A PROPOSAL FOR A COMPUTER COMPILER*

Gernot Metze and Sundaram Seshu

Coordinated Science Laboratory and Department of Electrical Engineering
University of Illinois

INTRODUCTION

In recent years digital computers have been applied, with great success, to the automation of an increasing variety of tasks in the design of digital systems, from the printing of wiring tables and the drawing of logical diagrams to the optimization, according to certain criteria, of the layout of components and wiring, and even the actual computer-controlled production of subassemblies such as printed circuit boards or integrated circuits. Similarly, the design of circuits, especially those involving nonlinear elements, has been made easier by computer programs (e.g., which perform tolerance analyses). On the system level, the use of digital computers has been limited to tasks which are equally mechanical, such as programs which check for violations of fan-in, fan-out, and cascading rules.

More recently, languages have been developed which permit the simulation of a proposed system on an existing digital computer. Alternative system designs can be evaluated not only on the basis of performance statistics produced by the simulator, e.g., timing and utilization of machine components, but also by permitting the execution of programs written in the instruction language of the system being simulated.1

The system designer, however, needs a language which is powerful enough to permit the description of the macroscopic structure of the system independent of the microscopic structure of its components. While the description of the system in this language may also be used for simulation purposes, the primary objective is the description of the relationships between system components in such a way that a compiler program can supply the detailed structure of the components, guided by certain design and optimization criteria which are stated explicitly or built into the program. Thus, the program should be a true compiler, and the system design language should permit a description of the system on a higher level than the languages proposed by Proctor,2 Schlaeppi,3 and Schorr4.

The linguistic aspects of a system design language, while interesting, have not been considered here, except that the language was developed in close analogy to the programming language FORTRAN, reflecting the feeling that computer programming and computer design are related fields. In particular, concepts such as modularity (subroutine structure), interfaces (subroutine argument linkages), parallel operations (multiprogramming), flow diagrams, etc., pervade both philosophies.

Just as FORTRAN translates arithmetic statements written in near-human language into a computer program, the computer compiler will translate a system description, given essentially in the near-human language of the programmer's manual, into a description of the hardware, e.g., ANDs, ORs,
NOTs, and FLIPFLOPs, and their interconnections.

GENERAL DESCRIPTION OF THE COMPILER

In comparison to manual approaches, the design of a digital system by compiler methods can be expected to be much faster and much cheaper, making it possible to examine (either externally or within the compiler itself) many alternative designs and select the one that is best according to some criterion. In particular, one could automatically examine the design for such features as speed, cost, or maintainability. If the input language is sufficiently powerful, the effect of adding special features such as buffered input/output, look-ahead controls, etc., can be examined with a minimum of changes to the specifications. Perhaps most important is the prospect that one good system designer can design the entire system, leading to a more uniform and more balanced result.

It is convenient to break the computer compiler program into two parts: a hardware-independent system compiler which reads the input language and produces an intermediate language output, similar to the assembly language output of most compilers, and a hardware-dependent logic compiler which reads the intermediate language and produces a detailed machine description in terms of the basic building blocks specified.

The system compiler incorporates several standard assembler features, such as "macro," "repeat," and "library," as well as the facility of interspersing intermediate language statements if desired. The intermediate language output is a microinstruction string for each subprogram (subcontrol), optimized according to a specified measure. Although the system compiler is otherwise hardware-independent, this measure may involve hardware cost. The microinstruction string output of the system compiler includes a specification of the time-hierarchy and is thus equivalent to a flow chart.

The intermediate language may also be used to drive a simulator program which permits experimental programming in the instruction language of the proposed system.

The specification and the development of the logic compiler is fairly straightforward conceptually. The Boolean minimizations required introduce bookkeeping problems but no other difficulties. The logic compiler is not discussed any further in this preliminary report.

The concept of a library subroutine enters the discussion of a computer compiler in two distinct ways. The conventional notion is similar to that of a subcontrol (e.g. arithmetic control, I/O control), but in addition open subroutines ("built-in functions" in FORTRAN terminology) may be used, such as algorithms for arithmetic operations. However, one could now have several algorithms which are equivalent in their final answers, say for division in two's complement representation, and ask the compiler to choose the algorithm which fits best with the rest of the design. Thus one may want to call for any subroutine from a class of subroutines which is identified by a class name. With a sufficiently rich library one could conceive of "dime a dozen" designs that one could choose from.

The proposed compiler is also a good research tool. Since designs can be produced simply, one could produce examples rapidly to study new design ideas. Finally, the concepts generated here might well suggest procedures for the synthesis of non-computer systems thus providing a formal basis for "systems engineering."

THE INPUT LANGUAGE

The description of a digital system involves two aspects: the global description, and the subcontrol (subprogram) description. The subdivision of the system into subcontrols is similar to the subdivision of a program into subprograms, and must be done by the system designer. (Thus we implicitly seek modular designs.) However, in contrast to conventional programming, subcontrols may operate in parallel, i.e., simultaneously, thus giving rise to the need of a global description of the system. Counterparts to these concepts will become necessary when multiprogramming compilers are written.

Global Description

The global description carries the special identifier

MACHINE xxxx

followed by the following types of global headers:

1. Definitions of global constants by the operation SYN (see register declarations below) which define word length, memory size, etc.

2. Declaration of subcontrols which may operate in parallel.

3. Information necessary for optimization, such as cost, time and other measures.
There are no program statements in the global description.

Subcontrol Description

Each subcontrol description has an identifier and the necessary header statements followed by the instruction statements, i.e., the program, which describe the subcontrol:

1. The identifier is a statement of the type

   SUBCONTROL ARITH

where ARITH is a name chosen by the designer. (The subroutine linkage mechanism is further discussed under Subroutine CALLs below.)

2. Register Declarations are analogous to DIMENSION and COMMON statements in FORTRAN except that we follow the machine language convention and demand that even single bit registers be declared. There are five statement types in this category:

   a) REGISTER A(L) defines a register A of L bits where L is an integer or a previously defined symbol. Individual bits in the register are referred to by subscripts which normally range from 0 to (L - 1). Other ranges of consecutive subscripts must be specified explicitly, as for example in

      REGISTER A(-1,...,L - 2).

   b) SYN (F,N) assigns the value N, which must be a positive integer, to the symbol F, which may then be used in register definitions and subscripts.

   c) CONNECT (EAQ(-1,...,38)) = ES.A (0,...,19). Q(1,...,19) permits the concatenation of registers. The registers on the right must have been previously defined but need not be full registers. In the example above, Q(0) is not part of the extended AQ register.

   d) EQUIV (FNCTN(0,...,9) = IR(3,...,13)) labels (a part of) a register by another name and is thus the inverse of CONNECT.

   e) INTERFACE (ARITH) M,A,Q defines registers M, A, Q as interface registers in common between the current control and the subcontrol ARITH. The INTERFACE statement is similar to the FORTRAN COMMON statement but differs from it in two respects. First, several subcontrols in a machine may be operating simultaneously, which is not the case in present programs. Since the compiler would normally try to use existing registers for temporary storage, it must be aware of the interface registers which may be used by parallel controls. Secondly, for the convenience of the logic compiler as well as for readability, the alternate control with which the register is shared should be identified. Interface registers must be dimensioned by REGISTER, CONNECT, or EQUIV statements, and must be referred to by the same names, in each of the subcontrols which share them. However, the order in which they are listed in the INTERFACE statement is not important.

3. Instruction Decoding. The assignment of bit configurations for the various instructions is a task that is best left to the logic compiler. We therefore allow the design engineer to use mnemonics for instructions. There are two types of instructions involved. First we have the instructions that are to be decoded and obeyed by the current subcontrol. Second there are instructions to be given to other subcontrols (for example, main control may request a memory subcontrol to read or write a word). In the first case, we need to decode and jump to the appropriate control sequence. In the second case, we need only set up a configuration of bits in an appropriate register. In both cases, the function is undefined. We must, however, specify (to the logic compiler) the bits that are to be used to define the function.

   a) The format of the decode and jump statement is

      DECODE (IR(0,...,9))

      HLT, LLS, LRS, JMP, JAN,...

where IR(0,...,9) is (part of) a previously defined register and HLT, LLS, etc., are mnemonics which must appear as location field symbols in the main program. The DECODE statement is itself part of the main program since it serves as a multi-way branch, analogous to a computed GO TO.

   b) The format of a translation (or decoder) statement is

      UNDF (IR(0,...,9)) RM, WM, RMW

Here IR is a previously defined register and RM, WM and RMW are instructions to be passed on to other subcontrols. The system compiler generates a decoder for each such UNDF statement. Each decoder is defined in detail by the logic compiler. UNDF is a header statement.

The mnemonics on the right of the parentheses in both statements must be single-valued Boolean functions of the bits that are enclosed within the parentheses. For example, consider the execution of the instruction REPLACE ADD MEMORY, which replaces the contents of the memory cell by the sum of the previous contents and the contents of the
accumulator. We need to set up first a READ MEMORY (RM) instruction and then a WRITE MEMORY (WM) instruction in the instruction register of the memory subcontrol. If FN is the function part of the main instruction register, we cannot write

UNDF (FN), RM, WM

for RM and WM are not single-valued functions of FN alone. Some control flip-flop is also involved and must therefore be defined as

REGISTER CN (1)
UNDF (FN, CN) RM, WM

The symbols that are used on the right must appear exactly once in the DECODE statement of another subcontrol to permit correlation by the logic compiler.

4. Program Statements, which may contain a label, include the following types:

a) Register Transfers. The gating of information from a register A to a register B is specified by

\[ \text{B} = \text{A}. \]

The symbol on the right must either be a register or an undefined function. Partial register transfers are indicated by subscripting. Gating is assumed to be parallel.

b) Branch Statements.

i) DECODE, the counterpart to a computed GO TO, has been discussed in the preceding section.

ii) An unconditional branch is indicated by simply writing the symbol (without the words GO TO).

iii) A conditional branch is indicated by

\[ \text{IF (A(0) = 1) JMP} \]

where the true exit is the statement labeled JMP, the false exit the next statement. The condition must be based on a single bit being 0 or 1.

iv) The WAIT statement is similar to the IF statement, except that the true exit is the next statement, and the false exit is the WAIT statement itself, e.g.,

\[ \text{WAIT (RQ = 1)} \]

permits the subcontrol to go on to the next statement only after RQ has been set to 1.

c) SET and CLEAR permit individual bits, or entire registers, to be set to 1, or cleared to 0. Subscripts are allowed.

d) SUBROUTINE INSERTIONS are accomplished by writing the name of the subroutine, with the argument list in parentheses. Both library subroutines and programmer-defined subroutines are treated as macros. Subroutines may be called by their class name if the choice of the particular subroutine is to be left to the compiler.

e) Subcontrols call other subcontrols through the statement CALL. Since subcontrols may be parallel or sequential (see the following section), and one would like to be free to define them either way by means of global headers, we provide three formats for the CALL statement:

\[ \text{CALLS SUB(RQ)} \]
\[ \text{CALLP SUB(RQ)} \]
\[ \text{CALL SUB(RQ)} \]

By convention the argument in parentheses (RQ) is the name of the request flag. The terminals S and P designate the CALL as sequential or parallel and override the global definition. In the simple CALL, the global definition prevails. In each case a string of statements which load interface registers follows the CALL statement, terminated by an ENDC. Consider the following example of a main control to core control CALL (Store Accumulator instruction):

\[ \text{CALL CORE (MCRQ)} \]
\[ \text{MCAR = ADDR} \]
\[ \text{M = A} \]
\[ \text{MCIR = WM} \]
\[ \text{ENDC} \]

If the memory control is defined as sequential in the global headers the compiler produces the microinstruction string

\[ \text{GATE MCAR = ADDR} \]
\[ \text{GATE M = A} \]
\[ \text{DECODER 2 = WM} \]
\[ \text{CONNECT DECODER 2 to MCIR} \]
\[ \text{SET MCRQ} \]
\[ \text{WAIT (MCRQ = 0)} \]

If on the other hand the global definition states that the memory control operates in parallel with main control, the following microinstruction string
results:

WAIT (MCRQ = 0)
GATE MCAR = ADDR
GATE M = A
DECODER 2 = WM
CONNECT DECODER 2 to MCIR
SET MCRQ

5. The Slash Notation. One of the most common operations in a computer is to read a word from memory into a register or store a word from a register into memory. Therefore we invent a special shorthand notation for this purpose. The notation \(/\text{REG}/\) refers to the memory location whose address is in register \(\text{REG}\). Thus

\[ \text{IR} = /P/ \]

states that the word whose address is in the program counter \(P\) is to be read and loaded into the instruction register \(\text{IR}\). Similarly

\[ /\text{ADDR}/ = A \]

states that the contents of \(A\) are to be stored in the memory location whose address is in register \(\text{ADDR}\).

Naturally the compiler must be given the interpretations of the two statements by means of macro definitions. This macro is given the special name MEMORY. Since the memory address and buffer registers are unique to the calling program, this memory definition must be part of the calling program. Alternatively it may also be defined in detail in the global headers as a macro with a local macro MEMORY (calling the global one) defining the interface registers.

PARALLEL AND SEQUENTIAL SUBCONTROLS

As remarked earlier, a subcontrol is similar to a subprogram. Thus one intuitively expects to use some type of LINK JUMP (or RETURN JUMP). Since a subcontrol may be called from several places (in the same or different controls) it appears intuitively necessary to store the calling address in some register. If such a procedure were followed, the subcontrol would have to interpret the contents of this register and return to the calling point. The FORTRAN analog is the ASSIGNED GO TO. This technique is aesthetically unappealing since the subcontrol has to know the various points from which it can be called—an impractical procedure for library routines. Also the notion of a parallel subcontrol has no exact analog in subroutines. An interrupt subroutine comes close but a more exact analogy is the communication between two computers. In both of these cases, the standard communication technique is the use of flags rather than LINK JUMPS.

Thus parallel subcontrols must be initiated into action by means of a flag flip-flop and must similarly indicate the completion of the action by a flag flip-flop. There appears to be no reason why these two flip-flops could not be the same. We label it the REQUEST flip-flop. By convention the CALL sets the REQUEST and the subcontrol clears it when it is through.

One would like to be able to write library subroutines (subcontrols) with the parallel/sequential consideration. The executive program (or in our case, the global headers) should decide whether the subcontrol is to be used as a parallel or a sequential subcontrol. The REQUEST convention permits one to achieve this objective.

There is one further distinction between parallel and sequential usage which must be mentioned. If a parallel subcontrol can be called from two (or more) other subcontrols then it should have two (or more) sets of interface registers and request flip-flops. If \(RQ_1, RQ_2, \ldots\) are the different request flip-flops, then a subroutine can serve a fixed hierarchy of requests by the statement

\[ \text{WAIT} (RQ_1 + RQ_2 + \cdots = 1) \]

where + denotes the Boolean OR. Similarly, a scanning procedure can be arranged by using a string of IF statements. In order to make it possible to write library subroutines independent of the number of requests, a simple extension of the INDEFINITE REPEAT feature of macro compilers may be used, with the necessary information carried in the library call.

In a sequential subcontrol multiplicity of interface registers is not necessary. It may be used without harm, of course.

We may finally note one distinction between intercomputer communication and parallel subcontrols. If two computers are tied together, either computer may request action by the other (and conflicts are somehow resolved). In our case, however, the standard hierarchical structure of programming must be observed. If subcontrol \(A\) can call on sub-
control B, subcontrol B may not call on subcontrol A. Thus the problem is simpler.

TIME AND CONTROL HIERARCHIES

We have implicitly noted that there are two notions of hierarchy among subcontrols. The different subcontrols form a partially ordered set under the relation of extended CALL. As in conventional programming we insist that this relation define a true partial ordering. Beyond this fact, however, we are not too concerned with this logical hierarchy.

A second partial ordering, which is not ab initio a partial ordering but may be converted into one, is by time of operation. If two subcontrols may operate at the same time they are at the same level in this partial ordering independent of the logical hierarchy. They are parallel subcontrols in our earlier terminology. Consider for example a main control, a buffered input/output subcontrol and a memory subcontrol. Since I/O is buffered, it may operate at the same time as main control. Since the I/O subcontrol may call on memory subcontrol, the memory subcontrol may operate at the same time as main control. Thus the logical hierarchy is that shown at the left in Fig. 1, while the time hierarchy is the one shown at the right.

![Figure 1. Hierarchy of subcontrols (Hasse diagrams): Logical (left); Time (right).](image)

(For some, “parallel” is a binary relation which we extend by transitivity. The partial ordering relation in the time diagram is “not parallel, and below in logical ordering.”)

THE LIBRARY

The library material for the compiler should contain two classes of programs: subroutines and subcontrols. A subroutine is an “open” subroutine or a macro. Algorithms for arithmetic operations, incrementors, etc., come under this category. These algorithms are divided into types and are called by type names; the detailed choice is left to the compiler. Subcontrols are complete subprograms but are processed according to macro conventions, that is, they are stored in source language with dummy dimensions and dummy register names. They may contain such macro features as “indefinite repeat,” “If True,” “If False,” etc. An example of the library call for such a library subcontrol is

\[ \text{ARITH LIB AC5(A, Q, OV, N)} \]

where AC5 is the identifier of the library routine, ARITH is the name assigned in the machine, A, Q and OV are registers and N is the dimension (defined in global headers).

It is sometimes necessary to label the “next statement” while using the indefinite repeat directive. An example is a “scanner” which services requests in sequence. For this purpose we introduce the CONTINUE statement. An example follows:

```plaintext
SUBROUTINE CORE (RQ, M1, MAD, M2, AL, WL)
REGISTER MAR (AL), MBR (WL)
IRP (RQ, M1, MAD, CT, AL)
REGISTER MIOO, MIOI, MIAJ, MI (I)
STM ST (M1 + CI)
DELEG (MI) XE, XH
ST M1 + XE
CLEAN XE
CT
ST M1 + XJ
CLEAN XJ
.DOREP
CI CONTINUE
IRP
STM
END
```

If there are three controls which wish to use this memory control in parallel, one may use the library call

\[ \text{LIB CORE ((RQ1, M1, MA1, MI1),}
(RQ2, M2, MA2, MI2),
(RQ3, M3, MA3, MI3), AL, WL) \]

In the subroutine, .CORERD and .COREWR are library subroutines which set up the signals for reading and writing core memory. We note that CI is a repeated argument which has not been specified in the library call. Hence it becomes a created symbol, a different symbol for each repetition. On the other hand, STM is not an argument. Hence this symbol is assigned to the first occurrence. The repetition IRP uses simultaneous substitutions for all arguments. (This is the simple extension referred to earlier.) The CONTINUE statement is not
translated; its label is assigned to the “next” micro-instruction.

It is easily verified that the “IF, CONTINUE” arrangement in the subroutine is in fact a scanner.

THE MICROLANGUAGE (OUTPUT LANGUAGE OF THE SYSTEM COMPILER)

The output of the system compiler is a preamble followed by a string of microinstructions. The preamble contains the information necessary for the logic compiler. Wherever possible, one would like to perform microoperations in parallel. For this purpose the system compiler will associate an ordered-pair level index with each microinstruction and specify in the WAIT field the ordered pair indices of the microsteps which must be previously completed. Thus the output becomes a description of the flow diagram.

The microlanguage is permissible in the source program as well (without the ordered-pair indices, of course). In fact, arithmetic algorithms have to be written in microlanguage. In the source program a switch to the microlanguage is initiated by

— MICRO

and terminated by

— COMPILE.

All arithmetic operations in the microlanguage are Boolean. The conventions are

\[
\begin{align*}
A + B & \quad A \text{ OR } B \\
-\text{A} & \quad \text{NOT A (outer parentheses essential)} \\
A \times B & \quad A \text{ AND } B \\
A \oplus B & \quad A \text{ EXCLUSIVE OR } B
\end{align*}
\]

Other operations can be added later. Subscripts are allowed, and the RANGE of symbolic subscripts may be specified.

An equality sign denotes a definition. If the variable on the left is a flip-flop, the quantity on the right decides whether the flip-flop is set (1) or cleared (0). Otherwise the equation is taken as a signal definition (decoder output for example).

Other microoperations (all self-explanatory) are:

1. GATE RA = RB
2. IF (BIT = 1 (or 0)) LABEL
3. OFF GATE
4. STOP

SAMPLE DESIGN OF A SMALL DIGITAL SYSTEM

In order to demonstrate the versatility and power of the input language, we present here the system design of a small digital computer with a sequential arithmetic subcontrol and a parallel input/output subcontrol which handles one-word transfers to and from memory.

The card format is essentially that of FORTRAN. A detailed discussion of the example will be found in the following section. The register layout and data paths are shown in Fig. 2.

![Figure 2. System layout and data paths in the sample computer.](From the collection of the Computer History Museum (www.computerhistory.org))
DISCUSSION OF THE EXAMPLE

The purpose of the global header subprogram with the identifier “MACHINE” is to provide maximum flexibility in the system design. One can change word length or memory size simply by changing one synonym. One can change from buffered I/O to sequential I/O by removing one statement. One can change from one type of memory to another by changing macros MEMRD and MEMWR. We note incidentally the convention ENDM as macro termination and the use of macros within macros.

The statements IFF and IFT mean “IF FALSE” and “IF TRUE” and cause conditional compilation of the next statement (MAP convention). All subroutines whose names start with a period are assumed to be library subroutines (or classes of them). The subroutines used have the following interpretations:

.INCR (SOURCE, DESTINATION, LENGTH): DESTINATION = SOURCE + 1

.DECR (SOURCE, DESTINATION, LENGTH): DESTINATION = SOURCE - 1

.ADD2 (OPERAND 1, OPERAND 2, DESTINATION, OVERFLOW, WORD LENGTH): 2’s complement addition.

.SUBL2, .MUFL2, .NDVF2 have the same arguments as .ADD2 and refer to 2’s complement subtraction, multiplication and non-restoring division.

.IOBFRRD AND .IOBFWR are subroutines to activate external I/O devices to read into and write out of the IO buffer register, respectively. Similarly .CORERD and .COREWR generate signals to read from and write into the core memory.

Since we have adopted the IPLV execution convention for GO TO, certain statement tags are not admissible. These are: DECODE, CALL, MACRO, ENDC, ENDM, IF, IFF, IFT, IRP, END, WAIT, CLEAR, SET, REGISTER, INTERFACE, PARALLEL, etc.
Special Points to be Noted

Main Control: ACTR is a counter register for arithmetic control. RUNSW is a flip-flop controlled by console switches. When it is off (RNSW = 0), the instruction HTR stops the main control. However, the I/O and memory controls may continue to run. In SAD and LJP we could have used a read/modify/write procedure if the memory control had such capability. For example, if RMW is the appropriate core memory instruction, the appropriate string for SAD is:

```
CALL CORE (MCRQ)
M = A
MCAD = ADDR
MIR = RMW
ENDC
```

One would define such a string by a macro in the global description.

Note that register M appears in the interface between MAIN and ARITH, as well as in the interface between MAIN and CORE. This is permissible in this case since MAIN calls ARITH sequentially.

Arithmetic Control: We should note the formation of the long EAQ register by the use of CONNECT. The sign bit of Q is excluded in this long register and the sign bit of A is extended by one bit (ES).

I/O Control: We note the use of CALLS when subcontrol CORE is called. In the global definition, I/O and CORE have been defined as parallel as they could operate at the same time. However, when CORE is called by I/O, we have to wait for the CORE control to finish. In reading from an I/O device, we have to signal MAIN that the word has been written into memory; in writing to an I/O device, we need the word from memory to write. Thus in both cases a WAIT (ICRQ = 0) is necessary after the initiation of a memory request. If a simple CALL were used, the ENDC would have to be followed by such a wait, whereupon the first WAIT (ICRQ = 0) produced by the compiler becomes redundant. (Incidentally, in such a case the .IOBFRRD must be inside the CALL string.) We also note that the I/O/CORE interface register IOBFRI is being used also as I/O/EXTERNAL DEVICE interface. While the compiler itself will never use an interface register between parallel controls for any other purpose, the programmer may choose to do so. The compiler will bring this fact to the attention of the programmer but will not label it as an error.

Core Control: We note that the initial wait string assigns a higher priority to IO than to main control. If a scanner type of arrangement is desired we should replace the first two statements by

```
START IF (ICRQ = 1) IOD
STR1 IF (MCRQ = 0) START
```

and return to STRT1 instead of START after RCI and WCI. Clearly one could, by additional IF statements, permit more complicated scanning procedures (such as a 2:1 priority for IO over MAIN). Also we could have saved some writing through the use of IRP.

Library Subroutine Usage: An example of a library subroutine is shown under .RIPLADD. The subroutine belongs to the class .ADD2 and is called by that class name in the subcontrol ARITH. The class definition is contained in a table within the library directory. We note a new statement ASSERT. This statement is an assertion known to the programmer but difficult to detect by program. The assertion is passed on to the logic compiler for its use.

CONCLUSION

The purpose of this paper was to present a proposal for a computer compiler system of programs. As of the time of writing this paper, the system of programs is not available and hence no experimental data can be provided. The example given establishes the following facts:

1. The input language is simple and versatile.
2. The input language is complete. That is, one can describe any existing computer unambiguously in this language.
3. The language is translatable. That is, there appear to be no conceptual reasons why the input cannot be algorithmically translated to produce optimized hardware designs.

The basic concept proposed here is not entirely new. Similar work has been reported earlier by Proctor and Schorr. The present paper differs from the earlier proposals in several respects. First the philosophy is different. The basis of each decision has been user convenience rather than linguistic structure. In fact we have chosen to disregard the linguistic aspects. By the same token, the user is not required to specify any more information than he absolutely has to. For example, no registers that are not directly referred to in the input need be
defined. MACRO, SUBROUTINE, LIBRARY and conditional compilations are new features (an elementary MACRO was used by Schlaeppi). The use of global headers and the notion of parallel/sequential CALLs to subcontrols introduces a flexibility that was not previously available.

The compiler (when completed) will optimize more extensively than is humanly possible, maintaining the modularity specified in the input. Thus the system compiler will attempt to merge micro-instruction strings both between instructions (of the object machine) and within an instruction, inserting conditional branch statements where necessary. The hardware compiler will attempt as much Boolean minimization as practical. The assignment of bit configurations to instructions and the combination of different decoders (within one subcontrol) are places where substantial gains are expected.

One last distinction which is conceptually trivial, but to us important, is that our designs will be asynchronous. It is our claim that asynchronous computers are faster and more reliable in addition to being more maintainable. Design difficulty, which has in the past been the main disadvantage, is eliminated by the computer compiler.

REFERENCES
