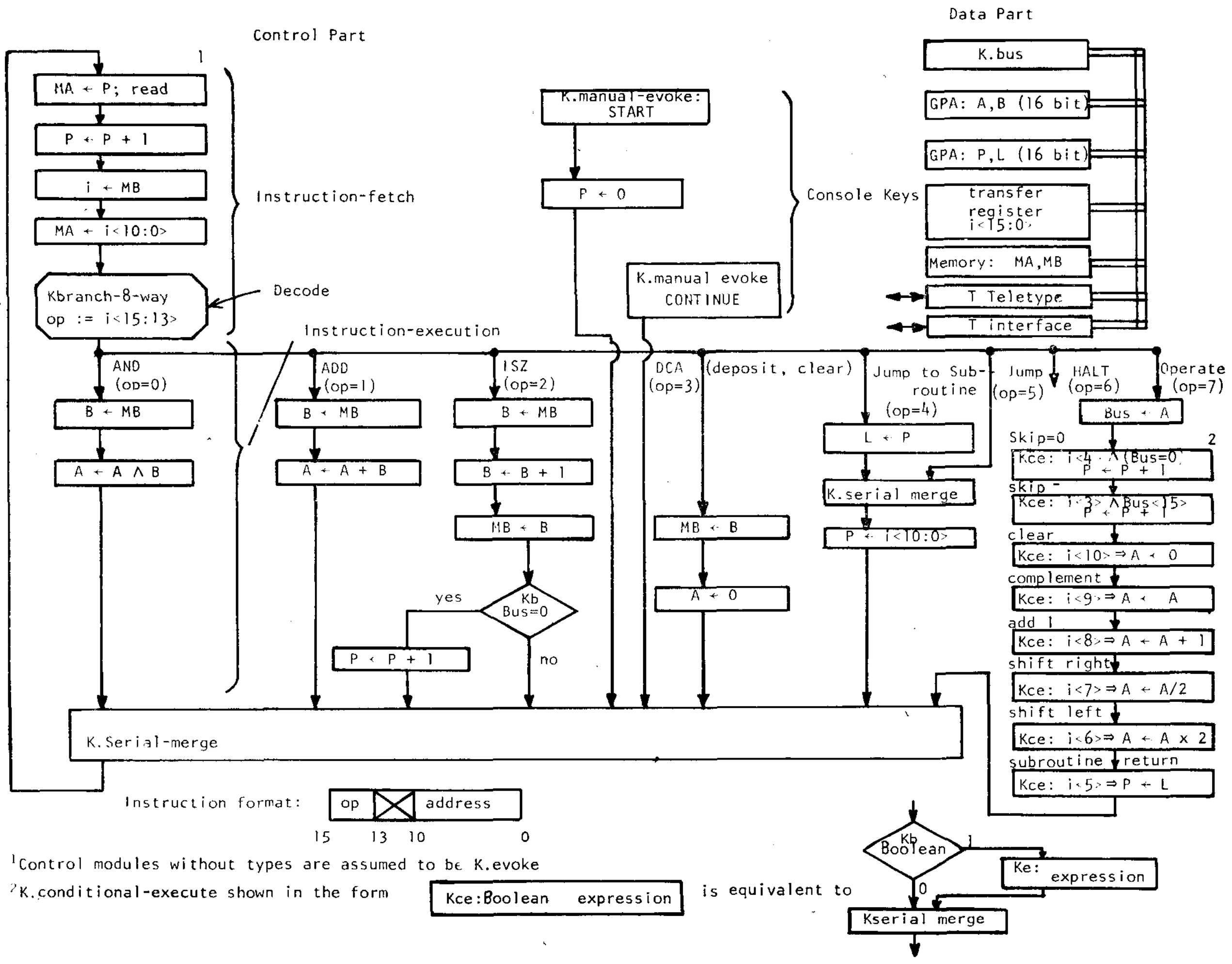
Conclusions

The concept of using high-level building blocks is not new, but we think this particular implementation of a set of simple blocks is quite useful to many digital systems engineers. The design time using this approach is significantly less than with conventional logical design. The modules are



Ke ≡ K.evoke module
Kb ≡ K.branch module
———— Control flow and evoke wires
════ Bus for data wires
—+—+— Boolean variable wires

Fig. 3. RTM digital system to take a value from a switch register input and to sum the integers to the input value.



[1]Control modules without types are assumed to be K.evoke

[2]K.conditional-execute shown in the form

Fig. 4. RTM design of a small stored program digital computer.

especially useful for teaching digital system design. We have solved many benchmark designs, with reasonably consistent results: the modules can be applied quickly and economically where there are between 4 and 100 control steps, a small READ-WRITE memory (100 words), and perhaps some READ-ONLY memory. Larger system problems are usually solved better with a stored program computer, although such a computer can be designed using RTM's. The user need only be familiar with the concept of registers and register operations on data, and have a fundamental understanding of a flow chart.

Acknowledgment

These modules were formally proposed in March 1970 in a form essentially described herein by one of the authors, C. G. Bell. In June 1970 the project was seriously started by constructing the computer of the previous example using them. The authors gratefully acknowledge the organization and management contributions of F. Gould, A. Devault, and S. Olsen (Digital Equipment Corporation) without whose goal-oriented commitment the RTM's could not have been built. The authors are also indebted to Mrs. D. Josephson of Carnegie-Mellon University for typing the manuscript.

References

[1] T. C. Bartee, I. L. Lebow, and I. S. Reed, *Theory and Design of Digital Systems.* New York: McGraw-Hill, 1962.
[2] Y. Chu, *Introduction to Computer Organization.* Englewood Cliffs, N. J.: Prentice-Hall, 1970.
[3] T. D. Friedman and S. C. Yang, "Methods used in an automatic logic design generator (ALERT)," *IEEE Trans. Comput.*, vol. C-18, pp. 593–614, July 1969.
[4] W. A. Clark, "Macromodular computer systems," in *1967 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 30. Washington, D. C.: Thompson, 1967, pp. 335–336. (Introduction to a set of six papers, pp. 337–400, in the same conference.)
[5] C. G. Bell and J. Grason, "Register transfer modules (RTM) and their design," *Comput. Design*, May 1971.
[6] C. G. Bell and A. Newell, "The PMS and ISP descriptive systems for computer structures," in *1970 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 33. Montvale, N. J.: AFIPS Press, 1966, pp. 351–374.
[7] ——, *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.