Data-Flow Assembly Language

Moez Ayed and Jean-Luc Gaudiot

CENG 94-38

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

May 5, 1994

Data-Flow Assembly Language[1]

Moez Ayed
Electrical Engineering Systems
University of Southern California
Los Angeles, California 90089-0781
ayed@priam.usc.edu
(213) 743-0403

Jean-Luc Gaudiot
Electrical Engineering Systems
University of Southern California
Los Angeles, California 90089-0781
gaudiot@paris.usc.edu
(213) 743-0249

May 5, 1994

[1]This work was supported in part by the U.S. Department of Energy, Office of Energy Research under Grant No. DE-FG03-87ER25043.

1 Informal Description of the Language

A data-flow graph is coded in an assembly like language that we call DFA (Data-Flow Assembly). A data-flow program consists of a collection of functions. Each function is a data-flow subgraph which is independent from the other functions and in which actors could be micro or macro. It has a name and a code block number. One of the functions is called MAIN and has no input or output arguments. The outer-most function calls (zero or more) are done in the body of function MAIN and the activation of all other functions is initiated during the execution of this function. A function in a data-flow program is like a function in a high level language. Each function except MAIN has one or more input arguments and *one* output argument which are names of ports belonging to the same function. All functions except MAIN can be called from any other function. For the sake of simplicity, we don't have any nested functions. In other words all functions are at the outermost level. Thus the structure of a data-flow program is similar the one of the C programming language. Input and output ports are identified by names. Within each function there is a one to one mapping between arcs of the graph and identifiers (variable names). There is the notion of scope of port names (different functions could have the same port names). Each actor has its own mapping function represented by an integer.

1.1 General format

- Actor specification:

  A actor-name $(x_0 \ldots x_n) \rightarrow y_0 \ldots y_m$: m=i :
  assembly-like micro-code inside the actor

  $x_0 \ldots x_n$: input ports
  $y_0 \ldots y_m$: output ports
  i = integer representing mapping function of actor.

- Function call specification:

  F f $(x_0 \ldots x_n) \rightarrow y$

  $x_0 \ldots x_n$: actual input parameters
  y : actual output parameter

- Initial input token on arc x :

  x = type token-value ;

- Function specification :

  DEFINE f (IN $x_0 \ldots x_n$ OUT y)
  CONST
  initial input tokens
  BEGIN
  actors and function calls specification
  END

- Header for function MAIN : DEFINE MAIN () .
- The collection of all function specifications constitute the data-flow program.
- The order of function specifications is not important.

The complete data-flow program written in DFA has the following format:

```
DEFINE function_1 (IN formal input parameters
                   OUT formal output parameter)

   CONST  % initial input tokens
     x1= type value;
     x2= type value;
     .
     .
     .

   BEGIN   % body of block

     A actor1 (input port names) -> output port names : mapping_func_number :

       microinstruction-1 ;
       microinstruction-2 ;
```

```
        .
        .
        .
        microinstruction-N ;

      A actor2 ...
       .
       .
       .

      A actorj ...

     % function call

      F function_k (actual input parameters) ->
                    actual output parameter

     % end of function call
     % function_k could be defined before or after function_1

      A actorj' ...

   END

DEFINE function_2 ....

.
.
.

DEFINE function_i .....

.
.
.
```

1.1.1 Remark about function calls

A function call inside a function is equivalent to having a virtual macro-actor whose body is the body of the called function, whose input ports are the actual input parameters (virtual input ports) and whose output ports are the actual output parameters (virtual output ports). When the call is made, the tokens carried by the virtual input ports are tranfered to the formal input paramaters of the called function. The corresponding result tokens which are generated at the formal output parameters of the called function are transfered to the virtual output ports.

Another thing that is worthwhile noting is that if a function is to be called, it has to have some formal input parameters. This is true because in data-flow all actors are triggered by the arrival of data. Therefore, to activate a function using a function call, we have to input data tokens to the function, and this is done through the actual input parameters.

Finally, we assume that THE FORMAL INPUT PARAMETERS AND THE ACTUAL OUTPUT PARAMETER GO TO *one* INPUT PORT OF *one* ACTOR ONLY, AND THE OUTPUT OF A FUNCTION CALL (the actual output parameter) CANNOT BE AN INPUT TO ANOTHER FUNCTION CALL (an actual input parameter).

1.1.2 Remark about actors

By convention, the input and output port names are listed from left to right starting from port 0. For example, if $(x_1 x_2 ... x_n)$ is the list of input port names of some actor, then x_1 is the port name of input port 0, x_2 is the port name of input port 1, ... , x_n is the port name of input port $(n-1)$.

1.2 Instruction set inside a macro-actor

Inside a macro-actor, we have an assembly like format for the micro-instructions. We have 500 registers to store the data. The registers are R0[00], ... , R0[99], R1[00], ... , R1[99], ... , R4[00], ... , R4[99]. The general notation of a register is Ri[mn] where i has values from 0 to 4 and m and n have values from 0 to 9.

In What follows, Ri denotes one of the registers listed above. Note that this notation will sometimes become ambiguous. In some cases we will use the notation Ri[l], in which case Ri denotes one of the arrays R0,R1,R2,R3 or R4.

1.2.1 Arithmetic instructions

```
ADD Ri,Rj,Rk      Ri <- Rj + Rk.
SUB Ri,Rj,Rk      Ri <- Rj - Rk.
MUL Ri,Rj,Rk      Ri <- Rj * Rk.
DIV Ri,Rj,Rk      Ri <- Rj / Rk.
INC Ri            Ri <- Ri + 1.
DEC Ri            Ri <- Ri - 1.
MOD Ri,Rj,Rk      Ri <- Rj mod Rk.
FLR Ri,Rj         Ri <- floor of Rj.
CLG Ri,Rj         Ri <- ceiling of Rj.
TRC Ri,Rj         Ri <- floating point number
                        stored in Rj truncated
                        to an integer.
CMP Ri,Rj,Rk      Ri <-  1   if Rj > Rk.
                  Ri <-  0   if Rj = Rk.
                  Ri <- -1   if Rj < Rk.
NOP               no operation.
```

1.2.2 Logic instructions

```
AND Ri,Rj,Rk      Ri <- Rj AND Rk.
OR  Ri,Rj,Rk      Ri <- Rj OR Rk.
XOR Ri,Rj,Rk      Ri <- Rj XOR Rk  (exclusive OR).
NOT Ri,Rj         Ri <- NOT Rj.
```

1.2.3 Load and exchange instructions

```
LD Ri,immediate          Ri <- immediate operand.
MOV Ri,Rj                Ri <- Rj.
MOVB Ri[l1],Rj[l2],Rk    Ri[l1] <- Rj[l2]
                         Ri[l1+1] <- Rj[l2+1]
                         ...
                         Ri[l1+c-1] < Rj[l2+c-1]
                         Where c is the content
                         of register Rk,
                         this is called block move.
```

1.2.4 Flow of control instructions

```
JMP label                PC <- label,
                         Unconditionel jump.
CBR condition,Ri,label   If condition=true
                         then PC <- label,
                         condition can be
                         one of the following:
                         EQ0,NE0,GT0,LT0,GE0,LE0.
                         The contents of Ri are
                         tested for the condition.
```

1.2.5 Input/output

Input:
Each macro-actor has a maximum of 5 input ports and 5 output ports. When the macro-actor fires, the data at input port i is stored in array Ri. If the data token is a single value, then it is stored in Ri[0]. If it is a vector $< v0, v1, ..., vn >$ then vj is stored in Ri[j], where $0 <= j <= n$ and $1 <= n <= 99$.

Output:

```
OUTS  (k),Ri             output port k <- Ri
```

```
OUTV (k),Ri[l],Rj       output port k <- vector v,
                        v= <Ri[l],Ri[l+1],...,Ri[l+c-1]>
                        where c is the content of Rj,
                        and c is greater than 1.
```

1.2.6 Vector operations

```
INIVEC      initialize for vector gathering.
ACCVEC      accumulate into a vector token.
```

1.2.7 I-structure operations

```
CRE (i)       create an array with dimension i.
APP (i)       append one element to an array
              with dimension i.
APX (i)       append a set of elements to an
              array with dimension i.
SEL (i)       select an element from an array
              with dimension i.
DEL (i)       delete an array with dimension i.
```

1.2.8 Tag manipulation instructions

```
GST   generate stream tokens with iteration tags.
DML   multiply iteration tag with specified operand.
DAD   add iteration tag with specified operand.
TTG   retrieve tag.
RTG   read tag.
WTG   write tag.
ATG   add tag with current tag.
DDD   D actor.
IDD   inverse D actor.
```

```
LLL   L actor.
ILL   inverse L actor.
```

1.2.9 Other useful instructions

```
TRU             true gate.
FAL             false gate.
SWI             switch gate.
MRG             merge gate.
IDN             identity token generation.
TPR             token print (for debugging purpuse).
CON x,y         constant token generation,
                if y= -1 then the output is x,
                otherwise the output is the
                vector [x,y].
```

1.2.10 Exiting

```
EXT       exit macro-actor.
```

1.2.11 Adressing modes

Both direct and indirect adressing modes are available.

```
Direct:      Ri[j]    denotes the contents of
                      register Ri[j].
Indirect:    (Ri[j])  denotes the contents of
                      register Rk[k1k2],where kk1k2 is
                      the contents of register Ri[j].
                      k, k1 and k2 are digits such that
                      0 <= k <= 4, 0 <= k1,k2 <= 9.
```

2 Lexical Elements Description

```
Reserved tokens:
  DEFINE MAIN IN OUT CONST CHAR INT REAL BEGIN END F A
  DIV MOD ADD SUB MUL INC DEC FLR CLG TRC CMP NOP
  AND OR XOR NOT LD MOV MOVB JMP CBR OUTS OUTV INIVEC
  ACCVEC CRE APP APX SEL DEL GST DML DAD TTG RTG WTG
  ATG DDD IDD LLL ILL TRU FAL SWI MRG IDN TPR CON EXT
  EQ0 NE0 GT0 LT0 GE0 LE0

Literal tokens (strings used exacly as declared):
  ( ) = ; -> : ,

The rest of the tokens are generated by the following
regular definition:

letter :        a|b|c|...|z|A|B|C|...|Z
digit :         0|1|2|...|9
pos_digit :     1|2|3|...|9
character :     letter|digit|_
integer :       ("+"|-)?(0|(pos_digit digit*))
int_part  :     0|(pos_digit digit*)
frac_part :     digit+
real :          ("+"|-)? int_part? "." frac_part
index1 :        0|1|2|3|4
index2 :        digit digit
int_vec :       [ integer (, integer)* ]
real_vec :      [ real (, real)* ]
char_vec :      [' character ' (, ' character ')* ]
register :      R index1 [ index2 ]

MAP_FUNC :      m = integer
ID :            letter character*
SCALER :        integer|real|' character '
VECTOR :        int_vec|real_vec|char_vec
REG :           register|"(" register ")"
```

```
POS_INT :       "(" "+"? (0|pos_digit digit*) ")"

Notational convention:
()        used for grouping terms together
|         alteration symbol
space     used for catenation of terms
?         preceding term is optional
*         0 or more occurences of the preceding term
+         1 or more occurences of the preceding term
" "       the symbol enclosed between quotes is used exacly as written
          e.g. "+" means the symbol + and not the positive closure symbol
```

3 Syntax of the Language

The context free grammar of the language is defined as follows:

```
program : function                          /* Start symbol  */
        | program function

function : header const_decl body

header : DEFINE ID form_para
       | DEFINE MAIN ( )

form_para : ( form_in_para )
          | ( form_in_para form_out_para )

form_in_para : IN parameters

form_out_para : OUT ID

parameters : ID
           | parameters ID
```

```
const_decl :
            | CONST assignments

assignments : assignment
             | assignments assignment

assignment : ID = type value ;

value : SCALER
       | VECTOR

type : CHAR
      | INT
      | REAL

body : BEGIN statements END

statements : statement
            | statements statement

statement : actor
           | call

call : F ID act_para

act_para : ( act_in_para )
          | ( act_in_para ) -> act_out_para

act_in_para : parameters

act_out_para : ID

actor : act_in act_out micro_ins_list

act_in : A ID ( inports )

act_out : outports : MAP_FUNC :
```

```
inports : parameters

outports :
          | -> parameters

micro_ins_list : micro_instr
               | micro_ins_list micro_instr

micro_instr : micro ;

micro : opcode operands
       | label opcode operands

opcode : DIV
        | MOD
        | ADD
        | SUB
        | MUL
        | INC
        | DEC
        | FLR
        | CLG
        | TRC
        | CMP
        | NOP
        | AND
        | OR
        | XOR
        | NOT
        | LD
        | MOV
        | MOVB
        | JMP
        | CBR
        | OUTS
        | OUTV
```

```
        | INIVEC
        | ACCVEC
        | CRE
        | APP
        | APX
        | SEL
        | DEL
        | GST
        | DML
        | DAD
        | TTG
        | RTG
        | WTG
        | ATG
        | DDD
        | IDD
        | LLL
        | ILL
        | TRU
        | FAL
        | SWI
        | MRG
        | IDN
        | TPR
        | CON
        | EXT

operands :
          | oper
          | oper1 , oper2
          | oper1 , oper2 , oper3

oper : REG
      | label
      | POS_INT

label : ID
```

```
oper1 : REG
      | POS_INT
      | cond
      | SCALER

cond : EQ0
     | NE0
     | GT0
     | LT0
     | GE0
     | LE0

oper2 : REG
      | SCALER

oper3 : REG
      | label
```

4 Example

Consider the data-flow program graph shown in figures 1 to 3. The same program written in DFA is given below.

```
DEFINE MAIN ()

  CONST
    x1= INT 10;
    x2= INT 3;
    x3= INT 3;
    x4= INT 3;
    x5= INT [0,1,2,0,1,2,0,1,2];  % vector token
    x7= INT 1;
    x8= INT 3;
```

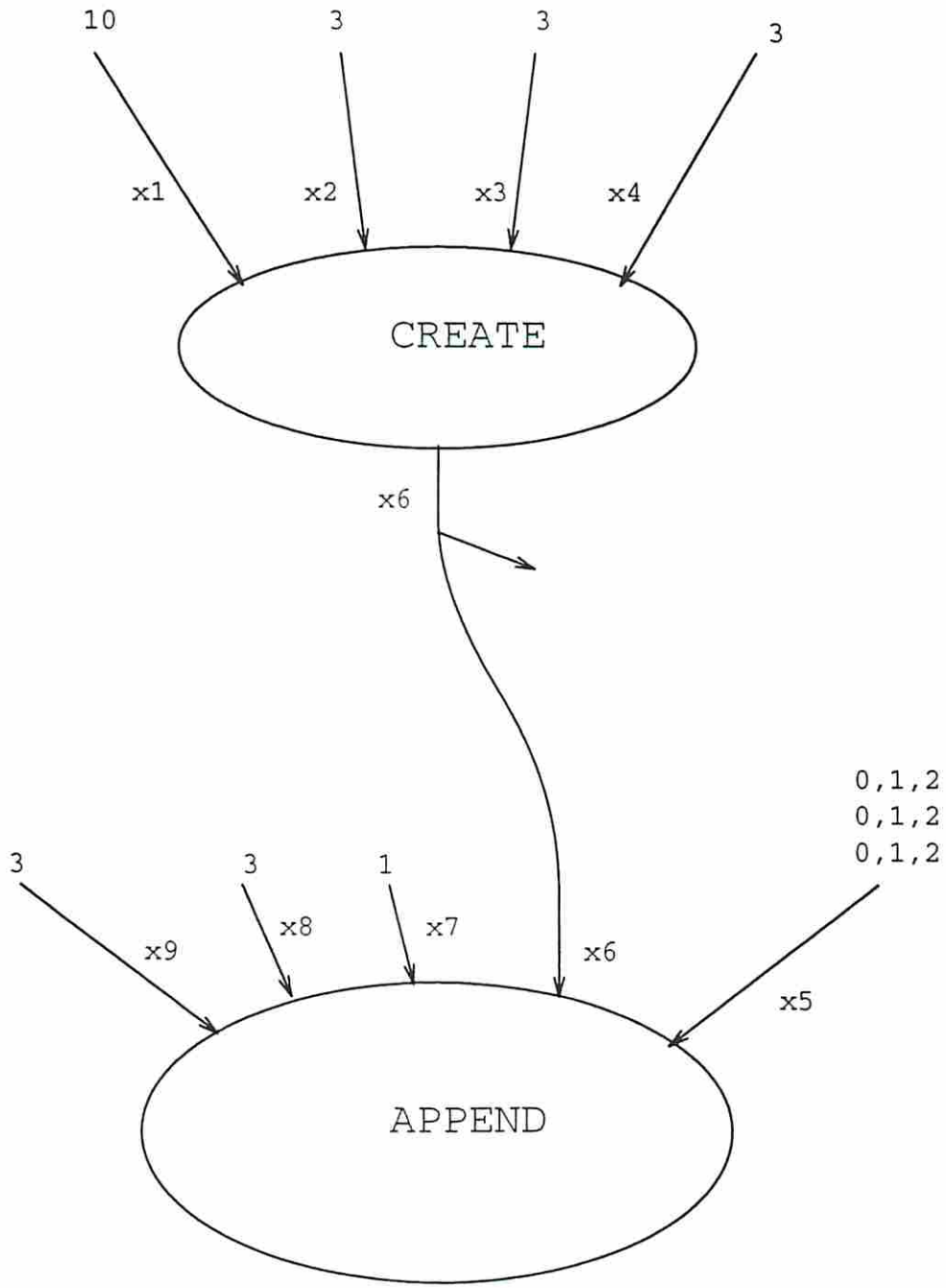

Figure 1: Function MAIN

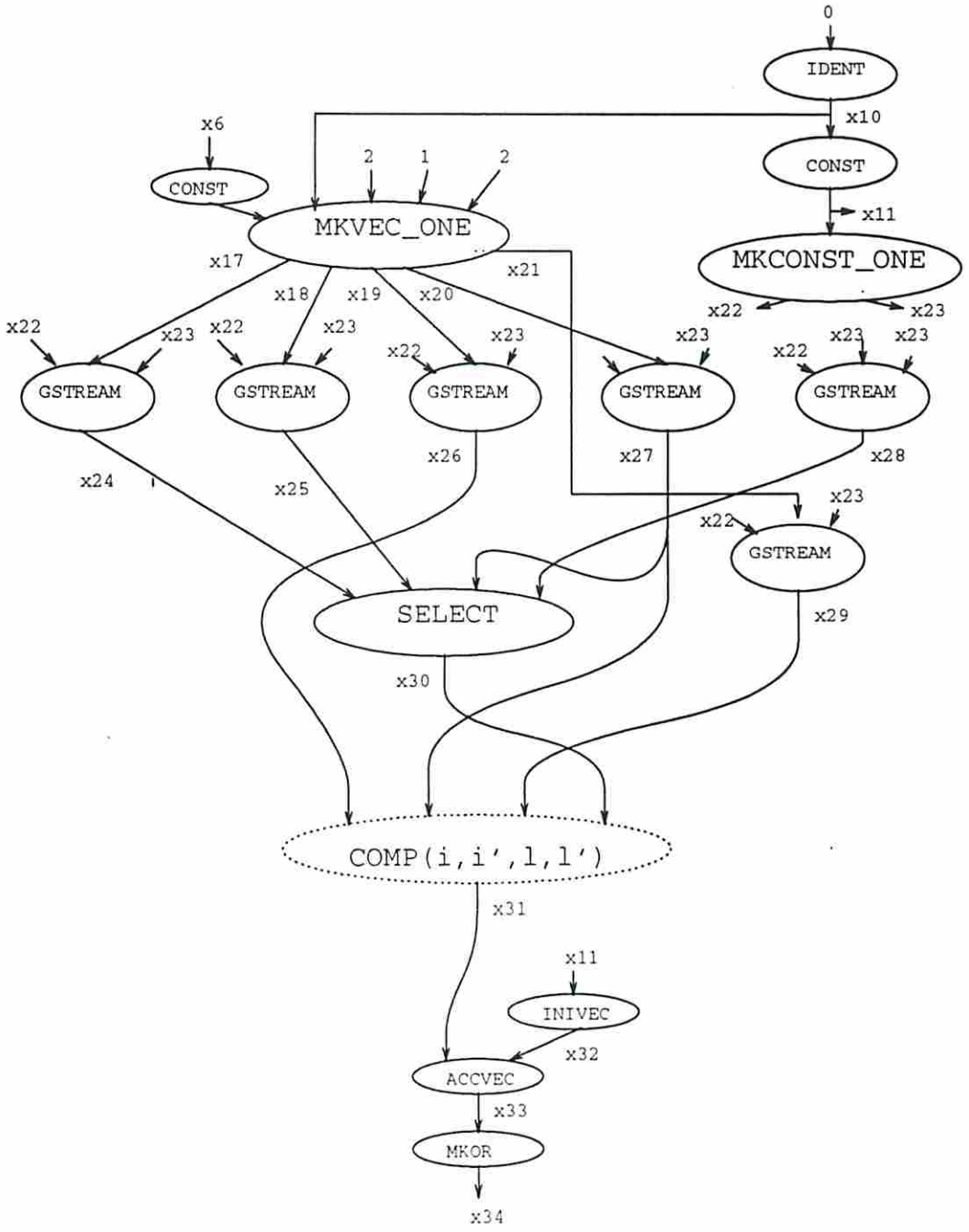

Figure 2: Following function MAIN

Code block c=10.
Ports are numbered from left
to right starting from 0.

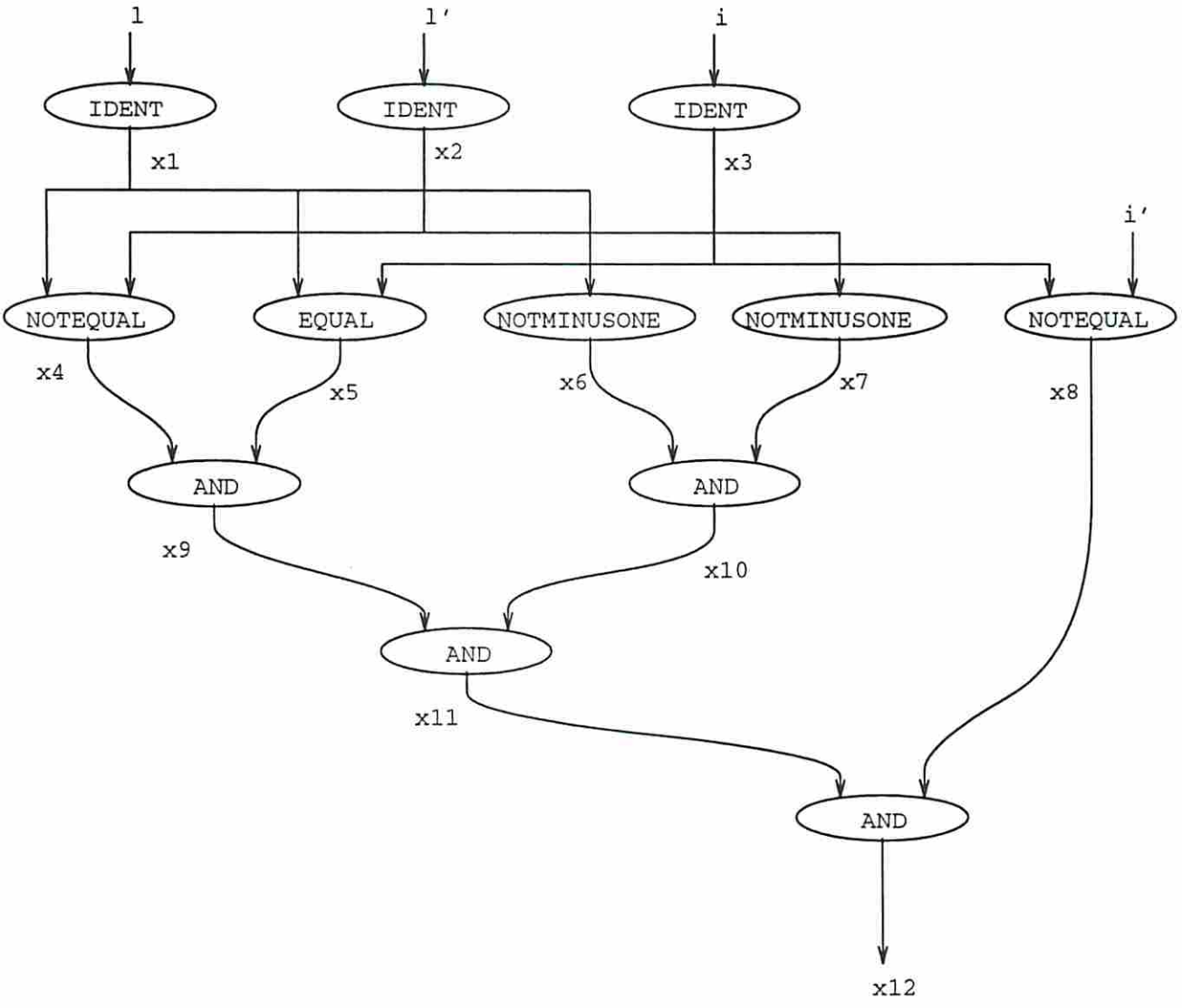

Figure 3: Function COMP

```
  x9= INT 3;
  x15= INT 0;
  x12= INT 2;
  x13= INT 1;
  x14= INT 2;

BEGIN

  A CREATE (x1 x2 x3 x4) -> x6 : m=0 :

      CRE (3) ;

  A APPEND (x5 x6 x7 x8 x9) : m=1 :
     APP (3) ;

  A IDENT (x15) -> x10 : m=2 :
     IDN ;

  A CONST (x10) -> x11 : m=3 :
     CON 3,-1 ;

  A CONST (x6) -> x16 : m=4 :
     CON 10,-1 ;

  A MKCONST_ONE (x11) -> x22 x23 : m=5 :

      MOV R1[01],R0[00];
      DEC R1[01];
      LD R1[00],0;
      LD R2[00],0;
      LD R2[01],1;
      LD R3[00],2;
      OUTV (0),R1[00],R3[00];
      OUTV (1),R2[00],R3[00];
      EXT ;

  A MKVEC_ONE (x16 x10 x14 x13 x12) ->
```

```
                x17 x18 x19 x20 x21    : m=6 :

   LD R0[01],0;
   LD R3[99],2;
   OUTV (0),R0[00],R3[99];
   LD R1[01],0;
   OUTV (1),R1[00],R3[99];
   LD R2[01],0;
   OUTV (2),R2[00],R3[99];
   LD R3[01],0;
   OUTV (3),R3[00],R3[99];
   LD R4[01],0;
   OUTV (4),R4[00],R3[99];
   EXT ;

A GSTREAM (x22 x17 x23) -> x24 : m=7 :

   GST;

A GSTREAM (x22 x18 x23) -> x25 : m=8 :

   GST;

A GSTREAM (x22 x19 x23) -> x26 : m=9 :

   GST;

A GSTREAM (x22 x20 x23) -> x27 : m=10 :

   GST;

A GSTREAM (x22 x23 x23) -> x28 : m=11 :

   GST;

A GSTREAM (x22 x21 x23) -> x29 : m=12 :
```

```
      GST;

   A SELECT (x24 x25 x27 x28) -> x30 : m=13 :

      SEL (3);

   F COMP (x26 x27 x29 x30) -> x31

   A INIVEC (x11) -> x32 :  m=14 :
      INIVEC ;

   A ACCVEC (x32 x31) -> x33 : m=15 :
      ACCVEC ;

   A MKOR (x33) -> x34 : m=16 :
      LD R2[00],0;
      LD R2[01],1;
      LD R3[00],1;
      LD R1[00],0;
   l0 ADD R1[00],R1[00],(R3[00]);
      DEC R0[00];
      INC R3[00];
      CBR EQ0,R0[00],l1;
      JMP l0;
   l1 CBR EQ0,R1[00],l2;
      OUTS (0),R2[01];
      EXT;
   l2 OUTS (0),R2[00];
      EXT ;

  END

DEFINE COMP (IN i i' l l' OUT x12)


  BEGIN
```

```
A IDENT (l) -> x1 : m=17 :
   IDN ;

A IDENT (l') -> x2 : m=18 :
   IDN ;

A IDENT (i) -> x3 : m=19 :
   IDN ;

A NOTEQUAL (x1 x2) -> x4 : m=20 :
   LD R3[00],0;
   LD R3[01],1;
   SUB R2[00], R1[00], R0[00];
   CBR NE0, R2[00], 1;
   OUTS (0),R3[00];
   EXT;
 1 OUTS (0),R3[01];
   EXT ;

A EQUAL (x1 x3) -> x5 : m=21 :
   LD R3[00],0;
   LD R3[01],1;
   SUB R2[00],R1[00],R0[00];
   CBR EQ0,R2[00],1;
   OUTS (0),R3[00];
   EXT;
 1 OUTS (0),R3[01];
   EXT ;

A NOTMINUSONE (x1) -> x6 : m=22 :
   LD R3[00],0 ;
   LD R3[01],1;
   INC R0[00];
   CBR NE0,R0[00],1;
   OUTS (0),R3[00];
   EXT;
 1 OUTS (0),R3[01];
```

```
   EXT ;

A NOTMINUSONE (x2) -> x7 : m=23 :
   LD R3[00],0 ;
   LD R3[01],1;
   INC R0[00];
   CBR NE0,R0[00],1;
   OUTS (0),R3[00];
   EXT;
 1 OUTS (0),R3[01];
   EXT ;


A NOTEQUAL (x3 i') -> x8 : m=24 :
   LD R3[00],0;
   LD R3[01],1;
   SUB R2[00],R1[00],R0[00];
   CBR NE0,R2[00],1;
   OUTS (0),R3[00];
   EXT;
 1 OUTS (0),R3[01];
   EXT ;

A AND (x4 x5) -> x9 : m=25 :
   MUL R0[00],R0[00],R1[00];
   OUTS (0),R0[00];
   EXT ;

A AND (x6 x7) -> x10 : m=26 :
   MUL R0[00],R0[00],R1[00];
   OUTS (0),R0[00];
   EXT ;

A AND (x9 x10) -> x11 : m=27 :
   MUL R0[00],R0[00],R1[00];
   OUTS (0),R0[00];
   EXT ;
```

```
  A AND (x11 x8) -> x12 : m=28 :
     MUL R0[00],R0[00],R1[00];
     OUTS (0),R0[00];
     EXT ;

END
```