Some Aspects of the Symbolic Manipulation of Computer Descriptions *

M.R. Barbacci and D.P. Siewiorek
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

July, 1974

ABSTRACT

Traditionally computer descriptive languages have been designed primarily for human communication and/or simulation. Due to this narrow range of applications the existing languages have taken on a strong degree of similarity. In this paper we present some applications in the realm of automatic design of both hardware and software where a computer description language could serve as the information exchange media between the user and the design automation system. The paper discusses an environment for research on the applications of computer descriptive languages, emphasizing the multiplicity of of users and tasks that may coexist an any point in time. Some properties needed in a computer descriptive language are presented. A structured programming approach to hardware design is presented by example.

---

* This paper describes a current research effort at Carnegie-Mellon University. The authors wish to make clear the active role being played in this research project by many other members of the CMU community: Samuel Fuller, Paul Hilfinger, David Jefferson, Karla Martin, Joseph Newcomer, Allen Newell, John Oakley, Mary Shaw, Richard Swan, and William Wulf.

This work is supported in part by the Advanced Research Projects Agency (ARPA) of the Department of Defense, under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research and by the National Science Foundation under grant GJ 32758X.

INTRODUCTION

Traditionally computer descriptive languages have been designed primarily for human communication and/or simulation [Chu, 1965; Bell, 1971]. Due to this narrow range of applications the existing languages have taken on a strong degree of similarity [Barbacci, 1973a]. There are other applications in the realm of automatic design of both hardware and software where a computer description language could serve as the information exchange media between the user and the design automation system. By examining these applications the information requirements can be determined and from these a language that serves for several (but still not necessarily for all) applications can be designed.

This paper describes some preliminary results of a research group at Carnegie-Mellon University. We present a case for machine-relative software and other related areas of research. A brief discussion of the domain of tasks we are considering is followed by a more detailed description of the requirements for two of them, namely the design of machine relative compiler-compilers and the design of modular hardware systems. We present an overview of an environment for research in these multiple applications. The key word here is "multiple". We visualize a system that will support multiple, concurrent users, investigating different aspects of the problem domain, implementing subsystems in different *programming* languages which manipulate machine descriptions given in different *computer description* languages. One of the key issues is the specification of adequate computer description languages. We

discuss some properties desired in such notations and, finally an example in a structured programming approach to top-down computer design is used to present some of our ideas in just one of the several areas of our research interests, albeit a crutial one.

MACHINE RELATIVE SOFTWARE

There is a continual stream of new machines spurred by the advent of minicomputers and microprocessors. Each machine has a different Instruction Set Processor (ISP) [Bell, 1971]. The emergence of microcoded systems with the option of user defined instructions has increased this flow of ISPs. Each new system requires supporting software and the amount of software grows for any individual system as user requirements grow.

There are a number of directions in which to seek a solution to ease the burden of software development. Standardization of software packages written in high level languages such as Algol, FORTRAN, and COBOL is one approach. It reduces the amount of software needed for each new machine. A second direction is in terms of better software production systems. This may be sought either in terms of implementation systems (high level languages specifically designed to aid implementation) or in terms of better software methodologies (e.g., structured programming). Another direction, which we will consider in detail, is to relativize the production of software to the description of the machine.

The central ingredient of this latter approach is the description of computer systems in a symbolic form, such that a range of problems can be solved by manipulation of these descriptions. We stress the need for diversity in the problem domain if we are really to understand how to operate relative to computer descriptions.

The next section will illustrate some points in the problem domain.

APPLICATIONS OF COMPUTER DESCRIPTIONS

To be clear about the multipurpose character of a computer description, let us list several kinds of problems that one might want to solve, each of which requires an abstract description of a computer.

1) ***Compiler-Compiler.***– A system that takes as input a description of a language and a description of a machine and outputs a compiler for that computer. Given the state of the art, the language would probably be restricted to be Algol-like. [Miller, 1971] is an early attempt at a solution to this problem.

2) ***Verification of I/O programs.***– Given an I/O program, such as a device handler, and a description of both the computer and the hardware device controller, verify that the program works. This problem has some special features that set it apart from the general program verification problem, besides its importance as an applied task: (a) its strong dependence on the description of computer systems in classic form (i.e., at the Register Transfer level) rather than in some abstract semantics, (b) the programs themselves may not be very complex in terms of their algorithms; rather the complexity of the task arises from the openness of the environmental states that have to cope with (timing, concurrency, etc.)

3) ***Programming of Microcoded Special Computers.***– The ability to create specialized computers to perform particular narrow classes of algorithms economically opens a world of device dependent, one-time programming tasks

that poses an immense problem. These systems attempt to optimize performance; their organization cannot be dictated by considerations of programming ease. Their programming will become difficult in the extreme, especially when no opportunity will exists for the growth of programming know-how. This suggests that what the human will do is to program relative to a machine description that he has barely assimilated. Hence it is reasonable to construct programming systems that operate relative to machine descriptions of a class of machines.

4) *Design of Modular Systems.*- Given a desired machine described in terms of some specification language, and given a space of machines defined by a class of Register Transfer [Bell, 1971] level modules, design a machine according to various constraints and criterion functions. This is a classic design situation which is worth studying, both in terms of understanding the nature of design and in terms of automating computer design. The feasibility of this approach has been demonstrated by the EXPL system [Barbacci, 1973b].

5) *Design to specification.*- Given a functional specification for a computer and a space of computer systems defined by a computer description language, design a computer that performs to the specification. This is another form of the classical design task. It differs from (4) above. A typical task here is: given some general functions, create an ISP for a computer. A typical task in (4) is: given an ISP, design it in terms of Register Transfer level modules. Formally they may seem identical, but the design spaces look quite different.

6) *Design Verification.*- Given a specification for a computer and a description of that computer in the language, verify that the computer satisfies the specification. We can also include here the automatic generation of testing and diagnostic programs.

7) *Manual generation.*- Given a computer defined in the language, create the documentation for the computer. This task is quite different from the ones above, but also involves understanding and manipulating a computer description.

The applications listed above place a variety of demands on the computer descriptive language and it is hardly clear whether a single language can cover the entire spectrum. The next sub-sections give some examples of the requirements for two rather different tasks and an outline of a possible system to meet the variety of requirements.

Machine Relative Compiler-Compilers.– By "machine relative" we imply an extension to the traditional definition of a compiler–compiler, in which a specific target machine is assumed. Due to this limitation, compiler–compilers have solved only part of the automatic programming problem and as a result they have not been very succesful. A better approach has been to produce a compiler that generates pseudo–machine code. For each new ISP the programmer simply provides the equivalent of the pseudo–machine instructions in terms of macros written in the target machine language [Feldman, 1966]. While runnable programs are produced by this technique they are poor in terms of size and run time efficiency. There are several reasons for this lack of efficiency: built–in preconceptions about existing instructions, the introduction of an extra level of abstraction that must be hand translated, the lack of consideration for specific machine features that can do certain things more efficiently that others, etc.

Hence we are primarily interested in generating an optimizing compiler. In order to generate machine code that will rival that of a good programmer, a compiler–compiler must extract the idiosyncrasies of the machine. For example, one way to add four to a register in the PDP–11 [DEC, 1973] is to use the instruction "ADD #4,R1". This requires two 16–bit words, one for the instruction and one for the immediate operand 4. However, the autoincrement addressing mode adds two to a designated register after using its contents as the address of an operand. Thus an instruction that effectively is a No–Operation code and uses the autoincrement mode on the register for both source and destination operands can achieve the effect of adding 4 to the register. Thus "CMP (R1)+,(R1)+" will add 4 to R1 and requires only one 16–bit word. Note that the compare instruction is not a true NOOP since it will set the

condition code registers according to the result of the comparison. The compiler has to insure that this side effect is not critical. One such critical case would be if the contents of R1 is used as a loop index and a loop exiting branch was to follow the addition. Note further that knowledge of the relative speed of instructions and addressing modes may be necessary to make a choice on the basis of speed.

Some of the information that needs to be extracted from the machine description is: the data types (address, integers, floating point, etc), operations on the data types (add, subtract, multiply, etc), location of data types (memory, register, etc), and instruction side effects (condition codes, use of hidden operands, etc). Instruction side effects are particularly important. The following PDP-11 code sequence is a good example:

```
SUB  A,B
TST  B
BLE  LABEL
```

where the TST instruction serves only to clear the overflow condition code. If the Branch on Less or Equal instruction (which is conditioned by the overflow condition code) is replaced by a Branch on Equal instruction (not dependant on the overflow condition) then the test instruction is superfluous and can be deleted.

One of the desired goals of a compiler is to produce the minimum cost code sequence which evaluates a given program. It is therefore necessary to explore all possible sequences that represent the evaluation and are semantically equivalent and eliminate those that exceed the least-cost criteria. This semantic equivalence is

related to the effect on the global program state in the context in which the sequence is to be executed. It is therefore necessary to express the global program state conditions under which a code sequence can be applied, as well as the resulting transformations on the state. This synergistic effect of machine language instructions has not been considered part of the realm of traditional computer description languages.

The cost of compile time generation of cases must be weighted against the advantages of finding the best code sequences. An intermediate solution is the exhaustive generation of templates to guide the code generation, as in traditional compilers. This once-only exhaustive generation process is more likely to find all the obscure cases and discover unspected semantic equivalences than hand-designed templates [Newcomer, 1974].

Modular Design.- Now consider a modular design program that produces a finished machine design in terms of a predescribed module set. A modular implementation of a system can usually be divided into a data part and a control part that directs the actions of the data part [Bell, 1972]. The data types and their operations can be implemented via templates of modules. Again, as in the case of the compiler-compiler, synergistic effects must be discovered in order to produce the most efficient network of modules for a given machine description. This implies certain commonality of information required by this two applications. However, there are many details of a module set that the compiler-compiler does not need to know. Assume that the modules are commercially available semiconductor chips and that the output from the design program is a printed circuit board layout. Knowledge of chip orientation, power requirements,

and chip spacing is needed by the design automation system to produce a wiring list.

Hence there is information contained in the computer description that is required by two or more applications while some other information is particular to a single application.

A research environment for the symbolic manipulation of machine descriptions. – The similar requirements among the several applications of computer description languages suggest a research environment centered around a data base in which machine descriptions and manipulation programs are maintained, as depicted in Figure 1.

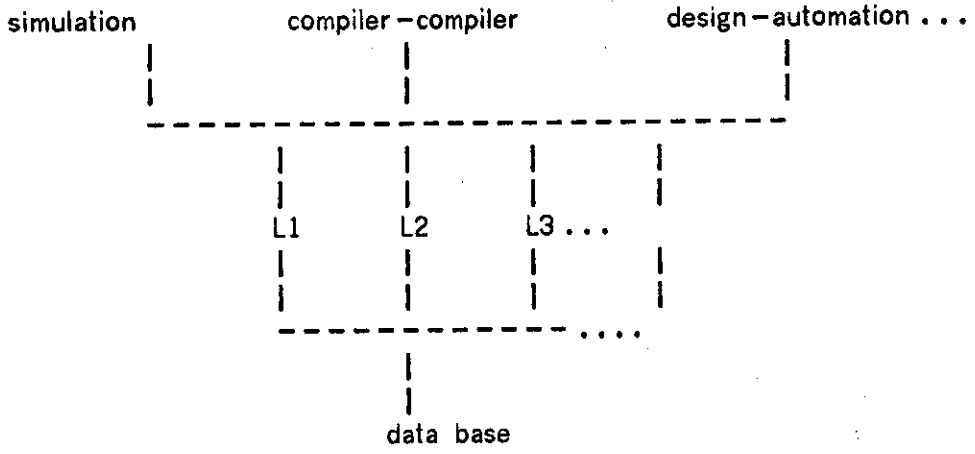

Figure 1. The environment

The user inputs information into the data base via one or more computer description languages. The application programs manipulate the global data base to extract information in the format desired by the application.

The data base and its manipulation programs must be able to support many

different notations and areas of application. This can be expressed by the following set of required features:

1) Must hold all computer descriptions for the different applications.

2) Must be reasonably independent of any particular *programming* language. This is necessary to allow researchers the flexibility to implement application programs (i.e. computer description manipulators) in a programming language of their choice (e.g., FORTRAN, Algol, APL, LISP, BLISS, etc.)

3) Must be independent of any particular *computer description* language. The reason is that the computer descriptive language used to create elements of the date base is a moving target. It is also the case that some notations may be more suitable than others for specific parts of a machine description. This implies an evolutionary process, during which many different notations can be in use simultaneously.

4) Must be interactive to allow casual and non-casual use. This requires a set of facilities for interaction in at least one language.

5) Must allow incremental use by many simultaneous users. By incremental use we mean the ability to carry a design through stages of completeness during which different users add application dependant details to a computer description. This is needed for experimentation.

The features outlined above present a set of requirements that may be conflicting. One of the reasons for this generality, not addressed in previous applications, is that the objects we want to manipulate, namely computer descriptions represent a tremendously large domain. We are talking not only about hardware (Logic, Register Transfer, and PMS levels [Bell, 1971]) but also about algorithms (Instruction Set Processors and programs). It is also the case that we are trying to apply a coherent methodology to hardware design, a domain characterized by rather abrupt transitions between its descriptive levels (more so than among software levels).

Ideally we would like to converge on a single computer descriptive language so

that people in the environment can interact more easily among themselves. On the other hand, we recognize the fact that notations go through evolutions and the research environment must be open along this dimension. Any kind of tight association between a computer description language and the data base will reduce the latter's usefulness.

The next section describes some thoughts about the requirements of a computer descriptive language. At this point in time, however, we hold no commitments to any particular existing language or combination of languages. This allows us the freedom to speculate and experiment with several, perhaps conflicting ideas. Therefore, our use of a particular syntax in the example given as a structured programming approach should not be construed as a language definition.

REQUIREMENTS OF A COMPUTER DESCRIPTIVE LANGUAGE

One of the problems with existing hardware descriptive languages is that they tend to bind the user to a view of the world that is rigid and difficult to modify. We feel that the semantics of the language should be under control of the designer. The following are a desireable, but by no means exhaustive, set of properties for the language:

1) *Neutrality.*– The language should not make any assumptions about the physical implementation. The control primitives available in the language determine the control structures that are easy to describe. If the language control primitives are too rigid they will limit the implementation alternatives. For instance, CASSANDRE [Anceau, 1969] uses state registers as primitives. Systems which do not decode values from centralized state registers are therefore difficult to describe.

2) ***Fidelity.***– The description should make the intentions of the designers transparent to the users. This is somewhat in conflict with the neutrality property.

2.1) ***Timing Fidelity.***– Existing languages such as ISP [Bell, 1971] describe algorithms with no reference to timing. Thus it becomes difficult to express the behavior of low level components. Another example is the description of cooperating parallel processes, such as interrupt systems, where timing is critical.

2.2) ***Structural Fidelity.***– Data paths can be inferred from the description but these may be a maximal set and may not reflect the actual structure of the machine. At some level of description the transfer operation, usually denoted by "←", means "by whatever path available". For a more detailed description the "←" correspond one–to–one with physical data paths. The same remarks can be applied to the specification of the functional units in the system. The presence of a "+" operator in a register transfer expression does not indicate which of possibly many functional units is to carry out the operation.

3) ***Hierarchy.***– Frequently systems design is conducted in a top down manner. The various portions of the system are first described at a high level. Then the designer specifies one subsystem in more detail, then another, and so forth. At any given time a systems design might consist of some subsystems designed down to the gate level, some less detailed designed at the register transfer level, and some merely described as algorithms. The coexistance of multiple levels of description is difficult to attain in existing design languages where top down refinements, if possible at all, are performed on a global basis by ad–hoc manual procedures. The addition of a clock at some level of detail, for instance, requires the rewriting of the entire description. Any validation that has been performed on part of the description would have to be redone.

The final section introduces, via examples, some thoughts on new mechanisms for a computer descriptive language that attempt to satisfy some of the above requirements.

A STRUCTURED PROGRAMMING APPROACH TO A COMPUTER DESCRIPTION PROBLEM

This section presents, via examples, some aspects of the use of new computer description concepts. We will present our ideas as an exercise in top down design. The objective is to design a PDP-8 like minicomputer, starting from a high level description and carrying the design down to a level in which the specific implementation of the machine is described. We will make use of some structured programming concepts that allow us to define entities of the machine (e.g., memories, registers, functional units) independently from the use of the entities in the description. These concepts will be added to the descriptive language ISP [Bell, 1971]. The choice of ISP as a framework is based on the authors familiarity with the notation and not, on a commitment to addopt an ISP derived notation as the only vehicle for our research. Our concern for allowing evolutionary notations is also reflected in certain liberties we have taken with respect to the syntax of the language as published in [Bell, 1971].

The concept of form [Wulf, 1974] allows us to define the data types available in the language by specifying not only the representation of the typed objects but also the operations that can be performed on these objects. A typical form declaration consists of a header and a body. The form header specifies the form name and the formal parameters used inside the form body. The form body consists of a declaration part, in which variables to be used in the form functions can be defined, and a set of functions and operations describing the operations that can be performed on variables declared as instances of the form.

For instance, we can define a form "memory" that describes a particular hardware component. At some early point in the design process a memory can be considered as a vector of integers, thus avoiding the specification of things like word length, number representation, addressing, etc. The following example is an instance of such high level memory definition*. Two functions (operations), "read" and "write" are defined as accesses to a vector of integers:

```
form memory(integer size) =
    {declare m = integer vector(size);
    function read(integer addr) = return m[addr];
    function write(integer addr,val) = m[addr]←val;
    export read, write }
```

The export statement is used to indicate the form entities (variables and operations) that are accessible to the rest of the program. Thus we can restrict the access to certain elements of the form by not exporting them. The read and write functions are evoked automatically, depending on the context in wich the memories appear, i.e., as a source (read) or a destination (write) in a statement.

Similarly, we can define a form "register" that behaves like an integer:

---

* In order to keep the examples within a reasonable size, we are appealing to the intuition of the readers to supply some of the missing details concerning the semantics of the forms. In order to make the process easier, we have taken some liberties with the syntax of ALPHARD and its forms [Wulf, 1974].

```
form register =
     {declare r=integer;
     infix +(register a,b) = return a+b;
     infix -(register a,b) = return a-b;
     infix *(register a,b) = return a*b;
     infix ÷(register a,b) = return a÷b;
     function read = return r;
     function write(integer val) = r←val;
     export +,-,*,÷, read, write}
```

The infix declaration is used to define binary infix operations on instances of the form. Notice that there is nothing in this definition that reveals the nature of the register and its structure. A more realistic definition would be the following:

```
form register(integer size) =
     {declare r=bit vector(size);
     function value =
          begin declare integer sum;
               sum←-r[1];
               incr i from 2 to r.size do sum←sum*2+r[i];
               return sum;
          end;
     infix +(register a,b) = return a.value+b.value;
     infix -(register a,b) = return a.value-b.value;
     infix *(register a,b) = return a.value*b.value;
     infix ÷(register a,b) = return a.value÷b.value;
     function read = return r.value;
     function write(integer val) =
               decr i from r.size to 1 do begin r[i]←val mod 2; val←val ÷ 2; end;
     export +, -, *, ÷, read, write };
```

In the example above, the register is defined as a vector of bits and the value of the register is encoded using the two's complement representation. The function "value" is not exported, thus the real nature of the register as a bit vector is hidden. The read and write functions are redefined to allow the transfer of values in and out of

the register. The "dot" notation is used here to indicate the access to an attribute of a register. Thus r.size is the register size, as specified in the declaration.

Top Level Description.– The following description of the PDP–8 assumes the register and memory forms defined previously. For the sake of brevity we are not defining the IO_EXECUTE and OPR_EXECUTE processes evoked by the EXECUTE process.

```
declare memory M[0:4096];
declare register AC<0:11>,
     IR<0:11>,
     PC<0:11>,
     L<>,
     LAC<0:12>:=L□AC,
     DATA_SWITCHES<0:11>,
     STOP_SWITCH<>,
     CMPA<0:11>,
     OP_CODE:=IR<0:2>,
     PAGE_BIT:=IR<4>,
     INDIRECT_BIT:=IR<3>;

INTERPRETER:=(IFETCH;next DFETCH;next EXECUTE;next INTERPRETER);

IFETCH:=(IR←M[PC];PC←PC+1);

DFETCH:=(COMPUTE_ADDRESS;next DEFER_ADDRESS);

EXECUTE:=(
     (OP_CODE='AND'⇒AC←AC∧M[CPMA]);
     (OP_CODE='TAD'⇒LAC←LAC+M[CPMA]);
     (OP_CODE='ISZ'⇒M[CPMA]←M[CPMA]+1;next
          (M[CPMA]<0⇒PC←PC+1) );
     (OP_CODE='DCA'⇒M[CPMA]←AC;AC←0);
     (OP_CODE='JMS'⇒M[CPMA]←PC;PC←CPMA+1);
     (OP_CODE='JMP'⇒PC←CPMA);
     (OP_CODE='IO'⇒IO_EXECUTE);
     (OP_CODE='OPR'⇒OPR_EXECUTE)
     );

COMPUTE_ADDRESS:=(
          (PAGE_BIT=1⇒CPMA←PAGE_NUMBER□PAGE_ADDRESS);
          (PAGE_BIT=0⇒CPMA←0□PAGE_ADDRESS)
```

```
        );

DEFER_ADDRESS := (
        INDIRECT_BIT = 1 ⇒
                (10↓8 ≤ CPMA ≤ 17↓8 ⇒ M[CPMA] ← M[CPMA] + 1);
                next CPMA ← M[CPMA]
        );
```

Redefinition of the memory form. – After the above definition, the design can proceed in several directions, for instance, we can define the register operations in terms of bits, we can define the interpretation of the instruction register, or we can define the memory operations in more detail. We choose the latter, at least because it will produce a more homogeneous description (i.e., the operations will be in terms of registers).

Defining the memory as a vector of registers requires two parameters, the number of registers (words) and the length of each register. The memory in the following definition requires two auxiliary registers to perform the read and write operations. These registers are not exported out of the form, i.e., they are local to the memory module.

```
form memory(integer size,wlength) =
     {declare m = register(wlength) vector(size);
          mar = register(log2(size));
          mbr = register(wlength);
     access m[register x] = m[x.value];
     function read(register addr) =
          begin mar ← addr; mbr ← m[mar]; return mbr; end;
     function write(register addr, val) =
          begin mar ← addr; mbr ← val; m[mar] ← mbr; end;
     export read, write }
```

The access declaration indicates that the value of the register is used as the index in the memory vector. The effect of the redefinition of the memory is illustrated in the following description, in which the read and write operations on the memory have been replaced by the corresponding sequences given in the form. The description of the machine itself has not changed, only the definition of one of its components. This allows us to redefine the memory at any point in time without having to change the description.

```
declare memory M[0:4095]<0:11>;
declare register AC<0:11>,
     IR<0:11>,
     PC<0:11>,
     L<>,
     LAC<0:12>:=L□AC,
     DATA_SWITCHES<0:11>,
     STOP_SWITCH<>,
     CMPA<0:11>,
     OP_CODE:=IR<0:2>,
     PAGE_BIT:=IR<4>,
     INDIRECT_BIT:=IR<3>;

INTERPRETER:=(IFETCH;next DFETCH;next EXECUTE;next INTERPRETER);

IFETCH:=(mar←PC;next mbr←M[mar];next IR←mbr;PC←PC+1);

DFETCH:=(COMPUTE_ADDRESS;next DEFER_ADDRESS);

EXECUTE:=(
     (OP_CODE='AND'⇒mar←CPMA;next mbr←M[mar];next AC←AC∧mbr);
     (OP_CODE='TAD'⇒mar←CPMA;next mbr←M[mar];next LAC←LAC+mbr);
     (OP_CODE='ISZ'⇒mar←CPMA;next mbr←M[mar];next mbr←mbr+1;next
          M[mar]←mbr;next (mbr<0⇒PC←PC+1) );
     (OP_CODE='DCA'⇒mar←CPMA;next mbr←PC;next M[mar]←mbr; AC←0);
     (OP_CODE='JMS'⇒mar←CPMA;next mbr←PC;next
          M[mar]←mbr; PC←CPMA+1);
     (OP_CODE='JMP'⇒PC←CPMA);
     (OP_CODE='IO'⇒IO_EXECUTE);
     (OP_CODE='OPR'⇒OPR_EXECUTE)
     );
```

```
COMPUTE_ADDRESS := (
        (PAGE_BIT = 1 ⇒ CPMA ← PAGE_NUMBER □ PAGE_ADDRESS);
        (PAGE_BIT = 0 ⇒ CPMA ← 0 □ PAGE_ADDRESS)
        );

DEFER_ADDRESS := (
        INDIRECT_BIT = 1 ⇒
                (10↓8 ≤ CPMA ≤ 17↓8 ⇒ mar ← CPMA ; next mbr ← M[mar] ; next
                        mbr ← mar + 1 ; next M[mar] ← mbr) ; next
                mar ← CPMA ; next mbr ← M[mar] ; next CPMA ← mbr
        );
```

Redefinition of the register form. – So far we have been dealing with registers as if they were integers. This is simply an abstraction. Hardware registers are built as array of bits and therefore the operations must be ultimately defined in terms of logic networks operating on individual bits. The form network is not defined. Informally, it represents a set of wires (memoryless components) used to carry information back and forth between other components. The following definition of a register indicates how the operations could be performed:

```
form register(integer size) =
    {declare r = bit vector(size);
    access r<integer x> =
        begin declare n = network(1); n[1] ← r[x]; n;  end;
    access r<integerpair x> =
        begin     declare n = network(x.ub.value - x.lb.value + 1);
            forall i in n do n[i] ← r[i + x.lb]; n
        end;
```

```
infix +(register a,b) =
     begin     declare x = network(a.size+1); carry = network(a.size+1);
          carry[carry.size]←0;
          decr i from a.size to 1 do
               begin
                    x[i+1]←a[i]⊕b[i]⊕carry[i+1];
                    carry[i]←a[i]∧b[i]∨a[i]∧carry[i+1]∨b[i]∧carry[i+1];
               end;
          x[1]←carry[1];
          return x;
     end;
infix −(register a,b) = ....
infix +(register a; network b) =
     begin declare x = register(b.size); x←b; return a+x; end;
infix +(network a; register b) = ....
infix +(register a; integer b) =
     begin declare x = register(a.size); x←b; return a+x; end;
function read = return r;
function write(integer val) =
          decr i from r.size to 1 do begin r[i]←val mod 2; val←val ÷ 2; end;
infix write(register b) = forall i in b do r[i]←b[i];
infix write(network b) = forall i in b do r[i]←b[i];
export  +,−,*,÷,read,write }
```

With the last example the power of the form mechanism is more apparent. We can define and redefine data types and operations without disturbing the rest of the description. The example also shows a possible way of implementing the adder. If the description is taking literally, it implies that every register is in fact a functional unit, capable of performing any arithmetic operation. For a first approximation this may be an acceptable definition. A better definition would declare a single functional unit and all register operations could then be defined using this unit. It is clear also that we can declare other types of registers, for instance, counters that would look like any other register but with the property that some simple operations (e.g., add 1, subtract 1, set to 0, etc) would be performed directly in the register.

Signals and Control Expressions.– Let us assume that we are satisfied with our previous description (it is by no means complete, but for the sake of brevity let us accept it). The sequencing of operations as expressed in the description does not indicate how the control passes through the machine description, i.e., the semantics of "next" and ";" is specified only to the point of knowing that certain actions are performed concurrently or that some actions must be completed before others can start.

We can formalize the sequencing of the operation by using control expressions, based on an underlying finite state machine, of the following type:

pre–condition : action | post–condition

The pre–condition represents the condition that must be met before the action can be executed. The action is initiated as soon as the pre–condition is satisfied. The post–condition indicates the conditions that exist upon completion of the action. The pre–condition is expressed as a conjunction of signals and boolean expressions. The evaluation of the pre–condition must be an indivisible, timeless action. The post–condition is expressed in terms of the signal operator, Δ. The Δ operator generates a signal that can be used by the pre–conditions. The signals are assumed to be unit pulses, therefore they exist only for a brief time, enough to evaluate the pre–conditions. The latching operator & can be used to store a signal for later use in a pre–condition. Latched signals will obviously exist for longer periods of time but they will dissapear as soon as they are used i.e., as soon as the pre–condition that contains the latched signal is met. There is a memory device associated with each

instance of the & operator.

Examples:

s1: . . . . | $\mathcal{D}$(s2)

s1 ∧ a=1: . . . | $\mathcal{D}$(s2,s3)

s1 ∧ s2: . . . .

&(s1) ∧ &(s2): . . . .

Given the following BNF description of ISP, we can algorithmically transform the ISP description into a set of control expressions, according to the rules given later on:

```
<process> ::= <label> := ( <s-action> )
<s-action> ::= <p-action> | <p-action> next <s-action>
<p-action> ::= <c-action> ; <p-action>
<c-action> ::= <action> | ( <exp> ⇒ <s-action> ) |
          (decode <exp> ⇒ <action-list> )
<action-list> ::= <action> | <action-list> ; <action>
<action> ::= <r-transfer> | <label> | ( <s-action> )
```

ISP to Control Expressions Translation Rules.-

description	pre-condition	action	post-condition
1) label := (s-action)	label	: s-action	| $\mathcal{D}$(label_done)
2) Si : α ; β | $\mathcal{D}$(Sj)	Si	:α	| $\mathcal{D}$(Si')
	Si	:β	| $\mathcal{D}$(Si")
	&(Si') ∧ &(Si")	:	| $\mathcal{D}$(Sj)
3) Si : α next β | $\mathcal{D}$(Sj)	Si	: α	| $\mathcal{D}$(Si')
	Si'	: β	| $\mathcal{D}$(Sj)
4) Si : $\alpha \Rightarrow \beta$ | $\mathcal{D}$(Sj)	Si ∧ α	: β	| $\mathcal{D}$(Sj)

	Si ∧ ¬∝	:	| ⊅(Sj)
5) Si : decode ∝⇒β0, ..., βn	| ⊅(Sj)		
	Si ∧ ∝ = 0	: β0	| ⊅(Sj)
	....		
	Si ∧ ∝ = n	: βn	| ⊅(Sj)
6) Si : (∝) | ⊅(Sj)	Si	: ∝	| ⊅(Sj)
7) Si : label | ⊅(Sj]	Si	:	| ⊅(label)
	Si ∧ label_done	:	| ⊅(Sj)

As an example, we wil apply the above rules to part of the PDP8 description, specifically, the ISZ instruction.

1) Applying rule 0 to the EXECUTE process:

EXECUTE	:	| ⊅(EXECUTE_DONE)

2) Applying rule 5 to separate the individual instructions:

.....		
EXECUTE ∧ OP_CODE = 'ISZ'	:	| ⊅(EXECUTE_DONE)
.....		

3) Applying rule 3 several times to the s-action describing the instruction:

EXECUTE ∧ OP_CODE = 'ISZ'	: mar ← CPMA	| ⊅(S1)
S1	: mbr ← M[mar]	| ⊅(S2)
S2	: mbr ← mbr + 1	| ⊅(S3)
S3	: M[mar] ← mbr	| ⊅(S4)
S4	: (mbr < 0 ⇒ PC ← PC + 1)	| ⊅(EXECUTE_DONE)

4) Applying rule 4 to the last component:

EXECUTE ∧ OP_CODE = 'ISZ'	: mar ← CPMA	| ⊅(S1)
S1	: mbr ← M[mar]	| ⊅(S2)
S2	: mbr ← mbr + 1	| ⊅(S3)
S3	: M[mar] ← mbr	| ⊅(S4)
S4 ∧ mbr < 0	: PC ← PC + 1	| ⊅(EXECUTE_DONE)
S4 ∧ ¬(MBR < 0)	:	| ⊅(EXECUTE_DONE)

The complexity of the actions in the control expressions can be arbitrary. This allows us to expand the description in selected parts of the machine. If we want to be more precise about the timing of the operations we can add, to the form defining the machine component, the information necessary to indicate how the signals are produced. For instance, we can add to the write operation in some register form the statement "signal after 200" to indicate that the write operation takes 200 nanoseconds. A synchronous machine could be specified by the statement "signal on P3" where P3 is the name of a clock phase.

Since we can in fact represent all the operations in terms of register transfers with the appropriate delays, we have a convenient mechanism to indicate the timing in a machine. All we have to do is provide the appropriate signal on or signal after statements in the write function of the forms describing each component.

At this point a reshuffling of the description may be desired to simplify the control logic. It is advantageous to group the primitive operations by the pre-conditions under which they are triggered rather than by the position in an opcode sequence. This is straightforward and easy to verify the correctness of the transformation. A related problem is the reduction of the number of control expressions by renaming signals that are produced under similar circumstances and with similar effects. The importance of this "optimization" is evident since the number of different signals is related to the number of possible states of the machine, part of which must be encoded in the instruction code.

CONCLUSIONS

We attempt to ease the task of programming by relativizing the production of software to the machines in which it will execute. The vehicle is a symbolic description of the hardware machine. This new area of application for computer descriptive languages requires some properties that are not found in existing notations. The use of structured programming concepts in the description of a computer system will allow the solution of a range of problems, as outlined in this paper, by manipulation of these symbolic descriptions.

REFERENCES

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

[Barbacci, 1973a] Barbacci, M.R.: "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems". Department of Computer Science, Carnegie Mellon University. PB-221591.

[Barbacci, 1973b] Barbacci, M.R. and D.P. Siewiorek: "The Automated Exploration of the Design Space for Register Transfer (RT) Systems". First Annual Symposium on Computer Architecture, University of Florida, Gainesville, December 1973.

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

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

[Chu, 1965] Chu, Y.: "An Algol-like Computer Design Language". CACM,

Vol. 8, October 1965, pp. 607-615.

[DEC, 1973] Digital Equipment Corporation: "PDP-11 Processor Handbook", Maynard Mass. 1973.

[Feldman, 1966] Feldman, J.: "A Formal Semantics for Computer Languages and its Application in a Compiler Compiler". CACM Vol. 9, January 1966, pp. 3-9.

[Miller, 1971] Miller, P.L.: "Automatic Creation of a Code Generator From a Machine Description". Massachusetts Institute of Technology, Project MAC report TR-85, May 1971.

[Newcomer, 1974] Newcomer, J.: "Machine Independant Generation of Optimal Local Code". PhD Thesis, Department of Computer Science, Carnegie-Mellon University, (in preparation).

[Wulf, 1974] Wulf, W.A.: "ALPHARD: Toward a Language to Support Structured Programs". Computer Science Department, Carnegie-Mellon University, 1974.