SLIM: A Language for Microcode Description and Simulation in VLSI[1]

John Hennessy
Computer Systems Laboratory
Stanford University

Abstract

SLIM (Stanford Language for Implementing Microcode) is a programming language based system for specifying and simulating microcode in a VLSI chip. The language is oriented towards PLA implementations of microcoded machines using either a microprogram counter or a finite state machine. The system supports simulation of the microcode and will drive a PLA layout program to automatically create the PLA.

1 Introduction

VLSI chip design has rapidly become an area of great importance and interest. Mead and Conway [6] have proposed a design methodology for VLSI systems that has been widely employed. Their design methodology proposes a chip organization using: finite state control implemented with a PLA, functional units controlled by the PLA, and a set of data paths. This design methodology has been used in a number of large chip designs [5, 1, 2]. The finite state control can be thought of as microcode. Within a design that follows the microcoded control approach, designing and debugging the microcode appears to constitute a significant portion of the work involved in the design process [1, 2]. This paper describes a language for synthesizing the control units of a chip from a high level language description.

Presently few tools exist to assist the user in designing and debugging the microcoded control. Programs to construct PLA's from boolean equation are widespread; however, the difficult component of control unit design is to specify and debug the microcode. This difficulty arises in generating the boolean equations that describe the finite state machine control. Transforming these

---

[1]This research was partially supported by the Join Services Electronics Program under contract # DAAG29-79-C-0047 and the Defense Advanced Research Projects Agency under contract # NDA903-79-C-0680.

to a PLA layout is tedious and error-prone but mechanically straightforward. Some work has been done on describing PLA's at a higher level [7] and on synthesizing PLA descriptions from low level state machine descriptions in DDL [4].

SLIM (Stanford Language for Implementing Microcode) is a programming language useful for the design of a microcoded system that will employ PLA implementation techniques. Unlike earlier work SLIM is functionally oriented. Control in SLIM is based on a finite state machine, but SLIM deals with objects that can be more abstract than the actual PLA inputs and outputs. The SLIM system supports both microcode simulation and automatic synthesis of the microcoded control function either in ROM or PLA. SLIM will also accommodate either finite state machine control or control with a program counter.

Correct microprograms are both tedious and difficult to write for several reasons. First, the programming language is extremely low level. Typically, the designer must deal with a primitive finite state machine without the benefit of a human-engineered interface. Secondly, many of the microprograms are large. This leads to a relatively complex program without a great deal of structure; this is especially true if the finite state machine is coded as boolean equations. A boolean equation approach makes it difficult to consider altering the microcode, even during the debugging process.

Another major difficulty is the significant level of detail that must be expressed. This leads to one of two pitfalls: either the microcode description is very low level and cluttered with details, which makes it impossible to understand; or the designer uses an ad hoc higher level description of the microcode. An ad hoc description is unsuitable because the translation to the low level microcode must be done by hand, and the description tends to be too informal and vague. Without a higher level standard representation, microcode programs are difficult to write correctly and virtually impossible to understand. The SLIM system is also able to translate the boolean equation representation of the PLA into a layout.

We can summarize the goals of SLIM as

- a symbolic higher level language suitable for designing and documenting the microprogram and oriented towards implementation with PLA technology,
- simulation tools to debug the microcode,
- automatic layout of the PLA based on the microcode.

The SLIM design goals spawn a set of language and system requirements. The microcode simulation requirement implies the ability to describe the subsystems that interact with the microcontroller; we will refer to these subsystems as the *environment*. Describing the environment can be easily done in a conventional programming language, if the interaction with the microcode occurs in a restricted and well defined manner. Separating the micromachine description from the environment description has two benefits. The separation increases comprehensibility of the micromachine structure. A specialized language is also more appropriate for the microprogram design; without the separation the translation process is difficult or impossible.

The environment of the finite state micromachine can be described in a conventional programming language. The environment consists of data structures and variables which can be used to simulate the structure of the subsystems. The environment/controller interface is based on a set of functions and procedures. The functions, which must be type boolean, correspond to the inputs to the microcode machine, while the procedures correspond to outputs. We have chosen Pascal to represent the environment. The Pascal data structures provide additional support in describing the functional components. The wide variety of data types coupled with strong type checking also provides support for checking the microcode and making design restrictions explicit in the SLIM program.

Since the end product of SLIM program is a finite state machine implemented with PLA techniques, a SLIM program must incorporate details about the implementation. This specification should include: mappings between functions and procedures in the environment, actual PLA inputs and outputs, and timing specifications that force outputs to occur earlier or later than they occur in the program. Including these details separately allows a more functional orientation in the microcode description. Lastly, details concerning the actual PLA layout are needed, e.g. the number of PLA's and the positioning on the PLA of each signal.

2 Specifying the microcode

A SLIM program consists of a finite state machine. Each state in the SLIM program contains a series of conditional actions that may cause one or more outputs to be high, or may specify the next state. The next state may be specified by default or explicitly. The outputs associated with a given state are conditional on a set of product terms only. Although arbitrary boolean expressions could be used, SLIM does not because it requires a significant amount of processing to transform the expressions to a PLA oriented sum of products form. In this process the number of product terms

added to the PLA may be substantial (up to 2^n terms for an expression of length n). The property that – the number of product terms in the PLA is approximately equal to the number of preconditions for the outputs in a SLIM program – has been useful in estimating the PLA size.

There are two major schemes for implementing the state component of a finite state machine. A standard finite state implementation uses a fixed state assignment and includes an encoding of the next-state function in the PLA. An alternative implementation uses a microprogram counter that is incremented under external control. Each approach has benefits that depend on the the micro-program being implemented. The tradeoffs and the advantages of the two different VLSI control implementations are discussed in [3]. SLIM supports both control implementations, provides default next states for program counter implementations, and will support subroutines with call and return in either case.

A SLIM microprogram consists of a set of states listed sequentially. Each state may optionally have a label, which denotes the state name. The specification of the first state is preceded by a set of specifications for outputs which are state independent. Figure 1 shows the format of a the state machine specification.

```
fsm
state-specification (for state independent outputs)
state-name (optional) : [state-specification]
. . . . . .
state-name (optional) : [state-specification] .
```

Figure 1: Specifying the state machine

A state specification is a list whose elements are either unconditional actions or conditional commands. A conditional command consists of a condition and a list of actions. A condition consists of a list of one or more product terms that are joined with **or**, and a product term is a series of predicates joined with **and**. A predicate must be a call to a function in the environment; predicates correspond to one or more PLA inputs. The interpretation of the command is: if the entire condition evaluates to true, then the actions should be executed. If there are no predicates, the condition is assumed to be true and the action is always executed in that state. The form of a state specification is given in Figure 2.

```
if p1 and ..... and pn or q1..... or qm=>  action

Where the pi are function invocations and the qj are product terms,
like the first term.
```

Figure 2: State specifications

Each state may contain a list of such specifications and the entire state is bracketed. During simulation, state specifications are evaluated and executed sequentially, but in the actual PLA implementation these operations will occur in parallel. Therefore, side effects between procedures that are outputs and functions that are inputs in the same state should be employed with great care.

2.1 Actions

There are two types of actions allowed: outputs and state change operations. A list of actions can be used as a single compound action by bracketing the list. Outputs are invocations of procedures in the environment and correspond to PLA outputs. The state change directives dictate the next state. All state change directives have effect only after the current state is completed; thus, all state specifications with true conditions will be executed in a state. The state change directives are:

next *state-name* - makes *state-name* the next state.

call *state-name* - does a microcode call to the routine at *state-name*.

return- returns to the state sequentially following the calling state.

2.2 A short example

Figure 3 shows the finite state machine controller for the traffic light example from [6]. (The entire example is given in the appendix.) The state independent component is for simulation purposes. The procedures Farmlight and Highlight alter the color (which is a parameter) of the traffic light at the farmroad and the highway. Timeout looks for the timeout condition, which is either short or long as dictated by the parameter. The function Cars corresponds to the test for a car.

Figure 3: Microcode specification for the Mead/Conway traffic controller

```
fsm
   [ getinput; timer ] { state independent component }
highgrn: [ highlight(green); farmlight(red); {Highway green and farmroad red}
           if notcars or nottimeout(long) => next highgrn;
           if cars and timeout(long) => [ starttimer; next highyel ] ]
highyel: [ highlight(yellow); farmlight(red); {Highway yellow and farmroad red}
           if nottimeout(short) => next highyel;
           if timeout(short) => [ starttimer; next farmgrn ] ]
farmgrn: [ highlight(red); farmlight(green); {Highway red and farmroad green}
           if cars and nottimeout(long) => next farmgrn;
           if notcars or timeout(long) => [ starttimer; next farmyel ] ]
farmyel: [ highlight(red); farmlight(yellow); {Highway red and farmroad yellow}
           if nottimeout(short) => next farmyel;
           if timeout(short) => [ starttimer; next highgrn ] ].
```

3 Defining the Relationship to the PLA

The relationship between the microcode specification of the control program and the PLA is defined by: declaring the input and output signals for the PLA and defining the mappings between environment functions/procedures and input/output signals. The expressive power of this mapping is one of the advantages of SLIM.

3.1 Defining input and output signals

PLA signals are defined by means of input and output signal declarations, which appear just before the definition of the environment procedures. Signal declarations begin with the keyword **inputs** or **outputs**, as appropriate. The general form of each declaration is then:

{name ['(' bounds ')']} [':' parameters] ';'

The list of names are the names of input or output signals being declared. The optional bounds designator indicates whether a particular signal is a single bit or a vector of bits. In the latter case the line can be treated as an integer-encoded number; the order of the bounds (low to high or high to low) specifies the order of the lines in the signal vector. If any optional parameters appear they are associated with all input/output names in the declaration. Table 1 defines the legal parameters.

Table 1: Signal parameters

{Syntax	Meaning	For input/output}
pla (*n*)	Associate signal with pla # *n*	both
top	Position signal on top of pla	both
bottom	Position signal on bottom of pla	both
renames (*id*)	Give the signal *id* another name	both
earlier (*n*)	Move the signal n states earlier	output
later (*n*)	Move signal *n* states later	output

A signal declaration specifies physical placement information using the directives **top** and **bottom**. The order of the signals on the PLA is given by the order of their declaration. The state signals are added by SLIM and appear last in the PLA inputs and first in the outputs; this facilitates interconnection. When more than a single PLA is specified SLIM determines which outputs should appear from which PLA's (by declaration or default to PLA 1). Only the necessary inputs are generated for each PLA; these are based on the outputs that are specified in that PLA.

The optional pipelined directives, i.e. **earlier** and **later**, move an output signal forward or backward in the state graph. This is very useful when a particular signal, which is logically associated

with a single operation, must occur earlier. A frequently occurring example of this is precharging or enabling of alu's. Although the functional operation *add* appears to occur in a single state the alu must be precharged/enabled one state earlier. The pipelined directives provide a convienent way to express such relationships without adding needless details to the microcode description. If an output signal *x* appears in a state *s* conditional on input *c* and *x* is pipelined **earlier**(*i*), then the output *x* will appear, conditional on *c*, in all the states that precede *s* by *i* states. Although pipelining can be done into both predecessor and successor states, by far the most common situation is pipelining into the immediate successor state. SLIM finds all predecessor or successor states, including those that occur when the state that is pipelined from is the target of a branch or call. Pipelining is not permitted across a procedure return, i.e. in the state following a call. The **renames** directive gives a signal another name, without associating the other characteristics (e.g. pipelining) of the renamed signal. This is useful if a particular signal must be pipelined nearly all the time, but occasionally nonpipelined generation of the signal is needed.

3.2 Describing the relationship between environment and outputs

Since a procedure or function in the environment can logically correspond to one or more signals, SLIM provides a method of defining the mapping between environment routines and signals. This method allows the microcode description to be functionally oriented, and to significantly decrease the amount of code needed to describe the PLA implementation of the microcode.

The mapping between environment procedures and signals to be generated in the PLA is given in the definition section of an environment procedure or function. The definition section starts with the keyword **definition** and appears immediately after the function or procedure header. Procedures in the environment without a definition section are presumed to be for simulation purposes only. The definition section consists of a list of signal definitions which are separated by semicolons; the definition section is terminated by **end**.

A signal definition has the form:

[*pattern-string* :] *signal-expression*

The optional *pattern-string* is used to specify different signal combinations based on the values of the parameters to the environment procedure. The *pattern-string* consists of a list of string patterns separated by commas and enclosed in parenthesis. If the pattern list matches the list of actual parameters in a call to this procedure, then the signals in the signal list are generated as outputs. Each string pattern can either be a alphanumeric string or a "*". The latter is a wild card match, indicating that any actual parameter value should generate a match for the corresponding parameter.

The *signal-expression* specifies what signals to generate; it may also contain invocation of other environment procedures. Before it is evaluated any identifiers in the signal-list that correspond to formal parameters are replaced by the actual parameter values in the call for which signals are being generated. The types of signal expressions are defined in Table 1.

Signal-expression	Meaning
signal name	emit the signal
procedure-name(parameters)	emit the signals for the named procedure
signal-expression and *signal-expression*	emit both sets of *signal-expressions*
$expr_1$ & $expr_2$	Emit $expr_2$ concatenated to $expr_1$
signal name = integer constant	emit encoded constant to the signal vector
not *signal-expression*	emit inverse of a simple *signal-expression*
signal-name[constant]	emit a single signal within a signal vector

Table 2: *Signal-expressions*

If the signal identifier is an environment procedure and not an signal name, the definition section of the referenced environment procedure is used for that signal. Naturally, the procedure name can be followed by parameter strings. This facility allows multi-level environment procedures to produce signals by composing the definition list in each procedure.

In Figure 4 some input/output declarations and two of the procedures from the Mead/Conway traffic light example are given. The highway traffic light is encoded as a two-element vector; the input testing for cars is a single bit. Note that PLA signals may have the same name as components of the Pascal program.

Figure 4: An example from the Mead/Conway Traffic Controller

```
type    colortype = (green,yellow,red);

inputs      c: bottom;
outputs     hl[1..0] : bottom;

procedure highlight(color: colortype);
definition
        (green): hl = 0 ;
        (yellow): hl = 1 ;
        (red): hl = 2 ;
begin       hl := color end;

function cars :boolean;
definition c;
begin       cars := (c = 1) end;
```

4 Using SLIM

A SLIM program can be used to drive a microcode simulation as well as generate a PLA layout. A SLIM simulation requires a microcode description with all of the environment procedures and functions. The simulation is presently done by creating a Pascal program which embodies the semantics of the microcode. A SLIM simulation can be requested with state tracing.

PLA generation is a straightforward process, which is done in two parts. The first part analyzes the microcode structure and creates product term lists for each output. The effect of signal definition and pipelining is integrated before making these lists. The PLA layout is then done by a separate program which inputs the signal descriptions and the product term lists. The intermediate form uses boolean expressions; this allows the use of any PLA generator that accepts boolean equations as input and the use of PLA optimizers prior to layout.

Another program in the SLIM system can be used to assist in choosing a state encoding (applicable only for finite state implementations). The program acceptances output from SLIM with the state entries unencoded. It computes a matrix whose *i,j* entry is the saving in product term count that will result if states *i* and *j* are encoded so that they can be uniquely distinguished from all other states with a single product term.

4.1 Ensuring microcode correctness

There are several useful types of debugging and checking of microcode that can be done in the process of simulation. Most important among these are detecting potential errors which arise because the simulation does not exactly match the PLA implementation, or because the microcode does not employ the environment in a manner that the hardware is designed to support. Another class of errors may arise because the simulation may fail to test all possible combinations of inputs or fail to test all states.

The major reasons that the simulation and PLA implementation might behave differently is because the simulation treats outputs, environment procedures, and the state as unique entities in a sequential manner. In the PLA these objects are interrelated. Problems such as assigning two next states are resolved into a single, well defined action in the simulation, but these actions result in a disaster in the PLA implementation, since both sets of state bits are set high. Certain classes of these errors can be caught by predefined, microcode independent methods, but others require a more general scheme, which we can also employ to find errors concerning the use of the hardware environment by the

microcode. SLIM checks for common sorts of errors, such as failing to assign a next-state in a finite state machine implementation, or attempting to assign more than one next-state.

Many of the hardware/microcode inconsistencies arise from situations where certain outputs are being incorrectly used, perhaps with respect to timing, or the hardware is being instructed to preform some task it is not physically able to undertake. Many of the latter types of errors can be caught using a strictly type-checked environment specification. For example, suppose that the register file on some microcoded processor is divided into two sections in such a way that two registers from the same section can not be gated to the alu (many hardware micromachines have this property). Microcode errors that arise because two registers from the same section are being sent to the alu can be detected by defining the machine structure with two different types for the registers and specifying that the alu environment procedures have two parameters — one from each register section. This class of simple errors is detected at compile-time.

A more complex class of errors can not be detected with a straightforward compile-time scheme. Some examples of this type of error are: attempts to use the bus for two different quantities in the same time frame, overlapping use of environment hardware (such as an alu), and incorrect timing of an output in a state. Many of thes errors can be detected during simulation using a set of assertions, which can be checked during simulation. We divide these assertions into two groups: invariant assertions and state dependent assertions. The invariant assertions specify conditions which must hold regardless of the current state, e.g. if an alu output occurs in this state, the alu was precharged in the previous state and was not doing any other operation. State dependent assertions specify properties which should hold at a particular state, e.g. a certain part of the machine should have a certain value.

In SLIM anywhere an action can occur, an assertion can be specified. Although the assertion generates code for simulation purposes, no PLA entries are affected or generated. Assertions are only used to ensure that certain properties hold. An assertion has the form **assert** invocation, where invocation must be the invocation of a boolean function. Whenever execution reaches an assert statement at simulation time, the simulation invokes the specified function. If the function returns false the simulation is halted with an appropriate error message.

In using SLIM, we have found that the expressive power of SLIM's pipelining and signal definitions is one of its major advantages. However, the mechanism can also lead to errors, since the specifications are not reflected in the simulation. To assist in ensuring that the signal specifications in

a SLIM program are consistent and correct, two types of output-generation checking are supported. Pipeline checking will cause a warning to be generated whenever a signal component both occurs in a state and is pipelined into that state from another state. This appears to catch most errors in the use of pipelining. Another powerful check is examining sets of mutually exclusive signals. A SLIM program can specify one or more *exclusive sets*. SLIM will check that no two signals in the same exclusive set can be generated in the same state.

5 Current status and concluding remarks

This paper describes SLIM, a language and processing system for describing microcode whose implementation orientation is PLA based. The purposes of this language are: to document the microcode at a reasonable, logical level while providing a firm specification; to allow extensive simulation, debugging, and error detection; and to automatically create the PLA layout necessary to implement the microcode description.

SLIM has been working for approximately one year. It is coded in standard Pascal. To date, experience with SLIM has been highly favorable. It has been used in the development of two large chip designs [1, 2], both of these contain extensive microcoding. It has also been used in a number of smaller projects with favorable results.

The most significant observation we have made in using SLIM is the enormous significance of the control function and its design. For large projects, we have found that 60-75% of the design time is spent in constructing and debugging the control as specified by SLIM. A large amount of this time is spent is constructing an accurate functional specification of the data components as a SLIM environment. In many instances, the construction of SLIM environment has uncovered bugs in the data components being described. The specification of the control program itself is also time consuming especially in the debugging process.

There are many interesting questions concerning the applicability of SLIM that have not been investigated. It would be interesting to examine the use of SLIM for microcode machines whose architecture is not strictly PLA based, but whose microcontrol is straightforward. We are also interested in supporting a wide variety of PLA implementations and in PLA optimization.

Appendix 1. Annotated Syntax of SLIM

This is the syntax for the non-Pascal portion of SLIM. Nonterminal symbols appear to the left of =; terminal symbols in the grammar are distinguished by being in quotes. The metasyntax $[\alpha]$ means that the string α is optional, and $\{\alpha\}$ means that the string α may be repeated zero or more times. Comments can appear at the end of a production and are started with --.

```
Program = 'program' '<id>' Programparms ';' Outerblock
Outerblock = Constpart Typedefpart Vardeclpart Iopart Procpart Fsm -- A Pascal program with a fsm body
Procheading = 'procedure' '<id>' Formalparms ';' Definitionpart -- Procedures contain definitions
Funcheading = 'function' '<id>' Formalparms ':' '<id>' ';' Definitionpart
Iopart = ['inputs' Spec { ';' Spec}] ['outputs' Spec { ';' Spec}] -- Input/output declarations
Spec = Vector { ',' Vector} [':' Parameter {Parameter} ';'] -- An input/output vector
Vector = '<id>' [ '[' '<int>' ':' '<int>' ']' ] -- Vector has integer bounds
Parameter = 'pla' '(' '<int>' ')' -- PLA number
      = 'top' -- Top of PLA
      = 'bottom' -- Bottom of PLA
      = 'earlier' '(' '<int>' ')' -- Pipeline into earlier states
      = 'later' '(' '<int>' ')' -- Pipeline into later states
      = 'renames' '(' '<id>' ')' -- Rename a signal (without pipelining)
Definitionpart = [ 'definition' Definition {Definition} ]
Definition = [ '(' Patternlist ')' ':' ] Output { 'and' Output} ';' -- Definition is a series of pattern lists
Patternlist = Pattern {',' Pattern} -- Each pattern list must match the parameter list
Pattern = '*' -- Wild card match
     = '<id>' -- Name match
Output = [ 'not' ] Plainoutput -- Outputs can be inverted
Plainoutput = Invocation [ '&' Output] -- Outputs can be composed by concatenation
       = '<id>' '=' Constant -- A vector can output an encoded integer
       = '<id>' '[' '<int>' ']' -- A single line from a vector can be made high
Fsm = 'fsm' Stateindpart {State} '.' -- The FSM contains a state independent part and a list of states
Stateindpart = '[' Statespecifiers ']'
State = [ '<id>' ':' ] '[' Statespecifiers ']' -- States are optionally labelled
Statespecifiers = Statespec { ';' Statespec}
Statespec = [ 'if' Cond { 'or' Cond } '=>' Action ] -- A state is conditional on a sum of product terms
Cond = {Invocation 'and' } Invocation --Form of a product term, the invocations are functions
Invocation = '<id>' [ '(' Constant {',' Constant} ')' ] -- Limited
function invocation, constant can be a variable
Action = '[' Action { ';' Action} ']' -- Composite action
    = 'assert' invocation -- Assert action
    = Invocation -- Procedure invocation
    = 'next' '<id>' -- Goto specified State
    = 'call' '<id>' -- A microcode subroutine call
    = 'return' -- A microcode subroutine return
```

Appendix 2. More Examples

The Full Traffic Controller from Mead/Conway

```
program traffic(input,output);
const   short = 2;          long  = 4;
type    colortype = (green,yellow,red);
         signaltype = 0..1;
var     time: integer;          hl,fl: colortype;          c: signaltype;
inputs        c,tl,ts  : bottom;
outputs        st,hl[1..0],fl[1..0]  :bottom;
procedure getinput;  { for simulation purposes only }
begin       write('cars? ');read(c); end;
procedure timer;   { for simulation purposes only }
begin       if time < long then time := time + 1 end;
procedure highlight(color: colortype);
definition
        (green): hl = 0 ;
        (yellow): hl = 1 ;
        (red): hl = 2 ;
begin       hl := color end;
procedure farmlight(color: colortype);
definition
        (green): fl = 0 ;
        (yellow): fl = 1 ;
        (red): fl = 2 ;
begin       fl := color end;
procedure starttimer;
definition st;
begin       time := 0 end;
function cars :boolean;
definition c;
begin       cars := (c = 1) end;
function notcars :boolean;
definition not c;
begin       notcars := not cars end;
function timeout(length: integer) :boolean;
definition
        (long): tl ;
        (short): ts ;
begin       timeout := (time >= length) end;
function nottimeout(length: integer) :boolean;
definition
        (long): not tl ;
        (short): not ts ;
begin       nottimeout := not timeout(length)end;

fsm
   [ getinput; timer ] { state independent component }
highgrn: [ highlight(green); farmlight(red);
           if notcars or nottimeout(long) => next highgrn;
           if cars and timeout(long) => [ starttimer; next highyel ] ]
highyel: [ highlight(yellow); farmlight(red);
           if nottimeout(short) => next highyel;
           if timeout(short) => [ starttimer; next farmgrn ] ]
farmgrn: [ highlight(red); farmlight(green);
           if cars and nottimeout(long) => next farmgrn;
           if notcars or timeout(long) => [ starttimer; next farmyel ] ]
farmyel: [ highlight(red); farmlight(yellow);
           if nottimeout(short) => next farmyel;
           if timeout(short) => [ starttimer; next highgrn ] ].
```

Example — Computing GCD

```
program test (input,output);
var x,y: integer;
inputs
        eq1,eq0,gtx, gty: bottom;
outputs
        aluop[1..2] : bottom ;
        enablex,enabley: top earlier (1);
procedure init;
begin       read(x); read(y);       end;
procedure subt (var a,b: integer);
definition
         enable & a and enable & b and aluop = 1;
begin  a := a-b  end;
function greater (x,y:integer): boolean;
definition
         gt & x ;
begin greater := x>y  end;
function equal (x,y:integer): boolean;
definition  eq & y;
begin eq := x = y;  end;
function ne(x,y:integer): boolean;
definition  not equal (x,y);
begin ne := not equal(x,y); end;

fsm
        [;]
one :   [ init ;
          assert ne(y,0);
           if equal(x,0) => next endstate ]
        [ call two ]
        [ next one ]
two:    [ if greater(x,y) =>  [subt (x,y); next two ];
              if greater(y,x) =>  [subt (y,x); next two ]]
three:  [ assert equal(x,y);
              if equal(x,1) => [writeln(1); return];
              if ne(x,1) => [writeln(y); return]]
endstate: [ halt ] .
```

References

1. Clark, J.H. "A VLSI Geometry Processor for Graphics." *Computer 13*, 7 (July 1980), 59-68.

2. Clark, J.H. and Hannah, M.R. "Distributed Processing in a High-Performance Smart Image Memory." *Lambda 1*, 3 (1980), 40-45.

3. Clark, J.H., Hennessy, J.L., Hannah M.R. A comparasion of two different VLSI control structures. Computer Systems Laboratory, Stanford University, Dec, 1980.

4. Duley, J.R. and Dietmeyer, D.L. "Translation of DDL digital system specification to Boolean equations." *IEEE Trans. Computers c-18*, 4 (Apr 1969), 305-313.

5. Holloway J., Steele, G., Sussman, G., Bell, A. The Scheme-79 Chip. Tech. Rept. 599, Artificial Intelligence Laboratory, MIT, Jan, 1980.

6. Mead, C. and Conway, L.. *Introduction to VLSI Systems.* Addison-Wesley, Menlo Park, Ca., 1980.

7. Weber, H. High Level Design for Programmed Logic Arrays. Proceedings of Fourth Conf. on Computer Hardware Description Languages, May, 1979, pp. 96-101.