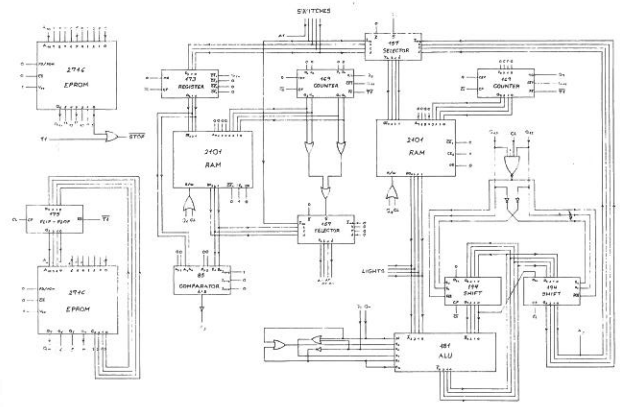
PB-161

T. Hagerup: HSIM - a hardware simulator

TRYK: DAIMI/RECAU

ISSN 0105-8517

HSIM - a hardware simulator



Torben Hagerup

DAIMI PB-161
May 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

HSIM -

A HARDWARE SIMULATOR.

In this report a system capable of simulating hardware systems is presented. It is described from a user as well as from an implementation point of view.

The intended reader has a broad knowledge of computer science and an interest in hardware.

Contents.

Introduction.

Simulation may play an important role in the development of a piece of hardware. A hardware simulator is a computer program or system to carry out this task. One such system is described here.

The system can be used to simulate any of a large number of hardware circuits. Therefore some notation must exist to describe a particular circuit to the system. This is the subject of chapter 1.

Chapters 2, 3 and 4 describe the implementation, and chapter 5 reports on some practical experience gained with the system. Finally, chapter 6 attempts a comparison with other simulation systems.

Terminology.

In what follows, "hardware" invariably means digital, logic hardware. The reader is assumed to be familiar with basic hardware terminology as it is used in [1], [2] and [3]. Some knowledge of Pascal and possibly Concurrent Pascal is also desirable since many concepts are explained relative to their closest analogue in one of these languages, and because the simulation system is implemented as a collection of Pascal programs.

The remainder of this section introduces some additional terminology.

To avoid confusion with the simulation system, the hardware system it simulates is often referred to as *the object system*. Occasionally, the word "circuit" will be used in the same sense.

A hardware circuit is a physical ("real") device, but it is normally thought of in terms of a *logical model* that attempts to describe its behaviour. In particular, a continuum of possible values (voltages) is replaced by only two values called *logical values* and denoted, for instance, by HIGH and LOW.

Background.

The history of this work dates back to the spring of 1980 when I took an undergraduate course (so-called 3rd year module) called "Digital Logic Components", at that time given by Becky Clarke assisted by Kurt Andersen. The course has since then been offered regularly to students, most recently by Peter Møller-Nielsen; I shall refer to it henceforth as the "hardware" course. In it, basic hardware design methods are presented, and the participants use "breadboard" kits to build a number of circuits including a

rudimentary micro-computer.

At the beginning of the following semester, I had two incentives to start work in the direction described here. First, although I had no longer a kit at my disposal, I wanted to experiment further with the constrution of hardware circuitry, and on a somewhat larger scale.

Second, it seemed rather archaic to carry out a logical task like the design and testing of a hardware circuit, using a physical system at the mercy of faulty connections, defective IC's etc., and I thought it an exciting challenge to design a computer program that would assist in the process.

I therefore set out to build a system capable of simulating as faithfully as possible a breadboard circuit. In April 1981, when a preliminary version had been implemented, Peter Møller-Nielsen showed interest in presenting the system to the participants of a future hardware course as an experimental design tool.

During the summer, I worked to complete the system, give it a friendlier and more robust user interface, and write a user's manual for it. It was used successfully in the fall 1981 hardware course, giving rise to only minor problems that were corrected "on the spot".

Design objectives.

What I wanted to do was to simulate breadboard circuits, and this emphasis was what distinguished my work from other treatments of simulation. It would therefore not be reasonable to come up with a general system capable of simulating a range of widely differing systems, such as a Simula 67 system. On the other hand, it would be foolish to reject any generality that might be avaliable for free. I therefore decided to make the system as general as possible without sacrificing anything substantial in the process of generalization from the most special-purpose system conceivable.

Further, the system should be simple and easy to use, "clean" without too many special "features" and conventions, and easy to implement. These requirements are especially easy to justify here, as I thought of the system as a pilot project in a relatively new field. Until the system was demonstrated to be of any practical use at all, it would be a waste of time to muddle it up with "programmer's short cuts" etc.

Lastly, during the design phase, it was generally feared that the program might turn out to be hideously slow. For this reason, efficiency of the simulation was also a major concern.

1. THE PROGRAMMING LANGUAGE HL.

HL is a notation that allows a certain class of hardware systems to be precisely specified. The simulation system expects an HL program as its input.

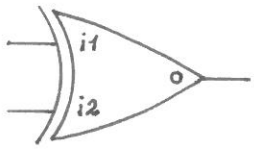
In this chapter I describe and discuss the syntax and semantics of HL. I concentrate on what I see as the most important features of the language.

A concise and formalized definition is outside the scope of this report; I allow myself to rely heavily on the reader's intuition as stimulated by an example program. Further examples and a BNF grammar for HL may be found in appendices A and B.

An example program.

The following HL program defines a simple 4-bit binary counter and uses it in a trivial test circuit. The program text is complemented by a number of diagrams.

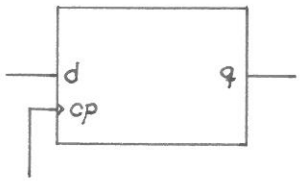
```
PRIMITIVE xor;
  (* exclusive OR *)

  IN i1,i2;
  OUT o;

BEGIN
  o:=i1<>i2;
END;
```



xor

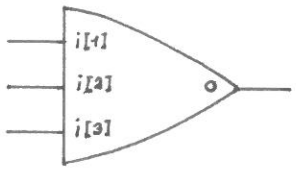
```
(*********************************)


TYPE counter;
  (* 4-bit binary counter *)

  IN cp; (* clock input *)
  OUT q[0..3];


  PRIMITIVE flipflop;
    (* D flip-flop *)

    IN d,cp;
    OUT q;
    VAR s; (* internal state *)
```

```
BEGIN
  IF cp
  THEN q:=s
  ELSE s:=d;
END;

(*****************************)
```



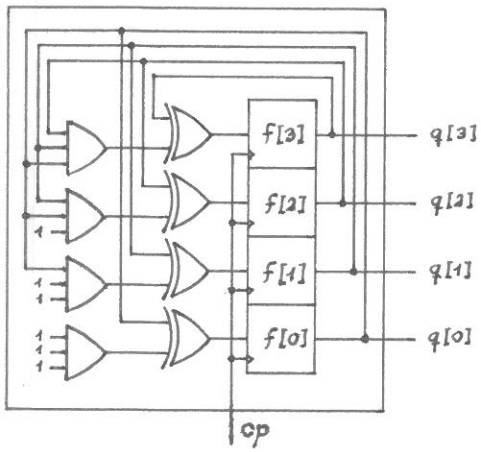
flipflop

```
PRIMITIVE and3;
  (* 3-input AND gate *)

  IN i[1..3];
  OUT o;
  VAR k,n;

BEGIN
  n:=i[1];
  FOR k:=2 TO 3
  DO n:=n*i[k];
  o:=n;
END;

(*****************************)
```



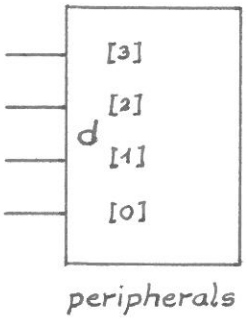
and3

```
COMP f[0..3]:flipflop;
     a3[0..3]:and3;
     x[0..3]:xor;

VAR k,l;


BEGIN (* counter *)
  FOR k:=0 TO 3
  DO BEGIN
       cp->f[k].cp;
       FOR l:=1 TO k
       DO f[l-1].q->a3[k].i[l];
       FOR l:=k+1 TO 3
       DO 1->a3[k].i[l];
       a3[k].o->x[k].i1;
       f[k].q->x[k].i2;
       x[k].o->f[k].d;
       f[k].q->q[k];
     END;
END;

(*********************************)
```

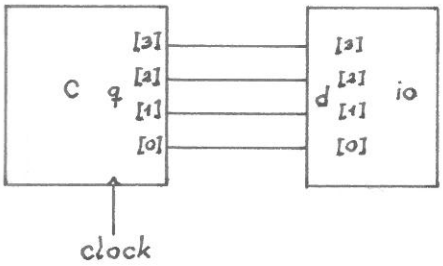


counter

```
PRIMITIVE peripherals;

  IN d[0..3];
  VAR k,dummy;

BEGIN
  FOR k:=0 TO 3
  DO BEGIN
       REACH(10,42-k);
       WRITE(d[k]:1);
     END;
  REACH(12,40);
  READ(dummy);
END;
```




```
(*********************************)


  COMP c:counter;
       io:peripherals;

  VAR k;

BEGIN (* main *)
  clock->c.cp;
  FOR k:=0 TO 3
  DO c.q[k]->io.d[k];
END;
```



1.1 Syntax.

I shall not attempt a rather pointless rigorous separation of syntax and semantics. This section deals mainly with syntactic issues and with "obvious" semantics that are mostly implied, such as those of arithmetic operators. The more interesting parts of the semantics are given in section 1.2.

1.1.1 HL program.

The hardware system described by an HL program has the following characteristics:

It consists of a finite number of named components and a finite number of wires.

Each component has a finite number of named ports divided into input ports and output ports. Each wire connects an output port to an input port.

At any given time, each port has a value that can be represented as an integer. Connected ports always have the

same value. Values of input and output ports are called inputs and outputs, respectively.

An HL program consists of a declarative part followed by a statement part. The declarative part establishes the set of components, and the statement part specifies the pattern of connecting wires.

1.1.2 Main declarative part.

The main declarative part contains declarations of variables, components, types and primitives.

Variables.

Variables are declared following the key word VAR. They correspond to Pascal integer variables.

Components.

Components are declared following the key word COMP. They correspond to the physical components mentioned above.

Each component is an instance of a particular circuit (the concept of "circuit" is similar to that of "system type" in Concurrent Pascal). In a component declaration, a circuit identifier is given following a colon, as in a Pascal variable declaration.

A circuit is either a type or a primitive. Instances of types and primitives are called type components and primitive components, respectively.

Types.

A type is a template for a component that may interact with other components through a number of ports, but is otherwise structured like the entire system.

Input and output ports are declared following the key words IN and OUT. Apart from the differences arising from the handling of ports (and the heading TYPE <type identifier> ;), syntax and semantics of a type are those of an entire program; alternatively, the program may be viewed as an anonymous type without ports.

The components and ports declared in a type T (and in no inner circuit) are called the constituent parts of T.

Primitives.

A primitive is a template for a component that is not further decomposable. Declarations in a primitive may be of variables and ports only.

The primitive body is an algorithm describing how the primitive's outputs may be computed from its inputs and possibly some internal state.

1.1.3 Main statement part.

The main statement part is an algorithm to connect components in a certain way. The central element is the connection statement.

Connection statements.

There are two forms of connection statements.

The most important form has an output port and an input port separated by the connection operator ->.

X.A -> Y.B

is an instruction to connect output A of component X to input B of component Y.

The other form I shall also call a fix statement. It is an instruction to feed a particular (integer) value n permanently to an input port. The syntax is

n -> Y.B .

After execution of this statement, I shall say that Y.B as well as all ports connected to it have been fixed at the value n.

No port may participate more than once as the right operand in a connection statement.

Other statements.

The following statement types have been adopted from Pascal:

Assignment statement
IF statement
WHILE statement
REPEAT statement
FOR statement

Targets of assignment statements and iteration indices of FOR statements must be variables; and expressions must be composed of variables, integer literals and the usual arithmetic operators of Pascal.

An integer expression is permitted in an IF, WHILE or REPEAT statement where Pascal requires a boolean expression. The expression is interpreted as FALSE exactly if it evaluates to 0.

Relational operators (=,<,...) yield the value 1 corresponding to TRUE and 0 corresponding to FALSE.

1.1.4 Type bodies.

The contents of the previous section also apply to the statement part of types. In addition,

1). The following restrictions apply to connection statements in the body of a type T:

a) The port to the right of the connection operator must be declared within T.

b) At least one of the involved ports (in a fix statement: The involved port) must belong to a component declared within T.

2). Inside a type T (textually), input ports of T are treated as output ports (as regards the legality of connection statements), and output ports as input ports. They are denoted without qualification (e.g. "cp" instead of "T.cp").

1.1.5 Primitive bodies.

Statement parts of primitives must not contain connection statements. Inside primitives, input ports may be evaluated to yield integer values, but assignment to input ports is illegal. Integer values may be assigned to output ports, but output ports cannot be evaluated.

1.1.6 Scope rules.

Let the types and primitives of an HL program define a set of nested blocks in the usual way. The smallest block containing the declaration of a given object is said to *define* that object and to be its *defining circuit*.

The scope rules may be stated as follows:

1). The scope of types, primitives, variables and components extends from their point of declaration to the end of their defining circuit, with the exception of inner blocks that are either primitives or that redeclare the identifier concerned. In the latter case, the outer declaration is valid until the point of redeclaration.

2). An unqualified port name follows the rule just given. If the component C is an instance of a circuit defining a port A, the name C.A is valid at any point in the scope of C.

This rule is the same as that which governs the scope of record fields in Pascal.

Variables and ports defined by a type are called *type*

variables and type ports. Primitive variables and primitive ports are defined analogously.

1.1.7 Arrays.

Variables, ports and components may be declared as arrays rather than as single objects. This is done by following the object name by the desired index range(s) enclosed in square brackets. Array bounds and indices must be integer-valued.

Once declared, an array cannot be manipulated as a whole. Single array elements are denoted as in Pascal.

All arrays are dynamic. Bounds are evaluated at array declaration time and may be arbitrary expressions.

1.1.8 Input-output.

Input and output of integers from/to a TTY have been implemented. Further, strings appearing literally in the source program may be output on the TTY.

Coordinates have been assigned to the terminal screen, and there is an instruction REACH to move the cursor to an arbitrary position.

1.2 Semantics.

The semantics of HL tell two things:

1). How to associate a circuit with an HL program.
2). How to simulate that circuit.

I shall describe both, but will put the emphasis on 2) above, since it is probably the least obvious. The first of the following sections deals with 1), the remaining three with 2).

1.2.1 Type expansion.

The simulation will be explained in terms of an object system called the expanded system. The system described by an HL program is converted into its corresponding expanded system by the following algorithm, similar to macro expansion:

> Repeatedly replace globally declared type components by their constituent parts and appropriate connections, until all type components have been replaced by primitive components.

1.2.2 Component execution.

In the expanded system, "component" means primitive component. Execution of a component is execution of the body of the primitive of which the component is an instance, with references to ports and variables causing manipulation of the component's ports and variables.

Call a component excited, if at least one of its inputs has changed since the component was last executed. The expanded system is simulated according to the following three principles:

1). Any excited component is eventually executed.

2). Each change in the value of an output port is immediately propagated to the set of connected (possibly indirectly: via a number of type ports) input ports.

3). Values of component variables are preserved between executions.

1.2.3 The predefined clock.

There is one predefined "system" output port with the name "clock". This port always has one of the values 0 and 1. Its value is made to change whenever there are no excited components.

1.2.4 Initialization.

Initially (when simulation starts), all components are regarded as being excited.

All ports that have not been fixed at another value, and all primitive variables initially have the value 0.

1.3 Discussion.

In this section I shall try to evaluate the characteristics of the language in a number of ways. I shall also mention some desirable features that have been left out.

1.3.1 Types.

The type concept is a powerful structuring tool. It allows complex data types to be built in a hierarchical and orderly fashion.

The range of expressible hardware systems does not in principle depend on the availability of types. But the exclusion of the type concept would require the entire object system to be described on one single hierarchical level. In practice, this is adequate only for very small systems; compare the role of procedures in other programming languages. It is feasible to design an entire computer starting from single transistors, but not by considering each transistor in turn.

The type concept is absolutely essential to the usefulness of the system.

Types describe the static structure of a system. Therefore, operations on port values in type bodies do not make any sense. Likewise, type variables are requisite in constructing a circuit but have got nothing to do with its simulation.

1.3.2 Primitives.

Primitives may be thought of as defining a dividing line: Above this line, "objects" are described in hardware terms, as being composed of other objects connected in certain ways. Below it, objects are described in whatever way seems practicable, using the power of a general programming language.

Defining a group of primitives amounts to choosing a set of basic building blocks or axioms. Clearly, there must be such a basic platform; one cannot go on forever defining objects in terms of simpler objects, especially since these objects are supposed to model a physical reality.

In HL, great flexibility is derived from the fact that primitives may be arbitrarily defined. The programmer is free to choose his personal level of interest. Although physical hardware normally can be analysed down to the level of single gates before quantum mechanics must be invoked, it is a perfectly sound approach to view entire micro-computers as the basic units of hardware.

Assignment to input ports is not allowed for fairly obvious reasons (what would be the physical interpretation ?). Evaluation of output ports has been ruled out to achieve a kind of symmetry and to prevent the surreptitious use of output ports as state variables.

Primitives describe the dynamic behaviour of components in a fixed circuit. Therefore, connection statements in primitive bodies are meaningless.

1.3.3 Design of primitives.

The rules defining when a component is executed are not very explicit. For instance, if two inputs to a component change within a single clock half-cycle, the rules do not even state whether the component is executed once or twice.

This means that primitives must be of a rugged and robust design. But it is neither impossible nor particularly difficult to build properly functioning equivalents of common SSI and MSI circuits.

This is a trivial fact for combinational circuits; the XOR-gate of the example program may serve to illustrate the point. It is less obvious how clocked circuits may be implemented. An internal state can be stored in a set of variables, since these are preserved between executions. But how is it possible to ensure that output changes are caused by the clock transition ?

The primitive "flipflop" of the example program is supposed to be a normal D-type flip-flop triggered by the HIGH-going (0 -> 1) clock transition. The reader is invited to convince himself that it will work correctly in all circumstances.

Using the same principle, or in fact this particular primitive, it is possible to construct a very wide range of clocked circuits. This is intuitively clear and confirmed by practical experience.

1.3.4 The clock.

The predefined clock output serves, much as in real hardware, as the one point of independent action that can spark off a chain reaction in connected circuits.

1.3.5 Initialization.

Experience has shown that it is necessary to initialize a circuit before the simulation proper. Otherwise, components may be in an "inconsistent" state (e.g. an inverter with input=output), and in most cases, the simulation never gets under way.

This is the reason why all components are initially regarded as being excited. Components are supposed to be designed so that, once executed, they will be in the

equivalent of a physically possible state.

Another possibility would have been to provide each primitive with an explicit initial statement (Concurrent Pascal terminology). I decided against this option because the initial statement for most common primitives seems to be either immaterial or to naturally coincide with the normal statement part (the latter being the case, for instance, for all combinational primitives). An even better choice might have been to have an optional initial statement.

A few primitives do need a special initialization algorithm to be carried out. This can be accomodated in the following not very nice way:

```
IF "first time"
THEN
  .
  .  (* initialization *)
  .
ELSE
  .
  .  (* normal operation *)
  .
```

The "first time" condition must be represented by some variable. This means that at least some variables must have a known initial value. Hence the convention that all primitive variables are initialized to the value 0.

1.3.6 Representing logical values.

The logical values are represented by integers. Although it is not logically necessary, I have always used 1 for logical HIGH or TRUE and 0 for logical LOW or FALSE.

It might be argued that this is a kind of built-in "type anarchy", in that the user is forced to mix up two concepts that have got nothing to do with each other. But clearly, however logical values are represented, they must be amenable to manipulation inside primitives, and the alternatives to the chosen path all seemed rather unpleasant:

- Providing a new data type
  ```
  signal=(HIGH,LOW)
  ```
  with only the comparison operator defined.

- Providing a new data type with a welter of more or less familiar operations.

- Insisting on explicit conversion to and from a data type having useful operations ("integer" or "boolean").

For instance for the XOR-gate:

```
o:=logic(int(i1)<>int(i2));
```

An even more important reason is that the signals carried by the wires not necessarily represent the two logical values. For convenience, a wire may be used to represent an entire data bus, or a completely different data channel. Insisting on one-bit communication would be like requiring all primitives to be of gate complexity.

This also explains why the system does not restrict the integers transmitted to two values. Even in traditional low-level design, one might want to use more than two values, for instance to express a "don't know" condition.

1.3.7 Connection statements.

A pattern of interconnections may be specified in many ways. I have chosen a highly "operational" one because I think its meaning is particularly easy to grasp. Furthermore, it facilitates the use of an HL program as a "check list" when constructing the physical circuit corresponding to the program. See [4] for a quite different approach.

There are three reasons for the insistance that the left and right operands to the connection operator must be an output and an input port, respectively:

- I think this is the "right" way to think about a connection. Connecting two input ports, for instance, does not really have much sense to it.
- It allows a kind of type checking to be performed.
- Less importantly, the restricted form of connection statement seems to have a simpler implementation.

Restrictions.

The requirement that no input port may be connected-to more than once is the counterpart of the basic hardware design rule stating that it is illegal (except in special cases) to connect two outputs.

The rules discussed below are those given in section 1.1.4.

Rule 2) is perfectly natural. This ought to be clear from the example program and its accompanying diagrams.

Fig. 1 illustrates the reason for rule 1 a).

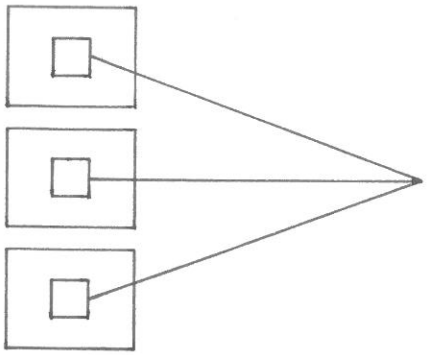


Fig. 1

If the rule is violated in the body of a type T, and two or more instances of T are subsequently declared, there will be multiple connections to the same port.

Rule 1 b) is included solely for efficiency reasons. The connection statements prohibited by this rule, but otherwise legal, do not seem particularly useful, and their implementation presented a problem.

1.3.8 Pascal-like notation.

HL has been designed so as to ressemble as closely as possible the well-known programming language Pascal. This has a number of advantages:

- The user will (presumably) have little difficulty understanding syntax and semantics.
- Design pitfalls are easier to avoid.
- Parsing and part of the implementation are standard.
- Standard terminology, for instance on syntactic structure, can be used to talk about the system.

However, using the same language constructs both in type bodies and in primitive bodies also has a disadvantage:

- The important distinction between circuit construction and simulation is blurred.

The choice of Pascal as the "host" language is less important. Pascal is generally regarded as a "good" language, and it was familiar to me.

1.3.9 Scope rules.

The basic decision was to the use the scope rules of Pascal, and to amend them wherever necessary. There is one obvious and one more subtle modification.

Port names cannot, of course, comply with Pascal scope rules, since they would be invisible outside their defining circuit and therefore useless. Adequate scope rules (and notation) are those of record fields.

Only locally defined names may occur in primitives. This has a number of advantages:

- The clarity of meaning that goes with the absence of free identifiers.
- Efficient component execution.
- An insignificantly simpler implementation.

The really important point, however, is that "foreign" names would present a serious conceptual problem. The most obvious candidates would be variables declared in surrounding blocks, but such variables exist only during (part of) a circuit construction phase (see section 1.3.1), whereas primitive bodies are not executed until simulation. It is hard to see how a sensible meaning can be attached to a type variable whose value or address is suddenly required in the course of the simulation.

Nevertheless, it is desirable in some situations to be able to use non-local objects in primitives. A really good solution might be to provide primitives with parameters. This idea is elaborated in section 1.3.14.

1.3.10 Typelessness.

In this section only, the word "type" will be used in its usual Pascal sense.

HL as presented is a "typeless" language, in that only integer variables are supported. This does not mean that I reject one of the major advances in language design, the idea of types and type checking. In fact, I only reluctantly refrained from having type declarations and proper handling of other basic types than integers. Ideally, I would have liked to be able to send values of arbitrary types through typed wires. But it would have added non-trivially to the complexity of the compiler, and it did not after all seem worth while for a project of this kind.

I do not personally regard the use of integers as booleans as particularly disturbing. But the absence of a type "character" and of character input-output is a nuisance.

1.3.11 Flow-of-control in types.

The reader might initially wonder why control structures have been introduced in type bodies. The pattern of wires inside a type is a concept that does not inherently possess any aspect of time or ordering of wires. However, the reader must concede that FOR statements are useful when handling data buses (see the example program). More complex but still regular structures are conveniently expressed using the full range of control structures.

Nevertheless, I think it is unfortunate that the concept of flow-of-control in types had to be introduced, and that an artificial ordering of wires is imposed.
But the problem seems largely a theoretical one, and to solve it would involve departing radically from the ideas of Pascal. I did not want to become sidetracked by this issue.

1.3.12 Input-output.

Input-output was designed with the sole purpose of being handy in the context of the hardware course. The direct addressing of "fields" on the terminal screen is intended to be reminiscent of the control panel found on most computers.

1.3.13 Simulating reality ?

A crucial question is, of course: Which aspects of a real hardware system can be dealt with successfully by the system ? And what kind of questions cannot even be formulated within its framework ?

Logics versus physics.

The system is intended to simulate the logical behaviour of a hardware system, and not its detailed physical behaviour. The values assigned to ports come from a discrete domain and are supposed to model logical values, not voltages.

Geometry and topology.

Atomic objects (primitive components) interact exclusively through ports and are connected by wires having no other properties than the identities of their endpoints.

Therefore, all questions regarding geometry and topology [2] are meaningless.

Delays.

The system handles delays only in a very limited way. The mentioning of a specific delay time is conspicuously absent from the sentence "Any excited component is eventually executed" (section 1.2.2). In order to use the system successfully, one must adopt the assumption

All delay times are positive but otherwise unknown.

The adhesion to this principle is also regarded as an element of good style in ordinary hardware design [1]. It amounts to not making assumptions about the relative sizes of delays.

Hold and setup times.

Hold times are implicitly assumed to be zero. The same is true of setup times, but in the (common) case of setup times leading up to the clock transition, this has no significance (see below).

Clock period.

The value of the clock is changed only when there are no more excited components. This means that the clock period is always "long enough". In the real world, a circuit may fail to work due to a too fast clock; this has no analogue in the system.

Other limitations.

Other concepts that have no place in the system include

- Fan-out limitations.
- Independent clocks.
- (not surprisingly) Noise problems.

Three-state outputs.

Three-state outputs cannot be directly incorporated, since it is illegal to connect two outputs. However, at the price of having to put in an extra "junction" component wherever three-state outputs meet, it is possible to make a reasonable substitute.

A value different from the values representing the two usual logical values would then be chosen to represent the "high Z"-state, and the "junction" component would simply check that at most one input was not "high Z", and otherwise pass on its one valid input.

"Open collectors" can be handled in much the same way.

1.3.14 Proposed additions to HL.

Circuit parameters.

It would be most useful and satisfying to have a kind of generic facility: Circuit parameters. The example illustrates the idea:

```
PRIMITIVE and(m);
  IN i[1..m];
  OUT o;
  VAR n,k;
BEGIN
  n:=i[1];
  FOR k:=2 TO m
  DO n:=n*i[k];
  o:=n;
END;
```

The primitive shown is an AND-gate with an unspecified number of inputs. A concrete AND-gate with six inputs would subsequently be declared as follows:

```
COMP a6: and(6);
```

I imagine actual parameters to be evaluated to an integer value when a component is declared, and the formal parameter to act as a constant having that value.

The inclusion of circuit parameters would involve only small changes to the implementation. The main precondition of dynamic arrays is already fulfilled.

Other additions.

Some or all of the following language elements might prove useful to an HL programmer. They have all been taken from Pascal.

Constants
Procedures and functions
WITH statement
Records
CASE statement

The WITH statement is intended for use in connection with circuits rather than records.

2. SIMULATION.

This chapter gives the internal view of what happens during the actual simulation. The relevant algorithms are described on two levels of detail.

The first level is intended for the reader who wants to glance "behind the curtain" to get a better feeling for the semantics of HL, the same way that knowledge of a run-time stack may clarify the concept of recursion. It is also a necessary preamble to the more detailed description.

The second level may be helpful to anyone who wants to modify, copy or understand in detail the presented system. My goal is to supply just enough information to enable an experienced programmer to rebuild the system easily, without cluttering the picture by not very interesting details.

2.1 A broad outline.

During simulation, the system goes through a series of simulated clock cycles. Every time the clock value changes, the components connected to the clock are executed. In general, a component is executed whenever one or more of its inputs change. This in turn may cause inputs to other components to change.

When this process stops (if ever), the simulated time is advanced to the next clock change, i.e. the clock value is made to change.

Before the first change of the clock value, the circuit is initialized by executing every component at least once, as explained in the previous chapter.

The conceptually simplest implementation might assign a processor to each component. But since this implementation uses a sequential machine, some form of time-sharing among components is necessary.

To state the algorithm in a more lucid and precise way, I shall make use of a Pascal-like mixture of formal and informal elements, the latter being explained below.

Main simulation algorithm:

```
01  "Put all components into Q";
02  Clock_voltage:=0;
03  REPEAT
04    REPEAT
05      "Take a component out of Q";
06      "Execute it";
07    UNTIL "Q is empty";
08    Clock_voltage:=1-clock_voltage;
09    "For all input ports connected to the clock"
10    DO "value of the port" := clock_voltage;
11    "For all components connected to the clock"
12    DO "put the component into Q";
13  UNTIL "stop criterium";
```

Explanatory remarks:

1). "Q" is an unbounded FIFO queue.

2). "Clock_voltage" is a variable holding the current clock value.

3). I shall not go into the details of how a component is executed. The only point of interest is the algorithm carried out when a value is assigned to an output port of the component, say the value e to the port P.

Port assignment algorithm:

```
31  IF "value of P"<>e
32  THEN
33   BEGIN
34    "Value of P":=e;
35    "For all input ports connected to P"
36    DO BEGIN
37        "Value of the port":=e;
38        IF "the component containing the port is not in Q"
39        THEN "put it into Q";
40       END;
41   END;
```

2.2 A closer look.

This section centers around the main data structures used in the simulation. They are explained as to purpose and logical structure, and a carefully worked-out example is given. The reader is advised to study the example while reading the description.

After this, I explain how the informal parts of the two algorithms of the previous section are implemented.

2.2.1 The data structures.

Three large Pascal arrays together specify an internal model of the object system. I distinguish between dynamic data that change in the course of the simulation, and static data that do not.

M contains static data describing port interconnections and dynamic data giving, among other things, port values.

D contains static dope vectors for array variables and array ports, and dynamic values of variables.

C contains the static code for each primitive. This is some representation of the primitive body.

All three arrays are indexed by positive integers I shall often call "addresses" (the address of a chunk of data being the smallest index of a variable holding part of the data).

The elements of M are of (Pascal) type

```
Record
  value,
  next,
  link,
  father,
  last:integer;
end;
```

Every port corresponds to an element of M. Every component is represented by a contiguous part of M. I shall not always distinguish between elements of M and the objects they represent.

The "next" and "link" fields are static.

"Next" fields partition the set of ports into singly linked lists I call equivalence lists. Two ports belong to the same equivalence list exactly if they are connected in the internal model. If an equivalence list contains an output port, it is the first list element.

"Link" fields serve as pointers. Some allow the address of a component to be obtained from the address of one of its ports, others provide access to the information stored in D.

The "next" and "link" fields obey the following conventions (suppose the fields in question belong to M[k]):

Next>0 => M[k] represents an input port.

Next<0 => M[k] represents an output port.

Next<>0 => ABS(next) is the address in M of the successor of M[k] in its equivalence list.

Next=0 => M[k] is the last element in its equivalence list.

Link>0 => Link is the address in M of the component containing M[k]. Link<>k.

Link<0 => k is the address in M of the component containing M[k].
(-Link) is the address in D of a block of data belonging to the component containing M[k].

The "value", "father" and "last" fields are dynamic.

The "value" field simply contains the value of the port.

"Father" and "last" fields will be discussed in section 2.2.2.

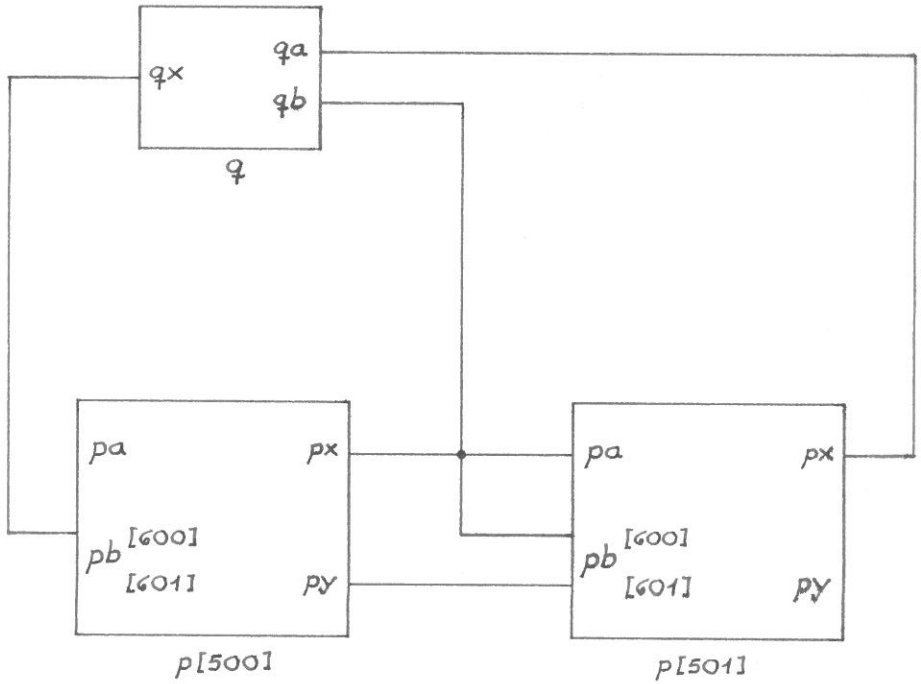
Data in D belonging to primitive components declared together (i.e. in the same <itemlist> : <primitive identifier> declaration) form a contiguous block that is logically divided into a common segment, common to all the components in question, and a private segment consisting of one local store for each component.

The common segment contains a pointer to the code in C of the relevant primitive, and dope vectors for all array variables and array ports belonging to the primitive. The exact format of dope vectors is not important here.
The local store of a component contains values of the component's local variables as well as a pointer to the common segment.

An example.

The example on the following pages shows an HL program, the object system it describes and the static parts of its corresponding internal model.
For the purpose of clarity, data in M, D and C have been offset by fictitious displacements.

```
  PRIMITIVE pp;
    IN pa,pb[600..601];
    OUT px,py;
    VAR k,n[800..802];
  BEGIN
  END;

  PRIMITIVE qq;
    IN qa,qb;
    OUT qx;
  BEGIN
  END;

  COMP p[500..501]:pp;
       q:qq;

BEGIN
  p[500].px -> p[501].pa;
  p[500].px -> p[501].pb[600];
  p[500].px -> q.qb;
  p[500].py -> p[501].pb[601];
  p[501].px -> q.qa;
  q.qx -> p[500].pb[600];
END;
```



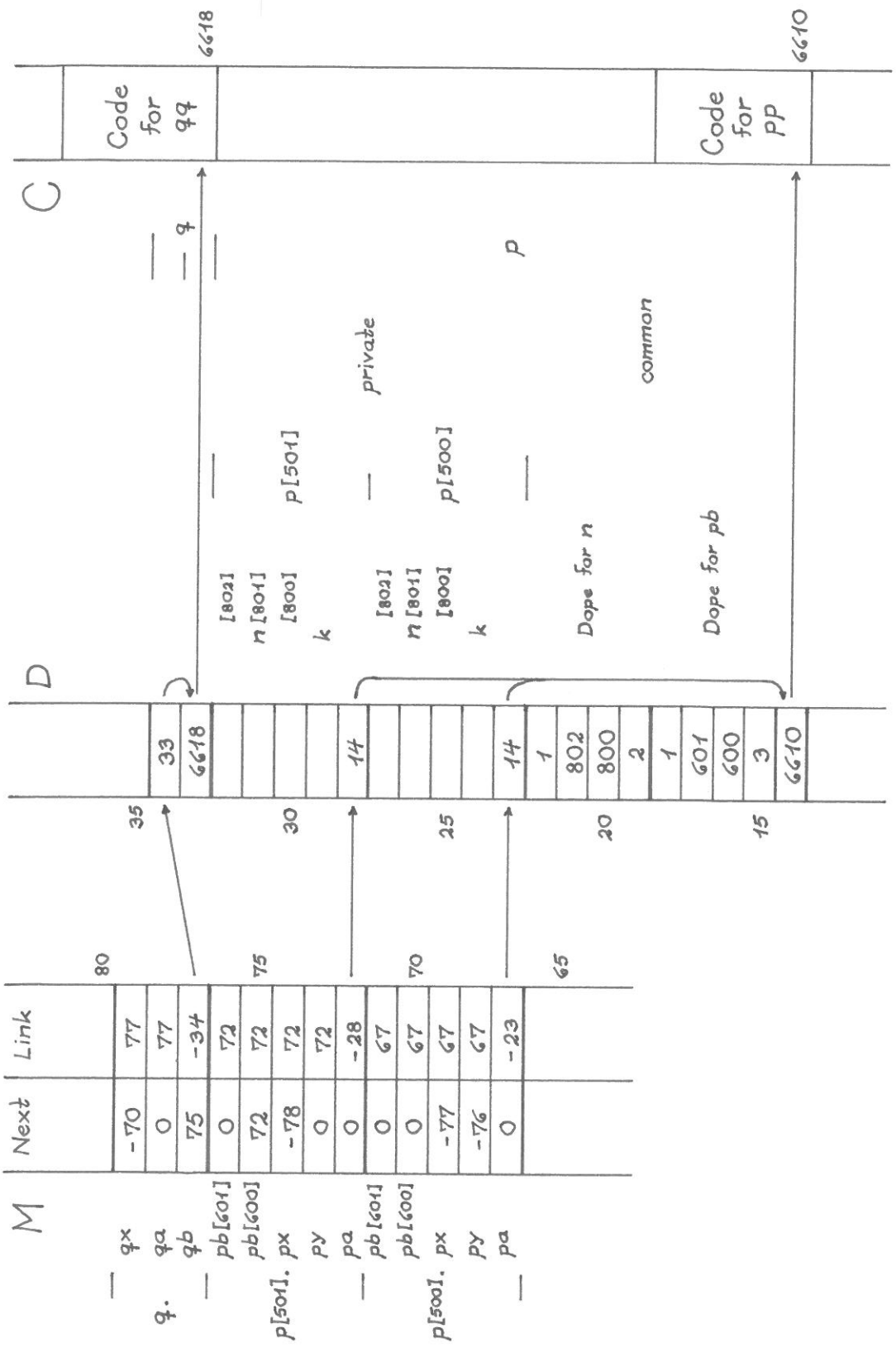

M
Next
Link
— qx -70 77
q. qa 0 77
qb 75 -34
— pb[601] 0 72
pb[600] 72 72
p[501]. px -78 72
py 0 72
pa 0 -28
— pb[601] 0 67
pb[600] 0 67
p[500]. px -77 67
py -76 67
pa 0 -23
80
75
70
65
35
30
25
20
15
D
33
6618
14
14
1
802
800
2
1
601
600
3
6610
[802]
n[801]
[800]
k
p[501]
[802]
n[801]
[800]
k
p[500]
Dope for n
Dope for pb
C
— q
private
common
p
Code for qq
Code for pp
6618
6610

Fig. 2

2.2.2 The algorithms.

Given these data structures, it is fairly easy to see how the informal algorithms presented earlier may be stated precisely. In the following, numbers in square brackets refer to line numbers of the informal algorithms.

[10,31,34,37]: All references to the value of a port translate in a straightforward manner into manipulations of the "value" field of an element of M.

[01,05,07,12,38,39]: The "father" field of elements of M is used to chain together components to form the queue Q. In this context, components are identified by their addresses in M.

[01]: The statement "Put all components into Q" is executed only once. It is implemented by stepping through all used elements of M. Boundaries between components are recognized by inspection of the sign of "link" fields.

[06]: Execution of a component requires access to

1). Its code.
2). Its ports.
3). Its local variables.
4). Dope vectors for 2) and 3).

From the address in M of the component, the D address of its private segment can be immediately obtained through the "link" field. Following another pointer, the common segment is found. And finally, a pointer in the common segment gives access to the code of the component.

[09-12]: Actually, this part of the algorithm is rather

```
09'  "For all input ports connected to the clock"
10'  DO BEGIN
11'       "Value of the port":=clock_voltage;
12'       "Find the component to which the port belongs";
13'       IF "that component is not in Q"
14'       THEN "put it into Q";
15'     END;
```

[09']: All input ports connected to the clock can be found by going through the equivalence list that has the clock port as its first element.

[12']: "Find the component" means determine its address. This is done by considering the "link" field of the port in hand.

[13']: The "last" field is used to signal whether a given component belongs to Q or not. It is updated every time the component is put into or taken out of Q.

[13]: In principle, the simulation can be thought of as going on forever. Therefore, the stop criterium, added for obvious practical reasons, need not be discussed here.

[35]: Given an output port, the input ports it is connected to can be found as described above for [9'].

[38]: See the remarks pertaining to [12'] and [13'].

2.3 Discussion.

Of course, Q has the role of what in other simulation systems is termed an "event list". Its simple form is due to the fact that the logical model implied by the semantics of HL does not assign definite delay times to circuits. Rather than stating that some event takes place at a specific point in time, it merely asserts that it will happen sometime in the future. Using the queue strategy, the simulation system is guaranteed to be in accordance with the model (under the obvious assumption that the execution of every primitive takes finite time). The very important conclusion is the following:

If a property of a circuit can be proved within the logical model, the circuit, when simulated, will exhibit the same property.

Loosely: If a circuit is "logically OK", it can be simulated. The converse is not true. To give an example of this would probably be rather cumbersome, but an informal argument goes as follows: The logical model demands that a circuit be "correct" under any distribution of relative delays. The simulation system instead picks a distribution "at random" and might hit one that makes the circuit "work", even if others wouldn't.

2.3.1 Scheduling.

A component is scheduled for execution (put into Q) as soon as one of its inputs changes, while another component is still in the process of execution (see the Port assignment algorithm).

Another possibility would have been to execute a com-

ponent C without immediate scheduling of other components, and thereafter go through the ports of C, looking for output ports with changed values, and scheduling as appropriate.

The difference between the two algorithms is so small that the choice between them is largely a matter of efficiency considerations.

A more radical change would be to queue (port,new value)-pairs instead of components. Processing a queue element (P,v) would mean

```
Give P the value v
Execute the component containing P
```

This would give the HL programmer a somewhat more explicit control over component execution. He would for example be able the deduce the exact number of executions of a particular component, since it would equal the number of input changes. But there seems to be little use for that kind of information, the axioms defining the simulation being apparently strong enough.

One last field of experiment could be the data structure Q. The "exact opposite" of a queue, a stack, would also work correctly except in cases where a component is executed an infinite number of times with no intervening clock changes. Such "pathological" cases model no physical reality; an example is given below.

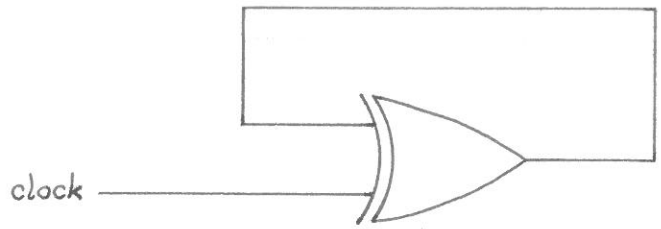


Fig. 3

2.3.2 Equivalence lists.

If port interconnections are viewed as a graph, ports being vertices and wires being directed edges, the graph has the properties

1) Each output port has inward valency 0.
2) Each input port has outward valency 0.

This means that the graph can be unambiguously represented as a set of equivalence lists. Since the only inquiries made

about the graph structure are of the form

Given an output port P.
What are the input ports connected to P ?

the equivalence lists are also a very economical way, in time and space, of representing the graph. One advantage is that there are no variable-size pointer areas in spite of the varying outward valencies of output ports.

2.3.3 Space considerations.

The conventions for "next" and "link" fields may seem a little strange. It would in fact be cleaner to use a few more fields to contain the same information in a straightforward manner. But M is by far the biggest data structure used by the system, and any field saved or wasted has considerable effect on the maximum size of object systems that can be simulated. I have accordingly bent a little the declared principle of simplest possible implementation.

2.3.4 Execution times.

The fears that the simulator might turn out to be too slow for interactive use have proved completely unfounded, at least for the kinds of circuits that have been simulated so far. With a realistic program, the user is hardly aware of the time spent in computation.

3. COMPILATION.

There is clearly a wide gap between HL and the data structures required by the simulation algorithms. Some form of translation is needed to bridge this gap. In fact, some of the language constructs of HL are so dynamic in nature that it appeared easier to divide the translation into two separate phases: compilation and run.

The compilation is very similar to the compilation of a language like Pascal. Its input is the "raw" HL program, and its output some form of "executable code". The compiler uses completely traditional techniques, it is slow and non-optimizing, and it was built around an automatically generated LALR(1)-parser.

4. RUN.

The run phase lies between compilation and simulation. It uses the code generated by the compiler to build (the static parts of) the internal model.

In this chapter, I shall first give an overview of the technique used. I shall then proceed to discuss in greater detail some issues of particular interest.

4.1 Overview.

The run phase is in many ways similar to the execution of a Pascal program, provided types and primitives are viewed as procedures with the side effects of extending the internal model.

Declarations as well as statements are executed. The declaration of a component is thus a "call" of a circuit. The effect of such a call is to augment the internal model by a new instance of the circuit. Calls may be nested, and a traditional run-time stack, placed in C, is employed to hold type variables and handle the flow-of-control. Execution is initiated by a "system" call of the anonymous "main type".

As regards primitives, only the declarations are executed, the statements in the body having no effect until simulation. The call therefore simply allocates space in M and D for the component's ports and local variables.

For types, declarations as well as bodies are executed at run-time. Declared ports are allocated space in M, and declared components are eventually also placed in M and D by their respective circuit calls. The net effect of the declarations is to assemble a collection of the constituent parts of the type. The connection statements in the type body thereafter modify this block of data to reflect the pattern of interconnections.

When two or more components are declared together, as in

```
COMP a,b:t    ,
```

the declaration could theoretically be compiled into a series of identical calls. For efficiency reasons, however, a single circuit call is instead followed by a duplication operation that expands the newly created instance into as many copies as needed.

The last action of the run phase is the compaction:

Types are effectively eliminated from the internal model by removing all type ports from the equivalence lists.

4.2 Dope vectors.

For the purpose of simplicity, the question of dope vectors was not considered above. Since arrays are dynamic, dope vectors are created at run time when array declarations are executed.

There are dope vectors describing arrays of objects of the following kinds:

1) Components
2) Primitive variables
3) Primitive ports
4) Type variables
5) Type ports

1). Dope vectors for component arrays are placed in the stack frame (in C) of the defining type.

2) and 3). The placement of primitive variable and port array dope vectors was explained in chapter 2.

4). Type variable array dope vectors are placed in the stack frame of the defining type along with the variables they describe.

5). Type port array dope vectors for ports of a type T are also placed on the run-time stack, but in the stack frame below the frame of T, i.e. in the stack frame of the type (possibly the main program) from which T was called.

4.3 Algorithms.

I do not want to explain the chosen algorithms in complete detail. They are not as transparent as could have been wished. In fact, one of the points I want to make is that a slip in the language design has made the implementation of the run phase more complicated than would otherwise have been the case.

In a later section, I shall expose the cause of the problem, propose and justify a change to HL, and hint at how the implementation has been carried out in spite of the difficulties. For the time being, I shall describe only some simpler parts of two central algorithms.

4.3.1 Connection.

In this section, I describe how the connection operator -> is implemented.

Suppose a connection statement prescribes that some output port X is to be connected to some input port Y.

If the language had not supported the type construct, the following simple algorithm would have been possible: Simply change the situation

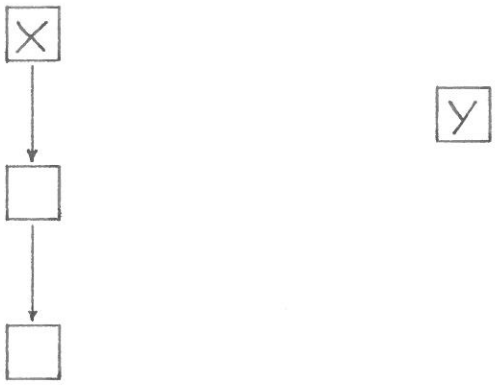


Fig. 4

to

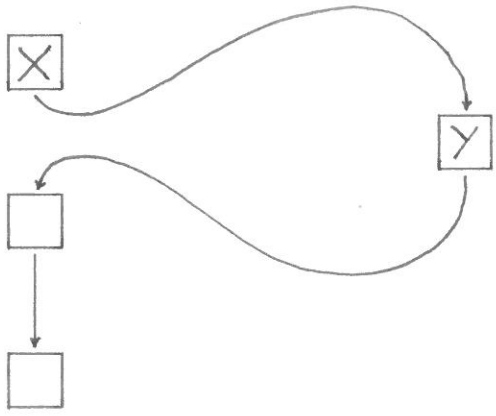


Fig. 5

(the boxes represent ports, and the arrows indicate the equivalence lists).

The scheme is not quite good enough as presented since it does not provide for a way of detecting the error of connecting two output ports to the same input port (in the first picture, Y would already be part of a non-trivial equivalence list). The field "father" of each port is therefore used to indicate whether that port is pointed to by some "next" field.

The presence of types complicates matters somewhat, and the algorithm pictured above is not adequate. The crucial point is that type ports do not have a fixed status (input or output). As explained in chapter 1, a port of a type T is treated as an input port when used within T, and as an output port when used outside T, or vice versa.

Taking into account the existance of type ports, the situation before a connection operation may look like

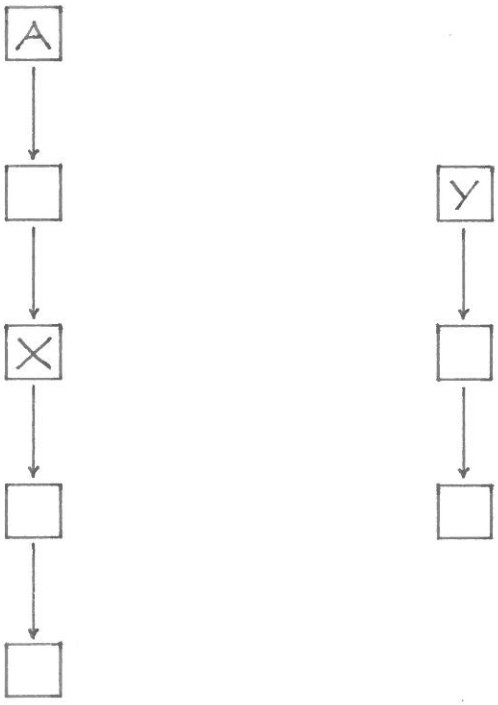


Fig. 6

What is required of the connection operation is to merge the two lists into one list with A as its head. With no regard to efficiency, this might be done by searching to the end of the list to the left, and changing a pointer in the obvious way. However, a pointer to the last element of the right list, stored in the "last" field of Y, makes it possible to perform the operation in constant time:

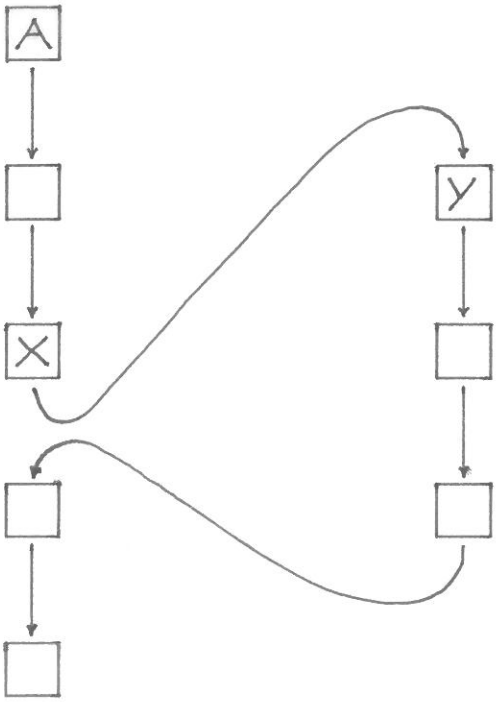


Fig. 7

Lastly a word about the fix statement. Suppose the input port Y is fixed at the value v. This is implemented simply by going through the equivalence list having Y as its head, and setting each "value" field equal to v.

4.3.2 Circuit duplication.

This section deals with the duplicating operation that was briefly mentioned in section 4.1. Since the data structure concerned consists mainly of pointers into the structure itself, the duplicating is not a simple "literal" copying and, in fact, it is not an entirely trivial problem.

The situation is as follows: Blocks of data in each of M and D, of sizes MSIZE and DSIZE, respectively, represent a particular circuit. I shall describe how to make a logical replica of this data structure. The reader may easily infer how to make more than one.

I call the circuit the active circuit (for this duplication), and its representation the active area.

The elements of M and D above the active area are not yet in use, so the actual finding of sufficient free space presents no problem.

The following schematic example may help to show what is involved:

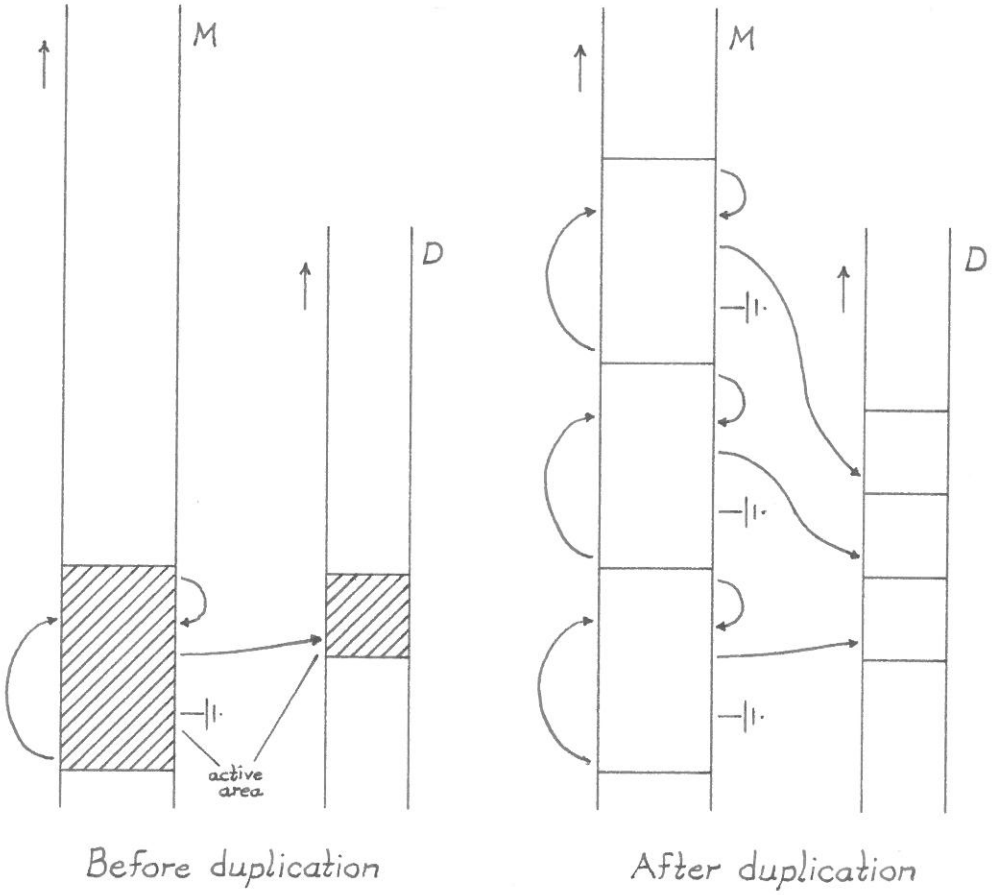


Fig. 8

Data in D.

Suppose the active circuit is a primitive. It then suffices to set up a new local store. This store must contain all zeros except for a pointer to the common segment.

If the active circuit is a type, its data in D has a more complex logical structure of some succession of common and private segments. It is a rather surprising fact that the simplest possible algorithm will do, namely the copying of the entire block of DSIZE integers with no modification at all.

Data in M.

Elements of M in the active area are considered one by one and each "translated" into a new record. The various fields are treated separately.

"Value" fields are simply copied.

"Link" fields are incremented by MSIZE, if they are positive. Negative "link" fields are decremented by DSIZE, if the active circuit is a type, and by the size of a local store, if it is a primitive.

"Next", "last" and "father" fields are changed according to a more complex algorithm. See section 4.4.2.

4.4 Discussion.

The design of the run-time system was to a large extent guided by the principle:

> If possible, the execution of a connection statement should take constant time, and a duplication operation should take time proportional to the size of the active area.

This goal has been fully achieved, but only at the price of a difficult-to-understand implementation. It may well be questioned whether this was a reasonable choice. The time spent in the run phase is short in absolute terms, and often negligble compared to either compilation or simulation time. A simple implementation would multiply execution times by a factor proportional to the length of equivalence lists, and this would perhaps be entirely acceptable.

4.4.1 Connection.

At first glance, the algorithm suggested by figures 6 and 7 seems to contain a flaw: For each port, a field "last" pointing to the last element in the port's equivalence list is postulated. But after the connection has been made, "last" fields in all elements of the original right list are wrong and cannot be updated in constant time.

However, it can be proved that these "last" fields, except that of Y itself, never are used again, neither in connection statements nor in duplications, and therefore need not be updated. The proof uses the scope rules and flow-of-control of HL and each of the restrictions placed on connection statements.

The same is not true of the "last" fields in the list to the left. But with the chosen merging algorithm, their value is still correct.

4.4.2 Duplication.

The purpose of the duplication operation is to provide one or more circuits that are different from, but "functionally equivalent" to a given circuit. This is what governs the reasoning below.

Data in D.

It may be observed that the duplication of types is somewhat wasteful of space: Dope vectors are copied but never used for address computation. But the waste is not inordinate, and to avoid it would be complicated as well as expensive.

Data in M.

"Value" fields.
If a port has been fixed to a value, that value is found in its "value" field. Otherwise, the "value" field contains 0. Therefore, value fields can be copied without modification.

"Link" fields.
The seemingly complex algorithm is best understood by looking at the example data structures and observing that positive "link" fields point into M, negative into D.

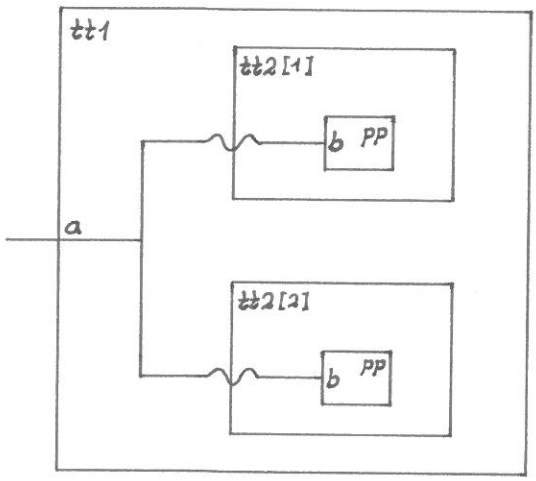
"Next", "link" and "father" fields.
I shall first propose a simple algorithm to deal with these fields, and will then show how it fails in certain circumstances.

Proposed algorithm:

> Increment "next" and "link" fields by MSIZE, if they are positive (decrement them by MSIZE, if they are negative), since they are pointers into the active area in M. If they are 0, copy them, since they point to nothing at all (0 is used as pointer value NIL).
>
> Copy "father" fields, since they indicate whether the port they belong to is pointed to by some "next" field.

Basically, the algorithm is correct. But consider the following correct HL program:

```
61    TYPE t1;
62      IN a;
63
64      TYPE t2;
65
66        PRIMITIVE p;
67          IN b;
68        BEGIN
69        END;
70
71        COMP pp:p;
72
73      BEGIN (* t2 *)
74        a->pp.b;
75      END;
76
77      COMP tt2[1..2]:t2;
78
79    BEGIN (* t1 *)
80    END;
81
82    COMP tt1:t1;
83
84  BEGIN (* main *)
85  END;
```



The connection statement a->pp.b in the body of t2 connects a port outside t2 to a port of a component declared inside t2. When the data structure corresponding to t2 is duplicated in line 77, this means that a "next" pointer <u>from outside</u> points into the active area. I refer to this phenomenon as an <u>external pointer</u>.

Fig. 9 illustrates the point and also shows what the result of the duplication must be.

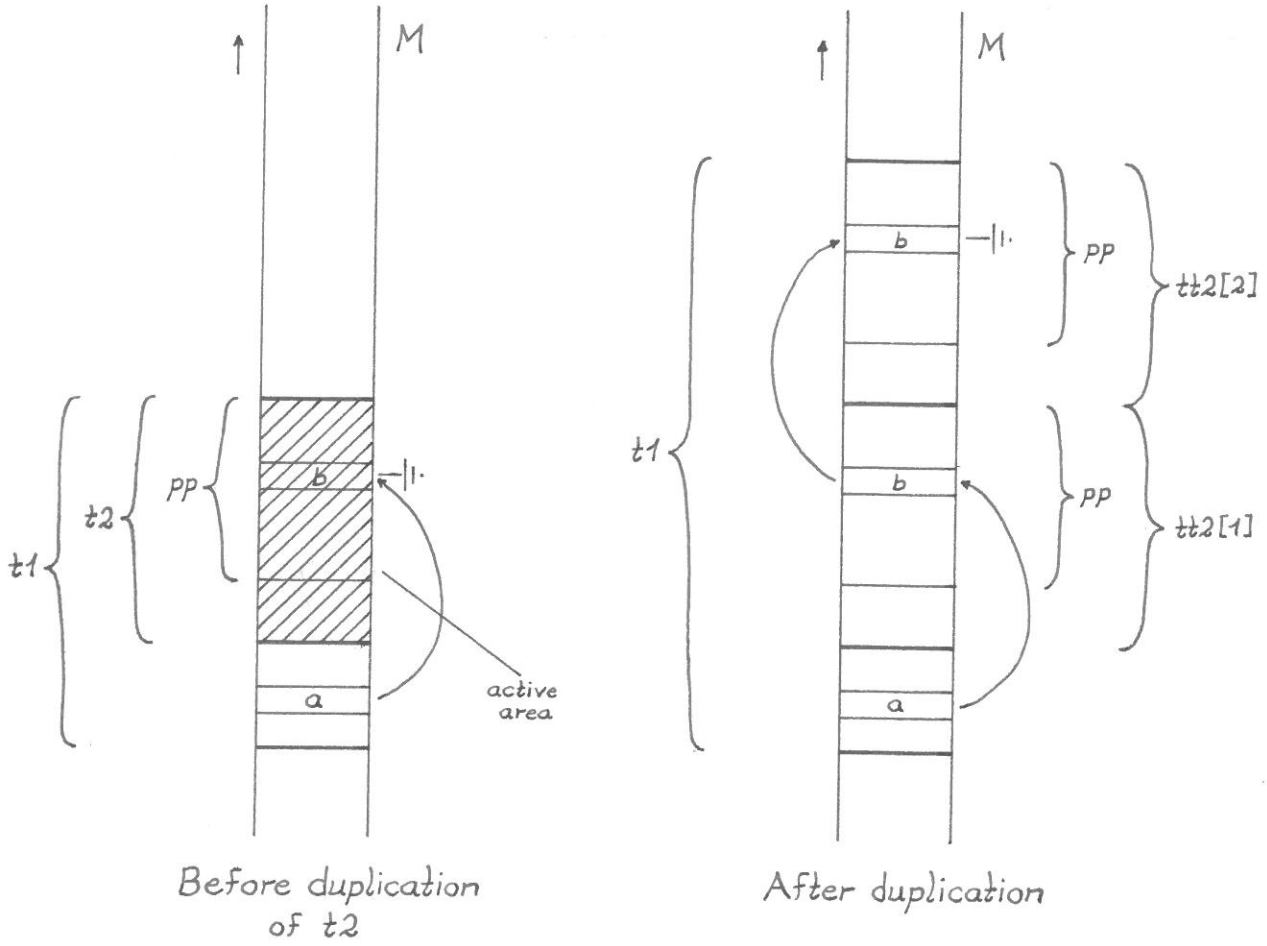


Fig. 9

This is clearly not what happens according to the simple proposed algorithm. Extra data fields must be added, or existing ones modified, to indicate the presence of external pointers, and extra algorithms must be added to take appropriate action.

I shall not go into the solution I have implemented. Instead, I want to ask the question: Is it really necessary to permit external pointers ?

I think not. In more than 2000 lines of HL programs, I have never used this facility, except when testing the implementation of HL. Furthermore, external pointers have a curious interpretation in the physical systems HL is supposed to model: Connecting a port outside a type T directly to a component inside T seems to suggest that you can pull a wire from outside a system to some point inside the system, but without ever crossing that system's boundary or interface to the outer world.

In programming terms, external pointers share some of the characteristics of side effects. It seems likely that

their use can result in needlessly obscure programs.

In conclusion, I think it would be a good idea to ban external pointers from HL.

This would mean a further departure from the scope rules of Pascal. But it would remove an unpleasant asymmetry from HL; connections "from outside to inside" would be prohibited as well as connections "from inside to outside". In practical terms, condition 1 a) page 11 would be changed to

> Both involved ports (in a fix statement: The involved port) must be declared within T. ,

and a minor change would be made to the compiler.

4.4.3 Compaction.

The compaction phase was added to speed up the simulation by shortening pointer chains.
In order to make a simple and fast algorithm possible, type ports are distinguished by having link=0.

4.4.4 Data allocation.

In the course of the run phase, storage for some objects, such as type variables, is naturally allocated on a stack. I call such objects non-permanent data, since they only exist during part of the run phase. On the other hand, some objects, such as primitive variables, must be preserved throughout the run phase (and simulation). They are permanent data.
If permanent and non-permanent data were allocated from the same pool, it would be close to impossible to reclaim the space taken up by non-permanent data. This is the reason why two arrays are used to hold integer data. D holds the permanent data and continually grows. C holds the non-permanent data and shrinks and grows in a stack fashion.

Type variables and their dope vectors, and dope vectors for arrays of components are non-permanent data kept in C.

Primitive variables and dope vectors for primitive variables and ports are permanent data kept in D.

Dope vectors for type ports are non-permanent data. However, since the scope of type ports extends one block level further than given by the scope rules of Pascal, a suitable place to store their dope vectors is in the stack frame below the topmost (currently active) stack frame.

It might be a good idea also to divide M into a permanent part holding primitive ports and a stack part holding type ports.

This would require the compiler and the run-time system to be slightly more complicated, and it would be necessary to perform a compaction at each type exit instead of once at the end of the run phase. But the scheme would probably result in a significant space saving.

5. THE SYSTEM'S USE IN THE HARDWARE COURSE.

The students attending the hardware course in the fall of 1981 were invited to use the simulation system. It was presented to them in a one-hour lecture and a few shorter follow-up sessions, and a user's manual was handed out. It was stressed that the system did not represent a compulsory part of the course.

The context.

The idea was to let the system appear as similar as possible to the breadboard kit. Students should be able to carry out a design in HL almost as they would have done it using the kit, and then translate the debugged end product almost automatically into a physical circuit.

For this reason, counterparts of the IC's in the kit were programmed in HL and collected in a context. Prior to compilation, the system added the context to the user program, so that the IC's appeared to be predefined.

One IC, a monostable multivibrator, could not be defined at all, and a few had to be slightly modified, for instance those having three-state outputs.

The context, some 1200 lines of HL, is appendix B to this report. It also defines most of the input-output facilities found on the breadboard, namely:

A group of 8 lights
A group of 8 manual switches
A group of 3 hexadecimal displays
A clock output
An input to stop the clock
A button to start the clock
A button to single-step the clock

The first three and the last two have fixed positions on the terminal screen corresponding to their position on the breadboard. The clock output is not identical to the predefined "clock" output described in section 1.2.3.

One important facility missing from the context are two so-called "dynamic inputs", buttons often used to effect an initialization of the circuit. As a kind of compensation, a special output "reset" having no breadboard analogue is provided. It has the value 1 prior to the first clock pulse and 0 ever afterwards.

Simulating EPROM's.

Micro-programming using an EPROM plays an important

role in the hardware course. Students have access to an EPROM burner controlled by a special program "PROM". Input to "PROM" is a description in some format of the desired memory contents.

To model the EPROM in HL, a program was written that takes as input a description in the same format and converts it into an HL primitive behaving like the programmed EPROM.

5.1 User reactions.

Towards the end of the semester, the course participants were asked to fill in a questionnaire about the use they had made of the system.

Some of the information obtained from this and other sources is given below together with some comments.

The user's manual was read by all students. About half actually tried out the system. Asked whether they judged it a useful design tool, the most common reaction was one of reservation and scepticism. However, this must be put into perspective by remarking that the students had spent on the average less than 10 hours on all related activities, attending lectures, reading the manual, programming and running programs.

Positive comments centered around the readability of a well-structured and well-documented HL program as compared to a "bird's nest" of wires. Small changes are easily and safely made in an editor. Although none of the students remarked so, this is especially true in the case of an EPROM. Reprogramming a physical EPROM takes some 30 minutes, albeit without the need for permanent human attention, whereas a new HL EPROM can be obtained in less than a minute.

Finally, the type concept was commended as a good way to convey the structure of a design.

As for negative comments, the most frustrating thing about the system appeared to be its long "start-up" time. The complete "want list" was, in order of decreasing importance:

- Faster compilation
- Dynamic inputs
- Adjustable clock period
- WITH statement

Common to these complaints and wishes is that they do not point to any basic deficiency of the system.

Faster compilation.

Compilation of the context takes approximately 40

seconds of machine time. The user program in most cases adds insignificantly to this figure. Compilation could be speeded up by the following means:

1) More efficient compilation techniques.
   It would probably be difficult to gain very much in this direction.

2) Introduction of circuit parameters.
   The context could be shortened somewhat.

3) Precompiling the context.
   What I mean is: Compiling the context once and for all, and not every time anew. This way, compilation time could be reduced to a fraction.

Dynamic inputs.

The analogue of dynamic inputs would be keyboard commands having an immediate effect, even when the simulation system is not expecting any input. The only reason why this was not included in the context were implementational difficulties: The local version of Pascal does not provide a way of inquiring about the presence of a character in the keyboard buffer without actually waiting until a key is depressed.

Dynamic inputs could be implemented by resorting to subroutines in assembly language. They could be interfaced to the system through a small number of predefined input routines and a predefined output port indicating the presence of an unprocessed character. In effect, this would amount to an interrupt system, and the component(s) connected to the new port would constitute the interrupt handler.

Adjustable clock period.

Whereas the breadboard clock period can be adjusted within certain limits, the system's clock at present runs as fast as possible (real time). An adjustable clock period could be obtained simply by introducing a predefined procedure "setclock".

WITH statement.

Was discussed in section 1.3.14.

Personal ideas.

Something I have been missing myself is a debugging system. After all, the system is supposed to support the activity of design and testing, and I feel that its usefulness would be greatly enhanced by the addition of carefully

designed debugging aids.

It is possible to do debugging with the present system; one can, for example, declare a special debugging component, give it an input from every "strategic" point in the circuit and let it continuously evaluate some expression representing "all is OK". When this condition fails, the debugging component displays the information available to it. However, this is a comparatively tedious process, and there might be better ways.

Taking speculations one step further, one could imagine the simulation system coupled to some sort of graphics system, so that the user could

a) Give the input to the system interactively in the form of a series of (changes to) diagrams.

b) Make the system display the object system it had perceived, or some specified part of it.

6. OTHER SIMULATION SYSTEMS.

In this chapter, I shall try to compare the simulation system to other systems designed to perform similar tasks. For ease of reference, the system described in this report will be called "HSIM".

I have never actually worked with another hardware simulation system, and my knowledge is based exclusively on a small number of articles, some of which do not even deal principally with simulation. I shall therefore not attempt any extensive analysis but restrict myself to a few general and qualitative remarks.

My most important sources were [5] and [6]. [5] offers a useful classification of simulators in various directions, and [6] is my only detailed example of a description language.
I have also studied [7] and [8].

Delays.

The most significant distinction between HSIM and other systems probably concerns the handling of delays. The great majority of simulators presented in [7] have some form of assignable delays, i.e. delay times are specified by the user. Simulation with zero delays, equivalent to the simulation done by HSIM, is termed "logic verification" in [5] and is by and large discarded as too primitive and too far removed from physical reality.

Primitives.

[5] discusses whether basic circuit building blocks should be restricted to gates or whether elements such as flip-flops should also be supported (parameterized versions are also considered). In [6], the set of basic elements is small and fixed, although general combinational circuits can be specified in another way (through boolean equations). The idea that the user should be given the freedom to define whatever primitives he wishes does not seem to enjoy great popularity; at least, I have not found a single example in the literature of anything comparable to primitive definitions.
It might be suspected that a flexible and extensible set of primitives were a mere theoretical gadget with no practical usefulness. However, at least for the use I have made of HSIM, this is not true.

Types.

Unless the macro expansion is supposed to be restricted

to one level, the macro preprocessor approach in [5] gives the user the same advantages that the type concept gives the HL programmer. In [6], there is no equivalent to types.

Comparing HL directly to the one competitor I have discovered in the literature, the language described in [6], it can hardly be disputed that HL is superior as concerns generality and elegance, probably also, except in the case of very small systems, ease of use. But this is not surprising; you cannot fairly compare two systems separated by nearly 20 years.

Concluding, it may be said that HSIM is definitively in the lightweight end of simulation systems, but that it still compares well with other systems in a number of ways.

7. CONCLUSION.

The presented system is simple and easy to use. It is probably not sufficiently sophisticated to be of much use in a commercial context, but it is well suited to assist in designs that are carried out according to simple logical models without definite circuit delays.

Being a pilot project, it lacks all but the most essential facilities. There is ample room for improvement and extension.

References.

[1]: David Winkel and Franklin Prosser: "The Art of Digital Design. An Introduction to Top-Down Design". Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[2]: Carver Mead and Lynn Conway: "Introduction to VLSI Systems". Addison-Wesley, Reading, Massachusetts, 1980.

[3]: Arthur D. Friedman and Premachandran R. Menon: "Theory & Design of Switching Circuits". Computer Science Press, Woodland Hills, California, 1975.

[4]: Luca Cardelli: "Sticks & Stones: An Applicative VLSI Design Language". Internal report, University of Edinburgh, July, 1981.

[5]: S. A. Szygenda and E. W. Thompson: "Digital Logic Simulation In a Time-Based, Table-Driven Environment". Part 1: "Design Verification". Computer, vol. 8, no. 3, Mar. 1975, pp. 24-36.

[6]: Robert M. McClure: "A Programming Language for Simulating Digital Systems". Journal of the ACM, vol. 12, no. 1, Jan. 1965, pp. 14-22.

[7]: Melvin A. Breuer, Arthur D. Friedman and Alexander Iosupowicz: "A Survey of the State of the Art of Design Automation". Computer, vol. 14, no. 10, Oct. 1981, pp. 58-75.

[8]: James R. Armstrong and Garry W. Woodruff: "Chip-level Simulation of Microprocessors". Computer, vol. 13, no. 1, Jan. 1980, pp. 94-100.

Appendices.

A: A BNF grammar for HL.

B: Context used in the hardware course.

Index.

The following terms are used in a precise and technical sense in the report. The page numbers indicate their point of introduction.

Appendix A

A BNF grammar for HL.

```
<program> ::= <declaration part> <body> ;

<declaration part> ::= <declaration sequence>
                     / <empty>

<declaration sequence> ::= <declaration>
                         / <declaration sequence>
                           <declaration>

<declaration> ::= <header> ; <declaration part> <body> ;
                / VAR <item list> ;
                / IN <item list> ;
                / OUT <item list> ;
                / COMP <component declaration sequence>

<header> ::= PRIMITIVE <identifier>
           / TYPE <identifier>

<itemlist> ::= <item>
             / <item list> , <item>

<item> ::= <identifier>
         / <identifier> [ <range list> ]

<range list> ::= <range>
               / <range list> , <range>

<range> ::= <expression> .. <expression>

<component declaration sequence> ::= <component declaration>
                                   / <component declaration sequence>
                                     <component declaration>

<component declaration> ::= <item list> : <identifier>

<body> ::= BEGIN <statement sequence> END

<statement sequence> ::= <statement>
                       / <statement sequence> ; <statement>

<statement> ::= <elementary statement>
              / <if part> <then part> ELSE <statement>
              / <if part> THEN <statement>
              / <while part> DO <statement>
              / <for part> DO <statement>

<if part> ::= IF <expression>
```

```
<then part> ::= THEN <restricted statement>

<while part> ::= WHILE <expression>

<for part> ::= FOR <identifier> := <expression>
                 TO <expression>
             / FOR <identifier> := <expression>
              STEP <expression> TO <expression>

<restricted statement> ::= <elementary statement>
       / <if part> <then part> ELSE <restricted statement>
       / <while part> DO <restricted statement>
       / <for part> DO <restricted statement>

<elementary statement> ::= <variable> := <expression>
      / <compound> -> <compound>
      / <source> -> <compound>
      / REPEAT <statement sequence> UNTIL <expression>
      / REACH ( <expression> , <expression> )
      / REACH ( <expression> , <expression> : <expression> )
      / READ ( <variable> )
      / WRITE ( <field list> )
      / WRITE ( <string> )
      / <body>
      / <empty>

<variable> ::= <identifier>
             / <identifier> [ <expression list> ]

<compound> ::= <variable>
             / <variable> . <variable>

<source> ::= <constant>
           / ( <expression> )

<expression list> ::= <expression>
                    / <expression list> , <expression>

<expression> ::= <arithmetic expression>
               <relational operator> <arithmetic expression>
               / <arithmetic expression>

<arithmetic expression> ::= <arithmetic expression>
                              <adding operator> <term>
                          / <term>

<term> ::= <term> <multiplying operator> <primary>
         / <primary>

<primary> ::= <variable>
            / <constant>
            / ( <expression> )
```

```
<field list> ::= <field>
               / <field list> , <field>

<field> ::= <expression>
          / <expression> : <expression>

<relational operator> ::= =
                        / <>
                        / <
                        / >
                        / <=
                        / >=

<adding operator> ::= +
                    / -

<multiplying operator> ::= *
                         / DIV
                         / MOD

<constant> ::= <digits>
             / + <digits>
             / - <digits>
```

<Identifier>, <digits> and <empty> have not been defined. <Empty> means the empty string, <digits> is one or more digits, and <identifier> is a letter followed by zero or more letters or digits.

Appendix B

Context used in the hardware course.

```
(*****************************************)
(*                                       *)
(*                CONTEXT                *)
(*    DESCRIBING INTEGRATED CIRCUITS     *)
(*                                       *)
(*           To be used with             *)
(*      hardware simulation system       *)
(*                                       *)
(*****************************************)


(* QUARTER 74LS02 *)

  primitive nor;
    in i[1..2];
    out o;
  begin
       if i[1]
       then o:=0
       else o:=1-i[2];
  end;


  primitive nor3;
    in i[1..3];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then begin
                       o:=0;
                       k:=5;
                  end
             else k:=k+1;
       until k>3;
       if k=4
       then o:=1;
  end;


  primitive nor4;
    in i[1..4];
```

```
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then begin
                        o:=0;
                        k:=6;
                  end
             else k:=k+1;
       until k>4;
       if k=5
       then o:=1;
  end;


(* HALF 74LS260 *)

  primitive nor5;
    in i[1..5];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then begin
                        o:=0;
                        k:=7;
                  end
             else k:=k+1;
       until k>5;
       if k=6
       then o:=1;
  end;


  primitive nor6;
    in i[1..6];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then begin
                        o:=0;
                        k:=8;
                  end
             else k:=k+1;
       until k>6;
       if k=7
       then o:=1;
```

```
  end;


(* QUARTER 74LS08 *)

  primitive and;
    in i[1..2];
    out o;
  begin
       if i[1]
       then o:=i[2]
       else o:=0;
  end;


  primitive and3;
    in i[1..3];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then k:=k+1
             else begin
                       o:=0;
                       k:=5;
                  end;
       until k>3;
       if k=4
       then o:=1;
  end;


(* HALF 74LS21 *)

  primitive and4;
    in i[1..4];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then k:=k+1
             else begin
                       o:=0;
                       k:=6;
                  end;
       until k>4;
       if k=5
       then o:=1;
  end;
```

```
primitive and5;
  in i[1..5];
  out o;
  var k;
begin
     k:=1;
     repeat
           if i[k]
           then k:=k+1
           else begin
                     o:=0;
                     k:=7;
                end;
     until k>5;
     if k=6
     then o:=1;
end;


primitive and6;
  in i[1..6];
  out o;
  var k;
begin
     k:=1;
     repeat
           if i[k]
           then k:=k+1
           else begin
                     o:=0;
                     k:=8;
                end;
     until k>6;
     if k=7
     then o:=1;
end;


(* SIXTH 74LS14 *)

primitive inv;
  in i;
  out o;
begin
     o:=1-i;
end;


(* QUARTER 74LS132 *)

primitive nand;
  in i[1..2];
  out o;
```

```
  begin
       if i[1]
       then o:=1-i[2]
       else o:=1;
  end;


(* HALF 74LS20 *)

  primitive nand4;
    in i[1..4];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then k:=k+1
             else begin
                       o:=1;
                       k:=6;
                  end;
       until k>4;
       if k=5
       then o:=0;
  end;


  primitive nand5;
    in i[1..5];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then k:=k+1
             else begin
                       o:=1;
                       k:=7;
                  end;
       until k>5;
       if k=6
       then o:=0;
  end;


(* 74LS133 *)

  primitive nand13;
    in i[1..13];
    out o;
    var k;
  begin
```

```
        k:=1;
        repeat
              if i[k]
              then k:=k+1
              else begin
                        o:=1;
                        k:=15;
                   end;
        until k>13;
        if k=14
        then o:=0;
  end;


(* QUARTER 74LS32 *)

  primitive or;
    in i[1..2];
    out o;
  begin
       if i[1]
       then o:=1
       else o:=i[2];
  end;


  primitive or4;
    in i[1..4];
    out o;
    var k;
  begin
       k:=1;
       repeat
             if i[k]
             then begin
                       o:=1;
                       k:=6;
                  end
             else k:=k+1;
       until k>4;
       if k=5
       then o:=0;
  end;


(* HALF 74LS74 *)

  primitive flipflop;
    in d,cp,n_sd,n_cd;
    out q,n_q;
    var n;

  begin
       if n_sd
```

```
        then if n_cd
             then if cp
                  then begin
                            q:=n;
                            n_q:=1-n;
                       end
                  else n:=d
             else begin
                       n:=0;
                       q:=0;
                       n_q:=1;
                  end
        else if n_cd
             then begin
                       n:=1;
                       q:=1;
                       n_q:=0;
                  end
             else begin
                       q:=1;
                       n_q:=1;
                  end;
  end;

(* QUARTER 74LS86 *)

  primitive xor;
    in i[1..2];
    out o;
  begin
       o:=i[1]<>i[2];
  end;


(* 74LS138 *)

  type decoder;
    in a[0..2],n_e1,n_e2,e3;
    out n_o[0..7];
    comp y[0..2]:inv;
         no:nor;
         a2:and;
         na4[0..7]:nand4;
    var j,k,l,n;

  begin
       n_e1->no.i[1];
       n_e2->no.i[2];
       no.o->a2.i[1];
       e3->a2.i[2];
       n:=1;
       for k:=0 to 2
       do begin
```

```
                a[k]->y[k].i;
                l:=0;
                repeat
                      for j:=l to l+n-1
                      do y[k].o->na4[j].i[k+1];
                      l:=l+2*n;
                      for j:=l-n to l-1
                      do a[k]->na4[j].i[k+1];
                until l=8;
                n:=n*2;
          end;
       for k:=0 to 7
       do begin
                a2.o->na4[k].i[4];
                na4[k].o->n_o[k];
          end;
  end;


(* 74LS153 *)

  type select2;
    in i[1..2,0..3],s[0..1],n_e[1..2];
    out z[1..2];
    comp sy[0..1],ey[1..2]:inv;
         a4[1..2,0..3]:and4;
         u4[1..2]:or4;
    var j,k,l;

  begin
       for k:=1 to 2
       do begin
                s[k-1]->sy[k-1].i;
                n_e[k]->ey[k].i;
                for l:=0 to 1
                do for j:=0 to 1
                   do begin
                            s[l]->a4[k,3-j*(2-l)].i[l+1];
                            sy[l].o->a4[k,j*(2-l)].i[l+1];
                      end;
                for l:=0 to 3
                do begin
                        i[k,l]->a4[k,l].i[3];
                        ey[k].o->a4[k,l].i[4];
                        a4[k,l].o->u4[k].i[l+1];
                   end;
                u4[k].o->z[k];
          end;
  end;


(* 74LS157 *)

  type select4;
```

```
    in i[0..1,0..3],s,n_e;
    out y[0..3];
    comp no[0..1]:nor;
         ys:inv;
         a[0..3,0..1]:and;
         u[0..3]:or;
    var k,l;

  begin
       for k:=0 to 1
       do n_e->no[k].i[1];
       s->no[0].i[2];
       s->ys.i;
       ys.o->no[1].i[2];
       for k:=0 to 3
       do begin
                for l:=0 to 1
                do begin
                        i[l,k]->a[k,l].i[1];
                        no[l].o->a[k,l].i[2];
                        a[k,l].o->u[k].i[l+1];
                   end;
                u[k].o->y[k];
          end;
  end;


(* 74LS161 *)

  type counter1;
    in d[0..3],cp,n_mr,n_pe,cep,cet;
    out q[0..3],tc;

    primitive mbit;
      in i,cp,n_mr;
      out o;
      var n;
    begin
         if n_mr
         then if cp
              then o:=n
              else n:=i
         else begin
                   n:=0;
                   o:=0;
              end;
    end;

    comp y:inv;
         a3:and3;
         a4[0..3]:and4;
         ac[0..3],ap[0..3]:and;
         x[0..3]:xor;
         u[0..3]:or;
```

```
        b[0..3]:mbit;
        a5:and5;
    var k,l;

begin
      n_pe->y.i;
      n_pe->a3.i[1];
      cep->a3.i[2];
      cet->a3.i[3];
      for k:=0 to 3
      do begin
            n_pe->ac[k].i[1];
            y.o->ap[k].i[1];
            d[k]->ap[k].i[2];
            a3.o->a4[k].i[1];
            a4[k].o->x[k].i[1];
            ac[k].o->x[k].i[2];
            x[k].o->u[k].i[1];
            ap[k].o->u[k].i[2];
            u[k].o->b[k].i;
            n_mr->b[k].n_mr;
            cp->b[k].cp;
            b[k].o->q[k];
            b[k].o->ac[k].i[2];
            for l:=k+1 to 3
            do b[k].o->a4[l].i[k+2];
            for l:=k+2 to 4
            do l->a4[k].i[l];
            b[k].o->a5.i[k+1];
         end;
      cet->a5.i[5];
      a5.o->tc;
end;


(* 74LS169 *)

  type counter2;
    in d[0..3],up,cp,n_pe,n_cep,n_cet;
    out q[0..3],n_tc;

    primitive bit;
      in i,cp;
      out o;
      var n;
    begin
        if cp
        then o:=n
        else n:=i;
    end;

    comp eno,tno:nor;
         py,uy,by[0..3],ey:inv;
         a1[0..3],a2[0..3]:and;
```

```
        x[0..3]:xor;
        u[0..3,1..2],u1[0..3]:or;
        b[0..3]:bit;
        a4[0..3,1..2]:and4;
        a6[1..2]:and6;
    var j,k,l;

begin
     n_cep->eno.i[1];
     n_cet->eno.i[2];
     n_pe->py.i;
     up->uy.i;
     for k:=0 to 3
     do begin
             eno.o->a1[k].i[1];
             a1[k].o->x[k].i[1];
             x[k].o->u[k,1].i[1];
             py.o->u[k,1].i[2];
             d[k]->u[k,2].i[1];
             n_pe->u[k,2].i[2];
             for l:=1 to 2
             do u[k,l].o->a2[k].i[l];
             a2[k].o->b[k].i;
             cp->b[k].cp;
             b[k].o->q[k];
             b[k].o->x[k].i[2];
             b[k].o->by[k].i;
             up->a4[k,1].i[1];
             uy.o->a4[k,2].i[1];
             for l:=0 to k-1
             do begin
                     b[l].o->a4[k,1].i[l+2];
                     by[l].o->a4[k,2].i[l+2];
                end;
             for l:=k+2 to 4
             do for j:=1 to 2
                do 1->a4[k,j].i[l];
             for l:=1 to 2
             do a4[k,l].o->u1[k].i[l];
             u1[k].o->a1[k].i[2];
             b[k].o->a6[1].i[k+2];
             by[k].o->a6[2].i[k+2];
        end;
     up->a6[1].i[1];
     uy.o->a6[2].i[1];
     n_cet->ey.i;
     for k:=1 to 2
     do begin
             ey.o->a6[k].i[6];
             a6[k].o->tno.i[k];
        end;
     tno.o->n_tc;
end;
```

```
(* 74LS175 *)

  type flipflop4;
    in d[0..3],cp,n_mr;
    out q[0..3],n_q[0..3];

    primitive mbit;
      in i,cp,n_mr;
      out o;
      var n;
    begin
         if n_mr
         then if cp
              then o:=n
              else n:=i
         else begin
                   n:=0;
                   o:=0;
              end;
    end;

    comp b[0..3]:mbit;
         y[0..3]:inv;
    var k;

  begin
       for k:=0 to 3
       do begin
                d[k]->b[k].i;
                cp->b[k].cp;
                n_mr->b[k].n_mr;
                b[k].o->q[k];
                b[k].o->y[k].i;
                y[k].o->n_q[k];
          end;
  end;


(*********************************)
(*                               *)
(*    Modified version of        *)
(*          74LS181              *)
(*                               *)
(*    Output a_eq_b is not       *)
(*       open collector          *)
(*                               *)
(*********************************)

  type alu;
    in n_a[0..3],n_b[0..3],s0,s1,s2,s3,ci,m;
    out n_f[0..3],co,n_g,n_p,a_eq_b;
    comp yb[0..3],ym:inv;
         a3[0..3,0..4]:and3;
```

```
        no[0..3]:nor;
        no3[0..3]:nor3;
        x1[0..3],x2[0..3]:xor;
        a5[1..13]:and5;
        no4[0..3]:nor4;
        na5[1..2]:nand5;
        na0,nac:nand;
        a4:and4;
  var j,k,l,n;

begin
     n:=1;
     for k:=0 to 3
     do begin
             for l:=2 to 4
             do n_a[k]->a3[k,l].i[1];
             n_b[k]->a3[k,0].i[2];
             n_b[k]->a3[k,3].i[2];
             s0->a3[k,0].i[3];
             s1->a3[k,1].i[3];
             s2->a3[k,2].i[3];
             s3->a3[k,3].i[3];
             n_b[k]->yb[k].i;
             yb[k].o->a3[k,1].i[2];
             yb[k].o->a3[k,2].i[2];
             1->a3[k,0].i[1];
             1->a3[k,1].i[1];
             for l:=2 to 3
             do begin
                     1->a3[k,4].i[l];
                     a3[k,l].o->no[k].i[l-1];
                end;
             for l:=0 to 1
             do a3[k,l].o->no3[k].i[l+1];
             a3[k,4].o->no3[k].i[3];
             no[k].o->x1[k].i[1];
             no3[k].o->x1[k].i[2];
             x1[k].o->x2[k].i[1];
             for l:=0 to k
             do begin
                     no3[l].o->a5[n+l].i[2];
                     for j:=l+1 to k
                     do no[j].o->a5[n+l].i[j-l+2];
                     for j:=k-l+3 to 5
                     do 1->a5[n+l].i[j];
                     a5[n+l].o->no4[k].i[l+1];
                end;
             for l:=k+2 to 3
             do 0->no4[k].i[l];
             n:=n+k+2;
             if k<3
             then begin
                       ci->a5[n-1].i[5];
                       no4[k].o->x2[k+1].i[2];
```

```
                        for l:=0 to k
                        do no[l].o->a5[n-1].i[l+2];
                        for l:=k+3 to 4
                        do 1->a5[n-1].i[l];
                        a5[n-1].o->no4[k].i[4];
                    end;
                for l:=1 to 2
                do no[k].o->na5[l].i[k+1];
                x2[k].o->n_f[k];
                x2[k].o->a4.i[k+1];
            end;
        m->ym.i;
        for k:=1 to 9
        do ym.o->a5[k].i[1];
        for k:=10 to 13
        do 1->a5[k].i[1];
        ci->na0.i[1];
        ym.o->na0.i[2];
        na0.o->x2[0].i[2];
        1->na5[1].i[5];
        ci->na5[2].i[5];
        na5[1].o->n_p;
        na5[2].o->nac.i[1];
        no4[3].o->nac.i[2];
        nac.o->co;
        no4[3].o->n_g;
        a4.o->a_eq_b;
  end;


(* 74LS194 *)

  type shift;
    in d[0..3],dsr,dsl,s0,s1,cp,n_mr;
    out q[0..3];

    primitive mbit;
      in i,cp,n_mr;
      out o;
      var n;
    begin
        if n_mr
        then if cp
             then o:=n
             else n:=i
        else begin
                  n:=0;
                  o:=0;
             end;
    end;

    comp y0,y1:inv;
        a[0..3,1..4]:and3;
        u[0..3]:or4;
```

```
        b[0..3]:mbit;
    var k,l;

begin
      s0->y0.i;
      s1->y1.i;
      for k:=0 to 3
      do begin
            s0->a[k,1].i[3];
            s0->a[k,2].i[1];
            s1->a[k,2].i[2];
            s1->a[k,3].i[2];
            y0.o->a[k,3].i[1];
            y0.o->a[k,4].i[1];
            y1.o->a[k,1].i[2];
            y1.o->a[k,4].i[2];
            d[k]->a[k,2].i[3];
            b[k].o->a[k,4].i[3];
            for l:=1 to 4
            do a[k,l].o->u[k].i[l];
            u[k].o->b[k].i;
            cp->b[k].cp;
            n_mr->b[k].n_mr;
            b[k].o->q[k];
         end;
      for k:=0 to 2
      do begin
            b[k].o->a[k+1,1].i[1];
            b[k+1].o->a[k,3].i[3];
         end;
      dsr->a[0,1].i[1];
      dsl->a[3,3].i[3];
end;


(* 74LS377 *)

  type flipflop8;
    in d[0..7],cp,n_e;
    out q[0..7];

    primitive bit;
      in i,cp;
      out o;
      var n;
    begin
        if cp
        then o:=n
        else n:=i;
    end;

    comp y:inv;
        a1[0..7],a2[0..7]:and;
        u[0..7]:or;
```

```
        b[0..7]:bit;
    var k;

  begin
       n_e->y.i;
       for k:=0 to 7
       do begin
                y.o->a1[k].i[1];
                d[k]->a1[k].i[2];
                n_e->a2[k].i[1];
                a1[k].o->u[k].i[1];
                a2[k].o->u[k].i[2];
                u[k].o->b[k].i;
                cp->b[k].cp;
                b[k].o->a2[k].i[2];
                b[k].o->q[k];
          end;
  end;


(* 74LS379 *)

  type register379;
    in d[0..3],cp,n_e;
    out q[0..3],n_q[0..3];

    primitive bit;
      in i,cp,n_e;
      out o,n_o;
      var n,e;
    begin
         if cp
         then begin
                   if e
                   then begin
                             o:=n;
                             n_o:=1-n;
                        end;
              end
         else begin
                   n:=i;
                   e:=1-n_e;
              end;
    end;

    comp b[0..3]:bit;
    var k;

  begin
       for k:=0 to 3
       do begin
                d[k]->b[k].i;
                cp->b[k].cp;
                n_e->b[k].n_e;
```

```
                b[k].o->q[k];
                b[k].n_o->n_q[k];
            end;
    end;


(* 74LS398 *)

    type register398;
      in i[0..1,0..3],s,cp;
      out q[0..3],n_q[0..3];

      primitive bit;
        in i,cp;
        out o,n_o;
        var n;
      begin
           if cp
           then begin
                      o:=n;
                      n_o:=1-n;
                end
           else n:=i;
      end;

      comp y:inv;
           a[0..1,0..3]:and;
           u[0..3]:or;
           b[0..3]:bit;
      var k,l;

    begin
         s->y.i;
         for k:=0 to 3
         do begin
                  s->a[1,k].i[1];
                  y.o->a[0,k].i[1];
                  for l:=0 to 1
                  do begin
                           i[l,k]->a[l,k].i[2];
                           a[l,k].o->u[k].i[l+1];
                     end;
                  u[k].o->b[k].i;
                  cp->b[k].cp;
                  b[k].o->q[k];
                  b[k].n_o->n_q[k];
           end;
    end;
```

```
(*******************************)
(*                             *)
(*    Modified version of      *)
(*            2101             *)
(*                             *)
(*  Input OD is permanently    *)
(*         tied to 0           *)
(*   Inputs a[4]..a[7] are     *)
(*   permanently tied to 0     *)
(*                             *)
(*******************************)

  type ram;

    in a[0..3],di[1..4],n_w,n_ce1,ce2;
    out o[1..4];

    primitive wbit;
      in i,w;
      out o;
      var n;
    begin
         if w
         then begin
                    n:=i;
                    o:=n;
              end;
    end;

    primitive or16;
      in i[1..16];
      out o;
      var k;
    begin
         k:=1;
         repeat
               if i[k]
               then begin
                          o:=1;
                          k:=18;
                    end
               else k:=k+1;
         until k>16;
         if k=17
         then o:=0;
    end;

    comp no:nor;
         ea,aw[0..15],ar[0..15,1..4]:and;
         a4[0..15]:and4;
         b[0..15,1..4]:wbit;
         o16[1..4]:or16;
         y[0..3]:inv;
    var j,k,l,n;
```

```
begin
      n_w->no.i[1];
      n_ce1->no.i[2];
      ce2->ea.i[1];
      no.o->ea.i[2];
      for k:=0 to 15
      do begin
               ea.o->aw[k].i[1];
               a4[k].o->aw[k].i[2];
               for l:=1 to 4
               do begin
                        di[l]->b[k,l].i;
                        aw[k].o->b[k,l].w;
                        b[k,l].o->ar[k,l].i[1];
                        a4[k].o->ar[k,l].i[2];
                        ar[k,l].o->o16[l].i[k+1];
                  end;
         end;
      n:=1;
      for k:=0 to 3
      do begin
               a[k]->y[k].i;
               l:=0;
               repeat
                     for j:=l to l+n-1
                     do y[k].o->a4[j].i[k+1];
                     l:=l+2*n;
                     for j:=l-n to l-1
                     do a[k]->a4[j].i[k+1];
               until l=16;
               n:=n*2;
               o16[k+1].o->o[k+1];
         end;
end;


type zzzzzboard;
  in reset,clock,switch[1..8];
  out n_stop,light[1..8],display[1..3,0..3];


(* START OF USER PROGRAM *)
```

```
(* END OF USER PROGRAM *)


(*****************************************)
(*                                       *)
(*                CONTEXT                *)
(*   DESCRIBING BREADBOARD FACILITIES    *)
(*                                       *)
(*           To be used with             *)
(*      hardware simulation system       *)
(*                                       *)
(*****************************************)


  type peripherals;
    in clock,n_stop,light[1..8],display[1..3,0..3];
    out reset,cl,switch[1..8];


    primitive lights;
      in light[1..8];
      var i,n[1..8];
    begin
         for i:=1 to 8
         do if light[i]<>n[i]
            then begin
                      reach(11,35+i);
                      write(light[i]:1);
                      n[i]:=light[i];
                 end;
         reach(11,45);
    end;


    primitive displays;
      in display[1..3,0..3];
      var g,i,j,s,n[1..3],two[0..3];
    begin
         if g=0
         then begin
                   two[0]:=1;
                   for i:=1 to 3
                   do two[i]:=2*two[i-1];
                   g:=1;
              end;
         for i:=1 to 3
         do begin
                 s:=0;
                 for j:=0 to 3
                 do if display[i,j]
                    then s:=s+two[j];
                 if s<>n[i]
```

```
            then begin
                      reach(12,8+3*i);
                      write(s:3);
                      n[i]:=s;
                 end;
        end;
    reach(12,19);
end;


primitive switches;
  in clock,n_stop;
  out reset,cl,switch[1..8];
  var i,m,p,r,s,ten[1..8];
begin
    if s=3
    then begin
           if r
           then if n_stop
                then p:=2
                else p:=4
           else p:=4;
           if clock=0
           then begin
                     cl:=0;
                     s:=p;
                end;
        end
    else if s=2
         then begin
                if clock
                then begin
                          cl:=1;
                          s:=3;
                     end;
           end
         else if s=4
              then begin
                        if clock
                        then
                          begin
                            repeat
                              reach(14,34:10);
                              read(m);
                              for i:=1 to 8
                              do if m>=ten[i]
                                 then begin
                                        switch[i]:=1;
                                        m:=m-ten[i];
                                      end
                                 else switch[i]:=0;
                            until m=0;
                            s:=5;
                          end;
```

```
                    end
               else if s=5
                    then begin
                              if clock=0
                              then begin
                                        reach(13,64:3);
                                        read(r);
                                        s:=2;
                                   end;
                         end
                    else if s
                         then begin
                                   if clock=0
                                   then begin
                                             reset:=0;
                                             s:=4;
                                        end;
                              end
                         else begin
                                ten[8]:=1;
                                for i:=7 step -1 to 1
                                do ten[i]:=ten[i+1]*10;
                                reset:=1;
                                s:=1;
                              end;
  end;


  comp l:lights;
       d:displays;
       s:switches;
  var i,j;

begin
     reach(8,37);
     write('LIGHTS');
     reach(9,36);
     write('12345678');
     reach(10,36);
     write('--------');
     reach(15,36);
     write('--------');
     reach(16,36);
     write('12345678');
     reach(17,36);
     write('SWITCHES');
     reach(9,13);
     write('DISPLAY');
     reach(10,13);
     write('1  2  3');
     reach(11,12);
     write('--------                    ');
     write('                          RUN');
     reach(12,12);
```

```
        write('        ');
        reach(13,1);
        write('                                        ');
        write('                                        ');
        reach(14,1);
        write('                                          ');
        clock->s.clock;
        for i:=1 to 8
        do begin
                light[i]->l.light[i];
                s.switch[i]->switch[i];
           end;
        for i:=1 to 3
        do for j:=0 to 3
           do display[i,j]->d.display[i,j];
        s.reset->reset;
        s.cl->cl;
        n_stop->s.n_stop;
  end;


  comp breadboard:zzzzzboard;
       io:peripherals;
  var i,j;

begin
     clock->io.clock;
     io.reset->breadboard.reset;
     io.cl->breadboard.clock;
     for i:=1 to 8
     do begin
             io.switch[i]->breadboard.switch[i];
             breadboard.light[i]->io.light[i];
        end;
     for i:=1 to 3
     do for j:=0 to 3
        do breadboard.display[i,j]->io.display[i,j];
     breadboard.n_stop->io.n_stop;
end;
```