COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

A LANGUAGE FOR MICROCODE DESCRIPTION AND SIMULATION IN VLSI

John Hennessy

TECHNICAL REPORT NO. 193

July 1980

This research was partially supported by the Joint Services, Electronics Program under contract no.
DAAG29-79-C-0047

A Language for Microcode Description and Simulation in VLSI[†]

John Hennessy
Computer Systems Laboratory
Stanford University
Stanford, Ca. 94305

Technical Report 193

July 1980

Abstract

This paper presents 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.

Key Words and Phrases: microcode, VLSI, PLA, finite state machine, design automation, computer-aided design, simulation, layout.

[†] This research was partially supported by the Joint Services Electronics Program under contract no. DAAG29-79-C-0047.

1. Introduction

VLSI chip design has rapidly become an area of great importance and interest. Presently few tools exist to assist the user in designing and debugging a VLSI chip. This paper discusses a programming language useful for the design of a microcode controller which will employ PLA implementation techniques. Microcoded engines, implemented either with a finite state machine technique or a microprogram counter, frequently form the control unit on a VLSI chip and thus constitute a signifcant portion of the work involved in the design.

Design of a microcode controller is an error prone task for several reasons. First, the programming language is extremely low level. The designer must deal with a binary 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.

Another major difficulty is the significant level of detail which must be expressed. This leads to one of two pitfalls: either the microcode description is very low level and cluttered with details, which make it impossible to understand; or the designer uses an ad hoc higher level description of the microcode. This leads to problems, since the translation to the low level microcode must be done by hand, and the ad hoc description tends to be too informal and vague. Hence, without a higher level standard representation microcode programs are difficult to correctly write and virtually impossible to understand.

For microcode machines which will be implemented with PLA's, the process of laying out the PLA is both time consuming and tedious, leading to unnecessary errors. However, given the description of the PLA terms, in an equation or state table form, the process is mechanically straightforward. Therefore, such a system should provide for PLA layout as well as microcode design.

As a solution to these problems we propose a higher level language which has the following goals:

1. Provide a symbolic higher level language suitable for designing and documenting the microprogram and oriented towards implementation with PLA technology.

2. Provide simulation tools to debug the microcode.

3. Provide for automatic layout of the PLA based on the microprogram.

2. Requirements

The goals for this design aid lead to language requirements. The desire to simulate the microcode leads to the requirement to be able to describe the subsystems which interact with the microcontroller. We will refer to these subsystems as the *enviroment*. Describing the environment can be easily done in a conventional programming language, whose interaction with the microcode occurs in a restricted and well defined manner.

The micromachine description must be separate from the environment description. There are several reasons for this:

- it increases comprehensibility of the micromachine structure;
- a specialized language is more appropiate for the microprogram design, since in general it is impossible to determine if a given program describes a finite state machine;
- it simplifies the translation process.

Since the end product of this design is a finite state machine implemented with PLA techniques, a method of incorporating details about the implementation is required. This specification should include mappings between functional specifications in the environment and actual PLA outputs, as well as timing specifications which 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. These include such things as the number of PLA's and the positioning on the PLA of each signal.

3. Environment Definition

The environment of the finite state micromachine can be described in a conventional programming language; it must interface with the micromachine specification both for simulation and generation of the PLA. The environment will consist of data structures and variables which can be used to simulate the structure of the subsystems. The enviroment/controller interface is based on a set of functions and procedures. The functions, which must be type boolean, correspond to the inputs to the machine, while the procedures correspond to output signals.

Thus a conventional programming language can represent the environment as well as provide the necessary interface. We have chosen Pascal as the language for this purpose.

4. Control in the Micromachine

Control of program flow in a finite state machine uses a next-state function. Thus in the micromachine the next-state description should be included. Since the next state for most states in the machine is the sucessive sequential state, to have the designer specify the next state in these situations is wearisome. Therefore, support for a default next state is needed even though it may require extra work in implementation or add extra terms to the PLA.

The outputs associated with a given state should be conditional on a set of product terms only. Although arbitrary boolean expressions could be used, we choose not to 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 - is very useful.

There are at least two major schemes for implementing a finite state machine which must be considered: a standard finite state implementation with a fixed state assignment, and an implementation based on a microcode engine with a microprogram counter. In the next section we discuss the approaches and the required control mechanisms for each.

5. FSM and Program Counter Implcmentstions

The following table summarizes the advantages of the two approaches:

FSM (state assignment)	Microcoded (program counter)
Subroutines hard	Subroutines easy
Need next state terms in PLA	PC incremented by counter
Minimize states	Fixed assignment

It is clear the each approach has advantages and the language should support both approaches. In either case a method for explicitly specifying the next state is required. Additionally, in the microcoded approach a method for implementing subroutines with call and return is needed.

6. Defining the Micromachine

The micromachine is defined as a set of states, which are listed scqucntially. Each state may optionally have a label, which denotes the state name. The state specifications are prcccdcd by a set of specifications for outputs which arc state indcpendcnt. Thus the micromachinc specification looks like:

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

state-name (optional) : [state-specification] .

A state specification is a list whose elements arc either unconditional actions or conditional commands which arc similar to guarded commands. A conditional command consists of a guard and a list of actions. A guard consists of a list of one or more product terms with arc joined with or. A product term is a scrics of predicates joined with and. Thc predicatc must be a call to a function in thc environment; it will be associated with a PLA input. The interpretation of the command is: if all the predicates arc true the actions should be executed. If there are no prcdicates, the guard is assumed to be true and the action is always executed in that state. Thus the form of a state spccification is:

action

if p_1 and . . . . . and p_n or . . . . . or q_1 and . . . . . and q_m => action

wh^re the pi are function invocations and the q_i are product terms, like the first term.

Each state may contain a list of such specifications; the entirc state is bracketed. During simulation thc specifications are evaluated and executed sequentially; howevcr in the actual PLA implementation these operations will occur in parallel. Thcrcfore, side effccts betwcen procedures which are outputs and functions which are inputs in the same state should be cmployed with great carc.

6. I Actions

There arc two types of actions allowed: outputs and statc changc opcrations. A list of actions can be used as a single action by bracketing thcm. Outputs are invocations of procedures in the environment and correspond to PLA outputs. The statc changc directives dictate the next state: all state change directives have effect only after the current state is complctcd. That is, all state spccifcations with true guards will be executcd in a state, regardless of any state change dircctives in that state. The state change directives are:

next *stn te-name* - makes *state-name* thc next state

call *state-name* - docs a procedurc call to the routine at *statc-name*

return - rcturns to' the state scqucntially following thc calling statc

6.2 A short example

```
{ Multiplication of 10 bit unsigned integers }
program multiply;
const  maxint = 1048576: ( i.e. 2^20 }
       one_half = 524288;
```

```
type      register = O..maxint;
{ z ← x * y }
var       x, y, z : register;
          counter : 0..10; { count iterations }
procedure ZeroZ;
begin     Z : = 0 end;
procedure GetXY;
begin     { Read in X and Y, for simulation only }    read (x); read (y); end;
procedure ZeroCounter;
begin     Counter : = 0         end;
procedure SHL (r: register);
begin     if    r > one-half then     r : = r - one-half;    r := r * 2 end;
function MSBY: boolean;
begin     MSBY : = y > one-half end;
procedure AddXtoZ;
begin     Z := Z + X      end;
function LT: boolean;
begin LT : = counter < 1    end;
procedure Incr;
begin     counter : = succ(counter)          end;

fsm { Definition of the machine }
    [ { No state independent actions } ; ]
Start:    { Start state: Z, Counter ← 0; get simulation values for X, Y }
    [ZeroZ;            GetXY;            Zerocounter]
Loop: { Main multiplication loop }
    [SHL (z)]
    [if MSBY =>     AddXtoZ]
    [SHL (y)]
    [if LT = > Incr; next Loop   { End multiplication loop)] .
```

7. Defining the Relationship to the PLA

The relationship between the microcode specification of the control program and the PLA is defined by the input signals, the output signals, and the mapping between environment procedures and output signals. The following sections deal with these topics.

7.1 Defining input and output signals

PLA signals are defined by means of input and output signal declarations; their relationship to the environment routines is established through signal definitions. Input and output signal declarations appear just before the definition of the environment procedures.

Signal declarations begin with the keyword inputs or outputs, as appropiate. 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 line or a number of lines. In the latter case the line is treated as an integer encoded number. "he optional parameters are associated with all input/output names in the declaration.

There may be zero or more parameters; the following parameters are legal:

syntax	meaning	permitted on 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 (signal-name)	Gives a signal another name	both
earlier (*n*)	Move the signal n states earlier	output
later (*n*)	Move signal n states later	output

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 operation appears to occur in a single state the alu must be enabled one state earlier. The pipelined directives provide a convicnent way to express such relationships without adding needless details to the microcode description. 'The renames directive gives a signal another name, without associating the other characteristics (e.g. pipelining) of the signal.

7.2 Describing the relationship between environment and outputs

Since a procedure of function in the environment can logically correspond to one or more signals, a method of defining the mapping between environment functions/procedures and signals is needed. Allowing this capability has two significant effects: it allows the microcode description to be functionally oriented, and it significantly decreases the amount of code needed to describe the PLA implementation of the microcode.

The mapping between an environment procedures and signals to be generated in the PLA is given in the definition section of an environment procedure. 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-list*

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 set of string patterns separated by comma and enclosed in brackets. The string patterns are used to match the substitution list on a one to one basis. If the pattern matches the list of actual parameters, the signals in the signal list are generated as outputs. Each string pattern can either be a alphanumeric string or a *, the latter indicating that any actual parameter value should generate a match for the corresponding parameter.

The *signal-list* may contain the names of signals which have been declared in the output declaration section and other environment procedures. Each signal expression is a single signal, a conjunction of two signals, an assignment to a signal vector (i.e. a signal with more than one line), or a concatenation of identifiers and constants (which is then treated as a single signal). Concatenation is indicated by the symbol &. There are two fonns that each single signal can have. The first is a single identifier: the second format is that of a procedure invocation. in the second case any parameters in the parameter list which correspond to environment procedure formal parameters are replaced by the actual parameter strings. 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 procedures to produce signals by composing the definition list in each procedure.

7.3 An example

Consider the earlier example of 10 bit multiplication. We will assume that our machine will be controlled by PLA outputs which are defined using *r* as a register (which can have values O-7) as follows:

enabler - put *r* on an alu input bus, this must occur one state before an alu operation

loadr - put the alu output bus into *r*

shiftleft, shiftright - shift left and right

zero - send a zero to the alu

msbr - true if the msb of *r* is on

Itr - true if *r* is less than 1

```
add, incr • alu operations
```

Using these outputs the following declarations would bc added to the example for the procedure SHL:

```
inputs

lt[0..7], msb [0..7]: bottom;

outputs

        enable [0..7] : earlier (1) top:
        load [0..7], shiftleft, shiftright, zero, add, incr : top ;
```

With these output signals declared wc can specify the relationship between the environment procedures and thc signals they should generate.

```
procedure SHL (r: register);
definition
    enable <- r   ; shiftleft  ; load <- r  end;

procedure ZeroZ;
definition
    zero and load <- 2; {z register)

procedure ZeroCounter;
definition
    zero and load <- 3; {counter register)

function   MSBY: boolean:
definition
    msb [1]; { signal bit for Y (register 1) }

procedure AddXtoZ;
definition
    enable <- 2; add; load <- 2; (no signal for enable x since it is 0)

function LT: boolean;
definition
    Lt <- 3; (counter register}

procedure Incr;
```

```
    definition
        enable <- 3; Incr;
```

The appendix contains a more detailed example which requires more elaborate output definition facilities.

8. Using the Language

The description language can be used to drive a microcode simulation as well as generate a PLA layout. In order to do simulation the microcode description is supplied with the signal declarations and definitions. The user may supply start and end states for the finite state machine, if they exist. The simulation is presently done by creating a Pascal program which embodies the semantics of the microcode. Two types of simulation output can be produced:

1. Functional simulation with state traces and outputs of additional information as dictated by the environment procedures.

2. Functional simulation with input/output signal 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. This structure allows the insertion of programs which optimize the finite state machine.

9. Other Uses of the System

There are several other useful types of debugging and checking of microcode which 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 which the hardware is designed to support. Another class of errors arises from the fact that the simulation may fail to test all possible combinations of inputs or fail to test all states.

The major reasons why the simulation and PLA implementation might behave differently arise from the fact that the simulation treats outputs, environment procedures and the state as unique entities. In the PLA these objects are interelated, so problems such as assigning two next states are resolved into a single well defined action in the simulation, but result in a potentially harmful action in the PLA implementation. 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 resolve errors concerning the use of the hardware environment by the microcode.

Many of the errors in the use of the hardware environment by the microcode 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 complete. 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 which 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 of 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. Other potential types of timing errors which arise with regard to pipelining, will generate warnings.

The more complex class of errors can not be detected with a straightforward compile-time scheme. Some examples of the type of errors that may arise 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 with respect to which state an output actually occurs during. These sorts of errors can be detected during simulation if a set of assertions which are checkable during simulation can be specified. 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.

9. *1 Assertions*

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. Thus the usage of assertions is merely 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 appropiate error message.

10. Conclusions

This paper has proposed 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 last to automatically create the PLA layout necessary to implement the microcode description.

There are many interesting questions concerning the applicability of this tool. Although it is designed to support a wide variety of applications there may be unknown but fundamental limitations. It would also be interesting to examine its applicability to microcode machines whose architecture is not PLA based, but whose microcontrol is extremely straightforward; a major example of this implementation strategy is the two level microcode approach used in the Motorola 68000.

Acknowledgements

I am indebted to Jim Clark for providing motivation and insight during this research.

Appendix 1 Syntax of the Microcode Specification Language

This is the syntax for the non-Pascal portion of the language. Terminal symbols in the grammar are distinguished by being in quotes.

```
program
   -> ‘program’ ‘<id>’ program←parms ’;’   outer←block ;
outer+ block
   -> const←part type←def←part var←decl←part io←part proc←part
        fsm ;
proc←heading
   -> ‘procedure’ ‘<id>’ formal←parms ’;’ definition←part => proc;
func←heading
   -> ‘function’ ‘<id>’ formal←parms ’:’ ’<id>’ ’;’ definition←part   => func;
io←part
   -> inputs←designators        outputs←designators ;
inputs←designators -> ;
              -> ‘inputs’ designators => inputs;
outputs ← designators -> ;
              -> ‘outputs’ designators = > outputs;
designators
   -> spec ;
   -> spec designators ;
spec        .
   -> vector←list ’;’ =  >    io←list;
   -> vector←list ’:’ parameters ’;’ => io←list ;
vector←list ->    vector;
              -> vector ’,’ vector←list;
vector -> ‘<id>‘;
              -> ’<id>’ ’[’ ’<int>’ ’:’ ’<int>’ ’]’ => vector;
parameters
   -> parameter ;
   -> parameter parameters ;
parameter
   -> ’pla’   ’(’  ’<int>’  ’)’  = > planumber;
   -> ‘top’  => top;
   -> ‘bottom’  => bottom:
   -> ‘earlier’ ’(’ ’<int>’ ’)’ = > earlier;
   -> ‘later ’(’ ’<int>’ ’)’ => later;
   -> ’renames’ ’(’ ‘<id>’ ’)’ => renames;
definition←part
```

```
    -> 'definition' def←list   => definition;
    -> ;
def←list
    -> definition ;
    -> definition  def←list ;
definition
    -> patternmatch  ':' output←list ';' => defline;
    -> output+list ';' => defline;
patternmatch
    -> '(' pattern←list ')' => patterned;
pattern ←list
    -> pattern ;
    -> pattern ',' pattern←list ;
pattern
    -> '*'  => anything;
    -> '<id>' ;
output←list
    -> output ;
    -> output 'and' output←list => andnode;
output -> plain←output;
            -> 'not' plain+output => notnode ;
plain ← output -> invocation;
            -> invocation '&' output => cat;
            -> '<id>' ' = ' constant => vecout;
            -> '<id>' '[' '<int>' ']' => bitout;
fsm
    -> 'fsm' fsm←state '.' => fsmnode;
fsm←state
    -> state ← ind←part states ;
state←ind←part  ->  '[' state←specifiers ']'  => state←indnode;
states
    -> states   state ;
    -> state ;
state
    -> label   '[' state←specifiers ']' => statenode ;
state←specifiers
    -> state←spec ';'   state←specifiers ;
    ->   state←spec   ;
state←spec
    -> 'if or←cond '=>' action => condnode ;
    -> action ;
or←cond -> cond 'or' or←cond => ornode;
              -> cond;
cond   -> cond 'and' invocation => andnode ;
    -> invocation ;
invocation
    -> '<id>' parms  => invocationnode;
parms -> ;
              -> '(' id←or←const←list ')' ;
id←or←const←list
    -> constant ;
    -> constant ',' id←or←const←list ;
id←list -> '<id>';
              -> '<id>'  ',' id←list;
actions
    -> action ;
    -> action ';' actions ;
action
    -> '[' actions ']';
    -> 'assert' invocation                  => assertnode;
    -> invocation  =>  actionnode;
    -> 'next' '<id>' => gotonode;
    -> 'call' '<id>' => callnode;
    -> 'return'  =>  returnnode;
    ->   ;
label -> ;
              -> '<id>' ':'  => labelnode;
```

Appendix 2. Two More Examples

Appendix 2. 1 - 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:intcgcr): boolean;
dcfini tion
    gt & x ;
begin greater : = x>y end;
function equal (x,y:intcgcr): boolean;
definition eq & y;
begin eq : = x = y; end;
function ne(x,y:integcr): boolean;
definition not cqual (x,y);
begin ne : = not equal(x,y);
end;

fsm
[;]
one :     [ init ;
         a s s e r t  ne(y,0); .
          if cqual(x,0) = > next endstate ]
            [ call two ]
            [ next onc ]
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 ] .
```

Appendix 2.2 Traffic Light Example from Mead and Conway

```
program  traffic(input,output);
const short = 2;.
     long = 4;
type colortype = (green,yellow,red);
     signaltype = O..l;
```

```
var    timc: in tegcr;
       hl,fl: colortypc;
       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
       (grccn): hl = 0 ;
       (yellow): hl = 1 ;
       (red): hl = 2 ;
begin h  l : = color end;
procedure farmligh t(color: colortype);
definition
       (green): fl = 0 ;
       (yellow): fl = 1 ;
       (red): fl = 2 ;
begin    fl : = color end;
procedure starttimer;
dcfinition 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 timcout(lcngth: intcger) :boolcan;
definition
       (long): tl ;
       (short): ts ;
begin     timcout : =  (time > = lcngth) end;
function nottimcout(lcngth: intcger) :boolcan;
```

```
          definition
               (long): not tl ;
               (short): not ts ;
          begin    nottimeout : =  not timcou t(leng th)    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 ]
       ].
```