A COMPUTER ARCHITECTURE AND ITS PROGRAMMING LANGUAGE

Mario R. Schaffner
Massachusetts Institute of Technology
Cambridge, Massachusetts

1. INTRODUCTION

Computer architectures and programing languages are traditionally developed independently. Through suitable computer architecture, for instance, one can attempt to speed up the processing of a stream of data and instructions, leaving to the software the burden of preparing these streams. Through a suitable programing language, one aims at efficiently describing many classes of problems, in a phrase-structure form that is machine independent. This approach has led to the ever-increasing application of computers, but it has also brought about a growing complexity in the software systems. As a consequence of the latter, there is a new trend toward extending hardware implementation to replace the software ones, especially in view of the new technology of large scale integration.

In the past, several computer architectures were suggested toward the attainment of a larger computing power, such as the Solomon computer [1], the Holland machine [2], a spatially oriented computer [3], and a fixed plus variable structure computer [4]. Then, two basic techniques appeared of more general applicability, and led to actual implementations: parallel processing, and pipeline execution [5]. In parallel processing, an array of similar processors work simultaneously on different data, under the control of the same control unit. The modularity of such an organization is attractive in many respects. However, the performance is heavily dependent on parallelism in the problems [6], and programing techniques need to be developed, for exploiting the potential capabilities of the computer and the inherent parallelism in the computations [5]. Whereas for particular problems parallel computers can achieve a throughput which is orders of magnitude larger than that of conventional computers, for general problems they face a performance degradation that increases with the number of processors.

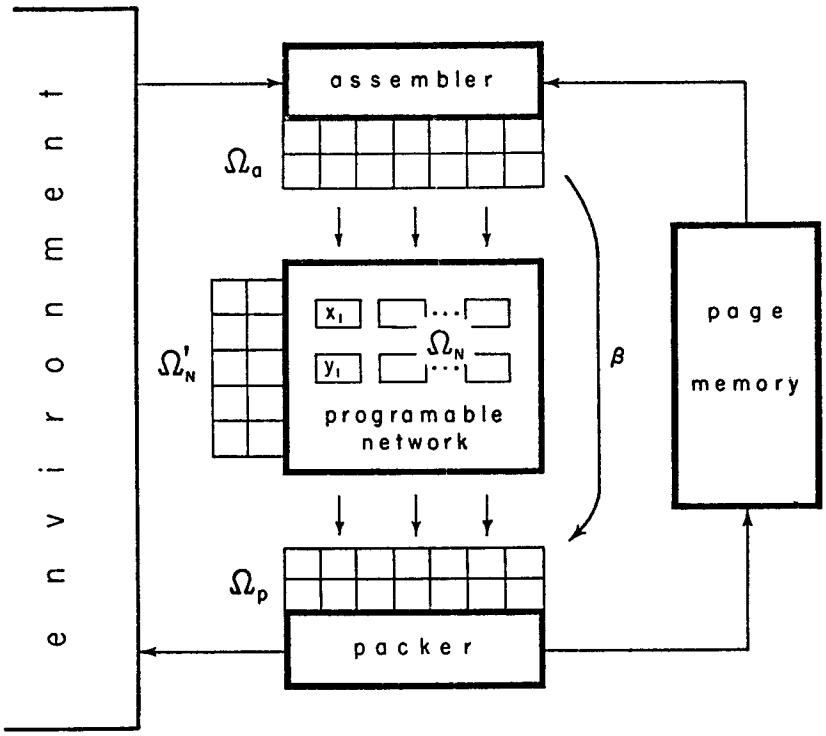
Pipelining consists of the concurrent execution of the various stages of the processing by independent units connected in cascade. This concurrency can be implemented at different levels [7]; the overlapping of the processor and memory operations [8], and that of the steps of arithmetic operations [9] are examples. The theoretical limits of pipelining have been analyzed [10]; in practice, advantages depend upon the presence of a stream of similar tasks [11]. All modern large computers have some degree of parallelism and pipelining. In order to analyze their organization, instruction and data streams can be defined, and the management of requests and services considered [12]. In this context, computer architectures are, at first, classified as single-instruction single-data streams, single-instruction multiple-data streams, multiple-instruction single-data streams, and multiple-instruction multiple-data streams.

The use of these computer architectures depends heavily on complex compilers, or interpreters. Compilation requires a preliminary run, generally produces no optimum codes and makes the debugging more difficult. Interpreters require a large memory space and produce a slow execution. For these reasons the question rises recurrently whether computers constructed to directly execute programs written in the user programing language could lead to a more efficient overall system [13]. The above question has prompted several works oriented to the hardware (or a mixture of hardware and software) implementation of the production of phrase-structure programing languages that are subsets of existing programing languages, or a slight variation of them. Some examples are: Anderson's [14] implementation of Algol 60, Bashkow's et al. [15] design of a Fortran machine, Weber's [16] implementation of EULER, Thurber's et al. [17] design of a cellular APL computer, the SYMBOL language and computer [18], and the APL implementation by Hassitt et al. [19]. All these studies show particular advantages in more closely relating the structure of the programing language and the structure of the computer hardware. However, no significant impact was made on the mainstream of computers, in which languages and hardware are developed independently. One can argue that in the above cases the languages used (at least basically) already existed and were developed independently of any particular architecture.

This paper shows a case in which computer architecture and programing language are not developed independently, but are treated as two isomorphic forms of representation of the same structure -- the abstract mechanisation of the processes as it is conceived by the user. The user models the desired process in the form of an abstract Finite State Machine (FSM), at a proper level, in terms of the elements of a language for describing FSMs. The description of this FSM constitutes a description of the desired process, but at the same time is also the specialized architecture of a hypothetical machine that executes that process. If a physical substratum (isomorphic with the language of the FSMs) is available, these hypothetical machines can be implemented, and the description of an FSM constitutes a program for this substratum. Such a substratum can be seen as an organizable computer. In this case, the distinction between hardware and software blurs.

This substratum, i.e. a programable architecture, is outlined in section 2; the isomorphic programing language is described in section 3; and in section 4, results and implications are discussed.

FIG 1



2. THE PROGRAMABLE ARCHITECTURE

The essential parts of this architecture are, Figure 1:

(1) a programable network PN comprising an array Ω_N of registers for holding a page of data, a second page array Ω'_N, and programable operational elements which can be connected to these registers with the related control circuitry for the execution of operations on the variables;

(2) a memory for holding pages of data, the structure of which can be programed in accordance with the data structures of the processes;

(3) an assembler which, receiving a page from the memory and new data from the environment, assembles the variables of a process and program words (that describe networks performing the operations of the present state of the process) into a page register-array Ω_a; and

(4) a packer which, receiving a page from PN into a page register-array Ω_p, provides the routing of output data to the environment, and the packing of the data needed in the future into the form of a page for the memory.

A page here is a self-sufficient set of data related to a job; in the memory, it contains the present variables of the process and a key word indicating the present state of the process; in the assembler and in PN, it also includes the new input data and the program words of the present state.

The basic page transfers can be described with the use of register transfer notations [20] by the expressions

$$t_1\alpha_1\Omega_a + t_2\gamma F_1(\Omega_N,\Omega'_N) + t_3\delta_1\Omega'_N \rightarrow \Omega_N$$

$$t_2\gamma F_2(\Omega_N,\Omega'_N) + t_4\delta_2\Omega_N \rightarrow \Omega'_N \qquad (1)$$

$$t_5\alpha_2\Omega_N + t_6\beta\Omega_a \rightarrow \Omega_p$$

where F_1 and F_2 are functions executed by the programable network; t_1, t_2, ... are Boolean time functions produced by the control; and α, β, ... are Boolean conditional coefficients with value, meaning, and constraints as shown in the table below.

α_1	α_2	β	γ	δ_1	δ_2	Condition
(1)	ϕ	ϕ	o	o	ϕ	acquisition of a new page
ϕ	(1)	o	o	ϕ	ϕ	recirculation of a page
ϕ	o	(1)	ϕ	ϕ	ϕ	recirculation of a page by-passing PN
o	o	ϕ	(1)	o	ϕ	processing of a page
o	ϕ	ϕ	o	(1)	ϕ	acquisition from storage
ϕ	ϕ	ϕ	ϕ	ϕ	(1)	storage of a page or data

The system can process in sequence all the pages through the paths α_1 and α_2; it can continuously process a single page, condition γ; it can input and output data without involving PN through the path β; it can buffer a page for a certain time in the auxiliary page array Ω'_N through the transfers δ_2; it can produce a new page in array Ω'_N during processing (combination of paths δ_1, δ_2 and γ) for the execution of a subtask; it can introduce the new page into circulation through transfer δ_1.

The registers of arrays Ω_a and Ω_p have one-to-one correspondence with the registers Ω_N embedded in the programable network. The packer transfers the data in Ω_p into the memory in an ordered form; the same order is used by the assembler to allocate the data of a page into Ω_a. In this way each variable of a process always goes into the same register of PN, during the circulation of the page, if not otherwise prescribed by the program. The memory moves the pages as a First-Input-First-Output storage, or with a different rule if indicated by the program. These features eliminate the need for explicit addresses. Addresses and their manipulation account for a large part of the memory capacity, and for most of the overhead of conventional computers. When selective access is required by a given process, the corresponding addresses are obviously part of the variables of that process; accordingly, the packer has the further feature of using some process variables also for directing other variables to specific parts of the data structure organized in the memory.

The programable network does not have *per se* a specific operational configuration. It is a collection of registers, multifunction elements, and preferred links among them. Program words enable simultaneous links and functions in order to implement specialized structures which perform the data transformations demanded in each state of a process. A kind of microprograming extended to its full allows the use of a large number of all possible combinations of the loose elements forming the programable network [21]. In this way, data transformations involving several variables are executed as a single large operation. Several different configurations can be implemented sequentially during one passage of a page through the network. The fact that the variables involved are all present in the network eliminates many intermediate steps and data movements that occur in conventional computers. When a process involves more variables than can be contained in PN, they are grouped in successive pages; the auxiliary register array Ω'_N, which is also part of PN, allows the sharing or transfer of data. The coefficients δ in expressions (1) can be applied to selected data or to the entire page.

It is interesting to note that the architecture of Figure 1 exhibits properties of many of the different architectures mentioned in section 1. The system has a pipeline configuration; while the programable network processes a page, the assembler assembles the next page, and the packer packs and routes the previous page. Parallel processing can be implemented simply by programing PN as a set of independent units, or, in virtual form, as a sequence of pages. The efficiency of a special-purpose computer can be achieved by structuring PN according to the specific process. But, because the specialization of PN can change at each cycle, the machine is a general-purpose computer. Because of the three basic features -- the organization of jobs into independent pages, the circulation of the pages in a pipeline configuration, and the loose structure of the processor and memory -- this architecture has been named the Circulating Page Loose (CPL) system.

3. THE PROGRAMING LANGUAGE

The most interesting peculiarity of the architecture described in the previous section is its programability. This programability permits the execution of the processes in terms of structures devised by the user each time, rather than as simulation by means of a given structure (arithmetic unit connected to a random access memory) and instructions of a given set. Thus, here, the programing language refers to operational structures and data structures, rather than to commands and declarations.

The primitive elements that have been found sufficient to efficiently express the variety of processes we give a computer are the following:

(i) a finite set of process variables x_r, a subset of which is indicated as X_q;
(ii) a finite set of input data u_r, a subset of which is indicated as U_q;
(iii) a finite set of output devices, and storages, z_r; and
(iv) a finite set of labeled process-states s_j, where a state is defined by:
  (v) a function F_j which produces new values for a subset X_a as a function of the values in subsets X_b and U_c,
  (vi) a function T_j which produces the label of the next state as a function of the values of subsets X_d and U_e, and
  (vii) a prescription R_j for routing some variables x_r to some output devices, or storages, z_r.

Time is represented as a sequence of discrete intervals i. A process is modeled as a finite-state machine represented by the following expressions, where the symbols refer to the primitives defined above:

$$\begin{aligned} X(i+1) &= F_{s(i)}\left[X(i),\ U(i)\right] \\ s(i+1) &= T_{s(i)}\left[X(i+1),\ U(i)\right] \\ s(i) &= s_1,\ s_2,\ \ldots\ s_j\ \ldots\ s_k \end{aligned} \qquad (1)$$

We must note that here states refer to phases of the model of the processes; they are neither the total internal states used in automata theory, nor the conditions of an implementation used in particular computers. These states are few and meaningful to the user. In each state, in general, there will be a different F and T. Functions F and T are thought of as operational networks; thus they can also be described in the form of digital words that implement those networks in a digital programable network [22]. In other words, we use the mapping

$$F \rightarrow N_F \rightarrow W_F \qquad (2)$$

where F stands for a description (in any language) of a data transformation, N_F stands for an operational network performing that data transformation, and W_F stands for a digital word describing (in a language) that network. This global treatment of the data transformations gives conciseness to the modeling of processes, and the use of corresponding global words W gives conciseness to the actual programs. A finite-state machine so formulated is denoted with capital initials Finite State Machine (FSM). The FSM is the modular block of the programs.

Complex processes are modeled in the form of several concurrent FSMs, each of which may be implemented simultaneously by many pages. The routing prescriptions R allow the interaction among FSMs necessary for their concurrent work. A page transfers through the states of an FSM, and can transfer also through different FSMs. Thus, a process is modeled as an interplay of processing structures (the FSMs) with data structures (the pages). A program can be composed of a single FSM and page; or one FSM related to many pages; or several FSMs, each one related to one page; or many FSMs, each with many pages.

The user develops the FSMs in the form of state diagrams. Figure 3 shows an example. The encircled domains represent a state; the data transformation F is described inside these domains; the transition functions T are described with conditions indicated below horizontal lines and arrows pointing to the new states; the routing prescriptions R are indicated, typically, in connection with those arrows. Which notations are used for expressing the variables, the F, T, and R is irrelevant at this stage. State diagrams of this form constitute a complete description of a process. As such they also constitute complete programs for a computer that is isomorphic to the language of the FSM. When all the elements of the state diagram (both those represented by graphic means, and those described by alphanumerical symbols) are expressed in the codes of that computer, the actual object program is obtained. The object program is in the form of a set of quadruplets

$$\left[W_I \, W_F \, W_T \, W_R \right]_j \qquad j = j_1, \; j_2, \ldots \; j_k \qquad (3)$$

where W_F and W_T are the words that implement specialized networks performing functions F and T, W_R is a coded form of the routing prescriptions R, and W_I is a coded representation of the input data set $U = U_c \cup U_e$. j is the state label, and k is the total number of states involved in that program.

The state diagram is problem oriented and machine independent in the sense that any hypothetical machine can be implied in its construction. When a state diagram is expressed in the form of specific quadruplets, it becomes machine dependent. The transformation between the ideal machine (the state diagram) and the executable program (the quadruplets), that is, the mapping (2), is made by the user. Because the user is expected to be familiar with his processes, and to know the preferred choices, the resulting object programs are efficient and easily understood.

On the other hand, one may think that in this way, the user is burdened with clerical tasks of which he is usually relieved by the compilers. But because of the isomorphism between the language of the FSM used for describing the processes and the architecture of the computer, it turns out that the user works on his problem and not on the intricacies of a computer which he is not interested in. Moreover, the results produced by the computer can be easily interpreted. As an example of the level of mechanization in which the user is involved, a program in the field of numerical solutions of partial differential equations is outlined below.

The dynamics of a hypothetical fluid are modeled in the form of an initial-value problem with boundary conditions. The analytical expressions considered are

$$\begin{aligned} \frac{\partial \eta}{\partial t} &= h_1 \frac{\partial \eta}{\partial x} + h_2 \frac{\partial \eta}{\partial y} \\ \frac{\partial \psi}{\partial t} &= h_3 \frac{\partial \psi}{\partial x} + h_4 \frac{\partial \psi}{\partial y} \\ \frac{\partial v}{\partial t} &= h_5 \frac{\partial v}{\partial x} + h_6 \frac{\partial v}{\partial y} \end{aligned} \qquad (4)$$

where η, ψ, and v are the variables of the system, and h_r the given parameters. The chosen finite-difference approximation is given by

$$\begin{aligned} \eta_{i,j}^{n+1} &= \eta_{i,j}^{n} - k_1\left(\eta_{i,j}^{n} - \eta_{i-1,j}^{n}\right) - k_2\left(\eta_{i,j}^{n} - \eta_{i,j-1}^{n}\right) \\ \psi_{i,j}^{n+1} &= \psi_{i,j}^{n} - k_3\left(\psi_{i,j}^{n} - \psi_{i-1,j}^{n}\right) - k_4\left(\psi_{i,j}^{n} - \psi_{i,j-1}^{n}\right) \\ v_{i,j}^{n+1} &= v_{i,j}^{n} - k_5\left(v_{i,j}^{n} - v_{i-1,j}^{n}\right) - k_6\left(v_{i,j}^{n} - v_{i,j-1}^{n}\right) \end{aligned} \qquad (5)$$

with the conventions:

$$\begin{aligned} x &= i\,\Delta x & i &= 1, 2, \ldots I \\ y &= j\,\Delta y & j &= 1, 2, \ldots J \\ t &= n\,\Delta t & n &= 1, 2, \ldots N \end{aligned}$$

and the k_r derived from the h_r.

To obtain the solution, an abstract machine is conceived, that has a page for each point of the two dimensional space of the system (Figure 2), a page for each boundary point, and a control page. The symbols η, ψ, and v, related to the variables of the process, are considered as names for three variables x_r; initial values a, b, and c of these variables are treated as input data u_r; the parameters k_r also are treated as input data. Moreover, an additional three variables x_r, named D, E and F, are used for temporary purposes. The pages related to the points of the fluid perform an FSM 3 as described in Figure 3, which implements expressions (5); the pages related to the boundary points perform an FSM 2 which implements a time evolution of the boundary values; and the control page performs an FSM 1 which controls the work of the entire system. The pages circulate in the structure shown in Figure 1, with different scanning as indicated in Figure 2.

Even without entering into the details of the language of the programable network, Figure 3 should convey the level of abstraction of these operational structures. For instance, FSM 1, which constructs and controls the entire

FIG 2

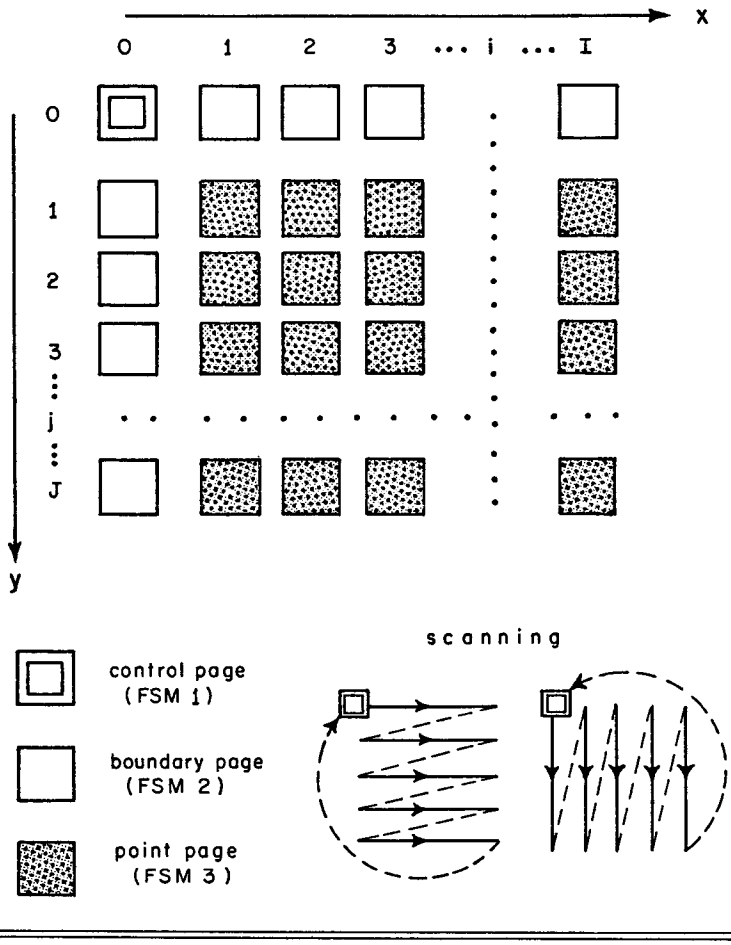
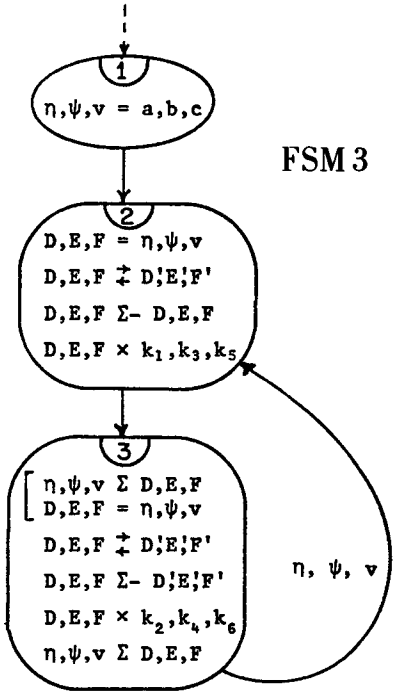
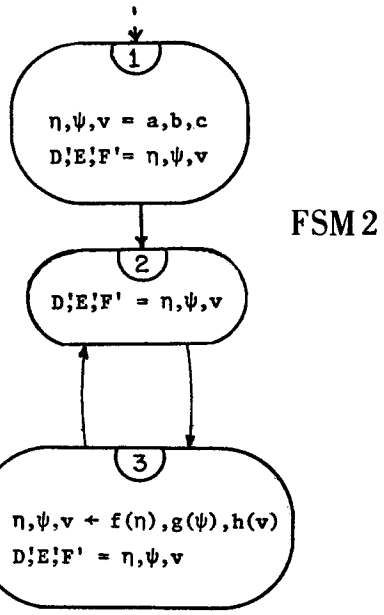
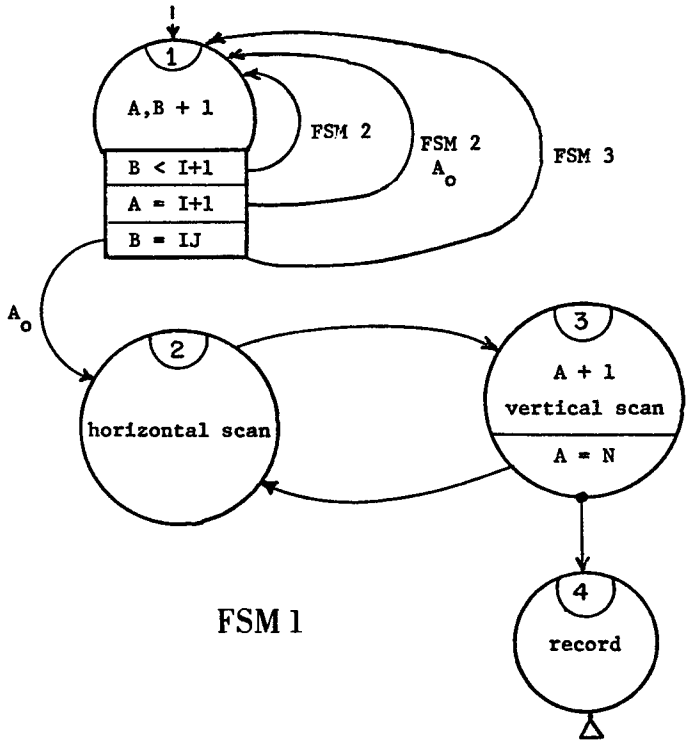

x
0 1 2 3 ... i ... I
0
1
2
3
j
J
y
control page (FSM 1)
boundary page (FSM 2)
point page (FSM 3)
scanning

FIG 3


1
η,ψ,v = a,b,c
FSM 3
2
D,E,F = η,ψ,v
D,E,F ⇄ D¦E¦F'
D,E,F Σ- D,E,F
D,E,F × k1,k3,k5
3
η,ψ,v Σ D,E,F
D,E,F = η,ψ,v
D,E,F ⇄ D¦E¦F'
D,E,F Σ- D¦E¦F'
D,E,F × k2,k4,k6
η,ψ,v Σ D,E,F
η, ψ, v


1
η,ψ,v = a,b,c
D¦E¦F'= η,ψ,v
FSM 2
2
D¦E¦F' = η,ψ,v
3
η,ψ,v ← f(η),g(ψ),h(v)
D¦E¦F' = η,ψ,v


1
A,B + 1
B < I+1
A = I+1
B = IJ
FSM 2
FSM 2
Ao
FSM 3
Ao
2
horizontal scan
3
A + 1
vertical scan
A = N
4
record
FSM 1

machine, has four states. State 1 is devoted to creating the page array indicated in Figure 2. In this state, function F consists simply in incrementing variables A and B by one. Function T is expressed as a self-explanatory decision table (the transition from the corner corresponds to the "else" condition). The routing is different for the different transitions and consists of creating pages related to given FSMs and in clearing variable A. State 2 prescribes a horizontal scanning of the pages. State 3 prescribes a vertical scanning, and provides for the test of the number of time steps. State 4 orders an output record of the computed quantities, and makes the pages disappear (transition to a triangle).

The computation of the variables η, ψ, and v at each point (FSM 3) is obtained by means of simple networks of a parallel nature established by the user in accordance with expressions (5). As an example, in state 1 of FSM 3, the input set U, consisting of the data a, b, and c, is transferred in parallel into the set X, consisting of the variables η, ψ, and v. In state 3 of FSM 3, there is a succession of five networks: the first produces simultaneously the accumulation of the original values of D, E, F into η, ψ, v, and the transfer of the original values of η, ψ, v into D, E, F; the second produces an interchange of values between D, E, F and D', E', F', which are variables that remain in Ω'_N of the network during the circulation of the pages; the third produces the subtraction of D', E', F' from D, E, F; the fourth produces the multiplication of D, E, F by the data set k_2, k_4, k_6; and the fifth the accumulation of the present values of D, E, F into η, ψ, v. A routing prescription sends the present values of η, ψ, v to an output storage.

Obviously, the interest for such constructs is not to make the user do what can be provided by a compiler, but to give the user the possibility either of providing what has not been anticipated by the software systems, or of obtaining specific optimizations. In this example, the aim was to minimize the memory and the execution time. The entire computation is made with 6IJ + 3(I+J) + 2 memory words. The machine cycles are (2N+2)(I+1)(J+1), with an average of four to five networks per cycle.

4. RESULTS AND DISCUSSION

The results obtained from the use of this architecture for processing in real-time radar signals have already been reported [23,24,25,26] Obviously, in these applications, advantage is derived from the capability of the programable network to perform complex operations in one cycle, and to structure the memory in accordance to the stream of data. An application of significant interest is a program for processing weather-radar signals in real time that discriminates weather echoes from ground echoes, during the normal operation of the radar.

The easy interaction between user and computer is also very significant. The fact that the same FSM is both the model of the process used by the user and the program actually executed by the computer, makes it possible to develop a program in "real time" as suggested by the results. In the line of the mechanization of Figure 3, programs have been experimented that acquire actual initial data, in real time, from a weather radar, and then produce different evolutions of the precipitation pattern in terms of the values of parameters set by the user at each time, or modifications of the programs.

In research work, data processing is typically achieved today by means of systems comprising several special-purpose units and a general-purpose computer. The former efficiently execute the particular data transformations demanded in the process , and the latter provides for the computation and the control of the entire system. In these cases, a single computer with a CPL architecture could advantageously perform all these activities. The programable network is capable of executing both the particular data transformations and the computations; the programs in the FSM form are particularly suitable for controlling complex activities; and the organization of data in the form of pages makes efficient use of the memory capacity.

Another field for which this architecture is particularly efficient is that in which differential analyzers were advantageously used [27]. The PN can be programed in the form of integrators, and each page takes on the role of a term in a system of equations. Transformations such as the Fast Fourier Transform, similarly, can be executed efficiently by properly organizing the pages and configurating PN for complex butterflies [28]. This architecture has also been suggested for the computers in an integrated telecommunication network [29].

But the fact that this architecture accepts directly programs expressed in the FSM form and these programs correspond to the image of the process as developed by the user, triggers a more general interest in this approach [30]. The next subject of study that seems deserving of attention is the feasibility of a programing language that shares the flexibility and conciseness of the FSM and the adaptability to different forms of expression offered by the well-established use of compilers.

As far as the hardware software trade-off is concerned, this architecture constitutes an interesting new approach. The programability of the hardware configuration allows the efficiency peculiar to the hardware implementations together with the flexibility characteristic of the software implementations. Moreover, the description of these configurations is also interesting as a programing language *per se*. In using the first machine constructed with this architecture [31], we consistently find that programs in the form of FSM are much simpler than the equivalent programs in conventional machine language, and they have a complexity comparable to that of programs expressed in high level language. For complex processes, the programs in FSM form seem to be simpler than the equivalent ones in high level language. This finding has an interesting similarity with von Neumann's contention that for complex automata the description of an automaton is simpler than the description of the process performed by the automaton [32].

ACKNOWLEDGEMENT

This work has been supported, at different times, by the National Aeronautics and Space

Administration under contracts NASW-2276, NSR-09-015-033, and NASr-158.

REFERENCES

1. Slotnick, D.L., W.C. Borck, and R.C. McReynolds, "The Solomon Computer", Proc. Fall Joint Computer Conf., 97-107, 1962.
2. Holland, J.H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously", Proc.Eastern Joint Computer Conf., 108-113, 1959.
3. Unger, S.H., "A Computer Oriented Toward Spatial Problems", Proc.IRE, 1744-1750, 1958.
4. Estrin, G., "Organization of Computer Systems - The Fixed Plus Variable Structure Computer", Proc.Western Joint Computer Conf., 33-40, 1960.
5. Hobbs, L.C., and al. (Ed.), Parallel Processor Systems, Technologies, and Applications, Spartan Books, N.Y., 1970.
6. Chen, T.C., "Parallelism, Pipelining and Computer Efficiency", Comp.Res., 10, 69-74, 1971.
7. Ramarmoorthy, C.V., and S.S.Reddi, "Towards a Theory of Pipelined Computing Systems", Proc. 10th Allerton Conf. on Circuit and System Theory, University of Illinois, Urbana, Ill., 759-768, 1972.
8. Buchholz, W. (Ed.), Planning a Computer System, McGraw-Hill, N.J., 1962.
9. Hallin, T.G., and M.J.Flynn, "Pipelining of Arithmetic Functions", IEEE Trans. C-21, 880-886, 1972.
10. Cotten, L.W., "Maximum-Rate Pipeline Systems", Proc.Spring Joint Computer Conf., 581-586, 1969.
11. Graham, W.R., "The Parallel and the Pipeline Computers", Datamation, Vol.16, April, 1970.
12. Flynn,M.J., "Some Computer Organizations and their Effectiveness", IEEE Trans. C-21, 948-960, 1972.
13. McKeeman, W.M., "Language Directed Computer Design", Proc.FJCC, AFIPS Vol.31, 413-417, 1967.
14. Anderson, J.P., "A Computer for Direct Execution of Algorithmic Languages", Proc. Eastern JCC, 184-193, 1961.
15. Bashkow, T.R., A.Sasson, and A.Kronfeld, "System Design of a Fortran Machine", IEEE Trans. EC-16, 485-499, 1967.
16. Weber, H., "A Microprogrammed Implementation of EULER on IBM System/360 Model 30", Comm. ACM, 10, 549-558, 1967.
17. Thurber, K.J., and J.M.Myrna, "System Design for a Cellular APL Computer", IEEE Trans. C-19, 291-300, 1970.
18. Chelsey, G.D., and W.A.Smith, "The Hardware -Implementated High-Level Machine Language for SYMBOL", Proc. SJCC, AFIPS Vol.39, 563-573, 1971.
19. Hassitt, A., J.W.Lageschulte, and L.E.Lyon, "Implementation of a High Level Language Machine", Comm.ACM, 16, 199-212, 1973.
20. Bartee, T.C., I.L.Lebow, and I.S.Reed, Theory and Design of Digital Machines, McGraw-Hill Co., New York, 1962.
21. Schaffner, M.R., "A System with Programable Hardware", Digest 5th IEEE Int.Computer Conf., Boston Mass., 17-18, 1971.
22. Schaffner, M.R., "A Procedure for Describing Discrete Processes", Proc. 10th Allerton Conf. on Circuit and System Theory, Urbana, Ill.,462-470, 1972.
23. Austin, P.M., and M.R.Schaffner, "Computations and Experiments Relevant to Digital Processing Weather-Radar Echoes", Prepr. 14th Weather Radar Conf., 375-380, 1970.
24. Schaffner, M.R., "Computers Formed by the Problems, rather than Problems Deformed by the Computers", Digest 6th IEEE Int.Computer Conf., San Francisco, Cal., 259-264, 1972.
25. Schaffner, M.R., "On the Data Processing for Weather Radar", Prepr. 15th Conf. on Radar Meteor.,368-373, 1972.
26. Schaffner, M.R., "Echo Movement and Evolution from Real-Time Processing", Prepr. 15th Radar Meteor. Conf., 374-348, 1972.
27. Sizer, T.R.H., (Ed.), The Digital Differential Analyzer, Chapman and Hall, Ltd., London, 1968.
28. Schaffner, M.R., "Study of the Applicability of the CPL System to Doppler Radar Signal Processing", National Center for Atmospheric Research, Boulder, Col., 1973.
29. Cappetti, I., and M.R.Schaffner, "Structure of a Communication Network and Its Control Computers", Proc. Symp. Computer-Communications Networks and Teletraffic, M.R.I., Vol. XXII, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 1972.
30. Schaffner, M.R., "Study of a Self-Organizing Computer", Final Report, Contract NASW-2276, National Aeronautics Space Administration, 1973.
31. Schaffner, M.R., "A Computer Modeled After an Automaton", Proc.Symp.Computers and Automata, M.R.I. Symp., Vol.XXI, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 635-650, 1971.
32. von Neumann, J., "The General and Logical Theory of Automata", Hixon Symposium, 1948, Pasadena, Cal.; reprinted in John von Neumann, Collected Works, (A.H.Taub, Ed.), Pergamon Press, 1963.